E-MapReduce

开发指南

开发指南

开发准备

您已经开通了阿里云服务，并创建了 Access Key ID 和 Access Key Secret。文中的 ID指的是 Access Key ID，KEY 指的是 Access Key Secret。如果您还没有开通或者还不了解OSS，请登录 OSS 产品主页获取更多的帮助。

您已经对 Spark、Hadoop、Hive 和 Pig 具备一定的认识。文中不对 Spark、Hadoop、Hive 和 Pig 开发实践做额外的介绍。更多的开发文档资料可以到 apache 官网获取。

您已经对 Java 和 Scala 语法具备一定的认识，文中的某些例子以 Scala 语言为例。

您已经对阿里云E-MapReduce开发组件有一定了解。欢迎开源爱好者向本项目贡献代码，包括且不局限于：反馈问题，修复BUG，增加新组件等等。

OSS URI

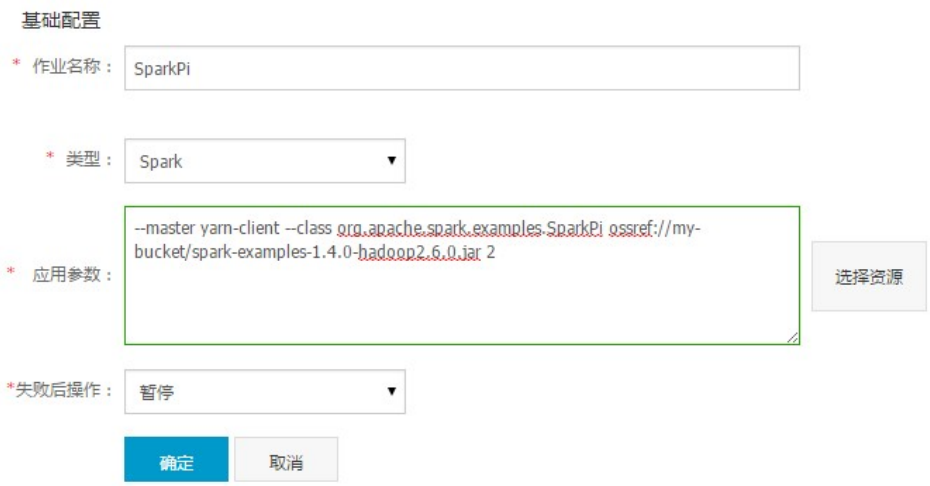
在使用 E-MapReduce 时，用户将会使用两种 OSS URI，分别是：

native URI： oss://[accessKeyId:accessKeySecret@]bucket[.endpoint]/object/path

用户在作业中指定输入输出数据源时使用这种 URI，可以类比 hdfs://。用户操作 OSS 数据时，可以将 accessKeyId，accessKeySecret 以及 endpoint 配置到 Configuration 中，也可以在 URI 中直接指定 accessKeyId，accessKeySecret 以及 endpoint。

ref URI: ossref://bucket/object/path

只在 E-MapReduce 作业配置时有效，用来指定作业运行需要的资源。例如以下作业配置示例：



我们把 oss 与 ossref 这样的前缀称为 scheme。在使用过程中，需要特别注意 URI 中 scheme 的不同。

注意事项

在支持向 OSS 写数据时，E-MapReduce 使用 OSS 的 multipart 分片上传方式。这里需要提醒的是，当作业异常中断后，OSS 中会残留作业已经生产的部分数据，需要您手动删掉。这里的行为和作业输出到 HDFS 是一致的，作业异常中断后，HDFS 也会残留数据，也需要手动删掉。但有一个区别，OSS 对使用 multipart 上传的文件，它是先放在碎片管理中，所以您不仅要删除 OSS 文件管理中的输出目录残留文件，还需要在 OSS 的碎片管理中清理一次，否则会产生数据存储费用。

示例项目

本项目是一个完整的可编译可运行的项目，包括 MapReduce、Pig、Hive 和 Spark 示例代码。请查看开源项目，详情如下：

MapReduce

- WordCount： 单词统计。

Hive

- sample.hive：表的简单查询。

Pig

- sample.pig：Pig 处理 OSS 数据实例。

Spark

SparkPi: 计算 Pi。

SparkWordCount： 单词统计。

LinearRegression： 线性回归。

OSSSample： OSS 使用示例。

ONSSample： ONS 使用示例。

ODPSSample： ODPS 使用示例。

MNSSample：MNS 使用示例。

LoghubSample：Loghub 使用示例。

依赖资源

测试数据（data 目录下）：

The_Sorrows_of_Young_Werther.txt：可作为 WordCount（MapReduce/Spark）的输入数据。

patterns.txt：WordCount（MapReduce）作业的过滤字符。

u.data：sample.hive 脚本的测试表数据。

abalone：线性回归算法测试数据。

依赖jar包（lib目录下）

- tutorial.jar：sample.pig作业需要的依赖jar包

准备工作

本项目提供了一些测试数据，您可以简单地将其上传到 OSS 中即可使用。其他示例，例如ODPS、MNS、ONS 和 LogService 等等，需要您自己准备数据如下：

【可选】 创建 LogService，参考日志服务用户指南。

【可选】 创建 ODPS 项目和表，参考ODPS快速开始。

【可选】 创建 ONS，参考消息队列快速开始。

【可选】 创建 MNS，参考消息服务控制台使用帮助。

基本概念

OSSURI：oss://accessKeyId:accessKeySecret@bucket.endpoint/a/b/c.txt，用户在作业中指定输入输出数据源时使用，可以类比 hdfs://。

阿里云 AccessKeyId/AccessKeySecret 是您访问阿里云 API 的密钥，您可以在这里获取。

集群运行

Spark

SparkWordCount： spark-submit --class SparkWordCount examples-1.0-SNAPSHOT-shaded.jar <inputPath> <outputPath> <numPartition>

参数说明如下：

inputPath： 输入数据路径。

outputPath： 输出路径。

numPartition： 输入数据 RDD 分片数目。

SparkPi： spark-submit --class SparkPi examples-1.0-SNAPSHOT-shaded.jar

OSSSample：spark-submit --class OSSSample examples-1.0-SNAPSHOT-shaded.jar <inputPath> <numPartition>

参数说明如下：

inputPath: 输入数据路径。

numPartition：输入数据RDD分片数目。

ONSSample： spark-submit --class ONSSample examples-1.0-SNAPSHOT-shaded.jar <accessKeyId> <accessKeySecret> <consumerId> <topic> <subExpression> <parallelism>

参数说明如下：

accessKeyId： 阿里云 AccessKeyId。

accessKeySecret：阿里云 AccessKeySecret。

consumerId: 请参考 Consumer ID 说明。

topic: 每个消息队列都有一个 topic。

subExpression: 参考消息过滤。

parallelism：指定多少个接收器来消费队列消息。

ODPSSample: spark-submit --class ODPSSample examples-1.0-SNAPSHOT-shaded.jar <accessKeyId> <accessKeySecret> <envType> <project> <table> <numPartitions>

参数说明如下：

accessKeyId： 阿里云 AccessKeyId。

accessKeySecret：阿里云 AccessKeySecret。

envType： 0 表示公网环境，1 表示内网环境。如果是本地调试选择 0，如果是在 E-MapReduce 上执行请选择 1。

project：请参考ODPS-快速开始。

table：请参考 ODPS 术语介绍。

numPartition：输入数据 RDD 分片数目。

MNSSample: spark-submit --class MNSSample examples-1.0-SNAPSHOT-shaded.jar <queueName> <accessKeyId> <accessKeySecret> <endpoint>

参数说明如下：

queueName：队列名，请参考 MNS 名词解释。

accessKeyId： 阿里云 AccessKeyId。

accessKeySecret：阿里云 AccessKeySecret。

endpoint：队列数据访问地址。

LoghubSample: spark-submit --class LoghubSample examples-1.0-SNAPSHOT-shaded.jar <sls project> <sls logstore> <loghub group name> <sls endpoint> <access key id> <access key secret> <batch interval seconds>

参数说明如下：

sls project: LogService 项目名。

sls logstore：日志库名。

loghub group name：作业中消费日志数据的组名，可以任意取。sls project 和 sls store 相同时，相同组名的作业会协同消费 sls store 中的数据；不同组名的作业会相互隔离地消费 sls store 中的数据。

sls endpoint： 请参考日志服务入口。

accessKeyId： 阿里云 AccessKeyId。

accessKeySecret：阿里云 AccessKeySecret。

batch interval seconds： Spark Streaming 作业的批次间隔，单位为秒。

LinearRegression: spark-submit --class LinearRegression examples-1.0-SNAPSHOT-shaded.jar <inputPath> <numPartitions>

参数说明如下：

inputPath：输入数据。

numPartition：输入数据 RDD 分片数目。

Mapreduce

WordCount： hadoop jar examples-1.0-SNAPSHOT-shaded.jar WordCount -Dwordcount.case.sensitive=true <inputPath> <outputPath> -skip <patternPath>

参数说明如下：

inputPathl：输入数据路径。

outputPath：输出路径。

patternPath：过滤字符文件，可以使用 data/patterns.txt。

Hive

hive -f sample.hive -hiveconf inputPath=<inputPath>

参数说明如下：

- inputPath：输入数据路径。

Pig

pig -x mapreduce -f sample.pig -param tutorial=<tutorialJarPath> -param input=<inputPath> -param result=<resultPath>

参数说明如下：

tutorialJarPath：依赖 Jar 包，可使用 lib/tutorial.jar。

inputPath：输入数据路径。

resultPath：输出路径。

注意：

- 在 E-MapReduce 上使用时，请将测试数据和依赖 jar 包上传到 OSS 中，路径规则遵循 OSSURI 定义，见上。
- 如果集群中使用，可以放在机器本地。

本地运行

这里主要介绍如何在本地运行 Spark 程序访问阿里云数据源，例如 OSS 等。如果希望本地调试运行，最好借助一些开发工具，例如 Intellij IDEA 或者 Eclipse，尤其是对于 Windows环境，否则需要在 Windows 机器上配置 Hadoop 和 Spark 运行环境。

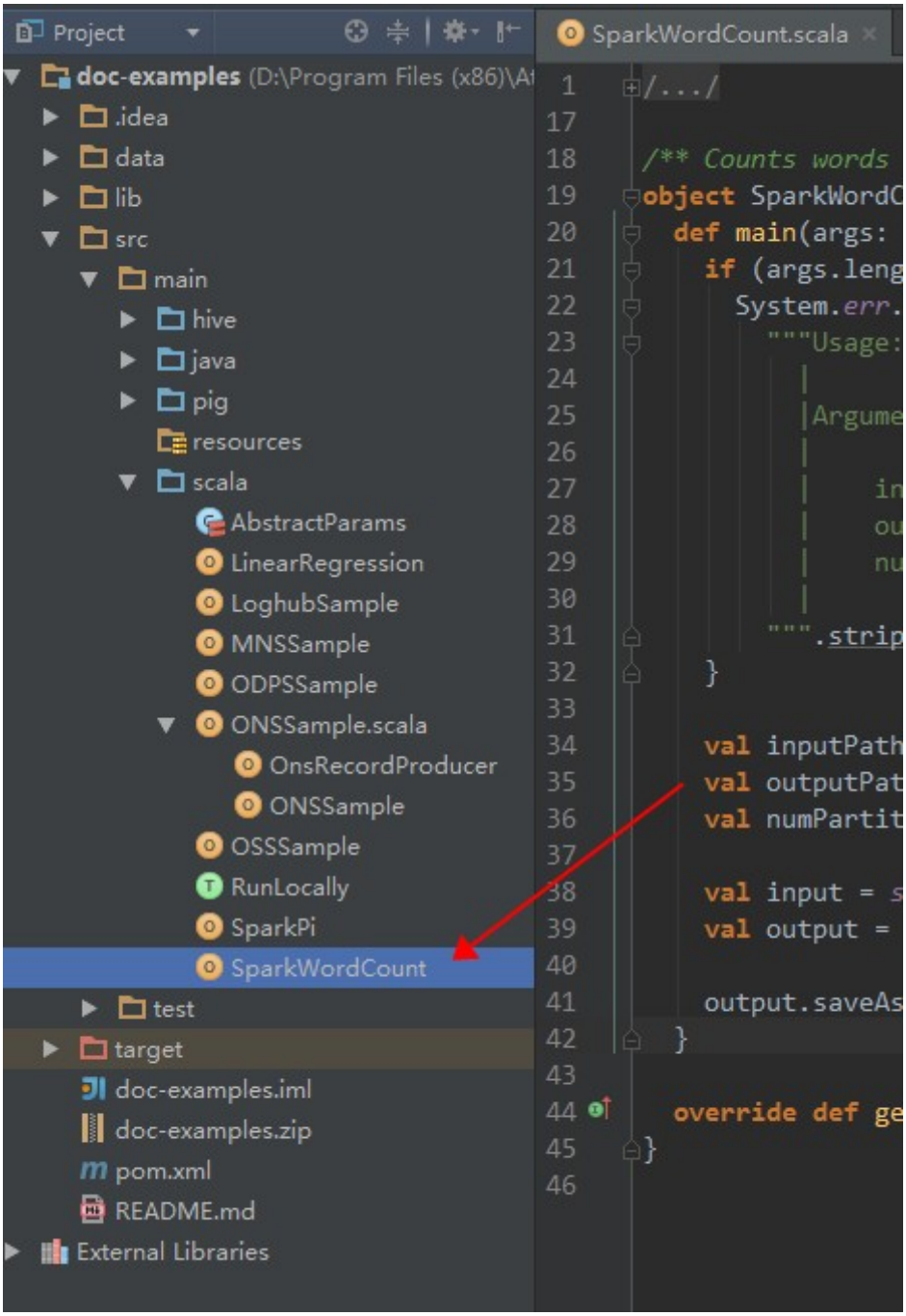
Intellij IDEA

准备工作

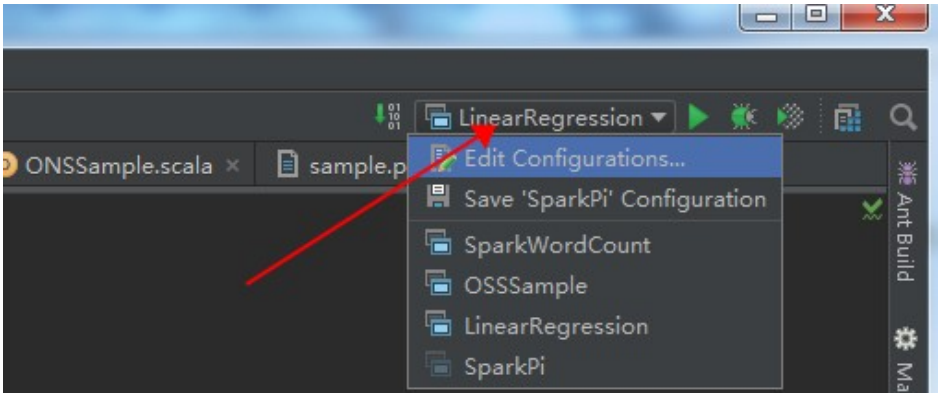
安装Intellij IDEA，Maven， Intellij IDEA Maven插件，Scala，Intellij IDEA Scala插件。

