UniCloud BDP 大数据组件

用户手册

紫光云技术有限公司
www.unicloud.com

资料版本：5W100-20230223
产品版本：UniCloud BDP (E6105)

© 紫光云技术有限公司 2023 版权所有，保留一切权利。

未经本公司书面许可，任何单位和个人不得擅自摘抄、复制本书内容的部分或全部，并不得以任何形式传播。

对于本手册中出现的其它公司的商标、产品标识及商品名称，由各自权利人拥有。

由于产品版本升级或其他原因，本手册内容有可能变更。紫光云保留在没有任何通知或者提示的情况下对本手册的内容进行修改的权利。本手册仅作为使用指导，紫光云尽全力在本手册中提供准确的信息，但是紫光云并不确保手册内容完全没有错误，本手册中的所有陈述、信息和建议也不构成任何明示或暗示的担保。

前 言

本手册主要介绍了 UniCloud BDP 中 HDFS、YARN&MapReduce、Spark、Flink、Storm、Hive、HBase、Phoenix、Solr、Elasticsearch、Kafka、Redis、DLH 的组件概述、组件架构、快速入门、使用指南、开发指南、最佳实践以及相关的常见问题解答等内容。

前言部分包含如下内容：

- 读者对象
- 本书约定
- 资料意见反馈

读者对象

本手册主要适用于如下工程师：

- 网络规划人员
- 现场技术支持与维护人员
- 负责网络配置和维护的网络管理员

本书约定

1. 图形界面格式约定

格 式	意 义
< >	带尖括号“< >”表示按钮名，如“点击<确定>按钮”。
[ ]	带方括号“[ ]”表示窗口名、菜单名和数据表，如“弹出[新建用户]窗口”。
/	多级菜单用“/”隔开。如[文件/新建/文件夹]多级菜单表示[文件]菜单下的[新建]子菜单下的[文件夹]菜单项。

2. 各类标志

本书还采用各种醒目标志来表示在操作过程中应该特别注意的地方，这些标志的意义如下：

注意	提醒操作中应注意的事项，不当的操作可能会导致数据丢失或者无法使用。
说明	对操作内容的描述进行必要的补充和说明。

3. 示例约定

由于设备型号不同、配置不同、版本升级等原因，可能造成本手册中的内容与用户使用的设备显示信息不一致。实际使用中请以设备显示的内容为准。

本手册中出现的 IP 地址仅作示例，并不代表设备上实际具有此 IP 地址，实际使用中请以设备上配置的 IP 地址为准。

资料意见反馈

如果您在使用过程中发现产品资料的任何问题，可以通过以下方式反馈：

E-mail：unicloud-ts@unicloud.com

感谢您的反馈，让我们做得更好！

概 览

HDFS ······6

YARN&MapReduce ······96

Spark ······151

Flink ······227

Storm ······401

Hive ······457

HBase ······554

Phoenix ······632

Solr ······675

Elasticsearch ······728

Kafka ······811

Redis ······866

DLH ······909

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.2 组件架构 1-1
1.2.1 HDFS 集群架构 1-1
1.2.2 HDFS HA 架构 1-2
1.3 特点和应用场景 1-4
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 数据目录检查 2-1
2.1.2 组件数据存放位置 2-2
2.1.3 查看组件的日志信息 2-2
2.2 运行状态监控 2-3
2.2.1 查看组件详情 2-3
2.2.2 组件检查 2-4
2.3 快速使用指导 2-5
2.3.1 非 kerberos 环境 2-6
2.3.2 Kerberos 环境 2-7
2.4 快速链接 2-9
2.4.1 配置组件快速链接 2-9
2.4.2 访问 HDFS 快速链接 2-9
3 使用指南 3-1
3.1 Shell 命令 3-1
3.2 Client 下载/安装/使用/卸载 3-4
3.2.1 下载 Client 安装包 3-4
3.2.2 安装 Client 3-6
3.2.3 访问组件 3-7
3.2.4 卸载 Client 客户端 3-7
3.3 权限访问控制 3-7
3.3.1 权限说明 3-8
3.3.2 权限使用操作示例 3-9
3.4 密钥管理 3-11
3.4.1 使用指导 3-12

3.4.2 HDFS配置密钥信息 3-13
3.4.3 密钥使用操作示例 3-14
3.5 HDFS集群扩容 3-17
3.5.1 使用场景 3-17
3.5.2 扩容前准备 3-17
3.5.3 扩容约束 3-18
3.5.4 扩容影响 3-18
3.5.5 扩容操作指导 3-18
3.5.6 DataNode扩容后数据均衡 3-19
3.5.7 扩容验证 3-19
3.6 HDFS集群缩容 3-19
3.6.1 使用场景 3-20
3.6.2 缩容前准备 3-20
3.6.3 缩容约束 3-20
3.6.4 缩容影响 3-21
3.6.5 缩容操作指导 3-21
3.6.6 缩容验证 3-26
3.7 租户管理 3-26
3.7.1 租户介绍 3-26
3.7.2 新增租户 3-26
3.7.3 租户使用操作示例 3-28
3.8 备份恢复 3-29
3.8.1 新建HDFS同步任务 3-30
3.8.2 源集群和目的集群配置互信 3-31
3.8.3 HDFS同步任务相关配置 3-33
3.8.4 HDFS加密区数据同步 3-34
3.9 数据重分布 3-35
3.9.1 数据均衡 3-35
3.9.2 磁盘均衡 3-38
3.10 HDFS分级存储 3-42
3.10.1 存储策略与存储类型关系 3-42
3.10.2 HDFS存储策略命令 3-43
3.10.3 存储类型设置 3-43
3.11 HDFS纠删码 3-44
3.11.1 HDFS纠删码命令 3-45
3.11.2 操作示例 3-45

4 开发指南 4-1
4.1 API 介绍 4-1
4.1.1 HDFS 常用接口 4-1
4.2 模块样例代码介绍 4-2
4.2.1 开发环境简介 4-2
4.2.2 配置并导入样例代码 4-2
4.2.3 HDFS 配置文件介绍 4-5
4.2.4 示例公共代码 4-5
4.2.5 创建目录 4-6
4.2.6 写文件 4-6
4.2.7 文件追加内容 4-7
4.2.8 读文件 4-7
4.2.9 删除文件 4-8
4.2.10 删除目录 4-8
4.2.11 设置存储策略 4-9
4.2.12 多线程任务 4-10
4.2.13 编译并运行程序 4-11
5 最佳实践 5-1
5.1 数据迁移 5-1
5.1.1 迁移操作示例 5-1
5.2 目录离线备份 5-4
5.3 全量离线备份 5-4
6 常见问题解答 6-1
6.1 调优 6-1
6.2 通用类 6-1

1 组件简介

1.1 组件概述

HDFS（Hadoop Distributed File System）是一种具有高容错、高可靠性的分布式文件系统，能够提供高吞吐量的数据访问，是分布式计算中数据存储和管理的基础。

HDFS 采用“一次写入，多次读取”的模式，可以提供高吞吐量的数据访问，支持文件系统数据的流式访问，适用于超大文件存储。

1.2 组件架构

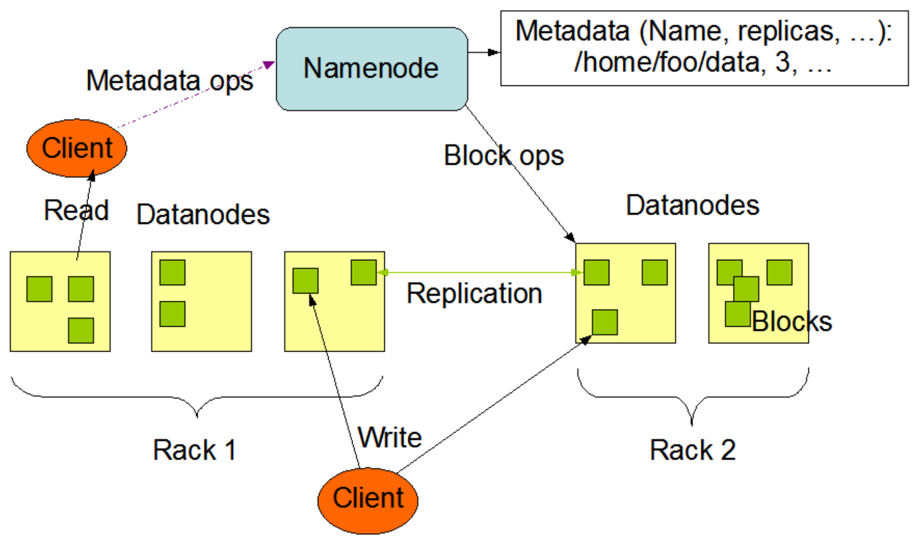
1.2.1 HDFS 集群架构

如图 1-1 所示，HDFS 采用主从（Master/Slave）体系结构，主要由 NameNode 和 DataNode 组成。其中：

- NameNode（主备配置）：NameNode 是一个中心服务器，维护整个文件系统的命名空间、文件/目录的元信息、文件的数据块索引以及处理客户端的读写请求。主要功能：对外提供服务、管理 HDFS 名称空间、管理数据块到具体的存储节点的映射、在其同意调度下进行数据块的创建、删除、复制。若 NameNode 失效则整个 HDFS 都失效了，所以要保证 NameNode 的可用性。
- DataNode（多个，可动态扩展）：DataNode 是 HDFS 的实际存储节点，负责管理它所在节点的存储；负责响应 HDFS 客户端的读写请求，执行 NameNode 下发的指令（例如创建数据块、删除数据块或复制数据块等指令）；DataNode 会定期向 NameNode 发送自身所存储的块(Block)信息，定期向 NameNode 做心跳汇报。
- Block：HDFS 将文件分割成一个或多个 Block，每个 Block 作为一个独立的单元保存在 DataNode 中。

  说明：元数据是指每个 Block 与其归属文件的对应关系。
- Client：包括命令行、应用程序和 Web 管理界面等，是用户和 HDFS 的交互手段，通过它用户可以与 NN、DN 进行通信。

图1-1 HDFS 体系架构图



1.2.2 HDFS HA 架构

HA 即为高可用（High Availability），用于解决 NameNode 单点故障的问题，该特性通过热备的方式为 Active NameNode 提供一个备用者，一旦 Active NameNode 出现故障，可以迅速切换至 Standby NameNode，从而不间断地对外提供服务。HDFS HA 架构如图 1-2 所示。其中：

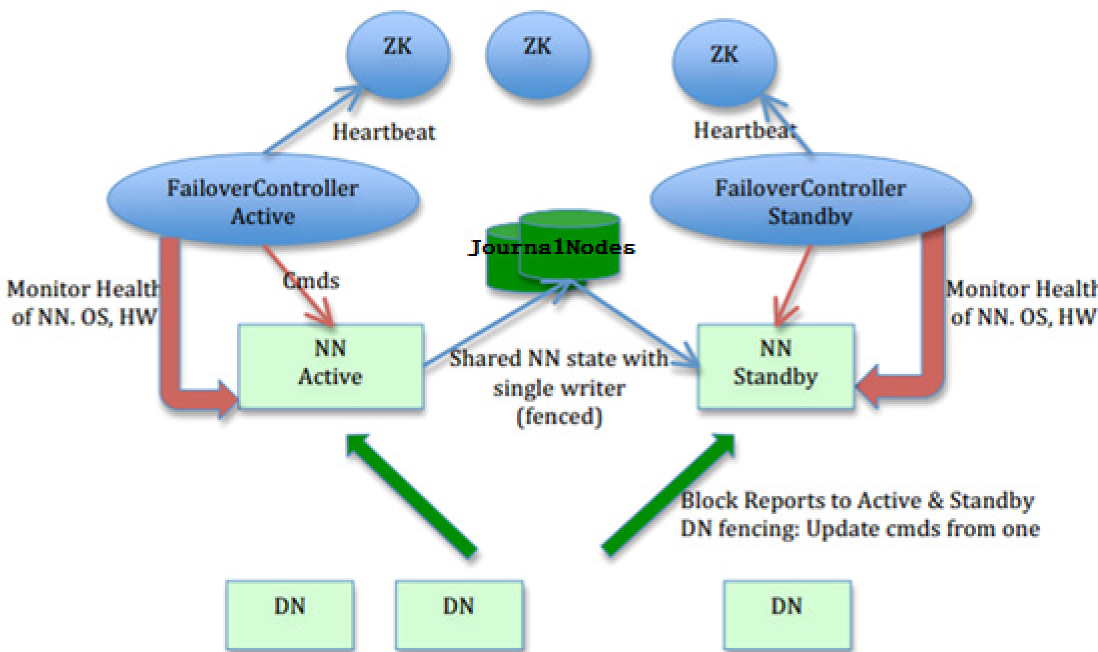
- ZooKeeperFailoverController（ZKFC）：用于监控和管理 NameNode 健康状态，对 NameNode 的主备切换进行总体控制。ZKFC 需要和 NameNode 提供一一对应的服务，即每个 NameNode 所在节点都要部署 ZKFC。ZKFC 会周期性的检查 NameNode 的健康状态，当发现 Active NameNode 故障时，会释放 Znode 锁，之后 Standby NameNode 节点所在的 ZKFC 会获取到 Znode 锁，并将 Standby NameNode 切换为 Active NameNode，从而不间断地对外提供服务，保证 HDFS 的高可靠性。
- ZooKeeper（ZK）：主要用来存储 HA 下的状态文件，提供主备选举支持。
- Standby NameNode：持续监测 JournalNodes 上操作日志记录的变化，当 Standby NameNode 发现操作日志记录发生改变的时候，也会将这些变化同步到自己的命名空间里。当 Active NameNode 发生故障时，Standby NameNode 切换到 Active 之前会确保其已经从 JournalNodes 上获取了所有的日志记录，保证故障转移时，主备 NameNode 的文件系统元数据是完全同步的。
- JournalNodes（JNS）：HA 模式下，NameNode 主备节点会通过 JournalNode 的独立进程进行相互通信，用于同步主备 NameNode 之间的元数据信息。

说明

为了提高系统承受故障的能力，要求运行奇数个 JournalNodes（3，5，7 等），至少要求 3 个 JournalNodes 守护进程。若有 N 个 JournalNodes 运行时，该系统最多可以容忍（N - 1）/ 2 个的 JournalNodes 失败。

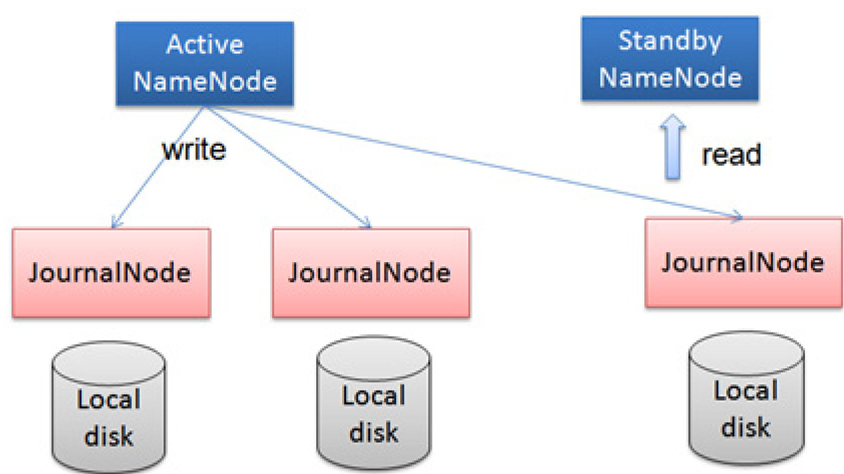
- 为了实现快速故障转移，DataNodes 会同时向 Active NameNode 和 Standby NameNode 发送 Block 的位置信息和心跳信息。

图1-2 HDFS HA 架构



HDFS HA 模式采用的是 QJM（Quorum Journal Manager）架构，如图 1-3 所示。主备 NameNode 会同时与一组称为 JournalNodes（JNS）的独立进程进行通信来保持元数据同步。当 Client 对 HDFS 命名空间进行了修改，如删除了某个文件，Active NameNode 会向超过半数的 JournalNodes 中写入此次操作的日志记录。

图1-3 QJM 架构



1.3 特点和应用场景

HDFS 具有以下特点：

- 高可靠性
- 流式数据访问
- 数据一致性模型
- 超大文件存储

HDFS 适用于海量数据存储、离线计算、对吞吐要求较高的大数据分析和机器学习的业务场景。

2 快速入门

2.1 组件安装

说明

- HDFS 通常和 YARN、MapReduce 一起使用，所以安装组件时，HDFS、MapReduce、YARN 必须同时安装，对外呈现名称为 Hadoop。
- HDFS 依赖 Zookeeper，安装 HDFS 时必须同时安装 Zookeeper 组件。
- 在 Hadoop 集群中，安装 HDFS 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- 部署 HDFS 时，根据实际生产环境的数据量，需要调整 DataNode 节点数量。

2.1.1 数据目录检查

根据现场磁盘分区方案和挂盘方案的不同，HDFS 安装完成后，必须对其数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表 2-1 所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
HDFS	是（配置项的参数值默认选择3个挂载路径）	dfs.namenode.name.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
	是（配置项的参数值默认使用全部挂载路径）	dfs.datanode.data.dir	
	未开启高可用时，需要检查该配置项（配置项的参数值默认使用全部挂载路径）	dfs.namenode.checkpoint.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
	开启高可用时，需要检查该配置项（配置项的参数值默认选择1个挂载路径）	dfs.journalnode.edits.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
YARN	是（配置项的参数值默认使用全部挂载路径）	yarn.nodemanager.local-dirs	此目录为数据目录，用于存放应用程序的运行依赖包等信息。检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置
		yarn.nodemanager.log-dirs	

组件	是否需要检查	被影响的配置项	如何解决
			为对应的数据目录
	是（配置项的参数值默认使用某一个挂载路径）	yarn.timeline-service.leveldb-state-store.path	此目录为数据目录，用于记录应用程序运行状态等信息。检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
Zookeeper	是（配置项的参数值默认使用某一个挂载路径）	dataDir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录

2.1.2 组件数据存放位置

表2-2 组件数据的存放位置

组件	存放位置	说明
ZooKeeper	配置项dataDir的值	存放ZooKeeper数据 【注意】修改当前配置将导致当前组件的数据丢失
HDFS-JournalNode	配置项dfs.journalnode.edits.dir的值	存放HDFS的JournalNode数据
HDFS-NameNode	配置项dfs.namenode.name.dir的值	存放HDFS的NameNode数据
HDFS-DataNode	配置项dfs.datanode.data.dir的值	• 推荐把每个物理磁盘挂载在/opt/disknn（nn 为 1 至 2 位的数字）上不同的挂载点 • 存放 HDFS 的数据

2.1.3 查看组件的日志信息

表2-3 组件日志路径说明

组件	日志路径
HDFS	/var/de_log/hadoop/user_hdfs
MapReduce2	HistoryServer日志路径：/var/de_log/hadoop-yarn/user_mapred/和/var/de_log/hadoop-mapreduce/mapred/
YARN	/var/de_log/hadoop-yarn/user_yarn 【说明】上述的日志是YARN本身的日志。另外： • 执行在 YARN 上的应用日志，可以通过 YARN 的 UI 界面查看，日志文件实际存储位置默认是在 HDFS 上，默认路径为/app-logs • 如果日志不聚合，可以配置 yarn.log-aggregation-enable 为 false • 内嵌的 HBase 日志路径为：

组件	日志路径
	/var/de_log/hadoop-yarn/embedded-yarn-ats-hbase，在 timeline 安装的节点上可看到（前提条件：配置项 use_external_hbase、is_hbase_system_service_launch 的值均为 false）
ZooKeeper	/var/de_log/zookeeper/user_{user.name}/，其中${user.name}是指执行任务的用户名

2.2 运行状态监控

2.2.1 查看组件详情

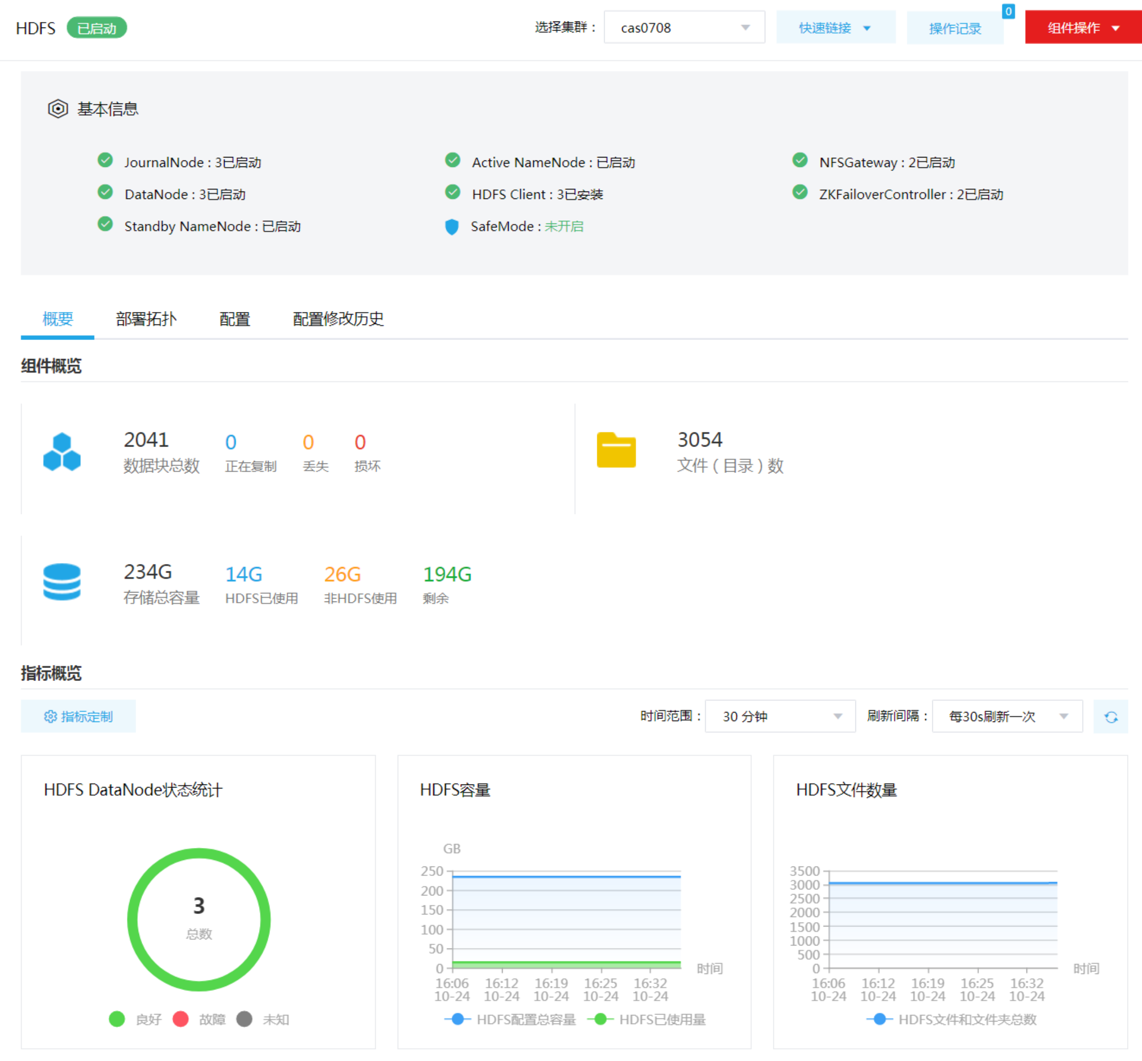
进入 HDFS 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、概要、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。
- 概要：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 组件详情



2.2.2 组件检查

执行 HDFS 组件检查时，会向 HDFS 上传测试文件并检查 HDFS 文件系统的 UI 页面响应，同时检测 HDFS 相关进程的运行状态，若 HDFS 组件检查成功则表示向 HDFS 上传文件和页面响应正常，且各进程运行正常。

集群在使用过程中，根据实际需要，可对 HDFS 执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：

- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 HDFS 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 HDFS 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态。如图 2-2 所示，表示该组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“HDFS Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情

2.3 快速使用指导

注意

根据大数据集群是否开启 Kerberos 认证，用户访问 HDFS 文件系统时的认证方式不同，详情请参见本章节内容。

HDFS 文件系统既可以通过集群用户访问，又可以通过组件超级用户访问。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
  - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。
  - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 HDFS 组件的 hdfs 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 kerberos 环境

说明

- 非 Kerberos 环境下，不需要用户做身份认证即可直接对 HDFS 执行管理操作。
- 本章节操作需要切换至具有操作 HDFS 文件系统权限的用户。

HDFS 安装完成后，连接集群内的节点（该节点要求安装了 NameNode、DataNode 或 HDFS Client）即可使用 ***hdfs*** 命令来操作 HDFS 文件系统。

1. 目录操作

- 查看目录结构

  hdfs dfs -ls /
- 创建目录

  hdfs dfs -mkdir /test
- 删除目录

  hdfs dfs -rm -r /test

2. 文件操作

- 上传文件到 HDFS
  - 创建 hdfs.txt 文件并编辑文件的内容为“hdfs put test”，使用如下命令：

    vi hdfs.txt
  - 上传文件到 HDFS 根目录，使用如下命令：

    hdfs dfs -put hdfs.txt /
- 查看文件内容

  hdfs dfs -cat /hdfs.txt
- 删除文件

  hdfs dfs -rm /hdfs.txt
- 移动文件

  hdfs dfs -mv /hdfs.txt /tmp

- 复制文件

  hdfs dfs -cp /tmp/hdfs.txt /

2.3.2 Kerberos 环境

说明

- Kerberos 环境下，若想访问 HDFS 文件系统并对 HDFS 执行管理操作，则必须首先进行用户身份认证，认证方式请参见2.3.2 1. Kerberos 环境下用户身份认证。
- 本章节操作需要切换至具有操作 HDFS 文件系统权限的用户。

1. Kerberos 环境下用户身份认证

如果大数据集群开启 Kerberos，若想操作 HDFS 文件系统，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

HDFS 文件系统还可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**

  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的/etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：

  chown user1 /etc/security/keytabs/user1.keytab

  b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：

  klist -k user1.keytab

  【说明】如图 2-4 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-4 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

su user1

kinit -kt user1.keytab user1@TENANTC.COM

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-5 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

（二）组件超级用户身份认证

HDFS 文件系统可以通过组件超级用户访问，比如 hdfs 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 hdfs 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 hdfs 的认证文件“hdfs.headless.keytab”。

【说明】在 HDFS Client 节点上，需要将 hdfs 的认证文件“hdfs.headless.keytab”上传至节点的/etc/security/keytabs/目录下进行认证。

(2) 使用 ***klist*** 命令查看 hdfs.headless.keytab 的 principal 名称，命令如下：

klist -k hdfs.headless.keytab

【说明】如图 2-6所示，红框内容即为 hdfs.headless.keytab 的 principal 名称。

图2-6 认证文件的 principal 名称

```
[root@hppnode1 keytabs]# klist -k hdfs.headless.keytab
Keytab name: FILE:hdfs.headless.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
```

(3) 切换至用户 hdfs，并执行身份验证的命令如下：

su hdfs

kinit -kt hdfs.headless.keytab hdfs-testshare@TESTSHARE.COM

【说明】其中：hdfs.headless.keytab 为 hdfs 的认证文件，hdfs-testshare@TESTSHARE.COM 为 hdfs.headless.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

2. 管理 HDFS 文件系统

当用户身份认证成功后，即可参见 2.3.1 非 kerberos 环境下的使用示例命令来操作 HDFS 文件系统。

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

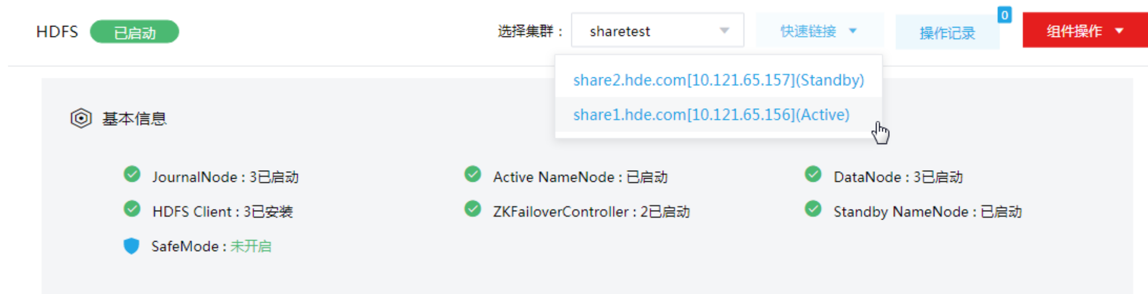
2.4.2 访问 HDFS 快速链接

HDFS 提供了文件系统管理页面（即 NameNode UI），支持在线查看文件系统相关详细信息。

(1) 如图 2-7 所示，在 HDFS 组件详情页面的右上角[快速链接]的下拉框中，可以获取 HDFS 文件系统的访问入口信息。

【说明】当集群开启高可用时，HDFS 同步开启 HA，此时有两个访问入口，推荐选择 active 状态的链接。

图2-7 HDFS 快速链接



(2) 根据集群是否开启 Kerberos，访问 HDFS 快速链接分为两种情况：

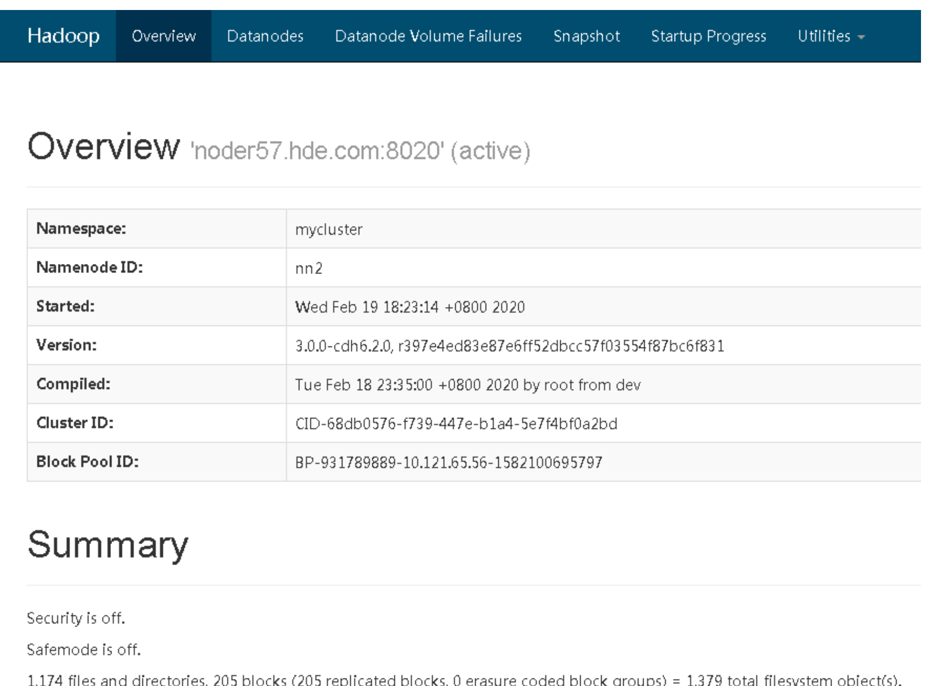
- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，可直接跳转访问对应的 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。

(3) 在 HDFS 文件系统管理页面，可查看以下信息：

- Overview 页面

  如图 2-8 所示，该页面用于展示集群的基本状态等信息。

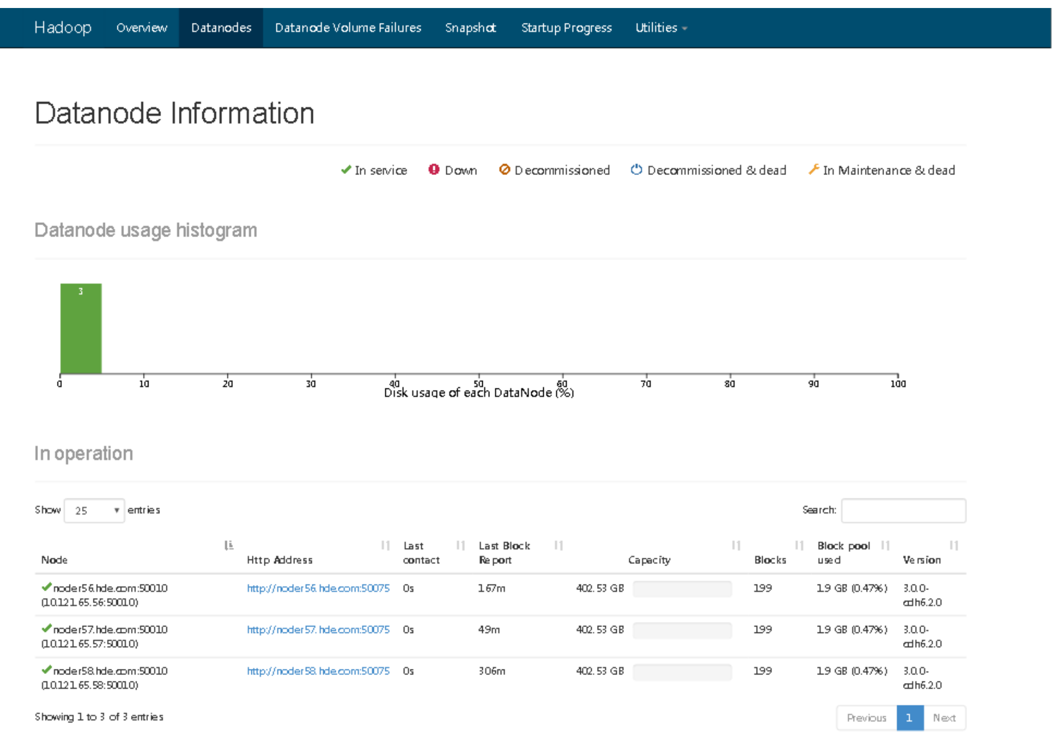
  图2-8 Overview 页面



- DataNodes 页面

选择[DataNodes]页签，进入 DataNodes 页面。如图 2-9 所示，该页面可以展示 DataNode 的基本信息，包括 DataNode 所在的节点、端口号、节点的状态、容量以及磁盘使用情况等。

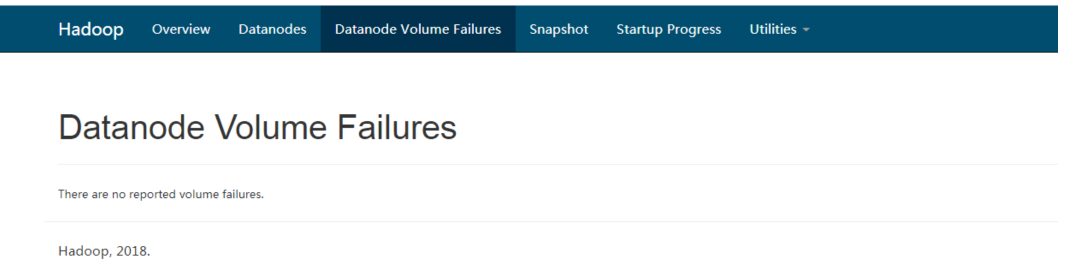
图2-9 DataNodes 页面



- Datanode Volume Failures 页面

  选择[Datanode Volume Failures]页签，进入 Datanode Volume Failures 页面。如图 2-10 所示，当 Datanode Volume 存在错误时，会在该页面上进行展示。

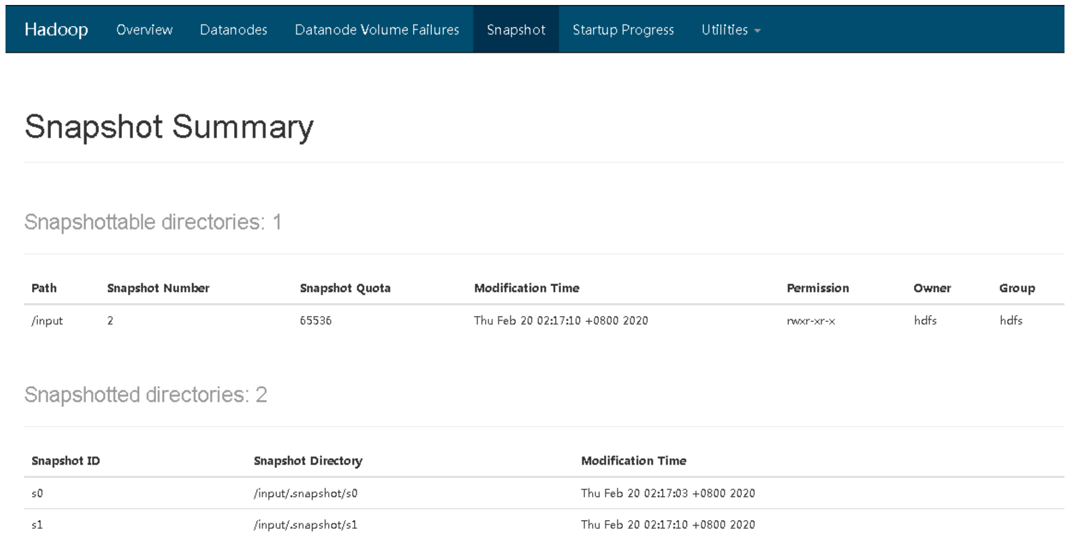
  图2-10 Datanode Volume Failures 页面



- Snapshot 页面

  选择[Snapshot]页签，进入 Snapshot 页面。该页面可以展示文件系统的快照信息。如图 2-11 所示，/input 目录包含两个快照 s0 和 s1，/input 目录下的文件快照存储在 /input/.snapshot 路径下。

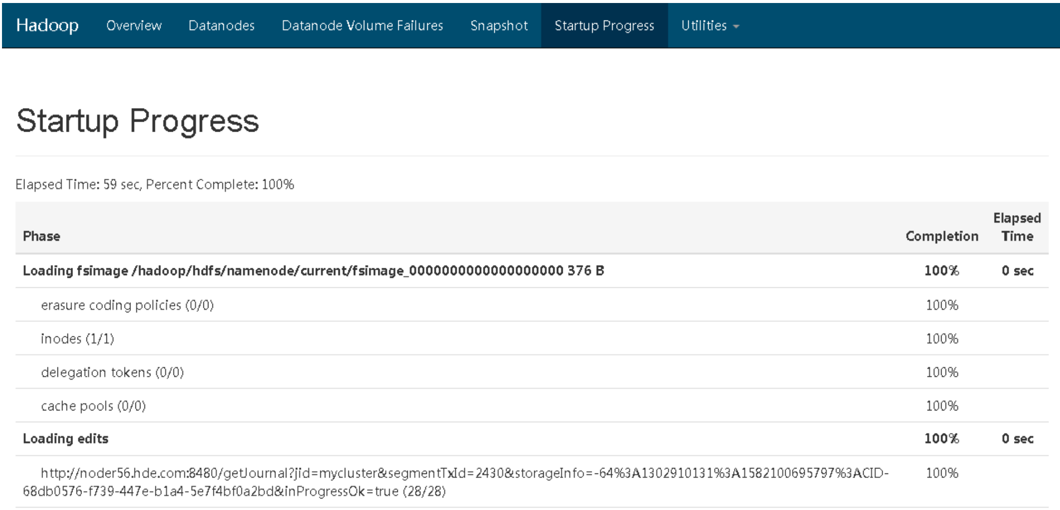
图2-11 Snapshot页面



- Startup Progress页面

  选择[Startup Progress]页签，进入Startup Progress页面。如图2-12所示，该页面可以展示NameNode的启动过程等信息。

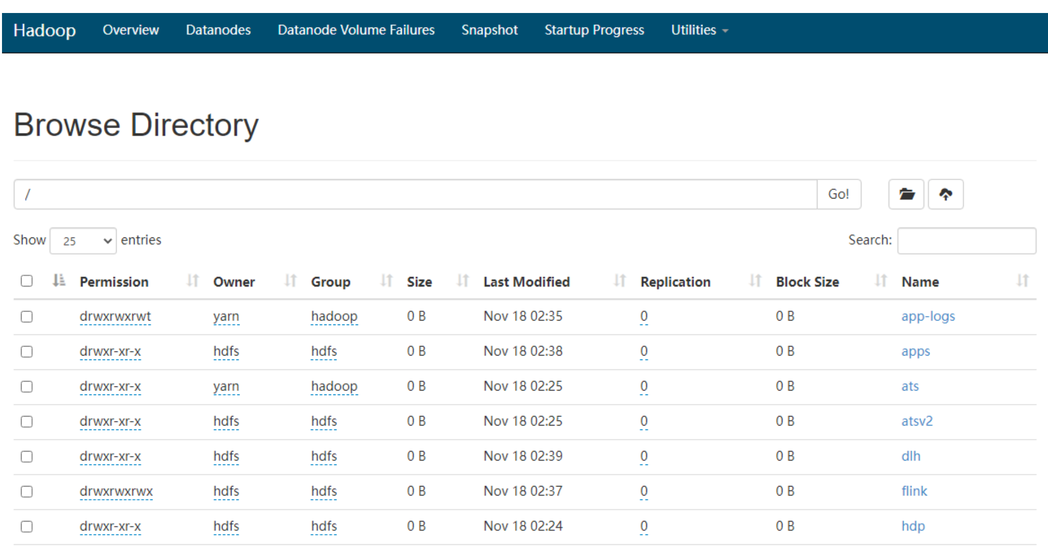
  图2-12 Startup Progress页面



- Browse Directory页面

  在Utilities下拉框中点击<Browse the file system>即可进入Browse Directory页面。如图2-13所示，该页面详细展示了HDFS分布式文件系统上的目录结构，并可在该页面上执行文件上传、目录创建等操作。

图2-13 Browse Directory 页面



- 日志展示页面

  在 Utilities 下拉框中点击<Logs>即可进入日志展示页面。如图 2-14 该页面展示了 HDFS 组件相关进程的日志，可以直接点击日志文件进行查看。

  【说明】当集群开启了 Kerberos 时，该页面无法展示，此时只能去后台 /var/log/hadoop/user_hdfs 路径下查看。

图2-14 日志展示页面

Directory: /logs/

SecurityAuth.audit	102350 bytes	Feb 20, 2020 2:28:29 AM
SecurityAuth.audit-2020-02-19-1.gz	15798 bytes	Feb 20, 2020 12:00:01 AM
datanode-metrics-noder57.hde.com.log	0 bytes	Feb 19, 2020 4:24:41 PM
hadoop-hdfs-datanode-noder57.hde.com.log	21160 bytes	Feb 20, 2020 2:25:40 AM
hadoop-hdfs-datanode-noder57.hde.com.log-2020-02-19-1.gz	47389 bytes	Feb 20, 2020 12:22:56 AM
hadoop-hdfs-datanode-noder57.hde.com.out	1174 bytes	Feb 19, 2020 6:22:44 PM
hadoop-hdfs-datanode-noder57.hde.com.out.1	1174 bytes	Feb 19, 2020 5:50:40 PM
hadoop-hdfs-datanode-noder57.hde.com.out.2	1174 bytes	Feb 19, 2020 4:24:42 PM
hadoop-hdfs-journalnode-noder57.hde.com.log	49833 bytes	Feb 20, 2020 2:28:32 AM
hadoop-hdfs-journalnode-noder57.hde.com.log-2020-02-19-1.gz	28384 bytes	Feb 20, 2020 12:00:28 AM
hadoop-hdfs-journalnode-noder57.hde.com.out	1174 bytes	Feb 19, 2020 6:22:55 PM
hadoop-hdfs-journalnode-noder57.hde.com.out.1	1174 bytes	Feb 19, 2020 5:50:51 PM
hadoop-hdfs-journalnode-noder57.hde.com.out.2	1174 bytes	Feb 19, 2020 4:24:46 PM
hadoop-hdfs-namenode-noder57.hde.com.log	113128 bytes	Feb 20, 2020 2:28:32 AM

3 使用指南

3.1 Shell命令

Hadoop 包含丰富的 Shell 的命令，可用于 HDFS Client 与 HDFS 文件系统之间的交互。

说明

以下命令均需要切换至具有操作 HDFS 文件系统权限的用户，例如可以执行 ***su hdfs*** 命令切换至 hdfs 用户下进行相关操作。

1. hdfs 命令

表3-1 文件系统操作命令

命令	格式	说明
hdfs dfs -ls	-ls <路径>	查看指定路径的目录结构
hdfs dfs -ls -R	-ls -R<路径>	递归查看指定路径的目录结构
hdfs dfs -cat	-cat <文件>	将指定文件内容输出到stdout
hdfs dfs -text	-text <文件>	查看文件内容
hdfs dfs -du	-du <路径>	统计目录下某个文件大小
hdfs dfs -du -s	-du -s <路径>	汇总统计目录下的文件（文件夹）大小
hdfs dfs -mv	-mv <源路经> ＜目的路径>	移动
hdfs dfs -cp	-cp <源路经> ＜目的路径>	复制
hdfs dfs -rm	-rm [-skipTrash] <路径>	删除文件/空白文件夹
hdfs dfs -rmr	-rmr [-skipTrash] <路径>	递归删除
hdfs dfs -copyFromLocal	-copyFromLocal <多个linux上的文件〉 < HDFS路径>	从本地复制到HDFS
hdfs dfs -copyToLocal	-copyToLocal <HDFS文件> <linux本地路径>	复制HDFS上的文件到本地
hdfs dfs -moveFromLocal	-moveFromLocal <多个linux上的文件> <HDFS路径>	从本地移动
hdfs dfs -getmerge	-getmerge <源路经> <linux路经>	合并到本地
hdfs dfs -put	-put linux上文件 HDFS路径	上传文件
hdfs dfs -get	-get <HDFS文件> <linux本地路径>	下载HDFS上的文件到本地
hdfs dfs -count	-count [-q] PATH	统计文件夹数量
hdfs dfs -mkdir	-mkdir <HDFS路径>	创建空白文件夹

命令	格式	说明
hdfs dfs -setrep	-setrep <副本数><路径>	修改副本数量
hdfs dfs -touchz	-touchz<文件路径>	创建空白文件
hdfs dfs -stat	-stat [format] <路径>	显示文件统计信息
hdfs dfs -tail	-tail [-f] <文件>	查看文件尾部信息
hdfs dfs -chmod	-chmod [-R]<权限模式> [路径]	修改权限
hdfs dfs -chown	-chown [-R][属主][:[属主]]路径	修改属主
hdfs dfs -chgrp	-chgrp [-R] 属组名称 路径	修改属组
hdfs dfs -checksum	-checksum <文件>	返回文件的校验信息
hdfs dfs -help	-help [命令选项]	帮助

2. 文件合法性检查

说明

使用 ***fsck*** 命令可以检查系统中文件的合法性，检查报告文件中存在的各种问题，比如文件缺少数据块或者副本数目不够等。不同于传统文件系统（例如 linux 文件系统）上的 ***fsck*** 工具，***hdfs fsck*** 命令并不会修正它检测到的错误。

表3-2 文件合法性检查命令

命令	选项	说明
hdfs fsck <path>	-	检查这个目录中的文件是否完整
hdfs fsck -move	<路径> -move	破损的文件移至/lost+found目录
hdfs fsck -delete	<路径> -delete	删除破损的文件
hdfs fsck -openforwrite	<路径> -openforwrite	打印正在打开写操作的文件
hdfs fsck -files	<路径> -files	打印正在check的文件名
hdfs fsck -blocks	<路径> -blocks	打印block报告（需要和-files参数一起使用）
hdfs fsck -locations	<路径> -locations	打印每个block的位置信息（需要和-files参数一起使用）
hdfs fsck -racks	<路径> -racks	打印位置信息的网络拓扑图（需要和-files参数一起使用）

3. 版本信息

表3-3 版本信息查看命令

命令	选项	说明
hdfs version	-	显示版本信息

4. dfsadmin

dfsadmin 命令支持一些管理 HDFS 相关的操作。使用 hdfs dfsadmin -help 命令可以查看目前所支持的命令，命令说明如表 3-4 所示。

表3-4 管理 HDFS 相关的操作命令

命令	选项	说明
hdfs dfsadmin -report	-report	显示HDFS基本的统计信息
hdfs dfsadmin -safemode	-safemode enter | leave | get | wait	手工地进入或离开安全模式
hdfs dfsadmin -printTopology	-openforwrite	打印出集群的拓扑结构。显示出机架树、机架上DataNode信息等

5. balancer

用法：hdfs balancer [-threshold <threshold>] [-policy<policy>]

HDFS 的 balance 工具常用于平衡 HDFS 集群中多个 DataNode 之间的文件块分布，以避免出现多个 DataNode 之间出现磁盘利用率不均衡的情况。

表3-5 容量均衡命令

命令	格式	说明
hdfs balancer -threshold xx –policy yy	-threshold xx -policy yy	• xx 是一个百分比，用来进行平衡。当某个 DataNode 节点的磁盘利用率高于该值时会自动进行均衡 • yy 表示平衡策略，可设置为 blockpool 或 datanode

6. snapshot

快照不仅是数据的简单拷贝，还可以进行数据备份、出错保护和容灾恢复。

表3-6 快照相关命令

命令	格式	说明
hdfs dfsadmin -allowSnapshot	-allowSnapshot <path>	指定文件目录（/input）开启快照功能
hdfs dfs -createSnapshot	-createSnapshot <path> [<snapshotName>]	在指定目录下创建快照
hdfs dfs -cp	-cp <path> <path>	文件目录创建快照之后，误删可以执行恢复命令

命令	格式	说明
hdfs lsSnapshottableDir	-	查看所有允许创建快照的目录
hdfs snapshotDirr	<path> <fromSnapshot> <toSnapshot>	比较两个快照之间的差异
hdfs dfs -renameSnapshot	-renameSnapshot <path> <oldName> <newName>	修改快照名称
hdfs dfs -deleteSnapshot	-deleteSnapshot <path> <snapshotName>	删除快照
hdfs dfsadmin -disallowSnapshot	-disallowSnapshot <path> <snapshotName>	关闭快照

7. storagepolicies

关于 HDFS 存储策略命令，详情请参见 3.10.2 HDFS 存储策略命令。

8. ec

关于 HDFS 纠删码策略命令，详情请参见 3.11.1 HDFS 纠删码命令。

3.2 Client下载/安装/使用/卸载

大数据集群提供了下载 HDFS Client 的功能。在客户端节点上安装 HDFS 的 Client 后，即可直接连接大数据集群中的 HDFS，进行组件维护、任务管理等操作。

3.2.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作，租户的 HDFS 组件也支持下载 Client），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在[集群管理/集群列表]页面，单击集群名称进入集群详情页面，在集群已安装的组件列表中单击 HDFS 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-1 所示。

图3-1 下载 Client 窗口

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：
   - 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
   - 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中的组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。
   - 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
   - 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.2.2 安装 Client

注意

- 安装 Client 的节点必须与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置网络连接，仅非 root 用户需要执行此操作，root 用户可跳过此步骤。

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

```
./install.sh <安装路径> -o
```

说明

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为 /usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

```
./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>
```

3.2.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

  ```
  source bigdata_env
  ```

- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同：
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行校验即可直接访问组件并执行相关操作。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行身份认证之后，才可访问组件并执行相关操作。关于 Kerberos 环境下，进行用户认证的操作方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

3.2.4 卸载 Client 客户端

大数据集群重装之后，之前安装的 Client 客户端将不可用。此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 安装目录下执行以下命令：

```
./uninstall.sh
```

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

3.3 权限访问控制

说明

仅开启“安全管理/权限管理”且运行正常的集群可使用角色管理功能。

权限管理是安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

3.3.1 权限说明

在开启权限管理的集群中，用户对HDFS文件或文件夹的操作需被赋予HDFS相关权限后才能执行。HDFS 支持对存储路径配置权限（支持使用通配符*模糊适配），权限包括：read、write 和 execute。HDFS 操作所需权限对应关系如表 3-7 所示。

表3-7 HDFS 权限说明

组件	权限类型	对应的组件常用操作
HDFS	read	文件（夹）的访问等相关操作，如：hdfs dfs -cat 操作
	write	数据写入等相关操作，如：hdfs dfs -touch 操作
	execute	文件（夹）执行等相关操作，可以和read或write权限结合使用。如：hdfs dfs -ls操作需要read和execute权限

开启权限管理后，除了用户角色配置的权限外，HDFS 组件默认对一些路径提供了读写操作权限，为了数据安全，建议用户不要对这些路径及其子目录进行修改和操作，提供默认权限的路径如表 3-8 所示。

表3-8 HDFS 默认权限路径列表

序号	路径	默认权限	说明
1	/app-logs	所有用户读、写、执行权限	应用程序的日志（如跑MapReduce任务的日志）
2	/apps	hdfs用户有读、写、执行权限，其他用户有读、执行权限	HBase等组件的数据路径
3	/ats	yarn用户有读、写、执行权限，其他用户有读、执行权限	TimelineService中依赖的HBase数据路径
4	/atsv2	hdfs用户有读、写、执行权限，其他用户有读、执行权限	TimelineService 2.0中依赖的HBase数据路径
5	/flink	hdfs用户有读、写、执行权限	Flink应用的日志路径
6	/hdp	hdfs用户有读、写、执行权限，其他用户有读、执行权限	应用程序依赖的jar包等文件
7	/livy2-recovery	livy用户有读、写、执行权限	Livy程序的数据
8	/mapred	mapred用户有读、写、执行权限，其他用户有读、执行权限	MapReduce服务使用该路径
9	/mlsql/models	所有用户有读、写、执行权限	Sparrow中存放机器学习的路径
10	/mr-history	所有用户有读、写、执行权限	存放MapReduce应用程序日志等内容
11	/spark2-history	所有用户有读、写、执行权限	存放Spark应用程序日志等内容
12	/sparrow-history	所有用户有读、写、执行权限	存放Sparrow应用程序日志等内容
13	/tmp	所有用户有读、写、执行权限	可以放些临时文件
14	/user	所有用户有读、写、执行权限	程序运行过程中，会存放一些用户的临时文件

序号	路径	默认权限	说明
15	/warehouse	hdfs用户有读、写、执行权限，其他用户有读、执行权限	存放Hive的数据
16	/tenant	hdfs用户有读、写、执行权限	租户模式集群下，tenant目录的文件是用户申请的HDFS资源

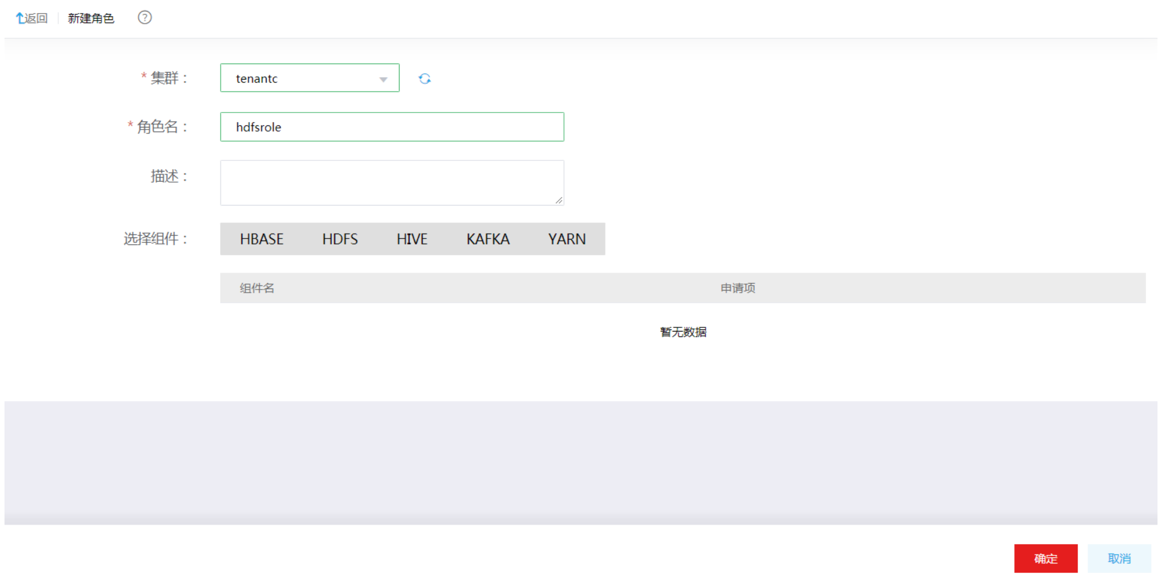
3.3.2 权限使用操作示例

下面以授予“/ats/done”文件夹的“execute、read”权限给“hdfsrole”角色，然后“hdfsrole”角色绑定给“hdfsuser”用户为例，介绍 HDFS 组件的权限访问控制。操作步骤如下：

(1) 新建角色

在[集群权限/角色管理]页面，创建角色 hdfsrole，不选择任何组件权限，如图 3-2 所示。

图3-2 新建 hdfsrole 角色



(2) 为用户绑定角色

在[集群权限/用户管理]页面，创建用户 hdfsuser，并为用户绑定角色 hdfsrole，如图 3-3 所示。

图3-3 对用户 hdfsuser 绑定角色 hdfsrole

(3) 使用 hdfsuser 用户对/ats/done 目录执行 ls 操作，如图 3-4 所示，红框内表示 hdfsuser 用户没有/ats/done 文件夹的 read 和 execute 权限。

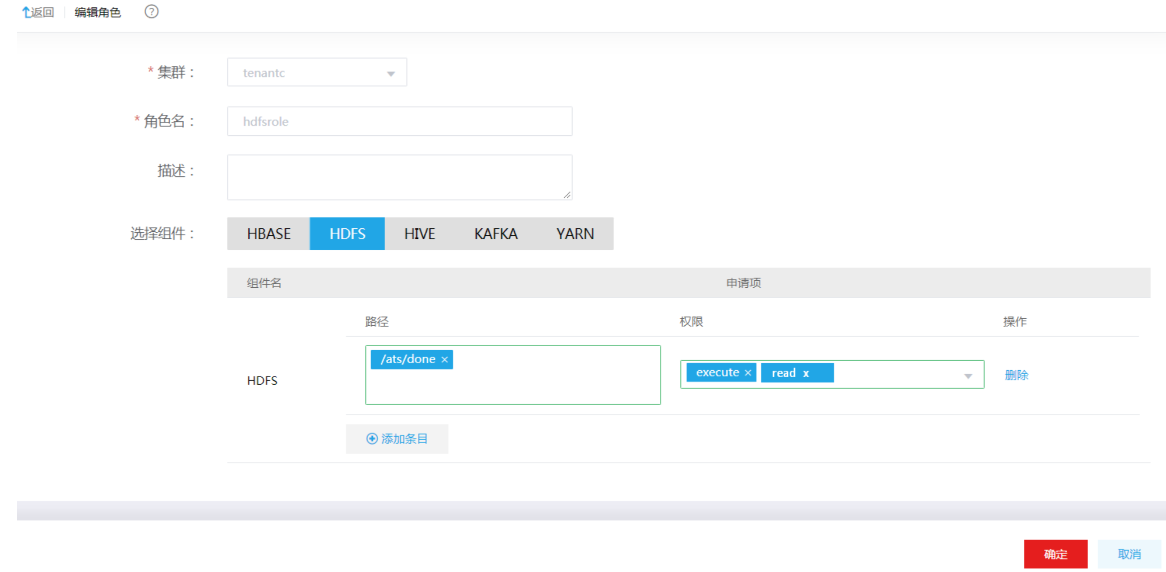
图3-4 通过 hdfsuser 用户访问/ats/done 文件夹失败

```
sh-4.2$ hdfs dfs -ls /ats/done
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1
.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-imp
l-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
ls: Permission denied: user=hdfsuser, access=READ_EXECUTE, inode="/ats/done":yarn
:hadoop:drwx------
```

(4) 修改角色授权

在[集群权限/角色管理]页面，修改角色 hdfsrole 的权限设置，授予 hdfsrole 角色对/ats/done 文件夹的 execute、read 权限，如图 3-5 所示。

图3-5 修改 hdfsrole 角色授权



(5) 等待一段时间后重新执行步骤 3，使用 hdfsuser 用户对/ats/done 目录执行 ls 操作，如图 3-6 所示，命令执行成功。

图3-6 通过 hdfsuser 用户访问/ats/done 文件夹成功

```
sh-4.2$ hdfs dfs -ls /ats/done
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1
.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-imp
l-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$
```

3.4 密钥管理

说明

- 仅开启“安全管理/密钥管理”且运行正常的集群可使用密钥管理功能。
- HDFS 组件中默认未配置密钥信息。若规划对 HDFS 数据进行透明加密，则需要手动配置，请参见 3.4.2 HDFS 配置密钥信息。
- 当前版本中，仅 HDFS 超级用户（hdfs）可创建加密区。
- 系统中的缺省密钥授权“all - keyname”不支持删除。

密钥管理是将多个集群中的密钥进行统一管理，提供密钥的创建、删除、授权等操作。在 HDFS 中使用密钥创建加密区，并实现对数据的加密功能。

3.4.1 使用指导

密钥管理使用指导如图 3-7 所示，流程说明如表 3-9 所示。

密钥授权用户说明如表 3-10 所示，密钥授权权限说明如表 3-11 所示。

图3-7 密钥管理使用指导

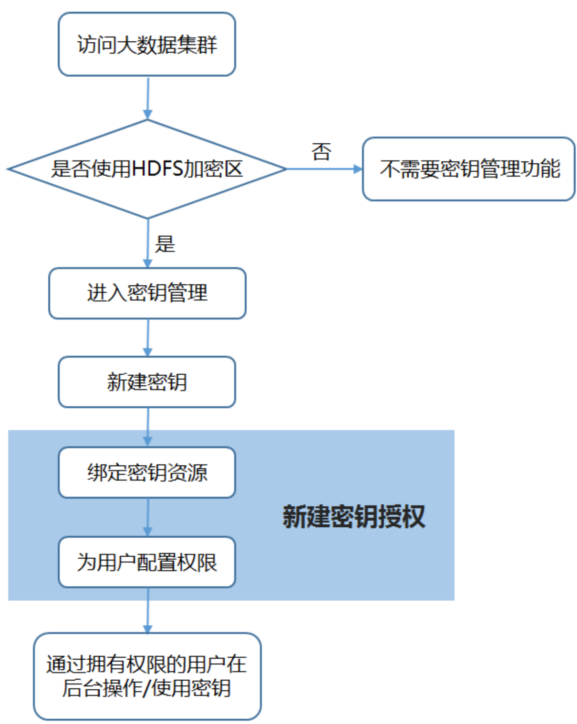


表3-9 密钥管理使用指导说明

<table>
<tr><th colspan="2">步骤</th><th>说明</th></tr>
<tr><td colspan="2">访问大数据集群</td><td>用户在后台访问大数据集群（示例用户user01）</td></tr>
<tr><td colspan="2">是否使用HDFS加密区</td><td>若需要创建HDFS加密区（示例加密区/zone）存储需要加密的数据</td></tr>
<tr><td colspan="2">进入密钥管理</td><td>则需要使用“密钥管理”功能</td></tr>
<tr><td colspan="2">新建密钥</td><td>在[密钥管理/密钥]页签，为对应集群新增密钥（示例key01）</td></tr>
<tr><td rowspan="2">新建密钥授权</td><td>绑定密钥资源</td><td>新增密钥授权时，需要绑定密钥资源（示例密钥key01）</td></tr>
<tr><td>为用户配置权限</td><td>新增密钥授权时，根据需要，可为不同用户配置不同权限（示例为用户use01绑定密钥key01，并为其授予解密加密区的权限）</td></tr>
<tr><td colspan="2">通过拥有权限的用户在后台操作/使用密钥</td><td>拥有密钥权限的用户可在后台执行对应操作，比如通过密钥向加密区上传文件或查看加密文件
【注意】当前版本中，仅HDFS超级用户（hdfs）可创建加密区
示例：hdfs用户可通过密钥key01创建加密区/zone。因用户use01已绑定解密加密区的权限，所以用户use01可通过密钥key01查看加密区/zone的里的文件</td></tr>
</table>

表3-10 密钥授权用户说明

用户类型	说明
集群中部分组件的缺省用户	比如：HDFS组件的缺省用户hdfs等
用户管理中的用户	在[集群权限/用户管理]中已新增的用户

表3-11 密钥授权权限说明

权限类型	说明
生成加密key	创建HDFS加密区的权限
解密加密key	向HDFS加密区读取、写入数据的权限

3.4.2 HDFS 配置密钥信息

HDFS 组件中默认未配置密钥信息。若规划对 HDFS 数据进行透明加密，则需要手动修改 HDFS 组的“dfs.encryption.key.provider.uri”和“hadoop.security.key.provider.path”两个配置项的值，两个配置项“dfs.encryption.key.provider.uri”和“hadoop.security.key.provider.path”的值相同，均需修改为“kms://http@<RANGER_KMS 组件的主机名>:9292/kms”，其中 RANGER_KMS 组件的主机名可在系统组件 RANGER_KMS 组件详情页面的[部署拓扑]页签获取，比如：kms://http@hostname1.hde.com;hostname2.hde.com:9292/kms。配置完成后，需要对 HDFS 和 YARN 组件进行重启。

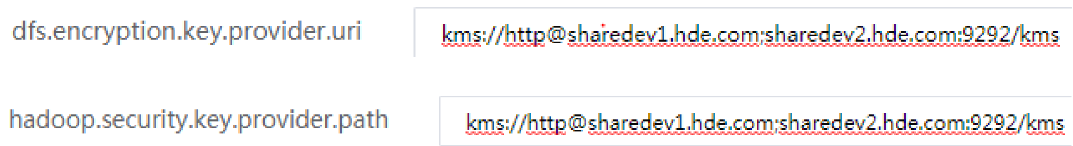
修改 HDFS 配置项的方式如下：

(1) 在集群详情的[组件/系统组件]页签，单击 RANGER_KMS 组件名，进入 RANGER_KMS 组件详情页面。查看[部署拓扑]页签，获取当前已安装启动 Ranger KMS Server 的主机名列表，如图 3-8 所示。

图3-8 RANGER_KMS 部署拓扑

(2) 在集群详情的[组件/业务组件]页签，单击 HDFS 组件名，进入 HDFS 组件详情页面。在组件详情的[配置]页签，配置 HDFS 参数项。在[高级配置]页面搜索"dfs.encryption.key.provider.uri"和"hadoop.security.key.provider.path"两个配置项，进行修改，如图 3-9 所示。

图3-9 配置 HDFS 参数项



(3) HDFS 配置项修改完成并保存后，重启 HDFS、YARN 等相关组件。

3.4.3 密钥使用操作示例

说明

执行 HDFS 密钥管理操作前，需完成相关配置，详情请参见 3.4.2 章节。

(1) 新增密钥

在[集群权限/密钥管理]页面，单击<新增密钥>按钮，新增密钥 key1，如图 3-10 所示。

图3-10 新增密钥

新增密钥

* 集群 sharedevtest

* 密钥名 key1

* 加密算法 AES/CTR/NoPadding

* 加密长度 128

描述

确定 取消

(2) 创建 HDFS 加密区

后台登录集群任一节点，切换到 hdfs 用户，创建目录/zone_encr，进而创建加密区，创建完毕后可以查看到该加密区，如图 3-11 所示。

图3-11 创建加密区

```
[root@sharedev1 ~]# su hdfs
[hdfs@sharedev1 root]$ hdfs dfs -mkdir /zone_encr
[hdfs@sharedev1 root]$ hdfs crypto -createZone -keyName key1 -path /zone_encr
Added encryption zone /zone_encr
[hdfs@sharedev1 root]$ hdfs crypto -listZones
/zone_encr  key1

You have new mail in /var/spool/mail/root
[hdfs@sharedev1 root]$
```

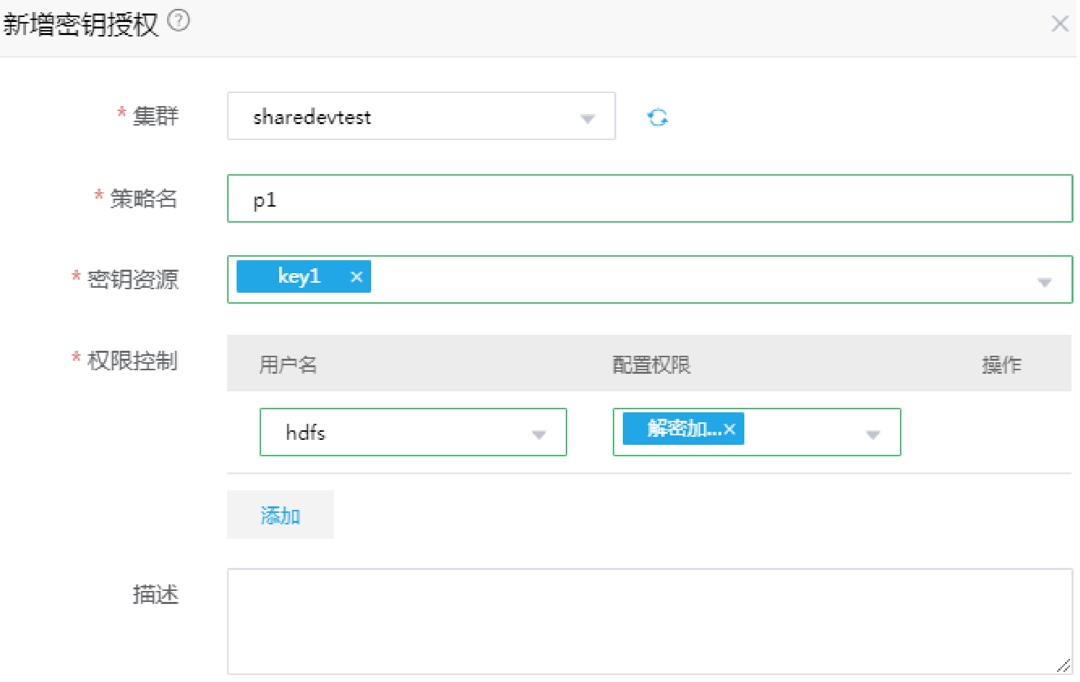
(3) 数据文件上传到加密区，提示没有解密加密 key 权限，如图 3-12 所示。

图3-12 上传数据文件

```
[hdfs@sharedev1 root]$ hdfs dfs -put /tmp/a.txt  /zone_encr
put: User:hdfs not allowed to do 'DECRYPT_EEK' on 'key1'
2021-12-28 14:59:28,900 ERROR hdfs.DFSClient (DFSClient.java:closeAllFil
to close file: /zone_encr/a.txt._COPYING_ with inode: 24158
org.apache.hadoop.ipc.RemoteException: File does not exist: /zone_encr/a
older DFSClient_NONMAPREDUCE_2020310613_1 does not have any open files.
```

(4) 在[集群权限/密钥管理]页面，选择[密钥授权]页签，单击<新增密钥授权>按钮，为用户 hdfs 配置密钥 key1 的解密加密 key 权限，如图 3-13 所示。

图3-13 新增密钥授权



(5) 再次执行数据文件上传到加密区操作，此时数据文件可成功上传到加密区。因为是透明加密，所以加密区文件数据为明文，其密文在/.reserved/raw/zone_encr/下，如图 3-14 所示。

图3-14 数据文件上传成功

```
[hdfs@sharedev1 root]$ hdfs dfs -put /tmp/a.txt  /zone_encr
[hdfs@sharedev1 root]$ hdfs dfs -ls /zone_encr
Found 2 items
drwxrwxrwt   - hdfs hdfs          0 2021-12-28 14:52 /zone_encr/.Trash
-rw-------   3 hdfs hdfs         47 2021-12-28 15:04 /zone_encr/a.txt
You have new mail in /var/spool/mail/root
[hdfs@sharedev1 root]$ hdfs dfs -cat /zone_encr/a.txt | more
hello spark
hello hadoop
wordcount
flink spark
[hdfs@sharedev1 root]$ hdfs dfs -cat  /.reserved/raw/zone_encr/a.txt | more
\▒:▒ ▒#,j▒gG.▒
```

(6) 集群超级用户拥有所有权限，可查看数据，结果如图 3-15 所示。

图3-15 查看数据

```
[root@sharedev1 ~]# su user01
sh-4.2$ kinit
Password for user01@SHAREDEVTEST.COM:
sh-4.2$ hdfs dfs -cat /zone_encr/a.txt | more
hello spark
hello hadoop
wordcount
flink spark
```

【注意】若删除密钥授权 p1 和密钥 key1，然后再新建密钥 key1 和 p1。此时，集群超级用户再查看数据会出现乱码，hdfs 用户查看数据会提示没有解密加密 key 权限，因为此时的 key1 已与之前删除的 key1 不同，如图 3-16 和图 3-17 所示。

图3-16 集群超级用户查看数据

```
sh-4.2$ hdfs dfs -cat /zone_encr/a.txt | more
▒▒▒^K``Ui}l)du▒S▒▒▒▒R▒▒b▒E$QV▒▒h▒▒▒▒|
Z
```

图3-17 hdfs 用户查看数据

```
[hdfs@sharedev1 root]$  hdfs dfs -cat /zone_encr/a.txt | more
cat: User:hdfs not allowed to do 'DECRYPT_EEK' on 'key1'
```

3.5 HDFS集群扩容

HDFS 集群扩容是指在某节点上新增安装 DataNode。

3.5.1 使用场景

随着业务量的增长，集群存储容量无法满足业务需求时，需要考虑对 HDFS 集群进行扩容。

HDFS 集群扩容的场景主要有：

- 当 HDFS 集群中每个 DataNode 的磁盘使用率超过阈值时，即产生相关告警时，可以考虑扩容。
- DataNode 的磁盘使用率查看方式：通过 HDFS 组件的快速链接，进入 HDFS 文件系统管理页面，在[DataNodes]页签查看集群 DataNode 信息。如果看到每个 DataNode 的磁盘使用率很高，则可以考虑进行扩容。

3.5.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在大数据集群中新增安装 DataNode 进程。

- 如果集群中有节点没有安装 DataNode，直接在集群节点中添加 DataNode 进程。
- 如果集群中所有节点均已安装 DataNode，进行 DataNode 扩容前则需要先添加主机。

2. 环境检查

(1) 登录大数据平台管理系统，查看 HDFS 组件的状态是否正常。

(2) 进入 HDFS 组件详情页，查看 HDFS 的部署拓扑，确保集群中每个服务的状态正常，HDFS NameNode、JournalNodes、DataNode、ZKFailoverController 处于“已启动”状态，HDFS Client 处于“已安装”状态。

3.5.3 扩容约束

- 扩容节点操作系统版本与集群内部版本需保持一致。
- 扩容操作一旦开始，不支持中止。

3.5.4 扩容影响

- 一旦扩容失败，需要及时将扩容失败的节点剔除。
- 扩容成功后，HDFS 集群的数据存储能力会得到增强。

3.5.5 扩容操作指导

注意

若集群中所有节点均已安装 DataNode，进行 DataNode 扩容前则需要先添加主机，然后再进行 DataNode 扩容。如果集群中已有扩容所需主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 HDFS 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-18 所示。

a. 选择进程及主机

在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

b. 部署进程

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-18 添加进程

(3) 查看进程变化

DataNode 扩容完成之后，在组件详情页面[部署拓扑]页签中可以查看 DataNode 进程的数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.5.6 DataNode 扩容后数据均衡

HDFS 执行 DataNode 进程扩容操作后，各个 DataNode 节点之间需要进行数据均衡，此时需要跨数据节点进行数据迁移。

HDFS 数据均衡为 HDFS 原生支持的功能，操作方式请参见 3.9.1 数据均衡。

3.5.7 扩容验证

(1) 登录大数据平台管理系统。

(2) 执行 HDFS 组件检查，确保 HDFS 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 HDFS 组件部署拓扑里是否已有新增的扩容节点。

(4) 打开 HDFS 快速链接，在 HDFS UI 页面查看 DataNode 的最新状态信息是否符合预期。

3.6 HDFS集群缩容

HDFS 集群缩容是指将某节点上已安装的 DataNode 进行删除。

3.6.1 使用场景

HDFS 集群缩容的场景主要有：

- 初始 DataNode 节点规划不合理。
- NameNode 和 DataNode 部署在同一个节点上，导致该节点压力过大，需要进行分离。
- 当 DataNode 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.6.2 缩容前准备

1. 缩容规划

(1) 进行缩容分析，确定缩容场景。

(2) 要求缩容后 DataNode 的数量不得少于 HDFS 副本数。

2. 环境检查

(1) 登录大数据平台管理系统，查看 HDFS 组件的状态是否正常。

(2) 进入 HDFS 组件详情页，查看 HDFS 的部署拓扑，确保集群中每个服务的状态正常，HDFS NameNode、JournalNodes、DataNode、ZKFailoverController 处于“已启动”状态，HDFS Client 处于“已安装”状态。

(3) 检查 HDFS 文件副本数，HDFS 的 DataNode 缩容前要进行数据备份，需确保 HDFS 文件为 3 副本。

a. 登录集群节点，以 hdfs 用户执行命令查询 HDFS 文件是否存在 0 副本、1 副本、2 副本。命令如下：

hdfs fsck / -files -blocks | grep 'replication=X'命令

【说明】

– 参数 X：表示副本数，取值为：0、1、2；

– 示例：若 X 值为 1，命令即 hdfs fsck / -files -blocks | grep 'replication=1'，表示查看单副本（即 1 副本）文件。

b. 执行查询命令后，若查询结果存在 0 副本、1 副本、2 副本，则需要设置对应副本文件为 3 副本，命令如下：

hdfs dfs -setrep -R 3 {查询出来的文件}

说明

若 HDFS 副本数不为 3，将要缩容节点上的数据进行拷贝后再执行缩容操作时，要缩容的 DataNode 实例会一直处于数据块拷贝中。

3.6.3 缩容约束

- 为保证数据安全，一般不建议对 HDFS 集群进行缩容操作。
- 在生产环境中，缩容不可回退或暂停，请谨慎使用。
- 删除进程前请确认该节点上无任务在执行。

3.6.4 缩容影响

- DataNode 缩容后，HDFS 的存储空间会减小，需要评估对数据存储业务的影响。

3.6.5 缩容操作指导

注意

HDFS 的 DataNode 缩容退役过程比较慢，可以在 hdfs-site 的自定义配置增加或调整如下 3 个参数进行提速：

dfs.namenode.replication.work.multiplier.per.iteration = 32

dfs.namenode.replication.max-streams = 64

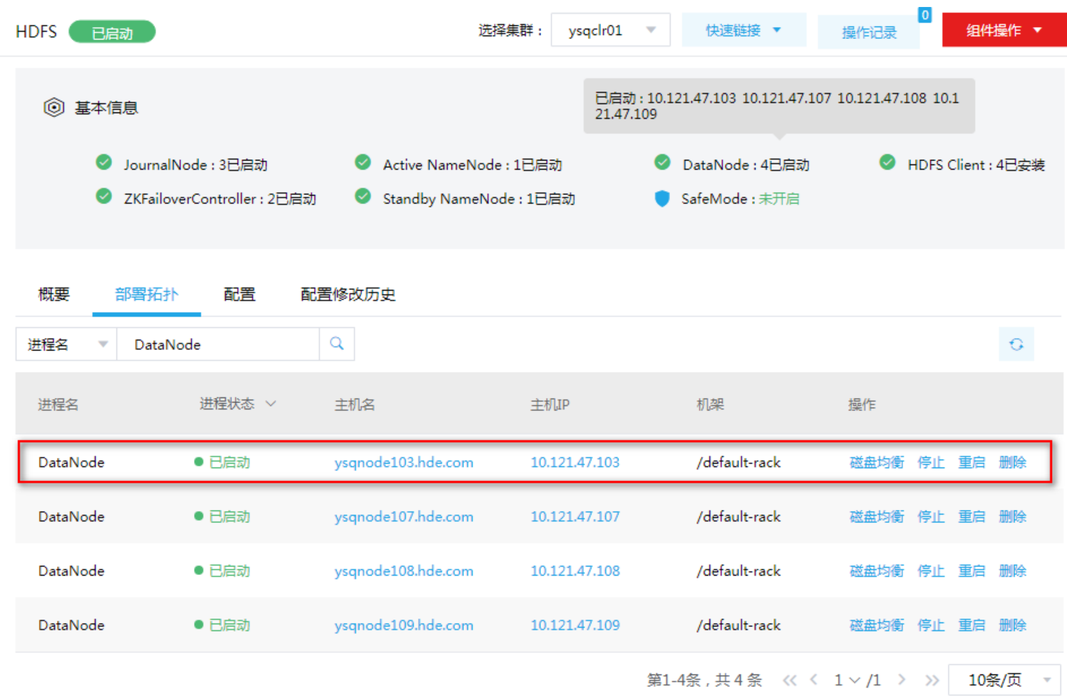
dfs.namenode.replication.max-streams-hard-limit = 128

参数配置完成后，需重启 HDFS 使配置生效。

为了避免数据丢失，需将要缩容节点上的数据进行拷贝后再执行缩容操作，操作步骤如下：

(1) 在 HDFS 组件详情页面，选择缩容节点，下面以对 ysqnode103.hde.com 节点执行 DataNode 缩容操作为例进行说明，如图 3-19 所示。

图3-19 选择缩容 DataNode



(2) 登录到Active Namenode节点后台，把需要缩容的DataNode的主机名ysqnode103.hde.com 加入到/etc/hadoop/conf/hdfs-site.xml 的配置项 dfs.hosts.exclude 所指定的文件中。该文件默认为/etc/hadoop/conf/dfs.exclude，若文件不存在，则需要创建文件并设置该文件的用户组为 hdfs:hadoop，否则没有权限，如图 3-20 所示。

图3-20 查看/etc/hadoop/conf/dfs.exclude

```
[root@ysqnode107 ~]# cat /etc/hadoop/conf/dfs.exclude
ysqnode103.hde.com
```

说明

若/etc/hadoop/conf/hdfs-site.xml 中不存在 dfs.hosts.exclude 的配置项，则需在 Active Namenode 节点手动在将该属性加入/etc/hadoop/conf/hdfs-site.xml 配置文件中，如下所示：

<property>

<name>dfs.hosts.exclude</name>

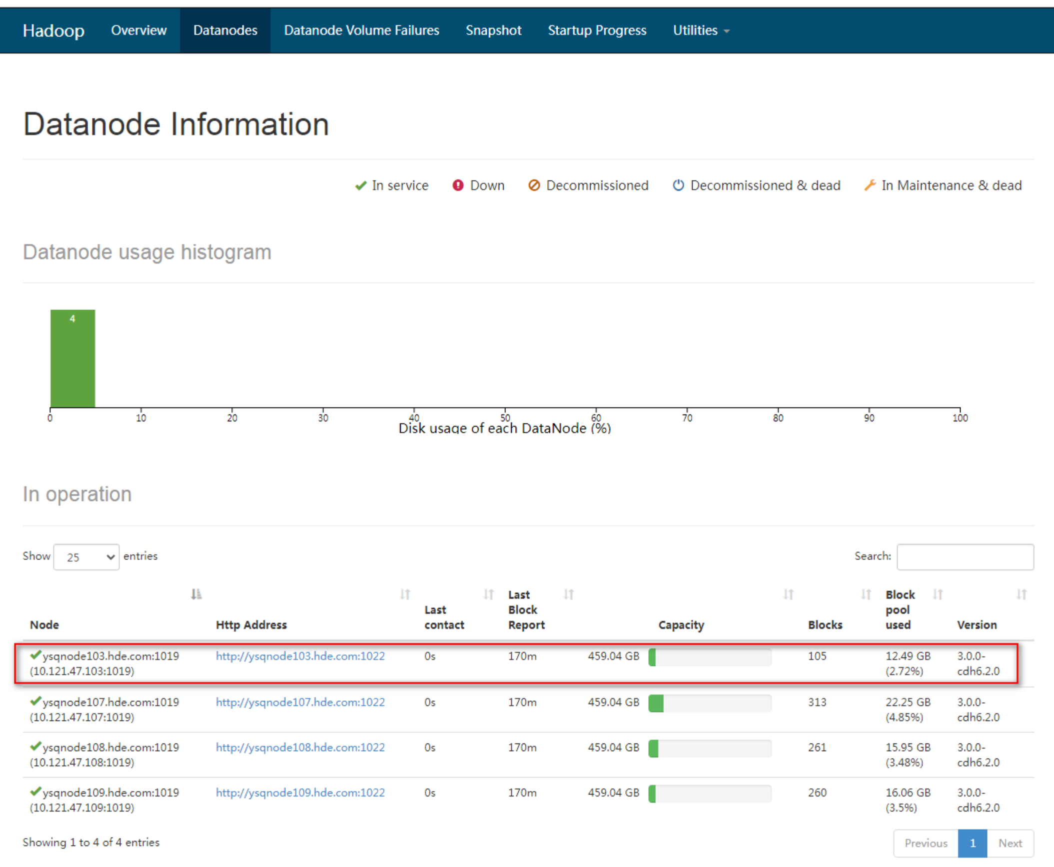
<value>/etc/hadoop/conf/dfs.exclude</value>

</property>

(3) 在 Active NameNode 节点，以 hdfs 用户执行命令：hdfs dfsadmin -refreshNodes

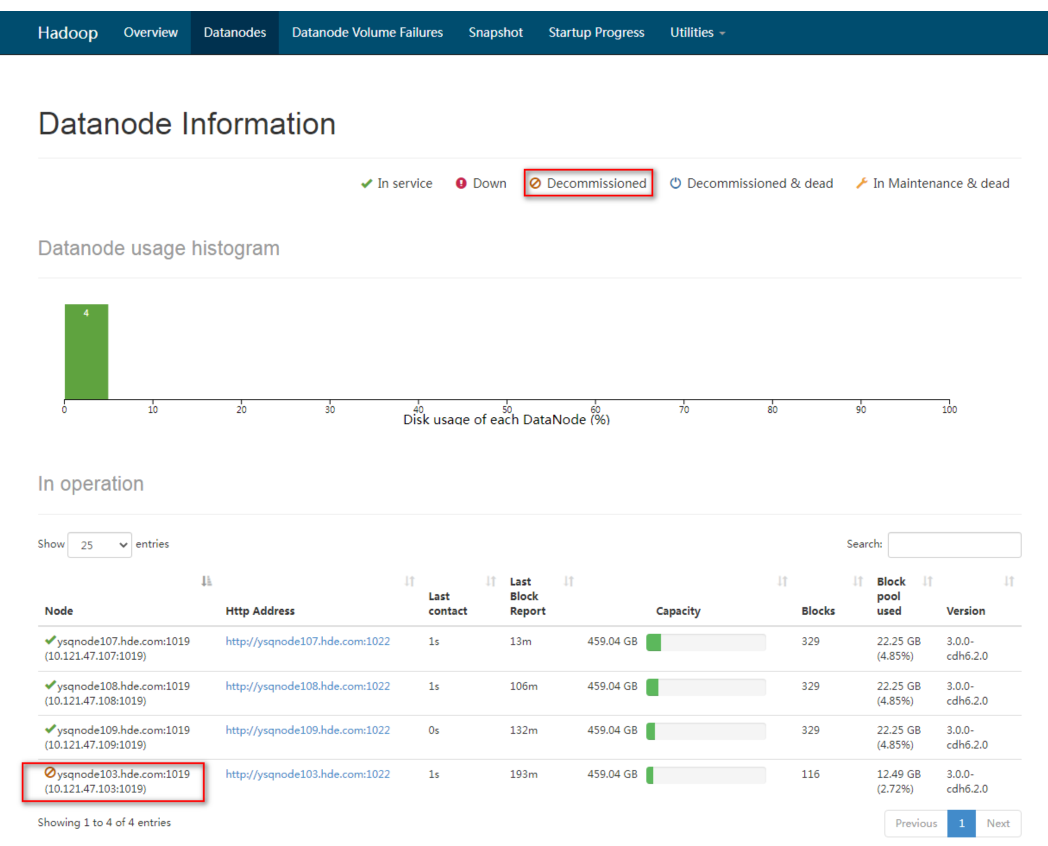
(4) 打开 NameNode Web UI，跳转到 Datanodes 页面，检查要缩容的主机 ysqnode103.hde.com 的 DataNode 的状态是否已更改为如图 3-21 的状态，若主机名称 ysqnode103.hde.com 前没有“√”，表示此时 DataNode 正在复制数据块。

图3-21 查看 Datanodes 状态



(5) 当节点所有块均已复制完成后，要缩容的主机 ysqnode103.hde.com 的状态将变为“Decommissioned”，如图 3-22 所示。

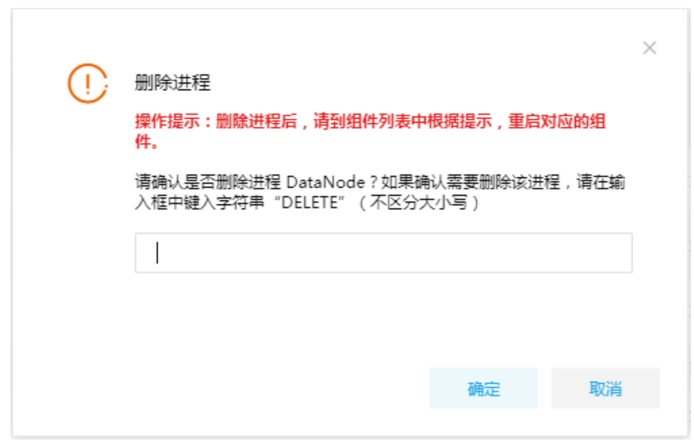
图3-22 查看要缩容主机状态



(6) 在 HDFS 组件详情页面的[部署拓扑]页签下，先停止 DataNode 进程，然后再单击<删除>按钮，在弹出的窗口中输入“DELETE”后单击<确定>，即可完成删除 DataNode，如图 3-23 和图 3-24 所示。

图3-23 停止 DataNode

图3-24 删除 DataNode



说明

删除 DataNode 不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。此处仅以“在组件详情页面的[部署拓扑]页签下执行删除 DataNode 操作”为例进行说明，在主机详情页面执行删除 DataNode 操作，与其类似不再进行说明。

(7) 在 Active NameNode 主机上，清空步骤(2)中/etc/hadoop/conf/dfs.exclude 文件添加的要缩容的节点的主机名，以 hdfs 用户执行命令 hdfs dfsadmin –refreshNodes 即可。

3.6.6 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 HDFS 组件检查，确保 HDFS 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 HDFS 组件部署拓扑里相关缩容节点是否已经删除。

(4) 打开 HDFS 快速链接，在 HDFS UI 页面查看 DataNode 的最新状态信息是否符合预期。

3.7 租户管理

注意

- 新增租户时，要求系统中必须已成功创建租户模式的 Hadoop 集群。
- HDFS 组件默认对一些路径提供了读写操作权限（比如：/tmp、/user 目录），为了数据安全，建议用户不要对这些路径及其子目录进行修改和操作，提供默认权限的路径如表 3-8。
- 关于租户管理模块功能的配置或使用详情请参见产品在线联机帮助。

多个租户之间共享一套集群，共享网络和集群资源，并且不同租户之间保证资源隔离。

- 新增租户

  普通用户在自己创建的租户集群中申请租户时，无需审批，会直接触发新增租户的操作。普通用户在其他用户创建的租户集群中申请租户时，则需要走流程审批，待审批人审批通过后才能触发新增租户的操作。管理员用户新增租户时，无需审批，会直接触发新增租户的操作。

- 租户管理操作

  普通用户在自己创建的租户集群中执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。普通用户在其他用户创建的租户集群中执行租户续期、资源扩缩容操作时，则需要走流程审批，待审批人审批通过后才能触发相关操作。管理员用户执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。

3.7.1 租户介绍

租户申请的HDFS资源对应一个或多个限额的存储目录，用户可根据实际需要申请HDFS组件资源。

【说明】

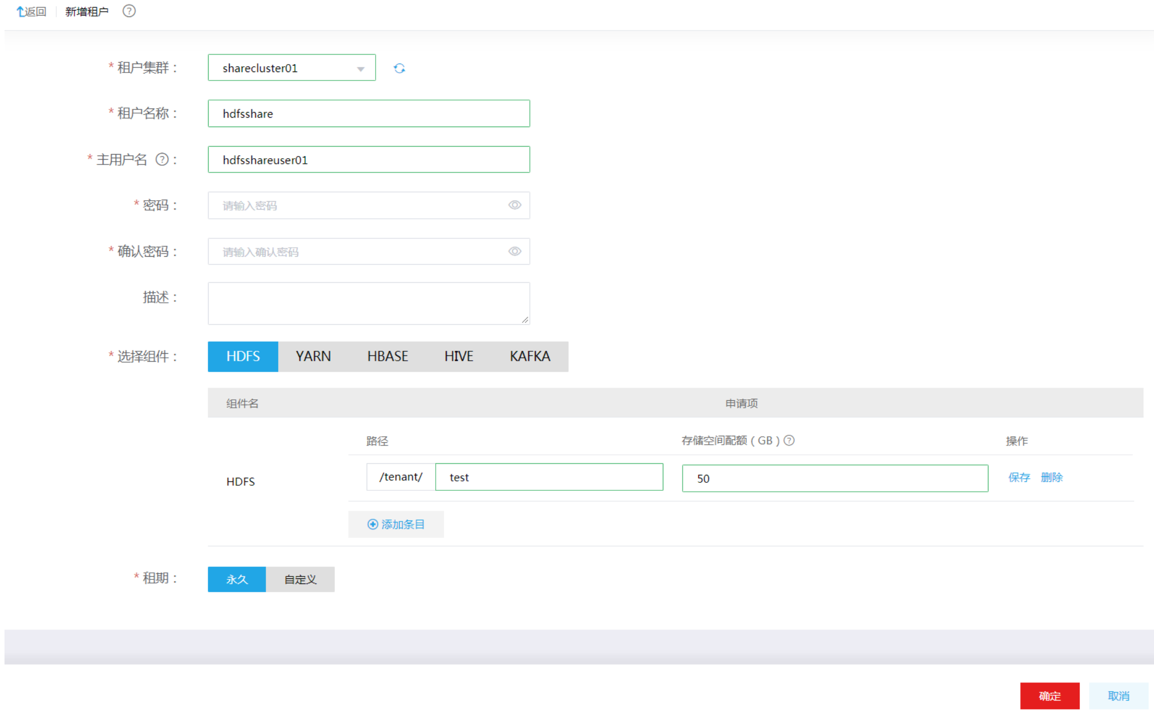
大数据集群的存储资源是HDFS中可分配的数据存储空间。目录是HDFS存储资源分配的基本单位，租户通过指定 HDFS 文件系统的目录来获取存储资源。

租户申请的 HDFS 资源，系统默认分配在“/tenant”目录下（该路径不支持自定义修改），所以为租户申请 HDFS 资源时需要在“/tenant”目录中创建文件夹并指定该文件夹对应的存储空间配额，例如在“/tenant/test”目录中申请 10GB 的存储空间配额。

3.7.2 新增租户

(1) 在[集群权限/租户管理]页面，单击<新增租户>按钮，跳转至新增租户页面，如图 3-25 所示。在租户集群 sharecluster01 中新增租户 hdfsshare，主用户 hdfsshareuser01，并为该租户配置 HDFS 组件资源（路径/tenant/test、50GB 存储空间配额）。

图3-25 新增租户

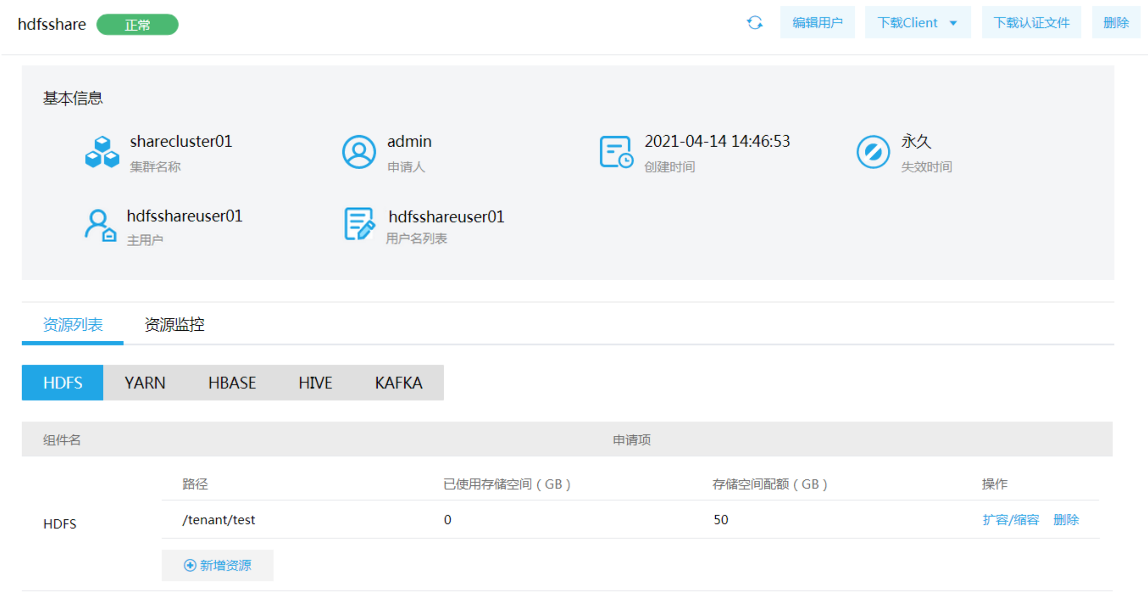


(2) 新增租户成功后，用户可在租户列表查看到已创建的租户，同时可以看到其所属集群、申请人、用户名列表、创建时间、失效时间等相关信息，如图 3-26 所示。

图3-26 查看租户

(3) 单击租户名称，可查看租户详情，如图 3-27 所示，可以看到对应的 HDFS 路径和存储空间配额等信息。用户 hdfsshareuser01 拥有 HDFS 租户 hdfsshare 的所有权限，若资源不够/过多时，可编辑租户对其执行扩容/缩容操作。

图3-27 查看租户详情



3.7.3 租户使用操作示例

注意

- 租户集群缺省开启 Kerberos 认证，在使用租户时需要通过该租户的用户对应的认证文件，对租户的用户进行身份认证。租户的用户认证文件的下载在租户列表页面或租户详情页面执行，关于对租户的用户进行认证的方式与集群中用户的身份认证方式相同，详情请参见1. Kerberos 环境下用户身份认证。
- HDFS 租户包括客户端（即组件 Client），系统提供了下载 HDFS 客户端的功能。在客户端节点上安装 HDFS 的 Client 后，即可连接租户中的此组件，执行组件维护、任务管理等操作。租户组件 Client 的下载在租户详情页面执行，关于租户组件 Client 的安装方式与集群组件 Client 相同，详情请参见3.2 Client 下载/安装/使用/卸载。

租户集群缺省开启 Kerberos 认证，通过租户用户对查看租户或对租户执行管理操作时，均需要提前对租户的用户执行身份认证操作。

(1) 通过租户用户查看租户

使用 hdfsshare01 用户登录到租户集群，使用 ***hdfs dfs -ls /tenant/test*** 命令可查看到 HDFS 中已申请的***/tenant/test*** 租户路径，如图 3-28 所示。

图3-28 查看命名空间

```
[root@hppnode1 tmp]# su hdfsshare01
sh-4.2$ hdfs dfs -ls /tenant/test
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-
1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-im
pl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$
```

(2) 使用 hdfsshare01 用户登录到租户集群，在/tenant/test 租户路径可执行增删改查等管理操作，如图 3-29 所示。

- 创建/tenant/test/txt 子目录：hdfs dfs -mkdir /tenant/test/txt
- 将 hdfs.txt 文件上传至 txt 目录下：hdfs dfs -put /home/hdfsshare01/hdfs.txt /tenant/test/txt
- 查看 hdfs.txt 文件上传是否成功：hdfs dfs -ls /tenant/test/txt

图3-29 操作 HDFS

```
sh-4.2$ hdfs dfs -mkdir /tenant/test/txt
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-
1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-im
pl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$ hdfs dfs -put /home/hdfsshare01/hdfs.txt /tenant/test/txt
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-
1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-im
pl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$ hdfs dfs -ls /tenant/test/txt
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-
1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-im
pl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Found 1 items
-rw-------   3 hdfsshare01 hdfs         19 2020-04-14 10:48 /tenant/test/txt/hdf
s.txt
sh-4.2$
```

3.8 备份恢复

备份恢复可以提供跨集群之间的数据同步功能，当前版本支持对 HDFS 组件中的指定数据进行同步备份，以保证数据内容不丢失。

备份恢复功能主要是通过创建同步任务实现集群间的数据同步能力。使用同步任务功能，可以为集群中的数据提供同步能力，以满足日常数据备份的需求，保证系统或机器故障时的数据不丢失。

HDFS 同步任务为周期性调度任务。HDFS 同步任务周期性执行时，默认首次执行是全量数据备份，后续执行是增量数据备份，备份周期间隔时间内的数据不会实时同步到目的集群。

3.8.1 新建 HDFS 同步任务

注意

- 源集群为同步任务的数据输出集群（一般指新建同步任务的本集群）；目的集群为同步任务的数据输入集群。
- 当前版本中，仅当集群存储类型为 HDFS 时（且源集群和目的集群的存储类型必须均为 HDFS），才支持创建同步任务。
- 新建同步任务时，要求源集群与目的集群的集群类型、集群模式均相同。
- 新建同步任务时，要求源集群与目的集群的安全管理策略相同，即同时开启 Kerberos 认证或同时都没有开启 Kerberos 认证。
- 新建同步任务时，要求源集群与目的集群的集群名称、节点主机名不相同，否则配置跨集群互信时可能出错。
- 新建 HDFS 同步任务时，建议任务执行集群选择为目的集群，用以避免同步任务执行时影响源集群的业务。
- 新建 HDFS 同步任务时，若目的集群中已存在配置的目的路径，则同步数据采用增量同步的方式。即执行 HDFS 同步任务后，将修改此路径下已存在同名文件与源集群文件中数据保持一致，此路径下其他文件不删除。
- 执行 HDFS 同步任务后，目录或文件的删除操作不会被同步。
- 使用 HDFS 同步任务备份 HDFS 加密区的数据时，需要根据集群的配置情况进行相关配置修改，详情请参见 3.8.4 HDFS 加密区数据同步。

集群在使用过程中，根据实际需要，可新建 HDFS 同步任务，将源集群中的 HDFS 某些目录下的数据周期性拷贝到目的集群中。

1. 前提条件

- 新建 HDFS 同步任务前，需要对源集群与目的集群配置跨集群互信。配置方法详情请参见 3.8.2 源集群和目的集群配置互信。
- 运行 HDFS 同步任务前，需要根据集群的配置情况进行相关配置修改，详情请参见 3.8.3 HDFS 同步任务相关配置。

2. 新建 HDFS 同步任务

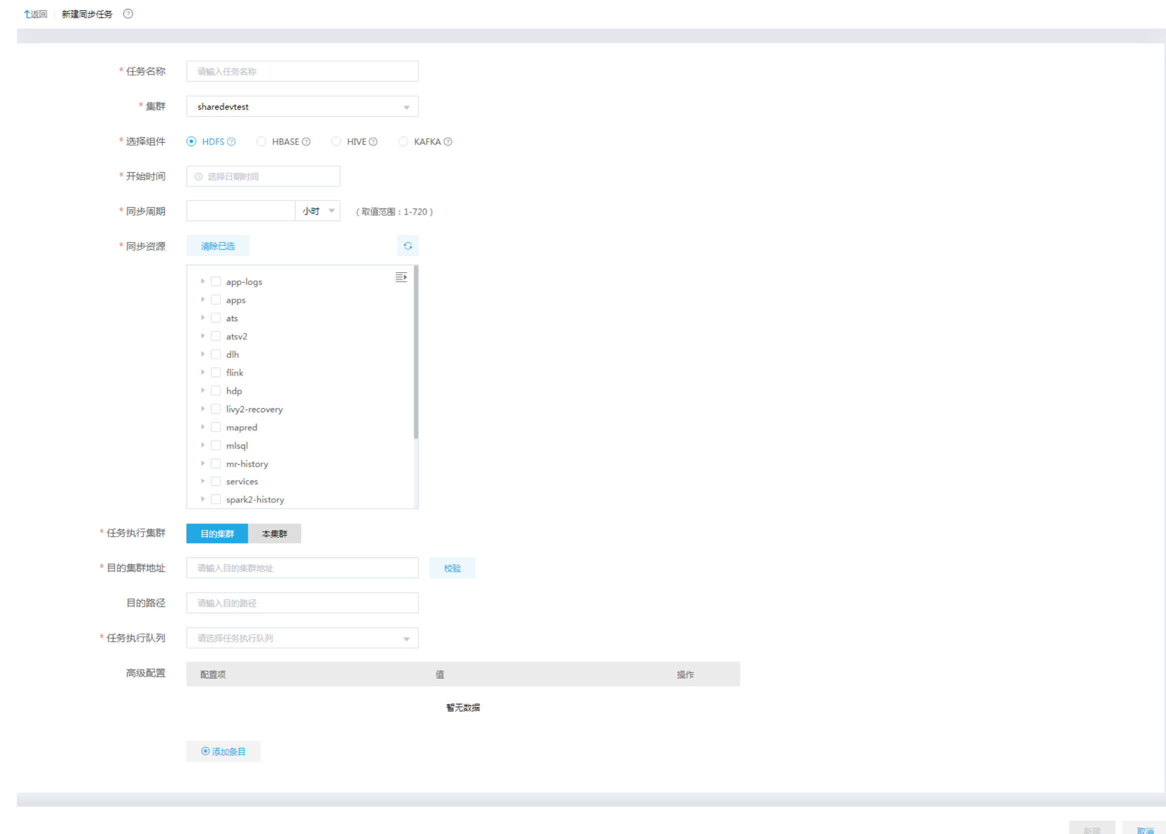
新建 HDFS 同步任务的前提条件准备完成后，即可开始创建 HDFS 同步任务。步骤如下：

(1) 在[集群管理/备份恢复]页面，选择[同步任务]页签，单击<新建同步任务>按钮，进入新建同步任务页面。

(2) 选择 HDFS 组件，如图 3-30 所示，根据提示配置对应参数项的值，关于各参数项配置详情请参见产品在线联机帮助。

(3) 相关信息配置完成后，单击<新建>按钮即可成功新建 HDFS 同步任务，此时任务到达调度时间后即可被执行。

图3-30 新建 HDFS 同步任务



3.8.2 源集群和目的集群配置互信

注意

不同集群之间可通过组件同步任务进行数据同步，但创建同步任务之前必须配置源集群和目的集群互信。

示例集群如下：

- 源集群
  - 开启 Kerberos 认证集群 clusterA，kerberos realm 为 CLUSTERA.COM。
- 目的集群
  - 开启 Kerberos 认证集群 clusterB，kerberos realm 为 CLUSTERB.COM。

1. 开启 kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

(3) 修改源集群和目的集群中所有节点的/etc/krb5.conf 文件，要求：

- 修改 realms，要求同时包含源集群和目的集群的 realms 内容（互相拷贝即可），如图 3-31 所示。
- 修改 domain_realm，要求同时包含源集群和目的集群的 domain_realm 内容。
  - 当源集群与目的集群的主机名后缀相同时，需要将 domain_realm 内容修改为“集群中各节点主机名=对应的 realms 名”。示例：源集群和备集群的主机名后缀均为.hde.com，则配置如图 3-32 所示。
  - 当源集群与目的集群的主机名后缀不同时，则可直接将两个集群的 domain_realm 中内容合并（不需要修改，直接互相拷贝即可）。示例：源集群的主机名后缀为.hde.com，目的集群的主机名后缀为.hadoop.com，则配置如图 3-33 所示。

图3-31 realms 修改后示例

```
[realms]
    CLUSTERA.COM = {
        kdc = clustera1.hde.com
        admin_server = clustera1.hde.com
        database_module = openldap_ldapconf
    }
    CLUSTERB.COM = {
        kdc = clusterb1.hde.com
        admin_server = clusterb1.hde.com
        database_module = openldap_ldapconf
    }
```

图3-32 domain_realm 修改后示例（源集群和目的集群主机名后缀相同）

```
[domain_realm]
   clustera1.hde.com=CLUSTERA.COM
   clustera2.hde.com=CLUSTERA.COM
   clustera3.hde.com=CLUSTERA.COM
   clusterb1.hde.com=CLUSTERB.COM
   clusterb2.hde.com=CLUSTERB.COM
   clusterb3.hde.com=CLUSTERB.COM
```

图3-33 domain_realm 修改后示例（源集群和目的集群主机名后缀不同）

```
[domain_realm]
   .hde.com = CLUSTERA.COM
   hde.com = CLUSTERA.COM
   .hadoop.com = CLUSTERB.COM
   hadoop.com = CLUSTERB.COM
```

(4) 在源集群和目的集群的 Master 节点上，执行添加 principal 操作。

a. 在源集群 Master 节点上，分别执行以下两条命令，以添加 principal（命令执行后，需要分别输入密码）：

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '

b. 在目的集群 Master 节点上，分别执行以下两条命令，添加 principal（命令执行后，需要分别输入密码）：

```
kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '
```

```
kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '
```

【注意】

- CLUSTERA.COM 和 CLUSTERB.COM 分别为源集群和目的集群的域名，请根据实际情况进行修改。添加 principal 时需确保两个集群输入的密码相同（即上述四条命令运行后输入的密码均相同），且密码要求至少 8 位，否则会提示密码太短导致设置无效。
- 若添加 principal 时输入的密码不同，可在源集群和目的集群上进行删除，然后重新执行第 4 步添加 principal 的操作。删除命令如下：

```
kadmin.local -q ' delprinc krbtgt/CLUSTERA.COM@CLUSTERB.COM'
```

```
kadmin.local -q ' delprinc krbtgt/CLUSTERB.COM@CLUSTERA.COM'
```

(5) 源集群与目的集群互信配置完成后，可登录目的集群进行校验。校验方式示例：在目的集群后台切换至集群超级用户，执行命令 hdfs dfs -ls hdfs://<源集群 Active NameNode IP 地址>:8020/，查看源集群的 HDFS 是否可正常访问，若能正常访问则表示源集群与目的集群的互信配置成功。

2. 不开 kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

3.8.3 HDFS 同步任务相关配置

1. 开启 Kerberos 认证的环境

说明

HDFS 配置项 hadoop.security.auth_to_local 的值只能通过 Default 配置组进行修改，其他配置组下此配置项的值会自动从 Default 配置组同步。

若源集群和目的集群都开启了 Kerberos 认证，创建 HDFS 同步任务之前，需进行以下配置修改：

- 对源集群进行配置修改：在源集群的[集群管理/集群详情/业务组件/HDFS 组件详情]页面，修改 HDFS 配置项 hadoop.security.auth_to_local 的值。要求在末尾 DEFAULT 之前增加内容“ RULE:[1:$1@$0](.*@CLUSTERB.COM)s/@.*// ”，其中 CLUSTERB.COM 为目的集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 HDFS 组件。
- 对目的集群进行配置修改：在目的集群的[集群管理/集群详情/业务组件/HDFS 组件详情]页面，修改 HDFS 配置项 hadoop.security.auth_to_local 的值。要求在末尾 DEFAULT 之前增加

内容“RULE:[1:$1@$0](.*@CLUSTERA.COM)s/@.*//”，其中 CLUSTERA.COM 为源集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 HDFS 组件。

2. 不开 Kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则在集群之间运行 HDFS 同步任务时，不需要进行相关配置的修改。

3.8.4 HDFS 加密区数据同步

说明

无论集群是否开启 Kerberos，使用 HDFS 同步任务备份 HDFS 加密区的数据时，需要在新建 HDFS 同步任务时进行高级配置，将配置项 distcp.skip.crc 的值设置为 true，否则会抛出 Checksum mismatch 错误。

1. 开启 Kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，在集群之间备份 HDFS 加密区数据时，需提前进行以下配置修改：

- 对源集群进行配置修改，说明如下：

  在源集群的[集群列表/集群详情/系统组件/RANGER_KMS 组件详情]页面，修改 RANGER_KMS 配置项 hadoop.kms.authentication.kerberos.name.rules 的值。要求在末尾 DEFAULT 之前增加内容“RULE:[1:$1@$0](.*@CLUSTERB.COM)s/@.*//”，其中 CLUSTERB.COM 为目的集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 RANGER_KMS 组件。
- 对目的集群进行配置修改，说明如下：

  在目的集群的[集群列表/集群详情/系统组件/RANGER_KMS 组件详情]页面，修改 RANGER_KMS 配置项 hadoop.kms.authentication.kerberos.name.rules 的值。要求在末尾 DEFAULT 之前增加内容“RULE:[1:$1@$0](.*@CLUSTERA.COM)s/@.*//”，其中 CLUSTERA.COM 为源集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 RANGER_KMS 组件。

2. 不开启 Kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则在集群之间备份 HDFS 加密区数据时，不需要进行相关配置的修改。

3.9 数据重分布

3.9.1 数据均衡

注意

- 数据均衡指在 HDFS 的 DataNode 节点之间进行数据均衡，此时需要跨数据节点进行数据迁移，且该操作仅支持在同一个机架上 DataNode 节点之间进行数据均衡。
- 数据均衡在执行过程中会消耗集群资源，影响集群的性能，建议在集群中无业务运行时执行此操作。
- 在数据量较大的情况下，数据均衡操作需要较长的时间才能执行完成，会对集群性能造成影响。
- 数据均衡操作的运行时间取决于集群大小和数据的不平衡程度。

大数据集群在使用过程中，存储的数据可能会出现分布不均衡的现象。当 HDFS 数据出现分布不均衡的时候，会引发很多问题，比如：MapReduce 程序无法很好地利用本地计算的优势、机器之间无法达到更好的网络带宽使用率等。此时，可以使用 HDFS 数据均衡的功能，平衡集群不同 DataNode 节点上的数据分布。

数据均衡功能是 HDFS 原生支持，该功能的操作包含两种方式：

- 在后台使用命令行进行数据均衡，命令行说明详情请参见 3.1 5. 章节，操作方式请参见 3.9.1 2. 章节。
- 在大数据平台管理系统直接通过页面操作进行数据均衡，操作方式请参见 3.9.1 1. 章节。

1. 页面操作方式

在大数据平台管理系统，直接通过页面操作进行数据均衡的操作步骤如下：

(1) 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。

(2) 在集群详情页面选择[组件]页签，单击组件列表中 HDFS 名称进入组件详情页面，在右上角组件操作的下拉框中选择<数据均衡>按钮，弹出数据均衡窗口，如图 3-34 所示。

参数说明如下：

- 均衡阈值：指每个 DataNode 节点上的磁盘容量使用率与 HDFS 集群中各个 DataNode 节点上的磁盘容量使用率的平均值之间的差值，此差值即为数据均衡后允许的阈值范围。
- 带宽：HDFS 执行数据均衡操作时允许占用的最大网络带宽，最大带宽的设置会影响数据均衡的速率，单位 MB/s。

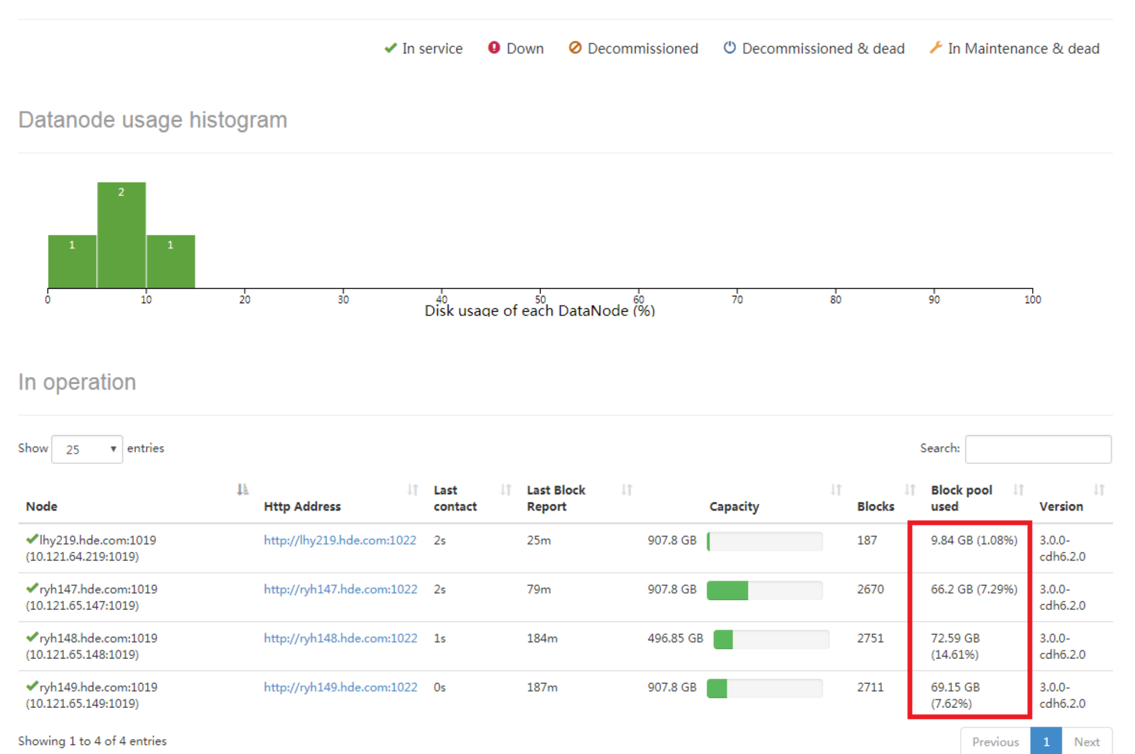
图3-34 数据均衡

(3) 参数配置完成后，单击<确定>按钮即可对 HDFS 数据执行均衡操作。

(4) 查看数据均衡效果

在 HDFS 组件详情页面的右上角[快速链接]的下拉框中，通过选择进入 HDFS 文件系统管理页面。选择[DataNodes]页签，进入 DataNodes 页面，可以查看集群 DataNode 信息及均衡效果，如图 3-35 所示。

图3-35 DataNode 节点均衡情况



Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✔lhy219.hde.com:1019 (10.121.64.219:1019)	http://lhy219.hde.com:1022	2s	25m	907.8 GB	187	9.84 GB (1.08%)	3.0.0-cdh6.2.0
✔ryh147.hde.com:1019 (10.121.65.147:1019)	http://ryh147.hde.com:1022	2s	79m	907.8 GB	2670	66.2 GB (7.29%)	3.0.0-cdh6.2.0
✔ryh148.hde.com:1019 (10.121.65.148:1019)	http://ryh148.hde.com:1022	1s	184m	496.85 GB	2751	72.59 GB (14.61%)	3.0.0-cdh6.2.0
✔ryh149.hde.com:1019 (10.121.65.149:1019)	http://ryh149.hde.com:1022	0s	187m	907.8 GB	2711	69.15 GB (7.62%)	3.0.0-cdh6.2.0

2. 命令行操作方式

数据均衡功能是 HDFS 原生支持，可在后台使用命令行直接进行数据均衡，命令行说明详情请参见 3.1 5. 章节，操作示例如下：

(1) 切换至 hdfs 用户，执行数据均衡命令，如下：

hdfs balancer -threshold 10

图3-36 通过命令行进行数据均衡

```
[hdfs@li217 ~]$ hdfs balancer -threshold 10
WARNING: HADOOP_BALANCER_OPTS has been replaced by HDFS_BALANCER_OPTS. Using value of
 HADOOP_BALANCER_OPTS.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.2
5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.
12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2021-09-09 17:12:26,297 INFO  balancer.Balancer (Balancer.java:parse(818)) - Using a
threshold of 10.0
2021-09-09 17:12:26,304 INFO  balancer.Balancer (Balancer.java:run(684)) - namenodes
 = [hdfs://li217.hde.com:8020]
2021-09-09 17:12:26,304 INFO  balancer.Balancer (Balancer.java:run(685)) - parameters
 = Balancer.BalancerParameters [BalancingPolicy.Node, threshold = 10.0, max idle iter
ation = 5, #excluded nodes = 0, #included nodes = 0, #source nodes = 0, #blockpools =
 0, run during upgrade = false]
2021-09-09 17:12:26,304 INFO  balancer.Balancer (Balancer.java:run(686)) - included n
odes = []
```

(2) 查看数据均衡效果

- 访问 HDFS 快速链接，查看数据均衡效果，详情请参见 3.9.1 1. (4)查看数据均衡效果。
- 在后台直接通过命令行，查看数据均衡效果，如下：

  hdfs dfsadmin -report

图3-37 通过命令行查看数据均衡效果

```
[hdfs@test2 ~]$ hdfs dfsadmin -report
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/sl
f4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org
/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Configured Capacity: 11985849311232 (10.90 TB)
Present Capacity: 10998247245623 (10.00 TB)
DFS Remaining: 9239432693560 (8.40 TB)
DFS Used: 1758814552063 (1.60 TB)
DFS Used%: 15.99%
Replicated Blocks:
        Under replicated blocks: 0
        Blocks with corrupt replicas: 0
        Missing blocks: 0
        Missing blocks (with replication factor 1): 0
        Low redundancy blocks with highest priority to recover: 0
        Pending deletion blocks: 0
Erasure Coded Block Groups:
        Low redundancy block groups: 0
        Block groups with corrupt internal blocks: 0
```

3.9.2 磁盘均衡

注意

- 磁盘均衡指在 HDFS 内各个 DataNode 节点上的不同磁盘之间进行数据均衡，此时不需要跨数据节点，仅需要在同一个 DataNode 节点上的不同磁盘之间进行数据迁移。
- 磁盘均衡为异步操作命令。磁盘均衡的命令下发给 HDFS 组件后，由 HDFS 自行判断该节点各磁盘中的数据是否需要均衡。只有当各磁盘间数据发生倾斜时才会真正执行磁盘数据均衡的操作，若磁盘数据状态是均衡的则不会再执行磁盘数据均衡的操作。
- 磁盘均衡在执行过程中会消耗集群资源（比如：磁盘 IO、网络带宽），影响集群的性能，建议在集群中无业务运行时执行此操作。
- 在数据量较大的情况下，磁盘均衡操作需要较长的时间才能执行完成，且需要长时间读写磁盘，会对集群性能造成影响。

磁盘均衡与数据均衡不同，磁盘均衡实现的是单一 DataNode 节点上不同磁盘之间数据的均衡。磁盘均衡通过一组命令的协同来实现，命令包括：

- hdfs diskbalancer - plan，plan 命令生成一个计划文件
- hdfs diskbalancer - execute，execute 命令会根据计划文件执行磁盘均衡
- hdfs diskbalancer - query，query 命令可以查看磁盘均衡执行的情况
- hdfs diskbalancer - cancel，cancel 可以取消正在执行的磁盘均衡操作

磁盘均衡功能是 HDFS 原生支持，该功能的操作包含两种方式：

- 在大数据平台管理系统直接通过页面操作进行磁盘均衡，操作方式请参见 3.9.2 1. 章节。
- 在后台也可直接使用一组命令行实现磁盘均衡，操作方式请参见 3.9.2 2. 章节。

1. 页面操作方式

在大数据平台管理系统，直接通过页面操作进行磁盘均衡的操作步骤如下：

(1) 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。

(2) 在集群详情页面选择[组件]页签，单击组件列表中 HDFS 名称进入组件详情页面，在部署拓扑页签的进程列表中单击某进程对应的<磁盘均衡>按钮，弹出磁盘均衡窗口，如图 3-38 所示。

参数说明如下：

- 均衡容忍百分比：DataNode 节点上的各个磁盘之间计划迁移的数据量目标值与实际迁移的数据量之间的差值除以计划迁移的数据量得到的百分比。

  【说明】此参数是磁盘均衡成功的判断条件。比如：若指定均衡容忍百分比为 10%，计划移动的数据量大小为 20GB，当时实际移动的数据量为 18GB 时即会认为此次磁盘均衡操作是成功的。

- 最大带宽：执行磁盘均衡操作时允许占用的最大带宽，带宽阈值的设置会影响磁盘间数据均衡的速率，单位 M/s。

图3-38 磁盘均衡窗口

(3) 均衡容忍百分比和最大带宽配置完成后，单击<确定>按钮即可对 HDFS 的该 DataNode 节点执行磁盘均衡操作。

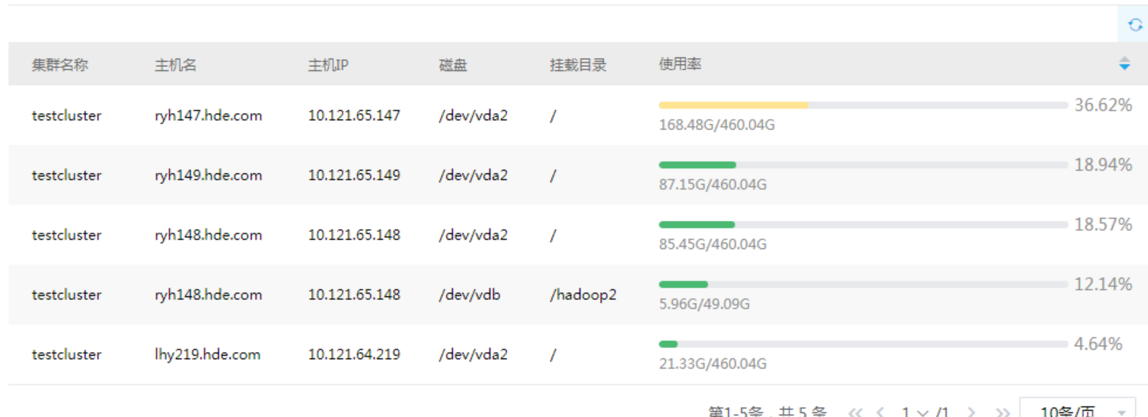
【说明】

- 磁盘均衡为异步操作，即单击<确定>按钮之后，磁盘均衡命令会被下发给 HDFS 组件，但 HDFS 组件会自行判断是否需要真正去执行磁盘均衡的操作。
- 单击<确定>按钮之后，查看集群操作记录，在操作记录窗口中查看到的磁盘均衡进度仅表示下发磁盘均衡命令的进度，并不代表该 DataNode 节点执行磁盘均衡的进度。

(4) 查看磁盘均衡效果

在[集群管理/主机管理/磁盘监控]页面，可以查看各主机上各个磁盘当前阶段的使用率，如图3-39 所示。

图3-39 磁盘监控信息



磁盘分区列表

集群名称	主机名	主机IP	磁盘	挂载目录	使用率
testcluster	ryh147.hde.com	10.121.65.147	/dev/vda2	/	36.62% 168.48G/460.04G
testcluster	ryh149.hde.com	10.121.65.149	/dev/vda2	/	18.94% 87.15G/460.04G
testcluster	ryh148.hde.com	10.121.65.148	/dev/vda2	/	18.57% 85.45G/460.04G
testcluster	ryh148.hde.com	10.121.65.148	/dev/vdb	/hadoop2	12.14% 5.96G/49.09G
testcluster	lhy219.hde.com	10.121.64.219	/dev/vda2	/	4.64% 21.33G/460.04G

第1-5条，共 5 条 1 /1 10条/页

2. 命令行操作方式

磁盘均衡通过一组命令的协同来实现，其中：plan 命令生成一个计划文件，execute 命令会根据计划文件执行磁盘均衡，query 命令可以查看磁盘均衡执行的情况，cancel 可以取消正在执行的磁盘均衡操作。

在后台直接使用一组命令行实现磁盘均衡的操作步骤如下：

(1) plan 命令生成一个计划文件

在待执行磁盘均衡操作的 DataNode 节点上运行 plan 命令，如下：

```
hdfs diskbalancer -plan <nodename>
```

其中：<nodename>表示 DataNode 节点的主机名，本章节截图中<nodename>示例为 test2.hde.com。

plan 命令执行之后会有两个输出文件：<nodename>.before.json（用来记录执行命令前的状态）、<nodename>.plan.json（执行磁盘均衡的计划文件）。

```
[hdfs@test2 ~]$ hdfs diskbalancer -plan test2.hde.com
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/sl
ticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org
StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2021-09-09 15:53:03,583 INFO  balancer.KeyManager (KeyManager.java:<init>(75)) - Block token param
rom NN: update interval=10hrs, 0sec, token lifetime=10hrs, 0sec
2021-09-09 15:53:03,595 INFO  block.BlockTokenSecretManager (BlockTokenSecretManager.java:addKeys(
ing block keys
2021-09-09 15:53:03,596 INFO  balancer.KeyManager (KeyManager.java:<init>(172)) - Update block key
, 30mins, 0sec
2021-09-09 15:53:03,987 INFO  planner.GreedyPlanner (GreedyPlanner.java:plan(70)) - Starting plan
est2.hde.com:8010
2021-09-09 15:53:03,990 INFO  planner.GreedyPlanner (GreedyPlanner.java:balanceVolumeSet(134)) - D
et fa594344-661f-418c-a440-35708ab2ccff Type : DISK plan completed.
2021-09-09 15:53:03,990 INFO  planner.GreedyPlanner (GreedyPlanner.java:plan(83)) - Compute Plan f
st2.hde.com:8010 took 5 ms
2021-09-09 15:53:04,202 INFO  command.Command (Command.java:recordOutput(496)) - Writing plan to:
2021-09-09 15:53:04,203 INFO  command.Command (Command.java:recordOutput(496)) - /system/diskbalan
-09-15-53-03/test2.hde.com.plan.json
```

(2) execute 命令会根据计划文件 nodename.plan.json 执行磁盘均衡操作

hdfs diskbalancer -execute /system/diskbalancer/xxx/<nodename>.plan.json

其中：/system/diskbalancer/xxx/<nodename>.plan.json 为运行 plan 命令时生成的计划文件默认的存储路径。

```
[hdfs@test2 ~]$ hdfs diskbalancer -execute /system/diskbalancer/2021-Sep-09-16-06-29/test2.hde.com
.plan.json
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/sl
f4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org
/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2021-09-09 16:42:45,836 INFO  command.Command (ExecuteCommand.java:execute(62)) - Executing "execu
te plan" command
```

(3) query 命令可以查看磁盘均衡执行的情况

hdfs diskbalancer -query <nodename>

```
[hdfs@test2 ~]$ hdfs diskbalancer -query test2.hde.com
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/sl
f4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org
/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2021-09-09 16:44:47,760 INFO  command.Command (QueryCommand.java:execute(58)) - Executing "query p
lan" command.
Plan File: /system/diskbalancer/2021-Sep-09-16-06-29/test2.hde.com.plan.json
Plan ID: 2b3f1a0c73745d42e74f289218b072353fabf37d
Result: PLAN_UNDER_PROGRESS
```

(4) cancel 可以取消该节点正在执行的磁盘均衡操作

hdfs diskbalancer -cancel /system/diskbalancer/xxx/<nodename>.plan.json

其中：/system/diskbalancer/xxx/<nodename>.plan.json 为运行 plan 命令时生成的计划文件默认的存储路径。

```
[hdfs@test2 ~]$ hdfs diskbalancer -cancel /system/diskbalancer/2021-Sep-09-16-06-29/test2.hde.com.
plan.json
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/sl
f4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org
/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2021-09-09 16:46:04,928 INFO  command.Command (CancelCommand.java:execute(60)) - Executing "Cancel
 plan" command.
```

3.10 HDFS分级存储

HDFS 分级存储（Archival Storage）是 Hadoop-2.6.0 之后新增的一个特性，是 Hadoop 异构存储中的一部分。HDFS 分级存储可实现将数据按照策略分开存储，解耦存储与计算能力，比如将部分冷数据归档存储至拥有廉价高密度存储介质但计算能力不强的机器上。

- 存储策略分为：HOT、COLD、WARM、ALL_SSD、ONE_SSD、LAZY_PERSIST 等。
- 存储类型分为：ARCHIVE、DISK、SSD、RAM_DISK。

3.10.1 存储策略与存储类型关系

存储策略允许不同的文件存储在不同的存储类型上，目前存储策略包括：

- HOT：存储与计算都热的数据，存储在 DISK。
- COLD：不再使用或需要归档的数据可移动到冷存储，用于有限计算，存储在 ARCHIVE。
- WARM：半冷半热的数据。一部分在热存储（DISK），其余的在冷存储（ARCHIVE）。
- ALL_SSD：所有数据都存储在 SSD。
- ONE_SSD：一个副本存储在 SSD，其他副本都存储在 DISK。
- LAZY_PERSIST：对于只有一个副本的块，存储在 RAM_DISK，然后会惰性的持久化到 DISK。但由于进程/节点的重启，文件随时可能丢失，因此该策略一般用于存储应用可恢复的中间数据。

表3-12 存储策略与存储类型对应表

策略名	块存储（n 副本）	用于创建的备份存储	用于复制的备用存储
Hot (default)	DISK: n	<none>	ARCHIVE
Cold	ARCHIVE: n	<none>	<none>
Warm	DISK: 1，ARCHIVE: n-1	ARCHIVE，DISK	ARCHIVE，DISK
All_SSD	SSD: n	DISK	DISK
One_SSD	SSD: 1，DISK: n-1	SSD，DISK	SSD，DISK
Lazy_Persist	RAM_DISK: 1，DISK: n-1	DISK	DISK

3.10.2 HDFS 存储策略命令

说明

以下命令均需要切换至具有操作 HDFS 文件系统权限的用户，例如可以执行 **su hdfs** 命令切换至 hdfs 用户下进行相关操作。详情可查看官网：https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html。

表3-13 存储策略操作命令

命令	格式	说明
hdfs storagepolicies -listPolicies	-	查询存储策略
hdfs storagepolicies -setStoragePolicy	-path <path> -policy <policy>	为文件/目录设置存储策略
hdfs storagepolicies -getStoragePolicy	-path <path>	获取文件/目录存储策略
hdfs storagepolicies -unsetStoragePolicy	-path <path>	取消文件/目录存储策略

3.10.3 存储类型设置

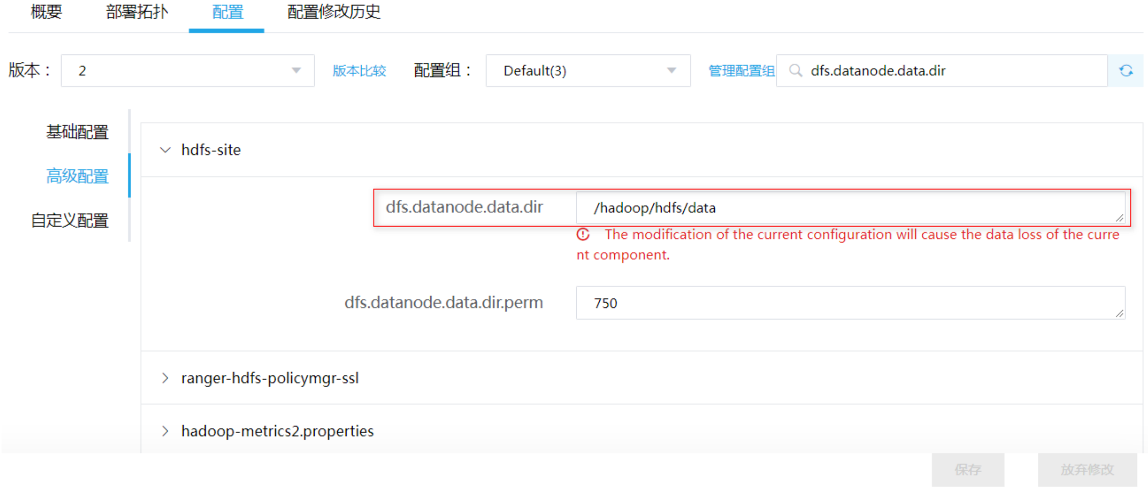
1. 修改存储类型

通过修改 HDFS 配置项 dfs.datanode.data.dir 的值可以更改数据存储类型，配置项修改完成后需根据界面提示重启 DataNode 或相关组件。

HDFS 配置项 dfs.datanode.data.dir 的值修改参考格式如下：

```
[DISK]/hadoop/hdfs/data1,[ARCHIVE]/hadoop/hdfs/data2,[SSD]/hadoop/hdfs/data3,
[SSD]/hadoop/hdfs/data4
```

图3-40 修改 HDFS 配置项 dfs.datanode.data.dir 的值



2. 操作示例

(1) 查询 HDFS 当前数据存储策略，命令如下：

hdfs storagepolicies -listPolicies

```
[hdfs@zrqnode1 root]$ hdfs storagepolicies -listPolicies
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerB
inder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLogg
erBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Block Storage Policies:
        BlockStoragePolicy{COLD:2, storageTypes=[ARCHIVE], creationFallbacks=[], replicationFallbacks=[]}
        BlockStoragePolicy{WARM:5, storageTypes=[DISK, ARCHIVE], creationFallbacks=[DISK, ARCHIVE], replicationFallbacks
=[DISK, ARCHIVE]}
        BlockStoragePolicy{HOT:7, storageTypes=[DISK], creationFallbacks=[], replicationFallbacks=[ARCHIVE]}
        BlockStoragePolicy{ONE_SSD:10, storageTypes=[SSD, DISK], creationFallbacks=[SSD, DISK], replicationFallbacks=[SS
D, DISK]}
        BlockStoragePolicy{ALL_SSD:12, storageTypes=[SSD], creationFallbacks=[DISK], replicationFallbacks=[DISK]}
        BlockStoragePolicy{LAZY_PERSIST:15, storageTypes=[RAM_DISK, DISK], creationFallbacks=[DISK], replicationFallback
s=[DISK]}
```

(2) 为目录设置新的数据存储策略，命令如下：

hdfs storagepolicies -setStoragePolicy -path /tmp/hot -policy HOT

```
[root@nodeq41 ~]# hdfs storagepolicies -setStoragePolicy -path /tmp/hot -policy HOT
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Set storage policy HOT on /tmp/hot
```

(3) 查询目录修改后的数据存储策略，命令如下：

hdfs storagepolicies -getStoragePolicy -path /tmp/hot/a.txt

```
[root@nodeq41 ~]# hdfs storagepolicies -getStoragePolicy -path /tmp/hot/a.txt
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/sl
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
The storage policy of /tmp/hot/a.txt:
BlockStoragePolicy{HOT:7, storageTypes=[DISK], creationFallbacks=[], replicationFallbacks=[ARCHIVE]}
```

3.11 HDFS纠删码

HDFS 通过多副本机制来保证数据的可靠性，即：在 HDFS 中每一份数据都有两个副本，因此 1TB 的原始数据需要占用 3TB 的磁盘空间，存储利用率只有 1/3。Hadoop 3.0 引入了纠删码技术（Erasure Coding），它可以提高 50%以上的存储利用率，并且仍能保证数据的可靠性。

纠删码技术（Erasure Coding，简称 EC）是一种编码容错技术，最早应用于通信行业数据传输过程中的数据恢复。它通过对数据进行分块，然后计算出校验数据，使得各个部分的数据产生关联性。当一部分数据块丢失时，可以通过剩余的数据块和校验块计算出丢失的数据块。由此可以看出，纠删码技术是一种使用计算与网络带宽换存储的策略，因此该技术一般用于冷数据集群。

纠删码策略包括：RS-10-4-1024k、RS-3-2-1024k、RS-6-3-1024k、RS-LEGACY-6-3-1024k、XOR-2-1-1024k、REPLICATION。

以 RS-6-3-1024k 为例进行说明：

- RS：表示 EC 码中的一种类型。
- 6-3：表示每 6 个数据单元，生成 3 个校验单元，这 9 个单元中只要有任意 6 个单元存在，即可保证数据完整性。注意：使用该策略要求集群至少有 9 个 DataNode。
- 1024k：表示每个数据单元的大小。

3.11.1 HDFS 纠删码命令

说明

以下相关操作均需要切换至 hdfs 用户执行，通过 ***su hdfs*** 可切换至 hdfs 用户。详情可查看官网：https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html。

表3-14 纠删码操作命令

命令	格式	说明
hdfs ec -listPolicies	-	查看hdfs注册的所有策略
hdfs ec -addPolicies	-policyFile <file>	添加用户自定义策略
hdfs ec -getPolicy	-path <path>	查看指定文件/目录策略
hdfs ec -removePolicy	-policy <policy>	删除用户自定义策略
hdfs ec -setPolicy	-path <path> [-policy <policy>] [-replicate]	在指定目录设置策略。只能使用已启用的策略进行设置
hdfs ec -unsetPolicy	-path <path>	取消指定目录策略
hdfs ec -listCodecs	-mv <源路经> ＜目的路径>	查看支持的编解码器
hdfs ec -enablePolicy	-policy <policy>	启用策略
hdfs ec -disablePolicy	-policy <policy>	禁用策略
hdfs ec -verifyClusterSetup	[-policy <policy>...<policy>]	验证集群是否支持所有已启用策略。若指定策略，则验证集群是否支持该策略

3.11.2 操作示例

(1) 通过 ***su hdfs*** 命令切换至 hdfs 用户

(2) 查询 HDFS 当前数据纠删码策略，命令如下：

```
hdfs ec -listPolicies
```

```
[hdfs@zrqnode1 root]$ hdfs ec -listPolicies
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerB
inder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLogg
erBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Erasure Coding Policies:
ErasureCodingPolicy=[Name=RS-10-4-1024k, Schema=[ECSchema=[Codec=rs, numDataUnits=10, numParityUnits=4]], CellSize=10485
76, Id=5], State=DISABLED
ErasureCodingPolicy=[Name=RS-3-2-1024k, Schema=[ECSchema=[Codec=rs, numDataUnits=3, numParityUnits=2]], CellSize=1048576
, Id=2], State=DISABLED
ErasureCodingPolicy=[Name=RS-6-3-1024k, Schema=[ECSchema=[Codec=rs, numDataUnits=6, numParityUnits=3]], CellSize=1048576
, Id=1], State=ENABLED
ErasureCodingPolicy=[Name=RS-LEGACY-6-3-1024k, Schema=[ECSchema=[Codec=rs-legacy, numDataUnits=6, numParityUnits=3]], Ce
llSize=1048576, Id=3], State=DISABLED
ErasureCodingPolicy=[Name=XOR-2-1-1024k, Schema=[ECSchema=[Codec=xor, numDataUnits=2, numParityUnits=1]], CellSize=10485
76, Id=4], State=DISABLED
```

(3) 启用 XOR-2-1-1024k 策略，命令如下：

hdfs ec -enablePolicy -policy XOR-2-1-1024k

```
[hdfs@nodeq41 ~]$ hdfs ec -enablePolicy -policy  XOR-2-1-1024k
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-imp
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Erasure coding policy XOR-2-1-1024k is enabled
```

(4) 为指定目录/tmp/xor 设置 XOR-2-1-1024k 策略，命令如下：

hdfs ec -setPolicy -path /tmp/xor -policy XOR-2-1-1024k

```
[hdfs@nodeq41 opt]$ hdfs ec -setPolicy -path /tmp/xor -policy XOR-2-1-1024k
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanat
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Set XOR-2-1-1024k erasure coding policy on /tmp/xor
```

(5) 上传文件查看 HDFS 前后占用存储大小：

a. 准备原始文件，约 8GB

b. 使用命令（hdfs dfs -put fileName /tmp/xor）将此文件上传至/tmp/xor 目录下

c. 上传前后分别查看 HDFS 文件系统的 UI 页面，上传前如图 3-41 所示，上传后如图 3-42 所示。可以发现：上传前 DFS Used 值为 890.32MB，上传一个约 8G 的文件（3 副本，占用空间约 24GB），则上传后 DFS Used 值理论应增加 24GB，但是开启 XOR-2-1-1024k 纠删码策略后实际却仅增加了 12GB 左右。相比之前传统的三副本，纠删码策略可使数据冗余率大幅度降低。

图3-41 上传文件前

Configured Capacity:	1.18 TB
DFS Used:	890.32 MB (0.07%)
Non DFS Used:	0 B
DFS Remaining:	1.11 TB (94.06%)
Block Pool Used:	890.32 MB (0.07%)
DataNodes usages% (Min/Median/Max/stdDev):	0.07% / 0.07% / 0.07% / 0.00%
Live Nodes	3 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	Mon Nov 25 18:44:49 +0800 2019
Last Checkpoint Time	Wed Nov 27 17:12:10 +0800 2019
Enabled Erasure Coding Policies	RS-6-3-1024k, XOR-2-1-1024k

图3-42 上传文件后

Configured Capacity:	1.18 TB
DFS Used:	13.44 GB (1.11%)
Non DFS Used:	0 B
DFS Remaining:	1.1 TB (93.01%)
Block Pool Used:	13.44 GB (1.11%)
DataNodes usages% (Min/Median/Max/stdDev):	1.11% / 1.11% / 1.11% / 0.00%
Live Nodes	3 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	Mon Nov 25 18:44:49 +0800 2019
Last Checkpoint Time	Wed Nov 27 17:12:10 +0800 2019
Enabled Erasure Coding Policies	RS-6-3-1024k, XOR-2-1-1024k

4 开发指南

4.1 API介绍

说明

关于 HDFS 接口更多信息，详情请参见官网 http://hadoop.apache.org/docs/r3.1.0/api/index.html。

4.1.1 HDFS 常用接口

(1) FileSystem：是客户端应用的核心类。

表4-1 类 FileSystem 常用接口说明

接口	说明
public static FileSystem get(Configuration conf)	Hadoop类库中最终面向用户提供的接口类是FileSystem，该类是个抽象类，只能通过该类的get方法得到具体类。get方法存在几个重载版本，常用的是这个。
public FSDataOutputStream create(Path f)	通过该接口可在HDFS上创建文件，其中f为文件的完整路径。
public void copyFromLocalFile(Path src, Path dst)	通过该接口可将本地文件上传到HDFS的指定位置上，其中src和dst均为文件的完整路径。
public boolean mkdirs(Path f)	通过该接口可在HDFS上创建文件夹，其中f为文件夹的完整路径。
public abstract boolean rename(Path src, Path dst)	通过该接口可为指定的HDFS文件重命名，其中src和dst均为文件的完整路径。
public abstract boolean delete(Path f, boolean recursive)	通过该接口可删除指定的HDFS文件，其中f为需要删除文件的完整路径，recuresive用来确定是否进行递归删除。
public boolean exists(Path f)	通过该接口可查看指定HDFS文件是否存在，其中f为文件的完整路径。
public FileStatus getFileStatus(Path f)	通过该接口可以获取文件或目录的FileStatus对象，该对象记录着该文件或目录的各种状态信息，其中包括修改时间、文件目录等等。
public BlockLocation[] getFileBlockLocations(FileStatus file, long start, long len)	通过该接口可查找指定文件在HDFS集群上块的位置，其中file为文件的完整路径，start和len来标识查找文件的块的范围。

(2) FileStatus：记录文件和目录的状态信息。

表4-2 类 FileStatus 常用接口说明

接口	说明
public long getModificationTime()	通过该接口可查看指定HDFS文件的修改时间。
public Path getPath()	通过该接口可查看指定HDFS中某个目录下的所有文件。

4.2 模块样例代码介绍

4.2.1 开发环境简介

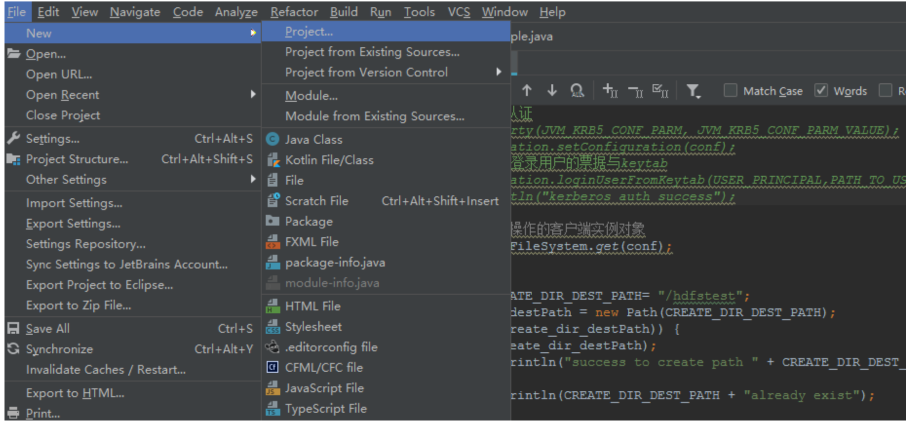
准备项	说明
操作系统	Windows系统，推荐Windows7以上版本
安装JDK	DE集群的服务端和客户端仅支持自带的Oracle JDK（版本为1.8），不允许替换
maven	/
IDEA	IDEA需整合maven环境

4.2.2 配置并导入样例代码

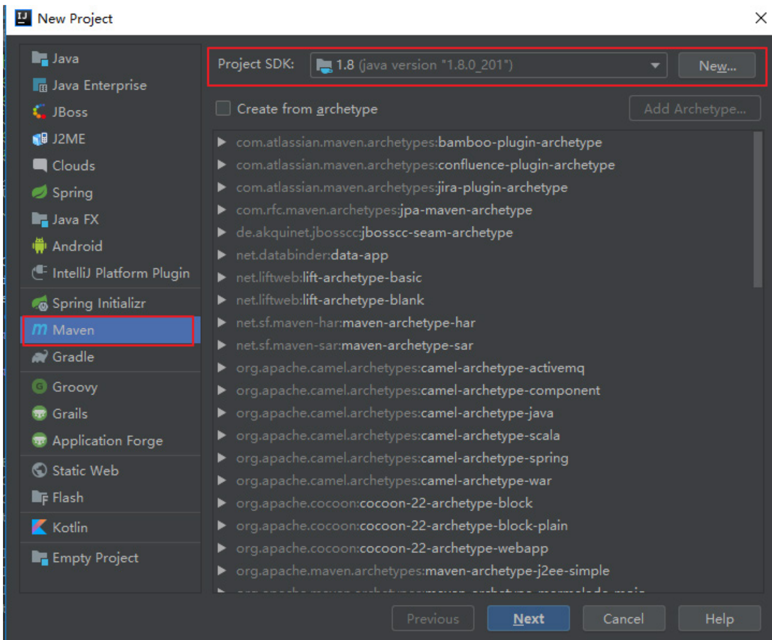
如果有样例工程，可以通过 IDEA 直接导入。如果想从头开始创建一个 hdfs 工程，操作步骤如下：

(1) 创建项目并选择 Maven。

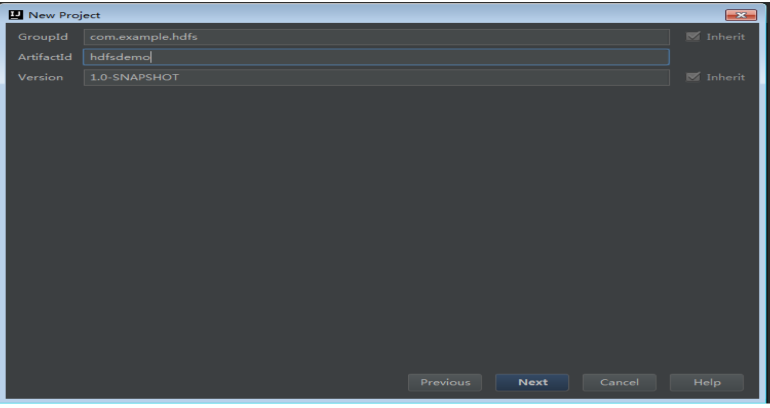
a. 创建项目



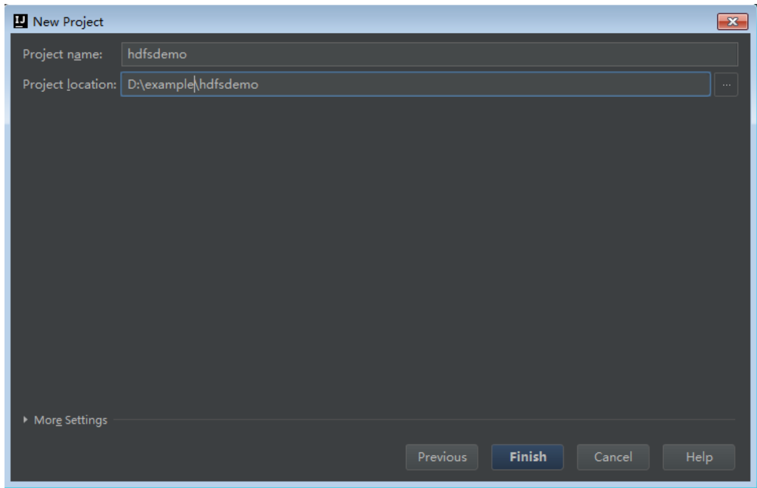
b. 选择 Maven



(2) 填写 GAV 信息



(3) 填写 projectname 并点击<Finish>按钮

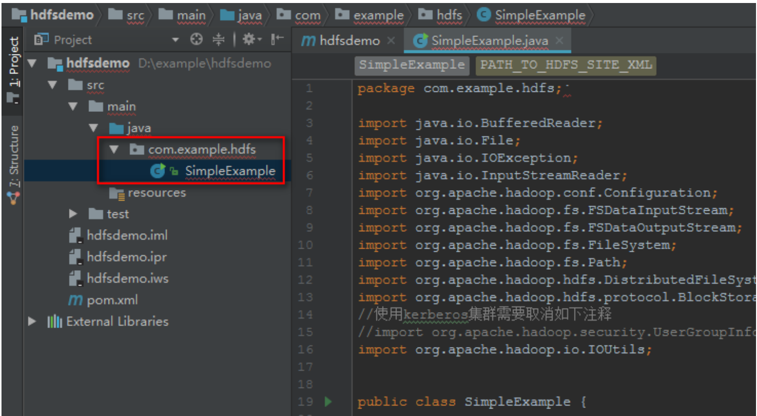


(4) 编辑 pom.xml 文件，引入相关依赖

```
<!--配置项目中某些 jar 包的版本-->
   <properties>
       <hadoop.version>3.1.0</hadoop.version>
   </properties>

   <dependencies>
       <dependency>
           <groupId>org.apache.hadoop</groupId>
           <artifactId>hadoop-common</artifactId>
           <version>${hadoop.version}</version>
       </dependency>
       <dependency>
           <groupId>org.apache.hadoop</groupId>
           <artifactId>hadoop-client</artifactId>
           <version>${hadoop.version}</version>
       </dependency>
       <dependency>
           <groupId>org.apache.hadoop</groupId>
           <artifactId>hadoop-hdfs</artifactId>
           <version>${hadoop.version}</version>
       </dependency>
   </dependencies>
```

(5) Java 目录新建 package（如：com.example.hdfs）和 class（如：SimpleExample.java）文件



说明：可以参见 4.2.3 ~4.2.12 章节来编写业务逻辑的相关代码。

4.2.3 HDFS 配置文件介绍

表4-3 配置文件

文件名称	作用	获取地址
core-site.xml	配置HDFS详细参数	/usr/hdp/current/hadoop-client/conf/core-site.xml
hdfs-site.xml	配置HDFS详细参数	/usr/hdp/current/hadoop-client/conf/hdfs-site.xml
user.keytab	对于Kerberos安全认证提供HDFS用户信息	可以联系管理员获取相应帐号对应权限的keytab文件和krb5文件
krb5.conf	Kerberos Server配置信息	

4.2.4 示例公共代码

```
package com.example.hdfs;
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hdfs.DistributedFileSystem;
import org.apache.hadoop.hdfs.protocol.BlockStoragePolicy;
//使用 kerberos 集群需要取消如下注释
//import org.apache.hadoop.security.UserGroupInformation;
import org.apache.hadoop.io.IOUtils;

public class SimpleExample {
    private final static String PATH_TO_HDFS_SITE_XML="/etc/hadoop/conf/hdfs-site.xml";
```

```
    private final static String PATH_TO_CORE_SITE_XML="/etc/hadoop/conf/core-site.xml";
    /**
     * 使用kerberos集群需要取消如下注释
     private final static String JVM_KRB5_CONF_PARM="java.security.krb5.conf";
     private final static String JVM_KRB5_CONF_PARM_VALUE="/etc/krb5.conf";
     private final static String USER_PRINCIPAL="diao2@HDE.COM";
     private final static String PATH_TO_USER_KEYTAB="/etc/security/keytabs/diao2.keytab";
     */

    public static void main(String[] args) throws Exception {
        //创建conf实例对象，并添加hdfs相关配置文件
        Configuration conf = new Configuration();
        conf.addResource(new Path(PATH_TO_HDFS_SITE_XML));
        conf.addResource(new Path(PATH_TO_CORE_SITE_XML));

        /**
         * 使用kerberos集群需要取消如下注释
         // kerberos身份认证
         System.setProperty(JVM_KRB5_CONF_PARM, JVM_KRB5_CONF_PARM_VALUE);
         UserGroupInformation.setConfiguration(conf);
         // 安全登录，指定登录用户的票据与keytab
         UserGroupInformation.loginUserFromKeytab(USER_PRINCIPAL,PATH_TO_USER_KEYTAB);
         System.out.println("kerberos auth success");
         */
        //拿到一个文件系统操作的客户端实例对象
        FileSystem fs = FileSystem.get(conf);
        //这里填写对目录或文件的操作代码，可参考章节4.2.5~4.2.12

        //关闭客户端
        fs.close();
    }
}
```

4.2.5 创建目录

1. 创建目录过程

(1) 调用FileSystem实例的exists方法查看创建的目录是否存在。

(2) 如果不存在，则调用FileSystem实例的mkdirs方法创建目录。

2. 代码样例

```
//创建目录
final String CREATE_DIR_DEST_PATH= "/hdfstest";
Path create_dir_destPath = new Path(CREATE_DIR_DEST_PATH);
if (!fs.exists(create_dir_destPath)) {
    fs.mkdirs(create_dir_destPath);
    System.out.println("success to create path " + CREATE_DIR_DEST_PATH);
}else{
    System.out.println(CREATE_DIR_DEST_PATH + "already exist");
}
```

4.2.6 写文件

新建文件，并写入数据。

1. 写文件过程

(1) 使用 FileSystem 实例的 create 方法获取写文件的输出流。

(2) 使用该输出流将内容写入到 HDFS 的指定文件中。

2. 代码示例

```
// 新建文件，并写入数据
final String FILE_CONTENT = "It is successful to create new file if you can see me";
final String CREATE_FILE_DEST_PATH="/hdfstest";
final String CREATE_FILE_NAME="writetest";
FSDataOutputStream create_file_out = null;
try {
    create_file_out = fs.create(new Path(CREATE_FILE_DEST_PATH + File.separator +
CREATE_FILE_NAME));
    create_file_out.write(FILE_CONTENT.getBytes());
    create_file_out.hsync();
    System.out.println("success to write.");
} finally {
    // make sure the stream is closed finally.
    IOUtils.closeStream(create_file_out);
}
```

4.2.7 文件追加内容

文件追加内容，是指在 HDFS 的某个指定文件后面，追加指定的内容。

1. 文件追加内容过程

(1) 使用 FileSystem 实例的 append 方法获取追加写入的输出流。

(2) 使用该输出流将待追加内容添加到 HDFS 的指定文件后面。

2. 代码示例

```
 // 向某个文件追加内容
final String APPEND_CONTENT = "I append this content.\n";
final String APPEND_DEST_PATH="/hdfstest";
final String APPEND_FILE_NAME="appendtest";
FSDataOutputStream append_out = null;
try {
    append_out = fs.append(new Path(APPEND_DEST_PATH + File.separator + APPEND_FILE_NAME));
    append_out.write(APPEND_CONTENT.getBytes());
    append_out.hsync();
    System.out.println("success to append.");
} finally {
    // make sure the stream is closed finally.
    IOUtils.closeStream(append_out);
}
```

4.2.8 读文件

获取 HDFS 上某个指定文件的内容

1. 读文件过程

(1) 使用 FileSystem 实例的 open 方法获取读取文件的输入流。

(2) 使用该输出流将内容写入到 HDFS 的指定文件中。

2. 代码示例

```
//读文件
final String READ_DEST_PATH="/hdfstest";
final String READ_FILE_NAME="readtest";
final String READ_STR_PATH = READ_DEST_PATH + File.separator + READ_FILE_NAME;
Path path = new Path(READ_STR_PATH);
FSDataInputStream in = null;
BufferedReader reader = null;
StringBuffer strBuffer = new StringBuffer();
try {
    in = fs.open(path);
    reader = new BufferedReader(new InputStreamReader(in));
    String sTempOneLine;
    // write file
    while ((sTempOneLine = reader.readLine()) != null) {
        strBuffer.append(sTempOneLine);
    }
    System.out.println("result is : " + strBuffer.toString());
    System.out.println("success to read.");
} finally {
    // make sure the streams are closed finally.
    IOUtils.closeStream(reader);
    IOUtils.closeStream(in);
}
```

4.2.9 删除文件

删除 HDFS 上某个指定文件。

说明

被删除的文件会被直接删除，无法恢复，请谨慎进行删除操作。

1. 代码示例

```
//删除文件
final String DEL_DEST_PATH="/hdfstest";
final String DEL_FILE_NAME="deltest";
Path beDeletedPath = new Path(DEL_DEST_PATH + File.separator + DEL_FILE_NAME);
if (fs.delete(beDeletedPath, true)) {
    System.out.println("success to delete the file " + DEL_DEST_PATH + File.separator +
DEL_FILE_NAME);
} else {
    System.out.println("failed to delete the file " + DEL_DEST_PATH + File.separator +
DEL_FILE_NAME);
}
```

4.2.10 删除目录

删除 HDFS 上某个指定目录。

说明

被删除的目录会被直接删除，且无法恢复。所以，执行删除操作时需谨慎。

1. 代码样例

```
//递归删除目录
final String DEL_RECU_DEST_PATH="/hdfstest";
Path del_recu_destPath = new Path(DEL_RECU_DEST_PATH);
if (!fs.exists(del_recu_destPath)) {
    System.out.println(DEL_RECU_DEST_PATH + "doesn't exist!");
    return;
}
if (!fs.delete(del_recu_destPath, true)) {
    System.out.println("failed to delete destPath " + DEL_RECU_DEST_PATH);
}
System.out.println("success to delete path " + DEL_RECU_DEST_PATH);
```

4.2.11 设置存储策略

为 HDFS 上某个文件或文件夹指定存储策略。

1. 设置存储策略过程

(1) 获取 hdfs 上支持的存储策略。

(2) 如果 hdfs 支持指定的存储策略，则将该路径设置为对应的存储策略，如果 hdfs 不支持，则设置为 hdfs 支持的某种存储策略。

2. 代码样例

```
//设置存储策略
final String SET_POLICY_DEST_PATH="/aaa";
String policyName="WARM";
if (fs instanceof DistributedFileSystem) {
    DistributedFileSystem dfs = (DistributedFileSystem) fs;
    Path set_policy_destPath = new Path(SET_POLICY_DEST_PATH);
    Boolean flag = false;
    BlockStoragePolicy[] storage = dfs.getStoragePolicies();

    for (BlockStoragePolicy bs : storage) {
        if (bs.getName().equals(policyName)) {
            flag = true;
        }
        System.out.println("StoragePolicy:" + bs.getName());
    }
    if (!flag) {
        policyName = storage[0].getName();
    }
    dfs.setStoragePolicy(set_policy_destPath, policyName);
    System.out.println("success to set Storage Policy path " + SET_POLICY_DEST_PATH);
} else {
    System.out.println("SmallFile not support to set Storage Policy !!!");
}
```

4.2.12 多线程任务

1. 功能简介

建立多线程任务，同时启动多个实例执行文件操作。

2. 代码样例

多线程实例增加的代码包括公共代码中填写的代码和在同文件下增加两个类的代码。

```
  //在公共代码中填写的代码
//多线程
final int THREAD_COUNT = 2;
for (int threadNum = 0; threadNum < THREAD_COUNT; threadNum++) {
    HdfsExampleThread example_thread = new HdfsExampleThread("hdfs_example_" + threadNum);
    example_thread.start();
}
//在同文件下增加两个类的代码
static class HdfsExampleThread extends Thread {

    public HdfsExampleThread(String threadName) {
        super(threadName);
    }
    public void run() {
        HdfsExample example;
        try {
            example = new HdfsExample("/user/hdfstest/" + getName() + "write.txt");
            example.write();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

static class HdfsExample {
    private final static String PATH_TO_HDFS_SITE_XML="/etc/hadoop/conf/hdfs-site.xml";
    private final static String PATH_TO_CORE_SITE_XML="/etc/hadoop/conf/core-site.xml";
    final String content = "It is successful to create new file if you can see me";
    String path="";

    public HdfsExample(String path) {
        this.path = path;
    }

    public void write() throws IOException{
        Configuration conf = new Configuration();
        conf.addResource(new Path(PATH_TO_HDFS_SITE_XML));
        conf.addResource(new Path(PATH_TO_CORE_SITE_XML));
        /**
         * 使用 kerberos 集群需要取消如下注释
         // kerberos 身份认证
         System.setProperty(JVM_KRB5_CONF_PARM, JVM_KRB5_CONF_PARM_VALUE);
         UserGroupInformation.setConfiguration(conf);
         // 安全登录，指定登录用户的票据与 keytab
         UserGroupInformation.loginUserFromKeytab(USER_PRINCIPAL,PATH_TO_USER_KEYTAB);
         System.out.println("kerberos auth success");
         */
        //拿到一个文件系统操作的客户端实例对象
```

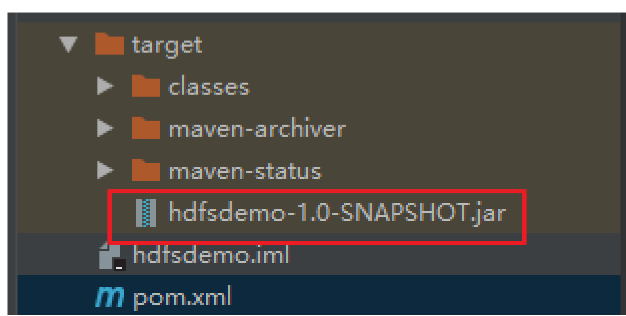
```
        FileSystem fs = FileSystem.get(conf);
        FSDataOutputStream out = null;
        try {
            out = fs.create(new Path(path));
            out.write(content.getBytes());
            out.hsync();
            System.out.println("success to write.");
        } finally {
            // make sure the stream is closed finally.
            IOUtils.closeStream(out);
        }
        //关闭客户端
        fs.close();
    }
}
```

4.2.13 编译并运行程序

(1) 用 IDEA 打开 terminal 窗口并输入如下命令编译打包。

mvn clean package

(2) 编译成功之后可以在工程目录下看到 target 文件夹和 jar 包。



(3) 将 hdfsdemo-1.0-SNAPSHOT.jar 通过工具上传到集群节点某个目录下，如 /home/hdfsdemo。

(4) 通过如下命令运行 jar 包。

yarn jar /home/hdfsdemo/hdfsdemo-1.0-SNAPSHOT.jar com.example.hdfs.SimpleExample

(5) 程序运行过程中，会打印输出相关信息，可以通过 hdfs shell 命令查看运行结果是否正确。如运行创建/hdfstest 目录代码之后，可通过“hdfs dfs -ls /”命令来查看是否创建成功。

```
drwxr-xr-x   - hdfs supergroup          0 2019-03-24 19:44 /aaa
drwxr-xr-x   - hdfs supergroup          0 2019-03-25 02:40 /hdfstest
drwxr-xr-x   - hdfs supergroup          0 2019-03-23 19:13 /input
drwxr-xr-x   - hdfs supergroup          0 2018-10-26 18:39 /test
drwx------   - hdfs supergroup          0 2019-03-24 19:57 /tmp
drwxr-xr-x   - hdfs supergroup          0 2019-03-25 02:03 /user
drwxr-xr-x   - hdfs supergroup          0 2018-09-11 23:01 /wordcount
```

5 最佳实践

5.1 数据迁移

DistCp（分布式拷贝）是大规模集群内部和集群之间进行数据拷贝的工具。它使用 Map/Reduce 进行文件分发、错误处理和恢复以及报告生成等操作，把文件和目录的列表作为 map 任务的输入，每个任务会完成源列表中部分文件的拷贝。

数据迁移任务为一次性全量数据备份任务，适用于数据完整迁移的场景。

5.1.1 迁移操作示例

说明

- 源集群为数据输出集群，目标集群为数据输入集群。
- 本章节示例是将 nn1 源集群中的数据迁移至 nn2 目标集群上，以下操作均在 nn2 集群上执行。

1. 源集群和目标集群均开启 Kerberos 认证时

数据迁移步骤如下：

(1) 配置源集群和目标集群互信，配置详情请参见 3.8.2 1. 开启 kerberos 认证的环境。

(2) Kerberos 环境下，执行数据迁移操作之前，需要首先准备一个通过 Kerberos 认证的用户（比如：集群超级用户、组件用户等）。

(3) 执行数据迁移操作（3 种数据迁移方式），命令如下：

- 集群之间的迁移（以下示例是将 nn1 集群中的数据迁移至 nn2 集群上）

  ```
  hadoop distcp hdfs://nn1:8020/foo/bar  hdfs://nn2:8020/bar/foo
  ```

  将 nn1 集群的/foo/bar 目录下的所有文件或目录名展开并存储到一个临时文件中，这些文件内容的拷贝工作被分配给多个 map 任务，然后每个 NodeManager 分别执行从集群 nn1 到 nn2 的拷贝操作。

  【说明】：distcp 需要使用绝对路径进行操作。

- 指定多个源目录

  ```
  hadoop distcp hdfs://nn1:8020/foo/a hdfs://nn1:8020/foo/b hdfs://nn2:8020/bar/foo
  ```

- 从文件里获取多个源

  ```
  hadoop distcp -f hdfs://nn1:8020/srclist  hdfs://nn2:8020/bar/foo
  ```

  其中，srclist 的内容是：

  ```
  hdfs://nn1:8020/foo/a
  hdfs://nn1:8020/foo/b
  ```

说明

不同 Hadoop 版本之间的数据迁移时，操作方式与上面类似。但是对于不同 Hadoop 版本间的拷贝，应该使用 HftpFileSystem。HftpFileSystem 是一个只读文件系统，所以 distcp 必须运行在目标端集群上。源的格式是 hftp://<dfs.http.address>/<path>，（默认情况下，dfs.http.address 是 <namenode>:50070）。

(4) 迁移完成后，建议生成源端和目的端文件的列表，交叉检查，以确认拷贝是否成功。

2. 源集群和目标集群均未开启 Kerberos 认证时

数据迁移步骤如下：

(1) 配置源集群和目标集群互信，配置详情请参见 3.8.2 2. 不开 kerberos 认证的环境。

(2) 执行数据迁移操作（3 种数据迁移方式），命令如下：

- 集群之间的迁移（以下示例是将 nn1 集群中的数据迁移至 nn2 集群上）

  ```
  hadoop distcp hdfs://nn1:8020/foo/bar  hdfs://nn2:8020/bar/foo
  ```

  将 nn1 集群的/foo/bar 目录下的所有文件或目录名展开并存储到一个临时文件中，这些文件内容的拷贝工作被分配给多个 map 任务，然后每个 NodeManager 分别执行从集群 nn1 到 nn2 的拷贝操作。

  【说明】：distcp 需要使用绝对路径进行操作。

- 指定多个源目录

  ```
  hadoop distcp hdfs://nn1:8020/foo/a hdfs://nn1:8020/foo/b hdfs://nn2:8020/bar/foo
  ```

- 从文件里获取多个源

  ```
  hadoop distcp -f hdfs://nn1:8020/srclist  hdfs://nn2:8020/bar/foo
  ```

  其中，srclist 的内容是：

  ```
  hdfs://nn1:8020/foo/a
  hdfs://nn1:8020/foo/b
  ```

说明

不同 hadoop 版本之间的数据迁移时，操作方式与上面类似。但是对于不同 Hadoop 版本间的拷贝，应该使用 HftpFileSystem。HftpFileSystem 是一个只读文件系统，所以 distcp 必须运行在目标端集群上。源的格式是 hftp://<dfs.http.address>/<path>，（默认情况下，dfs.http.address 是 <namenode>:50070）。

(3) 迁移完成后，建议生成源端和目的端文件的列表，交叉检查，以确认拷贝是否成功。

3. 源集群和目标集群一个开启 Kerberos 认证一个未开启 Kerberos 认证时

说明

- Kerberos 与非 Kerberos 集群间进行数据迁移时，迁移命令都必须在 Kerberos 集群节点执行。
- 本章节示例是将未开启 Kerberos 认证的 nn1 源集群中的数据迁移至开启 Kerberos 认证的 nn2 目标集群上，以下操作均在 nn2 集群上执行。

数据迁移步骤如下：

(1) 配置源集群和目标集群互信，如下：

  a. 检查并修改源集群和目标集群的时区、时间均同步。

  b. 修改源集群和目标集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目标集群的所有节点信息（互相拷贝即可）。

  【说明】可先修改源集群或目标集群中的某一个节点的/etc/hosts 文件，然后通过批量拷贝的方式实现所有节点的/etc/hosts 文件修改。

(2) Kerberos 环境下，执行数据迁移操作之前，需要首先准备一个通过 Kerberos 认证的用户（比如：集群超级用户、组件用户等）。

(3) 执行数据迁移操作（3 种数据迁移方式），命令如下：

【说明】此时，迁移命令需要增加配置“-D ipc.client.fallback-to-simple-auth-allowed=true”。

- 集群之间的迁移（以下示例是将 nn1 集群中的数据迁移至 nn2 集群上）

```
hadoop distcp -D ipc.client.fallback-to-simple-auth-allowed=true hdfs://nn1:8020/foo/bar
hdfs://nn2:8020/bar/foo
```

  将 nn1 集群的/foo/bar 目录下的所有文件或目录名展开并存储到一个临时文件中，这些文件内容的拷贝工作被分配给多个 map 任务，然后每个 NodeManager 分别执行从集群 nn1 到 nn2 的拷贝操作。

  【说明】：distcp 需要使用绝对路径进行操作。

- 指定多个源目录

```
hadoop distcp -D ipc.client.fallback-to-simple-auth-allowed=true hdfs://nn1:8020/foo/a
hdfs://nn1:8020/foo/b hdfs://nn2:8020/bar/foo
```

- 从文件里获取多个源

```
hadoop distcp -D ipc.client.fallback-to-simple-auth-allowed=true -f hdfs://nn1:8020/srclist
hdfs://nn2:8020/bar/foo
```

  其中，srclist 的内容是：

```
hdfs://nn1:8020/foo/a
hdfs://nn1:8020/foo/b
```

说明

不同 Hadoop 版本之间的数据迁移时，操作方式与上面类似。但是对于不同 Hadoop 版本间的拷贝，应该使用 HftpFileSystem。HftpFileSystem 是一个只读文件系统，所以 distcp 必须运行在目标端集群上。源的格式是 hftp://<dfs.http.address>/<path>，（默认情况下，dfs.http.address 是 <namenode>:50070）。

(4) 迁移完成后，建议生成源端和目的端文件的列表，交叉检查，以确认拷贝是否成功。

5.2 目录离线备份

目录离线备份是将 HDFS 上的数据备份到本地磁盘，如果 HDFS 数据丢失，可以再将备份到本地的数据上传到 HDFS，以达到恢复数据的目的。

如图 5-1 所示，使用 ***get*** 命令将 HDFS 的/2G 目录下的内容备份到本地/tmp/2G_bak 目录下，之后可以使用 ***put*** 命令将备份的数据上传到 HDFS 的/2G_new 目录下。

图5-1 目录离线备份示例

```
[root@node1 opt]# mkdir -p /tmp/2G_bak
[root@node1 opt]# chmod -R 777 /tmp/2G_bak
[root@node1 opt]# su hdfs
sh-4.1$ hdfs dfs -get /2G/*  /tmp/2G_bak/
sh-4.1$ ls /tmp/2G_bak/
call_center      catalog_sales     customer_demographics  household_demograph
catalog_page     customer          date_dim               income_band
catalog_returns  customer_address  dbgen_version          inventory
sh-4.1$ hdfs dfs -mkdir /2G_new
sh-4.1$ hdfs dfs -put /tmp/2G_bak/*  /2G_new/
sh-4.1$ hdfs dfs -ls /tmp/2G_new/
ls: `/tmp/2G_new/': No such file or directory
sh-4.1$ hdfs dfs -ls /2G_new/
Found 26 items
-rw-------   3 hdfs hdfs          0 2018-09-19 15:45 /2G_new/_SUCCESS
drwx-----x   - hdfs hdfs          0 2018-09-19 15:45 /2G_new/call_center
drwx-----x   - hdfs hdfs          0 2018-09-19 15:45 /2G_new/catalog_page
drwx-----x   - hdfs hdfs          0 2018-09-19 15:45 /2G_new/catalog_returns
```

5.3 全量离线备份

说明

建议进行离线备份之前先停止 HDFS。

全量离线备份可以将 NameNode 保存的元数据信息和 DataNode 保存的文件数据块拷贝到备份目录，实现全量离线备份。

${HADOOP_HOME}/conf/hdfs-site.xml 配置项：

- dfs.namenode.name.dir：设置 NameNode 保存元数据信息的本地路径。
- dfs.datanode.data.dir：设置 DataNode 节点存储文件数据块的本地路径。
- 在所有 NameNode 节点备份数据，此操作前建议停止 HDFS。

```
[root@node1 ~]# mkdir /opt/hdfs_bak
```

```
[root@node1 ~]# mkdir /opt/hdfs_bak/hdfs_namenode_bak
[root@node1 ~]# mkdir /opt/hdfs_bak/hdfs_datanode_bak
[root@node1 ~]# cp -a ${dfs.namenode.name.dir} /opt/hdfs_bak/hdfs_namenode_bak
[root@node1 ~]# cp -a ${dfs.datanode.data.dir} /opt/hdfs_bak/hdfs_datanode_bak
```

- 在所有 DataNode 节点上备份数据。

```
[root@node1 ~]# mkdir /opt/hdfs_bak
[root@node1 ~]# mkdir /opt/hdfs_bak/hdfs_datanode_bak
[root@node1 ~]# cp -a ${dfs.datanode.data.dir} /opt/hdfs_bak/hdfs_datanode_bak/
```

- 在所有 NameNode 节点恢复数据。

```
[root@node1 ~]# cp -a /opt/hdfs_bak/hdfs_namenode_bak/*  ${dfs.namenode.name.dir}
[root@node1 ~]# cp -a /opt/hdfs_bak/hdfs_datanode_bak/*  ${dfs.datanode.data.dir}
```

- 在所有 DataNode 节点恢复数据。

```
[root@node1 ~]# cp -a /opt/hdfs_bak/hdfs_datanode_bak/* ${dfs.datanode.data.dir}
```

6 常见问题解答

6.1 调优

1. 作业参数调优-小文件存储优化

小文件是指小于 HDFS 系统 Block 大小的文件（Hadoop 3.1.0 版本默认 128M），由于 HDFS 实现机制，每一个存储在 HDFS 中的文件、目录和块映射为一个对象存储在 NameNode 服务器内存中，因此，HDFS 不适合存储大量的小文件。

Hadoop 提供了两种机制来解决小文件存储问题，包括 HAR files 和 SequeueFile。其中：

- HRA files：可以将众多小文件打包成一个大文件进行存储，并且打包后原来的文件仍然可以通过 Map-reduce 进行操作。但 HAR files 一旦创建就不能修改，要做增加和修改文件必须重新打包。
- Sequence file：由一系列的二进制 key/value 组成，如果 key 为小文件名，value 为文件内容，则可以将大批小文件合并成一个大文件。

2. 参数调优

- dfs.blocksize

  HDFS 作为 MapReduce 的基础分布式文件系统，会直接影响 MapReduce 的运行效果。在网络环境很好的集群中，建议将 dfs.blocksize 参数提升，大小可以到 256M 或更大（hadoop 3.1.0 版本默认 128M）。
- dfs.namenode.handler.count

  将该参数大小设置为 20logN，N 为集群大小。如果该值设的太小，有可能导致 DataNode 连接 NameNode 的时候总是超时或者连接被拒，如果该值太大，会造成 NameNode 的远程过程调用队列很大，远程过程调用延时就会加大。在查找故障时，检查该参数值是很必要的。
- dfs.datanode.failed.volumes.tolerated

  该参数默认值为 0，意味着只要有一个磁盘出现故障就会导致整个 DataNode 不可用，管理员可以通过增大该值来保证在出现部分磁盘故障时，该节点的 DataNode 仍能持续运行。

6.2 通用类

1. HDFS 长时间处于安全模式，如何解决？

可能原因：

- 存储文件出现故障
- DataNode 的磁盘已满

解决方法：

- 管理员可以通过 Hadoop dfsadmin –safemode leave 命令强制离开，若强制离开后 NameNode 很快又进入 safemode。可以使用 hdfs fsck / 检测文件健康状态，并使用 hdfs fsck /path -delete 删除损坏文件。
- 若磁盘空间不足，则需要扩容。

2. HDFS 一直异常，如何解决？

可能原因：

- HDFS 进入了安全模式
- HDFS 依赖的 ZooKeeper 异常

解决方法：

- 若 HDFS 是否处于安全模式，直接退出安全模式。
- 在 Zookeeper 组件详情页面查看 Zookeeper 状态，例如：
    - 若 ZooKeeper 存在告警，可根据告警提示进行修复。
    - 若 ZooKeeper 状态正常，可尝试重启 HDFS。
    - 若重启后组件依然异常，检测 NameNode 是否故障，并尝试修复。

3. 由于 DataNode 副本不可用导致文件写入操作失败，如何解决？

可能原因：

- DataNode 的数据接收器不可用
- DataNode 的磁盘空间不足
- DataNode 的心跳有延迟

解决方法：

- 查看 DataNode 日志，如果日志中有“java.io.IOException: Xceiver count 8193 exceeds the limit of concurrent xcievers:8192”，表示 DataNode 的数据接收器不可用。
- 在组件详情页面，修改参数 dfs.datanode.max.transfer.threads 的值，要求该值大于 8192，例如 16384。
- 如果没有足够的硬盘空间或者 CPU，尝试增加新的数据节点或确保资源是可用的。
- 检查网络，以确保网络是可用的。

4. Namenode 一致处于加载 fsimage 过程，无法正常启动，如何解决？

可能原因：

- 该问题通常发生在 hdfs 数据块多的时候，一般是小文件过多造成 fsimage 太大，NameNode 的内存小于 fsimage 的大小导致。

解决方法：

- 检查 fsimage 大小，并检查 NameNode 内存配置。
- 通过 HDFS 配置界面调整 NameNode 内存大小。

5. NameNode 启动过程一致处于 safemode 模式，如何解决？

可能原因：

- 该阶段是 DataNode 向 NameNode 上报数据阶段，DE 默认超过 20 分钟如果还未结束，即视为无法退出 safemode，该现象是 HDFS 有数据坏块导致，这些数据坏块会显示在 HDFS 的 UI 界面最上部分。

解决方法：

- 大数据集群默认启动 20 分钟后就不再启动 NameNode，之后检查 HDFS 是否有坏块。

- 查看 HDFS 的 UI 上显示哪些坏块，并找到公共路径 path，然后切换到 hdfs 用户，执行 hdfs dfsadmin -safemode leave 强制退出安全模式，然后执行 hdfs fsck /path -delete 把这些坏的数据块删除。

6. DataNode 数据目录配置错误导致根目录写满，要求数据迁移保证不能丢失数据，如何解决？

可能原因：

- 目录写满了，要把数据迁移到其它地方。

解决方法：

- 对 DataNode 逐个进行如下操作进行数据迁移：手动 kill DataNode 进程，修改 hdfs-site.xml 数据目录，用 hdfs 用户在正常数据目录下创建 hdfs/data 目录，并按照原始目录结构复制数据到该目录下。命令启动 DataNode，通过 hadoop fsck /观察副本数和是否丢块，删除根目录下 hdfs 数据。

7. HDFS 频繁丢块且处于 safemode 模式，如何解决？

可能原因：

- 元数据过大导致。

解决方法：

- 在对 NameNode 的 fsimage 文件分析时，发现 ui 界面上 hdfs 的所有组件都正常工作，然后开始排查日志，在去 secondary namenode 节点查看元数据时，发现该目录下的文件日期都是 3 月 10 日左右的数据，且该目录已满，然后继续观察日志，发现从去年开始就一直报异常，时间和 hdfs 去年频繁异常的时间也基本吻合。
- 由于 secondarynamenode 挂载的目录最大只有 195M，就在页面更改了挂载路径，然后把数据复制到新路径，最后启动 hdfs。

8. HDFS DataNode 无法启动，同时 Hive、HBase、Spark、Spark2、MapReduce2 也无法启动，如何解决？

可能原因：

- dfs.datanode.data.dir 只配了一个目录。
- 配置内存大于主机剩余内存。
- 数据盘有一块盘出现故障。

解决方法：

- 排查 DataNode 日志及 hdfs 配置项，发现 dfs.datanode.data.dir 只配了一个目录，并且 dfs.datanode.failed.volumes.tolerated 设置成了 1。只有一个目录，还允许一个目录不可用。
- 查看日志是否有 insufficient memory。
- 1 个数据盘坏了，hdfs 配置的是 disk_fault_tolerance 为 0，即不允许坏盘。
- dfs.datanode.failed.volumes.tolerated 配置为 0；或给 hdfs 多分几个数据盘。
- 调小堆内存配置。
- 调大允许坏盘数，恢复集群，然后与一线沟通更换磁盘。

9. HDFS 有部分块一直处于复制中，如何解决？

可能原因：

- HDFS 上面一部分块的个数超过了 HDFS 上 DataNode 的个数，导致副本没有地方放置，所以一直处于复制中。

解决方法：

- 排查 DataNode 日志：Replicas is 10 but found 6 live replica(s)。
- 查看 hdfs 的副本配置是几个，采用 hdfs dfs -setrep 3 / 将对应目录重置为 3（假设副本数配置为 3，对应目录为根目录）。
- 另外检查 MapReduce2 组件的 mapred-site 配置文件的 mapreduce.client.submit.file.replication 配置参数，若没有配置，则在 mapred-site 的自定义配置中增加且值配置为 3（默认为 10，若集群数据节点少于 10 个时容易出现部分数据块一直处于复制中）。

10. HDFS NameNode 节点启动失败，如何解决？

可能原因：

- 元信息损坏错误。

解决方法：

- 查看 NameNode 日志看是否有元信息错误的报错，例如：
  - Faimage 镜像文件损坏。
  - Editlog 日志错误，如“EditLogInputException：Error replaying edit log at offset 0”等。
- 利用 hadoop namenode -recover 修复元数据，若仍无法恢复，请联系技术支持工程师。

11. HDFS 无法远程 put 数据，如何解决？

可能原因：

- 指定的端口错误。

解决方法：

- 查看客户的使用方式，文件系统端口为 8020。
- 远程 put 命令：hdfs dfs -put test.txt hdfs://ip:8020/usr。

12. HDFS 两个 NameNode 都处于 Standby 的状态，均无法转为 Active，如何解决？

可能原因：

- Zookeeper 或 ZKFC 异常。
- 元信息错误。

解决方法：

- 查看集群 Zookeeper 是否正常（至少半数以上为启动状态），然后查看 ZKFC 启动是否正常，后台日志是否有报错信息。
- 若 Zookeeper 与 ZKFC 都正常，可以查看 NameNode 日志看是否有元信息错误的报错。若是元数据的问题，可以利用 hadoop namenode -recover 修复元数据。
- 若元数据问题修复后依然没有选出 Active NameNode，可以手动停掉其中一个 ZKFC 的进程，回避竞争，这样另一个节点的 NameNode 就会被自动选举为 Active。若几分钟内仍未选举成功，可执行 hdfs zkfc -formatZK -force 重新选举。

- 若以上方案均无法恢复，请联系技术支持工程师解决。

13. HDFS DataNode 状态正常，却在 excludedNodes 中无法写入数据，如何解决？

可能原因：

- 在 HDFS Client 进行大量数据写入时，由于网络、超时、DataNode 状态异常等原因导致建立 DataNode 连接失败，将会把 DataNode 放入 excludedNodes 缓存中（DataNode 添加后会在 excludedNodes 中保留默认 10 分钟），下次 Client 重试连接时将会连接其他的 DataNode。所以若在 10 分钟内，HDFS 的所有 DataNode 都加入到了 excludedNodes 缓存中，没有可用 DataNode 供 Client 连接时，则会导致写入数据失败。

解决方法：

- 将 excludedNodes 缓存默认时间改小或禁用掉，即在 hdfs-site.xml 中增加或修改配置项 dfs.client.write.exclude.nodes.cache.expiry.interval.millis 的值。当 dfs.client.write.exclude.nodes.cache.expiry.interval.millis=0（单位为毫秒），即配置项的值为 0 时就表示禁用。
- 另外检查 DataNode 的磁盘空间剩余是否已达到预留大小（磁盘预留大小可以通过 HDFS 组件的配置 hdfs-site 中的 dfs.datanode.du.reserved 参数获取，单位：字节），若达到预留大小则需要扩容或清理历史数据。

14. HDFS 组件异常，JournalNode 日志中报元数据的错误：WARN namenode.FSImage (FSEditLogLoader.java:scanEditLog(1252)) - Caught exception after scanning through 0 ops from /hadoop/hdfs/journal/mycluster/current/edits_inprogress_0000000000000006594 while determining its valid length. Position was 1048576 java.io.IOException: Can't scan a pre-transactional edit log.

可能原因：

- 服务器断电重启时，某些节点上 HDFS 的元数据同步有问题，造成各个节点上的元数据的不一致。

解决方法：

(1) 查看是否有正常运行的 journalNode 节点，或者登录每个 JournalNode 节点上找到最新的元数据文件所在的节点，手动将元数据文件拷贝到其他节点上，即手动同步下元数据。元数据文件所在的路径可以根据 hdfs-site 文件中配置项 dfs.journalnode.edits.dir 获取。

(2) 重启下 HDFS 组件。

(3) 如果 HDFS 组件仍异常，请联系技术支持工程师解决。

15. HDFS Client 进行大量数据写入时出现“There are 10 datanode(s) running and 10 node(s) are excluded in this operation”类似的问题，DataNode 状态为正常运行中

可能原因：

- 在 HDFS Client 进行大量数据写入时，出现“There are 10 datanode(s) running and 10 node(s) are excluded in this operation”类似的问题，DataNode 状态为正常运行中。

解决方法：

- 偶现问题。可以在 hdfs-site.xml 中增加配置 dfs.client.write.exclude.nodes.cache.expiry.interval.millis=0 来处理。

16. HDFS 出现块丢失或损坏问题

可能原因：

- 集群多个主机同时断电时可能导致 HDFS 中部分块丢失或损坏。

解决方法：

- 偶现问题，解决步骤如下：
  a. 在集群内切换至用户 hdfs。
  b. 查看损坏的文件块信息，执行 hdfs fsck / -list-corruptfileblocks 获取损坏文件块的位置。
  c. 根据损坏文件块的位置（例如：/tmp/test.txt），执行 hdfs debug recoverLease -path /tmp/test.txt -retries 5 进行手动恢复。若尝试多次仍无法恢复，此时可以使用 hdfs fsck /tmp/test.txt -delete 删除损坏文件以恢复 HDFS 的健康状态。

17. 执行 hadoop distcp 跨集群拷贝 hdfs 文件报错：YarnException: Failed to submit application_xxx，Failed to renew token

可能原因:

- 用于身份验证的令牌不能更新导致执行任务失败。

解决方法：

- 通过增加参数-Dmapreduce.job.hdfs-servers.token-renewal.exclude 来解决，例如：

  hadoop distcp -Dmapreduce.job.hdfs-servers.token-renewal.exclude="192.1.1.1,192.2.2.2" hdfs://192.1.1.1:8020/t01 hdfs://192.2.2.2:8020/tmp

目 录

1 组件简介 1-1
1.1 YARN 1-1
1.1.1 YARN 概述 1-1
1.1.2 YARN 架构 1-1
1.1.3 YARN 工作步骤 1-2
1.1.4 YARN 特点 1-3
1.2 MapReduce 1-3
1.2.1 MapReduce 概述 1-3
1.2.2 MapReduce 基本概念 1-3
1.2.3 MapReduce 特点 1-4
1.2.4 MapReduce 应用程序运行过程 1-4
1.2.5 MapReduce 应用场景 1-5
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 数据目录检查 2-1
2.1.2 查看组件的日志路径 2-2
2.2 运行状态监控 2-2
2.3 快速使用指导 2-6
2.3.1 非 kerberos 环境 2-6
2.3.2 Kerberos 环境 2-7
2.4 快速链接 2-9
2.4.1 配置组件快速链接 2-9
2.4.2 访问 MapReduce 快速链接 2-9
2.4.3 访问 YARN 快速链接 2-10
3 使用指南 3-1
3.1 YARN Shell 命令 3-1
3.2 Client 下载/安装/使用/卸载 3-2
3.2.1 下载 Client 安装包 3-2
3.2.2 安装 Client 3-3
3.2.3 访问组件 3-4
3.2.4 卸载 Client 客户端 3-5
3.3 YARN 刷新队列 3-6

3.4 YARN 集群扩容 3-6
3.4.1 使用场景 3-6
3.4.2 扩容前准备 3-6
3.4.3 扩容约束 3-7
3.4.4 扩容影响 3-7
3.4.5 扩容操作指导 3-7
3.4.6 扩容验证 3-8
3.5 YARN 集群缩容 3-8
3.5.1 使用场景 3-8
3.5.2 缩容前准备 3-9
3.5.3 缩容约束 3-9
3.5.4 缩容影响 3-9
3.5.5 缩容操作指导 3-9
3.5.6 缩容验证 3-10
3.6 YARN 权限访问控制 3-11
3.6.1 权限说明 3-11
3.6.2 权限使用操作示例 3-11
3.7 YARN 租户管理 3-14
3.7.1 租户介绍 3-14
3.7.2 新增租户 3-14
3.7.3 租户使用操作示例 3-17
3.8 YARN 调度器 3-18
3.8.1 YARN 调度器类型 3-18
3.8.2 大数据集群调度器的修改说明 3-19
4 开发指南 4-1
4.1 常用 API 4-1
4.2 MapReduce 统计词频示例 4-3
4.3 作业提交示例 4-9
4.4 作业监控 4-10
5 常见问题解答 5-1
5.1 调优 5-1
5.2 运维类问题 5-3

1 组件简介

1.1 YARN

1.1.1 YARN 概述

- YARN 是 Hadoop2.0 新增的分布式资源管理系统，可以为上层应用提供统一的资源管理和调度。YARN 为集群的资源利用率、资源统一管理等方面带来了明显提升。
- YARN 与运行的应用程序完全解耦，只要符合 YARN 规范资源请求机制的应用程序都可以整合在 YARN 上运行，比如 MapReduce、Storm，Spark，Tez 等运算框架的应用程序。

1.1.2 YARN 架构

如图 1-1 所示，YARN 架构主要由 Resource Manager、Application Master（AM）和 Node Manager 组成。

图1-1 YARN 架构

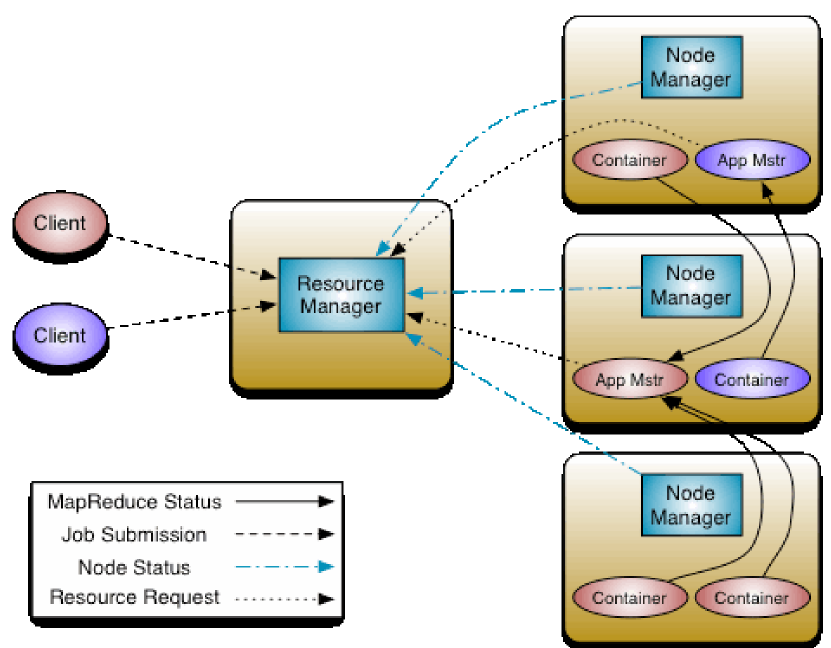


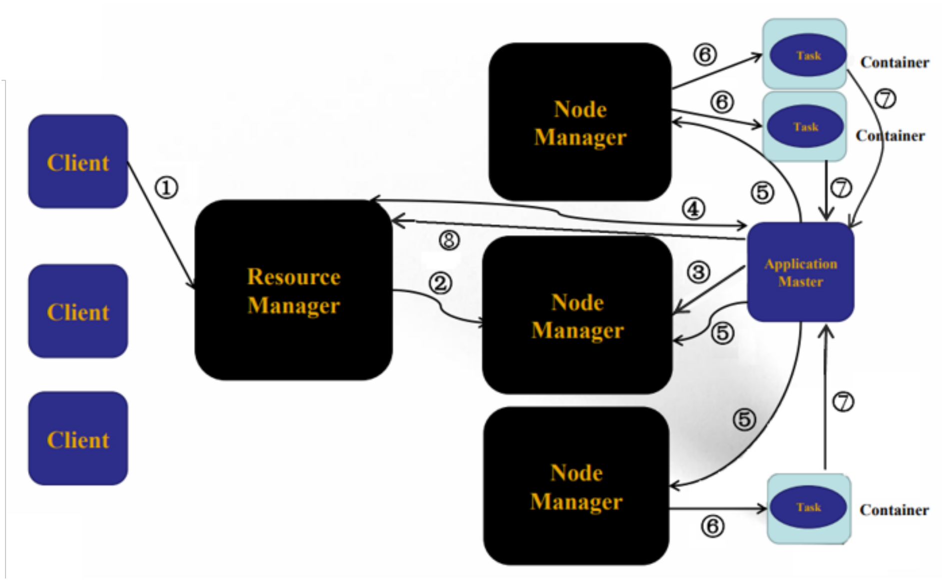
表1-1 进程介绍

进程	说明
ResourceManager（RM）	RM是一个全局资源管理器，负责整个系统所有资源的管理和分配，它主要有两个组件构成，其中： • 调度器（Scheduler）：负责为应用程序分配资源，但不监控和跟踪应用程序的状态，也不保证会重启由于应用程序本身或硬件出错而执行失败的应用程序 • 应用管理器（ApplicationsManager，ASM）：负责管理整个系统中的所有应用程序，包括应用程序提交、向调度器申请资源以启动 ApplicationMaster、监控

进程	说明
	ApplicationMaster 运行状态并在失败时重启等
NodeManager（NM）	NM是YARN中每个节点上的资源和任务管理器，管理集群中单个计算节点。它的功能包括： • 与 RM 保持通信，通过心跳向 RM 汇报自己本节点的资源使用情况、各个 Container 的运行情况、节点健康状况等 • 负责启动应用所需的容器，监控资源使用（内存、CPU 等）情况并将之汇报给调度器（Scheduler）
ApplicationMaster（AM）	AM负责一个Application生命周期内的所有工作，包括： • 负责从调度器（Scheduler）申请资源，将得到的资源进一步分配给内部任务（资源的二次分配），并监控所有任务的运行状态 • 与 NM 通信，启动或停止任务，并当任务运行失败时重新为任务申请资源以重启任务
Container	Container是YARN中资源的抽象，它将内存、CPU、磁盘等资源封装在一起。当AM向RM申请资源时，RM为AM返回的资源是用Container表示的，Container是一个动态的资源划分单位，是根据实际提交的应用程序所需要的资源自动生成的。YARN会为每个任务分配一个Container，且该任务只能使用该Container中的资源
Client	YARN Application的客户端，用户可通过客户端向ResourceManager提交任务，并可查询Application的运行状态等

1.1.3 YARN 工作步骤

图1-2 YARN 工作流程图



YARN 工作步骤如下：

(1) 用户向 YARN 中提交应用程序/作业，其中包括 ApplicaitonMaster 程序、启动 ApplicationMaster 的命令、用户程序等。

(2) ResourceManager 为该应用程序分配第一个 Container，并与对应的 NodeManager 通信，要求它在这个 Containter 中启动该作业的 ApplicationMaster。

(3) ApplicationMaster 首先向 ResourceManager 注册，这样用户可以直接通过 ResourceManager 查询作业的运行状态；然后它将为该应用程序中的各个任务申请资源并监控任务的运行状态，直到运行结束。重复步骤 4-7。

(4) ApplicationMaster 采用轮询的方式通过 RPC 协议向 ResourceManager 申请和领取资源。

(5) 一旦 ApplicationMaster 申请到资源后，便与该 Container 对应的 NodeManager 通信，要求它在该 Container 中启动任务。

(6) NodeManager 启动任务。

(7) 各个任务向对应的 ApplicationMaster 汇报自己的状态和进度，以让 ApplicaitonMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；在作业运行过程中，用户可随时向 ApplicationMaster 查询作业当前运行状态。

(8) 作业完成后，ApplicationMaster 向 ResourceManager 注销并关闭自己。

1.1.4 YARN 特点

- 良好的扩展性
- 高可用性
- 支持多种计算框架的应用程序

1.2 MapReduce

1.2.1 MapReduce 概述

- MapReduce 是一种编程模型，适用于大规模数据集的并行运算。其采用“分而治之”的思想，将大规模数据集的操作分发给各个子节点来共同完成，并整合各个子节点的中间结果得到最终结果。简单地说，MapReduce 就是“任务的分解与结果的汇总”。
- 在分布式计算中，MapReduce 框架负责处理并行编程中的分布式存储、工作调度、负载均衡、容错均衡等复杂问题，把处理过程抽象为 Map 和 Reduce 两个函数，Map 负责把任务分解成多个任务，Reduce 负责汇总多任务处理的结果。

1.2.2 MapReduce 基本概念

- split

  split 是 MapReduce 框架根据一定的规则将源数据分成若干个小数据集的过程。其中一个小数据集被称为一个 split。
- Map

  Map 是将一个 split 根据用户定义的 Map 逻辑处理后，经 MapReduce 框架处理形成输出结果，供后续 Reduce 过程使用。
- Reduce

  Reduce 过程是将 Map 的结果作为输入，根据用户定义的 Reduce 逻辑，将结果处理、汇总并输出。

- Combine

  Combine 是一个可由用户自定义的过程，类似于 Map 和 Reduce，它负责在本地聚集中间过程的输出，降低从 Map 到 Reduce 的数据传输量。

  【说明】不是所有用 MapReduce 框架实现的算法都适合增加 Combine 过程，例如求平均值不适合增加 Combine 过程。

- Partition
  - Partition 的数量与一个作业中 Reduce 任务的数量是一样的(无 Partition 指定时，Partition 数量默认为 1 个）。可以自定义 Partitioner 来控制 partition 的数目。
  - MapReduce 默认的 Partitioner 是 HashPartitioner。
- 输入输出（InputFormat，OutputFormat）
  - 根据用户指定的 InputFormat 来读取数据、切割数据集并将切割形成的多条键值对提供给 Map 任务进行处理，决定并行启动的 Map 任务数。
  - 根据用户指定的 OutputFormat，把生成的键值对输出为特定格式的数据。
  - Map、Reduce 两个阶段都是处理<key,value>键值对，即 MapReduce 框架把作业的输入作为一组<key,value>键值对，同样也产出一组<key,value>键值对作为输出，但这两组键值对的类型可能不同。对单个 Map 和 Reduce 而言，对键值对的处理为单线程串行处理。
- Shuffle

  Shuffle 描述数据从 Map 流向 Reduce 的过程。

说明

MapReduce 整体处理过程为：Input split->map->combiner->shuffle->reduce->output。

1.2.3 MapReduce 特点

- 大规模并行计算
- 高容错性
- 良好的扩展性
- 适用于大型数据集（如 TB 级别）

1.2.4 MapReduce 应用程序运行过程

MapReduce 应用程序的运行过程如下：

(1) 将 HDFS 上的输入文件进行 Split 切分，生成多个 input split，一个 input split 由一个 Map 任务进行处理。

(2) Map 阶段调用 InputFormat 实现类（默认是 FileInputFormat）中的 getRecordReader 方法获取 RecordReader 对象（默认是 LineRecordReader），该对象的 next 方法会迭代读取 split 分片上的内容，输出键值对 key-value（具体逻辑可由用户编码实现）。

(3) Map 阶段把处理之后的数据写入循环的内存缓冲区，当内存缓冲区达到一定的阀值（默认 80%），MapReduce 框架会启动后台线程将缓冲区的内容写入磁盘中的临时溢出文件（即 spill

阶段）。在写文件前，后台线程会根据 partitioner 划分相应的分区，最终每个 Key 会写入对应的分区，并且每个分区内按照 Key 进行排序。

(4) Map 阶段完成之前，会合并多个临时溢出文件，并删除之前所有的临时溢出文件。

(5) 一个 Partition 对应一个 Reduce 任务。Reduce 阶段会启动数据拷贝线程，通过 HTTP 方式请求 Map 阶段的输出文件，Reduce 会将多个 Map 的结果放入内存缓冲区并进行 merge，使用 Reduce 方法处理数据，并将结果输出到 HDFS。

【说明】由于 Map 任务可能会在多个节点上运行，因此 Reduce 阶段会从多个节点上拷贝数据。

1.2.5 MapReduce 应用场景

- MapReduce 适用于批处理任务，即在可接受的时间内对整个数据集计算出某个特定的结果，不适合需要实时反映数据变化状态的应用场景。
- MapReduce 计算模型是以“行”为处理单位的，无法回溯已处理过的“行”。每行日志都必须是一个独立的语义单元，行与行之间不能有语义上的关联。
- 相对于传统的关系型数据库管理系统，MapReduce 计算模型更适合于处理半结构化或非结构化的数据。
- MapReduce 的典型应用场景有日志分析、搜索索引、数据挖掘、信息提取、大规模的算法图形处理、文字处理 、分布排序、Web 连接图反转和 Web 访问日志分析。

2 快速入门

2.1 组件安装

说明

- 通常 HDFS、YARN、MapReduce 一起使用，所以安装组件时，HDFS、MapReduce、YARN 必须同时安装，对外呈现名称为 Hadoop。
- YARN、HDFS 依赖 Zookeeper，安装 HDFS 时必须同时安装 Zookeeper 组件。
- 在 Hadoop 集群中，安装 YARN 和 MapReduce 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。

2.1.1 数据目录检查

根据现场磁盘分区方案和挂盘方案的不同，YARN 安装完成后，必须对其数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表 2-1 所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
HDFS	是（配置项的参数值默认选择3个挂载路径）	dfs.namenode.name.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
	是（配置项的参数值默认使用全部挂载路径）	dfs.datanode.data.dir	
	未开启高可用时，需要检查该配置项（配置项的参数值默认使用全部挂载路径）	dfs.namenode.checkpoint.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
	开启高可用时，需要检查该配置项（配置项的参数值默认选择1个挂载路径）	dfs.journalnode.edits.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
YARN	是（配置项的参数值默认使用全部挂载路径）	yarn.nodemanager.local-dirs	此目录为数据目录，用于存放应用程序的运行依赖包等信息。检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
		yarn.nodemanager.log-dirs	

组件	是否需要检查	被影响的配置项	如何解决
	是（配置项的参数值默认使用某一个挂载路径）	yarn.timeline-service.leveldb-state-store.path	此目录为数据目录，用于记录应用程序运行状态等信息。检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
Zookeeper	是（配置项的参数值默认使用某一个挂载路径）	dataDir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录

2.1.2 查看组件的日志路径

表2-2 组件日志路径说明

组件	日志路径
HDFS	/var/de_log/hadoop/user_hdfs
MapReduce2	HistoryServer日志路径：/var/de_log/hadoop-yarn/user_mapred/和/var/de_log/hadoop-mapreduce/mapred/
YARN	/var/de_log/hadoop-yarn/user_yarn 【说明】上述的日志是YARN本身的日志。另外： • 执行在 YARN 上的应用日志，可以通过 YARN 的 UI 界面查看，日志文件实际存储位置默认是在 HDFS 上，默认路径为/app-logs • 如果日志不聚合，可以配置 yarn.log-aggregation-enable 为 false • 内嵌的 HBase 日志路径为：/var/de_log/hadoop-yarn/embedded-yarn-ats-hbase，在 timeline 安装的节点上可看到（前提条件：配置项 use_external_hbase、is_hbase_system_service_launch 的值均为 false）
ZooKeeper	/var/de_log/zookeeper/user_{user.name}/，其中${user.name}是指执行任务的用户名

2.2 运行状态监控

1. 查看组件详情

在 Yarn 或 MapReduce 组件详情页面可查看组件的基本配置信息、进程部署拓扑详情、组件部署配置详情及配置修改历史等，如图 2-1、图 2-2 所示，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。

- 概要（仅 YARN 组件具有该项功能）：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 YARN 组件详情

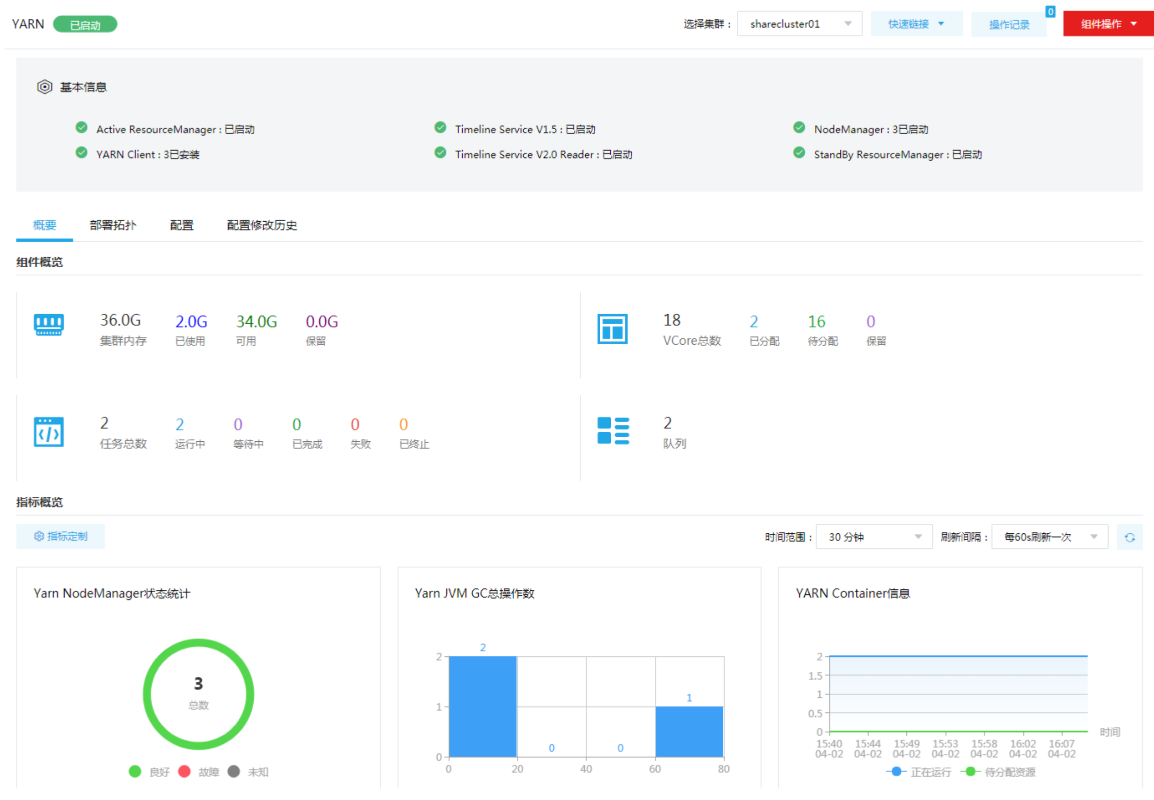
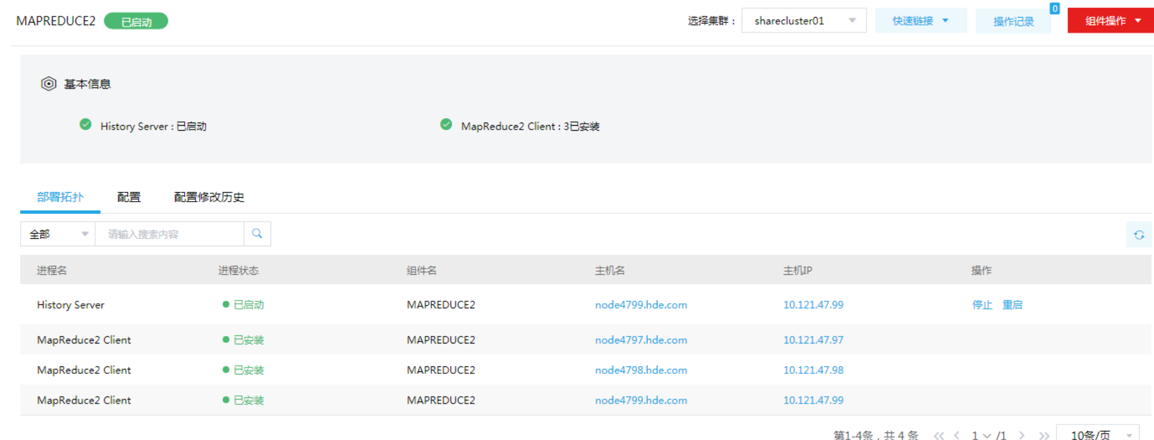


图2-2 MapReduce 组件详情



2. 组件检查

执行 MapReduce 或 YARN 组件检查时，内置操作方案如下：

- MapReduce 组件检查时会执行一个 wordcount 任务，确认 MapReduce 是否可用，如果任务执行失败，检查会失败。
- YARN 组件检查时会执行一个 distributedshell 任务，查看是否有 active ResourceManager 并检查 ResourceManager UI 是否可用，如果一切正常，则检查通过，表示 YARN 组件可正常使用。

集群在使用过程中，根据实际需要，可对 MapReduce 或 YARN 执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：

- o 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - – 在集群详情页面选择[组件]页签，单击业务组件列表中 MapReduce 或 YARN 组件对应的<组件检查>按钮。
  - – 在集群详情页面选择[组件]页签，单击业务组件列表中 MapReduce 或 YARN 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- o 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态。如图 2-3 所示，表示该组件检查成功，可正常使用。

图2-3 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“MAPREDUCE2 Service Check”或“YARN Service Check”组件操作执行的详细信息以及操作日志详情，如图 2-4、图 2-5 所示，根据操作日志可判断组件检查的具体情况。

图2-4 MapReduce 组件检查日志详情

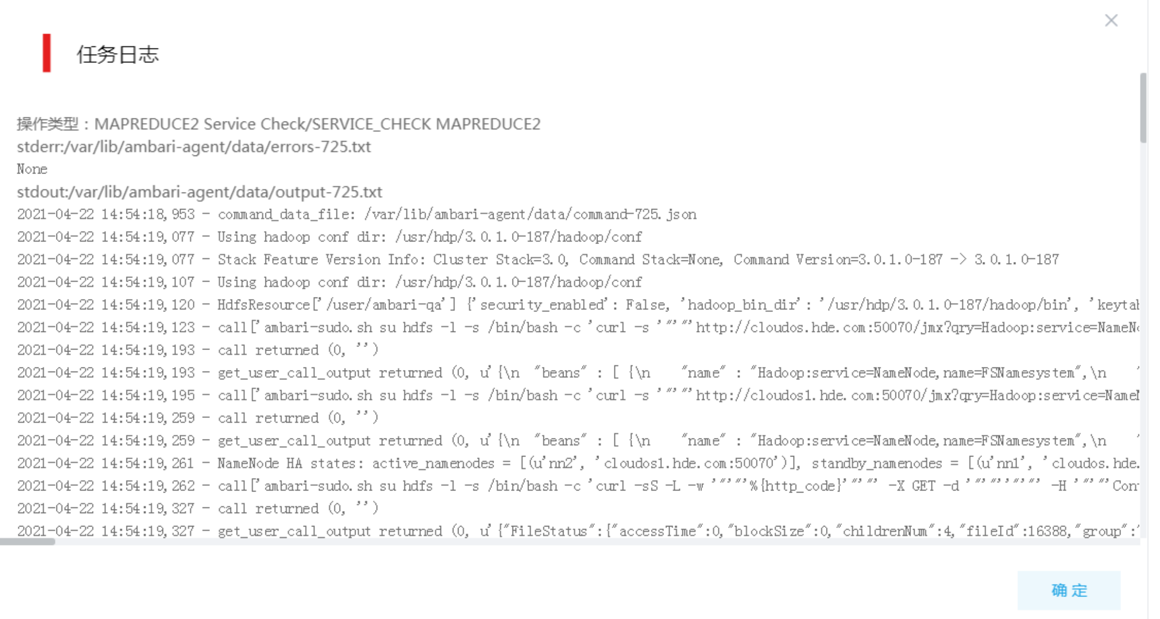


图2-5 YARN 组件检查日志详情

```
任务日志

操作类型：YARN Service Check/SERVICE_CHECK YARN
stderr:/var/lib/ambari-agent/data/errors-726.txt
None
stdout:/var/lib/ambari-agent/data/output-726.txt
2021-04-22 14:56:55,460 - command_data_file: /var/lib/ambari-agent/data/command-726.json
2021-04-22 14:56:55,595 - Using hadoop conf dir: /usr/hdp/3.0.1.0-187/hadoop/conf
2021-04-22 14:56:55,596 - Stack Feature Version Info: Cluster Stack=3.0, Command Stack=None, Command Version=3.0.1.0-187 -> 3.0.1.0-187
2021-04-22 14:56:55,625 - Using hadoop conf dir: /usr/hdp/3.0.1.0-187/hadoop/conf
2021-04-22 14:56:55,639 - HdfsResource['/user/ambari-qa'] {'security_enabled': False, 'hadoop_bin_dir': '/usr/hdp/3.0.1.0-187/hadoop/bin', 'keytal
2021-04-22 14:56:55,642 - call['ambari-sudo.sh su hdfs -l -s /bin/bash -c 'curl -s '"'"'http://cloudos.hde.com:50070/jmx?qry=Hadoop:service=NameN
2021-04-22 14:56:55,707 - call returned (0, '')
2021-04-22 14:56:55,707 - get_user_call_output returned (0, u'{\n "beans" : [ {\n "name" : "Hadoop:service=NameNode,name=FSNamesystem",\n
2021-04-22 14:56:55,709 - call['ambari-sudo.sh su hdfs -l -s /bin/bash -c 'curl -s '"'"'http://cloudos1.hde.com:50070/jmx?qry=Hadoop:service=NameN
2021-04-22 14:56:55,769 - call returned (0, '')
2021-04-22 14:56:55,769 - get_user_call_output returned (0, u'{\n "beans" : [ {\n "name" : "Hadoop:service=NameNode,name=FSNamesystem",\n
2021-04-22 14:56:55,770 - NameNode HA states: active_namenodes = [(u'nn2', 'cloudos1.hde.com:50070')], standby_namenodes = [(u'nn1', 'cloudos.hde.
2021-04-22 14:56:55,772 - call['ambari-sudo.sh su hdfs -l -s /bin/bash -c 'curl -sS -L -w '"'"'%{http_code}'"'"' -X GET -d '"'"''"'"' -H '"'"'Con
2021-04-22 14:56:55,835 - call returned (0, '')
2021-04-22 14:56:55,835 - get_user_call_output returned (0, u'{"FileStatus":{"accessTime":0,"blockSize":0,"childrenNum":4,"fileId":16388,"group":

确定
```

2.3 快速使用指导

注意

根据大数据集群是否开启 Kerberos 认证，用户访问时的认证方式不同，详情请参见本章节内容。

YARN&MapReduce 既可以通过集群用户访问，又可以通过组件超级用户访问。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
  - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。
  - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户。比如 YARN 组件的 yarn 用户，HDFS 组件的 hdfs 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 kerberos 环境

说明

- 非 Kerberos 环境下，不需要用户做身份认证即可直接对组件执行管理操作。
- 本章节操作需要切换至具有相关权限的用户。

大数据集群中安装 MapReduce2 和 YARN 之后，可以运行 WordCount 测试用例，该测试用例可以对文件内容进行词频统计。运行 wordcount 示例的操作步骤如下：

(1) 创建 /tmp/wordcount.txt 测试文件，并编辑文件内容为 “this is a test”，执行如下命令：

```
vi /tmp/wordcount.txt
```

(2) 切换至具有创建目录权限的用户（例如 hdfs 用户），创建 input 目录，执行如下命令：

```
hdfs  dfs -mkdir /input
```

(3) 把测试文本上传到 HDFS 的 input 目录上，执行如下命令：

```
hdfs dfs -put /tmp/wordcount.txt /input
```

(4) 运行 WordCount 实例，执行如下命令：

```
yarn jar /usr/hdp/3.0.1.0-187/hadoop-mapreduce/hadoop-mapreduce-examples-3.0.0-cdh6.2.0.jar wordcount /input /output
```

参数说明：

- yarn：YARN 组件基础脚本
- jar：提交任务的命令参数
- /usr/hdp/3.0.1.0-187/hadoop-mapreduce/hadoop-mapreduce-examples-3.0.0-cdh6.2.0.jar：mapreduce 组件自带的 wordcount 样例 jar 包路径

- /input：输入路径，该路径下包含了需要 wordcount 样例处理的文件
- /output：计算结果的输出路径

(5) 查看计算输出结果

```
hdfs dfs -cat /output/part-r-00000
```

结果如下：

```
a      1
is     1
test   1
this   1
```

2.3.2 Kerberos 环境

注意

- Kerberos 环境下，若想对 YARN 和 MapReduce 执行管理操作，则必须首先进行用户身份认证，认证方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。
- 本章节操作需要切换至具有相关权限的用户。

1. Kerberos 环境下用户身份认证

如果大数据集群开启 Kerberos，若想操作 YARN 和 MapReduce，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证（仅 YARN 支持）

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

YARN 和 MapReduce 组件可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**
  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的/etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：
  ```
  chown user1 /etc/security/keytabs/user1.keytab
  ```
  b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：
  ```
  klist -k user1.keytab
  ```

【说明】如图 2-6 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-6 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

su user1

kinit -kt user1.keytab user1@TENANTC.COM

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-7 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

（二）组件超级用户身份认证(仅 YARN 支持)

YARN 可以通过组件超级用户访问，比如 yarn 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 yarn 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 yarn 的认证文件“yarn.service.keytab”。

【说明】在 YARN Client 节点上，需要将 YARN 的认证文件“yarn.service.keytab”上传节点的/etc/security/keytabs/目录下进行认证。

(2) 使用 ***klist*** 命令查看 yarn.service.keytab 的 principal 名称，命令如下：

klist -k yarn.service.keytab

【说明】如图 2-8 所示，红框内容即为 yarn.service.keytab 的 principal 名称。

图2-8 认证文件的 principal 名称

```
[root@hppnode1 keytabs]# klist -k yarn.service.keytab
Keytab name: FILE:yarn.service.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 yarn/hppnode1.hde.com@TESTSHARE.COM
   2 yarn/hppnode1.hde.com@TESTSHARE.COM
   2 yarn/hppnode1.hde.com@TESTSHARE.COM
   2 yarn/hppnode1.hde.com@TESTSHARE.COM
   2 yarn/hppnode1.hde.com@TESTSHARE.COM
```

(3) 切换至用户 yarn，并执行身份验证的命令如下：

su yarn

kinit -kt yarn.service.keytab yarn/hppnode1.hde.com@TESTSHARE.COM

【说明】其中：yarn.service.keytab 为 YARN 的认证文件，yarn/hppnode1.hde.com@TESTSHARE.COM 为 yarn.service.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

2. 运行 wordcount 示例

用户身份认证成功后，即可运行 WordCount 测试用例，详情请参见 2.3.1 非 kerberos 环境运行 wordcount 实例。

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

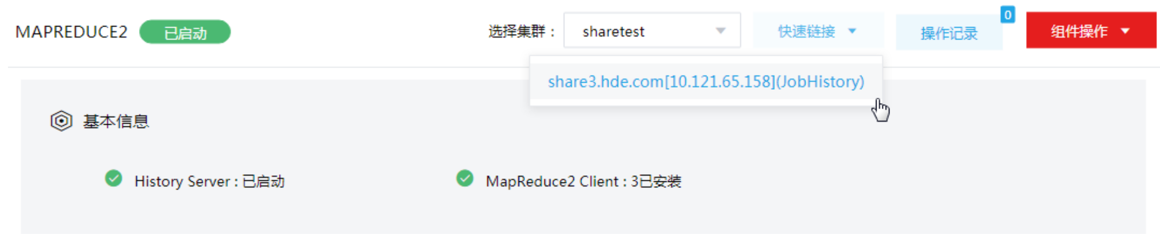
(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

2.4.2 访问 MapReduce 快速链接

MapReduce 提供了应用程序历史页面（即 JobHistory UI），通过该页面可以查看应用程序的运行情况。

(1) 如图 2-9 所示，在 MapReduce 组件详情页面的右上角[快速链接]的下拉框中，可以获取访问入口信息。

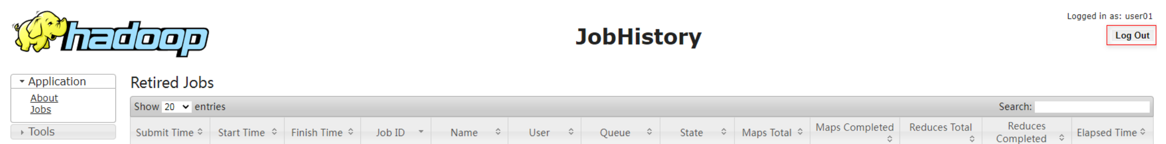
图2-9 MapReduce 快速链接



(2) 根据集群是否开启 Kerberos，访问 MapReduce 快速链接分为两种情况：

- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，可直接跳转访问对应的 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。在 UI 页面支持退出登录，如图 2-10 所示。单击页面右上角的<Log Out>按钮，即可清除当前用户的登录信息重新跳转至登录页面。

图2-10 退出登录



(3) 在 MapReduce 应用程序历史页面（即：MapReduce 的 JobHistory 页面），可以查看应用程序的运行情况。如图 2-11 所示，红框内可以查看当前的 WordCount 作业 ID 以及运行结果等信息。

图2-11 JobHistory Web UI 界面

2.4.3 访问 YARN 快速链接

说明

对于集群超级用户，可查看提交至 YARN 上所有的任务；对于集群普通用户，能查看的任务与该用户绑定的角色权限有关，拥有 submit-app 权限的用户可查看该用户的所有任务，拥有 admin-queue 权限的用户可查看该用户的所有任务和该队列上的所有任务。

YARN 提供了作业监控页面（即 ResourceManager UI），通过该页面可以查看运行的作业信息。

(1) 如图 2-12 所示，在 YARN 组件组件详情页面的右上角[快速链接]的下拉框中，可以获取访问入口信息。

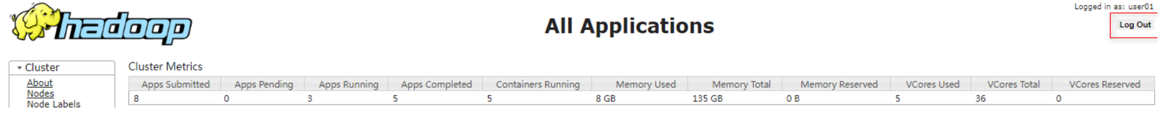
【说明】当集群开启高可用时，YARN 同步开启 HA，此时有两个访问入口，推荐选择 active 状态的链接。

图2-12 YARN 快速链接

(2) 根据集群是否开启 Kerberos，访问 YARN 快速链接分为两种情况：

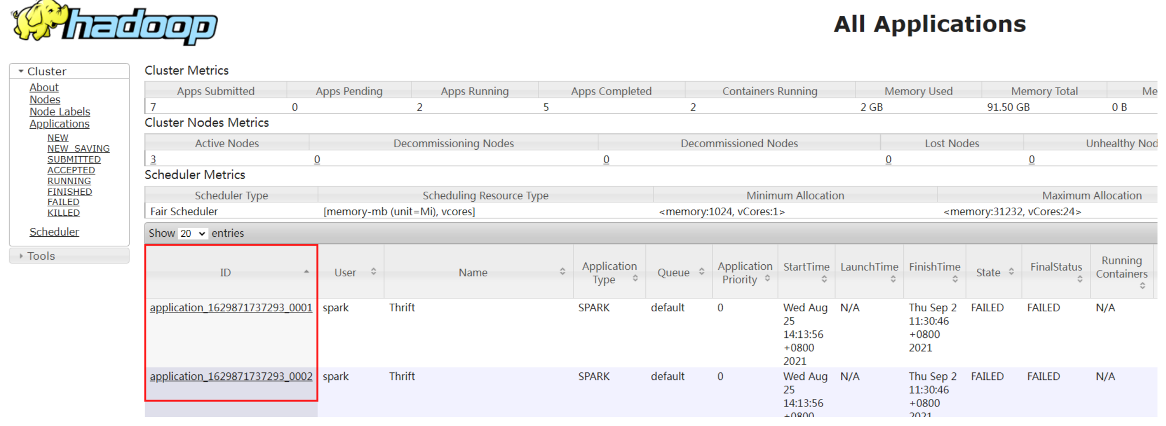
- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，可直接跳转访问对应的 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。在 UI 页面支持退出登录，如图 2-13 所示。单击页面右上角的<Log Out>按钮，即可清除当前用户的登录信息重新跳转至登录页面。

图2-13 退出登录



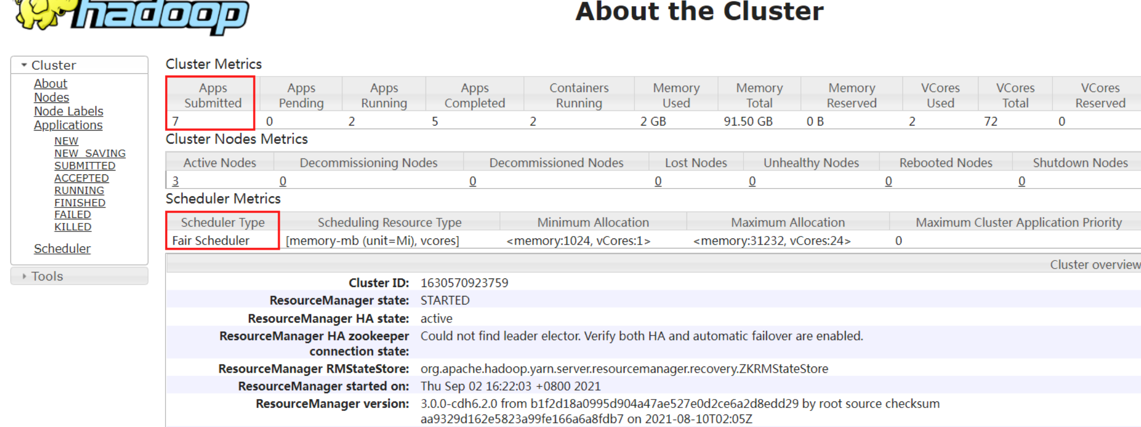
(3) 在 YARN 作业监控页面（即：YARN 的 All Applications 页面），展示集群运行任务的历史记录。如图 2-14 所示，点击红框内的任务 ID 可以查看相关任务的运行结果等信息。

图2-14 All Applications 页面



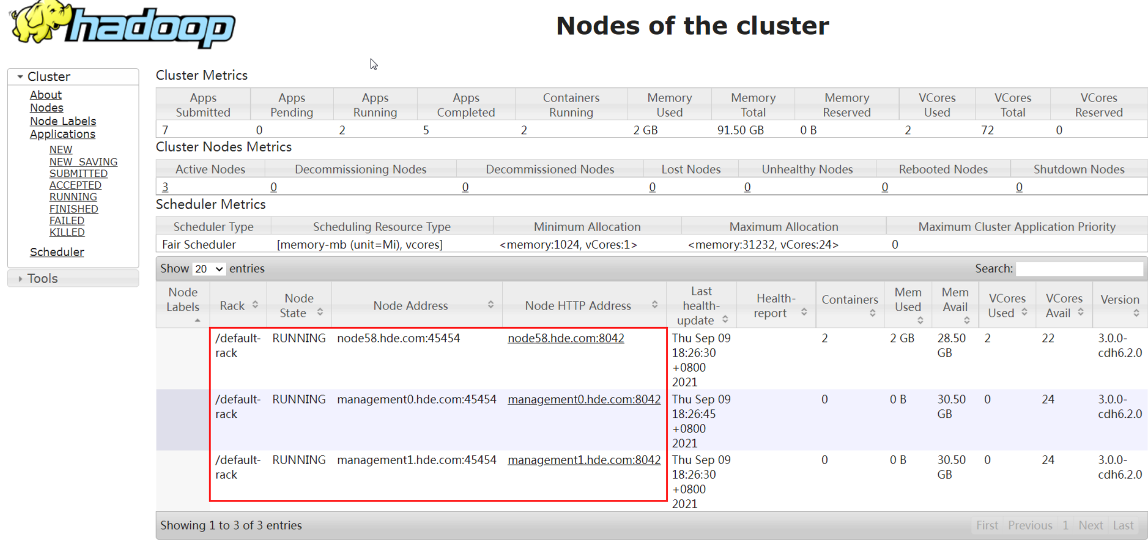
(4) 选择左侧[About]页签，进入 About the Cluster 页面，该页面展示了提交任务数，调度器类型等信息。如图 2-15 所示，红框中所示提交的任务总数为 7 个，调度器类型为公平调度器。

图2-15 About the Cluster 页面



(5) 选择左侧[Nodes]页签，进入 Nodes of the cluster 页面。如图 2-16 所示，该页面可以展示 NodeManager 节点的相关信息。

图2-16 Nodes of the cluster 页面



3 使用指南

3.1 YARN Shell命令

说明

关于 YARN Shell 命令的使用，详情请参见官网 http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YarnCommands.html - application。

1. yarn application [options]

命令选项	描述
-appStates <States>	使用-appStates命令，基于应用程序的状态来过滤应用程序。如果应用程序的状态有多个，用逗号分隔。有效的应用程序状态包含如下：ALL, NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED
-appTypes <Types>	从RM返回的应用程序列表，使用-appTypes参数，支持基于应用程序类型的过滤，使用-appStates参数，支持对应用程序状态的过滤
-list	使用-list命令，基于应用程序类型来过滤应用程序。如果应用程序的类型有多个，用逗号分隔
-kill <ApplicationId>	kill掉指定的应用程序
-status <ApplicationId>	打印应用程序的状态

(1) 查看 YARN 上处于 ACCEPTED 状态的作业，执行如下命令：

yarn application -list -appStates ACCEPTED

(2) 查看 YARN 上运行的作业，执行如下命令：

yarn application -list

(3) 手动 kill 掉指定的应用程序，执行如下命令：

yarn application -kill application_ID

2. yarn container [options]

命令选项	描述
-help	帮助
-list <Application Attempt Id>	应用程序尝试的Containers列表
-status <ContainerId>	打印Container的状态

3. 提交作业

用户可以将写好的应用程序打包成 jar 文件，向集群提交作业，执行如下命令：

```
yarn jar <jar> [mainClass] args...
```

例如，将写好的应用程序 jar 包提交至集群，执行如下命令：

```
yarn jar mapreduce-example.jar WordCount /input /output
```

其中：

- mapreduce-example.jar：jar 包名称
- WordCount：jar 包的主类
- /input 和/output：jar 包运行的参数

3.2 Client下载/安装/使用/卸载

HDFS、MapReduce2、YARN 三个组件的 Client 相同，在客户端节点上安装其中一个即可。本章节以 YARN Client 为例进行介绍。在客户端节点上安装 Client 后，即可直接连接集群中的 YARN/MapReduce2，进行组件维护、任务管理等操作。

3.2.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作，租户集群中的租户的 YARN 组件也支持下载 Client），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在[集群管理/集群列表]页面中，点击集群名称进入集群详情页面，在集群已安装的组件列表中单击 YARN 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-1 所示。

图3-1 下载 Client 窗口

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：
  - 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
  - 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中的组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。
  - 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
  - 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.2.2 安装 Client

注意

- 安装 Client 的节点必须能与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置网络连接，仅非 root 用户需要执行此操作，root 用户可跳过此步骤。

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

```
./install.sh <安装路径> -o
```

说明

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为 /usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

```
./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>
```

3.2.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

  ```
  source bigdata_env
  ```

- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同：
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行校验即可直接访问组件。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行认证之后，才可访问组件。进行用户认证的操作方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

1. Kerberos 环境使用 Client 示例

用户身份认证成功后，在 kerberos 环境下，使用 MapReduce2/YARN 客户端提交一个 WordCount 作业示例的操作步骤如下：

说明

- 以下命令都是在客户端节点执行。
- 用户执行本章节操作时需要具有相关操作权限，关于权限管理的相关操作请参见 3.6 YARN 权限访问控制。

(1) 将需要进行词频统计的示例文件（/tmp/test.txt）上传到 HDFS 的/input 目录，执行如下命令：

```
hdfs dfs -put /tmp/test.txt /input
```

(2) 将 wordcount 应用程序的 jar 包上传到客户端节点。该示例使用 MapReduce2 和 YARN 自带的测试用例 jar 包，自带的测试用例所在集群中的位置为 /usr/hdp/3.0.1.0-187/hadoop-mapreduce/hadoop-mapreduce-examples-3.0.0-cdh6.2.0.jar

(3) 运行 wordcount 作业，执行如下命令：

```
yarn jar /opt/hadoop-mapreduce-examples-3.0.0-cdh6.2.0.jar wordcount /input /output
```

其中：

- /opt/hadoop-mapreduce-examples-3.0.0-cdh6.2.0.jar：客户端节点 wordcount 作业 jar 包
- wordcount：jar 包主类
- /input：需要进行词频统计的文件目录
- /output：词频统计结果的存放目录

3.2.4 卸载 Client 客户端

集群重装之后，之前安装的 Client 客户端将不可用。此时需要卸载 Client，执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 的安装包目录下执行以下命令：

```
./uninstall.sh
```

(2) 卸载脚本执行结束后，安装目录将自动删除。

3.3 YARN刷新队列

说明

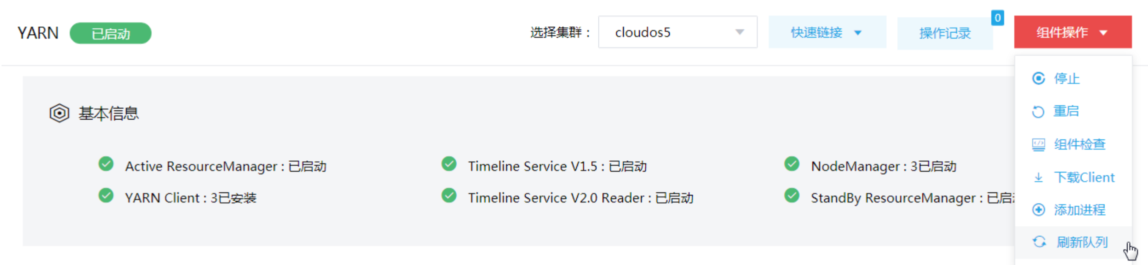
若在 YARN 中新增或修改了一个队列，执行刷新队列操作可使该队列立即生效，而不用重启整个 YARN。

集群在使用过程中，根据实际需要，可对 YARN 执行刷新队列的操作。

(1) 对 YARN 执行刷新队列的方式有以下两种，任选其一即可：
   - 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。在集群详情页面选择[组件]页签，单击组件列表中 YARN 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<刷新队列>按钮。
   - 在 YARN 组件管理的组件详情页面右上角组件操作的下拉框中选择<刷新队列>按钮。

(2) 然后在弹窗中进行确定后，即可对 YARN 队列进行即时刷新。

图3-2 刷新队列



3.4 YARN集群扩容

YARN 集群扩容是指在某节点上新增安装 NodeManager。

3.4.1 使用场景

随着业务量的增长，集群计算能力无法满足业务需求时，需要考虑对 YARN 集群进行扩容。

3.4.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在大数据集群中新增安装 NodeManager 进程。
   - 如果集群中有节点没有安装 NodeManager，直接在集群节点中添加 NodeManager 进程。
   - 如果集群中所有节点均已安装 NodeManager，进行 NodeManager 扩容前则需要先添加主机。

