弹性 MapReduce

EMR 开发指南

腾讯云

文档目录

EMR 开发指南
Hadoop 开发指南
HDFS 常见操作

最近更新时间：2023-12-14 14:53:12

腾讯云 EMR 的 Hadoop 集成了腾讯云对象存储，如果您在购买的时候勾选了支持 COS，那么您也可以通过常见的 hadoop 命令操作 COS 上的数据。可通过如下命令操作集群中的数据。

```
#cat 数据
hadoop fs -cat /usr/hive/warehouse/hivewithhdfs.db/record/data.txt
#修改目录或者文件权限
hadoop fs -chmod -R 777 /usr
#改变文件或者目录 owner
hadoop fs -chown -R root:root /usr
#创建文件夹
hadoop fs -mkdir <paths>
#本地文件发送到 HDFS 上
hadoop fs -put <localsrc> ... <dst>
#拷贝本地文件到 HDFS 上
hadoop fs -copyFromLocal <localsrc> URI
#查看文件或者目录的存储使用量
hadoop fs -du URI [URI …]
#删除文件
hadoop fs -rm URI [URI …]
#设置目录或者文件的拷贝数
hadoop fs-setrep [-R] [-w] REP PATH [PATH …]
#检查集群文件坏块
hadoop fsck <path> [-move | -delete | -openforwrite] [-files [-blocks [-locations | -racks]]]
```

更多 HDFS 命令请参考 社区文档，此外如果您的集群是 HA 集群（双 namenode），您可以通过如下命名查看哪个 namenode 是 active 的。

```
#nn1 是 namenode 的 ID，一般为 nn1 和 nn2
hdfs haadmin -getServiceState nn1
#查看当前集群报告
hdfs dfsadmin -report
#namenode 离开安全模式
hdfs dfsadmin -safemode leave
```

HDFS 联邦管理开发指南

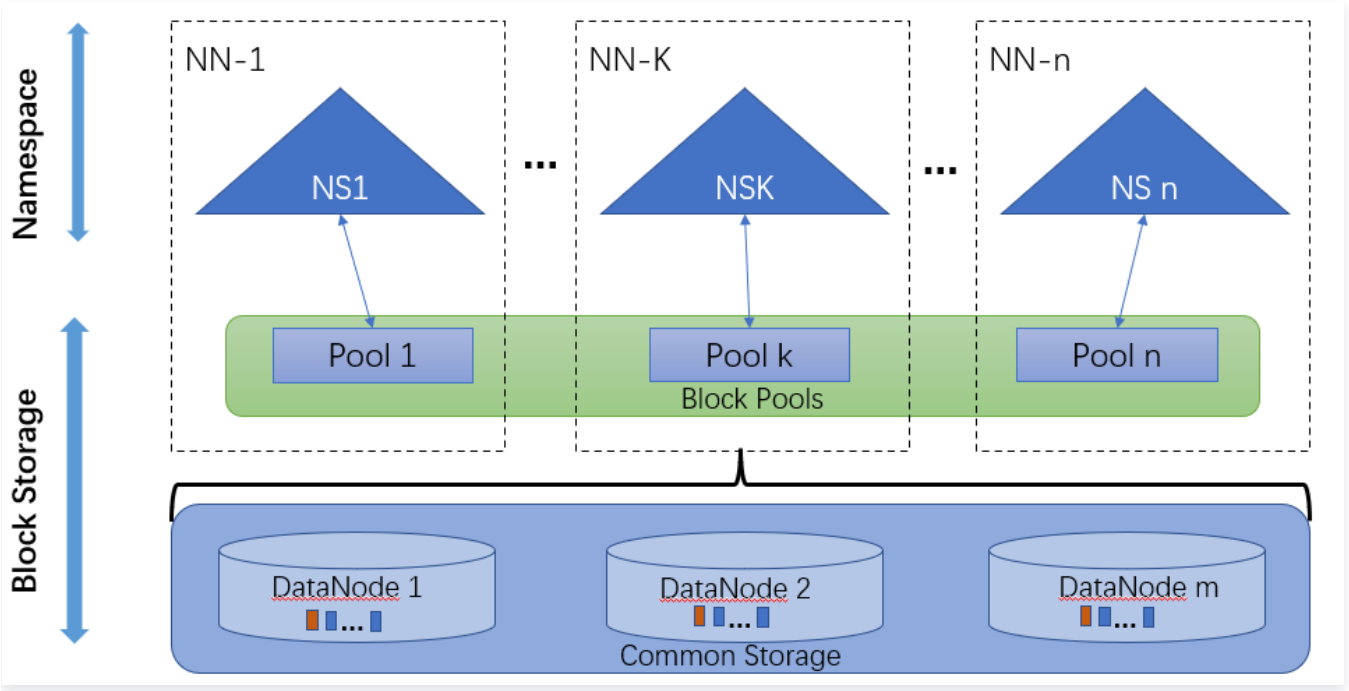
最近更新时间：2023-06-27 15:04:02

HDFS 联邦管理类型选择

> **注意**
> 当前 HDFS 联邦管理为白名单开放，如需要可联系工单开通。

HDFS 联邦管理架构

HDFS Federation（联邦）使用多个独立的 NameNode/Namespace 来使 HDFS 的命名服务能够水平扩展，联邦中的 DataNode 被所有的 NameNode 用作公共存储块的地方；每一个 DataNode 都会向所在集群中所有的 NameNode 注册，并且会周期性地发送心跳和块信息报告，同时处理来自所有 NameNode 的指令。



有关更多 HDFS Federation介绍，请参阅 Hadoop 文档。

ViewFs Federation 原理

为了方便管理多个命名空间，HDFS ViewFs Federation 采用了经典的 Client Side Mount Table（社区 ViewFs 功能）。通过客户端侧将应用的不同路径映射到具体的 NameService 上，从而达成存储分离或者性能分离的场景。HDFS ViewFs Federation 采用 Client-Side Mount Table 分摊文件和负载，该方法更多的需要人工介入（明确的规划）以达到理想的负载均衡。

Router-based Federation 原理

Router-based Federation 提供了一个软件层来管理多个 NameSpace 空间。相较于 ViewFs 通过在客户端维护挂载表信息，Router-based Federation 是真正做到了对客户端的完全透明。因为这部分映射信息将会被额外的保存下来，还会持久化下来。

HDFS 联邦管理配置

主要是要考虑两个方面：所需的 NameService 数量，业务数据目录挂载方式。

NameService 数量的规划原则

1. 单 NameService 的安全存储容量上限为1亿文件，如果超过该数量，该 NameService 的访问速度、读写吞吐量都会严重降低。所以，如果文件存储量（预计）超过1亿文件，则需要新增 NameService。
2. 对于 HDFS 的重度使用（即大量读写 HDFS 中文件）应用，有必要单独划分一个 NameService 来处理其请求，以保证该应用的数据可以独占该 NameService 的所有处理能力，也避免了该应用对其他应用的影响。
3. 对于可靠性要求严苛的应用，可以划分一个 NameService，以免其他应用的过高的读写频率导致该应用无法访问 HDFS，造成该应用业务的不稳定。

业务数据目录挂载方式的规划原则

1. 业务数据相关联的服务的数据目录，尽量挂载在同一 NameService 下面。否则，跨 NameService 的文件读写速率较慢，会降低应用的文件存储性能。
2. 数据量大，且对其他服务的业务无关联的，可以直接使用一个 NameService。
3. 对业务量较小的业务，建议直接将其目录挂载在默认的 NameService（HDFS${clusterid}，HDFS 拼接数字形式 clusterid）中，这样就不用做数据迁移，可以降低配置为 Federation 的复杂性。
4. 建议只对全局一级目录进行 NameService 映射以减少配置复杂度。

方案比较

ViewFs Federation 类型

1. 直接使用 ViewFs
   优点：统一视图，不同的应用有相同的使用方式。
   缺点：ViewFs 挂载表的变更需要所有使用集群的应用同步读取最新的挂载点。
2. 指定 NameService
   优先：不需要同步更改所有应用的配置。
   缺点：不同组件和应用需要明确各自的使用目录，组件路径之间耦合的场景将复杂化。

Router-based Federation 类型配置方式

1. 直接使用 Router-based Federation
   优点：统一视图，不同的应用有相同的使用方式。与 ViewFs 相比，Router-based Federation 挂载表的变更直接生效，不需要使用集群的应用同步更新挂载表。
   缺点：客户端访问时，增加 DFSRouter 转发层，对性能略有影响。
2. 指定 NameService
   优点：不需要同步更改所有应用的配置。
   缺点：不同组件和应用需要明确各自的使用目录，组件路径之间耦合的场景将复杂化。

HDFS 联邦管理

最近更新时间：2025-05-26 14:34:42

功能介绍

HDFS 联邦管理是基于 HDFS Federation 特性提供的 HDFS 联邦集群部署管理能力，包含 NameService 管理以及挂载表管理。在 Hadoop 集群类型 HA 模式下支持联邦管理，支持 ViewFS Federation 和 Router-based Federation 两种联邦类型选择，联邦类型选择后不可更改。Router 节点会用于新扩展的 NameNode 部署，用作 NameNode 部署后的 Router 节点不支持销毁和节点维度所有角色启停。

> **注意**
>
> 1. 当前 HDFS 联邦管理为白名单开放，如需要可 提交工单 联系我们开通。
> 2. EMR 所有产品版本均支持 ViewFs Federation 联邦类型；因 HDFS-2.9.0及以上支持 Router-based Federation 联邦类型，所以 EMR 仅 EMR-V3.x.x 及以上产品版本支持 Router-based Federation 联邦类型, EMR-V2.x.x 系列不支持。
> 3. 在角色管理页中，将联邦节点 NameNode 角色进程暂停时，会影响集群的扩容操作，需恢复 NameNode 角色进程后再执行扩容。

操作步骤

1. 登录 EMR 控制台，在集群列表中单击对应的**集群 ID/名称**进入**集群详情页**。
2. 在集群详情页中单击**集群服务**，然后选择 **HDFS 组件右上角操作>联邦管理**，即可进入联邦管理页面。
3. 单击**添加 NameService** 即可进行 HDFS 联邦新建，需要输入 NameService 名称，选择联邦类型、NameNode 节点、DFSRouter 节点（Router-based Federation 选择）等。

4. 选择**添加联邦节点**
   联邦节点采用集群中的 Router 节点，需先在资源管理页中扩容增加 Router 节点后，再将其设置为联邦节点；NameNode 进程需要选择2个节点，每个节点将会部署 NameNode 进程和 ZKFC 进程。
   第一次新建联邦类型 Router-based Federation 时，部署 DFSRouter 进程需要选择至少2个节点；当再次新建联邦时 DFSRouter 节点可复用，节点数量可大于等于0。
   联邦类型为 Router-based Federation 时，如 DFSRouter 进程负载较高，可进入**资源管理页**扩容 Router 节点并部署 DFSRouter 进程；如 DFSRouter 进程负载较低，可进入**资源管理页**缩容部署 DFSRouter 进程的 Router 联邦节点。

> **注意**
>
> - HDFS-3.3.0以下的版本，在**非首次**新增 NameService 成功后且联邦类型为 Router-based Federation 时，需在角色管理页重启历史的 DFSRouter 进程（为保证业务正常建议在业务低峰期执行）；HDFS-3.3.0版本及以上支持了热加载配置，无需此操作。

- 开启了 Kerberos 的集群，添加联邦 NameService 后，提交到 yarn 的任务如要用到新 NameService 上的文件，需要先重启 yarn 的 ResourceManager（为保证业务正常建议在业务低峰期执行）。
- 设置 NameService 名称后，不可修改也不可以删除且 NameService 名称不能为“nsfed”、“haclusterX”、“ClusterX”等系统关键词。

5. 添加挂载表
当新增 NameService 成功后，才能添加挂载表。为了避免配置复杂度过高，对当前集群目录进行映射，建议仅全局一级目录进行NameService 映射，例如挂载“/tmp”，“/user”，“/srv”等；可批量添加挂载路径。
联邦类型为 Router-based Federation 时，支持全局路径和目标 NameService 的一对多挂载，同时支持通过文本方式批量添加挂载表。
   - 全局路径：在 ViewFs 统一命名空间的路径名称，在 Router-based Federation 统一命名空间的路径名称，也称挂载点。
   - 目标NameService：挂载点映射到真实路径对应的 NameService。
   - 目标路径：在对应 NameService 上的真实路径，与全局路径的名字无需保持一致。

   注意
   - 路径指向：登录 NameNode 节点，执行 hdfs dfs -ls /指向的是这个 NameNode 所管理的 namespace 下的路径，对于 ViewFs 联邦需要用 hdfs dfs -ls viewfs://ClusterX/ 才会指向全局路径，对于 Router-based 联邦需要用 hdfs dfs -ls hdfs://nsfed/ 才会指向全局路径。
   - 登录其它节点，例如充当客户端的Router节点，hdfs dfs -ls / 指向的是全局路径。
   - 各业务组件数据放在一级目录之下，不支持直接放在根目录下访问，根目录不支持挂载。
   - 默认的 NameService 上是有/emr 目录，需要挂载。
   - 使用 Router-based 联邦时，若有多个目标NameService使用同一挂载点，当多个目标NameService存在同目录同文件时，只会获取一个该同名文件。为避免出现此类情况，建议开启联邦后使用hdfs://nsfed来进行统一的读写。

6. 同步挂载表
联邦类型为 Router-based Federation 时，支持同步后台所有挂载表信息到控制台，以便用户进行挂载表的统一管理。

提交 MapReduce 任务

最近更新时间：2023-06-13 10:44:37

本操作手册只描述了命令行模式下基本的 MapReduce 任务操作以及 MapReduce 计算任务如何访问腾讯云对象存储 COS 上面的数据，详细资料可以参考 社区资料 。

- 本次提交的任务为 wordcount 任务即统计单词个数，提前需要在集群中上传需要统计的文件。
- Hadoop 等相关软件路径在 `/usr/local/service/` 下。
- 相关日志路径在 `/data/emr` 下。

1. 开发准备

- 由于任务中需要访问腾讯云对象存储 COS，所以需要在 COS 中先 创建存储桶（Bucket）。
- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群时在基础配置页面勾选了**开启 COS**，并在下方填写自己的 SecretId 和 SecretKey。SecretId 和 SecretKey 可以在 API 密钥管理界面 查看。如果还没有密钥，请单击**新建密钥**建立一个新的密钥。

2. 登录 EMR 服务器

在做相关操作前需要登录到 EMR 集群中的任意一个机器，建议登录到 Master 节点。EMR 是建立在 Linux 操作系统的腾讯云服务器 CVM 上的，所以在命令行模式下使用 EMR 需要登录 CVM 服务器。
创建 EMR 集群后，在控制台中选择弹性 MapReduce。在**集群资源 > 资源管理**中单击 **Master 节点**，选择 Master 节点的**资源 ID**，即可进入云服务器控制台并且找到 EMR 对应的云服务器。
登录 CVM 的方法可参见 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的**登录**，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。

输入正确后，即可进入 EMR 集群的命令行界面。所有的 Hadoop 操作都在 Hadoop 用户下，登录 EMR 节点后默认在 root 用户，需要切换到 Hadoop 用户。使用如下命令切换用户，并且进入 Hadoop 文件夹下：

```
[root@172 ~]# su hadoop
[hadoop@172 root]$ cd /usr/local/service/hadoop
[hadoop@172 hadoop]$
```

3. 数据准备

您需要准备统计的文本文件。分为两种方式：**数据存储在 HDFS 集群**和**数据存储在 COS**。
首先要把本地的数据上传到云服务器。可以使用 scp 或者 sftp 服务来把本地文件上传到 EMR 集群的云服务器中。在本地命令行使用：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，$localfile 是您的本地文件的路径加名称；root 为 CVM 服务器用户名；公网 IP 地址可以在 EMR 控制台的节点信息中或者在云服务器控制台查看；$remotefolder 是您要存放文件的 CVM 服务器路径。
上传成功后，在 EMR 集群命令行中即可查看对应文件夹下是否有相应文件。

```
[hadoop@172 hadoop]$ ls -l
```

数据存放在 HDFS

将数据上传到腾讯云服务器之后，可以把数据拷贝到 HDFS 集群。这里使用 `/usr/local/service/hadoop` 目录下的 README.txt 文本文件作为说明。通过如下指令把文件拷贝到 Hadoop 集群：

```
[hadoop@172 hadoop]$ hadoop fs -put README.txt /user/hadoop/
```

拷贝完成后使用如下指令查看拷贝好的文件：

```
[hadoop@172 hadoop]$ hadoop fs -ls /user/hadoop
输出：
-rw-r--r-- 3 hadoop supergroup 1366 2018-06-28 11:39 /user/hadoop/README.txt
```

如果 Hadoop下面没有 `/user/hadoop` 文件夹，用户可以自己创建，指令如下：

```
[hadoop@172 hadoop]$ hadoop fs -mkdir /user/hadoop
```

更多 Hadoop 指令见 HDFS 常见操作 。

数据存放在 COS

数据存放在 COS 中有两种方式：**在本地通过 COS 的控制台上传和在 EMR 集群通过 Hadoop 命令上传。**

- 在本地通过 COS 控制台直接上传，如果数据文件已经在 COS 可以通过如下命令查看：

  ```
  [hadoop@10 hadoop]$ hadoop fs -ls cosn://$bucketname/README.txt
  -rw-rw-rw- 1 hadoop hadoop 1366 2017-03-15 19:09 cosn://$bucketname /README.txt
  ```

  其中 $bucketname 替换成您的储存桶的名字和路径。
- 在 EMR 集群通过 Hadoop 命令上传，指令如下：

  ```
  [hadoop@10 hadoop]$ hadoop fs -put README.txt cosn://$bucketname/
  [hadoop@10 hadoop]$ bin/hadoop fs -ls cosn://$bucketname/README.txt
  -rw-rw-rw- 1 hadoop hadoop 1366 2017-03-15 19:09 cosn://$bucketname/README.txt
  ```

4. 通过 MapReduce 提交任务

本次提交的任务是 Hadoop 集群自带的例程 wordcount，wordcount 已被压缩为 jar 包上传到了创建好的 Hadoop 中，用户可以直接调来使用。示例使用 hadoop 3.2.2 版本，其他版本示例代码中需使用对应版本的 hadoop-mapreduce-examples 包， 可以在 hadoop 目录下./share/hadoop/mapreduce/路径中查看实际存在的版本。

统计 HDFS 中的文本文件

进入 `/usr/local/service/hadoop` 目录，和数据准备中一样。通过如下命令来提交任务：

```
[hadoop@10 hadoop]$ bin/yarn jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar wordcount
/user/hadoop/README.txt /user/hadoop/output
```

> ⚠ **注意**
> 以上整个命令为一条完整的指令， `/user/hadoop/README.txt` 为输入的待处理文件， `/user/hadoop/output` 为输出文件夹，在提交命令之前要保证 output 文件夹尚未创建，否则提交会出错。

执行完成后，通过如下命令查看执行输出文件：

```
[hadoop@10 hadoop]$ bin/hadoop fs -ls /user/hadoop/output
Found 2 items
```

```
-rw-r--r-- 3 hadoop supergroup 0 2017-03-15 19:52 /user/hadoop/output/_SUCCESS
-rw-r--r-- 3 hadoop supergroup 1306 2017-03-15 19:52 /user/hadoop/output/part-r-00000
```

通过如下指令查看 part-r-00000 中的统计结果：

```
[hadoop@10 hadoop]$ bin/hadoop fs -cat /user/hadoop/output/part-r-00000
(BIS),  1
(ECCN)  1
(TSU)   1
(see    1
5D002.C.1,  1
740.13) 1
<http://www.wassenaar.org/> 1
……
```

统计 COS 中的文本文件

进入 `/usr/local/service/hadoop` 目录，通过如下命令来提交任务：

```
[hadoop@10 hadoop]$ bin/yarn jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar  wordcount
cosn://$bucketname/README.txt /user/hadoop/output
```

命令的输入文件改为了 `cosn:// $bucketname /README.txt` ，即处理 COS 中的文件，其中 $bucketname 为您的存储桶的名字加路径。依然输出到 HDFS 集群中，也可以选择输出到 COS 中。查看输出的方法和上文一样。

查看任务日志

```
#查看任务状态
bin/mapred job -status jobid
#查看任务日志
yarn logs -applicationId id
```

自动扩容 Task 节点不分配 ApplicationMaster

最近更新时间：2023-06-27 15:04:46

功能介绍

在自动扩缩容场景下，缩容规则触发时，即将被销毁的 Task 节点若正在运行着 ApplicationMaster（AM），那么节点在销毁的同时，运行的 AM 也会随之销毁，会导致当前 Job 整体失败。
在自动扩缩容场景下，通过扩容添加的 Task 节点默认不分配 ApplicationMaster，保证缩容Task节点时，运行的 ApplicationMaster不被销毁，Job 可以正常进行。

功能特性

通过对 YARN 的源码改造以及对自动扩容流程增加配置项，实现自动扩容添加的 Task 节点将不会被分配 AM，AM 将会全部分配到非自动扩容的 Task 节点，自动扩容添加的 Task 节点仅承担计算任务；当对 Task 节点进行缩容时，只有正在运行的计算任务的节点会被销毁，但对应 AM 仍然保持存活状态，AM 可以重新在其他节点重试相关的计算任务，当前整个 Job 可以继续进行下去。

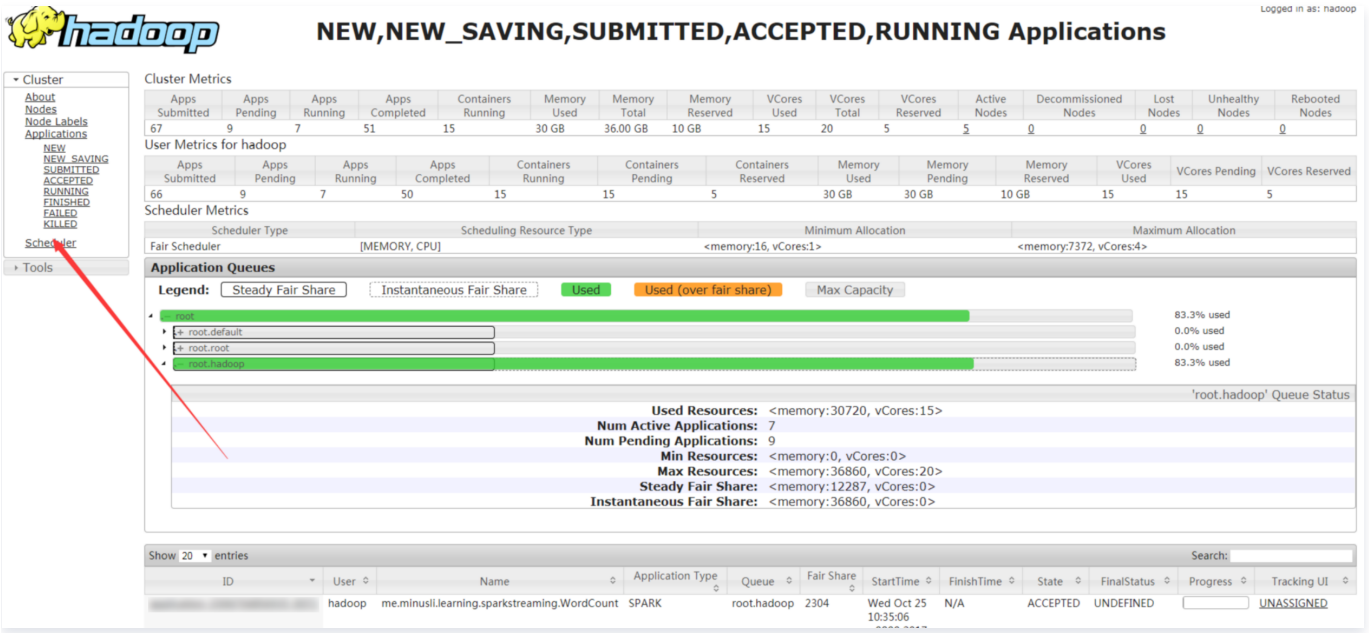
适用范围

仅对自动扩容的 Task 节点生效，手动扩容不生效。

YARN 任务队列管理

最近更新时间：2023-11-14 17:04:31

您可以登录 YARN 自带的 YARN WebUI，登录方式为通过 EMR 提供的快捷入口，查看快捷入口请参考 软件 WebUI 入口 。登录后单击下图所示左侧列表中 **Scheduler** ，即可看到任务队列信息。



从图中可以看到整个集群的一些监控信息：

- 应用信息：9个等待，7个执行，51个完成，总计67个；其中有15个 container 正在执行。
- 内存信息：总共36G，使用了30G，保留10G。
- 虚拟核信息：总共20个，使用了15个，保留5个。
- 节点信息：可用节点5个，退服0个，丢失节点0个，不健康节点0个，重启节点0个。
- 调度器信息：使用 Fair Scheduler（公平调度器），最大最小的内存和 CPU 分配信息。

其中 Application Queues 栏包含了集群的任务队列信息（以 root.hadoop 队列为例）：

- 使用的资源：内存30720M（30G），虚拟核15个。
- 正在执行的应用7个。
- 等待的应用9个。
- 占用最小资源：0M，0核。
- 占用最大资源：36860M，20核。
- 队列资源的稳定的公共共享。
- 队列的瞬时公平共享资源。
- 该队列占用资源的比例为83.3%。

YARN 标签调度实践

最近更新时间：2023-12-15 16:51:22

功能介绍

Spark on Yarn 使用 Yarn 作为资源调度器，在 Hadoop2.7.2 之后 Yarn 在容器调度器 Capacity Scheduler 基础上增加了标签调度。
Capacity Scheduler 是一个多租户资源调度器，其核心思想是 Hadoop 集群中的可用资源由多个组织共享，这些组织根据自己的计算需求共同为集群提供计算能力。Capacity Scheduler 具有分层队列、容量保证、安全性、弹性资源、多租户、基于资源的调度、映射等特性。集群的所有资源全部分配给多个队列，提交到队列的所有应用程序都可以使用分配给队列的资源，其他组织未使用的空闲资源可以非抢占式地分配给运行低于容量的队列上的应用程序，确保应用程序获得资源容量下限的同时能弹性的满足不同应用程序的资源需求。
Capacity Scheduler 将集群资源粗略地分配给不同的队列，不能指定队列中应用程序的运行位置。标签调度在 Capacity Scheduler 之上通过给集群各节点打上不同的 Node Label 进行更细粒度的资源划分，应用程序可以指定运行的位置。节点标签具有以下特点：

1. 一个节点只有一个节点标签（也即属于一个节点分区），集群按节点分区被划分为多个不相交的子集群。
2. 根据匹配策略，节点分区具有两种类型：独占和非独占。独占的节点分区将容器分配给完全匹配节点分区的节点；非独占的节点分区将空闲资源共享给请求 default 分区的容器。
3. 每个队列通过设置可访问的节点标签来指定应用程序运行的节点分区。
4. 可以设置每个节点分区内不同队列的资源占比。
5. 在资源请求中指定所需的节点标签，仅在节点具有相同标签时才分配。如果未指定节点标签要求（可通过配置项修改队列的默认标签名），则仅在属于 default 分区的节点上分配此类资源请求。

配置说明

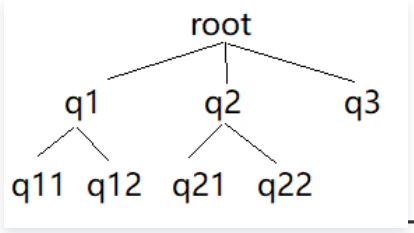
1. 设置 ResourceManager 启用 Capacity Scheduler

标签调度无法单独使用，只能配合 Capacity Scheduler 使用。Yarn 使用 Capacity Scheduler 作为默认调度器，如果您现在正使用其他调度器，请先启动 Capacity Scheduler。在 `${HADOOP_HOME}/etc/hadoop/yarn-site.xml` 中设置：

```
<property>
    <name>yarn.resourcemanager.scheduler.class</name>
    <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>
</property>
```

2. 配置 Capacity Scheduler 参数

在 `${HADOOP_HOME}/etc/hadoop/capacity-scheduler.xml` 中设置 `yarn.scheduler.capacity.root` 是 Capacity Scheduler 预定义的根队列，其他队列均为根队列的子队列。所有队列以树的形式组织。`yarn.scheduler.capacity.<queue-path>.queues` 用于设置 queue-path 路径下的子队列，使用逗号分隔。

示例： 

上图结构配置如下：

```
<property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>q1,q2,q3</value>
</property>
<property>
    <name>yarn.scheduler.capacity.root.q1.queues</name>
    <value>q11,q12</value>
</property>
<property>
    <name>yarn.scheduler.capacity.root.q2.queues</name>
    <value>q21,q22</value>
</property>
```

其他 Capacity Scheduler 配置请查询文档。

3. 设置 ResourceManager 启用 Node Label

在 `yarn-site.xml` 中设置。

```
<property>
    <name>yarn.node-labels.fs-store.root-dir</name>
    <value>hdfs://namenode:port/path-to-store/node-labels/</value>
</property>
<property>
    <name>yarn.node-labels.enabled</name>
    <value>true</value>
</property>
<property>
    <name>yarn.node-labels.configuration-type</name>
    <value>centralized or delegated-centralized or distributed</value>
</property>
```

注意

1. 确保已创建 `yarn.node-labels.fs-store.root-dir`，且 ResourceManager 有权访问。
2. 若将节点标签保存在 RM 的本地文件系统，可使用 `file://home/yarn/node-label` 等路径。但是为了保证集群的高可用，避免 RM 宕机而丢失标签信息，建议将标签信息保存在 HDFS 上。
3. 在 hadoop2.8.2 下需要配置 `yarn.node-labels.configuration-type` 配置项。

4. 配置 Node Label

在 `etc/hadoop/capacity-scheduler.xml` 中设置。

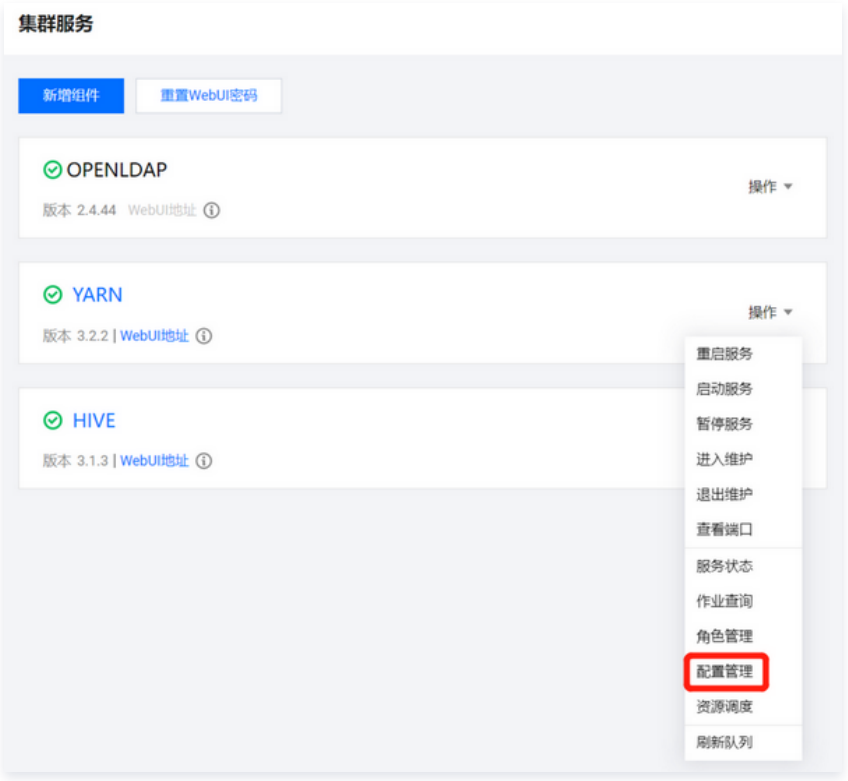
配置项	描述
yarn.scheduler.capacity.`<queue-path>`.capacity	设置队列可以访问属于 DEFAULT 分区的节点的百分比。**每个 parent 队列下直接子队列的 DEFAULT 容量总和必须等于100。**
yarn.scheduler.capacity.`<queue-path>`.accessible-node-labels	设置队列可以访问的特定标签列表，用逗号分隔，如“HBASE，STORM”意味着队列可以访问标签 HBASE 和 STORM。所有队列都可以在没有标签的情况下访问节点，如果不指定此字段，则将从其父字段继承。如果用户想限制队列仅访问没有标签的节点，该字段只需留空即可。
yarn.scheduler.capacity.`<queue-path>`.accessible-node-labels.`<label>`.capacity	设置队列可以访问属于 `<label>` 分区的节点百分比。**每个 parent 队列下直接子队列的 `<label>` 容量总和必须等于100，默认为0。**
yarn.scheduler.capacity.`<queue-path>`.accessible-node-labels.`<label>`.maximum-capacity	与 Capacity Scheduler 配置项 yarn.scheduler.capacity.`<queue-path>`.maximum-capacity 类似，它指定了 `<queue-path>` 在 `<label>` 分区的最大容量，默认为100。
yarn.scheduler.capacity.`<queue-path>`.default-node-label-expression	当资源请求未指定节点标签时，应用将被提交到该值对应的分区。默认情况下，该值为空，即应用程序将被分配没有标签的节点上的容器。

应用示例

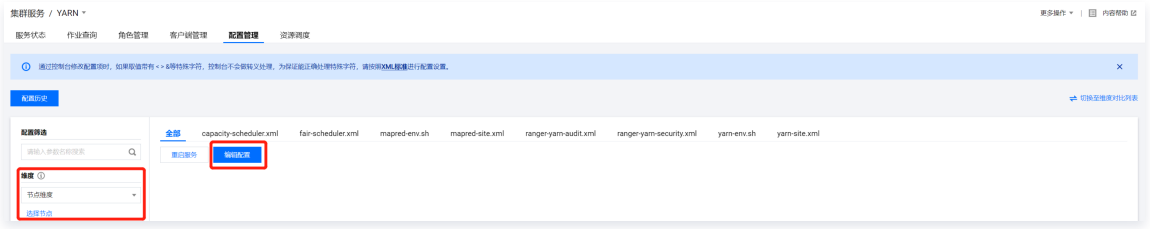
准备工作

1. 准备集群
   确认您已经开通了腾讯云，并且创建了一个 EMR 集群。
2. 检查 YARN 组件配置
   在“集群服务”页面中，选择 YARN 组件进入组件管理界面，然后切换至配置管理标签页，修改 `yarn-site.xml` 中相关参数，保存并重启所有 YARN 组件。
   在角色管理标签页确认 ResourceManager 服务所在节点 IP，之后切换至配置管理标签页修改 `yarn-site.xml` 中相关参数，保存并重启所有 YARN 组件。

- 在集群列表中单击集群实例 ID，进入集群信息页面，然后单击左侧菜单栏**集群服务**，选择 YARN 组件管理中**操作 > 配置管理**。



- 确认 RM 的 IP 地址。
- 在 YARN 组件“配置管理”页面，选择**维度**为节点维度，选择节点为 RM 的 IP 地址，单击**编辑配置**修改 RM 所在节点 `yarn-site.xml` 的 `yarn.resourcemanager.scheduler.class` 参数。



在 Capacity-Scheduler.xml 中配置 Node Label 与队列的映射关系和占比

1. 创建存储节点标签的 HDFS 目录。

```
[root@172 ~]# cd /usr/local/service/hadoop/
[root@172 hadoop]# su hadoop
[hadoop@172 hadoop]$ hadoop fs -mkdir -p /yarn/node-labels
```

2. 在 `core-site.xml` 中获取 NN 的 IP 和 Port。

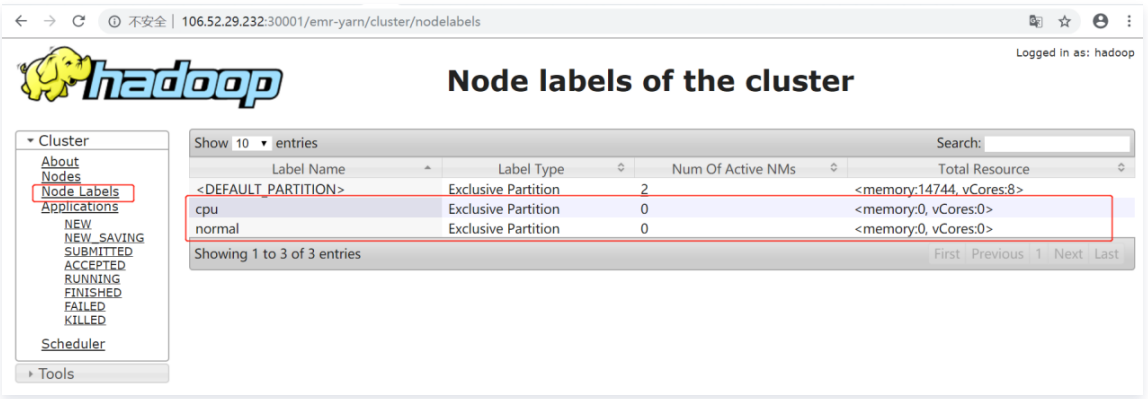
```
<property>
        <name>fs.defaultFS</name>
        <value>hdfs://172.16.32.47:4007</value>
</property>
```

3. 在 master 节点 `yarn-site.xml` 中新建配置项后，重启 ResourceManager。

4. 使用 `yarn rmadmin -addToClusterNodeLabels` 命令新增标签。

```
[hadoop@172 hadoop]$ yarn cluster --list-node-labels
Node Labels:
[hadoop@172 hadoop]$ yarn rmadmin -addToClusterNodeLabels "normal,cpu"
[hadoop@172 hadoop]$ yarn cluster --list-node-labels
Node Labels: <normal:exclusivity=true>,<cpu:exclusivity=true>
```

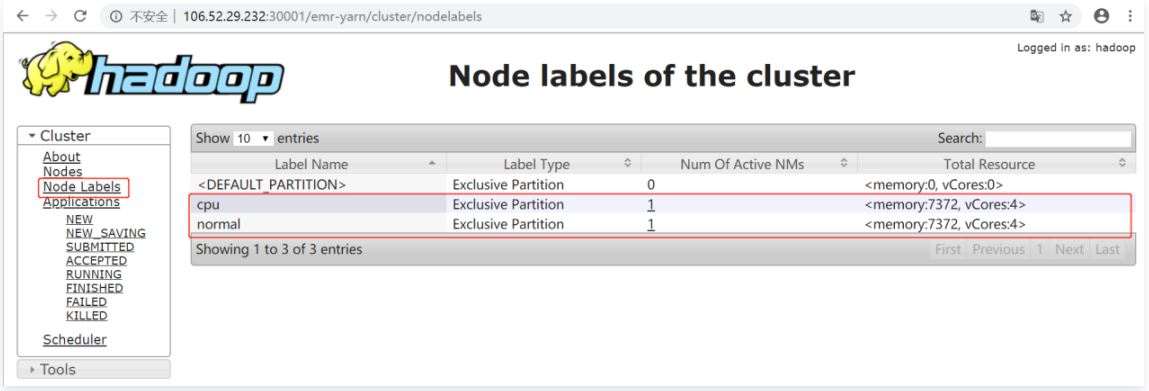
打开 YARN 组件的 WebUI 界面，在 NodeLabels 面板中可以看到集群所有的标签。



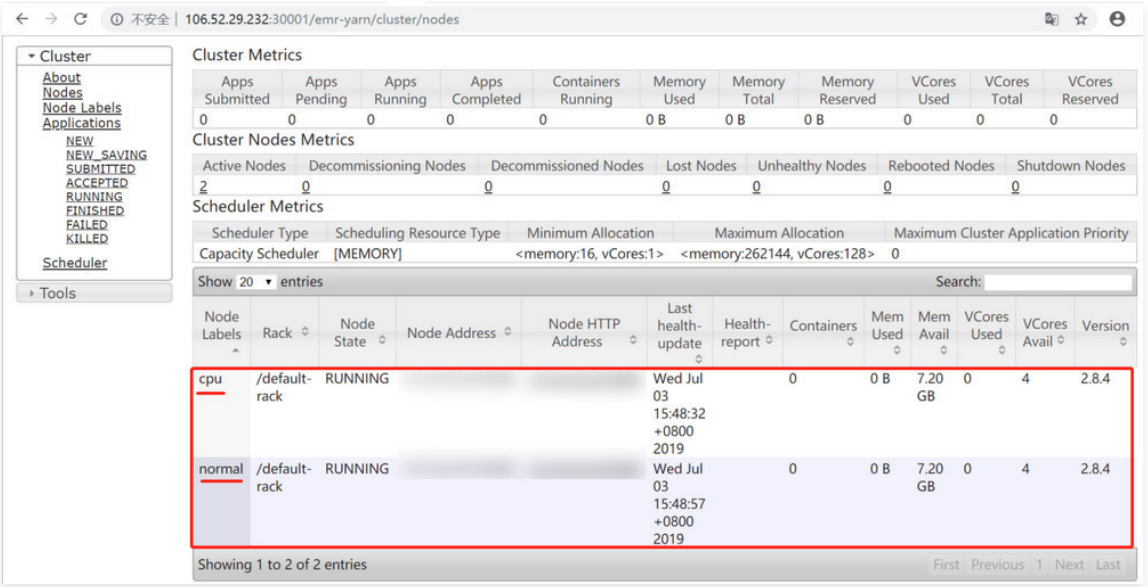
5. 使用 `yarn rmadmin -replaceLabelsOnNode` 命令给节点打标签。

```
[hadoop@172 hadoop]$ yarn rmadmin -replaceLabelsOnNode "172.16.32.43=normal"
[hadoop@172 hadoop]$ yarn rmadmin -replaceLabelsOnNode "172.16.32.25=cpu"
```

在 NodeLabels 面板中可以看到 normal、cpu 分区的节点个数从0变为1。



在 Scheduler 面板中可以看到，测试系统的两个节点对应的标签已经发生改变。

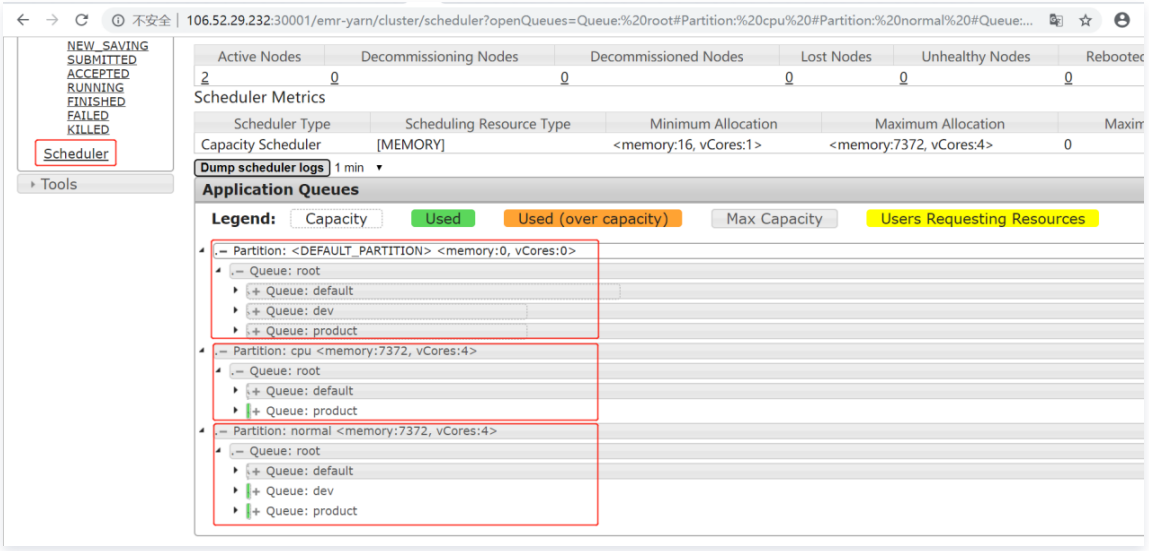


6. 编辑 `capacity-scheduler.xml` 中的配置项，配置集群队列、队列的资源占比和队列的可访问标签。示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration><property>
 <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
 <value>0.8</value>
</property>
<property>
 <name>yarn.scheduler.capacity.maximum-applications</name>
 <value>1000</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.queues</name>
 <value>default,dev,product</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.default.capacity</name>
 <value>20</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.dev.capacity</name>
 <value>40</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.product.capacity</name>
 <value>40</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.accessible-node-labels.cpu.capacity</name>
 <value>100</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.accessible-node-labels.normal.capacity</name>
 <value>100</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.accessible-node-labels</name>
 <value>*</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.dev.accessible-node-labels.normal.capacity</name>
 <value>100</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.product.accessible-node-labels.cpu.capacity</name>
 <value>100</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.dev.accessible-node-labels</name>
 <value>normal</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.dev.default-node-label-expression</name>
 <value>normal</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.product.accessible-node-labels</name>
 <value>cpu</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.product.default-node-label-expression</name>
```

```
  <value>cpu</value>
</property>
<property>
 <name>yarn.scheduler.capacity.normal.sharable-partitions</name>
 <value>cpu</value>
</property>
<property>
 <name>yarn.scheduler.capacity.normal.require-other-partition-resource</name>
 <value>true</value>
</property>
<property>
 <name>yarn.scheduler.capacity.cpu.sharable-partitions</name>
 <value></value>
</property>
<property>
 <name>yarn.scheduler.capacity.cpu.require-other-partition-resource</name>
 <value>true</value>
</property>
</configuration>
```

在 Scheduler 面板中可以看到，测试集群的3个分区、分区资源分配情况、包含队列情况。在 Application Queues 面板中，共有3个分区：default、normal、cpu，其中 default 分区是默认分区，normal 分区是由带有 normal 标签的节点组成的分区、cpu 分区是由带有 cpu 标签的节点组成的分区。在测试环境中，共有两个节点，这两个节点分别被标记为 normal 和 cpu。单击分区左侧的 + 号，能够展开该分区中包含的队列。

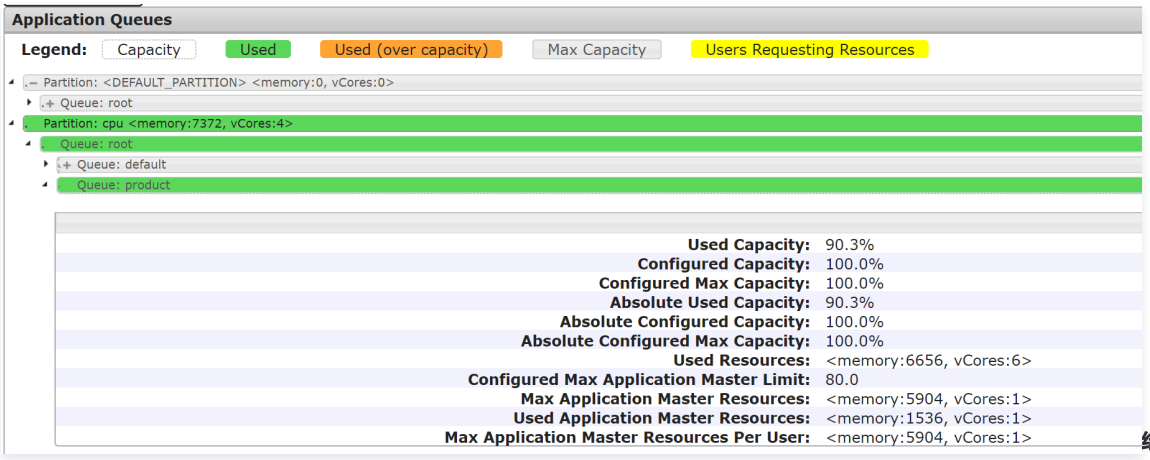


验证标签调度

- **测试一：将任务提交到 product 队列**

```
[hadoop@172 hadoop]$ cd /usr/local/service/hadoop
[hadoop@172 hadoop]$ yarn jar ./share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.8.4-tests.jar
sleep -Dmapreduce.job.queuename=product -m 32 -mt 1000
```

任务提交后各分区的队列资源占用情况如下图所示：

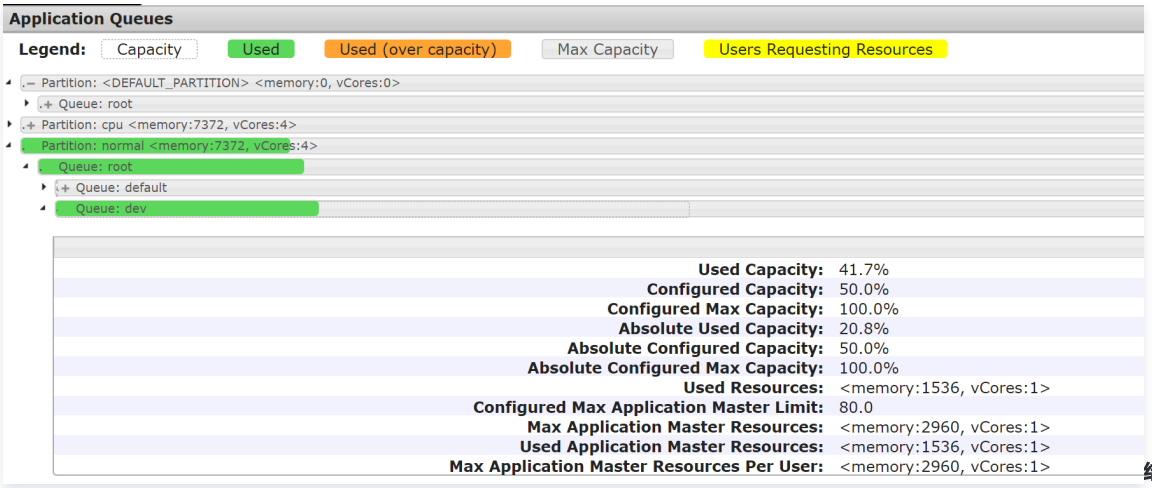


结论：product 队列与 cpu 标签存在映射且默认使用的标签为 cpu，提交到 product 队列的任务运行在标记了 cpu 的节点上。

- **测试二：将任务提交到 dev 队列**

```
[hadoop@172 hadoop]$ cd /usr/local/service/hadoop
[hadoop@172 hadoop]$ yarn jar ./share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.8.4-tests.jar
sleep -Dmapreduce.job.queuename=dev -m 32 -mt 1000
```

任务提交后各分区的队列资源占用情况如下图所示：



结论：dev 队列与 normal 标签存在映射且默认使用的标签为 normal，提交到 dev 队列的任务运行在标记了 normal 的节点上。

Hadoop 实践教程

最近更新时间：2021-07-02 11:08:46

Hadoop 部分包含了分布式文件系统 HDFS、资源调度框架 YARN 以及迭代式计算框架 MR，腾讯的 Hadoop 版本集成了腾讯云对象存储，让您以 `hadoop fs` 命令行的方式使用对象存储，从而实现计算存储分离，这里的最佳实践包含如下内容。

- HDFS无论您是 HA 集群还是非 HA 集群，**请务必记住不能格式化 namenode**，否则会造成数据丢失，如果是您格式化 namenode 造成的数据丢失腾讯云不承担任何责任。
- YARN腾讯云默认开启的是公平调度，您可以根据您的实际需要修改调度器。

使用 API 分析 HDFS/COS 上的数据

最近更新时间：2023-11-16 14:18:31

学习 MapReduce 会接触的第一个程序通常都是 WordCount，统计给定文件的单词的词频。本节将会介绍如何自己建立一个工程并编写程序，并且使用编译打包好的程序去统计 HDFS 和腾讯云对象存储 COS 上面的数据，使用的程序基本和 Hadoop 社区的示例程序相同。

1. 开发准备

- 由于任务中需要访问腾讯云对象存储（COS），所以需要在 COS 中先 创建一个存储桶（Bucket）。
- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要选择包含 HDFS 的集群类型，并在基础配置页面开启对象存储的授权。

2. 登录 EMR 服务器

在做相关操作前需要登录到 EMR 集群中的任意一个机器，最好是登录到 Master 节点。EMR 是建立在 Linux 操作系统的腾讯云服务器 CVM 上的，所以在命令行模式下使用 EMR 需要登录 CVM 服务器。
创建 EMR 集群后，在控制台中选择弹性 MapReduce。在**集群资源 > 资源管理**中单击 **Master 节点**，选择 Master 节点的资源 ID，即可进入云服务器控制台并且找到 EMR 对应的云服务器。
登录 CVM 的方法请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的**登录**，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。
输入正确后，即可进入 EMR 集群的命令行界面。所有的 Hadoop 操作都在 Hadoop 用户下，登录 EMR 节点后默认在 root 用户，需要切换到 Hadoop 用户。使用如下命令切换用户，并且进入 Hadoop 文件夹下：

```
[root@172 ~]# su hadoop
[hadoop@172 root]$ cd /usr/local/service/hadoop
[hadoop@172 hadoop]$
```

3. 数据准备

您需要准备统计的文本文件。分为两种方式：**将数据存储在 HDFS 集群**和**数据存储在 COS**。
首先在本地新建一个 txt 文件 test.txt，并且添加一些英语单词：

```
Hello World.
this is a message.
this is another message.
Hello world, how are you?
```

把本地的数据上传到云服务器。可以使用 scp 或者 sftp 服务来把本地文件上传到 EMR 集群的云服务器中。在本地 shell 使用：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，$localfile 是您的本地文件的路径加名称；root 为 CVM 服务器用户名；公网 IP 地址可以在 EMR 控制台的节点信息中或者在云服务器控制台查看；$remotefolder 是您想存放文件的 CVM 服务器路径。
上传完成后，在 EMR 集群命令行中即可查看对应文件夹下是否有相应文件，此处上传到了 EMR 集群的 `/usr/local/service/hadoop` 路径下。

```
[hadoop@172 hadoop]$ ls -l
```

数据存放在 HDFS

将数据上传到腾讯云服务器后，可以把数据拷贝到 HDFS 集群。通过如下指令把文件拷贝到 Hadoop 集群：

```
[hadoop@172 hadoop]$ hadoop fs -put /usr/local/service/hadoop/test.txt /user/hadoop/
```

拷贝完成后，使用以下指令查看拷贝好的文件：

```
[hadoop@172 hadoop]$ hadoop fs -ls /user/hadoop
输出：
-rw-r--r-- 3 hadoop supergroup 85 2018-07-06 11:18 /user/hadoop/test.txt
```

如果 Hadoop 下面没有 `/user/hadoop` 文件夹，用户可以自己创建，指令如下：

```
[hadoop@172 hadoop]$ hadoop fs -mkdir /user/hadoop
```

更多 hadoop 指令见 HDFS 常见操作 。

数据存放在 COS

数据存放在 COS 中有两种方式：**从本地直接通过 COS 的控制台上传和通过 Hadoop 命令上传。**

- 从本地直接通过 COS 控制台直接上传，数据文件上传后，可通过如下命令查看：

```
[hadoop@10 hadoop]$ hadoop fs -ls cosn://$bucketname/test.txt
-rw-rw-rw- 1 hadoop hadoop 1366 2017-03-15 19:09 cosn://$bucketname/test.txt
```

其中 $bucketname 替换成您的储存桶的名字加路径。

- 通过 Hadoop 命令上传，指令如下：

```
[hadoop@10 hadoop]$ hadoop fs -put test.txt cosn://$bucketname/
[hadoop@10 hadoop]$ hadoop fs -ls cosn://$bucketname/test.txt
-rw-rw-rw- 1 hadoop hadoop 1366 2017-03-15 19:09 cosn://$bucketname/test.txt
```

4. 使用 Maven 创建工程

推荐使用 Maven 来管理工程。Maven 是一个项目管理工具，可以方便地管理项目的依赖信息，即它可通过 pom.xml 文件的配置获取 jar 包，而不用手动去添加。
首先下载并安装 Maven，配置好 Maven 的环境变量，如果您使用 IDE，请在 IDE 中设置好 Maven 相关配置。

新建一个 Maven工程

在命令行下进入您想要新建工程的目录，例如 `D://mavenWorkplace` 中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=$yourgroupID -DartifactId=$yourartifactID
-DarchetypeArtifactId=maven-archetype-quickstart
```

其中 `$yourgroupID` 即为您的包名； `$yourartifactID` 为您的项目名称； `maven-archetype-quickstart` 表示创建一个 Maven Java 项目，工程创建过程中需要下载一些文件，请保持网络通畅。
创建成功后，在 `D://mavenWorkplace` 目录下就会生成一个名为 `$yourartifactID` 的工程文件夹。其中的文件结构如下所示：

```
simple
    ---pom.xml          核心配置，项目根下
    ---src
        ---main
            ---java          Java 源码目录
            ---resources     Java 配置文件目录
        ---test
            ---java          测试源码目录
            ---resources     测试配置目录
```

主要关注 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。
首先在 pom.xml 中添加 Maven 依赖：

```
<dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
```

```
            <artifactId>hadoop-mapreduce-client-core</artifactId>
            <version>2.7.3</version>
        </dependency>
</dependencies>
```

继续在 pom.xml 中添加打包和编译插件：

```
<build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>utf-8</encoding>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
</build>
```

在 src>main>java 下右键新建一个 Java Class，输入 Class 名，这里使用 WordCount，在 Class 添加样例代码：

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

import java.io.IOException;
import java.util.StringTokenizer;

/**
 * Created by tencent on 2018/7/6.
 */
public class WordCount {
    public static class TokenizerMapper
            extends Mapper<Object, Text, Text, IntWritable>
    {
```

```
        private static final IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Mapper<Object, Text, Text, IntWritable>.Context context)
                throws IOException, InterruptedException
        {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens())
            {
                this.word.set(itr.nextToken());
                context.write(this.word, one);
            }
        }
    }

    public static class IntSumReducer
            extends Reducer<Text, IntWritable, Text, IntWritable>
    {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Reducer<Text, IntWritable, Text,
IntWritable>.Context context)
                throws IOException, InterruptedException
        {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            this.result.set(sum);
            context.write(key, this.result);
        }
    }

    public static void main(String[] args)
            throws Exception
    {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length < 2)
        {
            System.err.println("Usage: wordcount <in> [<in>...] <out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        for (int i = 0; i < otherArgs.length - 1; i++) {
            FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
        }
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[(otherArgs.length - 1)]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

可以看到其中有一个 Map 函数和一个 Reduce 函数。

如果您的 Maven 配置正确并且成功的导入了依赖包，那么整个工程即可直接编译。在本地 shell 下进入工程目录，执行下面的命令对整个工程进行打包：

```
mvn package
```

运行过程中可能还需要下载一些文件，直到出现 build success 表示打包成功。然后您可以在工程目录下的 target 文件夹中看到打好的 jar 包。
使用 scp 或者 sftp 服务来把打包好的工程文件上传到 EMR 集群的云服务器中。在本地 shell 使用：

```
scp $jarpackage root@公网IP地址: /usr/local/service/hadoop
```

其中，`$jarpackage` 是您的本地 jar 包的路径加名称；root 为 CVM 服务器用户名；公网 IP 地址可以在 EMR 控制台的节点信息中或者在云服务器控制台查看。这里上传到了 EMR 集群的 `/usr/local/service/hadoop` 文件夹下。

统计 HDFS 中的文本文件

进入 `/usr/local/service/hadoop` 目录，和数据准备中一样。通过如下命令来提交任务：

```
[hadoop@10 hadoop]$ bin/hadoop jar
/usr/local/service/hadoop/WordCount-1.0-SNAPSHOT-jar-with-dependencies.jar
WordCount /user/hadoop/test.txt /user/hadoop/WordCount_output
```

> **注意**
> 以上整个命令为一条完整的指令，`/user/hadoop/test.txt` 为输入的待处理文件，`/user/hadoop/ WordCount_output` 为输出文件夹，在提交命令前要保证 `WordCount_output` 文件夹尚未创建，否则提交会出错。

执行完成后，通过如下命令查看执行输出文件：

```
[hadoop@172 hadoop]$ hadoop fs -ls /user/hadoop/WordCount_output
Found 2 items
-rw-r--r-- 3 hadoop supergroup 0 2018-07-06 11:35 /user/hadoop/MEWordCount_output/_SUCCESS
-rw-r--r-- 3 hadoop supergroup 82 2018-07-06 11:35 /user/hadoop/MEWordCount_output/part-r-00000
```

通过如下指令查看 part-r-00000 中的统计结果：

```
[hadoop@172 hadoop]$ hadoop fs -cat /user/hadoop/MEWordCount_output/part-r-00000
Hello   2
World.  1
a   1
another 1
are 1
how 1
is  2
message.    2
this    2
world,  1
you?    1……
```

统计 COS 中的文本文件

进入 `/usr/local/service/hadoop` 目录。通过如下命令来提交任务：

```
[hadoop@10 hadoop]$ hadoop jar
/usr/local/service/hadoop/WordCount-1.0-SNAPSHOT-jar-with-dependencies.jar
WordCount cosn://$bucketname/test.txt cosn://$bucketname /WordCount_output
```

命令的输入文件改为了 `cosn:// $bucketname/test.txt`，其中 $bucketname 为您的存储桶名字加路径。处理结果同样也输出到 COS 中。使用如下指令查看输出文件：

```
[hadoop@10 hadoop]$ hadoop fs -ls cosn:// $bucketname /WordCount_output
Found 2 items
```

```
-rw-rw-rw- 1 hadoop Hadoop 0 2018-07-06 10:34 cosn://$bucketname /WordCount_output/_SUCCESS
-rw-rw-rw- 1 hadoop Hadoop 1306 2018-07-06 10:34 cosn://$bucketname /WordCount_output/part-r-00000
```

查看最后输出的结果：

```
[hadoop@10 hadoop]$ hadoop fs -cat cosn:// $bucketname /WordCount_output1/part-r-00000
Hello   2
World.  1
a   1
another 1
are 1
how 1
is  2
message.    2
this    2
world,  1
you?    1
```

YARN 作业日志转存 COS

最近更新时间：2023-06-27 14:48:57

Hadoop 默认将 YARN 的作业日志存储在 hdfs 上，腾讯云 EMR 还提供了将 YARN 作业日志存储在外部存储 COS 上。

前提条件

EMR 集群需要支持 COS，详情可参考 使用 API 分析 HDFS/COS 上的数据 。

操作步骤

1. 登录 EMR 控制台，在集群列表中选择对应的集群，单击**集群服务**进入集群服务列表。
2. 通过 YARN 服务面板右上角**操作 > 配置管理**，找到 `yarn-site.xml` 配置文件并修改以下配置，下发所有节点。

```
yarn.nodemanager.remote-app-log-dir=cosn://$bucketname/$logs_dirs
```

其中，$bucketname 为您的存储桶名，$logs_dirs 为您希望将 yarn 作业日志转存的文件目录。

3. 通过 HDFS 服务面板右上角**操作 > 配置管理**，找到 `core-site.xml` 并新增以下配置项，下发所有节点。

```
fs.AbstractFileSystem.cosn.impl=org.apache.hadoop.fs.cosnative.COS
```

4. 在**集群服务 > HDFS > 角色管理**中重启集群所有 `nodemanager/datanode` 服务。
5. 运行 `hive/spark` 作业后，可通过以下命令查看存储在 COS 上的作业日志。

```
hdfs dfs -ls cosn://$bucketname/$logs_dirs
```

Spark 开发指南
Spark 环境信息

最近更新时间：2023-12-15 10:33:24

腾讯云 EMR 提供 Spark 3.x、2.x 多版本支持，软件环境信息如下：

- Spark 默认安装在 master 节点。
- 登录机器后使用命令 `su hadoop` 切换到 Hadoop 用户。
- Spark 软件路径在 `/usr/local/service/spark` 下。
- 相关日志路径在 `/data/emr` 下。

更多详细资料请参考 社区文档，这里主要介绍基于 Spark 访问腾讯云对象存储相关操作。

Spark 分析 COS 上的数据

最近更新时间：2024-08-15 14:44:01

Spark 作为 Apache 高级的开源项目，是一个快速、通用的大规模数据处理引擎，与 Hadoop 的 MapReduce 计算框架类似，但是相对于 MapReduce，Spark 凭借其可伸缩、基于内存计算等特点以及可以直接读写 Hadoop 上任何格式数据的优势，进行批处理时更加高效，并有更低的延迟。实际上，Spark 已经成为轻量级大数据快速处理的统一平台，各种不同的应用，如实时流处理、机器学习、交互式查询等，都可以通过 Spark 建立在不同的存储和运行系统上。
Spark 是基于内存计算的大数据并行计算框架。Spark 基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将 Spark 部署在大量廉价硬件之上，形成集群。
本教程演示的是提交的任务为 wordcount 任务即统计单词个数，提前需要在集群中上传需要统计的文件。

1. 开发准备

- 因为任务中需要访问腾讯云对象存储（COS），所以需要在 COS 中先 创建一个存储桶（Bucket）。
- 确认您已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Spark 组件，并且在**实例信息 > 基础配置**中开启对象存储的授权。

2. 使用 Maven 创建工程

在本次演示中，不再采用系统自带的演示程序，而是自己建立工程编译打包之后上传到 EMR 集群运行。推荐您使用 Maven 来管理您的工程。Maven 是一个项目管理工具，能够帮助您方便的管理项目的依赖信息，即它可以通过 pom.xml 文件的配置获取 jar 包，而不用去手动添加。
首先下载并安装 Maven，配置 Maven 的环境变量。如果您使用 IDE，请在 IDE 中设置 Maven 相关配置。

新建一个 Maven 工程

在本地 shell 下进入您想要新建工程的目录，例如 `D://mavenWorkplace` 中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=$yourgroupID -DartifactId=$yourartifactID -DarchetypeArtifactId=maven-
archetype-quickstart
```

其中 $yourgroupID 即为您的包名。$yourartifactID 为您的项目名称，而 maven-archetype-quickstart 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。
创建成功后，在 `D://mavenWorkplace` 目录下就会生成一个名为 $yourartifactID 的工程文件夹。其中的文件结构如下所示：

```
simple
    |--pom.xml        核心配置，项目根下
    |--src
        |--main
            |--java          Java 源码目录
            |--resources     Java 配置文件目录
        |--test
             |--java          测试源码目录
             |--resources     测试配置目录
```

其中我们主要关心 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。
首先在 pom.xml 中添加 Maven 依赖：

```
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>2.0.2</version>
    </dependency>
</dependencies>
```

继续在 pom.xml 中添加打包和编译插件：

```
<build>
<plugins>
  <plugin>
```

```
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

在 src>main>Java 下右键新建一个Java Class，输入您的 Class 名，这里使用 WordCountOnCos，在 Class 添加样例代码：

```
import java.util.Arrays;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import scala.Tuple2;

/**
 * Created by tencent on 2018/6/28.
 */
public class WordCountOnCos {
    public static void main(String[] args){
        SparkConf sc = new SparkConf().setAppName("spark on cos");
        JavaSparkContext context = new JavaSparkContext(sc);
        JavaRDD<String> lines = context.textFile(args[0]);

        lines.flatMap(x -> Arrays.asList(x.split(" ")).iterator())
                .mapToPair(x -> new Tuple2<String, Integer>(x, 1))
                .reduceByKey((x, y) -> x+y)
                .saveAsTextFile(args[1]);
    }
}
```

如果您的 Maven 配置正确并且成功地导入了依赖包，那么整个工程应该没有错误可以直接编译。在本地命令行模式下进入工程目录，执行下面的命令对整个工程进行打包：

```
mvn package
```

运行过程中可能还需要下载一些文件，直到出现 build success 表示打包成功。然后您可以在工程目录下的 target 文件夹中看到打好的 jar 包。

数据准备

首先需要把压缩好的 jar 包上传到 EMR 集群中，使用 scp 或者 sftp 工具来进行上传。在本地命令行模式下运行：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，$localfile 是您的本地文件的路径加名称；root 为 CVM 服务器用户名；公网 IP 可以在 EMR 控制台的节点信息中或者在云服务器控制台查看；$remotefolder 是您想存放文件的 CVM 服务器路径。上传完成后，在 EMR 命令行中即可查看对应文件夹下是否有相应文件。
需要处理的文件需要事先上传到 COS 中。如果文件在本地则可以通过 COS 控制台直接上传 。如果文件在 EMR 集群上，可以使用 Hadoop 命令上传。指令如下：

```
[hadoop@10 hadoop]$ hadoop fs -put $testfile cosn://$bucketname/
```

其中 $testfile 为要统计的文件的完整路径加名字，$bucketname 为您的存储桶名。上传完成后可以在 COS 控制台中查看文件是否已经在 COS 中。

运行样例

首先需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户：

```
[root@172 ~]# su hadoop
```

然后进入您存放 jar 包的文件夹下，执行以下指令：

```
[hadoop@10spark]$ spark-submit --class $WordCountOnCOS --master yarn-cluster $packagename.jar
cosn://$bucketname/$testfile cosn://$bucketname/output
```

其中 $WordCountOnCOS 为您的 Java Class 名字，$packagename 为您新建 Maven 工程中生成的 jar 包名字，$bucketname 为您的存储桶名和路径，$testfile 为您要统计的文件名。最后输出的文件在 output 这个文件夹中，**这个文件夹事先不能被创建，不然运行会失败。**
运行成功后，在指定的存储桶和文件夹下可以看到 wordcount 的结果。

```
[hadoop@172 /]$ hadoop fs -ls cosn://$bucketname/output
Found 3 items
-rw-rw-rw- 1 hadoop Hadoop  0 2018-06-28 19:20 cosn://$bucketname/output/_SUCCESS
-rw-rw-rw- 1 hadoop Hadoop 681 2018-06-28 19:20 cosn://$bucketname/output/part-00000
-rw-rw-rw- 1 hadoop Hadoop 893 2018-06-28 19:20 cosn://$bucketname/output/part-00001

[hadoop@172 demo]$ hadoop fs -cat cosn://$bucketname/output/part-00000
18/07/05 17:35:01 INFO cosnative.NativeCosFileSystem: Opening 'cosn://$bucketname/output/part-00000' for
reading
(under,1)
(this,3)
(distribution,2)
(Technology,1)
(country,1)
(is,1)
(Jetty,1)
(currently,1)
(permitted.,1)
(Security,1)
(have,1)
(check,1)
```

通过 Spark Python 分析 COS 上的数据

最近更新时间：2024-08-15 14:44:01

本节主要是通过 Spark Python 来进行 wordcount 的工作。

开发准备

- 因为任务中需要访问腾讯云对象存储（COS），所以需要在 COS 中先 创建一个存储桶（Bucket）。
- 确认您已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置页面选择 Spark 组件，并且在基础配置页面开启对象存储的授权。

数据准备

需要处理的文件需要事先上传到 COS 中。如果文件在本地那么就可以通过 COS 控制台直接上传。如果文件在 EMR 集群上，可以使用 Hadoop 命令上传。指令如下：

```
[hadoop@10 hadoop]$ hadoop fs -put $testfile cosn://$bucketname/
```

其中 $testfile 为要统计的文件的完整路径加名字，$bucketname 为您的存储桶名。上传完成后可以查看文件是否已经在 COS 中。

运行样例

首先需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Spark 安装目录 `/usr/local/service/spark`：

```
[root@172 ~]# su hadoop
[hadoop@172 root]$ cd /usr/local/service/spark
```

新建一个 Python 文件 wordcount.py，并添加如下代码：

```
from __future__ import print_function

import sys
from operator import add
from pyspark.sql import SparkSession

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)

    spark = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .getOrCreate()

    sc = spark.sparkContext

    lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)

    output = counts.collect()
    counts.saveAsTextFile(sys.argv[2])

    spark.stop()
```

通过如下指令提交任务：

```
[hadoop@10   spark]$   ./bin/spark-submit   --master yarn   ./wordcount.py
cosn://$bucketname/$yourtestfile cosn://$bucketname/$output
```

其中 $bucketname 为您的 COS 存储桶名，$yourtestfile 为您的测试文件在存储桶中的完整路径加名字。$output 为您的输出文件夹。$output 为一个未创建的文件夹，如果执行指令前该文件夹已经存在，会导致程序运行失败。
成功后程序自动运行，可以在目标存储桶中查看到输出文件：

```
[hadoop@172 spark]$ hadoop fs -ls cosn://$bucketname/$output
Found 2 items
-rw-rw-rw- 1 hadoop Hadoop 0 2018-06-29 15:35 cosn://$bucketname/$output/_SUCCESS
-rw-rw-rw- 1 hadoop Hadoop 2102 2018-06-29 15:34 cosn://$bucketname/$output/part-00000
```

最后的结果也可以通过如下指令查看：

```
[hadoop@172 spark]$ hadoop fs -cat cosn://$bucketname/$output /part-00000
(u'', 27)
(u'code', 1)
(u'both', 1)
(u'Hadoop', 1)
(u'Bureau', 1)
(u'Department', 1)
```

同样可以把结果输出到 HDFS 中，只需要更改指令中的输出位置即可，如下所示：

```
[hadoop@10spark]$   ./bin/spark-submit   ./wordcount.py
cosn://$bucketname/$yourtestfile/user/hadoop/$output
```

其中 `/user/hadoop/` 为 HDFS 中的路径，如果不存在用户可以自己创建。
任务结束后，可以通过如下命令看到 Spark 运行日志：

```
[hadoop@10 spark]$  /usr/local/service/hadoop/bin/yarn logs -applicationId $yourId
```

其中 $yourId 应该替换为您的任务 ID。任务 ID 可以在 YARN 的 WebUI 上面进行查看。

SparkSQL 的使用

最近更新时间：2023-12-15 10:33:24

Spark 为结构化数据处理引入了一个称为 Spark SQL 的编程模块。它提供了一个称为 DataFrame 的编程抽象，并且可以充当分布式 SQL 查询引擎。

1. 开发准备

确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择了 Spark 组件。

2. 使用 SparkSQL 交互式控制台

在使用 SparkSQL 之前请登录 EMR 集群的 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入 EMR 命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入目录 `/usr/local/service/spark`：

```
[root@172 ~]# su hadoop
[hadoop@172 root]$ cd /usr/local/service/spark
```

通过如下命令您可以进入 SparkSQL 的交互式控制台：

```
[hadoop@10spark]$ bin/spark-sql --master yarn --num-executors 64 --executor-memory 2g
```

其中 --master 表示您的 master URL，--num-executors 表示 executor 数量，--executor-memory 表示 executor 的储存容量。以上参数也可以根据您的实际情况作出修改，您也可以通过 `sbin/start-thriftserver.sh` 或者 `sbin/stop-thriftserver.sh` 来启动或者停止一个 SparkSQLthriftserver。
下面介绍一些 SparkSQL 的基本操作：

- 新建一个数据库并查看：

```
spark-sql> create database sparksql;
Time taken: 0.907 seconds
spark-sql> show databases;
default
sparksql
test
Time taken: 0.131 seconds, Fetched 5 row(s)
```

- 在新建的数据库中新建一个表，并进行查看：

```
spark-sql> use sparksql;
Time taken: 0.076 seconds
spark-sql> create table sparksql_test(a int,b string);
Time taken: 0.374 seconds
spark-sql> show tables;
sparksql_test   false
Time taken: 0.12 seconds, Fetched 1 row(s)
```

- 向表中插入两行数据并查看：

```
spark-sql> insert into sparksql_test values (42,'hello'),(48,'world');
Time taken: 2.641 seconds
spark-sql> select * from sparksql_test;
42  hello
48  world
Time taken: 0.503 seconds, Fetched 2 row(s)
```

更多命令行参数使用教程请参考 社区文档 。

3. 使用 Maven 创建工程

首先下载并安装 Maven，配置好 Maven 的环境变量，如果您使用 IDE，请在 IDE 中设置好 Maven 相关配置。

新建一个 Maven 工程

在命令行下进入您想要新建工程的目录，例如 `D://mavenWorkplace` 中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=$yourgroupID -DartifactId=$yourartifactID -DarchetypeArtifactId=maven-
archetype-quickstart
```

其中 $yourgroupID 即为您的包名。$yourartifactID 为您的项目名称， maven-archetype-quickstart 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。
创建成功之后，在 `D://mavenWorkplace` 目录下就会生成一个名为 $yourartifactID 的工程文件夹。其中的文件结构如下所示：

```
simple
     ---pom.xml        核心配置，项目根下
     ---src
        ---main
           ---java          Java 源码目录
           ---resources      Java 配置文件目录
       ---test
          ---java         测试源码目录
          ---resources     测试配置目录
```

其中我们主要关心 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置， Java 文件夹下放置您的源代码。

添加 Hadoop 依赖和样例代码

在集群上找一个节点，看下依赖的 spark 版本：

```
cd /usr/local/service/spark/jars && ll

-rw-r--r-- 1 hadoop hadoop 31870390 Jun 20 16:05 hudi-spark3.2-bundle_2.12-0.11.0.jar
-rw-r--r-- 1 hadoop hadoop 16094752 Jun 20 16:05 kudu-spark3_2.12-1.15.0.jar
-rw-r--r-- 1 hadoop hadoop   186484 Jun 20 16:05 spark-avro_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop 11645730 Jun 20 16:05 spark-catalyst_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop 10841257 Jun 20 16:05 spark-core_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   431111 Jun 20 16:05 spark-graphx_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    11983 Jun 20 16:05 spark-hadoop-cloud_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   700945 Jun 20 16:05 spark-hive_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   570134 Jun 20 16:05 spark-hive-thriftserver_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   453567 Jun 20 16:05 spark-kubernetes_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    61811 Jun 20 16:05 spark-kvstore_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    76516 Jun 20 16:05 spark-launcher_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  6137585 Jun 20 16:05 spark-mllib_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   116164 Jun 20 16:05 spark-mllib-local_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  2412460 Jun 20 16:05 spark-network-common_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   160623 Jun 20 16:05 spark-network-shuffle_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    52799 Jun 20 16:05 spark-repl_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    30670 Jun 20 16:05 spark-sketch_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  8341311 Jun 20 16:05 spark-sql_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  1139792 Jun 20 16:05 spark-streaming_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    15156 Jun 20 16:05 spark-tags_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop     9913 Jun 20 16:05 spark-tags_2.12-3.2.1-tests.jar
-rw-r--r-- 1 hadoop hadoop    51765 Jun 20 16:05 spark-unsafe_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   349957 Jun 20 16:05 spark-yarn_2.12-3.2.1.jar
```

首先在 pom.xml 文件中添加 Maven 依赖（根据上面 ll 的结果修改 scala.version、spark.version 对应的值）：

```
    <properties>
        <scala.version>2.12</scala.version>
        <spark.version>3.2.1</spark.version>
```

```
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_${scala.version}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_${scala.version}</artifactId>
            <version>${spark.version}</version>
        </dependency>
    </dependencies>
```

继续在 pom.xml 文件中添加打包和编译插件：

```
<build>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

完整的 pom.xml 文件如下所示：

```
<?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>$yourgroupID </groupId>
    <artifactId>$yourartifactID </artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
```

```
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_${scala.version}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_${scala.version}</artifactId>
            <version>${spark.version}</version>
        </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>
                    <encoding>utf-8</encoding>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

注意

修改其中的 $yourgroupID 和 $yourartifactID 为您自己的设置。

接下来添加样例代码，在 main>Java 文件夹下新建一个 Java Class 取名为 Demo.java，并将以下代码加入其中：

```
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

/**
 * Created by tencent on 2018/6/28.
 */
public class Demo {
    public static void main(String[] args){
        SparkSession spark = SparkSession
                .builder()
                .appName("Java Spark Hive Example")
```

```
                .enableHiveSupport()
                .getOrCreate();

        Dataset<Row> df = spark.read().json(args[0]);

        RDD<Row> test = df.rdd();

        test.saveAsTextFile(args[1]);
    }
}
```

编译代码并打包上传

使用本地命令行进入工程目录，执行以下指令对工程进行编译打包：

```
mvn package
```

在显示 build success 表示操作成功，在工程目录下的 target 文件夹中能够看到打包好的文件。
使用 scp 或者 sftp 工具把打包好的文件上传到 EMR 集群。在本地命令行模式下运行：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，$localfile 是您的本地文件的路径加名称，root 为 CVM 服务器用户名，公网 IP 可以在 EMR 控制台的节点信息中或者在云服务器控制台查看。
$remotefolder 是您想存放文件的 CVM 服务器路径。上传完成后，在 EMR 集群命令行中即可查看对应文件夹下是否有相应文件。

4. 准备数据并运行样例

使用 sparkSQL 来操作存放在 HDFS 上的数据。首先将数据上传到 HDFS 中，这里我们使用自带的文件 people.json，存放在路径 `/usr/local/service/spark/examples/src/main/resources/` 下，使用如下指令把该文件上传到 HDFS 中：

```
[hadoop@10 hadoop]$ hadoop fs -put /usr/local/service/spark/examples/src/main/resources/people.json
/user/hadoop
```

测试文件用户也可以另选，这里 `/user/hadoop/` 是 HDFS 下的文件夹，如果没有用户可以自己创建。
执行样例，首先请登录 EMR 集群的 master 节点，并且切换到 Hadoop 用户，如 步骤2 使用 SparkSQL 交互式控制台 中所示，使用以下命令执行样例：

```
[hadoop@10spark]$ bin/spark-submit --class Demo --master yarn --deploy-mode client $yourjarpackage /
/user/hadoop/people.json  /user/hadoop/$output
```

其中 --class 参数表示要执行的入口类，在本例子中即为 Demo，即在添加 Hadoop 依赖和样例代码中创建的 Java Class 的名字，--master 为集群主要的 URL，$yourjarpackage 是您打包后的包名，$output 为结果输出文件夹（**$output 为一个未创建的文件夹，如果执行指令前该文件夹已经存在，会导致程序运行失败**）。
成功运行后，可以在 `/user/hadoop/$output` 查看结果：

```
[hadoop@172 spark]$ hadoop fs -cat /user/hadoop/$output/part-00000
[null,Michael]
[30,Andy]
[19,Justin]
```

spark-submit 的更多参数，在命令行输入以下命令进行查看，或者请参考 官方文档 。

```
[hadoop@10spark]$ spark-submit -h
```

SparkStreaming 对接 Ckafka 服务

最近更新时间：2024-10-11 15:17:11

基于腾讯云的 EMR 服务您可以轻松结合腾讯云的 Ckafka 服务实现以下流式应用：

- 日志信息流式处理
- 用户行为记录流式处理
- 告警信息收集及处理
- 消息系统

1. 开发准备

- 因为任务中需要访问腾讯云消息队列 CKafka，所以需要先创建一个 CKafka 实例，具体见 消息队列 CKafka 。
- 确认您已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群时需要在软件配置界面选择 Spark 组件。

2. 在 EMR 集群使用 Kafka 工具包

首先需要查看 CKafka 的内网 IP 与端口号。登录消息队列 CKafka 的控制台，选择您要使用的 CKafka 实例，在基本消息中查看其内网 IP 为 $kafkaIP，而端口号一般默认为9092。在 topic 管理界面新建一个 topic 为 spark_streaming_test。
登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里可选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入目录 `/usr/local/service/spark`：

```
[root@172 ~]# su hadoop
[root@172 root]$ cd /usr/local/service/spark
```

从 Kafka 官网 下载安装包，注意选择合适的版本，具体可参考 EMR 各版本 Kafka 与 Spark 版本说明 。kafka 客户端版和腾讯云 ckafka 兼容性强，安装对应的 kafka 客户端版本即可。本文以 kafka_2.10-0.10.2.0 版本为示例，请注意在命令及代码中使用正确版本，解压压缩包并将解压出来的文件夹移动到/opt目录下：

```
[hadoop@172 data]$ tar -xzvf kafka_2.10-0.10.2.0.tgz
[hadoop@172 data]$ mv kafka_2.10-0.10.2.0 /opt/
```

解压完成后，Kafka 工具直接能使用。可以使用 `telnet` 命令来测试 EMR 集群是否能够连接到 CKafka 实例：

```
[hadoop@172 kafka_2.10-0.10.2.0]$ telnet $kafkaIP 9092
Trying $kafkaIP...
Connected to $kafkaIP.
```

其中 $kafkaIP 为您创建的 CKafka 实例的内网 IP 地址。
下面可以简单测试 Kafka 工具包，同时用两个 WebShell 登录 EMR 集群并切换到 Hadoop 用户，进入 Kafka 的安装路径：

```
[root@172 ~]# su hadoop
[hadoop@172 root]$ cd /opt/kafka_2.10-0.10.2.0/
```

在第一个终端上连接 CKafka，并向其发送消息：

```
[hadoop@172 kafka_2.10-0.10.2.0]$ bin/kafka-console-producer.sh --broker-list $kafkaIP:9092
--topic spark_streaming_test
hello world
this is a message
```

在另一个终端上连接 CKafka，并作为消费者获得其中的数据：

```
[hadoop@172 kafka_2.10-0.10.2.0]$ bin/kafka-console-consumer.sh --bootstrap-server
$kafkaIP:9092 --from-beginning --new-consumer --topic spark_streaming_test
hello world
```

```
this is a message
```

3. 使用 SparkStreaming 对接 CKafka 服务

在消费者一端，我们利用 Spark Streaming 从 CKafka 中不断拉取数据进行词频统计，即对流数据进行 WordCount 的工作。在生产者一端，也采用程序不断地产生数据，来不断输送给 CKafka。
首先 下载并安装 Maven，配置好 Maven 的环境变量，如果您使用 IDE，请在 IDE 中设置好 Maven 相关配置。

创建 Spark Streaming 消费者工程

在本地命令行下进入您想要新建工程的目录，例如 `D://mavenWorkplace` 中，输入如下命令新建一个 Maven 工程：

```
mvn   archetype:generate   -DgroupId=$yourgroupID   -DartifactId=$yourartifactID
-DarchetypeArtifactId=maven-archetype-quickstart
```

其中 $yourgroupID 即为您的包名。$yourartifactID 为您的项目名称，maven-archetype-quickstart 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。
创建成功后，在 `D://mavenWorkplace` 目录下就会生成一个名为 $yourartifactID 的工程文件夹。其中的文件结构如下所示：

```
simple
     ---pom.xml        核心配置，项目根下
     ---src
        ---main
           ---java         Java 源码目录
           ---resources     Java 配置文件目录
       ---test
          ---java         测试源码目录
          ---resources     测试配置目录
```

其中我们主要关心 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。
在集群上找一个节点，看下依赖的 spark 版本：

```
cd /usr/local/service/spark/jars && ll

-rw-r--r-- 1 hadoop hadoop 31870390 Jun 20 16:05 hudi-spark3.2-bundle_2.12-0.11.0.jar
-rw-r--r-- 1 hadoop hadoop 16094752 Jun 20 16:05 kudu-spark3_2.12-1.15.0.jar
-rw-r--r-- 1 hadoop hadoop   186484 Jun 20 16:05 spark-avro_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop 11645730 Jun 20 16:05 spark-catalyst_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop 10841257 Jun 20 16:05 spark-core_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   431111 Jun 20 16:05 spark-graphx_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    11983 Jun 20 16:05 spark-hadoop-cloud_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   700945 Jun 20 16:05 spark-hive_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   570134 Jun 20 16:05 spark-hive-thriftserver_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   453567 Jun 20 16:05 spark-kubernetes_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    61811 Jun 20 16:05 spark-kvstore_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    76516 Jun 20 16:05 spark-launcher_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  6137585 Jun 20 16:05 spark-mllib_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   116164 Jun 20 16:05 spark-mllib-local_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  2412460 Jun 20 16:05 spark-network-common_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   160623 Jun 20 16:05 spark-network-shuffle_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    52799 Jun 20 16:05 spark-repl_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    30670 Jun 20 16:05 spark-sketch_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  8341311 Jun 20 16:05 spark-sql_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop  1139792 Jun 20 16:05 spark-streaming_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop    15156 Jun 20 16:05 spark-tags_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop     9913 Jun 20 16:05 spark-tags_2.12-3.2.1-tests.jar
-rw-r--r-- 1 hadoop hadoop    51765 Jun 20 16:05 spark-unsafe_2.12-3.2.1.jar
-rw-r--r-- 1 hadoop hadoop   349957 Jun 20 16:05 spark-yarn_2.12-3.2.1.jar
```

首先在 pom.xml 文件中添加 Maven 依赖：

```
    <properties>
        <scala.version>2.12</scala.version>
        <spark.version>3.2.1</spark.version>
    </properties>
<dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_${scala.version}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_${scala.version}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka-0-10_${scala.version}</artifactId>
            <version>${spark.version}</version>
        </dependency>
</dependencies>
```

继续在 pom.xml 文件中添加打包和编译插件：

```
<build>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

> **注意**
> 修改其中的 $yourgroupID 和 $yourartifactID 为您自己的设置。

接下来添加样例代码，在 main>Java 文件夹下新建一个 Java Class 取名为 KafkaTest.java，并将以下代码加入其中：

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.JavaInputDStream;
import org.apache.spark.streaming.api.java.JavaPairDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.apache.spark.streaming.kafka010.ConsumerStrategies;
import org.apache.spark.streaming.kafka010.KafkaUtils;
import org.apache.spark.streaming.kafka010.LocationStrategies;
import scala.Tuple2;

import java.util.*;
import java.util.concurrent.TimeUnit;

/**
 * Created by tencent on 2018/7/3.
 */
public class KafkaTest {
    public static void main(String[] args) throws InterruptedException {
        String brokers = "$kafkaIP:9092";
        String topics = "spark_streaming_test1";  // 订阅的话题，多个话题','分隔
        int durationSeconds = 60;  // 间隔时间
        SparkConf conf = new SparkConf().setAppName("spark streaming word count");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaStreamingContext ssc = new JavaStreamingContext(sc, Durations.seconds(durationSeconds));
        Collection<String> topicsSet = new HashSet<>(Arrays.asList(topics.split(",")));
        //kafka相关参数
        Map<String, Object> kafkaParams = new HashMap<>();
        kafkaParams.put("metadata.broker.list", brokers) ;
        kafkaParams.put("bootstrap.servers", brokers);
        kafkaParams.put("group.id", "group1");
        kafkaParams.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        kafkaParams.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        kafkaParams.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        //创建连接
        JavaInputDStream<ConsumerRecord<Object,Object>> lines = KafkaUtils.createDirectStream(
                ssc,
                LocationStrategies.PreferConsistent(),
                ConsumerStrategies.Subscribe(topicsSet, kafkaParams)
        );
        //wordcount逻辑
        JavaPairDStream<String, Integer> counts = lines
                .flatMap(x -> Arrays.asList(x.value().toString().split(" ")).iterator())
                .mapToPair(x -> new Tuple2<String, Integer>(x, 1))
                .reduceByKey((x, y) -> x + y);
        // 保存结果
        counts.dstream().saveAsTextFiles("$hdfsPath","result");
//
        ssc.start();
        ssc.awaitTermination();
        ssc.close();
    }
}
```

代码中要注意以下几点设置：

- brokers 变量要设置为在第二步中查找到的 CKafka 实例的内网 IP。
- topics 变量要设置为自己创建的 topic 的名字，这里为 spark_streaming_test1。
- durationSeconds 为程序去 CKafka 中消费数据的时间间隔，这里为60秒。
- $hdfsPath 为 HDFS 中的路径，结果将会输出到该路径下。

使用本地命令行进入工程目录，执行以下指令对工程进行编译打包：

