阿里云
E-MapReduce

最佳实践

文档版本：20190905

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的”现状“、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含”阿里云”、Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **禁止:** 重置操作将丢失用户配置数据。
	该类警示信息可能导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **警告:** 重启操作将导致业务中断，恢复业务所需时间约10分钟。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 **说明:** 您也可以通过按Ctrl + A选中全部文件。
>	多级菜单递进。	设置 > 网络 > 设置网络类型
粗体	表示按键、菜单、页面名称等UI元素。	单击 **确定**。
`courier` 字体	命令。	执行 `cd /d C:/windows` 命令，进入Windows系统文件夹。
*## *	表示参数、变量。	`bae log list --instanceid` *`Instance_ID`*
[]或者[a|b]	表示可选项，至多选择一个。	`ipconfig` *`[-all|-t]`*
{}或者{a|b}	表示必选项，至多选择一个。	`swich` *`{stand | slave}`*

目录

1 使用 E-MapReduce 采集 Kafka 客户端 Metrics 数据

本文介绍通过使用 E-MapReduce 产品从 Kafka 客户端采集 Metrics 数据从而有效地进行性能监控。

背景信息

Kafka 提供了一套非常完善的 Metrics 数据，覆盖 Broker、Consumer、Producer、Stream 以及 Connect 。E-MapReduce 通过 Ganglia 收集了 Kafka Broker Metrics 信息，可以很好地监控 Broker 运行状态。但完整的 Kafka 应用包括 Kafka Broker 和 Kafka 客户端这两个角色，当发生读写性能问题时，仅从 Broker 角度难以发现问题，需要结合客户端的运行状况来联合分析才行。那么 Kafka 客户端 Metrics 就是一类非常重要的数据。

实现原理

- 如何采集 Metrics

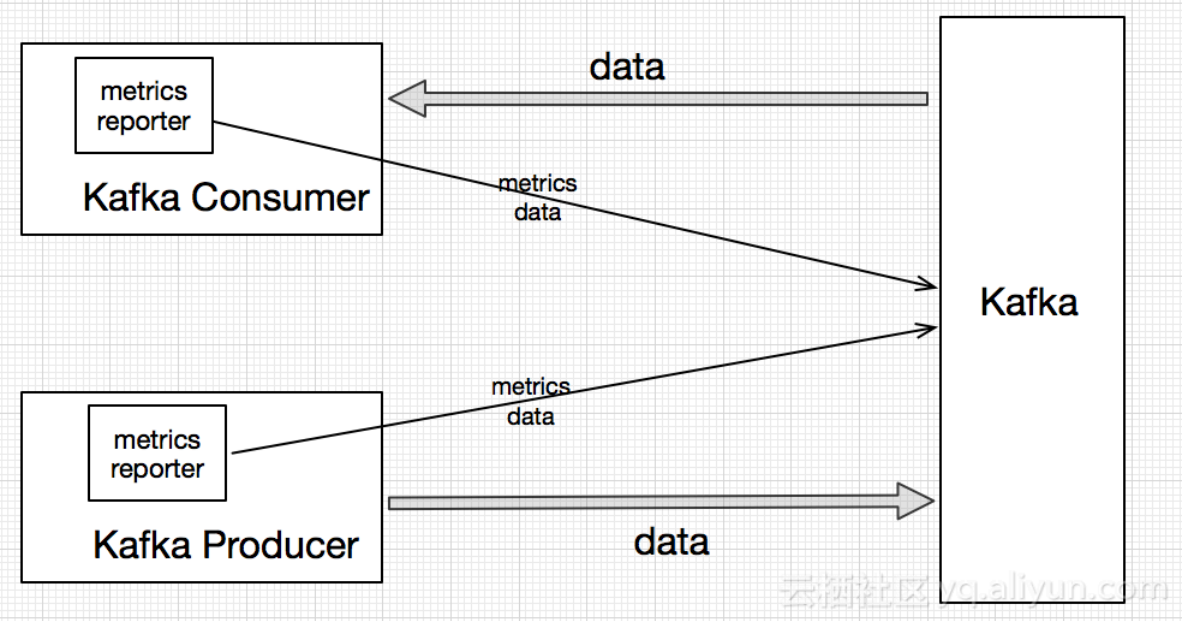
  Metrics Kafka 默认提供了包含 JmxReporter 的 Metrics Reporter 插件扩展功能，即我们已经可以通过 JMX 工具来查看 Kafka 的 Metrics 。所以，我们可以自己实现一套 Metrics Reporter（实现`org.apache.kafka.common.metrics.MetricsReporter`），来自定义获取这些 Metrics。

- 如何存放 Metrics

  以上我们实现了自定义采集 Kafka Metrics，还需要选择一个存储系统将这些 Metrics存储起来，以便后续的使用和分析。考虑到 Kafka 自身就是一种存储系统，我们可以将 Metrics 存储到 Kafka 中，这样做有以下几种优势：

  - 无需第三方存储系统支持
  - 数据很方便地接入到其他系统中

  所以，完整的客户端 Metrics 采集方案如下图所示：



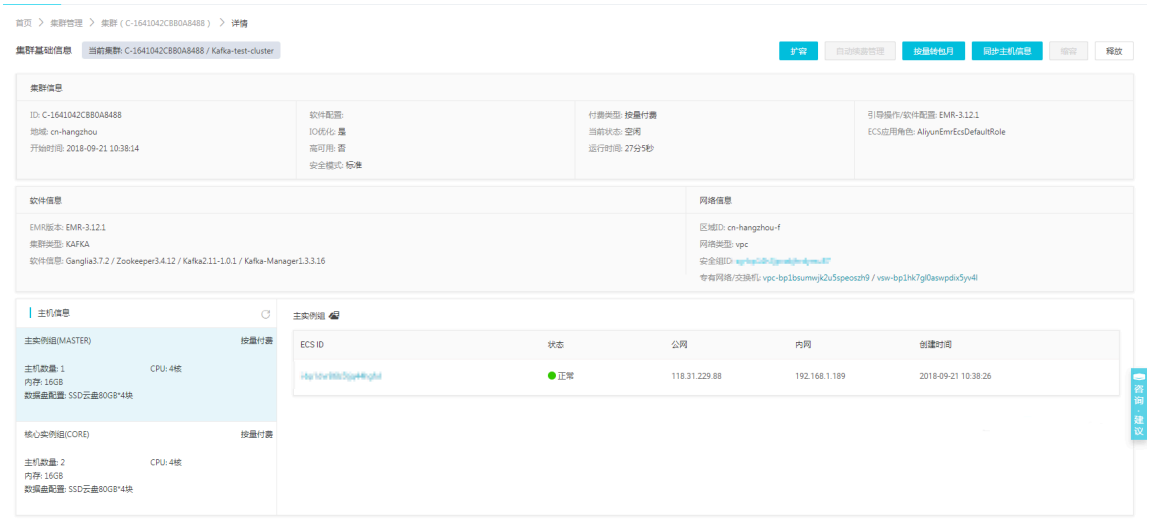
环境准备

- 限制条件
  - 只支持 Java 类客户端程序
  - 只支持 0.10 之后版本 Kafka 客户端
- 无需自己编译，EMR 已经向 Maven 发布了 jar 包，直接下载即可。

- 本文使用阿里云 EMR 服务自动化搭建 Kafka 集群，详细过程请参见创建集群。

  本文使用的 EMR Kafka 版本信息如下：

  - EMR 版本: EMR-3.12.1
  - 集群类型：Kafka
  - 软件信息：Kafka-Manager (1.3.3.16)/Kafka (2.11-1.0.1)/ZooKeeper (3.4.12)/ Ganglia (3.7.2)
  - Kafka 集群使用专有网络，区域为华东 1（杭州），主实例组 ECS 计算资源配置公网及内网 IP，具体配置如下所示：



  - 配置

配置项	说明
metric.reporters	使用 EMR 的实现：*org.apache.kafka.clients.reporter.EMRClientMetricsReporter*
emr.metrics.reporter.bootstrap.servers	Metrics 存储 Kafka 集群的 bootstrap.servers
emr.metrics.reporter.zookeeper.connect	Metrics 存储 Kafka 集群的 Zookeeper 地址

  - 如何加载 Metrics

    ■ 将 emr-kafka-client-metrics 的 jar 包放在客户端程序的 Classpath 可以加载到的机器上即可。

    ■ 直接将 emr-kafka-client-metrics 依赖打进客户端程序 jar 包中。

实施步骤

1. **下载最新版本 emr-kafka-client-metrics 包。**

```
wget http://central.maven.org/maven2/com/aliyun/emr/emr-kafka-client
-metrics/1.4.3/emr-kafka-client-metrics-1.4.3.jar
```

2. **将 emr-kafka-client-metrics 包放到 Kafka 的 lib 中**

```
cp emr-kafka-client-metrics-1.4.3.jar /usr/lib/kafka-current/libs/
```

3. **创建一个测试 Topic**

```
kafka-topics.sh --zookeeper emr-header-1:2181/kafka-1.0.1 --
partitions 10 --replication-factor 2 --topic test-metrics  --create
```

4. **向测试 Topic 写入数据，这里我们将 producer 的配置项配置到本地文件** *client.conf* **中：**

```
## client.conf:
metric.reporters=org.apache.kafka.clients.reporter.EMRClientM
etricsReporter
emr.metrics.reporter.bootstrap.servers=emr-worker-1:9092
emr.metrics.reporter.zookeeper.connect=emr-header-1:2181/kafka-1.0.1
bootstrap.servers=emr-worker-1:9092
## 命令:
kafka-producer-perf-test.sh --topic test-metrics --throughput 1000
 --num-records 100000
--record-size 1024 --producer.config client.conf
```

5. **查看这次客户端的 Metrics，注意默认的 Metrics Topic 是** *_emr-client-metrics* **。**

```
kafka-console-consumer.sh --topic _emr-client-metrics --bootstrap-
server emr-worker-1:9092
--from-beginning
```

返回的消息如下所示：

```
{prefix=kafka.producer, client.ip=192.168.xxx.xxx, client.process=
25536@emr-header-1.cluster-xxxx,
attribute=request-rate, value=894.4685104965012, timestamp=
1533805225045, group=producer-metrics,
tag.client-id=producer-1}
```

参数说明：

字段名	说明
client.ip	客户端所在机器 IP
client.process	客户端程序进程 ID
attribute	Metric 属性名
value	Metric 值
timestamp	Metric 采集的时间戳

字段名	说明
tag.xxx	Metric 其他 tag 信息

2 使用 E-MapReduce 处理离线作业

本文介绍使用 E-MapReduce（以下简称 EMR） 产品从 OSS 服务上读取数据 ，并进行数据采集、分析等一系列离线数据处理的使用场景。

背景信息

EMR 集群适用于多种场景。EMR 本质就是 Hadoop 和 Spark 的集群服务，所以 Hadoop ecosystem 以及 Spark 能够支持的场景，EMR 都可以支持。您完全可以将 EMR 集群使用的阿里云 ECS 主机视为自己专属的物理主机。

大数据处理目前比较常见的有两种方法：

- 离线处理：只是希望得到数据的分析结果，对处理的时间要求不严格，例如批量数据处理，用户将数据传输到 OSS 服务，OSS 服务作为 EMR 产品的输入输出，利用 MapReduce、Hive、Pig、Spark 处理离线数据。
- 在线处理：对于数据的分析结果在时间上有比较严格的要求，例如实时流式数据处理，使用 Spark Streaming 对消息数据进行处理，与 Spark mllib，GrapX，SQL 深度整合。

下面将使用 EMR 产品运行一个 word count 离线作业。

基本架构

OSS -> EMR -> Hadoop MapReduce

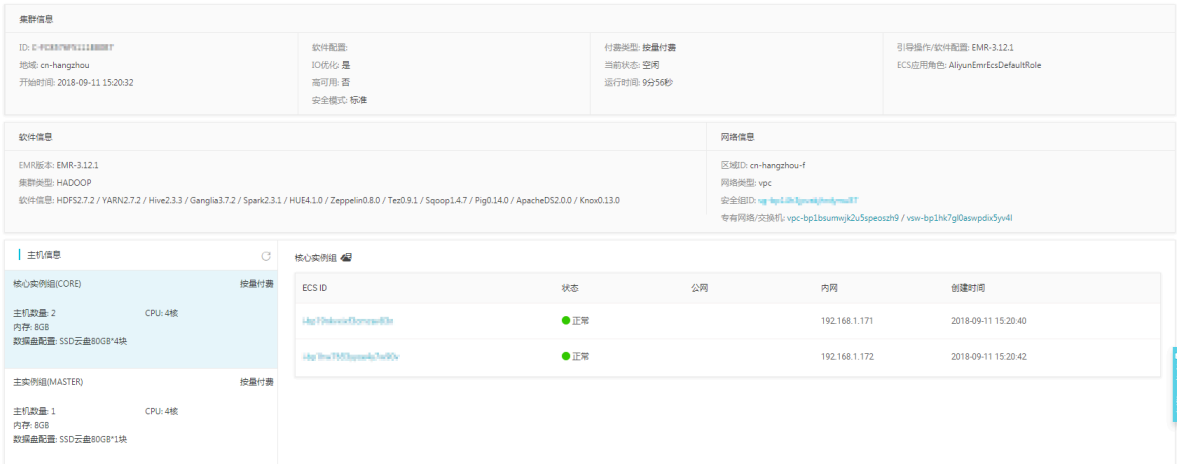
上述链路主要包含两个过程：

1. 把数据存储到 OSS 服务中。
2. 通过 EMR 服务将 OSS 中的数据读取出来，进行分析。

环境准备

- 本文以 Windows 环境为例，请确保 Git、Maven、 Java 已经安装并配置成功。

- 本文使用阿里云 EMR 服务自动化搭建 Hadoop 集群，详细步骤请参见创建集群。
  - EMR 版本：EMR-3.12.1
  - 集群类型：Hadoop
  - 软件信息：HDFS 2.7.2/YARN 2.7.2/Hive 2.3.3/Ganglia 3.7.2/Spark 2.3.1/HUE 4.1.0/Zeppelin 0.8.0/Tez 0.9.1/Sqoop 1.4.7/Pig 0.14.0/ApacheDS 2.0.0/Knox 0.13.0
  - Hadoop 集群使用专有网络，区域为华东 1（杭州），主实例组 ECS 计算资源配置公网及内网 IP，高可用选择为否（非 HA 模式）。



操作步骤

1. 下载示例代码到本地 。

   在本地打开 git bash 运行 clone 命令：

   ```
   git clone https://github.com/aliyun/aliyun-emapreduce-demo.git
   ```

   执行 `mvn install` 进行编译。

2. 创建 OSS 存储空间，详细步骤请参见创建存储空间。

   > **说明:**
   > Bucket 应与 E-MapReduce 集群在同一个区域。

3. 上传 jar 包和资源文件。

   a. 登录 OSS 管理控制台，单击文件管理。

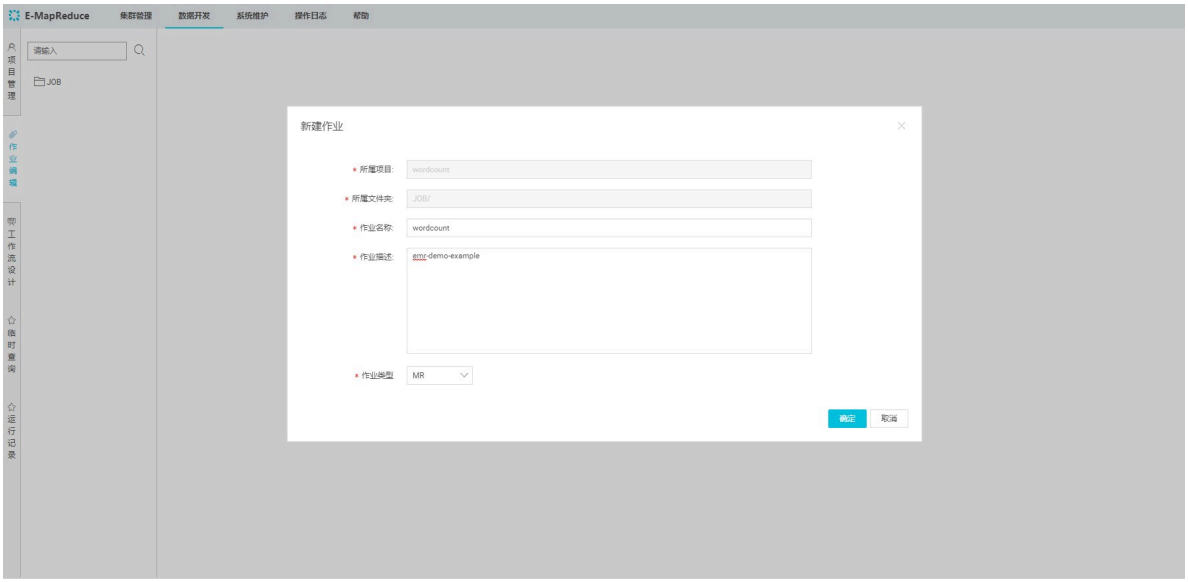
   b. 单击上传文件，上传 *aliyun-emapreduce-demo/resources* 目录下的资源文件以及 *aliyun-emapreduce-demo/target* 目录下的 jar 文件。

4. 创建工作流项目。

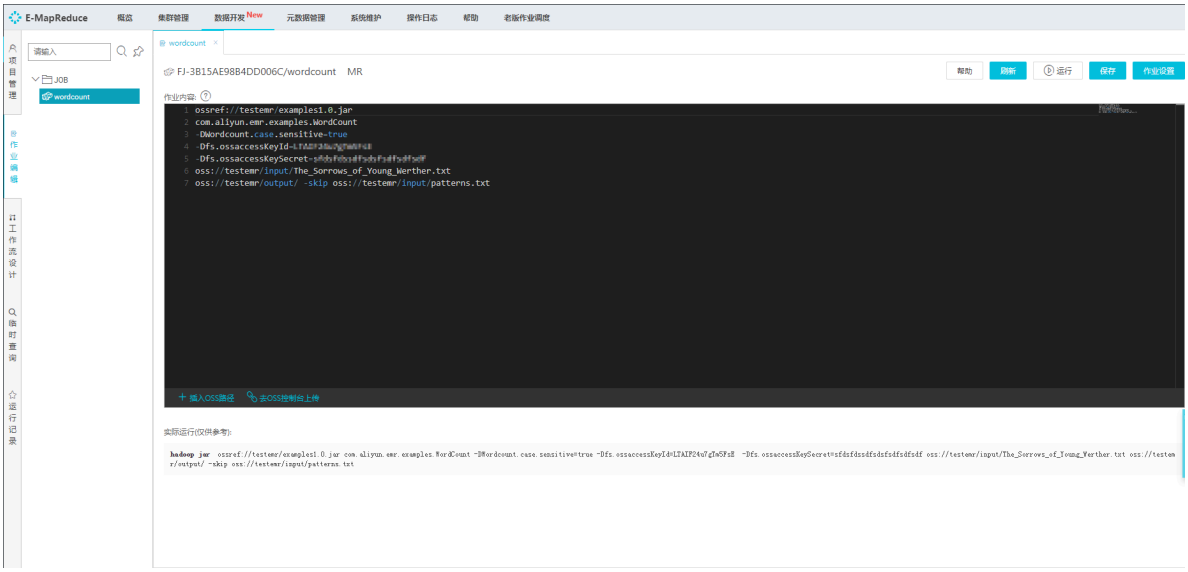
   详细过程请参考工作流项目管理。

5. 创建作业。

   详细步骤请参见作业编辑，这里我们以 MapReduce 为例，分别填写所属项目、作业名称、作业描述，并选择作业类型 MR。



6. 配置作业内容，单击运行。



- 关于 OSS 使用，请参考 OSS 参考使用说明
- 关于各类作业的具体开发，请参见 EMR 用户指南 《作业》 部分。

> **说明:**
>
> - OSS 输出路径如果已经存在，在执行作业时会报错。
> - 插入 OSS 路径时，如果选择 OSSREF 文件前缀，系统会把 OSS 文件下载到集群本地，并添加到 classpath 中。
> - 当前所有操作都只支持标准存储类型的 OSS。

日志查看

作业运行成功后，您可以在页面下方的运行记录页签中查看作业的运行日志。单击详情跳转到运行记录中该作业的详细日志页面，可以查看作业的提交日志、YARN Container 日志。您也可以通过 SSH 方式登录到集群查看相关日志，详细步骤请参见 SSH 登录集群。

总结

至此，我们成功在 E-MapReduce 上部署一套 Hadoop 集群，将 OSS 服务作为数据源输入，并运行 MapReduce 作业消费 OSS 上数据。E-MapReduce 支持多种类型作业的开发，例如 Storm 作业消费 Kafka 数据， Spark Streaming 和 Flink 组件，同样可以方便地在 Hadoop 集群上运行，处理 Kafka 数据。

3 使用 E-MapReduce 提交 Storm 作业处理 Kafka 数据

本文介绍如何使用阿里云 E-MapReduce（以下简称 EMR） 部署 Storm 集群和 Kafka 集群，并运行 Storm 作业消费 Kafka 数据。

环境准备

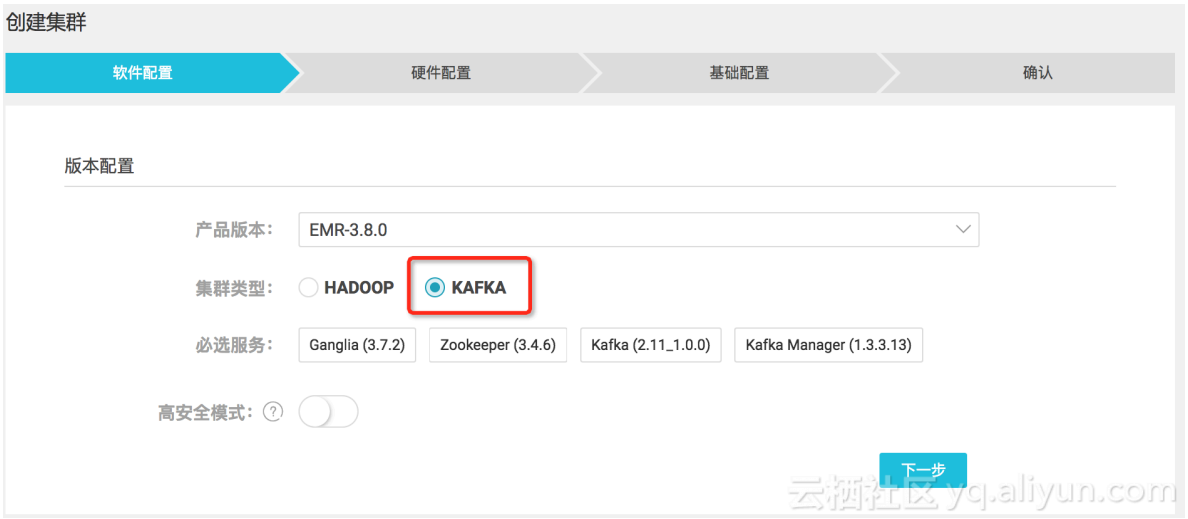
本文选择在杭州 Region 进行测试，版本选择 EMR-3.8.0，本次测试需要的组件版本有：

- Kafka：2.11_1.0.0
- Storm: 1.0.1

本文使用阿里云 EMR 服务自动化搭建Kafka集群，详细过程请参考创建集群。

- 创建 Hadoop 集群

- 创建 Kafka 集群



说明:

- 如果使用经典网络，请注意将 Hadoop 集群和Kafka集群放置在同一个安全组下面，这样可以省去配置安全组，避免网络不通的问题。
- 如果使用 VPC 网络，请注意将 Hadoop 集群和 Kafka 集群放置在同一个 VPC/VSwitch 以及安全组下面，这样同样省去配置网路和安全组，避免网络不通。
- 如果您熟悉ECS的网络和安全组，可以按需配置。