2. 环境检查

(1) 登录大数据平台管理系统，查看 YARN 组件的状态是否正常。

(2) 进入 YARN 组件详情页，查看 YARN 的部署拓扑，确保每个服务的状态正常，ResourceManager、NodeManager、Timeline Service 处于“已启动”状态，YARN Client 处于“已安装”状态。

3.4.3 扩容约束

- 扩容节点操作系统版本与集群内部版本需保持一致。
- 扩容操作一旦开始，不支持中止。

3.4.4 扩容影响

- 一旦扩容失败，需要及时将扩容失败的节点剔除。
- 扩容成功后，集群计算资源增多。

3.4.5 扩容操作指导

注意

若集群中所有节点均已安装 NodeManager，进行 NodeManager 扩容前则需要先添加主机，然后再进行 NodeManager 扩容。如果集群中有扩容所用主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 YARN 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-3 所示。

a. 选择进程及主机。

在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

b. 部署进程。

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程。

图3-3 添加 NodeManager

(3) 查看组件数量

NodeManager 扩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 NodeManager 安装数量的变化。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.4.6 扩容验证

(1) 登录大数据平台管理系统。

(2) 执行 YARN 组件检查，确保 YARN 组件可正常使用，详情请参考 2.2 2. 组件检查。

(3) 查看 YARN 组件部署拓扑里是否已有新增的扩容节点。

(4) 打开 YARN 快速链接，在 YARN UI 页面查看 YARN 最新资源信息是否符合预期。

3.5 YARN集群缩容

YARN 集群缩容是指将某节点上已安装的 NodeManager 进行删除。

3.5.1 使用场景

YARN 集群缩容的场景主要有：

- 初始 NodeManager 节点规划不合理。
- 当 NodeManager 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.5.2 缩容前准备

1. 缩容规划

缩容可以删除任意 NodeManager 节点。

2. 环境检查

(1) 登录大数据平台管理系统，查看 YARN 组件的状态是否正常。

(2) 进入 YARN 组件详情页，查看 YARN 的部署拓扑，确保每个服务的状态正常，ResourceManager、NodeManager、Timeline Service 处于“已启动”状态，YARN Client 处于“已安装”状态。

3.5.3 缩容约束

- 在生产环境中，缩容不可回退或暂停，请谨慎使用。
- 删除进程前请确认该节点上无任务在执行。

3.5.4 缩容影响

- 为保证业务安全，一般不建议对 YARN 进行缩容。

3.5.5 缩容操作指导

说明

NodeManager 缩容操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行 NodeManager 缩容操作”为例进行说明，在主机详情页面执行 NodeManager 缩容操作，与其类似不再进行说明。

缩容操作步骤如下：

(1) 在 YARN 组件详情页面，选择缩容节点。

(2) 在组件详情页面[部署拓扑]页签下，选择已安装 NodeManager 进程且需要缩容的主机，然后单击该进程右侧操作中的<停止>按钮，停止 NodeManager，如图 3-4 所示。

图3-4 停止 NodeManager

(3) 删除 NodeManager

待 NodeManager 停止后，如图 3-5 所示，在该组件右侧操作栏中选择<删除>按钮，然后按照弹窗中的提示进行相关操作后，可删除该组件，即完成 NodeManager 缩容。

图3-5 删除 NodeManager

(4) 查看组件状态

NodeManager 缩容完成之后，在[部署拓扑]页面可以看到 NodeManager 安装数量变化，此时根据界面提示需要重新启动已失效的所有组件。

(5) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.5.6 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 YARN 组件检查，确保 YARN 组件可正常使用，详情请参考 2.2 2. 组件检查。

(3) 查看 YARN 组件部署拓扑里相关缩容节点是否已经删除。

(4) 打开 YARN 快速链接，在 YARN UI 页面查看 YARN 最新资源信息是否符合预期。

3.6 YARN权限访问控制

说明

仅开启“安全管理/权限管理”且运行正常的集群可使用角色管理功能。

权限管理是安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

3.6.1 权限说明

在开启权限管理的集群中，用户对 YARN 的操作需被赋予相关权限后才能执行。YARN 支持对队列配置权限（支持使用通配符*模糊适配），权限包括：submit-app、admin-queue。YARN 权限对应的常用操作如表 3-1 所示。

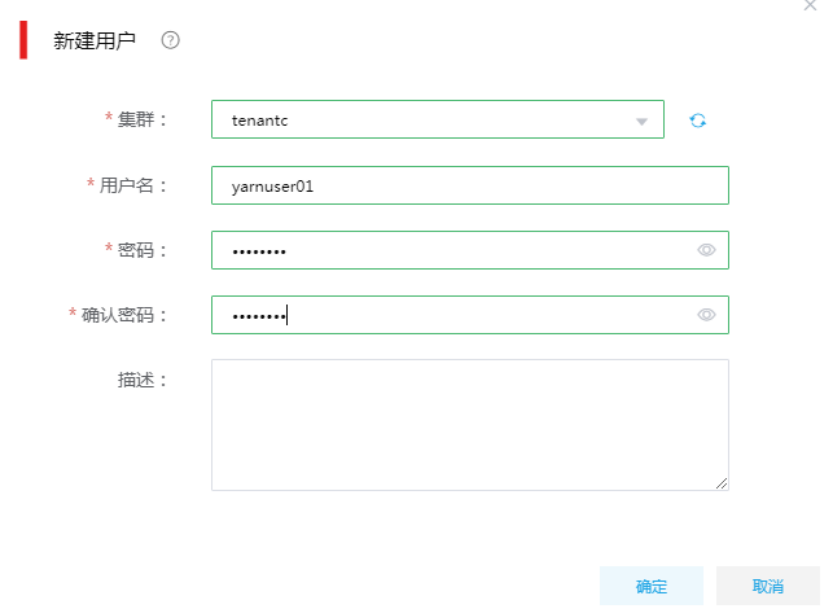
表3-1 YARN 权限说明

权限类型	对应的组件操作
submit-app	提交任务到队列，yarn application可查看该用户的所有任务
admin-queue	包含submit-app权限，同时yarn application可查看该用户的所有任务和该队列上的所有任务

3.6.2 权限使用操作示例

(1) 在[集群权限/用户管理]页面，新建用户 yarnuser01，如图 3-6 所示，新建用户默认没有任何权限。

图3-6 新建用户 yarnuser01



(2) 使用 yarnuser01 用户执行 MR 任务，如图 3-7 所示，用户 yarnuser01 没有提交到 default 队列任务的权限，如图 3-8。

图3-7 执行 MR 任务

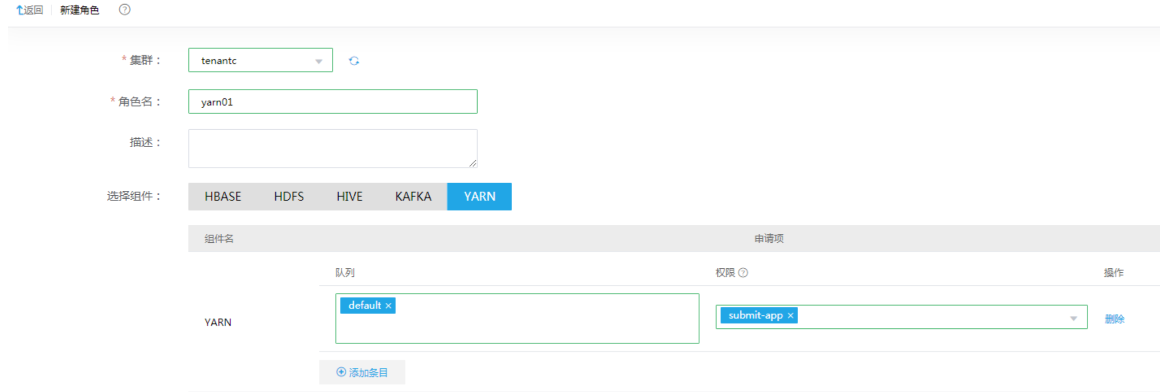
```
sh-4.2$ yarn jar /usr/hdp/3.0.1.0-187/hadoop-mapreduce/hadoop-mapreduce-examples-3.0.0-cdh6.2.0.jar  wordcount /tmp/input  /tmp/output1
```

图3-8 执行失败

```
2020-04-10 10:42:42,401 INFO  mapreduce.JobSubmitter (JobSubmitter.java:submitJobInternal(260)) - Cleaning up the staging area /user/yarnuser01/.staging/job_1586227513169_0006
java.io.IOException: org.apache.hadoop.yarn.exceptions.YarnException: org.apache.hadoop.security.AccessControlException: User yarnuser01 does not have permission to submit application_1586227513169_0006 to queue default
```

(3) 在[集群权限/角色管理]页面，新建角色 yarn01，并进行权限设置，授予 yarn01 角色队列 default 的 submit-app 权限。

图3-9 新建角色 yarn01



(4) 在[集群权限/用户管理]页面，为 yarnuser01 用户授予角色 yarn01。此时，用户 yarnuser01 也拥有了角色 yarn01 对应权限。

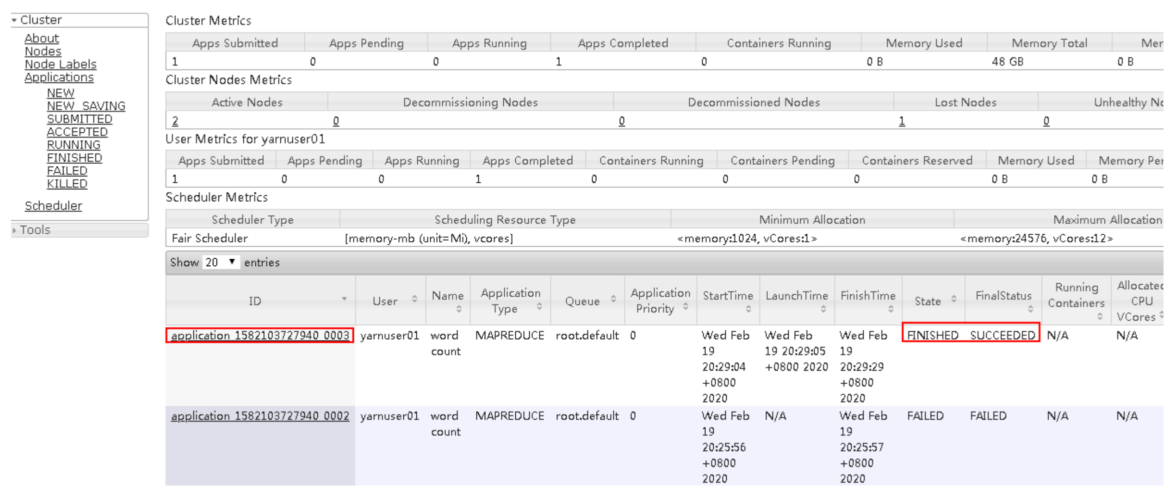
(5) 使用 yarnuser01 用户重新执行 MR 操作。如图 3-10 所示，任务成功执行。

图3-10 执行成功

```
2020-04-10 10:47:09,167 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647
)) -  map 0% reduce 0%
2020-04-10 10:47:17,252 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647
)) -  map 100% reduce 0%
2020-04-10 10:47:23,287 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647
)) -  map 100% reduce 100%
2020-04-10 10:47:23,298 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1658
)) - Job job_1586227513169_0007 completed successfully
2020-04-10 10:47:23,382 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1665
)) - Counters: 54
```

(6) yarnuser01 用户通过 YARN 快速链接查看执行的任务，可以看到 yarnuser01 提交的两次任务，第一次没有权限时失败，第二次有权限时成功，如图 3-11 所示。

图3-11 任务监控



3.7 YARN租户管理

注意

- 新增租户时，要求系统中必须已成功创建租户模式的 Hadoop 集群。
- 关于租户管理模块功能的配置或使用详情请参见产品在线联机帮助。
- 新增 YARN 租户或为 YARN 租户执行扩/缩容操作时，若 YARN 租户资源较小则可能导致任务无法执行，请根据实际需求申请足够资源。

多个租户之间共享一套集群，共享网络和集群资源，并且不同租户之间保证资源隔离。

- 新增租户

  普通用户在自己创建的租户集群中申请租户时，无需审批，会直接触发新增租户的操作。普通用户在其他用户创建的租户集群中申请租户时，则需要走流程审批，待审批人审批通过后才能触发新增租户的操作。管理员用户新增租户时，无需审批，会直接触发新增租户的操作。

- 租户管理操作

  普通用户在自己创建的租户集群中执行租户续期、资源扩缩容、配置 YARN 动态策略管理操作时，无需审批，会直接触发相关操作。普通用户在其他用户创建的租户集群中执行租户续期、资源扩缩容、配置 YARN 动态策略管理操作时，则需要走流程审批，待审批人审批通过后才能触发相关操作。管理员用户执行租户续期、资源扩缩容、配置 YARN 动态策略管理操作时，无需审批，会直接触发相关操作。

3.7.1 租户介绍

租户申请的 YARN 资源对应一个资源队列，资源名称即是队列名，每个资源队列提供配额的 CPU 和内存资源，用户可根据实际需要申请 YARN 组件资源。

【说明】

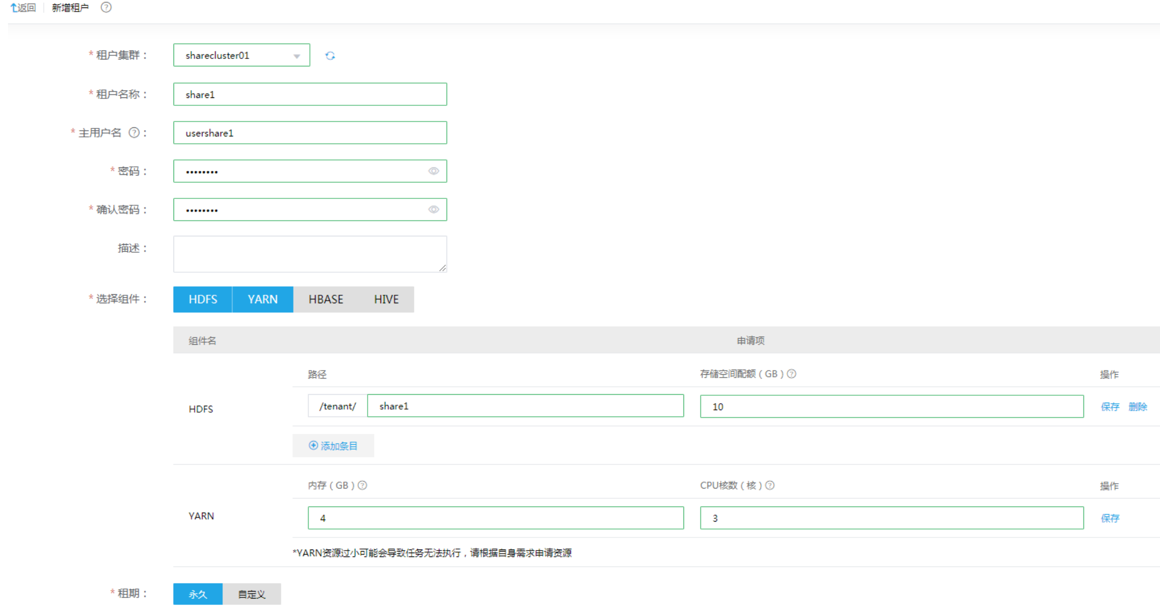
大数据集群的计算资源由 YARN 进行分配和调度。YARN 任务队列是大数据集群计算资源分配的基本单位，租户通过指定 YARN 任务队列可限定租户内任务运行可使用的计算资源。

- 租户申请 YARN 资源时，会单独为该租户创建一个队列（与租户名同名的队列），该队列的资源是从 default 队列里分出来的。
- 对于租户集群，YARN 仅支持公平调度（即 Fair cheduler），且禁止切换为其他调度器，否则租户功能将不可用。

3.7.2 新增租户

(1) 在[集群权限/租户管理]页面，单击<新增租户>按钮，跳转至新增租户页面，如图 3-12 所示。在租户集群 sharecluster01 中新增租户 share1，主用户 usershare1，并为该租户配置 YARN（4GB 内存、3 核 CPU）和 HDFS（/tenant/share1 下 10G 存储空间）组件资源。

图3-12 新增租户

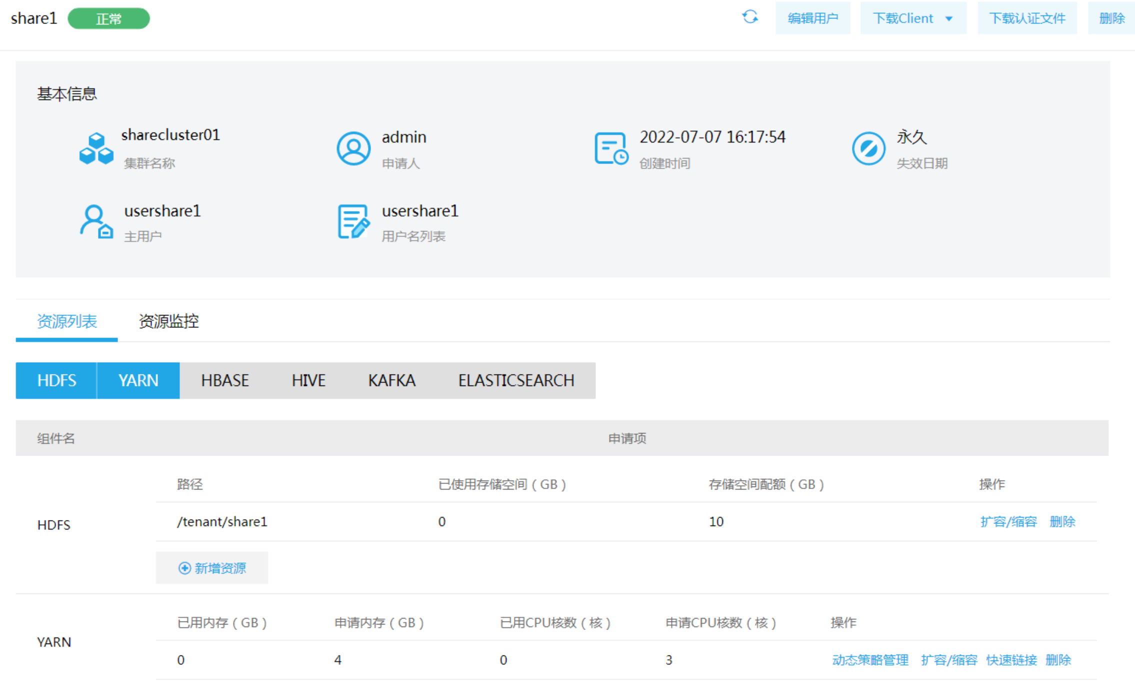


(2) 新增租户成功后，用户可在租户列表查看到已创建的租户，同时可以看到其所属集群、申请人、用户名列表、创建时间、失效时间等相关信息，如图 3-13 所示。

图3-13 查看租户

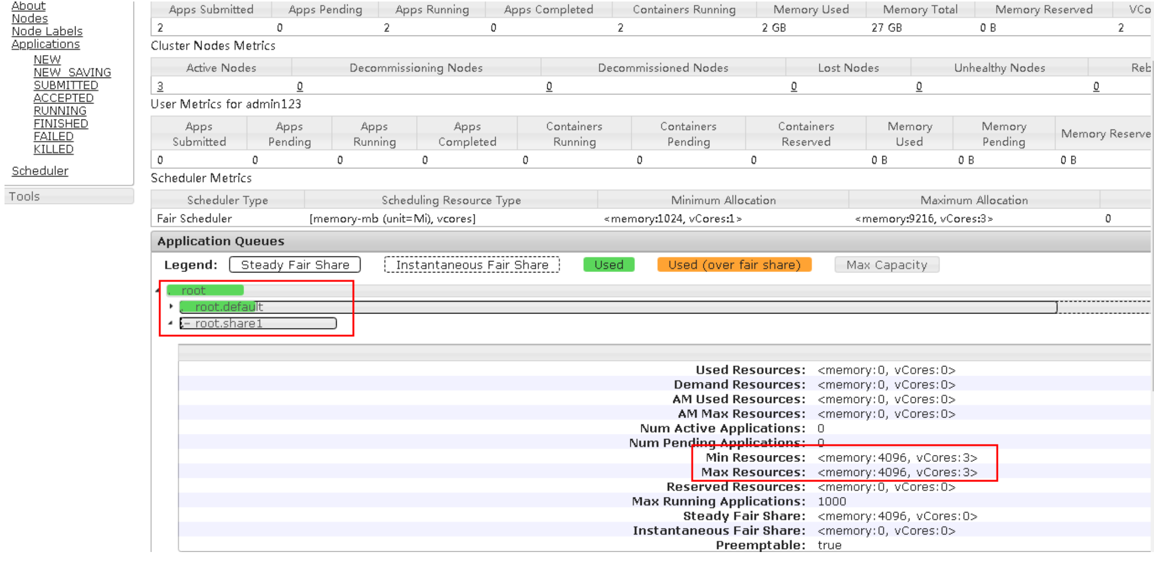
(3) 单击租户名称，可查看租户详情，如图 3-14 所示，可以看到对应的 YARN 资源队列和容量、HDFS 路径和存储空间配额等信息。用户 usershare1 就拥有 YARN 队列 share1、HDFS 目录 /tenant/share1 资源的所有权限，若资源不够/过多，可编辑租户对其进行扩容/缩容。

图3-14 查看租户详情



(4) 通过 ResourceManager UI 页面 Scheduler 可查看到，资源队列由原来的 root 下只有 default 队列变为 default 和 share1 队列，且 share1 的队列大小为 4GB 内存、3 核 CPU，如图 3-15 所示。

图3-15 查看资源队列



3.7.3 租户使用操作示例

注意

租户集群缺省开启 Kerberos 认证，在使用租户时需要通过该租户的用户对应的认证文件，对租户的用户进行身份认证。租户的用户认证文件的下载在租户列表页面或租户详情页面执行，关于对租户的用户进行认证的方式与集群中用户的身份认证方式相同，详情请参见2.3.2 1. Kerberos 环境下用户身份认证。

租户集群缺省开启 Kerberos 认证，通过租户用户对查看租户或对租户执行管理操作时，均需要提前对租户的用户执行身份认证操作。

用户 usershare1 默认提交任务到队列 share1，且拥有目录/tenant/share1 的操作权限，下面以测试 wordcount 示例。

(1) 使用用户 usershare1 在/tenant/share1 下创建 wordin 目录并上传示例数据到该目录，如图 3-16 所示。

图3-16 创建 wordin 目录并上传示例数据到该目录

```
sh-4.2$ hdfs dfs -mkdir -p /tenant/share1/wordin
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.
7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl
-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$ vi /tmp/in.txt
sh-4.2$ hdfs dfs -put /tmp/in.txt /tenant/share1/wordin
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.
7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl
-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

(2) 提交 MapReduce 任务，如图 3-17 所示；任务执行成功，如图 3-18 所示。通过 ResourceManager UI 页面可看到任务默认提交到 share1 队列，如图 3-19 所示。

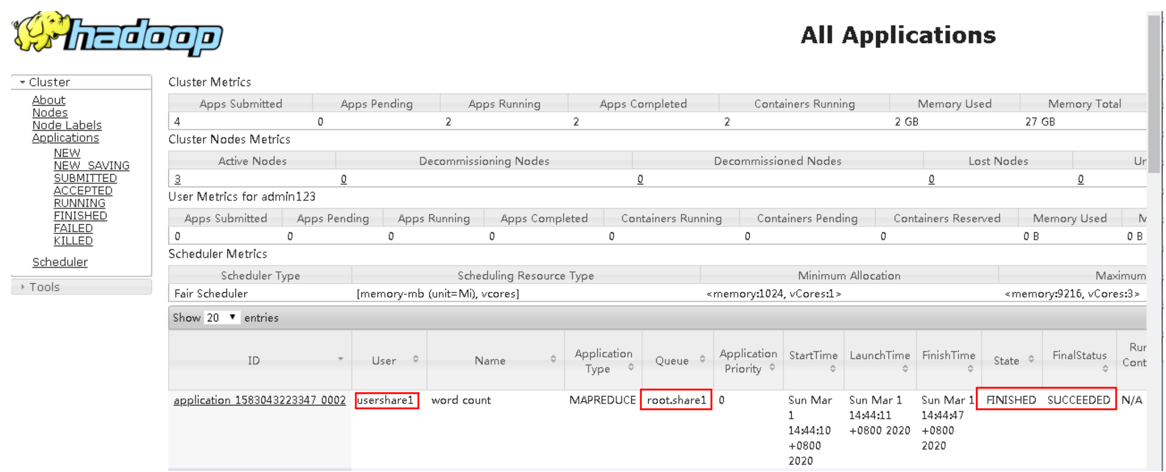
图3-17 执行 MapReduce 任务

```
sh-4.2$ yarn jar /usr/hdp/3.0.1.0-187/hadoop-mapreduce/hadoop-mapreduce-examples-3
.0.0-cdh6.2.0.jar   wordcount  /tenant/share1/wordin  /tenant/share1/wordout2
```

图3-18 执行成功

```
2020-04-10 11:17:22,983 INFO  impl.YarnClientImpl (YarnClientImpl.java:submitAppli
cation(310)) - Submitted application application_1586250381479_0012
2020-04-10 11:17:23,032 INFO  mapreduce.Job (Job.java:submit(1574)) - The url to t
rack the job: http://mh002.hde.com:8088/proxy/application_1586250381479_0012/
2020-04-10 11:17:23,033 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1619)) -
Running job: job_1586250381479_0012
2020-04-10 11:17:33,413 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1640)) -
Job job_1586250381479_0012 running in uber mode : false
2020-04-10 11:17:33,438 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647)) -
 map 0% reduce 0%
2020-04-10 11:17:41,700 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647)) -
 map 100% reduce 0%
2020-04-10 11:17:46,735 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647)) -
 map 100% reduce 100%
2020-04-10 11:17:47,756 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1658)) -
Job job_1586250381479_0012 completed successfully
2020-04-10 11:17:48,303 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1665)) -
Counters: 54
```

图3-19 查看任务



3.8 YARN调度器

3.8.1 YARN 调度器类型

YARN 调度器的基本作用就是根据节点资源的使用情况和作业需求，将任务调度到各个节点上执行。调度器会根据容量、队列等限制条件（如每个队列分配一定的资源，最多执行一定数量的作业等），将系统中的资源分配给各个正在运行的应用程序。

YARN 提供了多种调度器和可配置的策略：

- FIFO Scheduler（先进先出调度器）

  把应用按提交的顺序排成一个队列，在进行资源分配的时候，先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个分配，以此类推。

- Capacity Scheduler（容器调度器）

允许将整个集群的资源分成多个部分，每个组织使用其中的一部分，即每个组织有一个专门的队列，这样整个集群就可以通过设置多个队列的方式给多个组织提供服务。除此之外，每个组织的队列还可以进一步划分成层次结构，从而允许组织内部的不同用户组的使用。

CapacityScheduler 允许多个组织共享整个集群，每个组织可以获得集群的一部分计算能力，适用于一个集群中运行多个 Application 的情况，目标是最大化吞吐量和集群利用率。

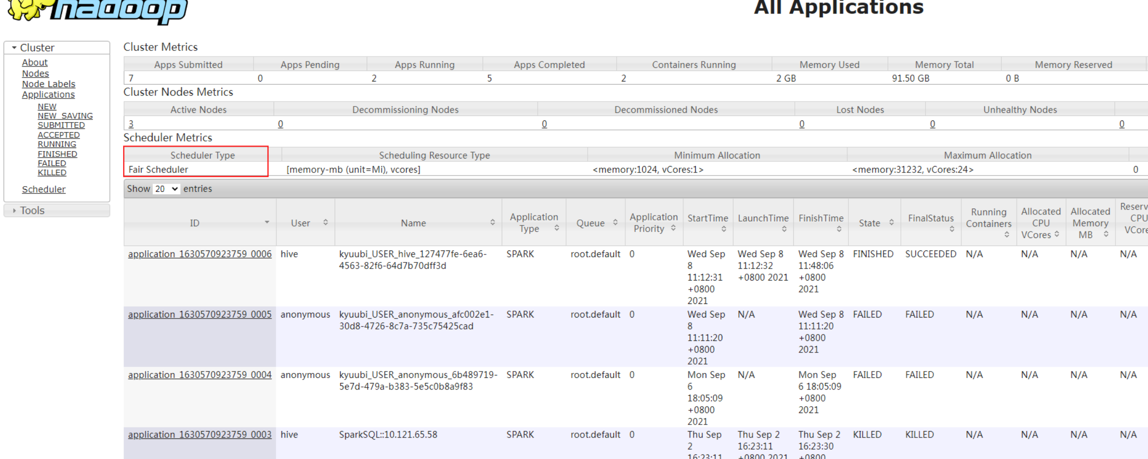
- Fair Scheduler（公平调度器）

  其设计目标是为所有的应用分配公平的资源，即每一个作业都不会预先占用资源，公平调度器会为所有运行的作业动态的调整系统资源。

3.8.2 大数据集群调度器的修改说明

在 YARN 的快速链接页面，可查看 YARN 当前调度器（Scheduler Type），如图 3-20 所示。

图3-20 查看 YARN 调度器



在大数据平台中，独立集群和租户集群中 YARN 的调度器缺省均为公平调度，其中独立集群的调度器可根据实际使用需求更改，租户集群的调度器不允许更改。

- 独立集群的调度器更改方式

  在 YARN 的配置中，在 YARN 组件详情页面的[配置]页签，选择高级配置->yarn-site，修改参数 yarn.resourcemanager.scheduler.class 的值，即可进行调度方式切换，修改说明如表 3-2 所示。调度方式修改后，需要重启 YARN 组件使配置生效。

表3-2 独立集群中 YARN 调度方式修改说明

<table>
<tr><th>参数</th><th>值</th><th>调度方式</th></tr>
<tr><td rowspan="3">yarn.resourcemanager.scheduler.class</td><td>org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler</td><td>公平调度（默认值）</td></tr>
<tr><td>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</td><td>容量调度</td></tr>
<tr><td>org.apache.hadoop.yarn.server.resourcemanager.scheduler.fifo.FifoScheduler</td><td>FIFO</td></tr>
</table>

- 租户集群的调度器禁止修改

  对于租户集群，若 YARN 的调度方式不是公平调度，则租户功能不可使用，因此租户集群的调度器禁止修改。

4 开发指南

4.1 常用 API

说明

关于 MapReduce API 更多信息，详情请参见官网 http://hadoop.apache.org/docs/stable/api/index.html。

实际的代码中，需要三个元素，分别是 Map 类、Reduce 类和运行任务的代码。其中：

- Map 类：继承了 org.apache.hadoop.mapreduce.Mapper，并实现其中的 map 方法
- Reduce 类：继承了 org.apache.hadoop.mapreduce.Reducer，并实现其中的 reduce 方法
- 运行任务的代码：程序的入口

MapReduce 中常见的类有：

- org.apache.hadoop.mapreduce.Job：用户提交 MapReduce 作业的接口，用于设置作业参数、提交作业、控制作业执行以及查询作业状态
- org.apache.hadoop.mapred.JobConf：MapReduce 作业的配置类，是用户向 Hadoop 提交作业的主要配置接口

表4-1 类 org.apache.hadoop.mapreduce.Job 的常用接口

功能	说明
Job(Configuration conf, String jobName)，Job(Configuration conf)	新建一个MapReduce客户端，用于配置作业属性、提交作业
setMapperClass(Class<extends Mapper> cls)	• 核心接口，指定 MapReduce 作业的 Mapper 类，默认为空 • 可以在“mapred-site.xml”中配置“mapreduce.job.map.class”项
setReducerClass(Class<extends Reducer> cls)	• 核心接口，指定 MapReduce 作业的 Reducer 类，默认为空 • 可以在“mapred-site.xml”中配置“mapreduce.job.reduce.class”项
setCombinerClass(Class<extends Reducer> cls)	• 指定 MapReduce 作业的 Combiner 类，默认为空 • 可以在“mapred-site.xml”中配置“mapreduce.job.combine.class”项 【说明】：reduce的输入输出key，value类型需要相同才可以使用，需谨慎使用

功能	说明
setInputFormatClass(Class<extends InputFormat> cls)	• 核心接口，指定 MapReduce 作业的 InputFormat 类，默认为 TextInputFormat • 可以在“mapred-site.xml”中配置“mapreduce.job.inputformat.class”项。该设置用来指定处理不同格式的数据时需要的 InputFormat 类，用来读取数据，切分数据块
setJarByClass(Class< > cls)	• 核心接口，指定执行类所在的 jar 包本地位置 • java 通过 class 文件找到执行 jar 包，该 jar 包被上传到 HDFS
setJar(String jar)	• 指定执行类所在的 jar 包本地位置。设置执行 jar 包所在位置，该 jar 包被上传到 HDFS。与 setJarByClass(Class< > cls)选择使用一个。 • 可以在“mapred-site.xml”中配置“mapreduce.job.jar”项
setOutputFormatClass(Class<extends OutputFormat> theClass)	• 核心接口，指定 MapReduce 作业的 OutputFormat 类，默认为 TextOutputFormat • 可以在“mapred-site.xml”中配置“mapred.output.format.class”项，指定输出结果的数据格式。例如默认的 TextOutputFormat 把每条 key，value 记录写为文本行。通常场景不配置特定的 OutputFormat
setOutputKeyClass(Class< > theClass)	• 核心接口，指定 MapReduce 作业的输出 key 的类型 • 可以在“mapred-site.xml”中配置“mapreduce.job.output.key.class”项
setOutputValueClass(Class< > theClass)	• 核心接口，指定 MapReduce 作业的输出 value 的类型 • 可以在“mapred-site.xml”中配置“mapreduce.job.output.value.class”项
setPartitionerClass(Class<extends Partitioner> theClass)	• 指定 MapReduce 作业的 Partitioner 类 • 可以在“mapred-site.xml”中配置“mapred.partitioner.class”项。该方法用来分配 map 的输出结果到哪个 reduce 类，默认使用 HashPartitioner，均匀分配 map 的每条键值对记录。例如在 hbase 应用中，不同的键值对应的 region 不同，这就需要设定特殊的 partitioner 类分配 map 的输出结果
setSortComparatorClass(Class<extends RawComparator> cls)	• 指定 MapReduce 作业的 map 任务的输出结果压缩类，默认不使用压缩 • 可以在“mapred-site.xml”中配置“mapreduce.map.output.compress”和“mapreduce.map.output.compress.codec”项。当 map 的输出数据大，减少网络压力，使用压缩传输中间数据
setPriority(JobPriority priority)	• 指定 MapReduce 作业的优先级，共有 VERY_HIGH、HIGH、NORMAL、LOW 和 VERY_LOW5 个级别，默认为 NORMAL • 可以在“mapred-site.xml”中配置“mapreduce.job.priority”项

表4-2 类 org.apache.hadoop.mapred.JobConf 的常用接口

功能	说明
setNumMapTasks(int n)	• 核心接口，指定 MapReduce 作业的 map 个数 • 可以在“mapred-site.xml”中配置“mapreduce.job.maps”项 【说明】：指定的InputFormat类用来控制map任务个数，注意该类是否支持客户端设定的map个数
setNumReduceTasks(int n)	• 核心接口，指定 MapReduce 作业的 reduce 个数。默认只启动 1 个 • 可以在“mapred-site.xml”中配置“mapreduce.job.reduces”项。 【说明】：reduce个数由用户控制，通常场景reduce个数是map个数的1/4
setQueueName(String queueName)	• 指定 MapReduce 作业的提交队列。默认使用 default 队列 • 可以在“mapred-site.xml”中配置“mapreduce.job.queuename”项

4.2 MapReduce统计词频示例

下面为词频统计示例，一个 MapReduce 作业的输入和输出类型为：

(input)<k1,v1> —> map —> <k2,v2> —> 汇总数据 —> <k2,List(v2)> —> reduce —> <k3,v3>(output)

1. 建立 maven 工程

Pom.xml 文件内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.test.mapreduce</groupId>
    <artifactId>mapreducedemo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--配置项目中某些 jar 包的版本-->
    <properties>
        <hadoop.version>3.0.0-cdh6.2.0</hadoop.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
```

```
            <version>${hadoop.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>${hadoop.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>${hadoop.version}</version>
        </dependency>
    </dependencies>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <artifactId>maven-clean-plugin</artifactId>
                    <version>3.0.0</version>
                </plugin>
                <plugin>
                    <artifactId>maven-resources-plugin</artifactId>
                    <version>3.0.2</version>
                </plugin>
                <plugin>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.7.0</version>
                    <configuration>
                        <encoding>utf8</encoding>
                    </configuration>
                </plugin>
                <plugin>
                    <artifactId>maven-jar-plugin</artifactId>
                    <version>3.0.2</version>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>

</project>
```

2. 未开启 Kerberos 代码样例

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

```
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.util.GenericOptionsParser;

import java.io.IOException;
import java.util.StringTokenizer;
/**
 * 词频统计示例代码
 */
public class WordCount {
    public WordCount(){}
    /***
     * main()方法创建一个 job，指定参数，提交作业到 hadoop 集群
     */
    public static void main(String[] args) throws Exception{
        // 初始化环境变量
        Configuration conf = new Configuration();

        // 获取入参
        String[] otherArgs = (new GenericOptionsParser(conf, args)).getRemainingArgs();
        if(otherArgs.length < 2){
            System.err.println("Usage: wordcount <in> [<in>...] <out>");
            System.exit(2);
        }
        // 初始化 Job 任务对象
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);

        // 设置运行时执行 map，reduce 的类
        job.setMapperClass(WordCount.TokenizerMapper.class);
        // 设置 combiner 类，默认不使用，使用时通常使用和 reduce 一样的类
        job.setCombinerClass(WordCount.IntSumReducer.class);
        job.setReducerClass(WordCount.IntSumReducer.class);

        // 设置作业的输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

        // 提交任务交到远程环境上执行
        System.exit(job.waitForCompletion(true)?0:1);
    }

    /***
     * 类 IntSumReducer 定义 Reducer 抽象类的 reduce()方法
     */
```

```
    public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable>{
        // 统计结果
        private IntWritable result = new IntWritable();

        public IntSumReducer(){}

        /***
         *
         * @param key Text : Mapper 后的 key 项
         * @param values Iterable : 相同 key 项的所有统计结果
         * @param context
         * @throws IOException
         * @throws InterruptedException
         */
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException,InterruptedException{
            int sum =0;
            for(IntWritable val :values){
                sum +=val.get();
            }
            result.set(sum);
            // reduce 输出为 key：字符串，value：字符串出现的总数
            context.write(key,result);
        }
    }

    /***
     * 类 TokenizerMapper 定义 Mapper 抽象类的 map()方法
     */
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        // 输出的 key,value 要求是序列化的
        private static final IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public TokenizerMapper(){}

        /***
         *
         * @param key Object : 原文件位置偏移量
         * @param value Text : 原文件的字符串数据
         * @param context Context : 出参
         * @throws IOException
         * @throws InterruptedException
         */
        public void map(Object key, Text value, Context context) throws IOException,
InterruptedException{
            // 读取的一行字符串数据，按分隔符进行分割
            StringTokenizer itr = new StringTokenizer(value.toString(),",");

            while (itr.hasMoreTokens()){
```

```
                word.set(itr.nextToken());
                // map 输出 key，value 键值对
                context.write(word, one);
            }
        }
    }
}
```

3. 开启 Kerberos 代码样例

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.security.UserGroupInformation;
import org.apache.hadoop.util.GenericOptionsParser;
import java.io.IOException;
import java.util.StringTokenizer;

/**
 * 词频统计示例代码
 */
public class WordCount {
    public WordCount(){}

    /***
     * main()方法创建一个 job，指定参数，提交作业到 hadoop 集群
     */
public static void main(String[] args) throws Exception{
      final String PATH_TO_HDFS_SITE_XML="/etc/hadoop/conf/hdfs-site.xml";
      final String PATH_TO_CORE_SITE_XML="/etc/hadoop/conf/core-site.xml";
final String JVM_KRB5_CONF_PARM="java.security.krb5.conf";
  final String JVM_KRB5_CONF_PARM_VALUE="/etc/krb5.conf";
  final String USER_PRINCIPAL="diao2@HDE.TEST.COM";
  final String PATH_TO_USER_KEYTAB="/etc/security/keytabs/diao2.keytab";
  // 初始化环境变量
     Configuration conf = new Configuration();
  conf.addResource(new Path(PATH_TO_HDFS_SITE_XML));
  conf.addResource(new Path(PATH_TO_CORE_SITE_XML));

 // kerberos 身份认证
      System.setProperty(JVM_KRB5_CONF_PARM, JVM_KRB5_CONF_PARM_VALUE);
      UserGroupInformation.setConfiguration(conf);
  // 安全登录，指定登录用户的票据与 keytab
  UserGroupInformation.loginUserFromKeytab(USER_PRINCIPAL,PATH_TO_USER_KEYTAB);
```

```
        System.out.println("kerberos auth success");
      // 获取入参
        String[] otherArgs = (new GenericOptionsParser(conf, args)).getRemainingArgs();
        if(otherArgs.length < 2){
            System.err.println("Usage: wordcount <in> [<in>...] <out>");
            System.exit(2);
        }
        // 初始化 Job 任务对象
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);

       // 设置运行时执行 map，reduce 的类
        job.setMapperClass(WordCount.TokenizerMapper.class);
       // 设置 combiner 类，默认不使用，使用时通常使用和 reduce 一样的类
        job.setCombinerClass(WordCount.IntSumReducer.class);
        job.setReducerClass(WordCount.IntSumReducer.class);

        // 设置作业的输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

        // 提交任务交到远程环境上执行
        System.exit(job.waitForCompletion(true)?0:1);
    }

    /***
     * 类 IntSumReducer 定义 Reducer 抽象类的 reduce()方法
     */
    public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable>{
        // 统计结果
        private IntWritable result = new IntWritable();

        public IntSumReducer(){}

        /***
         *
         * @param key Text : Mapper 后的 key 项
         * @param values Iterable : 相同 key 项的所有统计结果
         * @param context
         * @throws IOException
         * @throws InterruptedException
         */
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException,InterruptedException{
            int sum =0;
            for(IntWritable val :values){
                sum +=val.get();
```

```
            }
            result.set(sum);
            // reduce 输出为 key：字符串，value：字符串出现的总数
            context.write(key,result);
        }
    }

    /***
     * 类 TokenizerMapper 定义 Mapper 抽象类的 map()方法
     */
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        // 输出的 key,value 要求是序列化的
        private static final IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public TokenizerMapper(){}

        /***
         *
         * @param key Object : 原文件位置偏移量
         * @param value Text : 原文件的字符串数据
         * @param context Context : 出参
         * @throws IOException
         * @throws InterruptedException
         */
        public void map(Object key, Text value, Context context) throws IOException,
InterruptedException{
            // 读取的一行字符串数据，按分隔符进行分割
            StringTokenizer itr = new StringTokenizer(value.toString(),",");

            while (itr.hasMoreTokens()){
                word.set(itr.nextToken());
                // map 输出 key，value 键值对
                context.write(word, one);
            }
        }
    }
}
```

4.3 作业提交示例

作业提交操作步骤如下：

(1) 将 4.2 MapReduce 统计词频示例中的代码编译打包，生成 jar 包（mapreducedemo-1.0-SNAPSHOT.jar）。

(2) 上传生成的 jar 包“mapreducedemo-1.0-SNAPSHOT.jar”到 Linux 客户端上。

(3) 执行如下命令，提交作业：

yarn jar mapreducedemo-1.0-SNAPSHOT.jar.jar com.test.mapreduce.WordCount <inputPath> <outputPath>

说明，该命令包含了入口类以及设置参数和提交 job 的操作，其中：

- com.test.mapreduce：为示例程序 WordCount 存放的包名
- <inputPath>：HDFS 文件系统中 input 的路径
- <outputPath>：HDFS 文件系统中 output 的路径且不存在

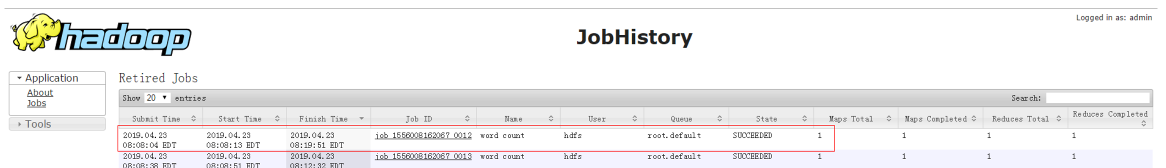
4.4 作业监控

MapReduce 应用程序运行完成后，可通过以下 3 种方式查看应用程序的运行情况。

1. JobHistory UI

参见 2.4.2 访问 MapReduce 快速链接章节，进入 JobHistory Web 页面，可以查看应用程序的运行情况。如图 4-1 所示，红框内可以查看当前的 WordCount 作业 ID 以及运行结果等信息。

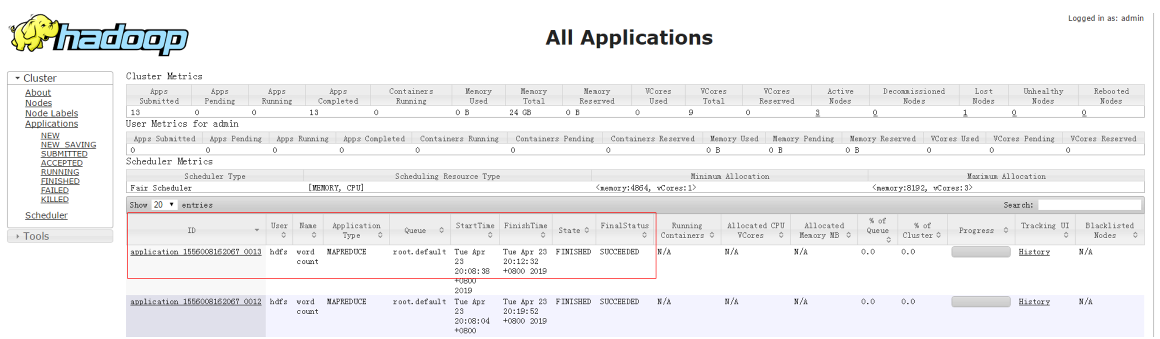
图4-1 JobHistory Web UI 界面



2. ResourceManager UI

参见 2.4.3 访问 YARN 快速链接章节，进入 All Applications Web 页面，可以查看任务执行状态。如图 4-2 所示，红框内可以查看当前的 WordCount 作业 ID 以及运行结果等信息。

图4-2 ResourceManager Web UI 界面



3. 应用程序运行结果

使用 ***yarn jar*** 命令后，可以在控制台查看应用程序的运行情况。如图 4-3 所示，红框内表示作业运行成功。

图4-3 应用程序运行结果

```
2020-04-10 11:17:22,983 INFO  impl.YarnClientImpl (YarnClientImpl.java:submitAppli
cation(310)) - Submitted application application_1586250381479_0012
2020-04-10 11:17:23,032 INFO  mapreduce.Job (Job.java:submit(1574)) - The url to t
rack the job: http://mh002.hde.com:8088/proxy/application_1586250381479_0012/
2020-04-10 11:17:23,033 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1619)) -
Running job: job_1586250381479_0012
2020-04-10 11:17:33,413 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1640)) -
Job job_1586250381479_0012 running in uber mode : false
2020-04-10 11:17:33,438 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647)) -
 map 0% reduce 0%
2020-04-10 11:17:41,700 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647)) -
 map 100% reduce 0%
2020-04-10 11:17:46,735 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1647)) -
 map 100% reduce 100%
2020-04-10 11:17:47,756 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1658)) -
Job job_1586250381479_0012 completed successfully
2020-04-10 11:17:48,303 INFO  mapreduce.Job (Job.java:monitorAndPrintJob(1665)) -
Counters: 54
```

5 常见问题解答

5.1 调优

1. YARN 资源分配调优

Container 是 YARN 资源分配的基本单元，YARN 作为资源调度器，应该考虑每个节点的计算资源能力，根据任务申请的资源进行分配 Container。资源分配性能调优相关参数如表 5-1 所示。

表5-1 YARN 资源分配调优参数

参数	描述	默认值
yarn.nodemanager.resource.memory-mb	节点最大可用内存	12288MB
yarn.scheduler.minimum-allocation-mb	分配给AM单个容器可申请的最小内存，可以计算一个节点最大Container数量	1024MB
yarn.scheduler.maximum-allocation-mb	分配给AM单个容器可申请的最大内存	8192MB
yarn.nodemanager.resource.cpu-vcores	节点最大可用vcores	3
yarn.nodemanager.vmem-pmem-ratio	虚拟内存率	2.1
mapreduce.map.memory.mb	分配给map Container的内存大小	1024MB
mapreduce.reduce.memory.mb	分配给reduce Container的内存大小	1024MB
mapreduce.map.java.opts	运行map任务的JVM参数	-
mapreduce.reduce.java.opts	运行reduce任务的JVM参数	-

2. 确定 Job 基线

确定 Job 基线是调优的基础，一切调优项效果的检查，都是通过与基线数据做对比来获得。

Job 基线的确定有以下三个原则：

(1) 充分利用集群资源

Job 运行时，会让所有的节点都有任务处理，且处于繁忙状态，这样才能保证资源充分利用，任务的并发度达到最大。可以调整处理的数据量大小、Map 个数和 Reduce 个数来实现。

- Reduce 个数使用“mapreduce.job.reduces”参数控制。
- Map 个数取决于使用了哪种 InputFormat，以及待处理的数据文件是否可分割。默认的 TextFileInputFormat 将根据 Block 的个数来分配 Map 数，通常一个 Block 对应一个 Map。

可以通过如表 5-2 所示的配置参数进行调优。

表5-2 配置参数说明

参数	描述	默认值
mapreduce.input.fileinputformat.split.maxsize	• 可以设置数据分片的数据最大值 • 由用户定义的分片大小的设置及每个文件 block 大小的设置，可以计算分片的大小	-

参数	描述	默认值
	计算公式如下： splitSize = Math.max(minSize, Math.min(maxSize, blockSize)) **说明：** 如果maxSize设置大于blockSize，那么每个block就是一个分片，否则就会将一个block文件分隔为多个分片，如果block中剩下的一小段数据量小于splitSize，还是认为它是独立的分片	
mapreduce.input.fileinputformat.split.minsize	设置数据分片的数据最小值	0

(2) 避免数据倾斜

在一个作业的运行过程中，某些节点计算量少的 Reduce 任务完成，而某些节点计算量大的 Reduce 任务还在运行，将会延长作业的完成时间，这种情况通常是因为数据分布不均导致的，会带来数据倾斜。可以通过以下方法避免数据倾斜：

- 自定义 partitioner，使分配到多个 reduce 任务的数据量大致相等。
- 增加 reduce 个数。

(3) 每个 task 的执行时间要合理

如果一个 job，每个 Map 或 Reduce 的执行时间只有几秒钟，意味着这个 job 的大部分时间都消耗在 task 的调度和进程启停上了，此时需要增加每个 task 处理的数据大小。建议一个 task 处理时间为 1 分钟。控制单个 task 处理时间的大小，可参考表 5-2 中的参数配置来调整。

3. Shuffle 调优

Shuffle 阶段是 MapReduce 性能的关键部分，包括从 Map task 将中间数据写入磁盘、Reduce task 拷贝数据以及放到 Reduce 函数的全部过程。Hadoop 提供了如下三个阶段的参数调优。

(1) Map 阶段的调优

- 判断 Map 使用的内存大小

  判断 Map 分配的内存是否足够，一个简单的办法是查看运行完成的 job 的 Counters 中，对应的 task 是否发生过多次 GC，以及 GC 时间占总 task 运行时间之比。

  说明：GC 时间不应超过 task 运行时间的 10%，即 GC time elapsed (ms)/CPU time spent (ms)<10%。

表5-3 Shuffle 调优参数

参数	描述	默认值
mapreduce.map.memory.mb	设置Map调度内存	4096
mapreduce.map.java.opts	设置Map进程JVM参数 **建议：**配置该参数中“-Xmx”值为“mapreduce.map.memory.mb”参数值的0.8倍	-Xmx2048M -Djava.net.preferIPv4Stack=true

- 使用 Combiner

在 Map 阶段，combiner 过程将同一个 key 值的中间结果合并，该过程可选。一般将 reduce 类设置为 combiner。通过 combiner，可以显著减少 Map 输出的中间结果，从而减少 shuffle 过程的网络带宽占用。

(2) Copy 阶段的调优

Copy 阶段的调优主要是对 Map 阶段的计算结果进行压缩，以减少网络传输的数据量，但由于多了压缩和解压过程，带来了更多的 CPU 消耗。因此需要做好权衡。当任务属于网络瓶颈类型时，采用压缩数据的方式会有比较明显的效果。

针对 bulkload 调优，压缩中间结果后性能可以提升 60%左右。配置方法如下：

- 将“mapreduce.map.output.compress”参数值设置为“true”
- 将“mapreduce.map.output.compress.codec”参数值设为“org.apache.hadoop.io.compress.SnappyCodec”。

(3) Merge 阶段的调优

调整如表 5-4 所示的参数可以减少 reduce 写磁盘的次数。

表5-4 Merge 调优参数

参数	描述	默认值
mapreduce.reduce.merge.inmem.threshold	允许多少个文件同时存在reduce内存里。当达到这个阈值时，reduce就会触发mergeAndSpill，将数据写到硬盘上。	1000
mapreduce.reduce.shuffle.merge.percent	当reduce中存放map中间结果的buffer使用达到多少百分比时，会触发merge操作。	0.66
mapreduce.reduce.shuffle.input.buffer.percent	允许map中间结果占用reduce堆大小的百分比。	0.70
mapreduce.reduce.input.buffer.percent	当开始执行reduce函数时，允许map文件占reduce堆大小的百分比。 当map文件比较小时，可以将这个值设置成1.0，这样可以避免reduce将拷贝过来的map中间结果写入磁盘。	0

5.2 运维类问题

1. 公平调度中，若 ResourceManager 发生切换或者重启，对于执行结束的 application，其队列为什么会从申请的队列切换到 default 队列？

ResourceManager 切换或者重启后，ResourceManager 会从它的元数据库遍历所有已提交的任务。

- 如果任务未执行结束，则根据调度策略，决定任务提交到哪个队列执行。
- 如果任务已执行完，在遍历时不会再调用调度策略选择队列，默认会选择 default 队列。

因此，会出现执行结束的 application，其队列会从申请队列切换到 default 队列，但不会影响未结束任务的正常执行。

2. 任务一直处于 accept 状态，无法切换到 running 状态

- 可能原因

  YARN 资源不足导致。

- 解决方法

  查看 YARN 资源参数配置，如 vcores 是否充足，给相应租户的 YARN 队列添加资源。

3. YARN UI 页面出现 YARN 告警

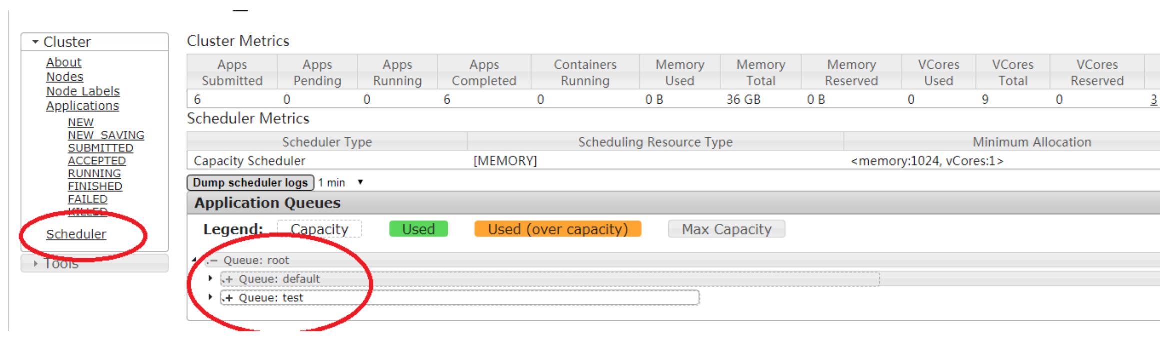
- 可能原因

  YARN 的某个队列资源使用情况超过或临近资源配置。

- 解决方法

  通过 YARN UI 页面查看资源队列使用情况，如图 5-1 所示。检查下各个队列的使用情况，如果向队列添加资源，最好通过租户管理添加，不要通过 YARN 配置文件更改。

图5-1 YARN UI 页面



4. YARN UI 启动失败

- 现象描述

  在大数据平台管理系统通过快速连接进入 YARN UI，提示无法访问此网站。

- 可能原因

  tomcat 异常停止。

- 解决方法

  到对应节点执行 ***service tomcatd status*** 命令，查看 tomcatd 是否启动。如果未启动，使用 service tomcatd start 命令启动 tomcatd。

5. YARN UI 登录失败

- 现象描述

  在大数据平台管理系统通过快速连接进入 YARN UI，通过用户名、密码登录时报错。

- 可能原因

  krb5、kadmin 异常停止。

- 解决方法

  到对应节点执行 service krb5kdc status、service kadmin status，查看状态否正常。如果为启动，则执行如下命令分别启动 krb5kdc 和 kadmin。

  service krb5kdc start

  service kadmin start

6. YARN 任务较多时，点击 YARN UI 界面卡顿

- 现象描述

UI 界面有时能打开，有时打不开

- 可能原因

  ResourceManager 进程异常导致。

- 解决方法

  使用 jmap 堆栈分析发现 resourcemanager 的堆空间只配置了 1G，使用率接近 100%。根据数据量和系统资源情况，在 YARN 配置界面，增大 resourcemanager 的堆内存，推荐为 4G。

7. 执行 MapReduce 程序过程中报 OOM

- 可能原因

  该现象通常是 MapReduce 的 task 任务内存不足导致。

- 解决方法

  检查日志，并根据 MapReduce 的执行阶段来判断异常。如果是 Map 阶段抛出异常，则调大 mapreduce.map.memory.mb，如果是 Reduce 阶段抛出 oom 异常，则调大 mapreduce.reduce.memory.mb 大小。

8. 执行 Hadoop 或 Spark 任务时，指定 yarn 队列名称为 root.default，任务报错：Failed to submit application_xxxx to YARN: Application application_xxxx submitted by user XXX to unKnown queue: root.default

- 可能原因

  YARN 组件采用容量调度时，任务提交时指定队列时无需添加根队列信息，即不能带 root。

- 解决方法

  提交任务指定队列名称为：default 即可。

9. 执行 Spark 任务时，使用集群超级用户登录 YARN UI，点击 ApplicationMaster 跳转 Spark 的 UI 页面，上报权限不足问题

- 可能原因

  只有提交任务的用户才可查看对应任务的 Spark UI 页面。

- 解决方法

  使用提交任务的用户登录 YARN UI。

10. NodeManager 频繁挂断，一直处于不可用状态，查看后台日志，报错：java.io.IOException: org.iq80.leveldb.DBException:org.fusesource.leveldbjni.internal.NativeDB$DBException: Invalid argument: notan sstable (bad magic number)

- 可能原因

  因主机异常断电等原因，导致应用状态文件损坏，无法恢复。

- 解决方法

  删除当前节点持久化目录即可，再次启动时，会重建一个新目录 /var/de_log/hadoop-yarn/nodemanager/recovery-state/yarn-nm-state。

目 录

1 组件简介 ······ 1-1
1.1 组件概述 ······ 1-1
1.2 基本概念 ······ 1-1
1.3 组件架构 ······ 1-2
1.4 应用场景 ······ 1-3
2 快速入门 ······ 2-1
2.1 组件安装 ······ 2-1
2.1.1 查看组件的日志信息 ······ 2-1
2.2 运行状态监控 ······ 2-1
2.2.1 查看组件详情 ······ 2-1
2.2.2 组件检查 ······ 2-2
2.3 快速使用指导 ······ 2-4
2.3.1 Spark Shell ······ 2-4
2.3.2 Spark Submit ······ 2-5
2.3.3 Spark SQL ······ 2-6
2.3.4 Spark Thriftserver（非 Kerberos 环境） ······ 2-7
2.3.5 Spark Thriftserver（Kerberos 环境） ······ 2-8
2.4 快速链接 ······ 2-11
2.4.1 配置组件快速链接 ······ 2-11
2.4.2 Spark 任务监控 ······ 2-11
3 使用指南 ······ 3-1
3.1 Spark SQL ······ 3-1
3.1.1 Spark 创建 Hive 表 ······ 3-1
3.1.2 Spark 创建数据源表 ······ 3-2
3.2 Spark Thriftserver HA ······ 3-5
3.3 Spark Livy ······ 3-6
3.4 Client 下载/安装/使用/卸载 ······ 3-7
3.4.1 下载 Client 安装包 ······ 3-8
3.4.2 安装 Client ······ 3-9
3.4.3 访问组件 ······ 3-10
3.4.4 使用客户端 Client ······ 3-10
3.4.5 卸载 Client 客户端 ······ 3-11

3.5 添加/删除进程 3-11
3.5.1 添加进程 3-11
3.5.2 删除进程 3-13
3.6 权限访问控制 3-14
3.6.1 权限操作示例 3-15
3.7 备份恢复 3-21
3.8 Hudi 表操作 3-22
3.8.1 Spark Shell 操作 Hudi 表 3-23
3.8.2 Spark SQL 操作 Hudi 表 3-27
4 开发指南 4-1
4.1 RDD 操作 4-1
4.1.1 RDD 的创建方式 4-1
4.1.2 RDD 的常用方法 4-1
4.1.3 示例：WordCount 代码编写 4-3
4.2 DataSet 操作 4-6
4.2.1 Pom 依赖 4-6
4.2.2 代码实现 4-6
4.2.3 创建样例文件 4-6
4.2.4 提交任务 4-7
4.3 JDBC/ODBC 连接 Spark SQL 4-7
4.3.1 pom 依赖 4-7
4.3.2 代码实现 4-7
4.3.3 创建样例 4-8
4.3.4 提交任务 4-8
5 最佳实践 5-1
5.1 未开启 Kerberos 环境下 Spark-HBase 实践案例 5-1
5.2 Kerberos 环境下 Spark-HBase 实践案例 5-3
5.3 SparkStreaming-Kafka 实践案例 5-8
6 常见问题解答 6-1
6.1 调优类 6-1
6.1.1 增加并行度 6-1
6.1.2 序列化和数据压缩 6-1
6.1.3 文件格式 6-2
6.1.4 使用广播变量 6-2
6.1.5 优化数据结构 6-2
6.1.6 对多次使用的 RDD 进行持久化 6-2

6.1.7 缓存表 6-2
6.2 运维类 6-3

1 组件简介

1.1 组件概述

Spark 是基于内存的分布式计算框架。相对于 MapReduce 的批量计算、迭代型计算，Spark 可以带来上百倍的性能提升。Spark 提供了全方位的软件栈，只要掌握 Spark 使用的编程语言就可以编写不同应用场景的应用程序（批处理、流计算、机器学习、图计算等）。

图1-1 Spark 软件栈

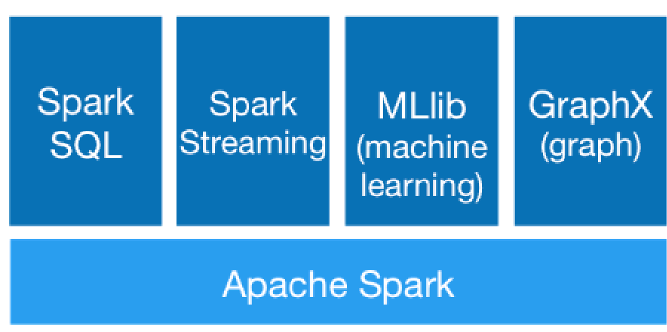


表1-1 Spark 功能模块

模块名称	功能
Spark Core	Spark的基本功能，包含任务调度、内存管理、容错机制等，内部定义了RDDs（弹性分布式数据集），提供了很多API来创建和操作这些RDDs。为其他组件提供底层的服务
Spark SQL	Spark处理结构化数据的库，就像Hive SQL、MySQL一样，企业中用来做报表统计
Spark Streaming	实时数据流处理组件
MLlib（Machine learning lib）	一个包含通用机器学习功能的包
Graphx（graph）	处理图的库（例如，社交网络图），并进行图的并行计算

说明

本文介绍内容基于大数据平台中提供的 Spark2 组件（Spark 2.4.0）。

1.2 基本概念

- Application

  Spark 用户程序，用户提交一次应用为一个 Application。Application 由一个 Driver Program 和多个 Executor 组成。

- Job

  一个 Application 可被划分为多个 Job，每个 action 类的算子（如 collect、count 等）会产生一个 Job。
- Stage

  每个 Job 会被划分多个 Stage，以 shuffle 为边界进行划分，Stage 按照血统关系依次执行。每个 Stage 是一个 Task 集合，由 DAG 分割而成。
- Task

  Task 是 Spark 最小的执行单位。
- Driver Program

  Driver Program 是 Application 的一部分，负责 Job 的初始化，将 Job 转换成 Task 并提交执行。
- DAGScheduler

  根据 Job 构建基于 Stage 的 DAG（Directed Acyclic Graph，有向无环图），并提交 Stage 给 TaskScheduler。
- TaskScheduler

  TaskScheduler 维护所有 TaskSet，当 Executor 向 Driver 发送心跳时，TaskScheduler 会根据其资源剩余情况分配相应的 Task。另外 TaskScheduler 还维护着所有 Task 的运行状态，重试失败的 Task。
- BlockManager

  负责存储管理、创建和查找块。
- Executor

  Executor 是 Application 运行在 Worker 节点上负责运行 Task 的进程，生命周期和 Application 相同。一个 Spark 应用一般包含多个 Executor，每个 Executor 接收 Driver 的命令，并执行一个或多个 Task。
- Worker Node

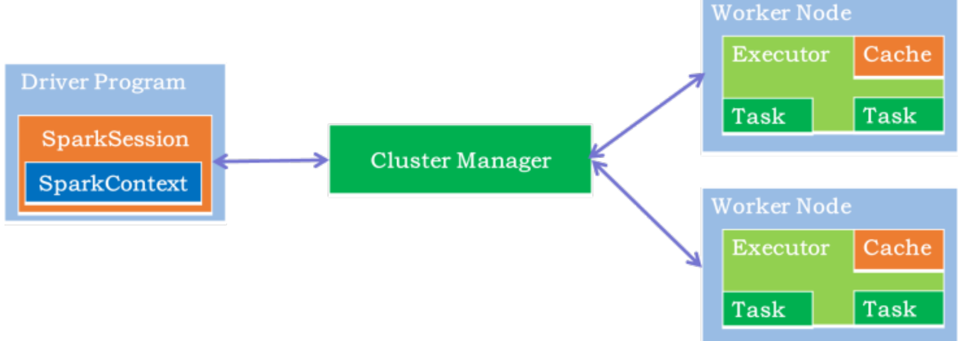
  集群中负责启动并管理 Executor 以及资源的节点。
- Cluster Manager

  集群资源管理器。Spark 支持多种集群管理器，Spark 自带的 Standalone 集群管理器或 YARN 等，大数据平台中的 Spark 集群默认采用 YARN 模式。

1.3 组件架构

Spark 应用程序的运行架构由三部分组成，包括 Driver Program（驱动程序）、Cluster Manager（集群资源管理器）和 Executor（任务执行进程）组成，其执行流程如图 1-2 所示。

图1-2 Spark 通用框架



Spark 执行流程说明：

(1) Spark 应用程序在 Driver 上运行，通过在 main 中定义 SparkSession 来协调。

(2) SparkSession 向资源管理器 Cluster Manager（YARN）申请运行 Executor 资源。

(3) Executor 与 Driver 通信。

(4) Driver 端中，启动应用程序 DAG 调度、Stage 划分、TaskSet 生成，并通过 Task Scheduler 调度 Taskset，将 Task 发放给 Executor 运行。

(5) Executor 端，将 Task 运行结果返回给 Driver 端。

(6) 所有 Task 执行完成后，应用程序运行完毕，释放所有资源。

1.4 应用场景

- 数据处理

  Spark 拥有强大的计算能力，并且兼具容错性和可扩展性，使得它可以用来快速处理数据。

- 迭代计算

  Spark 使用内存存储，减少了磁盘 I/O 时间，因此可以很好的支持需要迭代计算的场景，有效应对多步数据处理。

- 数据挖掘

  Spark 支持数据挖掘和机器学习，可以在海量数据基础上进行复杂的挖掘分析。

2 快速入门

2.1 组件安装

说明

- 在 Hadoop 集群中，安装 Spark 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- 安装 Spark 组件之前，需要安装 HDFS、MapReduce2、YARN、Zookeeper、Hive。

2.1.1 查看组件的日志信息

表2-1 组件日志路径说明

组件	日志路径
Spark	/var/de_log/spark2和/var/de_log/spark2/user_spark/ 【说明】上述的日志是Spark的HistoryServer和ThriftServer的日志存放路径。另外： • 使用 Spark 客户端（如使用 spark-sql）执行任务时，日志存放路径为 /var/de_log/spark2/user_${user.name}/，其中${user.name}是指执行任务的用户名

2.2 运行状态监控

2.2.1 查看组件详情

进入 Spark 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、概要、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

(1) 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。

(2) 概要：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。

(3) 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

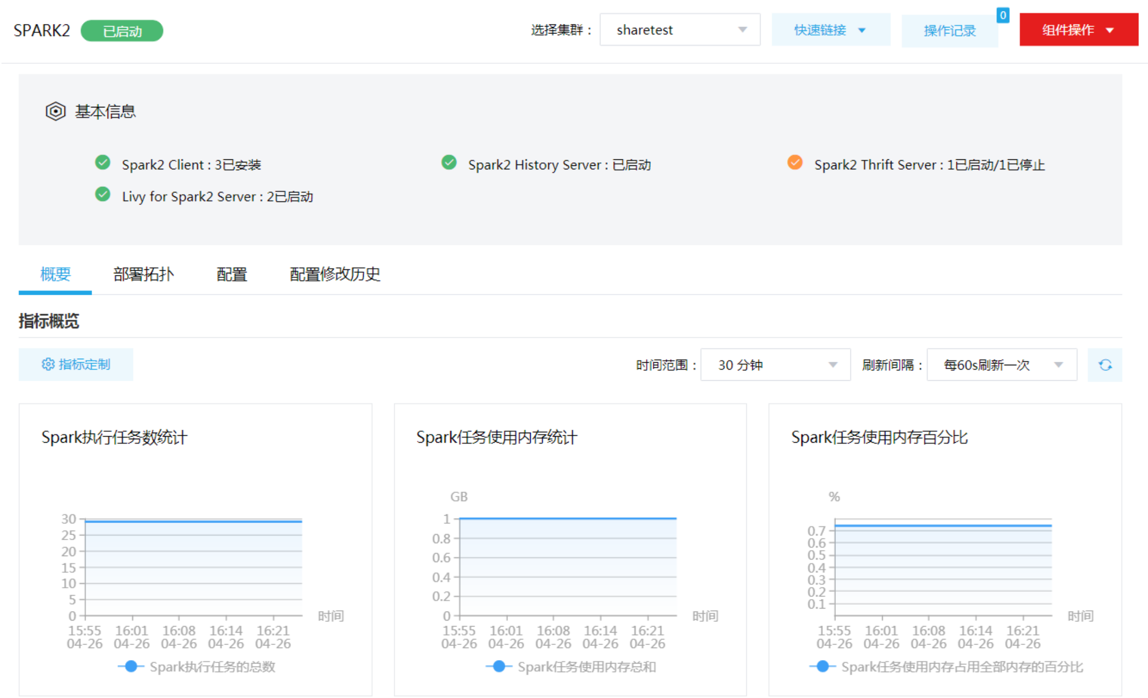
【说明】进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。

(4) 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。

(5) 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。

(6) 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 组件详情



2.2.2 组件检查

执行 Spark 组件检查时，会检查 Livy2 Server 和 Spark2 Jobhistory Server 是否可以正常连接。

集群在使用过程中，根据实际需要，可对 Spark 执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：

- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Spark 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Spark 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

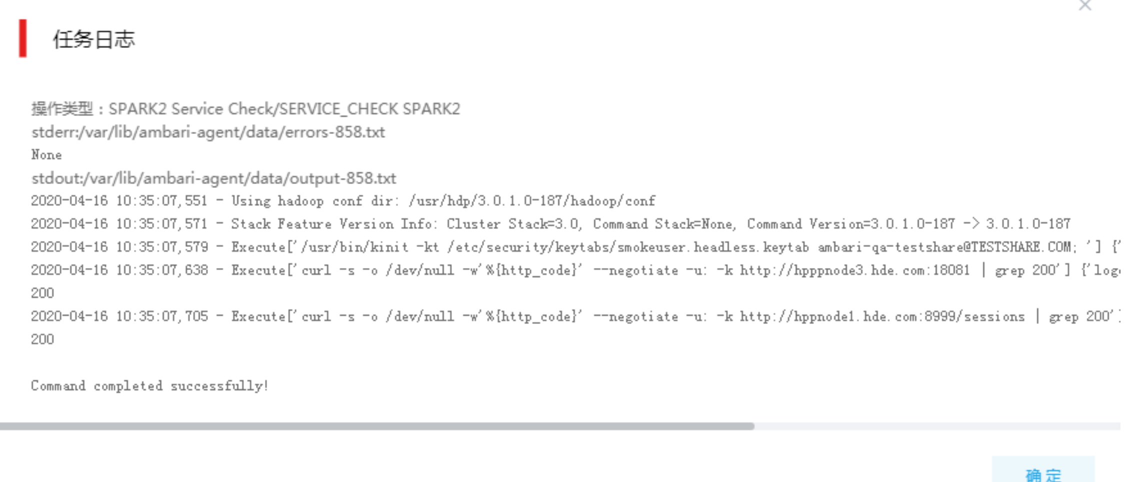
(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗中会显示组件检查成功或失败的状态。如图 2-2 所示，表示该组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“Spark2 Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情



2.3 快速使用指导

注意

- 根据大数据集群是否开启 Kerberos 认证，用户访问 Spark 时的认证方式不同，详情请参见本章节内容。
- Kerberos 环境下，若想访问 Spark 并对 Spark 执行管理操作，则必须首先进行用户身份认证，认证方式请参见 2.3.5 1. Kerberos 环境下用户身份认证。非 Kerberos 环境下，不需要用户做身份认证即可直接对 Spark 执行管理操作。
- 在租户集群中，普通用户提交 Spark 任务至 YARN 之前，需提前申请对应的 YARN 资源队列。关于 YARN 资源队列的申请及使用，详情请参见 YARN 手册。

Spark 既可以通过集群用户访问，又可以通过组件超级用户访问。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
    - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为"集群超级用户"。
    - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 Spark 组件的 spark 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

Spark 安装完成后，连接集群内的节点（该节点要求安装了 Livy for Spark2 Server、Spark Thrift Server、Spark Job History Server 或 Spark Client）即可操作 Spark。

2.3.1 Spark Shell

Spark 带有交互式 shell，可以执行即时数据分析。下面示例中通过 Spark Shell 执行 WordCount 程序。

1. 数据准备

(1) 创建 data.txt 文件，文件单词以逗号（英文）间隔。

```
Hello,World,Word,Count
```

(2) 上传文件至 HDFS 上。

```
hdfs dfs -put data.txt /tmp
```

2. 在 HDFS 用户下执行 spark-shell 命令，启动 Spark Shell

```
spark-shell
```

3. 在 Spark Shell 上提交任务

在 Spark Shell 中统计文件中单词个数：

```
val file = sc.textFile("/tmp/data.txt")
val counts = file.flatMap(line => line.split(",")).map(word => (word, 1)).reduceByKey(_ +_)
```

counts.saveAsTextFile("/tmp/wordcount")

counts.collect()

4. 查看输出结果

(1) 切换 hdfs 用户：su - hdfs

(2) 查看 WordCount 任务输出：hdfs dfs -ls /tmp/wordcount

可以看到以下信息：

/tmp/wordcount/_SUCCESS

/tmp/wordcount/part-00000

/tmp/wordcount/part-00001

(3) 使用 HDFS cat 查看文件内容： hdfs dfs -cat /tmp/wordcount/part*

结果如图 2-4 所示。

图2-4 单词统计结果

```
(Hello,1)
(Word,1)
(World,1)
(Count,1)
```

2.3.2 Spark Submit

Spark Submit 用于 Spark application 的提交和运行。

- Spark Submit 提交命令如下：

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

其中：

  - --class 指定提交 Spark 任务的类名
  - --master 指定 Spark 连接的 Master，如 yarn-client、yarn-cluster、local 等
  - application-jar 指定 Spark 应用的 jar 包的路径
  - application-arguments 指提交 Spark 任务的所需要的参数（可以为空）

- 用户可以通过 spark-submit -h 查看完整的参数说明。

示例：Yarn Client 模式提交 SparkPi，提交命令为：

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn --deploy-mode client
--executor-memory 2G --num-executors 5
/usr/hdp/3.0.1.0-187/spark2/examples/jars/spark-examples_2.11-2.4.0-cdh6.2.0.jar 100
```

2.3.3 Spark SQL

1. 启动SparkSQL

使用SparkSQL可以通过两种方式：SparkSQL CLI、Spark ThriftServer。

- SparkSQL CLI

  SparkSQL CLI是Spark提供SQL查询工具。

  在命令行中执行spark-sql命令启动SparkSQL CLI，执行SQL语句：

  ```
  [root@node1 opt]# spark-sql
  spark-sql>
  ```

- Spark ThriftServer

  Spark ThriftServer是Spark提供的JDBC/ODBC接口，可以通过JDBC/ODBC连接ThriftServer来访问Spark SQL的数据。

  用户可通过Spark提供的beeline工具连接Spark ThriftServer，可以使用以下任意方式连接：

  【方式一】

  a. 启动beeline，进入beeline交互界面，执行以下命令：

  ```
  /usr/hdp/3.0.1.0-187/spark2/bin/beeline
  ```

  b. 连接Spark ThriftServer，执行以下命令：

  ```
  !connect jdbc:hive2://<host_ip>:10016
  ```

  使用示例：

  ```
  [root@node1 opt]# /usr/hdp/3.0.1.0-187/spark2/bin/beeline
  beeline> !connect jdbc:hive2://node2:10016
  0: jdbc:hive2://node2:10016>
  ```

  其中：

  - node2为ThriftServer所在节点的IP域名。
  - 10016为Thriftserver的端口号。

  【方式二】

  a. 登录集群中安装Spark Client的某台机器，运行如下命令：

  ```
  beeline -u "jdbc:hive2://<host_ip>:10016/" -n spark -p ""
  ```

  使用示例：

  ```
  beeline -u "jdbc:hive2://node1:10016/" -n spark -p ""
  ```

  其中：

  - -u指定连接Spark ThriftServer的地址，其中Spark ThriftServer的端口号默认为10016。
  - -n用于指定连接所需的用户名（可为空值）。
  - -p用于指定连接所需的密码（可为空值）。

说明

- 在Kerberos环境中，以上两种方式通过beeline连接Spark ThriftServer时均需要带上ThriftServer的principal信息: jdbc:hive2://<*host_ip*>:10016/;principal=<*principal_name*>，

其中 host_ip 是 ThriftServer 所在节点 IP，principal_name 为启动 ThriftServer 的 principal（principal 获取方式请参见 2.3.5 2. (1)），需根据实际情况修改。

- 命令中提到的 IP 可以为真实 IP，也可以为该 IP 对应的域名，这两种表示等效。

2. 执行 SQL 语句

SparkSQL CLI 和 Spark ThriftServer 在执行 SQL 操作方面没有差别。下面示例以 SparkSQL CLI 方式执行 SQL 语句：

- 创建以 person 为表名，以 id，name 为列的表：

```
spark-sql> create table person (id int, name string);
```

- 插入数据：

```
spark-sql> insert into person values(1, 'xiaoming');
```

- 查询数据：

```
spark-sql> select * from person;
```

2.3.4 Spark Thriftserver（非 Kerberos 环境）

说明

非 Kerberos 环境下，不需要用户做身份认证即可直接对 Spark 执行管理操作。

在非 Kerberos 环境下通过控制台执行创建表、查询表等相关操作。操作示例如下：

(1) 连接 Spark ThriftServer

在集群中包含 Spark Client 的某台机器中运行如下命令：

```
beeline -u "jdbc:hive2://node1:10016/" -n spark -p ""
```

说明

在 beeline 命令中：

- -u：指定连接 Spark ThriftServer 的地址，其中 Spark ThriftServer 的端口号默认为 10016
- -n：指定连接所需用户名
- -p：指定连接所需密码（可为空值）

(2) 创建表

```
create table a(i int,s string);
```

(3) 插入数据

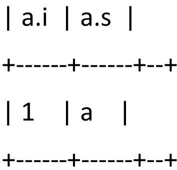
```
insert into table a values(1, 'a');
```

(4) 查看数据

```
select * from a;
```

结果：

```
+------+------+--+
```



```
| a.i  | a.s  |
+------+------+--+
| 1    | a    |
+------+------+--+
```

2.3.5 Spark Thriftserver（Kerberos 环境）

说明

Kerberos 环境下，若想访问 Spark 并对 Spark 执行管理操作，则必须首先进行用户身份认证，认证方式请参见 2.3.5 1. Kerberos 环境下用户身份认证。

1. Kerberos 环境下用户身份认证

Kerberos 环境下进行用户身份认证的方式，根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

Spark 还可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**

  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的/etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：

  ```
  chown user1 /etc/security/keytabs/user1.keytab
  ```

  b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：

  ```
  klist -k user1.keytab
  ```

  【说明】如图 2-5 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-5 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

su user1

kinit -kt user1.keytab user1@TENANTC.COM

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-6 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

（二）组件超级用户身份认证

Spark 可以通过 Spark Thrift Server 超级用户访问，比如 spark 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 spark 用户示例）认证的步骤如下：

(1) 在集群内安装了 Spark Thrift Server 的节点的/etc/security/keytabs/目录下，查找 spark 的认证文件“spark.service.keytab”。

(2) 使用 ***klist*** 命令查看 spark.service.keytab 的 principal 名称，命令如下：

klist -k spark.service.keytab

【说明】如图 2-7 所示，红框内容即为 spark.service.keytab 的 principal 名称。

图2-7 认证文件的 principal 名称

```
[root@node3 keytabs]# klist -k spark.service.keytab
Keytab name: FILE:spark.service.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 spark/node3.hde.com@TESTSHARE.COM
   2 spark/node3.hde.com@TESTSHARE.COM
   2 spark/node3.hde.com@TESTSHARE.COM
   2 spark/node3.hde.com@TESTSHARE.COM
   2 spark/node3.hde.com@TESTSHARE.COM
```

(3) 切换至用户 spark，并执行身份验证的命令如下：

su spark

kinit -kt spark.service.keytab spark/node3.hde.com@TESTSHARE.COM

【说明】其中：spark.service.keytab 为 spark 的认证文件，spark/hpppnode3.hde.com@TESTSHARE.COM 为 spark.service.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

2. Spark 操作说明

当用户身份认证成功后，在 Kerberos 环境下通过控制台执行创建表、查询表等相关操作。操作示例如下：

(1) Spark ThriftServer Principal 配置值获取

登录集群 Spark ThriftServer 所在节点，运行以下命令获取 principal：

klist -kt /etc/security/keytabs/spark.service.keytab

结果：

```
Keytab name: FILE:/etc/security/keytabs/spark.service.keytab
KVNO Timestamp        Principal
---- ------------------- ------------------------------------------------------
  2 04/02/2020 18:05:12 spark/node1.hde.com@TENANTC.COM
  2 04/02/2020 18:05:12 spark/node1.hde.com@TENANTC.COM
  2 04/02/2020 18:05:12 spark/node1.hde.com@TENANTC.COM
  2 04/02/2020 18:05:12 spark/node1.hde.com@TENANTC.COM
  2 04/02/2020 18:05:12 spark/node1.hde.com@TENANTC.COM
```

(2) 连接 Spark ThriftServer

登录集群中安装 Spark Client 的某台机器，运行如下命令：

beeline -u " jdbc:hive2://node1:10016/;principal=spark/node1.hde.com@TENANTC.COM " -n spark -p ""

其中：

- -u 用于指定连接 HiveServer2 的地址和端口号。
  - Spark ThriftServer 的端口号默认为 10016，principal 名称为 spark/node1.hde.com@TENANTC.COM，node1.hde.com 是 Spark ThriftServer 的域名地址。
- -n 用于指定连接所需的用户名（可为空值）。

- -p 用于指定连接所需的密码（可为空值）。

(3) 创建表

create table a(i int,s string);

(4) 插入数据

insert into table a values(1, 'a');

(5) 查看数据

select * from a;

结果：

```
+------+------+--+
| a.i  | a.s  |
+------+------+--+
| 1    | a    |
+------+------+--+
```

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

2.4.2 Spark 任务监控

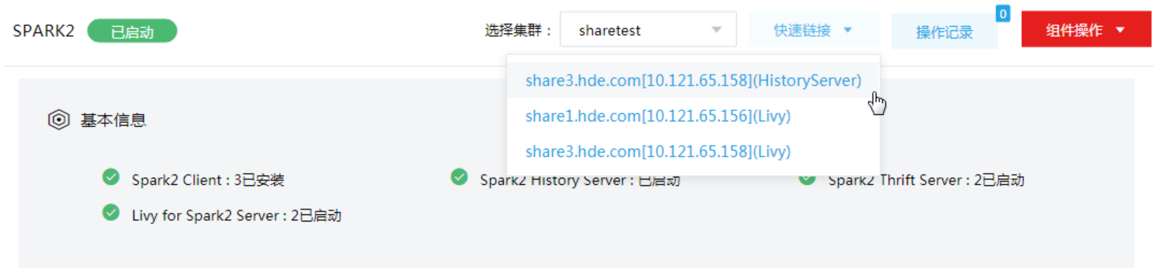
说明

运行 Spark 任务时，Spark Web UI 默认开启，此时通过 Spark History Server 服务可以查看任务状态。若计划开启 Spark Web UI 功能，需要修改 Spark 的配置，即修改 spark2-defaults 中 spark.ui.enabled 配置项的值为 true。

Spark History Server 用于监控正在运行的或者历史的 Spark 作业，显示 Spark 框架各个阶段的细节以及日志，帮助用户更细粒度地去开发、配置和调优作业。

(1) 如图 2-8 所示，在 Spark 组件详情页面的右上角[快速链接]的下拉框中，可以获取 Spark 的访问入口信息。

图2-8 Spark 快速链接



(2) 根据集群是否开启 Kerberos，访问 Spark 快速链接分为两种情况：

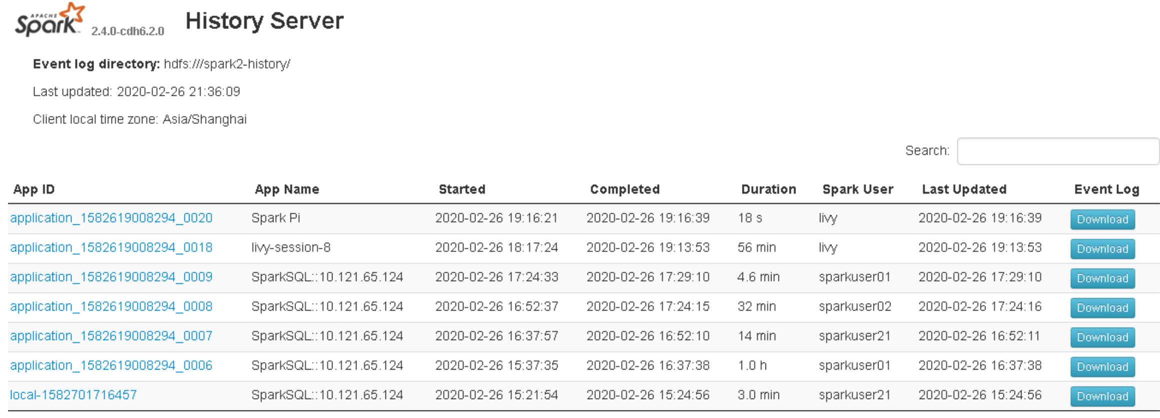
- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，可直接跳转访问对应的 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。在 UI 页面支持退出登录，如图 2-9 所示。单击页面右上角的<Log Out>按钮，即可清除当前用户的登录信息重新跳转至登录页面。

图2-9 退出登录

(3) 在 Spark History Server 页面，可查看以下信息：

- 默认展示已完成的历史任务列表。可以单击左下角“Show incomplete applications”跳转到正在运行的任务列表。

图2-10 Spark History Server 页面



Spark 2.4.0-cdh6.2.0 History Server

Event log directory: hdfs:///spark2-history/

Last updated: 2020-02-26 21:36:09

Client local time zone: Asia/Shanghai

Search:

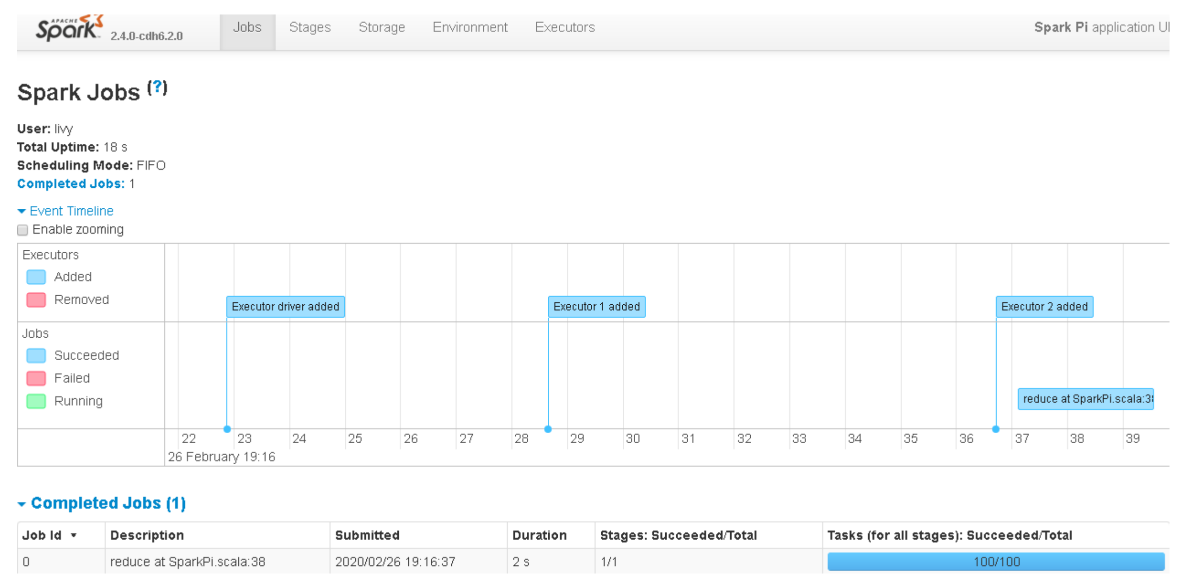
App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
application_1582619008294_0020	Spark Pi	2020-02-26 19:16:21	2020-02-26 19:16:39	18 s	livy	2020-02-26 19:16:39	Download
application_1582619008294_0018	livy-session-8	2020-02-26 18:17:24	2020-02-26 19:13:53	56 min	livy	2020-02-26 19:13:53	Download
application_1582619008294_0009	SparkSQL::10.121.65.124	2020-02-26 17:24:33	2020-02-26 17:29:10	4.6 min	sparkuser01	2020-02-26 17:29:10	Download
application_1582619008294_0008	SparkSQL::10.121.65.124	2020-02-26 16:52:37	2020-02-26 17:24:15	32 min	sparkuser02	2020-02-26 17:24:16	Download
application_1582619008294_0007	SparkSQL::10.121.65.124	2020-02-26 16:37:57	2020-02-26 16:52:10	14 min	sparkuser21	2020-02-26 16:52:11	Download
application_1582619008294_0006	SparkSQL::10.121.65.124	2020-02-26 15:37:35	2020-02-26 16:37:38	1.0 h	sparkuser01	2020-02-26 16:37:38	Download
local-1582701716457	SparkSQL::10.121.65.124	2020-02-26 15:21:54	2020-02-26 15:24:56	3.0 min	sparkuser21	2020-02-26 15:24:56	Download

Showing 1 to 7 of 7 entries

Show incomplete applications

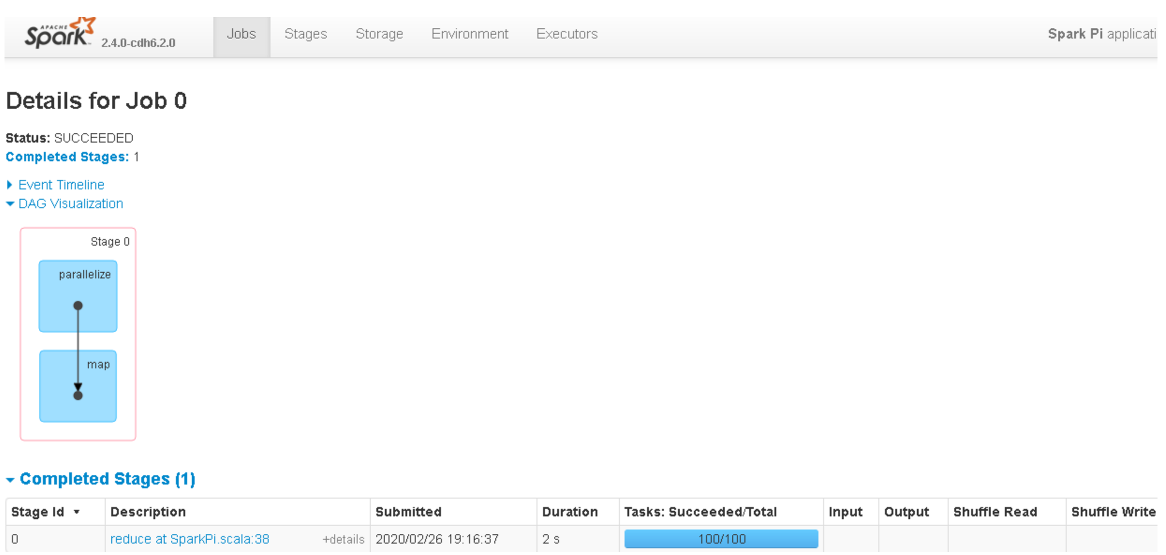
- 单击 App ID 列表中的任务，可以进入该任务的 Jobs 列表。

图2-11 Spark Jobs 列表



- 单击“Completed Jobs”列表中的任意 Job，进入该 Job 的详情页面，可查看 Stage 列表。

图2-12 Job 页面



- 单击“Completed Stages”列表中任意 Stage，可以进入该 Stage 的详细页面，包含每个 task 的执行时间，GC 时间等。

图2-13 Stage 详细信息

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 1 s
Locality Level Summary: Process local: 100

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 100 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	9 ms	11 ms	11 ms	13 ms	95 ms
GC Time	0 ms	0 ms	0 ms	0 ms	35 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Blacklisted
1 stdout stderr	noder125.hde.com:42353	2 s	50	0	0	50	false
2 stdout stderr	noder126.hde.com:36573	2 s	50	0	0	50	false

Tasks (100)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	2	noder126.hde.com stdout stderr	2020/02/26 19:16:37	95 ms		

3 使用指南

3.1 Spark SQL

Spark SQL 是用于结构化数据处理的 Spark 模块。它可以通过执行 SQL 查询完成复杂的数据分析工作。

- SparkSQL 支持大部分 Hive DDL 及 DML 语法，具体使用方法可参看 Hive 官网：
  https://cwiki.apache.org/confluence/display/Hive/LanguageManual
- 相较于 Hive，SparkSQL 新增了如下建表语法：
  CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type [COMMENT col_comment], ...)]
  [USING datasource]
  [OPTIONS property]
  [AS select_statement];
  该建表语法可以让 SparkSQL 创建数据源表。

3.1.1 Spark 创建 Hive 表

SparkSQL 支持对 Hive 表的读写操作，例如：

- 通过 SparkSQL 的 CREATE TABLE 命令，创建一张 Hive 表。
  CREATE TABLE table_hive(id int, name string);
- 通过 SHOW CREATE TABLE 命令，显示表详细信息，如图 3-1 所示。
  SHOW CREATE TABLE table_hive;

图3-1 详细信息

```
spark-sql> SHOW CREATE TABLE table_hive;
CREATE TABLE `table_hive`(`id` int, `name` string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = '1'
)
STORED AS
  INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
  OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
TBLPROPERTIES (
  'rawDataSize' = '-1',
  'numFiles' = '0',
  'transient_lastDdlTime' = '1504005284',
  'totalSize' = '0',
  'COLUMN_STATS_ACCURATE' = 'false',
  'numRows' = '-1'
)
```

3.1.2 Spark 创建数据源表

1. Parquet

Parquet 是一种列数据格式，广泛应用于数据处理系统。SparkSQL 支持将数据以 Parquet 格式存储。

- 创建 Parquet 表可以通过 SparkSQL 的建表语法，并指定 USING 的数据源为 Parquet。语法如下：

  CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name

  [(col_name data_type [COMMENT col_comment], ...)]

  USING parquet

  [OPTIONS property]

  [AS select_statement];

- 通过已有 Parquet 文件建表

  在创建 Parquet 数据源表时，可以通过已有的 Parquet 文件，通过解析 Parquet 获取建表字段来创建表，并导入已有 Parquet 文件中的数据。使用这种方法建表时，通过 OPTIONS 中的 path 参数指定 Parquet 文件路径。这样就不需要手动设定表中字段，SparkSQL 可以自动从文件中获取字段信息（建表语句中的 path 指定到文件或路径均可）。

  使用示例：

  a. 创建一个 users_parquet 表，如图 3-2 所示，在建表时不指定表字段，建表语句为：

  CREATE TABLE users_parquet USING parquet OPTIONS(path '/users/users.parquet');

注意

- path 指定的路径为 HDFS 路径。
- 在上面例子中，需将/usr/hdp/3.0.1.0-187/spark2/examples/src/main/resources/users.parquet 上传到 HDFS 的/users 目录下，上传前请确认目录存在。

图3-2 创建 users_parquet 表

```
CREATE TABLE `users_parquet` (`name` STRING, `favorite_color` STRING
, `favorite_numbers` ARRAY<INT>)
USING parquet
OPTIONS (
  `serialization.format` '1',
  path '/users/users.parquet'
)
```

  b. 执行 SHOW CREATE TABLE users_parquet 命令可以看到表字段已经自动获取。查询 users_parquet 表中数据：

  SELECT * FROM users_parquet;

图3-3 show create tableusers_parquet 表

```
CREATE TABLE `users_parquet` (`name` STRING, `favorite_color` STRING
, `favorite_numbers` ARRAY<INT>)
USING parquet
OPTIONS (
  `serialization.format` '1',
  path '/users/users.parquet'
)
```

图3-4 查询结果

```
Alyssa  NULL    [3,9,15,20]
Ben     red     []
```

- 新建 Parquet 数据源表

  创建一张 Parquet 数据源表时，需要在建表时指定字段信息。

  使用示例：

  a. 指定字段创建一个名为 m_parquet 表：

     CREATE TABLE m_parquet(id int, name string) USING parquet;

  b. 向 m_parquet 表中插入数据：

     INSERT INTO m_parquet values(1,"Spark"),(2,"Hive"),(3,"Parquet");

  c. 查询 m_parquet 表中数据，查询指令：

     SELECT * FROM m_parquet;

图3-5 查询结果

```
3       Parquet
1       Spark
2       Hive
Time taken: 0.765 seconds, Fetched 3 row(s)
```

2. JSON

SparkSQL 能自动解析 JSON 数据集的 schema。需要注意的是，这里的 JSON 文件不是常规的 JSON 格式。JSON 文件每一行必须包含一个独立的、自满足有效的 JSON 对象。如果用多行描述一个 JSON 对象，会导致读取出错。

使用示例：

(1) 使用 JSON 数据集创建表，建表语句为：

CREATE TABLE table_json USING json OPTIONS(path 'example/resources/people.json');

注意

- path 指定的路径为 HDFS 路径。
- 在上面例子中，需将/usr/hdp/3.0.1.0-187/spark2/examples/src/main/resources/people.json 上传到 HDFS 的/users 目录下，上传前请确认目录存在。

(2) 通过 SHOW CREATE TABLE 命令可以看出 SparkSQL 自动解析 JSON 数据集中的字段信息，如图 3-6 所示。

SHOW CREATE TABLE table_json;

图3-6 查询字段信息

```
spark-sql> SHOW CREATE TABLE table_json;
CREATE TABLE `table_json` (`age` BIGINT, `name` STRING)
USING json
OPTIONS (
  `serialization.format` '1',
  path '/example/resources/people.json'
)
```

(3) 查看 table_json 表中数据：

SELECT * FROM table_json;

图3-7 查询结果

```
NULL    Michael
30      Andy
19      Justin
```

3. 数据库作为数据源

SparkSQL 也可以将数据库作为它的数据源，来获取数据库中的数据。

(1) 将数据库作为数据源时，需要在 spark classpath 添加 JDBC Driver 依赖包。例如，SparkSQL 首次连接 PostgreSQL 数据库时，首先需要在$SPARK-HOME$/jars/文件夹下添加 PostgreSQL 的 JDBC Driver 依赖包，然后重新启动 SparkSQL。

(2) 在创建数据表时，用户需要指定 JDBC 数据源连接属性。例如，需要指定数据库的 user 和 password。

使用示例：

(1) 在 Hive 数据库中创建“table_info”表

create table table_info(id int, name string, job string);

(2) 在 SparkSQL 中创建映射表“jdbcTableInfo”：

CREATE TABLE jdbcTableInfo

USING org.apache.spark.sql.jdbc

OPTIONS(

URL "jdbc:postgresql://100.4.129.101:5432/hive",

dbtable "table_info",

user "hive",

password "passwd");

(3) 用户可以在 SparkSQL 中通过 jdbcTableInfo 表，对 Hive 的“table_info”表执行 SQL 操作。例如执行插入操作、查询操作等。

- 向 jdbcTableInfo 表插入数据：

  INSERT INTO jdbcTableInfo VALUES(10, "libai","teacher");

- 查询 jdbcTableInfo 表中数据：

  SELECT * FROM jdbcTableInfo;

图3-8 查询结果

```
1       wanghua teacher
2       zhangjie        singer
3       kenan   student
4       wangwu  teacher
10      libai   teacher
```

3.2 Spark Thriftserver HA

Spark 的 ThriftServer 提供了 HA 的功能。此功能要求在两个或两个以上节点安装 ThriftServer。用户连接 ThriftServer 时，Spark 会随机选择一个节点上的 ThriftServer 与用户建立连接。当前连接节点上的 ThriftServer 服务挂掉后，Spark 会自动将连接切换到其它节点的 ThriftServer 上。

注意

在获取查询结果集时发生 ThriftServer 切换，将不会重新获取结果集，可能造成查询结果不准确。

- 开启 ThriftServer HA 后，Spark 会将 ThriftServer 信息注册到 Zookeeper 上，可以在 Zookeeper 上看到 ThriftServer 注册的信息。

  使用示例：

  在 Zookeeper 上查看 Spark ThriftServer 注册的信息

  ```
  [zk: noder1:2181(CONNECTED) 1] ls /spark2_server
  [serverUri=noder1.hde.com:10016;version=2.1.1-cdh6.2.0;sequence=0000000000,
  serverUri=noder2.hde.com:10016;version=2.1.1-cdh6.2.0;sequence=0000000001]
  ```

- 开启 ThriftServer HA 后，JDBC 的 url 需要使用以下格式：

  jdbc:hive2://<zk_host_list>/;serviceDiscoveryMode=zookeeper;zooKeeperNamespace=<namespace_directory>

  其中：

  - zk_host_list 表示 Zookeeper 的节点 IP 和端口号，可以是一个或多个。如 zk_host1:zk_port,zk_host2:zk_port,zk_host3:zk_port；
  - namespace_directory 表示 Spark 组件在 Zookeeper 上所在的命名空间。该参数值由“hive.server2.zookeeper.namespace”配置，默认值为 spark2_server。

使用示例：

使用 beeline 连接 ThriftServer：

[root@node1 opt]# .cd /usr/hdp/3.0.1.0-187/spark2/bin

[root@node1 opt]# ./beeline

```
beeline> !connect
jdbc:hive2://node1:2181,node2:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespa
ce=spark2_server
0: jdbc:hive2://node1:2181,node2:2181/> select * from mltable;
+-----+--------+-------+-------+-------+--+
| id  | label  | col1  | col2  | col3  |
+-----+--------+-------+-------+-------+--+
| 1   | 0.0    | 2.3   | 4.5   | 5.6   |
| 2   | 1.0    | 6.5   | 3.8   | 4.9   |
+-----+--------+-------+-------+-------+--+
```

3.3 Spark Livy

Livy 是基于 Spark 的开源 REST 服务。用户能够通过 REST 的方式将代码片段或序列化的二进制代码提交到 Spark 集群中去执行。根据处理交互方式的不同，Livy 将会话分成了两种类型：

- 交互式会话（interactive session），这与 Spark 中的交互式处理相同，交互式会话在其启动后可以接收用户所提交的代码片段，在远端的 Spark 集群上编译并执行。
- 批处理会话（batch session），用户可以通过 Livy 以批处理的方式启动 Spark 应用，这样的一个方式在 Livy 中称之为批处理会话，这与 Spark 中的批处理是相同的。

1. 交互式会话（Interactive Session）

使用交互式会话与使用 Spark 所自带的 Spark-Shell、pyspark 或 SparkR 相类似，它们都是由用户提交代码片段给 REPL，由 REPL 来编译成 Spark 作业并执行。

- 创建交互式会话

  使用交互式会话的前提是需要先创建会话。当我们提交请求创建交互式会话时，需要指定会话的类型（“kind”），比如“spark”，Livy 会根据我们所指定的类型来启动相应的 REPL，当前 Livy 可支持 Spark、pyspark、Sparkr 三种不同的交互式会话类型。

  - 执行下面代码，创建一个 Spark 会话：

    curl -X POST -d '{"kind": "spark"}' -H 'Content-Type: application/json' node1:8999/sessions

  - 当创建完会话后，Livy 会返回一个 JSON 格式的数据结构表示当前会话的所有信息：

    {"id":0,"appId":null,"owner":null,"proxyUser":null,"state":"starting","kind":"spark","appInfo":{"driverLogUrl":null,"sparkUiUrl":null},"log":["stdout: ","\nstderr: ","\nYARN Diagnostics: "]}

    其中需要关注的是 id 字段，id 字段代表了此会话，所有基于该会话的操作都需要指明其 id。

- 提交代码

  创建完交互式会话后，可以提交代码到该会话上去执行：

  curl node1:8999/sessions/0/statements -X POST -H 'Content-Type: application/json' -d '{"code": "var a = 1; var b = a + 1"}'

  与创建会话相同的是，提交代码同样会返回一个 id 用来标识该次请求，可以用 id 来查询该段代码执行的结果。返回信息如下：

```
{"id":0,"code":"var a = 1; var b = a + 1","state":"waiting","output":null,"progress":0.0}
```

- 查询执行结果

  发送下面命令，查询执行结果：

  curl node1:8999/sessions/0/statements/0

  返回信息如下：

```
{"id":0,"code":"var a = 1; var b = a + 1","state":"available","output":{"status":"ok","execution_count":0,"data":{"text/plain":"a: Int = 1\nb: Int = 2"}},"progress":1.0}
```

- 删除会话

  根据会话 id 可以删除正在运行的会话

  curl -X DELETE node1:8999/sessions/0

  返回信息如下：

```
{"msg":"deleted"}
```

2. 批处理会话（Batch Session）

批处理会话是用户将业务逻辑编译打包成 jar 包，并通过 spark-submit 启动 Spark 集群来执行业务逻辑。Livy 也为用户带来相同的功能，用户可以通过 REST 的方式来创建批处理应用。

下面示例中，使用 Livy 提交 SparkPi 任务。

- 上传 SparkPi 的 jar 包到 hdfs：

  hdfs dfs -put /usr/hdp/3.0.1.0-187/spark2/examples/jars/spark-examples_2.11-2.4.0-cdh6.2.0.jar /tmp

- 执行 livy 命令提交会话：

  curl -X POST -H 'Content-Type: application/json' node1:8999/batches --data '{ "file":"/tmp/spark-examples_2.11-2.4.0-cdh6.2.0.jar","className":"org.apache.spark.examples.SparkPi","name":"livy pi","args":["100"] }'

  提交成功后会返回会话信息，返回内容如下：

```
{"id":2,"state":"starting","appId":null,"appInfo":{"driverLogUrl":null,"sparkUiUrl":null},"log":["stdout: ","\nstderr: ","\nYARN Diagnostics: "]}
```

- 查看任务执行结果

  curl node1:8999/batches/2

  返回信息如下：

```
{"id":2,"state":"success","appId":"application_1582619008294_0020","appInfo":{"driverLogUrl":null,"sparkUiUrl":"http://node2.hde.com:8088/proxy/application_1582619008294_0020/"},"log":["stdout: ","Warning: Master yarn-cluster is deprecated since 2.0. Please use master \"yarn\" with specified deploy mode instead.","Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties","\nstderr: ","\nYARN Diagnostics: "]}
```

3.4 Client下载/安装/使用/卸载

用户可下载 Spark Client，在客户端节点上安装 Spark 的 Client 后，即可直接连接集群中的 Spark，进行组件维护、任务管理等操作。

3.4.1 下载 Client 安装包

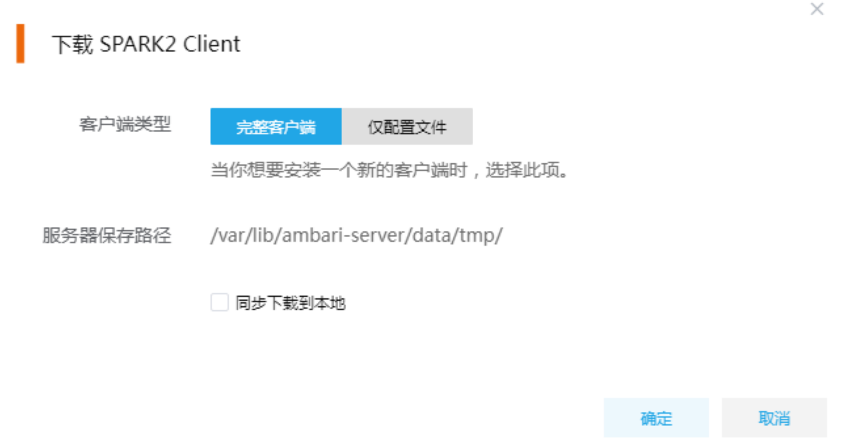
注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在集群管理页面的集群列表中，点击集群名称进入集群详情页面，在集群已安装的组件列表中单击 Spark 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-9 所示。

图3-9 下载 Client 窗口



(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：
   - 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
   - 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中的组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。
   - 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
   - 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.4.2 安装 Client

注意

- 安装 Client 的节点必须能与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置网络连接，仅非 root 用户需要执行此操作，root 用户可跳过此步骤。

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

./install.sh <安装路径> -o

说明

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为 /usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>

3.4.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

  source bigdata_env
- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同：
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行校验即可直接访问组件并使用组件的 Client 进行相关操作。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行认证之后，才可访问组件并使用组件的 Client 进行相关操作。关于 Kerberos 环境下，进行用户认证的操作方式请参见 2.3.5 1. Kerberos 环境下用户身份认证。

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

3.4.4 使用客户端 Client

1. 非 Kerberos 操作步骤

- 使用 Spark Shell

  使用示例：

  a.进入 Spark 客户端安装目录。执行 spark-shell 命令，启动 spark-shell，进入 shell 命令行。

  b.编写 WordCount 代码：

  ```
  scala> var result = sc.textFile("/tmp/data.txt").flatMap(_.split(",")).map((_,1)).reduceByKey(_+_)
  scala> result.foreach(println)
  ```
- 使用 spark-submit 提交任务

  使用示例：

  进入 Spark 客户端安装目录，执行以下代码提交 SparkPi 任务。

  ```
  spark-submit --master yarn  --class org.apache.spark.examples.SparkPi
  opt/SparkClient/Spark2/spark2/examples/jars/spark-examples_2.11-2.1.2.jar 10
  ```
- 使用 spark-sql

  使用示例：

  进入 Spark 客户端安装目录，执行./bin/spark-sql 启动 SparkSQL，进入 spark-sql 命令行。
- 使用 beeline

  使用示例：

进入 Spark 客户端安装目录，执行./beeline 启动 beeline，进入 beeline 命令行。执行以下命令，并根据提示输入 username 和 password。

[root@node1 opt]# ./beeline

beeline> !connect jdbc:hive2://node2:10016

其中，node2 为 Spark ThriftServer 安装节点的 IP 域名，10016 为 Spark ThriftServer 的端口号。

2. Kerberos 环境操作步骤

如果集群开启 Kerberos，在使用 Spark 客户端之前，需要对用户主体执行用户身份认证。

Kerberos 认证参考章节执行步骤请参见 2.3.5 1. Kerberos 环境下用户身份认证。

说明

开启 Kerberos 后，除了使用 Beeline 连接外，其他操作与非 Kerberos 环境相同。

使用示例：

(1) 使用 beeline

进入 Spark 客户端安装目录，执行./beeline 启动 beeline。

(2) 连接 Spark ThriftServer

进入 beeline 命令行，执行!connect 命令连接 Spark ThriftServer，如图 3-10 所示：

beeline>!connect jdbc:hive2://test01:10016/;principal=spark/test01.hde.com@TEST01.COM

图3-10 Kerberos 环境下使用 beeline

```
beeline> !connect jdbc:hive2://test01:10016/;principal=spark/test01.hde.com@TEST01.COM
Connecting to jdbc:hive2://test01:10016/;principal=spark/test01.hde.com@TEST01.COM
2021-11-10 14:18:43,873 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop library for your platform...
cable
Connected to: Spark SQL (version 2.4.0-cdh6.2.0)
Driver: Hive JDBC (version 2.1.1-cdh6.2.0)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://test01:10016/>
```

3.4.5 卸载 Client 客户端

集群重装之后，之前安装的 Client 客户端将不可用。此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 安装目录下执行以下命令：

./uninstall.sh

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

3.5 添加/删除进程

3.5.1 添加进程

Spark 支持添加 Spark2 Thrift Server、Spark2 Client 进程。

- 在并发查询任务较多的情况下，可以考虑增加 Spark2 Thrift Server 进程，分摊查询负载，提高并发数。
- 若主机新扩容机器时未勾选 client，后期可以通过添加 DLH Client 进程在新机器上增加。

1. 操作示例

说明

- 本章节仅以添加 Spark2 Thrift Server 进程为例进行说明，其它进程操作类似不再进行说明。
- 若集群中所有节点均已安装 Spark2 Thrift Server，添加 Spark2 Thrift Server 进程前则需要先在集群中添加主机，然后再执行添加 Spark2 Thrift Server 进程的操作。如果集群中已有添加进程所需要的主机，则可直接执行添加 Spark2 Thrift Server 进程的操作。

添加 Spark2 Thrift Server 进程的操作步骤如下：

(1) 在 Spark 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-11 所示。

a. 选择进程及主机。

在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

b. 部署进程。

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程。

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-11 添加进程

(3) 查看进程变化

Spark2 Thrift Server 添加完成之后，在组件详情页面[部署拓扑]页签可以查看 Spark2 Thrift Server 安装数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.5.2 删除进程

Spark 支持删除 Spark2 Thrift Server、Livy for Spark2 Server 进程。

- 在并发任务较少的情况下，可以考虑减少 Spark2 Thrift Server 和 Livy for Spark2 Server 进程，节省集群资源。
- 删除进程后，Spark2 Thrift Server、Livy for Spark2 Server 分别对应进程的个数均不能少于 1 个。

1. 操作示例

说明

- 删除进程操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行删除 Spark2 Thrift Server 进程操作”为例进行说明，其它进程操作类似不再进行说明。
- 执行删除 Spark2 Thrift Server 进程操作前，请确认 Spark2 Thrift Server 是否有运行的任务，如果有运行的任务时，删除进程会影响任务执行。

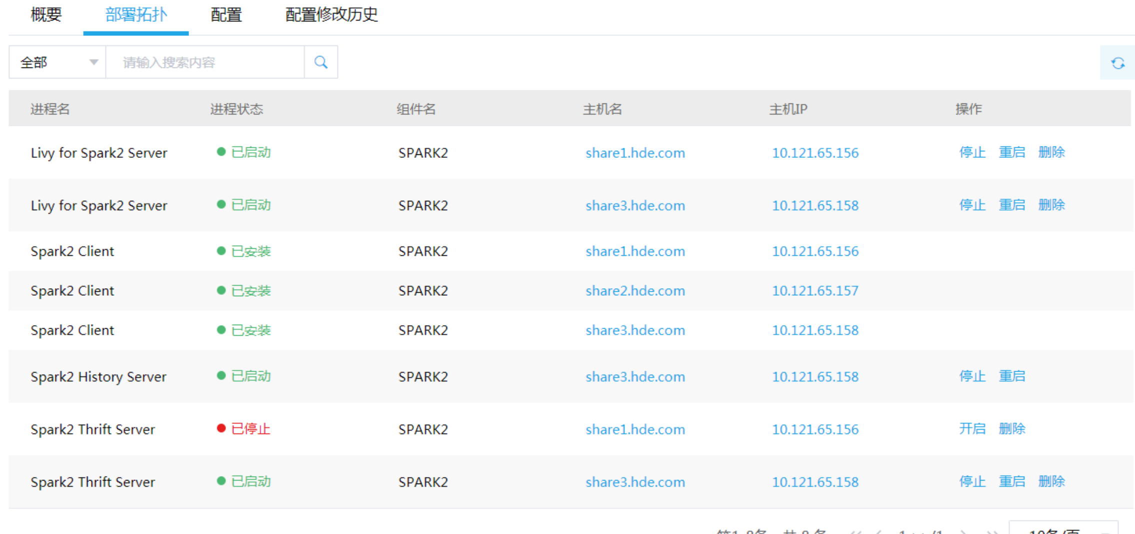
删除 Spark2 Thrift Server 进程的操作步骤如下：

(1) 在 Spark 组件详情页面[部署拓扑]页签下，选择要删除 Spark2 Thrift Server 进程的主机，然后单击该进程右侧操作中的<停止>按钮，停止 Spark2 Thrift Server。

(2) 删除 Spark2 Thrift Server

待 Spark2 Thrift Server 停止成功后，如图 3-12 所示，在该进程右侧操作中单击<删除>按钮，即可完成删除 Spark2 Thrift Server。

图3-12 删除 Spark2 Thrift Server



(3) 查看进程变化

Spark2 Thrift Server 删除完成之后，在组件详情页面[部署拓扑]页签中可以查看 Spark2 Thrift Server 进程的数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.6 权限访问控制

注意

集群新建用户的组件权限会因为集群是否开启权限管理功能而有所不同：

- 未开启权限管理时，用户可进行库表的创建、修改、插入、删除等操作。
- 开启权限管理后，组件权限需通过[集群权限/角色管理]中的角色分配给用户，用户通过绑定角色进行赋权后，才能对组件执行操作。

权限管理是安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

Spark 支持对数据库、表配置权限，可对数据库、数据表、列配置权限，权限包括：select、update、create、drop、alter，其中 all 表示配置所有权限。Spark 操作所需权限对应关系如表 3-1 所示。

表3-1 Spark 权限说明

权限类型	对应的组件常用操作
select	查询库表等相关操作，如：select、export、show、describe
update	更新库表等相关操作，如：insert、insert overwrite、load
create	创建库表等相关操作，如：create table、create database
drop	删除库表等相关操作，如：drop table
alter	修改库表等相关操作，如：alter table
all	支持以上所有操作

说明

- spark-sql -master yarn 方式连接时需要授予角色 YARN default 队列的 submit-app 权限。
- Spark 使用 Hive 数据库表的权限管理模块，库的创建者拥有该库下表的创建操作权限，但是查询、插入等需要授权才能进行操作。同样，表的创建者，也仅拥有创建权限。
- 创建数据库时，需赋予 create 权限，且要求数据表和列都为*；创建数据表时，需赋予 create 权限，且要求列为*。
- 默认 use 不受权限控制。

3.6.1 权限操作示例

1. 数据库权限控制

表3-2 授权配置

操作	权限要求（数据库示例为 sparktest）
create database	数据库：*或sparktest，数据表：*，列：*，权限：create
drop database	数据库：*或sparktest，数据表：*，列：*，权限：drop

以授予 create 权限为例，对数据库 sparktest 授予 create 权限（其它权限的授予方式与 create 方式操作类似），操作步骤如下：

(1) 新建用户 spark01，授权前执行创建库操作。如图 3-13 所示，提示用户 spark01 没有 sparktest 数据库的 create 权限。

图3-13 执行结果

```
spark-sql> create database sparktest;
org.apache.ranger.authorization.spark.authorizer.SparkAccessControlException: Permission denied: user [sparkuse
r01] does not have [CREATE] privilege on [sparktest]
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1.ap
ply(RangerSparkAuthorizer.scala:123)
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1.ap
ply(RangerSparkAuthorizer.scala:98)
```

(2) 角色授权

在角色管理页面，新建角色 spark01，本示例授予 spark01 角色 sparktest 数据库的 create 权限。本示例用 spark-sql 方式测试(默认是 YARN 模式)，需要同时授予角色 YARN default 队列的 submit-app 权限。

图3-14 为 spark01 角色授予权限

(3) 用户绑定角色

在用户管理页面，在 spark01 用户的操作列，单击<修改用户授权>按钮，选择角色 spark01。

图3-15 用户绑定角色

(4) 重新执行创建 sparktest 数据库操作。如图 3-16 所示，红框内信息表明数据库 sparktest 成功创建。

图3-16 执行结果

```
spark-sql> create database sparktest;
2020-04-07 18:28:56,124 INFO  [main] thriftserver.SparkSQLCLIDriver: Time taken: 1.421 seconds
```

2. 表权限控制

表3-3 授权配置

操作	权限要求（数据库示例为 sparktest，表示例为 sparktab1）
create table	数据库：*或sparktest，数据表：*或sparktab1，列：*，权限：create
alter table	数据库：*或sparktest，数据表：*或sparktab1，列：*，权限：alter
insert into table	• Hive：数据库：*或 sparktest，数据表：*或 sparktab1，列：*，权限：update • YARN：队列：*或 default，权限：submit-app
select * from table	• Hive：数据库：*或 sparktest，数据表：*或 sparktab1，列：*，权限：select • YARN：队列：*或 default，权限：submit-app
drop table	数据库：*或sparktest，数据表：*或sparktab1，列：*，权限：drop

以授予“update”权限为例，对数据库 sparktest 数据表 sparktab1 授予 update 权限，操作步骤如下：

(1) 新建用户 sparkuser21，授权前，执行插入表 sparktest.sparktab1 操作。如图 3-17 所示，提示 sparkuser21 用户没有 sparktest.sparktab1 数据表的 update 权限。

图3-17 插入表

```
spark-sql> insert into sparktest.sparktab1 values(1, 'v1');
org.apache.ranger.authorization.spark.authorizer.SparkAccessControlException: Permission denied: user [sparkuse
r21] does not have [UPDATE] privilege on [sparktest/sparktab1/id,name]
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1$$a
nonfun$apply$2.apply(RangerSparkAuthorizer.scala:115)
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1$$a
nonfun$apply$2.apply(RangerSparkAuthorizer.scala:113)
```

(2) 角色授权

在[集群权限/角色管理]页面，新建角色 spark21，本示例授予 spark21 角色 sparktest.sparktab1 数据表的 update 权限。本示例用 spark-sql 方式测试(默认是 YARN 模式)，需要同时授予 YARN default 队列的 submit-app 权限

图3-18 为 spark21 角色授予权限

(3) 用户绑定角色

在[集群权限/用户管理]页面，在 sparkuser21 用户的操作列，单击<修改用户授权>按钮，选择角色 spark21。

图3-19 用户绑定角色

(4) 重新执行插入表 sparktest.sparktab1 操作。如图 3-20 所示，提示 sparktest.sparktab1 数据表成功插入数据。

图3-20 重新执行插入操作

```
spark-sql> insert into sparktest.sparktab1 values(1, 'v1');
2020-04-07 18:48:28,989 INFO  [main] thriftserver.SparkSQLCLIDriver: Time taken: 7.837 seconds
```

以授予“SELECT”权限为例，对数据库 sparktest 数据表 sparktab1 授予 select 权限，操作步骤如下：

(1) 执行查询表 sparktest.sparktab1 操作。如图 3-21 所示，截图中表示 sparkuser21 用户没有 sparktest.sparktab1 数据表的 select 权限。

图3-21 执行查询操作

```
spark-sql> select * from sparktest.sparktab1;
org.apache.ranger.authorization.spark.authorizer.SparkAccessControlException: Permission denied: user [sparkuse
r21] does not have [SELECT] privilege on [sparktest/sparktab1/id,name]
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1$$a
nonfun$apply$2.apply(RangerSparkAuthorizer.scala:115)
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1$$a
nonfun$apply$2.apply(RangerSparkAuthorizer.scala:113)
```

(2) 角色授权

在[集群权限/角色管理]页面，单击角色列表中 spark21 角色操作列的<编辑>按钮，可以修改角色组件权限设置和描述信息。本示例授予 spark21 角色 sparktest.sparktab1 数据表的 select 权限。

图3-22 角色授权

(3) 重新执行查询表 sparktest.sparktab1 操作。如图 3-23 所示，信息表明可以查询 sparktest.sparktab1 数据表数据

图3-23 重新执行查询操作

```
spark-sql> select * from sparktest.sparktab1;
1          v1
2020-04-07 18:50:09,726 INFO  [main] thriftserver.SparkSQLCLIDriver: Time taken: 4.559 seconds, Fetched 1 row(s
)
```

3. 列级别权限控制

表3-4 授权配置

操作	权限要求（数据库示例为 sparktest，数据表示例为 sparktab1，列示例为 name）
select name from table	数据库：*或sparktest，数据表：*或sparktab1，列：name，权限：select

(1) 新建用户 spark02，授权前执行查询操作，如图 3-24 所示，提示当前用户没有表 sparktab1 的查询权限。

图3-24 执行结果

```
spark-sql> select name from sparktest.sparktab1;
org.apache.ranger.authorization.spark.authorizer.SparkAccessControlException: Permission denied: user [sparkuser02] does not have [SELECT] privilege on [sparktest/sparktab1/name]
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1.apply(RangerSparkAuthorizer.scala:123)
        at org.apache.ranger.authorization.spark.authorizer.RangerSparkAuthorizer$$anonfun$checkPrivileges$1.apply(RangerSparkAuthorizer.scala:98)
```

(2) 角色授权

在[集群权限/角色管理]页面，新建角色 spark02，本示例授予 spark02 角色，sparktest 库下 sparktab1 表的 name 列查询权限。本示例用 spark-sql 方式测试(默认是 YARN 模式)，需要同时授予 YARN default 队列的 submit-app 权限。

图3-25 为 spark02 角色授权

(3) 用户绑定角色

在[集群权限/用户管理]页面，在 sparkuser02 用户的操作列，单击<修改用户授权>按钮，选择角色 spark02。

图3-26 用户绑定角色

(4) 重新执行查询操作，如图 3-27 所示，select 操作执行成功。

图3-27 执行结果

```
spark-sql> select name from sparktest.sparktab1;
v1
2020-04-07 18:57:10,481 INFO  [main] thriftserver.SparkSQLCLIDriver: Time taken: 6.746 seconds, Fetched 1 row(s
)
```

3.7 备份恢复

备份恢复提供大数据平台跨集群之间的数据同步功能，支持对 HDFS、Hive、HBase、Kafka 组件中的指定数据进行同步备份，以保证数据内容不丢失。

Spark 数据同步主要同步 Spark 的数据库和表，通过备份恢复中的 Hive 同步任务来完成，具体内容请参见产品在线联机帮助。

说明

- 新建同步任务时，要求源集群与目标集群的集群类型、集群模式均相同。
- 新建同步任务时，要求源集群与目标集群的安全管理策略相同，即同时开启 Kerberos 认证或同时都没有开启 Kerberos 认证。
- 新建同步任务时，要求源集群与目标集群的集群名称、集群中节点主机名均不相同，否则配置跨集群互信时可能出错。
- 源集群为同步任务的数据输出集群（一般指新建同步任务的本集群）；目标集群为同步任务的数据输入集群。
- Hive 同步任务不支持事务表的同步，不支持多个主集群备份到同一个备集群。
- Hive 同步任务为周期性调度任务。Hive 同步任务周期性执行时，默认首次执行是全量数据备份，后续执行是增量数据备份，备份周期间隔时间内的数据不会实时同步到目的集群。

3.8 Hudi表操作

说明

- 读写 Hudi 时依赖 hudi-spark-bundle 和 spark-avro 包，spark 安装目录默认已包含 hudi-spark-bundle 和 spark-avro 包。
- Spark 默认集成 Hive 的环境为 Hive 服务的地址，在使用 Spark 写 Hudi 表同步 DLH 时需要调整为 DLH 服务的地址，否则会因地址不一致导致同步失败。

Hudi 表可通过 Spark Shell 或 Spark SQL 进行操作，需注意：

(1) Spark Shell 或 Spark SQL 启动时，需指定配置：

spark.serializer=org.apache.spark.serializer.KryoSerializerKryoSerializer

(2) 若需调整 Spark 集成的 Hive 环境为 DLH（默认为 Hive），则要指定 DLH 的 metastore 配置：

spark.hadoop.hive.metastore.uris=thrift://sharedev1.hde.com:19083,thrift://sharedev2.hde.com:19083

【说明】metastore 的具体配置值可以在 DLH 服务的 dlh-site 的配置 hive.metastore.uris 获取。

操作 Hudi 表前，需对 Hudi 表做相应的了解，常见的 Hudi 表属性如表 3-5 所示。

表3-5 Hudi 表属性说明

属性	描述
hoodie.datasource.write.table.type	Hudi表类型，可以设置COPY_ON_WRITE或MERGE_ON_READ两种类型，缺失时默认为COPY_ON_WRITE
hoodie.datasource.write.recordkey.field	记录键字段。用作HoodieKey中recordKey部分的值，默认值为uuid
hoodie.datasource.write.precombine.field	实际写入之前在preCombining中使用的字段。当两个记录具有相同的键值时，使用precombine字段中最大的记录
hoodie.datasource.write.partitionpath.field	分区路径字段。用作HoodieKey中partitionPath部分的值

属性	描述
hoodie.table.name	Hive中注册的表名
hoodie.datasource.write.operation	写操作，支持upsert、insert、bulk_insert、delete、insert_overwrite等。默认为upsert
as.of.instant	时间点查询，未指定时查询最新
hoodie.datasource.query.type	查询类型，支持快照查询（snapshot）、增量查询(incremental)、读优化查询(read_optimized)
hoodie.datasource.read.begin.instanttime	增量查询时的开始时间
hoodie.datasource.read.end.instanttime	增量查询时的结束时间

说明

更多关于 Hudi 表参数的说明请参见 Hudi 官网 Spark Datasource Configs 和 Write Client Configs 章节，链接为：https://hudi.apache.org/docs/configurations。

3.8.1 Spark Shell 操作 Hudi 表

说明

操作 Hudi 的测试数据（模拟行程的数据）由 Hudi 自带的 DataGenerator 数据生成器生成。

操作 Hudi 表前需先完成以下准备工作：

(1) 执行 Spark-shell 启动命令：

```
spark-shell --conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer'
```

(2) 执行依赖包导入及变量定义（设置表名、基本路径和数据生成器），命令如下：

```
import org.apache.hudi.QuickstartUtils._
import scala.collection.JavaConversions._
import org.apache.spark.sql.SaveMode._
import org.apache.hudi.DataSourceReadOptions._
import org.apache.hudi.DataSourceWriteOptions._
import org.apache.hudi.config.HoodieWriteConfig._

val tableName = "hudi_trips_cow"
// 表默认存储在hdfs，使用file:///tmp/hudi_trips_cow 可将表数据存储到本地
val basePath = "/tmp/hudi_trips_cow"
val dataGen = new DataGenerator
```

【说明】导入 DataSourceReadOptions、DataSourceWriteOptions、HoodieWriteConfig 依赖后，可使用其封装的相关 hudi 表属性字段，也可以不导入依赖直接使用表 3-5 中的属性值。Sprk shell 操作 Hudi 表时，涉及的重要参数如表 3-6 所示。

表3-6 参数说明

参数名	说明
PRECOMBINE_FIELD_OPT_KEY	与 hoodie.datasource.write.precombine.field配置相同
RECORDKEY_FIELD_OPT_KEY	与hoodie.datasource.write.recordkey.field配置相同
PARTITIONPATH_FIELD_OPT_KEY	与hoodie.datasource.write.partitionpath.field配置相同
TABLE_NAME	与hoodie.datasource.write.table.name配置相同
TABLE_TYPE	与hoodie.datasource.write.table.type配置相同
QUERY_TYPE_OPT_KEY	与hoodie.datasource.query.type配置相同
BEGIN_INSTANTTIME_OPT_KEY	与hoodie.datasource.read.begin.instanttime配置相同
END_INSTANTTIME_OPT_KEY	与hoodie.datasource.read.end.instanttime配置相同
OPERATION_OPT_KEY	与hoodie.datasource.write.operation配置相同

1. 创建表

使用 spark shell 时，不需要执行单独的 create table 命令。如果表不存在，第一批数据写入时将会创建该表，且第一次数据写入时 save mode 需为 Overwrite。

2. 插入数据

插入数据时会自动创建表，表类型默认为 COPY_ON_WRITE。表类型支持 MERGE_ON_READ、COPY_ON_WRITE 两种，用户可通过 TABLE_TYPE 或 hoodie.datasource.write.table.type 来指定表类型。

示例：

(1) 插入表数据

```
val inserts = convertToStringList(dataGen.generateInserts(10))
val df = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
df.write.format("hudi").
options(getQuickstartWriteConfigs).
option(PRECOMBINE_FIELD_OPT_KEY, "ts").
option(RECORDKEY_FIELD_OPT_KEY, "uuid").
option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
option(TABLE_NAME, tableName).
mode(Overwrite).
save(basePath)
```

(2) 插入数据后，可在'/tmp/ hudi_trips_cow /<region>/<country>/<city>/'下查看生成数据。

3. 查询数据

查询数据时会将 HDFS 上存储的 Hudi 表数据加载到 DataFrame 中。

表查询类型支持：snapshot（快照查询）、incremental（增量查询）、read_optimized（读优化）。默认查询类型为 snapshot，可通过 QUERY_TYPE_OPT_KEY 或 hoodie.datasource.query.type 指定表查询类型。COPY_ON_WRITE 类型表不支持 read_optimized 查询类型。

（一）快照查询

示例：

- 遵循基础 select 语法，查询表数据

```
// basePath 支持使用分区路径 , 如/tmp/hudi_cow_table/americas/brazil/sao_paulo
val tripsSnapshotDF = spark.read.format("hudi").load(basePath)
tripsSnapshotDF.createOrReplaceTempView("hudi_trips_snapshot")
spark.sql("select fare, begin_lon, begin_lat, ts from hudi_trips_snapshot where fare > 20.0").show()
spark.sql("select _hoodie_commit_time, _hoodie_record_key, _hoodie_partition_path, rider, driver, fare from hudi_trips_snapshot").show()
```

- 支持时间点数据查询，数据加载时 option 中指定时间点即可

```
spark.read.format("hudi").option("as.of.instant",
"20220325103107282").load(basePath)
```

（二）增量查询

- 支持查询某个提交/压缩（时间点）后写入表中的新数据。
- 支持时间段查询，可通过 BEGIN_INSTANTTIME_OPT_KEY 或 hoodie.datasource.read.begin.instanttime 配置开始时间、END_INSTANTTIME_OPT_KEY 或 hoodie.datasource.read.end.instanttime 配置结束时间。

示例：

- 指定开始时间，查询从开始时间到现在的所有数据

```
spark.read.format("hudi").load(basePath).createOrReplaceTempView("hudi_trips_snapshot")
val commits = spark.sql("select distinct(_hoodie_commit_time) as commitTime from hudi_trips_snapshot order by commitTime").map(k => k.getString(0)).take(50)
val beginTime = commits(commits.length - 2)
val tripsIncrementalDF = spark.read.format("hudi").
option(QUERY_TYPE_OPT_KEY, QUERY_TYPE_INCREMENTAL_OPT_VAL).
option(BEGIN_INSTANTTIME_OPT_KEY, beginTime).
load(basePath)
tripsIncrementalDF.createOrReplaceTempView("hudi_trips_incremental")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from hudi_trips_incremental where fare > 20.0").show()
```

- 指定开始时间和结束时间，查询指定时间段的数据

```
val beginTime = "000" // Represents all commits > this time.
val endTime = commits(commits.length - 2) // commit time we are interested in
// 增量查询
val tripsPointInTimeDF = spark.read.format("hudi").
option(QUERY_TYPE_OPT_KEY, QUERY_TYPE_INCREMENTAL_OPT_VAL).
option(BEGIN_INSTANTTIME_OPT_KEY, beginTime).
option(END_INSTANTTIME_OPT_KEY, endTime).
load(basePath)
tripsPointInTimeDF.createOrReplaceTempView("hudi_trips_point_in_time")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from hudi_trips_point_in_time where fare > 20.0").show()
```

4. 更新数据

类似于插入新数据。使用 DataGenerator 数据生成器生成对现有数据（行程信息）的更新，加载到 DataFrame 中并将 DataFrame 写入 hudi 表中。

保存模式为 Append。通常情况下，除第一次插入数据（需创建表）使用 Overwrite 外，其他情况推荐使用 Append。

示例：

- 更新表数据

```
val updates = convertToStringList(dataGen.generateUpdates(10))
val df = spark.read.json(spark.sparkContext.parallelize(updates, 2))
df.write.format("hudi").
options(getQuickstartWriteConfigs).
option(PRECOMBINE_FIELD_OPT_KEY, "ts").
option(RECORDKEY_FIELD_OPT_KEY, "uuid").
option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
option(TABLE_NAME, tableName).
mode(Append).
save(basePath)
```

5. 删除数据

删除操作只支持 Append 模式。

示例：

- 删除表数据

```
// 获取记录总数
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").count()
// 拿到两条将要删除的数据
val ds = spark.sql("select uuid, partitionpath from hudi_trips_snapshot").limit(2)

// 执行删除
val deletes = dataGen.generateDeletes(ds.collectAsList())
val df = spark.read.json(spark.sparkContext.parallelize(deletes, 2))

df.write.format("hudi").
options(getQuickstartWriteConfigs).
option(OPERATION_OPT_KEY,"delete").
option(PRECOMBINE_FIELD_OPT_KEY, "ts").
option(RECORDKEY_FIELD_OPT_KEY, "uuid").
option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
option(TABLE_NAME, tableName).
mode(Append).
save(basePath)

// 向以上一样运行查询
spark.read.format("hudi").load(basePath).registerTempTable("hudi_trips_snapshot")
// 应返回 (total - 2) 条记录
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").count()
```

6. 覆盖写数据

覆盖写数据即使用新值覆盖表中的现有数据。插入的行可以由值表达式或查询结果指定。

示例：

- 覆盖写数据

```
spark.read.format("hudi").load(basePath).select("uuid","partitionpath").sort("partitionpath","uuid").show(100, false)

val inserts = convertToStringList(dataGen.generateInserts(10))
val df = spark.read.json(spark.sparkContext.parallelize(inserts, 2)).filter("partitionpath = 'americas/united_states/san_francisco'")
df.write.format("hudi").
options(getQuickstartWriteConfigs).
option(OPERATION.key(),"insert_overwrite").
```

```
option(PRECOMBINE_FIELD.key(), "ts").
option(RECORDKEY_FIELD.key(), "uuid").
option(PARTITIONPATH_FIELD.key(), "partitionpath").
option(TBL_NAME.key(), tableName).
mode(Append).
save(basePath)

// 和之前查询相比，san_francisco 分区数据有不一样的 key
spark.read.format("hudi").load(basePath).select("uuid","partitionpath").sort("parti
tionpath","uuid").show(100, false)
```

3.8.2 Spark SQL 操作 Hudi 表

操作 Hudi 表前需先完成以下准备工作：

(1) 执行 Spark-sql 启动命令：

```
spark-sql --conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer'
--conf 'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension'
```

1. 创建表

创建 Hudi 表时可以指定表属性值，主要配置如表 3-7 所示。

表3-7 参数说明

参数名	默认值	说明
primaryKey	uuid	• 表主键名称，支持多字段，用逗号分隔 • 与 hoodie.datasource.write.recordkey.field 配置相同
preCombineField	无	• 表的 pre-combine 字段 • 与 hoodie.datasource.write.precombine.field 配置相同
type	cow	• cow 代表写时复制（COPY-ON-WRITE）类型表，mor 代表读时合并（MERGE-ON-READ）类型表 • 与 hoodie.datasource.write.table.type 配置相同

说明

更多其它自定义 hudi 配置(如 index type、max parquest size 等)，请参见官网 hudi 配置，链接为：https://hudi.apache.org/docs/configurations。

示例：

• 创建非分区表

```
-- 默认 primaryKey 为 'uuid' ，当表定义 uuid 字段时，表参数中可以不指定 primaryKey
create table hudi_cow_nonpcf_tbl (
uuid int,
name string,
price double
) using hudi;

-- 指定 id 为 primaryKey
create table hudi_mor_tbl (
```

```
id int,
name string,
price double,
ts bigint
) using hudi
tblproperties (
type = 'mor',
primaryKey = 'id',
preCombineField = 'ts'
);
```

- 创建分区表

```
-- 创建一个 cow 类型的分区表，指定 preCombineField
create table hudi_cow_pt_tbl (
 id bigint,
 name string,
 ts bigint,
 dt string,
 hh string
) using hudi
tblproperties (
type = 'cow',
primaryKey = 'id',
preCombineField = 'ts'
)
partitioned by (dt, hh)
```

- 从已有表中创建表

从已存在的表（通过 spark shell 创建）中创建表，除了分区列（如果存在）之外，不需要指定 schema 和 properties。Hudi 可以自动识别 schema 和 properties。

```
-- 非分区表，location 需指向已存在表存储路径
create table hudi_existing_tbl0 using hudi
location '/tmp/hudi/dataframe_hudi_nonpt_table';

-- 分区表，location 需指向已存在表存储路径
create table hudi_existing_tbl1 using hudi
partitioned by (dt, hh)
location '/tmp/hudi/dataframe_hudi_pt_table';
```

- CTAS 创建表

为了更好的性能，表数据加载时，CTAS 使用 BULK_INSERT（批插入）作为 write operation。

```
-- CTAS: 创建 cow 类型的非分区表，不指定 preCombineField
create table hudi_ctas_cow_nonpcf_tbl
using hudi
tblproperties (primaryKey = 'id')
as
select 1 as id, 'a1' as name, 10 as price;

-- CTAS: 创建 cow 类型的分区表，指定 preCombineField
create table hudi_ctas_cow_pt_tbl
using hudi
tblproperties (type = 'cow', primaryKey = 'id', preCombineField = 'ts')
partitioned by (dt)
as
select 1 as id, 'a1' as name, 10 as price, 1000 as ts, '2021-12-01' as dt;
```

```
# CTAS ：将已有表的数据加载到新表中，分区字段在已有表中需存在。
create table hudi_ctas_cow_pt_tbl2 using hudi location '/tmp/ hudi_ctas_cow_pt_tbl2'
options (
type = 'cow',
primaryKey = 'id',
preCombineField = 'ts'
)
partitioned by (dt, hh) as select * from hudi_cow_pt_tbl;

# CTAS ：将已有文件数据加载到新表中，需先从文件中创建源表，然后在使用 create as select 加在数据到新表
# create managed parquet table
create table parquet_mngd using parquet location
'file:///tmp/parquet_dataset/*.parquet';
# CTAS by loading data into hudi table
create table hudi_ctas_cow_pt_tbl2 using hudi location 'file:/tmp/hudi/hudi_tbl/'
options (
  type = 'cow',
  primaryKey = 'id',
  preCombineField = 'ts'
 )
partitioned by (datestr) as select * from parquet_mngd;
```

2. 插入数据

- 默认情况下，若创建表时指定preCombineKey，执行insert into时upsert作为write operation，若未指定使用 insert 作为 write operation。
- 支持使用 bulk_insert 作为 write operation，只需设置 hoodie.sql.bulk.insert.enable 和 hoodie.sql.insert.mode 两个配置。

示例：

```
-- CTAS: 创建 cow 类型的非分区表，不指定 preCombineField
create table hudi_ctas_cow_nonpcf_tbl
using hudi
tblproperties (primaryKey = 'id')
as
select 1 as id, 'a1' as name, 10 as price;

-- CTAS: 创建 cow 类型的分区表，指定 preCombineField
create table hudi_ctas_cow_pt_tbl
using hudi
tblproperties (type = 'cow', primaryKey = 'id', preCombineField = 'ts')
partitioned by (dt)
as
select 1 as id, 'a1' as name, 10 as price, 1000 as ts, '2021-12-01' as dt;

# CTAS ：将已有表的数据加载到新表中，分区字段在已有表中需存在。
create table hudi_ctas_cow_pt_tbl2 using hudi location '/tmp/ hudi_ctas_cow_pt_tbl2'
options (
type = 'cow',
primaryKey = 'id',
preCombineField = 'ts'
)
partitioned by (dt, hh) as select * from hudi_cow_pt_tbl;
```

```
-- 指定preCombineField的表，使用upsert模式写数据
insert into hudi_mor_tbl select 1, 'a1_1', 20, 1001;
select id, name, price, ts from hudi_mor_tbl;
1   a1_1   20.0   1001

-- 指定preCombineField的表，使用bulk_insert模式写数据
set hoodie.sql.bulk.insert.enable=true;
set hoodie.sql.insert.mode=non-strict;

insert into hudi_mor_tbl select 1, 'a1_2', 20, 1002;
select id, name, price, ts from hudi_mor_tbl;
1   a1_1   20.0   1001
1   a1_2   20.0   1002
```

3. 查询数据

示例：

- 查询表数据

```
select fare, begin_lon, begin_lat, ts from hudi_trips_snapshot where fare > 20.0
```

4. 更新数据

Spark SQL 支持两种类型操作更新 hudi 表：MergeInto 和 Update。

示例：

- Update 更新数据

```
update hudi_mor_tbl set price = price * 2, ts = 1111 where id = 1;
update hudi_cow_pt_tbl set name = 'a1_1', ts = 1001 where id = 1;
```

【说明】Update 操作，需表指定 preCombineField。

- MergeInto 更新数据

```
--  merge into 非分区表，源表为hudi表
create table merge_source (id int, name string, price double, ts bigint) using hudi
tblproperties (primaryKey = 'id', preCombineField = 'ts');
insert into merge_source values (1, "old_a1", 22.22, 900), (2, "new_a2", 33.33, 2000),
(3, "new_a3", 44.44, 2000);

merge into hudi_mor_tbl as target
using merge_source as source
on target.id = source.id
when matched then update set *
when not matched then insert *;

-- merge into 分区表，源表为parquet表
create table merge_source2 (id int, name string, flag string, dt string, hh string) using
parquet;
insert into merge_source2 values (1, "new_a1", 'update', '2021-12-09', '10'), (2,
"new_a2", 'delete', '2021-12-09', '11'), (3, "new_a3", 'insert', '2021-12-09', '12');

merge into hudi_cow_pt_tbl as target
using (
select id, name, '1000' as ts, flag, dt, hh from merge_source2
) source
on target.id = source.id
when matched and flag != 'delete' then
```

```
update set id = source.id, name = source.name, ts = source.ts, dt = source.dt, hh =
source.hh
when matched and flag = 'delete' then delete
when not matched then
insert (id, name, ts, dt, hh) values(source.id, source.name, source.ts, source.dt,
source.hh);
```

5. 删除数据

示例：

- 删除表数据

```
delete from hudi_cow_nonpcf_tbl where uuid = 1;
delete from hudi_mor_tbl where id % 2 = 0;
```

6. 覆盖写数据

覆盖写数据即使用新值覆盖表中的现有数据。插入的行可以由值表达式或查询结果指定。

示例：

- 覆盖写数据

```
-- insert overwrite 非分区表
insert overwrite table hudi_mor_tbl select 99, 'a99', 20.0, 900;
insert overwrite table hudi_cow_nonpcf_tbl select 99, 'a99', 20.0;

-- insert overwrite 分区表，动态指定分区
insert overwrite table hudi_cow_pt_tbl select 10, 'a10', 1100, '2021-12-09', '10';

-- insert overwrite 分区表，静态指定分区
insert overwrite table hudi_cow_pt_tbl partition(dt = '2021-12-09', hh='12') select 13,
'a13', 1100;
```

4 开发指南

4.1 RDD操作

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是 Spark 中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD 具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

4.1.1 RDD 的创建方式

1. 通过读取文件生成

由外部存储系统的数据集创建，包括本地的文件系统，还有所有 Hadoop 支持的数据集，比如 HDFS、Cassandra、HBase 等。

使用示例：

```
val file = sc.textFile("/spark/hello.txt")
```

2. 通过并行化的方式创建 RDD

由一个已经存在的 Scala 集合创建。

使用示例：

```
scala> val array = Array(1,2,3,4,5)
array: Array[Int] = Array(1, 2, 3, 4, 5)
scala> val rdd = sc.parallelize(array)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[27] at parallelize at
<console>:26
```

3. 其他方式

- 读取数据库等其他的操作，也可以生成 RDD。
- RDD 可以通过其他的 RDD 转换而来。

4.1.2 RDD 的常用方法

1. Transformation

Transformation 主要做的就是将一个已有的 RDD 生成另外一个 RDD。Transformation 具有 lazy 特性(延迟加载)，Transformation 算子的代码不会真正被执行。只有当我们的程序里面遇到一个 action 算子的时候，代码才会真正的被执行，这种设计让 Spark 更加有效率地运行。

表4-1 常用的 Transformation

转换	说明
map(func)	返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成
filter(func)	返回一个新的RDD，该RDD由经过func函数计算后返回值为true的输入元素组成
flatMap(func)	类似于map，但是每一个输入元素可以被映射为0或多个输出元素（所以func应该返回一个序列，而不是单一元素）

转换	说明
mapPartitions(func)	类似于map，但独立地在RDD的每一个分片上运行，因此在类型为T的RDD上运行时，func的函数类型必须是Iterator[T] => Iterator[U]
sample(withReplacement, fraction, seed)	根据fraction指定的比例对数据进行采样，可以选择是否使用随机数进行替换，seed用于指定随机数生成器种子
union(otherDataset)	对源RDD和参数RDD求并集后返回一个新的RDD
intersection(otherDataset)	对源RDD和参数RDD求交集后返回一个新的RDD
distinct([numTasks]))	对源RDD进行去重后返回一个新的RDD
groupByKey([numTasks])	在一个(K,V)的RDD上调用，返回一个(K, Iterator[V])的RDD
reduceByKey(func, [numTasks])	在一个(K,V)的RDD上调用，返回一个(K,V)的RDD，使用指定的reduce函数，将相同key的值聚合到一起，与groupByKey类似，reduce任务的个数可以通过第二个可选的参数来设置
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	先按分区聚合，再总的聚合，每次要跟初始值交流，例如：aggregateByKey(0)(_+_,_+_) 对k/y的RDD进行操作
sortByKey([ascending], [numTasks])	在一个(K,V)的RDD上调用，K必须实现Ordered接口，返回一个按照key进行排序的(K,V)的RDD
sortBy(func,[ascending], [numTasks])	与sortByKey类似，但是更灵活。第一个参数是根据什么排序，第二个是怎么排序false倒序，第三个排序后分区数，默认与原RDD一样
join(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的RDD上调用，返回一个相同key对应的所有元素对在一起的(K,(V,W))的RDD，相当于内连接（求交集）
cogroup(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的RDD上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的RDD
cartesian(otherDataset)	两个RDD的笛卡尔积
repartition(numPartitions)	重新分区
repartitionAndSortWithinPartitions(partitioner)	重新分区+排序
combineByKey	合并相同的key的值
partitionBy（partitioner）	对RDD进行分区
cache	RDD缓存，可以避免重复计算从而减少时间，区别：cache内部调用了persist算子，cache默认就一个缓存级别MEMORY-ONLY，而persist则可以选择缓存级别
persist	
leftOuterJoin	leftOuterJoin类似于SQL中的左外关联left outer join，返回结果以前面的RDD为主，关联不上的记录为空。只能用于两个RDD之间的关联，如果要多个RDD关联，多关联几次即可
rightOuterJoin	rightOuterJoin类似于SQL中的有外关联right outer join，返回结果以参数中的RDD为主，关联不上的记录为空。只能用于两个RDD之间的关联，如果要多个RDD关联，多关联几次即可

2. Action

触发代码的运行，一段 Spark 代码里面至少需要有一个 action 操作。

表4-2 常用的 Action

动作	说明
reduce(func)	通过func函数聚集RDD中的所有元素
collect()	在驱动程序中，以数组的形式返回数据集的所有元素
count()	返回RDD的元素个数
first()	返回RDD的第一个元素（类似于take(1)）
take(n)	返回一个由数据集的前n个元素组成的数组
takeSample(withReplacement,num, [seed])	返回一个数组，该数组由从数据集中随机采样的num个元素组成，可以选择是否用随机数替换不足的部分，seed用于指定随机数生成器种子
saveAsTextFile(path)	将数据集的元素以textfile的形式保存到HDFS文件系统或者其他支持的文件系统，对于每个元素，Spark将会调用toString方法，将它装换为文件中的文本
saveAsSequenceFile(path)	将数据集中的元素以Hadoop sequencefile的格式保存到指定的目录下，可以使HDFS或者其他Hadoop支持的文件系统
countByKey()	针对(K,V)类型的RDD，返回一个(K,Int)的map，表示每一个key对应的元素个数
foreach(func)	在数据集的每一个元素上，运行函数func进行更新
aggregate	先对分区进行操作，再总体操作

4.1.3 示例：WordCount 代码编写

1. 创建名称为 sparkdemo 的 Maven 工程

2. 添加 pom 依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.test.spark</groupId>
    <artifactId>sparkdemo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <scala.version>2.11.4</scala.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.11</artifactId>
            <version>2.4.0-cdh6.2.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.4.0-cdh6.2.0</version>
        </dependency>
```

```
    </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
            <version>3.2.2</version>
            <configuration>
                <scalaVersion>2.11.4</scalaVersion>
                <javacArgs>
                    <javacArg>-source</javacArg>
                    <javacArg>${java.version}</javacArg>
                    <javacArg>-target</javacArg>
                    <javacArg>${java.version}</javacArg>
                </javacArgs>
            </configuration>
            <executions>
                <execution>
                    <id>scala-compile</id>
                    <phase>process-resources</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>scala-test-compile</id>
                    <phase>process-test-resources</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.6</version>
            <configuration>
                <skipIfEmpty>true</skipIfEmpty>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

3. 代码实现

```
package com.test.spark
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SparkSession

object SparkWordCount {
  def main(args: Array[String]) {
    // $example on:init_session$
    val spark = SparkSession
      .builder
      .appName("Spark WordCount")
```

```
        .getOrCreate()

      //读取文件 生成 RDD
      val file : RDD[String] = spark.sparkContext.textFile(args(0))
      //把每一行数据按照，分割
      val word: RDD[String] = file.flatMap(_.split(","))
      //让每一个单词都出现一次
      val wordOne: RDD[(String, Int)] = word.map((_,1))
      //单词计数
      val wordCount: RDD[(String, Int)] = wordOne.reduceByKey(_+_)
      //按照单词出现的次数 降序排序
      val sortRdd: RDD[(String, Int)] = wordCount.sortBy(tuple => tuple._2,false)
      //将最终的结果进行保存
      sortRdd.saveAsTextFile(args(1))

      spark.stop()
    }
  }
```

4. 创建样例文件

(1) 切换用户：

su - hdfs

(2) 创建 word.txt 文件：

yarn,hdfs,spark,hive,zookeeper,spark,mapreduce,yarn,hdfs,hive,zookeeper,hdfs

(3) 将 word.txt 上传 hdfs：

hdfs dfs -put word.txt /tmp

5. 提交 wordcount 任务

上传 jar 包到/opt 目录下

spark-submit --master yarn --class com.test.spark.SparkWordCount /opt/sparkdemo-1.0-SNAPSHOT.jar /tmp/word.txt /tmp/15

6. 执行结果

查看输出结果，执行以下命令：

hdfs dfs -cat /tmp/15/*

图4-1 执行结果

```
sh-4.1$ hdfs dfs -cat /tmp/15/*
(hdfs,3)
(hive,2)
(zookeeper,2)
(yarn,2)
(spark,2)
(mapreduce,1)
```

说明

在执行 spark-submit 提交任务时，如果指定 master 为 yarn-client 和 yarn-cluster，则不能用 root 用户提交任务。

4.2 DataSet操作

DataSet 是分布式的数据集合。它集中了 RDD 的优点（强类型和可以用强大 lambda 函数）以及 Spark SQL 优化的执行引擎。DataSet 可以通过 JVM 的对象进行构建，可以用函数式的转换（map/flatmap/filter）进行多种操作。DataSet API 在 Scala 和 Java 中都是可以用的。

4.2.1 Pom 依赖

首先需要在 pom.xml 文件中添加依赖。

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.4.0-cdh6.2.0</version>
</dependency>
```

4.2.2 代码实现

```
import org.apache.spark.sql.SparkSession

object SparkSQLExample {
  def main(args: Array[String]) {
  // $example on:init_session$
  val spark = SparkSession
    .builder()
    .appName("Spark SQL basic example")
    .getOrCreate()
  val df = spark.read.parquet("/tmp/users.parquet")

  // Displays the content of the DataFrame to stdout
  df.show()

  import spark.implicits._
  // Print the schema in a tree format
  df.printSchema()
  df.select("name").show()
  df.createOrReplaceTempView("users")
  val sqlDF = spark.sql("SELECT * FROM users")
  sqlDF.show()
  spark.stop()
}
}
```

4.2.3 创建样例文件

(1) 切换到 hdfs 用户：

su - hdfs

(2) 将 spark 安装目录下自带的样例文件上传到 hdfs：

hdfs dfs -put /usr/hdp/3.0.1.0-187/spark2/examples/src/main/resources/users.parquet /tmp

4.2.4 提交任务

spark-submit --master yarn --class com.test.spark.SparkSQLExample /opt/sparkdemo-1.0-SNAPSHOT.jar

图4-2 执行结果

```
[hdfs@test01 opt]$ spark-submit --master yarn --class com.test.spark.SparkSQLExample /opt/sparkdemo-1.0-SNAPSHOT.jar
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+

root
 |-- name: string (nullable = true)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+------+
|  name|
+------+
|Alyssa|
|   Ben|
+------+

+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

4.3 JDBC/ODBC连接Spark SQL

在项目开发中，用户可以通过 JDBC 方式连接 Spark ThriftServer。

4.3.1 pom 依赖

首先需要在 pom.xml 文件中添加 hive-jdbc 的依赖

```
<!--添加 hive-jdbc 依赖-->
 <dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-jdbc</artifactId>
    <version>2.1.1-cdh6.2.0</version>
</dependency>
```

4.3.2 代码实现

```
package com.test.spark

import java.sql.DriverManager
object JDBCTest {

    def main(args: Array[String]): Unit = {
      Class.forName("org.apache.hive.jdbc.HiveDriver")
      val conn = DriverManager.getConnection("jdbc:hive2://192.168.0.2:10016","hdfs","")
      val pstmt = conn.prepareStatement("select * from person")
      val rs = pstmt.executeQuery()
```

```
        while (rs.next()) {
          println("id:" + rs.getInt("id") +
            " , name:" + rs.getString("name"))
        }
        rs.close()
        pstmt.close()
        conn.close()
      }
}
```

4.3.3 创建样例

1. 启动 SparkSQL，执行命令：

beeline -u jdbc:hive2://node2:10016

其中，node2 为 Spark ThriftServer 安装节点的 IP 域名。

2. 创建表并插入数据

- 创建表：

  create table person (id int, name string);

- 插入数据：

  insert into person values(1,'jdbcTest');

- 执行查询：

  select * from person;

4.3.4 提交任务

spark-submit --master yarn --class com.test.spark.JDBCTest /opt/sparkdemo-1.0-SNAPSHOT.jar

图4-3 执行结果

```
sh-4.2$ spark-submit --master yarn --class com.test.spark.JDBCTest /opt/sparkdemo-1.0-SNAPSHOT.jar
id:1 , name:jdbcTest
```

5 最佳实践

5.1 未开启Kerberos环境下Spark-HBase实践案例

1. 场景描述

使用 Spark 读取 HBase 中数据。

2. 准备工作

创建 HBase 表：

(1) 在命令行执行 hbase shell 进入 HBase 命令行：

```
[root@node4 ~]# hbase shell
hbase(main):001:0>
```

(2) 创建名字为 test 的 HBase 表，并插入两行数据：

```
hbase(main):002:0> create 'test','info'
hbase(main):003:0> put 'test','row1','info:name','xiaoming'
hbase(main):004:0> put 'test','row2','info:name','xiaohong'
```

3. 添加依赖

Spark 读取 HBase 数据需要依赖以下 jar 包，在上述 pom 文件下添加：

```
<dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.4.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.4.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-common</artifactId>
      <version>2.1.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-client</artifactId>
      <version>2.1.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-mapreduce</artifactId>
      <version>2.1.0-cdh6.2.0</version>
    </dependency>
```

```
  </dependencies>
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

4. 非 kerberos 代码实现

```
package com.test.spark
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.spark.sql.SparkSession
object HBaseTest {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession
      .builder
      .appName("Spark HBase Test")
        .master("local[2]")
      .getOrCreate()

    val conf = HBaseConfiguration.create()
    conf.set(TableInputFormat.INPUT_TABLE, "test")

    val hBaseRDD = spark.sparkContext.newAPIHadoopRDD(conf, classOf[TableInputFormat],
classOf[ImmutableBytesWritable], classOf[Result])

    println(hBaseRDD.count())
  }
}
```

5. 非 kerberos 提交任务

使用带依赖的包 sparkdemo-1.0-SNAPSHOT-jar-with-dependencies.jar 进行测试。

spark-submit --master yarn --driver-class-path /etc/hbase/conf:/etc/hadoop/conf --class com.test.spark.HBaseTest /opt/sparkdemo-1.0-SNAPSHOT-jar-with-dependencies.jar

图5-1 执行结果

```
[hdfs@cloudos opt]$ spark-submit --master yarn --driver-class-path /etc/hbase/
conf:/etc/hadoop/conf --class com.test.spark.HBaseTest /opt/sparkdemo-1.0-SNAP
SHOT-jar-with-dependencies.jar
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.propertie
s
2
```

5.2 Kerberos环境下Spark-HBase实践案例

1. 场景描述

Kerberos 环境下使用 Spark 读取 HBase 中数据。

2. 准备工作

(1) 创建 HBase 表：

a. 在命令行执行 hbase shell 进入 HBase 命令行：

[root@node4 ~]# hbase shell

hbase(main):001:0>

b. 创建名字为 test 的 HBase 表，并插入数据：

hbase(main):002:0> create 'test','cf1'

hbase(main):005:0> put 'test','r1','cf1:f1','value1'

Took 0.8027 seconds

hbase(main):006:0> put 'test','r1','cf1:f2','value2'

Took 0.0098 seconds

hbase(main):007:0> put 'test','r2','cf1:f1','v1'

Took 0.0107 seconds

hbase(main):008:0> put 'test','r2','cf1:f2','v2'

Took 0.0465 seconds

hbase(main):009:0> scan 'test'

```
hbase(main):009:0> scan 'test'
ROW                                              COLUMN+CELL
 r1                                              column=cf1:f1, timestamp=1600067338184, value=value1
 r1                                              column=cf1:f2, timestamp=1600067344710, value=value2
 r2                                              column=cf1:f1, timestamp=1600067355825, value=v1
 r2                                              column=cf1:f2, timestamp=1600067362980, value=v2
2 row(s)
```

(2) 将相关配置文件等 copy 到 resource 目录。

创建/opt/resource 目录，并将 core-site.xml、hbase.headless.keytab、hbase-site.xml、hdfs-site.xml、krb5.conf 文件放置到该目录。

(3) 将 hbase-site.xml 文件拷贝到 spark 的 conf 目录下。

3. 添加依赖

Kerberos 环境下 Spark 读取 HBase 数据需要依赖以下 jar 包，在上述 pom 文件下添加以下内容：

```xml
<dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.4.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.4.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-common</artifactId>
      <version>2.1.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-client</artifactId>
      <version>2.1.0-cdh6.2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-mapreduce</artifactId>
      <version>2.1.0-cdh6.2.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
          <archive>
            <manifest>
            </manifest>
          </archive>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
```

```
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>net.alchim31.maven</groupId>
        <artifactId>scala-maven-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <scalaVersion>2.11.5</scalaVersion>
          <javacArgs>
            <javacArg>-source</javacArg>
            <javacArg>${java.version}</javacArg>
            <javacArg>-target</javacArg>
            <javacArg>${java.version}</javacArg>
          </javacArgs>
        </configuration>
        <executions>
          <execution>
            <id>scala-compile</id>
            <phase>process-resources</phase>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
          <execution>
            <id>scala-test-compile</id>
            <phase>process-test-resources</phase>
            <goals>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <skipIfEmpty>true</skipIfEmpty>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

4. kerberos 代码实现

```
package com.test.spark

import java.io.{BufferedWriter, FileOutputStream, OutputStreamWriter}
import java.util.Base64;
```

```scala
import com.esotericsoftware.kryo.Kryo
import org.apache.hadoop.fs.Path
import org.apache.hadoop.hbase.client.{Result, Scan}
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.hadoop.hbase.protobuf.ProtobufUtil
import org.apache.hadoop.hbase.util.Bytes
import org.apache.hadoop.hbase.{CellUtil, HBaseConfiguration}
import org.apache.hadoop.security.UserGroupInformation
import org.apache.spark.serializer.KryoRegistrator
import org.apache.spark.{SparkConf, SparkContext}



object TableOutPutData {
  def main(args: Array[String]) {
    System.setProperty("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    System.setProperty("spark.kryo.registrator", "com.test.spark.MyRegistrator")


    val resource = "/opt/resource/"
    val keytab = resource + "hbase.headless.keytab"
    val principal = "hbase-hpp0910@HPP0910.COM"
    val krb5Path = resource + "krb5.conf"
    System.setProperty("java.security.krb5.conf", krb5Path)


    val hbConf = HBaseConfiguration.create
    hbConf.addResource(new Path(resource + "hbase-site.xml"))
    hbConf.addResource(new Path(resource + "core-site.xml"))
    hbConf.addResource(new Path(resource + "hdfs-site.xml"))



    System.out.println(hbConf.get("hbase.security.authentication"))
    hbConf.set("hadoop.security.authentication", "kerberos")
    UserGroupInformation.setConfiguration(hbConf)
    UserGroupInformation.loginUserFromKeytab(principal, keytab)


    val conf = new SparkConf()
    val sc = new SparkContext(conf)


    val scan = new Scan()
    scan.addFamily(Bytes.toBytes("cf1"))
```

```
    val proto = ProtobufUtil.toScan(scan)



    val scanToString = Base64.getEncoder().encodeToString(proto.toByteArray);
    //需要事先在hbase建好"test"表，并有列簇"cf1"
    hbConf.set(TableInputFormat.INPUT_TABLE, "test")
    hbConf.set(TableInputFormat.SCAN, scanToString)



    //  Obtain the data in the table through the Spark interface.
    val rdd = sc.newAPIHadoopRDD(hbConf, classOf[TableInputFormat],
classOf[ImmutableBytesWritable], classOf[Result])



    // Traverse every row in the HBase table and print the results
    val out = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(resource +
"out.log"), "UTF-8"))



    rdd.collect().foreach(x => {
      val key = x._1.toString
      val it = x._2.listCells().iterator()
      while (it.hasNext) {
        val c = it.next()
        val family = Bytes.toString(CellUtil.cloneFamily(c))
        val qualifier = Bytes.toString(CellUtil.cloneQualifier(c))
        val value = Bytes.toString(CellUtil.cloneValue(c))
        val tm = c.getTimestamp
        println(" Family=" + family + " Qualifier=" + qualifier + " Value=" + value + "
TimeStamp=" + tm)
        out.write(" Family=" + family + " Qualifier=" + qualifier + " Value=" + value + "
TimeStamp=" + tm)
        out.write("\n")
      }
    })



    out.flush()
    out.close()
    sc.stop()



  }
}
```

```
class MyRegistrator extends KryoRegistrator {
  override def registerClasses(kryo: Kryo) {
    kryo.register(classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable])
    kryo.register(classOf[org.apache.hadoop.hbase.client.Result])
    kryo.register(classOf[Array[(Any, Any)]])
    kryo.register(classOf[Array[org.apache.hadoop.hbase.Cell]])
    kryo.register(classOf[org.apache.hadoop.hbase.NoTagsKeyValue])


kryo.register(classOf[org.apache.hadoop.hbase.protobuf.generated.ClientProtos.RegionLoad
Stats])
  }
}
```

5. kerberos 环境提交任务

使用带依赖的包 testspark-1.0-SNAPSHOT-jar-with-dependencies.jar 进行测试。

spark-submit --class com.test.spark.TableOutPutData --master yarn
--keytab=/opt/resource/hbase.headless.keytab --principal=hbase-hpp0910@HPP0910.COM
--deploy-mode client /opt/testspark-1.0-SNAPSHOT-jar-with-dependencies.jar

图5-2 执行结果

```
kerberos
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
 Family=cf1 Qualifier=f1 Value=value1 TimeStamp=1600067338184
 Family=cf1 Qualifier=f2 Value=value2 TimeStamp=1600067344710
 Family=cf1 Qualifier=f1 Value=v1 TimeStamp=1600067355825
 Family=cf1 Qualifier=f2 Value=v2 TimeStamp=1600067362980
```

5.3 SparkStreaming-Kafka实践案例

1. 场景描述

利用 SparkStreaming 消费指定 topic 数据，并打印到控制台。

2. 创建 Kafka 生产者

(1) 创建 topic

/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --create --zookeeper
node1:2181,node2:2181,node3:2181 --replication-factor 1 --partitions 1 --topic test

其中：node1:2181,node2:2181,node3:2181 为 Zookeeper 服务器地址。

(2) 生产数据

/usr/hdp/3.0.1.0-187/kafka/bin/kafka-console-producer.sh --broker-list node1:6667 --topic test

其中：node1 为 KAFKA_BROKER 所在节点的 IP 域名。

3. pom 依赖文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>com.test.spark</groupId>
    <artifactId>sparkdemo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.11</artifactId>
            <version>2.4.0-cdh6.2.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.11</artifactId>
            <version>2.4.0-cdh6.2.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka_2.11</artifactId>
            <version>2.1.0-cdh6.2.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka-0-10_2.11</artifactId>
            <version>2.4.0-cdh6.2.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

4. 代码实现

```
import java.util.*;
import org.apache.spark.SparkConf;
import org.apache.spark.TaskContext;
```

```
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.*;
import org.apache.spark.streaming.kafka010.*;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.common.serialization.StringDeserializer;

public class JavaSparkStreamingKafka {
   public static void main(String[] args) throws Exception {
      Map<String, Object> kafkaParams = new HashMap<>();
      kafkaParams.put("bootstrap.servers", "node1:6667");
      kafkaParams.put("key.deserializer", StringDeserializer.class);
      kafkaParams.put("value.deserializer", StringDeserializer.class);
      kafkaParams.put("group.id", "use_a_separate_group_id_for_each_stream");
      kafkaParams.put("auto.offset.reset", "latest");
      kafkaParams.put("enable.auto.commit", false);

      Collection<String> topics = Arrays.asList("test");

      SparkConf sparkConf = new SparkConf().setAppName("javasparkstreamingkafkademo");
      JavaStreamingContext javaStreamingContext = new JavaStreamingContext(sparkConf,
Durations.seconds(1));
      JavaInputDStream<ConsumerRecord<String, String>> dStream =
KafkaUtils.createDirectStream(javaStreamingContext,
            LocationStrategies.PreferConsistent(),
            ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams));

      dStream.foreachRDD(rdd -> {
         OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();
         rdd.foreachPartition(consumerRecords -> {
            OffsetRange o = offsetRanges[TaskContext.get().partitionId()];
            //System.out.println(o.topic() + " " + o.partition() + " " + o.fromOffset() +
" " + o.untilOffset());
            //Lambda expressions are not supported at this language
            while(consumerRecords.hasNext()){
               String value = consumerRecords.next().value();
               System.out.println(value);
            }
         });
         ((CanCommitOffsets) dStream.inputDStream()).commitAsync(offsetRanges);
      });

      javaStreamingContext.start();
      javaStreamingContext.awaitTermination();
   }
}
```

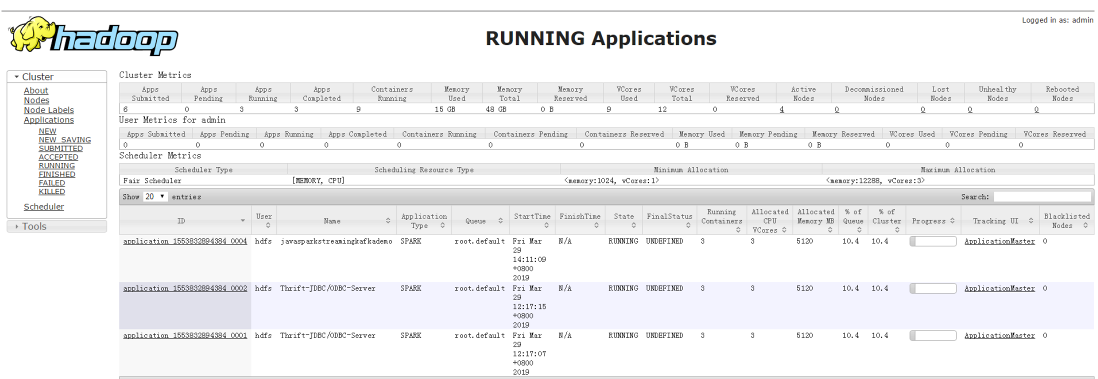
5. 提交任务

spark-submit --class com.test.spark.JavaSparkStreamingKafka --master yarn --deploy-mode client /opt/sparkdemo-1.0-SNAPSHOT.jar

6. 执行结果

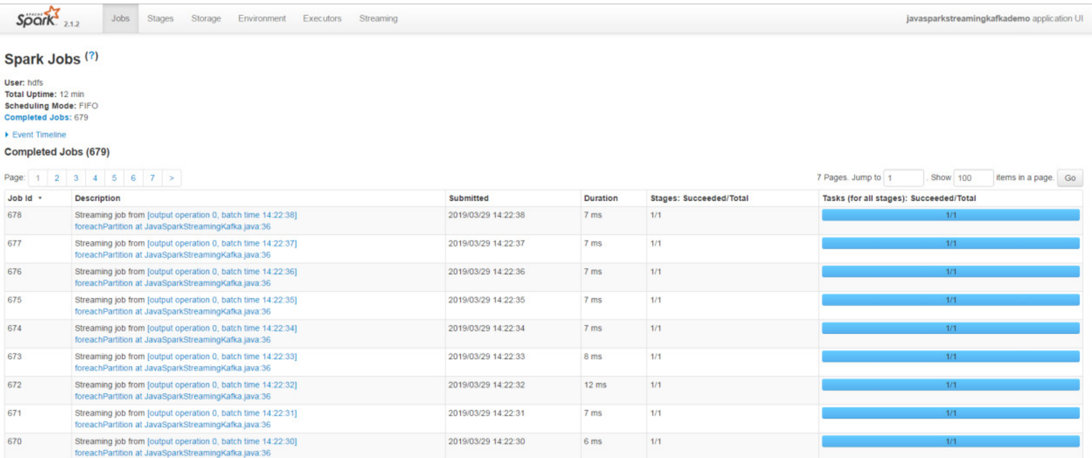
(1) 通过 YARN 的 ResourceManager UI 可以查看到正在运行的“javasparkstreamingkafkademo”任务。

图5-3 查看任务



(2) 点击某个任务的 Tracking UI，可以进入该任务的 Spark 监控界面查看执行情况。

图5-4 执行情况



6 常见问题解答

6.1 调优类

6.1.1 增加并行度

并行度是指在 Spark 任务中，各个 stage 的 task 的数量，也就代表 Spark 任务各个 stage 的并行度。合理的并行度设置，能够充分利用集群资源，提升任务执行效率。在设置并行度时，可以从以下几个方面考虑：

1. spark.default.parallelism

该参数用户设置每个 stage 的默认 task 数量。如果不设置这个参数，会导致 Spark 根据 HDFS 的 block 数量来设置 task 的数量，默认是一个 HDFS block 对应一个 task。这个默认设置的数量是偏少的，因此会导致集群资源浪费。Spark 官网建议的设置原则是，设置该参数为：num-executors * executor-cores 的 2~3 倍。例如任务分配的 executor 的总 CPU 数量为 300 个，那么设置 900 个 task 是合适的。

2. RDD.repartition

给 RDD 重新设置 partition 的数量。

3. reduceByKey 算子指定 partition 的数量

例如：val rdd2 = rdd1.reduceByKey(_+_, 10)

4. 执行 join 算子时增加父 RDD 的 partition 数量

在执行 join 算子时，子 RDD 的 partition 由父 RDD 中最多的 partition 的数量来决定，因此使用 join 算子时，增加父 RDD 中 partition 的数量。

5. spark.sql.shuffle.partitions

该参数设置 Spark SQL 中 shuffle 过程的 partition 数量。

6.1.2 序列化和数据压缩

1. 通过序列化优化

Spark 中提供了两种序列化的方式：使用标准序列化库进行序列化和使用 Kyro 库进行序列化。标准序列化库兼容性好，但体积大、速度慢，Kyro 库兼容性略差，但是体积小，速度快。可以通过 spark.serializer=org.apache.spark.serializer.KyroSerializer 来判断是否使用 Kyro 进行序列化，这个配置参数决定了 Shuffle 进行网络传输和 RDD 写入磁盘时，是否使用这个序列化器。注意 Spark 任务序列化只支持 JavaSerializer。

2. 通过压缩方式优化

在 Spark 中对 RDD 或者 Broadcast 数据进行压缩，是提高数据吞吐量和性能的一种手段。压缩和解压缩也会带来额外的 CPU 开销，但可以大量减少磁盘的存储空间，同时压缩后的文件在磁盘间传输和 I/O 以及网络传输的通信开销也会减少。因此，最好对 I/O 密集型的作业使用数据压缩。

Spark 支持 LZF 和 Snappy 等压缩方式，相关的配置如表 6-1 所示。

表6-1 压缩配置

参数名称	参数说明	建议值
spark.broadcast.compress	决定Broadcast变量是否进行压缩	true
spark.rdd.compress	决定是否压缩一个已经序列化的RDD	true
spark.io.compression.codec	决定压缩方式	org.apache.spark.io.LZFCompressionCodec
spark.io.compression.snappy.block.size	设置Snappy压缩算法的块大小	32k

6.1.3 文件格式

Spark 支持多种文件格式及压缩方式，根据不同的应用环境进行合理的选择。如果每次计算只需要其中的某几列，可以使用列式文件格式，以减少磁盘 I/O，常用的列式有 parquet、rcfile。如果文件过大，将原文件压缩可以减少磁盘 I/O，例如：gzip、snappy、lzo。

6.1.4 使用广播变量

- 如果 Spark 任务在执行过程中需要使用一个特别大的数据集合时，推荐将该数据集合广播到每个节点上，此时 task 就会在本地查找 Broadcast 的数据集合。如果不使用 Broadcast，则每次任务需要数据集合时，都会把数据序列化到 task 任务中，不但耗时，还使任务变得很大。使用广播变量能够有效的减少内存消耗以及数据到节点的网络传输消耗。
- 大表和小表做 join 操作时可以把小表 Broadcast 到各个节点，从而把 join 操作转变成普通的操作，减少了 shuffle 操作。

6.1.5 优化数据结构

优化数据结构，可以避免 Java 语法特性中所导致的额外内存的开销。

- 优先使用数组以及字符串，而不是集合类。即优先用 array，而不是 ArrayList、LinkedList、HashMap 等集合。
- 避免使用多层嵌套的对象结构。
- 对于有些能够避免的场景，尽量使用 int 替代 String。

6.1.6 对多次使用的 RDD 进行持久化

如果程序中，基于某个 RDD 进行了多次 transformation 或者 action 操作。那么就非常有必要对其进行持久化操作，从而避免对一个 RDD 反复进行计算。

此外，如果 RDD 的持久化数据可能丢失的情况下仍然需要保证其高性能，那么可以对 RDD 进行 Checkpoint 操作。

6.1.7 缓存表

对于一条 SQL 语句中可能多次使用到的表，可以对其进行缓存。

6.2 运维类

1. 执行大数据量查询，查询结束停止 driver 时，报错 ERROR LiveListenerBus: SparkListenerBus has already stopped! Dropping event SparkListenerExecutorMetricsUpdate(2999,WrappedArray())

- 可能原因

  Spark 中，netty 通信信息是异步处理的。当 Driver 退出后，SparkListenerBus 消息队列已关闭，导致 SparkListenerExecutorMetricsUpdate 信息无法处理。

- 解决方法

  该错误只是日志显示问题，不影响任务运行。

2. 执行大数据量查询，出现报错：java.lang.IllegalArgumentException: Cannot allocate a page with more than 17179869176 bytes

- 可能原因

  在 executor 的 ShuffleTask 中，会对数据进行外部排序，排序过程先将数据加载到内存缓冲区，如果内存达到一定比例时，才会将数据溢写到磁盘，在 Spark 源码中溢写磁盘前会判断数据所占内存大小是否超过了 17179869176(17G)，如果超过 17G 就会报出上述异常。

  该 Spark 任务启动脚本为：

  ```
  spark-sql --driver-memory 30G --driver-cores 5--num-executors 2000 --executor-cores 6
  --executor-memory 30G -master yarn—conf spark.network.timeout=500 -conf
  spark.scheduler.listenerbus.eventqueue.size=10000000—conf spark.driver.maxResultSize=0
  ```

  其中：设置 executor 的内存为 30G，最大排序内存空间为 30G*0.6=18G，就会大于 17G，导致报错。

- 解决方法

  将 executor 内存调小，不超过 28G。

3. 执行大数据量查询，查询结束（已显示出查询结果，及所用时间）但 state 进度条还在显示运行

- 可能原因

  当 task 个数超过 spark.ui.retainedTasks（默认值为 100000）时，task 信息就会被垃圾回收站回收，而导致 stage 进度条有问题。

- 解决方法

  可调大 spark.ui.retainedTasks 参数值，如：10000000。

4. 执行大数据量查询，报错 org.apache.spark.storage.BlockNotFoundException: Block broadcast_41_piece0 not found

- 可能原因

  该错误是表示block信息丢失。造成block信息丢失的原因有很多，如网络通信异常，executor 挂掉等。但 Spark 中有重试机制，会从其他地方获取 block，然后执行任务。

- 解决方法

  该错误不影响任务执行。

5. Spark 任务运行失败，提示 Running as root is not allowed，spark 任务提交后一直未运行

- 可能原因：

- 不支持使用 root 用户提交 yarn 任务（yarn 源码限定）。
- yarn 资源不足。

- 解决方法

a. 查看当前用户，shell 输入 whoami，在 yarn UI 页面查看对应服务是否处于 ACCEPTED 状态。

b. 使用集群用户或 hdfs 用户提交任务；或者停掉部分无用的服务，释放资源后重试服务是否运行。

6. Spark 启动失败，报错 KeyError: 'getgrnam(): name not found: spark' Writing File['/usr/hdp/current/spark2-historyserver/conf/spark-defaults.conf'] because contents don't match

- 可能原因

系统异常导致/usr/hdp/current/spark2-historyserver/conf/目录下 Spark 相关文件所属分组被置为 20012。

- 解决方法

a. 查看/usr/hdp/current/spark2-historyserver/conf/所属组。

b. 执行以下操作：

groupadd spark \\ 新增 spark 组

chown spark:spark spark-defaults.conf \\ 修改（所有）异常文件的分组

7. 修改 Zookeeper 端口后，spark2 thrift Server 无法启动，报错 Connection timed out for connection string([IP]:2181)

- 可能原因

检查是否修改了 Zookeeper 端口，spark2 thrift server 的 HA 依赖于 Zookeeper，修改 Zookeeper 端口后 Spark2 无法连接到 Zookeeper 报错。

- 解决方法

a. 查看 Spark2 参数中的{{zookeeperConnectString}}是否与 ZooKeeper 的一致。

b. 如果不一致，将 spark2 参数中的{{zookeeperConnectString}}改为[IP]:[port]，此处填写 Zookeeper 的 IP 和 port。

8. Spark2 historyserver 界面快速链接选项加载慢，Spark thriftserver 启动后，过一段时间就会挂掉

- 可能原因

Spark 任务 eventlog 存在 HDFS 的/spark2-history 路径，该路径占用空间太大，会造成页面加载缓慢问题。

- 解决方法

a. 检查 HDFS 上/spark2-history 路径下 eventlog 的数量。

b. 切换到 hdfs 用户，执行 hdfs dfs –du –s -h /spark2-history，检查其大小。如果太大，则需要配置 Spark 相关参数对该目录进行清理。

9. Spark 连接 MySQL 失败

- 可能原因

未加载MySQL连接驱动包。

- 解决方法

  后台连接MySQL查看MySQL服务器是否正常，如果正常，再查看代码是否加载MySQL驱动包，正确加载MySQL连接驱动包即可。

10. 重启YARN、MapReduce组件之后，Spark ThriftServer偶现告警无法恢复

- 可能原因

  重启YARN、MapReduce组件导致Spark ThriftServer服务长时间无法注册，重启失败。

- 解决方法

  重启YARN、MapReduce组件之后，观察Spark是否出现告警且长时间未消失现象。然后重新启动Spark组件。

11. 运行Spark压力测试脚本，Spark用户执行Spark-sql提交SQL任务后概率失败，SparkContext异常停止

- 可能原因

  Spark用户执行Spark-sql提交的SQL任务会概率失败，SparkContext异常停止；删除YARN的Minimum user ID for submitting job：1000参数限制后，异常消失。

- 解决方法

  Spark用户uid大于YARN设置1000所致。建议修改YARN配置项min_user_id小于Spark用户id号，然后重启Spark组件。

12. 运行Spark任务时，Spark Web UI无法访问

- 可能原因

  Spark Web UI默认关闭，此时通过Spark History Server服务可以查看任务状态。

- 解决方法

  若计划开启Spark Web UI功能，需要修改Spark的配置，即修改spark2-defaults中spark.ui.enabled配置项的值为true。

13. 执行Spark任务时，使用集群超级用户登录YARN UI，点击ApplicationMaster跳转Spark的UI页面，上报权限不足问题

- 可能原因

  只有提交任务的用户才可查看对应任务的Spark UI页面。

- 解决方法

  使用提交任务的用户登录YARN UI。

14. Spark任务作业出现300秒无法执行广播broadcast而失败

- 问题现象

  Spark组件通过oozie批量下发较多（200）任务，运行大概2小时左右执行失败failed，任务日志：org.apache.spark.SparkException: Could not execute broadcast in 300 secs。

- 可能原因

  Spark对于小于指定阈值的表进行join关联处理时，会进行数据广播，将数据广播到执行的工作节点。若此时网络IO占用紧张，将会导致300秒内，无法广播完成而超时失败。

- 解决方法

可以通过调整阈值或超时时间进行调整，比如：

- 调整阈值，如：spark.sql.autoBroadcastJoinThreshold=-1（不广播）。
- 调整超时时间，如：spark.sql.broadcastTimeout=600（需要根据当时网络 IO 进行调整，单位为秒）。

15. Spark 任务偶现 java.io.FileNotFoundException

- 问题现象

  运行 Spark 任务偶现 java.io.FileNotFoundException: File does not exist:hdfs://mycluster/warehouse/tenant-hive/oms_prd/fct_oms_setl_info_detl_di/dt=20190204/000366_0，Itis possible the underlying files have been updated. You can explicitlyinvalidate the cache in Spark by running 'REFRESH TABLE tableName' command inSQL or by recreating the Dataset/DataFrame involved。

- 可能原因

  为提升性能，SparkSQL 在内存中缓存文件 meta 信息，并发对表同时进行读/写时，可能出现缓存元数据信息不一致的现象。

- 解决方法

  出现问题的表进行 select 查询前，执行一次 refresh table 操作。

16. Spark-sql 针对 Hive 分区表通过 insert [overwrite] into 插入大量动态分区导致 HDFS 小文件过多进而 HDFS 响应缓慢

- 问题现象

  由于对插入的动态分区的源数据不具有重复性或维度性，导致每条数据一个动态分区，造成了大量的动态分区（百万、千万甚至上亿）。

- 可能原因

  对于插入的业务数据不了解或误操作。

- 解决方法

  通过删除 HDFS 上大量的小文件后 HDFS 恢复正常后，重新分析插入语句及业务数据是否适合该种场景。

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.2 组件架构 1-1
1.2.1 Flink 集群架构 1-1
1.2.2 Flink 特点 1-2
1.3 应用场景 1-2
1.3.1 事件驱动应用 1-2
1.3.2 数据分析应用 1-3
1.3.3 数据管道 1-3
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 查看组件的日志信息 2-1
2.2 运行状态监控 2-1
2.3 快速使用指导 2-4
2.3.1 非 Kerberos 环境 2-4
2.3.2 Kerberos 环境 2-5
2.4 快速链接 2-8
3 使用指南 3-1
3.1 常用命令 3-1
3.2 Client 下载/安装/使用/卸载 3-3
3.2.1 下载 Client 安装包 3-3
3.2.2 安装 Client 3-4
3.2.3 访问组件 3-5
3.2.4 卸载 Client 客户端 3-6
3.3 添加/删除进程 3-6
3.3.1 添加进程 3-6
3.3.2 删除进程 3-7
4 开发指南 4-1
4.1 开发模型 4-1
4.1.1 Flink 抽象层次 4-1
4.1.2 Flink 程序和数据流 4-2
4.1.3 Flink 程序并行度 4-4

4.2 常用 API ······4-5
4.2.1 Java API ······4-6
4.2.2 Scala API ······4-12
4.3 Flink on YARN 任务提交模式 ······4-17
4.3.1 Application Mode（推荐） ······4-17
4.3.2 Per-Job Cluster Mode（推荐） ······4-18
4.3.3 Session Mode（可选） ······4-18
5 最佳实践 ······5-1
5.1 Flink 开发程序（非 Kerberos 环境） ······5-1
5.1.1 Flink DataStream 读 Kafka 写 Kafka ······5-1
5.1.2 Flink DataStream 读 Kafka 写 Elasticsearch ······5-10
5.1.3 Flink DataStream 读 Kafka 写 HBase ······5-28
5.1.4 Flink DataStream 读 Kafka 写 Hive ······5-39
5.1.5 Flink DataStream 读 Kafka 写 Redis ······5-50
5.1.6 Flink DataStream 读 Kafka 写 Hudi 同步 Hive ······5-59
5.2 Flink 开发程序（Kerberos 环境） ······5-63
5.2.1 Flink DataStream 读 Kafka 写 Kafka ······5-63
5.2.2 Flink DataStream 读 Kafka 写 Elasticsearch ······5-73
5.2.3 Flink DataStream 读 Kafka 写 HBase ······5-91
5.2.4 Flink DataStream 读 Kafka 写 Hive ······5-101
5.2.5 Flink DataStream 读 Kafka 写 Redis ······5-113
5.2.6 Flink DataStream 读 Kafka 写 Hudi 同步 Hive ······5-122
5.3 FlinkSQL-cdc ······5-125
5.3.1 CDC 简介 ······5-125
5.3.2 FlinkCDC 使用 ······5-126
6 常见问题解答 ······6-1
6.1 常用配置 ······6-1
6.2 调优 ······6-2
6.3 运维类问题 ······6-4

1 组件简介

1.1 组件概述

Flink 是一个流处理框架和分布式计算引擎，用于在无边界和有边界的数据流上进行有状态的计算，并能以内存速度和任意规模进行流和批计算。

1.2 组件架构

1.2.1 Flink 集群架构

如图 1-1 所示，Flink 集群由 Flink Client、JobManager 和 TaskManager 三部分组成。其中：

- Flink Client

  Client 主要为用户提供向 Flink 系统提交任务（流式作业）的能力。

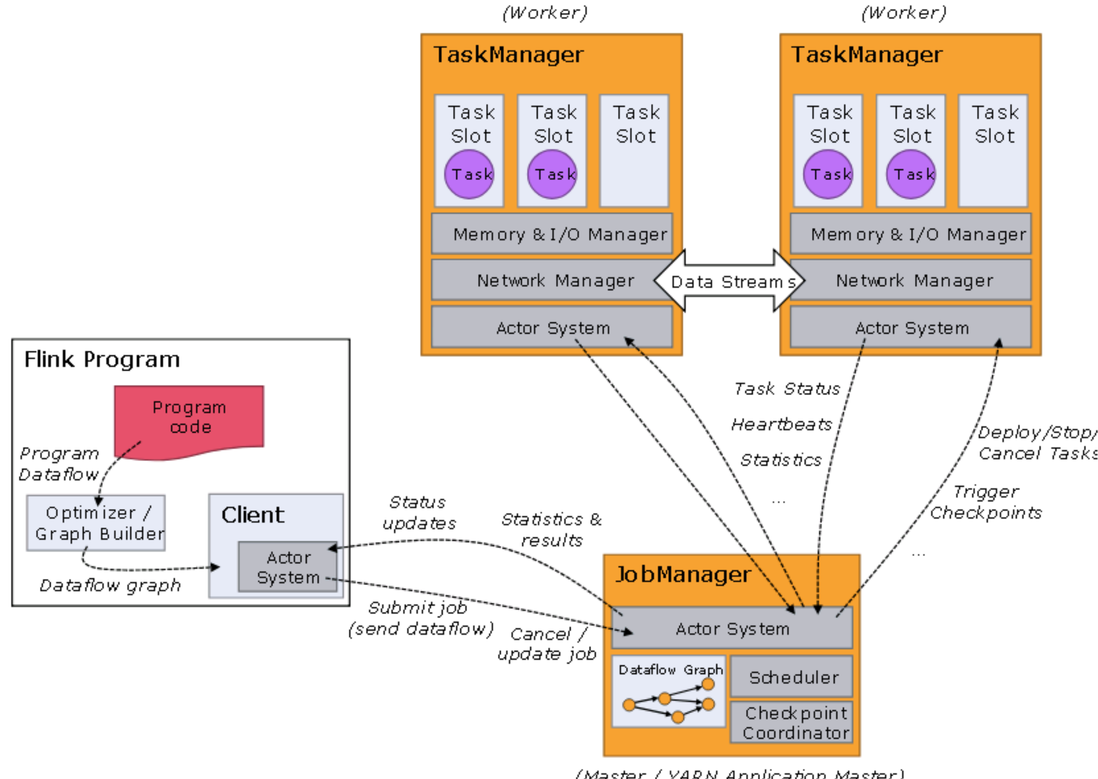
- JobManager（Master）

  JobManager 是 Flink 系统的管理节点，管理所有的 TaskManager，可以决定用户任务在哪些 Taskmanager 执行。JobManager 可以有多个，但只有一个主 JobManager。

- TaskManager（Worker）

  TaskManager 是 Flink 系统的业务执行节点，执行具体的用户任务。TaskManager 可以有多个，且各个 TaskManager 平等。

图1-1 Flink 运行架构



1.2.2 Flink 特点

Flink 系统提供的关键能力如下：

- 低时延：提供毫秒级（ms）低时延的处理能力。
- Exactly Once：提供异步快照机制，保证所有数据真正只处理一次。Flink 通过状态和两次提交协议来保证了端到端的 Exactly Once 语义。
- 水平扩展能力：TaskManager 支持手动水平扩展。

1.3 应用场景

1.3.1 事件驱动应用

Flink 强大的状态管理机制使得其能够很好的支持 CEP 和 Savepoint 等功能。Flink 支持事件时间、高度自定义的窗口逻辑，以及 ProcessFunction 提供的精细时间控制，以此实现高级的业务逻辑。而且，Flink 为复杂事件处理中的模型提供了库。

Flink 的 Event-Driven 应用具有以下优势：

- 和查询远程数据库不同，事件驱动应用程序访问其本地的数据性能优越，包括吞吐量和延迟。事件驱动应用程序可以异步和增量地周期性把检查点持久化到远程存储，且检出检查点的动作对常规的计算影响很小。

- 在分层架构中，多个应用程序经常共享相同的数据库，数据库中的任何改变（例如：数据布局的改变）均会导致应用程序更新或服务扩展需要重新协调。但对于事件驱动应用程序，每个程序只负责自己的数据，数据改变和应用程序的更改仅需要较少的协调成本。

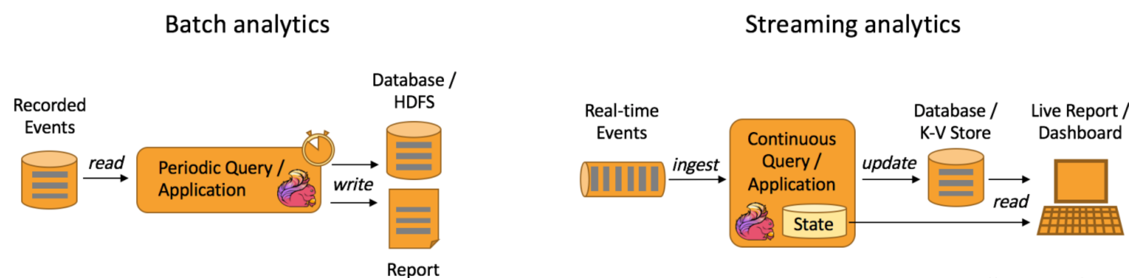
1.3.2 数据分析应用

Flink 使用统一的 SQL 即可实现流式数据和批量数据的数据分析功能，并支持用户自定义 UDF 函数，同时还具有低延迟的优势。Flink 为批量分析和持续流处理提供了很好的支持，包括：

- Flink 提供 ANSI 兼容 SQL 接口，以统一的语义进行批处理和流查询。
- 无论运行在批数据还是实时事件流上，SQL 查询/计算结果相同。
- 支持丰富的用户自定义函数，保证自定义代码可以在 SQL 查询上执行。另外，若需要更多的自定义逻辑，Flink 的 DataStream API 或 DataSet API 可以提供很多低级别的控制。
- Flink 的 gelly 库提供算法，能够支持在批量数据集上进行大规模和高性能的分析。

Flink 支持流和批处理数据分析应用程序，如图 1-2 所示。

图1-2 Flink 支持流和批处理分析应用程序图

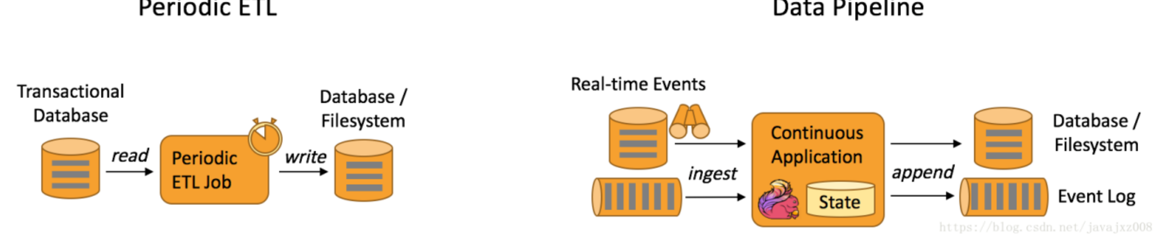


1.3.3 数据管道

Flink 的数据管道功能类似于传统的 ETL 工具，支持对数据执行提取、转换、加载到其他数据源中的操作。与传统 ETL 工具相比，Flink 还能对流式数据进行加工处理，且具有低延迟的优异性能。Flink 的 SQL 接口（或表 API）以及其对用户定义函数的支持，可以执行许多常见的数据转换或数据处理任务。使用更通用的 DataStream API 可以实现具有更高级需求的数据管道。Flink 为 Kafka、Kinesis、ElasticSearch 和 JDBC 数据库系统等提供了丰富的连接器。

周期性 ETL 作业与连续数据管道之间的区别如图 1-3 所示。

图1-3 周期性 ETL 作业与连续数据管道的对比示意图



2 快速入门

2.1 组件安装

说明

- 在 Hadoop 集群中，安装 Flink 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- Flink 依赖 Zookeeper、HDFS 和 YARN，安装 Flink 时必须同时安装 Zookeeper 和 Hadoop。其中 Hadoop 包括 HDFS、MapReduce、YARN 组件。
- Flink 包括 Client 客户端、SQL Gateway 进程。Flink 任务运行在 YARN 模式时，内部保证作业的 HA，安装时无特殊要求。

2.1.1 查看组件的日志信息

表2-1 组件日志路径说明

组件	日志路径
Flink	• Flink 启动日志路径：/var/de_log/flink/user_${user.name}/，其中${user.name}是指执行任务的用户名 • Flink 任务运行日志路径，可以通过 YARN 的 UI 界面查看

2.2 运行状态监控

1. 查看组件详情

进入 Flink 组件详情页面，如图 2-1 所示。组件详情页面主要展示概要、部署拓扑、配置详情和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 概要：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。

- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：添加进程、下载 Client、组件检查、查看操作记录等。

图2-1 组件详情

2. 组件检查

执行 Flink 组件检查时，会提交 Flink 自带的 WordCount 示例任务至 YARN 上并检测该任务的运行结果，若 Flink 组件检查成功，则表示 Flink 可以提交任务至 YARN，并且 YARN 也有资源可以接受并运行 Flink 任务。

集群在使用过程中，根据实际需要，可对 Flink 执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：
  - 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
    - 在集群详情页面选择[组件]页签，单击业务组件列表中 Flink 组件对应的<组件检查>按钮。
    - 在集群详情页面选择[组件]页签，单击业务组件列表中 Flink 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
  - 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

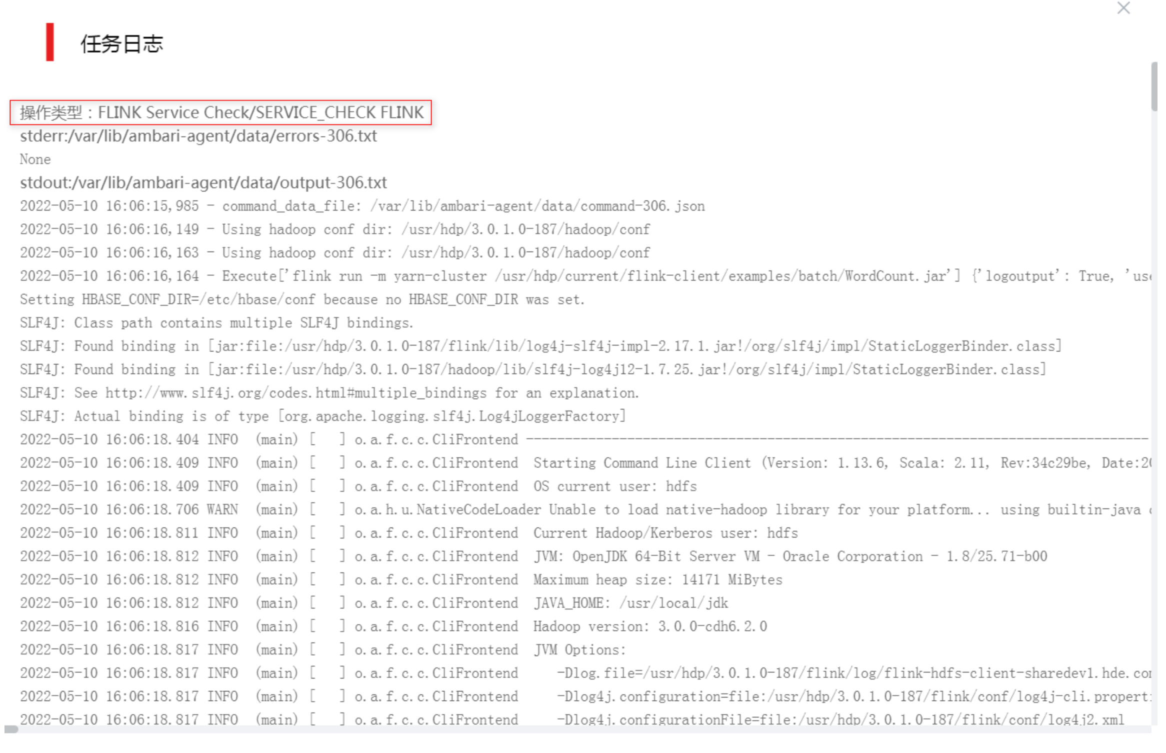
(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态。如图 2-2 所示，表示该组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“FLINK Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情



2.3 快速使用指导

注意

- 根据大数据集群是否开启 Kerberos 认证，用户提交 Flink 任务时的认证方式不同，详情请参见本章节内容。
- 在租户集群中，普通用户提交 Flink 任务至 YARN 之前，需提前申请对应的 YARN 资源队列。关于 YARN 资源队列的申请及使用，详情请参见 YARN 手册。

Flink 任务既可以通过集群用户提交，又可以通过组件超级用户提交。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
    - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。
    - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 HDFS 组件的 hdfs 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 Kerberos 环境

说明

非 Kerberos 环境下，组件超级用户或集群用户均不需要用户做身份认证即可直接提交 Flink 任务。

Flink 组件中提供基本测试样例，本章节以 Flink 自带测试样例展示其基本使用。

(1) 连接大数据集群内安装 Flink 的任意一节点或安装 Flink Client 的节点。

(2) 提交 Flink 自带样例任务，命令如下：

```
/usr/hdp/3.0.1.0-187/flink/bin/flink  run  -t  yarn-per-job
/usr/hdp/3.0.1.0-187/flink/examples/batch/WordCount.jar
```

其中各参数说明如下：

- /usr/hdp/3.0.1.0-187/flink/bin/：表示绝对路径，可缺省。
- -t yarn-per-job：表示在 YARN 上以 Per-Job Cluster 模式运行。
- /usr/hdp/3.0.1.0-187/flink/examples/batch/WordCount.jar：Flink 自带样例 jar 包的路径。

图2-4 提交 Flink 自带样例任务示例

```
[root@node35 ~]# flink run -t yarn-per-job /usr/hdp/3.0.1.0-187/flink/examples/batch/WordCount.jar
Setting HBASE_CONF_DIR=/etc/hbase/conf because no HBASE_CONF_DIR was set.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/flink/lib/log4j-slf4j-impl-2.17.1.jar!/org/s
lf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf
4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
2022-07-28 02:05:59.697 INFO  (main) [   ] o.a.f.c.c.CliFrontend --------------------------------
--------------------------------------------------------------
2022-07-28 02:05:59.702 INFO  (main) [   ] o.a.f.c.c.CliFrontend  Starting Command Line Client (Ver
sion: 1.13.6, Scala: 2.11, Rev:0f40199, Date:2022-05-27T10:35:20+02:00)
```

(3) 任务提交后，观察控制台打印信息，若出现“Program execution finished”，即说明任务运行完成。

2.3.2 Kerberos 环境

注意

- Kerberos 环境下，Flink 默认使用集群超级用户提交任务，且无需进行用户身份认证。
- Kerberos 环境下，若想使用集群普通用户或组件超级用户提交 Flink 任务，则必须首先进行用户身份认证，认证方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。
- 本章节操作需要切换至具有相关权限的用户。

1. Kerberos 环境下用户身份认证

如果大数据集群开启 Kerberos，若想使用集群普通用户或组件超级用户提交 Flink 任务，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。使用集群超级用户提交 Flink 任务时，无需进行用户身份认证。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

Flink 任务可以通过集群用户提交到 YARN 上。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**
  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的/etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：

  ```
  chown user1 /etc/security/keytabs/user1.keytab
  ```

b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：

klist -k user1.keytab

【说明】如图 2-5 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-5 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

su user1

kinit -kt user1.keytab user1@TENANTC.COM

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-6 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

注意

通过集群普通用户提交 Flink 任务时，当集群普通用户完成身份认证之后，还需要通过后台修改如下配置：

- 需自行下载普通用户的 keytab 文件，下载完普通用户的 keytab 文件之后，需要修改该文件的所有者为普通用户本身，并修改 flink-conf 配置 security.kerberos.login.keytab 为自定义 keytab 文件路径。
- 修改 flink-conf 配置 security.kerberos.login.principal 为普通用户的 keytab 文件对应 principal。

配置修改完成后，重启 Flink，然后集群普通用户即可提交 Flink 任务成功。

（二）组件超级用户身份认证

Flink 任务可以通过组件超级用户提交到 YARN 上，比如 hdfs 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 hdfs 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 hdfs 的认证文件"hdfs.headless.keytab"。

【说明】在 Flink Client 节点上，需要将 hdfs 的认证文件"hdfs.headless.keytab"上传节点的/etc/security/keytabs/目录下进行认证。

(2) 使用 ***klist*** 命令查看 hdfs.headless.keytab 的 principal 名称，命令如下：

```
klist -k hdfs.headless.keytab
```

【说明】如图 2-7 所示，红框内容即为 hdfs.headless.keytab 的 principal 名称。

图2-7 认证文件的 principal 名称

```
[root@hppnode1 keytabs]# klist -k hdfs.headless.keytab
Keytab name: FILE:hdfs.headless.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
```

(3) 切换至用户 hdfs，并执行身份验证的命令如下：

```
su hdfs
kinit -kt hdfs.headless.keytab hdfs-testshare@TESTSHARE.COM
```

【说明】其中：hdfs.headless.keytab 为 hdfs 的认证文件，hdfs-testshare@TESTSHARE.COM 为 hdfs.headless.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

2. 运行 Flink 基本测试样例

用户身份认证成功后，即可运行 Flink 组件中提供基本测试样例，详情请参见 2.3.1 非 Kerberos 环境。

2.4 快速链接

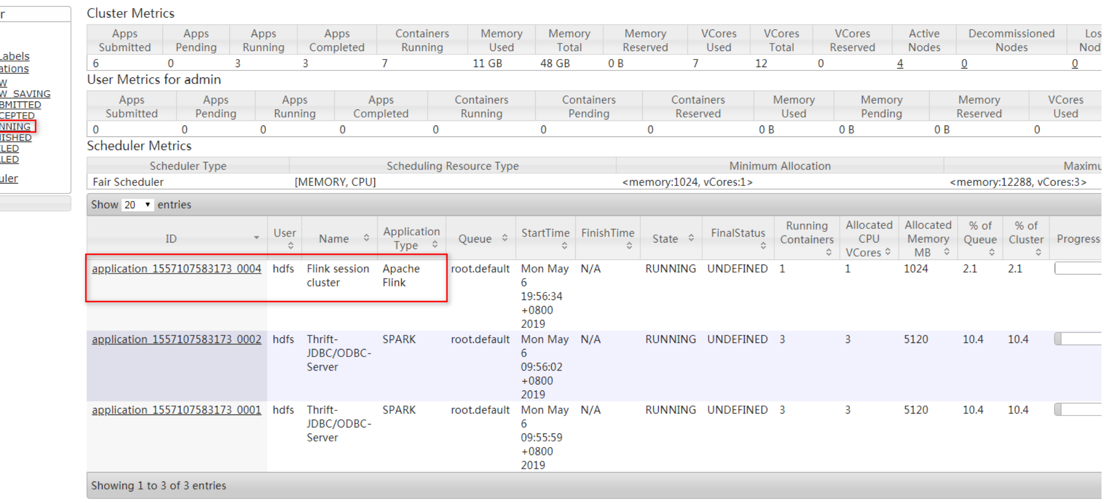
说明

Flink 无快速链接，Flink 任务通过 YARN 的快速链接进行监控，关于 YARN 快速链接的访问方式详情请参见 YARN 组件手册。

Flink 任务提交后，一般需要打开 YARN 作业监控页面了解该任务的执行情况。

(1) 进入 YARN 作业监控页面，如图 2-8 所示。在 YARN 上启动 Flink 后，Flink 会被 YARN 当做一个 application 运行。

图2-8 YARN 作业监控页面



(2) 如图 2-8 所示，Flink 提交的作业在运行状态时，可以在 YARN 作业监控页面的 RUNNING 状态的作业栏中查找到对应作业，点击 ID 进入该作业的详情页面，此时再点击 Tracking URL 即可进入 Flink 的 Dashboard 界面，如图 2-9 所示。此时，在 Overview 页面可以查看 Flink 集群资源情况以及任务执行状况。

- 点击具体的 Job Name，可以查看提交 Flink 任务每个子任务之间执行流程。
- 点击指定子任务，可以查看对应的信息，如：Subtasks、Task Metrics、Watermarks、Accumulators、BackPressure 等。如图 2-10 所示，其中：
  - Subtasks 部分可查看各个 Subtasks 运行 Start Time(开始时间)、End Time(结束时间)、Duration(运行时长)、Bytes Received、Records Received、Bytes Sent、Records Sent、Status(Subtasks 状态)等信息。
  - Watermarks 下显示作业各个 Subtask 中 Watermark 数值。
  - Accumulators 下能观察 Subtask 在运行期间的数据变化，但是只能在任务执行结束之后才能获得累加器的最终结果。
  - BackPressure 对作业运行期间 Checkpoint 信息进行汇总。

图2-9 Flink 的 Web 界面

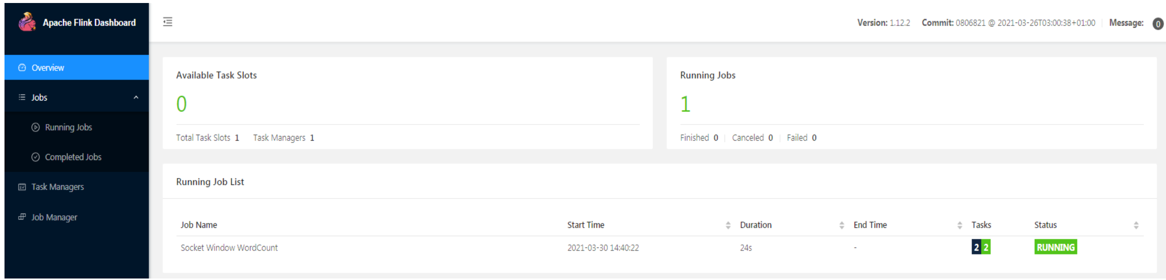
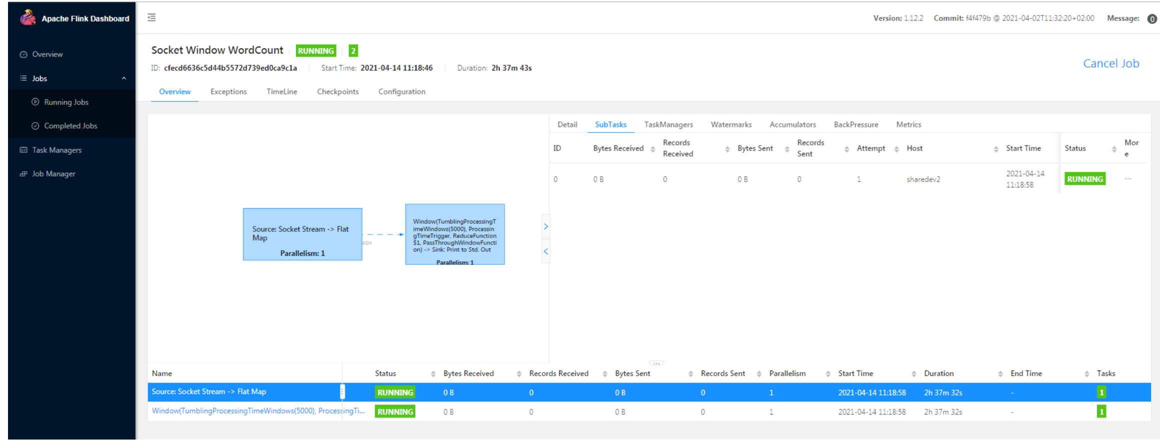


图2-10 Flink 作业运行时的 Subtasks 展示信息



3 使用指南

3.1 常用命令

Flink 命令用于 Flink 任务的提交和运行控制，Flink 完整的命令形式如下：

/usr/hdp/3.0.1.0-187/flink/bin/flink <ACTION> [OPTIONS] [arguments]

执行 Flink 命令时，自动打印图 3-1 所示提示信息。

图3-1 执行 Flink 命令的提示信息

```
[root@node1 flink]# /usr/hdp/3.0.1.0-187/flink/bin/flink
Setting HADOOP_CONF_DIR=/etc/hadoop/conf because no HADOOP_CONF_DIR was set.
./flink <ACTION> [OPTIONS] [ARGUMENTS]

The following actions are available:

Action "run" compiles and runs a program.
```

Flink 命令中的<ACTION>选项对应不同的控制功能，不同选项的功能及详细参数不同，如下：

- flink run [OPTIONS] <arguments>用于提交流处理作业，并指定作业运行时的参数。

表3-1 flink run 部分参数说明

参数名称	参数说明
-c,--class <classname>	指定入口类
-C,--classpath <url>	指定需加载进job的类所在路径。该参数需保证集群内所有节点均能访问
-d,--detached	指定-d参数则job提交到cluster模式 默认（未指定该参数）情况下，job提交到client模式，此时会在客户端多出一个CliFrontend进程。cluster模式下，job提交完成后客户端会自动退出
-n,--allowNonRestoredState	指定允许跳过无法恢复的svaepoint状态。如果用户程序中移除了savepoint触发时涉及到的操作，则提交任务时需要指定该参数
-p,--parallelism <parallelism>	指定job运行时并行度。如果配置中也指定了并行度，实际运行时的并行度以该参数为准
-q,--sysoutLogging	压缩日志打印为标准输出
-s,--fromSavepoint <savepointPath>	指定job恢复使用的svaepoint所在的路径
-m,yarn-cluster	YARN模式下，-m参数值必须为yarn-cluster
-D <property=value>	运行在YARN时，指定运行时使用的配置值，如：yarn.application.queue、yarn.application.name、yarn.application.node-label等
-yd,--yarndetached	提交任务到cluster模式下，与-d参数功能类似
-yid,--yarnapplicationId <arg>	运行在YARN时，指定applicationId
-yj,--yarnjar <arg>	运行在YARN时，指定jar路径

参数名称	参数说明
-yjm,--yarnjobManagerMemory <arg>	指定JobManager内存。默认单位是MB，用户可自行指定
-yn,--yarncontainer <arg>	指定container个数。一个TaskManager对应一个Container
-ys,--yarnslots <arg>	指定taskManager中slot数量
-yt,--yarnship <arg>	加载指定目录下的文件
-ytm,--yarntaskManagerMemory <arg>	指定每个container的内存数。默认单位是MB，用户可自行指定
-yz,--yarnzookeeperNamespace <arg>	运行在YARN时，HA模式下用于创建ZooKeeper子目录的命名空间
-z,--zookeeperNamespace <arg>	HA模式下用于创建ZooKeeper子目录的命名空间

- flink info [OPTIONS] <jar-file> <arguments>用于查看程序的优化后执行计划。

表3-2 flink info 参数说明

参数名称	参数说明
-c,--class <classname>	指定入口类
-p,--parallelism <parallelism>	job运行时并行度。如果配置中也指定了并行度，实际运行时的并行度以该参数为准

- flink list [OPTIONS]用于查看正在运行或者已被调度的程序。

表3-3 flink list 部分参数说明

参数名称	参数说明
-a,--all	打印出所有job以及jobId
-r,--running	仅打印出正在运行的job以及jobId
-s,--scheduled	仅打印出被调度但未开始执行的job以及jobId
-yid,--yarnapplicationId <arg>	yarn-cluster模式下，指定applicationId
-z,--zookeeperNamespace <arg>	yarn-cluster模式下，指定用于创建ZooKeeper子路经的命名空间

- flink stop [OPTIONS] <Job Id>用于停止正在运行的流任务。

表3-4 flink stop 部分参数说明

参数名称	参数说明
-d,--drain	在savepoint和停止pipeline操作前发送MAX_WATERMARK
-p,--savepointPath <savepointPath>	Savepoint数据保存路径，未指定时默认保存在state.savepoints.dir配置的目录下
-yid,--yarnapplicationId <arg>	yarn-cluster模式下，指定applicationId
-z,--zookeeperNamespace <arg>	yarn-cluster模式下，指定用于创建ZooKeeper子路经的命名空间

- flink cancel [OPTIONS] <Job Id>用于取消正在运行的任务。

表3-5 flink cancel 参数说明

参数名称	参数说明
-s,--withSavepoint <targetDirectory>	取消时触发svaepoint，其中targetDirectory目标目录可选
-yid,--yarnapplicationId <arg>	yarn-cluster模式下，指定applicationId
-z,--zookeeperNamespace <arg>	yarn-cluster模式下，指定用于创建ZooKeeper子路经的命名空间

- flink savepoint [OPTIONS] <arguments> [<target directory>]用于触发正在运行中任务的 savepoint 或者触发已存在的 savepoint。

表3-6 flink savepoint 部分参数说明

参数名称	参数说明
-d,--dispose <arg>	触发的svaepoint所在的路径
-j,--jarfile <jarfile>	Flink程序的jar文件
-yid,--yarnapplicationId <arg>	yarn-cluster模式下，指定applicationId
-z,--zookeeperNamespace <arg>	yarn-cluster模式下，指定用于创建ZooKeeper子路经的命名空间

3.2 Client下载/安装/使用/卸载

大数据集群提供了下载 Flink Client 的功能。在客户端节点上安装 Flink 的 Client 后，即可直接连接大数据集群中的 Flink，进行组件维护、任务管理等操作。

3.2.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在[集群管理/集群列表]页面，单击集群名称进入集群详情页面，在集群已安装的组件列表中单击 Flink 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-2 所示。

图3-2 下载 Client 窗口

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：

- 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
- 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中的组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。

- 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
- 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.2.2 安装 Client

注意

- 安装 Client 的节点必须与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置网络连接，仅非 root 用户需要执行此操作，root 用户可跳过此步骤。

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

```
./install.sh <安装路径> -o
```

【说明】

- 可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为 /usr/hdp/3.0.1.0-187/。
- 还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

```
./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>
```

3.2.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

  ```
  source bigdata_env
  ```

- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同：
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行校验即可直接访问组件并执行相关操作。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行身份认证之后，才可访问组件并执行相关操作。关于 Kerberos 环境下，进行用户认证的操作方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。Kerberos 环境下，在集群外的 Client 节点上执行 Flink 任务时，需要修改任务代码，详情请参考 5 最佳实践章节中的 Kerberos 场景相关说明。

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

3.2.4 卸载 Client 客户端

大数据集群重装之后，之前安装的 Client 客户端将不可用。此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 安装目录下执行以下命令：

```
./uninstall.sh
```

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

3.3 添加/删除进程

3.3.1 添加进程

Flink 支持添加 Flink Sql Gateway、Flink Client 进程。

- 在并发查询任务较多的情况下，可以考虑增加 Flink Sql Gateway 进程，分摊查询负载，提高并发数。
- 若主机新扩容机器时未勾选 client，后期可以通过添加 Flink Client 进程在新机器上增加。

1. 操作示例

说明

- 本章节仅以添加 Flink Sql Gateway 进程为例进行说明，其它进程操作类似不再进行说明。
- 若集群中所有节点均已安装 Flink Sql Gateway，添加 Flink Sql Gateway 进程前则需要先在集群中添加主机，然后再执行添加 Flink Sql Gateway 进程的操作。如果集群中已有添加进程所需要的主机，则可直接执行添加 Flink Sql Gateway 进程的操作。

添加 Flink Sql Gateway 进程的操作步骤如下：

(1) 在 Flink 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-3 所示。

a. 选择进程及主机

在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

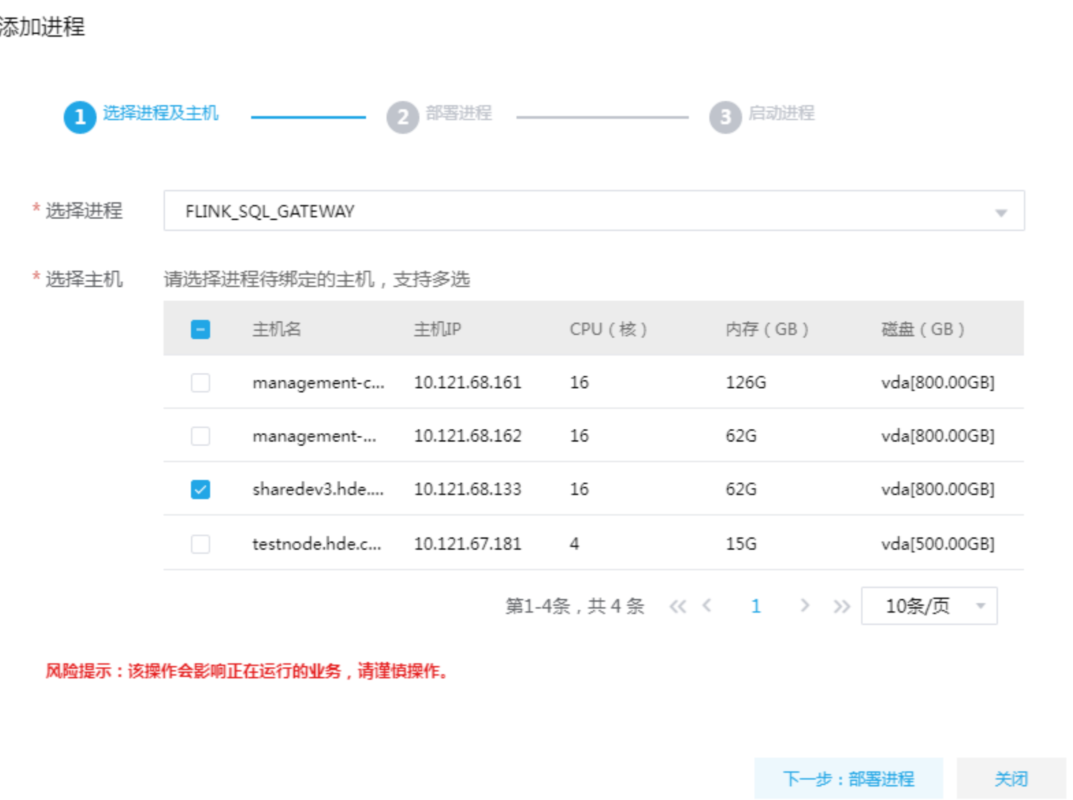
b. 部署进程

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-3 添加进程



(3) 查看进程变化

Flink Sql Gateway 添加完成之后，在组件详情页面[部署拓扑]页签中可以查看 Flink Sql Gateway 进程的数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.3.2 删除进程

Flink 支持删除 Flink Sql Gateway 进程。

- 在并发任务较少的情况下，可以考虑减少 Flink Sql Gateway 进程，节省集群资源。
- 删除进程后，Flink Sql Gateway 对应进程的个数不能少于 1 个。

1. 操作示例

说明

- 删除进程操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行删除 Flink Sql Gateway 进程操作”为例进行说明。
- 执行删除 Flink Sql Gateway 进程操作前，请确认 Flink Sql Gateway 是否有运行的任务，如果有运行的任务时，删除进程会影响任务执行。

删除 Flink Sql Gateway 进程的操作步骤如下：

(1) 在 Flink 组件详情页面[部署拓扑]页签下，选择要删除 Flink Sql Gateway 进程的主机，然后单击该进程右侧操作中的<停止>按钮，停止 Flink Sql Gateway。

(2) 删除 Flink Sql Gateway

待 Flink Sql Gateway 停止成功后，如图 3-4 所示，在该进程右侧操作中单击<删除>按钮，即可完成删除 Flink Sql Gateway。

图3-4 删除 Flink Sql Gateway

(3) 查看进程变化

Flink Sql Gateway 删除完成之后，在组件详情页面[部署拓扑]页签中可以查看 Flink Sql Gateway 进程的数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

4 开发指南

说明

大数据平台中的 Flink 完全兼容 Apache 社区 Flink 1.13.6 版本的 API，关于 Flink 详细接口信息请参见官网。

4.1 开发模型

4.1.1 Flink 抽象层次

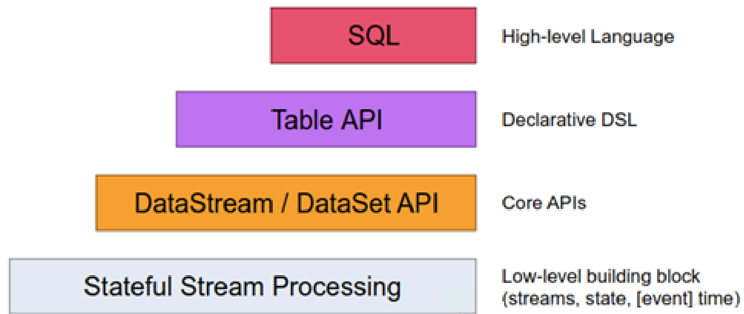
Flink 为流式计算和批处理计算编程提供了不同层次的抽象，如图 4-1 所示，各抽象层次的特点如下：

- Stateful Stream Porcessing
  - 最底层的抽象仅仅提供有状态的流处理，通过处理函数（Process Function）嵌入到 DataStream API 中。
  - 用户可以自行处理来自多个数据流的事件，并使用 Flink 提供的容错机制。
  - 用户也可以自定义事件时间和处理时间回调，以便实现复杂的计算逻辑。

  【说明】：大部分程序一般并不会直接使用 Statefull Stream Processing 层提供的功能，而是直接使用 Flink 提供的 Core API。
- DataStream / DataSet API
  - Flink 提供的 Core API 包括用于处理流数据的 DataStream API 和用于处理批数据的 DataSet API。
  - Core API 提供用户进行流数据处理所需的大部分功能，包括数据的 transform、join、aggregation、windows、状态操作等。
  - 底层的 Process Function 和 DataStream API 的集成，使得 DataStream API 可以在特定算子中使用底层功能。
  - DataSet API 针对有界数据集额外提供了一些原语，比如 loops 和 iterations。
- Table API
  - Table API 是围绕 Table 提供的领域描述语言(DSL)。
  - Table API 遵循关系模型：Table 都有自己的 Schema，API 提供标准的 SQL 操作，包括 select、project、join、groupBy 和聚合。
  - Table API 程序侧重于描述进行何种操作，而不是指定具体的代码实现。
  - Table API 程序在执行之前经过优化程序优化处理，用户可以无缝的在 Table API 和 Core API 间切换，允许程序混用 Table API 和 Core API。
  - 虽然 Table API 可以通过各种用户定义的函数进行扩展，易于用户使用，但却没有 Core API 那么强的表述能力。

- SQL
    - SQL 是 Flink 提供的最高层次的抽象，该层次抽象在语法和表述方面与 Table API 类似。

图4-1 Flink 不同层次抽象接口列表



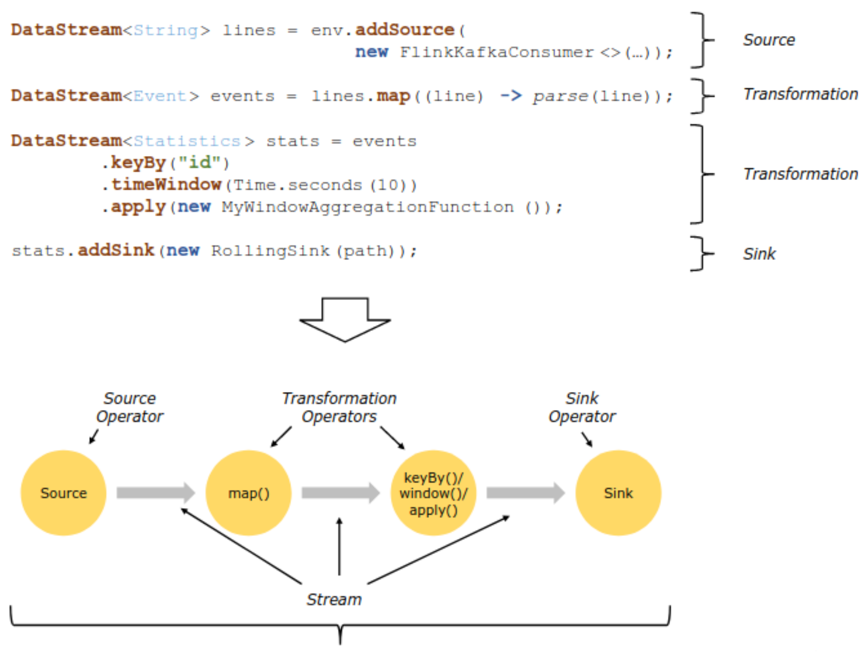
4.1.2 Flink 程序和数据流

Flink 程序的基础构建模块是流（Streams）与转换（Transformations），其中：

- 数据记录的流动形成了 Stream。
- Transformation 则是对数据的操作，接受一个或多个 Stream 作为输入，产生一个或多个输出 Stream 作为结果。

Flink 程序在执行时将被映射成 Streaming DataFlows（由 Streams 和组成 Transformations 的算子构成），每一个 DataFlow 起始于一个或多个 Source，并终止于一个或多个 Sink。一般情况下，Flink 程序中的 Transformation 和 DataFlow 中的算子一一对应，但有时一个 Transformation 也可对应 DataFlow 中的多个算子。

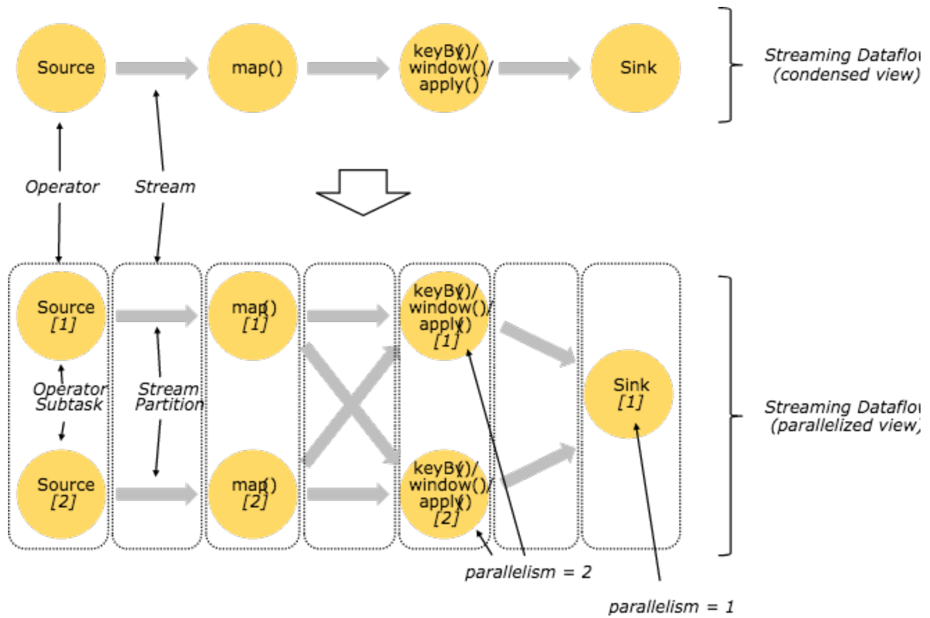
图4-2 Flink 程序运行示意图



流（Streams）在两个算子之间传输数据，传输模式有 forwarding 模式和 redistributing 模式，其中：

- forwarding 模式（即一对一模式），如图 4-3 所示的 Source 与 map()之间，map()算子的子任务将与 Source 的子任务生成相同的顺序查看到相同的元素。
- redistributing 模式，如图 4-3 所示的 map()与 keyBy/window 之间和 keyBy/window 与 Sink 之间，则改变了流的分区，每一个算子子任务根据所选择的转换，向不同的目标子任务发送数据。在一次 redistributing 交换中，元素间的排序只保留在每对发送与接受子任务中。

图4-3 流在两个算子之间传输数据的模式



4.1.3 Flink 程序并行度

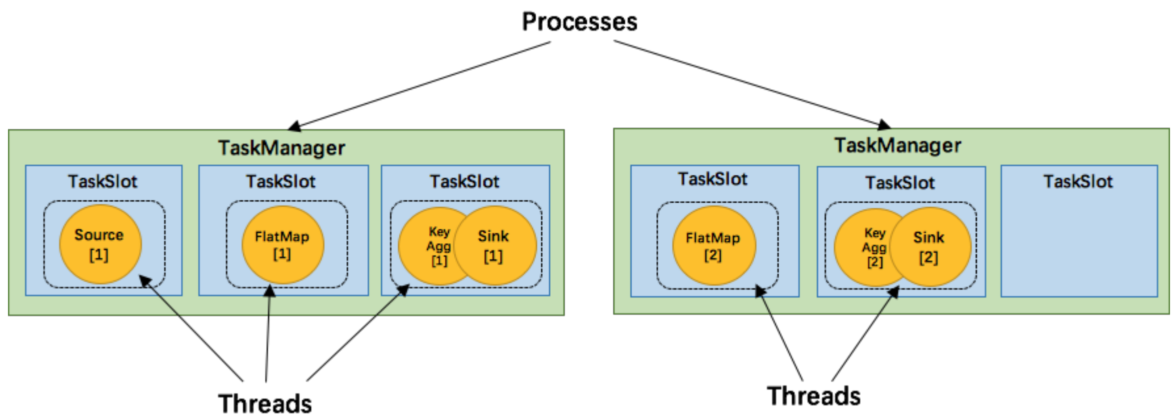
Flink 程序支持并行和分布式计算。在程序执行时，流（Stream）对应多个流分区（Stream partition），每个算子对应多个算子子任务（Subtask）。算子的各个子任务之间相互独立，子任务可分配到不同的线程或容器中执行。

Stream 的并行度指算子的并行度，算子的并行度指子任务的数量。同一个 Flink 程序中各个算子的并行度可以不一样。

Flink 的每个 TaskManager 即是一个 JVM 进程，并且可以在不同线程中执行一个或多个 Subtask。TaskManager 通过 Task Solts（任务槽位）控制可以接收的 Tasks 数量，每个 TaskManager 至少有一个任务槽位。

每个 Task Solt 代表 TaskManager 中一个固定的资源子集。例如：TaskManage 中有 3 个 Task Solts，则表示此 TaskManage 的内存资源被划分为了 3 份，划分资源意味着此槽位内的 Subtasks 不与其他槽位的 Subtasks 竞争资源，但同时也意味着此槽位内的 Subtasks 只拥有固定的内存资源。另外，通过槽位划分资源时并不进行 CPU 隔离，仅划分内存资源给不同的 Subtasks，通过调整 Task Solts 个数可以调整 Subtasks 之间的隔离方式。同一个 JVM 中的 Subtasks 共享 TCP 连接（通过多路复用技术）和心跳消息，同时可能还会共享数据集和数据结构，从而减少每个 Subtask 的开销。

图4-4 Task Solt 内的 Subtasks 分布示例



默认情况下，当 Subtasks 是来自同一个 Job 时，Flink 允许不同算子的 Subtasks 共享 Task Solts。允许 Task Solt sharing 有以下好处：

- Flink 集群需要的 Task Solts 的数量和 Job 的最高并发度相同，所以不需要计算一个 Job 总共包含多少个算子（具有不同并行度）。
- Task Solt sharing 可提升资源利用率。