```
mvn package
```

显示 build success 表示操作成功，在工程目录下的 target 文件夹中能够看到打包好的文件。
使用 scp 或者 sftp 工具来把打包好的文件上传到 EMR 集群，注意一定要上传依赖一起打包的 jar 包：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，$localfile 是您的本地文件的路径加名称，root 为 CVM 服务器用户名，公网 IP 可以在 EMR 控制台的节点信息中或者在云服务器控制台查看。
$remotefolder 是您想存放文件的 CVM 服务器路径。上传完成后，在 EMR 集群命令行中即可查看对应文件夹下是否有相应文件。

创建 Spark Streaming 生产者工程

在本地命令行下进入您想要新建工程的目录，例如 `D://mavenWorkplace` 中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=$yourgroupID -DartifactId=$yourartifactID
-DarchetypeArtifactId=maven-archetype-quickstart
```

首先在 pom.xml 文件中添加 Maven 依赖：

```
<dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka_2.11</artifactId>
            <version>0.10.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>0.10.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>0.10.1.0</version>
        </dependency>
    </dependencies>
```

继续在 pom.xml 文件中添加打包和编译插件：

```
<build>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
    <executions>
```

```
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

> **注意**
> 修改其中的 $yourgroupID 和 $yourartifactID 为您自己的设置。

接下来添加样例代码，在 main>Java 文件夹下新建一个 Java Class 取名为 SendData.java，并将以下代码加入其中：

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;


/**
 * Created by tencent on 2018/7/4.
 */
public class SendData {
    public static void main(String[] args) {

        Properties props = new Properties();
        props.put("bootstrap.servers", "$kafkaIP:9092");
        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");
        //生产者发送消息
        String topic = "spark_streaming_test1";
        org.apache.kafka.clients.producer.Producer<String, String> procuder = new
KafkaProducer<String,String>(props);
        while(true){
            int num = (int)((Math.random())*10);
            for (int i = 0; i <= 10; i++) {
                int tmp = (num+i)%10;
                String value = "value_" + tmp;
                ProducerRecord<String, String> msg = new ProducerRecord<String, String>(topic, value);
                procuder.send(msg);
            }

            try {Thread.sleep(1000*10);}
            catch (InterruptedException e) {}
        }
    }
}
```

修改其中的 $kafkaIP 为您的 CKafka 的内网 IP 地址。
这个程序每10秒向 CKafka 发送10条消息从 value_0 到 value_9，其开始的顺序随机。程序中的参数信息参考消费者程序。
使用本地命令行进入工程目录，执行以下指令对工程进行编译打包：

```
mvn package
```

显示 build success 表示操作成功，在工程目录下的 target 文件夹中能够看到打包好的文件。
使用 scp 或者 sftp 工具来把打包好的文件上传到 EMR 集群，注意一定要上传依赖一起打包的 jar 包：

```
scp $localfile root@公网IP地址:$remotefolder
```

使用程序消费 CKafka 的数据

使用两个界面分别登录 EMR 集群的 Web Shell。
第一个界面：登录 EMR 集群的 Master 节点，并且切换到 Hadoop 用户如2节中所示，使用以下命令执行样例：

```
[hadoop@172 ~]$ bin/spark-submit --class KafkaTest --master yarn --deploy-mode cluster $consumerpackage
```

其中参数如下：

- --class 参数表示要执行的入口类，在本例中即为 KafkaTest。
- --master 为集群主要的 URL。
- $ consumerpackage 是您的消费者打包后的包名。

程序开始执行后，将会在 yarn 集群上一直运行，使用以下指令可以查看到程序运行的状态：

```
[hadoop@172 ~]$ yarn application -list
```

第二个界面：登录 EMR 的 Web Shell，然后运行生产者程序，以便 Spark Streaming 能够从中取数据消费。

```
[hadoop@172 spark]$ bin/spark-submit --class SendData $producerpackage
```

其中 $producerpackage 为您的生产者打包后的包名。等待一段时间后，会在指定的 HDFS 文件夹中输出 wordcount 的结果，可以到 HDFS 中查看 Spark Streaming 消费 CKafka 数据后输出的结果：

```
[hadoop@172 root]$ hdfs dfs -ls /user
Found 9 items
drwxr-xr-x - hadoop supergroup  0 2018-07-03 16:37 /user/hadoop
drwxr-xr-x - hadoop supergroup  0 2018-06-19 10:10 /user/hive
-rw-r--r-- 3 hadoop supergroup 0 2018-06-29 10:19 /user/pythontest.txt
drwxr-xr-x - hadoop supergroup 0 2018-07-05 20:25 /user/sparkstreamingtest-1530793500000.result

[hadoop@172 root]$ hdfs dfs -cat /user/sparkstreamingtest-1530793500000.result/*
(value_6,16)
(value_7,22)
(value_8,18)
(value_0,18)
(value_9,17)
(value_1,18)
(value_2,17)
(value_3,17)
(value_4,16)
(value_5,17)
```

最后需要退出 yarn 集群中的 KafkaTest 程序：

```
[hadoop@172 ~]$ yarn application -kill $Application-Id
```

其中 $Application-Id 为使用 `yarn application -list` 命令查找到的 ID。
更多 Kafka 的相关信息请查看 官方文档 。

Spark 资源动态调度实践

最近更新时间：2024-08-15 14:44:01

开发准备

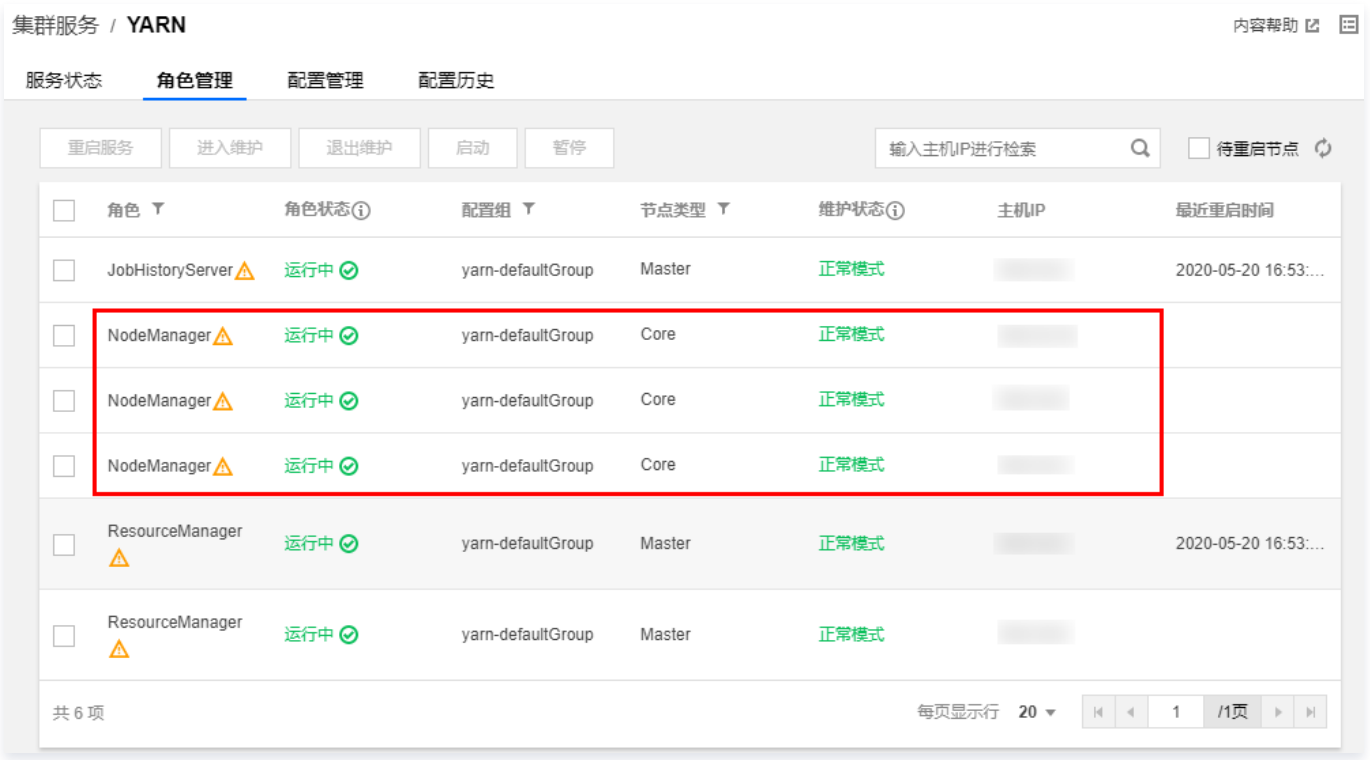
确定您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候，需要在软件配置界面选择 spark_hadoop 组件。
Spark 安装在 EMR 云服务器的 `/usr/local/service` 路径下（ `/usr/local/service/spark` ）。

拷贝 jar 包

需要将 `spark-<version>-yarn-shuffle.jar` 拷贝到集群各个节点的 `/usr/local/service/hadoop/share/hadoop/yarn/lib` 目录路径下。

方法一：SSH 控制台操作

1. 在**集群服务 > YARN 组件**中，选择**操作 > 角色管理**，确定 NodeManager 所在节点 IP。



2. 依次登录每个 NodeManager 所在节点。
   - 首先，需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例，这里我们可以使用 XShell 登录 Master 节点。
   - 使用 SSH 登录到其他 NodeManager 所在节点。指令为 `ssh $user@$ip` ，$user 为登录的用户名，$ip 为远程节点 IP（即步骤1中确定的 IP 地址）。

```
[root@172 ~]# ssh root@172.16.
The authenticity of host '                    ' can't be established.
ECDSA key fingerprint is b9:2f:9c:97:91:66:b4:4b:2a:89:                .
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '            ' (ECDSA) to the list of known hosts.
root@172.16          password:
```

- 验证已经成功切换。

```
[root@172 ~]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet [redacted]  netmask 255.255.240.0  broadcast 172.16.47.255
        ether 52:54:00:0d:ad:e3  txqueuelen 1000  (Ethernet)
        RX packets 212406  bytes 110895121 (105.7 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 178536  bytes 26768271 (25.5 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet [redacted]  netmask 255.0.0.0
        loop  txqueuelen 1  (Local Loopback)
        RX packets 378979  bytes 167242376 (159.4 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 378979  bytes 167242376 (159.4 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. 搜索 `spark-<version>-yarn-shuffle.jar` 文件路径。

```
[root@172 ~]# find / -name *shuffle.jar
/usr/local/service/hadoop/share/hadoop/yarn/spark-2.3.2-yarn-shuffle.jar
/usr/local/service/spark/yarn/spark-2.3.2-yarn-shuffle.jar
/usr/local/service/hive/spark/yarn/spark-2.0.2-yarn-shuffle.jar
/usr/local/service/apps/kylin-2.5.2/spark/yarn/spark-2.1.2-yarn-shuffle.jar
```

4. 将 `spark-<version>-yarn-shuffle.jar` 拷贝到 `/usr/local/service/hadoop/share/hadoop/yarn/lib` 下。

```
[root@172 ~]# cd /usr/local/service/hadoop/share/hadoop/yarn/lib/
[root@172 lib]# cp /usr/local/service/spark/yarn/spark-2.3.2-yarn-shuffle.jar spark-2.3.2-yarn-shuffle.jar
[root@172 lib]# ls
activation-1.1.jar             hadoop-lzo-0.4.20.jar          jetty-util-6.1.26.jar
aopalliance-1.0.jar            jackson-core-asl-1.9.13.jar    json-io-2.5.1.jar
asm-3.2.jar                    jackson-jaxrs-1.9.13.jar       jsr305-3.0.0.jar
commons-cli-1.2.jar            jackson-mapper-asl-1.9.13.jar  leveldbjni-all-1.8.jar
commons-codec-1.4.jar          jackson-xc-1.9.13.jar          log4j-1.2.17.jar
commons-collections-3.2.2.jar  javassist-3.18.1-GA.jar        netty-3.6.2.Final.jar
commons-compress-1.4.1.jar     java-util-1.9.0.jar            protobuf-java-2.5.0.jar
commons-io-2.4.jar             javax.inject-1.jar             ranger-plugin-classloader-0.7.1.jar
commons-lang-2.6.jar           jaxb-api-2.2.2.jar             ranger-yarn-plugin-impl
commons-logging-1.1.3.jar      jaxb-impl-2.2.3-1.jar          ranger-yarn-plugin-shim-0.7.1.jar
commons-math-2.2.jar           jersey-client-1.9.jar          servlet-api-2.5.jar
curator-client-2.7.1.jar       jersey-core-1.9.jar            spark-2.3.2-yarn-shuffle.jar
curator-test-2.7.1.jar         jersey-guice-1.9.jar           stax-api-1.0-2.jar
fst-2.50.jar                   jersey-json-1.9.jar            xz-1.0.jar
guava-11.0.2.jar               jersey-server-1.9.jar          zookeeper-3.4.6.jar
guice-3.0.jar                  jettison-1.1.jar               zookeeper-3.4.6-tests.jar
guice-servlet-3.0.jar          jetty-6.1.26.jar
[root@172 lib]# ls | grep spark-*
spark-2.3.2-yarn-shuffle.jar
```

5. 退出登录，并切换其余节点。

```
[root@172 lib]# exit
logout
Connection to [redacted] closed.
```

方法二：批量部署脚本

首先，需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例，这里我们可以使用 XShell 登录 Master 节点。

编写如下批量传输文件的 Shell 脚本。当集群节点很多时，为了避免多次输入密码，可以使用 sshpass 工具传输。sshpass 的优势在于可以免密传输避免多次输入，但其缺点在于密码是明文容易暴露，可以使用 history 命令找到。

1. 免密，安装 sshpass。

```
[root@172 ~]# yum install sshpass
```

编写如下脚本：

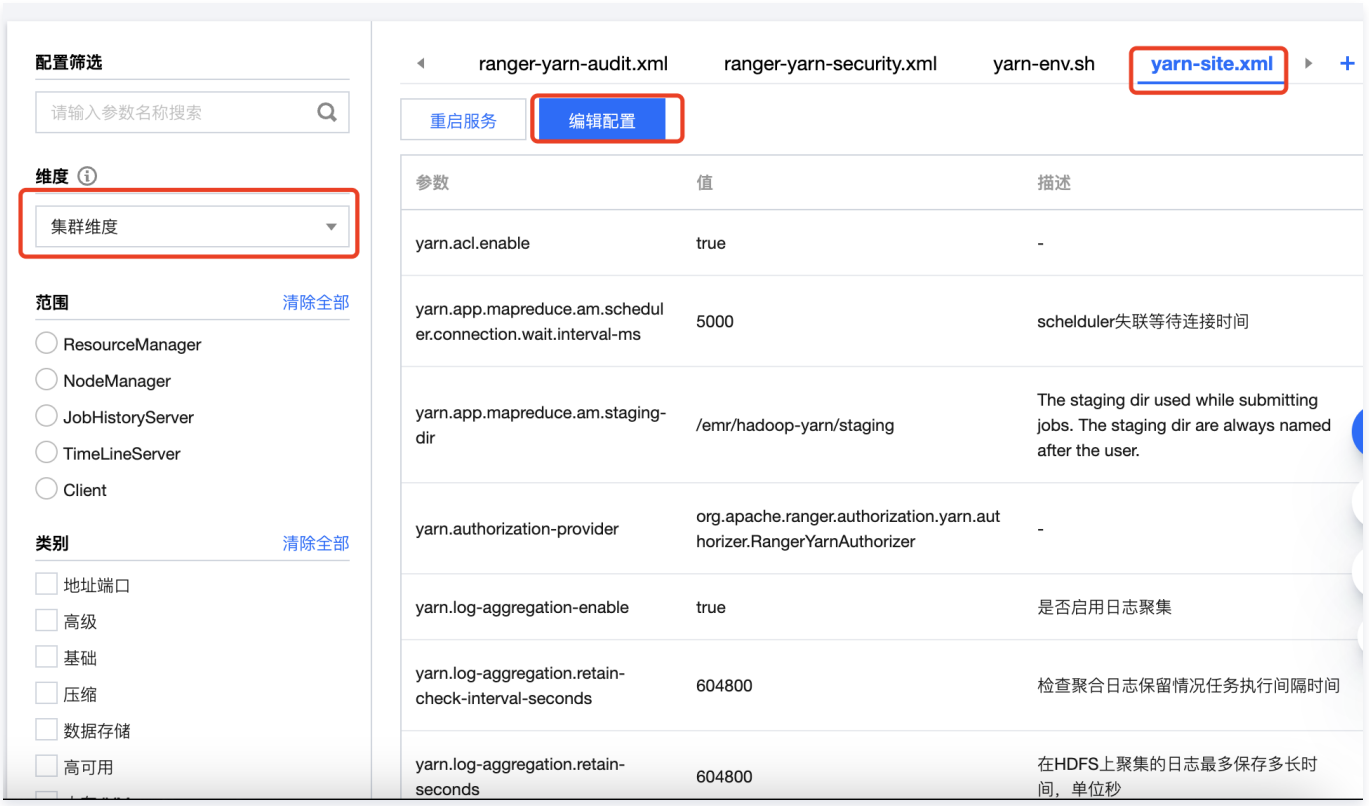
```
#!/bin/bash
nodes=(ip1 ip2 … ipn) #集群各节点 IP 列表，空格分隔
len=${#nodes[@]}
password=<your password>
file=" spark-2.3.2-yarn-shuffle.jar "
source_dir="/usr/local/service/spark/yarn"
target_dir="/usr/local/service/hadoop/share/hadoop/yarn/lib"
echo $len
for node in ${nodes[*]}
do
  echo $node;
  sshpass -p $password scp "$source_dir/$file"root@$node:"$target_dir";
done
```

2. **非免密。**
   编写如下脚本：

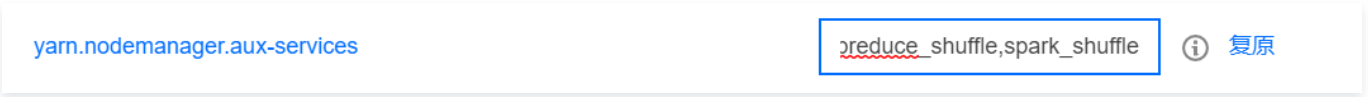
```
#!/bin/bash
nodes=(ip1 ip2 … ipn) #集群各节点 IP 列表，空格分隔
len=${#nodes[@]}
password=<your password>
file=" spark-2.3.2-yarn-shuffle.jar "
source_dir="/usr/local/service/spark/yarn"
target_dir="/usr/local/service/hadoop/share/hadoop/yarn/lib"
echo $len
for node in ${nodes[*]}
do
    echo $node;
    scp "$source_dir/$file" root@$node:"$target_dir";
done
```

修改 Yarn 配置

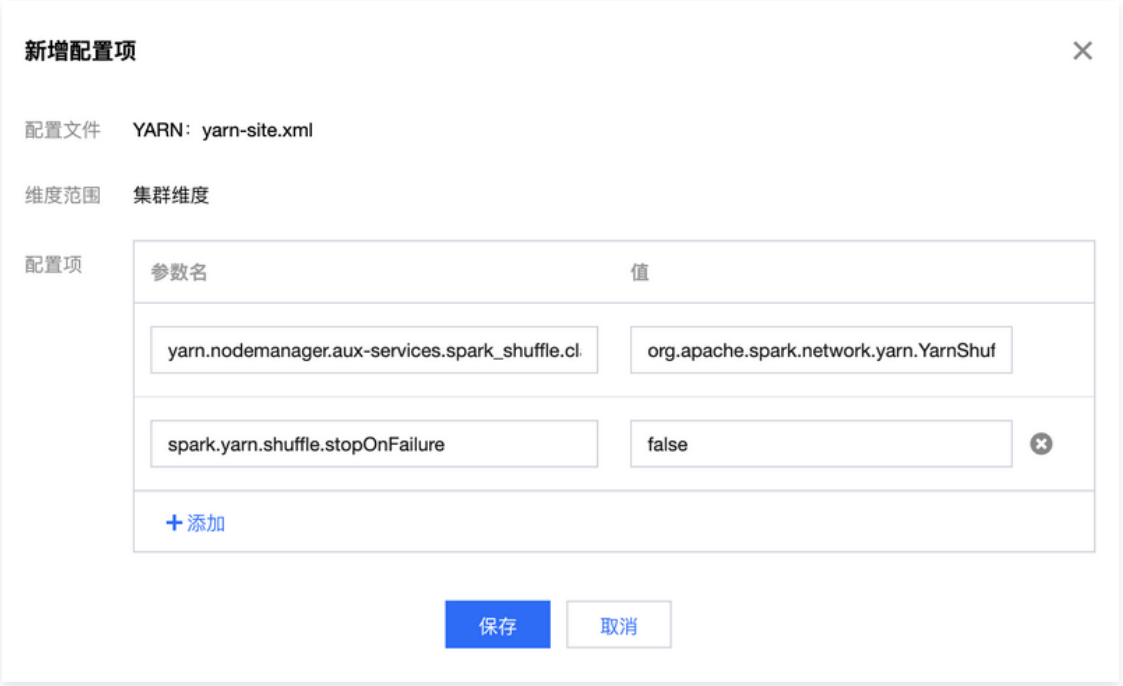
1. **在集群服务 > YARN组件中，选择操作 > 配置管理。选中配置文件 yarn-site.xml，维度范围选择“集群维度”（集群维度的配置项修改将应用到所有节点），然后单击编辑配置。**



2. 修改配置项 `yarn.nodemanager.aux-services`，添加 spark_shuffle。



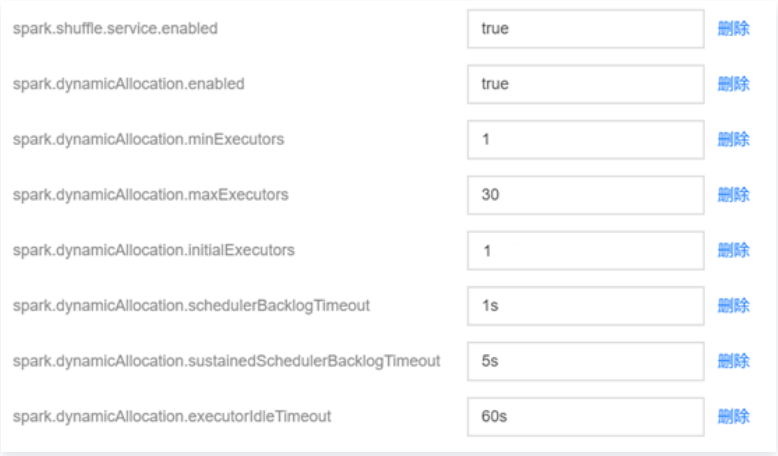
3. 新增配置项 yarn.nodemanager.aux-services.spark_shuffle.class，该配置项的值设置为 org.apache.spark.network.yarn.YarnShuffleService。
4. 新增配置项 spark.yarn.shuffle.stopOnFailure，该配置项的值设置为 false。



5. 保存设置并下发，重启 YARN 组件使得配置生效。

修改 Spark 配置

1. 在**集群服务 > SPARK组件**中，选择**操作 > 配置管理**。
2. 选中配置文件“spark-defaults.conf”，单击**编辑配置**。新建配置项如下：



配置项	值	备注
spark.shuffle.service.enabled	true	启动 shuffle 服务。
spark.dynamicAllocation.enabled	true	启动动态资源分配。
spark.dynamicAllocation.minExecutors	1	每个 Application 最小分配的 executor 数。
spark.dynamicAllocation.maxExecutors	30	每个 Application 最大分配的 executor 数。
spark.dynamicAllocation.initialExecutors	1	一般情况下与 spark.dynamicAllocation.minExecutors 值相同。
spark.dynamicAllocation.schedulerBacklogTimeout	1s	已有挂起的任务积压超过此持续事件，则将请求新的执行程序。
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout	5s	带处理任务队列依然存在，则此后每隔几秒再次出发，每轮请求的 executor 数目与上轮相比呈指数增长。
spark.dynamicAllocation.executorIdleTimeout	60s	Application 在空闲超过几秒钟时会删除 executor。

3. 保存配置、下发并重启组件。

测试 Spark 资源动态调整

1. 测试环境资源配置说明

测试环境下，共两个部署 NodeManager 服务的节点，每个节点资源配置为4核 CPU、8GB内存，集群总资源为8核 CPU、16GB内存。

2. 测试任务说明

测试一：

- 在 EMR 控制台中，进入 `/usr/local/service/spark` 目录，切换 hadoop 用户，使用 `spark-submit` 提交一个任务，数据需存储在 hdfs 上。

```
[root@172 ~]# cd /usr/local/service/spark/
[root@172 spark]# su hadoop
[hadoop@172 spark]$  hadoop fs -put ./README.md /
[hadoop@172 spark]$ spark-submit --class org.apache.spark.examples.JavaWordCount --master yarn --num-executors 10 --driver-memory 4g --executor-memory 4g --executor-cores 2 ./examples/jars/spark-examples_2.11-2.3.2.jar /README.md /output
```

- 在 YARN 组件的 WebUI 界面 Application 面板中，可以观察到配置前后容器和 CPU 分配情况。

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blackl Node
application_1562242321020_0001	hadoop	spark word count	SPARK	default	0	Thu Jul 4 20:17:09 +0800 2019	N/A	RUNNING	UNDEFINED	3	3	9920	168.2	67.3		ApplicationMaster	0

- 未设置资源动态调度前，CPU 最多分配个数为3。

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklis Nodes
application_1562243747104_0001	hadoop	spark word count	SPARK	root.default	0	Thu Jul 4 20:40:57 +0800 2019	N/A	RUNNING	UNDEFINED	3	5	11264	76.4	76.4		ApplicationMaster	0

- 设置资源动态调度后，CPU 最多分配个数为5。

结论：配置资源动态调度后，调度器会根据应用程序的需要动态地增加分配的资源。

测试二：

- 在 EMR 控制台中，进入 `/usr/local/service/spark` 目录，切换 hadoop 用户，使用 `spark-sql` 启动 SparkSQL 交互式控制台。交互式控制台被设置成占用测试集群的大部分资源，观察设置资源动态调度前后资源分配情况。

```
[root@172 ~]# cd /usr/local/service/spark/
[root@172 spark]# su hadoop
[hadoop@172 spark]$ spark-sql --master yarn --num-executors 5 --driver-memory 4g --executor-memory 2g --
executor-cores 1
```

- 使用 spark2.3.0 自带的计算圆周率的 example 作为测试任务，提交任务时将任务的 executor 数设置为5，driver 内存设置为4g，executor 内存设置为4g，executor 核数设置为2。

```
[root@172 ~]# cd /usr/local/service/spark/
[root@172 spark]# su hadoop
[hadoop@172 spark]$ spark-submit --class org.apache.spark.examples.SparkPi --master yarn --num-executors 5
--driver-memory 4g --executor-memory 4g --executor-cores 2 examples/jars/spark-examples_2.11-2.3.2.jar 500
```

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	% of Queue	% of Cluster	Progress	Tracking UI
application_1562243747104_0005	hadoop	SparkSQL::172.16.32.47	SPARK	root.default	0	Thu Jul 4 21:03:15 +0800 2019	N/A	RUNNING	UNDEFINED	5	5	13312	90.3	90.3		ApplicationMaster

- 只运行 SparkSQL 任务时资源占用率90.3%。

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	% of Queue	% of Cluster	Progress	Tracking UI
application_1562243747104_0006	hadoop	Spark Pi	SPARK	root.default	0	Thu Jul 4 21:04:22 +0800 2019	N/A	ACCEPTED	UNDEFINED	1	1	1024	6.9	6.9		ApplicationMaster
application_1562243747104_0005	hadoop	SparkSQL::172.16.32.47	SPARK	root.default	0	Thu Jul 4 21:03:15 +0800 2019	N/A	RUNNING	UNDEFINED	2	2	4096	27.8	27.8		ApplicationMaster

- 提交 SparkPi 任务后，SparkSQL 资源占用率27.8%。

结论：SparkSQL 任务虽然在提交时申请了大量资源，但并未执行任何分析任务，因此实际上有大量空闲的资源。当超过 `spark.dynamicAllocation.executorIdleTimeout` 设置时间时，空闲的 executor 被释放，其他任务获得资源。在本次测试中 SparkSQL 任务的集群资源占用率从90%降至28%，空闲资源分配给圆周率计算任务，自动调度有效。

> **说明**
> 配置项 `spark.dynamicAllocation.executorIdleTimeout` 的值将影响资源动态调度的快慢，测试发现资源调度用时基本与该值相等，建议您根据实际需求调整该配置项的值以获得最佳性能。

Spark 集成 Kafka 说明

最近更新时间：2022-10-31 10:44:00

依赖关系

Spark 从2.3 起不再支持 Kafka0.8.2，EMR 现网版本已集成 Spark2.4.3及以上版本，需要集成 kafka 0.10.0及更高版本。

查找方法

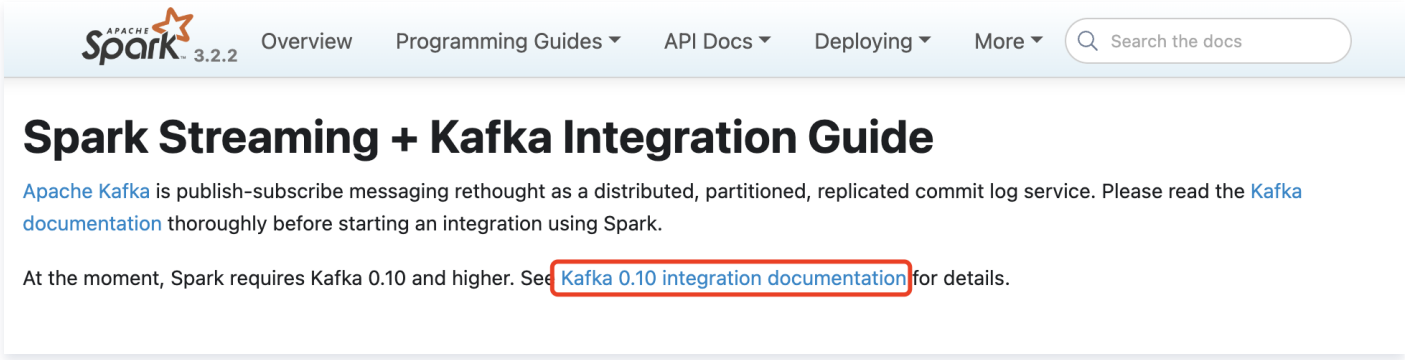
1. 访问官网链接，输入版本号链接模板：

```
https://spark.apache.org/docs/{spark.version}/streaming-kafka-integration.html
```

将 `{spark.version}` 替换为对应的 Spark 版本。例如查看3.2.2版本的依赖关系，访问链接如下：

```
https://spark.apache.org/docs/3.2.2/streaming-kafka-integration.html
```

2. 单击相应链接可看到详细集成说明。



EMR 各版本 Spark 相关依赖说明

最近更新时间：2023-12-15 16:51:22

依赖关系

Spark	scala	Python	R	Java
2.4.3	2.12.x	2.7+/3.4+	3.1+	8+
3.0.0	2.12.x	2.7+/3.4+	3.1+	8/11
3.0.2	2.12.x	2.7+/3.4+	3.5+	8/11
3.2.1	2.12.x/2.13.x	3.6+	3.5+	8/11
3.2.2	2.12.x/2.13.x	3.6+	3.5+	8/11
3.3.2	2.12.x/2.13.x	3.7+	3.5+	8/11/17

查找方法

1. 访问官网链接，输入版本号链接模板：

```
https://spark.apache.org/docs/{spark.version}/index.html
```

将 `{spark.version}` 替换为对应的 spark 版本，例如查看3.2.2版本的依赖关系，访问链接如下：

```
https://spark.apache.org/docs/3.2.2/index.html
```

2. 查看依赖

Downloading

Get Spark from the downloads page of the project website. This documentation is for Spark version 3.2.2. Spark uses Hadoop's client libraries for HDFS and YARN. Downloads are pre-packaged for a handful of popular Hadoop versions. Users can also download a "Hadoop free" binary and run Spark with any Hadoop version by augmenting Spark's classpath. Scala and Java users can include Spark in their projects using its Maven coordinates and Python users can install Spark from PyPI.

If you'd like to build Spark from source, visit Building Spark.

Spark runs on both Windows and UNIX-like systems (e.g. Linux, Mac OS), and it should run on any platform that runs a supported version of Java. This should include JVMs on x86_64 and ARM64. It's easy to run locally on one machine — all you need is to have `java` installed on your system `PATH`, or the `JAVA_HOME` environment variable pointing to a Java installation.

Spark runs on Java 8/11, Scala 2.12/2.13, Python 3.6+ and R 3.5+. Python 3.6 support is deprecated as of Spark 3.2.0. Java 8 prior to version 8u201 support is deprecated as of Spark 3.2.0. For the Scala API, Spark 3.2.2 uses Scala 2.12. You will need to use a compatible Scala version (2.12.x).

For Python 3.9, Arrow optimization and pandas UDFs might not work due to the supported Python versions in Apache Arrow. Please refer to the latest Python Compatibility page. For Java 11, `-Dio.netty.tryReflectionSetAccessible=true` is required additionally for Apache Arrow library. This prevents `java.lang.UnsupportedOperationException: sun.misc.Unsafe or java.nio.DirectByteBuffer.(long, int) not available` when Apache Arrow uses Netty internally.

使用 Spark Native 引擎（Beta）

最近更新时间：2024-07-05 10:45:41

随着 SSD 的广泛应用及网卡性能的显著提升，Spark 引擎的性能瓶颈更多由传统认知的 IO 转变为以 CPU 为主的计算资源。而围绕 JVM 的 CPU 优化方案（如 Codegen ）存在诸多约束，如字节码长度、参数个数等存在限制，开发者也很难在 JVM 上利用现代 CPU 的一些特性。
Spark Native 引擎在 Spark 物理计划进行转换，使用 C++ 实现的向量化加速库执行计算，并将执行完的数据以列式方式返回，提升内存及带宽利用效率，进而突破性能瓶颈，可以使 Spark 作业效率得到有效的提升。

使用限制

Spark Native 引擎目前存在使用场景限制，在限制场景时将 Spark Native引擎会在对应的 Stage 进行 fallback，回退到原生 Spark 引擎进行执行，由于 fallback 需要进行必要的数据转换，fallback 次数过多，可能会导致总体运行时间比原生 Spark 引擎更慢。
请您提前了解 Spark Native 引擎部分主要的使用限制：

- 支持 Parquet 数据格式，ORC 支持暂不完善，其他数据格式暂不支持。
- 暂不支持 Iceberg、Hudi 等湖格式。
- 暂不支持 ANSI 模式。
- 暂不支持 Date、Timestamp、Decimal、NaN 数据类型。
- 暂不支持动态分区写入。
- 暂不支持原生 Spark 的 columnar reading。

配置方式

> **注意：**
> Spark Native 引擎目前仍在测试阶段，您可以通过 EMR- V3.6.1（Beta）版本使用该特性进行测试，不建议用于生产作业。

创建 EMR- V3.6.1 版本集群，您可以通过 EMR 控制台 配置管理 功能，在 spark-defaults.conf 配置文件中新增以下配置使用该特性：

参数	说明
spark.plugins	Spark 用到的插件，参数值设置为 io.glutenproject.GlutenPlugin（如果已经配置了 spark.plugins，则可以将 io.glutenproject.GlutenPlugin 加到其中，用逗号","隔开）
spark.memory.offHeap.enabled	设置为 true，Native 加速需要用到 JVM 的 off memory
spark.memory.offHeap.size	设置 offHeap 内存的大小，根据实际情况设置，初始可设置为1G
spark.shuffle.manager	gluten 使用的列式 shuffle manager， 参数值设置为：org.apache.spark.shuffle.sort.ColumnarShuffleManager
spark.driver.extraClassPath	Spark 用到的 gluten native jar，jar 包默认路径在 /usr/local/service/spark/gluten 下
spark.executor.extraClassPath	Spark 用到的 gluten native jar，jar 包默认路径在 /usr/local/service/spark/gluten 下
spark.executorEnv.LIBHDFS3_CONF	集成的 HDFS 集群配置文件的路径，默认在/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml

HBase 开发指南
通过 API 使用 HBase

最近更新时间：2025-03-17 11:34:02

HBase 是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，是 Google BigTable 的开源实现。HBase 利用 Hadoop HDFS 作为其文件存储系统；Hadoop MapReduce 来处理 HBase 中的海量数据；Zookeeper 来做协同服务。
HBase 主要由 Zookeeper、HMaster 和 HRegionServer 组成。其中：

- ZooKeeper 可避免 HMaster 的单点故障，其 Master 选举机制可保证一个 Master 提供服务。
- HMaster 管理用户对表的增删改查操作，管理 HRegionServer 的负载均衡。并可调整 Region 的分布，在 HRegionServer 退出时迁移其内的 HRegion 到其他 HRegionServer 上。
- HRegionServer 是 HBase 中最核心的模块，其主要负责响应用户的 I/O 请求，向 HDFS 文件系统中读写数据。HRegionServer 内部管理了一系列 HRegion 对象，每个 HRegion 对应一个 Region，HRegion 中由多个 Store 组成。每个 Store 对应了 Column Family 的存储。

本开发指南将从技术人员的角度帮助用户使用 EMR 集群开发。考虑用户数据安全，EMR 中当前只支持 VPC 网络访问。

1. 开发准备

确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择了 HBase 组件和 Zookeeper 组件。

2. 使用 HBase Shell

在使用 HBase Shell 之前请登录 EMR 集群的 Master 节点。登录 EMR 的方式可参考 登录 Linux 实例 。这里可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入 EMR 命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入目录 `/usr/local/service/hbase`：

```
[root@172 ~]# su hadoop
[hadoop@10root]$ cd /usr/local/service/hbase
```

通过如下命令您可以进入 HBase Shell：

```
[hadoop@10hbase]$ bin/hbase shell
```

在 hbase shell 下输入 help 可以查看基本的使用信息和示例的指令。接下来我们使用以下指令建立一个新表：

```
hbase(main):001:0> create 'test', 'cf'
```

表格建立后，可以使用 `list` 指令来查看您建立的表是否已经存在。

```
hbase(main):002:0> list 'test'
TABLE
test
1 row(s) in 0.0030 seconds

=> ["test"]
```

使用 `put` 指令来为您创建的表加入元素：

```
hbase(main):003:0> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.0850 seconds

hbase(main):004:0> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0110 seconds

hbase(main):005:0> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0100 seconds
```

我们在创建的表中加入了三个值，第一次在“row1”行“cf:a”列插入了一个值“value1”，以此类推。

使用 `scan` 指令来遍历整个表：

```
hbase(main):006:0> scan 'test'
ROW  COLUMN+CELL
row1   column=cf:a, timestamp=1530276759697, value=value1
row2   column=cf:b, timestamp=1530276777806, value=value2
row3   column=cf:c, timestamp=1530276792839, value=value3
3 row(s) in 0.2110 seconds
```

使用 `get` 指令来取得表中指定行的值：

```
hbase(main):007:0> get 'test', 'row1'
COLUMN  CELL
 cf:a       timestamp=1530276759697, value=value
1 row(s) in 0.0790 seconds
```

使用 `drop` 指令来删除一个表，在删除表之前需要先使用 `disable` 指令来禁用一个表：

```
hbase(main):010:0> disable 'test'
hbase(main):011:0> drop 'test'
```

最后可以使用 `quit` 指令来关闭 HBase Shell。
更多的 HBase shell 指令请查看 官方文档 。

3. 通过 API 使用 HBase

首先 下载并安装 Maven，配置 Maven 的环境变量，如果您使用 IDE，请在 IDE 中设置 Maven 相关配置。

新建一个 Maven 工程

在命令行下进入您想要新建工程的目录，例如 `D://mavenWorkplace` 中，输入如下命令新建一个 Maven 工程：

```
mvn      archetype:generate      -DgroupId=$yourgroupID       -DartifactId=$yourartifactID
-DarchetypeArtifactId=maven-archetype-quickstart
```

其中 $yourgroupID 即为您的包名。$yourartifactID 为您的项目名称，而 maven-archetype-quickstart 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。
创建成功后，在 `D://mavenWorkplace` 目录下就会生成一个名为 $yourartifactID 的工程文件夹。其中的文件结构如下所示：

```
simple
     ---pom.xml        核心配置，项目根下
     ---src
        ---main
           ---java        Java 源码目录
           ---resources     Java 配置文件目录
       ---test
           ---java        测试源码目录
           ---resources     测试配置目录
```

其中我们主要关心 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。

添加 Hadoop 依赖和样例代码

首先在 pom.xml 文件中添加 Maven 依赖：

```
<dependencies>
     <dependency>
         <groupId>org.apache.hbase</groupId>
         <artifactId>hbase-client</artifactId>
         <version>1.2.4</version>
```

```
    </dependency>
</dependencies>
```

然后在 pom.xml 文件中添加打包和编译插件：

```
<build>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

在添加样例代码之前，需要用户获取 HBase 集群的 zookeeper 地址。登录 EMR 任意一台 Master 节点或者 Core 节点，进入 `/usr/local/service/hbase/conf` 目录，查看 hbase-site.xml 的 hbase.zookeeper.quorum 配置获得 zookeeper 的 IP 地址 $quorum，hbase.zookeeper.property.clientPort 配置获得 zookeeper 的端口号 $clientPort，zookeeper.znode.parent 配置获得 hbase 使用的 znode 路径 $znodePath。
接下来添加样例代码，在 main>java 文件夹下新建一个 Java Class 取名为 PutExample.java，并将以下代码加入其中：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.io.compress.Compression.Algorithm;

import java.io.IOException;

/**
 * Created by tencent on 2018/6/30.
 */
public class PutExample {
    public static void main(String[] args) throws IOException {
        Configuration conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum","$quorum");
        conf.set("hbase.zookeeper.property.clientPort","$clientPort");
        conf.set("zookeeper.znode.parent", "$znodePath");

        Connection connection = ConnectionFactory.createConnection(conf);
```

```
        Admin admin = connection.getAdmin();

        HTableDescriptor table = new HTableDescriptor(TableName.valueOf("test1"));
        table.addFamily(new HColumnDescriptor("cf").setCompressionType(Algorithm.NONE));

        System.out.print("Creating table. ");
        if (admin.tableExists(table.getTableName())) {
            admin.disableTable(table.getTableName());
            admin.deleteTable(table.getTableName());
        }
        admin.createTable(table);

        Table table1 = connection.getTable(TableName.valueOf("test1"));
        Put put1 = new Put(Bytes.toBytes("row1"));
        put1.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("a"),
                Bytes.toBytes("value1"));
        table1.put(put1);
        Put put2 = new Put(Bytes.toBytes("row2"));
        put2.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("b"),
                Bytes.toBytes("value2"));
        table1.put(put2);
        Put put3 = new Put(Bytes.toBytes("row3"));
        put3.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("c"),
                Bytes.toBytes("value3"));
        table1.put(put3);

        System.out.println(" Done.");
    }
}
```

编译代码并打包上传

使用本地命令行进入工程目录，执行以下指令对工程进行编译打包：

```
mvn package
```

显示 build success 表示操作成功，在工程目录下的 target 文件夹中能够看到打包好的文件。
使用 scp 或者 sftp 工具把打包好的文件上传到 EMR 集群。**这里一定要上传把依赖一起进行打包的 jar 包**。在本地命令行模式下运行：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，$localfile 是您的本地文件的路径加名称，root 为 CVM 服务器用户名，公网 IP 可以在 EMR 控制台的节点信息中或者在云服务器控制台查看。
$remotefolder 是您想存放文件的 CVM 服务器路径。上传完成后，在 EMR 集群命令行中即可查看对应文件夹下是否有相应文件。

4. 运行样例

登录 EMR 集群的 Master 节点，并且切换到 hadoop 用户。使用如下命令执行样例：

```
[hadoop@10 hadoop]$ java -jar $package.jar
```

在控制台输出“Done”后，说明所有的操作已完成。可切换到 HBase Shell 中使用 `list` 命令来查看使用 API 创建的 HBase 表是否成功。如果成功可使用 `scan` 命令来查看表的具体内容。

```
[hadoop@10hbase]$ bin/hbase shell
hbase(main):002:0> list 'test1'
TABLE
Test1
1 row(s) in 0.0030 seconds

=> ["test1"]
```

```
hbase(main):006:0> scan 'test1'
ROW  COLUMN+CELL
row1   column=cf:a, timestamp=1530276759697, value=value1
row2   column=cf:b, timestamp=1530276777806, value=value2
row3   column=cf:c, timestamp=1530276792839, value=value3
3 row(s) in 0.2110 seconds
```

更多的 API 使用说明请参见 官方文档 。

通过 Thrift 使用 HBase

最近更新时间：2025-03-17 11:34:02

Apache Thrift 是一个跨平台、跨语言的开发框架，提供多语言的编译功能，并提供多种服务器工作模式。用户通过 Thrift 的 IDL（接口描述语言）来描述接口函数及数据类型，然后通过 Thrift 的编译环境生成各种语言类型的接口文件，用来进行可扩展且跨语言的服务的开发。
它结合了功能强大的软件堆栈和代码生成引擎，以构建在 C++、Java、Go、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、JavaScript、Node.js、Smalltalk 和 OCaml 编程语言间无缝结合的、高效的服务。
Thrift server 是 HBase 中的一种服务，主要用于对多语言 API 的支持。基于 Apache Thrift 开发。Thrift API 依赖于客户端和服务器进程。
本节以 Python2 版本为例，说明如何通过 Thrift 利用 Python 编程来使用 HBase。

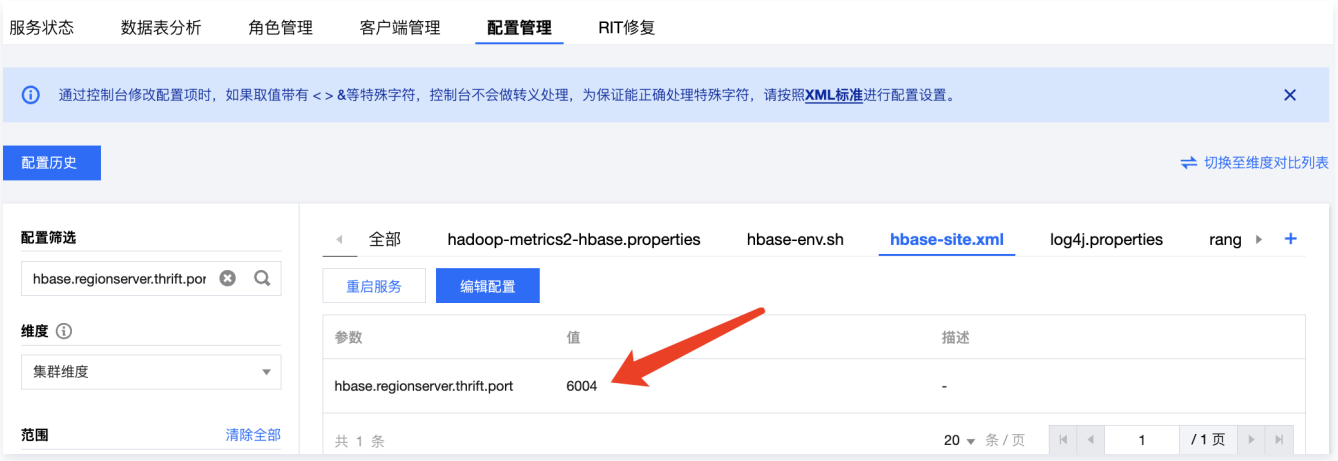
1. 开发准备

确认您已开通腾讯云，并且创建了一个 EMR 集群。创建 EMR 集群时需要在软件配置界面选择 HBase 组件。

2. 通过 Python API 使用 HBase

EMR 集群中 HBase 默认集成了 Thrift，并在 Master1（外网 IP 节点）节点上启动了 Thrift Server。
1. 在集群详情页面中选择**集群服务**，单击 HBase 组件右上角**操作 > 角色管理**，即可查看 Thrift Server 运行状态及 IP 地址。

2. 单击**配置管理**页面，在 hbase-site.xml 文件中通过 hbase.regionserver.thrift.port thrift 配置项，查看thrift 的端口号：



下面我们可以直接使用 Python 编程来操作 HBase。

负载均衡

HA 集群有两个 master 节点，两个节点默认都启动了 Thrift Server。若需要实现负载均衡，客户端代码需要自定义策略将请求分散到两台 Thrift Server 上，这两台 Thrift Server 是完全独立的，之间没通信。

准备数据

使用 HBase Shell 在 HBase 中新建一个表，如果您使用过 EMR 的 HBase 并且创建过自己的表，那么该步骤可以略过：

```
[hadoop@172 hbase]$ hbase shell

hbase(main):001:0> create 'thrift_test', 'cf'
hbase(main):005:0> list
thrift_test
1 row(s) in 0.2270 seconds

hbase(main):001:0> quit
```

使用 Python 查看 HBase 中的表

首先需要安装 Python 依赖包，切换到 root 用户下，密码即为创建 EMR 集群时您设置的密码，先安装 python-pip 工具再安装依赖包：

```
[hadoop@172 hbase]$ su
Password: ********
[root@172 hbase]# yum install python-pip
[root@172 hbase]# pip install hbase-thrift
```

然后切换回 Hadoop 用户并新建一个 Python 文件 Hbase_client.py，在其中加入以下代码：

```
#! /usr/bin/env python
#coding=utf-8

from thrift.transport import TSocket,TTransport
from thrift.protocol import TBinaryProtocol
from hbase import Hbase

socket = TSocket.TSocket('$thriftIP', $port)
socket.setTimeout(5000)

transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)

client = Hbase.Client(protocol)
transport.open()

print client.getTableNames()
```

> **注意**
> 其中 $thriftIP为 Master 节点在内网的 IP 地址，$port 为 ThriftService 的端口号，下同。

保存后直接运行程序，会直接在控制台输出 HBase 中的存在的表：

```
[hadoop@172 hbase]$ python Hbase_client.py
['thrift_test']
```

使用 Python 创建一个 HBase 表

新建一个 Python 文件 Create_table.py，把以下代码加入其中：

```
#! /usr/bin/env python
#coding=utf-8
```

```
from thrift import Thrift
from thrift.transport import TSocket,TTransport
from thrift.protocol import TBinaryProtocol
from hbase import Hbase
from hbase.ttypes import ColumnDescriptor,Mutation,BatchMutation,TRegionInfo
from hbase.ttypes import IOError,AlreadyExists

socket = TSocket.TSocket('$thriftIP',$port)
socket.setTimeout(5000)

transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)

client = Hbase.Client(protocol)
transport.open()

new_table = ColumnDescriptor(name = 'cf:',maxVersions = 1)
client.createTable('thrift_test_1',[new_table])

tables = client.getTableNames()
socket.close()

print tables
```

该程序会在 HBase 中添加一个名为 thrift_test_1 的新表，并且输出所有存在的表，运行效果如下：

```
[hadoop@172 hbase]$ python Create_table.py
['thrift_test', 'thrift_test_1']
```

使用 Python 在 HBase 表中插入数据

新建一个 Python 文件 Insert.py，把以下代码加入其中：

```
#! /usr/bin/env python
#coding=utf-8

from thrift import Thrift
from thrift.transport import TSocket,TTransport
from thrift.protocol import TBinaryProtocol
from hbase import Hbase
from hbase.ttypes import ColumnDescriptor,Mutation,BatchMutation,TRegionInfo
from hbase.ttypes import IOError,AlreadyExists

socket = TSocket.TSocket('$thriftIP', $port)
socket.setTimeout(5000)

transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)

client = Hbase.Client(protocol)
transport.open()

mutation1 = [Mutation(column = "cf:a",value = "value1")]
client.mutateRow('thrift_test_1',"row1",mutation1)

mutation2 = [Mutation(column = "cf:b",value = "value2")]
client.mutateRow('thrift_test_1',"row1",mutation2)

mutation1 = [Mutation(column = "cf:a",value = "value3")]
client.mutateRow('thrift_test_1',"row2",mutation1)
```

```
mutation2 = [Mutation(column = "cf:b",value = "value4")]
client.mutateRow('thrift_test_1',"row2",mutation2)

socket.close()
```

该程序会在 HBase 的 thrift_test_1 表中添加两行数据，每行分别有两个数据，可以在 HBase Shell 中查看插入的数据：

```
hbase(main):005:0> scan 'thrift_test_1'
ROW        COLUMN+CELL
row1        column=cf:a, timestamp=1530697238581, value=value1
row1        column=cf:b, timestamp=1530697238587, value=value2
row2        column=cf:a, timestamp=1530704886969, value=value3
row2        column=cf:b, timestamp=1530704886975, value=value4
2 row(s) in 0.0190 seconds
```

使用 Python 查看 HBase 表中的数据

有两种查看方式，一种是查看一行，一种是使用 Scan 来查看全部数据，新建一个 Python 文件 Scan_table.py，加入以下代码：

```
#! /usr/bin/env python
#coding=utf-8

from thrift import Thrift
from thrift.transport import TSocket,TTransport
from thrift.protocol import TBinaryProtocol
from hbase import Hbase
from hbase.ttypes import ColumnDescriptor,Mutation,BatchMutation,TRegionInfo
from hbase.ttypes import IOError,AlreadyExists

socket = TSocket.TSocket('$thriftIP', $port)
socket.setTimeout(5000)

transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)

client = Hbase.Client(protocol)
transport.open()

result1 = client.getRow("thrift_test_1","row1")
print result1
for r in result1:
        print 'the rowname is ',r.row
        print 'the frist value is ',r.columns.get('cf:a').value
        print 'the second value is ',r.columns.get('cf:b').value

scanId = client.scannerOpen('thrift_test_1',"",["cf"])
result2 = client.scannerGetList(scanId,10)
print result2

client.scannerClose(scanId)
socket.close()
```

其中使用 GetRow 来取得一行的数据，使用 scannerGetList 来得到整个表格中的数据，运行该程序后输出如下：

```
[hadoop@172 hbase]$ python Scan_table.py
[TRowResult(columns={'cf:a': TCell(timestamp=1530697238581, value='value1'), 'cf:b':
TCell(timestamp=1530697238587, value='value2')}, row='row1')]
the rowname is  row1
the frist value is  value1
the second value is  value2
```

```
[TRowResult(columns={'cf:a': TCell(timestamp=1530697238581, value='value1'), 'cf:b':
TCell(timestamp=1530697238587, value='value2')}, row='row1'), TRowResult(columns={'cf:a':
TCell(timestamp=1530704886969, value='value3'), 'cf:b': TCell(timestamp=1530704886975, value='value4')},
row='row2')]
```

分别输出了第一行的数据和整个表中的数据。

使用 Python 删除 HBase 中的数据

新建一个 Python 文件 Delete_row.py，把以下代码加入其中：

```
#! /usr/bin/env python
#coding=utf-8

from thrift import Thrift
from thrift.transport import TSocket,TTransport
from thrift.protocol import TBinaryProtocol
from hbase import Hbase
from hbase.ttypes import *

socket = TSocket.TSocket('$thriftIP',$port)
socket.setTimeout(5000)

transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)

client = Hbase.Client(protocol)
transport.open()

client.deleteAllRow("thrift_test_1","row2")

socket.close()
```

该程序会删除测试表中的第二行数据，运行后可以在 HBase Shell 中查看该表中的内容：

```
[hadoop@172 hbase]$ python Delete_row.py
[hadoop@172 hbase]$ hbase shell

hbase(main):004:0> scan 'thrift_test_1'
ROW     COLUMN+CELL
 row1      column=cf:a, timestamp=1530697238581, value=value1
 row1      column=cf:b, timestamp=1530697238587, value=value2
1 row(s) in 0.2050 seconds
```

此时表中只剩第一行中的数据。
更多关于 Thrift 的操作详见 如何使用 Thrift 。

Spark On HBase

最近更新时间：2024-08-15 14:44:01

关于 Spark on HBase 的详细操作，具体可查看 Github 主页 Spark-HBase Connector 。

MapReduce On HBase

最近更新时间：2024-08-15 14:44:01

关于 MapReduce on HBase 的读写操作等内容，具体可参见 使用示例 。

Phoenix on HBase 开发指南
Phoenix 客户端使用

最近更新时间：2024-08-15 14:44:01

Phoenix 查询引擎支持使用 SQL 进行 HBase 数据的查询，会将 SQL 查询转换为一个或多个 HBase API，协同处理器与自定义过滤器的实现，并编排执行。使用 Phoenix 进行简单查询，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。**EMR 中选择 HBase 组件的集群，默认集成 phoenix 客户端。**

1. **启动客户端**

   切换成 hadoop 用户，进入/usr/local/service/hbase/phoenix-client/bin 目录，使用 Phoenix 的 Python 命令行工具：

   ```
   ./sqlline.py
   ```

   执行成功后显示：

   ```
   [hadoop@172 /usr/local/service/hbase/phoenix-client/bin]$ ./sqlline.py
   Setting property: [incremental, false]
   Setting property: [isolation, TRANSACTION_READ_COMMITTED]
   issuing: !connect -p driver org.apache.phoenix.jdbc.PhoenixDriver -p user "none" -p password "none" "jdbc:phoenix:"
   SLF4J: Class path contains multiple SLF4J bindings.
   SLF4J: Found binding in [jar:file:/usr/local/service/hbase/phoenix-client/phoenix-client-hbase-2.4-5.1.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
   SLF4J: Found binding in [jar:file:/usr/local/service/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.25.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]
   Connecting to jdbc:phoenix:
   Connected to: Phoenix (version 5.1)
   Driver: PhoenixEmbeddedDriver (version 5.1)
   Autocommit status: true
   Transaction isolation: TRANSACTION_READ_COMMITTED
   sqlline version 1.9.0
   0: jdbc:phoenix:>
   ```

2. **Phoenix 引擎支持使用 SQL 进行数据查询，一些常见操作如下：**

   - **创建表**

     ```
     0: jdbc:phoenix:> CREATE TABLE IF NOT EXISTS TEST (
      host char(50) not null,
      txn_count bigint
      CONSTRAINT pk PRIMARY KEY (host)
     );
     ```

   - **插入数据**

     ```
     0: jdbc:phoenix:>UPSERT INTO TEST(host,txn_count) VALUES('192.168.1.1',1);
     0: jdbc:phoenix:>UPSERT INTO TEST(host,txn_count) VALUES('192.168.1.2',2);
     ```

   - **查询数据**

     ```
     0: jdbc:phoenix:>SELECT * FROM TEST;
     ```

   - **删除数据表**

     ```
     0: jdbc:phoenix:>DROP TABLE IF EXISTS TEST;
     ```

更多操作及说明，可参考 社区文档 。

Phoenix JDBC 使用

最近更新时间：2025-03-17 11:34:02

新建 Maven 工程

在本地 shell 下进入您想要新建工程的目录，例如 D://mavenWorkplace 中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=$yourgroupID -DartifactId=$yourartifactID -DarchetypeArtifactId=maven-archetype-quickstart
```

其中 **$yourgroupID** 即为您的包名。**$yourartifactID** 为您的项目名称，而 **maven-archetype-quickstart** 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。
创建成功后，在 D://mavenWorkplace 目录下就会生成一个名为 $yourartifactID 的工程文件夹。其中的文件结构如下所示：

```
simple
    ---pom.xml        核心配置，项目根下
    ---src
        ---main
            ---java          Java 源码目录
            ---resources     Java 配置文件目录
        ---test
             ---java          测试源码目录
             ---resources     测试配置目录
```

其中我们主要关心 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。

添加phoenix依赖

```
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <phoenix.version>5.1.2</phoenix.version>
</properties>

<dependency>
      <groupId>org.apache.phoenix</groupId>
      <artifactId>phoenix-hbase-compat-2.3.0</artifactId>
      <version>${phoenix.version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.phoenix</groupId>
      <artifactId>phoenix-core</artifactId>
      <version>${phoenix.version}</version>
    </dependency>
<build>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
```

```
      </descriptorRefs>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

其中，phoenix.version 与集群中 phoenix 版本保持一致。

创建 JDBC 连接对象

```
package Phoenix;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class PhoenixJdbc {

    static {
        try {
            Class.forName("org.apache.phoenix.jdbc.PhoenixDriver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        String createSql = "create table if not exists user_tb(id VARCHAR,uname VARCHAR  CONSTRAINT pk
PRIMARY KEY (id))";
        Connection connection = null;
        try {

            String ipList = "${zk_ip1},${zk_ip2},${zk_ip3}:2181";
            String znodeParent = "${znodeParent}";
            String url = "jdbc:phoenix:" + ipList + ":" +znodeParent;
            connection = DriverManager.getConnection(url);

            Statement statement = connection.createStatement();

            statement.executeUpdate(createSql);
            statement.executeUpdate("upsert into user_tb values ('1','张三')");
            statement.executeUpdate("upsert into user_tb values ('2','李四')");
            connection.commit();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
    }
}
```

其中 ${zk_ip1} , ${zk_ip2} , ${zk_ip3} 为 Zookeeper 节点内网 ip，2181 为 Zookeeper 默认监听端口；${znodeParent} 为 hbase 中 hbase-site.xml 配置文件里 zookeeper.znode.parent 参数的值。

编译代码并打包上传

如果您的 Maven 配置正确并且成功的导入了依赖包，那么整个工程应该没有错误可以直接编译。在本地命令行模式下进入工程目录，执行下面的命令对整个工程进行打包：

```
mvn package
```

上传到EMR集群的 HBase 节点，在本地命令行模式下运行：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，$localfile 是您的本地文件的路径加名称；root 为 CVM 服务器用户名；公网 IP 可以在 EMR 控制台的节点信息中或者在云服务器控制台查看；$remotefolder 是您想存放文件的 CVM 服务器路径。上传完成后，在 EMR 命令行中即可查看对应文件夹下是否有相应文件。

运行样例

首先需要登录 EMR 集群中的 HBase 组件节点，建议登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 hadoop 用户：

```
su hadoop
```

然后进入您存放 jar 包的文件夹下，执行以下指令：

```
java -cp $jar包名 $类全限定名
```

其中 **$jar包名**为先前上传的 EMR 节点的包名，**$类全限定名**为 java main 方法的全限定名 package.mainclass。

Phoenix 实践教程

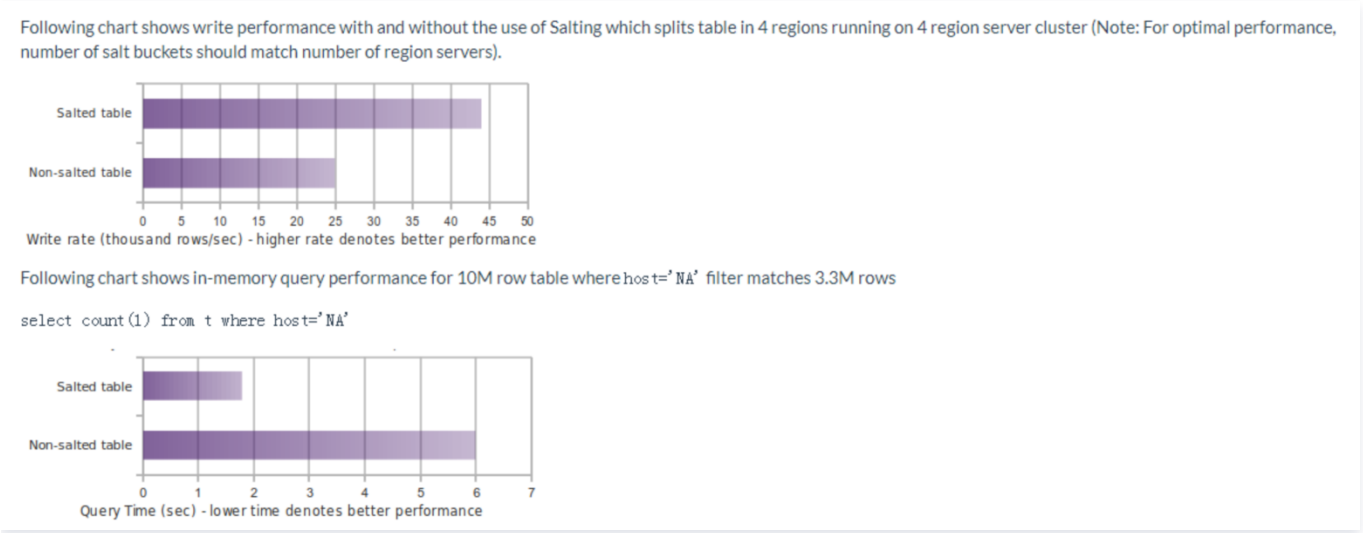
最近更新时间：2025-03-17 11:34:02

使用 Phoenix salted 表

HBase 的 Row Key 假如没有经过精心设计，如果它又是自增长的，那么顺序写很可能会导致热点问题。Phoenix 提供了一种透明的方法，关联 salt 和 RowKey 到一张指定表的方案。只需要在创建表的时候添加 SALT_BUCKETS 关键字，这个值的范围是1到256。例如：

```
0: jdbc:phoenix:>CREATE TABLE my_table (a_key VARCHAR PRIMARY KEY, a_col VARCHAR) SALT_BUCKETS = 20;
```

可以免除使用 HBase API 时需要精心设计 RowKey 的麻烦。如果用户对 HBase Row Key 的设计理解不够，建议使用 salted 表。Phoenix 官方提供的写和查询，salted 表和非 salted 表性能对比：



更多 salted 性能或者操作说明，可查看 Phoenix salted 表 社区文档 。

Phoenix 二级索引

对于 HBase 而言，如果想精确地定位到某行记录，唯一的办法是通过 rowkey 来查询。如果不通过 rowkey 来查找数据，就必须逐行地比较每一列的值，即全表扫描。对于较大的表，全表扫描的代价是不可接受的。但是，很多情况下，需要从多个角度查询数据。这需要二级索引（secondary index）来完成这件事。二级索引的原理很简单，但是如果自己维护的话则会麻烦一些。

Phoenix 二级索引配置

EMR 中 Phoenix 直接可支持 Phoenix 的二级索引。如果需要使用非事务性的，可变的索引只需按照以下步骤配置即可。打开 EMR 组件管理页面，单击 **HBase**，选择**配置 > 配置管理**，增加 hbase-site.xml 的 `hbase.regionserver.wal.codec` 、`hbase.region.server.rpc.scheduler.factory.class` 和 `hbase.rpc.controllerfactory.class` 三个配置项即可，详细配置如下：

```
  <property>
    <name>hbase.regionserver.wal.codec</name>
    <value>org.apache.hadoop.hbase.regionserver.wal.IndexedWALEditCodec</value>
    </property>
  <property>
    <name>hbase.region.server.rpc.scheduler.factory.class</name>
    <value>org.apache.hadoop.hbase.ipc.PhoenixRpcSchedulerFactory</value>
    <description>Factory to create the Phoenix RPC Scheduler that uses separate queues for index and
metadata updates</description>
  </property>
  <property>
    <name>hbase.rpc.controllerfactory.class</name>
    <value>org.apache.hadoop.hbase.ipc.controller.ServerRpcControllerFactory</value>
    <description>Factory to create the Phoenix RPC Scheduler that uses separate queues for index and
metadata updates</description>
```

```
    </property>
```

Phoenix 二级索引使用

创建二级索引，命令如下：

```
0: jdbc:phoenix:>CREATE INDEX my_index ON my_table (v1) INCLUDE (v2);
```

更多二级索引操作说明，可查看 Phoenix 二级索引社区文档 。

Hive 开发指南
Hive 概述

最近更新时间：2023-09-08 16:01:01

Hive 是一个建立在 Hadoop 文件系统上的数据仓库架构，它为数据仓库的管理提供了许多功能，包括数据 ETL（抽取、转换和加载）工具、数据存储管理和大型数据集的查询和分析能力。同时 Hive 还定义了类 SQL 的开发语言，允许用户将结构化的数据文件映射为一张数据库表，并提供简单的 SQL 查询功能。

- EMR 中 Hive 安装在路径在 EMR 节点的 /usr/local/service/hive 路径下。
- 关于 Hive 更多的介绍，可以参考 Apache Hive官网 。

Hive 服务角色

角色名称	说明
HiveServer2	Hive 的 ThriftServer 服务，用于接收客户端的查询请求并进行 SQL 编译及解析，支持多客户端并发以及身份验证。 一个 EMR 集群可部署多个 HiveServer2，支持拓展至 Router 节点并配置负载均衡。
Hive MetaStore	Hive 的元数据服务， 用于维护 Hive Database 和 Hive Table 的元数据信息。该模块的元数据管理能力也被 Spark 、Trino 等引擎所集成。 一个 EMR 集群可部署多个 Hive MetaStore，支持拓展至 Router 节点。
Hive Client	Hive 客户端，提供 Beeline、JDBC 等应用驱动，可以向 HiveServer2 提交 SQL 作业。部署 Hive 服务的节点均会进行安装。
Hive WebHCat	WebHCat 是为 HCatalog 提供 REST API 的服务，提供 Rest 接口，通过 Rest 执行 Hive 命令，提交 MapReduce 任务。 一个集群内可部署多个 WebHCat，支持拓展至 Router 节点。

Hive 内部表与外部表

- **内部表：** Hive 管理内部表的元数据以及表的实际数据，使用 DROP 语法删除内部表的时候，表的元数据及对应的数据都会被删除。创建内部表后会将 HDFS 的文件映射成 Table，然后 Hive 的数据仓库生成对应的目录，EMR 默认的仓库路径为 /usr/hive/warehouse/${tablename}，这个路径在 HDFS 上面，其中 ${tablename} 是您创建的表名。
- **外部表：** Hive 中的外部表和内部表很类似，但是其数据不是放在自己表所属的目录中，而是存放到其他地方。这样的好处是如果您要删除这个外部表，此外部表所指向的数据是不会被删除的，它只会删除外部表对应的元数据。

Hive 语法

弹性 MapReduce 中的 Hive 完全兼容开源社区语法，可通过 HiveQL社区语法手册 进行查阅。

基础使用
Hive 基础操作

最近更新时间：2023-09-08 16:30:51

本文为您演示如何使用 EMR 上的 Hive 创建库表、导入数据、执行查询等基础操作。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考 创建集群 。
- 在创建 EMR 集群时在软件配置界面选择 Hive 组件。
- 示例中存在需要访问腾讯云对象存储 COS 的可选内容，可参考 创建存储桶 在 COS 中创建一个存储桶（Bucket），并于 EMR 控制台 实例信息 页面开启对象存储授权。

> **说明：**
> - 登录 EMR 节点的方式可参考登录 Linux 实例 。在集群详情页中选择 **集群资源 > 资源管理**，单击对应节点资源 ID 进入云服务器列表，单击右侧**登录**，即可使用 WebShell 登录实例。
> - 登录 Linux 实例用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
> - 本文操作均是以 hadoop 用户进行，请在登录命令行界面后切换用户身份。

准备样例数据

登录到 Master 节点，在 EMR 命令行使用以下命令切换到 hadoop 用户，并进入 Hive 文件夹：

```
su hadoop
cd /usr/local/service/hive
```

新建一个 bash 脚本文件 gen_data.sh，在其中添加以下代码：

```
#!/bin/bash
MAXROW=1000000 #指定生成数据行数
for((i = 0; i < $MAXROW; i++))
do
    echo $RANDOM, \"$RANDOM\"
done
```

并按如下方式赋权并执行脚本，该脚本文件会生成 1000000 个随机数对，并且保存到文件 hive_test.data 中：

```
chmod +x 脚本名称
./gen_data.sh > hive_test.data
```

使用如下命令把生成的测试数据先上传到 HDFS 中，其中 ${hdfspath} 为 HDFS 上的您存放文件的路径：

```
hdfs dfs -put ./hive_test.data /${hdfspath}
```

也可以使用 COS 上面的数据。将数据上传到 COS 中，如果数据在本地，那么可以使用 COS 控制台来上传数据。如果数据在 EMR 集群，那么使用如下命令来上传数据，其中 ${bucketname} 为您创建的 COS 桶名：

```
hdfs dfs -put ./hive_test.data cosn://${bucketname}/
```

Hive 基础操作

登录 EMR 集群的 Master 节点，切换 hadoop 用户并通过 Hive 客户端进入 Hive 命令行：

```
hive
```

创建库表

使用 SHOW 语法展示当前所有数据库：

```
hive> show databases;
OK
default
Time taken: 0.26 seconds, Fetched: 1 row(s)
```

使用 CREATE DATABASE 语法创建一个数据库 test：

```
hive> create database if not exists test;
OK
Time taken: 0.176 seconds
```

使用 USE 语法转到刚刚创建的 test 数据库下：

```
hive> use test;
OK
Time taken: 0.176 seconds
```

使用 CREATE TABLE语法在 test 数据库下创建一个新的名为 hive_test 的内部表：

```
hive> create table hive_test (a int, b string)
hive> ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
-- 创建数据表 hive_test, 并指定列分割符为','
OK
Time taken: 0.204 seconds
```

最后可用 SHOW TABLES 语法查看表是否创建成功：

```
hive> show tables;
OK
hive_test
Time taken: 0.176 seconds, Fetched: 1 row(s)
```

导入数据

对于存放在 HDFS 中的数据，使用如下指令来将其导入表中：

```
hive> load data inpath "/${hdfspath}/hive_test.data" into table hive_test;
```

对于存放在 COS 中的数据，使用如下指令来将其导入表中：

```
hive> load data inpath "cosn://${bucketname}/hive_test.data" into table hive_test;
```

也可以将存放在 EMR 集群本地的数据导入到 Hive 中，使用如下指令：

```
hive>load data local inpath "/${localpath}/hive_test.data" into table hive_test;
```

> **说明：**
> 其中 ${hdfspath} 为 HDFS 上的您存放文件的路径，${bucketname} 为您的 COS 桶名， ${localpath} 为您的 EMR 集群本地存放数据的路径。导入完成后，源数据会被删除。

执行查询

查询表中前10个元素：

```
hive> select * from hive_test limit 10;
OK
30847   "31583"
14887   "32053"
19741   "16590"
8104    "20321"
29030   "32724"
27274   "5231"
10028   "22594"
924 "32569"
10603   "27927"
4018    "30518"
Time taken: 2.133 seconds, Fetched: 10 row(s)
```

统计表中一共有多少行数据：

```
hive> select count(*) from hive_test;
OK
1000000
Time taken: 18.504 seconds, Fetched: 1 row(s)
```

删除库表

使用 DROP TABLE 语法来删除 Hive 表：

```
hive> drop table if exists hive_test;
Moved: 'hdfs://HDFS/usr/hive/warehouse/hive_test' to trash at: hdfs://HDFS/user/hadoop/.Trash/Current
OK
Time taken: 2.327 seconds
```

使用 DROP DATABASE 语法来删除 Hive 库：

```
hive> drop database if exists test;
OK
Time taken: 0.531 seconds
```

Hive 连接方式

最近更新时间：2024-11-18 16:49:42

本文为您介绍在弹性 MapReduce 中使用 Hive 客户端、Beeline、Java 三种方式连接 Hive。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情请参见 创建集群 。
- 在创建 EMR 集群时在软件配置界面选择 Hive 组件。

> **说明：**
> - 登录 EMR 节点的方式可参考 登录 Linux 实例 。在集群详情页中选择 **集群资源 > 资源管理**，单击对应节点资源 ID 进入云服务器列表，单击右侧 **登录** 即可使用 WebShell 登录实例。
> - 登录 Linux 实例用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
> - 本文操作均是以 hadoop 用户进行，请在登录命令行界面后使用 su hadoop 命令切换用户身份。

连接Hive

Hive 服务默认部署在 Master 节点，您也可以参考 集群扩容 将 HiveServer2 部署至 Router 节点，本文以 Master 节点为示例连接 Hive 服务。

方式一：通过 Hive 客户端

登录 EMR 集群的 Master 节点，切换到 Hadoop 用户，执行以下命令即可进入 Hive 命令行：

```
hive
```

您也可以使用 -h 参数来获取 Hive 指令的基本信息。

方式二：通过 Beeline 连接 HiveServer2

登录 EMR 的 Master 节点，通过 beeline 命令连接 Hive:

```
beeline -u "jdbc:hive2://${hs2_ip}:${hs2_port}" -n hadoop
```

> **说明：**
> 1. ${hs2_ip} 为集群 HiveServer2 服务部署节点的内网 ip，您可以在集群详情页中 **集群服务 > Hive > 角色管理** 中查看。
> 2. ${hs2_port} 为集群 HiveServer2 的端口号，默认值是 7001，您可以在集群详情页中 **集群服务 > Hive > 配置管理** 中 hive-site.xml 配置文件的 hive.server2.thrift.port 配置项中查看。

方式三：通过 Java 连接 Hive

本文以 Maven 来为示例管理您的工程。Maven 是一个项目管理工具，能够帮助您方便的管理项目的依赖信息，即它可以通过 pom.xml 文件的配置获取 jar 包，而不用去手动添加。
首先在本地 下载并安装 Maven，配置好 Maven 的环境变量，如果您使用 IDE，请在 IDE 中设置好 Maven 相关配置。
在本地 shell 下进入要新建工程的目录，例如/tmp/mavenWorkplace中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=${yourgroupID} -DartifactId=${yourartifactID} -DarchetypeArtifactId=maven-archetype-quickstart
```

> **说明：**
> 1. ${yourgroupID} 即为您的包名；${yourartifactID} 为您的项目名称；
> 2. maven-archetype-quickstart 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。

其中我们主要关心 pom.xml 文件和 main 下的 java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。
首先在 pom.xml 文件中配置项目依赖（ hadoop-common 和 hive-jdbc ）：

```
<dependencies>
       <dependency>
           <groupId>org.apache.hive</groupId>
           <artifactId>hive-jdbc</artifactId>
           <version>${hive_version}</version>
       </dependency>
       <dependency>
           <groupId>org.apache.hadoop</groupId>
           <artifactId>hadoop-common</artifactId>
           <version>${hadoop_version}</version>
       </dependency>
</dependencies>
```

说明：
其中 ${hive_version} 为您集群中的 Hive 版本，${hadoop_version} 为您集群中的 Hadoop 版本。

继续在 pom.xml 中添加打包和编译插件：

```
<build>
<plugins>
 <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
     <source>1.8</source>
     <target>1.8</target>
     <encoding>utf-8</encoding>
   </configuration>
 </plugin>
 <plugin>
   <artifactId>maven-assembly-plugin</artifactId>
   <configuration>
     <descriptorRefs>
     <descriptorRef>jar-with-dependencies</descriptorRef>
     </descriptorRefs>
   </configuration>
   <executions>
     <execution>
       <id>make-assembly</id>
       <phase>package</phase>
       <goals>
         <goal>single</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
</plugins>
</build>
```

在 src>main>java 下右键新建一个 Java Class，输入您的 Class 名，本示例使用 App.java，在 Class 添加样例代码：

```
package org.example;

import java.sql.*;
/**
 * Created by tencent on 2023/8/11.
 */
public class App {
    private static final String DRIVER_NAME = "org.apache.hive.jdbc.HiveDriver";
    public static void main(String[] args) throws SQLException {
```

```
        try {
            // 加载hive-jdbc驱动
            Class.forName(DRIVER_NAME);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }
        // 根据连接信息和账号密码获取连接
        Connection conn = DriverManager.getConnection("jdbc:hive2://$hs2_ip:$hs2_port/default", "hadoop",
"");
        // 创建状态参数（使用conn.prepareStatement(sql)预编译sql防止sql注入，但常用于参数化执行sql，批量执行不同的sql建议
使用下面这种方式）
        Statement stmt = conn.createStatement();
        // 以下是执行简单的建表和增查操作
        String tableName = "hive_test";
        stmt.execute("drop table if exists " + tableName);
        stmt.execute("create table " + tableName + " (key int, value string)");
        System.out.println("Create table success!");
        // show tables
        String sql = "show tables '" + tableName + "'";
        System.out.println("Running: " + sql);
        ResultSet res = stmt.executeQuery(sql);
        if (res.next()) {
            System.out.println(res.getString(1));
        }
        // describe table
        sql = "describe " + tableName;
        System.out.println("Running: " + sql);
        res = stmt.executeQuery(sql);
        while (res.next()) {
            System.out.println(res.getString(1) + "\t" + res.getString(2));
        }
        sql = "insert into " + tableName + " values (42,\"hello\"),(48,\"world\")";
        stmt.execute(sql);
        sql = "select * from " + tableName;
        System.out.println("Running: " + sql);
        res = stmt.executeQuery(sql);
        while (res.next()) {
            System.out.println(res.getInt(1) + "\t" + res.getString(2));
        }
        sql = "select count(1) from " + tableName;
        System.out.println("Running: " + sql);
        res = stmt.executeQuery(sql);
        while (res.next()) {
            System.out.println(res.getString(1));
        }
    }
}
```

程序会先连接 HiveServer2 服务，然后在 default 数据库中建立一个名为 hive_test 的表。之后在该表中插入两个元素，并输出整个表的内容。
如果您的 Maven 配置正确并且成功的导入了依赖包，那么整个工程即可直接编译。在本地 shell 下进入工程目录，执行下面的命令对整个工程进行打包：

```
mvn clean package -DskipTests
```

运行过程中可能还需要下载一些文件，直到出现 build success 表示打包成功。然后您可以在工程目录下的 target 文件夹中看到打好的 jar 包。

上传并运行程序

首先需要把压缩好的 jar 包上传到 EMR 集群中，使用 scp 或者 sftp 工具来进行上传。在本地 shell 下运行：（输入 yes 后输入密码验证）

```
scp ${localfile} root@${master_public_ip}:/usr/local/service/hive
```

> **说明：**
> 1. ${localfile} 是您的本地文件的路径加名称，root 为 CVM 服务器用户名，公网 IP 可以在 EMR 控制台的节点信息中或者在云服务器控制台查看。
> 2. ${master_public_ip} 是您集群 Master 节点的公网IP。

将打好的 jar 包上传到 EMR 集群的 /home/hadoop/ 目录下。上传完成后，在 EMR 命令行中即可查看对应文件夹下是否有相应文件。**一定要上传具有依赖的 jar 包。**
登录 EMR 集群切换到 hadoop 用户并且进入目录 /home/hadoop/。执行程序：

```
yarn jar ./hive-test-1.0-SNAPSHOT-jar-with-dependencies.jar org.example.App
```

> **说明：**
> 其中 ./hive-test-1.0-SNAPSHOT-jar-with-dependencies.jar 为您的 jar 包的路径 + 名字，org.example.App 为之前的 Java Class 的包名 + 类名。

运行结果如下：

```
Create table success!
Running: show tables 'hive_test'
hive_test
Running: describe hive_test
key     int
value   string
Running: select * from hive_test
42      hello
48      world
Running: select count(1) from hive_test
2
```

方式四：通过 Python 连接 Hive

本文示例使用 PyHive项目 通过 Python 3 进行 hive 连接。
首先登录 EMR 集群的 Master 节点，切换到 root 用户，进入 /usr/local/service/hive/ 目录下执行命令安装需要的工具及依赖包：

```
pip3 install sasl
pip3 install thrift
pip3 install thrift-sasl
pip3 install pyhive
```

安装完成后切换回 hadoop 用户。然后在 /usr/local/service/hive/ 目录下新建一个 Python 文件 hivetest.py，并且添加以下代码：

```
from pyhive import hive

import sys

default_encoding = 'utf-8'

conn = hive.connect(host='${hs2_host}',
                    port='${hs2_port}',
                    username='hadoop',
                    password='hadoop',
                    database='default',
                    auth="CUSTOM",)


tablename = 'HiveByPython'
cur = conn.cursor()

print("\n")
```

```
print('show the tables in default: ')
cur.execute('show tables')
for i in cur.fetchall():
        print(i)

cur.execute('drop table if exists ' + tablename)
cur.execute('create table ' + tablename + ' (key int,value string)')

print("\n")
print('show the new table: ')
cur.execute('show tables ' +"'" +tablename+"'")
for i in cur.fetchall():
        print(i)

print("\n")
print("contents from " + tablename + ":")
cur.execute('insert into ' + tablename + ' values (42,"hello"),(48,"world")')
cur.execute('select * from ' + tablename)
for i in cur.fetchall():
        print(i)
```

该程序连接 HiveServer2 后，首先输出所有的数据库，然后显示“default”数据库中的表。创建一个名为“hivebypython”的表，在表中插入两个数据并输出。

说明：

1. ${hs2_host} 为集群 HiveServer2 的 hostID，您可以在集群详情页中 **集群服务 > Hive > 配置管理** 中 hive-site.xml 配置文件的 hive.server2.thrift.bind.host 配置项中查看。
2. ${hs2_port} 为集群 HiveServer2 的端口号，默认值是 7001，您可以在集群详情页中 **集群服务 > Hive > 配置管理** 中 hive-site.xml 配置文件的 hive.server2.thrift.port 配置项中查看。

保存后直接运行该程序：

```
python3 hivetest.py
```

可以看到命令行输出以下信息：

```
show the tables in default:

show the new table:
('hivebypython',)

contents from HiveByPython:
(42, 'hello')
(48, 'world')
```

配置 Hive 执行引擎

最近更新时间：2023-11-07 10:38:51

Hive 支持 MapReduce、Tez、Spark 作为执行引擎，弹性 MapReduce 中默认为 MapReduce ，本文为您介绍如何在 Hive 中配置执行引擎。
一般情况下建议您使用执行引擎 Tez 替代 MapReduce，这样您会获得更好的计算效率。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考 创建集群 。
- 在创建 EMR 集群时在软件配置界面选择 Hive 组件，如果需要切换 Tez 或 Spark 执行引擎，在创建集群时需要勾选对应组件。

配置引擎

登录 Hive 客户端，具体操作参考 Hive 连接方式，Hive 的执行引擎由 hive.execution.engine 参数配置，默认执行引擎为 mr，执行如下命令可查看当前执行引擎：

```
hive> set hive.execution.engine;
hive.execution.engine=mr
```

在命令行中切换执行引擎

通过以下指令可以修改本次客户端会话中的执行引擎 。
修改执行引擎为 Tez：

```
hive> set hive.execution.engine=tez;
```

修改执行引擎为 Spark：

```
hive> set hive.execution.engine=spark;
```

执行完成后可以输入以下命令进行验证：

```
hive> set hive.execution.engine;
```

返回信息为您设置的执行引擎则说明配置成功。

> **说明：**
> 以上切换执行引擎的操作只对单个客户端会话生效，由其他客户端进入或者退出后重新连接会恢复默认执行引擎。

在配置文件中切换执行引擎

全局切换默认执行引擎可通过 EMR 控制台 **集群服务 > HIVE > 配置管理** 页面，找到 hive-site.xml 配置文件，设置以下参数值：
修改执行引擎为 Tez：

```
hive.execution.engine = tez
```

修改执行引擎为 Spark：

```
hive.execution.engine = spark
```

执行保存 **保存配置 > 保存并下发，并重启 HiveServer2**，执行完成后即可修改 Hive 的默认执行引擎。

高阶使用
配置 LDAP 认证

最近更新时间：2023-09-08 16:30:52

本文为您介绍弹性 Mapreduce 中 Hive 开启 LDAP 配置及使用。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考 创建集群 。
- 创建 Hadoop 默认场景集群并在软件配置界面选择 Hive 组件。

开启 LDAP 认证

进入 EMR 控制台 ，点击 **集群服务 > HIVE > 配置管理**，选择 hive-site.xml 配置文件，新增以下配置项并设置参数值，保存配置并下发，然后重启 HiveServer2：

参数	值	备注
hive.server2.authentication	LDAP	设置 authentication 认证机制为 LDAP。
hive.server2.authentication.ldap.url	ldap://${ldap_ip}:389	指定 LDAP 服务的 url。 ${ldap_ip} 为 OPENLDAP 服务所在节点的 ip，可以在 EMR 控制台 **集群服务 > OPENLADP** 下获取。 自建 LDAP 服务根据实际情况填写。
hive.server2.authentication.ldap.baseDN	ou=People,dc=emr,dc=cloud,dc=tencent,dc=com	EMR 中 LDAP 服务用户所在的 Base DN，自建 LDAP 服务请根据实际情况填写。
hive.server2.authentication.ldap.guidKey	cn	EMR 中 LDAP 服务用户名格式，自建 LDAP 服务请根据实际情况填写。

访问 HiveServer2

开启 LDAP 认证后，访问 HiveServer2 需要提供 LDAP 的用户名和密码。

Beeline 客户端连接 Hive

```
beeline -u "jdbc:hive2://${hs2_ip}:${hs2_port}"  -n ${user} -p ${password}
```

JDBC 连接 Hive

```
jdbc:hive2://${hs2_ip}:${hs2_port}/default;user=${user};password=${password}
```

说明：

1. ${user} 为 LDAP 的用户名。
2. ${password} 为 LDAP 的密码。
3. ${hs2_ip} 为集群 HiveSever2 服务部署节点的内网ip，您可以在集群详情页中 **集群服务>Hive>角色管理** 中查看。
4. ${hs2_port} 为集群 HiveServer2 的端口号，默认值是 7001，您可以在集群详情页中 **集群服务>Hive>配置管理** 中 hive-site.xml 配置文件的 hive.server2.thrift.port 配置项中查看。

注意：

Hive 集成 EMR OpenLdap 并新增用户后，使用新用户访问需对 hdfs 目录下 /emr/hive 赋予 644 权限。

HiveServer2 负载均衡

最近更新时间：2023-09-08 16:01:02

当 EMR 集群有多个 HiveServer2 服务时，可以借助 Zookeeper 服务实现访问 HiveServer2 的负载均衡，本文详细介绍 HiveServer2 负载均衡的使用方法。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考 创建集群 。
- 创建 Hadoop 默认场景 **高可用** 集群，并在软件配置界面选择 Hive 组件。

通过 Zookeeper 实现 hs2 负载均衡

Hadoop 默认场景中高可用集群默认安装了 Zookeeper 服务，您可以使用以下连接方式连接 HiveServer2，借助 Zookeeper 达到负载均衡的效果：

```
beeline -u 'jdbc:hive2://${hive.zookeeper.quorum}/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=
<hive.server2.zookeeper.namespace>' -n ${user} -p ${password}
```

说明：

1. ${hive.zookeeper.quorum}、${hive.server2.zookeeper.namespace}是 hive 中 Zookeer Server 相关配置，您可以在 EMR 控制台 **集群服务 > HIVE > 配置管理** 中 hive-site.xml 配置文件中查找 hive.zookeeper.quorum 、 hive.server2.zookeeper.namespace 参数对应的值。
2. user、password 为您设置的 LDAP 用户名及登录密码。

Hive 元数据管理

最近更新时间：2023-11-06 14:43:51

当选择部署 Hive 组件时，Hive 元数据库提供了两种存储方式：第一种集群默认，Hive 元数据存储于集群独立购买 MetaDB；第二种是关联外部 Hive 元数据库，可选择关联 EMR-MetaDB 或自建 MySQL 数据库，元数据将存储于关联的数据库中，不随集群销毁而销毁。
集群默认是独立自动购买一个 MetaDB 云数据库实例存储单元作为元数据存储地，与其余组件元数据一起存储，并随集群销毁而销毁 MetaDB 云数据库，若需保存元数据，需提前在云数据库中手动保存元数据。