- 配置 Storm 环境

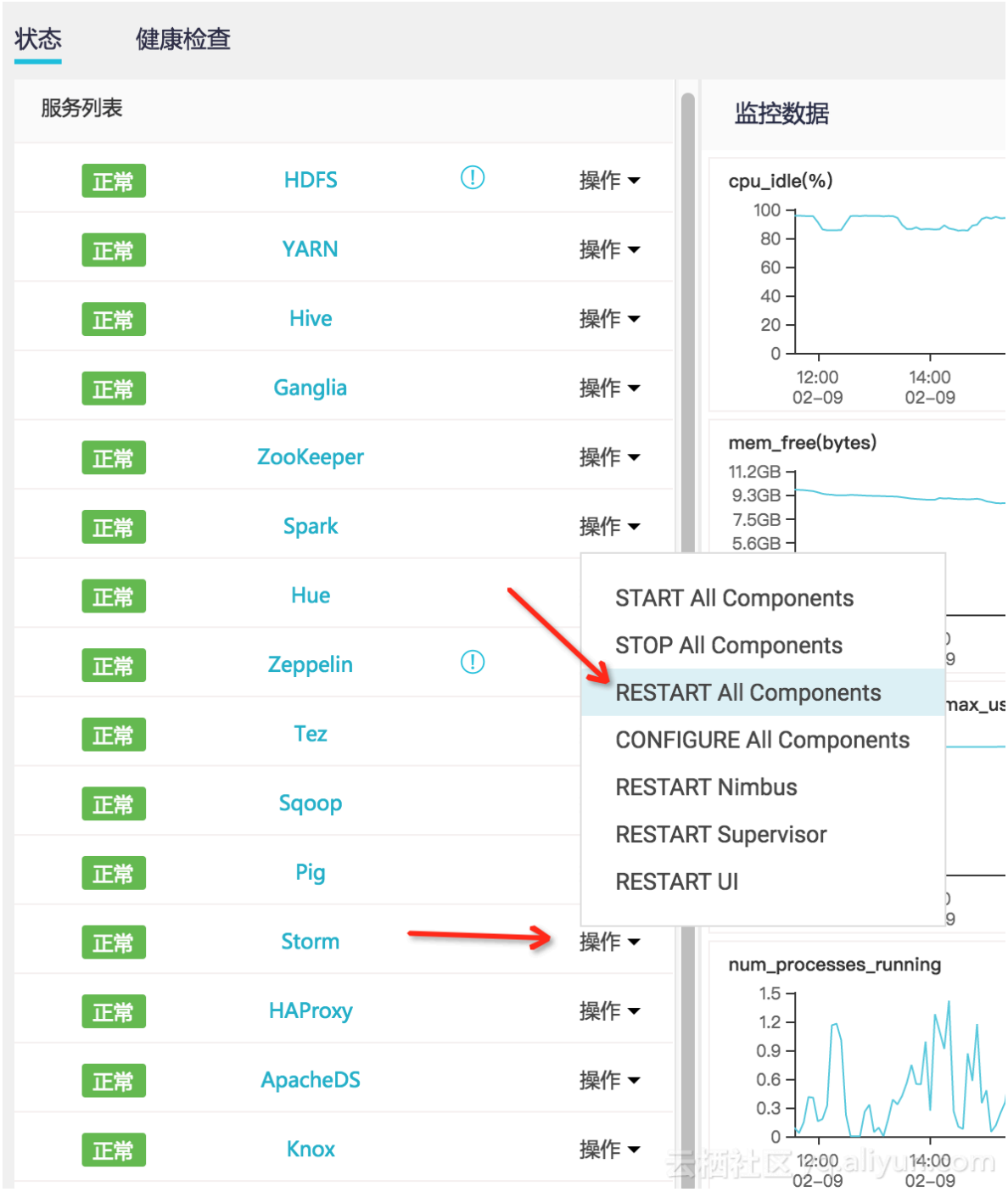
  如果我们想在 Storm 上运行作业消费 Kafka 的话，集群初始环境下是会失败的，因为 Storm 运行环境缺少了必须的依赖包，如下：

  - curator-client
  - curator-framework
  - curator-recipes
  - json-simple
  - metrics-core
  - scala-library
  - zookeeper
  - commons-cli
  - commons-collections
  - commons-configuration
  - htrace-core
  - jcl-over-slf4j
  - protobuf-java
  - guava
  - hadoop-common
  - kafka-clients
  - kafka
  - storm-hdfs
  - storm-kafka

  以上版本依赖包经过测试可用，如果你再测试过程中引入了其他依赖，也一同添加在 Storm lib 中，具体操作如下：

```
[hadoop@emr-header-1 ~]$ ll
total 8524
-rw-rw-r-- 1 hadoop hadoop   52988 Jun 14  2015 commons-cli-1.3.1.jar
-rw-rw-r-- 1 hadoop hadoop  588337 Nov 13  2015 commons-collections-3.2.2.jar
-rw-rw-r-- 1 hadoop hadoop  298829 Feb  5  2009 commons-configuration-1.6.jar
-rw-r--r-- 1 root   root     73448 Feb  9 14:01 curator-client-2.10.0.jar
-rw-r--r-- 1 root   root    195437 Feb  9 14:01 curator-framework-2.10.0.jar
-rw-r--r-- 1 root   root    281476 Feb  9 14:01 curator-recipes-2.10.0.jar
-rw-rw-r-- 1 hadoop hadoop   31212 Apr 19  2014 htrace-core-3.0.4.jar
-rw-rw-r-- 1 hadoop hadoop   17289 Jun 11  2012 jcl-over-slf4j-1.6.6.jar
-rw-rw-r-- 1 hadoop hadoop   16046 Aug 13  2009 json-simple-1.1.jar
-rw-rw-r-- 1 hadoop hadoop   82123 Nov 27  2012 metrics-core-2.2.0.jar
-rw-rw-r-- 1 hadoop hadoop  533455 Mar  8  2013 protobuf-java-2.5.0.jar
-rw-r--r-- 1 root   root   5745606 Feb  9 14:01 scala-library-2.11.7.jar
-rw-rw-r-- 1 hadoop hadoop  792964 Feb 24  2014 zookeeper-3.4.6.jar
[hadoop@emr-header-1 ~]$ pwd
/home/hadoop
[hadoop@emr-header-1 ~]$ sudo cp ./* /usr/lib/storm-current/lib/
```


上述操作需要在 Hadoop 集群的每台机器执行一遍。执行完在 E-MapReduce 控制台重启 Storm 服务，如下：



查看操作历史，待 Storm 重启完毕：

开发Storm和Kafka作业

- EMR 已经提供了现成的示例代码，直接使用即可，地址如下：
  - e-mapreduce-demo
  - e-mapreduce-sdk
- Topic 数据准备
  1. 登录到 Kafka 集群
  2. 创建一个test topic，分区数10，副本数2

```
/usr/lib/kafka-current/bin/kafka-topics.sh --partitions 10 --
replication-factor 2 --zookeeper emr-header-1:/kafka-1.0.0 --topic
 test --create
```

  3. 向 test topic 写入100条数据

```
/usr/lib/kafka-current/bin/kafka-producer-perf-test.sh --num-
records 100 --throughput 10000 --record-size 1024 --producer-props
 bootstrap.servers=emr-worker-1:9092 --topic test
```

说明:

以上命令在 kafka 集群的emr-header-1节点执行，当然也可以客户端机器上执行。

- 运行 Storm 作业

  登录到 Hadoop 集群，将编译得到的*/target/shaded*目录下的*examples-1.1-shaded.jar*拷贝到集群 emr-header-1上，这里以放在 root 根目录下面为例提交作业：

```
/usr/lib/storm-current/bin/storm jar examples-1.1-shaded.jar com.
aliyun.emr.example.storm.StormKafkaSample test aaa.bbb.ccc.ddd hdfs
://emr-header-1:9000 sample
```

- 查看作业运行

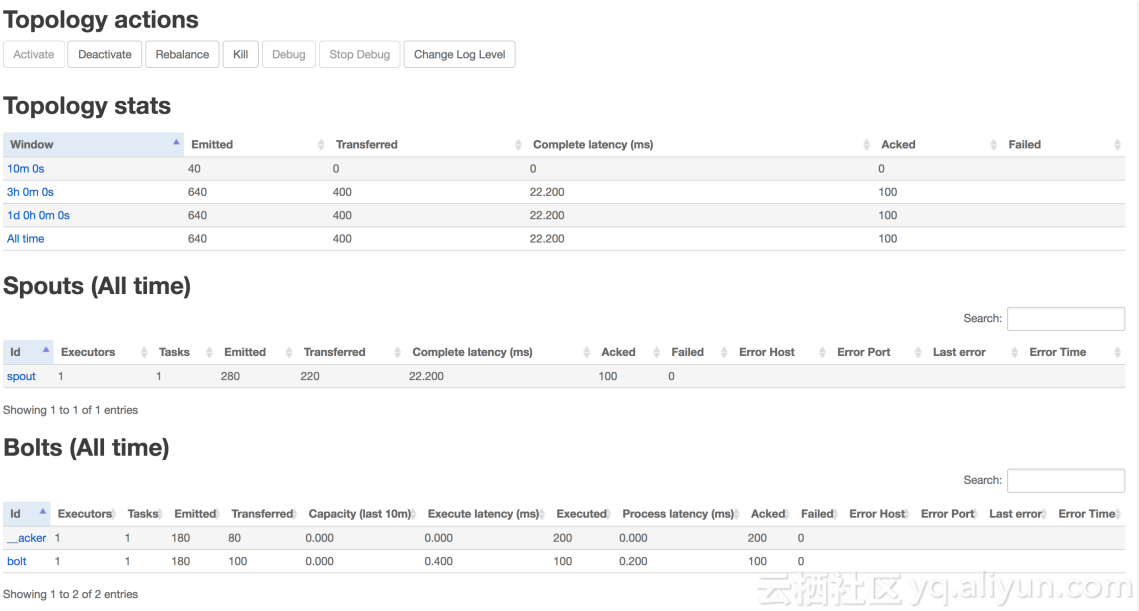
  - 查看 Storm 运行状态

    查看集群上服务的 WebUI 有2种方式：

    ■ 通过 Knox 方式，参考文档 Knox 使用说明

    ■ SSH 隧道，参考文档SSH 登录集群

    本文选择使用 SSH 隧道方式，访问地址：*http://localhost:9999/index.html*。可以看到我们刚刚提交的 Topology。点进去可以看到执行详情：



  - 查看 HDFS 输出

    ■ 查看 HDFS 文件输出

```
[root@emr-header-1 ~]# hadoop fs -ls /foo/
-rw-r--r--   3 root hadoop     615000 2018-02-11 13:37 /foo/
bolt-2-0-1518327393692.txt
-rw-r--r--   3 root hadoop     205000 2018-02-11 13:37 /foo/
bolt-2-0-1518327441777.txt
[root@emr-header-1 ~]# hadoop fs -cat /foo/bolt-2-0-1518327441
777.txt | wc -l
```

```
200
```

■ 向 Kafka写120条数据

```
[root@emr-header-1 ~]# /usr/lib/kafka-current/bin/kafka-
producer-perf-test.sh --num-records 120 --throughput 10000 --
record-size 1024 --producer-props bootstrap.servers=emr-worker-
1:9092 --topic test
120 records sent, 816.326531 records/sec (0.80 MB/sec), 35.37
ms avg latency, 134.00 ms max latency, 35 ms 50th, 39 ms 95th,
41 ms 99th, 134 ms 99.9th.
```

■ 查看 HDFS 文件输出

```
[root@emr-header-1 ~]# hadoop fs -cat /foo/bolt-2-0-1518327441
777.txt | wc -l
320
```

总结

至此，我们成功实现了在 E-MapReduce 上部署一套 Storm 集群和一套 Kafka 集群，并运行 Storm 作业消费 Kafka 数据。当然，E-MapReduce 也支持 Spark Streaming 和 Flink 组件，同样可以方便在 Hadoop 集群上运行，处理 Kafka 数据。

说明:

由于 E-MapReduce 没有单独的 Storm 集群类别，所以我们是创建的 Hadoop 集群，并安装了 Storm 组件。如果你在使用过程中用不到其他组件，可以很方便地在 E-MapReduce 管理控制台将那些组件停掉。这样，可以将 Hadoop 集群作为一个纯粹的 Storm 集群使用。

4 在 E-MapReduce 中使用 ES-Hadoop

ES-Hadoop 是 Elasticsearch(ES) 推出的专门用于对接 Hadoop 生态的工具，使得用户可以使用 Mapreduce(MR)、Spark、Hive 等工具处理 ES 上的数据（ES-Hadoop 还包含另外一部分：将 ES 的索引 snapshot 到 HDFS，对于该内容本文暂不讨论）。

背景信息

Hadoop 生态的长处是处理大规模数据集，但是其缺点也很明显，就是当用于交互式分析时，查询时延会比较长。而 ES 是这方面的好手，对于很多查询类型，特别是 ad-hoc 查询，基本可以做到秒级。ES-Hadoop 的推出提供了一种组合两者优势的可能性。使用 ES-Hadoop，用户只需要对自己代码做出很小的改动，即可以快速处理存储在 ES 中的数据，并且能够享受到 ES 带来的加速效果。

ES-Hadoop 的逻辑是将 ES 作为 MR/Spark/Hive 等数据处理引擎的“数据源”，在计算存储分离的架构中扮演存储的角色。这和 MR/Spark/Hive 的其他数据源并无差异。但相对于其他数据源， ES 具有更快的数据选择过滤能力。这种能力正是分析引擎最为关键的能力之一。

EMR 中已经添加了对 ES-Hadoop 的支持，用户不需要做任何配置即可使用 ES-Hadoop。下面我们通过几个例子，介绍如何在 EMR 中使用 ES-Hadoop。

准备工作

ES 有自动创建索引的功能，能够根据输入数据自动推测数据类型。这个功能在某些情况下很方便，避免了用户很多额外的操作，但是也产生了一些问题。最重要的问题是 ES 推测的类型和我们预期的类型不一致。比如我们定义了一个字段叫 *age*，INT 型，在 ES 索引中可能被索引成了 LONG 型。在执行一些操作的时候会带来类型转换问题。为此，我们建议手动创建索引。

在下面几个例子中，我们将使用同一个索引 *company*和一个类型 *employees*（ES 索引可以看成一个 database，类型可以看做 database 下的一张表），该类型定义了四个字段（字段类型均为 ES 定义的类型）：

```
{
  "id": long,
  "name": text,
  "age": integer,
  "birth": date
}
```

在 kibana 中运行如下命令创建索引（或用相应的 curl 命令）：

```
PUT company
{
  "mappings": {
    "employees": {
```

```
      "properties": {
        "id": {
          "type": "long"
        },
        "name": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "birth": {
          "type": "date"
        },
        "addr": {
          "type": "text"
        }
      }
    }
  },
  "settings": {
    "index": {
      "number_of_shards": "5",
      "number_of_replicas": "1"
    }
  }
}
```

说明:

其中 settings 中的索引参数可根据需要设定，也可以不具体设定 settings。

准备一个文件，每一行为一个 json 对象，如下所示:

```
{"id": 1, "name": "zhangsan", "birth": "1990-01-01", "addr": "No.969,
wenyixi Rd, yuhang, hangzhou"}
{"id": 2, "name": "lisi", "birth": "1991-01-01", "addr": "No.556, xixi
 Rd, xihu, hangzhou"}
{"id": 3, "name": "wangwu", "birth": "1992-01-01", "addr": "No.699
wangshang Rd, binjiang, hangzhou"}
```

并保存至 HDFS 指定目录（如 *es-hadoop/employees.txt*）。

Mapreduce

在下面这个例子中，我们读取 HDFS上 *es-hadoop* 目录下的 json 文件，并将这些 json 文件中的每一行作为一个 document 写入 es。写入过程由 EsOutputFormat 在 map 阶段完成。

这里对 ES 的设置主要是如下几个选项:

- **es.nodes: ES 节点，为 host:port 格式。对于阿里云托管式 ES，此处应为阿里云提供的 ES 访问域名**
- **es.net.http.auth.user: 用户名**
- **es.net.http.auth.pass: 用户密码**

- **es.nodes.wan.only: 对于阿里云托管式 ES，此处应设置为 `true`**
- **es.resource: ES 索引和类型**
- **es.input.json: 如果原始文件为 json 类型，设置为 `true`，否则，需要在 map 函数中自己解析原始数据，生成相应的 Writable 输出**

 **注意:**

关闭 map 和 reduce 的推测执行机制

```
package com.aliyun.emr;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.elasticsearch.hadoop.mr.EsOutputFormat;

public class Test implements Tool {

  private Configuration conf;

  @Override
  public int run(String[] args) throws Exception {

    String[] otherArgs = new GenericOptionsParser(conf, args).
getRemainingArgs();

    conf.setBoolean("mapreduce.map.speculative", false);
    conf.setBoolean("mapreduce.reduce.speculative", false);
    conf.set("es.nodes", "<your_es_host>:9200");
    conf.set("es.net.http.auth.user", "<your_username>");
    conf.set("es.net.http.auth.pass", "<your_password>");
    conf.set("es.nodes.wan.only", "true");
    conf.set("es.resource", "company/employees");
    conf.set("es.input.json", "yes");

    Job job = Job.getInstance(conf);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(EsOutputFormat.class);
    job.setMapOutputKeyClass(NullWritable.class);
    job.setMapOutputValueClass(Text.class);
    job.setJarByClass(Test.class);
    job.setMapperClass(EsMapper.class);

    FileInputFormat.setInputPaths(job, new Path(otherArgs[0]));

    return job.waitForCompletion(true) ? 0 : 1;
  }

  @Override
  public void setConf(Configuration conf) {
```

```
    this.conf = conf;
  }

  @Override
  public Configuration getConf() {
    return conf;
  }

  public static class EsMapper extends Mapper<Object, Text, NullWritab
le, Text> {
    private Text doc = new Text();

    @Override
    protected void map(Object key, Text value, Context context) throws
 IOException, InterruptedException {
      if (value.getLength() > 0) {
        doc.set(value);
        context.write(NullWritable.get(), doc);
      }
    }
  }

  public static void main(String[] args) throws Exception {
    int ret = ToolRunner.run(new Test(), args);
    System.exit(ret);
  }
}
```

将该代码编译打包为 *mr-test.jar*, 上传至装有 emr 客户端的机器（如 gateway，或者 EMR cluster 任意一台机器）。

在装有 EMR 客户端的机器上运行如下命令执行 mapreduce 程序：

```
hadoop jar mr-test.jar com.aliyun.emr.Test -Dmapreduce.job.reduces=0 -
libjars mr-test.jar /es-hadoop
```

即可完成向 ES 写数据。具体写入的数据可以通过 kibana 查询（或者通过相应的 curl 命令）：

```
GET
{
  "query": {
    "match_all": {}
  }
}
```

Spark

本示例同 Mapreduce 一样，也是向 ES 的一个索引写入数据，只不过是通过 spark 来执行。这里 spark 借助 JavaEsSpark 类将一份 RDD 持久化到 es。同上述 Mapreduce 程序一样，用户也需要注意上述几个选项的设置。

```
package com.aliyun.emr;

import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaRDD;
```

```
import org.apache.spark.api.java.function.Function;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.elasticsearch.spark.rdd.api.java.JavaEsSpark;
import org.spark_project.guava.collect.ImmutableMap;

public class Test {

  public static void main(String[] args) {
    SparkConf conf = new SparkConf();
    conf.setAppName("Es-test");
    conf.set("es.nodes", "<your_es_host>:9200");
    conf.set("es.net.http.auth.user", "<your_username>");
    conf.set("es.net.http.auth.pass", "<your_password>");
    conf.set("es.nodes.wan.only", "true");

    SparkSession ss = new SparkSession(new SparkContext(conf));
    final AtomicInteger employeesNo = new AtomicInteger(0);
    JavaRDD<Map<Object, ?>> javaRDD = ss.read().text("hdfs://emr-
header-1:9000/es-hadoop/employees.txt")
        .javaRDD().map((Function<Row, Map<Object, ?>>) row ->
ImmutableMap.of("employees" + employeesNo.getAndAdd(1), row.mkString
()));

    JavaEsSpark.saveToEs(javaRDD, "company/employees");
  }
}
```

将其打包成 spark-test.jar，运行如下命令执行写入过程：

```
spark-submit --master yarn --class com.aliyun.emr.Test spark-test.jar
```

待任务执行完毕后可以使用 kibana 或者 curl 命令查询结果。

除了 `spark rdd` 操作，es-hadoop 还提供了使用 sparksql 来读写 ES。详细请参考 ES-Hadoop 官方页面。

Hive

这里展示使用 Hive 通过 SQL 来读写 ES 的方法。

首先运行 `hive`命令进入交互式环境，先创建一个表：

```
CREATE DATABASE IF NOT EXISTS company;
```

之后创建一个外部表，表存储在 ES 上，通过 TBLPROPERTIES 来设置对接 ES 的各个选项：

```
CREATE EXTERNAL table IF NOT EXISTS employees(
  id BIGINT,
  name STRING,
  birth TIMESTAMP,
  addr STRING
)
STORED BY 'org.elasticsearch.hadoop.hive.EsStorageHandler'
TBLPROPERTIES(
    'es.resource' = 'tpcds/ss',
    'es.nodes' = '<your_es_host>',
    'es.net.http.auth.user' = '<your_username>',
```

```
    'es.net.http.auth.pass' = '<your_password>',
    'es.nodes.wan.only' = 'true',
    'es.resource' = 'company/employees'
);
```

说明:

在 Hive 表中我们将 birth 设置成了 TIMESTAMP 类型，而在 ES 中我们将其设置成了 DATE 型。这是因为 Hive 和 ES 对于数据格式处理不一致。在写入时，Hive 将原始 date 转换后发送给 ES 可能会解析失败，相反在读取时，ES 返回的格式 Hive 也可能解析失败。参见这里。

往表中插入一些数据：

```
INSERT INTO TABLE employees VALUES (1, "zhangsan", "1990-01-01","No.
969, wenyixi Rd, yuhang, hangzhou");
INSERT INTO TABLE employees VALUES (2, "lisi", "1991-01-01", "No.556,
xixi Rd, xihu, hangzhou");
INSERT INTO TABLE employees VALUES (3, "wangwu", "1992-01-01", "No.699
 wangshang Rd, binjiang, hangzhou");
```

执行查询即可看到结果：

```
SELECT * FROM employees LIMIT 100;
OK
1    zhangsan    1990-01-01    No.969, wenyixi Rd, yuhang, hangzhou
2    lisi    1991-01-01    No.556, xixi Rd, xihu, hangzhou
3    wangwu    1992-01-01    No.699 wangshang Rd, binjiang, hangzhou
```

5 在 E-MapReduce中使用 Mongo-Hadoop

Mongo-Hadoop 是 MongoDB 推出的用于 Hadoop 系列组件连接 MongoDB 的组件。其原理跟我们上一篇文章介绍的 ES-Hadoop 类似。E-MapReduce 中已经集成了 Mongo-Hadoop，用户不用做任何部署配置，即可使用 Mongo-Hadoop。本文通过几个例子来展示一下 Mongo-Hadoop 的用法。

准备

在下面这几个例子中，我们使用一个统一的数据模型：

```
{
  "id": long,
  "name": text,
  "age": integer,
  "birth": date
}
```

由于我们是要通过 Mongo-Hadoop 向 MongoDB 的特定 collection （可以理解成数据库中的表）写数据，因此需要首先确保 MongoDB 上存在这个 collection。为此，首先需要在一台能够连接到 MongoDB 的客户机上运行 Mongo client（你可能需要安装一下客户端程序，客户端程序可在 Mongo 官网下载）。我们以连接阿里云数据库 MongoDB 版为例：

```
mongo --host dds-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717
 --authenticationDatabase admin -u root -p 123456
```

其中 *dds-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com* 为 MongoDB 的主机名，3717 为端口号（该端口号根据您的 MongoDB 集群而定，对于自建集群，默认为 27017），-p 为密码（这里假设密码为 123456）。进入交互式页面，运行如下命令，在 company 数据库下创建名为 employees 的 collection：

```
> use company;
> db.createCollection("employees")
```