开发流程

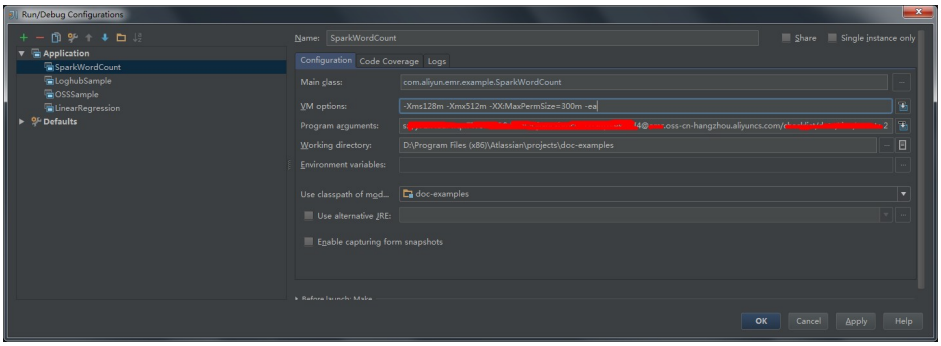
双击进入 SparkWordCount.scala。



从下图箭头所指处进入作业配置界面。

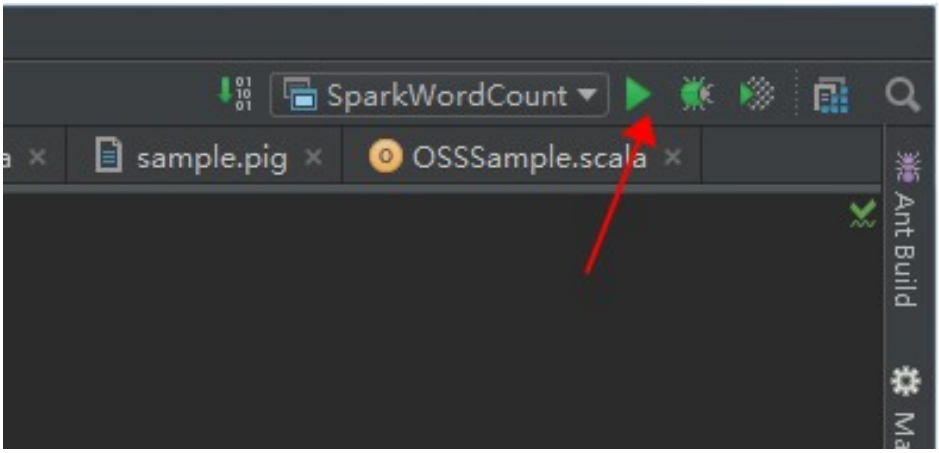


选择 SparkWordCount，在作业参数框中按照所需传入作业参数。

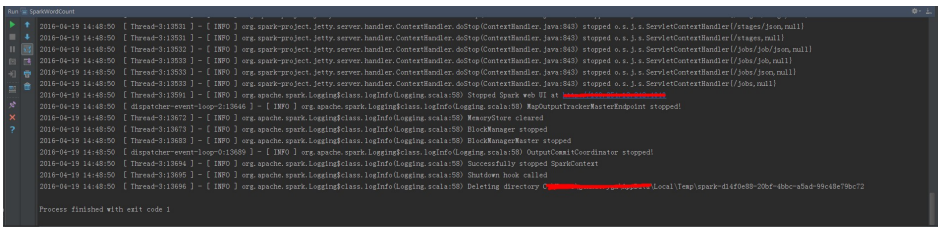


点击 **OK**。

点击**运行按钮**，执行作业。



查看作业执行日志



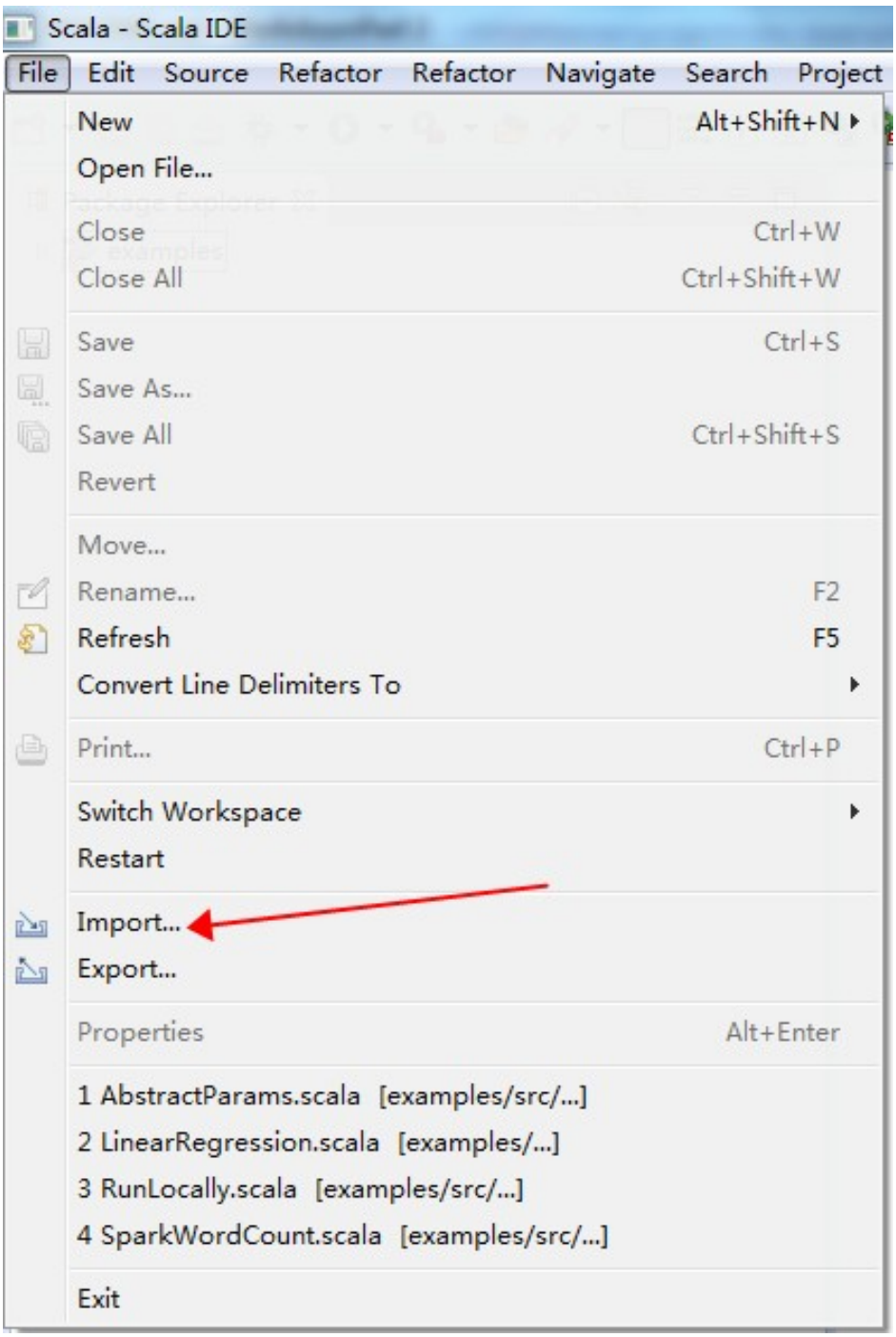
Scala IDE for Eclipse

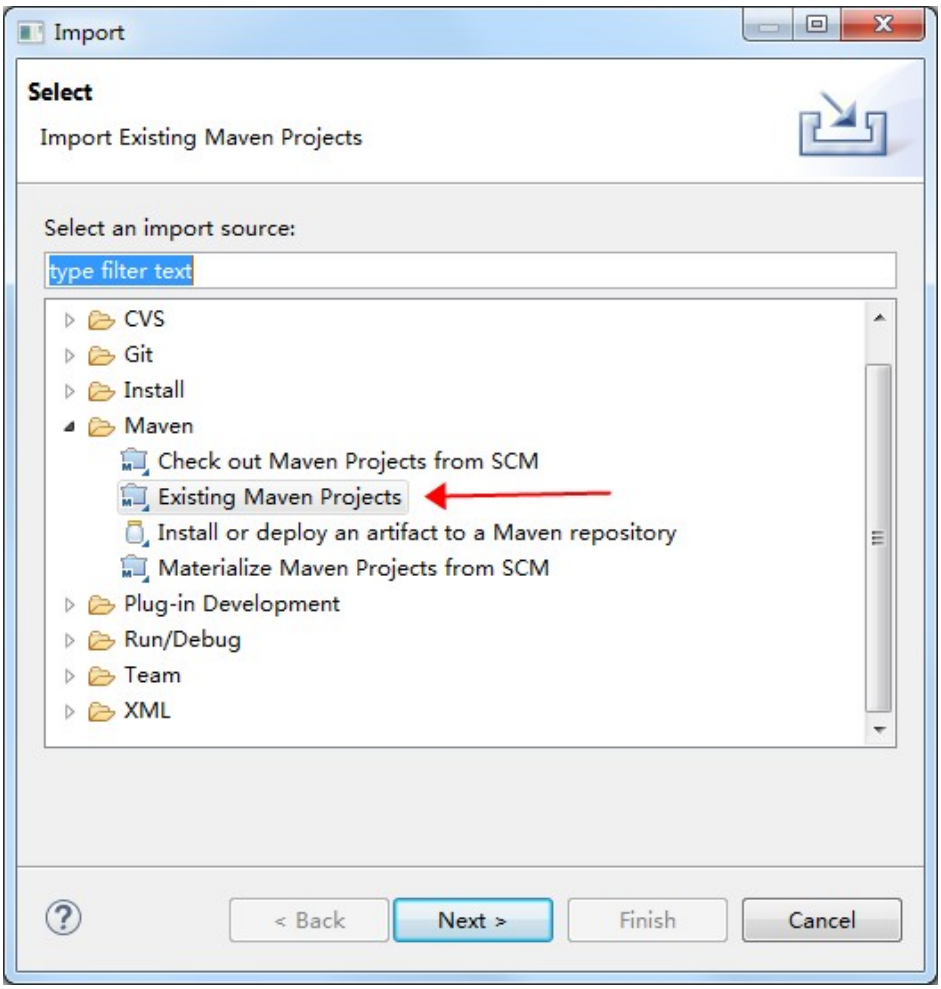
准备工作

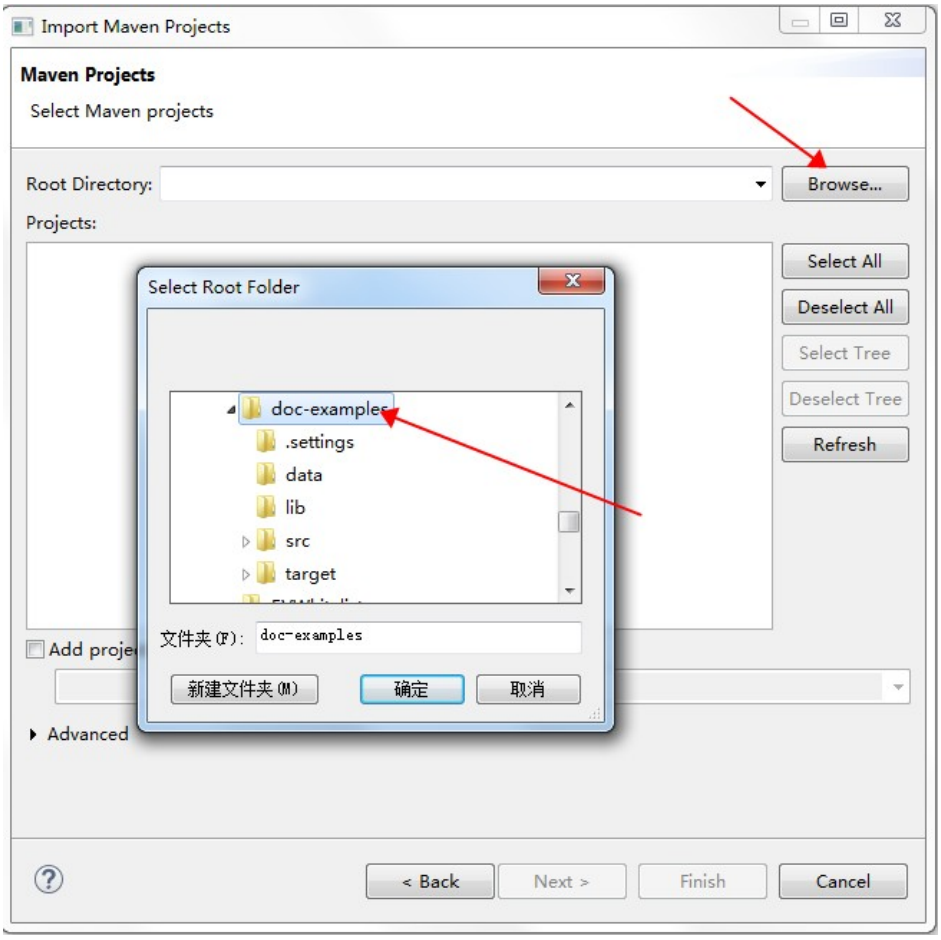
安装 Scala IDE for Eclipse、Maven、Eclipse Maven 插件。

开发流程

请根据以下图示导入项目。


Scala - Scala IDE
File Edit Source Refactor Refactor Navigate Search Project
New Alt+Shift+N
Open File...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
1 AbstractParams.scala [examples/src/...]
2 LinearRegression.scala [examples/...]
3 RunLocally.scala [examples/src/...]
4 SparkWordCount.scala [examples/src/...]
Exit

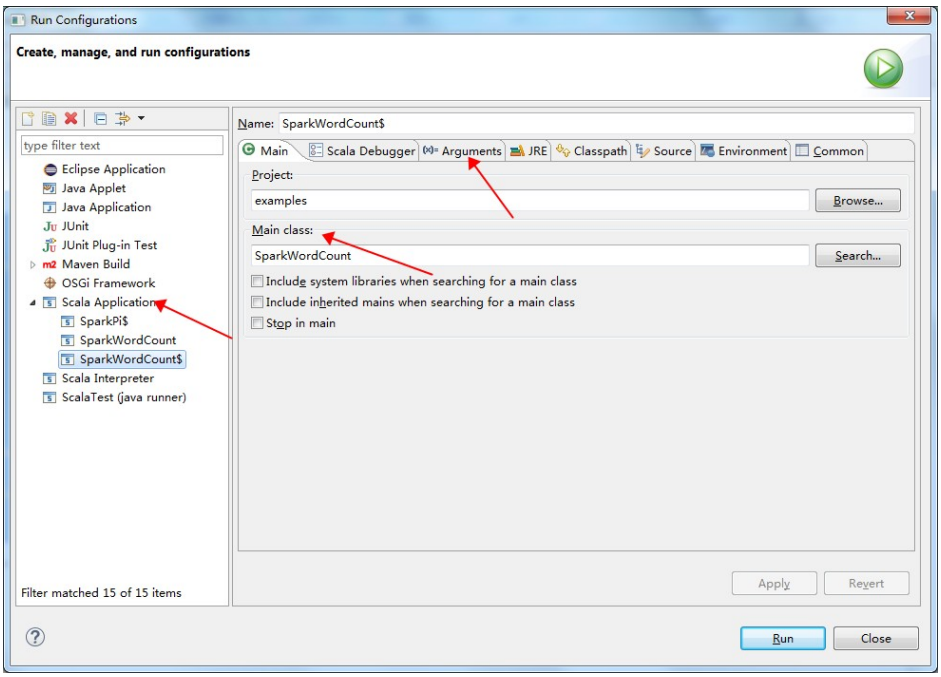

Import
Select
Import Existing Maven Projects
Select an import source:
type filter text
CVS
Git
Install
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM
Plug-in Development
Run/Debug
Team
XML
< Back
Next >
Finish
Cancel