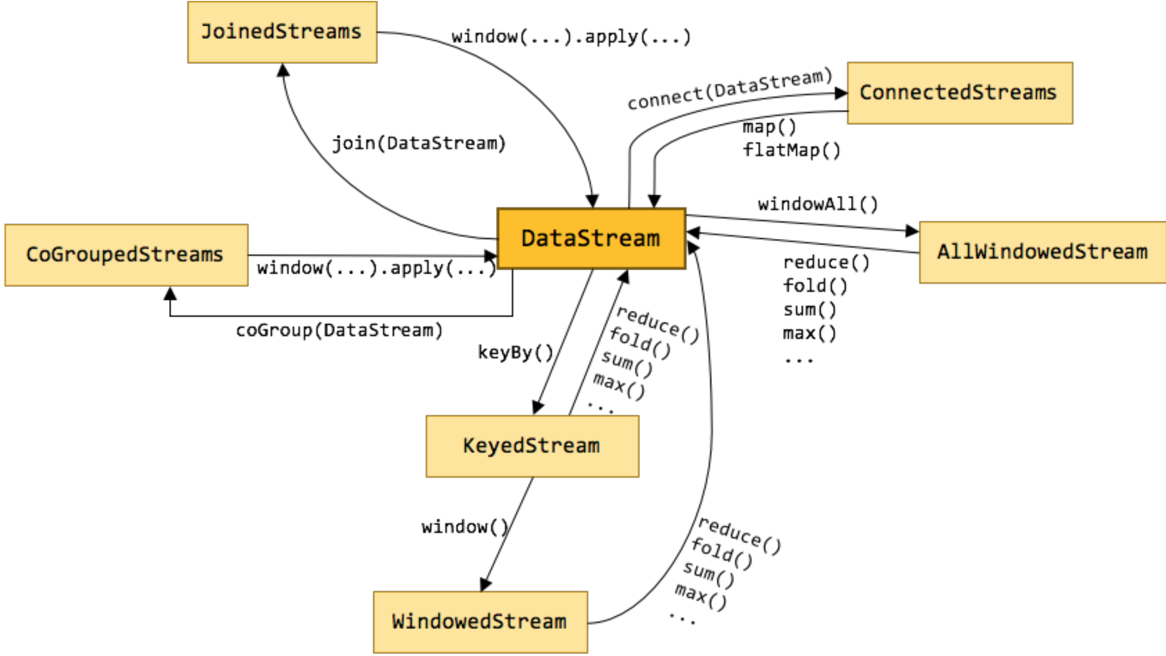
4.2 常用API

Flink 主要使用到如下这几个类：

- StreamExecutionEnvironment：是 Flink 流处理的基础，提供了程序的执行环境。
- DataStream：类 DataStream 来表示程序中的流式数据。流式数据是含有重复数据且不可修改的集合(collection)，DataStream 中元素的数量是无限的。
- KeyedStream：DataStream 通过 keyBy 分组操作生成流，通过设置的 key 值对数据进行分组。
- WindowedStream：KeyedStream 通过 window 窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- AllWindowedStream：DataStream 通过 window 窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- ConnectedStreams：将两条 DataStream 流连接起来并且保持原有流数据的类型，然后进行 map 或者 flatMap 操作。
- JoinedStreams：在窗口上对数据进行等值 join 操作（等值就是判断两个值相同的 join，比如 a.id = b.id），join 操作是 coGroup 操作的一种特殊场景。
- CoGroupedStreams：在窗口上对数据进行 coGroup 操作，可以实现流的各种 join 类型。

图 4-5 展示了 Flink 中目前支持的主要几种流的类型，以及它们之间的转换关系。

图4-5 Flink 支持的部分流类型及其之间的转换关系图



4.2.1 Java API

1. 流数据输入

表4-1 流数据输入 API 说明

<table>
<tr><th>API</th><th>说明</th></tr>
<tr><td>public final <X> DataSource<X> fromElements(X... data)</td><td rowspan="2">获取用户定义的多个元素的数据，作为输入流数据，其中：
• type 是指元素的数据类型
• data 是指多个元素的具体数据</td></tr>
<tr><td>public final <X> DataSource<X> fromElements(Class<X> type, X... data)</td></tr>
<tr><td>public <X> DataSource<X> fromCollection(Collection<X> data)</td><td rowspan="5">获取用户定义的集合数据，作为输入流数据，其中：
• type 是指集合中元素的数据类型
• typeInfo 是指集合中根据元素数据类型获取的类型信息
• data 是指集合数据或者可迭代的数据体</td></tr>
<tr><td>public <X> DataSource<X> fromCollection(Collection<X> data, TypeInformation<X> type)</td></tr>
<tr><td>public <X> DataSource<X> fromCollection(Iterator<X> data, TypeInformation<X> type)</td></tr>
<tr><td>public <X> DataSource<X> fromCollection(Collection<X> data, TypeInformation<X> type)</td></tr>
<tr><td>public <X> DataSource<X> fromCollection(Iterator<X> data, Class<X> type)</td></tr>
<tr><td>public <X> DataSource<X> fromParallelCollection(SplittableIterator<X> iterator, Class<X> type)</td><td rowspan="2">获取用户定义的集合数据，作为输入并行流数据，其中：
• type 是指集合中元素的数据类型
• typeInfo 是指集合中根据元素数据类型获取的类型信息</td></tr>
<tr><td>public <X> DataSource<X> fromParallelCollection(SplittableIterator<X> iterator, TypeInformation<X> type)</td></tr>
</table>

API	说明
private <X> DataSource<X> fromParallelCollection(SplittableIterator<X> iterator, TypeInformation<X> type, String callLocationName)	• iterator 是指可被分割成多个 partition 的迭代数据体
public DataSource<Long> generateSequence(long from, long to)	获取用户定义的一串序列数据，作为输入流数据，其中： • from 是指数值串的起点 • to 是指数值串的终点
public DataSource<String> readTextFile(String filePath)	获取用户定义的某路径下的文本文件数据，作为输入流数据，其中： • filePath 是指文本文件的路径 • charsetName 是指编码格式的名字
public DataSource<String> readTextFile(String filePath, String charsetName)	
public <X> DataSource<X> readFile(FileInputFormat<X> inputFormat, String filePath)	获取用户定义的某路径下的文件数据，作为输入流数据，其中： • filePath 是指文件的路径 • inputFormat 是指文件的格式 • watchType 是指文件的处理模式，包括 “PROCESS_ONCE”或者 “PROCESS_CONTINUOUSLY” • interval 是指经过多长时间判断目录或文件变化进行处理
public <OUT> DataStreamSource<OUT> readFile(FileInputFormat<OUT> inputFormat,String filePath,FileProcessingMode watchType,long interval,TypeInformation<OUT> typeInformation)	
public DataStreamSource<String> socketTextStream(String hostname, int port, String delimiter, long maxRetry)	获取用户定义的Socket数据，作为输入流数据，其中： • hostname 是指 Socket 的服务器端的主机名称 • port 是指服务器的监听端口 • delimiter 是指消息之间的分隔符 • maxRetry 是指由于连接异常可以触发的最大重试次数
public DataStreamSource<String> socketTextStream(String hostname, int port, String delimiter)	
public DataStreamSource<String> socketTextStream(String hostname, int port)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function)	用户自定义SourceFunction，addSource方法可以添加Kafka等数据源，主要实现方法为SourceFunction的run，其中： • function 是指用户自定义的 SourceFunction 函数 • sourceName 是指定义该数据源的名称 • typeInfo 是指根据元素数据类型获取的类型信息
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, String sourceName)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, TypeInformation<OUT> typeInfo)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, String sourceName, TypeInformation<OUT> typeInfo)	

2. 流数据输出

表4-2 流数据输出 API 说明

API	说明
public DataStreamSink<T> print()	数据输出以标准输出流打印出来
public DataStreamSink<T> printToErr(String sinkIdentifier)	数据输出以标准error输出流打印出来
public DataStreamSink<T> writeToSocket(String hostName, int port, SerializationSchema<T> schema)	数据输出写入到Socket连接中，其中： • hostname 是指 Socket 的服务器端的主机名称 • port 是指服务器的监听端口
public DataStreamSink<T> addSink(SinkFunction<T> sinkFunction)	用户自定义的数据输出，addSink方法可以添加kafka等数据输出，主要实现方法为SinkFunction的invoke方法

3. 过滤和映射能力

表4-3 过滤和映射能力 API 说明

API	说明
public <R> SingleOutputStreamOperator<R> map(MapFunction<T, R> mapper)	输入一个元素，生成另一个元素，元素类型不变
public <R> SingleOutputStreamOperator<R> flatMap(FlatMapFunction<T, R> flatMapper)	输入一个元素，生成零个、一个或者多个元素
public SingleOutputStreamOperator<T> filter(FilterFunction<T> filter)	对每个元素执行一个布尔函数，只保留返回true的元素

4. 聚合能力

表4-4 聚合能力 API 说明

API	说明
public KeyedStream<T, Tuple> keyBy(int... fields)	将流逻辑分区成不相交的分区，每个分区包含相同key的元素，其内部通过hash分区实现。这个转换返回了一个KeyedStream。KeyBy操作之后返回KeyedStream，然后再调用KeyedStream的函数（例如reduce/fold/min/minby/max/maxby/sum/sumby等）进行相应操作。其中： • Fields 是指数据某几列的序号或者成员变量的名称 • Key 是指用户自定义的指定分区依据的方法
public KeyedStream<T, Tuple>keyBy(String... fields)	
private KeyedStream<T, Tuple> keyBy(Keys<T> keys)	
public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key)	
public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key, TypeInformation<K> keyType)	
public SingleOutputStreamOperator<T> reduce(ReduceFunction<T> function)	在一个KeyedStream上“滚动”reduce，即合并当前元素与上一个被reduce的值，然后输出新的值
public SingleOutputStreamOperator<T> sum(int positionToSum)	在一个KeyedStream上滚动求和操作 其中：positionToSum和field表示对某一列求和
public SingleOutputStreamOperator<T> sum(String field)	
public SingleOutputStreamOperator<T> min(int	在一个KeyedStream上滚动求最小值。min返回了最

API	说明
positionToMin)	小值，但不保证非最小值列的准确性。 其中：positionToMin和field表示对某一列求最小值
public SingleOutputStreamOperator<T> min(String field)	
public SingleOutputStreamOperator<T> max(int positionToMax)	在一个KeyedStream上滚动求最大值。max返回了最大值，但不保证非最大值列的准确性。 其中：positionToMax和field表示对某一列求最大值
public SingleOutputStreamOperator<T> max(String field)	
public SingleOutputStreamOperator<T> minBy(String field, boolean first)	在一个KeyedStream上求某一列最小值所在的该行数据，minBy返回了该行数据的所有元素，其中： • positionToMinBy 表示对哪一列做 minBy 操作 • first 为 true 表示按最先遇到的最小值输出，first 为 false 表示按最后遇到的最小值输出
public SingleOutputStreamOperator<T> minBy(String positionToMinBy)	
public SingleOutputStreamOperator<T> minBy(String field)	
public SingleOutputStreamOperator<T> minBy(int positionToMinBy, boolean first)	
public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy)	在一个KeyedStream上求某一列最大值所在的该行数据，maxBy返回了该行数据的所有元素，其中： • positionToMaxBy 表示对哪一列做 maxBy 操作 • first 为 true 表示按最先遇到的最小值输出，first 为 false 表示按最后遇到的最小值输出
public SingleOutputStreamOperator<T> maxBy(String field)	
public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy, boolean first)	
public SingleOutputStreamOperator<T> maxBy(String field, boolean first)	

5. 数据流分发能力

表4-5 数据流分发能力 API 说明

API	说明
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, int field)	使用一个用户自定义的Partitioner对每一个元素选择目标task，其中： • Partitioner 是指用户自定义的分区类重写 partition 方法 • Field 是指 partitioner 的输入参数 • keySelector 是指用户自定义的 partitioner 的输入参数
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, String field)	
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, KeySelector<T, K> keySelector)	
private <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, Keys<T> keys)	
public DataStream<T> shuffle()	以均匀分布的形式将元素随机地进行分区
public DataStream<T> rebalance()	基于round-robin对元素进行分区，使得每个分区负责均衡
public DataStream<T> rescale()	以round-robin的形式将元素分区到下游操作的子集中
public DataStream<T> broadcast()	广播每个元素到所有分区
public BroadcastStream<T> broadcast(final	

API	说明
MapStateDescriptor<?, ?>... broadcastStateDescriptors)	

6. 设置 eventtime 属性的能力

表4-6 设置 eventtime 的 API 说明

API	说明
public SingleOutputStreamOperator<T> assignTimestampsAndWatermarks(AssignerWithPeriodicWatermarks<T> timestampAndWatermarkAssigner)	为了能让event time窗口可以正常触发窗口计算操作，需要从记录中提取时间戳
public SingleOutputStreamOperator<T> assignTimestampsAndWatermarks(AssignerWithPunctuatedWatermarks<T> timestampAndWatermarkAssigner)	

7. 窗口能力

窗口分为跳跃窗口和滑动窗口，功能如下：

- 支持 Window、TimeWindow、CountWindow 以及 WindowAll、TimeWindowAll、CountWindowAll API 窗口生成。
- 支持 Window Apply、Window Reduce、Window Fold、Aggregations on windows API 窗口操作。
- 支持多种 Window Assigner，比如：TumblingEventTimeWindows、TumblingProcessingTimeWindows、SlidingEventTimeWindows、SlidingProcessingTimeWindows、EventTimeSessionWindows、ProcessingTimeSessionWindows、GlobalWindows。
- 支持三种时间 ProcessingTime、EventTime 和 IngestionTime。
- 支持两种 EventTime 时间戳方式：AssignerWithPeriodicWatermarks 和 AssignerWithPunctuatedWatermarks。

表4-7 窗口生成类相关 API 说明

API	说明
public <W extends Window> WindowedStream<T, KEY, W> window(WindowAssigner<? super T, W> assigner)	窗口可以被定义在已经被分区的KeyedStreams上。窗口会对数据的每一个key根据一些特征（例如：在最近5秒中内到达的数据）进行分组
public <W extends Window> AllWindowedStream<T, W> windowAll(WindowAssigner<? super T, W> assigner)	窗口可以被定义在DataStream上
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size)	时间窗口定义在已经被分区的KeyedStreams上，根据environment.getStreamTimeCharacteristic()参数可以选择是ProcessingTime还是EventTime，根据参数个数可以确定是TumblingWindow还是SlidingWindow，其中： • Size 是指窗口时间的大小 • slide 是指窗口的滑动时间 说明：WindowedStream和AllWindowedStream代表不
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size, Time slide)	

API	说明
	同的两种流。接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow
public AllWindowedStream<T, GlobalWindow> countWindowAll(long size)	时间窗口定义在DataStream上，根据environment.getStreamTimeCharacteristic()参数可以选择是ProcessingTime还是EventTime，根据参数个数可以确定是TumblingWindow还是SlidingWindow，其中： • Size 是指窗口时间的大小 • Slide 是指窗口的滑动时间
public AllWindowedStream<T, GlobalWindow> countWindowAll(long size, long slide)	
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size)	按照元素个数区分窗口，定义在已经被分区的KeyedStreams上，其中： • Size 是指窗口时间的大小 • Slide 是指窗口的滑动时间 说明：WindowedStream和AllWindowedStream代表不同的两种流。接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size, Time slide)	

8. 多流合并能力

表4-8 多流合并能力 API 说明

API	说明
public final DataStream<T> union(DataStream<T>... streams)	Union两个或多个数据流，生成一个新的包含了来自所有流的所有数据的数据流 说明：如果一个数据流与其自身进行了合并，则在结果流中对于每个元素都将拿到两份
public <R> ConnectedStreams<T, R> connect(DataStream<R> dataStream)	“连接”两个数据流并保持原先的类型。Connect可以让两条流之间共享状态，产生ConnectedStreams之后，调用map或者flatmap进行操作计算
public <R> BroadcastConnectedStream<T, R> connect(BroadcastStream<R> broadcastStream)	
public <R> SingleOutputStreamOperator<R> map(CoMapFunction<IN1, IN2, R> coMapper)	在ConnectedStreams上做元素映射，类似DataStream的map操作，元素映射之后流数据类型统一
public <R> SingleOutputStreamOperator<R> flatMap(CoFlatMapFunction<IN1, IN2, R> coFlatMapper)	在ConnectedStreams上做元素映射，类似DataStream的flatMap操作，元素映射之后流数据类型统一

9. Join 能力

表4-9 Join 能力 API 说明

API	说明
public <T2> JoinedStreams<T, T2> join(DataStream<T2> otherStream)	通过给定的key在一个窗口范围内join两条数据流
public <T2> CoGroupedStreams<T, T2>	通过给定的key在一个窗口范围内co-group两条数据流

API	说明
coGroup(DataStream<T2> otherStream)	

4.2.2 Scala API

1. 流数据输入

表4-10 流数据输入 API 说明

<table>
<tr><th>API</th><th>说明</th></tr>
<tr><td>def fromElements[T: ClassTag : TypeInformation](data: T*): DataSet[T]</td><td>获取用户定义的多个元素的数据，作为输入流数据，其中：
• data 是指多个元素的具体数据</td></tr>
<tr><td>def fromCollection[T: ClassTag : TypeInformation](data: Iterable[T]): DataSet[T]</td><td rowspan="2">获取用户定义的集合数据，作为输入流数据，其中：
• data 是指集合数据或者可迭代的数据体</td></tr>
<tr><td>def fromCollection[T: ClassTag : TypeInformation](data: Iterator[T]): DataSet[T]</td></tr>
<tr><td>def fromParallelCollection[T: TypeInformation] (data: SplittableIterator[T]): DataStream[T]</td><td>获取用户定义的集合数据，作为输入并行流数据，其中：
• data 是指可被分割成多个 partition 的迭代数据体</td></tr>
<tr><td>def generateSequence(from: Long, to: Long): DataSet[Long]</td><td>获取用户定义的一串序列数据，作为输入流数据，其中：
• from 是指数值串的起点
• to 是指数值串的终点</td></tr>
<tr><td>def readTextFile(filePath: String): DataStream[String]</td><td rowspan="2">获取用户定义的某路径下的文本文件数据，作为输入流数据，其中：
• filePath 是指文本文件的路径
• charsetName 是指编码格式的名字</td></tr>
<tr><td>def readTextFile(filePath: String, charsetName: String): DataStream[String]</td></tr>
<tr><td>def readFile[T: TypeInformation](inputFormat: FileInputFormat[T], filePath: String): DataStream[T]</td><td rowspan="2">获取用户定义的某路径下的文件数据，作为输入流数据，其中：
• filePath 是指文件的路径
• inputFormat 是指文件的格式
• watchType 是指文件的处理模式，包括“PROCESS_ONCE”或者“PROCESS_CONTINUOUSLY”
• interval 是指经过多长时间判断目录或文件变化进行处理</td></tr>
<tr><td>def readFile[T: TypeInformation](inputFormat: FileInputFormat[T],filePath: String,watchType: FileProcessingMode,interval: Long): DataStream[T]</td></tr>
<tr><td>def socketTextStream(hostname: String, port: Int, delimiter: Char = '\n', maxRetry: Long = 0):DataStream[String]</td><td>获取用户定义的Socket数据，作为输入流数据，其中：
• hostname 是指 Socket 的服务器端的主机名称
• port 是指服务器的监听端口
• delimiter 和 maxRetry 两个参数 scala 接口暂时不支持设置</td></tr>
<tr><td>def addSource[T: TypeInformation](function: SourceFunction[T]): DataStream[T]</td><td rowspan="2">用户自定义SourceFunction，addSource方法可以添加Kafka等数据源，主要实现方法为SourceFunction的run，其中：</td></tr>
<tr><td>def addSource[T: TypeInformation](function:</td></tr>
</table>

API	说明
SourceContext[T] => Unit): DataStream[T]	• function 是指用户自定义的 SourceFunction 函数 说明：Scala支持简化写法

2. 流数据输出

表4-11 流数据输出 API 说明

API	说明
def print(): DataStreamSink[T]	数据输出以标准输出流打印出来
def print(sinkIdentifier: String): DataStreamSink[T]	
def printToErr()	数据输出以标准error输出流打印出来
def printToErr(sinkIdentifier: String)	
def writeAsText(path: String): DataStreamSink[T]	数据输出写入到某个文本文件中，其中： • Path 是指文本文件的路径 • writeMode 是指文本文件写入模式，包括 “OVERWRITE”或者“NO_OVERWRITE”
def writeAsText(path: String, writeMode: FileSystem.WriteMode): DataStreamSink[T	
def writeAsCsv(path: String, writeMode: FileSystem.WriteMode): DataStreamSink[T]	数据输出写入到某个csv格式的文件中，其中： • path 是指文本文件的路径 • writeMode 是指文本文件写入模式，包括 “OVERWRITE”或者“NO_OVERWRITE” • rowDelimiter 是指行分隔符 • fieldDelimiter 是指列分隔符
def writeAsCsv(path: String, writeMode: FileSystem.WriteMode): DataStreamSink[T]	
def writeAsCsv(path: String,writeMode: FileSystem.WriteMode,rowDelimiter: String,fieldDelimiter: String):DataStreamSink[T]	
def writeUsingOutputFormat(format: OutputFormat[T]): DataStreamSink[T]	数据输出到普通文件中，例如二进制文件
def writeToSocket(hostname: String,port: Integer,schema: SerializationSchema[T]): DataStreamSink[T]	数据输出写入到Socket连接中，其中： • hostname 是指 Socket 的服务器端的主机名称 • port 是指服务器的监听端口
def addSink(sinkFunction: SinkFunction[T]): DataStreamSink[T]	用户自定义的数据输出，addSink方法通过 flink-connectors支持数据输出到Kafka，主要实现方法为SinkFunction的invoke方法
def addSink(fun: T => Unit): DataStreamSink[T]	

3. 过滤和映射能力

表4-12 过滤和映射能力 API 说明

API	说明
def map[R: TypeInformation](mapper: MapFunction[T, R]): DataStream[R]	输入一个元素，生成另一个元素，元素类型不变
def map[R: TypeInformation](fun: T => R): DataStream[R]	
def flatMap[R: TypeInformation](fun: (T, Collector[R]) => Unit): DataStream[R]	输入一个元素，生成零个、一个或者多个元素

API	说明
def flatMap[R: TypeInformation](flatMapper: FlatMapFunction[T, R]): DataStream[R]	
def flatMap[R: TypeInformation](fun: T => TraversableOnce[R]): DataStream[R]	
def filter(filter: FilterFunction[T]): DataStream[T]	对每个元素执行一个布尔函数，只保留返回true的元素
def filter(fun: T => Boolean): DataStream[T]	

4. 聚合能力

表4-13 聚合能力 API 说明

API	说明
def keyBy(fields: Int*): KeyedStream[T, JavaTuple]	将流逻辑分区成不相交的分区，每个分区包含相同key的元素，其内部通过hash分区实现。这个转换返回了一个KeyedStream。KeyBy操作之后返回KeyedStream，然后再调用KeyedStream的函数（例如reduce/fold/min/minby/max/maxby/sum/sumby等）进行相应操作。其中： • Fields 是指数据某几列的序号 • firstField 和 otherFields 是指数据结构的成员变量的名称 • key 是指用户自定义的指定分区依据的方法
def keyBy(firstField: String, otherFields: String*): KeyedStream[T, JavaTuple]	
def keyBy[K: TypeInformation](fun: T => K): KeyedStream[T, K]	
def keyBy[K: TypeInformation](fun: KeySelector[T, K]): KeyedStream[T, K]	
def reduce(fun: (T, T) => T): DataStream[T]	在一个KeyedStream上“滚动”reduce，即合并当前元素与上一个被reduce的值，然后输出新的值 注意：三者的类型是一致的
def reduce(reducer: ReduceFunction[T]): DataStream[T]	
def sum(position: Int): DataStream[T]	在一个KeyedStream上滚动求和操作 其中：positionToSum和field表示对某一列求和
def sum(field: String): DataStream[T]	
def min(position: Int): DataStream[T]	在一个KeyedStream上滚动求最小值。min返回了最小值，但不保证非最小值列的准确性。 其中：positionToMin和field表示对某一列求最小值
def min(field: String): DataStream[T]	
def max(position: Int): DataStream[T]	在一个KeyedStream上滚动求最大值。max返回了最大值，但不保证非最大值列的准确性。 其中：positionToMax和field表示对某一列求最大值
def max(field: String): DataStream[T]	
def minBy(position: Int): DataStream[T]	在一个KeyedStream上求某一列最小值所在的该行数据，minBy返回了该行数据的所有元素，其中： • positionToMinBy 表示对哪一列做 minBy 操作
def minBy(field: String): DataStream[T]	
def maxBy(position: Int): DataStream[T]	在一个KeyedStream上求某一列最大值所在的该行数据，maxBy返回了该行数据的所有元素，其中： • positionToMaxBy 表示对哪一列做 maxBy 操作
def maxBy(field: String): DataStream[T]	

5. 数据流分发能力

表4-14 数据流分发能力 API 说明

<table>
<tr><th>API</th><th>说明</th></tr>
<tr><td>def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], field: Int) : DataStream[T]</td><td rowspan="3">使用一个用户自定义的Partitioner对每一个元素选择目标task，其中：
• Partitioner 是指用户自定义的分区类重写 partition 方法
• Field 是指 partitioner 的输入参数
• keySelector 是指用户自定义的 partitioner 的输入参数</td></tr>
<tr><td>def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], field: String) : DataStream[T]</td></tr>
<tr><td>def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], fun: T => K): DataStream[T]</td></tr>
<tr><td>def shuffle: DataStream[T]</td><td>以均匀分布的形式将元素随机地进行分区</td></tr>
<tr><td>def rebalance: DataStream[T]</td><td>基于round-robin对元素进行分区，使得每个分区负责均衡</td></tr>
<tr><td>def rescale: DataStream[T]</td><td>以round-robin的形式将元素分区到下游操作的子集中</td></tr>
<tr><td>def broadcast: DataStream[T]</td><td rowspan="2">广播每个元素到所有分区</td></tr>
<tr><td>def broadcast(broadcastStateDescriptors: MapStateDescriptor[_, _]*): BroadcastStream[T]</td></tr>
</table>

6. 设置 eventtime 属性的能力

表4-15 设置 eventtime 的 API 说明

<table>
<tr><th>API</th><th>说明</th></tr>
<tr><td>def assignTimestampsAndWatermarks(assigner: AssignerWithPeriodicWatermarks[T]): DataStream[T]</td><td rowspan="2">为了能让event time窗口可以正常触发窗口计算操作，需要从记录中提取时间戳</td></tr>
<tr><td>def assignTimestampsAndWatermarks(assigner: AssignerWithPunctuatedWatermarks[T]): DataStream[T]</td></tr>
</table>

7. 窗口能力

窗口分为跳跃窗口和滑动窗口，功能如下：

- 支持 Window、TimeWindow、CountWindow 以及 WindowAll、TimeWindowAll、CountWindowAll API 窗口生成。
- 支持 Window Apply、Window Reduce、Window Fold、Aggregations on windows API 窗口操作。
- 支持多种 Window Assigner，比如：TumblingEventTimeWindows、TumblingProcessingTimeWindows、SlidingEventTimeWindows、SlidingProcessingTimeWindows、EventTimeSessionWindows、ProcessingTimeSessionWindows、GlobalWindows。
- 支持三种时间 ProcessingTime、EventTime 和 IngestionTime。
- 支持两种 EventTime 时间戳方式：AssignerWithPeriodicWatermarks 和 AssignerWithPunctuatedWatermarks。

表4-16 窗口生成类相关 API 说明

API	说明
def window[W <: Window](assigner: WindowAssigner[_ >: T, W]): WindowedStream[T, K, W]	窗口可以被定义在已经被分区的KeyedStreams上。窗口会对数据的每一个key根据一些特征（例如：在最近5秒中内到达的数据）进行分组
def windowAll[W <: Window](assigner: WindowAssigner[_ >: T, W]): AllWindowedStream[T, W]	窗口可以被定义在DataStream上
def timeWindow(size: Time): WindowedStream[T, K, TimeWindow]	时间窗口定义在已经被分区的KeyedStreams上，根据environment.getStreamTimeCharacteristic()参数可以选择是ProcessingTime还是EventTime，根据参数个数可以确定是TumblingWindow还是SlidingWindow，其中： • Size 是指窗口时间的大小 • Slide 是指窗口的滑动时间 说明：WindowedStream和AllWindowedStream代表不同的两种流。接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow
def timeWindow(size: Time, slide: Time): WindowedStream[T, K, TimeWindow]	
def countWindow(size: Long, slide: Long): WindowedStream[T, K, GlobalWindow]	
def timeWindowAll(size: Time): AllWindowedStream[T, TimeWindow]	时间窗口定义在DataStream上，根据environment.getStreamTimeCharacteristic()参数可以选择是ProcessingTime还是EventTime，根据参数个数可以确定是TumblingWindow还是SlidingWindow，其中： • Size 是指窗口时间的大小 • Slide 是指窗口的滑动时间
def timeWindowAll(size: Time, slide: Time): AllWindowedStream[T, TimeWindow]	
def countWindow(size: Long, slide: Long): WindowedStream[T, K, GlobalWindow]	按照元素个数区分窗口，定义在已经被分区的KeyedStreams上，其中： • Size 是指窗口时间的大小 • Slide 是指窗口的滑动时间 说明：WindowedStream和AllWindowedStream代表不同的两种流。接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow
def countWindow(size: Long): WindowedStream[T, K, GlobalWindow]	
def countWindowAll(size: Long, slide: Long): AllWindowedStream[T, GlobalWindow]	按照元素个数区分窗口，定义在DataStream上，其中： • Size 是指窗口时间的大小 • Slide 是指窗口的滑动时间
def countWindowAll(size: Long): AllWindowedStream[T, GlobalWindow]	

8. 多流合并能力

表4-17 多流合并能力 API 说明

API	说明
def union(dataStreams: DataStream[T]*): DataStream[T]	Union两个或多个数据流，生成一个新的包含了来自所有流的所有数据的数据流 说明：如果一个数据流与其自身进行了合并，则在结果流中对于每个元素都将拿到两份

API	说明
def connect[T2](dataStream: DataStream[T2]): ConnectedStreams[T, T2]	“连接”两个数据流并保持原先的类型。Connect可以让两条流之间共享状态，产生ConnectedStreams之后，调用map或者flatmap进行操作计算
def map[R: TypeInformation](coMapper: CoMapFunction[IN1, IN2, R]): DataStream[R]	在ConnectedStreams上做元素映射，类似DataStream的map操作，元素映射之后流数据类型统一
def map[R: TypeInformation](fun1: IN1 => R, fun2: IN2 => R): DataStream[R]	
def flatMap[R: TypeInformation](fun1: (IN1, Collector[R]) => Unit, fun2: (IN2, Collector[R]) => Unit): DataStream[R]	在ConnectedStreams上做元素映射，类似DataStream的flatMap操作，元素映射之后流数据类型统一
def flatMap[R: TypeInformation](coFlatMapper: CoFlatMapFunction[IN1, IN2, R]): DataStream[R]	
def flatMap[R: TypeInformation](fun1: IN1 => TraversableOnce[R],fun2: IN2 => TraversableOnce[R]): DataStream[R]	

9. Join 能力

表4-18 Join 能力 API 说明

API	说明
def join[T2](otherStream: DataStream[T2]): JoinedStreams[T, T2]	通过给定的key在一个窗口范围内join两条数据流，其中：join操作的key值通过where和eaualTo方法进行指定，代表两条流过滤出包含等值条件的数据
def coGroup[T2](otherStream: DataStream[T2]): CoGroupedStreams[T, T2]	通过给定的key在一个窗口范围内co-group两条数据流，其中：coGroup操作的key值通过where和eaualTo方法进行指定，代表两条流通过该等值条件进行分区处理

4.3 Flink on YARN任务提交模式

说明

- 考虑到资源隔离性，对于生产环境，推荐使用 Application 模式或者 Per-Job Cluster 模式。注意：与 Per-Job Cluster 模式相比，Application 模式 Flink 程序 main()方法运行在 JobManager，而非 Client 端。相同代码并不一定同时支持 Application 模式和 Per-Job Cluster 模式（比如：涉及读取 Client 本地文件的程序，并不支持运行在 Application 模式）。
- 本章节的命令示例均需执行 cd /usr/hdp/3.0.1.0-187/flink 命令进入 Flink 安装路径后执行。

4.3.1 Application Mode（推荐）

Application 模式为生产环境推荐模式，其实质是在 YARN 上启动一个 Flink 集群，用于运行单独的 job。该模式下 Flink 集群与 job 是一一对应的关系，即当 job 运行结束时对应的 Flink 集群也停止。

Application 模式下，Flink 程序 main()方法运行在 JobManager，以节省 Client 端资源开销。

示例：./bin/flink run-application –t yarn-application ./examples/batch/WordCount.jar

其中：run-application –t yarn-application 表示指定任务以 Application 模式提交。

4.3.2 Per-Job Cluster Mode（推荐）

Application 模式为生产环境推荐模式，其实质是在 YARN 上启动一个 Flink 集群，用于运行单独的 job。该模式下 Flink 集群与 job 是一一对应的关系，即当 job 运行结束时对应的 Flink 集群也停止。

示例：./bin/flink run –t yarn-per-job ./examples/batch/WordCount.jar

其中：-t yarn-per-job 表示指定任务以 Per-Job 模式提交。

4.3.3 Session Mode（可选）

yarn-session 模式的实质是提前在 YARN 上启动一个常驻的 Flink 集群。

示例：./bin/yarn-session.sh -n 2 -d

待常驻集群启动成功之后，会打印提示信息“The Flink YARN client has been started in detached mode. In order to stop Flink on YARN, use the following command or a YARN web interface to stop it:yarn application -kill <application_1540989860694_0006>”。此时即可向该常驻集群提交一个或多个 Flink 任务。

示例：./bin/flink run -yid <application_1540989860694_0006> ./examples/batch/WordCount.jar

其中：如果提交任务节点与启动常驻集群节点相同，可不指定-yid 参数。由于 Flink 组件包默认开启基于 Zookeeper 的 HA 功能，不支持-m 参数指定 jobmanager ip:port 方式提交任务。

5 最佳实践

5.1 Flink开发程序（非Kerberos环境）

说明

- Flink 脚本所在路径/usr/hdp/3.0.1.0-187/flink/bin（即在大数据平台安装的默认路径）。
- Kafka 脚本所在路径/usr/hdp/3.0.1.0-187/kafka/bin（即在大数据平台安装的默认路径）。
- 本章节的最佳实践仅为示例（比如：文件名、文件路径、表名称、主题名称、集群 IP 等），在使用过程中本章节的命令和代码仅供参考，请根据实际情况进行调整。

5.1.1 Flink DataStream 读 Kafka 写 Kafka

1. 样例主要功能

从 Kafka 一个 topic 中读取数据进行 wordcount 之后，将结果写入到一个新的 topic 中。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 执行 Flink 作业

```
flink run -t yarn-per-job -d ${jar path }/KafkaFlink1.13.6Kafka-1.0.jar --path FlinkJob_Kafka2Kafka.properties
```

其中：

- ${jar path}表示样例 jar 包路径。

(3) 生产消息

```
./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name}
```

消息格式：name ss

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(4) 消费消息

./kafka-console-consumer.sh --bootstrap-server ${ bootstrap_server_list} --topic ${topic_name}

运行结果为：(name,1) (ss,1)

其中：

- ${bootstrap_server_list} 表示 Kafka 集群中 Kafka Broker 地址，端口号默认为 6667，多个 Kafka Broker 节点用逗号分割。

3. 样例代码

(1) Pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>demo</artifactId>
        <groupId>com.zyf</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>KafkaFlink1.13.6Kafka</artifactId>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <flink.version>1.13.6</flink.version>
        <scala.binary.version>2.11</scala.binary.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
```

```xml
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-clients_${scala.binary.version}</artifactId>
      <version>${flink.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.0.0</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <artifactSet>
                <excludes>
                  <exclude>com.google.code.findbugs:jsr305</exclude>
                  <exclude>org.slf4j:*</exclude>
                  <exclude>log4j:*</exclude>
                </excludes>
              </artifactSet>
              <filters>
                <filter>
                  <!-- Do not copy the signatures in the META-INF folder.
                  Otherwise, this might cause SecurityExceptions when using the JAR. -->
                  <artifact>*:*</artifact>
                  <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
```

```
                        </excludes>
                      </filter>
                    </filters>
                    <transformers>
                      <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <mainClass>KafkaFlinkKafka</mainClass>
                      </transformer>
                    </transformers>
                  </configuration>
                </execution>
              </executions>
            </plugin>
          </plugins>
        </build>

</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

  private Properties producerProp = new Properties();
  private Properties consumerProp = new Properties();
  private Properties commonProp = new Properties();

  public ConfigUtils(String KafkaConfigPath) {
    try {
      Properties prop = new Properties();
      FileInputStream in = new FileInputStream(KafkaConfigPath);
      prop.load(in);
      in.close();

      String keyPrefix;
      String keyValue;
      for (String key : prop.stringPropertyNames()) {
```

```
            keyPrefix = key.trim().split("\\.")[0];
            keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
            switch (keyPrefix.toLowerCase()) {
                case "producer":
                    this.producerProp.put(keyValue, prop.getProperty(key));
                    break;
                case "consumer":
                    this.consumerProp.put(keyValue, prop.getProperty(key));
                    break;
                default:
                    this.commonProp.put(key, prop.getProperty(key));
                    break;
            }
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public Properties getProducerProp() {
    return producerProp;
}

public void setProducerProp(Properties producerProp) {
    this.producerProp = producerProp;
}

public Properties getConsumerProp() {
    return consumerProp;
}

public void setConsumerProp(Properties consumerProp) {
    this.consumerProp = consumerProp;
}

public Properties getCommonProp() {
    return commonProp;
}
```

```
    public void setCommonProp(Properties commonProp) {
        this.commonProp = commonProp;
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) KafkaFlinkKafka 类文件

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;
import org.apache.flink.util.Collector;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;
```

```
public class KafkaFlinkKafka {
  public static void main(String[] args) throws Exception {
    Logger logger = LoggerFactory.getLogger(KafkaFlinkKafka.class);

    final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
    if (!multipleParameterTool.has("path")) {
      System.out.println("Error: not exist --path /opt/your.properties");
      System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
      System.exit(0);
    }
    ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
    logger.info(configUtils.toString());
    Properties commonProp = configUtils.getCommonProp();

    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setUseSnapshotCompression(true);
    env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

    Properties consumerProp = configUtils.getConsumerProp();
    if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
        .equalsIgnoreCase("true")) {
      consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
      consumerProp.setProperty("sasl.mechanism", "GSSAPI");
      consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
    }

    Properties producerProp = configUtils.getProducerProp();
    if (commonProp.containsKey("sink.kafka.security.enable") &&
commonProp.getProperty("sink.kafka.security.enable")
        .equalsIgnoreCase("true")) {
      producerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
      producerProp.setProperty("sasl.mechanism", "GSSAPI");
      producerProp.setProperty("sasl.kerberos.service.name", "kafka");
    }
```

```
        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }

        FlinkKafkaProducer<Tuple2<String, Integer>> kafkaProducer = new
FlinkKafkaProducer<>(commonProp.getProperty("sink.kafka.topic"),
                new MySchema(), producerProp);

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
                .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        DataStream<Tuple2<String, Integer>> counts = sourceDataStream.flatMap(new
Tokenizer()).keyBy(new MyKeySelector()).sum(1);


counts.addSink(kafkaProducer).uid("sink_kafka").setParallelism(Integer.valueOf(commonProp.getPr
operty("sink.kafka.parallelism")));

        env.execute("KafkaFlink1.13.6KafkaKerOrNot");
    }

    private static final class Tokenizer implements FlatMapFunction<String, Tuple2<String, Integer>> {

        @Override
        public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
            // normalize and split the line
            String[] tokens = value.toLowerCase().split("\\W+");

            // emit the pairs
            for (String token : tokens) {
                if (token.length() > 0) {
                    out.collect(new Tuple2<>(token, 1));
```

```
            }
          }
        }
      }

      private static final class MyKeySelector implements KeySelector<Tuple2<String, Integer>, String> {
        @Override
        public String getKey(Tuple2<String, Integer> value) throws Exception {
          return value.f0;
        }
      }
    }
```

(4) MySchema 类文件

```
import org.apache.flink.api.common.serialization.DeserializationSchema;
import org.apache.flink.api.common.serialization.SerializationSchema;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.tuple.Tuple2;

import java.io.IOException;

public class MySchema implements DeserializationSchema<Tuple2<String, Integer>>,
SerializationSchema<Tuple2<String, Integer>> {
  @Override
  public Tuple2<String, Integer> deserialize(byte[] message) throws IOException {
    return null;
  }

  @Override
  public boolean isEndOfStream(Tuple2<String, Integer> nextElement) {
    return false;
  }

  @Override
  public byte[] serialize(Tuple2<String, Integer> element) {
    return element.toString().getBytes();
  }

  @Override
```

```
    public TypeInformation<Tuple2<String, Integer>> getProducedType() {
        return null;
    }
}
```

(5) FlinkJob_Kafka2Kafka.properties 配置文件

```
############################## Kafka 配置 ########################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_kafka
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka Producer 专用配置，支持所有原生配置
producer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
producer.acks=1
producer.compression.type=snappy
#Kafka 其他配置
#source
source.kafka.topic=in
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=false
#sink
sink.kafka.topic=out
sink.kafka.parallelism=1
sink.kafka.security.enable=false
```

5.1.2 Flink DataStream 读 Kafka 写 Elasticsearch

1. 样例主要功能

从 Kafka 一个 topic 读取数据，写入到 Elasticsearch。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic
${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 执行 Flink 作业

```
flink run -t yarn-per-job -d ${jar path}/KafkaFlink1.13.6Es-1.0.jar --path
FlinkJob_Kafka2ES.properties
```

其中：

- ${jar path}表示样例 jar 包路径。

(3) 生产消息

```
./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name}
```

消息格式：name

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(4) 查看执行结果

```
curl http://zyf53:9200/kafka-flink-es-index/_search?pretty
```

3. 样例代码

(1) Pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <parent>
      <artifactId>demo</artifactId>
      <groupId>com.zyf</groupId>
      <version>1.0</version>
   </parent>
   <modelVersion>4.0.0</modelVersion>

   <artifactId>KafkaFlink1.13.6Es</artifactId>

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <flink.version>1.13.6</flink.version>
      <scala.binary.version>2.11</scala.binary.version>
```

```
</properties>

<dependencies>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-elasticsearch7_2.11</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.0.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
```

```
                    <goal>shade</goal>
                  </goals>
                  <configuration>
                    <artifactSet>
                      <excludes>
                        <exclude>com.google.code.findbugs:jsr305</exclude>
                        <exclude>org.slf4j:*</exclude>
                        <exclude>log4j:*</exclude>
                      </excludes>
                    </artifactSet>
                    <filters>
                      <filter>
                        <!-- Do not copy the signatures in the META-INF folder.
                        Otherwise, this might cause SecurityExceptions when using the JAR. -->
                        <artifact>*:*</artifact>
                        <excludes>
                          <exclude>META-INF/*.SF</exclude>
                          <exclude>META-INF/*.DSA</exclude>
                          <exclude>META-INF/*.RSA</exclude>
                        </excludes>
                      </filter>
                    </filters>
                    <transformers>
                      <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <mainClass>KafkaFlinkES</mainClass>
                      </transformer>
                    </transformers>
                  </configuration>
                </execution>
              </executions>
            </plugin>
          </plugins>
        </build>

</project>
```

(2) ConfigUtils 类文件

```java
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

    private Properties producerProp = new Properties();
    private Properties consumerProp = new Properties();
    private Properties commonProp = new Properties();

    public ConfigUtils(String KafkaConfigPath) {
        try {
            Properties prop = new Properties();
            FileInputStream in = new FileInputStream(KafkaConfigPath);
            prop.load(in);
            in.close();

            String keyPrefix;
            String keyValue;
            for (String key : prop.stringPropertyNames()) {
                keyPrefix = key.trim().split("\\.")[0];
                keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
                switch (keyPrefix.toLowerCase()) {
                    case "producer":
                        this.producerProp.put(keyValue, prop.getProperty(key));
                        break;
                    case "consumer":
                        this.consumerProp.put(keyValue, prop.getProperty(key));
                        break;
                    default:
                        this.commonProp.put(key, prop.getProperty(key));
                        break;
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
```

```
public Properties getProducerProp() {
    return producerProp;
}

public void setProducerProp(Properties producerProp) {
    this.producerProp = producerProp;
}

public Properties getConsumerProp() {
    return consumerProp;
}

public void setConsumerProp(Properties consumerProp) {
    this.consumerProp = consumerProp;
}

public Properties getCommonProp() {
    return commonProp;
}

public void setCommonProp(Properties commonProp) {
    this.commonProp = commonProp;
}

@Override
public String toString() {
    return "ConfigUtils{" +
            "producerProp=" + producerProp +
            ", consumerProp=" + consumerProp +
            ", commonProp=" + commonProp +
            '}';
}

public static void main(String[] args) {
    System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
    System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
    System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
}
```

```
}
```

(3) KafkaFlinkES 类文件

```
import org.apache.flink.api.common.functions.RuntimeContext;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.elasticsearch.ElasticsearchSinkFunction;
import org.apache.flink.streaming.connectors.elasticsearch.RequestIndexer;
import
org.apache.flink.streaming.connectors.elasticsearch.util.RetryRejectedExecutionFailureHandler;
import org.apache.flink.streaming.connectors.elasticsearch7.ElasticsearchSink;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.http.HttpHost;
import org.elasticsearch.action.index.IndexRequest;
import org.elasticsearch.client.Requests;
import org.elasticsearch.common.settings.SecureString;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Properties;

public class KafkaFlinkES {
  public static void main(String[] args) throws Exception {
    Logger logger = LoggerFactory.getLogger(KafkaFlinkES.class);

    final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
    if (!multipleParameterTool.has("path")) {
      System.out.println("Error: not exist --path /opt/your.properties");
      System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
      System.exit(0);
    }
```

```
        ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
        logger.info(configUtils.toString());
        Properties commonProp = configUtils.getCommonProp();

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.getConfig().setUseSnapshotCompression(true);
        env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

        Properties consumerProp = configUtils.getConsumerProp();
        if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
                .equalsIgnoreCase("true")) {
            consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
            consumerProp.setProperty("sasl.mechanism", "GSSAPI");
            consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
                .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        List<HttpHost> httpHosts = new ArrayList<>();
        for (String ipPort : commonProp.getProperty("sink.es.servers").trim().split(",")) {
            String[] hostPort = ipPort.trim().split(":");
            httpHosts.add(new HttpHost(hostPort[0].trim(), Integer.valueOf(hostPort[1].trim()), "http"));
        }
```

```
    // use a ElasticsearchSink.Builder to create an ElasticsearchSink
    ElasticsearchSink.Builder<String> esSinkBuilder = new ElasticsearchSink.Builder<>(
        httpHosts,
        new ElasticsearchSinkFunction<String>() {
          public IndexRequest createIndexRequest(String element) {
            HashMap<String, String> json = new HashMap<>();
            json.put("data", element);

            return Requests.indexRequest()
                .index(commonProp.getProperty("sink.es.index"))
                .type(commonProp.getProperty("sink.es.type"))
                .source(json);
          }

          @Override
          public void process(String element, RuntimeContext runtimeContext, RequestIndexer
requestIndexer) {
            requestIndexer.add(createIndexRequest(element));
          }
        }
    );

    // configuration for the bulk requests; this instructs the sink to emit after every element,
otherwise they would be buffered

esSinkBuilder.setBulkFlushMaxActions(Integer.valueOf(commonProp.getProperty("sink.es.bulk.flush
.max.actions")));

esSinkBuilder.setBulkFlushInterval(Long.valueOf(commonProp.getProperty("sink.es.bulk.flush.inter
val.ms")));

esSinkBuilder.setBulkFlushMaxSizeMb(Integer.valueOf(commonProp.getProperty("sink.es.bulk.flush
.max.size.mb")));
    esSinkBuilder.setFailureHandler(new RetryRejectedExecutionFailureHandler());

    if (commonProp.getProperty("sink.es.security.enable").equalsIgnoreCase("true")) {
      esSinkBuilder.setRestClientFactory(restClientBuilder -> {
        if (commonProp.getProperty("sink.es.usr.pwd.enable").equalsIgnoreCase("true")) {
```

```
                restClientBuilder.setHttpClientConfigCallback(new
SpnegoHttpClientConfigCallbackHandler(commonProp.getProperty("sink.es.usr"),
                    new SecureString(commonProp.getProperty("sink.es.pwd")), false));
            } else {
                restClientBuilder.setHttpClientConfigCallback(new
SpnegoHttpClientConfigCallbackHandler(commonProp.getProperty("sink.es.usr"),
                    commonProp.getProperty("sink.es.keytab.path"), false));
            }

        });
    }


sourceDataStream.addSink(esSinkBuilder.build()).uid("sink_es").setParallelism(Integer.valueOf(com
monProp.getProperty("sink.es.parallelism")));

    env.execute("KafkaFlink1.13.6ESKerOrNot");
  }
}
```

(4) SpnegoHttpClientConfigCallbackHandler 类文件

```
import org.apache.http.auth.AuthSchemeProvider;
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.Credentials;
import org.apache.http.auth.KerberosCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.client.config.AuthSchemes;
import org.apache.http.config.Lookup;
import org.apache.http.config.RegistryBuilder;
import org.apache.http.impl.auth.SPNegoSchemeFactory;
import org.apache.http.impl.nio.client.HttpAsyncClientBuilder;
import org.elasticsearch.ExceptionsHelper;
import org.elasticsearch.client.RestClientBuilder.HttpClientConfigCallback;
import org.elasticsearch.common.settings.SecureString;
import org.ietf.jgss.*;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
```

```
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.kerberos.KerberosPrincipal;
import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.Configuration;
import javax.security.auth.login.LoginContext;
import java.io.IOException;
import java.security.AccessControlContext;
import java.security.AccessController;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class SpnegoHttpClientConfigCallbackHandler implements HttpClientConfigCallback {
   private static final String SUN_KRB5_LOGIN_MODULE =
"com.sun.security.auth.module.Krb5LoginModule";
   private static final String CRED_CONF_NAME = "ESClientLoginConf";
   private static final Oid SPNEGO_OID = getSpnegoOid();

   private static Oid getSpnegoOid() {
      Oid oid = null;
      try {
         oid = new Oid("1.3.6.1.5.5.2");
      } catch (GSSException gsse) {
         throw ExceptionsHelper.convertToRuntime(gsse);
      }
      return oid;
   }

   private final String userPrincipalName;
   private final SecureString password;
   private final String keytabPath;
   private final boolean enableDebugLogs;
   private LoginContext loginContext;
```

```
    /**
     * Constructs {@link SpnegoHttpClientConfigCallbackHandler} with given
     * principalName and password.
     *
     * @param userPrincipalName user principal name
     * @param password          password for user
     * @param enableDebugLogs   if {@code true} enables kerberos debug logs
     */
    public SpnegoHttpClientConfigCallbackHandler(final String userPrincipalName, final SecureString
password,
                                                  final boolean enableDebugLogs) {
        this.userPrincipalName = userPrincipalName;
        this.password = password;
        this.keytabPath = null;
        this.enableDebugLogs = enableDebugLogs;
    }

    /**
     * Constructs {@link SpnegoHttpClientConfigCallbackHandler} with given
     * principalName and keytab.
     *
     * @param userPrincipalName User principal name
     * @param keytabPath        path to keytab file for user
     * @param enableDebugLogs   if {@code true} enables kerberos debug logs
     */
    public SpnegoHttpClientConfigCallbackHandler(final String userPrincipalName, final String
keytabPath, final boolean enableDebugLogs) {
        this.userPrincipalName = userPrincipalName;
        this.keytabPath = keytabPath;
        this.password = null;
        this.enableDebugLogs = enableDebugLogs;
    }

    @Override
    public HttpAsyncClientBuilder customizeHttpClient(HttpAsyncClientBuilder httpClientBuilder) {
        setupSpnegoAuthSchemeSupport(httpClientBuilder);
        return httpClientBuilder;
    }
```

```
  private void setupSpnegoAuthSchemeSupport(HttpAsyncClientBuilder httpClientBuilder) {
    final Lookup<AuthSchemeProvider> authSchemeRegistry =
RegistryBuilder.<AuthSchemeProvider>create()
        .register(AuthSchemes.SPNEGO, new SPNegoSchemeFactory()).build();

    final GSSManager gssManager = GSSManager.getInstance();
    try {
      final GSSName gssUserPrincipalName = gssManager.createName(userPrincipalName,
GSSName.NT_USER_NAME);
      login();
      final AccessControlContext acc = AccessController.getContext();
      final GSSCredential credential = doAsPrivilegedWrapper(loginContext.getSubject(),
          (PrivilegedExceptionAction<GSSCredential>) () ->
gssManager.createCredential(gssUserPrincipalName,
              GSSCredential.DEFAULT_LIFETIME, SPNEGO_OID, GSSCredential.INITIATE_ONLY),
          acc);

      final KerberosCredentialsProvider credentialsProvider = new KerberosCredentialsProvider();
      credentialsProvider.setCredentials(
          new AuthScope(AuthScope.ANY_HOST, AuthScope.ANY_PORT, AuthScope.ANY_REALM,
AuthSchemes.SPNEGO),
          new KerberosCredentials(credential));
      httpClientBuilder.setDefaultCredentialsProvider(credentialsProvider);
    } catch (GSSException e) {
      throw new RuntimeException(e);
    } catch (PrivilegedActionException e) {
      throw new RuntimeException(e.getCause());
    }
    httpClientBuilder.setDefaultAuthSchemeRegistry(authSchemeRegistry);
  }

  /**
   * If logged in {@link LoginContext} is not available, it attempts login and
   * returns {@link LoginContext}
   *
   * @return {@link LoginContext}
   * @throws PrivilegedActionException
```

```
    */
   public synchronized LoginContext login() throws PrivilegedActionException {
      if (this.loginContext == null) {
         AccessController.doPrivileged((PrivilegedExceptionAction<Void>) () -> {
            final Subject subject = new Subject(false, Collections.singleton(new
KerberosPrincipal(userPrincipalName)),
                  Collections.emptySet(), Collections.emptySet());
            Configuration conf = null;
            final CallbackHandler callback;
            if (password != null) {
               conf = new PasswordJaasConf(userPrincipalName, enableDebugLogs);
               callback = new KrbCallbackHandler(userPrincipalName, password);
            } else {
               conf = new KeytabJaasConf(userPrincipalName, keytabPath, enableDebugLogs);
               callback = null;
            }
            loginContext = new LoginContext(CRED_CONF_NAME, subject, callback, conf);
            loginContext.login();
            return null;
         });
      }
      return loginContext;
   }

   /**
    * Privileged Wrapper that invokes action with Subject.doAs to perform work as
    * given subject.
    *
    * @param subject {@link Subject} to be used for this work
    * @param action  {@link PrivilegedExceptionAction} action for performing inside
    *            Subject.doAs
    * @param acc     the {@link AccessControlContext} to be tied to the specified
    *            subject and action see
    *            {@link Subject#doAsPrivileged(Subject, PrivilegedExceptionAction,
AccessControlContext)
    * @return the value returned by the PrivilegedExceptionAction's run method
    * @throws PrivilegedActionException
    */
```

```
    static <T> T doAsPrivilegedWrapper(final Subject subject, final PrivilegedExceptionAction<T>
action, final AccessControlContext acc)
            throws PrivilegedActionException {
        try {
            return AccessController.doPrivileged((PrivilegedExceptionAction<T>) () ->
Subject.doAsPrivileged(subject, action, acc));
        } catch (PrivilegedActionException pae) {
            if (pae.getCause() instanceof PrivilegedActionException) {
                throw (PrivilegedActionException) pae.getCause();
            }
            throw pae;
        }
    }

    /**
     * This class matches {@link AuthScope} and based on that returns
     * {@link Credentials}. Only supports {@link AuthSchemes#SPNEGO} in
     * {@link AuthScope#getScheme()}
     */
    private static class KerberosCredentialsProvider implements CredentialsProvider {
        private AuthScope authScope;
        private Credentials credentials;

        @Override
        public void setCredentials(AuthScope authscope, Credentials credentials) {
            if (authscope.getScheme().regionMatches(true, 0, AuthSchemes.SPNEGO, 0,
AuthSchemes.SPNEGO.length()) == false) {
                throw new IllegalArgumentException("Only " + AuthSchemes.SPNEGO + " auth scheme is
supported in AuthScope");
            }
            this.authScope = authscope;
            this.credentials = credentials;
        }

        @Override
        public Credentials getCredentials(AuthScope authscope) {
            assert this.authScope != null && authscope != null;
            return authscope.match(this.authScope) > -1 ? this.credentials : null;
```

```
    }

    @Override
    public void clear() {
      this.authScope = null;
      this.credentials = null;
    }
  }

  /**
   * Jaas call back handler to provide credentials.
   */
  private static class KrbCallbackHandler implements CallbackHandler {
    private final String principal;
    private final SecureString password;

    KrbCallbackHandler(final String principal, final SecureString password) {
      this.principal = principal;
      this.password = password;
    }

    public void handle(final Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
      for (Callback callback : callbacks) {
        if (callback instanceof PasswordCallback) {
          PasswordCallback pc = (PasswordCallback) callback;
          if (pc.getPrompt().contains(principal)) {
            pc.setPassword(password.getChars());
            break;
          }
        }
      }
    }
  }

  /**
   * Usually we would have a JAAS configuration file for login configuration.
   * Instead of an additional file setting as we do not want the options to be
```

```
 * customizable we are constructing it in memory.
 * <p>
 * As we are using this instead of jaas.conf, this requires refresh of
 * {@link Configuration} and reqires appropriate security permissions to do so.
 */
private static class PasswordJaasConf extends AbstractJaasConf {

    PasswordJaasConf(final String userPrincipalName, final boolean enableDebugLogs) {
        super(userPrincipalName, enableDebugLogs);
    }

    public void addOptions(final Map<String, String> options) {
        options.put("useTicketCache", Boolean.FALSE.toString());
        options.put("useKeyTab", Boolean.FALSE.toString());
    }
}

/**
 * Usually we would have a JAAS configuration file for login configuration. As
 * we have static configuration except debug flag, we are constructing in
 * memory. This avoids additional configuration required from the user.
 * <p>
 * As we are using this instead of jaas.conf, this requires refresh of
 * {@link Configuration} and requires appropriate security permissions to do so.
 */
private static class KeytabJaasConf extends AbstractJaasConf {
    private final String keytabFilePath;

    KeytabJaasConf(final String userPrincipalName, final String keytabFilePath, final boolean
enableDebugLogs) {
        super(userPrincipalName, enableDebugLogs);
        this.keytabFilePath = keytabFilePath;
    }

    public void addOptions(final Map<String, String> options) {
        options.put("useKeyTab", Boolean.TRUE.toString());
        options.put("keyTab", keytabFilePath);
        options.put("doNotPrompt", Boolean.TRUE.toString());
```

```java
        }

    }

    private abstract static class AbstractJaasConf extends Configuration {
        private final String userPrincipalName;
        private final boolean enableDebugLogs;

        AbstractJaasConf(final String userPrincipalName, final boolean enableDebugLogs) {
            this.userPrincipalName = userPrincipalName;
            this.enableDebugLogs = enableDebugLogs;
        }

        @Override
        public AppConfigurationEntry[] getAppConfigurationEntry(final String name) {
            final Map<String, String> options = new HashMap<>();
            options.put("principal", userPrincipalName);
            options.put("isInitiator", Boolean.TRUE.toString());
            options.put("storeKey", Boolean.TRUE.toString());
            options.put("debug", Boolean.toString(enableDebugLogs));
            addOptions(options);
            return new AppConfigurationEntry[]{new
AppConfigurationEntry(SUN_KRB5_LOGIN_MODULE,
                AppConfigurationEntry.LoginModuleControlFlag.REQUIRED,
Collections.unmodifiableMap(options))};
        }

        abstract void addOptions(Map<String, String> options);
    }
}
```

(5) FlinkJob_Kafka2ES.properties 配置文件

```properties
############################# Kafka 配置 ######################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_es
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
```

```
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_es
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=false
######################################## ES 配置##################################
sink.es.servers=10.121.65.53:9200,10.121.65.54:9200,10.121.65.55:9200
sink.es.index=kafka-flink-es-index
sink.es.type=kafka-flink-es-type
sink.es.bulk.flush.max.actions=1
sink.es.bulk.flush.max.size.mb=50
sink.es.bulk.flush.interval.ms=1000
sink.es.parallelism=1
#kerberos 相关
sink.es.security.enable=false
sink.es.usr.pwd.enable=true
sink.es.usr=useradmin
sink.es.pwd=admin@123
sink.es.keytab.path=/opt/useradmin.keytab
```

5.1.3 Flink DataStream 读 Kafka 写 HBase

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据写入到 HBase 的一个表中。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 创建 HBase 表

```
su hbase
登录 hbase shell
create 'test','cf'
```

(3) 执行 Flink 作业

flink run -t yarn-per-job -d ${jar path}/KafkaFlink1.13.6Hbase-1.0.jar --path FlinkJob_Kafka2Hbase.properties

其中：

- ${jar path}表示样例 jar 包路径。

(4) 生产消息

./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name}

消息格式：Tom

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(5) 查看执行结果

登录 hbase shell

scan 'test’

3. 样例代码

(1) Pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>demo</artifactId>
    <groupId>com.zyf</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>KafkaFlink1.13.6Hbase</artifactId>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <flink.version>1.13.6</flink.version>
    <scala.binary.version>2.11</scala.binary.version>
    <hbase.version>2.1.0-cdh6.2.0</hbase.version>
  </properties>
```

```xml
<dependencies>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-server</artifactId>
    <version>${hbase.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.0.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
```

```
                <artifactSet>
                  <excludes>
                    <exclude>com.google.code.findbugs:jsr305</exclude>
                    <exclude>org.slf4j:*</exclude>
                    <exclude>log4j:*</exclude>
                  </excludes>
                </artifactSet>
                <filters>
                  <filter>
                    <!-- Do not copy the signatures in the META-INF folder.
                    Otherwise, this might cause SecurityExceptions when using the JAR. -->
                    <artifact>*:*</artifact>
                    <excludes>
                      <exclude>META-INF/*.SF</exclude>
                      <exclude>META-INF/*.DSA</exclude>
                      <exclude>META-INF/*.RSA</exclude>
                    </excludes>
                  </filter>
                </filters>
                <transformers>
                  <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                    <mainClass>KafkaFlinkHbase</mainClass>
                  </transformer>
                </transformers>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
  </build>

</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;
```

```
public class ConfigUtils {

    private Properties producerProp = new Properties();
    private Properties consumerProp = new Properties();
    private Properties commonProp = new Properties();

    public ConfigUtils(String KafkaConfigPath) {
        try {
            Properties prop = new Properties();
            FileInputStream in = new FileInputStream(KafkaConfigPath);
            prop.load(in);
            in.close();

            String keyPrefix;
            String keyValue;
            for (String key : prop.stringPropertyNames()) {
                keyPrefix = key.trim().split("\\.")[0];
                keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
                switch (keyPrefix.toLowerCase()) {
                    case "producer":
                        this.producerProp.put(keyValue, prop.getProperty(key));
                        break;
                    case "consumer":
                        this.consumerProp.put(keyValue, prop.getProperty(key));
                        break;
                    default:
                        this.commonProp.put(key, prop.getProperty(key));
                        break;
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public Properties getProducerProp() {
        return producerProp;
    }
```

```
    public void setProducerProp(Properties producerProp) {
        this.producerProp = producerProp;
    }

    public Properties getConsumerProp() {
        return consumerProp;
    }

    public void setConsumerProp(Properties consumerProp) {
        this.consumerProp = consumerProp;
    }

    public Properties getCommonProp() {
        return commonProp;
    }

    public void setCommonProp(Properties commonProp) {
        this.commonProp = commonProp;
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) KafkaFlinkHbase 类文件

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class KafkaFlinkHbase {
  public static void main(String[] args) throws Exception {
    Logger logger = LoggerFactory.getLogger(KafkaFlinkHbase.class);

    final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
    if (!multipleParameterTool.has("path")) {
      System.out.println("Error: not exist --path /opt/your.properties");
      System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
      System.exit(0);
    }
    ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
    logger.info(configUtils.toString());
    Properties commonProp = configUtils.getCommonProp();

    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setUseSnapshotCompression(true);
    env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

    Properties consumerProp = configUtils.getConsumerProp();
```

```
        if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
            .equalsIgnoreCase("true")) {
          consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
          consumerProp.setProperty("sasl.mechanism", "GSSAPI");
          consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
            new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
            .equalsIgnoreCase("latest")) {
          kafkaConsumer.setStartFromLatest();
        }

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
            .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        DataStream<Map<Object, Object>> convertData = sourceDataStream.map(new
MapFunction<String, Map<Object, Object>>() {
          @Override
          public Map<Object, Object> map(String value) {
            Map<Object, Object> dataMap = new HashMap<>();
            dataMap.put("rowKey", "defaultRowKey" + System.currentTimeMillis());
            dataMap.put("columnFamily", "cf");
            dataMap.put("cfQualifier", "cfq");
            dataMap.put("data", value);
            return dataMap;
          }
        }).uid("convert_data").setParallelism(Integer.valueOf(commonProp.getProperty("convert.forma
t.parallelism")));

        convertData.addSink(new SinkToHbase<>(commonProp)).uid("sink_hbase")
            .setParallelism(Integer.valueOf(commonProp.getProperty("sink.hbase.parallelism")));

        env.execute("KafkaFlink1.13.6Hbase kerberos or not");
```

```
    }
}
```

(4) SinkToHbase 类文件

```
import org.apache.flink.annotation.Internal;
import org.apache.flink.api.common.functions.RuntimeContext;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.HashMap;
import java.util.Properties;

public class SinkToHbase<IN> extends RichSinkFunction<IN> implements CheckpointedFunction {
    private Logger logger = LoggerFactory.getLogger(SinkToHbase.class);

    private static final long serialVersionUID = 1L;

    private String hbaseConfPath;

    private Connection conn;

    private BufferedMutator mutator;

    private String tableName;

    private long writeBufferSize;
```

```
SinkToHbase(Properties properties) {
    this.hbaseConfPath = properties.getProperty("sink.hbase.conf.path");
    this.writeBufferSize = Long.valueOf(properties.getProperty("sink.hbase.client.write.buffer"));

    this.tableName = properties.getProperty("sink.hbase.table.name");

}

@Override
public void open(Configuration parameters) throws Exception {
    RuntimeContext ctx = getRuntimeContext();
    org.apache.hadoop.conf.Configuration config = HBaseConfiguration.create();
    config.addResource(new Path(hbaseConfPath));
    conn = ConnectionFactory.createConnection(config);

    BufferedMutatorParams bmParams = new
BufferedMutatorParams(TableName.valueOf(tableName));

    bmParams.writeBufferSize(this.writeBufferSize);

    mutator = conn.getBufferedMutator(bmParams);

    logger.info("Starting SinkToHbase ({}/{}) to produce into table {}",
        ctx.getIndexOfThisSubtask() + 1, ctx.getNumberOfParallelSubtasks(), this.tableName);
}


@Override
public void invoke(IN value, Context context) throws Exception {
    if (value != null) {
        HashMap<String, String> dataMap = (HashMap<String, String>) value;
        mutator.mutate(new Put(Bytes.toBytes(dataMap.get("rowKey")))
            .addColumn(Bytes.toBytes(dataMap.get("columnFamily")),
                Bytes.toBytes(dataMap.get("cfQualifier")),
                Bytes.toBytes(dataMap.get("data"))));
    }
}
```

```
    @Override
    public void close() throws Exception {
        mutator.close();
        conn.close();
    }

    @Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception {

    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {

    }
}
```

(5) FlinkJob_Kafka2Hbase.properties 配置文件

```
############################## Kafka 配置 ########################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_hbase
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_hbase
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=false
######################################### Hbase 配置###################################
sink.hbase.conf.path=/etc/hbase/conf/hbase-site.xml
sink.hbase.client.write.buffer=1
sink.hbase.table.name=test
convert.format.parallelism=1
sink.hbase.parallelism=1
```

5.1.4 Flink DataStream 读 Kafka 写 Hive

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据写入到 Hive 的一个表中。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 创建 Hive 表

```
beeline
create table IF NOT EXISTS hc_table(id int,name string)ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
```

(3) 执行 Flink 作业

```
flink run -m yarn-cluster -d /opt/KafkaFlink1.13.6HiveSink-1.0.jar --path /opt/FlinkJob_Kafka2Hive.properties
```

(4) 生产消息

```
./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name}
```

消息格式：Tom

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(5) 查看执行结果

```
beeline
select * from hc_table;
```

3. 样例代码

(1) Pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>demo</artifactId>
```

```xml
    <groupId>com.zyf</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>KafkaFlink1.13.6HiveSink</artifactId>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <flink.version>1.13.6</flink.version>
    <scala.binary.version>2.11</scala.binary.version>
    <hive.version>2.1.1-cdh6.2.0</hive.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-clients_2.11</artifactId>
      <version>${flink.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.hive</groupId>
      <artifactId>hive-jdbc</artifactId>
      <version>${hive.version}</version>
    </dependency>
    <dependency>
```

```
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-filesystem_2.11</artifactId>
      <version>${flink.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.0.0</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <artifactSet>
                <excludes>
                  <exclude>com.google.code.findbugs:jsr305</exclude>
                  <exclude>org.slf4j:*</exclude>
                  <exclude>log4j:*</exclude>
                </excludes>
              </artifactSet>
              <filters>
                <filter>
                  <!-- Do not copy the signatures in the META-INF folder.
                  Otherwise, this might cause SecurityExceptions when using the JAR. -->
                  <artifact>*:*</artifact>
                  <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
                  </excludes>
                </filter>
              </filters>
```

```
                    <transformers>
                      <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <mainClass>kafkaFlinkHive</mainClass>
                      </transformer>
                    </transformers>
                  </configuration>
                </execution>
              </executions>
            </plugin>
          </plugins>
        </build>

</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

  private Properties producerProp = new Properties();
  private Properties consumerProp = new Properties();
  private Properties commonProp = new Properties();

  public ConfigUtils(String KafkaConfigPath) {
    try {
      Properties prop = new Properties();
      FileInputStream in = new FileInputStream(KafkaConfigPath);
      prop.load(in);
      in.close();

      String keyPrefix;
      String keyValue;
      for (String key : prop.stringPropertyNames()) {
        keyPrefix = key.trim().split("\\.")[0];
        keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
        switch (keyPrefix.toLowerCase()) {
```

```
            case "producer":
              this.producerProp.put(keyValue, prop.getProperty(key));
              break;
            case "consumer":
              this.consumerProp.put(keyValue, prop.getProperty(key));
              break;
            default:
              this.commonProp.put(key, prop.getProperty(key));
              break;
          }
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public Properties getProducerProp() {
    return producerProp;
}

public void setProducerProp(Properties producerProp) {
    this.producerProp = producerProp;
}

public Properties getConsumerProp() {
    return consumerProp;
}

public void setConsumerProp(Properties consumerProp) {
    this.consumerProp = consumerProp;
}

public Properties getCommonProp() {
    return commonProp;
}

public void setCommonProp(Properties commonProp) {
    this.commonProp = commonProp;
```

```
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) kafkaFlinkHive 类文件

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.Encoder;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.DefaultRollingPolicy;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.security.UserGroupInformation;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.io.PrintStream;
import java.util.Properties;
```

```
import java.util.concurrent.TimeUnit;
import java.util.regex.Pattern;


public class kafkaFlinkHive {
  static Logger logger = LoggerFactory.getLogger(kafkaFlinkHive.class);

  public static void main(String[] args) throws Exception {


    final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
    if (!multipleParameterTool.has("path")) {
      System.out.println("Error: not exist --path /opt/your.properties");
      System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
      System.exit(0);
    }
    ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
    logger.info(configUtils.toString());
    Properties commonProp = configUtils.getCommonProp();

    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setUseSnapshotCompression(true);
    env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

    Properties consumerProp = configUtils.getConsumerProp();
    if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
        .equalsIgnoreCase("true")) {
      consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
      consumerProp.setProperty("sasl.mechanism", "GSSAPI");
      consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
    }

    FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
```

```
            new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
            .equalsIgnoreCase("latest")) {
          kafkaConsumer.setStartFromLatest();
        }

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
            .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        DataStream<String> convertData = sourceDataStream.map(new MapFunction<String, String>()
{
          @Override
          public String map(String value) {
            return value;
          }
        }).uid("convert_data").setParallelism(Integer.valueOf(commonProp.getProperty("convert.forma
t.parallelism")));

        final StreamingFileSink<String> hdfsSink = StreamingFileSink
            .forRowFormat(new Path("hdfs:///tmp"), (Encoder<String>) (element, stream) -> {
              PrintStream out = new PrintStream(stream);
              String[] strArr = element.split(",");
              if (strArr.length == 2 && isInteger(strArr[0])) {
                out.println(element);
              } else {
                logger.error("data format must be : int,string");
              }

              if (commonProp.containsKey("sink.hive.security.enable") &&
commonProp.getProperty("sink.hive.security.enable")
                  .equalsIgnoreCase("true")) {
                Configuration conf = new Configuration();
                conf.set("hadoop.security.authentication", "Kerberos");
                UserGroupInformation.setConfiguration(conf);
                try {
```

```
UserGroupInformation.loginUserFromKeytab(commonProp.getProperty("sink.hive.keytab.principal
"),
                        commonProp.getProperty("sink.hive.keytab.path"));
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

        }).withRollingPolicy(DefaultRollingPolicy.create()
                .withRolloverInterval(TimeUnit.SECONDS.toMillis(Integer.valueOf(commonProp.getPro
perty("hdfs.rollover.interval.sec"))))
                .withInactivityInterval(TimeUnit.SECONDS.toMillis(Integer.valueOf(commonProp.getPr
operty("hdfs.inactivity.interval.sec"))))
                .withMaxPartSize(1024L * 1024L * 256L).build())
        .build();


convertData.addSink(hdfsSink).uid("sink_hdfs").setParallelism(Integer.valueOf(commonProp.getPro
perty("sink.hdfs.parallelism")));

    convertData.addSink(new
SinkToHive<>(commonProp)).uid("sink_hive").setParallelism(Integer.valueOf(commonProp.getProp
erty("sink.hdfs.parallelism")));

    env.execute("Kafka2Flink2Hive kerberos Example");
  }

  public static boolean isInteger(String str) {
    Pattern pattern = Pattern.compile("^[-\\+]?[\\d]*$");
    return pattern.matcher(str).matches();
  }
}
```

(4) SinkToHive 类文件

```
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
```

```
import org.apache.hive.service.cli.HiveSQLException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.util.Properties;

public class SinkToHive<IN> extends RichSinkFunction<IN> implements CheckpointedFunction {

    private Logger logger = LoggerFactory.getLogger(SinkToHive.class);

    private String url;
    private String tableName;

    private Connection conn = null;
    private PreparedStatement preparedStatement = null;

    SinkToHive(Properties properties) {
        this.url = properties.getProperty("sink.hive.jdbc.url");
        this.tableName = properties.getProperty("sink.hive.table.name");
    }

    @Override
    public void open(Configuration parameters) throws Exception {
        Class.forName("org.apache.hive.jdbc.HiveDriver");
        conn = DriverManager.getConnection(url);
        logger.info("================open");
    }

    @Override
    public void close() throws Exception {
        try {
            if (preparedStatement != null) {
                preparedStatement.close();
            }
```

```
            if (conn != null) {
                conn.close();
            }
        } catch (Exception ex) {
            logger.error(ex.getMessage());
        }
    }

    @Override
    public void invoke(IN value, Context context) throws Exception {
        try {
            preparedStatement = conn.prepareStatement("load data inpath '/tmp/2020*/part-*' into
table " + tableName);
            preparedStatement.execute();
        } catch (HiveSQLException ex) {
            logger.debug(ex.getMessage());
        }
    }

    @Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception {

    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {

    }


}
```

(5) FlinkJob_Kafka2Hive.properties 配置文件

```
############################## Kafka 配置 #######################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_hive
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
```

```
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_hive
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=false
###################################### hive 配置###############################
convert.format.parallelism=1
sink.hive.table.name=hc_table
sink.hdfs.parallelism=1
sink.hive.jdbc.url=jdbc:hive2://zyf53.hde.com:2181,zyf55.hde.com:2181,zyf54.hde.com:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2
hdfs.rollover.interval.sec=1
hdfs.inactivity.interval.sec=1
sink.hive.security.enable=false
sink.hive.keytab.path=/opt/useradmin.keytab
sink.hive.keytab.principal=useradmin
```

5.1.5 Flink DataStream 读 Kafka 写 Redis

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据写入到 Redis。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 执行 Flink 作业

```
flink run -m yarn-cluster -d KafkaFlink1.13.6Redis-1.0.jar --path FlinkJob_Kafka2Redis.properties
```

(3) 生产消息

```
./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name}
```

消息格式：Tom

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(4) 查看执行结果

```
redis-cli -p 7001 -a CloudOS5#DE3@Redis -h 10.121.65.53 -c
get 'kafka_flink_redis'
```

3. 样例代码

(1) Pom 文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>demo</artifactId>
    <groupId>com.zyf</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>KafkaFlink1.13.6Redis</artifactId>



  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <flink.version>1.13.6</flink.version>
    <scala.binary.version>2.11</scala.binary.version>
    <hbase.version>2.1.0-cdh6.2.0</hbase.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
```

```
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-clients_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>3.3.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <artifactSet>
                            <excludes>
                                <exclude>com.google.code.findbugs:jsr305</exclude>
                                <exclude>org.slf4j:*</exclude>
                                <exclude>log4j:*</exclude>
                            </excludes>
                        </artifactSet>
```

```
                    <filters>
                      <filter>
                        <!-- Do not copy the signatures in the META-INF folder.
                        Otherwise, this might cause SecurityExceptions when using the JAR. -->
                        <artifact>*:*</artifact>
                        <excludes>
                          <exclude>META-INF/*.SF</exclude>
                          <exclude>META-INF/*.DSA</exclude>
                          <exclude>META-INF/*.RSA</exclude>
                        </excludes>
                      </filter>
                    </filters>
                    <transformers>
                      <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <mainClass>KafkaFlinkRedis</mainClass>
                      </transformer>
                    </transformers>
                  </configuration>
                </execution>
              </executions>
            </plugin>
          </plugins>
        </build>
      </project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

  private Properties producerProp = new Properties();
  private Properties consumerProp = new Properties();
  private Properties commonProp = new Properties();

  public ConfigUtils(String KafkaConfigPath) {
    try {
```

```
        Properties prop = new Properties();
        FileInputStream in = new FileInputStream(KafkaConfigPath);
        prop.load(in);
        in.close();

        String keyPrefix;
        String keyValue;
        for (String key : prop.stringPropertyNames()) {
            keyPrefix = key.trim().split("\\.")[0];
            keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
            switch (keyPrefix.toLowerCase()) {
                case "producer":
                    this.producerProp.put(keyValue, prop.getProperty(key));
                    break;
                case "consumer":
                    this.consumerProp.put(keyValue, prop.getProperty(key));
                    break;
                default:
                    this.commonProp.put(key, prop.getProperty(key));
                    break;
            }
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public Properties getProducerProp() {
    return producerProp;
}

public void setProducerProp(Properties producerProp) {
    this.producerProp = producerProp;
}

public Properties getConsumerProp() {
    return consumerProp;
}
```

```
    public void setConsumerProp(Properties consumerProp) {
        this.consumerProp = consumerProp;
    }

    public Properties getCommonProp() {
        return commonProp;
    }

    public void setCommonProp(Properties commonProp) {
        this.commonProp = commonProp;
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) KafkaFlinkRedis 类文件

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

import java.util.Date;
import java.util.Properties;

public class KafkaFlinkRedis {
    public static void main(String[] args) throws Exception {
        Logger logger = LoggerFactory.getLogger(KafkaFlinkRedis.class);

        final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
        if (!multipleParameterTool.has("path")) {
            System.out.println("Error: not exist --path /opt/your.properties");
            System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
            System.exit(0);
        }
        ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
        logger.info(configUtils.toString());
        Properties commonProp = configUtils.getCommonProp();

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.getConfig().setUseSnapshotCompression(true);
        env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

        Properties consumerProp = configUtils.getConsumerProp();
        if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
                .equalsIgnoreCase("true")) {
            consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
            consumerProp.setProperty("sasl.mechanism", "GSSAPI");
            consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
```

```
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
                .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        DataStream<String> convertData = sourceDataStream.map(new MapFunction<String, String>()
{
            @Override
            public String map(String value) {
                return value + new Date();
            }
        }).uid("convert_data").setParallelism(Integer.valueOf(commonProp.getProperty("convert.forma
t.parallelism")));

        convertData.addSink(new SinkToRedis<String>(commonProp)).uid("sink_redis")
                .setParallelism(Integer.valueOf(commonProp.getProperty("sink.redis.parallelism")));

        env.execute("KafkaFlink1.13.6Redis kerberos or not Example");
    }
}
```

(4) SinkToRedis类文件

```
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import redis.clients.jedis.HostAndPort;
import redis.clients.jedis.JedisCluster;
import redis.clients.jedis.JedisPoolConfig;

import java.util.HashSet;
```

```
import java.util.Properties;
import java.util.Set;

public class SinkToRedis<IN> extends RichSinkFunction<IN> implements CheckpointedFunction {

  private Logger logger = LoggerFactory.getLogger(SinkToRedis.class);

  private String ipPorts;
  private String passWord;
  private JedisCluster cluster;

  SinkToRedis(Properties properties) {
    this.ipPorts = properties.getProperty("sink.redis.ip.ports");
    this.passWord = properties.getProperty("sink.redis.password");
  }

  @Override
  public void open(Configuration parameters) throws Exception {
    Set<HostAndPort> nodes = new HashSet<>();
    String[] ipPorts = this.ipPorts.split(",");
    for (String ipPort : ipPorts) {
      nodes.add(new HostAndPort(ipPort.split(":")[0], Integer.valueOf(ipPort.split(":")[1])));
    }
    this.cluster = new JedisCluster(nodes, 5000, 3000, 10, this.passWord, new JedisPoolConfig());
    logger.info("================open");
  }

  @Override
  public void close() throws Exception {
    this.cluster.close();
  }

  @Override
  public void invoke(IN value, Context context) throws Exception {
    logger.info("================invoke");
    logger.info("result : " + this.cluster.set("kafka_flink_redis", value.toString()));
    logger.info(this.cluster.get("kafka_flink_redis"));
  }
```

```
    @Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception {

    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {

    }


}
```

(5) FlinkJob_Kafka2Redis.properties 配置文件

```
############################ Kafka 配置 #####################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_redis
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_redis
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=false
##################################### redis 配置##################################
sink.redis.ip.ports=10.121.65.53:7001,10.121.65.53:7002
sink.redis.password=CloudOS5#DE3@Redis
convert.format.parallelism=1
sink.redis.parallelism=1
```

5.1.6 Flink DataStream 读 Kafka 写 Hudi 同步 Hive

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据以 Hudi 格式写入到 HDFS 并同步到 Hive。

2. 样例执行主要步骤

说明

本章节的操作命令需分别在 Flink、Kafka 组件对应的安装路径下执行，其中：

- 关于 Flink 的命令示例需执行 cd /usr/hdp/3.0.1.0-187/flink 命令进入 Flink 安装路径后执行。
- 关于 Kafka 的命令示例需执行 cd /usr/hdp/3.0.1.0-187/kafka 命令进入 Kafka 安装路径后执行。

(1) 创建 Kafka 主题

```
./bin/kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 准备数据 schema.avsc 文件

将 schema.avsc 文件上传到 Hdfs 目录(如：hdfs dfs –put schema.avsc /flink)。

文件内容如下：

```
{
 "type" : "record",
 "name" : "record",
 "fields" : [ {
  "name" : "uuid",
  "type" : [ "null", "string" ],
  "default" : null
 }, {
  "name" : "name",
  "type" : [ "null", "string" ],
  "default" : null
 }, {
  "name" : "age",
  "type" : [ "null", "int" ],
  "default" : null
 }, {
  "name" : "ts",
  "type" : [ "null", {
   "type" : "long",
   "logicalType" : "timestamp-millis"
```

```
    } ],
    "default" : null
  }, {
    "name" : "partition1",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "table_name",
    "type" : [ "null", "string" ],
    "default" : null
  }]
}
```

(3) 执行 Flink 作业

```
./bin/flink run -t yarn-per-job -c org.apache.hudi.streamer.HoodieFlinkStreamer
-d ../opt/hudi-flink-bundle_2.11-0.10.0.jar \
--kafka-bootstrap-servers node53:6667 \
--kafka-topic hudi_kafka \
--kafka-group-id group1 \
--checkpoint-interval 10000 \
--target-base-path hdfs:///flink/t_hudi \
--table-type MERGE_ON_READ \
--target-table t_hudi \
--partition-path-field partition1 \
--source-avro-schema-path hdfs:///flink/schema.avsc \
--bucket-assign-num 1 \
--write-task-num 1 \
--compaction-tasks 1 \
--hive-sync-enable \
--hive-sync-skip-ro-suffix \
--compaction-trigger-strategy num_or_time \
--compaction-delta-commits 1 \
--compaction-delta-seconds 5 \
--hive-sync-partition-extractor-class  org.apache.hudi.hive.SlashEncodedDayPartitionValueExtractor
\
--hive-sync-db default \
--hive-sync-table t_hudi \
--hive-sync-mode hms \
--hive-sync-metastore-uris "thrift://node53.hde.com:9083"
```

(4) 生产消息

```
./bin/kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name} < kafka_data.txt
```

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。
- Kafka_data.txt 内容如下：


```
{"uuid": "id1", "name": "Danny", "age": 23, "ts": "1970-01-01T00:00:01", "partition1": "2022/03/01", "table_name":"table2"}
{"uuid": "id2", "name": "Stephen", "age": 33, "ts": "1970-01-01T00:00:02", "partition1": "2022/03/01", "table_name":"table2"}
{"uuid": "id3", "name": "Julian", "age": 53, "ts": "1970-01-01T00:00:03", "partition1": "2022/03/02", "table_name":"table2"}
{"uuid": "id4", "name": "Fabian", "age": 31, "ts": "1970-01-01T00:00:04", "partition1": "2022/03/02", "table_name":"table2"}
{"uuid": "id5", "name": "Sophia", "age": 18, "ts": "1970-01-01T00:00:05", "partition1": "2022/03/03", "table_name":"table2"}
{"uuid": "id6", "name": "Emma", "age": 20, "ts": "1970-01-01T00:00:11", "partition1": "2022/03/03", "table_name":"table2"}
{"uuid": "id7", "name": "Bob", "age": 44, "ts": "1970-01-01T00:00:23", "partition1": "2022/03/04", "table_name":"table2"}
{"uuid": "id8", "name": "Han", "age": 56, "ts": "1970-01-01T00:00:45", "partition1": "2022/03/04", "table_name":"table2"}
```


(5) 查看执行结果

将/usr/hdp/3.0.1.0-187/flink/opt/hudi-hadoop-mr-bundle-0.10.0.jar 拷贝到所有 HiveServer2 所在节点/usr/hdp/3.0.1.0-187/hive/lib/，重启 Hive 服务。

```
beeline
show tables;
select * from t_hudi;
```

5.2 Flink开发程序（Kerberos环境）

说明

- Flink 脚本所在路径/usr/hdp/3.0.1.0-187/flink/bin（即在大数据平台安装的默认路径）。
- Kafka 脚本所在路径/usr/hdp/3.0.1.0-187/kafka/bin（即在大数据平台安装的默认路径）。
- 本章节的最佳实践仅为示例（比如：文件名、文件路径、表名称、主题名称、集群 IP 等），在使用过程中本章节的命令和代码仅供参考，请根据实际情况进行调整。
- 提交 Flink 作业时必须切换至符合权限要求的用户，本章节全部以用户(adminzyf)示例提交 Flink 作业。

执行本章节操作之前，需提前满足以下条件：

(1) 提前准备用户（示例：adminzyf），也可以使用集群的超级用户。

(2) 下载准备用户对应的 keytab 文件（示例：adminzyf.keytab）。

(3) keytab 文件（示例：adminzyf.keytab）必须在集群每个节点都存在，且要求路径相同、权限相同（示例命令：chown adminzyf:adminzyf /opt/adminzyf.keytab）。

5.2.1 Flink DataStream 读 Kafka 写 Kafka

1. 样例主要功能

从 Kafka 一个 topic 中读取数据进行 wordcount 之后，将结果写入到一个新的 topic 中。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic
${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 修改 Flink 配置

通过 Flink 配置界面修改如下配置项、保存、重启

```
security.kerberos.login.keytab: /opt/adminzyf.keytab
security.kerberos.login.principal: adminzyf
```

(3) 执行 Flink 作业（切换至符合权限要求的用户，本章节以 adminzyf 用户示例）

```
su adminzyf
flink run -t yarn-per-job -d ${jar path }/KafkaFlink1.13.6Kafka-1.0.jar --path
FlinkJob_Kafka2Kafka_Ker.properties
```

其中：

- ${jar path}表示样例 jar 包路径。

(4) 生产消息

```
./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name}
--producer-property security.protocol=SASL_PLAINTEXT
```

消息格式：name ss

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(5) 消费消息

```
./kafka-console-consumer.sh –bootstrap-server ${bootstrap_server_list} --topic ${topic_name}
--consumer-property security.protocol=SASL_PLAINTEXT
```

运行结果为：(name,1) (ss,1)

其中：

- ${bootstrap_server_list}表示 Kafka 集群中 Kafka Broker 地址，端口号默认 6667，多个 Kafka Broker 节点用逗号分隔。

3. 样例代码

(1) Pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>demo</artifactId>
    <groupId>com.zyf</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>KafkaFlink1.13.6Kafka</artifactId>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <flink.version>1.13.6</flink.version>
    <scala.binary.version>2.11</scala.binary.version>
  </properties>
```

```
<dependencies>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.0.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <artifactSet>
              <excludes>
                <exclude>com.google.code.findbugs:jsr305</exclude>
                <exclude>org.slf4j:*</exclude>
                <exclude>log4j:*</exclude>
```

```
                    </excludes>
                  </artifactSet>
                  <filters>
                    <filter>
                      <!-- Do not copy the signatures in the META-INF folder.
                      Otherwise, this might cause SecurityExceptions when using the JAR. -->
                      <artifact>*:*</artifact>
                      <excludes>
                        <exclude>META-INF/*.SF</exclude>
                        <exclude>META-INF/*.DSA</exclude>
                        <exclude>META-INF/*.RSA</exclude>
                      </excludes>
                    </filter>
                  </filters>
                  <transformers>
                    <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                      <mainClass>KafkaFlinkKafka</mainClass>
                    </transformer>
                  </transformers>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>

</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

  private Properties producerProp = new Properties();
  private Properties consumerProp = new Properties();
  private Properties commonProp = new Properties();
```

```
public ConfigUtils(String KafkaConfigPath) {
   try {
      Properties prop = new Properties();
      FileInputStream in = new FileInputStream(KafkaConfigPath);
      prop.load(in);
      in.close();

      String keyPrefix;
      String keyValue;
      for (String key : prop.stringPropertyNames()) {
         keyPrefix = key.trim().split("\\.")[0];
         keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
         switch (keyPrefix.toLowerCase()) {
            case "producer":
               this.producerProp.put(keyValue, prop.getProperty(key));
               break;
            case "consumer":
               this.consumerProp.put(keyValue, prop.getProperty(key));
               break;
            default:
               this.commonProp.put(key, prop.getProperty(key));
               break;
         }
      }
   } catch (Exception ex) {
      ex.printStackTrace();
   }
}

public Properties getProducerProp() {
   return producerProp;
}

public void setProducerProp(Properties producerProp) {
   this.producerProp = producerProp;
}
```

```
    public Properties getConsumerProp() {
        return consumerProp;
    }

    public void setConsumerProp(Properties consumerProp) {
        this.consumerProp = consumerProp;
    }

    public Properties getCommonProp() {
        return commonProp;
    }

    public void setCommonProp(Properties commonProp) {
        this.commonProp = commonProp;
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) KafkaFlinkKafka 类文件

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.utils.MultipleParameterTool;
```

```
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;
import org.apache.flink.util.Collector;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;

public class KafkaFlinkKafka {
  public static void main(String[] args) throws Exception {
    Logger logger = LoggerFactory.getLogger(KafkaFlinkKafka.class);

    final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
    if (!multipleParameterTool.has("path")) {
      System.out.println("Error: not exist --path /opt/your.properties");
      System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
      System.exit(0);
    }
    ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
    logger.info(configUtils.toString());
    Properties commonProp = configUtils.getCommonProp();

    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setUseSnapshotCompression(true);
    env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

    Properties consumerProp = configUtils.getConsumerProp();
    if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
        .equalsIgnoreCase("true")) {
      consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
```

```
            consumerProp.setProperty("sasl.mechanism", "GSSAPI");
            consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        Properties producerProp = configUtils.getProducerProp();
        if (commonProp.containsKey("sink.kafka.security.enable") &&
commonProp.getProperty("sink.kafka.security.enable")
                .equalsIgnoreCase("true")) {
            producerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
            producerProp.setProperty("sasl.mechanism", "GSSAPI");
            producerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }

        FlinkKafkaProducer<Tuple2<String, Integer>> kafkaProducer = new
FlinkKafkaProducer<>(commonProp.getProperty("sink.kafka.topic"),
                new MySchema(), producerProp);

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
                .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        DataStream<Tuple2<String, Integer>> counts = sourceDataStream.flatMap(new
Tokenizer()).keyBy(new MyKeySelector()).sum(1);


counts.addSink(kafkaProducer).uid("sink_kafka").setParallelism(Integer.valueOf(commonProp.getPr
operty("sink.kafka.parallelism")));

        env.execute("KafkaFlink1.13.6KafkaKerOrNot");
    }
```

```
  private static final class Tokenizer implements FlatMapFunction<String, Tuple2<String, Integer>> {

    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split("\\W+");

      // emit the pairs
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(new Tuple2<>(token, 1));
        }
      }
    }
  }

  private static final class MyKeySelector implements KeySelector<Tuple2<String, Integer>, String> {
    @Override
    public String getKey(Tuple2<String, Integer> value) throws Exception {
      return value.f0;
    }
  }
}
```

(4) MySchema 类文件

```
import org.apache.flink.api.common.serialization.DeserializationSchema;
import org.apache.flink.api.common.serialization.SerializationSchema;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.tuple.Tuple2;

import java.io.IOException;

public class MySchema implements DeserializationSchema<Tuple2<String, Integer>>,
SerializationSchema<Tuple2<String, Integer>> {
  @Override
  public Tuple2<String, Integer> deserialize(byte[] message) throws IOException {
    return null;
  }
```

```
    @Override
    public boolean isEndOfStream(Tuple2<String, Integer> nextElement) {
        return false;
    }

    @Override
    public byte[] serialize(Tuple2<String, Integer> element) {
        return element.toString().getBytes();
    }

    @Override
    public TypeInformation<Tuple2<String, Integer>> getProducedType() {
        return null;
    }
}
```

(5) FlinkJob_Kafka2Kafka_Ker.properties 配置文件

```
############################## Kafka 配置 #########################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_kafka
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka Producer 专用配置，支持所有原生配置
producer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
producer.acks=1
producer.compression.type=snappy
#Kafka 其他配置
#source
source.kafka.topic=in
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=true
#sink
sink.kafka.topic=out
```

sink.kafka.parallelism=1

sink.kafka.security.enable=true

5.2.2 Flink DataStream 读 Kafka 写 Elasticsearch

1. 样例主要功能

从 Kafka 一个 topic 读取数据，写入到 Elasticsearch。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 修改 Flink 配置

通过 Flink 配置界面修改如下配置项、保存、重启

security.kerberos.login.keytab: /opt/adminzyf.keytab

security.kerberos.login.principal: adminzyf

(3) 执行 Flink 作业（切换至符合权限要求的用户，本章节以 adminzyf 用户示例）

su adminzyf

flink run -t yarn-per-job -d ${jar path}/KafkaFlink1.13.6Es-1.0.jar --path FlinkJob_Kafka2ES_Ker.properties

其中：

- ${jar path}表示样例 jar 包路径。

(4) 生产消息

./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name} --producer-property security.protocol=SASL_PLAINTEXT

消息格式：name

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(5) 查看执行结果

curl -XGET --negotiate -u adminzyf:admin@123 'http://zyf53:9200/kafka-flink-es-index/_search?pretty' -H 'Content-Type:application/json'

3. 样例代码

(1) Pom 文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>demo</artifactId>
        <groupId>com.zyf</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>KafkaFlink1.13.6Es</artifactId>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <flink.version>1.13.6</flink.version>
        <scala.binary.version>2.11</scala.binary.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-clients_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
        </dependency>
```

```
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-elasticsearch7_2.11</artifactId>
      <version>${flink.version}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.0.0</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <artifactSet>
                <excludes>
                  <exclude>com.google.code.findbugs:jsr305</exclude>
                  <exclude>org.slf4j:*</exclude>
                  <exclude>log4j:*</exclude>
                </excludes>
              </artifactSet>
              <filters>
                <filter>
                  <!-- Do not copy the signatures in the META-INF folder.
                  Otherwise, this might cause SecurityExceptions when using the JAR. -->
                  <artifact>*:*</artifact>
                  <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
                  </excludes>
                </filter>
```

```
                    </filters>
                    <transformers>
                        <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                            <mainClass>KafkaFlinkES</mainClass>
                        </transformer>
                    </transformers>
                </configuration>
            </execution>
        </executions>
    </plugin>
  </plugins>
</build>

</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

  private Properties producerProp = new Properties();
  private Properties consumerProp = new Properties();
  private Properties commonProp = new Properties();

  public ConfigUtils(String KafkaConfigPath) {
    try {
      Properties prop = new Properties();
      FileInputStream in = new FileInputStream(KafkaConfigPath);
      prop.load(in);
      in.close();

      String keyPrefix;
      String keyValue;
      for (String key : prop.stringPropertyNames()) {
        keyPrefix = key.trim().split("\\.")[0];
        keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
```

```
            switch (keyPrefix.toLowerCase()) {
              case "producer":
                this.producerProp.put(keyValue, prop.getProperty(key));
                break;
              case "consumer":
                this.consumerProp.put(keyValue, prop.getProperty(key));
                break;
              default:
                this.commonProp.put(key, prop.getProperty(key));
                break;
            }
          }
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }

      public Properties getProducerProp() {
        return producerProp;
      }

      public void setProducerProp(Properties producerProp) {
        this.producerProp = producerProp;
      }

      public Properties getConsumerProp() {
        return consumerProp;
      }

      public void setConsumerProp(Properties consumerProp) {
        this.consumerProp = consumerProp;
      }

      public Properties getCommonProp() {
        return commonProp;
      }

      public void setCommonProp(Properties commonProp) {
```

```
        this.commonProp = commonProp;
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) KafkaFlinkES 类文件

```
import org.apache.flink.api.common.functions.RuntimeContext;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.elasticsearch.ElasticsearchSinkFunction;
import org.apache.flink.streaming.connectors.elasticsearch.RequestIndexer;
import
org.apache.flink.streaming.connectors.elasticsearch.util.RetryRejectedExecutionFailureHandler;
import org.apache.flink.streaming.connectors.elasticsearch7.ElasticsearchSink;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.http.HttpHost;
import org.elasticsearch.action.index.IndexRequest;
import org.elasticsearch.client.Requests;
import org.elasticsearch.common.settings.SecureString;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Properties;

public class KafkaFlinkES {
  public static void main(String[] args) throws Exception {
    Logger logger = LoggerFactory.getLogger(KafkaFlinkES.class);

    final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
    if (!multipleParameterTool.has("path")) {
      System.out.println("Error: not exist --path /opt/your.properties");
      System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
      System.exit(0);
    }
    ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
    logger.info(configUtils.toString());
    Properties commonProp = configUtils.getCommonProp();

    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setUseSnapshotCompression(true);
    env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

    Properties consumerProp = configUtils.getConsumerProp();
    if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
        .equalsIgnoreCase("true")) {
      consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
      consumerProp.setProperty("sasl.mechanism", "GSSAPI");
      consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
    }

    FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
```

```
        new SimpleStringSchema(), consumerProp);
    kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
    if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
        .equalsIgnoreCase("latest")) {
      kafkaConsumer.setStartFromLatest();
    }

    DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
        .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

    List<HttpHost> httpHosts = new ArrayList<>();
    for (String ipPort : commonProp.getProperty("sink.es.servers").trim().split(",")) {
      String[] hostPort = ipPort.trim().split(":");
      httpHosts.add(new HttpHost(hostPort[0].trim(), Integer.valueOf(hostPort[1].trim()), "http"));
    }

    // use a ElasticsearchSink.Builder to create an ElasticsearchSink
    ElasticsearchSink.Builder<String> esSinkBuilder = new ElasticsearchSink.Builder<>(
        httpHosts,
        new ElasticsearchSinkFunction<String>() {
          public IndexRequest createIndexRequest(String element) {
            HashMap<String, String> json = new HashMap<>();
            json.put("data", element);

            return Requests.indexRequest()
                .index(commonProp.getProperty("sink.es.index"))
                .type(commonProp.getProperty("sink.es.type"))
                .source(json);
          }

          @Override
          public void process(String element, RuntimeContext runtimeContext, RequestIndexer
requestIndexer) {
            requestIndexer.add(createIndexRequest(element));
          }
        }
    );
```

```
        // configuration for the bulk requests; this instructs the sink to emit after every element, otherwise they would be buffered

esSinkBuilder.setBulkFlushMaxActions(Integer.valueOf(commonProp.getProperty("sink.es.bulk.flush.max.actions")));

esSinkBuilder.setBulkFlushInterval(Long.valueOf(commonProp.getProperty("sink.es.bulk.flush.interval.ms")));

esSinkBuilder.setBulkFlushMaxSizeMb(Integer.valueOf(commonProp.getProperty("sink.es.bulk.flush.max.size.mb")));
        esSinkBuilder.setFailureHandler(new RetryRejectedExecutionFailureHandler());

        if (commonProp.getProperty("sink.es.security.enable").equalsIgnoreCase("true")) {
            esSinkBuilder.setRestClientFactory(restClientBuilder -> {
                if (commonProp.getProperty("sink.es.usr.pwd.enable").equalsIgnoreCase("true")) {
                    restClientBuilder.setHttpClientConfigCallback(new
SpnegoHttpClientConfigCallbackHandler(commonProp.getProperty("sink.es.usr"),
                            new SecureString(commonProp.getProperty("sink.es.pwd")), false));
                } else {
                    restClientBuilder.setHttpClientConfigCallback(new
SpnegoHttpClientConfigCallbackHandler(commonProp.getProperty("sink.es.usr"),
                            commonProp.getProperty("sink.es.keytab.path"), false));
                }

            });
        }


sourceDataStream.addSink(esSinkBuilder.build()).uid("sink_es").setParallelism(Integer.valueOf(commonProp.getProperty("sink.es.parallelism")));

        env.execute("KafkaFlink1.13.6ESKerOrNot");
    }
}
```

(4) SpnegoHttpClientConfigCallbackHandler 类文件

```
import org.apache.http.auth.AuthSchemeProvider;
```

```
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.Credentials;
import org.apache.http.auth.KerberosCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.client.config.AuthSchemes;
import org.apache.http.config.Lookup;
import org.apache.http.config.RegistryBuilder;
import org.apache.http.impl.auth.SPNegoSchemeFactory;
import org.apache.http.impl.nio.client.HttpAsyncClientBuilder;
import org.elasticsearch.ExceptionsHelper;
import org.elasticsearch.client.RestClientBuilder.HttpClientConfigCallback;
import org.elasticsearch.common.settings.SecureString;
import org.ietf.jgss.*;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.kerberos.KerberosPrincipal;
import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.Configuration;
import javax.security.auth.login.LoginContext;
import java.io.IOException;
import java.security.AccessControlContext;
import java.security.AccessController;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class SpnegoHttpClientConfigCallbackHandler implements HttpClientConfigCallback {
  private static final String SUN_KRB5_LOGIN_MODULE =
"com.sun.security.auth.module.Krb5LoginModule";
  private static final String CRED_CONF_NAME = "ESClientLoginConf";
  private static final Oid SPNEGO_OID = getSpnegoOid();
```

```
    private static Oid getSpnegoOid() {
        Oid oid = null;
        try {
            oid = new Oid("1.3.6.1.5.5.2");
        } catch (GSSException gsse) {
            throw ExceptionsHelper.convertToRuntime(gsse);
        }
        return oid;
    }

    private final String userPrincipalName;
    private final SecureString password;
    private final String keytabPath;
    private final boolean enableDebugLogs;
    private LoginContext loginContext;

    /**
     * Constructs {@link SpnegoHttpClientConfigCallbackHandler} with given
     * principalName and password.
     *
     * @param userPrincipalName user principal name
     * @param password          password for user
     * @param enableDebugLogs   if {@code true} enables kerberos debug logs
     */
    public SpnegoHttpClientConfigCallbackHandler(final String userPrincipalName, final SecureString
password,
                                    final boolean enableDebugLogs) {
        this.userPrincipalName = userPrincipalName;
        this.password = password;
        this.keytabPath = null;
        this.enableDebugLogs = enableDebugLogs;
    }

    /**
     * Constructs {@link SpnegoHttpClientConfigCallbackHandler} with given
     * principalName and keytab.
     *
     * @param userPrincipalName User principal name
```

```
     * @param keytabPath       path to keytab file for user
     * @param enableDebugLogs   if {@code true} enables kerberos debug logs
     */
    public SpnegoHttpClientConfigCallbackHandler(final String userPrincipalName, final String
keytabPath, final boolean enableDebugLogs) {
        this.userPrincipalName = userPrincipalName;
        this.keytabPath = keytabPath;
        this.password = null;
        this.enableDebugLogs = enableDebugLogs;
    }

    @Override
    public HttpAsyncClientBuilder customizeHttpClient(HttpAsyncClientBuilder httpClientBuilder) {
        setupSpnegoAuthSchemeSupport(httpClientBuilder);
        return httpClientBuilder;
    }

    private void setupSpnegoAuthSchemeSupport(HttpAsyncClientBuilder httpClientBuilder) {
        final Lookup<AuthSchemeProvider> authSchemeRegistry =
RegistryBuilder.<AuthSchemeProvider>create()
                .register(AuthSchemes.SPNEGO, new SPNegoSchemeFactory()).build();

        final GSSManager gssManager = GSSManager.getInstance();
        try {
            final GSSName gssUserPrincipalName = gssManager.createName(userPrincipalName,
GSSName.NT_USER_NAME);
            login();
            final AccessControlContext acc = AccessController.getContext();
            final GSSCredential credential = doAsPrivilegedWrapper(loginContext.getSubject(),
                    (PrivilegedExceptionAction<GSSCredential>) () ->
gssManager.createCredential(gssUserPrincipalName,
                            GSSCredential.DEFAULT_LIFETIME, SPNEGO_OID, GSSCredential.INITIATE_ONLY),
                    acc);

            final KerberosCredentialsProvider credentialsProvider = new KerberosCredentialsProvider();
            credentialsProvider.setCredentials(
                    new AuthScope(AuthScope.ANY_HOST, AuthScope.ANY_PORT, AuthScope.ANY_REALM,
AuthSchemes.SPNEGO),
                    new KerberosCredentials(credential));
```

```
            httpClientBuilder.setDefaultCredentialsProvider(credentialsProvider);
        } catch (GSSException e) {
            throw new RuntimeException(e);
        } catch (PrivilegedActionException e) {
            throw new RuntimeException(e.getCause());
        }
        httpClientBuilder.setDefaultAuthSchemeRegistry(authSchemeRegistry);
    }

    /**
     * If logged in {@link LoginContext} is not available, it attempts login and
     * returns {@link LoginContext}
     *
     * @return {@link LoginContext}
     * @throws PrivilegedActionException
     */
    public synchronized LoginContext login() throws PrivilegedActionException {
        if (this.loginContext == null) {
            AccessController.doPrivileged((PrivilegedExceptionAction<Void>) () -> {
                final Subject subject = new Subject(false, Collections.singleton(new
KerberosPrincipal(userPrincipalName)),
                        Collections.emptySet(), Collections.emptySet());
                Configuration conf = null;
                final CallbackHandler callback;
                if (password != null) {
                    conf = new PasswordJaasConf(userPrincipalName, enableDebugLogs);
                    callback = new KrbCallbackHandler(userPrincipalName, password);
                } else {
                    conf = new KeytabJaasConf(userPrincipalName, keytabPath, enableDebugLogs);
                    callback = null;
                }
                loginContext = new LoginContext(CRED_CONF_NAME, subject, callback, conf);
                loginContext.login();
                return null;
            });
        }
        return loginContext;
    }
```

```
  /**
   * Privileged Wrapper that invokes action with Subject.doAs to perform work as
   * given subject.
   *
   * @param subject {@link Subject} to be used for this work
   * @param action  {@link PrivilegedExceptionAction} action for performing inside
   *             Subject.doAs
   * @param acc     the {@link AccessControlContext} to be tied to the specified
   *             subject and action see
   *             {@link Subject#doAsPrivileged(Subject, PrivilegedExceptionAction,
AccessControlContext)
   * @return the value returned by the PrivilegedExceptionAction's run method
   * @throws PrivilegedActionException
   */
  static <T> T doAsPrivilegedWrapper(final Subject subject, final PrivilegedExceptionAction<T>
action, final AccessControlContext acc)
        throws PrivilegedActionException {
      try {
        return AccessController.doPrivileged((PrivilegedExceptionAction<T>) () ->
Subject.doAsPrivileged(subject, action, acc));
      } catch (PrivilegedActionException pae) {
        if (pae.getCause() instanceof PrivilegedActionException) {
          throw (PrivilegedActionException) pae.getCause();
        }
        throw pae;
      }
  }

  /**
   * This class matches {@link AuthScope} and based on that returns
   * {@link Credentials}. Only supports {@link AuthSchemes#SPNEGO} in
   * {@link AuthScope#getScheme()}
   */
  private static class KerberosCredentialsProvider implements CredentialsProvider {
      private AuthScope authScope;
      private Credentials credentials;
```

```
    @Override
    public void setCredentials(AuthScope authscope, Credentials credentials) {
      if (authscope.getScheme().regionMatches(true, 0, AuthSchemes.SPNEGO, 0,
AuthSchemes.SPNEGO.length()) == false) {
        throw new IllegalArgumentException("Only " + AuthSchemes.SPNEGO + " auth scheme is
supported in AuthScope");
      }
      this.authScope = authscope;
      this.credentials = credentials;
    }

    @Override
    public Credentials getCredentials(AuthScope authscope) {
      assert this.authScope != null && authscope != null;
      return authscope.match(this.authScope) > -1 ? this.credentials : null;
    }

    @Override
    public void clear() {
      this.authScope = null;
      this.credentials = null;
    }
  }

  /**
   * Jaas call back handler to provide credentials.
   */
  private static class KrbCallbackHandler implements CallbackHandler {
    private final String principal;
    private final SecureString password;

    KrbCallbackHandler(final String principal, final SecureString password) {
      this.principal = principal;
      this.password = password;
    }

    public void handle(final Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
```

```
        for (Callback callback : callbacks) {
          if (callback instanceof PasswordCallback) {
            PasswordCallback pc = (PasswordCallback) callback;
            if (pc.getPrompt().contains(principal)) {
              pc.setPassword(password.getChars());
              break;
            }
          }
        }
    }
}

/**
 * Usually we would have a JAAS configuration file for login configuration.
 * Instead of an additional file setting as we do not want the options to be
 * customizable we are constructing it in memory.
 * <p>
 * As we are using this instead of jaas.conf, this requires refresh of
 * {@link Configuration} and reqires appropriate security permissions to do so.
 */
private static class PasswordJaasConf extends AbstractJaasConf {

    PasswordJaasConf(final String userPrincipalName, final boolean enableDebugLogs) {
        super(userPrincipalName, enableDebugLogs);
    }

    public void addOptions(final Map<String, String> options) {
        options.put("useTicketCache", Boolean.FALSE.toString());
        options.put("useKeyTab", Boolean.FALSE.toString());
    }
}

/**
 * Usually we would have a JAAS configuration file for login configuration. As
 * we have static configuration except debug flag, we are constructing in
 * memory. This avoids additional configuration required from the user.
 * <p>
 * As we are using this instead of jaas.conf, this requires refresh of
```

```
 * {@link Configuration} and requires appropriate security permissions to do so.
 */
private static class KeytabJaasConf extends AbstractJaasConf {
    private final String keytabFilePath;

    KeytabJaasConf(final String userPrincipalName, final String keytabFilePath, final boolean
enableDebugLogs) {
        super(userPrincipalName, enableDebugLogs);
        this.keytabFilePath = keytabFilePath;
    }

    public void addOptions(final Map<String, String> options) {
        options.put("useKeyTab", Boolean.TRUE.toString());
        options.put("keyTab", keytabFilePath);
        options.put("doNotPrompt", Boolean.TRUE.toString());
    }

}

private abstract static class AbstractJaasConf extends Configuration {
    private final String userPrincipalName;
    private final boolean enableDebugLogs;

    AbstractJaasConf(final String userPrincipalName, final boolean enableDebugLogs) {
        this.userPrincipalName = userPrincipalName;
        this.enableDebugLogs = enableDebugLogs;
    }

    @Override
    public AppConfigurationEntry[] getAppConfigurationEntry(final String name) {
        final Map<String, String> options = new HashMap<>();
        options.put("principal", userPrincipalName);
        options.put("isInitiator", Boolean.TRUE.toString());
        options.put("storeKey", Boolean.TRUE.toString());
        options.put("debug", Boolean.toString(enableDebugLogs));
        addOptions(options);
        return new AppConfigurationEntry[]{new
AppConfigurationEntry(SUN_KRB5_LOGIN_MODULE,
```

```
                AppConfigurationEntry.LoginModuleControlFlag.REQUIRED,
Collections.unmodifiableMap(options))};
        }

        abstract void addOptions(Map<String, String> options);
    }
}
```

(5) FlinkJob_Kafka2ES_Ker.properties 配置文件

```
############################ Kafka 配置 #####################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_es
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_es
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=true
###################################### ES 配置################################
sink.es.servers=10.121.65.53:9200,10.121.65.54:9200,10.121.65.55:9200
sink.es.index=kafka-flink-es-index
sink.es.type=kafka-flink-es-type
sink.es.bulk.flush.max.actions=1
sink.es.bulk.flush.max.size.mb=50
sink.es.bulk.flush.interval.ms=1000
sink.es.parallelism=1
#kerberos 相关
sink.es.security.enable=true
sink.es.usr.pwd.enable=true
sink.es.usr=adminzyf
sink.es.pwd=admin@123
sink.es.keytab.path=/opt/adminzyf.keytab
```

5.2.3 Flink DataStream 读 Kafka 写 HBase

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据写入到 HBase 的一个表中。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称

(2) 修改 Flink 配置

通过 Flink 配置界面修改如下配置项、保存、重启

security.kerberos.login.keytab: /opt/adminzyf.keytab

security.kerberos.login.principal: adminzyf

(3) 创建 Hbase 表

su adminzyf

登录 hbase shell

create 'test','cf'

(4) 执行 Flink 作业（切换至符合权限要求的用户，本章节以 adminzyf 用户示例）

su adminzyf

flink run -m yarn-cluster -d ${jar path}/KafkaFlink1.13.6Hbase-1.0.jar --path FlinkJob_Kafka2Hbase_ker.properties

其中：

- ${jar path}表示样例 jar 包路径。

(5) 生产消息

./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name} --producer-property security.protocol=SASL_PLAINTEXT

消息格式：Tom

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(6) 查看执行结果

登录 hbase shell

scan 'test’

3. 样例代码

(1) Pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>demo</artifactId>
        <groupId>com.zyf</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>KafkaFlink1.13.6Hbase</artifactId>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <flink.version>1.13.6</flink.version>
        <scala.binary.version>2.11</scala.binary.version>
        <hbase.version>2.1.0-cdh6.2.0</hbase.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
             <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
```

```
        <artifactId>flink-clients_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-server</artifactId>
        <version>${hbase.version}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <artifactSet>
                            <excludes>
                                <exclude>com.google.code.findbugs:jsr305</exclude>
                                <exclude>org.slf4j:*</exclude>
                                <exclude>log4j:*</exclude>
                            </excludes>
                        </artifactSet>
                        <filters>
                            <filter>
                                <!-- Do not copy the signatures in the META-INF folder.
                                Otherwise, this might cause SecurityExceptions when using the JAR. -->
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
```

```
                        <exclude>META-INF/*.RSA</exclude>
                    </excludes>
                </filter>
            </filters>
            <transformers>
                <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                    <mainClass>KafkaFlinkHbase</mainClass>
                </transformer>
            </transformers>
        </configuration>
    </execution>
  </executions>
 </plugin>
 </plugins>
</build>

</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

  private Properties producerProp = new Properties();
  private Properties consumerProp = new Properties();
  private Properties commonProp = new Properties();

  public ConfigUtils(String KafkaConfigPath) {
    try {
      Properties prop = new Properties();
      FileInputStream in = new FileInputStream(KafkaConfigPath);
      prop.load(in);
      in.close();

      String keyPrefix;
      String keyValue;
```

```
        for (String key : prop.stringPropertyNames()) {
          keyPrefix = key.trim().split("\\.")[0];
          keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
          switch (keyPrefix.toLowerCase()) {
            case "producer":
              this.producerProp.put(keyValue, prop.getProperty(key));
              break;
            case "consumer":
              this.consumerProp.put(keyValue, prop.getProperty(key));
              break;
            default:
              this.commonProp.put(key, prop.getProperty(key));
              break;
          }
        }
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }

    public Properties getProducerProp() {
      return producerProp;
    }

    public void setProducerProp(Properties producerProp) {
      this.producerProp = producerProp;
    }

    public Properties getConsumerProp() {
      return consumerProp;
    }

    public void setConsumerProp(Properties consumerProp) {
      this.consumerProp = consumerProp;
    }

    public Properties getCommonProp() {
      return commonProp;
```

```
    }

    public void setCommonProp(Properties commonProp) {
        this.commonProp = commonProp;
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) KafkaFlinkHbase 类文件

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class KafkaFlinkHbase {
```

```
    public static void main(String[] args) throws Exception {
        Logger logger = LoggerFactory.getLogger(KafkaFlinkHbase.class);

        final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
        if (!multipleParameterTool.has("path")) {
            System.out.println("Error: not exist --path /opt/your.properties");
            System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
            System.exit(0);
        }
        ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
        logger.info(configUtils.toString());
        Properties commonProp = configUtils.getCommonProp();

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.getConfig().setUseSnapshotCompression(true);
        env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

        Properties consumerProp = configUtils.getConsumerProp();
        if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
                .equalsIgnoreCase("true")) {
            consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
            consumerProp.setProperty("sasl.mechanism", "GSSAPI");
            consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }
```

```
        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
            .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        DataStream<Map<Object, Object>> convertData = sourceDataStream.map(new
MapFunction<String, Map<Object, Object>>() {
          @Override
          public Map<Object, Object> map(String value) {
            Map<Object, Object> dataMap = new HashMap<>();
            dataMap.put("rowKey", "defaultRowKey" + System.currentTimeMillis());
            dataMap.put("columnFamily", "cf");
            dataMap.put("cfQualifier", "cfq");
            dataMap.put("data", value);
            return dataMap;
          }
        }).uid("convert_data").setParallelism(Integer.valueOf(commonProp.getProperty("convert.forma
t.parallelism")));

        convertData.addSink(new SinkToHbase<>(commonProp)).uid("sink_hbase")
            .setParallelism(Integer.valueOf(commonProp.getProperty("sink.hbase.parallelism")));

        env.execute("KafkaFlink1.13.6Hbase kerberos or not");

    }
}
```

(4) SinkToHbase 类文件

```
import org.apache.flink.annotation.Internal;
import org.apache.flink.api.common.functions.RuntimeContext;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
```

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.HashMap;
import java.util.Properties;

public class SinkToHbase<IN> extends RichSinkFunction<IN> implements CheckpointedFunction {
  private Logger logger = LoggerFactory.getLogger(SinkToHbase.class);

  private static final long serialVersionUID = 1L;

  private String hbaseConfPath;

  private Connection conn;

  private BufferedMutator mutator;

  private String tableName;

  private long writeBufferSize;


  SinkToHbase(Properties properties) {
    this.hbaseConfPath = properties.getProperty("sink.hbase.conf.path");
    this.writeBufferSize = Long.valueOf(properties.getProperty("sink.hbase.client.write.buffer"));

    this.tableName = properties.getProperty("sink.hbase.table.name");

  }

  @Override
  public void open(Configuration parameters) throws Exception {
    RuntimeContext ctx = getRuntimeContext();
    org.apache.hadoop.conf.Configuration config = HBaseConfiguration.create();
    config.addResource(new Path(hbaseConfPath));
    conn = ConnectionFactory.createConnection(config);
```

```
        BufferedMutatorParams bmParams = new
BufferedMutatorParams(TableName.valueOf(tableName));

        bmParams.writeBufferSize(this.writeBufferSize);

        mutator = conn.getBufferedMutator(bmParams);

        logger.info("Starting SinkToHbase ({}/{}) to produce into table {}",
                ctx.getIndexOfThisSubtask() + 1, ctx.getNumberOfParallelSubtasks(), this.tableName);
    }


    @Override
    public void invoke(IN value, Context context) throws Exception {
        if (value != null) {
            HashMap<String, String> dataMap = (HashMap<String, String>) value;
            mutator.mutate(new Put(Bytes.toBytes(dataMap.get("rowKey")))
                    .addColumn(Bytes.toBytes(dataMap.get("columnFamily")),
                            Bytes.toBytes(dataMap.get("cfQualifier")),
                            Bytes.toBytes(dataMap.get("data"))));
        }
    }

    @Override
    public void close() throws Exception {
        mutator.close();
        conn.close();
    }

    @Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception {

    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {

    }
```

}

(5) FlinkJob_Kafka2Hbase_Ker.properties 配置文件

```
############################## Kafka 配置 ########################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_hbase
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_hbase
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=true
######################################## Hbase 配置##################################
sink.hbase.conf.path=/etc/hbase/conf/hbase-site.xml
sink.hbase.client.write.buffer=1
sink.hbase.table.name=test
convert.format.parallelism=1
sink.hbase.parallelism=1
```

5.2.4 Flink DataStream 读 Kafka 写 Hive

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据写入到 Hive 的一个表中。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic
${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称

(2) 创建 Hive 表

```
create table IF NOT EXISTS hc_table(id int,name string)ROW FORMAT DELIMITED FIELDS
TERMINATED BY ',' STORED AS TEXTFILE;
```

(3) 修改 Flink 配置

通过 Flink 配置界面修改如下配置项、保存、重启

security.kerberos.login.keytab: /opt/adminzyf.keytab

security.kerberos.login.principal: adminzyf

(4) 执行 Flink 作业（切换至符合权限要求的用户，本章节以 adminzyf 用户示例）

su adminzyf

flink run -m yarn-cluster -d /opt/KafkaFlink1.13.6HiveSink-1.0.jar --path /opt/FlinkJob_Kafka2Hive_Ker.properties

(5) 生产消息

./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name} –producer-property security.protocol=SASL_PLAINTEXT

消息格式：Tom

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(6) 查看执行结果

beeline

select * from hc_table;

3. 样例代码

(1) Pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <parent>
      <artifactId>demo</artifactId>
      <groupId>com.zyf</groupId>
      <version>1.0</version>
   </parent>
   <modelVersion>4.0.0</modelVersion>

   <artifactId>KafkaFlink1.13.6HiveSink</artifactId>

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <flink.version>1.13.6</flink.version>
```

```xml
    <scala.binary.version>2.11</scala.binary.version>
    <hive.version>2.1.1-cdh6.2.0</hive.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-clients_2.11</artifactId>
        <version>${flink.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
         <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-jdbc</artifactId>
        <version>${hive.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-filesystem_2.11</artifactId>
        <version>${flink.version}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
```

```xml
<artifactId>maven-shade-plugin</artifactId>
<version>3.0.0</version>
<executions>
  <execution>
    <phase>package</phase>
    <goals>
      <goal>shade</goal>
    </goals>
    <configuration>
      <artifactSet>
        <excludes>
          <exclude>com.google.code.findbugs:jsr305</exclude>
          <exclude>org.slf4j:*</exclude>
          <exclude>log4j:*</exclude>
        </excludes>
      </artifactSet>
      <filters>
        <filter>
          <!-- Do not copy the signatures in the META-INF folder.
          Otherwise, this might cause SecurityExceptions when using the JAR. -->
          <artifact>*:*</artifact>
          <excludes>
            <exclude>META-INF/*.SF</exclude>
            <exclude>META-INF/*.DSA</exclude>
            <exclude>META-INF/*.RSA</exclude>
          </excludes>
        </filter>
      </filters>
      <transformers>
        <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
          <mainClass>kafkaFlinkHive</mainClass>
        </transformer>
      </transformers>
    </configuration>
  </execution>
</executions>
```

```
        </plugin>
      </plugins>
    </build>

</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

    private Properties producerProp = new Properties();
    private Properties consumerProp = new Properties();
    private Properties commonProp = new Properties();

    public ConfigUtils(String KafkaConfigPath) {
        try {
            Properties prop = new Properties();
            FileInputStream in = new FileInputStream(KafkaConfigPath);
            prop.load(in);
            in.close();

            String keyPrefix;
            String keyValue;
            for (String key : prop.stringPropertyNames()) {
                keyPrefix = key.trim().split("\\.")[0];
                keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
                switch (keyPrefix.toLowerCase()) {
                    case "producer":
                        this.producerProp.put(keyValue, prop.getProperty(key));
                        break;
                    case "consumer":
                        this.consumerProp.put(keyValue, prop.getProperty(key));
                        break;
                    default:
                        this.commonProp.put(key, prop.getProperty(key));
                        break;
                }
```

```
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public Properties getProducerProp() {
    return producerProp;
}

public void setProducerProp(Properties producerProp) {
    this.producerProp = producerProp;
}

public Properties getConsumerProp() {
    return consumerProp;
}

public void setConsumerProp(Properties consumerProp) {
    this.consumerProp = consumerProp;
}

public Properties getCommonProp() {
    return commonProp;
}

public void setCommonProp(Properties commonProp) {
    this.commonProp = commonProp;
}

@Override
public String toString() {
    return "ConfigUtils{" +
            "producerProp=" + producerProp +
            ", consumerProp=" + consumerProp +
            ", commonProp=" + commonProp +
            '}';
}
```

```
    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) kafkaFlinkHive 类文件

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.Encoder;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.DefaultRollingPolicy;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.security.UserGroupInformation;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.io.PrintStream;
import java.util.Properties;
import java.util.concurrent.TimeUnit;
import java.util.regex.Pattern;


public class kafkaFlinkHive {
    static Logger logger = LoggerFactory.getLogger(kafkaFlinkHive.class);

    public static void main(String[] args) throws Exception {
```

```
        final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
        if (!multipleParameterTool.has("path")) {
            System.out.println("Error: not exist --path /opt/your.properties");
            System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
            System.exit(0);
        }
        ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
        logger.info(configUtils.toString());
        Properties commonProp = configUtils.getCommonProp();

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.getConfig().setUseSnapshotCompression(true);
        env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

        Properties consumerProp = configUtils.getConsumerProp();
        if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
                .equalsIgnoreCase("true")) {
            consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
            consumerProp.setProperty("sasl.mechanism", "GSSAPI");
            consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
                .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));
```

```
    DataStream<String> convertData = sourceDataStream.map(new MapFunction<String, String>()
{
      @Override
      public String map(String value) {
        return value;
      }
    }).uid("convert_data").setParallelism(Integer.valueOf(commonProp.getProperty("convert.forma
t.parallelism")));

    final StreamingFileSink<String> hdfsSink = StreamingFileSink
        .forRowFormat(new Path("hdfs:///tmp"), (Encoder<String>) (element, stream) -> {
          PrintStream out = new PrintStream(stream);
          String[] strArr = element.split(",");
          if (strArr.length == 2 && isInteger(strArr[0])) {
            out.println(element);
          } else {
            logger.error("data format must be : int,string");
          }

          if (commonProp.containsKey("sink.hive.security.enable") &&
commonProp.getProperty("sink.hive.security.enable")
                .equalsIgnoreCase("true")) {
            Configuration conf = new Configuration();
            conf.set("hadoop.security.authentication", "Kerberos");
            UserGroupInformation.setConfiguration(conf);
            try {

UserGroupInformation.loginUserFromKeytab(commonProp.getProperty("sink.hive.keytab.principal
"),
                    commonProp.getProperty("sink.hive.keytab.path"));
            } catch (IOException e) {
              e.printStackTrace();
            }
          }

        }).withRollingPolicy(DefaultRollingPolicy.create()
            .withRolloverInterval(TimeUnit.SECONDS.toMillis(Integer.valueOf(commonProp.getPro
perty("hdfs.rollover.interval.sec"))))
```

```
                    .withInactivityInterval(TimeUnit.SECONDS.toMillis(Integer.valueOf(commonProp.getPr
operty("hdfs.inactivity.interval.sec"))))
                    .withMaxPartSize(1024L * 1024L * 256L).build())
            .build();


convertData.addSink(hdfsSink).uid("sink_hdfs").setParallelism(Integer.valueOf(commonProp.getPro
perty("sink.hdfs.parallelism")));

        convertData.addSink(new
SinkToHive<>(commonProp)).uid("sink_hive").setParallelism(Integer.valueOf(commonProp.getProp
erty("sink.hdfs.parallelism")));

        env.execute("Kafka2Flink2Hive kerberos Example");
    }

    public static boolean isInteger(String str) {
        Pattern pattern = Pattern.compile("^[-\\+]?[\\d]*$");
        return pattern.matcher(str).matches();
    }
}
```

(4) SinkToHive 类文件

```
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
import org.apache.hive.service.cli.HiveSQLException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.util.Properties;

public class SinkToHive<IN> extends RichSinkFunction<IN> implements CheckpointedFunction {

    private Logger logger = LoggerFactory.getLogger(SinkToHive.class);
```

```
private String url;
private String tableName;

private Connection conn = null;
private PreparedStatement preparedStatement = null;

SinkToHive(Properties properties) {
  this.url = properties.getProperty("sink.hive.jdbc.url");
  this.tableName = properties.getProperty("sink.hive.table.name");
}

@Override
public void open(Configuration parameters) throws Exception {
  Class.forName("org.apache.hive.jdbc.HiveDriver");
  conn = DriverManager.getConnection(url);
  logger.info("================open");
}

@Override
public void close() throws Exception {
  try {
    if (preparedStatement != null) {
      preparedStatement.close();
    }

    if (conn != null) {
      conn.close();
    }
  } catch (Exception ex) {
    logger.error(ex.getMessage());
  }
}

@Override
public void invoke(IN value, Context context) throws Exception {
  try {
```

```
            preparedStatement = conn.prepareStatement("load data inpath '/tmp/2020*/part-*' into table " + tableName);
            preparedStatement.execute();
        } catch (HiveSQLException ex) {
            logger.debug(ex.getMessage());
        }
    }

    @Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception {

    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {

    }


}
```

(5) FlinkJob_Kafka2Hive_Ker.properties 配置文件

```
############################## Kafka 配置 ########################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_hive
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_hive
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=true
######################################### hive 配置#################################
convert.format.parallelism=1
sink.hive.table.name=hc_table
```

```
sink.hdfs.parallelism=1
sink.hive.jdbc.url=jdbc:hive2://zyf53.hde.com:2181,zyf55.hde.com:2181,zyf54.hde.com:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2 ;principal=hive/zyf53.hde.com@ZYFTEST.COM
hdfs.rollover.interval.sec=1
hdfs.inactivity.interval.sec=1
sink.hive.security.enable=true
sink.hive.keytab.path=/opt/adminzyf.keytab
sink.hive.keytab.principal=adminzyf
```

5.2.5 Flink DataStream 读 Kafka 写 Redis

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据写入到 Redis。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic ${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.53:2181,10.121.65.54:2181,10.121.65.55:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 修改 Flink 配置

通过 Flink 配置界面修改如下配置项、保存、重启

```
security.kerberos.login.keytab: /opt/adminzyf.keytab
security.kerberos.login.principal: adminzyf
```

(3) 执行 Flink 作业（切换至符合权限要求的用户，本章节以 adminzyf 用户示例）

```
su adminzyf
flink run -m yarn-cluster -d KafkaFlink1.13.6Redis-1.0.jar --path FlinkJob_Kafka2Redis_Ker.properties
```

(4) 生产消息

```
./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name} –producer-property security.protocol=SASL_PLAINTEXT
```

消息格式：Tom

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667。

(5) 查看执行结果

```
redis-cli -p 7001 -a CloudOS5#DE3@Redis -h 10.121.65.53 -c
get 'kafka_flink_redis'
```

3. 样例代码

(1) Pom 文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>demo</artifactId>
    <groupId>com.zyf</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>KafkaFlink1.13.6Redis</artifactId>



  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <flink.version>1.13.6</flink.version>
    <scala.binary.version>2.11</scala.binary.version>
    <hbase.version>2.1.0-cdh6.2.0</hbase.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
```

```xml
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-clients_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>3.3.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <artifactSet>
                            <excludes>
                                <exclude>com.google.code.findbugs:jsr305</exclude>
                                <exclude>org.slf4j:*</exclude>
                                <exclude>log4j:*</exclude>
                            </excludes>
                        </artifactSet>
                        <filters>
                            <filter>
                                <!-- Do not copy the signatures in the META-INF folder.
                                Otherwise, this might cause SecurityExceptions when using the JAR. -->
```

```
                        <artifact>*:*</artifact>
                        <excludes>
                            <exclude>META-INF/*.SF</exclude>
                            <exclude>META-INF/*.DSA</exclude>
                            <exclude>META-INF/*.RSA</exclude>
                        </excludes>
                    </filter>
                </filters>
                <transformers>
                    <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <mainClass>KafkaFlinkRedis</mainClass>
                    </transformer>
                </transformers>
            </configuration>
        </execution>
    </executions>
  </plugin>
 </plugins>
</build>
</project>
```

(2) ConfigUtils 类文件

```
import java.io.FileInputStream;
import java.util.Properties;

public class ConfigUtils {

    private Properties producerProp = new Properties();
    private Properties consumerProp = new Properties();
    private Properties commonProp = new Properties();

    public ConfigUtils(String KafkaConfigPath) {
        try {
            Properties prop = new Properties();
            FileInputStream in = new FileInputStream(KafkaConfigPath);
            prop.load(in);
            in.close();
```

```
            String keyPrefix;
            String keyValue;
            for (String key : prop.stringPropertyNames()) {
                keyPrefix = key.trim().split("\\.")[0];
                keyValue = key.trim().substring(key.trim().indexOf(".") + 1);
                switch (keyPrefix.toLowerCase()) {
                    case "producer":
                        this.producerProp.put(keyValue, prop.getProperty(key));
                        break;
                    case "consumer":
                        this.consumerProp.put(keyValue, prop.getProperty(key));
                        break;
                    default:
                        this.commonProp.put(key, prop.getProperty(key));
                        break;
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public Properties getProducerProp() {
        return producerProp;
    }

    public void setProducerProp(Properties producerProp) {
        this.producerProp = producerProp;
    }

    public Properties getConsumerProp() {
        return consumerProp;
    }

    public void setConsumerProp(Properties consumerProp) {
        this.consumerProp = consumerProp;
    }
```

```
    public Properties getCommonProp() {
        return commonProp;
    }

    public void setCommonProp(Properties commonProp) {
        this.commonProp = commonProp;
    }

    @Override
    public String toString() {
        return "ConfigUtils{" +
                "producerProp=" + producerProp +
                ", consumerProp=" + consumerProp +
                ", commonProp=" + commonProp +
                '}';
    }

    public static void main(String[] args) {
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").producerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").consumerProp);
        System.out.println(new ConfigUtils("d://ConfigUtils.properties").commonProp);
    }

}
```

(3) KafkaFlinkRedis 类文件

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Date;
import java.util.Properties;
```

```
public class KafkaFlinkRedis {
  public static void main(String[] args) throws Exception {
    Logger logger = LoggerFactory.getLogger(KafkaFlinkRedis.class);

    final MultipleParameterTool multipleParameterTool = MultipleParameterTool.fromArgs(args);
    if (!multipleParameterTool.has("path")) {
      System.out.println("Error: not exist --path /opt/your.properties");
      System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
      System.exit(0);
    }
    ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
    logger.info(configUtils.toString());
    Properties commonProp = configUtils.getCommonProp();

    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setUseSnapshotCompression(true);
    env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

    Properties consumerProp = configUtils.getConsumerProp();
    if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
        .equalsIgnoreCase("true")) {
      consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
      consumerProp.setProperty("sasl.mechanism", "GSSAPI");
      consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
    }

    FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
        new SimpleStringSchema(), consumerProp);
    kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
    if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
        .equalsIgnoreCase("latest")) {
```

```
            kafkaConsumer.setStartFromLatest();
        }

        DataStream<String> sourceDataStream = env.addSource(kafkaConsumer).uid("source_kafka")
            .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parallelism")));

        DataStream<String> convertData = sourceDataStream.map(new MapFunction<String, String>()
{
            @Override
            public String map(String value) {
                return value + new Date();
            }
        }).uid("convert_data").setParallelism(Integer.valueOf(commonProp.getProperty("convert.forma
t.parallelism")));

        convertData.addSink(new SinkToRedis<String>(commonProp)).uid("sink_redis")
            .setParallelism(Integer.valueOf(commonProp.getProperty("sink.redis.parallelism")));

        env.execute("KafkaFlink1.13.6Redis kerberos or not Example");
    }
}
```

(4) SinkToRedis 类文件

```
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import redis.clients.jedis.HostAndPort;
import redis.clients.jedis.JedisCluster;
import redis.clients.jedis.JedisPoolConfig;

import java.util.HashSet;
import java.util.Properties;
import java.util.Set;

public class SinkToRedis<IN> extends RichSinkFunction<IN> implements CheckpointedFunction {
```

```
private Logger logger = LoggerFactory.getLogger(SinkToRedis.class);

private String ipPorts;
private String passWord;
private JedisCluster cluster;

SinkToRedis(Properties properties) {
   this.ipPorts = properties.getProperty("sink.redis.ip.ports");
   this.passWord = properties.getProperty("sink.redis.password");
}

@Override
public void open(Configuration parameters) throws Exception {
   Set<HostAndPort> nodes = new HashSet<>();
   String[] ipPorts = this.ipPorts.split(",");
   for (String ipPort : ipPorts) {
      nodes.add(new HostAndPort(ipPort.split(":")[0], Integer.valueOf(ipPort.split(":")[1])));
   }
   this.cluster = new JedisCluster(nodes, 5000, 3000, 10, this.passWord, new JedisPoolConfig());
   logger.info("================open");
}

@Override
public void close() throws Exception {
   this.cluster.close();
}

@Override
public void invoke(IN value, Context context) throws Exception {
   logger.info("================invoke");
   logger.info("result : " + this.cluster.set("kafka_flink_redis", value.toString()));
   logger.info(this.cluster.get("kafka_flink_redis"));
}

@Override
public void snapshotState(FunctionSnapshotContext context) throws Exception {
```

```
    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {

    }


}
```

(5) FlinkJob_Kafka2Redis_Ker.properties 配置文件

```
############################ Kafka 配置 #####################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.13.6_redis
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_redis
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=true
##################################### redis 配置################################
sink.redis.ip.ports=10.121.65.53:7001,10.121.65.53:7002
sink.redis.password=CloudOS5#DE3@Redis
convert.format.parallelism=1
sink.redis.parallelism=1
```

5.2.6 Flink DataStream 读 Kafka 写 Hudi 同步 Hive

1. 样例主要功能

从 Kafka 的一个 topic 中读取数据以 Hudi 格式写入到 HDFS 并同步到 Hive。

2. 样例执行主要步骤

(1) 创建 Kafka 主题

```
./kafka-topics.sh --create --zookeeper ${zkUrl} --replication-factor 2 --partitions 3 --topic
${topic_name}
```

其中：

- ${zkUrl}表示 Zookeeper URL，由 Zookeeper 节点的 IP:Port 组成。集群中部署多个 Zookeeper 节点时，以逗号分隔，例如 10.121.65.7:2181,10.121.65.8:2181,10.121.65.9:2181。
- ${topic_name}表示自定义的 topic 名称。

(2) 准备数据 schema.avsc 文件

将 schema.avsc 文件上传到 Hdfs 目录(如：hdfs dfs –put schema.avsc /flink)。

文件内容如下：

```
{
 "type" : "record",
 "name" : "record",
 "fields" : [ {
  "name" : "uuid",
  "type" : [ "null", "string" ],
  "default" : null
 }, {
  "name" : "name",
  "type" : [ "null", "string" ],
  "default" : null
 }, {
  "name" : "age",
  "type" : [ "null", "int" ],
  "default" : null
 }, {
  "name" : "ts",
  "type" : [ "null", {
   "type" : "long",
   "logicalType" : "timestamp-millis"
  } ],
  "default" : null
 }, {
  "name" : "partition1",
  "type" : [ "null", "string" ],
  "default" : null
 }, {
  "name" : "table_name",
  "type" : [ "null", "string" ],
  "default" : null
 }]
```

}

(3) 修改 Flink 配置

通过 Flink 配置界面修改如下配置项、保存、重启

security.kerberos.login.keytab: /opt/adminzyf.keytab

security.kerberos.login.principal: adminzyf

(4) 在所有集群节点创建软链接

ln -s /etc/hive/conf/hive-site.xml /etc/hadoop/conf/

(5) 准备/opt/hudi-defaults.conf 文件，文件内容如下：

security.protocol=SASL_PLAINTEXT

sasl.mechanism=GSSAPI

sasl.kerberos.service.name=kafka

(6) 执行 Flink 作业（切换至符合权限要求的用户，本章节以 adminzyf 用户示例）

su adminzyf

export HUDI_CONF_DIR=/opt/hudi-defaults.conf

./flink run -t yarn-per-job -c org.apache.hudi.streamer.HoodieFlinkStreamer
-d ../opt/hudi-flink-bundle_2.11-0.10.0.jar \

--kafka-bootstrap-servers node7:6667 \

--kafka-topic hudi_kafka \

--kafka-group-id group1 \

--checkpoint-interval 10000 \

--target-base-path hdfs:///flink/t_hudi \

--table-type MERGE_ON_READ \

--target-table t_hudi \

--partition-path-field partition1 \

--source-avro-schema-path hdfs:///flink/schema.avsc \

--bucket-assign-num 1 \

--write-task-num 1 \

--compaction-tasks 1 \

--hive-sync-enable \

--hive-sync-skip-ro-suffix \

--compaction-trigger-strategy num_or_time \

--compaction-delta-commits 1 \

--compaction-delta-seconds 5 \

--hive-sync-partition-extractor-class org.apache.hudi.hive.SlashEncodedDayPartitionValueExtractor
\

--hive-sync-db default \

--hive-sync-table t_hudi \

--hive-sync-mode hms \

--hive-sync-metastore-uris "thrift://node7.hde.com:9083"

(7) 生产消息

./kafka-console-producer.sh --broker-list ${kafka_broker_list} --topic ${topic_name} --producer.config kerberos < kafka_data.txt

其中：

- ${kafka_broker_list}表示 Kafka broker 节点列表，由 Kafka broker 节点的 IP:Port 组成。集群中部署多个 Kafka broker 节点时，以逗号分隔，例如 10.121.65.7:6667,10.121.65.7:6667,10.121.65.7:6667。
- Kafka_data.txt 内容如下：

{"uuid": "id1", "name": "Danny", "age": 23, "ts": "1970-01-01T00:00:01", "partition1": "2022/03/01", "table_name":"table2"}

{"uuid": "id2", "name": "Stephen", "age": 33, "ts": "1970-01-01T00:00:02", "partition1": "2022/03/01", "table_name":"table2"}

{"uuid": "id3", "name": "Julian", "age": 53, "ts": "1970-01-01T00:00:03", "partition1": "2022/03/02", "table_name":"table2"}

{"uuid": "id4", "name": "Fabian", "age": 31, "ts": "1970-01-01T00:00:04", "partition1": "2022/03/02", "table_name":"table2"}

{"uuid": "id5", "name": "Sophia", "age": 18, "ts": "1970-01-01T00:00:05", "partition1": "2022/03/03", "table_name":"table2"}

{"uuid": "id6", "name": "Emma", "age": 20, "ts": "1970-01-01T00:00:11", "partition1": "2022/03/03", "table_name":"table2"}

{"uuid": "id7", "name": "Bob", "age": 44, "ts": "1970-01-01T00:00:23", "partition1": "2022/03/04", "table_name":"table2"}

{"uuid": "id8", "name": "Han", "age": 56, "ts": "1970-01-01T00:00:45", "partition1": "2022/03/04", "table_name":"table2"}

(8) 查看执行结果

将/usr/hdp/3.0.1.0-187/flink/opt/hudi-hadoop-mr-bundle-0.10.0.jar 拷贝到所有 HiveServer2 所在节点/usr/hdp/3.0.1.0-187/hive/lib/，重启 Hive 服务。

beeline

show tables;

select * from t_hudi;

5.3 FlinkSQL-cdc

5.3.1 CDC 简介

CDC 即 Change Data Capture 变更数据捕获，为 Flink 1.11 中一个新增功能。我们可以通过 CDC 得知数据源表的更新内容（包含 Insert Update 和 Delete），并将这些更新内容作为数据流发送到下游系统。捕获到的数据操作具有一个标识符，分别对应数据的增加、修改和删除。

- +I：新增数据。
- -U：一条数据的修改会产生两个 U 标识符数据。其中-U 含义为修改前数据。

- +U：修改之后的数据。
- -D：删除的数据。

5.3.2 FlinkCDC 使用

1. 开启 mysql 的 binlog

说明

使用 CDC 之前必须开启 MySQL 的 binlog。

(1) 修改 my.cnf 配置文件，vi /etc/my.cnf 增加：

server-id=1

log_bin=mysql-bin

binlog_format=ROW

expire_logs_days=30

binlog_do_db=db_a

binlog_do_db=db_b

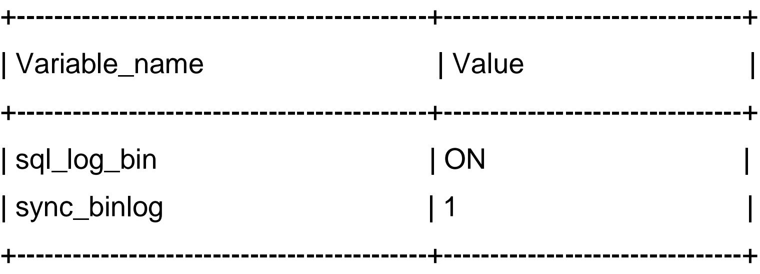
【说明】配置项的解释如下：

- server-id：MySQL5.7 及以上版本开启 binlog 必须要配置这个选项。对于 MySQL 集群，不同节点的 server_id 必须不同。对于单实例部署则没有要求。
- log_bin：指定 binlog 文件名和储存位置。如果不指定路径，默认位置为/var/lib/mysql/。
- binlog_format：binlog 格式。有 3 个值可以选择：ROW：记录哪条数据被修改和修改之后的数据，会产生大量日志。STATEMENT：记录修改数据的 SQL，日志量较小。MIXED：混合使用上述两个模式。CDC 要求必须配置为 ROW。
- expire_logs_days：bin_log 过期时间，超过该时间的 log 会自动删除。
- binlog_do_db：binlog 记录哪些数据库。如果需要配置多个库，如例子中配置多项。切勿使用逗号分隔。

(2) 配置文件修改完毕后，保存并重启 MySQL。然后进入 MySQL 命令行，验证是否已启用 binlog：

mysql -uadmin -pPassw0rd@_ -h127.0.0.1 -P3307

mysql> show variables like '%bin%';



```
+--------------------------------------------+--------------------------------+
| Variable_name                              | Value                          |
+--------------------------------------------+--------------------------------+
| sql_log_bin                                | ON                             |
| sync_binlog                                | 1                              |
+--------------------------------------------+--------------------------------+
```

29 rows in set (0.00 sec)

(3) 根据以上验证情况，可以看出 log_bin 的值为 ON，即表示 binlog 配置已生效。

2. 初始化 MySQL 源数据表

MySQL 环境配置完毕后，需准备测试表和数据。

示例：

- 创建演示数据库 demo 和一张 student 表

  create database demo character set utf8mb4;

  use demo;

  create table student(`id` int primary key, `name` varchar(128), `age` int);

3. FlinkCDC 使用示例（使用 SQL Client 读取 CDC）

相比较创建一个 Java 项目以 jar 包的方式创建作业，Fllink 提供了一个更为简单的方式：使用 SQL Client，有两种创建表的方式：

- MYSQL-CDC 连接方式
- FlinkSQL JDBC 连接方式

（一）MYSQL-CDC 连接方式

使用 MYSQL-CDC 读取 JDBC，操作步骤如下：

(1) 配置 Flink 环境：

启动 flink 的 standalone 模式，适合测试

cd /usr/hdp/3.0.1.0-187/flink/bin

./start-cluster.sh

(2) 启动 Flink SQL Client，进入 SQL Client。

./sql-client.sh embedded -j ../opt/flink-sql-connector-mysql-cdc-2.1.1.jar

(3) 在 SQL Client 中执行如下 SQL：

说明

- SQL Client 默认使用 hive catalog，需安装 Hive 组件；也可以通过修改 conf/sql-client-defaults.yaml 配置文件进行指定。
- 本示例以 hive catalog 为例进行说明。

```
use catalog flink_hive_catalog;
CREATE TABLE mysql_binlog (
 id INT NOT NULL,
 name STRING,
 age INT
) WITH (
 'connector' = 'mysql-cdc',
 'scan.incremental.snapshot.enabled'='false',
 'hostname' = '10.121.68.131',
 'port' = '3307',
 'username' = 'admin',
```

```
 'password' = 'Passw0rd@_',
 'database-name' = 'demo',
 'table-name' = 'student');
CREATE TABLE sink_table (
 id INT NOT NULL,
 name STRING,
 age INT
) WITH (
   'connector' = 'print'
);
INSERT INTO sink_table SELECT id, name, age FROM mysql_binlog;
```

(4) 在 MySQL 命令行，执行 insert 语句插入数据。

示例：

```
insert into student values(1,'paul',20);
update student set age=30 where id=1;
delete from student where id=1;
```

【注意】sink_table 的输出无法在 SQL client 上面查看，需打开 Flink Web UI 的 Task Managers 页面的 stdout 标签查看或者打开后台日志 /usr/hdp/3.0.1.0-187/flink/log/flink-root-taskexecutor*.out 查看，后台日志输出如图 5-1 所示，即表明 Flink 已经成功捕获到 MySQL 的数据变更。

图5-1 输出

```
2022-04-08 17:42:47.875 INFO  (debezium-engine) [    ] i.d.c.c.BaseSourceTask 4 records s
, ts_sec=1649410968, file=mysql-bin.000004, pos=68503667, row=1, server_id=1, event=2}
-U[1, paul, 20]
+U[1, paul, 30]
-D[1, paul, 30]
~
```

（二）FlinkSQL JDBC 连接方式

使用 SQL Client 读取 JDBC，操作步骤如下：

(1) 配置 Flink 环境：

启动 flink 的 standalone 模式，适合测试。

```
cd /usr/hdp/3.0.1.0-187/flink/bin
./start-cluster.sh
```

(2) 启动 Flink SQL Client，进行 SQL Client。

```
./sql-client.sh embedded -j /usr/hdp/3.0.1.0-187/dlh/hive/lib/mysql-connector-java.jar
```

【说明】此命令适用于集群安装了 DLH 组件的场景，可直接使用 catalog flink_hive_catalog。

(3) 在 SQL Client 中执行如下 SQL：

```
use catalog flink_hive_catalog;
CREATE TABLE flink_mysql (
```

```
 id INT NOT NULL,
 name STRING,
 age INT
) WITH (
 'connector' = 'jdbc',
 'url'='jdbc:mysql://10.121.68.131:3307/demo',
 'table-name' = 'student',
 'username' = 'admin',
 'password' = 'Passw0rd@_');
```

(4) 查询结果，如图 5-2 所示。

示例：

```
set sql-client.execution.result-mode=tableau;
select * from flink_mysql;
```

图5-2 查询结果

```
Flink SQL> select * from flink_mysql;
+----+-------------+-------------------------------+-------------+
| op |          id |                          name |         age |
+----+-------------+-------------------------------+-------------+
| +I |           1 |                          paul |          20 |
| +I |           2 |                          paul |          20 |
+----+-------------+-------------------------------+-------------+
Received a total of 2 rows
```

6 常见问题解答

6.1 常用配置

1. 基础配置

表6-1 基础配置说明

配置参数	说明
jobmanager.rpc.address	Jobmanager 的 IP 地址
jobmanager.rpc.port	Jobmanger端口号
jobmanager.heap.mb	Jobmanager JVM heap内存大小
taskmanager.heap.mb	Taskmanager JVM heap内存大小
taskmanager.numberOfTaskSlots	Taskmanger提供的任务slots数量大小
parallelism.default	Flink任务默认并行度
web.port	Historyserver端口号
jobmanager.archive.fs.dir	已完成作业上传的目录
historyserver.archive.fs.refreash-interval	刷新存档作业的作业目录的时间间隔
state.backend	存储检查点的存储类型
state.backend.fs.checkpointdir	存储检查点的数据文件和元数据的默认目录
state.checkpoints.dir	Savepoint 目录
state.savepoint.num_retained	保留最近的检查点数量
state.backend.incremental	开启增量ck全局生效
akka.ask.timeout	超时
akka.watch.heartbeat.pause	由于丢失或延迟的心跳消息而错误的将taskmanager标记为无效
taskmanager.network.memory.max	网络缓冲区的最大内存大小
taskmanager.network.memory.min	网络缓冲区的最小内存大小
taskmanager.network.memory.fraction	网络缓冲区的JVM内存的分数
fs.hdfs.hadoopconf	Hadoop配置文件地址
yarn.application-attempts	任务失败尝试次数

2. Hadoop 配置说明

表6-2 Hadoop 配置说明

配置参数	说明
yarn.nodemanager.resource.cpu-vcores	每个节点最大使用的vcore数，可以适当放大
yarn.scheduler.minimum-allocation-mb	设置container最小可申请内存量，默认1024
yarn.scheduler.minimum-allocation-vcores	设置container最小可申请CPU数量，默认是1
yarn.scheduler.maximum-allocation-mb	单个container最大可申请内存量，默认是单个NodeManager最大的内存
yarn.scheduler.maximum-allocation-vcores	单个container最大可申请CPU数量

6.2 调优

1. 设置并行度

并行度控制任务的数量，影响操作后数据被切分成的块数。适当的并行度可以让任务的数量和每个任务处理的数据与机器的处理能力达到最优。

查看 CPU 使用情况和内存占用情况，当任务和数据不是平均分布在各节点，而是集中在某个别节点时，可以增大并行度使任务和数据更均匀的分布在各个节点。增加任务的并行度，可以充分利用集群机器的计算能力，一般将并行度设置为集群 CPU 核数总和的 2-3 倍。

任务的并行度可以通过以下四种层次（按优先级从高到低排列）指定，用户可以根据实际的内存、CPU、数据以及应用程序逻辑的情况调整并行度参数。

- 算子层次

  一个算子、数据源和 sink 的并行度可以通过调用 setParallelism()方法来指定。例如：

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<String> text = [...]
DataStream<Tuple2<String, Integer>> wordCounts = text
.flatMap(new LineSplitter())
.keyBy(0)
.timeWindow(Time.seconds(5))
.sum(1).setParallelism(5);
wordCounts.print();
env.execute("Word Count Example");
```

- 执行环境层次

  Flink 程序运行在执行环境中。执行环境为所有执行的算子、数据源、data sink 定义了一个默认的并行度。

  执行环境的默认并行度可以通过调用 setParallelism()方法指定。例如：

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(3);
DataStream<String> text = [...]
DataStream<Tuple2<String, Integer>> wordCounts = [...]
```

```
wordCounts.print();
env.execute("Word Count Example");
```

- 客户端层次

  并行度可以在客户端将 job 提交到 Flink 时设定。对于 CLI 客户端，可以通过“-p”参数指定并行度。例如：

  ```
  ./bin/flink run -m yarn-cluster -p 10 ../examples/*WordCount-java*.jar
  ```

- 系统层次

  在系统级可以通过修改 Flink 客户端 conf 目录下的“flink-conf.yaml”文件中的“parallelism.default”配置选项来指定所有执行环境的默认并行度。

2. 配置内存

Flink 依赖内存计算，计算过程中内存不够对 Flink 的执行效率影响很大。可以通过监控 GC（Garbage Collection），评估内存使用及剩余情况来判断内存是否变成了性能瓶颈，并根据实际情况优化。监控节点进程的 YARN 的 Container GC 日志，如果频繁出现 Full GC，则需要优化 GC。

GC 日志的配置方法：在客户端的“/usr/hdp/3.0.1.0-187/flink/conf/flink-conf.yaml”配置文件中，添加配置项“env.java.opts”，然后添加参数:“-Xloggc:<LOG_DIR>/gc.log -XX:+PrintGCDetails -XX:-OmitStackTraceInFastThrow -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=20 -XX:GCLogFileSize=20M”。

3. 配置进程参数

Flink on YARN 模式下，有 JobManager 和 TaskManager 两种进程。在任务调度和运行的过程中，JobManager 和 TaskManager 的参数配置对 Flink 应用的执行有着很大的影响意义。

(1) 配置 JobManager 内存

JobManager 负责任务的调度，以及 TaskManager、ResourceManager 之间的消息通信。当任务数变多，任务并发度增大时，JobManager 内存都需要相应增大。

根据实际任务数量的个数，需要为 JobManager 设置一个合适的内存，方法如下：

- 在使用 yarn-session 命令时，添加“-jm MEM”参数设置内存。
- 在使用 yarn-cluster 命令时，添加“-yjm MEM”参数设置内存。

(2) 配置 TaskManager 个数

每个 TaskManager 每个核同时能运行一个 task，增加了 TaskManager 的个数相当于增大了任务的并发度。所以，在资源充足的情况下，可以相应增加 TaskManager 的个数，以提高运行效率，方法如下：

- 在使用 yarn-session 命令时，添加“-n NUM”参数设置 TaskManager 个数。
- 在使用 yarn-cluster 命令时，添加“-yn NUM”参数设置 TaskManager 个数。

4. 设计分区方法

合理的分区设计可以优化 task 的切分。在 Flink 程序编写过程中要尽量分区均匀，这样可以实现每个 task 数据不倾斜，防止由于某个 task 的执行时间过长导致整个任务执行缓慢。

5. 配置 netty 网络通信

Flink 通信主要依赖 netty 网络，所以在 Flink 应用执行过程中，netty 的设置尤为重要，网络通信的好坏直接决定着数据交换的速度以及任务执行的效率。

6.3 运维类问题

1. 隐式转换问题

问题现象：

could not find implicit value for evidence parameter of type org.apache.flink.api.common.typeinfo.TypeInformation[String]

解决方法：

(1) 导入包：import org.apache.flink.api.scala._

(2) 在类中显示定义隐式转换，代码示例：implicit val typeInfo:TypeInformation[String]= TypeInformation.of(classOf[String])，这里的 String 和具体情况相关，如果是 Tuple2[String,String]，则需要对应修改。

2. ClassNotFoundException，NoSuchMethodError 相关错误

表6-3 错误现象对应的排查思路

错误现象	排查思路
ClassNotFoundException	• 检查是否缺少 jar 包 • 检查 jar 包中是否没有这个类。可能原因：Flink 程序在运行时，会先加载自己 lib 下的类，然后再加载用户 jar 中的类。若发现自己 lib 中有同名的类路径，但是没有类，如 flink lib 中有 org.apache.hadoop.fs 路径，但是没有 FileSystem 类，而运行时又需要这个类，即使后面的 jar 包中有 org.apache.hadoop.fs.FileSystem 也不会去加载
NoSuchMethodError	• 检查类中是否没有这个方法 • 检查多个 jar 包是否冲突。通过 maven helper 可以看 jar 包是否冲突，或者 mvn dependency:tree 看 jar 依赖

3. Savepoints 相关问题解决方案

- 用户必须为 job 中的所有算子都分配 ID 吗？

  严格的说，用户只给有状态的算子分配 IDs 即可，因为在 savepoint 中仅包括有状态算子的状态，没有状态的算子并不包含在 savepoint 中。但是在实际应用中，强烈建议用户给所有的算子均分配 ID，因为有些 Flink 的内置算子（比如 window 算子）也是有状态的。

- 如果用户在升级作业时新添加一个有状态的算子有什么影响？

  当用户在作业中新添加一个有状态的算子时，由于该算子是新添加的，无保存的旧状态，因此无状态恢复，将从 0 开始运行。

- 如果用户在升级作业时从作业中删除一个有状态的算子有什么影响？

  默认情况下，savepoint 会尝试将所有保存的状态恢复。如果用户使用的 savepoint 中包含已经删除算子的状态，恢复将会失败。用户可以通过--allowNonRestoredState(简写为-n)参数跳过恢复已经删除的算子的状态。

- 如果用户重新编排有状态的算子的顺序有什么影响？

  如果用户已经给这些算子分配 IDs，那么这些状态会正常恢复。

  如果用户没有给这些算子分配 IDs，则这些有状态算子将会按新的顺序自动分配新的 ID，这将导致状态恢复失败。

- 如果用户在作业中删除或添加或更改无状态算子的顺序有什么影响？

  如果用户已经给有状态的算子分配 IDs，那么无状态的算子并不会影响从 savepoint 进行状态恢复。

  如果用户没有给有状态的算子分配 IDs，则有状态算子的 IDs 由于顺序变化可能会被分配新的 IDs，这将导致状态恢复失败。

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.2 基本概念 1-1
1.3 组件架构 1-2
1.3.1 Storm 集群架构 1-2
1.3.2 Topology 任务处理原理 1-2
1.4 应用场景 1-4
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 数据目录检查 2-1
2.1.2 查看组件的日志信息 2-1
2.2 运行状态监控 2-2
2.3 快速使用指导 2-4
2.3.1 非 Kerberos 环境 2-5
2.3.2 Kerberos 环境 2-5
2.4 快速链接 2-7
2.4.1 配置组件快速链接 2-7
2.4.2 访问 Storm 快速链接 2-8
3 使用指南 3-1
3.1 常用命令 3-1
3.2 Client 下载/安装/使用/卸载 3-3
3.2.1 下载 Client 安装包 3-4
3.2.2 安装 Client 3-5
3.2.3 访问组件 3-6
3.2.4 卸载 Client 客户端 3-6
3.3 Storm 集群扩容 3-6
3.3.1 使用场景 3-6
3.3.2 扩容前准备 3-7
3.3.3 扩容约束 3-7
3.3.4 扩容影响 3-7
3.3.5 扩容操作指导 3-7
3.3.6 扩容验证 3-8

3.4 Storm 集群缩容 3-8
3.4.1 使用场景 3-8
3.4.2 缩容前准备 3-9
3.4.3 缩容约束 3-9
3.4.4 缩容影响 3-9
3.4.5 缩容操作指导 3-9
3.4.6 缩容验证 3-10
4 开发指南 4-1
4.1 Java API 4-1
4.2 Restful API 4-3
4.3 JavaAPI 基础开发样例 4-4
5 最佳实践 5-1
5.1 Storm-Kafka 开发样例 5-1
5.2 Storm-Redis 开发样例 5-6
5.3 Storm-HBase 开发样例 5-11
5.4 Storm-JDBC 开发样例 5-18
6 常见问题解答 6-1
6.1 调优 6-1
6.2 运维类问题 6-2

1 组件简介

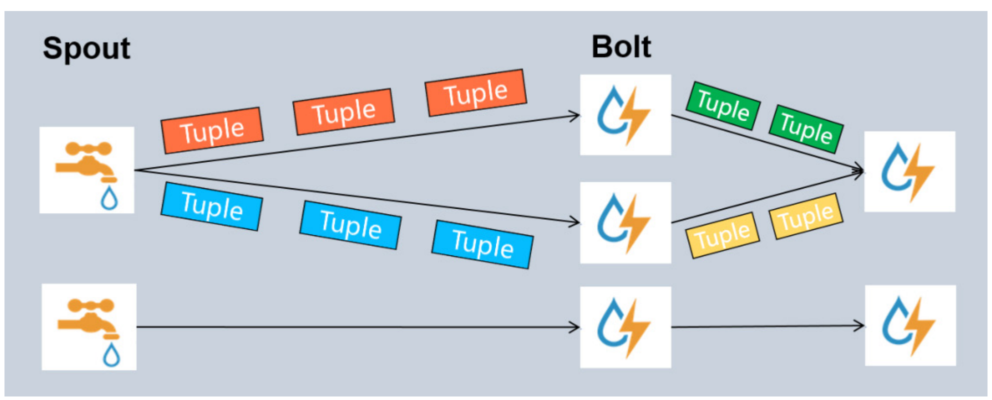
1.1 组件概述

Storm 是高可靠、高度容错的分布式流计算服务。Storm 将计算任务下发给架构内不同类型的组件，每个组件仅负责处理一项特定的任务，具有运维简单、无数据丢失且支持多语言的特点。

1.2 基本概念

- Topology：Storm 中运行的一个实时应用程序。Storm 的各个组件间的消息流动会形成逻辑上的一个拓扑结构，其中：Topology 内部源头为 Spout，后面接每一个处理 Bolt。Spout 和 Bolt 之间连接的线称为 Stream，线上的每个点称为 Tuple。
- Spout：产生源数据流的组件。通常情况下 Spout 会从外部数据源中读取数据，然后转换为 Topology 内部源数据。
- Bolt：在一个 Toplogy 中接受数据然后执行处理的组件。Bolt 可以接收任意多个输入的 Stream，有些 Bolt 可能也会发射一些新的 Stream。Bolt 处理输入的 Stream，并产生新的输出 Stream。Bolt 可以执行过滤、函数操作、Join、操作数据库等操作。
- Stream：一个没有边界的、源源不断的、连续的 Tuple 序列就组成了 Stream。
- Tuple：一次消息传递的基本单元。

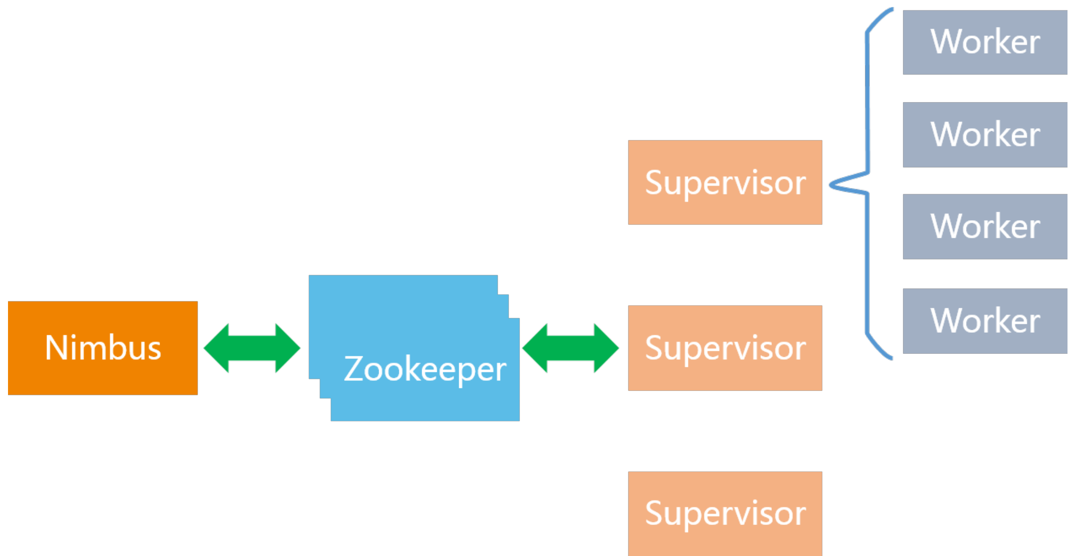
图1-1 Topolog 结构示意图



1.3 组件架构

1.3.1 Storm 集群架构

图1-2 Storm 集群架构



Storm 集群由一个或多个主节点（多个主节点选举出 Leader）和多个工作节点组成。其中：

- 在主节点上运行 Nimbus 和 UI 守护进程
  - Nimbus 用于响应分布在集群中的节点，负责资源分配和任务调度。角色为 Leader 的 Nimbus 进程负责在集群中分发代码，为工作节点分配任务并监控故障等
  - UI 进程提供了对集群监控的一个 Web 实现
- 在工作节点上运行 Supervisor 守护进程

  Supervisor 负责接收 Nimbus 分配的任务，然后启动或停止其管理的 Worker 进程来运行具体的逻辑。Worker 是一个 Java 进程，用来运行具体逻辑，一个 Storm 拓扑任务通常运行在多个工作节点的 Worker 进程中，用于提高吞吐量。

另外，Storm 集群基于 ZooKeeper 维持集群信息（如心跳信息、集群状态、配置信息等），Zookeeper 协调共有数据的存放，Nimbus 将分配给 Supervisor 的任务写在 Zookeeper 中。守护进程可以是无状态的，并且当失效或重启时不影响整个系统的正常运转。

1.3.2 Topology 任务处理原理

Storm 集群上运行的是一个个 Topology 任务，每个 Topology 由 Spouts 和 Bolts 组成。Topology 任务提交后，Nimbus 首先负责分配任务，并将任务分配信息保存到 ZooKeeper。Supervisor 通过保存在 zookeeper 上的任务信息来获取自己的任务，并把自身需要执行的任务及 jar 包等保存到本地磁盘上。然后 Superviosr 再启动对应数量的 Worker 进程来执行具体任务（Worker 中的每个 Executor 就是一个个 Bolt 或 Spout）。

图1-3 Topology 任务处理流程

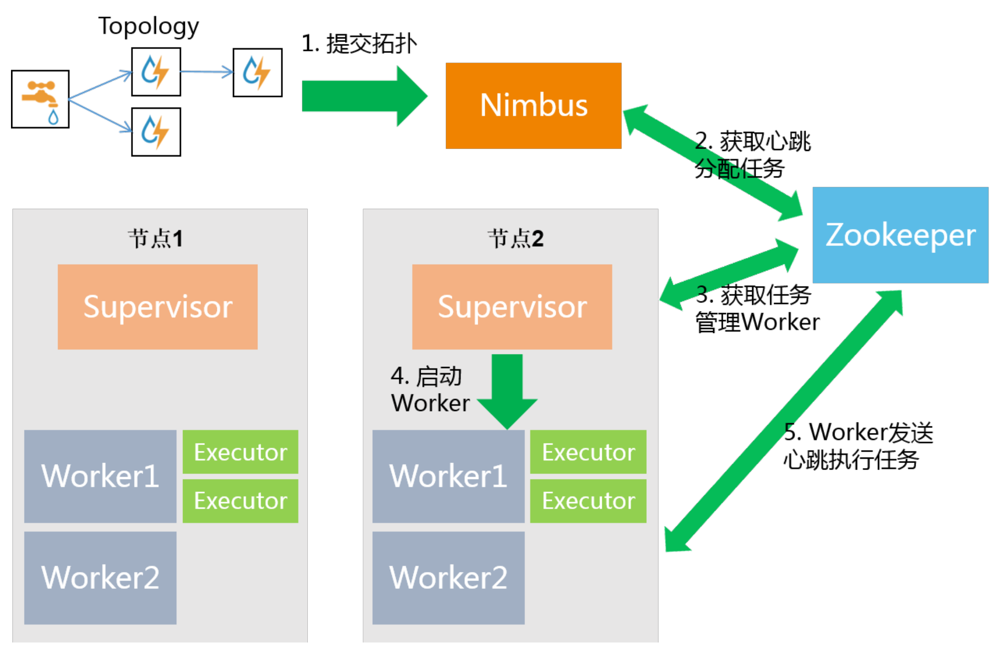
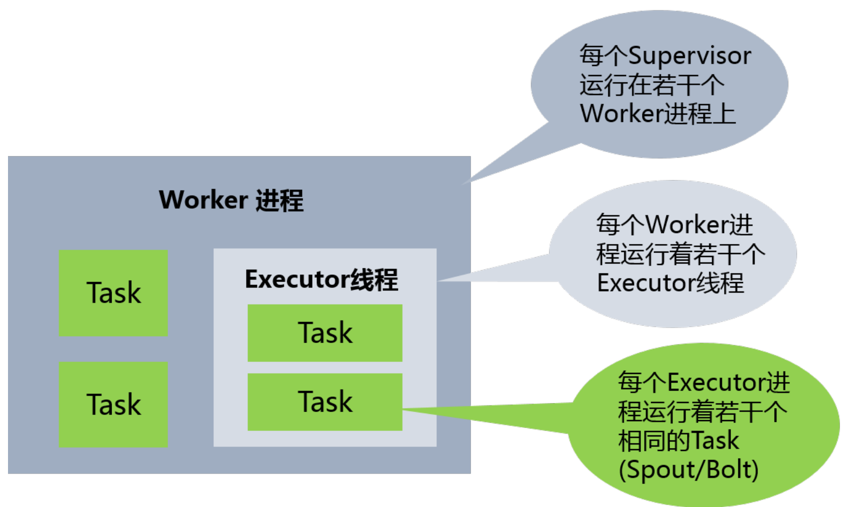


图1-4 Worker 进程图解



Topology 的任务处理说明：

- Storm 集群里的一台物理机会启动 1 个或多个 Work 进程，Worker 进程执行的是 Topology 任务的子集（每个 Worker 只为其中一个 Topology 任务服务）。1 个 Worker 进程会启动 1 个或多个 Executor 线程来执行 1 个 Topology 任务的 component（Spout 或 Bolt）逻辑。因此，1 个运行中的 Topology 就是由集群中多台物理机上的多个 Worker 进程组成的。
- Executor 是被 Worker 进程启动的单独线程。每个 Executor 只会运行 1 个 Topology 的 1 个 component（Spout 或 Bolt）的 Task（Task 可以是 1 个或多个，Storm 默认是 1 个 component 只生成 1 个 Task，Executor 线程会在每次循环里顺序调用所有 Task 实例）。

- Task 是最终运行 Spout 或 Bolt 中代码的单元（1 个 Task 即为 Spout 或 Bolt 的 1 个实例，Executor 线程在执行期间会调用该 Task 的 nextTuple 或 execute 方法）。Topology 启动后，1 个 component（Spout 或 Bolt）的 Task 数目是固定不变的，但该 component 使用的 Executor 线程数可以动态调整。例如 1 个 Executor 线程可以执行该 component 的 1 个或多个 Task 实例，当执行多个 Task 实例时 Executor 的数量就减少了。这意味着，对于 1 个 component 存在这样的条件：#threads<=#Tasks，即线程数小于等于 Task 数目。默认情况下 Task 的数目等于 Executor 线程数目，即 1 个 Executor 线程只运行 1 个 Task。
- Spout 或者 Bolt 的 Task 个数一旦指定之后就不能再改变，而 Executor 的数量可以根据情况来进行动态调整。默认情况下#Executor=#Tasks，即一个 Executor 中运行着一个 Task。一个 Topology 可以通过 setNumWorkers 来设置 Worker 的数量，通过设置 parallelism 来规定 Executor 的数量（一个 component（spout/bolt）可以由多个 Executor 来执行），通过 setNumTasks 来设置每个 Executor 跑多少个 Task（默认一对一）。

1.4 应用场景

Storm 适用的场景包括：实时分析、持续计算和分布式 ETL 等，具有易于扩展、支持容错的特点，可确保数据得到处理且易于构建和操控。主要适用场景说明如下：

- 信息流处理

  Storm 可用来实时处理数据、更新数据，兼具容错性和可扩展性。即 Storm 可以用来处理源源不断流进来的消息，处理之后将结果写入存储。

- 持续计算

  Storm 可连续发送数据到客户端，并实时更新和显示结果。比如将微博上的热门话题转发给用户。

- 分布式远程调用

  Storm 可通过网络从远程计算机程序上请求服务，用来并行处理密集查询并进行计算，同时返回查询结果。

- 其他

  Storm 可用于实时分析、在线机器学习、ETL（数据抽取、转换和加载）等。

2 快速入门

2.1 组件安装

说明

- 在 Hadoop 集群中，安装 Storm 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- Storm 依赖 Zookeeper，安装 Storm 时必须同时安装 Zookeeper 组件。
- Supervisor 主要负责管理工作进程、工作进程数量或内存配置，对资源占用较大。因此，如需提供大量计算能力，Supervisor 可部署多个，且建议部署在独立节点上。

2.1.1 数据目录检查

根据现场磁盘分区方案和挂盘方案的不同，Storm 安装完成后，必须对其数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表 2-1 所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
Storm	是（配置项的参数值默认使用某一个挂载路径）	storm.local.dir	此目录为Storm使用的本地文件系统目录，用于保存少量状态信息。检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录 • 建议此目录为 Storm 服务独立使用。如果该目录同时被其他服务使用，需手动修改为其他路径
Zookeeper	是（配置项的参数值默认使用某一个挂载路径）	dataDir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录

2.1.2 查看组件的日志信息

表2-2 组件日志路径说明

组件	日志路径
Storm	/var/de_log/storm

组件	日志路径
ZooKeeper	/var/de_log/zookeeper/user_{user.name}/，其中${user.name}是指执行任务的用户名

2.2 运行状态监控

1. 查看组件详情

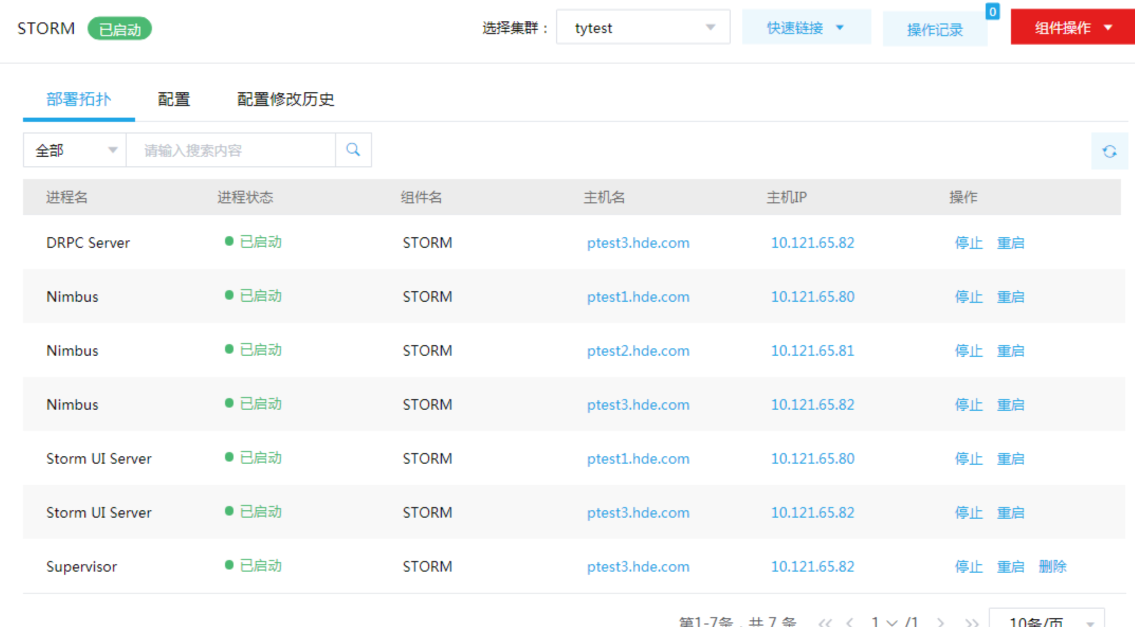
进入 Storm 组件详情页面，如图 2-1 所示。组件详情页面主要展示部署拓扑、配置详情和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 组件详情



2. 组件检查

Storm 组件检查主要用于检查在 Storm 集群中提交和执行拓扑任务的功能是否正常。执行 Storm 组件检查时，会提交 Storm 自带的 WordCount 示例任务并检测该任务的运行结果，若 Storm 组件检查成功则表示在 Storm 集群中可以正常提交和执行拓扑任务。

集群在使用过程中，根据实际需要，可对 Storm 执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：
  - 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
    - 在集群详情页面选择[组件]页签，单击组件列表中 Storm 组件对应的<组件检查>按钮。
    - 在集群详情页面选择[组件]页签，单击组件列表中 Storm 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
  - 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

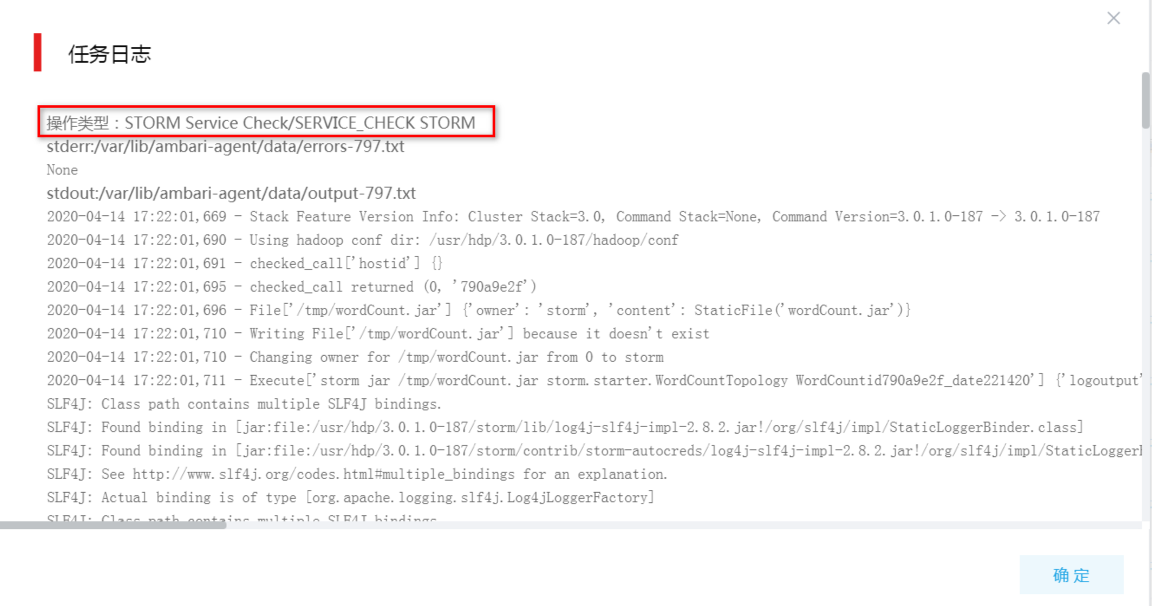
(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态。如图 2-2 所示，表示组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“STORM Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的结果。

图2-3 组件检查日志详情



2.3 快速使用指导

注意

根据大数据集群是否开启 Kerberos 认证，用户提交 Storm 任务时的认证方式不同，详情请参见本章节内容。

Storm 任务既可以通过集群用户提交，又可以通过组件超级用户提交。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
    - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。
    - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 HDFS 组件的 hdfs 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 Kerberos 环境

说明

- 非 Kerberos 环境下，不需要用户做身份认证即可直接提交 Storm 任务。
- 本章节操作需要切换至具有相关权限的用户。

Storm 组件中提供基本测试样例，本章节以 Storm 自带测试样例展示其基本使用。

(1) 连接大数据集群内安装 Storm 的任意一节点或安装 Storm Client 的节点。

(2) 提交 Storm 自带样例任务，命令如下：

```
/usr/hdp/3.0.1.0-187/storm/bin/storm jar
/usr/hdp/3.0.1.0-187/storm/contrib/storm-starter/storm-starter-1.2.1.3.0.1.0-187.jar
storm.starter.StatefulTopology test
```

其中各参数说明如下：

- storm：Storm 基础脚本，已设置全局变量
- jar：提交任务的固定参数
- /usr/hdp/3.0.1.0-187/storm/contrib/storm-starter/storm-starter-1.2.1.3.0.1.0-187.jar：Storm 自带样例 jar 包的路径
- storm.starter.StatefulTopology：Storm 自带样例的入口类
- test：该 topology 任务的自定义名称（注意：若集群中已存在该自定义任务名，则任务将无法提交）。

(3) 任务提交后，观察控制台打印信息，若出现“Finished submitting topology: test”，即说明任务运行完成。

2.3.2 Kerberos 环境

注意

- Kerberos 环境下，若想提交 Storm 任务，则必须首先进行用户身份认证，认证方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。
- 本章节操作需要切换至具有相关权限的用户。

1. Kerberos 环境下用户身份认证

如果大数据集群开启 Kerberos，若想提交 Storm 任务，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

Storm 任务通过集群用户提交时，在开启 Kerberos 的大数据集群中，进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**
  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的/etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：

  chown user1 /etc/security/keytabs/user1.keytab

  b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：

  klist -k user1.keytab

  【说明】如图 2-4 所示，红框内容即为 user1.keytab 的 principal 名称。

  图2-4 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

  c. 切换至用户 user1，并执行身份验证的命令如下：

  su user1

  kinit -kt user1.keytab user1@TENANTC.COM

  【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

  d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**
  a. 输入以下命令：kinit user1
  b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>
  c. 输入 ***klis***t 命令可查看认证结果。

图2-5 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

（二）组件超级用户身份认证

Storm 任务可以通过组件超级用户提交，比如 hdfs 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 hdfs 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 hdfs 的认证文件“hdfs.headless.keytab”。

【说明】在 Storm Client 节点上，需要将 hdfs 的认证文件“hdfs.headless.keytab”上传节点的/etc/security/keytabs/目录下进行认证。

(2) 使用 ***klist*** 命令查看 hdfs.headless.keytab 的 principal 名称，命令如下：

klist -k hdfs.headless.keytab

【说明】如图 2-6 所示，红框内容即为 hdfs.headless.keytab 的 principal 名称。

图2-6 认证文件的 principal 名称

```
[root@hppnode1 keytabs]# klist -k hdfs.headless.keytab
Keytab name: FILE:hdfs.headless.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
   2 hdfs-testshare@TESTSHARE.COM
```

(3) 切换至用户 hdfs，并执行身份验证的命令如下：

su hdfs

kinit -kt hdfs.headless.keytab hdfs-testshare@TESTSHARE.COM

【说明】其中：hdfs.headless.keytab 为 hdfs 的认证文件，hdfs-testshare@TESTSHARE.COM 为 hdfs.headless.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

2. 运行基本测试样例

用户身份认证成功后，即可运行 Storm 组件中提供基本测试样例，详情请参见 2.3.1 非 Kerberos 环境。

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

2.4.2 访问 Storm 快速链接

Storm 提供了作业监控页面页面（即 Storm UI），通过该页面可以查看任务的执行情况。

(1) 如图 2-7 所示，在 Storm 组件详情页面的右上角[快速链接]的下拉框中，可以获取访问入口信息。

【说明】当集群开启高可用时，Storm 同步开启 HA，此时有两个访问入口，任选其一即可。

图2-7 Storm 快速链接

(2) 根据集群是否开启 Kerberos，访问 Storm 快速链接分为两种情况：

- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，可直接跳转访问对应的 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。

(3) 在 Storm 作业监控界面，如图 2-8 所示，在“Topology Summary”监控栏内找到之前提交的 topology 任务“test”。

图2-8 Storm 作业监控内容

Storm UI

Cluster Summary

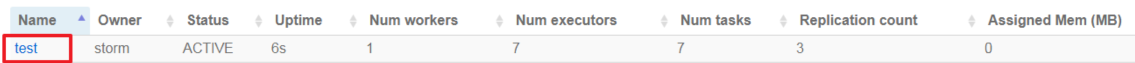
Version	Supervisors	Used slots	Free slots	Total slots	Executors
1.2.1.3.0.1.0-187	1	1	1	2	7

Nimbus Summary

Host	Port	Status	Version	UpTim
host102.hde.com	6627	Not a Leader	1.2.1.3.0.1.0-187	10h 8m
host103.hde.com	6627	Leader	1.2.1.3.0.1.0-187	10h 8m
host104.hde.com	6627	Not a Leader	1.2.1.3.0.1.0-187	10h 8m

Showing 1 to 3 of 3 entries

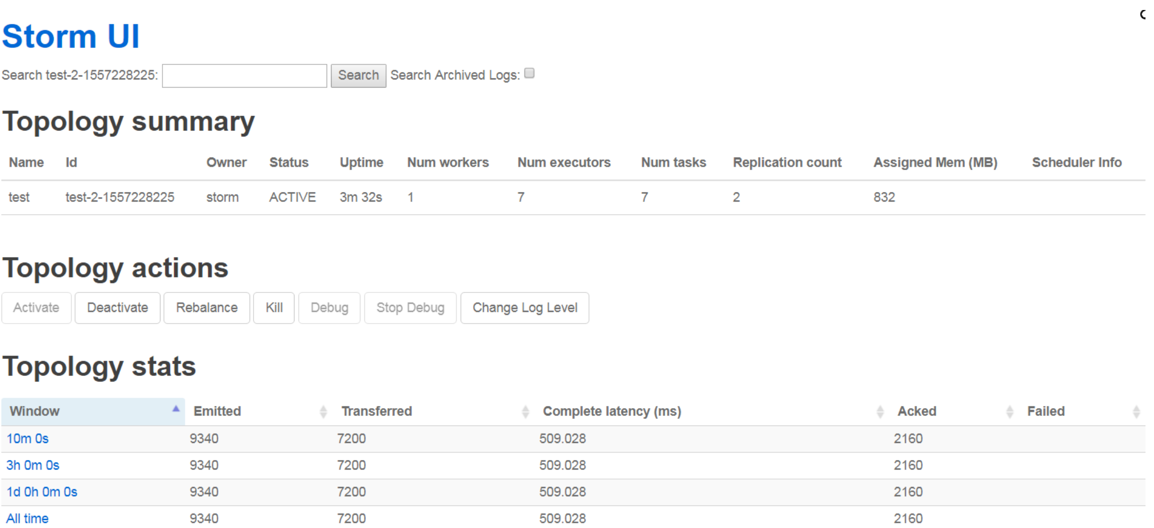
Topology Summary



Name	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)
test	storm	ACTIVE	6s	1	7	7	3	0

(4) 点击任务名称“test”即可进入该任务的监控页面，如图 2-9 所示，此时可查看 test 任务的执行状态、资源占用、各组件处理数据量以及延迟等 Storm 监控常用指标。

图2-9 Storm 拓扑任务监控指标



Storm UI

Search test-2-1557228225: Search Search Archived Logs:

Topology summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Assigned Mem (MB)	Scheduler Info
test	test-2-1557228225	storm	ACTIVE	3m 32s	1	7	7	2	832	

Topology actions

Activate Deactivate Rebalance Kill Debug Stop Debug Change Log Level

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	9340	7200	509.028	2160	
3h 0m 0s	9340	7200	509.028	2160	
1d 0h 0m 0s	9340	7200	509.028	2160	
All time	9340	7200	509.028	2160	

3 使用指南

3.1 常用命令

在大数据集群内安装 Storm 的任意一节点或安装 Storm Client 的节点，通过命令 ***storm help*** 可以查看支持的命令列表，如图 3-1 所示。

图3-1 Storm 支持的 shell 命令

```
[root@node1 /]# storm help
Commands:
        activate
        blobstore
        classpath
        deactivate
        dev-zookeeper
        drpc
        get-errors
        heartbeats
        help
        jar
        kill
        kill_workers
        list
        localconfvalue
        logviewer
        monitor
        nimbus
        node-health-check
        pacemaker
        rebalance
        remoteconfvalue
        repl
        set_log_level
        shell
        sql
        supervisor
        ui
        upload-credentials
        version
```

表3-1 Storm Shell 命令功能详解

命令	说明	语法
activate	激活指定的拓扑任务	storm activate topology-name
blobstore	执行blobstore相关命令。支持的具体命令可通过命令***storm help blobstore***的打印信息可查看	storm blobstore <cmd>
classpath	打印出Storm客户端运行命令时使用的类路径（classpath）	storm classpath
deactivate	禁用指定的拓扑	storm deactivate topology-name

命令	说明	语法
dev-zookeeper	以dev.zookeeper.path配置的值作为本地目录，以storm.zookeeper.port配置的值作为端口，启动一个新的ZooKeeper，一般仅用在开发/测试时	storm dev-zookeeper
drpc	启动一个DRPC守护进程	storm drpc
get-errors	获取指定拓扑的最新错误信息	storm get-errors topology-name
heartbeats	列出/获取集群中指定目录下的心跳数据	storm heartbeats list PATH storm heartbeats get PATH
help	打印一条帮助消息或者可用命令的列表，直接输入不带参数的storm，也可以启动storm help命令	storm help storm help <command>
jar	运行类的指定参数的main方法，向集群提交Topology就是通过该命令	storm jar topology-jar-path Mainclass args
kill	杀死名为topology-name的拓扑，Storm首先会在拓扑的消息超时时间期间禁用Spout，以允许所有正在处理的消息完成处理。然后，Storm将会关闭Worker并清理它们的状态。可以使用-w标记覆盖Storm在禁用与关闭期间等待的时间长度	storm kill topology-name [-w wait-time-secs]
kill_workers	杀死运行在当前Supervisor上的所有Worker进程。该命令需要在Supervisor节点上执行，如果集群运行在安全模式下，用户需要拥有该节点的管理权限	storm kill_workers
list	列出正在运行的拓扑及其状态	storm list
localconfvalue	打印出本地Storm配置的conf-name的值	storm localconfvalue conf-name
logviewer	启动Logviewer守护进程，Logviewer提供一个Web接口查看Storm日志文件	storm logviewer
monitor	监控指定拓扑的吞吐量信息，其中： • [-i interval-secs]：指定监听间隔时间，默认为 4s • [-m component-id]：组件 id，默认为所有组件 id • [-s stream-id]：流 id，默认为 default • [-w [emitted | transferred]：监控事件，默认监控 emitted	storm monitor topology-name [-i interval-secs] [-m component-id] [-s stream-id] [-w [emitted | transferred]]
nimbus	启动Nimbus守护进程	storm nimbus
node-health-check	在Supervisor本地执行健康检查	storm node-health-check
pacemaker	启动pacemaker守护进程	storm pacemaker
rebalance	重新分配集群中每个Supervisor节点上工作的Worker 例如：假设在一个10节点的集群中，每个节点运行了4个Worker，当新增另外10个节点到集群中后，可能希望有Spout扩散正在运行中的拓扑的Worker，以实现每个节点运行两个Worker。此时通过rebalance命令即可直接实现上述需求，操作	storm rebalance topology-name [-w wait-time-secs]

命令	说明	语法
	简单（另一种解决方法：杀死拓扑并重新提交拓扑，但比较麻烦）	
remoteconfvalue	打印出远程集群Storm配置的conf-name的值。集群Storm配置是$STORM-PATH/conf/storm.yaml与defaults.yaml合并的结果	storm remoteconfvalue conf-name
repl	打开一个包含类路径（classpath）中的jar文件和配置的Clojure REPL，以便调试时使用。Clojure可以作为一种脚本语言，但是Clojure的首选编程方式是使用REPL，REPL是一个简单的命令行接口。使用REPL，可以输入命令并执行，然后查看结果	storm repl
set_log_level	动态调整日志级别，其中： • [LEVEL]：调整后的日志级别 • [TIMEOUT]：动态日志级别持续时间	storm set_log_level -l [logger name]=[LEVEL]:[TIMEOUT]
shell	执行Shell脚本	storm shell resourcesdir command args
sql	将SQL语句编程为Trident拓扑并提交到Storm集群中。其中： • <sql-file>包含 SQL 语句列表 • <topology-name>为转换后的拓扑名	storm sql <sql-file> < topology-name>
supervisor	启动Supervisor守护进程	storm supervisor
ui	启动UI守护进程。UI为Storm集群提供了一个Web界面并显示运行拓扑的详细统计信息	storm ui
upload-credentials	上传指定拓扑的信任列表	storm upload_credentials topology-name [credkey credvalue]*
version	打印Storm发布的版本号	storm version

3.2 Client下载/安装/使用/卸载

大数据集群提供了下载 Storm Client 的功能。在客户端节点上安装 Storm 的 Client 后，即可直接连接大数据集群中的 Storm，进行组件维护、任务管理等操作。

3.2.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在[集群管理/集群列表]页面，单击集群名称进入集群详情页面，在集群已安装的组件列表中单击 Storm 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-2 所示。

图3-2 下载 Client 窗口

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：
  - 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
  - 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。
  - 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
  - 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.2.2 安装 Client

注意

- 安装 Client 的节点必须与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置网络连接，仅非 root 用户需要执行此操作，root 用户可跳过此步骤。

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

./install.sh <安装路径> -o

说明

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为 /usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>

3.2.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

  source bigdata_env

- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同：
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行校验即可直接访问组件并执行相关操作。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行身份认证之后，才可访问组件并执行相关操作。关于 Kerberos 环境下，进行用户认证的操作方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。Kerberos 环境下，在集群外的 Client 节点上执行 Storm 任务时，需要修改任务代码，详情请参考 5 最佳实践章节中的 Kerberos 场景相关说明。

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

3.2.4 卸载 Client 客户端

大数据集群重装之后，之前安装的 Client 客户端将不可用。此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 安装目录下执行以下命令：

./uninstall.sh

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

3.3 Storm集群扩容

Storm 集群扩容是指在某节点上新增安装 Supervisor。

3.3.1 使用场景

随着业务量的增长，集群计算能力无法满足业务需求时，需要考虑对 Storm 集群进行扩容。

3.3.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在大数据平台上新增安装 Supervisor 进程。

  - 如果集群中有节点没有安装 Supervisor，直接在集群节点中添加 Supervisor 进程。
  - 如果集群中所有节点均已安装 Supervisor，进行 Supervisor 扩容前则需要先添加主机。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Storm 组件的状态是否正常。

(2) 进入 Storm 组件详情页，查看 Storm 的部署拓扑，确保每个服务的状态正常，Nimbus、Storm UI Server、DRPC Server、Supervisor 处于“已启动”状态。

3.3.3 扩容约束

- 扩容节点操作系统版本与集群内部版本需保持一致。
- 扩容操作一旦开始，不支持中止。

3.3.4 扩容影响

- 一旦扩容失败，需要及时将扩容失败的节点剔除。
- 扩容成功后，集群计算资源增多。

3.3.5 扩容操作指导

注意

若集群中所有节点均已安装 Supervisor，进行 Supervisor 扩容前则需要先添加主机，然后再进行 Supervisor 扩容。如果集群中有扩容所用主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 Storm 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-3所示。

  a. 选择进程及主机

  在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

  b. 部署进程

  选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

  c. 启动进程

  部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-3 添加进程

(3) 查看进程变化

Supervisor 扩容完成之后，在组件详情页面[部署拓扑]页签中可以查看 Supervisor 进程的数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.3.6 扩容验证

(1) 登录大数据平台管理系统。
(2) 执行 Storm 组件检查，确保 Storm 组件可正常使用，详情请参考 2.2 2. 组件检查。
(3) 查看 Storm 组件部署拓扑里是否已有新增的扩容节点。
(4) 打开 Storm 快速链接，在 Storm UI 页面查看 Supervisor 最新信息是否符合预期。

3.4 Storm集群缩容

Storm 集群缩容是指将某节点上已安装的 Supervisor 删除。

3.4.1 使用场景

Storm 集群缩容的场景主要有：

- 初始 Supervisor 节点规划不合理。
- 当 Supervisor 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.4.2 缩容前准备

1. 缩容规划

缩容可以删除任意 Supervisor 节点。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Storm 组件的状态是否正常。
(2) 进入 Storm 组件详情页，查看 Storm 的部署拓扑，确保每个服务的状态正常，Nimbus、Storm UI Server、DRPC Server、Supervisor 处于“已启动”状态。

3.4.3 缩容约束

- 在生产环境中，缩容不可回退或暂停，请谨慎使用。
- 执行缩容前，请检查 Supervisor 中是否有正在运行的任务，如有正在运行的任务，缩容时将会影响任务的执行。

3.4.4 缩容影响

- 为保证业务安全，一般不建议对 Storm 进行缩容。

3.4.5 缩容操作指导

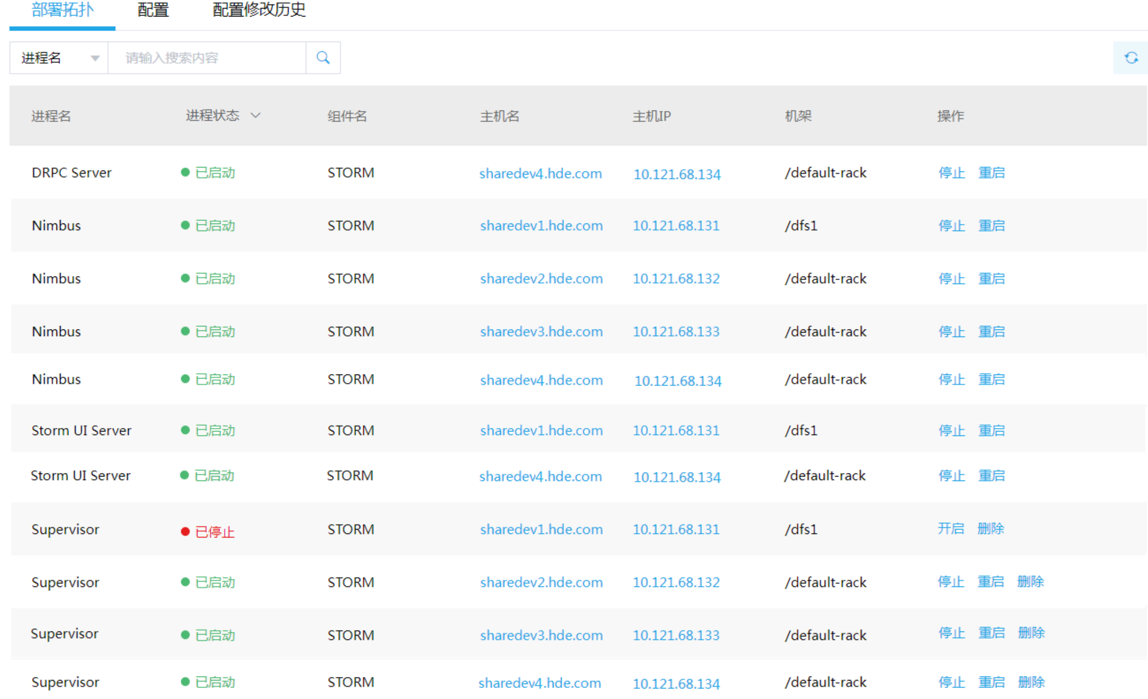
说明

Supervisor 缩容操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行 Supervisor 缩容操作”为例进行说明，在主机详情页面执行 Supervisor 缩容操作，与其类似不再进行说明。

缩容操作步骤如下：

(1) 在 Storm 组件详情页面，选择缩容节点。
(2) 在组件详情页面[部署拓扑]页签下，选择已安装 Supervisor 进程且需要缩容的主机，然后单击该进程右侧操作中的<停止>按钮，停止 Supervisor。
(3) 删除 Supervisor
待 Supervisor 停止成功后，如图 3-4 所示，在该进程右侧操作中单击<删除>按钮，即可完成 Supervisor 的缩容。

图3-4 删除进程



进程名	进程状态	组件名	主机名	主机IP	机架	操作
DRPC Server	已启动	STORM	sharedev4.hde.com	10.121.68.134	/default-rack	停止 重启
Nimbus	已启动	STORM	sharedev1.hde.com	10.121.68.131	/dfs1	停止 重启
Nimbus	已启动	STORM	sharedev2.hde.com	10.121.68.132	/default-rack	停止 重启
Nimbus	已启动	STORM	sharedev3.hde.com	10.121.68.133	/default-rack	停止 重启
Nimbus	已启动	STORM	sharedev4.hde.com	10.121.68.134	/default-rack	停止 重启
Storm UI Server	已启动	STORM	sharedev1.hde.com	10.121.68.131	/dfs1	停止 重启
Storm UI Server	已启动	STORM	sharedev4.hde.com	10.121.68.134	/default-rack	停止 重启
Supervisor	已停止	STORM	sharedev1.hde.com	10.121.68.131	/dfs1	开启 删除
Supervisor	已启动	STORM	sharedev2.hde.com	10.121.68.132	/default-rack	停止 重启 删除
Supervisor	已启动	STORM	sharedev3.hde.com	10.121.68.133	/default-rack	停止 重启 删除
Supervisor	已启动	STORM	sharedev4.hde.com	10.121.68.134	/default-rack	停止 重启 删除

(4) 查看进程变化

Supervisor 缩容完成之后，在组件详情页面[部署拓扑]页签中可以查看 Supervisor 进程的数量变化以及状态。

(5) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.4.6 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Storm 组件检查，确保 Storm 组件可正常使用，详情请参考 2.2 2. 组件检查。

(3) 查看 Storm 组件部署拓扑里相关缩容节点是否已经删除。

(4) 打开 Storm 快速链接，在 Storm UI 页面查看 Supervisor 最新信息是否符合预期。

4 开发指南

说明

大数据平台中的 Storm 完全兼容 Apache 社区 Storm 1.2.1 版本的 API，关于 Storm 详细接口信息请参见官网。

Storm 作为常用的流式处理框架，支持大部分数据源服务如：Kafka、HBase、HDFS、Hive、Solr、Cassandra、JDBC、Redis、Elasticsearch、openTSDB、Druid 等。

同时，理论上 Storm 可支持各种编程语言，默认已经支持的语言包括 Clojure、JAVA、Ruby 和 Python。若需要增加对其他语言的支持，需实现相应的 Storm 通信协议。

4.1 Java API

Storm Topology 主要使用到如下这几个类：

- TopologyBuilder：Topology 的构建器，定义一个 Topology 由哪些 Spout、Bolt 组成及其消息流动的路径。常用接口请参见表 4-1。
- Config：Topology 运行的配置。常用接口及其说明请参见表 4-2。
- StormSubmitter：向集群提交 Topology 的工具类。常用接口及其说明请参见表 4-3。
- LocalCluster：本地模式运行 Topology 的工具类。常用接口及其说明请参见表 4-4。
- BaseRichSpout：定义数据源可以继承该类，实现其必要的函数即可。常用接口及其说明请参见表 4-5。
- BaseRichBolt：定义数据处理 Bolt 可以继承该类，实现其必要函数即可。常用接口及其说明请参见表 4-6。
- SpoutConfig：集成 Kafka 开发时，KafkaSpout 的构造参数配置类。常用配置及其说明请参见表 4-7。

表4-1 TopologyBuilder

方法	说明
setSpout(String spoutId, Class < extends BaseRichSpout > cls, int taskNum)	参数1：Spout实例的Id 参数2：Spout实例，用户自定义BaseRichSpout子类 参数3：分配该Spout实例的并发数，该并发数将会控制该实例在集群中并行执行的线程数
setBolt(String boltId, Class < extends BaseRichBolt > cls, int taskNum)	参数1：Bolt实例的Id 参数2：具体Bolt，用户自定义BaseRichBolt子类 参数3：分配该Bolt实例的并发数，该并发数将会控制该实例在集群中并行执行的线程数
createTopology()	创建Topology的函数

表4-2 Config

方法	说明
setDebug（boolean ifDebug）	如果设置成true，Storm会记录下每个组件所发射的每条消息。此方法主要用于本地环境调试Topology，在线上使用会影响性能
setNumWorkers(int nums)	控制集群来执行这个topology分配的工作进程个数
setMaxTaskParallelism(int nums)	设置一个组件最多能够分配的Executor 数（线程数上限），一般用于在本地模式运行拓扑时测试分配线程的数量限制
setMaxSpoutPending(int max)	设置当前Storm应用程序的活跃batch数据。该配置用于缓存Spout发送出去的Tuple，即当下游的Bolt还有max个Tuple没有消费完时，Spout会停下来等待下游Bolt去消费。该配置可降低Storm吞吐量来保证消息可靠处理

表4-3 StormSubmitter

方法	说明
submitTopologyWithProgressBar(String topologyId, Config conf, StormTopology topology);	向集群提交一个Topology，为Topology指定一个唯一的Id（该Id如果在集群中被使用，需先kill掉对应Topology，然后才能成功提交），其中： • conf：运行时的配置 • StormTopology：由 TopologyBuilder 的 createTopology() 函数创建

表4-4 LocalCluster

方法	说明
submitTopology (String topologyId, Config conf, StormTopology topology);	本地模式运行Topology，为Topology指定一个Id，其中： • conf：运行时的配置 • StormTopology：由 TopologyBuilder 的 createTopology() 函数创建

表4-5 BaseRichSpout

方法	说明
open(Map conf, TopologyContext context, SpoutOutputCollector outputCollector)	open方法进行Spout类的初始化工作
nextTuple()	在SpoutTracker类中被调用，每调用一次就可以向Storm集群中发射一条数据（一个Tuple元组），该方法会被不停的调用
declareOutputFields(OutputFieldsDeclarer declarer)	定义发射出去的消息的字段Id，该Id在简单模式下没有用处，但在按照字段分组的模式下有很大的用处

表4-6 BaseRichBolt

方法	说明
prepare(Map map, TopologyContext topologyContext, OutputCollector outputCollector)	prepare方法进行Bolt类的初始化工作 例如：连接数据库时，需要进行一次数据库连接操作，我们就可以把该操作放入prepare中，只需要执行一次
execute(Tuple tuple)	处理接收到的消息，并发射新的消息
declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer)	定义发射出去的消息的字段Id，该Id在简单模式下没有用处，但在按照字段分组的模式下有很大的用处

表4-7 SpoutConfig (BrokerHosts hosts, String topic, String zkRoot, String id)

构造函数参数配置	说明
Hosts	构造ZKHosts实例，添加Broker的ZooKeeper地址以及Broker在ZooKeeper中的路径默认为/brokers
Topic	消费订阅的Kafka的topic
zkRoot	在ZooKeeper中存储消费进度信息的根路径。建议设置为"/consumers"
Id	进度记录ID可以理解为Kafka 消费组的groupId

4.2 Restful API

Storm UI 进程提供了 REST API 可以和 Storm 集群进行交互，包含获取性能指标、配置信息和管理操作（启动、停止 Topology 任务）等。

REST API 接口通过 HTTP 请求执行，HTTP 请求的格式如下：

http://<STORM-UI_IP>:<STORM-UI_Port><Path>

其中：

- STORM-UI_IP 指 STORM UI 进程所在节点的 IP 地址
- STORM-UI_Port 指 STORM-UI 进程的监听端口
- Path 指路径，例如：http://IP:8774/api/v1/cluster/configuration。更多关于 Path 路径的信息请参见表 4-8 所示。

表4-8 Path 路径说明

Path	说明
/api/v1/cluster/configuration	返回集群配置
/api/v1/cluster/summary	返回集群概要信息，如：Nimbus运行时间、Supervisor节点数
/api/v1/supervisor/summary	返回所有Supervisor节点的概要信息
/api/v1/nimbus/summary	返回所有Nimbus节点的概要信息
/api/v1/history/summary	返回当前用户提交的Topology任务列表

Path	说明
/api/v1/supervisor	返回指定主机或指定ID的Supervisor概要信息
/api/v1/topology/summary	返回所有Topology任务的概要信息
/api/v1/topology-workers/:id	返回指定Topology任务的Worker信息（主机和端口）
/api/v1/topology/:id	返回指定Topology任务的拓扑信息和统计
/api/v1/topology/:id/metrics	返回指定Topology任务的明细指标
/api/v1/topology/:id/component/:component	返回指定Topology任务组件的明细指标和执行信息
/api/v1/topology/:id/profiling/start/:host-port/:timeout	在指定Topology上的指定Worker上开启profiler跟踪，返回状态和跟踪情况的链接
/api/v1/topology/:id/profiling/dumpprofile/:host-port	在指定Topology上的指定Worker上dump profiler跟踪，返回状态和Worker的ID
/api/v1/topology/:id/profiling/stop/:host-port	在指定Topology上的指定Worker上停止profiler跟踪，返回状态和Worker的ID
/api/v1/topology/:id/profiling/dumpjstack/:host-port	在指定Topology上的指定Worker上dump jstack，返回状态和Worker的ID
/api/v1/topology/:id/profiling/dumpheap/:host-port	在指定Topology上的指定Worker上dump heap，返回状态和Worker的ID
/api/v1/topology/:id/profiling/restartworker/:host-port	请求重启指定Topology上的指定Worker，返回状态和Worker的ID
/api/v1/topology/:id/activate	激活指定Topology任务（POST请求）
/api/v1/topology/:id/deactivate	关闭指定Topology任务（POST请求）
/api/v1/topology/:id/rebalance/:wait-time	重新平衡指定Topology任务（POST请求）
/api/v1/topology/:id/kill/:wait-time	杀掉指定Topology任务（POST请求）

4.3 JavaAPI基础开发样例

1. 样例场景

一个动态单词统计系统，数据源为持续生产随机文本的逻辑单元，业务处理流程如下：

- 数据源持续不断地发送随机文本给文本拆分逻辑，如“apple orange apple”。
- 单词拆分逻辑将数据源发送的每条文本按空格进行拆分，如“apple”、“orange”、“apple”，随后将每个单词逐一发给单词统计逻辑。
- 单词统计逻辑每收到一个单词就进行加一操作，并将实时结果打印输出，如：

  apple：1

  orange：1

  apple：2

2. 开发思路

根据上述场景进行功能分解，如表 4-9 所示。

表4-9 功能分解

序号	步骤
1	创建一个Spout用来生成随机文本
2	创建一个Bolt用来将收到的随机文本拆分成一个个单词
3	创建一个Blot用来统计收到的各单词次数
4	创建Topology

3. 样例代码

(1) Pom 文件依赖

```
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.2.1</version>
    <scope>provided</scope>
</dependency>
```

(2) 创建 Spout

Spout 是 Storm 的消息源，它是 Topology 的消息生产者，一般来说消息源会从一个外部源读取数据并向 Topology 中发送消息（Tuple）。

一个消息源可以发送多条消息流，可以使用 OutputFieldsDeclarer.declarerStream 来定义多个 Stream，然后使用 SpoutOutputCollector 来发射指定的 Stream。

下面代码片段实现功能为：从字符串数组中随机抽取一条字符串并发送到下游。

```
import org.apache.storm.spout.SpoutOutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;
import java.util.Map;
import java.util.Random;

public class WordCountSpout extends BaseRichSpout{
    SpoutOutputCollector _collector;
Random random;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector
collector){
        _collector = collector;
        random = new Random();
    }

    @Override
    public void nextTuple() {
```

```
            Utils.sleep(100);
            String[] sentences = new String[] {"the cow jumped over the moon",
                            "an apple a day keeps the doctor away",
                            "four score and seven years ago",
                            "snow white and the seven dwarfs",
                            "i am at two with nature"};
            String sentence = sentences[random.nextInt(sentences.length)];
            _collector.emit(new Values(sentence));
        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
    }
```

(3) 创建 Bolt

所有的消息处理逻辑都被封装在各个 Bolt 中。Bolt 包含多种功能，如：过滤、聚合等。如果 Bolt 之后还有其他拓扑算子，可以使用 OutputFieldsDeclarer.declareStream 定义 Stream，使用 OutputCollector.emit 来选择要发射的 Stream。

下面 SplitSentenceBolt 类实现功能为：拆分每条语句为单个单词并发送到下游。

```
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;
import java.util.Map;

public class SplitSentenceBolt extends BaseRichBolt {
    private OutputCollector collector;
    public void execute(Tuple input) {
        // TODO Auto-generated method stub
        String line = input.getStringByField("word");
        String[] words = line.split(" ");
        for (String word : words){
            collector.emit(new Values(word));
        }
        this.collector.ack(input);
    }

    public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector) {
        // TODO Auto-generated method stub
        this.collector = collector;
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // TODO Auto-generated method stub
        declarer.declare(new Fields("splitedWord"));
    }
}
```

下面 WordCounBolt 类实现功能为：接收上游数据并统计各单词的出现次数。

```
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;
import java.util.HashMap;
import java.util.Map;

public class WordCounBolt extends BaseRichBolt {
    private OutputCollector collector;
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void execute(Tuple input) {
        // TODO Auto-generated method stub
        String word = input.getString(0);
        Integer count = counts.get(word);
        if (null == count){
            count = 0;
        } else {
            count ++;
            counts.put(word, count);
            collector.emit(new Values(word, count));
        }

        this.collector.ack(input);
    }

    public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector) {
        // TODO Auto-generated method stub
        this.collector = collector;
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // TODO Auto-generated method stub
        declarer.declare(new Fields("word", "count"));
    }
}
```

下面 WordReportBolt 类实现功能为：接收上游统计结果并将结果打印出来。

```
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Tuple;
import java.util.*;

public class WordReportBolt extends BaseRichBolt {
    private OutputCollector collector;
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void execute(Tuple input) {
        // TODO Auto-generated method stub
```

```
            String word = input.getStringByField("word");
            Integer count = input.getIntegerByField("count");
            this.counts.put(word, count);
            this.collector.ack(input);
        }

        public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector) {
            // TODO Auto-generated method stub
            this.collector = collector;
        }

        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            // TODO Auto-generated method stub
        }

        @Override
        public void cleanup() {
            System.out.println("--------FINAL COUNTS START-----------");
            List<String> keys = new ArrayList<String>();
            keys.addAll(this.counts.keySet());
            Collections.sort(keys);
            for (String key : keys) {
                System.out.println(key + " : " + this.counts.get(key));
            }
            System.out.println("-----------FINAL COUNTS END------------");
        }
}
```

(4) 创建 Topology

Topology 是 Spouts 和 Bolts 组成的有向无环图。应用程序通过 storm jar 方式提交，因此需要在 main 函数中调用创建 Topology 的函数，并在 storm jar 参数中指定 main 函数所在类。

下面代码片段实现功能为：构建应用程序并提交。

```
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.generated.AlreadyAliveException;
import org.apache.storm.generated.AuthorizationException;
import org.apache.storm.generated.InvalidTopologyException;
import org.apache.storm.topology.TopologyBuilder;

public class WordCount {
    public static void main(String[] args){
        if (args.length != 1){
            System.out.println("Usage: <topic name>");
            System.exit(1);
        }

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new WordCountSpout(), 3);
        builder.setBolt("split", new SplitSentenceBolt(),4).shuffleGrouping("spout");
        builder.setBolt("count", new WordCounBolt(), 5).shuffleGrouping("split");
        builder.setBolt("report", new WordReportBolt(), 6).shuffleGrouping("count");

        Config config = new Config();
```

```
        if (args != null && args.length > 0){
            config.setNumWorkers(3);
            try {
                StormSubmitter.submitTopologyWithProgressBar(args[0], config, 
builder.createTopology());
            } catch (AuthorizationException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }catch (AlreadyAliveException e) {
                // TODO: handle exception
                e.printStackTrace();
            }catch (InvalidTopologyException e2) {
                // TODO: handle exception
                e2.printStackTrace();
            }
        } else {
            config.setMaxTaskParallelism(3);
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("word-count",config,builder.createTopology());
        }
    }
}
```

(5) 提交拓扑

a. 登录大数据集群内安装 Storm 的任意一节点或安装 Storm Client 的节点。

b. 将准备依赖的 Jar 包和配置文件上传到任意目录下。

c. 提交拓扑，命令如下：

   storm jar <jar 包路径> <入口类名> <该拓扑名>

d. 任务提交成功以后，参考 2.4 快速链接章节可查看任务执行情况。

5 最佳实践

5.1 Storm-Kafka开发样例

1. 使用场景

流计算业务场景中，数据通常缓存在 Kafka 中，消费 kafka 数据是流计算业务中最常见的场景之一。本章节通过简单的 WordCount 样例来说明 Storm 消费 Kafka 的 API 调用过程。

2. 创建 Kafka Topic

创建名为 input 的 Topic，命令如下：

/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --create

--zookeeper node1:2181,node2:2181,node3:2181

--topic input

--partitions 3

--replication-factor 3

其中：ZooKeeper 节点的 IP：Port 请以实际集群为准，端口默认为 2181。

3. 模拟发送消息到 Topic（Topic 名为 input）

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;
import java.util.Random;

public class ProducerClient {
    private static final String[] PROVINCES_CITIES = new String[]{
            "the cow jumped over the moon",
            "an apple a day keeps the doctor away",
            "four score and seven years ago",
            "snow white and the seven dwarfs",
            "i am at two with nature";

    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put("bootstrap.servers", "node1:6667,node2:6667,node3:6667");
        props.put("acks", "-1");
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        KafkaProducer<String, String> kafkaProducer = new KafkaProducer(props);
        boolean flag = true;
        if (flag) {
            for (int i = 0; i < 2000; i++) {
                String sentence = sentences[random.nextInt(sentences.length)];
```

```
                kafkaProducer.send(new ProducerRecord("input", sentence));
            }
            Thread.sleep(1000);
            kafkaProducer.flush();
        }
        kafkaProducer.close();
    }
}
```

4. 编写 Storm 程序

- Pom 文件依赖

  【说明】：以下 pom 依赖 jar 包的版本以大数据平台中对应组件的版本为基准。

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>0.10.0.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
        </exclusion>
        <exclusion>
            <artifactId>slf4j-api</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
        <exclusion>
            <artifactId>log4j</artifactId>
            <groupId>log4j</groupId>
        </exclusion>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-kafka</artifactId>
    <version>1.1.1</version>
    <exclusions>
        <exclusion>
            <artifactId>slf4j-api</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
        <exclusion>
            <groupId>org..log4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.2.1</version>
    <scope>provided</scope>
```

```
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>log4j-over-slf4j</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org..log4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.zookeeper/zookeeper -->
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.6</version>
    <type>pom</type>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

- Spout

  Storm 消费 Kafka 时直接使用 KafkaSpout 对象即可，此时仅需设置对应配置项，无需自定义实现 Spout。

- Bolt

  直接使用 4.3 3. (3)创建 Bolt 章节中的 SplitSentenceBolt、WordCounBolt 和 WordReportBolt 即可。

- Topology

```
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.generated.AlreadyAliveException;
import org.apache.storm.generated.AuthorizationException;
import org.apache.storm.generated.InvalidTopologyException;
import org.apache.storm.topology.TopologyBuilder;

public class WordCountTopology {
    public static void main(String[] args){
```

```
        if (args.length != 1){
            System.out.println("Usage: <topic name>");
            System.exit(1);
        }

      String topicName = "input"
      String brokerZkStr  = "node1:2181,node2:2181,node3:2181";//zk 节点信息
      String id = "storm_kafka";
      String brokerList = " node1:6667,node2:6667,node3:6667";//kafka broker 地址 String
      String kafkaSer = "kafka.serializer.StringEncoder";
      String groupId = "groupId_1";
      BrokerHosts brokerHosts = new ZkHosts(brokerZkStr);
      SpoutConfig spoutConfig = new SpoutConfig(brokerHosts, topicName, "/consumers",
id);

      Config conf = new Config();
      Properties props = new Properties();
      props.put("metadata.broker.list", brokerList);
      props.put("serializer.class", kafkaSer);
      props.put("group.id", groupId);
      conf.put("kafka.broker.properties", props);
      conf.put("topic", topicName);
      spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
      KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new kafkaSpout (), 3);
        builder.setBolt("split", new SplitSentenceBolt(),2).shuffleGrouping("spout");
        builder.setBolt("count", new WordCounBolt(), 3).shuffleGrouping("split");
        builder.setBolt("report", new WordReportBolt(), 2).shuffleGrouping("count");

        Config config = new Config();
        if (args != null && args.length > 0){
            config.setNumWorkers(3);
            try {
                StormSubmitter.submitTopologyWithProgressBar(args[0], config,
builder.createTopology());
            } catch (AuthorizationException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }catch (AlreadyAliveException e) {
                // TODO: handle exception
                e.printStackTrace();
            }catch (InvalidTopologyException e2) {
                // TODO: handle exception
                e2.printStackTrace();
            }
        } else {
            config.setMaxTaskParallelism(3);

            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("word-count",config,builder.createTopology());
        }
    }
}
```

5. 提交拓扑

(1) 登录大数据集群内安装 Storm 的任意一节点或安装 Storm Client 的节点。

(2) 将准备依赖的 Jar 包和配置文件上传到任意目录下。

(3) 提交拓扑，命令如下：

```
storm jar <jar 包路径> <入口类名> <该拓扑名>
```

(4) 任务提交成功以后，参考 2.4 快速链接章节可查看任务执行情况。

6. Kerberos 场景

说明

- Kerberos 场景普通用户提交 Storm 作业时，需检查普通用户 keytab 文件属性及权限。
- 更多 Kerberos 场景下，Storm 使用问题可参考官网。

大数据集群开启 Kerberos 之后，在运行任务前还需要修改代码，提交 Topolopy 增加权限认证，增加的代码内容如下：

```
System.setProperty("java.security.auth.login.config","/绝对路径/jaas.conf");
System.setProperty("java.security.krb5.conf","/etc/krb5.conf");
```

其中：

- jaas.conf 文件需要提前准备，krb5.conf 文件文件由集群自动生成（即/etc/krb5.conf）。
- jaas.conf 文件示例如下，jaas.conf 文件的 KafkaClient 模块 keytab 与 principal 配置普通用户，创建 topic 时需要指定权限给普通用户；或者使用 Kafka 管理用户的 keytab、principal 创建 topic，此时不需要指定即可直接使用。

```
StormClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   keyTab="/etc/security/keytabs/test_user.keytab "
   storeKey=true
   useTicketCache=false
   serviceName="nimbus"
   principal=" test_user.keytab@HDE.COM";
};
KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   keyTab="/etc/security/keytabs/test.keytab"
   storeKey=true
   useTicketCache=false
   serviceName="kafka"
   principal="test@HDE.COM";
};
```

5.2 Storm-Redis开发样例

1. 使用场景

对基站产生的通话数据进行总量统计。将数据保存在本地，首先需要使用 Flume 将数据拉取到 Kafka 内，然后通过 Storm 消费数据并进行聚合统计，同时将结果写入到 Reids 中。

2. 数据格式

number String, id String, city String, timestamp String

【说明】：本章节场景中，需要提取每次通话数据中的电话号码信息，并保存到 Redis 中。

3. Flume 配置

说明

进行 Flume 配置时，要求在集群中已安装 Flume。

在 Flume 的 flume.conf 配置项中添加如下内容：

```
a1.sources = r1
a1.sinks = k1
a1.channels = c1
# Describe/configure the source
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir=/opt/stream/                    # 该路径用户保存源数据，可自定义
a1.sources.r1.fileHeader = true
# Describe the sink
a1.sinks.k1.type =org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.brokerList=node1:6667,node2:6667 # kafka broker 节点信息，根据实际集群配置
a1.sinks.k1.topic=topic_number                         # kafka 用户保存数据的 topic
a1.sinks.k1.serializer.class=kafka.serializer.StringEncoder
# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 10000
a1.channels.c1.transactionCapacity = 10000
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

4. 创建 Kafka topic

创建名为 topic_number 的 Topic，命令如下：

```
/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --create
--zookeeper node1:2181,node2:2181,node3:2181
```

--topic topic_number

--partitions 3

--replication-factor 3

其中：ZooKeeper 节点的 IP：Port 请以实际集群为准，端口默认为 2181。

5. 编写 Storm 程序

- Pom 文件依赖

  【说明】：以下 pom 依赖 jar 包的版本以大数据平台中对应组件的版本为基准。

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>0.10.0.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
        </exclusion>
        <exclusion>
            <artifactId>slf4j-api</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
        <exclusion>
            <artifactId>log4j</artifactId>
            <groupId>log4j</groupId>
        </exclusion>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-kafka</artifactId>
    <version>1.2.1</version>
    <exclusions>
        <exclusion>
            <artifactId>slf4j-api</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
        <exclusion>
            <groupId>org..log4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.2.1</version>
    <scope>provided</scope>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
```

```
                <artifactId>log4j-over-slf4j</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org.apache.logging.log4j</groupId>
                <artifactId>log4j-slf4j-impl</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org..log4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.zookeeper/zookeeper -->
    <dependency>
        <groupId>org.apache.zookeeper</groupId>
        <artifactId>zookeeper</artifactId>
        <version>3.4.6</version>
        <type>pom</type>
        <exclusions>
            <exclusion>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
            </exclusion>
            <exclusion>
                <groupId>log4j</groupId>
                <artifactId>log4j</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>storm-redis</artifactId>
        <version>0.10.0</version>
        <type>jar</type>
    </dependency>
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>2.8.1</version>
    </dependency>
```

- Spout

  Storm 消费 Kafka 时直接使用 KafkaSpout 对象即可，此时仅需设置对应配置项，无需自定义实现 Spout。

- Bolt

```
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Tuple;
import redis.clients.jedis.HostAndPort;
```

```
import redis.clients.jedis.JedisCluster;
import java.util.HashSet;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class Storm2RedisBolt extends BaseRichBolt {
    private OutputCollector collector;
    private JedisCluster jc;

    public void execute(Tuple input) {
        String number = input.getString(0);

        String value = jc.get(number);
        if (null == value) {
            jc.set(number, "1");
        } else {
            int a = Integer.parseInt(value);
            a++;
            String tmp = String.valueOf(a);
            jc.set(number, tmp);
        }

        this.collector.ack(input);
    }

    public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector) {
        this.collector = collector;
        int port = 7001; //大数据平台当前版本默认为7001，老版本可能为6379
        String node1 = "node1";
        String node2 = "node2";
        String node3 = "node3";
        Set<HostAndPort> nodes = new HashSet<HostAndPort>();
        nodes.add(new HostAndPort(node1, port));
        nodes.add(new HostAndPort(node2, port));
        nodes.add(new HostAndPort(node3, port));

        jc = new JedisCluster(nodes, 5000, 5000);
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
    }
}
```

- Topology

```
import com.utils.ConfigMananger;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.generated.AlreadyAliveException;
import org.apache.storm.generated.AuthorizationException;
import org.apache.storm.generated.InvalidTopologyException;
import org.apache.storm.kafka.*;
import org.apache.storm.spout.SchemeAsMultiScheme;
import org.apache.storm.topology.TopologyBuilder;
import java.util.Properties;
```

```
public class Storm2RedisTopology {

    public static void main(String[] args) {
        if (args.length != 1){
            System.out.println("Usage: <topic name>");
            System.exit(1);
        }

        String topicName = "topic_number";
        String brokerZkStr  = "node1:2181,node2:2181,node3:2181";
        String id = "storm_redis";
        String brokerList = " node1:6667,node2:6667,node3:6667";
        String kafkaSer = "kafka.serializer.StringEncoder";
        String groupId = "groupId";
        BrokerHosts brokerHosts = new ZkHosts(brokerZkStr);
        SpoutConfig spoutConfig = new SpoutConfig(brokerHosts, topicName, "/consumers",
id);

        Config conf = new Config();
        Properties props = new Properties();
        props.put("metadata.broker.list", brokerList);
        props.put("serializer.class", kafkaSer);
        props.put("group.id", groupId);
        conf.put("kafka.broker.properties", props);
        conf.put("topic", topicName);
        spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
        KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("msgKafkaSpout", kafkaSpout, 3);
        builder.setBolt("Store2RedisBolt",
                        new Storm2RedisBolt(), 3).shuffleGrouping("msgKafkaSpout");

        if (args != null && args.length > 0){
            conf.setNumWorkers(numWorkers);
            try {
                StormSubmitter.submitTopologyWithProgressBar(args[0], conf,
builder.createTopology());
            } catch (AuthorizationException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }catch (AlreadyAliveException e) {
                // TODO: handle exception
                e.printStackTrace();
            }catch (InvalidTopologyException e2) {
                // TODO: handle exception
                e2.printStackTrace();
            }
        }else {
            conf.setMaxSpoutPending(3);

            LocalCluster cluster = new LocalCluster();

cluster.submitTopology("topicMsgTopology",conf,builder.createTopology());
        }
```

```
        }
    }
```

6. 提交拓扑

(1) 登录大数据集群内安装 Storm 的任意一节点或安装 Storm Client 的节点。

(2) 将准备依赖的 Jar 包和配置文件上传到任意目录下。

(3) 提交拓扑，命令如下：

storm jar <jar 包路径> <入口类名> <该拓扑名>

(4) 任务提交成功以后，参考 2.4 快速链接章节可查看任务执行情况。

7. Kerberos 场景

说明

- Kerberos 场景普通用户提交 Storm 作业时，需检查普通用户 keytab 文件属性及权限。
- 更多 Kerberos 场景下，Storm 使用问题可参考官网。

大数据集群开启 Kerberos 之后，在运行任务前还需要修改代码，提交 Topolopy 增加权限认证，增加的代码内容如下：

```
System.setProperty("java.security.auth.login.config","/绝对路径/jaas.conf");
System.setProperty("java.security.krb5.conf","/etc/krb5.conf");
```

其中：

- jaas.conf 文件需要提前准备，krb5.conf 文件文件由集群自动生成（即/etc/krb5.conf）。
- jaas.conf 文件示例如下：

```
StormClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   keyTab="/etc/security/keytabs/test_user.keytab "
   storeKey=true
   useTicketCache=false
   serviceName="nimbus"
   principal=" test_user.keytab @HDE.COM";
};
```

5.3 Storm-HBase开发样例

1. 使用场景

对某个广告在某个省、某个市、某个用户的当前投放量做历史投放趋势或点击量等统计。首先需要将数据拉取到 Kafka 内，然后通过 Storm 消费 Kafka 内的数据并进行聚合统计，同时将结果写入到 HBase 中。

广告实时计算系统：

- 某个广告在某个省的当前投放量
- 某个广告在某个市的当前投放量
- 某个广告在某个用户客户端上的当前投放量

- 某个广告在累加一段时间内的某个省的历史投放趋势
- 某个广告在累加一段时间内的某个市的历史投放趋势
- 某个广告在累加一段时间内的某个客户端的历史投放趋势
- 某个广告的当前点击量
- 某个广告在累加一段时间内的点击趋势

2. 数据格式

示例：

2014-01-13\t19:11:55\t{"adid":"31789","uid":"9871","action":"view"}\t63.237.239.3\t 北京\t 北京

其中：

- 日期：2014-01-13
- 时间：19:11:55
- Json：方便扩展
    - adid：广告 ID
    - uid：用户 ID
    - action：用户行为 click、view
- IP：63.237.239.3
- 省：北京
- 市：北京

3. HBase 建表

- 表名：realtime_ad_stat
- 行键：ADID_Province_20181212 ADID_City_20181212 ADID_UID_20181212
- 列簇：stat
- 列：view_cnt、click_cnt

说明

HBase 建表的相关命令如下：

- 创建表

  create 'realtime_ad_stat',{NAME => 'stat', VERSIONS =>3}
- 查看表

  list

---

4. 编写 Storm 程序

- ADModel 数据模型

```
public class ADModel {
    private String adid;
    private String uid;
    private String action;

    public String getAdid() {
```

```
        return adid;
    }
    public void setAdid(String adid) {
        this.adid = adid;
    }
    public String getUid() {
        return uid;
    }
    public void setUid(String uid) {
        this.uid = uid;
    }
    public String getAction() {
        return action;
    }
    public void setAction(String action) {
        this.action = action;
    }
}
```

- AdTopology

```
import com.utils.ConfigMananger;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.generated.StormTopology;
import org.apache.storm.kafka.*;
import org.apache.storm.spout.SchemeAsMultiScheme;
import org.apache.storm.topology.TopologyBuilder;
import java.util.Properties;

//从 Kafka 里读取 Topic 为 AD 的最新的日志消息并发送个 LogToModelBolt
public class AdTopology {

    public static void main(String[] args) throws Exception {
        String topicName = "topicName";
        String brokerZkStr  = "node1:2181,node2:2181,node3:2181";
        String brokerList = "node1:6667,node2:6667,node3:6667";
        String kafkaSer = "kafka.serializer.StringEncoder";
        String groupId = "AD";
        int parallelism_read_kafka = 3;
        int parallelism_write_redis = 3;
        int numWorkers = 6;

        BrokerHosts brokerHosts = new ZkHosts(brokerZkStr);
        SpoutConfig spoutConfig = new SpoutConfig(brokerHosts, topicName, "/consumers",
"AD");

        Config conf = new Config();
        Properties props = new Properties();
        props.put("metadata.broker.list", brokerList);
        props.put("serializer.class", kafkaSer);
        props.put("group.id", groupId);
        conf.put("kafka.broker.properties", props);
        conf.put("topic", topicName);

        spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
        KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
```

```
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("KafkaSpout", kafkaSpout, parallelism_read_kafka);
        builder.setBolt("LogToModelBolt", new Log2ModelBolt(), 
parallelism_write_redis).shuffleGrouping("KafkaSpout");
        builder.setBolt("ToHbaseBolt",new ToHBaseBolt(), 
parallelism_write_redis).shuffleGrouping("LogToModelBolt");

        StormTopology topology = builder.createTopology();
        Config config = new Config();
        config.setDebug(false);

        if (args != null && args.length > 0) {
            //运行集群模式
            config.setNumWorkers(numWorkers);
            StormSubmitter.submitTopology(args[0], config, builder.createTopology());
        } else {
            LocalCluster localCluster = new LocalCluster();
            localCluster.submitTopology("AdTopology", config, topology);
        }
    }
}
```

- Log2ModelBolt

```
import com.google.gson.Gson;
import org.apache.storm.shade.org.apache.commons.lang.StringUtils;
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;
import java.util.Map;

 // 解析 Log 并转化为 Model，发送给 ToHbaseBolt
public class Log2ModelBolt extends BaseRichBolt {
    private OutputCollector collector;

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector 
collector) {
        this.collector = collector;
    }

    @Override
    public void execute(Tuple input) {
        // 2014-01-13   19:11:55    {"adid":"31789","uid":"9871","action":"view"}
63.237.239.3    北京 北京
        String line = input.getStringByField("value");

        String[] arr = line.split("\t", -1);
        if (arr.length == 6) {
            String date = arr[0].trim().replace("-", "");
            String time = arr[1].trim();
            String json = arr[2].trim();
            String ip = arr[3].trim();
```

```
            String province = arr[4].trim();
            String city = arr[5].trim();

            if (StringUtils.isNotEmpty(json)) {
                ADModel ad = new Gson().fromJson(json, ADModel.class);
                if (null != ad && StringUtils.isNotEmpty(ad.getAdid())) {
                    // 省
                    if (StringUtils.isNotEmpty(province)) {
                        String rowkey = ad.getAdid() + "_" + province + "_" + date;
                        collector.emit(new Values(ad.getAction(), rowkey, 1L));
                    }

                    // 市
                    if (StringUtils.isNotEmpty(city)) {
                        String rowkey = ad.getAdid() + "_" + city + "_" + date;
                        collector.emit(new Values(ad.getAction(), rowkey, 1L));
                    }

                    // 客户端
                    if (StringUtils.isNotEmpty(province)) {
                        String rowkey = ad.getAdid() + "_" + ad.getUid() + "_" + date;
                        collector.emit(new Values(ad.getAction(), rowkey, 1L));
                    }
                }
            }
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("action", "rowkey", "cnt"));
    }
}
```

- ToHBaseBolt

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Tuple;

import java.io.IOException;
import java.util.Map;

 //ToHbaseBolt 将处理后的数据写入到 Hbase 表里
public class ToHBaseBolt extends BaseRichBolt {
    private Table table;

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector
```

```java
collector) {
        try {
            Configuration conf = HBaseConfiguration.create();
            conf.set("hbase.zookeeper.quorum",
"hadoop1:2181,hadoop2:2181,hadoop3:2181");
            Connection conn = ConnectionFactory.createConnection(conf);
            table = conn.getTable(TableName.valueOf("realtime_ad_stat"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void execute(Tuple input) {
        String action = input.getStringByField("action");
        String rowkey = input.getStringByField("rowkey");
        Long pv = input.getLongByField("cnt");

        try {
            if ("view".equals(action)) {
                table.incrementColumnValue(Bytes.toBytes(rowkey),
Bytes.toBytes("stat"), Bytes.toBytes("view_cnt"), pv);
            }
            if ("click".equals(action)) {
                table.incrementColumnValue(Bytes.toBytes(rowkey),
Bytes.toBytes("stat"), Bytes.toBytes("click_cnt"), pv);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {

    }
}
```