准备一个文件，每一行为一个 json 对象，如下所示：

```
{"id": 1, "name": "zhangsan", "birth": "1990-01-01", "addr": "No.969,
wenyixi Rd, yuhang, hangzhou"}
{"id": 2, "name": "lisi", "birth": "1991-01-01", "addr": "No.556, xixi
 Rd, xihu, hangzhou"}
{"id": 3, "name": "wangwu", "birth": "1992-01-01", "addr": "No.699
wangshang Rd, binjiang, hangzhou"}
```

并保存至 HDFS 指定目录（如 */mongo-hadoop/employees.txt*）。

Mapreduce

在下面这个例子中，我们读取 HDFS 上 *mongo-hadoop* 目录下的 json 文件，并将这些 json 文件中的每一行作为一个 document 写入 MongoDB。

```
package com.aliyun.emr;

import com.mongodb.BasicDBObject;
import com.mongodb.hadoop.MongoOutputFormat;
import com.mongodb.hadoop.io.BSONWritable;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class Test implements Tool {

  private Configuration conf;

  @Override
  public int run(String[] args) throws Exception {

    String[] otherArgs = new GenericOptionsParser(conf, args).
getRemainingArgs();

    conf.set("mongo.output.uri", "mongodb://<your_username>:<
your_password>@dds-xxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717
/company.employees?authSource=admin");

    Job job = Job.getInstance(conf);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(MongoOutputFormat.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(BSONWritable.class);

    job.setJarByClass(Test.class);
    job.setMapperClass(MongoMapper.class);

    FileInputFormat.setInputPaths(job, new Path(otherArgs[0]));

    return job.waitForCompletion(true) ? 0 : 1;
  }

  @Override
  public Configuration getConf() {
    return conf;
  }

  @Override
  public void setConf(Configuration conf) {
    this.conf = conf;
  }

  public static class MongoMapper extends Mapper<Object, Text, Text,
BSONWritable> {
```

```
    private BSONWritable doc = new BSONWritable();
    private int employeeNo = 1;
    private Text id;

    @Override
    protected void map(Object key, Text value, Context context) throws
 IOException, InterruptedException {
      if (value.getLength() > 0) {
        doc.setDoc(BasicDBObject.parse(value.toString()));
        id = new Text("employee" + employeeNo++);
        context.write(id, doc);
      }
    }
  }

  public static void main(String[] args) throws Exception {
    int ret = ToolRunner.run(new Test(), args);
    System.exit(ret);
  }
}
```

将该代码编译打包为*mr-test.jar*， 运行：

```
hadoop jar mr-test.jar com.aliyun.emr.Test -Dmapreduce.job.reduces=0 -
libjars mr-test.jar /mongo-hadoop
```

待任务执行完毕后可以使用 MongoDB 客户端查询结果：

```
> db.employees.find();
{ "_id" : "employee1", "id" : 1, "name" : "zhangsan", "birth" : "1990-
01-01", "addr" : "No.969, wenyixi Rd, yuhang, hangzhou" }
{ "_id" : "employee2", "id" : 2, "name" : "lisi", "birth" : "1991-01-
01", "addr" : "No.556, xixi Rd, xihu, hangzhou" }
{ "_id" : "employee3", "id" : 3, "name" : "wangwu", "birth" : "1992-01
-01", "addr" : "No.699 wangshang Rd, binjiang, hangzhou" }
```

Spark

本示例同 Mapreduce 一样，也是向 MongoDB 写入数据，只不过是通过 Spark 来执行。

```
package com.aliyun.emr;

import com.mongodb.BasicDBObject;
import com.mongodb.hadoop.MongoOutputFormat;
import java.util.concurrent.atomic.AtomicInteger;
import org.apache.hadoop.conf.Configuration;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.bson.BSONObject;
import scala.Tuple2;

public class Test {

  public static void main(String[] args) {
```

```
    SparkSession ss = new SparkSession(new SparkContext());

    final AtomicInteger employeeNo = new AtomicInteger(0);
    JavaRDD<Tuple2<Object, BSONObject>> javaRDD =
        ss.read().text("hdfs://emr-header-1:9000/mongo-hadoop/
employees.txt")
            .javaRDD().map((Function<Row, Tuple2<Object, BSONObject
>>) row -> {
          BSONObject bson = BasicDBObject.parse(row.mkString());
          return new Tuple2<>("employee" + employeeNo.getAndAdd(1),
bson);
        });

    JavaPairRDD<Object, BSONObject> documents = JavaPairRDD.fromJavaRD
D(javaRDD);

    Configuration outputConfig = new Configuration();
    outputConfig.set("mongo.output.uri", "mongodb://<your_username>:<
your_password>@dds-xxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717
/company.employees?authSource=admin");

    // 将其保存为一个 "hadoop 文件"，实际上通过 MongoOutputFormat 写入 mongo
。
    documents.saveAsNewAPIHadoopFile(
        "file:///this-is-completely-unused",
        Object.class,
        BSONObject.class,
        MongoOutputFormat.class,
        outputConfig
    );
  }
}
```

将其打包成 *spark-test.jar*，运行如下命令执行写入过程：

```
spark-submit --master yarn --class com.aliyun.emr.Test spark-test.jar
```

待任务执行完毕后可以使用 MongoDB 客户端查询结果。

Hive

这里展示使用 Hive 通过 SQL 来读写 MongoDB 的方法。

首先运行 `hive`命令进入交互式环境，先创建一个表：

```
CREATE DATABASE IF NOT EXISTS company;
```

之后创建一个外部表，表存储在 MongoDB 上。但是创建外部表之前，注意像第一小节中介绍的，需要首先创建 MongoDB Collection — employees。

下面回到 Hive 交互式控制台，运行如下 SQL 创建一个外部表，MongoDB 连接通过 TBLPROPERTIES 来设置：

```
CREATE EXTERNAL TABLE IF NOT EXISTS employees(
  id BIGINT,
  name STRING,
  birth STRING,
```

```
  addr STRING
)
STORED BY 'com.mongodb.hadoop.hive.MongoStorageHandler'
WITH SERDEPROPERTIES('mongo.columns.mapping'='{"id":"_id"}')
TBLPROPERTIES('mongo.uri'='mongodb://<your_username>:<your_password
>@dds-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717/company.
employees?authSource=admin');
```

注意:

这里通过 SERDEPROPERTIES 把 Hive 的字段*id*和 MongoDB 的字段 *_id* 做了映射（用户可以根据自身需要选择做或者不做某些映射）。另外注意在 Hive 表中我们将 birth 设置成了 STRING 类型。这是因为 Hive 和 MongoDB 对于数据格式处理的不一致造成的问题。Hive 将原始 date 转换后发送给 MongoDB，然后再从 Hive 中查询可能会得到 NULL。

往表中插入一些数据：

```
INSERT INTO TABLE employees VALUES (1, "zhangsan", "1990-01-01","No.
969, wenyixi Rd, yuhang, hangzhou");
INSERT INTO TABLE employees VALUES (2, "lisi", "1991-01-01", "No.556,
xixi Rd, xihu, hangzhou");
INSERT INTO TABLE employees VALUES (3, "wangwu", "1992-01-01", "No.699
 wangshang Rd, binjiang, hangzhou");
```

执行查询即可看到结果：

```
SELECT * FROM employees LIMIT 100;
OK
1    zhangsan    1990-01-01    No.969, wenyixi Rd, yuhang, hangzhou
2    lisi    1991-01-01    No.556, xixi Rd, xihu, hangzhou
3    wangwu    1992-01-01    No.699 wangshang Rd, binjiang, hangzhou
```

6 在 E-MapReduce 上使用 Intel Analytics Zoo 进行深度学习

本文简单介绍了如何在阿里云 E-MapReduce 使用 Analytics Zoo 来进行深度学习。

简介

Analytics Zoo 是由 Intel 开源，基于Apache Spark 和 Inte BigDL 的大数据分析和 AI 平台，方便用户开发基于大数据、端到端的深度学习应用。

系统要求

- JDK 8
- Spark 集群(推荐使用EMR支持的 Spark 2.x)
- Python-2.7(python 3.5，3.6 也支持)，pip

安装 Analytics Zoo

1. Scala 安装

   a. 下载 pre-build 版本

   可以从 github，analytics 主页下载到pre-build 版本

   b. 通过 script build

   安装 Apache Maven，设置 Maven 环境：

   ```
   export MAVEN_OPTS="-Xmx2g -XX:ReservedCodeCacheSize=512m"
   ```

   如果使用 ECS 机器进行编译，推荐修改 Maven 仓库 mirror：

   ```
   <mirror>
       <id>nexus-aliyun</id>
       <mirrorOf>central</mirrorOf>
       <name>Nexus aliyun</name>
       <url>http://maven.aliyun.com/nexus/content/groups/public</url>
   ```

```
</mirror>
```

c. 下载 Analytics Zoo release 版本，解压后在目录下运行：

```
bash make-dist.sh
```

d. build 结束后，在 dist 目录中包含了所有的运行环境。将 dist 目录放到 EMR 软件栈运行时统一目录：

```
cp -r dist/ /usr/lib/analytics_zoo
```

2. Python 安装

Analytics Zoo 支持 pip 安装和非 pip 安装，pip 安装会安装 pyspark，bigdl等，由于EMR集群已经安装了 pyspark，通过 pip 安装有可能引起冲突，所以采用非 pip 安装。

- 非 pip 安装

首先要运行：

```
bash make-dist.sh
```

进入 pyzoo 目录，安装 analytcis zoo：

```
python setup.py install
```

3. 设置环境变量

在 scala 安装结束后将 dist 目录放到了 EMR 软件栈统一目录，然后设置环境变量。编辑 */etc/profile.d/analytics_zoo.sh*，加入：

```
export ANALYTICS_ZOO_HOME=/usr/lib/analytics_zoo
export PATH=$ANALYTICS_ZOO_HOME/bin:$PATH
```

EMR 已经设置了 SPARK_HOME，所以无需再次设置。

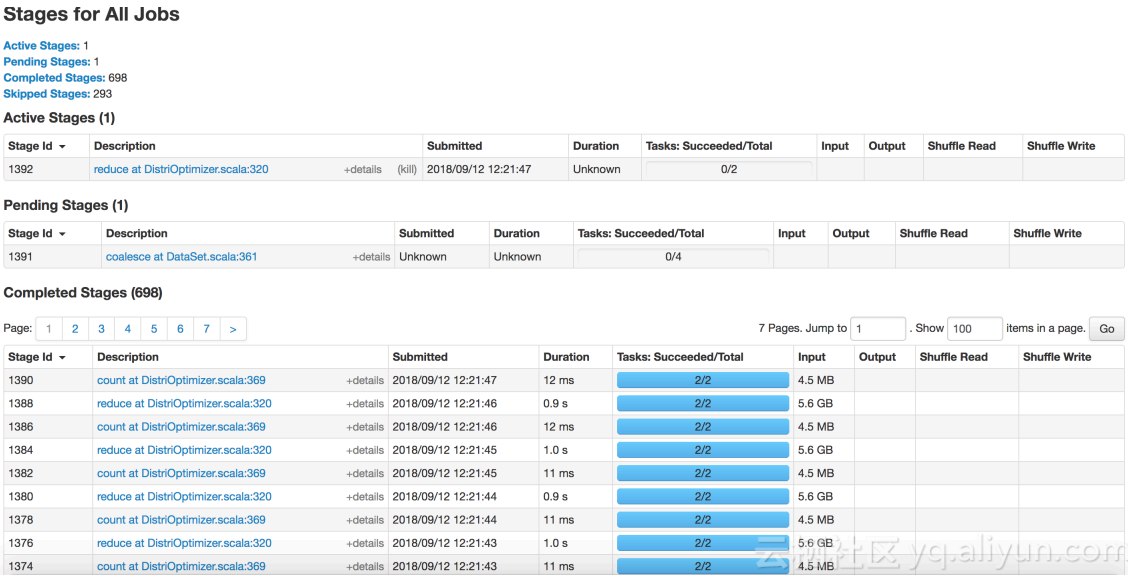
使用 Analytics Zoo

- 使用 Spark 来训练和测试深度学习模型
  - 使用 Analytics Zoo 来做文本分类，代码和说明在GitHub。根据说明下载必须的数据。提交命令：

```
spark-submit --master yarn \
--deploy-mode cluster --driver-memory 8g \
--executor-memory 20g --class com.intel.analytics.zoo.examples.
textclassification.TextClassification \
```

```
/usr/lib/analytics_zoo/lib/analytics-zoo-bigdl_0.6.0-spark_2.1.0-0
.2.0-jar-with-dependencies.jar --baseDir /news
```

- 通过 SSH proxy来查看 Spark 运行详情页面。



同时查看日志，能够看到每个 epoch 的 accuracy 信息等。

```
INFO optim.DistriOptimizer$: [Epoch 2 9600/15107][Iteration 194
][Wall Clock 193.266637037s] Trained 128 records in 0.958591653
 seconds. Throughput is 133.52922 records/second. Loss is 0.
74216986.
INFO optim.DistriOptimizer$: [Epoch 2 9728/15107][Iteration 195
][Wall Clock 194.224064816s] Trained 128 records in 0.957427779
 seconds. Throughput is 133.69154 records/second. Loss is 0.
51025534.
INFO optim.DistriOptimizer$: [Epoch 2 9856/15107][Iteration 196
][Wall Clock 195.189488678s] Trained 128 records in 0.965423862
seconds. Throughput is 132.58424 records/second. Loss is 0.553785.
INFO optim.DistriOptimizer$: [Epoch 2 9984/15107][Iteration 197
][Wall Clock 196.164318688s] Trained 128 records in 0.97483001
```

```
seconds. Throughput is 131.30495 records/second. Loss is 0.5517549
.
```

- 在 Analytics Zoo 中使用 pyspark 和 Jupyter 来进行深度学习训练

  1. 安装 Jupyter

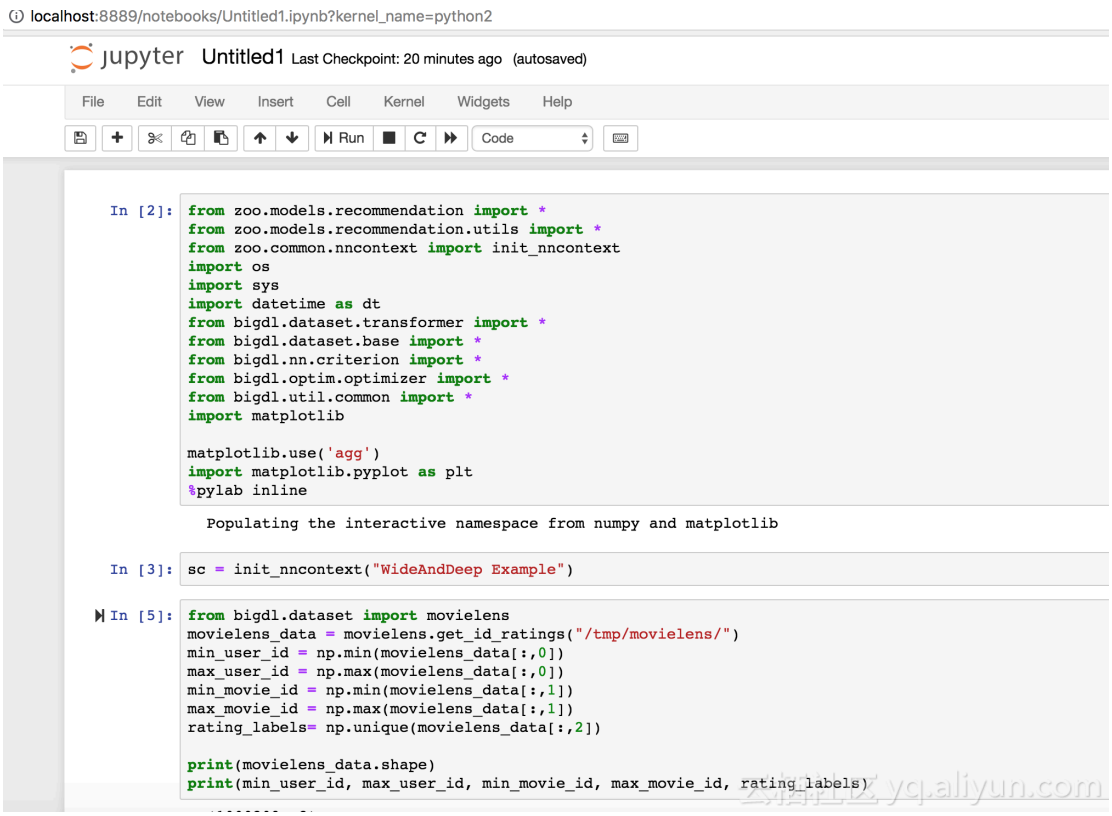
     ```
     pip install jupyter
     ```

  2. 使用以下命令启动：

     ```
     jupyter-with-zoo.sh
     ```

  3. 使用 Analytics Zoo，推荐采用内置的 Wide And Deep 模型来进行，相关内容可参见 GitHub。

     a. 导入数据



     b. 定义模型和优化器

```
In [10]: wide_n_deep = WideAndDeep(5, column_info, "wide_n_deep")
```

```
creating: createZooWideAndDeep
```

```
In [11]: # Create an Optimizer
batch_size = 8000

optimizer = Optimizer(
    model=wide_n_deep,
    training_rdd=train_data,
    criterion=ClassNLLCriterion(),
    optim_method=Adam(learningrate = 0.001, learningrate_decay=0.00005),
    end_trigger=MaxEpoch(10),
    batch_size=batch_size)

# Set the validation logic
optimizer.set_validation(
    batch_size=batch_size,
    val_rdd=test_data,
    trigger=EveryEpoch(),
    val_method=[Top1Accuracy(), Loss(ClassNLLCriterion())]
)
log_dir='/tmp/bigdl_summaries/'
app_name='wide_n_deep-'+dt.datetime.now().strftime("%Y%m%d-%H%M%S")
train_summary = TrainSummary(log_dir=log_dir,
                             app_name=app_name)
val_summary = ValidationSummary(log_dir=log_dir,
                                app_name=app_name)
optimizer.set_train_summary(train_summary)
optimizer.set_val_summary(val_summary)
print("saving logs to %s" % (log_dir + app_name))
```


c. 进行训练

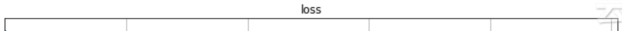
```
In [12]: %%time
# Boot training process
optimizer.optimize()
print("Optimization Done.")
```

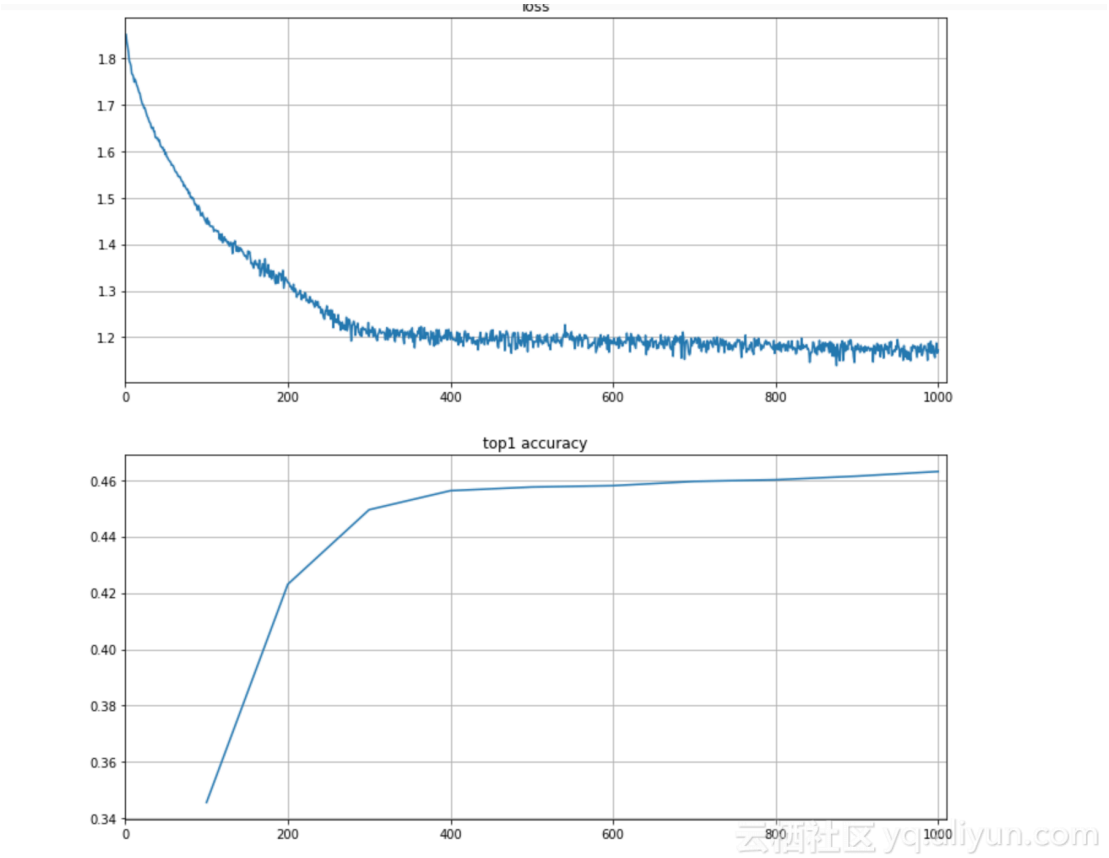
```
Optimization Done.
CPU times: user 85.9 ms, sys: 16.7 ms, total: 103 ms
Wall time: 2min 52s
```


d. 查看训练结果

```
In [16]: loss = np.array(train_summary.read_scalar("Loss"))
top1 = np.array(val_summary.read_scalar("Top1Accuracy"))

plt.figure(figsize = (12,12))
plt.subplot(2,1,1)
plt.plot(loss[:,0],loss[:,1],label='loss')
plt.xlim(0,loss.shape[0]+10)
plt.grid(True)
plt.title("loss")
plt.subplot(2,1,2)
plt.plot(top1[:,0],top1[:,1],label='top1')
plt.xlim(0,loss.shape[0]+10)
plt.title("top1 accuracy")
plt.grid(True)
```





7 SparkSQL 自适应执行

阿里云 E-MapReduce-3.13.0 版本的 SparkSQL 支持自适应执行功能，用来解决 Reduce 个数的动态调整/数据倾斜/执行计划的动态优化问题。

解决问题

SparkSQL 自适应执行解决以下问题:

- Shuffle partition个数

  目前 SparkSQL 中 reduce 阶段的 task 个数取决于固定参数 `spark.sql.shuffle.partition`(默认值 200)，一个作业一旦设置了该参数，它运行过程中的所有阶段的 reduce 个数都是同一个值。

  而对于不同的作业，以及同一个作业内的不同 reduce 阶段，实际的数据量大小可能相差很大，比如 reduce 阶段要处理的数据可能是 10MB，也有可能是 100GB, 如果使用同一个值对实际运行效率会产生很大影响，比如 10MB的数据一个 task 就可以解决，如果 `spark.sql.shuffle.partition`使用默认值 200 的话，那么 10MB 的数据就要被分成 200 个 task 处理，增加了调度开销，影响运行效率。

  SparkSQL 自适应框架可以通过设置 shuffle partition 的上下限区间，在这个区间内对不同作业不同阶段的 reduce 个数进行动态调整。

  通过区间的设置，一方面可以大大减少调优的成本(不需要找到一个固定值)，另一方面同一个作业内部不同 reduce 阶段的 reduce 个数也能动态调整。

  参数:

属性名称	默认值	备注
spark.sql.adaptive.enabled	false	自适应执行框架的开关
spark.sql.adaptive.minNumPostShufflePartitions	1	reduce 个数区间最小值
spark.sql.adaptive.maxNumPostShufflePartitions	500	reduce 个数区间最大值