Run As Maven build，快捷键是“Alt + Shilft + X, M”；也可以在项目名上右键，“Run As”选择“Maven build”。

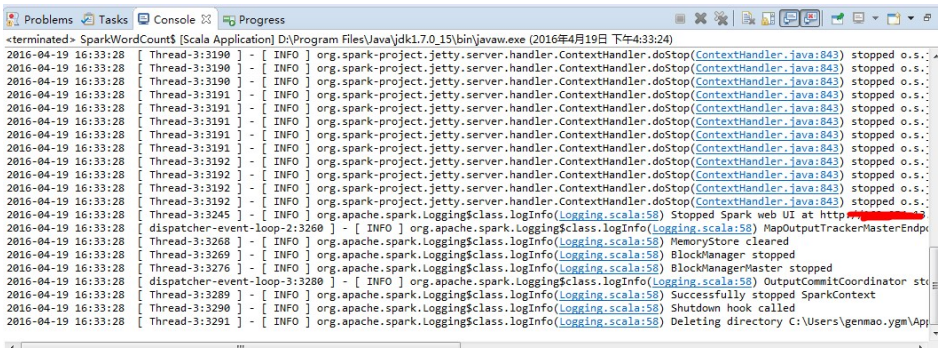
等待编译完后，在需要运行的作业上右键，选择“Run Configuration”，进入配置页。

在配置页中，选择 Scala Application，并配置作业的 Main Class 和参数等等。如下图所示：



点击“Run”。

查看控制台输出日志，如下图所示：



Spark

安装 E-MapReduce SDK

你可以通过以下两种方法安装 E-MapReduce SDK。

方法一：直接在 Eclipse 中使用 JAR 包，步骤如下：

从Maven库源下载E-MapReduce开发的依赖包。

将所需的jar包拷贝到您的工程文件夹中。（SDK版本是否选择2.10还是2.11主要根据您运行作业的集群Spark版本，Spark1.x版本建议使用2.10，Spark2.x建议使用2.11）

在 Eclipse 中右击工程的名字，然后单击**Properties** -> **Java Build Path** -> **Add JARs** 。

选择您下载的 SDK。

经过上面几步之后，您就可以在工程中读写 OSS、LogService、MNS、ONS、OTS、MaxCompute 等数据了。

方法二：Maven 工程的方式，请添加如下依赖：

```
   <!--支持OSS数据源 -->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-core</artifactId>
<version>1.4.1</version>
</dependency>

<!--支持OTS数据源-->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-tablestore</artifactId>
<version>1.4.1</version>
</dependency>

<!-- 支持 MNS、ONS、LogService、MaxCompute数据源 (Spark 1.x环境)-->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-mns_2.10</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-logservice_2.10</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-maxcompute_2.10</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-ons_2.10</artifactId>
```

```
<version>1.4.1</version>
</dependency>

<!-- 支持 MNS、ONS、LogService、MaxCompute数据源 (Spark 2.x环境)-->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-mns_2.11</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-logservice_2.11</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-maxcompute_2.11</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-ons_2.11</artifactId>
<version>1.4.1</version>
</dependency>
```

Spark 代码本地调试

注意：“spark.hadoop.mapreduce.job.run-local”这个配置项只是针对需要在本地调试 Spark 代码读写 OSS 数据的场景，除此之外只需要保持默认即可。

如果需要本地调试运行 Spark 代码读写 OSS 数据，则需要对 SparkConf 进行配置，将“spark.hadoop.mapreduce.job.run-local”设为“true”，例如如下代码：

```
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")
val sc = new SparkContext(conf)

val data = sc.textFile("oss://...")
println(s"count: ${data.count()}")
```

三方依赖说明

为了支持在E-MapReduce上操作阿里云的数据源（包括 OSS、MaxCompute等），需要您的作业依赖一些三方包。

您可以参照这个pom文件来增删需要依赖的三方包。

垃圾清理

Spark作业失败后，已产生的数据是不会主动清理掉的。当您运行Spark作业失败后，请检查一下OSS输出目录是否有文件存在，您还需要检查OSS碎片管理中是否还有没有提交的碎片存在，如果存在请及时清理掉。

pyspark 使用说明

如果您使用的是python来编写Spark作业，那么请参考这里配置您的环境。

Spark 代码中可使用如下参数配置：

属性名	默认值	说明
spark.hadoop.fs.oss.accessKeyId	无	访问 OSS 所需的 Access Key ID（可选）
spark.hadoop.fs.oss.accessKeySecret	无	访问 OSS 所需的 Access Key Secret（可选）
spark.hadoop.fs.oss.securityToken	无	访问 OSS 所需的 STS token（可选）
spark.hadoop.fs.oss.endpoint	无	访问 OSS 的 endpoint（可选）
spark.hadoop.fs.oss.multipart.thread.number	5	并发进行 OSS 的 upload part copy 的并发度
spark.hadoop.fs.oss.copy.simple.max.byte	134217728	使用普通接口进行 OSS 内部 copy 的文件大小上限
spark.hadoop.fs.oss.multipart.split.max.byte	67108864	使用普通接口进行 OSS 内部 copy 的文件分片大小上限
spark.hadoop.fs.oss.multipart.split.number	5	使用普通接口进行 OSS 内部 copy 的文件分片数目，默认和拷贝并发数目保持一致
spark.hadoop.fs.oss.impl	com.aliyun.fs.oss.nat.NativeOssFileSystem	OSS 文件系统实现类
spark.hadoop.fs.oss.buffer.dirs	/mnt/disk1,/mnt/disk2,...	OSS 本地临时文件目录，默认使用集群的数据盘
spark.hadoop.fs.oss.buffer.dirs.exists	false	是否确保 OSS 临时目录已经存在
spark.hadoop.fs.oss.client.connection.timeout	50000	OSS Client 端的连接超时时间（单位毫秒）
spark.hadoop.fs.oss.client.socket.timeout	50000	OSS Client 端的 socket 超时时间（单位毫秒）
spark.hadoop.fs.oss.client.co	-1	连接存活时间

nnection.ttl		
spark.hadoop.fs.oss.connection.max	1024	最大连接数目
spark.hadoop.job.runlocal	false	当数据源是 OSS 时，如果需要本地调试运行 Spark 代码，需要设置此项为“true”，否则为“false”
spark.logservice.fetch.interval.millis	200	Receiver 向 LogHub 取数据的时间间隔
spark.logservice.fetch.inOrder	true	是否有序消费分裂后的 Shard 数据
spark.logservice.heartbeat.interval.millis	30000	消费进程的心跳保持间隔
spark.mns.batchMsg.size	16	批量拉取 MNS 消息条数，最大不能超过 16
spark.mns.pollingWait.seconds	30	MNS 队列为空时的拉取等待间隔
spark.hadoop.io.compression.codec.snappy.native	false	标识 Snappy 文件是否为标准 Snappy 文件，Hadoop 默认识别的是 Hadoop 修改过的 Snappy 格式文件

使用 OSS SDK 存在的问题

若在 Spark 或者 Hadoop 作业中无法直接使用 OSS SDK 来操作 OSS 中的文件，是因为OSS SDK 中依赖的 http-client-4.4.x 版本与 Spark 或者 Hadoop 运行环境中的 http-client 存在版本冲突。如果要这么做，就必须先解决这个依赖冲突问题。实际上在 E-MapReduce 中，Spark 和 Hadoop 已经对 OSS 做了无缝兼容，可以像使用 HDFS 一样来操作 OSS 文件。

- 当前E-MapReduce环境支持MetaService服务，可以支持在E-MapReduce环境面AK访问OSS数据。旧的显示写AK的方式依旧支持，请注意在操作OSS的时候优先使用内网的Endpoint。
- 当您需要在本地进行测试的时候，才要用到OSS的外网的Endpoint，这样才能从本地访问到OSS的数据。

所有的Endpint可以参考 OSS Endpoint。

推荐做法（以免AK方式为例）

请您使用如下方法来查询 OSS 目录下的文件：

```
[Scala]
```

```
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.{Path, FileSystem}

val dir = "oss://bucket/dir"
val path = new Path(dir)
val conf = new Configuration()
conf.set("fs.oss.impl", "com.aliyun.fs.oss.nat.NativeOssFileSystem")

val fs = FileSystem.get(path.toUri, conf)
val fileList = fs.listStatus(path)

...


[Java]

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;

String dir = "oss://bucket/dir";
Path path = new Path(dir);
Configuration conf = new Configuration();
conf.set("fs.oss.impl", "com.aliyun.fs.oss.nat.NativeOssFileSystem");

FileSystem fs = FileSystem.get(path.toUri(), conf);
FileStatus[] fileList = fs.listStatus(path);

...
```

Spark + OSS

Spark 接入 OSS

当前E-MapReduce支持MetaService服务，支持用户在E-MapReduce环境免AK访问OSS数据源。旧的显式写AK和Endpoint方式也支持，但需要注意OSS Endpoint请使用内网域名，所有的Endpoint可以参考 OSS Endpoint。

下面这个例子演示了Spark如何免AK从OSS中读入数据，并将处理完的数据写回到OSS 中。

```
    val conf = new SparkConf().setAppName("Test OSS")
val sc = new SparkContext(conf)

val pathIn = "oss://bucket/path/to/read"
val inputData = sc.textFile(pathIn)
val cnt = inputData.count
println(s"count: $cnt")
```

```
val outputPath = "oss://bucket/path/to/write"
val outpuData = inputData.map(e => s"$e has been processed.")
outpuData.saveAsTextFile(outputPath)
```

附录

示例代码请看:

- Spark接入OSS

Spark + MaxCompute

Spark 接入 MaxCompute

本章节将介绍如何使用E-MapReduce SDK在Spark中完成一次MaxCompute数据的读写操作。

初始化一个OdpsOps对象。在 Spark 中，MaxCompute的数据操作通过OdpsOps类完成，请参照如下步骤创建一个OdpsOps对象：

```
import com.aliyun.odps.TableSchema
import com.aliyun.odps.data.Record
import org.apache.spark.aliyun.odps.OdpsOps
import org.apache.spark.{SparkContext, SparkConf}

object Sample {
def main(args: Array[String]): Unit = {
// == Step-1 ==
val accessKeyId = "<accessKeyId>"
val accessKeySecret = "<accessKeySecret>"
// 以内网地址为例
val urls = Seq("http://odps-ext.aliyun-inc.com/api", "http://dt-ext.odps.aliyun-inc.com")

val conf = new SparkConf().setAppName("Test Odps")
val sc = new SparkContext(conf)
val odpsOps = OdpsOps(sc, accessKeyId, accessKeySecret, urls(0), urls(1))

// 下面是一些调用代码
// == Step-2 ==
...
// == Step-3 ==
...
}

// == Step-2 ==
// 方法定义1
// == Step-3 ==
```

```
// 方法定义2
 }
```

从MaxCompute中加载表数据到Spark中。通过OdpsOps对象的readTable方法，可以将MaxCompute中的表加载到Spark中，即生成一个RDD，如下所示：

```
// == Step-2 ==
val project = <odps-project>
val table = <odps-table>
val numPartitions = 2
val inputData = odpsOps.readTable(project, table, read, numPartitions)
inputData.top(10).foreach(println)

// == Step-3 ==
...
```

在上面的代码中，您还需要定义一个read函数，用来解析和预处理MaxCompute表数据，如下所示：

```
def read(record: Record, schema: TableSchema): String = {
record.getString(0)
}
```

这个函数的含义是将MaxCompute表的第一列加载到Spark运行环境中。

将 Spark 中的结果数据保存到MaxCompute表中。通过OdpsOps对象的saveToTable方法，可以将Spark RDD持久化到MaxCompute中。

```
val resultData = inputData.map(e => s"$e has been processed.")
odpsOps.saveToTable(project, table, dataRDD, write)
```

在上面的代码中，您还需要定义一个write函数，用作写MaxCompute表前数据预处理，如下所示：

```
def write(s: String, emptyReord: Record, schema: TableSchema): Unit = {
val r = emptyReord
r.set(0, s)
}
```

这个函数的含义是将RDD的每一行数据写到对应MaxCompute表的第一列中。

分区表参数写法说明

SDK支持对MaxCompute分区表的读写，这里分区名的写法标准是：分区列名=分区名，多个分区时以逗号分隔，例如有分区列pt和ps：

- 读分区pt为1的表数据：pt= ‘1’
- 读分区pt为1和分区ps为2的表数据：pt= ‘1’ ,ps= ‘2’

附录

示例代码请看:

- Spark接入MaxCompute

Spark + ONS

Spark 接入 ONS

下面这个例子演示了 Spark Streaming 如何消费 ONS 中的数据，统计每个 batch 内的单词个数。

```
    val Array(cId, topic, subExpression, parallelism, interval) = args

val accessKeyId = "<accessKeyId>"
val accessKeySecret = "<accessKeySecret>"

val numStreams = parallelism.toInt
val batchInterval = Milliseconds(interval.toInt)

val conf = new SparkConf().setAppName("Test ONS Streaming")
val ssc = new StreamingContext(conf, batchInterval)
def func: Message => Array[Byte] = msg => msg.getBody
val onsStreams = (0 until numStreams).map { i =>
println(s"starting stream $i")
OnsUtils.createStream(
ssc,
cId,
topic,
subExpression,
accessKeyId,
accessKeySecret,
StorageLevel.MEMORY_AND_DISK_2,
func)
}

val unionStreams = ssc.union(onsStreams)
unionStreams.foreachRDD(rdd => {
rdd.map(bytes => new String(bytes)).flatMap(line => line.split(" "))
.map(word => (word, 1))
.reduceByKey(_ + _).collect().foreach(e => println(s"word: ${e._1}, cnt: ${e._2}"))
```

```
})

ssc.start()
ssc.awaitTermination()
```

附录

示例代码请看:

- Spark接入ONS

Spark + TableStore

Spark接入TableStore

- 准备一张数据表

创建一张表pet，其中name为主键。

name	owner	species	sex	birth	death
Fluffy	Harold	cat	f	1993-02-04	
Claws	Gwen	cat	m	1994-03-17	
Buffy	Harold	dog	f	1989-05-13	
Fang	Benny	dog	m	1990-08-27	
Bowser	Diane	dog	m	1979-08-31	1995-07-29
Chirpy	Gwen	bird	f	1998-09-11	
Whistler	Gwen	bird		1997-12-09	
Slim	Benny	snake	m	1996-04-29	
Puffball	Diane	hamster	f	1999-03-30	

- 下面这个例子示例了如何在Spark中消费TableStore中的数据。

```
private static RangeRowQueryCriteria fetchCriteria() {
RangeRowQueryCriteria res = new RangeRowQueryCriteria("pet");
res.setMaxVersions(1);
List<PrimaryKeyColumn> lower = new ArrayList<PrimaryKeyColumn>();
List<PrimaryKeyColumn> upper = new ArrayList<PrimaryKeyColumn>();
lower.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MIN));
upper.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MAX));
res.setInclusiveStartPrimaryKey(new PrimaryKey(lower));
```

```
res.setExclusiveEndPrimaryKey(new PrimaryKey(upper));
return res;
}

public static void main(String[] args) {
SparkConf sparkConf = new SparkConf().setAppName("RowCounter");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

Configuration hadoopConf = new Configuration();
JavaSparkContext sc = null;
try {
sc = new JavaSparkContext(sparkConf);
Configuration hadoopConf = new Configuration();
TableStore.setCredential(
hadoopConf,
new Credential(accessKeyId, accessKeySecret, securityToken));
Endpoint ep = new Endpoint(endpoint, instance);
TableStore.setEndpoint(hadoopConf, ep);
TableStoreInputFormat.addCriteria(hadoopConf, fetchCriteria());

JavaPairRDD<PrimaryKeyWritable, RowWritable> rdd = sc.newAPIHadoopRDD(
hadoopConf, TableStoreInputFormat.class,
PrimaryKeyWritable.class, RowWritable.class);
System.out.println(
new Formatter().format("TOTAL: %d", rdd.count()).toString());
} finally {
if (sc != null) {
sc.close();
}
}
}
```