> **注意**
> 1. Hive 元数据与 Druid、Superset、Hue、Ranger、Oozie、Presto 组件元数据一起存储。
> 2. 集群需要单独购买一个 MetaDB 作为元数据存储单元。
> 3. MetaDB 随集群销毁而销毁，即元数据随集群而销毁。

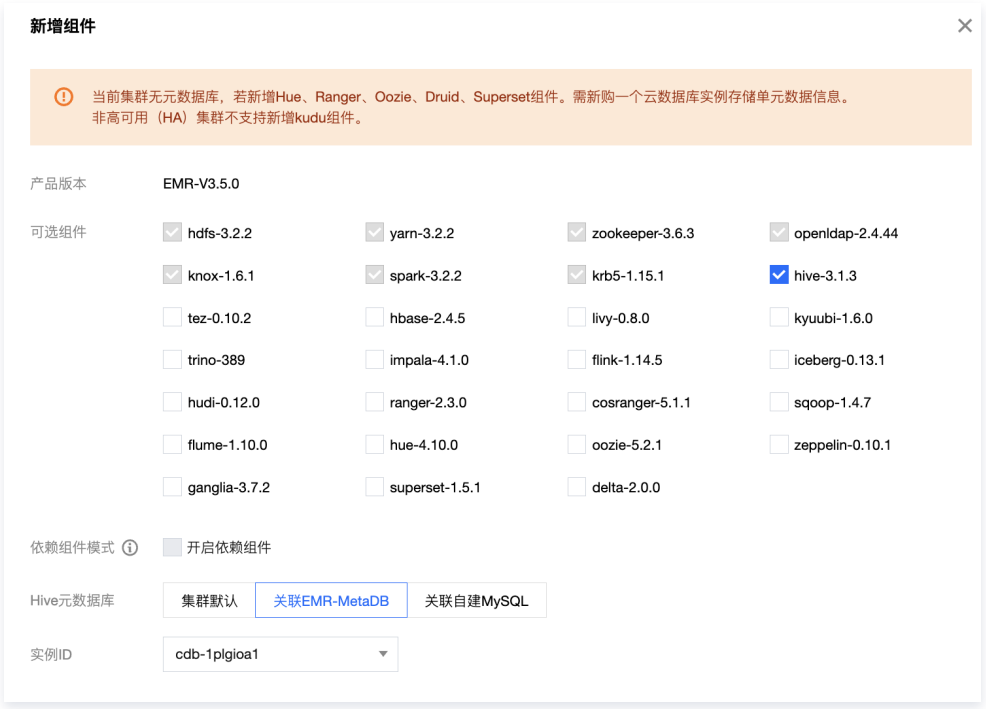
关联 EMR-MetaDB 共享 Hive 元数据

集群创建时系统会拉取云上可用的 MetaDB，用于新集群 Hive 组件存储元数据，无需单独购买 MetaDB 存储 Hive 元数据节约成本；并且 Hive 元数据不会随当前集群的销毁而销毁。

> **注意**
> 1. 可用 MetaDB 实例 ID 为同一账号下 EMR 集群中已有的 MetaDB。
> 2. 当选择 Hue、Ranger、Oozie、Druid、Superset 一个或多个组件时系统会自动购买一个 MetaDB 用于除 Hive 外的组件元数据存储。
> 3. 要销毁关联的 EMR-MetaDB 需前往云数据库销毁，销毁后 Hive 元数据库将无法恢复。
> 4. 需保持关联的 EMR-MetaDB 网络与当前新建集群在同一网络环境下。

1. 新建集群并选择 Hive 组件后，单击**下一步**并选择关联的 EMR-MetaDB：

2. 未安装 Hive 组件的集群，在新增 Hive 组件时，选择关联的 EMR-MetaDB：



关联自建 MySQL 共享 Hive 元数据

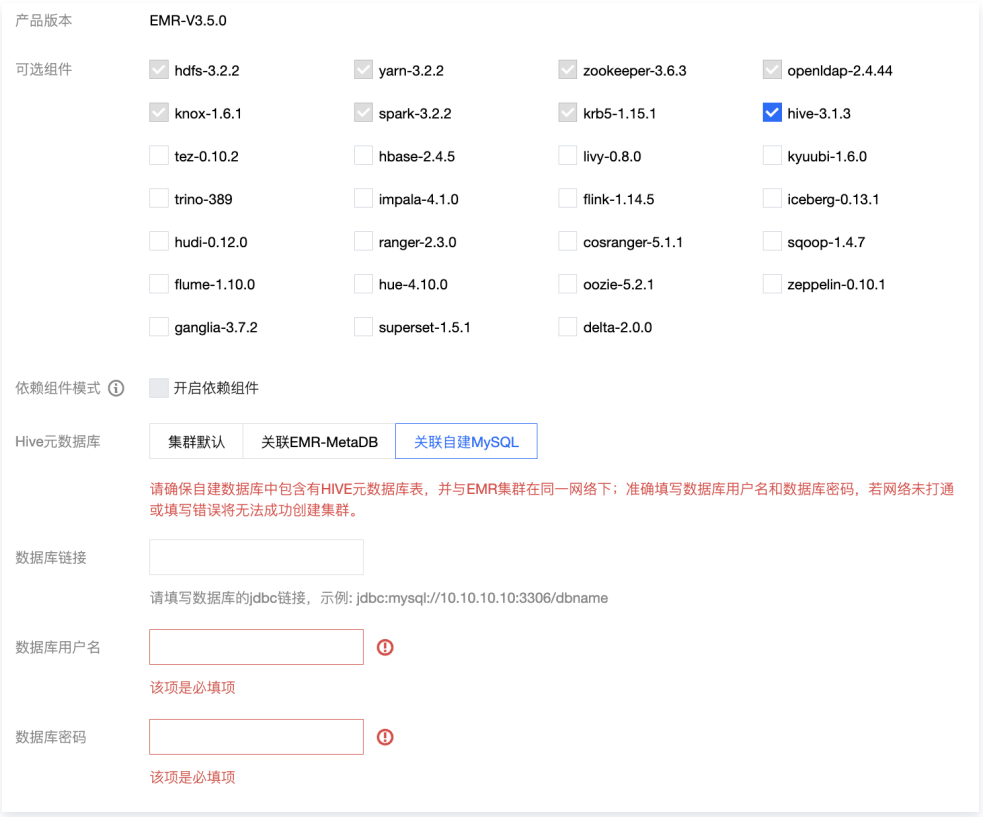
关联自己本地自建 MySQL 数据库作为 Hive 元数据存储，也无需单独购买 MetaDB 存储 Hive 元数据节约成本，需准确填写输入以“jdbc:mysql://开头”的本地地址、数据库名字、数据库登录密码，并确保网络与当前集群网络打通。

注意

1. 请确保自建数据库与 EMR 集群在同一网络下。
2. 准确填写数据库用户名和数据库密码。
3. 当选择 Hue、Ranger、Oozie、Druid、Superset 一个或多个组件时系统会自动购买一个 MetaDB 用于除 Hive 外的元数据存储。
4. 需保证自定义数据库中的 Hive 元数据版本大于等于新集群中的 Hive 版本。
5. 请确保“自建 MySQL”中包含已经初始化的 HIVE 元数据库表。

1. 新建集群并选择 Hive 组件后，单击**下一步**并关联自建的 Mysql 数据库：

2. 未安装 Hive 组件的集群，在新增 Hive 组件时，关联自建的 Mysql 数据库：



HIVE 关联自建元数据异常修复方法

由于在创建 EMR 集群时选用了关联自建 MySQL，且自建 MySQL 无 HIVE 的元数据，这会导致 HIVE 进程异常。

问题复现

解决方案

对于无数据的 hive 元数据操作步骤如下：

> **说明**
> 操作时替换成用户实际的 ${ip}、${port}、${database}。

1. 控制台将 HIVE 的 hs2 和 metastore 停掉。
2. hive 组件修改 hive-site.xml proto-hive-site.xml 下发。
   配置项：javax.jdo.option.ConnectionURL

```
jdbc:mysql://${ip}:${port}/${database}?useSSL=false&amp;createDatabaseIfNotExist=true&amp;characterEncodin
```

```
g=UTF-8
```

3. CDB 数据库里删除库操作：

```
drop database ${database};
```

4. hadoop 用户执行如下命令：

```
/usr/local/service/hive/bin/schematool -dbType mysql -initSchema
```

5. 控制台 HIVE 启动 hs2和 metastore。
6. 访问 hive 是否异常。
   如果有字符异常，CDB 再执行如下命令：

```
alter database ${database} character set latin1;
flush privileges;
```

自定义函数 UDF

最近更新时间：2023-12-15 10:33:25

本文为您介绍自定义函数（UDF）及开发和使用流程。

UDF 分类

UDF 分类	描述
UDF（User Defined Scalar Function）	自定义标量函数，通常称为 UDF 。其输入与输出是一对一的关系，即读入一行数据，写出一条输出值。
UDTF（User Defined Table-valued Function）	自定义表值函数，用来解决一次函数调用输出多行数据场景的，也是唯一一个可以返回多个字段的自定义函数。
UDAF（User Defined Aggregation Function）	自定义聚合函数，其输入与输出是多对一的关系，即将多条输入记录聚合成一条输出值，可以与 SQL 中的 Group By 语句联合使用。

更多说明可参考社区文档：UDF 、UDAF 、UDTF 。

开发 UDF

使用 IDE，创建 Maven 工程。工程基本信息如下，您可以自定义 groupId 和 artifactId：

```
<groupId>org.example</groupId>
<artifactId>hive-udf</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

添加 pom 依赖：

```
<!-- https://mvnrepository.com/artifact/org.apache.hive/hive-exec -->
<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-exec</artifactId>
    <version>3.1.3</version>
    <exclusions>
         <exclusion>
           <groupId>org.pentaho</groupId>
           <artifactId>*</artifactId>
         </exclusion>
      </exclusions>
</dependency>
```

创建一个类，类名您可以自定义，本文以 nvl 为例：
方式 1：继承 UDF 重写 evaluate 方法：

```
package org.example;

import org.apache.hadoop.hive.ql.exec.UDF;

public class nvl extends UDF {
    public String evaluate(final String s) {
        if (s == null) { return null; }
        return s + ":HelloWorld";
    }
}
```

方式 2（推荐，适用于传参情况复杂的场景）：继承 GenericUDF 重写 initialize、evaluate、getDisplayString：

```
package org.example;

import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentTypeException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDF;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDFUtils;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;

@Description(name = "nvl",
        value = "nvl(value, default_value) - Returns default value if value is null else returns value",
        extended = "Example: SELECT nvl(null, default_value);")
public class MyUDF extends GenericUDF {

    private GenericUDFUtils.ReturnObjectInspectorResolver returnOIResolver;
    private ObjectInspector[] argumentOIs;

    /**
     * 根据函数的入参类型确定出参类型
     */
    public ObjectInspector initialize(ObjectInspector[] arguments) throws UDFArgumentException {
        argumentOIs = arguments;
        if(arguments.length != 2) {
            throw new UDFArgumentException("The operator 'NVL' accepts 2 arguments.");
        }
        returnOIResolver = new GenericUDFUtils.ReturnObjectInspectorResolver(true);
        if(!(returnOIResolver.update(arguments[0]) && returnOIResolver.update(arguments[1]))) {
            throw new UDFArgumentTypeException(2, "The 1st and 2nd args of function NLV should have the same
type, "
                    + "but they are different: \""+arguments[0].getTypeName()+"\" and \"" +
arguments[1].getTypeName() + "\"");
        }
        return returnOIResolver.get();
    }

    /**
     * 计算结果，最后结果的数据类型会根据initialize方法的返回值类型确定函数的返回值类型
     */
    public Object evaluate(DeferredObject[] arguments) throws HiveException {
        Object retVal = returnOIResolver.convertIfNecessary(arguments[0].get(), argumentOIs[0]);
        if(retVal == null) {
            retVal = returnOIResolver.convertIfNecessary(arguments[1].get(), argumentOIs[1]);
        }
        return retVal;
    }

    /**
     * 获取要在explain中显示的字符串
     */
    public String getDisplayString(String[] children) {
        StringBuilder builder = new StringBuilder();
        builder.append("if ");
        builder.append(children[0]);
        builder.append(" is null ");
        builder.append("returns ");
        builder.append(children[1]);
        return builder.toString();
    }
}
```

以方式 2 为例，将自定义的代码打成 jar 包。在 pom.xml 所在目录，执行如下命令制作 jar 包。

```
mvn clean package -DskipTests
```

target 目录下会出现 hive-udf-1.0-SNAPSHOT.jar 的 jar 包，即代表完成了 UDF 开发工作。

使用UDF

将生产的 jar 上传至 EMR 集群 Master 节点：

```
scp ./target/hive-udf-1.0-SNAPSHOT.jar root@${master_public_ip}:/usr/local/service/hive
```

切换 hadoop 用户并执行以下命令将 jar 上传到 HDFS 中：

```
su hadoop
hadoop fs -put ./hive-udf-1.0-SNAPSHOT.jar /
```

查看上传到 HDFS 中的 jar：

```
hadoop fs -ls /
Found 5 items
drwxr-xr-x   - hadoop supergroup          0 2023-08-22 09:20 /data
drwxrwx---   - hadoop supergroup          0 2023-08-22 09:20 /emr
-rw-r--r--   2 hadoop supergroup       3235 2023-08-22 15:39 /hive-udf-1.0-SNAPSHOT.jar
drwx-wx-wx   - hadoop supergroup          0 2023-08-22 09:20 /tmp
drwxr-xr-x   - hadoop supergroup          0 2023-08-22 09:20 /user
```

连接 Hive：

```
hive
```

执行以下命令，应用生成的 JAR 包创建函数。

```
hive> create function nvl as "org.example.MyUDF" using jar "hdfs:///hive-udf-1.0-SNAPSHOT.jar";
```

说明：

1. nvl 是 UDF 函数的名称。
2. org.example.MyUDF 是项目中创建的类全名。
3. hdfs:///user/hive/warehouse/hiveudf-1.0-SNAPSHOT.jar 为上传 jar 包到 HDFS 的路径。

出现以下信息时，表示创建成功：

```
Added [/data/emr/hive/tmp/1b0f12a6-3406-4700-8227-37dec721297b_resources/hive-udf-1.0-SNAPSHOT.jar] to class
path
Added resources: [hdfs:///hive-udf-1.0-SNAPSHOT.jar]
OK
Time taken: 1.549 seconds
```

您也可以通过命令 SHOW FUNCTIONS LIKE '*nvl*'，验证函数是否创建成功。
执行以下命令，使用 UDF 函数。该函数与内置函数使用方式一样，直接使用函数名称即可访问：

```
hive> select nvl("tur", "def");
OK
tur
Time taken: 0.344 seconds, Fetched: 1 row(s)
hive> select nvl(null, "def");
```

```
OK
def
Time taken: 0.471 seconds, Fetched: 1 row(s)
```

查询管理开启说明

最近更新时间：2025-02-07 15:03:12

Hive查询管理开启步骤

Step1 检查是否有开启HIVE查询管理

1. 依次进入EMR 控制台，在集群列表中单击对应的集群 ID/名称计入集群详情页。
2. 在集群详情页中单击**集群服务**，然后选择 HIVE 服务，查看页面否有"查询管理"页签，如果有说明已经开启查询管理，如果没有需要执行后续步骤自动开启或提交工单开启 HIVE 查询管理。

Step2 检查 hive 采集插件是否部署

```
# 对于hive2.x版本，如下文件存在即是已经安装
ls /usr/local/service/hive/lib/*hive2-emr-hook*
# 对于hive3.x版本，如下文件存在即是已经安装
ls /usr/local/service/hive/lib/*hive3-emr-hook*
```

其中大部分版本默认会安装 Hive 采集插件，如果发现未安装，请通过 提交工单 进行 hive 插件安装。

Step3 修改 hive 采集插件相关配置

HIVE 服务页面，单击**配置管理**页签，进行如下配置更新及调整。

1. 修改 hive-site.xml 文件，修改如下3个配置项。
   - hive.exec.failure.hooks

   用逗号隔开添加：org.apache.hadoop.hive.ql.hooks.emr.ExecuteWithHookContextImpl

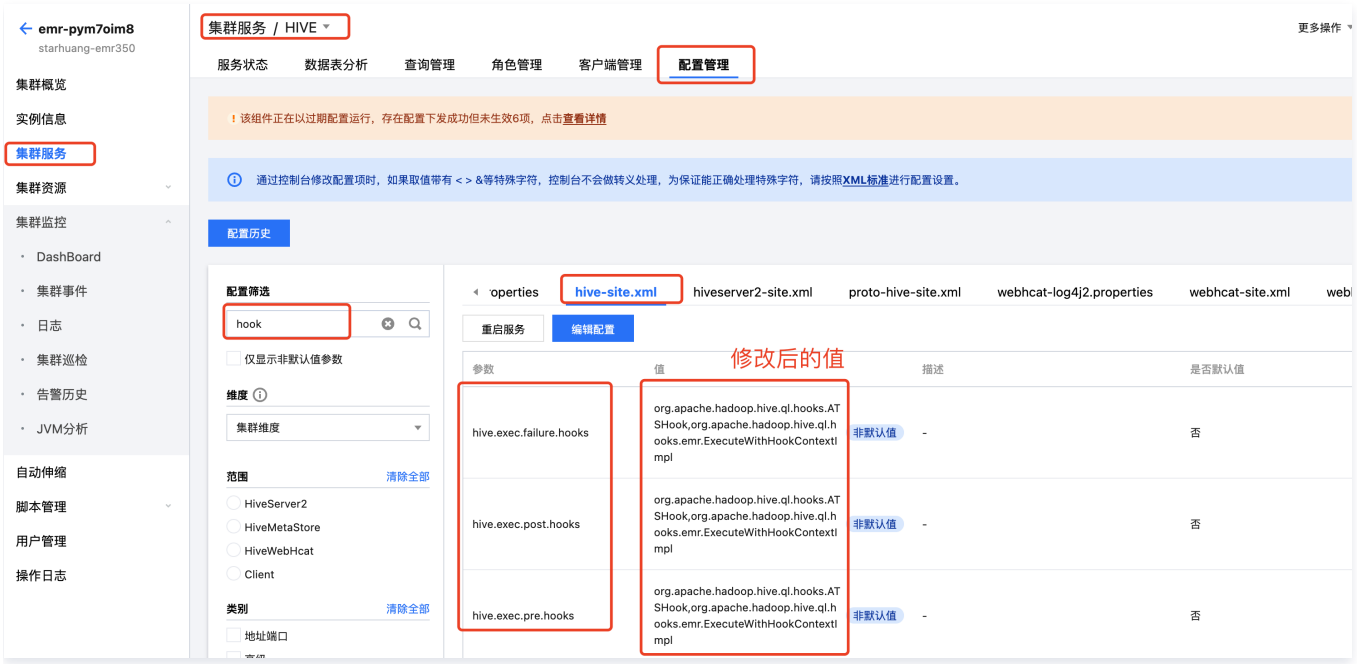
   1) 如果原来的值为：org.apache.hadoop.hive.ql.hooks.ATSHook

   添加后的值为：org.apache.hadoop.hive.ql.hooks.ATSHook,org.apache.hadoop.hive.ql.hooks.emr.ExecuteWithHookContextImpl

   （这个添加操作本质就是多个 hook 类之间用逗号隔开）。

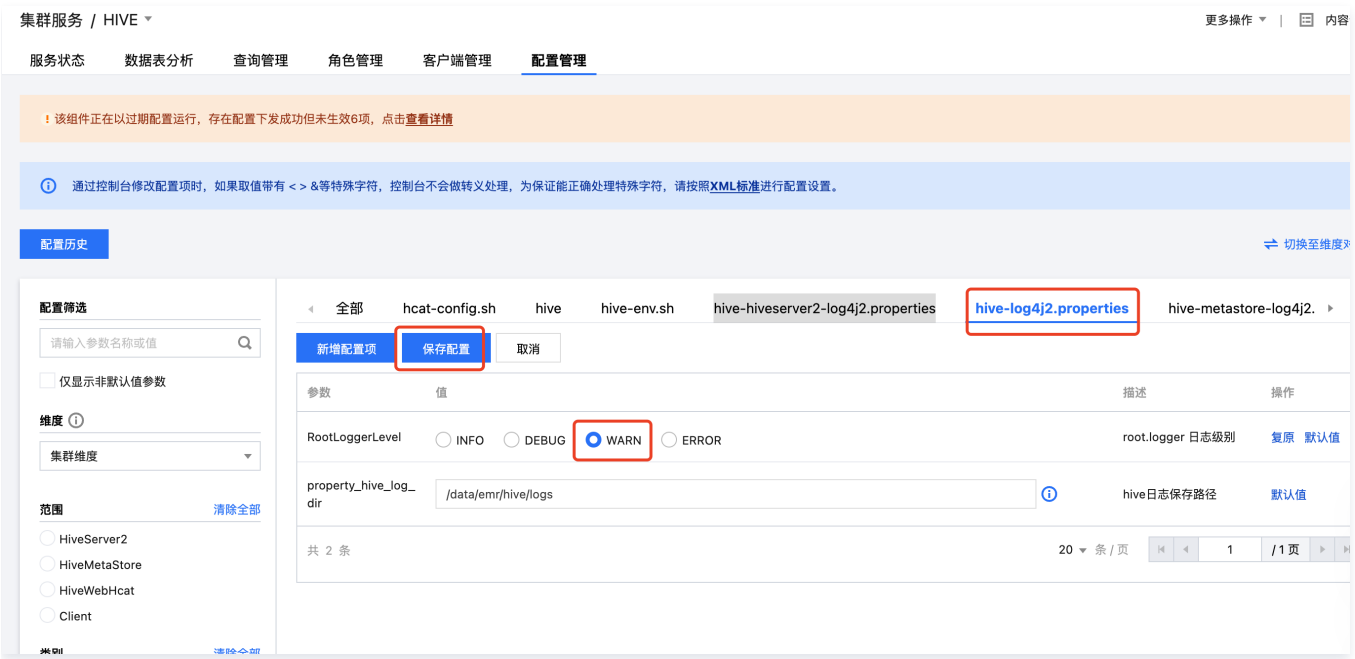
   2) 如果原来为空

   添加后的值为 org.apache.hadoop.hive.ql.hooks.emr.ExecuteWithHookContextImpl
   - hive.exec.post.hooks 同上。
   - hive.exec.pre.hooks 同上。



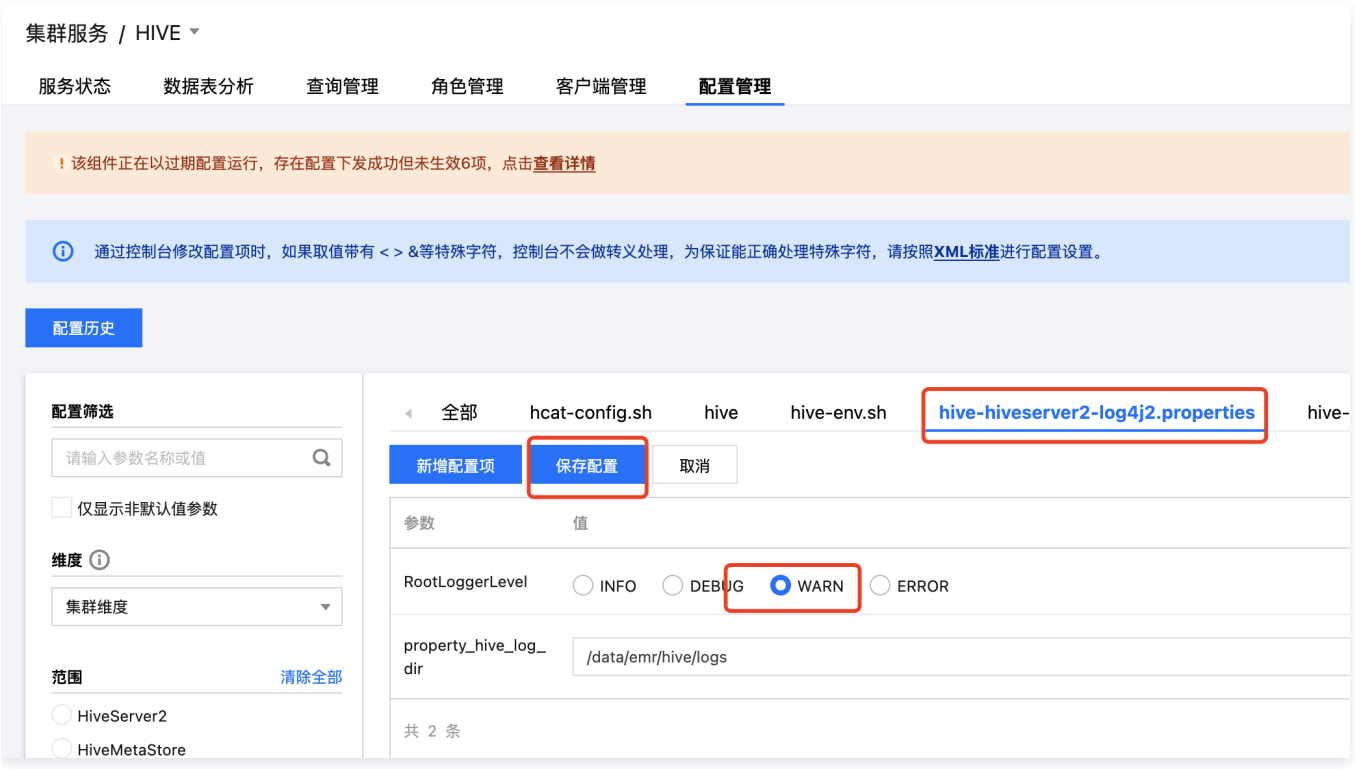
2. 重新下发 hive-log4j2.properties

选择 hive-log4j2.properties 文件，单击编辑配置，RootLoggerLevel 参数选中 WARN 后保存配置。



3. 重新下发 hive-hiveserver2-log4j2.properties

选择 hive-hiveserver2-log4j2.properties 文件，单击编辑配置，RootLoggerLevel 选中 WARN 后保存配置。



Step4 重启所有 HiveServer2

重启所有的 Hiveserver2 角色，但不用重启 HiveMetaStore 也不用重启 HiveWebHcat。

Step5 检查是否生效

任意运行一条简单的 HiveSQL，检查 HIVE 查询管理是否有运行的 SQL 信息。

实践教程
如何映射 HBase 表

最近更新时间：2023-12-15 10:33:25

使用 Hive 来映射 Hbase 表，可以使用 Hive 来读取 Hbase 上的数据，使用 Hive-SQL 语句在 Hbase 表上进行查询、插入等操作。

开发准备

- 确认已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群时需要在软件配置界面选择 Hive、Hbase 组件。
- Hive 等相关软件安装在路径 EMR 云服务器的 `/usr/local/service/` 路径下。

创建一个 Hbase 表

首先需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，进入 Hbase 文件夹并进入 Hbase shell：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/hbase
[hadoop@10hbase]$ bin/hbase shell
```

在 Hbase 中建立一个新表，如下所示：

```
hbase(main):001:0> create 'test', 'cf'
hbase(main):003:0> put 'test', 'row1', 'cf:a', 'value1'
hbase(main):004:0> put 'test', 'row1', 'cf:b', 'value2'
hbase(main):005:0> put 'test', 'row1', 'cf:c', 'value3'
```

更多在 Hbase 中的操作详见 Hbase 操作指南，或者查看 官方文档 。
创建完成后，可使用 `list` 和 `scan` 操作来查看新建的表。

```
hbase(main):001:0> list 'test'
TABLE
test
1 row(s) in 0.0030 seconds
=> ["test"]

hbase(main):002:0> scan 'test'
ROW  COLUMN+CELL
row1   column=cf:a, timestamp=1530276759697, value=value1
row2   column=cf:b, timestamp=1530276777806, value=value2
row3   column=cf:c, timestamp=1530276792839, value=value3
3 row(s) in 0.2110 seconds
```

映射 Hive 表

切换到 Hive 文件夹下，并且连接到 Hive 上：

```
[hadoop@172 hive]$ cd /usr/local/service/hive/
[hadoop@172 hive]$ bin/hive
```

接下来创建一个 Hive 外部表让它映射到第二步中创建的 Hbase 表上：

```
hive> CREATE EXTERNAL TABLE hive_test (
    rowkey string,
    a string,
    b string,
    c string
    ) STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH
    SERDEPROPERTIES("hbase.columns.mapping" = ":key,cf:a,cf:b,cf:c")
    TBLPROPERTIES("hbase.table.name" = "test");
OK
```

```
Time taken: 2.086 seconds
```

这样就建立了一个 Hive 表到 Hbase 表的映射。可使用以下指令来查看 Hive 表中的元素：

```
hive> select * from hive_test;
OK
row1 value1 value2 value3
Time taken: 0.305 seconds, Fetched: 1 row(s)
```

Hive 加载 json 数据实践

最近更新时间：2023-06-16 16:16:54

1. 连接 Hive

登录 EMR 集群的 Master 节点，切换到 hadoop 用户并且进入 hive 目录：

```
[root@10 ~]# su hadoop
[hadoop@10 root]$ cd /usr/local/service/hive
```

2. 准备数据

创建数据文件（JSON 格式）：

```
vim test.data
```

编译以下内容并保存：

```
{"name":"Mary","age":12,"course":[{"name":"math","location":"b208"},
{"name":"english","location":"b702"}],"grade":[99,98,95]}
{"name":"Bob","age":20,"course":[{"name":"music","location":"b108"},
{"name":"history","location":"b711"}],"grade":[91,92,93]}
```

将数据文件存储在 hdfs 上：

```
hadoop fs -put ./test.data /
```

3. 创建表格

连接 Hive：

```
[hadoop@10 hive]$ hive
```

根据映射关系创建表格：

```
hive> CREATE TABLE test (name string, age int, course array<map<string,string>>, grade array<int>) ROW FORMAT
SERDE 'org.apache.hive.hcatalog.data.JsonSerDe' STORED AS TEXTFILE;
```

4. 导入数据

```
hive>LOAD DATA INPATH '/test.data' into table test;
```

5. 检查数据是否导入成功

查询所有数据：

```
hive> select * from test;
OK
Mary    12  [{"name":"math","location":"b208"},{"name":"english","location":"b702"}]    [99,98,95]
Bob 20  [{"name":"music","location":"b108"},{"name":"history","location":"b711"}]   [91,92,93]
Time taken: 0.153 seconds, Fetched: 2 row(s)
```

查询每条记录的第一个得分：

```
hive> select grade[0] from test;
OK
```

```
99
91
Time taken: 0.374 seconds, Fetched: 2 row(s)
```

查询每条记录的第一个课程的名称和地点：

```
hive> select course[0]['name'], course[0]['location'] from test;
OK
math    b208
music   b108
Time taken: 0.162 seconds, Fetched: 2 row(s)
```

Hive 访问 Iceberg 数据

最近更新时间：2024-02-27 14:30:32

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考创建集群。
- 在创建 EMR 集群时在软件配置界面选择 Hive、Spark、Iceberg 组件。

通过 spark 创建 Iceberg 表

登录 Master 节点并切换 hadoop 用户，执行以下命令启动 SparkSQL：

```
spark-sql --master local[*] --conf
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions --conf
spark.sql.catalog.local=org.apache.iceberg.spark.SparkCatalog --conf spark.sql.catalog.local.type=hadoop --
conf spark.sql.catalog.local.warehouse=/usr/hive/warehouse --jars /usr/local/service/iceberg/iceberg-spark-
runtime-3.2_2.12-0.13.0.jar
```

说明：
Iceberg 相关的包放置在 /usr/local/service/iceberg/ 目录下，--jars 使用的依赖包版本在不同 EMR 版本中可能存在差异，请查看并使用正确的依赖包。

建表：

```
spark-sql> CREATE TABLE local.default.t1 (id int, name string) USING iceberg;
Time taken: 2.752 seconds
```

插入数据：

```
spark-sql> INSERT INTO local.default.t1 values(1, "tom");
Time taken: 2.71 seconds
```

查询数据：

```
spark-sql> SELECT * from local.default.t1;
1       tom
Time taken: 0.558 seconds, Fetched 1 row(s)
```

通过 Hive 查看 Iceberg 数据

登录 Master 节点并切换 hadoop 用户，执行以下命令连接 Hive：

```
hive
```

添加 Iceberg 依赖包：

```
hive> add jar /usr/local/service/iceberg/iceberg-hive-runtime-0.13.0.jar;
```

创建外部表：

```
hive> CREATE EXTERNAL TABLE t1
STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'
LOCATION '/usr/hive/warehouse/default/t1'
TBLPROPERTIES ('iceberg.catalog'='location_based_table');
```

查询 t1 表记录数：

```
hive> select count(*) from t1;
OK
1
Time taken: 26.255 seconds, Fetched: 1 row(s)
```

Hive 访问 Hudi 数据

最近更新时间：2024-02-21 17:39:01

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考创建集群。
- 在创建 EMR 集群时在软件配置界面选择 Hive、Spark、Hudi 组件。

通过 spark 读写 Hudi

登录 master 节点并切换 hadoop 用户，使用 SparkSQL 的 HoodieSparkSessionExtension 扩展进行数据读写：

```
spark-sql --master yarn \
--num-executors 2 \
--executor-memory 1g \
--executor-cores 2 \
--jars /usr/local/service/hudi/hudi-bundle/hudi-spark3.3-bundle_2.12-0.13.0.jar \
--conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer' \
--conf 'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension' \
--conf 'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog'
```

说明：
其中 --master 表示您的 master URL，--num-executors 表示 executor 数量，--executor-memory 表示 executor 的储存容量，以上参数也可以根据您的实际情况作出修改。--jars 使用的依赖包版本在不同 EMR 版本中可能存在差异，**请在 /usr/local/service/hudi/hudi-bundle 目录下查看并使用正确的依赖包。**

建表：

```
-- 创建COW非分区表


spark-sql> create table hudi_cow_nonpcf_tbl (
  uuid int,
  name string,
  price double
) using hudi
tblproperties (
  primaryKey = 'uuid'
);


-- 创建COW分区表


spark-sql> 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);
```

```
-- 创建MOR分区表


spark-sql> create table hudi_mor_tbl (
  id int,
  name string,
  price double,
  ts bigint,
  dt string
) using hudi
tblproperties (
  type = 'mor',
  primaryKey = 'id',
  preCombineField = 'ts'
)
partitioned by (dt);
```

写入数据：

```
-- insert into non-partitioned table
spark-sql> insert into hudi_cow_nonpcf_tbl select 1, 'a1', 20;


-- insert dynamic partition
spark-sql> insert into hudi_cow_pt_tbl partition (dt, hh) select 1 as id, 'a1' as name, 1000 as ts, '2021-12-
09' as dt, '10' as hh;


-- insert static partition
spark-sql> insert into hudi_cow_pt_tbl partition(dt = '2021-12-09', hh='11') select 2, 'a2', 1000;
spark-sql> insert into hudi_mor_tbl partition(dt = '2021-12-09') select 1, 'a1', 20, 1000;
```

使用 Hive 查询 Hudi 表

登录 Master 节点并切换 hadoop 用户，执行以下命令连接 hive：

```
hive
```

添加 hudi 依赖包：

```
hive> add jar /usr/local/service/hudi/hudi-bundle/hudi-hadoop-mr-bundle-0.13.0.jar;
```

查看表：

```
hive> show tables;
OK
hudi_cow_nonpcf_tbl
hudi_cow_pt_tbl
hudi_mor_tbl
hudi_mor_tbl_ro
hudi_mor_tbl_rt
Time taken: 0.023 seconds, Fetched: 5 row(s)
```

查询数据：

```
hive> select * from hudi_cow_nonpcf_tbl;
OK
20230905170525412       20230905170525412_0_0   1               8d32a1cc-11f9-437f-9a7b-8ba9532223d3-0_0-17-
15_20230905170525412.parquet        1       a1      20.0
```

```
Time taken: 1.447 seconds, Fetched: 1 row(s)

hive> set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
hive> select * from hudi_mor_tbl_ro;
OK
20230808174602565   20230808174602565_0_1   id:1    dt=2021-12-09   af40667d-1dca-4163-89ca-2c48250985b2-0_0-
34-1617_20230808174602565.parquet   1   a1  20.0    1000    2021-12-09
Time taken: 0.159 seconds, Fetched: 1 row(s)


hive> set hive.vectorized.execution.enabled=false;
hive> select name, count(*) from hudi_mor_tbl_rt group by name;
a1  1
Time taken: 17.618 seconds, Fetched: 1 row(s)
```

使用 Hive 在 COS/CHDFS 中创建库表

最近更新时间：2024-04-03 10:07:21

本文为您介绍如何使用 Hive 在 COS 及 CHDFS 上创建库表。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Hive 组件。
- 示例中存在需要访问腾讯云对象存储 COS 的内容，可参考 创建存储桶 在 COS 中创建一个存储桶（Bucket），并于 EMR 控制台 实例信息 页面开启对象存储授权。
- 示例中存在需要访问腾讯云对象存储 CHDFS 的内容，可参考 挂载 CHDFS 创建挂载点并挂载至 EMR 集群。

使用 Hive 在 COS 上创建库表

> **说明：**
> EMR 已默认集成 Hadoop-COS，您在 EMR 控制台开启对象存储授权后，会使用腾讯云 EMR 实例绑定的角色，获取访问 COS 的临时密钥进行访问，该方式相比固定密钥更为安全。
> 其他配置方式可参考对象存储-Hadoop 工具。

方式一：将整个数据库建立在 COS 上

登录 EMR 集群的 Master 节点，切换到 Hadoop 用户，执行以下命令即可进入 Hive 命令行：

```
hive
```

执行下面的命令，在您的 COS 桶下创建名为 hivewithcos 的数据库。

```
hive> create database hivewithcos location 'cosn://${bucketname}/${path}';
```

> **说明：**
> 其中 ${bucketname}为您创建的 COS 存储桶名称，${path} 为存储路径。

查看执行结果，可以看到我们在 COS 上创建的 hivewithcos 数据库：

```
hive> show databases;
OK
default
hivewithcos
Time taken: 0.094 seconds, Fetched: 2 row(s)
```

创建一张名为 record 的数据表（向表中 load 数据的方式和 HDFS 相同）：

```
hive> use hivewithcos;
hive> create table record(id int, name string) row format delimited fields terminated by ',' stored as
textfile;
```

查看表：

```
hive> show tables;
OK
record
Time taken: 0.063 seconds, Fetched: 1 row(s)
```

方式二：将指定表放在 COS 上

在 Hive 中创建一个数据库：

```
hive> create database test;
hive> use test;
```

执行如下语句在 COS 桶路径下创建名为 record 的表（向表中 load 数据的方式和 HDFS 相同）：

```
hive> create table record(id int, name string) row format delimited fields terminated by ',' stored as
textfile location 'cosn://$bucketname/$path';
```

查看表：

```
hive> show tables;
OK
record
Time taken: 0.063 seconds, Fetched: 1 row(s)
```

使用 Hive 在 CHDFS 创建库表

方式一：将整个数据库建立在 CHDFS上

登录 EMR 集群的 Master 节点，切换到 Hadoop 用户，执行以下命令即可进入 Hive 命令行：

```
hive
```

执行下面的命令，在您的 CHDFS 目录下创建名为 hivewithofs 的数据库：

```
hive> create database hivewithofs location 'ofs://${mountpoint}/${path}';
```

> **说明：**
> 其中 ${mountpoint} 为您创建的 CHDFS 挂载地址，${path} 为路径。

查看执行结果，可以看到我们在 CHDFS 上创建的 hivewithofs 数据库：

```
hive> show databases;
OK
default
hivewithofs
Time taken: 0.094 seconds, Fetched: 2 row(s)
```

创建一张名为 record 的数据表（向表中 load 数据的方式和 HDFS 相同）

```
hive> use hivewithofs;
hive> create table record(id int, name string) row format delimited fields terminated by ',' stored as
textfile;
```

查看表：

```
hive> show tables;
OK
record
Time taken: 0.063 seconds, Fetched: 1 row(s)
```

方式二：将指定表放在 CHDFS 上

在 Hive 创建一个数据库 test2：

```
hive> create database test2;
```

```
hive> use test2;
```

执行如下语句在 chdfs 下创建名为record的表：

```
hive> create table record(id int, name string) row format delimited fields terminated by ',' stored as
textfile location 'cosn://$mountpoint/$path';
```

查看表：

```
hive> show tables;
OK
record
Time taken: 0.063 seconds, Fetched: 1 row(s)
```

Trino(Presto) 开发指南
Presto 服务 UI

最近更新时间：2023-08-17 17:04:41

Presto 是一个开源的分布式 SQL 查询引擎，适用于交互式分析查询，数据量支持 GB 到 PB 字节。Presto 的设计和编写是为了解决大规模的甚至超大规模商业数据仓库的交互式分析和处理速度的问题。
Presto 支持在线数据查询，包括 Hive、Cassandra、关系数据库以及专有数据存储。一条 Presto 查询可以将这些多个数据源的数据进行合并，可以跨越整个组织进行分析。
EMR 代理了 Presto 原生 Web UI，可以直接在 EMR 控制台查看。登录 EMR 控制台，单击**集群实例 ID**，进入实例管理页面。单击左侧菜单栏**集群服务**，即可看到 WebUI 地址页面快捷入口，单击 Presto 的入口即可。登录用户名为 root，密码为创建集群时设置的密码。如下图：

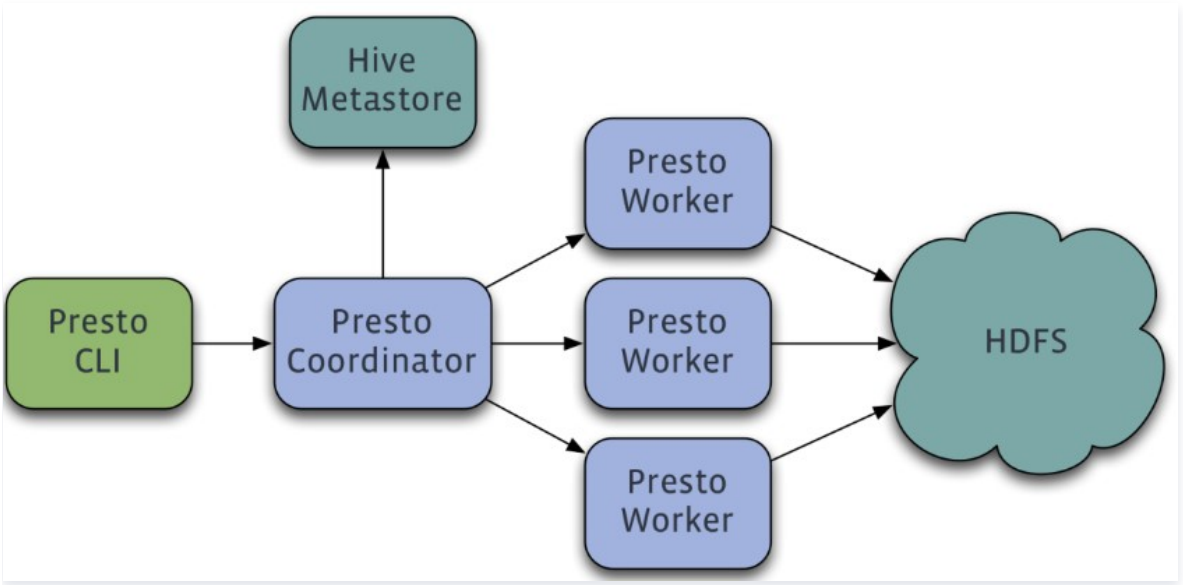
访问地址需要进行身份验证，用户名为 root，默认密码为创建集群时输入的密码，如需修改密码，可在该页面中单击**重置WebUI密码**进行修改。

> **说明：**
> 添加配置文件、使用集群脚本不能使用 /data/emr/prestosql/etc、/data/emr/presto/etc。

连接器

最近更新时间：2024-10-11 15:17:11

Presto 是由 Facebook 开发的一个分布式 SQL 查询引擎，被设计为用来专门进行高速、实时的数据分析，适用于交互式分析查询，数据量支持 GB 到 PB 字节。支持标准的 ANSI SQL，包括复杂查询、聚合（aggregation）、连接（join）和窗口函数（window functions）。采用 Java 实现。Presto 的数据源包括 Hive、HBase、关系数据库，甚至专有数据存储。其架构图如下所示：



Presto 是一个运行在多台服务器上的分布式系统，采用了主从（Master-Slave）架构，包括一个主节点 Coordinator 和多个从节点 Worker。客户端 Presto CLI 负责提交查询到 Coordinator 节点；Coordinator 节点负责解析 SQL 语句、生成查询执行计划、管理 Worker 节点等；Worker 节点负责实际执行查询任务。
EMR 中 Presto 组件预置了 Hive、Mysql 和 Kafka 等连接器，本节将以 Hive 连接器为例说明 Presto 读取 Hive 的表信息进行查询的使用，EMR 集群机器配置了 presto-client 的相关环境变量，可直接切换 Hadoop 用户并使用 Presto 客户端工具。

开发准备

- 确认您已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Presto 组件。
- Presto 等相关软件安装在路径 EMR 云服务器的 `/usr/local/service/` 路径下。

使用连接器操作 Hive

首先需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Presto 文件夹：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/presto
```

在 `etc/config.properties` 配置文件中查看 uri 的值：

```
[hadoop@172 presto]$ vim etc/config.properties
http-server.http.port=$port
discovery.uri=http://$host:$port
```

其中 $host 为您的 host 地址；$port 为您的端口号。然后切换到 presto-client 文件夹中，并且使用 Presto 连接 Hive：

```
[hadoop@172 presto]# cd /usr/local/service/presto/presto-client
[hadoop@172 presto-client]$ ./presto --server $host:$port --catalog hive --schema default
```

其中 --catalog 参数表示要操作的数据库类型，--schema 表示数据库名，这里进入的是默认的 default 数据库。更多的参数信息，可以通过命令 `presto -h` 来查看，或者查看 官方文档 。

执行成功后即可进入 Presto 的界面，并且直接进入指定的数据库。可以使用 Hive-SQL 来查看 Hive 数据库中的表：

```
presto:default> show tables;
     Table
---------------
hive_from_cos
test
(2 rows)

Query 20180702_140619_00006_c4qzg, FINISHED, 2 nodes
Splits: 2 total, 2 done (100.00%)
0:00 [3 rows, 86B] [17 rows/s, 508B/s]
```

其中表 hive_from_cos 是在 Hive 开发指南中创建的表。
更多 Presto 操作请查看 官方文档 。

分析 COS 上的数据

最近更新时间：2024-08-15 14:44:01

本节将基于腾讯云对象存储 COS 展示 Hive 连接器更多使用方法，数据来源于直接插入数据、COS 数据和 lzo 压缩数据。

开发准备

- 因为任务中需要访问腾讯云对象存储（COS），所以需要在 COS 中先 创建一个存储桶（Bucket）。
- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Presto 组件，并且在基础配置页面开启对象存储的授权。
- Presto 等相关软件安装在路径 EMR 云服务器的 `/usr/local/service/` 路径下。

数据准备

首先需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Hive 文件夹：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/hive
```

新建文件 cos.txt，并添加数据如下：

```
5,cos_patrick
6,cos_stone
```

使用 HDFS 指令把文件上传到 COS 中。其中 $bucketname 为您创建的存储桶的名字和路径。

```
[hadoop@172 hive]# hdfs dfs -put cos.txt cosn://$bucketname/
```

再新建文件 lzo.txt，并添加数据如下：

```
10,lzo_pop
11,lzo_tim
```

将其压缩为 .lzo 文件：

```
[hadoop@172 hive]$ lzop -v lzo.txt
compressing hive_test.data into lzo.txt.lzo
```

> **注意**
> 压缩 lzo 文件需要先安装 lzo 和 lzop，安装执行命令：`yum -y install lzo lzop`。

新建 Hive 表并使用 Presto 查询

这里使用了一个脚本文件来生成进行 Hive 数据库和表的创建。新建一个脚本文件 presto_on_cos_test.sql，并添加以下程序：

```
create database if not exists test;
use test;
create external table if not exists presto_on_cos (id int,name string) ROW FORMAT DELIMITED FIELDS TERMINATED
BY ',';
insert into presto_on_cos values (12,'hello'),(13,'world');
load data inpath "cosn://$bucketname/cos.txt" into table presto_on_cos;
load data local inpath "/$yourpath/lzo.txt.lzo" into table presto_on_cos;
```

其中 $bucketname 为您的 COS 存储桶名加路径，$yourpath 为您放置 lzo.txt.lzo 文件的路径。

脚本文件首先新建一个数据库“test”，在新建的数据库中新建一个表“presto_on_cos”。分三步进行了数据的插入操作，首先采用直接插入的方法。然后插入了 COS 中数据，最后插入 lzo 压缩包中的数据。
建议如示例一样，使用外部表进行 Hive 测试，以免删除重要数据。使用 hive-cli 执行这个脚本：

```
[hadoop@172 hive]$ hive -f "presto_on_cos_test.sql"
```

执行完成之后，就可以进入 Presto 查看表中的数据。使用上一节的方法进入 Presto，不过需要改动 schema 参数。

```
[hadoop@172 presto-client]$ ./presto --server $host:$port --catalog hive --schema test
```

对刚刚创建的 Hive 表进行查询：

```
presto:test> select * from presto_on_cos ;
 id |    name
----+-------------
  5 | cos_patrick
  6 | cos_stone
 10 | lzo_pop
 11 | lzo_tim
 12 | hello
 13 | world
(6 rows)

Query 20180702_150000_00011_c4qzg, FINISHED, 3 nodes
Splits: 4 total, 4 done (100.00%)
0:03 [6 rows, 127B] [1 rows/s, 37B/s]
```

更多 Presto 操作请查看 官方文档 。

Trino HA 介绍

最近更新时间：2024-07-17 09:46:31

Trino HA 通过 keepalived + 高可用虚拟 IP（HAVIP）来实现，整个过程自动化，无需人工干预。
keepalived 如何实现高可用的原理可以参考 用 HAVIP+Keepalived 搭建高可用主备集群 。创建的高可用虚拟 IP 可以通过 私有网络控制台 进行查看。

注意：

1. 仅在 EMR on CVM 的 Hadoop 集群类型中，Trino 应用场景适用于高可用集群。
2. 无需在 私有网络-高可用虚拟 IP 控制台去操作或解绑，集群自动切换所属主机，且随集群生命周期保持一致。
3.

Trino HA 集群创建完后，可以通过**集群服务>角色管理**中的右上**角查看信息**查看 HAVIP。

Trino 就可以通过虚拟 IP 进行查询了。

```
/usr/local/service/trino/client/trino-cli --server {{高可用虚拟ip}}:9000 --catalog hive --user hadoop
```

Trino 查询管理开启说明

最近更新时间：2024-09-25 14:31:11

Trino 查询管理功能分为采集端+展示端。此功能新购集群默认开启，存量集群需要开启白名单功能。

- 采集端分 ems-agent 和插件包要求如下：
  - ems-agent 保证为 3.0.4 以上版本
  - 插件包为 trino-audit 和 trino-audit-loader
- 相关版本及采集方式信息参见下表：

包类别	插件包	支持emr版本	部署位置	下载方式
audit	trino-audit-jar-with-dependencies.jar	emr280-trino435，emr340-trino372，emr270-trino389，emr350-trino389，emr360-trino389	/usr/local/service/trino/trino-audit	默认下载
audit	presto-audit-jar-with-dependencies.jar	emr250-presto332 emr330-presto332	/usr/local/service/presto/presto-audit	默认下载
audit	presto-audit-jar-with-dependencies.jar	emr230-presto0.228	/usr/local/service/presto/presto-audit	默认下载
loader	trino-audit-loader	emr270-trino389，emr340-trino389，emr350-trino389，emr360-trino389	/	新集群默认在镜象中，存量集群配置后下
loader	presto-audit-loader	emr250-presto332，emr330-presto332	/	新集群默认在镜象中，存量集群配置后下
loader	presto-audit-loader	emr230-presto0.228	/	新集群默认在镜象中，存量集群配置后下

开启方法

> **说明：**
> 配置前务必确认当前集为哪个 EMR 版本。

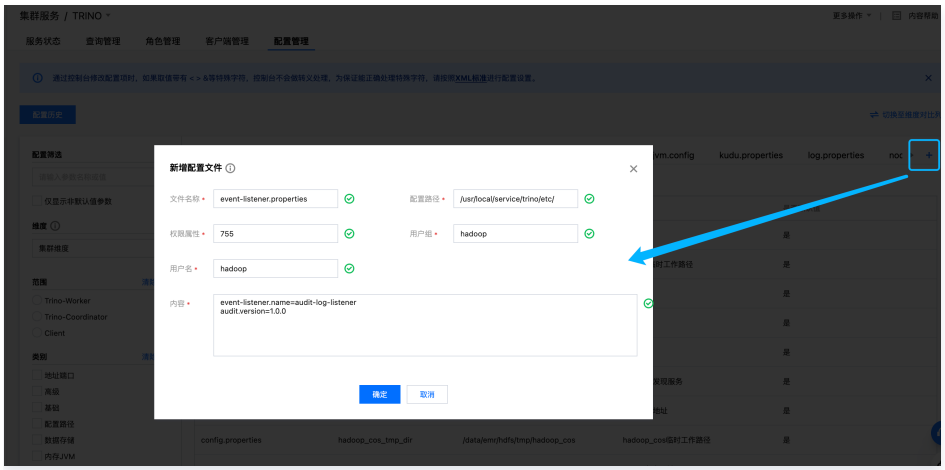
新集群（2024-01-30 00：00以后创建集群）

新集群默认开启查询管理，但需要手动添加配置文件，重启集群操作如下：

- 添加配置文件如下 event-listener.properties，

其中：presto 的路径为：/usr/local/service/presto/etc/
trino 的路径为：/usr/local/service/trino/etc/
用户名：hadoop
权限属性：755
内容：

```
event-listener.name=audit-log-listener
audit.version=1.0.0
```



- 重启 trino 集群的 Coordinators 服务

存量集群

开启方法：请通过 提交工单 申请开通。

Sqoop 开发指南
关系型数据库和 HDFS 的导入导出

最近更新时间：2023-12-15 16:51:22

Sqoop 是一款开源的工具，主要用于在 Hadoop 和传统数据库（MySQL、PostgreSQL 等）之间进行数据传递，可以将一个关系型数据库（例如 MySQL、Oracle、Postgres 等）中的数据导入到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导入到关系型数据库中。Sqoop 中一大亮点就是可以通过 Hadoop 的 MapReduce 把数据从关系型数据库中导入数据到 HDFS。
本文介绍了使用腾讯云 Sqoop 服务将数据在 MySQL 和 HDFS 之间导入/导出的使用方法。

开发准备

- 确认已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Sqoop 组件。
- Sqoop 等相关软件安装在路径 EMR 云服务器的 `/usr/local/service/` 路径下。

新建一个 MySQL 表

首先要连接已经创建好的 MySQL 数据库，进入 EMR 控制台，复制目标集群的实例 ID，即集群的名字。然后进入关系型数据库控制台，使用 Ctrl+F 进行搜索，找到集群对应的 MySQL 数据库，查看该数据库的内网地址 $mysqlIP。
登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Sqoop 文件夹：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/sqoop
```

连接 MySQL 数据库：

```
[hadoop@172 sqoop]$ mysql -h $mysqlIP -p
Enter password:
```

密码为您创建 EMR 集群的时候设置的密码。
MySQL 数据库连接后，进入 test 数据库并且新建一个表，用户也可以自己选择目标数据库：

```
mysql> use test;
Database changed

mysql> create table sqoop_test(id int not null primary key auto_increment, title varchar(64), time timestamp,
content varchar(255));
Query ok , 0 rows affected(0.00 sec)
```

该指令创建了一个 MySQL 表，它的主键为 ID，然后还有三列分别为 title、time 和 content。向该表中插入数据如下：

```
mysql> insert into sqoop_test values(null, 'first', now(), 'hdfs');
Query ok, 1 row affected(0.00 sec)

mysql> insert into sqoop_test values(null, 'second', now(), 'mr');
Query ok, 1 row affected (0.00 sec)

mysql> insert into sqoop_test values(null, 'third', now(), 'yarn');
Query ok, 1 row affected(0.00 sec)
```

使用如下指令可以查看表中的数据：

```
Mysql> select * from sqoop_test;
+----+--------+---------------------+---------+
| id | title  | time                | content |
+----+--------+---------------------+---------+
```

```
|  1 | first  | 2018-07-03 15:29:37 | hdfs    |
|  2 | second | 2018-07-03 15:30:57 | mr      |
|  3 | third  | 2018-07-03 15:31:07 | yarn    |
+----+--------+---------------------+---------+
3 rows in set (0.00 sec)
```

退出 MySQL 数据库：

```
Mysql> exit;
```

将 MySQL 的数据导入到 HDFS 中

使用 sqoop-import 把上一步中创建的 sqoop_test 表中数据导入到 HDFS 中：

```
[hadoop@172 sqoop]$ bin/sqoop-import --connect jdbc:mysql://$mysqlIP/test --username root -P --table
sqoop_test --target-dir /sqoop
```

其中 --connect 用于连接 MySQL 数据库，test 也可以换成您的数据库名字，-P 表示之后需要输入密码，--table 为您想要导出的数据库的名字，--target-dir 为导出到 HDFS 中的路径。** `/sqoop` 文件夹在执行命令之前并未创建，如果文件夹已经存在则会出错。**
回车后需要您输入密码，密码为您创建 EMR 时设置的密码。
执行成功后，可以在 HDFS 的相应路径下查看导入的数据：

```
[hadoop@172 sqoop]$ hadoop fs -cat /sqoop/*
1, first, 2018-07-03 15:29:37.0,hdfs
2, second, 2018-07-03 15:30:57.0,mr
3, third, 2018-07-03 15:31:07.0,yarn
```

将 HDFS 的数据导入到 MySQL 中

首先需要在 MySQL 新建一个表准备存放 HDFS 中的数据：

```
[hadoop@172 sqoop]$ mysql -h $mysqlIP -p
Enter password:
mysql> use test;
Database changed

mysql> create table sqoop_test_back(id int not null primary key auto_increment, title varchar(64), time
timestamp, content varchar(255));
Query ok , 0 rows affected(0.00 sec)
```

查看表是否创建成功后退出 MySQL：

```
mysql> show tables;
+-----------------+
| Tables_in_test  |
+-----------------+
| sqoop_test      |
| sqoop_test_back |
+-----------------+
2 rows in set (0.00 sec)

mysql> exit;
```

使用 sqoop-export 把上一步导入 HDFS 中的数据再一次导入到 MySQL 中：

```
[hadoop@172 sqoop]$ bin/sqoop-export --connect jdbc:mysql://$mysqlIP/test --username
root -P --table sqoop_test_back --export-dir /sqoop
```

参数和 sqoop-import 类似，只不过变成了 --export-dir，该参数为 HDFS 中的存放数据的路径。回车后也需要输入密码。
执行成功后，即可验证数据库 sqoop_test_back 中的数据：

```
[hadoop@172 sqoop]$ mysql -h $mysqlIP -p
Enter password:
mysql> use test;
Database changed

mysql> select * from sqoop_test_back;
+----+---------+---------------------+---------+
| id | title   | time                | content |
+----+---------+---------------------+---------+
|  1 | first   | 2018-07-03 15:29:37 | hdfs    |
|  2 | second  | 2018-07-03 15:30:57 | mr      |
|  3 | third   | 2018-07-03 15:31:07 | yarn    |
+----+---------+---------------------+---------+
3 rows in set (0.00 sec)
```

更多的 Sqoop 操作可以查看 官方文档 。

增量 DB 数据到 HDFS

最近更新时间：2024-10-11 15:17:11

Sqoop 是一款开源的工具，主要用于在 Hadoop 和传统数据库（MySQL、PostgreSQL 等）之间进行数据传递，可以将一个关系型数据库（例如 MySQL、Oracle、Postgres 等）中的数据导入到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导入到关系型数据库中。Sqoop 中一大亮点就是可以通过 Hadoop 的 MapReduce 把数据从关系型数据库中导入数据到 HDFS。
本文介绍了 Sqoop 的增量导入操作，即在数据库中的数据增加或更新后，把数据库的改动同步到导入 HDFS 的数据中。其中分为 append 模式和 lastmodified 模式，append 模式只能用在数据库的数据增加但不更新的场景，lastmodified 模式用在数据增加并且更新的场景。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Sqoop 组件。
- Sqoop 等相关软件安装在路径 EMR 云服务器的 `/usr/local/service/` 路径下。

使用 append 模式

本节将继续使用上一节的用例。
进入 EMR 控制台，复制目标集群的实例 ID，即集群的名字。再进入关系型数据库控制台，使用 Ctrl+F 进行搜索，找到集群对应的 MySQL 数据库，查看该数据库的内网地址 $mysqlIP。
登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Sqoop 文件夹：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/sqoop
```

连接 MySQL 数据库：

```
[hadoop@172 sqoop]$ mysql -h $mysqlIP -p
Enter password:
```

密码为您创建 EMR 集群的时候设置的密码。
连接了 MySQL 数据库之后，在表 sqoop_test 中新增一条数据，如下：

```
mysql> use test;
Database changed

mysql> insert into sqoop_test values(null, 'forth', now(), 'hbase');
Query ok, 1 row affected(0.00 sec)
```

查看表中的数据：

```
Mysql> select * from sqoop_test;
+----+--------+---------------------+---------+
| id | title  | time                | content |
+----+--------+---------------------+---------+
|  1 | first  | 2018-07-03 15:29:37 | hdfs    |
|  2 | second | 2018-07-03 15:30:57 | mr      |
|  3 | third  | 2018-07-03 15:31:07 | yarn    |
|  4 | forth  | 2018-07-03 15:39:38  | hbase    |
+----+--------+---------------------+---------+
4 rows in set (0.00 sec)
```

使用 append 模式将新增的数据同步到上一节中储存数据的 HDFS 路径中：

```
[hadoop@172 sqoop]$ bin/sqoop-import --connect jdbc:mysql://$mysqlIP/test --username
root -P --table sqoop_test --check-column id  --incremental append --last-value 3 --target-dir
```

```
/sqoop
```

其中 $mysqlIP 为您的 MySQL 数据库的内网地址。
执行命令会需要您输入数据库的密码，默认为您创建 EMR 集群时设置的密码。比普通的 sqoop-import 命令多出一些参数，其中 --check-column 为导入时参照的数据，--incremental 为导入的模式，在此例中为 append，--last-value 为参考数据的参考值，比该值更新的数据都会导入到 HDFS 中。
执行成功后，可以查看 HDFS 相应目录下更新后的数据：

```
[hadoop@172 sqoop]$ hadoop fs -cat /sqoop/*
1, first, 2018-07-03 15:29:37.0,hdfs
2, second, 2018-07-03 15:30:57.0,mr
3, third, 2018-07-03 15:31:07.0,yarn
4,forth,2018-07-03 15:39:38.0,hbase
```

使用 Sqoop job

使用 append 同步 HDFS 中的数据每次需要手动输入 --last-value，也可以使用 sqoop job 的方式，Sqoop 会自动保存上次导入成功的 last-value 值。如果要使用 sqoop job，需要启动 sqoop-metastore 进程，操作步骤如下：
首先在 conf/sqoop-site.xml 中启动 sqoop-metastore 进程：

```
<property>
  <name>sqoop.metastore.client.enable.autoconnect</name>
  <value>true</value>
</property>
```

然后在 bin 目录下启动 sqoop-metastore 服务：

```
./sqoop-metastore &
```

使用如下指令创建 Sqoop job：

> **说明**
> 此命令适用于 Sqoop 1.4.6 版本。

```
[hadoop@172 sqoop]$ bin/sqoop job --create job1 -- import --connect
jdbc:mysql://$mysqlIP/test --username root -P --table sqoop_test --check-column id
--incremental append --last-value 4 --target-dir /sqoop
```

其中 $mysqlIP 为您的 MySQL 的内网地址。使用该命令就成功创建了一个 Sqoop job，每一次执行，会自动从上次更新的 last-value 值自动更新。
为 MySQL 中的 sqoop_test 表格新增一条记录：

```
mysql> insert into sqoop_test values(null, 'fifth', now(), 'hive');
Query ok, 1 row affected(0.00 sec)

Mysql> select * from sqoop_test;
+----+--------+---------------------+---------+
| id | title  | time                | content |
+----+--------+---------------------+---------+
|  1 | first  | 2018-07-03 15:29:37 | hdfs    |
|  2 | second | 2018-07-03 15:30:57 | mr      |
|  3 | third  | 2018-07-03 15:31:07 | yarn    |
|  4 | forth  | 2018-07-03 15:39:38  | hbase    |
|  5 | fifth  | 2018-07-03 16:02:29  | hive    |
+----+--------+---------------------+---------+
5 rows in set (0.00 sec)
```

然后执行 Sqoop job：

```
[hadoop@172 sqoop]$ bin/sqoop job --exec job1
```

执行该命令会让您输入 MySQL 的密码。执行成功后，可以查看 HDFS 相应目录下更新后的数据：

```
[hadoop@172 sqoop]$ hadoop fs -cat /sqoop/*
1, first, 2018-07-03 15:29:37.0,hdfs
2, second, 2018-07-03 15:30:57.0,mr
3, third, 2018-07-03 15:31:07.0,yarn
4,forth,2018-07-03 15:39:38.0,hbase
5,fifth,2018-07-03 16:02:29.0,hive
```

使用 lastmodified 模式

直接创建一个 sqoop-import 的 lastmodified 模式的 Sqoop job，首先查询 sqoop_test 中最后更新的时间：

```
mysql> select max(time) from sqoop_test;
```

创建一个 Sqoop job：

```
[hadoop@172 sqoop]$ bin/sqoop job --create job2 -- import --connect jdbc:mysql://$mysqlIP/test --username
root -P --table sqoop_test --check-column time --incremental lastmodified --merge-key id --last-value '2018-
07-03 16:02:29' --target-dir /sqoop
```

参数说明：

- $mysqlIP 为您的 MySQL 的内网地址。
- --check-column 必须使用 timestamp 类型的列。
- --incremental 模式选择 lastmodified。
- --merge-key 选择 ID。
- --last-value 为我们查询到的表中的最后更新时间。在此时间后做出的更新都会被同步到 HDFS 中，而 Sqoop job 每次会自动保存和更新该值。

对 MySQL 中的 sqoop_test 表添加数据并做出更改：

```
mysql> insert into sqoop_test values(null, 'sixth', now(), 'sqoop');
Query ok, 1 row affected(0.00 sec)

mysql> update sqoop_test set time=now(), content='spark' where id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 changed: 1 warnings: 0

Mysql> select * from sqoop_test;
+----+--------+---------------------+---------+
| id | title  | time                | content |
+----+--------+---------------------+---------+
|  1 | first  | 2018-07-03 16:07:46 | spark   |
|  2 | second | 2018-07-03 15:30:57 | mr      |
|  3 | third  | 2018-07-03 15:31:07 | yarn    |
|  4 | forth  | 2018-07-03 15:39:38 | hbase   |
|  5 | fifth  | 2018-07-03 16:02:29 | hive    |
|  6 | fifth  | 2018-07-03 16:09:58 | sqoop   |
+----+--------+---------------------+---------+
6 rows in set (0.00 sec)
```

执行 Sqoop job：

```
[hadoop@172 sqoop]$ bin/sqoop job --exec job2
```

执行该命令会让您输入 MySQL 的密码。执行成功后，可以查看 HDFS 相应目录下更新后的数据：

```
[hadoop@172 sqoop]$ hdfs dfs -cat /sqoop/*
1,first,2018-07-03 16:07:46.0,spark
2,second,2018-07-03 15:30:57.0,mr
3,third,2018-07-03 15:31:07.0,yarn
4,forth,2018-07-03 15:39:38.0,hbase
5,fifth,2018-07-03 16:02:29.0,hive
6,sixth,2018-07-03 16:09:58.0,sqoop
```

更多的 Sqoop 操作可以查看 官方文档 。

Hive 存储格式和关系型数据库之间进行导入导出

最近更新时间：2023-12-15 10:33:25

本文介绍了使用腾讯云 Sqoop 服务将数据在 MySQL 和 Hive 之间相互导入导出的方法。

开发准备

- 确认已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Sqoop、Hive 组件。
- Sqoop 等相关软件安装在路径 EMR 云服务器的 `/usr/local/service/` 路径下。

将关系型数据库导入到 Hive 中

本节将继续使用上一节的用例。
进入 弹性 MapReduce 控制台，复制目标集群的实例 ID，即集群的名字。再进入关系型数据库控制台，使用 Ctrl+F 进行搜索，找到集群对应的 MySQL 数据库，查看该数据库的内网地址 $mysqlIP。
登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Hive 文件夹：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/hive
```

新建一个 Hive 数据库：

```
[hadoop@172 hive]$ hive
hive> create database hive_from_sqoop;
OK
Time taken: 0.167 seconds
```

使用 sqoop-import 命令把上一节中创建的 MySQL 数据库导入到 Hive 中：

```
[hadoop@172 hive]# cd /usr/local/service/sqoop
[hadoop@172 sqoop]$ bin/sqoop-import --connect  jdbc:mysql://$mysqlIP/test --username
root -P --table sqoop_test_back --hive-database hive_from_sqoop --hive-import --hive-table hive_from_sqoop
```

- $mysqlIP：腾讯云关系型数据库（CDB）的内网地址。
- test：MySQL 数据库名称。
- --table：要导出的 MySQL 表名。
- --hive-database：Hive 数据库名。
- --hive-table：导入的 Hive 表名。

执行指令需要输入您的 MySQL 密码，默认为您创建 EMR 集群时设置的密码。执行成功后，可以在 Hive 中查看导入的数据库：

```
hive> select * from hive_from_sqoop;
OK
1    first    2018-07-03 16:07:46.0    spark
2    second   2018-07-03 15:30:57.0    mr
3    third    2018-07-03 15:31:07.0    yarn
4    forth    2018-07-03 15:39:38.0    hbase
5    fifth    2018-07-03 16:02:29.0    hive
6    sixth    2018-07-03 16:09:58.0    sqoop
Time taken: 1.245 seconds, Fetched: 6 row(s)
```

将 Hive 导入到关系型数据库中

Sqoop 支持将 Hive 表中的数据导入到关系型数据库中。先在 Hive 中创建新表并导入数据。
登录 EMR 集群中的任意机器，最好是登录到 Master 节点。在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Hive 文件夹：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/hive
```

新建一个 bash 脚本文件 gen_data.sh，在其中添加以下代码：

```
#!/bin/bash
MAXROW=1000000 #指定生成数据行数
for((i = 0; i < $MAXROW; i++))
do
     echo $RANDOM, \"$RANDOM\"
done
```

并按如下方式执行：

```
[hadoop@172 hive]$ ./gen_data.sh > hive_test.data
```

这个脚本文件会生成1,000,000个随机数对，并且保存到文件 hive_test.data 中。
使用如下指令把生成的测试数据先上传到 HDFS 中：

```
[hadoop@172 hive]$ hdfs dfs -put ./hive_test.data /$hdfspath
```

其中 $hdfspath 为 HDFS 上的您存放文件的路径。
连接 Hive 并创建测试表：

```
[hadoop@172 hive]$ bin/hive
hive> create database hive_to_sqoop;          #创建数据库 hive_to_sqoop
OK
Time taken: 0.176 seconds
hive> use hive_to_sqoop;                              #切换数据库
OK
Time taken: 0.176 seconds
hive> create table hive_test (a int, b string)
hive> ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
                          #创建数据表 hive_test, 并指定列分割符为','
OK
Time taken: 0.204 seconds
hive> load data inpath "/$hdfspath/hive_test.data" into table hive_test;    #导入数据
```

$hdfspath 为 HDFS 上的您存放文件的路径。
成功后可使用 `quit` 命令退出 Hive 数据仓库。连接关系型数据库并创建对应的表格：

```
[hadoop@172 hive]$ mysql -h $mysqlIP -p
Enter password:
```

其中 $mysqlIP 为该数据库的内网地址，密码为您创建集群时设置的密码。
在 MySQL 中创建一个名为 test 的表格，**MySQL 中的表字段名字和 Hive 中的表字段名字必须完全一致**：

```
mysql> create table table_from_hive (a int,b varchar(255));
```

成功创建表格后即可退出 MySQL。
使用 Sqoop 把 Hive 数据仓库中的数据导入到关系型数据库中有两种方法，可以直接使用 HDFS 存储的 Hive 数据，也可以使用 Hcatalog 来进行数据的导入。

使用 HDFS 中的 Hive 数据

切换进入 Sqoop 文件夹，然后使用以下指令把 Hive 数据库中的数据导出到关系型数据库中：

```
[hadoop@172 hive]$ cd  ../sqoop/bin
```

```
[hadoop@172 bin]$ ./sqoop-export --connect jdbc:mysql://$mysqlIP/test --username root -P
--table table_from_hive --export-dir /usr/hive/warehouse/hive_to_sqoop.db/hive_test
```

其中 $mysqlIP 为您的关系型数据库的内网 IP 地址，test 为关系型数据库中的数据库名，--table 后跟的参数为您的关系型数据库的表名，--export-dir 后跟的参数为 Hive 表中的数据在 HDFS 中存储的位置。

使用 Hcatalog 进行导入

切换进入 Sqoop 文件夹，然后使用以下指令把 Hive 数据库中的数据导出到关系型数据库中：

```
[hadoop@172 hive]$ cd  ../sqoop/bin
[hadoop@172 bin]$ ./sqoop-export --connect jdbc:mysql://$mysqlIP/test --username root -P
--table table_from_hive --hcatalog-database hive_to_sqoop --hcatalog-table hive_test
```

其中 $mysqlIP 为您的关系型数据库的内网 IP 地址，test 为关系型数据库中的数据库名，--table 后跟的参数为您的关系型数据库的表名，--hcatalog-database 后面跟的参数是要导出的 Hive 表所在的数据库的名称，--hcatalog-table 后面跟的参数是要 Hive 中要导出的表的名称。
操作完成后可以进入关系型数据库查看是否导入成功：

```
[hadoop@172 hive]$ mysql -h $mysqlIP -p                  #连接 MySQL
Enter password:
mysql> use test;
Database changed
mysql> select count(*) from table_from_hive;      #现在表中有1000000条数据
+----------+
| count(*) |
+----------+
| 1000000 |
+----------+
1 row in set (0.03 sec)
mysql> select * from table_from_hive limit 10;     #查看表中前10条记录
+-------+----------+
| a     | b        |
+-------+----------+
| 28523 |  "3394"  |
| 31065 |  "24583" |
|   399 |  "23629" |
| 18779 |  "8377"  |
| 25376 |  "30798" |
| 20234 |  "22048" |
| 30744 |  "32753" |
| 21423 |  "6117"  |
| 26867 |  "16787" |
| 18526 |  "5856"  |
+-------+----------+
10 rows in set (0.00 sec)
```

更多关于 sqoop-export 命令的参数可以通过如下命令查看：

```
[hadoop@172 bin]$ ./sqoop-export --help
```

将 orc 格式的 Hive 表格导入到关系型数据库中

orc 是按列存储的一种文件存储格式，使用该格式能够极大的提升 Hive 的性能。本节介绍了如何创建一个 orc 格式的表并载入数据，然后使用腾讯云 Sqoop 服务把 Hive 中以 orc 格式进行存储的数据导出到关系型数据库。

> **注意**
> 将 orc 存储格式的 Hive 表格导入到关系型数据库中不能直接使用 HDFS 中存储的数据，只能使用 Hcatalog 进行操作。

本节将继续使用上一节的用例。
登录 EMR 集群的 Master 节点后，在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Hive 文件夹：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]# cd /usr/local/service/hive
```

在上一节中创建的 hive_from_sqoop 数据库中创建一个新表格：

```
[hadoop@172 hive]$ hive
hive> use hive_to_sqoop;
OK
Time taken: 0.013 seconds
hive> create table if not exists orc_test(a int,b string) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
stored as orc;
```

可以通过如下指令来查看表格中数据的存储格式：

```
hive> show create table orc_test;
OK
CREATE TABLE `orc_test`(
  `a` int,
  `b` string)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.orc.OrcSerde'
WITH SERDEPROPERTIES (
  'field.delim'=',',
  'serialization.format'=',')
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'
LOCATION
  'hdfs://HDFS2789/usr/hive/warehouse/hive_to_sqoop.db/orc_test'
TBLPROPERTIES (
  'COLUMN_STATS_ACCURATE'='{\"BASIC_STATS\":\"true\"}',
  'numFiles'='0',
  'numRows'='0',
  'rawDataSize'='0',
  'totalSize'='0',
  'transient_lastDdlTime'='1533563293')
Time taken: 0.041 seconds, Fetched: 21 row(s)
```

由返回的数据可以看出该表格中的数据存储格式为 orc。
有多种方式可以向 orc 格式的 Hive 表格导入数据，下面主要介绍通过创建临时的存储格式为 text 的 Hive 表格来向 orc 存储格式的表格导入数据，这里我们使用上一节中创建的 hive_test 表格作为临时表格，使用以下指令来导入数据：

```
hive> insert into table orc_test select * from hive_test;
```

导入成功后可通过 `select` 指令查看表格中的数据。
然后使用 Sqoop 把 orc 格式的 Hive 表格导出到 MySQL中。连接关系型数据库并创建对应的表格，连接关系型数据库的具体方式见上文：

```
[hadoop@172 hive]$ mysql -h $mysqlIP -p
Enter password:
```

其中 $mysqlIP 为该数据库的内网地址，密码为您创建集群时设置的密码。
在 MySQL 中创建一个名为 test 的表格，**MySQL 中的表字段名字和 Hive 中的表字段名字必须完全一致**：

```
mysql> create table table_from_orc (a int,b varchar(255));
```

成功创建表格后即可退出 MySQL。
切换进入 Sqoop 文件夹，然后使用以下指令把 Hive 数据库中以 orc 格式存储的数据导出到关系型数据库中：

```
[hadoop@172 hive]$ cd  ../sqoop/bin
[hadoop@172 bin]$ ./sqoop-export --connect jdbc:mysql://$mysqlIP/test --username root -P
--table table_from_orc --hcatalog-database hive_to_sqoop --hcatalog-table orc_test
```

其中 $mysqlIP 为您的关系型数据库的内网 IP 地址，test 为关系型数据库中的数据库名，--table 后跟的参数为您的关系型数据库的表名，--hcatalog-database 后面跟的参数是要导出的 Hive 表所在的数据库的名称，--hcatalog-table 后面跟的参数是要 Hive 中要导出的表的名称。
导入成功后可以在 MySQL 中查看相应表中的数据：