属性名称	默认值	备注
spark.sql.adaptive.shuffle.targetPostShuffleInputSize	67108864	动态调整 reduce 个数的 partition 大小依据，如设置 64MB 则reduce 阶段每个 task 最少处理 64MB 的数据
spark.sql.adaptive.shuffle.targetPostShuffleRowCount	20000000	动态调整 reduce个数的 partition 条数依据，如设置 20000000 则reduce 阶段每个 task 最少处理 20000000 条的数据

- 数据倾斜

  Join 中会经常碰到数据倾斜的场景，导致某些 task 处理的数据过多，出现很严重的长尾。目前 SparkSQL 没有对倾斜的数据进行相关的优化处理。

  SparkSQL 自适应框架可以根据预先的配置在作业运行过程中自动检测是否出现倾斜，并对检测到的倾斜进行优化处理。

  优化的主要逻辑是对倾斜的 partition 进行拆分由多个 task 来进行处理，最后通过union 进行结果合并。

  支持的 Join 类型：

join类型	备注
Inner	左/右表均可处理倾斜
Cross	左/右表均可处理倾斜
LeftSemi	只对左表处理倾斜
LeftAnti	只对左表处理倾斜
LeftOuter	只对左表处理倾斜
RightOuter	只对右表处理倾斜

  参数：

属性名称	默认值	备注
spark.sql.adaptive.enabled	false	自适应执行框架的开关
spark.sql.adaptive.skewedJoin.enabled	false	倾斜处理开关

属性名称	默认值	备注
spark.sql.adaptive.skewedPartitionFactor	10	当一个 partition 的 size 大小 大于 该值(所有 parititon 大小的中位数) 且 大于spark.sql.adaptive.skewedPartitionSizeThreshold，或者 parition的条数 大于 该值(所有parititon条数的中位数) 且 大于 spark.sql.adaptive.skewedPartitionRowCountThreshold， 才会被当做倾斜的partition 进行相应的处理
spark.sql.adaptive.skewedPartitionSizeThreshold	67108864	倾斜的 partition 大小不能小于该值
spark.sql.adaptive.skewedPartitionRowCountThreshold	10000000	倾斜的 partition 条数不能小于该值
spark.shuffle.statistics.verbose	false	打开后 MapStatus会采集每个 partition 条数的信息，用于倾斜处理

- Runtime执行计划优化

  SparkSQL 的 Catalyst 优化器会将 sql 语句转换成物理执行计划，然后真正运行物理执行计划。但是 Catalyst 转换物理执行计划的过程中，由于缺少 Statistics 统计信息，或者 Statistics 统计信息不准等原因，会到时转换的物理执行计划可能并不是最优的，比如转换为 SortMergeJoinExec，但实际 BroadcastJoin 更合适。

  SparkSQL 自适应执行框架会在物理执行计划真正运行的过程中，动态的根据shuffle 阶段 shuffle write 的实际数据大小，来调整是否可以用 BroadcastJoin 来代替SortMergeJoin，提高运行效率。

  参数：

属性名称	默认值	备注
spark.sql.adaptive.enabled	false	自适应执行框架的开关
spark.sql.adaptive.join.enabled	true	开关

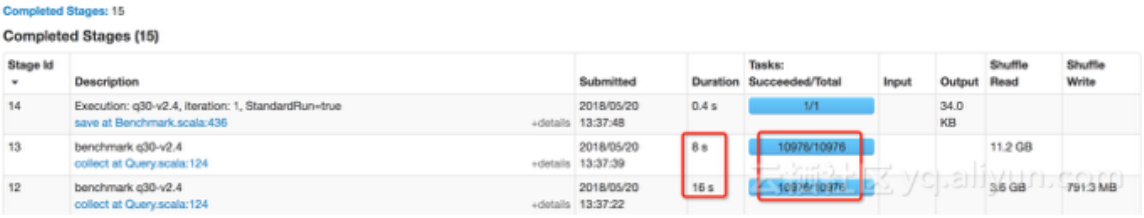
属性名称	默认值	备注
spark.sql.adaptiveBroadcastJoinThreshold	等于spark.sql.autoBroadcastJoinThreshold	运行过程中用于判断是否满足BroadcastJoin 条件

测试

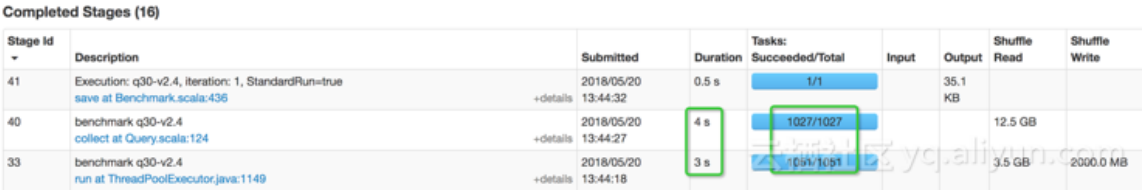
以TPC-DS 中某些 query 为例

- shuffle partition 个数
  - Query 30

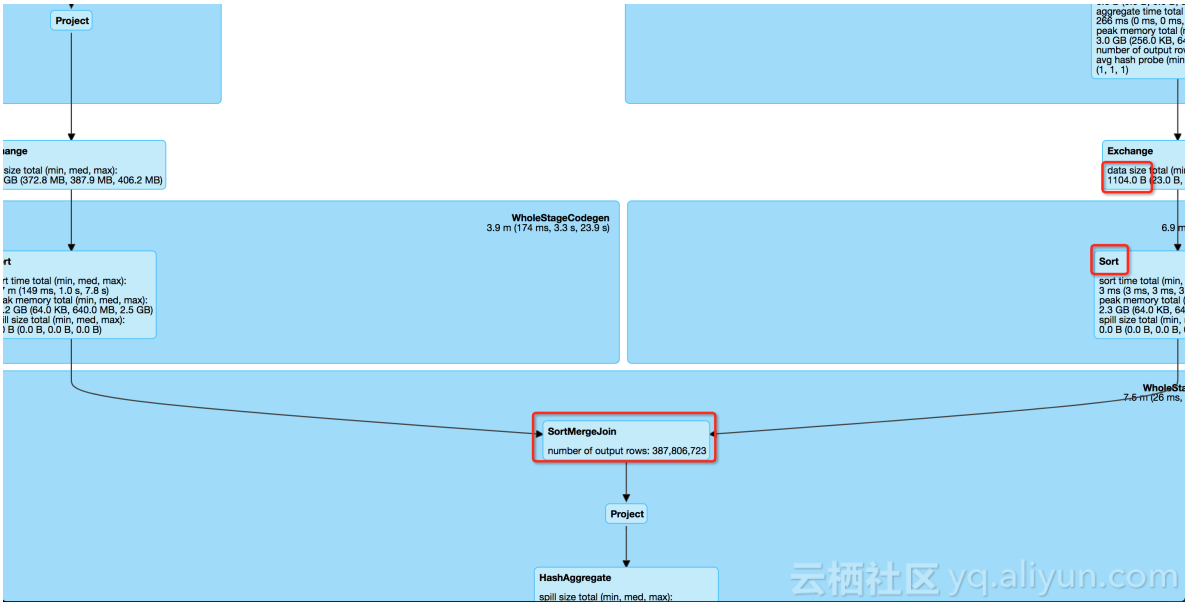
    原生Spark:



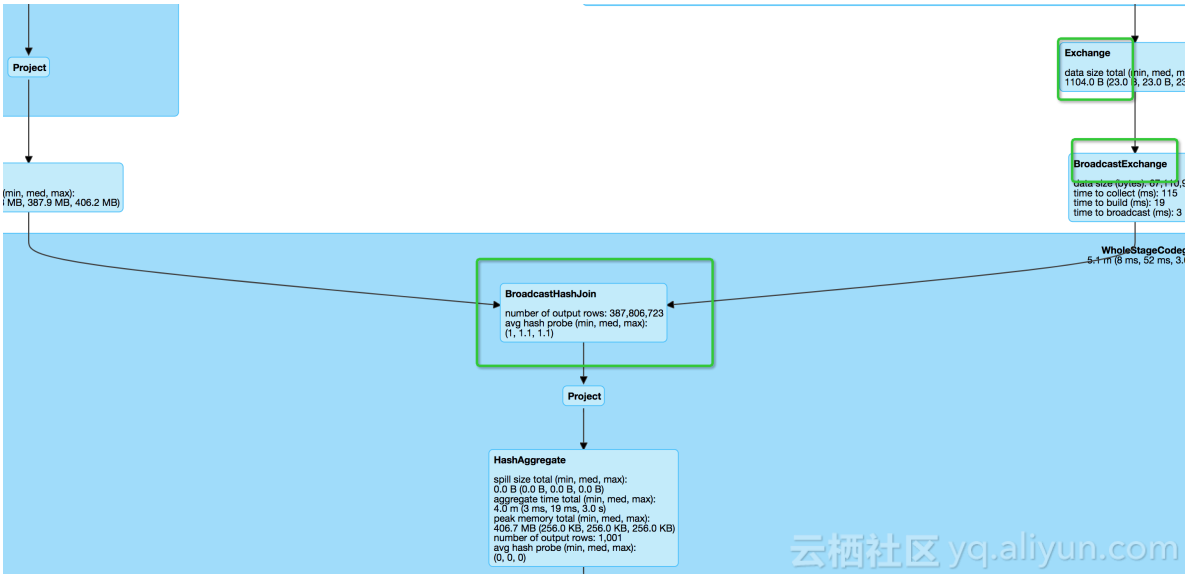
  - 自适应调整 reduce 个数:



- Runtime 执行计划优化(SortMergeJoin转BroadcastJoin)



自适应转换为 BroadcastJoin



8 E-MapReduce 数据迁移方案

在开发过程中我们通常会碰到需要迁移数据的场景，本篇介绍如何将自建集群数据迁移到 E-MapReduce 集群中。

适用范围：

- 线下 Hadoop 到 EMR 迁移
- 线上 ECS 自建 Hadoop 到 EMR 迁移

迁移场景：

- HDFS 增量上游数据源包括 RDS 增量数据、flume

新旧集群网络打通

- 线下 IDC 自建 Hadoop

  现在自建 Hadoop 迁移到 EMR 可以通过 OSS 进行过度，或者使用阿里云高速通道产品建立线下 IDC 和线上 EMR 所在 VPC 网络的连通。
- 利用 ECS 自建 Hadoop

  由于 VPC 实现用户专有网络之间的逻辑隔离，EMR 建议使用 VPC 网络。

  - 经典网络与 VPC 网络打通

    如果 ECS 自建 Hadoop，需要通过 ECS 的classiclink的方式将经典网络和 VPC 网络打通，参见建立 ClassicLink 连接。
  - VPC 网络之间连通

    数据迁移一般需要较高的网络带宽连通，建议新旧集群尽量处在同一个区域的同一个可用区内。

HDFS 数据迁移

- Distcp 工具同步数据

  请参考Hadoop DistCp 工具官方说明文档。

  HDFS 数据迁移可以通过 Hadoop 社区标准的 distcp 工具迁移，可以实现全量和增量的数据迁移。为减轻现有集群资源压力，建议在新旧集群网络连通后在新集群执行 `distcp` 命令。

  - 全量数据同步

    ```
    hadoop distcp -pbugpcax -m 1000 -bandwidth 30 hdfs://oldclusterip:8020/user/hive/warehouse /user/hive/warehouse
    ```

  - 增量数据同步

    ```
    hadoop distcp -pbugpcax -m 1000 -bandwidth 30  -update -delete hdfs://oldclusterip:8020/user/hive/warehouse /user/hive/warehouse
    ```

  参数说明：

  - *hdfs://oldclusterip:8020* 填写旧集群 namenode ip，多个namenode 情况填写当前状态为active的。
  - 默认副本数为 3，如想保留原有副本数，-p 后加 r 如 `-prbugpcax`。如果不同步权限和 ACL，-p 后去掉 p 和 a。
  - `-m` 指定 map 数，和集群规模，数据量有关。比如集群有 2000 核 CPU，就可以指定 2000 个 map。
  - `-bandwidth` 指定单个map的同步速度，是靠控制副本复制速度实现的，是大概值。
  - `-update`，校验源和目标文件的 checksum 和文件大小，如果不一致源文件会更新掉目标集群数据，新旧集群同步期间还有数据写入，可以通过 `-update` 做增量数据同步。
  - `-delete`，如果源集群数据不存在，新集群的数据也会被删掉。

  说明：

  - 迁移整体速度受集群间带宽，集群规模影响。同时文件越多，checksum需要的时间越长。如果迁移数据量大，可以先试着同步几个目录评估一下整体时间。如果只能在指定时间段内同步，可以将目录切为几个小目录，依次同步。
  - 一般全量数据同步，需要有个短暂的业务停写，以启用双写双算或直接将业务切换到新集群上。

- HDFS 权限配置

  HDFS 有权限设置，确定旧集群是否有 ACL 规则，是否要同步，检查 `dfs.permissions.enabled` 和 `dfs.namenode.acls.enabled` 的配置新旧集群是否一致，按照实际需要修改。

  如果有 ACL 规则要同步，`distcp` 参数后要加 `-p` 同步权限参数。如果 distcp 操作提示 xx 集群不支持 ACL，说明对应集群没配置 ACL 规则。新集群没配置 ACL 规则可以修改配置并重启 namenode。旧集群不支持，说明旧集群根本就没有 ACL 方面的设置，也不需要同步。

Hive元数据同步

- 概述

  Hive 元数据，一般存在 MySQL 里，与一般 MySQL 同步数据相比，要注意两点：

  - Location 变化
  - Hive 版本对齐

  EMR 支持 Hive Meta DB：

  - 统一元数据库，EMR管控RDS，每个用户一个 Schema
  - 用户自建 RDS
  - 用户 ECS 自建 MySQL

  为了保证迁移之后新旧数据完全一致，最好是在迁移的时候将老的 metastore 服务停掉，等迁移过去之后，再把旧集群上的 metastore 服务打开，然后新集群开始提交业务作业。

- 操作步骤：

  1. 将新集群的元数据库删除，直接输出命令 `drop database xxx`；
  2. 将旧集群的元数据库的表结构和数据通过 `mysqldump`命令全部导出；
  3. 替换 location，Hive 元数据中的表，分区等信息均带有 location 信息的，带dfs nameservices 前缀，如 *hdfs://mycluster:8020/*，而 EMR 集群的 nameservices 前缀是统一的 emr-cluster，所以需要订正。

     订正的最佳方式是先导出数据

     ```
     mysqldump --databases hivemeta --single-transaction -u root -p > hive_databases.sql
     ```

     用 sed 替换*hdfs://oldcluster:8020/* 为 *hdfs://emr-cluster/*，再导入新db中。

     ```
     mysql hivemeta -p < hive_databases.sql
     ```
  4. 在新集群的界面上，停止掉 hivemetastore 服务；
  5. 登陆新的元数据库，`create database` 创建数据库；
  6. 在新的元数据库中，导入替换 location 字段之后的老元数据库导出来的所有数据；
  7. 版本对齐，EMR 的 Hive 版本一般是当前社区最新的稳定版，自建集群Hive 版本可能会更老，所以导入的旧版本数据可能不能直接使用。需要执行 Hive的升级脚本（期间会有表、字段已存在的问题可以忽略），可以参见Hive升级脚本。例如 Hive 从 1.2 升级到 2.3.0，需要依次执行`upgrade-1.2.0-to-2.0.0.mysql.sql`，`upgrade-2.0.0-to-2.1.0.mysql.sql`，`upgrade-2.1.0-to-2.2.0.mysql.sql`，`upgrade-2.2.0-to-2.3.0.mysql.sql`。脚本主要是建表，加字段，改内容，如有表已存在，字段已存在的异常可以忽略。
  8. meta 数据全部订正后，就可以重启 metaserver 了。命令行输入`hive`，查询库和表，查询数据，验证数据的正确性。

Flume数据迁移

- Flume 双写配置

  在新集群上也开启 flume 服务，并且将数据按照和老集群完全一致的规则写入到新集群中。
- Flume 分区表写入

  Flume 数据双写，双写时需控制开始的时机，要保证 flume 在开始一个新的时间分区的时候来进行新集群的同步。 如 flume 每小时整点会同步所有的表，那就要整点之前，开启 flume 同步服务，这样 flume 在一个新的小时内写入的数据，在旧集群和新集群上是完全一致的。而不完整的旧数据在 distcp 的时候，全量的同步会覆盖它。而开启双写时间点后的新数据，在数据同

步的时候不进行同步。 这个新的写入的数据，我们在划分数据阶段，记得不要放到数据同步的目录里。

作业同步

Hadoop，Hive，Spark，MR等如果有较大的版本升级，可能涉及作业改造，要视具体情况而定。

常见问题：

- Gateway OOM

  修改 */etc/ecm/hive-conf/hive-env.sh*

  `export HADOOP_HEAPSIZE=512` 改成 1024

- 作业执行内存不足

  `mapreduce.map.java.opts` 调整的是启动 JVM 虚拟机时，传递给虚拟机的启动参数，而默认值 -Xmx200m 表示这个 Java 程序可以使用的最大堆内存数，一旦超过这个大小，JVM 就会抛出 Out of Memory 异常，并终止进程

  ```
  set mapreduce.map.java.opts=-Xmx3072m
  ```

  mapreduce.map.memory.mb 设置的是 Container 的内存上限，这个参数由 NodeManager 读取并进行控制，当 Container 的内存大小超过了这个参数值，NodeManager 会负责 kill Container。

  ```
  set mapreduce.map.memory.mb=3840
  ```

数据校验

由客户自行抽检报表完成

Presto集群迁移

如果有单独的Presto集群仅仅用来做数据查询，需要修改 Hive 中配置文件，请参见Presto文档。

需要修改hive.properties：

- `connector.name=hive-hadoop2`
- `hive.metastore.uri=thrift://emr-header-1.cluster-500148414:9083`
- `hive.config.resources=/etc/ecm/hadoop-conf/core-site.xml, /etc/ecm/hadoop-conf/hdfs-site.xml`
- `hive.allow-drop-table=true`
- `hive.allow-rename-table=true`

- hive.recursive-directories=true

附录

Hive 1.2 升级到 2.3 的版本对齐示例：

```
source /usr/lib/hive-current/scripts/metastore/upgrade/mysql/upgrade-1
.2.0-to-2.0.0.mysql.sql
CREATE TABLE COMPACTION_QUEUE (
  CQ_ID bigint PRIMARY KEY,
  CQ_DATABASE varchar(128) NOT NULL,
  CQ_TABLE varchar(128) NOT NULL,
  CQ_PARTITION varchar(767),
  CQ_STATE char(1) NOT NULL,
  CQ_TYPE char(1) NOT NULL,
  CQ_WORKER_ID varchar(128),
  CQ_START bigint,
  CQ_RUN_AS varchar(128),
  CQ_HIGHEST_TXN_ID bigint,
  CQ_META_INFO varbinary(2048),
  CQ_HADOOP_JOB_ID varchar(32)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
CREATE TABLE TXNS (
  TXN_ID bigint PRIMARY KEY,
  TXN_STATE char(1) NOT NULL,
  TXN_STARTED bigint NOT NULL,
  TXN_LAST_HEARTBEAT bigint NOT NULL,
  TXN_USER varchar(128) NOT NULL,
  TXN_HOST varchar(128) NOT NULL,
  TXN_AGENT_INFO varchar(128),
  TXN_META_INFO varchar(128),
  TXN_HEARTBEAT_COUNT int
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
CREATE TABLE HIVE_LOCKS (
  HL_LOCK_EXT_ID bigint NOT NULL,
  HL_LOCK_INT_ID bigint NOT NULL,
  HL_TXNID bigint,
  HL_DB varchar(128) NOT NULL,
  HL_TABLE varchar(128),
  HL_PARTITION varchar(767),
  HL_LOCK_STATE char(1) not null,
  HL_LOCK_TYPE char(1) not null,
  HL_LAST_HEARTBEAT bigint NOT NULL,
  HL_ACQUIRED_AT bigint,
  HL_USER varchar(128) NOT NULL,
  HL_HOST varchar(128) NOT NULL,
  HL_HEARTBEAT_COUNT int,
  HL_AGENT_INFO varchar(128),
  HL_BLOCKEDBY_EXT_ID bigint,
  HL_BLOCKEDBY_INT_ID bigint,
  PRIMARY KEY(HL_LOCK_EXT_ID, HL_LOCK_INT_ID),
  KEY HIVE_LOCK_TXNID_INDEX (HL_TXNID)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
CREATE INDEX HL_TXNID_IDX ON HIVE_LOCKS (HL_TXNID);
source /usr/lib/hive-current/scripts/metastore/upgrade/mysql/upgrade-1
.2.0-to-2.0.0.mysql.sql
source /usr/lib/hive-current/scripts/metastore/upgrade/mysql/upgrade-2
.0.0-to-2.1.0.mysql.sql

CREATE TABLE TXN_COMPONENTS (
  TC_TXNID bigint,
  TC_DATABASE varchar(128) NOT NULL,
```

```
  TC_TABLE varchar(128),
  TC_PARTITION varchar(767),
  FOREIGN KEY (TC_TXNID) REFERENCES TXNS (TXN_ID)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
source /usr/lib/hive-current/scripts/metastore/upgrade/mysql/upgrade-2
.0.0-to-2.1.0.mysql.sql
source /usr/lib/hive-current/scripts/metastore/upgrade/mysql/upgrade-2
.1.0-to-2.2.0.mysql.sql
CREATE TABLE IF NOT EXISTS `NOTIFICATION_LOG`
(
    `NL_ID` BIGINT(20) NOT NULL,
    `EVENT_ID` BIGINT(20) NOT NULL,
    `EVENT_TIME` INT(11) NOT NULL,
    `EVENT_TYPE` varchar(32) NOT NULL,
    `DB_NAME` varchar(128),
    `TBL_NAME` varchar(128),
    `MESSAGE` mediumtext,
    PRIMARY KEY (`NL_ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
CREATE TABLE IF NOT EXISTS `PARTITION_EVENTS` (
  `PART_NAME_ID` bigint(20) NOT NULL,
  `DB_NAME` varchar(128) CHARACTER SET latin1 COLLATE latin1_bin
DEFAULT NULL,
  `EVENT_TIME` bigint(20) NOT NULL,
  `EVENT_TYPE` int(11) NOT NULL,
  `PARTITION_NAME` varchar(767) CHARACTER SET latin1 COLLATE
latin1_bin DEFAULT NULL,
  `TBL_NAME` varchar(128) CHARACTER SET latin1 COLLATE latin1_bin
DEFAULT NULL,
  PRIMARY KEY (`PART_NAME_ID`),
  KEY `PARTITIONEVENTINDEX` (`PARTITION_NAME`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE COMPLETED_TXN_COMPONENTS (
  CTC_TXNID bigint NOT NULL,
  CTC_DATABASE varchar(128) NOT NULL,
  CTC_TABLE varchar(128),
  CTC_PARTITION varchar(767)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
 source /usr/lib/hive-current/scripts/metastore/upgrade/mysql/upgrade-
2.1.0-to-2.2.0.mysql.sql
 source /usr/lib/hive-current/scripts/metastore/upgrade/mysql/upgrade-
2.2.0-to-2.3.0.mysql.sql
  CREATE TABLE NEXT_TXN_ID (
  NTXN_NEXT bigint NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
INSERT INTO NEXT_TXN_ID VALUES(1);
CREATE TABLE NEXT_LOCK_ID (
  NL_NEXT bigint NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
INSERT INTO NEXT_LOCK_ID VALUES(1);
```

9 在阿里云 E-MapReduce 上使用 Data Science 集群进行深度学习

Data Science 集群是阿里云 E-MapReduce 在 3.13.0 版本以后推出的专门用于机器学习，深度学习的新的机型。客户可以通过 Data Science 集群选用 GPU 或者 CPU 机型对数据进行训练，训练的数据可以存储在 HDFS 和 OSS 上，目前支持 TensorFlow 进行分布式训练，方便用户开发基于大数据存储，分布式调度的深度学习应用。