附录

完整示例代码请参考：

- Spark接入TableStore

Spark + LogService

Spark 接入 LogService

下面这个例子演示了Spark Streaming如何消费LogService中的日志数据，统计日志条数。

方法一：Receiver Based DStream

```
    val logServiceProject = args(0)    // LogService 中 project 名
```

```
val logStoreName = args(1) // LogService 中 logstore 名
val loghubConsumerGroupName = args(2) // loghubGroupName 相同的作业将共同消费 logstore 的数据
val loghubEndpoint = args(3) // 阿里云日志服务数据类 API Endpoint
val accessKeyId = "<accessKeyId>" // 访问日志服务的 AccessKeyId
val accessKeySecret = "<accessKeySecret>" // 访问日志服务的 AccessKeySecret
val numReceivers = args(4).toInt // 启动多少个 Receiver 来读取 logstore 中的数据
val batchInterval = Milliseconds(args(5).toInt * 1000) // Spark Streaming 中每次处理批次时间间隔

val conf = new SparkConf().setAppName("Test Loghub Streaming")
val ssc = new StreamingContext(conf, batchInterval)
val loghubStream = LoghubUtils.createStream(
ssc,
logServiceProject,
logStoreName,
loghubConsumerGroupName,
loghubEndpoint,
numReceivers,
accessKeyId,
accessKeySecret,
StorageLevel.MEMORY_AND_DISK)

loghubStream.foreachRDD(rdd => println(rdd.count()))

ssc.start()
ssc.awaitTermination()
```

方法二: Direct API Based DStream

```
    val logServiceProject = args(0)
val logStoreName = args(1)
val loghubConsumerGroupName = args(2)
val loghubEndpoint = args(3)
val accessKeyId = args(4)
val accessKeySecret = args(5)
val batchInterval = Milliseconds(args(6).toInt * 1000)
val zkConnect = args(7)
val checkpointPath = args(8)

def functionToCreateContext(): StreamingContext = {
val conf = new SparkConf().setAppName("Test Direct Loghub Streaming")
val ssc = new StreamingContext(conf, batchInterval)
val zkParas = Map("zookeeper.connect" -> zkConnect, "enable.auto.commit" -> "false")
val loghubStream = LoghubUtils.createDirectStream(
ssc,
logServiceProject,
logStoreName,
loghubConsumerGroupName,
accessKeyId,
accessKeySecret,
loghubEndpoint,
zkParas,
LogHubCursorPosition.END_CURSOR)

ssc.checkpoint(checkpointPath)
```

```
val stream = loghubStream.checkpoint(batchInterval)
stream.foreachRDD(rdd => {
println(rdd.count())
loghubStream.asInstanceOf[DirectLoghubInputDStream].commitAsync()
})

ssc
}

val ssc = StreamingContext.getOrCreate(checkpointPath, functionToCreateContext _)

ssc.start()
ssc.awaitTermination()
```

从E-MapReduce SDK 1.4.0版本开始，提供基于Direct API的实现方式。这种方式可以避免将Loghub数据重复存储到Write Ahead Log中，也即无需开启Spark Streaming的WAL特性即可实现数据的at least once。目前Direct API实现方式处于experimental状态，需要注意的地方有：

- 在DStream的action中，必须做一次commit操作。
- 一个Spark Streaming中，不支持对logstore数据源做多个action操作。
- Direct API方式需要zookeeper服务的支持。

支持MetaService

上面的例子中，我们都是显式地将AK传入到接口中。不过从E-MapReduce SDK 1.3.2版本开始，Spark Streaming可以基于MetaService实现免AK处理LogService数据。具体可以参考E-MapReduce SDK中的LoghubUtils类说明:

```
LoghubUtils.createStream(ssc, logServiceProject, logStoreName, loghubConsumerGroupName, storageLevel)
LoghubUtils.createStream(ssc, logServiceProject, logStoreName, loghubConsumerGroupName, numReceivers,
storageLevel)
LoghubUtils.createStream(ssc, logServiceProject, logStoreName, loghubConsumerGroupName, storageLevel,
cursorPosition, mLoghubCursorStartTime, forceSpecial)
LoghubUtils.createStream(ssc, logServiceProject, logStoreName, loghubConsumerGroupName, numReceivers,
storageLevel, cursorPosition, mLoghubCursorStartTime, forceSpecial)
```

说明

- E-MapReduce SDK支持LogService的三种消费模式，即“BEGIN_CURSOR”，“END_CURSOR”和“SPECIAL_TIMER_CURSOR”，默认是“END_CURSOR”。
  - BEGIN_CURSOR：从日志头开始消费，如果有checkpoint记录，则从checkpoint处开始消费。
  - END_CURSOR：从日志尾开始消费，如果有checkpoint记录，则从checkpoint处开始消费。
  - SPECIAL_TIMER_CURSOR：从指定时间点开始消费，如果有checkpoint记录，则从

checkpoint处开始消费。单位为秒。
- 以上三种消费模式都受到checkpoint记录的影响，如果存在checkpoint记录，则从checkpoint处开始消费，不管指定的是什么消费模式。E-MapReduce SDK基于“SPECIAL_TIMER_CURSOR”模式支持用户强制在指定时间点开始消费：在LoghubUtils#createStream接口中，以下参数需要组合使用：
    - cursorPosition：LogHubCursorPosition.SPECIAL_TIMER_CURSOR
    - forceSpecial：true

- E-MapReduce 的机器（除了 Master 节点）无法连接公网。配置 LogService endpoint 时，请注意使用 Log Service 提供的内网 endpoint，否则无法请求到 Log Service。
- 更多关于 LogService，请查看文档。

附录

完整示例代码请看:

- Spark 接入 LogService

Spark + MNS

Spark 接入 MNS

下面这个例子演示了 Spark Streaming 如何消费 MNS 中的数据，统计每个 batch 内的单词个数。

```
val conf = new SparkConf().setAppName("Test MNS Streaming")
val batchInterval = Seconds(10)
val ssc = new StreamingContext(conf, batchInterval)

val queuename = "queuename"
val accessKeyId = "<accessKeyId>"
val accessKeySecret = "<accessKeySecret>"
val endpoint = "http://xxx.yyy.zzzz/abc"

val mnsStream = MnsUtils.createPullingStreamAsRawBytes(ssc, queuename, accessKeyId, accessKeySecret,
endpoint,
StorageLevel.MEMORY_ONLY)
mnsStream.foreachRDD( rdd => {
rdd.map(bytes => new String(bytes)).flatMap(line => line.split(" "))
.map(word => (word, 1))
.reduceByKey(_ + _).collect().foreach(e => println(s"word: ${e._1}, cnt: ${e._2}"))
})

ssc.start()
ssc.awaitTermination()
```

支持MetaService

上面的例子中，我们都是显式地将AK传入到接口中。不过从E-MapReduce SDK 1.3.2版本开始，Spark Streaming可以基于MetaService实现免AK处理MNS数据。具体可以参考E-MapReduce SDK中的MnsUtils类说明：

```
MnsUtils.createPullingStreamAsBytes(ssc, queueName, endpoint, storageLevel)
MnsUtils.createPullingStreamAsRawBytes(ssc, queueName, endpoint, storageLevel)
```

附录

完整示例代码请看:

- Spark 接入 MNS

Spark + Hbase

Spark 接入 Hbase

下面这个例子演示了 Spark 如何向 Hbase 写数据。需要指出的是，计算集群需要和 Hbase 集群处于一个安全组内，否则网络无法打通。在 E-Mapreduce 创建集群时，请注意选择 Hbase 集群所处的安全组。

```
    object ConnectionUtil extends Serializable {
private val conf = HBaseConfiguration.create()
conf.set(HConstants.ZOOKEEPER_QUORUM,"ecs1,ecs1,ecs3")
conf.set(HConstants.ZOOKEEPER_ZNODE_PARENT, "/hbase")
private val connection = ConnectionFactory.createConnection(conf)

def getDefaultConn: Connection = connection
}

//创建数据流 unionStreams
unionStreams.foreachRDD(rdd => {
rdd.map(bytes => new String(bytes))
.flatMap(line => line.split(" "))
.map(word => (word, 1))
.reduceByKey(_ + _)
.mapPartitions {words => {
val conn = ConnectionUtil.getDefaultConn
val tableName = TableName.valueOf(tname)
val t = conn.getTable(tableName)
try {
words.sliding(100, 100).foreach(slice => {
val puts = slice.map(word => {
```

```
println(s"word: $word")
val put = new Put(Bytes.toBytes(word._1 + System.currentTimeMillis()))
put.addColumn(COLUMN_FAMILY_BYTES, COLUMN_QUALIFIER_BYTES,
System.currentTimeMillis(), Bytes.toBytes(word._2))
put
}).toList
t.put(puts)
})
} finally {
t.close()
}

Iterator.empty
}}.count()
})

ssc.start()
ssc.awaitTermination()
```

附录

完整示例代码请看:

- Spark 接入 Hbase

本章节将介绍如何在 E-MapReduce 场景下设置 spark-submit 的参数。

集群配置

软件配置

E-MapReduce 产品版本 1.1.0

Hadoop 2.6.0

Spark 1.6.0

硬件配置

Master 节点

8 核 16G 500G 高效云盘

1 台

Worker 节点 x 10 台

8 核 16G 500G 高效云盘

10 台

总资源：8 核 16G（Worker）x 10 + 8 核 16G（Master）

注意：由于作业提交的时候资源只计算 CPU 和内存，所以这里磁盘的大小并未计算到总资源中。

Yarn 可分配总资源：12 核 12.8G（worker）x 10

注意：默认情况下，yarn 可分配核 = 机器核 x 1.5，yarn 可分配内存 = 机器内存 x 0.8。

提交作业

创建集群后，您可以提交作业。首先，您需要在 E-MapReduce 中创建一个作业，配置如下：

上图所示的作业，直接使用了 Spark 官方的 example 包，所以不需要自己上传 jar 包。

参数列表如下所示：

```
--class org.apache.spark.examples.SparkPi --master yarn --deploy-mode client --driver-memory 4g –num-executors 2 --executor-memory 2g --executor-cores 2 /opt/apps/spark-1.6.0-bin-hadoop2.6/lib/spark-examples*.jar 10
```

参数说明如下所示：

参数	参考值	说明
class	org.apache.spark.examples.SparkPi	作业的主类。
master	yarn	因为 E-MapReduce 使用 Yarn 的模式，所以这里只能是 yarn 模式。
	yarn-client	等同于 –-master yarn —deploy-mode client， 此时不需要指定deploy-mode。
	yarn-cluster	等同于 –-master yarn —deploy-mode cluster， 此时不需要指定deploy-mode。
deploy-mode	client	client 模式表示作业的 AM 会放在 Master 节点上运行。要注意的是，如果设置这个参数，那么需要同时指定上面 master 为 yarn。
	cluster	cluster 模式表示 AM 会随机的在 worker 节点中的任意一台上启动运行。要注意的是，如果设置这个参数，那么需要同时指定上面 master 为yarn。
driver-memory	4g	driver 使用的内存，不可超过单机的 core 总数。
num-executors	2	创建多少个 executor。
executor-memory	2g	各个 executor 使用的最大内存，不可超过单机的最大可使用内存。
executor-cores	2	各个 executor 使用的并发线程数目，也即每个 executor 最大可并发执行的 Task 数目。

资源计算

在不同模式、不同的设置下运行时，作业使用的资源情况如下表所示：

varn-client 模式的资源计算

节点	资源类型	资源量（结果使用上面的例子计

		算得到）
master	core	1
	mem	driver-memroy = 4G
worker	core	num-executors * executor-cores = 4
	mem	num-executors * executor-memory = 4G

作业主程序（Driver 程序）会在 master 节点上执行。按照作业配置将分配 4G（由 —driver-memroy 指定）的内存给它（当然实际上可能没有用到）。

会在 worker 节点上起 2 个（由 —num-executors 指定）executor，每一个 executor 最大能分配 2G（由 —executor-memory 指定）的内存，并最大支持 2 个（由- -executor-cores 指定）task 的并发执行。

yarn-cluster 模式的资源计算

节点	资源类型	资源量（结果使用上面的例子计算得到）
master		一个很小的 client 程序，负责同步 job 信息，占用很小。
worker	core	num-executors * executor-cores+spark.driver.cores = 5
	mem	num-executors * executor-memory + driver-memroy = 8g

注意：这里的 spark.driver.cores 默认是 1，也可以设置为更多。

资源使用的优化

yarn-client 模式

若您有了一个大作业，使用 yarn-client 模式，想要多用一些这个集群的资源，请参见如下配置：

注意：

- Spark 在分配内存时，会在用户设定的内存值上溢出 375M 或 7%（取大值）。
- Yarn 分配 container 内存时，遵循向上取整的原则，这里也就是需要满足 1G 的整数倍。

```
--master yarn-client --driver-memory 5g –-num-executors 20 --executor-memory 4g --executor-cores 4
```

按照上述的资源计算公式，

master 的资源量为：

- core：1
- mem：6G (5G + 375M 向上取整为 6G)

workers 的资源量为：

- core: 20*4 = 80
- mem: 20*5G (4G + 375M 向上取整为 5G) = 100G

可以看到总的资源没有超过集群的总资源，那么遵循这个原则，您还可以有很多种配置，例如：

```
--master yarn-client --driver-memory 5g –num-executors 40 --executor-memory 1g --executor-cores 2

--master yarn-client --driver-memory 5g –num-executors 15 --executor-memory 4g --executor-cores 4

--master yarn-client --driver-memory 5g –num-executors 10 --executor-memory 9g --executor-cores 6
```

原则上，按照上述的公式计算出来的需要资源不超过集群的最大资源量就可以。但在实际场景中，因为系统，hdfs 以及 E-MapReduce 的服务会需要使用 core 和 mem 资源，如果把 core 和 mem 都占用完了，反而会导致性能的下降，甚至无法运行。

executor-cores 数一般也都会被设置成和集群的可使用核一致，因为如果设置的太多，CPU 会频繁切换，性能并不会提高。

yarn-cluster 模式

当使用 yarn-cluster 模式后，Driver 程序会被放到 worker 节点上。资源会占用到 worker 的资源池子里面，这时若想要多用一些这个集群的资源，请参加如下配置：

```
--master yarn-cluster --driver-memory 5g –num-executors 15 --executor-memory 4g --executor-cores 4
```

配置建议

如果将内存设置的很大，要注意 gc 所产生的消耗。一般我们会推荐一个 executor 的内存 <= 64G。

如果是进行 HDFS 读写的作业，建议是每个 executor 中使用 <= 5个并发来读写。

如果是进行 OSS 读写的作业，我们建议是将 executor 分布在不同的 ECS 上，这样可以将每一个

ECS 的带宽都用上。比如有 10 台 ECS，那么就可以配置 num-executors=10，并设置合理的内存和并发。

如果作业中使用了非线程安全的代码，那么在设置 executor-cores 的时候需要注意多并发是否会造成作业的不正常。如果会，那么推荐就设置 executor-cores=1。

