尚硅谷大数据技术之Spark

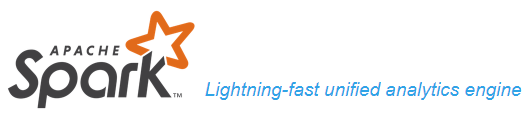
版本：V3.0

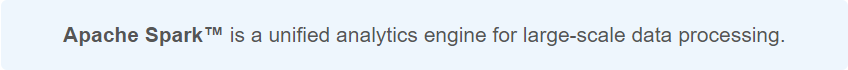


作者：尚硅谷大数据研发部

# **Spark概述**

## **Spark是什么**





Spark是一种基于内存的快速、通用、可扩展的大数据分析计算引擎。

## **Spark and Hadoop**

在之前的学习中，Hadoop的MapReduce是大家广为熟知的计算框架，那为什么咱们还要学习新的计算框架Spark呢，这里就不得不提到Spark和Hadoop的关系。

首先从时间节点上来看:

* Hadoop
* 2006年1月，Doug Cutting加入Yahoo，领导Hadoop的开发
* 2008年1月，Hadoop成为Apache顶级项目
* 2011年1.0正式发布
* 2012年3月稳定版发布
* 2013年10月发布2.X (Yarn)版本
* Spark
* 2009年，Spark诞生于伯克利大学的AMPLab实验室
* 2010年，伯克利大学正式开源了Spark项目
* 2013年6月，Spark成为了Apache基金会下的项目
* 2014年2月，Spark以飞快的速度成为了Apache的顶级项目
* 2015年至今，Spark变得愈发火爆，大量的国内公司开始重点部署或者使用Spark

然后我们再从功能上来看:

* Hadoop
* Hadoop是由java语言编写的，在分布式服务器集群上存储海量数据并运行分布式分析应用的开源框架
* 作为Hadoop分布式文件系统，HDFS处于Hadoop生态圈的最下层，存储着所有的数据，支持着Hadoop的所有服务。它的理论基础源于Google的TheGoogleFileSystem这篇论文，它是GFS的开源实现。
* MapReduce是一种编程模型，Hadoop根据Google的MapReduce论文将其实现，作为Hadoop的分布式计算模型，是Hadoop的核心。基于这个框架，分布式并行程序的编写变得异常简单。综合了HDFS的分布式存储和MapReduce的分布式计算，Hadoop在处理海量数据时，性能横向扩展变得非常容易。
* HBase是对Google的Bigtable的开源实现，但又和Bigtable存在许多不同之处。HBase是一个基于HDFS的分布式数据库，擅长实时地随机读/写超大规模数据集。它也是Hadoop非常重要的组件。
* Spark
* Spark是一种由Scala语言开发的快速、通用、可扩展的大数据分析引擎
* Spark Core中提供了Spark最基础与最核心的功能
* Spark SQL是Spark用来操作结构化数据的组件。通过Spark SQL，用户可以使用SQL或者Apache Hive版本的SQL方言（HQL）来查询数据。
* Spark Streaming是Spark平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的API。

由上面的信息可以获知，Spark出现的时间相对较晚，并且主要功能主要是用于数据计算，所以其实Spark一直被认为是Hadoop 框架的升级版。

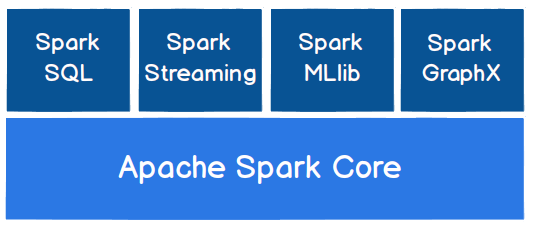
## **Spark or Hadoop**

Hadoop的MR框架和Spark框架都是数据处理框架，那么我们在使用时如何选择呢？

* Hadoop MapReduce由于其设计初衷并不是为了满足循环迭代式数据流处理，因此在多并行运行的数据可复用场景（如：机器学习、图挖掘算法、交互式数据挖掘算法）中存在诸多计算效率等问题。所以Spark应运而生，Spark就是在传统的MapReduce 计算框架的基础上，利用其计算过程的优化，从而大大加快了数据分析、挖掘的运行和读写速度，并将计算单元缩小到更适合并行计算和重复使用的RDD计算模型。
* 机器学习中ALS、凸优化梯度下降等。这些都需要基于数据集或者数据集的衍生数据反复查询反复操作。MR这种模式不太合适，即使多MR串行处理，性能和时间也是一个问题。数据的共享依赖于磁盘。另外一种是交互式数据挖掘，MR显然不擅长。而Spark所基于的scala语言恰恰擅长函数的处理。
* Spark是一个分布式数据快速分析项目。它的核心技术是弹性分布式数据集（Resilient Distributed Datasets），提供了比MapReduce丰富的模型，可以快速在内存中对数据集进行多次迭代，来支持复杂的数据挖掘算法和图形计算算法。
* Spark和Hadoop的根本差异是多个作业之间的数据通信问题 : Spark多个作业之间数据通信是基于内存，而Hadoop是基于磁盘。
* Spark Task的启动时间快。Spark采用fork线程的方式，而Hadoop采用创建新的进程的方式。
* Spark只有在shuffle的时候将数据写入磁盘，而Hadoop中多个MR作业之间的数据交互都要依赖于磁盘交互
* Spark的缓存机制比HDFS的缓存机制高效。

经过上面的比较，我们可以看出在绝大多数的数据计算场景中，Spark确实会比MapReduce更有优势。但是Spark是基于内存的，所以在实际的生产环境中，由于内存的限制，可能会由于内存资源不够导致Job执行失败，此时，MapReduce其实是一个更好的选择，所以Spark并不能完全替代MR。

## **Spark 核心模块**



* **Spark Core**

Spark Core中提供了Spark最基础与最核心的功能，Spark其他的功能如：Spark SQL，Spark Streaming，GraphX, MLlib都是在Spark Core的基础上进行扩展的

* **Spark SQL**

Spark SQL是Spark用来操作结构化数据的组件。通过Spark SQL，用户可以使用SQL或者Apache Hive版本的SQL方言（HQL）来查询数据。

* **Spark Streaming**

Spark Streaming是Spark平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的API。

* **Spark MLlib**

MLlib是Spark提供的一个机器学习算法库。MLlib不仅提供了模型评估、数据导入等额外的功能，还提供了一些更底层的机器学习原语。

* **Spark GraphX**

GraphX是Spark面向图计算提供的框架与算法库。

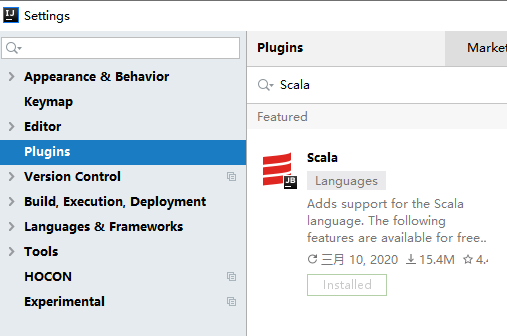
# **Spark快速上手**

在大数据早期的课程中我们已经学习了MapReduce框架的原理及基本使用，并了解了其底层数据处理的实现方式。接下来，就让咱们走进Spark的世界，了解一下它是如何带领我们完成数据处理的。