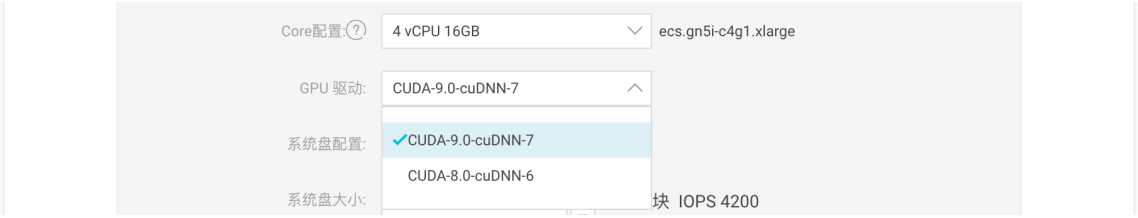
集群创建

- EMR Data Science 集群创建有以下要求：
  - EMR 3.13.0 版本及以上
  - 集群类型选取 Data Science 类型

- **创建集群**

在机型的选择上，Master 节点用户选取 CPU 机型即可，根据需要选择。Core 节点用户可以选取 GPU 机型。

如果用户选择了 GPU 机型，EMR 会提供 Nvidia GPU 驱动以及对应 Cudnn 安装，用户选择对应的驱动进行安装，方便使用。



至此，集群创建完毕，对应服务会被安装，驱动以及 Cudnn 将会被安装，同时在所有core 节点上也会安装 docker 服务，用于深度学习训练工具使用。

在Data Science集群上运行TensorFlow

- TensorFlow

  TensorFlow 是谷歌开源的深度学习框架，用于机器学习任务和训练神经网络模型等深度学习。更多的关于 TensorFlow 的信息可以参见TensorFlow官网。

- TensorFlow on YARN

  TensorFlow on YARN 是 EMR 内核团队开发的基于 YARN 调度的分布式TensorFlow 运行框架，支持在 YARN 上运行 TensorFlow job 并运用 GPU 能力来进行训练。相应的使用说明请参考TensorFlow on YARN使用说明。

- 使用 TensorFlow on YARN 进行深度学习

  目前 TensorFlow on YARN 支持使用高阶 API 进行训练，代码更加简洁。选取Wide And Deep 来进行训练，模型可以参见github。下载的数据地址链接。训练需要的数据为adult.data和adult.test 。用户按照单机版本通过 python 来写训练步骤：

  1. 首先定义好训练数据，上传训练数据和验证数据到 HDFS 上。

     将训练数据放到 HDFS 的 */ml/* 目录下：

     ```
     hdfs dfs -put adut.data adult.test /ml/
     ```

  2. 在训练代码中指定训练数据路径：

     ```
     TRAIN_FILES = ['hdfs://emr-header-1.cluster-500157403:9000/ml/
     adult.data']
     EVAL_FILES = ['hdfs://emr-header-1.cluster-500157403:9000/ml/adult
     .test']
     ```

     其中 HDFS 的 Schema 需要用户根据自己的集群来设置，如果不是 HA 集群，请查找配置管理中 *core-site.xml* 中的 `fs.defaultFS` 属性。如果是 HA 集群，则默认为 *emr-cluster*。

  3. 定义特征列

     根据 Wide and Deep，分别定义对应特征：

     ```
     """Build a wide and deep model for predicting income category.
     ```

```
"""
(gender, race, education, marital_status, relationship,
workclass, occupation, native_country, age,
education_num, capital_gain, capital_loss, hours_per_week) =
INPUT_COLUMNS
# Continuous columns can be converted to categorical via
bucketization
age_buckets = tf.feature_column.bucketized_column(
age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
# Wide columns and deep columns.
wide_columns = [
# Interactions between different categorical features can also
# be added as new virtual features.
tf.feature_column.crossed_column(
['education', 'occupation'], hash_bucket_size=int(1e4)),
tf.feature_column.crossed_column(
[age_buckets, race, 'occupation'], hash_bucket_size=int(1e6)),
tf.feature_column.crossed_column(
['native_country', 'occupation'], hash_bucket_size=int(1e4)),
gender,
native_country,
education,
occupation,
workclass,
marital_status,
relationship,
age_buckets,
]
deep_columns = [
# Use indicator columns for low dimensional vocabularies
tf.feature_column.indicator_column(workclass),
tf.feature_column.indicator_column(education),
tf.feature_column.indicator_column(marital_status),
tf.feature_column.indicator_column(gender),
tf.feature_column.indicator_column(relationship),
tf.feature_column.indicator_column(race),
# Use embedding columns for high dimensional vocabularies
tf.feature_column.embedding_column(
native_country, dimension=embedding_size),
tf.feature_column.embedding_column(occupation, dimension=
embedding_size),
age,
education_num,
capital_gain,
capital_loss,
hours_per_week,
]
```

4. **定义 input_fn**

   input_fn 方法用于用户获取训练数据。

```
def input_fn(filenames,
num_epochs=None,
shuffle=True,
skip_header_lines=0,
batch_size=200):
"""Generates features and labels for training or evaluation.
"""
dataset = tf.data.TextLineDataset(filenames).skip(skip_heade
r_lines).map(parse_csv)
if shuffle:
dataset = dataset.shuffle(buffer_size=batch_size * 10)
```

```
dataset = dataset.repeat(num_epochs)
dataset = dataset.batch(batch_size)
iterator = dataset.make_one_shot_iterator()
features = iterator.get_next()
return features, parse_label_column(features.pop(LABEL_COLUMN))
train_input = lambda: input_fn(
TRAIN_FILES,
batch_size=40
)
# Don't shuffle evaluation data
eval_input = lambda: input_fn(
EVAL_FILES,
batch_size=40,
shuffle=False
)
```

5. **初始化 Estimator**

   这里我们使用 TensorFlow 预定义的 Wide And Deep 模型来构建 Estimator。

```
tf.estimator.DNNLinearCombinedClassifier(
config=config,
linear_feature_columns=wide_columns,
dnn_feature_columns=deep_columns,
dnn_hidden_units=hidden_units or [100, 70, 50, 25]
)
```

6. **模型训练**

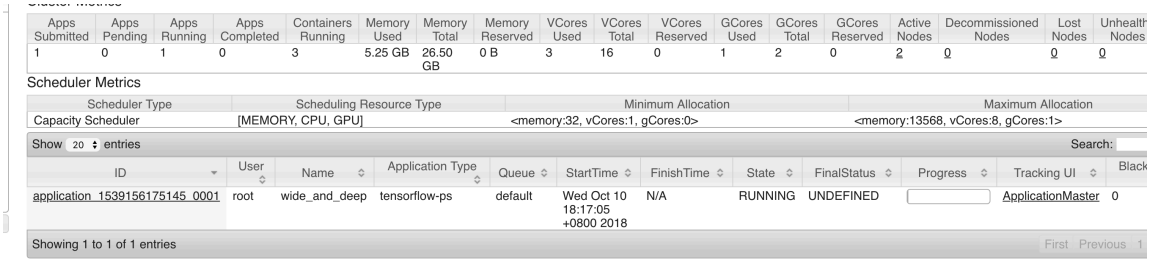
```
train_spec = tf.estimator.TrainSpec(train_input,
max_steps=1000
)
exporter = tf.estimator.FinalExporter('census',
json_serving_input_fn)
eval_spec = tf.estimator.EvalSpec(eval_input,
steps=100,
exporters=[exporter],
name='census-eval'
)
```

```
tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

代码完成后用户可以向集群提交任务，推荐用户使用 standalone 模型先将训练任务发送到集群中进行一个单机训练，验证代码的正确性，当确认任务没有问题后，可以提交分布式版本，指定ps worker的资源进行训练。示例的提交命令如下：

```
el_submit -t tensorflow-ps -a wide_and_deep -m local -x True -
f ./ -pn 1 -pc 1 -pm 2000 -wn 1 -wc 1 -wg 1 -wm 2000 -c python
census_single.
```

任务提交后可以到 YARN 页面查看任务运行情况



Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	GCores Used	GCores Total	GCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealth Nodes
1	0	1	0	3	5.25 GB	26.50 GB	0 B	3	16	0	1	2	0	2	0	0	0

Scheduler Metrics

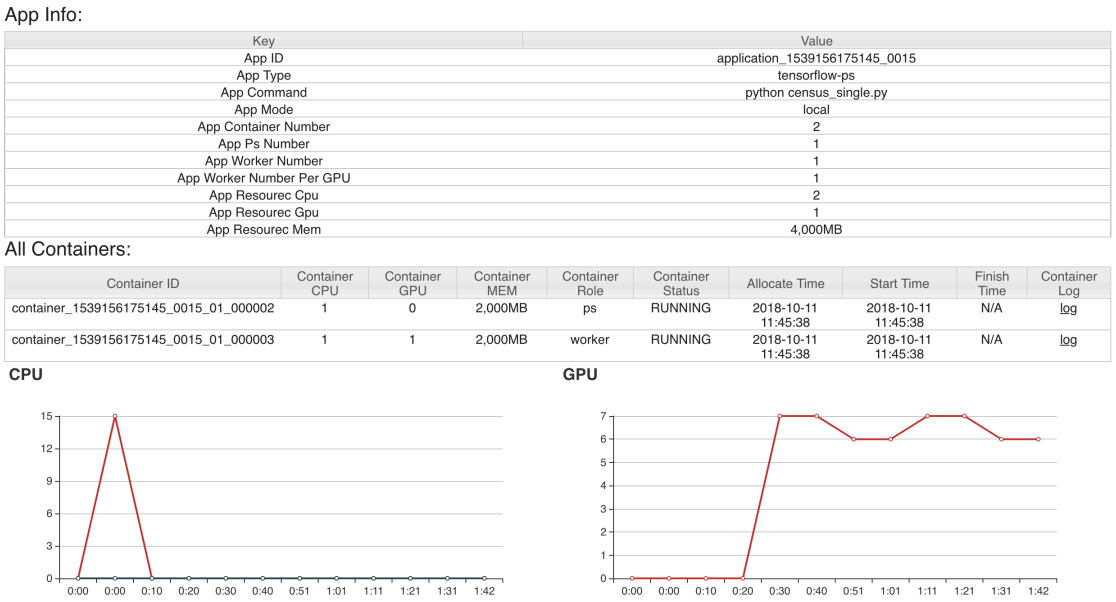
Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY, CPU, GPU]	<memory:32, vCores:1, gCores:0>	<memory:13568, vCores:8, gCores:1>

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Black
application_1539156175145_0001	root	wide_and_deep	tensorflow-ps	default	Wed Oct 10 18:17:05 +0800 2018	N/A	RUNNING	UNDEFINED		ApplicationMaster	0

Showing 1 to 1 of 1 entries First Previous 1

单击 ApplicationMaster 链接后可以看到任务的运行情况和详细信息



App Info:

Key	Value
App ID	application_1539156175145_0015
App Type	tensorflow-ps
App Command	python census_single.py
App Mode	local
App Container Number	2
App Ps Number	1
App Worker Number	1
App Worker Number Per GPU	1
App Resourec Cpu	2
App Resourec Gpu	1
App Resourec Mem	4,000MB

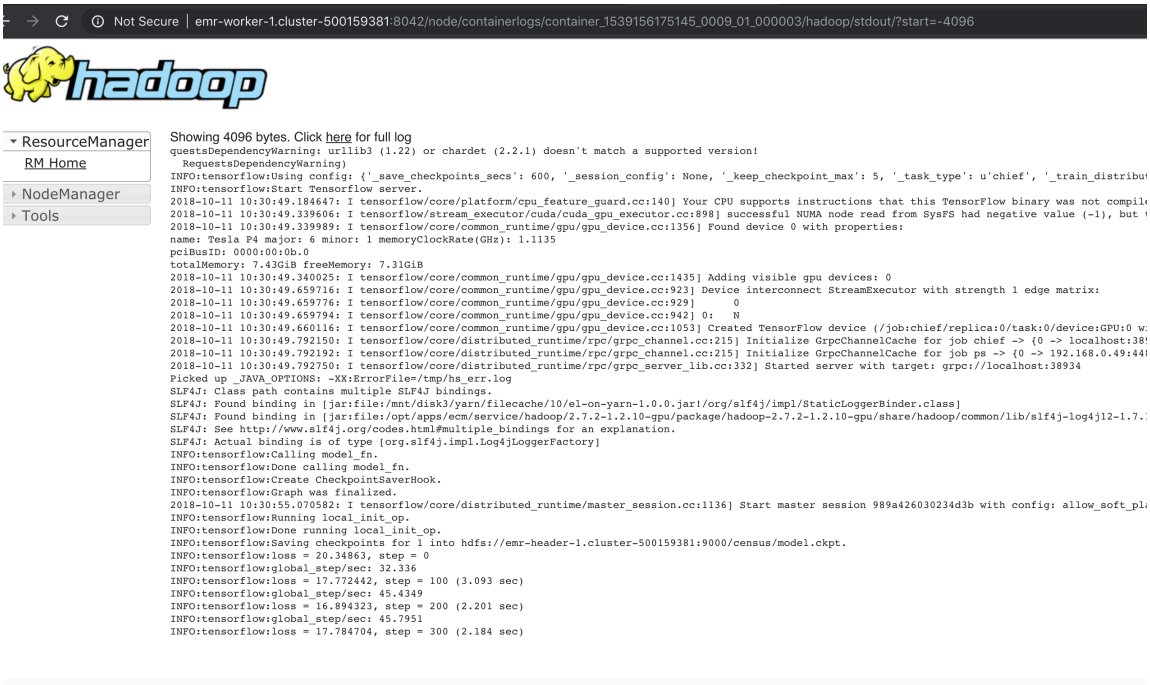
All Containers:

Container ID	Container CPU	Container GPU	Container MEM	Container Role	Container Status	Allocate Time	Start Time	Finish Time	Container Log
container_1539156175145_0015_01_000002	1	0	2,000MB	ps	RUNNING	2018-10-11 11:45:38	2018-10-11 11:45:38	N/A	log
container_1539156175145_0015_01_000003	1	1	2,000MB	worker	RUNNING	2018-10-11 11:45:38	2018-10-11 11:45:38	N/A	log

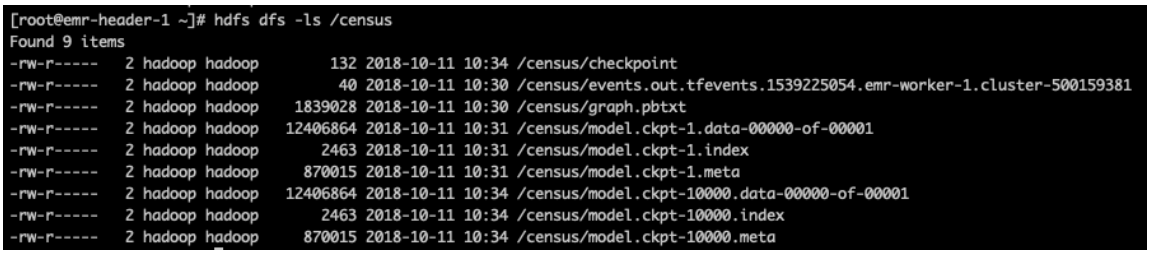
CPU

GPU

单击 log 后能够跳转到 ps 或者 worker 上查看训练信息。



本示例中，模型最后输出到 HDFS 的路径*/census*中，训练结束后可以在 HDFS 中找到输出的模型。



问题说明

如果报以下错误：

```
tensorflow.python.framework.errors_impl.InvalidArgumentError: Expect
15 fields but have 0 in record 0
[[Node: DecodeCSV = DecodeCSV[OUT_TYPE=[DT_INT32, DT_STRING, DT_INT32
, DT_STRING, DT_INT32, ..., DT_INT32, DT_INT32, DT_INT32, DT_STRI
NG, DT_STRING], field_delim=",", na_value="", use_quote_delim=true](
ExpandDims, DecodeCSV/record_defaults_0, DecodeCSV/record_defaults_1,
D
ecodeCSV/record_defaults_2, DecodeCSV/record_defaults_3, DecodeCSV/
record_defaults_4, DecodeCSV/record_defaults_5, DecodeCSV/record_def
ault
s_6, DecodeCSV/record_defaults_7, DecodeCSV/record_defaults_8,
DecodeCSV/record_defaults_9, DecodeCSV/record_defaults_10, DecodeCSV/
record_
defaults_11, DecodeCSV/record_defaults_12, DecodeCSV/record_def
aults_13, DecodeCSV/record_defaults_14)]]
[[Node: IteratorGetNext = IteratorGetNext[output_shapes=[[?,1], [?,1],
 [?,1], [?,1], [?,1], [?,1], [?,1], [?,1], [?,1], [?,1], [?,1],
[?,1], [?,1], [?,1]], output_types=[DT_INT32, DT_INT32, DT_INT32
, DT_STRING, DT_INT32, DT_STRING, DT_INT32, DT_STRING, DT_STRING,
DT_STRING
```

```
, DT_STRING, DT_STRING, DT_STRING, DT_STRING], _device="/job:chief/
replica:0/task:0/device:CPU:0"](OneShotIterator)]]
[[Node: global_step/cond/pred_id_S615 = _HostRecv[client_terminated=
false, recv_device="/job:ps/replica:0/task:0/device:CPU:0", send_d
evice="/job:chief/replica:0/task:0/device:GPU:0", send_device_incarnat
ion=6104642431418663740, tensor_name="edge_602_global_step/cond/pred_
id", tensor_type=DT_BOOL, _device="/job:ps/replica:0/task:0/device:CPU
:0"]()]]
2
```

检查一下 adult.data 和 adult.test 是否有空行存在。

10 通过Flink作业处理OSS数据

本节介绍如何在E-MapReduce上创建Hadoop集群，然后在Hadoop集群中运行Flink作业来消费OSS数据。

前提条件

- 已注册阿里云账号，详情请参见注册云账号。
- 已开通E-MapReduce服务和OSS服务。
- 已完成云账号的授权，详情请参见#unique_23。

背景信息

在开发过程中，通常会遇到需要消费存储在阿里云OSS中的数据的场景。在阿里云E-MapReduce中，您可通过运行Flink作业来消费OSS存储空间中的数据。本节将在E-MapReduce上创建一个Flink作业，然后在Hadoop集群上运行这个Flink作业来读取并打印OSS中指定文件的内容。

步骤一 准备环境

在创建Flink作业前，您需要在本地安装Maven和Java环境，以及在E-MapReduce上创建Hadoop集群。如果Maven是3.0以上版本，则建议Java选择2.0及以下版本，否则会造成不兼容情况。

1. 在本地安装Maven和Java环境。
2. 登录阿里云 E-MapReduce 控制台。
3. 创建Hadoop集群（可选服务中必须选中Flink服务），详情请参见创建集群。

步骤二 准备测试数据

在创建Flink作业前，您需要在OSS上传测试数据。本例以上传一个`test.txt`文件为例，文件内容为：Nothing is impossible for a willing heart. While there is a life, there is a hope~。

1. 登录 OSS 管理控制台。
2. 创建存储空间并上传测试数据文件，详情请参见#unique_25和#unique_26。

   测试数据的上传路径在后续步骤的代码中会使用，本例的上传路径为`oss://emr-logs2/hengwu/test.txt`。

   > **说明:**
   > 上传文件后，请保留OSS的登录窗口，后续仍会使用。

步骤三 制作JAR包并上传到OSS或Hadoop集群

本例JAR包来源：下载E-MapReduce示例代码aliyun-emapreduce-demo，编译生成JAR包。JAR包可上传到Hadoop集群的header主机中，也可上传到OSS中，本例以上传到OSS为例。

1. 下载E-MapReduce示例代码aliyun-emapreduce-demo到本地。
2. 运行`mvn clean package -DskipTests`命令打包代码。

   打包好的JAR包为存储在*../target/*目录下，例如，*target/examples-1.2.0.jar*。
3. 返回到 OSS 管理控制台。
4. 上传JAR包到OSS任一路径下。

   JAR包的上传路径在后续步骤的代码中会使用，本例的上传路径为*oss://emr-logs2/hengwu/examples-1.2.0.jar*。

步骤四 创建并运行Flink作业

1. 返回到阿里云 E-MapReduce 控制台。
2. 在数据开发页面创建项目，详情请参见#unique_27。
3. 进入新建的项目，在作业编辑页面新建Flink类型的作业。
4. 新建Flink作业后，配置其作业内容。

作业内容是一段代码，本节使用的示例代码如下：

```
run -m yarn-cluster  -yjm 1024 -ytm 1024 -yn 4 -ys 4 -ynm flink-oss-
sample -c com.aliyun.emr.example.flink.FlinkOSSSample  ossref://emr-
```