5. Kafka 模块操作

(1) 创建名为 AD 的 Topic，命令如下：

/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --create

--zookeeper node1:2181,node2:2181,node3:2181

--topic AD

--partitions 3

--replication-factor 3

(2) 模拟发送消息

```java
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;
import java.util.Random;

public class ProducerClient {
```

```
    private static final String[] PROVINCES_CITIES = new String[]{
            "山东\t 济南", "河北\t 石家庄", "吉林\t 长春", "黑龙江\t 哈尔滨", "辽宁\t 沈阳", "
内蒙古\t 呼和浩特","新疆\t 乌鲁木齐", "甘肃\t 兰州", "宁夏\t 银川", "山西\t 太原", "陕西\t 西安",
"河南\t 郑州", "安徽\t 合肥", "江苏\t 南京", "浙江\t 杭州", "福建\t 福州", "广东\t 广州", "江西
\t 南昌", "海南\t 海口", "广西\t 南宁","贵州\t 贵阳", "湖南\t 长沙", "湖北\t 武汉", "四川\t 成都
", "云南\t 昆明", "西藏\t 拉萨", "青海\t 西宁","天津\t 天津", "上海\t 上海", "重庆\t 重庆", "北
京\t 北京"};

    private static final String[] ACTIONS = new String[]{"view", "click"};
    private static final String[] ADIDS = new String[]{"1", "2", "3", "4", "5"};

    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put("bootstrap.servers", "node1:6667,node2:6667,node3:6667");
        props.put("acks", "-1");
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        KafkaProducer<String, String> kafkaProducer = new KafkaProducer(props);
        boolean flag = true;
        if (flag) {
            for (int i = 0; i < 2000; i++) {
                //3、发送数据
                //2014-01-13  19:11:55   {"adid":"31789","uid":"9871"}   63.237.239.3
北京市  北京市
                StringBuilder sb = new StringBuilder();
                //sb.append(new SimpleDateFormat("yyyy-MM-dd").format(date));
                sb.append("2018-08-10");
                sb.append("\t");
                sb.append("12:00:00");
                sb.append("\t");
                sb.append("{\"adid\":\"" + ADIDS[new Random().nextInt(ADIDS.length)]
+ "\",\"uid\":\"" + new Random().nextInt(200) + "\",\"action\":\"" + ACTIONS[new
Random().nextInt(ACTIONS.length)] + "\"}");
                sb.append("\t");
                sb.append(new Random().nextInt(255) + "." + new Random().nextInt(255)
+ "." + new Random().nextInt(255) + "." + new Random().nextInt(255));
                sb.append("\t");
                sb.append(PROVINCES_CITIES[new
Random().nextInt(PROVINCES_CITIES.length)]);
                kafkaProducer.send(new ProducerRecord("AD", sb.toString()));
            }
            Thread.sleep(1000);
            kafkaProducer.flush();
        }
        kafkaProducer.close();
    }
}
```

6. 提交拓扑

(1) 登录大数据集群内安装 Storm 的任意一节点或安装 Storm Client 的节点。

(2) 将准备依赖的 Jar 包和配置文件上传到任意目录下。

(3) 提交拓扑，命令如下：

storm jar <jar 包路径> <入口类名> <该拓扑名>

(4) 任务提交成功以后，参考 2.4 快速链接章节可查看任务执行情况。

7. Kerberos 场景

可参见 5.2 7. Kerberos 场景章节进行相关操作。

5.4 Storm-JDBC开发样例

1. 实践场景

流计算业务场景中，可能需要读写关系型数据库的表数据以实现流数据补齐或者维度统计等业务。本章节提供的开发样例中，通过模拟简单的 WordCount 场景来展示 Storm 业务开发中对关系型数据库的操作方法，

2. 操作

- 首先需要准备一个数据库，此时可根据具体场景选择适合的数据库。
- 代码样例如下：

```
import com.examples.storm.wordcount.WordCountSpout;
import org.apache.storm.Config;
import org.apache.storm.StormSubmitter;
import org.apache.storm.jdbc.bolt.JdbcInsertBolt;
import org.apache.storm.jdbc.bolt.JdbcLookupBolt;
import org.apache.storm.jdbc.common.Column;
import org.apache.storm.jdbc.common.ConnectionProvider;
import org.apache.storm.jdbc.common.HikariCPConnectionProvider;
import org.apache.storm.jdbc.mapper.JdbcMapper;
import org.apache.storm.jdbc.mapper.SimpleJdbcLookupMapper;
import org.apache.storm.jdbc.mapper.SimpleJdbcMapper;
import org.apache.storm.shade.com.google.common.collect.Lists;
import org.apache.storm.shade.com.google.common.collect.Maps;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.tuple.Fields;

import java.sql.Types;
import java.util.List;
import java.util.Map;

public class JDBCTopology {
    public static void main(String[] args) throws Exception {
        //connectionProvider 配置。PG 连接 URL: jdbc:postgresql://127.0.0.1:543
        Map hikariConfigMap = Maps.newHashMap();
        hikariConfigMap.put("dataSourceClassName", "org.postgresql.Driver");
        hikariConfigMap.put("dataSource.serverName", "127.0.0.1");//请改为实际的 IP
        hikariConfigMap.put("dataSource.portNumber", "5432");//请改为实际的端口

        hikariConfigMap.put("dataSource.databaseName", "example");
        hikariConfigMap.put("dataSource.user", "app");
        hikariConfigMap.put("dataSource.password", "mine");
        hikariConfigMap.put("connectionTestQuery", "select COUNT from  GOAL");
        Config conf = new Config();
```

```
        ConnectionProvider connectionProvider = new
HikariCPConnectionProvider(hikariConfigMap);
        //JdbcLookupBolt 实例化
        Fields outputFields = new Fields("WORD", "COUNT");
        List<Column> queryParamColumns = Lists.newArrayList(new Column("WORD",
Types.VARCHAR));
        SimpleJdbcLookupMapper jdbcLookupMapper = new
SimpleJdbcLookupMapper(outputFields, queryParamColumns);
        String selectSql = "select COUNT from ORIGINAL where WORD = ?";
        JdbcLookupBolt wordLookupBolt = new JdbcLookupBolt(connectionProvider,
selectSql, jdbcLookupMapper);

        //JdbcInsertBolt 实例化
        String tableName = "GOAL";
        JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper(tableName,
connectionProvider);
        JdbcInsertBolt userPersistanceBolt = new JdbcInsertBolt(connectionProvider,
simpleJdbcMapper).withTableName("GOAL").withQueryTimeoutSecs(30);
        WordCountSpout wordSpout = new WordCountSpout();
        TopologyBuilder builder = new TopologyBuilder();

        //构建topology
        builder.setSpout("word_spout", wordSpout);
        builder.setBolt("jdbc_lookup_bolt", wordLookupBolt,
3).fieldsGrouping("word_spout", new Fields("word"));
        builder.setBolt("jdbc_insert_bolt", userPersistanceBolt,
3).fieldsGrouping("jdbc_lookup_bolt", new Fields("word"));
        StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
    }
}
```

3. Kerberos 场景

可参见5.2 7. Kerberos 场景章节进行相关操作。

6 常见问题解答

6.1 调优

1. 调整并行度

- 增加 Worker 并行度

  性能调优最常用的调优方式为：调整运行拓扑的并行度。

  调整并行度包括：调整拓扑使用的 Worker 数量和每个组件（Spout/Bolt）的执行线程数量。

  其中，拓扑程序中配置的Worker数量就是拓扑运行占用的进程数量，当流处理数据量较大时，增加 Worker 数量，可显著提高流处理计算速度。Storm 流计算任务调优时首先考虑根据集群资源剩余情况适量增加该拓扑的 Worker 数量。

  Worker 数量调整方式：应用程序中，在调用 StormSubmitter.submitTopologyWithProgressBar()方法提交拓扑任务之前，调用 conf.setNumWorkers(numWorkers)来设置拓扑运行时的 Worker 数量。

- 设置 Spout task 处于 pending 状态的最大 Tuple 数量

  当集群中的剩余资源有限无法显著增加 Worker 数量，或者增加 Worker 数量已无法提高流计算速度时，可考虑增加组件的线程数量（并行度）。

  组件并行度调整方式：应用程序中，在调用 builder.setSpout(“id_name”, new Spout(), parallelism_hint)/build.setBolt(“id_name”, new Bolt(), parallelism_hint)时，适量增加并行度参数，即根据集群资源情况适量增加 parallelism_hint 的数量（注意：并行度最大只能调整到设置的 task 个数）。

2. 调整 Storm 参数

- 调整 worker.childopts

  Worker 为实际拓扑运行所在的进程，当拓扑任务计算的数据量较大时，可能出现 GC 时间过长等性能问题。目前大数据平台中 worker.childopts 默认为 768MB，调优时可考虑适当增加 Worker 内存大小。

- 调整 supervisor.childopts

  Supervisor 负责当前节点中拓扑任务的监控、作业加载等任务，如果 Worker 数量较大或者拓扑任务较复杂，可考虑适当增大 Supervisor 内存大小。由于 Supervisor 进程不参与实际的计算任务，所以该内存值不需要设置过大。

- 调整 supervisor.worker.timeout.secs

  该参数用于设置 Supervisor 中 Worker 心跳超时时间，如果 Worker 中计算任务较重或者 Worker 数量较多，可能出现网络通信超时的异常，此时可考虑适当增加 Worker 心跳超时时间。目前大数据平台中 supervisor.worker.timeout.secs 默认为 30s。

- 调整 topology.message.timeout.secs

  该参数用于设置拓扑中 Spout 发送消息的最大响应时间，如果数据量较大导致 Bolt 处理过慢以致超时，可增加该参数。目前大数据平台中 topology.message.timeout.secs 默认为 30s。

3. 调整 ZooKeeper 参数

Storm 集群的元数据管理、作业分配、状态管理等都依赖 ZooKeeper，Storm 集群中各组件及 Worker 都需要与 ZooKeeper 保持通信，因此当集群中设置的 Worker 个数较多时，可能出现 ZooKeeper 客户端连接数占满的情况。目前大数据平台中默认的 Zookeeper 最大连接数为 500，一般不需要调整。

6.2 运维类问题

1. 发布 Topology 到远程集群中，出现 AlreadyAliveException（msg:XXX is already active）异常

原因分析：

可能是提供的 Topology 名称与已经在运行的 Topology 名称重名。

解决方法：

发布提交时更换 Topology 名称，要求与已运行的 Topology 名称不同。

2. Bolt 在处理较大数据量时，Worker 的日志中出现 Failing message

原因分析：

可能是 Topology 的消息处理超时。

解决方法：

提交 Topology 时设置适当的消息超时时间，比默认消息超时时间（30s）更长。比如设置超时时间为 60s，即 conf.setMessageTimeoutSecs(60)。

3. 发布 Topology 至远程集群后，在 Storm UI 中查看 Topopoly 的 Num Workers、Num Executors、Num Tasks 均为 0，如何解决？

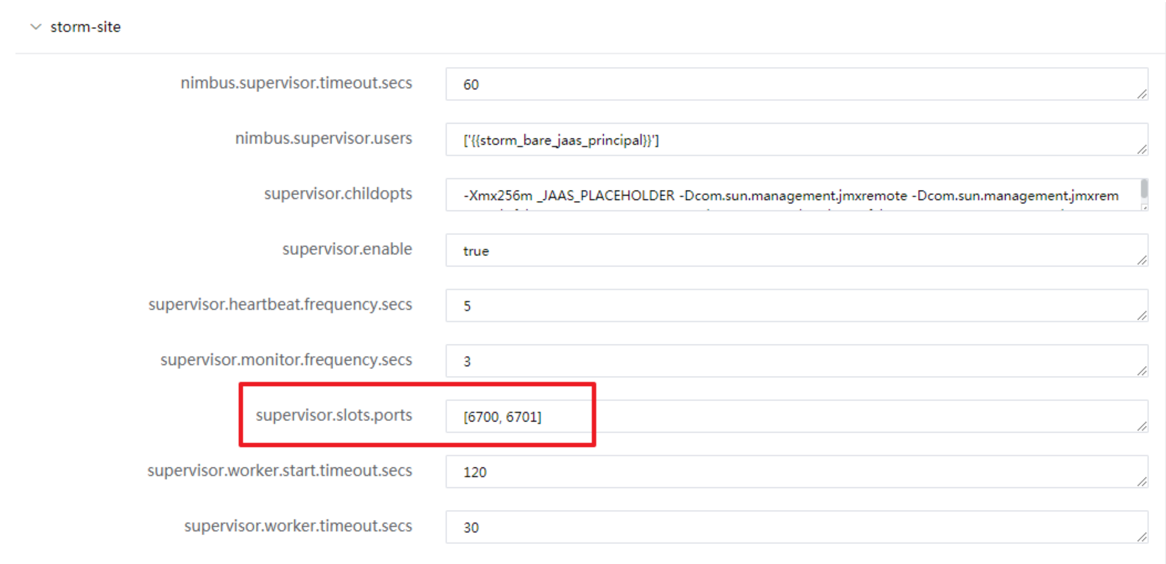
原因分析：

可能是因为 Supervisor 节点可用的 Slots 数不足

解决方法：

可以对 Storm 集群的 Supervisor 进行扩容，或者在大数据平台的 Storm 的 Supervisor 配置 supervisor.slots.ports，如图 6-1 所示，增加可用的 Slot 端口数，然后重启 Supervisor 节点。

图6-1 增加可用的 Slot 端口数



4. Storm 消费 Kafka 的业务场景中，无法管理和查看对应消费者的 group.id 以及消费情况，如何解决？

原因分析：

大数据平台中 Kafka 默认的 consumer 保存和管理路径为 ZooKeeper 的/consumers。如果编写 Storm 程序时设置的 group.id 所在路径不是/consumers，则 Kafka 将无法管理到该 group.id

解决方法：

Storm 消费 Kafka 的业务代码中，SpoutConfig()的构造方法中将第三个参数设置为"/consumer"。

5. Topology 中设置 worker 数量大时，多次提交 Storm 任务，在 Storm UI 界面 kill 所有任务后出现 Supervisor 挂掉的情况，如何解决？

原因分析：

Storm 社区已知的稳定性问题，暂无法解决，给出规避措施。

解决方法：

当 Storm 集群中 worker 较多且拓扑任务较多时，如需要强制停止任务，需等待上一个任务完全关闭后，再去手动停止后面的任务。

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.2 组件架构 1-1
1.3 应用场景 1-2
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 查看组件的日志信息 2-1
2.2 运行状态监控 2-1
2.2.1 查看组件详情 2-1
2.2.2 组件检查 2-2
2.3 快速使用指导 2-3
2.3.1 非 Kerberos 环境 2-4
2.3.2 Kerberos 环境 2-5
2.4 快速链接 2-8
2.4.1 配置组件快速链接 2-8
2.4.2 访问 Hive 快速链接 2-8
3 使用指南 3-1
3.1 Client 下载/安装/使用/卸载 3-1
3.1.1 下载 Client 安装包 3-1
3.1.2 安装 Client 客户端 3-2
3.1.3 访问组件 3-3
3.1.4 非 Kerberos 环境下使用 Hive Client 3-3
3.1.5 Kerberos 环境下使用 Hive Client 3-5
3.1.6 卸载 Client 客户端 3-7
3.2 添加/删除进程 3-7
3.2.1 添加进程 3-7
3.2.2 删除进程 3-8
3.3 权限访问控制 3-9
3.3.1 权限说明 3-10
3.3.2 权限使用操作示例 3-11
3.4 租户管理 3-32
3.4.1 租户介绍 3-33

3.4.2 新增租户 3-33
3.4.3 租户使用操作示例 3-35
3.5 备份恢复 3-36
3.5.1 新建 Hive 同步任务 3-37
3.5.2 源集群和目的集群配置互信 3-38
3.5.3 Hive 同步任务相关配置修改 3-40
3.5.4 HDFS 加密区数据同步 3-41
3.5.5 Hive 备份恢复示例 3-41
3.6 LDAP 认证 3-44
3.7 事务性操作 3-45
3.7.1 ACID 的含义和使用 3-45
3.7.2 开启事务 3-46
3.7.3 事务表操作 3-46
3.8 HiveServer2 高可用功能 3-46
3.9 Hive 分区表生命周期管理 3-48
3.9.1 总体要求 3-48
3.9.2 使用限制 3-49
3.9.3 使用说明 3-49
3.10 HQL 操作 3-55
3.10.1 创建表 3-55
3.10.2 数据加载 3-58
3.10.3 数据查询 3-59
3.10.4 视图操作 3-60
3.10.5 函数介绍 3-60
3.11 JDBC 方式连接 HiveServer2 3-69
4 最佳实践 4-1
4.1 Hive 操作 HBase 表 4-1
4.1.1 概述 4-1
4.1.2 操作示例 4-1
4.2 Hive on Spark 4-2
4.2.1 概述 4-2
4.2.2 操作示例 4-3
5 常见问题解答 5-1
5.1 调优类 5-1
5.2 运维类问题 5-5

1 组件简介

1.1 组件概述

Hive 是建立在 Hadoop 上的数据仓库框架，提供一种类 SQL 的语言 HQL（Hive Query Language），对结构化和半结构化数据进行批量分析，完成数据计算。

HQL 具有对海量数据处理的能力，将执行的 HQL 语句转换为分布式计算任务，从而完成海量数据的查询和分析工作。同时，为了满足不同场景的需求，HQL 能通过实现用户自定义函数（UDF）、用户自定义聚合函数（UDAF）以及用户自定义表函数（UDTF）对其进行扩展。

Hive 具有以下特点：

- 易于进行数据的抽取、转换和加载（ETL）
- 支持多种数据存储格式
- 能直接访问存储在 HDFS 或其他数据存储系统上的文件
- 支持多种使用方式（如 JDBC、WebUI 等）

1.2 组件架构

图1-1 Hive 整体架构

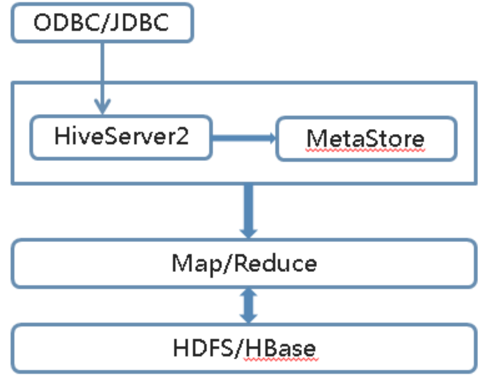


表1-1 模块说明

模块名称	说明
HiveServer2	• 一个集群内可部署多个 HiveServer2 • 对外提供 Hive 数据库服务，将用户提交的 HQL 语句进行编译，解析成对应的 Map/Reduce 任务或者 HDFS 操作，从而完成数据的提取、转换和分析 • 提供 JDBC/ODBC 接口

模块名称	说明
MetaStore	• 一个集群内可部署多个 MetaStore • 提供 Hive 的元数据服务，负责 Hive 表结构和属性信息的读、写、维护和修改 • 提供 Thrift 接口，供 HiveServer2、Spark 和 WebHCat 等 MetaStore 客户端来访问，操作元数据
Map/Reduce	• Hive 将 SQL 语句拆分成 Map 和 Reduce 任务，然后通过计算引擎 MapReduce（或 TEZ、Spark）执行 • Hive（version: 2.1.1-cdh6.2.0）使用 MapReduce、Tez 和 Spark 计算引擎

1.3 应用场景

Hive 常用于以下场景：

- 数据离线分析：可以统计一个网址在某段时间内的页面访问量等。
- 数据挖掘：Hive 与 Spark 结合可以完成数据挖掘工作，如用户行为分析、热点推荐等。
- 低成本数据分析：不需要编写 MapReduce 程序，可以使用 SQL 语句替代。

2 快速入门

2.1 组件安装

说明

在 Hadoop 集群中，安装 Hive 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。

2.1.1 查看组件的日志信息

表2-1 组件日志路径说明

组件	日志路径
Hive	/var/de_log/hive

2.2 运行状态监控

2.2.1 查看组件详情

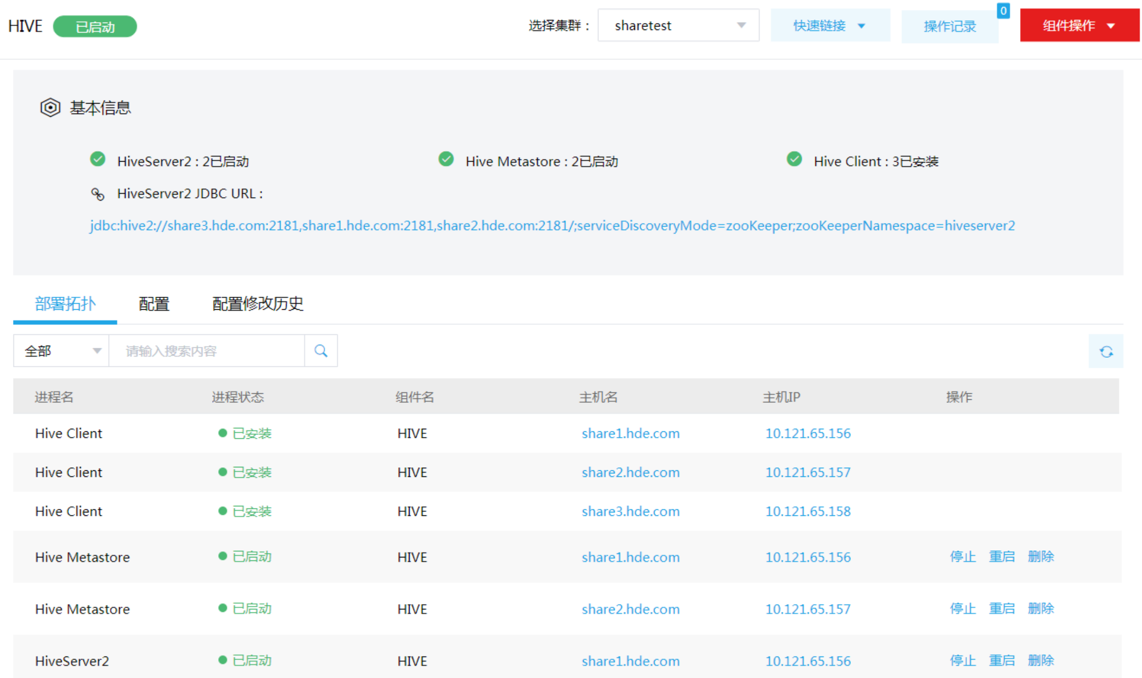
进入 Hive 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、部署拓扑、配置详情和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 组件详情



2.2.2 组件检查

执行 Hive 组件检查时，会检查 HiveServer 是否可以正常连接。

集群在使用过程中，根据实际需要，可对 Hive 组件执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：
- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Hive 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中Hive组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

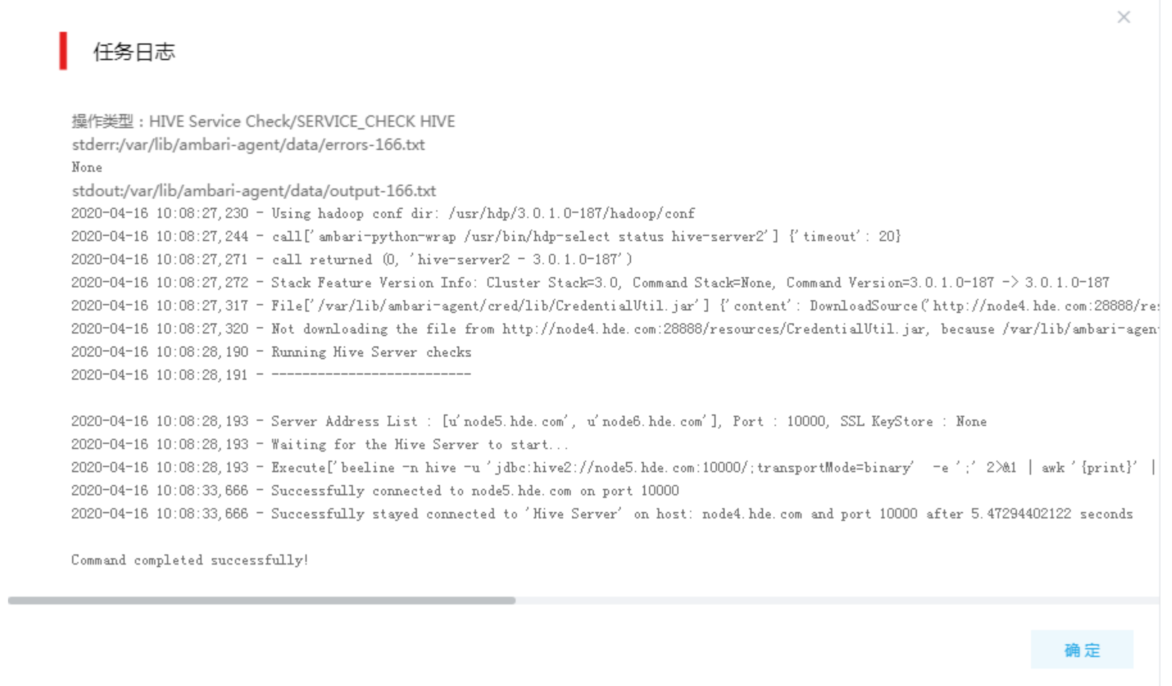
(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗中会显示组件检查成功或失败的状态，如图 2-2 所示，表示该组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“Hive Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情



2.3 快速使用指导

注意

根据大数据集群是否开启 Kerberos 认证，用户访问 Hive 时的认证方式不同，详情请参见本章节内容。

Hive 既可以通过集群用户访问，又可以通过组件超级用户访问。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
    - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。
    - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色即可拥有该角色所具有的权限；不开启权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 Hive 组件的 hive 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 Kerberos 环境

说明

- 非 Kerberos 环境下，不需要用户做身份认证即可直接对 Hive 执行操作。
- 本章节操作需要切换至具有操作 Hive 权限的用户。

Hive 安装完成后，连接集群内的节点（该节点要求安装了 Hive Client）即可操作 Hive。在非 Kerberos 环境下通过控制台执行创建表、查询表等相关操作。操作步骤如下：

(1) 连接 HiveServer2

在集群中包含 Hive Client 的某台机器中，使用以下任意方式连接：

- 方式一：默认参数连接

  直接输入 beeline，将会使用当前会话框用户自动连接集群的 HiveServer。

- 方式二：指定参数连

```
beeline -u "jdbc:hive2://<HiveServer2 地址>:10000/" -n hive -p ""
```

  其中：<HiveServer2 地址>为 HiveServer2 所在的机器 IP 地址或主机名（在 Client 节点上通过主机名连接 HiveServer2 时，该节点的/etc/hosts 文件中必须包含对应集群的/etc/hosts 文件的节点信息）；10000 为 HiveServer2 的默认端口号。

说明

指定参数连接时，在 beeline 命令中：

- -u：指定连接 HiveServer2 的地址和端口号。
- -n：指定连接所需用户名。
- -p：指定连接所需密码（使用 LDAP 认证时，需正确填写用户对应的密码。其他情况可为空值，开启 LDAP 认证的具体操作请参见 3.6 LDAP 认证章节）。

(2) 创建表

```
create table a(i int,s string);
```

(3) 插入数据

```
insert into table a values(1, 'a');
```

(4) 查看数据

```
select * from a;
```

结果：

```
+------+------+--+
| a.i  | a.s  |
+------+------+--+
| 1    | a    |
+------+------+--+
```

2.3.2 Kerberos 环境

说明

- Kerberos 环境下，用户需要做身份认证才可直接对 Hive 执行操作，认证方式请参见 2.3.2 1. Kerberos 环境下的用户身份认证。
- 本章节操作需要切换至具有操作 Hive 文件系统权限的用户。

1. Kerberos 环境下的用户身份认证

Kerberos 环境下进行用户身份认证的方式（本章节示例用户为 user1），认证有以下两种方式（根据实际情况任选其一即可）：

- 集群用户身份认证
- 组件超级用户身份认证

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

Hive 还可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**
  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的 /etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：
  ```
  chown user1 /etc/security/keytabs/user1.keytab
  ```
  b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：
  ```
  klist -k user1.keytab
  ```

【说明】如图 2-4 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-4 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

su user1

kinit -kt user1.keytab user1@TENANTC.COM

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-5 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

（二）组件超级用户身份认证

Hive 可以通过组件超级用户访问，比如 hive 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 hive 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 hive 的认证文件“hive.service.keytab”。

【说明】在 Hive Client 节点上，需要将 hive 的认证文件“hive.service.keytab”上传节点的任意目录下，将 keytab 文件的所有者修改为 hive，然后切换至 hive 用户下进行认证，其中修改所有者命令为：

chown hive hive.service.keytab

(2) 使用 ***klist*** 命令查看 hive.service.keytab 的 principal 名称，命令如下：

klist -k hive.service.keytab

【说明】如图 2-6 所示，红框内容即为 hive.service.keytab 的 principal 名称。

图2-6 认证文件的 principal 名称

```
[root@node1 keytabs]# klist -k hive.service.keytab
Keytab name: FILE:hive.service.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
```

(3) 切换至用户 hive，并执行身份验证的命令如下：

su hive

kinit -kt hive.service.keytab hive/node1.hde.com@TESTSHARE.COM

【说明】其中：hive.service.keytab 为 hive 的认证文件，hive/node1.hde.com@TESTSHARE.COM 为 hive.service.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

2. Hive 操作说明

Hive 安装完成后，连接集群内的节点（该节点要求安装了 Hive Client）即可操作 Hive。当用户身份认证成功后，在 Kerberos 环境下通过控制台执行创建表、查询表等相关操作。操作步骤如下：

(1) Hive 的 Principal 配置值获取

登录集群 HiveServer2 所在节点（示例：node2.hde.com），运行以下命令获取 principal：

klist -kt /etc/security/keytabs/hive.service.keytab

结果：

Keytab name: FILE:/etc/security/keytabs/hive.service.keytab

KVNO Timestamp Principal

---- ------------------- --

1 04/15/2020 21:35:56 hive/node2.hde.com@TENANTC.COM

1 04/15/2020 21:35:56 hive/node2.hde.com@TENANTC.COM

1 04/15/2020 21:35:56 hive/node2.hde.com@TENANTC.COM

1 04/15/2020 21:35:56 hive/node2.hde.com@TENANTC.COM

(2) 连接 HiveServer2

在集群中包含 Hive Client 的某台机器（示例：node2.hde.com）中，使用以下任意方式连接：

- 方式一：默认参数连接

  直接输入 beeline，将会使用当前会话框用户自动连接集群的 HiveServer。

- 方式二：指定参数连接

  beeline -u " jdbc:hive2://node2:10000/;principal=hive/node2.hde.com@TENANTC.COM " -n hive -p ""

  其中：

  – -u 用于指定连接 HiveServer2 的地址和端口号。HiveServer2 的端口号默认为 10000，principal 名称为 hive/node2.hde.com@TENANTC.COM。

  – -n 用于指定连接所需的用户名（可为空值）。

– -p 用于指定连接所需的密码（使用 LDAP 认证时，需正确填写用户对应的密码；其他情况可为空值）。

(3) 创建表

create table a(i int,s string);

(4) 插入数据

insert into table a values(1, 'a');

(5) 查看数据

select * from a;

结果：

```
+------+------+--+
| a.i  | a.s  |
+------+------+--+
| 1    | a    |
+------+------+--+
```

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

2.4.2 访问 Hive 快速链接

Hive 提供了 Hive 监控页面（即 HiveServer UI），可以直观的看到当前链接的会话、历史日志、配置参数以及度量信息。

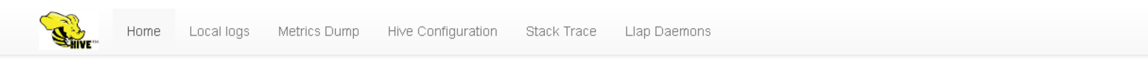
(1) 如图 2-7所示，在 Hive 组件详情页面的右上角[快速链接]的下拉框中，可以获取访问入口信息。

【说明】当集群开启高可用时，Hive 同步开启 HA，此时有两个访问入口，任选其一即可。

图2-7 Hive 快速链接

(2) HiveServer 页面默认主页显示的是当前链接的会话，包括 IP、用户名、当前执行的查询数量、链接总时长、空闲时长；如果有会话执行查询，页面中的 Queries 模块会显示查询的语句、执行耗时等。

图2-8 HiveServer UI 页面



HiveServer2

Active Sessions

User Name	IP Address	Operation Count	Active Time (s)	Idle Time (s)
hive	10.121.65.200	0	966	799
hive	10.121.65.200	0	15014	15014
anonymous	10.121.65.198	0	569	559
Total number of sessions: 3				

Open Queries

User Name	Query	Execution Engine	State	Opened Timestamp	Opened (s)	Latency (s)	Drilldown Link
Total number of queries: 0							

3 使用指南

3.1 Client下载/安装/使用/卸载

用户可下载 Hive Client，在客户端节点上安装 Hive 的 Client 后，即可直接连接集群中的 Hive，进行组件维护、任务管理等操作。

3.1.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。
- 租户中的 Hive 组件也支持下载 Client，下载 Client 按钮在租户详情页面。本文以下载集群中组件 Client 为例进行操作，租户中的 Client 下载操作类似。

下载 Client 安装包的步骤如下：

(1) 在集群管理页面的集群列表中，单击集群名称进入集群详情页面，在集群已安装的组件列表中单击 Hive 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-1所示。

图3-1 下载客户端

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：
    - 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
    - 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中的组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。
    - 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
    - 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.1.2 安装 Client 客户端

说明

- 安装 Client 的节点必须能与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点上或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置网络连接，仅非 root 用户需要执行此操作，root 用户可跳过此步骤。

    在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

    在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

    ./install.sh <安装路径> -o

说明

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为 /usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

```
./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>
```

3.1.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

```
source bigdata_env
```

- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同：
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行身份认证即可直接访问组件并执行相关操作。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行身份认证之后，才可访问组件并执行相关操作。关于 Kerberos 环境下，进行用户认证的操作方式请参见 2.3.2 1. Kerberos 环境下的用户身份认证。

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

3.1.4 非 Kerberos 环境下使用 Hive Client

1. 使用 beeline 命令连接 HiveServer2

使用 beeline 命令连接 HiveServer2，操作步骤如下：

(1) Hive 启动 beeline
  - 方式一：以当前用户 beeline 连接 HiveServer2

```
beeline
```

- 方式二：自定义用户 beeline 连接 HiveServer2

  beeline -u url -n username -p password

  在 beeline 中连接 Hiveserver2，分为两种模式：

  - 非 HA 模式下连接，执行如下命令：

    url: jdbc:hive2://<HiveServer2 地址>:10000

    其中：<HiveServer2 地址>为 HiveServer2 所在的机器 IP 地址或主机名（在集群外通过主机名连接 HiveServer2 时必须配置本地 hosts 文件）；10000 为 HiveServer2 的默认端口号。

    需要根据实际环境输入用户名（示例用户为 hive）和密码，密码为空，连接示例命令如下：

    beeline -u “jdbc:hive2://node2:10000/” -n hive -p ""

  - HA 模式下连接，执行如下命令：

    url: jdbc:hive2:// <zk_url>/:serviceDiscoveryMode=zookeeper;
    zooKeeperNamespace=hiveserver2;

    其中，<zk_url>是 ZooKeeper 集群地址，例如“node1:2181,node2:2181,node3:2181”。zooKeeperNamespace 默认 HiveServer2 对应配置文件中 hive.server2.zookeeper.namespace 的值

    需要根据实际环境输入用户名（示例用户为 hive）和密码，密码为空，连接示例命令如下：

    beeline -u
    "jdbc:hive2://node1:2181,node2:2181,nod3e:2181/;serviceDiscoveryMode=zookeeper;zookeeperNamespace=hiveserver2" -n hive -p ""

2. HQL 示例

通过 beeline 连接 Hive 进行建表，操作步骤如下：

(1) 在本地目录下创建 teacher_li.txt 文件，内容如下：

```
John     19
Marry    18
Poli     19
```

(2) 将文件上传到 HDFS 的 tmp 目录下，执行如下命令：

hdfs dfs -put teacher_li.txt /tmp

说明

由于 Hive 权限的限制，用户向 HDFS 上传文件时，要求该用户和启动 beeline 时的登录用户相同。

(3) 使用 beeline 连接 Hive 组件：

beeline –u “!connect jdbc:hive2://node2:10000” –n hdfs –p “”

说明：此处输入示例用户名 hdfs，输入密码为空

(4) 创建表，执行如下语句：

```
CREATE TABLE student1_info( name STRING, age INT)
PARTITIONED BY (teacher STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY  '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

(5) 向表中加载数据，执行如下语句：

```
load data inpath '/tmp/teacher_li.txt' into table student_info partition( teacher='Mr.li' );
```

(6) 查询数据结果，执行如下语句：

```
select * from student_info;
```

图3-2 查询数据结果

```
+---------------------+--------------------+-------------------------+--+
| student_info.name   | student_info.age   | student_info.teacher    |
+---------------------+--------------------+-------------------------+--+
| hn                  | 19                 | Mr.li                   |
| Marry               | 18                 | Mr.li                   |
| Poli                | 19                 | Mr.li                   |
+---------------------+--------------------+-------------------------+--+
```

3.1.5 Kerberos 环境下使用 Hive Client

1. Kerberos 认证

Kerberos 环境下使用 Hive Client 需要先进行用户身份认证，用户身份认证详情请参见 2.3.2 1. Kerberos 环境下的用户身份认证。

2. Hive Client 使用 beeline

Hive Client 使用 beeline 操作步骤如下：

(1) Hive 启动 beeline

- 方式一：以当前用户 beeline 连接 HiveServer2

  ```
  beeline
  ```

- 方式二：自定义用户 beeline 连接 HiveServer2

  在 beeline 中连接 Hiveserver2，分为两种模式：

  – 非 HA 模式下连接，执行如下命令：

  ```
  beeline  !connect
  jdbc:hive2://node:10000/;principal=hive/node.hde.share1.com@SHARETEST.COM
  ```

  其中：node 为 HiveServer2 服务所在机器 IP 地址或主机名（在集群外通过主机名连接 HiveServer2 时必须配置本地 hosts 文件）；hive/node.hde.share1.com@SHARETEST.COM 为 HiveServer2 的 principal 的名称。

  – HA 模式下连接，执行如下命令：

```
beeline -u
"jdbc:hive2://node1:2181,node2:2181,nod3e:2181/;serviceDiscoveryMode=zookeeper;zook
eeperNamespace=hiveserver2" -n hive -p ""
```

其中：node1:2181,node2:2181,node3:2181 是 ZooKeeper 集群地址；zooKeeperNamespace 默认 HiveServer2 对应配置文件中 hive.server2.zookeeper.namespace 的值。

(2) 需要根据实际环境选择输入用户名和密码，密码为空。

3. HQL 示例

通过 beeline 连接 Hive 组件进行建表，操作步骤如下：

(1) 在本地目录下创建 teacher_li.txt 文件，内容格式如下：

```
John	19
Marry	18
Poli	19
```

(2) 将文件上传到 HDFS 的 tmp 目录下，步骤如下：

```
hdfs dfs -put teacher_li.txt /tmp
```

说明

由于 Hive 权限的限制，用户向 HDFS 上传文件时，要求该用户和启动 beeline 时的登录用户相同。

---

a. 使用 beeline 连接 Hive 组件，执行如下语句：

```
!connect jdbc:hive2://node2:10000/;principal=hive/share1.hde.com@SHARETEST.COM
```

说明：此处输入示例用户名 hive，输入密码为空。

b. 创建表，执行如下语句：

```
CREATE TABLE student_info( name STRING, age INT)
PARTITIONED BY (teacher STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY  '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

c. 向表中加载数据，执行如下语句：

```
load data inpath '/tmp/teacher_li.txt' into table student_info partition( teacher='Mr.li' );
```

d. 查询数据结果，执行如下语句：

```
select * from student_info;
```

图3-3 查询数据结果

```
+--------------------+------------------+-----------------------+--+
| student_info.name  | student_info.age | student_info.teacher  |
+--------------------+------------------+-----------------------+--+
| hn                 | 19               | Mr.li                 |
| Marry              | 18               | Mr.li                 |
| Poli               | 19               | Mr.li                 |
+--------------------+------------------+-----------------------+--+
```

3.1.6 卸载 Client 客户端

集群重装之后，之前安装的 Client 客户端将不可用。此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 安装目录下执行以下命令：

./uninstall.sh

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

3.2 添加/删除进程

3.2.1 添加进程

Hive 支持添加 HiveServer2、Hive Metastore、Hive Client 进程。

- 在并发查询任务较多的情况下，可以考虑增加 HiveServer2 和 Hive Metastore 进程，分摊查询负载，提高并发数。
- 若主机新扩容机器时未勾选 client，后期可以通过添加 Hive Client 进程在新机器上增加。

1. 操作示例

说明

- 本章节仅以添加 HiveServer2 进程为例进行说明，其它进程操作类似不再进行说明。
- 若集群中所有节点均已安装 Hive Server2，添加 Hive Server2 进程前则需要先在集群中添加主机，然后再执行添加 Hive Server2 进程的操作。如果集群中有添加进程所需要的主机，则可直接执行添加 Hive Server2 进程的操作。

添加 HiveServer2 进程的操作步骤如下：

(1) 在 Hive 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-4 所示。

a. 选择进程及主机

在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

b. 部署进程

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-4 添加进程

(3) 查看进程变化

HiveServer2 扩容完成后，在组件详情页面[部署拓扑]页签中可以查看 HiveServer2 安装数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.2.2 删除进程

Hive 支持删除 HiveServer2 或 Hive Metastore 进程。

- 在并发任务较少的情况下，可以考虑减少 Hive Metastore 和 HiveServer2 进程，节省集群资源。
- 删除进程后，Hive Metastore、HiveServer2 分别对应进程的个数均不能少于 1 个。

1. 操作示例

说明

- 删除进程操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行删除 HiveServer2 进程操作”为例进行说明，其它进程操作类似不再进行说明。
- 执行删除 HiveServer2 进程操作前，请确认 HiveServer2 是否有运行的任务，如果有运行的任务时，删除进程会影响任务执行。

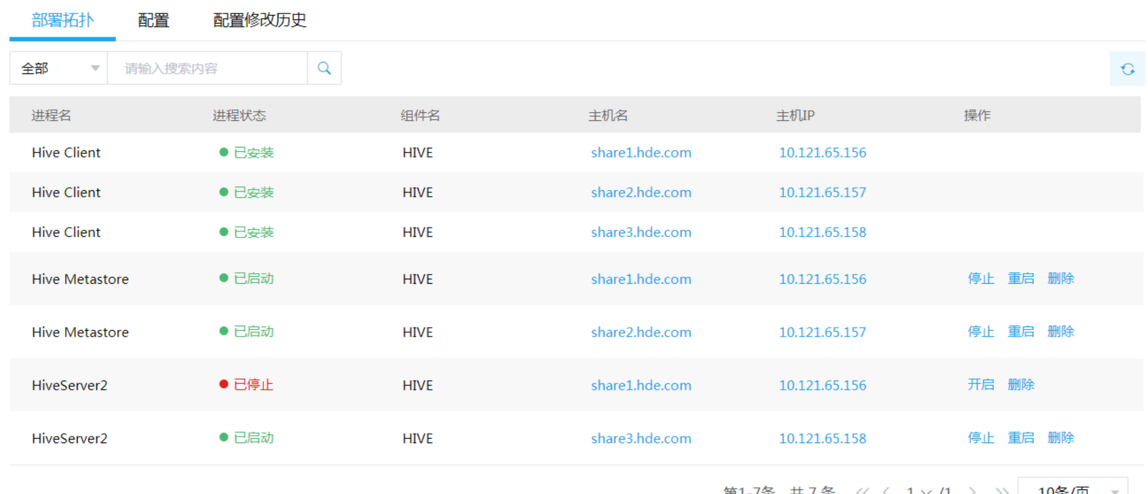
删除 HiveServer2 进程的操作步骤如下：

(1) 在 Hive 组件详情页面[部署拓扑]页签下，选择需要删除 HiveServer2 进程的主机，然后单击该进程右侧操作中的<停止>按钮，停止 HiveServer2。

(2) 删除 HiveServer2

待 HiveServer2 停止成功后，如所示，单击操作列的<删除>按钮，即可完成删除 HiveServer2。

图3-5 删除进程



(3) 查看进程变化

HiveServer2 删除完成之后，在组件详情页面[部署拓扑]页签中可以查看 HiveServer2 进程的数量变化情况以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.3 权限访问控制

注意

集群新建用户的组件权限会因为集群是否开启权限管理功能而有所不同：

- 未开启权限管理时，用户可进行库表的创建、修改、插入、删除等操作。
- 开启权限管理后，组件权限需通过[集群权限/角色管理]中的角色分配给用户，用户通过绑定角色进行赋权后，才能对组件执行操作。

权限管理是安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

3.3.1 权限说明

Hive 支持对数据库表和数据库 UDF 配置权限，并且提供数据脱敏和行级过滤功能。

- 数据库的配置权限：数据库表可对数据库、数据表、列配置权限，权限包括：select、update、create、drop、alter、index、use，其中 all 表示配置所有权限。注意：Hive 的权限同样也适用于 Impala（Impala 只支持查询操作的列权限）、Spark 组件，即 Impala、Spark SQL 也支持对库表的权限访问控制。
- 数据库 UDF 的配置权限：数据库 UDF 可对数据库、UDF 配置权限，权限包括 create、drop。
- 数据脱敏：是指按照某种规则对指定列进行脱敏，保证数据安全性，Hive 数据脱敏策略如表 3-2 所示。
- 行级过滤：是指对表配置行级过滤策略，仅展示满足过滤策略的数据（若过滤策略配置为空则不过滤）。行级过滤类似 where 子句，示例：配置策略 id>1 时，查询表时会自动获取 id 大于 1 的数据。

Hive 操作所需权限对应关系如表 3-1 所示，Hive 数据脱敏策略如表 3-2 所示。

表3-1 Hive 权限说明

权限类型	对应的组件常用操作
select	库表查询等相关操作，如：select、export、show、describe等
update	更新库表等相关操作，如：insert、insert overwrite、delete、update、load等
create	创建库表等相关操作，如create table、create database等
drop	删除库表等相关操作，如：drop table等
alter	修改库表等相关操作，如：alter table等
index	索引等相关操作，如：create index等
use	执行show database、use database等操作
all	支持以上所有操作

表3-2 Hive 数据脱敏策略说明

数据脱敏策略	策略规则	详细说明
MASK	对字母和数字脱敏	• string：x 替代小写字母，X 替代大写字母，n 替代数字 • int：1 替代数字 • date："1900-01-01"替代原值 • decimal：NULL 值替代原值
MASK_SHOW_LAST_4	后四位不脱敏	• string：仅显示后四位，其他用 x 代替 • int：仅显示后四位，其他用 1 代替; • date："1900-01-01"替代原值 • decimal：NULL 值替代原值
MASK_SHOW_FIRST_4	前四位不脱敏	• string：仅显示前四位，其他用 x 代替

数据脱敏策略	策略规则	详细说明
		• int：仅显示前四位，其他用 1 代替 • date："1900-01-01"替代原值 • decimal：NULL 值替代原值
MASK_HASH	显示HASH值	• string：Hash 值替代原值 • int/date/decimal：NULL 值替代原值
MASK_NULL	显示NULL值	NULL值替代原值
MASK_NONE	显示原值	显示原值
MASK_DATE_SHOW_YEAR	脱敏时间信息	• string：x 替换数字 • int：1 替代数字 • date：显示日期年份，月份和日期均用 01 代替 • decimal：NULL 值替代原值

说明

新建拥有 Hive 权限的角色时，若要使用数据脱敏和行级过滤则必须首先配置数据库表的 select 权限。

3.3.2 权限使用操作示例

1. 数据库权限控制

表3-3 授权配置

操作	权限要求（数据库示例为 hivetest）
use database	数据库：*或hivetest，数据表：*，列：*，权限：use
create database	数据库：*或hivetest，数据表：*，列：*，权限：create
drop database	数据库：*或hivetest，数据表：*，列：*，权限：drop

以授予 create 权限为例，对数据库 hivetest 授予 create 权限（其它权限的授予方式与 create 方式操作类似），操作步骤如下：

(1) 新建用户 hiveuser01，授权前执行创建库操作。如图 3-6 所示，表示 hiveuser01 用户没有 hivetest 数据库的 create 权限。

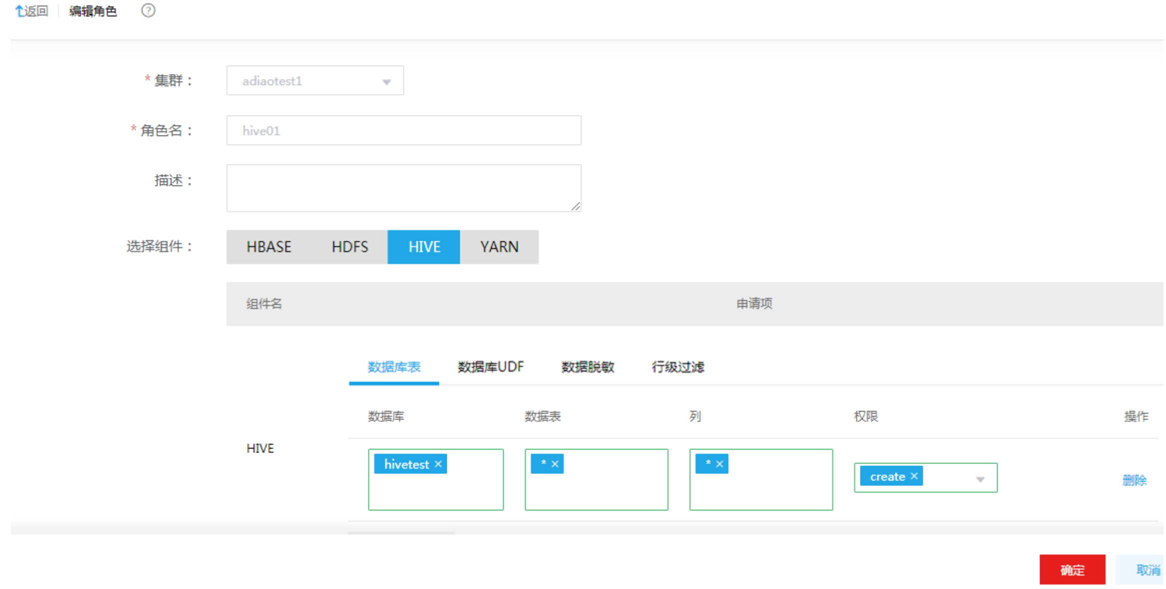
图3-6 执行结果

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> create database hivetest;
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [hiveuser01]
 does not have [CREATE] privilege on [hivetest] (state=42000,code=40000)
```

(2) 角色授权

在[集群权限/角色管理]页面，新建角色 hive01，本示例授予 hive01 角色 hivetest 数据库的 create 权限。

图3-7 为 hive01 角色授予权限



(3) 用户绑定角色

在[集群权限/用户管理]页面，在 hiveuser01 用户的操作列，单击<修改用户授权>按钮，选择角色 hive01。

图3-8 用户绑定角色

(4) 重新执行创建 hivetest 数据库操作。如图 3-9 所示，表示数据库 hivetest 成功创建。

图3-9 执行结果

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> create database hivetest;
No rows affected (0.143 seconds)
INFO  : Compiling command(queryId=hive_20200403095303_b0a2f17d-c47d-4adb-b54a-c64b648acc0b): create database hi
vetest
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20200403095303_b0a2f17d-c47d-4adb-b54a-c64b648acc0b); Time tak
en: 0.029 seconds
INFO  : Executing command(queryId=hive_20200403095303_b0a2f17d-c47d-4adb-b54a-c64b648acc0b): create database hi
vetest
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20200403095303_b0a2f17d-c47d-4adb-b54a-c64b648acc0b); Time tak
en: 0.096 seconds
INFO  : OK
```

2. 表权限控制

表3-4 授权配置

操作	权限要求（数据库示例为 hivetest，表示例为 hivetab1）
create table	数据库：*或hivetest，数据表：*或hivetab1，列：*，权限：create
alter table	数据库：*或hivetest，数据表：*或hivetab1，列：*，权限：alter
insert into table	• Hive：数据库：*或 hivetest，数据表：*或 hivetab1，列：*，权限：update • YARN：队列：*或 default，权限：submit-app
delete from table（针对ACID表）	• Hive：数据库：*或 hivetest，数据表：*或 hivetab1，列：*，权限：update • YARN：队列：*或 default，权限：submit-app
update table（针对ACID表）	• Hive：数据库：*或 hivetest，数据表：*或 hivetab1，列：*，权限：update • YARN：队列：*或 default，权限：submit-app
select * from table	• Hive：数据库：*或 hivetest，数据表：*或 hivetab1，列：*，权限：select • YARN：队列：*或 default，权限：submit-app
create index	数据库：*或hivetest，数据表：*或hivetab1，列：*，权限：index
drop table	数据库：*或hivetest，数据表：*或hivetab1，列：*，权限：drop

以授予 update 权限为例，对数据库 hivetest 数据表 hivetab1 授予 update 权限，操作步骤如下：

(1) 新建用户 hiveuser21，授权前，执行插入表 hivetest.hivetab1 操作如图 3-10 所示，表示 hiveuser21 用户没有 hivetest.hivetab1 数据表的 update 权限。

图3-10 无权限

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> insert into hivetest.hivetab1 values(1, 'abc');
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [hiveuser21]
 does not have [UPDATE] privilege on [hivetest/hivetab1/*] (state=42000,code=40000)
```

(2) 角色授权

在[集群权限/角色管理]页面，新建角色 hive21，本示例授予 hive21 角色 hivetest.hivetab1 数据表的 update 权限。

图3-11 为 hive21 角色授予权限

(3) 用户绑定角色

在[集群权限/用户管理]页面，在 hiveuser21 用户的操作列，单击<修改用户授权>按钮，选择角色 hive21。

图3-12 用户绑定角色

(4) 重新执行插入表 hivetest.hivetab1 操作

如图 3-13 所示，红框内信息表明 hiveuser21 没有提交到 yarn default 队列的权限。

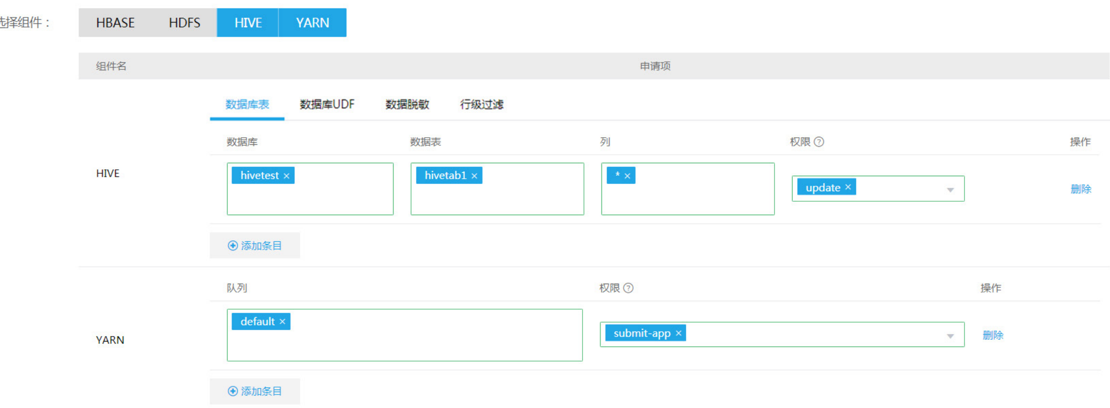
图3-13 无权限

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> insert into hivetest.hivetab1 values(1, 'abc');
INFO  : Compiling command(queryId=hive_20200403101412_9b848c70-9ace-4d8f-ae69-763e170b2ae4): insert into hivete
st.hivetab1 values(1, 'abc')
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:_col0, type:int, comment:null), FieldSchem
a(name:_col1, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200403101412_9b848c70-9ace-4d8f-ae69-763e170b2ae4); Time tak
en: 0.812 seconds
INFO  : Executing command(queryId=hive_20200403101412_9b848c70-9ace-4d8f-ae69-763e170b2ae4): insert into hivete
st.hivetab1 values(1, 'abc')
WARN  :
INFO  : Query ID = hive_20200403101412_9b848c70-9ace-4d8f-ae69-763e170b2ae4
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in serial mode
INFO  : Number of reduce tasks is set to 0 since there's no reduce operator
INFO  : number of splits:1
INFO  : Submitting tokens for job: job_1585828046495_0006
INFO  : Executing with tokens: [Kind: kms-dt, Service: 10.121.47.157:9292, Ident: (kms-dt owner=hiveuser21, ren
ewer=yarn, realUser=hive, issueDate=1585880053424, maxDate=1586484853424, sequenceNumber=29, masterKeyId=2), Ki
nd: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:mycluster, Ident: (token for hiveuser21: HDFS_DELEGATION_TOKEN owne
r=hiveuser21, renewer=yarn, realUser=hive/hppnode1.hde.com@TESTSHARE.COM, issueDate=1585880053360, maxDate=1586
484853360, sequenceNumber=60, masterKeyId=8), Kind: kms-dt, Service: 10.121.47.156:9292, Ident: (kms-dt owner=h
iveuser21, renewer=yarn, realUser=hive, issueDate=1585880053383, maxDate=1586484853383, sequenceNumber=27, mast
erKeyId=2), Kind: HIVE_DELEGATION_TOKEN, Service: HiveServer2ImpersonationToken, Ident: 00 0a 68 69 76 65 75 73
 65 72 32 31 0a 68 69 76 65 75 73 65 72 32 31 23 68 69 76 65 2f 68 70 70 6e 6f 64 65 31 2e 68 64 65 2e 63 6f 6d
 40 54 45 53 54 53 48 41 52 45 2e 43 4f 4d 8a 01 71 3d cc de 80 8a 01 71 61 d9 62 80 8e 09 a8 0b]
INFO  : Cleaning up the staging area /user/hiveuser21/.staging/job_1585828046495_0006
ERROR : Job Submission failed with exception 'java.io.IOException(org.apache.hadoop.yarn.exceptions.YarnExcepti
on: Failed to submit application_1585828046495_0006 to YARN : User hiveuser21 cannot submit applications to que
ue root.default(requested queuename is default))'
java.io.IOException: org.apache.hadoop.yarn.exceptions.YarnException: Failed to submit application_158582804649
5_0006 to YARN : User hiveuser21 cannot submit applications to queue root.default(requested queuename is defaul
t)
```

(5) 角色授权

在[集群权限/角色管理]页面，单击 hive21 角色操作列的<编辑>按钮，修改角色组件权限设置。本示例授予 hive21 角色队列 default 的 submit-app 权限。

图3-14 为 hive21 角色授予权限



(6) 重新执行插入表 hivetest.hivetab1 操作。如图 3-15 所示，表明 hivetest.hivetab1 数据表成功插入数据

图3-15 插入成功

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> insert into hivetest.hivetab1 values(1, 'abc');
1 row affected (33.431 seconds)
INFO  : Compiling command(queryId=hive_20200403101900_ff3bcf5d-1a1a-41d3-80f7-62140dd3c11b): insert into hivete
st.hivetab1 values(1, 'abc')
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:_col0, type:int, comment:null), FieldSchem
a(name:_col1, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200403101900_ff3bcf5d-1a1a-41d3-80f7-62140dd3c11b); Time tak
en: 1.153 seconds
INFO  : Executing command(queryId=hive_20200403101900_ff3bcf5d-1a1a-41d3-80f7-62140dd3c11b): insert into hivete
st.hivetab1 values(1, 'abc')
WARN  :
INFO  : Query ID = hive_20200403101900_ff3bcf5d-1a1a-41d3-80f7-62140dd3c11b
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in serial mode
INFO  : Number of reduce tasks is set to 0 since there's no reduce operator
INFO  : number of splits:1
INFO  : Submitting tokens for job: job_1585828046495_0007
INFO  : Executing with tokens: [Kind: kms-dt, Service: 10.121.47.157:9292, Ident: (kms-dt owner=hiveuser21, ren
ewer=yarn, realUser=hive, issueDate=1585880342123, maxDate=1586485142123, sequenceNumber=30, masterKeyId=2), Ki
nd: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:mycluster, Ident: (token for hiveuser21: HDFS_DELEGATION_TOKEN owne
r=hiveuser21, renewer=yarn, realUser=hive/hppnode1.hde.com@TESTSHARE.COM, issueDate=1585880342043, maxDate=1586
485142043, sequenceNumber=61, masterKeyId=8), Kind: kms-dt, Service: 10.121.47.156:9292, Ident: (kms-dt owner=h
iveuser21, renewer=yarn, realUser=hive, issueDate=1585880342066, maxDate=1586485142066, sequenceNumber=28, mast
erKeyId=2), Kind: HIVE_DELEGATION_TOKEN, Service: HiveServer2ImpersonationToken, Ident: 00 0a 68 69 76 65 75 73
 65 72 32 31 0a 68 69 76 65 75 73 65 72 32 31 23 68 69 76 65 2f 68 70 70 6e 6f 64 65 31 2e 68 64 65 2e 63 6f 6d
 40 54 45 53 54 53 48 41 52 45 2e 43 4f 4d 8a 01 71 3d cc de 80 8a 01 71 61 d9 62 80 8e 09 a8 0b]
INFO  : The url to track the job: http://hppnode2.hde.com:8088/proxy/application_1585828046495_0007/
INFO  : Starting Job = job_1585828046495_0007, Tracking URL = http://hppnode2.hde.com:8088/proxy/application_15
85828046495_0007/
INFO  : Kill Command = /usr/hdp/current/hadoop-client/bin/hadoop job  -kill job_1585828046495_0007
INFO  : Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
INFO  : 2020-04-03 10:19:14,912 Stage-1 map = 0%,  reduce = 0%
INFO  : 2020-04-03 10:19:26,419 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 3.93 sec
INFO  : MapReduce Total cumulative CPU time: 3 seconds 930 msec
INFO  : Ended Job = job_1585828046495_0007
```

以授予 select 权限为例，对数据库 hivetest 数据表 hivetab1 授予 select 权限，操作步骤如下：

(1) 执行查询表 hivetest.hivetab1 操作。如图 3-16 所示，截图中表示 hiveuser21 用户没有 hivetest.hivetab1 数据表的 select 权限。

图3-16 无权限

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> select * from hivetest.hivetab1;
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [hiveuser21]
 does not have [SELECT] privilege on [hivetest/hivetab1/*] (state=42000,code=40000)
```

(2) 角色授权

在[集群权限/角色管理]页面，单击 hive21 角色操作列的<编辑>按钮，修改角色组件权限设置。本示例授予 hive21 角色 hivetest.hivetab1 数据表的 select 权限。

图3-17 授予权限

(3) 重新执行查询表 hivetest.hivetab1 操作。如图 3-18 所示，信息表明可以查询 hivetest.hivetab1 数据表数据。

图3-18 查询成功

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> select * from hivetest.hivetab1;
+---------------+-----------------+
| hivetab1.id   | hivetab1.name   |
+---------------+-----------------+
| 1             | abc             |
+---------------+-----------------+
1 row selected (0.673 seconds)
INFO  : Compiling command(queryId=hive_20200403102648_1b91fb35-c66b-44c8-a349-4beca025e39a): select * from hive
test.hivetab1
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:hivetab1.id, type:int, comment:null), Fiel
dSchema(name:hivetab1.name, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200403102648_1b91fb35-c66b-44c8-a349-4beca025e39a); Time tak
en: 0.319 seconds
INFO  : Executing command(queryId=hive_20200403102648_1b91fb35-c66b-44c8-a349-4beca025e39a): select * from hive
test.hivetab1
INFO  : Completed executing command(queryId=hive_20200403102648_1b91fb35-c66b-44c8-a349-4beca025e39a); Time tak
en: 0.002 seconds
INFO  : OK
```

3. 列级别权限控制

表3-5 授权配置

操作	权限要求（数据库示例为 hivetest，数据表示例为 hivetab1，列示例为 name）
insert into table	• Hive：数据库：*或 hivetest，数据表：*或 hivetab1，列：name，权限：update • YARN：队列：*或 default，权限：submit-app
select name from table	数据库：*或hivetest，数据表：*或hivetab1，列：name，权限：select

以授予 select 权限为例，对数据库 hivetest 数据表 hivetab1 的 name 列授予 select 权限，操作步骤如下：

(1) 新建用户 hiveuser02，授权前进行查询操作。如图 3-19 所示，提示当前用户没有表 hivetab1 的查询权限。

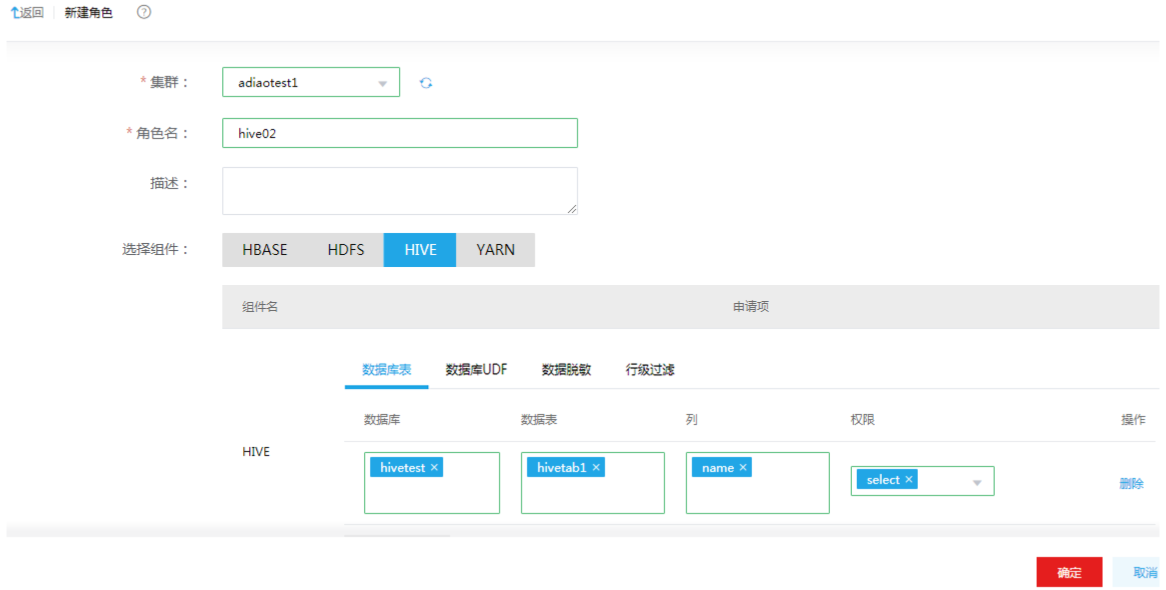
图3-19 执行结果

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> select name from hivetest.hivetab1;
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [hiveuser02]
 does not have [SELECT] privilege on [hivetest/hivetab1/*] (state=42000,code=40000)
```

(2) 角色授权

在[集群权限/角色管理]页面，授权 hivetest 库下 hivetab1 表的列 name 查询权限。

图3-20 授予权限



(3) 用户绑定角色

在[集群权限/用户管理]页面，hiveuser02 用户修改用户授权选择 hive02 角色，进行用户绑定角色。

图3-21 绑定角色

(4) 重新执行查询操作，select 操作执行成功。

图3-22 执行结果

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> select name from hivetest.hivetab1;
+--------+
| name  |
+--------+
| abc   |
+--------+
1 row selected (0.682 seconds)
```

以授予 update 权限为例，对数据库 hivetest 数据表 hivetab1 的 name 列授予 update 权限，操作步骤如下：

(1) 执行插入表 hivetest.hivetab1 列 name 的操作，提示当前没有 update 权限。

图3-23 无权限

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> insert into hivetest.hivetab1(name) values('sss');
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [hiveuser02]
 does not have [UPDATE] privilege on [hivetest/hivetab1/*] (state=42000,code=40000)
```

(2) 角色授权

在[集群权限/角色管理]页面，单击 hive02 角色操作列的<编辑>按钮，修改角色组件权限设置。本示例授予 hive02 角色 hivetest.hivetab1 中 name 列的 update 权限，及 YARN 队列 default 的 submit-app 权限。

图3-24 授予权限

(3) 重新执行插入操作，insert 操作执行成功。

图3-25 执行成功

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> insert into hivetest.hivetab1(name) values('sss');
1 row affected (33.032 seconds)
```

图3-26 查询插入结果

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> select name from hivetest.hivetab1;
+--------+
| name   |
+--------+
| abc    |
| sss    |
+--------+
2 rows selected (0.831 seconds)
```

4. UDF 权限控制

表3-6 授权配置

操作	权限要求（数据库示例为 hivetest，UDF 示例为 mylowerperm1）
create function	数据库：*或hivetest，UDF：*或mylowerperm1，权限：create
drop function	数据库：*或hivetest，UDF：*或mylowerperm1，权限：drop
select function	不需要授权
show functions	不需要授权

(1) 创建函数类文件

```
package com.test.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public final class LowerFunc extends UDF{
    public Text evaluate(final Text s){
        if(s == null){return null;}
        return new Text(s.toString().toLowerCase());
    }
}
```

其中 pom.xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.test</groupId>
    <artifactId>hiveUdfTest</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-exec</artifactId>
        <version>2.1.1-cdh6.2.0</version>
```

```
        </dependency>
    </dependencies>
    <build>

        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.4.3</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.0</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>

    </build>

</project>
```

(2) 编译打包后上传到测试集群，示例中打包为 hiveUdfTest-1.0-SNAPSHOT.jar。

(3) 新建用户 udfuser03，将 udf 测试 jar 包上传到 hdfs，授权前进行 create 操作。如图 3-27 所示，提示当前用户没有库 hivetest 下 udf mylowerperm1 的 create 权限。

图3-27 无权限

```
[root@hppnode1 ~]# su udfuser03
sh-4.2$ kinit
Password for udfuser03@TESTSHARE.COM:
sh-4.2$ hdfs dfs -mkdir -p /tmp/udflib
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$ hdfs dfs -put /opt/hiveUdfTest-1.0-SNAPSHOT.jar /tmp/udflib/
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> create function hivetest.mylowerperm1 as 'com.h3c.hive.udf.LowerFunc' using jar 'hdfs://mycluster/tmp/udflib/hiveUdfTest-1.0-SNAPSHOT.jar';
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [udfuser03] does not have [CREATE] privilege on [hivetest/mylowerperm1] (state=42000,code=40000)
```

```
sh-4.2$ hdfs dfs -chmod -R 777 /tmp/udflib
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/Stat
icLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/S
taticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

(4) 修改/tmp/udflib 目录权限为 777 目的是其他用户调用该函数时拥有该 jar 包的访问权限。

(5) 角色授权

在[集群权限/角色管理]页面，新建角色 udf03，授权 hivetest 库下 udf mylowerperm1 的 create 权限。

图3-28 授予权限

(6) 绑定角色

在[集群权限/用户管理]页面，udfuser03 用户修改用户授权选择 udf03 角色，进行用户绑定角色。

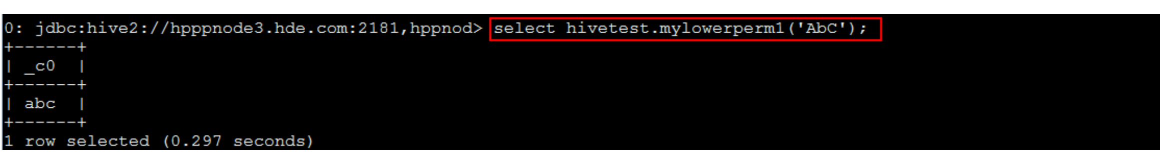
图3-29 绑定角色

(7) 重新执行操作，create 操作执行成功。

图3-30 执行成功

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> create function hivetest.mylowerperm1 as 'com.h3c.hive.udf.Lower
Func' using jar 'hdfs://mycluster/tmp/udflib/hiveUdfTest-1.0-SNAPSHOT.jar';
No rows affected (0.227 seconds)
INFO  : Compiling command(queryId=hive_20200403110812_f2e68a4f-9d31-4786-9d91-f1a260c65f33): create function hi
vetest.mylowerperm1 as 'com.h3c.hive.udf.LowerFunc' using jar 'hdfs://mycluster/tmp/udflib/hiveUdfTest-1.0-SNAP
SHOT.jar'
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20200403110812_f2e68a4f-9d31-4786-9d91-f1a260c65f33); Time tak
en: 0.056 seconds
INFO  : Executing command(queryId=hive_20200403110812_f2e68a4f-9d31-4786-9d91-f1a260c65f33): create function hi
vetest.mylowerperm1 as 'com.h3c.hive.udf.LowerFunc' using jar 'hdfs://mycluster/tmp/udflib/hiveUdfTest-1.0-SNAP
SHOT.jar'
INFO  : Starting task [Stage-0:FUNC] in serial mode
INFO  : Added [/tmp/4b90af3e-e0e1-4dcf-9349-a5d897ba0117_resources/hiveUdfTest-1.0-SNAPSHOT.jar] to class path
INFO  : Added resources: [hdfs://mycluster/tmp/udflib/hiveUdfTest-1.0-SNAPSHOT.jar]
INFO  : Completed executing command(queryId=hive_20200403110812_f2e68a4f-9d31-4786-9d91-f1a260c65f33); Time tak
en: 0.152 seconds
INFO  : OK
```

图3-31 查看创建的内容



说明

- 在新 Session 中执行 show functions 命令或 show functions like '*函数名*'命令可能查看不到新建函数，因为之前创建 function 时连接的 HiveServer2 和当前连接是不同的，执行 reload

functions 命令或重启当前连接的 HiveServer2，即可重新读取数据库的函数加载到缓存，此时重新执行命令即可查看到新建函数。

- 在创建或删除临时函数（如：create temporary function ...或 drop temporary function ...）时，由于临时函数属于会话 session 级别，不属于某个数据库，使用时也不需要追加库名前缀。因此授权时，选择数据库需要配置对所有库都有权限，即数据库配置为*。

5. 数据脱敏配置

数据脱敏是指按照某种规则对指定列进行脱敏，保证数据安全性。配置数据脱敏规则需先配置数据库表的 select 权限，然后进行数据脱敏配置。下面简单介绍数据脱敏配置的整体流程。

【示例一】

以配置脱敏规则为 MASK 为例，对数据库 mask01 数据表 t01 的 name 列进行数据脱敏，操作步骤如下：

(1) 配置数据库表 mask01.t01 的 select 权限，即对数据库 mask01 数据表 t01 授予 select 权限。新建用户 hivemask01 和角色 hivemask01，为角色授予 select 权限后，将该用户绑定到 hivemask01 角色，并执行查询表操作。详细过程请参考 3.3.2 2. 章节，查询结果如图 3-32 所示。

图3-32 mask01.t01 原始数据

```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t01;
+---------+-----------------+------------+
| t01.id  |    t01.name     | t01.birth  |
+---------+-----------------+------------+
| 410426  | testmaskhihihi  | 2020-07-21 |
| 123     | hi778899mask    | 2020-07-21 |
| 410     | 410aaXNNXbb426  | 2020-07-21 |
+---------+-----------------+------------+
```

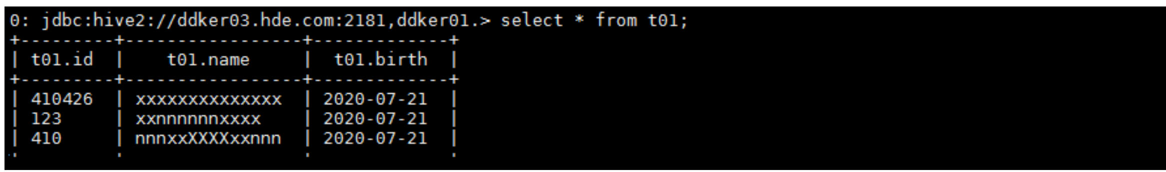
(2) 配置脱敏规则 MASK

在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的列 name 的脱敏规则为 MASK，即通过规则“x 替代小写字母，X 替代大写字母，n 替代数字”来实现对列 name 的数据脱敏。

图3-33 配置脱敏规则 MASK

(3) 执行查询表 mask01.t01 操作。如图 3-34 所示，信息表明对 mask01 库下 t01 表的列 name 脱敏成功。

图3-34 MASK 脱敏成功



```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t01;
+---------+----------------+------------+
| t01.id  |    t01.name    | t01.birth  |
+---------+----------------+------------+
| 410426  | xxxxxxxxxxxxxx | 2020-07-21 |
| 123     | xxnnnnnnxxxx   | 2020-07-21 |
| 410     | nnnxxXXXXxnnn  | 2020-07-21 |
```

【示例二】

以配置脱敏规则为 MASK_SHOW_LAST_4 为例，对数据库 mask01 数据表 t01 的 name 列进行数据脱敏，操作步骤如下：

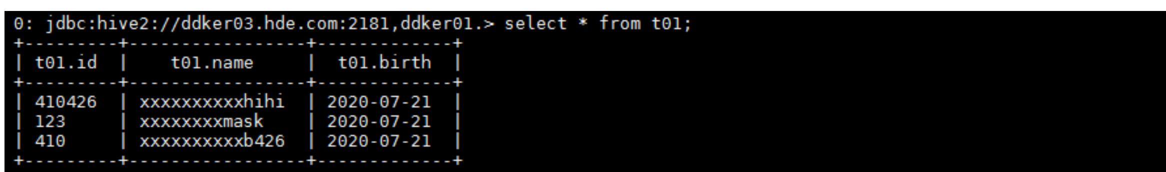
(1) 配置脱敏规则 MASK_SHOW_LAST_4

在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的列 name 的脱敏规则的脱敏规则为 MASK_SHOW_LAST_4，即通过规则“仅显示后四位，其他用 x 代替”来实现对列 name 的数据脱敏。

图3-35 配置脱敏规则 MASK_SHOW_LAST_4

(2) 执行查询表 mask01.t01 操作。如图 3-36 所示，信息表明对 mask01 库下 t01 表的列 name 脱敏成功。

图3-36 MASK_SHOW_LAST_4 脱敏成功



```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t01;
+---------+-----------------+------------+
| t01.id  |    t01.name     | t01.birth  |
+---------+-----------------+------------+
| 410426  | xxxxxxxxxxhihi  | 2020-07-21 |
| 123     | xxxxxxxxmask    | 2020-07-21 |
| 410     | xxxxxxxxxxb426  | 2020-07-21 |
+---------+-----------------+------------+
```

【示例三】

以配置脱敏规则为 MASK_SHOW_FIRST_4 为例，对数据库 mask01 数据表 t01 的 name 列进行数据脱敏，操作步骤如下：

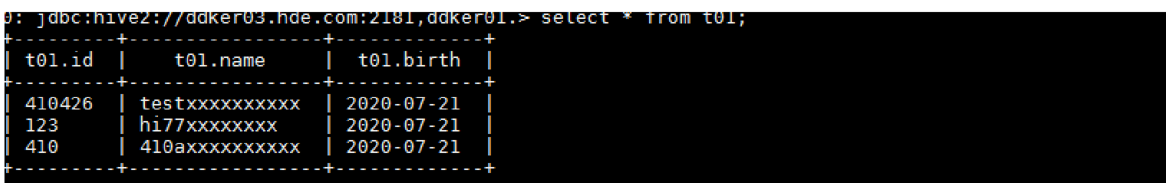
(1) 配置脱敏规则 MASK_SHOW_FIRST_4

在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的列 name 的脱敏规则的脱敏规则为 MASK_SHOW_FIRST_4，即通过规则“仅显示前四位，其他用 x 代替”来实现对列 name 的数据脱敏。

图3-37 配置脱敏规则 MASK_SHOW_FIRST_4

(2) 执行查询表 mask01.t01 操作。如图 3-38 所示，信息表明对 mask01 库下 t01 表的列 name 脱敏成功。

图3-38 MASK_SHOW_FIRST_4 脱敏成功



```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t01;
+---------+----------------+------------+
| t01.id  |    t01.name    | t01.birth  |
+---------+----------------+------------+
| 410426  | testxxxxxxxxxx | 2020-07-21 |
| 123     | hi77xxxxxxxx   | 2020-07-21 |
| 410     | 410axxxxxxxxxx | 2020-07-21 |
+---------+----------------+------------+
```

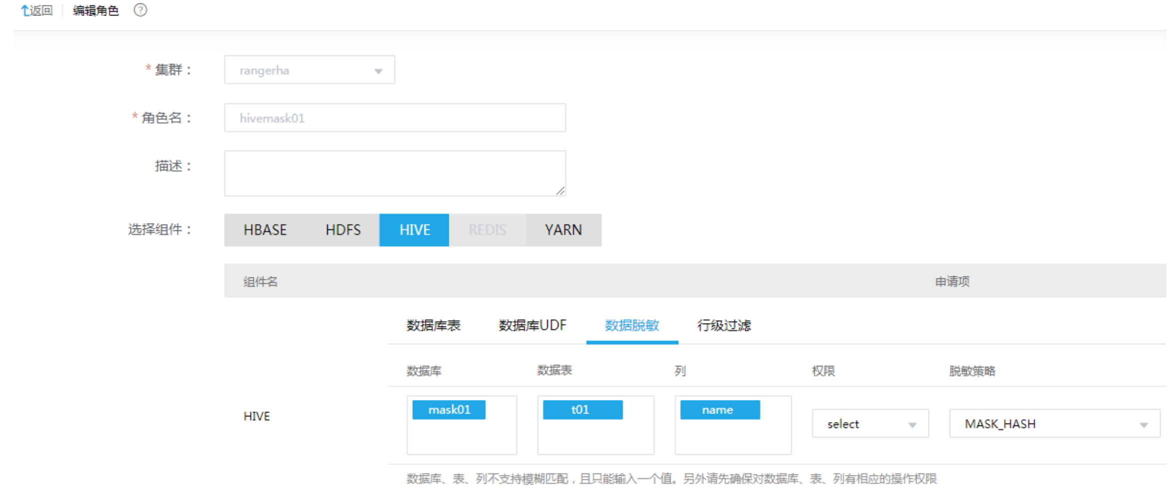
【示例四】

以配置脱敏规则为 MASK_HASH 为例，对数据库 mask01 数据表 t01 的 name 列进行数据脱敏，操作步骤如下：

(1) 配置脱敏规则 MASK_HASH

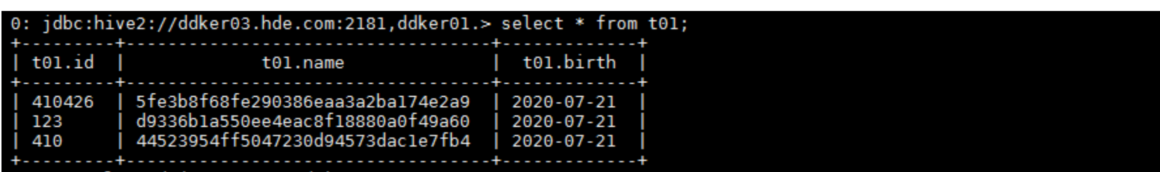
在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的列 name 的脱敏规则的脱敏规则为 MASK_HASH，即通过规则“Hash 值替代原值”来实现对列 name 的数据脱敏。

图3-39 配置脱敏规则 MASK_HASH



(2) 执行查询表 mask01.t01 操作。如图 3-40 所示，信息表明对 mask01 库下 t01 表的列 name 脱敏成功。

图3-40 MASK_HASH 脱敏成功



```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t01;
+---------+-----------------------------------+-------------+
| t01.id  |              t01.name             |  t01.birth  |
+---------+-----------------------------------+-------------+
| 410426  | 5fe3b8f68fe290386eaa3a2ba174e2a9  | 2020-07-21  |
| 123     | d9336b1a550ee4eac8f18880a0f49a60  | 2020-07-21  |
| 410     | 44523954ff5047230d94573dac1e7fb4  | 2020-07-21  |
+---------+-----------------------------------+-------------+
```

【示例五】

以配置脱敏规则为 MASK_NULL 为例，对数据库 mask01 数据表 t01 的 name 列进行数据脱敏，操作步骤如下：

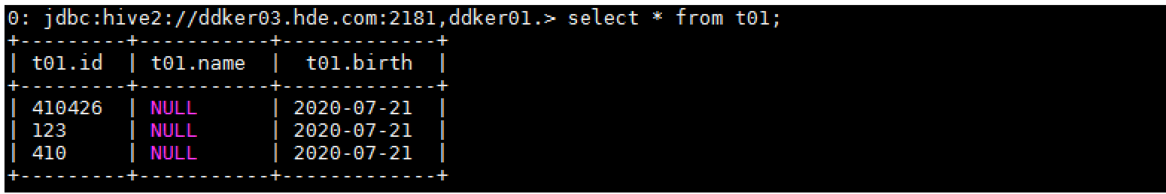
(1) 配置脱敏规则 MASK_NULL

在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的列 name 的脱敏规则的脱敏规则为 MASK_NULL，通过规则“NULL 值替代原值”来实现对列 name 的数据脱敏。

图3-41 配置脱敏规则 MASK_NULL

(2) 执行查询表 mask01.t01 操作。如图 3-42 所示，信息表明对 mask01 库下 t01 表的列 name 脱敏成功。

图3-42 MASK_NULL 脱敏成功



```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t01;
+---------+----------+------------+
| t01.id  | t01.name |  t01.birth |
+---------+----------+------------+
| 410426  | NULL     | 2020-07-21 |
| 123     | NULL     | 2020-07-21 |
| 410     | NULL     | 2020-07-21 |
+---------+----------+------------+
```

【示例六】

以配置脱敏规则为 MASK_NONE 为例，对数据库 mask01 数据表 t01 的 name 列进行数据脱敏，操作步骤如下：

(1) 配置脱敏规则 MASK_NONE

在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的列 name 的脱敏规则的脱敏规则为 MASK_NONE，即“显示原值”。

图3-43 配置脱敏规则 MASK_NONE

(2) 执行查询表 mask01.t01 操作。如图 3-44 所示，数据显示原值。

图3-44 MASK_NONE 脱敏成功。

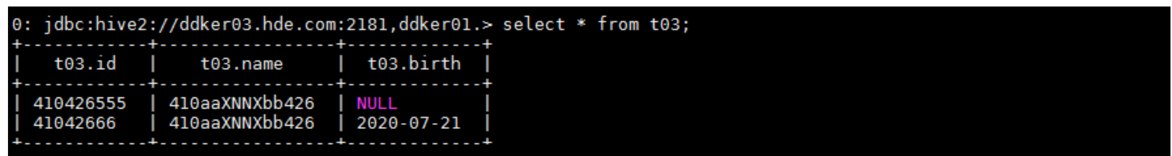
```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t01;
+---------+-----------------+-------------+
| t01.id  |    t01.name     |  t01.birth  |
+---------+-----------------+-------------+
| 410426  | testmaskhihihi  | 2020-07-21  |
| 123     | hi778899mask    | 2020-07-21  |
| 410     | 410aaXNNXbb426  | 2020-07-21  |
+---------+-----------------+-------------+
```

【示例七】

以配置脱敏规则为 MASK_DATE_SHOW_YEAR 为例，对数据库 mask01 数据表 t03 的 birth 列进行数据脱敏，操作步骤如下：

(1) 配置数据库表 mask01.t03 的 select 权限，即对数据库 mask01 数据表 t03 授予 select 权限，执行查询表操作。详细过程请参考 3.3.2 2. 章节，查询结果如图 3-45 所示。

图3-45 mask01.t03 原始数据



```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t03;
+------------+-----------------+-------------+
|   t03.id   |    t03.name     |  t03.birth  |
+------------+-----------------+-------------+
| 410426555  | 410aaXNNXbb426  | NULL        |
| 41042666   | 410aaXNNXbb426  | 2020-07-21  |
+------------+-----------------+-------------+
```

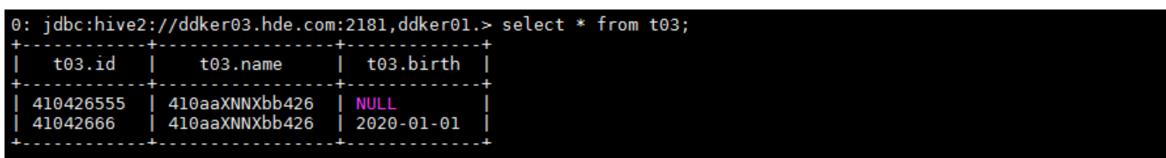
(2) 配置脱敏规则 MASK_DATE_SHOW_YEAR

在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的列 birth 的脱敏规则的脱敏规则为 MASK_DATE_SHOW_YEAR。

图3-46 配置脱敏规则 MASK_DATE_SHOW_YEAR

(3) 执行查询表 mask01. t03 操作。如图 3-47 所示，若 birth 列为 date 类型，通过规则“显示日期年份，月份和日期均用 01 代替”来实现对列 birth 的数据脱敏。

图3-47 MASK_DATE_SHOW_YEAR 脱敏成功



```
0: jdbc:hive2://ddker03.hde.com:2181,ddker01.> select * from t03;
+------------+----------------+-------------+
|   t03.id   |    t03.name    |  t03.birth  |
+------------+----------------+-------------+
| 410426555  | 410aaXNNXbb426 | NULL        |
| 41042666   | 410aaXNNXbb426 | 2020-01-01  |
+------------+----------------+-------------+
```

注意：若 birth 列为 string 类型，配置脱敏规则 MASK_DATE_SHOW_YEAR 时，年月日均用 x 替换来实现脱敏。

6. 行级过滤配置

配置行级过滤规则需先配置数据库表的 select 权限，然后配置行级过滤权限。对数据库 mask01 数据表 t01 配置行级过滤策略，操作步骤如下：

(1) 配置数据库表 mask01.t01 的 select 权限，即对数据库 mask01 数据表 t03 授予 select 权限，执行查询表操作。详细过程请参考 3.3.2 2. 章节（本章仍以 hivemask01 用户和 hivemask01 角色为例进行说明），当前若已完成该配置，则跳过此步。

(2) 配置行级过滤规则

在[集群权限/角色管理]页面，单击角色列表中 hivemask01 角色名进入角色详情页面，在页面右上角单击<编辑>按钮，配置 mask01 库下 t01 表的的过滤规则为“id>123 and name like '%mask%'”（若配置为空则不过滤）。

图3-48 配置行级过滤规则

(3) 执行查询表 mask01. t01 操作。如图 3-49 所示，即为“id>123 and name like '%mask%'”过滤后的结果。

图3-49 行级过滤结果

```
0: jdbc:hive2://ccdd03.hde.com:2181,ccdd02.hd> select id, name  from t01;
+---------+----------------+
|   id    |      name      |
+---------+----------------+
| 410426  | testmaskhihihi |
+---------+----------------+
1 row selected (0.655 seconds)
```

3.4 租户管理

注意

- 新增租户时，要求系统中必须已成功创建租户模式的 Hadoop 集群。
- 关于租户管理模块功能的配置或使用详情请参见产品在线联机帮助。

多个租户之间共享一套集群，共享网络和集群资源，并且不同租户之间保证资源隔离。

- 新增租户

  普通用户在自己创建的租户集群中申请租户时，无需审批，会直接触发新增租户的操作。普通用户在其他用户创建的租户集群中申请租户时，则需要走流程审批，待审批人审批通过后才能触发新增租户的操作。管理员用户新增租户时，无需审批，会直接触发新增租户的操作。

- 租户管理操作

  普通用户在自己创建的租户集群中执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。普通用户在其他用户创建的租户集群中执行租户续期、资源扩缩容操作时，则需要走流程审批，待审批人审批通过后才能触发相关操作。管理员用户执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。

3.4.1 租户介绍

租户申请的 Hive 资源对应一个或多个 database，每个 database 具有独立的存储和计算资源，用户可根据实际需要申请 Hive 组件资源。

【说明】

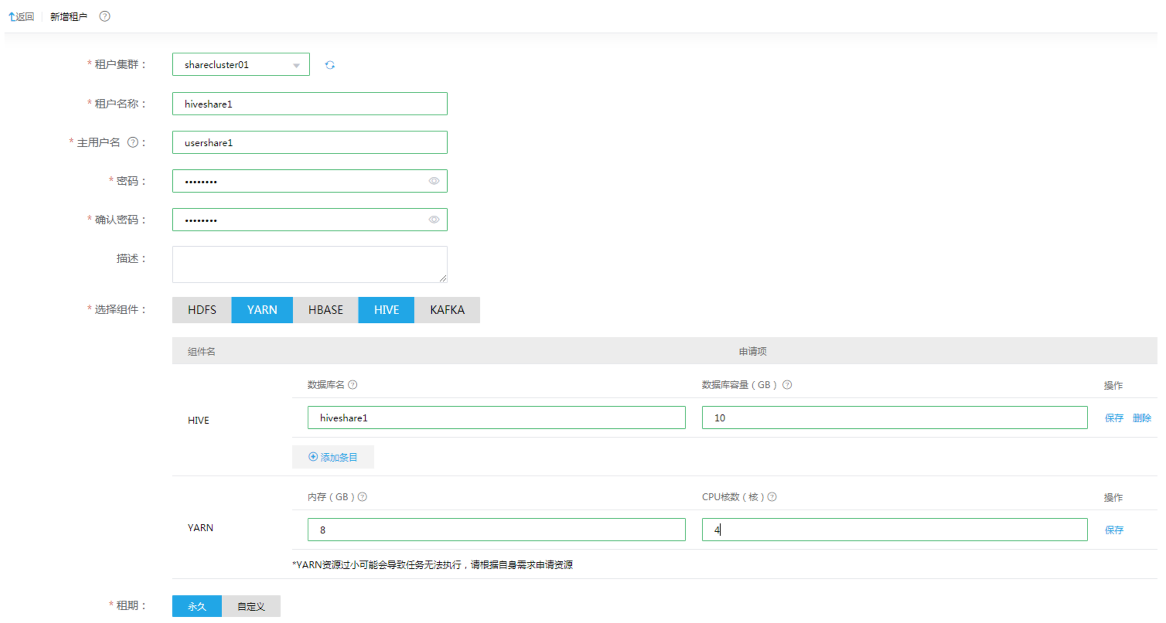
租户申请 Hive 资源时，Hive 依赖 YARN，即申请 Hive 资源必须同时申请 YARN 资源。

- Hive 的计算资源由 YARN 进行分配和调度。
- Hive 的存储资源对应 HDFS 资源。

3.4.2 新增租户

(1) 在[集群权限/租户管理]页面，单击<新增租户>按钮，跳转至新增租户页面，如图 3-50 所示。在租户集群 sharecluster01 中新增租户 hiveshare1，主用户 usershare1，并为该租户配置 Hive 组件资源（数据库名 hiveshare1、数据库容量 10GB）、YARN 组件资源（内存 8GB、CPU 核数 4）。使用 Hive 资源需要依赖 YARN 资源。

图3-50 新增租户

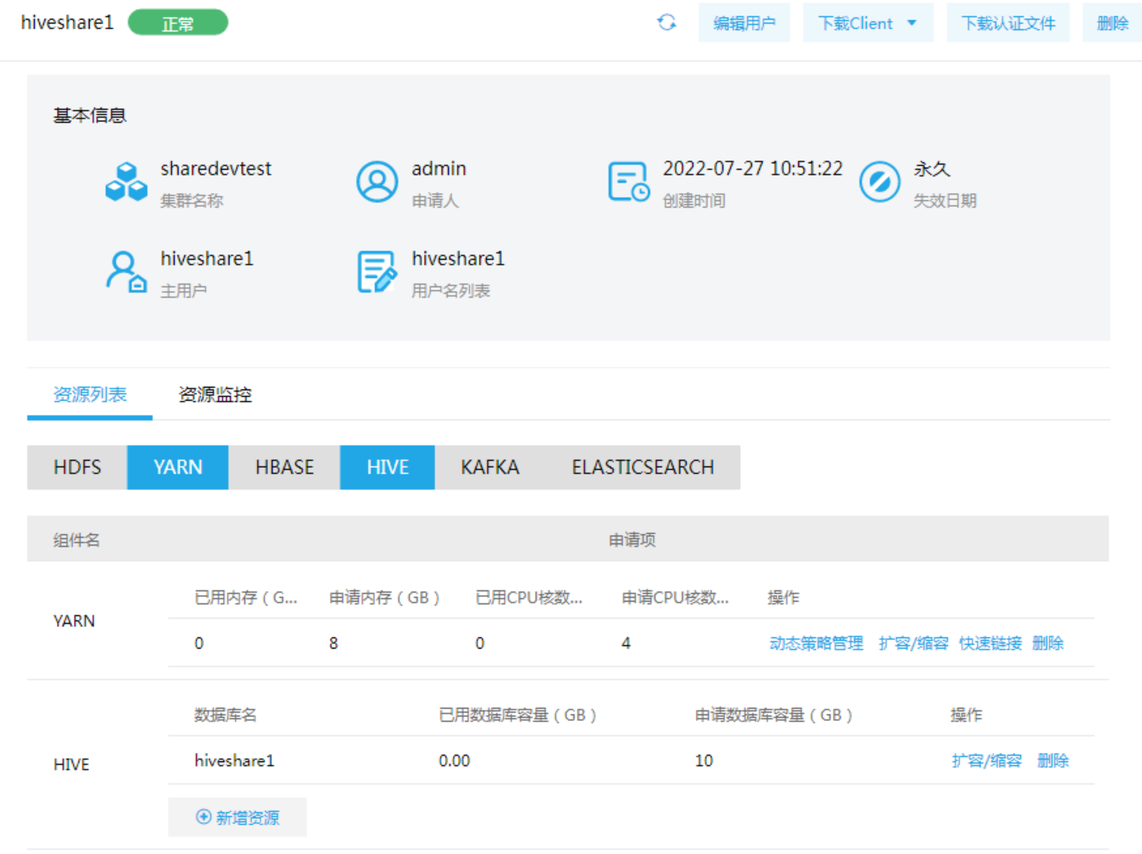


(2) 新增租户成功后，用户可在租户列表查看到已创建的租户，同时可以看到其所属集群、申请人、用户名列表、创建时间、失效时间等相关信息，如图 3-51 所示。

图3-51 查看租户

(3) 单击租户名称，可查看租户详情，如图3-52所示，可以看到对应的Hive数据库名和容量、YARN队列和容量等信息。用户usershare1拥有该租户资源的所有权限，若资源不够/过多时，可编辑租户对其执行扩容/缩容操作。

图3-52 查看租户详情



(4) 确认租户

租户创建成功后，其中的用户usershare1、队列hiveshare1、数据库hiveshare1均创建成功。

- Hive：数据库hiveshare1创建成功，大小为10G。

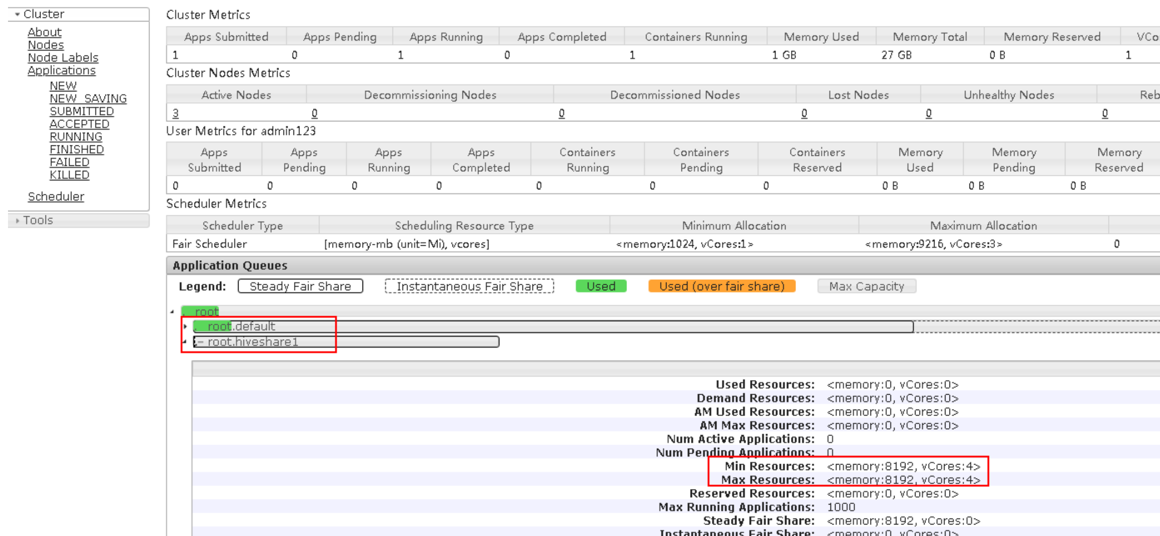
图3-53 查看Hive资源

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> show create database hiveshare1;
+----------------------------------------------------+
|                   createdb_stmt                    |
+----------------------------------------------------+
| CREATE DATABASE `hiveshare1`                       |
| LOCATION                                           |
|   'hdfs://mycluster/warehouse/tenant-hive/hiveshare1' |
+----------------------------------------------------+
3 rows selected (0.718 seconds)
INFO  : Compiling command(queryId=hive_20200403111959_e813352f-6508-4041-a5d7-8821b1533176): show create databa
se hiveshare1
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:createdb_stmt, type:string, comment:from d
eserializer)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200403111959_e813352f-6508-4041-a5d7-8821b1533176); Time tak
en: 0.317 seconds
INFO  : Executing command(queryId=hive_20200403111959_e813352f-6508-4041-a5d7-8821b1533176): show create databa
se hiveshare1
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20200403111959_e813352f-6508-4041-a5d7-8821b1533176); Time tak
en: 0.037 seconds
INFO  : OK
```

```
sh-4.2$ hdfs dfs -count -q -v -h hdfs://mycluster/warehouse/tenant-hive/hiveshare1
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
       QUOTA       REM_QUOTA     SPACE_QUOTA REM_SPACE_QUOTA    DIR_COUNT   FILE_COUNT       CONTENT_SIZE PATHNAME
        none             inf            10 G          10.0 G            2            1                290 hdfs://mycluster/warehouse/tenant-hive/hiveshare1
```

- YARN：资源队列由原来的 root 下只有 default 队列变为 default 和 hiveshare1 队列，且 hiveshare1 的队列大小为 8G 4core，可通过 ResourceManager UI 页面 Scheduler 查看。

图3-54 查看 YARN 资源



3.4.3 租户使用操作示例

注意

- 租户模式的集群缺省开启 Kerberos 认证，在使用租户时需要通过该租户的用户对应的认证文件，对租户的用户进行身份认证。租户的用户认证文件的下载在租户列表页面或租户详情页面执行，关于对租户资源的用户进行认证的方式与集群中用户的身份认证方式相同，详情请参见 2.3.2 1. Kerberos 环境下的用户身份认证。
- Hive 资源的租户包括客户端（即组件 Client），系统提供了下载 Hive 客户端的功能。在客户端节点上安装 Hive 的 Client 后，即可连接租户中的此组件，执行组件维护、任务管理等操作。租户资源组件 Client 的下载在租户列表页面执行，关于租户资源组件 Client 的安装方式与集群组件 Client 相同，详情请参见 3.1 Client 下载/安装/使用/卸载。

在 hiveshare1 数据库中执行创建表并插入、查询数据等操作。

图3-55 执行创建表并插入、查询数据等操作

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> use hiveshare1;
No rows affected (0.295 seconds)
INFO  : Compiling command(queryId=hive_20200403112139_74a9d9f9-15ad-4051-973a-a0b4cda8d20e): use hiveshare1
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20200403112139_74a9d9f9-15ad-4051-973a-a0b4cda8d20e); Time tak
en: 0.041 seconds
INFO  : Executing command(queryId=hive_20200403112139_74a9d9f9-15ad-4051-973a-a0b4cda8d20e): use hiveshare1
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20200403112139_74a9d9f9-15ad-4051-973a-a0b4cda8d20e); Time tak
en: 0.24 seconds
INFO  : OK
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> create table hivetab1(id int, name string);
No rows affected (0.349 seconds)
INFO  : Compiling command(queryId=hive_20200403112156_93ded7c1-e2fb-4824-b95f-63c462a70950): create table hivet
ab1(id int, name string)
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20200403112156_93ded7c1-e2fb-4824-b95f-63c462a70950); Time tak
en: 0.041 seconds
INFO  : Executing command(queryId=hive_20200403112156_93ded7c1-e2fb-4824-b95f-63c462a70950): create table hivet
ab1(id int, name string)
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20200403112156_93ded7c1-e2fb-4824-b95f-63c462a70950); Time tak
en: 0.09 seconds
INFO  : OK
```

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> insert into hivetab1 values(1, 'test');
1 row affected (32.794 seconds)
INFO  : Compiling command(queryId=hive_20200403112211_22e2f68f-3807-49e1-b0d8-3f0e3cc6dc61): insert into hiveta
b1 values(1, 'test')
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:_col0, type:int, comment:null), FieldSchem
a(name:_col1, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200403112211_22e2f68f-3807-49e1-b0d8-3f0e3cc6dc61); Time tak
en: 0.796 seconds
INFO  : Executing command(queryId=hive_20200403112211_22e2f68f-3807-49e1-b0d8-3f0e3cc6dc61): insert into hiveta
b1 values(1, 'test')
WARN  :
INFO  : Query ID = hive_20200403112211_22e2f68f-3807-49e1-b0d8-3f0e3cc6dc61
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in serial mode
INFO  : Number of reduce tasks is set to 0 since there's no reduce operator
INFO  : number of splits:1
INFO  : Submitting tokens for job: job_1585828046495_0009
```

```
0: jdbc:hive2://hpppnode3.hde.com:2181,hppnod> select * from hivetab1;
+--------------+----------------+
| hivetab1.id  | hivetab1.name  |
+--------------+----------------+
| 1            | test           |
+--------------+----------------+
1 row selected (0.696 seconds)
INFO  : Compiling command(queryId=hive_20200403112421_7d4df78f-b314-4bc5-b90c-52be4dfcf6a4): select * from hive
tab1
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:hivetab1.id, type:int, comment:null), Fiel
dSchema(name:hivetab1.name, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200403112421_7d4df78f-b314-4bc5-b90c-52be4dfcf6a4); Time tak
en: 0.56 seconds
INFO  : Executing command(queryId=hive_20200403112421_7d4df78f-b314-4bc5-b90c-52be4dfcf6a4): select * from hive
tab1
INFO  : Completed executing command(queryId=hive_20200403112421_7d4df78f-b314-4bc5-b90c-52be4dfcf6a4); Time tak
en: 0.002 seconds
INFO  : OK
```

3.5 备份恢复

备份恢复可以提供大数据平台跨集群之间的数据同步功能，当前版本支持对 Hive 组件中的指定数据进行同步备份，以保证数据内容不丢失。

备份恢复功能主要是通过创建同步任务实现集群间的数据同步能力。使用同步任务功能，可以为集群中的数据提供同步能力，以满足日常数据备份的需求，保证系统或机器故障时的数据不丢失。

3.5.1 新建 Hive 同步任务

注意

- 源集群为同步任务的数据输出集群（一般指新建同步任务的本集群）；目的集群为同步任务的数据输入集群。
- 新建同步任务时，要求源集群与目的集群的安全管理策略相同，即同时开启 Kerberos 认证或同时都没有开启 Kerberos 认证。
- 新建同步任务时，要求源集群与目的集群的集群类型、集群模式均相同。
- 新建同步任务时，要求源集群与目的集群的集群名称、集群中节点主机名均不相同，否则配置跨集群互信时可能出错。
- Hive 同步任务不支持事务表的同步，不支持多个主集群备份到同一个备集群。
- Hive 同步任务为周期性调度任务。Hive 同步任务周期性执行时，默认首次执行是全量数据备份，后续执行是增量数据备份，备份周期间隔时间内的数据不会实时同步到目的集群。
- 使用 Hive 同步任务备份 HDFS 加密区的数据时，需要根据集群的配置情况进行相关配置修改，详情请参见 3.5.4 HDFS 加密区数据同步。

集群在使用过程中，根据实际需要，可新建 Hive 同步任务，将源集群中的 Hive 某些库或表中的数据周期性拷贝到目的集群中。

1. 前提条件

- 新建 Hive 同步任务前，需要对源集群与目的集群配置跨集群互信。配置方法详情请参见 3.5.2 源集群和目的集群配置互信。
- 新建 Hive 同步任务前，需要根据集群的配置情况进行相关配置修改，详情请参见 3.5.3 Hive 同步任务相关配置修改。

2. 新建 Hive 同步任务

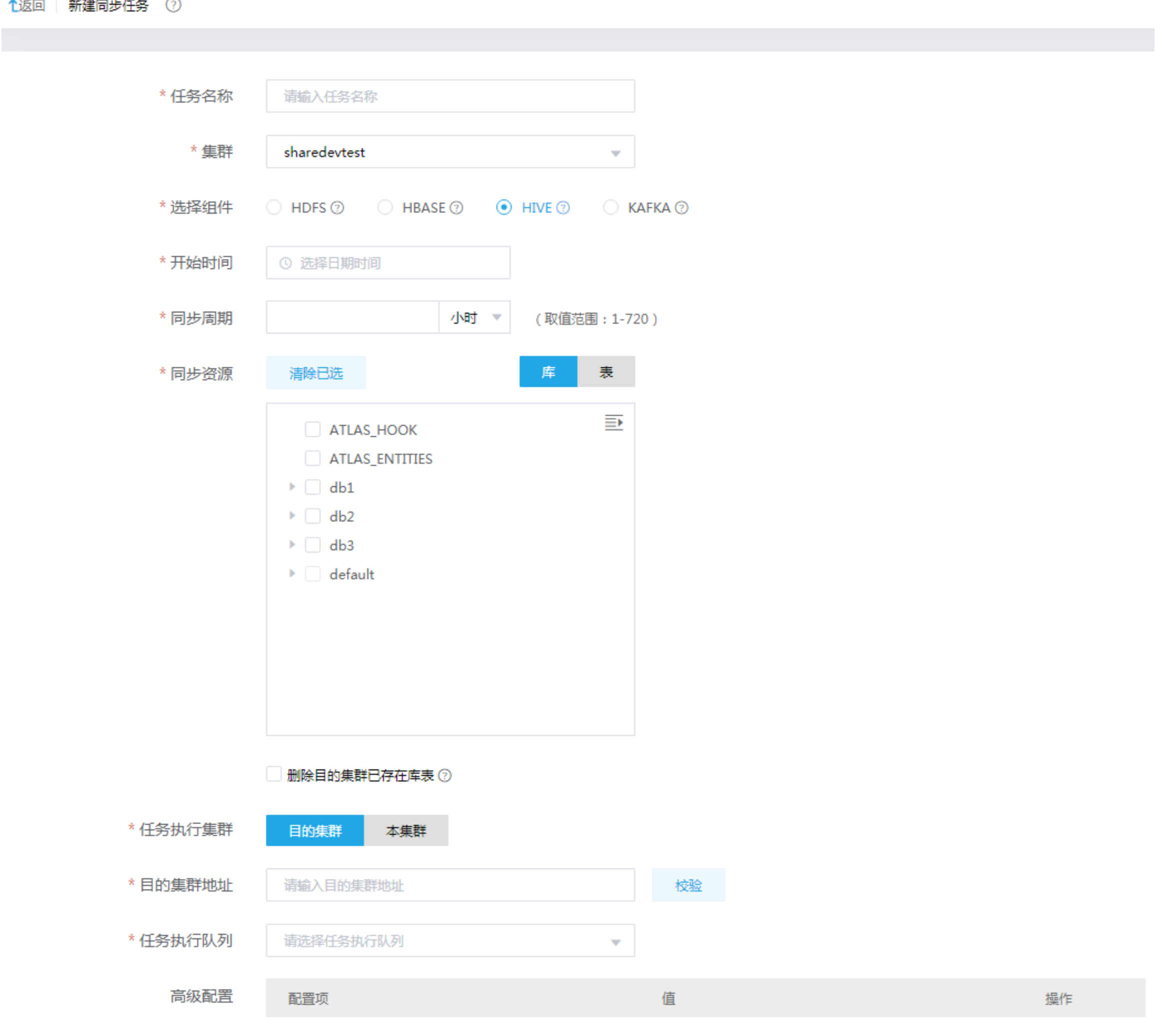
新建 Hive 同步任务的前提条件准备完成后，即可开始创建 Hive 同步任务。步骤如下：

(1) 在[集群管理/备份恢复]页面，选择[同步任务]页签，单击<新建同步任务>按钮，进入新建同步任务页面。

(2) 选择 Hive 组件，如图 3-56 所示，根据提示配置对应参数项的值，关于各参数项配置详情请参见产品在线联机帮助。

(3) 相关信息配置完成后，单击<新建>按钮即可成功新建 Hive 同步任务，此时任务到达调度时间后即可被执行。

图3-56 新建 Hive 同步任务



3.5.2 源集群和目的集群配置互信

注意

不同集群之间可通过组件同步任务进行数据同步，但创建同步任务之前必须配置源集群和目的集群互信。

示例集群如下：

- 源集群
  - 开启 Kerberos 认证集群 clusterA，kerberos realm 为 CLUSTERA.COM。
- 目的集群
  - 开启 Kerberos 认证集群 clusterB，kerberos realm 为 CLUSTERB.COM。

1. 开启 kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

(3) 修改源集群和目的集群中所有节点的/etc/krb5.conf 文件，要求：

- 修改 realms，要求同时包含源集群和目的集群的 realms 内容（互相拷贝即可），如图 3-57 所示。
- 修改 domain_realm，要求同时包含源集群和目的集群的 domain_realm 内容。
  - 当源集群与目的集群的主机名后缀相同时，需要将 domain_realm 内容修改为“集群中各节点主机名=对应的 realms 名”。示例：源集群和备集群的主机名后缀均为.hde.com，则配置如图 3-58 所示。
  - 当源集群与目的集群的主机名后缀不同时，则可直接将两个集群的 domain_realm 中内容合并（不需要修改，直接互相拷贝即可）。示例：源集群的主机名后缀为.hde.com，目的集群的主机名后缀为.hadoop.com，则配置如图 3-59 所示。

图3-57 realms 修改后示例

```
[realms]
    CLUSTERA.COM = {
        kdc = clustera1.hde.com
        admin_server = clustera1.hde.com
        database_module = openldap_ldapconf
    }
    CLUSTERB.COM = {
        kdc = clusterb1.hde.com
        admin_server = clusterb1.hde.com
        database_module = openldap_ldapconf
    }
```

图3-58 domain_realm 修改后示例（源集群和目的集群主机名后缀相同）

```
[domain_realm]
   clustera1.hde.com=CLUSTERA.COM
   clustera2.hde.com=CLUSTERA.COM
   clustera3.hde.com=CLUSTERA.COM
   clusterb1.hde.com=CLUSTERB.COM
   clusterb2.hde.com=CLUSTERB.COM
   clusterb3.hde.com=CLUSTERB.COM
```

图3-59 domain_realm 修改后示例（源集群和目的集群主机名后缀不同）

```
[domain_realm]
   .hde.com = CLUSTERA.COM
   hde.com = CLUSTERA.COM
   .hadoop.com = CLUSTERB.COM
   hadoop.com = CLUSTERB.COM
```

(4) 在源集群和目的集群的 Master 节点上，执行添加 principal 操作。

a. 在源集群 Master 节点上，分别执行以下两条命令，以添加 principal（命令执行后，需要分别输入密码）：

```
kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '
```

```
kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '
```

b. 在目的集群 Master 节点上，分别执行以下两条命令，以添加 principal（命令执行后，需要分别输入密码）：

```
kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '
```

```
kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '
```

【注意】

- CLUSTERA.COM 和 CLUSTERB.COM 分别为源集群和目的集群的域名，请根据实际情况进行修改。添加 principal 时需确保两个集群输入的密码相同（即上述四条命令运行后输入的密码均相同），且密码要求至少 8 位，否则会提示密码太短导致设置无效。
- 若添加 principal 时输入的密码不同，可在源集群和目的集群上进行删除，然后重新执行第 4 步添加 principal 的操作。删除命令如下：

```
kadmin.local -q ' delprinc krbtgt/CLUSTERA.COM@CLUSTERB.COM'
```

```
kadmin.local -q ' delprinc krbtgt/CLUSTERB.COM@CLUSTERA.COM'
```

(5) 源集群与目的集群互信配置完成后，可登录目的集群进行校验。校验方式示例：在目的集群后台切换至集群超级用户，执行命令 hdfs dfs -ls hdfs://<源集群 Active NameNode IP 地址>:8020/，查看源集群的 HDFS 是否可正常访问，若能正常访问则表示源集群与目的集群的互信配置成功。

2. 不开 kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

3.5.3 Hive 同步任务相关配置修改

- 开启 Kerberos 认证的环境

  若源集群和目的集群都开启了 Kerberos 认证，创建 Hive 同步任务之前，需进行以下配置修改：

  - 对源集群进行配置修改，说明如下：

    在源集群的[集群列表/集群详情/业务组件/HDFS 组件详情]页面，修改 HDFS 配置项 hadoop.security.auth_to_local 的值。要求在末尾 DEFAULT 之前增加内容“ RULE:[1:$1@$0](.*@CLUSTERB.COM)s/@.*// ”，其中 CLUSTERB.COM 为目的集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 HDFS 组件。

  - 对目的集群进行配置修改，说明如下：

在目的集群的[集群列表/集群详情/业务组件/HDFS 组件详情]页面，修改 HDFS 配置项 hadoop.security.auth_to_local 的值。要求在末尾 DEFAULT 之前增加内容“ RULE:[1:$1@$0](.*@CLUSTERA.COM)s/@.*// ”，其中 CLUSTERA.COM 为源集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 HDFS 组件。

- 不开 Kerberos 认证的环境

  若源集群和目的集群都未开启 Kerberos 认证，则在集群之间运行 Hive 同步任务时，不需要进行相关配置的修改。

3.5.4 HDFS 加密区数据同步

说明

无论集群是否开启 Kerberos，使用 Hive 同步任务备份 HDFS 加密区的数据时，需要在新建 Hive 同步任务时进行高级配置，将配置项 distcp.skip.crc 的值设置为 true，否则会抛出 Checksum mismatch 错误。

1. 开启 Kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，在集群之间备份 HDFS 加密区数据时，需提前进行以下配置修改：

- 对源集群进行配置修改，说明如下：

  在源集群的[集群列表/集群详情/系统组件/RANGER_KMS 组件详情]页面，修改 RANGER_KMS 配置项 hadoop.kms.authentication.kerberos.name.rules 的值。要求在末尾 DEFAULT 之前增加内容“ RULE:[1:$1@$0](.*@CLUSTERB.COM)s/@.*// ”，其中 CLUSTERB.COM 为目的集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 RANGER_KMS 组件。

- 对目的集群进行配置修改，说明如下：

  在目的集群的[集群列表/集群详情/系统组件/RANGER_KMS 组件详情]页面，修改 RANGER_KMS 配置项 hadoop.kms.authentication.kerberos.name.rules 的值。要求在末尾 DEFAULT 之前增加内容“RULE:[1:$1@$0](.*@CLUSTERA.COM)s/@.*// ”，其中 CLUSTERA.COM 为源集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启 RANGER_KMS 组件。

2. 不开启 Kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则在集群之间备份 HDFS 加密区数据时，不需要进行相关配置的修改。

3.5.5 Hive 备份恢复示例

示例两个集群在一个系统中，均开启 Kerberos 和安全管理，按照3.5.1 1. 前提条件配置完互信及 Hive 相关配置后，将源集群中的一个数据库同步到目的集群。

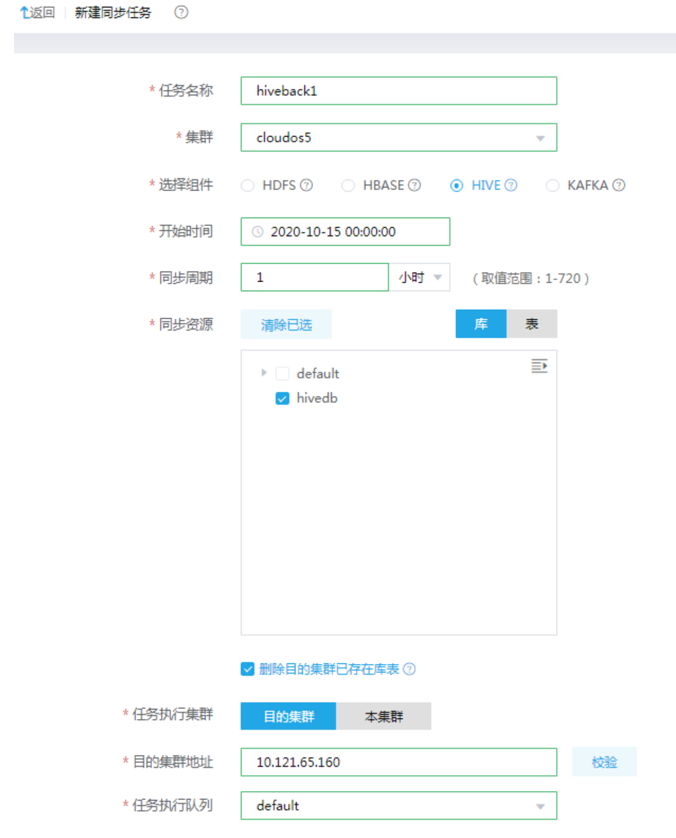
(1) 在目的集群后台切换至 hdfs 用户，查看源集群的 HDFS 是否可正常访问，如所示表明可正常访问说明源集群与目的集群互信配置成功：

图3-60 集群互信

```
[hdfs@test003 root]$ hdfs dfs -ls hdfs://10.121.65.200:8020/
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/Stat
icLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/S
taticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Found 15 items
drwxrwxrwt   - yarn   hadoop          0 2020-04-07 11:23 hdfs://10.121.65.200:8020/app-logs
drwxr-xr-x   - hdfs   hdfs            0 2020-04-07 11:04 hdfs://10.121.65.200:8020/apps
drwxr-xr-x   - yarn   hadoop          0 2020-04-06 21:04 hdfs://10.121.65.200:8020/ats
drwxr-xr-x   - hdfs   hdfs            0 2020-04-06 21:04 hdfs://10.121.65.200:8020/atsv2
drwx------   - diao1  hdfs            0 2020-04-07 11:20 hdfs://10.121.65.200:8020/diao1
drwxr-xr-x   - hdfs   hdfs            0 2020-04-06 21:04 hdfs://10.121.65.200:8020/hdp
drwx------   - livy   hdfs            0 2020-04-06 21:08 hdfs://10.121.65.200:8020/livy2-recovery
drwxr-xr-x   - mapred hdfs            0 2020-04-06 21:04 hdfs://10.121.65.200:8020/mapred
drwxr-xr-x   - hdfs   hdfs            0 2020-04-06 21:11 hdfs://10.121.65.200:8020/mlsql
drwxrwxrwx   - mapred hadoop          0 2020-04-06 21:04 hdfs://10.121.65.200:8020/mr-history
drwxrwxrwx   - spark  hadoop          0 2020-04-07 11:45 hdfs://10.121.65.200:8020/spark2-history
drwxrwxrwx   - hive   hadoop          0 2020-04-07 13:11 hdfs://10.121.65.200:8020/sparrow-history
drwxrwxrwx   - hdfs   hdfs            0 2020-04-07 13:08 hdfs://10.121.65.200:8020/tmp
drwxrwxrwx   - hdfs   hdfs            0 2020-04-07 11:23 hdfs://10.121.65.200:8020/user
drwxr-xr-x   - hdfs   hdfs            0 2020-04-06 21:06 hdfs://10.121.65.200:8020/warehouse
```

(2) 新建同步任务，填写开始时间、同步周期、目的集群地址等信息，默认选择任务执行集群为目的集群（不会影响源集群的业务），具体填写规则请参见联机帮助。单击<校验>按钮可获取任务执行队列，根据进行情况选取即可。

图3-61 新建同步任务



(3) 任务在调度时间会自动执行，也可手动单击<启动>执行即时备份，在<更多/运行记录>中可查看执行记录及日志详情。

图3-62 查看执行结果

图3-63 查看运行记录

(4) 登录到目的集群，可查看数据库表同步的数据正常。

图3-64 同步成功

```
0: jdbc:hive2://test002.hde.com:2181,test001.> select * from hivedb1.tab1;
+----------+------------+
| tab1.id  | tab1.name  |
+----------+------------+
| 1        | ss         |
+----------+------------+
1 row selected (7.237 seconds)
INFO  : Compiling command(queryId=hive_20200407153125_6d4df24e-0228-44ad-938d-d036fa45415a): select * from hive
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:tab1.id, type:int, comment:null), FieldSch
e, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200407153125_6d4df24e-0228-44ad-938d-d036fa45415a); Time tak
INFO  : Executing command(queryId=hive_20200407153125_6d4df24e-0228-44ad-938d-d036fa45415a): select * from hive
INFO  : Completed executing command(queryId=hive_20200407153125_6d4df24e-0228-44ad-938d-d036fa45415a); Time tak
INFO  : OK
0: jdbc:hive2://test002.hde.com:2181,test001.> select * from hivedb1.tab2;
+----------+------------+
| tab2.id  | tab2.name  |
+----------+------------+
| 1        | ds         |
+----------+------------+
1 row selected (1.653 seconds)
INFO  : Compiling command(queryId=hive_20200407153137_da699c28-1cda-4711-8403-0a0460555257): select * from hive
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:tab2.id, type:int, comment:null), FieldSch
e, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20200407153137_da699c28-1cda-4711-8403-0a0460555257); Time tak
INFO  : Executing command(queryId=hive_20200407153137_da699c28-1cda-4711-8403-0a0460555257): select * from hive
INFO  : Completed executing command(queryId=hive_20200407153137_da699c28-1cda-4711-8403-0a0460555257); Time tak
INFO  : OK
```

3.6 LDAP认证

注意

建议仅在未开启 Kerberos，开启权限管理的集群中，开启 LDAP 认证。

HiveServer2 支持多种认证方式，包括 none、ldap、kerberos、pam、custom，通过 hive.server2.authentication 参数可设置认证方式。

开启 LDAP 认证，需要修改如表 3-7 所示的高级配置，并在 Hive 组件的自定义配置 hive-site 下新增如表 3-8 所示配置。

表3-7 修改配置项

参数名称	参数值
hive.server2.authentication	LDAP
alert_ldap_username	集群超级用户
alert_ldap_password	集群超级用户密码

表3-8 新增配置

参数名称	参数值
hive.server2.authentication.ldap.url	/etc/openldap/ldap.conf文件中如下内容 ldap://hivedebug0908-vserver.hde.test.com/
hive.server2.authentication.ldap.baseDN	dc=hde,dc=com
hive.server2.authentication.ldap.groupDNPattern	CN=%s,ou=Groups,dc=hde,dc=com

参数名称	参数值
hive.server2.authentication.ldap.userDNPattern	uid=%s,ou=Users,dc=hde,dc=com

开启 LDAP 认证后，在连接集群中包含 Hive Client 的某台机器时，需指定参数进行连接：

beeline -u "jdbc:hive2://<HiveServer2 地址>:10000/" -n hive -p "password"

说明

指定参数连接时，在 beeline 命令中：

- -u：指定连接 HiveServer2 的地址和端口号。
- -n：指定连接所需用户名。
- -p：指定连接所需密码（使用 LDAP 认证时，需正确填写用户对应的密码。其他情况可为空值）。

3.7 事务性操作

3.7.1 ACID 的含义和使用

数据库事务具有四个特性，即 ACID。各特性含义如下：

- 原子性（Atomicity）：一个事务是一个不可分割的工作单位，事务中包含的操作要么都执行，要么都不执行。
- 一致性（Consistency）：事务必须是使数据库从一个一致性状态变为另一个一致性状态。
- 隔离性（Isolation）：一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不会互相干扰。
- 持久性（Durability）：指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。后续的其他操作或故障不应该对其有任何影响。

Hive 支持事务，能支持行级的 ACID，因此，在同一个分区中，一个应用在进行读操作时，另一个应用能进行新增操作，而不会相互影响。

Hive 支持事务的特性能用于以下情况：

- 流数据的采集
- 数据的修改，包括 INSERT、UPDATE 和 DELETE

Hive 中对事务的支持具有以下限制：

- 不支持 BEGIN、COMMIT 和 ROLLBACK，所有的操作都是自动提交
- 目前只有存储格式为 ORC 文件的才能支持事务
- 使用事务的表必须是分桶表
- 目前只支持快照级别的隔离
- 事务不能与已有的 ZooKeeper 和 in-memory lock managers 兼容
- 对于建表时已经指明了支持 ACID 属性的数据表，不能使用 ALTER TABLE 语句修改模式定义

3.7.2 开启事务

Hive 中的事务默认关闭，开启事务前需要进行相关参数的设置，参数相关配置如表 3-9 所示。

表3-9 开启事务的相关参数配置

配置参数	值
hive.txn.manager	org.apache.hadoop.hive.ql.lockmgr.DbTxnManager
hive.support.concurrency	true

3.7.3 事务表操作

- 创建事务表示例：

```
create table t1 (id int, name string)
clustered by (id) into 8 buckets
stored as orc
tblproperties('transactional'='true');
```

说明

- 为了使数据表支持 ACID，建表时必须将表的 transactional 属性设置为 true。
- 建表语句必须带有 into buckets 子句和 stored as orc tblproperties('transactional'='true')子句，但不能带有 stored by 子句。

- 使用 insert 命令向表中插入数据示例：

```
insert into table t1 values(1,"string1");
insert into table t1 values(2,"string2");
insert into table t1 values(3,"string3");
insert into table t1 values(4,"string4");
```

- 使用 update 命令修改表中数据示例：

```
update t1 set name = "string33" where id = 3;
```

- 使用 delete 命令删除表中数据示例：

```
delete from t1 where id = 4;
```

3.8 HiveServer2 高可用功能

安装 Hive 时，若集群开启高可用则默认会在两个节点上安装 HiveServer2，以提供 HA 功能。

HiveServer2 启动后会向 Zookeeper 注册临时节点，当用户以 JDBC 方式连接 HiveServer2 时，Hive 会随机选择一个节点上的 HiveServer2 与用户建立连接。若当前连接节点上的 HiveServer2 服务挂掉后，Hive 会自动将连接切换到其它节点的 HiveServer2 上。

说明

若在获取查询结果集时发生 HiveServer2 切换，则需重新执行 SQL 语句获取结果。（use、set 等操作也需要重新执行，否则 SQL 语句可能会执行失败。）

表3-10 HiveServer2 相关配置

参数名称	参数说明	参数值
hive-site中 hive.server2.zookeeper.namespace	HiveServer2在Zookeeper上注册的znode节点	hiveserver2
hive-site中 hive.server2.support.dynamic.service.discovery	HiveServer2是否支持HA	true
高级hive-env下hive-env template中 export HIVE2SERVER2_ZNODE_FILE=…	指定一个文件路径，HiveServer2在Zookeeper上注册的节点信息会保存在文件中。	/var/run/hive/hive2server2_znode.txt
高级hive-env下hive-env template中 export ZOOKEEPER_QUORUM=…	DateEngine集群中Zookeeper的uri	{{hive_zookeeper_quorum}}

使用 HA 时，JDBC 的 url 是 jdbc:hive2://<zk 集群地址>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=[hive.server2.zookeeper.namespace 参数值]。其中，zk 集群地址格式为“zk_node1host:zk_node1port,...”。

1. HiveServer2 使用 HA 示例

(1) 在集群中节点上安装多个 HiveServer2 并启动。在 Zookeeper 上可以看到注册的 znode 信息为：

```
[root@node1 ~]# zookeeper-client -server node1:2181
[zk: localhost:2181(CONNECTED) 0] ls /hiveserver2
[serverUri=node1.hde.com:10000;version=2.1.1-cdh6.2.0;sequence=0000000007,
serverUri=node2.hde.com:10000;version=2.1.1-cdh6.2.0;sequence=0000000006]
```

(2) 使用 beeline 连接 HiveServer2，示例代码如下：

```
[root@node1 ~]#beeline -u
"jdbc:hive2://node1:2181,node2:2181,node3:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2" -n hive -p ""
2020-02-12 15:51:41,477 INFO  jdbc.Utils (Utils.java:parseURL(443)) - Resolved authority: node1.hde.com:10000
2020-02-12 15:51:41,591 INFO  jdbc.HiveConnection (HiveConnection.java:logZkDiscoveryMessage(799)) - Connected to node1.hde.com:10000
2020-02-12 15:51:41,694 WARN  curator.CuratorZookeeperClient (CuratorZookeeperClient.java:<init>(96)) - session timeout [3000] is less than connection timeout [15000]
Connected to: Apache Hive (version 2.1.1-cdh6.2.0)
Driver: Hive JDBC (version 2.1.1-cdh6.2.0)
Transaction isolation: TRANSACTION_REPEATABLE_READ
1: jdbc:hive2://node1:2181,node2:2181/> create table mltable(id double,label double);
```

```
No rows affected (0.796 seconds)
1: jdbc:hive2://node1:2181,node2:2181/> show tables;
+------------+--+
| tab_name   |
+------------+--+
| mltable    |
+------------+--+
1 row selected (0.211 seconds)
```

说明

当前系统支持输入 beeline 后回车，可直接连接上 HiveServer2。

(3) 通过 Beeline 日志信息，可以看到此次连接了 node1 节点的 HiveServer2。用户可以手动将 node1 节点 HiveServer2 停掉，在 Beeline 中再次尝试执行 SQL 语句：

```
1: jdbc:hive2://node1:2181,node2:2181/> show tables;
+-----------+--+
| tab_name  |
+-----------+--+
| mltable   |
+-----------+--+
1 row selected (0.211 seconds)
```

可以看到，用户不需重新连接，JDBC 连接仍然有效。

3.9 Hive分区表生命周期管理

在 Hive 中支持对分区表的数据生命周期管理，为分区数据设置过期时间，定期释放过期分区存储空间，简化数据回收过程。

说明

对 Hive 分区表的数据生命周期管理，在使用过程中如遇到问题请及时联系技术支持。

3.9.1 总体要求

- 不影响现有表使用
- 设置表分区生命周期易于使用
- 支持修改分区过期时间
- 支持表级生命周期管理
- 根据分区数据的最后更新时间进行判断，超过分区生命周期时间进行清理
- 支持设置数据清理周期，默认为一小时

3.9.2 使用限制

- 仅支持分区表使用，非分区表进行忽略
- 分区过期检测周期为周期性操作。分区过期检测周期全局有效，不支持对单独的表设置，并且在检测周期时间内数据清理存在延迟
- 生命周期时间只支持天，且只能为整数
- 分区生命周期过期功能属于高风险操作，使用前需要充分了解，确定生命周期之外的分区数据不再使用后再进行配置，避免数据误清理

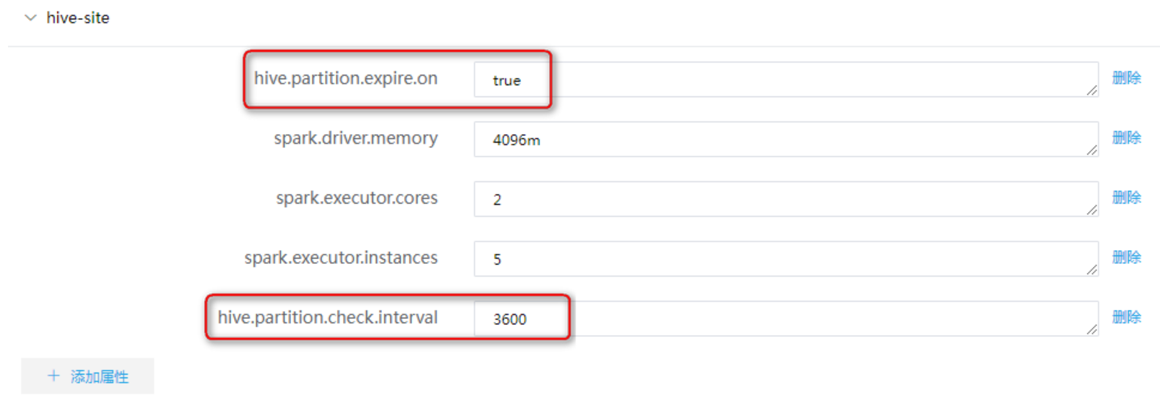
3.9.3 使用说明

说明

用户使用 Hive 分区表的数据生命周期管理时，需要先进入[组件管理/组件详情/配置]页面，然后在自定义配置的 hive-site 中配置分区生命周期参数。

用户在 hive-site 配置 Hive 分区生命周期参数如图 3-65 所示。

图3-65 分区生命周期参数配置



Hive 中分区生命周期参数说明如表 3-11 所示。

表3-11 分区生命周期参数说明

配置参数	参数说明
hive.partition.expire.on	是否开启分区过期线程，默认false，即不进行数据清理
hive.partition.check.interval	分区过期检测周期，分区清理线程启动后生效，若此参数不进行配置，默认为3600s

1. 分区生命周期设置语法

创建表时支持指定 TBLPROPERTIES 属性，不增加 Hive 语法，指定 lifecycle 参数，注意设定 lifecycle 后需要开启 hive.partition.expire.on 才可生效。

(1) 创建分区表时设置分区生命周期

创建表时通过 TBLPROPERTIES 指定 lifecycle 属性：

CREATE TABLE <table_name> (<col_name> <data_type>, <col_name> <data_type>, ...)
PARTITIONED BY (<col_name> <data_type>) TBLPROPERTIES('lifecycle'='days');

注：lifecycle 为表属性，days 为天数，只支持整数。

【操作示例】

a. 创建分区表testlifecycle01，并指定lifecycle属性值为1，如图3-66所示。执行命令如下：

create table testlifecycle01(id int,name string) partitioned by (age int,dept string,score int) tblproperties('lifecycle'='1');

图3-66 创建分区表并设置 lifecycle

```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> create table testlifecycle01(id int,name str
ing) partitioned by (age int,dept string,score int) tblproperties('lifecycle'='1');
No rows affected (0.367 seconds)
INFO  : Compiling command(queryId=hive_20211014161734_4df7b320-b3c1-4fe5-9b99-9f18a08914bb)
: create table testlifecycle01(id int,name string) partitioned by (age int,dept string,scor
e int) tblproperties('lifecycle'='1')
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20211014161734_4df7b320-b3c1-4fe5-9b99-9f1
8a08914bb); Time taken: 0.092 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=hive_20211014161734_4df7b320-b3c1-4fe5-9b99-9f18a08914bb)
: create table testlifecycle01(id int,name string) partitioned by (age int,dept string,scor
e int) tblproperties('lifecycle'='1')
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20211014161734_4df7b320-b3c1-4fe5-9b99-9f1
8a08914bb); Time taken: 0.257 seconds
INFO  : OK
```

(2) 查询表分区生命周期

针对已经创建好的表或者设置过 lifecycle 的表，需要查询 lifecycle 参数：

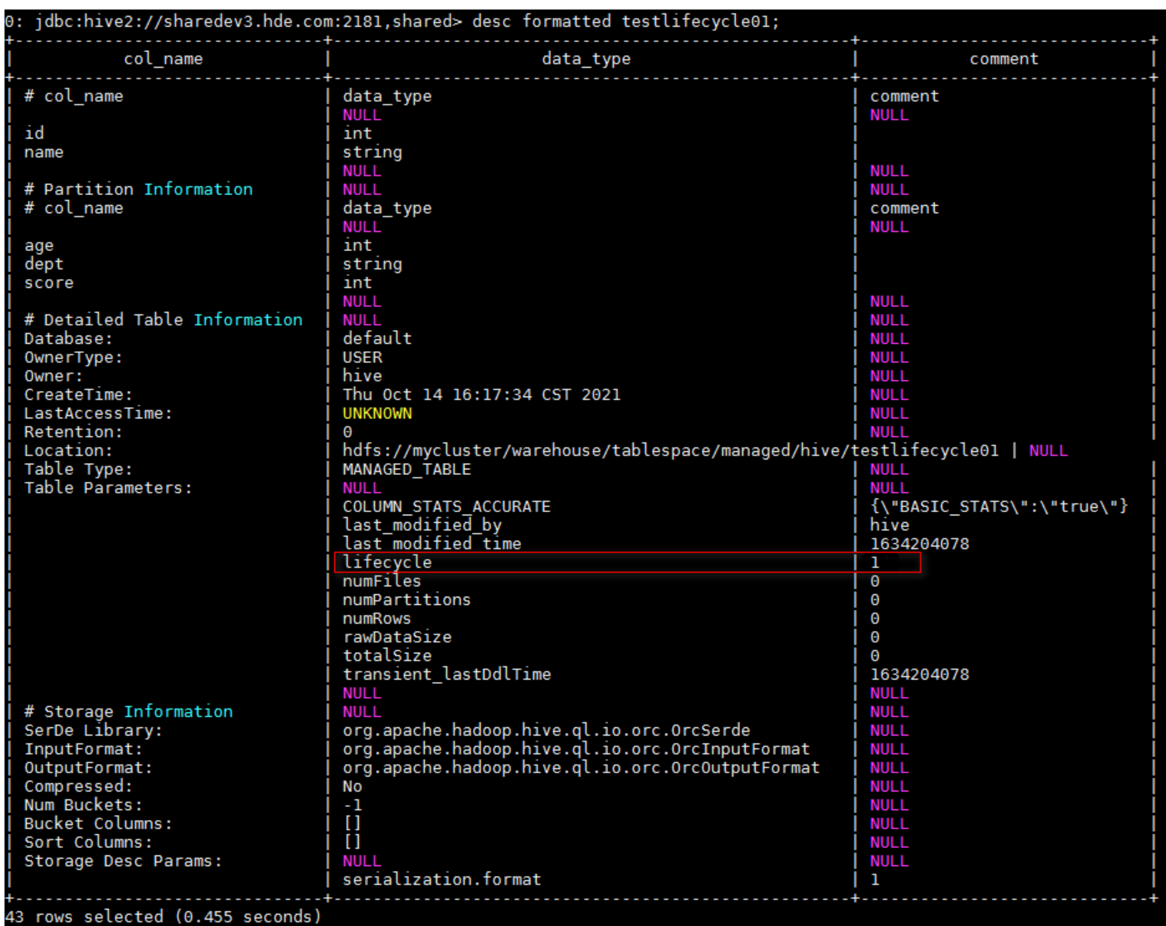
desc formatted <table_name>;

【操作示例】

a. 查询 testlifecycle01 分区表 lifecycle 参数值，如图 3-67 所示。执行命令如下：

desc formatted testlifecycle01;

图3-67 查询 lifecycle



0: jdbc:hive2://sharedev3.hde.com:2181,shared> desc formatted testlifecycle01;

col_name	data_type	comment
# col_name	data_type	comment
	NULL	NULL
id	int	
name	string	
	NULL	NULL
# Partition Information	NULL	NULL
# col_name	data_type	comment
	NULL	NULL
age	int	
dept	string	
score	int	
	NULL	NULL
# Detailed Table Information	NULL	NULL
Database:	default	NULL
OwnerType:	USER	NULL
Owner:	hive	NULL
CreateTime:	Thu Oct 14 16:17:34 CST 2021	NULL
LastAccessTime:	UNKNOWN	NULL
Retention:	0	NULL
Location:	hdfs://mycluster/warehouse/tablespace/managed/hive/testlifecycle01	NULL
Table Type:	MANAGED_TABLE	NULL
Table Parameters:	NULL	NULL
	COLUMN_STATS_ACCURATE	{\"BASIC_STATS\":\"true\"}
	last_modified_by	hive
	last_modified_time	1634204078
	lifecycle	1
	numFiles	0
	numPartitions	0
	numRows	0
	rawDataSize	0
	totalSize	0
	transient_lastDdlTime	1634204078
	NULL	NULL
# Storage Information	NULL	NULL
SerDe Library:	org.apache.hadoop.hive.ql.io.orc.OrcSerde	NULL
InputFormat:	org.apache.hadoop.hive.ql.io.orc.OrcInputFormat	NULL
OutputFormat:	org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat	NULL
Compressed:	No	NULL
Num Buckets:	-1	NULL
Bucket Columns:	[]	NULL
Sort Columns:	[]	NULL
Storage Desc Params:	NULL	NULL
	serialization.format	1

43 rows selected (0.455 seconds)

(3) 修改表分区生命周期

针对已经创建好的表或者设置过 lifecycle 表，需要增加或修改 lifecycle 参数：

ALTER TBALE <table_name> SET TBLPROPERTIES('lifecycle'='300');

【示例操作】

a. 修改 testlifecycle01 分区表 lifecycle 参数值，如图 3-68 所示。执行命令如下：

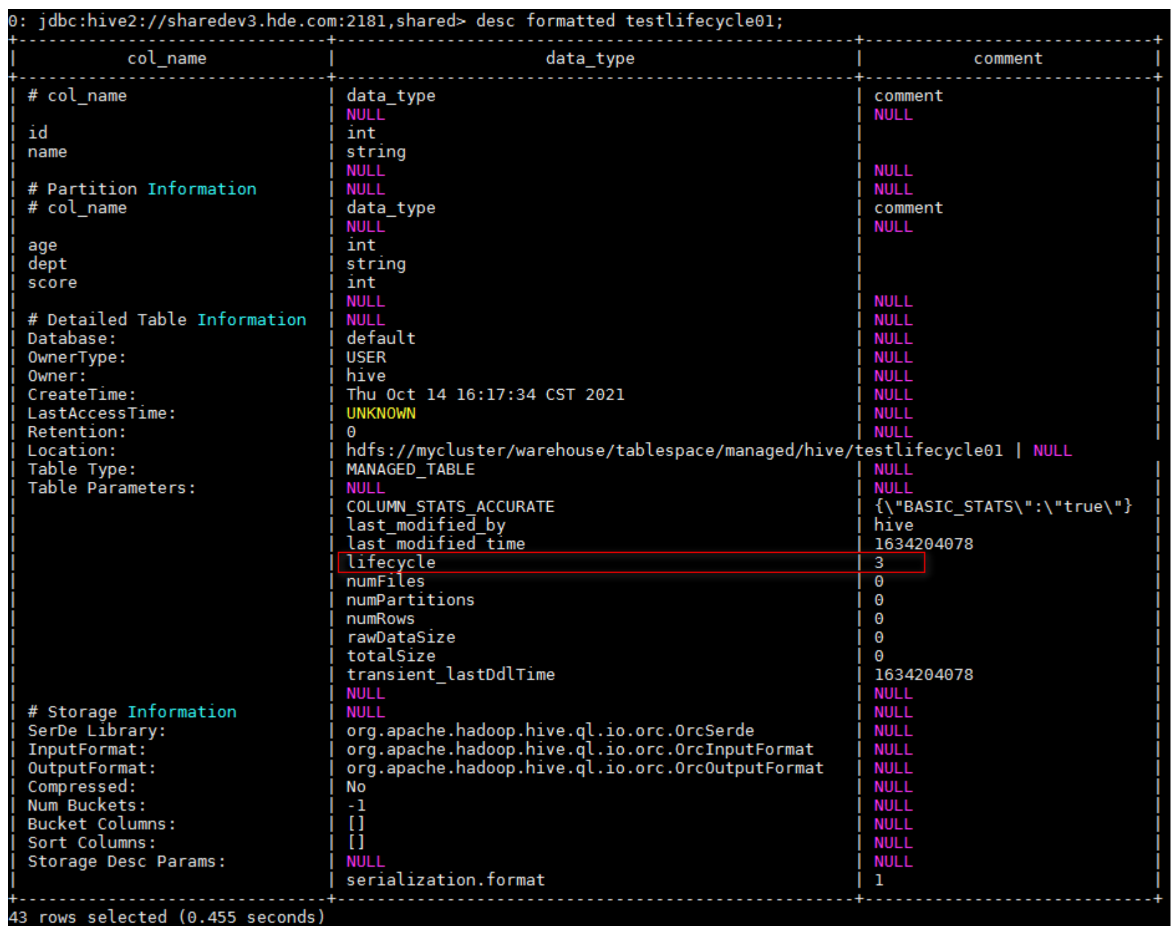
alter table testlifecycle01 set tblproperties('lifecycle'='3');

b. 修改 lifecycle 参数值后，执行查询结果如图 3-69 所示。

图3-68 修改 lifecycle

```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> alter table testlifecycle01 set tblproperties
('lifecycle'='3');
No rows affected (0.443 seconds)
INFO  : Compiling command(queryId=hive_20211014175743_e0bd85f7-5d12-4112-bc9c-070fa31cbb9b):
 alter table testlifecycle01 set tblproperties('lifecycle'='3')
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20211014175743_e0bd85f7-5d12-4112-bc9c-070f
a31cbb9b); Time taken: 0.042 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=hive_20211014175743_e0bd85f7-5d12-4112-bc9c-070fa31cbb9b):
 alter table testlifecycle01 set tblproperties('lifecycle'='3')
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20211014175743_e0bd85f7-5d12-4112-bc9c-070f
a31cbb9b); Time taken: 0.391 seconds
INFO  : OK
```

图3-69 修改后查询 lifecycle



```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> desc formatted testlifecycle01;
+-------------------------------+----------------------------------------------------+------------------------------+
|           col_name            |                     data_type                      |           comment            |
+-------------------------------+----------------------------------------------------+------------------------------+
| # col_name                    | data_type                                          | comment                      |
|                               | NULL                                               | NULL                         |
| id                            | int                                                |                              |
| name                          | string                                             |                              |
|                               | NULL                                               | NULL                         |
| # Partition Information       | NULL                                               | NULL                         |
| # col_name                    | data_type                                          | comment                      |
|                               | NULL                                               | NULL                         |
| age                           | int                                                |                              |
| dept                          | string                                             |                              |
| score                         | int                                                |                              |
|                               | NULL                                               | NULL                         |
| # Detailed Table Information  | NULL                                               | NULL                         |
| Database:                     | default                                            | NULL                         |
| OwnerType:                    | USER                                               | NULL                         |
| Owner:                        | hive                                               | NULL                         |
| CreateTime:                   | Thu Oct 14 16:17:34 CST 2021                       | NULL                         |
| LastAccessTime:               | UNKNOWN                                            | NULL                         |
| Retention:                    | 0                                                  | NULL                         |
| Location:                     | hdfs://mycluster/warehouse/tablespace/managed/hive/testlifecycle01 | NULL         |
| Table Type:                   | MANAGED_TABLE                                      | NULL                         |
| Table Parameters:             | NULL                                               | NULL                         |
|                               | COLUMN_STATS_ACCURATE                              | {\"BASIC_STATS\":\"true\"}   |
|                               | last_modified_by                                   | hive                         |
|                               | last_modified_time                                 | 1634204078                   |
|                               | lifecycle                                          | 3                            |
|                               | numFiles                                           | 0                            |
|                               | numPartitions                                      | 0                            |
|                               | numRows                                            | 0                            |
|                               | rawDataSize                                        | 0                            |
|                               | totalSize                                          | 0                            |
|                               | transient_lastDdlTime                              | 1634204078                   |
|                               | NULL                                               | NULL                         |
| # Storage Information         | NULL                                               | NULL                         |
| SerDe Library:                | org.apache.hadoop.hive.ql.io.orc.OrcSerde          | NULL                         |
| InputFormat:                  | org.apache.hadoop.hive.ql.io.orc.OrcInputFormat    | NULL                         |
| OutputFormat:                 | org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat   | NULL                         |
| Compressed:                   | No                                                 | NULL                         |
| Num Buckets:                  | -1                                                 | NULL                         |
| Bucket Columns:               | []                                                 | NULL                         |
| Sort Columns:                 | []                                                 | NULL                         |
| Storage Desc Params:          | NULL                                               | NULL                         |
|                               | serialization.format                               | 1                            |
+-------------------------------+----------------------------------------------------+------------------------------+
43 rows selected (0.455 seconds)
```

(4) 删除表分区生命周期

由于 Hive 中属性一旦设置不可删除，通过设置 lifecycle 为-1 可以忽略过期：

ALTER TBALE <table_name> SET TBLPROPERTIES('lifecycle'='-1');

【示例操作】

a. 设置 testlifecycle01 分区表 lifecycle 为-1，如图 3-70 所示。命令如下：

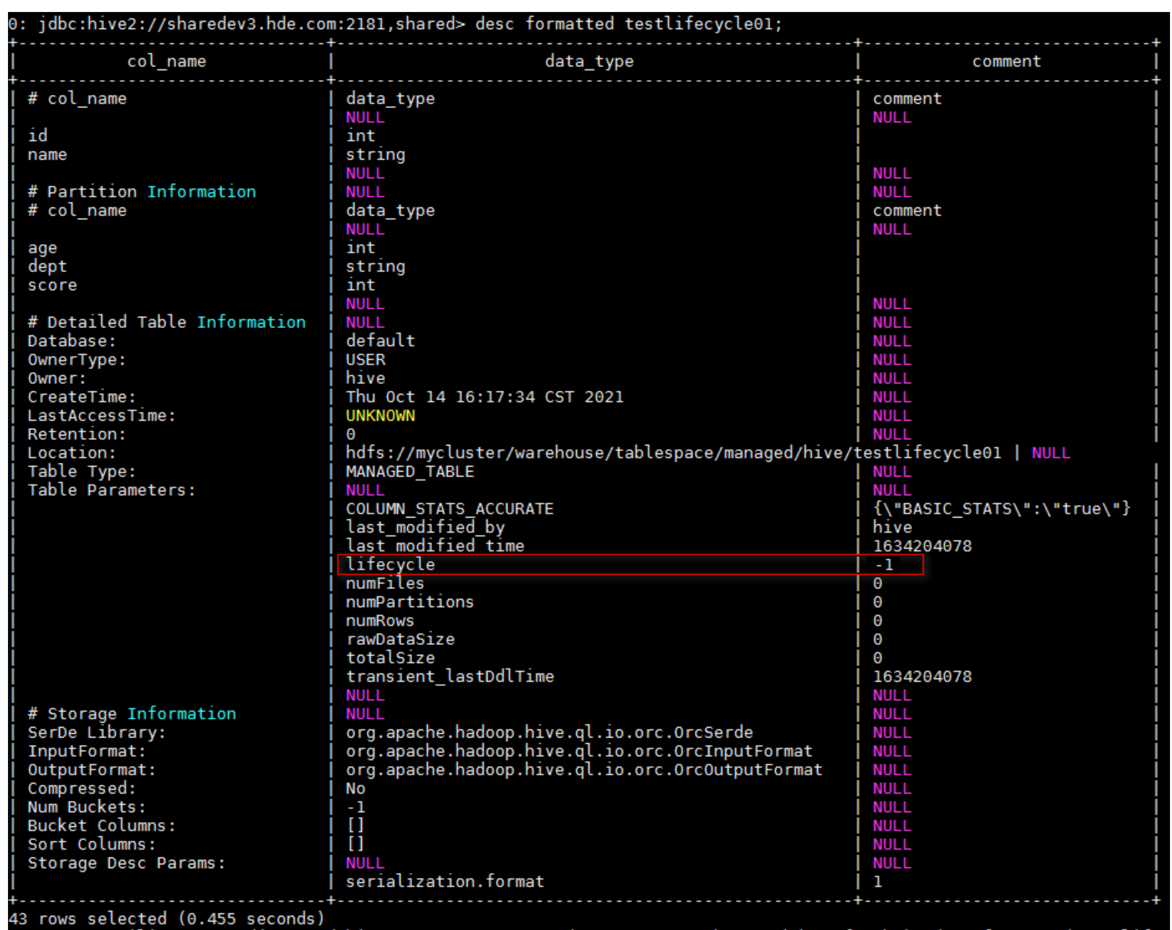
alter table testlifecycle01 set tblproperties('lifecycle'='-1');

b. 修改 lifecycle 参数值后，执行查询结果如图 3-71所示。

图3-70 修改 lifecycle

```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> alter table testlifecycle01 set tblproperties
('lifecycle'='-1');
No rows affected (0.767 seconds)
INFO  : Compiling command(queryId=hive_20211014180210_c6348330-6cd9-4c18-ac08-50bed3a094ae):
 alter table testlifecycle01 set tblproperties('lifecycle'='-1')
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20211014180210_c6348330-6cd9-4c18-ac08-50be
d3a094ae); Time taken: 0.257 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=hive_20211014180210_c6348330-6cd9-4c18-ac08-50bed3a094ae):
 alter table testlifecycle01 set tblproperties('lifecycle'='-1')
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20211014180210_c6348330-6cd9-4c18-ac08-50be
d3a094ae); Time taken: 0.496 seconds
INFO  : OK
```

图3-71 查询 lifecycle 参数



```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> desc formatted testlifecycle01;
+-------------------------------+----------------------------------------------------+-----------------------------+
|           col_name            |                     data_type                      |           comment           |
+-------------------------------+----------------------------------------------------+-----------------------------+
| # col_name                    | data_type                                          | comment                     |
|                               | NULL                                               | NULL                        |
| id                            | int                                                |                             |
| name                          | string                                             |                             |
|                               | NULL                                               | NULL                        |
| # Partition Information       | NULL                                               | NULL                        |
| # col_name                    | data_type                                          | comment                     |
|                               | NULL                                               | NULL                        |
| age                           | int                                                |                             |
| dept                          | string                                             |                             |
| score                         | int                                                |                             |
|                               | NULL                                               | NULL                        |
| # Detailed Table Information  | NULL                                               | NULL                        |
| Database:                     | default                                            | NULL                        |
| OwnerType:                    | USER                                               | NULL                        |
| Owner:                        | hive                                               | NULL                        |
| CreateTime:                   | Thu Oct 14 16:17:34 CST 2021                       | NULL                        |
| LastAccessTime:               | UNKNOWN                                            | NULL                        |
| Retention:                    | 0                                                  | NULL                        |
| Location:                     | hdfs://mycluster/warehouse/tablespace/managed/hive/testlifecycle01 | NULL        |
| Table Type:                   | MANAGED_TABLE                                      | NULL                        |
| Table Parameters:             | NULL                                               | NULL                        |
|                               | COLUMN_STATS_ACCURATE                              | {\"BASIC_STATS\":\"true\"}  |
|                               | last_modified_by                                   | hive                        |
|                               | last_modified_time                                 | 1634204078                  |
|                               | lifecycle                                          | -1                          |
|                               | numFiles                                           | 0                           |
|                               | numPartitions                                      | 0                           |
|                               | numRows                                            | 0                           |
|                               | rawDataSize                                        | 0                           |
|                               | totalSize                                          | 0                           |
|                               | transient_lastDdlTime                              | 1634204078                  |
|                               | NULL                                               | NULL                        |
| # Storage Information         | NULL                                               | NULL                        |
| SerDe Library:                | org.apache.hadoop.hive.ql.io.orc.OrcSerde          | NULL                        |
| InputFormat:                  | org.apache.hadoop.hive.ql.io.orc.OrcInputFormat    | NULL                        |
| OutputFormat:                 | org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat   | NULL                        |
| Compressed:                   | No                                                 | NULL                        |
| Num Buckets:                  | -1                                                 | NULL                        |
| Bucket Columns:               | []                                                 | NULL                        |
| Sort Columns:                 | []                                                 | NULL                        |
| Storage Desc Params:          | NULL                                               | NULL                        |
|                               | serialization.format                               | 1                           |
+-------------------------------+----------------------------------------------------+-----------------------------+
43 rows selected (0.455 seconds)
```

2. 分区清理规则

分区过期检查线程默认每小时（3600s）会执行一次，查找设置分区 lifecycle 表，并根据当前时间与分区数据最后一次更新时间进行比对，超过生命周期时间分区会被清理。

【示例操作】

(1) 创建分区表并设置生命周期，然后查询分区表

a. 创建分区表 testlifecycle01，并指定 lifecycle 属性值为 1，命令如下：

```
create table testlifecycleceshi(id int,name string) partitioned by (age int,dept string,score int) tblproperties('lifecycle'='1');
```

b. 查询表 testlifecycleceshi，如图 3-72 所示，命令如下：

```
select * from testlifecycleceshi;
```

图3-72 分区表创建后查询表

```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> select * from testlifecycleceshi;
+------------------------+------------------------+------------------------+------------------------+
--------------------------+
| testlifecycleceshi.id  | testlifecycleceshi.name  | testlifecycleceshi.age  | testlifecycleceshi.dept  |
 testlifecycleceshi.score  |
+------------------------+------------------------+------------------------+------------------------+
--------------------------+
+------------------------+------------------------+------------------------+------------------------+
--------------------------+
No rows selected (0.487 seconds)
```

(2) 在表中插入数据

a. 对 testlifecycle01 分区表执行插入数据操作，命令如下：

```
insert into table testlifecycleceshi partition(age='19',dept='IS',score='85') values ('95002','liyong1');
insert into table testlifecycleceshi partition(age='20',dept='IS',score='86') values ('95002','liyong2');
insert into table testlifecycleceshi partition(age='21',dept='IS',score='87') values ('95002','liyong3');
insert into table testlifecycleceshi partition(age='22',dept='IS',score='88') values ('95002','liyong4');
insert into table testlifecycleceshi partition(age='23',dept='IS',score='89') values ('95002','liyong5');
insert into table testlifecycleceshi partition(age='23',dept='IS',score='90') values ('95002','liyong6');
```

b. 数据插入成功后，执行查询表操作，如图 3-73 所示。

图3-73 插入数据后查询表

```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> select * from testlifecycleceshi;
+-------------------------+--------------------------+--------------------------+--------------------------+
--------------------------+
| testlifecycleceshi.id  | testlifecycleceshi.name  | testlifecycleceshi.age  | testlifecycleceshi.dept  |
 testlifecycleceshi.score  |
+-------------------------+--------------------------+--------------------------+--------------------------+
--------------------------+
| 95002                  | liyong1                  | 19                      | IS                       |
 85                        |
| 95002                  | liyong2                  | 20                      | IS                       |
 86                        |
| 95002                  | liyong3                  | 21                      | IS                       |
 87                        |
| 95002                  | liyong4                  | 22                      | IS                       |
 88                        |
| 95002                  | liyong5                  | 23                      | IS                       |
 89                        |
| 95002                  | liyong6                  | 23                      | IS                       |
 90                        |
| 95002                  | liyong6                  | 23                      | IS                       |
 90                        |
+-------------------------+--------------------------+--------------------------+--------------------------+
--------------------------+
7 rows selected (0.475 seconds)
```

(3) 到生命周期设置的时间后，超过生命周期时间分区会被清理，查询表结果如图 3-74 所示。

图3-74 到生命周期设置时间后查询表

```
0: jdbc:hive2://sharedev3.hde.com:2181,shared> select * from testlifecycleceshi;
+-------------------------+--------------------------+--------------------------+--------------------------+
--------------------------+
| testlifecycleceshi.id  | testlifecycleceshi.name  | testlifecycleceshi.age  | testlifecycleceshi.dept  |
 testlifecycleceshi.score  |
+-------------------------+--------------------------+--------------------------+--------------------------+
--------------------------+
+-------------------------+--------------------------+--------------------------+--------------------------+
--------------------------+
No rows selected (0.487 seconds)
```

3.10 HQL操作

注意

只有 orc 表支持 update 操作，普通表和内外部表不支持 update 操作。

Hive 提供类 SQL 的 Hive Query Language 语言来操作结构化数据。

3.10.1 创建表

1. 概述

语法说明：

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name

[(col_name data_type [column_constraint_specification] [COMMENT col_comment], ... [constraint_specification])]

[COMMENT table_comment]

[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]

```
 [CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets
BUCKETS]
 [SKEWED BY (col_name, col_name, ...)
   ON ((col_value, col_value, ...), (col_value, col_value, ...), ...)
   [STORED AS DIRECTORIES]
 [
  [ROW FORMAT row_format]
  [STORED AS file_format]
   | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]
 ]
 [LOCATION hdfs_path]
 [TBLPROPERTIES (property_name=property_value, ...)]
 [AS select_statement];
```

创建表主要有以下三种方式：

- 自定义表结构，以关键字 EXTERNAL 来区分创建内部表或外部表。
  - 内部表：如果对数据的处理都由 Hive 完成，则应该使用内部表。在删除内部表时，元数据和数据一起被删除。
  - 外部表：如果数据要被多种工具（如 Pig 等）共同处理，则应该使用外部表，可避免对该数据的误操作。删除外部表时，只删除掉元数据。
- 根据已有表来创建新表，使用 CREATE LIKE 句式，可以完全复制原有的表结构，包括表的存储格式。
- 根据查询结果创建新表，使用 CREATE AS SELECT 句式。

2. Hive 数据类型

Hive 中的数据类型包括基本数据类型和复杂数据类型。

表3-12 基本数据类型

数据类型	描述	示例
TINYINT	1byte有符号整数	100Y
SMALLINT	2byte有符号整数	100S
INT	4byte有符号整数	100
BIGINT	8byte有符号整数	100L
FLOAT	4byte单精度浮点数	3.14F
DOUBLE	8byte双精度浮点数	3.14D
DECIMAL	允许用户自定义规模和精度，比DOUBLE表示的范围大、更精确	3.14BD
TIMESTAMP	整数（距离UNIX新纪元时间的秒数）、浮点数（距离UNIX新纪元时间的秒数，精确到纳秒）或字符串（JDBC约定的时间字符串格式：YYYY--MM--DD hh:mm:ss:ffffffff）	-
DATE	日期，格式为YYYY--MM--DD，DATE类型只能和DATE、	-

数据类型	描述	示例
	TIMESTAMP、STRING相互转换	
STRING	字符串，可使用单引号或双引号	-
VARCHAR	指定长度的字符串	-
CHAR	指定长度，长度不够会以空格填充	-
BOOLEAN	布尔类型，true或false	-
BINARY	字节数组	-

表3-13 复杂数据类型

数据类型	描述	示例
STRUCT	类似于C中的struct，可用“点”符号访问元素	STRUCT<col_name : data_type [COMMENT col_comment], ...>
MAP	一组键值对集合，用map[key]访问元素	MAP<primitive_type, data_type>
ARRAY	数组，用编号访问，编号从0开始	ARRAY<data_type>
UNION	可以综合上面的数据类型组合到一起	UNIONTYPE<data_type, data_type, ...>

3. 创建表示例

(1) 内部表创建，执行如下命令：

```
create table employees(name string, salary float, address string) row format delimited fields terminated by ',' stored as textfile;
```

其中：

- "delimited fields terminated by"用于指定列与列之间的分隔符
- “stored as textfile”用于指定表的存储格式为 textfile

(2) 外部表创建，执行如下命令：

```
create external table stocks(sysbol string, price_open float, price_colse float ) ;
```

(3) 使用 CREATE Like 创建表，执行如下命令：

```
create table employees_like LIKE employees;
```

4. 扩展

- 创建分区表

一个表可以拥有一个或者多个分区，每个分区以文件夹的形式单独存在表文件夹的目录下。对分区内数据进行查询，可缩小查询范围，加快数据的检索速度和可对数据按照一定的条件进行管理。

分区可在创建表时用 PARTITIONED BY 子句进行定义。

```
CREATE EXTERNAL TABLE IF NOT EXISTS employees_info_extended
 (
 id INT,
```

```
name STRING,
usd_flag STRING,
salary DOUBLE,
deductions MAP<STRING, DOUBLE>,
address STRING
) PARTITIONED BY (entrytime STRING)
STORED AS TEXTFILE;
```

- 更新表的结构

  一个表在创建完成后，还可以使用 ALTER TABLE 执行增/删字段、修改表属性以及添加分区等操作。例如为表 employees_info_extended 增加 tel_phone 和 email 字段。执行如下命令：

```
ALTER TABLE employees_info_extended ADD COLUMNS (tel_phone STRING, email STRING);
```

3.10.2 数据加载

1. 概述

使用 HQL 可以向已有的表中加载数据。数据加载中可以从本地文件系统、HDFS 集群中加载数据，以关键字 LOCAL 来区分数据源是否来自本地。

加载数据语句中有关键字 LOCAL，表明从本地加载数据，除要求对相应表的 UPDATE 权限外，还要求该数据在当前连接的 HiveServer 节点上。

说明

- LOCAL 模式和 HDFS 模式都需要用户拥有文件的所有者权限。
- 如果加载数据语句中有关键字 OVERWRITE，表示加载的数据会覆盖表中原有的数据，否则加载的数据会追加到表中。

语法说明：

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]
```

【注意】Hive 使用 LOAD DATA LOCAL INPATH 时，要求数据文件必须与所连接的 HiveServer2 服务在同一个节点。

2. 数据加载示例

数据加载示例操作步骤如下：

(1) 在 HDFS 上创建文件/hivetest/employee.txt 文件

(2) 将/hivetest/employee.txt 文件加载至 employees 表中，执行如下命令：

```
LOAD DATA INPATH '/hivetest/employee.txt ' OVERWRITE INTO TABLE employees;
```

(3) 查询 employees 中的数据，执行如下命令：

```
Select * from employees;
```

图3-75 执行结果

```
9: jdbc:hive2://101.8.134.10:2181/default> select * from employees;
+-----------------+-------------------+---------------------+--+
| employees.name  | employees.salary  | employees.address   |
+-----------------+-------------------+---------------------+--+
| xiaoming        | 5000.57           | zhengzhou           |
| lihua=          | 8000.79           | zhengzhou           |
+-----------------+-------------------+---------------------+--+
```

3.10.3 数据查询

1. 概述

使用 HQL 可以对数据进行查询分析。查询分析方法有：

- SELECT 查询的常用特性，如 JOIN 等。
- 加载数据至指定分区。
- 使用 Hive 自带函数。
- 使用自定义函数。

说明

在集群中执行如下操作，对操作的表需要拥有对应的操作权限。

语法说明：

```
[WITH CommonTableExpression (, CommonTableExpression)*]
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
 FROM table_reference
 [WHERE where_condition]
 [GROUP BY col_list]
 [ORDER BY col_list]
 [CLUSTER BY col_list
   | [DISTRIBUTE BY col_list] [SORT BY col_list]
 ]
[LIMIT [offset,] rows]
```

2. 数据查询示例

(1) 查询 name 列的数据，执行如下语句：

```
SELECT name from employees;
```

(2) 使用 Hive 自带函数 COUNT()，统计表 employees 中有多少条记录，执行如下语句：

```
SELECT COUNT(*) FROM employees_info_extended;
```

3.10.4 视图操作

说明

Hive 0.6 及以上版本支持视图。

1. 概述

Hive View 具有以下特点：

- View 是逻辑存在。
- View 只读，且不支持 LOAD/INSERT/ALTER。需要改变 View 定义时，可以使用 Alter View。
- View 内可能包含 OPDER BY/LIMIT 语句，若一个对 View 的查询包含这些语句，则该语句优先级高。
- Hive 支持迭代视图。
- Hive 中的视图查询和普通查询类似，查询时把表名更换为视图名即可。
- 语法说明：

```
CREATE VIEW [IF NOT EXISTS] [db_name.]view_name [(column_name [COMMENT
column_comment], ...) ]
 [COMMENT view_comment]
 [TBLPROPERTIES (property_name = property_value, ...)]
 AS SELECT ...;
```

2. 视图操作示例

(1) 创建视图，执行如下命令：

```
create view employees_name_info as select name from employees;
```

(2) 查询视图，执行如下命令：

```
select * from employees_name_info;
```

(3) 删除视图，执行如下命令：

```
drop view employees_name_info;
```

3.10.5 函数介绍

说明

Hive 提供了丰富的函数操作，详情请参考 Hive 官网 https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF。

- 使用 ***Show functions*** 命令可以列举出当前 Hive 会话中所加载的函数名称，包括内置的和用户加载进来的函数。
- 系统函数通常都有其自身的使用文档，使用 ***describe function*** 命令可以查看对应函数的简短介绍。

- 执行命令：describe function abs;
- 执行结果：abs(x) - returns the absolute value of x

- 关于函数更多的详细文档，可以在 ***describe function*** 命令中使用 ***extended*** 关键字来查看。
  - 执行命令：describe function extended abs;
  - 执行结果如图 3-76 所示。

图3-76 abs 函数详细文档

```
0: jdbc:hive2://node2:10000> describe function extended abs;
+-------------------------------------------+--+
|                  tab_name                 |
+-------------------------------------------+--+
| abs(x) - returns the absolute value of x  |
| Example:                                  |
|   > SELECT abs(0) FROM src LIMIT 1;       |
|   0                                       |
|   > SELECT abs(-5) FROM src LIMIT 1;      |
|   5                                       |
+-------------------------------------------+--+
6 rows selected (0.025 seconds)
```

下面介绍 Hive 内置的部分系统函数。

表3-14 数学函数

Return Type	Name (Signature)	Description
DOUBLE	round(DOUBLE a)	返回对a四舍五入的BIGINT值
DOUBLE	round(DOUBLE a, INT d)	返回对a保留d位小数四舍五入的近似值，
DOUBLE	bround(DOUBLE a)	四舍五入法（1~4：舍，6~9：进，5->前位数是偶：舍，5->前位数是奇：进）
DOUBLE	bround(DOUBLE a, INT d)	四舍五入法，保留d位小数
BIGINT	floor(DOUBLE a)	向下取整，数轴上最接近a的左边整数值 如：6.10->6， -3.4->-4
BIGINT	ceil(DOUBLE a), ceiling(DOUBLE a)	求不小于给定实数的最小整数，如：ceil(6)=6，ceil(6.1)= ceil(6.9) = 7
DOUBLE	rand(), rand(INT seed)	返回一个DOUBLE型随机数 其中：seed是生成随机数的随机种子
DOUBLE	exp(DOUBLE a), exp(DECIMAL a)	返回e的a幂次方，a可为小数
DOUBLE	ln(DOUBLE a), ln(DECIMAL a)	以自然数为底的对数，a可为小数
DOUBLE	log10(DOUBLE a), log10(DECIMAL a)	以10为底的对数，a可为小数
DOUBLE	log2(DOUBLE a), log2(DECIMAL a)	以2为底数的对数，a可为小数
DOUBLE	log(DOUBLE base, DOUBLE a) log(DECIMAL base, DECIMAL a)	以base为底a的对数，base 与 a都是DOUBLE类型
DOUBLE	pow(DOUBLE a, DOUBLE p), power(DOUBLE a, DOUBLE p)	计算a的p次幂
DOUBLE	sqrt(DOUBLE a), sqrt(DECIMAL a)	计算a的平方根

Return Type	Name (Signature)	Description
STRING	bin(BIGINT a)	计算a的二进制，结果转为string类型
STRING	hex(BIGINT a) hex(STRING a) hex(BINARY a)	计算a的16进制，结果转为STRING类型，
BINARY	unhex(STRING a)	hex的逆方法
STRING	conv(BIGINT num, INT from_base, INT to_base), conv(STRING num, INT from_base, INT to_base)	将GIGINT/STRING类型的num从from_base进制转换成to_base进制
DOUBLE	abs(DOUBLE a)	计算a的绝对值
INT or DOUBLE	pmod(INT a, INT b), pmod(DOUBLE a, DOUBLE b)	a对b取模
DOUBLE	sin(DOUBLE a), sin(DECIMAL a)	求a的正弦值
DOUBLE	asin(DOUBLE a), asin(DECIMAL a)	求a的反正弦值
DOUBLE	cos(DOUBLE a), cos(DECIMAL a)	求余弦值
DOUBLE	acos(DOUBLE a), acos(DECIMAL a)	求反余弦值
DOUBLE	tan(DOUBLE a), tan(DECIMAL a)	求正切值
DOUBLE	atan(DOUBLE a), atan(DECIMAL a)	求反正切值
DOUBLE	degrees(DOUBLE a), degrees(DECIMAL a)	将弧度值转换成角度值
DOUBLE	radians(DOUBLE a), radians(DOUBLE a)	将角度值转换成弧度值
INT or DOUBLE	positive(INT a), positive(DOUBLE a)	返回a
INT or DOUBLE	negative(INT a), negative(DOUBLE a)	返回a的相反数
DOUBLE or INT	sign(DOUBLE a), sign(DECIMAL a)	如果a是正数则返回1.0，是负数则返回-1.0，否则返回0.0
DOUBLE	e()	数学常数e
DOUBLE	pi()	数学常数pi
BIGINT	factorial(INT a)	求a的阶乘
DOUBLE	cbrt(DOUBLE a)	求a的立方根
INT BIGINT	shiftleft(TINYINT|SMALLINT|INT a, INT b) shiftleft(BIGINT a, INT b)	按位左移
INT BIGINT	shiftright(TINYINT|SMALLINT|INT a, INTb) shiftright(BIGINT a, INT b)	按拉右移
INT BIGINT	shiftrightunsigned(TINYINT|SMALLINT|INTa , INT b), shiftrightunsigned(BIGINT a, INT b)	无符号按位右移
T	greatest(T v1, T v2, ...)	求最大值
T	least(T v1, T v2, ...)	求最小值

表3-15 集合函数

Return Type	Name(Signature)	Description
int	size(Map<K.V>)	求map的长度
int	size(Array<T>)	求数组的长度
array<K>	map_keys(Map<K.V>)	返回map中的所有key
array<V>	map_values(Map<K.V>)	返回map中的所有value
boolean	array_contains(Array<T>, value)	如该数组Array<T>包含value返回true，否则返回false
array	sort_array(Array<T>)	按自然顺序对数组进行排序并返回

表3-16 类型转换函数

Return Type	Name(Signature)	Description
binary	binary(string|binary)	Casts the parameter into a binary 将输入的值转换成二进制
Expected "=" to follow "type"	cast(expr as <type>)	将expr转换成type类型 如：cast("1" as BIGINT) 将字符串1转换成了BIGINT类型，如果转换失败将返回NULL

表3-17 日期函数

Return Type	Name(Signature)	Description
string	from_unixtime(bigint unixtime[, string format])	将时间的秒值转换成format格式（format可为“yyyy-MM-dd hh:mm:ss”,“yyyy-MM-dd hh”,“yyyy-MM-dd hh:mm”等等），如 from_unixtime(1250111000,"yyyy-MM-dd ") 得到2009-03-12
bigint	unix_timestamp()	获取本地时区下的时间戳
bigint	unix_timestamp(string date)	将格式为yyyy-MM-dd HH:mm:ss的时间字符串转换成时间戳，如 unix_timestamp('2009-03-20 11:30:01') = 1237573801
bigint	unix_timestamp(string date, string pattern)	将指定时间字符串格式字符串转换成Unix时间戳，如果格式不对返回0，如：unix_timestamp('2009-03-20', 'yyyy-MM-dd') = 1237532400
string	to_date(string timestamp)	返回时间字符串的日期部分
int	year(string date)	返回时间字符串的年份部分
int	quarter(date/timestamp/string)	返回当前时间属于哪个季度，如 quarter('2015-04-08') = 2
int	month(string date)	返回时间字符串的月份部分

Return Type	Name(Signature)	Description
int	day(string date) dayofmonth(date)	返回时间字符串的天
int	hour(string date)	返回时间字符串的小时
int	minute(string date)	返回时间字符串的分钟
int	second(string date)	返回时间字符串的秒
int	weekofyear(string date)	返回时间字符串位于一年中的第几个周内 如weekofyear("1970-11-01 00:00:00") = 44, weekofyear("1970-11-01") = 44
int	datediff(string enddate, string startdate)	计算开始时间startdate到结束时间enddate相差的天数
string	date_add(string startdate, int days)	从开始时间startdate加上days
string	date_sub(string startdate, int days)	从开始时间startdate减去days
timestamp	from_utc_timestamp(timestamp, string timezone)	如果给定的时间戳并非UTC，则将其转化成指定的时区下时间戳
timestamp	to_utc_timestamp(timestamp, string timezone)	如果给定的时间戳是指定时区下时间戳，则将其转化成UTC下的时间戳
date	current_date	返回当前时间日期
timestamp	current_timestamp	返回当前时间戳
string	add_months(string start_date, int num_months)	返回当前时间下再增加num_months个月的日期
string	last_day(string date)	返回这个月的最后一天的日期，忽略时分秒部分（HH:mm:ss）
string	next_day(string start_date, string day_of_week)	返回当前时间的下一个星期X所对应的日期 如：next_day('2015-01-14', 'TU') = 2015-01-20 以2015-01-14为开始时间，其下一个星期二所对应的日期为2015-01-20
string	trunc(string date, string format)	返回时间的最开始年份或月份，如trunc("2016-06-26",“MM”)=2016-06-01 trunc("2016-06-26",“YY”)=2016-01-01，注意所支持的格式为MONTH/MON/MM, YEAR/YYYY/YY
double	months_between(date1, date2)	返回date1与date2之间相差的月份，如date1>date2，则返回正，如果date1<date2，则返回负，否则返回0.0，如：months_between('1997-02-28 10:30:00', '1996-10-30') = 3.94959677 1997-02-28 10:30:00与1996-10-30相差3.94959677个月
string	date_format(date/timestamp/string ts, string fmt)	按指定格式返回时间date 如：date_format("2016-06-22","MM-dd")=06-22

表3-18 条件函数

Return Type	Name(Signature)	Description
T	if(boolean testCondition, T valueTrue, T valueFalseOrNull)	如果testCondition 为true就返回valueTrue,否则返回valueFalseOrNull，（valueTrue，valueFalseOrNull为泛型）
T	nvl(T value, T default_value)	如果value值为NULL就返回default_value，否则返回value
T	COALESCE(T v1, T v2, ...)	返回第一非null的值，如果全部都为NULL就返回NULL 如：COALESCE (NULL,44,55)=44
T	CASE a WHEN b THEN c [WHEN d THEN e]* [ELSE f] END	如果a=b就返回c，a=d就返回e，否则返回f 如CASE 4 WHEN 5 THEN 5 WHEN 4 THEN 4 ELSE 3 END 将返回4
T	CASE WHEN a THEN b [WHEN c THEN d]* [ELSE e] END	如果a=ture就返回b，c=ture就返回d，否则返回e 如：CASE WHEN 5>0 THEN 5 WHEN 4>0 THEN 4 ELSE 0 END 将返回5；CASE WHEN 5<0 THEN 5 WHEN 4<0 THEN 4 ELSE 0 END 将返回0
boolean	isnull(a)	如果a为null就返回true，否则返回false
boolean	isnotnull (a)	如果a为非null就返回true，否则返回false

表3-19 字符函数

Return Type	Name(Signature)	Description
int	ascii(string str)	返回str中首个ASCII字符串的整数值
string	base64(binary bin)	将二进制bin转换成64位的字符串
string	concat(string|binary A, string|binary B...)	对二进制字节码或字符串按次序进行拼接
array<struct<string,double>>	context_ngrams(array<array<string>>, array<string>, int K, int pf)	与ngram类似，但context_ngram()允许你预算指定上下文(数组)来去查找子序列
string	concat_ws(string SEP, string A, string B...)	与concat()类似，但使用指定的分隔符进行分隔
string	concat_ws(string SEP, array<string>)	拼接Array中的元素并用指定分隔符进行分隔
string	decode(binary bin, string charset)	使用指定的字符集charset将二进制值bin解码成字符串，支持的字符集有：'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'，如果任意输入参数为NULL都将返回NULL
binary	encode(string src, string charset)	使用指定的字符集charset将字符串编码成二进制值，支持的字符集有：'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'，如果任一输入参数为NULL都将返回NULL
int	find_in_set(string str, string strList)	返回以逗号分隔的字符串中str出现的位

Return Type	Name(Signature)	Description
		置，如果参数str为逗号或查找失败将返回0，如果任一参数为NULL将返回NULL回
string	format_number(number x, int d)	将数值X转换成"#,###,###.##"格式字符串，并保留d位小数，如果d为0，将进行四舍五入且不保留小数
string	get_json_object(string json_string, string path)	从指定路径上的JSON字符串抽取出JSON对象，并返回这个对象的JSON格式，如果输入的JSON是非法的将返回NULL。 **注意**：此路径上JSON字符串只能由数字、字母、下划线组成且不能有大写字母和特殊字符，且key不能由数字开头，这是由于Hive对列名的限制
boolean	in_file(string str, string filename)	如果文件名为filename的文件中有一行数据与字符串str匹配成功就返回true
int	instr(string str, string substr)	查找字符串str中子字符串substr出现的位置，如果查找失败将返回0，如果任一参数为Null将返回null **注意**：位置为从1开始
int	length(string A)	返回字符串的长度
int	locate(string substr, string str[, int pos])	查找字符串str中的pos位置后字符串substr第一次出现的位置
string	lower(string A) lcase(string A)	将字符串A的所有字母转换成小写字母
string	lpad(string str, int len, string pad)	从左边开始对字符串str使用字符串pad填充，最终len长度为止，如果字符串str本身长度比len大的话，将去掉多余的部分
string	ltrim(string A)	去掉字符串A前面的空格
array<struct<string,double>>	ngrams(array<array<string>>, int N, int K, int pf)	返回出现次数TOP K的的子序列，n表示子序列的长度
string	parse_url(string urlString, string partToExtract [, string keyToExtract])	返回从URL中抽取指定部分的内容，参数url是URL字符串，而参数partToExtract是要抽取的部分，这个参数包含(HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO,例如：parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST') ='facebook.com'，如果参数partToExtract值为QUERY则必须指定第三个参数key如：parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1') =‘v1’
string	printf(String format, Obj... args)	按照printf风格格式输出字符串
string	regexp_extract(string subject, string pattern, int index)	抽取字符串subject中符合正则表达式pattern的第index个部分的子字符串。 注意预定义字符的使用，如第二个参数如

Return Type	Name(Signature)	Description
		果使用'\s'将被匹配到s，'\\s'才是匹配空格
string	regexp_replace(string INITIAL_STRING, string PATTERN, string REPLACEMENT)	按照正则表达式PATTERN将字符串INTIAL_STRING中符合条件的部分成REPLACEMENT所指定的字符串，如里REPLACEMENT这空的话，抽符合正则的部分将被去掉，如：regexp_replace("foobar", "oo|ar", "") = 'fb.' 注意预定义字符的使用，如第二个参数如果使用'\s'将被匹配到s,'\\s'才是匹配空格
string	repeat(string str, int n)	重复输出n次字符串str
string	reverse(string A)	反转字符串
string	rpad(string str, int len, string pad)	从右边开始对字符串str使用字符串pad填充，最终len长度为止，如果字符串str本身长度比len大的话，将去掉多余的部分
string	rtrim(string A)	去掉字符串后面出现的空格
array<array<string>>	sentences(string str, string lang, string locale)	字符串str将被转换成单词数组，如：sentences('Hello there! How are you?') =(("Hello", "there"), ("How", "are", "you"))
string	space(int n)	返回n个空格
array	split(string str, string pat)	按照正则表达式pat来分割字符串str，并将分割后的数组字符串的形式返回
map<string,string>	str_to_map(text[, delimiter1, delimiter2])	将字符串str按照指定分隔符转换成Map，第一个参数是需要转换字符串，第二个参数是键值对之间的分隔符，默认为逗号；第三个参数是键值之间的分隔符，默认为"="
string	substr(string|binary A, int start) substring(string|binary A, int start)	对于字符串A，从start位置开始截取字符串并返回
string	substr(string|binary A, int start, int len) substring(string|binary A, int start, int len)	对于二进制/字符串A,从start位置开始截取长度为length的字符串并返回
string	substring_index(string A, string delim, int count)	截取第count分隔符之前的字符串，如count为正则从左边开始截取，如果为负则从右边开始截取
string	translate(string|char|varchar input, string|char|varchar from, string|char|varchar to)	将input出现在from中的字符串替换成to中的字符串 如：translate("MOBIN","BIN","M")="MOM"
string	trim(string A)	将字符串A前后出现的空格去掉
binary	unbase64(string str)	将64位的字符串转换二进制值
string	upper(string A) ucase(string A)	将字符串A中的字母转换成大写字母
string	initcap(string A)	将字符串A的每个单词转换为首字母大写的字符串
int	levenshtein(string A, string B)	计算两个字符串之间的差异大小，如：levenshtein('kitten', 'sitting') = 3

Return Type	Name(Signature)	Description
string	soundex(string A)	将普通字符串转换成soundex字符串

表3-20 聚合函数

Return Type	Name(Signature)	Description
BIGINT	count(*), count(expr), count(DISTINCT expr[, expr...])	统计总行数，包括含有NULL值的行 统计提供非NULL的expr表达式值的行数 统计提供非NULL且去重后的expr表达式值的行数
DOUBLE	sum(col), sum(DISTINCT col)	sum(col),表示求指定列的和，sum(DISTINCT col)表示求去重后的列和
DOUBLE	avg(col), avg(DISTINCT col)	avg(col)表示求指定列的平均值，avg(DISTINCT col)表示求去重后的列的平均值
DOUBLE	min(col)	求指定列的最小值
DOUBLE	max(col)	求指定列的最大值
DOUBLE	variance(col), var_pop(col)	求指定列数值的方差
DOUBLE	var_samp(col)	求指定列数值的样本方差
DOUBLE	stddev_pop(col)	求指定列数值的标准偏差
DOUBLE	stddev_samp(col)	求指定列数值的样本标准偏差
DOUBLE	covar_pop(col1, col2)	求指定列数值的协方差
DOUBLE	covar_samp(col1, col2)	求指定列数值的样本协方差
DOUBLE	corr(col1, col2)	返回两列数值的相关系数
DOUBLE	percentile(BIGINT col, p)	返回col的p%分位数

表3-21 表生成函数

Return Type	Name(Signature)	Description
Array Type	explode(array<TYPE> a)	对于a中的每个元素，将生成一行且包含该元素
N rows	explode(ARRAY)	每行对应数组中的一个元素
N rows	explode(MAP)	每行对应每个map键-值，其中一个字段是map的键，另一个字段是map的值
N rows	posexplode(ARRAY)	与explode类似，不同的是还返回各元素在数组中的位置
N rows	stack(INT n, v_1, v_2, ..., v_k)	把M列转换成N行，每行有M/N个字段，其中n必须是个常数
tuple	json_tuple(jsonStr, k1, k2, ...)	从一个JSON字符串中获取多个键并作为一个元组返回，与get_json_object不同的

Return Type	Name(Signature)	Description
		是此函数能一次获取多个键值
tuple	parse_url_tuple(url, p1, p2, ...)	返回从URL中抽取指定N部分的内容，参数url是URL字符串，而参数p1,p2,是要抽取的部分，这个参数包含HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, USERINFO, QUERY:<KEY>
-	inline(ARRAY<STRUCT[,STRUCT]>)	将结构体数组提取出来并插入到表中

3.11 JDBC方式连接HiveServer2

除了使用 beeline 命令提交 HQL 语句外，还可以使用 JDBC 方式连接 HiveServer2。

通过 JDBC 方式连接 HiveServer2，并对表进行创建和查询操作。操作步骤如下：

(1) 建立 maven 工程

Pom.xml 文件内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.test.hive</groupId>
    <artifactId>hiveDemo</artifactId>
    <version>1.0</version>
    <dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-jdbc</artifactId>
            <version>2.1.1-cdh6.2.0</version>
        </dependency>
</dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <archive>
                        <manifest>
                            <mainClass>HIVE_JDBC.Hive_JDBC</mainClass>
                        </manifest>
```

```
                </archive>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
</project>
```

(2) 准备建立连接参数

- 非 Kerberos 环境

```
private static String driveName = "org.apache.hive.jdbc.HiveDriver" ;
//开启 kerberos 的集群环境，请根据实际环境修改配置
private static String url = "jdbc:hive2://192.168.0.1:10000/default";
private static String user = "hive";
private static String passwd = null;
private static String sql = "";
private static ResultSet res;
private static ResultSetMetaData m;
```

- Kerberos 环境

```
private static String driveName = "org.apache.hive.jdbc.HiveDriver" ;
//开启 kerberos 的集群环境，请根据实际环境修改配置
private static String url =
"jdbc:hive2://192.168.0.1:10000/default;principal=hive/node1.hde.com@TESTSHARE.COM
";
private static String user = "user1";
private static String passwd = null;
private static String sql = "";
private static ResultSet res;
private static ResultSetMetaData m;
```

说明

- 参数（user 和 url 地址）需要根据具体环境设计，url 参数需要填写 hiveServer2 的连接地址
- Kerberos 环境下的 principal 配置值请参考 2.3.2 Kerberos 环境。

(3) 建立连接

- 非 Kerberos 环境

```
private static Connection getConnection() throws ClassNotFoundException,SQLException{
    Class.forName(driveName);
    Connection con = DriverManager.getConnection(url,user,passwd);
    System.out.println("connection success!");
```

```
        return con;
    }
```

- Kerberos 环境

```
private static Connection getConnection() throws ClassNotFoundException,SQLException{
    String krb5ConfPath = "./krb5.conf";
    String keytabPath = "./user1.keytab";
    String principalName = "user1@TESTSHARE.COM ";
    //Kerberos 身份认证，请将集群中 krb5.conf 及 keytab 文件放入该工程 resource 文件夹中
    org.apache.hadoop.conf.Configuration conf = new
org.apache.hadoop.conf.Configuration();
    conf.set("hadoop.security.authentication", "Kerberos");

    //获取 krb5.conf 配置
    if(System.getProperty("os.name").toLowerCase().startsWith("win")) {
        System.setProperty("java.security.krb5.conf",
Hive_JDBC.class.getClassLoader().getResource(krb5ConfPath).getPath());
    }

    //进行身份认证
    try{
        UserGroupInformation.setConfiguration(conf);
        UserGroupInformation.loginUserFromKeytab(principalName,
Hive_JDBC.class.getClassLoader().getResource(keytabPath).getPath());
    }catch(IOException e1){
        e1.printStackTrace();
    }

    Class.forName(driveName);
    Connection con = DriverManager.getConnection(url,user,passwd);
    System.out.println("connection success!");
    return con;
}
```

说明

参数 krb5ConfPath、principalName 和 keytabPath 的值需要用户根据具体环境进行相应调整。

(4) 对表进行操作

```
private static void dropTable(Statement stm, String tableName)throws SQLException{
    sql = "drop table if exists "+tableName;
    System.out.println("Running:"+sql);
    stm.executeUpdate(sql);
}

private static void createTable(Statement stm, String tableName)throws SQLException{
    sql = "create table if not exists "+tableName+" (stuid string, name string, sex
string, age int)"
            +" row format delimited fields terminated by '\t'"
            + " lines terminated by '\n' "
            + " stored as textfile";
    System.out.println("Running:"+sql);
    stm.executeUpdate(sql);
}
```

```
private static void showTables(Statement stm, String tableName)throws SQLException{
    sql = "show tables in default";
    System.out.println("Running:"+sql);
    res = stm.executeQuery(sql);
    System.out.println("执行 show tables 的运行结果如下：");
    m=res.getMetaData();
    int columns=m.getColumnCount();
    while(res.next())
    {
        for(int i=1;i<=columns;i++)
        {
            System.out.print(res.getString(i)+"\t\t");
        }
        System.out.println();
    }
}

private static void describeTale(Statement stm, String tableName)throws SQLException{
        sql = "describe " + tableName;
        System.out.println("Running:"+sql);
        res = stm.executeQuery(sql);
        System.out.println("执行 describe table 的运行结果如下：");
        while (res.next()) {
            System.out.println(res.getString(1) + "\t" + res.getString(2));
        }
    }

private static void selectData(Statement stm, String tableName)throws SQLException{
        sql = "select * from "+tableName;
        System.out.println("Running:"+sql);
        res = stm.executeQuery(sql);
        System.out.println("执行 select *  的运行结果如下：");
        while (res.next()) {
            System.out.println(res.getString(1) + "\t" + res.getString(2)+ "\t" +
res.getString(3)+ "\t" + res.getString(4));
        }
}
```

(5) 主类操作

```
import org.apache.hadoop.security.UserGroupInformation;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public static void main(String[] args){
    Connection con = null;
    Statement stm = null;
```

```
        try{
            con = getConnection();//创建连接
            stm = con.createStatement();

            String tableName = "stu2";
            dropTable(stm,tableName);//若表存在则先删除表
            createTable(stm,tableName);//若表不存在则新建
            showTables(stm, tableName);//查看当前数据库下的所有表
            describeTale(stm, tableName);//查看创建的表的表结构
            selectData(stm, tableName);//全表查询
        }catch (ClassNotFoundException e){
            e.printStackTrace();
            System.out.println(driveName + " not found! ");
            System.out.println(e.getMessage());
        }catch (SQLException e1){
            e1.printStackTrace();
            System.out.println("connection error! ");
            System.out.println(e1.getMessage());
        }finally {
            try{
                if(res!=null){
                    res.close();
                    res=null;
                }
                if(stm!=null){
                    stm.close();
                    stm=null;
                }
                if(con!=null){
                    con.close();
                    con=null;
                }
            }catch (SQLException e2){
                e2.printStackTrace();
                System.out.println("close connection or statement error! " );
                System.out.println(e2.getMessage());
            }
        }
    }
```

(6) 打包运行

a. 将步骤（1）中的 maven 工程进行编译打包，生成的带依赖的 jar 包（hiveDemo-1.0-jar-with-dependencies.jar）

b. 在客户端服务器上，执行如下命令：

hadoop jar hiveDemo-1.0-jar-with-dependencies.jar HIVE_JDBC.Hive_JDBC

4 最佳实践

4.1 Hive操作HBase表

4.1.1 概述

Hive 中的 StorageHandlers 功能不但可以让 Hive 基于 HBase 实现，还可以支持基于 Cassandra、Azure Table、JDBC(MYSQL and others)、MongoDB、ElasticSearch、Phoenix HBase、VoltDB 和 Google Spreadsheets 等实现。

Hive 的 StorageHandlers 原理是基于 Hive 以及 Hadoop 的可扩展性：

- 输入格式化
- 输出格式化
- 序列化/反序列化包

StorageHandlers 基于以上可扩展性外，还需实现元数据勾子接口，此接口允许使用 Hive 的 DDL 语句定义和管理 Hive 元数据及其它系统的目录。

通过 StorageHandlers，可以实现 Hive 与 HBase 的整合。Hive 与 HBase 的整合是利用两者本身对外的 API 接口相互通信来实现，相互通信主要是依靠 hive-hbase-handler.jar 工具类（Hive Storage Handlers），即 Hive Storage Handlers 负责 HBase 和 Hive 之间的通信。

说明

Hive 表若为分区表，则 insert 操作时会报错 must specify table name 错误，所以暂不支持 Hive 分区表操作 HBase 表。

4.1.2 操作示例

1. 示例一：在 Hive 中新建表，并关联到 HBase，通过向 Hive 表插入数据来向关联的 HBase 表中插入数据。

(1) 在 Hive 中创建 HBase 相关联的的表。执行如下命令：

```
CREATE TABLE hive_table(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")
TBLPROPERTIES ("hbase.table.name"= "hbase_table") ;
```

其中，hbase.table.name 用于定义在 HBase 中的 table 名称，hbase.columns.mapping 用于定义 HBase 中的列族。

(2) 在 Hive 中创建新表（步骤（1）Hive 中创建的与 HBase 整合的表不支持 load data 导入数据）。

```
create table pokes (foo int, bar string) row format delimited fields terminated by ',' stored as textFile;
```

(3) 将数据文件 1.txt 上传到 HDFS，然后向新表中导入数据：

load data inpath '/tmp/1.txt' overwrite into table pokes;

(4) 将数据导入到 hive_table 中，执行如下命令：

insert overwrite table hive_table select * from pokes;

(5) 可以在 Hive 中查看导入的数据。

(6) 进入 HBase shell 控制台，可以查看表 hbase_table 的表结构，查询该表中的数据。

```
# hbase shell
hbase(main):005:0>describe 'hbase_table'
hbase(main):005:0>scan 'hbase_table'
```

后续，在Hive中对hive_table表的更新操作都会相应更新HBase中的hbase_table表，反之，在 Hbase 中对 hbase_table 表的更新操作也会相应更新到 Hive 中的 hive_table 表。

2. 示例二： 在 Hive 中建立关联 HBase 的外部表，并查询 HBase 表中的数据。

(1) 在 Hive 中建立关联 HBase 的外部表，执行如下命令：

```
CREATE EXTERNAL TABLE hive_table2(key int, info string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = "cf1:val")
TBLPROPERTIES("hbase.table.name" = "hbase_table");
```

(2) 在 Hive 中查看到 HBase 表中的数据，执行如下命令：

select * from hive_table2;

4.2 Hive on Spark

4.2.1 概述

- Hive on Spark 是由 Cloudera 发起，由 Intel、MapR 等公司共同参与的开源项目，其目的是把 Spark 作为 Hive 的一个计算引擎，将 Hive 的查询作为 Spark 的任务提交到 Spark 集群上进行计算。通过该项目，可以提高 Hive 查询的性能，同时为已经部署了 Hive 或者 Spark 的用户提供了更加灵活的选择。
- Hive on Spark 总体的设计思路是，尽可能重用 Hive 逻辑层面的功能；从生成物理计划开始，提供一整套针对 Spark 的实现，比如 SparkCompiler、SparkTask 等，这样 Hive 的查询就可以作为 Spark 的任务来执行了。
- Hive on Spark 可在 Hive 组件自定义配置 hive-site 中添加以下参数以控制 Hive on Spark 的资源使用，具体说明如表 4-1 所示，其余参数可参考 Spark 的参数设置。

表4-1 参数说明

参数名称	参数说明
spark.executor.instances	executor个数
spark.driver.memory	driver内存大小
spark.executor.memory	excutor内存大小

4.2.2 操作示例

(1) 连接 Hive，执行如下命令，设置 Hive on Spark：

0: jdbc:hive2://cloudos2.hde.com:2181,cloudos> set hive.execution.engine=spark;

(2) 执行插入查询等操作：

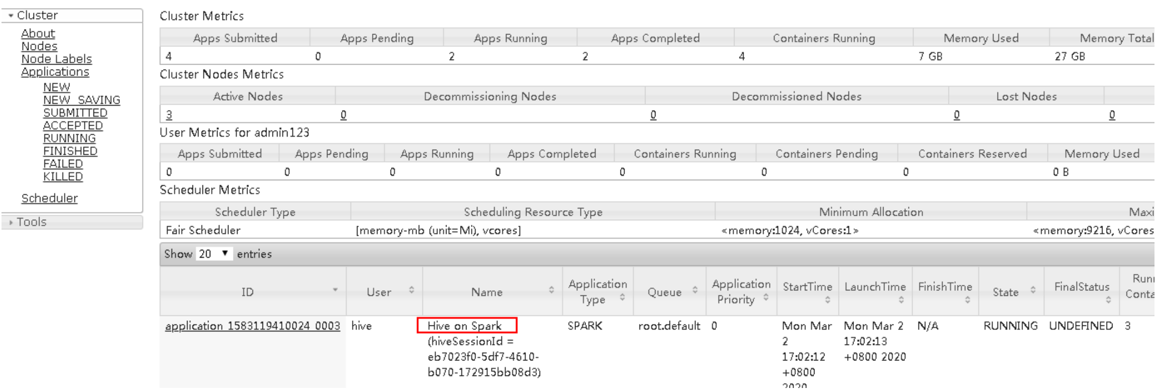
0: jdbc:hive2://cloudos2.hde.com:2181,cloudos> insert into hive1 values(1, 'dfs');

0: jdbc:hive2://cloudos2.hde.com:2181,cloudos> select * from hive1;

```
+-----------+-------------+
| hive1.id  | hive1.name  |
+-----------+-------------+
| 1         | dfs         |
+-----------+-------------+
```

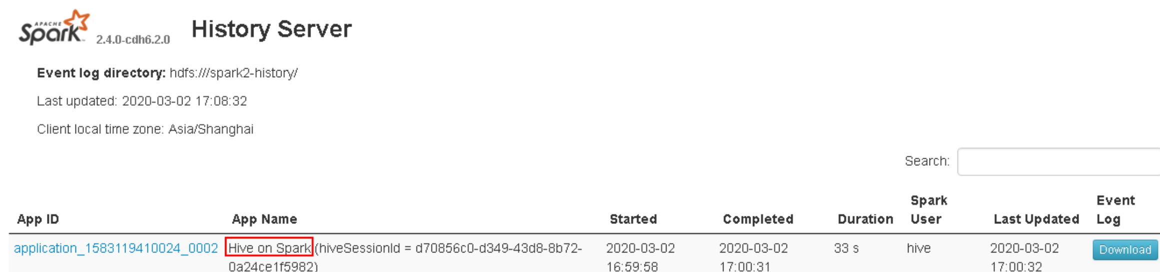
(3) 从 ResouceManager UI 中可看到，Hive 命令是通过 Spark 引擎执行的。

图4-1 ResouceManager UI



(4) 同时也可以从 Spark history UI 中查看到刚执行成功的任务。

图4-2 Spark history UI



5 常见问题解答

5.1 调优类

1. 使用 ORC 文件格式，提高查询速度

ORCfile 使用了 predicate push-down，compression 等多种技术。Hive 使用 ORCfile 作为表结构不仅可以节省存储空间，而且能够快速提高 Hive Query 的速度。

```
create table t_orc(
        id int,
        name string,
        age int,
        address string
) stored as ORC tblproperties("orc.compress"="SNAPPY");
```

2. 使用 VECTORIZATION，提高执行速度

矢量(Vectorized query)每次处理数据时会将 1024 行数据组成一个 batch 进行处理，而不是一行一行进行处理，这样能够显著提高执行速度。可以通过如下配置项来开启。

```
set hive.vectorized.execution.enabled = true;
set hive.vectorized.execution.reduce.enabled = true;
```

Vectorized 仅支持如下数据类型：

- tinyint
- smallint
- int
- bigint
- boolean
- float
- double
- decimal
- date
- timestamp
- string

【说明】：若使用其他数据类型会使 vetorized 失效，即每次还是一行一行读取数据。

当使用支持的数据类型时，也支持如下表达式：

- 数学运算：+,-,*,/,%
- and, or, not
- <, >, <=, >=, =, !=, between, in
- 使用 and, or, not, <, >, <=, >=, = !=的布尔表达式

- is [not] null
- 数学函数
- 字符串函数：substr, concat, trim, ltrim, rtrim, lower, upper, length
- 类型转换
- UDFs
- 时间函数
- If 表达式等

3. 使用 CBO 优化查询

CBO(COST BASED QUERY OPTIMIZATION)可以优化 Hive 的每次查询。如果想要使用 CBO，需要开启如下选项：

```
set hive.cbo.enable=true;
set hive.compute.query.using.stats=true;
set hive.stats.fetch.column.stats=true;
set hive.stats.fetch.partition.stats=true;
```

如果想要使用 CBO，需要通过 Hive 的分析模式来收集表的不同统计数据，执行如下命令：

```
analyze table table_name compute statistics for columns;
```

至此，Hive 才可以通过消耗评估和不同的执行计划来让查询跑的更快。

4. 修改 SQL 语句，优化执行计划

可以采用多种 SQL 语句实现该功能。不同 SQL 语句，执行计划也不同，导致执行时间也不同。因此分析执行计划，选择最优的 SQL 语句，是很有必要的。可通过 ***EXPLAIN*** 命令来查看 SQL 的执行计划：

```
EXPLAIN [EXTENDED|DEPENDENCY|AUTHORIZATION] query
```

场景示例：创建一个单击事件表，表中的每条数据代表一个单击事件。

- 创建表语句如下：

  ```
  create table t_click(
  click_time date,
  sessionId string,
  url string,
  source_ip string
  ) stored as ORC tblproperties("orc.compress"="SNAPPY");
  ```

  如果想要查询每个 sessionId 最后访问的 url，可以采用下面的查询语句。
- 查询语句示例（一）：

  ```
  select t.* from t_click t inner join
  ( select sessionId, max(click_time) as max_cs from t_click group by sessionId
  ) l on t.sessionId = l.sessionId and t.click_time = l.max_cs;
  ```

  - 查看示例（一）中语句的 explain 为：

    ```
    Vertex dependency in root stage
    Map 3 <- Reducer 2 (BROADCAST_EDGE)
    ```

```
Reducer 2 <- Map 1 (SIMPLE_EDGE)

Stage-0
  Fetch Operator
    limit:-1
    Stage-1
     ...
      Map Join Operator [MAPJOIN_28]
      ...
```

- 查询语句示例（二）：

  select * from (select * , rank() over (partition by sessionId order by click_time desc) as rank from t_click) r where r.rank=1;

  - 查看示例（二）中语句的 explain 为：

```
Vertex dependency in root stage
Reducer 2 <- Map 1 (SIMPLE_EDGE)

Stage-0
  Fetch Operator
    limit:-1
    Stage-1
      Reducer 2
      File Output Operator [FS_8]
        compressed:false
        ...
```

通过查看 explain 语句可知，查询示例（一）通过一个子查询获取每个 sessionId 最后访问时间，之后通过 inner join 来过滤其它的事件；查询示例（二）通过 hive 的开窗函数避免了两个大表的 join，查询示例（二）查询效率显著提高。

5. Hive 利用严格模式，防止执行效果差的语句

Hive 提供了严格模式，可以防止用户执行一些可能产生不好效果的查询，即在严格模式下，可以禁止用户执行某些操作。当设置 hive.mapred.mode=strict，会禁止执行以下三种查询：

- 带有分区的表的查询

  如果在一个分区表执行 Hive，除非 where 语句中包含分区字段过滤条件来显示数据范围，否则不允许执行，即在严格模式下用户不允许扫描所有的分区。

  进行这个限制的原因是，通常分区表都拥有非常大的数据集，而且数据增加迅速。没有进行分区限制的查询可能会消耗巨大资源来处理这个表。

- 带有 orderby 的查询

  对于使用了 orderby 的查询，要求必须有 limit 语句。因为 orderby 为了执行排序过程会将所有的结果分发到同一个 reducer 中进行处理，强烈要求用户增加这个 limit 语句用于防止 reducer 额外执行太长时间。

- 限制笛卡尔积的查询

  对关系型数据库非常了解的用户，可能期望在执行 join 查询的时候不使用 on 语句而是使用 where 语句，这样关系数据库的执行优化器就可以高效的将 where 语句转换成 on 语句。而 Hive 不会执行这种优化。如果表非常大时，这个查询就可能会出现 OOM（Out Of Memory，内存溢出）等情况。

6. Hive/Spark 任务性能调优

Spark 作为 Hive 的一个计算引擎，将 Hive 的查询作为 Spark 的任务提交到 Spark 集群上进行计算。当 Hive 查询出现性能问题时，需要先进行问题定位分析性能瓶颈，然后进行性能调优。

若确定业务 sql 实现方式无严重性能问题，则可通过以下步骤进行逐步的性能瓶颈分析：

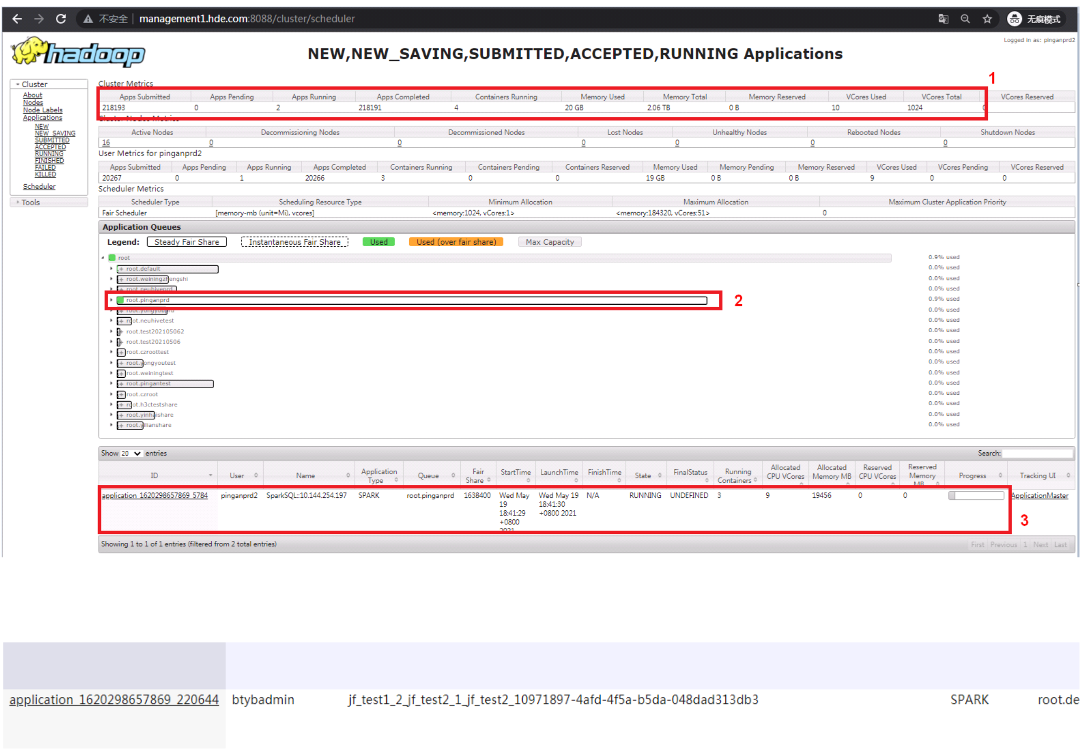
(1) 通过监控界面查看任务监控细节

打开任务运行监控界面并进入任务运行列表页，根据融合集成平台中 Hive/Spark 运行提交时间或根据运营平台 Hive/Spark 任务返回的 applicationId，找到需要对应的任务。

(2) 确定任务计算资源分配是否充足

进入任务列表页，查看该任务执行时占用的计算资源量，如图 5-1 所示。计算资源包括 vcore 数和内存资源。其中，vcore 数决定任务执行并行度，内存决定分布式计算内存数。任务分配的计算资源较少，即任务计算资源分配不足。

图5-1 ResouceManager UI



(3) 确认 sql 执行过程中是否出现数据倾斜

首先确定任务的 applicationId，然后点击该任务链接，进入任务监控详情查看 MapReduce 任务和 Spark 任务相关信息。

- MapReduce 任务：任务运行一段时间后，进入任务详情查看该任务的 Map 及 Reduce 列表。若出现任务执行后期仅有 1～3 个 Map 或 Reduce 在同时运行，基本可以认定存在数据倾斜。
- Spark 任务： 任务运行一段时间后，进入任务详情查看该任务 Executor 列表。若存在其中 1～3 个 Executor 执行了查询中 50%以上的任务，基本可以认定存在数据倾斜。

根据以上性能瓶颈分析定位的问题，分别进行性能调优，性能调优策略如下：

- 若出现任务计算资源分配不足
  - 根据 sql 关联查询的表数据量及关联查询复杂度评估并调整计算资源。
- 若 sql 计算资源分配足够，但任务运行仍然较慢
  - 分析 sql 执行过程中是否出现笛卡尔积。通常生产场景下严格禁止笛卡尔积，若出现笛卡尔积，则该 sql 禁止执行。
  - 分析业务 sql 实现上是否性能最佳。sql 实现中需避免使用无用的关联查询、聚合分组等严重影响性能的实现方式。
  - 业务 sql 中存在关联查询时， 小于 25MB 的小表强烈建议放在关联操作的左边。
  - 确定业务 sql 实现上已保证性能最佳，但实际执行过程仍然较慢，建议分析 sql 执行过程中是否出现数据倾斜。若出现数据倾斜， 首先考虑 sql 实现上避免严重数据倾斜。如无法避免，建议增加并行度(若关联表中字段值存在少数 key 数据量比较大，也可考虑数据打散操作)，同时适当降低每个执行线程的内存。因为在此性能瓶颈下，再增加内存资源对性能优化效果不再明显，并且由于计算资源有限，增加并行度会成倍的增加总内存消耗。
  - SparkSQL 业务 sql 中若使用了临时表或者中间表，建议增加 driver 端内存。
  - sql 关联查询的表数据量较大或者关联查询较复杂，但是无法再分配更多的计算资源。此场景下建议将该 sql 语句拆分成多条子 sql 执行，sql 中可以按过滤字段值或分区字段值对该 sql 进行拆分。

5.2 运维类问题

1. 使用 HiveServer2 创建 HBase 数据源表时，界面卡住，后台日志出现 KeeperErrorCode = NoNode for /hbase/hbaseid 提示

- 在 Hive 中创建 HBase 数据源表时，要求集群中必须安装 HBase 组件。若在 HiveServer2 启动时，集群中已经安装 HBase，则 HiveServer 创建 HBase 数据源表可以成功。
- 若在 HiveServer2 启动时，集群中没有安装 HBase，则需要在安装了 HBase 后重启 Hive 组件，然后 HiveServer2 才能创建 HBase 数据源表成功。

2. Hive JDBC 连接执行 load data local inpath 时路径指定为/tmp 下报无效路径，如何解决？

Hive jdbc 连接执行加载本地文件时，要求：

- 每个节点都有该数据文件，同时要求 location 的 owner 和建表用户一致，且拥有 rw 权限。

3. Hive server2 状态不稳定，客户执行 beeline 有时候报错

- 原因分析：Hive server2 内存不足，Tez container 内存不足
- 定位思路：通过日志查看，若 tez heap 内存溢出较频繁，则可能是处理的数据量较大导致 heap 内存溢出导致。

- 解决方法：根据客户数据量在 DE 中调大 Hive 中的 hive.tez.container.size 和 hive.heapsize 参数

4. 任务执行出现异常，报错：Invalid resource request,requested memory < 0, or requested memory > max configured, requestedMemory=6826, maxMemory=4096

- 原因分析：该问题是任务所需内存超出 yarn container 的最大内存。
- 定位思路：查看 yarn container 配置大小是否小于 6826M。
- 解决方法：在 YARN 配置界面调整 container 的最大值超过所需内存大小便可解决。

5. 集群中添加新节点进行扩容后，Hive 或 Spark 的 beeline 连接出现异常

- 原因分析：集群中添加新节点后，HDFS 以及系统 hosts 配置文件都进行了更新。但 HiveServer 及 SparkThriftServer 加载的配置信息却仍为添加节点之前的信息，所以在客户端连接时会出现 HiveServer&SST 与 HDFS 配置不一致的情况，导致连接失败。
- 定位思路：检查 HiveServer 和 SparkThriftServer 是否已经重启
- 解决方法：集群扩容增加新节点后，需要重启 HiveServer 和 SparkThriftServer，以保证服务可加载集群的最新配置，确保 JDBC 连接正常。

6. Hive 在 hdfs 用户下创建外部表时指定外部表在 hdfs 的存储路径后显示没有权限

此问题由操作方式不当引起。因为创建 Hive 外部表时，会对指定的 location 进行权限检查，要求：

- 如果 location 存在，要求 location 的 owner 和建表用户一致，且拥有 rw 权限
- 如果 location 不存在，则会检查 location 的父级目录，要求父级目录的 owner 和建表用户一致；且要求父级目录所有子目录（包括子目录的子目录）的 owner 和建表用户一致，且拥有 rw 权限。

7. Hive 提交 join 任务执行失败

- 原因分析：Hive 进行大小表的 join，且开启了自动转换；此时 Hive 会识别小表，并用 mapJoin 来实现两个表的联合，造成内存溢出，MapReduce 任务执行失败。
- 定位思路：用两张普通表测试 hive 的 join 是否正常；排查客户同步的两张表结构；排查 hive 是否开启自动转换。
- 解决方法：关闭自动转换：set hive.auto.convert.join = false。

8. Hive 提交带函数的语句时报错

- 原因分析：提交任务的用户错误。
- 定位思路：排查错误日志，发现：running as root is not allow。
- 解决方法：处理步骤是切换到 hdfs 用户进行提交。

9. 执行 Hive SQL 语句失败

- 原因分析：Hive 堆内存过小导致任务无法执行
- 定位思路：排查错误日志，发现：java.lang.OutOfMemoryError:GC overhead limit exceeded。
- 解决方法：在 hive 界面调整堆内存大小不生效，hive 堆内存大小对应的是 hdfs 的 hadoop_heapsize 参数。

10. Hue 执行 Hive 语句报错或无法执行

- 原因分析：

- 可能是硬盘或网络原因导致 YARN 写 filecache（/hadoop/yarn/local/filecache/）失败，Hive 提交 MapReduce 任务无法执行。
- SQL 语句错误。
- 权限不足。

- 定位思路：
  - 排查 Hive 日志：Error while processing statement:FAILED:Execution Error. Return code 2 from org.apache.hadoop.hive.ql.exec.mr.MapRedTask。
  - 报错信息：ParseException line 1:9 character ‘ ’not supported here，客户输入的 sql 语句有非法空格；
  - 报错信息：Principal [name=admin,type=USER] does not have following privileges for operation QUERY [[SELECT]]，客户利用 admin 用户查询 Hive 表，但没有对应表权限。
- 解决方法：
  - 重启 YARN 即可。
  - 调整 SQL 语句后正常。
  - 退出登录后，采用有权限操作的用户进行查询后正常。

11. Hive beeline 连接失败，报错：GSS initiate failed

- 原因分析：beeline 连接时候 principle 配置错误。
- 定位思路：查看 HDFS 对应的 principle，确认配置是否正确。
- 解决方法：beeline 连接时采用正确的 principle 即可。

12. Hive 客户端提交 SQL 语句，报错：SemanticException Failed to get a spark session

- 原因分析：Hive 使用 Spark 计算引擎，在执行 SQL 语句时会向 Yarn 申请资源启动 SparkSession，当资源不足时，SparkSession 会处于等待中，此时就会引起启动 SparkSession 超时；或者当网络异常或者节点响应较慢时，也会引起该问题。
- 定位思路：修改 Hive 对应配置项。
- 解决方法：Hive 服务中 hive-site 配置中，修改配置 hive.spark.client.server.connect.timeout，默认值为 600000，可以适当调大，如 1200000。

13. 客户端获取结果时造成 HiveServer2 服务关掉

- 原因分析：客户端提交 SQL 语句到 HiveServer2，并从 HiveServer2 端获取 SQL 语句执行结果，默认是每次以 1000 条拉取结果，该值可以满足常用场景。若获取 SQL 语句中每条结果太大时会造成 HiveServer2 出现 OOM 现象，引起该服务停掉。
- 定位思路：修改 Hive 对应配置项。
- 解决方法：在 HiveServer2 服务中 hive-site 配置中，添加 hive.server2.thrift.resultset.max.fetch.size 配置项，设置较小值（如 300）。

14. Hive-jdbc 高并发执行 SQL 语句时，HiveServer2 由于 jvm 堆内存溢出：java.lang.OutOfMemoryError: Java heap space 不可用

- 原因分析：客户端高并发（HiveServer2 默认最大支持 500 并发）通过 hive-jdbc 提交 SQL 语句到 HiveServer2，并从 HiveServer2 端获取 SQL 语句执行结果。由于存在结果缓存，高并发下需要占用较多堆内存，若 HiveServer2 进程配置的内存过小则容易出现该异常。
- 定位思路：修改 Hive 对应配置项。

- 解决方法：在 HiveServer2 服务组件的基础配置中 hive-site 配置中调大 HiveServer2 Heap Size 的值（比如 16GB）。

15. Hive 任务 Insert overwrite 语句，目标表分区 13 万左右，任务执行到最后 move 阶段，移动临时结果至对应分区执行缓慢

- 原因分析：过多的分区会导致 HiveMetaStore 的元数据更新压力大，同时也增加 NameNode 的负担，进而影响一系列依赖于 NameNode 的服务。
- 解决方法：
    - 对表设置合理的分区，单表分区数不要超过 10 万。
    - 生成数据时尽量避免生成小文件，从而减少 NameNode 的压力。

16. Hive 执行 LOAD DATA LOCAL INPATH 语句报错 Invalid path "***": No files matching path file

- 原因分析：Hive 使用 LOAD DATA LOCAL INPATH 时，数据文件必须与 HiveServer2 服务在同一个节点。否则，当 HiveServer2 有多个，并且以 HA 模式使用 beeline 连接 Hive 时，会随机选择一个 HiveServer2 进行连接，然后在执行 LOAD DATA LOCAL INPATH 语句的时候，如果在连接到的 HiveServer2 节点对应目录不存在指定的文件就会报该错误。
- 解决方法：当以 HA 模式使用 beeline 连接 Hive 时，需要上传数据文件到所有的 HiveServer2 节点对应目录。

目 录

1 组件简介 ······ 1-1
1.1 组件概述 ······ 1-1
1.2 组件架构 ······ 1-1
1.2.1 HBase 系统架构 ······ 1-1
1.2.2 HBase 数据模型 ······ 1-3
1.2.3 元数据表 hbase:meta ······ 1-4
1.2.4 HBase HA ······ 1-4
1.3 应用场景 ······ 1-4
2 快速入门 ······ 2-1
2.1 组件安装 ······ 2-1
2.1.1 数据目录检查 ······ 2-1
2.1.2 查看相关组件的日志信息 ······ 2-2
2.2 运行状态监控 ······ 2-2
2.2.1 查看组件详情 ······ 2-2
2.2.2 组件检查 ······ 2-3
2.3 快速使用指导 ······ 2-5
2.3.1 非 Kerberos 环境 ······ 2-5
2.3.2 Kerberos 环境 ······ 2-10
2.4 快速链接 ······ 2-12
2.4.1 配置组件快速链接 ······ 2-12
2.4.2 访问 HBase 监控页面 ······ 2-12
3 使用指南 ······ 3-1
3.1 Client 下载/安装/使用/卸载 ······ 3-1
3.1.1 下载 Client 安装包 ······ 3-1
3.1.2 安装 Client ······ 3-2
3.1.3 访问组件 ······ 3-3
3.1.4 卸载 Client 客户端 ······ 3-3
3.2 HBase 集群的扩容 ······ 3-4
3.2.1 使用场景 ······ 3-4
3.2.2 扩容前准备 ······ 3-4
3.2.3 扩容约束 ······ 3-4
3.2.4 扩容影响 ······ 3-4

3.2.5 扩容操作指导 3-5
3.2.6 RegionServer 扩容后 Balance 3-6
3.2.7 扩容验证 3-6
3.3 HBase 集群的缩容 3-6
3.3.1 使用场景 3-6
3.3.2 缩容前准备 3-7
3.3.3 缩容约束 3-7
3.3.4 缩容影响 3-7
3.3.5 缩容操作指导 3-7
3.3.6 缩容验证 3-8
3.4 权限访问控制 3-9
3.4.1 权限说明 3-9
3.4.2 权限使用操作示例 3-9
3.5 租户管理 3-11
3.5.1 租户介绍 3-12
3.5.2 新增租户 3-12
3.5.3 租户使用操作示例 3-13
3.6 备份恢复 3-14
3.6.1 新建 HBase 同步任务 3-15
3.6.2 源集群和目的集群配置互信 3-17
3.6.3 HBase 同步任务相关配置 3-19
3.6.4 HBase 历史数据同步 3-19
3.7 数据迁移 3-20
3.7.1 数据迁移常用方案 3-20
3.7.2 数据迁移案例 3-21
3.8 HBCK 工具使用指南 3-22
3.8.1 Procedure 简介 3-22
3.8.2 HBCK 工具命令格式 3-23
3.8.3 HBCK 功能 3-23
4 开发指南 4-1
4.1 常用 API 示例 4-1
4.1.1 开发环境准备及注意事项 4-1
4.1.2 非 Kerberos 环境 Java 开发样例 4-1
4.1.3 Kerberos 环境 Java 开发样例 4-10
4.1.4 HBase 高级专题 4-19
4.1.5 HBase 其它常用接口 4-25

5 常见问题解答 5-1
5.1 HBase Schema 及 Rowkey 设计 5-1
5.1.1 HBase 表 Schema 设计原则 5-1
5.1.2 HBase 表 Rowkey 设计原则 5-1
5.2 调优 5-3
5.2.1 组件端调优 5-3
5.2.2 其它调优 5-3
5.3 运维类问题 5-4
5.3.1 日志查看方法 5-4
5.3.2 常见问题 5-5

1 组件简介

1.1 组件概述

HBase是一个高可靠性、高性能、面向列、可伸缩的分布式NoSQL数据库。HBase适合于存储大表数据（表的规模可以达到数十亿行以及数百万列），并且对大表数据的读、写访问可以达到实时级别。

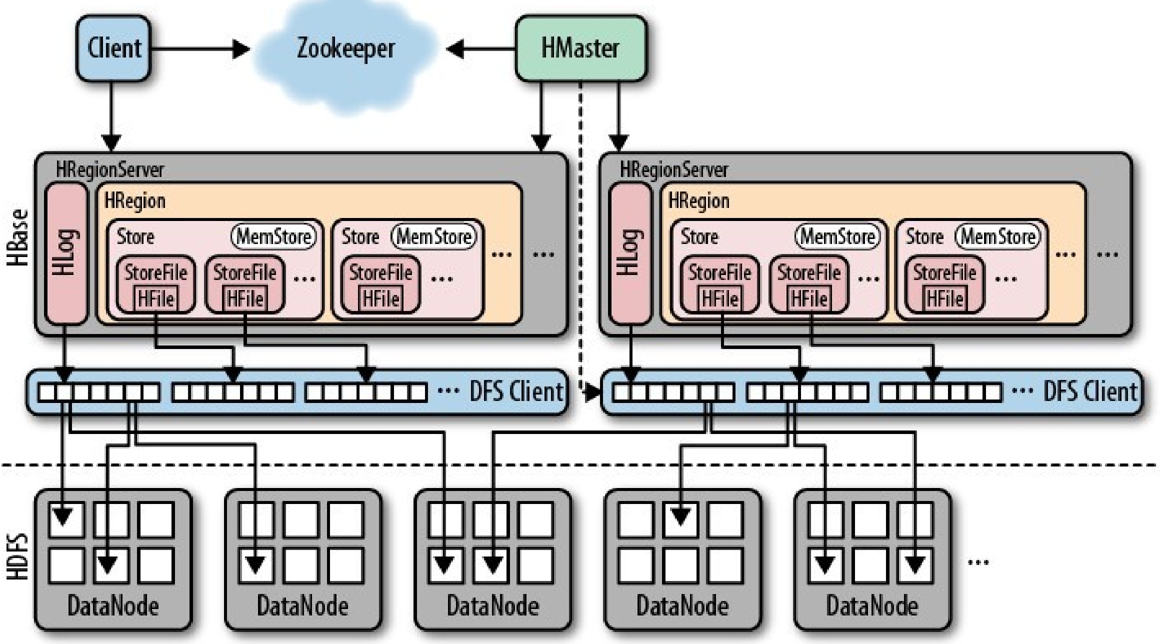
HBase依赖于HDFS及Zookeeper，将HDFS作为底层文件存储系统，ZooKeeper作为协同组件，并为Spark和MapReduce提供海量数据实时处理能力。

1.2 组件架构

1.2.1 HBase系统架构

HBase 系统架构由 Client、ZooKeeper、HMaster、HRegionServer、HRegion、HStore、HLog、HDFS等部件组成，如图 1-1 所示。

图1-1 HBase整体架构



1. Client

Client使用HBase RPC机制与HMaster进行通信，与HRegionServer进行数据读写操作。

2. ZooKeeper

- ZooKeeper Quorum存储HMaster的地址。

- HRegionServer 以 Ephedral 方式注册到 ZooKeeper 中，HMaster 随时感知各个 HRegionServer 的健康状况。
- ZooKeeper 避免了 HMaster 单点故障问题。

3. HMaster

HBase 中可以启动多个 HMaster，通过 ZooKeeper 的 Master Election 机制保证总有一个 Master 在运行，不会出现单点故障。

HMaster 主要负责 Table 和 Region 的管理工作：

- 管理用户对表的增、删、改、查操作。
- 管理 HRegionServer 的负载均衡，调整 Region 分布。
- Region Split 后，负责新 Region 的自动平衡分布。
- 在 HRegionServer 停止后，负责失效 HRegionServer 上 Region 迁移。

4. HRegionServer

HRegionServer 是 HBase 中最核心的模块，主要负责响应用户 I/O 请求，向 HDFS 文件系统中读写数据。HRegionServer 管理 HRegion 对象；每个 HRegion 对应 Table 中一个 Region，HRegion 由多个 HStore 组成；每个 HStore 对应 Table 中一个 Column Family 的存储。RegionServer 结构说明如表 1-1 所示。

表1-1 RegionServer 结构说明表

名称	说明
Store	一个Region由一个或多个Store组成，每个Store对应一个Column Family
MemStore	一个Store包含一个MemStore，MemStore缓存客户端向Region插入的数据，当RegionServer中的MemStore大小达到配置的容量上限时，RegionServer会将MemStore中的数据“flush”到HDFS中
StoreFile	MemStore的数据flush到HDFS后成为StoreFile，随着数据的插入，一个Store会产生多个StoreFile，当StoreFile的个数达到配置的最大值时，RegionServer会将多个StoreFile合并为一个大的StoreFile
HFile	StoreFile在文件系统中的存储格式
HLog	HLog文件保证了当RegionServer故障的情况下用户写入的数据不丢失，每个RegionServer对应一个Hlog

5. Region

当 Table 随着记录数不断增加而变大后，会逐渐分裂成多份 splits 成为 Regions，一个 Region 由 [startkey,endkey)表示，不同的 Region 会被 Master 分配给相应的 RegionServer 进行管理。

6. HStore

HBase 存储的核心由 MemStore 和 StoreFile 组成。

7. HLog

在分布式系统环境中，无法避免系统出错或者宕机，一旦 HRegionServer 意外退出，MemStore 中的内存数据就会丢失，为防止数据丢失，引入了 HLog。

每次用户操作写入 Memstore 的同时，也会写一份数据到 HLog 文件，HLog 文件定期会滚动更新，并删除旧的文件（已持久化到 StoreFile 中的数据）。当 HRegionServer 意外终止后，HMaster 会通过 ZooKeeper 感知，HMaster 首先处理遗留的 HLog 文件，将不同 Region 的 log 数据拆分，分别放到相应 Region 目录下，然后再将失效的 Region 重新分配，领取到这些 Region 的 HRegionServer 在 Load Region 的过程中，会发现有历史 HLog 需要处理，因此会 Replay HLog 中的数据到 MemStore 中，然后 flush 到 StoreFiles，完成数据恢复。

8. HDFS

HDFS 是一个分布式文件系统，通过将一个大的文件划分成多个固定大小的 Block 来实现分布式存储。每一个 Block 都存在多个备份，并且被部署在不同的数据节点上，来保障数据的安全。目前，HBase 的所有底层数据都以文件的形式交由 HDFS 存储，HBase 本身不固化保存数据信息。

1.2.2 HBase 数据模型

HBase 以表的形式存储数据，表中的数据划分为多个 Region，并由 Master 分配给对应的 RegionServer 进行管理。

每个 Region 包含了表中一段 Row Key 区间范围内的数据，HBase 的一张数据表开始只包含一个 Region，随着表中数据的增多，当一个 Region 的大小达到容量上限后会分裂成两个 Region。在创建表时可以定义 Region 的 Row Key 区间，或者在配置文件中定义 Region 的大小。

HBase 以表的形式存储数据。表由行和列族组成，列族由若干个列组成，其逻辑视图如表 1-2 所示。

表1-2 员工出差表

行键（name）	时间戳（time）	列族（address）	
		列（province）	列（city）
Lilei	time3	Shandong	Yantai
	time2	Zhejiang	Hangzhou
	time1	Jiangsu	Nanjing
Wanggang	time5	Guangdong	Guangzhou
	time4	Sichuan	Chengdu

HBase 的数据模型关键概念：

- 行键（RowKey）

  行键是字节数组，任何字符串都可以作为行键。HBase 表中的行根据行键进行排序，数据按照 Row key 的字节序(byte order)排序存储。所有对表的访问（单个 RowKey 访问、RowKey 范围访问或全表扫描)都要通过行键。

- 列族（ColumnFamily）

  列族必须在表定义时给出，每个列族可以有一个或多个列，列不需要在表定义时给出，新的列可以随后按需加入，数据按列族存储，这种设计非常适合数据分析的情形。

- 列（Column）

  与传统的数据库类似，HBase 的表中也有列的概念，列用于表示相同类型或相同 schema 的数据。

- 时间戳（TimeStamp）

  每个 Cell 可能有多个版本，它们之间用时间戳区分。

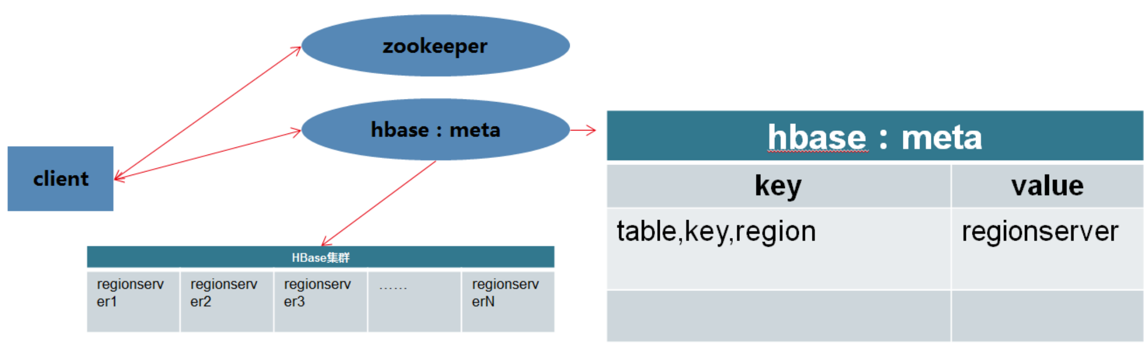
- 单元格（Cell）

  Cell 由行键、列族：列名、时间戳唯一决定；Cell 中的数据是没有类型的，全部以字节码形式存储。

1.2.3 元数据表 hbase:meta

HBase 中有张存放用户元数据的表，名为 hbase:meta，和用户表的关系如图 1-2 所示。

图1-2 hbase:meta 表结构



客户端访问数据的流程为：

Client → ZooKeeper →hbase:meta → User table

Client 访问 ZooKeeper，查找到 hbase:meta 表的存放位置，通过 hbase:meta 获取存放数据的 Region 信息（找到 RegionServer），通过 RegionServer 获取查找的数据。

1.2.4 HBase HA

HBase 中的 HMaster 负责 Region 分配，当 RegionServer 进程停止后，HMaster 把相应 Region 迁移到其他 RegionServer。如果 HBase 中只配置一个 HMaster，那么 HMaster 存在单点故障问题，引入 HMaster HA 模式，可以增强 HMaster 的可靠性。

HBase 默认开启 HA。大数据集群中将两台独立的机器配置为 HMaster，在任何时候只有一个 HMaster 处于 Active 状态，另一个 HMaster 处于 Standby 状态。Active HMaster 负责所有的管理工作，并且当 Active HMaster 无法对外提供服务时，Standby HMaster 将接替 Active HMaster 的工作。

1.3 应用场景

1. 半结构化或非结构化数据

对于数据结构字段不够确定或者杂乱无章很难进行抽取的数据适合使用 HBase。

2. 记录非常稀疏

在关系型数据库中，一个表有多少列是固定的，当记录非常稀疏时会浪费存储空间，这种情况下适合使用 HBase。

3. 多版本数据

HBase 通过时间戳可以有任意数量的版本值，因此对于需要存储历史数据的场景，用 HBase 非常方便。

4. 业务场景简单

HBase 不支持复杂的查询，当业务场景比较简单时适合使用 HBase。

5. 超大数据量

HBase 使用 HDFS 作为文件系统，易于横向扩展。当数据量很大时，查询效率优于关系型数据库。

2 快速入门

2.1 组件安装

注意

- 在 Hadoop 集群中，安装 HBase 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- HBase HMaster 与 RegionServer 可以装在同一个实例上，为了不影响高性能的情况下，建议不要安装在同一实例上。
- 部署 HBase 时，集群节点规模应随业务需求及数据量进行调整。

2.1.1 数据目录检查

根据现场磁盘分区方案和挂盘方案的不同，组件安装完成后，必须对各组件的数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表 2-1 所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
HDFS	是（配置项的参数值默认选择3个挂载路径）	dfs.namenode.name.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
	是（配置项的参数值默认选择全部挂载路径）	dfs.datanode.data.dir	
	未开启高可用时，需要检查该配置项（配置项的参数值默认使用全部挂载路径）	dfs.namenode.checkpoint.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
	开启高可用时，需要检查该配置项（配置项的参数值默认选择一个挂载路径）	dfs.journalnode.edits.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
YARN	是（配置项的参数值默认使用全部挂载路径）	yarn.nodemanager.local-dirs	此目录为数据目录，用于存放应用程序的运行依赖包等信息。检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
		yarn.nodemanager.log-dirs	
	是（配置项的参数值	yarn.timeline-service.l	此目录为数据目录，用于记录应用程序运行状