## **2.1 创建Maven项目**

### **2.1.1 增加Scala插件**

Spark由Scala语言开发的，所以本课件接下来的开发所使用的语言也为Scala，咱们当前使用的Spark版本为3.0.0，默认采用的Scala编译版本为2.12，所以后续开发时。我们依然采用这个版本。开发前请保证IDEA开发工具中含有Scala开发插件



### **2.1.2 增加依赖关系**

修改Maven项目中的POM文件，增加Spark框架的依赖关系。本课件基于Spark3.0版本，使用时请注意对应版本。

<dependencies>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-core\_2.12</artifactId>

<version>3.0.0</version>

</dependency>

</dependencies>

<build>

<plugins>

<!-- 该插件用于将Scala代码编译成class文件 -->

<plugin>

<groupId>net.alchim31.maven</groupId>

<artifactId>scala-maven-plugin</artifactId>

<version>3.2.2</version>

<executions>

<execution>

<!-- 声明绑定到maven的compile阶段 -->

<goals>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<version>3.1.0</version>

<configuration>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

<executions>

<execution>

<id>make-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

### **2.1.3 WordCount**

为了能直观地感受Spark框架的效果，接下来我们实现一个大数据学科中最常见的教学案例WordCount

// 创建Spark运行配置对象

val sparkConf = new SparkConf().setMaster("local[\*]").setAppName("WordCount")

// 创建Spark上下文环境对象（连接对象）

val sc : SparkContext = new SparkContext(sparkConf)

// 读取文件数据

val fileRDD: RDD[String] = sc.textFile("input/word.txt")

// 将文件中的数据进行分词

val wordRDD: RDD[String] = fileRDD.flatMap( \_.split(" ") )

// 转换数据结构 word => (word, 1)

val word2OneRDD: RDD[(String, Int)] = wordRDD.map((\_,1))

// 将转换结构后的数据按照相同的单词进行分组聚合

val word2CountRDD: RDD[(String, Int)] = word2OneRDD.reduceByKey(\_+\_)

// 将数据聚合结果采集到内存中

val word2Count: Array[(String, Int)] = word2CountRDD.collect()

// 打印结果

word2Count.foreach(println)

//关闭Spark连接

sc.stop()

执行过程中，会产生大量的执行日志，如果为了能够更好的查看程序的执行结果，可以在项目的resources目录中创建log4j.properties文件，并添加日志配置信息：

log4j.rootCategory=ERROR, console

log4j.appender.console=org.apache.log4j.ConsoleAppender

log4j.appender.console.target=System.err

log4j.appender.console.layout=org.apache.log4j.PatternLayout

log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

# Set the default spark-shell log level to ERROR. When running the spark-shell, the

# log level for this class is used to overwrite the root logger's log level, so that

# the user can have different defaults for the shell and regular Spark apps.

log4j.logger.org.apache.spark.repl.Main=ERROR

# Settings to quiet third party logs that are too verbose

log4j.logger.org.spark\_project.jetty=ERROR

log4j.logger.org.spark\_project.jetty.util.component.AbstractLifeCycle=ERROR

log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=ERROR

log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=ERROR

log4j.logger.org.apache.parquet=ERROR

log4j.logger.parquet=ERROR

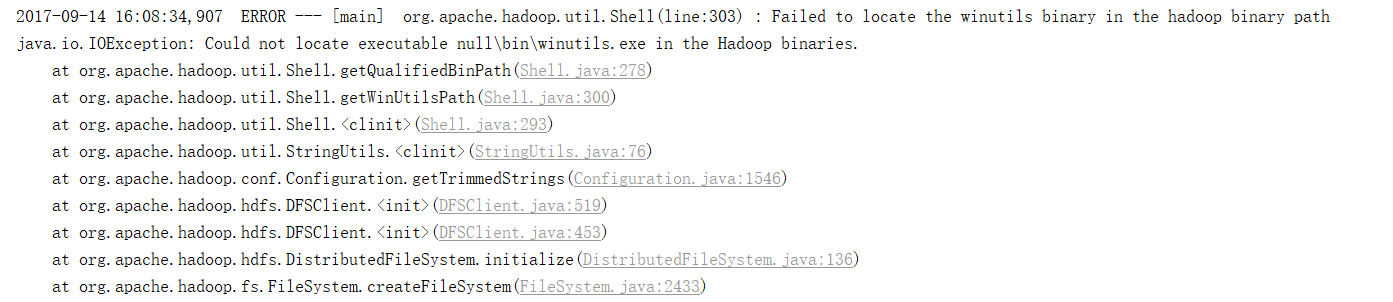
# SPARK-9183: Settings to avoid annoying messages when looking up nonexistent UDFs in SparkSQL with Hive support

log4j.logger.org.apache.hadoop.hive.metastore.RetryingHMSHandler=FATAL

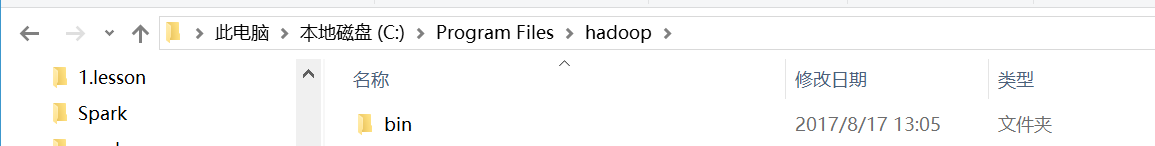
log4j.logger.org.apache.hadoop.hive.ql.exec.FunctionRegistry=ERROR

### **2.1.4 异常处理**

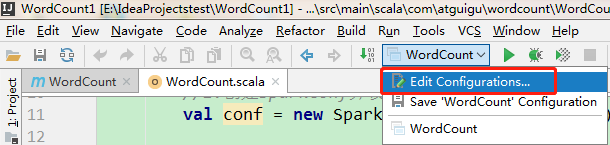
如果本机操作系统是Windows，在程序中使用了Hadoop相关的东西，比如写入文件到HDFS，则会遇到如下异常：