```
logs2/hengwu/examples-1.2.0.jar --input oss://emr-logs2/hengwu/test.
txt
```

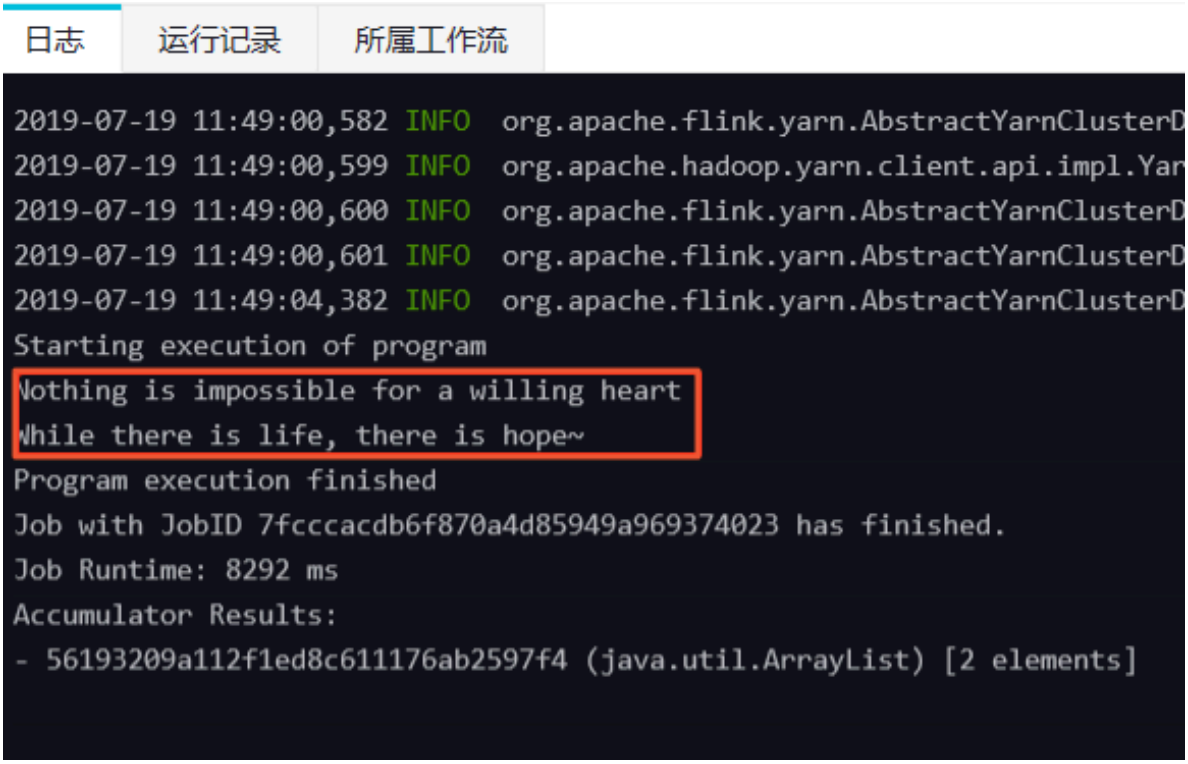
示例代码中的关键参数说明如下：

- *ossref://emr-logs2/hengwu/examples-1.2.0.jar*：上传至OSS的JAR包。
- *oss://emr-logs2/hengwu/test.txt*：上传到OSS的测试数据。

> **说明：**
> 实际操作时，您需要根据步骤一 准备环境和步骤三 制作JAR包并上传到OSS或Hadoop集群中的配置来替换这两个参数。

5. 作业配置完成后，单击右上方的运行，在弹出的对话框中选择执行集群为新建的Hadoop集群。
6. 单击确定，运行Flink作业。

作业开始运行时，会自动弹出日志。作业成功运行后，会从OSS读取指定文件内容并打印在日志中。至此，我们成功实现了在E-MapReduce集群上运行Flink作业消费OSS数据。



步骤五 查看作业提交日志和作业信息（可选）

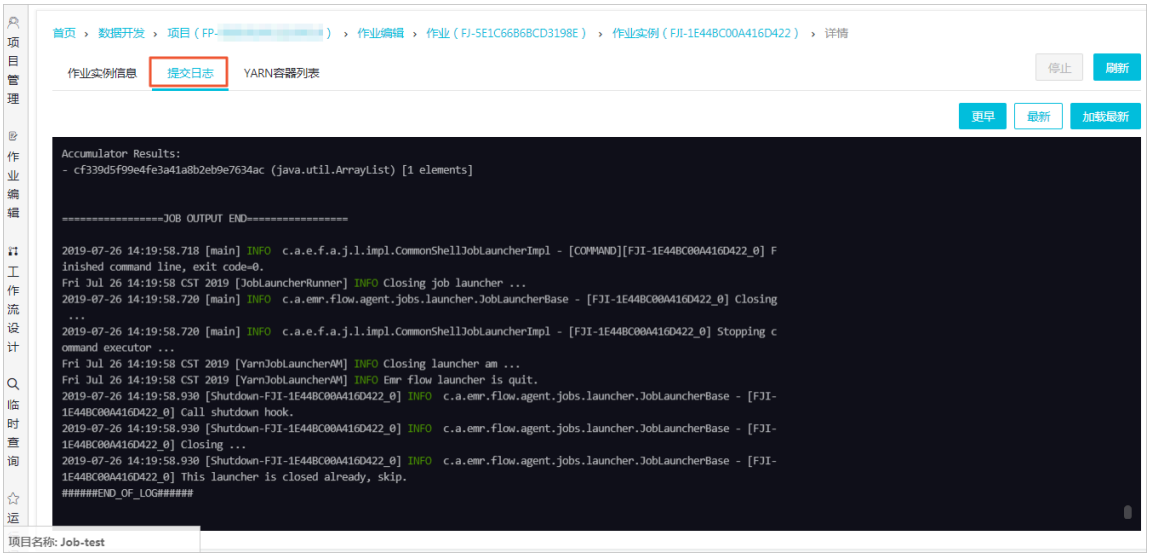
如果需要定位作业失败的原因或了解作业的详细信息，则您可查看作业的日志和作业信息。

1. 查看作业提交日志。

   当前提交日志支持在E-MapReduce控制台查看，也支持在SSH客户端查看。

   - 登录阿里云 E-MapReduce 控制台查看提交日志。

     在控制台提交作业后，可通过运行记录列表进入某次作业运行的详情页面，在详情页面可查看作业的日志。



   - 通过SSH客户端登录到Hadoop集群的header主机查看提交日志。

     默认情况下，根据Flink的log4j配置（详情请参见*/etc/ecm/flink-conf/log4j-yarn-session.properties*），提交日志会保存在*/mnt/disk1/log/flink/flink-{user}-client-{hostname}.log*。

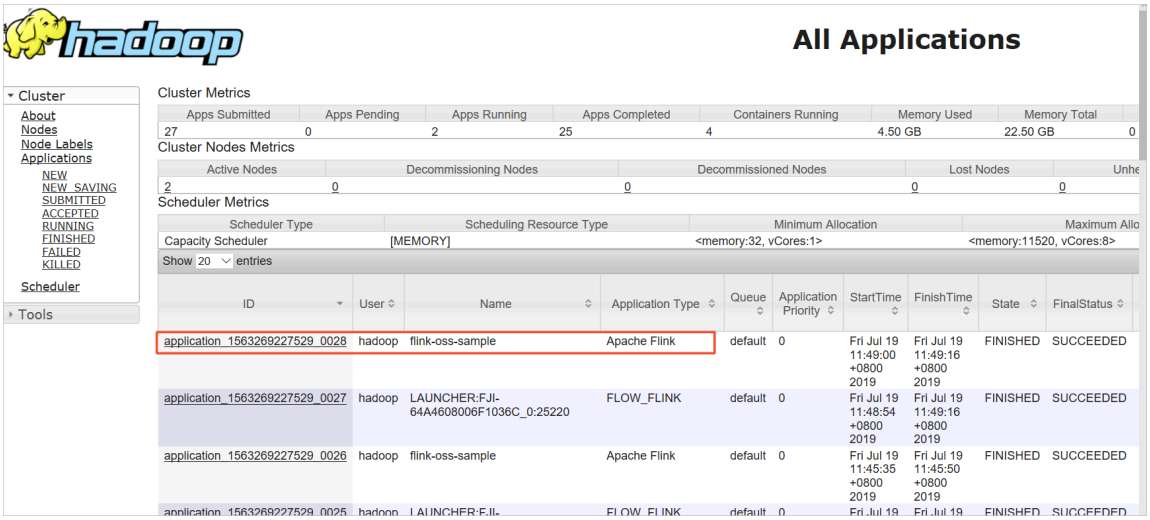
     其中，user为提交Flink作业的用户，hostname为提交作业所在的节点。以root用户在emr-header-1节点提交Flink作业为例，日志的路径为*/mnt/disk1/log/flink/flink-flink-historyserver-0-emr-header-1.cluster-126601.log*。

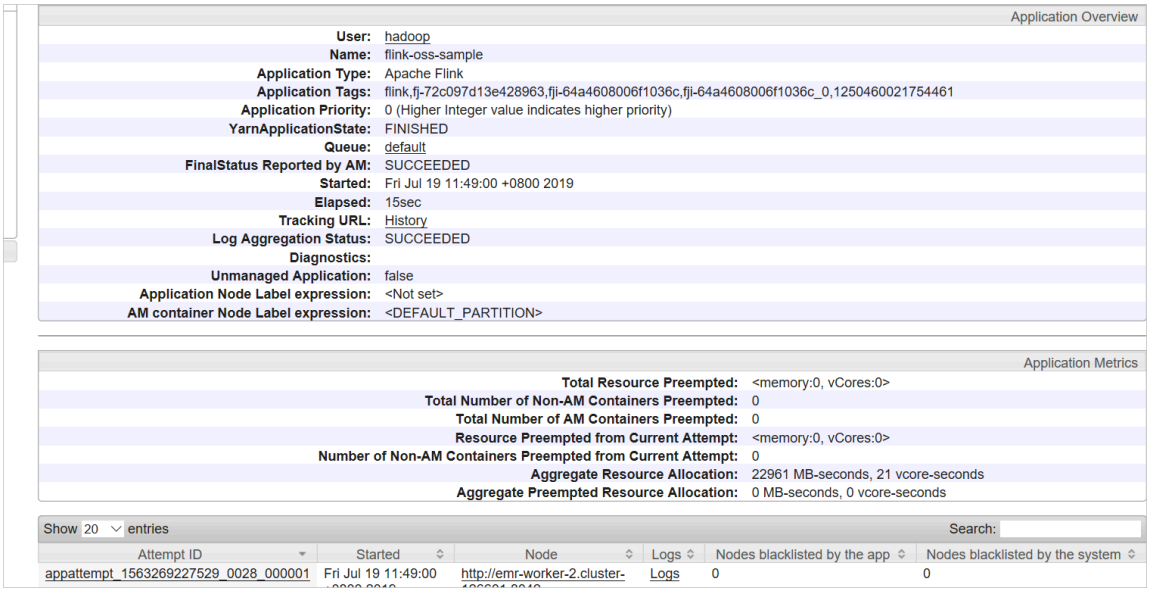
2. 查看作业信息。

通过Yarn UI可查看Flink作业的信息。访问Yarn UI有SSH隧道和Knox两种方式，SSH隧道方式请参见#unique_28，Knox方式请参见#unique_29和#unique_30。下面以Knox方式为例进行介绍。

a) 在Hadoop集群的访问链接与端口页面中，单击Yarn UI后的链接，进入Hadoop控制台。

b) 在Hadoop控制台，单击作业的ID，查看作业运行详情。





c) 如果需要查看运行中的Flink作业，则可在作业详情页面单击Tracking URL后面的链接，进入Flink Dashboard查看。

d) 作业运行结束后，通过访问http://emr-header-1:8082，可以查看所有已经完成的作业列表。

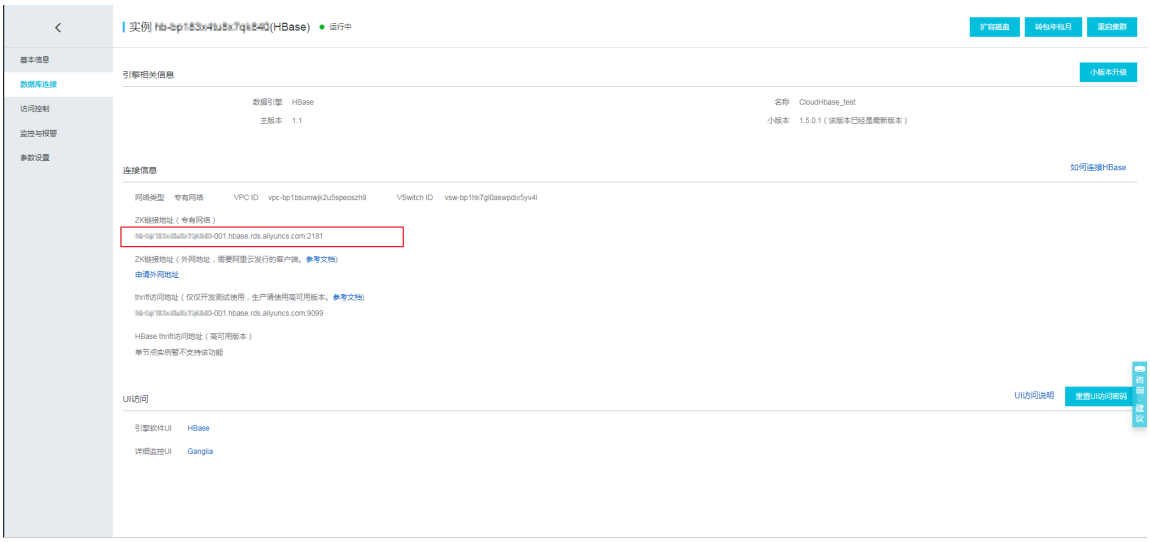
11 使用 E-MapReduce Hive 关联云 HBase

本文简单介绍如何使用 E-MapReduce Hive 关联云 HBase 的表。云HBase 需要借助外部 Hive 对多表进行关联分析。

> **说明:**
> 后续云 HBase 将集成 Spark，更加建议使用Spark 分析 HBase 数据。

准备工作

- 购买按量计费的 EMR 集群，配置依据实际场景确定，注意云 HBase 要和 EMR 处在同一 VPC 下，建议不开启高可用。
- 将 EMR 所有节点的 IP 加入到云 HBase 白名单。
  - 获取云 HBase 的 Zookeeper 访问地址，可在云 HBase 控制台查看。



- 由于云 HBase 的 HDFS 端口默认是不开的，需要联系工作人员开通。

实施步骤

1. 使用 SSH 方式登录到集群，可参考 SSH 登录集群。
2. 修改 Hive 配置
   - 进入 Hive 配置目录 */etc/ecm/hive-conf/*
   - 修改 *hbase-site.xml*，将 hbase.zookeeper.quorum 的值修改为云 HBase 的 Zookeeper 访问连接:

```
<property>
            <name>hbase.zookeeper.quorum</name>
```

```
            <value>hb-bp183x4tu8x7q****-001.hbase.rds.aliyuncs.com
,hb-bp1mhyea7754b****-002.hbase.rds.aliyuncs.com,hb-bp1mhyea7754b
****-003.hbase.rds.aliyuncs.com</value>
      </property>
```

3. **Hive 中创建云 HBase 表**

   如果 HBase 表不存在，可在 Hive 中直接创建云 HBase 关联表

   a. **输入** `hive` **命令进入 Hive cli 命令行**

   b. **创建 HBase 表:**

```
CREATE TABLE hive_hbase_table(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")
```

```
TBLPROPERTIES ("hbase.table.name" = "hive_hbase_table", "hbase.
mapred.output.outputtable" = "hive_hbase_table");
```

c. Hive 中向 HBase 插入数据

```
insert into hive_hbase_table values(212,'bab');
```

```
hive> insert into hive_hbase_table values(212,'bab');
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
Query ID = root_20181014173030_a0e99198-9aa5-4d29-b011-dc7b36365a20
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1536221485395_0084, Tracking URL = http://emr-header-1.cluster-74778:20888/proxy/application_1536221485395_0084/
Kill Command = /usr/lib/hadoop-current/bin/hadoop job  -kill job_1536221485395_0084
Hadoop job information for Stage-3: number of mappers: 1; number of reducers: 0
2018-10-14 17:30:40,833 Stage-3 map = 0%,  reduce = 0%
2018-10-14 17:30:47,252 Stage-3 map = 100%,  reduce = 0%, Cumulative CPU 3.66 sec
MapReduce Total cumulative CPU time: 3 seconds 660 msec
Ended Job = job_1536221485395_0084
MapReduce Jobs Launched:
Stage-Stage-3: Map: 1   Cumulative CPU: 3.66 sec   HDFS Read: 11867 HDFS Write: 0 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 660 msec
OK
Time taken: 17.385 seconds
```


d. 查看云 HBase 表，HBase 表已创建，数据也已经写入

```
[root@iZbp16ku9i9clejitib6dzZ ~]# hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/apps/t-apsara-hbase-1.4.6.3/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/apps/t-emr-hadoop-2.7.2.2/share/hadoop/common/lib/slf4j-log4j12-1.7.10.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]
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.4.6.3, r89ac288a5add370c07548ec3ce25f6e1f3210d23, Fri Jul  6 14:13:46 CST 2018

hbase(main):001:0> list
TABLE
A_TABLE
BAKE
BASE_TABLE
B_IDX
B_IDX1
B_IDX2
DEFAULT.TEST_IDX
MY_TABLE
PROD_METRICS
SYSTEM.MUTEX
SYSTEM:CATALOG
SYSTEM:FUNCTION
SYSTEM:SEQUENCE
SYSTEM:STATS
hive_hbase_table
tv
18 row(s) in 0.2110 seconds
```



```
hbase(main):004:0* scan 'hive_hbase_table'
ROW                                                    COLUMN+CELL
 212                                                   column=cf1:val, timestamp=1539509446271, value=bab
1 row(s) in 0.0950 seconds
```


e. 在 HBase 中写入数据，并在 Hive 中查看

```
hbase(main):005:0> put 'hive_hbase_table','132','cf1:val','acb'
0 row(s) in 0.0430 seconds
```


在 Hive 中查看：

```
hive> select * from hive_hbase_table;
OK
132     acb
212     bab
Time taken: 0.273 seconds, Fetched: 2 row(s)
```


f. Hive 删除表，HBase 表也一并删除

```
hive>
    >
    >
    > drop table hive_hbase_table;
OK
Time taken: 6.307 seconds
```

查看 HBase 表，报错不存在表：

```
hbase(main):008:0* scan 'hive_hbase_table'
ROW                                                    COLUMN+CELL

ERROR: Unknown table hive_hbase_table!
```


说明：

如果 HBase 表已存在，可在 Hive 中 HBase 外表进行关联，外部表在删除时不影响 HBase 已创建表

a. 云 HBase 中创建 HBase 表，并 put 测试数据：

```
hbase(main):020:0* create 'hbase_table','f'
0 row(s) in 1.3010 seconds

=> Hbase::Table - hbase_table
hbase(main):021:0> put 'hbase_table','1122','f:col1','hello'
0 row(s) in 0.0190 seconds

hbase(main):022:0> put 'hbase_table','1122','f:col2','hbase'
0 row(s) in 0.0110 seconds
```

b. Hive 中创建 HBase 外部关联表，并查看数据

```
hive> create external table hbase_table(key int, col1 string, col2 string)
    > STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
    > WITH SERDEPROPERTIES ("hbase.columns.mapping" = "f:col1,f:col2")
    > TBLPROPERTIES("hbase.table.name" = "hbase_table", "hbase.mapred.output.outputtable" = "hbase_table");
OK
Time taken: 0.129 seconds
hive> select * from hbase_table;
OK
1122    hello   hbase
Time taken: 0.181 seconds, Fetched: 1 row(s)
hive>
```

c. 删除 Hive 表不影响 HBase 已存在的表：

```
hive> drop table hbase_table;
OK
Time taken: 0.102 seconds
hive>
```

```
hbase(main):023:0> scan 'hbase_table'
ROW                                                    COLUMN+CELL
 1122                                                  column=f:col1, timestamp=1539510170256, value=hello
 1122                                                  column=f:col2, timestamp=1539510181752, value=hbase
1 row(s) in 0.0160 seconds
```

总结

Hive 更多操作 HBase 步骤，可参考 HBaseIntegration。如果使用 ECS 自建 MR 集群的Hive 时，操作步骤跟 EMR 操作类似，需要注意的是自建 Hive 的 *hbase-site.xml* 部分配置项可能与云 HBase 不一致，简单来说网络和端口开放后，只保留 `hbase.zookeeper.quorum` 即可与云 HBase 进行关联。

12 使用 E-MapReduce 进行 MySQL Binlog 日志准实时传输

本文介绍如何利用阿里云的 SLS 插件功能和 E-MapReduce 集群进行 MySQL binlog 的准实时传输。

基本架构

RDS -> SLS -> Spark Streaming -> Spark HDFS

上述链路主要包含3个过程：

1. 如何把 RDS 的 binlog 收集到 SLS。
2. 如何通过 Spark Streaming 将 SLS 中的日志读取出来，进行分析。
3. 如何把链路 2 中读取和处理过的日志，保存到 Spark HDFS中。

环境准备

1. 安装一个 MySQL 类型的数据库（使用 MySQL 协议，例如 RDS、DRDS 等），开启 log-bin 功能，且配置 binlog 类型为 ROW 模式（RDS默认开启）。
2. 开通 SLS 服务。

操作步骤

1. 检查 MySQL 数据库环境。

   a. 查看是否开启 log-bin 功能。

   ```
   mysql> show variables like "log_bin";
   +---------------+-------+
   | Variable_name | Value |
   +---------------+-------+
   | log_bin       | ON    |
   +---------------+-------+
   1 row in set (0.02 sec)
   ```

   b. 查看 binlog 类型。

   ```
   mysql> show variables like "binlog_format";
   +---------------+-------+
   | Variable_name | Value |
   +---------------+-------+
   | binlog_format | ROW   |
   +---------------+-------+
   1 row in set (0.03 sec)
   ```

2. 添加用户权限。（也可以直接通过RDS控制台添加）

   ```
   CREATE USER canal IDENTIFIED BY 'canal';
   ```

```
GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'canal
'@'%';
FLUSH PRIVILEGES;
```

3. 为 SLS 服务添加对应的配置文件，并检查数据是否正常采集。

   a. 在 SLS 控制台添加对应的 project 和 logstore，例如：创建一个名称为 canaltest 的 project，然后创建一个名称为 canal 的 logstore。

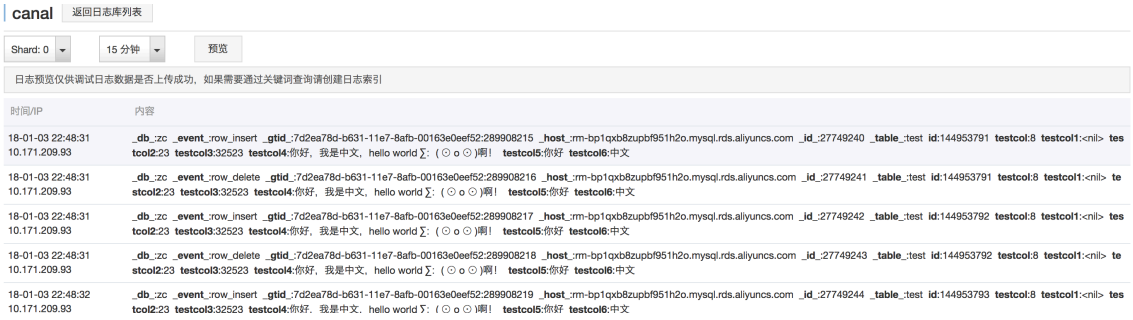
   b. 对 SLS 进行配置：在 */etc/ilogtail* 目录下创建文件user_local_config.json，具体配置如下：

```
{
"metrics": {
   "##1.0##canaltest$plugin-local": {
       "aliuid": "****",
       "enable": true,
       "category": "canal",
       "defaultEndpoint": "*******",
       "project_name": "canaltest",
       "region": "cn-hangzhou",
       "version": 2
       "log_type": "plugin",
       "plugin": {
           "inputs": [
               {
                   "type": "service_canal",
                   "detail": {
                       "Host": "*****",
                       "Password": "****",
                       "ServerID": ****,
                       "User" : "***",
                       "DataBases": [
                           "yourdb"
                       ],
                       "IgnoreTables": [
                           "\\S+_inner"
                       ],
                        "TextToString" : true
                   }
               }
           ],
           "flushers": [
               {
                   "type": "flusher_sls",
                   "detail": {}
               }
           ]
       }
   }
}
```

```
}
```

其中 detail 中的 Host 和 Password 等信息为 MySQL 数据库信息，User 信息为之前授权过的用户名。aliUid、defaultEndpoint、project_name、category 请根据自己的实际情况填写对应的用户和 SLS 信息。

c. 等待约 2 分钟，通过 SLS 控制台查看日志数据是否上传成功，具体如图所示。



如果日志数据没有采集成功，请根据SLS的提示，查看SLS的采集日志进行排查。

4. 准备代码，将代码编译成 jar 包，然后上传到 OSS。

a. 将 EMR 的示例代码通过 git 复制下来，然后进行修改，具体命令为：`git clone https://github.com/aliyun/aliyun-emapreduce-demo.git`。示例代码中已经有 LoghubSample 类，该类主要用于从 SLS 采集数据并打印。以下是修改后的代码，供参考：

```
package com.aliyun.emr.example
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.aliyun.logservice.LoghubUtils
import org.apache.spark.streaming.{Milliseconds, StreamingContext}
object LoghubSample {
def main(args: Array[String]): Unit = {
if (args.length < 7) {
 System.err.println(
   """Usage: bin/spark-submit --class LoghubSample examples-1.0-
SNAPSHOT-shaded.jar
      |
      |
   """.stripMargin)
 System.exit(1)
}
val loghubProject = args(0)
val logStore = args(1)
val loghubGroupName = args(2)
val endpoint = args(3)
val accessKeyId = args(4)
val accessKeySecret = args(5)
val batchInterval = Milliseconds(args(6).toInt * 1000)
val conf = new SparkConf().setAppName("Mysql Sync")
//    conf.setMaster("local[4]");
val ssc = new StreamingContext(conf, batchInterval)
val loghubStream = LoghubUtils.createStream(
 ssc,
 loghubProject,
 logStore,
```

```
 loghubGroupName,
 endpoint,
 1,
 accessKeyId,
 accessKeySecret,
 StorageLevel.MEMORY_AND_DISK)
loghubStream.foreachRDD(rdd =>
   rdd.saveAsTextFile("/mysqlbinlog")
)
ssc.start()
ssc.awaitTermination()
}
}
```

其中的主要改动是：`loghubStream.foreachRDD(rdd => rdd.saveAsObjectFile("/mysqlbinlog") )`**。这样在 EMR 集群中运行时，就会把Spark Streaming 中流出来的数据，保存到 EMR 的 HDFS 中。**

说明:

- **由于如果要在本地运行，请在本地环境提前搭建 Hadoop 集群。**
- **由于 EMR 的 Spark SDK 做了升级，其示例代码比较旧，不能直接在参数中传递 OSS 的 AccessKeyId、AccessKeySecret， 而是需要通过 SparkConf 进行设置，如下所示。**