```
mysql> select count(*) from table_from_orc;
+----------+
| count(*) |
+----------+
| 1000000 |
+----------+
1 row in set (0.24 sec)
mysql> select * from table_from_orc limit 10;
+-------+----------+
| a     | b        |
+-------+----------+
| 28523 |  "3394"  |
| 31065 |  "24583" |
|   399 |  "23629" |
| 18779 |  "8377"  |
| 25376 |  "30798" |
| 20234 |  "22048" |
| 30744 |  "32753" |
| 21423 |  "6117"  |
| 26867 |  "16787" |
| 18526 |  "5856"  |
+-------+----------+
10 rows in set (0.00 sec)
```

更多的 Sqoop 操作可以查看 官方文档 。

Hue 开发指南
Hue 简介

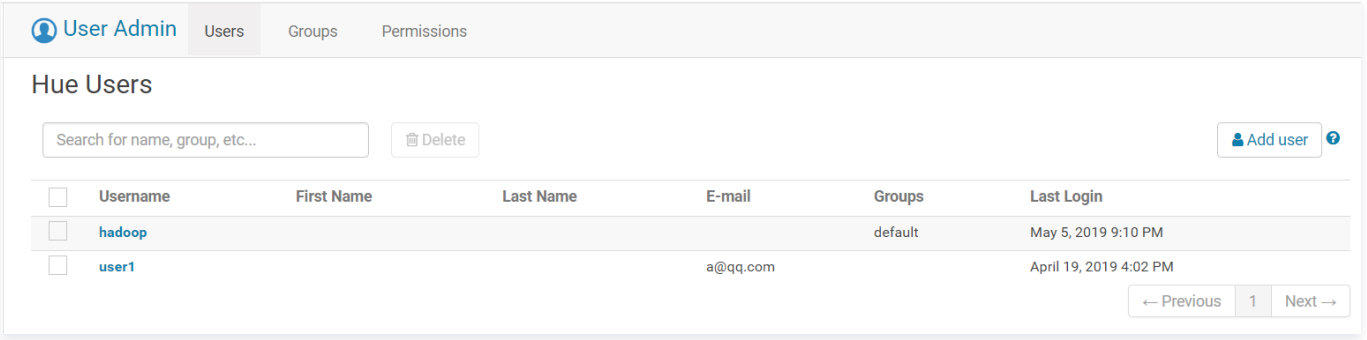
最近更新时间：2024-08-15 14:44:01

Hue 是一个开源的 Apache Hadoop UI 系统，由 Cloudera Desktop 演化而来，最后 Cloudera 公司将其贡献给 Apache 基金会的 Hadoop 社区，它是基于 Python Web 框架 Django 实现的。通过使用 Hue 我们可以在浏览器端的 Web 控制台上与 Hadoop 集群进行交互来分析处理数据，例如操作 HDFS 上的数据、运行 MapReduce Job、执行 Hive 的 SQL 语句和浏览 HBase 数据库等。

访问 Hue WebUI

使用 Hue 组件管理工作流时，请先登录 Hue 控制台页面，具体步骤如下：

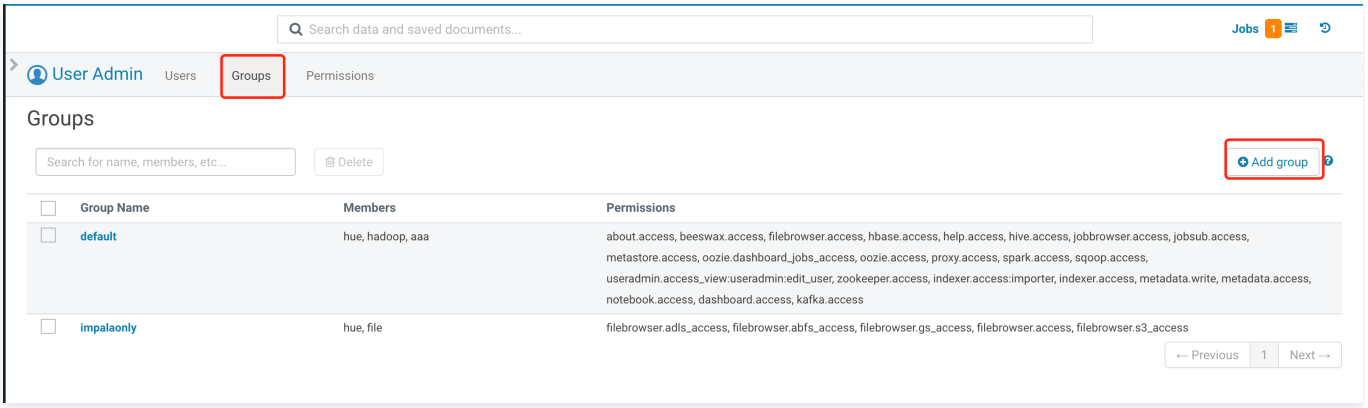
1. 登录 EMR 控制台，单击对应集群 **ID/名称**，进入集群详情页面，然后单击**集群服务**。
2. 在列表页找到 Hue 组件，单击 **WebUI 访问地址**进入 Hue 页面。
3. 首次登录 Hue 控制台页面，请使用 hadoop 账号，密码为创建集群时提供的密码。



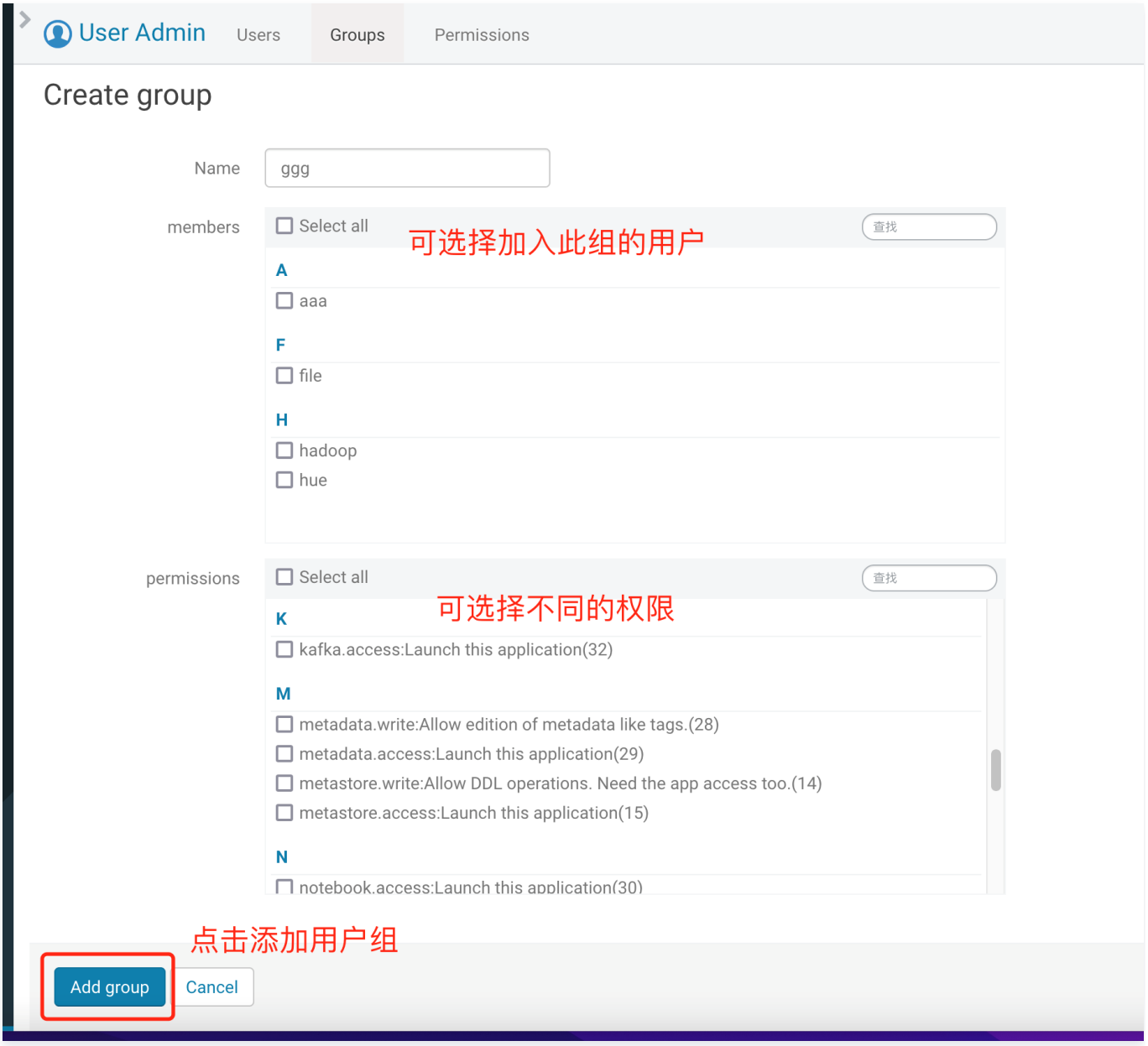
> **注意**
> EMR-V2.5.0及以前版本、EMR-V3.1.0及以前版本未集成 OpenLDAP ，需要在首次以 root 账号登录 Hue 控制台，参考 社区官方文档 于 WebUI 新建账号。EMR 产品的组件启动账号为 hadoop，历史版本建议首次登录 Hue 控制台后，新建 hadoop 账号，后续可以通过 hadoop 账号来提交作业。

用户权限管理

1. 添加用户。
   1.1 登录 EMR 控制台，使用 用户管理 功能添加新用户。
   1.2 如果您的集群部署了 Ranger，添加新用户后，需要手动触发 ranger-ugsync-site.xml 的配置下发，重启 EnableUnixAuth 服务进行用户同步，具体操作步骤可参考 用户管理 。然后进入 Ranger WebUI 设置新用户访问权限。
   1.3 在列表页找到 Hue 组件，单击 WebUI 访问地址进入 Hue 页面，完成新用户登录及使用。
2. 权限控制。
   Hue 通过将不同的权限添加到组，用户通过加入不同的组获得对应权限。
   2.1 单击用户管理页面上方的 **Groups**，然后单击右侧的 **Add group**。

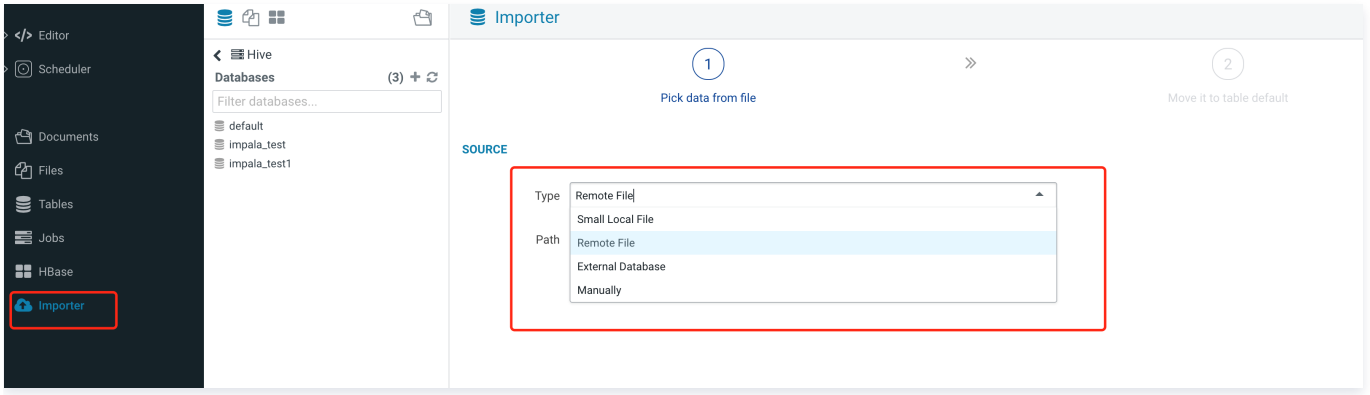


2.2 填写用户组信息，可勾选目标用户加入此组，并勾选此用户组的权限，单击下方的 **Add Group**。



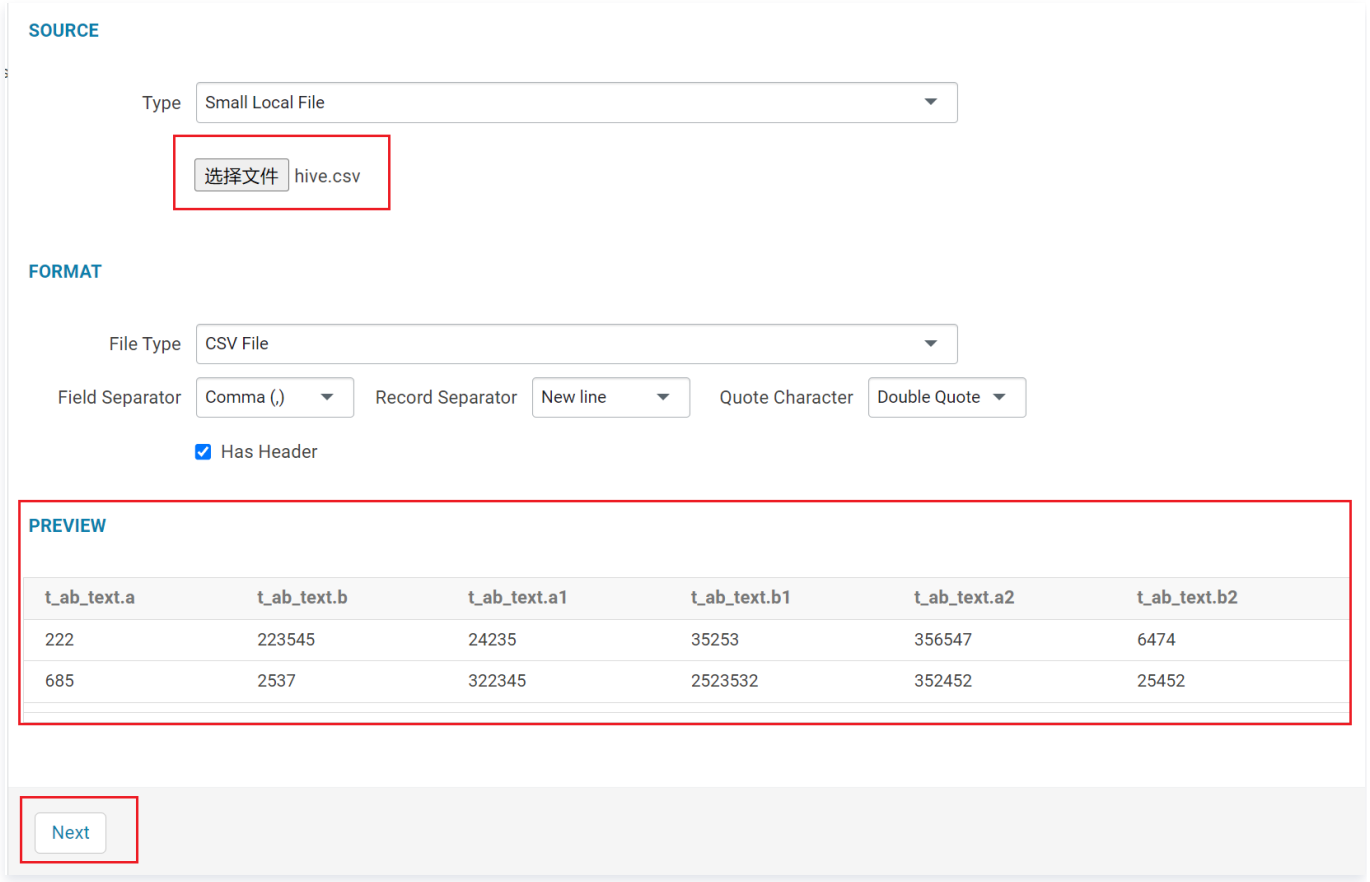
数据导入

Hue 支持4种导入方式：本地文件、HDFS 上的文件、外部数据库以及人工导入。

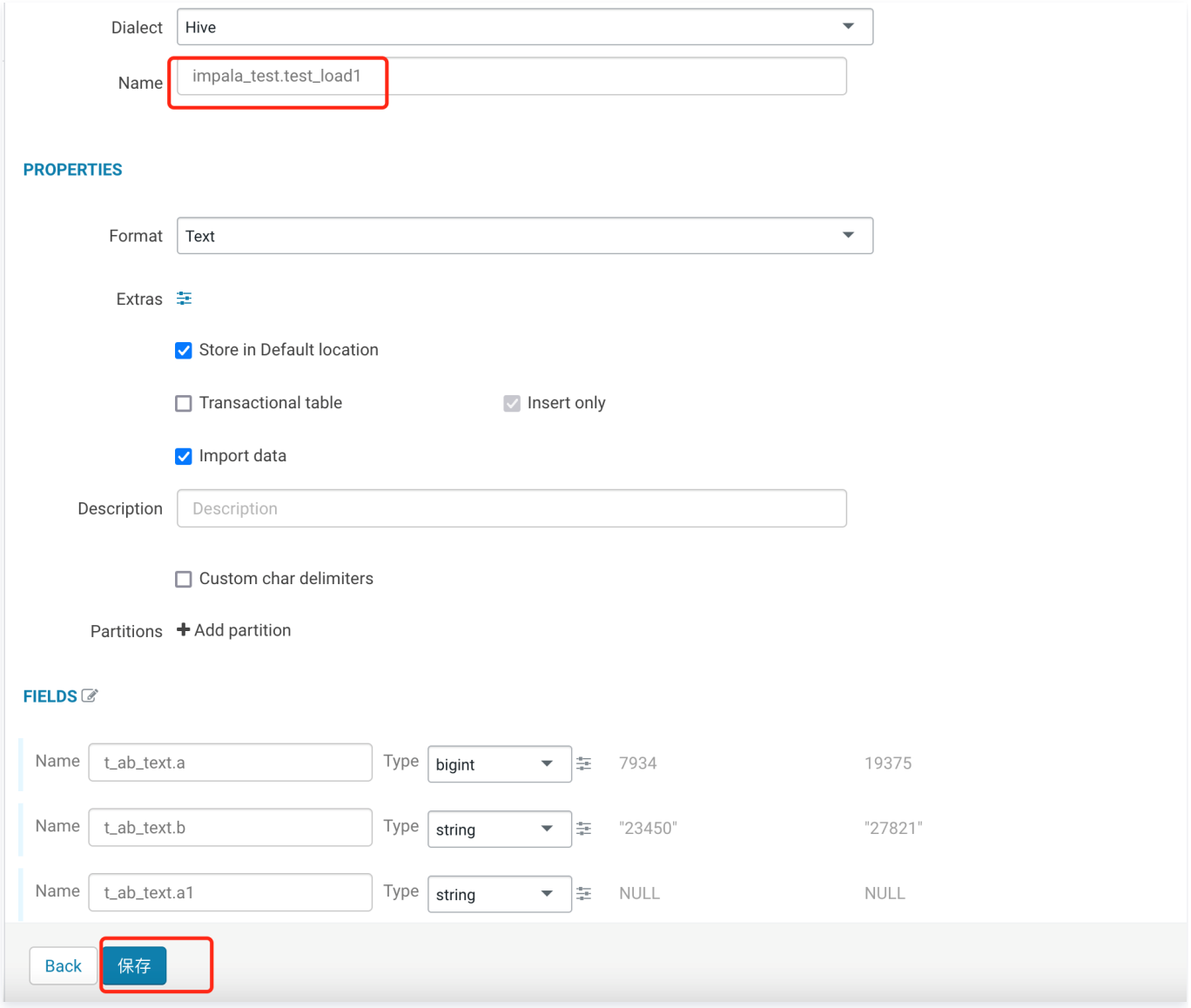


1. 本地文件导入。

1.1 单击**选择文件**，hue 会自动识别出分隔符并生成预览，单击 **Next** 导入到表。

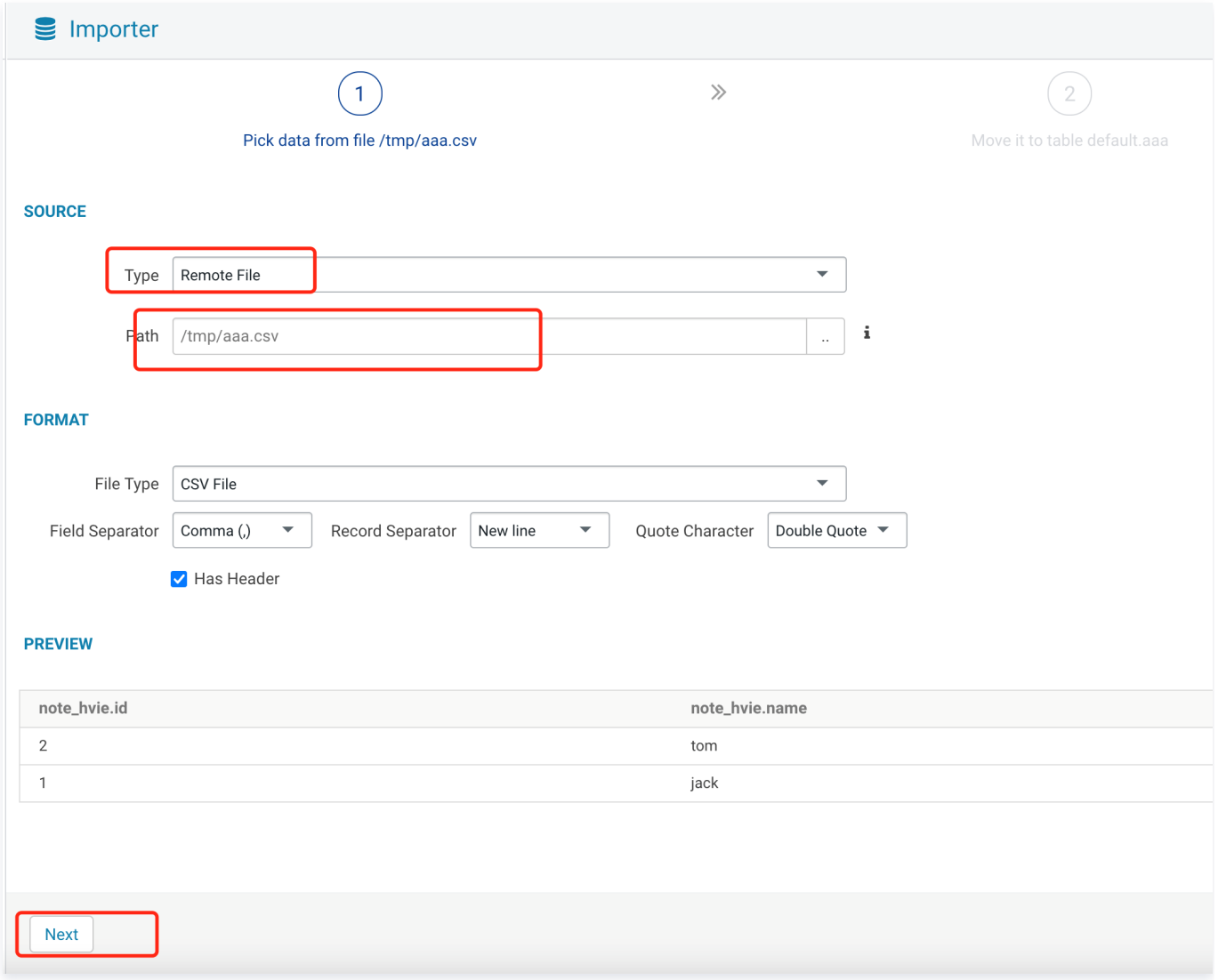


1.2 填写需要导入的表信息等，单击保存。

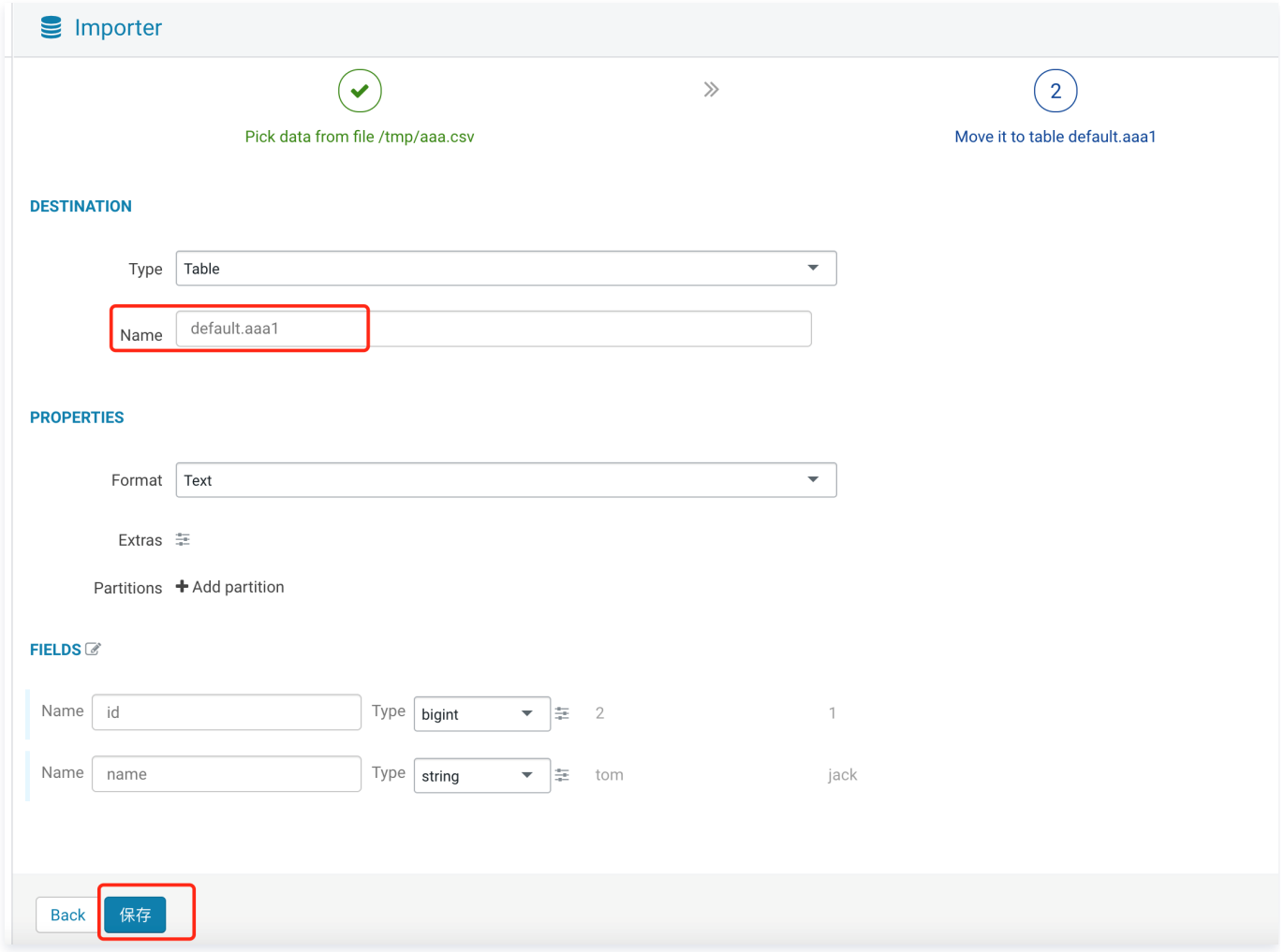


2. HDFS 文件导入。

2.1 选择 HDFS 上的 csv 文件。

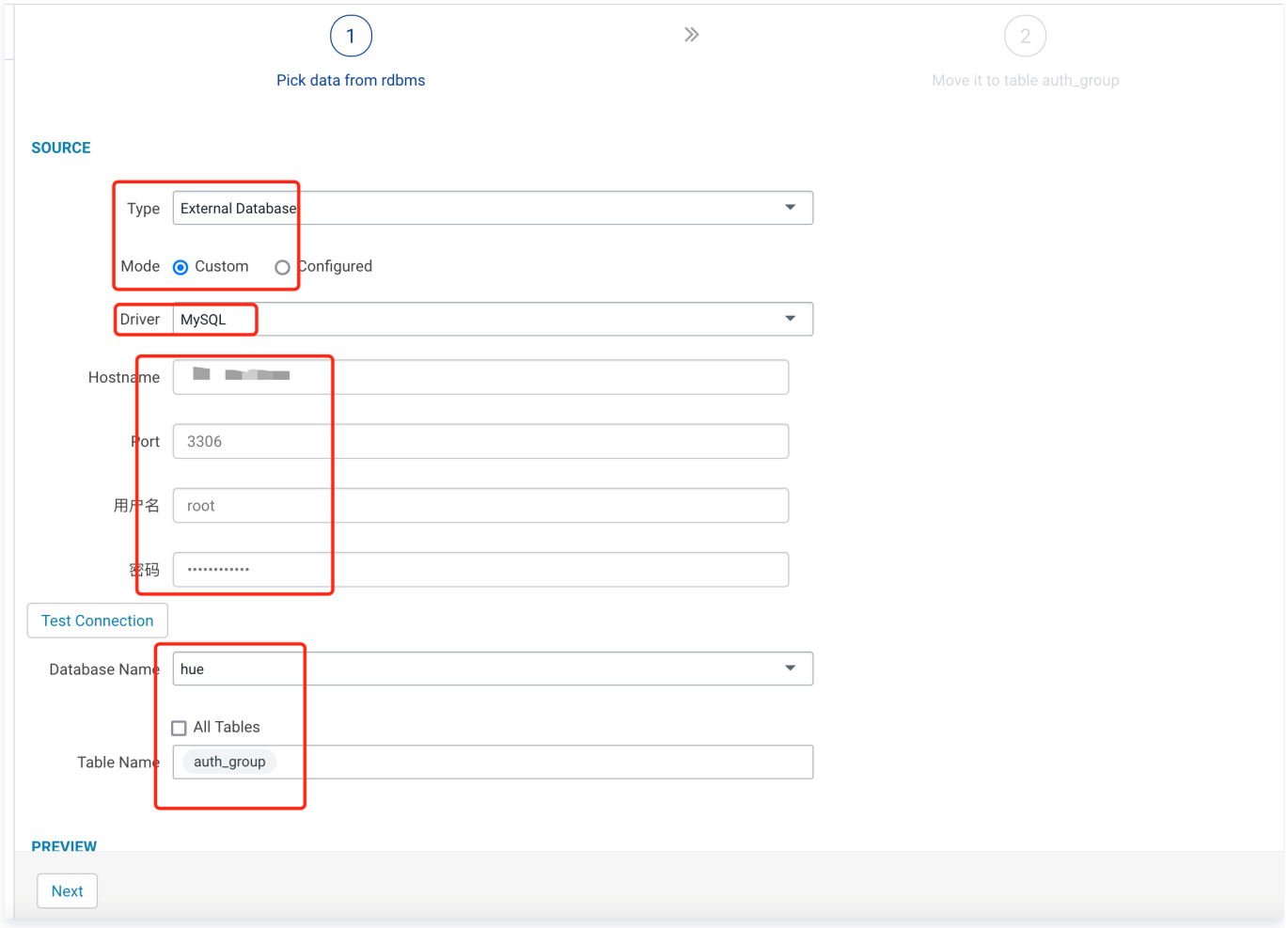


2.2 填写需要导入的表信息等，单击保存。

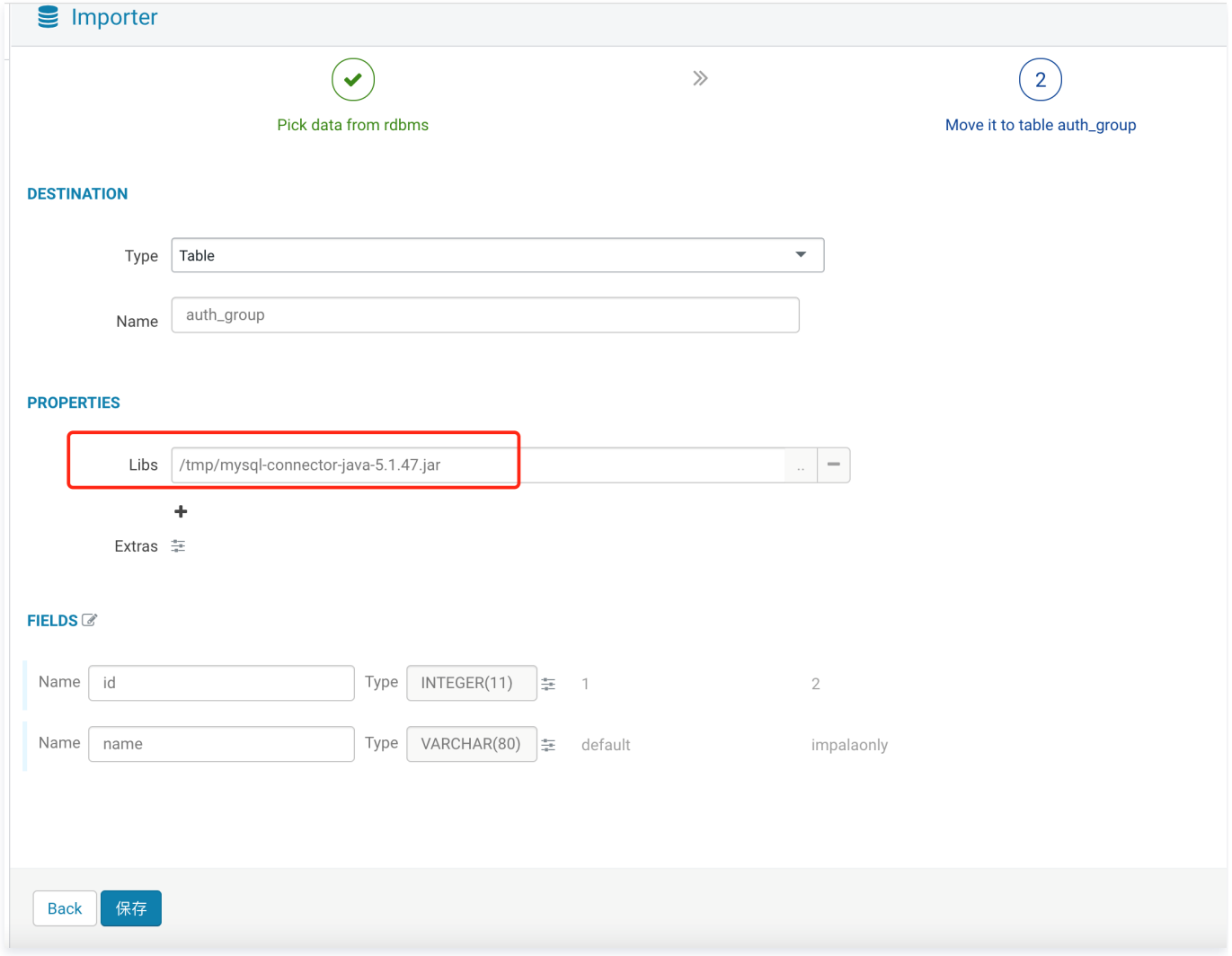


3. 外部数据库 External Database.

3.1 填写外部数据库信息，单击 **Test Connection** 获取到数据库信息，选择库和表后单击 Next。



3.2 填写需要导入的目的表信息，并单击 lib 选择 mysql 驱动，然后单击保存。



Job 管理

单击左侧的 Jobs标签，即可进入任务管理页面，单击上方的各个任务类型标签，可进行查看管理。

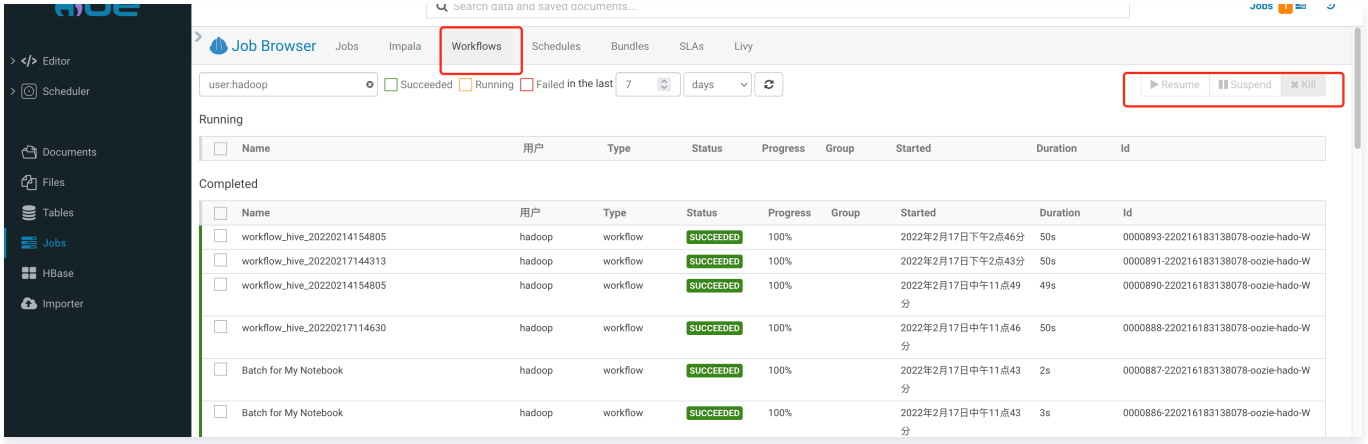
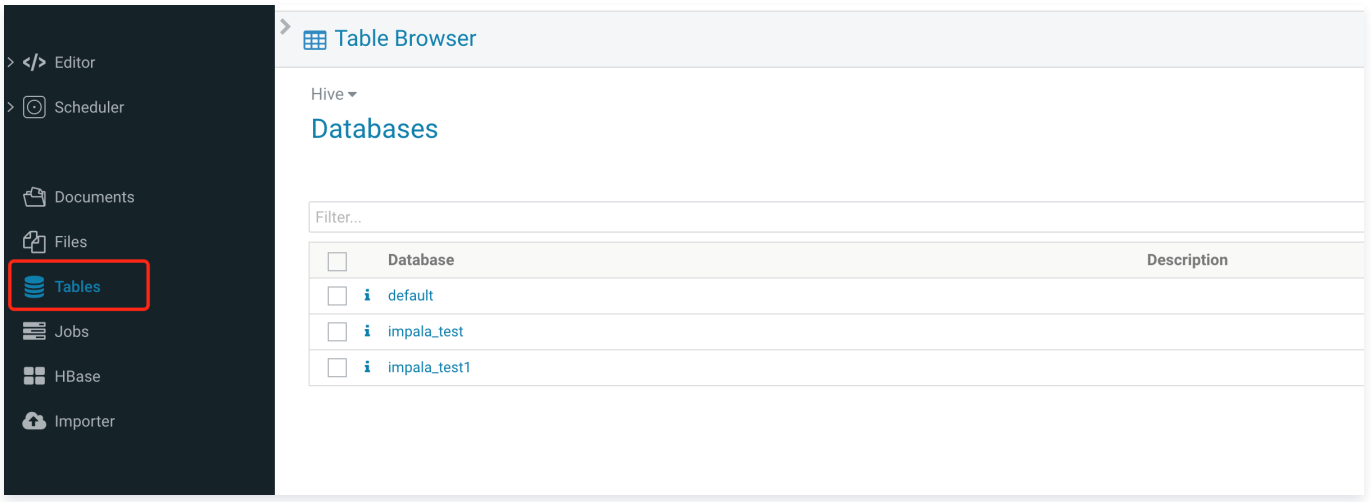
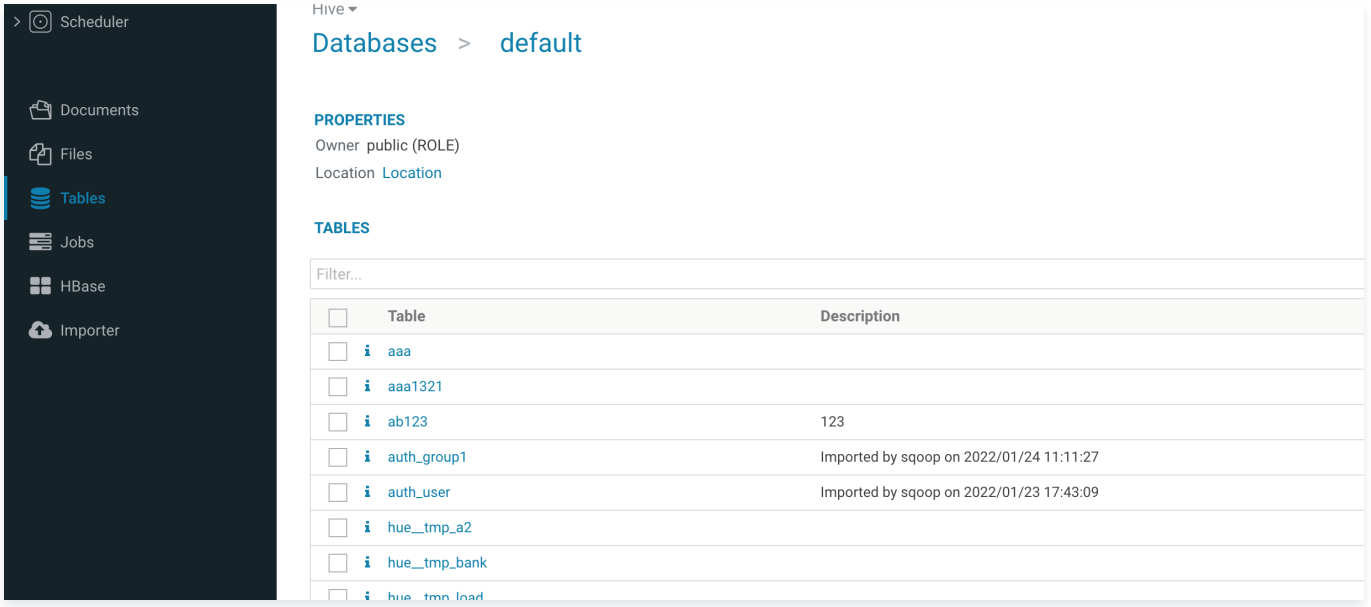


Table 管理

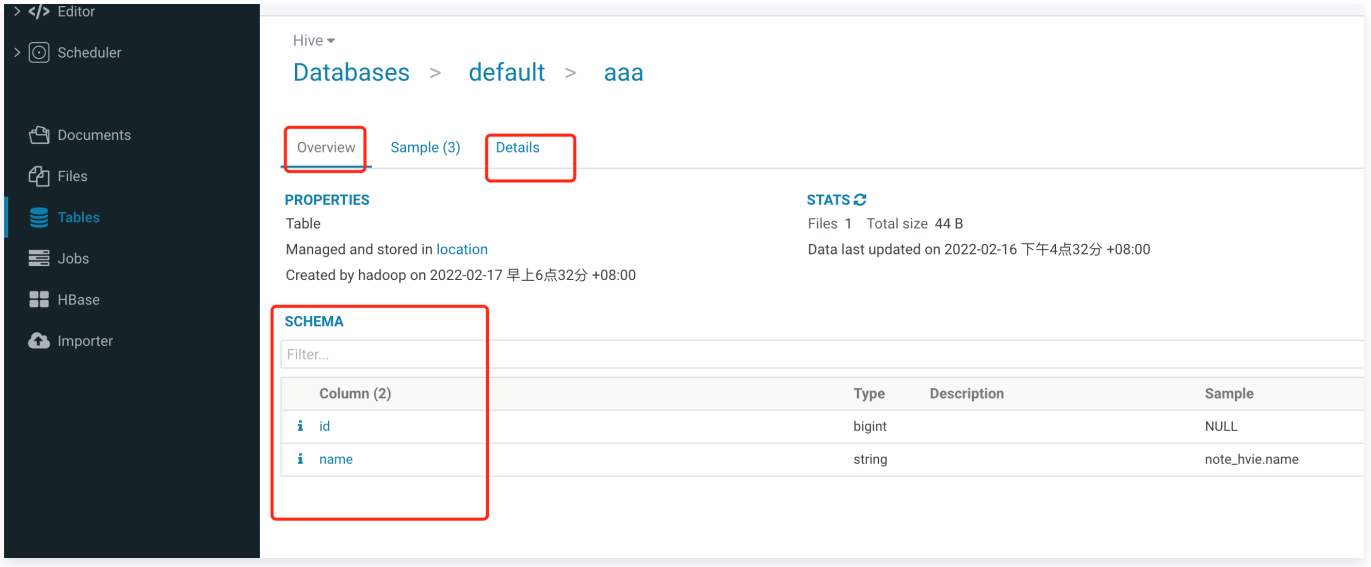
1. 单击左侧的 **Tables** 进入到 Table 管理页面，可以查看到基本的数据库信息。



2. 单击其中一个数据库，可查看此数据库的表。



3. 单击各个表，可以查看表的具体详细信息。



Hue 实践教程

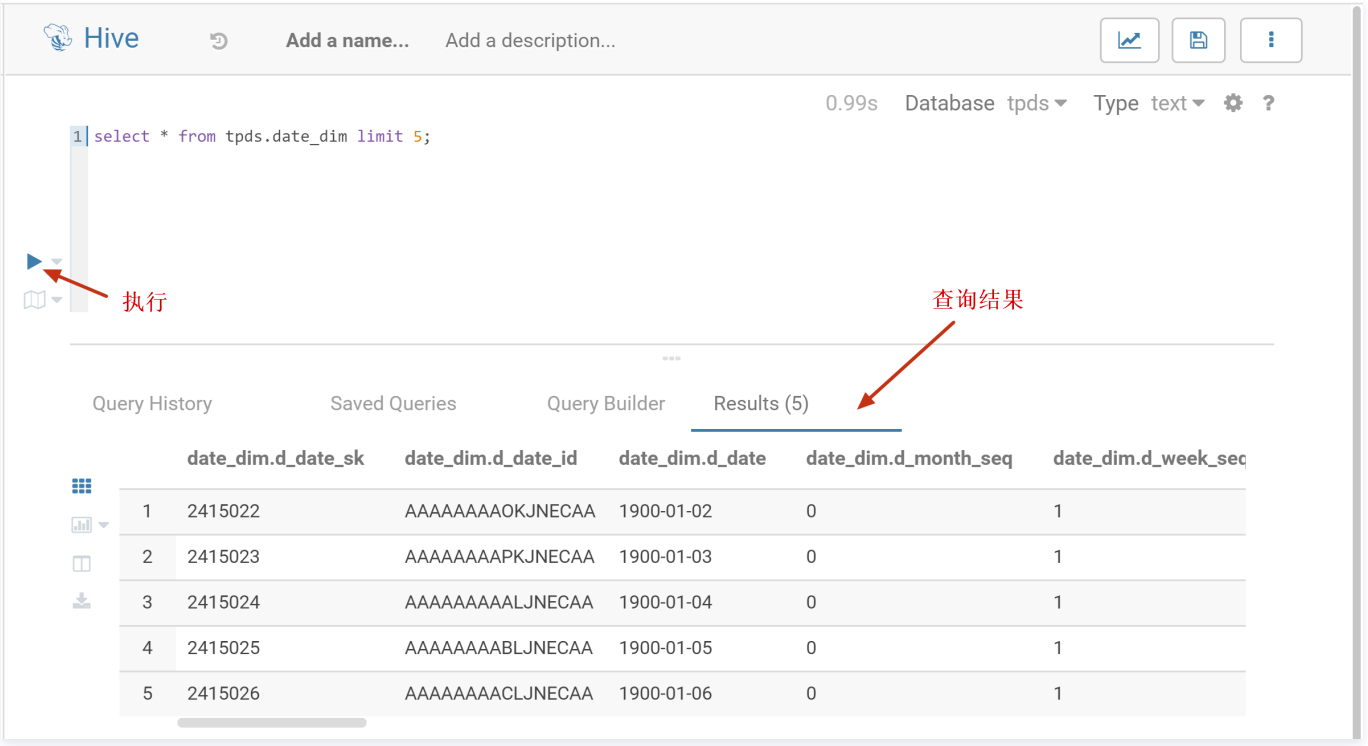
最近更新时间：2023-07-11 21:40:37

本文主要介绍 Hue 的实践用法。

Hive SQL 查询

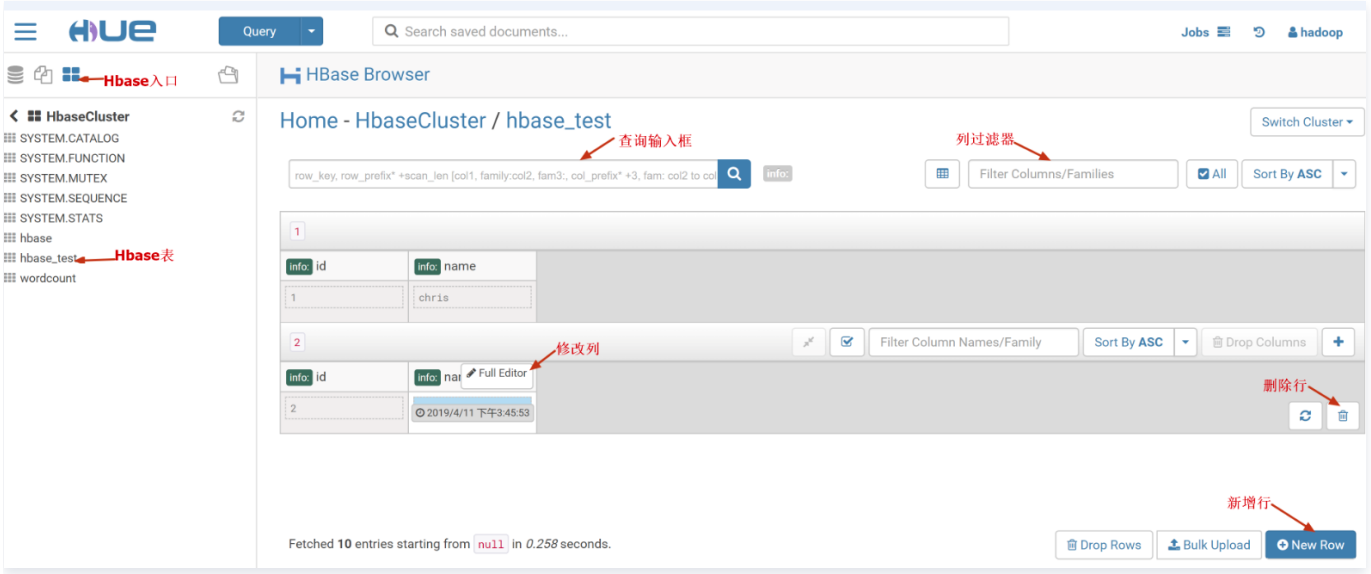
Hue 的 beeswax App 提供了友好方便的 Hive 查询功能，可以选择不同的 Hive 数据库、编写 HQL 语句、提交查询任务、查看结果。

1. 在 Hue 控制台上方，选择 Editor > Hive。
2. 在语句输入框中输入要执行语句，然后单击**执行**，执行语句。



Hbase 数据查询和修改、数据展示

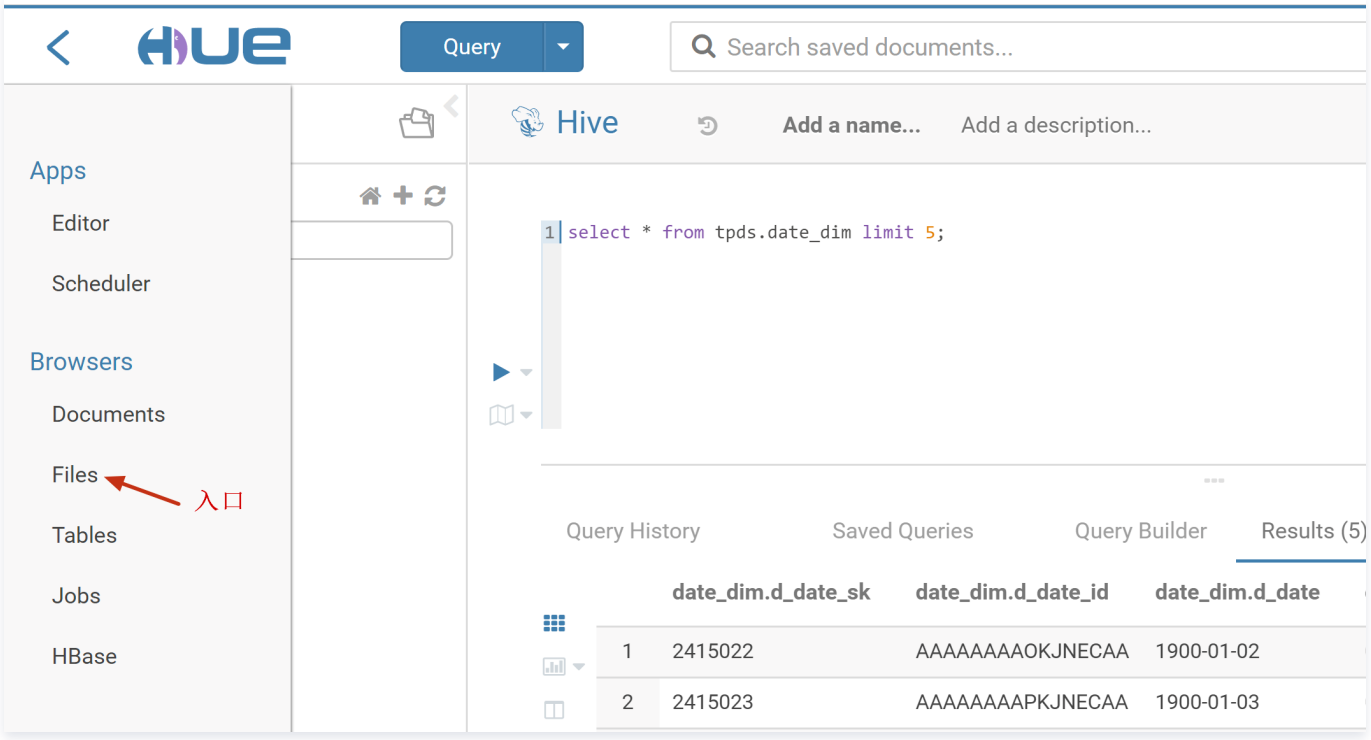
使用 Hbase Browser 可以查询、修改、展示 Hbase 集群中表的数据。



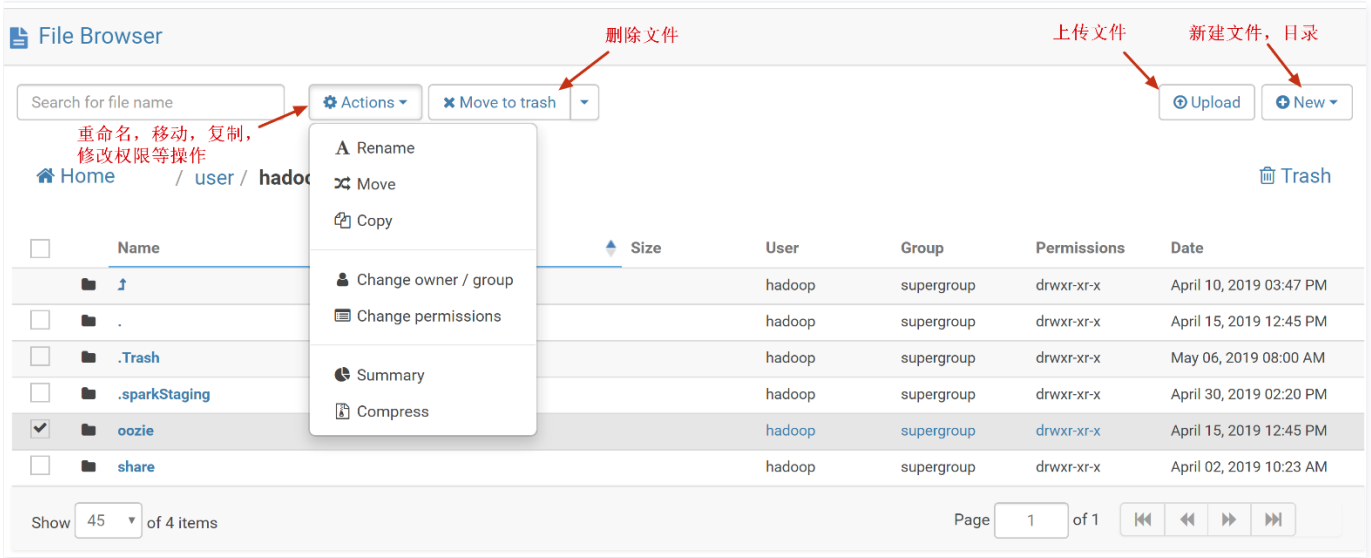
访问 HDFS 和文件浏览

通过 Hue 控制台左侧，选择 Files 进入 HDFS 文件浏览，可方便查看 HDFS 中的文件和文件夹，并对其进行创建、下载、上传、复制、修改和删除等操作。

1. 在 Hue 控制台左侧，选择 Browsers > Files 进入 HDFS 文件浏览。



在 Hue 控制台左侧，选择 Browsers > Files 进入 HDFS 文件浏览。



Oozie 任务的开发

1. 准备工作流数：Hue 的任务调度基于工作流，先创建一个包含 Hive script 脚本的工作流，Hive script 脚本的内容如下：

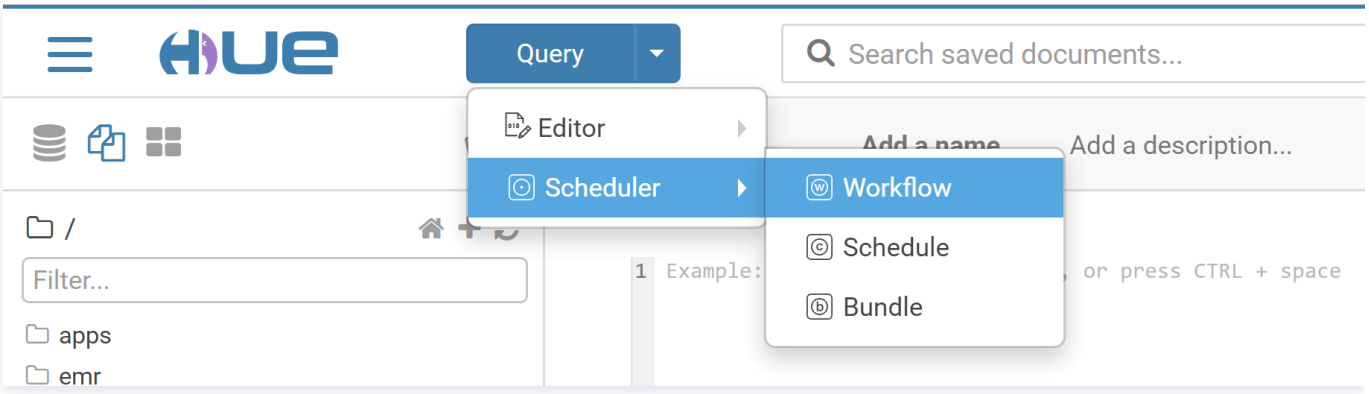
```
create database if not exists hive_sample;
show databases;
use hive_sample;
show tables;
create table if not exists hive_sample (a int, b string);
show tables;
insert into hive_sample select 1, "a";
```

```
select * from hive_sample;
```

将以上内容保存为 hive_sample.sql 文件。Hive 工作流还需要一个 hive-site.xml 配置文件，此配置文件可以在集群中安装了 Hive 组件的节点上找到。具体路径：/usr/local/service/hive/conf/hive-site.xml，复制一个 hive-site.xml 文件。然后上传 Hive script 文件和 hive-site.xml 到 hdfs 的目录，例如：/user/hadoop。

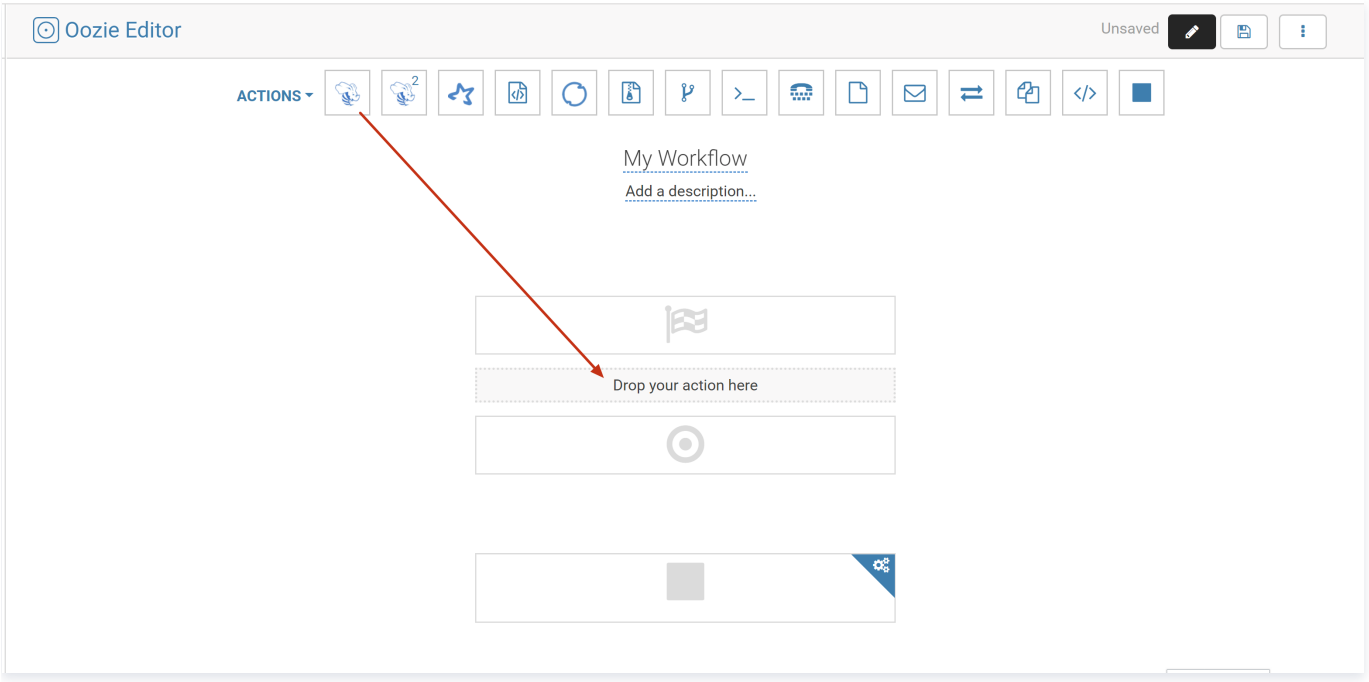
2. 创建工作流。

2.1 切换到 hadoop 用户，Hue 控制台左侧选择 Scheduler > Workflow。

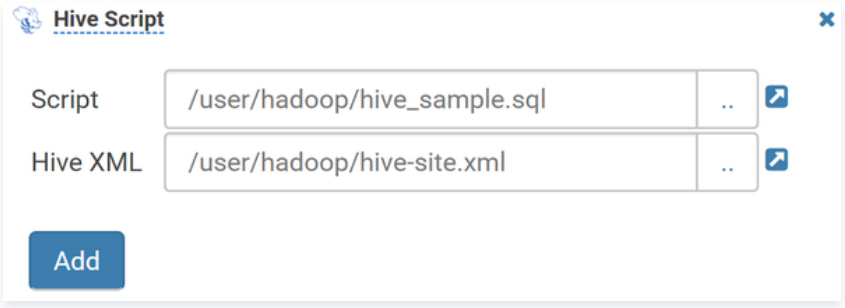


2.2 在工作流编辑页面中拖一个 Hive Script。

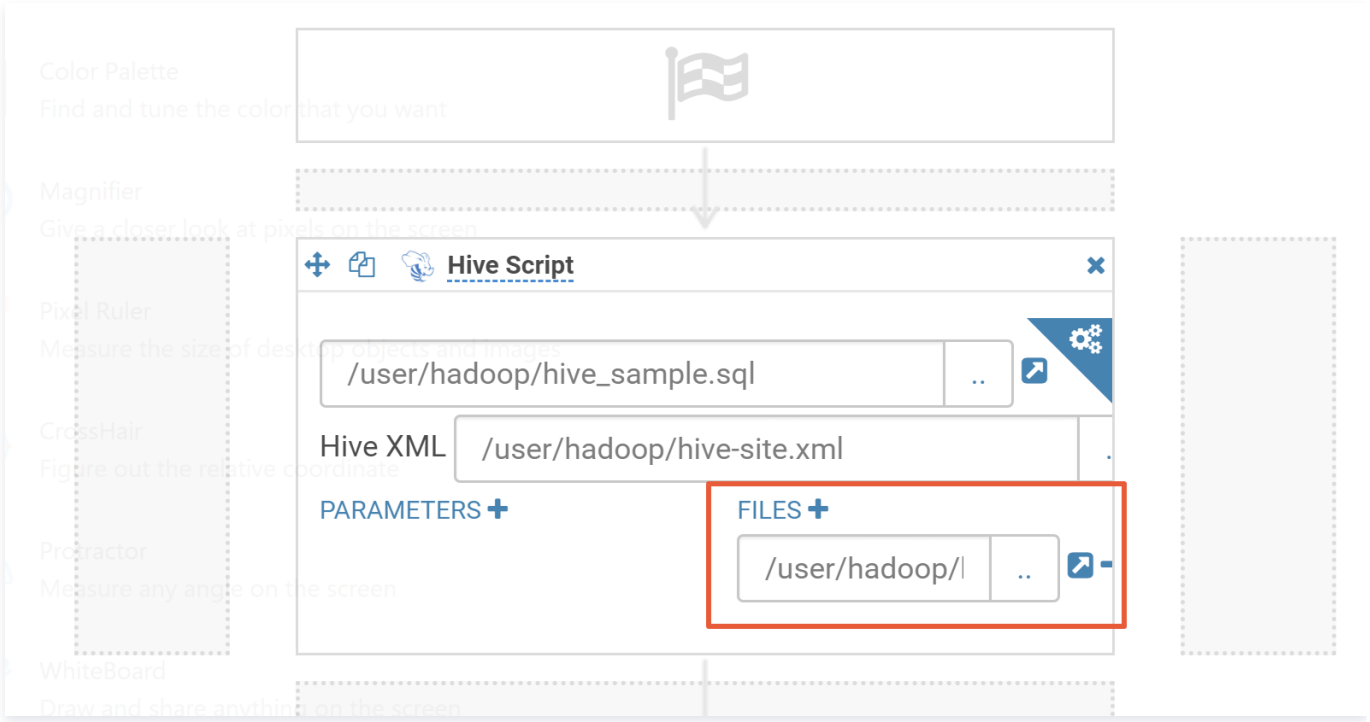
> **注意**
> 本文以安装 Hive 版本为 Hive1 为例，配置参数为 HiveServer1。与其他 Hive 版本混合部署时（即配置其他版本的配置参数时），会报错。



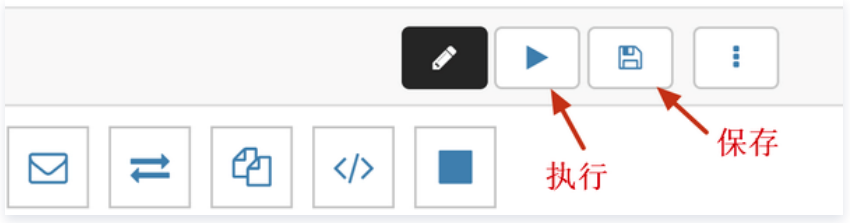
2.3 选择刚上传的 Hive scipt 文件和 hive-site.xml 文件。



2.4 单击 Add 后，还需在 FILES 中指定 hive script 文件。



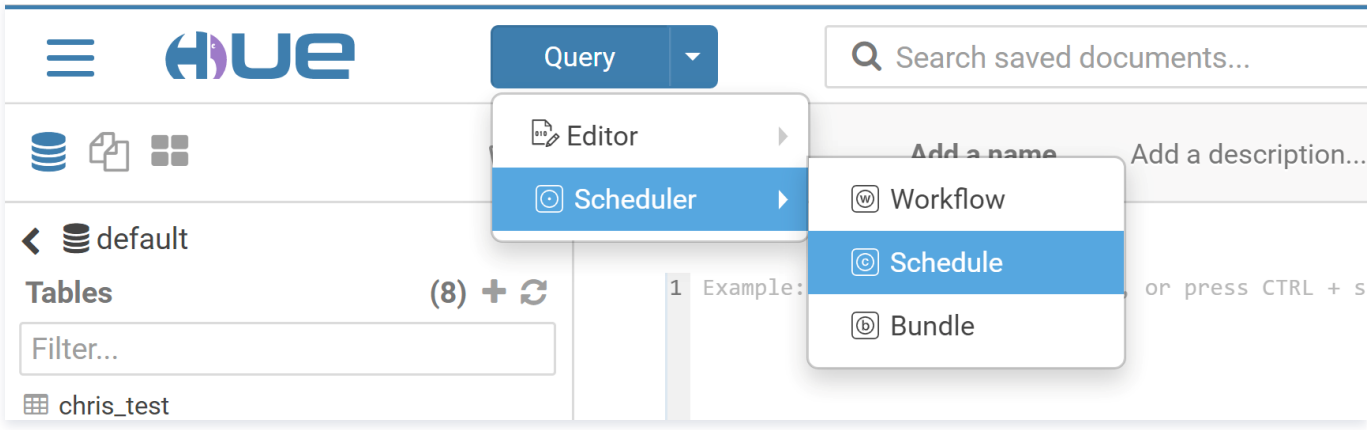
2.5 单击右上角**保存**，然后单击**执行**，运行 workflow。



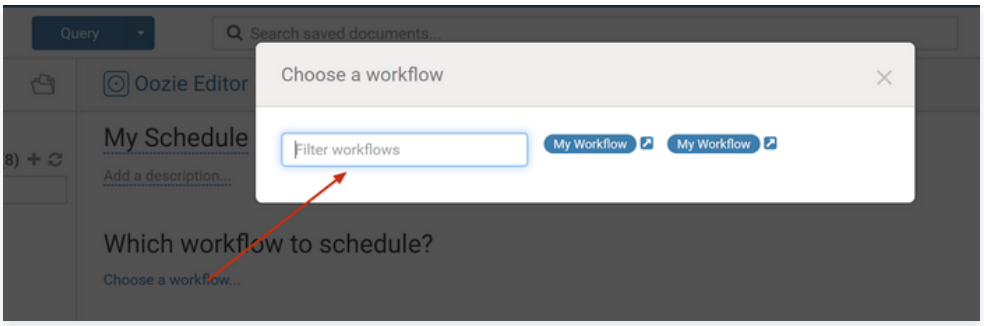
3. 创建定时调度任务。

Hue 的定时调度任务是 schedule，类似于 Linux 的 crontab，支持的调度粒度可以到分钟级别。

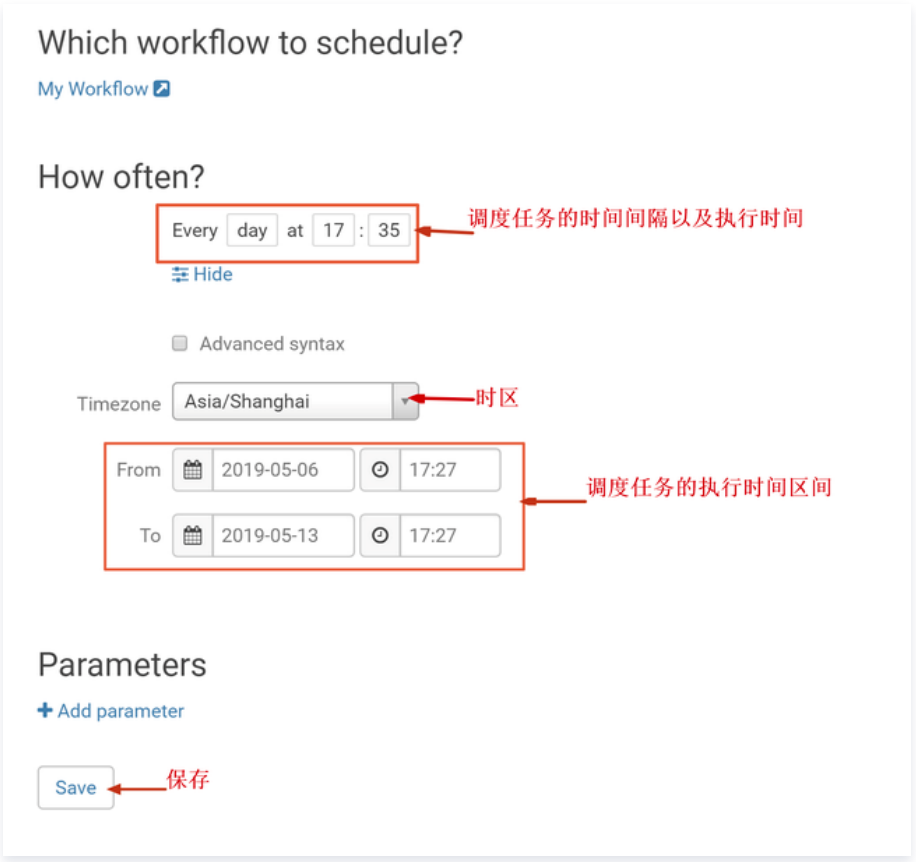
3.1 左侧选择 Scheduler > Schedule，创建 Schedule。



3.2 单击 **Choose a workflow**，选择一个创建好的工作流。

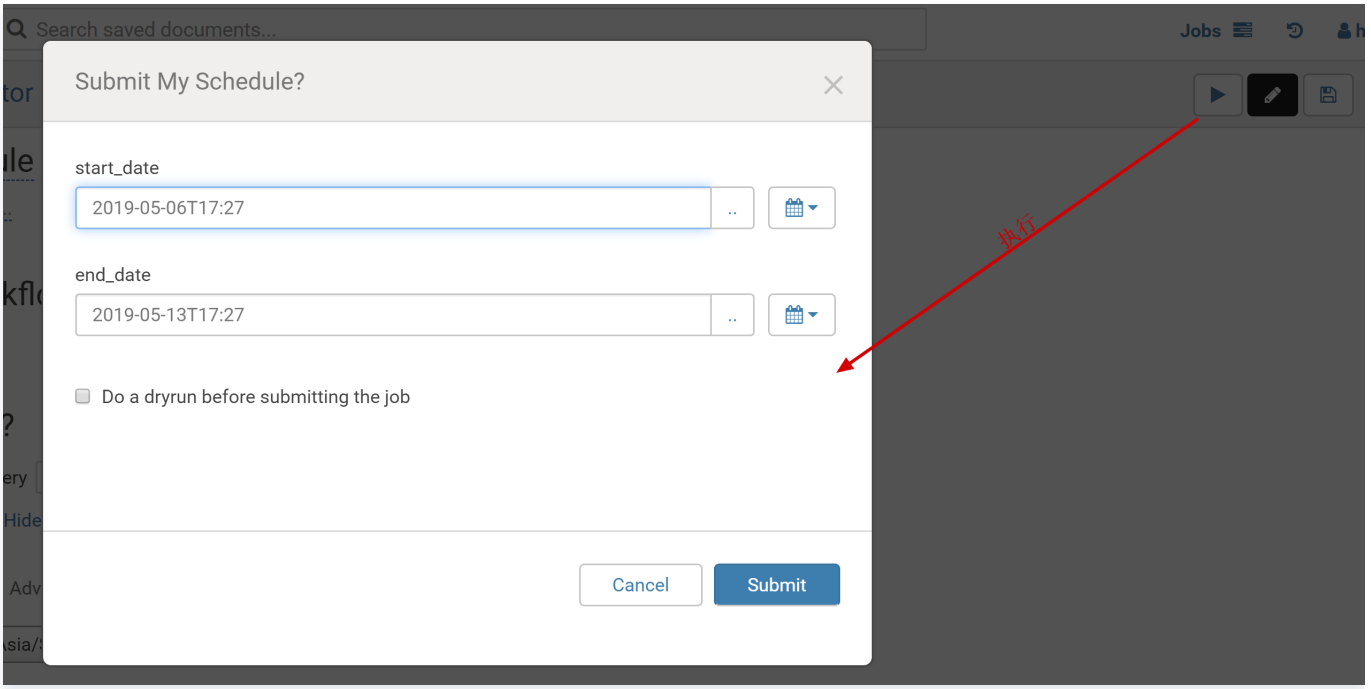


3.3 选择需要调度的时间点和时间间隔、时区、调度任务的开始时间和结束时间，然后单击 **Save** 保存。

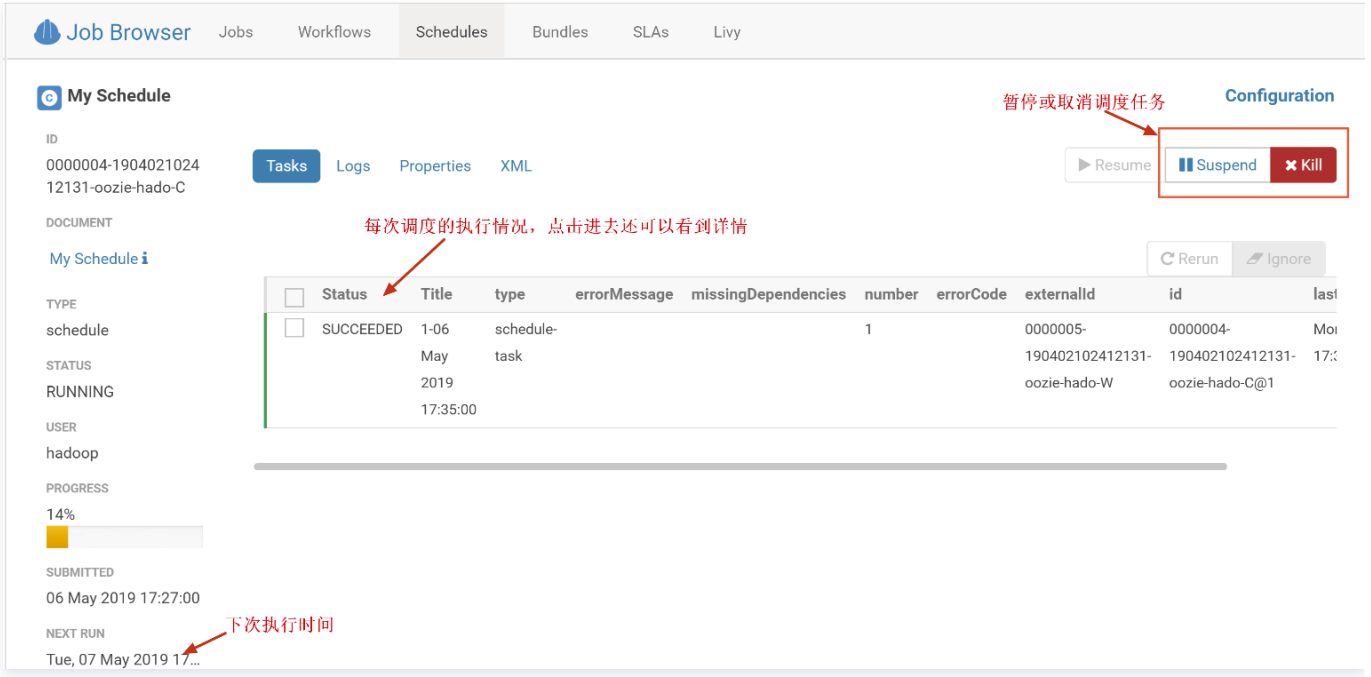


4. 创建定时调度任务。

4.1 单击右上角的**提交**，提交调度任务。



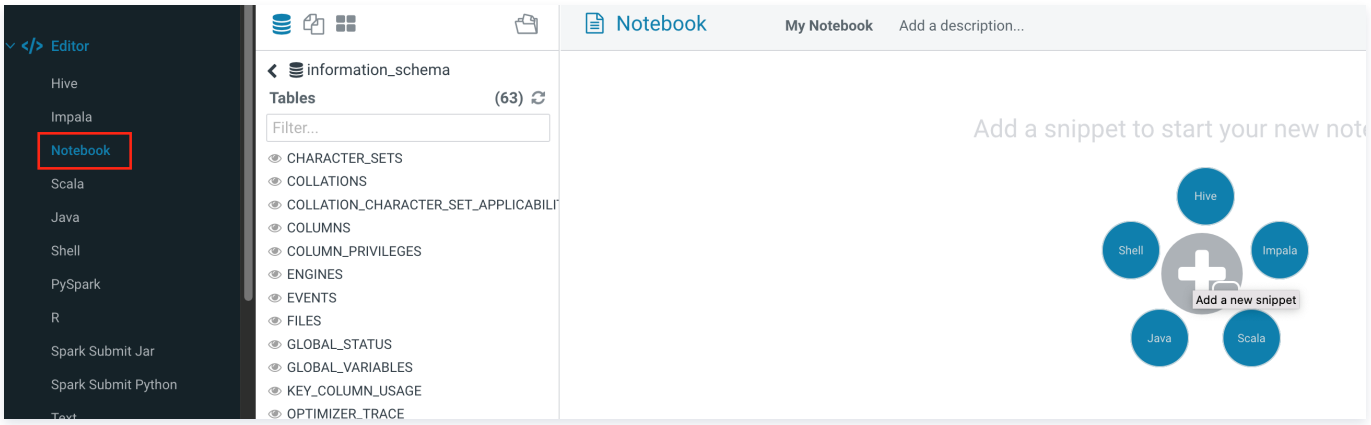
4.2 在 schedulers 的监控页面可以查看任务调度情况。



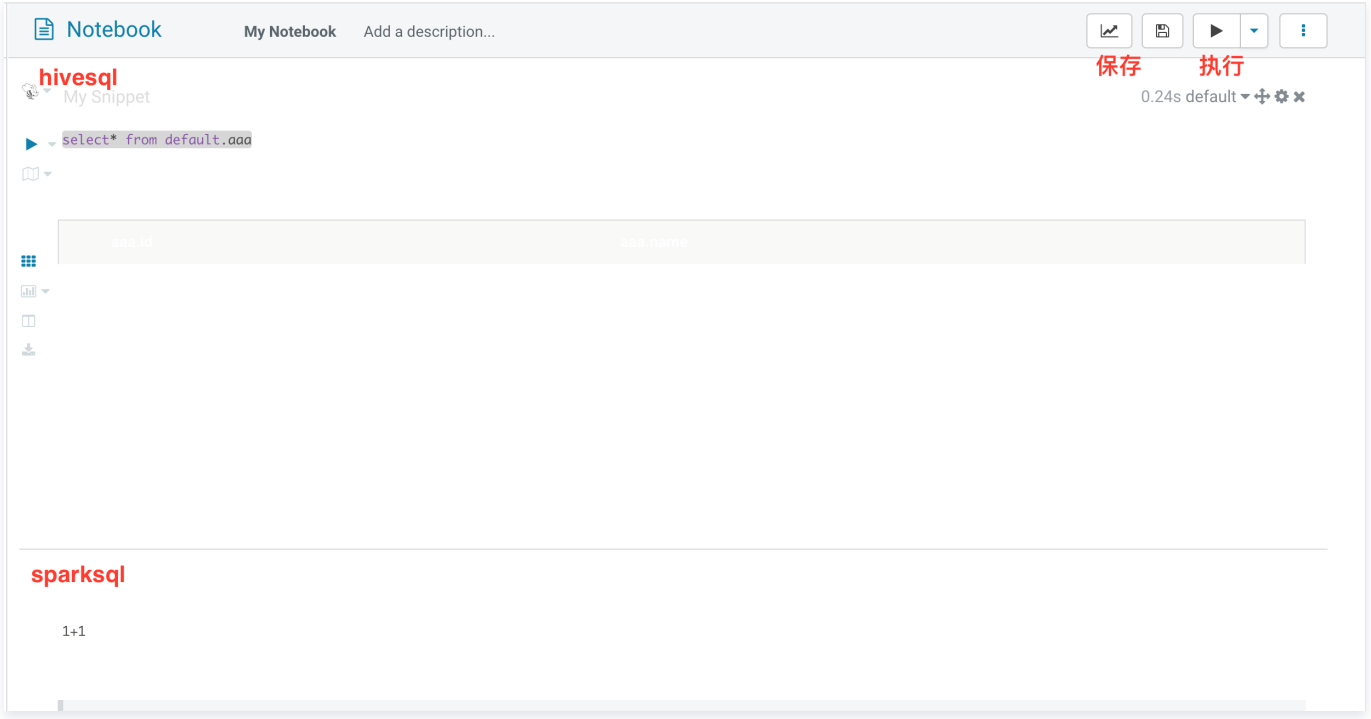
Notebook 查询对比分析

notebook 可以快速的构建间的查询，将查询结果放到一起做对比分析，支持5种类型：hive，impala，spark，java，shell。

1. 依次单击 Editor，Notebook，单击"+" 号添加需要的查询。



2. 依次单击**保存**可保存添加的notebook，单击**执行**可以执行整个 notebook。



Oozie 开发指南

最近更新时间：2023-08-28 10:17:51

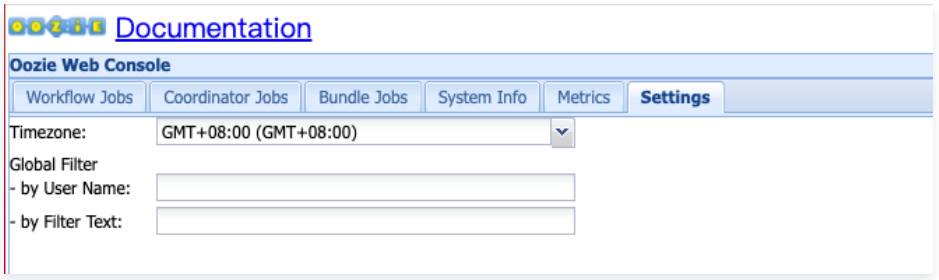
Apache Oozie 是一个开源的工作流引擎，被设计将 hadoop 生态组件的任务编排成 Workflow，然后对其进行调度、执行、管理。本文通过示例简单介绍如何在 EMR 上使用 Oozie，详细的使用文档可进一步参考社区文档。
另外这里建议用户通过 Hue 的图像化界面来使用 Oozie，使用文档请移步 Hue 开发文档。

前提条件

已创建弹性 MapReduce（简称 EMR）的 Hadoop 集群，并选择了 Oozie 、Hive 、Spark 服务，详情请参见 创建 EMR 集群。

访问 Oozie WebUI

- 如果您在购买集群时勾选了开启集群节点外网，可以通过 EMR 控制台 **集群服务** 单击 WebUI 链接来访问。
- 对于国内用户，建议将 WebUI 时区设置为 GMT+08:00。



sharelib 的更新

在 EMR 集群中，已安装了 sharelib，所以您使用 Oozie 提交 Workflow 作业时，不需要再安装 sharelib。当然您也可以对 sharelib 进行编辑与更新，操作步骤如下：

```
cd /usr/local/service/oozie
tar -xf oozie-sharelib.tar.gz
```

添加 jar 包到解压出的 share 目录下要支持的 action 对应的目录下：

```
bin/oozie-setup.sh sharelib create -fs hdfs://active-namenode-ip:4007 -locallib shareoozie admin --oozie
http://oozie-server-ip:12000/oozie -sharelibupdate
```

提交 Workflow 示例

EMR 集群中 Oozie 的安装目录为/usr/local/service/oozie，里面有 Oozie 支持的组件的 Workflow 示例 oozie-examples.tar.gz。进入安装目录后，切换 Hadoop 用户通过以下命令解压示例文件并上传到 hdfs；

```
su hadoop
tar -xf oozie-examples.tar.gz
hadoop fs -put examples
```

说明：
以下示例中参数：
1、**$fs** 为 HDFS 的访问链接，可以在 EMR 控制台集群详情页通过 **集群服务>HDFS>配置管理** 中查看 core-site.xml 配置文件 fs.defaultFS 的值。
2、**$rm** 为 Yarn ResourceManager 的地址，可以在 EMR 控制台集群详情页通过 **集群服务>YARN>配置管理** 中查看 yarn-site.xml 配置文件，高可用部署下为 yarn.resourcemanager.ha.rm-ids 的值，非高可用部署下为 yarn.resourcemanager.address 的值。
3、**$hs2_ip** 为 HiveSever2 服务部署节点的内网 ip，可以在集群详情页中 **集群服务>HIVE>角色管理** 中查看。
4、**$oozie_server_ip** 为 Oozie 服务部署节点的内网 ip，可以在集群详情页中 **集群服务>OOZIE>角色管理** 中查看。
5、**$job** 为提交 Oozie Job 后返回的 id。

Hive 任务示例

示例文件 ./examples/apps/hive2 下 job.properties 为 Hive 示例任务的配置文件，根据实际情况修改该文件中的 HDFS 和 YARN 地址、HiveServer2 地址等：

```
nameNode=$fs
resourceManager=$rm
queueName=default
jdbcURL=jdbc:hive2://$hs2_ip:7001/default
examplesRoot=examples
```

提交 Oozie job：

```
oozie job -debug -oozie http://$oozie_server_ip:12000/oozie -config examples/apps/hive2/job.properties -run
```

可以执行以下命令或访问 Oozie web ui 查看 job 运行信息。

```
oozie job -info $job
```

Spark 任务示例

示例文件 ./examples/apps/spark 下 job.properties 为 Spark 示例任务的配置文件，根据实际情况修改该文件中的 HDFS 和 YARN 地址等：

```
nameNode=$fs
resourceManager=$rm
master=local[*]
mode=client
queueName=default
examplesRoot=examples
oozie.use.system.libpath=true
```

提交 Oozie job：

```
oozie job -debug -oozie http://$oozie_server_ip:12000/oozie -config examples/apps/spark/job.properties -run
```

可以执行以下命令或访问 Oozie web ui 查看 job 运行信息。

```
oozie job -info $job
```

Flume 开发指南
Flume 简介

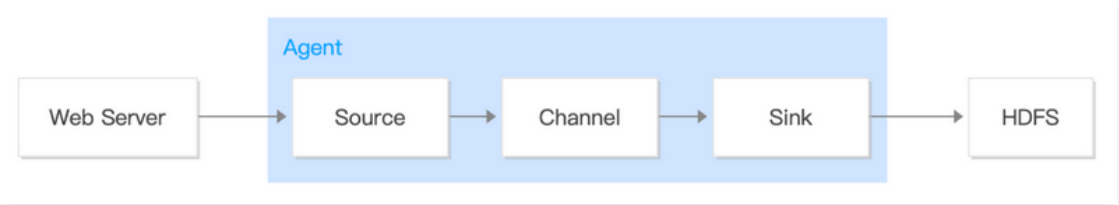
最近更新时间：2024-08-15 14:44:01

Flume 简介

Apache Flume 是可以收集例如日志、事件等数据资源，并将这些数量庞大的数据从各项数据资源中集中起来存储的工具/服务。Flume 具有高可用、分布式、配置工具等特性，其设计原理也是将数据流（例如日志数据）从各种网站服务器上汇集起来存储到 HDFS、HBase 等集中存储器中。

Flume 架构

一个 Flume 事件被定义为一个数据流单元。Flume agent 其实是一个 JVM 进程，该进程中包含完成任务所需要的各个组件，其中最核心的三个组件是 Source、Channel 以及 Sink。



- **Source**
  消费外部源（例如 Web 服务器或者其他 Source）传递给它的事件，并将其保存到 Channel（一个或多个）中。
- **Channel**
  Channel 位于 Source 和 Sink 之间，用于缓存进来的 events，当 Sink 成功的将 events 发送到下一跳的 Channel 或最终目的，events 从 Channel 移除。
- **Sink**
  Sink 负责将 events 传输到下一跳或最终目的，成功完成后将 events 从 Channel 移除。

使用指南

使用准备

- 已创建一个 EMR 集群。创建 EMR 集群 时需要在软件配置界面选择 flume 组件。
- flume 安装在 EMR 云服务器（core 节点和 task 节点）的 `/usr/local/service/flume` 路径下；master 节点的安装路径是 `/usr/local/service/apps/` 。

配置 Flume

进入 `/usr/local/service/flume` 文件夹，并创建 example.conf 文件。

```
[hadoop@10 /usr/local/service/flume]$ cd /usr/local/service/flume/
[hadoop@10 /usr/local/service/flume]$ vim example.conf
[hadoop@10 /usr/local/service/flume]$
```

```
# example.conf: A single-node Flume configuration

# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Describe the sink
a1.sinks.k1.type = logger
```

```
# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

启动 Flume

```
bin/flume-ng agent --conf conf --conf-file example.conf --name a1 -Dflume.root.logger=INFO,console
```

配置测试用例

配置后将会看到之前启动的 Flume Agent 向终端打印。

```
telnet localhost 44444
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Hello world! <ENTER>
OK
```

Kafka 数据通过 Flume 存储到 Hive

最近更新时间：2023-07-05 16:20:08

场景说明

将 Kafka 中的数据通过 Flume 收集并存储到 Hive。

开发准备

- 因为任务中需要访问腾讯云消息队列 CKafka，所以需要先创建一个 CKafka 实例，具体见 消息队列 CKafka 。
- 确认您已开通腾讯云，且已创建一个 EMR 集群。创建 EMR 集群时，需要在软件配置界面选择 Flume 组件。

在 EMR 集群使用 Kafka 工具包

首先需要查看 CKafka 的内网 IP 与端口号。登录消息队列 CKafka 的控制台，选择您要使用的 CKafka 实例，在**基本信息**中查看其内网 IP 为 $kafkaIP，而端口号一般默认为9092。在 topic 管理界面新建一个 topic 为 kafka_test。

配置 flume

1. 创建 flume 的配置文件 `hive_kafka.properties`

```
vim hive_kafka.properties
agent.sources = kafka_source
agent.channels = mem_channel
agent.sinks = hive_sink
# 以下配置 source
agent.sources.kafka_source.type = org.apache.flume.source.kafka.KafkaSource
agent.sources.kafka_source.channels = mem_channel
agent.sources.kafka_source.batchSize = 5000
agent.sources.kafka_source.kafka.bootstrap.servers = $kafkaIP:9092
agent.sources.kafka_source.kafka.topics = kafka_test
# 以下配置 sink
agent.sinks.hive_sink.channel = mem_channel
agent.sinks.hive_sink.type = hive
agent.sinks.hive_sink.hive.metastore = thrift://172.16.32.51:7004
agent.sinks.hive_sink.hive.database = default
agent.sinks.hive_sink.hive.table = weblogs
agent.sinks.hive_sink.hive.partition = asia,india,%y-%m-%d-%H-%M
agent.sinks.hive_sink.useLocalTimeStamp = true
agent.sinks.hive_sink.round = true
agent.sinks.hive_sink.roundValue = 10
agent.sinks.hive_sink.roundUnit = minute
agent.sinks.hive_sink.serializer = DELIMITED
agent.sinks.hive_sink.serializer.delimiter = ","
agent.sinks.hive_sink.serializer.serdeSeparator = ','
agent.sinks.hive_sink.serializer.fieldnames =id,msg
# 以下配置 channel
agent.channels.mem_channel.type = memory
agent.channels.mem_channel.capacity = 100000
agent.channels.mem_channel.transactionCapacity = 100000
```

其中 hive.metastore 可以通过以下方式确认：

```
grep "hive.metastore.uris" -C 2 /usr/local/service/hive/conf/hive-site.xml
```

```
<property>
<name>hive.metastore.uris</name>
<value>thrift://172.16.32.51:7004</value>
</property>
```

2. 创建 hive 表

```
create table weblogs ( id int , msg string )
partitioned by (continent string, country string, `time` string)
clustered by (id) into 5 buckets
stored as orc TBLPROPERTIES ('transactional'='true');
```

> **注意**
> 一定要是分区且分桶的表，存储为 orc 且设置 TBLPROPERTIES ('transactional'='true')，以上条件缺一不可。

3. 开启 hive 事务
在控制台给 `hive-site.xml` 添加以下配置项。

```
<property>
<name>hive.support.concurrency</name>
<value>true</value>
</property>
<property>
<name>hive.exec.dynamic.partition.mode</name>
<value>nonstrict</value>
</property>
<property>
<name>hive.txn.manager</name>
<value>org.apache.hadoop.hive.ql.lockmgr.DbTxnManager</value>
</property>
<property>
<name>hive.compactor.initiator.on</name>
<value>true</value>
</property>
<property>
<name>hive.compactor.worker.threads</name>
<value>1</value>
</property>
<property>
<name>hive.enforce.bucketing</name>
<value>true</value>
</property>
```

> **注意**
> 配置下发并重启后，在 `hadoop-hive` 日志中会提示 metastore 无法连接，请忽略该错误。由于进程启动顺序导致，需先启动 metastore 再启动 hiveserver2。

4. 复制 hive 的 `hive-hcatalog-streaming-xxx.jar` 到 flume 的 lib 目录

```
cp -ra /usr/local/service/hive/hcatalog/share/hcatalog/hive-hcatalog-streaming-2.3.3.jar
/usr/local/service/flume/lib/
```

5. 运行 flume

```
./bin/flume-ng agent --conf ./conf/ -f hive_kafka.properties -n agent -Dflume.root.logger=INFO,console
```

6. 运行 kafka producer

```
[hadoop@172 kafka]$ ./bin/kafka-console-producer.sh --broker-list $kafkaIP:9092 --topic kafka_test
1,hello
2,hi
```

测试

- 在 kafka 生产者客户端输入信息并回车。
- 观察 hive 表中是否有相应数据。

参考文档

- hive-sink 配置说明
- hive 日志配置说明

Kafka 数据通过 Flume 存储到 HDFS 或 COS

最近更新时间：2024-01-05 14:11:21

场景说明

将 Kafka 中的数据通过 Flume 收集并存储到 HDFS 或 COS。

开发准备

- 因为任务中需要访问腾讯云消息队列 CKafka，所以需要先创建一个 CKafka 实例，具体见 消息队列 CKafka 。
- 确认您已开通腾讯云，且已创建一个 EMR 集群，详情参考创建集群 。创建 EMR 集群时，在软件配置界面选择 Flume 组件。
- 示例中存在需要访问腾讯云对象存储 COS 的可选内容，可参考 创建存储桶 在 COS 中创建一个存储桶（Bucket），并于 EMR 控制台 实例信息 页面开启对象存储授权。

在 EMR 集群使用 Kafka 工具包

首先需要查看 CKafka 的内网 IP 与端口号。登录消息队列 CKafka 的控制台，选择您要使用的 CKafka 实例，在基本消息中查看其内网 IP 为 $kafkaIP，而端口号一般默认为9092。在 topic 管理界面新建一个 topic 为 kafka_test。

配置 flume

1. 创建 flume 的配置文件 `kafka.properties`

```
vim kafka.properties
agent.sources = kafka_source
agent.channels = mem_channel
agent.sinks = hdfs_sink
# 以下配置 source
agent.sources.kafka_source.type = org.apache.flume.source.kafka.KafkaSource
agent.sources.kafka_source.channels = mem_channel
agent.sources.kafka_source.batchSize = 5000
agent.sources.kafka_source.kafka.bootstrap.servers = $kafkaIP:9092
agent.sources.kafka_source.kafka.topics = kafka_test
# 以下配置 sink
agent.sinks.hdfs_sink.type = hdfs
agent.sinks.hdfs_sink.channel = mem_channel
agent.sinks.hdfs_sink.hdfs.path = /data/flume/kafka/%Y%m%d（或cosn://bucket/xxx）
agent.sinks.hdfs_sink.hdfs.rollSize = 0
agent.sinks.hdfs_sink.hdfs.rollCount = 0
agent.sinks.hdfs_sink.hdfs.rollInterval = 3600
agent.sinks.hdfs_sink.hdfs.threadsPoolSize = 30
agent.sinks.hdfs_sink.hdfs.fileType=DataStream
agent.sinks.hdfs_sink.hdfs.useLocalTimeStamp=true
agent.sinks.hdfs_sink.hdfs.writeFormat=Text
# 以下配置 channel
agent.channels.mem_channel.type = memory
agent.channels.mem_channel.capacity = 100000
agent.channels.mem_channel.transactionCapacity = 10000
```

2. 运行 flume

```
./bin/flume-ng agent --conf ./conf/ -f kafka.properties -n agent -Dflume.root.logger=INFO,console
```

3. 运行 kafka producer

```
[hadoop@172 kafka]$ ./bin/kafka-console-producer.sh --broker-list $kafkaIP:9092 --topic kafka_test
test
hello
```

测试

- 在 kafka 生产者客户端输入信息并回车。
- 观察 hdfs 是否生成相应目录和文件 `hadoop fs -ls /data/flume/kafka/` 。

参考文档

kafka-source 配置说明

Kafka 数据通过 Flume 存储到 HBase

最近更新时间：2023-09-26 15:13:11

场景说明

将 Kafka 中的数据通过 Flume 收集并存储到 Hbase。

开发准备

- 因为任务中需要访问腾讯云消息队列 CKafka，所以需要先创建一个 CKafka 实例，具体见 消息队列 CKafka 。
- 确认您已开通腾讯云，且已创建一个 EMR 集群。创建 EMR 集群时，需要在软件配置界面选择 Flume 组件。

在 EMR 集群使用 Kafka 工具包

首先需要查看 CKafka 的内网 IP 和端口号。登录消息队列 CKafka 的控制台，选择您要使用的 CKafka 实例，在基本消息中查看其内网 IP 为 $kafkaIP，而端口号一般默认为9092。在 topic 管理界面新建一个 topic 为 kafka_test。

配置 flume

1. 创建 flume 的配置文件 `hbase_kafka.properties`

```
vim hbase_kafka.properties
agent.sources = kafka_source
agent.channels = mem_channel
agent.sinks = hbase_sink
# 以下配置 source
agent.sources.kafka_source.type = org.apache.flume.source.kafka.KafkaSource
agent.sources.kafka_source.channels = mem_channel
agent.sources.kafka_source.batchSize = 5000
agent.sources.kafka_source.kafka.bootstrap.servers = $kafkaIP:9092
agent.sources.kafka_source.kafka.topics = kafka_test
# 以下配置 sink
agent.sinks.hbase_sink.type = hbase2
agent.sinks.hbase_sink.channel = mem_channel
agent.sinks.hbase_sink.table = foo_table
agent.sinks.hbase_sink.columnFamily = cf
agent.sinks.hbase_sink.serializer = org.apache.flume.sink.hbase.RegexHBase2EventSerializer
# 以下配置 channel
agent.channels.mem_channel.type = memory
agent.channels.mem_channel.capacity = 100000
agent.channels.mem_channel.transactionCapacity = 10000
```

2. 创建 hbase 表

```
hbase shell
create 'foo_table','cf'
```

3. 运行 flume

```
./bin/flume-ng agent --conf ./conf/ -f hbase_kafka.properties -n agent -Dflume.root.logger=INFO,console
```

4. 运行 kafka producer

```
[hadoop@172 kafka]$ ./bin/kafka-console-producer.sh --broker-list $kafkaIP:9092 --topic kafka_test
hello
hbase_test
```

测试

- 在 kafka 生产者客户端输入信息并回车。

- 观察 hbase 表中是否有相应数据。

参考文档

hbase-sink 配置说明

Kerberos 开发指南

Kerberos 简介

最近更新时间：2023-09-08 16:30:52

当前 EMR-V2.1.0及以后的版本都支持 kerberos 创建安全类型集群，即集群中除 alluxio、zeppelin、kylin 外所有已支持的开源组件以 Kerberos 的安全模式启动，在这种安全环境下，只有经过认证的客户端（Client）才能访问集群的服务（Service，例如 HDFS）。