Hadoop

Hadoop 代码中可使用如下参数配置：

属性名	默认值	说明
fs.oss.accessKeyId	无	访问 OSS 所需的 Access Key ID（可选）
fs.oss.accessKeySecret	无	访问 OSS 所需的 Access Key Secret（可选）
fs.oss.securityToken	无	访问 OSS 所需的 STS token（可选）
fs.oss.endpoint	无	访问 OSS 的 endpoint（可选）
fs.oss.multipart.thread.number	5	并发进行 OSS 的 upload part copy 的并发度
fs.oss.copy.simple.max.byte	134217728	使用普通接口进行 OSS 内部 copy 的文件大小上限
fs.oss.multipart.split.max.byte	67108864	使用普通接口进行 OSS 内部 copy 的文件分片大小上限
fs.oss.multipart.split.number	5	使用普通接口进行 OSS 内部 copy 的文件分片数目，默认和拷贝并发数目保持一致
fs.oss.impl	com.aliyun.fs.oss.nat.NativeOssFileSystem	OSS 文件系统实现类
fs.oss.buffer.dirs	/mnt/disk1,/mnt/disk2,...	OSS 本地临时文件目录，默认使用集群的数据盘
fs.oss.buffer.dirs.exists	false	是否确保 OSS 临时目录已经存在
fs.oss.client.connection.timeout	50000	OSS Client 端的连接超时时间（单位毫秒）
fs.oss.client.socket.timeout	50000	OSS Client 端的 socket 超时时间（单位毫秒）
fs.oss.client.connection.ttl	-1	连接存活时间

fs.oss.connection.max	1024	最大连接数目
io.compression.codec.snappy.native	false	标识 Snappy 文件是否为标准 Snappy 文件，Hadoop 默认识别的是 Hadoop 修改过的 Snappy 格式文件

在 MapReduce 中使用 OSS

要在 MapReduce 中读写 OSS，需要配置如下的参数

```
    conf.set("fs.oss.accessKeyId", "${accessKeyId}");
conf.set("fs.oss.accessKeySecret", "${accessKeySecret}");
conf.set("fs.oss.endpoint","${endpoint}");
```

参数说明：

${accessKeyId}： 您账号的 AccessKeyId。

${accessKeySecret}： 该 AccessKeyId 对应的密钥。

${endpoint}： 访问 OSS 使用的网络，由您集群所在的 region 决定，当然对应的 OSS 也需要是在集群对应的 region。

具体的值请参考 OSS Endpoint

Word Count

以下示例介绍了如何从 OSS 中读取文本，然后统计其中单词的数量。其操作步骤如下：

程序编写。以 JAVA 代码为例，将 Hadoop 官网 WordCount 例子做如下修改。对该实例的修改只是在代码中添加了 Access Key ID 和 Access Key Secret 的配置，以便作业有权限访问 OSS 文件。

```
package org.apache.hadoop.examples;

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

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;
```

```
public class EmrWordCount {

public static class TokenizerMapper
extends Mapper<Object, Text, Text, IntWritable>{

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

public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
word.set(itr.nextToken());
context.write(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,
Context context
) throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values) {
sum += val.get();
}
result.set(sum);
context.write(key, 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);
}

conf.set("fs.oss.accessKeyId", "${accessKeyId}");
conf.set("fs.oss.accessKeySecret", "${accessKeySecret}");
conf.set("fs.oss.endpoint","${endpoint}");

Job job = Job.getInstance(conf, "word count");
job.setJarByClass(EmrWordCount.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);
}
}
```

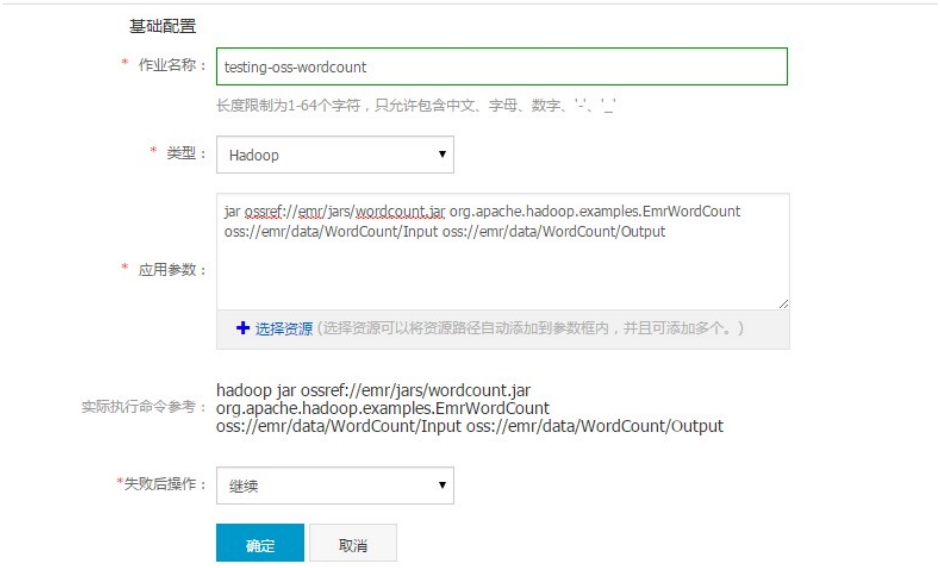
编译程序。首先要将 jdk 和 Hadoop 环境配置好，然后进行如下操作：

```
mkdir wordcount_classes
javac -classpath ${HADOOP_HOME}/share/hadoop/common/hadoop-common-
2.6.0.jar:${HADOOP_HOME}/share/hadoop/mapreduce/hadoop-mapreduce-client-core-
2.6.0.jar:${HADOOP_HOME}/share/hadoop/common/lib/commons-cli-1.2.jar -d wordcount_classes
EmrWordCount.java
jar cvf wordcount.jar -C wordcount_classes .
```

创建作业。

将上一步打好的 jar 文件上传到 OSS，具体可登录 OSS 官网进行操作。假设 jar 文件在 OSS 上的路径为 oss://emr/jars/wordcount.jar, 输入输出路径分别为 oss://emr/data/WordCount/Input 和 oss://emr/data/WordCount/Output。

在 E-MapReduce作业 中创建如下作业：



创建执行计划。在 E-MapReduce 执行计划中创建执行计划，将上一步创建好的作业添加到执行计划中，策略选择“立即执行”，这样 wordcount 作业就会在选定集群中运行起来了。

使用 Maven 工程来管理 MR 作业

当您的工程规模越来越大时，会变得非常复杂，不易管理。我们推荐你采用类似 Maven 这样的软件项目管理工具来进行管理。其操作步骤如下：

安装 Maven。首先确保您已经安装了 Maven。

生成工程框架。在您的工程根目录处（假设您的工程开发根目录位置是 D:/workspace）执行如下命令：

```
mvn archetype:generate -DgroupId=com.aliyun.emr.hadoop.examples -DartifactId=wordcountv2 -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

mvn 会自动生成一个空的 Sample 工程位于 D:/workspace/wordcountv2（和您指定的 artifactId 一致），里面包含一个简单的 pom.xml 和 App 类（类的包路径和您指定的 groupId 一致）。

加入 Hadoop 依赖。使用任意 IDE 打开这个工程，编辑 pom.xml 文件。在 dependencies 内添加如下内容：

```
    <dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-mapreduce-client-common</artifactId>
<version>2.6.0</version>
</dependency>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-common</artifactId>
<version>2.6.0</version>
</dependency>
```

编写代码。在 com.aliyun.emr.hadoop.examples 包下和 App 类平行的位置添加新类 WordCount2.java。内容如下：

```
package com.aliyun.emr.hadoop.examples;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;

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.mapreduce.Counter;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.StringUtils;

public class WordCount2 {

public static class TokenizerMapper
extends Mapper<Object, Text, Text, IntWritable>{

static enum CountersEnum { INPUT_WORDS }

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

private boolean caseSensitive;
private Set<String> patternsToSkip = new HashSet<String>();

private Configuration conf;
private BufferedReader fis;

@Override
public void setup(Context context) throws IOException,
InterruptedException {
conf = context.getConfiguration();
caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
if (conf.getBoolean("wordcount.skip.patterns", true)) {
URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
for (URI patternsURI : patternsURIs) {
Path patternsPath = new Path(patternsURI.getPath());
String patternsFileName = patternsPath.getName().toString();
parseSkipFile(patternsFileName);
}
}
}

private void parseSkipFile(String fileName) {
try {
fis = new BufferedReader(new FileReader(fileName));
String pattern = null;
while ((pattern = fis.readLine()) != null) {
patternsToSkip.add(pattern);
}
} catch (IOException ioe) {
System.err.println("Caught exception while parsing the cached file '"
+ StringUtils.stringifyException(ioe));
}
}
```

```
@Override
public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
String line = (caseSensitive) ?
value.toString() : value.toString().toLowerCase();
for (String pattern : patternsToSkip) {
line = line.replaceAll(pattern, "");
}
StringTokenizer itr = new StringTokenizer(line);
while (itr.hasMoreTokens()) {
word.set(itr.nextToken());
context.write(word, one);
Counter counter = context.getCounter(CountersEnum.class.getName(),
CountersEnum.INPUT_WORDS.toString());
counter.increment(1);
}
}
}

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

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

public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();

conf.set("fs.oss.accessKeyId", "${accessKeyId}");
conf.set("fs.oss.accessKeySecret", "${accessKeySecret}");
conf.set("fs.oss.endpoint","${endpoint}");

GenericOptionsParser optionParser = new GenericOptionsParser(conf, args);
String[] remainingArgs = optionParser.getRemainingArgs();
if (!(remainingArgs.length != 2 || remainingArgs.length != 4)) {
System.err.println("Usage: wordcount <in> <out> [-skip skipPatternFile]");
System.exit(2);
}
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount2.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

```
List<String> otherArgs = new ArrayList<String>();
for (int i=0; i < remainingArgs.length; ++i) {
if ("-skip".equals(remainingArgs[i])) {
job.addCacheFile(new Path(EMapReduceOSSUtil.buildOSSCompleteUri(remainingArgs[++i],
conf)).toUri());
job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
} else {
otherArgs.add(remainingArgs[i]);
}
}
FileInputFormat.addInputPath(job, new
Path(EMapReduceOSSUtil.buildOSSCompleteUri(otherArgs.get(0), conf)));
FileOutputFormat.setOutputPath(job, new
Path(EMapReduceOSSUtil.buildOSSCompleteUri(otherArgs.get(1), conf)));

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

其中的 EMapReduceOSSUtil 类代码请参见如下示例，放在和 WordCount2 相同目录：

```
package com.aliyun.emr.hadoop.examples;

import org.apache.hadoop.conf.Configuration;

public class EMapReduceOSSUtil {

private static String SCHEMA = "oss://";
private static String AKSEP = ":";
private static String BKTSEP = "@";
private static String EPSEP = ".";
private static String HTTP_HEADER = "http://";

/**
* complete OSS uri
* convert uri like: oss://bucket/path to oss://accessKeyId:accessKeySecret@bucket.endpoint/path
* ossref do not need this
*
* @param oriUri original OSS uri
*/
public static String buildOSSCompleteUri(String oriUri, String akId, String akSecret, String endpoint) {
if (akId == null) {
System.err.println("miss accessKeyId");
return oriUri;
}
if (akSecret == null) {
System.err.println("miss accessKeySecret");
return oriUri;
}
if (endpoint == null) {
System.err.println("miss endpoint");
return oriUri;
}

int index = oriUri.indexOf(SCHEMA);
```

```
if (index == -1 || index != 0) {
return oriUri;
}

int bucketIndex = index + SCHEMA.length();
int pathIndex = oriUri.indexOf("/", bucketIndex);
String bucket = null;
if (pathIndex == -1) {
bucket = oriUri.substring(bucketIndex);
} else {
bucket = oriUri.substring(bucketIndex, pathIndex);
}

StringBuilder retUri = new StringBuilder();
retUri.append(SCHEMA)
.append(akId)
.append(AKSEP)
.append(akSecret)
.append(BKTSEP)
.append(bucket)
.append(EPSEP)
.append(stripHttp(endpoint));

if (pathIndex > 0) {
retUri.append(oriUri.substring(pathIndex));
}

return retUri.toString();
}

public static String buildOSSCompleteUri(String oriUri, Configuration conf) {
return buildOSSCompleteUri(oriUri, conf.get("fs.oss.accessKeyId"), conf.get("fs.oss.accessKeySecret"),
conf.get("fs.oss.endpoint"));
}

private static String stripHttp(String endpoint) {
if (endpoint.startsWith(HTTP_HEADER)) {
return endpoint.substring(HTTP_HEADER.length());
}
return endpoint;
}
}
```

编译并打包上传。在工程的目录下，执行如下命令：

```
mvn clean package -DskipTests
```

您即可在工程目录的 target 目录下看到一个 wordcountv2-1.0-SNAPSHOT.jar，这个就是作业 jar 包了。请您将这个 jar 包上传到 OSS 中。

创建作业。在 E-MapReduce 中新建一个作业，请使用类似如下的参数配置：

```
 jar ossref://yourBucket/yourPath/wordcountv2-1.0-SNAPSHOT.jar
com.aliyun.emr.hadoop.examples.WordCount2 -Dwordcount.case.sensitive=true
oss://yourBucket/yourPath/The_Sorrows_of_Young_Werther.txt oss://yourBucket/yourPath/output -skip
oss://yourBucket/yourPath/patterns.txt
```

这里的 yourBucket 是您的一个 OSS bucket，yourPath 是这个 bucket 上的一个路径，需要您按照实际情况填写。请您将oss://yourBucket/yourPath/The_Sorrows_of_Young_Werther.txt和oss://yourBucket/yourPath/patterns.txt这两个用来处理相关资源的文件下载下来并放到您的 OSS 上。作业需要资源可以从下面下载，然后放到您的 OSS 对应目录下。

资源下载：The_Sorrows_of_Young_Werther.txtpatterns.txt

创建执行计划并运行。在 E-MapReduce 中创建执行计划，关联这个作业并运行。

在 Hive 中使用 OSS

要在 Hive 中读写 OSS，需要在使用 OSS 的 URI 时加上 AccessKeyId、AccessKeySecret 以及 endpoint。如下示例介绍了如何创建一个 external 的表：

```
CREATE EXTERNAL TABLE eusers (
userid INT)
LOCATION 'oss://emr/users';
```

为了保证能正确访问 OSS，这个时候需要修改这个 OSS URI 为：

```
CREATE EXTERNAL TABLE eusers (
userid INT)
LOCATION 'oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/users';
```

参数说明：

${accessKeyId}：您账号的 accessKeyId。

${accessKeySecret}：该 accessKeyId 对应的密钥。

${endpoint}：访问 OSS 使用的网络，由您集群所在的 region 决定，对应的 OSS 也需要是在集群对应的 region。

具体的值参考 OSS Endpoint

使用Tez作为计算引擎

从E-MapReduce产品版本2.1.0+开始，引入了Tez，Tez是一个用来优化处理复杂DAG调度任务的计算框架。在很多场景下可以有效的提高Hive作业的运行速度。

用户在作业中，可以通过设置Tez作为执行引擎来优化作业，如下：

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

示例 1

请参见如下步骤：

编写如下脚本，保存为 hiveSample1.sql，并上传到 OSS 上。

```
 USE DEFAULT;
set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
set hive.stats.autogather=false;

DROP TABLE emrusers;
CREATE EXTERNAL TABLE emrusers (
userid INT,
movieid INT,
rating INT,
unixtime STRING )
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION 'oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/yourpath';

SELECT COUNT(*) FROM emrusers;

SELECT * from emrusers limit 100;

SELECT movieid,count(userid) as usercount from emrusers group by movieid order by usercount desc limit
50;
```

测试用数据资源。您可通过下面的地址下载 Hive 作业需要的资源，然后将其放到您 OSS 对应的目录下。

资源下载：公共测试数据

创建作业。在 E-MapReduce 中新建一个作业,使用类似如下的参数配置：