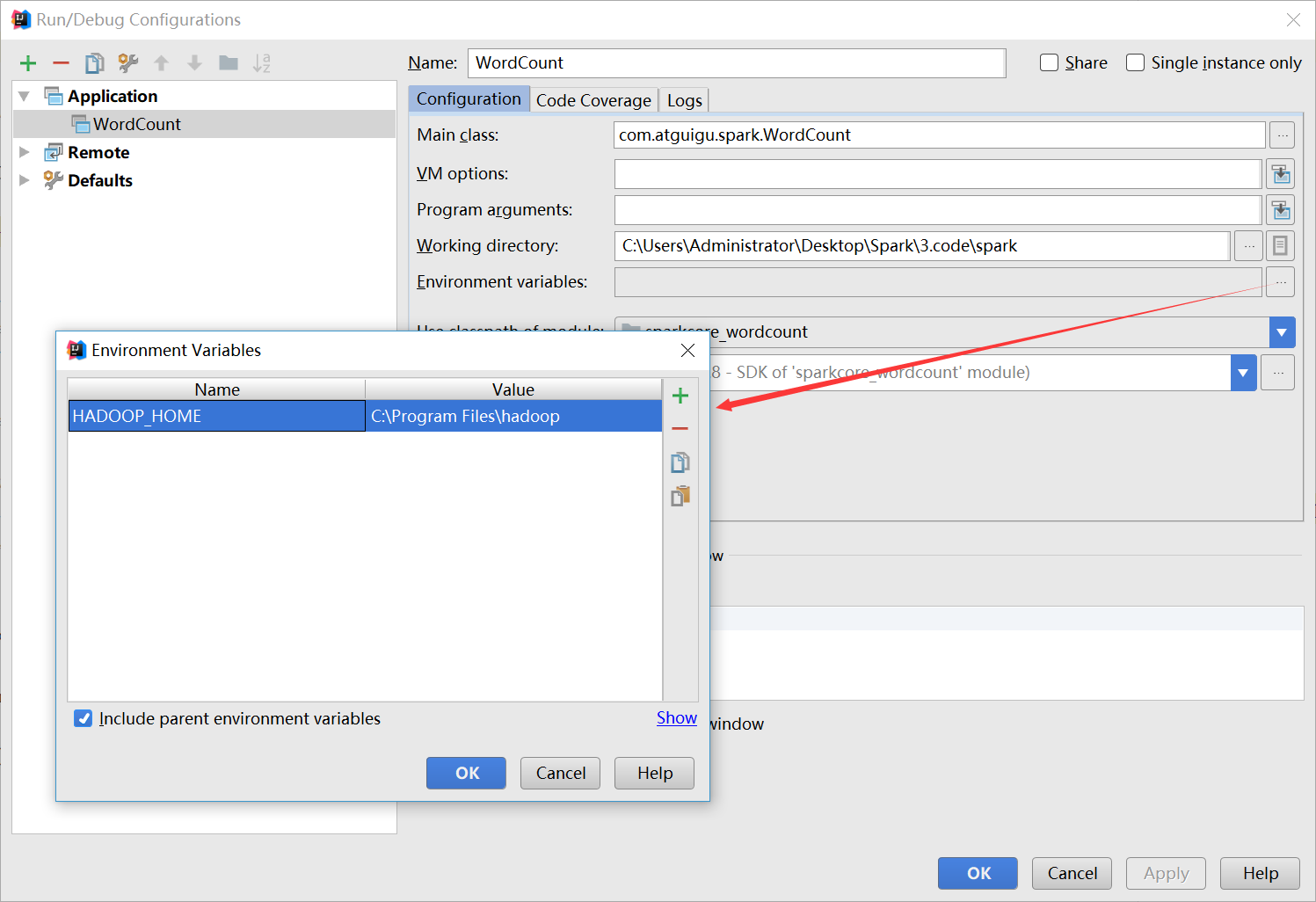


出现这个问题的原因，并不是程序的错误，而是windows系统用到了hadoop相关的服务，解决办法是通过配置关联到windows的系统依赖就可以了



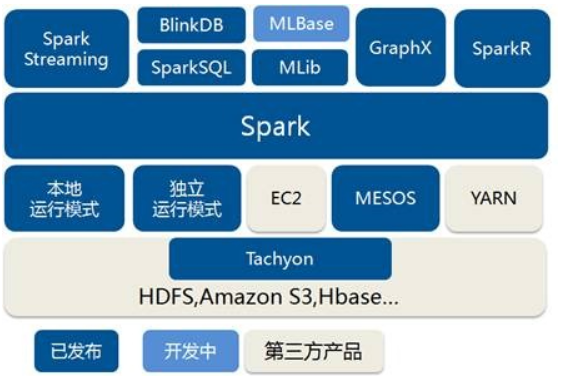
在IDEA中配置Run Configuration，添加HADOOP\_HOME变量





# **Spark运行环境**

Spark作为一个数据处理框架和计算引擎，被设计在所有常见的集群环境中运行, 在国内工作中主流的环境为Yarn，不过逐渐容器式环境也慢慢流行起来。接下来，我们就分别看看不同环境下Spark的运行



## **3.1 Local模式**

想啥呢，你之前一直在使用的模式可不是Local模式哟。所谓的Local模式，就是不需要其他任何节点资源就可以在本地执行Spark代码的环境，一般用于教学，调试，演示等，之前在IDEA中运行代码的环境我们称之为开发环境，不太一样。

### **3.1.1 解压缩文件**

将spark-3.0.0-bin-hadoop3.2.tgz文件上传到Linux并解压缩，放置在指定位置，路径中不要包含中文或空格，课件后续如果涉及到解压缩操作，不再强调。

tar -zxvf spark-3.0.0-bin-hadoop3.2.tgz -C /opt/module

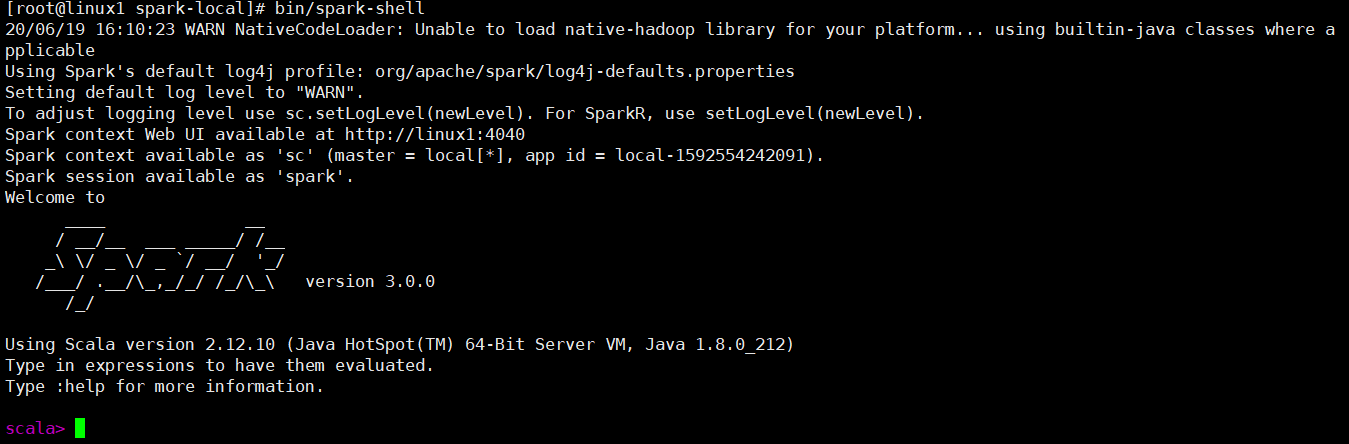
cd /opt/module

mv spark-3.0.0-bin-hadoop3.2 spark-local

### **3.1.2 启动Local环境**

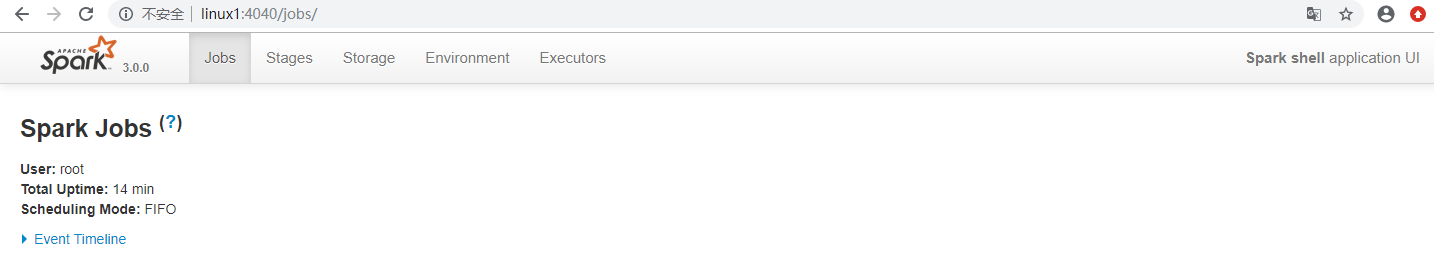
1. 进入解压缩后的路径，执行如下指令

bin/spark-shell



1. 启动成功后，可以输入网址进行Web UI监控页面访问

http://虚拟机地址:4040



### **3.1.3 命令行工具**

在解压缩文件夹下的data目录中，添加word.txt文件。在命令行工具中执行如下代码指令（和IDEA中代码简化版一致）

sc.textFile("data/word.txt").flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_).collect