组件	是否需要检查	被影响的配置项	如何解决
	默认使用某一个挂载路径）	eveldb-state-store.path	态等信息。检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
		yarn.timeline-service.leveldb-timeline-store.path	
Zookeeper	是（配置项的参数值默认使用某一个挂载路径）	dataDir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录

2.1.2 查看相关组件的日志信息

表2-2 组件日志路径说明

组件	日志路径
HDFS	/var/de_log/hadoop/user_hdfs
MapReduce2	HistoryServer日志路径：/var/de_log/hadoop-yarn/user_mapred/和/var/de_log/hadoop-mapreduce/mapred/
YARN	/var/de_log/hadoop-yarn/user_yarn 【说明】上述的日志是YARN本身的日志。另外： • 执行在 YARN 上的应用日志，可以通过 YARN 的 UI 界面查看，日志文件实际存储位置默认是在 HDFS 上，默认路径为/app-logs • 如果日志不聚合，可以配置 yarn.log-aggregation-enable 为 false • 内嵌的 HBase 日志路径为：/var/de_log/hadoop-yarn/embedded-yarn-ats-hbase，在 timeline 安装的节点上可看到（前提条件：配置项 use_external_hbase、is_hbase_system_service_launch 的值均为 false）
HBase	/var/de_log/hbase/user_hbase
ZooKeeper	/var/de_log/zookeeper/user_{user.name}/，其中${user.name}是指执行任务的用户名

2.2 运行状态监控

2.2.1 查看组件详情

进入 HBase 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、概要、部署拓扑、配置详情和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

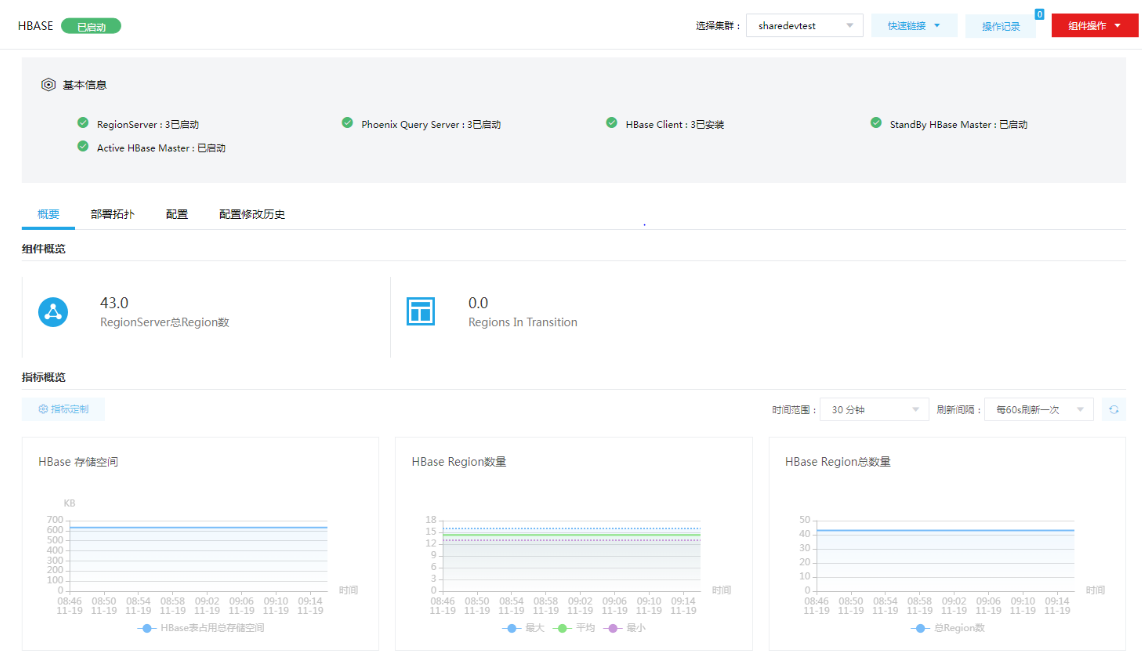
主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。

- 概要：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 组件详情



2.2.2 组件检查

执行 HBase 组件检查时，会执行创建表、禁用表、读写数据等一系列操作，验证 HBase 相关功能是否可用。组件检查成功表示 HBase 组件可正常使用。

集群在使用过程中，根据实际需要，可对组件执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：

   - 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
     - 在集群详情页面选择[组件]页签，单击业务组件列表中 HBase 组件对应的<组件检查>按钮。
     - 在集群详情页面选择[组件]页签，单击业务组件列表中 HBase 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。

- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

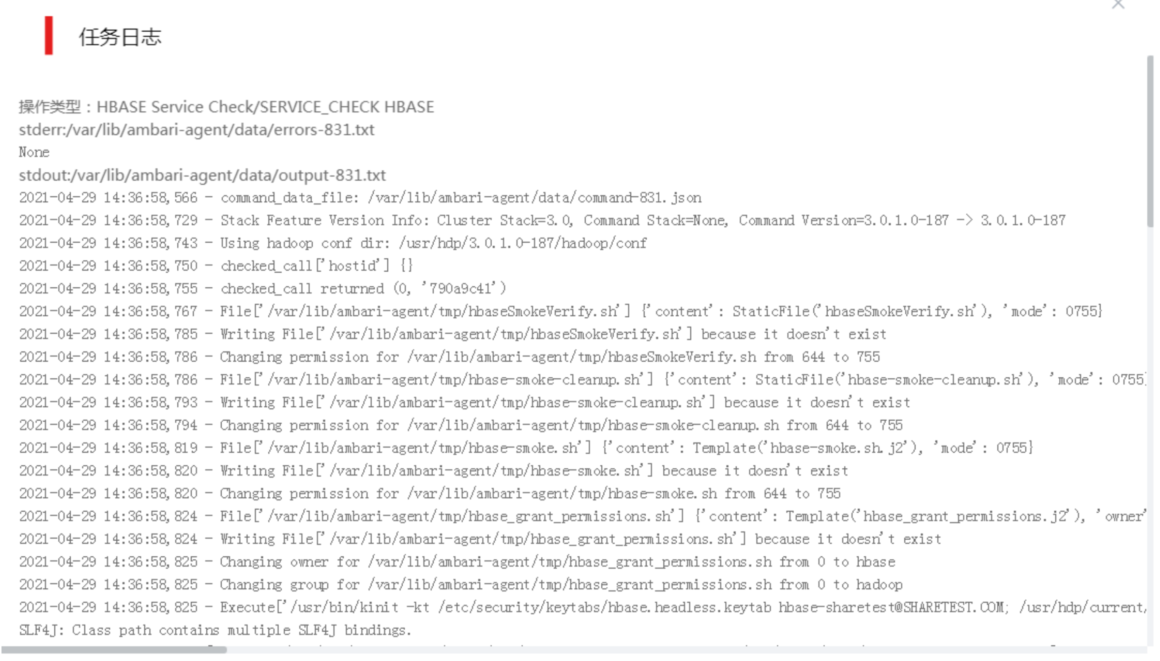
(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态，如图 2-2 所示，表示组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“HBase Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件日志检查详情



2.3 快速使用指导

注意

根据大数据集群是否开启 Kerberos 认证，用户访问 HBase 时的认证方式不同，详情请参见本章节内容。

HBase 既可以通过集群用户连接，又可以通过组件超级用户连接。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
  - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为"集群超级用户"。
  - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 HBase 组件的 hbase 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 Kerberos 环境

说明

- 非 Kerberos 环境下，不需要用户做身份认证即可直接对 HBase 执行管理操作。
- 本章节操作需要切换至具有 HBase 权限的用户。

1. 管理命名空间

在 HBase 中，命名空间指一组表的逻辑分组，类似于 RDBMS 中的 DataBase。命名空间的用法举例如下：

(1) 启动 HBase Shell

```
hbase shell
```

(2) 创建命名空间：

```
hbase>create_namespace 'ns01'
```

(3) 显示所有命名空间：

```
hbase>list_namespace
```

(4) 在命名空间下创建表：

```
hbase>create 'ns01:tab01' ,'cf'
```

(5) 显示命名空间下的所有表：

```
hbase>list_namespace_tables 'ns01'
```

(6) 删除命名空间：

hbase>drop_namespace 'ns01'

注意：只有命名空间为空时才允许删除。

2. 创建表

HBase Shell 客户端使用 create 关键字来创建表，可以通过 ***help 'create'***命令查看具体使用方法。创建表时，可以选择多个参数，但表名和列族名是必须的参数，用法举例如下：

- 指定命名空间和表名来创建表：

  hbase> create 'ns1:t1', {NAME => 'f1', VERSIONS => 5}

- 使用默认命名空间来创建表：

  hbase> create 't1', {NAME => 'f1'}, {NAME => 'f2'}, {NAME => 'f3'}

- 使用默认命名空间来创建表，可以简化成如下方式：

  hbase> create 't1', 'f1', 'f2', 'f3'

- 建表时可以指定版本数、TTL 等参数：

  hbase> create 't1', {NAME => 'f1', VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true}

  hbase> create 't1', {NAME => 'f1', CONFIGURATION => {'hbase.hstore.blockingStoreFiles' => '10'}}

- 建表时将 region 预分区：

  hbase> create 'ns1:t1', 'f1', SPLITS => ['10', '20', '30', '40']

- 建表时指定分区数量和分区方式：

  hbase> create 't1', 'f1', {NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

- 创建表时，可以指定一个引用，这样可以通过这个引用来调用方法：

  hbase> t1 = create 'table1', 'f1'

  hbase> t1.put 'rowkey1', 'f1:col1','value1'

  hbase> t1.scan

3. 插入数据

put 数据时，必选参数是表名、RowKey、列名（包括列族和列名）和值，可选参数包括时间戳，用法举例如下：

hbase> put 'ns1:t1', 'r1', 'c1', 'value'

hbase> put 't1', 'r1', 'c1', 'value'

hbase> put 't1', 'r1', 'c1', 'value', ts1

hbase> put 't1', 'r1', 'c1', 'value', {ATTRIBUTES=>{'mykey'=>'myvalue'}}

hbase> put 't1', 'r1', 'c1', 'value', ts1, {ATTRIBUTES=>{'mykey'=>'myvalue'}}

hbase> put 't1', 'r1', 'c1', 'value', ts1, {VISIBILITY=>'PRIVATE|SECRET'}

4. 查询一行数据

查询一行数据时，必选参数是表名和 RowKey，可选参数包括列名（包括列族和列名）、时间戳、版本数等，用法举例如下：

hbase> get 't1', 'r1'

hbase> get 't1', 'r1', {TIMERANGE => [ts1, ts2]}

hbase> get 't1', 'r1', {COLUMN => 'c1'}

hbase> get 't1', 'r1', {COLUMN => ['c1', 'c2', 'c3']}

hbase> get 't1', 'r1', {COLUMN => 'c1', TIMESTAMP => ts1}

hbase> get 't1', 'r1', {COLUMN => 'c1', TIMERANGE => [ts1, ts2], VERSIONS => 4}

hbase> get 't1', 'r1', {COLUMN => 'c1', TIMESTAMP => ts1, VERSIONS => 4}

hbase> get 't1', 'r1', 'c1'

hbase> get 't1', 'r1', 'c1', 'c2'

hbase> get 't1', 'r1', ['c1', 'c2']

5. 查询多行数据

查询多行数据，必选参数是表名，可选参数包括列名（包括列族和列名）、起止 Key、Filter，用法举例如下：

hbase> scan 'ns1:t1'

hbase> scan 'ns1:t1', {COLUMNS => 'info:regioninfo'}

hbase> scan 'ns1:t1', {COLUMNS => ['c1', 'c2'], LIMIT => 10, STARTROW => 'xyz'}

hbase> scan 'ns1:t1', {COLUMNS => 'c1', TIMERANGE => [1303668804, 1303668904]}

hbase> scan 'ns1:t1', {FILTER => "(PrefixFilter ('row2') AND (QualifierFilter s(>=, 'binary:xyz')))AND (TimestampsFilter (123, 456))"}

hbase> scan 'ns1:t1', {FILTER => org.apache.hadoop.hbase.filter.ColumnPaginationFilter.new(1, 0)}

hbase> scan 'ns1:t1', {COLUMNS => ['c1', 'c2'], CACHE_BLOCKS => false}

hbase> scan 'ns1:t1', {RAW => true, VERSIONS => 10}

6. 其他常用命令

HBase 还提供了一些其它的 Shell API，包括 general、ddl、dml、tools、replication 和 security 六组，每组又包括多个 Shell 命令。每组命令和每个命令的用法均可以在 HBase shell 中通过 ***help 'command'***命令查询其用法。

COMMAND GROUPS:

Group name: general

Commands: status, table_help, version, whoami

Group name: ddl

Commands: alter, alter_async, alter_status, create, describe, disable, disable_all, drop, drop_all, enable, enable_all, exists, get_table, is_disabled, is_enabled, list, show_filters

Group name: namespace

Commands: alter_namespace, create_namespace, describe_namespace, drop_namespace, list_namespace,
list_namespace_tables

Group name: dml

Commands: append, count, delete, deleteall, get, get_counter, incr, put, scan, truncate, truncate_preserve

Group name: tools

Commands: assign, balance_switch, balancer, catalogjanitor_enabled, catalogjanitor_run, catalogjanitor_switch,

close_region, compact, flush, hlog_roll, major_compact, merge_region, move, split, trace, unassign, zk_dump

Group name: replication

Commands: add_peer, disable_peer, disable_tablerep, enable_peer, enable_tablerep, list_peers, list_replicated_tables, remove_peer, set_peer_tableCFs, show_peer_tableCFs

Group name: snapshots

Commands: clone_snapshot, delete_snapshot, list_snapshots, rename_snapshot, restore_snapshot, snapshot

Group name: security

Commands: grant, revoke, user_permission

Group name: visibility labels

Commands: add_labels, clear_auths, get_auths, set_auths, set_visibility

Group name: backup/restore

Commands: backup, list_backups, remove_backups, restore, stop_backup, stop_restore

7. 导入数据

从分布式文件系统导入数据操作示例：

(1) 将/opt 下的数据 data1.csv 放到 HDFS 的/tmp 路径，如图 2-4。

```
# su hdfs
$ hdfs dfs -put /opt/data1.csv /tmp
$ hdfs dfs -cat /tmp/data1.csv
$ hdfs dfs -chmod 777 /tmp/data1.csv
```

图2-4 准备数据

```
sh-4.2$ hdfs dfs -put /opt/data1.csv /tmp
sh-4.2$  hdfs dfs -cat /tmp/data1.csv
a,a1
b,b1
c,c1
d,d1
sh-4.2$ hdfs dfs -chmod 777 /tmp/data1.csv
```

(2) 在 HBase 中创建表

```
create 'LOADTEST1','INFO'
```

(3) 执行下面步骤，将 HDFS 数据加载到 HBase 表中的一个临时区域中，如图 2-5。

```
#su hdfs
$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv
-Dimporttsv.columns=HBASE_ROW_KEY,INFO:NAME  -Dimporttsv.separator=,
-Dimporttsv.bulk.output=/tmp/hdfs_output1 LOADTEST1 /tmp/data1.csv
```

说明

- 使用方式：hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.separator='分隔符' -Dimporttsv.columns='HBase 表的列' HBase 表名 csv 文件。
- 通过 importtsv.separator 指定分隔符，否则默认的分隔符是 tab 键。
- 参数 importtsv.separator 仅支持以单个字符作为分隔符，不支持以字符串作为分隔符。
- 如果涉及认证权限管理请参考 3.4 权限访问控制，切换到具有操作权限的用户执行 hbase 操作，需要赋予该用户 hbase、hdfs 目录和 yarn 队列相关操作权限。

图2-5 加载数据成功

```
2019-06-24 09:14:26,126 INFO  [main] mapreduce.Job: Job job_1561181257684_0009 completed successfully
2019-06-24 09:14:26,214 INFO  [main] mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=110
                FILE: Number of bytes written=338403
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=141
                HDFS: Number of bytes written=5096
                HDFS: Number of read operations=8
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=3
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=2893
                Total time spent by all reduces in occupied slots (ms)=5002
                Total time spent by all map tasks (ms)=2893
                Total time spent by all reduce tasks (ms)=2501
                Total vcore-seconds taken by all map tasks=2893
                Total vcore-seconds taken by all reduce tasks=2501
                Total megabyte-seconds taken by all map tasks=2962432
                Total megabyte-seconds taken by all reduce tasks=5122048
        Map-Reduce Framework
                Map input records=2
                Map output records=2
                Map output bytes=100
                Map output materialized bytes=110
                Input split bytes=131
                Combine input records=2
                Combine output records=2
                Reduce input groups=2
                Reduce shuffle bytes=110
                Reduce input records=2
                Reduce output records=2
                Spilled Records=4
```

(4) 将 HDFS 数据导入 HBase 中

$ hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles /tmp/hdfs_output1 LOADTEST1

(5) 查看 HBase 表，可以看到导入数据成功，如图 2-6。

Scan 'LOADTEST1'

图2-6 查看 HBase 表

```
hbase(main):006:0> scan 'LOADTEST1'
ROW                                              COLUMN+CELL
 a                                               column=INFO:NAME, timestamp=1589457227577, value=a1
 b                                               column=INFO:NAME, timestamp=1589457227577, value=b1
 c                                               column=INFO:NAME, timestamp=1589457227577, value=c1
 d                                               column=INFO:NAME, timestamp=1589457227577, value=d1
4 row(s)
Took 0.1852 seconds
hbase(main):007:0>
```

2.3.2 Kerberos 环境

说明

- Kerberos 环境下，若想访问 HBase 并对其执行管理操作，则必须首先进行用户身份认证，认证方式请参见 2.3.2 1. Kerberos 环境下的用户身份认证。
- 本章节操作需要切换至具有 HBase 权限的用户。

1. Kerberos 环境下的用户身份认证

如果大数据集群开启 Kerberos，若想操作 HBase，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 用户的认证文件可在[集群权限/用户管理]页面单击用户列表用户对应的<下载认证文件>按钮进行下载。

HBase 还可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**
    a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的 /etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：
    chown user1 /etc/security/keytabs/user1.keytab
    b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：
    klist -k user1.keytab
    【说明】如图 2-7 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-7 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

su user1

kinit -kt user1.keytab user1@TENANTC.COM

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-8 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$ 
```

（二）组件超级用户身份认证

HBase 可以通过组件超级用户访问，比如 hbase 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 hbase 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 hbase 的认证文件“hbase.headless.keytab”。

【说明】在 HBase Client 节点上，需要将 hbase 的认证文件“hbase.headless.keytab”上传节点的/etc/security/keytabs/目录下进行认证。

(2) 使用 ***klist*** 命令查看 hbase.headless.keytab 的 principal 名称，命令如下：

klist -k hbase.headless.keytab

【说明】如图 2-9所示，红框内容即为 hbase.headless.keytab 的 principal 名称。

图2-9 认证文件的 principal 名称

```
[root@hppnode1 keytabs]# klist -k hbase.headless.keytab
Keytab name: FILE:hbase.headless.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 hbase-testshare@TESTSHARE.COM
   2 hbase-testshare@TESTSHARE.COM
   2 hbase-testshare@TESTSHARE.COM
   2 hbase-testshare@TESTSHARE.COM
   2 hbase-testshare@TESTSHARE.COM
```

(3) 切换至用户 hbase，并执行身份验证的命令如下：

su hbase

kinit -kt hbase.headless.keytab hbase-testshare@TESTSHARE.COM

【说明】其中：hbase.headless.keytab 为 hbase 的认证文件，hbase-testshare@TESTSHARE.COM 为 hbase.headless.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

2. 管理 HBase

当用户身份认证成功后，即可参见 2.3.1 非 Kerberos 环境访问 HBase 并对其执行管理操作。

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

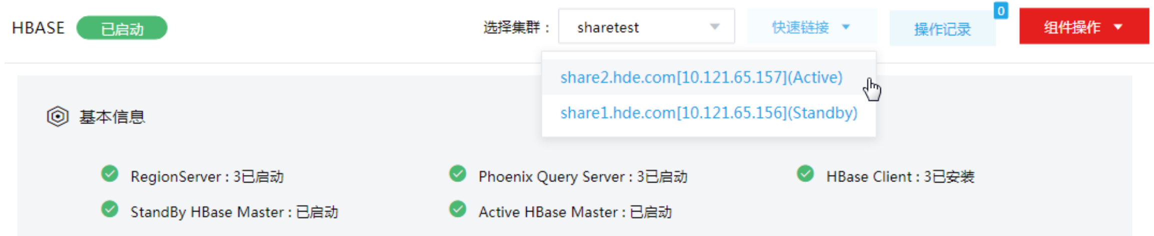
2.4.2 访问 HBase 监控页面

HBase 提供了监控页面，支持在线查看表详情及组件相关详细信息。

(1) 如图 2-10 所示，在 HBase 组件详情页面的右上角[快速链接]的下拉框中，可以获取 HBase 监控管理页面的访问入口信息。

【说明】当集群开启高可用时，HBase 同步开启 HA，此时有两个访问入口，推荐选择 active 状态的链接。

图2-10 HBase 快速链接



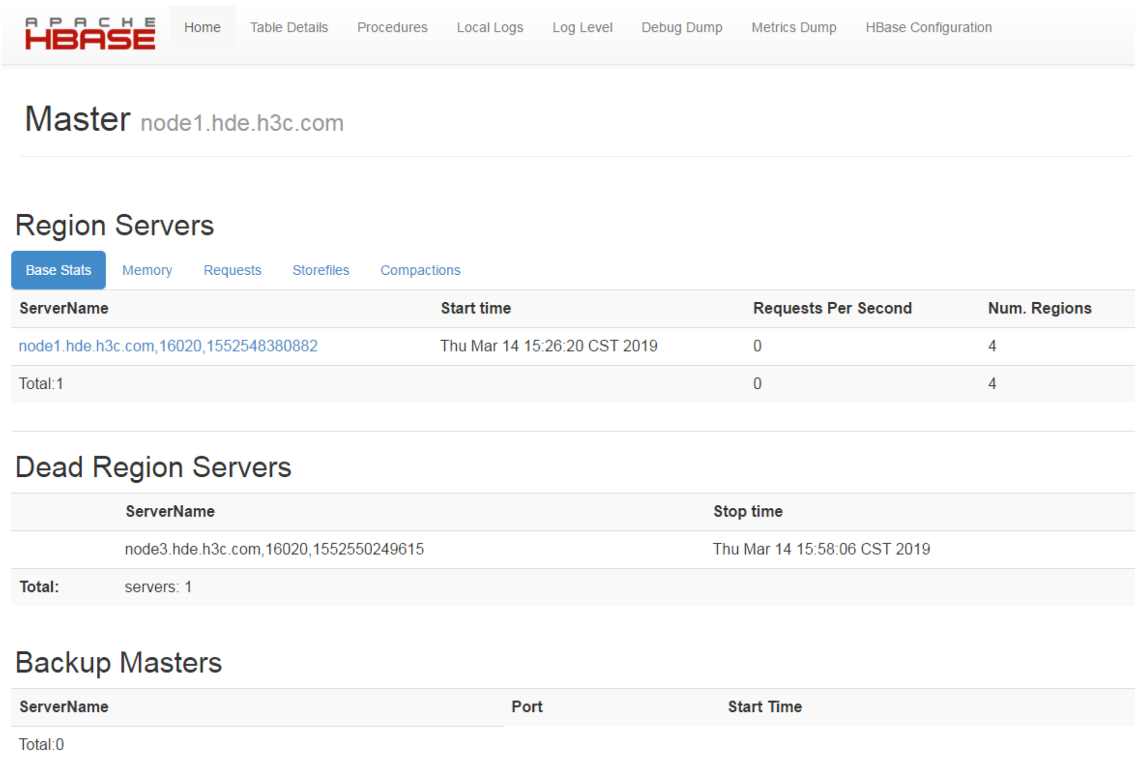
(2) 根据集群是否开启 Kerberos，访问 HBase 快速链接分为两种情况：

- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，可直接跳转访问对应的 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。

(3) 在 HBase 监控管理页面，可查看以下信息：

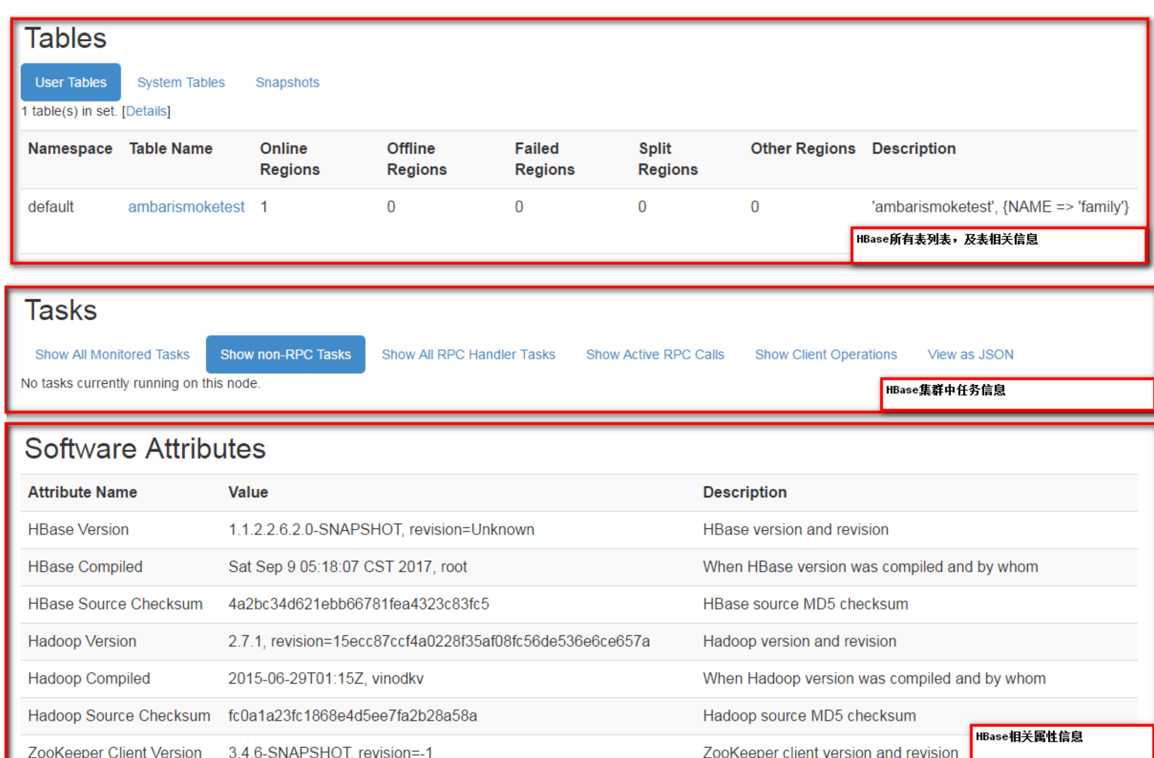
- 如图 2-11 所示，Home 页面展示 Master 节点信息、在线 Region Servers 信息、离线或已关闭 Region Servers 信息、备份 Master 信息等。
- 通过在顶部导航栏切换页签，还可以查看 HBase 中的表详情信息、HBase 集群日志信息及 HBase 配置信息等。

图2-11 HBase 监控页面



- 如图 2-12 所示，Table Details 页面展示 HBase 中的表详情信息。

图2-12 HBase 中的表详情信息



Tables

User Tables　System Tables　Snapshots

1 table(s) in set. [Details]

Namespace	Table Name	Online Regions	Offline Regions	Failed Regions	Split Regions	Other Regions	Description
default	ambarismoketest	1	0	0	0	0	'ambarismoketest', {NAME => 'family'}

Tasks

Show All Monitored Tasks　Show non-RPC Tasks　Show All RPC Handler Tasks　Show Active RPC Calls　Show Client Operations　View as JSON

No tasks currently running on this node.

Software Attributes

Attribute Name	Value	Description
HBase Version	1.1.2.2.6.2.0-SNAPSHOT, revision=Unknown	HBase version and revision
HBase Compiled	Sat Sep 9 05:18:07 CST 2017, root	When HBase version was compiled and by whom
HBase Source Checksum	4a2bc34d621ebb66781fea4323c83fc5	HBase source MD5 checksum
Hadoop Version	2.7.1, revision=15ecc87ccf4a0228f35af08fc56de536e6ce657a	Hadoop version and revision
Hadoop Compiled	2015-06-29T01:15Z, vinodkv	When Hadoop version was compiled and by whom
Hadoop Source Checksum	fc0a1a23fc1868e4d5ee7fa2b28a58a	Hadoop source MD5 checksum
ZooKeeper Client Version	3.4.6-SNAPSHOT, revision=-1	ZooKeeper client version and revision

3 使用指南

3.1 Client下载/安装/使用/卸载

大数据集群提供了下载 HBase Client 的功能。在客户端节点上安装 HBase 的 Client 后，即可直接连接集群中的 HBase，进行组件维护、任务管理等操作。

3.1.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作，租户的 HBase 组件也支持下载 Client），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，会有下载地址的详细提示信息，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在集群管理的左侧导航树中选择[集群列表]，进入集群列表页面，点击集群名称进入集群详情页面，在集群已安装的组件列表中单击 HBase 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-1 所示。

图3-1 下载 Client 窗口

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。

- 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
- 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中的组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。
- 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
- 若勾选同步下载到本地时，则下载的客户端文件会同步保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.1.2 安装 Client

注意

- 安装 Client 的节点必须能与大数据集群中的所有节点网络均互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点上或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置网络连接，仅非 root 用户需要执行此操作，root 用户可跳过此步骤。

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

./install.sh <安装路径> -o

说明

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为/usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

```
./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>
```

3.1.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

  ```
  source bigdata_env
  ```
- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行校验即可直接访问组件。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行认证之后，才可访问组件。关于 Kerberos 环境下，进行用户认证的操作方式请参见 2.3.2 1. Kerberos 环境下的用户身份认证。

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

3.1.4 卸载 Client 客户端

大数据集群重装之后，之前安装的 Client 客户端将不可用。此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 安装目录下执行以下命令：

```
./uninstall.sh
```

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

3.2 HBase集群的扩容

HBase 集群扩容是指在某节点上新增安装 RegionServer。

3.2.1 使用场景

随着业务量的增长，集群存储容量无法满足用户实际使用需求时，需要考虑对 HBase 集群进行扩容。当 HBase 集群出现以下几种情况，可以考虑扩容：

(1) 每个 RegionServer 上分配的的 region 数量超过阈值：

通过 HBase 组件的快速链接，打开 HBase UI 界面，如果看到每个 RegionServer 上均分配了大量的 region，分配的的 region 数量远超 200 个，且大部分 region 存储的数据都在 10GB 以上，此时可以考虑进行扩容。

(2) 每个 RegionServer 的读写压力过大：

登录 HBase 集群中 RegionServer 所在的节点，查看 RegionServer 的读内存使用情况，以及当前操作系统的 IO 情况，如果负载过重，且是 RegionServer 引起的，此时可以考虑进行扩容。

(3) Compaction 和 Flush 操作频繁：

在集群管理平台的日志管理中，查看HBase组件的日志信息，如果HBase集群在频繁地进行 Compaction 和 Flush 操作，此时可以考虑进行扩容。

3.2.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在大数据集群上新增安装 RegionServer 进程。

- 如果集群中有节点没有安装 RegionServer，直接在集群节点中添加 RegionServer 进程。
- 如果集群中所有节点均已安装 RegionServer，进行 RegionServer 扩容前则需要先添加主机。

2. 环境检查

(1) 登录大数据平台管理系统，查看 HBase 组件的状态是否正常。

(2) 进入HBase组件详情页，查看HBase的部署拓扑，确保集群中每个服务的状态正常，HBase Master、RegionServer 处于“已启动”状态，HBase Client 处于“已安装”状态。

3.2.3 扩容约束

- 扩容节点操作系统版本与集群内部版本需保持一致。
- 扩容操作一旦开始，不支持中止。
- 不支持对 HBase 集群中的 Master 服务进行扩容。

3.2.4 扩容影响

- 一旦扩容失败，需要及时将扩容失败的节点剔除。

- 扩容成功后，HBase 集群的数据存储能力会得到增强。

3.2.5 扩容操作指导

注意

若集群中所有节点均已安装 RegionServer，进行 RegionServer 扩容前则需要先添加主机，然后再进行 RegionServer 扩容。如果集群中已有扩容所需主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 HBase 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-2 所示。

  a. 选择进程及主机。

  在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

  b. 部署进程。

  选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

  c. 启动进程。

  部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-2 添加进程

(3) 查看组件数量

RegionServer 扩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 HBase RegionServer 安装数量的变化及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.2.6 RegionServer 扩容后 Balance

注意

本操作建议在业务相对空闲时执行，因为 Balance 操作会导致 region 转移，对性能会有一定的影响。

RegionServer 扩容后数据默认会自动均衡，如果扩容完成后 6 个小时，新节点的 region 数还为 0，则需要检查 Balance 是否开启并手动触发 Balance 操作。手动 Balance 的步骤如下：

(1) 通过 hbase shell 连接 HBase 集群。

(2) 开启 Balance

a. 执行 balancer_enabled，检查集群是否已开启 Balance。

b. 返回结果为 true，则说明 HBase 集群 Balance 已开启。

c. 返回结果为 false，说明 HBase 集群 Balance 未开启，需要手动开启。执行 balanc_switch true，开启 balance。

(3) 通过 HBase 快速链接进入 HBase UI 页面，观察扩容的 RegionServer 的 region 数量是否有变化并等待 region 均衡。

3.2.7 扩容验证

(1) 登录大数据平台管理系统。

(2) 执行 HBase 组件检查，确保 HBase 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 HBase 组件部署拓扑里是否已有新增的扩容节点。

(4) 打开 HBase 快速链接，在 HBase UI 页面查看 RegionServer 的最新状态信息是否符合预期。

3.3 HBase集群的缩容

HBase 集群缩容是指将某节点上已安装的 RegionServer 删除。

3.3.1 使用场景

注意

RegionServer 停止服务后，HMaster 会将自动该 RegionServer 上的 region 转移至其他可用的 RegionServer 节点来保证服务不受影响。

HBase 集群缩容的场景主要为：

- 初始 RegionServer 节点规划不合理。
- HMaster 和 RegionServer 部署在同一个节点上，导致该节点压力过大，需要进行分离。
- 当 RegionServer 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.3.2 缩容前准备

1. 缩容规划

减容只能减少 RegionServer 节点，不能减少 HMaster 节点。

2. 环境检查

(1) 登录大数据平台管理系统，查看 HBase 组件的状态是否正常。

(2) 进入 HBase 组件详情页，查看 HBase 的部署拓扑，确保集群中每个服务的状态正常，HBase Master、Region Server 处于“已启动”状态，HBase Client 处于“已安装”状态。

3.3.3 缩容约束

- 缩容操作一旦开始，不支持中止。
- HBase 集群缩容是指将某节点上已安装的 RegionServer 删除。执行缩容前，请先手动停止缩容节点对应的 HRegionServer 进程，并保证缩容后 HBase 集群 RegionServer 节点个数至少为 3 个。

3.3.4 缩容影响

- HBase 的数据存储能力会降低。
- HBase 的读写性能会降低。
- HBase 缩容后，可能会导致剩下的 RegionServer 节点管理的 region 数量超过阈值，导致集群出现故障。

3.3.5 缩容操作指导

说明

RegionServer 缩容操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行 RegionServer 缩容操作”为例进行说明，在主机详情页面执行 RegionServere 缩容操作，与其类似不再进行说明。

缩容操作步骤如下：

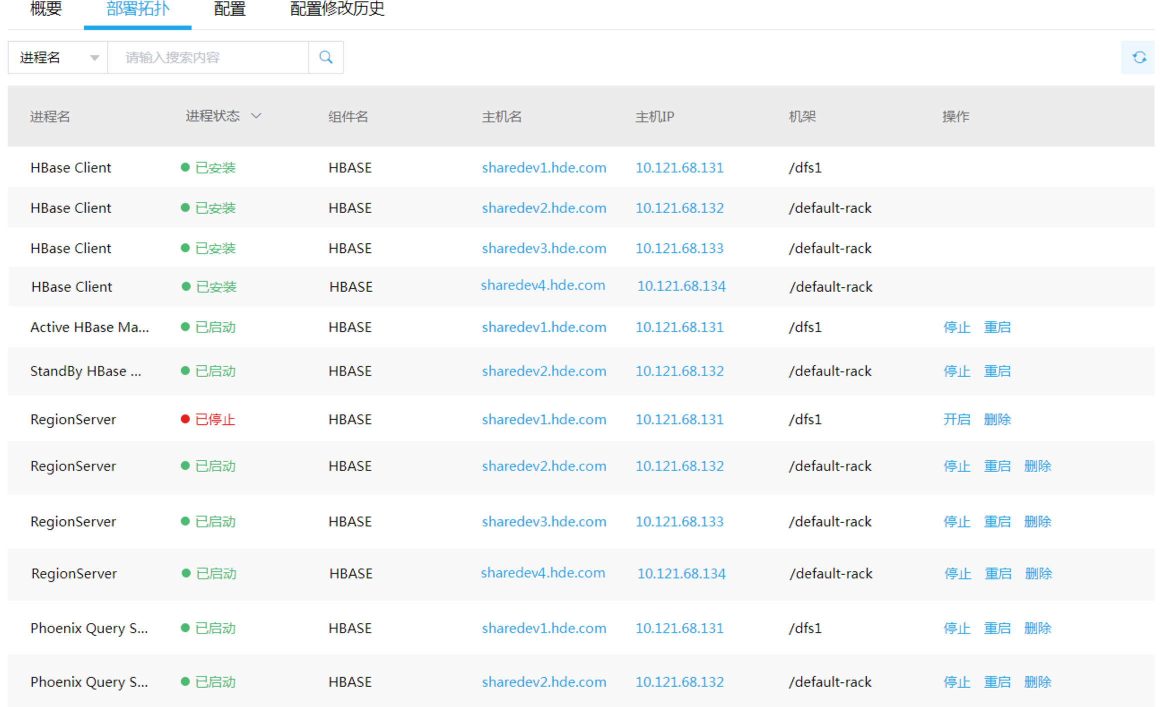
(1) 在 HBase 组件详情页面，选择缩容节点。

(2) 在组件详情页面[部署拓扑]页签下，选择已安装 RegionServer 进程且需要缩容的主机，然后单击该进程右侧操作中的<停止>按钮，停止 RegionServer。

(3) 删除 RegionServer

待 RegionServer 停止后，如图 3-3 所示，在该进程右侧操作中单击<删除>按钮，即可完成 RegionServer 缩容。

图3-3 删除 RegionServer



(4) 查看组件数量

RegionServer 缩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 HBase RegionServer 安装数量的变化及状态。

(5) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.3.6 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 HBase 组件检查，确保 HBase 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 HBase 组件部署拓扑里相关缩容节点是否已经删除。

(4) 打开 HBase 快速链接，在 HBase UI 页面查看 RegionServer 的最新状态信息是否符合预期。

3.4 权限访问控制

注意

集群新建用户的组件权限会因为集群是否开启权限管理功能而有所不同：

- 未开启权限管理时，用户可进行 HBase 表的创建、修改、读、写等操作。
- 开启权限管理后，组件权限需通过[集群权限/角色管理]中的角色分配给用户，用户通过绑定角色进行赋权后，才能对组件执行操作。

权限管理是安全管理的重要组成部分，在开启权限与密钥管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

3.4.1 权限说明

在开启权限管理的集群中，用户对 HBase 表的操作需被赋予 HBase 相关权限后才能执行。

HBase 支持对表、列族和列配置权限（支持使用通配符*模糊适配），权限包括：read、write、create、admin。HBase 操作所需权限对应关系如表 3-1 所示。

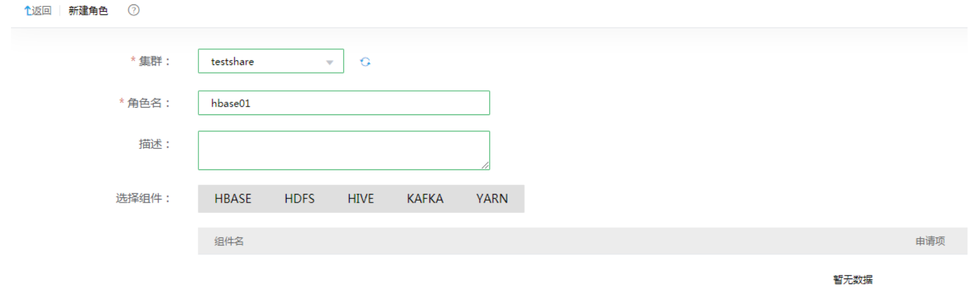
表3-1 HBase 权限说明

组件	权限类型	对应的组件操作
HBase（说明：所有用户默认可执行exists、split操作）	read	数据读取等相关操作，如：get、scan
	write	数据写入等相关操作，如：put、delete
	create	表创建、删除等操作，如：create、disable、enable、drop、compact、major_compact、alter
	admin	命名空间等相关操作，如：create_namespace、drop_namespace、snapshot、clone_snapshot、restore_snapshot、truncate

3.4.2 权限使用操作示例

(1) 在[集群权限/角色管理页面，创建角色 hbase01，不选择任何组件权限，如图 3-4 所示。

图3-4 创建角色



(2) 在[集群权限/用户管理]页面，创建用户 hbaseuser01，并为用户授权角色 hbase01，如图 3-5 所示。

图3-5 对用户授予角色

(3) 使用用户 hbaseuser01 对 HBase 中的表 t 执行 read 相关操作，如图 3-6 所示，执行 scan 表操作，提示没有权限。

图3-6 scan 表

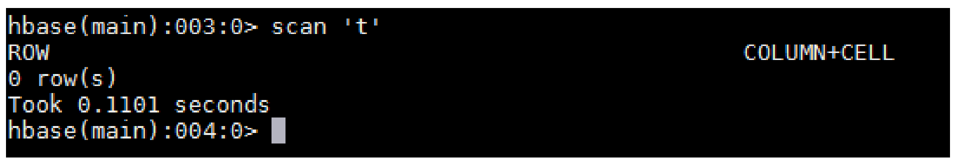
```
hbase(main):003:0> scan 't'
ROW                    COLUMN+CELL
org.apache.hadoop.hbase.security.AccessDeniedException: org.apache.hadoop.hbase.
security.AccessDeniedException: Insufficient permissions for user 'hbaseuser01@T
ESTSHARE.COM',action: scannerOpen, tableName:t, family:t.
```

(4) 在[集群权限/角色管理]页面，修改角色 hbase01 权限设置，授予 hbase01 角色对表 t 的 read 权限，如图 3-7 所示。此时，被授予该角色的用户 hbaseuser01 也拥有了对应权限。

图3-7 角色授权

(5) 使用用户 hbaseuser01 重新执行对表 t 的 read 操作。如图 3-8 所示，任务执行成功。

图3-8 查看表



3.5 租户管理

注意

- 新增租户时，要求系统中必须已成功创建租户模式的 Hadoop 集群。
- 关于租户管理模块功能的配置或使用详情请参见产品在线联机帮助。
- 对于 Hadoop 集群，若集群存储类型为对象存储，则：(1)对于 HBase 资源，执行新增或扩/缩容操作时，不检测存储空间配额大小是否满足实际情况（也不考虑存储层的副本机制）; (2)对于 HBase 资源，执行新增操作时，不支持对 RegionServer 个数进行选择。

多个租户之间共享一套集群，共享网络和集群资源，并且不同租户之间保证资源隔离。

- 新增租户

  普通用户在自己创建的租户集群中申请租户时，无需审批，会直接触发新增租户的操作。普通用户在其他用户创建的租户集群中申请租户时，则需要走流程审批，待审批人审批通过后才能触发新增租户的操作。管理员用户新增租户时，无需审批，会直接触发新增租户的操作。

- 租户管理操作

  普通用户在自己创建的租户集群中执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。普通用户在其他用户创建的租户集群中执行租户续期、资源扩缩容操作时，则需要走流程审批，待审批人审批通过后才能触发相关操作。管理员用户执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。

3.5.1 租户介绍

租户申请的 HBase 资源对应一个或多个限额的命名空间和若干个 RegionServer 组，用户可根据实际需要申请 HBase 组件资源。

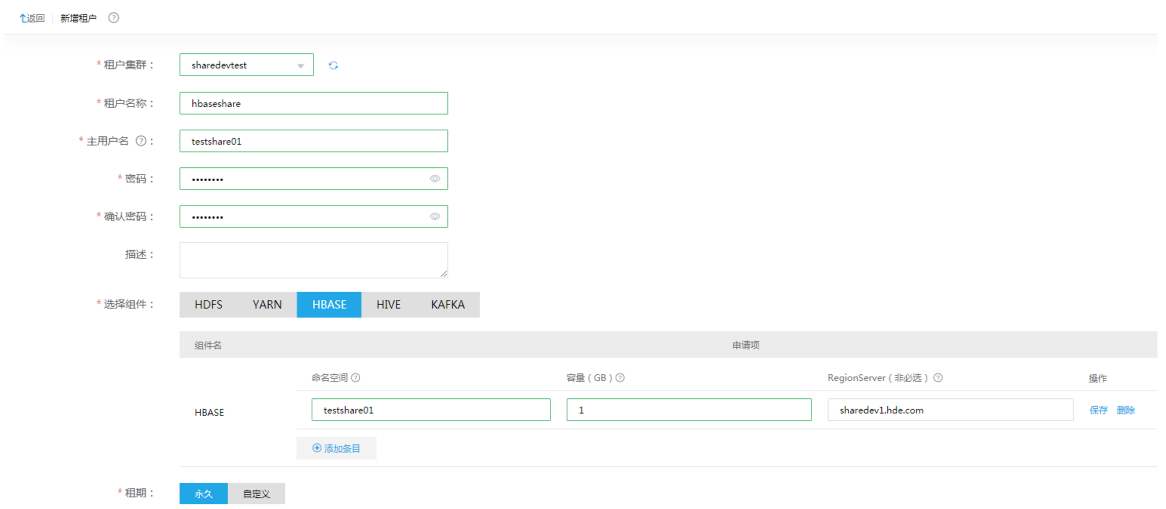
【说明】

- HBase 的存储资源对应 HDFS 资源。
- 【注意】租户申请 HBase 资源时，RegionServer 个数会影响操作性能。但是，系统默认保留一个 RegionServer 供 HBase 自身使用，且 HBase 资源申请成功后建议不要随便修改 RegionServer（会造成数据迁移，风险较大）。

3.5.2 新增租户

(1) 在[集群权限/租户管理]页面，单击<新增租户>按钮，跳转至新增租户页面，如图 3-9 所示。在租户集群 sharedevtest 中新增租户名称 hbaseshare，主用户名 testshare01，并为该租户配置 HBase 组件资源(命名空间 testshare01、1GB 容量、1 个 RegionServer）。

图3-9 新增租户

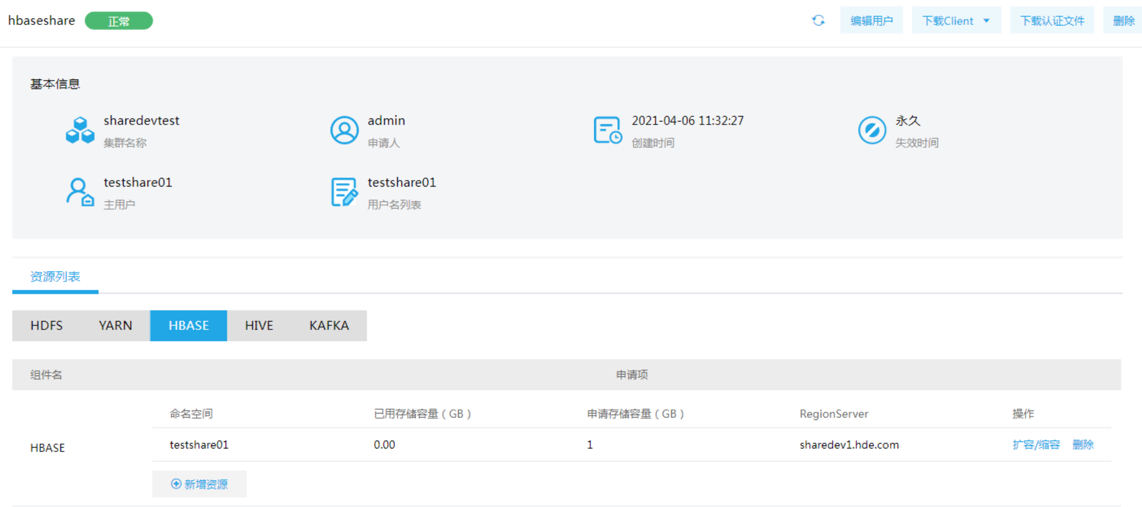


(2) 新增租户成功后，用户可在租户列表查看到已创建的租户，同时可以看到其所属集群、申请人、用户名、创建时间、失效时间等相关信息，如图 3-10 所示。

图3-10 查看租户

(3) 单击租户名称，可查看租户详情，如图 3-11 所示，可以看到对应的命名空间、容量和 RegionServer 信息。用户 testshare01 拥有租户 testshare01 的所有权限，若资源不够/过多，可编辑租户对其进行扩容/缩容。

图3-11 查看租户详情



3.5.3 租户使用操作示例

注意

- 租户集群缺省开启 Kerberos 认证，在使用租户时需要通过该租户的用户对应的认证文件，对租户的用户进行身份认证。租户的用户认证文件的下载在租户列表页面或租户详情页面执行，关于对租户的用户进行认证的方式与集群中用户的身份认证方式相同，详情请参见 2.3.2 1. Kerberos 环境下的用户身份认证。
- HBase 资源的租户包括客户端（即组件 Client），系统提供了下载 HBase 客户端的功能。在客户端节点上安装 HBase 的 Client 后，即可连接租户中的此组件，执行组件维护、任务管理等操作。租户资源组件 Client 的下载在租户列表页面执行，关于租户资源组件 Client 的安装方式与集群组件 Client 相同，详情请参见 3.1 Client 下载/安装/使用/卸载。

(1) 使用 testshare01 用户登录到租户集群，使用 ***list_namespace*** 命令可查看到 HBase 中已创建了 testshare01 命名空间，如图 3-12 所示。

图3-12 查看命名空间

```
hbase(main):001:0> list_namespace
NAMESPACE
default
hbase
testshare01
3 row(s)
Took 0.4930 seconds
```

(2) testshare01 用户可对 testshare01 命名空间进行建表、查询数据等操作，如图 3-13 所示。

- 创建表：create 'testshare01:tab01' ,'cf'。
- 查看表内容：list_namespace_tables 'testshare01'。
- 插入数据：put 'testshare01:tab01', 'r1', 'cf', 'value1'。
- 查询数据：scan 'testshare01:tab01'。

图3-13 操作 HBase

```
hbase(main):002:0> create 'testshare01:tab01' ,'cf'
Created table testshare01:tab01
Took 6.5888 seconds
=> Hbase::Table - testshare01:tab01
hbase(main):003:0> list_namespace_tables 'testshare01'
TABLE
tab01
1 row(s)
Took 0.1563 seconds
=> ["tab01"]
hbase(main):004:0> put 'testshare01:tab01', 'r1', 'cf', 'value1'
Took 1.4275 seconds
hbase(main):005:0> scan 'testshare01:tab01'
ROW                                        COLUMN+CELL
 r1                                        column=cf:, timestamp=1586327400569, value=value1
1 row(s)
Took 0.7124 seconds
hbase(main):006:0>
```

3.6 备份恢复

备份恢复提供大数据平台跨集群之间的数据同步功能，支持对 HBase 组件中的指定数据进行同步备份，以保证数据内容不丢失。

备份恢复功能主要是通过创建同步任务实现集群间的数据同步能力。使用同步任务功能，可以为集群中的数据提供同步能力，以满足日常数据备份的需求，保证系统或机器故障时的数据不丢失。

3.6.1 新建HBase同步任务

注意

- 源集群为同步任务的数据输出集群（一般指新建同步任务的本集群）；目的集群为同步任务的数据输入集群。
- 当前版本中，仅当集群存储类型为HDFS时（且源集群和目的集群的存储类型必须均为HDFS），才支持创建同步任务。
- 新建同步任务时，要求源集群与目的集群的集群类型、集群模式均相同。
- 新建同步任务时，要求源集群与目的集群的安全管理策略相同，即同时开启Kerberos认证或同时都没有开启Kerberos认证。
- 新建同步任务时，要求源集群与目的集群的集群名称、节点主机名不相同，否则配置跨集群互信时可能出错。
- HBase同步任务包括全量任务和增量任务两种，其中：全量任务用以同步历史数据，为一次性任务，任务启动后将一直运行至结束（可执行中途停止操作，但停止后再启动该任务将被重新执行）。增量任务用以同步数据变更，为持续运行任务，即任务启动后将一直运行，若想要停止只能手动触发。
- HBase同步任务的执行集群只能是源集群（即本集群）。
- 若待同步的资源中存在数据，请务必先执行HBase全量同步任务完成历史数据的同步后，再执行HBase增量任务，否则可能会导致数据同步失败；若待同步资源中不存在历史数据，则直接执行HBase增量任务即可实现数据同步。
- HBase增量同步任务第一次启动时，仅能同步任务启动后的数据，历史数据不会同步。历史数据若需要同步，有两种方式：一是通过创建全量同步任务，二是通过手动操作（比如通过bulkload方式将历史数据导入到目的集群，详情请参见3.6.4 HBase历史数据同步。
- HBase增量同步任务启动后若手动触发了任务停止，则当任务重新启动时，可能因触发MemStore Flush导致停止期间的数据变更无法全部同步到目的集群中，造成数据丢失（此时建议重新做历史数据的同步）。
- 若变更HBase数据时不通过WAL，则这部分数据变更无法通过HBase同步任务进行同步。
- HBase增量同步任务运行中不能启动相同资源的全量同步任务，否则会造成数据丢失。若不小心进行了上述操作，请删除增量任务并重新创建，然后先启动全量同步任务，待全量同步任务运行结束后，再启动新创建的增量同步任务。
- HBase同步任务启动后，若在源集群上执行了修改表结构的操作（如增加列族），则：1)对于全量任务，会提示"源集群和目的集群存在表结构不同，备份失败"的异常，此时需要在目的集群HBase中修改表结构与源集群保持一致，然后重新启动任务，新增列族的数据即可同步至目的集群（历史数据只能通过全量备份实现数据同步）；2)对于增量任务，在目的集群HBase中修改表结构与源集群保持一致后，需先停止再重新启动增量备份任务，此后源集群新增的数据才可同步至目的集群。
- HBase全量同步任务运行过程中，需暂停相关业务，否则同步期间的数据可能会丢失。

集群在使用过程中，根据实际需要，可新建HBase同步任务，将源集群中的HBase某些命名空间或表中的数据拷贝到目的集群中。

1. 前提条件

- 新建 HBase 同步任务前，需要对源集群与目的集群配置跨集群互信。配置方法详情请参见3.6.2 源集群和目的集群配置互信。
- 运行 HDFS 同步任务前，需要根据集群的配置情况进行相关配置修改，详情请参见 3.6.3 HBase 同步任务相关配置。

2. 新建 HBase 同步任务

新建 HBase 同步任务的前提条件准备完成后，即可开始创建 HBase 同步任务。步骤如下：

(1) 访问备份恢复。在备份恢复页面，选择[同步任务]页签，单击<新建同步任务>按钮，进入新建同步任务页面，并在页面中选择“HBase”组件，如图 3-14 所示。

(2) 根据提示配置对应参数项的值，关于各参数项配置详情请参见产品在线联机帮助。

(3) 相关信息配置完成后，若单击<新建>按钮可成功新建 HBase 同步任务，此时任务未启动（需要手动启动）；若单击<新建并启动>按钮则可完成 HBase 同步任务的新建且立刻启动该任务。

图3-14 新建 HBase 同步任务

3.6.2 源集群和目的集群配置互信

注意

不同集群之间可通过组件同步任务进行数据同步，但创建同步任务之前必须配置源集群和目的集群互信。

示例集群如下：

- 源集群
  - 开启 Kerberos 认证集群 clusterA，kerberos realm 为 CLUSTERA.COM。
- 目的集群
  - 开启 Kerberos 认证集群 clusterB，kerberos realm 为 CLUSTERB.COM。

1. 开启 kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

(3) 修改源集群和目的集群中所有节点的/etc/krb5.conf 文件，要求：

- 修改 realms，要求同时包含源集群和目的集群的 realms 内容（互相拷贝即可），如图 3-15 所示。
- 修改 domain_realm，要求同时包含源集群和目的集群的 domain_realm 内容。
  - 当源集群与目的集群的主机名后缀相同时，需要将 domain_realm 内容修改为“集群中各节点主机名=对应的 realms 名”。示例：源集群和备集群的主机名后缀均为.hde.com，则配置如图 3-16 所示。
  - 当源集群与目的集群的主机名后缀不同时，则可直接将两个集群的 domain_realm 中内容合并（不需要修改，直接互相拷贝即可）。示例：源集群的主机名后缀为.hde.com，目的集群的主机名后缀为.hadoop.com，则配置如图 3-17 所示。

图3-15 realms 修改后示例

```
[realms]
    CLUSTERA.COM = {
        kdc = clustera1.hde.com
        admin_server = clustera1.hde.com
        database_module = openldap_ldapconf
    }
    CLUSTERB.COM = {
        kdc = clusterb1.hde.com
        admin_server = clusterb1.hde.com
        database_module = openldap_ldapconf
    }
```

图3-16 domain_realm 修改后示例（源集群和目的集群主机名后缀相同）

```
[domain_realm]
   clustera1.hde.com=CLUSTERA.COM
   clustera2.hde.com=CLUSTERA.COM
   clustera3.hde.com=CLUSTERA.COM
   clusterb1.hde.com=CLUSTERB.COM
   clusterb2.hde.com=CLUSTERB.COM
   clusterb3.hde.com=CLUSTERB.COM
```

图3-17 domain_realm 修改后示例（源集群和目的集群主机名后缀不同）

```
[domain_realm]
    .hde.com = CLUSTERA.COM
    hde.com = CLUSTERA.COM
    .hadoop.com = CLUSTERB.COM
    hadoop.com = CLUSTERB.COM
```

(4) 在源集群和目的集群的 Master 节点上，执行添加 principal 操作。

a. 在源集群 Master 节点上，分别执行以下两条命令，以添加 principal（命令执行后，需要分别输入密码）：

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '

b. 在目的集群 Master 节点上，分别执行以下两条命令，以添加 principal（命令执行后，需要分别输入密码）：

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '

【注意】

- CLUSTERA.COM 和 CLUSTERB.COM 分别为源集群和目的集群的域名，请根据实际情况进行修改。添加 principal 时需确保两个集群输入的密码相同（即上述四条命令运行后输入的密码均相同），且密码要求至少 8 位，否则会提示密码太短导致设置无效。
- 若添加 principal 时输入的密码不同，可在源集群和目的集群上进行删除，然后重新执行第 4 步添加 principal 的操作。删除命令如下：

  kadmin.local -q ' delprinc krbtgt/CLUSTERA.COM@CLUSTERB.COM'

  kadmin.local -q ' delprinc krbtgt/CLUSTERB.COM@CLUSTERA.COM'

(5) 源集群与目的集群互信配置完成后，可登录目的集群进行校验。校验方式示例：在目的集群后台切换至集群超级用户，执行命令 hdfs dfs -ls hdfs://<源集群 Active NameNode IP 地址>:8020/，查看源集群的 HDFS 是否可正常访问，若能正常访问则表示源集群与目的集群的互信配置成功。

2. 不开 kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

3.6.3 HBase 同步任务相关配置

1. 开启 Kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，创建 HBase 同步任务之前，需进行如下配置修改：

- 需对源集群进行配置修改，说明如下：

  在源集群的[集群管理/集群详情/业务组件/HDFS 组件详情]页面，修改 HDFS 配置项 hadoop.security.auth_to_local 的值。要求在末尾 DEFAULT 之前增加内容“ RULE:[1:$1@$0](hbase-clusterB@CLUSTERB.COM)s/.*/hbase/ ”，其中 clusterB 为目的集群的集群名称，CLUSTERB.COM 为目的集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启相关组件。

  【注意】若 HDFS 配置项 hadoop.security.auth_to_local 的值中已存在 HDFS 同步任务需要的规则（即“ RULE:[1:$1@$0](.*@CLUSTERB.COM)s/@.*// ”），则再添加 HBase 同步任务需要的规则时应该在其上方。

- 对目的集群进行配置修改，说明如下：

  在目的集群的[集群管理/集群详情/业务组件/HDFS 组件详情]页面，修改 HDFS 配置项 hadoop.security.auth_to_local 的值。要求在末尾 DEFAULT 之前增加内容“RULE:[1@1@$0](hbase-clusterA@CLUSTERA.COM)s/.*/hbase/”，其中 clusterA 为源集群的集群名称，CLUSTERA.COM 为源集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启相关组件。

  【注意】若 HDFS 配置项 hadoop.security.auth_to_local 的值中已存在 HDFS 同步任务需要的规则（即“RULE:[1@1@$0](.*@CLUSTERA.COM)s/@.*//”），则再添加 HBase 同步任务需要的规则时应该在其上方。

2. 不开 Kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则在集群之间运行 HBase 同步任务时，不需要进行相关配置的修改。

3.6.4 HBase 历史数据同步

HBase 同步任务第一次启动时，仅能同步任务启动后的数据，历史数据不同步，建议启动 HBase 同步任务之前先完成历史数据的同步。历史数据同步有两种方式：一是通过创建全量同步任务，二是通过 bulkload 方式手动将历史数据导入到目的集群。

通过 bulkload 方式手动将历史数据导入到目的集群的步骤如下：

(1) 在源集群中，对需要拷贝的表进行 snapshot 快照，并记录快照时间。

(2) 在目的集群中，执行 ExportSnapshot 命令拷贝源集群的快照和数据。

(3) 等待源集群中的历史数据全部传入目的集群后，所有历史数据都在 archive 文件下。此时在目的集群上使用 bulkload 命令即可将 archive 文件下的数据（region）导入在线业务表。

3.7 数据迁移

3.7.1 数据迁移常用方案

HBase 数据迁移是很常见的操作，主要的迁移方式主要分为以下几类。

1. copyTable

copyTable 属于 HBase 数据迁移的工具之一，以表级别进行数据迁移。copyTable 的本质是利用 MapReduce 进行同步的，它是利用 MapReduce 去 scan 原表的数据，然后把 scan 出来的数据写入到目标集群的表。这种方式有很多局限，如一个表数据量达到 T 级，同时又在读写的情况下，全量 scan 表无疑会对集群性能造成影响。

copyTable 支持设定需要复制的表的时间范围、cell 的版本，也可以指定列族，设定从集群的地址、起始/结束行键等。

2. export/import 方式

此方式与 CopyTable 类似，主要是将 HBase 表数据转换成 Sequence File 并 dump 到 HDFS，也涉及 Scan 表数据，与 CopyTable 相比，还多支持不同版本数据的拷贝，同时它拷贝时不是将 HBase 数据直接 Put 到目标集群表中，而是先转换成文件，把文件同步到目标集群后再通过 Import 到线上表。主要有两个阶段：

(1) Export 阶段：将原集群表数据 Scan 并转换成 Sequence File 到 HDFS 上，因为 Export 也是依赖于 MapReduce 的，如果用到独立的 MapReduce 集群，只要保证在 MapReduce 集群上关于 HBase 的配置和原集群一样且能和原集群策略打通(master®ionserver 策略），就可直接用 Export 命令，如果没有独立 MapReduce 集群，则只能在 HBase 集群上开启 MapReduce，若需要同步多个版本数据，可以指定 versions 参数，否则默认同步最新版本的数据，还可以指定数据起始结束时间。

(2) Import 阶段：将原集群 Export 出的 SequenceFile 导到目标集群对应表。

3. snapshot

SnapShot 是通过快照来实现 HBase 数据一致性迁移，这种方式在创建快照的时候会创建文件的引用指针，在传输的时候，通过 MapReduce 直接拷贝底层 HDFS 文件，因此创建非常快，效率很高，并且对线上冲击很小，能很好的保证数据的一致性。

SnapShot 是目前比较推荐的迁移数据方式。

4. Replication

Replication 和 MySQL 同步的方案比较类似，是通过同步日志(WAL 日志)的方式实现 HBase 数据的增量同步，这个方案主要是基于增量数据的同步，并且主从集群版本相同，启用 replication 功能需要重启集群。

5. distcp

DistCp（分布式拷贝）是 Hadoop 用于大规模集群内部和集群之间拷贝的工具。它使用 Map/Reduce 实现文件分发、错误处理、恢复以及报告生成。它把文件和目录的列表作为 map 任务的输入，每个任务会完成源列表中部分文件的拷贝。

其拷贝本质过程是启动一个 MapReduce 作业，不过 DisctCp 只有 map，没有 reducer。在拷贝时，由于要保证文件块的有序性，转换的最小粒度是一个文件，而不像其它 MapReduce 作业一样可以

把文件拆分成多个块启动多个 map 并行处理。如果同时要拷贝多个文件，DisctCp 会将文件分配给多个 map，每个文件单独一个 map 任务。

3.7.2 数据迁移案例

根据不同的数据迁移场景、迁移数据量、迁移环境及迁移需求，需结合每个工具的优点及缺点混合使用，给出最有效的数据迁移方案。

下面列出两个常用使用场景案例。

1. 数据迁移场景一

场景描述：将 HBase 集群 A 表数据迁往 HBase 集群 B 中，且该表已停止数据写操作，集群 B 中不存在该表，两集群均未开启 Kerberos。

操作步骤如下：

(1) 进入集群 A，执行 hbase shell；

(2) 执行 snapshot 操作：snapshot '表名','快照名'；

(3) 通过 ExportSnapshot 将快照数据迁移到集群 B 中，权限开启情况下需在 hbase 用户下执行命令，该命令在目的集群 B 执行，命令示例如下：

hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot 快照名 -copy-from hdfs://集群 A 主节点:8020/apps/hbase/data/ -copy-to hdfs:// 集群 B 主节点:8020/ apps/hbase/data/

(4) 进入集群 B，执行 hbase shell；

(5) 恢复快照，命令如下：restore_snapshot '快照名'。

2. 数据迁移场景二

场景描述：将 HBase 集群 A 表数据迁往 HBase 集群 B 中，且该表已停止数据读写操作，集群 B 中存在该表且该表仍进行读写操作，两个集群表 region 分布不同，集群 B 开启 Kerberos。

操作步骤如下：

(1) 进入集群 A，执行 hbase shell；

(2) 执行 snapshot 操作，snapshot '表名'，'快照名'；

(3) 通过 ExportSnapshot 将快照数据迁移到集群 B 中，权限开启情况下需在 hbase 用户下执行命令，该命令在目的集群 B 执行，命令示例如下：

hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot 快照名 -copy-from hdfs://集群 A 主节点:8020/apps/hbase/data/ -copy-to hdfs:// 集群 B 主节点:8020/ apps/hbase/data/

(4) 此时所有的历史数据在 archive 文件下，我们只需要将其下的 HFiles 导入到现在运行的业务表中即可，在 B 集群执行 bulkload 命令，如下所示：

hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles
/apps/hbase/data/archive/data/default/表名 /region 名　表名

注：需要循环将/apps/hbase/data/archive/data/default 下该表所对应的所有 region 下的数据 bulkload 到集群 B 的对应表中。

3.8 HBCK工具使用指南

在实际生产环境中，会因为各种各样的因素（如用户的误操作、HDFS 故障、HBase 集群节点掉电等），可能使 HBase 集群出现故障，导致集群不可用，常见的故障有 Region 出现 RIT 问题、Procedure 出现卡死情况等。为了解决这些问题，可以使用 HBCK 工具来进行修复。

当前版本的 HBase 提供 HBCK 工具包，该工具包的存放位置是：/usr/hdp/3.0.1.0-187/hbase/tools/hbase-hbck2-1.1.0-SNAPSHOT.jar。

3.8.1 Procedure 简介

由于 HBCK 工具通常是通过修复 Procedure 来解决 HBase 集群故障的，因此在介绍 HBCK 工具的功能之前有必要先介绍 Procedure。

在 HBase2 集群中，几乎所有的集群操作都是通过 Procedure 来进行的，HBase 在执行 Procedure 的时候可能会存在一些不正常的情况，HBCK 的作用就是修复不正常的 Procedure。在 HBase2 中，一个 Procedure 是由一系列的操作组成的，可以将一个 Procedure 看作一次事务，Procedure 完成之后，只存在两种状态，要么执行成功，要么失败回滚，并不存在中间状态。

一次 Procedure 包含了多个操作，每一个操作的执行都会以 log 的形式持久化在 WALs 中，这样可以保证即使 HBase 集群断电重启，也可以基于保存的日志来恢复之前的状态并继续执行。

由于 Procedure 在执行过程中，包含若干个步骤，在此过程中，可能会持有多个锁对象，因此在处理某一步骤的时候，可能由于异常原因导致锁无法释放，从而出现卡死情况，导致 Procedure 迟迟不能结束。

示例说明：HBase 的 assign region 操作是一个 Procedure，在此过程中，Procedure 会对这个 region 进行上锁操作，上锁的目的是为了防止在此过程中有其他用户去这个 region 执行操作，如 unassign region、drop 该 region 所对应的表，从而导致 HBase 集群中数据出现不一致情况。但是如果在 assign region 的时候，由于一些异常情况，导致迟迟无法分配完成，更加糟糕的是如果这个 region 恰恰就属于 meta 表，那么就会出现 HBase 的 HMaster 服务一直处于正在初始化状态，从而导致整个 HBase 集群不可用，如图 3-18 所示。

图3-18 HBase 故障示例

```
Took 8.2052 seconds
hbase(main):013:0> create_namespace 'ns1'

ERROR: org.apache.hadoop.hbase.PleaseHoldException: Master is initializing
        at org.apache.hadoop.hbase.master.HMaster.checkInitialized(HMaster.java:2976)
        at org.apache.hadoop.hbase.master.HMaster.createNamespace(HMaster.java:3152)
        at org.apache.hadoop.hbase.master.MasterRpcServices.createNamespace(MasterRpcServices.java:614)
        at org.apache.hadoop.hbase.shaded.protobuf.generated.MasterProtos$MasterService$2.callBlockingMethod(MasterProtos.java)
        at org.apache.hadoop.hbase.ipc.RpcServer.call(RpcServer.java:413)
        at org.apache.hadoop.hbase.ipc.CallRunner.run(CallRunner.java:130)
        at org.apache.hadoop.hbase.ipc.RpcExecutor$Handler.run(RpcExecutor.java:324)
        at org.apache.hadoop.hbase.ipc.RpcExecutor$Handler.run(RpcExecutor.java:304)

For usage try 'help "create_namespace"'
```

Procedure 在执行过程中，可能会持有两类锁：

- IdLock:Procedure 层级的锁，保证一个 Procedure 不会被多个线程同时执行。
- 资源锁：对 HBase 内部的资源进行加锁，不同的 Procedure 加锁的粒度是不同的，目前有 region、table、namespace、region server 层级。

3.8.2 HBCK 工具命令格式

HBCK 工具包的使用方式如下：

hbase hbck -j <HBCK 工具包路径> <命令参数>

3.8.3 HBCK 功能

1. bypass

bypass 命令可以将卡住的一个或多个 Procedure 释放掉。命令如下：

hbase hbck -j <HBCK jar 包路径> -skip bypass [options] <procedure PID>

其中 Procedure PID 可以在 HBase Master UI 中查看。

[options]的值可以是：

- -o，在执行 bypass 之前会先尝试获取 IdLock 对象，如果 Procedure 还在运行就可能会导致无法获取到 IdLock，从而导致执行超时返回 null，一旦设置了-o 参数，即使拿不到 IdLock 对象，也会将 Procedure 的 bypass flag 设置为 true。
- -r，一个 Procedure 可能会拥有子 Procedure，一旦设置了-r 那么就会递归 bypass，将该 Procedure 的所有子任务也 bypass 掉。
- –w，设置等待获取 idlock 的超时时间，默认是 1ms。

2. assign

该命令的作用就是将 region 再次随机分配到别的机器上，返回值如果是-1 表示执行失败，其他值则是 Procedure 的 PID，表示命令执行成功。命令如下：

hbase hbck -j <HBCK jar 包路径> -skip assigns [options] <regionname>

[options]的值可以是：

- -o ，此处的-o 参数和 bypass 的-o 参数意义不同，assign 命令是新创建一个 Procedure，因此与其他 procedure 的 IdLock 不冲突，但是如果其他的 procedure 是因为资源锁而卡住的话，就会影响 assign 的 Procedure，-o 的作用就是释放掉资源锁，防止因锁资源而出现卡死情况。

3. unassign

该命令可以将一个或者多个 region unassign 掉。该命令的返回值为-1 表示执行命令失败，返回值为其他时表示命令执行成功，返回值为创建的 Procedure PID。

hbase hbck -j <HBCK jar 包路径> -skip unassigns [options] <regionname>

[options]的值可以是：

- -o，此处的-o 和 assign 的-o 参数作用一样，不再赘述。

4. setTableState

HbaseTable 可能的状态有 ENABLE、DISABLE、DISABLING、ENABLING。

当 Table 的状态和所有的 region 状态不一致的时候可以使用此命令进行修复。

hbase hbck -j <HBCK jar 包路径> -skip setTableState <tablename> <STATE>

5. replication

在副本待删除队列中查找指定表所属的副本，并且删除。

hbase hbck -j <HBCK jar 包路径> -skip replication <tablename>

6. filesystem

检查指定表的 HFile 文件。

hbase hbck -j <HBCK jar 包路径> -skip filesystem <tablename>

7. reportMissingRegionsInMeta

检查指定表或者指定命名空间是否存在丢失 region。

hbase hbck -j <HBCK jar 包路径> -skip reportMissingRegionsInMeta <namespace|namespace:tablename>

8. addFsRegionsMissingInMeta

这个和 reportMissingRegionsInMeta 功能结合使用，当执行 reportMissingRegionsInMeta 命令发现有 missing 的 regions，可以执行此命令进行修复，能够修复成功的前提是 region 虽然 missing，但是所属的 HFile 在 HDFS 上还存在。

hbase hbck -j <HBCK jar 包路径> -skip addFsRegionsMissingInMeta <namespace|namespace:tablename>

9. setRegionState

此命令可以将指定的 region 设定到指定状态，此命令慎用，用户手动执行后，可能会导致 region 出现数据不一致情况。

hbase hbck -j <HBCK jar 包路径> -skip setRegionState <regionname> <state>

这里需要注意的是，通过此功能仅仅是更新了 HBase META 表中的对应 region 的状态值。

region 可能存在的状态有：OFFLINE，OPENING，OPEN，CLOSING，CLOSED，SPLITTING，SPLIT，FAILED_OPEN，FAILED_CLOSE，MERGING，MERGED，SPLITTING_NEW，MERGING_NEW，BNORMALLY_CLOSED。

4 开发指南

4.1 常用API示例

4.1.1 开发环境准备及注意事项

1. 开发所使用jar包

开发所使用jar包尽量从集群环境中获取，避免开发过程中出现包版本号不一致情况。

2. 开发所使用配置文件

HBase 开发中所使用的 hbase-site.xml、hdfs-site.xml、core-site.xml、Kerberos 认证所需要的krb5.conf及keytab文件等相关配置文件，应从相应的HBase集群中获取。

3. OpenJDK版本

OpenJDK版本为1.8。

4. 注意事项

- 保证开发环境与连接集群时间同步，避免出现连接不上、连接超时等问题。
- 需将连接集群/etc/hosts文件内容拷贝到开发环境的hosts中。
- 切记调用结束时注意关闭HTable、connection等连接。

4.1.2 非Kerberos环境Java开发样例

说明

HBase调用结束时注意关闭HTable、connection、ResultScanner等连接。

1. 创建HBaseConfigurtation

- 方式一：通过配置文件创建，代码示例如下：

```
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
```

- 方式二：给出必要配置项进行创建，代码示例如下：

```
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.set("hbase.zookeeper.quorum","101.8.89.150,101.8.89.151,101.8.89.152");
hbaseconf.set("hbase.zookeeper.property.clientPort","2181");
hbaseconf.set("zookeeper.znode.parent","/hbase-unsecure");
```

注意事项：

- 方式一与方式二相比尽量选取方式一，可以避免不必要的问题出现。

- 方式二中配置的 Zookeeper 地址为 HBase 集群所使用的 Zookeeper 地址，为了保证容灾，要配置 HBase 集群的所有 Zookeeper 地址；配置项“zookeeper.znode.parent”及"hbase.zookeeper.property.clientPort"的配置值与连接的 HBase 集群保持一致。

2. 创建 Connection

HBase 通过 ConnectionFactory.createConnection(configuration)方法创建 Connection 对象。传递的参数为上一步创建的 Configuration。

Connection 封装了底层与各 RegionServer 的连接以及与 ZooKeeper 的连接。Connection 通过 ConnectionFactory 类实例化。创建 Connection 是重量级操作，Connection 类是线程安全的，因此，多个客户端线程可以共享一个 Connection。

典型的用法，一个客户端程序共享一个单独的 Connection，每一个线程获取自己的 Admin 或 Table 实例，然后调用 Admin 对象或 Table 对象提供的操作接口。不建议缓存或者池化 Table、Admin。Connection 的生命周期由调用者维护，调用者通过调用 close()，释放资源。

具体示例代码如下：

```
Connection conn = null;
try {
  //根据 HBaseconf 创建 connection
  conn = ConnectionFactory.createConnection(hbaseconf);
  Admin admin = conn.getAdmin();
  TableName[] names = admin.listTableNames();
  for(TableName name : names){
      LOG.info("表名:",name.getNameAsString());
  }
} catch (IOException e) {
  LOG.error("list table failed " ,e);
}finally{
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Failed to close connection ", e);
      }
  }
}
```

3. 创建表

示例代码如下：

```
Connection conn = null;
Admin admin = null;
TableName tableName = TableName.valueOf("table-test");
try {
  conn = ConnectionFactory.createConnection(hbaseconf);
  //创建表描述符
  HTableDescriptor htd = new HTableDescriptor(tableName);
  //创建列族描述符
  HColumnDescriptor hcd = new HColumnDescriptor("info");
  // 设置压缩方法，HBase 提供了 GZ 和 SNAPPY 两种压缩方法，建议使用 SNAPPY
```

```
  hcd.setCompressionType(Compression.Algorithm.SNAPPY);
  htd.addFamily(hcd);
  admin = conn.getAdmin();
  if (!admin.tableExists(tableName)) {
      LOG.info("Creating table...");
      //创建表
      admin.createTable(htd);
      LOG.info("Table created successfully.");
  } else {
      LOG.warn("table already exists");
  }

} catch (IOException e) {
  LOG.error("Create table failed " ,e);
}finally{
  if (admin != null) {
      try {
          admin.close();
      } catch (IOException e) {
          LOG.error("Failed to close admin " ,e);
      }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Failed to close connection ", e);
      }
  }
}
```

注意事项：

- 可以设置列族的压缩方式，代码示例如下：

  //设置编码算法，HBase 提供了 DIFF，FAST_DIFF，PREFIX 三种编码算法。

  hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);

  //设置文件压缩方式，HBase 默认提供了 GZ 和 SNAPPY 两种压缩算法

  //其中 GZ 的压缩率高，但压缩和解压性能低，适用于冷数据

  //SNAPPY 压缩率低，但压缩解压性能高，适用于热数据

  //建议默认开启 SNAPPY 压缩

  hcd.setCompressionType(Compression.Algorithm.SNAPPY);

- 可以通过指定起始和结束 RowKey，或者通过 RowKey 数组预分 Region 两种方式建表，建表时进行 region 预分可以有效的提升表的读写性能，代码片段如下：

  //创建一个预划分 region 的表

  byte[][] splits = new byte[4][] ;

  splits[0] = Bytes.toBytes("A");

  splits[1] = Bytes.toBytes("H");

```
splits[2] = Bytes.toBytes("O");
splits[3] = Bytes.toBytes("U");
admin.createTable(htd, splits);
```

4. 删除表

注意

删除表之前必须 disable 要删除的表。

代码示例如下：

```
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
Connection conn = null;
TableName tableName = TableName.valueOf("table-test");
Admin admin = null;
try {
  conn = ConnectionFactory.createConnection(hbaseconf);
  admin = conn.getAdmin();
  if (admin.tableExists(tableName)) {
      // disable 需要删除的表
      admin.disableTable(tableName);
      // 删除表
      admin.deleteTable(tableName);//注[1]
  }
  LOG.info("Drop table successfully.");
} catch (IOException e) {
  LOG.error("Drop table failed " ,e);
} finally {
  if (admin != null) {
      try {
          admin.close();
      } catch (IOException e) {
          LOG.error("Close admin failed " ,e);
      }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
}
```

5. 插入数据

注意

- 尽量使用批量 put 接口，putlist 大小尽量大些（可以设置 1000 个 put 调用一次 put 接口进行提交），这样可以减少提交次数，提升数据 put 性能。
- 如果数据量比较大，可以通过 put 提交时关闭 wal 提升性能。

HBase 是一个面向列的数据库，一行数据，可能对应多个列族，而一个列族又可以对应多个列。通常，写入数据的时候，我们需要指定要写入的列（含列族名称和列名称）。

如果要往 HBase 表中写入一行数据，首先需要构建一个 Put 实例。Put 中包含了数据的 RowKey 值和指定列（含列族名称和列名称）的 Value 值，一个 RowKey 的 Value 值可以有多个（即包含多个列），示例如下：

(1) 实例化一个 put 对象

```
Put put = new Put(Bytes.toBytes("rowkey1"));
```

(2) add 要添加的数据

```
put.add(Bytes.toBytes("cf1"), Bytes.toBytes("column1"), Bytes.toBytes("aaa"));
put.add(Bytes.toBytes("cf1"), Bytes.toBytes("column2"), Bytes.toBytes("bbb"));
put.add(Bytes.toBytes("cf2"), Bytes.toBytes("column3"), Bytes.toBytes("111"));
put.add(Bytes.toBytes("cf2"), Bytes.toBytes("column4"), Bytes.toBytes("222"));
```

(3) 提交一次 put 数据请求

```
table.put(put);
```

批量 put 示例代码如下：

```
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
Connection conn = null;
HTable table = null;
try {
  conn = ConnectionFactory.createConnection(hbaseconf);
  String tablename = "test";
  TableName tName = TableName.valueOf(tablename);
  table = (HTable)conn.getTable(tName);
  List<Put> puts = Lists.newArrayList();
  //TODO 构建 put 放到 put 集合中
  //……
  table.put(puts);

} catch (IOException e) {

  LOG.error("put data failed ", e);
}finally{
  if (table != null) {
```

```
        try {
            table.close();
        } catch (IOException e) {
            LOG.error("Close htable failed " ,e);
        }
    }
    if(conn != null){
        try {
            conn.close();
        } catch (IOException e) {
            LOG.error("Close connection failed " ,e);
        }
    }
}
}
```

6. 删除数据

注意

- 如果删除数据较多，请使用批量 delete 接口。
- 如果被删除的 cell 所在的列族上设置了二级索引，也会同步删除索引数据。

示例代码如下所示：

```
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));

Connection conn = null;
HTable table = null;
try {
  conn = ConnectionFactory.createConnection(hbaseconf);
  String tablename = "test";
  TableName tName = TableName.valueOf(tablename);
  table = (HTable)conn.getTable(tName);
  byte[] rowKey = Bytes.toBytes("012005000201");
  Delete delete = new Delete(rowKey);
  List<Delete> deletes = Lists.newArrayList();
  //TODO 构建 put 放到 put 集合中
  //
  table.delete(delete);
} catch (IOException e) {
  LOG.error("put data failed ", e);
}finally{
  if (table != null) {
      try {
          table.close();
```

```
        } catch (IOException e) {
            LOG.error("Close htable failed " ,e);
        }
    }
    if(conn != null){
        try {
            conn.close();
        } catch (IOException e) {
            LOG.error("Close connection failed " ,e);
        }
    }
}
```

7. 读取一行数据

说明

要从表中读取数据，首先需要实例化该表对应的 Table 实例，然后创建一个 Scan 对象，并针对查询条件设置 Scan 对象的参数值，为了提高查询效率，最好指定 StartRow 和 StopRow。查询结果的多行数据保存在 ResultScanner 对象中，每行数据以 Result 对象形式存储，Result 中存储了多个 Cell。

从表中读取一条数据，要创建一个 Get 对象。查询结果的该行数据存储在 Result 对象中，Result 中存储了多个 KeyValue 对，示例程序如下：

```
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
Connection conn = null;
HTable table = null;
TableName tableName = TableName.valueOf("test");

// 指定列族名.
byte[] familyName = Bytes.toBytes("info");
// 指定列名.
byte[][] qualifier = { Bytes.toBytes("name"), Bytes.toBytes("address") };
// 指定 rowkey.
byte[] rowKey = Bytes.toBytes("012005000201");
try {
  conn = ConnectionFactory.createConnection(hbaseconf);
  table = (HTable)conn.getTable(tableName);
  Get get = new Get(rowKey);
  // 设置 rowkey 所要获取数据对应的列族名及列名
  get.addColumn(familyName, qualifier[0]);
  get.addColumn(familyName, qualifier[1]);

  Result result = table.get(get);
  //循环获取设置的列族及列下的对应的 value 值
```

```
  for(int i = 0; i < qualifier.length; i++){
      //根据列族名和列名获取数据
      byte[] sourceData = result.getValue(familyName,  qualifier[i]);
      String value = null;
      if (sourceData != null) {
      value = Bytes.toString(sourceData);
      }
      LOG.info("rowkey :{} get the value of the family:{} and qualifier:{}, the value is {}.",
rowKey, Bytes.toString(familyName), Bytes.toString(qualifier[i]));
  }

  LOG.info("Get data successfully.");
} catch (IOException e) {
  LOG.error("Get data failed " ,e);
} finally {
  if (table != null) {
    try {
      //关闭htable.
      table.close();
    } catch (IOException e) {
      LOG.error("Close table failed " ,e);
    }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
}
```

8. 读取多行数据

要从表中读取数据，首先需要实例化该表对应的HTable对象，然后创建一个Scan对象，并针对查询条件设置Scan的参数值，为了提高查询效率，最好指定StartKey和EndKey。查询结果的多行数据保存在ResultScanner对象，每行数据以Result对象形式存储，Result中存储了多个KeyValue对，示例程序如下：

```
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));

Connection conn = null;
HTable table = null;
ResultScanner resultScanner = null;
TableName tableName = TableName.valueOf("test");

try {
```

```
  conn = ConnectionFactory.createConnection(hbaseconf);
  table = (HTable)conn.getTable(tableName);
    Scan scan = new Scan();
    //设置 scan 的 start key
    scan.setStartRow(Bytes.toBytes("0001"));
    //设置 scan 的 end key
    scan.setStopRow(Bytes.toBytes("0009"));

    scan.setCaching(1000);

  resultScanner = table.getScanner(scan);
  for (Result rs = resultScanner.next(); rs != null; rs = resultScanner.next())
  {
      for (KeyValue kv : rs.list())
      {
          //TODO 数据操作
      }
  }
  LOG.info("Get data successfully.");
} catch (IOException e) {
  LOG.error("Get data failed " ,e);
} finally {
      if(resultScanner != null) {
      try {
          // Close the HTable object.
          resultScanner.close();
        } catch (IOException e) {
          LOG.error("Close resultScanner failed " ,e);
        }
  }
  if (table != null) {
    try {
      table.close();
    } catch (IOException e) {
      LOG.error("Close table failed " ,e);
    }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
}
```

4.1.3 Kerberos 环境 Java 开发样例

说明

HBase 调用结束时注意关闭 HTable、connection、ResultScanner 等连接。

1. 创建 HBaseConfigurtation

- 方式一：通过配置文件创建，代码示例如下：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");
```

- 方式二：给出必要配置项进行创建，代码示例如下所示：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.set("hbase.zookeeper.quorum","101.8.89.150,101.8.89.151,101.8.89.152");
hbaseconf.set("hbase.zookeeper.property.clientPort","2181");
hbaseconf.set("zookeeper.znode.parent","/hbase-secure");
hbaseconf.set("keytab.file" , "/conf/hbase.keytab");
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");
```

注意事项：

- 方式一与方式二相比尽量选取方式一，可以避免不必要的问题出现。
- 选取方式二，配置的 zookeeper 地址需要为 HBase 集群所使用的 zookeeper 地址，为了保证容灾，要配置 HBase 集群所有 zookeeper 地址，注意配置项“zookeeper.znode.parent”及"hbase.zookeeper.property.clientPort"的配置值与连接的 HBase 集群保持一致。
- krb5.conf、core-site.xml、hdfs-site.xml、hbase-site.xml，认证所需 keytab 等文件从所使用的 HBase 集群中获取，krb5.conf 在集群获取路径为安装 Kerberos 客户端服务器 /etc/krb5.conf，从集群中获取的配置文件按照开发的目录结构进行存放。
- 使用方式二进行 configuration 创建后，如果出现创建连接超时的问题，请在运行进程 classpath 中添加 hbase-site.xml。

2. 创建 Connection

代码示例如下所示：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
```

```
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");

Connection conn = null;
try {
  UserGroupInformation.setConfiguration(hbaseconf);
  UserGroupInformation.loginUserFromKeytab("hbase@HDE.TEST.COM",
"/etc/security/keytabs/hbase.keytab" );
  conn = ConnectionFactory.createConnection(hbaseconf);
  Admin admin = conn.getAdmin();
  TableName[] names = admin.listTableNames();
  for(TableName name : names){
      System.out.println("biaoming:" + name.getNameAsString());
  }
} catch (IOException e) {
  LOG.error("Create connection failed " ,e);
}finally{

  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
}
```

3. 创建表

示例代码如下所示：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));

hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");
Connection conn = null;
Admin admin = null;
      TableName tableName = TableName.valueOf("table-test");
try {
   UserGroupInformation.setConfiguration(hbaseconf);
  UserGroupInformation.loginUserFromKeytab("hbase@HDE.TEST.COM",
"/etc/security/keytabs/hbase.keytab" );
```

```
  conn = ConnectionFactory.createConnection(hbaseconf);
  //创建表描述符
  HTableDescriptor htd = new HTableDescriptor(tableName);
  //创建列族描述符
  HColumnDescriptor hcd = new HColumnDescriptor("info");
  // 设置压缩方法，HBase 提供了 GZ 和 SNAPPY 两种压缩方法，建议使用 SNAPPY
  hcd.setCompressionType(Compression.Algorithm.SNAPPY);
  htd.addFamily(hcd);
  admin = conn.getAdmin();
  if (!admin.tableExists(tableName)) {
      LOG.info("Creating table...");
      //创建表
      admin.createTable(htd);
      LOG.info("Table created successfully.");
  } else {
      LOG.warn("table already exists");
  }

} catch (IOException e) {
  LOG.error("Create table failed " ,e);
}finally{
  if (admin != null) {
      try {
          admin.close();
      } catch (IOException e) {
          LOG.error("Failed to close admin " ,e);
      }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Failed to close connection ", e);
      }
  }
  }
```

注意事项：

- 可以设置列族的压缩方式，代码示例如下：

  //设置编码算法，HBase 提供了 DIFF，FAST_DIFF，PREFIX 三种编码算法。

  hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);

  //设置文件压缩方式，HBase 默认提供了 GZ 和 SNAPPY 两种压缩算法

  //其中 GZ 的压缩率高，但压缩和解压性能低，适用于冷数据

  //SNAPPY 压缩率低，但压缩解压性能高，适用于热数据

  //建议默认开启 SNAPPY 压缩

  hcd.setCompressionType(Compression.Algorithm.SNAPPY);

- 可以通过指定起始和结束 RowKey，或者通过 RowKey 数组预分 Region 两种方式建表，建表时进行 region 预分可以有效的提升表的读写性能，代码片段如下：

```
//创建一个预划分 region 的表
byte[][] splits = new byte[4][] ;
splits[0] = Bytes.toBytes("A");
splits[1] = Bytes.toBytes("H");
splits[2] = Bytes.toBytes("O");
splits[3] = Bytes.toBytes("U");
admin.createTable(htd, splits);
```

4. 删除表

注意

删除表之前必须 disable 要删除的表。

代码示例如下所示：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");

Connection conn = null;
TableName tableName = TableName.valueOf("table-test");
Admin admin = null;
try {
    UserGroupInformation.setConfiguration(hbaseconf);
  UserGroupInformation.loginUserFromKeytab("hbase@HDE.TEST.COM",
"/etc/security/keytabs/hbase.keytab" );
  conn = ConnectionFactory.createConnection(hbaseconf);
  admin = conn.getAdmin();
  if (admin.tableExists(tableName)) {
      // disable 需要删除的表
      admin.disableTable(tableName);
      // 删除表
      admin.deleteTable(tableName);//注[1]
  }
  LOG.info("Drop table successfully.");
} catch (IOException e) {
  LOG.error("Drop table failed " ,e);
} finally {
  if (admin != null) {
```

```
        try {
            admin.close();
        } catch (IOException e) {
            LOG.error("Close admin failed " ,e);
        }
    }
    if(conn != null){
        try {
            conn.close();
        } catch (IOException e) {
            LOG.error("Close connection failed " ,e);
        }
    }
}
```

5. 插入数据

注意

- 尽量使用批量 put 接口，putlist 大小尽量大些（可以设置 1000 个 put 调用一次 put 接口进行提交），这样可以减少提交次数，提升数据 put 性能。
- 如果数据量比较大，可以通过 put 提交时关闭 wal 提升性能。

HBase 是一个面向列的数据库，一行数据，可能对应多个列族，而一个列族又可以对应多个列。通常，写入数据的时候，我们需要指定要写入的列（含列族名称和列名称）。

如果要往 HBase 表中写入一行数据，需要首先构建一个 Put 实例。Put 中包含了数据的 RowKey 值和指定列（含列族名称和列名称）的 Value 值，一个 RowKey 的 Value 值可以有多个（即包含多个列），示例如下：

(1) 实例化一个 put 对象

```
Put put = new Put(Bytes.toBytes("rowkey1"));
```

(2) add 要添加的数据

```
put.add(Bytes.toBytes("cf1"), Bytes.toBytes("column1"), Bytes.toBytes("aaa"));
put.add(Bytes.toBytes("cf1"), Bytes.toBytes("column2"), Bytes.toBytes("bbb"));
put.add(Bytes.toBytes("cf2"), Bytes.toBytes("column3"), Bytes.toBytes("111"));
put.add(Bytes.toBytes("cf2"), Bytes.toBytes("column4"), Bytes.toBytes("222"));
```

(3) 提交一次 put 数据请求

```
table.put(put);
```

批量 put 示例代码如下：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
```

```
hbaseconf.set("hadoop.security.authentication","kerberos");

Connection conn = null;
HTable table = null;
try {
  UserGroupInformation.setConfiguration(hbaseconf);
  UserGroupInformation.loginUserFromKeytab("hbase@HDE.TEST.COM",
"/etc/security/keytabs/hbase.keytab" );
  conn = ConnectionFactory.createConnection(hbaseconf);
  String tablename = "test";
  TableName tName = TableName.valueOf(tablename);
    table = (HTable)conn.getTable(tName);
    List<Put> puts = Lists.newArrayList();
    //TODO 构建put放到put集合中
    //……
    table.put(puts);
} catch (IOException e) {

  LOG.error("put data failed ", e);
}finally{
  if (table != null) {
      try {
          table.close();
      } catch (IOException e) {
          LOG.error("Close htable failed " ,e);
      }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
}
}
```

6. 删除数据

注意

- 如果删除数据较多，请使用批量delete接口。
- 如果被删除的cell所在的列族上设置了二级索引，也会同步删除索引数据。

示例代码如下所示：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
```

```
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");
Connection conn = null;
HTable table = null;
try {
  UserGroupInformation.setConfiguration(hbaseconf);
  UserGroupInformation.loginUserFromKeytab("hbase@HDE.TEST.COM",
"/etc/security/keytabs/hbase.keytab" );
  conn = ConnectionFactory.createConnection(hbaseconf);
  String tablename = "test";
  TableName tName = TableName.valueOf(tablename);
    table = (HTable)conn.getTable(tName);
    byte[] rowKey = Bytes.toBytes("012005000201");
    Delete delete = new Delete(rowKey);
    List<Delete> deletes = Lists.newArrayList();
    //TODO 构建put放到put集合中

    table.delete(delete);
} catch (IOException e) {
  LOG.error("put data failed ", e);
}finally{
  if (table != null) {
      try {
          table.close();
      } catch (IOException e) {
          LOG.error("Close htable failed " ,e);
      }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
  }
```

7. 读取一行数据

注意

要从表中读取数据，首先需要实例化该表对应的Table实例，然后创建一个Scan对象，并针对查询条件设置Scan对象的参数值，为了提高查询效率，最好指定StartRow和StopRow。查询结果的多行数据保存在ResultScanner对象中，每行数据以Result对象形式存储，Result中存储了多个Cell。

从表中读取一条数据，要创建一个 Get 对象。查询结果的该行数据存储 Result 对象中，Result 中存储了多个 KeyValue 对，示例程序如下：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");

Connection conn = null;
HTable table = null;
TableName tableName = TableName.valueOf("test");

// 指定列族名.
byte[] familyName = Bytes.toBytes("info");
// 指定列名.
byte[][] qualifier = { Bytes.toBytes("name"), Bytes.toBytes("address") };
// 指定 rowkey.
byte[] rowKey = Bytes.toBytes("012005000201");
try {
 UserGroupInformation.setConfiguration(hbaseconf);
 UserGroupInformation.loginUserFromKeytab("hbase@HDE.TEST.COM",
"/etc/security/keytabs/hbase.keytab" );
  conn = ConnectionFactory.createConnection(hbaseconf);
  table = (HTable)conn.getTable(tableName);
  Get get = new Get(rowKey);
  // 设置 rowkey 所要获取数据对应的列族名及列名
  get.addColumn(familyName, qualifier[0]);
  get.addColumn(familyName, qualifier[1]);

  Result result = table.get(get);
  //循环获取设置的列族及列下的对应的 value 值
  for(int i = 0; i < qualifier.length; i++){
    //根据列族名和列名获取数据
      byte[] sourceData = result.getValue(familyName,  qualifier[i]);
      String value = null;
      if (sourceData != null) {
          value = Bytes.toString(sourceData);
      }
      LOG.info("rowkey :{} get the value of the family:{} and qualifier:{}, the value is {}.",
rowKey, Bytes.toString(familyName), Bytes.toString(qualifier[i]));
  }

  LOG.info("Get data successfully.");
} catch (IOException e) {
  LOG.error("Get data failed " ,e);
```

```
} finally {
  if (table != null) {
    try {
      // Close the HTable object.
      table.close();
    } catch (IOException e) {
      LOG.error("Close table failed " ,e);
    }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
  }
```

8. 读取多行数据

要从表中读取数据，首先需要实例化该表对应的 HTable 对象，然后创建一个 Scan 对象，并针对查询条件设置 Scan 的参数值，为了提高查询效率，最好指定 StartKey 和 EndKey。查询结果的多行数据保存在 ResultScanner 对象，每行数据以 Result 对象形式存储，Result 中存储了多个 KeyValue 对，示例程序如下：

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
Configuration hbaseconf = HBaseConfiguration.create();
hbaseconf.addResource(new Path("conf/core-site.xml"));
hbaseconf.addResource(new Path("conf/hdfs-site.xml"));
hbaseconf.addResource(new Path("conf/hbase-site.xml"));
hbaseconf.set("keytab.file" , "/conf/hbase.keytab" );
hbaseconf.set("kerberos.principal" , "hbase@HDE.TEST.COM" );
hbaseconf.set("hadoop.security.authentication","kerberos");

Connection conn = null;
HTable table = null;
ResultScanner resultScanner = null;
TableName tableName = TableName.valueOf("test");

try {
  UserGroupInformation.setConfiguration(hbaseconf);
  UserGroupInformation.loginUserFromKeytab("hbase@HDE.TEST.COM",
"/etc/security/keytabs/hbase.keytab" );
  conn = ConnectionFactory.createConnection(hbaseconf);
  table = (HTable)conn.getTable(tableName);
    Scan scan = new Scan();
    //设置 scan 的 start key
    scan.setStartRow(Bytes.toBytes("0001"));
    //设置 scan 的 end key
    scan.setStopRow(Bytes.toBytes("0009"));
```

```
    scan.setCaching(1000);

  resultScanner = table.getScanner(scan);
  for (Result rs = resultScanner.next(); rs != null; rs = resultScanner.next())
  {
      for (KeyValue kv : rs.list())
      {
          //TODO 数据操作
      }
  }
  LOG.info("Get data successfully.");
} catch (IOException e) {
  LOG.error("Get data failed " ,e);
} finally {
  if(resultScanner != null) {
      try {
          // Close the HTable object.
          resultScanner.close();
        } catch (IOException e) {
          LOG.error("Close resultScanner failed " ,e);
        }
  }
  if (table != null) {
    try {
      table.close();
    } catch (IOException e) {
      LOG.error("Close table failed " ,e);
    }
  }
  if(conn != null){
      try {
          conn.close();
      } catch (IOException e) {
          LOG.error("Close connection failed " ,e);
      }
  }
}
```

4.1.4 HBase 高级专题

1. 过滤器

与传统的 RDBMS 相比，HBase 的基础查询只能根据特定的行键进行查询或根据行键的范围来查询，操作略显苍白。HBase 提供的过滤器来查询，可以根据列族、列、版本等更多的条件对数据进行过滤，过滤器可以高效的完成查询过滤的任务，带有过滤器条件的 RPC 查询请求会把过滤器分发到各个 RegionServer，叫谓词下推(predicate push down)，这样可以保证被过滤掉的数据不会被传送到客户端，降低网络传输的压力。

使用过滤器至少需要两类参数，一类是抽象的操作符，HBase 提供了枚举类型的变量来表示这些抽象的操作符：

- LESS
- LESS_OR_EQUAL
- EQUAL
- NOT_EQUAL
- GERATER_OR_EQUAL
- GREATER
- NO_OP

另一类是比较器，代表具体的比较逻辑，例如字节级别的比较、字符串级别的比较等。有了这两类参数就可以清晰的定义筛选条件，从而过滤数据。

下面列举部分过滤器的功能，详细说明请参照 HBase 官方 API 说明。

- 比较器

  比较器是过滤器的核心组成之一，用于处理具体的比较逻辑，下面列举常见的比较器。

  - RegexStringComparator：支持正则表达式的值的比较。
  - SubstringComparator：用于检查一个子串是否存在于值中，不区分大小写。
  - BinaryPrefixComparator：前缀二进制比较器，只比较前缀是否相同。
  - BinaryComparator：二进制比较器，用于按字典顺序比较 Byte 数据值。
- 列值过滤器
  - SingleColumnValueFilter：用于测试列值相等（CompareOp.EQUAL）、不等（CompareOp.NOT_EQUAL）、范围（CompareOp.GREATER）等情况。
  - SingleColumnValueExcludeFilter：用于单列值过滤，但并不查询出该列的值。
- 行键过滤器
  - RowFilter：对某一行进行过滤。
  - RandomRowFilter：随机选择一行进行过滤。
- 功能过滤器
  - PageFilter：用于按行分页。
  - FirstKeyOnlyFilter：只查每个行键的第一个键值对，用于在统计计数的时候提高效率。
  - KeyOnlyFilter：只查有“键”元数据信息，不显示值，可以提升扫描的效率。
  - InclusiveStopFilter：使 Scan 包含 stop-row（常规 Scan 包含 start-row 但不包含 stop-row）。
  - columnPaginationFilter：按列分页过滤器，针对列数很多的情况使用。

以行键过滤器为例，代码如下：

```
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("c0"));
Filter filter = new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL, new
BinaryComparator(Bytes.toBytes("row-1")));
scan.setFilter(filter);
ResultScanner scanner1= table.getScanner(scan);
for (Result res : scanner1) {
    System.out.println(res);
```

```
}
scanner1.close();
```

2. 计数器

计数器可以方便、快速地进行计数操作，而且避免了加锁等保证原子性的操作。但本质上，计数器还是列，有自己的列族和列名。值得注意的是，维护计数器的值最好使用 HBase 提供的 API，直接操作更新很容易造成数据混乱。计数器所负责的工作在 RegionServer 中完成。计数器的增量可以是正数或负数，正数代表加，负数代表减。

- 单列计数器

  单列计数器必须显式地指定确定的列，并且只能是一列。

  示例代码如下：

  ```
  _hTable.incrementColumnValue(Bytes.toBytes("row-001"),Bytes.toBytes("info"),
  Bytes.toBytes("pv"), 10L); // pv + 10
  _hTable.incrementColumnValue(Bytes.toBytes("row-001"),Bytes.toBytes("info"),
  Bytes.toBytes("pv"), -1L); // pv – 1
  ```

- 多列计数器

  多列计数器用于同时更新同一个行键下多列的计数器值，使用的方法是 increment()，参数是 Increment 实例。

  示例代码如下：

  ```
  Increment increment = new Increment(Bytes.toBytes("row"));
  increment.addColumn(Bytes.toBytes("info"), Bytes.toBytes("pv"), 1L); // pv + 1
  increment.addColumn(Bytes.toBytes("info"), Bytes.toBytes("uv"), 1L); // uv + 1
  _hTable.increment(increment);
  ```

3. 协处理器

如果要对 HBase 中的数据进行某种统计，比如统计某个字段最大值，统计满足某种条件的记录数，统计各种记录特点，并按照记录特点分类（类似于 sql 的 group by），常规的做法是把 HBase 中整个表的数据 scan 出来，或者加一个 filter，进行一些初步的过滤，但是这么做会有很大的副作用，比如占用大量的网络带宽（表级别到达千万级别、亿级别之后尤为明显），RPC 的通信数据量也骤然增加。于是，HBase 引入了协处理器(coprocessors)，能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制等。

系统协处理器可以全局导入 region server 上的所有数据表，表协处理器即用户可以指定一张表使用协处理器。协处理器框架为了更好支持其行为的灵活性，提供了两个不同方面的插件。

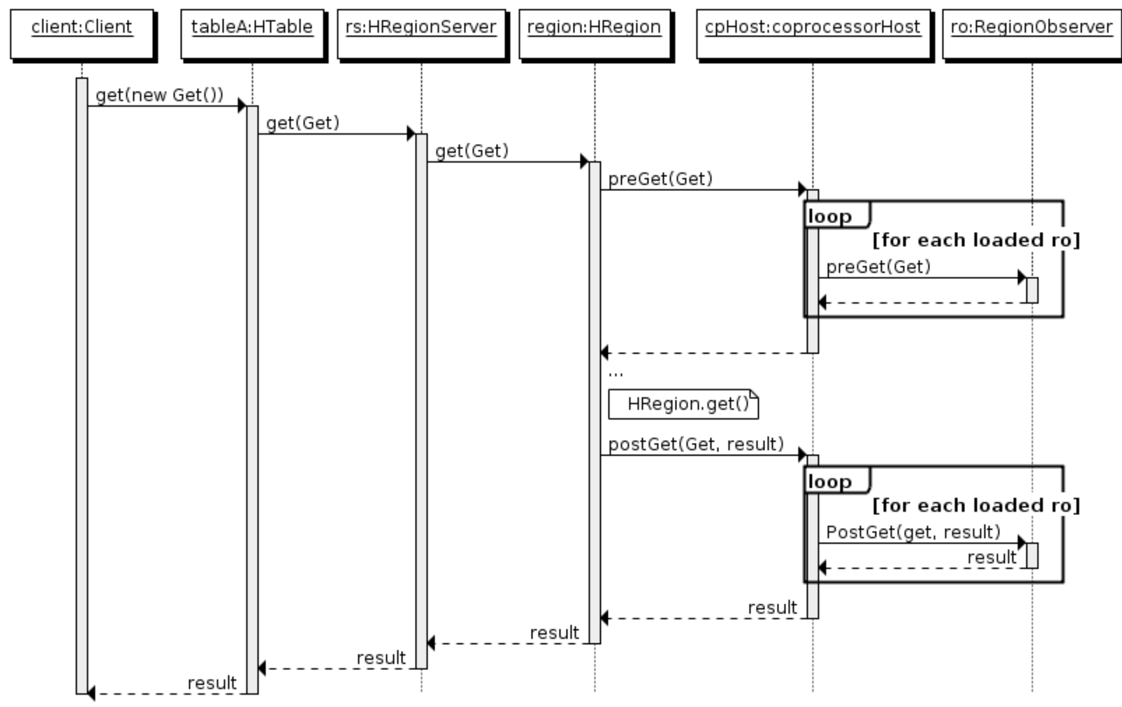
- 一个是观察者（observer），类似于关系数据库的触发器。
- 另一个是终端（endpoint），动态的终端有点像存储过程。

观察者的设计意图是允许用户通过插入代码来重载协处理器框架的 upcall 方法，而具体事件触发的 callback 方法由 HBase 的核心代码来执行。协处理器框架处理所有的 callback 调用细节，协处理器自身只需要插入添加或者改变的功能。它提供了三种观察者接口：

- RegionObserver：提供客户端的数据操纵事件钩子：Get、Put、Delete、Scan 等。
- WALObserver：提供 WAL 相关操作钩子。
- MasterObserver：提供 DDL-类型的操作钩子。如创建、删除、修改数据表等。

这些接口可以同时使用在同一个地方，按照不同优先级顺序执行。用户可以任意基于协处理器实现复杂的 HBase 功能层，其工作原理如图 4-1。

图4-1 RegionObserver 工作原理



终端是动态 RPC 插件的接口，它的实现代码被安装在服务器端，从而能够通过 HBase RPC 唤醒。客户端类库提供了非常方便的方法来调用这些动态接口，它们可以在任意时刻调用一个终端，它们的实现代码会被目标 region 远程执行，结果会返回到终端。用户可以结合使用这些强大的插件接口，为 HBase 添加全新的特性。终端的使用，流程如下：

(1) 定义一个新的 protocol 接口，必须继承 CoprocessorProtocol。

(2) 实现终端接口，该实现会被导入 region 环境执行。

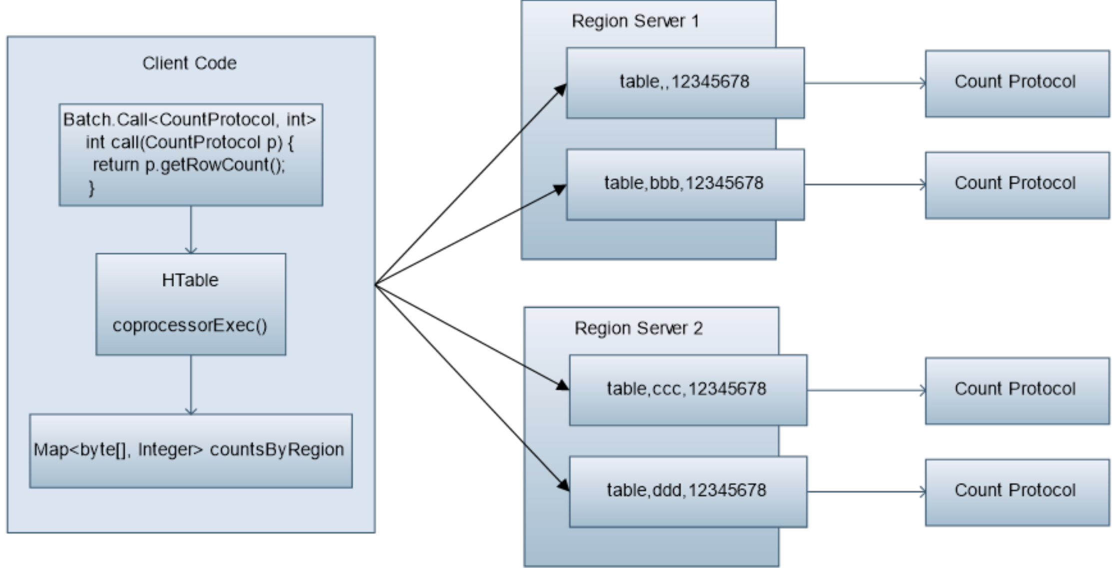
(3) 继承抽象类 BaseEndpointCoprocessor。

(4) 在客户端，终端可以被两个新的 HBase Client API 调用。

- 单个 region：HTableInterface.coprocessorProxy(Class<T> protocol, byte[] row)。
- regions 区域：HTableInterface.coprocessorExec(Class<T> protocol, byte[] startKey, byte[] endKey, Batch.Call<T,R> callable)。

整体的终端调用过程范例，如图 4-2 所示：

图4-2 终端调用过程



(5) 启动协处理器 Aggregation

有两个方法启动协处理器 Aggregation：

方法 1：启动全局 aggregation，能过操纵所有的表上的数据。通过修改 hbase-site.xml 这个文件来实现，只需要添加如下代码：

```
<property>
   <name>hbase.coprocessor.user.region.classes</name>
   <value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
 </property>
```

方法 2：启用表 aggregation，只对特定的表生效。通过 HBase Shell 来实现。

a. disable 指定表：

hbase> disable 'mytable'

b. 添加 aggregation：

hbase> alter 'mytable', METHOD =>
'table_att','coprocessor'=>'|org.apache.hadoop.hbase.coprocessor.AggregateImplementation||'

c. 重启指定表：

hbase> enable 'mytable'

4. 协处理器示例

(1) 示例一：统计行数

代码实现如下：

```
public class CountRows {
    private static final byte[] TABLE_NAME = Bytes.toBytes("mytable");
    private static final byte[] CF = Bytes.toBytes("vent");
    public static void main(String[] args) throws Throwable {
        Configuration customConf = new Configuration();
        customConf.setStrings("hbase.zookeeper.quorum", "node1,node2,node3");
        //提高 RPC 通信时长
```

```
        customConf.setLong("hbase.rpc.timeout", 600000);
        //设置Scan缓存
        customConf.setLong("hbase.client.scanner.caching", 1000);
        Configuration configuration = HBaseConfiguration.create(customConf);
        AggregationClient aggregationClient = new AggregationClient(configuration);
        Scan scan = new Scan();
        //指定扫描列族，唯一值
        scan.addFamily(CF);
        long rowCount = aggregationClient.rowCount(TABLE_NAME, null, scan);
        System.out.println("row count is " + rowCount);
    }
}
```

(2) 示例二：创建二级索引

继承BaseRegionObserver类，实现prePut方法，在插入订单详情表之前，向索引表插入索引数据，代码实现如下：

```
public class TestCoprocessor extends BaseRegionObserver {
    @Override
    public void prePut(final ObserverContext<RegionCoprocessorEnvironment> e,
                       final Put put, final WALEdit edit, final boolean writeToWAL)
throws IOException {
        Configuration conf = new Configuration();
        HTable table = new HTable(conf, "index_table");
        List<KeyValue> kv = put.get("data".getBytes(), "name".getBytes());
        Iterator<KeyValue> kvItor = kv.iterator();
        while (kvItor.hasNext()) {
            KeyValue tmp = kvItor.next();
            Put indexPut = new Put(tmp.getValue());
            indexPut.add("index".getBytes(), tmp.getRow(),
Bytes.toBytes(System.currentTimeMillis()));
            table.put(indexPut);
        }
        table.close();
    }
}
```

5. MOB特性

(1) 场景描述

在实际应用中，需要存储大大小小的数据，比如图像数据、文档。小于10MB的数据一般都可以存储在HBase上。对于小于100KB的数据，HBase的读写性能是最优的。如果存放在HBase的数据大于100KB甚至到10MB大小时，插入同样个数的数据文件，总的数据量会很大，会导致频繁的compaction和split，占用很多CPU，磁盘IO频率很高，性能严重下降。

通过将MOB（Medium-sized Objects）数据（即100KB到10MB大小的数据）直接以HFile的格式存储在文件系统上（例如HDFS文件系统），通过expiredMobFileCleaner和Sweeper工具集中管理这些文件，然后把这些文件的地址信息及大小信息作为value存储在普通HBase的store上。这样就可以大大降低HBase的compation和split频率，提升性能。

(2) 配置描述

为了开启HBase MOB功能，用户需要在创建表或者修改表属性时在指定的列族上指定使用mob方式存储数据。MOB_THRESHOLD的单位是Byte，默认值是100KB。如果你写的文件超过这个值的大小，它就会使用MOB的方式进行存储。

在 HBase Shell 使用 mob 的方式：

hbase(main):003:0> create 'mob_test' ,{NAME => 'd', MOB_THRESHOLD => '102400', IS_MOB => 'true'}

```
0 row(s) in 1.2280 seconds
```

hbase(main):005:0> describe 'mob_test'

```
Table mob_test is ENABLED
mob_test
COLUMN FAMILIES DESCRIPTION
{NAME => 'd', MOB_THRESHOLD => '102400', VERSIONS => '1', KEEP_DELETED_CELLS => 'FALSE',
DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLICATION_SCOPE
=> '0', BLOO
MFILTER => 'ROW', IN_MEMORY => 'false', IS_MOB => 'true', COMPRESSION => 'NONE',
BLOCKCACHE => 'true', BLOCKSIZE => '65536'}
1 row(s) in 0.1660 seconds
```

在 Java API 中使用 mob 的方式：

HcolumnDescriptor hcd = new HcolumnDescriptor（”f”）;

hcd.setMobEnabled(true);

hcd.setMobThreshold(102400L);

(3) 手工压缩 MOB 文件

使用 compact_mob 和 majar_compact_mob 命令进行手工压缩。命令后可配置两个参数。第一个参数是表名，只传表名，则表中所有的 MOB 列都进行压缩；第二个参数是列名，如果传入列名，则只压缩指定列。

命令举例如下：

hbase>compact_mob ‘t1’

hbase>compact_mob ‘t1’, ‘f1’

hbase>major_compact_mob ‘t1’

hbase>major_compact_mob ‘t1’, ‘f1’

4.1.5 HBase 其它常用接口

HBase 其它常用接口函数，详情请参见官网 http://hbase.apache.org/apidocs/index.html。

5 常见问题解答

5.1 HBase Schema及Rowkey设计

HBase 在实际生产使用中，需要根据具体的业务使用场景，给出相匹配的 Rowkey 设计及 HBase 表 Schema 设计，提升 HBase 的使用性能，同时避免出现热点瓶颈问题。

5.1.1 HBase 表 Schema 设计原则

在 HBase 中有许多不同的数据集，具有不同的访问 Schema 级别和服务级别的期望，下面的设计法则只是概述。

- 目标 region 的大小介于 10 到 50 GB 之间。
- 单个单元格大小不要超过 10 MB，如果使用 MOB，则为 50 MB。否则，请考虑将单元格数据存储在 HDFS 中，并在 HBase 中存储指向数据的指针。
- 典型的 Schema 在每个表中有 1 到 3 个列族。HBase 表不应该被设计成模拟 RDBMS 表。
- 对于具有 1 或 2 列族的表格，大约 50-100 个 region 数量比较合适。
- 尽可能短地保留列族名称。列族名称存储在每个值(忽略前缀编码)中。它们不应该像在典型的 RDBMS 中一样具有自我记录和描述性。
- 如果正在存储基于时间的机器数据或日志记录信息，并且行密钥基于设备 ID 或服务 ID 加上时间，则最终可能会出现一种 Schema，即旧数据区域在某个时间段之后永远不会有额外的写入操作。在这种情况下，最终会有少量活动 region 和大量没有新写入的较旧 region。对于这些情况，可以容忍更多 region 域，因为资源消耗仅由活动 region 驱动。
- 如果只有一个列族忙于写入，则只有该列族兼容内存。分配资源时请注意写入 Schema。
- 使用合适的压缩。

5.1.2 HBase 表 Rowkey 设计原则

HBase 中的行按行键及顺序排序。这种设计优化了扫描（scan），允许将相关的行或彼此靠近的行一起读取。但是，设计不佳的行键是 hotspotting 的常见来源。当大量客户端通信访问集群中的一个节点或仅少数几个节点时，会发生 Hotspotting。此客户端通信可能表示读取、写入或其他操作。客户端访问压倒负责托管该 region 的单个机器，从而导致性能下降并可能导致 region 不可用。这也会对由同一台 RegionServer 托管的其他 region 产生不利影响，因为该主机无法为请求的负载提供服务。设计数据访问 Schema 以使集群得到充分和均匀利用是非常重要的。

为了防止 hotspotting 写入，合理设计 rowkey，可以将数据读写压力分散到多个 region 上，而不是一个 region。以下描述了避免 hotspotting 的一些常用技术，以及各自优缺点。

1. Salting

Salting 是指将随机数据添加到行键的开头。在这种情况下，Salting 是指为行键添加一个随机分配的前缀，以使它的排序方式与其他方式不同。所有可能前缀的数量对应于要分散数据的 region 的数量。如果有一些“hotspotting”行键模式，反复出现在其他更均匀分布的行中，那么 Salting 可能会有帮助。

下面的例子说明了 Salting 能在多个 RegionServer 间分散负载，同时也说明了它在读操作时候的负面影响。

假设存在的行键列表，表按照每个字母对应一个 region 进行分割。前缀'a'是一个区域，前缀'b'是另一个区域。在此表中，所有以'f'开头的行都在同一个区域中。本示例重点关注具有以下键的行：

foo0001

foo0002

foo0003

foo0004

现在，假设想将它们分散到不同的 region 上，就需要用到四种不同的 salts：a，b，c，d。在这种情况下，每种字母前缀都对应着不同的一个 region。用上这些 salts 后，便有了下面这样的行键。由于现在想把它们分到四个独立的区域，理论上吞吐量会是之前写到同一 region 的情况的吞吐量的四倍。

a-foo0003

b-foo0001

c-foo0004

d-foo0002

如果想新增一行，新增的一行会被随机指定四个可能的 salt 值中的一个，并放在某条已存在的行的旁边。

a-foo0003

b-foo0001

c-foo0003

c-foo0004

d-foo0002

由于前缀的指派是随机的，因而如果想要按照字典顺序找到这些行，则需要做更多的工作。从这个角度上看，salting 增加了写操作的吞吐量，却也增大了读操作的开销。

2. Hashing

可用一个单向的 hash 散列来取代随机指派前缀。这样能使一个给定的行在“salted”时有相同的前缀，从某种程度上说，这在分散了 RegionServer 间的负载的同时，也允许在读操作时能够预测。确定性 hash（deterministic hash）能让客户端重建完整的行键，以及像正常的一样用 Get 操作重新获得想要的行。

考虑和上述 salting 一样的情景，现在可以用单向 hash 来得到行键 foo0003，并可预测得‘a’这个前缀。然后为了重新获得这一行，需要先知道它的键。可以进一步优化这一方法，如让特定的键对总是在相同的 region 上。

3. Reversing the Key(反转键)

通过反转一段固定长度或者可数的键，来让最常改变的部分（最低显著位，the least significant digit）在第一位，这样有效地打乱了行键，但是却牺牲了行排序的属性。

5.2 调优

5.2.1 组件端调优

可以通过修改 hbase-site.xml 中的配置调节 HBase 性能，部分配置如表 5-1 所示。

表5-1 HBase 配置优化

参数名称	参数说明	缺省值
hbase.regionserver.handler.count	该设置决定了处理RPC的线程数量，通常可以调大。当请求内容很大的时候，如果该值设置过大则会占用过多内存，导致频繁GC，或者出现OutOfMemory	30
hbase.hregion.memstore.flush.size	memstore的大小，超过该值数据将被flush到HDFS，默认128M，单位字节	134217728
hbase.hregion.memstore.block.multiplier	默认值4，如果memstore的内存大小已经超过了hbase.hregion.memstore.flush.size的4倍，则会阻塞memstore的写操作，直至降至hbase.hregion.memstore.flush.size以下。为避免阻塞，可适当调整该值	4
hbase.hstore.compactionThreshold	HStroe的storeFile数量大于该值时，则可能会进行compact。可以适当调大，减少compact的次数	3
hbase.client.scanner.caching	HBase scan缓存大小。应该根据业务特征配置，如果一行数据太大，则应该设置一个较小的值，如果一行数据很小，则可以设置一个较大的值	100

5.2.2 其它调优

1. 建议 rowkey 和列族名称越短越好

每个 cell 值均会存储一次 rowkey 和列族名称，rowkey 和列族名称越长，占用的存储空间越大，这样会极大影响 HFile 的存储效率。

2. rowkey 遵循散列原则

HBase 中的行是按照 rowkey 的字典顺序排序的，这种设计优化了 scan 操作，但是当数据集中到少数的 region，大量的访问会导致单个 region 超出自身承受能力，引起性能下降甚至 region 不可用。所以建议 rowkey 使用散列字段，使数据均衡分布在 RegionServer 上。

3. 列族越少越好

HBase 目前不能较好的处理两列族或三列族以上的业务应用，因此请将 HBase 表 schema 设计中的列族数量保持在较低水平。

目前，flushing 和 compactions 是按照每个 region 进行的，所以当一个列族操作大量数据的时候会引发一个 flush。那些邻近的列族也有进行 flush 操作，尽管它们没有操作多少数据。当许多列族存在时，flushing 和 compactions 相互作用可能会导致一堆不必要的 I/O（要通过更改 flushing 和 compactions 来针对每个列族解决该问题）。

列族是一个集中的存储单元，将具有相同 IO 特性的列放在一个列族中会更高效，并且过多的列族会相互影响，影响 HBase 的效率。

5.3 运维类问题

5.3.1 日志查看方法

查看 HBase 的日志信息，有两种方式：

(1) 通过日志管理页面查看。

(2) 登录 HBase 集群后台，直接查看日志信息。

1. 通过日志管理查看

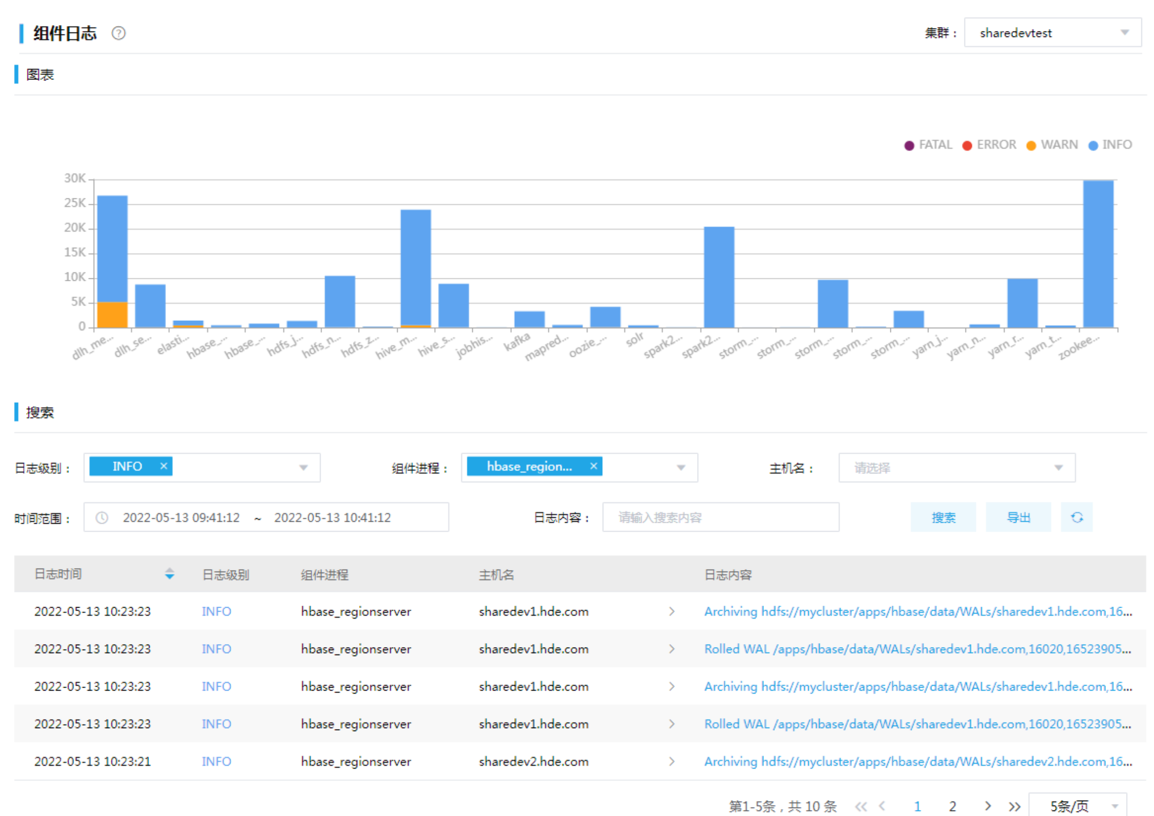
组件日志页面展示指定集群中产生的所有组件日志信息，支持通过日志级别、组件名、主机名、时间范围等筛选查询日志。

(1) 在集群管理的左侧导航树中选择[日志管理/组件日志]，进入组件日志页面。

(2) 组件日志以图表和日志列表形式展示系统中的组件日志。

- 图表：通过分析日志列表中日志的级别和数量等信息，以柱状图的形式展示不同组件产生的不同级别的日志数量。
- 日志列表：展示系统内指定集群的组件日志列表。展示信息包括日志产生的时间、日志级别、产生日志的组件和主机、日志内容。单击日志内容，可查看日志的详细信息。

图5-1 组件日志页面



2. 登录 HBase 集群后台，直接查看日志

HBase 集群日志默认路径在/var/de_log/hbase。日志路径信息，可以在 HBase 配置页面中，通过搜索 hbase_log_dir 来查看。

3. 日志查看注意点

先通过 HBase Web UI 或大数据平台管理页面找出有问题的 HBase 节点，如果为 HBase Master 问题，则找到 HBase Master 节点日志，在日志中寻找报错信息（搜索关键字：Exception、ERROR），根据报错信息分析出错原因，找到对应的处理方法。如果为 HBase Regionserver 问题，则找到 HBase Regionserver 节点日志，在日志中寻找报错信息（搜索关键字：Exception、ERROR），根据报错信息分析出错原因，找到对应的处理方法。

对于现场无法定位问题，需要将问题节点日志及相关 Master 相关日志发给技术服务人员进行问题定位解决。

5.3.2 常见问题

1. HBase 时钟不同步问题造成 HBase 进程异常退出

报错异常信息：

```
org.apache.hadoop.hbase.ClockOutOfSyncException: Server slave1,16020,1494163890158 has been rejected; Reported time is too far out of sync with master.  Time difference of 52782ms > max allowed of 30000ms
```

报错原因：报错节点与 HBase Master 节点时间差超过配置时间。

解决方案：

(1) 检查时钟同步服务是否开启，若未开启或安装，进行相应开启或安装操作。

(2) 进行报错节点与 HMaster 节点手动时钟同步。

2. org.apache.hadoop.hbase.security.AccessDeniedException 权限拒绝异常

访问 HBase 进行表创建、数据读取等，由于没有权限，会报该错。例如创建表没有权限，报错信息如下所示：

```
org.apache.hadoop.hbase.security.AccessDeniedException:
org.apache.hadoop.hbase.security.AccessDeniedException: Insufficient permissions for user 'mingtong' (action=create)
at
org.apache.ranger.authorization.hbase.AuthorizationSession.publishResults(AuthorizationSession.java:261)
at
org.apache.ranger.authorization.hbase.RangerAuthorizationCoprocessor.authorizeAccess(RangerAuthorizationCoprocessor.java:595)
at
org.apache.ranger.authorization.hbase.RangerAuthorizationCoprocessor.requirePermission(RangerAuthorizationCoprocessor.java:664)
at
org.apache.ranger.authorization.hbase.RangerAuthorizationCoprocessor.preCreateTable(RangerAuthorizationCoprocessor.java:769)
at
org.apache.ranger.authorization.hbase.RangerAuthorizationCoprocessor.preCreateTable(RangerAuthorizationCoprocessor.java:496)
at
org.apache.hadoop.hbase.master.MasterCoprocessorHost$11.call(MasterCoprocessorHost.java:222)
at
org.apache.hadoop.hbase.master.MasterCoprocessorHost.execOperation(MasterCoprocessorHost.java:1146)
```

```
at
org.apache.hadoop.hbase.master.MasterCoprocessorHost.preCreateTable(MasterCoprocessorHost.java:218)
at org.apache.hadoop.hbase.master.HMaster.createTable(HMaster.java:1603)
at
org.apache.hadoop.hbase.master.MasterRpcServices.createTable(MasterRpcServices.java:462)
at
org.apache.hadoop.hbase.protobuf.generated.MasterProtos$MasterService$2.callBlockingMethod(MasterProtos.java:57204)
at org.apache.hadoop.hbase.ipc.RpcServer.call(RpcServer.java:2127)
at org.apache.hadoop.hbase.ipc.CallRunner.run(CallRunner.java:107)
at org.apache.hadoop.hbase.ipc.RpcExecutor.consumerLoop(RpcExecutor.java:133)
at org.apache.hadoop.hbase.ipc.RpcExecutor$1.run(RpcExecutor.java:108)
at java.lang.Thread.run(Thread.java:748)

at sun.reflect.GeneratedConstructorAccessor15.newInstance(Unknown Source)
at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
```

报错原因：用户无访问权限。

解决方案：

(1) 根据报错原因，分析应该解决的权限问题，对于 HBase 表或命名空间权限问题，使用对应的权限用户（表/命名空间的创建用户，系统表、hbase 及 default 命名空间默认为 hbase 用户），及“grant”授权命令进行赋权操作，进行问题解决。

(2) 对于 HBase 作业类操作，分析是 HBase 表权限或命名空间权限问题，还是 HDFS 权限问题，若为 HBase 权限相关问题，使用上一步解决方法进行解决，若为 HDFS 问题，使用 hdfs chmod 命令修改文件目录的权限。

3. 运行 hbase shell 输入 list 等基本语句报错

报错异常信息如下：

```
ERROR: Can't get master address from ZooKeeper; znode data == nullHere is some help for this
command:List all tables in hbase. Optional regular expression parameter could
be used to filter the output. Examples:
hbase> list
hbase> list 'abc.*'
hbase> list 'ns:abc.*'
hbase> list 'ns:.*'
```

报错原因：

时间不同步，HBase 集群时间不同步可能会导致各种问题。

解决方案：

(1) 查看集群各节点时钟同步服务进程状态及时间，并进行时间同步操作。

(2) 不是上述原因造成的话，可查找集群日志报错信息，寻求对应的解决方法。

4. client.HConnectionManager$HConnectionImplementation: Can”t get connection to ZooKeeper: KeeperErrorCode = ConnectionLoss for /hbase

报错原因：未关闭防火墙

解决方案：关闭防火墙

下面以 H3Linux 为例关闭防火墙，方法如下所示：

systemctl stop firewalld.service #停止 firewall

systemctl disable firewalld.service #禁止 firewall 开机启动

firewall-cmd -state #查看默认防火墙状态（关闭后显示 notrunning，开启后显示 running）

5. The table xxx does not exist in meta but has a znode. run hbck to fix inconsistencies.

报错原因：HBase 表 xxx 的表描述相关信息在 hbase:meta 表不存在，但在 Zookeeper 中存在该 HBase 表信息。

解决方案：以 table hbasetest 为例：

- 若开启 Kerberos：
  a. 执行 hbase zkcli 命令进入 zookeeper client 模式
  b. 执行 ls /hbase-secure/table 命令查看是否有 hbasetest 这个表
  c. 使用 rmr /hbase-secure/table/hbasetest 命令删除表
  d. 重启 HBase
- 若未开启 Kerberos：
  a. 执行 hbase zkcli 命令进入 zookeeper client 模式
  b. 执行 ls /hbase-unsecure/table 命令查看是否有 hbasetest 这个表
  c. 使用 rmr /hbase-unsecure/table/hbasetest 命令删除表
  d. 重启 HBase

6. org.apache.hadoop.hbase.IPc.ServerNotRunningYetException: Server is not running yet

报错原因：Hadoop 处于 safe mode

解决方案：

(1) 查看 hadoop 当前启动状态是否为 safe mode

hadoop dfsadmin -safemode get

(2) 退出安全模式

hadoop dfsadmin -safemode leave

7. 磁盘故障，RegionServer 重启之后，Region offline

可能原因：磁盘故障，在 Region 重新分配过程中，被分配的 RegionServer 重启，Region 分配失败，Region offline。具体原因需结合排查磁盘故障原因，以及 RegionServer 被重启的原因进行分析。

解决方案：排除故障原因，恢复磁盘故障。当磁盘故障恢复，HBase 健康状态恢复，执行 hbase hbck -fixAssignments 使 Region 恢复上线。假如故障无法消除，收集过程日志并联系相应负责人。

8. 查看 HBase 组件日志，发现连接 Zookeeper 组件失败，若定位 HBase 组件没有问题，可能是 Zookeeper 服务存在问题

可能原因：Zookeeper 比较常见的问题就是因其数据存储目录空间不足，导致 Zookeeper Server 不能够正常提供服务，从而导致其他依赖于 Zookeeper 的服务如 HBase 组件连接 Zookeeper 失败。

解决办法：查看 Zookeeper 的数据存储目录所在的分区，清理该分区内无用的文件，释放空间，然后重启 Zookeeper 服务即可。

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.2 组件架构 1-1
1.2.1 Phoenix 架构图 1-1
1.2.2 Phoenix 数据映射特征 1-2
1.3 应用场景 1-2
2 快速入门 2-1
2.1 Phoenix-sqlline 使用说明 2-1
2.2 Phoenix QueryServer 安装指导 2-1
2.2.1 安装 HBase 时添加 2-1
2.2.2 安装 HBase 之后添加 2-2
2.3 快速使用指导 2-2
2.3.1 Kerberos 环境下的用户身份认证 2-2
2.3.2 使用 Phoenix 2-3
3 使用指南 3-1
3.1 组件特性 3-1
3.1.1 Secondary Indexing 二级索引 3-1
3.1.2 View 视图 3-1
3.1.3 Salted Tables 加盐表 3-1
3.1.4 User-defined functions 用户定义函数 3-1
3.2 组件特性使用 3-2
3.2.1 Secondary Indexing 二级索引 3-2
3.2.2 User-defined functions 用户定义函数 3-3
3.2.3 Statistics Collection 统计信息收集 3-5
3.2.4 Bulk CSV Data Loading 加载 CSV 数据 3-5
3.2.5 Salted Tables 3-5
3.2.6 View 视图 3-6
4 开发指南 4-1
4.1 二级索引性能对比 4-1
4.1.1 查询两个相同的表 4-1
4.1.2 Global Indexes（全局索引） 4-2
4.1.3 CoveredIndexex（覆盖索引） 4-2

4.1.4 Local Indexes（本地索引）4-4
4.2 结合 HBase 使用 4-4
4.3 非 Kerberos 环境下的开发 4-6
4.3.1 Phoenix api 开发的通用步骤 4-6
4.3.2 添加 pom 依赖 4-6
4.3.3 log4j.properties 配置 4-11
4.3.4 创建表 4-11
4.3.5 插入数据 4-13
4.3.6 查找数据 4-15
4.3.7 删除数据 4-17
4.3.8 二级索引 4-19
4.4 Kerberos 环境下的开发 4-21
4.4.1 通过 zookeeper 连接 phoenix 4-21
4.4.2 通过 queryserver 连接 4-24
5 版本新增特性 5-1
6 常见问题解答 6-1

1 组件简介

1.1 组件概述

Phoenix 是基于 HBase 的 SQL 中间件产品。对于熟悉关系型数据库的开发人员来说，通过 Phoenix 可以像使用 MySQL 等关系型数据库一样使用 HBase 中的数据表。

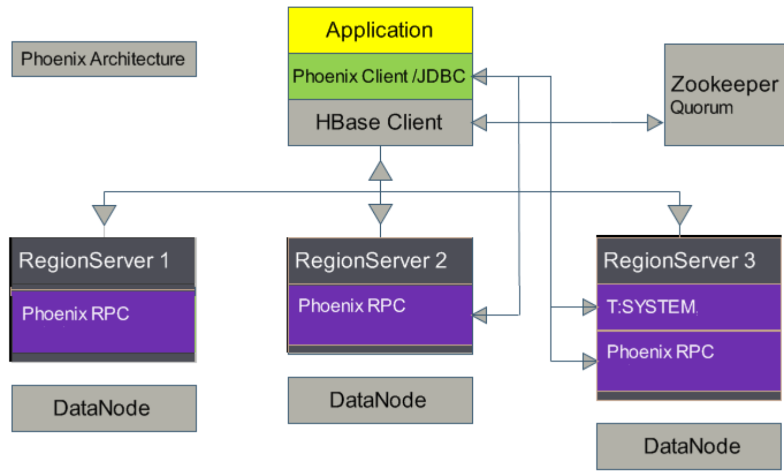
Phoenix 完全使用 Java 编写，作为 HBase 内嵌的 JDBC 驱动。Phoenix 查询引擎会将 SQL 查询转换为一个或多个 HBase 扫描，并编排执行以生成标准的 JDBC 结果集。由于 Phoenix 直接使用 HBase API、协同处理器与自定义过滤器，对于简单查询来说，其性能量级是毫秒，对于百万级别的行数来说，其性能量级是秒。Phoenix 除了和 HBase 对接具有良好的性能外，其也能和 Hive、Spark、Pig、Flume、MapReduce 等组件进行配合使用。

1.2 组件架构

1.2.1 Phoenix 架构图

Phoenix 作为 HBase 的中间件，其架构依附于 HBase。具体架构如图 1-1 所示。

图1-1 Phoenix 架构图



Phoenix 以 JDBC 驱动方式嵌入到 HBase 中，在部署时只需要把相关的 jar 包放到 HBase 的 lib 目录下即可。在每个 RS 节点上，都会有一个 Phoenix 的 coprocessor 来处理每个表，应用端通过 Phoenix 的客户端以 SQL 的方式与 HBase 打交道。

Phoenix 的 SQL 是基于一系列的 Scan 操作来实现的。Scan 是 HBase 的批量扫描过程，这一系列的操作会分发给 RegionServer，然后 RegionServer 通过 Coprocessor 来完成，注意这里的 Coprocessor 的类型为 RegionObserve，通过 RegionObserve 在 postScannerOpenHook 中将 RegionScanner 替换成支持聚合操作的定制化 Scanner，在真正执行聚合时，会通过自定义的 Scan 属性传递给 RegionScanner，也可加入一些过滤规则，减少返回的结果。

1.2.2 Phoenix 数据映射特征

HBase 是非关系型数据库，为了简化其操作，Phoenix 主要是将 HBase 的数据模型映射为关系型。HBase 和 Phoenix 表映射的基本对应关系为：

- HBase 表中的 RowKey 映射为 Phoenix 表中的 Primary Key。
- HBase 表中的 Cloumn family.qualifier（假设列族为 a，列为 b）映射为 Phoenix 表中的 Field(a.b)。

1.3 应用场景

Phoenix 作为 HBase 的插件，可以把 Phoenix 看成一种代替 HBase 语法的一个工具。通过 Phoenix 可以像使用 MySQL 等关系型数据库一样使用 HBase 中的数据表，并扩展了功能，包括二级索引、视图等，极大的方便了操作 HBase。其主要用于以下场景：

- Shell 端使用 SQL 对 HBase 访问。
- 程序开发使用标准的 JDBC API 来代替 HBase 的 API，访问更加灵活。
- 在大数据量的简单查询场景有着独有的优势。

2 快速入门

Phoenix 被看成一种代替 HBase 语法的一个工具，并扩展 HBase 的功能，包括二级索引、视图等，极大地方便了操作 HBase。Phoenix 完全依赖于 HBase 组件，HBase 的正常工作是 Phoenix 使用的前提。

2.1 Phoenix-sqlline使用说明

Phoenix 是对 HBase 使用的扩展，在大数据平台的 HBase 集群内任意节点上执行 phoenix-sqlline localhost 进入 shell 命令行，如果下列操作可以正常执行，则表明安装成功。

(1) 执行数据读写

- 帮助手册：!help
- 查看所有表：!tables
- 更新数据：upsert 代替 insert

(2) 退出客户端：!quit

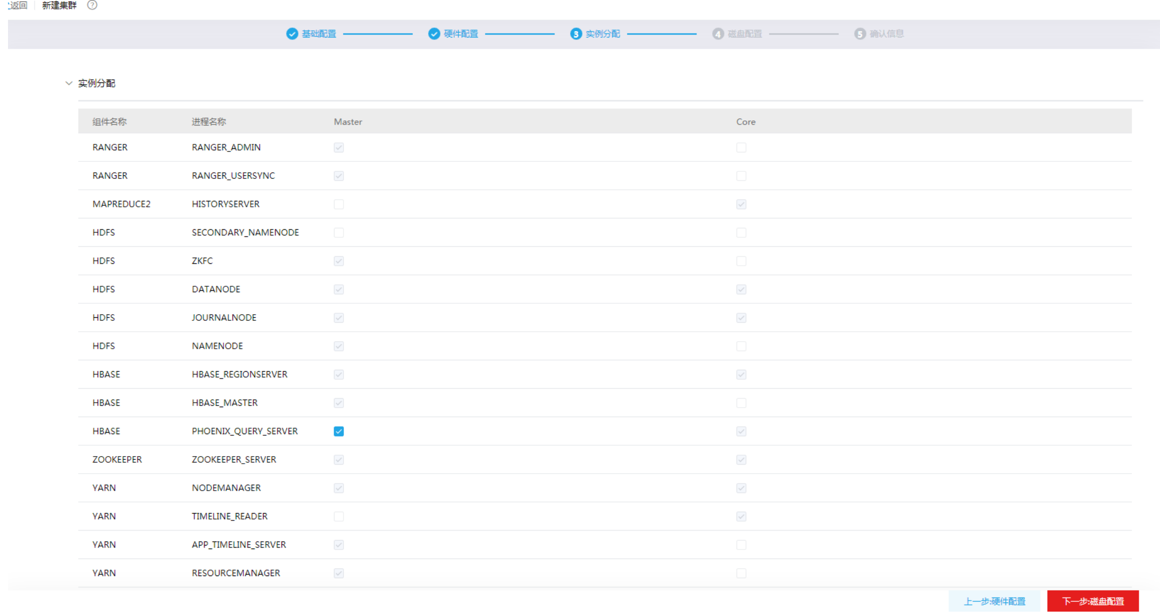
2.2 Phoenix QueryServer安装指导

Queryserver 可以支持 jdbc api 连接过程中使用 thin:url:*方式访问，如果同时部署多个，配合均衡器使用便可以达到 loadbalance 的效果。

2.2.1 安装 HBase 时添加

在大数据平台中，将 Phoenix 集成到了 HBase 组件中，安装 HBase 组件时默认安装了 Phoenix Query Server，如图 2-1 所示。

图2-1 Phoenix 安装



2.2.2 安装 HBase 之后添加

(1) 在 HBase 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 2-2 所示。

a. 选择进程及主机。

在选择进程项的下拉列表中选择 Phoenix Query Server，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

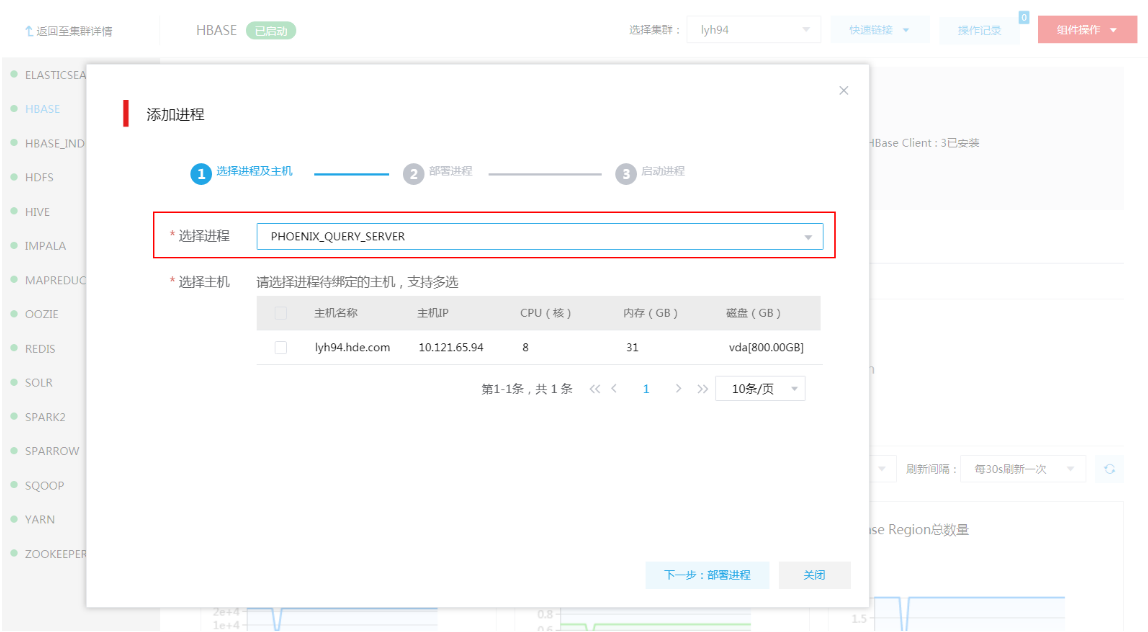
b. 部署进程。

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程。

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图2-2 添加 Phoenix Query Server



(3) Phoenix Query Server 添加完成之后，在组件详情页面[部署拓扑]页签中可以看到 Phoenix Query Server 安装数量的变化。

2.3 快速使用指导

2.3.1 Kerberos 环境下的用户身份认证

如果集群开启 Kerberos，在使用组件之前需要进行用户身份认证，才可以正常使用。Phoenix 是 HBase 的中间件，Phoenix 组件需要 HBase 用户进行身份认证后才能使用，HBase 用户身份认证的相关操作请参见 HBase 手册。

2.3.2 使用 Phoenix

- 进入 Phoenix CLI

  [root@hbase01 yum.repos.d]# su hbase

  [hbase@hbase01 yum.repos.d]$ phoenix-sqlline

- 创建表并插入数据，示例如下：

  jdbc:phoenix:localhost> create table if not exists Person (IDCardNum INTEGER not null primary key, Name varchar(20),Age INTEGER);

  jdbc:phoenix:localhost> upsert into Person (IDCardNum,Name,Age) values (100,'Jeams',12);

  jdbc:phoenix:localhost> upsert into Person (IDCardNum,Name,Age) values (101,'Kobi',15);

  jdbc:phoenix:localhost> upsert into Person (IDCardNum,Name,Age) values (103,'TT',22);

  jdbc:phoenix:localhost> select * from person;

  查看新创建的表，如图 2-3 所示。

图2-3 查看新创建的表

```
0: jdbc:phoenix:localhost>  select * from person;
+------------+--------+------+
| IDCARDNUM  |  NAME  | AGE  |
+------------+--------+------+
| 100        | Jeams  | 12   |
| 101        | Kobi   | 15   |
| 103        | TT     | 22   |
+------------+--------+------+
3 rows selected (0.169 seconds)
```

- 修改并更新表，示例如下：

  jdbc:phoenix:localhost>alter table person add sex varchar(10);

  jdbc:phoenix:localhost>upsert into person(IDCARDNUM,sex)values(100,'male');

  jdbc:phoenix:localhost>upsert into person(IDCARDNUM,sex)values(101,'male');

  jdbc:phoenix:localhost>upsert into person(IDCARDNUM,name,age,sex)values(102,'Judy',48,'female');

  jdbc:phoenix:localhost>upsert into person(IDCARDNUM,sex)values(103,'male');

  jdbc:phoenix:localhost>upsert into person(IDCARDNUM,name,sex)values(104,'BoBo','male');

  jdbc:phoenix:localhost>select * from person;

  查看更新后的表，如图 2-4 所示。

图2-4 查看更新后的表

```
0: jdbc:phoenix:localhost> select * from person;
+------------+--------+-------+---------+
| IDCARDNUM  |  NAME  |  AGE  |   SEX   |
+------------+--------+-------+---------+
| 100        | Jeams  | 12    | male    |
| 101        | Kobi   | 15    | male    |
| 102        | Judy   | 48    | female  |
| 103        | TT     | 22    | male    |
| 104        | BoBo   | null  | male    |
+------------+--------+-------+---------+
5 rows selected (0.091 seconds)
0: jdbc:phoenix:localhost>
```

- Phoenix 可以支持 where、group by、case when 等查询条件，示例如下：

  jdbc:phoenix:localhost>upsert into person(IDCARDNUM,name,age,sex)values(105,'Bo',39,'male');

  jdbc:phoenix:localhost>upsert into person(IDCARDNUM,name,sex)values(106,'Tk','male');

  jdbc:phoenix:localhost> select sex ,count(sex) as num,avg(age) as avg_age from person where age >20 group by sex;

  条件查询结果显示如图 2-5 所示。

图2-5 条件查询结果显示

```
0: jdbc:phoenix:localhost> select sex ,count(sex) as num,avg(age) as avg_age fro
m person where age >20 group by sex;
+---------+------+-----------+
|   SEX   | NUM  |  AVG_AGE  |
+---------+------+-----------+
| female  | 1    | 48        |
| male    | 2    | 30.5      |
+---------+------+-----------+
2 rows selected (4.137 seconds)
0: jdbc:phoenix:localhost>
```

jdbc:phoenix:localhost> select IDCARDNUM,name,(case when age is null then 100 else age end) as nage from person;

条件查询结果显示如图 2-6 所示。

图2-6 条件查询结果显示

```
0: jdbc:phoenix:localhost> select IDCARDNUM,name,(case when age is null then 100
 else age end) as nage from person;
+------------+--------+-------+
| IDCARDNUM  |  NAME  | NAGE  |
+------------+--------+-------+
| 100        | Jeams  | 12    |
| 101        | Kobi   | 15    |
| 102        | Judy   | 48    |
| 103        | TT     | 22    |
| 104        | BoBo   | 100   |
| 105        | Bo     | 39    |
| 106        | Tk     | 100   |
+------------+--------+-------+
7 rows selected (0.095 seconds)
0: jdbc:phoenix:localhost>
```

- 删除表数据，示例如下：

  jdbc:phoenix:localhost>delete from person where age is null and IDCARDNUM >105;

3 使用指南

可以把 Phoenix 看成一种代替 HBase 语法的一个工具。通过 Phoenix 可以像使用 MySQL 等关系型数据库一样使用 HBase 中的数据表，并扩展了功能，包括二级索引、视图等，极大地方便了操作 HBase。

3.1 组件特性

3.1.1 Secondary Indexing 二级索引

在 HBase 中，只有一个单一的按照字典序排序的 RowKey 索引，当使用 RowKey 来进行数据查询的时候速度较快，但是如果不使用 RowKey 来查询的话就会使用 filter 来对全表进行扫描，很大程度上降低了检索性能。而 Phoenix 提供了二级索引技术来应对这种使用 RowKey 之外的条件进行检索的场景。二级索引技术有本地索引、全局索引、函数索引和覆盖索引等。

3.1.2 View 视图

Phoenix 现在支持标准 SQL 视图语法，以允许多个虚拟表共享同一底层物理 HBase 表。

Phoenix 创建的视图是只读的，一般只可以进行查询操作。删除 Phoenix 中的表时，会将 HBase 对应的表删掉，但是删除 Phoenix 视图，却不会影响 HBase 中的表。

3.1.3 Salted Tables 加盐表

HBase 中的行是按照 RowKey 的字典顺序排序的，当 RowKey 自增时会将数据集中到少数 region 中，此时大量的数据访问会导致 region 超出承受能力，产生热点问题。Phoenix 通过 salt 的方式可以使数据均匀的分布在 RegionServer 中。

3.1.4 User-defined functions 用户定义函数

用户可以创建临时或永久的用户自定义函数。这些用户自定义函数可以像内置函数一样被调用。临时函数是针对特定的会话或连接，对其他会话或连接不可见。永久函数的元信息会被存储在 SYSTEM.FUNCTION 系统表中，对任何会话或连接均可见。Pheonix 使用 HBase 的动态类加载 jar，把自定义的函数打成 jar 包后放到 HDFS 对应的目录即可，不需要重启组件。

3.2 组件特性使用

说明

- Phoenix 只是 HBase 的中间件，不涉及扩容和缩容。
- Phoenix 是 HBase 的中间件，Phoenix 对表的操作权限受 HBase 用户权限的控制。
- Phoenix 只是中间件，其数据存储在 HBase 上，迁移过程和 HBase 一致。
- Phoenix 客户端命令只能在 HBase 集群之内使用，暂不支持在 HBase 集群之外使用。

3.2.1 Secondary Indexing 二级索引

在 HBase 中，只有一个单一的按照字典序排序的 RowKey 索引，当使用 RowKey 来进行数据查询的时候速度较快，但是如果不使用 RowKey 来查询的话就会使用 filter 对全表进行扫描，很大程度上降低了检索性能。而 Phoenix 提供了二级索引技术来应对这种使用 RowKey 之外的条件进行检索的场景。

1. Covered Indexes（覆盖索引）

下面以 student 表为例进行二级索引的使用示范。

(1) 创建 student 表：

```
jdbc:phoenix:localhost> create table student(id integer primary key,name varchar(20) ,age integer ,addr varchar(20));
```

(2) 插入数据：

```
jdbc:phoenix:localhost> upsert into student values(21,'xi',20,'henan');
jdbc:phoenix:localhost> upsert into student values(1,'xiaoxiao',23,'henan');
```

(3) 覆盖索引只通过索引就能返回所要查询的数据，所以索引的列必须包含所查询的列(SELECT 的列和 WHRER 的列)，示例如下：

```
jdbc:phoenix:localhost> create index stu_index on student(name) include(addr);
jdbc:phoenix:localhost> select addr from student where name='xiaoxiao';
```

(4) 当通过 name 来查询 addr 时就可以直接从索引上取回数据而无需再去数据表中查询数据，查询效率有很大提升。通过 explain 命令可以查看查询计划，如图 3-1、图 3-2 所示。

图3-1 未使用索引

```
0: jdbc:phoenix:localhost> explain select addr from student;
+------------------------------------------------------------+
|                            PLAN                            |
+------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL SCAN OVER STU_INDEX |
+------------------------------------------------------------+
1 row selected (0.148 seconds)
0: jdbc:phoenix:localhost>
```

图3-2 使用索引

```
0: jdbc:phoenix:localhost> explain select addr from student where name='xi';
+------------------------------------------------------------------------+
|                                 PLAN                                   |
+------------------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER STU_INDEX ['xi'] |
+------------------------------------------------------------------------+
1 row selected (0.037 seconds)
0: jdbc:phoenix:localhost>
```

2. Functional Indexes（函数索引）

函数索引不局限于列，可以以任意的表达式来创建索引，当在查询时用到了这些表达式时就直接通过索引返回表达式结果，示例如下：

jdbc:phoenix:localhost>create index stu_index on student (upper(name || ' ' || addr));

jdbc:phoenix:localhost>select upper(name || ' ' || addr) from student;

jdbc:phoenix:localhost>drop index stu_index on student;

3. Global Indexes（全局索引）

全局索引适用于多读少写的场景，在写操作上会给性能带来极大的开销，因为所有的更新和写操作（DELETE、UPSERT VALUES 和 UPSERT SELECT）都会引起索引的更新，在读数据时，Phoenix 将通过索引表来达到快速查询的目的，示例如下：

jdbc:phoenix:localhost>create index stu_index on student(name);

jdbc:phoenix:localhost>select id,name from student where name='xiaoxiao';

4. Local Indexes（本地索引）

本地索引适用于多写少读，空间有限的场景，和全局索引一样，Phoneix 在查询时会自动选择是否使用本地索引。使用本地索引，为避免进行写操作所带来的网络开销，索引数据和表数据都存放在相同的服务器中。与全局索引不同的是，所有的本地索引都单独存储在同一张共享表中，由于无法预先确定 region 的位置，所以在读取数据时会检查每个 region 上的数据因而带来一定性能开销。示例如下：

jdbc:phoenix:localhost> create local index stu_index on student(name);

jdbc:phoenix:localhost>select addr from student where name='Jeams';

3.2.2 User-defined functions 用户定义函数

用户可以创建临时或永久的用户自定义函数。这些用户自定义函数可以像内置函数一样被调用。临时函数针对特定的会话或连接，对其他会话或连接不可见。永久函数的元信息会被存储在 SYSTEM.FUNCTION 系统表中，对任何会话或连接均可见。

1. 编写自定义函数

用户自定义函数需要继承 ScalarFunction 类，实现 getDataType 和 evaluate 方法。自定义函数打包后需要放在 HDFS 目录下。

自定义函数示例如下（本例是把该函数打成 Jar 包，上传到 hdfs 的/apps/hbase/data/lib 路径下，并赋予 hbase 权限）：

```
public class PhoenixUDF  extends ScalarFunction {
   public PhoenixUDF(List list){
      super(list);
   }
   @Override
   public String getName(){
      return "UDFFlag";
   }
   public PDataType getDataType(){
      return PVarchar.INSTANCE;
   }
  public boolean evaluate(Tuple tuple, ImmutableBytesWritable ptr){
     Expression arg = getChildren().get(0);
     if (!arg.evaluate(tuple, ptr)) {
        return false;
     }
     int targetOffset = ptr.getLength();
     if (targetOffset == 0) {
        return true;
     }
     byte[] udf_flag = Bytes.toBytes("flag");
     byte[] target =new byte[targetOffset+udf_flag.length];
     System.arraycopy(ptr.get(),ptr.getOffset(), target, 0, targetOffset);
     System.arraycopy(udf_flag,0, target, targetOffset, udf_flag.length);
     ptr.set(target);
     return true;
   }
}
```

2. 使用自定义函数

(1) 进入 Phoenix CLI

```
[root@hbase01 yum.repos.d]# su hbase
[hbase@hbase01 yum.repos.d]$ phoenix-sqlline
```

(2) 创建自定义函数并使用

```
jdbc:phoenix:localhost> create function UDFFlag(varchar) returns varchar as
'com.phoenixudf.PhoenixUDF' using jar '/apps/hbase/data/lib/phoenixudf-1.0.jar';
jdbc:phoenix:localhost> select NAME from EMPLOYER_INFO;
jdbc:phoenix:localhost> select UDFFlag(NAME) from EMPLOYER_INFO;
```

查询结果如图 3-3 所示：

图3-3 自定义函数查询结果

```
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> select NAME from EMPI
+-------+
| NAME  |
+-------+
| Sary  |
+-------+
1 row selected (0.081 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> select UDFFlag(NAME)
+----------------+
| UDFFlag(NAME)  |
+----------------+
| Saryflag       |
+----------------+
```

3.2.3 Statistics Collection 统计信息收集

通过 UPDATE STATISTICS 语句可以更新表的统计信息，以提高查询性能。

jdbc:phoenix:localhost> update statistics student;

3.2.4 Bulk CSV Data Loading 加载 CSV 数据

可以通过 psql 命令以单线程的方式加载 CSV 数据到 Phoenix 表，数据量少的情况下适用。

(1) 创建 persondata.csv 文件，文件内容如下

12345,John,Doe

67890,Mary,Poppins

(2) 在 Phoenix 中创建表

jdbc:phoenix:localhost> create table person (my_pk bigint not null,first_name varchar(50),last_name varchar(50) constraint pk primary key (my_pk));

(3) 将数据导入 Phoenix（表名必须大写，localhost:2181 对应 Zookeeper 的主机名和端口号）

[root@node1 phoenix]# phoenix-psql -t PERSON localhost:2181 persondata.csv;

(4) 在 Phoenix 中查看数据

jdbc:phoenix:localhost> select * from person;

3.2.5 Salted Tables

HBase 中的行是按照 RowKey 的字典顺序排序的，当 RowKey 自增时会将数据集中到少数 region 中，此时大量的数据访问会导致 region 超出承受能力，产生热点问题。Phoenix 通过 salt 的方式可以使数据均匀的分布在 RegionServer 中。

在创建表时通过指定“SALT_BUCKETS”的方式声明表为 salted table。SALT_BUCKETS 是 1 到 256 的数值，SALT_BUCKETS 越大，数据分布越分散。

创建 salted table 的示例如下：

jdbc:phoenix:localhost>create table staff(id varchar primary key,name varchar) SALT_BUCKETS=20;

3.2.6 View 视图

Phoenix 支持标准 SQL 视图语法，允许多个虚拟表共享同一底层物理 HBase 表。

Phoenix 创建的视图是只读的，一般只可以进行查询操作。删除 Phoenix 中的表时，会将 HBase 对应的表删掉，但是删除 Phoenix 视图，却不会影响 HBase 中的表。

以下是视图的使用方式：

(1) 在 Phoenix table 的基础上创建视图

```
jdbc:phoenix:localhost> create table staff(id varchar primary key,name varchar);
jdbc:phoenix:localhost> create view my_staff as select * from staff;
jdbc:phoenix:localhost>select * from my_staff;
```

(2) 在视图之上建立视图

```
jdbc:phoenix:localhost> create view view_on_mystaff as select * from my_staff where name='tom';
jdbc:phoenix:localhost>select * from view_on_mystaff;
```

(3) 不断往 HBase 中添加数据，随着数据的增长，在 Phoenix 建的视图中可以看到数据的条数在同步增加

```
jdbc:phoenix:localhost>upsert into staff (id,name) values ('105','Ki');
jdbc:phoenix:localhost>upsert into staff (id,Name) values ('106','Jeams');
jdbc:phoenix:localhost> select * from my_staff;
```

(4) 删除视图

```
jdbc:phoenix:localhost>drop view my_staff;
```

4 开发指南

4.1 二级索引性能对比

4.1.1 查询两个相同的表

向两个表中插入10万条相同的数据，如图4-1。

图4-1 查看表数据条数

```
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> select count(1) from PHOENIX_TABLE_INDEX;
+-----------+
| COUNT(1)  |
+-----------+
| 100000    |
+-----------+
1 row selected (0.424 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> select count(1) from PHOENIX_TABLE_NO_INDEX;
+-----------+
| COUNT(1)  |
+-----------+
| 100000    |
+-----------+
```

查看表部分数据，如图4-2、图4-3。

图4-2 查看表部分数据-有二级索引

```
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT * FROM PHOENIX_TABLE_INDEX limit 10;
+-----+-------+
| ID  | NAME  |
+-----+-------+
| 0   | wu0   |
| 1   | wu1   |
| 2   | wu2   |
| 3   | wu3   |
| 4   | wu4   |
| 5   | wu5   |
| 6   | wu6   |
| 7   | wu7   |
| 8   | wu8   |
| 9   | wu9   |
+-----+-------+
10 rows selected (0.02 seconds)
```

图4-3 查看表部分数据-无二级索引

```
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT * FROM PHOENIX_TABLE_NO_INDEX limit 10;
+-----+-------+
| ID  | NAME  |
+-----+-------+
| 0   | wu0   |
| 1   | wu1   |
| 2   | wu2   |
| 3   | wu3   |
| 4   | wu4   |
| 5   | wu5   |
| 6   | wu6   |
| 7   | wu7   |
| 8   | wu8   |
| 9   | wu9   |
+-----+-------+
10 rows selected (0.031 seconds)
```

4.1.2 Global Indexes（全局索引）

创建全局索引，其中 PHOENIX_TABLE_INDEX 和 PHOENIX_TABLE_NO_INDEX 的表结构一样，都是有 ID 和 NAME 两个字段。在 NAME 字段创建全局索引，如图 4-4。

图4-4 创建全局索引

```
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> create index nameindex on PHOENIX_TABLE_INDEX(NAME);
100,000 rows affected (6.401 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur>
0: jdbc:phoenix:localhost:2181:/hbase-unsecur>
0: jdbc:phoenix:localhost:2181:/hbase-unsecur>
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> EXPLAIN SELECT NAME FROM PHOENIX_TABLE_INDEX WHERE NAME='wu100';
+-----------------------------------------------------------------------------------+
|                                       PLAN                                        |
+-----------------------------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER NAMEINDEX ['wu100']    |
|     SERVER FILTER BY FIRST KEY ONLY                                               |
+-----------------------------------------------------------------------------------+
2 rows selected (0.025 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT NAME FROM PHOENIX_TABLE_INDEX WHERE NAME='wu100';
+---------+
|  NAME   |
+---------+
| wu100   |
+---------+
1 row selected (0.026 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> EXPLAIN SELECT NAME FROM PHOENIX_TABLE_NO_INDEX WHERE NAME='wu100';
+-------------------------------------------------------------------------------------+
|                                        PLAN                                         |
+-------------------------------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL SCAN OVER PHOENIX_TABLE_NO_INDEX    |
|     SERVER FILTER BY NAME = 'wu100'                                                 |
+-------------------------------------------------------------------------------------+
2 rows selected (0.011 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT NAME FROM PHOENIX_TABLE_NO_INDEX WHERE NAME='wu100';
+---------+
|  NAME   |
+---------+
| wu100   |
+---------+
1 row selected (1.143 seconds)
```

通过上图可以看出，使用索引查询时使用的是 RANGE SCAN 策略，用时 0.026 秒，未使用索引的表使用 FULL SCAN 策略，用时 1.143 秒。二者相差 43 倍。

4.1.3 CoveredIndexex（覆盖索引）

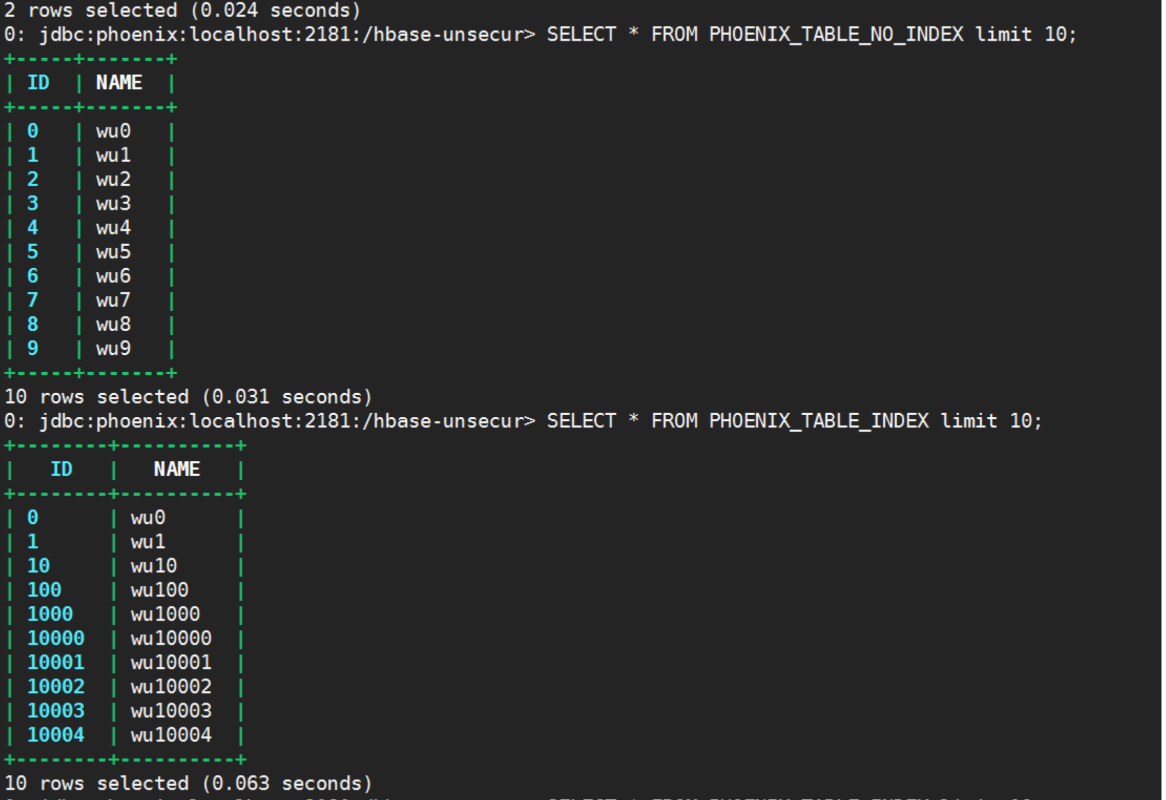
为了对比明确，在测试覆盖索引之前，我们首先删除之前创建的全局索引，然后再创建覆盖索引。操作过程如图 4-5。

图4-5 创建覆盖索引

```
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> DROP index nameindex on PHOENIX_TABLE_INDEX;
No rows affected (3.206 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> create index coveredindex on PHOENIX_TABLE_INDEX(NAME) include(ID);
100,000 rows affected (7.3 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> EXPLAIN SELECT NAME,ID FROM PHOENIX_TABLE_INDEX WHERE NAME='wu100';
+--------------------------------------------------------------------------------+
|                                      PLAN                                      |
+--------------------------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER COVEREDINDEX ['wu100'] |
|     SERVER FILTER BY FIRST KEY ONLY                                            |
+--------------------------------------------------------------------------------+
2 rows selected (1.035 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> EXPLAIN SELECT NAME,ID FROM PHOENIX_TABLE_NO_INDEX WHERE NAME='wu100';
+-------------------------------------------------------------------------------+
|                                      PLAN                                     |
+-------------------------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL SCAN OVER PHOENIX_TABLE_NO_INDEX |
|     SERVER FILTER BY NAME = 'wu100'                                           |
+-------------------------------------------------------------------------------+
2 rows selected (0.011 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT NAME,ID FROM PHOENIX_TABLE_INDEX WHERE NAME='wu100';
+--------+------+
|  NAME  |  ID  |
+--------+------+
| wu100  | 100  |
+--------+------+
1 row selected (0.045 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT NAME,ID FROM PHOENIX_TABLE_NO_INDEX WHERE NAME='wu100';
+--------+------+
|  NAME  |  ID  |
+--------+------+
| wu100  | 100  |
+--------+------+
1 row selected (1.054 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur>
```

通过上图可以看出，使用覆盖索引查询使用的是基于 COVEREDINDEX 的 RANGE SCAN 策略，用时 0.045 秒，未使用索引的表使用 FULL SCAN 策略，用时 1.054 秒。其中创建覆盖索引后，表数据排序会发生变化。在 PHOENIX_TABLE_INDEX 中，所有字段按前缀排列。具体如图 4-6。

图4-6 查看表数据



```
2 rows selected (0.024 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT * FROM PHOENIX_TABLE_NO_INDEX limit 10;
+-----+-------+
| ID  | NAME  |
+-----+-------+
| 0   | wu0   |
| 1   | wu1   |
| 2   | wu2   |
| 3   | wu3   |
| 4   | wu4   |
| 5   | wu5   |
| 6   | wu6   |
| 7   | wu7   |
| 8   | wu8   |
| 9   | wu9   |
+-----+-------+
10 rows selected (0.031 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT * FROM PHOENIX_TABLE_INDEX limit 10;
+--------+----------+
|   ID   |   NAME   |
+--------+----------+
| 0      | wu0      |
| 1      | wu1      |
| 10     | wu10     |
| 100    | wu100    |
| 1000   | wu1000   |
| 10000  | wu10000  |
| 10001  | wu10001  |
| 10002  | wu10002  |
| 10003  | wu10003  |
| 10004  | wu10004  |
+--------+----------+
10 rows selected (0.063 seconds)
```

4.1.4 Local Indexes（本地索引）

为了保证对比结果的可靠性，在测试本地索引前首先删除之前创建的索引，然后再建立本地索引，操作过程如图 4-7。

图4-7 建立本地索引

```
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> create local index localindex on PHOENIX_TABLE_INDEX(NAME);
100,000 rows affected (6.079 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> explain SELECT NAME FROM PHOENIX_TABLE_INDEX WHERE NAME='wu1000' ;
+---------------------------------------------------------------------------------------------+
|                                            PLAN                                             |
+---------------------------------------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER PHOENIX_TABLE_INDEX [2,'wu1000']  |
|     SERVER FILTER BY FIRST KEY ONLY                                                          |
+---------------------------------------------------------------------------------------------+
2 rows selected (0.023 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> explain SELECT NAME FROM PHOENIX_TABLE_NO_INDEX WHERE NAME='wu1000' ;
+----------------------------------------------------------------------------------+
|                                       PLAN                                       |
+----------------------------------------------------------------------------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL SCAN OVER PHOENIX_TABLE_NO_INDEX  |
|     SERVER FILTER BY NAME = 'wu1000'                                             |
+----------------------------------------------------------------------------------+
2 rows selected (0.011 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT NAME FROM PHOENIX_TABLE_INDEX WHERE NAME='wu1000' ;
+----------+
|  NAME    |
+----------+
| wu1000   |
+----------+
1 row selected (0.019 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur> SELECT NAME FROM PHOENIX_TABLE_NO_INDEX WHERE NAME='wu1000' ;
+----------+
|  NAME    |
+----------+
| wu1000   |
+----------+
1 row selected (0.89 seconds)
0: jdbc:phoenix:localhost:2181:/hbase-unsecur>
```

通过测试结果对比，本地索引查询数据用了 0.019 秒，未建立索引的表查询用时 0.89 秒。二者相差 46 倍。

4.2 结合HBase使用

1. HBase 创建表

在 HBase shell 中创建表并插入数据：

```
hbase(main)> create 'phoenix01','info'
hbase(main)> put 'phoenix01', 'row001','info:id','1001'
hbase(main)> put 'phoenix01', 'row001','info:position','manager'
hbase(main)> put 'phoenix01', 'row002','info:id','1002'
hbase(main)> put 'phoenix01', 'row002','info:position','boss'
hbase(main)> put 'phoenix01', 'row003','info:id','1003'
hbase(main)> put 'phoenix01', 'row003','info:position','staff'
hbase(main)> put 'phoenix01', 'row004','info:id','1004'
hbase(main)> put 'phoenix01', 'row004','info:position','staff'
hbase(main)> put 'phoenix01', 'row005','info:id','1005'
hbase(main)> put 'phoenix01', 'row005','info:position','staff'
```

2. Phoenix 创建映射表

(1) 进入 Phoenix CLI

```
[root@hbase01 yum.repos.d]# su hbase
[hbase@hbase01 yum.repos.d]$ phoenix-sqlline
```

(2) 创建映射表

```
jdbc:phoenix:localhost> create table "phoenix01"("ROW" varchar primary key, "info"."id" varchar,"info"."position" varchar) column_encoded_bytes=0;
```

注意：Phoenix 会将表名、列名自动转成字母大写。如果表名和列名需要为小写字母的，可以对表名、列名加双引号，Phoenix 不再自动转换。

(3) 查看创建的映射表，如图 4-8。

```
jdbc:phoenix:localhost> !table
```

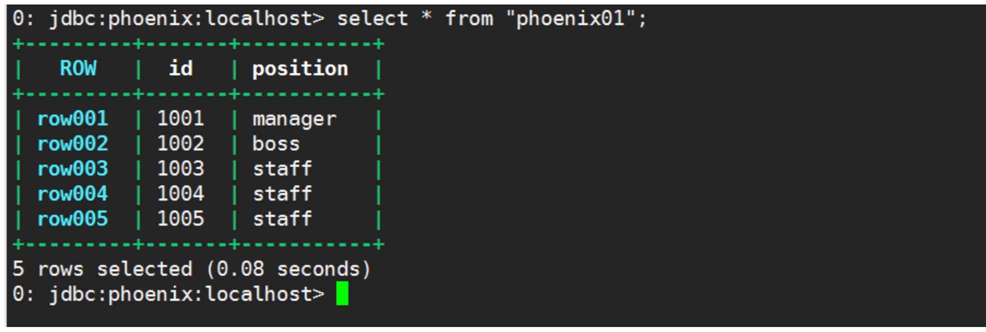
图4-8 查看表

```
0: jdbc:phoenix:localhost> !table
+------------+--------------+-------------+---------------+----------+-----------+
| TABLE_CAT  | TABLE_SCHEM  | TABLE_NAME  |  TABLE_TYPE   | REMARKS  | TYPE_NAME
+------------+--------------+-------------+---------------+----------+-----------+
|            | SYSTEM       | CATALOG     | SYSTEM TABLE  |          |
|            | SYSTEM       | FUNCTION    | SYSTEM TABLE  |          |
|            | SYSTEM       | SEQUENCE    | SYSTEM TABLE  |          |
|            | SYSTEM       | STATS       | SYSTEM TABLE  |          |
|            |              | PERSON      | TABLE         |          |
|            |              | phoenix01   | TABLE         |          |
+------------+--------------+-------------+---------------+----------+-----------+
0: jdbc:phoenix:localhost>
```

(4) 查看映射表中所有数据，如图 4-9。

```
jdbc:phoenix:localhost>select * from "phoenix01";
```

图4-9 查询所有数据



```
0: jdbc:phoenix:localhost> select * from "phoenix01";
+---------+-------+-----------+
|   ROW   |  id   | position  |
+---------+-------+-----------+
| row001  | 1001  | manager   |
| row002  | 1002  | boss      |
| row003  | 1003  | staff     |
| row004  | 1004  | staff     |
| row005  | 1005  | staff     |
+---------+-------+-----------+
5 rows selected (0.08 seconds)
0: jdbc:phoenix:localhost>
```

(5) 条件查询结果显示如图 4-10 所示。

```
jdbc:phoenix:localhost>select * from "phoenix01" where "position"='manager';
```

图4-10 条件查询

```
0: jdbc:phoenix:localhost> select * from "phoenix01" where "position"='manager';
+---------+-------+-----------+
|   ROW   |  id   | position  |
+---------+-------+-----------+
| row001  | 1001  | manager   |
+---------+-------+-----------+
1 row selected (0.115 seconds)
0: jdbc:phoenix:localhost>
```

(6) 更新数据

jdbc:phoenix:localhost>upsert into "phoenix01" ("ROW","id","position")values ('row001','1001','manager1');

(7) 删除表格

jdbc:phoenix:localhost> drop table "phoenix01";

需要注意的是，HBase 中 phoenix01 表也同时被删除。

4.3 非Kerberos环境下的开发

4.3.1 Phoenix api 开发的通用步骤

(1) 创建 MAVEN 的开发环境配置（比如 pom 依赖、log4j 配置）

(2) 将 hbase-site.xml 从/usr/hdp/3.0.1.0-187/hbase/conf/目录下拷贝到 resources 目录下，放在该路径下代码会自行解析，不用显式的在代码里面加载配置，推荐该方式。

(3) 设置登录用户：System.setProperty("HADOOP_USER_NAME", "hbase");

(4) 建立连接访问 Zookeeper 的/hbase-unsecure 的 url

(5) 使用上一步骤的 url 创建连接

(6) 编写 SQL 语句

(7) 创建 Statement

(8) 执行 SQL

(9) 处理结果

4.3.2 添加 pom 依赖

示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">


  <modelVersion>4.0.0</modelVersion>
```

```
<groupId>com.bigdata</groupId>
<artifactId>service_daemons</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>

<name>A Camel Route</name>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <hbase.version>2.1.0</hbase.version>
  <spark.version>2.2.1</spark.version>
  <hadoop.version>2.7.1</hadoop.version>
</properties>



<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>2.7.1</version>
    </dependency>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>2.7.1</version>
  </dependency>
  <!--<dependency>-->
    <!--<groupId>org.apache.hbase</groupId>-->
    <!--<artifactId>hbase-spark</artifactId>-->
    <!--<version>1.1.2.2.6.2.0-147</version>-->
    <!--<scope>provided</scope>-->
  <!--</dependency>-->

  <dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase-server</artifactId>
  <version>${hbase.version}</version>
```

```
</dependency>
 <dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-server</artifactId>
    <version>${hbase.version}</version>
 </dependency>
 <dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-common</artifactId>
    <version>${hbase.version}</version>
 </dependency>
 <dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-client</artifactId>
    <version>${hbase.version}</version>
    <exclusions>
       <exclusion>
          <artifactId>slf4j-log4j12</artifactId>
          <groupId>org.slf4j</groupId>
       </exclusion>
    </exclusions>
 </dependency>

 <dependency>
    <groupId>org.apache.phoenix</groupId>
    <artifactId>phoenix-core</artifactId>
    <version>5.0.0.3.0.0.0-1634</version>
 </dependency>

 <dependency>
    <groupId>org.apache.phoenix</groupId>
    <artifactId>phoenix-queryserver-client</artifactId>
    <version>5.0.0-HBase-2.0</version>
 </dependency>

 <dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
```

```
      <version>3.18.0</version>
    </dependency>

    <dependency>
      <groupId>com.google.protobuf</groupId>
      <artifactId>protobuf-java-util</artifactId>
      <version>3.18.0</version>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>

   <!-- logging -->


   <!-- testing -->

  </dependencies>

  <build>
   <defaultGoal>install</defaultGoal>

   <plugins>
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-compiler-plugin</artifactId>
     <version>3.8.0</version>
     <configuration>
      <source>1.8</source>
      <target>1.8</target>
     </configuration>
    </plugin>
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-resources-plugin</artifactId>
```

```
      <version>3.1.0</version>
      <configuration>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>

    <!-- Allows the example to be run via 'mvn compile exec:java' -->
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.6.0</version>
      <configuration>
        <mainClass>com.bigdata.MainApp</mainClass>
        <includePluginDependencies>false</includePluginDependencies>
      </configuration>
    </plugin>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
        <executions>
          <execution>
            <id>make-jar-with-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
  </plugins>
 </build>

</project>
```

4.3.3 log4j.properties 配置

如果开发过程中使用了 log4j，需要在 Java 工程的 resource 目录下添加如下配置：

```
log4j.rootLogger = INFO,Console

#Console

log4j.appender.Console = org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=%d [%t] %p  %m %n
```

4.3.4 创建表

Phoenix 表的创建示例如下，示例中 System.setProperty("HADOOP_USER_NAME","hbase")代表设置本次访问用户为 hbase，参数 zooks 代表 HBase 的 Zookeeper 集群节点。如果使用主机名代替主机 IP，需要在 hosts 文件中配置相应的 IP 和 hostname。

```
package com.bigdata;
import org.apache.log4j.Logger;
import java.sql.*;
import java.util.Date;

/**
 * Created by bigdata on 2019/3/8.
 */
public class PhoenixCreate {
   public static final Logger logger = Logger.getLogger(PhoenixCreate.class);
   public static Connection conn = null;
   public static Statement stml = null;
   public static ResultSet  rs = null;
   public static void main(String [] args){
      System.setProperty("HADOOP_USER_NAME", "hbase");
      try {
         String zooks =
"node1.hde.bigdata.com,node2.hde.bigdata.com,node3.hde.bigdata.com:/hbase-unsecure";
         conn = DriverManager.getConnection("jdbc:phoenix:"+zooks);
         if (conn != null){
            logger.info("good ,i got the connection");
         }
         stml = conn.createStatement();
         String sql = "create table test_phoenix_api( id integer not null primary key,name varchar)";
         logger.info(sql);
```

```
            Date start_time = new Date();
            logger.info(start_time);
            stml.executeUpdate(sql);
            conn.commit();

            String query = "select * from \" test_phoenix_api\"";
            rs = stml.executeQuery(query);

            ResultSetMetaData meta = rs.getMetaData();
            logger.info("----+----");
            logger.info(meta.getColumnLabel(1)+"  | "+meta.getColumnLabel(2));
            logger.info("----+----");
            Date end_time = new Date();
            logger.info(end_time);
        }catch (Exception e){
            logger.error(e);
        } finally {
            close(rs,stml,conn);
        }
    }
    private static void close(ResultSet rs,Statement stml,Connection conn){
        try{
            if (rs != null )
                rs.close();
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(stml != null){
                stml.close();
            }
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(conn != null){
                conn.close();
            }
```

```
        }catch (Exception e){
            logger.error(e);
        }
    }
}
```

4.3.5 插入数据

Phoenix 表的插入数据示例如下：

```
package com.bigdata;

import org.apache.log4j.Logger;

import java.sql.*;
import java.util.Date;

/**
 * Created by bigdata on 2019/3/8.
 */
public class PhoenixUpsert {
    public static final Logger logger = Logger.getLogger(PhoenixUpsert.class);
    public static Connection conn = null;
    public static Statement stml = null;
    public static ResultSet  rs = null;
    public static void main(String [] args){
        System.setProperty("HADOOP_USER_NAME", "hbase");
        try {
            String zooks =
"node1.hde.bigdata.com,node2.hde.bigdata.com,node3.hde.bigdata.com:/hbase-unsecure";
            conn = DriverManager.getConnection("jdbc:phoenix:"+zooks);
            if (conn != null){
                logger.info("good ,i got the connection");
            }
            stml = conn.createStatement();
            String sql = "upsert  into test_phoenix_api(\"ID\",\"NAME\") values(2,\'wu\')";
            logger.info(sql);
            Date start_time = new Date();
            logger.info(start_time);
            stml.executeUpdate(sql);
```

```
        conn.commit();
        String query =  "select * from \"test_phoenix_api\""
        rs = stml.executeQuery(query);

        ResultSetMetaData meta = rs.getMetaData();
        logger.info("----+----");
        logger.info(meta.getColumnLabel(1)+"  | "+meta.getColumnLabel(2));
        logger.info("----+----");

        while(rs.next()){
            String str = rs.getString(1)+"  | "+rs.getString(2);
            logger.info(str);
        }
        Date end_time = new Date();
        logger.info(end_time);
    }catch (Exception e){
        logger.error(e);
    } finally {
        close(rs,stml,conn);
    }
}
private static void close(ResultSet rs,Statement stml,Connection conn){
    try{
        if (rs != null )
            rs.close();
    }catch (Exception e){
        logger.error(e);
    }
    try{
        if(stml != null){
            stml.close();
        }
    }catch (Exception e){
        logger.error(e);
    }
    try{
        if(conn != null){
            conn.close();
```

```
        }
      }catch (Exception e){
        logger.error(e);
      }
    }
}
```

4.3.6 查找数据

Phoenix 表的查找数据示例如下：

```
package com.bigdata;

import org.apache.log4j.Logger;
import java.sql.*;
import java.util.Date;

/**
 * Created by bigdata on 2019/3/8.
 */
public class PhoenixSelect {
  public static final Logger logger = Logger.getLogger(PhoenixSelect.class);

  public static Connection conn = null;
  public static Statement stml = null;
  public static ResultSet  rs = null;
  public static void main(String [] args) {
    System.setProperty("HADOOP_USER_NAME", "hbase");
    try {
      String zooks =
"node1.hde.bigdata.com,node2.hde.bigdata.com,node3.hde.bigdata.com:/hbase-unsecure";
      Class.forName("org.apache.phoenix.jdbc.PhoenixDriver");
      conn = DriverManager.getConnection("jdbc:phoenix:" + zooks);
      if (conn != null) {
        logger.info("good ,i got the connection");
      }
      stml = conn.createStatement();
      Date start_time = new Date();
      logger.info(start_time.toString());
      String query = "select * from \"test_phoenix_api\"";
```

```
            rs = stml.executeQuery(query);

            ResultSetMetaData meta = rs.getMetaData();
            logger.info("----+----");
            logger.info(meta.getColumnLabel(1) + "  | " + meta.getColumnLabel(2));
            logger.info("----+----");
            while (rs.next()) {
                String str = rs.getString(1) + "  | " + rs.getString(2);
                logger.info(str);
            }

            Date end_time = new Date();
            logger.info(end_time.toString());
        } catch (Exception e) {
            logger.error(e);
        } finally {
            close(rs,stml,conn);
        }
    }
    private static void close(ResultSet rs,Statement stml,Connection conn){
        try{
            if (rs != null )
                rs.close();
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(stml != null){
                stml.close();
            }
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(conn != null){
                conn.close();
            }
        }catch (Exception e){
```

```
            logger.error(e);
        }
    }
    }
```

4.3.7 删除数据

Phoenix 表的删除数据示例如下：

```
package com.bigdata;

import org.apache.log4j.Logger;

import java.sql.*;
import java.util.Date;

/**
 * Created by bigdata on 2019/3/8.
 */
public class PhoenixDelete {
    public static final Logger logger = Logger.getLogger(PhoenixDelete.class);
    public static Connection conn = null;
    public static Statement stml = null;
    public static ResultSet  rs = null;
    public static void main(String [] args){
        System.setProperty("HADOOP_USER_NAME", "hbase");
        try {
            String zooks = "node1.hde. bigdata.com,node2.hde. bigdata.com,node3.hde.
bigdata.com:/hbase-unsecure";
            conn = DriverManager.getConnection("jdbc:phoenix:"+zooks);
            if (conn != null){
                logger.info("good ,i got the connection");
            }
            stml = conn.createStatement();
            String sql = "delete from test_phoenix_api where ID = 2 ";
            logger.info(sql);
            Date start_time = new Date();
            logger.info(start_time);
            stml.executeUpdate(sql);
            conn.commit();
```

```
            String query = "select * from \"test_phoenix_api\"";
            rs = stml.executeQuery(query);

            ResultSetMetaData meta = rs.getMetaData();
            logger.info("----+----");
            logger.info(meta.getColumnLabel(1)+"  | "+meta.getColumnLabel(2));
            logger.info("----+----");

            while(rs.next()){
                String str = rs.getString(1)+"  | "+rs.getString(2);
                logger.info(str);
            }

            Date end_time = new Date();
            logger.info(end_time);
        }catch (Exception e){
            logger.error(e);
        } finally {
            close(rs,stml,conn);
        }
    }
    private static void close(ResultSet rs,Statement stml,Connection conn){
        try{
            if (rs != null )
                rs.close();
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(stml != null){
                stml.close();
            }
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(conn != null){
```

```
            conn.close();
        }
    }catch (Exception e){
        logger.error(e);
    }
  }
}
```

4.3.8 二级索引

Phoenix 的索引创建后，会有一个索引名命名的表，可以通过 Phoenix-sqlline 输入！table 命令查看。

1. 配置 hbase-site

在创建索引前，需要在 HBase 组件的配置页面中，配置高级 hbase-site。

表4-1 参数配置说明

参数名称	参数说明	参数值
hbase.regionserver.wal.codec	该参数指明HBase的wal编码方式	org.apache.hadoop.hbase.regionserver.wal.IndexedWALEditCodec

2. 创建多列索引

示例代码如下：

```
package com.bigdata;

import org.apache.log4j.Logger;

import java.sql.*;
import java.util.Date;

/**
 * Created by bigdata on 2019/3/19.
 */
public class PhoenixIndex {
  public static final Logger logger = Logger.getLogger(PhoenixIndex.class);
  public static Connection conn = null;
  public static Statement stml = null;
  public static ResultSet  rs = null;
  public static void main(String [] args){
    System.setProperty("HADOOP_USER_NAME", "hbase");
```

```
        try {
            String zooks = "node1.hde. bigdata.com,node2.hde. bigdata.com,node3.hde.
bigdata.com:/hbase-unsecure";
            conn = DriverManager.getConnection("jdbc:phoenix:"+zooks);
            if (conn != null){
                logger.info("good ,i got the connection");
            }
            stml = conn.createStatement();
            String sql = "create index col_index1 on test_phoenix_api( id ,name)";
            logger.info(sql);
            Date start_time = new Date();
            logger.info(start_time);
            stml.executeUpdate(sql);
            conn.commit();

            String query ="select * from \"col_index1\"";
            rs = stml.executeQuery(query);

            ResultSetMetaData meta = rs.getMetaData();
            logger.info("----+----");
            logger.info(meta.getColumnLabel(1)+"  | "+meta.getColumnLabel(2));
            logger.info("----+----");
            while(rs.next()){
                String str = rs.getString(1)+"  | "+rs.getString(2);
                logger.info(str);
            }
            Date end_time = new Date();
            logger.info(end_time);
        }catch (Exception e){
            logger.error(e);
        } finally {
            close(rs,stml,conn);
        }
    }
    private static void close(ResultSet rs,Statement stml,Connection conn){
        try{
            if (rs != null )
```

```
            rs.close();
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(stml != null){
                stml.close();
            }
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(conn != null){
                conn.close();
            }
        }catch (Exception e){
            logger.error(e);
        }
    }
}
```

3. 删除索引

删除索引时，只需要将上述代码中对应的 SQL 语句更换即可。如删除上述代码的索引，其 SQL 语句为：drop index COL_INDEX1 on TEST_PHOENIX_API。

4. 其它类型的索引创建

其它类型的索引和多列索引的方式一致，只需要更改对应的 SQL 语句即可。具体 SQL 语句可参考 3.2.1 Secondary Indexing 二级索引。

4.4 Kerberos环境下的开发

4.4.1 通过 zookeeper 连接 phoenix

注意开启 Kerberos 后，需要进行如下配置：

(1) 需要确认访问集群的用户（比如 HBase 用户）。

(2) 创建 maven 项目和未开 Kerberos 一样。

(3) 从 DE 集群拷贝相关配置文件到本地，开发时需使用这些配置文件：

- 直接拷贝集群中 kdc 服务安装节点(默认为 node1 节点)的/etc/krb5.conf 到 maven 项目的 resources。
- 从集群 node1 节点/etc/security/keytabs/目录下拷贝 hbase.service.keytab 文件到 maven 项目的 resources。

- 另外 core-site.xml、hdfs-site.xml 从/usr/hdp/3.0.1.0-187/hadoop/conf/目录下拷贝到 maven 项目的 resources。
- hbase-site.xml 从/usr/hdp/3.0.1.0-187/hbase/conf/目录下拷贝到 maven 项目的 resources。

Phoenix 的开发示例代码如下：

```
package com.bigdata;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.security.UserGroupInformation;
import org.apache.log4j.Logger;

import java.sql.*;
import java.sql.Date;
import java.util.*;

/**
 * Created by bigdata on 2019/3/8.
 */
public class PhoenixSelectKerberos {
   public static final Logger logger = Logger.getLogger(PhoenixSelectKerberos.class);
   public static Connection conn = null;
   public static Statement stml = null;
   public static ResultSet  rs = null;
   public static void main(String [] args){

      try {
         String krbconfpath =
"C:\\Users\\bigdata\\Desktop\\service_daemons\\src\\main\\resources\\krb5.conf";
         String keytab =
"C:\\Users\\bigdata\\Desktop\\service_daemons\\src\\main\\resources\\hbase.service.keytab";

         String principal = "hbase/node1.hde. bigdata.com@HDE. BIGDATA.COM";
         System.setProperty("java.security.krb5.conf", krbconfpath);
          UserGroupInformation.loginUserFromKeytab(principal, keytab);

         String zooks = "node1.hde. bigdata.com,node2.hde. bigdata.com,node3.hde.
bigdata.com:/hbase-secure";
```

```
        conn = DriverManager.getConnection("jdbc:phoenix:"+zooks);
        if (conn != null){
            logger.info("good ,i got the connection");
        }
        stml = conn.createStatement();

        String query = "select * from \”test_phoenix_api\”";
        rs = stml.executeQuery(query);

        ResultSetMetaData meta = rs.getMetaData();
        logger.info("----+----");
        logger.info(meta.getColumnLabel(1)+"  | "+meta.getColumnLabel(2));
        logger.info("----+----");

        while(rs.next()){
            String str = rs.getString(1)+"  | "+rs.getString(2);
            logger.info(str);
        }

        java.util.Date end_time = new java.util.Date();
        logger.info(end_time);
    }catch (Exception e){
        logger.error(e);
    } finally {
        close(rs,stml,conn);
    }
}
private static void close(ResultSet rs,Statement stml,Connection conn){
    try{
        if (rs != null )
            rs.close();
    }catch (Exception e){
        logger.error(e);
    }
    try{
        if(stml != null){
            stml.close();
        }
```

```
        }catch (Exception e){
            logger.error(e);
        }
        try{
            if(conn != null){
                conn.close();
            }
        }catch (Exception e){
            logger.error(e);
        }
    }
}
```

上面代码中，开头部分表示 Kerberos 信息，下面的流程和未开启 Kerberos 的流程类似。

4.4.2 通过 queryserver 连接

在 HBase 集群的 Web 配置界面中，添加如下参数到自定义 hbase-site.xml 中：

phoenix.queryserver.http.port =8765（该值为默认值，如果不添加该值，默认将使用 8765 端口进行连接）。

phoenix.queryserver.serialization=PROTOBUF。

phoenix.queryserver.kerberos.principal=HTTP/_HOST@HDE. BIGDATA.COM

phoenix.queryserver.kerberos.file=/etc/security/keytab/spnego.service.service.keytab

代码实例：

```
package kerberos;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import java.sql.*;
import java.util.Properties;

/**
* Created by bigdata
 */
public class HbaseDemo {

public static Connection conn = null;
    public static void main(String [] args) {
```

```
    String resource = "D:\\IdeaProject\\services\\src\\main\\resources\\";
    String keytab = resource+"hbase.headless.keytab";
    String principal = "hbase-WH@HDE. BIGDATA.COM";
    String krb5path = resource+"krb5.conf";

    System.setProperty("java.security.krb5.conf", krb5path);
    Configuration hbaseconf = HBaseConfiguration.create();
    hbaseconf.addResource(resource+"hbase-site.xml");
    hbaseconf.set("hbase.client.retries.number","5");

    try {
      Properties pro = new Properties();
      pro.setProperty("keytab",keytab);
      pro.setProperty("principal",principal);
      String connUrl =
"http://node1.hde.bigdata.com:8765;serialization=PROTOBUF;authentication=SPNEGO";
      Class.forName("org.apache.phoenix.queryserver.client.Driver");
      conn = DriverManager.getConnection("jdbc:phoenix:thin:url=" + connUrl,pro);

      Statement st = conn.createStatement();
      st.execute("upsert into test_ker values(111,'bigdata')");
      st.execute("upsert into test_ker values(112,'com')");
      conn.commit();

      ResultSet rs = st.executeQuery("select * from  test_ker");
      while(rs.next()) {
        System.out.println(rs.getInt(1) +"......"+ rs.getString(2));
      }
      rs.close();
      st.close();
    } catch (Exception e) {
      System.out.println("list table failed " +e);
    }finally{
      try{
        if(conn != null ){
          System.out.println("0k, close connection");
          conn.close();
        }
      }catch (Exception e){
        System.out.println(e.getStackTrace());
     }
    }
```

```
  }
}
```

注意：需要根据自己的环境更改相关配置，比如 queryserver IP。

5 版本新增特性

目前使用的 Phoenix 版本是 5.0.0 的版本，新版本的亮点特性：

(1) 清理已弃用的 API 并利用新的 Performant API。

(2) 重构协处理器实现用于在 HBase 2.0 中使用新的 Coprocessor 或 Observer API。

6 常见问题解答

1. Phoenix 查询问题

- 找不到表异常

  Phoenix 对表大小写较敏感，如果小写命名的表，可以通过大写访问，或者小写表名带上双引号。另外对于 SYSTEM 开头的表，需要对其后的表名加上双引号，如 select * from SYSTEM."CATALOG"。

- phoenix-hive 创建 Hive 内表后执行 select 异常

  该问题是给 select 字段重复添加双引号导致的。

  首先检查输入格式、表字段、表名等是否正确，如果格式正确，然后排查 Phoenix 是否为 4.11 以下的版本。如果版本小于 4.11，更改方法如下：从/usr/hdp/3.0.1.0-187/phoenix 路径下找到对应的 phoenix-4.10.0-HBase-1.1-hive.jar 包，将其 copy 到本地，使用 idea 编译器打开，并找到 org.apache.phoenix.hive.query.PhoenixQueryBuilder.java 文件，然后删除 101 和 106 行的代码 readColumnList = coulmnMappingUtils.quoteColumns(readColumnList);，然后重新打成 jar 包并上传。

2. Phoenix 表的管理

Phoenix 创建的表如果要删除，不要通过 hbase shell 或者 hbase api 等方式删除，必须在 Phoenix 里面删除，否则会出现表已经删除，但是在 Phoenix 仍可以看见，而且操作该表会有异常。

3. Phoenix 表查询超时

默认情况下 Phoenix 查询超过 1 分钟，就可能抛出 TimeoutException，可以在自定义 hbase-site 添加 phoenix.query.timeoutMs=180000 和 hbase.rpc.timeout=180000 来更改超时时间。

4. Phoenix 二级索引表与源数据表保持同步问题

源数据表的增、删、改等操作必须通过 Phoenix 的客户端执行，才能保证二级索引与源表同步。

5. 大批量 upsert 插入 Phoenix 报错

异常：unable to find cached index metadata。

问题原因：在 cache 里的 index 失效了，需要增大缓存 index 的有效时间。

解决方法：在 HBase 组件配置页面中，向自定义 hbase-site 中添加如下配置：

```
<property>
<name>phoenix.coprocessor.maxServerCacheTimeToLiveMs</name>
<value><3000000</value>
</property>
```

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.2 特点及优势 1-1
1.3 组件架构 1-2
1.3.1 Solr/Lucene 体系架构 1-2
1.3.2 SolrCloud 架构 1-3
1.4 应用场景 1-4
1.5 增强特性 1-4
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 数据目录检查 2-1
2.1.2 查看组件的日志信息 2-2
2.2 运行状态监控 2-2
2.2.1 查看组件详情 2-2
2.2.2 组件检查 2-3
2.3 快速使用指导 2-4
2.3.1 非 Kerberos 环境 2-4
2.3.2 Kerberos 环境 2-6
2.3.3 Kerberos 环境下用户身份认证 2-8
2.4 快速链接 2-10
2.4.1 配置组件快速链接 2-10
2.4.2 访问 Solr 快速链接 2-10
3 使用指南 3-1
3.1 可视化界面 3-1
3.1.1 仪表盘 Dashboard 3-1
3.1.2 日志设置与查看 3-1
3.1.3 Cloud 集群展示 3-2
3.1.4 Collections 界面展示 3-2
3.1.5 索引工具 3-3
3.1.6 查询界面 3-3
3.2 Solr 集群扩容 3-4
3.2.1 使用场景 3-4

3.2.2 扩容前准备 3-4
3.2.3 扩容约束 3-4
3.2.4 扩容时禁止的操作 3-5
3.2.5 扩容操作指导 3-5
3.2.6 Solr 扩容后操作 3-6
3.2.7 扩容验证 3-6
3.3 Solr 集群缩容 3-6
3.3.1 使用场景 3-6
3.3.2 缩容前准备 3-6
3.3.3 缩容约束 3-7
3.3.4 缩容影响 3-7
3.3.5 缩容时禁止的操作 3-7
3.3.6 缩容操作指导 3-7
3.3.7 缩容验证 3-8
3.4 分片扩容与缩容 3-8
3.5 权限访问控制 3-10
3.5.1 权限说明 3-10
3.5.2 权限使用操作示例 3-10
3.6 数据备份 3-14
3.6.1 创建备份 3-14
3.6.2 恢复备份 3-15
4 开发指南 4-1
4.1 API 使用 4-1
4.1.1 增删改操作 4-1
4.1.2 管理常用操作 4-3
4.1.3 查询 Search 4-5
4.1.4 客户端 API SolrJ 使用 4-9
5 最佳实践 5-1
5.1 MySQL 数据同步 5-1
5.1.1 创建数据库 5-1
5.1.2 配置文件修改 5-1
5.1.3 创建 collection 5-2
5.1.4 导入数据 5-3
6 常见问题解答 6-1
6.1 调优 6-1
6.1.1 Schema 优化 6-1

6.1.2 索引更新与提交调优 ······ 6-1
6.1.3 索引合并性能调优 ······ 6-2
6.1.4 查询性能优化 ······ 6-2
6.2 通用类 ······ 6-2
6.2.1 创建 core 失败的情况下，UI 界面会出现错误提示，这个信息怎么删除？ ······ 6-2
6.2.2 Solr 使用 SQL 语句精确查询中文字段，出现中文转换编码问题，怎么解决? ······ 6-3
6.3 运维类问题 ······ 6-3
6.3.1 日志查看方法 ······ 6-3
6.3.2 组件停止 ······ 6-4
6.3.3 修改系统组件 Infra-solr 的参数 infra_solr_znode，出现日志异常，怎么解决？ ······ 6-5

1 组件简介

1.1 组件概述

Solr 是一个基于高性能全文搜索引擎库 Lucene 的企业级搜索平台。其对 Lucene 进行了扩展，提供了比 Lucene 更为丰富的查询语言，同时实现了可配置、可扩展并对查询性能进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的全文搜索平台。

1.2 特点及优势

1. Solr 特性

- 高级的全文搜索功能
- 针对大流量进行了优化
- 基于标准的开放接口（XML、JSON、CSV 等）
- 综合的管理界面
- 高度的可扩展性以及容错性
- 灵活，适应性强，配置简单
- 可扩展的插件体系

2. SolrCloud 特性

SolrCloud 是 Solr4.0 版本之后开发出的具有开创意义的基于 Solr 和 Zookeeper 的分布式搜索方案。或者可以说，SolrCloud 是 Solr 的一种部署方式。SolrCloud 特性包括：

- 集中式的配置信息

  使用 Zookeeper 进行集中配置。启动时可以把指定的 Solr 相关配置文件上传 Zookeeper，多机器共用。这些 Zookeeper 中的配置不会再拿到本地缓存，Solr 直接读取 Zookeeper 中的配置信息。配置文件的变动，所有机器都可以感知到。

  另外，Solr 的一些任务也是通过 Zookeeper 作为媒介发布的。目的是为了容错。任务执行过程中服务器宕机，服务器重启或者集群选出新的 leader 时，任务可以继续执行。
- 自动容错

  SolrCloud 中索引的每个分片可以创建多个 Replication。每个 Replication 都可以对外提供服务。一个 Replication 挂掉不会影响索引服务。更强大的是，它还能自动的在其它机器上帮用户把失败机器上的索引 Replication 重建并投入使用。
- 近实时搜索

  立即推送式的 Replication（也支持慢推送）。可以在秒级内检索到新加入索引数据。
- 查询时自动负载均衡

  SolrCloud 索引的多个 Replication 可以分布在多台机器上，均衡查询压力。如果查询压力大，可以通过扩展机器，增加 Replication 来减缓。
- 自动分发的索引和索引分片

  发送文档到任何节点，它都会转发到正确节点。

- 事务日志

  事务日志确保更新无丢失，即使文档没有索引到磁盘。
- 可直接将索引数据存储在 HDFS 上

  当有上亿数据来建索引，可以和“通过 MR 批量创建索引”联合使用。
- 通过 MapReduce 等批量创建索引

  与分布式计算框架结合，大大提高对海量数据离线建索引的效率。
- 强大的 RESTFUL API

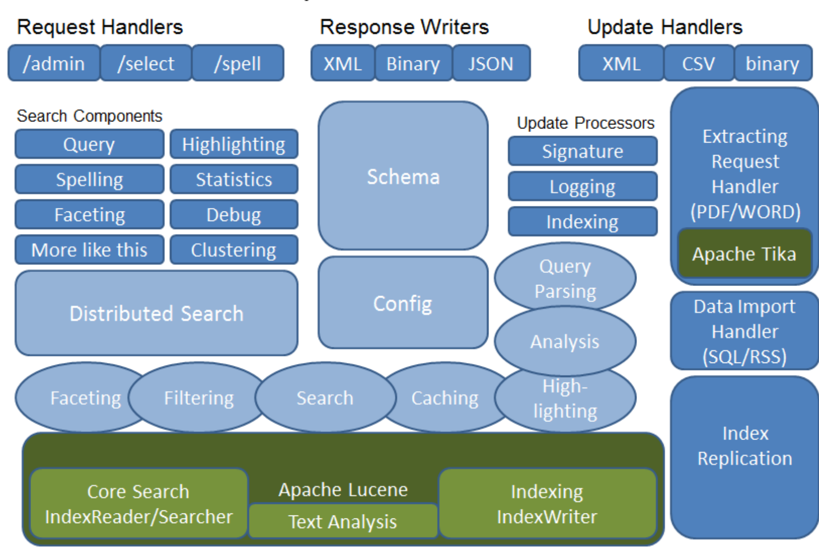
  通用的管理功能，都可以通过此 API 方式调用。
- 优秀的管理界面

  主要信息一目了然，可以清晰的以图形化方式看到 SolrCloud 的部署分布，当然还有不可或缺的 Debug 功能。

1.3 组件架构

1.3.1 Solr/Lucene 体系架构

图1-1 Solr 组件架构



Solr 的主要构建块：

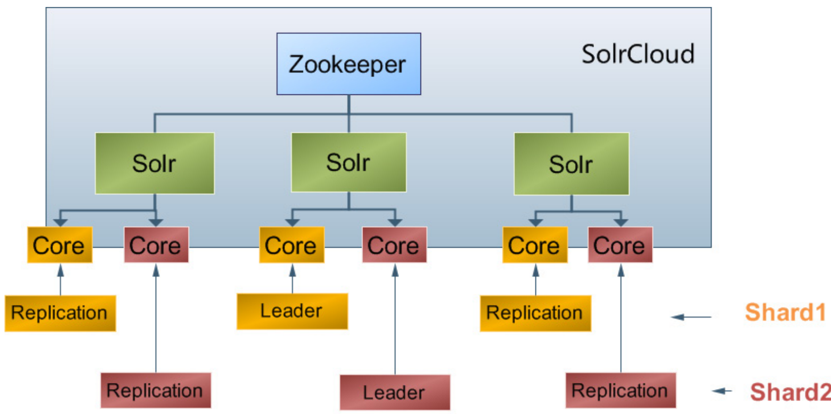
- 请求处理程序 Request Handlers：发送到 Solr 的请求由这些请求处理程序处理。请求可以是查询请求或索引更新请求。根据这些请求类型来选择对应请求处理程序。为了将请求传递给 Solr，通常将处理器映射到某个 URI 端点，并且它将为指定的请求提供服务。
- 搜索组件 Search Components：搜索组件是 Solr 中提供的搜索类型（功能）。它可能是拼写检查、查询、切面、高亮显示等，这些搜索组件被注册为搜索处理程序。
- 查询解析器 Query Parsing：Solr 查询解析器解析传递给 Solr 的查询，并验证查询的语法是否有错误。解析查询后，将它们转换为 Lucene 理解的格式。

- 响应写入器 Response Writers：Solr 中的响应写入器是为用户查询生成格式化输出的组件。Solr 支持 XML，JSON，CSV 等响应格式。对每种类型的响应都有不同的响应写入。
- 分析器/分词器 Analysis：Lucene 以 token 的形式识别数据。Solr 分析内容并将其拆分成 tokens，并将这些 tokens 传递给 Lucene。Solr 中的分析器检查字段的文本并生成 tokens 流。分词器将分析器准备的 tokens 流分解成 token。
- 更新请求处理器 Update Handlers：每当向 Solr 发送更新请求时，请求都通过一组称为更新请求处理器的插件（包括签名、日志记录、索引）运行。这个处理器负责修改，例如删除字段、添加字段等。

1.3.2 SolrCloud 架构

SolrCloud 是基于 Solr 和 Zookeeper 的分布式搜索方案，它的主要思想是使用 Zookeeper 作为集群的配置信息中心。图 1-2 为 3 节点的 SolrCloud 架构图，有一个 Collection 索引，包含 3 个 Replicas 和 2 个 Shards。

图1-2 SolrCloud 架构



基本概念：

- Collection：在 SolrCloud 集群中逻辑意义上的完整的索引。它可以被划分为一个或者多个 Shard，它们使用相同的 Config Set。
- Config Set：Solr Core 提供服务必须的一组配置文件。包括 solrconfig.xml 和 managed-schema 等。
- Core：即 Solr Core，一个 Solr 实例中包含一个或者多个 Solr Core，每个 Solr Core 可以独立提供索引和查询功能，每个 Solr Core 对应一个索引或者 Collection 的 Shard 的副本(replica)。
- Shard：Collection 的逻辑分片。每个 Shard 都包含一个或者多个 replicas，通过选举确定哪个是 Leader。
- Replica：Shard 的拷贝。一个 Replica 存在于 Solr 的一个 Core 中。
- Leader：赢得选举的 Shard replicas。当索引 documents 时，SolrCloud 会传递它们到此 Shard 对应的 leader，leader 再分发它们到 Shard 的全部 replicas。
- ZooKeeper：它在 SolrCloud 是必须的，提供分布式锁、处理 Leader 选举等功能。

1.4 应用场景

1. 日志分析

- 运维分析：对 IT 设备进行运维分析与故障定位、对业务指标分析运营效果。
- 实时高效：从入库到能够被检索到，时间差在数秒到数分钟之间。

2. 站内搜索

- 实时检索：站内资料或商品信息更新数秒至数分钟即可被检索。
- 分类统计：检索同时可以将符合条件的商品进行分类统计。
- 高亮显示：提供高亮能力，页面可自定义高亮显示方式。

1.5 增强特性

- 增加了 Kerberos 认证，保障了索引数据的安全性。
- 增加 Collection 操作的权限控制。
- 增加了 Kerberos 模式下 Solr Admin 的单点登录功能。

2 快速入门

2.1 组件安装

说明

- 在Hadoop集群或Solr集群中，安装Solr时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- 大数据平台中的Solr默认采用SolrCloud部署模式，该模式依赖Zookeeper，因此安装Solr之前需要安装Zookeeper。

大数据集群中，部署Solr包括以下两种方式：

- 在集群类型为Hadoop的大数据集群中安装Solr，此时在集群中同时还能部署其他大数据组件。
- 在集群类型为Solr的大数据集群中安装Solr，此时在集群中仅能部署Solr及其依赖的组件Zookeeper。

2.1.1 数据目录检查

根据现场磁盘分区方案和挂盘方案的不同，组件安装完成后，必须对各组件的数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表2-1所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
Solr	是（配置项的参数值默认使用某一个挂载路径）	solr.data.home	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录 • 仅支持单路径挂载使用，所以只允许配置一个数据目录
Zookeeper	是（配置项的参数值默认使用某一个挂载路径）	dataDir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录

2.1.2 查看组件的日志信息

表2-2 组件日志路径说明

组件	日志路径
Solr	/var/de_log/solr
ZooKeeper	/var/de_log/zookeeper/user_{user.name}/，其中${user.name}是指执行任务的用户名

2.2 运行状态监控

2.2.1 查看组件详情

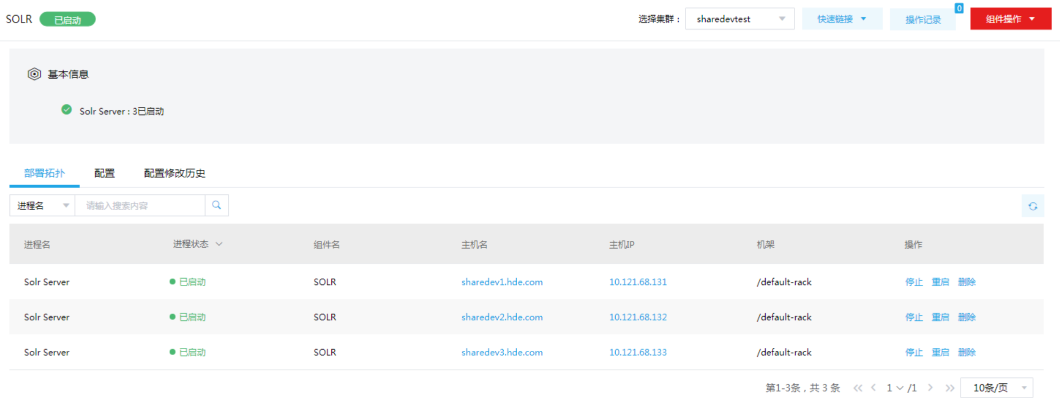
进入 Solr 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、访问快速链接、查看操作历史等。

图2-1 组件详情



2.2.2 组件检查

Solr 组件检查时会执行查看集群中有多少索引的操作，组件检查成功表示 Solr 组件可正常使用。集群在使用过程中，根据实际需要，可对 Solr 执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：
  - 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
    - 在集群详情页面选择[组件]页签，单击业务组件列表中 Solr 组件对应的<组件检查>按钮。
    - 在集群详情页面选择[组件]页签，单击业务组件列表中 Solr 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
  - 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态，如图 2-2 所示，表示该 Solr 组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“Solr Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情

2.3 快速使用指导

说明

- 本章节操作需要切换至具有操作 Solr 权限的用户。
- 根据大数据集群是否开启 Kerberos 认证，用户访问 Solr 时的认证方式不同，详情请参见本章节内容。

Solr 既可以通过集群用户连接，又可以通过组件超级用户连接。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
    - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为"集群超级用户"。
    - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 Solr 组件的 solr 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 Kerberos 环境

说明

非 Kerberos 环境下，不需要用户做身份认证即可直接对 Solr 执行操作。

1. curl 命令访问测试

在非 Kerberos 环境下通过控制台执行 curl 命令进行创建索引、插入数据、查询数据等操作。操作步骤如下：

(1) 上传配置

在安装有 Solr 节点的服务器上执行以下命令，该命令的作用是将 Solr 自带的配置文件上传到 Zookeeper 中，并通过-confname 选项指定配置文件名称。

```
/usr/hdp/3.0.1.0-187/solr/server/scripts/cloud-scripts/zkcli.sh -zkhost <ZKHOST>:2181/solr -cmd upconfig -confdir /usr/hdp/3.0.1.0-187/solr/server/solr/configsets/_default/conf/ -confname <CONFIG_NAME>
```

其中：

- <ZKHOST>：大数据集群中安装有 ZooKeeper Server 节点的服务器 IP 地址
- <CONFIG_NAME>：上传的配置文件名称，该名称由用户指定

(2) 创建索引

curl
'http://<HOST>:8983/solr/admin/collections?action=CREATE&name=<NAME>&numShards=<Shards>&replicationFactor=<Replicas>&collection.configName=<CONFIG_NAME>&wt=json'

其中：

- <HOST>：大数据集群中安装有 Solr 节点的服务器 IP 地址
- <NAME>：需要创建的索引名称
- <Shards>：索引的分片数，注意分片数不能大于大数据集群中 Solrcloud 的节点数
- <Replicas>：索引的副本数
- <CONFIG_NAME>：第一步上传的配置文件的名称

(3) 插入数据

curl http://<HOST>:8983/solr/<NAME>/update?commitWithin=1000 -d '[{"id":"1"}]'

其中：

- <HOST>：大数据集群中安装有 Solr 节点的服务器 IP 地址
- <NAME>：第二步创建的索引名称

(4) 查询数据

curl http://<HOST>:8983/solr/<NAME>/select?q=*

其中：

- <HOST>：大数据集群中安装有 Solr 节点的服务器 IP 地址
- <NAME>：第二步创建的索引名称

2. SolrJ 访问测试

SolrJ 是操作 Solr 的 Java 客户端，它提供了增加、修改、删除、查询 Solr 索引的 Java 接口。通过 SolrJ 提供的 API 接口来操作 Solr 服务，SolrJ 底层是通过使用 httpClient 中的方法来完成 Solr 的操作。

通过 Java Client 访问 Solr 集群的 8983 端口进行测试，在 pom.xml 文件中引入依赖：

```
<dependency>
    <artifactId>solr-solrj</artifactId>
    <groupId>org.apache.solr</groupId>
    <version>7.4.0-cdh6.2.0</version>
</dependency>
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.6</version>
</dependency>
```

下面的代码是介绍如何使用 solrj 在指定索引中添加以及查询文档。

```
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.impl.HttpSolrClient;
import org.apache.solr.client.solrj.response.QueryResponse;
import org.apache.solr.common.SolrDocumentList;
import org.apache.solr.common.SolrInputDocument;
```

```
public class SolrjTest {
    public static void main(String[] args) throws Exception{
        HttpSolrClient solrClient = new
HttpSolrClient.Builder("http://<HOST>:8983/solr/<Collection_NAME>").build();
        //添加文档
        SolrInputDocument document = new SolrInputDocument();
        document.addField("id", "100");
        solrClient.add(document);
        solrClient.commit();
        //查询文档
        SolrQuery query = new SolrQuery();
        query.set("q", "*");
        QueryResponse response = solrClient.query(query);
        SolrDocumentList results = response.getResults();
        System.out.println("索引中文档总数为：" + results.getNumFound());
    }
}
```

2.3.2 Kerberos 环境

说明

- Kerberos 环境下，用户需要做身份认证才可访问 Solr 并对 Solr 执行操作，认证方式请参见 2.3.3 Kerberos 环境下用户身份认证。
- 本章节操作需要切换至具有操作 Solr 权限的用户，例如 solr 用户。
- Kerberos 环境下，curl 命令需增加参数项"--negotiate -u :"，具体参考本章节示例。

1. curl 命令访问测试

在 Kerberos 环境下通过控制台执行 curl 命令进行创建索引、插入数据、查询数据等操作。操作步骤如下：

(1) 创建索引

curl --negotiate -u : 'http://<HOST>:8983/solr/admin/collections?action=CREATE&name=<NAME>&numShards=<Shards>&replicationFactor=<Replicas>&collection.configName=<CONFIG_NAME>&wt=json'

其中：

- <HOST>：大数据集群中安装有 Solr 节点的服务器 IP 地址
- <NAME>：需要创建的索引名称
- <Shards>：索引的分片数，注意分片数不能大于大数据集群中 Solrcloud 的节点数
- <Replicas>：索引的副本数
- <CONFIG_NAME>：第一步上传的配置文件的名称

(2) 插入数据

curl --negotiate -u : http://<HOST>:8983/solr/<NAME>/update?commitWithin=1000 -d '[{"id":"1"}]'

其中：

- <HOST>：大数据集群中安装有 Solr 节点的服务器 IP 地址
- <NAME>：第二步创建的索引名称

(3) 查询数据

curl --negotiate -u : http://<HOST>:8983/solr/<NAME>/select?q=*

其中：

- <HOST>：大数据集群中安装有 Solr 节点的服务器 IP 地址
- <NAME>：第二步创建的索引名称

2. SolrJ 访问测试

通过 Java Client 访问 Solr 集群的 8983 端口进行测试，在 pom.xml 文件中引入依赖：

```
<dependency>
    <artifactId>solr-solrj</artifactId>
    <groupId>org.apache.solr</groupId>
    <version>7.4.0-cdh6.2.0</version>
</dependency>
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.6</version>
</dependency>
```

另外，Kerberos 环境下 Java Client 需要指定 krb5.conf 和 jaas.conf 配置文件，其中 krb5.conf 是 solr 集群的 Kerberos 客户端配置文件，jaas.conf 文件中可指定操作 solr 的用户以及使用的 keytab 文件，keytab 文件获取方式可参考 2.3.3 Kerberos 环境下用户身份认证。代码中操作用户以 user1 用户为例，用户可根据实际情况指定用户及对应的 keytab 文件。

jaas.conf 文件内容如下（首行内容为空）：

```

Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="conf/user1.keytab"
  storeKey=true
  useTicketCache=false
  debug=false
  principal="user1@SOLRSSO.COM";
};
```

下面的代码是介绍如何在 Kerberos 环境下使用 solrj 在指定索引中添加以及查询文档。

```
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.impl.*;
import org.apache.solr.client.solrj.response.QueryResponse;
```

```
import org.apache.solr.common.SolrDocument;
import org.apache.solr.common.SolrDocumentList;
import org.apache.solr.common.SolrInputDocument;

public class SolrjTest {
    public static void main(String[] args) throws Exception{
        System.setProperty("java.security.auth.login.config", "conf/jaas.conf");
        System.setProperty("java.security.krb5.conf", "conf/krb5.conf");
        System.setProperty("javax.security.auth.useSubjectCredsOnly", "false");

        Krb5HttpClientBuilder krbBuild = new Krb5HttpClientBuilder();
        SolrHttpClientBuilder kb = krbBuild.getBuilder();
        HttpClientUtil.setHttpClientBuilder(kb);
HttpSolrClient solrClient = new
HttpSolrClient.Builder("http://<HOST>:8983/solr/<Collection_NAME>").build();
        //添加文档
        SolrInputDocument document = new SolrInputDocument();
        document.addField("id", "100");
        solrClient.add(document);
        solrClient.commit();
        //查询文档
        SolrQuery query = new SolrQuery();
        query.set("q", "*");
        QueryResponse response = solrClient.query(query);
        SolrDocumentList results = response.getResults();
        System.out.println("索引中文档总数为：”+ results.getNumFound());
    }
}
```

2.3.3 Kerberos 环境下用户身份认证

如果大数据集群开启 Kerberos，若想操作 Solr，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

1. 集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

Solr 组件还可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**
  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的 /etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：

  chown user1 /etc/security/keytabs/user1.keytab

  b. 使用 klist 命令查看 user1.keytab 的 principal 名称，命令如下：

  klist -k user1.keytab

  【说明】如图 2-4 所示，红框内容即为 user1.keytab 的 principal 名称。

  图2-4 认证文件的 principal 名称

```
[root@sharedev1 keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@SHAREDEVTEST.COM
   1 user1@SHAREDEVTEST.COM
   1 user1@SHAREDEVTEST.COM
   1 user1@SHAREDEVTEST.COM
   1 user1@SHAREDEVTEST.COM
   1 user1@SHAREDEVTEST.COM
   1 user1@SHAREDEVTEST.COM
   1 user1@SHAREDEVTEST.COM
```

  c. 切换至用户 user1，并执行身份验证的命令如下：

  su user1

  kinit -kt user1.keytab user1@SHAREDEVTEST.COM

  【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@SHAREDEVTEST.COM 为 user1.keytab 的 principal 名称。

  d. 输入 ***klist*** 命令可查看认证结果。

  图2-5 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10002
Default principal: user1@SHAREDEVTEST.COM

Valid starting       Expires              Service principal
10/15/2021 11:17:57  09/19/2026 11:17:57  krbtgt/SHAREDEVTEST.COM@SHAREDEVTEST.COM
sh-4.2$
```

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**
  a. 输入以下命令：kinit user1

  b. 根据提示输入密码 Password for user1@SHAREDEVTEST.COM：<密码>

  c. 输入 ***klist*** 命令可查看认证结果。

  图2-6 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10002
Default principal: user1@SHAREDEVTEST.COM

Valid starting       Expires              Service principal
10/15/2021 11:17:57  09/19/2026 11:17:57  krbtgt/SHAREDEVTEST.COM@SHAREDEVTEST.COM
sh-4.2$
```

2. 组件超级用户身份认证

可以通过组件超级用户访问 Solr 组件，比如 solr 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 solr 用户示例）的认证，操作如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 solr 的认证文件“solr.service.keytab”。

(2) 使用 klist 命令查看 solr.service.keytab 的 principal 名称，命令如下：

klist -k solr.service.keytab

如图 2-7 所示，红框内容即为 solr.service.keytab 文件的 principal 名称。

图2-7 认证文件的 principal 名称

```
[root@sharedev1 keytabs]# klist -k solr.service.keytab
Keytab name: FILE:solr.service.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 solr/sharedev1.hde.com@SHAREDEVTEST.COM
   2 solr/sharedev1.hde.com@SHAREDEVTEST.COM
   2 solr/sharedev1.hde.com@SHAREDEVTEST.COM
   2 solr/sharedev1.hde.com@SHAREDEVTEST.COM
   2 solr/sharedev1.hde.com@SHAREDEVTEST.COM
```

(3) 切换至用户 solr，并执行身份验证的命令如下：

su solr

kinit -kt solr.service.keytab solr/sharedev1.hde.com@SHAREDEVTEST.COM

【说明】其中：solr.service.keytab 为 solr 的认证文件，solr/sharedev1.hde.com@SHAREDEVTEST.COM 为 solr.service.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

图2-8 查看认证结果

```
[solr@sharedev1 ~]$ klist
Ticket cache: FILE:/tmp/krb5cc_1008
Default principal: solr/sharedev1.hde.com@SHAREDEVTEST.COM