```
  -f ossref://${bucket}/yourpath/hiveSample1.sql
```

这里的 ${bucket} 是您的一个 OSS bucket，yourPath 是这个 bucket 上的一个路径，需要您填写实际保存 Hive 脚本的位置。

创建执行计划并运行。创建一个执行计划，您可以关联一个已有的集群，也可以自动按需创建一个

，然后关联上这个作业。请将作业保存为“手动运行”，回到界面上就可以点击“立即运行”来运行作业了。

示例 2

以 HiBench 中的 scan 为例，若输入输出来自 OSS，则需进行如下过程运行 Hive 作业，代码修改源于 AccessKeyId、AccessKeySecret 以及存储路径的设置。请注意 OSS 路径的设置形式为 oss://${accesskeyId}:${accessKeySecret}@${bucket}.${endpoint}/object/path。

请参见如下操作步骤：

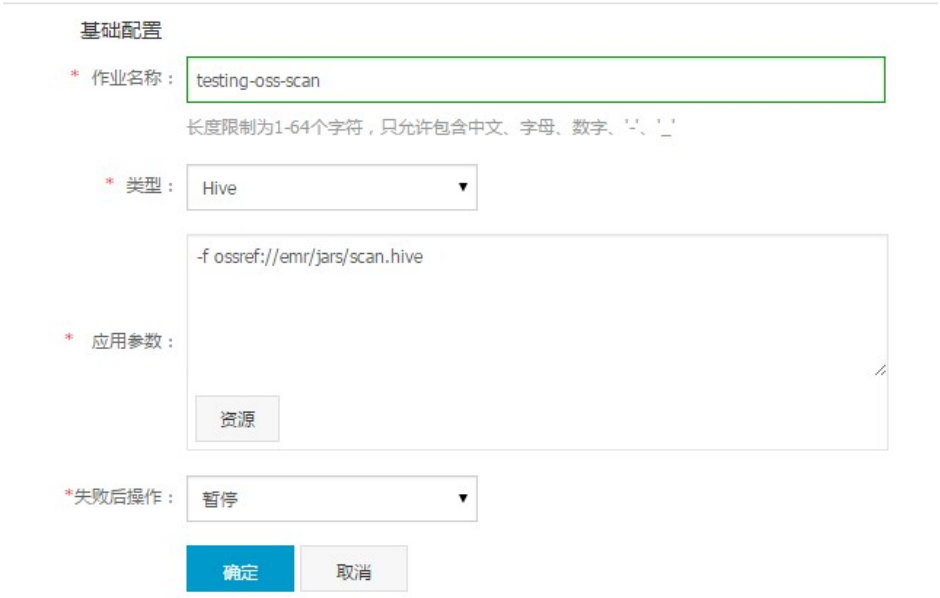
编写如下脚本。

```
USE DEFAULT;
set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
set mapreduce.job.maps=12;
set mapreduce.job.reduces=6;
set hive.stats.autogather=false;

DROP TABLE uservisits;
CREATE EXTERNAL TABLE uservisits (sourceIP STRING,destURL STRING,visitDate STRING,adRevenue
DOUBLE,userAgent STRING,countryCode STRING,languageCode STRING,searchWord STRING,duration INT
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS SEQUENCEFILE LOCATION
'oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/sample-data/hive/Scan/Input/uservisits';
```

准备测试数据。您可通过下面的地址下载作业需要的资源，然后将其放到您 OSS 对应的目录下。

资源下载：uservisits

创建作业。将步骤 1 中编写的脚本存储到 OSS 中，假设存储路径为oss://emr/jars/scan.hive，在 E-MapReduce 中创建如下作业：



创建执行计划并运行。创建一个执行计划，您可以关联一个已有的集群，也可以自动按需创建一个，然后关联上这个作业。请将作为保存为“手动运行”，回到界面上就可以点击“立即运行”来运行作业了。

在 Pig 中使用 OSS

在使用 OSS 路径的时候，请使用类似如下的形式

oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/${path}

参数说明：

${accessKeyId}：您账号的 AccessKeyId。

${accessKeySecret}：该 AccessKeyId 对应的密钥。

${bucket}： 该 AccessKeyId 对应的 bucket。

${endpoint}：访问 OSS 使用的网络，由您集群所在的 region 决定，对应的 OSS 也需要是在集群对应的 region。

${path}：bucket 中的路径。

具体的值请参考 OSS Endpoint

以 Pig 中带的 script1-hadoop.pig 为例进行说明，将 Pig 中的 tutorial.jar 和 excite.log.bz2 上传到 OSS 中，假设上传路径分别为oss://emr/jars/tutorial.jar和oss://emr/data/excite.log.bz2。

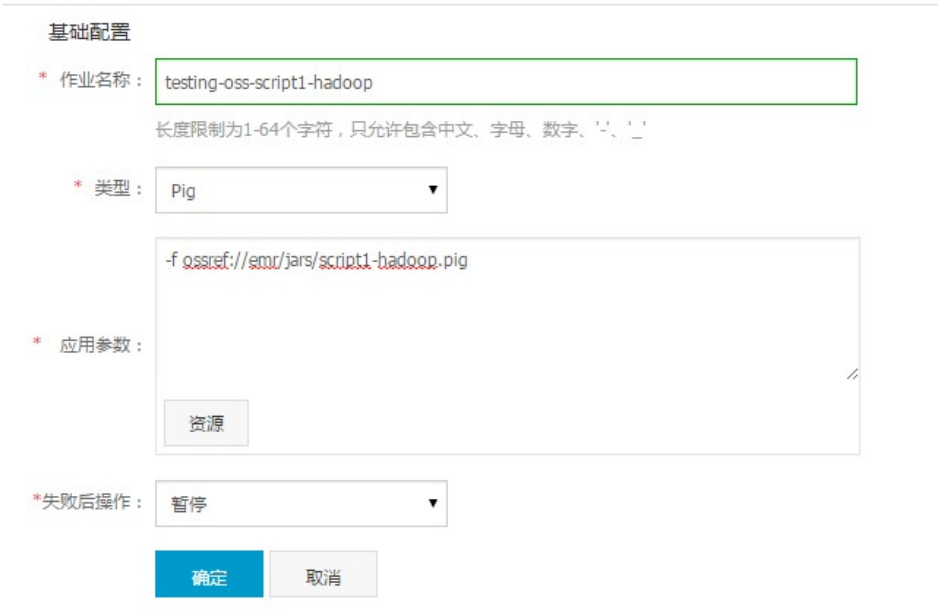
请参见如下操作步骤：

1. 编写脚本。将脚本中的 jar 文件路径和输入输出路径做了修改，如下所示。注意 OSS 路径设置形式为 oss://${accesskeyId}:${accessKeySecret}@${bucket}.${endpoint}/object/path。

```
/*
* Licensed to the Apache Software Foundation (ASF) under one
* or more contributor license agreements. See the NOTICE file
* distributed with this work for additional information
* regarding copyright ownership. The ASF licenses this file
* to you under the Apache License, Version 2.0 (the
* "License"); you may not use this file except in compliance
* with the License. You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

-- Query Phrase Popularity (Hadoop cluster)

-- This script processes a search query log file from the Excite search engine and finds search phrases that occur with particular high frequency during certain times of the day.

-- Register the tutorial JAR file so that the included UDFs can be called in the script.
REGISTER oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/data/tutorial.jar;

-- Use the PigStorage function to load the excite log file into the raw bag as an array of records.
-- Input: (user,time,query)
raw = LOAD 'oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/data/excite.log.bz2' USING PigStorage('\t') AS (user, time, query);

-- Call the NonURLDetector UDF to remove records if the query field is empty or a URL.
clean1 = FILTER raw BY org.apache.pig.tutorial.NonURLDetector(query);

-- Call the ToLower UDF to change the query field to lowercase.
clean2 = FOREACH clean1 GENERATE user, time, org.apache.pig.tutorial.ToLower(query) as query;

-- Because the log file only contains queries for a single day, we are only interested in the hour.
-- The excite query log timestamp format is YYMMDDHHMMSS.
-- Call the ExtractHour UDF to extract the hour (HH) from the time field.
houred = FOREACH clean2 GENERATE user, org.apache.pig.tutorial.ExtractHour(time) as hour, query;

-- Call the NGramGenerator UDF to compose the n-grams of the query.
ngramed1 = FOREACH houred GENERATE user, hour, flatten(org.apache.pig.tutorial.NGramGenerator(query)) as ngram;

-- Use the DISTINCT command to get the unique n-grams for all records.
ngramed2 = DISTINCT ngramed1;

-- Use the GROUP command to group records by n-gram and hour.
hour_frequency1 = GROUP ngramed2 BY (ngram, hour);
```

```
-- Use the COUNT function to get the count (occurrences) of each n-gram.
hour_frequency2 = FOREACH hour_frequency1 GENERATE flatten($0), COUNT($1) as count;

-- Use the GROUP command to group records by n-gram only.
-- Each group now corresponds to a distinct n-gram and has the count for each hour.
uniq_frequency1 = GROUP hour_frequency2 BY group::ngram;
-- For each group, identify the hour in which this n-gram is used with a particularly high frequency.
-- Call the ScoreGenerator UDF to calculate a "popularity" score for the n-gram.
uniq_frequency2 = FOREACH uniq_frequency1 GENERATE flatten($0),
flatten(org.apache.pig.tutorial.ScoreGenerator($1));

-- Use the FOREACH-GENERATE command to assign names to the fields.
uniq_frequency3 = FOREACH uniq_frequency2 GENERATE $1 as hour, $0 as ngram, $2 as score, $3 as count, $4 as
mean;

-- Use the FILTER command to move all records with a score less than or equal to 2.0.
filtered_uniq_frequency = FILTER uniq_frequency3 BY score > 2.0;

-- Use the ORDER command to sort the remaining records by hour and score.
ordered_uniq_frequency = ORDER filtered_uniq_frequency BY hour, score;

-- Use the PigStorage function to store the results.
-- Output: (hour, n-gram, score, count, average_counts_among_all_hours)
STORE ordered_uniq_frequency INTO
'oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/data/script1-hadoop-results' USING PigStorage();
```

创建作业。将步骤 1 中编写的脚本存放到 OSS 上，假设存储路径为 oss://emr/jars/script1-hadoop.pig，在 E-MapReduce 作业中创建如下作业：



创建执行计划并运行。在 E-MapReduce 执行计划中创建执行计划，将上一步创建好的 Pig 作业添加到执行计划中，策略请选择“立即执行”，这样 script1-hadoop 作业就会在选定集群中运行起来了。

python 写hadoop streaming作业

mapper代码如下

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
line = line.strip()
words = line.split()
for word in words:
print '%s\t%s' % (word, 1)
```

reducer代码如下

```
#!/usr/bin/env python
from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
line = line.strip()

word, count = line.split('\t', 1)

try:
count = int(count)
except ValueError:
continue

if current_word == word:
current_count += count
else:
if current_word:
print '%s\t%s' % (current_word, current_count)
current_count = count
current_word = word

if current_word == word:
print '%s\t%s' % (current_word, current_count)
```

假设mapper代码保存在/home/hadoop/mapper.py， reducer代码保存在/home/hadoop/reducer.py ，输入路径为hdfs文件系统的/tmp/input，输出路径为hdfs文件系统的/tmp/output。则在E-MapReduce集群上提交下面的hadoop命令

hadoop jar /usr/lib/hadoop-current/share/hadoop/tools/lib/hadoop-streaming-*.jar -file

/home/hadoop/mapper.py -mapper mapper.py -file /home/hadoop/reducer.py -reducer reducer.py -input /tmp/hosts -output /tmp/output

Hive + TableStore

Hive接入TableStore

- 准备一张数据表

创建一张表pet，其中name为主键。

name	owner	species	sex	birth	death
Fluffy	Harold	cat	f	1993-02-04	
Claws	Gwen	cat	m	1994-03-17	
Buffy	Harold	dog	f	1989-05-13	
Fang	Benny	dog	m	1990-08-27	
Bowser	Diane	dog	m	1979-08-31	1995-07-29
Chirpy	Gwen	bird	f	1998-09-11	
Whistler	Gwen	bird		1997-12-09	
Slim	Benny	snake	m	1996-04-29	
Puffball	Diane	hamster	f	1999-03-30	

- 下面这个例子示例了如何在Hive中处理TableStore中的数据。

1.命令行

```
$ HADOOP_HOME=YourHadoopDir HADOOP_CLASSPATH=emr-tablestore-<version>.jar:tablestore-4.1.0-jar-with-dependencies.jar:joda-time-2.9.4.jar bin/hive
```

2.创建一张外表

```
CREATE EXTERNAL TABLE pet
(name STRING, owner STRING, species STRING, sex STRING, birth STRING, death STRING)
STORED BY 'com.aliyun.openservices.tablestore.hive.TableStoreStorageHandler'
WITH SERDEPROPERTIES(
"tablestore.columns.mapping"="name,owner,species,sex,birth,death")
TBLPROPERTIES (
"tablestore.endpoint"="YourEndpoint",
"tablestore.access_key_id"="YourAccessKeyId",
"tablestore.access_key_secret"="YourAccessKeySecret",
```

```
"tablestore.table.name"="pet");
```

3.向外表中插入数据

```
INSERT INTO pet VALUES("Fluffy", "Harold", "cat", "f", "1993-02-04", null);
INSERT INTO pet VALUES("Claws", "Gwen", "cat", "m", "1994-03-17", null);
INSERT INTO pet VALUES("Buffy", "Harold", "dog", "f", "1989-05-13", null);
INSERT INTO pet VALUES("Fang", "Benny", "dog", "m", "1990-08-27", null);
INSERT INTO pet VALUES("Bowser", "Diane", "dog", "m", "1979-08-31", "1995-07-29");
INSERT INTO pet VALUES("Chirpy", "Gwen", "bird", "f", "1998-09-11", null);
INSERT INTO pet VALUES("Whistler", "Gwen", "bird", null, "1997-12-09", null);
INSERT INTO pet VALUES("Slim", "Benny", "snake", "m", "1996-04-29", null);
INSERT INTO pet VALUES("Puffball", "Diane", "hamster", "f", "1999-03-30", null);
```

4.查询数据

```
> SELECT * FROM pet;
Bowser Diane dog m 1979-08-31 1995-07-29
Buffy Harold dog f 1989-05-13 NULL
Chirpy Gwen bird f 1998-09-11 NULL
Claws Gwen cat m 1994-03-17 NULL
Fang Benny dog m 1990-08-27 NULL
Fluffy Harold cat f 1993-02-04 NULL
Puffball Diane hamster f 1999-03-30 NULL
Slim Benny snake m 1996-04-29 NULL
Whistler Gwen bird NULL 1997-12-09 NULL
> SELECT * FROM pet WHERE birth > "1995-01-01";
Chirpy Gwen bird f 1998-09-11 NULL
Puffball Diane hamster f 1999-03-30 NULL
Slim Benny snake m 1996-04-29 NULL
Whistler Gwen bird NULL 1997-12-09 NULL
```

数据类型转换

	TINYINT	SMALLINT	INT	BIGINT	FLOAT	DOUBLE	BOOLEAN	STRING	BINARY
INTEGER	支持,损失精度	支持,损失精度	支持,损失精度	支持	支持,损失精度	支持,损失精度			
DOUBLE	支持,损失精度	支持,损失精度	支持,损失精度	支持,损失精度	支持,损失精度	支持			
BOOLEAN							支持		
STRING								支持	
BINARY									支持