### **3.1.4 退出本地模式**

按键Ctrl+C或输入Scala指令

:quit

### **3.1.5 提交应用**

bin/spark-submit \

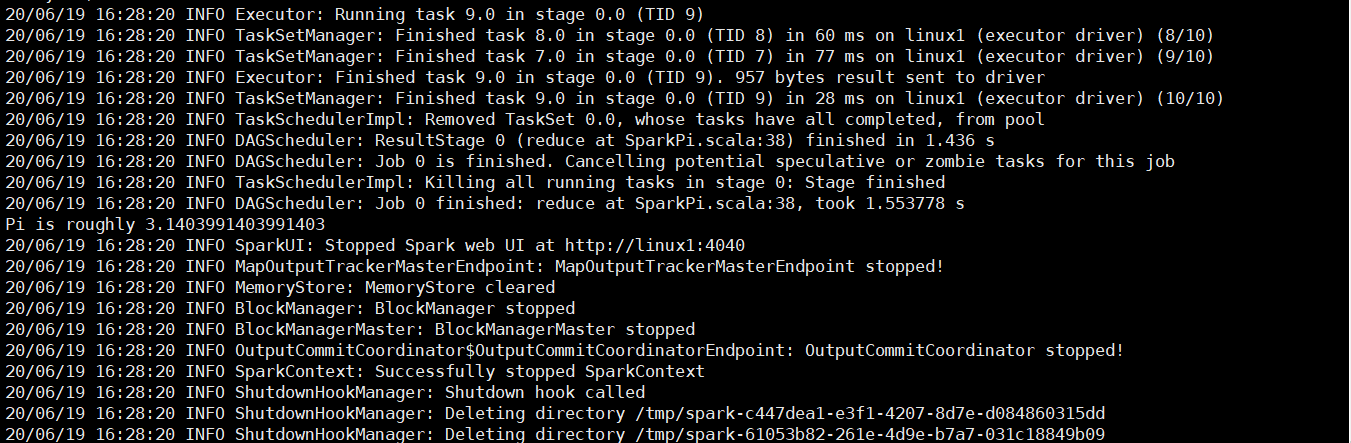
--class org.apache.spark.examples.SparkPi \

--master local[2] \

./examples/jars/spark-examples\_2.12-3.0.0.jar \

10

1. --class表示要执行程序的主类，此处可以更换为咱们自己写的应用程序
2. --master local[2] 部署模式，默认为本地模式，数字表示分配的虚拟CPU核数量
3. spark-examples\_2.12-3.0.0.jar 运行的应用类所在的jar包，实际使用时，可以设定为咱们自己打的jar包
4. 数字10表示程序的入口参数，用于设定当前应用的任务数量



## **3.2 Standalone模式**

local本地模式毕竟只是用来进行练习演示的，真实工作中还是要将应用提交到对应的集群中去执行，这里我们来看看只使用Spark自身节点运行的集群模式，也就是我们所谓的独立部署（Standalone）模式。Spark的Standalone模式体现了经典的master-slave模式。

集群规划:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linux1 | Linux2 | Linux3 |
| Spark | Worker Master | Worker | Worker |

### **3.2.1 解压缩文件**

将spark-3.0.0-bin-hadoop3.2.tgz文件上传到Linux并解压缩在指定位置

tar -zxvf spark-3.0.0-bin-hadoop3.2.tgz -C /opt/module

cd /opt/module

mv spark-3.0.0-bin-hadoop3.2 spark-standalone

### **3.2.2 修改配置文件**

1. 进入解压缩后路径的conf目录，修改slaves.template文件名为slaves

mv slaves.template slaves

1. 修改slaves文件，添加worker节点

linux1

linux2

linux3

1. 修改spark-env.sh.template文件名为spark-env.sh

mv spark-env.sh.template spark-env.sh

1. 修改spark-env.sh文件，添加JAVA\_HOME环境变量和集群对应的master节点

export JAVA\_HOME=/opt/module/jdk1.8.0\_144

SPARK\_MASTER\_HOST=linux1

SPARK\_MASTER\_PORT=7077

注意：7077端口，相当于hadoop3内部通信的8020端口，此处的端口需要确认自己的Hadoop配置

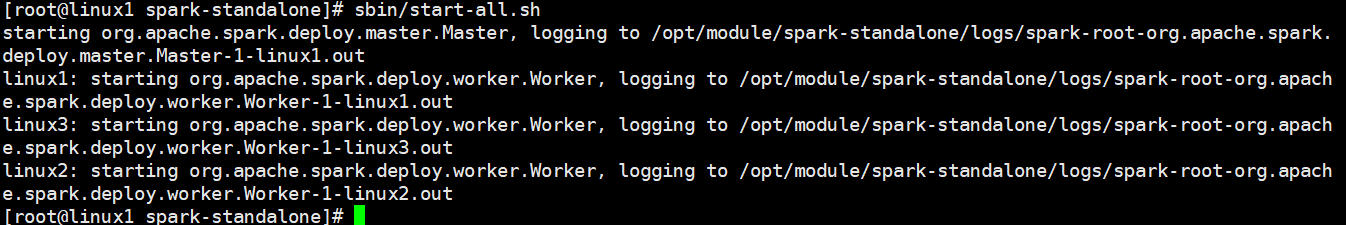
1. 分发spark-standalone目录

xsync spark-standalone

### **3.2.3 启动集群**

1. 执行脚本命令：

sbin/start-all.sh



1. 查看三台服务器运行进程

================linux1================

3330 Jps

3238 Worker

3163 Master

================linux2================

2966 Jps

2908 Worker

================linux3================

2978 Worker

3036 Jps

1. 查看Master资源监控Web UI界面: [http://linux1:8080](http://linux1:8080,)