> **注意**
> impala 服务组件仅支持 EMR-V3.1.0 及以后的版本，以 kerberos 的安全模式启动。

重要概念

全称	简称	作用
key distributed center	KDC	整个安全认证过程的票据生成管理服务，其中包含两个服务：AS 和 TGS。
authentication service	AS	为 client 生成 TGT 的服务。
ticket granting service	TGS	为 client 生成某个服务的 ticket。
account database	AD	存储所有 client 的白名单，只有存在于白名单的 client 才能顺利申请到 TGT。
ticket-granting ticket	TGT	用于获取 ticket 的票据。
client	-	想访问某个 server 的客户端。
server	-	提供某种业务的服务。

其他概念

全称	作用
principal	认证的主体，即“用户名”。
realm	realm 有点像编程语言中的 namespace。在编程语言中，变量名只有在某个 namespace 里才有意义。同样的，一个 principal 只有在某个 realm 下才有意义。所以 realm 可以看成是 principal 的一个“容器”或者“空间”。相对应的，principal 的命名规则是 what_name_you_like@realm。在 kerberos，约定俗成用大写来命名 realm，例如：EXAMPLE.COM。
password	某个用户的密码，对应于 kerberos 中的 master_key。password 可以存在一个 keytab 文件中。所以 kerberos 中需要使用密码的场景都可以用一个 keytab 作为输入。
credential	credential 是“证明某个人确定是他自己/某一种行为的确可以发生”的凭据。在不同的使用场景下，credential 的具体含义也略有不同： • 对于某个 principal 个体而言，他的 credential 就是他的 password。 • 在 kerberos 认证的环节中，credential 就意味着各种各样的 ticket。

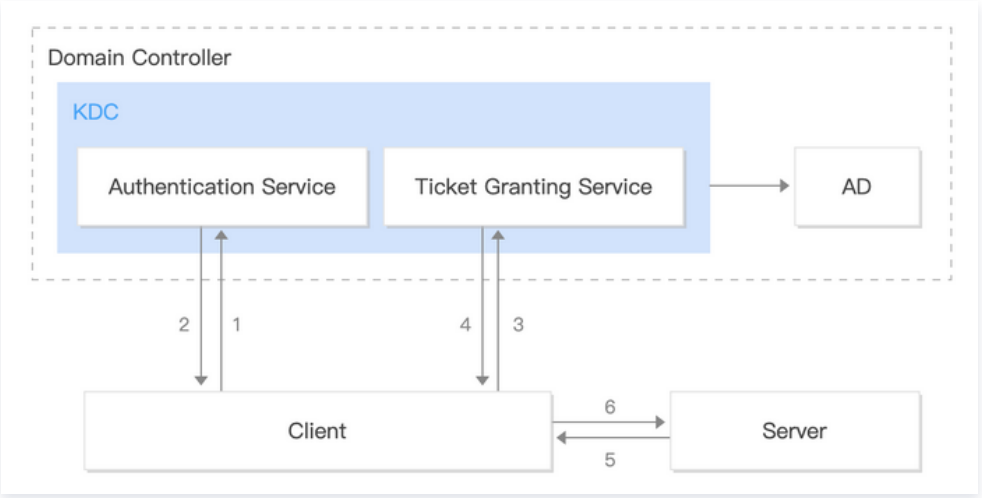
认证流程

client 访问 server 的过程中，想确保 client 和 server 都是可靠的，必然要引入第三方公证平台，因此有了 AS 和 TGS 这两个服务，AS 与 TGS 通常位于同一个服务进程中，对于 mit 的 kerberos 实现，均由 kdc 提供。

认证流程分为以下三步：

1. client 向 kerberos 服务请求，希望获取访问 server 的权限。kerberos 首先判断 client 是否可信赖，通过在 AD 中存储黑名单和白名单来区分 client。成功后，AS 返回 TGT 给 client。
2. client 得到了 TGT 后，继续向 kerberos 请求，希望获取访问 server 的权限。kerberos 通过 client 消息中的 TGT，判断 client 拥有权限，给 client 访问 server 的权限 ticket。

3. client 得到 ticket 后，就可以访问 server 了，但这个 ticket 只是针对这个 server，访问其他 server 需要重新向 TGS 申请。



Kerberos 基础使用

最近更新时间：2024-08-15 14:44:01

EMR 集群使用 mit 的 kerberos 来作为 kdc 服务，使用 kerberos，首先要创建域（realm），再添加相关角色的 principal（包括 server 和 client），新建的 EMR 集群已经完成相关创建工作，并将支持的组件和 hadoop 用户加入数据库中，默认为 hadoop 用户生成了 keytab 文件，路径为 /var/krb5kdc/emr.keytab。

> **说明：**
> EMR-V2.6.0 及 EMR-V3.2.1 以上新版本 Kerberos 已集成 LDAP，并在控制台支持 用户管理 功能，建议使用新版本创建 EMR 安全集群。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考 创建集群 。
- 在创建 EMR 集群时选择 Hadoop 集群类型，选择集成 LDAP 组件的版本，并开启 Kerberos 认证。

> **说明：**
> - 登录 EMR 节点的方式可参考 登录 Linux 实例 。在集群详情页中选择 **集群资源 > 资源管理**，单击对应节点资源 ID 进入云服务器列表，单击右侧**登录**，即可使用 WebShell 登录实例。
> - 登录 Linux 实例用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
> - EMR 暂不支持集群创建完成后开启和关闭 Kerberos 服务。

创建用户

使用 用户管理 功能新建一个用户，此处以 tencent 用户为例，用户创建后 EMR 集群将自动创建此用户 principal 并加入 Kerberos 数据库中。
登录到 Master 节点，在 root 用户下，使用以下命令查看当前 principal：

```
kadmin.local
```

返回以下信息，说明新建用户已经成功加入 Kerberos 数据库中。

```
Authenticating as principal root/admin@EMR-RIELJCFJ with password.
kadmin.local:  list_principals
...
tencent@EMR-RIELJCFJ
```

删除用户

和创建用户相同，删除用户也可以在 用户管理 界面上完成，删除用户后，kerberos 数据库中该用户也会同步删除。

kinit 验证

登录到 Master 节点，使用以下命令进行 kinit 验证：

```
kinit ${username}
```

> **说明：**
> 其中 ${username} 为您在用户管理创建的用户名，输入该命令后，会提示您输入密码，此时输入创建用户时的密码即可完成验证。

kinit 对应的是向 kdc 获取 TGT 的步骤。它会向 /etc/krb5.conf 中指定的 kdc server 发送请求，此时使用 klist 命令：

```
klist
```

返回以下信息说明 TGT 请求成功。

```
Ticket cache: FILE:/tmp/krb5cc_0
```

```
Default principal: tencent@EMR-RIELJCFJ

Valid starting       Expires              Service principal
2024-01-08T14:25:43  2024-01-09T02:25:43  krbtgt/EMR-RIELJCFJ@EMR-RIELJCFJ
    renew until 2024-01-15T14:25:43
```

除了使用用户密码的方式进行 kinit 验证，也可以直接使用本地 keytab 的方式进行 kinit 验证，在 用户管理 界面下载对应用户 keytab 文件，使用本地 keytab 文件进行登录验证：

```
kinit -kt ${keytab_file_path} ${kerberos_user_name}
```

说明：
其中 ${keytab_file_path} 为保存的 keytab 路径，${kerberos_user_name} 为您创建的 kerberos 用户名。

Knox 开发指南
Knox 指引

最近更新时间：2024-06-03 11:01:13

EMR 各版本已支持 Apache Knox，完成以下准备工作后，即可在公网直接访问 Yarn、HDFS 等服务的 Web UI。

准备工作

确认您已开通腾讯云，并且创建了一个 EMR 集群。

开始访问 Knox

使用集群公网 IP 地址访问。建议修改集群拥有公网 IP 的 CVM 安全组规则，限定 TCP:30002 端口的访问 IP 端为您的 IP 端。

1. 通过集群详情查看公网 IP。
2. 在浏览器中访问相应服务的 URL。
   - HDFS UI：https://{集群公网ip}:30002/gateway/emr/hdfs
   - Yarn UI：https://{集群公网ip}:30002/gateway/emr/yarn
   - Hive UI：https://{集群公网ip}:30002/gateway/emr/hive
   - Hbase UI：https://{集群公网ip}:30002/gateway/emr/hbase/webui
   - Storm UI：https://{集群公网ip}:30002/gateway/emr/stormui
   - Ganglia UI：https://{集群公网ip}:30002/gateway/emr/ganglia/
   - Presto UI：https://{集群公网ip}:30002/gateway/emr/presto/
   - Oozie UI：https://{集群公网ip}:30002/gateway/emr/oozie/
   - Livy UI：https://{集群公网ip}:30002/gateway/emr/livy/v1/
   - Ranger UI：https://{集群公网ip}:30002/gateway/emr/ranger/
   - Alluxio UI：https://{集群公网ip}:30002/gateway/emr/alluxio/
   - Impala UI：https://{集群公网ip}:30002/gateway/emr/impalastore/
   - Ganglia UI：https://{集群公网ip}:30002/gateway/emr/ganglia/
   - Spark UI：https://{集群公网ip}:30002/gateway/emr/sparkhistory/
   - Tez UI：https://{集群公网ip}:30002/gateway/emr/tez/
3. 浏览器显示**您的链接不是私密链接**，是因为 Knox 服务使用了自签名证书，请再次确认访问的是自己集群的公网 IP，并且端口为30002。选择**高级 > 继续前往**。
4. 弹出的验证登录框，用户名为 root，默认密码为创建集群时输入的密码。建议您修改密码，可以在该页面中单击**重置原生 UI 密码**进行修改。

Alluxio 开发指南
Alluxio 开发文档

最近更新时间：2024-08-15 14:44:01

背景

Alluxio 通过文件系统接口提供对数据的访问。Alluxio 中的文件提供一次写入语义：它们在被完整写下之后不可变，在完成之前不可读。Alluxio 提供了两种不同的文件系统 API：Alluxio API 和 Hadoop 兼容的 API。Alluxio API 提供了额外的功能，而 Hadoop 兼容的 API 为用户提供了无需修改现有代码使用 Hadoop API 的灵活性。
所有使用 Alluxio Java API 的资源都是通过 AlluxioURI 指定的路径实现。

获取文件系统客户端

要使用 Java 代码获取 Alluxio 文件系统客户端，请使用：

```
FileSystem fs = FileSystem.Factory.get();
```

创建一个文件

所有的元数据操作，以及打开一个文件读取或创建一个文件写入都通过 FileSystem 对象执行。由于 Alluxio 文件一旦写入就不可改变，创建文件的惯用方法是使用 `FileSystem#createFile(AlluxioURI)` ，它返回一个可用于写入文件的流对象。例如：

```
FileSystem fs = FileSystem.Factory.get();
AlluxioURI path = new AlluxioURI("/myFile");
// Create a file and get its output stream
FileOutStream out = fs.createFile(path);
// Write data
out.write(...);
// Close and complete file
out.close();
```

指定操作选项

对于所有的文件系统操作，可以指定一个额外的 options 字段，它允许用户可以指定操作的非默认设置。例如：

```
FileSystem fs = FileSystem.Factory.get();
AlluxioURI path = new AlluxioURI("/myFile");
// Generate options to set a custom blocksize of 128 MB
CreateFileOptions options = CreateFileOptions.defaults().setBlockSize(128 * Constants.MB);
FileOutStream out = fs.createFile(path, options);
```

IO 选项

Alluxio 使用两种不同的存储类型：Alluxio 管理存储和底层存储。Alluxio 管理存储是分配给 Alluxio worker 的内存 SSD 或 HDD。底层存储是由在最下层的存储系统（如 S3、Swift 或 HDFS）管理的资源。用户可以指定通过 ReadType 和 WriteType 与 Alluxio 管理的存储交互。ReadType 指定读取文件时的数据读取行为。WriteType 指定数据编写新文件时写入行为，例如，数据是否应该写入 Alluxio Storage。
下面是 ReadType 的预期行为表。读取总是偏好 Alluxio 存储优先于底层存储。

Read Type	Behavior
CACHE_PROMOTE	• 如果读取的数据在 Worker 上时，该数据被移动到 Worker 的最高层。 • 如果该数据不在本地 Worker 的 Alluxio 存储中，那么就将一个副本添加到本地 Alluxio Worker 中。 • 如果 `alluxio.user.file.cache.partially.read.block` 设置为 true，没有完全读取的数据块也会被全部存到 Alluxio 中。相反，一个数据块只有完全被读取时，才能被缓存。
CACHE	• 如果该数据不在本地 Worker 的 Alluxio 存储中，那么就将一个副本添加到本地 Alluxio Worker 中。

	• 如果 `alluxio.user.file.cache.partially.read.block` 设置为 true，没有完全读取的数据块也会被全部存到 Alluxio 中。相反，一个数据块只有完全被读取时，才能被缓存。
NO_CACHE	仅读取数据，不在 Alluxio 中存储副本。

下面是 WriteType 的预期行为表。

Write Type	Behavior
CACHE_THROUGH	数据被同步地写入到 Alluxio 的 Worker 和底层存储系统。
MUST_CACHE	数据被同步地写入到 Alluxio 的 Worker。但不会被写入到底层存储系统。这是默认写类型。
THROUGH	数据被同步地写入到底层存储系统。但不会被写入到 Alluxio 的 Worker。
ASYNC_THROUGH	数据被同步地写入到 Alluxio 的 Worker，并异步地写入到底层存储系统。处于实验阶段。

位置策略

Alluxio 提供了位置策略来选择要存储文件块到哪一个 worker。使用 Alluxio 的 Java API，用户可以设置策略在 CreateFileOptions 中向 Alluxio 写入文件和在 OpenFileOptions 中读取文件。
用户可以轻松地覆盖默认的策略类配置文件中的属性 `alluxio.user.file.write.location.policy.class` 。内置的策略包括：

- LocalFirstPolicy (alluxio.client.file.policy.LocalFirstPolicy)：首先返回本地节点，如果本地 worker 没有足够的块容量，它从活动 worker 列表中随机选择一名 worker。这是默认的策略。
- MostAvailableFirstPolicy (alluxio.client.file.policy.MostAvailableFirstPolicy)：返回具有最多可用字节的 worker。
- RoundRobinPolicy (alluxio.client.file.policy.RoundRobinPolicy)：以循环方式选择下一个 worker，跳过没有足够容量的 worker。
- SpecificHostPolicy (alluxio.client.file.policy.SpecificHostPolicy)：返回具有指定节点名的 worker。此策略不能设置为默认策略。

Alluxio 支持自定义策略，所以您也可以通过实现接口 `alluxio.client.file.policyFileWriteLocationPolicy` 制定适合自己的策略。

> ⚠ **注意**
> 默认策略必须有一个空的构造函数。并使用 ASYNC_THROUGH 写入类型，所有块的文件必须写入同一个 worker。

Alluxio 允许客户在向本地 worker 写入数据块时选择一个层级偏好。目前这种策略偏好只适用于本地 worker 而不是远程 worker；远程 worker 会写到最高层。默认情况下，数据被写入顶层。用户可以通过 `alluxio.user.file.write.tier.default` 配置项修改默认设置，或通过 `FileSystem#createFile(AlluxioURI)API` 调用覆盖它。
对现有文件或目录的所有操作都要求用户指定 AlluxioURI。使用 AlluxioURI，用户可以使用 FileSystem 中的任何方法来访问资源。AlluxioURI 可用于执行 Alluxio FileSystem 操作，例如修改文件元数据、ttl 或 pin 状态，或者获取输入流来读取文件。例如，要读取一个文件：

```
FileSystem fs = FileSystem.Factory.get();
AlluxioURI path = new AlluxioURI("/myFile");
// Open the file for reading
FileInStream in = fs.openFile(path);
// Read data
in.read(...);
// Close file relinquishing the lock
in.close();
```

Alluxio 常用命令

最近更新时间：2023-07-14 16:17:23

操作	语法	描述
cat	cat "path"	将 Alluxio 中的一个文件内容打印在控制台中。
checkConsistency	checkConsistency "path"	检查 Alluxio 与底层存储系统的元数据一致性。
checksum	checksum "path"	计算一个文件的 md5 校验码。
chgrp	chgrp "group" "path"	修改 Alluxio 中的文件或文件夹的所属组。
chmod	chmod "permission" "path"	修改 Alluxio 中文件或文件夹的访问权限。
chown	chown "owner" "path"	修改 Alluxio 中文件或文件夹的所有者。
copyFromLocal	copyFromLocal "source path""remote path"	"remote path" 将 "source path" 指定的本地文件系统中的文件拷贝到 Alluxio 中 "remote path" 指定的路径，如果 "remote path" 已经存在该命令会失败。
copyToLocal	copyToLocal "remote path" "local path"	将 "remote path" 指定的 Alluxio 中的文件复制到本地文件系统中。
count	count "path"	输出 "path" 中所有名称匹配一个给定前缀的文件及文件夹的总数。
cp	cp "src" "dst"	在 Alluxio 文件系统中复制一个文件或目录。
du	du "path"	输出一个指定的文件或文件夹的大小。
fileInfo	fileInfo "path"	输出指定的文件的数据块信息。
free	free "path"	将 Alluxio 中的文件或文件夹移除，如果该文件或文件夹存在于底层存储中，那么仍然可以在那访问。
getCapacityBytes	getCapacityBytes	获取 Alluxio 文件系统的容量。
getUsedBytes	getUsedBytes	获取 Alluxio 文件系统已使用的字节数。
help	help "cmd"	打印给定命令的帮助信息，如果没有给定命令，打印所有支持的命令的帮助信息。
leader	leader	打印当前 Alluxio leader master 节点名。
load	load "path"	将底层文件系统的文件或者目录加载到 Alluxio 中。
loadMetadata	loadMetadata "path"	将底层文件系统的文件或者目录的元数据加载到 Alluxio 中。
location	location "path"	输出包含某个文件数据的节点。
ls	ls "path"	列出给定路径下的所有直接文件和目录的信息，例如大小。
masterInfo	masterInfo	打印 Alluxio master 容错相关的信息，例如，leader 的地址、所有 master 的地址列表以及配置的 Zookeeper 地址。
mkdir	mkdir "path1" ... "pathn"	在给定路径下创建文件夹，以及需要的父文件夹，多个路径用空格或者 tab 分隔，如果其中的任何一个路径已经存在，该命令失败。
mount	mount "path" "uri"	将底层文件系统的 "uri" 路径挂载到 Alluxio 命名空间中的 "path" 路径下，"path" 路径事先不能存在并由该命令生成。没有任何数据或者元数据从底层文件系统加载。当挂载完成后，对该挂载路径下的操作会同时作用于底层文件系统的挂载点。
mv	mv "source" "destination"	将 "source" 指定的文件或文件夹移动到 "destination" 指定的新路径，如果 "destination" 已经存在该命令失败。
persist	persist "path1" ... "pathn"	将仅存在于 Alluxio 中的文件或文件夹持久化到底层文件系统中。
pin	pin "path"	将给定文件锁定到内容中以防止剔除。如果是目录，递归作用于其子文件以及里面新

		创建的文件。
report	report "path"	向 master 报告一个文件已经丢失。
rm	rm "path"	删除一个文件，如果输入路径是一个目录该命令失败。
setTtl	setTtl "path" "time"	设置一个文件的 TTL 时间，单位毫秒。
stat	stat "path"	显示文件和目录指定路径的信息。
tail	tail "path"	将指定文件的最后1KB内容输出到控制台。
test	test "path"	测试路径的属性，如果属性正确，返回0，否则返回1。
touch	touch "path"	在指定路径创建一个空文件。
unmount	unmount "path"	卸载挂载在 Alluxio 中 "path" 指定路径上的底层文件路径，Alluxio 中该挂载点的所有对象都会被删除，但底层文件系统会将其保留。
unpin	unpin "path"	将一个文件解除锁定从而可以对其剔除，如果是目录则递归作用。
unsetTtl	unsetTtl "path"	删除文件的 ttl 值。

cat

背景

cat 命令将 Alluxio 中的一个文件内容全部打印在控制台中，这在用户确认一个文件的内容是否和预想的一致时非常有用。如果您想将文件拷贝到本地文件系统中，使用 copyToLocal 命令。

操作示例

例如，当测试一个新的计算任务时，cat 命令可以用来快速确认其输出结果。

```
$ alluxio fs cat /output/part-00000
```

checkConsistency

背景

checkConsistency 命令会对比给定路径下 Alluxio 以及底层存储系统的元数据，如果该路径是一个目录，那么其所有子内容都会被对比。该命令返回包含所有不一致的文件和目录的列表，系统管理员决定是否对这些不一致数据进行调整。为了防止 Alluxio 与底层存储系统的元数据不一致，应将您的系统设置为通过 Alluxio 来修改文件和目录，而不是直接访问底层存储系统进行修改。
如果使用了 -r 选项，那么 checkConsistency 命令会去修复不一致的文件或目录，对于只存在底层存储的文件或者文件夹会从 Alluxio 中删除，对于在底层文件系统中，但是，文件内容发生变化的文件，该文件的元数据会重新 load 到 Alluxio。

> **注意**
> 该命令需要请求将要被检查的目录子树的读锁，这意味着在该命令完成之前无法对该目录子树的文件或者目录进行写操作或者更新操作。

操作示例

例如，checkConsistency 命令可以用来周期性地检查命名空间的完整性。
列出不一致的文件或者目录：

```
$ alluxio fs checkConsistency /
```

修复不一致的文件或者目录：

```
$ alluxio fs checkConsistency -r /
```

checksum

背景

checksum 命令输出某个 Alluxio 文件的 md5 值。

操作示例

例如，checksum 可以用来验证 Alluxio 中的文件内容与存储在底层文件系统或者本地文件系统中的文件内容是否匹配。

```
$ alluxio fs checksum /LICENSE
```

chgrp

背景

chgrp 命令可以改变 Alluxio 中的文件或文件夹的所属组，Alluxio 支持 POSIX 标准的文件权限，组在 POSIX 文件权限模型中是一个授权实体，文件所有者或者超级用户可以执行这条命令从而改变一个文件或文件夹的所属组。
加上 -R 选项可以递归的改变文件夹中子文件和子文件夹的所属组。

操作示例

使用 chgrp 命令能够快速修改一个文件的所属组。

```
$ alluxio fs chgrp alluxio-group-new /input/file1
```

chmod

背景

chmod 命令修改 Alluxio 中文件或文件夹的访问权限，目前可支持八进制模式：三位八进制的数字分别对应于文件所有者、所属组以及其他用户的权限。以下是数字与权限的对应表：

Number	Permission	rwx
7	read, write and execute	rwx
6	read and write	rw-
5	read and execute	r-x
4	read only	r--
3	write and execute	-wx
2	write only	-w-
1	execute only	--x
0	none	---

加上 -R 选项可以递归的改变文件夹中子文件和子文件夹的权限。

操作示例

使用 chmod 命令可以快速修改一个文件的权限。

```
$ alluxio fs chmod 755 /input/file1
```

chown

背景

chown 命令用于修改 Alluxio 中文件或文件夹的所有者，出于安全方面的考虑，只有超级用户能够更改一个文件的所有者。
加上 -R 选项可以递归的改变文件夹中子文件和子文件夹的所有者。

使用示例

使用 chown 命令可以快速修改一个文件的所有者。

```
$ alluxio fs chown alluxio-user /input/file1
```

copyFromLocal

背景

copyFromLocal 命令将本地文件系统中的文件拷贝到 Alluxio 中，如果您运行该命令的机器上有 Alluxio worker，那么数据便会存放在这个 worker 上，否则，数据将会随机地复制到一个运行 Alluxio worker 的远程节点上。如果该命令指定的目标是一个文件夹，那么这个文件夹及其所有内容都会被递归复制到 Alluxio 中。

操作示例

使用 copyFromLocal 命令可以快速将数据复制到 alluxio 系统中以便后续处理。

```
$ alluxio fs copyFromLocal /local/data /input
```

copyToLocal

背景

copyToLocal 命令将 Alluxio 中的文件复制到本地文件系统中，如果该命令指定的目标是一个文件夹，那么该文件夹及其所有内容都会被递归地复制。

操作示例

使用 copyToLocal 命令可以快速将输出数据下载下来从而进行后续研究或调试。

```
$ alluxio fs copyToLocal /output/part-00000 part-00000
```

count

背景

count 命令输出 Alluxio 中所有名称匹配一个给定前缀的文件及文件夹的总数，以及它们总的大小，该命令对文件夹中的内容递归处理。当用户对文件有预定义命名习惯时，count 命令很有用。

操作示例

若文件是以它们的创建日期命名，使用 count 命令可以获取任何日期、月份以及年份的所有文件的数目以及它们的总大小。

```
$ alluxio fs count /data/2014
```

cp

背景

cp 命令拷贝 Alluxio 文件系统中的一个文件或者目录，也可以在本地文件系统和 Alluxio 文件系统之间相互拷贝。filescheme 表示本地文件系统，alluxioscheme 或不写 scheme 表示 Alluxio 文件系统。如果使用了 -R 选项，并且源路径是一个目录，cp 将源路径下的整个子树拷贝到目标路径。

操作示例

例如，cp 可以在底层文件系统之间拷贝文件。

```
$ alluxio fs cp /hdfs/file1 /s3/
```

du

背景

du 命令输出一个文件的大小，如果指定的目标为文件夹，该命令输出该文件夹下所有子文件及子文件夹中内容的大小总和。

操作示例

如果 Alluxio 空间被过分使用，使用 du 命令可以检测到哪些文件夹占用了大部分空间。

```
$ alluxio fs du /\\*
```

fileInfo

背景

fileInfo 命令从1.5开始不再支持，请使用 stat 命令。fileInfo 命令将一个文件的主要信息输出到控制台，这主要是为了让用户调试他们的系统。一般来说，在 Web UI 上查看文件信息要容易理解得多。

操作示例

使用 fileInfo 命令能够获取到一个文件的数据块的位置，这在获取计算任务中的数据局部性时非常有用。

```
$ alluxio fs fileInfo /data/2015/logs-1.txt
```

free

背景

free 命令请求 Alluxio master 将一个文件的所有数据块从 Alluxio worker 中剔除，如果命令参数为一个文件夹，那么会递归作用于其子文件和子文件夹。该请求不保证会立即产生效果，因为该文件的数据块可能正在被读取。free 命令在被 master 接收后会立即返回。注意该命令不会删除底层文件系统中的任何数据，而只会影响存储在 Alluxio 中的数据。另外，该操作也不会影响元数据，这意味着如果运行 ls 命令，该文件仍然会被显示。

操作示例

使用 free 命令可以手动管理 Alluxio 的数据缓存。

```
$ alluxio fs free /unused/data
```

getCapacityBytes

背景

getCapacityBytes 命令返回 Alluxio 被配置的最大字节数容量。

操作示例

使用 getCapacityBytes 命令能够确认您的系统是否正确启动。

```
$ alluxio fs getCapacityBytes
```

getUsedBytes

背景

getUsedBytes 命令返回 Alluxio 中以及使用的空间字节数。

操作示例

使用 getUsedBytes 命令能够监控集群健康状态。

```
$ alluxio fs getUsedBytes
```

leader

背景

leader 命令打印当前 Alluxio 的 leader master 节点名。

操作示例

```
$ alluxio fs leader
```

load

背景

load 命令将底层文件系统中的数据载入到 Alluxio 中。如果运行该命令的机器上正在运行一个 Alluxio worker，那么数据将移动到该 worker 上，否则，数据会被随机移动到一个 worker 上。如果该文件已经在 Alluxio 中存在，该命令不进行任何操作。如果该命令的目标是一个文件夹，那么其子文件和子文件夹会被递归载入。

操作示例

使用 load 命令能够获取用于数据分析作用的数据。

```
$ alluxio fs load /data/today
```

loadMetadata

背景

loadMetadata 命令查询本地文件系统中匹配给定路径名的所有文件和文件夹，并在 Alluxio 中创建这些文件的镜像。该命令只创建元数据，例如文件名及文件大小，而不会传输数据。

操作示例

当其他系统将数据输出到底层文件系统中（不经过 Alluxio），而在 Alluxio 上运行的某个应用又需要使用这些输出数据时，就可以使用 loadMetadata 命令。

```
$ alluxio fs loadMetadata /hdfs/data/2015/logs-1.txt
```

location

背景

location 命令返回包含一个给定文件包含的数据块的所有 Alluxio worker 的地址。

操作示例

当使用某个计算框架进行作业时，使用 location 命令可以调试数据局部性。

```
$ alluxio fs location /data/2015/logs-1.txt
```

ls

背景

ls 命令列出一个文件夹下的所有子文件和子文件夹及文件大小、上次修改时间以及文件的内存状态。对一个文件使用 ls 命令仅仅会显示该文件的信息。ls 命令也将任意文件或者目录下的子目录的元数据从底层存储系统加载到 Alluxio 命名空间，如果 Alluxio 还没有这部分元数据的话。ls 命令查询底层文件系统中匹配给定路径的文件或者目录，然后会在 Alluxio 中创建一个该文件的镜像文件。只有元数据，例如文件名和大小，会以这种方式加载而不发生数据传输。
选项：

- -d 选项将目录作为普通文件列出。例如，`ls -d /` 显示根目录的属性。
- -f 选项强制加载目录中的子目录的元数据。默认方式下，只有当目录首次被列出时，才会加载元数据。
- -h 选项以可读方式显示文件大小。
- -p 选项列出所有固定的文件。
- -R 选项可以递归的列出输入路径下的所有子文件和子文件夹，并列出从输入路径开始的所有子树。

操作示例

使用 ls 命令可以浏览文件系统。

```
$ alluxio fs mount /cos/data cosn://data-bucket/
```

验证：

```
$ alluxio fs ls /s3/data/
```

masterInfo

背景

masterInfo 命令打印与 Alluxio master 容错相关的信息，例如 leader 的地址、所有 master 的地址列表以及配置的 Zookeeper 地址。如果 Alluxio 运行在单 master 模式下，masterInfo 命令会打印出该 master 的地址；如果 Alluxio 运行在多 master 容错模式下，masterInfo 命令会打印出当前的 leader 地址、所有 master 的地址列表以及 Zookeeper 的地址。

操作示例

使用 masterInfo 命令可以打印与 Alluxio master 容错相关的信息。

```
$ alluxio fs masterInfo
```

mkdir

背景

mkdir 命令在 Alluxio 中创建一个新的文件夹。该命令可以递归创建不存在的父目录。注意在该文件夹中的某个文件被持久化到底层文件系统之前，该文件夹不会在底层文件系统中被创建。对一个无效的或者已存在的路径使用 mkdir 命令会失败。

操作示例

管理员使用 mkdir 命令可以创建一个基本文件夹结构。

```
$ alluxio fs mkdir /users
$ alluxio fs mkdir /users/Alice
$ alluxio fs mkdir /users/Bob
```

mount

背景

mount 命令将一个底层存储中的路径链接到 Alluxio 路径，并且在 Alluxio 中该路径下创建的文件和文件夹会在对应的底层文件系统路径进行备份。访问统一命名空间获取更多相关信息。
选项：
--readonly 选项在 Alluxio 中设置挂载点为只读。
--option `<key>=<val>` 选项传递一个属性到这个挂载点（如 S3 credential）。

操作示例

使用 mount 命令可以让其他存储系统中的数据在 Alluxio 中也能获取。

```
$ alluxio fs mount /mnt/hdfs hdfs://host1:9000/data/
```

mv

背景

mv 命令将 Alluxio 中的文件或文件夹移动到其他路径。目标路径一定不能事先存在或者是一个目录。如果是一个目录，那么该文件或文件夹会成为该目录的子文件或子文件夹。mv 命令仅仅对元数据进行操作，不会影响该文件的数据块。mv 命令不能在不同底层存储系统的挂载点之间操作。

操作示例

使用 mv 命令可以将过时数据移动到非工作目录。

```
$ alluxio fs mv /data/2014 /data/archives/2014
```

persist

背景

persist 命令将 Alluxio 中的数据持久化到底层文件系统中。该命令是对数据的操作，因而其执行时间取决于该文件的大小。在持久化结束后，该文件即在底层文件系统中有了备份，因而该文件在 Alluxio 中的数据块被剔除甚至丢失的情况下，仍能够访问。

操作示例

在从一系列临时文件中过滤出包含有用数据的文件后，便可以使用 persist 命令对其进行持久化。

```
$ alluxio fs persist /tmp/experimental-logs-2.txt
```

pin

背景

pin 命令对 Alluxio 中的文件或文件夹进行标记。该命令只针对元数据进行操作，不会导致任何数据被加载到 Alluxio 中。如果一个文件在 Alluxio 中被标记了，该文件的任何数据块都不会从 Alluxio worker 中被剔除。如果存在过多的被锁定的文件，Alluxio worker 将会剩余少量存储空间，从而导致无法对其他文件进行缓存。

操作示例

如果管理员对作业运行流程十分清楚，那么可以使用 pin 命令手动提高性能。

```
$ alluxio fs pin /data/today
```

report

背景

report 命令向 Alluxio master 标记一个文件为丢失状态。该命令应当只对使用 Lineage API 创建的文件使用。将一个文件标记为丢失状态将导致 master 调度重计算作业从而重新生成该文件。

操作示例

使用 report 命令可以强制重新计算生成一个文件。

```
alluxio fsadmin report /tmp/lineage-file
```

rm

背景

rm 命令将一个文件从 Alluxio 以及底层文件系统中删除。该命令返回后该文件便立即不可获取，但实际的数据要过一段时间才被真正删除。
加上 -R 选项可以递归的删除文件夹中所有内容后再删除文件夹自身。加上 -U 选项将会在尝试删除持久化目录之前不会检查将要删除的 UFS 内容是否与 Alluxio 一致。

操作示例

使用 rm 命令可以删除掉不再需要的临时文件。

```
$ alluxio fs rm /tmp/unused-file
```

setTtl

背景

setTtl 命令设置一个文件或者文件夹的 ttl 时间，单位为毫秒。如果当前时间大于该文件的创建时间与 ttl 时间之和时，行动参数将指示要执行的操作。delete 操作（默认）将同时删除 Alluxio 和底层文件系统中的文件，而 free 操作将仅仅删除 Alluxio 中的文件。

操作示例

管理员在知道某些文件经过一段时间后便没用时，可使用带有 delete 操作的 setTtl 命令来清理文件；如果仅希望为 Alluxio 释放更多的空间，可使用带有 free 操作的 setTtl 命令来清理 Alluxio 中的文件内容。

```
$ alluxio fs setTtl -action free /data/good-for-one-day 86400000
```

stat

背景

stat 命令将一个文件或者文件夹的主要信息输出到控制台，这主要是为了让用户调试他们的系统。一般来说，在 Web UI 上查看文件信息要容易理解得多。
可以指定 -f 来按指定格式显示信息：

- “%N”：文件名。
- “%z”：文件大小（bytes）。
- “%u”：文件拥有者。
- “%g”：拥有者所在组名。
- “%y” or “%Y”：编辑时间，`%y shows 'yyyy-MM-dd HH:mm:ss' (the UTC date), %Y` 为自从 January 1, 1970 UTC 以来的毫秒数。
- “%b”：为文件分配的数据块数。

操作示例

例如，使用 stat 命令能够获取到一个文件的数据块的位置，这在获取计算任务中的数据局部性时非常有用。

```
$ alluxio fs stat /data/2015/logs-1.txt
$ alluxio fs stat /data/2015
$ alluxio fs stat -f %z /data/2015/logs-1.txt
```

tail

背景

tail 命令将一个文件的最后1kb内容输出到控制台。

操作示例

使用 tail 命令可以确认一个作业的输出是否符合格式或者包含期望的值。

```
$ alluxio fs tail /output/part-00000
```

test

背景

test 命令测试路径的属性。
选项：
-d 选项测试路径是否是目录。
-e 选项测试路径是否存在。
-f 选项测试路径是否是文件。
-s 选项测试路径是否为空。
-z 选项测试文件长度是否为0。

操作示例

```
$ alluxio fs test -d /someDir
```

touch

背景

touch 命令创建一个空文件。由该命令创建的文件不能被覆写，大多数情况是用作标记。

操作示例

使用 touch 命令可以创建一个空文件用于标记一个文件夹的分析任务完成了。

```
$ alluxio fs touch /data/yesterday/_DONE_
```

unmount

背景

unmount 将一个 Alluxio 路径和一个底层文件系统中的目录的链接断开。该挂载点的所有元数据和文件数据都会被删除，但底层文件系统会将其保留。访问 Unified Namespace 获取更多信息。

操作示例

当不再需要一个底层存储系统中的数据时，使用 unmont 命令可以移除该底层存储系统。

```
$ alluxio fs unmount /s3/data
```

unpin

背景

unpin 命令将 Alluxio 中的文件或文件夹解除标记。该命令仅作用于元数据，不会剔除或者删除任何数据块。一旦文件被解除锁定，Alluxio worker 可以剔除该文件的数据块。

操作示例

当管理员知道数据访问模式发生改变时，可以使用 unpin 命令。

```
$ alluxio fs unpin /data/yesterday/join-table
```

unsetTtl

背景

unsetTtl 命令删除 Alluxio 中一个文件的 TTL。该命令仅作用于元数据，不会剔除或者删除 Alluxio 中的数据块。该文件的 TTL 值可以由 setTtl 命令重新设定。

操作示例

在一些特殊情况下，当一个原本自动管理的文件需要手动管理时，可使用 unsetTtl 命令。

```
$ alluxio fs unsetTtl /data/yesterday/data-not-yet-analyzed
```

挂载文件系统到 Alluxio 统一文件系统

最近更新时间：2023-12-18 16:13:21

背景

Alluxio 提供了统一命名空间机制，允许将其他文件系统挂载到 Alluxio 的文件系统中。该机制允许上层应用使用统一的命名空间，访问分散在不同系统中的数据。

挂载 COS

示例：将 COS 中某个 bucket 挂载到 alluxio 目录中。

```
alluxio fs mount --option fs.cos.access.key=<COS_SECRET_ID> \
   --option fs.cos.secret.key=<COS_SECRET_KEY> \
   --option fs.cos.region=<COS_REGION> \
   --option fs.cos.app.id=<COS_APP_ID> \
   /cos cos://<COS_BUCKET>/
```

--option 中 COS 的配置信息如下，其中 fs.cos.secret.key 和 fs.cos.secret.id 可以通过 API 密钥管理 获取。

配置名称	解释
fs.cos.access.key	cos secret id
fs.cos.secret.key	cos secret key
fs.cos.region	cos region 名称，例如 `ap-beijing`
fs.cos.app.id	用户 AppID
COS_BUCKET	COS BUCKET 名称。只要名称，不要带 AppID 后缀

该命令，将 COS 的目录（通过 `cos://bucket/xxx` 指定）挂载到 alluxio 的 `/cos` 目录下。

挂载 HDFS

示例：将 HDFS 某目录挂载到 alluxio 目录中。

```
alluxio fs mount /hdfs hdfs://data
```

该命令将 HDFS 的 `/data` 目录挂载到 alluxio 的 `/hdfs` 子目录下。
挂载成功后，通过 `alluxio fs ls` 命令，查看挂载内容。

挂载 CHDFS

示例：将 CHDFS 通过 mount 挂载到 Alluxio

> **说明**
> 只有 EMR2.5.0 + alluxio2.3.0+ 以上才支持。

```
alluxio fs mount   \
--option alluxio.underfs.hdfs.configuration=/usr/local/service/hadoop/etc/hadoop/core-site.xml  \
/chdfs ofs://f4modr7kmvw-wMqw.chdfs.ap-chongqing.myqcloud.com
```

在腾讯云中使用 Alluxio 文档

最近更新时间：2024-06-07 15:41:41

概述

在腾讯云 EMR 上提供了开箱可用的 Alluxio 服务，以帮助腾讯云客户可以快速实现分布式内存级缓存加速、简化数据管理等。同时还可以通过腾讯云 EMR 控制台或 API 接口，使用配置下发功能，快速配置多层级缓存和元数据管理等，获取一站式监控告警等功能。

> **注意：**
> 开启 Kerberos 的 EMR 集群暂不支持购买时或后安装部署 Alluxio 组件。

准备

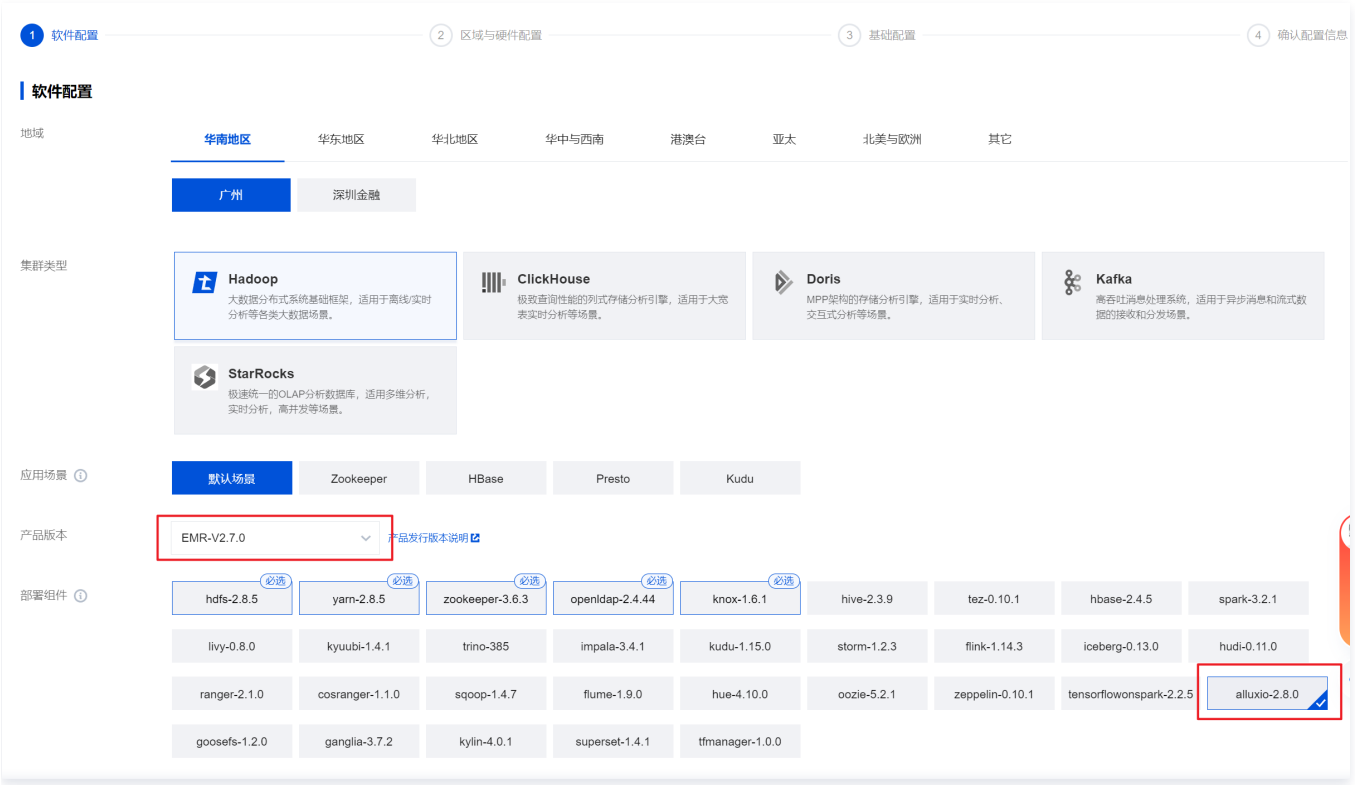
腾讯云 EMR 的 Hadoop 标准版本2.1.0版本及以上。
有关 EMR 支持具体的 Alluxio 的版本，可参考 组件版本 。

创建基于 Alluxio 的 EMR 集群

本节主要说明如何在腾讯云 EMR 上创建开箱即用的 Alluxio 集群。EMR 创建集群提供了购买页创建和 API 创建两种方式。

购买页创建集群

您需要登录腾讯云 EMR 购买页，在购买页选择支持的 Alluxio 发布版本，并且在可选组件列表中勾选 Alluxio 组件。



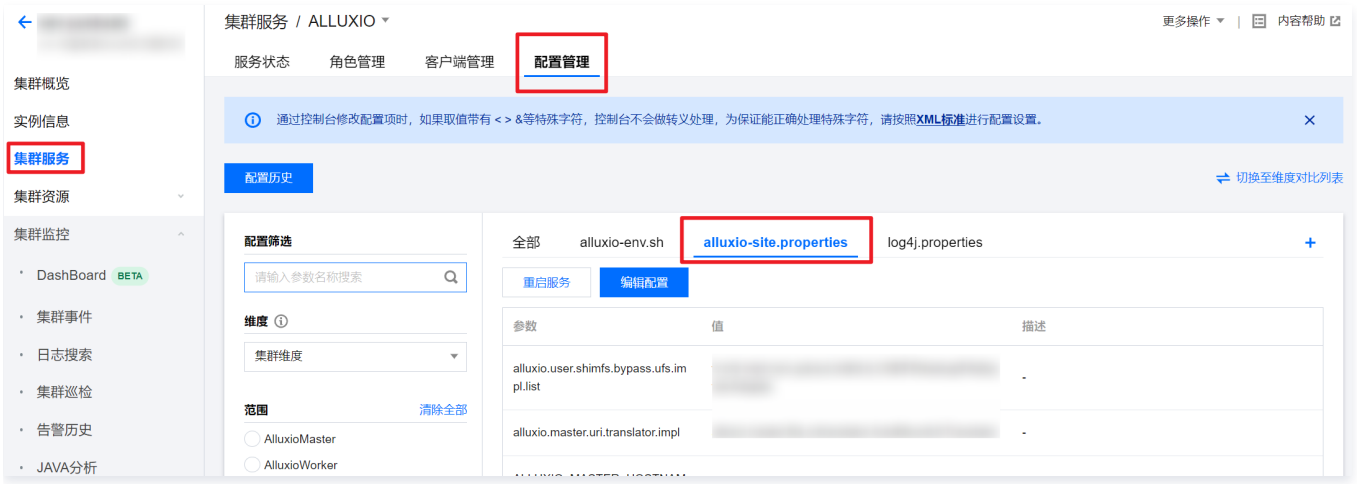
其他的选项可根据业务具体场景，进行个性化配置，创建过程中的具体选项可参考 创建 EMR 集群 。

API 创建集群

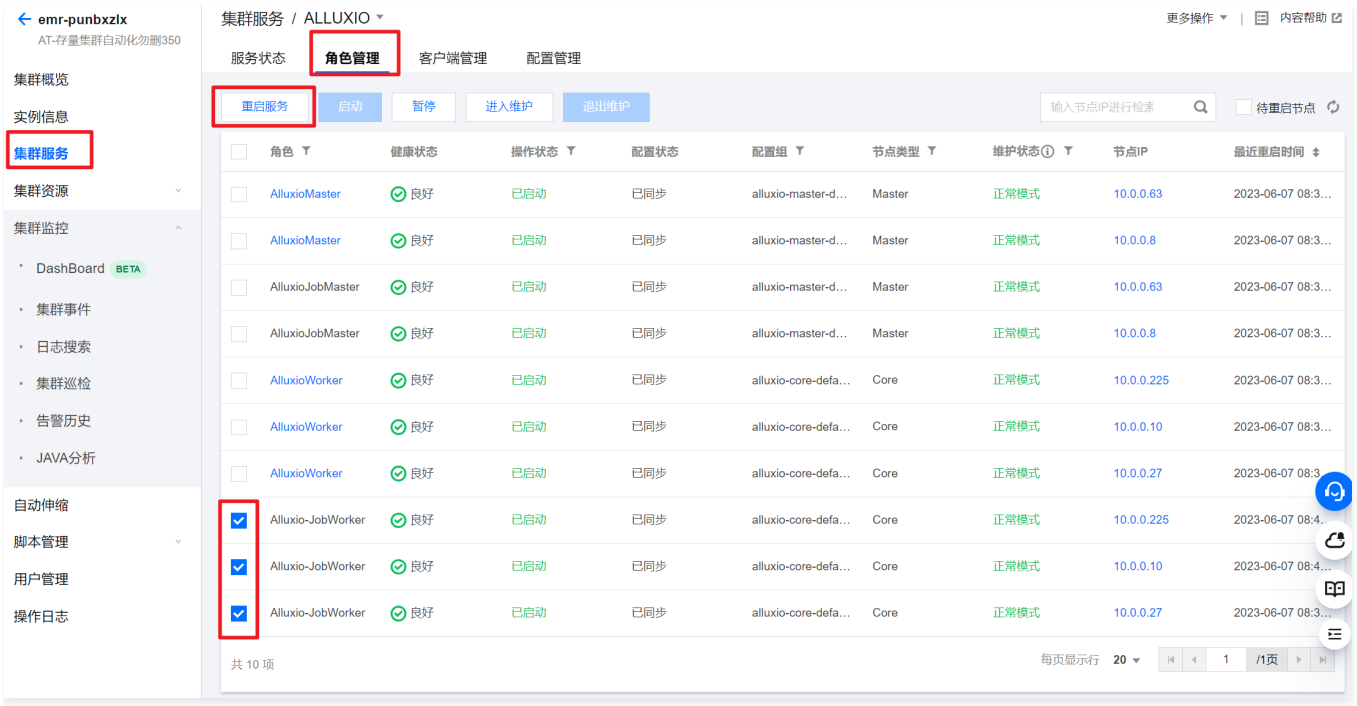
腾讯云 EMR 还提供了 API 方式构建基于 Alluxio 的大数据集群。具体可参考 查询硬件节点信息 。

基础配置

创建一个带 Alluxio 组件的腾讯云 EMR 集群，默认会把 HDFS 挂载到 Alluxio 上，并使用内存作为单层 level0 存储。如果有需要更改更符合业务特性的多级存储，或者其他对应优化项，可以使用配置下发功能来完成相关配置。



在配置下发后，部分配置需要重启 Alluxio 服务才能生效。



更多配置下发和重启策略细节，可参考 配置下发 和 重启组件 。

基于 Alluxio 加速计算存储分离

腾讯云 EMR 基于腾讯云对象存储（COS）提供了计算存储分离能力，默认直接访问对象存储中的数据时，应用程序没有节点级数据本地性或跨应用程序缓存。使用 Alluxio 加速将缓解这些问题。
在腾讯云 EMR 集群上默认已部署使用 COS 作为 UFS 的依赖 jar 包，只需授予访问 COS 的权限，并把 COS mount 到 Alluxio 上即可使用。

授权

若当前集群未开启对象存储，可在 访问管理控制台 > 角色 中进行授权，授权后 EMR 中节点可以通过临时密钥访问 COS 中数据。

Mount

登录到 EMR 任意一台机器，挂载 COS 到 Alluxio。

```
alluxio fs mount <alluxio-path> <source-path>
```

更多在腾讯云 EMR 中使用 Alluxio 开发细节，可查阅 Alluxio 开发文档 。

Alluxio 支持 COS 透明 URI

最近更新时间：2024-08-23 14:52:21

Alluxio 用户通常具有通过现有应用程序访问其底层存储系统（Under-FileSystem），将 Alluxio 添加到现有的生态系统中的需求，但现有应用程序必须更改的是需要在应用程序使用 Alluxio 的 URI。透明 URI 功能允许用户访问现有存储系统，且无需在应用程序级别更改 URI。

支持版本与配置 URI

1. 服务组件支持版本：Alluxio2.8.0版本。
2. 产品版本：Hadoop3.x 标准版本 EMR-V3.4.0 版本。
3. 配置支持透明 URI。使用 Alluxio 透明 URI，需要配置新的 Hadoop 兼容文件系统客户端实现。只要将客户端配置为接收外部 URI，此新的 ShimFileSystem 就会替换现有的 FileSystem。Hadoop 兼容的计算框架--Hadoop FileSystem 接口定义了从 FileSystem 方案到 FileSystem 实现的映射。为了配置 ShimFileSystem，请确保 core-site.xml 中配置了以下配置项：

配置项	配置项值
fs.cosn.impl	alluxio.hadoop.ShimFileSystem

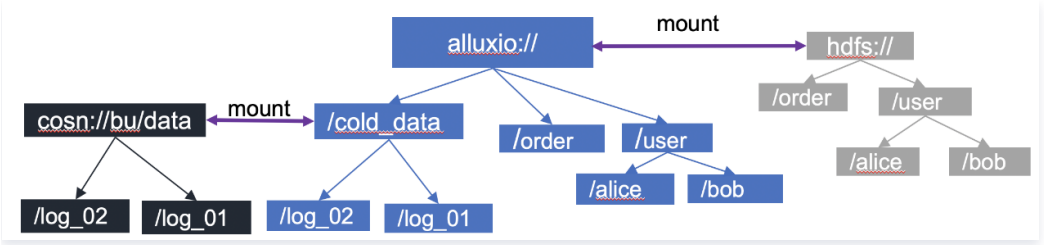
4. Alluxio 在兼容透明 URL Schema 时需要对其进行转换兼容，请确保 alluxio-site.properties 中配置了以下配置项：

配置项	配置项值
alluxio.master.uri.translator.impl	alluxio.master.file.uritranslator.AutoMountUriTranslator
alluxio.user.shimfs.bypass.ufs.impl.list	fs.cosn.impl:org.apache.hadoop.fs.cosnative.NativeCosFileSystem

> **说明**
> - 对 alluxio-site.properties 配置进行变更后需重启 Alluxio 服务。
> - 一旦配置了 ShimFileSystem，master 将需要将外部存储系统本地的 URI 路由到 Alluxio 名称空间。这要求 cosn 已 mount 在 Alluxio 名称空间中。
> - 关闭透明 URI 功能：只需回滚 core-site.xml中fs.cosn.impl 配置项。

mount

mount 命令可以说是 Alluxio 最有特色的命令之一。它类似于 Linux 里的 mount 命令---Linux 用户可以通过 Linux mount 把硬盘，SSD 等存储设备加载到这台 Linux 系统的本地文件系统中。而在 Alluxio 系统当中，mount 的概念进一步被扩展到了分布式系统一层：用户可以通过 Alluxio mount 把一个或多个其他的存储系统/云存储服务（例如 HDFS、COS 等），挂载到 Alluxio 这个分布式文件系统当中去。从而运行在 Alluxio 上的分布式应用，例如 Spark、Presto 或者 MapReduce 等，不需要去适配甚至了解具体的数据访问协议和路径，而只需要知道数据对应在 Alluxio 文件系统的路径就已足够，从而极大的方便了应用的开发和维护。



EMR-Alluxio 默认使用 hdfs 作为根目录挂载点

在 EMR-Alluxio2.5.1+后，Alluxio 的 UFS 开始支持 COSN 协议，COS UFS 存在读写性能较差以及不稳定的问题，为了解决此类问题，社区贡献了 COSN UFS 底层文件系统。COS 和 COSN UFS 都是用于访问腾讯云对象存储，COSN 相对于 COS 做了深度优化，其读写性能较COS 成倍提升，同时带来了更好的稳定性，所以强烈推荐使用 COSN。COS UFS 将于 EMR-Alluxio2.6.0 版本后停止维护。

Mount COSN 示例：

```
alluxio fs mount --option fs.cosn.userinfo.secretId=xx \
      --option fs.cosn.userinfo.secretKey=xx \
      --option fs.cosn.bucket.region=ap-xx \
      --option fs.cosn.impl=org.apache.hadoop.fs.cosnative.NativeCosFileSystem \
```

```
    --option fs.AbstractFileSystem.cosn.impl=org.apache.hadoop.fs.CosN \
    --option fs.cosn.userinfo.appid=xx \
    /cosn cosn://COS_BUCKET/path
```

其中，--options 中配置 COS 的配置。

配置项名称	解释
fs.cosn.userinfo.secretId	cos secret id
fs.cosn.userinfo.secretKey	cos secret key
fs.cosn.impl	固定值：`org.apache.hadoop.fs.CosFileSystem`
fs.AbstractFileSystem.cosn.impl	固定值：`org.apache.hadoop.fs.CosN`
fs.cosn.bucket.region cos region	名称，例如 ap-beijing
fs.cosn.userinfo.appid	用户主账号 AppID
COS_BUCKET COS BUCKET	名称。只要名称，不要带 AppID 后缀

Alluxio 支持 COS（元数据加速桶）透明 URI

最近更新时间：2023-12-18 16:13:21

元数据加速桶与普通 COS 存储桶不同的是，其在文件系统上提供了高性能的能力。具体可参见 元数据加速 文档。
Alluxio 支持元数据加速桶在透明 URL 使用和配置上有以下异同点：

使用层面

普通存储桶+Alluxio 透明 URL

```
hdfs dfs -ls cosn://COS_BUCKET/PATH/
```

元数据加速桶+Alluxio 透明 URL

```
hdfs dfs -ls ofs://MOUNT_POINT-APPID/
```

其中，MOUNT_POINT-APPID 例图如下：

配置层面

相同点

使用 Alluxio 对元数据加速桶加速的场景，core-site.xml 及 alluxio-site.properties 的配置与 Alluxio 支持 COS 透明 URI 中的**支持版本与配置 URI** 一致。

不同点

Mount

```
alluxio fs mount --option fs.AbstractFileSystem.ofs.impl=com.qcloud.chdfs.fs.CHDFSDelegateFSAdapter \
    --option fs.ofs.user.appid=xx \
    --option fs.ofs.tmp.cache.dir=xx \
    --option fs.ofs.impl=com.qcloud.chdfs.fs.CHDFSHadoopFileSystemAdapter \
    --option fs.ofs.bucket.region=xx
    /meta-accelarate ofs://MOUNT_POINT-APPID/
```

其中，--options 中配置 CHDFS 的配置。

配置项名称	解释
fs.AbstractFileSystem.ofs.impl	固定值：`com.qcloud.chdfs.fs.CHDFSDelegateFSAdapter`

fs.ofs.user.appid	用户主账号 AppID
fs.ofs.tmp.cache.dir	存放 chdfs 插件目录
fs.ofs.impl	固定值：`com.qcloud.chdfs.fs.CHDFSHadoopFileSystemAdapter`
fs.ofs.bucket.region	名称，例如 ap-beijing
MOUNT_POINT-APPID	携带 AppID 的挂载点名称

Alluxio 支持鉴权

最近更新时间：2022-06-24 14:39:19

Alluxio 用户在对已有的统一命名空间访问 COS、HDFS、CHDFS 上的数据，或者用户使用透明 URL 来访问 Alluxio 中的缓存数据时，会出现无鉴权情况，也就是说任何用户只要拿着对应 URI 就能获取到数据。云上 Alluxio 对此类场景结合 Ranger 和 CosRanger 完善了鉴权场景。

> **说明**
> 为了设配鉴权特性，请确保集群有以下组件集成：
> - 如果 Alluxio 中只挂载了 HDFS，那么需要集成 Ranger 组件。
> - 如果 Alluxio 中挂载了 COS、CHDFS，那么需要集成 CosRanger 组件。

支持版本

- 服务组件支持版本：Alluxio2.8.0版本。
- 产品版本：Hadoop3.x 标准版本 EMR-V3.4.0版本。

配置鉴权

前置配置

```
#新增hive组件ranger-hive-security.xml配置项
ranger.plugin.hive.urlauth.filesystem.schemes==hdfs:,file:,wasb:,adl:,alluxio:

#新增presto组件hive.properties配置项
hive.hdfs.authentication.type=NONE
hive.metastore.authentication.type=NONE
hive.hdfs.impersonation.enabled=true
hive.metastore.thrift.impersonation.enabled=true
```

> **说明**
> 以上前置配置需客户根据集群现有组件来配置。

HDFS 鉴权

软链接 ranger 相关配置文件：

```
[hadoop@172 conf]$ pwd
/usr/local/service/alluxio/conf
[hadoop@172 conf]$ ln -s /usr/local/service/hadoop/etc/hadoop/ranger-hdfs-audit.xml
ranger-hdfs-audit.xml
[hadoop@172 conf]$ ln -s /usr/local/service/hadoop/etc/hadoop/ranger-hdfs-security.xml ranger-hdfs-
security.xml
```

alluxio-site.properties 配置

建议使用 EMR 控制台进行集群维度配置下发。

```
# 鉴权开关(默认false)
alluxio.security.authorization.plugins.enabled=true
alluxio.security.authorization.plugin.name=ranger
alluxio.security.authorization.plugin.paths=/usr/local/service/alluxio/conf
alluxio.underfs.security.authorization.plugin.name=ranger
alluxio.underfs.security.authorization.plugin.paths=/usr/local/service/alluxio/conf
alluxio.master.security.impersonation.hadoop.users=*
alluxio.security.login.impersonation.username=_HDFS_USER_
```

> **说明**

下发完成后需重启 Alluxio 服务。

COS 及 CHDFS 鉴权

```
#新增core-site.xml配置项
fs.ofs.ranger.enable.flag=true
```

alluxio-site.properties 配置

建议使用 EMR 控制台进行集群维度配置下发。

```
# 鉴权开关(默认false)
# 鉴权开关(默认false)
alluxio.security.authorization.plugins.enabled=true
alluxio.security.authorization.plugin.name=ranger
alluxio.security.authorization.plugin.paths=/usr/local/service/alluxio/conf
alluxio.underfs.security.authorization.plugin.name=ranger
alluxio.underfs.security.authorization.plugin.paths=/usr/local/service/alluxio/conf
alluxio.cos.qcloud.object.storage.ranger.service.config.dir=/usr/local/service/cosranger/conf
alluxio.master.security.impersonation.hadoop.users=*
alluxio.security.login.impersonation.username=_HDFS_USER_
# 重试次数默认为5次
alluxio.cos.qcloud.object.storage.permission.check.max.retry=5
```

说明

下发完成后需重启 Alluxio 服务。

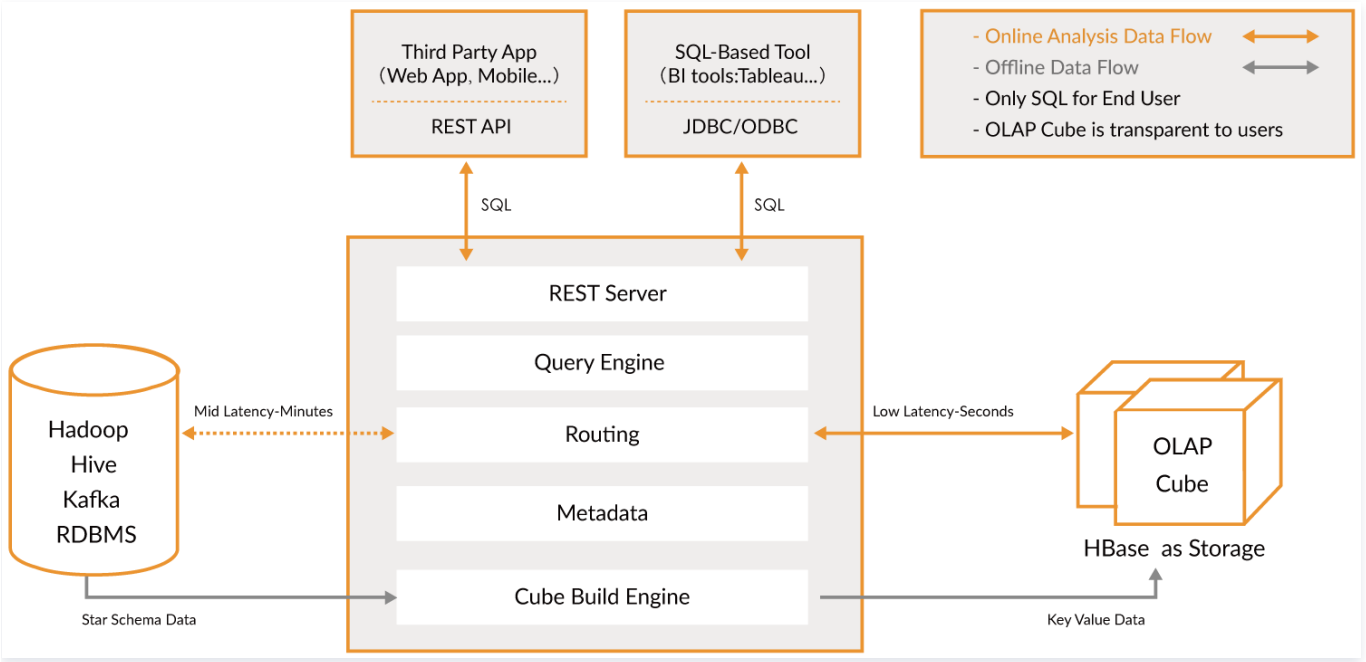
Kylin 开发指南
Kylin 简介

最近更新时间：2024-10-11 15:17:11

Apache Kylin™是一个开源的、分布式的分析型数据仓库，提供 Hadoop/Spark 之上的 SQL 查询接口及多维分析（OLAP）能力以支持超大规模数据，最初由 eBay 开发并贡献至开源社区。它能在亚秒内查询巨大的表。

Kylin 框架介绍

Kylin 能提供低延迟（sub-second latency）的秘诀就是预计算，即针对一个星型拓扑结构的数据立方体，预计算多个维度组合的度量，然后将结果保存在 hbase 中，对外提供 JDBC、ODBC、Rest API 的查询接口，即可实现实时查询。



Kylin 核心概念

- **表（table）：**表定义在 hive 中，是数据立方体（Data cube）的数据源，在 build cube 之前，必须同步在 kylin 中。
- **模型（model）：**模型描述了一个星型模式的数据结构，它定义了一个事实表（Fact Table）和多个查找表（Lookup Table）的连接和过滤关系。
- **Cube 描述：**描述一个 Cube 实例的定义和配置选项，包括使用了哪个数据模型、包含哪些维度和度量、如何将数据进行分区、如何处理自动合并等。
- **Cube 实例：**通过 Cube 描述 Build 得到，包含一个或者多个 Cube Segment。
- **分区(Partition)**：用户可以在 Cube 描述中使用一个 DATA/STRING 的列作为分区的列，从而将一个 Cube 按照日期分割成多个 segment。
- **立方体段（cube segment）：**它是立方体构建（build）后的数据载体，一个 segment 映射 hbase 中的一张表，立方体实例构建（build）后，会产生一个新的 segment，一旦某个已经构建的立方体的原始数据发生变化，只需刷新（fresh）变化的时间段所关联的 segment 即可。
- **聚合组：**每一个聚合组是一个维度的子集，在内部通过组合构建 cuboid。
- **作业（job）：**对立方体实例发出构建（build）请求后，会产生一个作业。该作业记录了立方体实例 build 时的每一步任务信息。作业的状态信息反映构建立方体实例的结果信息。例如，作业执行的状态信息为 RUNNING 时，表明立方体实例正在被构建；作业状态信息为 FINISHED，表明立方体实例构建成功；作业状态信息为 ERROR，表明立方体实例构建失败。
- **DIMENSION & MEASURE 种类**
- Mandatory：强制维度，所有 cuboid 必须包含的维度。
- Hierarchy：层次关系维度，维度之间具有层次关系性，只需要保留一定层次关系的 cuboid 即可。
- Derived：衍生维度，在 lookup 表中，有一些维度可以通过它的主键衍生得到，所以这些维度将不参加 cuboid 的构建。
- Count Distinct(HyperLogLog)：直接进行 count distinct 是很难去计算的，一个近似的算法 HyperLogLog 可以保持错误率在一个很低的范围内。
- Count Distinct(Precise)：将基于 RoaringBitMap 进行计算，目前只支持 int 和 BigInt。
- **Cube Action 种类**
- BUILD：给定一个分区列指定的时间间隔，对 Cube 进行 Build，创建一个新的 cube Segment。
- REFRESH：这个操作，将在一些分期周期内对 cube Segment 进行重新 build。

- MERGE：这个操作将合并多个 cube segments。这个操作可以在构建 cube 时，设置为自动完成。
- PURGE：清理一个 Cube 实例下的 segment，但是不会删除 HBase 表中的 Tables。
- **Job 状态**
- NEW：表示一个 job 已经被创建。
- PENDING：表示一个 job 已经被 job Scheduler 提交，等待执行资源。
- RUNNING：表示一个 job 正在运行。
- FINISHED：表示一个 job 成功完成。
- ERROR：表示一个 job 因为错误退出。
- DISCARDED：表示一个 job 被用户取消。
- **Job 执行**
- RESUME：这个操作将从失败的 Job 的最后一个成功点继续执行该 Job。
- DISCARD：无论工作的状态，用户可以结束它和释放资源。

更多使用方式请参考 官方文档 。

Livy 开发指南
Livy 简介

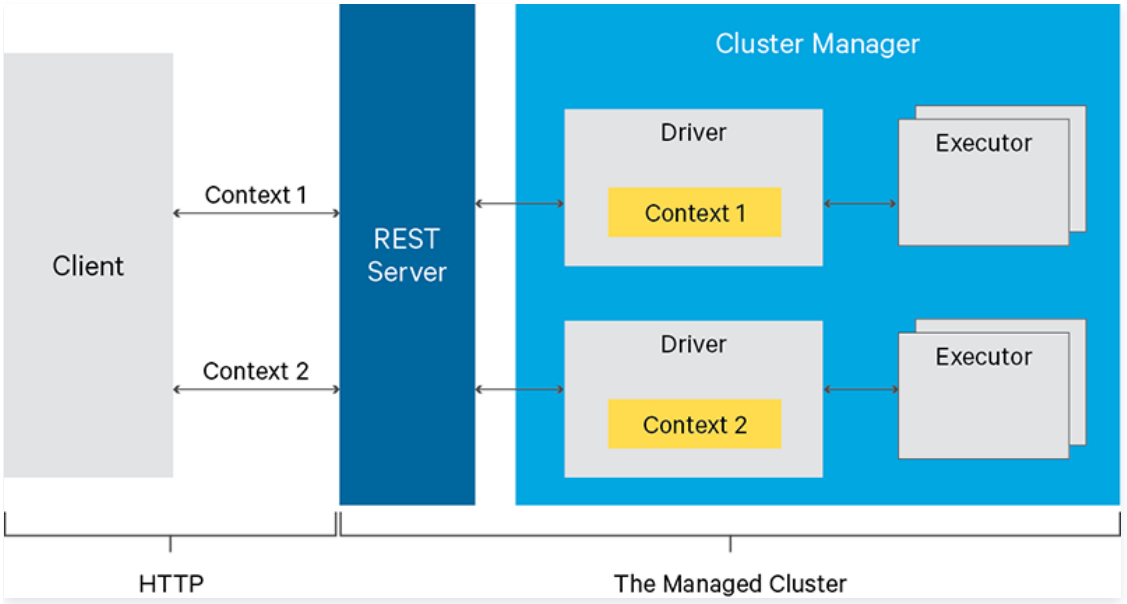
最近更新时间：2023-12-18 16:13:22

Apache Livy 是一个可以通过 REST 接口与 Spark 集群进行交互的服务，它可以提交 Spark 作业或者 Spark 代码片段，同步或者异步地进行结果检索以及 Spark Context 上下文管理，Apache Livy 简化 Spark 和应用程序服务器之间的交互，从而使 Spark 能够用于交互式 Web/移动应用程序。

Livy 特性

Livy 还支持如下功能：

- 由多个客户端长时间运行可用于多个 Spark 作业的 Spark 上下文。
- 跨多个作业和客户端共享缓存的 RDD 或数据帧。
- 可以同时管理多个 Spark 上下文，并且 Spark 上下文运行在群集（YARN/Mesos）而不是 Livy 服务器，以实现良好的容错性和并发性。
- 作业可以作为预编译的 jar，代码片段或通过 java/scala 客户端 API 提交。
- 通过安全的认证通信确保安全。



使用 Livy

1. 访问 `https://IP:8998/ui` 可以进入 Livy 的 UI 页面（**IP 为外网 IP，请自行为安装有 Livy 的机器申请外网 IP，并编辑设置安全组策略来开通对应的端口以进行访问**）。
2. 创建一个交互式会话。（下述示例代码**以 Linux/Mac OS 为例**）

```
curl -X POST --data '{"kind":"spark"}' -H "Content-Type:application/json" IP:8998/sessions
```

3. 查看 Livy 上存活的 sessions。

```
curl IP:8998/sessions
```

4. 执行代码片段，简单的加法操作（这里相当于指定的 session 0，如果有多个 session，也可以指定其他 session）。

```
curl -X POST IP:8998/sessions/0/statements -H "Content-Type:application/json" -d '{"code":"1+1"}'
```

5. 计算圆周率（执行 jar 包）。
   步骤1：上传 jar 包到 hdfs，如上传至/usr/local/spark-examples_2.11-2.4.3.jar。
   步骤2：执行命令：

```
curl -H "Content-Type: application/json" -X POST -d
'{ "file":"/usr/local/spark-examples_2.11-2.4.3.jar",
"className":"org.apache.spark.examples.SparkPi" }' IP:8998/batches
```

6. 查询代码片段执行是否成功，也可以直接在 UI 页面 `https://IP:8998/ui/session/0` 查看。

```
curl IP:8998/sessions/0/statements/0
```

7. 删除 session。

```
curl -X DELETE IP:8998/sessions/0
```

注意事项

开放的配置文件

目前开放的配置文件包括 `livy.conf` 和 `livy-env.sh`，这两个配置文件均可以通过配置下发的方式来修改配置。具体开放了哪几个配置文件，请以 EMR 控制台实际为准。

Livy 端口修改方法

目前的默认端口是8998，用户可以自行修改。修改配置文件 `livy.conf` 的 `livy.server.port` 属性即可。
若集群装有 Hue，由于 Hue 与 Livy 之间涉及联通性，因此 Hue 对应的配置端口也需要修改。其对应的配置文件为 `pseudo-distributed.ini`，路径为 `/usr/local/service/hue/desktop/conf`，对应的配置项为 `spark_livy_server_port=8998`。修改后要重启对应的服务。
如非必要，建议不要修改 Livy 的端口，如涉及安全要求，可以通过安全组的方式来进行控制。修改后可能会导致一些其他的潜在问题。

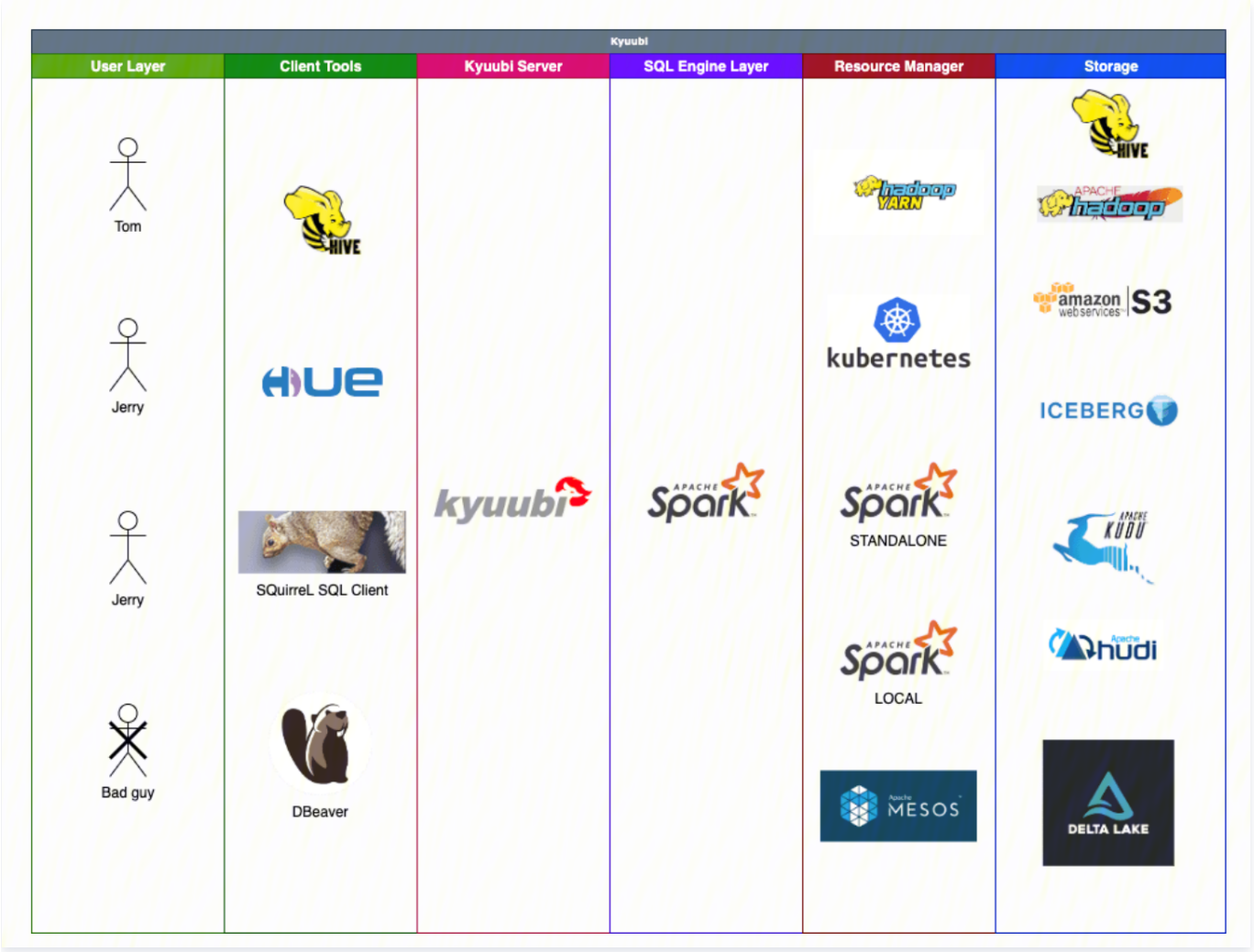
Livy 部署方式

目前 Livy 默认会在所有 master 节点上部署，用户也可以通过扩容 router 的方式在 router 上进行部署。

Kyuubi 开发指南
Kyuubi 简介

最近更新时间：2022-04-02 14:32:57

Apache Kyuubi（Incubating）是一个 Thrift JDBC/ODBC 服务，目前对接了 Apache Spark 计算框架（正在对接Apache Flink计算框架以及Trino），支持多租户和分布式等特性，可以满足企业内诸如 ETL、BI 报表等多种大数据场景的应用。



使用场景

- 替换 HiveServer2，轻松获得 10~100 倍性能提升。
  - Kyuubi 高度兼容 HiveServer2 接口及行为，支持无缝迁移。
  - Kyuubi 分层架构，消除客户端兼容性问题，支持无感升级。
  - Kyuubi 支持 Spark SQL 全链路优化及再增强，性能卓著。
  - 高可用、多租户、细粒度权限认证各种企业级特性都有。
- 构建 Serverless Spark 平台。
  - Serverless Spark 目标绝对不是让用户调用 Spark 的 API、继续写 Spark 作业。
  - 通过 Kyuubi 预置的 Engine 模块，用户无需理解 Spark 逻辑，入门门槛极低。
  - 用户只需通过 JDBC 及 SQL 操作数据专注自身业务开发即可，资源弹性伸缩，0运维。
  - 支持资源管理器（Kubernetes, YARN 等），Engine 生命周期，Spark 动态资源分配3级不同粒度全方位的资源弹性策略。
  - 支持 YARN/Kubernetes 多种资源管理器同时调度，保障历史作业安全迁移上云。
  - Spark 自适应查询引擎（AQE）及 Kyuubi AQE plus，提供澎湃动力。
- 构建统一数据湖探索分析管理平台（kyuubi-1.5以上版本）。

- 支持 Spark 所有官方数据源及第三方数据源。
- 支持 Spark DSv2 元数据管理，直观进行数据湖构建及管理。
- 支持 Apache Iceberg/Hudi, DeltaLake 等所有主流数据湖框架。
- 一个接口一个引擎一份数据，提供统一的分析查询、数据摄取、数据湖管理平台。
- 批流一体，支持流式作业（Upcoming）。

Kyuubi 实践教程

最近更新时间：2023-07-11 21:35:54

Beeline 连接 kyuubi

登录 EMR 集群的 Master 节点，切换到 Hadoop 用户并且进入 kyuubi 目录：

```
[root@172 ~]# su hadoop
[hadoop@172 root]$ cd /usr/local/service/kyuubi
```

连接 kyuubi：

```
[hadoop@10kyuubi]$ bin/beeline -u
"jdbc:hive2://${zkserverip1}:${zkport},${zkserverip2}:${zkport},${zkserverip3}:${zkport}/default;serviceDisco
veryMode=zooKeeper;zooKeeperNamespace=kyuubi" -n hadoop
```

或者

```
[hadoop@10kyuubi]$ bin/beeline -u "jdbc:hive2://${kyuubiserverip}:${kyuubiserverport}" -n hadoop
```

`${zkserverip}:${zkport}` 见 `kyuubi-defaults.conf` 的 `kyuubi.ha.zookeeper.quorum` 配置。
`${kyuubiserverip}` 为任意一个部署 kyuubi server 的节点 IP，可在 弹性 MapReduce 控制台，集群服务 > KYUUBI/角色管理中查看。
`${kyuubiserverport}` 见 `kyuubi-defaults.conf` 的 `kyuubi.frontend.bind.port` 配置，默认为10009。

新建数据库并查看

在新建的数据库中新建一个表，并进行查看：

```
0: jdbc:hive2://ip:port> create database sparksql;
+---------+
| Result  |
+---------+
+---------+
No rows selected (0.326 seconds)
```

向表中插入两行数据并查看：

```
0: jdbc:hive2://ip:port> use sparksql;
+---------+
| Result  |
+---------+
+---------+
No rows selected (0.077 seconds)
0: jdbc:hive2://ip:port> create table sparksql_test(a int,b string);
+---------+
| Result  |
+---------+
+---------+
No rows selected (0.402 seconds)
0: jdbc:hive2://ip:port> show tables;
+-----------+----------------+--------------+
| database  |   tableName    | isTemporary  |
+-----------+----------------+--------------+
| sparksql  | sparksql_test  | false        |
+-----------+----------------+--------------+
1 row selected (0.108 seconds)
```

Hue 连接 kyuubi

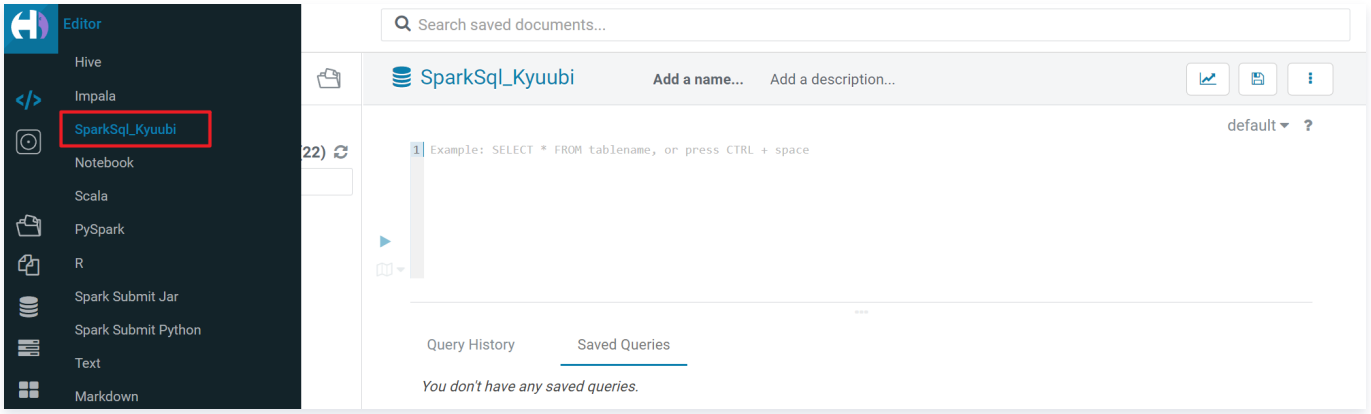
前提条件

针对在现有集群里后安装 kyuubi 场景，若想在 hue 上使用 kyuubi，需要做如下操作：

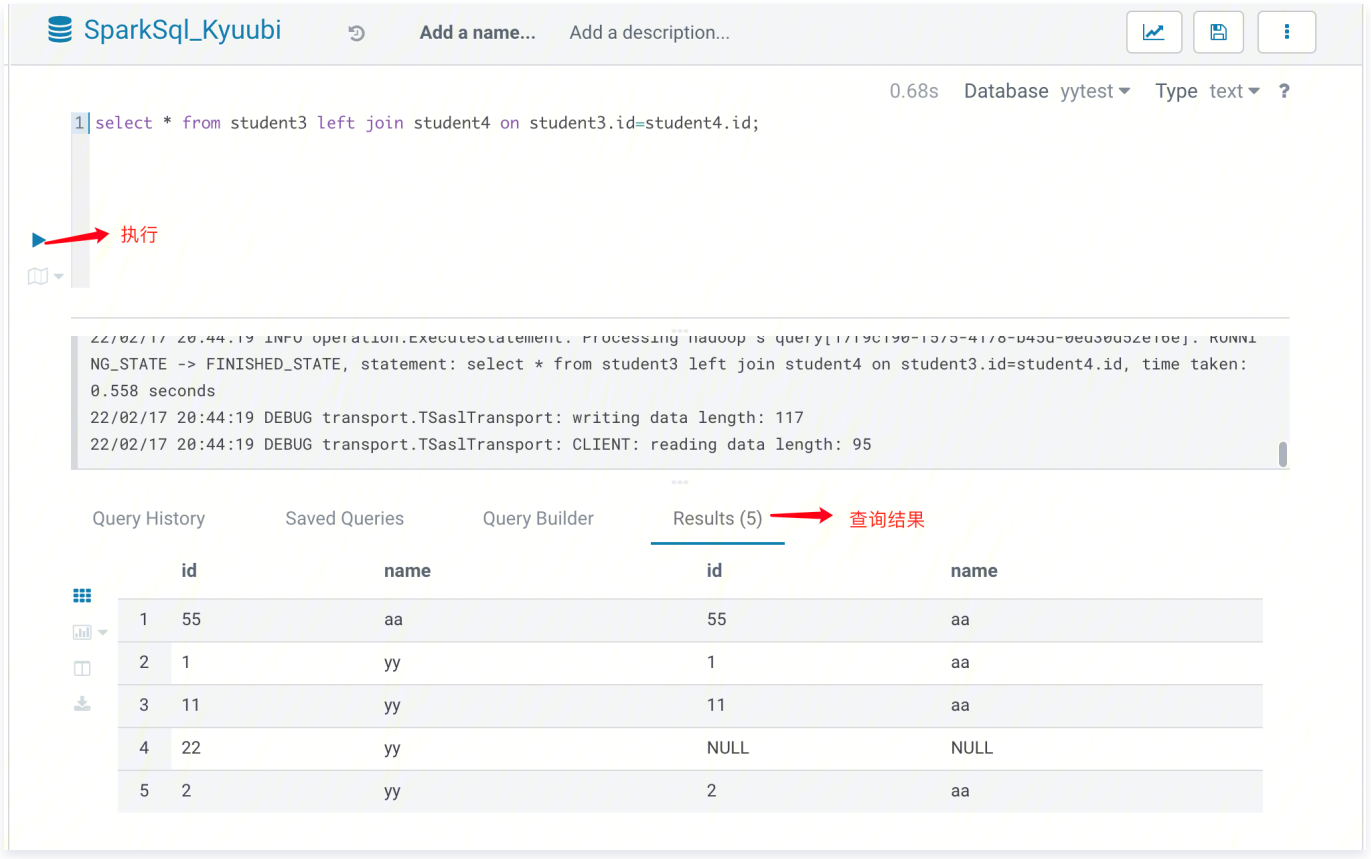
1. 进入 HDFS 的配置管理，core-site.xml 新增配置项 hadoop.proxyuser.hue.groups 值设为“*”和 hadoop.proxyuser.hue.hosts值设为“*”。
2. 重启 kyuubi 服务和 hue 服务。
3. 访问 Hue 控 制台，详情请参见 登录 Hue 控制台 。

Kyuubi 查询

1. 在 Hue 控制台左侧选择 **Editor > SparkSql_Kyuubi**。



2. 在语句输入框中输入要执行语句，然后单击**执行**，执行语句。



通过 Java 连接 Kyuubi

KyuubiServer 中集成了 Thrift 服务。Thrift 是 Facebook 开发的一个软件框架，它用来进行可扩展且跨语言的服务的开发。Kyuubi 就是基于 Thrift 的，所以能让不同的语言如 Java、Python 来调用 Kyuubi 的接口。对于 Java，Kyuubi 复用了hive jdbc 驱动，用户可以使用 Java 代码来连接 Kyuubi 并进行一系列操作。本节将演示如何使用 Java 代码来连接 Kyuubi。

1. 开发准备。
   - 确认您已经开通了腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Kyuubi 及 Spark 组件。
   - Kyuubi 等相关软件安装在路径 EMR 云服务器的/usr/local/service/路径下。

2. 使用 Maven 来创建您的工程。
推荐使用 Maven 来管理您的工程。Maven 是一个项目管理工具，能够帮助您方便的管理项目的依赖信息，即它可以通过 pom.xml 文件的配置获取 jar 包，而不用去手动添加。首先在本地下载并安装 Maven，配置好 Maven 的环境变量，如果您使用 IDE，请在 IDE 中设置好 Maven 相关配置。
在本地 shell 下进入要新建工程的目录，例如：D://mavenWorkplace 中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=$yourgroupID -DartifactId=$yourartifactID -DarchetypeArtifactId=maven-
archetype-quickstart
```

其中 $yourgroupID 即为您的包名；$yourartifactID 为您的项目名称；maven-archetype-quickstart 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。创建成功后，在 D://mavenWorkplace 目录下就会生成一个名为 $yourartifactID 的工程文件夹。其中的文件结构如下所示：

```
simple
---pom.xml          核心配置，项目根下
---src
 ---main
     ---java          Java 源码目录
     ---resources     Java 配置文件目录
 ---test
     ---java          测试源码目录
     ---resources     测试配置目录
```

其中我们主要关心 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。首先在 pom.xml 中添加 Maven 依赖：

```
<dependencies>
 <dependency>
     <groupId>org.apache.kyuubi</groupId>
     <artifactId>kyuubi-hive-jdbc-shaded</artifactId>
     <version>1.4.1-incubating</version>
 </dependency>
 <dependency>
     <groupId>org.apache.hadoop</groupId>
     <artifactId>hadoop-common</artifactId>
     <!-- keep consistent with the build hadoop version -->
     <version>2.8.5</version>
 </dependency>
</dependencies>
```

继续在 pom.xml 中添加打包和编译插件：

```
<build>
<plugins>
 <plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
  <source>1.8</source>
  <target>1.8</target>
  <encoding>utf-8</encoding>
</configuration>
 </plugin>
 <plugin>
<artifactId>maven-assembly-plugin</artifactId>
<configuration>
  <descriptorRefs>
  <descriptorRef>jar-with-dependencies</descriptorRef>
  </descriptorRefs>
</configuration>
<executions>
```

```
  <execution>
    <id>make-assembly</id>
    <phase>package</phase>
    <goals>
      <goal>single</goal>
    </goals>
  </execution>
</executions>
 </plugin>
</plugins>
</build>
```

在 src>main>Java 下右键新建一个 Java Class，输入您的 Class 名，这里使用 KyuubiJDBCTest.java，在 Class 添加样例代码：