附录

完整示例代码请参考：

- Hive+TableStore

MR + TableStore

MR接入TableStore

- 准备一张数据表

创建一张表pet，其中name为主键。

name	owner	species	sex	birth	death
Fluffy	Harold	cat	f	1993-02-04	
Claws	Gwen	cat	m	1994-03-17	
Buffy	Harold	dog	f	1989-05-13	
Fang	Benny	dog	m	1990-08-27	
Bowser	Diane	dog	m	1979-08-31	1995-07-29
Chirpy	Gwen	bird	f	1998-09-11	
Whistler	Gwen	bird		1997-12-09	
Slim	Benny	snake	m	1996-04-29	
Puffball	Diane	hamster	f	1999-03-30	

- 下面这个例子示例了如何在MR中消费TableStore中的数据。

```
public class RowCounter {
public static class RowCounterMapper
extends Mapper<PrimaryKeyWritable, RowWritable, Text, LongWritable> {
private final static Text agg = new Text("TOTAL");
private final static LongWritable one = new LongWritable(1);

@Override public void map(PrimaryKeyWritable key, RowWritable value,
Context context) throws IOException, InterruptedException {
context.write(agg, one);
}
}

public static class IntSumReducer
```

```
extends Reducer<Text,LongWritable,Text,LongWritable> {

@Override public void reduce(Text key, Iterable<LongWritable> values,
Context context) throws IOException, InterruptedException {
long sum = 0;
for (LongWritable val : values) {
sum += val.get();
}
context.write(key, new LongWritable(sum));
}
}

private static RangeRowQueryCriteria fetchCriteria() {
RangeRowQueryCriteria res = new RangeRowQueryCriteria("pet");
res.setMaxVersions(1);
List<PrimaryKeyColumn> lower = new ArrayList<PrimaryKeyColumn>();
List<PrimaryKeyColumn> upper = new ArrayList<PrimaryKeyColumn>();
lower.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MIN));
upper.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MAX));
res.setInclusiveStartPrimaryKey(new PrimaryKey(lower));
res.setExclusiveEndPrimaryKey(new PrimaryKey(upper));
return res;
}

public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
job.setJarByClass(RowCounter.class);
job.setMapperClass(RowCounterMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
job.setInputFormatClass(TableStoreInputFormat.class);

TableStore.setCredential(job, accessKeyId, accessKeySecret, securityToken);
TableStore.setEndpoint(job, endpoint, instance);
TableStoreInputFormat.addCriteria(job, fetchCriteria());
FileOutputFormat.setOutputPath(job, new Path(outputPath));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

- 下面这个例子示例了如何在MR中将数据写到TableStore。

```
public static class OwnerMapper
extends Mapper<PrimaryKeyWritable, RowWritable, Text, MapWritable> {
@Override public void map(PrimaryKeyWritable key, RowWritable row,
Context context) throws IOException, InterruptedException {
PrimaryKeyColumn pet = key.getPrimaryKey().getPrimaryKeyColumn("name");
Column owner = row.getRow().getLatestColumn("owner");
Column species = row.getRow().getLatestColumn("species");
MapWritable m = new MapWritable();
m.put(new Text(pet.getValue().asString()),
new Text(species.getValue().asString()));
context.write(new Text(owner.getValue().asString()), m);
```

```
}
}

public static class IntoTableReducer
extends Reducer<Text,MapWritable,Text,BatchWriteWritable> {

@Override public void reduce(Text owner, Iterable<MapWritable> pets,
Context context) throws IOException, InterruptedException {
List<PrimaryKeyColumn> pkeyCols = new ArrayList<PrimaryKeyColumn>();
pkeyCols.add(new PrimaryKeyColumn("owner",
PrimaryKeyValue.fromString(owner.toString())));
PrimaryKey pkey = new PrimaryKey(pkeyCols);
List<Column> attrs = new ArrayList<Column>();
for(MapWritable petMap: pets) {
for(Map.Entry<Writable, Writable> pet: petMap.entrySet()) {
Text name = (Text) pet.getKey();
Text species = (Text) pet.getValue();
attrs.add(new Column(name.toString(),
ColumnValue.fromString(species.toString())));
}
}
RowPutChange putRow = new RowPutChange(outputTable, pkey)
.addColumns(attrs);
BatchWriteWritable batch = new BatchWriteWritable();
batch.addRowChange(putRow);
context.write(owner, batch);
}
}

public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, TableStoreOutputFormatExample.class.getName());
job.setMapperClass(OwnerMapper.class);
job.setReducerClass(IntoTableReducer.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(MapWritable.class);
job.setInputFormatClass(TableStoreInputFormat.class);
job.setOutputFormatClass(TableStoreOutputFormat.class);

TableStore.setCredential(job, accessKeyId, accessKeySecret, securityToken);
TableStore.setEndpoint(job, endpoint, instance);
TableStoreInputFormat.addCriteria(job, ...);
TableStoreOutputFormat.setOutputTable(job, outputTable);
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

附录

完整示例代码请参考：

- MR读TableStore
- MR写TableStore

为了更好地使用 HBase，在创建集群过程中，推荐您使用如下配置:

公网状态选择打开。

可用区选择为访问 HBase 的应用服务器所在的可用区，请勿选择随机分配。

硬件节点数请选择 4 个及以上，其包含了 Master 和 Slave 节点，E-MapReduce 会在这些节点上创建 namenode、datanode、journalnode、hmaster、regionserver 和 zookeeper 角色。

服务器配置推荐选择 4 核 16G / 8 核 32G 这两款机型，过低的配置可能导致 HBase 集群无法稳定运行。

数据盘类型建议选择 SSD 云盘，会有更好的成本性价比。对于访问少，存储量大的业务，可以选择普通云盘。

数据容量请按实际需求配置。

HBase 集群支持扩容。

HBase 配置

创建 HBase 集群的时候，在创建页面可以利用软件配置功能，结合使用场景，对 HBase 的默认参数配置做一些优化修改，如下所示：

```
 {
"configurations": [
{
"classification": "hbase-site",
"properties": {
"hbase.hregion.memstore.flush.size": "268435456",
"hbase.regionserver.global.memstore.size": "0.5",
"hbase.regionserver.global.memstore.lowerLimit": "0.6"
}
}
]
}
```

HBase 集群一些默认的配置如下所示：

key	value
zookeeper.session.timeout	180000
hbase.regionserver.global.memstore.size	0.35

hbase.regionserver.global.memstore.lowerLimit	0.3
hbase.hregion.memstore.flush.size	128MB

访问 HBase

注意：

由于网络性能的考虑，通过 E-MapReduce 创建的 HBase 集群，我们只建议从同个可用区的 ECS 上发起访问。

访问 HBase 集群的 ECS 必须和 HBase 集群处于同一个安全组内，否则无法访问，所以在 EMR 中创建 Hadoop/Spark/Hive 集群时，如果它们需要访问 HBase，则在创建集群的时候一定要选择和 HBase 集群相同的安全组。

通过 E-MapReduce 控制台创建完 HBase 集群以后，用户可以开始使用 HBase 存储服务。其操作步骤如下：

获取 Master IP 和集群 ZK 地址。通过 E-MapReduce 控制台集群详情页面，用户可以查看集群的 Master 节点 IP 和 ZK 访问地址（内网IP），如下图所示：

对于开通了公网 IP 的 Master 节点，用户可以参考如何登录 master 节点，浏览 HMaster 的 WEB UI(localhost:16010)。

SSH 连接到集群主节点，使用 HBase Shell。用户可以直接通过 SSH 连接到集群的 Master 节点，切换到 HDFS 用户，通过 HBase Shell 访问集群（关于 HBase Shell 的更多介绍，请参考 Apache HBase 官网）。

```
[root@emr-header-1 ~]# su hdfs
[hadoop@emr-header-1 root]$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.1.1, r374488, Fri Aug 21 09:18:22 CST 2015
```

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

从其他 ECS 节点(同一个安全组内)，使用 HBase Shell 访问集群。从 Apache HBase 的官网下载 HBase-1.x 版本的资源包（下载链接），解压后，修改 conf/hbase-site.xml，添加集群的 ZK 地址，如下所示：

```
<configuration>
<property>
<name>hbase.zookeeper.quorum</name>
<value>$ZK_IP1,$ZK_IP2,$ZK_IP3</value>
</property>
</configuration>
```

然后就可以通过命令 bin/hbase shell 访问集群了。

若 ECS 是通过 EMR 创建的,则只需修改 /etc/emr/hbase-conf/hbase-site.xml,无需下载 HBase-1.x 版本的资源包。

通过 API 访问 HBase 集群，引入 Maven 依赖。

```
<groupId>org.apache.hbase</groupId>
<artifactId>hbase-client</artifactId>
<version>1.1.1</version>
```

配置正确的 ZK 地址，连接集群。

```
Configuration config = HBaseConfiguration.create();
config.set(HConstants.ZOOKEEPER_QUORUM,"$ZK_IP1,$ZK_IP2,$ZK_IP3");
Connection connection = ConnectionFactory.createConnection(config);
try {
Table table = connection.getTable(TableName.valueOf("myLittleHBaseTable"));
try {
//Do table operation
}finally {
if (table != null) table.close();
}
} finally {
connection.close();
}
```

更多开发介绍，请参考 Apache HBase 官网。

示例

前提条件

访问 Hbase 集群的 ECS 必须和 HBase 集群处于同一个安全组内。

Spark 访问 Hbase

请参照 spark-hbase-connector。

Hadoop 访问 Hbase

请参照 HBase MapReduce Examples。

Hive 访问 Hbase

EMR 1.2.0 及以上主版本的集群中的 Hive 才能访问 Hbase 集群。其步骤如下：

登录 Hive 集群，修改 hosts，增加如下一行:

```
$zk_ip emr-cluster //$zk_i p为 Hbase 集群的 zk 节点 IP
```

具体 Hive 操作请参照 Hive HBase Integration。

E-MapReduce的HBase集群，可以通过HBase内置的快照(snapshot)功能对HBase的表进行备份，并将备份的数据导出到阿里云存储服务OSS中,示例如下：

1.创建HBase集群

详见集群创建文档

2.创建table

```
>create 'test','cf'
```

3.添加数据

```
> put 'test','a','cf:c1',1
> put 'test','a','cf:c2',2
> put 'test','b','cf:c1',3
> put 'test','b','cf:c2',4
> put 'test','c','cf:c1',5
> put 'test','c','cf:c2',6
```

4.创建快照

```
hbase snapshot create -n test_snapshot -t test
```

查看快照

```
>list_snapshots
SNAPSHOT TABLE + CREATION TIME
test_snapshot test (Sun Sep 04 20:31:00 +0800 2016)
1 row(s) in 0.2080 seconds
```

5. 导出到OSS

```
hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot test_snapshot -copy-to
oss://$accessKeyId:$accessKeySecret@$bucket.oss-cn-hangzhou-internal.aliyuncs.com/hbase/snapshot/test
```

备注: OSS使用内网Endpoint

6.创建另一个HBase集群

7.OSS导出快照

```
hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot test_snapshot -copy-from
oss://$accessKeyId:$accessKeySecret@$bucket.oss-cn-hangzhou-internal.aliyuncs.com/hbase/snapshot/test -copy-
to /hbase/
```

8.从快照恢复数据

```
>restore _snapshot 'test_snapshot'
```

```
>scan 'test'
ROW COLUMN+CELL
a column=cf:c1, timestamp=1472992081375, value=1
a column=cf:c2, timestamp=1472992090434, value=2
b column=cf:c1, timestamp=1472992104339, value=3
b column=cf:c2, timestamp=1472992099611, value=4
c column=cf:c1, timestamp=1472992112657, value=5
c column=cf:c2, timestamp=1472992118964, value=6
3 row(s) in 0.0540 seconds
```

9.从快照创建新表

```
>clone_snapshot 'test_snapshot','test_2'
```

```
>scan 'test_2'
ROW COLUMN+CELL
```

```
a column=cf:c1, timestamp=1472992081375, value=1
a column=cf:c2, timestamp=1472992090434, value=2
b column=cf:c1, timestamp=1472992104339, value=3
b column=cf:c2, timestamp=1472992099611, value=4
c column=cf:c1, timestamp=1472992112657, value=5
c column=cf:c2, timestamp=1472992118964, value=6
3 row(s) in 0.0540 seconds
```

数据传输软件

Sqoop 是一款用来在不同数据存储软件之间进行数据传输的开源软件，它支持多种类型的数据储存软件。

安装 Sqoop

注意：目前，E-MapReduce 从版本 1.3 开始都会默认支持 Sqoop 组件，您无需再自行安装，可以跳过本节。

若您使用的 E-MapReduce 的版本低于 1.3，则还没有集成 Sqoop，如果需要使用请参考如下的安装方式。

从官网下载 Sqoop 1.4.6 版本（**点击下载**）。若下载的 sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz 无法打开，也可以使用镜像地址 http://mirror.bit.edu.cn/apache/sqoop/1.4.6/sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz进行下载，请执行如下命令：

```
wget http://mirror.bit.edu.cn/apache/sqoop/1.4.6/sqoop-1.4.6.bin__ha doop-2.0.4-alpha.tar.gz
```

执行如下命令，将下载下来的sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz解压到 Master 节点上：

```
tar zxf sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz
```

要从 Mysql 导数据需要安装 Mysql driver。请从官网下载最新版本（**点击下载**），或执行如下命令进行下载（此处以版本 5.1.38 为例）：

```
wget https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.38.tar.gz
```

解压以后把 jar 包放到 Sqoop 目录下的 lib 目录下。

数据传输

常见的使用场景如下所示：

Mysql -> HDFS

HDFS -> Mysql

Mysql -> Hive

Hive -> Mysql

使用 SQL 作为导入条件

注意：在执行如下的命令前，请先切换你的用户为 Hadoop。

```
su hadoop
```

从 Mysql 到 HDFS

在集群的 Master 节点上执行如下命令：

```
sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --
table <tablename> --target-dir <hdfs-dir>
```

参数说明：

dburi：数据库的访问连接，例如： jdbc:mysql://192.168.1.124:3306/ 如果您的访问连接中含有参数,那么请用单引号将整个连接包裹住，例如’ jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true’

dbname：数据库的名字，例如：user。

username：数据库登录用户名。

password：用户对应的密码。

tablename：Mysql 表的名字。

hdfs-dir：HDFS 的写入目录，例如：/user/hive/result。

更加详细的参数使用请参考 Sqoop Import。

从 HDFS 到 Mysql

需要先创建好对应 HDFS 中的数据结构的 Mysql 表，然后在集群的 Master 节点上执行如下命令，指定要导的

数据文件的路径。

```
sqoop export --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --
table <tablename> --export-dir <hdfs-dir>
```

参数说明：

dburi：数据库的访问连接，例如： jdbc:mysql://192.168.1.124:3306/。如果您的访问连接中含有参数,那么请用单引号将整个连接包裹住，例如’ jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true’

dbname：数据库的名字，例如：user。

username：数据库登录用户名。

password：用户对应的密码。

tablename：Mysql 的表的名字。

hdfs-dir：要导到 Mysql 去的 HDFS 的数据目录，例如：/user/hive/result.