### **3.2.4 提交应用**

bin/spark-submit \

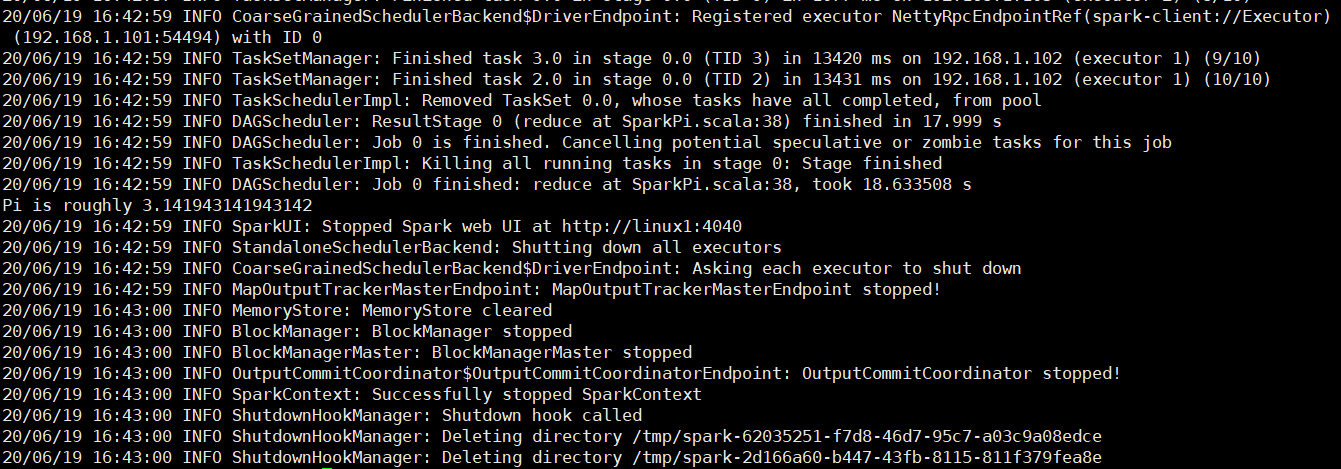
--class org.apache.spark.examples.SparkPi \

--master spark://linux1:7077 \

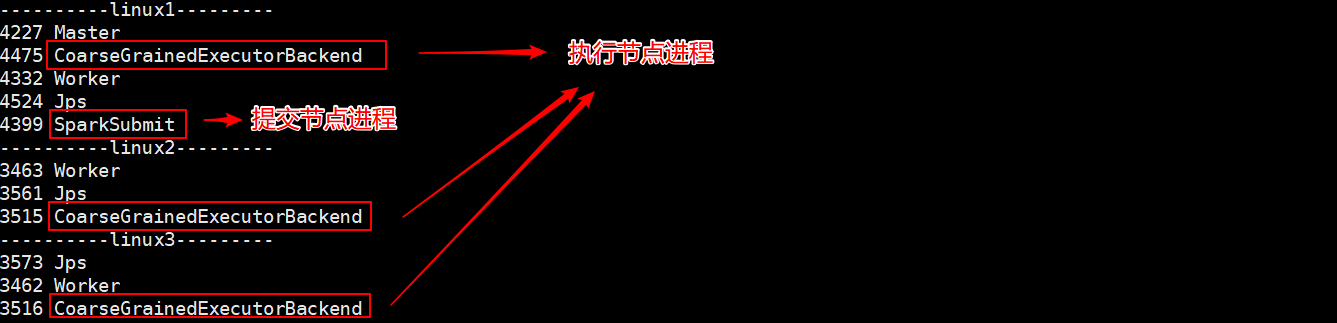
./examples/jars/spark-examples\_2.12-3.0.0.jar \

10

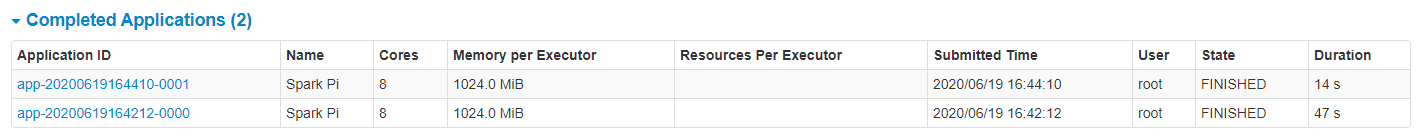
1. --class表示要执行程序的主类
2. --master spark://linux1:7077 独立部署模式，连接到Spark集群
3. spark-examples\_2.12-3.0.0.jar 运行类所在的jar包
4. 数字10表示程序的入口参数，用于设定当前应用的任务数量



执行任务时，会产生多个Java进程



执行任务时，默认采用服务器集群节点的总核数，每个节点内存1024M。



### **3.2.5 提交参数说明**

在提交应用中，一般会同时一些提交参数

bin/spark-submit \

--class <main-class>

--master <master-url> \

... # other options

<application-jar> \

[application-arguments]

|  |  |  |
| --- | --- | --- |
| 参数 | 解释 | 可选值举例 |
| --class | Spark程序中包含主函数的类 |  |
| --master | Spark程序运行的模式(环境) | 模式：local[\*]、spark://linux1:7077、  Yarn |
| --executor-memory 1G | 指定每个executor可用内存为1G | 符合集群内存配置即可，具体情况具体分析。 |
| --total-executor-cores 2 | 指定所有executor使用的cpu核数为2个 |
| --executor-cores | 指定每个executor使用的cpu核数 |
| application-jar | 打包好的应用jar，包含依赖。这个URL在集群中全局可见。 比如hdfs:// 共享存储系统，如果是file:// path，那么所有的节点的path都包含同样的jar |
| application-arguments | 传给main()方法的参数 |

### **3.2.6 配置历史服务**

由于spark-shell停止掉后，集群监控linux1:4040页面就看不到历史任务的运行情况，所以开发时都配置历史服务器记录任务运行情况。

1. 修改spark-defaults.conf.template文件名为spark-defaults.conf

mv spark-defaults.conf.template spark-defaults.conf

1. 修改spark-default.conf文件，配置日志存储路径

spark.eventLog.enabled true

spark.eventLog.dir hdfs://linux1:8020/directory

注意：需要启动hadoop集群，HDFS上的directory目录需要提前存在。

sbin/start-dfs.sh

hadoop fs -mkdir /directory

1. 修改spark-env.sh文件, 添加日志配置

export SPARK\_HISTORY\_OPTS="

-Dspark.history.ui.port=18080

-Dspark.history.fs.logDirectory=hdfs://linux1:8020/directory

-Dspark.history.retainedApplications=30"

* 参数1含义：WEB UI访问的端口号为18080
* 参数2含义：指定历史服务器日志存储路径
* 参数3含义：指定保存Application历史记录的个数，如果超过这个值，旧的应用程序信息将被删除，这个是内存中的应用数，而不是页面上显示的应用数。