Valid starting       Expires              Service principal
10/15/2021 11:20:06  09/19/2026 11:20:06  krbtgt/SHAREDEVTEST.COM@SHAREDEVTEST.COM
[solr@sharedev1 ~]$
```

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

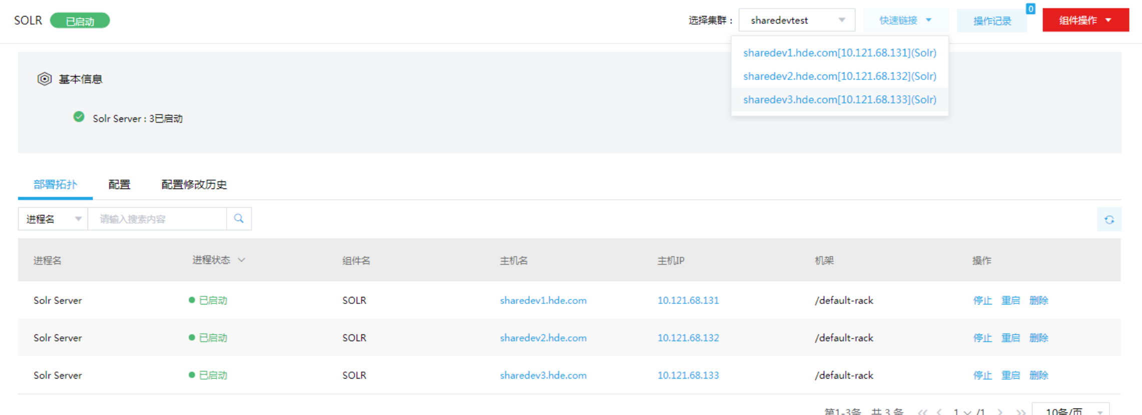
(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

2.4.2 访问 Solr 快速链接

Solr 提供了一个综合的 HTML 管理页面，支持查看 Solr 实例的运行情况以及集群的基本信息等。

(1) 如图 2-9 所示，在 Solr 组件详情页面的右上角[快速链接]的下拉框中，可以获取 Solr 管理页面的访问入口信息。

图2-9 Solr 快速链接



(2) 根据集群是否开启 Kerberos，访问 Solr 快速链接分为两种情况：

- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，可直接跳转访问对应的 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。

(3) 在 Solr 管理页面，可查看的信息请参见 3.1 可视化界面章节。

3 使用指南

3.1 可视化界面

说明

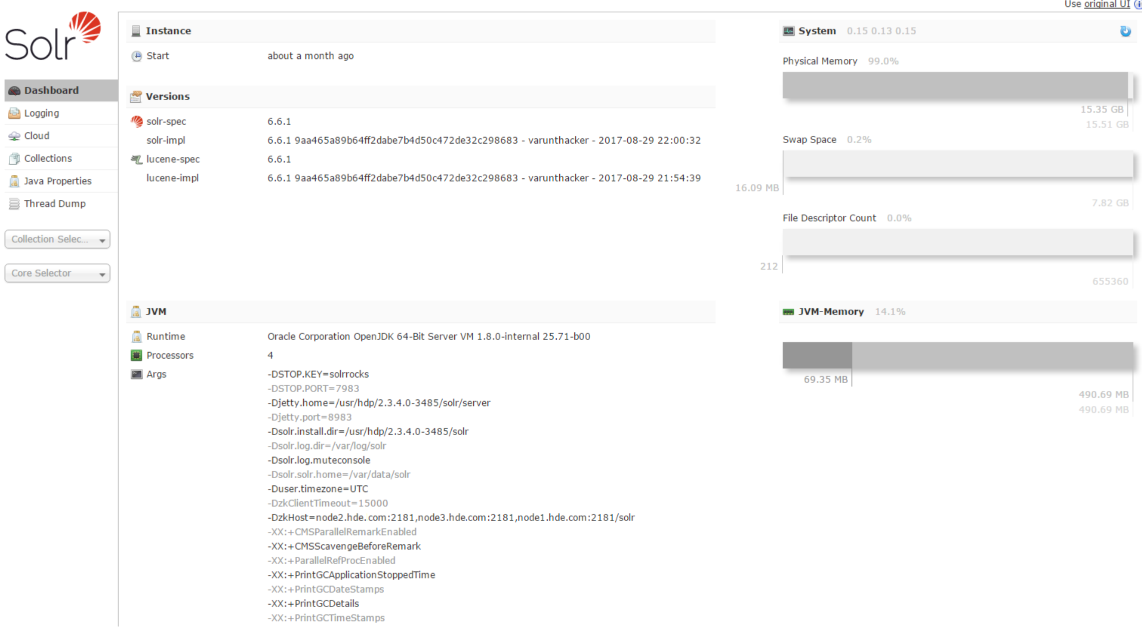
大数据集群部署完成后，用户需要配置组件快速链接，才能顺利跳转到可视化界面。配置组件快速访问链接的方法，详情请参考 2.4.1 配置组件快速链接。

可以通过每个 Solr 节点 hostname 访问 Solr 管理界面：http://<hostname>:8983/。登录用户必须是集群超级用户，只有集群超级用户才有 Solr 单点登录的权限。通过管理界面可以查看 Solr 实例的运行情况以及集群的基本信息等。

3.1.1 仪表盘 Dashboard

通过 Dashboard 可以查看基本信息，包括节点所在的机器、端口号、节点的状态、容量以及被使用情况、版本号等，如图 3-1 所示。

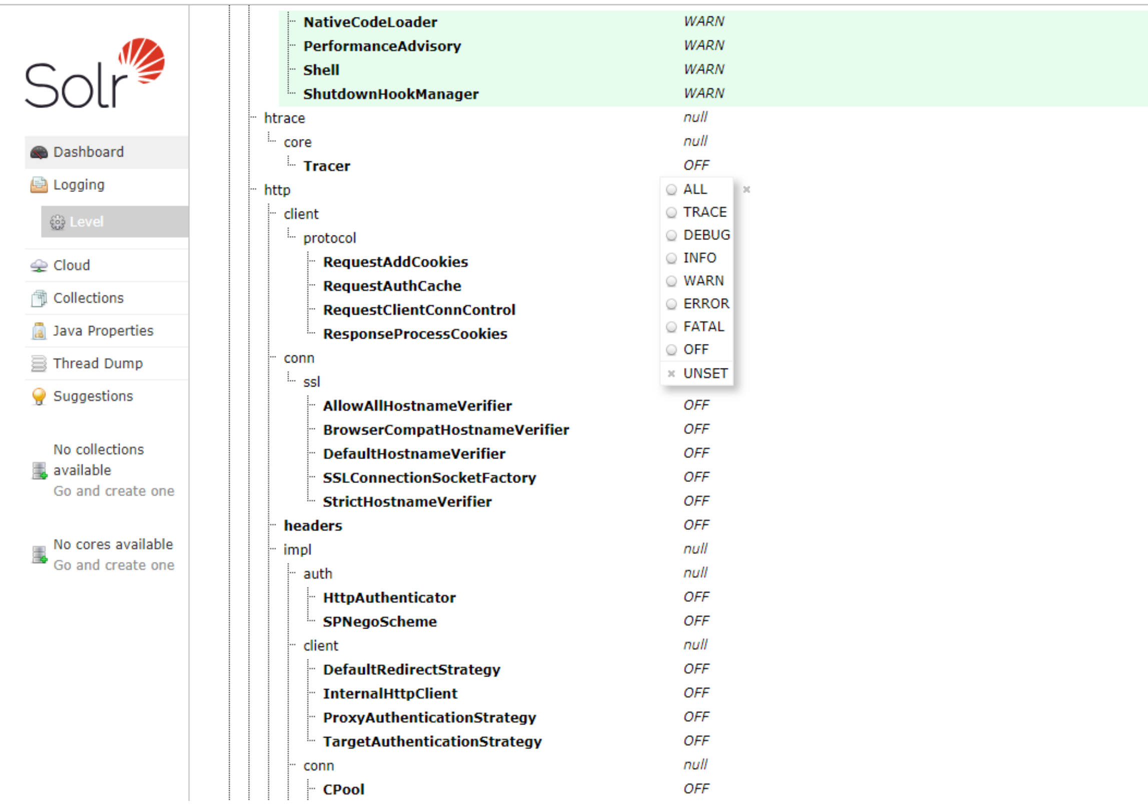
图3-1 仪表盘 Dashboard 展示



3.1.2 日志设置与查看

日志界面可以设置日志级别，包括 ALL、TRACE、DEBUG、INFO、WARN、ERROR、FATAL、OFF、UNSET 等，以及查看相关的日志信息，如图 3-2 所示。

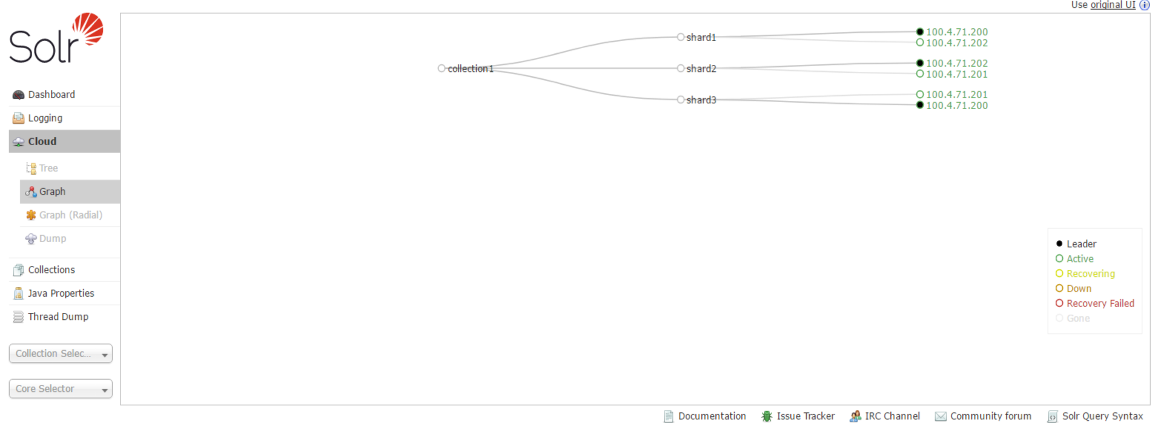
图3-2 日志查看界面



3.1.3 Cloud 集群展示

Cloud 展示界面可以对 SolrCloud 集群进行宏观监控，如一个 Collection 由哪些 shard 组成，每个 shard 所处的节点信息等，如图 3-3 所示。

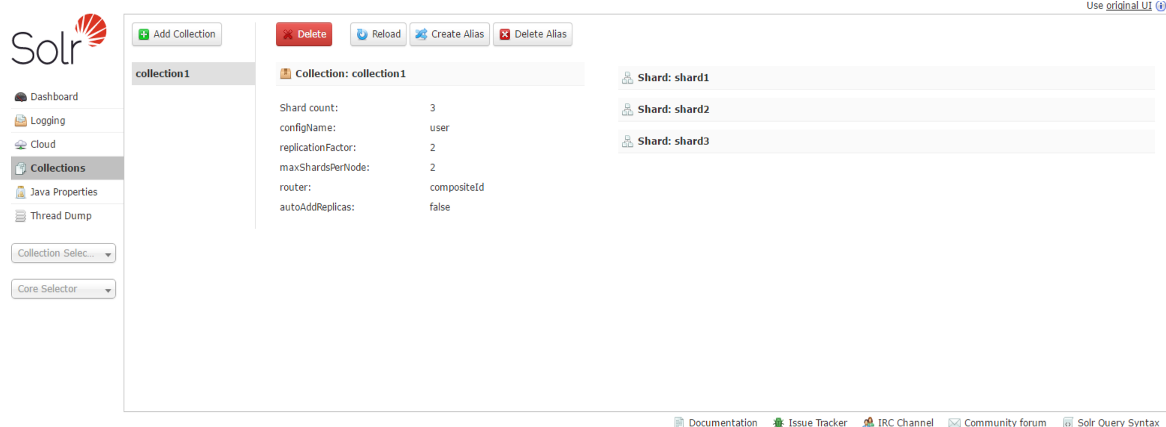
图3-3 Cloud 集群展示



3.1.4 Collections 界面展示

Collections 界面可以对相应 Collection 进行重载、重命名以及添加新索引等操作，如图 3-4 所示。

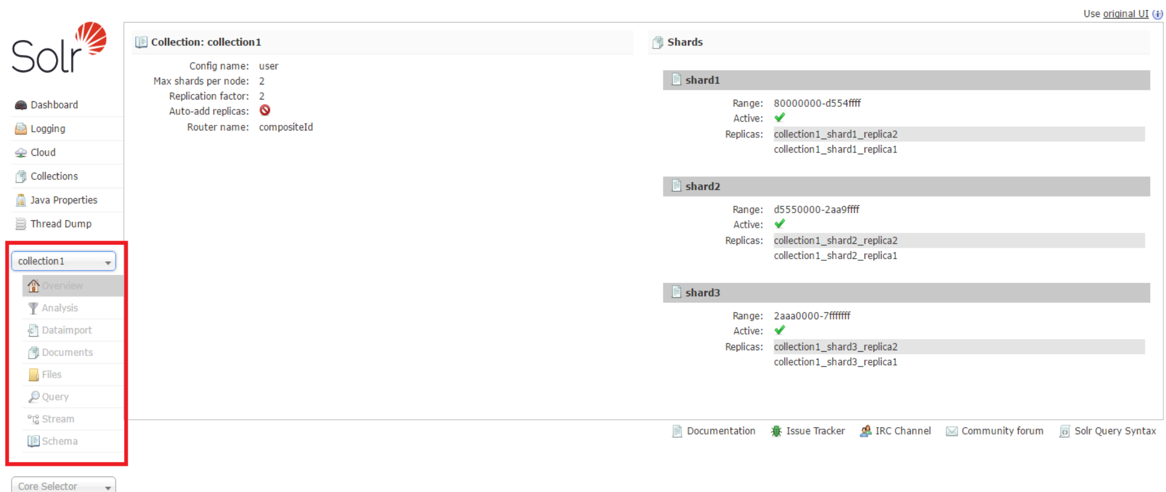
图3-4 Collections 界面展示



3.1.5 索引工具

索引工具包含查询、插入数据等 API 功能的界面化操作，如图 3-5 所示。

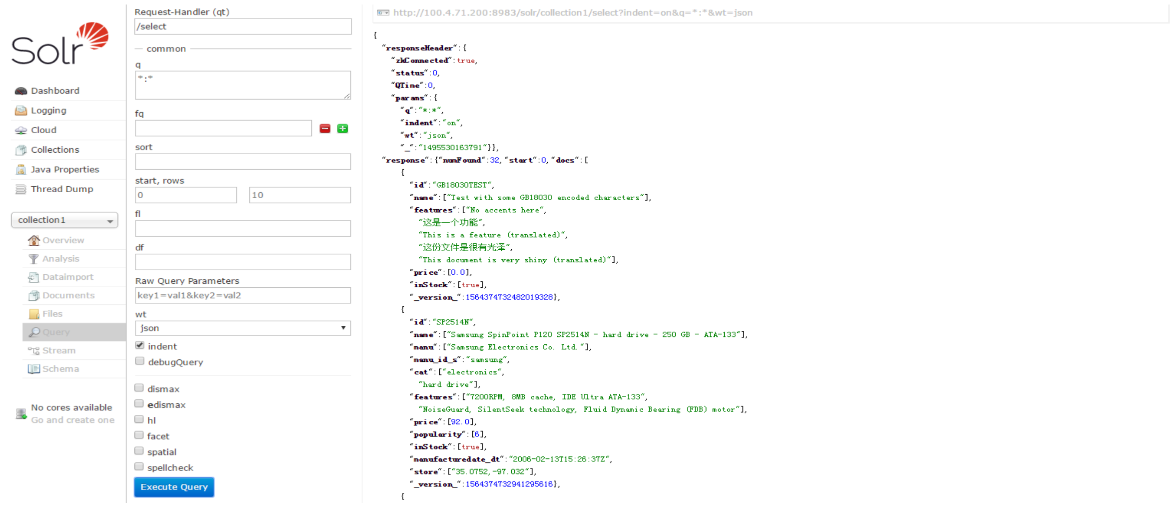
图3-5 索引工具



3.1.6 查询界面

查询界面（Query）是索引工具中使用最多的一部分，如图 3-6 所示。

图3-6 查询界面



3.2 Solr集群扩容

Solr 集群扩容是指在某节点上新增安装 SOLR_SERVER 进程。

3.2.1 使用场景

Solr 集群扩容的场景主要有两种：

- Solr 集群节点物理资源消耗过大，即 Solr 的服务节点的 CPU、内存占用率过高、磁盘空间不足场景。
- Solr 集群索引分片数过多，读写性能不足。

3.2.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在集群节点上新增安装 SOLR_SERVER：

- 如果集群中有节点没有安装 SOLR_SERVER 进程，直接在集群节点中添加 SOLR_SERVER 进程。
- 如果集群中所有节点均已安装 SOLR_SERVER 进程，进行 Solr 扩容前则需要先添加主机。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Solr 组件的状态是否正常。

(2) 进入 Solr 组件详情页，查看 Solr 的部署拓扑，确保集群中每个服务的状态正常，SOLR_SERVER 处于“已启动”状态。

3.2.3 扩容约束

- 请保障扩容节点操作系统版本与集群内部版本保持一致。

- 扩容操作一旦开始，不支持中止。

3.2.4 扩容时禁止的操作

为了避免数据丢失，Solr 组件扩容期间，不要进行创建索引操作。

3.2.5 扩容操作指导

注意

若集群中所有节点均已安装 SOLR_SERVER，进行 SOLR_SERVER 扩容前则需要先添加主机，然后再进行 SOLR_SERVER 扩容。如果集群中有扩容所用主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 Solr 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-7 所示。

a. 选择进程及主机

在选择进程项的列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

b. 部署进程

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-7 添加进程

(3) 查看进程变化

Solr 扩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 SOLR_SERVER 安装数量的变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.2.6 Solr 扩容后操作

Solr 集群扩容后，用户可根据实际业务需要来调整 Collection 的 Shard 个数，切分 Shard 充分利用集群的水平扩展能力，使 Solr 集群索引数据效率更高，查询响应更快。假如 Collection 创建时，Shard 个数较少，但预估的数据量比较大，如此会造成单个 Shard 比较大，这种情况下，可以增加相应 Shard 个数，将数据分布开来；或者实际业务需要指定满足某些条件的数据写入不同的 Shard，而相应的 Shard 无法提前确认，则需要能够动态的增加 Shard 来满足业务需要。具体操作请参见 3.4 分片扩容与缩容。

3.2.7 扩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Solr 组件检查，确保 Solr 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 Solr 组件部署拓扑里是否已有新增的扩容节点。

(4) 打开 Solr 快速链接，在 Solr UI 页面可查看新扩容的 SOLR 节点的信息。

3.3 Solr集群缩容

Solr 集群缩容是指将某节点上已安装的 SOLR_SERVER 删除。

3.3.1 使用场景

说明

缩容前，必须确认不存在某个 collection 只有一个 replica 并且该 collection 的部分 shard 分布在将要缩容的节点上的情况。

Solr 集群缩容的场景主要有：

- 初始 SOLR_SERVER 节点规划不合理。
- 当 SOLR_SERVER 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.3.2 缩容前准备

1. 环境检查

(1) 登录大数据平台管理系统，查看 Solr 组件的状态是否正常。

(2) 进入 Solr 组件详情页，查看 Solr 的部署拓扑，确保集群中每个服务的状态正常，SOLR_SERVER 处于“已启动”状态。

2. 数据迁移

Solr 集群缩容，可能会导致待缩容节点分片副本丢失，影响集群高可靠性，故需要将此节点上的分片副本迁移至其它节点。

(1) 登录 Solr 集群任一节点，执行数据迁移操作。

(2) 数据迁移命令如下：

```
curl 'http://node1.hde.com:8983/solr/admin/collections?action=REPLACENODE&source=node1.hde.com:8983_solr&target=node2.hde.com:8983_solr'
```

其中：

- action= REPLACENODE：表示该命令对节点上所有分片进行迁移。
- source=node1.hde.com:8983_solr：表示分片迁移的源节点。
- target=node2.hde.com:8983_solr：表示分片迁移的目的节点。

3.3.3 缩容约束

缩容操作一旦开始，不支持中止。

3.3.4 缩容影响

Solr 查询并发降低，本地索引存储空间减少。

3.3.5 缩容时禁止的操作

为了避免数据丢失，Solr 缩容期间，不要进行创建索引操作。

3.3.6 缩容操作指导

说明

SOLR_SERVER 缩容操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行 SOLR_SERVER 缩容操作”为例进行说明，在主机详情页面执行 SOLR_SERVER 缩容操作，与其类似不再进行说明。

缩容操作步骤如下：

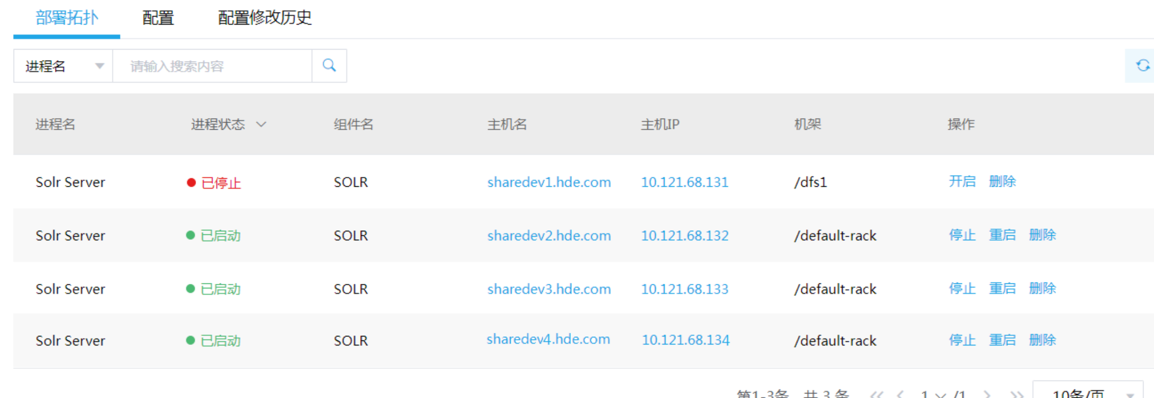
(1) 在 Solr 组件详情页面，选择缩容节点。

(2) 在组件详情页面[部署拓扑]页签下，选择已安装 SOLR_SERVER 进程且需要缩容的主机，然后单击该进程右侧操作中的<停止>按钮，停止 SOLR_SERVER。

(3) 删除 SOLR_SERVER

停止 SOLR_SERVER 后，如图 3-8 所示，在该进程右侧操作中单击<删除>按钮，即可完成 SOLR_SERVER 缩容。

图3-8 删除 SOLR_SERVER



(4) 查看进程变化

Solr 缩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 SOLR_SERVER 安装数量的变化以及状态。

(5) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.3.7 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Solr 组件检查，确保 Solr 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 Solr 组件部署拓扑里相关缩容节点是否已经删除。

(4) 打开 Solr 快速链接，在 Solr UI 页面查看 Solr 节点的最新状态信息是否符合预期。

3.4 分片扩容与缩容

1. Implicit 扩容与缩容

在创建 collections 的时候（如果指定 numshards 参数会自动切换到 router="compositeId"），如果采用 compositeId 方式，那么就不能动态改变 shard。如果采用的是 implicit 方式，就可以动态的增加与删除 shard。下面举例说明如何根据 implicit 路由方式增加与删除分片。

- 使用下面的命令创建 collection，指定路由方式为“implicit”，并创建三个分片名字分别为 2010、2011、2012，并指定“year”为路由字段来决定文档存放在哪个分片中。

  curl 'http://<Solr_IP>:8983/solr/admin/collections?action=CREATE&name=Test&router.name=implicit&shards=2010,2011,2012&router.field=year&propery.name=testcoll&wt=json'

- 使用下面的命令在 Test 索引中增加一个名字为“2013”的分片，所有 year 为 2013 的文档都会被索引到该分片中，以此达到分片扩容的目的。

  curl 'http://<Solr_IP>:8983/solr/admin/collections?action=CREATESHARD&collection=Test&shard=2013'

- 使用下面的命令在 Test 索引中删除名字为“2010”的分片，当某些场景中分片 2010 中的数据不再需要的时候可以删除该分片以达到分片缩容的目的。

  curl
  'http://<Solr_IP>:8983/solr/admin/collections?action=DELETESHARD&collection=Test&shard=2010'

2. CompositeID 扩展

当索引达到一定数量级的时候，每个 shard 也会变得很大，搜索的速度会达到一个瓶颈，这时可以对 shard 进行拆分，使单个 shard 中的数据量变少，从而达到集群纵向扩展的目的。只有使用 CompositeID 的 collection 才可以进行拆分。Split 能够在不影响索引正常提供服务的时候进行，但索引过大的时候，推荐在服务器空闲时进行，因为 Split 会做大量的 I/O 读写操作。在浏览器输入以下地址，可以对 collection 的 shard 进行分割。

curl
'http://<Solr_IP>:8983/solr/admin/collections?action=SPLITSHARD&collection=<Collection>&shard=<shard>'

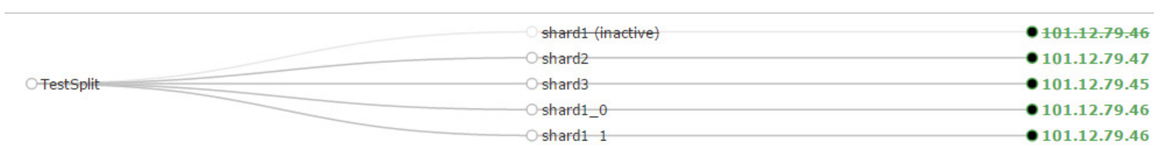
- <Solr_IP>：Solr 节点的 IP 地址
- <Collection>：将要进行分割的 collection 名称
- <shard>：将要进行分割的分片的名称

示例：以下对 TestSplit 索引的 shard1 进行分割。

curl
'http://101.12.79.45:8983/solr/admin/collections?action=SPLITSHARD&collection=TestSplit&shard=shard1'

分割完成之后索引的结构如图 3-9 所示，可以看到原来的 shard1 已经不可用，被分割为 shard1_0 和 shard1_1。

图3-9 索引分布结构



3.5 权限访问控制

注意

仅开启“Kerberos 认证”与“权限与密钥管理”的集群可以配置 Solr 组件的权限。

集群新建用户的组件权限会因为集群是否开启权限管理功能而有所不同：

- 未开启权限管理时，用户不需授权，可直接进行索引的创建、删除、插入、查询等操作。
- 开启权限管理后，组件权限需通过[集群权限/角色管理]中的角色分配给用户，用户通过绑定角色进行赋权后，才能对组件执行操作。
- 开启权限管理后，大部分 Solr 接口受权限管控，能够满足使用需要。

权限管理是安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

3.5.1 权限说明

在开启权限管理的集群中，用户需被赋予相应权限才可以对 Solr collection 执行相应的操作。

Solr 支持对索引的操作权限类型包括 query、update、solr_admin、others。Solr 具体操作所需权限对应关系如表 3-1 所示。

表3-1 Solr 权限说明

权限类型	对应的组件常用操作
query	查询索引中的数据，如/select
update	更新索引中的数据，如/update
solr_admin	创建、删除索引等，另外默认包含query、update、others权限
others	包含/query、/admin/file、/admin/mbeans等

3.5.2 权限使用操作示例

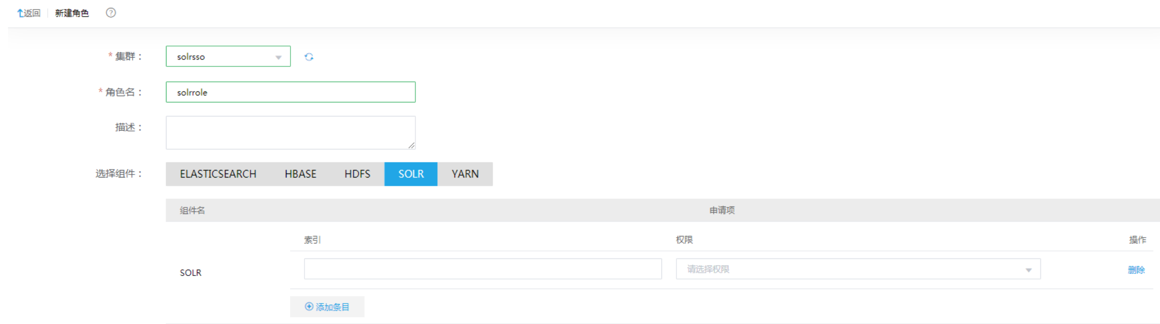
下面介绍如何对用户授予 Solr 操作权限以及 query、update、solr_admin、others 四种权限类型的简单验证。

1. 新建角色并将角色绑定给用户

(1) 新建角色

在[集群权限/角色管理]页面，创建角色 solrrole，不选择任何组件权限，如图 3-10 所示。

图3-10 新建 solrrole 角色



(2) 新建用户，并为用户授予角色

在[集群权限/用户管理]页面，创建用户 user01，并为用户授权角色 solrrole，如图 3-11 所示。

图3-11 用户授予角色

(3) 用户认证，并进行 Solr 相关操作

在集群节点内部，对 user01 用户进行 Kerberos 认证，然后通过 curl 命令执行相关操作。详细操作实例请参见 2. solr_admin、update、query、others 四种权限类型验证示例。

2. solr_admin、update、query、others 四种权限类型验证示例

(1) solr_admin 权限验证

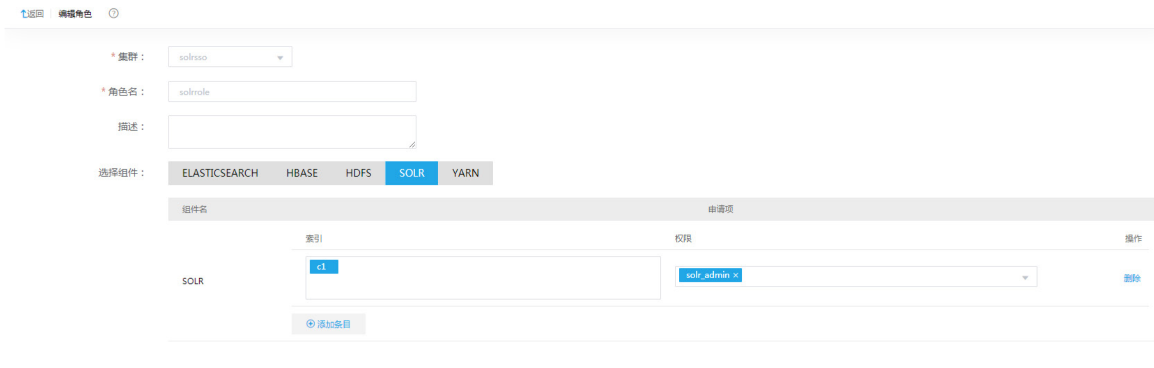
a. 授权前创建名称为 c1 的索引，界面提示请求未授权，如图 3-12 所示。

图3-12 创建失败提示

```
[root@node50 ~]# curl --negotiate -u : 'http://node50:8983/solr/admin/collections?action=CREATE&name=c
1&numShards=1&replicationFactor=1&collection.configName=_default&wt=json'
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8"/>
<title>Error 403 Unauthorized request, Response code: 403</title>
</head>
<body><h2>HTTP ERROR 403</h2>
<p>Problem accessing /solr/admin/collections. Reason:
<pre>    Unauthorized request, Response code: 403</pre></p>
</body>
</html>
```

b. 在[集群权限/角色管理]页面，修改角色 solrrole 权限设置，授予 solrrole 角色对索引 c1 有 solr_admin 权限，如图 3-13 所示。

图3-13 角色绑定权限



c. 授权后重新执行创建操作，创建成功，如图 3-14 所示。

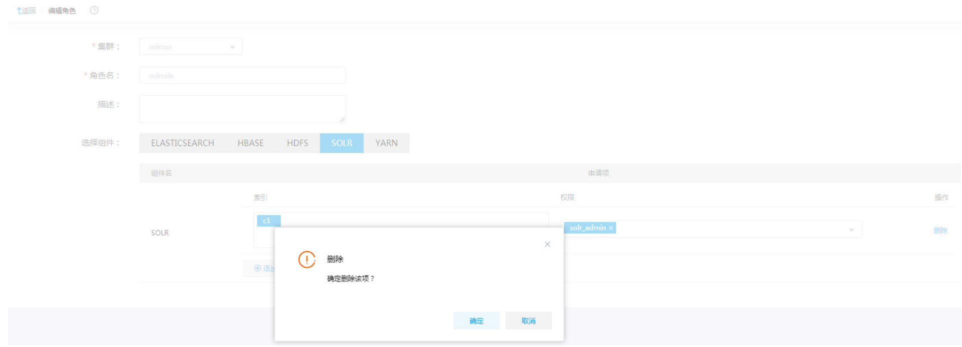
图3-14 索引创建成功

```
[root@node50 ~]# curl --negotiate -u : 'http://node50:8983/solr/admin/collections?action=CREATE&name=c
1&numShards=1&replicationFactor=1&collection.configName=_default&wt=json'
{
  "responseHeader":{
    "status":0,
    "QTime":3546},
  "success":{
    "node51.hde.com:8983_solr":{
      "responseHeader":{
        "status":0,
        "QTime":2167},
      "core":"c1_shard1_replica_n1"}},
  "warning":"Using _default configset. Data driven schema functionality is enabled by default, which i
s NOT RECOMMENDED for production use. To turn it off: curl http://{host:port}/solr/c1/config -d '{\"se
t-user-property\": {\"update.autoCreateFields\":\"false\"}}'"}
```

(2) update 权限验证

a. 在[集群权限/角色管理]页面，点击 solrrole 进入角色详情页，然后单击页面右上角<编辑>按钮，对 solrrole 角色权限进行编辑，删除上一步骤添加的 solr_admin 条目，如图 3-15 所示。

图3-15 删除授权



b. 授权前，user01 用户执行/update 进行更新操作，终端界面提示请求未授权，如图 3-16 所示。

图3-16 未授权提示

```
[root@node50 ~]# curl --negotiate -u : http://node50:8983/solr/c1/update?commitWithin=1000 -d '[{"id":
"1"}]'
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8"/>
<title>Error 403 Unauthorized request, Response code: 403</title>
</head>
<body><h2>HTTP ERROR 403</h2>
<p>Problem accessing /solr/c1/update. Reason:
<pre>    Unauthorized request, Response code: 403</pre></p>
</body>
</html>
```

c. 为角色 solrrole 赋予 update 索引 c1 的权限，然后再执行/update 更新操作，更新成功，如图 3-17 所示。

图3-17 更新成功

```
[root@node50 ~]# curl --negotiate -u : http://node50:8983/solr/c1/update?commitWithin=1000 -d '[{"id":
"1"}]'
{
  "responseHeader":{
    "status":0,
    "QTime":2}}
```

(3) query 权限验证

a. 编辑 solrrole 角色权限，对 solrrole 角色清除 update 权限，通过用户 user01 对索引 c1 执行/select 操作，查询失败，如图 3-18 所示。

图3-18 未授权提示

```
[root@node50 ~]# curl --negotiate -u : http://node50:8983/solr/c1/select?q=*
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8"/>
<title>Error 403 Unauthorized request, Response code: 403</title>
</head>
<body><h2>HTTP ERROR 403</h2>
<p>Problem accessing /solr/c1/select. Reason:
<pre>    Unauthorized request, Response code: 403</pre></p>
</body>
</html>
```

b. 为 solrrole 角色授予 query 权限，用户 user01 对索引 c1 执行/select 操作，查询成功，如图 3-19 所示。

图3-19 查询成功

```
[root@node50 ~]# curl --negotiate -u : http://node50:8983/solr/c1/select?q=*
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":20,
    "params":{
      "q":"*"}},
  "response":{"numFound":1,"start":0,"docs":[
      {
        "id":"1",
        "_version_":1680419230226317312}]
  }}
```

(4) other 权限验证

a. 对 solrrole 角色清除 query 权限，通过用户 user01 对索引 c1 执行“/query”操作，界面提示未授权，如图 3-20 所示。

图3-20 未授权提示

```
[root@node50 ~]# curl --negotiate -u : 'http://node50:8983/solr/c1/query' -d 'q=*'
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8"/>
<title>Error 403 Unauthorized request, Response code: 403</title>
</head>
<body><h2>HTTP ERROR 403</h2>
<p>Problem accessing /solr/c1/query. Reason:
<pre>    Unauthorized request, Response code: 403</pre></p>
</body>
</html>
```

b. 为 solrrole 角色授予 other 权限，用户 user01 对索引 c1 执行/query 操作，执行成功，如图 3-21 所示。

图3-21 查询成功

```
[root@node50 ~]# curl --negotiate -u : 'http://node50:8983/solr/c1/query' -d 'q=*'
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":1,
    "params":{
      "q":"*"}},
  "response":{"numFound":1,"start":0,"docs":[
      {
        "id":"1",
        "_version_":1680419230226317312}]
  }}
```

3.6 数据备份

3.6.1 创建备份

使用下面的命令创建备份。

curl
'http://<Solr_IP>:8983/solr/<Collection>/replication?command=backup&name=test001&location=<location>'

其中：

- <Solr_IP>：Solr 节点的 IP 地址。
- <Collection>：要备份的 collection 名称。
- <location>：备份存放的路径，注意集群中 solr 用户需要对该路径有写入的权限。

图3-22 创建备份

```
[root@sharedev1 opt]# curl --negotiate -u : http://sharedev1.hde.com:8983/solr/t
est01/replication?command=backup&name=test01\&location=/opt/solr-repl/
[1] 55998
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "status":"OK"}
[root@sharedev1 opt]# ll solr-repl/
total 0
drwxr-xr-x 2 solr hadoop 257 Oct 15 11:05 snapshot.test01
```

3.6.2 恢复备份

使用下面的命令恢复备份。

curl
'http://<Solr_IP>:8983/solr/<Collection>/replication?command=restore&name=test001&location=<location>'

其中：

- <Solr_IP>：Solr 节点的 IP 地址。
- <Collection>：索引将要恢复到的 collection。
- <location>：RESTORE 命令要读取的备份的存放路径。

图3-23 恢复备份

```
[root@sharedev1 opt]# curl --negotiate -u : http://sharedev1.hde.com:8983/solr/t
est01/replication?command=restore&name=test01\&location=/opt/solr-repl/
[1] 56598
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "status":"OK"}
```

4 开发指南

4.1 API使用

4.1.1 增删改操作

- Solr 对外提供标准的 http 接口来实现对数据索引的增加、删除、修改和查询。在 Solr 中，用户通过向部署在 servlet 容器中的 Solr Web 应用程序发送 HTTP 请求来启动索引和搜索。Solr 接收到请求，使用适当的 SolrRequestHandler 来处理请求，然后通过 HTTP 的方式返回响应。
- Solr 使用 add/update 标签来添加或更新文档，并且在提交后能搜索到这些添加和更新的文档。Solr 使用 commit 标签使上次 commit 以来所做的所有更改都可以搜索到。Solr 使用 delete 标签来删除特定 id 的文档；按照查询删除的方式将删除查询返回的所有文档。Solr 支持 JSON，XML，CSV 等格式的文档增删改交互。可以使用如下实例进行增删改操作。

1. Update

(1) 使用 XML 格式的文档实现对数据索引的更新，在 XML 文档中使用如下格式，可添加一条文档记录，并使用可选参数 boost（boost 参数可以增加检索时计算相关度评分）。

```
<add>
  <doc boost="2.5">
    <field name="employeeId">05991</field>
    <field name="office" boost="2.0">Bridgewater</field>
  </doc>
</add>
```

(2) 在 XML 文档中使用如下格式可以添加一条文档记录。

```
<add>
  <doc>
    <field name="employeeId">05991</field>
    <field name="office" update="set">Walla Walla</field>
    <field name="skills" update="add">Python</field>
  </doc>
</add>
```

(3) 在 XML 文档中使用如下格式可以添加一个多值（数组）的文档。

```
<add>
  <doc>
    <field name="employeeId">05991</field>
    <field name="skills" update="set">Python</field>
    <field name="skills" update="set">Java</field>
    <field name="skills" update="set">Jython</field>
  </doc>
</add>
```

(4) 在 XML 文档中使用如下格式可以添加一个空值 null 文档。

```
<add>
```

```
    <doc>
      <field name="employeeId">05991</field>
      <field name="skills" update="set" null="true" />
    </doc>
  </add>
```

(5) 使用 Linux 命令 curl 提交 REST 请求。

```
curl 'http://localhost:8983/solr/test/update?commit=true' -H "Content-Type: text/xml"
--data-binary '<add><doc><field name="id">testdoc</field></doc></add>'
```

2. Delete

使用 DELETE 可以对索引数据进行删除。

- 使用如下命令可以删除指定 id 的文档

  <delete><id>05991</id></delete>

- 使用如下命令可以删除指定 query 的文档

  <delete><query>office:Bridgewater</query></delete>

3. JSON 格式的使用方式

```
curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/my_collection/update' --data-binary '
{
  "add": {
    "doc": {
       "id": "DOC1",
       "my_boosted_field": {        /* use a map with boost/value for a boosted field */
       "boost": 2.3,
       "value": "test"
      },
      "my_multivalued_field": [ "aaa", "bbb" ]   /* Can use an array for a multi-valued field
*/
    }
  },
  "add": {
    "commitWithin": 5000,        /* commit this document within 5 seconds */
    "overwrite": false,      /* don't check for existing documents with the same uniqueKey
*/
    "boost": 3.45,         /* a document boost */
    "doc": {
      "f1": "v1",                 /* Can use repeated keys for a multi-valued field */
      "f1": "v2"
    }
  },

  "commit": {},
  "optimize": { "waitSearcher":false },

  "delete": { "id":"ID" },       /* delete by ID */
  "delete": { "query":"QUERY" } /* delete by query */
}'
```

4.1.2 管理常用操作

Solr 支持通过 http 请求对 core 进行一系列的 admin 操作。

1. Core Admin API

- STATUS：查看状态

  http://<Solr_IP>:8983/solr/admin/cores?action=STATUS&core=core0

- CREATE：创建 core
  - 创建 core，命令如下：

    http://<Solr_IP>:8983/solr/admin/cores?action=CREATE&name=coreX&instanceDir=path/to/dir&config=config_file_name.xml&schema=schem_file_name.xml&dataDir=data
  - 根据配置创建 core，配置集 Config Sets 需已经存在：

    http://<Solr_IP>:8983/solr/admin/cores?action=CREATE&name=my_core&collection=my_collection&shard=shard2

注意

创建属于特定 collection 的 core，该 collection 需已经存在。

- RELOAD：重载 core

  http://<Solr_IP>:8983/solr/admin/cores?action=RELOAD&core=core0

  当 solr core 的配置发生变化，比如添加新的字段，需要重载。

- RENAME：重命名 core

  http://<Solr_IP>:8983/solr/admin/cores?action=RENAME&core=core0&other=newcore

  参数 core 是需要被重命名的 core，参数 other 是新名称。

- SWAP：交换名称

  http://<Solr_IP>:8983/solr/admin/cores?action=SWAP&core=core1&other=core0

  参数 core 和参数 other 是需要被交换的两个 core 名称。

- UNLOAD：卸载

  http://<Solr_IP>:8983/solr/admin/cores?action=UNLOAD&core=core0

  卸载操作将从 solr 中移除该 core。

- MERGEINDEXES：合并索引

  http://<Solr_IP>:8983/solr/admin/cores?action=MERGEINDEXES&core=new_core_name&indexDir=/solr_home/core1/data/index&indexDir=/solr_home/core2/data/index

  这种方式可以合并任意基于 Lucene 的索引，而不仅是 solr core。

  http://<Solr_IP>:8983/solr/admin/cores?action=mergeindexes&core=new_core_name&srcCore=core1&srcCore=core2

  这种方式允许 target solr core 不在同一台服务器上。

- SPLIT：拆分索引

  http://<Solr_IP>:8983/solr/admin/cores?action=SPLIT&core=core0&targetCore=core1

&targetCore=core2

core index 将被拆分为两个以 targetCore 指定的索引。

http://<Solr_IP>:8983/solr/admin/cores?action=SPLIT&core=core0&path=/path/to/index/1&path=/path/to/index/2

core index 将被拆分为两个以 path 指定的索引。

- REQUESTSTATUS：请求状态

  http://<Solr_IP>:8983/solr/admin/cores?action=REQUESTSTATUS&requestid=1

  返回被提交的 CoreAdmin 异步请求状态。
- REQUESTRECOVERY：请求恢复

  http://<Solr_IP>:8983/solr/admin/cores?action=REQUESTRECOVERY&core=gettingstarted_shard1_replica1

  用于当 SolrCloud replica 无法自动变为 active 时，发起人为请求。

2. Collection Admin API

Solr 支持通过 http 请求对 collection 进行一系列的 admin 操作，具体命令如下：

- 创建 collection

  http://<Solr_IP>:8983/solr/admin/collections?action=CREATE
- RELOAD collection 命令

  http://<Solr_IP>:8983/solr/admin/collections?action=RELOAD
- 对 collection 的 shard 进行拆分

  http://<Solr_IP>:8983/solr/admin/collections?action=SPLITSHARD
- 创建数据 collection 的新 shard

  http://<Solr_IP>:8983/solr/admin/collections?action=CREATESHARD: create a new shard for collection that use the 'implicit' router
- 删除一个不活跃的 shard

  http://<Solr_IP>:8983/solr/admin/collections?action=DELETESHARD: delete an inactive shard
- 创建 collection 的别名

  http://<Solr_IP>:8983/solr/admin/collections?action=CREATEALIAS: create or modify an alias for a collection
- 删除 collection 的别名

  http://<Solr_IP>:8983/solr/admin/collections?action=DELETEALIAS: delete an alias for a collection
- 删除 collection

  http://<Solr_IP>:8983/solr/admin/collections?action=DELETE: delete a collection
- 删除 collection 中 shard 的 replica

  http://<Solr_IP>:8983/solr/admin/collections?action=DELETEREPLICA: delete a replica of a shard
- 给 collection 中的某个 shard 添加 replica

  http://<Solr_IP>:8983/solr/admin/collections?action=ADDREPLICA: add a replica of a shard
- 获取 collecion 的状态

http://<Solr_IP>:8983/solr/admin/collections?action=CLUSTERSTATUS: Get cluster status including collections, shards, replicas, configuration name as well as collection aliases and cluster properties

- 查看当前集群中有哪些 collection

  http://<Solr_IP>:8983/solr/admin/collections?action=LIST: List all collections

4.1.3 查询 Search

查询示例命令如下：

http://<Solr_IP>:8983/solr/my_collection/select?q=title:m*&rows=20&wt=json&indent=true

1. 常用关键字

表4-1 常用关键字

关键字	说明
q	Solr中用来搜索的查询。可以通过追加一个分号和已索引且未进行断词的字段的名称来包含排序信息。默认的排序是score desc，指按记分降序排序
start	将初始偏移量指定到结果集中。可用于对结果进行分页。默认值为0。如start=15 返回从第15个结果开始的结果
rows	返回文档的最大数目，默认值为10
fq	提供一个可选的筛选器查询。查询结果被限制为仅搜索筛选器查询返回的结果，筛选过的查询由Solr进行缓存，它们对提高复杂查询的速度非常有用
hl	当 hl=true 时，在查询响应中醒目显示片段。默认为 false。参看醒目显示参数上的Solr Wiki部分可以查看更多选项
fl	作为逗号分隔的列表指定文档结果中应返回的Field集。默认为"*"，指所有的字段。"score"指还应返回记分
q.op	表示q中查询语句的各条件的逻辑操作 AND(与) OR(或)
hl.fl	高亮field，hl.fl=Name,SKU
hl.snippets	默认是1，这里设置为3个片段
hl.simple.pre	高亮前面的格式
hl.simple.post	高亮后面的格式
facet	是否启动统计
facet.field	统计field

2. 运算符参数

表4-2 运算符参数

关键字	说明
:	指定字段查指定值，如返回所有值*:*
?	表示单个任意字符的通配
*	表示多个任意字符的通配（不能在检索的项开始使用*或者?符号）
AND、||、OR、&&、NOT、!、+、-	-（排除操作符不能单独与项使用构成查询） “+”存在操作符，要求符号”+”后的项必须在文档相应的域中存在
~	表示模糊检索，如检索拼写类似于“roam”的项这样写：roam~将找到形如foam和roams的单词；roam~0.8，检索返回相似度在0.8以上的记录邻近检索，如检索相隔10个单词的“apache”和“jakarta”，“jakarta apache”~10
^	控制相关度检索，如检索jakarta apache，同时希望去让“jakarta”的相关度更加好，那么在其后加上“^”符号和增量值，即jakarta^4 apache
( )	用于构成子查询
[]	包含范围检索，如检索某时间段记录，包含头尾，date:[200707 TO 200710]
{}	不包含范围检索，如检索某时间段记录，不包含头尾date:{200707 TO 200710}

3. 高亮参数

表4-3 高亮参数

关键字	说明
hl-highlight	h1=true，表示采用高亮。可以用h1.fl=field1,field2来设定高亮显示的字段
hl.fl	用空格或逗号隔开的字段列表。要启用某个字段的highlight功能，就得保证该字段在schema中是stored。如果该参数未被给出，那么就会高亮默认字段。standard handler会用df参数，dismax字段用qf参数。可以使用星号去方便的高亮所有字段。如果使用了通配符，那么要考虑启用hl.requiredFieldMatch选项
hl.requireFieldMatch	如果值为true，除非该字段的查询结果不为空才会被高亮。它的默认值是false，意味着它可能匹配某个字段却高亮一个不同的字段。如果hl.fl使用了通配符，那么就要启用该参数。尽管如此，如果查询是all字段（可能是使用 copy-field 指令），那么还是把它设为false，这样搜索结果能表明哪个字段的查询文本未被找到
hl.usePhraseHighlighter	如果一个查询中含有短语（引号框起来的）那么会保证一定要完全匹配短语的才会被高亮
hl.highlightMultiTerm	如果使用通配符和模糊搜索，那么会确保与通配符匹配的term会高亮。默认为false，同时hl.usePhraseHighlighter要为true
hl.snippets	这是highlighted片段的最大数。默认值为1，也几乎不会修改。如果某个特定的字段的该值被置为0（如f.allText.hl.snippets=0），这就表明该字段被禁用高亮了
hl.fragsize	每个snippet返回的最大字符数。默认是100。如果为0，那么该字段不会被fragmented且整个字段的值会被返回
hl.mergeContiguous	如果被置为true，当snippet重叠时会merge起来
hl.maxAnalyzedChars	会搜索高亮的最大字符，默认值为51200，如果需要禁用，设为-1

关键字	说明
hl.alternateField	如果没有生成snippet（没有terms匹配），那么使用另一个字段值作为返回
hl.maxAlternateFieldLength	如果hl.alternateField启用，则有时需要制定alternateField的最大字符长度，默认0是没有限制。所以合理的值应该为hl.snippets * hl.fragsize，这样返回结果的大小就能保持一致
hl.formatter	提供可替换的formatting算法的扩展点。默认值是simple
hl.fragmenter	solr制定fragment算法的扩展点。gap是默认值
hl.regex.pattern	正则表达式的pattern
hl.regex.slop	这是hl.fragsize能变化以适应正则表达式的因子。默认值是0.6，意思是如果hl.fragsize=100那么fragment的大小会从40-160

4. 分组查询

- Field Facet
  - Facet 字段通过在请求中加入“facet.field”参数加以声明，如果需要对多个字段进行 Facet 查询，那么将该参数声明多次。

    例如，通过 CPU 与 videoCard 对联想电脑进行分组查询：

    /select?q=联想&facet=on&facet.field=cpu&facet.field=videoCard。

    各个 Facet 字段互不影响，且可以针对每个 Facet 字段设置查询参数。
  - 表 4-4 介绍的参数既可以应用于所有的 Facet 字段，也可以应用于每个单独的 Facet 字段。应用于单独的字段时通过“f.字段名.参数名=参数值”这种方式调用，比如 facet.prefix 参数应用于 cpu 字段，可以采用如下形式：f.cpu.facet.prefix=Intel。

表4-4 参数说明

关键字	说明
facet.prefix	表示Facet字段值的前缀，比如“facet.field=cpu&facet.prefix=Intel”，那么对cpu字段进行Facet查询，返回的 cpu 都是以“Intel”开头的，“AMD”开头的cpu型号将不会被统计在内
facet.sort	表示Facet字段值以哪种顺序返回。可接受的值为true(count)|false(index, lex)。true(count)表示按照count值从大到小排列。false(index, lex)表示按照字段值的自然顺序 (字母，数字的顺序)排列。默认情况下为 true(count)。当 facet.limit 值为负数时，默认 facet.sort= false(index,lex)
facet.limit	限制Facet字段返回的结果条数，默认值为100。如果此值为负数，表示不限制
facet.offset	返回结果集的偏移量，默认为0。它与facet.limit配合使用可以达到分页的效果
facet.mincount	限制了Facet字段值的最小count，默认为 0。合理设置该参数可以将用户的关注点集中在少数比较热门的领域
facet.missing	默认为“”，如果设置为true或者on，那么将统计那些该Facet字段值为null的记录
facet.method	取值为enum或fc，默认为fc。该字段表示了两种Facet的算法，与执行效率相关。enum适用于字段值比较少的情况，比如字段类型为布尔型，或者字段表示中国的所有省份。Solr会遍历该字段的所有取值，并从filterCache里为每个值分配一个filter（这里要求solrconfig.xml里对filterCache的设置足够大。然后计算每个filter与主查询的交集。fc(表示Field Cache)适用于字段取值比较多，但在每个文档里出现次数比较少的情况。Solr会遍历所有的文档，在每个文档内搜索Cache内的值，如果找到就将Cache

关键字	说明
	内该值的count加1
facet.enum.cache.minDf	当facet.method=enum时，此参数中的minDf表示 minimum document frequency。也就是文档内出现某个关键字的最少次数，该参数默认值为0。设置该参数可以减少filterCache的内存消耗，但会增加总的查询时间（计算交集的时间增加了。如果设置该值的话，建议优先尝试25-50内的值）

- Date Facet
    - 日期类型的字段在文档中很常见，如商品上市时间，货物出仓时间，书籍上架时间等等。
    - 某些情况下需要针对这些字段进行 Facet，不过时间字段的取值有无限性，用户往往关心的不是某个时间点而是某个时间段内的查询统计结果，Solr 为日期字段提供了更为方便的查询统计方式。
    - 当然，字段的类型必须是 DateField(或其子类型)。需要注意的是，使用 Date Facet 时，字段名、起始时间、结束时间、时间间隔这 4 个参数都必须提供。与 Field Facet 类似，Date Facet 也可以对多个字段进行 Facet，并且针对每个字段都可以单独设置参数。

表4-5 参数说明

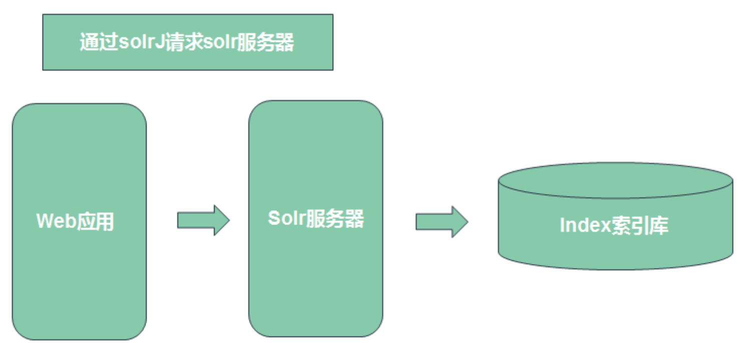
关键字	说明
facet.date	该参数表示需要进行Date Facet的字段名，与facet.field一样，该参数可以被设置多次，表示对多个字段进行Date Facet
facet.date.start	起始时间，时间的一般格式为“1995-12-31T23:59:59Z”，另外可以使用“NOW”、“YEAR”、“MONTH”等等
facet.date.end	结束时间
facet.date.gap	时间间隔。如果start为2009-1-1，end为2010-1-1。gap设置为“+1MONTH”表示间隔1个月，那么将会把这段时间划分为12个间隔段。注意“+”因为是特殊字符所以应该用“%2B”代替
facet.date.hardend	取值可以为true|false，默认为false。它表示gap迭代到end处采用何种处理。举例说明 start为 2009-1-1，end为2009-12-25，gap为“+1MONTH”，hardend为false的话最后一个时间段为2009-12-1至2010-1-1；hardend为true的话最后一个时间段为2009-12-1至2009-12-25
facet.date.other	取值范围为 before|after|between|none|all，默认为 none；before 会对 start 之前的值做统计；after 会对 end 之后的值做统计；between 会对 start 至 end 之间所有值做统计；如果 hardend 为 true 的话, 那么该值就是各个时间段统计值的和；none表示该项禁用；all 表示 before、after、all都会统计
facet.method	取值为enum或fc，默认为fc。该字段表示了两种Facet的算法，与执行效率相关。enum适用于字段值比较少的情况,比如字段类型为布尔型,或者字段表示中国的所有省份。Solr会遍历该字段的所有取值,并从filterCache里为每个值分配一个filter（这里要求solrconfig.xml里对filterCache的设置足够大。然后计算每个filter与主查询的交集。fc(表示Field Cache)适用于字段取值比较多，但在每个文档里出现次数比较少的情况。Solr会遍历所有的文档，在每个文档内搜索Cache内的值，如果找到就将Cache内该值的count加1
facet.enum.cache.minDf	当facet.method=enum时，此参数中的minDf表示minimum document frequency。也就是文档内出现某个关键字的最少次数，该参数默认值为0。设置该参数可以减少filterCache的内存消耗，但会增加总的查询时间（计算交集的时间增加了。如果设置该值的话，建议优先尝试25-50内的值）

4.1.4 客户端 API SolrJ 使用

1. SolrJ 介绍

SolrJ 是访问 Solr 组件的 JAVA 客户端，提供索引和搜索的请求方法，SolrJ 通常嵌入在业务系统中，通过 SolrJ 的 API 接口操作 Solr 组件。

图4-1 SolrJ 使用场景



SolrJ 连接 Solr 服务端有两种客户端，即 HttpSolrClient 和 CloudSolrClient，他们都是通过 HTTP 协议和 Solr 服务端进行通信，不同之处是前者显示配置一个 Solr 的 URL 地址，后者通过配置 zkHost 字符串进行连接。

(1) HttpSolrClient 连接

```
String urlString = "http://localhost:8983/solr/techproducts";
SolrClient solr = new HttpSolrClient.Builder(urlString).build();
```

(2) CloudSolrClient 连接

```
// Using a ZK Host String
String zkHostString = "zkServerA:2181,zkServerB:2181,zkServerC:2181/solr";
SolrClient solr = new CloudSolrClient.Builder().withZkHost(zkHostString).build();
// Using already running Solr nodes
SolrClient solr = new
CloudSolrClient.Builder().withSolrUrl("http://localhost:8983/solr").build();
```

2. 构建和运行 SolrJ 程序

SolrJ API 已经被包含在了 Solr 里面，不需要进行单独下载或者安装，在使用 SolrJ 访问 Solr 组件的时候，只需要从 Solr 组件目录的 classpath 中找到相应的 solr-solrj-x.y.z.jar，放到应用的 classpath 中。

例如：当使用 maven 构建自己的应用时，需要在自己项目的 pom.xml 中引入 solr-solrj 包。

```
<dependency>
<groupId>org.apache.solr</groupId>
<artifactId>solr-solrj</artifactId>
<version>x.y.z</version>
</dependency>
```

3. 使用 SolrJ 执行查询

(1) 使用默认 request handler 并设置查询字符串

```
SolrQuery query = new SolrQuery();
query.setQuery(mQueryString);
```

(2) 手动指定 request handler

```
query.setRequestHandler("/spellCheckCompRH");
```

(3) 查询时设置参数

```
query.set("fl", "category,title,price");
query.setFields("category", "title", "price");
query.set("q", "category:books");
```

(4) 提交查询

```
QueryResponse response = solr.query(query);
```

(5) 获取查询结果

```
SolrDocumentList list = response.getResults();
```

4. 使用 SolrJ 执行写入

添加文档：

```
SolrInputDocument document = new SolrInputDocument();
document.addField("id", "552199");
document.addField("name", "Gouda cheese wheel");
document.addField("price", "49.99");
UpdateResponse response = solr.add(document);
// Remember to commit your changes!
solr.commit();
```

5 最佳实践

5.1 MySQL数据同步

5.1.1 创建数据库

在 MySQL 数据库中创建一个名字为 solrdatabase 的数据库，并在该数据库中创建一个名字为 solrtable 的表，建表语句为：

create table solrtable (id int (20) primary key,name varchar(100), age int(20), updateTime timestamp);

在表中插入三条数据，如图 5-1 所示。

图5-1 Mysql 数据

```
mysql> select * from solrtable;
+----+------+------+---------------------+
| id | name | age  | updateTime          |
+----+------+------+---------------------+
|  1 | aa   |   18 | 2019-03-21 17:04:46 |
|  2 | bb   |   19 | 2019-03-21 17:04:50 |
|  3 | cc   |   20 | 2019-03-21 17:04:53 |
+----+------+------+---------------------+
3 rows in set (0.00 sec)
```

5.1.2 配置文件修改

所有配置文件的修改都基于某个安装有 Solr 节点的服务器的该目录下进行：/usr/hdp/3.0.1.0-187/solr/server/solr/configsets/_default/conf。

1. 修改 managed-schema

将源文件中的下面一行内容删掉：

```
<field name="id" type="string" indexed="true" stored="true" required="true" multiValued="false" />
```

并添加以下内容：

```
<field name="id" type="pint" indexed="true" stored="true" required="true" multiValued="false" />
<field name="name" type="string" indexed="true" stored="true" required="true" multiValued="false" />
<field name="age" type="pint" indexed="true" stored="true" required="true" multiValued="false" />
<field name="updateTime" type="string" indexed="true" stored="true" required="true" multiValued="false" />
```

2. 新建 data-config.xml

data-config.xml 内容如下所示，该配置文件中定义了 MySQL 数据库的信息：

```
<dataConfig>
  <dataSource name="db_1" type="JdbcDataSource"
          driver="com.mysql.jdbc.Driver"
          url="jdbc:mysql://101.12.100.102:3306/solrdatabase"
          user="root"
          password="passwd"
          />
  <document>
    <entity name="solrtable" dataSource="db_1" pk="id"
      query="select *from solrtable"
      deltaImportQuery="select *from solrtable where id='${dih.delta.id}'"
      deltaQuery="select id from solrtable where updateTime> '${dataimporter.last_index_time}'">
      <field column="id" name="id" />
      <field column="name" name="name" />
      <field column="age" name="age" />
    </entity>
  </document>
</dataConfig>
```

3. 修改 solrconfig.xml

在 solrconfig.xml 文件中添加以下内容：

```
<requestHandler name="/dataimport" class="org.apache.solr.handler.dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">data-config.xml</str>
  </lst>
</requestHandler>
```

4. 拷贝 jar 包

在所有安装 Solr 节点的服务器上执行以下操作：

- 拷贝 dataimport 包到指定目录：

  cp /usr/hdp/3.0.1.0-187/solr/dist/solr-dataimporthandler-*
  /usr/hdp/3.0.1.0-187/solr/server/solr-webapp/webapp/WEB-INF/lib

- 下载 mysql-connector-java-5.1.46-bin.jar 并拷贝至指定目录：

  cp /tmp/mysql-connector-java-5.1.46-bin.jar
  /usr/hdp/3.0.1.0-187/solr/server/solr-webapp/webapp/WEB-INF/lib

5.1.3 创建 collection

1. 上传配置文件

执行以下命令将配置文件上传到 Zookeeper 中：

/usr/hdp/3.0.1.0-187/solr/server/scripts/cloud-scripts/zkcli.sh -zkhost <ZKHOST>:2181/solr -cmd upconfig -confdir /usr/hdp/3.0.1.0-187/solr/server/solr/configsets/_default/conf/ -confname <Config_name>

其中：

- <ZKHOST>：大数据集群中某个装有 Zookeeper 的服务器的 IP 地址。
- <Config_name>：给配置文件指定的名称，在创建索引的时候需要用到。

2. 创建 collection

curl
'http://<Solr_IP>:8983/solr/admin/collections?action=CREATE&name=<Collection_name>&numShards=3
&replicationFactor=1&collection.configName=<Config_name>&wt=json'

其中：

- <Solr_IP>：Solr 节点的 IP 地址。
- <Collection_name>：创建的 collection 名称。
- <Config_name>：上一步指定的配置文件的名称。

5.1.4 导入数据

执行以下命令将 MySQL 中的数据导入到上一步创建的 collection 中：

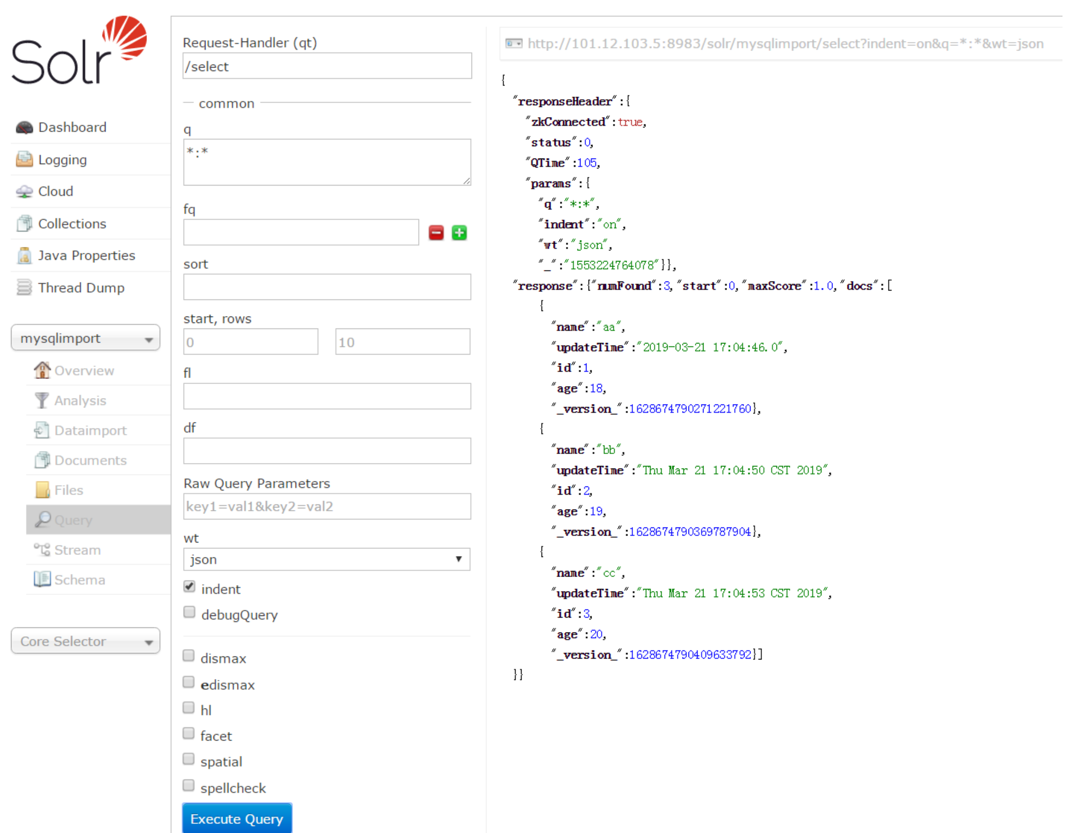
curl 'http://<Solr_IP>:8983/solr/<Collection_name>/dataimport?command=full-import&wt=json'

其中：

- <Solr_IP>：Solr 节点的 IP 地址。
- <Collection_name>：collection 的名称。

导入之后在 Solr web 管理页面可以看到数据导入成功。

图5-2 导入数据验证



6 常见问题解答

6.1 调优

6.1.1 Schema 优化

- index=true 比 index=false 在索引时占用更多的内存、索引合并和优化时间更长，索引体积也变的更大，如果不需要针对该域进行检索，可以设置为 index=false。
- 如果不关心 Term 在文档中出现的次数对最终文档的影响可以设置 omitNorms=true，即取消标准化因此对 score 的影响。它能减少磁盘空间的占用并加快索引速度。
- 如果不需要对该域进行高亮，你还可以设置 omitPositions=true 进一步减小索引体积。
- 如果只需要根据指定的 Term 找到对应的 document，不需要计算 Term 在 Document 中的出现频率来考虑每个索引文档的权重，可以设置 omitTermFreqAndPositions=true 即忽略 TF 计算以及 Term 在 TermVector 中的位置信息，这样能够进一步减小索引体积。
- 对于 stored 属性而言，在响应结果集中通过 FL 参数返回 stored=true 的域的执行开销很大，因为域值需要存储到硬盘写 I/O，查询时提取域值需要磁盘读 I/O，如果不需要存储可以设置 stored=false，进一步优化索引的体积。
- 如果想要存储的域值长度并不大，但是为了能够缓解提取存储域带来的磁盘 I/O，此时可以设置 compressed=true 即启用域值数据压缩。开启 compressed 会降低磁盘 I/O 但会增大 CPU 开销。
- 如果并不是一直都需要使用存储域，可以设置域延迟加载，尤其是开启了域值数据压缩。开启延迟加载之后，要返回的字段会被 SetNonLazyFieldSelector 立即加载，其他的域为延迟加载。启用域延迟加载，需要在 solrconfig.xml 中进行如下配置：<enableLazyFieldLoading>false</enableLazyFieldLoading>。
- 如果域值很大，可以使用 ExternalFileField 域(外部文件)，它不支持 Solr 查询，只能用于显示和 function 计算，还可以将域值存储在外部系统，比如 Redis 等，当需要域值的时候根据 Solr 的 UniqueKey 去缓存中提取。
- 对于 Java 里的日期时间类型的数据，建议使用 Solr 里的 date 域类型，如果需要进行日期时间范围区间查询，那么建议使用 Solr 里的 date 域类型，而不是使用 string 域类型。
- 可以对 facet 域、排序域设置为 docValue=true，它将会生成一个额外的正排表，会提升分面和排序的效率。

6.1.2 索引更新与提交调优

- 不建议使用显示硬提交，建议在 solrConfig 里面配置自动软/硬提交方式。
- 客户端在提交索引文档的时，建议使用批量软提交的方式添加索引文档。
- 单机模式下，在提交索引的时候建议使用 ConcurrentUpdateSolrClient 类，对于 solrCloud 模式下建议使用 CloudSolrClient 类来更新或提交索引。

- 默认情况下，Solr 会将 document 的每个域的域值进行索引，当对一些大文档进行索引的时候，因为创建索引过程中 Solr 需要将 document 缓存在内存中，如果域的域值很大，内存占用就很大，可能触发更频繁的 GC，GC 可能会导致暂停索引创建过程，对一些大文本域使用的域类型配置 LimitTokenCountFilterFactory 来限制实际索引的文本长度，从而减少索引过程中内存占用。
- 在创建索引的时候，需要对文本进行分词处理时，建议配置停止词来剔除掉无用的噪音词，从而减少索引体积，同时还可以避免噪音词影响最终的检索结果。
- 禁用 CompoundFile（复合）文件：开始复合文件虽然可以减少段文件个数，但是它会使得你的索引创建时间增加 7%～33%，具体配置如下：

```
<useCompoundFile>false</useCompoundFile>
<mergePolicy class="org.apache.lucene.index.TieredMergePolicy">
<float name=" noCFSRatio" >0.0</float>
</mergePolicy>
```

- 如果索引速度经过一系列优化还是比较慢，建议可以使用 MapReduce 框架，利用多台机器的资源并行创建 Solr 索引，从而加快索引速度。

6.1.3 索引合并性能调优

- 降低索引合并频率：索引合并之后能加快 Solr 查询性能，但是索引合并是一个执行开销很大的操作，因此你需要在保证查询性能的前提下，尽量的降低索引合并的频率。
- 加大 ramBufferSizeMB 和 maxBufferedDocs 参数值，并且尽量降低显式提交的频率：索引提交除了用户显式的执行 commit 操作之外，ramBufferSizeMB 或者 maxBufferedDocs 参数达到限定的阈值之后也会自动触发索引提交。因此，为了降低索引合并的频率，应该加大 ramBufferSizeMB 和 maxBufferedDocs 参数值，并且尽量降低显式提交的频率，比如采用批量 commit，或者直接在 solrconfig.xml 中配置自动提交并控制自动提交的频率，避免显式提交。
- 增大 mergeFactor 参数值：加大 mergeFactor 参数值可以加快索引创建速度，降低索引合并频率，但是同时它也会降低你的 Solr 查询响应速度。

6.1.4 查询性能优化

- 如果一个查询需要在三个域上进行，此时可以用 copyField 将三个域合并成为一个域，在合并之后的域上进行查询。因为在单个域上进行查询比在 N 个域上进行查询效率要高。但是使用 copyField 之后，无法为每个单独的域进行加权。
- 应该优先让那些能够过滤掉大部分索引文档的 FilterQuery 先执行。
- 在对数字域进行范围查询的时候，可以调整 precisionStep 来对 rangeQuery 进行优化。precisionStep 默认值是 4，这个值越大，分解出来的前缀索引就越多，数字范围查询越快，但是会增大索引体积。

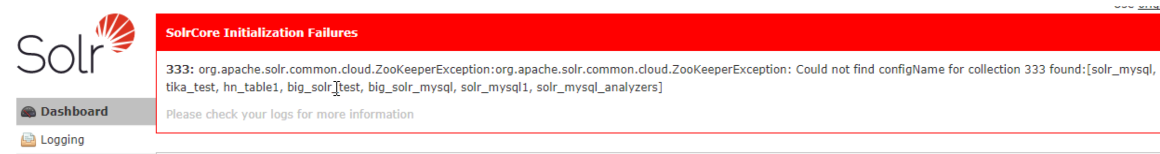
6.2 通用类

6.2.1 创建 core 失败的情况下，UI 界面会出现错误提示，这个信息怎么删除？

问题现象：

创建 core 时，在某些情况下会创建失败，此时 UI 界面会出现错误提示信息，且该信息不能在界面上手动删除。

图6-1 创建 core 错误提示



原因分析：

Solr 源码的实现机制导致操作错误后会在界面显示这些报错信息，但是并不能手动删除。

解决方法：

此时，通过重启 Solr 组件，可以在界面消除错误提示信息。

6.2.2 Solr 使用 SQL 语句精确查询中文字段，出现中文转换编码问题，怎么解决?

问题现象：

使用 SQL 语句精确查询中文字段，示例语句：

http://100.101.1.61:8983/solr/hn_table2/sql?stmt=select * from hn_table2 where c13='非疫区打算清道过时' limit 10&indent=true，出现以下错误提示：

图6-2 精确查询中文字段错误提示

HTTP ERROR 500

Problem accessing /solr/hn_table2/sql. Reason:

```
Server Error
```

Caused by:

```
java.lang.AssertionError: Internal error: while converting `hn_table2`.`c13` = '?????????'
	at org.apache.calcite.util.Util.newInternal(Util.java:795)
	at org.apache.calcite.sql2rel.ReflectiveConvertletTable$1.convertCall(ReflectiveConvertletTable.java:96)
	at org.apache.calcite.sql2rel.SqlNodeToRexConverterImpl.convertCall(SqlNodeToRexConverterImpl.java:59)
	at org.apache.calcite.sql2rel.SqlToRelConverter$Blackboard.visit(SqlToRelConverter.java:4278)
	at org.apache.calcite.sql2rel.SqlToRelConverter$Blackboard.visit(SqlToRelConverter.java:3657)
```

原因分析：

Solr 源码的实现机制不支持 SQL 中文查询。

解决方法：

Solr 使用 SQL 语句精确查询时，不支持中文。使用查询功能时，请参见 4.1.3 查询 Search 章节中支持的查询命令。

6.3 运维类问题

6.3.1 日志查看方法

查看 Solr 的日志信息，有两种方式：

(1) 通过日志管理查看。

(2) 登录 Solr 集群后台，直接查看日志信息。

1. 通过日志管理查看

组件日志页面展示指定集群中产生的所有组件日志信息，支持通过日志级别、组件名、主机名、时间范围等筛选查询日志。

(1) 在集群管理的左侧导航树中选择[日志管理/组件日志]，进入组件日志页面。

(2) 组件日志以图表和日志列表形式展示系统中的组件日志。

- 图表：通过分析日志列表中日志的级别和数量等信息，以柱状图的形式展示不同组件产生的不同级别的日志数量。
- 日志列表：展示系统内指定集群的组件日志列表。展示信息包括日志产生的时间、日志级别、产生日志的组件和主机、日志内容。单击日志内容，可查看日志的详细信息。

图6-3 组件日志页面

2. 登录 Solr 集群后台，直接查看日志

Solr 集群日志默认路径在/var/de_log/solr。日志路径信息，可以在大数据平台 Solr 配置页面中，通过搜索 solr_config_log_dir 来查看。

3. 日志查看注意点

先通过大数据平台管理页面找出有问题的 Solr 节点，查看对应节点日志，在日志中寻找报错信息（搜索关键字：Exception、ERROR），根据报错信息分析出错原因。

对于现场无法定位问题，需要将问题节点日志发给技术服务人员进行问题定位解决。

6.3.2 组件停止

如果在组件状态栏看到状态显示“已停止”，表明 Solr 组件已经停止运行，如图 6-4 所示。

图6-4 组件停止

造成组件停止的原因可能有以下几个：

- 端口号被占用

  在 linux shell 终端输入 netstat -anp | grep 8983，查看端口是否被别的进程占用，如果被占用杀掉该进程的进程号，然后重启组件。

- 主机故障

  组件所在的主机发生故障，在主机状态栏查看 Solr 组件所在的主机是否正常。

图6-5 主机列表

主机名	状态	主机IP	磁盘使用率	CPU使用率	内存使用率
test02.hde.com	在线	10.121.6...	1.88% 9.25G/490.88G	3.82%	49.88% 7.73G/15.49G
test01.hde.com	在线	10.121.6...	1.89% 9.28G/490.88G	5.99%	81.86% 12.68G/15.49G
test00.hde.com	在线	10.121.6...	3.08% 15.1G/490.88G	5.63%	81.86% 12.68G/15.49G

- 其他原因

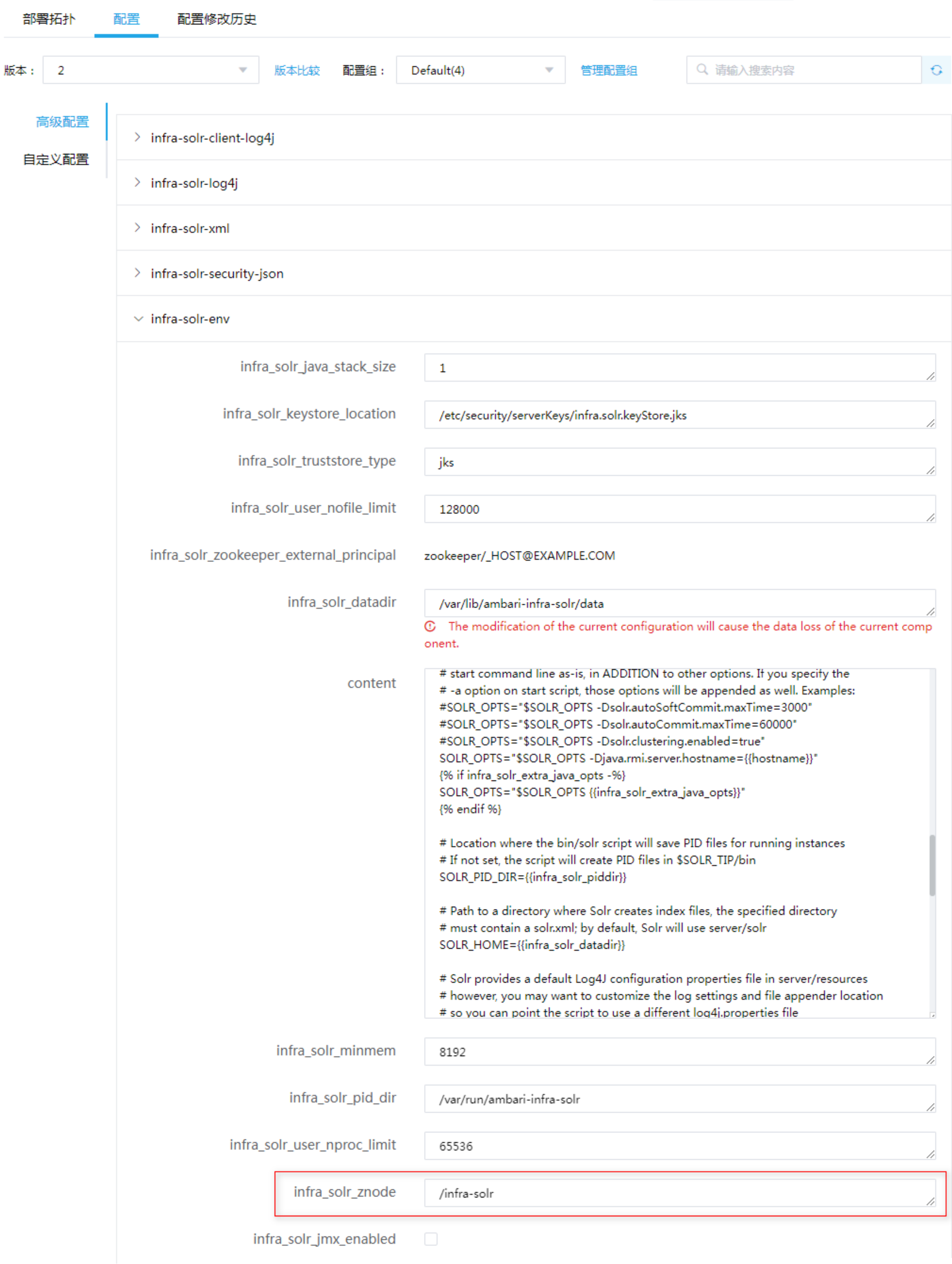
  查看/var/log/solr/solr.log 日志的报错信息，根据日志进行定位。

6.3.3 修改系统组件 Infra-solr 的参数 infra_solr_znode，出现日志异常，怎么解决？

问题现象：

在集群列表的[组件]页签下，选择系统组件 NFRA_SOLR，单击组件名称进入 NFRA_SOLR 详情页面。此时若修改[配置]页签下高级配置中 infra_solr_znode 配置项对应的值，如图 6-6 所示，保存配置后，会出现日志管理异常。

图6-6 高级配置



原因分析：

infra_solr_znode 配置项的值对应 Infra-solr 在 Zookeeper 中的 znode 名称，LOGSEARCH 根据该值在 Zookeeper 中查询相关信息。一旦修改，若 LOGSEARCH 不重启，就无法获取新的值，在 Zookeeper 中无法获取相关信息，从而导致 LOGSEARCH 不可用。

解决方法：

在集群列表的[组件]页签下，选择系统组件 LOGSEARCH，单击<重启>按钮，重启 LOGSEARCH 即可。

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.2 组件架构 1-1
1.3 基本概念 1-2
1.4 应用场景 1-3
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 数据目录检查 2-1
2.1.2 查看组件的日志存储路径 2-1
2.2 运行状态监控 2-2
2.2.1 查看组件详情 2-2
2.2.2 组件检查 2-3
2.3 快速使用指导 2-5
2.3.1 未开启权限管理及 Kerberos 认证操作示例 2-5
2.3.2 开启权限管理、未开启 Kerberos 认证操作示例 2-8
2.3.3 开启权限管理及 Kerberos 认证操作示例 2-9
2.4 快速链接 2-11
2.4.1 配置组件快速链接 2-11
2.4.2 Kibana 登录页 2-11
2.4.3 Kibana 常用模块介绍 2-12
3 使用指南 3-1
3.1 新字段类型 3-1
3.2 权限访问控制 3-1
3.2.1 权限说明 3-1
3.2.2 权限使用操作示例 3-2
3.3 Elasticsearch 集群扩容 3-3
3.3.1 使用场景 3-3
3.3.2 扩容前准备 3-4
3.3.3 扩容约束 3-4
3.3.4 扩容影响 3-4
3.3.5 扩容期间禁止的操作 3-4
3.3.6 扩容操作指导 3-5

3.3.7 扩容验证 3-6
3.4 Elasticsearch 集群缩容......3-6
3.4.1 使用场景 3-6
3.4.2 缩容前准备......3-6
3.4.3 缩容约束 3-6
3.4.4 缩容影响 3-6
3.4.5 缩容期间禁止的操作 3-7
3.4.6 缩容操作指导 3-7
3.4.7 缩容验证 3-7
3.5 租户管理......3-8
3.5.1 租户介绍 3-8
3.5.2 新增租户 3-8
3.5.3 租户使用操作示例......3-10
3.6 数据迁移......3-12
3.6.1 sshfs 跨集群迁移 3-12
3.7 Elasticsearch 多实例 3-15
3.7.1 配置单节点多实例......3-15
3.7.2 单节点多实例的操作注意事项......3-17
3.7.3 启停单节点多实例......3-18
3.8 Elasticsearch 插件可视化操作......3-19
3.8.1 查看 Elasticsearch 集群插件......3-19
3.8.2 安装插件 3-20
3.8.3 删除插件 3-22
3.9 Elasticsearch 快照备份和恢复......3-23
3.9.1 构建 Elasticsearch 集群的快照仓库......3-23
3.9.2 Elasticsearch 集群快照备份......3-27
3.9.3 Elasticsearch 集群快照恢复......3-29
4 开发指南 4-1
4.1 API 使用 4-1
4.1.1 集群 API......4-1
4.1.2 索引 API......4-3
4.1.3 文档 API......4-7
4.1.4 查询 API......4-17
4.1.5 排序 API......4-21
5 最佳实践 5-1
5.1 Spark 读写 Elasticsearch 数据 5-1

5.1.1 依赖包 5-1
5.1.2 完整代码 5-1
6 常见问题解答 6-1
6.1 调优 6-1
6.1.1 优化索引设计 6-1
6.1.2 调优索引性能 6-1
6.1.3 调优搜索性能 6-2
6.2 运维类问题 6-3
6.2.1 日志查看方法 6-3
6.2.2 组件停止 6-4
6.2.3 脑裂问题 6-4

1 组件简介

1.1 组件概述

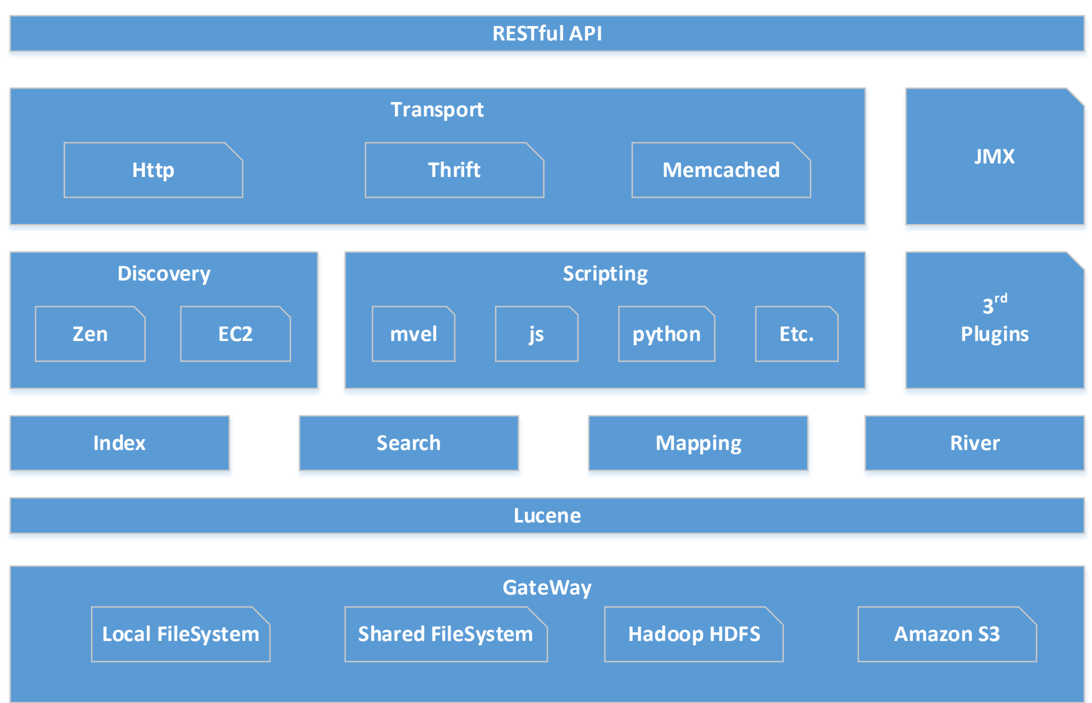
Elasticsearch 是一个基于 Apache Lucene 的搜索和数据分析工具。Elasticsearch 使用 Lucene 作为其核心来实现所有索引和搜索的功能，它通过简单的 RESTful API 来屏蔽 Lucene 的复杂性，从而让全文搜索变得简单。大数据平台中的 Elasticsearch 基于开源 7.10.0 版本。

Elasticsearch 的特点及优势如下：

- 分布式的实时文件存储，每个字段都被索引并可被搜索。
- 分布式的实时分析搜索引擎。
- 可弹性扩展到上百台服务器规模，处理 PB 级别结构化或非结构化数据。
- 支持多种分词插件。

1.2 组件架构

图1-1 Elasticsearch 组件架构



- Gateway 是 Elasticsearch 用来存储索引的文件系统，支持多种存储类型。
- Gateway 的上层是一个分布式的 lucene 框架。
- Lucene 之上是 Elasticsearch 的模块，包括：索引模块、搜索模块、映射解析模块、数据传输模块等。
- Elasticsearch 模块之上是 Discovery、Scripting 和第三方插件。

- Discovery 是 Elasticsearch 的节点发现模块，不同机器上的 Elasticsearch 节点组成集群需要进行消息通信，集群内部需要选举 Master 节点，这些工作都是由 Discovery 模块完成，支持多种发现机制，如 Zen、EC2。
- Scripting 用来支持在查询语句中插入 javascript、Python 等脚本语言，Scripting 模块负责解析这些脚本。
- Elasticsearch 也支持多种第三方插件。

- 再上层是 Elasticsearch 的传输模块和 JMX。
  - 传输模块支持多种传输协议，如 Thrift、Memecached、Http，默认使用 Http。
  - JMX 是 java 的管理框架，用来管理 Elasticsearch 应用。
- 最上层是 Elasticsearch 提供给用户的接口，可以通过 RESTful 接口和 Elasticsearch 集群进行交互。

1.3 基本概念

- cluster

  代表集群，集群包含多个节点，其中有一个 Master 节点，Master 节点通过选举产生，ES 集群中的主从节点是针对集群内部而言的。Elasticsearch 的一个概念就是去中心化，字面上理解就是无中心节点，这是对于集群外部而言的，因为从外部来看，Elasticsearch 集群在逻辑上是个整体，与集群中任何一个节点通信，其效果都是等价的。
- index

  代表索引，用于存储 Elasticsearch 的数据，类似于关系型数据库的 Database。是一个或多个分片分组在一起的逻辑空间。
- shards

  代表索引分片，Elasticsearch 可以把一个完整的索引分成多个分片，并分布到不同的节点上，构成分布式搜索。分片的数量只能在索引创建前指定，并且索引创建后不能更改。
- replicas

  代表索引副本，Elasticsearch 可以设置多个索引的副本。副本的作用，一是提高系统的容错性，当某个节点某个分片损坏或丢失时可以从副本中恢复。二是提高 Elasticsearch 的查询效率，Elasticsearch 会自动对搜索请求进行负载均衡。
- document

  代表文档，Elasticsearch 存储的实体，是可以被索引的基本单位，相当于关系型数据库中的行。
- field

  代表字段，是组成文档的最小单位。相当于关系型数据库中的 column。
- mapping

  代表类型映射，用来约束字段的类型，可以根据数据自动创建。相当于数据库中的 schema。
- recovery

  代表数据恢复或数据重新分布，Elasticsearch 在有节点加入或退出时会根据机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。
- gateway

代表 Elasticsearch 索引快照的存储方式，Elasticsearch 默认是先把索引存放到内存中，当内存满了时再持久化到本地磁盘。gateway 对索引快照进行存储，当这个 Elasticsearch 集群关闭再重启时就会从 gateway 中读取索引备份数据。Elasticsearch 支持多种类型的 gateway，有本地文件系统（默认）、分布式文件系统，如 Hadoop 的 HDFS。

- type

  type 代表文档类型，类似于关系型数据库中的表，用于区分不同的数据。需要注意的是，在 Elasticsearch7.x 版本中，一个索引只能有一个 type 且默认使用_doc 作为索引的 type。Elasticsearch 官方声明将在 Elasticsearch8 版本中彻底移除 type。

1.4 应用场景

1. 日志分析

- 运维分析：对 IT 设备进行运维分析与故障定位、对业务指标分析运营效果。
- 统计分析：20 余种统计分析方法、近 10 种划分维度。
- 实时高效：从入库到能够被检索到，时间差在数秒到数分钟之间。

2. 站内搜索

- 实时检索：站内资料或商品信息更新数秒至数分钟即可被检索。
- 分类统计：检索同时可以将符合条件的商品进行分类统计。
- 高亮显示：提供高亮能力，页面可自定义高亮显示方式。

2 快速入门

2.1 组件安装

注意

- 在 Hadoop 集群或 Elasticsearch 集群中，安装 Elasticsearch 时的注意事项、操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- 在大数据平台当前版本中，支持部署 Elasticsearch 专有实例，即在 Elasticsearch 专有实例上仅能部署 Elasticsearch 进程。

大数据集群中，部署 Elasticsearch 包括以下两种方式：

- 在集群类型为 Hadoop 的大数据集群中安装 Elasticsearch，此时在集群中同时还能部署其他大数据组件。
- 在集群类型为 Elasticsearch 的大数据集群中安装 Elasticsearch，此时在集群中仅能部署 Elasticsearch 组件（开启审计日志时，系统会默认安装 Zookeeper 组件）。

2.1.1 数据目录检查

根据现场磁盘分区方案和挂盘方案的不同，组件安装完成后，必须对各组件的数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表 2-1 所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
Elasticsearch	是（配置项的参数值默认使用全部挂载路径）	path.data	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录 • 请确保 Easticsearch 用户具备数据目录的读写权限

2.1.2 查看组件的日志存储路径

表2-2 组件日志路径说明

组件	日志路径
Elasticsearch	/var/de_log/elasticsearch

2.2 运行状态监控

2.2.1 查看组件详情

进入 Elasticsearch 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、概要、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。
- 概要：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 插件信息：查看 Elasticsearch 当前已安装的所有插件以及插件的详细信息。
- 组件相关操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、查看操作记录等。

图2-1 组件详情

2.2.2 组件检查

执行Elasticsearch组件检查时，主要检测当前Elasticsearch集群中的所有实例是否都在正常运行。集群在使用过程中，根据实际需要，可对 Elasticsearch 执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：

- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Elasticsearch 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Elasticsearch 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

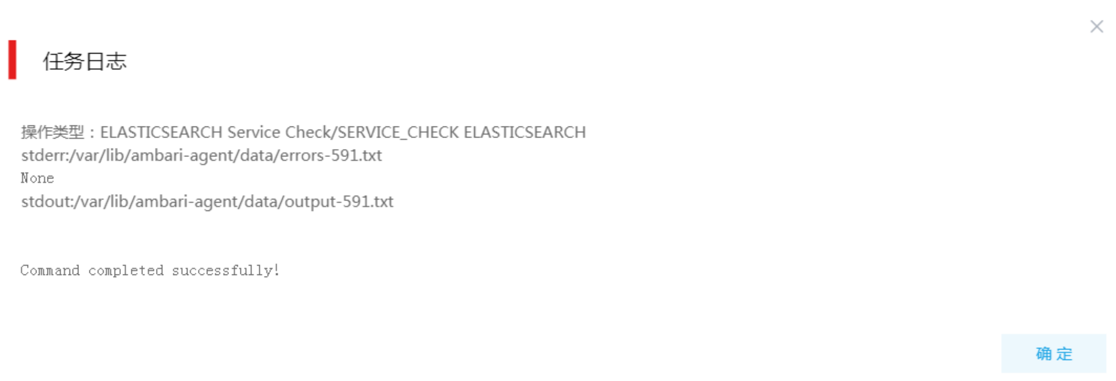
(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态，如图 2-2 所示，表示该组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“ELASTICSEARCH Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情



2.3 快速使用指导

注意

- 根据大数据集群是否开启 Kerberos 认证，用户访问 Elasticsearch 集群时的认证方式不同，详情请参见本章节内容。
- Elasticsearch 组件的操作命令会因为集群是否开启权限管理及 Kerberos 认证而有所不同，权限管理的相关内容可参见3.2 权限访问控制。

Elasticsearch 既可以通过集群用户连接，又可以通过组件超级用户连接。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
  - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为"集群超级用户"。
  - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 Elasticsearch 组件的 elasticsearch 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

用户可根据实际需要配置 Elasticsearch 集群是否开启 Kerberos 及权限管理。Elasticsearch 组件的操作命令会因为集群是否开启权限管理及 Kerberos 认证而有所不同：

- 未开启权限管理及 Kerberos 认证：按照一般方式使用 Elasticsearch 操作命令。
- 开启权限管理，未开启 Kerberos 认证：需在 Elasticsearch 操作命令里加上关键字 "-u username:*"，其中 username 为拥有该操作权限的用户名。
- 开启权限管理及 Kerberos 认证：需要在 Elasticsearch 操作命令里加上关键字 "--negotiate -u"。

2.3.1 未开启权限管理及 Kerberos 认证操作示例

注意

- 非 Kerberos 环境下，不需要用户做身份认证即可直接对 Elasticsearch 执行管理操作。
- 本章节操作需要切换至具有操作 Elasticsearch 组件权限的用户。

Elasticsearch 集群搭建好之后，可以通过调用 curl 命令进行测试，默认情况下，Elasticsearch 的监听端口是 9200，用户可以通过 Elasticsearch 的配置文件修改端口号信息。

1. 获取集群信息

在 Linux 环境中，基于 curl 命令，向 Elasticsearch 集群中的任意节点发送请求，命令如下：

```
curl -X GET http://<Elasticsearch_ip>:9200
```

其中 Elasticsearch_ip 表示安装 Elasticsearch 节点的服务器 IP 地址。

得到的返回结果示例：

```
{
  "name" : "test00.hde.com",
  "cluster_name" : "my_es_cluster",
  "cluster_uuid" : "YUCbB_1VTS6fmvPUfIDnUA",
  "version" : {
    "number" : "7.10.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "22e1767283e61a198cb4db791ea66e3f11ab9910",
    "build_date" : "2019-09-27T08:36:48.569419Z",
    "build_snapshot" : false,
    "lucene_version" : "8.7.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

2. 添加文档

通过 Linux 环境下的 curl 命令访问 Elasticsearch 的 9200 端口进行添加文档。

示例：

```
curl -H "Content-Type:application/json" -X POST
http://<Elasticsearch_ip>:9200/test_index/test_type?pretty -d '{"name": "xiaoming", "age": 15}'
```

其中：

- test_index：是索引名称，如果集群中不存在该索引则会自动创建。
- Elasticsearch_ip：表示安装 Elasticsearch 节点的服务器 IP 地址。
- 每个文档都有自己的 id 和 type，在返回结果中会显示，如果在创建时没有指定，则系统会为其随机生成一个。本例中指定了 type 为 test_type，没有指定 ID。

创建成功响应示例：

```
{
  "_index" : "test_index",
  "_type" : "test_type",
  "_id" : "LbXSC3IBzPEntZG0Oj8i",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

3. 更新文档

若 Elasticsearch 中存在文档，可以使用如下命令更新文档。

```
curl -H "Content-Type:application/json" -X PUT
http://<Elasticsearch_ip>:9200/test_index/test_type/<doc_id> -d '{"<key>":"<value>" }'
```

其中：

- Elasticsearch_ip：表示安装 Elasticsearch 节点的服务器 IP 地址。
- doc_id：表示文档 ID。
- key：表示文档的字段名。
- value：文档字段对应的内容。

示例：

```
curl -H "Content-Type:application/json" -X PUT
http://<Elasticsearch_ip>:9200/test_index/test_type/LbXSC3IBzPEntZG0Oj8i?pretty -d '{"name":
"xiaoming", "age": 17}'
```

更新成功响应示例如下：

```
{
  "_index" : "test_index",
  "_type" : "test_type",
  "_id" : "LbXSC3IBzPEntZG0Oj8i",
  "_version" : 2,
  "result" : "updated",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 1,
  "_primary_term" : 1
}
```

4. 检索文档

可以通过 GET 命令对文档进行检索查询。

示例：

```
curl -H "Content-Type:application/json"  -X GET
http://<Elasticsearch_ip>:9200/test_index/test_type/AWiyyYenTVhrxfir_T01?pretty
```

检索结果如下：

```
{
 "_index" : "test_index",
 "_type" : "test_type",
 "_id" : "AWiyyYenTVhrxfir_T01",
 "_version" : 2,
 "_seq_no" : 1,
 "_primary_term" : 1,
```

```
 "found" : true,
 "_source" : {
  "name" : "xiaoming",
  "age" : 17
 }
}
```

5. 搜索文档

可以通过 GET 或 POST 命令指定索引、类型或 URL 等参数对文档进行搜索。示例如下：

- 搜索集群中的所有文档

  ```
  curl -H "Content-Type:application/json" -X GET http://<Elasticsearch_ip>:9200/_search?pretty
  ```

- 搜索指定索引下面的所有文档

  ```
  curl -H "Content-Type:application/json" -X GET
  http://<Elasticsearch_ip>:9200/<index_name>/_search
  ```

- 搜索索引中指定类型的所有文档

  ```
  curl -H "Content-Type:application/json" -X GET
  http://<Elasticsearch_ip>:9200/<index_name>/<type_name>/_search
  ```

以上示例中的部分参数解释如下：

- <Elasticsearch_ip>：表示安装 Elasticsearch 节点的服务器 IP 地址。
- index_name：索引名称。
- type_name：类型名称。

6. 删除文档

```
curl  -H "Content-Type:application/json" -XDELETE
http://<Elasticsearch_ip>:9200/<index_name>/<index_type>/<id>
```

其中：

- Elasticsearch_ip：表示安装 Elasticsearch 节点的服务器 IP 地址。
- index_name：索引名称。
- index_type：类型名称。
- id：文档 ID。

7. 删除索引

```
curl -H "Content-Type:application/json" -XDELETE http://<Elasticsearch_ip>:9200/<index_name>
```

其中：

- Elasticsearch_ip：表示安装 Elasticsearch 节点的服务器 IP 地址。
- index_name：索引名称。

2.3.2 开启权限管理、未开启 Kerberos 认证操作示例

操作 Elasticsearch 集群的命令和操作非 Kerberos 环境下 Elasticsearch 的命令基本上一致，不同的就是需在 Elasticsearch 操作命令里加上关键字“-u username:*”，其中 username 为拥有该操作权限的用户名。

开启权限管理、未开启 Kerberos 认证时，通过 GET 命令对文档进行检索查询操作示例如下：

```
curl -H 'Content-Type:application/json' -XGET -u username:* 'localhost:9200/test001/user/1?pretty'
```

2.3.3 开启权限管理及 Kerberos 认证操作示例

注意

- Kerberos 环境下，在要进行 Elasticsearch 操作的节点上进行用户认证，获取票据。关于用户身份认证，详情请参见 2.3.3 1. Kerberos 环境下用户身份认证。
- 本章节操作需要切换至具有操作 Elasticsearch 文件系统权限的用户。
- Elasticsearch 组件超级用户名默认是 elasticsearch，超级用户拥有 Elasticsearch 集群的所有权限。
- Elasticsearch 组件超级用户由集群自动创建，用户不能自己创建名字与超级用户同名的普通用户。

1. Kerberos 环境下用户身份认证

如果大数据集群开启 Kerberos，若想操作 Elasticsearch 集群，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

（一）集群用户身份认证

Kerberos 环境下进行用户身份认证的方式（本章节示例用户为 user1），包括以下两种方式（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**
  a. 登录大数据平台，在集群管理下选择[集群权限/用户管理]，进入用户管理页面，选择对应用户，单击下载认证文件。
  b. 将用户的认证文件（即 keytab 配置包）解压后的文件上传到 user1 用户要访问节点的 /etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1 用户，命令如下：
  ```
  chown user1 /etc/security/keytabs/user1.keytab
  ```
  c. 使用 klist 命令查看 user1.keytab 的 principal 名称，命令如下：
  ```
  klist -k user1.keytab
  ```
  如图 2-4 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-4 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

d. 切换至用户 user1，并执行身份验证的命令。

kinit -kt user1.keytab user1@ TENANTC.COM

其中：user1.keytab 为用户的 keytab 文件

e. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 切换至用户 user1，然后输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-5 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

（二）组件超级用户身份认证

可以通过组件超级用户访问，比如 elasticsearch 用户。在开启 Kerberos 的大数据集群中进行组件超级用户（以 elasticsearch 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 elasticsearch 用户的认证文件 elasticsearch.service.keytab。

(2) 使用 ***klist*** 命令查看 elasticsearch.service.keytab 的 principle 名称，如图 2-6所示。

图2-6 认证文件的 principal 名称

```
[root@wxd003 ~]# klist -k /etc/security/keytabs/elasticsearch.service.keytab
Keytab name: FILE:/etc/security/keytabs/elasticsearch.service.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 elasticsearch/wxd003.hde.com@TEST123.COM
   2 elasticsearch/wxd003.hde.com@TEST123.COM
   2 elasticsearch/wxd003.hde.com@TEST123.COM
   2 elasticsearch/wxd003.hde.com@TEST123.COM
   2 elasticsearch/wxd003.hde.com@TEST123.COM
```

(3) 在集群节点内，切换到 elasticsearch 用户，使用 ***kinit*** 命令进行认证。

su elasticsearch

kinit -kt <elasticsearch 用户的认证文件> <认证文件的 principal 名称>

(4) 输入 ***klist*** 命令查看认证结果。

2. Elasticsearch 操作说明

当用户身份认证成功后，操作 Elasticsearch 集群的命令和操作非 Kerberos 环境下 Elasticsearch 的命令基本上一致，不同的就是需要每次都要带上“**--negotiate -u**”参数。

开启权限管理及 Kerberos 认证时，通过 GET 命令对文档进行检索查询操作示例如下：

curl -H 'Content-Type:application/json' -XGET --negotiate -u : 'localhost:9200/test001/user/1?pretty'

2.4 快速链接

Kibana 是一款开源的数据分析和可视化平台，它是 Elastic Stack 成员之一，设计用于和 Elasticsearch 协同工作。用户可以使用 Kibana 对 Elasticsearch 索引中的数据进行搜索、查看、交互操作。

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

2.4.2 Kibana 登录页

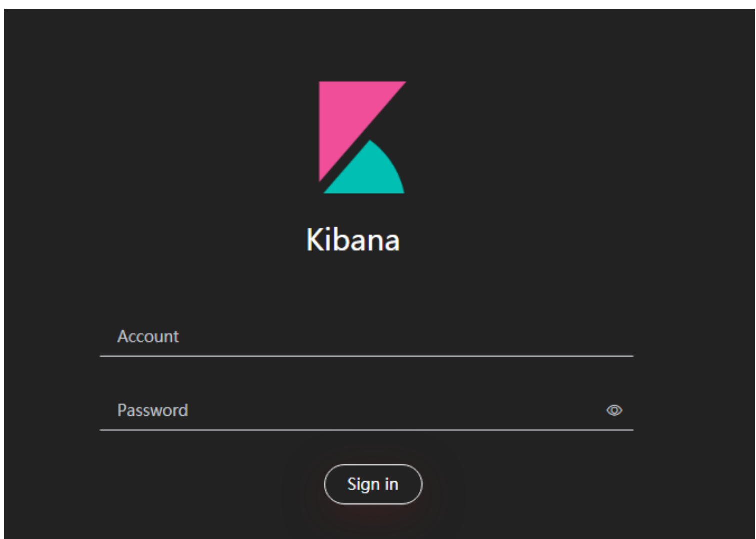
在Elasticsearch组件详情页右上角“快速链接”的下拉框中，可以获取访问Kibana的入口，如图2-7所示。

图2-7 快速链接

通过Elasticsearch的快速链接可进入Kibana的登录页面，如图2-8所示，此时：

- 若集群 Elasticsearch 服务开启了权限管理，对于集群超级用户，可直接访问 Kibana；对于集群普通用户，需要拥有所有索引的“all”权限，才可访问。
- Elasticsearch 服务未开启权限管理，集群普通用户即可直接访问。

图2-8 访问 Kibana 的入口

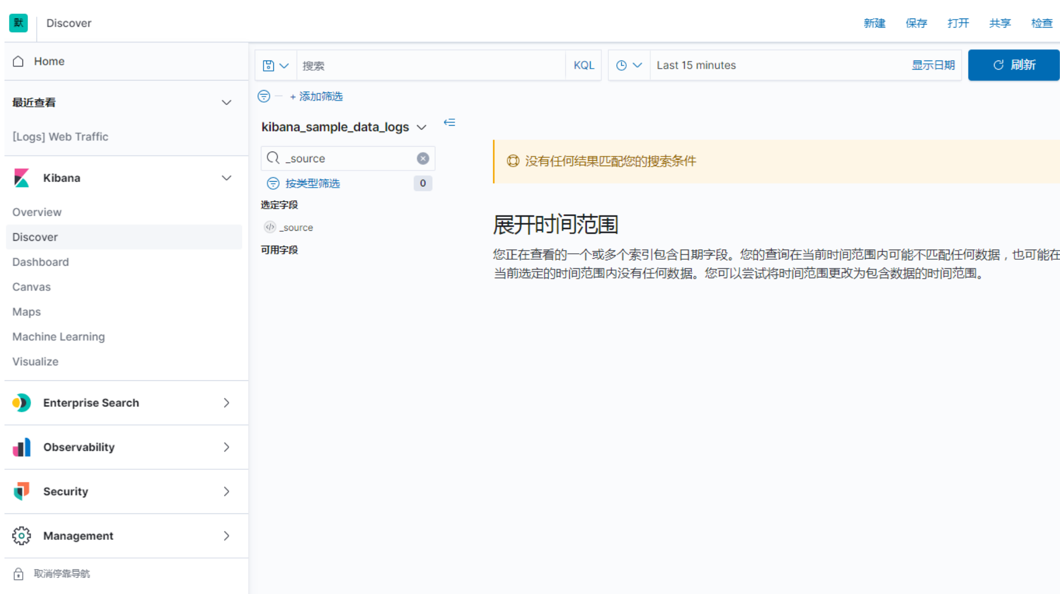


2.4.3 Kibana 常用模块介绍

1. 数据探索 Discover 页面

数据探索页面提供交互式地探索数据功能。用户可以访问与选定索引模式匹配的每个索引中的每个文档，提交搜索请求、过滤搜索结果，查看文档数据等。

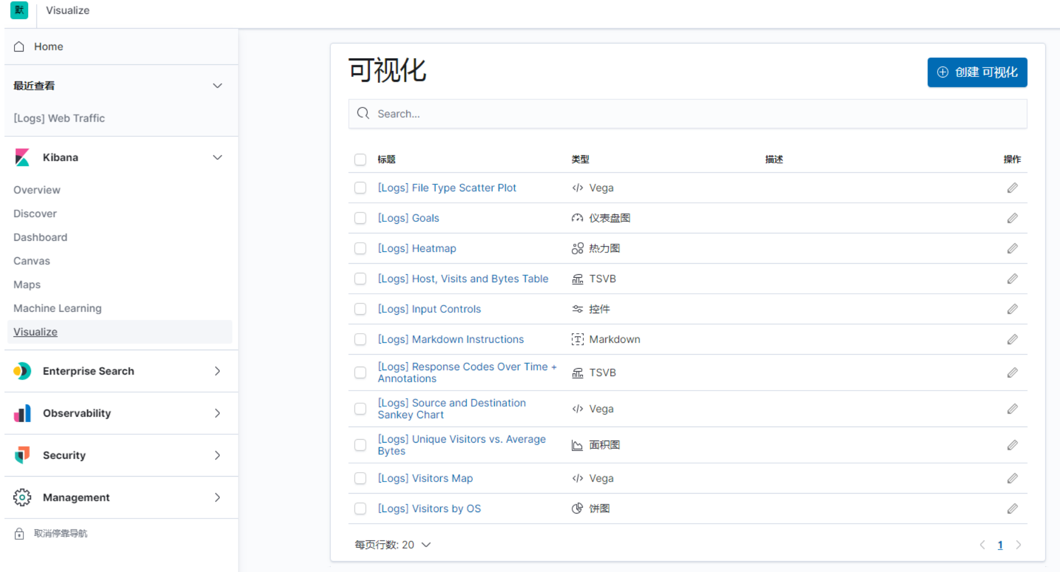
图2-9 数据探索 Discover 页面



2. 可视化页面

可视化功能可以为 Elasticsearch 数据创建可视化控件，可以创建仪表板将可视化控件整合到一起展示。Kibana 可视化控件基于 Elasticsearch 的查询，利用一系列的 Elasticsearch 查询聚合功能来提取和处理数据，还可以通过创建图表来呈现关系的数据分布和趋势。

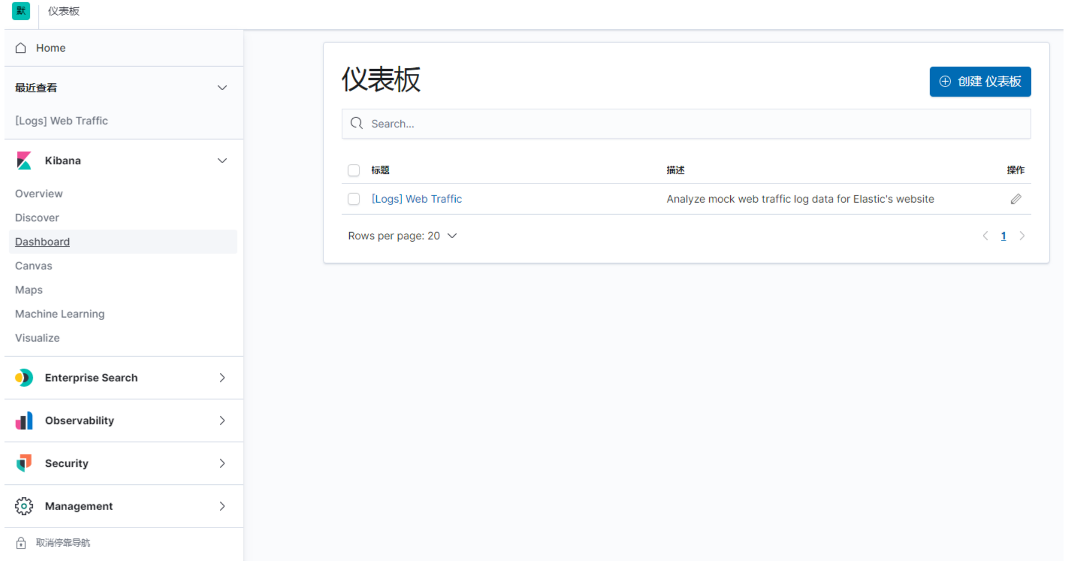
图2-10 可视化页面



3. 仪表板页面

仪表板用于展示保存的可视化结果集合。在编辑模式下，可以根据需要安排和调整可视化结果集，并保存仪表板，以便重新加载和共享。

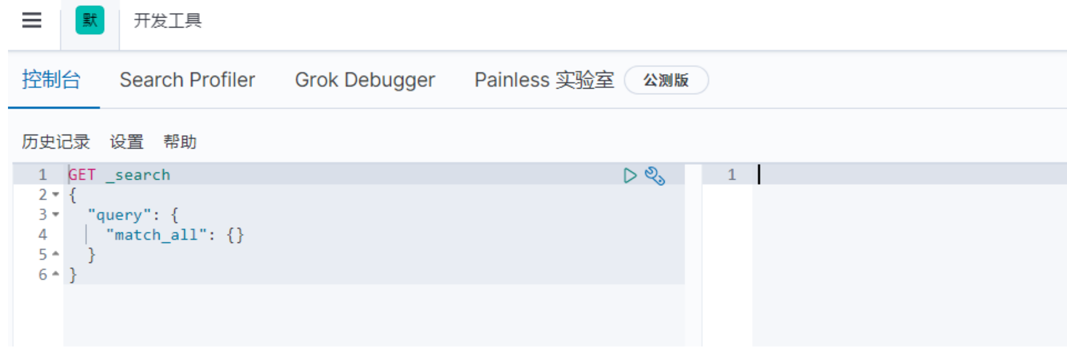
图2-11 仪表板页面



4. 开发工具页面

开发工具页用来和 Elastsicsearch 的 REST API 交互。界面有两个主要部分：编辑框和结果展示框。其中编辑框用来编写提交给 Elasticsearch 的请求；结果展示框用来展示请求结果的响应。

图2-12 开发工具页面



5. 堆栈监测页面

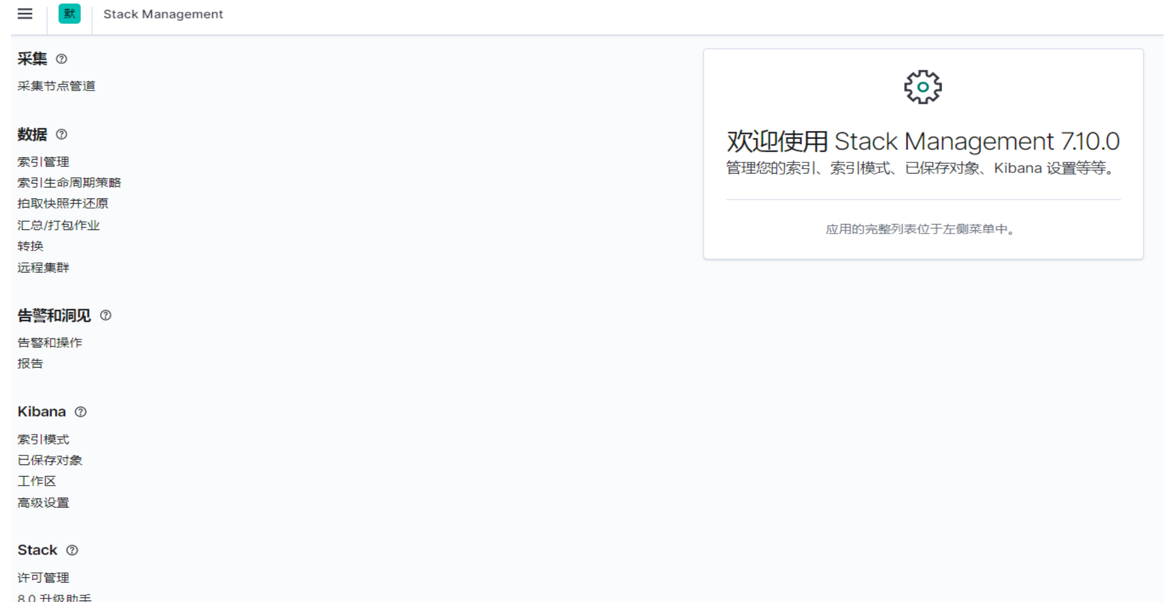
堆栈监测包括Elasticsearch监控和Kibana监控。其中Elasticsearch监控包括集群状态、概览信息、节点信息以及索引信息等；Kibana监控包括Kibana服务概览及实例信息等。

图2-13 堆栈监测页面

6. 管理页面

管理页面功能包括Elasticsearch索引管理、索引生命周期策略设置，快照存储库管理等，以及Kibana索引模式配置、高级配置等。

图2-14 管理页面



3 使用指南

3.1 新字段类型

Elasticsearch5.X之后的字段类型不再支持string，由text或keyword取代。如果仍使用string，会给出警告。

- 当一个字段需要被全文搜索，比如Email内容、产品描述，应该使用text类型。设置text类型以后，字段内容会被分析，在生成倒排索引以前，字符串会被分析器分成一个一个词项。
- text类型的字段不用于排序，较少用于聚合场景（termsAggregation除外）。
- keyword类型不会进行分词，适用于索引结构化的字段，比如Email地址、主机名、状态码和标签。如果字段需要进行过滤（比如查找已发布博客中status属性为published的文章）、排序、聚合等操作可以使用该类型。

3.2 权限访问控制

注意

集群新建用户的权限会因为集群是否开启权限管理功能而有所不同：

- 未开启权限管理时，用户可对索引进行创建、修改、读、写等操作。
- 开启权限管理后，组件权限需通过[集群权限/角色管理]中的角色分配给用户，用户通过绑定角色进行赋权后，才能对组件执行操作。

权限管理是安全管理的重要组成部分，在开启权限的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

3.2.1 权限说明

在开启权限管理的集群中，支持对索引配置权限（支持使用通配符*模糊适配），权限包括：read、index、create、delete、create_index、delete_index、view_index_metadata、monitor、manage、all。注意：Elasticsearch开启权限管理与Kerberos认证后操作命令的使用会发生变化，具体内容参见2.3 快速使用指导。

Elasticsearch操作所需权限对应关系如表3-1所示。

表3-1 Elasticsearch权限说明

组件	权限类型	对应的组件常用操作
Elasticsearch	read	查询索引中数据

组件	权限类型	对应的组件常用操作
	index	在索引中插入数据
	create	在索引中插入数据，等同于index权限
	delete	删除索引中数据
	creat_index	创建索引。创建Elasticsearch索引的用户不具有索引的所有权限，操作对应索引时也需要被授予相应的权限
	delete_index	删除索引
	view_index_metadata	查看索引元数据，当index为“*”时生效
	monitor	监控索引信息，如（_stats、_segments等）
	manage	管理索引信息，（如_refresh、_cache/clear等）
	all	all权限默认包含其它所有权限，当index为“*”时，默认拥有集群所有权限

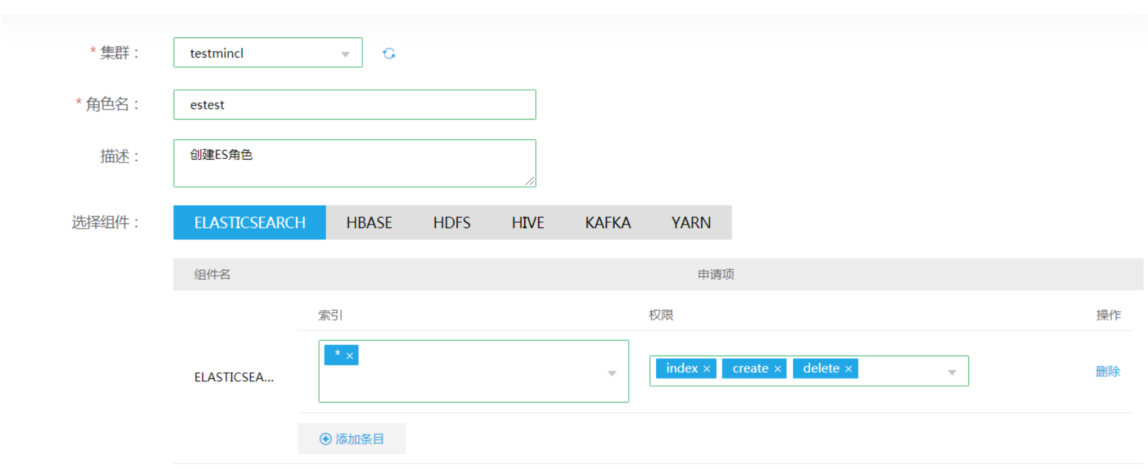
3.2.2 权限使用操作示例

当集群开启了权限管理，操作 Elasticsearch 集群之前，需要创建一个用户，并赋予其操作 Elasticsearch 集群的角色，然后再使用该用户去操作 Elasticsearch。

具体步骤如下：

(1) 在[集群权限/角色管理]页面，创建角色，并为角色授权。

图3-1 创建角色



(2) 在[集群权限/用户管理]页面，创建用户。返回用户管理页面，单击用户对应的<修改用户授权>按钮，为用户添加新角色。

图3-2 对用户授予角色

(3) 操作 Elasticsearch

开启权限管理后 Elasticsearch 集群的操作命令和未开启权限管理的 Elasticsearch 集群的基本一致，不同的就是需要加上“-u username:*”参数，其中 username 就是上一步创建的用户名，*表示密码任意（注：此处对用户的密码无校验，如需校验密码可通过开启 Kerberos 功能配合使用）。

```
curl  -H 'Content-Type:application/json'  -XGET -u <username>:*
'localhost:9200/test001/user/1?pretty'
```

3.3 Elasticsearch集群扩容

3.3.1 使用场景

随着业务量的增长，Elasticsearch 服务任务量过大，服务的存储资源、计算能力无法满足业务需求时，需要对 Elasticsearch 服务进行扩容。下面举例说明如何对 Elasticsearch 集群进行横向扩展。

现在有一个大数据集群，集群包含 5 个主机，目前已经在 node1、node2、node3 三个主机上安装有 Elasticsearch 节点。集群中有一个 user 索引包含 5 个分片和 1 个副本，也就是说该索引中的 5 个分片分布在 Elasticsearch 集群中的 3 个节点上，为了减轻 Elasticsearch 各节点的压力，可以对 Elasticsearch 集群进行扩容，即增加新的节点。

Elasticsearch 集群扩容的方式有两类：

(1) 新增节点：在新的节点上安装 Elasticsearch 组件，并将其加入到 Elasticsearch 集群中。

(2) 多实例：在已有的节点开启多实例。本章节内容只针对新增节点进行介绍，关于多实例的具体介绍请参见 3.7 Elasticsearch 多实例章节内容。

以上两种扩容类型考虑的场景是不同的。可以从以下方面进行考虑：

(1) Elasticsearch 集群中的每个实例的堆内存使用率都很高，但是所在机器的内存及 CPU 使用率在正常范围内，此时可以考虑使用多实例进行扩容。

(2) Elasticsearch 集群中的每个实例的堆内存使用率都很高，同时所在机器的内存及 CPU 使用率也很高，此时可以考虑通过新增节点进行扩容。

3.3.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在集群节点上新增安装 Elasticsearch 进程：

- 如果集群中有节点没有安装 Elasticsearch 进程，直接在集群节点中安装 Elasticsearch 进程。
- 如果集群中所有节点均已安装 Elasticsearch 进程，进行 Elasticsearch 扩容前则需要先添加主机。
- 开启多实例。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Elasticsearch 组件的状态是否正常。

(2) 进入 Elasticsearch 组件详情页，查看 Elasticsearch 的部署拓扑，确保集群中每个服务的状态正常，Elasticsearch 处于“已启动”状态。

3.3.3 扩容约束

- 扩容操作一旦开始，不支持中止。
- 建议一台服务器上 Elasticsearch 服务的总堆内存大小不超过当前机器内存的一半，另外一半主要用于文件系统缓存。
- 新增节点时，请提前挂载好磁盘，并和原有集群节点保持一致，否则可能会导致 Elasticsearch 服务因找不到路径而启动失败。建议集群中每个 Elasticsearch 实例挂载的磁盘大小保持一致，实例之间磁盘大小相差太大会降低 Elasticsearch 的整体性能。
- 各 Elasticsearch 实例磁盘大小保持一致，防止实例磁盘空间使用率超过 90%后触发节点之间的分片频繁迁移。

3.3.4 扩容影响

- Elasticsearch 的数据存储能力得到增强。
- Elasticsearch 每个节点的读写压力会得到缓解。
- Elasticsearch 扩容期间数据会重新分布，会消耗一定的 CPU 和宽带资源，业务性能可能会下降。

3.3.5 扩容期间禁止的操作

扩容期间 Elasticsearch 数据会进行重分布，为避免高 IO 操作导致数据重分布受影响，建议不要进行创建索引操作。

3.3.6 扩容操作指导

注意

若集群中所有节点均已安装 Elasticsearch，进行 Elasticsearch 扩容前则需要先添加主机，然后再进行 Elasticsearch 扩容。如果集群中有扩容所用主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 Elasticsearch 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-3 所示。

a. 选择进程及主机

在选择进程项的列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

b. 部署进程

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-3 添加进程

(3) 查看进程变化

Elasticsearch 扩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 Elasticsearch 安装数量的变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.3.7 扩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Elasticsearch 组件检查，确保 Elasticsearch 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 Elasticsearch 组件部署拓扑里是否已有新增的扩容节点。

(4) 通过 Elasticsearch 快速链接打开 Kibana 进行检查，查看 Elasticsearch 扩容后的整体运行状态。

3.4 Elasticsearch集群缩容

3.4.1 使用场景

当数据量下降或者 Elasticsearch 集群中的节点有冗余时，可以对集群中的节点数进行缩容。其主要操作方式就是停止对应节点上的 Elasticsearch 组件。

当 Elasticsearch 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.4.2 缩容前准备

(1) 登录大数据平台管理 系统，查看 Elasticsearch 组件的状态是否正常。

(2) 进入 Elasticsearch 组件详情页，查看 Elasticsearch 的部署拓扑，确保集群中每个服务的状态正常，Elasticsearch 处于“已启动”状态。

3.4.3 缩容约束

- 缩容操作一旦开始，不支持中止。
- 缩容时，一次性批量删除的 Elasticsearch 节点数量不可超过原集群的一半。
- 如果集群内存在无副本分片的索引时，请在缩容前为其增加副本分片。
- 多实例集群缩容时，强烈建议逐个删除节点，不要进行批量删除。

3.4.4 缩容影响

- Elasticsearch 缩容期间数据会重新分布，会消耗一定的 CPU 和宽带资源，业务性能可能会下降。
- Elasticsearch 缩容后数据重新分布，磁盘相应数据目录的数据量可能会增加，应保证磁盘空间足够使用。
- 当集群中存在没有副本分片的索引时，缩容时可能会造成数据丢失。

3.4.5 缩容期间禁止的操作

缩容期间 Elasticsearch 数据会进行重分布，为避免高 IO 操作导致数据重分布受影响，建议不要进行创建索引操作。

3.4.6 缩容操作指导

说明

Elasticsearch 缩容不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行 Elasticsearch 缩容操作”为例进行说明，在主机详情页面执行删除 Elasticsearch 缩容操作，与其类似不再进行说明。

缩容操作步骤如下：

(1) 在 Elasticsearch 组件详情页面，选择缩容节点。

(2) 在组件详情页面[部署拓扑]页签下，选择已安装 Elasticsearch 进程且需要缩容的主机，然后单击该进程右侧操作中的<停止>按钮，停止 Elasticsearch。

(3) 删除 Elasticsearch

停止 Elasticsearch 后，如图 3-4 所示，在该进程右侧操作中单击<删除>按钮，即可完成 Elasticsearch 缩容。

图3-4 删除进程

(4) 查看进程变化

Elasticsearch 缩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 Elasticsearch 安装数量的变化以及状态。

(5) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.4.7 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Elasticsearch 组件检查，确保 Elasticsearch 组件可正常使用，详情请参考 2.2.2 组件检查。

(3) 查看 Elasticsearch 组件部署拓扑里相关缩容节点已经删除。

(4) 通过 Elasticsearch 快速链接打开 Kibana 进行检查，查看 Elasticsearch 缩容后的整体运行状态。

3.5 租户管理

注意

- 新增租户时，要求系统中必须已成功创建租户模式的 Hadoop 集群或 Elasticsearch 集群。
- 关于租户管理模块功能的配置或使用详情请参见产品在线联机帮助。

多个租户之间共享一套集群，共享网络和集群资源，并且不同租户之间保证资源隔离。

- 新增租户

  普通用户在自己创建的租户集群中申请租户时，无需审批，会直接触发新增租户的操作。普通用户在其他用户创建的租户集群中申请租户时，则需要走流程审批，待审批人审批通过后才能触发新增租户的操作。管理员用户新增租户时，无需审批，会直接触发新增租户的操作。

- 租户管理操作

  普通用户在自己创建的租户集群中执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。普通用户在其他用户创建的租户集群中执行租户续期、资源扩缩容操作时，则需要走流程审批，待审批人审批通过后才能触发相关操作。管理员用户执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。

3.5.1 租户介绍

Elasticsearch 多租户可以实现索引在 Elasticsearch 集群物理主机级别的存储隔离，通过索引模板实现。租户申请的 Elasticsearch 资源对应一个或多个索引模板，一个索引模板对应集群中一个或多个主机资源，用户可根据实际需要申请 Elasticsearch 组件资源。

【说明】

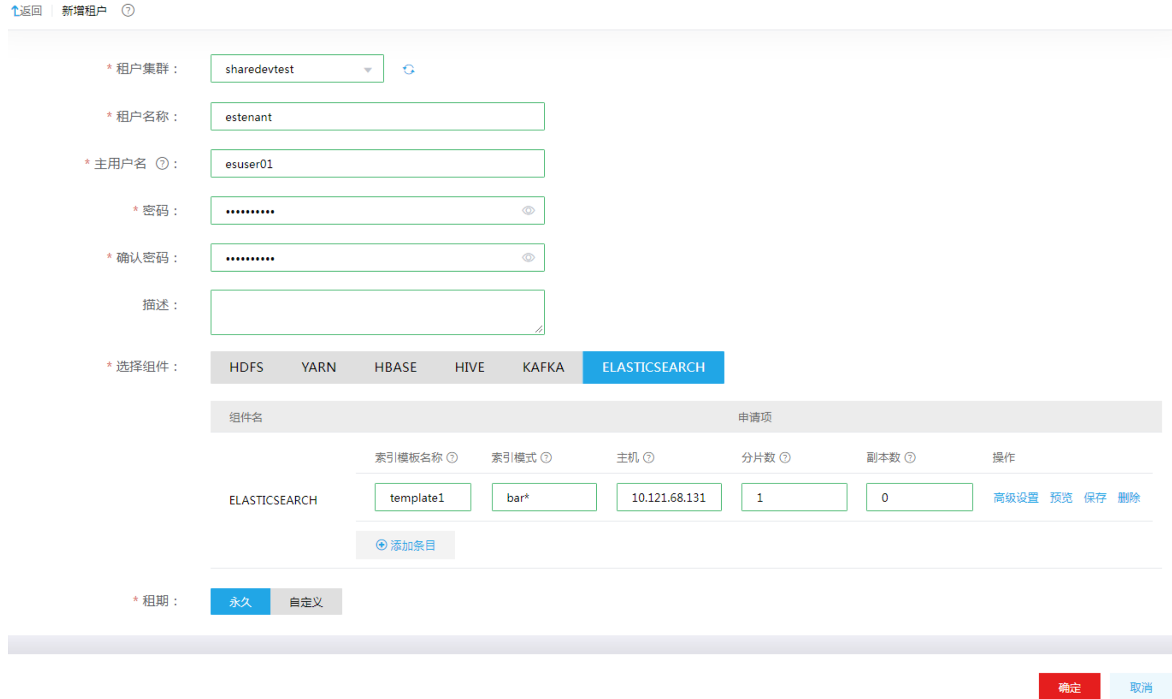
新增 Elasticsearch 租户资源时，可以创建一个或多个索引模板，每个索引模板是一组自定义的基础配置集合（索引模板包含索引模式、对应的实例节点、分片数、副本数以及其他高级配置项，用户在创建索引时可以通过索引名关联索引模板来直接使用这些基础配置）。即创建索引时，如果索引名称与某个索引模式匹配，则会关联到对应的索引模板，然后根据索引模板中的配置确定索引如何创建，比如：索引被分配到哪些 Elasticsearch 主机实例上等，从而实现了不同租户下索引的存储隔离和权限隔离。

3.5.2 新增租户

(1) 在[集群权限/租户管理]页面，单击<新增租户>按钮，跳转至新增租户页面，如图 3-5 所示。租户集群 sharedevtest 新增租户 estenant，主用户 esuser01，并为该租户配置 Elasticsearch

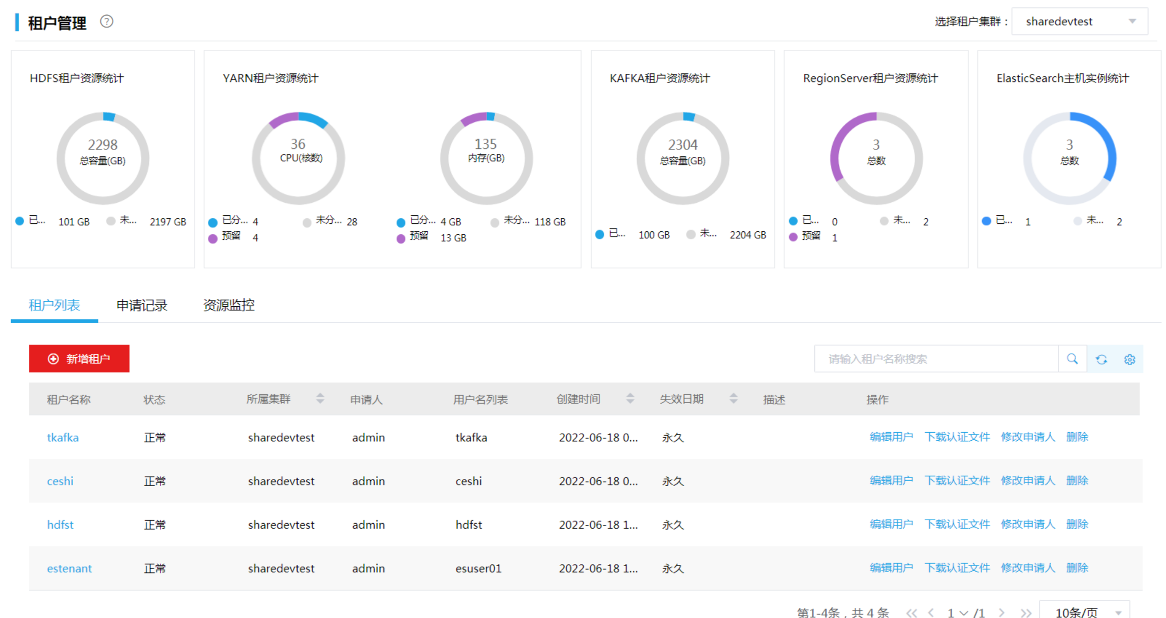
组件资源（索引模板名称 template1，索引模式 bar*，使用的主机实例为 10.121.68.131，分片数为 1，副本数为 0）。

图3-5 新增租户



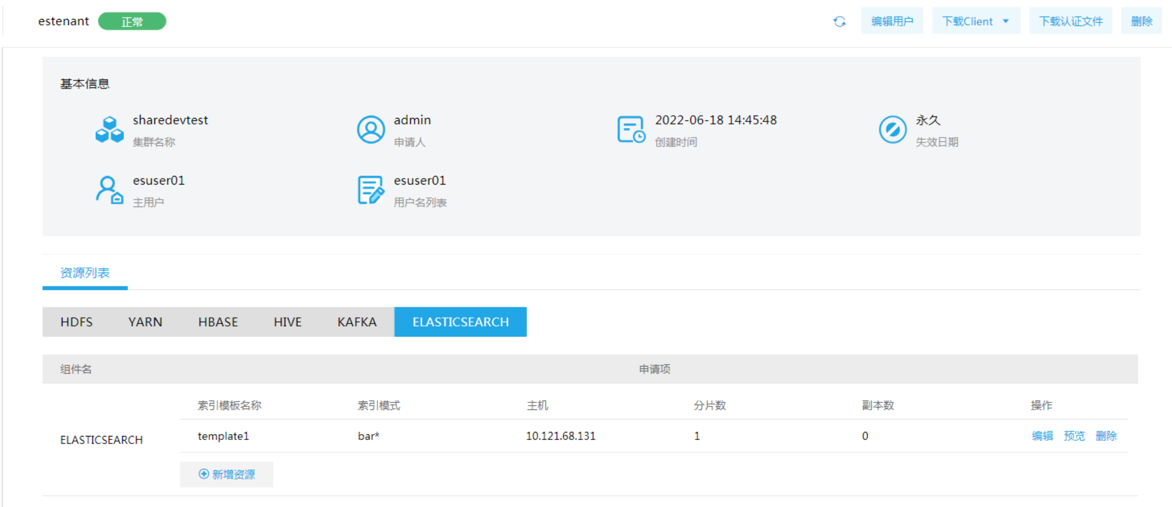
(2) 新增租户成功后，用户可在租户列表查看到已创建的租户，同时可以看到其所属集群、申请人、用户名列表、创建时间、失效时间等相关信息，如图 3-6 所示。

图3-6 查看租户



(3) 单击租户名称，可查看租户详情，如图3-7所示，可以看到对应的Elasticsearch索引模板名称、索引模式，索引模板指定的主机实例、分片数和副本数等信息。用户esuser01拥有租户estenant的所有权限，若资源不够/过多时，可编辑租户调整主机数实现扩容/缩容。

图3-7 查看租户详情



3.5.3 租户使用操作示例

注意

租户集群缺省开启Kerberos认证，在使用租户时需要通过该租户的用户对应的认证文件，对租户的用户进行身份认证。租户的用户认证文件的下载在租户列表页面或租户详情页面执行，关于对租户的用户进行认证的方式与集群中用户的身份认证方式相同，详情请参见2.3.3.1 kerberos环境下的用户身份认证。

租户集群缺省开启 Kerberos 认证，通过租户用户对查看租户或对租户执行管理操作时，均需要提前对租户的用户执行身份认证操作。

(1) 使用集群超级用户登录到租户集群，可以查看到esuser01用户对应的索引模板资源，如图3-8所示。

图3-8 查看索引模板

```
[root@sharedev1 ~]# curl -XGET -H'Content-Type:application/json'  --negotiate -u: 10.121.68.131:9200/_index_template/template1?pretty
{
  "index_templates" : [
    {
      "name" : "template1",
      "index_template" : {
        "index_patterns" : [
          "bar*"
        ],
        "template" : {
          "settings" : {
            "index" : {
              "number_of_shards" : "1",
              "number_of_replicas" : "0",
              "routing" : {
                "allocation" : {
                  "include" : {
                    "_ip" : "10.121.68.131"
                  }
                }
              }
            }
          }
        },
        "composed_of" : [ ]
      }
    }
  ]
}
```

(2) 切换到 esuser01 用户，然后根据索引模式来创建索引 bar1，以应用索引模板的设置，如图 3-9 所示。

图3-9 创建索引

```
-sh-4.2$ curl -XPUT -H'Content-Type:application/json'  --negotiate -u: 10.121.68.131:9200/bar1
{"acknowledged":true,"shards_acknowledged":true,"index":"bar1"}-sh-4.2$
-sh-4.2$ curl -XGET -H'Content-Type:application/json'  --negotiate -u: 10.121.68.131:9200/bar1?pretty
{
  "bar1" : {
    "aliases" : { },
    "mappings" : { },
    "settings" : {
      "index" : {
        "routing" : {
          "allocation" : {
            "include" : {
              "_ip" : "10.121.68.131"
            }
          }
        },
        "number_of_shards" : "1",
        "provided_name" : "bar1",
        "creation_date" : "1655538952468",
        "number_of_replicas" : "0",
        "uuid" : "Y2xgh0t4SgGzLcwkhjb7TA",
        "version" : {
          "created" : "7100099"
        }
      }
    }
  }
}
```

(3) 切换为其他没有 esuser01 权限的用户，无法对索引 bar1 进行操作，如图 3-10 所示。

图3-10 无权限查看索引

```
-sh-4.2$ curl -XGET -H'Content-Type:application/json'  --negotiate -u: 10.121.68.131:9200/bar1?pretty
{
  "error" : {
    "root_cause" : [
      {
        "type" : "status_exception",
        "reason" : "Error: User[ceshi] could not do action[indices:admin/get] on index[bar1], and need privilege[view_index_metadata]"
      }
    ],
    "type" : "status_exception",
    "reason" : "Error: User[ceshi] could not do action[indices:admin/get] on index[bar1], and need privilege[view_index_metadata]"
  },
  "status" : 403
}
```

3.6 数据迁移

3.6.1 sshfs 跨集群迁移

1. 安装 sshfs

在迁移集群中的每个节点上安装 sshfs，也可使用 yum -y install fuse sshfs 命令进行安装，无法用 yum 的话可使用以下步骤安装：

(1) 安装 fuse，命令为：yum -y install fuse。如无法安装请按以下方式操作：
   a. 下载 fuse-2.7.4.tar.gz 到服务器相关目录。
   b. 解压，命令为 tar -zxvf fuse-2.7.4.tar.gz，再运行./configure。
   c. 运行 make && make install 进行安装。
   d. 如在上面两步提示需要安装 glibc 相关组件等信息，请运行 yum -y install gtk2 gtk2-devel gtk2-devel-docs 安装 glibc 相关组件。

(2) 上传 sshfs-2.8.tar.gz 到相关目录中，解压命令为：tar -zxvf sshfs-2.8.tar.gz。

(3) 进入 sshfs 目录，命令为：cd sshfs-2.8。

(4) 编译，命令为：./configure。

(5) 安装，命令为：make && make install。

(6) 验证是否成功，命令为：sshfs -h。

说明

在安装 fuse 与 sshfs 时，由于操作系统不同，通过 yum 安装的依赖可能会有差异，需要根据报错信息自行安装所需依赖。

图3-11 sshfs 验证

```
[elasticsearch@node2 backup]$ sshfs -h
usage: sshfs [user@]host:[dir] mountpoint [options]

general options:
    -o opt,[opt...]        mount options
    -h   --help            print help
    -V   --version         print version

SSHFS options:
    -p PORT                equivalent to '-o port=PORT'
    -C                     equivalent to '-o compression=yes'
    -F ssh_configfile      specifies alternative ssh configuration file
    -1                     equivalent to '-o ssh_protocol=1'
    -o reconnect           reconnect to server
    -o delay_connect       delay connection to server
    -o sshfs_sync          synchronous writes
    -o no_readahead        synchronous reads (no speculative readahead)
    -o sync_readdir        synchronous readdir
    -o sshfs_debug         print some debugging information
    -o cache=BOOL          enable caching {yes,no} (default: yes)
    -o cache_max_size=N    sets the maximum size of the cache (default: 10000)
    -o cache_timeout=N     sets timeout for caches in seconds (default: 20)
    -o cache_X_timeout=N   sets timeout for {stat,dir,link} cache
    -o cache_clean_interval=N
                           sets the interval for automatic cleaning of the
                           cache (default: 60)
    -o cache_min_clean_interval=N
```

2. 建立共享文件夹并挂载

(1) 选定源 Elasticsearch 集群的任意一个节点的目录作为共享目录，并授读写权限，命令如下：

mkdir <共享目录>

chmod 777 <共享目录>

(2) 在源集群每个节点的相同位置创建目录(不同于上一步骤的目录)，并挂载共享目录。

a. 创建目录：

mkdir <Elasticsearch 节点目录>

b. 授权：

chmod 777 <Elasticsearch 节点目录>

c. 挂载共享目录：

sshfs root@<Elasticsearch_ip>:<共享目录> <Elasticsearch 节点目录> -o allow_other

d. 测试 Elasticsearch 用户是否对共享目录具有读写权限：

sudo -u Elasticsearch touch <共享目录>/test

3. 修改 Elasticsearch 集群配置

(1) 进入 Elasticsearch 组件详情页，选择[配置]按钮，进入配置管理页面。

(2) 打开 elasticsearch-config，在 content 字段的文本框末尾添加以下配置：

path.repo: <Elasticsearch 节点目录>

此处的 Elasticsearch 节点目录为 2. 建立共享文件夹并挂载中创建的 Elasticsearch 节点目录。

(3) 修改配置，保存之后重启 Elasticsearch 集群。

(4) 重启后创建仓库：

curl -H 'Content-Type:application/json' -XPUT http://<Elasticsearch_ip>:9200/<Backup_name> -d '{“type”:”fs”,”settings”:{“location”:” <Elasticsearch 节点目录>”}}'

其中 Backup_name 表示备份名称

4. 备份数据

- 备份所有索引：

  curl -H 'Content-Type:application/json' -XPUT http://<Elasticsearch_ip>:9200/_snapshot/<Backup_name>/snapshot_1

  示例：

  curl -H 'Content-Type:application/json' -XPUT http://101.10.36.11:9200/_snapshot/my_backup/snapshot_1

- 备份部分索引：

  curl -H 'Content-Type:application/json' -XPUT http://<Elasticsearch_ip>/_snapshot/<Backup_name>/snapshot_1 -d '{"indices":"<Indices_name>"}'

  示例：

  curl -H 'Content-Type:application/json' -XPUT http://101.10.36.11:9200/_snapshot/my_backup/snapshot_1 -d '{"indices":"panabit-2018*"}'

- 备份单个索引，多个索引备份部分：

  curl -H 'Content-Type:application/json' -XPUT http://<Elasticsearch_ip>:9200/_snapshot/<Backup_name>/snapshot_1 -d '{"indices":" <Indices_name>”}'

  示例：

  curl -H 'Content-Type:application/json' -XPUT http://101.10.36.11:9200/_snapshot/my_backup/snapshot_1 -d '{"indices": "panabit-2018.01.28, panabit-2018.01.29”}'

- 查看备份状态：

  curl -H 'Content-Type:application/json' -XGET http://<Elasticsearch_ip>:9200/_snapshot/<Backup_name>/snapshot_1/_status

  示例：

  curl -H 'Content-Type:application/json' -XGET http://101.10.36.11:9200/_snapshot/my_backup/snapshot_1/_status

- 查看备份信息：

  curl -H 'Content-Type:application/json' –XGET http://<Elasticsearch_ip>:9200/_snapshot/<Backup_name>/snapshot_1

  示例：

  curl -H 'Content-Type:application/json' -XGET http://101.10.36.11:9200/_snapshot/my_backup/snapshot_1

图3-12 备份数据文件

```
[elasticsearch@node2 backup]$ pwd
/mnt/backup
[elasticsearch@node2 backup]$ ll
total 52
-rw-r--r-- 1 root root    28 Apr 10 17:35 index
drwxr-xr-x 1 root root  4096 Apr 10 17:34 indices
-rw-r--r-- 1 root root 37377 Apr 10 17:34 meta-snapshot_1.dat
-rw-r--r-- 1 root root   456 Apr 10 17:35 snap-snapshot_1.dat
-rw-r--r-- 1 root root     0 Apr 10 17:28 test
[elasticsearch@node2 backup]$
```

5. 还原数据

(1) 在数据迁移的目标集群中重复步骤：1. 安装 sshfs、2. 建立共享文件夹并挂载、3. 修改 Elasticsearch 集群配置。

(2) 将源集群的备份内容（<Elasticsearch 节点目录>里面的所有文件），复制到目标的集群仓库目录里。

(3) 使用 Restful API 进行备份的恢复：

```
curl XGET http://<Elasticsearch_ip>:9200/_snapshot/<Backup_name>/snapshot_1/_restore
```

(4) 查看恢复状态：

```
curl -XGET http://<Elasticsearch_ip>:9200/_recovery
```

(5) 查看数据信息：

```
curl -XGET http://<Elasticsearch_ip>:9200/_cat/indices?v
```

3.7 Elasticsearch多实例

3.7.1 配置单节点多实例

注意

安装 Elasticsearch 组件时，缺省单节点只启动一个实例。若安装 Elasticsearch 组件后，需启动单节点的多实例，需要大数据平台管理系统的组件详情页面进行手动配置。

默认情况下，Elasticsearch 单节点只启动一个实例，若需启动单节点的多实例，可通过如下步骤进行配置：

(1) 修改配置项

在集群列表页面选中待启动 Elasticsearch 多实例的集群，修改 Elasticsearch 组件详情页[配置]页签下基础配置中 instances.memory 配置项对应的值（该配置项中的值默认只有一个，单位为 GB，表示启动 ES 实例的堆内存大小），如图 3-13。修改对应值时需以逗号间隔，值的个数表示单节点启动的实例个数，值的大小表示对应实例堆内存大小。例如“2,2,2”表示单节点启动 3 个实例，每个实例对应的堆内存都为 2GB。

图3-13 多实例配置

(2) 保存配置后，重启 Elasticsearch 组件，使配置生效。此时，从后台可以看到多个实例已经启动，如图 3-14。

```
ps -ef | grep org.elasticsearch.bootstrap.Elasticsearch | grep ^elastic
```

图3-14 多实例后台进程

```
[root@node50 ~]# ps -ef|grep org.elasticsearch.bootstrap.Elasticsearch|grep ^ela
stic
elastic+ 54450     1  0 10:53 ?        00:01:23 /usr/local/jdk/bin/java -Xms2g -
Xmx2g -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSI
nitiatingOccupancyOnly -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfil
e.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -Dio.netty.noUn
safe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPe
rThread=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.io.
tmpdir=/tmp/elasticsearch-1719226992446366l -XX:+HeapDumpOnOutOfMemoryError -XX:
HeapDumpPath=data -XX:ErrorFile=logs/hs_err_pid%p.log -XX:+PrintGCDetails -XX:+P
rintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTi
me -Xloggc:logs/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=32 -XX:G
CLogFileSize=64m -Dio.netty.allocator.type=pooled -XX:MaxDirectMemorySize=107374
1824 -Des.path.home=/usr/hdp/current/elasticsearch -Des.path.conf=/usr/hdp/3.0.1
.0-187/elasticsearch/conf -Des.distribution.flavor=default -Des.distribution.typ
e=rpm -Des.bundled_jdk=true -cp /usr/hdp/current/elasticsearch/lib/* org.elastic
search.bootstrap.Elasticsearch -d -p /var/run/elasticsearch/elasticsearch.pid
elastic+ 54632     1  0 10:53 ?        00:01:12 /usr/local/jdk/bin/java -Xms2g -
Xmx2g -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSI
nitiatingOccupancyOnly -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfil
e.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -Dio.netty.noUn
safe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPe
rThread=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.io.
tmpdir=/tmp/elasticsearch-797308314055072549 -XX:+HeapDumpOnOutOfMemoryError -XX
:HeapDumpPath=data -XX:ErrorFile=logs/hs_err_pid%p.log -XX:+PrintGCDetails -XX:+
PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedT
ime -Xloggc:logs/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=32 -XX:
GCLogFileSize=64m -Dio.netty.allocator.type=pooled -XX:MaxDirectMemorySize=10737
41824 -Des.path.home=/usr/hdp/current/elasticsearch_1 -Des.path.conf=/usr/hdp/3.
0.1.0-187/elasticsearch/conf -Des.distribution.flavor=default -Des.distribution.
type=rpm -Des.bundled_jdk=true -cp /usr/hdp/current/elasticsearch_1/lib/* org.el
asticsearch.bootstrap.Elasticsearch -d -p /var/run/elasticsearch/1.pid
elastic+ 54865     1  0 10:53 ?        00:01:13 /usr/local/jdk/bin/java -Xms2g -
Xmx2g -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSI
nitiatingOccupancyOnly -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfil
e.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -Dio.netty.noUn
safe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPe
rThread=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.io.
tmpdir=/tmp/elasticsearch-865357366775064291 -XX:+HeapDumpOnOutOfMemoryError -XX
:HeapDumpPath=data -XX:ErrorFile=logs/hs_err_pid%p.log -XX:+PrintGCDetails -XX:+
PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedT
ime -Xloggc:logs/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=32 -XX:
GCLogFileSize=64m -Dio.netty.allocator.type=pooled -XX:MaxDirectMemorySize=10737
41824 -Des.path.home=/usr/hdp/current/elasticsearch_2 -Des.path.conf=/usr/hdp/3.
0.1.0-187/elasticsearch/conf -Des.distribution.flavor=default -Des.distribution.
type=rpm -Des.bundled_jdk=true -cp /usr/hdp/current/elasticsearch_2/lib/* org.el
asticsearch.bootstrap.Elasticsearch -d -p /var/run/elasticsearch/2.pid
```

3.7.2 单节点多实例的操作注意事项

配置单节点多实例后，为避免 Elasticsearch 状态异常，在操作过程中务必注意以下事项：

- 已经启动 Elasticsearch 单节点多实例，若需动态减少 Elasticsearch 多实例的个数，注意需要一个实例一个实例的减少。这是由于 Elasticsearch 的选举机制，不允许跨度很大的实例个数减少，否则会出现 Elasticsearch 集群不可用的情况。但是，Elasticsearch 实例增加没有这个限制。
- 已经启动 Elasticsearch 单节点多实例，若需要删除 Elasticsearch 组件后再进行重新安装，为避免重新安装的 Elasticsearch 集群不可用，有以下两种处理方案：

- 删除 Elasticsearch 组件后，务必将数据目录（path.data）中的数据进行备份迁移，保证数据目录为空。否则重新安装 Elasticsearch 后（重新安装 Elasticsearch 时缺省单节点只启动一个实例），可能导致历史数据中的实例个数与安装后实例个数不同，出现重新安装的 Elasticsearch 集群不可用的情况。
- 删除 Elasticsearch 组件后再进行重新安装，重新安装 Elasticsearch 时缺省单节点只启动一个实例，此时将重新安装的 Elasticsearch 单节点实例个数修改为大于等于之前的实例个数（即≥卸载前 Elasticsearch 集群的单节点实例个数），则 Elasticsearch 集群即可正常使用。

3.7.3 启停单节点多实例

(1) 如需对某节点上的单个实例执行启/停操作，可进入节点后台执行。

注意

- *在启动实例时，由于 Elasticsearch 不能使用 root 用户启动，需要切换到 elasticsearch 用户进行操作。*
- *在对某节点上的单个实例执行启/停操作前，请确保已经在大数据平台管理系统的组件配置页面对 Elasticsearch 修改的实例配置重启并生效，操作详情请参见 3.7.1 配置单节点多实例，否则对应的实例配置信息可能错误或者找不到。*

通过 elasticsearch 用户登录 Elasticsearch 集群的某节点后台，即可对该节点上的单个实例执行启/停操作。

- 启动单实例，命令如下：

  /usr/hdp/current/elasticsearch/bin/instance <instance_number> start

```
[elasticsearch@sharedev3 root]$
[elasticsearch@sharedev3 root]$ /usr/hdp/current/elasticsearch/bin/instance 2 start
start elasticsearch instance 2 execution complete.
```

- 停止单实例，命令如下：

  /usr/hdp/current/elasticsearch/bin/instance <instance_number> stop

```
[elasticsearch@sharedev3 root]$
[elasticsearch@sharedev3 root]$ /usr/hdp/current/elasticsearch/bin/instance 2 stop
stop elasticsearch instance 2 complete.
```

(2) 若需关闭节点多个实例进程，通过在组件详情页面的[部署拓扑]下停止对应节点上的 Elasticsearch 进程即可，如图 3-15。

图3-15 停止组件

3.8 Elasticsearch插件可视化操作

注意

- 部署 Elasticsearch 集群成功之后，会默认预装一些插件，对于默认预装的插件不支持用户卸载，用户仅可对自行添加的插件进行卸载操作。
- 插件安装之后，用户需要重启 Elasticsearch 服务，从而使插件功能生效。
- 插件可能会导致 Elasticsearch 服务启动失败，用户需要确保自行安装的插件满足 Elasticsearch 要求。

Elasticsearch 插件能够丰富 Elasticsearch 集群的功能，用户在实际生产环境中，可根据自己需要安装对应的 Elasticsearch 插件，本产品提供插件的可视化操作，用户可以在界面完成插件的上传、安装、卸载操作。

3.8.1 查看 Elasticsearch 集群插件

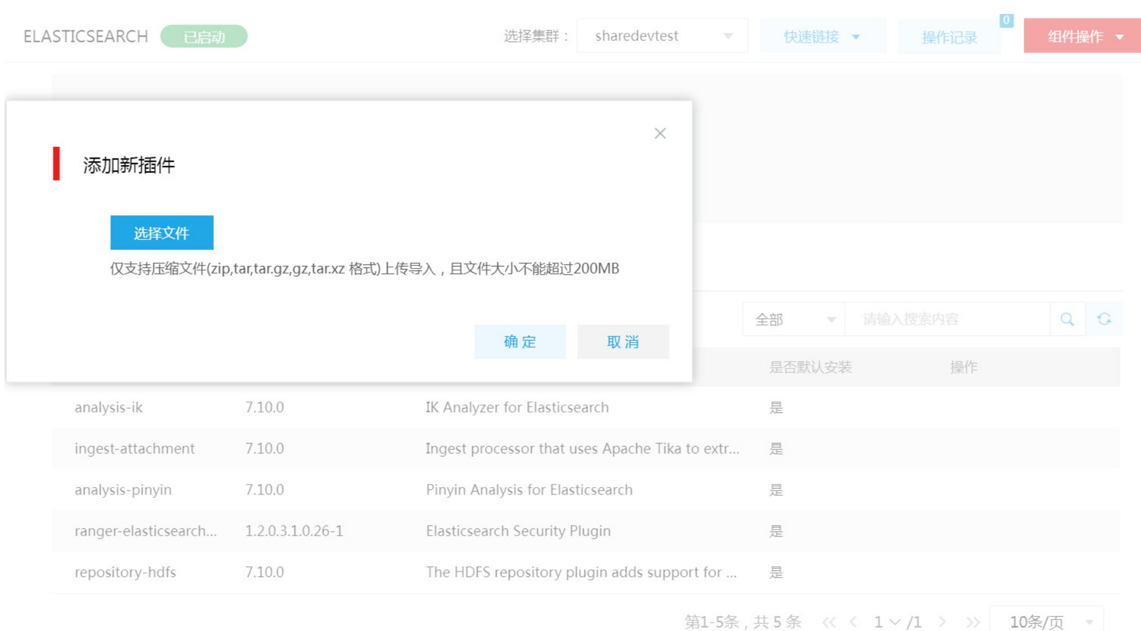
进入 Elasticsearch 组件的详情页，选择[插件信息]页签，可以查看当前 Elasticsearch 集群已经安装的插件信息，此页面会展示已安装的插件名字、插件版本、插件描述以及是否是默认预装等信息，如图 3-16 所示。

图3-16 插件信息

3.8.2 安装插件

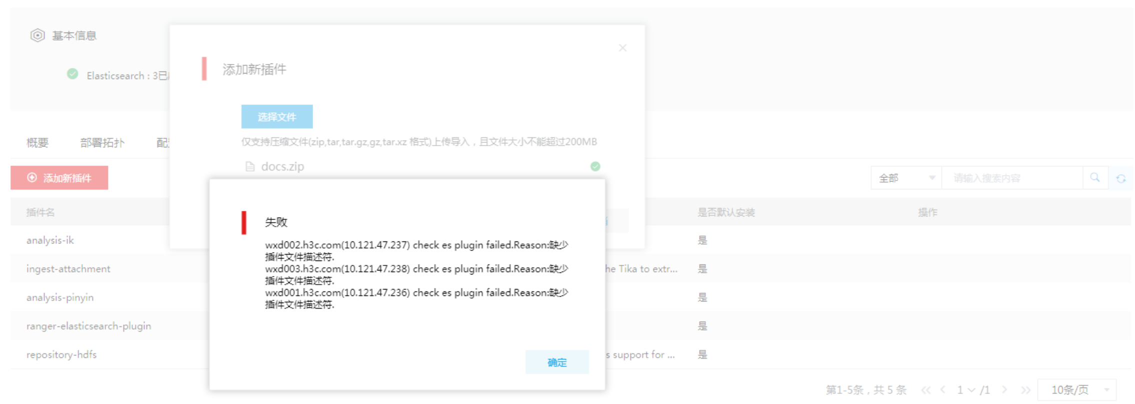
(1) <插件信息>页面，单击<添加新插件>按钮，并选择要添加的插件安装包，可实现插件安装操作，如图 3-17 所示。

图3-17 添加新插件



(2) 在安装插件过程中，会对用户提供的插件安装包进行验证，主要检查插件安装包是否符合 Elasticsearch 插件规范、Elasticsearch 版本要求等，一旦发现插件安装包不符合要求，将终止插件的安装操作，并返回错误信息，如图 3-18 所示，由于插件不符合要求，安装失败。

图3-18 错误信息



(3) 插件安装成功后，在 Elasticsearch 插件信息页面，可查看安装插件相关信息。这里以常用的 Elasticsearch 中文分词器插件 hanlp 简单描述一下插件的使用方式。当 hanlp 插件安装成功之后，重启 Elasticsearch 组件并进入后台，执行命令查看 Elasticsearch 已安装的插件，如图 3-19 所示。

图3-19 查看已安装的插件

```
[sysadmin@sharedev1 ~]$ curl --negotiate -u: sharedev1:9200/_cat/plugins?v
name                 component                      version
sharedev2.hde.com analysis-ik                   7.10.0
sharedev2.hde.com analysis-pinyin               7.10.0
sharedev2.hde.com ingest-attachment             7.10.0
sharedev2.hde.com ranger-elasticsearch-plugin 1.2.0.3.1.0.26-1
sharedev2.hde.com repository-hdfs               7.10.0
sharedev1.hde.com analysis-ik                   7.10.0
sharedev1.hde.com analysis-pinyin               7.10.0
sharedev1.hde.com ingest-attachment             7.10.0
sharedev1.hde.com ranger-elasticsearch-plugin 1.2.0.3.1.0.26-1
sharedev1.hde.com repository-hdfs               7.10.0
sharedev3.hde.com analysis-ik                   7.10.0
sharedev3.hde.com analysis-pinyin               7.10.0
sharedev3.hde.com ingest-attachment             7.10.0
sharedev3.hde.com ranger-elasticsearch-plugin 1.2.0.3.1.0.26-1
sharedev3.hde.com repository-hdfs               7.10.0
```

(4) 查看 hanlp 插件对中文的分词效果，如图 3-20 所示。

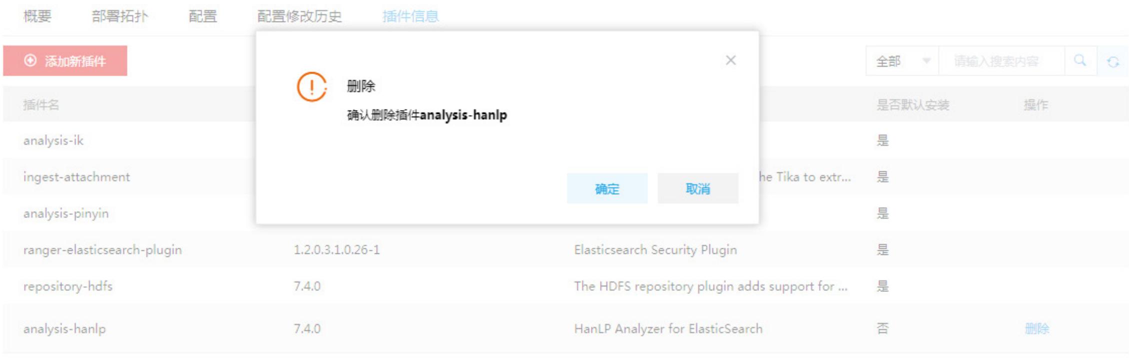
图3-20 查看中文分词效果

```
[root@wxd001 ~]#  curl -X GET -H "Content-type:application/json" --negotiate -u : wxd001:9200/_analyze?pretty -d '
> {
> "text":"地球潮汐是受月球影响。",
> "analyzer":"hanlp"
> }'
{
  "tokens" : [
    {
      "token" : "地球",
      "start_offset" : 0,
      "end_offset" : 2,
      "type" : "n",
      "position" : 0
    },
    {
      "token" : "潮汐",
      "start_offset" : 2,
      "end_offset" : 4,
      "type" : "n",
      "position" : 1
    },
    {
      "token" : "是",
      "start_offset" : 4,
      "end_offset" : 5,
      "type" : "v",
      "position" : 2
    },
    {
      "token" : "受",
      "start_offset" : 5,
      "end_offset" : 6,
      "type" : "v",
      "position" : 3
    },
    {
      "token" : "月球",
      "start_offset" : 6,
      "end_offset" : 8,
      "type" : "n",
      "position" : 4
    },
    {
      "token" : "影响",
      "start_offset" : 8,
      "end_offset" : 10,
      "type" : "vn",
      "position" : 5
    },
    {
      "token" : "。",
      "start_offset" : 10,
      "end_offset" : 11,
      "type" : "w",
      "position" : 6
    }
  ]
}
```

3.8.3 删除插件

对于系统默认预装的Elasticsearch插件，不支持卸载，用户仅可以对自行安装的插件进行卸载操作。用户在<插件信息>页面，单击要删除的插件对应的<删除>按钮，即可实现插件删除操作，如图3-21所示。插件删除完成之后，再次查看<插件信息>页面，发现删除的插件已不在当前Elasticsearch集群中。

图3-21 删除插件



3.9 Elasticsearch快照备份和恢复

注意

- Elasticsearch快照支持共享文件系统HDFS和NFS，其中HDFS需要插件支持repository-hdfs，该插件已经集成在当前集群，可以直接使用，用户可以根据实际情况选择快照存储的仓库介质。这里列举共享文件系统 HDFS 两种快照的备份和恢复的例子供参考，快照的仓库需要用户根据实际情况自行搭建。
- Elasticsearch 的快照管理依托它的可视化工具 Kibana 完成。
- Elasticsearch 的快照可用于 Elasticsearch 跨集群的容灾备份。
- 在升级前备份数据的时候，仅支持 Elasticsearch 同版本之间的快照备份和恢复，例如在 7.x 版本中创建的快照可以恢复到 7.x 版本。如果跨大版本数据迁移可以考虑使用 redindexAPI。

Elasticsearch 快照备份和恢复的整体思路如下：

- 构建 Elasticsearch 集群的快照仓库。
- 通过可视化工具 Kibana 进行快照备份和进度监控。
- 通过可视化工具 Kibana 进行快照恢复。

3.9.1 构建 Elasticsearch 集群的快照仓库

注意

多个集群注册同一个快照仓库，只有一个集群可以对仓库进行写访问，其他所有集群应该设置该仓库为 readonly 模式。

构建 Elasticsearch 集群的快照仓库时，可通过 HDFS 仓库构建或通过 NFS 仓库构建，用户根据实际情况二选一即可。

1. HDFS 仓库构建(推荐使用)

(1) 支持仓库的插件已经集成在安装 Elasticsearch 的集群，在安装有 HDFS 的集群任意节点，创建 hdfs 目录。

hdfs dfs -mkdir /elasticsearch。

(2) 执行如下命令为该目录赋予 elasticsearch 用户权限。

su hdfs(如果是开启了安全模式，请认证用户票据之后操作)

hdfs dfs -chown elasticsearch:elasticsearch /elasticsearch

(3) 查看文件是否赋权正确并且相关文件存在。

[hdfs@lbw237 root]$ hdfs dfs -ls /

drwxr-xr-x - elasticsearch elasticsearch 0 2021-01-16 15:58 /elasticsearch

(4) 然后使用 API 构建 HDFS 仓库，配置示例如下：

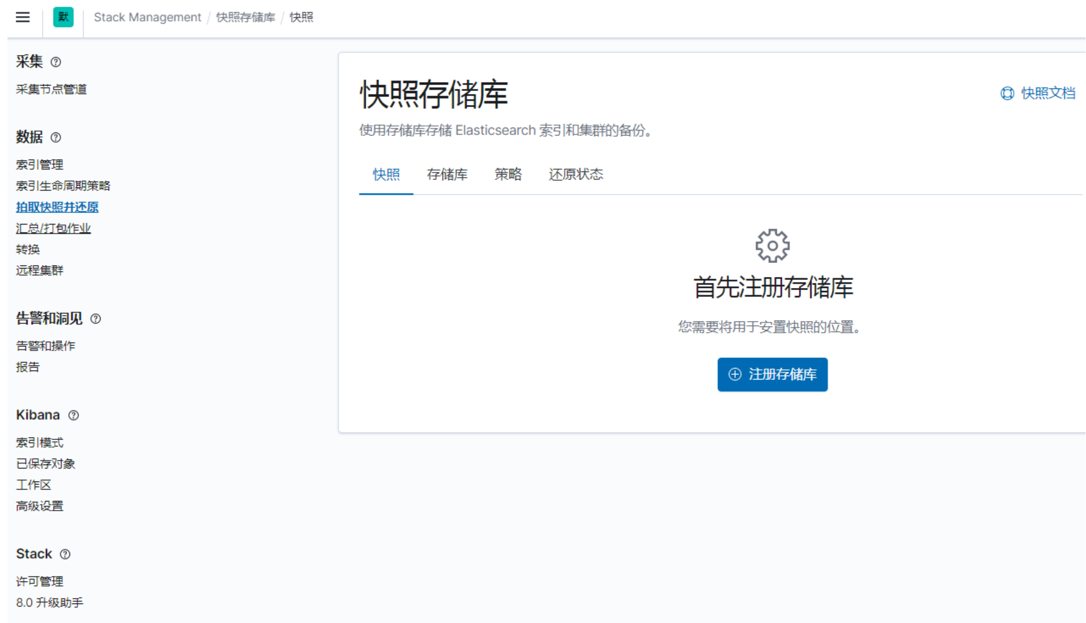
说明

- 快照名称 my_hdfs_repository 在同一个仓库里面唯一。
- "uri"：可以登录集群中任一节点，然后在/usr/hdp/3.0.1.0-187/hadoop/conf 配置文件中进行查看 uri。

```
curl -XPUT -H 'Content-type:application/json' http://ip:9200/_snapshot/my_hdfs_repository -d '
{
  "type":"hdfs",
  "settings":{
    "uri":"hdfs://10.121.64.237:8020",
    "path":"/elasticsearch"
  }
}'
```

(5) 除了使用 API 构建 HDFS 仓库，用户也可以选择在 Kibana 的仓库存储管理界面构建 HDFS 仓库，Kibana 的仓库存储管理界面如图 3-22 所示。

图3-22 Kibana 配置页面



(6) 在 Kibana 的仓库存储管理页面，单击<注册存储库>按钮，进入用户注册存储库页面，配置参数信息，如图 3-23 所示。

图3-23 注册存储库

注意：用户注册存储库时，配置信息因 Elasticsearch 集群是否开启 Kerberos 认证而不同。若 Elasticsearch 集群未开启 Kerberos 认证，在注册存储库时可以只填写 URL 和路径；若 Elasticsearch 集群开启了 Kerberos 认证，在注册存储库时必须填写 URL、路径和安全主体。获取集群超级用户的安全主体的方式如下：

a. ssh 登录集群任意节点，执行如下命令：

```
[root@sharedev1 ~]# klist -kt /etc/security/keytabs/user01.keytab
Keytab name: FILE:/etc/security/keytabs/user01.keytab
KVNO Timestamp           Principal
---- ------------------- ------------------------------------------------------
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
   1 02/28/2022 04:50:21 user01@SHAREDEVTEST.COM
```

b. 根据上述命令执行结果，可知安全主体为 user01@SHAREDEVTEST.COM。

(7) 相关信息正确配置后，HDFS 仓库即可构建成功。

2. NFS 仓库构建

1）NFS 服务端配置

(1) 在大数据集群任一节点执行 ***rpm -qa nfs-utils*** 命令确认是否安装了 nfs-utils 包，若未安装，使用 ***yum install*** 命令安装即可。

(2) 创建目录用于 nfs 共享，如 ***mkdir -p /opt/nfs***。(注意不要创建在/home 下，Elasticsearch 对其没有权限)；/opt/nfs 需要足够的存储空间用于备份 Elasticsearch 的快照数据，空间不足会导致备份失败。

(3) 执行如下命令进行授权：

```
chmod -R 777 /opt/nfs
chown -R elasticsearch:elasticsearch /opt/nfs
```

(4) 编辑/etc/exports 文件，在文件最后追加/opt/nfs *(rw)。

(5) 重启 nfs 服务：systemctl restart nfs。

(6) 检查 nfs 共享目录状态：cat /var/lib/nfs/etab。

2）Elasticsearch 集群节点配置

在 Elasticsearch 集群除了 NFS 服务端的节点，都要挂载 NFS 的数据。

(1) 执行 rpm -qa nfs-utils 确认是否安装了 showmount 包，若未安装，则使用 yum install 命令安装即可。

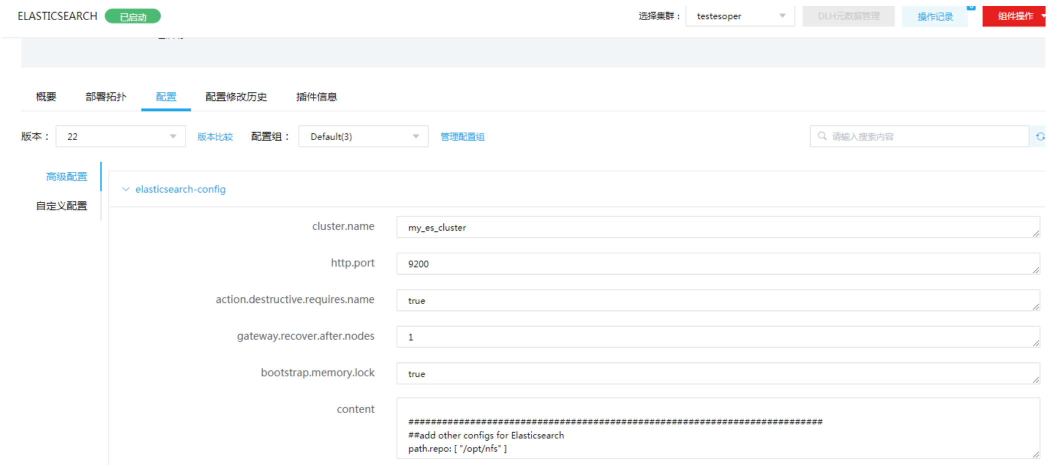
(2) 执行如下命令，挂载 NFS 的数据点，命令中的 IP 为服务端 IP：

```
showmount -e 10.121.47.236。
mkdir -p /opt/nfs
chmod -R 777 /opt/nfs
chown -R elasticsearch:elasticsearch /opt/nfs
mount -t nfs 10.121.47.236:/opt/nfs/ /opt/nfs
```

(3) Elasticsearch 集群配置仓库存储路径。

在 elasticsearch-config 中的 content 中增加 path.repo: ["/opt/nfs"]配置，然后重启 Elasticsearch。

图3-24 修改 Elasticsearch 集群配置



(4) 这样 Elasticsearch 的 NFS 存储的仓库构建完成。

(5) 也可以使用 API 构建 NFS 存储的仓库，配置示例如下：。

```
_snapshot/my_back1    PUT
{
  "type":"fs",
  "settings":{
    "location":"/opt/nfs"
  }
}
```

3.9.2 Elasticsearch 集群快照备份

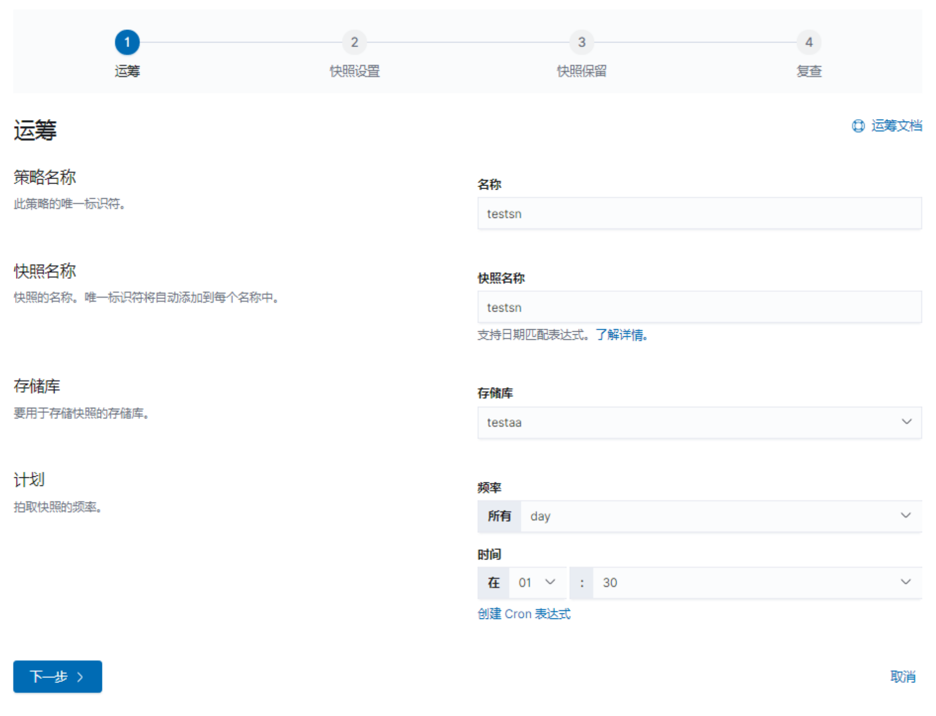
用户构建完成 Elasticsearch 集群的快照仓库后，可以通过可视化工具 Kibana 进行快照备份和进度监控。

(1) 在快照存储库中创建快照。

图3-25 创建快照

(2) 根据实际情况配置创建策略。

图3-26 配置创建策略



(3) 预览创建的快照的详细信息。

图3-27 快照预览

(4) 可在快照列表中查看创建的快照。

图3-28 快照创建成功

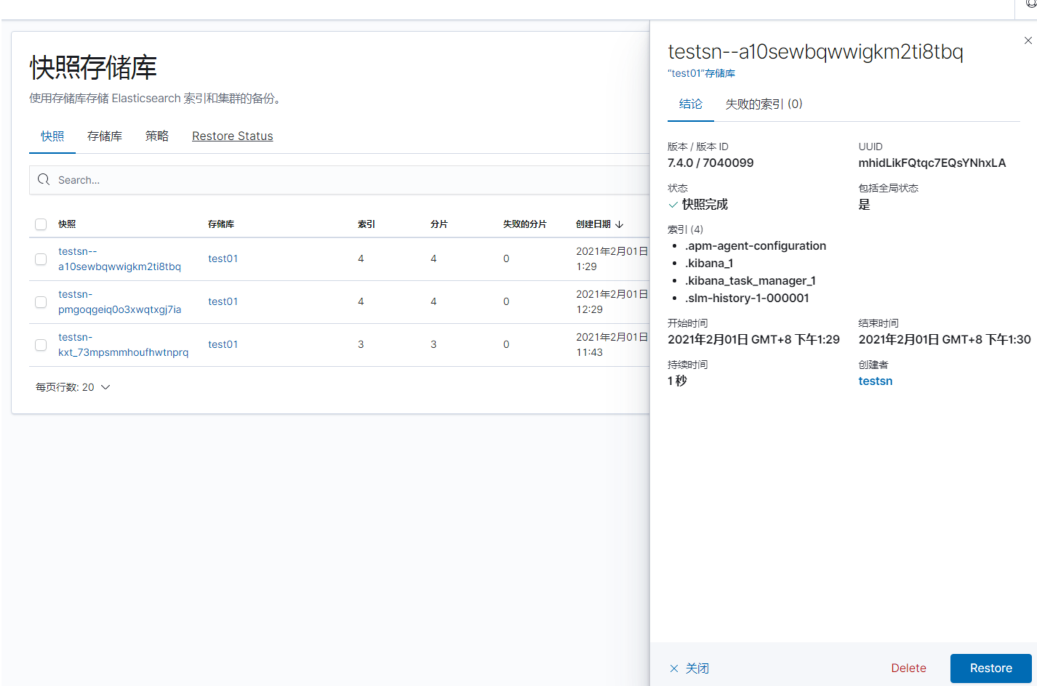
3.9.3 Elasticsearch 集群快照恢复

集群快照备份后，可以通过可视化工具 Kibana 进行快照恢复。

1. 同集群快照恢复

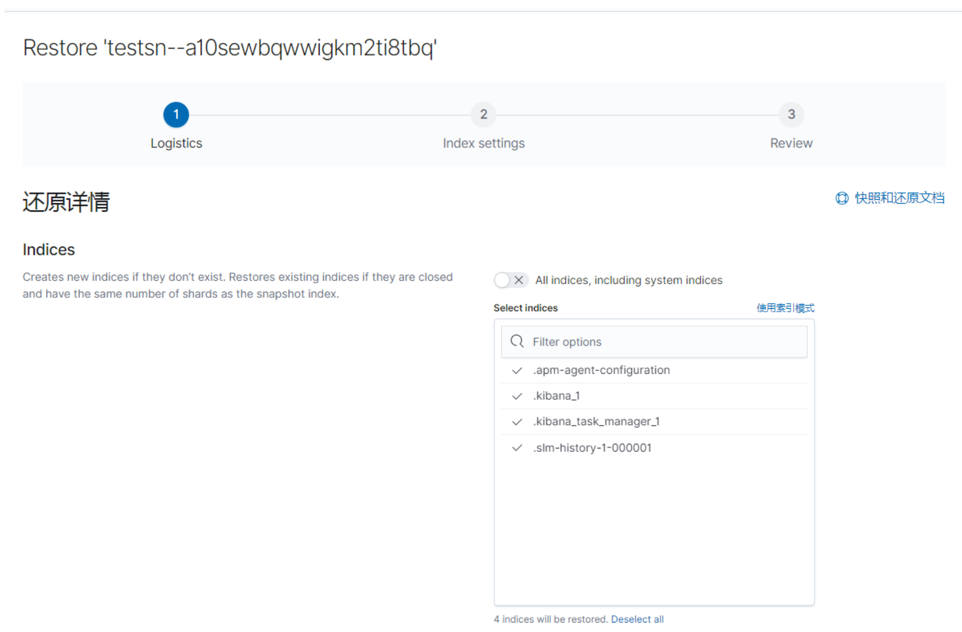
(1) 查看系统中的快照列表及详情。

图3-29 快照列表



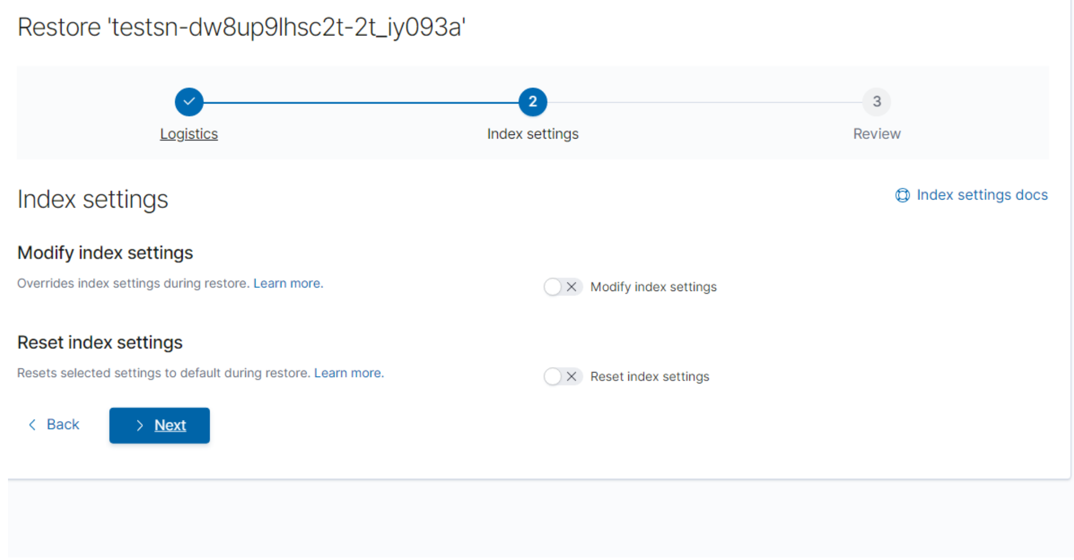
(2) 选择需要的快照并进行还原。

图3-30 快照还原



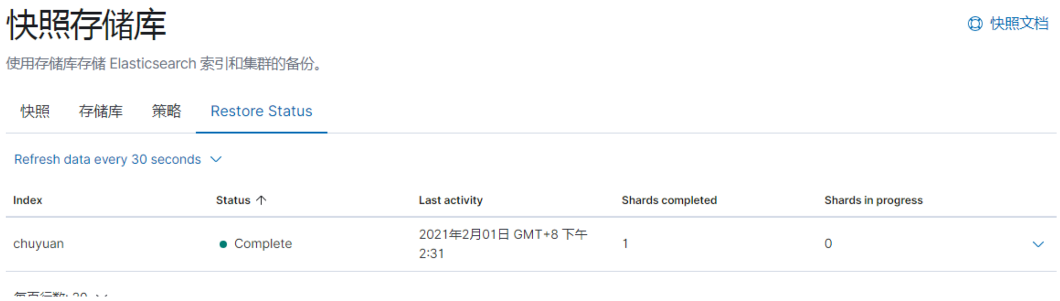
(3) 勾选需要恢复的索引。

图3-31 选择索引



(4) 查看快照恢复进度，等待快照恢复完成。

图3-32 进度监控



2. 跨集群快照恢复

需要在目的集群创建仓库，步骤和 3.9.1 中的 HDFS 仓库创建一样，注意 Elasticsearch 的快照，只对一个集群赋予了写的权限，其他集群只有读的权限。此时仓库创建好之后，可以直接看到源集群的快照。跨集群的快照恢复和同集群步骤相同。

图3-33 快照列表

快照存储库

快照文档

使用存储库存储 Elasticsearch 索引和集群的备份。

快照 存储库 策略 Restore Status

Search... 存储库 重新加载

快照	存储库	索引	分片	失败的分片	创建日期	持续时间	Actions
testsn--a10sewbqwwigkm2ti8tbq	test01	4	4	0	2021年2月01日 GMT+8 下午 1:29	1 秒	
testsn-pmgoqgeiq0o3xwqtxgj7ia	test01	4	4	0	2021年2月01日 GMT+8 下午 12:29	2 秒	
testsn-kxt_73mpsmmhoufhwtnprq	test01	3	3	0	2021年2月01日 GMT+8 上午 11:43	24 秒	

每页行数: 20

4 开发指南

4.1 API使用

4.1.1 集群 API

1. 集群健康状态

- 通过该 API 可以获取一个集群健康的简要状态。例如下面的 API 可以获取集群健康状态、节点数、分片数据及副本数等信息。

  curl -H 'Content-type:application/json' -XGET 'localhost:9200/_cluster/health?pretty'

  响应如下：

```
{
"cluster_name" : "testcluster",
"status" : "yellow",
"timed_out" : false,
"number_of_nodes" : 1,
"number_of_data_nodes" : 1,
"active_primary_shards" : 5,
"active_shards" : 5,
"relocating_shards" : 0,
"initializing_shards" : 0,
"unassigned_shards" : 5,
"delayed_unassigned_shards": 0,
"number_of_pending_tasks" : 0,
"number_of_in_flight_fetch": 0,
"task_max_waiting_in_queue_millis": 0,
"active_shards_percent_as_number": 50.0
}
```

- 该 API 也可以针对一个或多个索引执行请求以获得指定索引的健康状态：

  curl -XGET 'localhost:9200/_cluster/health/test1,test2?pretty'

  集群健康的状态是：green、yellow 或 red。从分片的角度来说：

  - red 表示索引存在不能够提供正常服务的主分片。
  - yellow 表示索引的主分片正常，但是存在不能够提供正常服务的副本分片。
  - green 表示索引完全正常，主分片和副本分片均正常。

  索引的状态由最差的分片状态决定。集群状态由最差的索引状态所决定。

- 该 API 规定了一个超时阈值，在达到这个阈值之前如果集群达到了某个标记的健康级别就返回。例如，若集群在 50 秒以内，达到 yellow 级别将立即返回集群状态（如果在 50 秒之前达到了 green 或 yellow 状态，它将在这时候返回）：

  curl -H "Content-type:application/json" -XGET
  'localhost:9200/_cluster/health?wait_for_status=yellow&timeout=50s&pretty'

2. 集群状态

通过该 API 可以获取一个集群更全面的状态（包含 Elasticsearch 集群节点信息、集群级别的设置、集群中索引信息以及分片的位置信息等）。

curl -H "Content-type:application/json" -XGET 'http://localhost:9200/_cluster/state?pretty'

默认情况下，该集群状态的请求被路由到 Master（主）节点，确保返回最新的集群状态信息。

3. Reroute

该 API 可以通过 ***reroute*** 命令对路由进行重新分配。例如，可以将一个分片使用“move”关键字从一个节点明确的移动到另一个节点。

下面是一个简单的重新路由 API 调用的例子（例子中索引名称是 route-test）。

(1) 查看索引中分片的详细信息：

curl -H 'Content-Type:application/json' -XGET http://101.66.62.160:9200/_cat/shards/route-test?v

返回信息如下：

```
index       shard   prirep   state     docs   store        ip              node
route-test    0       p      STARTED    0     230b     101.66.62.160    test160.hde.com
```

从返回信息中可以看到该索引有一个分片，位于节点 test160.hde.com 上，分片号是 0。

(2) 获取 Elasticsearch 集群中的节点信息，获知集群中每一个节点的名字。

curl -H 'Content-Type:application/json' -XGET http://101.66.62.160:9200/_cat/nodes?v

返回信息如下：

```
ip   heap.Percent  ram.percent  cpu  load_1m  load_5m  load_15m   node.role  master  name
101.66.62.161   34    63       1       0.68     0.67      0.64    dim    -    test161.hde.com
101.66.62.162    31    58      1        0.62     0.74      0.85    dim   *    test162.hde.com
101.66.62.160    20    76      2         0.76    0.79       1.05   dim    -    test160.hde.com
```

(3) 进行重新路由，将 route-test 中的分片 0 从 test160.hde.com 节点重新路由到节点 test161.hde.com 上。

```
curl -H 'Content-Type:application/json' -XPOST http://101.66.62.160:9200/_cluster/reroute?pretty
-d '{
"commands":[{
"move":{
"index":"route-test",
"shard":0,
"from_node":"test160.hde.com",
"to_node":"test161.hde.com"
}
}]
}'
```

4. 集群统计信息

通过该 API 可以获取集群范围内的统计信息。该 API 返回基本的索引 metric（分片数量、存储大小、内存使用）和关于当前集群（节点的编号、角色、系统等信息）中节点的信息。

curl -XGET 'http://localhost:9200/_cluster/stats?pretty'

5. 集群配置更新

通过该 API 可以更新集群范围中指定的配置。更新的配置既可以是永久的（需要重启集群），也可以是瞬时的（不需要重启集群）。示例：

```
curl -H "Content-type:application/json" -XPUT localhost:9200/_cluster/settings -d
'{
"persistent":{
"discovery.zen.minimum_master_nodes":2
}
}'
```

或者：

```
curl -H "Content-type:application/json" -XPUT localhost:9200/_cluster/settings -d '{
"transient" : {
"discovery.zen.minimum_master_nodes" : 2
}
}'
```

6. 集群节点信息

通过该 API 可以获取集群中一个或多个节点的信息。

- 获取集群中所有节点的概要信息

  curl -XGET 'http://localhost:9200/_cat/nodes?v'

- 获取集群中所有节点的详细信息

  curl -XGET 'http://localhost:9200/_nodes?pretty'

- 获取了 nodeId1 和 nodeId2 的节点的统计信息

  curl -XGET 'http://localhost:9200/_nodes/nodeId1,nodeId2'

4.1.2 索引 API

1. 创建索引

创建索引（index API）允许实例化一个索引。Elasticsearch 支持多种索引操作，包括跨多个索引执行操作。

- 索引设置

  每个索引创建的时候可以包含与之关联的特定设置。

  ```
  curl -H "Content-type:application/json" -XPUT 'localhost:9200/weibo?pretty' -d'
  {
  "settings": {
  "index": {
  "number_of_shards" : 3,
  "number_of_replicas" : 2
  }
  }
  }'
  ```

  上述 curl 命令创建了一个 weibo 索引，包含 3 个分片，2 个副本。默认 number_of_shards 为 5，默认 number_of_replicas 为 1(对于主 shard 只有一个副本)。

  命令可简化为：

```
curl -H "Content-type:application/json" -XPUT localhost:9200/weibo?pretty -d'
{
"settings" : {
"number_of_shards" :3,
"number_of_replicas" :2
}
}'
```

- Mappings/映射

  在 Elasticsearch7.10 中，创建映射时无需再指定 type，否则会报错。创建索引允许提供包含一个或多个映射的设置：

```
curl -H "Content-type:application/json" -XPUT 'localhost:9200/test?pretty' -d'
{
"settings" : {
"number_of_shards" : 1
},
"mappings" : {
"properties" : {
"field1" : { "type" : "text" }
}
}
}'
```

- Aliases/别名

  创建索引可以设置其别名：

```
curl -H "Content-type:application/json" -XPUT 'localhost:9200/test?pretty' -d'
{
"aliases" : {
"alias_1" : {},
"alias_2" : {
"filter" : {
"term"   : {"user" : "kimchy" }
},
"routing" : "kimchy"
}
}
}'
```

  上面的示例是在创建 test 索引的同时制定了两个别名 alias_1 和 alias_2。其中 alias_2 给该别名指定了过滤操作，所有使用该别名的操作都是在对 test 索引进行 filter 过滤之后的结果集上进行的。

2. 获取索引信息

获取索引信息 API 可以获得一个或多个索引的信息，API 必须指定一个索引、别名或通配符表达式。例如获得 weibo 索引的信息，命令如下：

```
curl -XGET 'localhost:9200/weibo?pretty'
```

3. 删除索引

删除索引 API 可以用来删除集群中的索引，删除必须指定索引、别名或通配符表达式，例如删除 weibo 索引，命令如下：

curl -H "Content-Type:application/json" -XDELETE 'localhost:9200/weibo?pretty'

删除索引 API 也可以应用于多个索引，通过使用逗号分隔。或者通过_all 或者*，来删除所有 index。为了安全起见，可以通过设置 action.destructive.requires.name 为 true（即打“√”的形式）来避免通配符或者_all 进行删除所有索引操作，这个设置也可以通过集群 API 设置。

4. 判断索引是否存在

该 API 用来检查索引是否存在。例如：

curl -H "Content-Type:application/json" -X GET 'localhost:9200/weibo?pretty'

HTTP 状态码表示 index 存在与否。如果不存在提示 404，如果存在，将返回索引信息。

5. 提交映射

提交映射允许提交自定义的类型映射至一个新的索引，或者往已存在的索引中新加一个字段。

- 创建一个名叫 weibo 的索引，包含一个字段 message。

```
curl -H "Content-type:application/json" -XPUT 'localhost:9200/weibo?pretty' -d'
{
"mappings": {
"properties": {
"message": {
"type":"text"
}
}
}
}'
```

- 使用 PUT mapping API 增加一个字段。

```
curl -H "Content-type:application/json" -XPUT 'localhost:9200/weibo/_mapping?pretty'
-d'
{
"properties": {
"name": {
"type": "text"
}
}
}'
```

6. 获取映射

获取映射 API 允许获取索引或索引类型的映射定义。

curl -H "Content-type:application/json" -XGET 'localhost:9200/weibo/_mapping?pretty'

7. 获取字段映射

获取字段映射 API 允许检索一个或多个字段的映射定义。

- 仅返回字段文本的映射。

  curl -XGET 'localhost:9200/weibo/_mapping/field/message?pretty'

- 返回多个字段（如获取 message、date 字段）的映射定义。

  curl -XGET 'localhost:9200/weibo/_mapping/field/message,date?pretty'

8. 索引统计信息

索引级统计信息 API 提供索引上发生不同操作的统计信息。API 提供了索引级范围的统计信息（虽然大多数统计信息可以使用节点级别范围检索）。

可以使用以下命令，返回所有索引的高聚合和索引等级的统计信息请求：

```
curl -X GET 'localhost:9200/_stats?pretty'
```

可以使用以下方式检索特定索引的统计信息：

```
curl -X GET 'localhost:9200/index1,index2/_stats?pretty'
```

默认情况下，返回所有统计信息。可以通过指定 URL 返回特定的统计信息。这些统计信息可以是表 4-1 中的任意一种。

表4-1 统计信息列表

字段	说明
docs	文档和已删除文档（尚未合并的文档）的数量 注意，此值受刷新索引的影响
store	索引的大小
indexing	索引统计信息，可以用逗号分隔的type列表组合，以提供文档级统计信息
get	获取统计信息，包括缺失的统计信息
search	包含建议统计信息的搜索统计信息。可以通过添加额外group参数（搜索操作可以与一个或多个group相关联）来包含自定义组的统计信息。groups参数接受以逗号分隔的group名称列表。使用_all返回所有组的统计信息
segments	检索打开的分段的内存使用。可以选择设置include_segment_file_sizes标志，报告每个Lucene索引文件的聚合磁盘使用情况
completion	完成建议统计
fielddata	正排索引统计信息
flush	刷新统计信息
merge	合并统计信息
request_cache	Shard request cache statistics
refresh	刷新统计信息
warmer	Warmer statistics
translog	事务日志统计信息

9. 清理缓存

清理缓存 API 可清除与一个或多个索引相关联的所有缓存和特定缓存。

```
curl -H "Content-Type:application/json"  -XPOST  'localhost:9200/weibo/_cache/clear?pretty'
```

默认情况下，API 会清除所有缓存。可以通过设置 query、fielddata 或者 request 来显式清除特定高速缓存。

与特定字段相关的所有高速缓存也可以通过使用逗号分隔符的相关字段列表指定 fields 参数来清除。

清除缓存 API 可以通过单个调用应用于多个索引，甚至可以应用于_all 索引。

curl -H "Content-Type:application/json" -XPOST 'localhost:9200/index1,index2/_cache/clear?pretty'

10. 刷新

刷新 API 允许通过 API 刷新一个或多个索引。索引的刷新过程基本上通过将数据刷新到索引存储、清除内部事务日志来释放索引的内存。默认情况下，Elasticsearch 使用内存触发的方式，以便根据需要自动触发刷新操作，以清理内存。

curl -H "Content-Type:application/json" -XPOST
'localhost:9200/weibo/_flush?wait_if_ongoing=true&force=true'

表4-2 请求参数列表

字段	说明
wait_if_ongoing	设置为true，如果另一个刷新操作正在执行，则当前刷新操作将阻塞，只到刷新可以被执行。默认值为false，如果另一个flush操作正在执行，将会导致抛出分片级别的异常
force	强制刷新缓存中的内容到磁盘

刷新 API 可以通过一次调用应用于一个或多个索引，甚至应用于_all 索引。

curl -H "Content-Type:application/json" -XPOST 'localhost:9200/index1,index2/_flush?pretty'

11. Refresh

该 API 允许用户显式地刷新一个或多个索引，同时默认情况下，Elasticsearch 内部也会周期性地调用该 API 对索引进行刷新，从而使得上次刷新操作与本次刷新操作期间对索引的更新能够被搜索到。

curl -H "Content-Type:application/json" -XPOST 'localhost:9200/weibo/_refresh?pretty'

重新加载 API 可以通过单个调用应用于多个索引，甚至可以应用在_all 索引上。

curl -H "Content-Type:application/json" -XPOST 'localhost:9200/index1,index2/_refresh?pretty'

12. 查看集群包含的索引

可通过该 API 查看集群中存在的索引，如下：

curl localhost:9200/_cat/indices

【说明】上述指令默认不会展示隐藏的索引名称，若需全部显示则增加参数即可，如下：

curl localhost:9200/_cat/indices?expand_wildcards=all

4.1.3 文档 API

文档是 Elasticsearch 中索引的主要实体，文档由字段构成。程序中大多的实体或对象能够被序列化为包含键值对的 JSON 对象，键（key）是字段（field）或属性（property）的名字，值（value）可以是字符串、数字、布尔类型、另一个对象、值数组或者其他特殊类型，比如表示日期的字符串或者表示地理位置的对象。以下是一个常见的文档：

```
{
    "name":         "John Smith",
    "age":          42,
    "confirmed":    true,
    "join_date":    "2014-06-01",
    "home": {
        "lat":      51.5,
```

```
        "lon":      0.1
    },
    "accounts": [
        {
            "type": "facebook",
            "id":   "johnsmith"
        },
        {
            "type": "weibo",
            "id":   "johnsmith"
        }
    ]
}
```

通常认为对象（object）和文档（document）是等价相通的。不过两者之间还存在差别：

- 对象（Object）是一个 JSON 结构体（类似于哈希、hashmap、字典或者关联数组）；对象（object）中还可能包含其他对象（object）。
- 在 Elasticsearch 中，文档（document）这个术语有着特殊含义。它特指最顶层结构或者根对象（root object）序列化成的 JSON 数据（以唯一 ID 标识并存储于 Elasticsearch 中）。

表4-3 文档元数据

节点	说明
_index	文档存储的地方
_type	文档代表的对象的类
_id	文档的唯一标识

1. 索引一个文档

文档通过 index API 被索引，使数据可以被存储和搜索。文档通过其_index、_type、_id 唯一确定。可以手动指定一个_id 或者使用默认生成。

- 手动指定 ID

  如果文档有自然的标识符（例如 user_account 字段或者其他值表示文档），就可以指定_id，使用以下形式的 index API：

```
curl -H "Content-Type:application/json" -XPUT
'localhost:9200/<index>/<type>/<id>?pretty'
{
      "field": "value",
        ...
}
```

  例如索引名称为“website”，类型为“blog”，手动指定的 ID 为“123”，索引请求如下：

```
curl -H "Content-Type:application/json"  -XPUT 'localhost:9200/weibo/blog/123?pretty'
-d'
{
     "title": "My first blog entry",
     "text":  "Just trying this out...",
     "date":  "2020/01/01"
```

```
}'
```

Elasticsearch 的返回值：

```
{
    "_index":    "website",
    "_type":     "blog",
    "_id":       "123",
    "_version":  1,
    "created":   true
}
```

响应表明请求的索引已经被成功创建，这个索引中包含_index、_type 和_id 元数据，以及一个新元素：_version。

Elasticsearch 中每个文档都有版本号，每当文档变化（包括删除），Version 都会增加。

- 自增 ID

  如果数据没有指定的 ID，Elasticsearch 可以自动生成。

  URL 现在只包含_index 和_type 两个字段：

```
curl -H "Content-Type:application/json" -XPUT 'localhost:9200/weibo/blog?pretty' -d'
{
     "title": "My second blog entry",
    "text":  "Still trying this out...",
    "date":  "2020/01/01"
}'
```

  响应内容与刚才类似，只有_id 字段变成了自动生成的值：

```
{
    "_index":    "website",
    "_type":     "blog",
    "_id":       "AVaC8E_4YMSUVyM31XBQ",
    "_version":  1,
    "created":   true
}
```

  自动生成的 ID 有 22 个字符长，是基于 URL-safe Base64 编码的全局唯一的标识符。

2. 检查文档是否存在

请求方式如下：

curl -i -XHEAD http://localhost:9200/website/blog/123

响应结果如下：

- 如果文档存在，Elasticsearch 将会返回 200 OK 状态：

```
HTTP/1.1 200 OK
Warning: 299 Elasticsearch-7.10.0-22e1767283e61a198cb4db791ea66e3f11ab9910 "[types removal] Specifying types in document get requests is deprecated, use the /{index}/_doc/{id} endpoint instead."
Content-Type: text/plain; charset=UTF-8
Content-Length: 200
```

- 如果不存在返回 404 Not Found：

```
HTTP/1.1 404 Not Found
```

```
Warning: 299 Elasticsearch-7.10.0-22e1767283e61a198cb4db791ea66e3f11ab9910 "[types removal] Specifying types in document get requests is deprecated, use the /{index}/_doc/{id} endpoint instead."
Content-Type: text/plain; charset=UTF-8
Content-Length: 89
```

3. 更新整个文档

- 在索引中已经存在的文档是不可变的。如果需要更新已存在的文档，可使用 index API 重建索引（reindex）或者替换掉它。

```
curl -H "Content-Type:application/json" -XPUT 'localhost:9200/weibo/blog/123?pretty'
{
  "title": "My first blog entry",
  "text":  "I am starting to get the hang of this...",
  "date":  "2020/01/02"
}
```

  在返回结果中，我们可以看到 Elasticsearch 把_version 增加了，即意味着原文档被替换为新文档。

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 2,
  "created":   false
}
```

  created 标识为 false，表示该索引已经存在相同 ID 的文档。

- Elasticsearch 已经标记旧文档为删除，并添加了一个完整的新文档。旧版本文档不会立即消失，但也不能被访问。Elasticsearch 会在后续操作自动清理被标记删除的文档。

  Update API 修改文档的局部过程如下：

  a. 从旧文档中检索 JSON
  b. 修改
  c. 删除旧文档
  d. 索引新文档

  唯一的不同是 Update API 完成这一过程只需要一个客户端请求即可，不再需要 get 和 index 请求。

4. 创建一个新文档

当索引一个新文档，_index、_type、_id 三者唯一确定一个文档。所以要想保证文档是新加入的，最简单的方式是使用 POST 方法让 Elasticsearch 自动生成唯一_id：

```
curl -H "Content-Type:application/json"  -XPUT  'localhost:9200/weibo/blog?pretty'
{ ... }
```

然而，如果想使用自定义的_id，我们必须告诉 Elasticsearch 应该在_index、_type、_id 三者都不同时才接受请求。为了做到这点有两种方法：

- 第一种方法使用 op_type 查询参数，如果索引中不存在该文档则添加成功，否则添加文档失败，并返回失败原因，索引中已经存在对应文档

```
curl -H "Content-Type:application/json"  -XPUT  'localhost:9200/weibo/blog/123?op_type=create'
{ ... }
```

- 第二种方法是在 URL 后加/_create 做为端点：

```
curl -H "Content-Type:application/json"  -XPUT  'localhost:9200/weibo/blog/123/_create'
{ ... }
```

如果请求成功的创建了一个新文档，Elasticsearch 将返回正常的元数据且响应状态码是 201 Created。

如果包含相同的_index、_type 和_id 的文档已经存在，Elasticsearch 将返回 409 Conflict 响应状态码，错误信息类似如下：

```
{
  "error" : "DocumentAlreadyExistsException[[website][4] [blog][123]:
                   document already exists]",
  "status" : 409
}
```

5. 删除文档

删除文档的语法模式与之前基本一致，需要使用 DELETE 方法：

```
curl -X DELETE  localhost:9200/website/blog/123?pretty
```

如果文档被找到，Elasticsearch 将返回 deleted。

```
{
  "_index" : "weibo",
  "_type" : "_doc",
  "_id" : "123",
  "_version" : 2,
  "result" : "deleted"
}
```

如果文档未找到，我们将得到一个 Not Found 响应，响应体是这样的：

```
{
  "_index" : "weibo",
  "_type" : "_doc",
  "_id" : "1111",
  "_version" : 1,
  "result" : "not_found"
}
```

即使文档不存在字段 found 的值是 false，字段_version 依旧增加 1。这是内部记录的一部分，可以保证多节点间数据操作的一致性。

6. 文档局部更新

使用 update API 请求来实现局部更新，例如增加数量的操作。Update API 必须遵循文档是不可变的，不能被更改，只能被替换的规则。Update API 处理的流程是：检索-修改-重建索引，这样可以减少其他进程可能导致冲突的修改。

最简单的 update 请求接受一个局部文档参数 doc id，它会与现有文档合并在一起，存在的标量字段被覆盖，新字段被添加，更新后的文档_version 字段加 1。

例如，使用以下请求为博客添加一个 tags 字段和一个 views 字段：

```
curl -H "Content-Type:application/json" -XPOST
localhost:9200/website/blog/1/_update?pretty
{
   "doc" : {
      "tags" : [ "tESting" ],
      "views": 0
   }
}
```

请求成功，我们将看到如下响应结果：

```
{
   "_index" :   "website",
   "_id" :      "1",
   "_type" :    "blog",
   "_version" : 3
}
```

检索文档显示被更新的_source 字段：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "1",
   "_version":  3,
   "found":     true,
   "_source": {
      "title":  "My first blog entry",
      "text":   "Starting to get the hang of this...",
      "tags": [ "tESting" ],
      "views":  0
   }
}
```

新添加的字段已经被添加到_source 字段中。

7. 检索多个文档

合并多个请求可以避免每个请求单独的网络开销。批量检索文档，可以使用 multi-get 或者 mget API。Mget API 参数是一个 docs 数组，数组的每个节点定义一个文档的_index、_type、_id 元数据。在检索一个或几个确定的字段，也可以定义一个_source 参数：

```
curl -H "Content-Type:application/json"  -XPOST localhost:9200/_mget?pretty
{
   "docs" : [
      {
         "_index" : "website",
         "_type" :  "blog",
         "_id" :    2
      },
      {
         "_index" : "website",
         "_type" :  "blog",
```

```
            "_id" :    1,
            "_source": "views"
        }
    ]
}
```

响应体也包含一个 docs 数组，每个文档还包含一个响应，它们按照请求定义的顺序排列。每个这样的响应与单独使用 get request 响应体相同：

```
{
    "docs" : [
        {
            "_index" :   "website",
            "_id" :      "2",
            "_type" :    "blog",
            "found" :    true,
            "_source" : {
                "text" :  "This is a piece of cake...",
                "title" : "My first external blog entry"
            },
            "_version" : 10
        },
        {
            "_index" :   "website",
            "_id" :      "1",
            "_type" :    "blog",
            "found" :    true,
            "_version" : 2,
            "_source" : {
                "views" : 2
            }
        }
    ]
}
```

如果检索的文档在同一个_index 中（甚至在同一个_type 中），可以在 URL 中定义一个默认的/_index 或者/_index/_type。

在单独的请求中使用这些值：

```
curl -H "Content-Type:application/json"  -XPOST localhost:9200/website/blog/_mget?pretty
{
    "docs" : [
        { "_id" : 2 },
        { "_type" : "blog", "_id" :   1 }
    ]
}
```

事实上，如果所有文档具有相同_index 和_type，也可以通过简单的 ids 数组来代替完整的 docs 数组：

```
curl -H "Content-Type:application/json"  -XPOST localhost:9200/website/blog/_mget?pretty
{
    "ids" : [ "2", "1" ]
```

```
}
```
注意到我们请求的第二个文档并不存在。我们定义了类型为blog，但是ID为1的文档类型为blog。这个不存在的文档会在响应体中被告知。

```
{
  "docs" : [
    {
      "_index" :   "website",
      "_type" :    "blog",
      "_id" :      "2",
      "_version" : 10,
      "found" :    true,
      "_source" : {
        "title":   "My first external blog entry",
        "text":    "This is a piece of cake..."
      }
    },
    {
      "_index" :   "website",
      "_type" :    "blog",
      "_id" :      "1",
      "found" :    false
    }
  ]
}
```
事实上第二个文档不存在并不影响第一个文档的检索，每个文档的检索和报告都是独立的。

8. 更省时的批量操作

就像mget允许一次性检索多个文档一样，bulk API允许使用单一请求来实现多个文档的create、index、update或delete。这对索引类似于日志这样的数据流非常有用，它们可以以成百上千的数据为一个批次按序进行索引。

bulk请求体如下：

```
{ action: { metadata }}\n
{ request body        }\n
{ action: { metadata }}\n
{ request body        }\n
...
```
这种格式类似于用"\n"符号连接起来的一行一行的JSON文档流（stream）。两个重要的点需要注意：

- 每行必须以"\n"符号结尾，包括最后一行。这些都是作为每行有效的分离而做的标记。
- 每一行的数据不能包含未被转义的换行符，会干扰分析，这意味着JSON不能被格式化输出。

action/metadata 这一行定义了文档行为(what action)发生在哪个文档（which document）。行为(action)必须是表 4-4 中几种行为之一：

表4-4 文档行为

行为	说明
create	创建文档

行为	说明
index	创建新文档或替换已有文档
update	局部更新文档
delete	删除一个文档

在索引、创建、更新或删除时必须指定文档的_index、_type、_id 这些元数据（metadata）。

例如删除请求格式如下：

```
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }}
```

请求体（request body）由文档的_source、文档所包含的一些字段以及其值组成，Index、create 操作请求体需要这样指定；同样 update 操作也需要这样指定，而且请求体的组成应该与 update API（doc，upsert，script 等等）一致。删除操作不需要请求体（request body）。

```
{ "create":  { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
```

如果定义_id，ID 将会被自动创建：

```
{ "index": { "_index": "website", "_type": "blog" }}
{ "title":    "My second blog post" }
```

bulk 请求表单如下：

```
curl -H "Content-Type:application/json"  -XPOST localhost:9200/_bulk?pretty
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
{ "index":  { "_index": "website", "_type": "blog" }}
{ "title":    "My second blog post" }
{ "update": { "_index": "website", "_type": "blog", "_id": "123", "_retry_on_conflict" : 3} }
{ "doc" : {"title" : "My updated blog post"} }
```

请注意 delete 行为（action）没有请求体，它紧接着另一个行为（action）。

上面请求响应结果包含一个 items 数组，其中包含每一个请求的结果，并且结果的顺序与请求的顺序相同：

```
{
   "took": 4,
   "errors": false,
   "items": [
      {  "delete": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 2,
            "status":   200,
            "found":    true
      }},
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
```

```
            "_version": 3,
            "status":   201
      }},
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "EiwfApScQiiy7TIKFxRCTw",
            "_version": 1,
            "status":   201
      }},
      {  "update": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 4,
            "status":   200
      }}
   ]
}}
```

每个子请求都被独立的执行，所以一个子请求的错误并不影响其它请求。如果任何一个请求失败，顶层的 error 标记将被设置为 true，然后错误的细节将在相应的请求中被报告：

```
curl -H "Content-Type:application/json"  -XPOST localhost:9200/_bulk?pretty
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "Cannot create - it already exists" }
{ "index":  { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "But we can update it" }
```

响应中我们将看到 create 文档 123 失败了，因为文档已经存在，但是后面在 123 上执行的 index 请求成功了：

```
{
   "took": 3,
   "errors": true,
   "items": [
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "status":   409,
            "error":    "DocumentAlreadyExistsException
                        [[website][4] [blog][123]:
                        document already exists]"
      }},
      {  "index": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 5,
            "status":   200
      }}
```

```
    ]
}
```

返回结果中部分字段含义如表 4-5 所示：

表4-5 字段含义

字段	说明
"errors": true	一个或多个请求失败
"status":409	这个请求的HTTP状态码被报告为409 CONFLICT
"error":"DocumentAlreadyExistsException [[website][4] [blog][123]: document already exists]"	错误消息说明了请求错误的具体信息

这些说明 bulk 请求不是原子操作，它们不能实现事务。每个请求操作是分开的，所以每个请求的成功与否不干扰其它操作。

当在同一个 index 下的同一个 type 里批量索引日志数据，为每个文档指定相同的元数据是多余的。就像 mget API，bulk 请求也可以在 URL 中使用/_index 或/_index/_type:

```
curl -H "Content-Type:application/json"  -XPOST localhost:9200/website/_bulk?pretty
{ "index": { "_type": "log" }}
{ "event": "User logged in" }
```

整个批量请求需要被加载到接收请求节点的内存里，所以请求越大，给其它请求可用的内存就越小。有一个最佳的 bulk 请求大小，超过这个大小，性能不再提升而且可能降低。最佳大小，并不是一个固定的数字。它取决于硬件、文档的大小和复杂度以及索引和搜索的负载。

最佳点（sweetspot）取值方式如下：批量索引标准的文档，随着大小的增长，当性能开始降低，说明每个批次的大小太大了。开始的数量可以在 1000~5000 个文档之间，如果文档非常大，可以使用较小的批次。基于请求批次的物理大小也是比较重要的方式。一千个 1kB 的文档和一千个 1MB 的文档大不相同，一个好的批次最好保持在 5-15MB。

4.1.4 查询 API

1. 请求体查询

简单查询语句是一种有效的命令行查询，但是对于复杂查询，使用请求体查询（request body search）API 是不可或缺的查询方式，这是因为大多数的参数以 JSON 格式定义而非查询字符串。

请求体查询（下文简称查询），并不仅仅用来处理查询，还可以高亮返回结果中的片段，并且标明用户最佳期望结果的相关数据。

2. 空查询

我们以最简单的 search API 开始，空查询将会返回索引中所有的文档。

```
curl -X GET localhost:9200/website/_search?pretty
{}
```

“{}”表明这是一个空查询数据。

同字符串查询一样，可以查询一个、多个或_all 索引（indices）或类型（types）：

```
curl -X GET localhost:9200/index1,index2/_search?pretty
{}
```

可以使用 from 及 size 参数进行分页：

```
curl -X GET localhost:9200/website/_search?pretty
{
  "from": 30,
  "size": 10
}
```

Elasticsearch 倾向于使用 GET 提交查询请求，这个词相比 POST 来说，能更好的描述这种行为。但是由于携带交互数据的 GET 请求并不被广泛支持，所以 search API 同样支持 POST 请求，类似于这样：

```
curl -H "Content-Type:application/json" -X POST localhost:9200/website/_search?pretty
{
  "from": 30,
  "size": 10
}
```

这个原理同样应用于其他携带交互数据的 GETAPI 请求中。

3. 结构化查询 Query DSL

结构化查询是一种灵活的，多表现形式的查询语言。Elasticsearch 在一个简单的 JSON 接口中用结构化查询来展现 Lucene 绝大多数能力。它使得查询更加灵活，精准，易于阅读并且易于 debug。

使用结构化查询，需要传递 query 参数：

```
curl -X GET localhost:9200/website/_search?pretty
{
    "query": YOUR_QUERY_HERE
}
```

空查询 - {} - 在功能上等同于使用 match_all 查询子句，即匹配所有的文档：

```
curl -X GET localhost:9200/website/_search?pretty
{
    "query": {
        "match_all": {}
    }
}
```

查询子句：一个查询子句一般使用如下结构：

```
{
    QUERY_NAME: {
        ARGUMENT: VALUE,
        ARGUMENT: VALUE,...
    }
}
```

或指向一个指定的字段：

```
{
    QUERY_NAME: {
        FIELD_NAME: {
            ARGUMENT: VALUE,
            ARGUMENT: VALUE,...
        }
    }
```

```
}
```

例如，你可以使用 match 查询子句用来在 tweet 字段中找寻包含 Elasticsearch 的成员：

```
{
    "match": {
        "tweet": "Elasticsearch"
    }
}
```

完整的查询请求如下：

```
curl -X GET localhost:9200/website/_search?pretty
{
    "query": {
        "match": {
            "tweet": "Elasticsearch"
        }
    }
}
```

合并多子句：查询子句就像是搭积木一样，可以合并简单的子句为一个复杂的查询语句，比如：复合子句（compound）用以合并其他的子句。例如，bool 子句允许合并其他的合法子句，must、must_not 或者 should，示例：

```
{
  "bool": {
    "must":     { "match": { "tweet": "Elasticsearch" }},
    "must_not": { "match": { "name":  "mary" }},
    "should":   { "match": { "tweet": "full text" }}
  }
}
```

复合子句能合并任意其他查询子句，包括其他的复合子句。这就意味着复合子句可以相互嵌套，从而实现非常复杂的逻辑。

示例：查询邮件正文中含有“businESs opportunity”字样的星标邮件或收件箱正文中含有“businESs opportunity”字样的非垃圾邮件：

```
{
    "bool": {
        "must": { "match":      { "email": "business opportunity" }},
        "should": [
             { "match":         { "starred": true }},
             { "bool": {
                   "must":      { "folder": "inbox" }},
                   "must_not":  { "spam": true }}
             }}
        ],
        "minimum_should_match": 1
    }
}
```

4. 模糊查询

使用模糊查询，需要传递 query 和 fuzzy 参数：

```
curl –X GET localhost:9200/website/mystyle/4?pretty
```

查询出如下内容：

```
{
"_index":"weibo",
"_type":"mystyle",
"_id":"4",
"_version":1,
"found":true,
"_source":{
"price":10,
"productID":"RTDK-193-JKHYS-YZK-IKDHJD-7",
"date":"2022-04-18 19:35:11"
}
}
```

RTDK-193-JKHYS-YZK-IKDHJD-7 被默认的分词器分为 rtdk、193、jkhys、yzk、ikdhjd 和 7，官网对分词后的字符模糊匹配规则如下：

```
0..2
      must match exactly
3..5
      one edit allowed
>5
      two edits allowed
```

根据测试，此处字符长度应为输入查询字符，而不是分词后的字符。我们以三个字符 yzk 进行验证。针对分词后的 yzk，为三个字符，当使用 yz 查询时，结果如下：

```
curl -XGET http://101.8.134.2:9200/weibo/mystyle/_search?pretty -d
'{"query":{"fuzzy":{"productID":"yz"}}}'
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : null,
    "hits" : [ ]
  }
}
```

查询结果为空（如果按照分词后的 yzk 字符长度应该匹配规则 3..5，使用 yz 符合相差一个字符的规则，应该会输出查询结果），说明使用 yz 查询时匹配 0..2 的规则，需要进行精准匹配，而分词后的内容不包含 yz，所以查询结果为空，使用 yzkd 和 yzb 进行查询，结果如下：

```
curl -XGET http://101.8.134.2:9200/weibo/mystyle/_search?pretty -d
'{"query":{"fuzzy":{"productID":"yzb"}}}'
```

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.49605155,
    "hits" : [
      {
        "_index" : "weibo",
        "_type" : "mystyle",
        "_id" : "4",
        "_score" : 0.49605155,
        "_source" : {
          "price" : 10,
          "productID" : "RTDK-193-JKHYS-YZK-IKDHJD-7",
          "date" : "2022-04-18 19:35:11"
        }
      }
    ]
  }
}
```

符合 3..5 规则规定相差一个字符，也可使用 5 个字符进行测试。其他规则均按查询时字符串长度对应相应规则即可。

4.1.5 排序 API

1. 相关性排序

默认情况下，结果集会按照相关性进行排序，相关性越高，排名越靠前。

2. 排序方式

为了使结果可以按照相关性进行排序，需要一个相关性的值。在 Elasticsearch 的查询结果中，相关性的值会用_score 字段来给出一个浮点型的数值，所以默认情况下，结果集以_score 进行倒序排列。

有时，即便如此，还是没有一个有意义的相关性的值。比如，以下语句返回所有 tweets 中 user_id 是否包含值 1。

```
curl -X GET localhost:9200/website/_search?pretty
{
    "query" : {
        "filtered" : {
            "filter" : {
                "term" : {
                    "user_id" : 1
```

```
                }
            }
        }
    }
}
```

过滤语句与_score 没有关系，但是有隐含的查询条件 match_all 为所有文档的_score 设值为 1。也就相当于所有的文档相关性是相同的。

3. 字段值排序

下面例子中，对结果集按照时间排序，这也是最常见的情形，将最新的文档排列靠前。我们使用 sort 参数进行排序：

```
curl -X GET localhost:9200/website/_search?pretty
{
    "query" : {
        "filtered" : {
            "filter" : { "term" : { "user_id" : 1 }}
        }
    },
    "sort": { "date": { "order": "desc" }}
}
```

响应：

```
"hits" : {
    "total" :           6,
    "max_score" :       null,
    "hits" : [ {
        "_index" :      "us",
        "_type" :       "tweet",
        "_id" :         "14",
        "_score" :      null, <1>
        "_source" :     {
             "date":    "2014-09-24",
             ...
        },
        "sort" :        [ 1411516800000 ]
    },
    ...
}
```

响应中"max_score"和"_score"表明_score 字段没有经过计算，因为它没有用作排序。"sort"表明 date 字段被转为毫秒当作排序依据。

首先，在每个结果中增加了一个 sort 字段，它所包含的值是用来排序的。在这个例子当中 date 字段在内部被转为毫秒，即长整型数字 1411516800000 等同于日期字符串 2014-09-24 00:00:00 UTC。

其次就是_score 和 max_score 字段都为 null。计算_score 是比较消耗性能的，而且通常主要用作排序，不是用相关性进行排序的时候，就不需要统计其相关性。如果强制计算其相关性，可以设置 track_scores 为 true。

4. 默认排序

作为缩写，你可以只指定要排序的字段名称：

```
"sort": "number_of_children"
```

字段值默认以顺序排列，而_score 默认以倒序排列。

5. 多级排序

如果合并一个查询语句，并且展示所有匹配的结果集，使用第一排序是 date，第二排序是_score，请求如下：

```
curl -X GET localhost:9200/website/_search?pretty
{
    "query" : {
        "filtered" : {
            "query":   { "match": { "tweet": "manage text search" }},
            "filter" : { "term" : { "user_id" : 2 }}
        }
    },
    "sort": [
        { "date":   { "order": "dESc" }},
        { "_score": { "order": "dESc" }}
    ]
}
```

结果集会先用第一排序字段来排序，当用作第一字段排序的值相同的时候，然后再用第二字段对第一排序值相同的文档进行排序，以此类推。

多级排序不需要包含_score，可以使用几个不同的字段，如位置距离或者自定义数值。

6. 字符串参数排序

字符查询也支持自定义排序，在查询字符串使用 sort 参数就可以：

```
curl -X GET localhost:9200/website/_search?sort=date:dESc&sort=_score&q=search
```

7. 为多值字段排序

一个拥有多值的字段就是一个集合，在为一个多值字段进行排序的时候，其实这些值本来是没有固定的顺序的。

对于数字和日期，你可以从多个值中取出一个来进行排序，支持使用 min、max、avg 或 sum 这些模式。比如可以在 dates 字段中用最早的日期来进行排序：

```
"sort": {
    "dates": {
        "order": "asc",
        "mode":  "min"
    }
}
```

5 最佳实践

5.1 Spark读写Elasticsearch数据

Spark 可以从 Elasticsearch 中读取数据，并运用 Elasticsearch 提供的过滤功能，在数据源头对数据进行过滤，可以避免将所有数据都加载到内存。

5.1.1 依赖包

在 pom 文件中添加以下依赖：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.1.2</version>
</dependency>
<dependency>
    <groupId>org.Elasticsearch</groupId>
    <artifactId>Elasticsearch-hadoop</artifactId>
    <version>7.10.0</version>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.0.0-cdh6.2.0</version>
</dependency>
</dependency>
```

5.1.2 完整代码

完整代码如下：

```
import org.apache.spark.{SparkConf, SparkContext}
import  org.Elasticsearch.spark._

object EsTest {
  def main(args: Array[String]): Unit = {

    case class Course(name: String, credit: Int)

    val conf = new SparkConf().setAppName(this.getClass.getSimpleName).setMaster("local")
    conf.set("es.nodes", "101.8.147.101")
    conf.set("es.port", "9200")
    conf.set("es.index.auto.create", "true")
    val sc = new SparkContext(conf)

    //rdd 写 es
```

```
    val courseRdd = sc.makeRDD(Seq(Course("Hadoop", 4), Course("Spark", 3), Course("Kafka",
2)))
    courseRdd.saveToEs("/course/rdd")

    //rdd 读 es
    val esCourseRdd = sc.esRDD("/course/rdd")
    esCourseRdd.foreach(c => {
      println(c.toString())
    })

  }
}
```

运行程序之后，数据可写入到 Elasticsearch 集群，在 Web 页面可查看写入的数据。

6 常见问题解答

6.1 调优

6.1.1 优化索引设计

- 如果查询有一个过滤字段并且它的值是可枚举的，那么把数据分成多个索引。
- 如果查询具有过滤字段并且其值不可枚举，请使用路由。可以通过使用过滤字段值作为路由键来将索引拆分成多个分片，然后删除过滤条件。
- 如果查询条件包含日期范围，则按日期对数据进行分组。这适用于大多数日志记录或监控场景。可以以每天、每周或每月分组索引，然后可以在指定的日期范围内获得索引列表。Elasticsearch只需要查询一个较小的数据集而不是整个数据集。此外，当数据过期时，很容易缩小/删除旧的索引。
- 明确地设置映射。Elasticsearch 可以动态地创建映射，但并不适用于所有场景。例如，Elasticsearch 中默认的字符串字段映射是“关键字”和“文本”类型，这在很多场景下是没有必要的。
- 如果文档使用用户定义的 ID 或路由索引，请避免不平衡分片。Elasticsearch 采用随机 ID 生成器和哈希算法来确保文档均匀地分配给分片。当使用用户定义的 ID 或路由时，ID 或路由键可能不够随机，并且一些分片可能明显比其它分片更大。在这种情况下，在这个分片上的读/写操作将比在其它分片上慢得多。可以优化 ID 或路由键。
- 使分片均匀分布在节点上。如果一个节点比其它节点有更多的分片，则会比其它节点承担更多的负载，并很有可能成为整个系统的瓶颈。

6.1.2 调优索引性能

- 使用批量请求。
- 使用多个线程/工作来发送请求。
- 增加刷新间隔。每次刷新事件发生时，Elasticsearch 都会创建一个新的 Lucene 段，并在稍后进行合并。增加刷新间隔将降低创建/合并的成本。请注意，只有在刷新事件发生后才能进行文件搜索。
- 减少副本数量。Elasticsearch 需要为每个索引请求将文档写入主要和所有副本分片。显然，一个大的副本数会减慢索引速度，但另一方面，增加副本数量将提高搜索性能。副本的作用一是提高系统的容错性，当某个节点某个分片损坏或丢失时可以从副本中恢复；二是提高 Elasticsearch 的查询效率，Elasticsearch 会自动对搜索请求进行负载均衡。
- 如果可能，使用自动生成的 ID。Elasticsearch 自动生成的 ID 可以确保是唯一的，以避免版本查询。如果客户真的需要使用自定义的 ID，建议选择一个对 Lucene 友好的 ID，比如零填充顺序 ID，UUID-1 或者 Nano time。这些 ID 具有一致的顺序模式，压缩良好。相比之下，像 UUID-4 这样的 ID 本质上仍旧是随机的，它提供了较差的压缩比，并降低了 Lucene 的速度。

6.1.3 调优搜索性能

- 一般情况下最好使用过滤语境而不是查询语境。一个查询子句用于回答“这个文档如何与查询子句匹配？”，过滤子句用于回答“这个文档是否匹配这个过滤子句？”。Elasticsearch 只需要回答“是”或“否”。它不需要计算过滤子句的相关性得分，并且可以高速缓存过滤结果。
- 增加刷新间隔。正如在调优索引性能部分所提到的，Elasticsearch 每次刷新时都会创建一个新的段。增加刷新间隔将有助于减少段数并降低搜索的 I/O 成本。而且一旦发生刷新并且数据改变，缓存将无效。增加刷新间隔可以使 Elasticsearch 更高效地利用缓存。
- 增加副本数量：Elasticsearch 可以在主分片或副本分片上执行搜索。拥有的副本越多，搜索中涉及的节点就越多。
- 尝试不同的分片数量。太小的分片数量会使搜索无法扩展。例如，如果分片数量设置为 1，则索引中的所有文档都将存储在一个分片中。如果文件很多，查询将耗费大量时间。另一方面，创建索引的分片太多也会对性能造成危害，因为 Elasticsearch 需要在所有分片上运行查询，除非在请求中指定了路由键，然后将所有返回的结果一起取出并合并。
- 根据经验，如果索引小于 1G，可以将分片数设置为 1。对于大多数情况，可以将分片数保留为默认值 5，但是如果分片大小超过 30GB，应该增加分片数量将索引分成更多的分片。
- 节点查询缓存。节点查询缓存只缓存正在过滤语境中使用的查询。与查询子句不同，过滤子句是“是”或“否”的问题。Elasticsearch 使用一个位设置机制来缓存过滤结果，以便后面的查询使用相同的过滤条件进行加速。请注意，只有保存超过 10,000 个文档的分段（或文档总数的 3%，以较大者为准）才能启用节点查询缓存。
- 分片查询缓存。如果大多数查询是聚合查询，应该使用分片查询缓存，它可以缓存聚合结果，以便 Elasticsearch 直接以低成本提供请求。
- 仅检索必要的字段。如果文档很大，并且只需要几个字段，请使用 stored_fields 检索所需要的字段而不是所有字段。
- 避免搜索停用词。诸如“a”和“the”这样的停用词可能导致查询命中结果计数爆炸。设想有一百万个文件，搜索“fox”可能会返回几十个结果，但搜索“the fox”可能会返回索引中的所有文件，因为“the”出现在几乎所有的文件中。

  Elasticsearch 需要对所有命中的结果进行评分和排序，导致像“the fox”这样的查询减慢整个系统。可以使用停止标记过滤来删除停用词，或使用“和”运算符将查询从“the fox”更改为“the AND fox”，以获得更精确的结果。如果某些词在索引中经常使用，但不在默认停用词列表中，则可以使用截止频率来动态处理它们。
- 如果不关心文档返回的顺序，则按_doc 排序。Elasticsearch 使用“_score”字段按默认分数排序。如果不关心顺序，可以使用“sort”：“_doc”让 Elasticsearch 按索引顺序返回。
- 避免使用脚本查询来计算不固定的匹配。在索引时存储计算的字段。例如，有一个包含大量用户信息的索引，需要查询以“1234”开头的所有用户。或许想运行一个脚本查询，如“source”：“doc ['num'].value.startsWith（'1234'）。”这个查询是非常耗费资源的，并且减慢整个系统。索引时考虑添加一个名为“num_prefix”的字段，然后只需要查询“name_prefix”：“1234”。
- 避免通配符查询。

6.2 运维类问题

6.2.1 日志查看方法

查看 Elasticsearch 的日志信息，有两种方式：

(1) 通过日志管理查看。

(2) 登录 Elasticsearch 集群后台，直接查看日志信息。

1. 通过日志管理查看

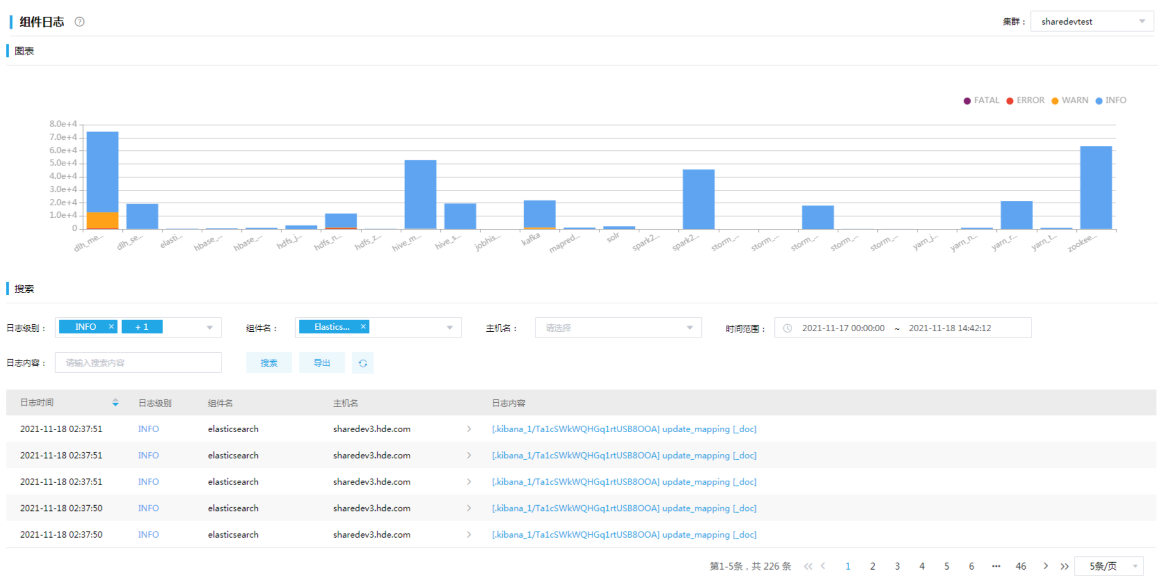
组件日志页面展示指定集群中产生的所有组件日志信息，支持通过日志级别、组件名、主机名、时间范围等筛选查询日志。

(1) 在集群管理的左侧导航树中选择[日志管理/组件日志]，进入组件日志页面。

(2) 组件日志以图表和日志列表形式展示系统中的组件日志。

- 图表：通过分析日志列表中日志的级别和数量等信息，以柱状图的形式展示不同组件产生的不同级别的日志数量。
- 日志列表：展示系统内指定集群的组件日志列表。展示信息包括日志产生的时间、日志级别、产生日志的组件和主机、日志内容。单击日志内容，可查看日志的详细信息。

图6-1 组件日志页面



2. 登录 Elasticsearch 集群后台，直接查看日志

Elasticsearch 集群日志默认路径在/var/de_log/elasticsearch。日志路径信息，可以在大数据平台 Elasticsearch 配置页面中，通过搜索 path.logs 来查看。

3. 日志查看注意点

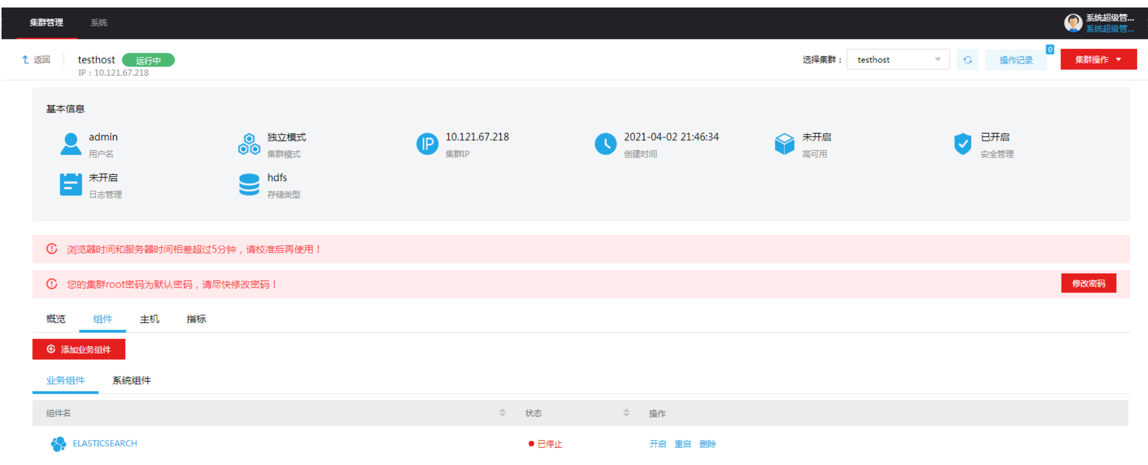
先通过大数据平台管理页面找出有问题的 Elasticsearch 节点，查看对应节点日志，在日志中寻找报错信息（搜索关键字：Exception、ERROR），根据报错信息分析出错原因。

对于现场无法定位问题，需要将问题节点日志发给技术服务人员进行问题定位解决。

6.2.2 组件停止

如果在集群详情页，选择[组件]页签，看到组件列表中 Elasticsearch 的状态为已停止，则表示 Elasticsearch 已经停止运行，如图 6-2 所示。

图6-2 组件停止



造成组件停止的原因可能有以下几个：

- 端口号被占用

  在 linux shell 终端输入 netstat -anp | grep 9200，查看端口是否被别的进程占用，如果被占用杀掉该进程的进程号，然后重启组件。

- 主机内存不足

  登录组件所在主机，执行 free -h 查看主机的内存使用情况，当主机内存被耗尽或不足以满足 Elasticsearch 服务启动，则会导致组件停止。

  Elasticsearc 所需内存可在组件配置页通过搜索 instance.memory 查看（单位为 GB），如果当前机器可用内存小于 Elasticsearch 所需内存，则可会导致组件停止。

- 主机故障

  组件所在的主机发生故障，在主机页面查看 Elasticsearch 组件所在的主机是否正常。

- 其他原因

  登录出现问题的 Elasticsearch 节点，根据日志文件的报错信息，进行问题定位。

6.2.3 脑裂问题

所谓脑裂问题，就是同一个集群中的不同节点，对于集群的状态有了不一样的理解。正常情况下，集群中的所有节点应该对集群中 Master 的选择是一致的，当集群中的节点对 Master 节点的选择出现不一致时就是产生了脑裂问题。脑裂状态将导致集群不能正常工作。

当通过以下命令查看集群状态，发现集群的总体状态是 red，在结果中显示的节点数量与实际不一致。并且将请求发向不同的节点之后，可用的节点数量不一致，此时就是出现了脑裂问题。