```
import java.sql.*;

public class KyuubiJDBCTest {
 private static String driverName =
         "org.apache.kyuubi.jdbc.KyuubiHiveDriver";
 public static void main(String[] args)
         throws SQLException {
     try {
         Class.forName(driverName);
     } catch (ClassNotFoundException e) {
         e.printStackTrace();
         System.exit(1);
     }
     Connection con = DriverManager.getConnection(
             "jdbc:hive2://$kyuubiserverhost:$kyuubiserverport/default", "hadoop", "");
     Statement stmt = con.createStatement();
     String tableName = "KyuubiTestByJava";
     stmt.execute("drop table if exists " + tableName);
     stmt.execute("create table " + tableName +
             " (key int, value string)");
     System.out.println("Create table success!");
     // show tables
     String sql = "show tables '" + tableName + "'";
     System.out.println("Running: " + sql);
     ResultSet res = stmt.executeQuery(sql);
     if (res.next()) {
         System.out.println(res.getString(1));
     }
     // describe table
     sql = "describe " + tableName;
     System.out.println("Running: " + sql);
     res = stmt.executeQuery(sql);
     while (res.next()) {
         System.out.println(res.getString(1) + "\t" + res.getString(2));
     }
     sql = "insert into " + tableName + " values (42,\"hello\"),(48,\"world\")";
     stmt.execute(sql);
     sql = "select * from " + tableName;
     System.out.println("Running: " + sql);
     res = stmt.executeQuery(sql);
     while (res.next()) {
         System.out.println(String.valueOf(res.getInt(1)) + "\t"
                 + res.getString(2));
     }
     sql = "select count(1) from " + tableName;
     System.out.println("Running: " + sql);
     res = stmt.executeQuery(sql);
     while (res.next()) {
```

```
            System.out.println(res.getString(1));
        }
    }
}
```

> **注意**
> 将程序中的参数 $kyuubiserverhost 和 $kyuubiserverport 分别修改为您查到的 KyuubiServer 的 ip 和端口号的值。

如果您的 Maven 配置正确并且成功的导入了依赖包，那么整个工程即可直接编译。在本地 shell 下进入工程目录，执行下面的命令对整个工程进行打包。

```
mvn package
```

3. 上传并运行程序。
首先需要把压缩好的 jar 包上传到 EMR 集群中，使用 scp 或者 sftp 工具来进行上传。在本地 shell 下运行：

```
scp $localfile root@公网IP地址:/usr/local/service/kyuubi
```

一定要上传具有依赖的 jar 包。登录 EMR 集群切换到 Hadoop 用户并且进入目录 /usr/local/service/kyuubi 。接下来可以执行程序：

```
[hadoop@172 kyuubi]$ yarn jar $package.jar KyuubiJDBCTest
```

其中 $package.jar 为您的 jar 包的路径 + 名字，KyuubiJDBCTest 为之前的 Java Class 的名字。运行结果如下：

```
Create table success!
Running: show tables 'KyuubiTestByJava'
default
Running: describe KyuubiTestByJava
key int
value   string
Running: select * from KyuubiTestByJava
42  hello
48  world
Running: select count(1) from KyuubiTestByJava
2
```

Zeppelin 开发指南
Zeppelin 简介

最近更新时间：2024-10-29 16:58:42

Apache Zeppelin 是一款基于 Web 的 Notebook 产品，能够交互式数据分析。使用 Zeppelin，您可以使用丰富的预构建语言后端（或解释器）制作交互式的协作文档，例如 Scala(Apache Spark)、Python(Apache Spark)、SparkSQL、 Hive、Shell 等。

> **说明**
> EMR-V3.3.0及以上、EMR-V2.6.0及以上，已默认配置了 flink、hbase、kylin、livy、spark 的 Interpreter，其他版本和组件可参考 官方文档 根据 Zeppelin 版本进行配置。

前提条件

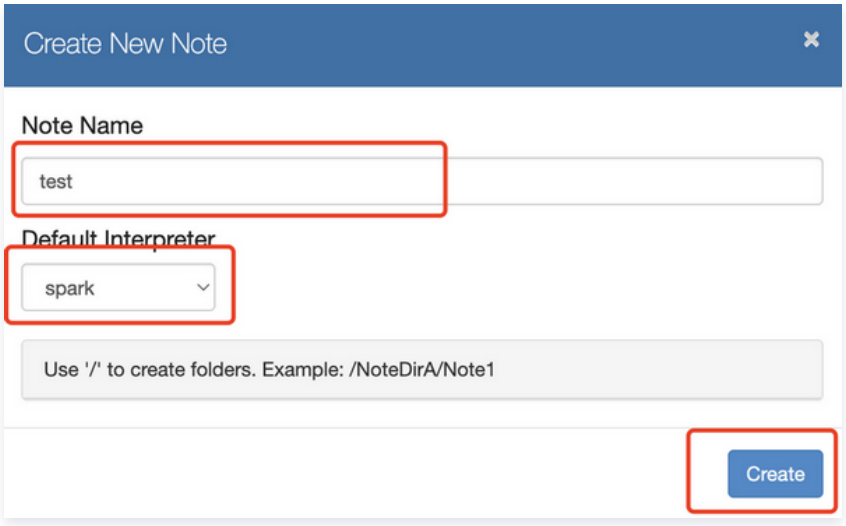
- 已创建集群，并选择 Zeppelin 服务，详情参见 创建 EMR 集群 。
- 在集群的 EMR 安全组中，开启22、30001和18000端口（新建集群默认开启22和30001）及必要的内网通信网段，新安全组以 emr-xxxxxxxx_yyyyMMdd 命名，请勿手动修改安全组名称。
- 按需添加所需服务，如，Spark、Flink、HBase、Kylin。

登录 Zeppelin

1. 创建集群，选择 Zeppelin 服务，详情参见 创建 EMR 集群 。
2. 在 EMR 控制台 左侧的导航栏，选择集群服务。
3. 单击 Zeppelin 所在的卡片，单击 **Web UI 地址**，访问 Web UI 页面。
4. 在 EMR-V2.5.0 及以前版本、EMR-V3.2.1 及以前版本，设置了默认登录权限，用户名密码为 admin:admin。如需更改密码，可修改配置文件/usr/local/service/zeppelin-0.8.2/conf/shiro.ini 中的 users 和 roles 选项。更多配置说明，可参见 文档 。
5. 在 EMR-V2.6.0 及以后版本、EMR-V3.3.0 及以后版本，Zeppelin 登录已集成 Openldap 账户，只能用 Openldap 账户密码登录，新建集群后 Openldap 默认账户是 root 和 hadoop，默认密码是集群密码，且只有 root 账户拥有 zeppelin 管理员权限，有权访问解析器配置页面。

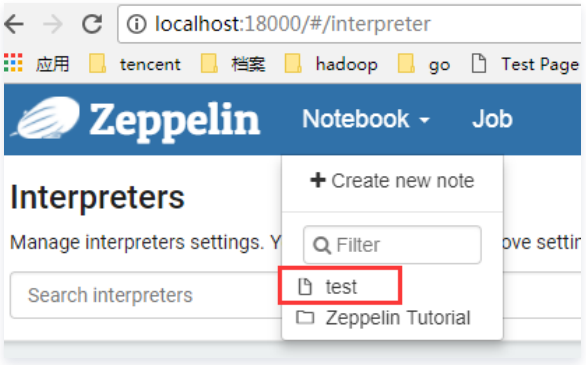
使用 spark 功能完成 wordcount

1. 单击页面左侧 **Create new note**，在弹出页面中创建 notebook。



2. EMR-V3.3.0 及以上、EMR-V2.6.0 及以上，已默认配置 Spark 对接 EMR 的集群（Spark On Yarn）。
   - 如果您的版本是 EMR-V3.1.0、EMR-V2.5.0、EMR-V2.3.0，请参考 文档 进行 Spark 解释器配置。
   - 如果您的版本是 EMR-V3.2.1，请参考 文档 进行 Spark 解释器配置。

3. 进入自己的 notebook。



4. 编写 Spark 程序，以下使用 Spark Scala 方式作为示例，其中 %spark 表示执行 Spark Scala 代码：

```
%spark

val df = spark.read.options(Map("inferSchema"->"true","delimiter"->";","header"->"true"))
.csv("file:///usr/local/service/spark/examples/src/main/resources/people.csv")
z.show(df)
df.registerTempTable("people")
```

5. 返回信息结果如图所示：

Zeppelin 解析器配置

最近更新时间：2023-07-11 21:50:57

本文以 Zepplin 0.91 以上版本为例，主要介绍常见 Zeppelin 解析的配置以及验证方法。
使用 root 用户登录 Zepplin WebUI，密码默认与创建集群时的密码相同，单击右上角**用户 > Interpreter**，搜索需配置解析器的组件，单击 **edit** 即可进行配置。

Spark 解析器

配置

```
SPARK_HOME: /usr/local/service/spark
spark.master: yarn
spark.submit.deployMode: cluster
spark.app.name: zeppelin-spark
```

验证

1. 先把 wordcount.txt 文件上传到 emr hdfs 的/tmp 路径下。
2. 通过 core-site.xml 找到 fs.defaultFS 配置项值 hdfs://HDFS45983。
3. 在 notebook 中执行 spark 相关代码。

```
%spark
val data = sc.textFile("hdfs://HDFS45983/tmp/wordcount.txt")
case class WordCount(word: String, count: Integer)
val result = data.flatMap(x => x.split(" ")).map(x => (x, 1)).reduceByKey(_ + _).map(x => WordCount(x._1,
x._2))
result.toDF().registerTempTable("result")

%sql
select * from result
```

Flink 解析器

配置

```
FLINK_HOME: /usr/local/service/flink

flink.execution.mode: yarn
```

验证

```
%flink
val data = benv.fromElements("hello world", "hello flink", "hello hadoop")
data.flatMap(line => line.split("\\s"))
            .map(w => (w, 1))
            .groupBy(0)
            .sum(1)
            .print()
```

HBase 解析器

配置

```
hbase.home: /usr/local/service/hbase

hbase.ruby.sources: lib/ruby
```

```
zeppelin.hbase.test.mode: false
```

> **注意**
> 此解析器依赖的 jar 包已集成到集群/usr/local/service/zeppelin/local-repo 路径下，因此不用配置 dependencies，如需已定义 jar 包才需配置 dependencies。

验证

```
%hbase
help 'get'

%hbase
list
```

Livy 解析器

配置

```
zeppelin.livy.url: http://ip:8998
```

验证

```
%livy.spark
sc.version

%livy.pyspark
print "1"

%livy.sparkr
hello <- function( name ) {
    sprintf( "Hello, %s", name );
}
hello("livy")
```

Kylin 解析器

配置

1. 在 Kylin 控制台页面中新建一个 default 的 Project。
2. 配置 zeppelin 的 kylin interpreter。

```
kylin.api.url: http://ip:16500/kylin/api/query

kylin.api.user: ADMIN

kylin.api.password: KYLIN

kylin.query.project: default
```

验证

```
%kylin(default)

select count(*) from table1
```

JDBC 解析器

1. MySQL 解析器配置

```
default.url: jdbc:mysql://ip:3306

default.user: xxx

default.password: xxx

default.driver: com.mysql.jdbc.Driver
```

> **注意**
> 此解析器依赖的 jar 包已集成到集群/usr/local/service/zeppelin/local-repo 路径下，因此不用配置 dependencies，如需已定义 jar 包才需配置 dependencies。

验证

```
%mysql
show databases
```

2. Hive 解析器配置

```
default.url: jdbc:hive2://ip:7001

default.user: hadoop

default.password:

default.driver: org.apache.hive.jdbc.HiveDriver
```

> **注意**
> 此解析器依赖的 jar 包已集成到集群/usr/local/service/zeppelin/local-repo 路径下，因此不用配置 dependencies，如需已定义 jar 包才需配置 dependencies。

验证

```
%hive
show databases

%hive
use default;
show tables;
```

3. Presto 解析器配置

```
default.url: jdbc:presto://ip:9000?user=hadoop

default.user: hadoop

default.password:

default.driver: io.prestosql.jdbc.PrestoDriver
```

> **注意**
> 此解析器依赖的 jar 包已集成到集群/usr/local/service/zeppelin/local-repo 路径下，因此不用配置 dependencies，如需已定义 jar 包才需配置 dependencies。

验证

```
%presto
show catalogs;

%presto
show schemas from hive;

%presto
show tables from hive.default;
```

更多版本及解析器配置请参考 Zeppelin 官网文档 。

Hudi 开发指南
Hudi 简介

最近更新时间：2023-12-20 17:06:31

Apache Hudi 在 HDFS 的数据集上提供了插入更新和增量拉取的流原语。一般来说，我们会将大量数据存储到 HDFS，新数据增量写入，而旧数据鲜有改动，特别是在经过数据清洗，放入数据仓库的场景。而且在数据仓库如 Hive 中，对于 update 的支持非常有限，计算昂贵。另一方面，若是有仅对某段时间内新增数据进行分析的场景，则 Hive、Presto、Hbase 等也未提供原生方式，而是需要根据时间戳进行过滤分析。在此需求下，Hudi 可以提供这两种需求的实现。第一个是对 record 级别的更新，另一个是仅对增量数据的查询。且 Hudi 提供了对 Hive、Presto、Spark 的支持，可以直接使用这些组件对 Hudi 管理的数据进行查询。
Hudi 是一个通用的大数据存储系统，主要特性：

- 摄取和查询引擎之间的快照隔离，包括 Hive、Presto 和 Spark。
- 支持回滚和存储点，可以恢复数据集。
- 自动管理文件大小和布局，以优化查询性能准实时摄取，为查询提供最新数据。
- 实时数据和列数据的异步压缩。

时间轴

Hudi 维护一条包含在不同的**即时**时间所有对数据集操作的**时间轴**，从而提供了从不同时间点出发得到不同的视图下的数据集。
Hudi 即时包含以下组件：

- 操作类型：对数据集执行的操作类型。
- 即时时间：即时时间通常是一个时间戳（例如20190117010349），该时间戳按操作开始时间的顺序单调增加。
- 状态：即时的状态。

文件组织

Hudi 将 DFS 上的数据集组织到 `基本路径` 下的目录结构中。数据集分为多个分区，这些分区是包含该分区的数据文件的文件夹，这与 Hive 表非常相似。
每个分区被相对于基本路径的特定 `分区路径` 区分开来。在每个分区内，文件被组织为 `文件组` ，由 `文件id` 唯一标识。每个文件组包含多个 `文件切片` ，其中每个切片包含在某个提交/压缩即时时间生成的基本列文件 `*.parquet` 以及一组日志文件 `*.log*` ，该文件包含自生成基本文件以来对基本文件的插入/更新。
Hudi 采用 MVCC 设计，其中压缩操作将日志和基本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 DFS 上的空间。Hudi 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 Upsert。
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。

存储类型

Hudi 支持以下存储类型：

- 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：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。

下表总结了这两种存储类型之间的权衡：

权衡	写时复制	读时合并
数据延迟	更高	更低
查询延迟	更低	更高
更新代价（I/O）	更高（重写整个 parquet 文件）	更低（追加到增量日志）
写放大	更高	更低（取决于压缩策略）

Hudi 对 EMR 底层存储支持

- HDFS
- COS

安装 Hudi

进入 EMR 购买页，选择产品版本为 **EMR-V3.6.0**，选择可选组件为 **Hudi 0.13.0**。各个产品版本对应的 Hudi 组件版本可以查看组件版本概览 页面。

> **注意：**
> Hudi 组件依赖 Hive 和 Spark 组件， 如果选择安装 Hudi 组件，需同时选择安装 Hive 和 Spark 组件。

使用示例

本示例将演示使用 SparkSQL 建表并写入数据，然后使用 Hive 和 Trino 进行查询分析。

使用 SparkSQL 读写 Hudi 表

- 登录 master 节点，切换为 hadoop 用户。
- Hudi 支持使用 SparkSQL 的 **HoodieSparkSessionExtension** 扩展进行数据读写：

```
spark-sql --master yarn \
--num-executors 2 \
--executor-memory 1g \
--executor-cores 2 \
--jars /usr/local/service/hudi/hudi-bundle/hudi-spark3.3-bundle_2.12-0.13.0.jar \
--conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer' \
--conf 'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension' \
--conf 'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog'
```

说明：
其中 --master 表示您的 master URL，--num-executors 表示 executor 数量，--executor-memory 表示 executor 的储存容量，以上参数也可以根据您的实际情况作出修改。--jars 使用的依赖包版本在不同 EMR 版本中可能存在差异，请在 /usr/local/service/hudi/hudi-bundle 目录下查看并使用正确的依赖包。

- 建表

```
-- 创建COW非分区表

spark-sql> create table hudi_cow_nonpcf_tbl (
  uuid int,
  name string,
  price double
) using hudi
tblproperties (
  primaryKey = 'uuid'
);

-- 创建COW分区表

spark-sql> 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);

-- 创建MOR分区表

spark-sql> create table hudi_mor_tbl (
  id int,
  name string,
  price double,
  ts bigint,
  dt string
) using hudi
```

```
tblproperties (
  type = 'mor',
  primaryKey = 'id',
  preCombineField = 'ts'
)
partitioned by (dt);
```

- 写入数据

```
-- insert into non-partitioned table
spark-sql> insert into hudi_cow_nonpcf_tbl select 1, 'a1', 20;

-- insert dynamic partition
spark-sql> insert into hudi_cow_pt_tbl partition (dt, hh) select 1 as id, 'a1' as name, 1000 as ts, '2021-12-
09' as dt, '10' as hh;

-- insert static partition
spark-sql> insert into hudi_cow_pt_tbl partition(dt = '2021-12-09', hh='11') select 2, 'a2', 1000;
spark-sql> insert into hudi_mor_tbl partition(dt = '2021-12-09') select 1, 'a1', 20, 1000;
```

- 查询数据

```
spark-sql> select * from hudi_cow_nonpcf_tbl;
20230808165859974   20230808165859974_0_1   uuid:1      f564e6c7-631c-4f32-a01d-e042f37ad6e6-0_0-21-
19_20230808165859974.parquet    1   a1  20.0
Time taken: 6.255 seconds, Fetched 1 row(s)

spark-sql> select count(*) from hudi_mor_tbl;
1
Time taken: 0.955 seconds, Fetched 1 row(s)

spark-sql> select name, count(*) from hudi_cow_pt_tbl group by name;
a2  1
a1  1
Time taken: 2.049 seconds, Fetched 2 row(s)
```

使用 Hive 查询 Hudi 表

以下演示在 Hive CLI 中如何查询上述创建的 Hudi 表：

> **说明：**
> 不同 EMR 版本中加载的 Hudi 依赖包版本可能存在不同，请在/usr/local/service/hudi/hudi-bundle目录下选择正确版本依赖包进行加载。

```
hive> add jar /usr/local/service/hudi/hudi-bundle/hudi-hadoop-mr-bundle-0.13.0.jar;
Added [/usr/local/service/hudi/hudi-bundle/hudi-hadoop-mr-bundle-0.13.0.jar] to class path

hive> show tables;
OK
hudi_cow_nonpcf_tbl
hudi_cow_pt_tbl
hudi_mor_tbl
hudi_mor_tbl_ro
hudi_mor_tbl_rt
Time taken: 0.023 seconds, Fetched: 5 row(s)

hive> set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
hive> select * from hudi_mor_tbl_ro;
OK
```

```
20230808174602565   20230808174602565_0_1   id:1    dt=2021-12-09   af40667d-1dca-4163-89ca-2c48250985b2-0_0-
34-1617_20230808174602565.parquet   1   a1  20.0    1000    2021-12-09
Time taken: 0.159 seconds, Fetched: 1 row(s)

hive> set hive.vectorized.execution.enabled=false;
hive> select name, count(*) from hudi_mor_tbl_rt group by name;
a1  1
Time taken: 17.618 seconds, Fetched: 1 row(s)
```

使用 Trino 查询 Hudi 表

以下演示如何在 Trino 中查询上述创建的 Hudi 表。

- EMR 3.6.0之前版本：
  - Trino 查询 Hudi 使用 Hive Connector，无需额外配置。
  - 将 /usr/local/service/hudi/hudi-bundle/hudi-hadoop-mr-bundle-$hudi_version.*jar* 拷贝到集群所有节点的/usr/local/service/trino/plugin/hive/路径下，并重启 Trino 服务。
- EMR-3.6.0版本及以后版本：
  - 需要在 Trino 组件配置管理中添加 hudi connector 配置文件。参考 配置更新 说明进行新增配置文件操作。其中 ${hivemetastore_uri} 需要改为集群的 Hive Metastore 服务地址，具体可以查看 Hive 组件配置文件 hive-site.xml 中的 hive.metastore.uris 配置项。
  - 配置完成后重启Trino服务。

```
connector.name=hudi
hive.metastore.uri=${hivemetatore_uri}
```

完成以上步骤后，可以在 Trino 中查询 Hudi 表：

```
trino:default> show tables;
        Table
---------------------
 hudi_cow_nonpcf_tbl
 hudi_cow_pt_tbl
 hudi_mor_tbl
 hudi_mor_tbl_ro
 hudi_mor_tbl_rt
(5 rows)

Query 20230808_113217_00005_r3tqk, FINISHED, 3 nodes
Splits: 12 total, 12 done (100.00%)
0.34 [11 rows, 359B] [32 rows/s, 1.03KB/s]

trino:default> select uuid, name, price from hudi_cow_nonpcf_tbl;
 uuid | name | price
------+------+-------
    1 | a1   |  20.0
(1 row)

Query 20230808_114808_00001_72vwk, FINISHED, 1 node
Splits: 5 total, 5 done (100.00%)
3.48 [1 rows, 440KB] [0 rows/s, 126KB/s]

trino:default> select id, name, price from hudi_mor_tbl_ro;
 id | name | price
----+------+-------
  1 | a1   |  20.0
(1 row)

Query 20230808_115532_00006_72vwk, FINISHED, 1 node
Splits: 5 total, 5 done (100.00%)
1.75 [1 rows, 440KB] [0 rows/s, 251KB/s]
```

```
trino:default> select name, count(*) from hudi_mor_tbl_rt group by name;
 name | _col1
------+-------
 a1   |     1
(1 row)

Query 20230808_115728_00009_72vwk, FINISHED, 2 nodes
Splits: 21 total, 21 done (100.00%)
0.63 [1 rows, 440KB] [1 rows/s, 698KB/s]
```

Superset 开发指南
Superset 简介

最近更新时间：2025-06-10 10:21:22

Apache Superset 是一个数据浏览和可视化 Web 应用程序。EMR 上的 Superset，原装了对 Mysql、Hive、Presto、Impala、Kylin 的支持。

Superset 特性

- 支持几乎所有主流的数据库，包括 MySQL、PostgresSQL、Oracle、SQL Server、SQLite、SparkSQL 等。
- 丰富的可视化展示，支持自定义创建 dashboard。
- 数据的展示完全可控，可自定义展示字段、聚合数据、数据源等。

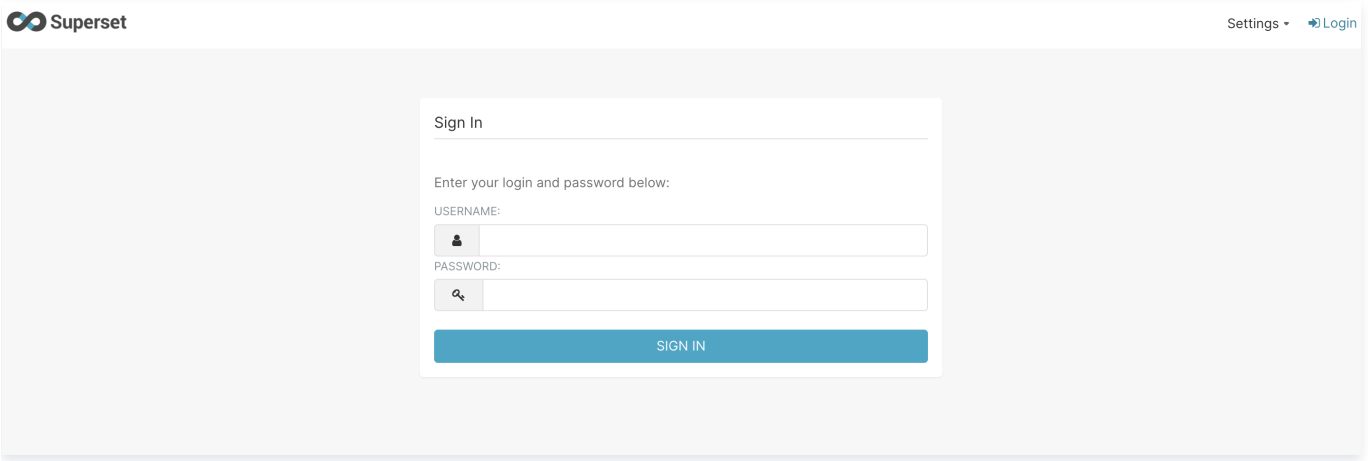
前提条件

1. 已创建弹性-MapReduce（简称EMR）的 Hadoop 或 Druid 集群，并选择了 Superset 服务，详情请参见 创建 EMR 集群 。
2. Superset 默认安装在集群的 master 节点上，打开 master 节点的安全组策略，确保您的网络可以访问 master 节点的18088端口。

登录

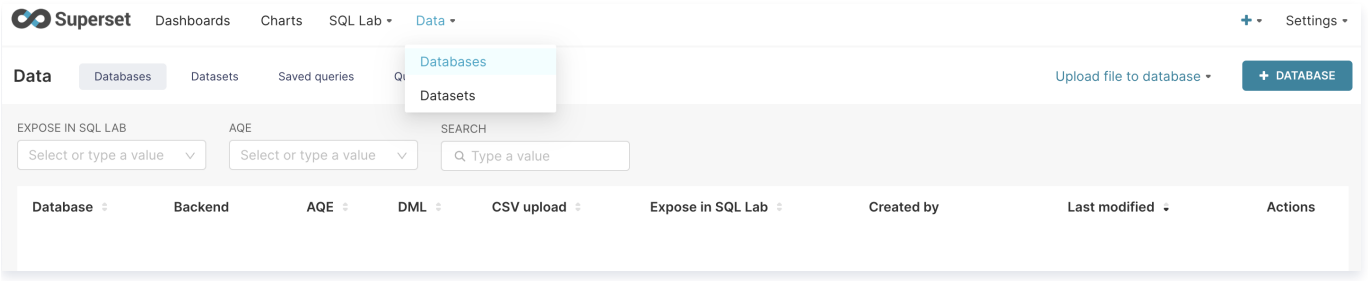
在浏览器地址栏中输入 `http://${master_ip}:18088` （或者通过 EMR 控制台 > **集群服务**），打开 Superset 登录界面，默认用户名为 admin，密码为您创建集群时的密码。

> **注意**
> 不同 Superset 社区版本 Web 页面交互可能存在差异，本文以 Superset 1.5.1 版本为示例，更多版本及使用方式可参考 Superset 官网文档 。

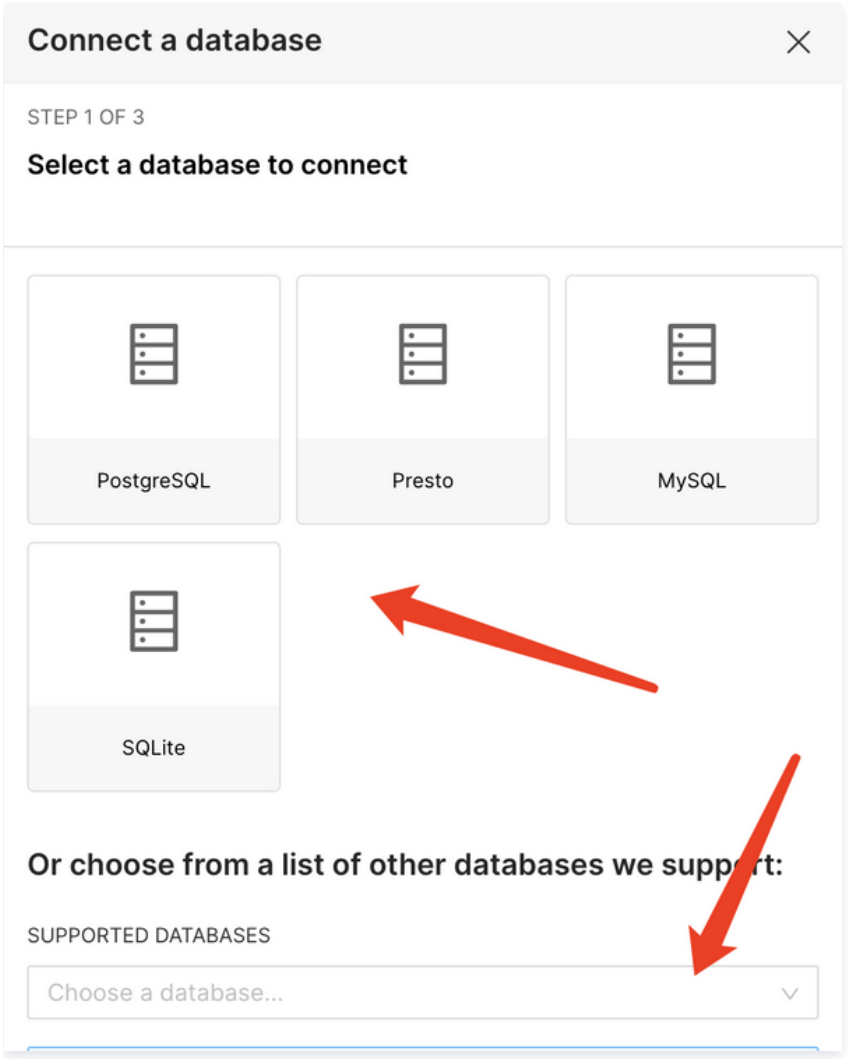


添加 DataBase

进入 **Data > Databases** 界面，单击 **+DATABASE**。



进入如下页面，即可选择您需要添加的数据源。



各个数据库的链接 SQLAlchemy URI 如下：

名称	SQLAlchemy URI	备注
Mysql	mysql+pymysql://<mysqlname>:<password>@<mysql_ip>:<mysql_port>/<your_database>	• mysqlname：连接 mysql 使用的用户名 • password：mysql 密码 • your_database：需要连接的 mysql 数据库
Hive	hive://hadoop@<master_ip>:7001/default?auth=NONE	Master_ip：EMR 集群的 master_ip
presto	presto://hive@<master_ip>:9000/hive/<hive_db_name>	• Master_ip：EMR 集群的 master_ip • hive_db_name：hive 中的数据库名称，不填默认为 default
impala	impala://<core_ip>:<port>	• core_ip：当前集群的 core_ip • port：当前集群 Impala 服务参数 hs2_port 的值
kylin	kylin://<kylin_user>:<password>@<master_ip>:16500/<kylin_project>	• kylin_user：kylin 的用户名 • password：kylin 的密码 • master_ip：EMR 集群的 master_ip • kylin_project：kylin 的项目

自行添加新 Database

Superset 支持 Database。如果您需要安装其它的数据库，可通过如下操作进行：

1. 登录 EMR 集群 master 所在机器。
2. 执行命令 `source /usr/local/service/superset/bin/activate` 。
3. pip3 install 对应的 Python 库。
4. 重启 Superset。

Impala 开发指南

Impala 简介

最近更新时间：2024-10-11 15:17:11

Apache Impala 项目为存储在 Apache Hadoop 文件格式的数据提供高性能、低延迟的 SQL 查询。它对查询进行快速响应，同时支持对分析查询进行交互式的数据探索和查询调整，而不是传统上那种与 SQL-on-Hadoop 技术相关联的长时间批量作业。
Impala 不同于 hive，hive 底层执行使用的是 MapReduce 引擎，仍然是一个批处理过程。而 impala 的中间结果不写入磁盘，即时通过网络以流的形式传递，大大降低了节点的 IO 开销。
Impala 与 Apache Hive 数据库集成，在两个组件之间共享数据库和表。通过与 Hive 的高度集成，以及与 HiveQL 语法的兼容性，您可以使用 Impala 或 Hive 创建表、发起查询、加载数据等。

前提条件

- 确认已开通腾讯云，并且创建了一个 EMR 集群。在创建 EMR 集群时，需要在软件配置界面选择 Impala 组件。
- Impala 安装在路径 EMR 云服务器的 `/data/` 路径下（ `/data/Impala` ）。

准备数据

首先需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例，可选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并进入 Impala 文件夹。

```
[root@10 ~]# su hadoop
[hadoop@10 root]$ cd /data/Impala/
```

新建一个 bash 脚本文件 gen_data.sh，在其中添加以下代码：

```
#!/bin/bash
MAXROW=1000000 #指定生成数据行数
for((i = 0; i < $MAXROW; i++))
do
      echo $RANDOM, \"$RANDOM\"
done
```

使用 chmod 命令为 gen_data.sh 赋予可执行权限后，执行如下命令：

```
[hadoop@10 ~]$ ./gen_data.sh > impala_test.data
```

这个脚本文件会生成1000000个随机数对，并且保存到文件 `impala_test.data` 中。然后把生成的测试数据上传到 HDFS 中，执行如下命令：

```
[hadoop@10 ~]$ hdfspath="/impala_test_dir"
[hadoop@10 ~]$ hdfs dfs -mkdir $hdfspath
[hadoop@10 ~]$ hdfs dfs -put ./impala_test.data $hdfspath
```

其中 $hdfspath 为 HDFS 中您存放文件的路径。最后可用如下命令，验证数据是否正常放到 hdfs 上。

```
[hadoop@10 ~]$ hdfs dfs -ls $hdfspath
```

Impala 基础操作

由于不同 Impala 版本社区组件接口协议及路径默认值变化，不同版本 impala-shell 路径如下表所示。

Imapala 版本	impala-shell 路径	impala-shell 默认连接端口
4.1.0/4.0.0	/data/Impala/shell	27009
3.4.0	/data/Impala/bin/impala-shell	27001
2.10.0	/data/Impala/bin	27001

以下操作以 Impala3.4.0版本为示例：

连接 Impala

登录 EMR 集群的 Master 节点，切换到 Hadoop 用户并且进入 Impala 目录，并连接 Impala：

```
[root@10 Impala]# cd /data/Impala/shell;./impala-shell -i $core_ip:27001
```

其中 core_ip 为 EMR 集群的 core 节点 IP，也可以用 task 节点的 IP，正常登录后显示如下：

```
Connected to $core_ip:27001
Server version: impalad version 3.4.1-RELEASE RELEASE (build Could not obtain git hash)
***********************************************************************************
Welcome to the Impala shell.
(Impala Shell 3.4.1-RELEASE (ebled66) built on Tue Nov 20 17:28:10 CST 2021)

The SET command shows the current value of all shell and query options.
***********************************************************************************
[$core_ip:27001] >
```

也可以登录 core 节点或者 task 节点后，直接连接，执行语句如下：

```
cd /data/Impala/shell;./impala-shell -i localhost:27001
```

创建 Impala 库

在 Impala 下执行以下语句，查看数据库：

```
[10.1.0.215:27001] > show databases;
Query: show databases
+------------------+-----------------------------------------------+
| name             | comment                                       |
+------------------+-----------------------------------------------+
| _impala_builtins | System database for Impala builtin functions  |
| default          | Default Hive database                         |
+------------------+-----------------------------------------------+
Fetched 2 row(s) in 0.09s
```

使用 `create` 指令创建一个数据库：

```
[localhost:27001] > create database experiments;
Query: create database experiments
Fetched 0 row(s) in 0.41s
```

使用 `use` 指令转到刚创建的 test 数据库下：

```
[localhost:27001] > use experiments;
Query: use experiments
```

查看当前所在库，执行如下语句：

```
select current_database();
```

创建 Impala 表

使用 `create` 指令在 experiments 数据库下创建一个新的名为 impala_test 的内部表：

```
[localhost:27001] > create table t1 (a int, b string) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
Query: create table t1 (a int, b string)
```

```
Fetched 0 row(s) in 0.13s
```

查看所有表：

```
[localhost:27001] > show tables;
Query: show tables
+------+
| name |
+------+
| t1   |
+------+
Fetched 1 row(s) in 0.01s
```

查看表结构：

```
[localhost:27001] > desc t1;
Query: describe t1
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| a    | int    |         |
| b    | string |         |
+------+--------+---------+
Fetched 2 row(s) in 0.01s
```

将数据导入表中

对于存放在 HDFS 中的数据，使用如下指令来将其导入表中：

```
LOAD DATA INPATH '$hdfspath/impala_test.data' INTO TABLE t1;
```

其中 $hdfspath 为 HDFS 中您存放文件的路径。导入完成后，HDFS 上导入路径上的源数据文件将会被删除。存放到 Impala 内部表的存放路径 `/usr/hive/warehouse/experiments.db/t1` 下。也可以建立外部表，语句如下：

> **注意**
> 这里只有一条指令，如果不输入分号“;”，可以把一条指令放在多行输入。

```
CREATE EXTERNAL TABLE t2
(
   a INT,
   b string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/impala_test_dir';
```

执行查询

```
[localhost:27001] > select count(*) from experiments.t1;
Query: select count(*) from experiments.t1
Query submitted at: 2019-03-01 11:20:20 (Coordinator: http://10.1.0.215:20004)
Query progress can be monitored at: http://10.1.0.215:20004/query_plan?
query_id=f1441478dba3a1c5:fa7a8eef00000000
+----------+
| count(*) |
+----------+
| 1000000  |
+----------+
Fetched 1 row(s) in 0.63s
```

最后输出结果为1000000。

删除表

```
[localhost:27001] > drop table experiments.t1;
Query: drop table experiments.t1
```

更多 Impala 的操作，详见 官方文档 。

通过 JDBC 连接 Impala

Impala 也可以通过 Java 代码来连接，步骤类似于 通过 Java 连接 Hive 。
唯一区别的是， `$hs2host` 和 `$hsport` ，其中 `$hs2host` 是 EMR 集群中任意 core 节点或者 task 节点的 IP。而 hsport 可以在对应节点的 Impala 目录下，配置文件 `conf/impalad.flgs` 中查看。

```
[root@10 ~]# su hadoop
[hadoop@10 root]$ cd /data/Impala/
[hadoop@10 Impala]$ grep hs2_port conf/impalad.flgs
```

如何映射 Hbase 表

Impala 会使用 Hive 的元数据信息，所有在 Hive 中的表，都可以在 Impala 中读到。可通过 在 Hive 中映射 HBase 表 达到在 Impala 中映射 HBase 表。

Impala 运维手册

最近更新时间：2023-06-21 15:58:46

数据变大，Impala 启动失败

背景

当 Impala 中的元数据信息太多（例如几百个库、几万个表），Impala 在启动的时候，需要把这些元数据信息广播到所有节点上，默认这个动作的超时时间是10s。当元数据较大且较易触发时，可通过在 Impala 的启动配置文件 `/data/Impala/conf/impalad.flgs` 中设置 `-statestore_subscriber_timeout_seconds=100` 。

问题排查确定

通常出现这个问题，会在 Impala 的日志 `/data/emr/impala/logs` 中出现如下内容：

```
Connection with state-store lost
Trying to re-register with state-store
```

配置低导致，Impala 查询慢

虽然 Impala 不是内存数据库，但在处理大型表、大型数据时，还是应该为 Impala 分配更多的物理内存，一般建议是使用128G或者更多的内存，并分配80%给到 Impala 进程。

SELECT 语句失败

可能原因如下：

1. 由于某个特定节点的性能，容量或网络问题而导致超时。查看 impala log，确定是什么节点，检查该节点机器网络是否有问题。
2. join 查询使用过多的内存，导致查询自动取消。检查 join 语句是否合理，或者加大机器内存。
3. 受节点上如何生成本机代码以处理查询中特定 WHERE 子句的影响。例如，可以生成特定节点的处理器不支持的机器指令。如果日志中的错误消息表明原因是非法指令，请考虑暂时关闭本机代码生成，然后再次尝试查询。
4. 格式错误的输入数据，例如具有超长行的文本数据文件，或者与 CREATE TABLE 语句的 FIELDS TERMINATED BY 子句中指定的字符不匹配的分隔符。检查是否有超长的数据。并检查建表语句，是否制定了正确的分隔符。

设置查询的使用内存限制

```
[localhost:27001] > set mem_limit=3000000000;
MEM_LIMIT set to 3000000000
[localhost:27001] > select 5;
Query: select 5
+---+
|5 |
+---+
|5 |
+---+
[localhost:27001] > set mem_limit=3g;
MEM_LIMIT set to 3g
[localhost:27001] > select 5;
Query: select 5
+---+
|5 |
+---+
|5 |
+---+
[localhost:27001] > set mem_limit=3gb;
MEM_LIMIT set to 3gb
[localhost:27001] > select 5;
+---+
|5 |
+---+
|5 |
```

```
+---+
[localhost:27001] > set mem_limit=3m;
MEM_LIMIT set to 3m
[localhost:27001] > select 5;
+---+
|5 |
+---+
|5 |
+---+
[localhost:27001] > set mem_limit=3mb;
MEM_LIMIT set to 3mb
[localhost:21000] > select 5;
+---+
|5 |
+---+
```

分析 COS/CHDFS 上的数据

最近更新时间：2024-08-23 14:52:21

本节将基于腾讯云对象存储 COS 展示 Impala 更多使用方法，数据来源于直接插入数据、COS 数据。

开发准备

1. 由于任务中需要访问腾讯云对象存储（COS），所以需要在 COS 中先 创建一个存储桶（Bucket）。
2. 确认您已开通腾讯云，且已创建一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Impala 组件，并且在基础配置页面开启对象存储的授权。
3. Impala 等相关软件安装在路径 EMR 云服务器的 `/usr/local/service/` 路径下。

操作步骤

登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考 登录 Linux 实例，可选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户，并连接到 Impala：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]$ impala-shell -i $host:27001
```

$host 为您的 Impala-Daemon 角色所在的内网 IP，27001是 impalad.flgs 中 beeswax_port 配置项的值。

步骤一：创建表（record）

```
[$host:27001 ] > create table record(id int, name string) row format delimited fields terminated by ','
stored as textfile location 'cosn://$bucketname/';
Query: create table record(id int, name string) row format delimited fields terminated by ',' stored as
textfile location 'cosn://$bucketname/'
Fetched 0 row(s) in 3.07s
其中 $bucketname 为您的 COS 存储桶名加路径，如果使用 CHDFS 将 location 的值换成 ofs://$mountname/，$mountname 为您的
CHDFS 挂载地址加路径
查看表信息，确认 location 是 cos 路径
[$host:27001 ] > show create table record2;
Query: show create table record2
+------------------------------------------------------------------------+
| result                                                                 |
+------------------------------------------------------------------------+
| CREATE TABLE default.record2 (                                         |
|   id INT,                                                              |
|   name STRING                                                          |
| )                                                                      |
| ROW FORMAT DELIMITED FIELDS TERMINATED BY ','                          |
| WITH SERDEPROPERTIES ('field.delim'=',', 'serialization.format'=',')   |
| STORED AS TEXTFILE                                                     |
| LOCATION 'cosn://$bucketname'                                          |
| TBLPROPERTIES ('numFiles'='19', 'totalSize'='1870')                    |
+------------------------------------------------------------------------+
Fetched 1 row(s) in 5.90s
```

步骤二：向表中插入数据

```
[$host:27001] > insert into record values(1,"test");
Query: insert into record values(1,"test")
Query submitted at: 2020-08-03 11:29:16 (Coordinator: http://$host:27004)
Query progress can be monitored at: http:/$host:27004/query_plan?query_id=b246d3194efb7a8f:bc60721600000000
Modified 1 row(s) in 0.64s
```

步骤三：使用 Impala 查询表

```
[$host:27001] > select * from record;
Query: select * from record
Query submitted at: 2020-08-03 11:29:31 (Coordinator: http://172.30.1.136:27004)
Query progress can be monitored at: http://$host:27004/query_plan?query_id=8148da96f8c0d369:4b26432a00000000
+----+----------+
| id | name     |
+----+----------+
| 1  | test     |
+----+----------+
Fetched 1 row(s) in 0.37s
```

Druid 开发指南
Druid 简介

最近更新时间：2024-08-15 14:44:01

Apache Druid 是一个分布式的、支持实时多维 OLAP 分析的数据处理系统，用于解决如何在大规模数据集下进行快速的、交互式的查询和分析的问题。

基本特点

Apache Druid 具有以下特点：

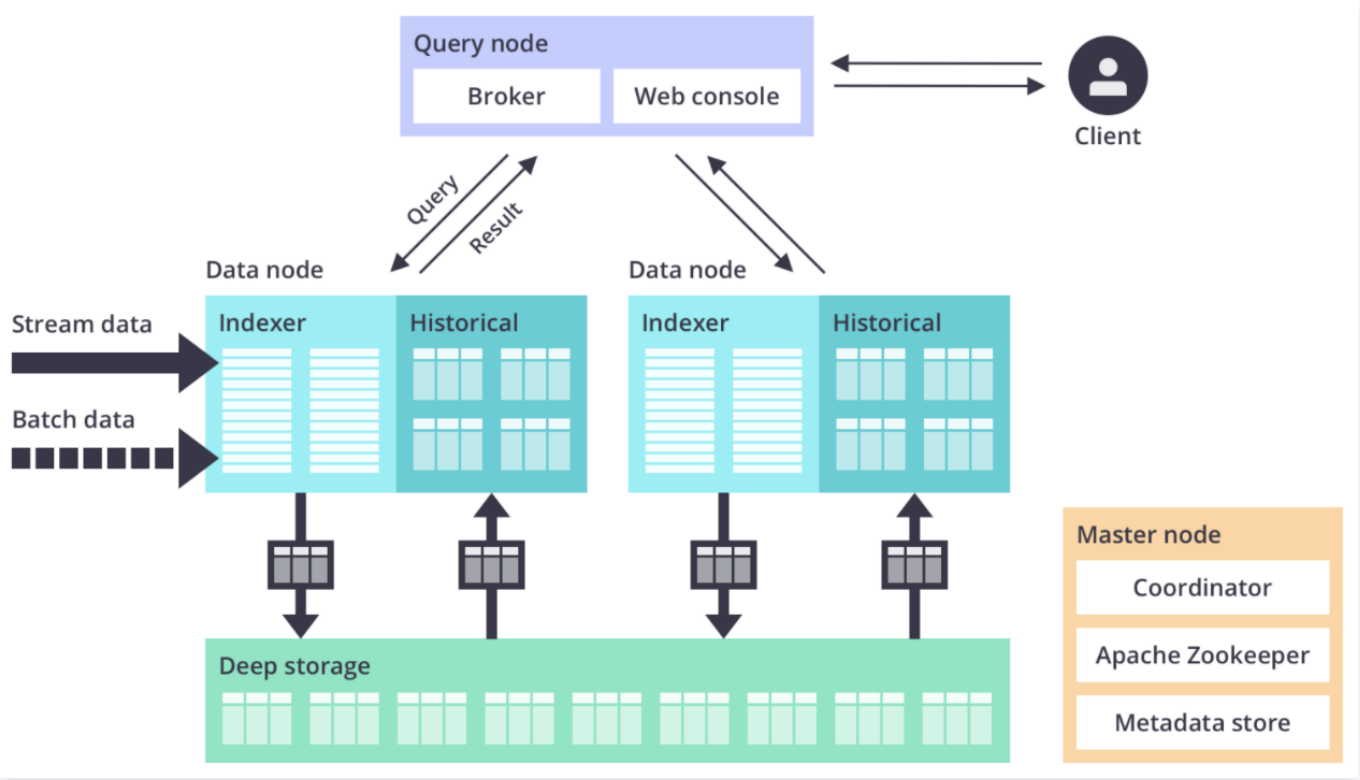
- 亚秒级响应的交互式查询，支持较高；包括多维过滤、Ad-hoc 的属性分组、快速聚合数据等。
- 支持高并发、实时数据摄入，真正做到数据摄入实时、查询结果实时。
- 扩展性强，采用分布式 shared-nothing 的架构，支持 PB 级数据、千亿级事件快速处理，支持每秒数千查询并发。
- 支持多租户同时在线查询。
- 支持高可用，且支持滚动升级。

应用场景

Druid 最常用的场景就是大数据背景下、灵活快速的多维 OLAP 分析。另外，Druid 还有一个关键的特点，它支持根据时间戳对数据进行预聚合摄入和聚合分析。因此也有用户经常在有时序数据处理分析的场景中用到它，例如广告平台、实时指标监控、推荐模型、搜索模型。

体系架构

Druid 是一个基于微服务的架构。Druid 中的每个核心服务均可以单独或联合部署在不同硬件上。



EMR 增强型 Druid

EMR Druid 基于 Apache Druid 做了大量的改进，包括与 EMR Hadoop 和云周边生态的集成、方便的监控与运维支持、易用的产品接口等，做到了即买即用和免运维。

EMR Druid 目前支持的特性如下所示：

- 方便和 EMR Hadoop 集群结合
- 方便快速弹性扩缩容
- 支持 HA
- 支持将 COS 作为 deep storage
- 支持将 COS 文件作为批量索引的数据源

- 支持将元数据存储到 TencentDB
- 集成了 Superset 等工具
- 丰富的监控指标和告警规则
- 故障迁移
- 高安全性

Druid 使用

最近更新时间：2023-12-20 17:06:31

EMR 支持将 E-MapReduce Druid 集群作为单独的集群类型，主要基于以下几方面的考虑：

- 使用场景：E-MapReduce Druid 可以脱离 Hadoop 使用来适配不同的业务应用场景。
- 资源抢占：E-MapReduce Druid 对内存要求比较高，尤其是 Broker节点和 Historical 节点。E-MapReduce Druid 本身资源使用不受 Hadoop YARN 统一调度，运行时容易发生资源抢夺。
- 集群规模：Hadoop 作为基础设施，其规模一般较大，而 E-MapReduce Druid 集群可能相对较小，部署在同一集群上，由于规模不一致可能造成资源浪费，单独部署则更加灵活。

购买建议

在创建 EMR 集群时选择 Druid 集群类型即可。Druid 集群自带了Hadoop HDFS 和 YARN 服务，并已经和 Druid 集群完成集成。但是建议仅用于测试，**对于线上生产环境，强烈推荐您采用专门的 Hadoop 集群**。
如果需要关闭 Druid 集群自带的 Hadoop 相关服务，可以在 EMR 控制台 的集群服务页，选择对应服务卡片，单击**操作 > 暂停服务**对服务进行暂停。

Hadoop 和 Druid 集群连通配置

本节介绍如何配置 Hadoop 集群和 Druid 集群的连通性。如果您使用 Druid 集群自带的 Hadoop 集群（生产环境不推荐这么做），则无须做额外设置即可正常连通使用，可以跳过此节。
如果您需要将索引数据存放在另外一个单独的 Hadoop 集群的 HDFS 上（生产环境推荐这种方式），则首先需要设置两个集群的连通性。具体步骤如下：

1. 确保 Druid 集群和 Hadoop 集群能够正常通信。
   两个集群在同一个 VPC 下，或两个集群在不同 VPC，但两个 VPC 之间能够正常通信（如通过云联网或者对等连接）。
2. 在 E-MapReduce Druid 集群的每个节点的 `/usr/local/service/druid/conf/druid/_common` 路径下，放置一份 Hadoop 集群中 `/usr/local/service/hadoop/etc/hadoop` 路径下的 core-site.xml、hdfs-site.xml、yarn-site.xml、mapred-site.xml 文件。

   > **注意**
   > Druid 集群由于自带 Hadoop 集群，因此 Druid 路径下已经提前创建了上述文件的相关软链接，需要先删除，再拷贝另一个 Hadoop 集群的配置过来。同时需要确保文件权限正确，能被 hadoop 用户正常访问。

3. 在 Druid 配置管理中修改 common.runtime.properties 配置文件。修改完成后，保存配置并重启 Druid 集群相关服务。

- druid.storage.type：默认为 hdfs，无需修改。
- druid.storage.storageDirectory：

```
如果另一个 Hadoop 集群为非 HA：hdfs://{namenode_ip}:4007
如果另一个 Hadoop 集群为 HA：hdfs://HDFSXXXXX
请配置全路径，详细地址可以在目标 Hadoop 集群的 core-site.xml 文件的 fs.defaultFS 配置项里面找到。
```

使用 COS

E-MapReduce Druid 支持以 COS 作为 deep storage，本节介绍如何使用 COS 作为 Druid 集群的 deep storage。
首先您需要确保 Druid 集群和目标 Hadoop 均开启了 COS 服务，可以在购买 Druid 集群和 Hadoop 集群时开启，也可以购买后在 EMR 控制台进行后配置 COS。

1. 在 Druid 配置管理中修改 common.runtime.properties 配置文件：
   - druid.storage.type：仍然为 hdfs。
   - druid.storage.storageDirectory：`cosn://{bucket_name}/druid/segments` 。
     可以到 COS 上预先创建并设置 segments 目录和权限。
2. 在 hdfs 配置管理中修改 core-site.xml 配置文件：
   - fs.cosn.impl：修改为 `org.apache.hadoop.fs.CosFileSystem` 。
   - 新增配置项 `fs.AbstractFileSystem.cosn.impl`：修改为 `org.apache.hadoop.fs.CosN` 。
3. 将 hadoop-cos 相关的 jar 包（例如：cos_api-bundle-5.6.69.jar、hadoop-cos-2.8.5-8.1.6.jar）放到集群各个节点的/usr/local/service/druid/extensions/druid-hdfs-storage、/usr/local/service/druid/hadoopdependencies/hadoop-client/2.8.5、/usr/local/service/hadoop/share/hadoop/common/lib/ 录下。

保存配置并重启 Druid 集群相关服务。

调整 Druid 参数

E-MapReduce Druid 在创建集群后会自动生成一套配置，不过建议您根据业务需求调整最优内存配置。要调整配置，您可以通过 配置管理 功能进行操作。
调整配置时，请确保调整正确：

```
MaxDirectMemorySize >= druid.processing.buffer.sizeByte *(druid.processing.numMergeBuffers +
druid.processing.numThreads + 1)
```

调整建议：

```
druid.processing.numMergeBuffers = max(2, druid.processing.numThreads / 4)
druid.processing.numThreads =  Number of cores - 1 (or 1)
druid.server.http.numThreads = max(10, (Number of cores * 17) / 16 + 2) + 30
```

更多配置请参考 Druid 组件配置 。

扩展 Router 作为查询节点

当前 Druid 集群默认部署 Broker 进程在 EMR Master 节点上，由于 Master 节点部署较多进程，进程之间影响可能出现内存不够的情况，影响查询效率；同时许多业务也希望查询节点和中心节点分离部署。在这些情况下您可以在控制台扩容一到多个 Router 节点并选择安装 Broker 进程，可以方便的扩展 Druid 集群的查询节点。

访问 Web

统一通过 console 访问 Druid 集群，端口开在主节点的18888端口，可以自行配置公网 IP，在安全组中开通18888的端口并设置带宽后，即可通过 `[http://{masterIp}:18888]()` 访问。

从 Hadoop 批量摄入数据

最近更新时间：2024-10-11 15:17:11

本节简单介绍如何从远程 Hadoop 集群中批量加载数据文件到 Druid 集群中。本文操作均是以 Hadoop 用户进行，请先在 Druid 集群和 Hadoop 集群上都切换到 Hadoop 用户。

批量加载数据到 Druid 集群

1. 在对应远程 Hadoop 集群上，以 Hadoop 用户执行以下新建目录命令：

```
hdfs dfs -mkdir /druid
hdfs dfs -mkdir /druid/segments
hdfs dfs -mkdir /quickstart
hdfs dfs -chmod 777 /druid
hdfs dfs -chmod 777 /druid/segments
hdfs dfs -chmod 777 /quickstart
```

> **注意**
> 如果 Druid 集群和 Hadoop 集群是两个独立集群，则目录需要建立在对应 Hadoop 集群上（之后的操作类似，注意分辨正确操作对应的集群）；如果在测试环境下 Druid 集群和 Hadoop 集群是同一个集群，则在同集群操作即可。

2. 上传测试包
Druid 集群下自带一个名为 Wikiticker 的数据集示例（默认路径 `/usr/local/service/druid/quickstart/tutorial/wikiticker-2015-09-12-sampled.json.gz`），将 Druid 集群内的数据集上传到对应远程 Hadoop 集群，**是在远程 Hadoop 集群上传**。

```
hdfs dfs -put wikiticker-2015-09-12-sampled.json.gz /quickstart/wikiticker-2015-09-12-sampled.json.gz
```

3. 编译索引文件
准备一个索引文件，仍然使用 Druid 集群的样例文件 `/usr/local/service/druid/quickstart/tutorial/wikipedia-index-hadoop.json`，命令如下：

```
{
  "type" : "index_hadoop",
  "spec" : {
 "dataSchema" : {
   "dataSource" : "wikipedia",
   "parser" : {
     "type" : "hadoopyString",
     "parseSpec" : {
       "format" : "json",
       "dimensionsSpec" : {
         "dimensions" : [
           "channel",
           "cityName",
           "comment",
           "countryIsoCode",
           "countryName",
           "isAnonymous",
           "isMinor",
           "isNew",
           "isRobot",
           "isUnpatrolled",
           "metroCode",
           "namespace",
           "page",
           "regionIsoCode",
           "regionName",
```

```
            "user",
            { "name": "added", "type": "long" },
            { "name": "deleted", "type": "long" },
            { "name": "delta", "type": "long" }
          ]
        },
        "timestampSpec" : {
          "format" : "auto",
          "column" : "time"
        }
      }
    },
    "metricsSpec" : [],
    "granularitySpec" : {
      "type" : "uniform",
      "segmentGranularity" : "day",
      "queryGranularity" : "none",
      "intervals" : ["2015-09-12/2015-09-13"],
      "rollup" : false
    }
  },
  "ioConfig" : {
    "type" : "hadoop",
    "inputSpec" : {
      "type" : "static",
      "paths" : "/quickstart/wikiticker-2015-09-12-sampled.json.gz"
    }
  },
  "tuningConfig" : {
    "type" : "hadoop",
    "partitionsSpec" : {
      "type" : "hashed",
      "targetPartitionSize" : 5000000
    },
    "forceExtendableShardSpecs" : true,
    "jobProperties" : {
      "yarn.nodemanager.vmem-check-enabled" : "false",
      "mapreduce.map.java.opts" : "-Duser.timezone=UTC -Dfile.encoding=UTF-8",
      "mapreduce.job.user.classpath.first" : "true",
      "mapreduce.reduce.java.opts" : "-Duser.timezone=UTC -Dfile.encoding=UTF-8",
      "mapreduce.map.memory.mb" : 1024,
      "mapreduce.reduce.memory.mb" : 1024
    }
  }
   },
   "hadoopDependencyCoordinates": ["org.apache.hadoop:hadoop-client:2.8.5"]
}
```

说明：

- hadoopDependencyCoordinates 为依赖的 Hadoop 版本。
- spec.ioConfig.inputSpec.paths 为输入文件路径。如果已经在 common.runtime.properties 配置中设置好集群连通性，可以使用相对路径（可参考 Druid 使用）。否则，应该根据情况使用以 `hdfs://` 或者 `cosn://` 开头的相对路径。
- tuningConfig.jobProperties 参数可以设置 mapreduce job 的相关参数。

4. 提交索引任务

   接下来可以在 Druid 集群上提交任务将数据摄入，在 Druid 目录下以 Hadoop 用户执行：

```
./bin/post-index-task --file quickstart/tutorial/wikipedia-index-hadoop.json --url http://localhost:8090
```

成功后则会有如下类似的输出：

```
...
Task finished with status: SUCCESS
Completed indexing data for wikipedia. Now loading indexed data onto the cluster...
wikipedia loading complete! You may now query your data
```

数据查询

Druid 支持类 SQL 和原生 JSON 查询，下面将分别介绍，更多内容可参考 官方文档 。

sql 方式查询

Druid 支持多种 SQL 查询方式：

- 在 Web UI 的 Query 菜单中查询。

```
SELECT page, COUNT(*) AS Edits
FROM wikipedia
WHERE TIMESTAMP '2015-09-12 00:00:00' <= "__time" AND "__time" < TIMESTAMP '2015-09-13 00:00:00'
GROUP BY page
ORDER BY Edits DESC
LIMIT 10
```

- 在查询节点上使用命令行工具 `bin/dsql` 进行交互式查询。

```
[hadoop@172 druid]$ ./bin/dsql
Welcome to dsql, the command-line client for Druid SQL.
Connected to [http://localhost:8082/].
Type "\h" for help.
dsql> SELECT page, COUNT(*) AS Edits FROM wikipedia WHERE "__time" BETWEEN TIMESTAMP '2015-09-12 00:00:00'
AND TIMESTAMP '2015-09-13 00:00:00' GROUP BY page ORDER BY Edits DESC LIMIT 10;
┌──────────────────────────────────────────────────────────┬───────┐
| page                                                     | Edits |
├──────────────────────────────────────────────────────────┼───────┤
| Wikipedia:Vandalismusmeldung                             |    33 |
| User:Cyde/List of candidates for speedy deletion/Subpage |    28 |
| Jeremy Corbyn                                            |    27 |
| Wikipedia:Administrators' noticeboard/Incidents          |    21 |
| Flavia Pennetta                                          |    20 |
| Total Drama Presents: The Ridonculous Race               |    18 |
| User talk:Dudeperson176123                               |    18 |
| Wikipédia:Le Bistro/12 septembre 2015                    |    18 |
| Wikipedia:In the news/Candidates                         |    17 |
| Wikipedia:Requests for page protection                   |    17 |
└──────────────────────────────────────────────────────────┴───────┘
Retrieved 10 rows in 0.06s.
```

- 用 HTTP 服务查询 SQL。

```
curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/wikipedia-top-pages-sql.json
http://localhost:18888/druid/v2/sql
```

格式化后的输出结果：

```
[
  {
      "page":"Wikipedia:Vandalismusmeldung",
      "Edits":33
  },
  {
      "page":"User:Cyde/List of candidates for speedy deletion/Subpage",
      "Edits":28
```

```
  },
  {
      "page":"Jeremy Corbyn",
      "Edits":27
  },
  {
      "page":"Wikipedia:Administrators' noticeboard/Incidents",
      "Edits":21
  },
  {
      "page":"Flavia Pennetta",
      "Edits":20
  },
  {
      "page":"Total Drama Presents: The Ridonculous Race",
      "Edits":18
  },
  {
      "page":"User talk:Dudeperson176123",
      "Edits":18
  },
  {
      "page":"Wikipédia:Le Bistro/12 septembre 2015",
      "Edits":18
  },
  {
      "page":"Wikipedia:In the news/Candidates",
      "Edits":17
  },
  {
      "page":"Wikipedia:Requests for page protection",
      "Edits":17
  }
]
```

原生 JSON 查询

- 在 Web UI 上 Query 菜单直接输入 json 查询。

```
{
   "queryType" : "topN",
   "dataSource" : "wikipedia",
   "intervals" : ["2015-09-12/2015-09-13"],
   "granularity" : "all",
   "dimension" : "page",
   "metric" : "count",
   "threshold" : 10,
   "aggregations" : [
       {
          "type" : "count",
          "name" : "count"
       }
   ]
}
```

- 在查询节点上 druid 方式目录下用 HTTP 提交。

```
curl -X 'POST' -H 'Content-Type:application/json' -d @quickstart/tutorial/wikipedia-top-pages.json
http://localhost:18888/druid/v2?pretty
```

输出结果：

```
[ {
"timestamp" : "2015-09-12T00:46:58.771Z",
"result" : [ {
  "count" : 33,
  "page" : "Wikipedia:Vandalismusmeldung"
}, {
  "count" : 28,
  "page" : "User:Cyde/List of candidates for speedy deletion/Subpage"
}, {
  "count" : 27,
  "page" : "Jeremy Corbyn"
}, {
  "count" : 21,
  "page" : "Wikipedia:Administrators' noticeboard/Incidents"
}, {
  "count" : 20,
  "page" : "Flavia Pennetta"
}, {
  "count" : 18,
  "page" : "Total Drama Presents: The Ridonculous Race"
}, {
  "count" : 18,
  "page" : "User talk:Dudeperson176123"
}, {
  "count" : 18,
  "page" : "Wikipédia:Le Bistro/12 septembre 2015"
}, {
  "count" : 17,
  "page" : "Wikipedia:In the news/Candidates"
}, {
  "count" : 17,
  "page" : "Wikipedia:Requests for page protection"
} ]
} ]
```

从 Kafka 实时摄入数据

最近更新时间：2024-08-15 14:44:01

本文介绍如何使用 Apache Druid Kafka Indexing Service 实时消费 Kafka 数据。开始本节前，类似 Hadoop 集群，需要确保 Kafka 集群和 Druid 集群之间能够正常通信。

> **说明**
> - 两个集群在同一个 VPC 下，或两个集群在不同 VPC，但两个 VPC 之间能够正常通信（如通过云联网或者对等连接）。
> - 如有必要需要将 Kafka 集群的 Host 信息配置到 Druid 集群中。

命令行方式

1. 首先在 Kafka 集群启动 kafka broker。

```
./bin/kafka-server-start.sh config/server.properties
```

2. 创建一个kafka topic，名为 mytopic。

```
./bin/kafka-topics.sh --create --zookeeper {kafka_zk_ip}:2181 --replication-factor 1 --partitions 1 --topic
mytopic
输出：
Created topic "mytopic".
```

`{kafka_zk_ip}:2181` 为 kafka 集群的 zookeeper 地址。

3. 在 Druid 集群上准备一个数据描述文件 kafka-mytopic.json。

```
{
  "type": "kafka",
  "dataSchema": {
      "dataSource": "mytopic-kafka",
      "parser": {
          "type": "string",
          "parseSpec": {
              "timestampSpec": {
                  "column": "time",
                  "format": "auto"
              },
              "dimensionsSpec": {
                  "dimensions": ["url", "user"]
              },
              "format": "json"
          }
      },
      "granularitySpec": {
          "type": "uniform",
          "segmentGranularity": "hour",
          "queryGranularity": "none"
      },
      "metricsSpec": [{
              "type": "count",
              "name": "views"
          },
          {
              "name": "latencyMs",
              "type": "doubleSum",
              "fieldName": "latencyMs"
          }
      ]
```

```
  },
  "ioConfig": {
      "topic": "mytopic",
      "consumerProperties": {
          "bootstrap.servers": "{kafka_ip}:9092",
          "group.id": "kafka-indexing-service"
      },
      "taskCount": 1,
      "replicas": 1,
      "taskDuration": "PT1H"
  },
  "tuningConfig": {
      "type": "kafka",
      "maxRowsInMemory": "100000"
  }
 }
```

`{kafka_ip}:9092` 为您 Kafka 集群的 bootstrap.servers IP 和端口。

4. 在 Druid 集群的 Master 节点上添加 Kafka supervisor。

```
curl -XPOST -H 'Content-Type: application/json' -d @kafka-mytopic.json
http://{druid_master_ip}:8090/druid/indexer/v1/supervisor
输出:
{"id":"mytopic-kafka"}
```

`{druid_master_ip}:8090` 为 overlord 进程部署的节点，一般是 Master 节点。

5. 在 Kafka 集群上开启一个 console producer。

```
./bin/kafka-console-producer.sh --broker-list {kafka_ip}:9092 --topic mytopic
```

`{kafka_ip}:9092` 为您 Kafka 集群的 bootstrap.servers IP 和端口。

6. 在 druid 集群准备一个查询文件，命名为 query-mytopic.json。

```
{
  "queryType" : "search",
  "dataSource" : "mytopic-kafka",
  "intervals" : ["2020-03-13T00:00:00.000/2020-03-20T00:00:00.000"],
  "granularity" : "all",
  "searchDimensions": [
      "url",
      "user"
  ],
  "query": {
      "type": "insensitive_contains",
      "value": "roni"
  }
 }
```

7. 在 Kafka 上实时输入一些数据。

```
{"time": "2020-03-19T09:57:58Z", "url": "/foo/bar", "user": "brozo", "latencyMs": 62}
{"time": "2020-03-19T16:57:59Z", "url": "/", "user": "roni", "latencyMs": 15}
{"time": "2020-03-19T17:50:00Z", "url": "/foo/bar", "user": "roni", "latencyMs": 25}
```

时间戳生成命令:

```
python -c 'import datetime; print(datetime.datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%SZ"))'
```

8. 在 Druid 集群上查询。

```
curl -XPOST -H 'Content-Type: application/json' -d @query-mytopic.json http://{druid_ip}:8082/druid/v2/?
pretty
```

`{druid_ip}:8082` 为您 Druid 集群的 broker 节点，一般在 Master 或 Router 节点上。

查询结果：

```
[ {
  "timestamp" : "2020-03-19T16:00:00.000Z",
  "result" : [ {
 "dimension" : "user",
 "value" : "roni",
 "count" : 2
  } ]
} ]
```

Web 可视化方式

您可通过 Druid Web UI 控制台可视化方式，从 Kafka 集群摄入数据并查询，详细可参考 通过 data loader 加载 Kafka 数据 。

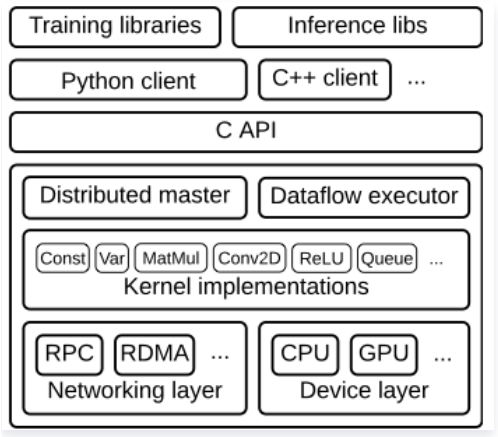
Tensorflow 开发指南
TensorFlow 简介

最近更新时间：2024-08-23 14:52:21

TensorFlow 是一个端到端开源机器学习平台。它拥有一个全面而灵活的生态系统，其中包含各种工具、库和社区资源，可助力研究人员推动先进机器学习技术的发展，并使开发者能够轻松地构建和部署由机器学习提供支持的应用。

- 轻松地构建模型
  在即刻执行环境中使用 Keras 等直观的高阶 API 轻松地构建和训练机器学习模型，此环境使我们能够快速迭代模型并轻松地调试模型。
- 随时随地进行可靠的机器学习生产
  无论您使用哪种语言，都可以在云端、本地、浏览器中或设备上轻松地训练和部署模型。
- 强大的研究实验
  一个简单而灵活的架构，可以更快地将新想法从概念转化为代码，然后创建出先进的模型，并最终对外发布。

TensorFlow 架构



- 客户端（Client）
  将计算过程定义为数据流图。使用 `_Session_` 初始化数据流图的执行。
- 分布式主控端（Master）
  修剪图中的某些特殊子图，即 `Session.run()` 中所定义的参数。将子图划分为在不同进程和设备中运行的多个部分。将图分发给不同的工作进程。由工作进程初始化子图的计算。
- 工作进程（Worker service）（每个任务的）
  使用内核实现调度图操作并在合适的硬件（CPU、GPU 等）执行。向其他工作进程发送或从其接收操作的结果。
- 内核实现
  执行一个独立的图操作计算。

EMR 支持 TensorFlow

- TensorFlow 版本：v1.14.0
- 目前 TensorFlow 只支持运行在 CPU 机型，暂不支持 GPU 机型。
- 支持 tensorflow on spark 做分布式训练

TensorFlow 开发示例

本文以 TensorFlow v1.4.4 版本为示例，首先需要安装 TensorFlow，切换到 root 用户下，密码为创建 EMR 集群时设置的密码，先安装 python-pip 工具再安装依赖包：

```
[hadoop@172 hbase]$ su
Password: ********
[root@172 hbase]# yum install python-pip
[root@172 hbase]# pip install Tensorflow
```

编写代码：`test.py`

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print sess.run(hello)
a = tf.constant(10)
b = tf.constant(111)
print sess.run(a+b)
exit()
```

执行如下命令：

```
python test.py
```

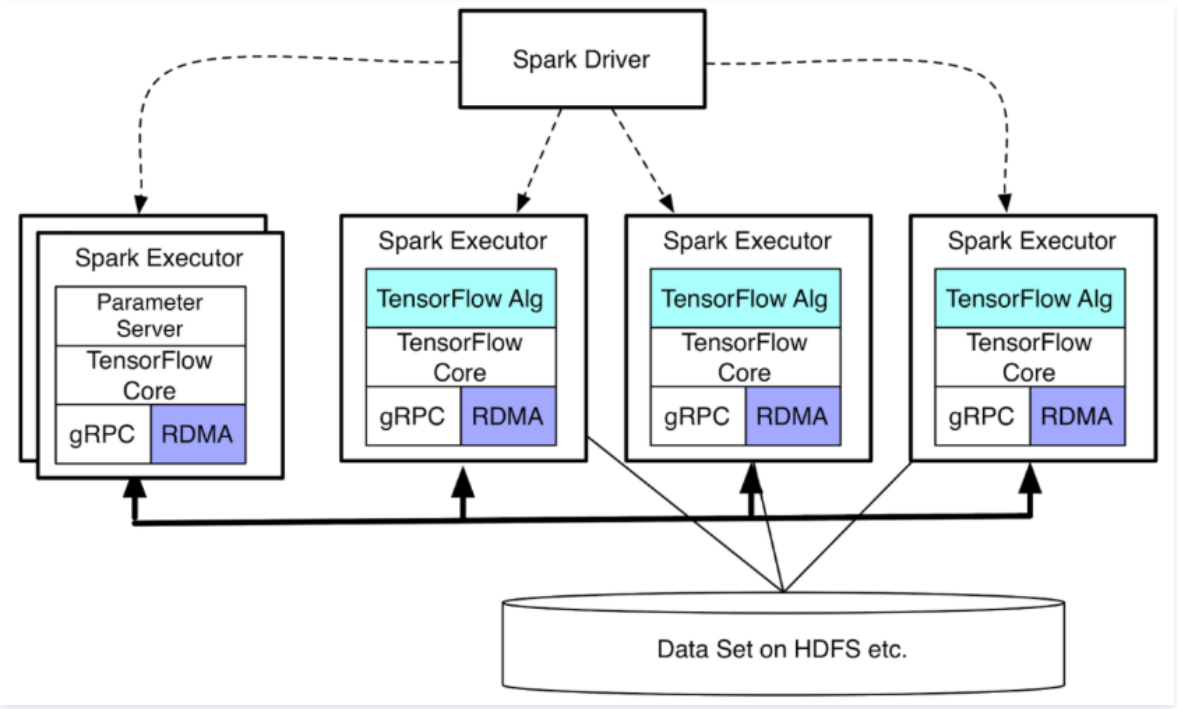
更多用法请参考 TensorFlow 官网 。

TensorFlowOnSpark 简介

最近更新时间：2023-07-11 21:43:16

TensorFlowOnSpark 为 Apache Hadoop 和 Apache Spark 集群提供了可扩展的深度学习，TensorFlowOnSpark 支持所有类型的 TensorFlow 程序，可以实现异步/同步的训练和推理，同时也支持模型并行性和数据的并行处理。详情可查阅 TensorFlowOnSpark 官网 。

TensorFlowOnSpark 架构图



TensorFlowOnSpark 支持 TensorFlow 进程（计算节点和参数服务节点）之间的直接张量通信。过程到过程的直接通信机制使 TensorFlowOnSpark 程序能够在增加的机器上很轻松的进行扩展。TensorFlowOnSpark 不涉及张量通信中的 Spark 驱动程序，因此实现了与独立 TensorFlow 集群类似的可扩展性。

安装 TensorFlowOnSpark

1. 进入 EMR 购买页，选择产品 EMR-2.x.x 版本。
2. 在**可选组件**列表中，勾选 tensorflowonspark 组件。
3. tensorflowonspark 默认安装在 `/usr/local/service/tensorflowonspark` 目录下。

> **注意**
> tensorflowonspark 依赖的组件包含 hive 和 spark，在 tensorflowonspark 的同时也需要安装 hive 和 spark 组件。

使用示例

在安装好的 tensorflowonspark 组件目录下，已经有完整的 example 代码，**本文以 tensorflowonspark 1.4.4**，可以按如下操作步骤：

- **下载测试数据**
  使用 hadoop 用户，在 /usr/local/service/tensorflow-on-spark 目录下，执行命令：

```
sh mnist_download.sh
cat mnist_download.sh
```

```
[hadoop@172 /usr/local/service/tensorflow-on-spark]$ cat mnist_download.sh
mkdir ${HOME}/mnist
pushd ${HOME}/mnist >/dev/null
curl -O "http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz"
zip -r mnist.zip *
popd >/dev/null
```

- **上传原始数据和依赖包**

```
hdfs dfs -mkdir -p /mnist/tools/
hdfs dfs -put ~/mnist/mnist.zip /mnist/tools
hdfs dfs -mkdir /tensorflow
hdfs dfs -put TensorFlowOnSpark/tensorflow-hadoop-1.10.0.jar /tensorflow
```

- **特征数据准备**

```
sh prepare_mnist.sh
```

可以看到特征数据已准备就绪：

```
hdfs dfs -ls /user/hadoop/mnist
Found 2 items
drwxr-xr-x - hadoop supergroup 0 2020-05-21 11:40 /user/hadoop/mnist/csv
drwxr-xr-x - hadoop supergroup 0 2020-05-21 11:41 /user/hadoop/mnist/tfr
```

- **基于 InputMode.SPARK 模型训练**

```
sh mnist_train_with_spark_cpu.sh
```

查看模型训练好的模型：

```
[hadoop@10 tensorflow-on-spark]$ hdfs dfs -ls /user/hadoop/mnist_model
Found 10 items
-rw-r--r-- 1 hadoop supergroup 128 2020-05-21 11:46 /user/hadoop/mnist_model/checkpoint
-rw-r--r-- 1 hadoop supergroup 243332 2020-05-21 11:46
/user/hadoop/mnist_model/events.out.tfevents.1590032704.10.0.0.114
-rw-r--r-- 1 hadoop supergroup 164619 2020-05-21 11:45 /user/hadoop/mnist_model/graph.pbtxt
-rw-r--r-- 1 hadoop supergroup 814168 2020-05-21 11:45 /user/hadoop/mnist_model/model.ckpt-0.data-00000-of-
00001
-rw-r--r-- 1 hadoop supergroup 375 2020-05-21 11:45 /user/hadoop/mnist_model/model.ckpt-0.index
-rw-r--r-- 1 hadoop supergroup 64658 2020-05-21 11:45 /user/hadoop/mnist_model/model.ckpt-0.meta
-rw-r--r-- 1 hadoop supergroup 814168 2020-05-21 11:46 /user/hadoop/mnist_model/model.ckpt-595.data-00000-
of-00001
-rw-r--r-- 1 hadoop supergroup 375 2020-05-21 11:46 /user/hadoop/mnist_model/model.ckpt-595.index
-rw-r--r-- 1 hadoop supergroup 64658 2020-05-21 11:46 /user/hadoop/mnist_model/model.ckpt-595.meta
drwxr-xr-x - hadoop supergroup 0 2020-05-21 11:46 /user/hadoop/mnist_model/train
```

- **基于 InputMode.SPARK 模型预测**

```
sh mnist_inference_with_spark_cpu.sh
```

查看预测结果：

```
hdfs dfs -cat /user/hadoop/predictions/part-00000 |more
2020-05-21T11:49:56.561506 Label: 7, Prediction: 7
2020-05-21T11:49:56.561535 Label: 2, Prediction: 2
2020-05-21T11:49:56.561541 Label: 1, Prediction: 1
2020-05-21T11:49:56.561545 Label: 0, Prediction: 0
2020-05-21T11:49:56.561550 Label: 4, Prediction: 4
2020-05-21T11:49:56.561555 Label: 1, Prediction: 1
2020-05-21T11:49:56.561559 Label: 4, Prediction: 4
2020-05-21T11:49:56.561564 Label: 9, Prediction: 9
2020-05-21T11:49:56.561568 Label: 5, Prediction: 6
2020-05-21T11:49:56.561573 Label: 9, Prediction: 9
2020-05-21T11:49:56.561578 Label: 0, Prediction: 0
2020-05-21T11:49:56.561582 Label: 6, Prediction: 6
2020-05-21T11:49:56.561587 Label: 9, Prediction: 9
2020-05-21T11:49:56.561603 Label: 0, Prediction: 0
2020-05-21T11:49:56.561608 Label: 1, Prediction: 1
2020-05-21T11:49:56.561612 Label: 5, Prediction: 5
```

- **基于 InputMode.TENSORFLOW 训练模型**

```
sh mnist_train_with_tf_cpu.sh
```

查看模型：

```
hdfs dfs -ls mnist_model
Found 25 items
-rw-r--r-- 1 hadoop supergroup 265 2020-05-21 14:58 mnist_model/checkpoint
-rw-r--r-- 1 hadoop supergroup 40 2020-05-21 14:53 mnist_model/events.out.tfevents.1590044017.10.0.0.144
-rw-r--r-- 1 hadoop supergroup 40 2020-05-21 14:57 mnist_model/events.out.tfevents.1590044221.10.0.0.144
-rw-r--r-- 1 hadoop supergroup 40 2020-05-21 14:57 mnist_model/events.out.tfevents.1590044227.10.0.0.144
-rw-r--r-- 1 hadoop supergroup 40 2020-05-21 14:57 mnist_model/events.out.tfevents.1590044232.10.0.0.144
-rw-r--r-- 1 hadoop supergroup 40 2020-05-21 14:57 mnist_model/events.out.tfevents.1590044238.10.0.0.144
-rw-r--r-- 1 hadoop supergroup 40 2020-05-21 14:58 mnist_model/events.out.tfevents.1590044303.10.0.0.114
-rw-r--r-- 1 hadoop supergroup 198078 2020-05-21 14:58 mnist_model/graph.pbtxt
drwxr-xr-x - hadoop supergroup 0 2020-05-21 14:58 mnist_model/inference
-rw-r--r-- 1 hadoop supergroup 814168 2020-05-21 14:57 mnist_model/model.ckpt-238.data-00000-of-00001
-rw-r--r-- 1 hadoop supergroup 375 2020-05-21 14:57 mnist_model/model.ckpt-238.index
-rw-r--r-- 1 hadoop supergroup 76255 2020-05-21 14:57 mnist_model/model.ckpt-238.meta
-rw-r--r-- 1 hadoop supergroup 814168 2020-05-21 14:57 mnist_model/model.ckpt-277.data-00000-of-00001
-rw-r--r-- 1 hadoop supergroup 375 2020-05-21 14:57 mnist_model/model.ckpt-277.index
-rw-r--r-- 1 hadoop supergroup 76255 2020-05-21 14:57 mnist_model/model.ckpt-277.meta
-rw-r--r-- 1 hadoop supergroup 814168 2020-05-21 14:57 mnist_model/model.ckpt-315.data-00000-of-00001
-rw-r--r-- 1 hadoop supergroup 375 2020-05-21 14:57 mnist_model/model.ckpt-315.index
-rw-r--r-- 1 hadoop supergroup 76255 2020-05-21 14:57 mnist_model/model.ckpt-315.meta
-rw-r--r-- 1 hadoop supergroup 814168 2020-05-21 14:57 mnist_model/model.ckpt-354.data-00000-of-00001
-rw-r--r-- 1 hadoop supergroup 375 2020-05-21 14:57 mnist_model/model.ckpt-354.index
-rw-r--r-- 1 hadoop supergroup 76255 2020-05-21 14:57 mnist_model/model.ckpt-354.meta
-rw-r--r-- 1 hadoop supergroup 814168 2020-05-21 14:58 mnist_model/model.ckpt-393.data-00000-of-00001
-rw-r--r-- 1 hadoop supergroup 375 2020-05-21 14:58 mnist_model/model.ckpt-393.index
-rw-r--r-- 1 hadoop supergroup 76255 2020-05-21 14:58 mnist_model/model.ckpt-393.meta
drwxr-xr-x - hadoop supergroup 0 2020-05-21 14:53 mnist_model/train
```

- **基于 InputMode.TENSORFLOW 模型预测**

```
sh mnist_train_with_tf_cpu.sh
```

查看预测结果：

```
hdfs dfs -cat predictions/part-00000 |more
9 4
9 9
4 4
1 1
4 4
8 8
9 9
2 2
3 5
6 6
9 9
2 2
6 6
0 0
7 7
5 5
3 3
```

Kudu 开发指南
Kudu 简介

最近更新时间：2024-08-23 14:52:21

Apache Kudu 是一个分布式，可水平扩展的列式存储系统，它完善了 Hadoop 的存储层，可对快速变化数据进行快速分析。

Kudu 基本特点

- 高效处理类 OLAP 负载。
- 与 MapReduce、Spark 以及 Hadoop 生态系统中其他组件进行友好集成。
- 可与 Impala 集成，替代目前 Impala 常用的 HDFS + Parquet 组合。
- 灵活的一致性模型。
- 顺序写和随机写并存的场景下，仍能达到良好的性能。
- 高可用，使用 Raft 协议保证数据高可靠存储。
- 结构化数据模型。

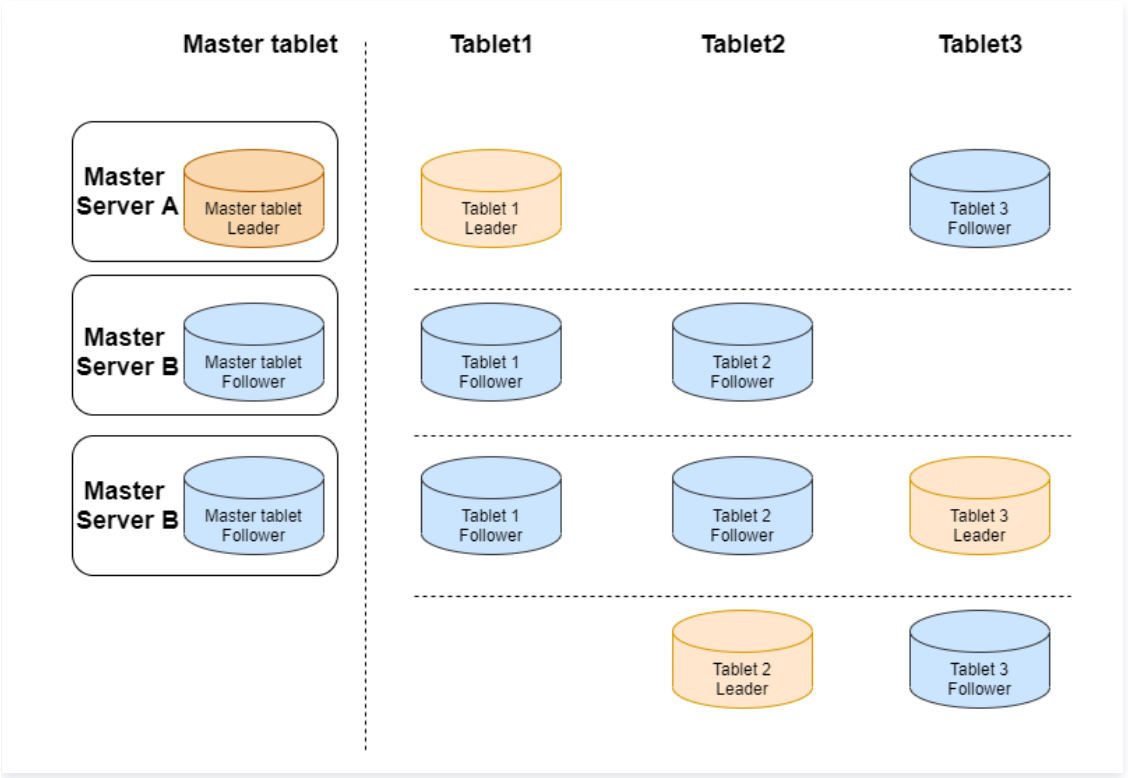
Kudu 使用场景

- 适用于那些既有随机访问，也有批量数据扫描的复合场景。
- 高计算量的场景。
- 实时预测模型的应用，支持根据所有历史数据周期地更新模型。
- 支持数据更新，避免数据反复迁移。
- 支持跨地域的实时数据备份和查询。

Kudu 基本架构

Kudu 包含如下两种类型的组件：

- master 主要负责管理元数据信息、监听 server，当 server 宕机后负责 tablet 的重分配。
- tserver 主要负责 tablet 的存储与数据的增删改查。



Kudu 使用

EMR-2.4.0版本以上支持了 Kudu 组件。在创建 Hadoop 集群时勾选 Kudu 组件，即会创建 Kudu 集群。默认情况下 Kudu 集群包含3个 Kudu Master 服务并开启 HA。

> **说明**
> 以下 Kudumaster_ip1、Kudumaster_ip2、Kudumaster_ip3 所用到的 IP 为 KuduMaster 角色所在节点内网 IP。

- Impala 与 Kudu 集成
  参考 Impala 简介 内容进入到 Impala 命令行，执行以下命令新建表格：

```
CREATE TABLE t2(id BIGINT,name STRING,PRIMARY KEY(id))PARTITION BY HASH PARTITIONS 2 STORED AS KUDU
TBLPROPERTIES (
'kudu.master_addresses' = '$Kudumaster_ip1,$Kudumaster_ip2,$Kudumaster_ip3',
'kudu.num_tablet_replicas' = '1');
```

  成功后返回以下提示信息：

```
Query: create TABLE t2 (id BIGINT,name STRING,PRIMARY KEY(id)) PARTITION BY HASH PARTITIONS 2 STORED AS
KUDU TBLPROPERTIES (
'kudu.master_addresses' = '$Kudumaster_ip1,$Kudumaster_ip2,$Kudumaster_ip3',
'kudu.num_tablet_replicas' = '1')
Fetched 0 row(s) in 0.12s
```

  输入以下命令可查看已创建表格：

```
usr/local/service/kudu/bin/kudu table list $Kudumaster_ip1,$Kudumaster_ip2,$Kudumaster_ip3
```

- 数据插入
  在 Impala 命令行，执行以下命令向表中插入数据：

```
insert into t2 values(1, 'test');
```

- 基于 Impala 查询数据
  在 Impala 命令行，执行以下命令查询表中数据：

```
[172.30.0.98:27001] > select * from t2;
```

  可查询到表中已插入数据
- 其他命令
- 集群健康检测

```
[hadoop@172 root]$ /usr/local/service/kudu/bin/kudu cluster ksck
172.30.0.240,172.30.1.167,172.30.0.96,172.30.0.94,172.30.0.214
```

- 创建表

```
 [hadoop@172 root]$ /usr/local/**service**/**kudu**/bin/kudu table create
'172.30.0.240,172.30.1.167,172.30.0.96,172.30.0.94,172.30.0.214' '{"table_name":"test","schema":{"columns":
[{"column_name":"id","column_type":"INT32","default_value":"1"},
{"column_name":"key","column_type":"INT64","is_nullable":false,"comment":"range key"},
{"column_name":"name","column_type":"STRING","is_nullable":false,"comment":"user
name"}],"key_column_names":["id","key"]},"partition":{"hash_partitions":[{"columns":
["id"],"num_buckets":2,"seed":100}],"range_partition":{"columns":["key"],"range_bounds":[{"upper_bound":
{"bound_type":"inclusive","bound_values":["2"]}},{"lower_bound":{"bound_type":"exclusive","bound_values":
["2"]},"upper_bound":{"bound_type":"inclusive","bound_values":["3"]}}]}},"extra_configs":{"configs":
{"kudu.table.history_max_age_sec":"3600"}},"num_replicas":1}'
```

- 查询创建的 test 表