1. 分发配置文件

xsync conf

1. 重新启动集群和历史服务

sbin/start-all.sh

sbin/start-history-server.sh

1. 重新执行任务

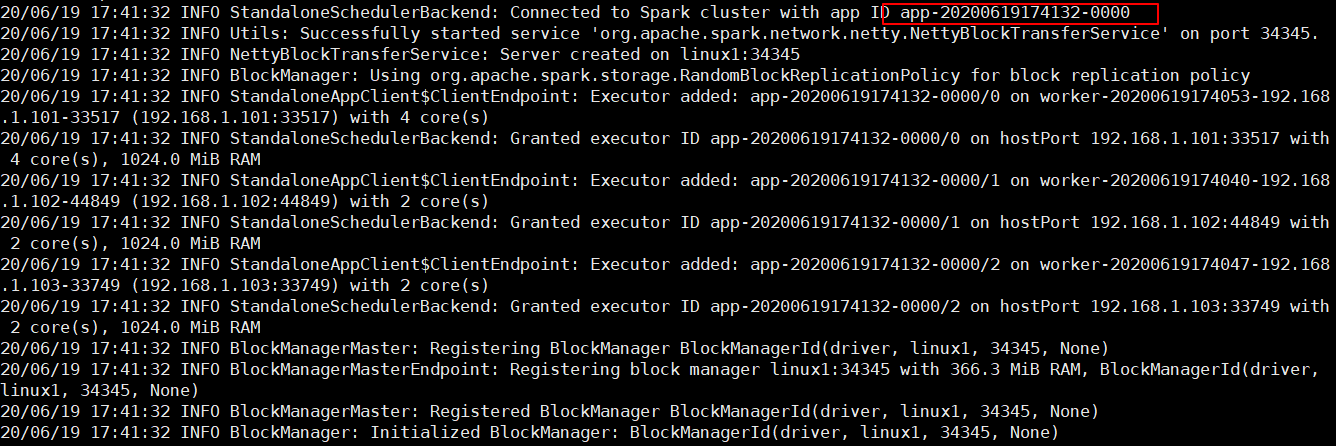
bin/spark-submit \

--class org.apache.spark.examples.SparkPi \

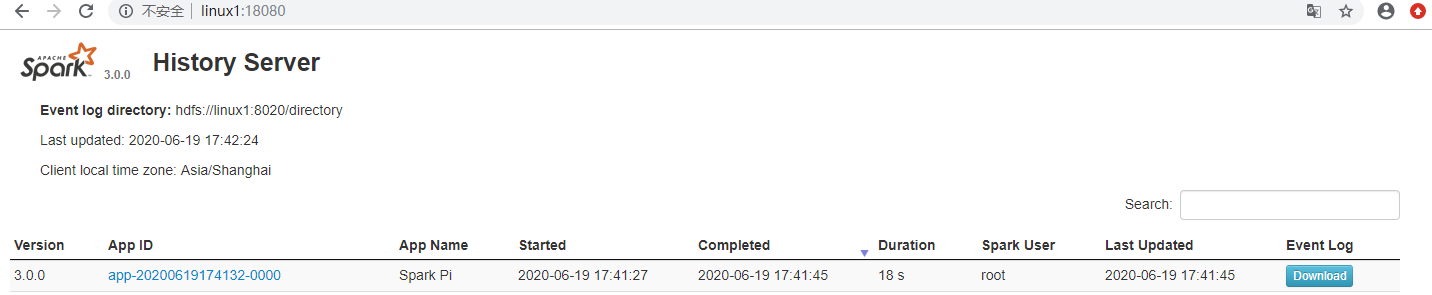
--master spark://linux1:7077 \

./examples/jars/spark-examples\_2.12-3.0.0.jar \

10



1. 查看历史服务：http://linux1:18080



### **3.2.7 配置高可用（HA）**

所谓的高可用是因为当前集群中的Master节点只有一个，所以会存在单点故障问题。所以为了解决单点故障问题，需要在集群中配置多个Master节点，一旦处于活动状态的Master发生故障时，由备用Master提供服务，保证作业可以继续执行。这里的高可用一般采用Zookeeper设置

**集群规划**:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linux1 | Linux2 | Linux3 |
| Spark | Master  Zookeeper  Worker | Master  Zookeeper  Worker | Zookeeper  Worker |

1. 停止集群

sbin/stop-all.sh

1. 启动Zookeeper

xstart zk

1. 修改spark-env.sh文件添加如下配置

注释如下内容：

#SPARK\_MASTER\_HOST=linux1

#SPARK\_MASTER\_PORT=7077

添加如下内容:

#Master监控页面默认访问端口为8080，但是可能会和Zookeeper冲突，所以改成8989，也可以自定义，访问UI监控页面时请注意

SPARK\_MASTER\_WEBUI\_PORT=8989

export SPARK\_DAEMON\_JAVA\_OPTS="

-Dspark.deploy.recoveryMode=ZOOKEEPER

-Dspark.deploy.zookeeper.url=linux1,linux2,linux3

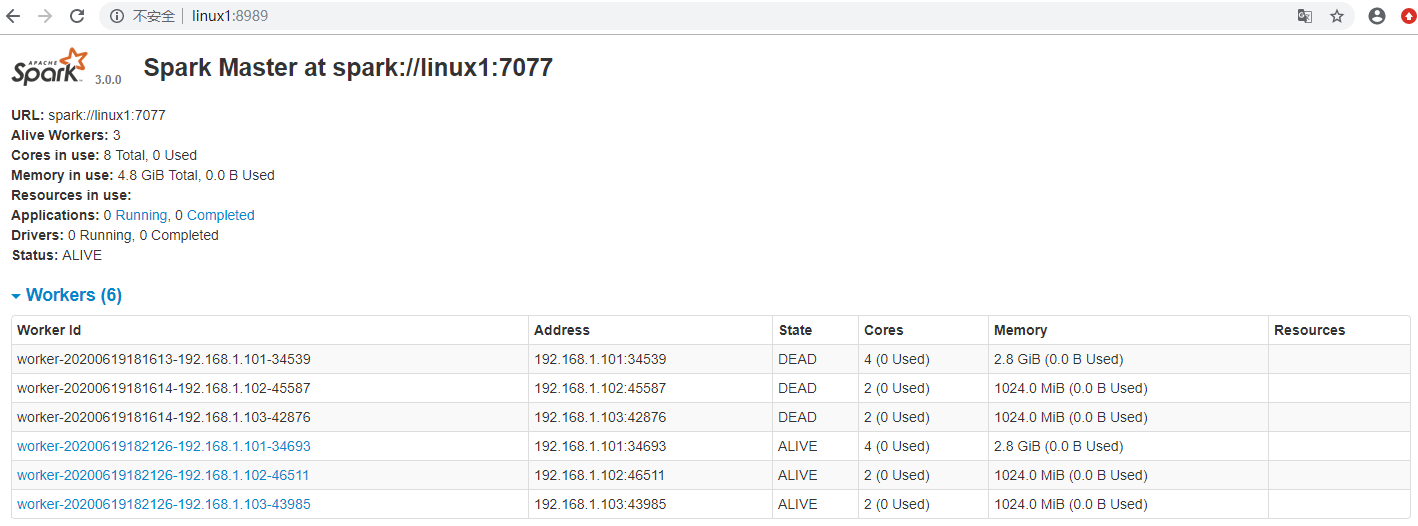
-Dspark.deploy.zookeeper.dir=/spark"