```
trait RunLocally {
val conf = new SparkConf().setAppName(getAppName).setMaster("
local[4]")
conf.set("spark.hadoop.fs.oss.impl", "com.aliyun.fs.oss.nat.
NativeOssFileSystem")
conf.set("spark.hadoop.mapreduce.job.run-local", "true")
conf.set("spark.hadoop.fs.oss.endpoint", "YourEndpoint")
conf.set("spark.hadoop.fs.oss.accessKeyId", "YourId")
conf.set("spark.hadoop.fs.oss.accessKeySecret", "YourSecret")
conf.set("spark.hadoop.job.runlocal", "true")
conf.set("spark.hadoop.fs.oss.impl", "com.aliyun.fs.oss.nat.
NativeOssFileSystem")
conf.set("spark.hadoop.fs.oss.buffer.dirs", "/mnt/disk1")
val sc = new SparkContext(conf)
def getAppName: String
```

```
}
```

- 在本地调试时，需要把 loghubStream.foreachRDD(rdd => rdd.saveAsObjectFile("/mysqlbinlog")) 中的 /mysqlbinlog 修改成本地 HDFS的地址。

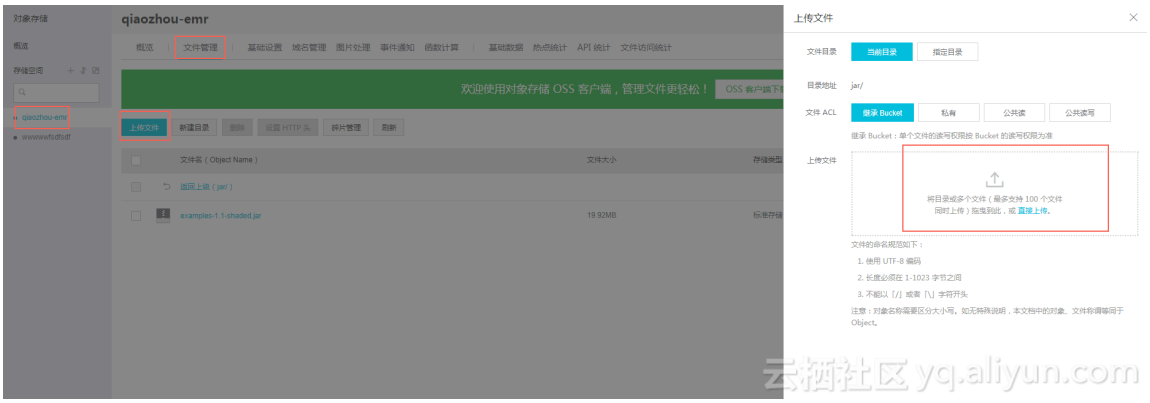
b. 代码编译。

在本地调试完成后，我们可以通过如下命令进行打包编译：

```
mvn clean install
```

c. 上传 jar 包。

请先在 OSS 上建立 bucket 为*qiaozhou-EMR/jar*的目录，然后通过OSS 控制台或 OSS 的 SDK 将*/target/shaded*目录下的*examples-1.1-shaded.jar*上传到 OSS 的这个目录下。上传后的 jar 包地址为*oss://qiaozhou-EMR/jar/examples-1.1-shaded.jar*，这个地址在后面会用上，如下图所示：



5. 搭建 EMR 集群，创建任务并运行执行计划。

a. 通过 EMR 控制台创建一个 EMR 集群，大约需要 10 分钟左右，请耐心等待。

b. 创建一个类型为 Spark 的作业。

请根据您具体的配置将 `SLS_endpoint` `$SLS_access_id` `$SLS_secret_key` 替换成真实值。请注意参数的顺序，否则可能会报错。

```
--master yarn --deploy-mode client --driver-memory 4g --executor
-memory 2g --executor-cores 2 --class com.aliyun.EMR.example
.LoghubSample ossref://EMR-test/jar/examples-1.1-shaded.jar
```

```
canaltest canal sparkstreaming $SLS_endpoint $SLS_access_id $
SLS_secret_key 1
```

c. 创建执行计划，将作业和 EMR 集群绑定后，开始运行。

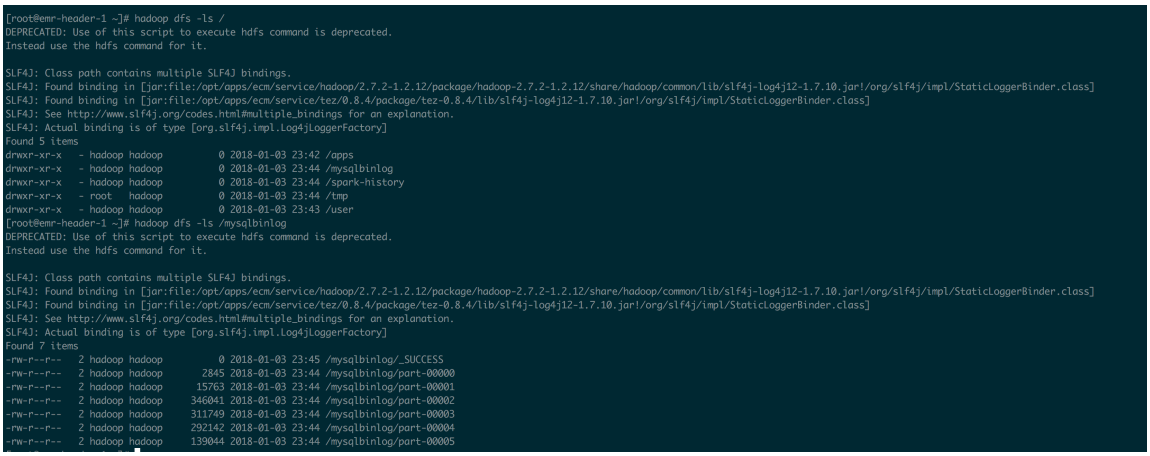
d. 查询 Master 节点的IP，如图所示：

通过 SSH 登录后，执行以下命令：

```
hadoop fs -ls /
```

可以看到 mysqlbinlog 开头的目录，再通过以下命令查看 mysqlbinlog 文件：

```
hadoop fs -ls /mysqlbinlog
```



还可以通过 `hadoop fs -cat /mysqlbinlog/part-00000` 命令查看文件内容。

6. 错误排查。

如果没有看到正常的结果，可以通过 EMR 的运行记录，来进行问题排查，如图所示：

13 Gateway 节点运行 Flume 进行数据同步

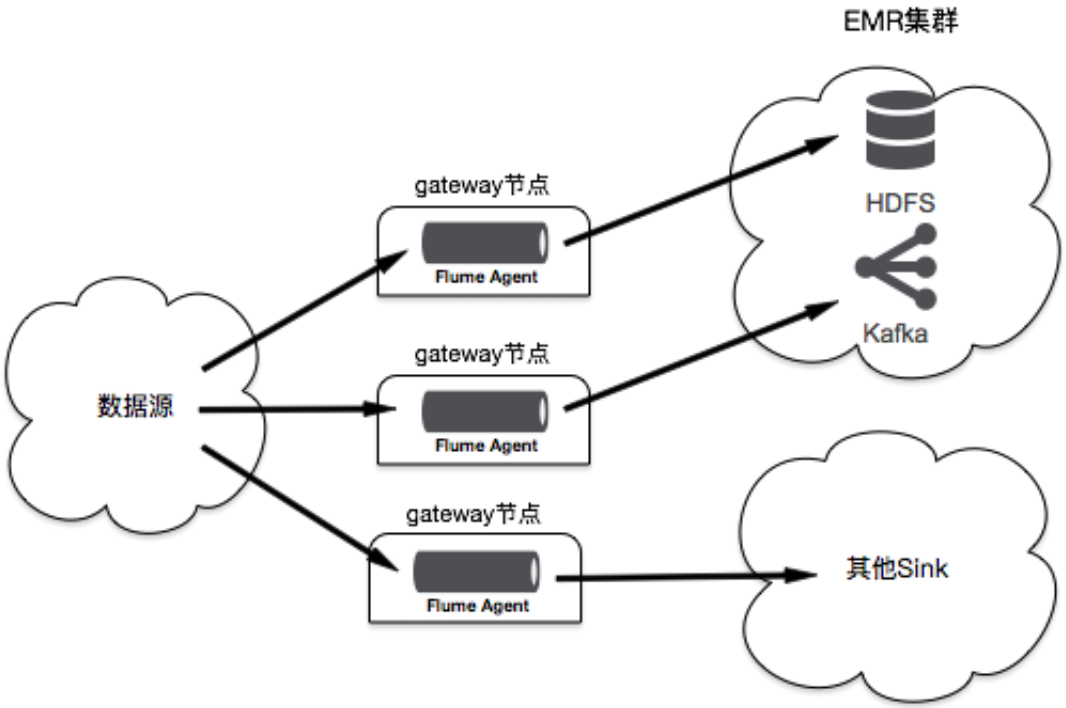
本文以阿里云 E-MapReduce 3.17.0 及以后的版本特性介绍使用 Gateway 节点运行 Flume 从而进行数据同步操作。

背景信息

E-MapReduce从 3.16.0 版本开始支持 Apache Flume，从 3.17.0 版本开始提供默认监控等特性。

- 基本数据流

  在 Gateway 节点运行 Flume 可以避免对 EMR Hadoop 集群产生影响。使用 Gateway节点部署 Flume Agent 的基本数据流如下图所示：



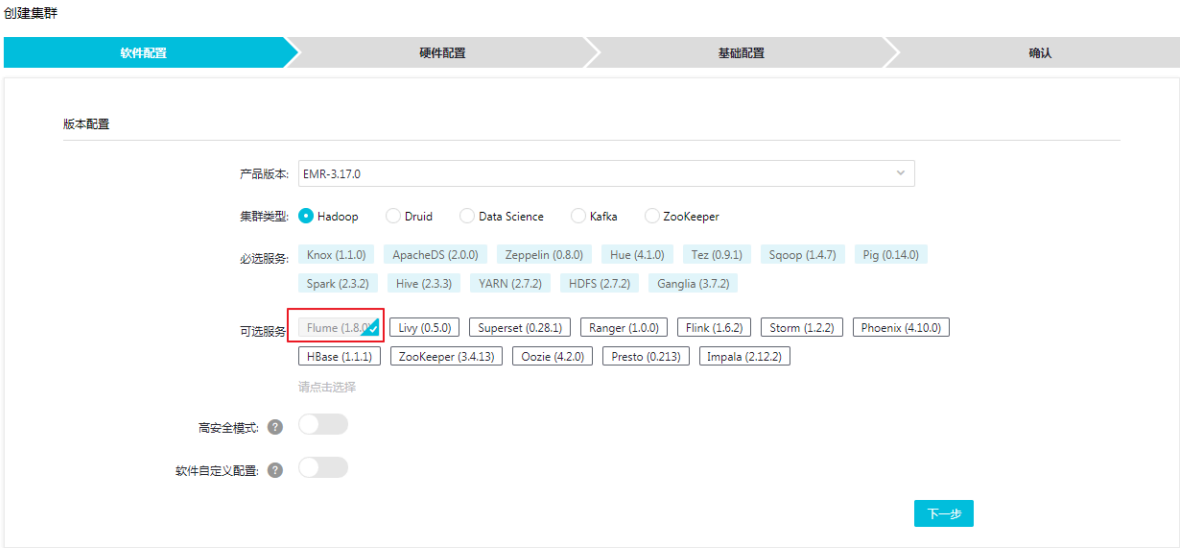
环境准备

本文选择在杭州 Region 进行测试，版本选择 EMR- 3.17.0，本次测试需要的组件版本有：

- Flume：1.8.0

本文使用阿里云 EMR 服务自动化搭建 Hadoop 集群，详细过程请参考创建集群。

- 创建 Hadoop 集群，在可选服务中选择 Flume。



- 创建 Gateway 节点，关联上一步已经创建好的 Hadoop 集群。

实施步骤

- 运行 Flume
  - Flume 的默认配置路径为 */etc/ecm/flume-conf*。参考 #unique_34创建 agent 配置文件 *flume.properties*后，可以使用以下命令运行一个Flume Agent：

    ```
    nohup flume-ng agent -n a1 -f flume.properties &
    ```

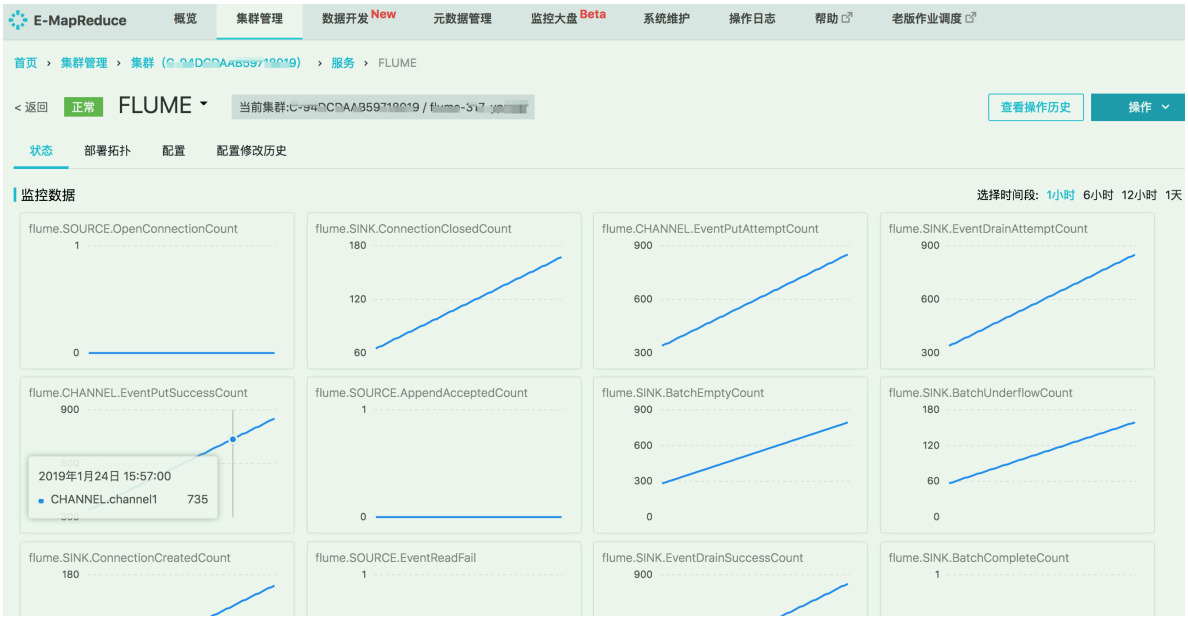
  - 如果想使用自定义的配置目录，可以使用 `-c` 或者 `--conf` 覆盖默认的配置，例如：

    ```
    nohup flume-ng agent -n a1 -f flume.properties -c path-to-flume-conf &
    ```

> **注意：**
>
> 参考 #unique_34在 Gateway 节点配置 Flume 时，如果 Sink 到 HBase，需要在配置文件 *flume.properties* 添加配置项 zookeeperQuorum，例如：
>
> ```
> a1.sinks.k1.zookeeperQuorum=emr-header-1.cluster-46349:2181
> ```
>
> 其中，Zookeeper的主机地址`emr-header-1.cluster-46349`可以在*/etc/ecm/hbase-conf/hbase-site.xml*的 hbase.zookeeper.quorum 配置项中找到。

- 查看监控信息

  默认情况下，集群的 Console 页面提供了 Flume agent 的监控信息。通过在集群与服务管理页面单击 Flume 服务进行访问，如下图所示：



  注意:

  监控数据以 agent 组件(source、channel 或 sink)的名称命名，例如 CHANNEL.channel1 表示名称为 channel1 的 channel 组件的监控指标，所以在配置不同的 agent 时请避免使用相同的组件名称。

  如果想通过 Ganglia 等方式查看 Flume Agent 的监控数据，可参考 Flume 官网进行配置。此时 Console 页面将不会显示 Flume agent 的监控数据。

- 查看日志

  默认情况下，Flume agent 日志的存放路径为 `/mnt/disk1/log/flume/${flume-agent-name}/flume.log`。可以通过修改 `/etc/ecm/flume-conf/log4j.properties` 进行配置(不建议修改日志路径)。

  注意:

  日志路径包含了 Flume agent 的名称，所以配置不同的 agent 时请勿使用相同的 agent 名称，以免日志混在一起。

14 OSS 数据权限隔离

本文将介绍如何使用RAM访问控制对不同子账号的OSS数据进行隔离。

前提条件

已获取云账号。

背景信息

E-MapReduce支持使用RAM来隔离不同子账号的数据。

步骤一 登录RAM控制台

1. 云账号登录RAM控制台。

步骤二 创建RAM用户

1. 在左侧导航栏的人员管理菜单下，单击用户。
2. 单击新建用户。

   > 说明:
   > 单击添加用户，可一次性创建多个RAM用户。

3. 输入登录名称和显示名称。
4. 在访问方式区域下，选择控制台密码登录或编程访问。

   > 说明:
   > 为了保障账号安全，建议仅为RAM用户选择一种登录方式，避免RAM用户离开组织后仍可以通过访问密钥访问阿里云资源。

5. 单击确认。

步骤三 新建权限策略

RAM除了提供常用的默认权限策略外，还支持让您自定义权限策略，使得授权更加灵活。根据不同需要，您可创建多个权限策略。

1. 在左侧导航栏的权限管理菜单下，单击权限策略管理。
2. 单击新建权限策略。
3. 填写策略名称和备注。

4. 配置模式选择脚本配置。

   脚本配置方法请参见语法结构编辑策略内容。 本例分别按照以下两个脚本示例来创建两个权限策略：

测试环境（test-bucket）	生产环境（prod-bucket）

<pre>{
"Version": "1",
"Statement": [
{
"Effect": "Allow",
"Action": [
"oss:ListBuckets"
],
"Resource": [
"acs:oss:*:*:*"
]
},
{
"Effect": "Allow",
"Action": [
"oss:Listobjects",
"oss:GetObject",
"oss:PutObject",
"oss:DeleteObject"
],
"Resource": [
"acs:oss:*:*:test-bucket",
"acs:oss:*:*:test-bucket/*"
]
}
]
}</pre>
	<pre>{
	"Version": "1",
	"Statement": [
	{
	"Effect": "Allow",
	"Action": [
	"oss:ListBuckets"
	],
	"Resource": [
	"acs:oss:*:*:*"
	]
	},
	{
	"Effect": "Allow",
	"Action": [
	"oss:Listobjects",
	"oss:GetObject",
	"oss:PutObject"
	],
	"Resource": [
	"acs:oss:*:*:prod-bucket",
	"acs:oss:*:*:prod-bucket/*"
	]
	}
	]
	}</pre>

   按上述脚本示例进行权限隔离后，RAM用户在E-MapReduce控制台的限制如下：

   - 在创建集群、创建作业和创建执行计划的 OSS 选择界面，可以看到所有的 bucket，但是只能进入被授权的 bucket。
   - 只能看到被授权的 bucket 下的内容，无法看到其他 bucket 内的内容。
   - 作业中只能读写被授权的 bucket，读写未被授权的 bucket 会报错。

5. 单击确认。

步骤四 为RAM用户授权

1. 在左侧导航栏的人员管理菜单下，单击用户。
2. 在用户登录名称/显示名称列表下，找到目标RAM用户。
3. 单击添加权限，被授权主体会自动填入。

4. 在左侧权限策略名称列表下，单击需要授予RAM用户的权限策略。

   > **说明:**
   > 在右侧区域框，选择某条策略并单击×，可撤销该策略。

5. 单击确定。
6. 单击完成。

步骤五 开启RAM用户的控制台登录权限（可选）

如果创建RAM用户时未开启控制台登录权限，则您可按以下方法进行开启。

1. 在左侧导航栏的人员管理菜单下，单击用户。
2. 在用户登录名称/显示名称列表下，单击目标RAM用户名称。
3. 在认证管理页签下的控制台登录管理区域，单击修改登录设置。
4. 控制台密码登录选择开启。
5. 单击确认。

步骤六 通过RAM用户登录E-MapReduce控制台

1. RAM用户登录控制台。
2. 登录控制台后，选择E-MapReduce产品即可。

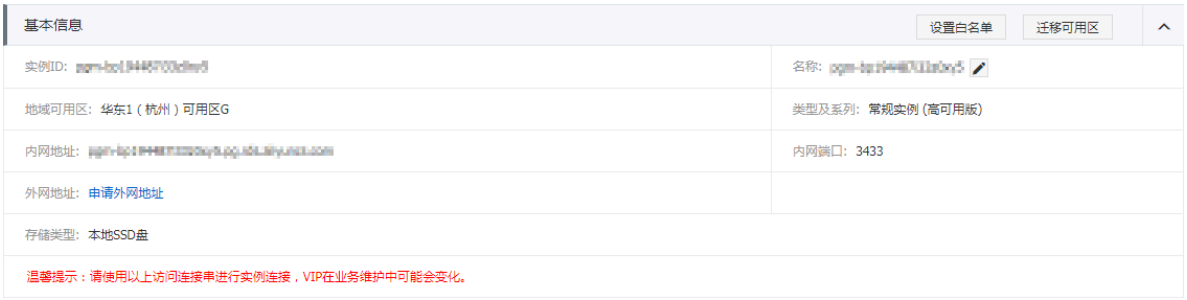
15 在 E-MapReduce 上使用 Sqoop 工具与数据库同步数据进行网络配置

如果您的 E-MapReduce（EMR） 集群需要和集群之外的数据库同步数据，需要确保网络是联通的。本文以 RDS，ECS 自建，云下私有数据库三种情况，分别介绍如何进行网络配置。

云数据库 RDS

- 经典网络 RDS

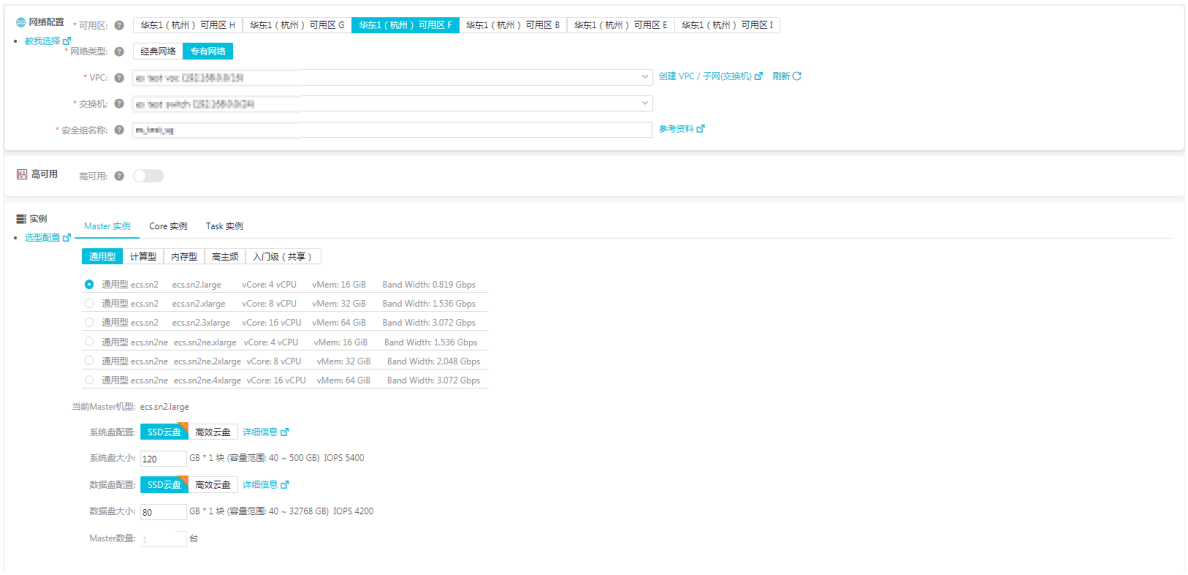
  想要访问经典网络RDS，EMR 最好也指定用经典网络。经典网络的 RDS 可以设置内网地址和外网地址。由于经典网络 EMR 集群只有 Master 节点可以访问公网，并且 Sqoop 是用 map 任务同步数据可能在任意节点上运行，所以 Sqoop 任务需要配置连接 RDS 的内网地址来连接。另外，需要确保 EMR 集群的内网 IP 在 RDS 白名单里。



  创建 EMR 集群并指定经典网络类型，具体步骤请参见#unique_38。

- VPC 网络 RDS

  如果 RDS 在 VPC 网络下，EMR 集群也需要指定用 VPC 网络。推荐 EMR 集群和 RDS 在同一个 VPC 网络内，这样可以直接访问 RDS 地址。如果在不同的 VPC 网络下，需要通过高速通道打通网络连接。



ECS 自建数据库

- 经典网络

  访问经典网络的 ECS 自建数据库与经典网络的 RDS 类似，也需要 EMR 集群指定使用经典网络，访问自建数据库的内网地址。区别是需要将数据库所在的 ECS 实例和 EMR 集群的实例放在同一个安全组内。您可以在 ECS 控制台安全组 > 管理实例 > 添加实例

- VPC 网络

  访问 VPC 网络的自建数据库跟 VPC 网络的 RDS 类似，EMR 集群指定使用 VPC 网络。额外要做的是将数据库 ECS 实例和 EMR 集群实例放到同一个安全组内。

云下私有数据库

有两种方式访问云下私有数据库，一种是绑定弹性IP（EIP）访问数据库的公网地址，一种是将云下数据库通过高速通道和 VPC 网络互联。

- 绑定 EIP

  如果云下私有数据库可以通过公网访问，推荐 EMR 集群使用 VPC 网络。创建一个 VPC 网络类型的 EMR 集群，创建成功后在 ECS 控制台管理 > 配置信息 > 更多 > 绑定弹性 IP给集群的每个 ECS 实例绑定一个 EIP，就可以访问私有数据库的公网地址了。

- **高速通道**

  如果私有数据库不能在公网暴露，可以创建一个 VPC 网络类型的 EMR 集群，通过高速通道连接私有 IDC 和阿里云上的 VPC 集群。高速通道详情请参见高速通道产品文档

16 通过Spark Streaming作业处理Kafka数据

本节介绍如何使用阿里云E-MapReduce部署Hadoop集群和Kafka集群，并运行Spark Streaming作业消费Kafka数据。

前提条件

- 已注册阿里云账号，详情请参见注册云账号。
- 已开通E-MapReduce服务。
- 已完成云账号的授权，详情请参见#unique_23。

背景信息

在开发过程中，通常会遇到消费Kafka数据的场景。在阿里云E-MapReduce中，您可通过运行Spark Streaming作业来消费Kafka数据。

步骤一 创建Hadoop集群和Kafka集群

推荐您将Hadoop集群和Kafka集群创建在同一个安全组下。如果Hadoop集群和Kafka集群不在同一个安全组下，则两者的网络默认是不互通的，您需要对两者的安全组分别进行相关配置，以使两者的网络互通。

1. 登录阿里云 E-MapReduce 控制台。
2. 创建Hadoop集群，详情请参见创建集群。

3. 创建Kafka集群，详情请参见创建集群。

步骤二 获取JAR包并上传到Hadoop集群

本例中的JAR包：对E-MapReduce的Demo进行了一定的修改后，编译生成的JAR包。JAR包需要上传到Hadoop集群的emr-header-1主机中。

1. 获取JAR包（本例JAR下载地址）。
2. 返回到阿里云 E-MapReduce 控制台。
3. 在集群管理页面，单击Hadoop集群的集群ID，进入Hadoop集群。
4. 在左侧导航树中选择主机列表，然后在右侧查看Hadoop集群中emr-header-1主机的IP信息。
5. 通过SSH客户端登录emr-header-1主机。
6. 上传JAR包到emr-header-1主机的某个目录。

> **说明:**
> 后续步骤中的代码有涉及到此路径，本例上传路径为*/home/hadoop*。上传JAR包，请保留该登录窗口，后续步骤仍将使用。

步骤三 在Kafka集群上创建Topic

您可直接在E-MapReduce上以可视化的方式来创建Topic（详情请参见#unique_40），也可登录Kafka集群的emr-header-1主机后以命令行的方式来创建Topic。本例以命令行方式创建一个分区数为10、副本数为2、名称为test的Topic。

1. 返回到阿里云 E-MapReduce 控制台。
2. 在集群管理页面，单击Kafka集群的集群ID，进入Kafka集群。
3. 在左侧导航树中选择主机列表，然后在右侧查看Kafka集群中emr-header-1主机的IP信息。

4. 在SSH客户端中新建一个命令窗口，登录Kafka集群的emr-header-1主机。
5. 通过以下命令创建Topic。