```
[hadoop@172 root]$ /usr/local/service/kudu/bin/kudu table list
172.30.0.240,172.30.1.167,172.30.0.96,172.30.0.94,172.30.0.214
test
```

- 查看表结构

```
[hadoop@172 root]$ /usr/local/service/kudu/bin/kudu table describe
172.30.0.240,172.30.1.167,172.30.0.96,172.30.0.94,172.30.0.214 test
TABLE test (
 id INT32 NOT NULL,
 key INT64 NOT NULL,
 name STRING NOT NULL,
 PRIMARY KEY (id, key)
)
HASH (id) PARTITIONS 2 SEED 100,
RANGE (key) (
 PARTITION VALUES < 3,
 PARTITION 3 <= VALUES < 4
)
REPLICAS 1
```

Kudu 节点缩容数据搬迁指南

最近更新时间：2025-04-03 09:44:32

本文档主要对 Kudu 集群类型 Core 节点 tserver 缩容时，需进行的数据搬迁进行介绍。

> **说明**
> Kudu 集群类型支持 Core 节点缩容，该功能未默认开放，如有需要请您 提交工单 申请开通。

在进行 tserver 节点下线前，可以使用 rebalance tool 做数据迁移。请注意，一次只能下线一台 tserver，如果下线多台，需重复执行下述步骤。

Kudu 基于 rebalance tool 迁移

1. 确保集群状态 ok。

```
/usr/local/service/kudu/bin/kudu cluster ksck
$KuduMaster1_ip:port,$KuduMaster2_ip:port,$KuduMaster3_ip:port
```

其中 $KuduMaster1_ip、$KuduMaster2_ip、$KuduMaster3_ip 为 EMR 集群中3个 KuduMaster 角色所在节点的 IP；port 为链接 kudu 的端口号，默认为7051。

```
==================
Warnings:
==================
Some masters have unsafe, experimental, or hidden flags set
Some tablet servers have unsafe, experimental, or hidden flags set

OK
[hadoop@10 bin]$
```

2. 使用步骤1的 ksck 命令，获取下线的节点 uid。

```
Tablet Server Summary
              UUID              |    Address     | Status  | Location | Tablet Leaders | Active Scanners
--------------------------------+----------------+---------+----------+----------------+-----------------
 20681b1d6b9942cbab95dded905406ec | 10.0.1.37:7050 | HEALTHY | <none>   |       9        |        0
 6929daf14f8647c89fb8cc51db5d70b6 | 10.0.1.15:7050 | HEALTHY | <none>   |       5        |        0
 b53b28bfad2c41d38d6f08a261ceb486 | 10.0.1.40:7050 | HEALTHY | <none>   |       2        |        0
 be018287364d4443a48ad1bba248c87f | 10.0.1.9:7050  | HEALTHY | <none>   |       0        |        0
 fb9afb1b2989456cac5800bf6990dfea | 10.0.1.45:7050 | HEALTHY | <none>   |       0        |        0
```

以 fb9afb1b2989456cac5800bf6990dfea 节点为例子。

3. 将 fb9afb1b2989456cac5800bf6990dfea 节点进入维护模式。

```
/usr/local/service/kudu/bin/kudu tserver state enter_maintenance
$KuduMaster1_ip:port,$KuduMaster2_ip:port,$KuduMaster3_ip:port fb9afb1b2989456cac5800bf6990dfea
```

4. 执行 rebalance 命令

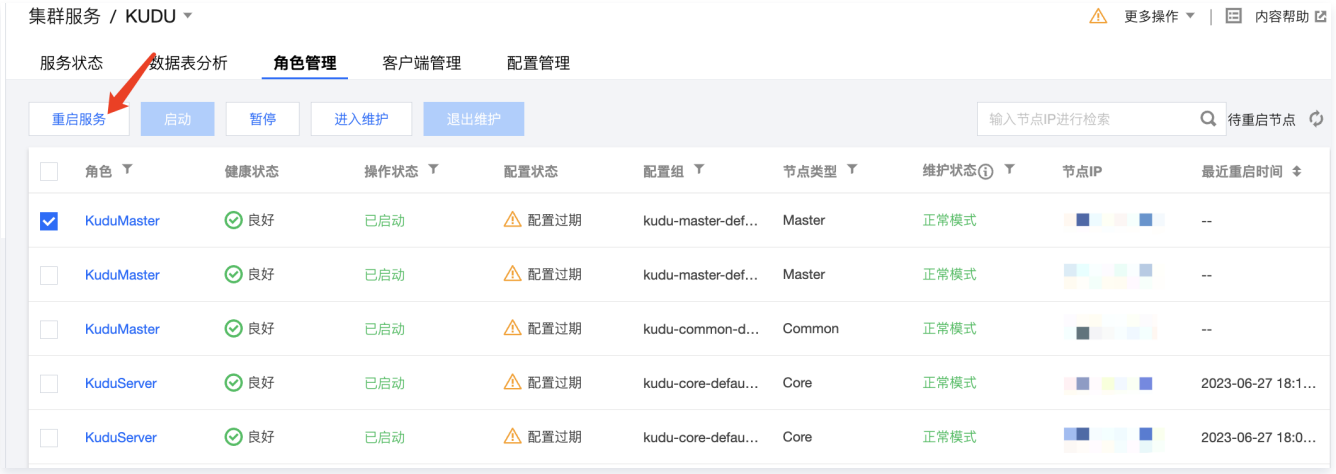
```
/usr/local/service/kudu/bin/kudu cluster rebalance
$KuduMaster1_ip:port,$KuduMaster2_ip:port,$KuduMaster3_ip:port --ignored_tservers
fb9afb1b2989456cac5800bf6990dfea --move_replicas_from_ignored_tservers
```

等待命令执行结束，再次用 ksck 检查，状态为 ok，继续后面步骤。

5. 暂停 fb9afb1b2989456cac5800bf6990dfea 对应节点10.0.1.45的 tserver 进程。注意此时，使用ksck命令，集群状态不健康，需要重启 master。

```
Tablet Server Summary
                UUID                |     Address     |   Status    | Location | Tablet Leaders | Active Scanners
------------------------------------+-----------------+-------------+----------+----------------+-----------------
 20681b1d6b9942cbab95dded905406ec   | 10.0.1.37:7050  | HEALTHY     | <none>   |       9        |        0
 6929daf14f8647c89fb8cc51db5d70b6   | 10.0.1.15:7050  | HEALTHY     | <none>   |       5        |        0
 b53b28bfad2c41d38d6f08a261ceb486   | 10.0.1.40:7050  | HEALTHY     | <none>   |       2        |        0
 be018287364d4443a48ad1bba248c87f   | 10.0.1.9:7050   | HEALTHY     | <none>   |       0        |        0
 fb9afb1b2989456cac5800bf6990dfea   | 10.0.1.45:7050  | UNAVAILABLE | <none>   | n/a            | n/a
```

6. 在 EMR 控制台 重启 master。注意需要手动一台一台的重启（不建议使用控制台的滚动重启）。重启结束后，使用ksck命令，确保集群状态健康。



Ranger 开发指南
Ranger 简介

最近更新时间：2024-04-18 11:26:41

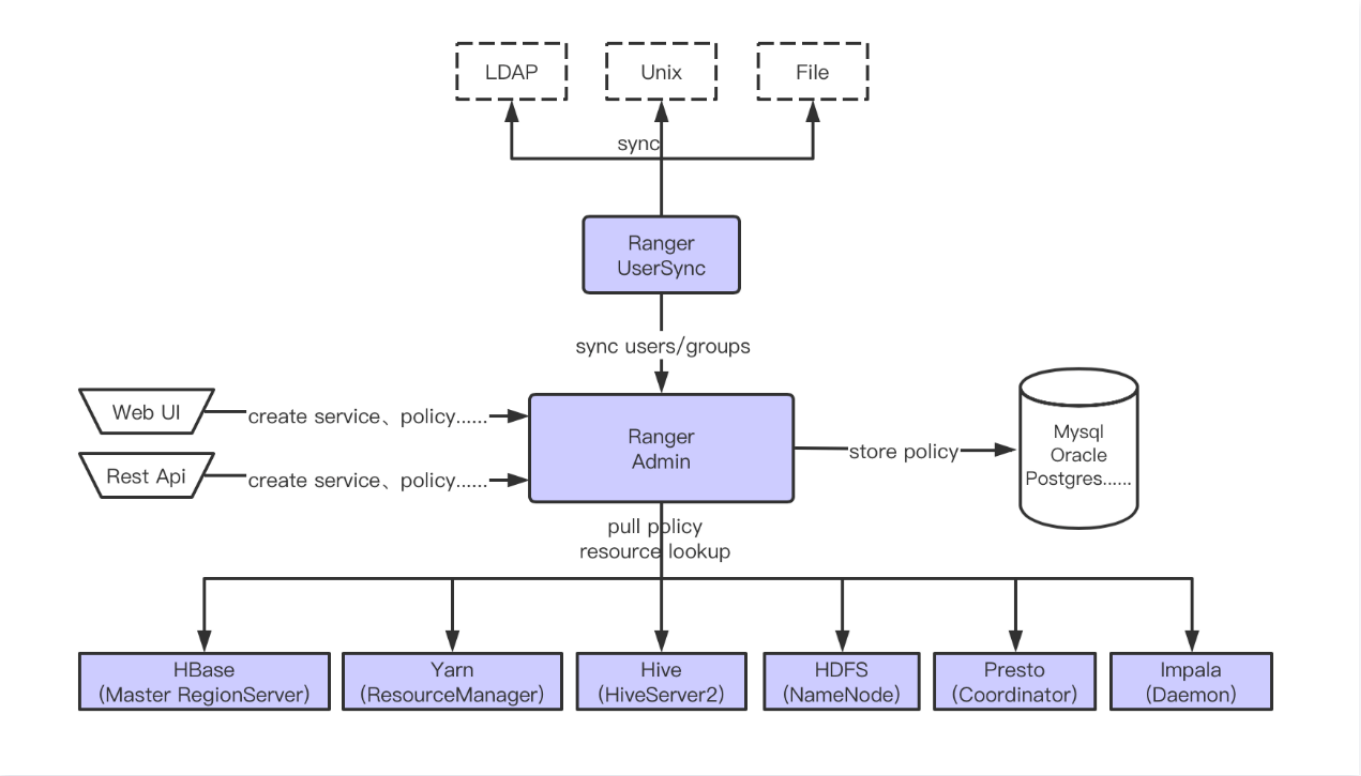
Ranger 简介

Ranger 是大数据领域的一个集中式安全管理框架，实现对 Hadoop 生态组件的集中式安全管理。用户可以通过 Ranger 实现对集群中数据的安全访问，它主要是对 Hadoop 平台组件进行监管、启动服务以及资源访问进行控制，其核心思想为：

- 用户可以使用 Ranger 提供的 REST API 或者使用 Ranger 提供 Web UI 对大数据组件进行集中化管理。
- 可以针对大数据组件进行基于角色、属性进行授权。
- 针对大数据组件所涉及安全的审计进行集中管理。

Ranger 架构

Ranger 主要是由 Ranger Admin、Ranger UserSync、Ranger Plugin 三个组件构成的，其中 Ranger Admin、Ranger UserSync 都是一个单独的 JVM 进程，而 Ranger Plugin 需要根据不同组件安装在不同节点上。



- Ranger Admin：管理用户配置好的策略及创建的服务、审计日志及 Report、将配置好的策略及创建的服务持久化到数据库中并提供给 Plugin 定期查询。
- Ranger UserSync：将 LDAP、File、Unix 的相关信息同步到 Ranger Admin 中，例如，将用户的 LDAP 目录访问系统或者 Unix 中用户的信息及组信息进行同步，同步 Unix 用户及组信息需要开启 unixAuthenticationService 进程，同时对同步过来的信息进行持久化。
- Ranger Plugin：Plugin 会被部署在需要的服务节点中，并会定期到 Ranger Admin 同步策略信息。

组件集成 Ranger 请参照如下表格：

Service	install Node	EMR 版本
HDFS	NameNode	EMR-V 2.0.1 及以上版本
Hbase	Master、RegionServer	EMR-V 2.0.1 及以上版本
Hive	HiveServer2	EMR-V 2.0.1 及以上版本
Yarn	ResourceManager	EMR-V 2.0.1 及以上版本
Presto	All Coordinator	EMR-V 2.0.1 及以上版本

Impala	All Daemon	EMR-V 2.2.0 及以上版本
Kudu	All Master	EMR-V 3.2.0 及以上版本

Ranger 使用指南

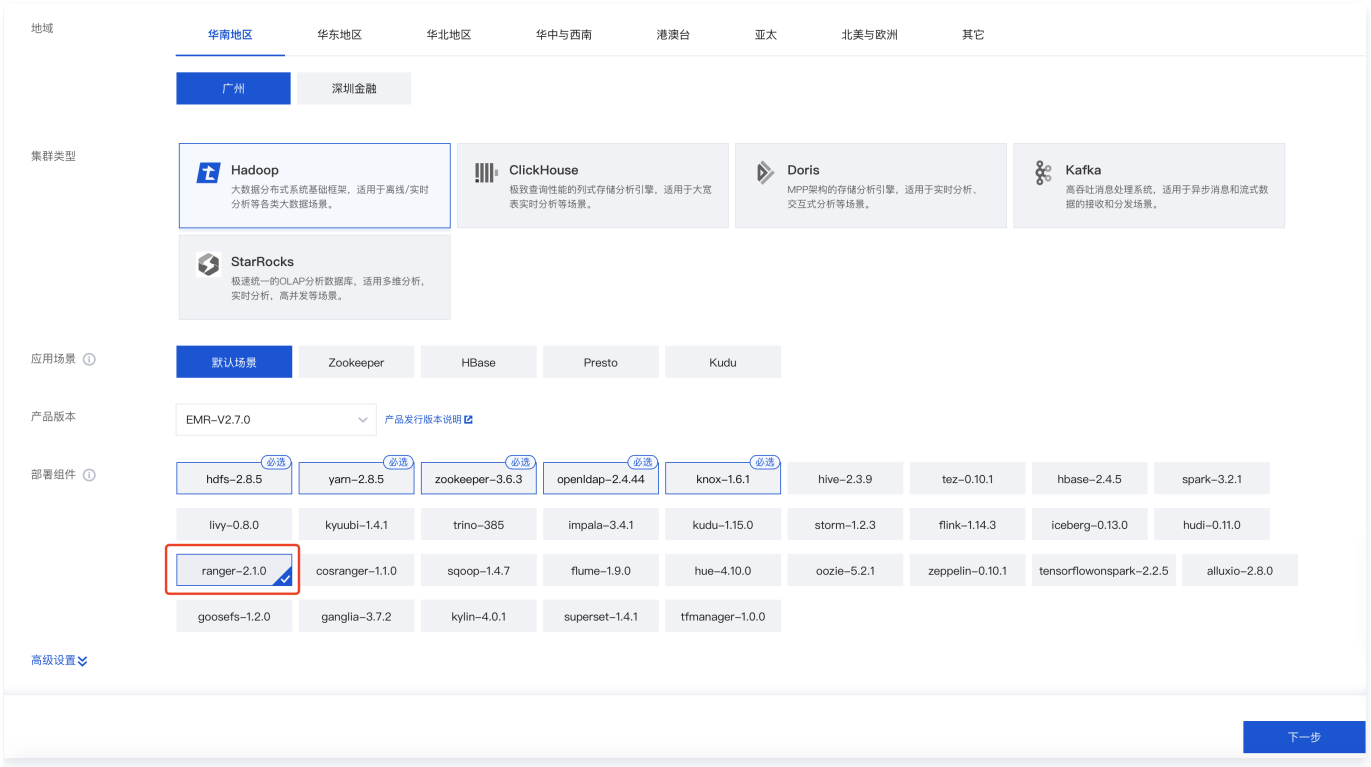
HDFS 集成 Ranger

最近更新时间：2023-06-16 14:44:57

使用准备

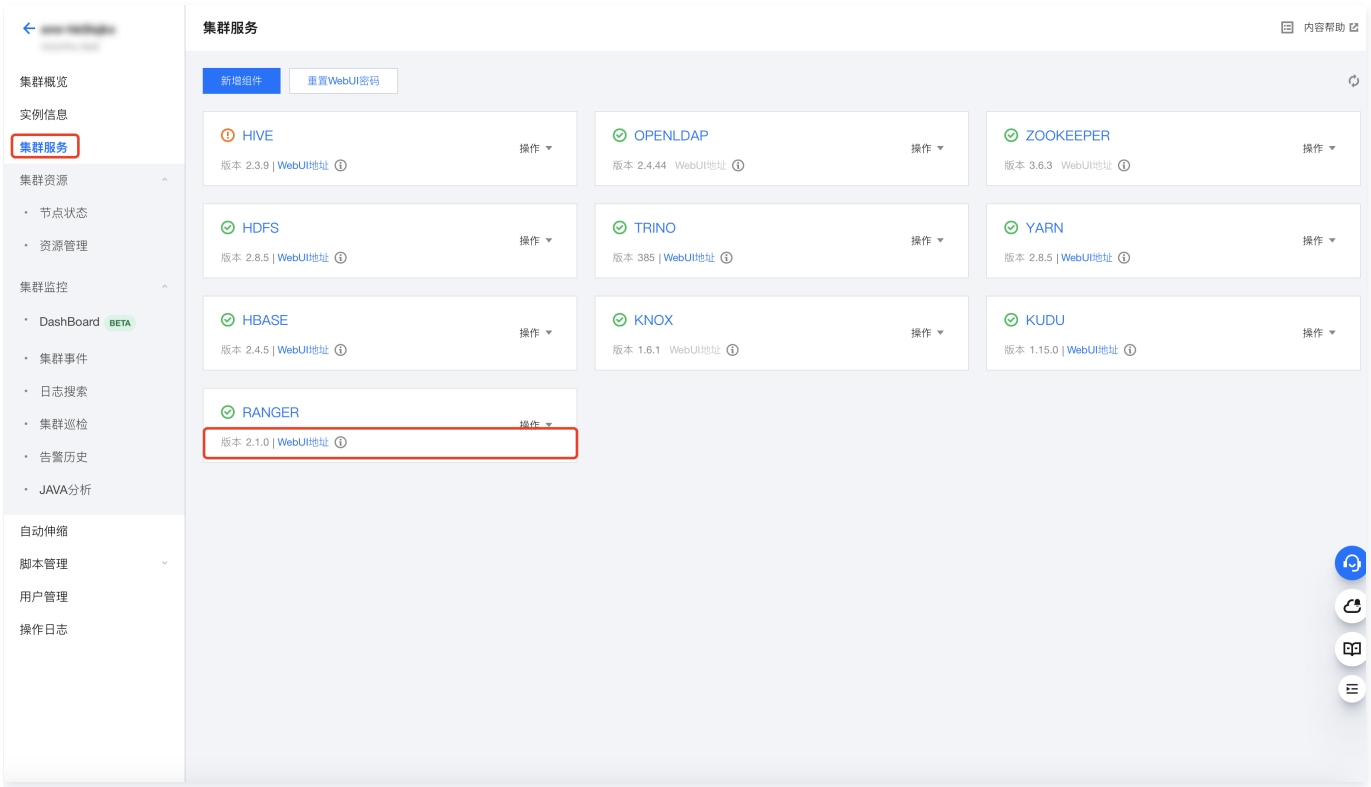
仅支持在购买集群时选择了可选组件 Ranger 的集群，若是在已创建的集群上新增 Ranger 组件，可能会出现 Web UI 无法访问的情况。默认 Ranger 安装时，Ranger Admin、Ranger UserSync 都是部署在 Master 节点上，Ranger Plugin 是部署嵌入组件主守护进程节点上。
创建集群时，在选择集群类型为 Hadoop 时可以在可选组件中选择 Ranger，Ranger 的版本根据您选择的 EMR 版本不同而存在差异。

> **说明**
> 集群类型为 Hadoop 且选择了可选组件 Ranger 时，EMR-Ranger 默认会为 HDFS、YARN 创建服务并设置默认策略。

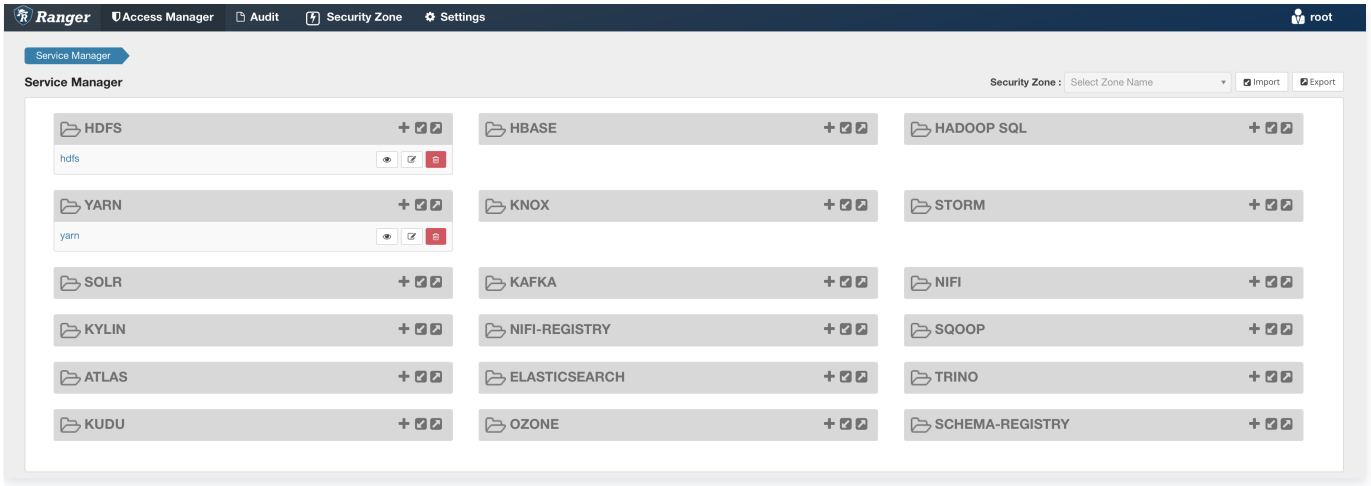


Ranger Web UI

在访问 Ranger Web UI 之前，请务必确认当前所购买的集群是否配置了公网 IP，然后在集群服务中单击 Ranger 组件的 Web UI 地址链接。



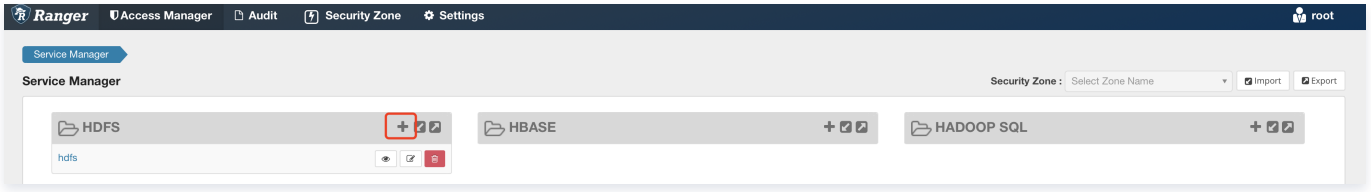
Web UI 地址链接跳转后，会提示输入用户名及密码，即在购买集群时设置的用户名及密码。



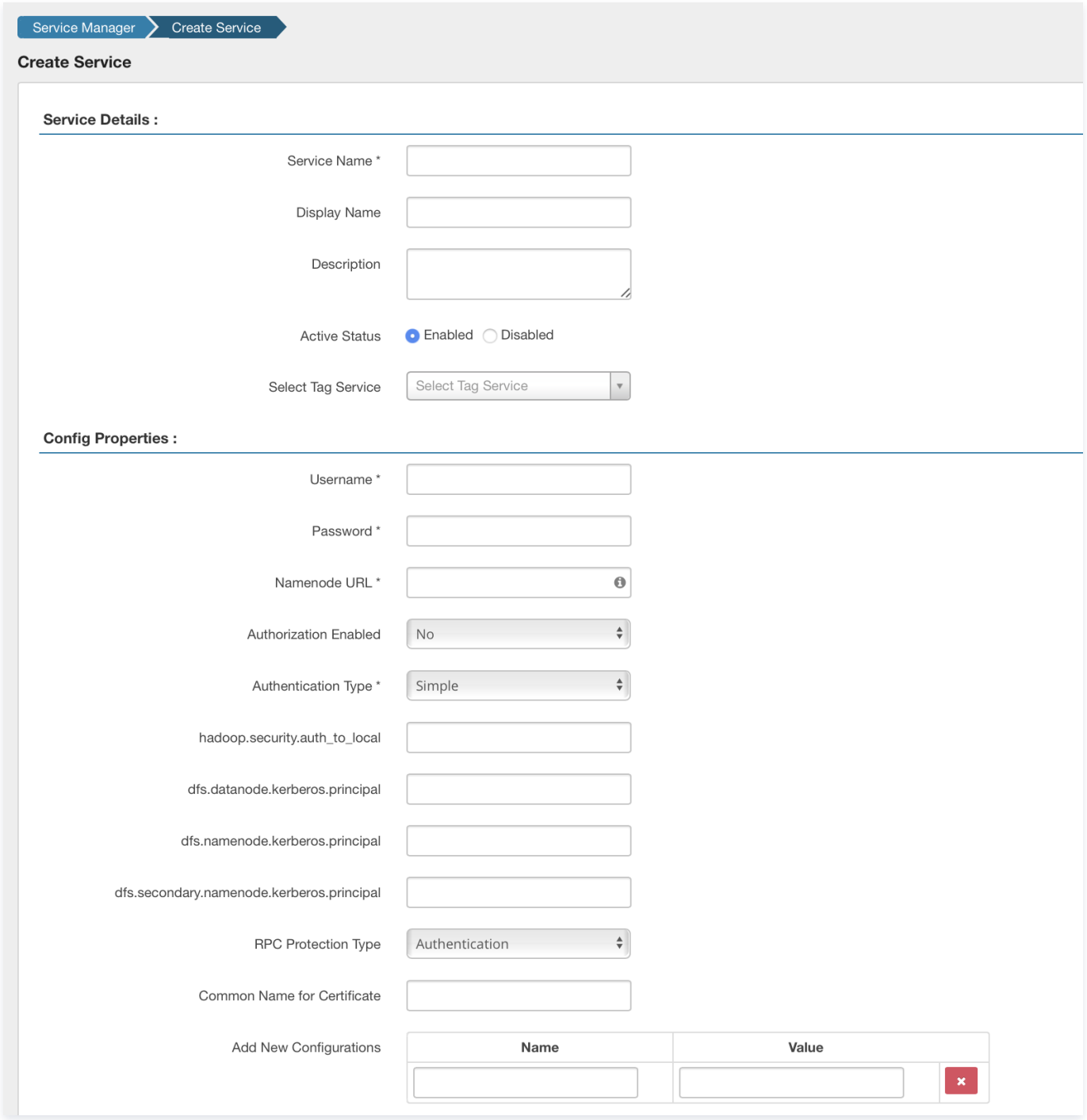
HDFS 集成 Ranger

> **注意**
> 请确保 HDFS 相关服务运行正常并且当前集群已安装 Ranger。

1. 使用 EMR Ranger Web UI 页面添加 EMR Ranger HDFS 服务。



2. 配置 EMR Ranger HDFS Service 相关参数。

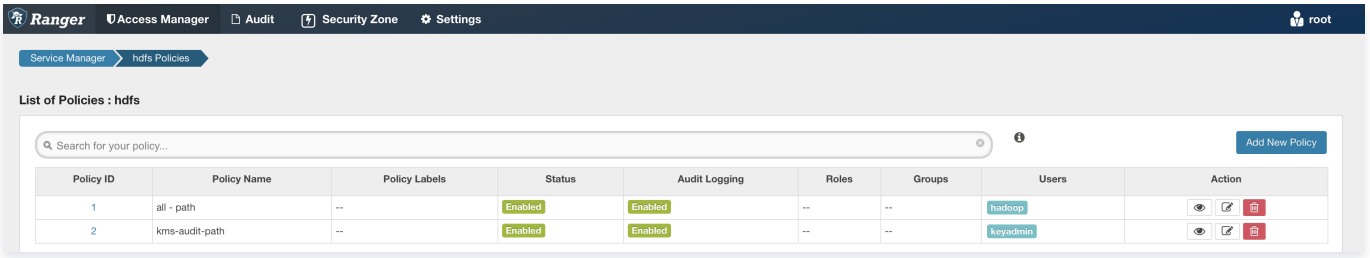


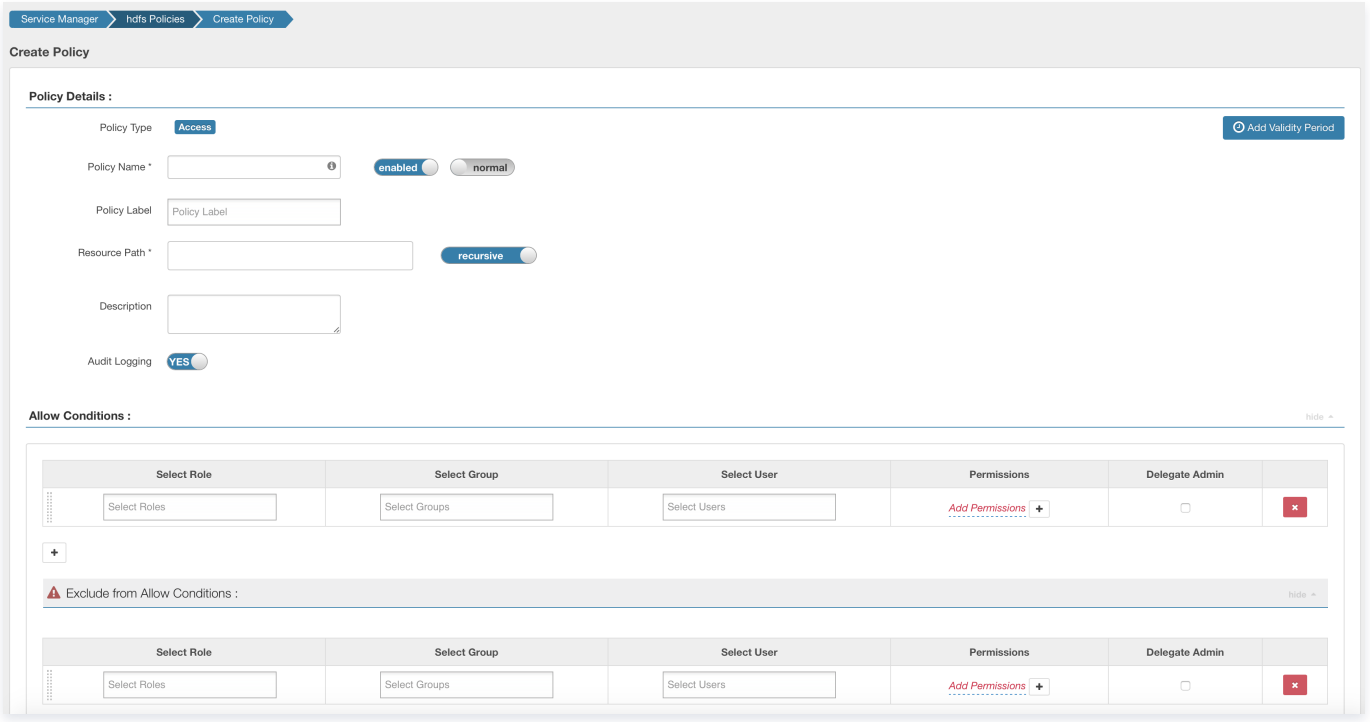
参数名	是否必填项	解释
Service Name	是	服务名称，Ranger Web UI 主 HDFS 组件显示服务名称
Description	否	服务描述信息
Active Status	默认	服务启用状态，默认启用
UserName	是	资源使用的用户名
Password	是	用户密码
NameNode URL	是	HDFS 地址
Authorization Enable	默认	标准集群选择 No；高安全集群选择 Yes

Authorization Type	是	Simple：标准集群；Kerberos：高安全集群

3. EMR Ranger HDFS 资源权限配置。
   - 单击配置好的 EMR Ranger HDFS Service

   - 配置 Policy





4. 添加完 Policy 后，稍等约半分钟等待 Policy 生效。生效后使用 user1 就可以对 HDFS 文件系统的 /user 进行读写操作。

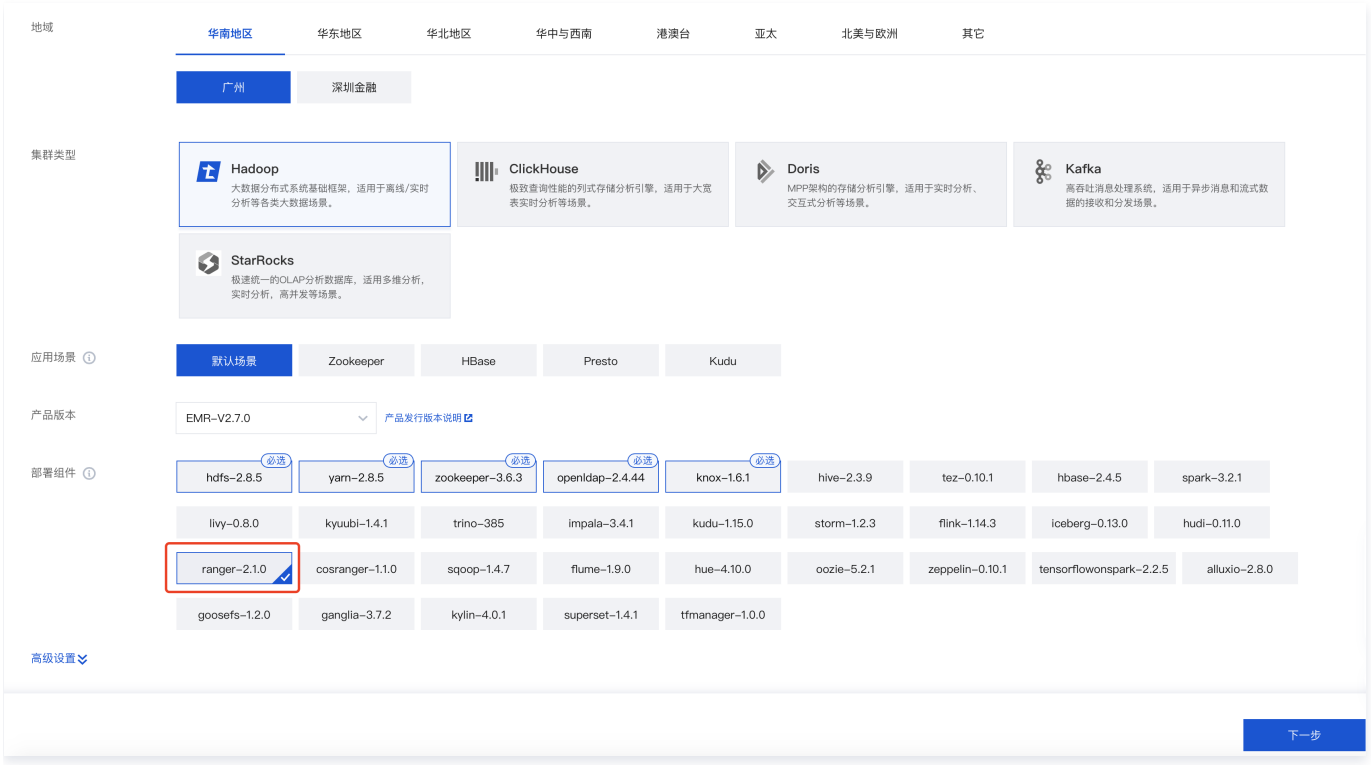
YARN 集成 Ranger

最近更新时间：2023-07-27 15:00:53

使用准备

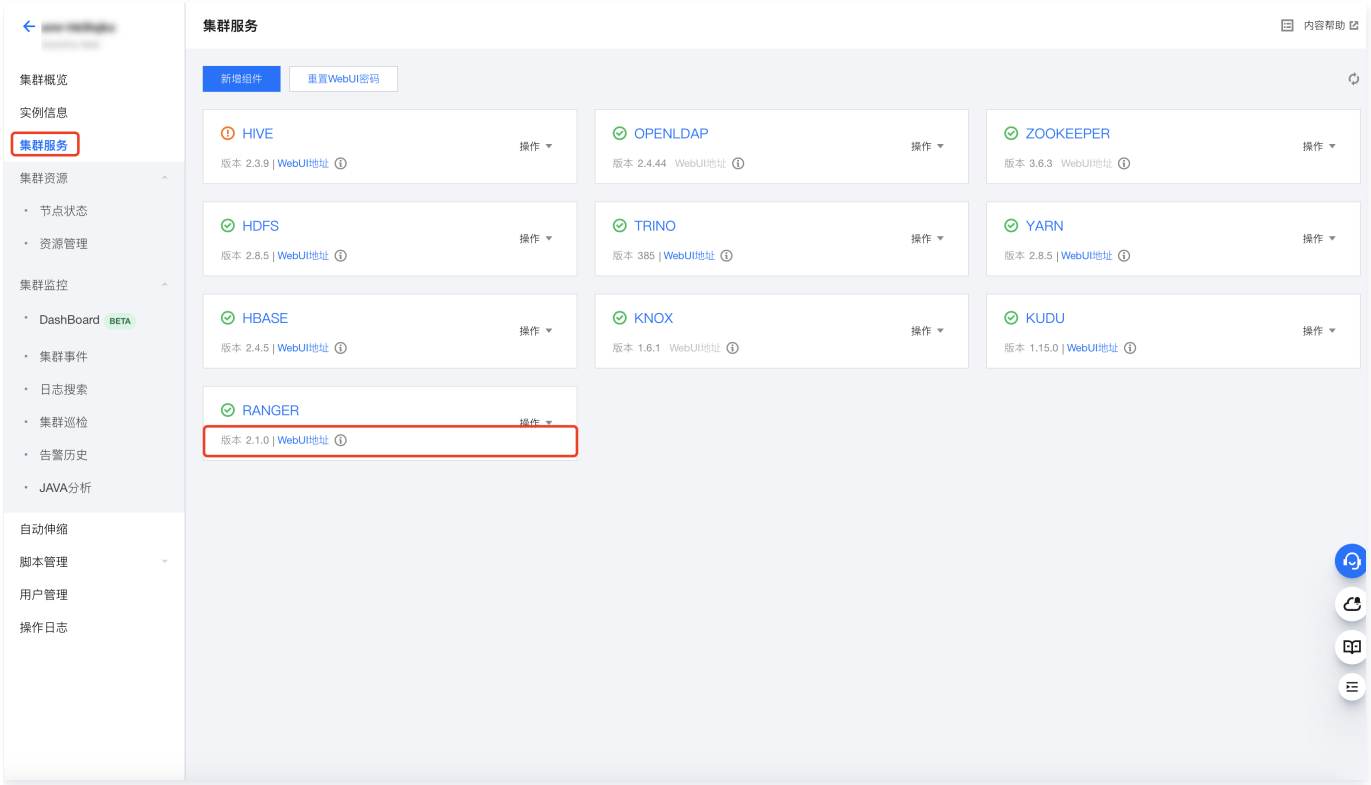
Ranger 安装时，Ranger Admin、Ranger UserSync 都是部署在 Master 节点上，Ranger Plugin 是部署在嵌入组件主守护进程节点上。
创建集群时，在选择集群类型为 Hadoop 时可以在可选组件中选择 Ranger，Ranger 的版本根据您选择的 EMR 版本不同而存在差异。

> **说明**
> 集群类型为 Hadoop 且选择了可选组件 Ranger 时，EMR-Ranger 默认会为 HDFS、YARN 创建服务并设置默认策略。

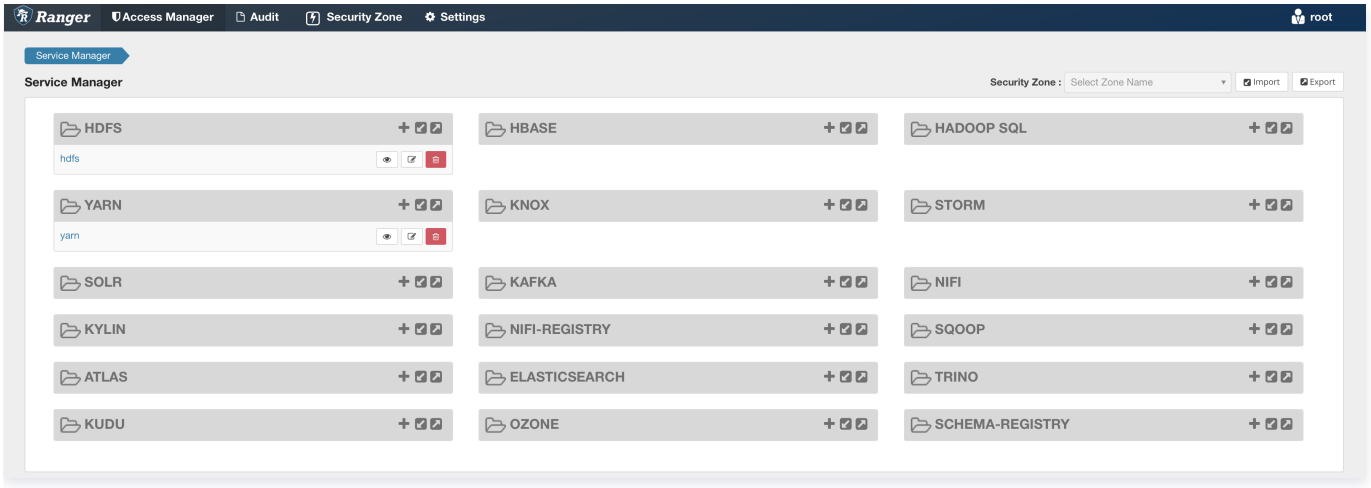


Ranger Web UI

在访问 Ranger Web UI 之前，请务必确认当前所购买的集群是否配置了公网 IP，然后在集群服务中单击 Ranger 组件的 Web UI 地址链接。



Web UI 地址链接跳转后，会提示输入用户名及密码，即在购买集群时设置的用户名及密码。

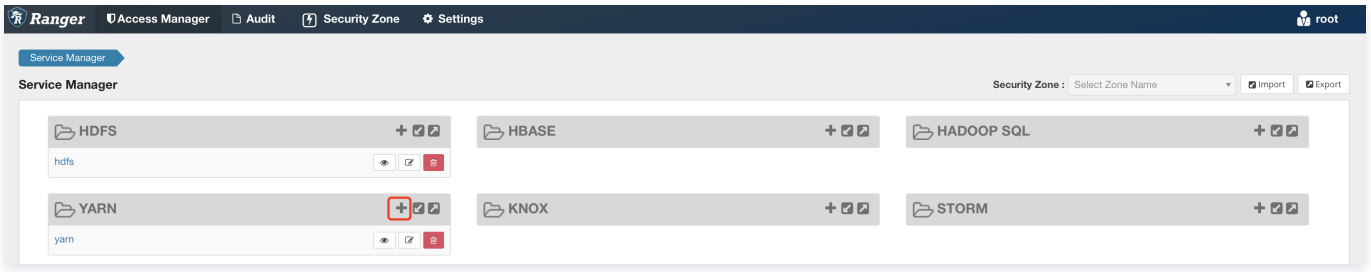


YARN 集成 Ranger

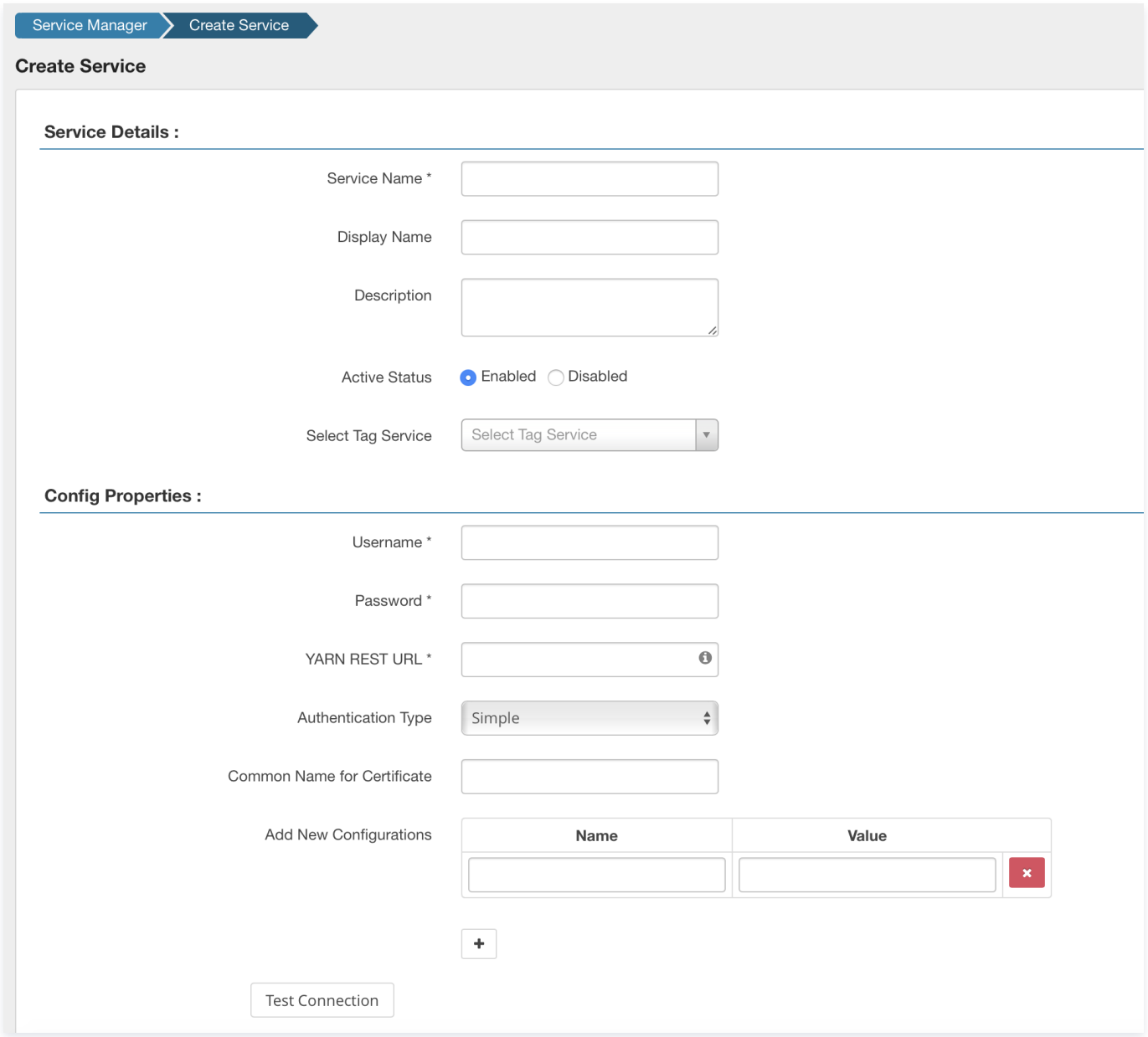
> **注意**
> 请确保 YARN 相关服务运行正常并且当前集群已安装 Ranger。

EMR Ranger YARN 目前仅支持 Capacity Scheduler 队列的 ACL，不支持 Fair Scheduler 队列的 ACL。Ranger YARN 队列 ACL 与 YARN 自带的 Capacity Scheduler 配置共同生效，且优先级低于 Capacity Scheduler 配置，只有在 YARN 自带的 Capacity Scheduler 配置拒绝校验时才会校验 Ranger YARN 权限。**建议不要在配置文件设置 ACL，而是使用 Ranger 设置 ACL。**

1. 使用 EMR Ranger Web UI 页面添加 EMR Ranger YARN 服务。



2. 配置 EMR Ranger YARN Service 相关参数。

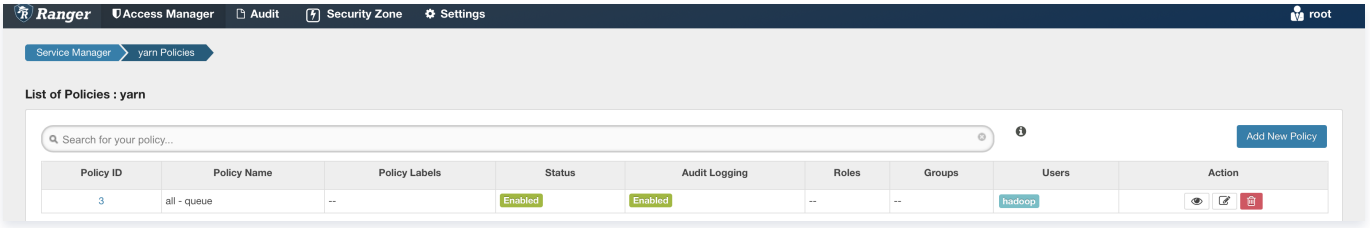


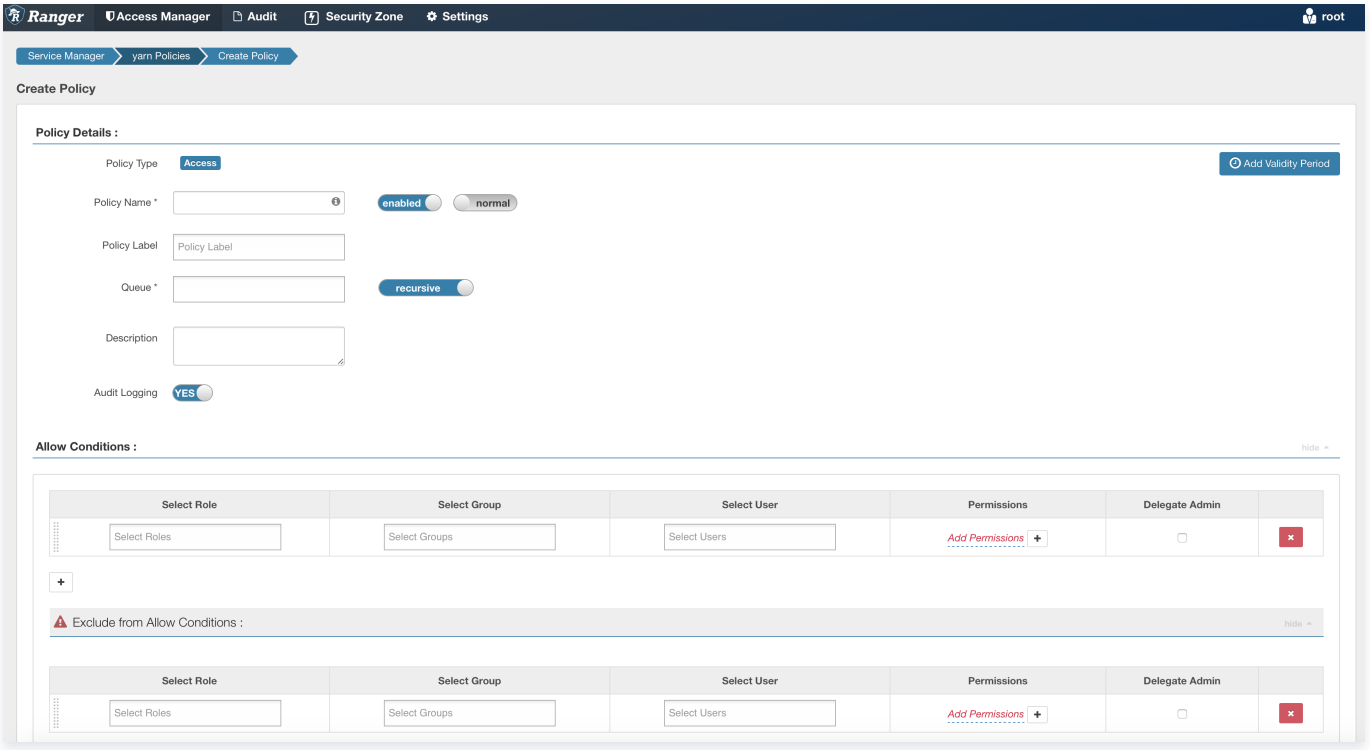
参数名	是否必填项	解释
Service Name	是	服务名称，Ranger Web UI 主 YARN 组件显示服务名称
Description	否	服务描述信息
Active Status	默认	服务启用状态，默认启用
UserName	是	资源使用的用户名

Password	是	用户密码
NameNode URL	是	YARN 地址
Authorization Enable	默认	标准集群选择 No；高安全集群选择 Yes
Authorization Type	是	Simple：标准集群；Kerberos：高安全集群

3. EMR Ranger YARN 资源权限配置。
   - 单击配置好的 EMR Ranger HDFS Service

   - 配置 Policy





4. 添加完 Policy 后，稍等约半分钟等待 Policy 生效。生效后使用 user1 就可以向 YARN 的 root.default 队列中提交、删除、查询作业等操作。

> **注意**
> 在配置 Ranger YARN Service 以及 Policy 时请务必确保期间没有 YARN 作业，否则会出现某些用户作业提交权限问题。

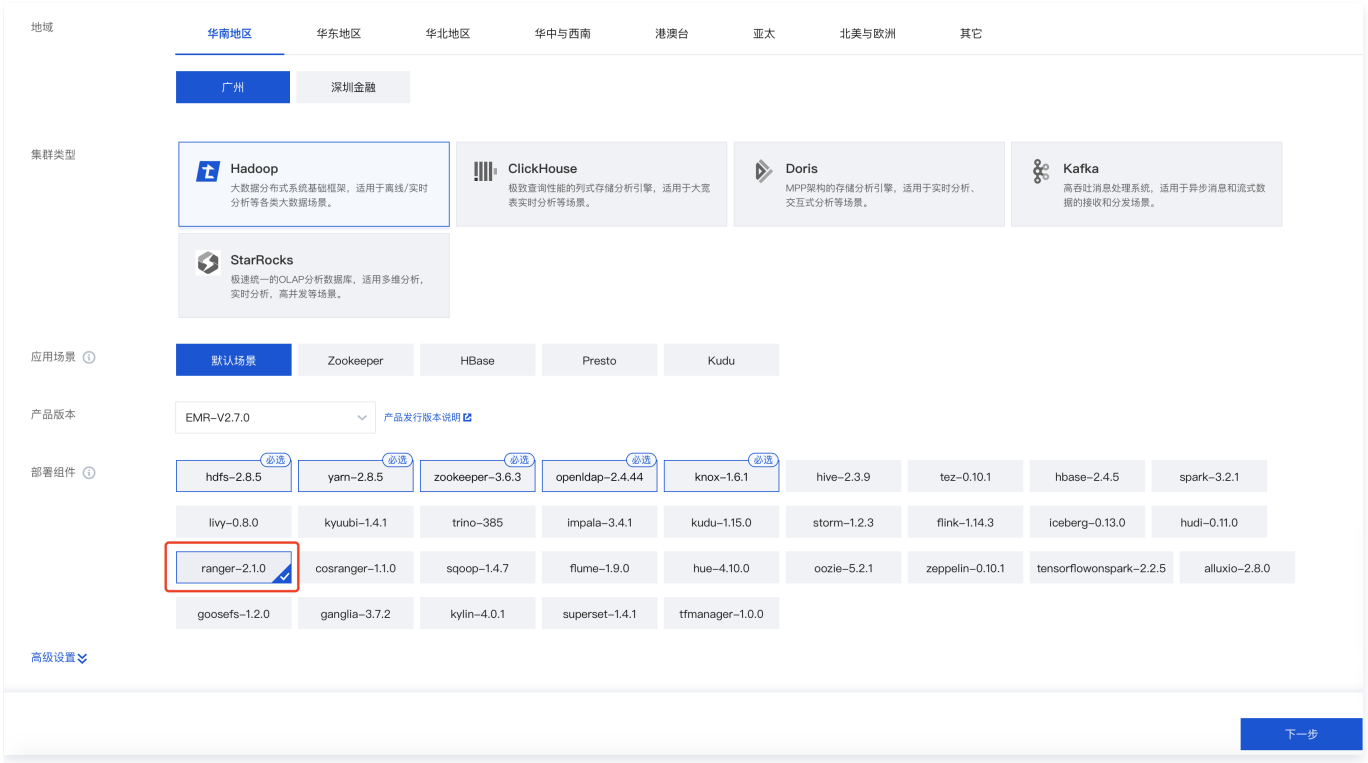
HBase 集成 Ranger

最近更新时间：2024-08-23 14:52:21

使用准备

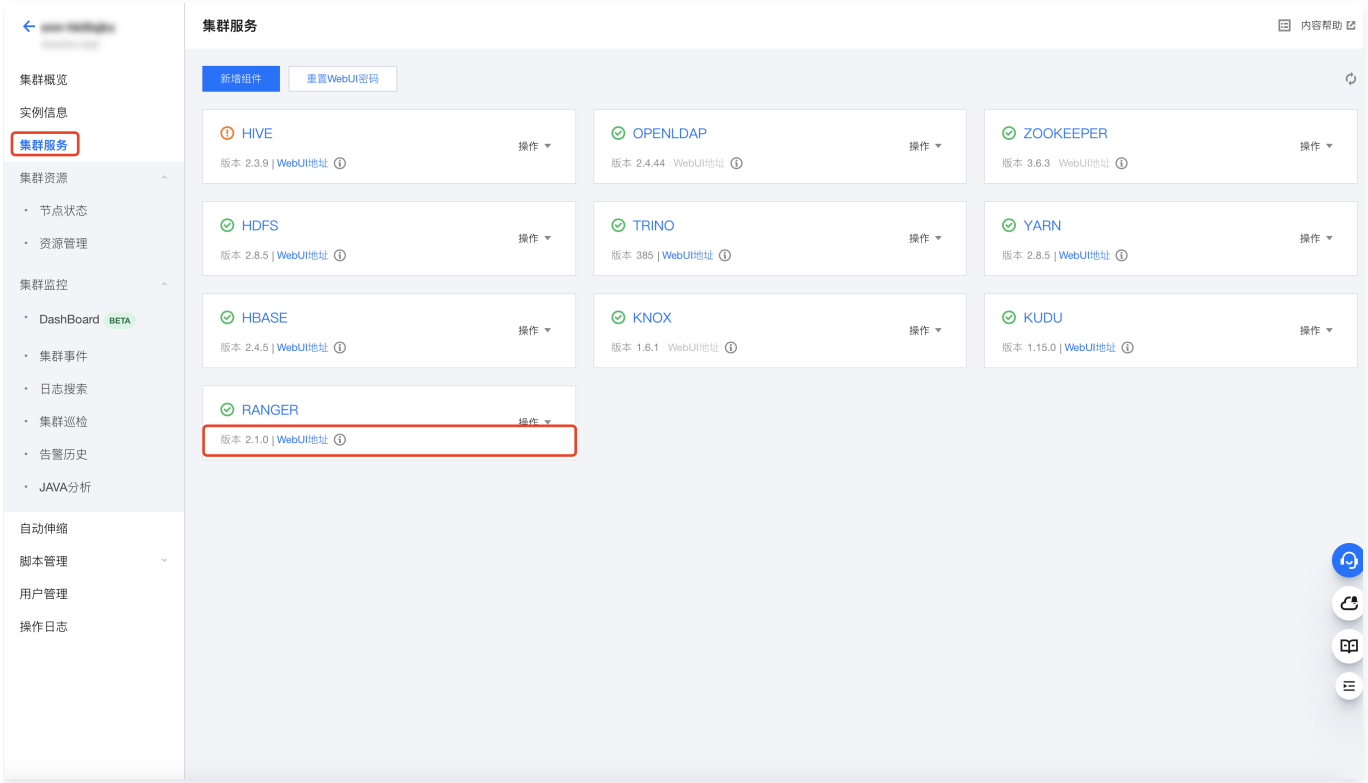
Ranger 安装时，Ranger Admin、Ranger UserSync 都是部署在 Master 节点上，Ranger Plugin 是部署在嵌入组件主守护进程节点上。
创建集群时，在选择集群类型为 Hadoop 时可以在可选组件中选择 Ranger，Ranger 的版本根据您选择的 EMR 版本不同而存在差异。

> **说明**
> 集群类型为 Hadoop 且选择了可选组件 Ranger 时，EMR-Ranger 默认会为 HDFS、YARN 创建服务并设置默认策略。

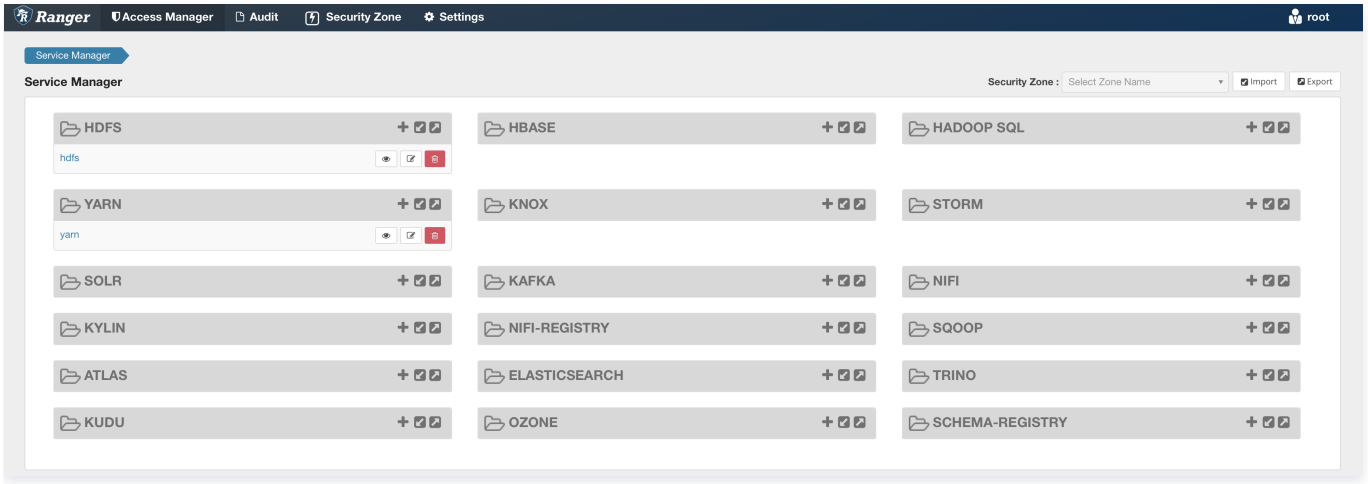


Ranger Web UI

在访问 Ranger Web UI 之前，请务必确认当前所购买的集群是否配置了公网 IP，然后在集群服务中单击 Ranger 组件的 Web UI 地址链接。



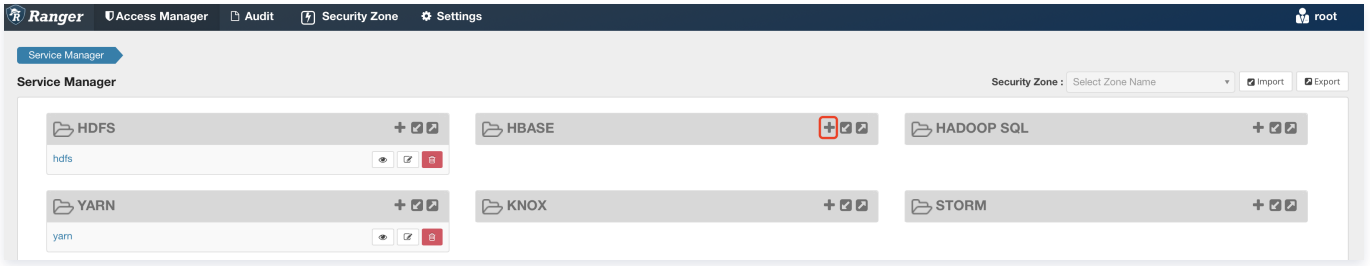
Web UI 地址链接跳转后，会提示输入用户名及密码，即在购买集群时设置的用户名及密码。



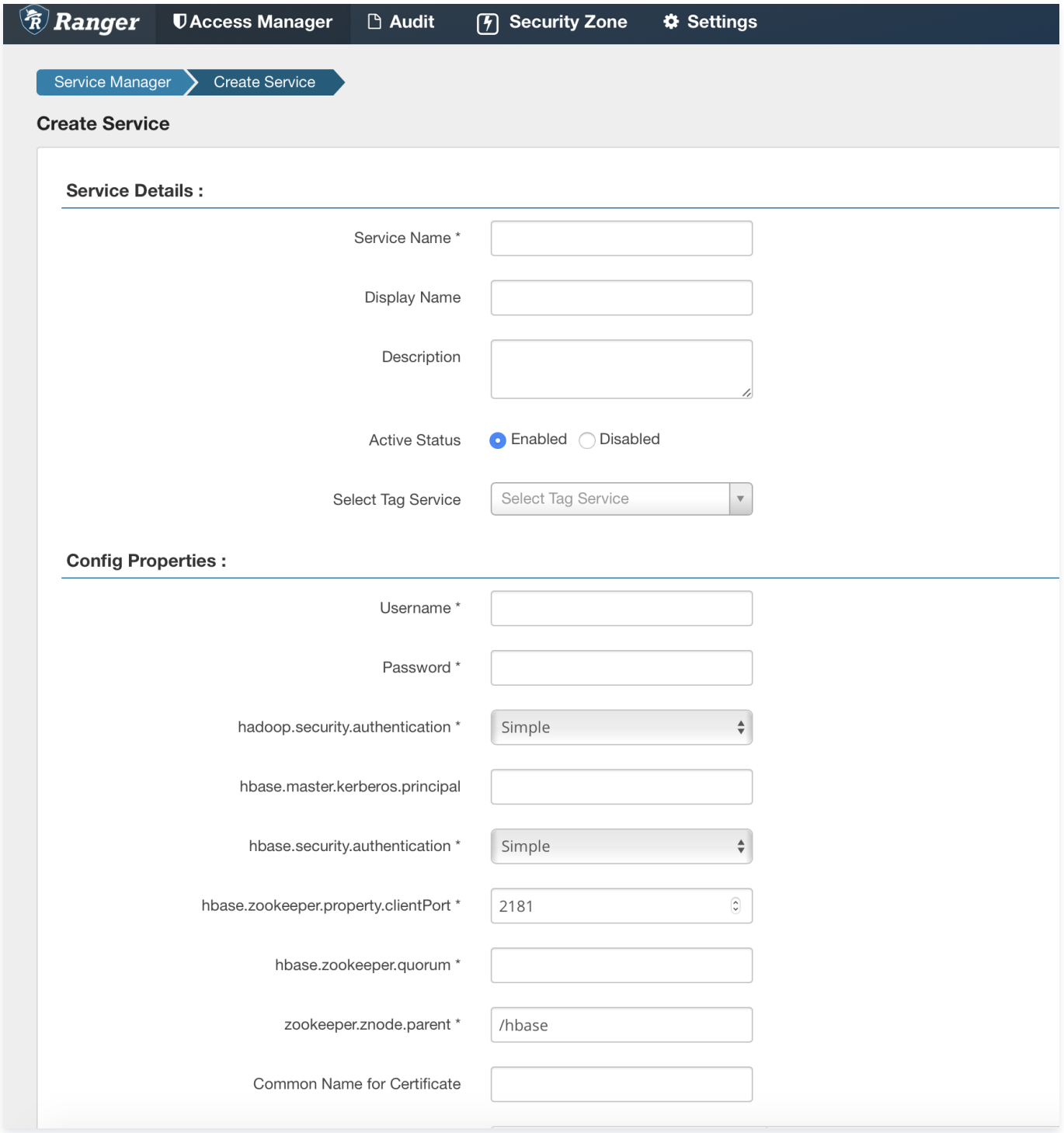
HBase 集成 Ranger

> **注意**
> 请确保 HBase 相关服务运行正常并且当前集群已安装 Ranger。

1. 使用 EMR Ranger Web UI 页面添加 EMR Ranger Hbase 服务。



2. 配置 EMR Ranger HBase Service 相关参数。

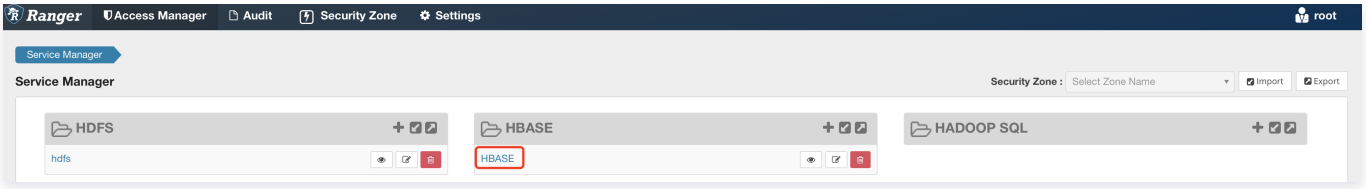


参数名	是否必填项	解释
Service Name	是	服务名称，Ranger Web UI 主 HBase 组件显示服务名称
Description	否	服务描述信息
Active Status	默认	服务启用状态，默认启用
UserName	是	资源使用的用户名
Password	是	用户密码
Hbase.zookeeper.property.clientPort	是	ZK 客户端请求端口

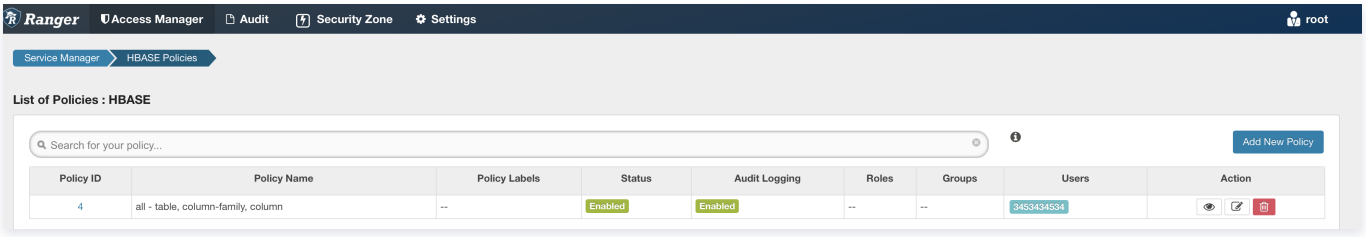
Hbase.zookeeper.quorum	是	ZK 集群 IP
Zookeeper.znode.parent	是	ZK 节点信息

3. EMR Ranger Hbase 资源权限配置。

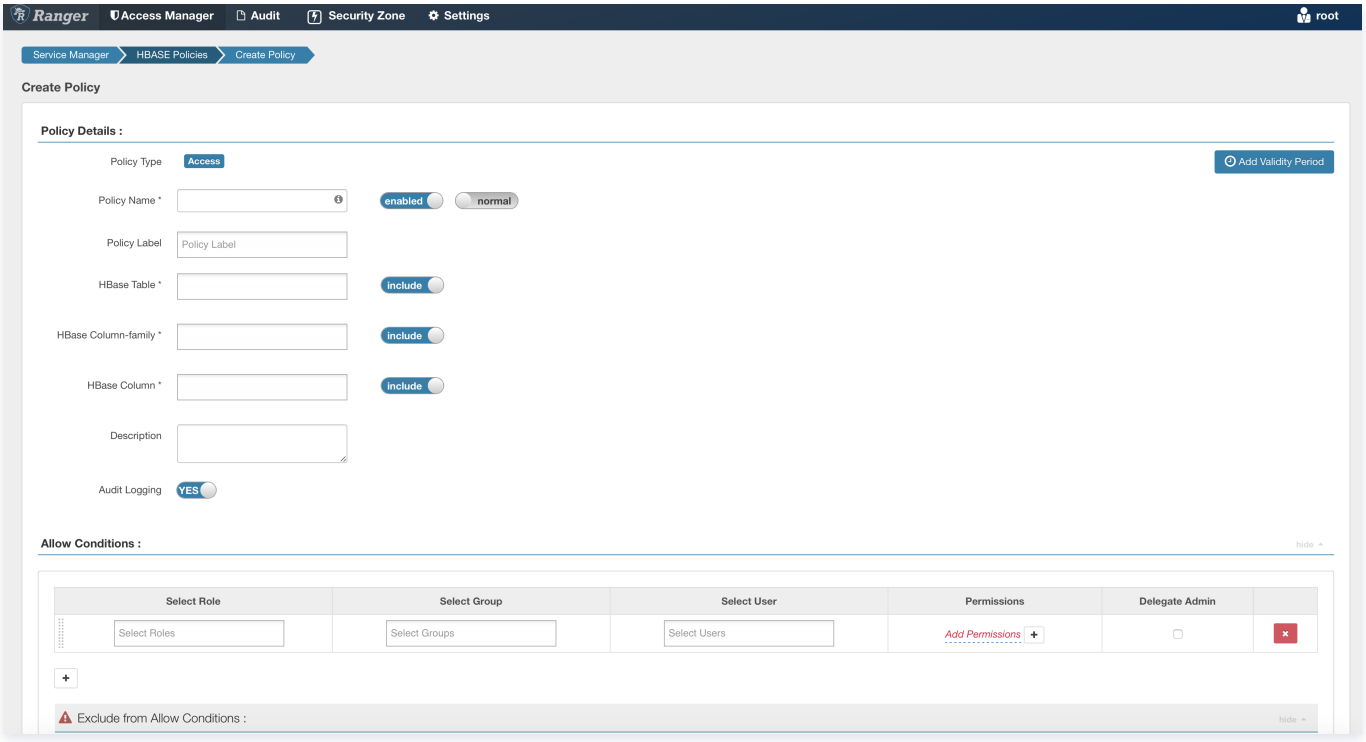
- 单击配置好的 EMR Ranger Hbase Service



- 配置 Policy



上图中的 Users 为 Hbase，它的 Policy Name 是 all-table、column-family、column，也就是 Hbase 用户具有 Region Balance、MemeStore Flush、Compaction、Split 权限。**请确保创建的 Service 是有这些权限的。**



参数	是否必选项	解释
HBase Table	是	HBase 表名
HBase Column-family	是	HBase 表中的列簇
HBase Column	是	HBase 表中的列簇下的限定符

4. 添加完 Policy 后，稍等约半分钟等待 Policy 生效。生效后使用 user1 就可以对 Hbase 的 order 表相关列簇和限定符进行相关操作。

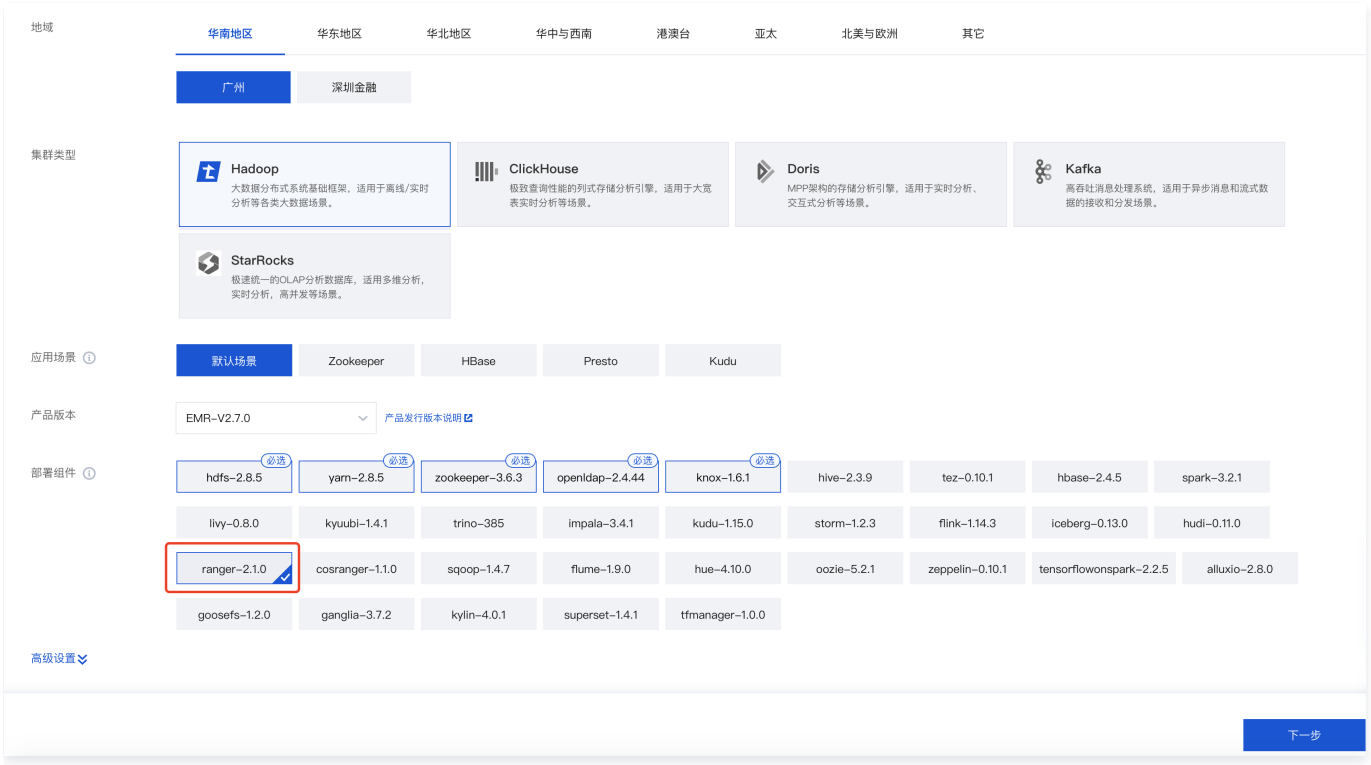
Presto 集成 Ranger

最近更新时间：2023-07-27 15:03:50

使用准备

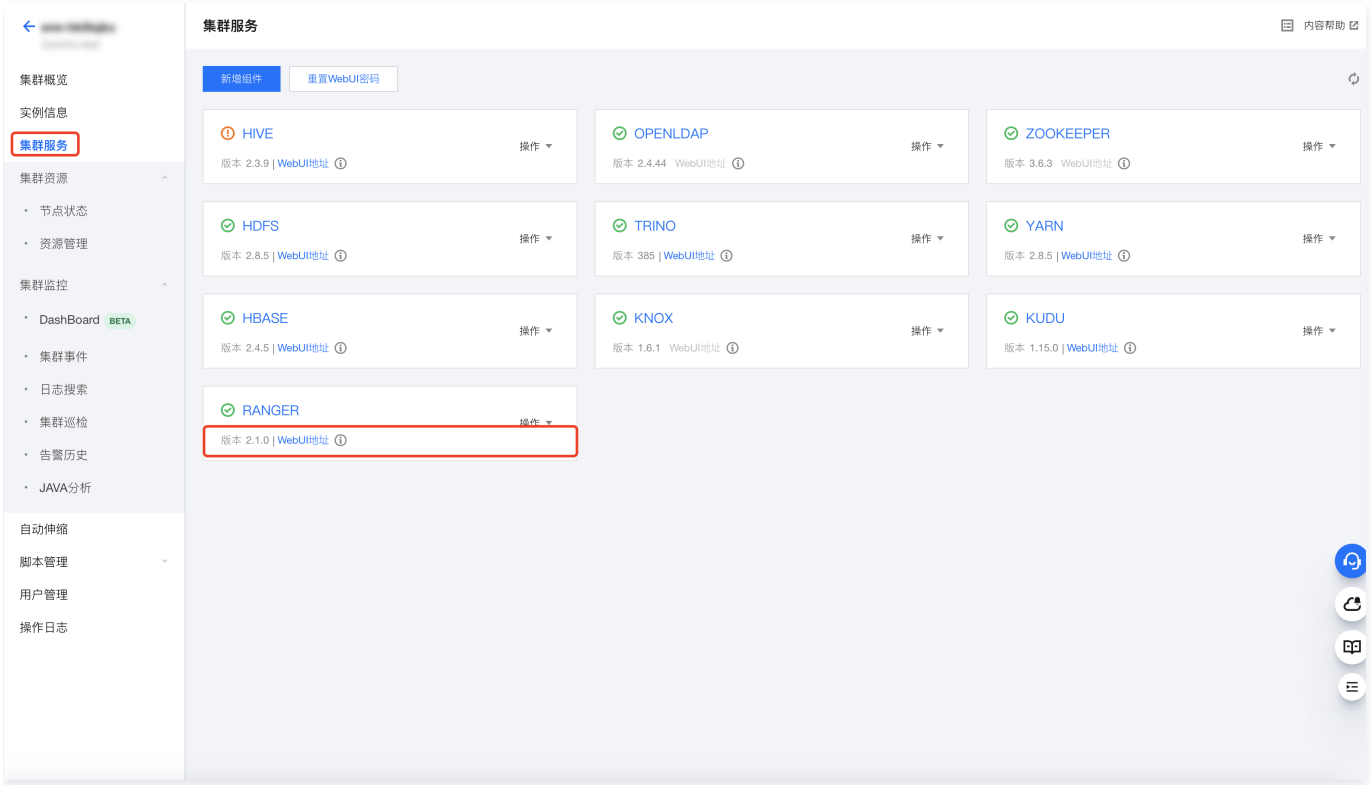
Ranger 安装时，Ranger Admin、Ranger UserSync 都是部署在 Master 节点上，Ranger Plugin 是部署在嵌入组件主守护进程节点上。
创建集群时，在选择集群类型为 Hadoop 时可以在可选组件中选择 Ranger，Ranger 的版本根据您选择的 EMR 版本不同而存在差异。

> **说明**
> 集群类型为 Hadoop 且选择了可选组件 Ranger 时，EMR-Ranger 默认会为 HDFS、YARN 创建服务并设置默认策略。

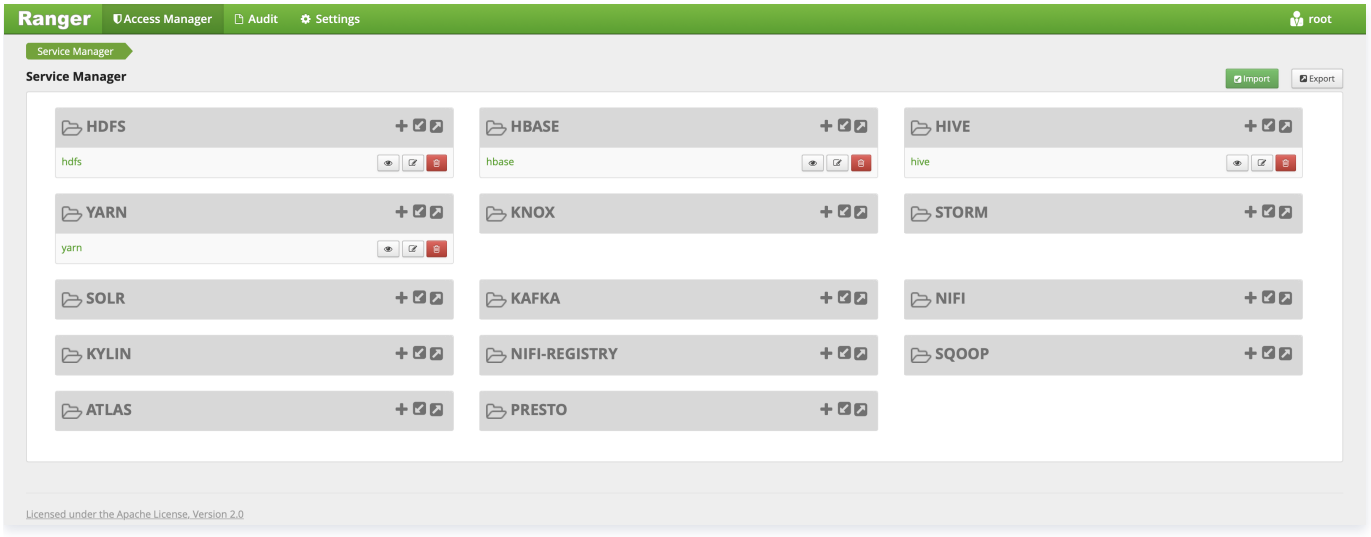


Ranger Web UI

在访问 Ranger Web UI 之前，请务必确认当前所购买的集群是否配置了公网 IP，然后在集群服务中单击 Ranger 组件的 Web UI 地址链接。



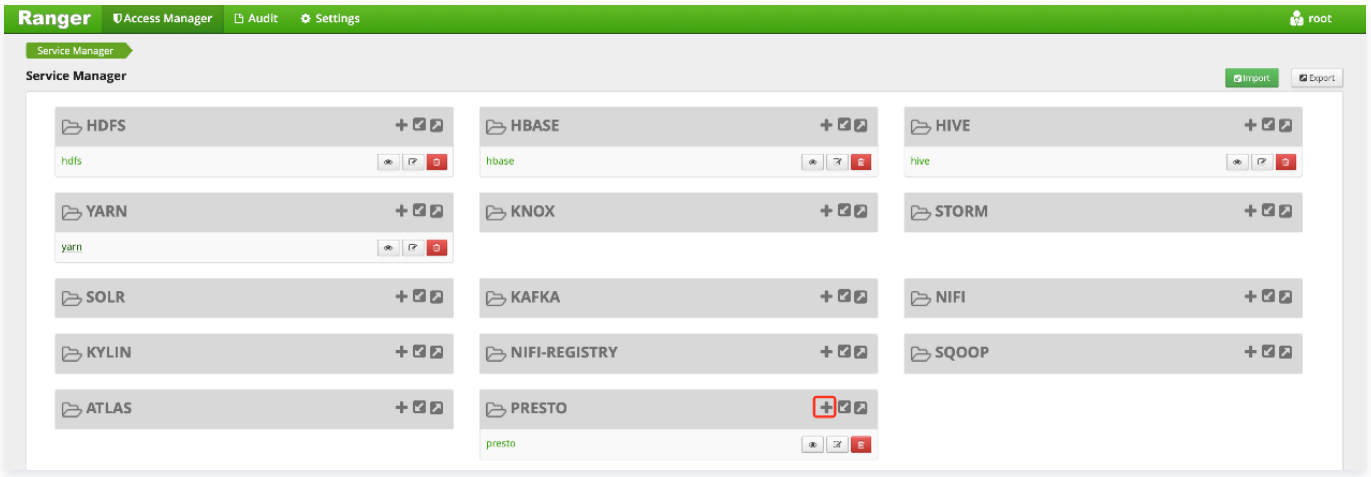
Web UI 地址链接跳转后，会提示输入用户名及密码，即在购买集群时设置的用户名及密码。



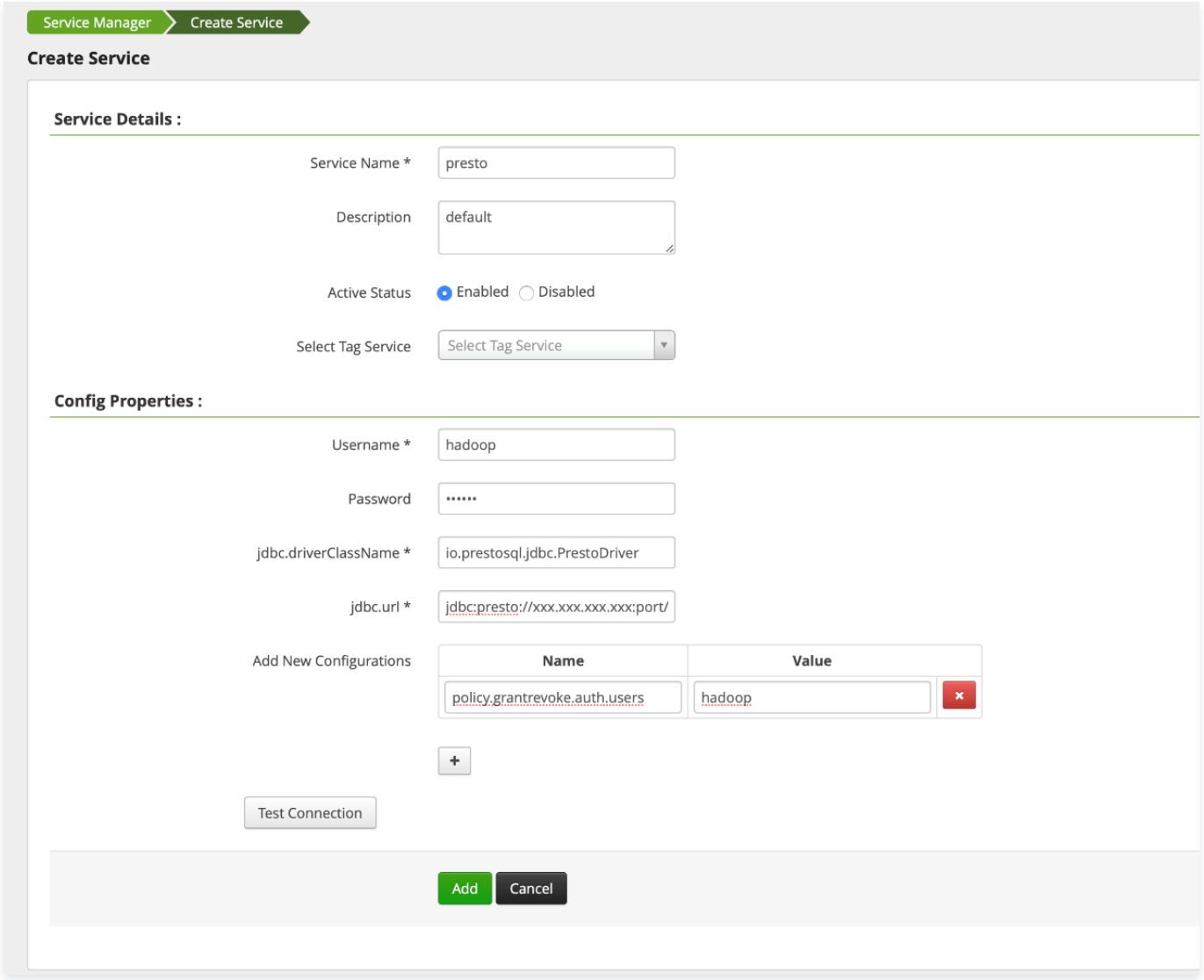
Presto 集成 Ranger

> **注意**
> 请确保 Presto 相关服务运行正常并且当前集群已安装 Ranger。

1. 使用 EMR Ranger Web UI 页面添加 EMR Ranger Presto 服务。



2. 配置 EMR Ranger Presto Service 相关参数。

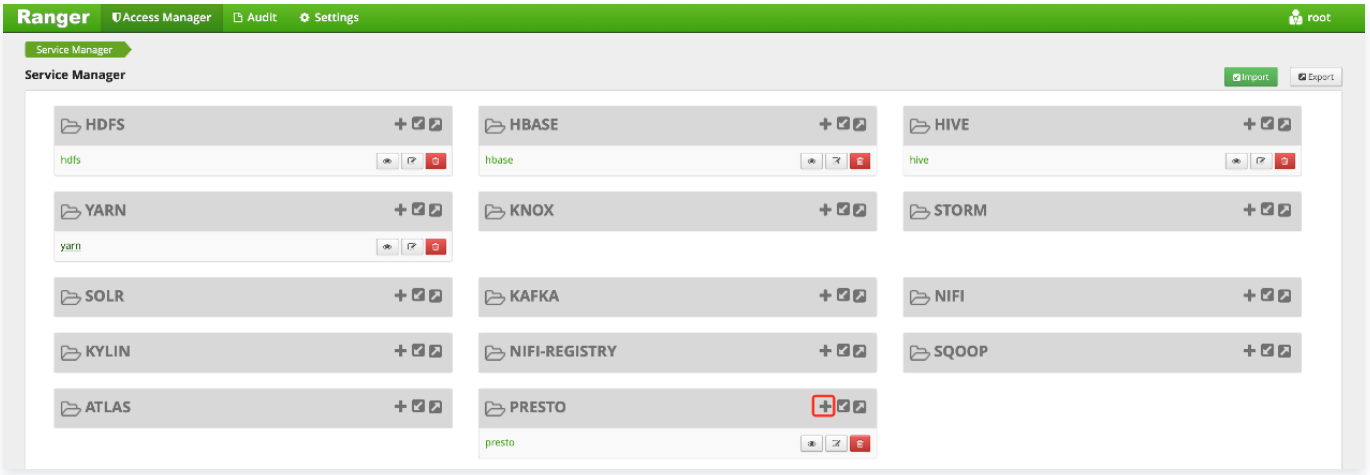


参数	是否必填项	解释
Service Name	是	服务名称，Ranger Web UI 主 Predo 组件显示服务名称
Description	否	服务描述信息

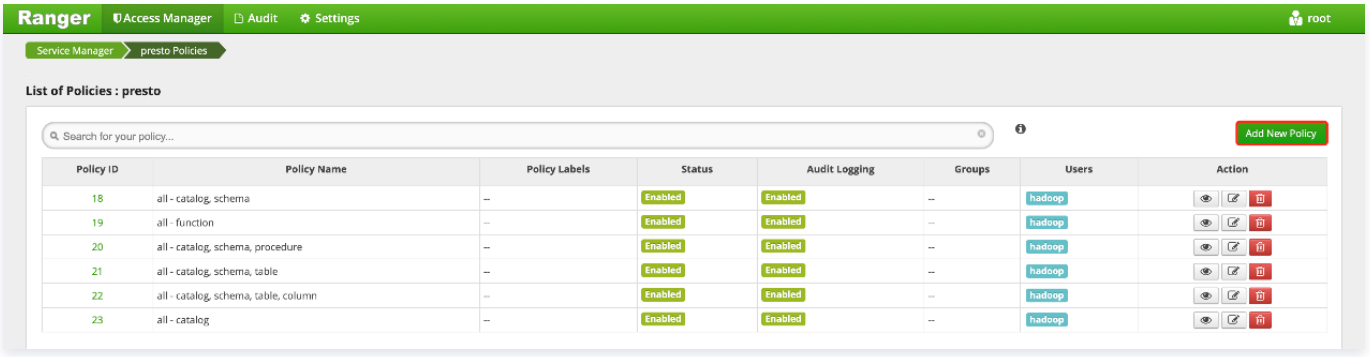
Active Status	默认	服务启用状态，默认启用
UserName	是	资源使用的用户名
Password	是	用户密码
JDBC.driverClassName	是	驱动的类名全路径
jdbc.url	是	Presto jdbc 连接形式的地址，例如 jdbc:presto://ip/hostname:port

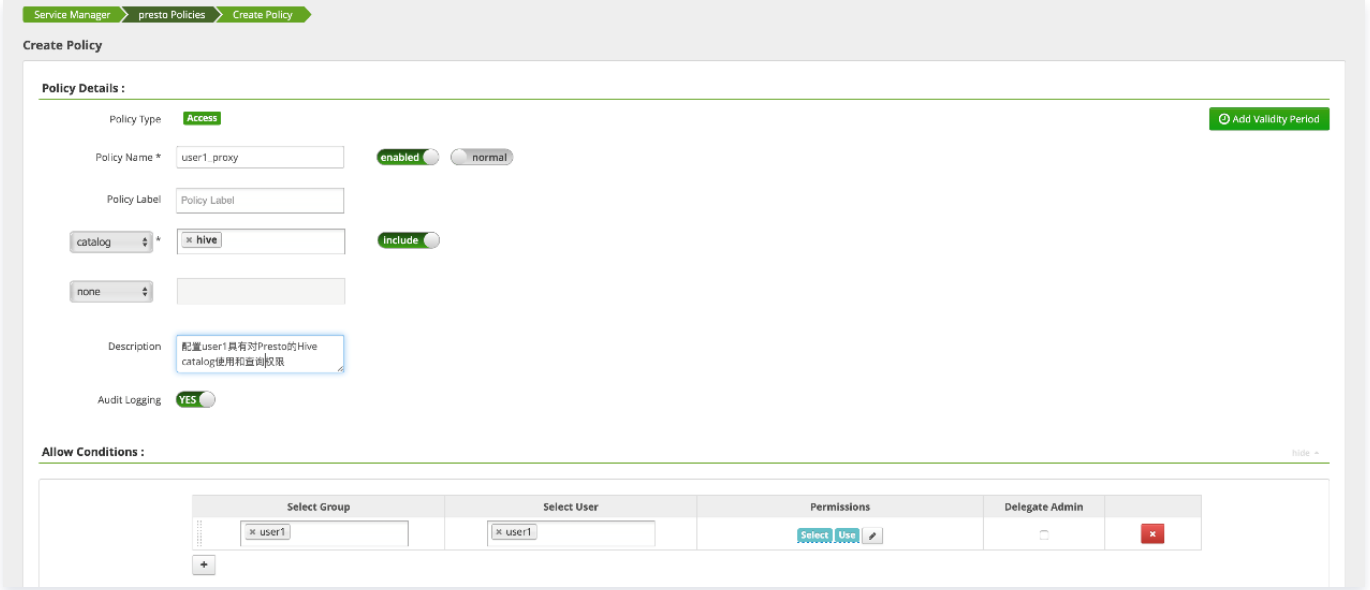
3. EMR Ranger Presto 资源权限配置

- 单击配置好的 EMR Ranger Presto Service



- 配置 Policy





4. 添加完 Policy 后，稍等约半分钟等待 Policy 生效。生效后使用 user1 就可以对 Presto 的 catalog 进行查看和使用。

Starrocks 集成 Ranger

最近更新时间：2025-06-20 16:25:42

前提条件

StarRocks 支持版本：3.1.9 及以后版本，Ranger 版本 2.1 及以后版本。

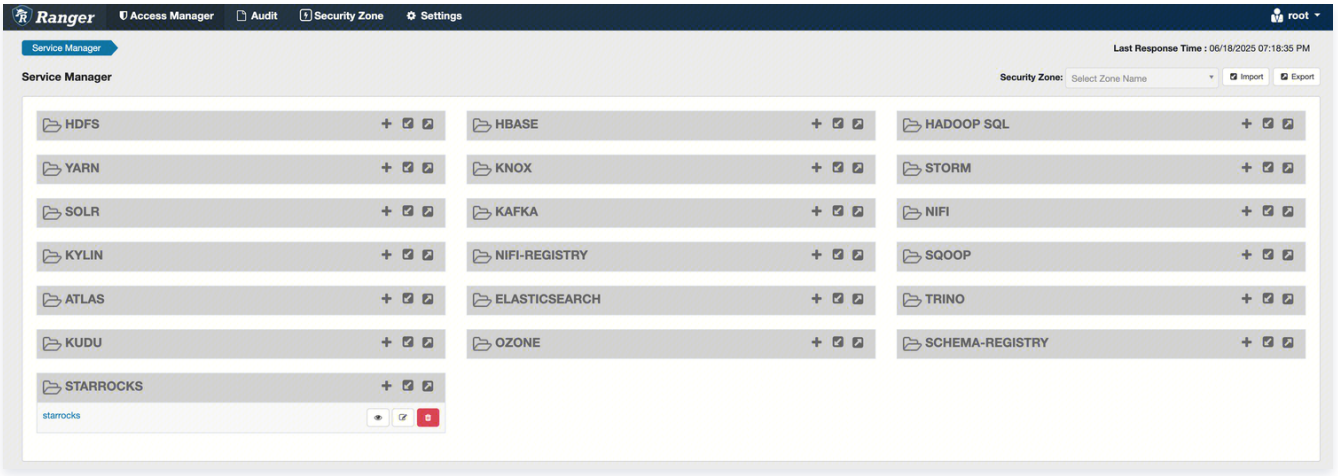
- 暂不支持 Kerberos 认证的 Ranger Server。
- 如果 StarRocks 与 Ranger 不在同一个集群，需确保 StarRocks 集群 Ranger 集群位于同一 VPC 内。
- 允许 StarRocks FE 节点可以访问 Ranger 服务的相关端口，如果显示 Connected to <ip>，则表示连接成功。