1. 分发配置文件

xsync conf/

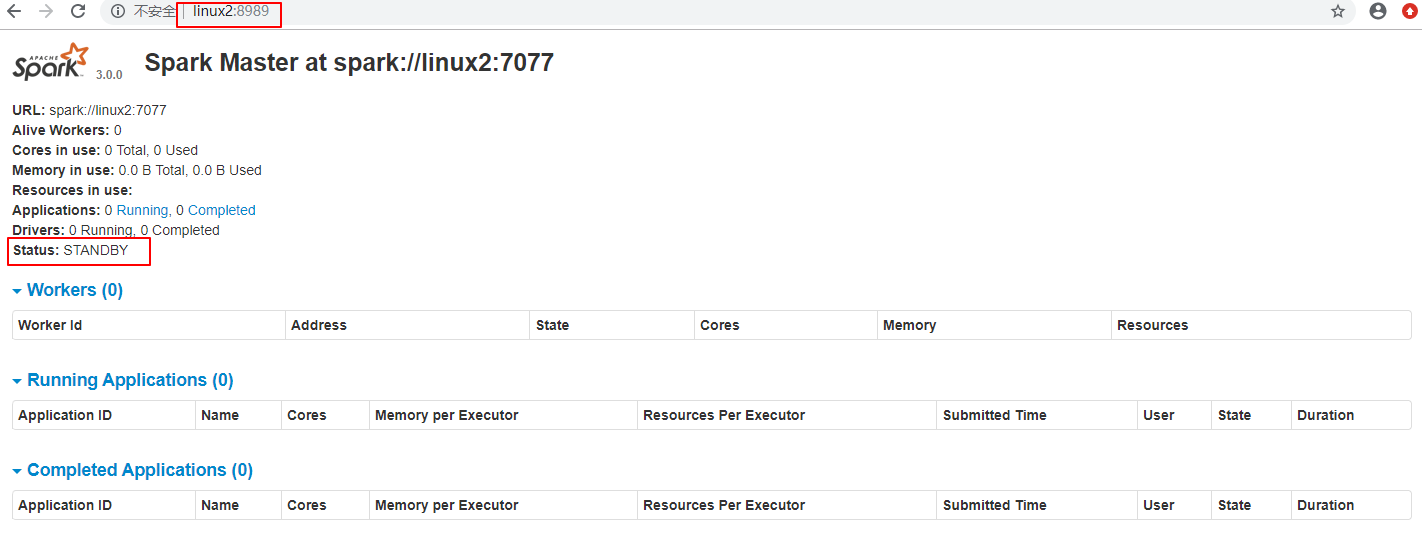
1. 启动集群

sbin/start-all.sh



1. 启动linux2的单独Master节点，此时linux2节点Master状态处于备用状态

[root@linux2 spark-standalone]# sbin/start-master.sh



1. 提交应用到高可用集群

bin/spark-submit \

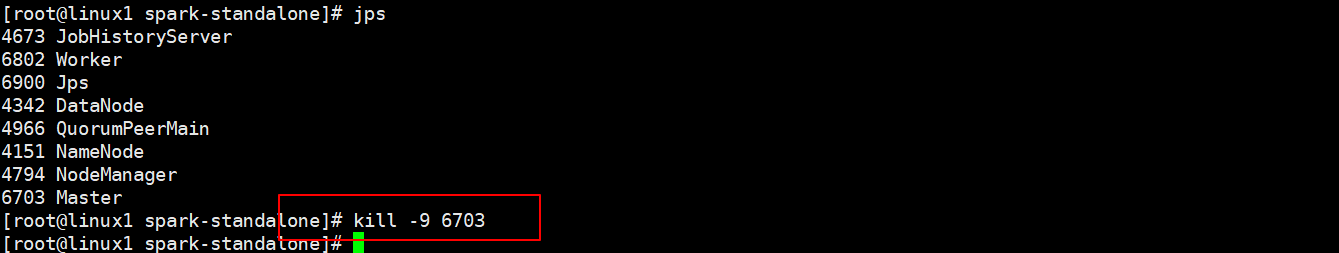
--class org.apache.spark.examples.SparkPi \

--master spark://linux1:7077,linux2:7077 \

./examples/jars/spark-examples\_2.12-3.0.0.jar \

10

1. 停止linux1的Master资源监控进程



1. 查看linux2的Master 资源监控Web UI，稍等一段时间后，linux2节点的Master状态提升为活动状态



## **3.3 Yarn模式**

独立部署（Standalone）模式由Spark自身提供计算资源，无需其他框架提供资源。这种方式降低了和其他第三方资源框架的耦合性，独立性非常强。但是你也要记住，Spark主要是计算框架，而不是资源调度框架，所以本身提供的资源调度并不是它的强项，所以还是和其他专业的资源调度框架集成会更靠谱一些。所以接下来我们来学习在强大的Yarn环境下Spark是如何工作的（其实是因为在国内工作中，Yarn使用的非常多）。

### **3.3.1 解压缩文件**

将spark-3.0.0-bin-hadoop3.2.tgz文件上传到linux并解压缩，放置在指定位置。

tar -zxvf spark-3.0.0-bin-hadoop3.2.tgz -C /opt/module

cd /opt/module

mv spark-3.0.0-bin-hadoop3.2 spark-yarn

### **3.3.2 修改配置文件**

1. 修改hadoop配置文件/opt/module/hadoop/etc/hadoop/yarn-site.xml, 并分发

<!--是否启动一个线程检查每个任务正使用的物理内存量，如果任务超出分配值，则直接将其杀掉，默认是true -->

<property>

<name>yarn.nodemanager.pmem-check-enabled</name>

<value>false</value>

</property>

<!--是否启动一个线程检查每个任务正使用的虚拟内存量，如果任务超出分配值，则直接将其杀掉，默认是true -->

<property>

<name>yarn.nodemanager.vmem-check-enabled</name>

<value>false</value>

</property>

1. 修改conf/spark-env.sh，添加JAVA\_HOME和YARN\_CONF\_DIR配置

mv spark-env.sh.template spark-env.sh

。。。

export JAVA\_HOME=/opt/module/jdk1.8.0\_144

YARN\_CONF\_DIR=/opt/module/hadoop/etc/hadoop

### **3.3.3 启动HDFS以及YARN集群**

瞅啥呢，自己启动去！

### **3.3.4 提交应用**

bin/spark-submit \

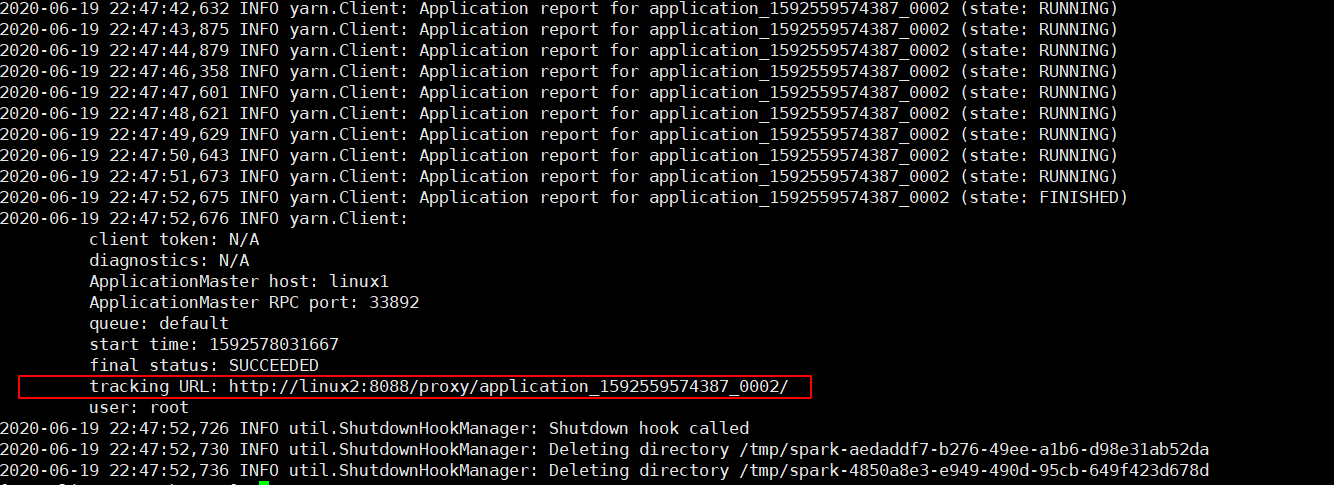
--class org.apache.spark.examples.SparkPi \