```
/usr/lib/kafka-current/bin/kafka-topics.sh --partitions 10 --
replication-factor 2 --zookeeper emr-header-1:2181 /kafka-1.0.0 --
topic test --create
```

说明:

创建Topic后，请保留该登录窗口，后续步骤仍将使用。

步骤四 运行Spark Streaming作业

完成上述操作后，您即可在Hadoop集群上运行Spark Streaming作业。本例将运行一个作业进行流式单词统计（WordCount）。

1. 返回到Hadoop集群的emr-header-1主机登录窗口。

   如果误关闭了此窗口，请重新登录，详情请参见步骤二 获取JAR包并上传到Hadoop集群中的相关步骤。

2. 通过如下作业命令来进行流式单词统计（WordCount）。

```
spark-submit --class com.aliyun.emr.example.spark.streaming.
KafkaSample  /home/hadoop/examples-1.2.0-shaded-2.jar 192.168.xxx.
xxx:9092 test 5
```

命令中JAR包后面的三个关键参数说明如下：

- 192.168.xxx.xxx：Kafka集群中任一Kafka Broker组件的内网或外网IP地址，示例如图16-1: Kafka集群组件所示。
- test：Topic名称。
- 5：时间间隔。

图 16-1: Kafka集群组件

步骤五 使用Kafka发布消息

进行本步骤操作时，需要保持Spark Streaming作业一直处于运行状态。运行Kafka的生产者（producer）后，在Kafka客户端的命令行中输入文本时，在Hadoop集群客户端的命令行中会实时显示单词统计结果。

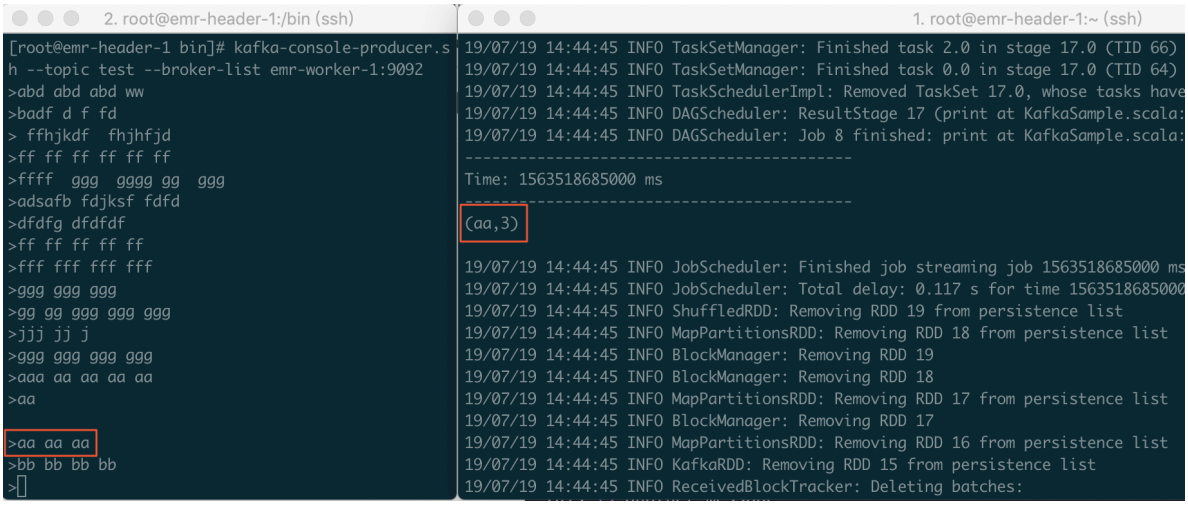
1. 返回到Kafka集群的emr-header-1主机登录窗口。

   如果误关闭了此窗口，请重新登录，详情请参见步骤三 在Kafka集群上创建Topic中的相关步骤。

2. 在Kafka集群的登录窗口中，通过如下命令来运行生产者（producer）。

```
/bin/kafka-console-producer.sh --topic test --broker-list emr-worker
-1:9092
```

3. 在Kafka登录窗口的命令行中不断输入文本，则在Hadoop集群登录窗口中实时显示文本的统计信息。

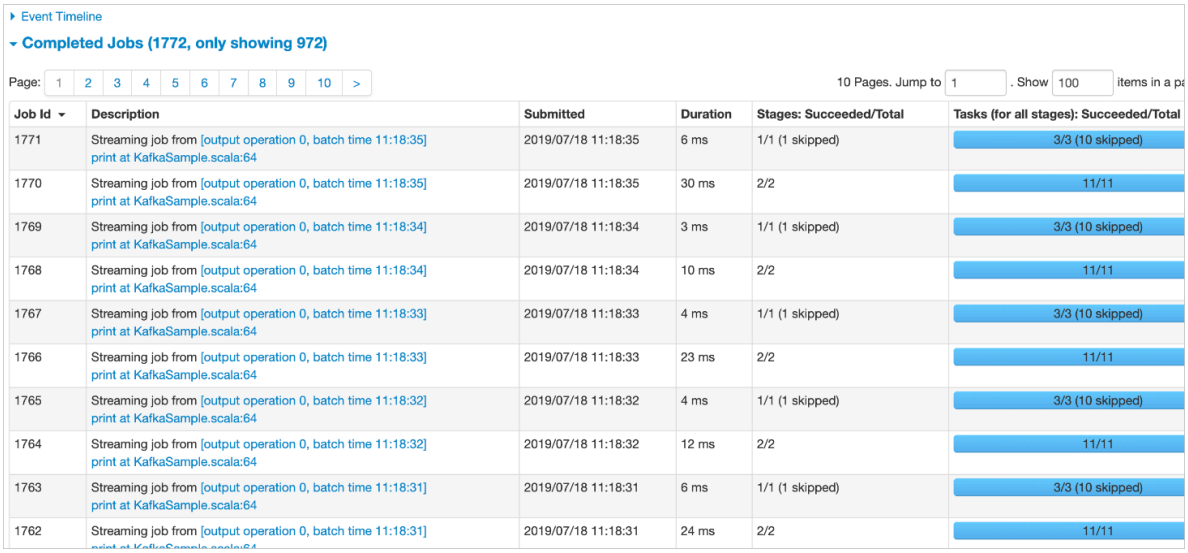


步骤六 查看Spark Streaming作业的进展

Spark Streaming作业开始运行后，您可在E-MapReduce上查看作业的状态。

1. 返回到阿里云 E-MapReduce 控制台。

2. 在Hadoop集群的访问链接与端口页面中，单击Spark History Server UI后的链接，查看Spark Streaming作业的状态。详情请参见访问链接与端口。



17 通过Kafka Connect进行数据迁移

在流式数据处理过程中，E-MapReduce经常需要在Kafka与其他系统间进行数据同步或者在Kafka集群间进行数据迁移。本节向您介绍如何在E-MapReduce上通过Kafka Connect快速的实现Kafka集群间的数据同步或者数据迁移。

前提条件

- 已注册云账号，详情请参见注册云账号。
- 已开通E-MapReduce服务。
- 已完成云账号的授权，详情请参见#unique_23。

背景信息

Kafka Connect是一种可扩展的、可靠的，用于在Kafka和其他系统之间快速的进行流式数据传输的工具。例如，Kafka Connect可以获取数据库的binlog数据，将数据库数据同步至Kafka集群，从而达到迁移数据库数据的目的。由于Kafka集群可对接流式处理系统，所以还可以间接实现数据库对接下游流式处理系统的目的。同时，Kafka Connect还提供了REST API接口，方便您创建和管理Kafka Connect。

Kakfa Connect分为standalone和distributed两种运行模式。在standalone模式下，所有的worker都在一个进程中运行。相比于standalone模式，distributed模式更具扩展性和容错性，是最常用的方式，也是生产环境推荐使用的模式。

本文介绍如何在E-MapReduce上使用Kafka Connect的REST API接口在Kafka集群间进行数据迁移，Kakfa Connect使用distributed模式。

步骤一 创建Kafka集群

在EMR上创建源Kafka集群和目的Kafka集群。Kafka Connect安装在Task节点上，所以目的Kafka集群必须创建Task节点。集群创建好后，Task节点上Kafka Connect服务会默认启动，端口号为8083。

推荐您将源Kafka集群和目的Kafak集群创建在同一个安全组下。如果源Kafka集群和目的Kafak集群不在同一个安全组下，则两者的网络默认是不互通的，您需要对两者的安全组分别进行相关配置，以使两者的网络互通。

1. 登录阿里云 E-MapReduce 控制台。
2. 创建源Kafka集群和目的Kafka集群，详情请参见#unique_38。

说明:

创建目的Kafka集群时，必须开启Task实例，即创建Task节点。

步骤二 准备待迁移数据Topic

在源Kafka集群上创建一个名称为connect的Topic。

1. 以SSH方式登录到源Kafka集群的header节点（本例为emr-header-1）。
2. 以root用户运行如下命令创建一个名称为connect的Topic。

```
kafka-topics.sh --create --zookeeper emr-header-1:2181 --replication
-factor 2 --partitions 10 --topic connect
```

```
[root@emr-header-1 ~]# kafka-topics.sh --create --zookeeper emr-header-1:2181 --replication-factor 2 --partitions 10 --topic connect
Created topic "connect".
[root@emr-header-1 ~]#
```

说明:

完成上述操作后，请保留该登录窗口，后续仍将使用。

步骤三 创建Kafka Connect的connector

在目的Kafka集群的Task节点上，使用curl命令通过JSON数据创建一个Kafka Connect的connector。

1. 以SSH方式登录到目的Kafka集群的Task节点（本节为emr-worker-3）。

2. （可选）自定义Kafka Connect配置。

   进入目的Kafka集群Kafka服务的配置页面，在connect-distributed.properties中自定义offset.storage.topic、config.storage.topic和status.storage.topic三个配置项，详情请参见#unique_42。

   Kafka Connect会将offsets、configs和任务状态保存在Topic中，Topic名对应offset.storage.topic、config.storage.topic和status.storage.topic三个配置项。Kafka Connect会自动使用默认的partition和replication factor创建这三个Topic，其中partition和repication factor配置项保存在*/etc/ecm/kafka-conf/connect-distributed.properties*文件中。

3. 以root用户运行如下命令创建一个Kafka Connect。

```
curl -X POST -H "Content-Type: application/json" --data '{"name": "
connect-test", "config": { "connector.class": "EMRReplicatorSourceC
onnector", "key.converter": "org.apache.kafka.connect.converters.
ByteArrayConverter", "value.converter": "org.apache.kafka.connect.
converters.ByteArrayConverter", "src.kafka.bootstrap.servers": "${
src-kafka-ip}:9092", "src.zookeeper.connect": "${src-kafka-curator-
ip}:2181", "dest.zookeeper.connect": "${dest-kakfa-curator-ip}:2181
", "topic.whitelist": "${source-topic}", "topic.rename.format": "${
dest-topic}", "src.kafka.max.poll.records": "300" } }' http://emr-
worker-3:8083/connectors
```

   在JSON数据中，`name`字段代表创建的Kafka Connect的名称，本例为`connect-test`；`config`字段需要根据实际情况进行配置，关键变量的说明如下：

变量	说明
${source-topic}	源Kafka集群中需要迁移的Topic，多个Topic需用英文逗号（,）隔开，例如connect。
${dest-topic}	目的Kafka集群中迁移后的Topic，例如connect.replica。
${src-kafka-curator-hostname}	源Kafka集群中安装了ZooKeeper服务的节点的内网IP地址。
${dest-kakfa-curator-hostname}	目的Kafka集群中安装了ZooKeeper服务的节点的内网IP地址。

    说明：
   完成上述操作后，请保留该登录窗口，后续仍将使用。

步骤四 查看Kafka Connect和Task节点状态

查看Kafka Connect和Task节点信息，确保两者的状态正常。

1. 返回到目的Kafka集群的Task节点（本节为emr-worker-3）的登录窗口。

2. 以root用户运行如下命令查看所有的Kafka Connect。

```
curl emr-worker-3:8083/connectors
```

```
[root@emr-worker-3 ~]# curl emr-worker-3:8083/connectors
["connect-test"][root@emr-worker-3 ~]#
```

3. 以root用户运行如下命令查看本例创建的Kafka Connect（本例为connect-test）的状态。

```
curl emr-worker-3:8083/connectors/connect-test/status
```

```
[root@emr-worker-3 ~]# curl emr-worker-3:8083/connectors/connect-test/status
{"name":"connect-test","connector":{"state":"RUNNING","worker_id":"192.168.  :8083"},"tasks":[{"state":"RUNNING","id":0,"worker_id":"192.168.  :8083"}],"type":"source"}[r
oot@emr-worker-3 ~]#
```

确保Kafka Connect（本例为connect-test）的状态为RUNNING。

4. 以root用户运行如下命令查看Task节点信息。

```
curl emr-worker-3:8083/connectors/connect-test/tasks
```

```
[root@emr-worker-3 ~]# curl emr-worker-3:8083/connectors/connect-test/status
{"name":"connect-test","connector":{"state":"RUNNING","worker_id":"192.168.  :8083"},"tasks":[{"state":"RUNNING","id":0,"worker_id":"192.168.  :8083"}],"type":"source"}[r
oot@emr-worker-3 ~]# curl emr-worker-3:8083/connectors/connect-test/tasks
[{"id":{"connector":"connect-test","task":0},"config":{"connector.class":"EMRReplicatorSourceConnector","src.zookeeper.connect":"emr-header-1.cluster-127390:2181","topic.rename.f
ormat":"connect.replica","dest.zookeeper.connect":"emr-header-1.cluster-127499:2181","task.class":"org.apache.kafka.connect.replicator.EMRReplicatorSourceTask","src.kafka.max.poll
.records":"300","name":"connect-test","task.id":"connect-test-0","value.converter":"org.apache.kafka.connect.converters.ByteArrayConverter","key.converter":"org.apache.kafka.con
nect.converters.ByteArrayConverter","src.kafka.bootstrap.servers":"192.168.  :9092","topic.whitelist":"connect","partition.assignment":"AAAAAAABAAdjb25uZWN0AAAACgAAAAAAAAABAA
AAAgAAAAMAAAAEAAAABQAAAAYAAAAHAAAACAAAAAkAAAAA"}}][root@emr-worker-3 ~]#
```

确保Task节点的返回信息中无错误信息。

步骤五 生成待迁移数据

通过命令向源集群中的connect Topic发送待迁移的数据。

1. 返回到源Kafka集群的header节点（本例为emr-header-1）的登录窗口。
2. 以root用户运行如下命令向connect Topic发送数据。

```
kafka-producer-perf-test.sh --topic connect --num-records 100000
 --throughput 5000 --record-size 1000 --producer-props bootstrap.
servers=emr-header-1:9092
```

```
[root@emr-header-1 ~]# kafka-producer-perf-test.sh --topic connect --num-records 100000 --throughput 5000 --record-size 1000  --producer-props bootstrap.servers=emr-header-1:909
2
24992 records sent, 4997.4 records/sec (4.77 MB/sec), 4.5 ms avg latency, 149.0 max latency.
25025 records sent, 5005.0 records/sec (4.77 MB/sec), 0.8 ms avg latency, 25.0 max latency.
25000 records sent, 5000.0 records/sec (4.77 MB/sec), 0.7 ms avg latency, 22.0 max latency.
24972 records sent, 4884.0 records/sec (4.66 MB/sec), 0.8 ms avg latency, 122.0 max latency.
100000 records sent, 4901.960784 records/sec (4.67 MB/sec), 1.73 ms avg latency, 393.00 ms max latency, 1 ms 50th, 4 ms 95th, 29 ms 99th, 77 ms 99.9th.
[root@emr-header-1 ~]#
```

步骤六 查看数据迁移结果

生成待迁移数据后，Kafka Connect会自动将这些数据迁移到目的集群的相应文件（本例为connect.replica）中。

1. 返回到目的Kafka集群的Task节点（本节为emr-worker-3）的登录窗口。

2. 以root用户运行如下命令查看数据迁移是否成功。

```
kafka-consumer-perf-test.sh --topic connect.replica --broker-list
emr-header-1:9092 --messages 100000
```

```
[root@emr-worker-3 ~]# kafka-consumer-perf-test.sh  --topic connect.replica --broker-list emr-header-1:9092 --messages 1000000
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.nMsg.sec
2019-07-22 10:13:17:855, 2019-07-22 10:13:32:055, 95.3674, 6.7160, 100000, 7042.2535, 3019, 11181, 8.5294, 8943.7439
[root@emr-worker-3 ~]#
```

从上述返回结果可以看出，在源Kafka集群发送的100000条数据已经迁移到了目的Kakfa集群。

小结

本文介绍并演示了使用Kafka Connect在Kafka集群间进行数据迁移的方法。如果需要了解Kafka Connect更详细的使用方法，请参见Kafka官网资料和REST API。

18 E-MapReduce弹性低成本离线大数据分析

大数据是一项涉及不同业务和技术领域的技术和工具的集合，海量离线数据分析可以应用于多种商业系统环境，例如，电商海量日志分析、用户行为画像分析、科研行业的海量离线计算分析任务等场景。

离线大数据分析概述

主流的三大分布式计算框架系统分别为Hadoop、Spark和Storm：

- Hadoop可以运用在很多商业应用系统，可以轻松集成结构化、半结构化以及非结构化数据集。
- Spark采用了内存计算，允许数据载入内存作反复查询，融合数据仓库、流处理和图形计算等多种计算范式，能够与Hadoop很好地结合。
- Storm适用于处理高速、大型数据流的分布式实时计算，为Hadoop添加可靠的实时数据处理能力。

海量离线数据分析可以应用于多种场景，例如：

- 商业系统环境：电商海量日志分析、用户行为画像分析。
- 科研行业：海量离线计算分析和数据查询。
- 游戏行业：游戏日志分析、用户行为分析。
- 商业用户：数据仓库解决方案的BI分析、多维分析报表。
- 大型企业：海量IT运维日志分析。

最佳实践概述

本实践结合阿里云E-MapReduce以及日志服务LOG、对象存储OSS、抢占式ECS实例、弹性伸缩等产品，以在电商网站进行日志埋点、采集、存储和投递并使用E-MapReduce进行日志消费分析为例，来展示如何实现离线海量日志的大数据分析。详情请参见E-MapReduce弹性低成本离线大数据分析最佳实践。

19 E-MapReduce本地盘实例大规模数据集测试

本文介绍如何使用阿里云E-MapReduce搭建本地盘机型集群节点，并进行大数据基准性能测试。

应用范围

- 需要使用阿里云E-MapReduce+本地盘进行大数据业务前进行性能测试的用户。
- 需要将线下自建大数据集群迁移到阿里云云上E-MapReduce+本地盘进行大数据分析和性能对比测试的用户。

最佳实践概述

为了满足大数据场景下的存储需求，阿里云在云上推出了本地盘D1机型。本地盘D1机型使用本地盘而非云盘作为存储，解决了之前使用云盘的多份冗余数据导致的高成本问题。同时，在使用本地盘D1机型时，数据的传输不需要全部通过网络，因此该场景提供了与磁盘相同的吞吐能力，可发挥Hadoop就近计算的优势。

阿里云E-MapReduce产品针对本地盘机型，推出了一整套的自动化运维方案，帮助阿里云用户方便可靠地使用本地盘机型。该运维方案即能让用户无须关心整个运维过程，又能保证数据高可靠和服务高可用。

大数据基准测试用于公平、客观评测不同大数据产品/平台的功能和性能，对用户选择合适的大数据平台产品具有重要的参考价值，TPC-DS逐渐成为了业界公认的大数据系统测试基准。

本文以阿里云E-MapReduce+D1本地盘方案模拟TPC-DS测试的演示方案，来展示如何使用阿里云大数据集群进行性能测试。详情请参见E-MapReduce本地盘实例大规模数据集测试最佳实践。