更加详细的参数使用请参考 Sqoop Export。

从 Mysql 到 Hive

导入的同时也新建一个 Hive 表，执行如下命令：

```
sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --
table <tablename> --fields-terminated-by "\t" --lines-terminated-by "\n" --hive-import --target-dir <hdfs-dir> --
hive-table <hive-tablename>
```

参数说明：

dburi：数据库的访问连接，例如： jdbc:mysql://192.168.1.124:3306/ 如果您的访问连接中含有参数,那么请用单引号将整个连接包裹住，例如’ jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true’

dbname：数据库的名字，例如：user。

username：据库登录用户名。

password：用户对应的密码。

tablename：Mysql 的表的名字。

hdfs-dir：要导到 Mysql 去的 HDFS 的数据目录，例如：/user/hive/result。

hive-tablename：对应的 Hive 中的表名，可以是 xxx.yyy。

更加详细的参数使用请参考 Sqoop Import。

从 Hive 到 Mysql

请参考上面的从 HDFS 到 Mysql的命令，只需要指定 Hive 表对应的 HDFS 路径就可以了。

从 Mysql 到 OSS

类似从 Mysql 到 HDFS，只是 —target-dir 不同。在集群的 Master 节点上执行如下命令：

注意1： OSS 地址中的 host 有内网地址、外网地址和 VPC 网络地址之分。如果用经典网络，需要指定内网地址，杭州是 oss-cn-hangzhou-**internal**.aliyuncs.com，VPC 要指定 VPC 内网，杭州是 **vpc100**-oss-cn-hangzhou.aliyuncs.com。

注意2： 目前同步到OSS不支持—delete-target-dir，用这个参数会报错Wrong FS。如果要覆盖以前目录的数据 ，可以在调用sqoop前，用hadoop fs -rm -r osspath先把原来的oss目录删了，

```
sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --
table <tablename> --target-dir <oss-dir>
```

参数说明：

dburi：数据库的访问连接，例如： jdbc:mysql://192.168.1.124:3306/ 如果您的访问连接中含有参数,那么请用单引号将整个连接包裹住，例如’ jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true’

dbname：数据库的名字，例如：user。

username：数据库登录用户名。

password：用户对应的密码。

tablename：Mysql 表的名字。

oss-dir：OSS 的写入目录，例如：oss://<accessid>:<accesskey>@<bucketname>.oss-cn-hangzhou-internal.aliyuncs.com/result。

更加详细的参数使用请参考 Sqoop Import。

6. 从Oss到Mysql

类似mysql到hdfs，只是—export-dir不同。需要创建好对应oss中的数据结构的Mysql表

然后在集群的Master节点上执行如下：指定要导的数据文件的路径

```
sqoop export --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --
table <tablename> --export-dir <oss-dir>
```

dburi:数据库的访问连接，例如： jdbc:mysql://192.168.1.124:3306/ 如果您的访问连接中含有参数,那么请用单引号将整个连接包裹住，例如’ jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true’

dbname:数据库的名字，例如：user

username:数据库登录用户名

password:用户对应的密码

tablename:Mysql的表的名字

oss-dir:oss的写入目录，例如：oss://<accessid>:<accesskey>@<bucketname>.oss-cn-hangzhou-internal.aliyuncs.com/result

注意： oss地址host有内网地址，外网地址，vpc网络地址之分。如果 用经典网络，需要指定内网地址，杭州是oss-cn-hangzhou-**internal**.aliyuncs.com,vpc要指定vpc内网，杭州是**vpc100**-oss-cn-hangzhou.aliyuncs.com

更加详细的参数使用请参考，Sqoop Export

使用 SQL 作为导入条件

除了指定 Mysql 的全表导入，还可以写 SQL 来指定导入的数据，如下所示：

```
sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --query <query-sql> --split-by <sp-column> --hive-import --hive-table <hive-tablename> --target-dir <hdfs-dir>
```

参数说明：

dburi：数据库的访问连接，例如： jdbc:mysql://192.168.1.124:3306/ 如果您的访问连接中含有参数,那么请用单引号将整个连接包裹住，例如’ jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true’

dbname：数据库的名字，例如：user。

username：数据库登录用户名。

password：用户对应的密码。

query-sql：使用的查询语句，例如：” SELECT * FROM profile WHERE id>1 AND \$CONDITIONS” 。记得要用引号包围，最后一定要带上 AND \$CONDITIONS。

sp-column：进行切分的条件,一般跟 Mysql 表的主键。

hdfs-dir：要导到 Mysql 去的 HDFS 的数据目录，例如：/user/hive/result。

hive-tablename：对应的 Hive 中的表名，可以是 xxx.yyy。

更加详细的参数使用请参考 Sqoop Query Import。

集群和其他数据库的网络配置请参考，用Aliyun E-MapReduce集群的 Sqoop 工具和数据库同步数据如何配置网络。

E-MapReduce集群运维指南

本篇主要旨在说明EMR集群的一些运维的手段，方便大家在使用的时候可以自主的进行服务的运维。

一些通用的环境变量

在集群上，输入env命令可以看到如下的环境变量配置：

```
JAVA_HOME=/usr/lib/jvm/java

HADOOP_HOME=/usr/lib/hadoop-current
HADOOP_CLASSPATH=/usr/lib/hbase-current/lib/*:/usr/lib/tez-current/*:/usr/lib/tez-current/lib/*:/etc/emr/tez-conf:/usr/lib/hbase-current/lib/*:/usr/lib/tez-current/*:/usr/lib/tez-current/lib/*:/etc/emr/tez-conf:/opt/apps/extra-jars/*:/opt/apps/extra-jars/*
HADOOP_CONF_DIR=/etc/emr/hadoop-conf

SPARK_HOME=/usr/lib/spark-current
SPARK_CONF_DIR=/etc/emr/spark-conf

HBASE_HOME=/usr/lib/hbase-current
HBASE_CONF_DIR=/etc/emr/hbase-conf

HIVE_HOME=/usr/lib/hive-current
HIVE_CONF_DIR=/etc/emr/hive-conf

PIG_HOME=/usr/lib/pig-current
PIG_CONF_DIR=/etc/emr/pig-conf

TEZ_HOME=/usr/lib/tez-current
TEZ_CONF_DIR=/etc/emr/tez-conf

ZEPPELIN_HOME=/usr/lib/zeppelin-current
ZEPPELIN_CONF_DIR=/etc/emr/zeppelin-conf

HUE_HOME=/usr/lib/hue-current
HUE_CONF_DIR=/etc/emr/hue-conf

PRESTO_HOME=/usr/lib/presto-current
PRESTO_CONF_DIR=/etc/emr/presot-conf
```

服务进程的启停

YARN

操作用账号：hadoop

- ResourceManager （Master节点）

```
// 启动
/usr/lib/hadoop-current/sbin/yarn-daemon.sh start resourcemanager
// 停止
/usr/lib/hadoop-current/sbin/yarn-daemon.sh stop resourcemanager
```

- NodeManager （Core节点）

```
// 启动
/usr/lib/hadoop-current/sbin/yarn-daemon.sh start nodemanager
// 停止
/usr/lib/hadoop-current/sbin/yarn-daemon.sh stop nodemanager
```

- JobHistoryServer（Master节点）

```
// 启动
/usr/lib/hadoop-current/sbin/mr-jobhistory-daemon.sh start historyserver
// 停止
/usr/lib/hadoop-current/sbin/mr-jobhistory-daemon.sh stop historyserver
```

- WebProxyServer（Master节点）

```
// 启动
/usr/lib/hadoop-current/sbin/yarn-daemon.sh start proxyserver
// 停止
/usr/lib/hadoop-current/sbin/yarn-daemon.sh stop proxyserver
```

HDFS

操作用账号：hdfs

- NameNode（Master节点）

```
// 启动
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh start namenode
// 停止
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh stop namenode
```

DataNode（Core节点）

```
// 启动
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh start datanode
// 停止
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh stop datanode
```

Hive

操作用账号：hadoop

MetaStore（Master节点）

```
// 启动，这里的内存可以根据需要扩大
HADOOP_HEAPSIZE=512 /usr/lib/hive-current/bin/hive --service metastore >/var/log/hive/metastore.log
2>&1 &
```

- HiveServer2（Master节点）

```
// 启动
HADOOP_HEAPSIZE=512 /usr/lib/hive-current/bin/hive --service hiveserver2
>/var/log/hive/hiveserver2.log 2>&1 &
```

HBase

操作用账号：hdfs

需要注意的，需要在组件中选择了HBase以后才能使用如下的方式来启动，否则对应的配置未完成，启动的时候会有错误。

- HMaster（Master节点）

```
// 启动
/usr/lib/hbase-current/bin/hbase-daemon.sh start master
```

- HRegionServer（Core节点）

```
// 启动
/usr/lib/hbase-curren/bin/hbase-daemon.sh start regionserver
```

- ThriftServer（Master节点）

```
// 启动
/usr/lib/hbase-current/bin/hbase-daemon.sh start thrift -p 9099 >/var/log/hive/thriftserver.log 2>&1 &
// 停止
/usr/lib/hbase-current/bin/hbase-daemon.sh stop thrift
```

Hue

操作用账号：hadoop

```
// 启动
su -l root -c "${HUE_HOME}/build/env/bin/supervisor >/dev/null 2>&1 &"

// 停止
```

```
ps axu | grep hue // 查找到所有的hue进程
kill -9 huepid // 杀掉查找到的对应hue进程
```

Zeppelin

操作用账号：hadoop

```
// 启动， 这里的内存可以根据需要扩大
su -l root -c "ZEPPELIN_MEM=\"-Xmx512m -Xms512m\" ${ZEPPELIN_HOME}/bin/zeppelin-daemon.sh start"
// 停止
su -l root -c "${ZEPPELIN_HOME}/bin/zeppelin-daemon.sh stop"
```

Presto

操作用账号：hdfs

- PrestoServer （Master节点）

```
// 启动
/usr/lib/presto-current//bin/launcher --config=/usr/lib/presto-current/etc/worker-config.properties start
// 停止
/usr/lib/presto-current//bin/launcher --config=/usr/lib/presto-current/etc/worker-config.properties stop
```

- （Core节点）

```
// 启动
/usr/lib/presto-current//bin/launcher --config=/usr/lib/presto-current/etc/coordinator-config.properties start
// 停止
/usr/lib/presto-current//bin/launcher --config=/usr/lib/presto-current/etc/coordinator-config.properties stop
```

批量操作

当需要对worker(Core)节点做统一操作时，可以写脚本命令，一键轻松解决。在EMR集群中，master到所有worker节点在hadoop和hdfs账号下是ssh打通的。

例如 需要对所有worker节点的nodemanager做停止操作，假设有10个worker节点，则可以这样做

```
for i in `seq 1 10`;do ssh emr-worker-$i /usr/lib/hadoop-current/sbin/yarn-daemon.sh stop nodemanager;done
```

Aliyun E-MapReduce SDK Release Note

说明

- emr-core：支持Hadoop/Spark与OSS数据源的交互，默认已经存在集群的运行环境中，作业打包时 **不需要** 将emr-core打进去。
- emr-tablestore: 支持Hadoop/Hive/Spark与OTS数据源的交互，使用时需要打进作业Jar包。
- emr-mns_2.10 / emr-mns_2.11：支持Spark读MNS数据源，使用时需要打进作业Jar包。
- emr-ons_2.10 / emr-ons_2.11：支持Spark读ONS数据源，使用时需要打进作业Jar包。
- emr-logservice_2.10 / emr-logservice_2.11：支持Spark读LogService数据源，使用时需要打进作业Jar包。
- emr-maxcompute_2.10 / emr-maxcompute_2.11: 支持Spark读写MaxCompute数据源，使用时需要打进作业Jar包。

```
   <!--支持OSS数据源 -->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-core</artifactId>
<version>1.4.1</version>
</dependency>

<!--支持OTS数据源-->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-tablestore</artifactId>
<version>1.4.1</version>
</dependency>

<!-- 支持 MNS、ONS、LogService、MaxCompute数据源 (Spark 1.x环境)-->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-mns_2.10</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-logservice_2.10</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-maxcompute_2.10</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
```

```
<artifactId>emr-ons_2.10</artifactId>
<version>1.4.1</version>
</dependency>

<!-- 支持 MNS、ONS、LogService、MaxCompute数据源 (Spark 2.x环境)-->
<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-mns_2.11</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-logservice_2.11</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-maxcompute_2.11</artifactId>
<version>1.4.1</version>
</dependency>

<dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-ons_2.11</artifactId>
<version>1.4.1</version>
</dependency>
```

v1.4.1

- MaxCompute: 修复datetime类型时间截断的问题
- MaxCompute: 修复SimpleDateFormat的线程安全问题

v1.4.0

- MaxCompute：新增datasource的实现方式 (只支持Spark2.x以上版本)。
- LogService：新增Direct API的实现方式 (只支持Spark2.x以上版本)。
- OTS：一些读写的优化。
- 修复读取LogService数据源时，用户AK被错误替换成集群应用角色AK的BUG。

v1.3.2

- 修复OTS的一些BUG。

v1.3.1

- 修复Spark+LogService部分场景下抛空指针问题。
- 从这个版本开始，SDK支持Spark2.x环境。

v1.3.0

- HadoopMR, Spark, SparkSQL, Hive读取OTS数据。
- MNS和LogService支持E-MapReduce的MetaServie功能，支持在E-MapReduce环境下免AK访问MNS和LogService数据。
- 升级部分依赖包版本。

v1.1.3.1

SDK：

- 解决MNS与Spark/Hadoop包的依赖冲突问题。
- 解决Spark Streaming + MNS某些场景下抛空指针问题。
- 解决python sdk的部分BUG。
- Spark Streaming + Loghub支持自定义时间位置的功能。

Core

- 解决Hadoop无法支持原生Snappy文件问题。目前E-MapReduce支持处理LogService以Snappy格式归档到OSS的文件。
- 解决Spark无法支持Snappy压缩文件的问题。
- 解决OSS不支持Hadoop2.7.2 OutputCommitter两种算法的问题。
- 改善Hadoop/Spark读写OSS的性能。
- 解决Spark作业打印的Log4j异常输出的问题。

v1.1.2

- 解决作业慢读写OSS出现的“ConnectionClosedException”问题。
- 解决OSS数据源时部分hadoop命令不可用问题。
- 解决“java.text.ParseException: Unparseable date”问题。
- 优化emr-core支持本地调试运行。
- 兼容老版本的产生的“_$folder$”文件，解释成目录，不再当作普通文件处理。
- Hadoop/Spark读写OSS增加失败重试机制。

v1.1.1

- 解决本地写OSS临时文件时导致多磁盘使用不均衡的问题。
- 去除作业执行过程中创建OSS目录时同时创建的$_folder$标记文件。

v1.1.0

- 升级LogHub SDK到0.6.2，废弃Client DB模式，使用Server DB模式。
- 升级OSS SDK到2.2.0，修复OSS SDK BUG导致的运行异常。
- 新增对MNS的支持。
- 兼容性
    - 对于1.0.x系列SDK
        - 接口：
            - 兼容
        - 命名空间：
            - 不兼容：调整包结构，将包名称com.aliyun更换为com.aliyun.emr
- 修改项目的groupId，从com.aliyun改为com.aliyun.emr。修改后的POM依赖为

```
 <dependency>
<groupId>com.aliyun.emr</groupId>
<artifactId>emr-sdk_2.10</artifactId>
<version>1.1.3.1</version>
</dependency>
```

v1.0.5

- 优化LoghubUtils接口，优化参数输入。
- 优化LogStore数据的输出格式，增加“**topic**” 和“**source**”两个字段。
- 增加LogStore数据拉取的时间间隔参数配置。参数“spark.logservice.fetch.interval.millis” ，默认值200毫秒。
- 更新依赖ODPS SDK版本到0.20.7-public。

v1.0.4

- 将guava的依赖版本降为11.0.2，避免和Hadoop中的guava版本冲突。
- 计算任务支持数据超过5GB的文件大小。

v1.0.3

- 增加OSS Client相关的配置参数。

v1.0.2

- 修复OSS URI解析出错的BUG。

v1.0.1

- 优化OSS URI设置。
- 增加对ONS的支持。
- 增加LogService的支持。
- 支持OSS的追加写特性。
- 支持以multi part方式上传OSS数据。
- 支持以upload part copy方式拷贝OSS数据。