```
curl -XGET 'Elasticsearch_ip:9200/_cluster/health'
```

1. 产生原因

可能导致脑裂的原因：

- 网络：由于网络问题造成某些节点认为 Master 死掉，从新选择了一个 Master。网络正常之后，原来的 Master 节点重新加入集群，导致出现两个 Maser。
- 节点负载：当 Master 节点与 Data 节点混在一起时，可能会因为工作节点的负载较大导致 Master 停止响应，此时一部分节点就会认为这个 Master 节点失效了，故重新选举新的 Master，这时就出现了脑裂。

2. 解决方法

当集群中出现脑裂时，可以重启整个 Elasticsearch 集群，让集群中的节点重新选举出一个 Master。

3. 预防措施

针对独立服务中的 Elasticsearch 集群，可以通过配置专有 master 实例，实现 master 实例和 Data 实例分离、专有 master 实例个数为奇数且至少 3 个的方式来进行预防。

目　录

1 组件简介······1-1
1.1 组件概述······1-1
1.2 组件架构······1-1
1.3 应用场景······1-2
2 快速入门······2-1
2.1 组件安装······2-1
2.1.1 数据目录检查······2-1
2.1.2 查看组件的日志信息······2-2
2.2 运行状态监控······2-2
2.3 快速使用指导······2-4
2.3.1 非 Kerberos 环境操作示例······2-5
2.3.2 Kerberos 环境操作示例······2-6
2.3.3 Kerberos 环境下的用户身份认证······2-7
2.4 快速链接······2-9
2.4.1 配置组件快速链接······2-9
2.4.2 访问 Kafka 消息监控系统······2-9
3 使用指南······3-1
3.1 常用命令······3-1
3.2 Client 下载/安装/使用/卸载······3-1
3.2.1 下载 Client 安装包······3-1
3.2.2 安装 Client······3-2
3.2.3 访问组件······3-3
3.2.4 卸载 Client 客户端······3-4
3.3 权限访问控制······3-4
3.3.1 权限说明······3-4
3.3.2 权限使用操作示例······3-5
3.4 Kafka 集群扩容······3-6
3.4.1 使用场景······3-6
3.4.2 扩容前准备······3-6
3.4.3 扩容约束······3-6
3.4.4 扩容影响······3-7
3.4.5 扩容操作指导······3-7

3.4.6 扩容验证 3-8
3.5 Kafka 集群缩容......3-8
3.5.1 使用场景 3-8
3.5.2 缩容前准备......3-9
3.5.3 缩容约束 3-9
3.5.4 缩容影响 3-9
3.5.5 缩容操作指导 3-9
3.5.6 缩容验证 3-10
3.6 租户管理......3-10
3.6.1 租户介绍 3-11
3.6.2 新增租户 3-11
3.6.3 租户使用操作示例......3-12
3.7 备份恢复......3-13
3.7.1 新建 Kafka 同步任务 3-14
3.7.2 源集群和目的集群配置互信 3-15
3.7.3 Kafka 同步任务相关配置......3-17
3.7.4 Kafka 备份恢复示例......3-18
3.8 Kafka API 使用指南......3-19
4 最佳实践......4-1
4.1 关键参数说明......4-1
4.2 Kakfa producer 实践案例 4-1
4.3 Kakfa consumer 实践案例......4-3
4.4 Flink-Kafka 实践案例 4-5
4.5 SparkStreaming-Kafka 实践案例......4-9
5 常见问题解答 5-1
5.1 性能调优......5-1
5.2 运维类问题 5-2

1 组件简介

1.1 组件概述

Kafka 是一个分布式流平台（不只是消息系统），具备以下关键特性：

- 发布-订阅消息流（类似于消息队列或企业消息系统）
- 同时支持离线数据处理和实时数据处理
- 以容错持久的方式存储消息流
- 支持在线水平扩展、高吞吐
- 提供 At-Least Once，At-Most Once，Exactly Once 消息可靠传递

Kafka 支持两类应用程序：

- 构建实时流数据管道，能够可靠地在系统之间获取数据
- 构建实时流应用程序，能够对数据流进行转换或响应

Kafka 应用特性如下：

- Kafka 可以运行在一个或多个服务器上，跨越多个数据中心
- Kafka 将消息流存储在 topic 中
- Kafka 消息由键（key）、值（value）和时间戳（timestamp）组成

Kafka 支持 topic 级别性能监控如下：

- topic 输入的字节流量
- topic 输出的字节流量
- topic 每秒失败的 fetch 请求数
- topic 每秒失败的 Produce 请求数
- topic 每秒输入的消息条数
- topic 每秒的 fetch 请求数
- topic 每秒的 produce 请求数

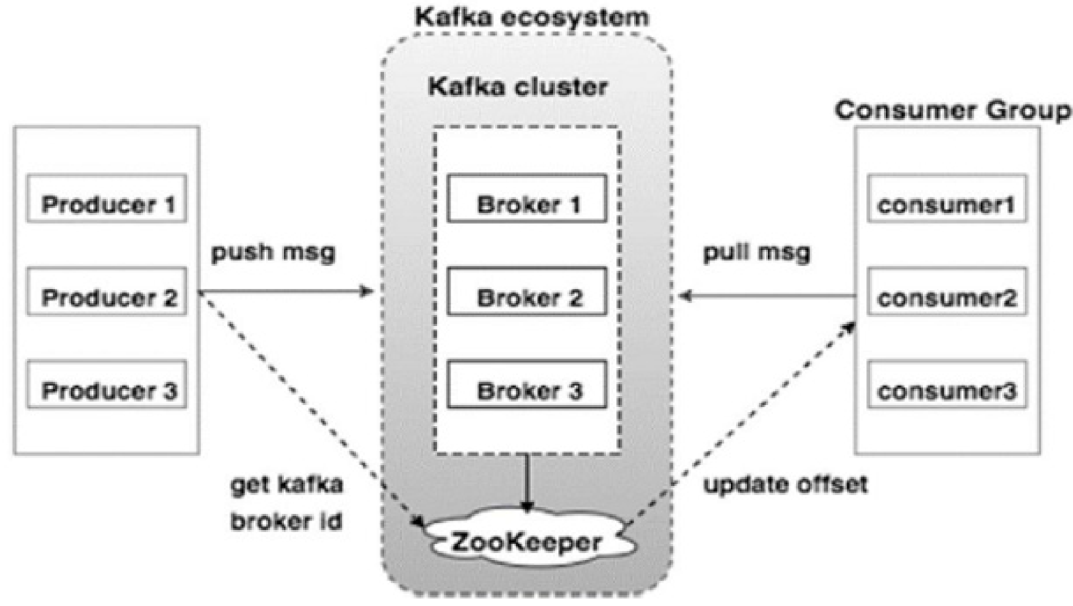
1.2 组件架构

Kafka 依赖 Zookeeper，Zookeeper 负责维护 Kafka 的元数据（broker、topic、partition）等信息，实现 Kafka 的动态管理。如图 1-1 所示，Kafka 架构主要由 Broker、Producer、Consumer 以及 Consumer Group 组成。Kafka Broker 上存在多个 topic，Producer 采用 push 的方式将消息发布到 topic，Consumer 对感兴趣的 Topic 进行订阅并采用 pull 的方式消费消息。对于一个 topic，可以分为多个 Partition，分区的每条消息都有一个偏移顺序（Offset）序列化编号，可唯一确定消息在 Partition 的位置。其中：

- Broker：Kafka 包含的一个或多个服务器
- topic：消息的一种逻辑分组，用于对消息分门别类，每一类消息称之为一个 topic
- Partition：消息的一种物理分组，一个主题被拆成多个分区，每一个分区是一个有序的、不可变的消息队列

- Producer：负责发布消息到 Kafka Broker
- Consumer：从 Kafka Broker 读取消息的客户端
- Consumer Group：每个 Consumer 属于一个特定的 Consumer Group

图1-1 Kafka 架构



1.3 应用场景

- 替换传统消息系统

  相比于传统的消息系统，Kafka 具备更好的吞吐量、低延迟、分区、副本、容错等特性，有利于处理大规模的消息。

- 网站活动追踪

  Kafka 可以用来记录用户的各种活动，例如网页浏览、搜索、点击等活动，这些活动被发布到 Kafka 的 topic 中，可用于实时监控处理、实时监测或加载到 Hadoop 或离线数据仓库中。

- 指标监控

  Kafka 可以用于实时监控性能指标等数据。

- 日志聚合

  日志聚合通常从服务器收集物理日志文件，并将它们放在日志收集中心（可以是文件服务器或 HDFS）进行处理。Kafka 抽象出文件的细节，将日志或事件数据更清晰地抽象为消息流，能够保证更低延迟的处理，也更容易支持多个生产者和消费者。

- 流处理

  流处理通常包含多个阶段，可以通过 Kafka 进行中转。例如新闻推荐场景中，新闻内容可以从“articles”主题获取，经过进一步处理得到新内容后再推荐给用户。

- 提交日志

  分布式系统可以提交日志到 Kafka，以帮助节点同步数据。

2 快速入门

2.1 组件安装

说明

- 在 Hadoop 集群或 Kafka 集群中，安装 Kafka 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- 部署 Kafka 时，根据实际生产环境的数据量，需要调整 Kafka Broker 节点数量。
- 在当前版本中，支持部署 Kafka 专有实例，即在 Kafka 专有实例上仅能部署 Kafka。

大数据集群中，部署 Kafka 包括以下两种方式：

- 在集群类型为 Hadoop 的大数据集群中安装 Kafka，此时在集群中同时还能部署其他大数据组件。
- 在集群类型为 Kafka 的大数据集群中安装 Kafka，此时在集群中仅能部署 Kafka 及其依赖的组件(Zookeeper)。

2.1.1 数据目录检查

根据现场磁盘分区方案和挂盘方案的不同，组件安装完成后，必须对组件的数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表 2-1 所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
Kafka	是（配置项的参数值默认使用全部挂载路径）	log.dirs	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录
Zookeeper	是（配置项的参数值默认使用某一个挂载路径）	dataDir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录

2.1.2 查看组件的日志信息

表2-2 组件日志路径说明

组件	日志路径
Kafka	/var/de_log/kakfa/user_${user.name}/，其中${user.name}是指执行任务的用户名
ZooKeeper	/var/de_log/zookeeper/user_{user_name}/，其中${user.name}是指执行任务的用户名

2.2 运行状态监控

1. 查看组件详情

进入 Kafka 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、概要、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：进程部署的个数、启动状态等，以便于快速了解组件。
- 概要：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 组件详情

2. 组件检查

执行 Kafka 组件检查时，会创建名称为 ambari_kafka_service_check 的 topic，组件检查成功表示 Kafka 组件可正常使用。

集群在使用过程中，根据实际需要，可对 Kafka 组件执行组件检查的操作。

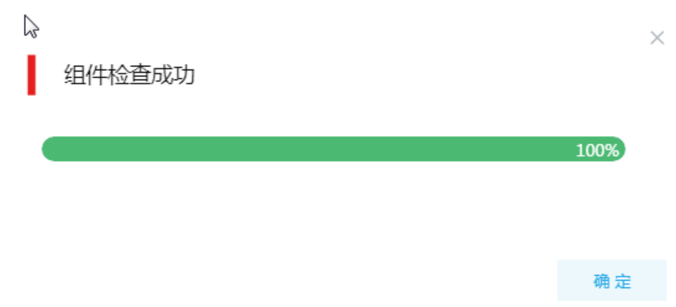
(1) 组件检查的方式有以下三种，任选其一即可：

- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Kafka 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Kafka 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

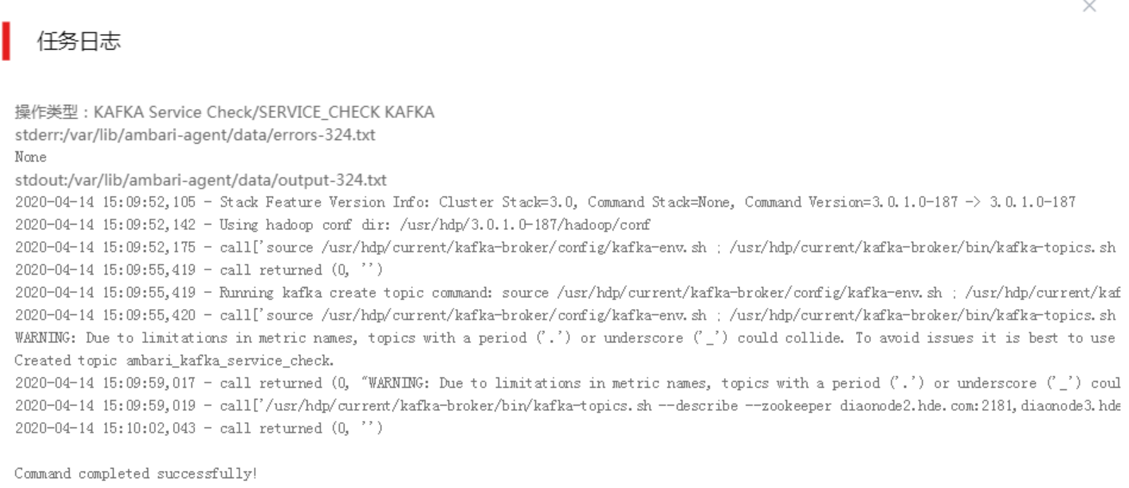
(3) 组件检查结束后，检查窗口中会显示组件检查成功或失败的状态，如图 2-2 所示，表示该组件检查成功，可正常使用。

图2-2 组件检查



(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作历史窗口。可查看“Kafka Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情



2.3 快速使用指导

注意

根据大数据集群是否开启Kerberos认证，用户访问Kafka组件时的认证方式不同，详情请参见本章节内容。

Kafka组件既可以通过集群用户访问，又可以通过组件超级用户访问。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
  - 集群超级用户：仅Hadoop集群拥有集群超级用户。新建Hadoop集群成功后，集群超级用户自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。

- 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。

- 组件超级用户：指组件内部的最高权限用户，如 Kafka 组件的 kafka 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 Kerberos 环境操作示例

说明

- 非 Kerberos 环境下，不需要用户做身份认证即可直接对 Kafka 执行管理操作。
- 本章节操作需要切换至拥有 Kafka 相应操作权限的用户，例如 root、kafka 用户。

以下示例场景为：在某一集群节点（例如 node1）上，同时安装了 Zookeeper 和 Kafka。通过控制台，进行 topic 的创建、生产和消费等操作。

1. 创建 topic

创建一个名为 test 的 topic，其分区数量和副本数量均为 3，执行如下命令：

```
./kafka-topics.sh --zookeeper node1:2181 --create --partitions 3 --replication-factor 3 --topic test
```

其中：

- --zookeeper：Kafka 集群中使用的 ZooKeeper，端口默认为 2181，多个 ZooKeeper 节点可以用逗号分割
- --create：创建 topic 的固定参数
- --partitions：该 topic 的分区数量
- --replication-factor：该参数为副本因子，用于指定 topic 的副本数，在生产环境中建议 3 副本
- --topic：创建的 topic 名称

2. 生产数据

向 test 中生产数据，执行如下命令：

```
./kafka-console-producer.sh --broker-list node1:6667 --topic test
```

命令执行后，在控制台交互界面中输入测试数据即可。

其中：

- --broker-list：Kafka 集群中 Kafka Broker 地址，端口号默认为 6667，多个 Kafka Broker 节点用逗号分割
- --topic：保存数据的 topic 名称

3. 消费数据

消费 test 中的数据，执行如下命令：

```
./kafka-console-consumer.sh --bootstrap-server node1:6667 --topic test --from-beginning
```

命令执行完成后，控制台会打印出 topic 中存储的数据。

其中：

- --bootstrap-server：Kafka集群中Kafka Broker地址，端口号默认为6667，多个Kafka Broker节点用逗号分割
- --topic：保存数据的 topic 名称
- --from-beginning：消费模式，从头开始消费该 topic 中数据

2.3.2 Kerberos 环境操作示例

注意

- 若集群开启 Kerberos，使用 Kafka 组件生产或消费数据前需要先进行用户身份认证。关于用户身份认证，详情请参见 2.3.3 Kerberos 环境下的用户身份认证。
- Kerberos 环境下只有 kafka 用户可以执行创建 topic 操作。
- 本章节操作需要切换至拥有 Kafka 相应操作权限的用户，例如 root、kafka 用户。

以下示例场景为：在某一集群节点（例如 node1）上，同时安装了 Zookeeper 和 Kafka。通过控制台，进行 topic 的创建、生产和消费等操作。

1. 创建 topic

创建一个名为 test 的 topic，其分区数量和副本数量均为 3，执行如下命令：

```
./kafka-topics.sh --zookeeper node1:2181 --create --partitions 3 --replication-factor 3 --topic test
```

其中：

- --zookeeper：Kafka 集群中使用的 ZooKeeper，端口默认为 2181，多个 ZooKeeper 节点可以用逗号分割
- --create：创建 topic 的固定参数
- --partitions：该 topic 的分区数量
- --replication-factor：该参数为副本因子，用于指定 topic 的副本数，在生产环境中建议 3 副本
- --topic：将要创建的 topic 名称

2. 生产数据

(1) 需要进行用户 kafka/node1.hde.com 的身份认证。关于用户身份认证，详情请参见 2.3.3 Kerberos 环境下的用户身份认证。

(2) 向 test 生产数据，执行如下命令：

```
./kafka-console-producer.sh --broker-list node1:6667 --topic test --producer.config producer.txt
```

命令执行后，在控制台交互界面中输入测试数据即可。

其中：

- --broker-list：Kafka 集群中 Kafka broker 地址，端口默认为 6667，多个 broker 节点用逗号分割
- --topic：保存数据的 topic 名称
- producer.txt 文件：需用户提前创建，文件名可自定义，文件内容为：
security.protocol=SASL_PLAINTEXT

3. 消费数据

(1) 进行用户 kafka/node1.hde.com 的身份认证。关于用户身份认证，详情请参见 2.3.3 Kerberos 环境下的用户身份认证。

(2) 消费 test 中的数据，执行如下命令：

```
./kafka-console-consumer.sh --bootstrap-server node1:6667 --topic kafkatopic --from-beginning
--consumer.config consumer.txt
```

命令执行完成后，控制台会打印出 topic 中的所有数据。

其中：

- --bootstrap-server：Kafka 集群中 Kafka Broker 地址，端口号默认为 6667，多个 Kafka broker 节点用逗号分割
- --topic：保存数据的 topic 名称
- consumer.txt 文件：需用户提前创建，文件名可自定义，文件内容为：security.protocol=SASL_PLAINTEXT
- --from-beginning：消费模式，从头开始消费该 topic 中数据

2.3.3 Kerberos 环境下的用户身份认证

如果大数据集群开启 Kerberos，若想操作 Kafka，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件超级用户身份认证

1. 集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- 集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

Kafka 组件还可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

- **方式一（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**

  a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的 /etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：

  ```
  chown user1 /etc/security/keytabs/user1.keytab
  ```

  b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：

  ```
  klist -k user1.keytab
  ```

  【说明】如图 2-4 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-4 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

su user1

kinit -kt user1.keytab user1@TENANTC.COM

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

- **方式二（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis***t 命令可查看认证结果。

图2-5 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

2. 组件超级用户身份认证

可以通过组件超级用户访问 Kafka 组件，比如 kafka 用户，在开启 Kerberos 环境的大数据集群中进行组件超级用户（示例用户为 kafka）的身份认证，操作如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 kafka 的认证文件“kafka.service.keytab”。

【说明】在 Kafka Client 节点上，需要将 Kafka 的认证文件“kafka.service.keytab”上传至节点的任意目录下，将 keytab 文件的所有者修改为 kafka，然后切换至 kafka 用户下进行认证，其中修改所有者命令为：

chown kafka kafka.service.keytab

(2) 使用 ***klist*** 命令查看 kafka.service.keytab 文件的 principal 名称，命令如下：

klist -k kafka.service.keytab

如图 2-6 所示，红框内容即为 kafka.service.keytab 文件的 principal 名称。

图2-6 认证文件的 principal 名称

```
[root@hppnode1 keytabs]# klist -k kafka.service.keytab
Keytab name: FILE:kafka.service.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 kafka/hppnode1.hde.com@TESTSHARE.COM
   2 kafka/hppnode1.hde.com@TESTSHARE.COM
   2 kafka/hppnode1.hde.com@TESTSHARE.COM
   2 kafka/hppnode1.hde.com@TESTSHARE.COM
   2 kafka/hppnode1.hde.com@TESTSHARE.COM
```

(3) 执行用户身份验证，命令如下：

su kafka

kinit -kt kafka.service.keytab kafka/hppnode1.hde.com@TESTSHARE.COM

(4) 输入 ***klist*** 命令可查看认证结果。

2.4 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。

(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。

(3) 在本地 hosts 文件中配置主机域名信息完成后，此时即可访问组件的快速链接。

2.4.2 访问 Kafka 消息监控系统

Kafka 提供消息监控系统页面，支持在线查看 Topics、Kafka Broker 等相关信息。

(1) 如图 2-7 所示，在 Kafka 组件详情页面右上角[快速链接]的下拉框中，可以获取 Kafka 消息监控系统的访问入口消息。

图2-7 Kafka 快速链接

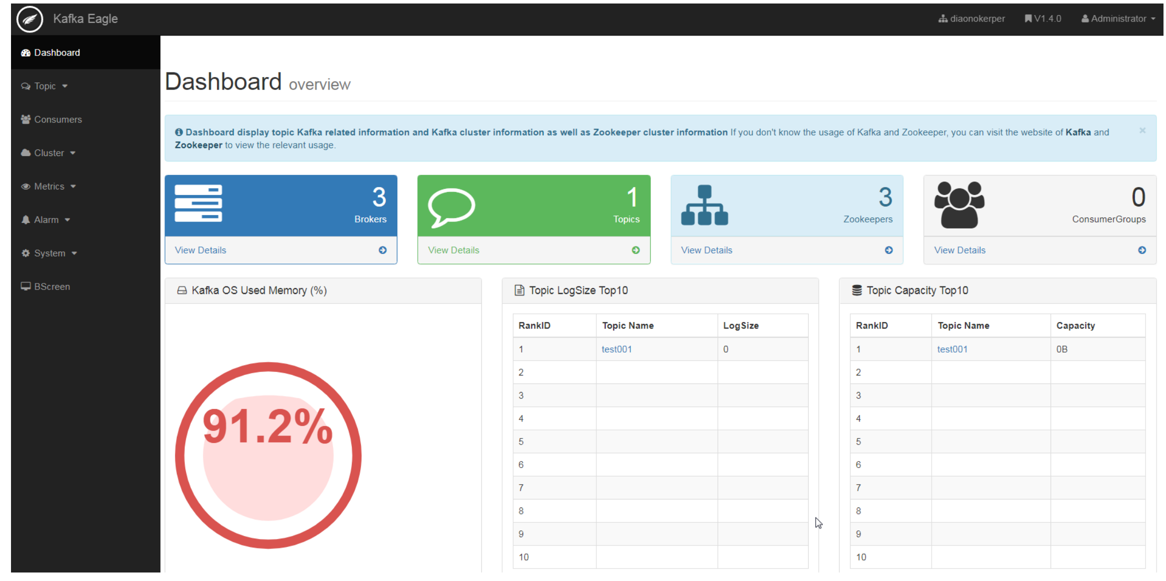
(2) 访问 Kafka 快速链接时，无论集群是否开启 Kerberos，均只能通过用户名（admin）和密码（CloudOS5#DE3@KE）进行登录，其他用户名/密码均无法访问。

(3) 在 Kafka 消息监控系统页面，可查看以下信息：

- Dashboard 页面

  如图 2-8 所示，在 Dashboard 页面可以查看 Kafka 组件中的 Topics 数量、消费者数量、Kafka 的 Brokers 数以及所属 Zookeeper 集群的相关信息。

图2-8 Dashboard 页面

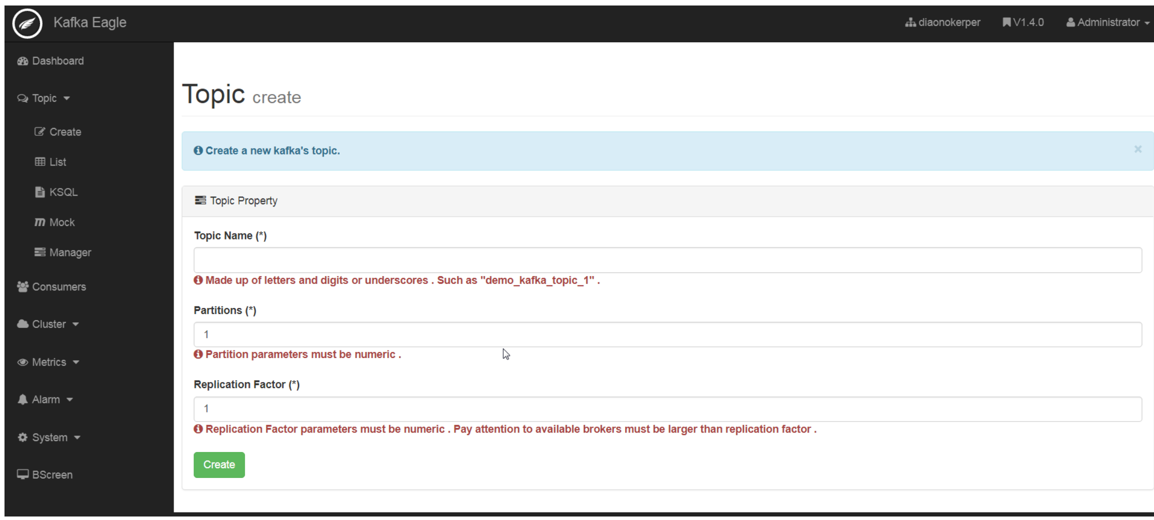


- Topic 页面

  如图 2-9 所示，在 Topic 页面可以查看 Create、List、KSQL、Mock、Manager 的相关信息，其中：

  - Create：可创建自定义分区和备份数的 topic。
  - List：展示 topics 列表，包含 Topic 的分区数、创建时间以及修改时间，单击每个 topic Name 可跳转至 Topic 详情页面。
  - Manager：支持 SQL 查询。
  - Mock：可用于测试发送消息。

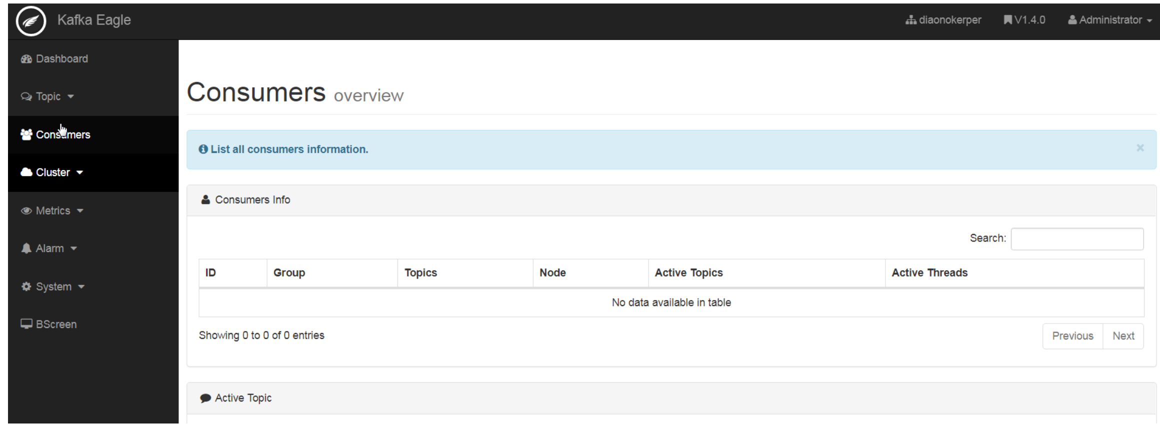
图2-9 Topic 页面



- Consumers 页面

  如图 2-10 所示，在 Consumers 页面可以查看 Consumers Info 及 Active Topic 的相关信息。

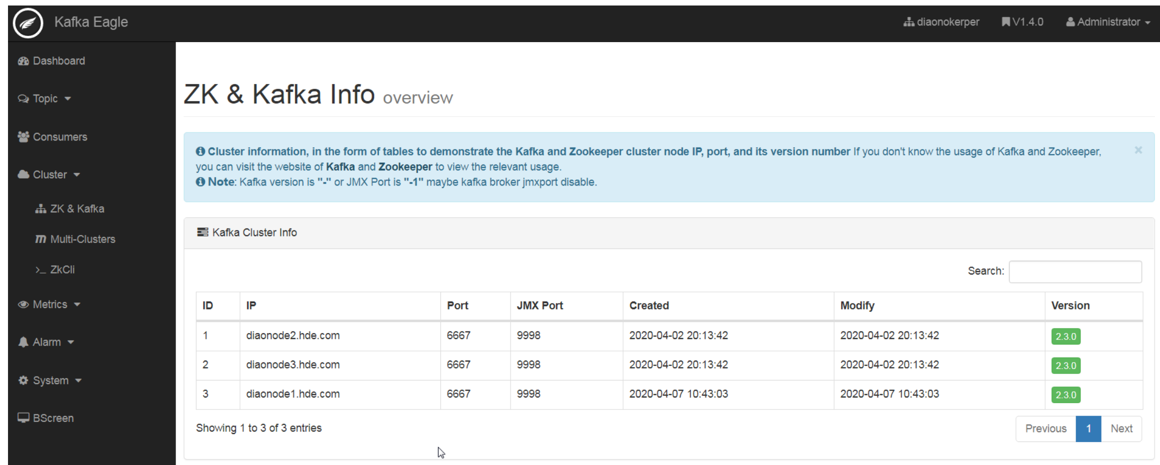
图2-10 Consumers 页面



- Cluster 页面

  如图 2-11 所示，在 Cluster 页面可以查看 ZK&Kafka Info、Multi-Clusters 及 ZkCli，且在 ZkCli 页面可以直接对 Zookeeper 进行相关操作。

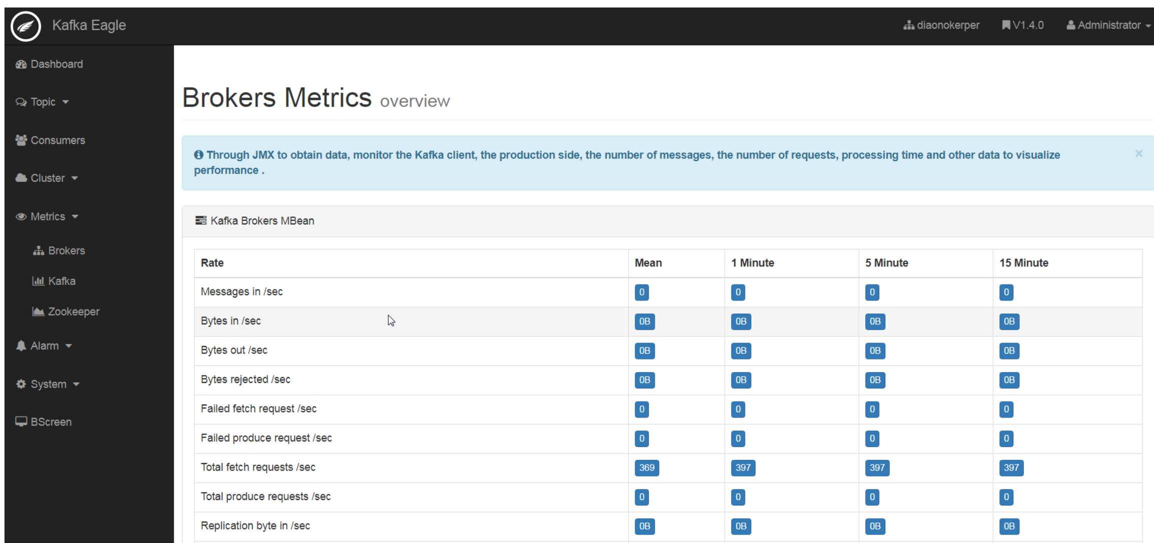
图2-11 Cluster 页面



- Metrics 页面

  如图 2-12 所示，在 Metrics 页面可以查看 Brokers、Zookeeper 的相关信息。

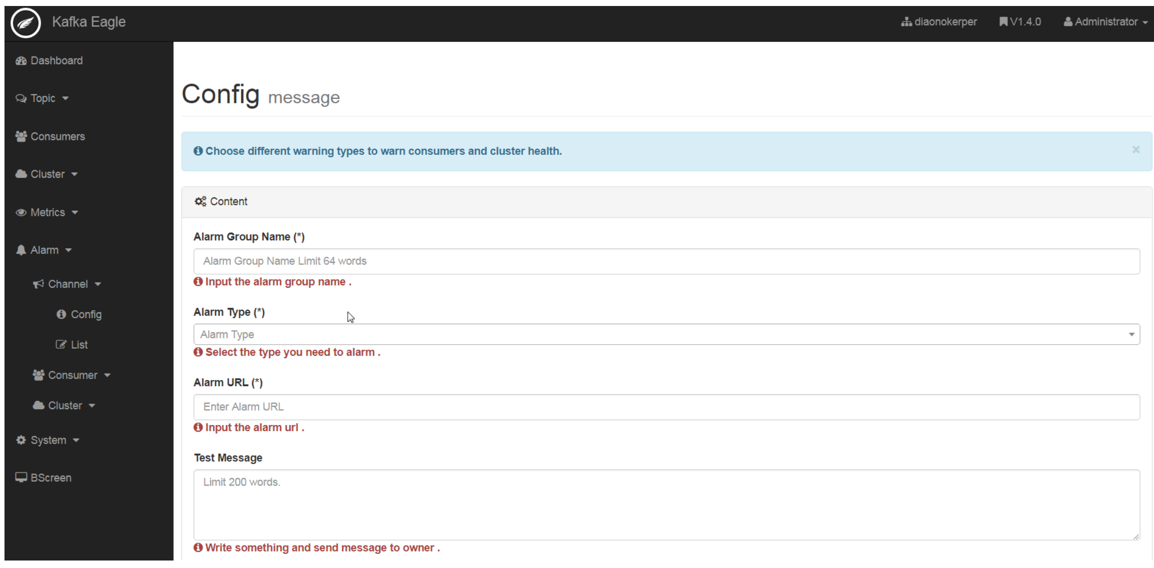
图2-12 Metrics 页面



- Alarm 页面

  如图 2-13 所示，在 Alarm 页面可以查看 Channel、Consumer 以及 Cluster 的相关信息。

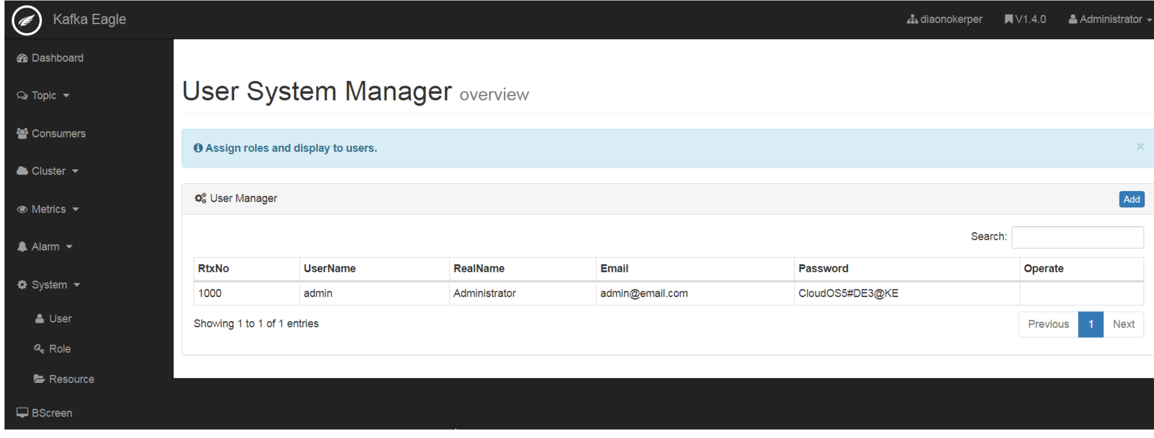
图2-13 Alarm 页面



- System 页面

  如图 2-14 所示，在 System 页面可以查看 User、Role 及 Resource 的相关信息。

图2-14 System 页面



3 使用指南

3.1 常用命令

(1) 创建 topic

./kafka-topics.sh --zookeeper node1:2181 --create --partitions 3 --replication-factor 3 --topic test

(2) 查看所有 topic 列表

./kafka-topics.sh --zookeeper node1:2181 --list

(3) 查看指定 topic 描述信息

./kafka-topics.sh --zookeeper node1:2181 –topic test --describe

(4) 向 topic 生产数据

./kafka-console-producer.sh --broker-list node1:6667 --topic test

(5) 消费 topic 数据

./kafka-console-consumer.sh --bootstrap-server node1:6667 --topic test --from-beginning

3.2 Client下载/安装/使用/卸载

大数据集群提供了下载 Kafka Client 的功能。在客户端节点上安装 Kafka 的 Client 后，即可直接连接创建的集群中的 Kafka，进行组件维护、任务管理等。

3.2.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作，租户的 Kafka 组件也支持下载 Client），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在集群管理的集群列表中，点击集群名称进入集群详情页面，在集群已安装的组件列表中单击 Kafka 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 3-1 所示。

图3-1 下载 Client 窗口

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：

- 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
- 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端，但集群中的组件配置信息或主机相关信息后来被更改的场景下，此时需要更新客户端节点上的配置文件。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。

- 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
- 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

3.2.2 安装 Client

注意

- 安装 Client 的节点必须能与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点上或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端。
- Client 配置文件更新。

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置映射关系，仅非 root 用户需要执行此操作，root 用户可跳过此步骤：

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

./install.sh <安装路径> -o

注意

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为/usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>

3.2.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

  source bigdata_env

- 根据大数据集群是否开启安全管理（即 Kerberos 认证），在集群外的客户端节点上访问组件的方式不同
  - 若集群未开启 Kerberos，则在 Client 节点上，不需要对用户进行校验即可直接访问组件并执行相关操作。
  - 若集群开启了 Kerberos，则在 Client 节点上，必须对用户进行认证之后，才可访问组件并执行相关操作。关于 Kerberos 环境下，进行用户认证的操作方式请参见2.3.3 Kerberos 环境下的用户身份认证。

说明

- 在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。
- 成功连接 Client 客户端后，关于 Client 使用的相关操作请参见 2.3 快速使用指导。

3.2.4 卸载 Client 客户端

大数据集群卸载或重装之后，之前安装的 Client 客户端将不可用，此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 的安装目录下启动卸载。

```
./uninstall.sh
```

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

3.3 权限访问控制

说明

- Kafka 权限管理只有开启 Kerberos 认证后才能支持，仅集群内的 kafka 用户才有权限进行 Kafka topic 创建和删除。
- 仅开启“安全管理/权限与密钥管理”且运行正常的集群可使用角色管理功能。

权限管理是安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

3.3.1 权限说明

在开启权限管理的集群中，用户对 Kafka topic 的操作需被赋予 topic 相关权限后才能执行。

Kafka 支持对主题配置权限（支持使用通配符*模糊适配），权限包括：publish、consume。Kafka 操作所需权限对应关系如表 3-1 所示。

表3-1 Kafka 权限说明

组件	权限类型	对应的组件常用操作
Kafka	publish	执行publish操作
	consume	执行consume操作

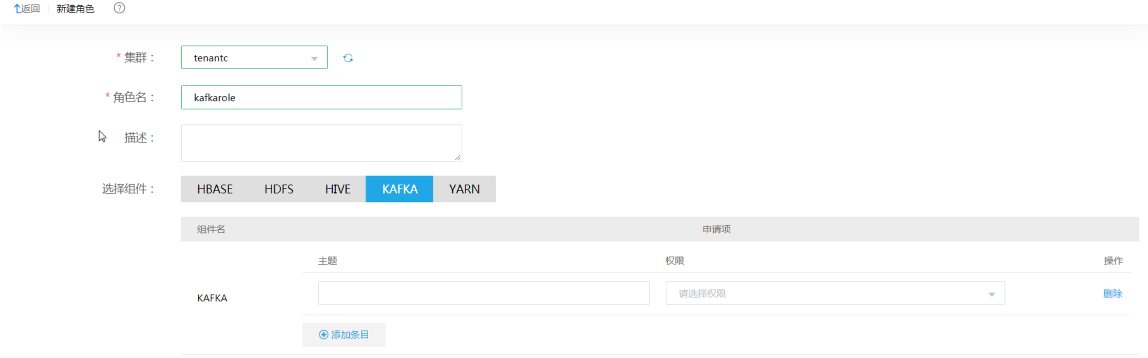
3.3.2 权限使用操作示例

下面以授予“test”主题的“publish”权限给“kafkarole”角色为例，介绍 Kafka 组件的权限访问控制。操作步骤如下：

(1) 新建角色

在[集群权限/角色管理]页面，创建角色 kafkarole，不选择任何组件权限，如图 3-2 所示。

图3-2 新建 kafkarole 角色



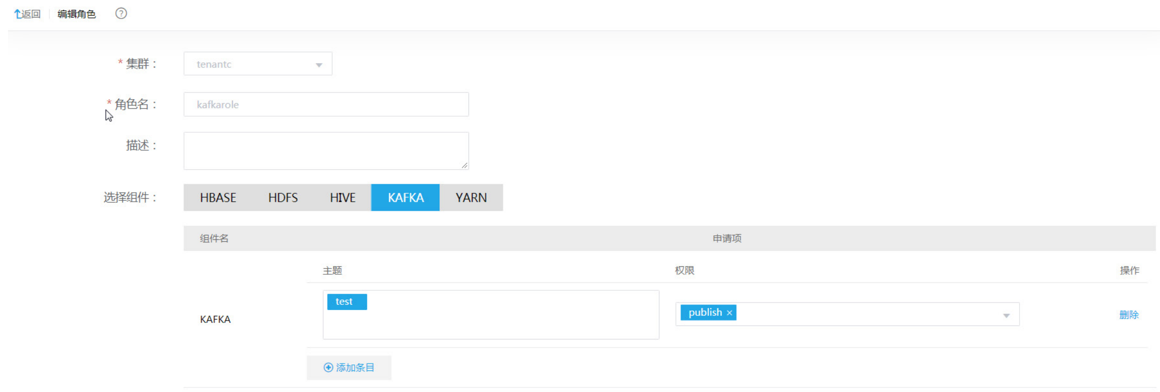
(2) 在[集群权限/用户管理]页面，创建用户 Kafkauser，并为用户授权角色 kafkarole，如图 3-3 所示。

图3-3 对用户授予角色

(3) 使用用户 Kafkauser 执行 ***./kafka-console-producer.sh --broker-list noder124:6667 --topic test --producer.config producer.txt*** 命令，会提示没有发布消息主题为 test 的权限。

(4) 在[集群权限/角色管理]页面，修改角色 kafkarole 权限设置，授予 kafkarole 角色对主题 test 有 publish 消息的权限。

图3-4 角色授权



(5) 使用 Kafkauser 用户再次执行执行 ***./kafka-console-producer.sh --broker-list noder124:6667 --topic test --producer.config producer.txt*** 命令，命令即可执行成功。

3.4 Kafka集群扩容

Kafka 集群扩容是指在某节点上新增安装 Kafka Broker 进程。

3.4.1 使用场景

随着业务量的增加，集群服务能力无法满足用户实际使用需求时，需要考虑对 Kafka 集群进行扩容。

3.4.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在大数据集群上新增安装 Kafka Broker 进程。

- 如果集群中有节点没有安装 Kafka Broker，直接在集群节点中添加 Kafka Broker 进程。
- 如果集群中所有节点均已安装 Kafka Broker，进行 Kafka Broker 扩容前则需要先添加主机。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Kafka 组件的状态是否正常。

(2) 进入 Kafka 组件详情页，查看 Kafka 的部署拓扑，确保每个服务的状态正常，Kafka Broker、Kafka Eagle 处于“已启动”状态。

3.4.3 扩容约束

- 扩容节点操作系统版本与集群内部版本需保持一致。
- 扩容操作一旦开始，不支持中止。

- 已有的 Topic 数据不会自动迁移到扩容节点上，之后新建的 Topic 数据会写到扩容节点上。

3.4.4 扩容影响

- 一旦扩容失败，需要及时将扩容失败的节点剔除。
- 扩容成功后，kafka 集群的读写并发、数据保存周期会得到增强。

3.4.5 扩容操作指导

注意

若集群中所有节点均已安装 Kafka Broker，进行 Kafka Broker 扩容前则需要先添加主机，然后再进行 Kafka Broker 扩容。如果集群中有扩容所用主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 Kafka 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-5所示。

a. 选择进程及主机

在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

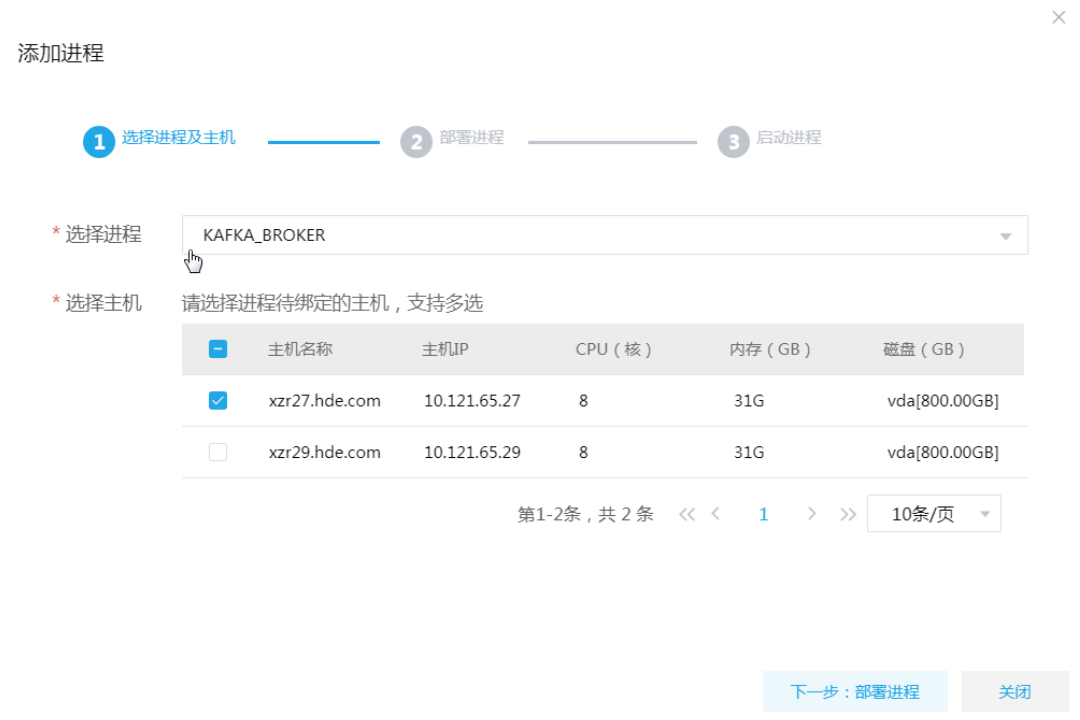
b. 部署进程

选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

c. 启动进程

部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-5 添加进程



(3) 查看进程变化

Kafka Broker 扩容完成之后，在组件详情页面[部署拓扑]页签中可以查看 Kafka Broker 安装数量的变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.4.6 扩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Kafka 组件检查，确保 Kafka 组件可正常使用，详情请参考 2.2 2. 组件检查。

(3) 查看 Kafka 组件部署拓扑，可看到已有新增的扩容节点。

(4) 打开 Kafka 快速链接，在 Kafka UI 首页查看 Kafka Broker 的信息。

3.5 Kafka集群缩容

Kafka 集群缩容是指将某节点上已安装的 Kafka Broker 删除。

3.5.1 使用场景

Kafka 集群缩容的场景主要有：

- 初始 Kafka Broker 节点规划不合理。
- 当 Kafka Broker 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.5.2 缩容前准备

1. 缩容规划

缩容可以删除任意 Broker 节点，但必须保证缩容后 Kafka 集群 Kafka Broker 节点个数至少为 3 个。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Kafka 组件的状态是否正常。

(2) 进入 Kafka 组件详情页，查看 Kafka 的部署拓扑，确保每个服务的状态正常，Kafka Broker、Kafka Eagle 处于“已启动”状态。

3.5.3 缩容约束

- 缩容操作一旦开始，不支持中止。
- 执行缩容前，请先手动停止缩容节点对应的 Kafka Broker 进程，并保证缩容后 Kafka 集群 Kafka Broker 节点个数至少为 3 个。

3.5.4 缩容影响

若缩容前未进行数据迁移，缩容节点上的数据可能会丢失。

3.5.5 缩容操作指导

说明

- Kafka Broker 缩容操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行 Kafka Broker 缩容操作”为例进行说明，在主机详情页面执行 Kafka Broker 缩容操作，与其类似不再进行说明。
- 缩容前，必须确认待删除的 Broker 中不存在有某个 partition 的全部副本。可以使用 ***kafka-topics.sh --zookeeper node1:2181 –topic test –describe*** 命令查看指定 topic 的描述信息。

缩容操作步骤如下：

(1) 在 Kafka 组件详情页面，选择缩容节点。

(2) 在组件详情页面[部署拓扑]页签下，选择已安装 Kafka Broker 进程且需要缩容的主机，然后单击该进程右侧操作中的<停止>按钮，停止 Kafka Broker。

(3) 删除 Kafka Broker

停止 Kafka Broker 后，如图 3-6 所示，在该进程右侧操作中单击<删除>按钮，即可完成 Kafka Broker 的缩容。

图3-6 删除进程

(4) 查看进程变化

Kafka Broker 缩容完成之后，在组件详情页面[部署拓扑]页签中可以查看 Kafka Broker 安装数量的变化以及状态。

(5) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.5.6 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Kafka 组件检查，确保 Kafka 组件可正常使用，详情请参考2.2 2. 组件检查。

(3) 查看 Kafka 组件部署拓扑，可看到缩容节点是否已经删除。

(4) 打开 Kafka 快速链接，在 Kafka UI 首页查看 Kafka Broker 的信息。

3.6 租户管理

注意

- 新增租户时，要求系统中必须已成功创建租户模式的 Hadoop 集群或 Kafka 集群。
- 关于租户管理模块功能的配置或使用详情请参见产品在线联机帮助。

多个租户之间共享一套集群，共享网络和集群资源，并且不同租户之间保证资源隔离。

- 新增租户

  普通用户在自己创建的租户集群中申请租户时，无需审批，会直接触发新增租户的操作。普通用户在其他用户创建的租户集群中申请租户时，则需要走流程审批，待审批人审批通过后才能触发新增租户的操作。管理员用户新增租户时，无需审批，会直接触发新增租户的操作。

- 租户管理操作

普通用户在自己创建的租户集群中执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。普通用户在其他用户创建的租户集群中执行租户续期、资源扩缩容操作时，则需要走流程审批，待审批人审批通过后才能触发相关操作。管理员用户执行租户续期、资源扩缩容操作时，无需审批，会直接触发相关操作。

3.6.1 租户介绍

租户申请的 Kafka 资源对应一个或多个 Topic，Topic 可设置副本数和存储空间

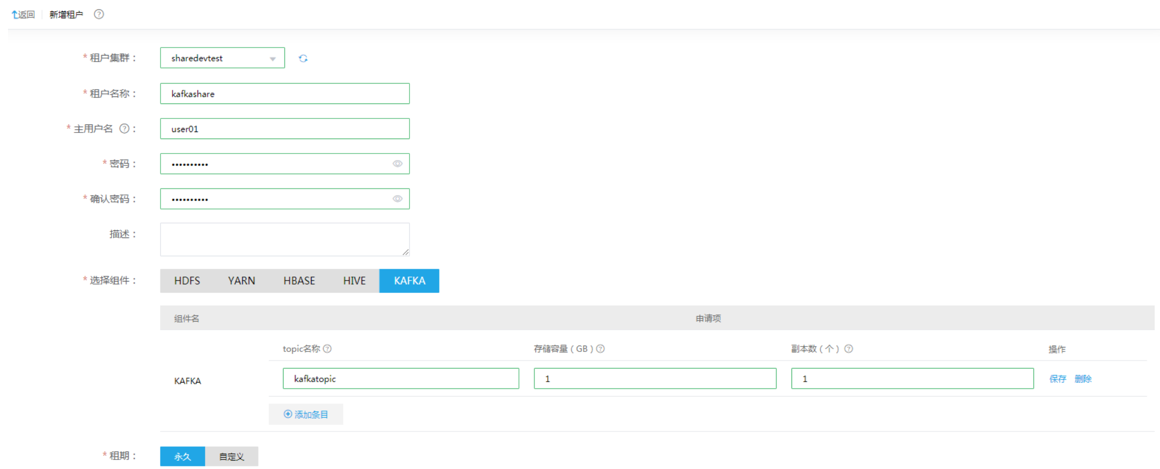
【说明】

- Kafka 存储在本地磁盘，不是 HDFS。
- 租户申请 Kafka 资源时，需提前规划副本数和存储空间，租户创建成功后，副本数不支持修改（一般推荐设置 3 个即可），存储空间仅支持扩容（不支持缩容）。

3.6.2 新增租户

(1) 在[集群权限/租户管理]页面，单击<新增租户>按钮，跳转至新增租户页面，如图 3-7 所示。在租户集群 sharedevtest 中新增租户 kafkashare，主用户 user01，并为该租户配置 Kafka 组件资源（topic 为 kafkatopic、1GB 容量、1 个副本数）。

图3-7 新增租户

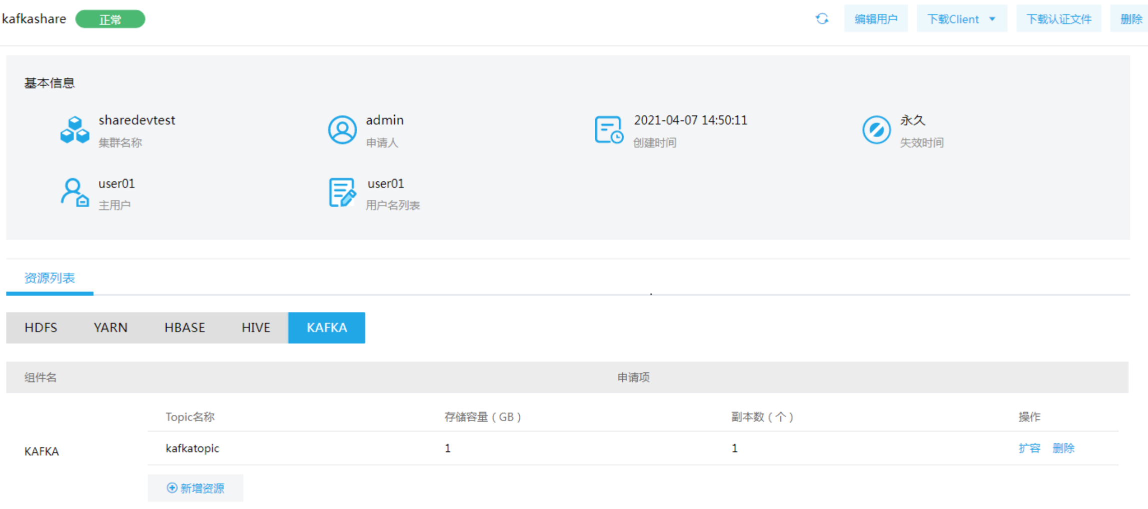


(2) 新增租户成功后，用户可在租户列表查看到已创建的租户，同时可以看到其所属集群、申请人、用户名、创建时间、失效时间等相关信息，如图 3-8 所示。

图3-8 查看租户

(3) 单击租户名称，可查看租户详情，如图 3-9 所示，可以看到对应的 topic 名称、容量和副本数信息。用户 user01 拥有 Kafka 租户 kafkashare 的所有权限，若资源不够/过多时，可编辑租户对其执行扩容/缩容操作。

图3-9 查看租户详情



3.6.3 租户使用操作示例

注意

- 租户集群缺省开启 Kerberos 认证，在使用租户时需要通过该租户的用户对应的认证文件，对租户的用户进行身份认证。租户的用户认证文件的下载在租户列表页面或租户详情页面执行，关于对租户的用户进行认证的方式与集群中用户的身份认证方式相同，详情请参见 2.3.3 Kerberos 环境下的用户身份认证。
- Kafka 租户包括客户端（即组件 Client），系统提供了下载 Kafka 客户端的功能。在客户端节点上安装 Kafka 的 Client 后，即可连接租户中的此组件，执行组件维护、主题管理等操作。租户组件 Client 的下载在租户详情页面执行，关于租户组件 Client 的安装方式与集群组件 Client 相同，详情请参见 3.2 Client 下载/安装/使用/卸载。

(1) 使用 user01 用户登录到租户集群，查看当前集群的所有 topic，如图 3-10 所示，可以看到该集群中已创建 kafkatopic 主题。

图3-10 查看 topic

```
sh-4.2$ /usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --zookeeper 10.121.47.158
:2181 --list
2020-04-14 17:58:26.774 ERROR (main-EventThread) [    ] k.z.ZooKeeperClient [ZooK
eeperClient] Auth failed.
__consumer_offsets
ambari_kafka_service_check
kafkatopic
test002
```

(2) 使用 user01 用户对 kafkatopic 主题执行生产，消费消息等操作，如图 3-11、图 3-12 所示。

- 向 topic 生产数据，命令如下：

  ./kafka-console-producer.sh --broker-list 10.121.47.158:6667 --topic kafkatopic --producer.config /home/user01/p.txt

- 消费 topic 数据，命令如下：

  ./kafka-console-consumer.sh --bootstrap-server 10.121.47.158:6667 --topic kafkatopic --from-beginning --consumer.config /home/user01/p.txt

【说明】p.txt 文件需用户提前创建，文件名可自定义，文件内容为：

security.protocol=SASL_PLAINTEXT

图3-11 生产消息

```
sh-4.2$ ./kafka-console-producer.sh --broker-list 10.121.47.158:6667 --topic kaf
katopic --producer.config /home/user01/p.txt
>source
```

图3-12 消费消息

```
sh-4.2$ ./kafka-console-consumer.sh --bootstrap-server 10.121.47.158:6667 --topi
c kafkatopic --from-beginning --consumer.config /home/user01/p.txt
test
exit
test
topic
kafkatopic
user01
source
exit
source
```

3.7 备份恢复

备份恢复可以提供跨集群之间的数据同步功能，当前版本支持对 Kafka 组件中的指定数据进行同步备份，以保证数据内容不丢失。

备份恢复功能主要是通过创建同步任务实现集群间的数据同步能力。使用同步任务功能，可以为集群中的数据提供同步能力，以满足日常数据备份的需求，保证系统或机器故障时的数据不丢失。

Kafka 数据备份恢复任务可将源集群中的某些 topic 数据拷贝到目的集群。

3.7.1 新建 Kafka 同步任务

注意

- 源集群为同步任务的数据输出集群（一般指新建同步任务的本集群）；目的集群为同步任务的数据输入集群。
- 新建同步任务时，要求源集群与目的集群的集群类型、集群模式均相同。
- 新建同步任务时，要求源集群与目的集群的安全管理策略相同，即同时开启 Kerberos 认证或同时都没有开启 Kerberos 认证。
- 新建同步任务时，要求源集群与目的集群的集群名称、节点主机名不相同，否则配置跨集群互信时可能出错。
- Kafka 同步任务的执行集群只能是目的集群，不影响源集群的业务。
- 新建 Kafka 同步任务时，默认不会同步历史数据，即"高级配置"中 consumer.auto.offset.reset 的值为 latest。此时，通过修改"高级配置"中 consumer.auto.offset.reset 的值为 earliest，并配置此任务的消费者组为一个新的消费者组，即可实现所有数据的同步（包括历史数据）。

Kafka 同步任务为持续运行任务，即 Kafka 同步任务启动后将一直运行，若想要停止只能手动触发。

1. 前提条件

- 新建 Kafka 同步任务前，需要对源集群与目的集群配置跨集群互信。配置方法详情请参见 3.7.2 源集群和目的集群配置互信。
- 运行 Kafka 同步任务前，需要根据集群的配置情况进行相关配置修改，详情请参见 3.7.3 Kafka 同步任务相关配置。

2. 新建 Kafka 同步任务

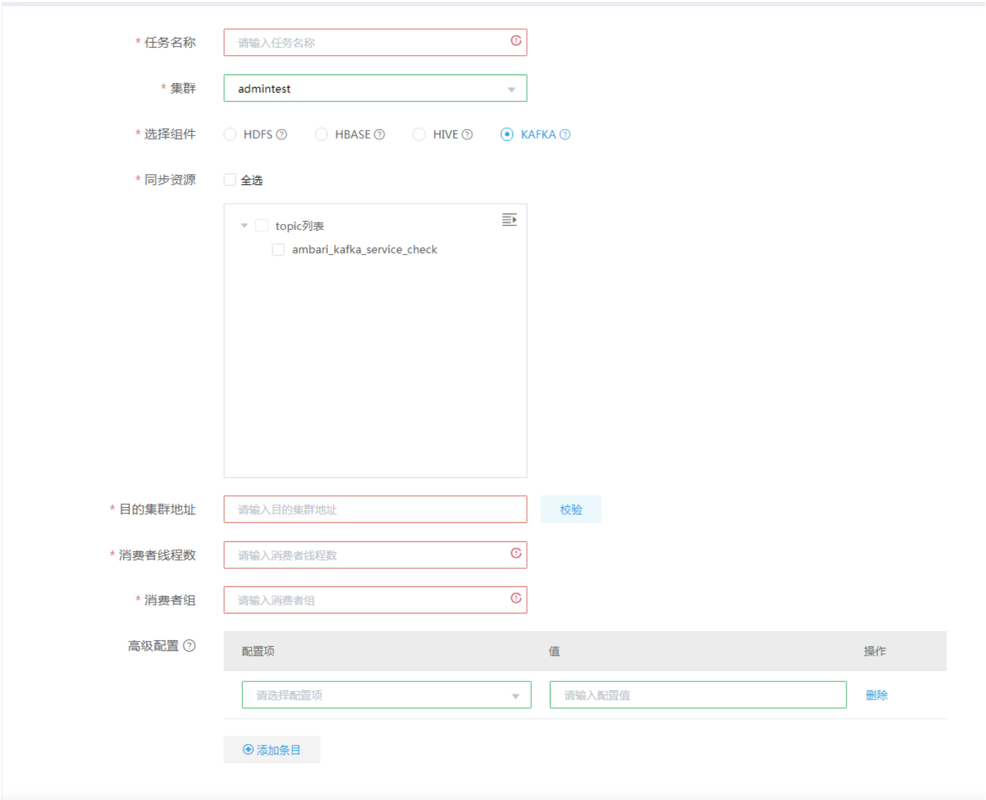
新建 Kafka 同步任务的前提条件准备完成后，即可开始创建 Kafka 同步任务。步骤如下：

(1) 在[集群管理/备份恢复]页面，单击<新建同步任务>按钮，进入新建同步任务页面。

(2) 选择 Kafka 组件，如图 3-13 所示，根据提示配置对应参数项的值，关于各参数项配置详情请参见产品在线联机帮助。

(3) 相关信息配置完成后，若单击<新建>按钮可完成 Kafka 同步任务的新建，此时任务未启动（需要手动启动）；若单击<新建并启动>按钮则可完成 Kafka 同步任务的新建并立刻启动该任务。

图3-13 新建 Kafka 同步任务



3.7.2 源集群和目的集群配置互信

注意

不同集群之间可通过组件同步任务进行数据同步，但创建同步任务之前必须配置源集群和目的集群互信。

示例集群如下：

- 源集群
  - 开启 Kerberos 认证集群 clusterA，kerberos realm 为 CLUSTERA.COM。
- 目的集群
  - 开启 Kerberos 认证集群 clusterB，kerberos realm 为 CLUSTERB.COM。

1. 开启 kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

(3) 修改源集群和目的集群中所有节点的/etc/krb5.conf 文件，要求：

- 修改 realms，要求同时包含源集群和目的集群的 realms 内容（互相拷贝即可），如图 3-14 所示。
- 修改 domain_realm，要求同时包含源集群和目的集群的 domain_realm 内容。
  – 当源集群与目的集群的主机名后缀相同时，需要将 domain_realm 内容修改为“集群中各节点主机名=对应的 realms 名”。示例：源集群和备集群的主机名后缀均为.hde.com，则配置如图 3-15 所示。
  – 当源集群与目的集群的主机名后缀不同时，则可直接将两个集群的 domain_realm 中内容合并（不需要修改，直接互相拷贝即可）。示例：源集群的主机名后缀为.hde.com，目的集群的主机名后缀为.hadoop.com，则配置如图 3-16 所示。

图3-14 realms 修改后示例

```
[realms]
    CLUSTERA.COM = {
        kdc = clustera1.hde.com
        admin_server = clustera1.hde.com
        database_module = openldap_ldapconf
    }
    CLUSTERB.COM = {
        kdc = clusterb1.hde.com
        admin_server = clusterb1.hde.com
        database_module = openldap_ldapconf
    }
```

图3-15 domain_realm 修改后示例（源集群和目的集群主机名后缀相同）

```
[domain_realm]
   clustera1.hde.com=CLUSTERA.COM
   clustera2.hde.com=CLUSTERA.COM
   clustera3.hde.com=CLUSTERA.COM
   clusterb1.hde.com=CLUSTERB.COM
   clusterb2.hde.com=CLUSTERB.COM
   clusterb3.hde.com=CLUSTERB.COM
```

图3-16 domain_realm 修改后示例（源集群和目的集群主机名后缀不同）

```
[domain_realm]
    .hde.com = CLUSTERA.COM
    hde.com = CLUSTERA.COM
    .hadoop.com = CLUSTERB.COM
    hadoop.com = CLUSTERB.COM
```

(4) 在源集群和目的集群的 Master 节点上，执行添加 principal 操作。

a. 在源集群 Master 节点上，分别执行以下两条命令，添加 principal（命令执行后，需要分别输入密码）：

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '

b. 在目的集群 Master 节点上，分别执行以下两条命令，添加 principal（命令执行后，需要分别输入密码）：

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERA.COM@CLUSTERB.COM '

kadmin.local -q ' addprinc -e "aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96" krbtgt/CLUSTERB.COM@CLUSTERA.COM '

【注意】

- CLUSTERA.COM 和 CLUSTERB.COM 分别为源集群和目的集群的域名，请根据实际情况进行修改。添加 principal 时需确保两个集群输入的密码相同，且密码要求至少 8 位（即上述四条命令运行后输入的密码均相同），否则会提示密码太短导致设置无效。
- 若添加 principal 时输入的密码不同，可在源集群和目的集群上进行删除，然后重新执行第 4 步添加 principal 的操作。删除命令如下：

  kadmin.local -q ' delprinc krbtgt/CLUSTERA.COM@CLUSTERB.COM'

  kadmin.local -q ' delprinc krbtgt/CLUSTERB.COM@CLUSTERA.COM'

(5) 源集群与目的集群互信配置完成后，可登录目的集群进行校验。校验方式示例：在目的集群后台切换至集群超级用户，执行命令 hdfs dfs -ls hdfs://<源集群 Active NameNode IP 地址>:8020/，查看源集群的 Kafka 是否可正常访问，若能正常访问则表示源集群与目的集群的互信配置成功。

2. 不开 kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则集群互信的配置步骤如下：

(1) 检查并修改源集群和目的集群的时区、时间均同步。

(2) 修改源集群和目的集群中所有节点的/etc/hosts 文件，要求所有节点的/etc/hosts 文件均同时包含源集群和目的集群的所有节点信息（互相拷贝即可）。

3.7.3 Kafka 同步任务相关配置

1. 开启 Kerberos 认证的环境

若源集群和目的集群都开启了 Kerberos 认证，创建 Kafka 同步任务之前，需进行以下配置修改：

- 对源集群进行配置修改，说明如下：

  在源集群的[集群列表/集群详情/Kafka 组件详情]页面，修改 Kafka 配置项 sasl.kerberos.principal.to.local.rules 的值。要求在末尾 DEFAULT 之前增加内容“RULE:[1:$1@$0](.*@\CLUSTERA.COM$)s/@\CLUSTERA.COM$//,RULE:[2:$1@$0](.*@\CLUSTERA.COM$)s/@\CLUSTERA.COM$//,RULE:[1:$1@$0](.*@\CLUSTERB.COM$)s/@\CLUSTERB.COM$//,RULE:[2:$1@$0](.*@\CLUSTERB.COM$)s/@\CLUSTERB.COM$//”，其中 clusterA 为源集群的集群名称，CLUSTERA.COM 为源集群的域名，clusterB 为目的集群的集群名称，CLUSTERB.COM 为目的集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启相关组件。
- 对目的集群进行配置修改，说明如下：

  在目的集群的[集群列表/集群详情/Kafka 组件详情]页面，修改 Kafka 配置项 sasl.kerberos.principal.to.local.rules 的值。要求在末尾 DEFAULT 之前增加内容“RULE:[1:$1@$0](.*@\CLUSTERB.COM$)s/@\CLUSTERB.COM$//,RULE:[2:$1@$0](.*@\CLUSTERB.COM$)s/@\CLUSTERB.COM$//,RULE:[1:$1@$0](.*@\CLUSTERA.COM$)s/@\CLUSTERA.COM$//,RULE:[2:$1@$0](.*@\CLUSTERA.COM$)s/@\CLUSTERA.COM

$//”，其中 clusterA 为源集群的集群名称，CLUSTERA.COM 为源集群的域名，clusterB 为目的集群的集群名称，CLUSTERB.COM 为目的集群的域名，请根据实际情况进行修改。此配置项修改完成后，需重启相关组件。

2. 不开 Kerberos 认证的环境

若源集群和目的集群都未开启 Kerberos 认证，则在集群之间运行 Kafka 同步任务时，不需要进行相关配置的修改。

3.7.4 Kafka 备份恢复示例

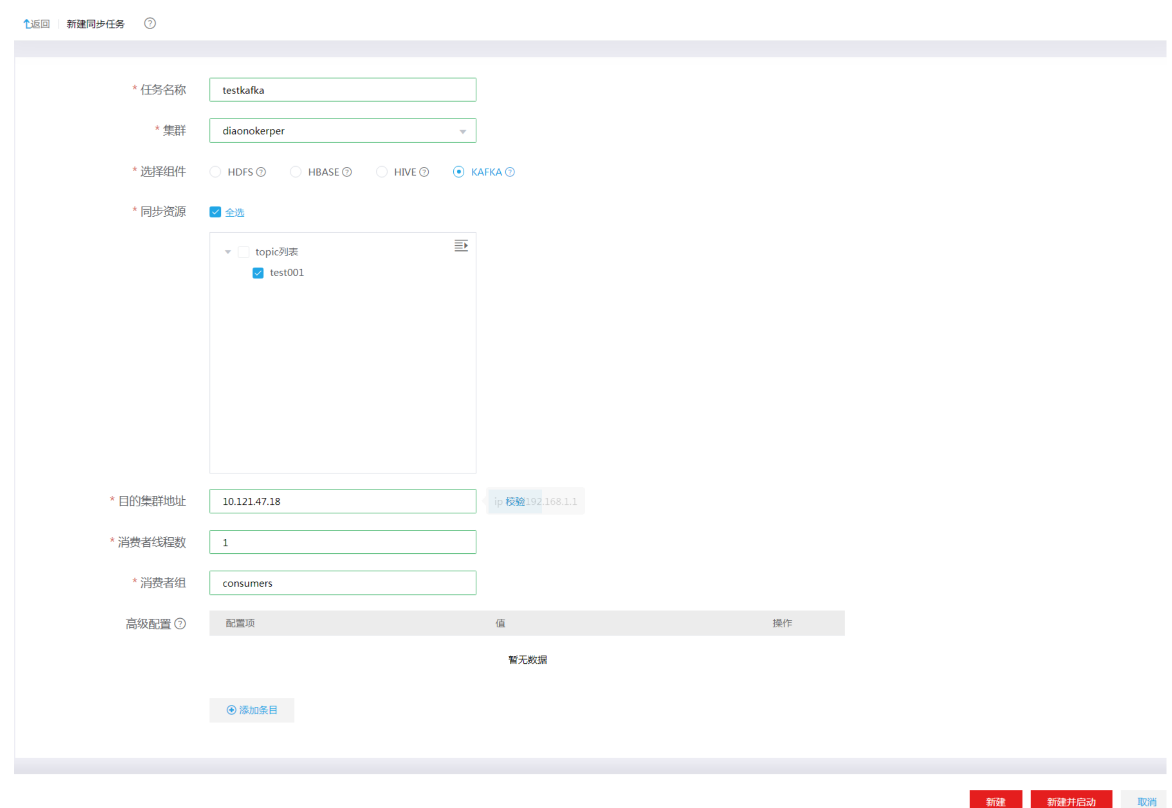
示例场景：ClusterA、ClusterB 两个集群都已安装 Kafka 组件，且均未开启 Kerberos 认证。现使用备份恢复功能将 ClusterA 集群（源集群）Kafka 组件中的数据备份至 ClusterB 集群（目的集群）中，操作步骤如下：

(1) 配置 ClusterA、ClusterB 集群互信，详情请参考 3.7.2 源集群和目的集群配置互信。

【说明】集群互信配置完成后，若集群 ClusterA、ClusterB 之间的 Kafka 组件可以互相进行读写操作则表示配置互信成功。

(2) 在备份恢复页面，单击<新建同步任务>按钮，进入新建同步任务页面，根据业务需要和页面提示配置相关参数后即可新建 testkafka 同步任务，如图 3-17 所示。

图3-17 新建 Kafka 同步任务



(3) 相关信息配置完成后，单击<新建>按钮可完成 Kafka 同步任务的新建，此时任务初始会处于未启动状态；单击<新建并启动>按钮可完成 Kafka 同步任务的创建并立刻启动该任务。

(4) 任务启动后即执行数据备份，在任务列表中选择<更多/运行记录>可查看任务执行记录以及日志详情。

3.8 Kafka API使用指南

1. Producer API 示例

用户应用程序可以利用该 API 发送消息流到一个或者多个 Kafka topic。

需要 maven 依赖如下代码：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version> 2.7.2</version>
</dependency>
```

简单示例代码如下：

```
//kafka producer 配置参数
Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:6667");
 props.put("acks", "all");
 props.put("retries", 0);
 props.put("batch.size", 16384);
 props.put("linger.ms", 1);
 props.put("buffer.memory", 33554432);
 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
//实例化 KafkaProducer
 Producer<String, String> producer = new KafkaProducer<>(props);
 for(int i = 0; i < 100; i++){
//发送消息流到 kafka
    producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i),
Integer.toString(i)));
}
//关闭 KafkaProducer
 producer.close();
```

2. Consumer API 示例

用户应用程序可以利用该 API 消费一个或者多个 Kafka topic 中的数据流。需要 maven 依赖如下代码：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.7.2</version>
</dependency>
```

简单示例如下：

```
// kafka consumer 配置参数
Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:6667");
    props.put("group.id", "test");
    props.put("enable.auto.commit", "false");
```

```
    props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
   //实例化 KafkaConsumer
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Arrays.asList("foo", "bar"));
    final int minBatchSize = 200;
    List<ConsumerRecord<String, String>> buffer = new ArrayList<>();
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            buffer.add(record);
        }
        if (buffer.size() >= minBatchSize) {
            insertIntoDb(buffer);
   //提交 offset 到 kafka
            consumer.commitSync();
            buffer.clear();
        }
    }
```

4 最佳实践

4.1 关键参数说明

1. min.insync.replicas

- 参数说明：该参数属于服务端配置，在大数据平台管理系统的组件详情页面的[配置]页签可进行配置。
- 可选值：>=1，默认值为 1。

  说明：当业务端生产者 acks 配置为-1 或 all 时，该参数决定生产者确认写成功的 isr 副本个数。

2. offsets.commit.required.acks

- 参数说明：该参数属于业务端生产者配置，即在业务代码中进行配置。
- 可选值：0、1、-1、all，默认值为-1。
  - 0：生产者不等待任何响应，即确认写成功。
  - 1：生产者等待 leader 确认写成功，即确认写成功。
  - -1 或 all：生产者等待 leader 和所有 isr 都写成功，即确认写成功。

3. unclean.leader.election.enable

- 参数说明：该参数表示是否允许非 isr 副本选举为 leader，属于服务端配置。
- 可选值：true、false，默认值为 true。
  - true：允许，优先保证 Topic 可用性。极端情况下，存在丢失数据的风险。
  - false：不允许，优先保证数据一致性。极端情况下，存在 Topic 不可用的风险。

4.2 Kakfa producer实践案例

场景描述：向指定 topic（in）发送数据，并打印发送内容。为提高吞吐量，该案例结合线程池使用。

1. producerAPI 案例代码 (Kerberos 环境)

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class kafkaProducerDemo {
    public static void main(String[] args) throws Exception{
        System.out.println("Usage: java -cp YourJar YourPackageClassName");
//kerberos 环境下，需额外加载如下两项配置文件
System.setProperty("java.security.auth.login.config", "/home/kafka_client_jaas.conf");
System.setProperty("java.security.krb5.conf", "/home/krb5.conf");
        Properties props = new Properties();
        props.put("bootstrap.servers", "node1:6667,node2:6667,node3:6667");
        props.put("acks", "1");
```

```
        props.put("retries",1);
        props.put("batch.size", 16384);
        props.put("linger.ms", 10);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
//kerberos 环境下，需额外添加如下三项配置
props.put("security.protocol", "SASL_PLAINTEXT");
props.put("sasl.mechanism", "GSSAPI");
props.put("sasl.kerberos.service.name", "kafka");
final KafkaProducer<String, String> producer = new KafkaProducer<>(props);
//线程数
ExecutorService executorService = Executors.newFixedThreadPool(1);
//kafka topic
final String sendTopic = "in";

executorService.execute(new Runnable() {
   @Override
   public void run() {
      while (true){
          //发送的数据内容
          String sendData = "timestamp:" + System.currentTimeMillis();
          producer.send(new ProducerRecord<String, String>(sendTopic, sendData));
          try{
             Thread.sleep(1000);
          }catch (Exception e){
             e.printStackTrace();
          }
          System.out.println( "Topic:" + sendTopic + " Data:" + sendData);
   }
}});}}
```

2. producerAPI 案例代码 (非 Kerberos 环境)

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class kafkaProducerDemo {
    public static void main(String[] args) throws Exception{
        System.out.println("Usage: java -cp YourJar YourPackageClassName");
        Properties props = new Properties();
        props.put("bootstrap.servers", "node1:6667,node2:6667,node3:6667");
        props.put("acks", "1");
        props.put("retries",1);
        props.put("batch.size", 16384);
        props.put("linger.ms", 10);
```

```
        props.put("buffer.memory", 33554432);
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
final KafkaProducer<String, String> producer = new KafkaProducer<>(props);
//线程数
ExecutorService executorService = Executors.newFixedThreadPool(1);
//kafka topic
final String sendTopic = "in";

executorService.execute(new Runnable() {
   @Override
   public void run() {
      while (true){
      //发送的数据内容
          String sendData = "timestamp:" + System.currentTimeMillis();
          producer.send(new ProducerRecord<String, String>(sendTopic, sendData));
          try{
             Thread.sleep(1000);
          }catch (Exception e){
             e.printStackTrace();
          }
          System.out.println( "Topic:" + sendTopic + " Data:" + sendData);
   }
}
});
}
}
```

4.3 Kakfa consumer实践案例

【场景描述】：消费者指定 topic 数据，并打印到控制台。为提高吞吐量，该案例结合多线程使用。

1. consumerAPI 案例代码 (Kerberos 环境)

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.util.Arrays;
import java.util.Properties;
public class kafkaConsumerDemo {
public static void main(String[] args) {
   System.out.println("java -cp ${jar} kafkaConsumerDemo ${bootstrap.servers} ${topic}");
//kerberos 环境下，需额外加载如下两项配置文件
System.setProperty("java.security.auth.login.config", "/home/kafka_client_jaas.conf");
System.setProperty("java.security.krb5.conf", "/home/krb5.conf");
   Properties props = new Properties();
   props.put("bootstrap.servers", " node1:6667,node2:6667,node3:6667");
   props.put("group.id", "kafkaConsumerDemo1");
```

```
   props.put("enable.auto.commit", "true");
   props.put("auto.offset.reset", "earliest");
   //props.put("auto.offset.reset", "latest");
   props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
  props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

//kerberos 环境下，需额外添加如下三项配置
props.put("security.protocol", "SASL_PLAINTEXT");
props.put("sasl.mechanism", "GSSAPI");
props.put("sasl.kerberos.service.name", "kafka");

     KafkaConsumer<String,String> consumer = new KafkaConsumer<>(props);
   consumer.subscribe(Arrays.asList("out"));
   while (true) {
          ConsumerRecords<String, String> records = consumer.poll(1000);
          for (ConsumerRecord<String, String> record : records) {
             System.out.println(record);
          }
          System.out.println("---");
       }
    }
}
```

2. consumerAPI 案例代码 (非 Kerberos 环境)

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.util.Arrays;
import java.util.Properties;

public class kafkaConsumerDemo {
public static void main(String[] args) {
   System.out.println("java -cp ${jar} kafkaConsumerDemo ${bootstrap.servers} ${topic}");

   Properties props = new Properties();
   props.put("bootstrap.servers", args[0]);
   props.put("group.id", "kafkaConsumerDemo1");
   props.put("enable.auto.commit", "true");
   props.put("auto.offset.reset", "earliest");
   //props.put("auto.offset.reset", "latest");
   props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
  props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

     KafkaConsumer<String,String> consumer = new KafkaConsumer<>(props);
   consumer.subscribe(Arrays.asList(args[1]));
```

```
    while (true) {
            ConsumerRecords<String, String> records = consumer.poll(1000);
            for (ConsumerRecord<String, String> record : records) {
                System.out.println(record);
            }
            System.out.println("---");
        }
    }
}
```

4.4 Flink-Kafka实践案例

【场景描述】：利用 Flink 消费指定 topic 数据，追加时间戳之后，发送到另一个指定 topic。

1. pom 依赖

```
<properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <flink.version>1.12.2</flink.version>
        <scala.binary.version>2.11</scala.binary.version>
    </properties>
<dependencies>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-clients_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-elasticsearch7_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>
    </dependencies>
```

2. Flink 消费 Kafka 案例代码(Kerberos 环境)

```
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
```

```
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;

public class KafkaFlinkKafkaDemo {
    public static void main(String[] args) throws Exception {
        Logger logger = LoggerFactory.getLogger(KafkaFlinkKafkaDemo.class);

        final MultipleParameterTool multipleParameterTool =
MultipleParameterTool.fromArgs(args);
        if (!multipleParameterTool.has("path")) {
            System.out.println("Error: not exist --path /opt/your.properties");
            System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
            System.exit(0);
        }
        ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
        logger.info(configUtils.toString());
        Properties commonProp = configUtils.getCommonProp();

        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.getConfig().setUseSnapshotCompression(true);
        env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

        Properties consumerProp = configUtils.getConsumerProp();
        if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
                .equalsIgnoreCase("true")) {
            consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
            consumerProp.setProperty("sasl.mechanism", "GSSAPI");
            consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }
```

```
        DataStream<String> sourceDataStream =
env.addSource(kafkaConsumer).uid("source_kafka")
                .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parall
elism")));
        sourceDataStream.print();

        env.execute();
    }
}
```

以上案例使用到的 your.properties 配置如下：

```
############################Kafka 配置 #####################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.11.1
consumer.bootstrap.servers=10.121.65.7:6667,10.121.65.8:6667,10.121.65.9:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_es
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=true
```

3. Flink 消费 Kafka 案例代码(非 Kerberos 环境)

```
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.MultipleParameterTool;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;

public class KafkaFlinkKafkaDemo {
    public static void main(String[] args) throws Exception {
        Logger logger = LoggerFactory.getLogger(KafkaFlinkKafkaDemo.class);

        final MultipleParameterTool multipleParameterTool =
MultipleParameterTool.fromArgs(args);
        if (!multipleParameterTool.has("path")) {
            System.out.println("Error: not exist --path /opt/your.properties");
            System.out.println("Usage: flink run -m yarn-cluster -d /opt/your.jar --path
/opt/your.properties");
            System.exit(0);
```

```
        }
        ConfigUtils configUtils = new ConfigUtils(multipleParameterTool.get("path"));
        logger.info(configUtils.toString());
        Properties commonProp = configUtils.getCommonProp();

        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.getConfig().setUseSnapshotCompression(true);
        env.enableCheckpointing(5000); // create a checkpoint every 5 seconds

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);//default
mode

        Properties consumerProp = configUtils.getConsumerProp();
        if (commonProp.containsKey("source.kafka.security.enable") &&
commonProp.getProperty("source.kafka.security.enable")
                .equalsIgnoreCase("true")) {
            consumerProp.setProperty("security.protocol", "SASL_PLAINTEXT");
            consumerProp.setProperty("sasl.mechanism", "GSSAPI");
            consumerProp.setProperty("sasl.kerberos.service.name", "kafka");
        }

        FlinkKafkaConsumer<String> kafkaConsumer = new
FlinkKafkaConsumer<>(commonProp.getProperty("source.kafka.topic"),
                new SimpleStringSchema(), consumerProp);
        kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
        if (commonProp.containsKey("source.kafka.offset.reset") &&
commonProp.getProperty("source.kafka.offset.reset")
                .equalsIgnoreCase("latest")) {
            kafkaConsumer.setStartFromLatest();
        }

        DataStream<String> sourceDataStream =
env.addSource(kafkaConsumer).uid("source_kafka")
                .setParallelism(Integer.valueOf(commonProp.getProperty("source.kafka.parall
elism")));
        sourceDataStream.print();

        env.execute();
    }
}
```

以上案例使用到的 your.properties 配置如下：

```
############################ Kafka 配置 #######################################
#Kafka Consumer 专用配置，支持所有原生配置
consumer.group.id=kafka_flink1.11.1_es
consumer.bootstrap.servers=10.121.65.53:6667,10.121.65.54:6667,10.121.65.55:6667
consumer.max.poll.records=10000
consumer.receive.buffer.bytes=1024000
```

```
consumer.send.buffer.bytes=1024000
consumer.heartbeat.interval.ms=30000
consumer.session.timeout.ms=60000
#Kafka 其他配置
source.kafka.topic=kafka_es
source.kafka.offset.reset=latest
source.kafka.parallelism=1
source.kafka.security.enable=false
```

4.5 SparkStreaming-Kafka实践案例

【场景描述】：利用 SparkStreaming 消费指定 topic 数据，并打印到控制台。

1. pom 依赖

```
<dependencies>
   <dependency>
      <groupId>org.apache.spark</groupId>
     <artifactId>spark-streaming_2.12</artifactId>
     <version>2.4.0</version>
      <scope>provided</scope>
   </dependency>
   <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
      <version>2.4.0</version>
   </dependency>
   <dependency>
           <groupId>org.apache.spark</groupId>
           <artifactId>spark-core_2.12</artifactId>
           <version>2.4.0</version>
           <scope>provided</scope>
       </dependency>
</dependencies>
```

2. Spark 消费 Kafka 案例代码(Kerberos 环境)

```
import java.util.*;
import org.apache.spark.SparkConf;
import org.apache.spark.TaskContext;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.*;
import org.apache.spark.streaming.kafka010.*;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.common.serialization.StringDeserializer;

public class javaSparkStreamingKafka {
   public static void main(String[] args) throws Exception {
      Map<String, Object> kafkaParams = new HashMap<>();
      kafkaParams.put("bootstrap.servers", "node1:6667");
      kafkaParams.put("key.deserializer", StringDeserializer.class);
      kafkaParams.put("value.deserializer", StringDeserializer.class);
      kafkaParams.put("group.id", "use_a_separate_group_id_for_each_stream");
      kafkaParams.put("auto.offset.reset", "latest");
```

```
        kafkaParams.put("enable.auto.commit", false);
//kerberos 环境下，需额外添加如下三项配置
        kafkaParams.put("security.protocol", "SASL_PLAINTEXT");
        kafkaParams.put("sasl.mechanism", "GSSAPI");
        kafkaParams.put("sasl.kerberos.service.name", "kafka");
        Collection<String> topics = Arrays.asList("test");

        SparkConf sparkConf = new SparkConf().setAppName("javasparkstreamingkafkademo");
        JavaStreamingContext javaStreamingContext = new JavaStreamingContext(sparkConf,
Durations.seconds(1));
        JavaInputDStream<ConsumerRecord<String, String>> dStream =
KafkaUtils.createDirectStream(javaStreamingContext,
                LocationStrategies.PreferConsistent(),
                ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams));

        dStream.foreachRDD(rdd -> {
            OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();
            rdd.foreachPartition(consumerRecords -> {
                OffsetRange o = offsetRanges[TaskContext.get().partitionId()];
                //System.out.println(o.topic() + " " + o.partition() + " " + o.fromOffset() + "
" + o.untilOffset());
                //Lambda expressions are not supported at this language
                while(consumerRecords.hasNext()){
                    String value = consumerRecords.next().value();
                    System.out.println(value);
                }
            });
            ((CanCommitOffsets) dStream.inputDStream()).commitAsync(offsetRanges);
        });

        javaStreamingContext.start();
        javaStreamingContext.awaitTermination();
    }
}
```

3. Spark 消费 Kafka 部署模式(Kerberos 环境)

- Cluster 模式案例代码

```
./spark-submit \
--files "/usr/test/kafka_client_jaas.conf,/usr/test/kafka.service.keytab" \
--driver-java-options "-Djava.security.auth.login.config=./kafka_client_jaas.conf" \
--conf
"spark.executor.extraJavaOptions=-Djava.security.auth.login.config=./kafka_client_j
aas.conf" \
--keytab=/usr/test/test_user.keytab \
--principal=test_user  \
--class com.spark.service.javaSparkStreamingKafka \
--master yarn  \
--deploy-mode cluster \
--executor-memory 1G \
--num-executors 1
```

```
/usr/test/sparkstreamingkafka-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```
其中：

- /user/test/为自定义目录，用户可以任意指定
- kafka_client_jaas.conf 内容如下：

```
[root@node1 test]# cat kafka_client_jaas.conf
   KafkaClient {
     com.sun.security.auth.module.Krb5LoginModule required
     useKeyTab=true
     keyTab="./kafka.service.keytab"
     storeKey=true
     useTicketCache=false
     serviceName="kafka"
     principal="kafka/node1.hde.com@HDE.COM";
   };
```

- Client 模式案例代码

```
./spark-submit \
--files
"/usr/test/kafka_client_jaas.conf,/usr/test/kafka_client_jaas1.conf,/etc/security/k
eytabs/kafka.service.keytab" \
--driver-java-options
"-Djava.security.auth.login.config=/usr/test/kafka_client_jaas1.conf" \
--conf
"spark.executor.extraJavaOptions=-Djava.security.auth.login.config=./kafka_client_j
aas.conf" \
--keytab=/usr/test/test_user.keytab \
--principal=test_user \
--class com.spark.service.javaSparkStreamingKafka \
--master yarn  \
--deploy-mode client \
--executor-memory 1G \
--num-executors 1
/usr/test/sparkstreamingkafka-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```
其中：

- /user/test/为自定义目录，用户可以任意指定
- kafka_client_jaas.conf 内容如下：

```
[root@node1 test]# cat kafka_client_jaas.conf
   KafkaClient {
     com.sun.security.auth.module.Krb5LoginModule required
     useKeyTab=true
     keyTab="./kafka.service.keytab"
     storeKey=true
     useTicketCache=false
     serviceName="kafka"
     principal="test_user";
   };
```

- kafka_client_jaas1.conf 内容如下：

```
[root@node1 test]# cat kafka_client_jaas1.conf
   KafkaClient {
     com.sun.security.auth.module.Krb5LoginModule required
     useKeyTab=true
     keyTab="/user/test/kafka.service.keytab"
     storeKey=true
     useTicketCache=false
```

```
          serviceName="kafka"
          principal="/user/test/kafka.service.keytab";
        };
```

4. Spark 消费 Kafka 案例代码(非 Kerberos 环境)

```
import java.util.*;
import org.apache.spark.SparkConf;
import org.apache.spark.TaskContext;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.*;
import org.apache.spark.streaming.kafka010.*;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.common.serialization.StringDeserializer;

public class javaSparkStreamingKafka {
  public static void main(String[] args) throws Exception {
    Map<String, Object> kafkaParams = new HashMap<>();
    kafkaParams.put("bootstrap.servers", "node1:6667");
    kafkaParams.put("key.deserializer", StringDeserializer.class);
    kafkaParams.put("value.deserializer", StringDeserializer.class);
    kafkaParams.put("group.id", "use_a_separate_group_id_for_each_stream");
    kafkaParams.put("auto.offset.reset", "latest");
    kafkaParams.put("enable.auto.commit", false);

    Collection<String> topics = Arrays.asList("test");

    SparkConf sparkConf = new SparkConf().setAppName("javasparkstreamingkafkademo");
    JavaStreamingContext javaStreamingContext = new JavaStreamingContext(sparkConf,
Durations.seconds(1));
    JavaInputDStream<ConsumerRecord<String, String>> dStream =
KafkaUtils.createDirectStream(javaStreamingContext,
        LocationStrategies.PreferConsistent(),
        ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams));

    dStream.foreachRDD(rdd -> {
      OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();
      rdd.foreachPartition(consumerRecords -> {
        OffsetRange o = offsetRanges[TaskContext.get().partitionId()];
        //System.out.println(o.topic() + " " + o.partition() + " " + o.fromOffset() + "
" + o.untilOffset());
        //Lambda expressions are not supported at this language
        while(consumerRecords.hasNext()){
          String value = consumerRecords.next().value();
          System.out.println(value);
        }
      });
      ((CanCommitOffsets) dStream.inputDStream()).commitAsync(offsetRanges);
    });

    javaStreamingContext.start();
```

```
        javaStreamingContext.awaitTermination();
    }
}
```

5. Spark 消费 Kafka 部署模式(非 Kerberos 环境)

- Cluster 模式案例代码

```
./spark-submit \
--class com.spark.service.javaSparkStreamingKafka \
--master yarn  \
--deploy-mode cluster \
--executor-memory 1G \
--num-executors 1
/usr/test/sparkstreamingkafka-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

- Client 模式案例代码

```
./spark-submit \
--class com.spark.service.javaSparkStreamingKafka \
--master yarn  \
--deploy-mode client \
--executor-memory 1G \
--num-executors 1
/usr/test/sparkstreamingkafka-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

5 常见问题解答

5.1 性能调优

1. Broker 调优

- 处理消息的最大线程数配置项：
  - num.network.threads：Broker 处理消息的最大线程数。主要用于处理网络 IO，读写缓冲区数据，基本没有 IO 等待。

    推荐配置：cpu 核数加 1。
  - num.io.threads：Broker 处理磁盘 IO 的线程数。主要进行磁盘 IO 操作，高峰期可能有些 IO 等待。

    推荐配置：cpu 核数 2 倍，最大不要超过 3 倍。
- socket server 可接受数据大小配置项：

  socket.request.max.bytes：推荐用户根据自己业务数据包的大小适当调大。该取值是 int 类型的，其受限于 java int 类型的取值范围不能太大。
- log 数据文件刷盘策略配置项：
  - log.flush.interval.messages：设置触发刷盘操作的数据量。
  - log.flush.interval.ms：设置刷盘操作的时间间隔。

  推荐配置：
  - 为了大幅度提高 Producer 写入吞吐量，需要定期批量写文件。一般无需改动，如果 Topic 的数据量较小可以考虑减少 log.flush.interval.ms 和 log.flush.interval.messages 的值来强制刷写数据，减少可能由于缓存数据未写盘带来的不一致问题。
  - 推荐设置 log.flush.interval.messages 为 10000，log.flush.interval.ms 为 1s。
- 日志保留策略配置项：
  - log.retention.hours：设置日志数据保留时长。
  - log.segment.bytes：设置段文件大小。

  推荐配置：日志默认保留七天，具体值需根据用户业务数据量以及 Kafka 容量进行配置。段文件配置 1GB，有利于快速回收磁盘空间，重启 Kafka 加载也会加快。如果文件过小，则文件数量会比较多。
- replica 复制：每个 follow 从 leader 拉取消息进行数据同步，follow 同步性能由以下参数决定：
  - 配置项（num.replica.fetchers）：fetcher 配置可以提高 follower 的 I/O 并发度，单位时间内 leader 持有更多请求，相应负载会增大，需要根据机器硬件资源做权衡，建议适当调大。
  - 配置项（replica.fetch.min.bytes）：一般无需更改，选择默认值即可。
  - 配置项（replica.fetch.max.bytes）：默认值为 1MB，推荐设为 5M，可以根据业务情况适当调整。
  - 配置项（replica.fetch.wait.max.ms）：follow 拉取频率，频率过高，leader 会积压大量无效请求情况，无法进行数据同步，导致 cpu 飙升。配置时谨慎使用，建议选择默认值。

2. Producer 调优

- 发送请求的频率。Producer 异步发送消息集到 Broker，定时定量触发。
  - 配置项（batch.size）：控制 Producer 缓存消息集达到多少字节数据时，将消息集发送到 Broker。在保证不超过可用内存前提下，尽可能加大该值。
  - 配置项（linger.ms）：控制 Producer 缓存消息达到多久时，将消息发送到 Broker。推荐配置 5 或 10。
- ACK 机制

  配置项（acks）：ack 校验机制，推荐配置 1。
- 重试次数

  配置项（retries）：发送失败，重试次数。推荐配置 1~10。

3. Consumer 调优

- offset 提交机制

  配置项（enable.auto.commit）：自动定时提交 offset。通常情况下，应用需要自行提交 offset，推荐配置 false。
- Consumer 实例数

  推荐 Consumer 实例数等于 topic 的所有 partition 数。

5.2 运维类问题

1. Kafka JVM 参数如何设置？

Kafka JVM 参数在 Kafka Broker 节点的 bin/kafka-server-start.sh 中设置，目前 Kafka Broker JVM 内存大小可以根据节点内存自动进行设置。

- 节点内存大于 32G：Kafka Broker 堆内存为 8G
- 节点内存为 16G~32G：Kafka Broker 堆内存为 4G
- 节点内存小于 16G：Kafka Broker 堆内存为 1G

说明：一般 Kafka Broker 堆内存不需要调整。

2. Kafka Topic 删除后，Zookeeper 里保存的 Consumer 信息仍保留，怎么解决？

Kafka Topic 删除后，Consumer 并不一定已经停止消费。因此 Kafka 中使用命令删除 Topic 后，实际上 Topic 中保存的数据依旧存在，不影响正常使用。

3. 在执行 bin/kafka-console-consumer.sh 等 Kafka 自带脚本时，出现错误“Unrecognized VM option '+UseCompressedOops' Could not create the Java virtual m”怎么解决？

解决方法：

(1) 在出现问题的 Kafka Broker 节点的/usr/hdp/3.0.1.0-187/kafka/bin/kafka-run-class.sh 脚本中找到：

```
if [ -z "$KAFKA_JVM_PERFORMANCE_OPTS" ]; then
    KAFKA_JVM_PERFORMANCE_OPTS="-server -XX:+UseCompressedOops
    -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled
    -XX:+CMSScavengeBeforeRemark -XX:+DisableExplicitGC -Djava.awt.headless=true"
fi
```

(2) 去掉代码中的 -XX:+UseCompressedOops

(3) 重启该 Kafka Broker 节点。

4. Kafka 使用过程中，出现超时异常类错误，怎么解决？

报错示例一：

通过 java 客户端访问 Kafka，当生产者线程向 Kafka 插入数据时候出现错误“org.apache.kafka.common.errors.TimeoutException: Batch Expired”或者“org.apache.kafka.common.errors.TimeoutException: Failed to update metadata after 3000 ms”

解决方法：

(1) 检查 Kafka Broker 节点网络连接是否正常。

(2) 网络连接正常情况下，建议检查 bootstrap.servers 等配置是否正确。

报错示例二：

Kafka 使用过程中出现超时异常提示“ java.util.concurrent.ExecutionException:org.apache.kafka.common.errors.TimeoutException: Expiring 1 record(s) for xx due to xx ms has passed since batch creation plus linger time”

解决方法：

可以增大 request.timeout.ms 参数值。

5. Kafka 使用中，出现错误提示“java.util.concurrent.ExecutionException:org.apache.kafka.common.errors.NotLeaderForPartitionException: This server is not the leader for that topic-partition”时，怎么解决？

可能原因：

Producer 向 Kafka Broker 写数据时，若正在进行 leader 选举，本来是向 Broker0 上写的，选举之后 Broker1 成为 leader，导致无法写成功，会抛异常“java.util.concurrent.ExecutionException: org.apache.kafka.common.errors.NotLeaderForPartitionException: This server is not the leader for that topic-partition”。

解决办法：

修改 Producer 的重试参数 retries，默认是 0，生产环境中一般建议设置为 10。

6. Kafka broker 挂掉时，怎么解决？

可能原因：

- 磁盘空间不足。例如，查看 Kafka 报错日志，看到报错信息“No space left on device”或查看组件磁盘使用情况，发现使用率为 95%。
- 数据目录权限不足。例如，查看 Kafka 报错日志，看到报错信息“Permission denied”或查看数据目录权限为“root”。
- 数据磁盘损坏。例如，查看 Kafka 报错日志，看到关键报错信息“read-only file system：’/opt/disk9/kafka-logs’”或查看对应数据磁盘发现磁盘为只读。
- Broker id 已经被使用。例如，查看 Kafka 报错日志，看到报错信息“A broker is already registered on the path /brokers/ids/1”。

解决方法：

- 磁盘空间不足：例如，现场跑了很多 bulkload 任务，且数据量较大，建议先停掉所有 bulkload 任务，然后清理磁盘中不需要的数据或者根据自身业务情况修改 Kafka 中磁盘数据保留时长，再重启 Kafka Broker 后正常。

- 数据目录权限不足：授予 Kafka 用户读写数据目录权限。
- 数据磁盘损坏：利用 fsck 进行磁盘修复。
- Broker.id 已经被使用：将 meta.properties 中的 broker.id 改为其他值后，重启所有的 Kafka Broker。

7. Kafka 生产者和消费者测试不通时，怎么解决？

可能原因：

生产者和消费者访问端口输入错误，访问端口与 Kafka 组件的配置端口不一致。

语法错误：例如端口号错误，操作命令格式参数错误。

解决办法：

- 定位思路：
  - 检查生产者和消费者访问端口。
  - 检查并修改错误语法。
- 处理方法：生产者和消费者的访问端口修改为 6667。

8. Topic 无法删除时，怎么解决？

可能原因：

- 参数 delete.topic.enable=false。
- 对应 topic 还有程序在读写。
- topic 所在节点磁盘异常（损坏或者满盘）。
- topic 所在节点 Kafka broker 挂掉。

解决办法：

- 定位思路：
  - 检查参数 delete.topic.enable 的配置。
  - 检查是否有程序在读写该 topic。
  - 检查 Kafka broker 进程是否存在、是否出现异常日志。
  - 检查 Kafka broker 节点 log.dirs 挂载磁盘数据目录是否异常。
- 处理步骤：
  - 修改参数 delete.topic.enable 值为“true”。
  - 停掉读写该 topic 的进程。
  - 查看 Kafka broker 进程日志，确保进程日志运行中且日志无异常。
  - 如果执行上述操作之后仍然无法删除，可先对数据目录中 topic 文件执行强制删除操作，然后重启整个 Kafka 服务。

9. Kafka 使用 partition 数与硬盘分区不匹配时，怎么解决？

可能原因：

Topic 分区数设置不合理。

解决办法：

- 定位思路：使用./kafka-topics.sh --zookeeper <ip>:2181 --describe --topic <topic-name>命令查看 topic 详情，看分区个数与计划使用硬盘数是否相同。

- 处理步骤：调整 topic 分区个数与规划的硬盘数一致，命令为：./kafka-topic.sh --zookeeper <ip>:2181 --alter --partitions <number> --topic <topic-name>。

10. 在执行 bin/kafka-console-consumer.sh 等 Kafka 自带脚本时，出现类似如下错误：Unrecognized VM option '+UseCompressedOops' Could not create the Java virtual m 时，怎么解决？

可能原因：

无法识别 UseCompressedOops 现象。

解决办法：

(1) 检查 bin/kafka-run-class.sh 脚本。在出问题的 Kafka Broker 节点上，在 bin/kafka-run-class.sh 脚本内找到如下内容：

```
if [ -z "$KAFKA_JVM_PERFORMANCE_OPTS" ]; then
    KAFKA_JVM_PERFORMANCE_OPTS="-server -XX:+UseCompressedOops
    -XX:+UseParNewGC -XX:+UseConcMarkSweepGC
-XX:+CMSClassUnloadingEnabled
    -XX:+CMSScavengeBeforeRemark -XX:+DisableExplicitGC
-Djava.awt.headless=true"
fi
```

(2) 删除 kafka-run-class.sh 脚本中的“-XX:+UseCompressedOops”。

(3) 重启该 Kafka Broker 节点即可。

目 录

1 组件简介 1-1
1.1 组件概述 1-1
1.1.1 特点及优势 1-1
1.2 组件架构 1-1
1.2.1 单机模式 1-1
1.2.2 集群模式 1-2
1.3 应用场景 1-3
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 组件数据存放位置 2-1
2.1.2 查看组件的日志信息 2-1
2.2 运行状态监控 2-2
2.3 Client 下载/安装/使用/卸载 2-4
2.3.1 下载 Client 安装包 2-5
2.3.2 安装 Client 2-6
2.3.3 访问组件 2-7
2.3.4 使用客户端 Client 2-7
2.3.5 卸载 Client 客户端 2-7
2.4 快速使用指导 2-7
2.4.1 连接 Redis 2-8
2.4.2 基础操作 2-9
3 使用指南 3-1
3.1 Shell 命令 3-1
3.2 权限访问控制 3-5
3.2.1 权限说明 3-5
3.2.2 权限使用操作示例 3-6
3.3 Redis 集群扩容 3-8
3.3.1 扩容场景 3-8
3.3.2 扩容前准备 3-9
3.3.3 扩容约束 3-9
3.3.4 扩容影响 3-9
3.3.5 扩容操作指导 3-10

3.3.6 扩容验证 3-11
3.4 Redis 集群缩容 3-11
3.4.1 缩容场景 3-11
3.4.2 缩容前准备 3-11
3.4.3 缩容约束 3-11
3.4.4 缩容影响 3-11
3.4.5 缩容操作指导 3-12
3.4.6 缩容验证 3-12
3.5 Redis 多实例 3-13
3.6 Java 方式使用 Redis 3-16
3.6.1 Redis 开发示例 3-16
3.6.2 Redis 适用场景 3-20
4 常见问题解答 4-1
4.1 调优 4-1
4.1.1 配置调优 4-1
4.1.2 其它调优 4-2
4.2 运维类问题 4-1

1 组件简介

1.1 组件概述

Redis 是一种支持多类型数据结构的内存数据库，可用于数据库、数据缓存、消息中间件等。

1.1.1 特点及优势

1. Redis 特点

- Redis 数据保存在内存中，磁盘仅用于持久化。
- 相比其他键值（Key-Value）数据存储，Redis 能够支持较为丰富的数据类型。
- Redis 可以将数据复制到任意数量的从服务器。

2. Redis 优势

- 响应快速：Redis 的数据存储在内存中，可有效提升系统的性能，其读写速度远超传统数据库，常用于缓存热点数据。
- 支持丰富的数据类型：Redis 支持列表、集合、有序集合、散列等数据类型。
- 操作原子性：Redis 操作是原子性的，可以保证当两个客户端同时访问 Redis 服务器时，都将获得更新后的数值。
- 应用广泛：Redis 可以在缓存、消息队列中使用，也可以在 Web 应用会话、网站网页点击计数等应用场景中使用。

1.2 组件架构

Redis 分为单机模式和集群模式这两种体系结构。其中，单机模式只包含一个 Redis 实例，集群模式则至少由三个 Redis 实例组成。

说明

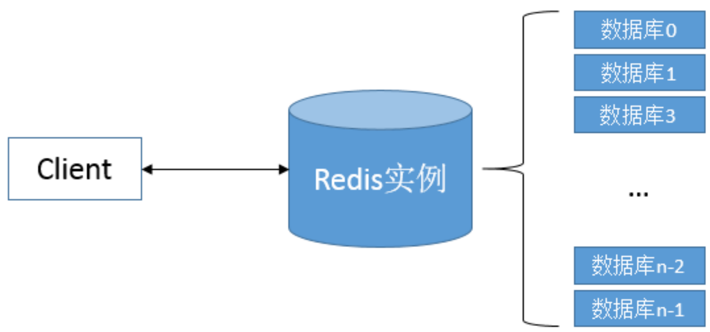
Redis 实例负责响应 Redis 客户端的操作请求。

1.2.1 单机模式

Redis 单机模式的架构如图 1-1 所示。其特点为：

- 用户可通过 Client 连接 Redis 实例，进行数据的读写操作。
- Redis 单机模式，实例包含 n（默认为 16）个数据库，可通过修改配置参数 redis.databases 的值来修改数据库数量。Redis 作为一个 Key-Value 内存数据库，同一数据库中 Key（键）不能重复，但不同数据库之间可以。

图1-1 Redis 单机模式



1.2.2 集群模式

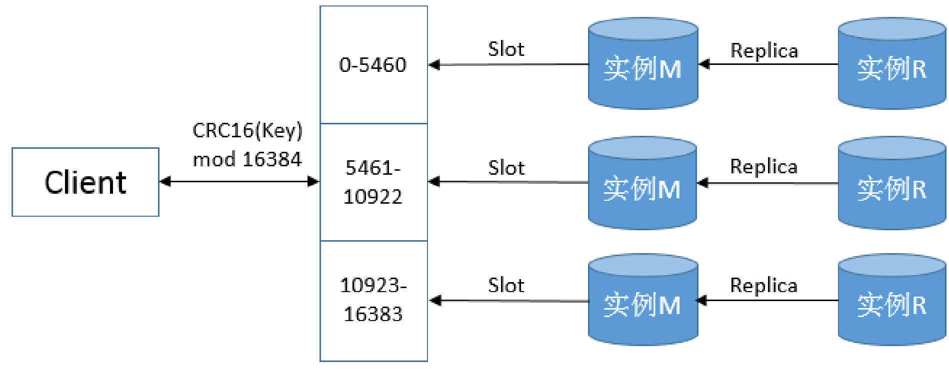
Redis 单机模式仅包含一个实例，即 Redis 只能安装在一个节点上，那么该节点上的主机资源（内存/CPU）会成为限制 Redis 大小的主要因素。

而 Redis 集群模式，可通过哈希槽（Hash Slot）方式极大提升 Redis 的扩展性，且其主从架构可有效保障 Redis 的稳定性。

Redis 集群模式的架构如图 1-2 所示。其特点为：

- Redis 集群模式通常由 3 个主（Master，图中实例 M）实例和 3 个副本（Replica，图中实例 R）实例组成。Client 可连接任一实例对 Redis 集群进行操作。其中，连接 Master 实例可读写数据，连接 Replica 实例可读取数据。
- 三个 Master 平分 16384 个 Slot（槽位）。Client 通过公式：HASH_SLOT = CRC16 (key) mod 16384 来计算出 Key 所映射的 Slot，然后 Redis 会在包含该 Slot 的实例上进行数据的读写操作。
- Redis 集群为无中心结构，实例间使用 Gossip 协议进行通信，每个实例都保存着所有实例和 Slot 的映射关系。实例间的互相通信能够及时发现故障的 Master，并从其它的 Replica 实例中选举出新的 Master。

图1-2 Redis 集群模式



1. Redis 集群模式的优点和缺点

Redis 集群模式相对于 Redis 单机模式的优缺点如下：

- Redis集群模式的优点
    - 水平扩缩容：通过增加新的Master实例到集群中，并移动一定数量的Slot到该实例上，即可实现Redis集群扩容；将Master实例上包含的Slot全部移动到其他Master实例上，并删除该实例，即可实现Redis集群缩容。
    - 读写分离：集群中Replica实例会全量备份对应Master实例的数据。Client写数据时可连接Master实例，读数据时可连接Replica实例。
    - 高可用：当Master实例意外宕机时，它的Replica实例会通过选举成为新的Master实例，确保Redis集群稳定可用。
- Redis集群模式的缺点
    - 集群模式的Redis实例不再支持多数据库，只能使用数据库0。
    - 不支持批量操作的命令。
    - 主从架构使内存占用翻倍。

1.3 应用场景

- 消息系统

  消息队列是大型网站常用的中间件，主要用于业务解耦、流量削峰及异步处理。Redis提供了发布/订阅及阻塞队列功能，可以实现一个简单的消息队列系统。
- 最新列表

  Redis的列表结构，可以使用push命令在列表头部插入一个内容ID作为关键字或使用trim命令来限制列表的数量。列表永远为N个ID，可直接根据ID查找对应的内容，无需查询最新的列表。
- 计数系统

  统计电商网站商品的浏览量、视频网站的播放量等。Redis提供的incr命令可以实现计数器功能，内存操作，性能非常好，适用于各类计数场景。

2 快速入门

2.1 组件安装

说明

- 在 Hadoop 集群或 Redis 集群中，安装 Redis 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- 在大数据平台当前版本中，支持部署 Redis 专有实例，即在 Redis 专有实例上仅能部署 Redis 组件。
- Redis 会在每个选择的节点上部署 Redis Node 进程，该进程会根据配置参数 redis.ports 和 redis.instance.ports 启动对应个数的 Redis 实例。

大数据平台中，部署 Redis 包括以下两种方式：

- 在集群类型为 Hadoop 的大数据集群中安装 Redis，此时在集群中即可以安装 Redis 组件，还可以同时部署其他大数据组件。
- 在集群类型为 Redis 的大数据集群中安装 Redis，此时在集群中仅能部署 Redis 及其依赖的组件。

2.1.1 组件数据存放位置

根据现场磁盘分区方案和挂盘方案的不同，Redis 安装完成后，必须对组件的数据目录配置结果进行检查，否则组件可能会使用异常。关于组件数据目录对应配置项的检查说明，如表 2-1 所示。

表2-1 组件需要检查的配置项

组件	是否需要检查	被影响的配置项	如何解决
Redis	是（配置项的参数值默认使用某一个挂载路径）	redis.dir	此目录为数据目录，检查此配置项的值时，需关注： • 不允许存在非数据目录 • 若现场数据目录是自定义的，则需要配置为对应的数据目录 • 仅支持单路径挂载使用，所以只允许配置一个数据目录

2.1.2 查看组件的日志信息

表2-2 组件日志路径说明

组件	日志路径
Redis	/var/de_log/redis

2.2 运行状态监控

1. 查看组件详情

说明

Redis集群实例数=配置参数redis.ports中分配的端口数量*Redis Node的数量。

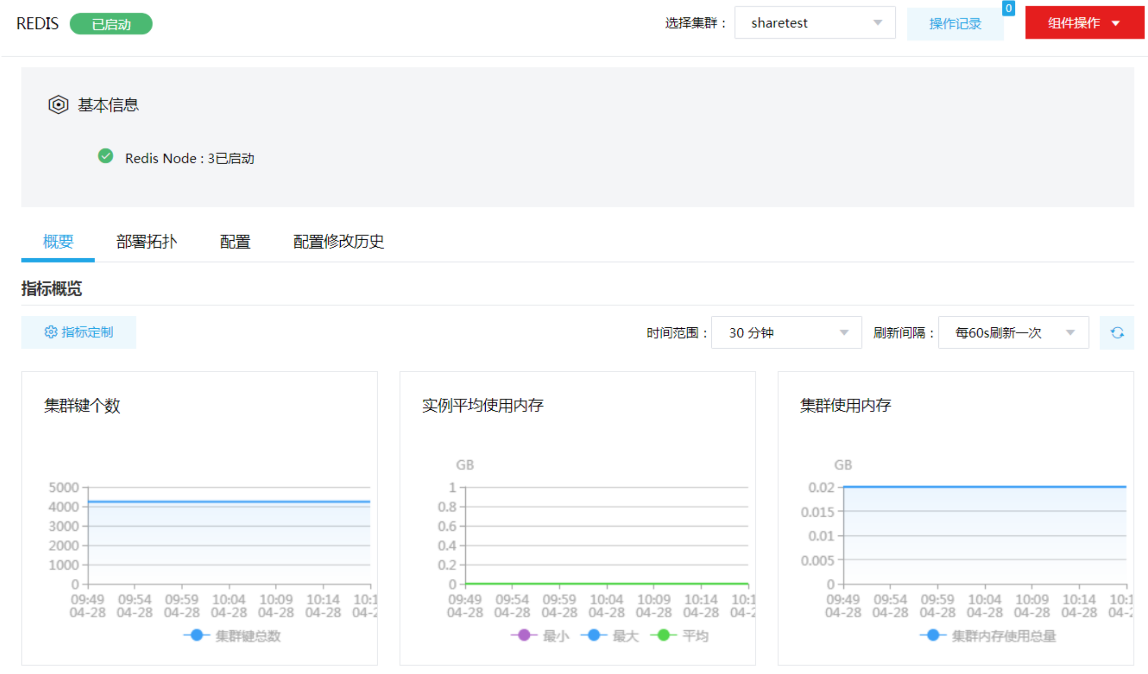
进入Redis组件详情页面，如图2-1所示。组件详情页面主要展示基本信息、概要、部署拓扑、配置详情和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

- 基本信息：展示组件的基本配置信息，比如：Redis Node节点启动状态、概要信息、部署拓扑等，以便于快速了解组件。
- 概要：在组件详情的[概要]页签，展示组件相关指标项的概要信息，可自定义进行指标定制，或设置指标信息展示的时间范围和刷新间隔。
- 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

  【说明】：进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机IP不同。
- 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息，还可对比查看不同版本配置项差异。
- 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。
- 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载Client、查看操作历史等。

图2-1 组件详情



2. 组件检查

执行 Redis 组件检查时，主要检查 Redis 集群中所有实例是否可正常连接，若 Redis 组件检查成功则表示 Redis 所有实例均可正常连接。

在使用集群的过程中，根据实际需要，可对 Redis 组件执行组件检查的操作。

(1) 组件检查的方式有以下三种，任选其一即可：

- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中某组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中某组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 然后在弹窗中进行确定后，即可对该组件进行检查。

(3) 组件检查结束后，检查弹窗中会显示组件检查成功或失败的状态，如图 2-2 所示，表示该组件检查成功，可正常使用。

图2-2 组件检查

(4) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看操作类型为“Redis Service Check”的组件操作执行详细信息，根据操作日志可判断组件检查的具体情况。

图2-3 组件检查日志详情

任务日志

```
操作类型：REDIS Service Check/SERVICE_CHECK REDIS
stderr:/var/lib/ambari-agent/data/errors-322.txt
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
stdout:/var/lib/ambari-agent/data/output-322.txt
hostname: diaonode1.hde.com
ambari_server: 10.121.47.165
master_node_ip: 10.121.47.165
redis_node_hosts: [u'diaonode1.hde.com', u'diaonode2.hde.com', u'diaonode3.hde.com']
host_ips: ['10.121.47.165', '10.121.47.166', '10.121.47.167']
Waiting for the Redis to check...
==Successfully connected to Redis 10.121.47.165:7001
==Successfully connected to Redis 10.121.47.165:7002
==Successfully connected to Redis 10.121.47.166:7001
==Successfully connected to Redis 10.121.47.166:7002
```

确 定

2.3 Client下载/安装/使用/卸载

大数据集群提供了下载 Redis Client 的功能。在客户端节点上安装 Redis 的 Client 后，即可直接连接集群中的 Redis，进行组件命令等客户端操作。

2.3.1 下载 Client 安装包

注意

- 组件 Client 下载方式较多（比如：集群支持批量下载 Client 操作，租户的 Redis 组件也支持下载 Client），下载要求也存在差异，但关于安装、使用、卸载等操作均类似，更多详情请参见产品在线联机帮助，本章节不做单独说明。
- 本章节仅以下载集群中单个组件 Client 为例进行操作指导。
- 组件 Client 下载成功后，将在服务器指定路径下或本地获得 Client 压缩包。
- Client 安装完成后，通过客户端可直接登录其对应的组件，在运维场景或业务场景中直接使用 shell 命令，或在应用程序开发场景中使用客户端中的样例工程。

下载 Client 安装包的步骤如下：

(1) 在集群管理的集群列表中，点击集群名称进入集群详情页面，在集群已安装的组件列表中单击 Redis 组件的<下载 Client>按钮，弹出下载 Client 窗口，如图 2-4 所示。

图2-4 下载 Client 窗口

下载 REDIS Client

客户端类型 完整客户端 仅配置文件

当你想要安装一个新的客户端时，选择此项。

服务器保存路径 /var/lib/ambari-server/data/tmp/

同步下载到本地

确定 取消

(2) 根据规划，选择下载的客户端类型，包括“完整客户端”和“仅配置文件”两种。其中：
   - 完整客户端，表示下载客户端的完整文件，适用于首次安装完整客户端的场景。
   - 仅配置文件，表示仅下载客户端的配置文件，适用于已下载并安装过完整客户端。

(3) 根据实际使用需要，可选择下载的 Client 压缩包仅保存在服务器指定目录下或同步下载到本地。
   - 保存到服务器上时，缺省保存在服务器的/var/lib/ambari-server/data/tmp/目录下（下载成功后，会有下载地址的详细提示信息），该路径不支持修改。
   - 若勾选同步下载到本地时，则下载的客户端文件会保存到用户本地下载目录下。

(4) 完成选择后，单击<确定>按钮，即可下载 Client 安装包。根据选择下载路径的不同或选择下载客户端类型的不同，得到的 Client 压缩包名称均不相同，详情请以实际为准。

2.3.2 安装 Client

注意

- 安装 Client 的节点必须能与大数据集群中的所有节点均网络互通。
- 安装 Client 的节点的操作系统必须与大数据集群中各节点的操作系统相同，否则会导致 Client 不完整无法正常使用。
- 下载的组件 Client 禁止安装在大数据平台管理节点上或者大数据集群的各个节点上，否则可能会导致已安装的服务或组件异常。
- 安装 Client 的节点必须启用 NTP 服务，且必须与大数据集群时间保持一致。
- 建议安装 Client 之前，关闭该节点的防火墙，否则可能会导致部分组件使用异常。
- 执行安装 Client 客户端的用户可以为 root 用户和所有被赋予权限的非 root 用户（比如权限为 755）。

与下载 Client 时可选择的客户端类型对应，安装 Client 也分为两种情况：

- 安装完整客户端
- Client 配置文件更新

1. 安装完整客户端

(1) 登录待安装 Client 的目标节点，将已下载的 Client 压缩包上传到任意路径下，进行解压。

(2) 配置映射关系，仅非 root 用户需要执行此操作，root 用户可跳过此步骤：

在解压得到的 Client 安装包文件夹中，查看 hosts 文件获得集群所有节点主机名和 IP 地址的映射关系，将集群各主机名和 IP 地址按照严格的映射关系，拷贝至该节点的“/etc/hosts”文件中。

(3) 启动安装

在解压得到的 Client 安装包文件夹下，启动安装脚本。命令如下：

```
./install.sh <安装路径> -o
```

注意

- 安装 Client 时，可自定义指定安装路径，若不指定则使用缺省安装路径。若安装目录不存在，则会自动创建；若安装目录已存在，则要求必须为空。当前版本中，Client 缺省安装目录为 /usr/hdp/3.0.1.0-187/。
- 安装 Client 时，还可以通过在末尾添加“-o”参数，限制安装后的组件 Client 仅能被执行安装的该用户使用；若不添加则表示任意用户均可使用此组件 Client。

2. 仅更新配置文件

(1) 登录待更新 Client 配置文件的目标节点，将已下载的 Client 压缩包上传到任意路径下。

(2) 在 Client 安装路径下，执行更新 Client 配置文件的命令。命令如下：

```
./refreshConfig.sh <Client 安装路径> <Client 配置文件包存放路径>/<Client 配置文件包>
```

2.3.3 访问组件

在集群外节点上，成功安装组件的 Client 之后，即可直接连接大数据集群中的该组件，执行相关管理、维护等操作。需要注意：

- Client 安装完成后，进入安装目录，配置客户端环境变量（仅针对 session 有效，建议每次使用 Client 客户端时均配置一遍环境变量），命令如下：

```
source bigdata_env
```

说明

在集群外安装 Client 的节点上，并没有集群用户或组件超级用户的用户信息。若想要通过集群用户或组件超级用户访问组件，则需要首先使用 useradd 命令添加对应用户。

2.3.4 使用客户端 Client

Client 使用示例如下：

(1) 向 redis 中插入数据数据。执行如下命令：

```
./redis-cli -c -h 10.121.65.36 -p 7001 -a CloudOS5#DE3@Redis set 'test' 'test'
```

(2) 获取数据。执行如下命令：

```
./redis-cli -c -h 10.121.65.36 -p 7001 -a CloudOS5#DE3@Redis get 'test'
```

2.3.5 卸载 Client 客户端

大数据集群卸载或重装之后，之前安装的 Client 客户端将不可用，此时需要卸载 Client。执行卸载 Client 的用户可以为 root 用户或所有被赋予权限的非 root 用户。

(1) 登录安装 Client 的节点，在 Client 的安装目录下启动卸载。

```
./uninstall.sh
```

(2) 卸载脚本执行结束后，安装目录将自动删除，此时 Client 卸载成功。

2.4 快速使用指导

以下示例场景为：在集群的某一节点上，安装了 Redis 组件。通过控制台，进行 Redis 组件的基本操作。

说明

以下操作需要:

- 登录 Redis 任一节点，切换至/usr/hdp/3.0.1.0-187/redis/bin 目录下。
- 切换至拥有 Redis 操作权限的用户，例如 root 或 redis 用户。
- 以下操作示例所使用到的密码为 redis.password 参数的配置值，请根据实际修改，默认密码为 CloudOS5#DE3@Redis。

Redis 既可以通过集群用户连接，又可以通过组件超级用户连接。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
  - 集群超级用户：仅 Hadoop 集群拥有集群超级用户。新建 Hadoop 集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。
  - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如 Redis 组件的 default 用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.4.1 连接 Redis

使用 redis-cli 命令连接 Redis。

示例命令如下：

- 组件超级用户（default）连接 Redis

  redis-cli -c -h 10.121.65.36 -p 7001 -a CloudOS5#DE3@Redis

  其中：
  - -c：连接 redis 集群时使用，用于实现连接实例跳转，避免“MOVED”错误。
  - -h：连接实例的 IP 地址。
  - -p：连接实例的端口。
  - -a：连接实例的密码。

```
[root@qcj36 ~]# redis-cli -c -h 10.121.65.36 -p 7001 -a C
loudOS5#DE3@Redis
Warning: Using a password with '-a' or '-u' option on the
 command line interface may not be safe.
10.121.65.36:7001>
```

- 普通用户（userredis）连接 Redis

  redis-cli -c -h 10.121.65.36 -p 7001 -c --user userredis --pass admin@123

  其中：
  - -c：连接 redis 集群时使用，用于实现连接实例跳转，避免“MOVED”错误。
  - -h：连接实例的 IP 地址。

- -p：连接实例的端口。
- --user：连接实例的用户名。
- --pass：连接实例用户名对应的密码。

```
[root@sharedev1 ~]# su userredis
sh-4.2$ redis-cli -c -h 10.121.68.131 -p 7001 -c --user userredis
--pass password@123
Warning: Using a password with '-a' or '-u' option on the command
line interface may not be safe.
10.121.68.131:7001>
```

2.4.2 基础操作

- 设置指定 *key* 的 value 值为字符串，并根据 *key* 值获取对应的 value 值。示例如下：

```
10.121.65.36:7001> set msg 'hello world'
-> Redirected to slot [6257] located at 10.121.65.37:7001
OK
10.121.65.37:7001> get msg
"hello world"
```

- 设置 hash field 的 value 值，并根据 hash field 值获取对应的 value 值。示例如下：

```
10.121.65.37:7001> hset myhash age 25
(integer) 0
10.121.65.37:7001> hget myhash age
"25"
```

- 在 *key* 对应的 list 的头部添加字符串元素，并根据 *key* 值获取对应的 value 值。示例如下：

```
10.121.65.37:7001> lpush mylist 'world'
-> Redirected to slot [5282] located at 10.121.65.37:7002
(integer) 1
10.121.65.37:7002> rpush mylist 'hello'
(integer) 2
10.121.65.37:7002> lpop mylist
"world"
10.121.65.37:7002> rpop mylist
"hello"
```

- 向名称为 *key* 的 set 中添加元素，并根据 *key* 获取对应的 value 值。示例如下：

```
10.121.65.38:7001> sadd fruits apple banana cherry
(integer) 3
10.121.65.38:7001> smembers fruits
1) "cherry"
2) "banana"
3) "apple"
```

- 向名称为 *key* 的 zset 中添加元素，并根据 *key* 值获取对应的 value 值的个数。示例如下：

```
10.121.65.38:7001> zadd myzset 1 'one'
-> Redirected to slot [1515] located at 10.121.65.37:7002
(integer) 1
10.121.65.37:7002> zadd myzset 2 'two'
(integer) 1
10.121.65.37:7002> zadd myzset 3 'three'
(integer) 1
10.121.65.37:7002> zcard myzset
(integer) 3
```

- 添加指定元素到 HyperLogLog 中，并返回给定 HyperLogLog 的基数估算值。示例如下：

```
10.121.65.37:7002> pfadd fruit 'apple'
(integer) 1
10.121.65.37:7002> pfadd fruit 'banana'
(integer) 1
10.121.65.37:7002> pfadd fruit 'cherry'
(integer) 1
10.121.65.37:7002> pfcount fruit
(integer) 3
```

- 设置有序集合 myzset 的过期时间为 10s，表示 10s 内该集合有效并可获取该集合内的值，10s 后服务器自动删除该集合。ttl *key* 表示以秒为单位，返回给定 key 的剩余生成时间。示例如下：

```
10.121.65.37:7002> zcard myzset
(integer) 3
10.121.65.37:7002> expire myzset 10
(integer) 1
10.121.65.37:7002> ttl myzset
(integer) 7
10.121.65.37:7002> zcard myzset
(integer) 0
10.121.65.37:7002> ttl myzset
(integer) -2
```

- 使用 ***type key*** 命令查看 value 对象的数据类型。示例如下：

```
10.121.65.37:7002> set msg 'hello world'
-> Redirected to slot [6257] located at 10.121.65.37:7001
OK
10.121.65.37:7001> type msg
string
10.121.65.37:7001>
10.121.65.37:7001> rpush numbers 1 2 3 4 5
(integer) 5
10.121.65.37:7001> type numbers
list
10.121.65.37:7001>
10.121.65.37:7001> sadd fruits apple banana cherry
-> Redirected to slot [14943] located at 10.121.65.38:7001
(integer) 0
10.121.65.38:7001> type fruits
set
10.121.65.38:7001>
10.121.65.38:7001> hmset profile name Tom age 25
OK
10.121.65.38:7001> type profile
hash
10.121.65.38:7001>
10.121.65.38:7001> zadd price 8.0 apple 4.0 cherry
-> Redirected to slot [5403] located at 10.121.65.37:7002
(integer) 2
10.121.65.37:7002> type price
zset
```

- 使用 ***object encoding*** 命令查看不同数据类型在 Redis 内部的存储方式。示例如下：

```
10.121.65.38:7001> object encoding msg
-> Redirected to slot [6257] located at 10.121.65.37:7001
"embstr"
10.121.65.37:7001> object encoding profile
-> Redirected to slot [16237] located at 10.121.65.38:7001
"ziplist"
10.121.65.38:7001> object encoding fruits
"hashtable"
```

3 使用指南

3.1 Shell命令

1. 字符串对象

字符串的编码对象可以是 int、raw 或者 embstr 类型，基本的操作示例和说明如表 3-1 所示。

表3-1 字符串操作

命令	说明	示例
SET	设置key（键）对应的value（值）	set name “hello world”
SETNX	设置key对应的value，如果key存在，返回0，不存在则会创建，并返回1	setnx name “hw”
SETEX	设置key对应的value，并指定此键值对应的有效期	setex haircolor 10 red
SETRANGE	用指定的字符串覆盖指定key所储存的字符串值，覆盖的位置从偏移量offset开始	setrange name 8 gmail.com
MSET	一次设置多个key的value，返回ok表示所有的值都设置了，返回0表示没有任何值被设置	mset key1 PP1 key2 PP2
MSETNX	一次设置多个key的value，返回ok表示所有的值都设置了，返回0表示没有任何值被设置，但是不会覆盖已经存在的key	msetnx key2 PP2_new key3 PP3
GET	获取key对应的value，如果key不存在则返回nil	get name
GETSET	设置key的value，并返回key的旧value	getset name PP_new
GETRANGE	获取指定key的value的子字符串	getrange name 0 6
MGET	一次获取多个key的value，如果对应key不存在，则对应返回nil	mget key1 key2 key3
INCR	对key中存储的数字增加1，并返回新的value。如果不是int类型的value会返回错误。如果key不存在，那么key的值会先被初始化为0，然后再执行INCR命令	incr age
INCRBY	对key中存储的数字增加指定的值，并返回新的value。如果不是int类型的value会返回错误。如果key不存在，那么key的值会先被初始化为0，然后再执行INCRBY命令	incrby age 5
DECR	对key中存储的数字减1，并返回新的value。如果不是int类型的value会返回错误。如果key不存在，那么key的值会先被初始化为0，然后再执行DECR命令	decr age
DECRBY	对key中存储的数字减为指定的值，并返回新的value。如果不是int类型的value会返回错误，如果key不存在，那么key的值会先被初始化为0，然后再执行DECRBY命令	decrby age 3
APPEND	如果key已经存在并且是一个字符串，APPEND命令会将value追加到key对应原value的末尾；如果key不存在，APPEND命令就会将指定key设为该value	append name @126.com

命令	说明	示例
STRLEN	获取指定key的value的长度	strlen name

2. 哈希对象

Redis hash 是一个 string 类型的 field 和 value 的映射表，特别适用于存储对象。哈希对象的编码可以是 ziplist 或者 hashtable，基本的操作示例和说明如表 3-2 所示。

表3-2 哈希操作

命令	说明	示例
HSET	设置hash field为指定value，如果key不存在，则先创建	hset myhash field1 hello
HSETNX	设置hash field为指定value，如果key不存在，则先创建，如果field已经存在，则返回0	hsetnx myhash field “Hello”
HMSET	同时设置hash的多个field	hmset myhash field1 Hello field2 World
HGET	获取指定的hash field	hget myhash field1
HMGET	获取全部指定的hash field	Hmget myhash field1 field2 field3
HINCRBY	给对应的hash field加上指定value	hincrby myhash field3 -8
HEXISTS	测试指定field是否存在	hexists myhash field1
HLEN	返回指定hash的field数量	hlen myhash
HDEL	删除hash key中的一个或多个指定域，不存在的域将被忽略	hdel myhash field1
HKEYS	返回hash的所有field	hkeys myhash
HVALS	返回hash的所有value	hvals myhash
HGETALL	获取某个hash中全部的field及value	hgetall myhash

3. 列表对象

- Redis 列表是简单的字符串列表，按照插入顺序排序。
- 列表可以添加一个元素到列表的头部（左边）或尾部（右边）。

列表对象的编码可以是 ziplist 或者 linkedlist，基本的操作示例和说明如表 3-3 所示。

表3-3 列表操作

命令	说明	示例
LPUSH	在指定key对应list的头部添加字符串	lpush mylist “world”
RPUSH	在指定key对应list的尾部添加字符串	rpush mylist “hello”
LINSERT	在指定key对应list的特定位置之前或之后添加字符串	linsert mylist before “world” “there”
LSET	在指定key对应list中指定下标的元素值（下标从0开始）	lset mylist -2 “five”
LREM	从指定key对应list中删除count个和value相同的元素，其中：	lrem mylist 2 “hello”

命令	说明	示例
	• count<0，按从尾到头的顺序删除 • count=0，删除全部 • count>0，按从头到尾的顺序删除	
LTRIM	保留指定key的值范围内的数据	ltrim mylist 1 -1
LPOP	从指定key对应list的头部删除元素，并返回删除元素	lpop mylist
RPOP	从指定key对应list的尾部删除元素，并返回删除元素	rpop mylist
RPOPLPUSH	从第一个list的尾部移除元素并添加到第二个list的头部，最后返回被移除的元素值，整个操作是原子的，如果第一个list是空或不存在则返回nil	rpoplpush mylist1 mylist2
LINDEX	返回指定key对应list中index位置的元素	lindex mylist 0
LLEN	返回指定key对应list的长度	llen mylist
LRANGE	用于获取key对应的list下标中的内容	lrange mylist 0 -1

4. 集合对象

- Redis 的 Set 是 String 类型的无序集合，且集合成员是唯一的。
- Redis 中集合是通过哈希表实现的，所以添加、删除、查找的复杂度都是 O(1)。

集合对象的编码可以是 intset 或 hashtable，基本的操作示例和说明如表 3-4 所示。

表3-4 集合操作

命令	说明	示例
SADD	向指定key的set中添加元素	sadd myset “hello”
SREM	删除指定key的set中的元素	srem myset “hello”
SPOP	随机返回并删除指定key的set中的一个元素	spop myset “one”
SDIFF	返回所有给定key与第一个key的差集	sdiff myset1 myset2
SDIFFSTORE	返回所有给定key与第一个key的差集，并将结果存为另一个key	sdiffstore myset3 myset1 myset2
SINTER	返回所有给定key的交集	sinter myset1 myset2
SINTERSTORE	返回所有给定key的交集，并将结果存为另一个key	sinterstore myset4 myset1 myset2
SUNION	返回所有给定key的并集	sunion myset1 myset2
SUNIONSTORE	返回所有给定key的并集，并将结果存为另一个key	sunionstore myset5 myset1 myset2
SMOVE	从第一个key对应的set中移除元素并添加到第二key对应的set中	smove myset2 myset6 three
SCARD	返回指定key的set的元素个数	scard myset
SISMEMBER	测试元素是否是指定key的set的元素	sismember myset one
SRANDMEMBER	随机返回指定key的set的一个元素，但是不删除元素	srandmember myset

命令	说明	示例
SMEMBERS	返回对象为set的key中所有元素	smembers myset

5. 有序集合对象

- Redis 有序集合对象和集合对象是 string 类型元素的集合对象，且不允许重复。
- 有序集合每个元素都会关联一个 double 类型的分数（score）。Redis 是通过分数为集合中的成员进行从小到大的排序。
- 有序集合的成员是唯一的，但分数（score）却可以重复。
- 集合是通过哈希表实现的，所以添加、删除和查找的复杂度都是 O(1)。

有序集合的编码可以是 ziplist 或 skiplist，基本操作示例和说明如表 3-5 所示。

表3-5 有序集合操作

命令	说明	示例
ZADD	向指定key的zset中添加元素。其中，score用于排序，如果该元素已存在，则会根据score更新元素的顺序	zadd myzset 1 “one” (1为score)
ZREM	删除指定key的zset中的元素	zrem myzset one
ZRANGE	返回指定key的zset（按score从小到大排序）中index从start到end的所有元素	zrange myzset 0-1 withscores
ZINCRBY	如果指定key的zset中元素已存在，则该元素的score会增加increment；不存在则向集合中添加该元素，其中score的值为increment	zincrby myzset 2 “one”
ZRANK	返回指定key的zset中元素的排名（按score从小到大排序）	zrank myzset two
ZREVRANK	返回指定key的zset中元素的排名（按score从大到小排序）	zrevrank myzset two
ZREVRANGE	返回指定key的zset（按score从大到小排序）中index从start到end的所有元素	zrevrange myzset 0 -1 withscores
ZRANGEBYSCORE	返回集合中score在给定区间的元素	zrangebyscore myzset 2 3 withscores
ZCOUNT	返回集合中score在给定区间的数量	zcount myzset 2 3
ZCARD	返回集合中元素个数	zcard myzset
ZSCORE	返回给定元素对应的score	zscore myzset two
ZREMRANGEBYRANK	删除集合中排名在给定区间的元素	zremrangebyrank myzset 3 3
ZREMRANGEBYSCORE	删除集合中score在给定区间的元素	zremrangebyscore myzset 1 2

6. HyperLogLog

- Redis HyperLogLog 可用于基数统计的算法，在输入元素的数量或者体积非常大时，计算基数所需的空间是固定的且是很小的。

- HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，即 HyperLogLog 不会返回输入的各个元素。

HyperLogLog 基本操作示例和说明如表 3-6 所示。

表3-6 HyperLogLog 操作

命令	说明	示例
PFADD	添加指定元素到HyperLogLog中	PFADD key “mysql”
PFCOUNT	返回给定HyperLogLog的基数估算值	PFCOUNT key
PFMERGE	将多个HyperLogLog合并为一个HyperLogLog	PFMERGE destkey sourcekey

3.2 权限访问控制

说明

仅开启“安全管理/权限管理”且运行正常的集群可使用角色管理功能。

权限管理是安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

在开启权限管理的集群中，用户需被指定角色（即赋予角色所拥有的 Redis 相关权限）后才能对 Redis 的 Key（键）资源执行操作。

为用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并指定用户的角色，用户即拥有角色所具有的权限。

3.2.1 权限说明

Redis 相关权限是通过在角色中配置对指定 Key（键）（支持使用通配符*模糊适配）的权限来实现的，权限包括：all、read、write、admin、string、list、hash、set、sortedset，其中 all 表示配置所有权限。若角色中同时配置了多个 Redis 权限时，权限范围取并集。

Redis 操作所需权限的对应关系如表 3-7 所示。

表3-7 Redis 权限说明

组件	权限类型	对应的常用操作
Redis	read	查询键等相关操作，如：get、hget、lrange、exists
	write	新增或删除键等相关操作，如：set、mset、rpush、xadd、del、hdel
	admin	实例的相关管理操作，如：acl、save、config、shutdown
	string	键的值为字符串对象时，对应的读写相关操作，如：get、set、decr、getset、append
	list	键的值为列表对象时，对应的读写相关操作，如：rpush、llen、sort、lrem

组件	权限类型	对应的常用操作
	hash	键的值为哈希对象时，对应的读写相关操作，如：hget、hset、hdel、hsetnx
	set	键的值为集合对象时，对应的读写相关操作，如：sscan、sadd、smove、sdiff
	sortedset	键的值为有序集合对象时，对应的读写相关操作，如：zadd、zrange、zcard、zunion、zscore
	all	支持以上所有操作

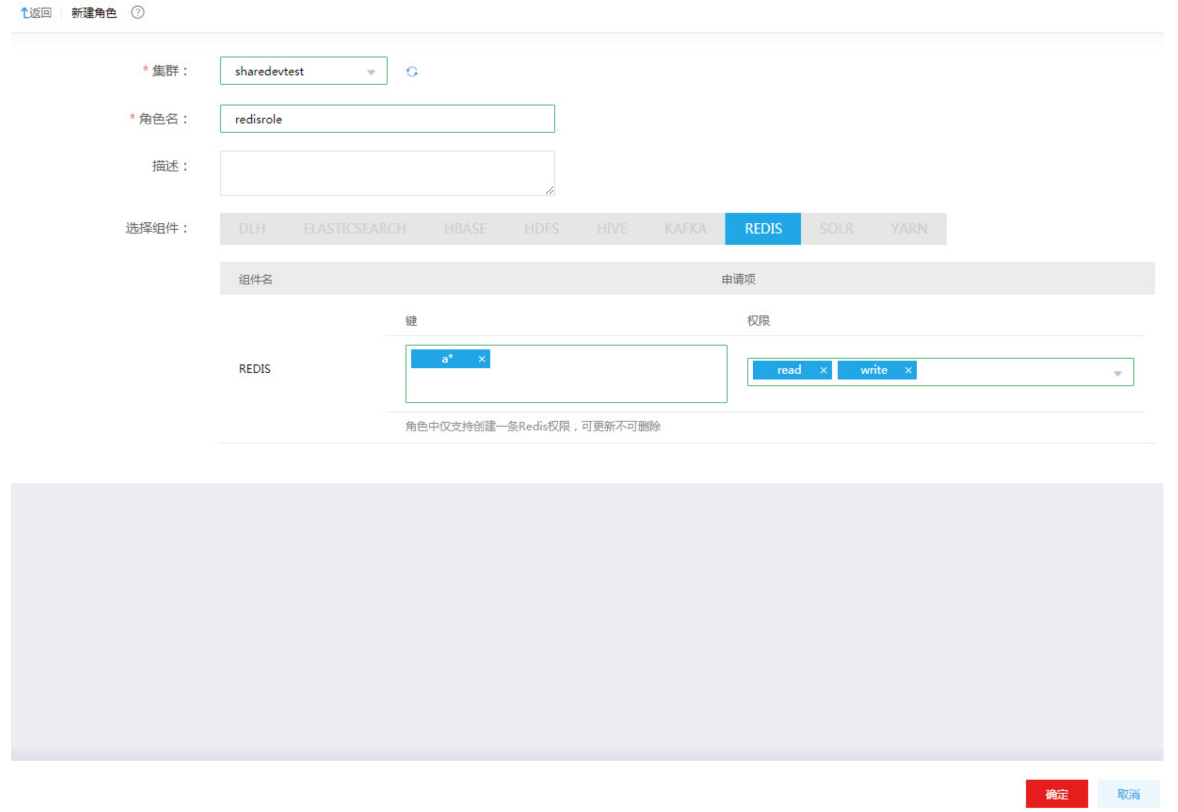
3.2.2 权限使用操作示例

示例：授予 Key 资源“a*”的“read、write”权限给“redisrole”角色，然后将“redisrole”角色绑定给“redisuser”用户，介绍 Redis 组件的权限访问控制。操作步骤如下：

(1) 新建角色

在[集群权限/角色管理]页面，创建角色 redisrole，不选择任何组件权限，如图 3-1 所示。

图3-1 新建 redisrole 角色



(2) 为用户绑定角色

在[集群权限/用户管理]页面，创建用户 redisuser，并为用户绑定角色 redisrole，如图 3-2 所示。

图3-2 用户 redisuser 绑定角色 redisrole

(3)	使用 redisuser 用户连接 Redis 执行读写操作，如图 3-3 所示，最后两个“error”表示 redisuser 用户没有对键 “b” 资源的读写权限。

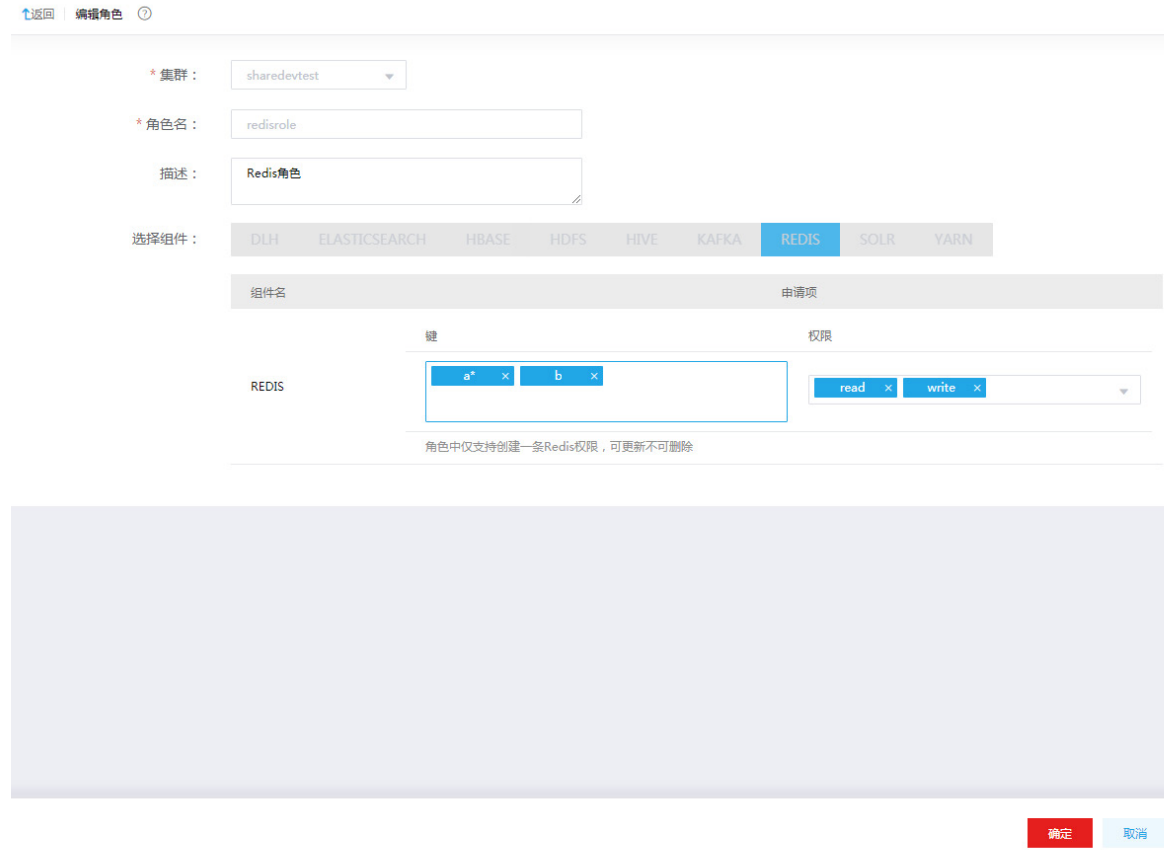
图3-3 通过 redisuser 用户读写 Redis 资源

```
-sh-4.2$ redis-cli -h 10.121.68.130 -p 7001 -c  --user redisuser --pass admin@123
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
10.121.68.130:7001> set a aa
-> Redirected to slot [15495] located at 10.121.68.131:7002
OK
10.121.68.131:7002> get a
"aa"
10.121.68.131:7002> set a2 aa2
OK
10.121.68.131:7002> get a2
"aa2"
10.121.68.131:7002> set b bb
(error) NOPERM this user has no permissions to access one of the keys used as arguments
10.121.68.131:7002> get b
(error) NOPERM this user has no permissions to access one of the keys used as arguments
```

(4)	修改角色授权

在[集群权限/角色管理]页面，修改角色 redisrole 的权限设置，授予 redisrole 角色对 “b” 资源的 read、write 权限，如图 3-4 所示。

图3-4 修改 redisrole 角色授权



(5) 更新成功后重新执行步骤(3)，使用 redisuser 用户读写“b”资源，如图 3-5 所示，命令执行成功。

图3-5 通过 redisuser 用户读写 “b” 资源成功

```
10.121.68.131:7001> set b bb
OK
10.121.68.131:7001> get b
"bb"
```

3.3 Redis集群扩容

Redis 集群扩容是指在某节点上新增安装 Redis Node 进程。

3.3.1 扩容场景

注意

Redis 单机模式不允许扩容。

随着业务量的增加，集群存储容量、服务能力无法满足业务需求时，需要考虑对 Redis 集群进行扩容。当 Redis 集群出现以下情况时，可以考虑扩容：

- 假定在系统中已设置告警“CPU 使用率”的监控阈值，当收到此告警且告警信息指明 Redis 所在主机节点出现 CPU 过载。
- 假定在系统中已设置告警“磁盘使用率”的监控阈值，当收到此告警且告警信息指明 Redis 所在主机节点的磁盘出现容量不足。
- 假定在系统中已设置告警“磁盘分区使用率”的监控阈值，当收到此告警且告警信息指明 Redis 所在主机节点的磁盘分区出现容量不足。
- 假定在系统中已设置告警“Redis 内存使用率”的监控阈值，当收到此告警且告警信息指明 Redis 节点的内存出现容量不足。

3.3.2 扩容前准备

1. 扩容规划

(1) 进行扩容分析，确定扩容场景。

(2) 在大数据集群中新增安装 Redis Node 进程。

  - 如果集群中有节点没有安装 Redis Node，直接在集群节点中添加 Redis Node 进程。
  - 如果集群中所有节点均已安装 Redis Node，进行 Redis Node 扩容前则需要先添加主机。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Redis 组件的状态是否正常。

(2) 进入 Redis 组件详情页，查看 Redis 的部署拓扑，确保每个服务的状态正常，Redis Node 均处于“已启动”状态。

3.3.3 扩容约束

- 扩容节点操作系统版本与集群内部版本需保持一致。
- 扩容操作一旦开始，不支持中止。
- Redis 在进行扩容时请严格遵守主从比公式[(实例端口数 * 主机数) /(副本数 + 1) = 整数]，如不符合该主从比公式，会造成扩容失败。

3.3.4 扩容影响

- 一旦扩容失败，需要及时将扩容失败的节点剔除。
- 扩容成功后，Redis Node 集群的性能会得到增强。
- Redis 集群扩容期间，由于数据迁移，可能会导致业务数据读写异常，建议扩容期间停止业务。

3.3.5 扩容操作指导

注意

若集群中所有主机节点均已安装 Redis Node，进行 Redis Node 扩容前则需要先添加主机，然后再进行 Redis Node 扩容。如果集群中有扩容所需使用的主机，则可直接跳过该步骤。关于添加主机的操作指导，详情请参见产品在线联机帮助。

扩容操作步骤如下：

(1) 在 Redis 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-6 所示。

a. 选择进程及主机

在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

b. 部署进程

选择结束后单击<下一步：部署进程>按钮，直至部署进度条结束（不支持中止）。

c. 启动进程

部署进程结束后单击<下一步：启动进程>按钮，直至启动进度条结束（不支持中止）。

图3-6 添加进程

(3) 查看进程变化

Redis Node 扩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 Redis Node 安装数量的变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.3.6 扩容验证

(1) 登录大数据平台管理系统。
(2) 执行 Redis 组件检查，确保 Redis 组件可正常使用，详情请参考 2.2 2. 组件检查。
(3) 查看 Redis 组件部署拓扑，可看到已有新增的扩容节点。

3.4 Redis集群缩容

Redis 集群缩容是指删除某节点上已安装的 Redis Node。

3.4.1 缩容场景

Redis 集群缩容的场景主要有：

- 初始 Redis Node 节点规划不合理。
- 当 Redis 缩容后，也要能满足客户业务需求，所以具体缩容量以客户需求规划为主。

3.4.2 缩容前准备

1. 缩容规划

- 集群模式会自动迁移缩容节点上 Redis Node 进程所产生的数据。
- 如果是单机模式，不建议进行缩容；若必须缩容，则缩容前请先手动备份数据文件，数据文件的存储路径请查看组件配置（以上操作指导为 Hadoop 集群，独立集群不支持缩容操作）。

2. 环境检查

(1) 登录大数据平台管理系统，查看 Redis 组件的状态是否正常。
(2) 进入 Redis 组件详情页，查看 Redis 的部署拓扑，确保每个服务的状态正常，Redis Node 均处于“已启动”状态。

3.4.3 缩容约束

- 缩容操作一旦开始，不支持中止。
- Redis 集群模式执行缩容前，请先手动停止缩容节点对应的 Redis Node 进程，并保证缩容后 Redis 集群 Redis Node 节点个数至少为 3 个。
- Redis 集群模式如果只有 3 个节点，则不允许删除 Redis Node。

3.4.4 缩容影响

- Redis 集群缩容期间，由于数据迁移，可能会导致业务数据读写异常，建议缩容期间停止业务。
- 缩容完成后，对应的主机 IP 地址需要在业务代码中进行删除。

3.4.5 缩容操作指导

说明

Redis Node 缩容操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在组件详情页面的[部署拓扑]页签下执行 Redis Node 缩容操作”为例进行说明，在主机详情页面执行 Redis Node 缩容操作，与其类似不再进行说明。

缩容操作步骤如下：

(1) 在 Redis 组件详情页面，选择缩容节点。

(2) 在组件详情页面[部署拓扑]页签下，选择已安装 Redis Node 进程且需要缩容的主机，然后单击该进程右侧操作中的<停止>按钮，停止 Redis Node。

(3) 待 Redis Node 停止成功后，如图 3-7 所示，在该进程右侧操作下拉框中选择<删除>按钮，即可完成 Redis Node 缩容。

图3-7 删除进程

概要 部署拓扑 配置 配置修改历史

进程名 ▾ 请输入搜索内容

进程名	进程状态	组件名	主机名	主机IP	机架	操作
Redis Node	已停止	REDIS	sharedev1.hde.com	10.121.68.131	/default-rack	开启 删除
Redis Node	已启动	REDIS	sharedev2.hde.com	10.121.68.132	/default-rack	停止 重启 删除
Redis Node	已启动	REDIS	sharedev3.hde.com	10.121.68.133	/default-rack	停止 重启 删除
Redis Node	已启动	REDIS	sharedev4.hde.com	10.121.68.134	/default-rack	停止 重启 删除

(4) Redis Node 缩容完成之后，在组件详情页面[部署拓扑]页签中可以看到 Redis Node 安装数量的变化以及状态。

(5) （根据实际情况选择）进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.4.6 缩容验证

(1) 登录大数据平台管理系统。

(2) 执行 Redis 组件检查，确保 Redis 组件可正常使用，详情请参考 2.2 2. 组件检查。

(3) 查看 Redis 组件部署拓扑，可看到相关缩容节点已经删除。

3.5 Redis多实例

注意

- Redis 多实例新增的端口请确保在大数据节点中未被占用，且端口 7001，7002 不允许缩容该实例。
- Redis 单机模式中执行多实例后的实例是相互独立的，且都是主实例。集群模式执行多实例后，会加入到原有集群中，且加入的实例有主从关系。
- Redis 在进行增减多实例时请严格遵守主从比公式[(实例端口数 * 主机数) / (副本数 + 1) = 整数]，如不符合该主从比公式，会造成操作失败。

Redis 多实例操作步骤如下：

(1) 在 Redis 组件详情页面，选择配置页签。

(2) 在组件详情页面[配置]页签下，选择基础配置，增加或减少 redis.instance.ports 端口，多个端口间使用英文逗号隔开，且需一次性成对增加或减少 redis.replicas+1 个相邻的端口，防止部分实例不具备主备关系。如图 3-8、图 3-9 和图 3-10 所示。

图3-8 副本数

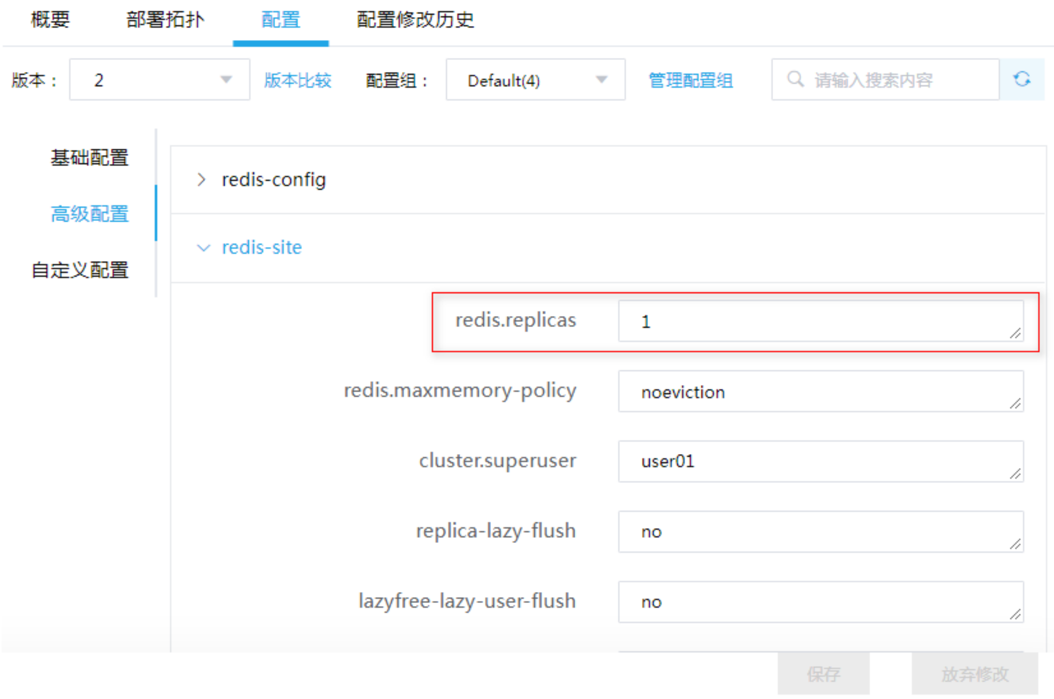


图3-9 增加多实例

图3-10 减少多实例

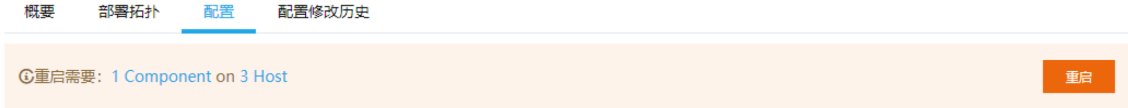
(3) 保存或放弃修改

单击<保存>按钮，可保存配置。如不需要前述步骤的修改，可单击<放弃修改>，取消修改。

(4) 重启组件

根据页面提示重新启动相关组件，如图 3-11 所示。

图3-11 重启组件



(5) 连接 Redis，查看多实例是否生效，如图 3-12 和图 3-13 所示。

图3-12 多实例进程（集群模式修改前后对比）

```
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
499f3df6431e7a8e2e8d4195c9cd01e545b76adb 10.121.65.46:7002@17002 slave 051c1491eddf9092fd906024ac7e9da144217e00 0 1619145469967 5 connected
051c1491eddf9092fd906024ac7e9da144217e00 10.121.65.48:7001@17001 master - 0 1619145467000 5 connected 10923-16383
1d848763222ab3d6ed167ee218833eed184f5d16 10.121.65.48:7002@17002 slave f7065c1fa2167cfdeadee80671b5cf4f5d3c28b6 0 1619145468965 3 connected
f7065c1fa2167cfdeadee80671b5cf4f5d3c28b6 10.121.65.47:7001@17001 master - 0 1619145468000 3 connected 5461-10922
6d5661d85614527892160c1e9389835532019455 10.121.65.47:7002@17002 slave f322ecb9a1e90c2768e192bf177608108047b90c 0 1619145468000 1 connected
f322ecb9a1e90c2768e192bf177608108047b90c 10.121.65.46:7001@17001 myself,master - 0 1619145469000 1 connected 0-5460
You have new mail in /var/spool/mail/root
```

```
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
01c9457e495a233d53cad02f503103fd1f2f40e3 10.121.65.47:7004@17004 master - 0 1619145816791 8 connected 5461-8192
61f90481ea101166bf10c6a47f3bf386d4d26406 10.121.65.48:7004@17004 master - 0 1619145817795 9 connected 10923-13653
9c6d88e72bd7f2085f8804a1ac886b5155f92260 10.121.65.47:7003@17003 slave 051c1491eddf9092fd906024ac7e9da144217e00 0 1619145818802 5 connected
6d5661d85614527892160c1e9389835532019455 10.121.65.47:7002@17002 slave f322ecb9a1e90c2768e192bf177608108047b90c 0 1619145813000 1 connected
499f3df6431e7a8e2e8d4195c9cd01e545b76adb 10.121.65.46:7002@17002 slave c2f73c119e55101445b3848d8b711dae4298ffde 0 1619145816000 10 connected
241a9840d28e379aad59fe9d2335dc6140194dea 10.121.65.46:7003@17003 slave 61f90481ea101166bf10c6a47f3bf386d4d26406 0 1619145817000 9 connected
2924d5131adb872cc9c40efc6d51c0d6177b5fdd 10.121.65.48:7003@17003 slave f7065c1fa2167cfdeadee80671b5cf4f5d3c28b6 0 1619145815785 3 connected
c2f73c119e55101445b3848d8b711dae4298ffde 10.121.65.46:7004@17004 master - 0 1619145815000 10 connected 0-2730
f322ecb9a1e90c2768e192bf177608108047b90c 10.121.65.46:7001@17001 myself,master - 0 1619145813000 1 connected 2731-5460
1d848763222ab3d6ed167ee218833eed184f5d16 10.121.65.48:7002@17002 slave 01c9457e495a233d53cad02f503103fd1f2f40e3 0 1619145814000 8 connected
f7065c1fa2167cfdeadee80671b5cf4f5d3c28b6 10.121.65.47:7001@17001 master - 0 1619145814173 3 connected 8193-10922
051c1491eddf9092fd906024ac7e9da144217e00 10.121.65.48:7001@17001 master - 0 1619145813000 5 connected 13654-16383
```

图3-13 多实例进程（单机模式修改前后对比）

```
root     16067     1  0 10:56 ?        00:00:05 /usr/hdp/current/redis-node/bin/redis-server 0.0.0.0:7001
root     16101     1  0 10:56 ?        00:00:05 /usr/hdp/current/redis-node/bin/redis-server 0.0.0.0:7002
root     16354 14818  0 10:58 ?        00:00:00 /redis_exporter/redis-exporter --redis.password CloudOS5#DE3@Redis
root     19713 22775  0 11:25 pts/3    00:00:00 grep --color=auto redis
[root@qcj46 ~]#
[root@qcj46 ~]#
[root@qcj46 ~]# redis-cli  -p 7001 -h 10.121.65.46 -a CloudOS5#DE3@Redis info Replication
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
# Replication
role:master
connected_slaves:0
master_failover_state:no-failover
master_replid:90335cb8efad0cc86a12756933bf031c4780e318
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

```
root      16354 14818  0 10:58 ?        00:00:00 /redis_exporter/redis-exporter --redis.password CloudOS5#DE3@Redis
root      20058     1  0 11:26 ?        00:00:00 /usr/hdp/current/redis-node/bin/redis-server 0.0.0.0:7001
root      20100     1  0 11:26 ?        00:00:00 /usr/hdp/current/redis-node/bin/redis-server 0.0.0.0:7002
root      20135     1  0 11:26 ?        00:00:00 /usr/hdp/current/redis-node/bin/redis-server 0.0.0.0:7003
root      20169     1  0 11:26 ?        00:00:00 /usr/hdp/current/redis-node/bin/redis-server 0.0.0.0:7004
root      20218 22775  0 11:26 pts/3    00:00:00 grep --color=auto redis
[root@qcj46 ~]#
[root@qcj46 ~]#
[root@qcj46 ~]#
[root@qcj46 ~]# redis-cli  -p 7003 -h 10.121.65.46 -a CloudOS5#DE3@Redis info Replication
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
# Replication
role:master
connected_slaves:0
master_failover_state:no-failover
master_replid:285d17e481d68714afda346065954a8fa5607e6d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
[root@qcj46 ~]#
[root@qcj46 ~]# redis-cli  -p 7004 -h 10.121.65.46 -a CloudOS5#DE3@Redis info Replication
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
# Replication
role:master
connected_slaves:0
master_failover_state:no-failover
master_replid:5aa6beb43f19d35db40f0e3c9eb1f295f1dd1c1e
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

3.6 Java方式使用Redis

3.6.1 Redis 开发示例

说明

- Redis 调用结束时注意关闭连接实例，若为连接池则使用完成后需释放连接实例。
- 使用 Redis 单机模式和集群模式时代码会有区别，请注意区分。

1. 建立 maven 工程

Pom.xml 文件内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>java</groupId>
    <artifactId>redis</artifactId>
    <version>1.0-SNAPSHOT</version>

        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>3.5.1</version>
        </dependency>
```

```
    </dependencies>
</project>
```

2. 准备建立连接参数

- Redis 单机模式

//请根据实际环境修改配置

```
private static final String ADDR = "10.121.65.37";//服务器 IP 地址
private static final int PORT = 7004;//端口
private static final String NAME = "default";//登录用户
private static final String AUTH = "CloudOS5#DE3@Redis";//用户密码
private static final int TIMEOUT = 10000;//连接超时的时间
private static final int DEFAULT_DATABASE = 0;//设置默认数据库
private static JedisPool jedisPool = null;
private static Jedis jedis = null;
```

- Redis 集群模式

```
private static final String ADDR =
"10.121.65.36:7001,10.121.65.36:7002,10.121.65.37:7001,10.121.65.37:7002,10.121.65.
38:7001,10.121.65.38:7002";
private static final String NAME = "default";//登录用户
    private static String AUTH = "CloudOS5#DE3@Redis";//用户密码
    private static String CLIENTNAME = "clientname";
    private static int MAX_ACTIVE = 1024;//连接实例的最大连接数
    private static int MAX_ATTEMPTS= 3;
    //等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出
JedisConnectionException
    private static int MAX_WAIT = 10000;
    private static int SOCKET_TIMEOUT = 10000;
    private static int TIMEOUT = 10000;//连接超时的时间
private static JedisCluster jedis;
```

说明

以上示例中的参数（ADDR、PORT、AUTH）需要根据实际环境修改。

3. 建立与关闭连接

- Redis 单机模式

```
static {
        try {
            //创建连接池，新建连接
            JedisPoolConfig config = new JedisPoolConfig();
            jedisPool = new JedisPool(config, ADDR, PORT, TIMEOUT,NAME,
AUTH,DEFAULT_DATABASE);;
          jedis = jedisPool.getResource();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    //释放资源
    public static void returnResource(final Jedis jedis) {
        if(jedis != null) {
            jedis.close();
        }
}
//关闭连接池
public static void closeJedisPool(JedisPool jedisPool) {
    if(jedisPool != null) {
        jedisPool.close();
    }
}
```

- Redis 集群模式

```
static {
        try {
            //创建连接
            Set<HostAndPort> hostAndPortsSet = new HashSet<HostAndPort>();
            // 添加节点
            List<String> ipPorts = Arrays.asList(ADDR.split(","));
            for (String ipPort : ipPorts) {
                hostAndPortsSet.add(new HostAndPort(ipPort.split(":")[0],
Integer.parseInt(ipPort.split(":")[1])));
            }
            // Jedis 连接池配置
            JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
            //对拿到的 connection 进行 validateObject 校验，判断是否可用
            jedisPoolConfig.setTestOnBorrow(true);
            jedis = new JedisCluster(hostAndPortsSet, TIMEOUT, SOCKET_TIMEOUT,
MAX_ATTEMPTS, NAME, AUTH, CLIENTNAME, jedisPoolConfig);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //关闭连接
    public static void closeJedisPool(JedisCluster jedis) {
        if(jedis != null) {
            try {
                jedis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

4. 基本数据操作

说明

以下数据操作不区分 Redis 单机模式或集群模式。

```
public static void testString() {
        //读写字符串对象数据
        String key1 = "foo1";
        String value1 = "bar1";
        jedis.set(key1, value1);
        System.out.println(key1 + ":" + jedis.get(key1));
    }

    public static void testStringExpire() throws InterruptedException {
        //写数据时，为 key 设置过期时间
        String key = "hello1";
        String value = "world1";
        int seconds = 10;
        jedis.setex(key, seconds, value);
        System.out.println(key + ":" + jedis.get(key));
        Thread.sleep(seconds*1000);
        System.out.println(key + " isExists:" + jedis.exists(key));
    }

    public static void testList(){
        //操作列表对象
        String key = "City";
        //存储数据到列表中
        jedis.lpush(key, "BeiJing");//从 list 头部（左边）插入数据
        jedis.lpush(key, "ZhengZhou");
        jedis.rpush(key, "HangZhou");//从 list 尾部（右边）插入数据
        System.out.println("list size:" + jedis.llen(key));
        System.out.println("list:"+jedis.lrange(key,0, -1));//获取全部列表数据
    }

    public static void testHash() {
        //操作哈希对象，该类型适合存储实体类对象
        String key = "website";
        String field = "google";
        String value = "google.com";
        String field2 = "baidu";
        String value2 = "baidu.com";
        jedis.del(key);
        jedis.hset(key, field, value);
        jedis.hset(key, field2, value2);
        jedis.hexists(key, field);//判断是否存在
        jedis.hlen(key);//获取指定 hash 大小
```

```
        System.out.println(field + ":" + jedis.hget(key, field));
        System.out.println(field2 + ":" + jedis.hget(key, field2));
    }

    public static void testSet() {
        //操作集合对象
        String key = "language";
        jedis.del(key);
        String[] members = {"java", "ruby", "python", "ruby"};
        jedis.sadd(key, members);
        jedis.scard(key);//获取集合大小
        System.out.println(key + ":" + jedis.smembers(key));
    }
```

3.6.2 Redis 适用场景

1. 计数器

- 网页访问量

  代码示例：

```
jedis.incr("webpage_01");//网页访问量加 1，key 若不存在则会自动创建
jedis.get("webpage_01");//获取网页访问量
```

- 网站用户访问量

  DAU（日活跃用户数量）可以很好地反映网站的运营情况。计算 DAU 时每个用户只会算一次，普通计数器不再适用。若保存所有用户的访问信息以达到去重目的，当用户基数巨大时资源消耗大。Redis HyperLogLog 可以用来做基数统计的算法，能够使用很小的内存计算庞大的基数。

  代码示例：

```
jedis.pfadd("web_view_day1", "ip_user", "ip2_user2");
jedis.pfcount("web_view_day1");//获取估算值
```

2. 最新发布

例如，博客网站有最新文章、最新评论等获取最新数据的需求，而频繁的读写操作会造成数据库的极大压力。Redis 支持丰富的数据结构，其列表可以方便的实现该功能，类似场景都可以使用 Redis 的列表实现。

代码示例：

```
jedis.lpush("web_article", "article01");//将最新发布文章信息存入 Redis
jedis.lpush("user2_comment", " article03_comment03");//将 user2 文章的评论存入 Redis
jedis.lrange("web_article", 0, 9);//获取网站最新 10 篇文章
jedis.lrange("user2_comment", 0, 4);//获取 user2 收到的最新 5 条评论
```

3. 最热榜单

通过使用 Redis 有序集合可以方便的实现热榜功能，能够及时展示用户最感兴趣的话题。

代码示例：

```
jedis.zincrby("hot_list", 1, "article01");//文章点击量加 1
jedis.zcard("hot_list");//获取热榜数据量
jedis.zrevrange("hot_list", 0, 49);//获取点击量最高前 50
```

4. 分布式锁

利用 Redis 中 key 的唯一性，实现一个简单的分布式锁。通过为 key 设置过期时间，来避免锁无法释放的问题。

代码示例：

```
private static final String lock_key = "redis_lock";//锁键
private static final long lockLeaseTime = 10 * 1000;//锁过期时间
private static final long getLockTimeout = 30 * 1000;//获取锁超时时间
private static final String LOCK_SUCCESS = "OK";
private static final String SET_IF_NOT_EXIST = "NX";
private static final String SET_WITH_EXPIRE_TIME = "PX";

//获取锁
public static boolean lock(String id){
        Long start = System.currentTimeMillis();
        for(;;){
            //SET 命令返回 OK ，则证明获取锁成功
            String lock = jedis.set(lock_key, id, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME,
lockLeaseTime);
            if(LOCK_SUCCESS.equals(lock)){
                return true;
            }
            //否则循环等待，在 timeout 时间内仍未获取到锁，则获取失败
            long l = System.currentTimeMillis() - start;
            if (l>=getLockTimeout) {
                return false;
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    //释放锁
    public static boolean releaseLock(String id){
        String script =
                "if redis.call('get',KEYS[1]) == ARGV[1] then" +
                        "   return redis.call('del',KEYS[1]) " +
                        "else" +
                        "   return 0 " +
                        "end";
        Object result = jedis.eval(script, Collections.singletonList(lock_key),
                Collections.singletonList(id));
        if("1".equals(result.toString())){
            return true;
        }
```

```
    return false;
}
```

4 常见问题解答

4.1 调优

4.1.1 配置调优

可以通过修改 redis-site.xml 中的配置参数来调节 Redis 的性能，部分配置参数说明如表 4-1 所示。

表4-1 Redis 配置参数说明

参数名称	参数说明	缺省值
redis.maxmemory	Redis实例最大内存限制，该配置默认为最小系统内存的40%（Redis独立集群）或15%（Hadoop集群），根据业务需要可自行调整该配置	host_mem*percentage 例如：Hadoop集群的所有主机中内存最小值为8G，则该值(8*15%)向下取整值为1G
redis.maxmemory-policy	Redis内存使用达到最大内存限制时Key的淘汰策略，若Redis写满后不淘汰数据，将会无法写入新数据。根据业务需要可选择使用以下策略： volatile-lru：从已设置过期时间的数据集合中淘汰最近很少使用的数据 volatile-ttl：从已设置过期时间的数据集合中淘汰即将过期的数据 volatile-random：从已设置过期时间的数据集合中随机淘汰数据 volatile-lfu：从已设置过期时间的数据集合中淘汰不常用的数据 allkeys-lfu：从所有数据中淘汰不常用的数据 allkeys-lru：从所有数据集合中淘汰最近很少使用的数据 allkeys-random：从数据集合中随机淘汰数据 noenviction：禁止淘汰数据	noenviction
redis.ports	当前节点部署实例个数，单节点部署多实例可提高资源利用率	7001,7002
redis.replicas	集群模式Redis副本数。副本实例会全量复制主实例数据，当主实例挂掉时副本实例会代替主实例继续工作。副本数越多，集群会更稳定，但性能会下降很多	1
redis.repl-backlog-size	复制积压缓存区大小。当集群模式数据量较大且读写频繁时，可调大此值，以提高主从实例的稳定性	1mb
stop-writes-on-bgsave-error	当启用了RDB且最后一次后台数据持久化保存失败，Redis是否停止接受写操作。建议设置成yes，防止Redis组件重启重新加载rdb文件时数据丢失	yes
cluster-allow-reads-when-down	是否允许集群在宕机时读取	no
cluster-replica-no-failover	是否禁止当主节点挂掉时，让从节点不能竞选为主节点	no
cluster-require-full-coverage	要求所有主节点正常工作，且所有hash slots被分配到工作的主节点，集群才能提供服务，如果仅需要一部分hash slots即可响应请求，则	yes

参数名称	参数说明	缺省值
	设置为no	
redis.appendonly	开启或关闭主节点的aof持久化功能，其中： • yes：默认值，开启 aof 持久化 • no：关闭 aof 持久化	yes
redis.appendfsync	aof写入磁盘的方式： • no：Redis 不做持久化，将数据交给操作系统来处理。此情况下，Redis 的处理速度加快，但数据不安全 • everysec：每秒执行一次持久化。此情况下，Redis 的处理速度足够快，并且在故障时只会丢失 1 秒钟的数据 • always：每次有新命令追加到.aof 文件时就执行一次持久化。此情况下，Redis 的处理速度较慢，但数据安全	everysec

4.1.2 其它调优

1. 建议对写入 Redis 的数据设置过期时间

一般主机内存资源不会特别大，若业务端不及时删除数据，一段时间后 Redis 就会将内存写满。建议在写入数据时根据业务需要来设置过期时间，数据到期后会自动删除，以确保 Redis 有足够的空间存储新的数据。

2. 精简 Redis 的 key 与 value

在保证可读性的基础上尽量精简 key 的名称，对于 value，一些判断的值可用 1、0 代替，这样能有效节省内存的使用。

3. 避免使用 keys *命令

keys *这个命令是阻塞的，即操作执行期间，其它任何命令在 Redis 实例中都无法执行。当 Redis 中 key 数据量大时影响会很大。实际业务场景建议使用 SCAN 来代替 keys *。

4.2 运维类问题

1. 通过 Redis 组件详情页面的配置栏修改配置项 stop-writes-on-bgsave-error 为 no，重启 Redis 组件该配置未生效（E5103P02 版本及之前版本）

问题现象：修改 stop-writes-on-bgsave-error 的 value 值为 no 后，重启 Redis 组件后配置项未生效，后端配置文件仍然为 yes。

原因分析：后端的配置项 stop-writes-on-bgsave-error 的 value 值被固化为 yes。

解决方法：

(1) 修改 ambari-server 节点 /var/lib/ambari-server/resources/stacks/HDP/3.0/services/REDIS/package/scripts/目录下的 status_params.py 文件（如果高可用，两个 ambair-server 节点都修改）。

stop-writes-on-bgsave-error yes 修改为 stop-writes-on-bgsave-error {{redis_stop_writes_on_bgsave_error}}

(2) 保存后重启 ambari-server（请确保所有业务允许被中断，方可执行该操作）

```
ambari-server restart
```

2. Redis 配置多实例且扩容后，出现 Redis 节点无法启动现象

问题现象：Redis 实例提示启动成功，且后台进程正常，但是组件界面仍显示该组件未启动，再次启动后台日志报进程已存在。

原因分析：组件自启动开启，所以手动启动 Redis 组件会偶现启动失败。

解决方法：首先关闭自启动按钮，kill 掉后台进程，然后再重新启动 Redis 组件。

3. 执行多实例，扩容或缩容主机操作后，告警管理中可能会出现“Redis 主副本关系异常”的告警，该告警提示用户 Redis 主副本关系异常，且该告警不会自动修复

问题现象：执行扩容或缩容主机操作后，告警管理中出现“Redis 主副本关系异常”告警。该告警是 Redis 的主副本关系异常所产生的告警，只有手动修复后，该告警才会消失。

原因分析：及时提示用户 Redis 主副本关系异常，防止该主机异常产生服务不可用现象。

解决方法：

手动修复（注意：手动修复过程中，可能会出现业务中断）的步骤如下：

(1) 在后台连接告警信息对应的 Redis 集群，并查看集群节点中对应主机上实例的对应关系。该示例中的集群是 sharedevtest，实例主从关系异常的节点为 sharedev1.hde.com（10.121.68.131），该主机主实例 7001，从实例是 7002。

```
[root@sharedev1 ~]# redis-cli -p 7001 -h 10.121.68.131 -a CloudOS5#DE3@Redis cluster nodes
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
6645839bd9457752966590739585a3140945c02d 10.121.68.132:7002@17002 slave 58f87d0552328153f5a3d670576b44c1e1418beb 0 1635408726466 5 connected
f6f351e5d7969192fdd606b07a656add1c064473 10.121.68.131:7002@17002 slave 4ec43e2bcda500303b96a1a88506b50d07b2af73 0 1635408724462 1 connected
df2855f14286e9355d497e033443e5b84b8d68cb 10.121.68.133:7002@17002 slave 8b8e051859f869e84eff92794d0864d73ef0c942 0 1635408727468 3 connected
58f87d0552328153f5a3d670576b44c1e1418beb 10.121.68.133:7001@17001 master - 0 1635408726000 5 connected 10923-16383
8b8e051859f869e84eff92794d0864d73ef0c942 10.121.68.132:7001@17001 master - 0 1635408725464 3 connected 5461-10922
4ec43e2bcda500303b96a1a88506b50d07b2af73 10.121.68.131:7001@17001 myself,master - 0 1635408727000 1 connected 0-5460
```

(2) 通过 redis-cli -c -p *port* –h *host* –a *password* cluster replicate *node_id* 命令将当前实例（host:port）设置为指定实例（node_id 的类型为 master 实例）的从实例。具体参数请根据现场情况填充。

执行示例如下：

redis-cli -c -p 7002 -h sharedev2.hde.com -a CloudOS5#DE3@Redis cluster replicate 4ec43e2bcda500303b96a1a88506b50d07b2af73

redis-cli -c -p 7001 -h sharedev1.hde.com -a CloudOS5#DE3@Redis cluster replicate 58f87d0552328153f5a3d670576b44c1e1418beb

```
[root@sharedev1 ~]# redis-cli -p 7002 -h 10.121.68.132 -a CloudOS5#DE3@Redis cluster replicate 4ec43e2bcda500303b96a1a88506b50d07b2af73
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
OK
[root@sharedev1 ~]# redis-cli -p 7002 -h 10.121.68.131 -a CloudOS5#DE3@Redis cluster replicate 58f87d0552328153f5a3d670576b44c1e1418beb
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
OK
[root@sharedev1 ~]# redis-cli -p 7001 -h 10.121.68.131 -a CloudOS5#DE3@Redis cluster nodes
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
6645839bd9457752966590739585a3140945c02d 10.121.68.132:7002@17002 slave 4ec43e2bcda500303b96a1a88506b50d07b2af73 0 1635409787740 1 connected
f6f351e5d7969192fdd606b07a656add1c064473 10.121.68.131:7002@17002 slave 58f87d0552328153f5a3d670576b44c1e1418beb 0 1635409785000 5 connected
df2855f14286e9355d497e033443e5b84b8d68cb 10.121.68.133:7002@17002 slave 8b8e051859f869e84eff92794d0864d73ef0c942 0 1635409786738 3 connected
58f87d0552328153f5a3d670576b44c1e1418beb 10.121.68.133:7001@17001 master - 0 1635409785736 5 connected 10923-16383
8b8e051859f869e84eff92794d0864d73ef0c942 10.121.68.132:7001@17001 master - 0 1635409787000 3 connected 5461-10922
4ec43e2bcda500303b96a1a88506b50d07b2af73 10.121.68.131:7001@17001 myself,master - 0 1635409786000 1 connected 0-5460
[root@sharedev1 ~]#
```

4. 异常断电重启等不定因素可能会造成 Redis 组件中某个已停止的 Redis Node 进程在组件管理页面，成功开启后立即变为已停止，/var/de_log/redis 日志报“Unrecoverable error: corrupted cluster configfile”

原因分析：异常断电导致 Redis 集群信息文件不完整，Redis Node 进程启动时加载失败。

```
115852:M 06 Dec 2021 16:03:34.004 # Unrecoverable error: corrupted cluster config file.
118936:C 06 Dec 2021 16:09:37.626 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
118936:C 06 Dec 2021 16:09:37.627 # Redis version=6.2.1, bits=64, commit=b69fdab8, modified=1, pid=118936, just started
118936:C 06 Dec 2021 16:09:37.627 # Configuration loaded
118936:M 06 Dec 2021 16:09:37.632 * monotonic clock: POSIX clock_gettime
118936:M 06 Dec 2021 16:09:37.634 # Unrecoverable error: corrupted cluster config file.
11048:C 06 Dec 2021 16:37:45.048 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
11048:C 06 Dec 2021 16:37:45.048 # Redis version=6.2.1, bits=64, commit=b69fdab8, modified=1, pid=11048, just started
11048:C 06 Dec 2021 16:37:45.049 # Configuration loaded
11048:M 06 Dec 2021 16:37:45.050 * monotonic clock: POSIX clock_gettime
11048:M 06 Dec 2021 16:37:45.051 # Unrecoverable error: corrupted cluster config file.
29426:C 06 Dec 2021 16:47:20.053 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
29426:C 06 Dec 2021 16:47:20.053 # Redis version=6.2.1, bits=64, commit=b69fdab8, modified=1, pid=29426, just started
29426:C 06 Dec 2021 16:47:20.053 # Configuration loaded
29426:M 06 Dec 2021 16:47:20.055 * monotonic clock: POSIX clock_gettime
```

解决方法：根据具体的集群节点和日志信息，查看是哪个集群节点的端口实例配置文件异常，然后重命名该文件，再重新启动 Redis 组件或 Redis Node 进程。

例如：127.0.0.1 7001 节点的端口配置文件异常，解决步骤如下：

(1) 重命名 nodes_7001.conf 文件（该文件路径不同版本可能不太一样，请在[集群管理/组件管理]页面的[配置]页签中查找）

```
[root@management0 conf]# ls
nodes_7001.conf  nodes_7002.conf  redis7001.conf  redis7002.conf  redis.conf  users_7001.acl  users_7002.acl
[root@management0 conf]# mv nodes_7001.conf nodes_7001.conf.bak
[root@management0 conf]#
```

(2) 重启 redis 组件或者开启 redis 进程。

目 录

1 DLH 简介 1-1
1.1 DLH 概述 1-1
1.2 DLH 架构 1-1
1.3 功能特点 1-2
1.4 应用场景 1-2
2 快速入门 2-1
2.1 组件安装 2-1
2.1.1 查看组件的日志信息 2-1
2.2 运行状态监控 2-1
2.2.1 查看组件详情 2-1
2.2.2 组件检查 2-5
2.3 快速使用指导 2-8
2.3.1 非 Kerberos 环境 2-9
2.3.2 Kerberos 环境 2-10
2.3.3 Presto 访问接口 2-14
2.4 快速链接使用 2-19
2.4.1 配置组件快速链接 2-20
2.4.2 访问监控页面 2-21
3 使用指南 3-1
3.1 离线查询＆交互式查询 3-1
3.1.1 语法介绍 3-1
3.1.2 配置说明 3-1
3.1.3 示例 3-2
3.2 外部数据源 3-2
3.2.1 语法介绍 3-2
3.2.2 Hive 数据源 3-3
3.2.3 SeaSQL MPP 数据源 3-7
3.2.4 DataEngine MPP 数据源 3-8
3.2.5 MySQL 数据源 3-9
3.2.6 HBase 数据源 3-10
3.2.7 跨源分析 3-14
3.3 Hudi 存储格式 3-15

3.3.1 Hudi 存储格式相关术语 3-15
3.3.2 文件组织形式 3-15
3.3.3 存储类型和视图 3-16
3.3.4 Hudi 属性说明 3-17
3.3.5 SQL 操作 3-18
3.4 流 SQL 操作 3-26
3.4.1 保留关键字 3-26
3.4.2 流 SQL UDF 3-28
3.4.3 创建表 3-28
3.4.4 数据插入 3-32
3.4.5 数据查询 3-46
3.5 MLSQL 3-48
3.5.1 监督算法 3-48
3.5.2 非监督算法 3-57
3.5.3 模型选择 3-61
3.5.4 特征工程 3-64
3.5.5 Pipeline 机制 3-73
3.6 列加密 3-74
3.6.1 公钥加密 3-75
3.6.2 私钥加密 3-76
3.7 权限访问控制 3-78
3.7.1 权限操作示例 3-79
3.8 添加/删除进程 3-91
3.8.1 添加进程 3-91
3.8.2 删除进程 3-93
4 配置说明 4-1
4.1 DLH 常用配置 4-1
4.1.1 sparrow-default 配置 4-1
4.1.2 sparrow-thrift-sparkconf 配置 4-1
4.1.3 sparrow-hive-site-override 配置 4-2
4.1.4 dlh-site 配置 4-2
4.2 Presto 常用配置 4-3
4.2.1 node 配置 4-3
4.2.2 resource-groups-json 配置 4-3
4.2.3 resource-groups-properties 配置 4-4
4.3 Atlas 常用配置 4-4

4.3.1 application-properties 配置 ······ 4-4

5 常见问题解答 ······ 5-5

1 DLH 简介

1.1 DLH概述

DLH（数据湖仓库）结合数据湖和数据仓库的优势，在数据湖存储上实现了与数据仓库类似的数据结构和数据管理功能，提供“湖仓一体化”的能力。

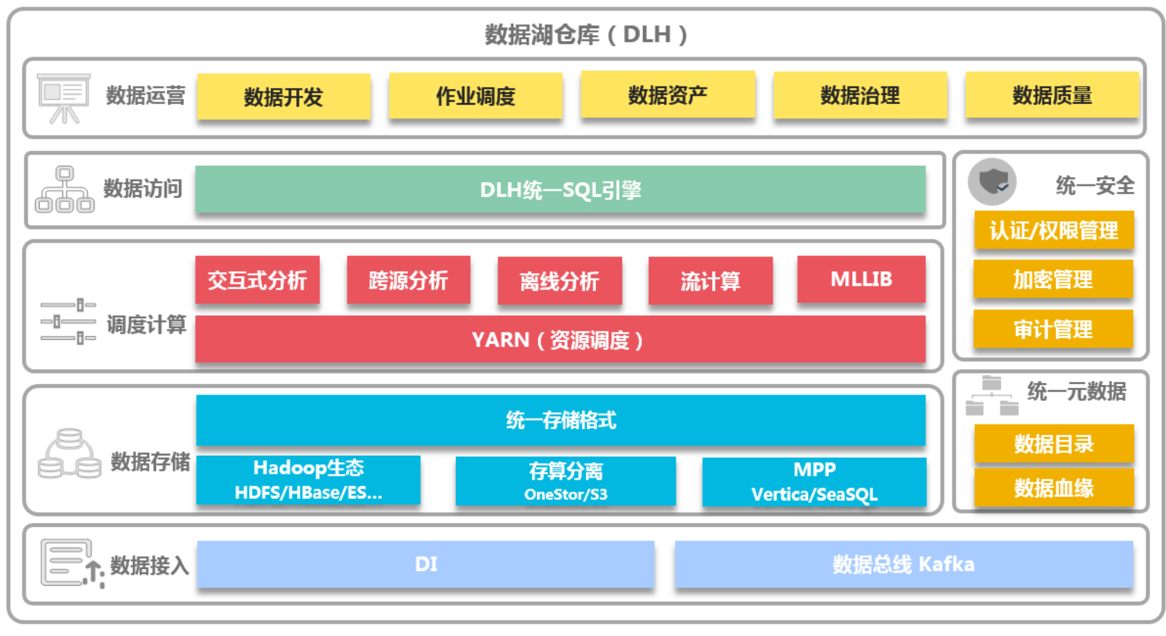
DLH 以 HDFS 和 ONEStor 作为数据湖的集中存储库，能够存储结构化、半结构化和非结构化的数据。借助数据集成服务能够将外部系统中数据接入到数据湖内，建立统一管理的数据目录、元数据信息及血缘关系展示等。同时，通过统一 SQL 接口能够对湖内数据进行离线查询、交互式分析、跨源分析、实时流计算以及机器学习算法训练等。

1.2 DLH架构

DLH 架构图如图 1-1 所示，说明如下：

- DLH 基于 Hive 提供统一 SQL 入口，不同场景下 SQL 语句底层执行引擎可自动进行切换，所有计算任务统一由 YARN 进行资源调度。
- 存储端支持多种存储格式和数据增量插入、增量查询等能力，并提供简单便捷的数据入湖工具。
- 通过统一的元数据管理界面，能够可视化管理库表结构信息及表数据量或外部数据源大小，并提供血缘关系展示等功能。
- 依赖大数据平台的认证/权限管理、加密管理、审计管理等模板保证 DLH 组件安全可靠。

图1-1 DLH 架构图



1.3 功能特点

DLH 的功能特点如下：

- 支持存储多种数据类型

  能够存储结构化、半结构化和非结构化数据以及流式数据访问。

- 完全兼容 Hive 标准

  Hive 作为 Hadoop 生态 SQL 事实标准，具有广泛的用户。数据湖仓库服务完全兼容 Hive 使用方式和语法，原有业务不需要做任何改动。

- 元数据管理

  DLH 对湖内元数据进行统一管理，能够以可视化的方式管理库表结构信息及表数据量或外部数据源大小等信息，并提供血缘关系展示等功能。

- 流批交互式融合

  通过 Hive 做为 SQL 统一入口，融合交互式查询、离线查询、流计算等引擎，并扩展 SQL 优化规则智能选择最佳执行引擎，提供大数据的能力、数据库的体验。

- 协同分析

  能够将湖内数据与外部数据源（如 MPP 数据库、关系型数据库、HBase 等）数据进行协同计算，无需搬迁数据即可进行联合分析。

- 数据安全管理

  依靠 Kerberos 和 Ranger 构建完善的企业级数据安权认证和权限管控。

1.4 应用场景

- 交互式分析

  在集群计算资源充足的情况下，DLH 提供 TB 级别的数据交互式查询及分析能力。适用于如报表或大盘查询等高吞吐低时延的场景。

- 跨源查询分析

  DLH 支持对不同数据中心的多种数据源进行跨源查询及联合分析，避免数据迁移。目前支持对 Hive、HBase、DataEngine MPP、SeaSQL MPP、MySQL 进行不同集群的跨源分析。

- 数据仓库

  数据仓库关注的是数据使用效率、大规模下的数据处理及管理能力。为保证数据和计算在湖和仓之间自由流动，DLH 基于 Hive 进行产品设计及功能开发，完全兼容 Hive 语法，基于 Hive 的数仓方案无需进行迁移适配。

- 大数据实时流计算

  DLH 兼容 Flink 并实现 FlinkSQL 远程提交运行，目前 DLH 支持 Flink 流 SQL 通过统一入口直接下发 Flink 集群运行且无性能损耗。

- 机器学习

  DLH 集成 Spark MLLib 和 GraphX 模块并实现 MLSQL 和 GraphSQL，可以方便的利用 DLH 计算能力对湖和仓中的数据进行机器学习算法训练和数据挖掘分析。

2 快速入门

2.1 组件安装

说明

- 关于在 Hadoop 集群中，安装 DLH 时的注意事项和操作指导以及部署过程中相关的参数说明等，详情请参见产品安装部署手册和在线联机帮助。
- DLH 依赖 Hadoop、Zookeeper、Kafka、HBase、Flink、Presto 以及 Infra_Solr 组件，安装 DLH 时要求完成上述组件的安装且组件状态正常，DLH 组件安装过程无其他特殊要求。DLH 安装完成后会自动同时安装 Atlas、Hudi 组件。

2.1.1 查看组件的日志信息

因 DLH 组件强依赖 Presto 和 Atlas 组件，所以进行 DLH 管理时可能会需要查看这 3 个组件的日志。其中：DLH 和 Presto 为业务组件，Altas 为系统组件。

表2-1 组件日志路径说明

组件	日志路径
DLH	• DLH Server 服务日志路径：/var/de_log/dlh/hive • DLH 流 SQL 服务日志路径：/var/de_log/flink-sql-gateway/user_hdfs
Presto	Presto跨源分析及交互式查询服务日志路径：/var/de_log/presto
Atlas	Atlas元数据服务日志路径：/var/de_log/atlas

2.2 运行状态监控

2.2.1 查看组件详情

因 DLH 组件强依赖 Presto、Atlas 和 Hudi 组件，所以进行 DLH 管理时可能会需要访问这 4 个组件的组件详情页面。其中：DLH、Presto、Hudi 为业务组件，Altas 为系统组件。

1. 查看 DLH 组件详情

进入 DLH 组件详情页面，如图 2-1 所示。组件详情页面主要展示基本信息、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

(1) 基本信息：展示组件的基本配置信息，比如：组件内各服务进程的状态（鼠标悬停在各进程名上时，可查看进程的安装地址）、DLH JDBC 连接 URL（支持点击 URL 直接进行复制）。

(2) 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

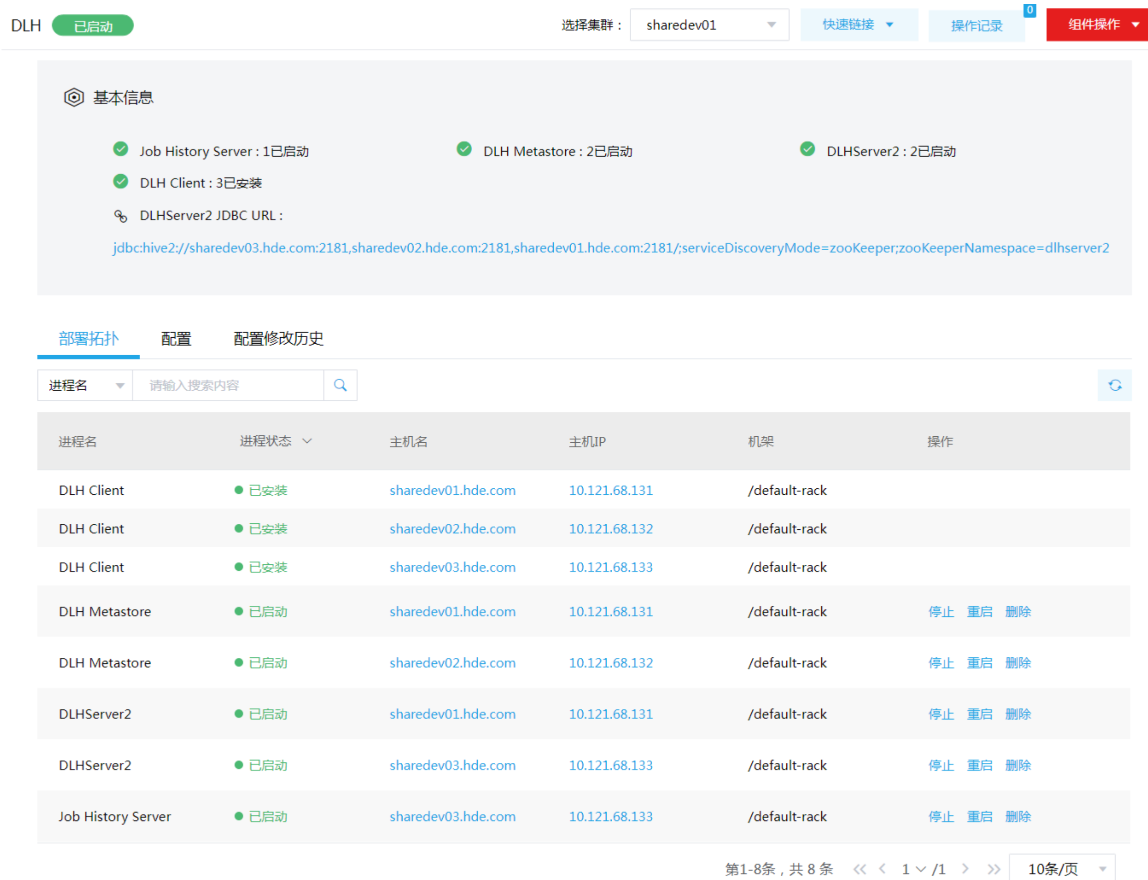
【说明】进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。

(3) 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。

(4) 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。

(5) 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、下载 Client、访问快速链接、查看操作记录等。

图2-1 DLH 组件详情



进程名	进程状态	主机名	主机IP	机架	操作
DLH Client	已安装	sharedev01.hde.com	10.121.68.131	/default-rack	
DLH Client	已安装	sharedev02.hde.com	10.121.68.132	/default-rack	
DLH Client	已安装	sharedev03.hde.com	10.121.68.133	/default-rack	
DLH Metastore	已启动	sharedev01.hde.com	10.121.68.131	/default-rack	停止 重启 删除
DLH Metastore	已启动	sharedev02.hde.com	10.121.68.132	/default-rack	停止 重启 删除
DLHServer2	已启动	sharedev01.hde.com	10.121.68.131	/default-rack	停止 重启 删除
DLHServer2	已启动	sharedev03.hde.com	10.121.68.133	/default-rack	停止 重启 删除
Job History Server	已启动	sharedev03.hde.com	10.121.68.133	/default-rack	停止 重启 删除

2. 查看 Presto 组件详情

进入 Presto 组件详情页面，如图 2-2 所示。组件详情页面主要展示基本信息、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

(1) 基本信息：展示组件的基本配置信息，比如：组件内各服务进程的状态（鼠标悬停在各进程名上时，可查看进程的安装地址）。

(2) 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启、删除等相关操作。

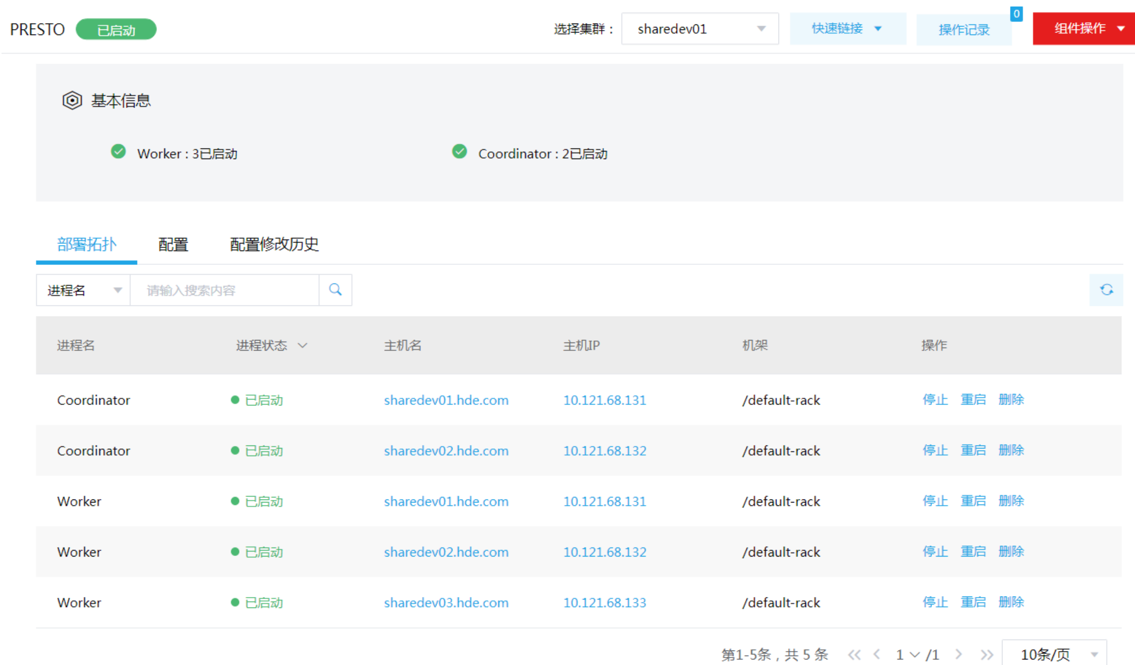
【说明】进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。

(3) 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。

(4) 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。

(5) 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、访问快速链接、查看操作记录等。

图2-2 Presto 组件详情



3. 查看 Atlas 组件详情

进入 Atlas 组件详情页面，如图 2-3 所示。组件详情页面主要展示基本信息、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

(1) 基本信息：展示组件的基本配置信息，比如：组件内各服务进程的状态（鼠标悬停在各进程名上时，可查看进程的安装地址）。

(2) 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行停止、重启等相关操作。

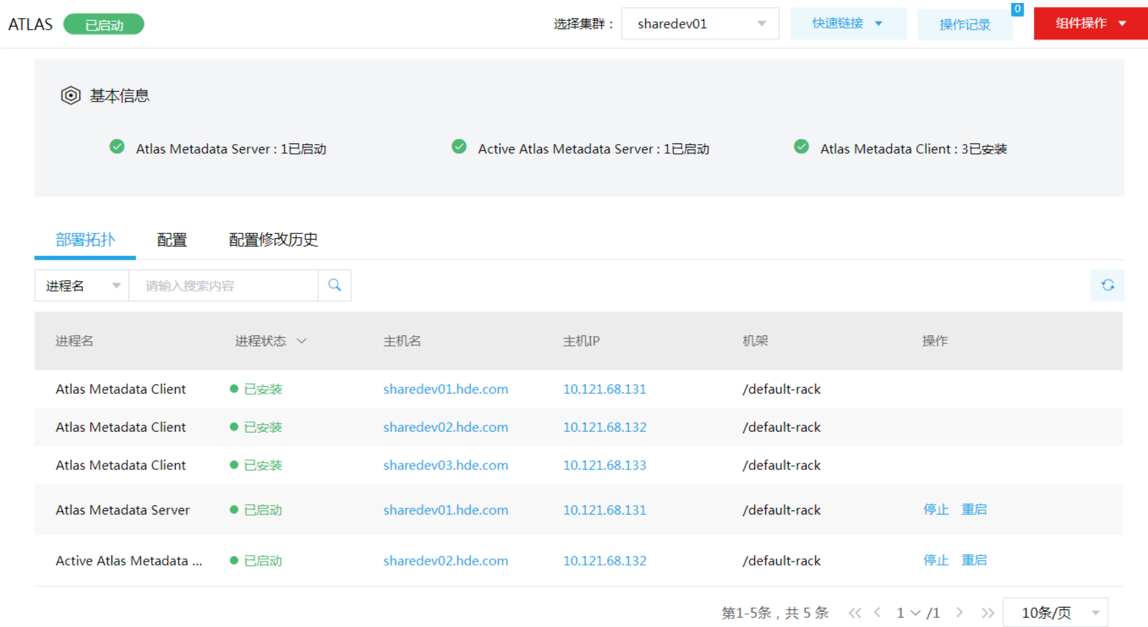
【说明】进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。

(3) 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。

(4) 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。

(5) 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、访问快速链接、查看操作记录等。

图2-3 Atlas 组件详情



4. 查看 Hudi 组件详情

进入 Hudi 组件详情页面，如图 2-4 所示。组件详情页面主要展示基本信息、部署拓扑、配置和配置修改历史等相关信息，同时可对组件或组件进程执行相关管理操作，可查看或修改组件的各配置项信息，也可查看组件的配置修改历史及当前使用配置版本。

主要功能如下：

(1) 基本信息：展示组件的基本配置信息，比如：组件内各服务进程的状态（鼠标悬停在各进程名上时，可查看进程的安装地址）。

(2) 部署拓扑：在组件详情的[部署拓扑]页签，可查看组件进程的安装详情以及运行状态详情，并可对组件执行删除等相关操作。

【说明】进程名：同一个进程可分别安装在多个主机节点上，所以进程列表中某一进程名可能重复出现，但同一进程名对应的主机名和主机 IP 不同。

(3) 配置：在组件详情的[配置]页签，可查看或修改组件各配置项的信息。

(4) 配置修改历史：在组件详情的[配置修改历史]页签，可查询组件的配置历史版本以及当前使用版本。

(5) 组件操作：在组件详情页面右上角，可对组件执行相关管理操作。比如：重启组件、添加进程、查看操作记录等。

图2-4 Hudi 组件详情

2.2.2 组件检查

因 DLH 组件强依赖 Presto 和 Atlas 组件，所以进行 DLH 管理时可能会需要对这 3 个组件执行组件检查操作。其中：DLH 和 Presto 为业务组件，Altas 为系统组件。

1. DLH 组件检查

集群在使用过程中，根据实际需要，可对 DLH 组件执行组件检查操作。

(1) 对 DLH 组件执行组件检查的方式有以下三种，任选其一即可：

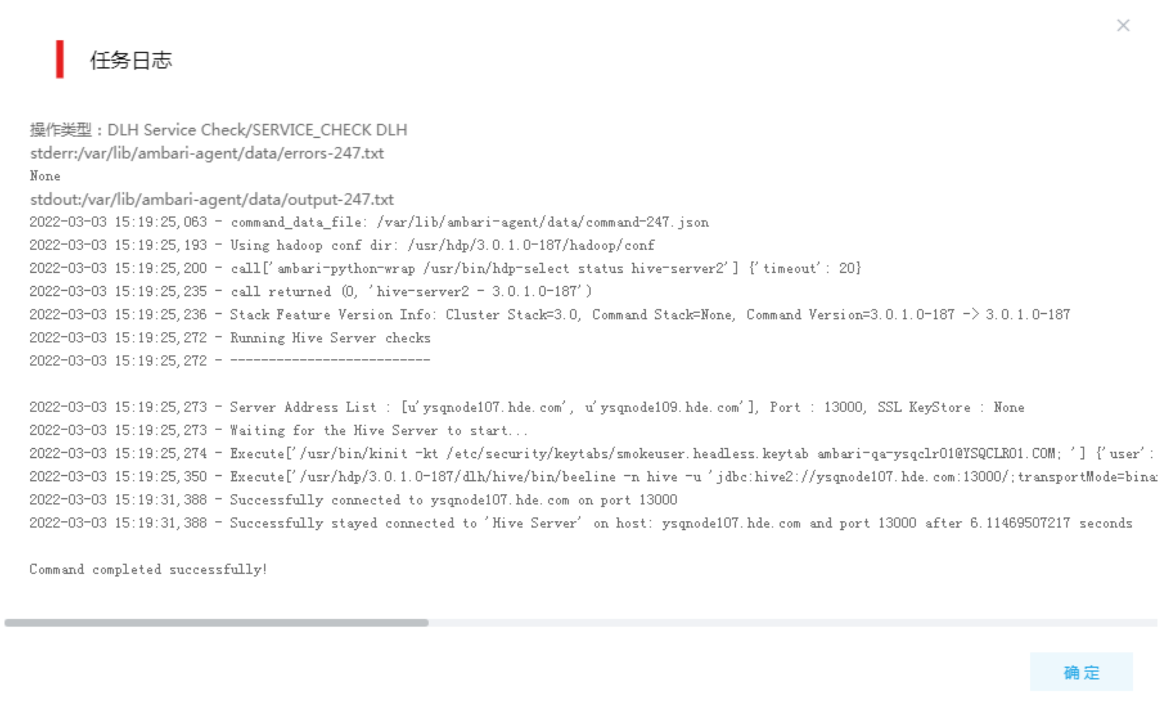
- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 DLH 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 DLH 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。
- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 组件检查结束后，检查窗中会显示组件检查成功或失败的状态。如图 2-5 所示，组件检查成功则表示该组件可正常使用。

图2-5 组件检查

(3) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“DLH Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-6 组件检查日志详情



2. Presto 组件检查

集群在使用过程中，根据实际需要，可对 Presto 组件执行组件检查操作。

(1) 对 Presto 组件执行组件检查的方式有以下三种，任选其一即可：

- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Presto 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击业务组件列表中 Presto 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。

- 在组件管理的组件详情页面右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 组件检查结束后，检查窗中会显示组件检查成功或失败的状态。如图 2-7 所示，组件检查成功则表示该组件可正常使用。

图2-7 组件检查

(3) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看"PRESTO Service Check"组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-8 组件检查日志详情

3. Altas 组件检查

集群在使用过程中，根据实际需要，可对 Altas 组件执行组件检查操作。

(1) 对 Altas 组件执行组件检查的方式有以下两种，任选其一即可：

- 在[集群管理/集群列表]页面，单击某集群名称可跳转至对应的集群详情页面。
  - 在集群详情页面选择[组件]页签，单击系统组件列表中 Altas 组件对应的<组件检查>按钮。
  - 在集群详情页面选择[组件]页签，单击系统组件列表中 Altas 组件名称进入组件详情页面，在右上角组件操作的下拉框中选择<组件检查>按钮。

(2) 组件检查结束后，检查窗中会显示组件检查成功或失败的状态。如图 2-9 所示，组件检查成功则表示该组件可正常使用。

图2-9 组件检查

(3) 组件检查结束后，在组件详情页面单击<操作记录>按钮，进入操作记录窗口。可查看“ATLAS Service Check”组件操作执行的详细信息以及操作日志详情，根据操作日志可判断组件检查的具体情况。

图2-10 组件检查日志详情

2.3 快速使用指导

DLH既可以通过集群用户访问，又可以通过组件超级用户访问。其中：

- 集群用户：指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。其中：
  - 集群超级用户：仅Hadoop集群拥有集群超级用户。新建Hadoop集群成功后，集群超级用户会自动同步到[集群权限/用户管理]页面，且对应描述为“集群超级用户”。
  - 集群普通用户：指在[集群权限/用户管理]页面新建的用户。开启权限管理后，普通用户绑定角色后即可拥有该角色所具有的权限；不开权限管理时，普通用户缺省仅拥有各组件原生的用户权限。
- 组件超级用户：指组件内部的最高权限用户，如Hive组件的hive用户。大数据集群中安装组件时均会缺省创建组件内置超级用户，也是集群用户的一种。

2.3.1 非 Kerberos 环境

说明

非 Kerberos 环境下，用户不需要做身份认证即可直接连接 DLH 执行相关管理操作。

非 Kerberos 环境下，通过控制台执行 DLH SQL 操作前需连接 DLHServer2，连接操作支持“通过 dlh 命令连接”和“通过 beeine 脚本连接”两种方式。

1. 通过 dlh 命令连接

登录集群内的任一节点（默认所有节点均安装了 DLH Client），直接执行 dlh 命令即可进行连接。通过该方式连接 DLHServer2 时会自动填入连接信息，连接用户使用当前用户。

图2-11 dlh 命令连接

```
[hdfs@node1 root]$ dlh
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLo
ggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/Stati
```

2. 通过 beeine 脚本连接

注意

执行连接操作的用户需具备相应的脚本执行权限。

登录集群内的任一节点（默认所有节点均安装了 DLH Client），执行如下命令：

/usr/hdp/3.0.1.0-187/dlh/hive/bin/beeline -u "{jdbc_url}" -n hive -p ""

【说明】

- 在 beeline 命令中：
  - -u：指定连接 DLHServer2 的 IP 地址，端口号默认为 13000。
  - -n：指定连接所需用户名。
  - -p：指定连接所需密码（可为空值）。
- DLH JDBC 连接 URL 信息获取包括以下两种方式：
  - 在 DLH 组件详情页面获取

    访问大数据平台管理系统，进入 DLH 组件详情页面，在基本信息模块可查看 DLHServer2 JDBC URL 信息，如图 2-12 所示，直接拷贝即可。

图2-12 DLH 组件 JDBC URL

- 手动拼接(未开启 Kerberos)

  jdbc:hive2://<DLHServer2 地址>:13000/

  其中：

  – jdbc:hive2://：为固定前缀。
  – <DLHServer2 地址>为 DLHServer2 所在的机器 IP 地址或主机名（在集群外通过主机名连接 HiveServer2 时必须配置本地 hosts 文件）。
  – 13000：为 DLHServer2 的默认端口号。

2.3.2 Kerberos 环境

说明

Kerberos 环境下，用户需要做身份认证才可直接连接 DLH 执行相关管理操作，认证方式请参见 2.3.2 1. Kerberos 环境下用户身份认证。

Kerberos 环境下，通过控制台执行 DLH SQL 操作前需执行以下两步操作：

(1) 首先进行用户身份认证，才可以正常访问 DLH。

(2) 连接 DLHServer2，连接操作支持“通过 dlh 命令连接”和“通过 beeine 脚本连接”两种方式。

1. Kerberos 环境下用户身份认证

如果大数据集群开启 Kerberos，若想操作 DLH，则必须首先进行用户身份认证。根据用户类型不同，分为以下两类：

- 集群用户身份认证
- 组件用户身份认证

（一）集群用户身份认证

说明

- 集群用户指在大数据集群的[集群权限/用户管理]页面可查看到的用户，包括集群超级用户和集群普通用户。
- Kerberos 环境下，集群用户的认证文件可在[集群权限/用户管理]页面，单击用户列表中用户对应的<下载认证文件>按钮进行下载。

DLH 还可以通过集群用户访问。在开启 Kerberos 的大数据集群中进行集群用户（以 user1 用户示例）身份认证的方式，包括以下两种（根据实际情况任选其一即可）：

(1) **方式一：（此方式不要求知道用户密码，直接使用 keytab 文件进行认证）**

a. 将用户 user1 的认证文件（即 keytab 配置包）解压后，上传至访问节点的/etc/security/keytabs/目录下，然后将 keytab 文件的所有者修改为 user1，命令如下：

```
chown user1 /etc/security/keytabs/user1.keytab
```

b. 使用 ***klist*** 命令查看 user1.keytab 的 principal 名称，命令如下：

```
klist -k user1.keytab
```

【说明】如图 2-13 所示，红框内容即为 user1.keytab 的 principal 名称。

图2-13 认证文件的 principal 名称

```
[root@tenant keytabs]# klist -k user1.keytab
Keytab name: FILE:user1.keytab
KVNO Principal
---- ----------------------------------------------------------------
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
   1 user1@TENANTC.COM
```

c. 切换至用户 user1，并执行身份验证的命令如下：

```
su user1
kinit -kt user1.keytab user1@TENANTC.COM
```

【说明】其中：user1.keytab 为用户 user1 的 keytab 文件，user1@TENANTC.COM 为 user1.keytab 的 principal 名称。

d. 输入 ***klist*** 命令可查看认证结果。

图2-14 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

(2) **方式二：（此方式要求用户密码已知，通过密码直接进行认证）**

a. 输入以下命令：kinit user1

b. 根据提示输入密码 Password for user1@TENANTC.COM：<密码>

c. 输入 ***klis*t** 命令可查看认证结果。

图2-15 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

（二）组件超级用户身份认证

DLH 可以通过组件超级用户访问，在开启 Kerberos 的集群中进行组件超级用户（以 hive 用户示例）认证的步骤如下：

(1) 在集群内节点的/etc/security/keytabs/目录下，查找 hive 的认证文件“hive.service.keytab”。

【说明】在 DLH Client 节点上，需要将 hive 的认证文件“hive.service.keytab”上传节点的/etc/security/keytabs/目录下进行认证。

(2) 使用 ***klist*** 命令查看 hive.service.keytab 的 principal 名称，命令如下：

klist -k hive.service.keytab

【说明】如图 2-16 所示，红框内容即为 hive.service.keytab 的 principal 名称。

图2-16 认证文件的 principal 名称

```
[root@node1 keytabs]# klist -k hive.service.keytab
Keytab name: FILE:hive.service.keytab
KVNO Principal
---- --------------------------------------------------------------------------
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
   2 hive/node1.hde.com@TESTSHARE.COM
```

(3) 切换至用户 hive，并执行身份验证的命令如下：

su hive

kinit -kt hive.service.keytab hive/node1.hde.com@TESTSHARE.COM

【说明】其中：hive.service.keytab 为 hive 的认证文件，hive/node1.hde.com@TESTSHARE.COM 为 hive.service.keytab 的 principal 名称。

(4) 输入 ***klist*** 命令可查看认证结果。

图2-17 查看认证结果

```
sh-4.2$ klist
Ticket cache: FILE:/tmp/krb5cc_10007
Default principal: user1@TENANTC.COM

Valid starting       Expires              Service principal
03/16/2022 18:41:44  02/18/2027 18:41:44  krbtgt/TENANTC.COM@TENANTC.COM
sh-4.2$
```

2. 通过 dlh 命令连接

登录集群内的任一节点（默认所有节点均安装了 DLH Client），直接执行 dlh 命令即可进行连接。通过该方式连接 DLHServer2 时会自动填入连接信息，连接用户使用当前用户。

图2-18 dlh 命令连接

```
[hdfs@node1 root]$ dlh
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLo
ggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/Stati
```

3. 通过 beeine 脚本连接

注意

执行连接操作的用户需具备相应的脚本执行权限。

登录集群内的任一节点（默认所有节点均安装了 DLH Client），执行如下命令：

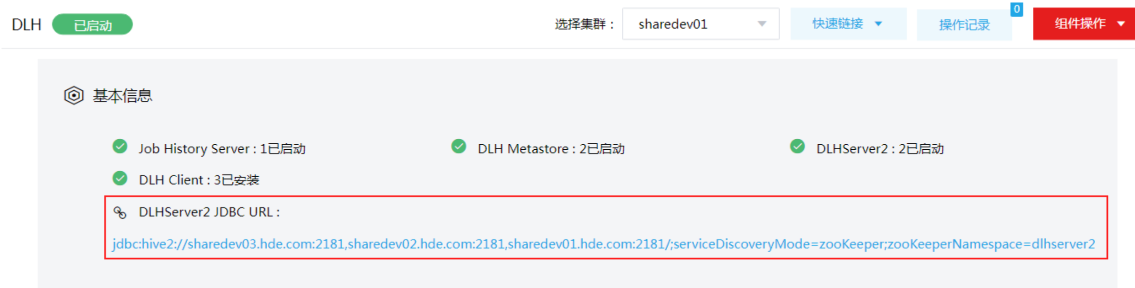
/usr/hdp/3.0.1.0-187/dlh/hive/bin/beeline -u "{jdbc_url}" -n hive -p ""

【说明】

- 在 beeline 命令中：
  - -u：指定连接 DLHServer2 的 IP 地址，端口号默认为 13000。
  - -n：指定连接所需用户名。
  - -p：指定连接所需密码（可为空值）。
- DLH JDBC 连接 URL 信息获取包括以下两种方式：
  - 在 DLH 组件详情页面获取

    访问大数据平台管理系统，进入 DLH 组件详情页面，在基本信息模块可查看 DLHServer2 JDBC URL 信息，如图 2-19 所示，直接拷贝即可。

图2-19 DLH 组件 JDBC URL



- 手动拼接 URL(开启 Kerberos)

  jdbc:hive2:// <DLHServer2 地址>:13000/;principal=hive/_HOST@SHAREDEVTEST.COM

  其中：

  – -u 用于指定连接 DLHServer2 的地址和端口号。

  – DLHServer2 的端口号默认为 13000，principal 名称为 hive/_HOST@SHAREDEVTEST.COM（其中_HOST 是 DLHServer2 进程所在主机的域名地址，principal 获取方式请参考下面的说明），示例：hive/sharedev1.hde.com@SHAREDEVTEST.COM。

  – -n 用于指定连接所需的用户名（可为空值）。

  – -p 用于指定连接所需的密码（可为空值）。

说明

关于 principal 的配置值获取，可以登录大数据平台管理系统，进入 DLH 组件详情页面，在组件详情的[配置]页签，可查看配置文件 dlh-site.xml 中“hive.server2.authentication.kerberos.principal”的配置项获取。

2.3.3 Presto 访问接口

说明

当前版本中，Presto 不支持 Kerberos 认证，所以使用方式也不区分集群是否开启 Kerberos。

1. Presto CLI 方式

登录集群内的任一节点，进入/usr/hdp/3.0.1.0-187/presto/bin 目录，执行如下命令可以连接到 CLI 控制台，如图 2-20 所示：

./lk --server sharedev1.hde.com:18089 --catalog default --schema default

其中：

- --server：用于指定 presto coodinator 进程所在主机名或 IP 及端口格式，如：sharedev1.hde.com:18089，端口默认为 18089。若集群开启了 HA，也可以通过连接到集群

的虚拟主机名以及代理端口来访问，比如：cloudos5-vserver.vip.hde.com:18090，端口默认为 18090，这部分信息可以从 Presto 组件的自定义 config 配置 http.proxy.url 获取。

- --catalog：用于指定 catalog 名称。
- --schema：用于指定某 catalog 下的数据库或模式。

【说明】其他更多的参数可以执行./lk --help 进行查看。

图2-20 Presto CLI 控制台

```
[hdfs@ysqnode107 bin]$ ./lk --server ysqnode107.hde.com:18089 --catalog default --schema
default lk:default> show catalogs;
 Catalog
---------
 default
 system
(2 rows)

Query 20220303_070926_00016_sp9yi, FINISHED, 3 nodes
Splits: 53 total, 53 done (100.00%)
0:00 [0 rows, 0B] [0 rows/s, 0B/s]

lk:default> use default;
USE
lk:default> show tables;
 Table
-------
 t1
 t2
 t3
 t4
(4 rows)

Query 20220303_070935_00020_sp9yi, FINISHED, 4 nodes
Splits: 53 total, 53 done (100.00%)
0:01 [4 rows, 76B] [2 rows/s, 47B/s]

lk:default> select * from t1 limit 10;
 id | name | age
----+------+-----
  1 | a    |  23
  4 | d    |  26
  5 | e    |  26
  2 | b    |  25
  3 | c    |  25
(5 rows)