```
telnet <ranger-ip> <ranger-port>
```

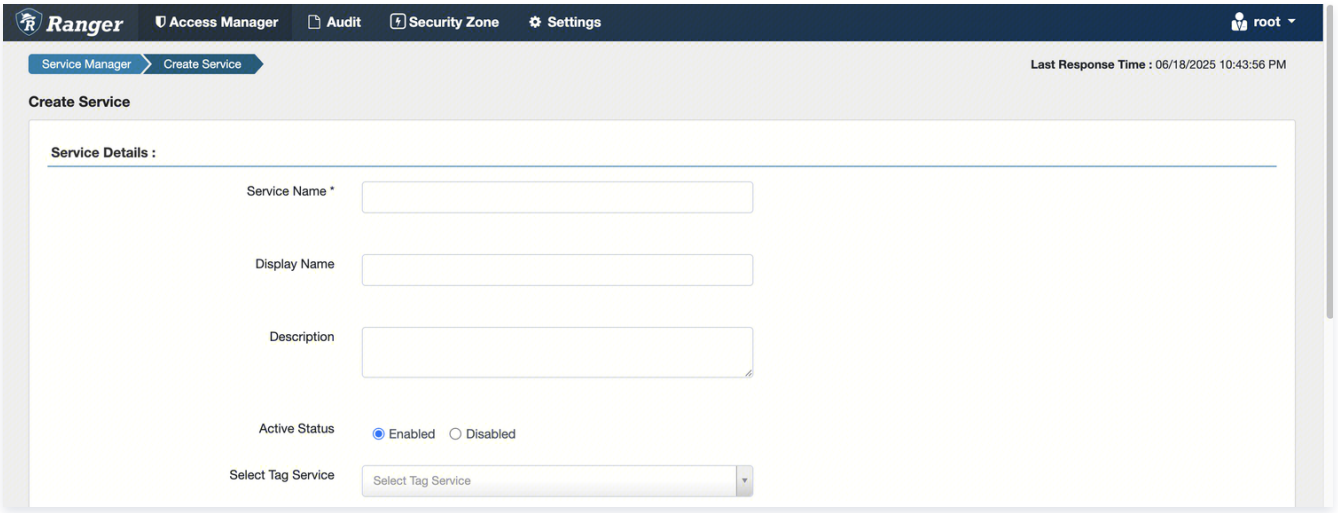
使用方式

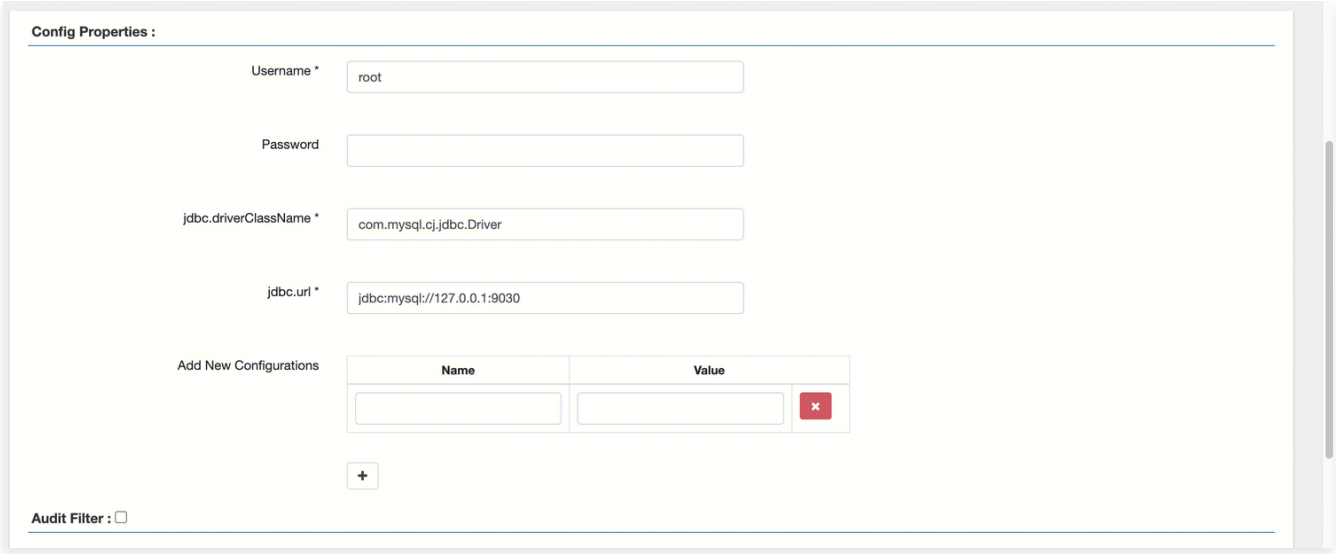
步骤一：在 Ranger Admin 上配置 StarRocks Service

1. EMR Ranger 在 StarRocks 存算分离集群中默认配置了 StarRocks Service ，因此登录 Ranger Admin 后，直接可以看到界面上已经配置了默认名称为 starorcks 的服务，可以直接使用。



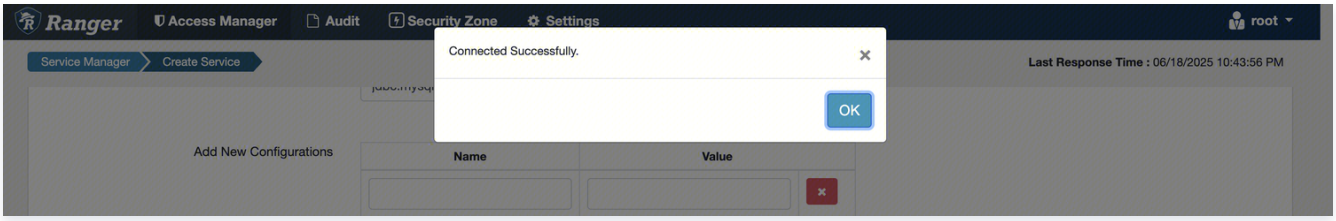
2. 如果不使用系统推荐，可以点击 STARROCKS 后的加号 (+) 配置新的 StarRocks Service 信息。





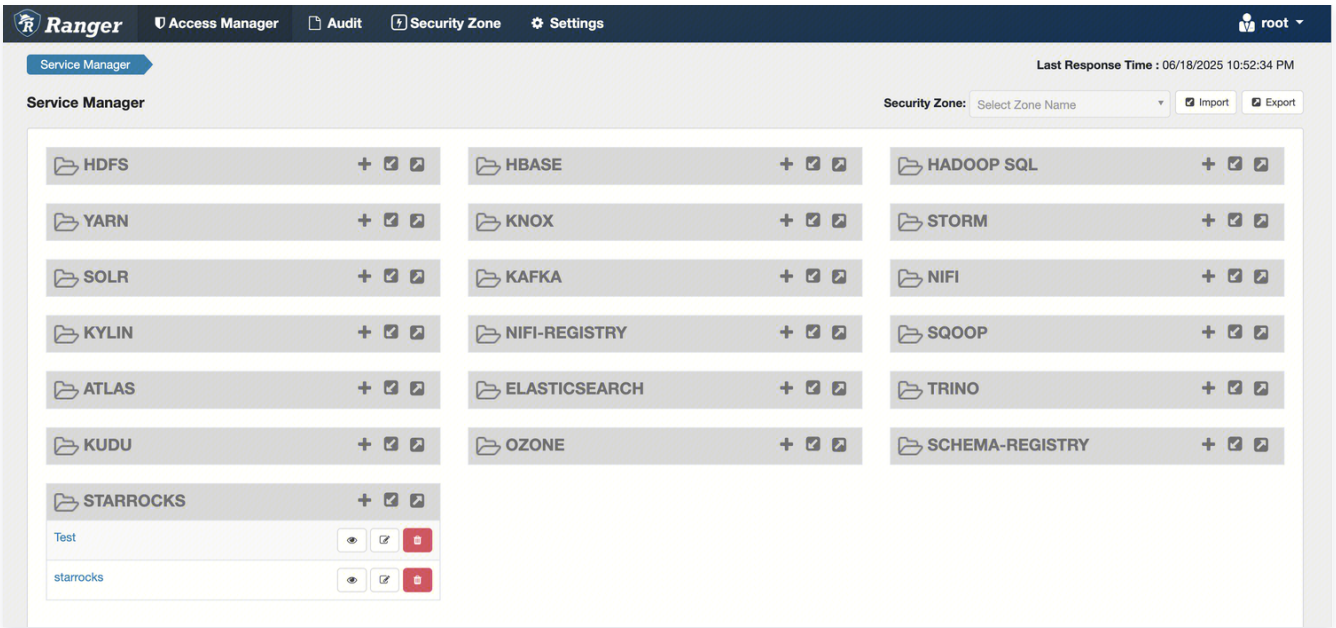
参数	说明
Service Name	服务名称，必填。
Username 和 Password	登录集群的用户名密码
jdbc.url	填写 StarRocks 实例 FE 点的 IP 和端口。格式为 jdbc:mysql://<FE地址>:<fe_query_port>。其中涉及参数如下： • <FE地址>：StarRocks 实例中 FE 节点的地址，请前往资源管理查看。 • <fe_query_port>：StarRocks 实例中 FE 的查询端口（默认为9030）

3. 单击 Test Connection，测试连通性。



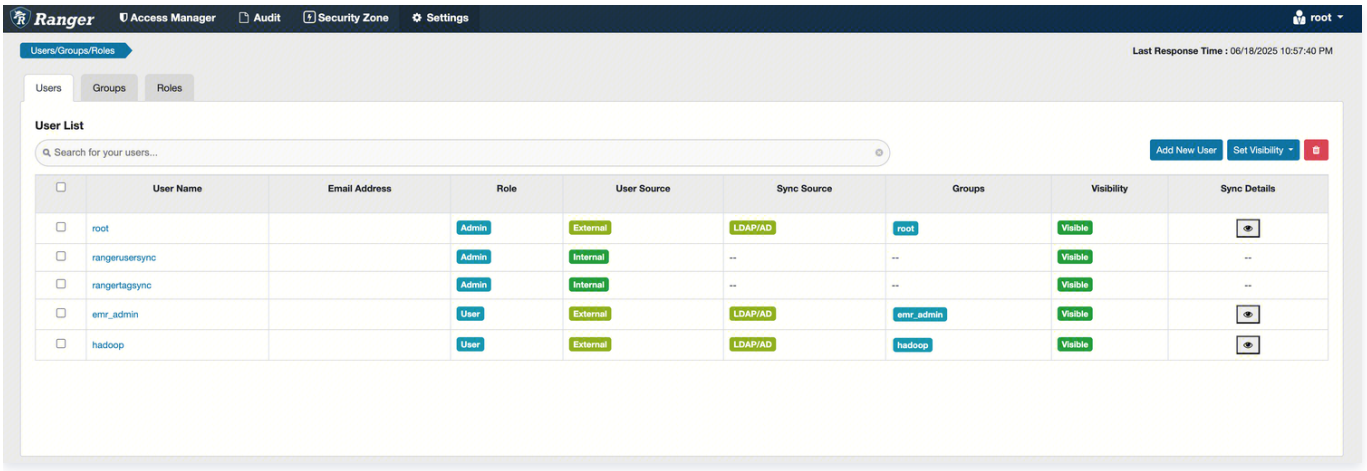
4. 测试成功后，单击 **Add**。

   自动跳转至 Service Manager 页面，即可看到添加好的 StarRocks 服务。

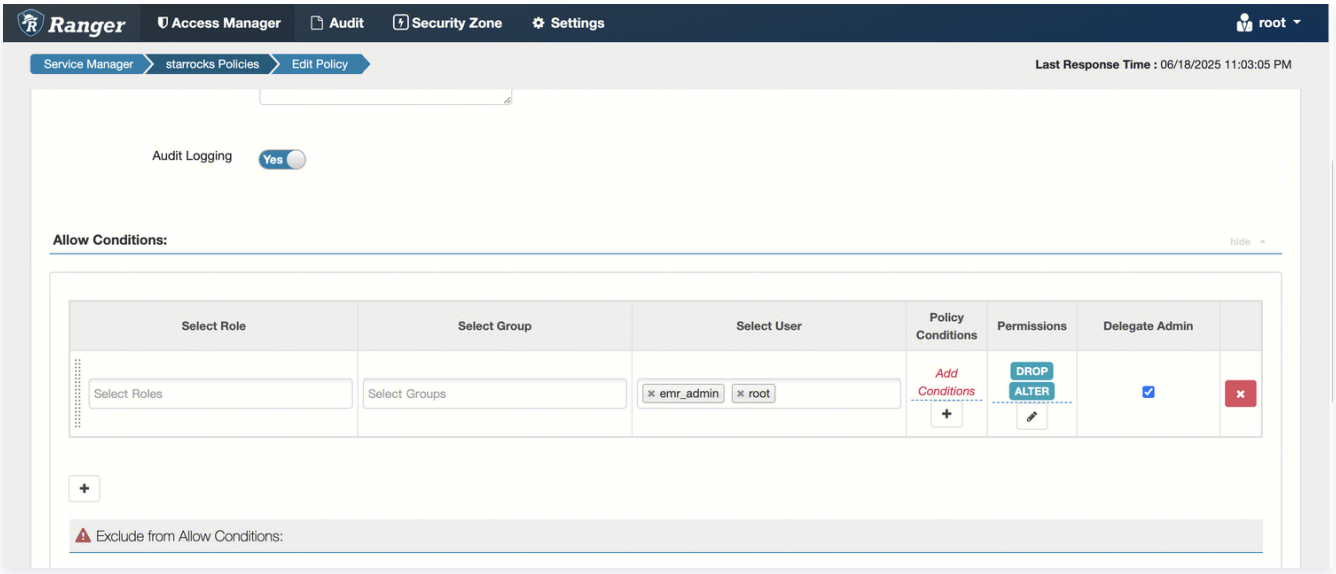


步骤二：配置管理员用户的权限

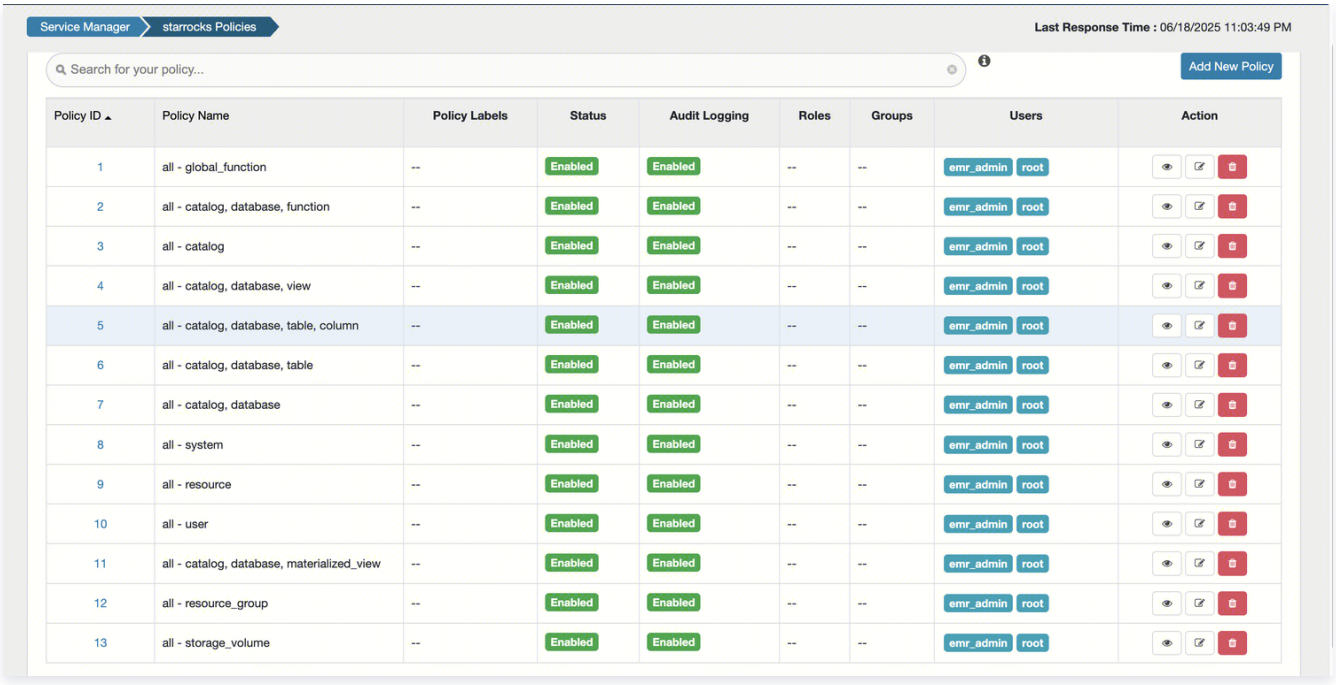
如果您需要使用 Ranger 管理 StarRocks 内表的权限，请在 Ranger Admin 中为管理用户 root 和 admin 授予所有 StarRocks 权限，确保 StarRocks 的管控功能正常使用。
EMR 默认情况下已创建好以下用户并授权完成。以下以 root 用户授权为例。



1. 在 Ranger UI 页面，单击上方的 Access Manager。
2. 在 Service Manager 页面，单击创建好的 StarRocks 服务进入 Poilcy 管理页面。
3. 单击每项策略 Action 列的图标。
4. 在 Allow Conditions 区域的 admin 处添加 root 用户（系统已默认配置）。



确保 root 用户拥有对数据库、表、列等对象的完全控制权限。



步骤三：前往 StarRocks 完成集成 Ranger配置

EMR 的开箱参数已经完成了 Ranger 中 StarRocks Service 名称为 starrocks 的相关配置，建议使用系统默认配置，如需修改请参考以下内容。

配置文件名称	参数	说明
ranger-starrocks-security.xml	ranger.plugin.starrocks.service.name	Ranger 中对应的 StarRocks 服务名称。例如 starrocks。
	ranger.plugin.starrocks.policy.source.impl	获取 Ranger 鉴权策略使用的类。填写为 org.apache.ranger.admin.client.RangerAdminRESTClient。
	ranger.plugin.starrocks.policy.rest.url	Ranger Admin 的地址，可以通过进入 Ranger 查看 WebUI 地址。
fe.conf	access_control	用于 StarRocks 的内表鉴权方式。填写为 ranger。

步骤四：复用其他 Service 来为外表鉴权

对于 External Catalog，可以复用外部 Service（如 Hive Service）实现访问控制。StarRocks 支持对于不同的 Catalog 匹配不同的 Ranger service。用户访问外表时，会直接根据对应外表的 Service 来进行访问控制。用户权限与 Ranger 同名用户一致。

1. 复制您 Hive 集群重的的 Ranger 相关配置文件 ranger-hive-security.xml 和 ranger-hive-audit.xml 拷贝至所有 FE 机器的 fe/conf 文件下，请确保配置文件中 Ranger 的 IP 以及端口正确。
2. 重启所有 FE。
3. 配置 Catalog。
   - 创建 External Catalog 时，添加 PROPERTIES "ranger.plugin.hive.service.name"

```
CREATE EXTERNAL CATALOG hive_catalog_1 PROPERTIES (
"type" = "hive",
"hive.metastore.type" = "hive",
"hive.metastore.uris" = "thrift://xx.xx.xx.xx:9083",
"ranger.plugin.hive.service.name" = "<ranger_hive_service_name>"
)
```

   - 也可以对已有的 External Catalog 添加该属性。将已有的 Catalog 转换为通过 Ranger 鉴权

```
ALTER CATALOG hive_catalog_1 SET ("ranger.plugin.hive.service.name" = "<ranger_hive_service_name>");
```

后续操作

StarRocks 中对于 Ranger 集成的配置请前往 集群服务 > 配置管理 中进行配置。
关于如何在 Ranger 上创建权限策略来管理数据安全，请参见 Apache Ranger 官网。

Ranger 审计日志指南
Ranger 审计日志存入 Solr

最近更新时间：2023-10-12 10:57:01

Ranger 审计日志默认关闭，可根据需要通过修改配置文件打开相应组件的审计功能。

> **说明：**
> EMR-2.7.0及以上版本安装 Ranger 时，默认会在单个 Master 节点上安装单机版的 Solr 服务，用于存储 Ranger 的审计日志，日志保留时间为7天。

1. 打开方法以 HDFS 为例，在 EMR 控制台 > **集群服务 > HDFS > 配置管理**，修改 ranger-hdfs-audit.xml 文件配置项如下：

```
xasecure.audit.is.enabled=true
xasecure.audit.solr.is.enabled=true
```

2. 在**集群服务 > HDFS > 角色管理**，重启 NameNode 角色。
3. 如客户有长期保留 Ranger 审计日志的需求，建议 Ranger 审计日志存入腾讯云 ElasticSearch 。

Ranger 审计日志存入腾讯云 ElasticSearch

最近更新时间：2023-10-12 10:57:01

前提条件

1. 支持 ranger2.1.0及以上版本，对应 EMR-2.7.0、EMR-3.2.1及以上版本。
2. 准备腾讯云ElasticSearch集群，创建过程可参考 ES 创建集群 。
   容量参考：以单个 Hive SQL 大小为1KB为例，单条记录大小约为20KB，即1GB可存储约52428条 Hive SQL。
3. 安全组开放9200端口，仅内网访问要求 EMR 集群和 ElasticSearch 集群在同一 VPC 内。
4. 示例：ElasticSearch 集群关键信息：
   - 用户名：elastic
   - 密码：MyPassword
   - 内网Ip：10.206.48.118
   - 端口：9200
   - 传输协议：http

EMR 控制台操作步骤

1. 登录 EMR 控制台，在集群服务 > RANGER > 配置管理 > ranger-admin-site.xml 文件，修改配置项如下：

```
ranger.audit.source.type = elasticsearch
ranger.audit.elasticsearch.user = elastic
ranger.audit.elasticsearch.password = MyPassword
ranger.audit.elasticsearch.urls = 10.206.48.118
ranger.audit.elasticsearch.protocol = httpot
ranger.audit.elasticsearch.port = 9200
ranger.audit.elasticsearch.index = ranger_audits
ranger.audit.elasticsearch.bootstrap.enabled = true
```

2. 重启 Ranger 服务角色 EmbeddedServer。
3. 以 HIVE 为例，其它服务类似。进入**集群服务 > HIVE > 配置管理**，修改 ranger-hive-audit.xml文件 配置项如下，相应值和 ranger-admin-site.xml 文件保持一致。

```
ranger.audit.source.type = elasticsearch
ranger.audit.elasticsearch.user = elastic
ranger.audit.elasticsearch.password = MyPassword
ranger.audit.elasticsearch.urls = 10.206.48.118
ranger.audit.elasticsearch.protocol = httpot
ranger.audit.elasticsearch.port = 9200
ranger.audit.elasticsearch.index = ranger_audits
ranger.audit.elasticsearch.bootstrap.enabled = true
```

4. 重启 HIVE 角色 HiveServer2。

Kafka 开发指南
Kafka 简介

最近更新时间：2022-06-10 11:14:29

腾讯云弹性 MapReduce（EMR）- Kafka 提供开源 Kafka 的云上托管服务，提供了便捷的 Kafka 集群部署、配置修改、监控告警等功能，为企业及用户提供安全稳定的 OLAP 解决方案。Kafka 数据管道是流计算系统中最常用的数据源（Source）和数据目的（Sink）。用户可以把流数据导入到 Kafka 的某个 Topic 中，通过 Flink 算子进行处理后，输出到相同或不同 Kafka 示例的另一个 Topic。Kafka 支持同一个 Topic 多分区读写，数据可以从多个分区读入，也可以写入到多个分区，以提供更高的吞吐量，减少数据倾斜和热点。

架构

支持单节点、多节点架构。根据业务需求，灵活选择。

运维

在控制台提供了开箱即用的监控、日志检索、参数调整等服务。

功能

- 收发解耦：有效解耦生产者、消费者之间的关系。在确保同样的接口约束的前提下，允许独立扩展或修改生产者/消费者间的处理过程。
- 削峰填谷：Kafka 集群能够抵挡突增的访问压力，不会因为突发的超负荷的请求而完全崩溃，有效提升系统健壮性。
- 顺序读写：Kafka集群能够保证一个 Partition 内消息的有序性。和大部分的消息队列一致，Kafka 集群可以保证数据按照顺序进行处理，极大提升磁盘效率。
- 异步通信：在业务无需立即处理消息的场景下，消息队列 Kafka 集群提供了消息的异步处理机制，访问量高时仅将消息放入队列中，在访问量降低后再对消息进行处理，缓解系统压力。

优势

100%兼容开源，轻松迁移

- Kafka 集群兼容开源 Kafka 1.1.1 版本。
- Kafka 集群业务系统基于现有的开源 Apache Kafka 生态的代码，无需任何改造，即可迁移上云，享受到腾讯云提供的高性能消息队列 Kafka 服务。

高性能

- 腾讯云专业团队对服务性能进一步调优，免除复杂的参数配置，提供更高性能。
- 界面化升降配能力，高性能 IaaS 层支撑。

高可用性

- 依托腾讯技术工程多年监控平台的技术积累，对集群全方位多角度监控，更有专业运维团队7 × 24小时处理告警保障 Kafka 集群服务的高可用性。
- 支持同地域自定义多可用区部署，提升容灾能力。

高可靠性

- 磁盘高可靠，即使服务器坏盘50%也不影响业务。
- 默认2副本，支持3副本，副本越多可靠性越高。

应用场景

最近更新时间：2022-01-05 11:06:39

网页行为分析

Kafka 集群通过实时处理网站活动（PV、搜索、用户其他活动等），并根据类型发布到 Topic 中，这些信息流可以被用于实时监控或离线统计分析等。
由于每个用户的 page view 中会生成许多活动信息，因此网站活动跟踪需要很高的吞吐量，Kafka 集群可以完美满足高吞吐、离线处理等要求。

日志聚合

Kafka 集群的低延迟处理特性，易于支持多个数据源和分布式的数据处理（消费）。相比于中心化的日志聚合系统，消息队列 Kafka 可以在提供同样性能的条件下，实现更强的持久化保证以及更低的端到端延迟。
Kafka 集群的特性决定它非常适合作为日志收集中心。多台主机/应用可以将操作日志批量异步地发送到 Kafka 集群，而无需保存在本地或者 DB 中。Kafka 集群可以批量提交消息/压缩消息，对于生产者而言，几乎感觉不到性能的开支。此时消费者可以使用 Hadoop 等其他系统化的存储和分析系统，对拉取日志进行统计分析。

在线/离线分析

在一些大数据相关的业务场景中，需要对大量并发数据进行处理和汇总，此时对集群的处理性能和扩展性都有很高的要求。Kafka 集群在实现上的数据分发机制，包括磁盘存储空间的分配、消息格式的处理、服务器选择以及数据压缩等方面，也决定了其适合处理海量的实时消息，并能汇总分布式应用的数据，方便系统运维。
在具体的大数据场景中，Kafka 集群能够很好地支持离线数据、流式数据的处理，并能够方便地进行数据聚合、分析等操作。

Kafka 使用

最近更新时间：2024-08-23 14:52:21

生成数据

Java 代码方式

```
@Component
@Slf4j
public class KafkaProducer {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    //自定义topic
    public static final String TOPIC_TEST = "topic.test";

    //
    public static final String TOPIC_GROUP1 = "topic.group1";

    //
    public static final String TOPIC_GROUP2 = "topic.group2";

    public void send(Object obj) {
        String obj2String = JSONObject.toJSONString(obj);
        log.info("准备发送消息为：{}", obj2String);
        //发送消息
        ListenableFuture<SendResult<String, Object>> future = kafkaTemplate.send(TOPIC_TEST, obj);
        future.addCallback(new ListenableFutureCallback<SendResult<String, Object>>() {
            @Override
            public void onFailure(Throwable throwable) {
                //发送失败的处理
                log.info(TOPIC_TEST + " - 生产者 发送消息失败：" + throwable.getMessage());
            }

            @Override
            public void onSuccess(SendResult<String, Object> stringObjectSendResult) {
                //成功的处理
                log.info(TOPIC_TEST + " - 生产者 发送消息成功：" + stringObjectSendResult.toString());
            }
        });
    }
}
```

命令方式

```
bin/kafka-console-producer.sh --broker-list node86:9092 --topic t_cdr
```

消费数据

Java 代码方式

```
@Component
@Slf4j
public class KafkaConsumer {

    @KafkaListener(topics = KafkaProducer.TOPIC_TEST, groupId = KafkaProducer.TOPIC_GROUP1)
    public void topic_test(ConsumerRecord<?, ?> record, Acknowledgment ack,
@Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {
```

```
        Optional message = Optional.ofNullable(record.value());
        if (message.isPresent()) {
            Object msg = message.get();
            log.info("topic_test 消费了:  Topic:" + topic + ",Message:" + msg);
            ack.acknowledge();
        }
    }

    @KafkaListener(topics = KafkaProducer.TOPIC_TEST, groupId = KafkaProducer.TOPIC_GROUP2)
    public void topic_test1(ConsumerRecord<?, ?> record, Acknowledgment ack,
@Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {

        Optional message = Optional.ofNullable(record.value());
        if (message.isPresent()) {
            Object msg = message.get();
            log.info("topic_test1 消费了:  Topic:" + topic + ",Message:" + msg);
            ack.acknowledge();
        }
    }
}
```

命令方式

```
bin/kafka-console-consumer.sh --zookeeper node01:2181 --topic t_cdr --from-beginning
```

新增 topic（命令方式）

```
bin/kafka-topics.sh --zookeeper node01:2181 --create --topic t_cdr --partitions 30  --replication-factor 2
```

详细使用可参考 kafka 官方文档 。

Iceberg 开发指南
Iceberg 基础使用

最近更新时间：2024-06-04 11:37:01

Iceberg 简介

Apache Iceberg 是一种新型的用于大规模数据分析的开源表格式。它被设计用于存储移动缓慢的大型表格数据。它旨在改善 Hive、Trino（PrestoSQL）和 Spark 中内置的事实上的标准表布局。Iceberg 可以屏蔽底层数据存储格式上的差异，向上提供统一的操作 API，使得不同的引擎可以通过其提供的 API 接入。
Apache Iceberg 具备以下能力：

- 模式演化（Schema evolution）：支持 Add（添加）、Drop（删除）、Update（更新）、Rename（重命名）和 Reorder（重排）表格式定义。
- 分区布局演变（Partition layout evolution）：可以随着数据量或查询模式的变化而更新表的布局。
- 隐式分区（Hidden partitioning）：查询不再取决于表的物理布局。通过物理和逻辑之间的分隔，Iceberg 表可以随着数据量的变化和时间的推移发展分区方案。错误配置的表可以得到修复，无需进行昂贵的迁移。
- 时光穿梭（Time travel）：支持用户使用完全相同的快照进行重复查询，或者使用户轻松检查更改。
- 版本回滚（Version rollback）：使用户可以通过将表重置为良好状态来快速纠正问题。

在可靠性与性能方面，Iceberg 可在生产中应用到数十 PB 的数据表，即使没有分布式 SQL 引擎，也可以读取这些巨大规模的表：

- 扫描速度快，无需使用分布式 SQL 引擎即可读取表或查找文件。
- 高级过滤，基于表元数据，使用分区和列级统计信息对数据文件进行裁剪。

Iceberg 被设计用来解决最终一致的云对象存储中的正确性问题：

- 可与任何云存储一起使用，并且通过避免调用 list 和 rename 来减少 HDFS 的 NameNode 拥塞。
- 可序列化的隔离，表更改是原子性的，用户永远不会看到部分更改或未提交的更改。
- 多个并发写入使用乐观锁机制进行并发控制，即使写入冲突，也会重试以确保兼容更新成功。

Iceberg 设计为以快照（Snapshot）的形式来管理表的各个历史版本数据。快照代表一张表在某个时刻的状态。每个快照中会列出表在某个时刻的所有数据文件列表。Data 文件存储在不同的 Manifest 文件中，Manifest 文件存储在一个 Manifest List 文件中，Manifest 文件可以在不同的 Manifest List 文件间共享，一个 Manifest List 文件代表一个快照。

- Manifest list 文件是元数据文件，其中存储的是 Manifest 文件的列表，每个 Manifest 文件占据一行。
- Manifest 文件是元数据文件，其中列出了组成某个快照的数据文件列表。每行都是每个数据文件的详细描述，包括数据文件的状态、文件路径、分区信息、列级别的统计信息（例如每列的最大最小值、空值数等）、文件的大小以及文件中数据的行数等信息。
- Data 文件是 Iceberg 表真实存储数据的文件，一般是在表的数据存储目录的 data 目录下。

使用示例

更多示例可参考 Iceberg 官网示例 。
本文以 EMR- V3.3.0中的 Iceberg0.11.0版本为示例，不同EMR版本相关jar包名称可能有所差异，请您根据路径下实际名称取用。

1. 登录 master 节点，切换为 hadoop 用户。
2. Iceberg 相关的包放置在 `/usr/local/service/iceberg/` 下面。
3. 使用计算引擎查询数据。

- Spark 引擎
  - Spark-SQL 交互式命令行

```
spark-sql --master local[*] --conf
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions --conf
spark.sql.catalog.local=org.apache.iceberg.spark.SparkCatalog --conf
spark.sql.catalog.local.type=hadoop --conf spark.sql.catalog.local.warehouse=/usr/hive/warehouse --jars
/usr/local/service/iceberg/iceberg-spark3-runtime-0.11.0.jar
```

  - 插入和查询数据

```
CREATE TABLE local.default.t1 (id int, name string) USING iceberg;
INSERT INTO local.default.t1 values(1, "tom");
SELECT * from local.default.t1;
```

- Hive 引擎

- 使用 beeline

```
beeline -u jdbc:hive2://[hiveserver2_ip:hiveserver2_port] -n hadoop --hiveconf
hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat --hiveconf hive.stats.autogather=false
```

- 查询数据

```
ADD JAR /usr/local/service/iceberg/iceberg-hive-runtime-0.11.0.jar;
CREATE EXTERNAL TABLE t1 STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler' LOCATION
'/usr/hive/warehouse/default/t1' TBLPROPERTIES ('iceberg.catalog'='location_based_table');

select count(*) from t1;
```

- Flink 引擎
  - 根据 Flink 和 Hive 版本在 Maven 仓库 下载相应版本 flink-sql-connector-hive 包，以 Flink standalone 模式为例，并使用 Flink shell 交互式命令行。

```
wget https://repo1.maven.org/maven2/org/apache/flink/flink-sql-connector-hive-3.1.2_2.11/1.12.1/flink-
sql-connector-hive-3.1.2_2.11-1.12.1.jar
/usr/local/service/flink/bin/start-cluster.sh
sql-client.sh embedded -j /usr/local/service/iceberg/iceberg-flink-runtime-0.11.0.jar -j flink-sql-
connector-hive-3.1.2_2.11-1.12.1.jar shell
```

  - 查询数据

```
CREATE CATALOG hive_catalog WITH ('type'='iceberg','catalog-
type'='hive','uri'='hivemetastore_ip:hivemetastore_port','clients'='5','property-
version'='1','warehouse'='hdfs:///usr/hive/warehouse/');
CREATE DATABASE hive_catalog.iceberg_db;
CREATE TABLE hive_catalog.iceberg_db.t1 (id BIGINT COMMENT 'unique id',data STRING);
INSERT INTO hive_catalog.iceberg_db.t1 values(1, 'tom');
SELECT count(*) from hive_catalog.iceberg_db.t1;
```

Iceberg 引擎集成

最近更新时间：2025-04-14 14:31:02

本文为您演示如何在 EMR 上使 Hive、Spark、Flink 引擎支持 Iceberg 表。

开发准备

- 确认您已经开通了腾讯云，并且创建了一个 EMR 集群，详情参考 创建集群 。
- 在创建 EMR 集群时在软件配置界面选择 Iceberg 组件，并按照需求勾选集成的 Spark、Hive、Flink 引擎。

Hive 集成

方式一：临时添加依赖环境

使用 Hive 命令启动 Hive，使用 add 命令添加 Hive 使用 Iceberg 依赖所需要的 jar 包：

```
//先启动hive引擎，在hive命令行添加依赖包
hive

ADD JAR {iceberg-hive-runtime};
```

{iceberg-hive-runtime} 是依赖 jar 包的绝对路径，jar 包位于 /usr/local/service/iceberg 目录下。

> ⚠ 注意：
> 使用临时添加依赖环境的方法，每次启动 Hive 时都需要使用 add 命令添加一次依赖。

方式二：使用额外 lib 目录

将 Hive 使用 Iceberg 依赖所需要的 jar 包拷贝到 auxlib 目录：

```
cp {iceberg-hive-runtime} /usr/local/service/hive/auxlib
```

将 Iceberg 依赖添加至 Hive 中之后，再启动 Hive 即可使用 Iceberg 表格。

Spark 集成

将 Spark 使用 Iceberg 依赖所需要的 jar 包拷贝到 jars 目录：

```
cp {iceberg-spark-runtime} /usr/local/service/spark/jars
```

{iceberg-spark-runtime} 是依赖 jar 包的绝对路径，jar 包位于 /usr/local/service/iceberg 目录下。
使用以下命令启动支持 Iceberg 表的 Spark sql 命令行：

```
spark-sql --conf spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions \
    --conf spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog \
    --conf spark.sql.catalog.spark_catalog.type=hive \
```

将 Iceberg 配置写入配置文件

可以通过将 Iceberg 配置写入配置文件的方式，让 Spark sql 默认支持 Iceberg。
进入EMR控制台，点击 **集群服务 > SPARK > 配置管理** 页面，选择 spark-defaults.conf 配置文件，新增以下参数值：

```
spark.sql.extensions = org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions
spark.sql.catalog.spark_catalog = org.apache.iceberg.spark.SparkSessionCatalog
spark.sql.catalog.spark_catalog.type = hive
```

> ⚠ 注意：

> 这里默认配置了 Hive 类型的 Catalog，其他类型的 Catalog 配置参考 官方文档 。

Flink 集成

将 Flink 使用 Iceberg 依赖所需要的 jar 包拷贝到 auxlib 目录。

```
cp {iceberg-flink-runtime} /usr/local/service/flink/lib
cp /usr/local/service/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-3.2.2.jar
/usr/local/service/flink/lib/
cp /usr/local/service/hive/lib/hive-exec-3.1.3.jar /usr/local/service/flink/lib/
cp /usr/local/service/hive/lib/libfb303-0.9.3.jar /usr/local/service/flink/lib/

//启动支持iceberg表的flink sql命令行
./bin/sql-client.sh embedded shell
```

{iceberg-flink-runtime} 是依赖 jar 包的绝对路径，jar 包位于 /usr/local/service/iceberg 目录下。
{hadoop-mapreduce-client-core-3.2.2.jar}、{hive-exec-3.1.3.jar}、{libfb303-0.9.3.jar}三个 jar 包需要更改为对应的版本。

StarRocks 开发指南
StarRocks 简介

最近更新时间：2023-07-27 15:06:46

StarRocks 是什么

- StarRocks 是新一代极速全场景 MPP 数据库。
- StarRocks 充分吸收关系型 OLAP 数据库和分布式存储系统在大数据时代的优秀研究成果，在业界实践的基础上，进一步改进优化、升级架构，并增添了众多全新功能，形成了全新的企业级产品。
- StarRocks 致力于构建极速统一分析体验，满足企业用户的多种数据分析场景，支持多种数据模型(明细模型、聚合模型、更新模型)，多种导入方式（批量和实时），支持导入多达10000列的数据，可整合和接入多种现有系统(Spark、Flink、Hive、 ElasticSearch)。
- StarRocks 兼容 MySQL 协议，可使用 MySQL 客户端和常用 BI 工具对接 StarRocks 来进行数据分析。
- StarRocks 采用分布式架构，对数据表进行水平划分并以多副本存储。集群规模可以灵活伸缩，能够支持10PB 级别的数据分析; 支持 MPP 框架，并行加速计算; 支持多副本，具有弹性容错能力。
- StarRocks 采用关系模型，使用严格的数据类型和列式存储引擎，通过编码和压缩技术，降低读写放大；使用向量化执行方式，充分挖掘多核 CPU 的并行计算能力，从而显著提升查询性能。

StarRocks 特性

StarRocks 的架构设计融合了 MPP 数据库，以及分布式系统的设计思想，具有以下特性：

架构精简

StarRocks 内部通过 MPP 计算框架完成 SQL 的具体执行工作。MPP 框架本身能够充分的利用多节点的计算能力，整个查询并行执行，从而实现良好的交互式分析体验。 StarRocks 集群不需要依赖任何其他组件，易部署、易维护，极简的架构设计，降低了 StarRocks 系统的复杂度和维护成本，同时也提升了系统的可靠性和扩展性。 管理员只需要专注于 StarRocks 系统，无需学习和管理任何其他外部系统。

全面向量化引擎

StarRocks 的计算层全面采用了向量化技术，将所有算子、函数、扫描过滤和导入导出模块进行了系统性优化。通过列式的内存布局、适配 CPU 的 SIMD 指令集等手段，充分发挥了现代 CPU 的并行计算能力，从而实现亚秒级别的多维分析能力。

智能查询优化

StarRocks 通过 CBO 优化器(Cost Based Optimizer)可以对复杂查询自动优化。无需人工干预，就可以通过统计信息合理估算执行成本，生成更优的执行计划，大大提高了 Adhoc 和 ETL 场景的数据分析效率。

联邦查询

StarRocks 支持使用外表的方式进行联邦查询，当前可以支持 Hive、MySQL、Elasticsearch 三种类型的外表，用户无需通过数据导入，可以直接进行数据查询加速。

高效更新

StarRocks 支持多种数据模型，其中更新模型可以按照主键进行 upsert/delete 操作，通过存储和索引的优化可以在并发更新的同时实现高效的查询优化，更好的服务实时数仓的场景。

智能物化视图

StarRocks 支持智能的物化视图。用户可以通过创建物化视图，预先计算生成预聚合表用于加速聚合类查询请求。StarRocks 的物化视图能够在数据导入时自动完成汇聚，与原始表数据保持一致。并且在查询的时候，用户无需指定物化视图，StarRocks 能够自动选择最优的物化视图来满足查询请求。

标准 SQL

StarRocks 支持标准的 SQL 语法，包括聚合、JOIN、排序、窗口函数和自定义函数等功能。StarRocks 可以完整支持 TPC-H 的22个 SQL 和 TPC-DS 的99个 SQL。此外，StarRocks还兼容 MySQL 协议语法，可使用现有的各种客户端工具、BI 软件访问 StarRocks，对 StarRocks 中的数据进行拖拽式分析。

流批一体

StarRocks 支持实时和批量两种数据导入方式，支持的数据源有 Kafka、HDFS、本地文件，支持的数据格式有 ORC、Parquet 和 CSV 等，支持导入多达10000列的数据。StarRocks 可以实时消费 Kafka 数据来完成数据导入，保证数据不丢不重（exactly once）。StarRocks 也可以从本地或者远程（HDFS）批量导入数据。

高可用易扩展

StarRocks 的元数据和数据都是多副本存储，并且集群中服务有热备，多实例部署，避免了单点故障。集群具有自愈能力，可弹性恢复，节点的宕机、下线、异常都不会影响 StarRocks 集群服务的整体稳定性。 StarRocks 采用分布式架构，存储容量和计算能力可近乎线性水平扩展。StarRocks 单集群的节点规模可扩展到数百节点，数据规模可达到10PB 级别。 扩缩容期间无需停服，可以正常提供查询服务。 另外StarRocks 中表模式热变更，可通过一条简单 SQL 命令动态地修改表的定义，例如增加列、减少列、新建物化视图等。同时，处于模式变更中的表也可以正常导入和查询数据。

StarRocks 适合什么场景

StarRocks 可以满足企业级用户的多种分析需求，包括 OLAP 多维分析、定制报表、实时数据分析和 Ad-hoc 数据分析等。具体的业务场景包括：

- OLAP 多维分析用户行为分析。
  - 用户画像、标签分析、圈人。
  - 高维业务指标报表。
  - 自助式报表平台。
  - 业务问题探查分析。
  - 跨主题业务分析。
  - 财务报表。
  - 系统监控分析。
- 实时数据分析电商大促数据分析。
  - 教育行业的直播质量分析。
  - 物流行业的运单分析。
  - 金融行业绩效分析、指标计算。
  - 广告投放分析。
  - 管理驾驶舱。
  - 探针分析 APM（Application Performance Management）。
- 高并发查询广告主报表分析。
  - 零售行业渠道人员分析。
  - SaaS 行业面向用户分析报表。
  - Dashbroad 多页面分析。
- 统一分析通过使用一套系统解决多维分析、高并发查询、预计算、实时分析、Adhoc查询等场景，降低系统复杂度和多技术栈开发与维护成本。

基础使用指南

最近更新时间：2023-12-20 17:06:31

目前 StarRocks 支持多种方式连接，下面简介使用 mysql 客户端连接 StarRocks 进行开发。

Root 用户登录

使用 MySQL 客户端连接某一个 FE 实例的 query_port(9030), StarRocks 内置 root 用户，密码默认与集群密码相同：

```
mysql -h fe_host -P9030 -u root -p
```

清理环境：

```
mysql > drop database if exists example_db;

mysql > drop user test;
```

查看部署节点

1. 查看 FE 节点。

```
mysql> SHOW PROC '/frontends'\G

*************************** 1. row ***********************
          Name: 172.16.139.24_9010_1594200991015
            IP: 172.16.139.24
      HostName: starrocks-sandbox01
   EditLogPort: 9010
      HttpPort: 8030
     QueryPort: 9030
       RpcPort: 9020
          Role: FOLLOWER
      IsMaster: true
     ClusterId: 861797858
          Join: true
         Alive: true
ReplayedJournalId: 64
 LastHeartbeat: 2020-03-23 20:15:07
      IsHelper: true
        ErrMsg:
1 row in set (0.03 sec)
```

   Role 为 FOLLOWER 说明这是一个能参与选主的 FE；IsMaster 为 true，说明该 FE 当前为主节点。

2. 查看 BE 节点。

```
mysql> SHOW PROC '/backends'\G

*************************** 1. row ***********************
         BackendId: 10002
           Cluster: default_cluster
                IP: 172.16.139.24
          HostName: starrocks-sandbox01
     HeartbeatPort: 9050
            BePort: 9060
          HttpPort: 8040
          BrpcPort: 8060
     LastStartTime: 2020-03-23 20:19:07
     LastHeartbeat: 2020-03-23 20:34:49
             Alive: true
```

```
 SystemDecommissioned: false
ClusterDecommissioned: false
        TabletNum: 0
  DataUsedCapacity: .000
     AvailCapacity: 327.292 GB
     TotalCapacity: 450.905 GB
           UsedPct: 27.41 %
    MaxDiskUsedPct: 27.41 %
            ErrMsg:
           Version:
1 row in set (0.01 sec)
```

如果 isAlive 为 true，则说明 BE 正常接入集群。如果 BE 没有正常接入集群，请查看 log 目录下的 be.WARNING 日志文件确定原因。

3. 查看 Broker 节点。

```
MySQL> SHOW PROC "/brokers"\G
*************************** 1. row ***************************
       Name: broker1
         IP: 172.16.139.24
       Port: 8000
      Alive: true
 LastStartTime: 2020-04-01 19:08:35
LastUpdateTime: 2020-04-01 19:08:45
     ErrMsg:
1 row in set (0.00 sec)
```

Alive 为 true 代表状态正常。

创建新用户

通过下面的命令创建一个普通用户：

```
mysql > create user 'test' identified by '123456';
```

创建数据库

StarRocks 中 root 账户才有权建立数据库，使用 root 用户登录，建立 example_db 数据库：

```
mysql > create database example_db;
```

数据库创建完成之后，可以通过 show databases 查看数据库信息：

```
mysql > show databases;

+--------------------+
| Database           |
+--------------------+
| example_db         |
| information_schema |
+--------------------+
2 rows in set (0.00 sec)
```

information_schema 是为了兼容 mysql 协议而存在，实际中信息可能不是很准确，所以关于具体数据库的信息建议通过直接查询相应数据库而获得。

账户授权

example_db 创建完成之后，可以通过 root 账户 example_db 读写权限授权给 test 账户，授权之后采用 test 账户登录就可以操作 example_db 数据库了：

```
mysql > grant all on example_db to test;
```

退出 root 账户，使用 test 登录 StarRocks 集群：

```
mysql > exit

mysql -h 127.0.0.1 -P9030 -utest -p123456
```

建表

StarRocks 支持单分区和复合分区两种建表方式。
在复合分区中：

- 第一级称为 Partition，即分区。用户可以指定某一维度列作为分区列（当前只支持整型和时间类型的列），并指定每个分区的取值范围。
- 第二级称为 Distribution，即分桶。用户可以指定某几个维度列（或不指定，即所有 KEY 列）以及桶数对数据进行 HASH 分布。

以下场景推荐使用复合分区：

- 有时间维度或类似带有有序值的维度：可以以这类维度列作为分区列。分区粒度可以根据导入频次、分区数据量等进行评估。
- 历史数据删除需求：如有删除历史数据的需求（例如仅保留最近 N天的数据）。使用复合分区，可以通过删除历史分区来达到目的。也可以通过在指定分区内发送 DELETE 语句进行数据删除。
- 解决数据倾斜问题：每个分区可以单独指定分桶数量。如按天分区，当每天的数据量差异很大时，可以通过指定分区的分桶数，合理划分不同分区的数据,分桶列建议选择区分度大的列。

用户也可以不使用复合分区，即使用单分区。则数据只做 HASH 分布。
下面分别演示两种分区的建表语句：

1. 首先切换数据库：mysql > use example_db;
2. 建立单分区表建立一个名字为 table1的逻辑表。使用全 hash 分桶，分桶列为 siteid，桶数为10。这个表的 schema 如下：
   - siteid：类型是 INT（4字节），默认值为10。
   - city_code：类型是 SMALLINT（2字节）。
   - username：类型是 VARCHAR，最大长度为32，默认值为空字符串。
   - pv：类型是 BIGINT（8字节），默认值是0；这是一个指标列, StarRocks 内部会对指标列做聚合操作，这个列的聚合方法是求和（SUM）。这里采用了聚合模型，除此之外StarRocks 还支持明细模型和更新模型，具体参考 数据模型介绍 。
     建表语句如下：

     ```
     mysql >
     CREATE TABLE table1
     (
      siteid INT DEFAULT '10',
      citycode SMALLINT,
      username VARCHAR(32) DEFAULT '',
      pv BIGINT SUM DEFAULT '0'
     )
     AGGREGATE KEY(siteid, citycode, username)
     DISTRIBUTED BY HASH(siteid) BUCKETS 10
     PROPERTIES("replication_num" = "1");
     ```

3. 建立复合分区表
   建立一个名字为 table2的逻辑表。这个表的 schema 如下：
   - event_day：类型是 DATE，无默认值。
   - siteid：类型是 INT（4字节），默认值为10。
   - city_code：类型是 SMALLINT（2字节）。
   - username：类型是 VARCHAR, 最大长度为32, 默认值为空字符串。
   - pv：类型是 BIGINT（8字节），默认值是0，这是一个指标列，StarRocks 内部会对指标列做聚合操作，这个列的聚合方法是求和（SUM）。
     我们使用 event_day 列作为分区列，建立3个分区：p1, p2, p3。
   - p1：范围为 [最小值, 2017-06-30)。
   - p2：范围为 [2017-06-30, 2017-07-31)。
   - p3：范围为 [2017-07-31, 2017-08-31)。

每个分区使用 siteid 进行哈希分桶，桶数为10。
建表语句如下：

```
CREATE TABLE table2
(
event_day DATE,
siteid INT DEFAULT '10',
citycode SMALLINT,
username VARCHAR(32) DEFAULT '',
pv BIGINT SUM DEFAULT '0'
)
AGGREGATE KEY(event_day, siteid, citycode, username)
PARTITION BY RANGE(event_day)
(
PARTITION p1 VALUES LESS THAN ('2017-06-30'),
PARTITION p2 VALUES LESS THAN ('2017-07-31'),
PARTITION p3 VALUES LESS THAN ('2017-08-31')
)
DISTRIBUTED BY HASH(siteid) BUCKETS 10
PROPERTIES("replication_num" = "1");
```

表建完之后，可以查看 example_db 中表的信息：

```
mysql> show tables;

+-------------------------+
| Tables_in_example_db    |
+-------------------------+
| table1                  |
| table2                  |
+-------------------------+
2 rows in set (0.01 sec)


mysql> desc table1;

+----------+-------------+------+-------+---------+-------+
| Field    | Type        | Null | Key   | Default | Extra |
+----------+-------------+------+-------+---------+-------+
| siteid   | int(11)     | Yes  | true  | 10      |       |
| citycode | smallint(6) | Yes  | true  | N/A     |       |
| username | varchar(32) | Yes  | true  |         |       |
| pv       | bigint(20)  | Yes  | false | 0       | SUM   |
+----------+-------------+------+-------+---------+-------+
4 rows in set (0.00 sec)


mysql> desc table2;

+-----------+-------------+------+-------+---------+-------+
| Field     | Type        | Null | Key   | Default | Extra |
+-----------+-------------+------+-------+---------+-------+
| event_day | date        | Yes  | true  | N/A     |       |
| siteid    | int(11)     | Yes  | true  | 10      |       |
| citycode  | smallint(6) | Yes  | true  | N/A     |       |
| username  | varchar(32) | Yes  | true  |         |       |
| pv        | bigint(20)  | Yes  | false | 0       | SUM   |
+-----------+-------------+------+-------+---------+-------+
5 rows in set (0.00 sec)
```

创建 StarRocks 存算分离集群

最近更新时间：2024-08-12 17:04:21

操作场景

本文为您介绍通过 EMR 控制台创建一个 EMR on CVM StarRocks 存算分离集群的操作步骤和相关配置。

操作步骤

登录 EMR 控制台 ，在 EMR on CVM 集群列表页单击**创建集群**，创建集群时，您需要对集群进行软件配置、区域与硬件配置、基础配置和确认配置信息四个步骤。

软件配置

配置项	配置说明
地域	集群所部署的物理数据中心，每个地域（region）都指一个独立的物理数据中心，不同地域间的云服务器内网不互通。注意：集群创建后，无法更改地域，请谨慎选择。
集群类型	选择 StarRocks 集群类型，更多集群类型介绍请参见 集群类型 。
应用场景	选择存算分离，StarRocks 提供存算一体、存算分离两种应用场景选择，更多应用场景介绍请参见 集群类型 。
产品版本	选择 StarRocks 产品版本，不同产品版本上捆绑的组件和组件的版本不同。
部署组件	无需选择，StarRocks 集群类型暂未提供非必选组件。
COS存储桶	选择 COS 对象存储桶，用于存放存算分离场景下 StarRocks 的内表数据。

注意：

1、创建 StarRocks 存算分离时，如果您选择的 COS 存储桶开启了 元数据加速 功能，集群创建后，请前往对象存储控制台，对 StarRocks 集群 VPC 网络及节点 IP 进行授权，想请参考 使用 HDFS 协议访问已开启元数据加速的存储桶 。

2、StarRocks 存算分离集群销毁时，不会同步删除存储在 COS 存储桶中的数据，如果您的集群销毁后存储数据无需保留，请前往对象存储控制台进行删除。

区域与硬件配置

配置项	配置说明
计费模式	选择集群的计费模式，详请参考 计费概述 。
（跨）可用区	StarRocks 集群类型仅支持单可用区部署。
集群网络	为保证 EMR 集群的安全性，我们会将集群各节点放入一个私有网络中，您需要设置一个私有网络以保证 EMR 集群的正确创建。
集群外网	可用于外网登录 SSH 和外网访问组件 WebUI，集群创建成功后可在控制台对该网络进行调整。默认开启 master1 节点的外网。
安全组	安全组具有防火墙功能，用于设置云服务器 CVM 的网络访问控制。如果没有安全组，EMR 会自动帮您新建一个安全组。若已经有在使用的安全组可以直接选择使用。若安全组数量已达到上限无法新建，可删除部分不再使用的安全组。查看已在使用的安全组。 创建安全组：EMR 帮助用户创建一个安全组，开启22和30001端口及必要的内网通信网段。 已有 EMR 安全组：选择已创建的 EMR 安全组作为当前实例的安全组，开启22和30001端口及必要的内网通信网段。
高可用（HA）	默认启动高可用，不同集群类型和应用场景在 HA 或非 HA 场景下，不同节点类型部署数量不同，详情请参见 集群类型 。
节点类型	根据业务需要为不同节点类型选择合适机型配置。详情请参见 业务评估 。 说明： 1. 目前支持 Task 节点和 Router 节点挂载多种云盘类型（每种云盘类型最多只能选择1次）和多块云盘（最多20块）。 2. 本地盘机型不支持部署在 Master 节点上，请选择非本地盘机型。
置放群组	可选配置，放置群组是云服务器实例在底层硬件上分布放置的策略，详情请参见 置放群组 。

基础配置

配置项	配置说明

所属项目	将当前集群分配给不同的项目组注意：集群创建后暂不支持修改所属项目。
集群名称	通过设置集群名称，来区分不同的 EMR 集群。系统随机生成支持修改，集群的名字，长度限制为6-36个字符，只允许包含中文、字母、数字、-、_。
登录方式	目前 EMR 提供两种登录集群服务、节点的方式，自定义设置密码方式和关联密钥方式；SSH 密钥仅用于 EMR-UI 快捷入口登录。其中，用户名默认为“root”。
引导操作	可选配置，引导脚本操作方便您在创建集群的过程中执行自定义脚本，以便您修改集群环境、安装第三方软件和使用自有数据；更多设置请参见 引导操作 。
标签	可选配置，您在创建时对集群或节点资源添加标签，以便于管理集群和节点资源，最多可绑定5条，标签键不可重复。

确认配置

配置项	配置说明
配置清单	确认所部署信息是否有误。
自动续费	可选项，系统将在集群到期前7天，每天检测用户账户上的可用余额是否充足，设置为自动续费的集群资源进行续费；包年包月集群默认勾选自动续费，用户可手动取消勾选。
协议条款	同意《弹性 MapReduce 服务等级协议》和《退费协议》。

完成以上配置后，单击购买进行支付，支付成功后 EMR 集群进入创建过程，集群创建完成后即可在 EMR 控制台中找到新建的集群。

> **注意**
> - 按量计费集群：立刻开始创建。 集群创建完成后，集群的状态变为运行中。
> - 包年包月集群：先生成订单，支付完成订单以后集群才会开始创建。
> - 您可以在 CVM 控制台中查看各节点的实例信息，为保证 EMR 集群的正常运行，请不要在 CVM 控制台中更改这些实例的配置信息。

后续步骤

集群创建成功后，您可根据自身情况登录集群后，对集群进行进一步的配置等操作，具体操作可参考如下文档：

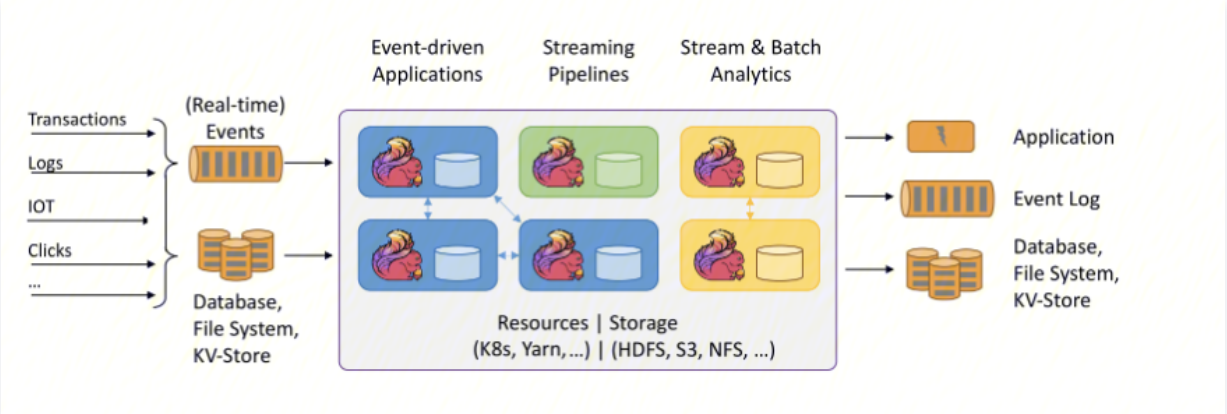
- 管理服务：角色管理 、配置管理 、启停服务 。
- 管理集群：设置标签 、设置引导操作 、集群销毁 。
- 使用 StarRocks 服务：StarRocks 基础使用指南 。

Flink 开发指南
Flink 简介

最近更新时间：2023-12-20 17:06:31

Flink 核心是一个开源的分布式、高性能、高可用、准确的数据流执行引擎，其针对数据流的分布式计算提供了数据分布、数据通信以及容错机制等功能。基于流执行引擎，Flink 提供了更高抽象层的 API 以便您编写分布式任务。

- 分布式：表示 Flink 程序可以运行在多台机器上。
- 高性能：表示 Flink 处理性能比较高。
- 高可用：表示 Flink 支持程序的自动重启机制。
- 准确的：表示 Flink 可以保证处理数据的准确性。



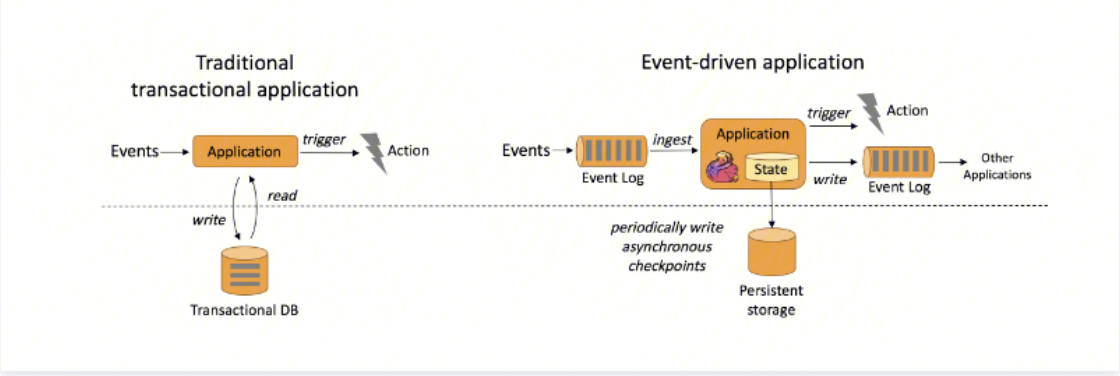
下图中左边是数据源，从这里可以看出来，这些数据是实时生产的一些日志，或者是数据库，文件系统，kv 存储系统中的数据。中间是 Flink，负责对数据进行梳理。右边是目的地，Flink 可以将计算好的数据输出到其它应用系统中，或者存储系统中。Flink 的三大核心组件如下：

- Data Source：也就是图中左边的数据源。
- Transformations：算子（负责对数据进行处理）。
- Data Sink：输出组件（负责把计算好的数据输出到其它应用系统中）。

使用场景

Flink 主要有以下三种应用场景：

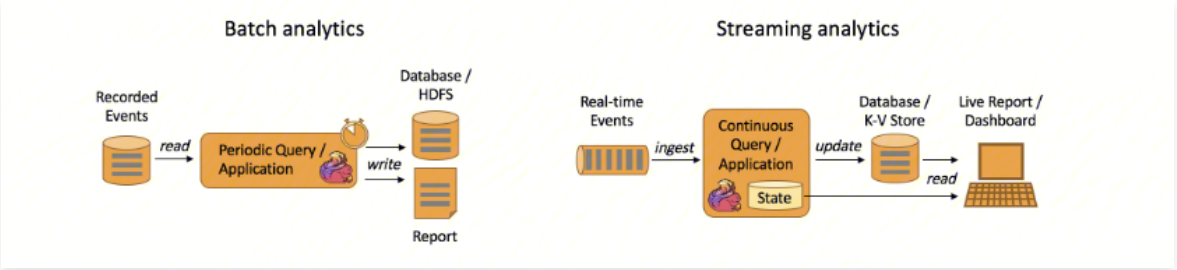
1. 事件驱动型应用
   事件驱动型应用是一类具有状态的应用，它从一个或多个事件流提取数据，并根据到来的事件触发计算、状态更新或其他外部动作。



在传统架构中（图左），我们需要读写远程事务型数据库，例如 MySQL。在事件驱动应用中数据和计算不会分离，应用只需访问本地（内存或磁盘）即可获取数据，所以具有更高的吞吐和更低的延迟。

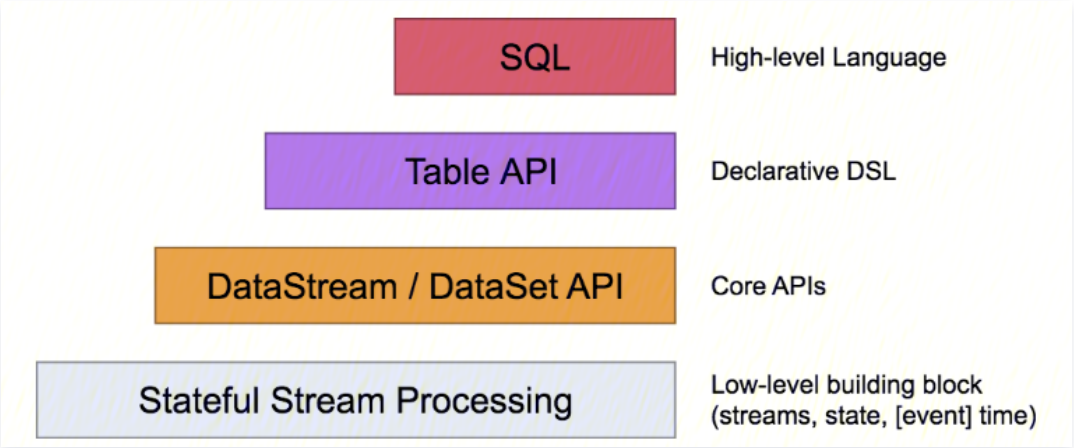
- Flink 的以下特性很好地支持了事件驱动型应用。
- 高效的状态管理，Flink 自带的 State Backend 可以很好的存储中间状态信息。
- 丰富的窗口支持，Flink 支持包含滚动窗口、滑动窗口及其他窗口。
- 多种时间语义，Flink 支持 Event Time、Processing Time 和 Ingestion Time。
  不同级别的容错，Flink 支持 At Least Once 或 Exactly Once 容错级别。

2. 实时数据分析应用：
   数据分析任务需要从原始数据中提取有价值的信息和指标。传统的分析方式通常是利用批查询，或将事件记录下来并基于此有限数据集构建应用来完成。为了得到最新数据的分析结果，必须先将它们加入分析数据集并重新执行查询或运行应用，随后将结果写入存储系统或生成报告。



3. 实时数据仓库和 ETL
   ETL（Extract-Transform-Load）的目的是将业务系统的数据经过抽取、清洗转换之后加载到数据仓库的过程。
   传统的离线数据仓库将业务数据集中进行存储后，以固定的计算逻辑定时进行 ETL 和其他建模后产出报表等应用。离线数据仓库主要是构建 T+1 的离线数据，通过定时任务每天拉取增量数据，然后创建各个业务相关的主题维度数据，对外提供 T+1 的数据查询接口。

上图展示了离线数据仓库 ETL 和实时数据仓库的差异，可以看到离线数据仓库的计算和数据的实时性均较差。数据本身的价值随着时间的流逝会逐步减弱，因此数据发生后必须尽快的达到用户的手中，实时数仓的构建需求也应运而生。
相关分层 API 请参考以下文档：



- Table API & SQL ：Table API 一般与 DataSet 或者 DataStream 紧密关联，可以通过一个 DataSet 或者 DataStream 创建出一个 Table，然后再使用类似 filter、sum、join、select 等这种操作。最近还可以将一个 Table 对象转换成 DataSet 或者 DataStream。SQL API 的底层是基于 Apache Calcite，Apache Calcite 实现了标准 SQL，使用起来比其它 API 更加灵活，因为可以直接使用 SQL 语句。Table API 和 SQL API 可以很容易地结合在一块使用。因为它们都返回 Table 对象。
- DataStream API & DataSet API ：主要提供针对流数据和批数据的处理，是对低级 API 进行了一些封装，提供了 filter、sum、max、min等高阶函数，简单易用，所以这些 API 在实际生产中应用还是比较广泛的。
- Stateful Stream Processing ：提供了对时间和状态的细粒度控制，简洁性和易用性较差，主要应用在一些复杂事件处理逻辑上。

环境信息

- Flink 默认部署在集群的 Master、Core 节点，安装了 Flink 组件的集群其功能都是开箱即用的。
- 登录机器后使用命令 su hadoop 切换到 hadoop 用户进行一些Flink的本地测试。
- Flink 软件路径在 /usr/local/service/flink 下。
- 相关日志路径在 /data/emr/flink/logs 下。

更多详细资料请参考 社区文档 。

Flink 分析 COS 上的数据

最近更新时间：2023-09-26 15:13:11

Flink 擅长处理无界和有界数据集精确的时间控制和状态化使得 Flink 的运行（runtime）能够运行任何处理无界流的应用。有界流则由一些专为固定大小数据集特殊设计的算法和数据结构进行内部处理，产生了出色的性能。
以下针对于在对象存储上的有界或者无界数据集的数据进行演示操作，这里为了更好地观察作业的运行情况使用 yarn-session 模式来提交任务。Flink On Yarn 支持 Session Mode、Application Mode、Per-Job Mode（deprecated in flink 1.15+）三种形式，具体可参考 社区文档 。
本教程演示的是提交的任务为 wordcount 任务即统计单词个数，提前需要在集群中上传需要统计的文件。

开发准备

1. 由于任务中需要访问腾讯云对象存储（COS），所以需要在 COS 中先 创建存储桶 。
2. 确认您已开通腾讯云，且已创建一个 EMR 集群。在创建 EMR 集群的时候需要在软件配置界面选择 Flink 组件，并且在**集群>集群信息**页面开启对象存储的授权。
3. 集群购买完成后，可以使用 HDFS 访问对象存储确保其基础功能可用。具体命令如下：

```
[hadoop@10 ~]$ hdfs dfs -ls cosn://$BUCKET_NAME/path
Found 1 items
-rw-rw-rw-   1 hadoop hadoop      27040 2022-10-28 15:08 cosn://$BUCKET_NAME/path/LICENSE
```

其中，$BUCKET_NAME/path 为您的存储桶名及路径。

示例

```
# -tm 表示每个TaskManager的内存大小
# -s 表示每个TaskManager的slots数量
# -d 表示以后台程序方式运行，后面接的session名字
[hadoop@10 ~]$ yarn-session.sh -jm 1024 -tm 1024 1 -s 1 -nm wordcount-example -d
```

```
/usr/local/service/flink/bin/flink run -m yarn-cluster /usr/local/service/flink/examples/batch/WordCount.jar
--input cosn://$BUCKET_NAME/path/LICENSE -output cosn://$BUCKET_NAME/path/wdp_test
[hadoop@10 ~]$ hdfs dfs -ls cosn://$BUCKET_NAME/path/wdp_test
-rw-rw-rw-   1 hadoop hadoop       7484 2022-11-04 00:47 cosn://$BUCKET_NAME/path/wdp_test
```

Maven 工程示例

在本次演示中，不再采用系统自带的演示程序，而是自己建立工程编译打包之后上传到 EMR 集群运行。推荐您使用 Maven 来管理您的工程。Maven 是一个项目管理工具，能够帮助您方便的管理项目的依赖信息，即它可以通过 pom.xml 文件的配置获取 jar 包，而不用去手动添加。
1. 首先 下载并安装 Maven，配置 Maven 的环境变量。如果您使用 IDE，请在 IDE 中设置 Maven 相关配置。
2. 在本地 shell 下进入您想要新建工程的目录，例如 D://mavenWorkplace 中，输入如下命令新建一个 Maven 工程：

```
mvn archetype:generate -DgroupId=$yourgroupID -DartifactId=$yourartifactID -DarchetypeArtifactId=maven-
archetype-quickstart
```

其中 yourgroupID 即为您的包名。yourartifactID 为您的项目名称，而 maven-archetype-quickstart 表示创建一个 Maven Java 项目。工程创建过程中需要下载一些文件，请保持网络通畅。
创建成功后，在 D://mavenWorkplace 目录下就会生成一个名为 $yourartifactID 的工程文件夹。其中的文件结构如下所示：

```
simple
 ---pom.xml       核心配置，项目根下
 ---src
     ---main
         ---java          Java 源码目录
         ---resources     Java 配置文件目录
     ---test
          ---java          测试源码目录
          ---resources     测试配置目录
```

其中我们主要关心 pom.xml 文件和 main 下的 Java 文件夹。pom.xml 文件主要用于依赖和打包配置，Java 文件夹下放置您的源代码。
首先在 pom.xml 中添加 Maven 依赖：

```
<properties>
 <scala.version>2.12</scala.version>
 <flink.version>1.14.3</flink.version>
</properties>

<dependencies>
 <dependency>
     <groupId>org.apache.flink</groupId>
     <artifactId>flink-java</artifactId>
     <version>1.14.3</version>
     <scope>provided</scope>
 </dependency>
 <dependency>
     <groupId>org.apache.flink</groupId>
     <artifactId>flink-streaming-scala_${scala.version}</artifactId>
     <version>${flink.version}</version>
     <scope>provided</scope>
 </dependency>
 <dependency>
     <groupId>org.apache.flink</groupId>
     <artifactId>flink-clients_${scala.version}</artifactId>
     <version>${flink.version}</version>
     <scope>provided</scope>
 </dependency>
</dependencies>
```

说明

scala.version 和 flink.version 请根据自身使用 EMR 版本中的组件版本保持一致。

继续在 pom.xml 中添加打包和编译插件：

```
<build>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

```
</build>
```

如果您的 Maven 配置正确并且成功的导入了依赖包，那么整个工程应该没有错误可以直接编译。在本地命令行模式下进入工程目录，执行下面的命令对整个工程进行打包：

```
mvn package
```

运行过程中可能还需要下载一些文件，直到出现 build success 表示打包成功。然后您可以在工程目录下的 target 文件夹中看到打好的 jar 包。

数据准备

首先需要把压缩好的 jar 包上传到 EMR 集群中，使用 scp 或者 sftp 工具来进行上传。在本地命令行模式下运行：

```
scp $localfile root@公网IP地址:$remotefolder
```

其中，是您的本地文件的路径加名称；为服务器用户名；公网可以在控制台的节点信息中或者在云服务器控制台查看；localfile 是您的本地文件的路径加名称；root 为 CVM 服务器用户名；公网 IP 可以在 EMR 控制台的节点信息中或者在云服务器控制台查看；remotefolder 是您想存放文件的 CVM 服务器路径。上传完成后，在 EMR 命令行中即可查看对应文件夹下是否有相应文件。
需要处理的文件需要事先上传到 COS 中。如果文件在本地则可以通过 COS 控制台直接上传 。如果文件在 EMR 集群上，可以使用 Hadoop 命令上传。指令如下：

```
[hadoop@10 hadoop]$ hadoop fs -put $testfile cosn://BUCKET_NAME/
```

运行样例

首先需要登录 EMR 集群中的任意机器，最好是登录到 Master 节点。登录 EMR 的方式请参考登录 Linux 实例 。这里我们可以选择使用 WebShell 登录。单击对应云服务器右侧的登录，进入登录界面，用户名默认为 root，密码为创建 EMR 时用户自己输入的密码。输入正确后，即可进入命令行界面。
在 EMR 命令行先使用以下指令切换到 Hadoop 用户：

```
[root@172 ~]# su hadoop
[hadoop@172 ~]$ flink  run  -m yarn-cluster -c com.tencent.flink.CosWordcount ./flink-example-1.0-
SNAPSHOT.jar cosn://$BUCKET_NAME/test/data.txt cosn://$BUCKET_NAME/test/result
[hadoop@172 ~]$ hdfs dfs -cat cosn://becklong-cos/test/result
(Flink,8)
(Hadoop,3)
(Spark,7)
(Hbase,3)
```

JupyterLab 开发指南

最近更新时间：2025-01-07 18:04:03

JupyterLab 是 Jupyter 项目下的一个高度可扩展、功能丰富的交互式的开发环境，可用于笔记本、代码和数据的编写。其灵活的界面允许用户配置和安排数据科学、科学计算、计算新闻和机器学习中的工作流程；模块化设计允许扩展以扩展和丰富功能。本文通过示例简单介绍如何在 EMR 上使用 JupyterLab，详细的使用文档可参见 JupyterLab 官网文档 。

前提条件

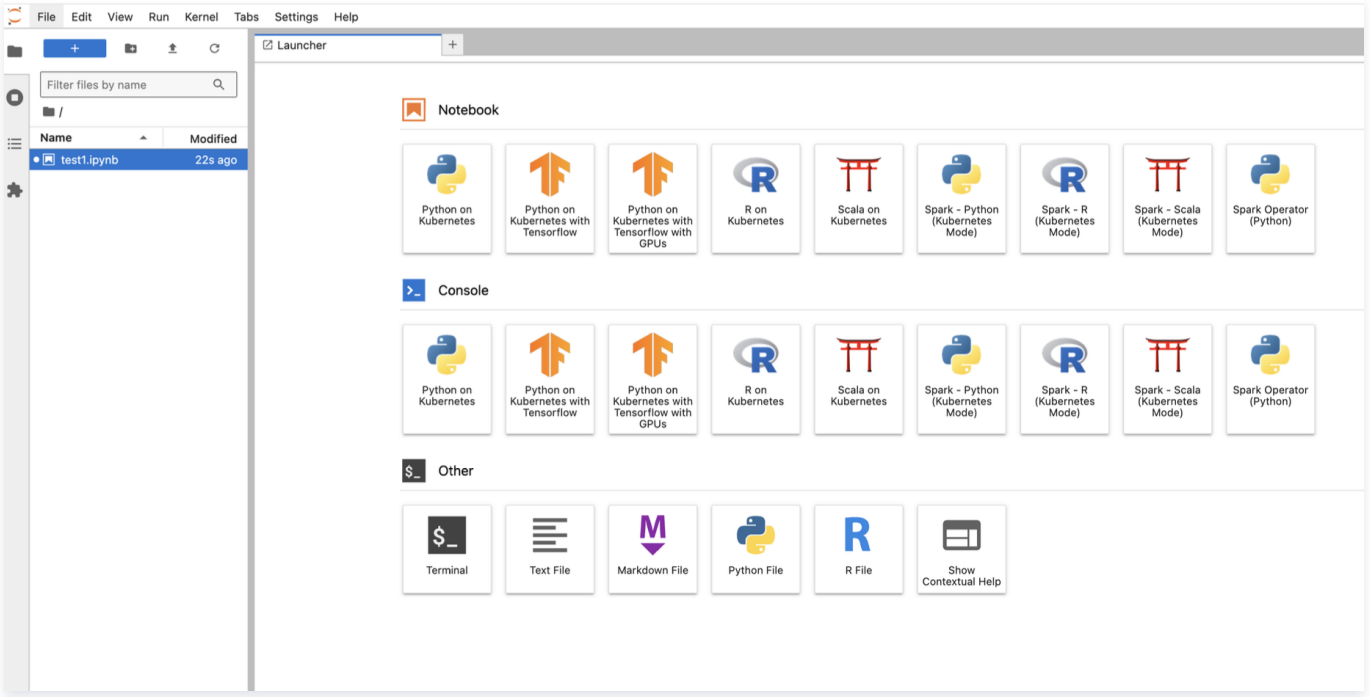
已创建 EMR on TKE 的机器学习集群，并选择了 JupyterLab，Eg 服务，详情请参见 创建集群 。

访问 JupyterLab WebUI

1. 您可以在购买集群时在 JUPYTERLAB 服务**编辑部署**页面开启外网访问，或者购买集群后进入集群控制台**集群服务**，选择 JUPYTERLAB 服务，在**角色管理**中单击**开启网络访问**。
2. 开启网络访问后，单击右上角**查看 WebUI**，即可打开 JupyterLab WebUI（需安全组开启8888端口）。

使用示例

- 登录 JupyterLab Web 页面，根据需要在工作区选择笔记本、控制台、文本或其他，以笔记本为例。



- 选择需要使用的 kernel，即可开始代码编写。

单元格工具栏

每个单元格中都有一个工具栏，快速访问常用功能。如果您想禁用单元格工具栏，请打开设置编辑器，选择左侧面板中的单元格工具栏，然后取消选中“显示单元格工具栏”。

- 管理员可以通过运行以下命令关闭单元格工具栏：

```
jupyter labextension disable @jupyterlab/cell-toolbar-extension
```

- 管理员可以通过运行以下命令重新打开单元格工具栏：

```
jupyter labextension enable @jupyterlab/cell-toolbar-extension
```

MLflow 开发指南

最近更新时间：2025-01-07 18:04:03

MLflow 是一个开源平台，专门用于帮助机器学习从业者和团队处理机器学习过程的复杂性。 MLflow 专注于机器学习项目的整个生命周期，确保每个阶段都是可管理、可追溯和可再现的。本文通过示例简单介绍如何在 EMR 上使用 MLflow，详细的使用文档可参见 MLflow 官网文档 。

前提条件

已创建 EMR on TKE 的机器学习集群，并选择了 MLflow 服务，详情请参见 创建集群 。

访问 MLflow WebUI

1. 您可以在购买集群时在 MLflow 服务**编辑部署**页面开启外网访问，或者购买集群后进入集群控制台**集群服务**，选择 **MLflow** 服务，在**角色管理**中单击**开启网络访问**。
2. 开启网络访问后，单击右上角**查看 WebUI** 即可打开 MLflow WebUI（需安全组开启5000端口）。

使用示例

- MLflow Tracking 是 MLflow 的主要服务组件之 ，本文以 notebook 演示使用 MLflow Tracking 为示例，代码示例可参见 MLflow 官网快速入门 。
- 操作步骤可分为：准备数据集，训练模型并将模型及其元数据记录到 MLflow，将模型加载为 Python 函数，使用加载的模型对新数据进行预测。
- 上述操作执行后，您可以在 MLflow UI 中查看运行结果，如下图，单击**运行任务名称**，即可进入**详情页**查看更多信息。