Query 20220303_070946_00021_sp9yi, FINISHED, 1 node
Splits: 18 total, 18 done (100.00%)
0:02 [5 rows, 513B] [2 rows/s, 219B/s]

lk:default>
```

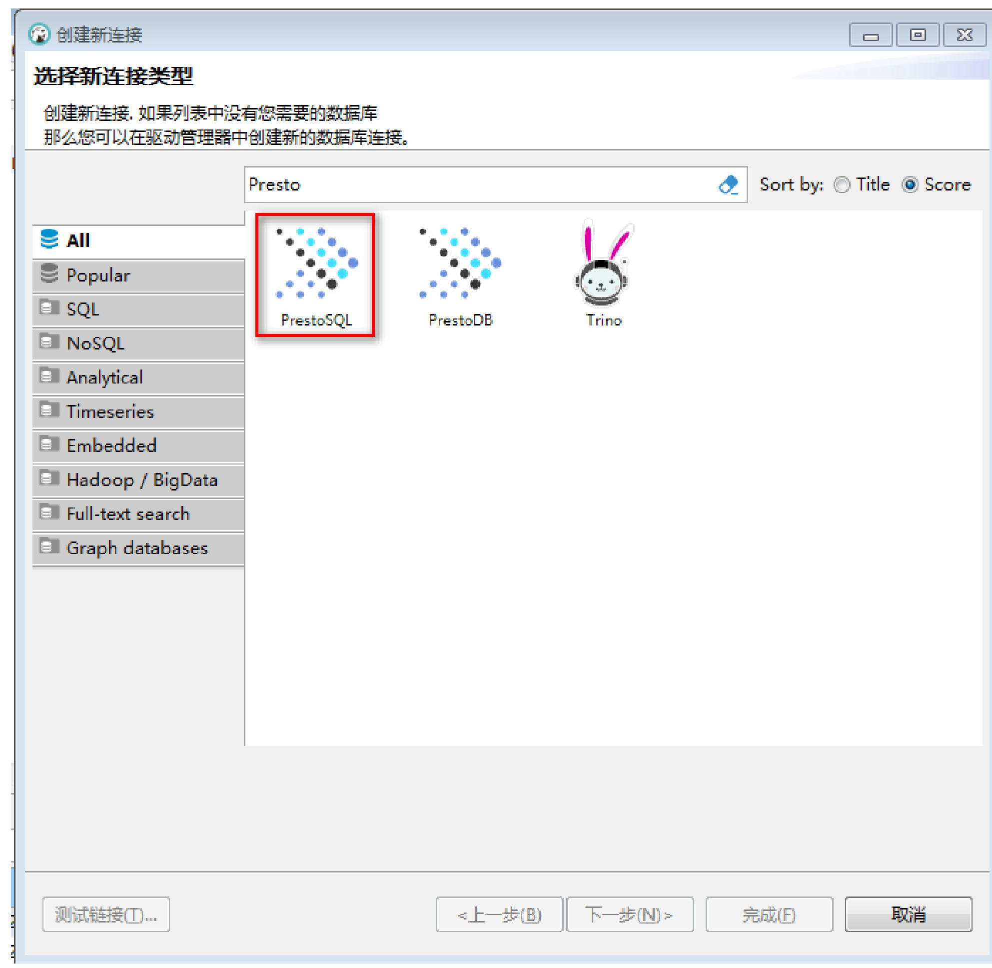
2. Presto JDBC 方式

Presto 提供 JDBC 驱动，支持程序或工具通过 jdbc 接口执行 Presto 任务。

本章节介绍通过 Dbeaver 工具连接 Presto，步骤如下：

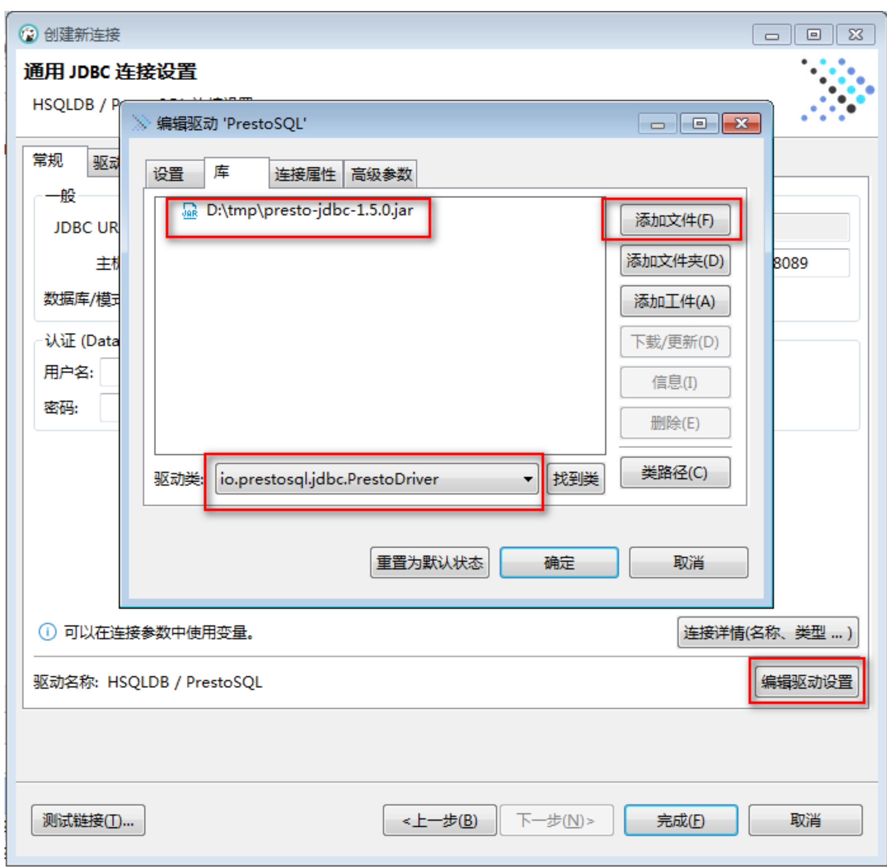
(1) 在 Dbeaver 工具中创建新连接，选择 PrestoSQL 连接类型，然后单击<下一步>。

图2-21 创建 PrestoSQL 类型的连接



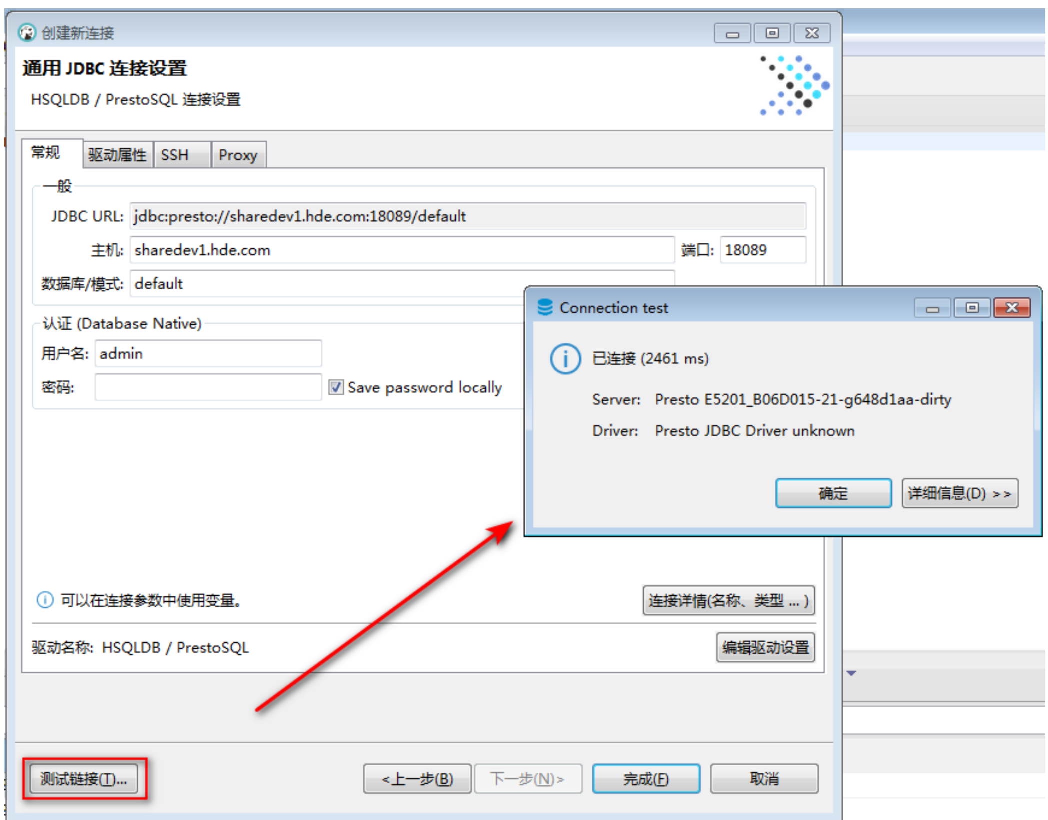
(2) 在通用 JDBC 连接设置的弹窗中，单击<编辑驱动设置>按钮。在编辑驱动'PrestoSQL'弹窗的“库”页签单击<添加文件>按钮，此时文件选择 Presto jdbc 驱动（该驱动可以从集群的/usr/hdp/3.0.1.0-187/presto/jdbc/presto-jdbc-1.5.0.jar 获取），编辑驱动完成后单击<确定>按钮关闭弹窗。

图2-22 编辑驱动'PrestoSQL'



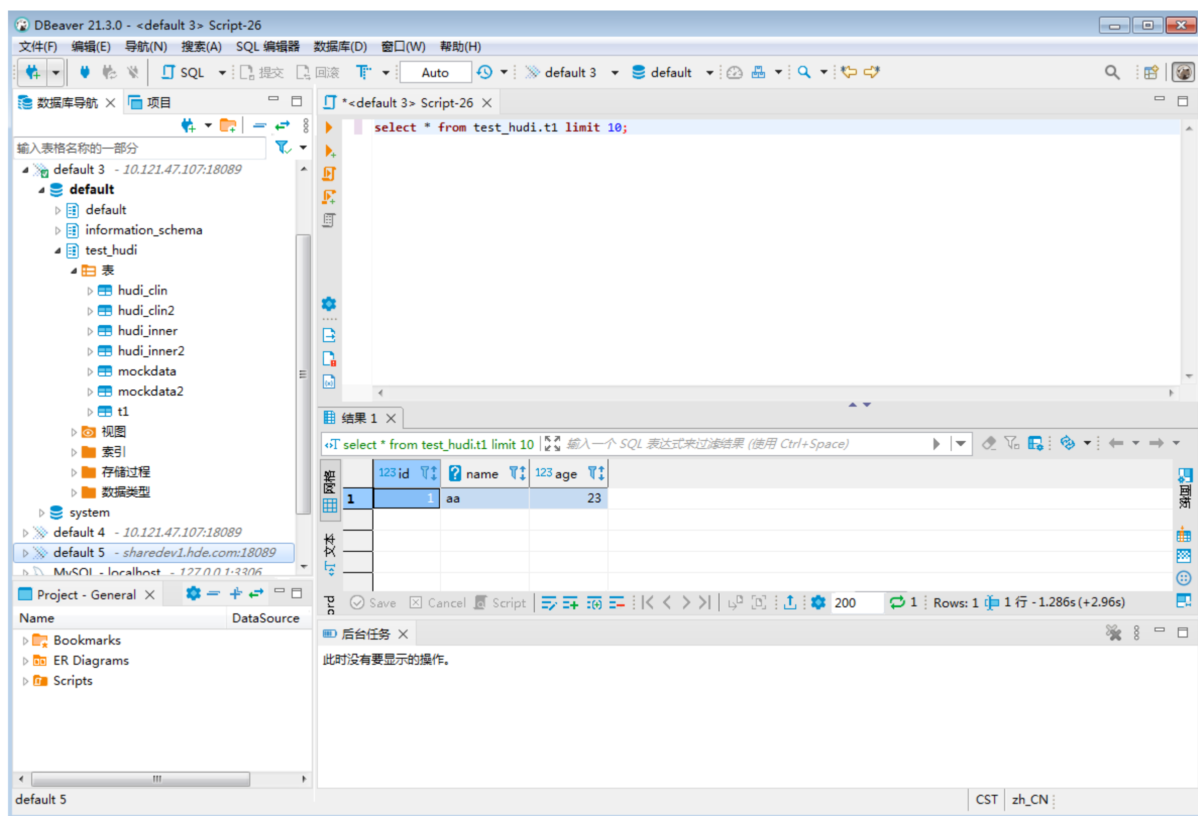
(3) 在通用 JDBC 连接设置的"常规"页签填写连接的相关信息，其中：主机为 Presto Coodinator 进程所在主机名或 IP，端口默认为 18089（若集群开启了 HA，也可以通过连接到集群的虚拟主机名以及代理端口来访问，比如：cloudos5-vserver.vip.hde.com:18090，端口默认为 18090，这部分信息可以从 Presto 组件的自定义 config 配置 http.proxy.url 获取），数据库/模式需填写对应的信息，认证信息中用户名需要填写执行的用户，密码不用填写。然后单击窗口左下角的<测试链接>按钮，可以验证连接是否正确。若测试连接时正确，单击<完成>按钮即可成功创建连接。

图2-23 创建连接



(4) 连接创建后，即可看到该数据源连接。如图 2-24 所示，显示 default3 下的 catalog、schema、table 信息，此时在左侧窗口中即可执行相关操作，如：查询 default 这个 catalog 下数据库 schema 为 test_hudi 的 t1 表。

图2-24 操作数据源



2.4 快速链接使用

因 DLH 组件强依赖 Presto 和 Atlas 组件，所以进行 DLH 管理时可能会需要访问这 3 个组件的快速链接页面。其中：DLH 和 Presto 为业务组件，Altas 为系统组件。

1. DLH 快速链接

DLH 快速链接包括 DLHServer 监控和 HistroyServer 任务监控，分别访问不同的监控页面可以查看不同的信息。如图 2-25 所示，在 DLH 组件详情页面的右上角[快速链接]的下拉框中，可以获取不同监控页面的访问入口信息。

【说明】当集群开启高可用时，DLH 组件会同步开启 HA，此时部分监控页面有两个访问入口，任选其一即可。

图2-25 DLH 快速链接

2. Presto 快速链接

Presto 快速链接为 Presto UI，提供了查看集群任务、节点等信息以及动态添加 catalog 等功能。如图 2-26 所示，在 Presto 组件详情页面的右上角[快速链接]的下拉框中，可以获取 Presto UI 访问入口信息。

【说明】当集群开启高可用时，Presto 组件会同步开启 HA，此时组件 UI 页面有两个访问入口，任选其一即可。

图2-26 Presto 快速链接

3. Atlas 快速链接

Atlas 快速链接为 Atlas UI，提供了基于 DLH 库、表、列及血缘等元数据管理功能。如图 2-27 所示，在 Atlas 组件详情页面的右上角[快速链接]的下拉框中，可以获取 Atlas UI 访问入口信息。

【说明】当集群开启高可用时，Atlas 组件会同步开启 HA，此时组件 UI 页面有两个访问入口，任选其一即可。

图2-27 Atlas 快速链接

2.4.1 配置组件快速链接

大数据集群部署完成后，若想要访问 DLH、Presto 或 Atlas 监控页面，首先需要修改本地 hosts 文件，用以确保组件的快速链接页面通过域名访问能够顺利跳转。

修改本地 hosts 文件的方法如下：

(1) 登录大数据集群中任意一节点，查看当前集群的 hosts 文件（Linux 环境下位置为/etc/hosts）。
(2) 将集群的 hosts 文件信息添加到本地 hosts 文件中。若本地电脑是 Windows 环境，则 hosts 文件位于 C:\Windows\System32\drivers\etc\hosts，修改该 hosts 文件并保存。
(3) 在本地 hosts 文件中配置主机域名信息完成后，即可访问组件的快速链接。

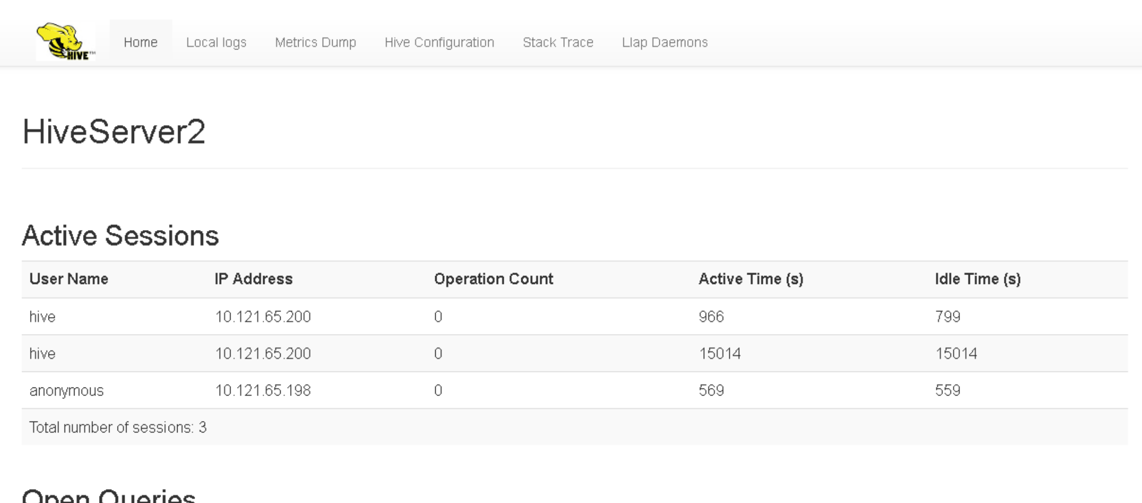
2.4.2 访问监控页面

1. DLHServer 监控

在 DLH 组件详情页面的右上角[快速链接]的下拉框中选择 DLHServer（如果多个快速链接，任选其一即可），此时不需要通过用户名和密码，即可直接跳转至 DLHServer 监控 UI 页面。

如图 2-28 所示，DLHServer 主页监控页面显示的是当前链接的会话，包括：IP、用户名、当前执行的查询数量、链接总时长、空闲时长等。如果有会话执行查询，页面中的 Queries 模块会显示查询的语句、执行耗时等。

图2-28 访问 DLHServer UI 页面



2. HistroyServer 任务监控

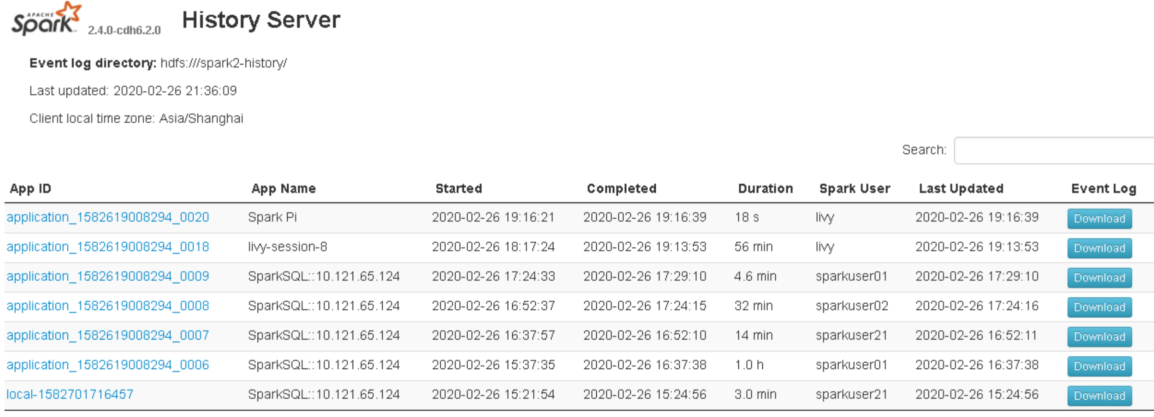
在 DLH 组件详情页面的右上角[快速链接]的下拉框中选择 HistoryServer，根据集群是否开启 Kerberos，访问 HistoryServer 快速链接分为两种情况：

- 若集群没有开启 Kerberos 认证，则此时点击访问入口链接，不需要通过用户名和密码，即可直接跳转访问 HistoryServer 任务监控 UI 页面。
- 若集群开启了 Kerberos 认证，则此时点击访问入口链接，需要输入用户名和密码进行认证（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），然后才可跳转访问对应的 UI 页面。

在 HistoryServer 监控页面，可查看如下信息：

(1) 如图 2-29 所示，History Server 页面默认展示已完成的历史任务列表。单击页面左下角“Show incomplete applications”可跳转到正在运行的任务列表。

图2-29 访问 HistroyServer UI 页面



2.4.0-cdh6.2.0 History Server

Event log directory: hdfs:///spark2-history/

Last updated: 2020-02-26 21:36:09

Client local time zone: Asia/Shanghai

Search:

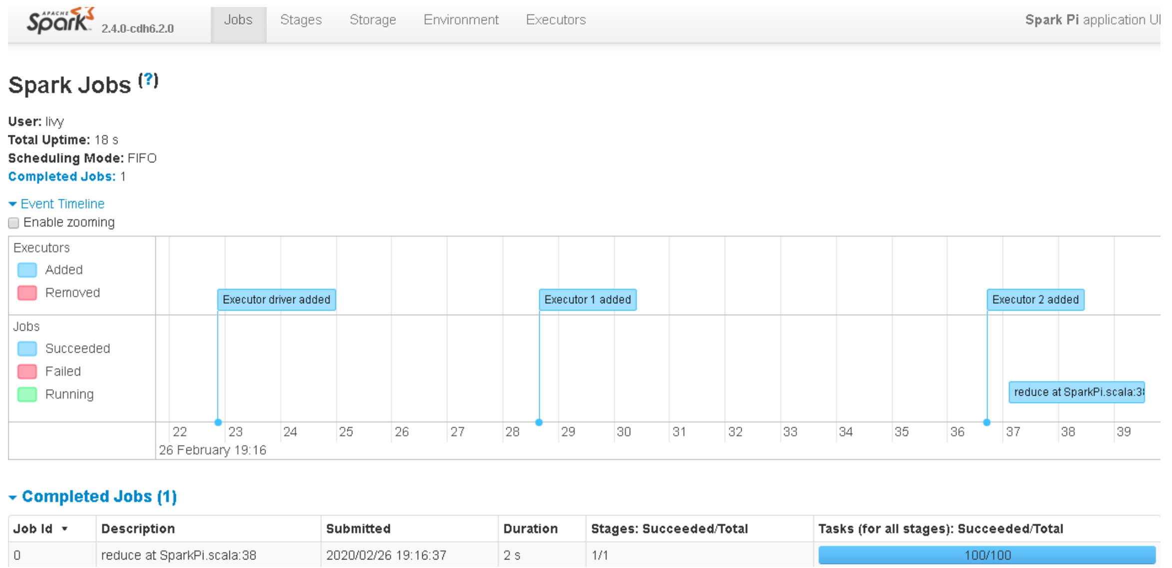
App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
application_1582619008294_0020	Spark Pi	2020-02-26 19:16:21	2020-02-26 19:16:39	18 s	livy	2020-02-26 19:16:39	Download
application_1582619008294_0018	livy-session-8	2020-02-26 18:17:24	2020-02-26 19:13:53	56 min	livy	2020-02-26 19:13:53	Download
application_1582619008294_0009	SparkSQL::10.121.65.124	2020-02-26 17:24:33	2020-02-26 17:29:10	4.6 min	sparkuser01	2020-02-26 17:29:10	Download
application_1582619008294_0008	SparkSQL::10.121.65.124	2020-02-26 16:52:37	2020-02-26 17:24:15	32 min	sparkuser02	2020-02-26 17:24:16	Download
application_1582619008294_0007	SparkSQL::10.121.65.124	2020-02-26 16:37:57	2020-02-26 16:52:10	14 min	sparkuser21	2020-02-26 16:52:11	Download
application_1582619008294_0006	SparkSQL::10.121.65.124	2020-02-26 15:37:35	2020-02-26 16:37:38	1.0 h	sparkuser01	2020-02-26 16:37:38	Download
local-1582701716457	SparkSQL::10.121.65.124	2020-02-26 15:21:54	2020-02-26 15:24:56	3.0 min	sparkuser21	2020-02-26 15:24:56	Download

Showing 1 to 7 of 7 entries

Show incomplete applications

(2) 单击任务列表中的 App ID，可进入该任务的 Jobs 列表页面，可对 Completed Jobs 已完成的任务查看详情。

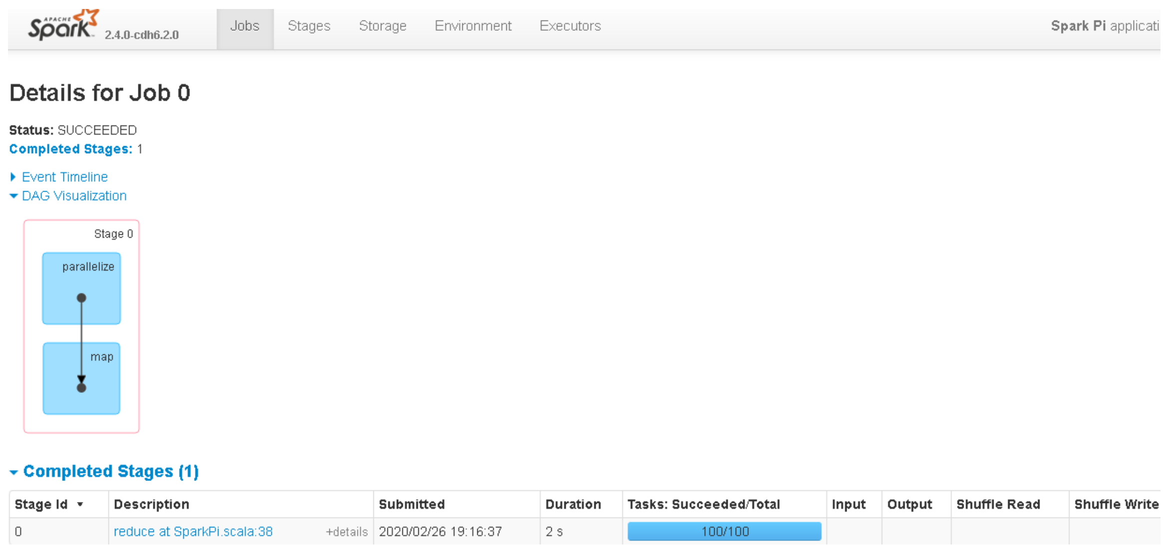
图2-30 Spark Jobs 列表页面



Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	reduce at SparkPi.scala:38	2020/02/26 19:16:37	2 s	1/1	100/100

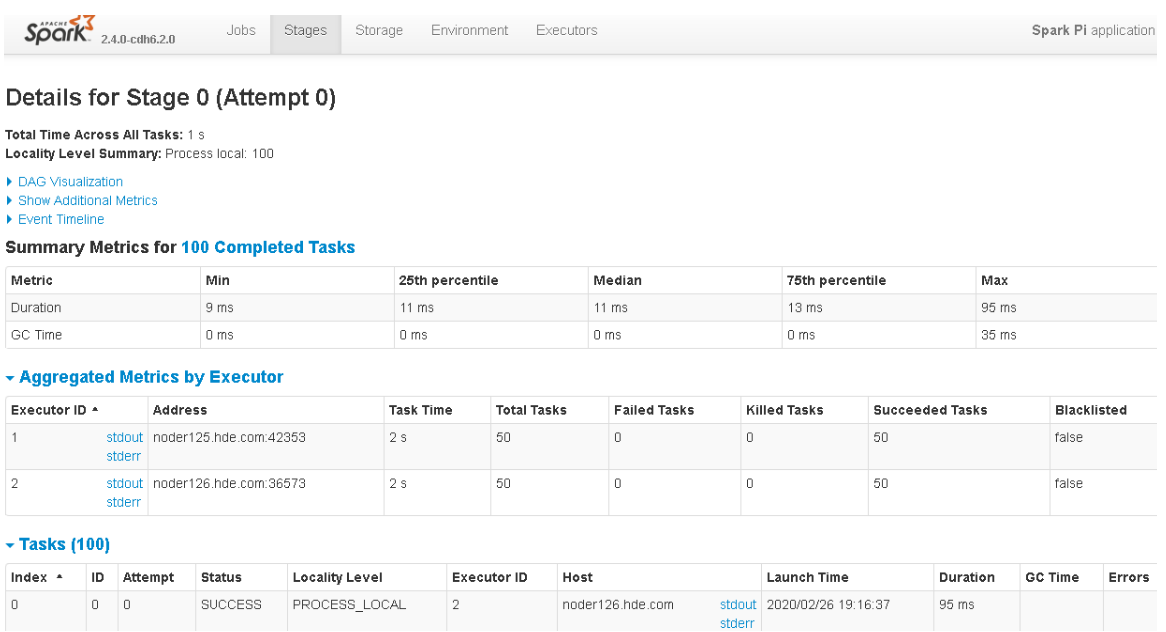
(3) 在 Spark Jobs 列表页面，单击“Completed Jobs”列表中的任意 Job，可进入该 Job 的详情页面，查看 Stage 列表。

图2-31 Job 详情页面



(4) 在 Job 详情页面，单击“Completed Stages”列表中任意 Stage，可进入该 Stage 的详情页面，查看每个 task 的执行时间、GC 时间等。

图2-32 Stage 详细页面



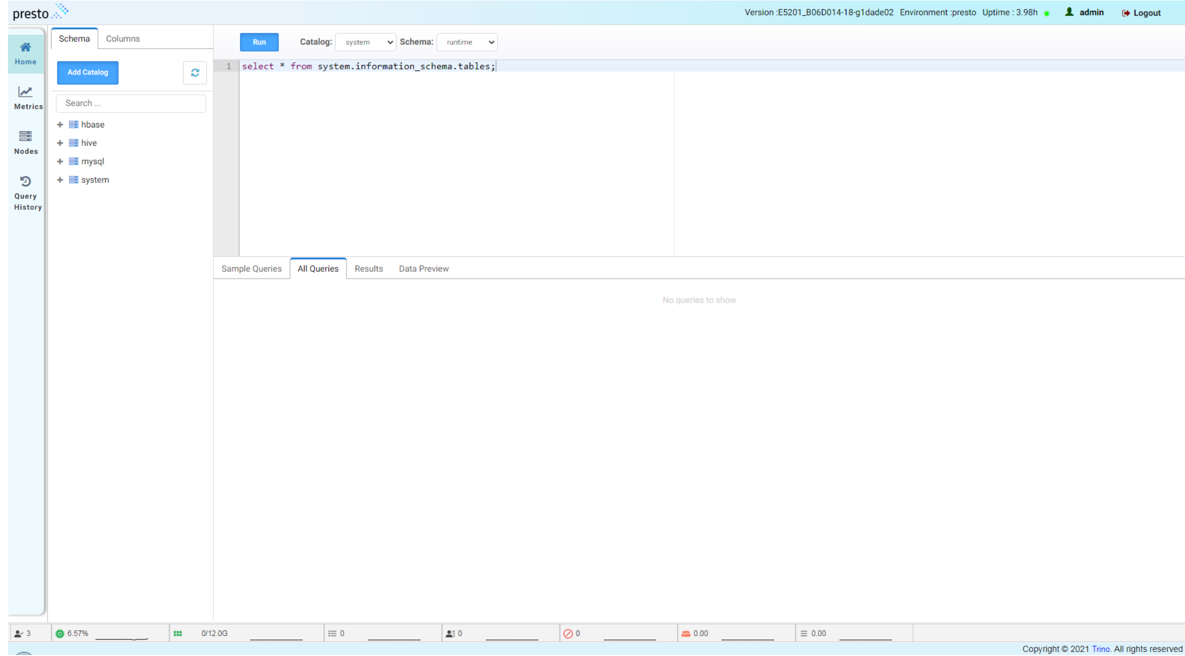
3. Presto 集群监控

在 Presto 组件详情页面的右上角[快速链接]的下拉框中，选择 Presto（如果多个快速链接，任选其一即可），此时需要通过用户名（可以使用集群创建时填写的超级用户，也可以使用用户管理中创建的用户），才可访问 Presto 集群监控 UI 页面。

(1) Presto UI 页面主要分为左右两部分，左部分为功能模块菜单，包括 Home、Metrics、Nodes 和 Query History，右部分为功能展示区。

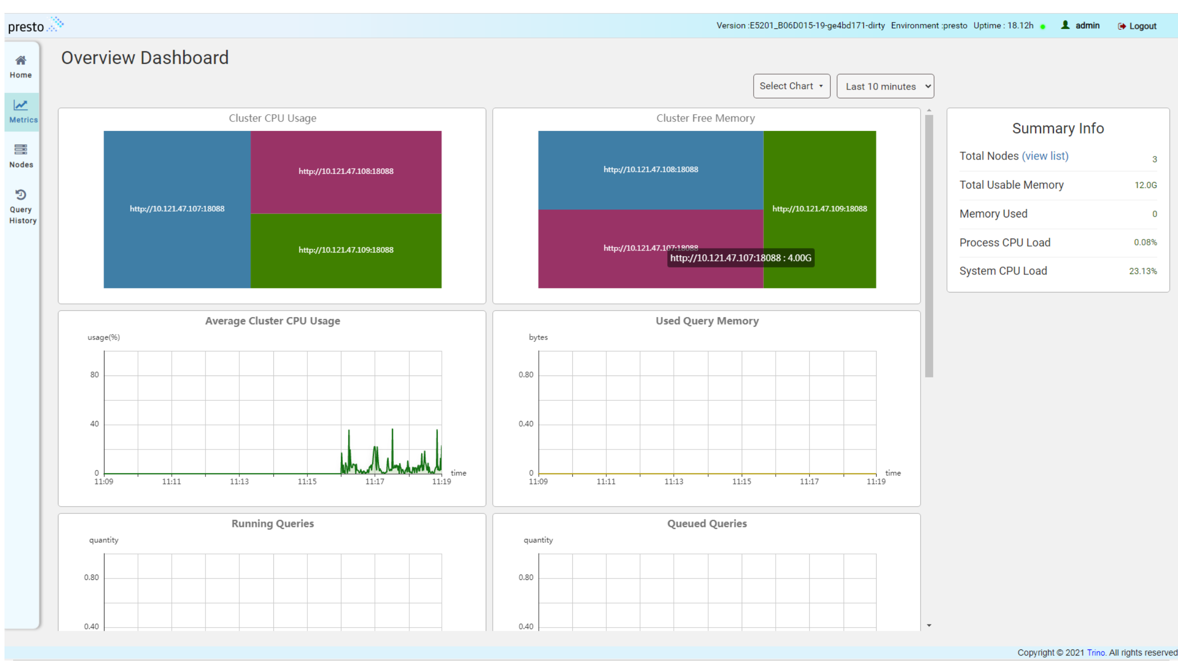
(2) 如图 2-33 所示，Home 主页可以添加 Catalog，并基于创建的 Catalog 进行 SQL 的查询，另外也支持对表的列信息进行查看。

图2-33 Presto UI Home 页面



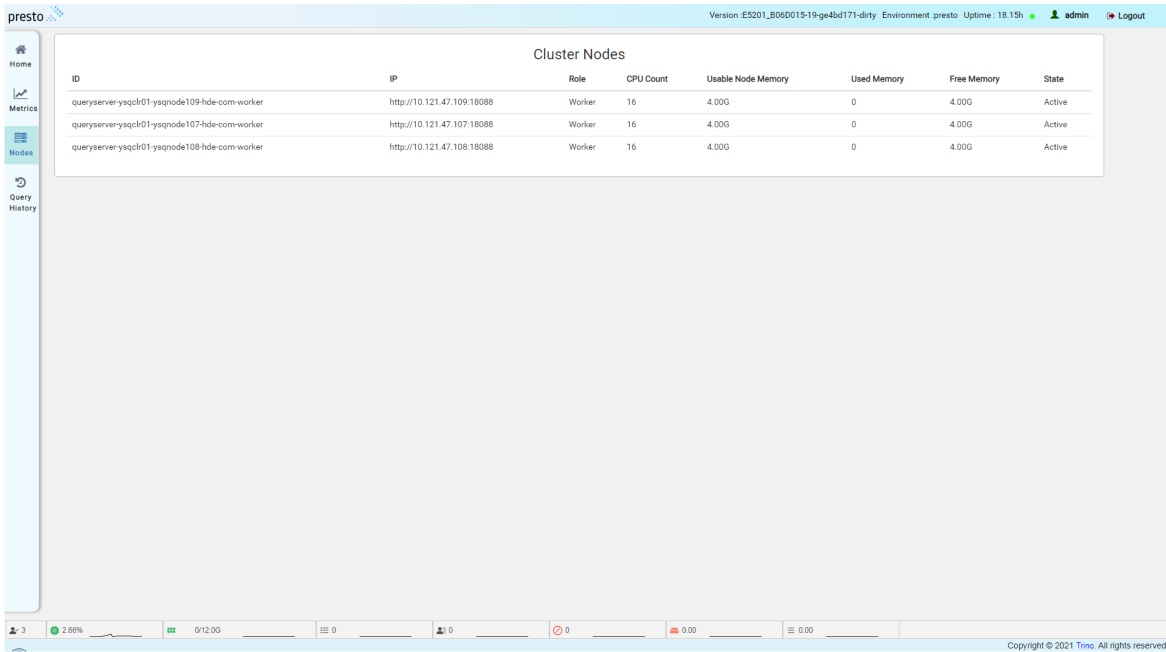
(3) 如图 2-34 所示，在 Metrics 页面，可以查看 Presto 集群 CPU 使用率、集群可用内存、平均 CPU 使用率、已使用查询内存等监控指标数据。

图2-34 Presto UI Metrics 页面



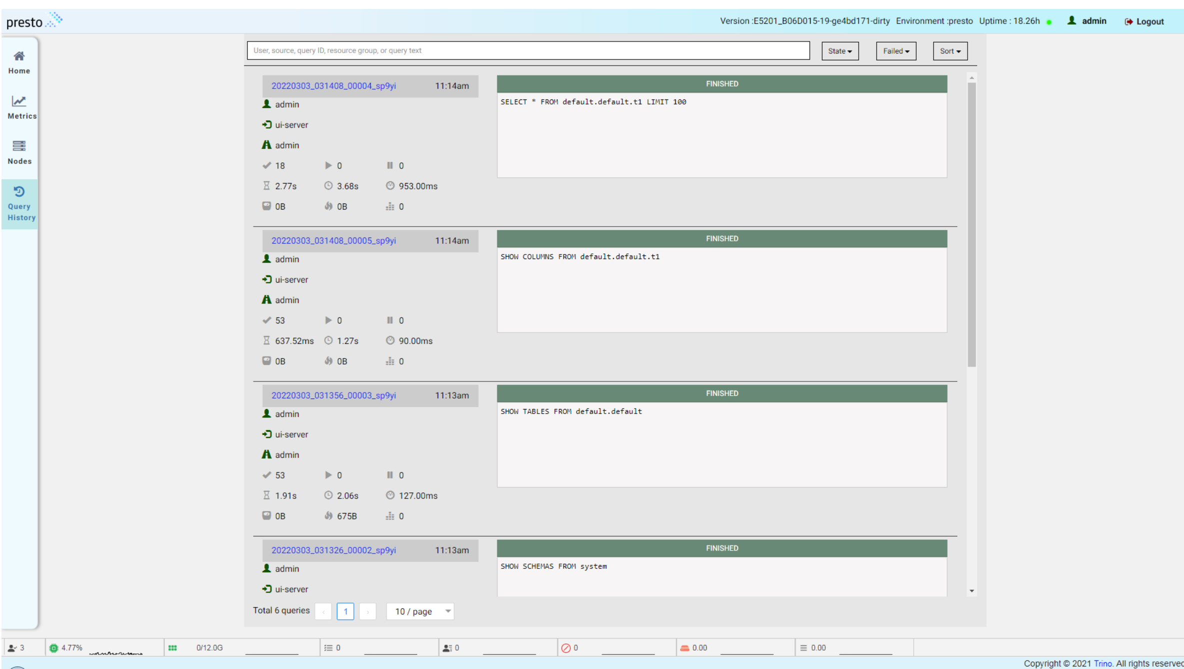
(4) 如图 2-35 所示，在 Nodes 页面，可以查看 Presto 集群节点信息。

图2-35 Presto UI Nodes 页面



(5) 如图 2-36 所示，在 Query History 页面，可以查看当前执行的查询（默认每页显示 10 个查询），在查询列表展示模块，当前列表中可以展示的查询的数量是可以配置的。同时，页面支持根据一些条件过滤和定位目标查询；页面提供了搜索输入框用于定位查询（匹配项包括：用户名、查询 source、查询 ID、resource group，甚至是 SQL 文本和查询状态等）；页面也支持根据后面预设的一些状态（running、queued、finished、failed 等）对查询进行筛选等。

图2-36 Presto UI Query History 页面



(6) 查询明细视图

在查询列表展示模块，点击查询 ID 可跳转到该查询的详情界面。

- Overview

  如图 2-37 所示，Overview 页签展示该查询概要详情信息、执行过程的资源消耗、执行语句等。

- 执行计划(Live Plan)

  如图 2-38 所示，该页面只有在查询结束后才可以访问。在查询执行期间，计划中的计数器在查询执行过程中更新，Live Plan 中的值与 Overview 选项卡中描述的相同，但是它们在查询执行计划上会实时覆盖。查看 Live Plan 该试图有助于可视化查询被阻塞或花费大量时间的位置，以便诊断或改进性能问题。

- Stage Preformance

  如图 2-39 所示，Stage Performance 页签展示查询处理完成后 Stage 性能的可视化详细信息。该视图可以看作是 Live Plan 视图的下钻，单击每个 operator（上图中箭头部分）或者 Stage Performance 页签可访问详细信息。

图2-37 Overview 页签信息

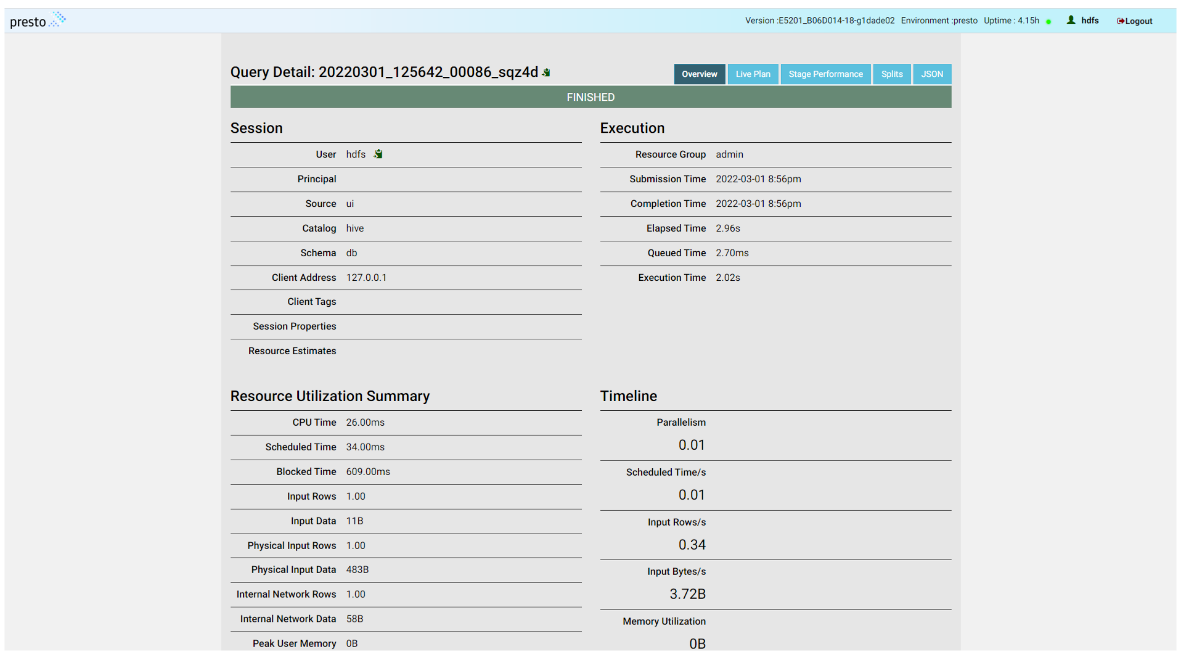


图2-38 执行计划页签信息

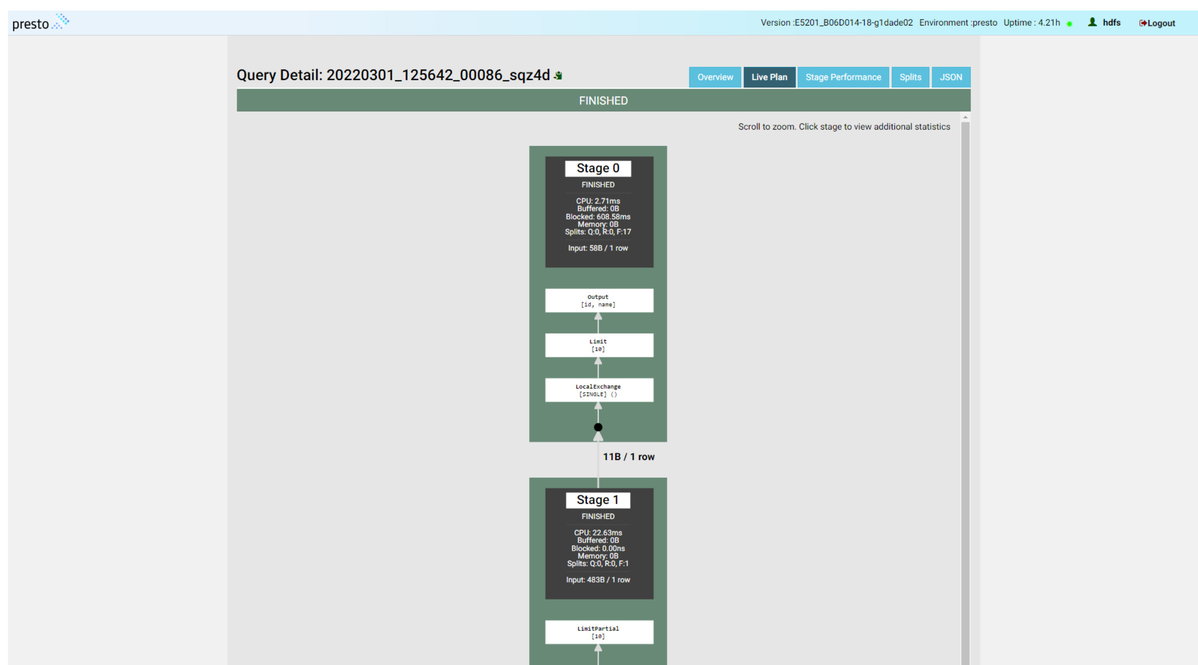
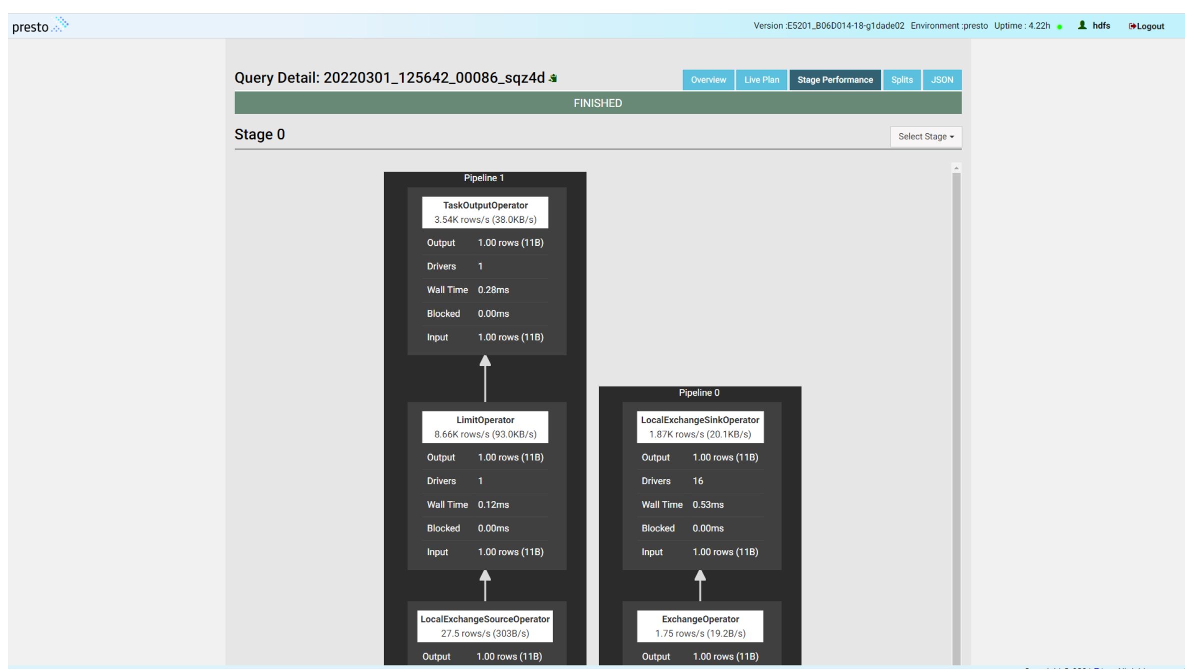


图2-39 Stage Performance 页签信息



4. Atlas 元数据管理

DLH 基于 Atlas 提供了 Hive 元数据管理功能，对 DLH 内部的数据库、数据表、数据流、数据列以及列血缘等元数据提供了查看基本属性（在基本属性中可以在业务层面增加属性以及标签）、血缘分析以及业务类别分类等能力。

Altas 为系统组件，访问 DLH 的 Atlas 元数据管理页面的方式如下：

(1) 在集群管理页面的左侧导航树中选择[集群列表]，进入集群列表页面，单击集群名称可跳转至集群详情页面。

(2) 在集群详情页面选择[组件]页签，单击系统组件列表中 Atlas 组件名进入 Atlas 组件详情页面。

(3) 在 Atlas 组件详情页面的右上角[快速链接]的下拉框中，选择 Atlas（如果多个快速链接，任选其一即可），可跳转至 DLH 对应的 Atlas 元数据管理登录页面，此时需要通过 admin 用户和密码（admin 用户的密码为 CloudOS5#DE3@Atlas），才可访问 Atlas 元数据管理 UI 页面。

说明

本章节仅对 Atlas 元数据管理 UI 页面进行简单说明，更多关于 Atlas 原生 UI 界面的功能特性说明请参见 atlas 官网。

下面对 Atlas 元数据管理 UI 页面进行说明：

(1) Atlas 元数据管理 UI 页面主要分为左右两部分，左部分为功能模块菜单，包括 Entities、Classification、Business Metadata、Glossaries 和 Custom Filters，右部分为功能展示区。

(2) 如图 2-40 所示，在 Entities 菜单中，单击具体的实体元数据（如：hive_table(4)）则显示对应表的元数据信息。如图 2-41 所示，在表的元数据信息中再单击具体的表 Name 可以跳转至表的详情页面。上面中间的搜索框可以对 Entities 菜单中的各类实体元数据进行全局搜索。

图2-40 Entities 菜单

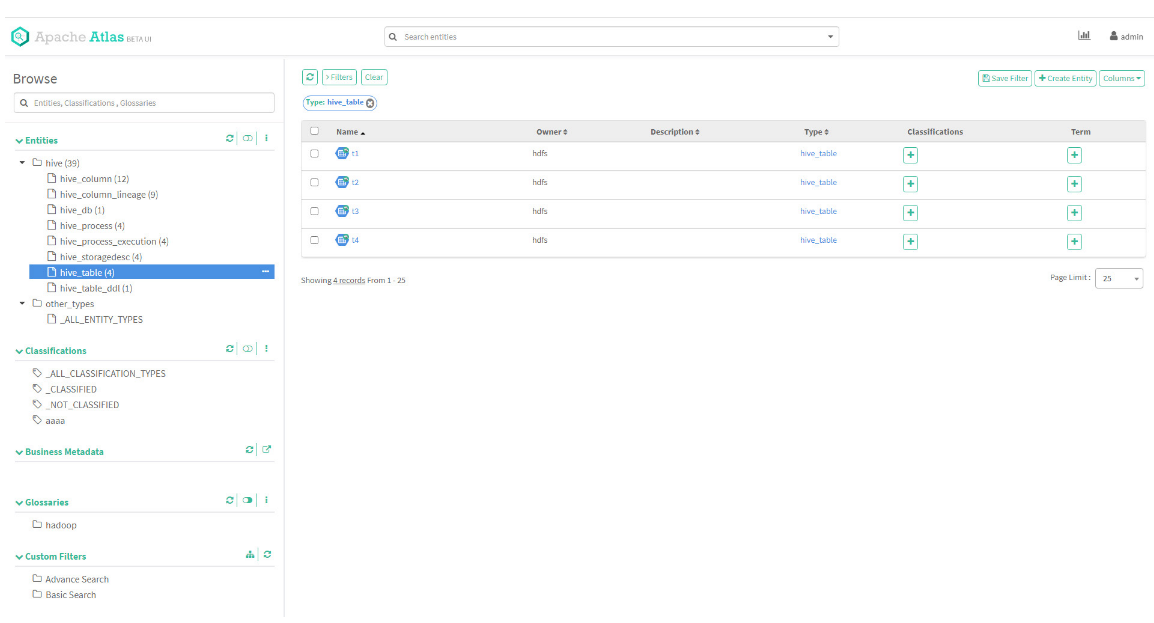
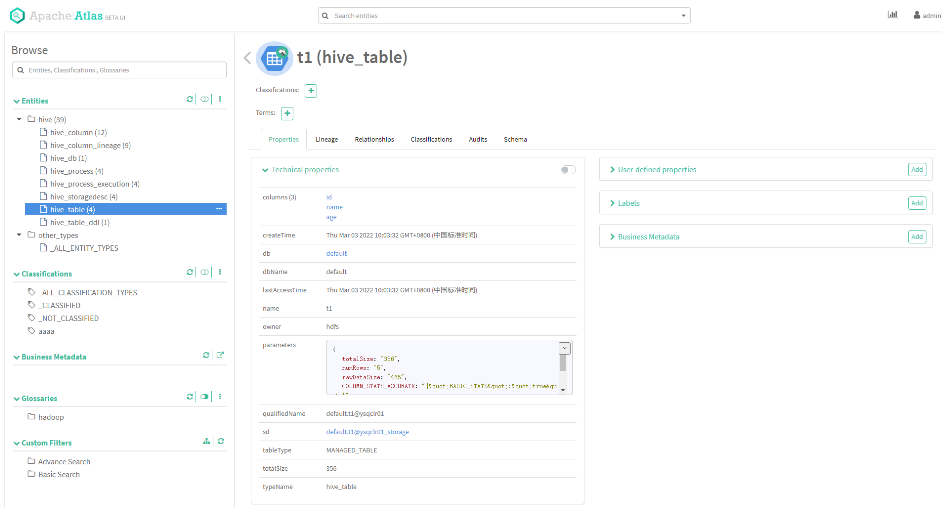
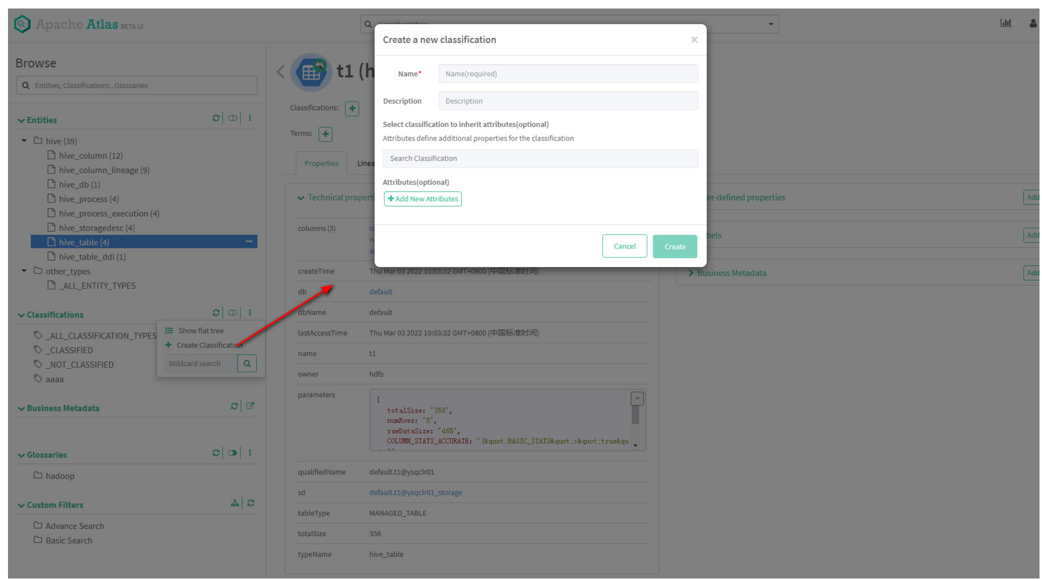


图2-41 通过表 Name 查看表的详情



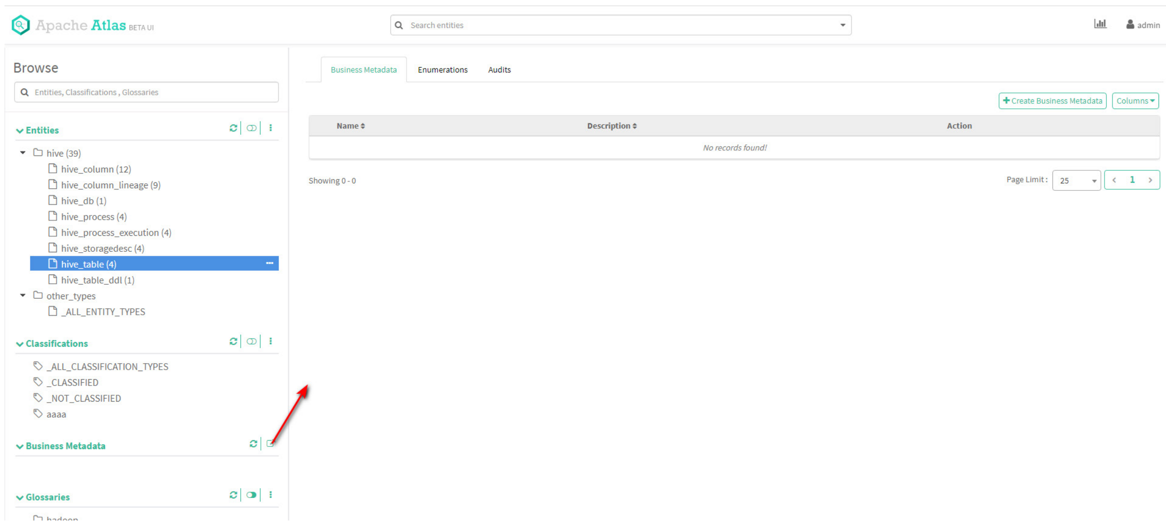
(3) 如图 2-42 所示，在 Classifications 菜单中，可以添加类别，供 Entities 中具体的元数据使用。

图2-42 Classifications 菜单



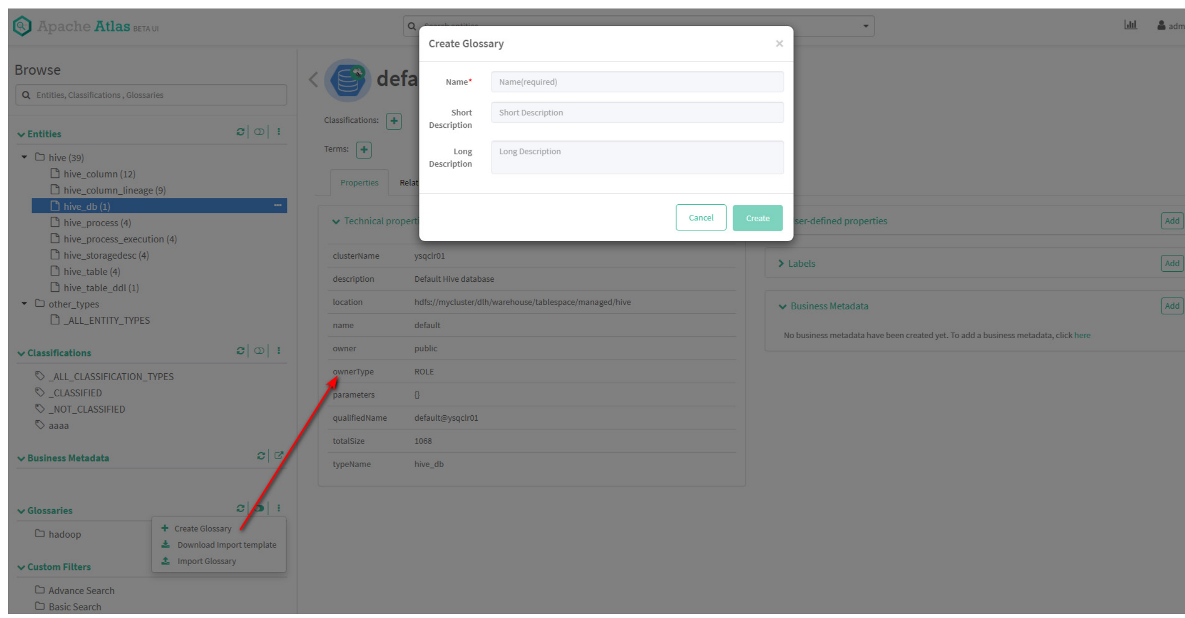
(4) 如图 2-43 所示，在 Business Metadata 菜单中，可以添加业务元数据、枚举值等进行管理，业务元数据可以供 Entities 中具体的元数据使用。

图2-43 Business Metadata 菜单



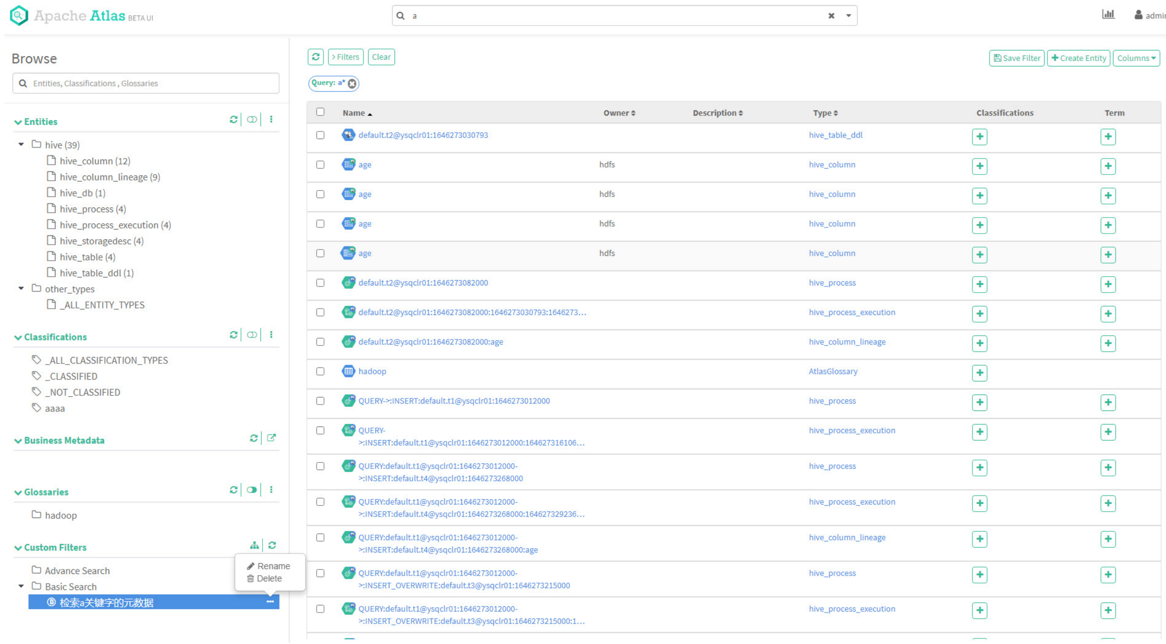
(5) 如图 2-44 所示，在 Glossaries 菜单中，可以管理词汇表，词汇表可以运用在 Entities 中具体的元数据中。

图2-44 Glossaries 菜单



(6) 如图 2-45 所示，在 Custom Filters 菜单中，显示保存的自定义查询，在此处可以进行重命名、删除以及快速定位原来查询条件的查询结果。

图2-45 Custom Filters 菜单



3 使用指南

3.1 离线查询＆交互式查询

DLH 离线查询功能完全兼容 Hive，同时支持基于 Presto 的交互式查询功能。并且，当前版本中，已实现离线查询和交互式查询引擎的自动选择功能。

3.1.1 语法介绍

1. 离线查询语法

DLH 基于 Hive 开发，离线查询语法完全兼容 Hive 语法。

2. 交互式查询语法

DLH 基于 Presto 提供交互式查询能力，SQL 语句兼容 Presto 基本查询语法。另外，Presto 自身也支持 CLI 和 JDBC 接口进行查询。

说明

- Presto 查询语法请参见 Trino 官网(v316)。
- 外部数据源表示数据源和 DLH 组件不属于同一个大数据集群。DLH 不支持对外部数据源的库表查询引擎切换。

3.1.2 配置说明

DLH 对非外部数据源的库表进行 SQL 查询时，可根据业务需要选择合适的执行引擎，引擎选择开关由表 3-1 中所示配置决定。

表3-1 离线查询＆交互式查询执行引擎相关配置

配置项	默认值	说明
dlh.query.engine	hive	执行引擎选项支持hive、presto、auto。该值为hive表示执行引擎固定为Hive，该值为presto表示执行引擎固定为Presto，该值为auto时默认执行引擎为Presto（若使用Presto执行失败或表大小超设定阈值则转交至Hive执行）
hive.presto.table.threshold	10737418240	表大小默认阈值为10GB，单位为B。当执行引擎为auto时，若表容量小于该阈值则查询走presto执行，若表容量大于该阈值则查询走hive执行
hive.presto.http-server.http.port	-	Presto的协调器节点的地址，比如：http://localhost:18089。若集群开启了HA，可配置为虚拟主机名及代理端口，比如：http://testclr-vserver.vip.hde.com:18090，这部分信息可以从Presto组件的自定义config配置http.proxy.url获取

3.1.3 示例

(1) 默认配置下，DLH 非外部数据源的库表查询默认由 Hive 执行

```
0: jdbc:hive2://qcj39.hde.com:2181,qcj40.hde.> set dlh.query.engine;
+-----------------------+
|         set           |
+-----------------------+
| dlh.query.engine=hive|
+-----------------------+
1 row selected (0.017 seconds)
```

(2) 设置 DLH 非外部数据源的库表查询引擎为 Presto，此时 SQL 查询由交互式查询引擎执行

```
0: jdbc:hive2://qcj39.hde.com:2181,qcj40.hde.> set dlh.query.engine = presto;
No rows affected (0.01 seconds)
```

(3) 执行 SQL 查询，此时查询任务由 Presto 完成（可通过 DLH 组件管理界面的快速链接查看对应任务）

```
0: jdbc:hive2://qcj39.hde.com:2181,qcj40.hde.> select count(*) from reason;
+------+
| c0   |
+------+
| 35   |
+------+
1 row selected (2.877 seconds)
```

(4) 设置 DLH 非外部数据源的库表查询引擎为 auto，表大小阈值为 1GB（表大小默认阈值为 10GB），此时超过 1G 的表走 Hive 执行，小于 1G 的表走 Presto 执行

```
0: jdbc:hive2://qcj39.hde.com:2181,qcj40.hde.> set dlh.query.engine = auto;
No rows affected (0.01 seconds)
0: jdbc:hive2://qcj39.hde.com:2181,qcj40.hde.> set
hive.presto.table.threshold=10485760;
No rows affected (0.01 seconds)
```

3.2 外部数据源

说明

- 外部数据源表示数据源和 DLH 组件不属于同一个大数据集群。
- 外部数据源配置存放资源文件的路径时（如：hdfs-site.xml、core-site.xml 或认证文件 krb5.conf、keytab 等），仅支持/etc、/opt、/usr/hdp、/tmp、/apps/presto 及其子目录。
- 外部数据源分析基于 Presto 实现，在 DLH 中执行需要把执行引擎设置为 presto。

DLH 组件支持对 Hive、SeaSQL MPP、DataEngine MPP、MySQL、HBase 等外部数据源进行跨源查询，避免数据迁移。

3.2.1 语法介绍

- 注册数据源

create catalog （if not exists) <catalogName> using <catalogType> with properties ("" = "", "" = "", ...);

其中：

- catalogType 支持 Hive、HBase、SeaSQL MPP、DataEngine MPP、MySQL 等，不区分大小写。
- 注册数据源时，不同的数据源需要的配置项不同，详情请参见 3.2.2 ~3.2.6 章节。比如，注册 hive 数据源时，命令如下：

  create catalog if not exists default using **hive** with properties ("hive.metastore.uri" = "thrift://node1.hde.com:19083","hive.config.resources"="/etc/hadoop/conf/core-site.xml,/etc/hadoop/conf/hdfs-site.xml","hive.allow-drop-table"="true","hive.allow-rename-table"="true","hive.parquet.use-column-names"="true");

- 获取所有的数据源信息

  show catalogs;

- 切换数据源

  use catalog <catalogName>;

- 删除数据源

  drop catalog <catalogName>;

- 切换到数据源下的某个数据库（或 schema）

  use <catalogName>.<databaseName>;

- 查看某个数据源的创建数据源语句

  show create catalog <catalogName>;

- 查看某个数据源下面的所有库列表

  show databases in <catalogName>或 show databases from <catalogName>;

- 查看某个数据源下面的所有表列表

  show tables in <catalogName>.<databaseName>;

- 读取某个数据源下的某个库下的某个表数据

  select * from <catalogName>.<databaseName>.<tableName>;

3.2.2 Hive 数据源

Hive 数据源支持通过 DLH 组件进行创库、建表、查询的操作。

1. 基础配置

表3-2 Hive 数据源相关配置

配置项	默认值	说明
hive.metastore.uri	-	使用Thrift协议连接Hive元存储的URL。如果提供了多个URL，则默认使用第一个URL。该属性必选。 示例：thrift://192.0.2.3:9083或thrift://192.0.2.3:9083,thrift://192.0.2.4:9083
hive.config.resources	-	HDFS配置文件列表，以逗号分隔。该文件必须存在于部署DLH组件的所有节点上。 示例：/etc/hdfs-site.xml

配置项	默认值	说明
hive.storage-format	ORC	创建新表时采用的默认文件格式，支持ORC、PARQUET、AVRO、RCBINARY、SEQUENCEFILE、JSON、TEXTFILE、CSV
hive.parquet.use-column-names	false	配置为true时，查询读取Parquet存储格式的表（如hudi表）时，使用列名而非下标进行组装数据，可避免元数据schema修改后列名与数据对应错位的问题

2. 安全相关配置

通过在 Hive 目录属性文件中设置 hive.security 属性可以启用 Hive 的授权检查，hive.security 属性必须是表 3-3 中的值。

表3-3 Hive 数据源安全相关配置

属性值	说明
legacy（默认值）	授权检查很少执行，因此大多数操作都是允许的。使用配置属性hive.allow-drop-table、hive.allow-rename-table、hive.allow-add-column、hive.allow-drop-column和hive.allow-rename-column来限定数据源是否进行相应操作
read-only	允许读数据和读取元数据的操作（如SELECT），不允许写数据或者写元数据的操作（如CREATE、INSERT或DELETE）

3. Kerberos 相关配置

Hive 数据源开启 Kerberos 时，需要的额外配置如表 3-4 所示。

表3-4 Kerberos 环境下额外配置

配置项	默认值	说明
hive.metastore.authentication.type	NONE	Hive元存储身份验证类型，可选择：NONE或KERBEROS
hive.metastore.service.principal	-	Hive元存储服务的Kerberos主体，Hive数据源开启Kerberos时需要。此属性值中可使用_HOST占位符 示例：hive/_HOST@DLHWHJ02.COM
hive.metastore.client.principal	-	在连接到Hive元存储服务时将使用的Kerberos主体，Hive数据源开启Kerberos时需要 示例：testshare@TESTSHARE.COM
hive.metastore.client.keytab	-	Hive元存储客户端的keytab位置，Hive数据源开启Kerberos时需要
hive.metastore.krb5.conf.path	-	Kerberos服务器地址，Hive数据源开启Kerberos时需要
hive.hdfs.authentication.type	-	身份验证类型，可选择：NONE或KERBEROS
hive.hdfs.presto.principal	-	HDFS客户端Kerberos主体，Hive数据源开启Kerberos时需要
hive.hdfs.presto.keytab	-	HDFS客户端Kerberos认证文件，Hive数据源开启Kerberos时需要

注意

Hive 数据源开启 Kerberos 时：

- 若想通过 DLH 集群访问 Hive 外部数据源，需要进行相关互信配置，详情请参见 3.2.2 4. (2)示例 2：创建并访问开启 Kerberos 的 Hive 数据源示例。
- 通过 DLH 组件查看外部 Hive 数据源的库表时，与通过 Hive 查看时数据类型有所不同。比如：show databases 时，会多显示一个 information_schema 数据库。

4. 示例

(1) 示例 1：创建并访问未开启 Kerberos 的 Hive 外部数据源

a. 将 Hive 外部数据源对应的 HDFS 配置文件：core-site.xml 文件和 hdfs-site.xml 文件，拷贝到安装 DLH 组件的所有节点上（示例：拷贝至/opt/190 目录下）

b. 将 Hive 外部数据源集群节点的映射关系（hosts 文件内容）添加到安装 DLH 组件的所有节点上的/etc/hosts 文件中

c. 后台登录 DLH 控制台，创建名为 hive2 数据源，该数据源访问集群中 xldwhj190.hde.com 节点的 hive

```
create catalog hive2
  using hive with properties (
  "hive.metastore.uri" = "thrift://xldwhj190.hde.com:9083",
  "hive.config.resources" = "/opt/190/core-site.xml,/opt/190/hdfs-site.xml"
);
```

d. 切换数据源为 hive2

```
use catalog hive2;
```

e. 创建名为 db2 的数据库

```
create database db2;
```

f. 切换到 hive2.db2 的数据库下

```
use hive2.db2;
```

g. 创建 t11 表

```
create table t11(id int, name string);
```

h. 查看 t11 表的建表语句

```
show create table t11;
```

i. 插入一条记录

```
insert into t11 values(0, 'a');
```

j. 查询 t11 表中数据

```
select * from t11;
```

(2) 示例 2：创建并访问开启 Kerberos 的 Hive 数据源

a. 将 Hive 外部数据源对应的 HDFS 配置文件：core-site.xml 文件和 hdfs-site.xml 文件，拷贝到安装 DLH 组件的所有节点上

b. 将 Hive 外部数据源集群节点的映射关系（hosts 文件内容）添加到安装 DLH 组件的所有节点上的/etc/hosts 文件中

c. 登录大数据集群的管理系统，在[集群权限/用户管理]页面下载对应外部集群用户的认证文件

d. 将下载的外部集群用户认证文件，拷贝到安装 DLH 组件的所有节点上（路径为创建 catalog 时指定的路径）

e. 将目的数据源集群的 krb5.conf 文件中并入到数据湖所在集群的/etc/krb5.conf 文件，realms 内容直接拷贝，domain 信息需要将目的集群和本集群的机器名逐一采用机器名=集群名的格式进行编辑（仅修改本地集群即可），示例如图 3-1 所示

图3-1 编辑数据湖所在集群的/etc/krb5.conf 文件

```
[realms]
    W14100A.COM = {
        kdc = de002.hde.com
        admin_server = de002.hde.com
        kdc = de001.hde.com
        admin_server = de001.hde.com
        database_module = openldap_ldapconf
    }
     DLHWHJ02.COM = {
        kdc = xldwhj190.hde.com
        admin_server = xldwhj190.hde.com
        database_module = openldap_ldapconf
    }
[domain_realm]
    de001.hde.com = W14100A.COM
    de002.hde.com = W14100A.COM
    de003.hde.com = W14100A.COM
    xldwhj190.hde.com = DLHWHJ02.COM
    xldwhj191.hde.com = DLHWHJ02.COM
    xldwhj192.hde.com = DLHWHJ02.COM
```

f. 后台登录 DLH 控制台，创建一个名为 hive190 的数据源，该数据源可以访问目的集群的 Hive

```
create catalog hive190 using hive with properties (
  "hive.metastore.uri" = "thrift://xldwhj190.hde.com:9083",-- 外部数据源 HiveMetastore 地址，多个英文逗号隔开
  "hive.allow-drop-table" = "true",
  "hive.config.resources" = "/opt/190/core-site.xml,/opt/190/hdfs-site.xml",
  "hive.hdfs.authentication.type" = "KERBEROS",
  "hive.metastore.authentication.type" = "KERBEROS",
  "hive.metastore.service.principal" = "hive/_HOST@DLHWHJ02.COM",--固定格式，域名为目的集群的 realm
  "hive.hdfs.presto.principal" = "admin123@DLHWHJ02.COM",-- 目的集群用户 principal
  "hive.hdfs.presto.keytab" = "/opt/190/admin123.keytab",-- 目的集群用户 keytabl
  "hive.metastore.krb5.conf.path" = "/etc/krb5.conf",
  "hive.metastore.client.keytab" = "/opt/190/admin123.keytab",-- 目的集群用户 keytab
  "hive.metastore.client.principal" = "admin123@DLHWHJ02.COM"--目的集群用户 principal
);
```

(3) 示例 3：创建指定存储格式的数据源（根据需要添加开启 Kerberos 或未开启 Kerberos 所需要的参数项）

其中："hive.storage-format" = "PARQUET"属性用于指定存储格式。

```
create catalog hive3 using hive with properties (
  "hive.metastore.uri" = "thrift://xldwhj190.hde.com:9083",
  "hive.storage-format" = "PARQUET",
  "hive.config.resources" = "/opt/190/core-site.xml,/opt/190/hdfs-site.xml"
```

```
    ……
);
验证建表后，查看表存储格式是否为parquet:
create table t12(id int, name string);
show create table t12;
```

(4) 示例 4：创建安全模式为 legacy 的数据源（根据需要添加开启 Kerberos 或未开启 Kerberos 所需要的参数项）

其中："hive.allow-drop-table" = "true","hive.allow-drop-column"="true","hive.allow-rename-column"="true","hive.allow-add-column"="true","hive.allow-rename-table"="true"等属性用于指定安全模式为 legacy。

……

```
create catalog hive4 using hive with properties (
  "hive.metastore.uri" = "thrift://xldwhj190.hde.com:9083",
  "hive.config.resources" = "/opt/190/core-site.xml,/opt/190/hdfs-site.xml",
  "hive.allow-drop-table" = "true",
  "hive.allow-drop-column"="true",
  "hive.allow-rename-column"="true",
  "hive.allow-add-column"="true",
  "hive.allow-rename-table"="true"
  ……
);
```

验证删除表、删除列、表重命名、添加列、列重命名是否可行（True 表示支持）：

```
alter table t11 RENAME TO t11new;
alter table t11 add column age string;
alter table t11 RENAME COLUMN name TO namenew ;
alter table t11 DROP COLUMN id;
drop table t11new;
```

(5) 示例 5：创建安全模式为 read-only 的数据源（根据需要添加开启 Kerberos 或未开启 Kerberos 所需要的参数项）

其中："hive.security" = "read-only"属性用于指定安全模式为 read-only。

```
create catalog hive3 using hive with properties (
  "hive.metastore.uri" = "thrift://xldwhj190.hde.com:9083",
  "hive.config.resources" = "/opt/190/core-site.xml,/opt/190/hdfs-site.xml",
  "hive.security" = "read-only"
  ……
);
验证 insert、alter 等操作是否已经不可用:
insert into t11 values(0, 'a');
```

3.2.3 SeaSQL MPP 数据源

SeaSQL MPP 数据源支持通过 DLH 组件进行查询和创建表。

注意

SeaSQL MPP 数据源不支持的功能包括：CREATE DATABASE、CREATE SCHEMA、UPDATE、DELETE、GRANT、REVOKE、SHOW GRANTS、SHOW ROLES、SHOW ROLE GRANTS 等语法。

1. 数据源配置

通过表 3-5 所示参数，可以配置 SeaSQL MPP 数据源连接器。

表3-5 SeaSQL MPP 数据源相关配置

配置项	默认值	说明
connection-url	-	连接SeaSQL MPP的地址，比如： jdbc:postgresql://example.net:5432/database
connection-user	-	连接SeaSQL MPP的用户名，比如：root
connection-password	-	连接SeaSQL MPP的密码，比如：secret

2. 示例

创建 SeaSQL MPP 数据源并对数据源进行查看数据库、创建表等操作。

- 创建一个名叫 seqsql1 的数据源

```
create catalog seqsql1 using seasql with properties (
  "connection-url"="jdbc:postgresql://10.121.88.114:5434/postgres",
  "connection-user"="ssadmin", "connection-password"="passwd"
);
```

- 查看 seqsql1 下的数据库

```
show databases from seqsql1;
```

- 查看 seqsql1.public 下的表

```
show tables  from seqsql1.public;
```

- 创建一个 t1 的表

```
create table seqsql1.public.t1 (i int , s string);
```

- 向表 t1 中插入一条记录

```
insert into seqsql1.public.t1 values(0, 'a');
```

- 查询表 t1 中数据

```
select * from seqsql1.public.t1;
```

3.2.4 DataEngine MPP 数据源

DataEngine MPP 数据源支持通过 DLH 组件进行查询和创建表。

注意

DataEngine MPP 数据源不支持的功能包括：CREATE DATABASE、CREATE SCHEMA、UPDATE、DELETE、GRANT、REVOKE、SHOW GRANTS、SHOW ROLES、SHOW ROLE GRANTS 等语法。

1. 数据源配置

通过表 3-6 所示参数，可以配置 DataEngine MPP 数据源连接器。

表3-6 DataEngine MPP 数据源相关配置

配置项	默认值	说明
connection-url	-	连接DataEngine MPP的地址，比如：jdbc:vertica://example.net:5433/database
connection-user	-	连接DataEngine MPP的用户名，比如：root
connection-password	-	连接DataEngine MPP的密码，比如：secret

2. 示例

创建 DataEngine MPP 数据源并对数据源进行查看数据库、创建表等操作。

- 创建一个名叫 test01 的数据源

```
create catalog test01 using vertica with properties (
  "connection-url"="jdbc:vertica://10.121.66.152:5433/Database",
  "connection-user"="dbadmin", "connection-password"="passwd"
);
```

- 查看 test01 下的模式

```
show databases from test01;
```

- 查看 test01.public 下的表

```
show tables  from test01.public;
```

- 切换到 test01.public 模式下

```
use test01.public;
```

- 创建一个 test1 的表

```
create table test1(i int ,s  string);
```

- 向表 test1 中插入一条记录

```
insert into  test1 values(0, 'a');
```

- 查询表 test1 中数据

```
select * from vertical.public. test1;
```

3.2.5 MySQL 数据源

MySQL 数据源支持通过 DLH 组件进行查询和创建表。

注意

MySQL 数据源不支持的功能包括：CREATE DATABASE、CREATE SCHEMA、UPDATE、DELETE、GRANT、REVOKE、SHOW GRANTS、SHOW ROLES、SHOW ROLE GRANTS 等语法。

1. 数据源配置

通过表 3-7 所示参数，可以配置 MySQL 数据源连接器。

表3-7 MySQL 数据源相关配置

配置项	说明
connection-url	连接MySQL的地址，比如：jdbc:mysql://example.net:3306
connection-user	连接MySQL的用户名，比如：root
connection-password	连接MySQL的密码，比如：secret

2. 示例

创建 MySQL 数据源并对数据源进行查看数据库、创建表等操作。

- 创建一个名叫 mysql12 的数据源

```
create catalog mysql12 using mysql with properties (
  "connection-url"="jdbc:mysql://10.121.65.63:53306", "connection-user"="user01",
  "connection-password"="passwd"
);
```

- 查看 mysql12 下的数据库

```
show databases from mysql12;
```

- 查看 mysql12.ambari_hivedebug0908 下的表

```
show tables  from mysql12.hivedebug0908;
```

- 创建一个 a 的表

```
create table mysql12.hivedebug0908.a(a string);
```

- 向表 a 中插入一条记录

```
insert into mysql12.hivedebug0908.a values('test');
```

- 查询表 a 中数据

```
select * from mysql12.hivedebug0908.a;
```

3.2.6 HBase 数据源

支持通过 DLH 在 HBase 实例上查询和创建表。用户可以在 HBase 连接器中创建表，并映射到 HBase Cluster 中已有的表（或创建新表），支持 insert、select 和 delete 操作。HBase 连接器维护着一个元存储，用于持久化 HBase 元数据，目前此元存储采用 MySQL。

注意

- 基本上支持所有的 SQL 语句，包括创建、查询、删除模式、添加、删除、修改表、插入数据、删除行等。但不支持对表进行重命名、不支持删除列。
- HBase 没有提供接口来检索表的元数据，所以 show tables 只能显示用户已与 HBase 数据源建立关联的表。
- 支持 10 种数据类型：VARCHAR、TINYINT、SMALLINT、INTEGER、BIGINT、DOUBLE、BOOLEAN、TIME、DATE 和 TIMESTAMP。
- HBase 连接器支持下推大部分运算符，比如基于 RowKey 的点查询、基于 RowKey 的范围查询等。此外，还支持通过谓词条件进行下推，比如：=、>=、>、<、<=、!=、in、not in、is null、is not null、between and。
- 在开启“安全管理/权限管理”的大数据集群中，访问 HBase 数据源时，需要保证用户拥有 HBase 数据源的相应操作权限。

1. 配置

表3-8 注册 HBase 数据源所需属性配置

配置项	默认值	必填	说明
hbase.zookeeper.quorum	-	是	ZooKeeper集群地址，若包含多个地址以逗号分隔
hbase.zookeeper.property.clientPort	-	是	Zookeeper客户端端口
hbase.zookeeper.znode.parent	/hbase	否	HBase在Zookeeper上的znode路径
hbase.client.retries.number	3	否	HBase客户端连接重试次数
hbase.client.pause.time	100	否	HBase客户端断连时间
hbase.rpc.protection.enable	false	否	通信隐私保护。可以从hbase-site.xml获取该属性的值
hbase.default.value	NULL	否	表中数据的默认值
hbase.authentication.type	NONE	否	HBase数据源的身份验证类型，可选择：NONE或KERBEROS
hbase.kerberos.principal	-	否	认证所需的principal信息。仅当hbase.authentication.type=KERBEROS时，需要配置
hbase.kerberos.keytab	-	否	认证所需的keytab文件位置。仅当hbase.authentication.type=KERBEROS时，需要配置
hbase.krb5.conf.path	-	否	KERBEROS的krb5.conf配置文件的路径。仅当hbase.authentication.type=KERBEROS时，需要配置

注意

Hive 数据源开启 Kerberos 时，若想通过 DLH 集群访问 HBase 外部数据源，需要进行相关互信配置，详情请参见 3.2.6 3. (2)示例 2：创建开启权限管理和开启 Kerberos 的 HBase 数据源示例。

2. 表属性

配置项	默认值	必填	说明
column_mapping	同一个族中的所有列	否	指定表中列族与列限定符的映射关系。 【说明】如果需要链接HBase数据源中的表，则column_mapping信息必须与HBase数据源一致；如果创建一个HBase数据源中不存在的新表，则column_mapping由用户指定。
row_id	第一个列名	否	HBase表中RowKey对应的列名
hbase_table_name	-	否	指定要链接的HBase数据源上的命名空间和表名，使用“:”连接表空间和表名，默认表空间为“default”。 【说明】要求该表在HBase中存在，且表对应的列族相同。
external	true	否	• 如果 external 为 true，表示该表是 HBase 数据源中表的映射表。此时，不支持删除 HBase 数据源上原有的表。 • 当 external 为 false 时，删除本地 HBase 表的同时也会删除 HBase 数据源上的表。

3. 示例

(1) 示例 1：创建未开启 Kerberos 的 HBase 数据源

a. 将 HBase 外部数据源对应的 HDFS 配置文件：hbase-site.xml 文件，拷贝到安装 DLH 组件的所有节点上

b. 将 HBase 外部数据源集群节点的映射关系（hosts 文件内容）添加到安装 DLH 组件的所有节点上的/etc/hosts 文件中

c. 登录管理系统，在[集群权限/用户管理]页面，在 HBase 外部数据源对应的集群中，创建 user01 用户，并赋予命名空间 d1 的 admin 权限，赋予表 h1 的所有权限

d. 后台登录 DLH 控制台，创建名为 hbase1 数据源：

```
create catalog hbase1 using hbase with properties
("hbase.zookeeper.quorum"="test01.hde.com,test02.hde.com,test03.hde.com",
"hbase.zookeeper.property.clientPort"="2181","hbase.zookeeper.znode.parent"="/hb
ase-unsecure");
```

(2) 示例 2：创建开启权限管理和开启 Kerberos 的 HBase 数据源

a. 将 HBase 外部数据源对应的 HDFS 配置文件：hbase-site.xml 文件，拷贝到安装 DLH 组件的所有节点上（如：/opt/190）

b. 将 HBase 外部数据源集群节点的映射关系（hosts 文件内容）添加到安装 DLH 组件的所有节点上的/etc/hosts 文件中

c. 登录大数据集群的管理系统，在[集群权限/用户管理]页面下载对应集群用户的认证文件

d. 将下载的集群用户认证文件，拷贝到安装 DLH 组件的所有节点上（路径为创建 catalog 时指定的路径，如：/opt/190）

e. 将目的数据源集群的 krb5.conf 文件中并入到数据湖所在集群的/etc/krb5.conf 文件，realms 内容直接拷贝，domain 信息需要将目的集群和本集群的机器名逐一采用机器名=集群名的格式进行编辑，示例如图 3-2 所示

图3-2 编辑数据湖所在集群的/etc/krb5.conf 文件

```
[realms]
    W14100A.COM = {
        kdc = de002.hde.com
        admin_server = de002.hde.com
        kdc = de001.hde.com
        admin_server = de001.hde.com
        database_module = openldap_ldapconf
    }
     DLHWHJ02.COM = {
        kdc = xldwhj190.hde.com
        admin_server = xldwhj190.hde.com
        database_module = openldap_ldapconf
    }
[domain_realm]
    de001.hde.com = W14100A.COM
    de002.hde.com = W14100A.COM
    de003.hde.com = W14100A.COM
    xldwhj190.hde.com = DLHWHJ02.COM
    xldwhj191.hde.com = DLHWHJ02.COM
    xldwhj192.hde.com = DLHWHJ02.COM
```

f. 后台登录 DLH 控制台，创建一个名为 hbase190 的数据源，该数据源可以访问目的集群的 HBase

```
create catalog if not exists hbase190 using hbase with properties (
  "hbase.zookeeper.quorum"="xldwhj190.hde.com,xldwhj191.hde.com,xldwhj192.hde.com",-- 目的集群 HBase 的 ZK 地址
  "hbase.zookeeper.property.clientPort"="2181",-- 目的集群 HBase 的 ZK 端口
  "hbase.zookeeper.znode.parent"="/hbase-secure",--目的集群 HBase 的 ZK 的 znode 路径
  "hbase.hbase.site.path"="/opt/190/hbase-site.xml",--目的集群 hbase-site.xml
  "hbase.krb5.conf.path"="/etc/krb5.conf",
  "hbase.kerberos.keytab"="/opt/190/admin123.keytab",--目的集群用户 keytab
  "hbase.kerberos.principal"="admin123@DLHWHJ02.COM",--目的集群用户 principal
  "hbase.authentication.type"="KERBEROS"
);
```

g. 创建 hbase190.d1 数据库

```
create database hbase190.d1;
```

h. 建表

– 创建 hbase190.d1.t1 表

```
create table hbase190.d1.t1 (rowid string,id int,name string,sex int)
with(external=false);
```

– 查看 hbase190 下的数据库

```
show databases in hbase190;
```

– 指定表属性 column_mapping、row_id、hbase_table_name 建表

```
CREATE TABLE hbase159. d1.h2(
   rowId       VARCHAR,
   qualifier1  VARCHAR,
   qualifier2  SMALLINT,
   qualifier3  INTEGER,
```

```
    qualifier4 BIGINT,
    qualifier5 DOUBLE,
    qualifier6 BOOLEAN,
    qualifier7 TIME,
    qualifier8 DATE,
    qualifier9 TIMESTAMP)
 WITH (
     column_mapping = 'qualifier1:f1:q1, qualifier2:f1:q2,
     qualifier3:f2:q3, qualifier4:f2:q4, qualifier5:f2:q5, qualifier6:f3:q1,
     qualifier7:f3:q2, qualifier8:f3:q3, qualifier9:f3:q4',
     row_id = 'rowId',
     hbase_table_name = 'noexternal1:table1'
 );
```

i. 插入一条数据

```
insert into hbase159.d1.h1 values('a', 0,'aaa', 1);
```

j. 查询 hbase159.d1.h1 表中数据

```
select * from hbase159.d1.h1;
```

3.2.7 跨源分析

DLH 支持 Hive、SeaSQL MPP、DataEngine MPP、MySQL、HBase 数据源的跨源分析，遵循标准的 select 语法。

示例：

结合 Mysql 数据源和 Hive 数据源（未开启 kerberos 的环境），进行 Mysql 和 Hive 之间的表联合查询的详细说明。操作如下：

(1) 进入 DLH 控制台，创建名为 mysql 数据源（执行此操作前，要求已存在 MySQL 外部数据源）

```
create catalog if not exists mysql using mysql with
properties("connector.name"="mysql","connection-url"="jdbc:mysql://0.0.0.0:3306","connection-user"="xxxxx","connection-password"="xxxxx");
```

(2) 切换数据源为 mysql

```
use catalog mysql;
```

(3) 查看数据源 mysql 中已有数据库（因连接器限制，不支持 create、insert、update、delete、drop 操作）

```
show databases;
```

(4) 切换到 bigdata_cluster 的数据库下

```
use bigdata_cluster;
```

(5) 查看 tbl_bigdata_resource_hosts 表的建表语句

```
show create table tbl_bigdata_resource_hosts;
```

(6) 创建名为 hive1 数据源，该数据源访问集群中 node1.hde.com 节点的 hive。（此时：value 值需根据情况改动）

```
create catalog hive1
  using hive with properties (
  "hive.metastore.uri" = "thrift://node1.hde.com:19083",
  "hive.config.resources" = "
/etc/hadoop/conf/core-site.xml,/etc/hadoop/conf/hdfs-site.xml"
);
```

(7) 切换数据源为 hive1

```
use catalog hive1;
```

(8) 创建名为 db2 的数据库

```
create database db2;
```

(9) 切换到 hive1.db2 的数据库下

```
use hive1.db2;
```

(10) 创建 t11 表

```
create table t11(host_id integer, name string);
```

(11) 查看 t11 表的建表语句

```
show create table t11;
```

(12) 插入一条记录

```
insert into t11 values(58, 'a');
```

(13) 查询 t11 表中数据

```
select * from t11;
```

(14) 进行 Mysql 与 Hive 之间的表联合查询

```
select * from mysql.bigdata_cluster.tbl_bigdata_resource_hosts t1 join hive1.db2.t11
b on t1.host_id=b.host_id;
```

3.3 Hudi存储格式

3.3.1 Hudi 存储格式相关术语

为方便理解与 Hudi 存储格式相关的重要概念，相关术语说明如表 3-9 所示。

表3-9 Hudi 存储格式重要概念说明

术语	描述
cow	一种存储类型，写时复制
mor	一种存储类型，读时合并
ro表	实现了由HoodieParquetInputFormat支持的数据集的读优化视图，从而提供了纯列式数据的视图
rt表	实现了由HoodieParquetRealtimeInputFormat支持的数据集的实时视图，从而提供了基础数据和日志数据的合并视图

3.3.2 文件组织形式

Hudi 存储格式将 HDFS 上的数据集组织到基本路径下的目录结构中。

(1) 数据集分为多个分区，这些分区是包含该分区数据文件的文件夹，这与 Hive 表非常相似。

(2) 在每个分区内，以文件组形式管理分区内各个文件，每个文件由文件 id 唯一标识。

(3) 每个文件组包含多个文件切片，其中每个切片包含数据文件(*.parquet)以及日志文件(*.log*)，日志文件包含对数据文件的插入/更新操作记录。

(4) 采用 MVCC 设计，其中压缩操作将日志文件和数据文件合并以产生新的文件片，而清理操作则将未使用的/较旧的文件片删除以回收 HDFS 上的空间。

(5) 通过索引机制将给定的 hoodie 键（记录键+分区路径）映射到文件组，从而提供了高效的更新操作。一旦将记录的第一个版本写入文件，记录键和文件组/文件 id 之间的映射就永远不会改变。简而言之，映射的文件组包含一组记录的所有版本。

3.3.3 存储类型和视图

Hudi 存储格式存储类型定义了在 HDFS 上对数据进行索引和布局的方式，以及在这种组织之上写入数据的方式。与之相反，视图则定义了读取数据的方式。

表3-10 存储类型和视图的关系

存储类型	支持的视图
写时复制	读优化 + 增量
读时合并	读优化 + 增量 + 近实时

1. 存储类型

- 写时复制：仅使用列文件格式（例如 parquet）存储数据。通过在写入过程中执行同步合并以更新版本并重写文件。
- 读时合并：使用列式（例如 parquet）+ 基于行的文件格式（例如 avro）组合来存储数据。更新记录到增量文件中，然后进行同步或异步压缩以生成列文件的新版本。

表3-11 存储类型特性对比

权衡指标	写时复制	读时合并
数据延迟	更高	更低
更新代价（I/O）	更高（重写整个parquet文件）	更低（追加到增量日志）
parquet文件大小	更小（高更新代价）	更大（低更新代价）
数据冗余（写放大）	更高	更低（取决于压缩策略）

2. 视图

- 读优化视图：在此视图上的查询，将查看指定提交或压缩操作中数据集的最新快照。该视图仅将最新文件切片中的数据文件暴露给查询，并保证与非 Hudi 列式数据集相比，具有相同的列式查询性能。
- 增量视图：在此视图上的查询，只能看到从某个提交或压缩操作后写入数据集的新数据。该视图有效地提供了更改流，来支持查看增量数据。
- 实时视图：在此视图上的查询，将查看某个增量提交操作中数据集的最新快照。该视图通过动态合并最新的数据文件（例如 parquet）和增量文件（例如 avro）来提供近实时数据集（几分钟的延迟）。

表3-12 视图特性对比

权衡指标	读优化	实时
数据延迟	更高	更低

权衡指标	读优化	实时
查询延迟	更低（原始列式性能）	更高（合并列式 + 基于行的增量）

3.3.4 Hudi 属性说明

关于 Hudi 的属性说明如下：

- 属性可以通过 dlh-site、表属性、会话 3 种方式设置。
- 属性优先级：会话 > 表属性 > dlh-site.xml > 默认值。

表3-13 dlh-site 属性说明

属性	描述
hoodie.upsert.shuffle.parallelism	upsert并发数，设置为10，可通过表属性或会话设置覆盖
hoodie.insert.shuffle.parallelism	insert并发数，设置为10，可通过表属性或会话设置覆盖
hoodie.blukinsert.shuffle.parallelism	blukinsert并发数，设置为10，可通过表属性或会话设置覆盖
hoodie.delete.shuffle.parallelism	delete并发数，设置为10，可通过表属性或会话设置覆盖
hoodie.datasource.write.hive_style_partitioning	默认true
hoodie.datasource.write.partitionpath.urlencode	默认true

表3-14 表属性说明

属性	描述
hoodie.datasource.write.table.type	Hudi表类型，可以设置COPY_ON_WRITE或MERGE_ON_READ两种类型，缺失时默认为COPY_ON_WRITE
hoodie.datasource.write.recordkey.field	Hudi表记录主键，value值需要在表的列中
hoodie.datasource.write.precombine.field	数据集插入前合并记录的判断字段，在recode字段值相当的情况下，默认取该字段数值大的记录进行插入
hoodie.table.name	Hudi表名，缺省时默认取Hive表名
hoodie.mor.isRealTimeInputFormat	hoodie.datasource.write.table.type=MERGE_ON_READ时，该属性设置为true，创建mor类型的rt表，否则创建ro表

表3-15 会话属性说明

属性	描述
hoodie.upsert.shuffle.parallelism	upsert并发数，设置为10，可通过表属性或会话设置覆盖
hoodie.insert.shuffle.parallelism	insert并发数，设置为10，可通过表属性或会话设置覆盖
hoodie.blukinsert.shuffle.parallelism	blukinsert并发数，设置为10，可通过表属性或会话设置覆盖
hoodie.delete.shuffle.parallelism	delete并发数，设置为10，可通过表属性或会话设置覆盖

属性	描述
hoodie.datasource.write.operation	Hudi表插入方式，取值为upsert、insert、blukinsert、delete等，缺失时默认为upsert。删除操作时，该值默认为delete

3.3.5 SQL 操作

1. 建表

遵循 Hive 原生建表语句，通过扩展 stored 类型和使用表属性来控制 Hudi 表的创建特性。

功能描述：

- 支持分区表、内部表、外部表语法。
- 建表语句可缺失内置字段，但是建表后查看表可以看到 Hudi 内置字段。建表语句如果包含内置字段，类型必须为 String 类型。
- 建表时需首先明确表明 stored as hudi 来标明表为 Hudi 表。
- 建表时可以通过表属性 hoodie.table.name 设置 hoodie 表名，可以与 Hive 表使用不同的名称。
- 建表时可以通过表属性 hoodie.datasource.write.table.type 设置 Hudi 表存储类型，默认缺失时为 COPY_ON_WRITE 类型。
- 可以通过设置表属性 hoodie.datasource.write.table.type=MERGE_ON_READ 和 hoodie.mor.isRealTimeInputFormat=true 创建 hudi 存储类型为 MERGE_ON_READ 的 rt 视图表。
- MERGE_ON_READ 存储类型的 Hudi 表，如果需要通过 Hive SQL 进行读优化和近实时视图查询，则需要同时建立 2 个 Hive 表同时读取同一套 Hudi 数据，此时需要保证 Hive 建表语句中只有 hoodie.mor.isRealTimeInputFormat=true 有差异，且 hoodie.table.name 必须配置。
- 建表时表属性 hoodie.datasource.write.recordkey.field 与 hoodie.datasource.write.precombine.field 的值必须在表字段中，且不是 hudi 内置字段。

语法：

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
 [(col_name data_type [COMMENT col_comment], ... [constraint_specification])]
 [COMMENT table_comment]
 [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
 STORED AS HUDI
 [LOCATION hdfs_path]
 [TBLPROPERTIES (property_name=property_value, ...)]
[AS select_statement];
```

表3-16 参数说明

表属性	描述
hoodie.datasource.write.table.type	Hudi表类型，可以设置COPY_ON_WRITE或MERGE_ON_READ两种类型，缺失时默认为COPY_ON_WRITE
hoodie.datasource.write.recordkey.field	Hudi表记录主键，value值需要在表的列中，支持多主键，逗号分隔
hoodie.datasource.write.precombine.field	数据集插入前合并记录的判断字段，在recode字段值相当的情况下，默认取该字段数值大的记录进行插入，value值需要在表的列中
hoodie.table.name	Hudi表名，缺省时默认取Hive表名
hoodie.mor.isRealTimeInputFormat	hoodie.datasource.write.table.type=MERGE_ON_READ时，该属性设置为true，创建mor类型的rt表，否则创建ro表

示例：

- 创建内部表

```
CREATE TABLE IF NOT EXISTS `hudi_inner`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
  STORED AS hudi
 TBLPROPERTIES(
 'hoodie.datasource.write.recordkey.field'= 'uuid',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 创建外部表

```
CREATE EXTERNAL TABLE  IF NOT EXISTS `hudi_outer`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
 STORED AS hudi
 TBLPROPERTIES(
 'hoodie.datasource.write.recordkey.field'= 'uuid',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 创建分区表

```
CREATE EXTERNAL TABLE  IF NOT EXISTS `hudi_part`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
 PARTITIONED BY (`driver` string,`name` string)
 STORED AS hudi
 TBLPROPERTIES(
 'hoodie.datasource.write.recordkey.field'= 'uuid',
```

```
 'hoodie.datasource.write.precombine.field'='ts');
```

- 缺失表属性 hoodie.datasource.write.table.type 创建 cow 类型表

```
CREATE EXTERNAL TABLE  IF NOT EXISTS `hudi_trips_cow`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
 PARTITIONED BY (`driver` string,`name` string)
 STORED AS hudi
 LOCATION '/tmp/hudi_trips_cow'
 TBLPROPERTIES(
 'hoodie.datasource.write.recordkey.field'= 'uuid',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 指定表属性'hoodie.datasource.write.table.type' = 'COPY_ON_WRITE'，创建 cow 类型表

```
CREATE EXTERNAL TABLE  IF NOT EXISTS `hudi_trips_cow`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
 PARTITIONED BY (`driver` string,`name` string)
 STORED AS hudi
 LOCATION '/tmp/hudi_trips_cow'
 TBLPROPERTIES(
 'hoodie.datasource.write.table.type' = 'COPY_ON_WRITE',
 'hoodie.datasource.write.recordkey.field'= 'uuid',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 指定表属性'hoodie.table.name' = xxx'，创建 cow 类型表

```
CREATE EXTERNAL TABLE  IF NOT EXISTS `hudi_trips_cow`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
 PARTITIONED BY (`driver` string,`name` string)
 STORED AS hudi
 LOCATION '/tmp/hudi_trips_cow'
 TBLPROPERTIES(
'hoodie.table.name' = 'hudi_trips_cow1 ',
 'hoodie.datasource.write.table.type' = 'COPY_ON_WRITE',
 'hoodie.datasource.write.recordkey.field'= 'uuid',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 创建 mor 的 ro 表

```
CREATE EXTERNAL TABLE  IF NOT EXISTS `default`.`hudi_trips_mor`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
 PARTITIONED BY (`driver` string)
```

```
 STORED AS hudi
 LOCATION '/tmp/hudi_trips_mor'
 TBLPROPERTIES (
'hoodie.table.name' = 'hudi_trips_mor ',
 'hoodie.datasource.write.table.type' = 'MERGE_ON_READ',
 'hoodie.datasource.write.recordkey.field'= 'uuid',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 创建 mor 的 rt 表

```
CREATE EXTERNAL TABLE  IF NOT EXISTS `default`.`hudi_trips_mor_rt`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
 PARTITIONED BY (`driver` string)
 STORED AS hudi
 LOCATION '/tmp/hudi_trips_mor'
 TBLPROPERTIES(
'hoodie.table.name' = 'hudi_trips_mor ',
 'hoodie.mor.isRealTimeInputFormat' = 'true',
 'hoodie.datasource.write.table.type' = 'MERGE_ON_READ',
 'hoodie.datasource.write.recordkey.field'= 'uuid',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 创建多主键表

```
CREATE TABLE IF NOT EXISTS `hudi_inner_multi`(
 `fare` bigint,
 `rider` string,
 `ts` double,
 `uuid` string)
  STORED AS hudi
 TBLPROPERTIES(
 'hoodie.datasource.write.recordkey.field'= 'uuid,fare',
 'hoodie.datasource.write.precombine.field'='ts');
```

- 复制表结构

```
create table `t1` like `hudi_trips_cow`;
```

2. 插入

功能说明：

- 支持 insert into values 语法
- 支持 insert into select 语法
- 支持 insert overwrite values 语法
- 支持 insert overwrite select 语法
- 支持分区表动静结合使用
- 可以通过设置 hoodie.datasource.write.operation 定义 hoodie 的写操作方式，默认为 upsert
- 支持通过 hoodie 属性调整并发数等

表3-17 参数说明

表属性	描述
hoodie.datasource.write.operation	Hudi表插入方式，取值为upsert、insert、blukinsert、delete等，缺失时默认为upsert。删除操作时，该值默认为delete
hoodie.upsert.shuffle.parallelism	upsert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖
hoodie.insert.shuffle.parallelism	insert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖
hoodie.blukinsert.shuffle.parallelism	blukinsert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖

示例：

- 插入表数据

```
insert into hudi_trips_cow partition (driver='driver-213',name)
values(58,'sdf',1.8,25,'nihao');
```

- 使用会话重置属性，插入表数据

```
set hoodie.datasource.write.operation=upsert;
set hoodie.upsert.shuffle.parallelism=2;
insert into hudi_trips_cow partition (driver='driver-213',name)
values(58,'sdf',1.8,25,'nihao');
```

- 将查询结果插入表中

```
insert into table hudi_trips_cow partition (driver,name) select
fare,rider,ts,uuid,'driver-214',name from hudi_trips_cow_test2;
```

- 覆盖插入

```
Insert overwrite table hudi_trips_cow partition (driver='driver-213',name)
values(56,'sdf',1.8,25,'sdf'),(57,'sdf',1.9,25,'nihao');
```

- 指定字段插入

```
insert into hudi_inner (uuid,ts) values('uid1', 22.0);
```

- 插入多主键表

```
insert into hudi_inner_multi(uuid,ts,fare) values('uid1', 22.0, 100);
```

3. 更新

功能说明：

- 遵循 Hive 标准 update 语法，需要手动开启 DLH 的事务特性（DLH 默认是关闭 Hive 的事务特性的），开启方式如下：
  - 设置 dlh-site 中的 hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager
  - 设置 dlh-site 中的 hive.support.concurrency=true

  配置修改完成后，重启 DLH 组件。

表3-18 参数说明

表属性	描述
hoodie.datasource.write.operation	Hudi表插入方式，取值为upsert、insert、blukinsert、delete等，缺失时默认为upsert。删除操作时，该值默认为delete
hoodie.upsert.shuffle.parallelism	upsert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖
hoodie.insert.shuffle.parallelism	insert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖
hoodie.blukinsert.shuffle.parallelism	blukinsert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖

示例：

- 更新表数据

```
update hudi_trips_cow set rider='test' where uuid='25';
```

- 使用 UDF 函数更新表数据

```
update hudi_trips_cow set fare=length('test') where uuid='25';
```

- 使用会话重置属性，更新表数据

```
set hoodie.datasource.write.operation=upsert;
set hoodie.upsert.shuffle.parallelism=3;
update hudi_trips_cow set fare=length('test') where uuid='25';
```

4. 删除

功能说明：

- 遵循 Hive 标准 delete 语法，需要手动开启 DLH 的事务特性（DLH 默认是关闭 Hive 的事务特性的），开启方式如下：
  - 设置 dlh-site 中的 hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager
  - 设置 dlh-site 中的 hive.support.concurrency=true

  配置修改完成后，重启 DLH 组件。

表3-19 参数说明

表属性	描述
hoodie.datasource.write.operation	Hudi表插入方式，取值为upsert、insert、blukinsert、delete等，缺失时默认为upsert。删除操作时，该值默认为delete
hoodie.upsert.shuffle.parallelism	upsert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖
hoodie.insert.shuffle.parallelism	insert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖
hoodie.blukinsert.shuffle.parallelism	blukinsert并发数，全局（hive-site.xml）设置为10，可通过表属性或会话设置覆盖

示例：

- 删除表数据

```
delete from hudi_trips_cow where uuid='25';
```

- 重置删除相关属性，删除表数据

```
set hoodie.delete.shuffle.parallelism=3;
delete from hudi_trips_cow where uuid='25';
```

5. 新增列

功能说明：

- cow 表支持，mor 表不支持
- 遵循 Hive 标准 add COLUMNS 语法

示例：

- 表 hudi_trips_cow 新增列名为 test，类型 String

```
alter table hudi_trips_cow add COLUMNS (`test` String);
```

6. 新增分区

功能说明：

- 遵循 Hive 标准 add partition 语法

示例：

- 新增表 hudi_trips_cow 分区 driver='driver-213',name="sdf"

```
alter table hudi_trips_cow add partition(driver='driver-213',name="sdf") location
'/tmp/hudi_trips_cow/driver=driver-213/name=sdf';
```

7. 删除分区

功能说明：

- 删除分区要求 Hudi 表是分区表，且是外部表。当 Hudi 表是分区表，且是内部表时，删除分区不支持

示例：

- 删除表 hudi_trips_cow 分区 driver='driver-213',name="sdf"

```
alter table hudi_trips_cow drop partition(driver='driver-213',name="sdf");
```

8. 表重命名

功能说明：

- 只修改 Hive 表的名字，Hudi 表名不变

示例：

- 重命名表名 hudi_trips_cow 为 hudi_trips_cow1

```
alter table hudi_trips_cow rename to hudi_trips_cow1;
```

9. 近实时查询

功能说明：

- 支持 COPY_ON_WRITE 存储类型的 Hudi 表
- 支持 MERGE_ON_READ 存储类型的 Hudi 表，且 Hive 表必须是 rt 类型表

表3-20 参数说明

表属性	描述
hoodie.hudi_trips_cow.consume.mode	查询方式：SNAPSHOT、INCREMENTAL。近实时查询只能配置为SNAPSHOT

示例：

- 查询 cow 表

```
set hoodie.hudi_trips_cow.consume.mode=SNAPSHOT;
select * from hudi_trips_cow;
```

- 查询 mor 表（rt 表）

```
set hoodie.hudi_trips_mor.consume.mode=SNAPSHOT;
select * from hudi_trips_mor_rt;
```

【说明】hudi_trips_mor 为 Hudi 表名，hudi_trips_mor_rt 为 Hive 表名，二者不一定相同，Hudi 表名可以通过“show create table +Hive 表名”进行查看。

10. 增量查询

功能说明：

- 查询前需要设置消费模式为 INCREMENTAL
- 支持通过设置开始和增量值进行增量查询

表3-21 参数说明

表属性	描述
hoodie.hudi_trips_cow.consume.mode	查询方式：SNAPSHOT、INCREMENTAL。增量查询只能配置为INCREMENTAL
hoodie.hudi_trips_cow.consume.start.timestamp	INCREMENTAL模式下使用
hoodie.hudi_trips_cow.consume.max.commits	INCREMENTAL模式下使用

示例：

前提条件：查询时需确保消费模式为 INCREMENTAL 类型。

- 查询 cow 表

```
set hoodie.hudi_trips_cow.consume.mode=INCREMENTAL;
set hoodie.hudi_trips_cow.consume.start.timestamp=20201019165434;
set hoodie.hudi_trips_cow.consume.max.commits=1;
select * from hudi_trips_cow where `_hoodie_commit_time` > '20201019165434';
```

- 查询 mor 表（rt 表）

```
set hoodie.hudi_trips_mor.consume.mode=INCREMENTAL;
set hoodie.hudi_trips_mor.consume.start.timestamp=20201019165434;
set hoodie.hudi_trips_mor.consume.max.commits=1;
select * from hudi_trips_mor_rt where `_hoodie_commit_time` > '20201019165434';
```

11. 读优化查询

功能说明：

- 查询前需要设置消费模式为 SNAPSHOT
- 读优化查询只读取 parquet 数据。对于 cow 类型表，读优化查询就是实时查询；但是对于 mor 表，读优化查询只能查询到插入的数据，对于更新参数的 log 数据则无法查询。

表3-22 参数说明

表属性	描述
hoodie.hudi_trips_cow.consume.mode	查询方式：SNAPSHOT、INCREMENTAL。读优化查询只能配置为SNAPSHOT

示例：

- 查询 mor 表（ro 表）

```
set hoodie.hudi_trips_mor.consume.mode=SNAPSHOT;
select * from hudi_trips_mor;
```

3.4 流SQL操作

说明

如果集群手动开启 kerberos 认证，需要在 flink 自定义配置 sql-gateway-flink-conf 新增配置项目 security.kerberos.login.keytab、security.kerberos.login.principal，具体值为集群超级用户对应的 keytab 和 principal，如：/etc/security/keytabs/admin123.keytab、admin123@TENANT11.COM。

3.4.1 保留关键字

DLH 流 SQL 兼容 Flink SQL 能力，虽然 Flink SQL 特性并未完全实现，但是一些字符串及其组合已经被预留为关键字以备未来使用，如表 3-23 所示。如果您希望使用以下字符串作为字段名，请在使用时使用反引号将该字段名包起来（示例`value`, `count` ）。

表3-23 预留字符串

A, ABS, ABSOLUTE, ACTION, ADA, ADD, ADMIN, AFTER, ALL, ALLOCATE, ALLOW, ALTER, ALWAYS,

AND, ANY, ARE, ARRAY, AS, ASC, ASENSITIVE, ASSERTION, ASSIGNMENT, ASYMMETRIC, AT, ATOMIC, ATTRIBUTE, ATTRIBUTES, AUTHORIZATION, AVG, BEFORE, BEGIN, BERNOULLI, BETWEEN, BIGINT, BINARY, BIT, BLOB, BOOLEAN, BOTH, BREADTH, BY, BYTES, C, CALL, CALLED, CARDINALITY, CASCADE, CASCADED, CASE, CAST, CATALOG, CATALOG_NAME, CEIL, CEILING, CENTURY, CHAIN, CHAR, CHARACTER, CHARACTERISTICS, CHARACTERS, CHARACTER_LENGTH, CHARACTER_SET_CATALOG, CHARACTER_SET_NAME, CHARACTER_SET_SCHEMA, CHAR_LENGTH, CHECK, CLASS_ORIGIN, CLOB, CLOSE, COALESCE, COBOL, COLLATE, COLLATION, COLLATION_CATALOG, COLLATION_NAME, COLLATION_SCHEMA, COLLECT, COLUMN, COLUMN_NAME, COMMAND_FUNCTION, COMMAND_FUNCTION_CODE, COMMIT, COMMITTED, CONDITION, CONDITION_NUMBER, CONNECT, CONNECTION, CONNECTION_NAME, CONSTRAINT, CONSTRAINTS, CONSTRAINT_CATALOG, CONSTRAINT_NAME, CONSTRAINT_SCHEMA, CONSTRUCTOR, CONTAINS, CONTINUE, CONVERT, CORR, CORRESPONDING, COUNT, COVAR_POP, COVAR_SAMP, CREATE, CROSS, CUBE, CUME_DIST, CURRENT, CURRENT_CATALOG, CURRENT_DATE, CURRENT_DEFAULT_TRANSFORM_GROUP, CURRENT_PATH, CURRENT_ROLE, CURRENT_SCHEMA, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_TRANSFORM_GROUP_FOR_TYPE, CURRENT_USER, CURSOR, CURSOR_NAME, CYCLE, DATA, DATABASE, DATE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, DAY, DEALLOCATE, DEC, DECADE, DECIMAL, DECLARE, DEFAULT, DEFAULTS, DEFERRABLE, DEFERRED, DEFINED, DEFINER, DEGREE, DELETE, DENSE_RANK, DEPTH, DEREF, DERIVED, DESC, DESCRIBE, DESCRIPTION, DESCRIPTOR, DETERMINISTIC, DIAGNOSTICS, DISALLOW, DISCONNECT, DISPATCH, DISTINCT, DOMAIN, DOUBLE, DOW, DOY, DROP, DYNAMIC, DYNAMIC_FUNCTION, DYNAMIC_FUNCTION_CODE, EACH, ELEMENT, ELSE, END, END-EXEC, EPOCH, EQUALS, ESCAPE, EVERY, EXCEPT, EXCEPTION, EXCLUDE, EXCLUDING, EXEC, EXECUTE, EXISTS, EXP, EXPLAIN, EXTEND, EXTERNAL, EXTRACT, FALSE, FETCH, FILTER, FINAL, FIRST, FIRST_VALUE, FLOAT, FLOOR, FOLLOWING, FOR, FOREIGN, FORTRAN, FOUND, FRAC_SECOND, FREE, FROM, FULL, FUNCTION, FUSION, G, GENERAL, GENERATED, GET, GLOBAL, GO, GOTO, GRANT, GRANTED, GROUP, GROUPING, HAVING, HIERARCHY, HOLD, HOUR, IDENTITY, IMMEDIATE, IMPLEMENTATION, IMPORT, IN, INCLUDING, INCREMENT, INDICATOR, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, INSERT, INSTANCE, INSTANTIABLE, INT, INTEGER, INTERSECT, INTERSECTION, INTERVAL, INTO, INVOKER, IS, ISOLATION, JAVA, JOIN, K, KEY, KEY_MEMBER, KEY_TYPE, LABEL, LANGUAGE, LARGE, LAST, LAST_VALUE, LATERAL, LEADING, LEFT, LENGTH, LEVEL, LIBRARY, LIKE, LIMIT, LN, LOCAL, LOCALTIME, LOCALTIMESTAMP, LOCATOR, LOWER, M, MAP, MATCH, MATCHED, MAX, MAXVALUE, MEMBER, MERGE, MESSAGE_LENGTH, MESSAGE_OCTET_LENGTH, MESSAGE_TEXT, METHOD, MICROSECOND, MILLENNIUM, MIN, MINUTE, MINVALUE, MOD, MODIFIES, MODULE, MONTH, MORE, MULTISET, MUMPS, NAME, NAMES, NATIONAL, NATURAL, NCHAR, NCLOB, NESTING, NEW, NEXT, NO, NONE, NORMALIZE, NORMALIZED, NOT, NULL, NULLABLE, NULLIF, NULLS, NUMBER, NUMERIC, OBJECT, OCTETS, OCTET_LENGTH, OF, OFFSET, OLD, ON, ONLY, OPEN, OPTION, OPTIONS, OR, ORDER, ORDERING, ORDINALITY, OTHERS, OUT, OUTER, OUTPUT, OVER, OVERLAPS, OVERLAY, OVERRIDING, PAD, PARAMETER, PARAMETER_MODE, PARAMETER_NAME, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_NAME, PARAMETER_SPECIFIC_SCHEMA, PARTIAL, PARTITION, PASCAL, PASSTHROUGH, PATH, PERCENTILE_CONT, PERCENTILE_DISC, PERCENT_RANK, PLACING, PLAN, PLI, POSITION, POWER, PRECEDING, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES, PROCEDURE, PUBLIC, QUARTER, RANGE, RANK, RAW, READ, READS, REAL, RECURSIVE, REF, REFERENCES, REFERENCING, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, RELATIVE, RELEASE, REPEATABLE, RESET, RESTART, RESTRICT, RESULT, RETURN, RETURNED_CARDINALITY, RETURNED_LENGTH, RETURNED_OCTET_LENGTH, RETURNED_SQLSTATE, RETURNS, REVOKE, RIGHT, ROLE, ROLLBACK, ROLLUP, ROUTINE, ROUTINE_CATALOG, ROUTINE_NAME, ROUTINE_SCHEMA, ROW, ROWS, ROW_COUNT, ROW_NUMBER, SAVEPOINT, SCALE, SCHEMA, SCHEMA_NAME, SCOPE, SCOPE_CATALOGS, SCOPE_NAME, SCOPE_SCHEMA, SCROLL, SEARCH, SECOND, SECTION, SECURITY, SELECT, SELF, SENSITIVE, SEQUENCE, SERIALIZABLE, SERVER, SERVER_NAME, SESSION, SESSION_USER, SET, SETS, SIMILAR, SIMPLE, SIZE, SMALLINT, SOME, SOURCE, SPACE, SPECIFIC, SPECIFICTYPE, SPECIFIC_NAME, SQL, SQLEXCEPTION, SQLSTATE, SQLWARNING, SQL_TSI_DAY, SQL_TSI_FRAC_SECOND, SQL_TSI_HOUR, SQL_TSI_MICROSECOND, SQL_TSI_MINUTE, SQL_TSI_MONTH, SQL_TSI_QUARTER, SQL_TSI_SECOND, SQL_TSI_WEEK, SQL_TSI_YEAR, SQRT, START, STATE, STATEMENT, STATIC, STDDEV_POP, STDDEV_SAMP, STREAM, STRING, STRUCTURE, STYLE, SUBCLASS_ORIGIN, SUBMULTISET, SUBSTITUTE, SUBSTRING, SUM, SYMMETRIC, SYSTEM, SYSTEM_USER, TABLE, TABLESAMPLE, TABLE_NAME, TEMPORARY, THEN, TIES, TIME, TIMESTAMP, TIMESTAMPADD, TIMESTAMPDIFF, TIMEZONE_HOUR, TIMEZONE_MINUTE, TINYINT, TO, TOP_LEVEL_COUNT, TRAILING, TRANSACTION, TRANSACTIONS_ACTIVE, TRANSACTIONS_COMMITTED, TRANSACTIONS_ROLLED_BACK, TRANSFORM, TRANSFORMS, TRANSLATE, TRANSLATION, TREAT, TRIGGER, TRIGGER_CATALOG, TRIGGER_NAME, TRIGGER_SCHEMA, TRIM, TRUE, TYPE, UESCAPE, UNBOUNDED, UNCOMMITTED, UNDER, UNION, UNIQUE, UNKNOWN, UNNAMED, UNNEST, UPDATE, UPPER, UPSERT, USAGE, USER,

USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_CODE, USER_DEFINED_TYPE_NAME, USER_DEFINED_TYPE_SCHEMA, USING, VALUE, VALUES, VARBINARY, VARCHAR, VARYING, VAR_POP, VAR_SAMP, VERSION, VIEW, WEEK, WHEN, WHENEVER, WHERE, WIDTH_BUCKET, WINDOW, WITH, WITHIN, WITHOUT, WORK, WRAPPER, WRITE, XML, YEAR, ZONE

3.4.2 流 SQL UDF

说明

当前版本中，DLH 流 SQL 暂不支持自定义 UDF，DLH 流 SQL 的内置 UDF 兼容 Flink1.12 版本中的内置 UDF，Flink1.12 的内置 UDF 请参见：https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/functions/systemFunctions.html。

3.4.3 创建表

DLH 流表的建表语法完全兼容开源 FlinkSQL 的建表语法。但当前版本中，DLH 流表连接器的数据源端仅支持 Kafka，数据目的端仅支持 Kafka、Mysql、PostgreSQL、Elasticsearch、Hive。其中，Kafka 连接器支持的数据格式为 csv 或 json，Hive 连接器支持的数据格式为 orc、parquet 或 textfile，Elasticsearch 支持的数据格式为 json，Mysql 和 PostgreSQL 连接器不区分数据格式。

当前版本中，DLH 流表的数据源端支持事件时间类型表、处理时间类型表。

1. DLH 流表的建表语句

图3-3 DLH 流表的建表语句中需设置的基本属性值

属性	默认值	是否必选	说明
connector	无	是	连接器类型。在当前版本中，DLH仅支持kafka、elasticsearch-7、jdbc
topic	无	是	连接器所连接的Kafka topic
properties.group.id	无	是	消费Kafka数据时指定的groupID。作为source表时此属性必填，作为sink表时此属性可不填
scan.startup.mode	group-offsets	否	Kafka消费端offset配置。可选值：latest-offset、earliest-offset、group-offsets
properties.bootstrap.servers	无	是	连接Kafka必需的boostrap.server配置
format	无	是	数据流格式。在当前版本中，DLH仅支持csv、json
csv.field-delimiter	无	是	数据流字符串分隔符。csv格式下必填
properties.security.protocol	无	否	Kerberos环境下，连接Kafka所需的安全协议，值为：SASL_PLAINTEXT
properties.sasl.mechanism	无	否	Kerberos环境下，连接Kafka所需的认证机制，值为：GSSAPI
properties.sasl.kerberos.service.name	无	否	Kerberos环境下，连接Kafka所需的认证服务名称，值为：kafka

2. 示例

场景 1：在数据源端创建表

- 创建 CSV 表
  - Kafka(csv)连接器事件时间下的建表语句：

```
CREATE TABLE csv_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'csv_source',
  'properties.group.id' = 'flink1',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'csv',
  'csv.field-delimiter' = '|'
);
```

  - Kafka(csv)连接器处理时间下的建表语句：

    注意：ts 为额外的处理时间字段，数据源中如果存在会被覆盖

```
CREATE TABLE csv_source(
  user_id INT,
  product STRING,
  amount INT,
  ts as proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'csv_source',
  'properties.group.id' = 'flink1',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'csv',
  'csv.field-delimiter' = '|'
);
```

- 创建 json 表
  - Kafka(json)连接器事件时间下的建表语句：

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
```

```
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'json'
);
```

- Kafka(json)连接器处理时间下的建表语句：

  【注意】ts 为额外的处理时间字段，数据源中如果存在会被覆盖。

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
  amount INT,
  ts as proctime()
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'json'
);
```

场景 2：在数据目的端创建表

- Kafka(csv)连接器的建表语句：

```
CREATE TABLE csv_source(
  user_id INT,
  product STRING,
  amount INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'csv_source',
  'properties.group.id' = 'flink1',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'csv',
  'csv.field-delimiter' = '|'
);
```

- Kafka(json)连接器的建表语句：

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
  amount INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
```

```
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'json'
);
```

- Elasticsearch(json)连接器的建表语句：

```
CREATE TABLE sk_es (
  user_id INT,
  product STRING,
  amount INT,
  PRIMARY KEY (user_id) NOT ENFORCED
) WITH (
  'connector' = 'elasticsearch-7',
  'hosts' = 'http://qcj36:9200',
  'index' = 'users'
);
```

- Mysql 连接器的建表语句：

```
CREATE TABLE sk_jdbc_mysql (
  user_id INT,
  product STRING,
  amount INT
) WITH (
   'connector' = 'jdbc',
   'url' = 'jdbc:mysql://10.121.65.53:53306/mydatabase',
   'table-name' = 'users',
   'username' = 'cluster',
   'password' = 'CloudOS5#DE3@Cluster'
);
```

- PostgreSQL 连接器的建表语句：

```
CREATE TABLE sk_jdbc_postgresql (
  user_id INT,
  product STRING,
  amount INT
) WITH (
   'connector' = 'jdbc',
   'url' = 'jdbc:postgresql://10.121.47.114:5432/mydatabase',
   'table-name' = 'users',
   'username' = 'postgres',
   'password' = 'postgres'
);
```

- Hive(textfile)连接器的建表语句：

```
CREATE TABLE hive_txt(
  user_id INT,
  product STRING,
  amount INT
)PARTITIONED BY(dt STRING, hr STRING) stored as TEXTFILE TBLPROPERTIES (
  'sink.rolling-policy.file-size'='1MB',
  'sink.rolling-policy.rollover-interval'='1 min',
  'sink.rolling-policy.check-interval'='1 s',
```

```
  'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',
  'sink.partition-commit.trigger'='partition-time',
  'sink.partition-commit.delay'='1 s',
  'sink.partition-commit.policy.kind'='metastore,success-file'
);
```

- Hive(orc)连接器的建表语句：

```
CREATE TABLE hive_orc(
  user_id INT,
  product STRING,
  amount INT
)PARTITIONED BY(dt STRING, hr STRING) stored as orc TBLPROPERTIES (
  'sink.rolling-policy.file-size'='1MB',
  'sink.rolling-policy.rollover-interval'='1 min',
  'sink.rolling-policy.check-interval'='1 s',
  'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',
  'sink.partition-commit.trigger'='partition-time',
  'sink.partition-commit.delay'='1 s',
  'sink.partition-commit.policy.kind'='metastore,success-file'
);
```

- Hive(parquet)连接器的建表语句：

```
CREATE TABLE hive_parquet(
  user_id INT,
  product STRING,
  amount INT
)PARTITIONED BY(dt STRING, hr STRING) stored as parquet TBLPROPERTIES (
  'sink.rolling-policy.file-size'='1MB',
  'sink.rolling-policy.rollover-interval'='1 min',
  'sink.rolling-policy.check-interval'='1 s',
  'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',
  'sink.partition-commit.trigger'='partition-time',
  'sink.partition-commit.delay'='1 s',
  'sink.partition-commit.policy.kind'='metastore,success-file'
);
```

3.4.4 数据插入

1. 插入语句

DLH 流表的数据插入语句兼容 FlinkSQL 的 insert into……select……语句：

INSERT INTO tablename1(col1,col2,col3,…)SELECT col1,col2,col3,…From tablename2

【注意】

- 非 Kerberos 环境下，可直接执行 DLH 流表的数据插入语句。
- Kerberos 环境下，执行 DLH 流表的数据插入语句，需满足以下条件：
  - 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。
  - 确保在 Flink 的所有节点上，都存在上述用户对应的 keytab 文件，且需要保证该用户为该 keytab 文件的所有者。

- 修改 Flink 的配置：security.kerberos.login.keytab= <keytab path>；security.kerberos.login.principal= <user principal>，以上两个配置项的值修改完成并保存后，重启 Flink。

2. 窗口类型

DLH 支持 FlinkSQL 所有窗口类型：tumble、hop、session、over 等。

3. 示例

(1) 示例 1：非 Kerberos 环境下进行 DLH 流表的数据插入（CSV），步骤如下：

a. 创建 Kafka topic：csv_source 和 csv_sink

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic csv_source --create --partitions 3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic csv_sink --create --partitions 3 --replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE csv_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'csv_source',
  'properties.group.id' = 'flink_csv',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'csv',
  'csv.field-delimiter' = '|'
);
```

c. 创建数据输出表

```
CREATE TABLE csv_sink (
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3)
) WITH (
  'connector' = 'kafka',
  'topic' = 'csv_sink',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'csv'
);
```

d. 提交流任务

```
INSERT INTO csv_sink(user_id, product, amount, ts) SELECT user_id, product, amount, ts FROM csv_source;
```

e. 向 Kafka 的 topic：csv_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 csv_input.txt）。

表3-24 示例数据

```
1|beer|3|2019-12-12 00:00:01
1|diaper|4|2019-12-12 00:00:02
2|pen|3|2019-12-12 00:00:04
2|rubber|3|2019-12-12 00:00:06
3|rubber|2|2019-12-12 00:00:05
4|beer|1|2019-12-12 00:00:08
```

向 Kafka 的 topic：csv_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667  --topic csv_source < csv_input.txt
```

f. 消费 Kafka 的 topic：csv_sink 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic csv_sink --from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的 csv_source、csv_sink 表

```
drop table csv_source;
drop table csv_sink;
```

【说明】此处的删除仅表示删除存在 DLH 中 Hive 里面的元数据，并不会删除实际存储的真实数据（即 Kafka 中的 topic 数据依旧存在）。

(2) 示例 2：非 Kerberos 环境下进行 DLH 流表的数据插入（JSON），步骤如下：

a. 创建 Kafka topic：json_source 和 json_sink

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_source --create --partitions 3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_sink --create --partitions 3 --replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE json_source(
    user_id INT,
    product STRING,
    amount INT,
    ts TIMESTAMP(3),
    WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
    'connector' = 'kafka',
    'topic' = 'json_source',
    'properties.group.id' = 'flink_json',
    'scan.startup.mode' = 'latest-offset',
    'properties.bootstrap.servers' = '10.121.65.7:6667',
    'format' = 'json'
```

```
);
```

c. 创建数据输出表

```
CREATE TABLE json_sink (user_id INT,product STRING,amount INT,ts TIMESTAMP(3)) WITH
(
  'connector' = 'kafka',
  'topic' = 'json_sink',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'format' = 'json'
);
```

d. 提交流任务

```
INSERT INTO json_sink(user_id, product, amount, ts) SELECT user_id, product, amount,
ts FROM json_source;
```

e. 向 Kafka 的 topic：json_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 json_input.txt）。

表3-25 示例数据

```
{"user_id":1, "product":"beer","amount":3, "ts":"2019-12-12 00:00:01"}
{"user_id":1, "product":"diaper","amount":4, "ts":"2019-12-12 00:00:02"}
{"user_id":2, "product":"pen","amount":200, "ts":"2019-12-12 00:00:03"}
{"user_id":2, "product":"rubber","amount":30, "ts":"2019-12-12 00:00:04"}
{"user_id":3, "product":"rubber","amount":3000, "ts":"2019-12-12 00:00:05"}
{"user_id":4, "product":"beer","amount":2222, "ts":"2019-12-12 00:00:09"}
```

向 Kafka 的 topic：json_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667  --topic json_source <
json_input.txt
```

f. 消费 Kafka 的 topic：json_sink 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic json_sink
--from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的 json_source、json_sink 表

```
drop table json_source;
drop table json_sink;
```

【说明】此处的删除仅表示删除存在 DLH 中 Hive 里面的元数据，并不会删除实际存储的真实数据（即 Kafka 中的 topic 数据依旧存在）。

(3) 示例 3：Kerberos 环境下进行 DLH 流表的数据插入（CSV），步骤如下：

a. 创建 Kafka topic：csv_source 和 csv_sink

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic csv_source --create --partitions
3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic csv_sink --create --partitions 3
--replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE csv_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'csv_source',
  'properties.group.id' = 'flink_csv',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'csv',
  'csv.field-delimiter' = '|'
);
```

c. 创建数据输出表

```
CREATE TABLE csv_sink (user_id INT,product STRING,amount INT,ts TIMESTAMP(3)) WITH
(
  'connector' = 'kafka',
  'topic' = 'csv_sink',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'csv'
);
```

d. 提交流任务

前置条件：

- 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。
- 确保在 Flink 的所有节点上，都存在上述用户对应的 keytab 文件（示例 /opt/admin123.keytab），且需要保证该用户为该 keytab 文件的所有者。
- 修改 Flink 的配置：security.kerberos.login.keytab= /opt/admin123.keytab；security.kerberos.login.principal= admin123@FLINKKER1.COM，以上两个配置项的值修改完成并保存后，重启 Flink。

以上前置条件全部满足之后，才可以提交流任务 SQL：

```
INSERT INTO csv_sink(user_id, product, amount, ts) SELECT user_id, product, amount,
ts FROM csv_source;
```

e. 向 Kafka 的 topic：csv_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 csv_input.txt）。

表3-26 示例数据

```
1|beer|3|2019-12-12 00:00:01
1|diaper|4|2019-12-12 00:00:02
2|pen|3|2019-12-12 00:00:04
2|rubber|3|2019-12-12 00:00:06
3|rubber|2|2019-12-12 00:00:05
4|beer|1|2019-12-12 00:00:08
```

向 Kafka 的 topic：csv_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667 --producer-property
security.protocol=SASL_PLAINTEXT --topic csv_source < csv_input.txt
```

f. 消费 Kafka 的 topic：csv_sink 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic csv_sink
--consumer-property security.protocol=SASL_PLAINTEXT --from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的 csv_source、csv_sink 表

```
drop table csv_source;
drop table csv_sink;
```

【说明】此处的删除仅表示删除存在 DLH 中 Hive 里面的元数据，并不会删除实际存储的真实数据（即 Kafka 中的 topic 数据依旧存在）。

(4) 示例 4：Kerberos 环境下进行 DLH 流表的数据插入（JSON），步骤如下：

a. 创建 Kafka topic：json_source 和 json_sink

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_source --create --partitions
3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_sink --create --partitions 3
--replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

c. 创建数据输出表

```
CREATE TABLE json_sink (user_id INT,product STRING,amount INT,ts TIMESTAMP(3)) WITH
(
  'connector' = 'kafka',
  'topic' = 'json_sink',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

d. 提交流任务

前置条件：

– 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。
– 确保在 Flink 的所有节点上，都存在上述用户对应的 keytab 文件（示例 /opt/admin123.keytab），且需要保证该用户为该 keytab 文件的所有者。
– 修改 Flink 的配置：security.kerberos.login.keytab= /opt/admin123.keytab；security.kerberos.login.principal= admin123@FLINKKER1.COM，以上两个配置项的值修改完成并保存后，重启 Flink。

以上前置条件全部满足之后，才可以提交流任务 SQL：

```
INSERT INTO json_sink(user_id, product, amount, ts) SELECT user_id, product, amount,
ts FROM json_source;
```

e. 向 Kafka 的 topic：json_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 json_input.txt）。

表3-27 示例数据

```
{"user_id":1, "product":"beer","amount":3, "ts":"2019-12-12 00:00:01"}
{"user_id":1, "product":"diaper","amount":4, "ts":"2019-12-12 00:00:02"}
        {"user_id":2, "product":"pen","amount":200, "ts":"2019-12-12 00:00:03"}
{"user_id":2, "product":"rubber","amount":30, "ts":"2019-12-12 00:00:04"}
{"user_id":3, "product":"rubber","amount":3000, "ts":"2019-12-12 00:00:05"}
{"user_id":4, "product":"beer","amount":2222, "ts":"2019-12-12 00:00:09"}
```

向 Kafka 的 topic：json_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667  --topic json_source
--producer-property security.protocol=SASL_PLAINTEXT < json_input.txt
```

f. 消费 Kafka 的 topic：json_sink 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic json_sink
--consumer-property security.protocol=SASL_PLAINTEXT --from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的 json_source、json_sink 表

```
drop table json_source;
drop table json_sink;
```

【说明】此处的删除仅表示删除存在 DLH 中 Hive 里面的元数据，并不会删除实际存储的真实数据（即 Kafka 中的 topic 数据依旧存在）。

(5) 示例 5：Kerberos 环境下进行 DLH 流表的数据插入（tumble 窗口），步骤如下：

a. 创建 Kafka topic：json_source 和 json_sink_tumble

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_source --create --partitions 3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_sink_tumble --create --partitions 3 --replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

c. 创建数据输出表

```
CREATE TABLE json_sink_tumble (
  cnt BIGINT,
  ts TIMESTAMP(3)
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_sink_tumble',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

d. 提交流任务

前置条件：

– 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。

– 确保在 Flink 的所有节点上，都存在上述用户对应的 keytab 文件（示例 /opt/admin123.keytab），且需要保证该用户为该 keytab 文件的所有者。
– 修改 Flink 的配置：security.kerberos.login.keytab= /opt/admin123.keytab；security.kerberos.login.principal= admin123@FLINKKER1.COM，以上两个配置项的值修改完成并保存后，重启 Flink。

以上前置条件全部满足之后，才可以提交流任务 SQL：

```
INSERT INTO json_sink_tumble SELECT  count(product), tumble_start(ts, interval '1'
minute) FROM json_source group by tumble(ts, interval '1' minute);
```

e. 向 Kafka 的 topic：json_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 json_input.txt）。

表3-28 示例数据

{"user_id":1, "product":"beer","amount":3, "ts":"2019-12-12 00:00:01"} {"user_id":1, "product":"diaper","amount":4, "ts":"2019-12-12 00:00:02"} {"user_id":2, "product":"pen","amount":200, "ts":"2019-12-12 00:00:03"} {"user_id":2, "product":"rubber","amount":30, "ts":"2019-12-12 00:00:04"} {"user_id":3, "product":"rubber","amount":3000, "ts":"2019-12-12 00:00:05"} {"user_id":4, "product":"beer","amount":2222, "ts":"2019-12-12 00:00:09"}

向 Kafka 的 topic：json_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667  --topic json_source
--producer-property security.protocol=SASL_PLAINTEXT < json_input.txt
```

f. 消费 Kafka 的 topic：json_sink_tumble 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic json_sink_tumble
--consumer-property security.protocol=SASL_PLAINTEXT --from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的 json_source、json_sink_tumble 表

```
drop table json_source;
drop table json_sink_tumble;
```

【说明】此处的删除仅表示删除存在 DLH 中 Hive 里面的元数据，并不会删除实际存储的真实数据（即 Kafka 中的 topic 数据依旧存在）。

(6) 示例 6：Kerberos 环境下进行 DLH 流表的数据插入（hop 窗口），步骤如下：

a. 创建 Kafka topic：json_source 和 json_sink_hop

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_source --create --partitions
3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_sink_hop --create
--partitions 3 --replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
```

```
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

c. 创建数据输出表

```
CREATE TABLE json_sink_hop (
  win_start TIMESTAMP(3),
  win_end TIMESTAMP(3),
  win_rowtime TIMESTAMP(3),
  user_id INT,
  product_cnt bigint
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_sink_hop',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

d. 提交流任务

前置条件：

- 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。
- 确保在 Flink 的所有节点上，都存在上述用户对应的 keytab 文件（示例/opt/admin123.keytab），且需要保证该用户为该 keytab 文件的所有者。
- 修改 Flink 的配置：security.kerberos.login.keytab= /opt/admin123.keytab；security.kerberos.login.principal= admin123@FLINKKER1.COM，以上两个配置项的值修改完成并保存后，重启 Flink。

以上前置条件全部满足之后，才可以提交流任务 SQL：

```
INSERT INTO json_sink_hop SELECT hop_start(ts, interval '2' second, interval '5'
second),hop_end(ts, interval '2' second, interval '5' second), hop_rowtime(ts,
interval '2' second, interval '5' second),user_id,count(product) FROM json_source
group by hop(ts, interval '2' second, interval '5' second),user_id;
```

e. 向 Kafka 的 topic：json_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 json_input.txt）。

表3-29 示例数据

```
{"user_id":1, "product":"beer","amount":3, "ts":"2019-12-12 00:00:01"}
{"user_id":1, "product":"diaper","amount":4, "ts":"2019-12-12 00:00:02"}
{"user_id":2, "product":"pen","amount":200, "ts":"2019-12-12 00:00:03"}
{"user_id":2, "product":"rubber","amount":30, "ts":"2019-12-12 00:00:04"}
{"user_id":3, "product":"rubber","amount":3000, "ts":"2019-12-12 00:00:05"}
{"user_id":4, "product":"beer","amount":2222, "ts":"2019-12-12 00:00:09"}
```

向 Kafka 的 topic：json_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667  --topic json_source
--producer-property security.protocol=SASL_PLAINTEXT < json_input.txt
```

f. 消费 Kafka 的 topic：json_sink_hop 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic json_sink_hop
--consumer-property security.protocol=SASL_PLAINTEXT --from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的 json_source、json_sink_hop 表

```
drop table json_source;
drop table json_sink_hop;
```

【说明】此处的删除仅表示删除存在 DLH 中 Hive 里面的元数据，并不会删除实际存储的真实数据（即 Kafka 中的 topic 数据依旧存在）。

(7) 示例 7：Kerberos 环境下进行 DLH 流表的数据插入（session 窗口），步骤如下：

a. 创建 Kafka topic：json_source 和 json_sink_session

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_source --create --partitions
3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_sink_session --create
--partitions 3 --replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
```

```
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

c. 创建数据输出表

```
CREATE TABLE json_sink_session (
  session_start TIMESTAMP(3),
  session_end TIMESTAMP(3),
  session_rowtime TIMESTAMP(3),
  user_id INT,
  product_cnt bigint
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_sink_session',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

d. 提交流任务

前置条件：

- 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。
- 确保在 Flink 的所有节点上，都存在上述用户对应的 keytab 文件（示例 /opt/admin123.keytab），且需要保证该用户为该 keytab 文件的所有者。
- 修改 Flink 的配置：security.kerberos.login.keytab= /opt/admin123.keytab；security.kerberos.login.principal= admin123@FLINKKER1.COM，以上两个配置项的值修改完成并保存后，重启 Flink。

以上前置条件全部满足之后，才可以提交流任务 SQL：

```
INSERT INTO json_sink_session SELECT session_start(ts, interval '5'
second),session_end(ts, interval '5' second), session_rowtime(ts, interval '5'
second),user_id,count(product) FROM json_source group by session(ts, interval '5'
second),user_id;
```

e. 向 Kafka 的 topic json_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 json_input.txt）。

表3-30 示例数据

{"user_id":1, "product":"beer","amount":3, "ts":"2019-12-12 00:00:01"} {"user_id":1, "product":"diaper","amount":4, "ts":"2019-12-12 00:00:02"} {"user_id":2, "product":"pen","amount":200, "ts":"2019-12-12 00:00:03"} {"user_id":2, "product":"rubber","amount":30, "ts":"2019-12-12 00:00:04"} {"user_id":3, "product":"rubber","amount":3000, "ts":"2019-12-12 00:00:05"} {"user_id":4, "product":"beer","amount":2222, "ts":"2019-12-12 00:00:09"}

向 Kafka 的 topic json_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667  --topic json_source
--producer-property security.protocol=SASL_PLAINTEXT < json_input.txt
```

f. 消费 Kafka 的 topic：json_sink_session 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic
json_sink_session --consumer-property security.protocol=SASL_PLAINTEXT
--from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的 json_source、json_sink_session 表

```
drop table json_source;
drop table json_sink_session;
```

【说明】此处的删除仅表示删除存在 DLH 中 Hive 里面的元数据，并不会删除实际存储的真实数据（即 Kafka 中的 topic 数据依旧存在）。

(8) 示例 8：Kerberos 环境下进行 DLH 流表的数据插入（over 窗口），步骤如下：

a. 创建 Kafka topic：json_source 和 json_sink_over

```
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_source --create --partitions
3 --replication-factor 2
./kafka-topics.sh --zookeeper zyf07:2181 --topic json_sink_over --create
--partitions 3 --replication-factor 2
```

b. 连接 DLH beeline 并创建数据源表

```
CREATE TABLE json_source(
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_source',
  'properties.group.id' = 'flink_json',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

c. 创建数据输出表

```
CREATE TABLE json_sink_over (
  user_id INT,
  product STRING,
  amount INT,
  maxAmount INT
```

```
) WITH (
  'connector' = 'kafka',
  'topic' = 'json_sink_over',
  'properties.group.id' = 'flink2',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '10.121.65.7:6667',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'format' = 'json'
);
```

d. 提交流任务

前置条件：

– 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。

– 确保在 Flink 的所有节点上，都存在上述用户对应的 keytab 文件（示例/opt/admin123.keytab），且需要保证该用户为该 keytab 文件的所有者。

– 修改 Flink 的配置：security.kerberos.login.keytab= /opt/admin123.keytab；security.kerberos.login.principal= admin123@FLINKKER1.COM，以上两个配置项的值修改完成并保存后，重启 Flink。

以上前置条件全部满足之后，才可以提交流任务 SQL：

```
insert into json_sink_over select user_id,product,amount,max(amount) over(partition by product order by ts rows between 2 preceding and current row) as maxAmount from json_source;
```

e. 向 Kafka 的 topic：json_source 生产数据

【注意】写入 topic 的数据字符串格式与数据源表的表结构务必一致，否则会造成数据丢失或任务失败。

可以提前准备数据，然后将数据写入到一个文件（示例 json_input.txt）。

表3-31 示例数据

{"user_id":1, "product":"beer","amount":3, "ts":"2019-12-12 00:00:01"} {"user_id":1, "product":"diaper","amount":4, "ts":"2019-12-12 00:00:02"} {"user_id":2, "product":"pen","amount":200, "ts":"2019-12-12 00:00:03"} {"user_id":2, "product":"rubber","amount":30, "ts":"2019-12-12 00:00:04"} {"user_id":3, "product":"rubber","amount":3000, "ts":"2019-12-12 00:00:05"} {"user_id":4, "product":"beer","amount":2222, "ts":"2019-12-12 00:00:09"}

向 Kafka 的 topic：json_source 生产数据，命令如下：

```
./kafka-console-producer.sh --broker-list zyf07:6667  --topic json_source
--producer-property security.protocol=SASL_PLAINTEXT < json_input.txt
```

f. 消费 Kafka 的 topic：json_sink_over 数据

```
./kafka-console-consumer.sh --bootstrap-server zyf07:6667 --topic json_sink_over
--consumer-property security.protocol=SASL_PLAINTEXT --from-beginning
```

g. 测试结束，停止流任务

```
yarn application -list  获取对应的 applicationId
yarn application --kill  <applicationId>
```

h. 删除已创建的json_source、json_sink_over表

```
drop table json_source;
drop table json_sink_over;
```

【说明】此处的删除仅表示删除存在DLH中Hive里面的元数据，并不会删除实际存储的真实数据（即Kafka中的topic数据依旧存在）。

3.4.5 数据查询

说明

- 流计算任务为长驻任务。在查询SQL执行过程中，如果出现Ctrl+C或其他操作导致查询异常中止，Flink查询任务也无法自动停止。在此情况下，若需要终止Flink查询任务，则需要根据查询SQL的开始执行时间在YARN上查询对应的Flink任务并手动终止。
- 在当前版本中，DLH会话中窗口查询结果不在控制台直接打印，实际计算结果需在对应的Flink作业监控页面进行查询。实际业务场景下，多数是通过insert into <目的表> select from <源表>语句来执行窗口操作。

1. 查询语法

DLH流表的数据查询语句兼容FlinkSQL的streaming模式下的查询语句。

- 非Kerberos环境下，可直接执行DLH流表的数据查询语句。
- Kerberos环境下，执行DLH流表的数据查询语句，需满足以下条件：
    - 准备一个同时具备输入表的读权限和输出表的写权限的用户（为方便起见，也可以直接使用集群超级用户）。
    - 确保在Flink的所有节点上，都存在上述用户对应的keytab文件，且需要保证该用户为该keytab文件的所有者。
    - 修改Flink的配置：security.kerberos.login.keytab= <keytab path>；security.kerberos.login.principal= <user principal>，以上两个配置项的值修改完成并保存后，重启Flink。

2. 示例

(1) 创建Kafka topic：test001，用于数据源产生（若开启kerberos则需追加--command-config kerberos参数）

```
./kafka-topics.sh --create --bootstrap-server rzt237:6667,rzt238:6667,rzt239:6667
--replication-factor 1 --partitions 1 --topic test001 [--command-config kerberos]
```

(2) 创建流表flink55

【说明】DLH流表的建表语句中需设置的基本属性值请参考3.4.3 章节。

```
CREATE TABLE flink55 (
  user_id INT,
  product STRING,
  amount INT,
  ts TIMESTAMP(3),
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND
) WITH (
```

```
  'connector' = 'kafka',
  'topic' = 'test001',
  'properties.group.id' = 'flink22',
  'scan.startup.mode' = 'earliest-offset',
  'properties.bootstrap.servers' = '10.121.64.238:6667',
  'format' = 'csv',
  'csv.field-delimiter' = '|'
);
```

(3) 向 Kafka 的 topic：test001 中生产数据

【注意】数据格式务必与建表语句中 format.type 和 formmat.field-delimiter 属性保持一致。

表3-32 测试数据

```
1|diaper|4|2019-12-12 00:00:02
2|pen|3|2019-12-12 00:00:04
2|rubber|3|2019-12-12 00:00:06
3|rubber|2|2019-12-12 00:00:05
4|beer|1|2019-12-12 00:00:08
1|diaper|4|2019-12-12 00:00:02
2|pen|3|2019-12-12 00:00:04
2|rubber|3|2019-12-12 00:00:06
3|rubber|2|2019-12-12 00:00:05
```

(4) 进行流表数据查询

- select 语句

  执行 SQL：

  select * from flink55;

- select as 语句

  执行 SQL：

  select user_id,amout as ddd from flink55;

  【注意】自定义字段名需避免使用保留关键字。

- 过滤查询 select * from [tab] where [filter_condition]语句

  执行 SQL：

  select * from flink55 where user_id > 2;

- 查询嵌套常用 udf

  执行 SQL：

  select UPPER(product) from flink55;

- 分组聚合

  执行 SQL：

  select user_id,sum(amount) as cnt from flink55 group by user_id

- 去重语法

  执行 SQL：

  select DISTINCT user_id from flink55;

- having 语法

  执行 SQL：

select sum(amount) from flink55 group by user_id having sum(amount) > 5;

- join 语法

  前置条件：

  – 创建 kafka topic：test002，并向 test002 中生产部分模拟数据。

  – 创建新的 Flink 表（表中字段与 topic test002 中字段格式需保持一致）。

  以上前置条件全部满足之后，执行 SQL：

  select flink55.user_id,flink55.product from flink55 inner join flink55_2 on flink55.user_id = flink55_2.user_id;

  【注意】此情况下，时间字段不能作为查询过滤条件及查询返回结果字段。

- 分组窗口聚合

  在当前版本中，DLH 会话中窗口查询结果不在控制台直接打印，实际计算结果需在对应的 Flink 作业监控页面进行查询。实际业务场景下，多数是通过 insert into <目的表> select from <源表>语句来执行窗口操作。

3.5 MLSQL

MLSQL 是基于 MLlib 的 SQL 中间件。MLlib 是 Spark 的机器学习库，目标是让机器学习更加容易且更具扩展性，使用 Mllib 时需要编写 MLlib 应用，操作复杂。但是通过 MLSQL，用户可直接通过 SQL 语句使用 MLlib 中的算法，使操作的便利性得到了很大提高。

MLSQL 目前支持监督算法（比如：分类算法、回归算法）、非监督算法（比如：聚类算法）、特征工程算法、模型选择、Pipeline 机制。

3.5.1 监督算法

1. 语法

- 训练语法

  SELECT

  TrainAlgorithmName('model_result_table',label,col1,col2,...,['parms'])

  from TrainTableName;

- 预测语法

  SELECT

  PredictAlgorithmName('model_result_table',col1,col2,col3,...,'id','predict_result_table')

  from PredictTableName;

2. 语法说明

表3-33 训练语法说明

参数	说明	常见配置选项
TrainAlgorithmName	训练算法名称，必须指定，支持的训练算法见算法说明：训练算法部分	无
model_result_table	算法训练结果模型表，必须指定	无

参数	说明	常见配置选项
TrainTableName	训练数据表名，用作算法的数据源，必须指定	无
label	训练数据表中的列，用作算法的参数，作为标签列，必须指定	无
col1,col2,…	训练数据表中的列，用作算法的参数，作为特征列，必须指定	无
parms	算法的其它参数，根据算法需要，可以不指定(内容格式：-ParamKey1 ParamValue1 -ParamKey2 ParamValue2 …)，参数说明	无

表3-34 预测语法说明

参数名称	参数解释	备注
PredictAlgorithmName	预测算法名称，必须指定，算法说明：预测算法部分	无
model_result_table	模型表名，必须指定	用来指定使用哪个模型进行预测，对应训练中的model_result_table
PredictTableName	预测数据表名，用来预测的数据，必须指定	无
id	预测数据表的id列，用作算法的参数，作为标签列，必须指定	无
col1,col2,…	表的列，用作算法的参数，必须指定	无
predict_result_table	使用模型进行预测后的结果表	临时表

3. 算法说明

表3-35 训练算法

算法名	算法说明	备注
LogisticRegression	逻辑回归训练函数	分类
DecisionTreeClassifier	决策树分类训练函数 [注意]：参与训练的数据属性列的值不能相同	分类
GBTClassifier	GBT分类训练函数 [注意]：参与训练的数据属性列的值不能相同	分类(二分类)
RandomForestClassifier	随机森林分类训练函数 [注意]：参与训练的数据属性列的值不能相同	分类
MultilayerPerceptronClassifier	多层感知机训练函数	分类
NaiveBayes	朴素贝叶斯训练函数	分类
OneVsRest	Onevsrest训练函数:一般用于多分类，需要设置基础二分类分类器	分类
LinearRegression	线性回归训练函数	回归

算法名	算法说明	备注
GeneralizedLinearRegression	广义线性回归训练函数	回归
DecisionTreeRegressor	决策树回归训练函数 [注意]：参与训练的数据属性列的值不能相同	回归
RandomForestRegressor	随机森林回归训练函数 [注意]：参与训练的数据属性列的值不能相同	回归
GBTRegressor	GBT回归训练函数 [注意]：参与训练的数据属性列的值不能相同	回归
IsotonicRegression	保序回归训练函数	回归
AFTSurvivalRegression	AFT-Survival回归训练函数	回归

表3-36 预测算法

算法名	算法说明	备注
LogisticRegressionPrediction	逻辑回归预测函数	分类预测
DecisionTreeClassificationPrediction	决策树分类预测函数	分类预测
GBTClassificationPrediction	GBT分类预测函数	分类预测
RandomForestClassificationPrediction	随机森林分类预测函数	分类预测
MultilayerPerceptronPrediction	多层感知机分类预测函数	分类预测
NaiveBayesPrediction	朴素贝叶斯分类预测函数	分类预测
OneVsRestPrediction	Onevsrest分类预测函数	分类预测
LinearRegressionPrediction	线性回归预测函数	回归预测
GeneralizedLinearRegressionPrediction	广义线性回归预测函数	回归预测
DecisionTreeRegressionPrediction	决策树回顾预测函数	回归预测
GBTRegressorPrediction	GBT回归预测函数	回归预测
RandomForestRegressorPrediction	随机森林预测函数	回归预测
IsotonicRegressionPrediction	保序回归预测函数	回归预测
AFTSurvivalRegressionPrediction	Aft-Survival回归预测函数	回归预测

4. 参数说明

表3-37 参数说明

参数名	参数说明	支持算法
aggregationdepth	设置分布式统计时的层数，主要用在treeAggregate中，数据量越大，可适当加大这个值，默认是2，int型	logisticregression、aftsurvivalregression、linearregression
standardization	训练模型前是否需要对训练特征进行标准化处理，	Logisticregression、

参数名	参数说明	支持算法
	boolean型	linearregression
threshold	二分类预测的阈值，范围[0,1]，double型	Logisticregression
thresholds	多分类预测的阈值，array[double]型	Logisticregression
fitintercept	是否训练intercept，boolean型	Logisticregression、aftsurvivalregression、linearregression、generalizedlinearregression
tol	优化算法迭代求解过程中的收敛阈值，默认值：1e-4，不能为负数，double型	Logisticregression、multilayerperceptronclassifier、aftsurvivalregression、generalizedlinearregression、linearregression
weightcol	列权重，string型	Logisticregression、naivebayes、isotonicregression、generalizedlinearregression、linearregression
regparam	正则化参数(>=0)，调节拟合程度，越小-过拟合，越大-欠拟合；double型	Logisticregression、generalizedlinearregression、linearregression
elasticnetparam	弹性网络混合参数(范围[0,1])，用于调节L1和L2之间的比例，两种正则化比例加起来是1，默认是0，即只使用L2正则化，设置为1，即只使用L1正则化；double型	Logisticregression、linearregression
family	模型中使用的误差分布类型 • Logisticregression 可选：auto,binomial,multinomial，不写默认使用 auto • GeneralizedLinearregression 可选：gaussian,binomial,poisson,gamma	Logisticregression、generalizedlinearregression
seed	随机种子，long型	Decisiontreeclassifier、gbtclassifier、randomforestclassifier、multilayerperceptronclassifier、decisiontreeregressor、gbtregressor、randomforestregressor
maxdepth	树的最大高度，int型	Decisiontreeclassifier、gbtclassifier、randomforestclassifier、decisiontreeregressor、gbtregressor、randomforestregressor
maxbins	每个特征分裂时，最大划分(桶)数量；int型	Decisiontreeclassifier、gbtclassifier、randomforestclassifier、decisiontreeregressor、gbtregressor、randomforestregressor

参数名	参数说明	支持算法
mininstancespernode	需保证节点分割出的子节点的最少样本数达到这个值，int型	Decisiontreeclassifier、gbtclassifier、randomforestclassifier、decisiontreeregressor、gbtregressor、randomforestregressor
mininfogain	最小信息增益，当前节点的所有属性分割带来的信息增益都要比这个值大，double型	Decisiontreeclassifier、gbtclassifier、randomforestclassifier、decisiontreeregressor、gbtregressor、randomforestregressor
subsamplingrate	二次抽样率，指定每棵树训练的数据比例，double型	gbtclassifier、randomforestclassifier、gbtregressor、randomforestregressor
maxiter	优化算法求解的最大迭代次数，默认值100，int型	LinearRegression、 Logisticregression、Gbtclassifier、multilayerperceptronclassifier、 AFTSurvivalRegression、GBTRegressor、GeneralizedLinearRegression
stepsize	每次迭代优化步长，double型，取值范围:(0,1]	Gbtclassifier、multilayerperceptronclassifier、gbtregressor
losstype	使用哪种损失函数 可选用：hinge、logistic、squared	Gbtclassifier、gbtregressor
blocksize	该参数被前馈网络训练器用来将训练样本数据的每个分区都按照blocksize大小分成不同组，并且每个组内的每个样本都会叠加成一个向量，以便在各种优化算法间传递；int型	multilayerperceptronclassifier
layers	**必须设置**，注意输入/输出的层数	multilayerperceptronclassifier
solver	优化的求解算法 • Multilayerperceptronclassifier 可选：l-bfgs,gd • Linearregression 可选：auto,l-bfgs,normal • Generalizedlinearregression 可选：irls	Multilayerperceptronclassifier、generalizedlinearregression、linearregression
modeltype	模型类型(Multinomial,Bernoulli)，实际上依据特征的分布不同，朴素贝叶斯又划分为多个子类别，string型	naivebayes
smoothing	一般采用拉普拉斯平滑进行处理，double型	naivebayes
classifier	设置基础二分类分类器，在onevsrest算法中必须要指定的	onevsrest
featuresubsetstrategy	特征子集策略，每棵树中使用那些特征做分裂，这个参数可以用小数比例形式指定，减少这个值会加速训	Randomforestclassifier、randomforestregressor

参数名	参数说明	支持算法
	练，但太小会影响效果	
numtrees	随机森林中决策树个数	Randomforestclassifier、randomforestregressor
censorcol	设置中心列	aftsurvivalregression
isotonic	递增保序回归或递减保序回归，boolean型	isotonicregression
link	连接函数名，描述线性预测器和分布函数均值之间关系 • family 为 gaussian 时可选：identity、log、inverse • family 为 binomial 时可选：logit、probit、cloglog • family 为 poisson 时可选：log、identity、sqrt • family 为 gamma 时可选：inverse、identity、log	generalizedlinearregression
linkpredictioncol	连接函数（线性预测器列名）	generalizedlinearregression

5. 示例

- LogisticRegression 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,0.0,2.3,4.5,5.6);
Insert into mltable values(2,1.0,6.5,3.8,4.9);
```

  运行语句示例：

```
------------------训练-----------------------
drop table if exists lor_model;
select LogisticRegression('lor_model',label,col1,col2,col3,'-MaxIter 20') from
mltable;
select * from lor_model;
--------------------预测---------------------
select LogisticRegressionPrediction('lor_model',col1,col2,col3,'id','lor_pred') from
mltable;
select * from lor_pred a,mltable b where a.id = b.id;
```

- DecisionTreeClassifier 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,0.0,5.7,5.9,7.9);
Insert into mltable values(2,1.0,7.8,2.6,12.1);
```

  语句示例：

```
------------------训练-----------------------
drop table if exists dtc_model;
select DecisionTreeClassifier('dtc_model',label,col1,col2,col3) from mltable;
select * from dtc_model;
--------------------预测---------------------
```

```
select
DecisionTreeClassificationPrediction('dtc_model',col1,col2,col3,'id','dtc_pred')
from mltable;
select * from dtc_pred a,mltable b where a.id = b.id;
```

- GBTClassifie 算法

数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,0.0,5.7,5.9,7.9);
Insert into mltable values(2,1.0,7.8,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists gbtc_model;
select GBTClassifier('gbtc_model',label,col1,col2,col3) from mltable;
select * from gbtc_model;
--------------------预测---------------------
select GBTClassificationPrediction('gbtc_model',col1,col2,col3,'id', 'gbtc_pred')
from mltable;
select * from gbtc_pred a,mltable b where a.id = b.id;
```

- RandomForestClassifier 算法

数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,0.0,5.7,5.9,7.9);
Insert into mltable values(2,1.0,7.8,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists rfc_model;
select RandomForestClassifier('rfc_model',label,col1,col2,col3) from mltable;
select * from rfc_model;
--------------------预测---------------------
select RandomForestClassificationPrediction('rfc_model',col1,col2,col3,'id',
'rfc_pred') from mltable;
select * from rfc_pred a,mltable b where a.id = b.id;
```

- MultilayerPerceptronClassifier 算法

数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,0.0,5.7,5.9,7.9);
Insert into mltable values(2,1.0,7.8,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists mpc_model;
select MultilayerPerceptronClassifier('mpc_model',label,col1,col2,col3, '-Layers 3 3
2') from mltable;
select * from mpc_model;
--------------------预测---------------------
select MultilayerPerceptronPrediction('mpc_model',col1,col2,col3,'id', 'mpc_pred')
from mltable;
select * from mpc_pred a,mltable b where a.id = b.id;
```

- NaiveBayes 算法

数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,0.0,5.7,5.9,7.9);
Insert into mltable values(2,1.0,7.8,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists nbc_model;
select NaiveBayes('nbc_model',label,col1,col2,col3) from mltable;
select * from nbc_model;
--------------------预测---------------------
select NaiveBayesPrediction('nbc_model',col1,col2,col3,'id', 'nbc_pred') from mltable;
select * from nbc_pred a,mltable b where a.id = b.id;
```

- OneVsRest 算法

数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,0.0,5.7,5.9,7.9);
Insert into mltable values(2,1.0,7.8,2.6,12.1);
```

语句示例：

```
---------------------训练-----------------------
drop table if exists ovrc_model;
select OneVsRest('ovrc_model',label,col1,col2,col3,'-Classifier NaiveBayes') from
mltable;
select * from ovrc_model;
--------------------预测------------------------
select OneVsRestPrediction('ovrc_model',col1,col2,col3,'id', 'ovrc_pred') from
mltable;
select * from ovrc_pred a,mltable b where a.id = b.id;
```

- LinearRegression 算法

数据准备：

```
Create table mltable2 (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable2 values(1,8.6,5.7,5.9,7.9);
Insert into mltable2 values(2,7.8,7.8,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists lr_model;
select LinearRegression('lr_model',label,col1,col2,col3) from mltable2;
select * from lr_model;
--------------------预测---------------------
select LinearRegressionPrediction('lr_model',col1,col2,col3,'id', 'lr_pred') from
mltable2;
select * from lr_pred a,mltable2 b where a.id = b.id;
```

- GeneralizedLinearRegression 算法

数据准备：

```
Create table mltable2 (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable2 values(1,8.6,5.7,5.9,7.9);
Insert into mltable2 values(2,7.8,7.8,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists glr_model;
select GeneralizedLinearRegression('glr_model',label,col1,col2,col3) from mltable2;
select * from glr_model;
--------------------预测---------------------
select GeneralizedLinearRegressionPrediction('glr_model',col1,col2,col3,'id',
'glr_pred') from mltable2;
select * from glr_pred a,mltable2 b where a.id = b.id;
```

- DecisionTreeRegression 算法

  数据准备：

```
Create table mltable2 (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable2 values(1,8.6,5.7,5.9,7.9);
Insert into mltable2 values(2,7.8,7.8,2.6,12.1);
```

  语句示例：

```
------------------训练-----------------------
drop table if exists dtr_model;
select DecisionTreeRegressor('dtr_model',label,col1,col2,col3) from mltable2;
select * from dtr_model;
-------------------预测---------------------
select DecisionTreeRegressionPrediction('dtr_model',col1,col2,col3,'id', 'dtr_pred')
from mltable2;
select * from dtr_pred a,mltable2 b where a.id = b.id;
```

- RandomForestRegression 算法

  数据准备：

```
Create table mltable2 (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable2 values(1,8.6,5.7,5.9,7.9);
Insert into mltable2 values(2,7.8,7.8,2.6,12.1);
```

  语句示例：

```
------------------训练-----------------------
drop table if exists rfr_model;
select RandomForestRegressor('rfr_model',label,col1,col2,col3) from mltable2;
select * from rfr_model;
-------------------预测---------------------
select RandomForestRegressorPrediction('rfr_model',col1,col2,col3,'id', 'rfr_pred')
from mltable2;
select * from rfr_pred a,mltable2 b where a.id = b.id;
```

- GBTRegression 算法

  数据准备：

```
Create table mltable2 (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable2 values(1,8.6,5.7,5.9,7.9);
Insert into mltable2 values(2,7.8,7.8,2.6,12.1);
```

  语句示例：

```
------------------训练-----------------------
drop table if exists gbtr_model;
select GBTRegressor('gbtr_model',label,col1,col2,col3) from mltable2;
select * from gbtr_model;
-------------------预测---------------------
```

```
select GBTRegressorPrediction('gbtr_model',col1,col2,col3,'id', 'gbtr_pred') from
mltable2;
select * from gbtr_pred a,mltable2 b where a.id = b.id;
```

- IsotonicRegression 算法

数据准备：

```
Create table mltable2 (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable2 values(1,8.6,5.7,5.9,7.9);
Insert into mltable2 values(2,7.8,7.8,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists gbtr_model;
select IsotonicRegression('isor_model',label,col1,col2,col3) from mltable2;
select * from isor_model;
-------------------预测---------------------
select IsotonicRegressionPrediction('isor_model',col1,col2,col3,'id', 'isor_pred')
from mltable2;
select * from isor_pred a,mltable2 b where a.id = b.id;
```

- AFTSurvivalRegressioin 算法

数据准备：

```
Create table afts_table (id int,label double,censor double,col1 double,col2 double);
Insert into afts_table values(1,8.6,1.0 ,5.9,7.9);
Insert into afts_table values(2,7.8,0.0,2.6,12.1);
```

语句示例：

```
------------------训练-----------------------
drop table if exists aftsr_model;
select AFTSurvivalRegression('aftsr_model',label,censor,col1,col2) from afts_table;
select * from aftsr_model;
-------------------预测---------------------
select AFTSurvivalRegressionPrediction('aftsr_model',col1,col2,'id', 'aftsr_pred')
from afts_table;
select * from aftsr_pred a,afts_table b where a.id = b.id;
```

3.5.2 非监督算法

1. 语法

- 训练语法

SELECT

TrainAlgorithmName(’model_result_table’,label,col1,col2,…,[’parms’])

from TrainTableName;

- 预测语法

SELECT

PredictAlgorithmName(‘model_result_table’,col1,col2,col3,…,’id’,’predict_result_table’)

from PredictTableName;

2. 语法说明

表3-38 训练语法说明

参数	说明	常见配置选项
TrainAlgorithmName	训练算法名称，必须指定，支持的训练算法见算法说明：训练算法部分	无
model_result_table	算法训练结果模型表，必须指定	无
TrainTableName	训练数据表名，用作算法的数据源，必须指定	无
label	训练数据表中的列，用作算法的参数，作为标签列，必须指定	无
col1,col2,…	训练数据表中的列，用作算法的参数，作为特征列，必须指定	无
parms	算法的其它参数，根据算法需要，可以不指定(内容格式：-ParamKey1 ParamValue1 -ParamKey2 ParamValue2 …)，详细解释见参数说明	无

表3-39 预测语法说明

参数名称	参数解释	备注
PredictAlgorithmName	预测算法名称，必须指定，支持的预测算法见算法说明：预测算法部分	无
model_result_table	模型表名，必须指定	用来指定使用哪个模型进行预测，对应训练中的model_result_table
PredictTableName	预测数据表名，用来预测的数据，必须指定	无
id	预测数据表的id列，用作算法的参数，作为标签列，必须指定	无
col1,col2,…	表的列，用作算法的参数，必须指定	无
predict_result_table	使用模型进行预测后的结果表	临时表

3. 算法说明

表3-40 训练算法

算法名	算法说明	备注
kmeans	Kmeans聚类训练函数	聚类
lda	Lda训练函数	聚类
gaussianmixture	高斯混合训练函数	聚类
als	Als协同过滤训练函数，算法必须加参数-ratingcol,-usercol,-itemcol	协同过滤

表3-41 预测算法

算法名	算法说明	备注
kmeansprediction	Kmeans聚类预测函数	聚类预测
ldaprediction	Lda预测函数	聚类预测
gmprediction	高斯混合预测函数	聚类预测
alsprediction	als预测函数	协同过滤预测

4. 参数说明

表3-42 参数说明

参数名	参数说明	支持算法
k	设置聚类K值，int型	Kmeans、lda、gaussianmixture
initmode	初始中心点选择方式，“random”“k-means”	Kmeans
initsteps	设置初始化步长	Kmeans
docconcentrations	文档中心，array[double]型	lda
docconcentration	Dirichlet分布的参数a，文档在主题上分布的先验参数(超参数a)。当前必须大于1，值越大，推断出的分布越平滑。默认为-1，自动设置	lda
optimizedocconcentration	是否优化文档中心列	lda
optimizer	优化器，用来学习LDA模型，一般是EMLDAOptimizer或OnlineLDAOptimizer；可选值：em或online	lda
alpha	ALS隐式反馈变化率	als
implicitprefs	使用显式反馈ALS变量或隐式反馈	als
itemcol	设置item列，必须要设置	als
nonnegative	是否支持非负	als
numblocks	并行计算的block数(-1为自动配置)	als
numitemblocks	Item的block数	als
numuserblocks	User的block数	als
rank	对应ALS模型中的隐藏因子数，即矩阵分解出的两个矩阵的新的行/列数	als
ratingcol	Rating列，必须要设置	als
usercol	User列，必须要设置	als
tol	优化算法迭代求解过程中的收敛阈值，默认值是1e-4，不能为负数	Kmeans、gaussianmixture
maxiter	优化算法求解的最大迭代次数(>=0)，默认值100	Kmeans、lda、gaussianmixture、als

参数名	参数说明	支持算法
regparam	控制模型的正则化过程，从而控制模型的过拟合情况	als
checkpointinterval	检查点间隔，当maxiter很大的时候，检查点可以帮助减少shuffle文件大小并且可以帮助故障恢复；int型	Lda、als
seed	随机种子，long型	Kmeans、lda、gaussianmixture、als

5. 示例

- KMeans 算法

  数据准备：

```
Create table mltable1 (id int,col1 double,col2 double,col3 double,col4 double);
Insert into mltable1 values(1,8.6,5.7,5.9,7.9);
Insert into mltable1 values(2,7.8,7.8,2.6,12.1);
```

  语句示例：

```
------------------训练-----------------------
drop table if exists km_result;
select KMeans('km_result',col1,col2,col3,col4,'-MaxIter 20 -K 2') from mltable1;
select * from km_result;
 -------------------预测---------------------
select KMeansPrediction('km_result',col1,col2,col3,col4,'id','km_pred') from
mltable1;
select * from km_pred a,mltable1 b where a.id = b.id;
```

- LDA 算法

  数据准备：

```
Create table mltable1 (id int,col1 double,col2 double,col3 double,col4 double);
Insert into mltable1 values(1,8.6,5.7,5.9,7.9);
Insert into mltable1 values(2,7.8,7.8,2.6,12.1);
```

  语句示例：

```
------------------训练-----------------------
drop table if exists lda_model;
select LDA('lda_model',col1,col2,col3,col4) from mltable1;
select * from lda_model;
-------------------预测---------------------
select LDAPrediction('lda_model',col1,col2,col3,col4,'id','lda_pred') from mltable1;
select * from lda_pred a,mltable1 b where a.id = b.id;
```

- GaussianMixture 算法

  数据准备：

```
Create table mltable1 (id int,col1 double,col2 double,col3 double,col4 double);
Insert into mltable1 values(1,8.6,5.7,5.9,7.9);
Insert into mltable1 values(2,7.8,7.8,2.6,12.1);
```

  语句示例：

```
------------------训练-----------------------
```

```
drop table if exists gm_model;
select GaussianMixture('gm_model',col1,col2,col3,col4) from mltable1;
select * from gm_model;
-------------------预测---------------------
select GMPrediction('gm_model',col1,col2,col3,col4,'id','gm_pred') from mltable1;
select * from gm_pred a,mltable1 b where a.id = b.id;
```

- ALS 算法

  数据准备：

```
Create table als_table (userid int,movieid int,rating double,timestamp bigint);
Insert into als_table values(0,2,3,1424380312);
Insert into als_table values(1,36,2,1424380312);
Insert into als_table values(2,4,3,1424380312);
```

  语句示例：

```
------------------训练-----------------------
drop table if exists als_model;
select ALS('als_model',userId,movieId,rating,'-UserCol userId -ItemCol movieId
-RatingCol rating') from als_table;
select * from als_model;
-------------------预测--------------------
select alsprediction('als_model',userId,movieId,rating,'userId', 'p_als_0') from
als_table;
select * from p_als_0;
```

3.5.3 模型选择

1. 语法

- 训练语法

  SELECT

  ModelSelectTypeAlgorithmName(label,col1,col2,col3,...,'model_result_table',

  '-Estimator TrainAlgorithmName [params1]',

  '[params2]',

  '-Evaluator EvaluatorName [params3]')

  from TrainTableName;

- 预测语法

  SELECT

  ModelSelectTypePredictAlgorithmName('model_result_table',col1,col2,col3,...,'id','predict_result_table')

  from PredictTableName;

2. 语法说明

表3-43 训练语法说明

参数	说明	常见配置选项
TrainAlgorithmName	训练算法名称，必须指定，支持的训练算法见算法说明：训练算法部分	无

参数	说明	常见配置选项
model_result_table	算法训练结果模型表，必须指定	无
TrainTableName	训练数据表名，用作算法的数据源，必须指定	无
label	训练数据表中的列，用作算法的参数，作为标签列，必须指定	无
col1,col2,…	训练数据表中的列，用作算法的参数，作为特征列，必须指定	无
EvaluatorName	模型选择所使用的评估器，可选：regression、multiclassclassification、binaryclassification	若使用二分类训练算法，则使用binaryclassification评估器；若使用多分类训练算法，则使用multiclassclassification评估器；若使用回归训练算法，则使用regression回归器；
params1	模型选择所使用的训练算法的参数，根据算法需要，可以不指定(内容格式：-ParamKey1 ParamValue1 -ParamKey2 ParamValue2 …)，详细解释见参数说明	无
params2	模型选择的网格参数设置，(内容格式：-N MultilayerPerceptronClassifierumFolds n -G_Param G_ParamValue …) 待选择的参数组，可以不指定，详细解释见参数说明	其中NumFolds参数必须指定
params3	评估算法设定的参数，可以不指定，(内容格式：-ParamKey1 ParamValue1 -ParamKey2 ParamValue2 …)	评估器参数ParamKey可选：metricname、labelcol、rawpredictioncol、predictioncol

表3-44 预测语法说明

参数名称	参数解释	备注
PredictAlgorithmName	预测算法名称，必须指定，支持的预测算法见算法说明：预测算法部分	无
model_result_table	模型表名，必须指定	用来指定使用哪个模型进行预测，对应训练中的model_result_table
PredictTableName	预测数据表名，用来预测的数据，必须指定	无
id	预测数据表的id列，用作算法的参数，作为标签列，必须指定	无
col1,col2,…	表的列，用作算法的参数，必须指定	无
predict_result_table	使用模型进行预测后的结果表	临时表

3. 算法说明

表3-45 训练算法

算法名	算法说明	备注
ModelSelect_CrossValidation	使用十则交叉的方式进行模型选择	十则交叉

表3-46 预测算法

算法名	算法说明	备注
CrossValidationPrediction	十则交叉模型选择预测函数	十则交叉

4. 参数说明

表3-47 参数说明

参数名	参数说明	备注
regParam	控制模型的正则化过程，从而控制模型的过拟合情况	无
maxIter	优化算法求解的最大迭代次数(>=0)，默认值100	无
threshold	二分类预测的阈值，范围[0,1]，double型	无
thresholds	多分类预测的阈值，array[double]型	无
checkpointInterval	检查点间隔，当maxiter很大的时候，检查点可以帮助减少shuffle文件大小并且可以帮助故障恢复，int型	无
fitIntercept	是否训练intercept，boolean型	无
standardization	训练模型前是否需要对训练特征进行标准化处理，boolean型	无
seed	随机种子	无
elasticNetParam	弹性网络混合参数(范围[0,1])，用于调节L1和L2之间的比例，两种正则化比例加起来是1，默认是0，即只使用L2正则化，设置为1，即只使用L1正则化，double型	无
tol	优化算法迭代求解过程中的收敛阈值，默认值：1e-4，不能为负数，double型	无
stepSize	每次迭代优化步长，double型	无
solver	优化的求解算法	无
aggregationDepth	设置分布式统计时的层数，主要用在treeAggregate中，数据量越大，可适当加大这个值，默认是2，int型	无
featuresCol	设置特征列	无
labelCol	设置标签列	无
predictionCol	设置预测列	无
inputCol	设置输入列，string型	无
inputCols	设置输入列，array[string]型	无

参数名	参数说明	备注
outputCol	设置输出列	无
weightCol	设置权重列	无

5. 示例

- 十则交叉模型选择

  数据准备：

```
Create table test_table (id int,label double,col1 double,col2 double,col3 double);
Insert into test_table values(1,8.6,5.7,5.9,7.9);
Insert into test_table values(2,7.8,7.8,2.6,12.1);
Insert into test_table values(3,8.8,9.6,4.6,32.1);
Insert into test_table values(4,11.9,12.5,8.6,62.1);
Insert into test_table values(5,16.9,17.5,6.6,32.1);
Insert into test_table values(6,21.9,23.4,10.6,43.2);
```

  语句示例：

```
select ModelSelect_CrossValidation(label,col1,col2,col3,'model_table1','-Estimator
LinearRegression -MaxIter 10000','-NumFolds 3 -G_elasticNetParam 0.4 0.5 -G_regParam
0.1 0.2','-Evaluator Regression -MetricName mse') from test_table;
select CrossValidationPrediction('model_table1',col1,col2,col3,'id','res_table1')
from test_table;
select * from res_table1;
```

3.5.4 特征工程

1. 语法

- 不包含训练过程的特征工程语法

  SELECT

  FeatureAlgorithmName(’transform_result_table’,’id’,col1,…,[’parms’])

  from TableName;

- 包含训练过程的特征工程语法

  SELECT

  FeatureAlgorithmName(‘model_result_table’,’transform_result_table’,’id’,col1,…,[’parms’]) from TableName;

2. 语法说明

表3-48 不包含训练过程的特征工程语法说明

参数	说明	常见配置选项
FeatureAlgorithmName	特征工程算法名称，必须指定	无
transform_result_table	转换结果表名，必须指定	无
TableName	表名，用作算法的数据源，必须指定	无
id	表的连接列的字段名称，指定tablename表中的列名作为该	无

参数	说明	常见配置选项
	参数值，必须指定	
col1,,…	表的列，用作算法的参数，必须指定	无
parms	算法的其它参数，根据算法需要，可以不指定(格式：-ParamKey1 ParamValue1 -ParamKey2 ParamValue2 …)	无

表3-49 包含训练过程的特征工程语法说明

参数	说明	常见配置选项
FeatureAlgorithmName	特征工程算法名称，必须指定	无
model_result_table	训练结果模型表，必须指定	无
transform_result_table	转换结果表名，必须指定	无
tablename	表名，用作算法的数据源，必须指定	无
id	表的连接列的字段名称，指定tablename表中的列名作为该参数值，必须指定	无
col1,,…	表的列，用作算法的参数，必须指定	无
parms	算法的其它参数，根据算法需要，可以不指定(格式：-ParamKey1 ParamValue1 -ParamKey2 ParamValue2 …)	无

3. 算法说明

表3-50 包含训练的特征工程算法

算法名	算法说明	支持参数
Word2vec	一种嵌入方法，可计算每个单词在给定语料库环境下的分布式词向量	InputCol：String //设置输入列 OutputCol：String //设置输出列 MaxIter：Int //设置最大迭代次数 MaxSentenceLength：Int //设置最大句子长度 MinCount：Int //设置 NumPartitions：Int //设置分区数 Seed：Long //设置种子 StepSize：Double //设置步长 VectorSize：Int //设置转换后的向量的长度 WindowSize：Int //设置
CountVectorizer	通过计数来将一个文档转换为向量	Binary：Boolean // InputCol：String //设置输入列 MinDF：Double // MinTF：Double // OutputCol：String //设置输出列 VocabSize：Int //设置词汇量
Pca	主成分分析，用来降低特征维度	InputCol：String //设置输入列—指定为features

算法名	算法说明	支持参数
		OutputCol：String //设置输出列 K：Int //设置主成分数
stringindexer	将字符串标签列编码为标签索引	InputCol：String //设置输入列 OutputCol：String //设置输出列
vectorindexer	解决向量数据集中的类别特征索引	MaxCategories：Int//最大类别数，小于它的认为是类别特征，否则被认为是连续特征 InputCol：String//设置输入列 OutputCol：String//设置输出列
standardscaler	用来转换一个向量行数据集，规范化每一个特征，使其标准差、均值为0	InputCol：String//设置输入列 OutputCol：String//设置输出列 WithStd：Boolean//设置是否使用标准差 WithMean：Boolean//设置是否使用均值
minmaxscaler	可将向量行缩放到范围[0,1]	InputCol：String//设置输入列 OutputCol：String//设置输出列
maxabsscaler	该特征的绝对值的最大值转换向量行，使其特征在范围[-1,1]内	InputCol：String//设置输入列 OutputCol：String//设置输出列
quantileDiscretizer	连续特征转化为类别特征,类别数通过numBuckets参数来设置	RelativeError：Double//近似的精度控制 NumBuckets：Int//箱化数 InputCol：String//设置输入列 OutputCol：String//设置输出列
RFormula	通过形如clicked ~ country + hour这个关系式，得到新的特征和标签列	FeaturesCol：String //设置特征列 LabelCol：String//设置标签列 Formula：Sring//设置关系式
ChiSqSelector	卡方选择器	FeaturesCol：String//设置特征列 LabelCol：String//设置标签列 OutputCol：Int//输出的属性 NumTopFeatures：Int//选中排在前几个的属性 Percentile：Double//选中排在前百分之几的属性 SelectorType：String//设置卡方选择器的类型

表3-51 不包含训练的特征工程算法

算法名	算法说明	支持参数
Tfidf	文本挖掘中使用的特征向量化方法	Binary：Boolean InputCol：String//设置输入列 OutputCol：String//设置输出列 NumFeatures：Int//设置特征数 MinDocFreq：Int
tokenizer	将文本(如：一个句子)分割成一个个	InputCol：String//设置输入列

算法名	算法说明	支持参数
	独立的词(通常是一个单词)	OutputCol：String//设置输出列
stopwordsremover	从数据中删除停用词	CaseSensitive：Boolean//设置大小写敏感 InputCol：String//设置输入列 OutputCol：String//设置输出列 StopWords：Array[String]//设置停用词
ngram	n-gram是一个序列，将输入特征转化为n-gram形式	N：Int //每个ngram中含有的元素个数 InputCol：String//设置输入列 OutputCol：String//设置输出列
binarize	二值化通过阈值将特征映射为(0/1)	InputCol：String//设置输入列 OutputCol：String//设置输出列 Threshold：Double//设置阈值
polynomialExpansion	多项式扩展是将特征映射到多项式空间的过程	Degree：Int InputCol：String//设置输入列 OutputCol：String//设置输出列
onehotencoder	将标签索引列映射为二进制向量列	InputCol：String//设置输入列 OutputCol：String//设置输出列 DropLast：Boolean//是否删除最后类别
normalizer	可以转换向量行的数据集，规范化每一向量(0到1范围内)	P：Double//设置P值 InputCol：String//设置输入列 OutputCol：String//设置输出列
elementwiseproduct	将每一个向量和权重向量使用元素对应相乘	InputCol：String//设置输入列 OutputCol：String//设置输出列 ScalingVec：Vector//设置乘数向量
vectorSlicer	输入特征向量，输出为新的特征向量，新的特征向量是原始特征向量的子集	Indices：Array[Int]//设置需要映射的索引 InputCol：String//设置输入列 OutputCol：String//设置输出列
Interaction	获取向量或实型列产生单一的向量列，向量列中包含输入向量中所有连接的结果	InputCols:：Array[String]//设置输入列 OutputCol:：String//设置输出列
VectorAssembler	可以将多个列合并为一个列	InputCols:：Array[String] //设置输入列 OutputCol：String//设置输出列

4. 示例

- TFIDF 算法

  数据准备：

```
Create table tfidf_table (id int,sentence string);
Insert into tfidf_table values(1, 'test');
Insert into tfidf_table values(2, 'mytest');
```

  运行语句：

```
select TFIDF('tfidf_trans_res','id',sentence ,'-InputCol sentence -OutputCol features') from tfidf_table;
select * from tfidf_trans_res;
select * from tfidf_trans_res a,tfidf_table b where a.id = b.id;
```

- Word2Vec 算法

  数据准备：

```
Create table word2vec_table (id int,text array<string>);
Insert into word2vec_table select 1,array('sparrow', 'mysparrow');
Insert into word2vec_table select 2,array('test','mytest');
```

  运行语句：

```
drop table if exists w2v_mod_res;
select Word2Vec ('w2v_mod_res','w2v_trans_res','id',text,'-VectorSize 3 -MinCount 0 -InputCol text -OutputCol result') from word2vec_table;
select * from w2v_mod_res;
select * from w2v_trans_res;
```

- CountVectorizer 算法

  数据准备：

```
Create table countvectorizer_table (id int,words array<string>);
Insert into countvectorizer_table select 1,array('sparrow', 'mysparrow');
Insert into countvectorizer_table select 2,array('test', 'mytest');
```

  运行语句：

```
drop table if exists countvec_mod_res;
select
CountVectorizer('countvec_mod_res','countvec_trans_res','id',words,'-VocabSize 3 –
MinDF 0 -InputCol words -OutputCol features') from countvectorizer_table;
select * from countvec_mod_res;
select * from countvec_trans_res;
```

- PCA 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.7,2.3,4.5,5.6);
Insert into mltable values(2,8.6,6.5,3.8,4.9);
```

  运行语句：

```
drop table if exists pca_mod_res;
select pca('pca_mod_res','pca_trans_res','id',col1,col2,col3,'-K 3 -InputCol features -OutputCol pcafeatures') from mltable;
select * from pca_mod_res;
select * from pca_trans_res;
```

- Tokenizer 算法

  数据准备：

```
Create table tokenizer_table (id int,sentence string);
Insert into tokenizer_table values(1, 'test');
Insert into tokenizer_table values(2, 'mytest');
```

  运行语句：

```
select Tokenizer('tokenizer_trans_res','id',sentence,'-InputCol sentence -OutputCol words') from tokenizer_table;
```

```
select * from tokenizer_trans_res;
```

- StopWordRemover 算法

  数据准备：

```
Create table stopwordremover_table (id int,raw array<string>);
Insert into stopwordremover_table select 1,array('sparrow', 'mysparrow');
Insert into stopwordremover_table select 2, array('test', 'mytest');
```

  运行语句：

```
select StopWordsRemover(' swr_trans_res ','id',raw,'-InputCol raw -OutputCol filtered')
from stopwordremover_table;
select * from swr_trans_res ;
```

- Ngram 算法

  数据准备：

```
Create table ngram_table (id int,words array<string>);
Insert into ngram_table select 1,array('sparrow', 'mysparrow');
Insert into ngram_table select 2, array('test', 'mytest');
```

  运行语句：

```
select NGram (' ngram_trans_res ','id', words,'-N 2 -InputCol words -OutputCol ngrams')
from ngram_table;
select * from ngram_trans_res;
```

- Binarizer 算法

  数据准备：

```
Create table binarizer_table (id int,feature double);
Insert into binarizer_table values(1,5.7);
Insert into binarizer_table values(2,2.6);
```

  运行语句：

```
select binarize('bin_trans_res','id', feature,'-Threshold 0.5 -InputCol feature
-OutputCol binarized_feature') from binarizer_table;
select * from bin_trans_res;
```

- PolynomialExpansion 算法

  数据准备：

```
Create table polynominalexpansion (id int,col1 double,col2 double);
Insert into polynominalexpansion values(1,5.7,11.8);
Insert into polynominalexpansion values(2,2.6,13.6);
```

  运行语句：

```
select PolynomialExpansion ('pe_trans_res','id',col1,col2,'-Degree 3 -InputCol
features -OutputCol polyFeatures') from polynominalexpansion;
select * from pe_trans_res;
```

- StringIndex 算法

  数据准备：

```
Create table stringindex_table (id int,category string);
Insert into stringindex_table values(1, 'test');
Insert into stringindex_table values(2, 'mytest');
```

  运行语句：

```
select StringIndexer ('si_mod_res',' si_trans_res ','id', category,' -InputCol
category -OutputCol categoryIndex') from stringindex_table;
```

```
select * from  si_mod_res;
select * from si_trans_res;
```

- OneHotEncoder 算法

  数据准备：

```
Create table onehotencoder_table (id int,category_index double);
Insert into onehotencoder_table values(1,5);
Insert into onehotencoder_table values(2,6);
```

  运行语句：

```
select OneHotEncoder ('onehot_trans_res','id',category_index,' -InputCol
category_index -OutputCol categoryVec') from onehotencoder_table;
select * from onehot_trans_res;
```

- VectorIndexer 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

  运行语句：

```
select VectorIndexer ('vi_mod_res','vi_trans_res','id',col1,col2,col3,'-
MaxCategories 10 -InputCol features -OutputCol indexed') from mltable;
select * from vi_mod_res;
select * from vi_trans_res;
```

- Normalizer 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

  运行语句：

```
select Normalizer('norm_trans_res','id',col1,col2,col3,'-P 1.0 -InputCol features
-OutputCol normFeatures') from mltable;
select * from norm_trans_res;
```

- StandardScaler 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

  运行语句：

```
select StandardScaler
('standard_mod_res','standard_trans_res','id',col1,col2,col3,'-WithStd true
-WithMean false -InputCol features -OutputCol scaledFeatures') from mltable;
select * from standard_mod_res;
select * from standard_trans_res;
```

- MinMaxScaler 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
```

```
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

运行语句：

```
select
MinMaxScaler('minmax_mod_res','minmax_trans_res','id',col1,col2,col3,'-InputCol
features -OutputCol scaledFeatures') from mltable;
select * from minmax_mod_res;
select * from minmax_trans_res;
```

- MaxAbsScaler 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

  运行语句：

```
select
MaxAbsScaler('maxabs_mod_res','maxabs_trans_res','id',col1,col2,col3,'-InputCol
features -OutputCol scaledFeatures') from mltable;


select * from  maxabs_mod_res;
select * from maxabs_trans_res;
```

- ElementwiseProduct 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

  运行语句：

```
select ElementwiseProduct('ep_trans_res','id',col1,col2,col3,'-ScalingVec 0.0 1.0 2.0
-InputCol vector -OutputCol transformedVector') from mltable;
select * from ep_trans_res;
```

- QuantileDiscretizer 算法

  数据准备：

```
Create table quantile_discret_table (id int,hour double);
Insert into quantile_discret_table values(1,5.6);
Insert into quantile_discret_table values(2,7.8);
```

  运行语句：

```
select QuantileDiscretizer('qd_mod_res','qd_trans_res','id',hour,'-NumBuckets 3
-InputCol hour -OutputCol result') from quantile_discret_table;
select * from qd_mod_res;
select * from qd_trans_res;
```

- VectorSlicer 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

  运行语句：

```
select VectorSlicer('trans_res','id',col1,col2,col3,'-Indices 1 2 -InputCol
userFeatures -OutputCol features') from mltable;
```

```
select * from trans_res;
```

- RFormula 算法

  数据准备：

```
Create table rformula_table (id int,counter string,hour double,clicked double);
Insert into rformula_table values(1, 'henan',2.3,5.6);
Insert into rformula_table values(2, 'hangzhou',6.5,3.8);
```

  运行语句：

```
select RFormula('rf_mod_res','rf_trans_res','id',counter,hour,clicked,'-Formula
clicked~counter+hour -FeaturesCol features -LabelCol label') from rformula_table;
select * from rf_mod_res;
select * from rf_trans_res;
```

- ChiSqSelector 算法

  数据准备：

```
Create table mltable (id int,label double,col1 double,col2 double,col3 double);
Insert into mltable values(1,5.6,2.3,4.5,5.6);
Insert into mltable values(2,7.8,6.5,3.8,4.9);
```

  运行语句示例：

```
Select
ChiSqSelector('css_mod_res','css_trans_res','id',col1,col2,col3,label,'-NumTopFeatu
res 1 -FeaturesCol features -LabelCol label -OutputCol selectedFeatures') from mltable;
select * from css_mod_res;
select * from css_trans_res;
```

- Interaction 算法

  数据准备：

```
Create table vectorassemble_table (id int,col1 double,col2 double,col3 double,col4
double,col5 double ,col6 double);
Insert into vectorassemble_table values(1,5.6,2.3,4.5,5.6,7.8,9.2);
Insert into vectorassemble_table values(2,7.8,6.5,3.8,4.9,8.7,2.6);
```

  运行语句示例：

```
select Interaction('inter_trans_res','id',col1,col2,col3,col4,col5,col6,'-InputCols
vec1 vec2 -OutputCol interactedCol') from vectorassemble_table;
Select * from inter_trans_res;
```

- VectorAssembler 算法

  数据准备：

```
Create table vectorassemble_table (id int,col1 double,col2 double,col3 double,col4
double,col5 double ,col6 double);
Insert into vectorassemble_table values(1,5.6,2.3,4.5,5.6,7.8,9.2);
Insert into vectorassemble_table values(2,7.8,6.5,3.8,4.9,8.7,2.6);
```

  运行语句示例：

```
Select VectorAssembler('va_trans_res','id',col1,col2,col3,'-InputCols col1 col2 col3
-OutputCol vec1') from vectorassemble_table;
select * from va_trans_res;
```

3.5.5 Pipeline 机制

1. 语法

- 训练语法

SELECT

pipelinetrain(‘pipeline_model_table’,’stage_algAndparms’,col1,col2,..)

from TableName;

- 预测语法

SELECT

pipelineprediction(‘pipeline_model_table’ ,col1,col2,..,’id’, ’predict_result_table’)

from TableName;

2. 语法说明

表3-52 训练语法说明

参数	说明	常见配置选项
pipelinetrain	指定为pipeline机制训练，且固定	无
pipeline_modelt_table	训练结果模型表，必须指定	无
Stage_algAndparms	Pipeline各阶段使用的算法和其参数	无
id	表的连接列的字段名称，指定tablename表中的列名作为该参数值，必须指定	无
col1,,…	表的列，用作算法的参数，必须指定	无
TableName	表名，用作算法的数据源，必须指定	无

表3-53 预测语法说明

参数	说明	常见配置选项
pipelineprediction	指定为pipeline机制预测，必须指定	无
pipeline_modelt_table	预测使用的模型表，必须指定	无
col1,,…	表的列，用作算法的参数，必须指定	无
id	表的连接列的字段名称，指定tablename表中的列名作为该参数值，必须指定	无
predict_result_table	预测结果表名，将预测结果存储在该表中，必须指定	无
tablename	表名，用作算法的数据源，必须指定	无

3. 参数说明

Stage_algAndparms：

通过-alg 来指定该阶段使用的算法（算法说明，算法说明_1和算法说明_2），其他参数参考对应算法中的说明。

4. 示例

- 创建训练数据

```
create table pipeline_train_table (id int,text string,label double);
insert into pipeline_train_table values (0,"a b c d e spark",1.0);
insert into pipeline_train_table values (1,"b d",0.0);
insert into pipeline_train_table values (2,"spark f g h",1.0);
insert into pipeline_train_table values (3,"hadoop mapreduce",0.0);
```

- 使用 SQL 调用 pipeline 机制，并查看训练模型存储信息

```
select piplinetrain('pipeline_model',
       '-alg tokenizer -inputcol text -outputcol words',
       '-alg hashingtf -inputcol words -outputcol features',
       '-alg logisticregression',
       id,text,label)
from pipeline_train_table;
select * from pipeline_model;
```

- 使用 SQL 预测新数据，并查看预测结果

```
create table pipeline_test_table (id int,text string);
insert into pipeline_test_table values (4,"spark i j k"),
(5,"l m n"),
(6,"spark hadoop spark"),
(7,"apache hadoop");
select pipelineprediction('pipeline_model',text,'id','pipeline_res_table') from
pipeline_test_table;
select * from pipeline_res_table;
```

3.6 列加密

DLH 支持对 String 数据类型的列数据进行加密，支持 AES、DES 和 SM4 三种加密算法及公钥和私钥两种加密方式。

注意

- 目前 DLH 只支持对 text、orc、parquet 格式的表进行列加密。
- DLH 不支持以 load 的方式将数据导入加密表。
- 当前版本中，暂不支持复杂类型的加密嵌套。

用户需要在建表时指定列加密相关参数。列加密参数如表 3-54 所示，加密参数一旦设置就不允许再进行修改。

表3-54 列加密参数

参数名	说明
encrypt.columns	该参数指定加密列，列必须是String类型
encrypt.algorithm	该参数指定加密算法，支持AES、DES和SM4
encrypt.type	该参数指定加密方式，支持public和private。可以不指定，默认public

参数名	说明
encrypt.key	该参数指定密钥。当设置encrypt.type为private时才需要设置此参数。DES加密算法要求密钥的长度大于8位，SM4加密算法要求长度为16
encrypt.pattern	该参数指定加密模式，支持CBC或ECB。在加密算法为SM4时设置

3.6.1 公钥加密

使用公钥加密，会将数据加密后再存储。只有使用 DLH 查询才能得到真实的数据，直接查看数据文件或者通过其他方式查询，只能查看到加密数据。

示例：

- 在 DLH 中建立 ORC 格式表，并使用公钥加密和 DES 加密算法对表的列 name、passwd 进行数据加密。

```
create table employee(id int,name string,passwd string)
tblproperties("encrypt.columns"="name,passwd","encrypt.algorithm"="DES");
insert into employee values(1,'zhangsan','43211');
select * from employee;
+-----+-----------+---------+
| id  |   name    | passwd  |
+-----+-----------+---------+
| 1   | zhangsan  | 43211   |
+-----+-----------+---------+
```

- 在 DLH 中建立 parquet 格式的表，并使用公钥加密方式和 AES 加密算法对列 salary 进行数据加密。

```
create table salary(id int,name string,salary string) stored as parquet tblproperties
("encrypt.columns"="salary","encrypt.algorithm"="AES");
insert into salary values(1,'zhangsan','8999.5');
select * from salary;
+-----+-----------+---------+
| id  |   name    | salary  |
+-----+-----------+---------+
| 1   | zhangsan  | 8999.5  |
+-----+-----------+---------+
```

- 在 DLH 中建立 textfile 格式的表，并使用公钥加密方式和 SM4 加密算法的 CBC 模式对列 salary 进行数据加密。

```
create table employeeinfo(id int,city string,address string) stored as textfile
tblproperties("encrypt.columns"="city,address","encrypt.algorithm"="SM4","encrypt.p
attern"="CBC");
insert into employeeinfo values (1,'hnzz','yulanjie');
select * from employeeinfo;
+-----+-------+-----------+
| id  | city  |  address  |
+-----+-------+-----------+
| 1   | hnzz  | yulanjie  |
+-----+-------+-----------+
```

- 配置 Hive 连接 DLH 使用的 metastore 地址（如：thrift://sharedev1.hde.com:19083,thrift://sharedev2.hde.com:19083），通过 Hive 查看这两个表，只能看到加密数据。

```
[root@node2~]# beeline --hiveconf
hive.metastore.uris=thrift://sharedev1.hde.com:19083,thrift://sharedev2.hde.com:190
83
beeline> !connect jdbc:hive2://node2:10000
Connecting to jdbc:hive2://node2:10000
Enter username for jdbc:hive2://node2:10000: hdfs
Enter password for jdbc:hive2://node2:10000:
Connected to: Apache Hive (version 2.1.1-cdh6.2.0)
Driver: Hive JDBC (version 2.1.1-cdh6.2.0)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://node2:10000> select * from employee;
+--------------+--------------------------+------------------+--+
| employee.id  |       employee.name      | employee.passwd  |
+--------------+--------------------------+------------------+--+
| 1            | OYuK0vlc7KHZo7+6lh609A== | xSIrlXaRby4=     |
+--------------+--------------------------+------------------+--+
1 row selected (3.923 seconds)
0: jdbc:hive2://node2:10000> select * from salary;
+------------+-------------+--------------------------+--+
| salary.id  | salary.name |       salary.salary      |
+------------+-------------+--------------------------+--+
| 1          | zhangsan    | 0K51RX+6P2FWJ6Rmd7A8Dg== |
+------------+-------------+--------------------------+--+
1 row selected (1.519 seconds)

0: jdbc:hive2://node2:10000> select * from employeeinfo;
+------------------+------------------------+---------------------------+--+
| employeeinfo.id  |   employeeinfo.name    |    employeeinfo.salary    |
+------------------+------------------------+---------------------------+--+
| 1                | eCccpUM/EjhaKlzkIl3qg== |    P3YPNJ8bm4Mc3lbdK2vr8w==  |
+------------------+------------------------+---------------------------+--+
1 row selected (1.519 seconds)
```

3.6.2 私钥加密

私钥加密也会将数据加密后再存储，同时要求在查询或者插入数据之前先设置私钥，否则认为操作是不被允许的。设置的私钥是对当前 Session 有效的。

设置私钥的语法如下：

set encrypt.key.db_name.table_name=encrypt_key;

示例：

- 在 DLH 中建立 orc 格式的表，并指定私钥加密和 DES 加密算法对数据表列 name、passwd 进行数据加密。

```
create table employee1(id int,name string,passwd string)
tblproperties("encrypt.columns"="name,passwd","encrypt.algorithm"="DES","encrypt.type"="private","encrypt.key"="qwertyui");
```

- 当不设置私钥时，插入和查询都是不允许的。

```
insert into employee1 values(1,'zhangsan','43211');
java.lang.Exception: encrypt.key is not right
```

- 设置私钥后可以正常插入和查询。

```
set encrypt.key.default.employee1=qwertyui;
insert into employee1 values(1,'zhangsan','43211');
select * from employee1;
+-----+-----------+---------+
| id  |   name    | passwd  |
+-----+-----------+---------+
| 1   | zhangsan  | 43211   |
+-----+-----------+---------+
Time taken: 0.111 seconds, Fetched 1 row(s)
```

- 查看表信息时，encrypt.key 也已经被加密。

```
sparrow> show create table employee1;
CREATE TABLE `employee1`(`id` int, `name` string, `passwd` string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = '1'
)
STORED AS
  INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
  OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
TBLPROPERTIES (
  'numFiles' = '1',
  'transient_lastDdlTime' = '1536046320',
  'totalSize' = '40',
  'encrypt.algorithm' = 'DES',
  'encrypt.key' = 'FumPB0oW5xSwdVxa/MkAnQ==',
  'encrypt.columns' = 'name,passwd',
  'encrypt.type' = 'private',
  'COLUMN_STATS_ACCURATE' = 'true'
)
```

3.7 权限访问控制

注意

根据集群是否开启“安全管理/权限管理”功能，集群中新建用户获取 DLH 组件权限的方式不同：

- 当集群未开启权限管理时，集群中新建用户可直接对 DLH 组件执行库表的创建、修改、插入、删除等操作，无需进行授权操作。
- 当集群开启权限管理时，集群中新建用户的 DLH 组件权限需要在[集群权限/角色管理]页面进行配置，集群用户绑定已配置 DLH 权限的角色后才可执行相关操作。

权限管理是集群安全管理的重要组成部分，在开启权限管理的集群中，权限基于角色进行统一管理，角色是权限的集合。

为集群中用户赋予权限的整体流程如下：

(1) 新建角色，并为角色配置权限。

(2) 新建用户，并将角色分配用户，用户即拥有角色所具有的权限。

在开启权限管理的集群中，用户对 DLH 组件的操作需被赋予相关权限后才能执行。

DLH 支持对数据库表和数据库 UDF 配置权限（支持使用通配符*模糊适配）。数据库表可对数据库、数据表、列配置权限，权限包括：select、update、create、drop、alter、index、use，其中 all 表示配置所有权限。数据库 UDF 可对数据库、UDF 配置权限，权限包括 create、drop。

DLH 操作所需权限对应关系如表 3-55 所示。

表3-55 DLH 权限说明

权限类型	对应的组件常用操作
select	库表查询等相关操作，如：select、export、show、describe等
update	插入或更新库表等相关操作，如：insert、insert overwrite、delete、update、load等
create	创建库表等相关操作，如create table、create database等 【说明】库表的创建者默认不拥有库表的使用权限，但是通过“权限管理”功能可授予其使用权限
drop	删除库表等相关操作，如：drop table等
alter	修改库表等相关操作，如：alter table等
index	索引等相关操作，如：create index等
use	执行show database、use database等操作
all	支持以上所有操作

3.7.1 权限操作示例

1. 数据库权限控制

表3-56 数据库授权配置

操作	权限要求（数据库示例为 dlhtest）
create database	数据库：*或dlhtest，数据表：*，列：*，权限：create
use database	数据库：*或dlhtest，数据表：*，列：*，权限：use
drop database	数据库：*或dlhtest，数据表：*，列：*，权限：drop

下面以授予 create 权限为例，对 dlhtest 数据库授予 create 权限（create 权限默认拥有 use 权限），介绍数据库的权限访问控制（其它权限的授予方式与 create 权限的操作类似，不再重复说明）。操作步骤如下：

(1) 新建用户

在[集群权限/用户管理]页面，新建 dlhuser01 用户，授权前执行创建 dlhtest 数据库操作。如图 3-4所示，此时显示 dlhuser01 用户没有 dlhtest 数据库的 create 权限。

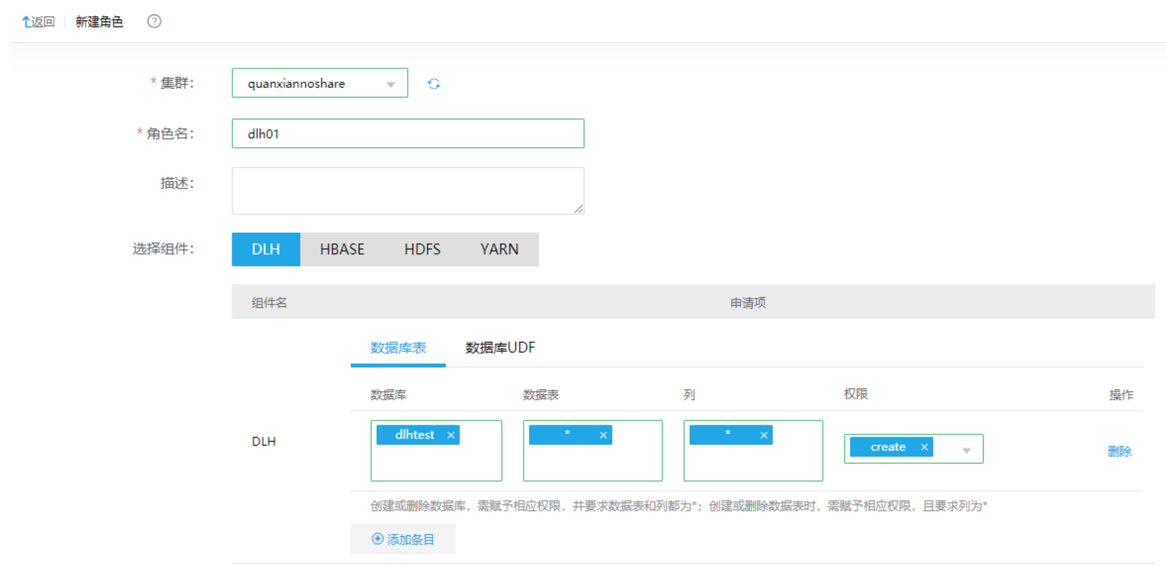
图3-4 无执行权限

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> create database dlhtest;
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied:
user [dlhuser01] does not have [CREATE] privilege on [dlhtest] (state=42000,code=40000)
```

(2) 新建角色

在[集群权限/角色管理]页面，新建 dlh01 角色，授予 dlh01 角色 dlhtest 数据库的 create 权限。

图3-5 新建角色



(3) 用户绑定角色

在[集群权限/用户管理]页面，单击 dlhuser01 用户对应的<修改用户授权>按钮，为其绑定 dlh01 角色。

图3-6 为用户绑定角色进行授权

(4) 使用 dlhuser01 用户重新执行创建 dlhtest 数据库的操作。如图 3-7 所示，此时显示 dlhtest 数据库创建成功。

图3-7 执行成功

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> create database dlhtest;
No rows affected (2.796 seconds)
INFO  : Compiling command(queryId=hive_20211111150250_5db12d2a-f776-4dff-8364-490e3fb4fb04): create database
dlhtest
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20211111150250_5db12d2a-f776-4dff-8364-490e3fb4fb04); Time
taken: 0.332 seconds
INFO  : Executing command(queryId=hive_20211111150250_5db12d2a-f776-4dff-8364-490e3fb4fb04): create database
dlhtest
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20211111150250_5db12d2a-f776-4dff-8364-490e3fb4fb04); Time
taken: 2.173 seconds
INFO  : OK
```

2. 表权限控制

表3-57 表授权配置

操作	权限要求（数据库示例为 dlhtest，表示例为 dlhtab1）
create table	数据库：*或dlhtest，数据表：*或dlhtab1，列：*，权限：create
alter table	数据库：*或dlhtest，数据表：*或dlhtab1，列：*，权限：alter
insert into table	• Hive 数据库：*或 dlhtest，数据表：*或 dlhtab1，列：*，权限：update • YARN 队列：*或 default，权限：submit-app
delete from table（针对ACID表）	• Hive 数据库：*或 dlhtest，数据表：*或 dlhtab1，列：*，权限：update • YARN

操作	权限要求（数据库示例为 dlhtest，表示例为 dlhtab1）
	队列：*或 default，权限：submit-app
update table（针对ACID表）	• Hive 数据库：*或 dlhtest，数据表：*或 dlhtab1，列：*，权限：update • YARN 队列：*或 default，权限：submit-app
select * from table	• Hive 数据库：*或 dlhtest，数据表：*或 dlhtab1，列：*，权限：select • YARN 队列：*或 default，权限：submit-app
create index	数据库：*或dlhtest，数据表：*或dlhtab1，列：*，权限：index
drop table	数据库：*或dlhtest，数据表：*或dlhtab1，列：*，权限：drop

示例 1：下面以授予 update 权限为例，对 dlhtest 数据库 dlhtab1 数据表授予 update 权限，介绍数据表的权限访问控制。操作步骤如下：

(1) 新建用户

在[集群权限/用户管理]页面，新建 dlhuser21 用户，授权前执行插入 dlhtest.dlhtab1 数据表的操作。如图 3-8 所示，此时显示 dlhuser21 用户没有 dlhtest.dlhtab1 数据表的 update 权限。

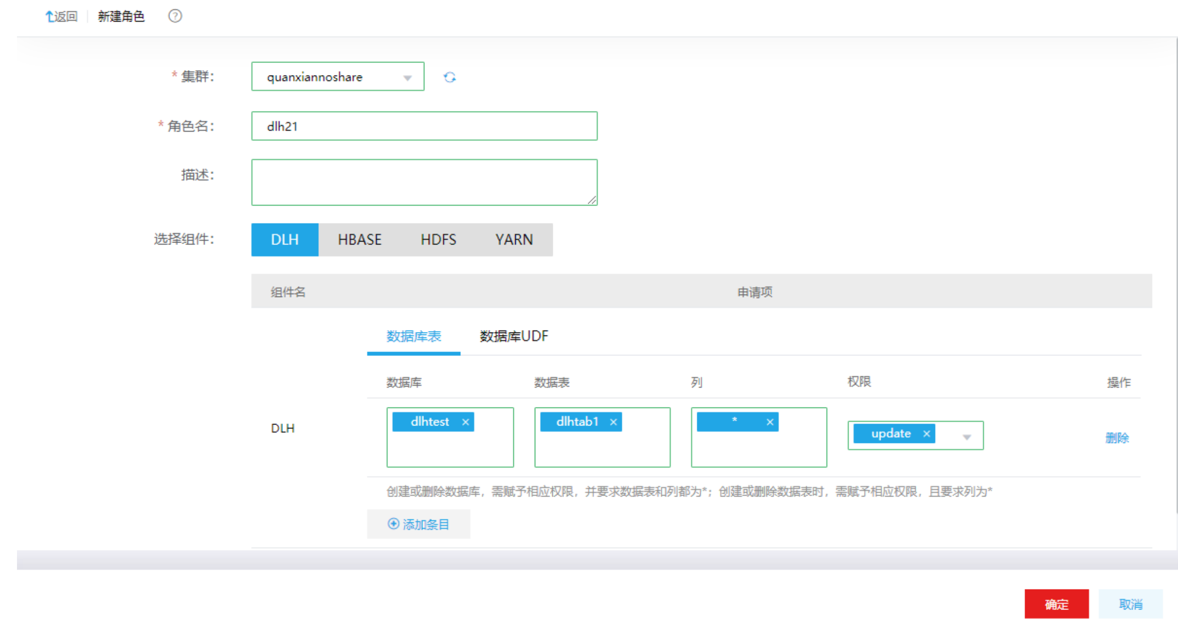
图3-8 无执行权限

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> insert into dlhtest.dlhtab1 values(1,'abc');
Error: Error while compiling statement: FAILED: SemanticException org.apache.hadoop.hive.ql.metadata.
InvalidTableException: Table not found dlhtab1 (state=42000,code=40000)
```

(2) 新建角色

在[集群权限/角色管理]页面，新建 dlh21 角色，授予 dlh21 角色 dlhtest.dlhtab1 数据表的 update 权限。

图3-9 新建角色



(3) 用户绑定角色

在[集群权限/用户管理]页面，单击 dlhuser21 用户对应的<修改用户授权>按钮，为其绑定角 dlh21 色。

(4) 使用 dlhuser21 用户重新执行插入 dlhtest.dlhtab1 数据表的操作。如图 3-10 所示，此时显示 dlhuser21 用户没有提交到 yarn default 队列的权限。

图3-10 无执行权限

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> insert into dlhtest.dlhtab1 values(1,'abc');
INFO  : Compiling command(queryId=hive_20211111152430_beab682f-c45a-43f0-be87-92543dfc7fe9): insert into dlhtest.dlhtab1
valu
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:_col0, type:int, comment:null), FieldSchema(name:_
col1, ment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20211111152430_beab682f-c45a-43f0-be87-92543dfc7fe9); Time taken: 2.87
secon
INFO  : Executing command(queryId=hive_20211111152430_beab682f-c45a-43f0-be87-92543dfc7fe9): insert into dlhtest.dlhtab1
valu
INFO  : Query ID = hive_20211111152430_beab682f-c45a-43f0-be87-92543dfc7fe9
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in serial mode
ERROR : FAILED: Execution Error, return code 30041 from org.apache.hadoop.hive.ql.exec.spark.SparkTask. Failed to create
Sparrk session 6eaa6ed6-264b-4a66-82eb-954b05a21832_0: java.lang.RuntimeException: spark-submit process failed with exit
code 1 a
INFO  : Completed executing command(queryId=hive_20211111152430_beab682f-c45a-43f0-be87-92543dfc7fe9); Time taken: 18.597
sec
Error: Error while processing statement: FAILED: Execution Error, return code 30041 from org.apache.hadoop.hive.ql.exec.s
parked to create Spark client for Spark session 6eaa6ed6-264b-4a66-82eb-954b05a21832_0: java.lang.RuntimeException:spark-
submit ith exit code 1 and error ? (state=42000,code=30041)
```

(5) 修改角色授权

在[集群权限/角色管理]页面，单击 dlh21 角色对应的<编辑>按钮，修改该角色的组件权限设置，授予 dlh21 角色 default 队列的 submit-app 权限。

图3-11 编辑角色

(6) 使用 dlhuser21 用户重新执行插入 dlhtest.dlhtab1 数据表的操作。如图 3-12 所示，此时显示 dlhtest. dlhtab1 数据表插入数据成功。

图3-12 执行成功

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> insert into dlhtest.dlhtab1 values(1,'abc');
1 row affected (60.116 seconds)
INFO  : Compiling command(queryId=hive_20211111152643_1bf65298-e12a-4ea2-afd1-d4560ca29084): insert into dlhtest.dlhtab1 valu
es(1,'abc')
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:_col0, type:int, comment:null), FieldSchema(name:_col1,
type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20211111152643_1bf65298-e12a-4ea2-afd1-d4560ca29084); Time taken: 0.375 seco
nds
INFO  : Executing command(queryId=hive_20211111152643_1bf65298-e12a-4ea2-afd1-d4560ca29084): insert into dlhtest.dlhtab1 valu
es(1,'abc')
INFO  : Query ID = hive_20211111152643_1bf65298-e12a-4ea2-afd1-d4560ca29084
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in serial mode
INFO  : Running with YARN Application = application_1636529544651_0024
INFO  : Kill Command = /usr/hdp/current/hadoop-client/bin/yarn application -kill application_1636529544651_0024
INFO  : Hive on Spark Session Web UI URL: http://ryh148.hde.com:35821
INFO  :
Query Hive on Spark job[0] stages: [0]
INFO  : Spark job[0] status = RUNNING
INFO  : Job Progress Format
CurrentTime StageId_StageAttemptId: SucceededTasksCount(+RunningTasksCount-FailedTasksCount)/TotalTasksCount
INFO  : 2021-11-11 15:27:32,710 Stage-0_0: 0(+1)/1
INFO  : 2021-11-11 15:27:35,745 Stage-0_0: 0(+1)/1
INFO  : 2021-11-11 15:27:36,748 Stage-0_0: 1/1 Finished
INFO  : Spark job[0] finished successfully in 7.14 second(s)
INFO  : Starting task [Stage-0:MOVE] in serial mode
INFO  : Loading data to table dlhtest.dlhtab1 from hdfs://mycluster/dlh/warehouse/tablespace/managed/hive/dlhtest.db/dlhtab1/
.hive-staging_hive_2021-11-11_15-26-43_257_1095522753384004648-2/-ext-10000
INFO  : Starting task [Stage-2:STATS] in serial mode
INFO  : Table dlhtest.dlhtab1 stats: [numFiles=1, numRows=1, totalSize=283, rawDataSize=91, numFilesErasureCoded=0]
INFO  : Completed executing command(queryId=hive_20211111152643_1bf65298-e12a-4ea2-afd1-d4560ca29084); Time taken: 54.703 sec
onds
INFO  : OK
```

示例 2：下面以授予 select 权限为例，对 dlhtest 数据库 dlhtab1 数据表授予 select 权限，介绍数据表的权限访问控制。操作步骤如下：

(1) 继续使用 dlhuser21 用户执行查询 dlhtest.dlhtab1 数据表的操作。如图 3-13 所示，此时显示 dlhuser21 用户没有 dlhtest. dlhtab1 数据表的 select 权限。

图3-13 无执行权限

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> select * from dlhtest.dlhtab1;
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [dlhuser21]
 does not have [SELECT] privilege on [dlhtest/dlhtab1/*] (state=42000,code=40000)
```

(2) 修改角色授权

在[集群权限/角色管理]页面，单击 dlh21 角色对应的<编辑>按钮，修改该角色的组件权限设置，授予 dlh21 角色 dlhtest. dlhtab1 数据表的 select 权限。

图3-14 编辑角色

(3) 使用 dlhuser21 用户重新执行查询 dlhtest. dlhtab1 数据表的操作。如图 3-15 所示，此时显示查询 dlhtest. dlhtab1 数据表数据成功。

图3-15 执行成功

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> select * from dlhtest.dlhtab1;
+-------------+---------------+
| dlhtab1.id  | dlhtab1.name  |
+-------------+---------------+
| 1           | abc           |
+-------------+---------------+
1 row selected (0.832 seconds)
INFO  : Compiling command(queryId=hive_20211111152947_3c724834-4988-4bc1-8d56-eabb77ea3ba6): select * from dlhtest.dlhtab1
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:dlhtab1.id, type:int, comment:null), FieldSchema(name:dl
htab1.name, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20211111152947_3c724834-4988-4bc1-8d56-eabb77ea3ba6); Time taken: 0.425 seco
nds
INFO  : Executing command(queryId=hive_20211111152947_3c724834-4988-4bc1-8d56-eabb77ea3ba6): select * from dlhtest.dlhtab1
INFO  : Completed executing command(queryId=hive_20211111152947_3c724834-4988-4bc1-8d56-eabb77ea3ba6); Time taken: 0.005 seco
nds
INFO  : OK
```

3. 列级别权限控制

表3-58 授权配置

操作	权限要求（数据库示例为 dlhtest，数据表示例为 dlhtab1，列示例为 name）
insert into table	• Hive 数据库：*或 dlhtest，数据表：*或 dlhtab1，列：name，权限：update • YARN 队列：*或 default，权限：submit-app
select name from table	数据库：*或dlhtest，数据表：*或dlhtab1，列：name，权限：select

示例 1：下面以授予 select 权限为例，对 dlhtest 数据库 dlhtab1 数据表的 name 列授予 select 权限，介绍数据列的权限访问控制。操作步骤如下：

(1) 新建用户

在[集群权限/用户管理]页面，新建 dlhuser02 用户，授权前执行在 dlhtest.dlhtab1 数据表中查询 name 列的操作。如图 3-16所示，此时显示 dlhuser02 用户没有 dlhtab1 数据表的查询权限。

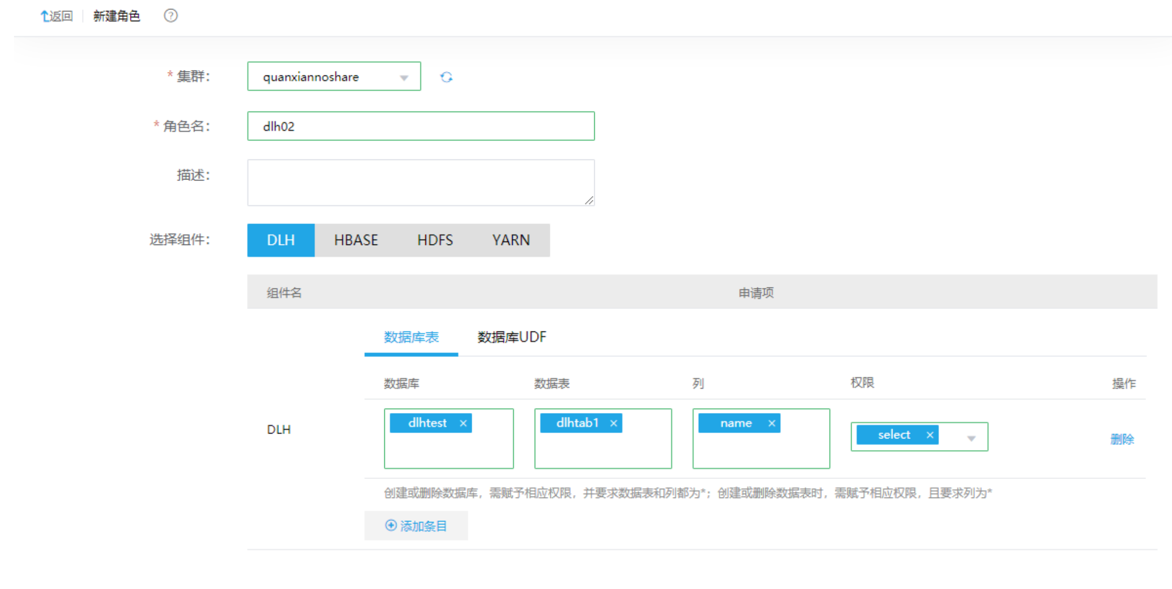
图3-16 无执行权限

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> select name from dlhtest.dlhtab1;
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user
[dlhuser02] does not have [SELECT] privilege on [dlhtest/dlhtab1/*] (state=42000,code=40000)
```

(2) 新建角色

在[集群权限/角色管理]页面，新建 dlh02 角色，授权 dlhtest 数据库 dlhtab1 数据表的 name 列的 select 权限。

图3-17 新建角色



(3) 用户绑定角色

在[集群权限/用户管理]页面，单击 dlhuser02 用户对应的<修改用户授权>按钮，为其绑定 dlh02 角色。

(4) 使用 dlhuser02 用户重新执行在 dlhtest.dlhtab1 数据表中查询 name 列的操作。如图 3-18 所示，此时显示查询操作执行成功。

图3-18 执行成功

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> select name from dlhtest.dlhtab1;
+-------+
| name  |
+-------+
| abc   |
+-------+
1 row selected (0.782 seconds)
INFO  : Compiling command(queryId=hive_20211111153303_62085fe5-6a2b-4b7a-b3d0-6840723e5928): select name from dlhtest.dlhtab1
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:name, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20211111153303_62085fe5-6a2b-4b7a-b3d0-6840723e5928); Time taken: 0.316 seconds
INFO  : Executing command(queryId=hive_20211111153303_62085fe5-6a2b-4b7a-b3d0-6840723e5928): select name from dlhtest.dlhtab1
INFO  : Completed executing command(queryId=hive_20211111153303_62085fe5-6a2b-4b7a-b3d0-6840723e5928); Time taken: 0.007 seconds
INFO  : OK
```

示例 2：下面以授予 update 权限为例，对 dlhtest 数据库 dlhtab1 数据表的 name 列授予 update 权限，介绍数据列的权限访问控制。操作步骤如下：

(5) 继续使用 dlhuser02 用户在 dlhtest. dlhtab1 数据表的 name 列执行插入的操作。如图 3-19 所示，此时显示没有 update 权限。

图3-19 无执行权限

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> insert into dlhtest.dlhtab1(name) values("sss");
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [dlhuser02] does not have [UPDATE] privilege on [dlhtest/dlhtab1/*] (state=42000,code=40000)
```

(6) 修改角色授权

在[集群权限/角色管理]页面，单击 dlh02 角色对应的<编辑>按钮，修改该角色的组件权限设置，授予 dlh02 角色 dlhtest. dlhtab1 中 name 列的 update 权限，及 YARN 的 default 队列的 submit-app 权限。

图3-20 编辑角色

(7) 使用 dlhuser02 用户重新在 dlhtest. dlhtab1 数据表的 name 列执行插入的操作。如图 3-21 和图 3-22 所示，此时显示插入操作执行成功。

图3-21 执行成功

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> insert into dlhtest.dlhtab1(name) values("sss");
1 row affected (55.078 seconds)
INFO  : Compiling command(queryId=hive_20211111153509_ab5bbcd6-eb69-4111-86d0-aa7c8339d2cb): insert into dlhtest.dlhtab1(name
) values("sss")
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:_col0, type:int, comment:null), FieldSchema(name:_col1,
type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20211111153509_ab5bbcd6-eb69-4111-86d0-aa7c8339d2cb); Time taken: 0.285 seco
nds
INFO  : Executing command(queryId=hive_20211111153509_ab5bbcd6-eb69-4111-86d0-aa7c8339d2cb): insert into dlhtest.dlhtab1(name
) values("sss")
INFO  : Query ID = hive_20211111153509_ab5bbcd6-eb69-4111-86d0-aa7c8339d2cb
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in serial mode
INFO  : Running with YARN Application = application_1636529544651_0025
INFO  : Kill Command = /usr/hdp/current/hadoop-client/bin/yarn application -kill application_1636529544651_0025
INFO  : Hive on Spark Session Web UI URL: http://ryh149.hde.com:40671
INFO  :
Query Hive on Spark job[0] stages: [0]
INFO  : Spark job[0] status = RUNNING
INFO  : Job Progress Format
CurrentTime StageId_StageAttemptId: SucceededTasksCount(+RunningTasksCount-FailedTasksCount)/TotalTasksCount
INFO  : 2021-11-11 15:35:51,968 Stage-0_0: 0/1
INFO  : 2021-11-11 15:35:54,984 Stage-0_0: 0/1
INFO  : 2021-11-11 15:35:57,997 Stage-0_0: 1/1 Finished
INFO  : Spark job[0] finished successfully in 9.11 second(s)
INFO  : Starting task [Stage-0:MOVE] in serial mode
INFO  : Loading data to table dlhtest.dlhtab1 from hdfs://mycluster/dlh/warehouse/tablespace/managed/hive/dlhtest.db/dlhtab1/
.hive-staging_hive_2021-11-11_15-35-09_171_5831359451213925835-7/-ext-10000
INFO  : Starting task [Stage-2:STATS] in serial mode
INFO  : Table dlhtest.dlhtab1 stats: [numFiles=2, numRows=2, totalSize=563, rawDataSize=178, numFilesErasureCoded=0]
INFO  : Completed executing command(queryId=hive_20211111153509_ab5bbcd6-eb69-4111-86d0-aa7c8339d2cb); Time taken: 49.729 sec
onds
INFO  : OK
```

图3-22 查询插入结果

```
0: jdbc:hive2://ryh149.hde.com:2181,ryh147.hd> select name from dlhtest.dlhtab1;
+-------+
| name  |
+-------+
| abc   |
| sss   |
+-------+
2 rows selected (0.899 seconds)
INFO  : Compiling command(queryId=hive_20211111153651_8db72462-4916-479e-884a-49aeae90c2c9): select name from dlhtest.dlhtab1
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:name, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20211111153651_8db72462-4916-479e-884a-49aeae90c2c9); Time taken: 0.659 seco
nds
INFO  : Executing command(queryId=hive_20211111153651_8db72462-4916-479e-884a-49aeae90c2c9): select name from dlhtest.dlhtab1
INFO  : Completed executing command(queryId=hive_20211111153651_8db72462-4916-479e-884a-49aeae90c2c9); Time taken: 0.004 seco
nds
INFO  : OK
```

4. UDF 权限控制

表3-59 授权配置

操作	权限要求（数据库示例为 dlhtest，UDF 示例为 mylowerperm1）
create function	数据库：*或dlhtest，UDF：*或mylowerperm1，权限：create
drop function	数据库：*或dlhtest，UDF：*或mylowerperm1，权限：drop
select function	不需要授权
show functions	不需要授权

(1) 创建函数类文件

```
package com.dlh.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public final class LowerFunc extends UDF{
    public Text evaluate(final Text s){
        if(s == null){return null;}
        return new Text(s.toString().toLowerCase());
    }
}
```

其中 pom.xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>dlhUdfTest</artifactId>
    <version>1.0</version>

    <dependencies>
```

```
        <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-exec</artifactId>
        <version>2.1.1-cdh6.2.0</version>
        </dependency>
    </dependencies>
    <build>

        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.4.3</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.0</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>

    </build>

</project>
```

(2) 将上述文件编译打包后上传到测试集群中，本示例中打包后名字为 dlhUdfTest-1.0.jar。

(3) 新建用户

在[集群权限/用户管理]页面，新建 udfuser03 用户。如图 3-23 所示，将 udf 测试 jar 包上传到 HDFS 的/tmp/udflib 目录下，在授权前执行 create 操作。如图 3-24 所示，此时显示用户没有 dlhtest 数据库下 UDF mylowerperm1 的 create 权限。

图3-23 将 udf 测试 jar 包上传到 HDFS

```
[root@sharedev1 ~]# su udfuser03
sh-4.2$ kinit
Password for udfuser03@SHAREDEVTEST.COM:
sh-4.2$ hdfs dfs -mkdir -p /tmp/udflib
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$ hdfs dfs -put /opt/dlhUdfTest-1.0.jar /tmp/udflib/
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
sh-4.2$
```

图3-24 无执行权限

```
0: jdbc:hive2://sharedev2.hde.com:2181,shared> create function dlhtest.mylowerperm1 as 'com.dlh.hive.udf.LowerFunc'
using jar 'hdfs://sharedev2.hde.com:8020/tmp/udflib/dlhUdfTest-1.0.jar';
Error: Error while compiling statement: FAILED: HiveAccessControlException Permission denied: user [udfuser03] does
not have [CREATE] privilege on [dlhtest/mylowerperm1] (state=42000,code=40000)
```

(4) 如图 3-25 所示，修改 HDFS 的/tmp/udflib 目录权限为 777，目的是当其他用户调用该函数时拥有该 jar 包的访问权限。

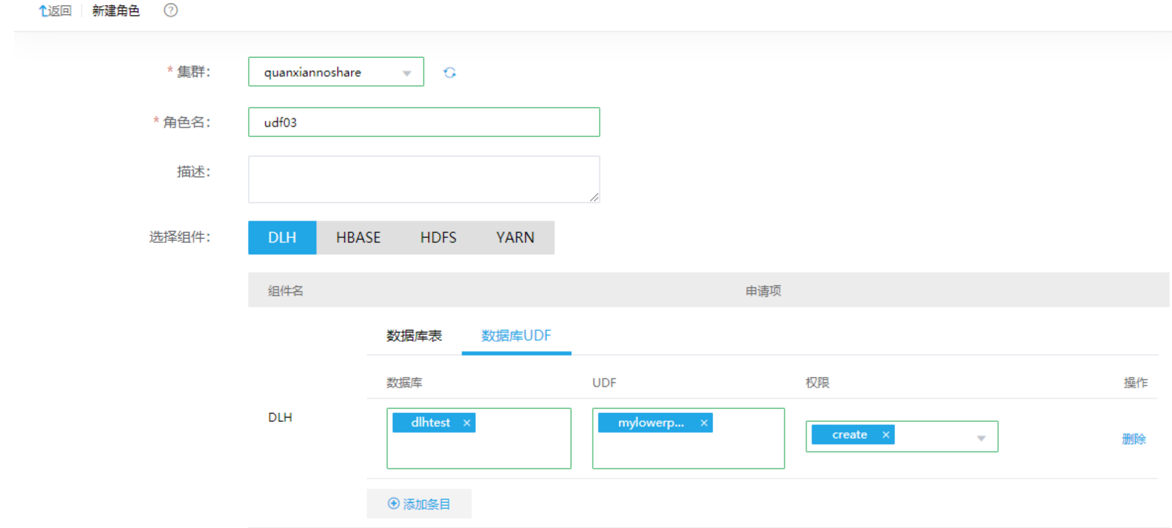
图3-25 修改 HDFS 的/tmp/udflib 目录权限

```
sh-4.2$ hdfs dfs -chmod -R 777 /tmp/udflib
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.0.1.0-187/hadoop/lib/log4j-slf4j-impl-2.12.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

(5) 新建角色

在[集群权限/角色管理]页面，新建 udf03 角色，授权 dlhtest 数据库下 UDF mylowerperm1 的 create 权限。

图3-26 新建角色



(6) 用户绑定角色

在[集群权限/用户管理]页面，单击 udfuser03 用户对应的<修改用户授权>按钮，为其绑定 udf03 角色。

(7) 使用 udfuser03 用户重新执行 dlhtest 数据库下 UDF mylowerperm1 的 create 操作，如图 3-27 和图 3-28 所示，此时显示 create 操作执行成功。

图3-27 执行成功

```
0: jdbc:hive2://sharedev2.hde.com:2181,shared> create function dlhtest.mylowerperm1 as 'com.dlh.hive.udf.LowerFunc' using jar 'hdfs://sharedev1.hde.com:8020/tmp/udflib/dlhUdfTest-1.0.jar';
No rows affected (0.595 seconds)
INFO  : Compiling command(queryId=hive_20211111171000_3f959b1a-2a50-45ce-975d-1efa0994a799): create function dlhtest.mylowerperm1 as 'com.dlh.hive.udf.LowerFunc' using jar 'hdfs://sharedev1.hde.com:8020/tmp/udflib/dlhUdfTest-1.0.jar'
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20211111171000_3f959b1a-2a50-45ce-975d-1efa0994a799); Time taken: 0.123 seconds
INFO  : Executing command(queryId=hive_20211111171000_3f959b1a-2a50-45ce-975d-1efa0994a799): create function dlhtest.mylowerperm1 as 'com.dlh.hive.udf.LowerFunc' using jar 'hdfs://sharedev1.hde.com:8020/tmp/udflib/dlhUdfTest-1.0.jar'
INFO  : Starting task [Stage-0:FUNC] in serial mode
INFO  : Added [/tmp/87193f95-1ac1-4f38-b7d3-0533ef043b13_resources/dlhUdfTest-1.0.jar] to class path
INFO  : Added resources: [hdfs://sharedev1.hde.com:8020/tmp/udflib/dlhUdfTest-1.0.jar]
INFO  : Completed executing command(queryId=hive_20211111171000_3f959b1a-2a50-45ce-975d-1efa0994a799); Time taken: 0.439 seconds
INFO  : OK
```

图3-28 查看创建的内容

```
0: jdbc:hive2://sharedev2.hde.com:2181,shared> select dlhtest.mylowerperm1('abc');
+------+
| _c0  |
+------+
| abc  |
+------+
1 row selected (1.67 seconds)
INFO  : Compiling command(queryId=hive_20211111171207_dabacd01-6736-4236-9eec-060a5747200b): select dlhtest.mylowerperm1('abc')
INFO  : Added [/tmp/90d958e5-5c26-4aa2-bd60-5906448b75af_resources/dlhUdfTest-1.0.jar] to class path
INFO  : Added resources: [hdfs://sharedev1.hde.com:8020/tmp/udflib/dlhUdfTest-1.0.jar]
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name: c0, type:string, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hive_20211111171207_dabacd01-6736-4236-9eec-060a5747200b); Time taken: 1.504 seconds
INFO  : Executing command(queryId=hive_20211111171207_dabacd01-6736-4236-9eec-060a5747200b): select dlhtest.mylowerperm1('abc')
INFO  : Completed executing command(queryId=hive_20211111171207_dabacd01-6736-4236-9eec-060a5747200b); Time taken: 0.012 seconds
INFO  : OK
```

说明

在新的 Session 中执行 show functions 命令时，可能查看不到新建函数，这是因为之前创建 function 时连接的 DLHServer2 和当前连接不同。重启当前连接的 DLHServer2，即可重新读取数据库的函数并加载到缓存，此时重新执行 show functions 命令即可查看到新建函数。

3.8 添加/删除进程

DLH 组件强依赖 Presto、Atlas 和 Hudi 组件，所以进行 DLH 管理时可能访问需要对这些组件执行添加/删除进程操作，其中部分组件的某些进程受约束不能删除，详情请以实际页面为准。

3.8.1 添加进程

1. DLH

DLH 支持添加 DLH Metastore、DLHServer2、DLH Client 进程。

- 在并发查询任务较多的情况下，可以考虑增加 DLH Metastore 和 DLHServer2 进程，分摊查询负载，提高并发数。
- 若主机新扩容机器时未勾选 client，后期可以通过添加 DLH Client 进程在新机器上增加。

2. Presto

Presto 仅支持添加 Worker 进程，可通过执行添加进程操作来实现 Presto 集群扩容，Presto 集群扩容是指在某节点上新增安装 Worker 进程。

随着业务量的增长，集群服务能力无法满足业务需求时，需要考虑对 Presto 集群进行扩容，扩容成功后，Presto 集群的数据查询能力会得到增强。进行扩容时，需注意：

- 扩容节点操作系统版本与集群内部版本需保持一致。
- 扩容操作一旦开始，不支持中止。
- 一旦扩容失败，需要及时将扩容失败的节点剔除。

3. Atlas

Atlas 仅支持添加 Atlas Client 进程。若主机新扩容机器时未勾选 client，后期可以通过添加 Atlas Client 进程在新机器上增加。

4. Hudi

Hudi 仅支持添加 Hudi Client 进程。若主机新扩容机器时未勾选 client，后期可以通过添加 Hudi Client 进程在新机器上增加。

5. 操作示例

说明

- 本章节仅以 DLH 组件添加 DLH Metastore 进程为例进行说明，其它进程操作类似不再进行说明。
- 若集群中所有节点均已安装 DLH Metastore，添加 DLH Metastore 进程前则需要先在集群中添加主机，然后再执行添加 DLH Metastore 进程的操作。如果集群中已有添加进程所需要的主机，则可直接执行添加 DLH Metastore 进程的操作。

添加 DLH Metastore 进程的操作步骤如下：

(1) 在 DLH 组件详情页面，在右上角组件操作的下拉框中选择<添加进程>按钮。

(2) 弹出添加进程窗口，如图 3-29 所示。

   a. 选择进程及主机

   在选择进程项的下拉列表中选择可添加的组件进程，在选择主机项的主机列表中勾选进程安装在哪一个主机上（支持多选）。

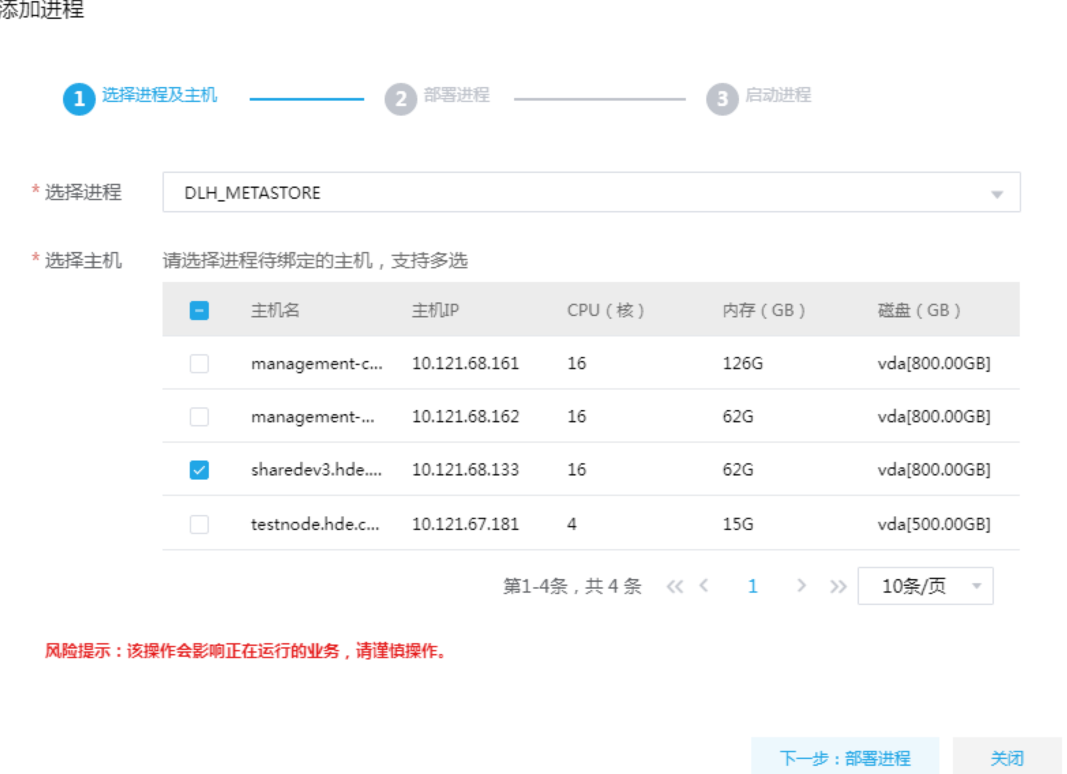
   b. 部署进程

   选择结束后单击下一步部署进程，直至部署进度条结束（不支持中止）。

   c. 启动进程

   部署进程结束后单击下一步启动进程，直至启动进度条结束（不支持中止）。

图3-29 添加进程



(3) 查看进程变化

DLH Metastore 添加完成之后，在组件详情页面[部署拓扑]页签中可以查看 DLH Metastore 进程的数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

3.8.2 删除进程

1. DLH

DLH 支持删除 DLH Metastore、DLHServer2 进程。

- 在并发任务较少的情况下，可以考虑减少 DLH Metastore 和 DLHServer2 进程，节省集群资源。
- 删除进程后 DLH Metastore、DLHServer2 分别对应进程的个数均不能少于 2 个。

2. Presto

Presto 支持删除 Worker 进程，可通过执行删除 Worker 进程操作来实现 Presto 集群缩容，Presto 集群扩容是指将某节点上已安装的 Worker 进程删除。

若集群初始规划的 Worker 节点不合理或 Worker 缩容后也要能满足客户业务需求，此时可根据客户需求规划考虑对 Presto 集群进行缩容，进行缩容时，需注意：

- 缩容前请检查 Worker 是否有运行的任务，若执行缩容操作时，对应缩容节点存在查询任务正在执行中，该缩容节点上的查询任务可能会失败。
- 在生产环境中，缩容不可回退或暂停，请谨慎使用。
- 执行缩容前，请先手动停止缩容节点对应的 Worker 进程。

3. 操作示例

说明

- 删除进程操作不仅可以在组件详情页面的[部署拓扑]页签下执行，也可以在[集群管理/主机管理/主机监控]下的主机详情页面执行。本章节仅以“在 DLH 组件详情页面的[部署拓扑]页签下执行删除 DLH Metastore 进程操作”为例进行说明，其它进程操作类似不再进行说明。
- 执行删除 DLH Metastore 进程操作前，请确认 DLH Metastore 是否有运行的任务，如果有运行的任务时，删除进程会影响任务执行。

删除 DLH Metastore 进程的操作步骤如下：

(1) 在 DLH 组件详情页面[部署拓扑]页签下，选择要删除 DLH Metastore 进程的主机，然后单击该进程右侧操作中的<停止>按钮，停止 DLH Metastore。

(2) 删除 DLH Metastore

待 DLH Metastore 停止成功后，如图 3-30 所示，在该进程右侧操作中单击<删除>按钮，即可完成删除 DLH Metastore。

图3-30 删除进程

(3) 查看进程变化

DLH Metastore 进程删除完成之后，在组件详情页面[部署拓扑]页签中可以查看 DLH Metastore 进程的数量变化以及状态。

(4) 重启组件（根据实际情况选择）

进入集群详情页面，选择[组件]页签，需根据页面提示重新启动相关组件。

4 配置说明

4.1 DLH常用配置

4.1.1 sparrow-default 配置

表4-1 sparrow-defaults 常用配置

参数名称	参数说明	默认值
spark.eventLog.dir	Sparrow事件日志目录	hdfs:///sparrow-history/
spark.eventLog.enabled	Sparrow事件日志启用标志	true
spark.history.fs.logDirectory	Sparrow历史信息日志目录	hdfs:///sparrow-history/
spark.history.ui.port	Sparrow历史信息UI端口	18082
spark.yarn.historyServer.address	HistoryServer UI的地址	无
spark.yarn.applicationMaster.waitTries	YARN模式下application等待Master的次数	10
spark.yarn.driver.memoryOverhead	YARN模式下driver占用的内存	384
spark.yarn.max.executor.failures	YARN模式下executor最大失败次数	3
spark.yarn.submit.file.replication	YARN模式下应用程序上载到HDFS上的最大复制数	3
spark.security.authorization.enable	是否开启权限验证	false

4.1.2 sparrow-thrift-sparkconf 配置

表4-2 sparrow-thrift-sparkconf 常用配置

参数名称	参数说明	默认值
spark.security.authorization.enable	JDBC连接是否开启权限验证	false
spark.master	ThriftServer的模式	当YARN的NodeManager只有一个时，该参数为local[4];否则为yarn-client
spark.yarn.security.tokens.hbase.enabled	安全环境中，ThriftServer YARN模式下是否token认证HBase组件	true
spark.yarn.security.credentials.hbase.enabled	安全环境中，ThriftServer YARN模式下是否使用HBase组件凭证	true
spark.yarn.queue	使用的YARN队列	default
spark.yarn.submit.file.replication	文件上传到HDFS的复本数	3

参数名称	参数说明	默认值
spark.yarn.driver.memoryOverhead	每个driver可以分配的非堆存储内存，这些内存用于如VM，字符串常量池以及其他本地额外开销等,单位：M	384
spark.yarn.executor.memoryOverhead	每个executor可以分配的非堆存储内存，这些内存用于如VM，字符串常量池以及其他本地额外开销等,单位：M	384

4.1.3 sparrow-hive-site-override 配置

表4-3 sparrow-hive-site-override 常用配置

参数名称	参数说明	默认值
hive.server2.thrift.port	JDBC连接的端口号	10017
hive.server2.transport.mode	传输模式，分为http和binary	binary
hive.metastore.client.socket.timeout	socket连接Hive Metastore的超时时长	1800
hive.security.authorization.manager	Metastore授权管理类的名称	org.apache.hadoop.hive.ql.security.authorization.StorageBasedAuthorizationProvider

4.1.4 dlh-site 配置

表4-4 dlh-site 常用配置

参数名称	参数说明	默认值
hive.input.format	输入格式，默认HoodieCombineHiveInputFormat适配hudi表	org.apache.hudi.hadoop.hive.HoodieCombineHiveInputFormat
hive.server2.thrift.port	Dlh组件JDBC连接的端口号	13000
hive.metastore.uris	Dlh组件元数据库对应的地址	-
hive.presto.enable	是否启用presto	true
hive.strict.checks.cartesian.product	是否开启笛卡尔积sql校验	true
hive.flink.enable	是否启用flink	true
hive.fetch.task.conversion	是否将简单的查询简化为直接从文件中读取。none：禁用该功能；minimal：select * ,limit, filter在一个表所属的分区表上操作，直接从文件中读取数据；more：select，filter，limit 都直接从文件中读取数据	more
hive.execution.engine	Dlh组件默认执行引擎	spark
hive.server2.webui.port	Dlh组件UI页面端口	13002

参数名称	参数说明	默认值
hive.presto.table.threshold	表大小默认阈值为10GB，单位为B。当执行引擎为auto时，若表容量小于该阈值则查询走presto执行，若表容量大于该阈值则查询走hive执行	10737418240
hive.presto.http-server.http.port	Presto的协调器节点的地址，比如：http://localhost:18089。若集群开启了HA，可配置为虚拟主机名及代理端口，比如：http://testclr-vserver.vip.hde.com:18090，这部分信息可以从Presto组件的自定义config配置http.proxy.url获取	-
hive.compute.query.using.stats	是否使用元数据统计信息返回count统计值	true

4.2 Presto常用配置

4.2.1 node 配置

表4-5 node 常用配置

参数名称	参数说明	默认值
log_dir	Presto Worker日志目录	/var/de_log/presto
loglevel	Presto Worker日志级别	INFO
catalog_config_dir	Presto catalog配置文件目录	/etc/presto
catalog_share_config_dir	Presto 共享catalog配置文件目录	/apps/presto/etc/catalog

4.2.2 resource-groups-json 配置

表4-6 resource-groups-json 常用配置

参数名称	参数说明	默认值
资源组属性		
name	（必须）：组名称，可以是一个模板	admin
maxQueued	（必须）：排队任务的最大数量，一旦达到这个数量，新的任务将被拒绝	100
hardConcurrencyLimit	（必须）：正在运行的任务的数量	50
softMemoryLimit	（必须）：这个组分布式内存的最大使用量，一旦到达，新任务排队。可以指定为一个绝对值，也可以指定对集群的百分比	100%
jmxExport	（可选）：如果设置为true，组相关的统计指标被吓到JMX以便监控，默认为false	true
选择规则		

参数名称	参数说明	默认值
group	（必须）：这些任务运行的组	admin
全局属性		
cpuQuotaPeriod	（可选）：cpu份额被强制执行的时间	1h

4.2.3 resource-groups-properties 配置

表4-7 resource-groups-properties 常用配置

参数名称	参数说明	默认值
resource_groups_config_file	指定资源组配置文件的绝对路径	/etc/presto/coordinator/resource-groups.json
resource-groups.configuration-manager	指定资源配置组的文件格式	file

4.3 Atlas常用配置

4.3.1 application-properties 配置

表4-8 application-properties 常用配置

参数名称	参数说明	默认值
atlas.rest.address	Atlas服务Rest接口地址	无
atlas.server.http.port	Atlas服务Rest接口服务端口	31000
atlas.notification.topics	Atlas使用Kafka topics	ATLAS_HOOK,ATLAS_ENTITIES
atlas.notification.replicas	Atlas使用Kafka topic的副本数	1
atlas.kafka.zookeeper.connect	Atlas使用Kafka的zookeeper地址	无
atlas.kafka.bootstrap.servers	Atlas使用Kafka的broker地址	无
atlas.graph.index.search.solr.zookeeper-url	Atlas使用Infra-solr的zookeeper地址	无
atlas.graph.index.search.solr.mode	Atlas使用Infra-Solr的集群模式	Cloud
atlas.audit.hbase.tablename	Atlas审计使用HBase的表名	ATLAS_ENTITY_AUDIT_EVENTS
atlas.graph.storage.hbase.table	Atlas图结构数据存储使用的HBase的表名	atlas_janus
atlas.audit.hbase.zookeeper.quorum	Atlas使用HBase的zookeeper地址	无
atlas.graph.storage.hostname	Atlas使用HBase的主机名地址	无

5 常见问题解答

1. DLH 执行简单 count 任务时，如 select count(1) from XXX，获取统计值不准确

- 原因分析：DLH 中默认 hive.compute.query.using.stats=true，为加快查询速度，使用元数据统计信息返回 count 统计值，但该统计信息会由于写表方式不同，导致统计不准确。
- 解决方法：DLH 引擎执行 count 任务前设置参数 set hive.compute.query.using.stats =false，启动 count 统计任务。

2. Presto UI 历史任务无法查看

- 原因分析：由于当前版本 Presto 组件没有持久化任务的功能，导致 Presto 组件重启以后，历史任务无法查看。
- 解决方法：该问题会在后续版本中完善。