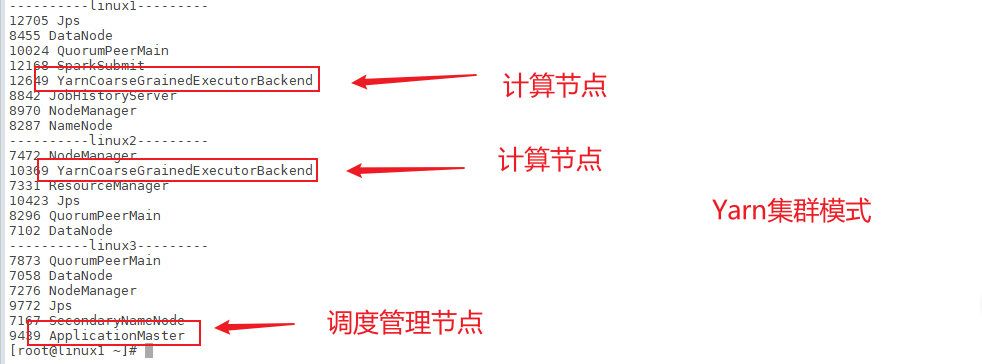
--master yarn \

--deploy-mode cluster \

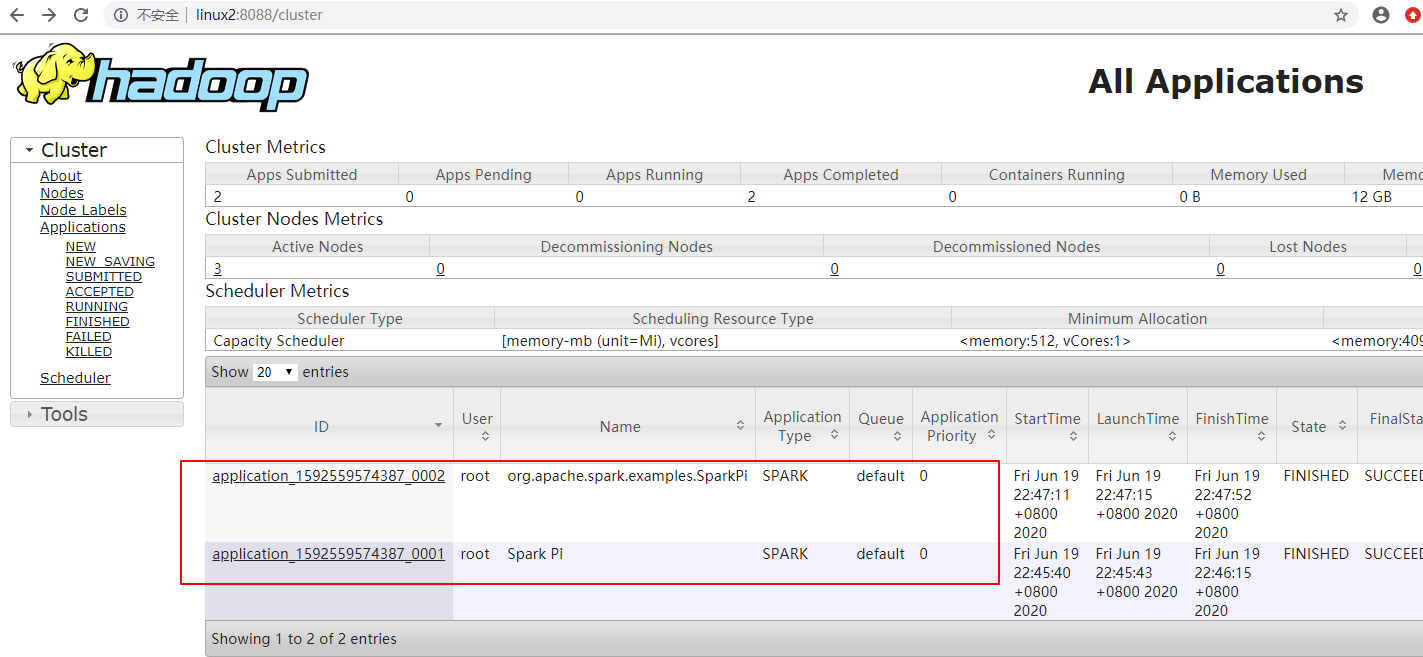
./examples/jars/spark-examples\_2.12-3.0.0.jar \

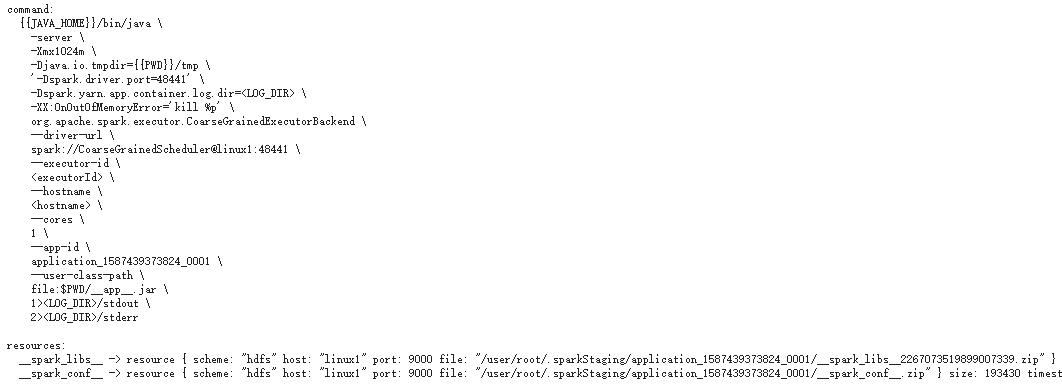
10





查看http://linux2:8088页面，点击History，查看历史页面





### **3.3.5 配置历史服务器**

1. 修改spark-defaults.conf.template文件名为spark-defaults.conf

mv spark-defaults.conf.template spark-defaults.conf

1. 修改spark-default.conf文件，配置日志存储路径

spark.eventLog.enabled true

spark.eventLog.dir hdfs://linux1:8020/directory

注意：需要启动hadoop集群，HDFS上的目录需要提前存在。

[root@linux1 hadoop]# sbin/start-dfs.sh

[root@linux1 hadoop]# hadoop fs -mkdir /directory

1. 修改spark-env.sh文件, 添加日志配置

export SPARK\_HISTORY\_OPTS="

-Dspark.history.ui.port=18080

-Dspark.history.fs.logDirectory=hdfs://linux1:8020/directory

-Dspark.history.retainedApplications=30"

* 参数1含义：WEB UI访问的端口号为18080
* 参数2含义：指定历史服务器日志存储路径
* 参数3含义：指定保存Application历史记录的个数，如果超过这个值，旧的应用程序信息将被删除，这个是内存中的应用数，而不是页面上显示的应用数。

1. 修改spark-defaults.conf

spark.yarn.historyServer.address=linux1:18080

spark.history.ui.port=18080

1. 启动历史服务

sbin/start-history-server.sh

1. 重新提交应用

bin/spark-submit \

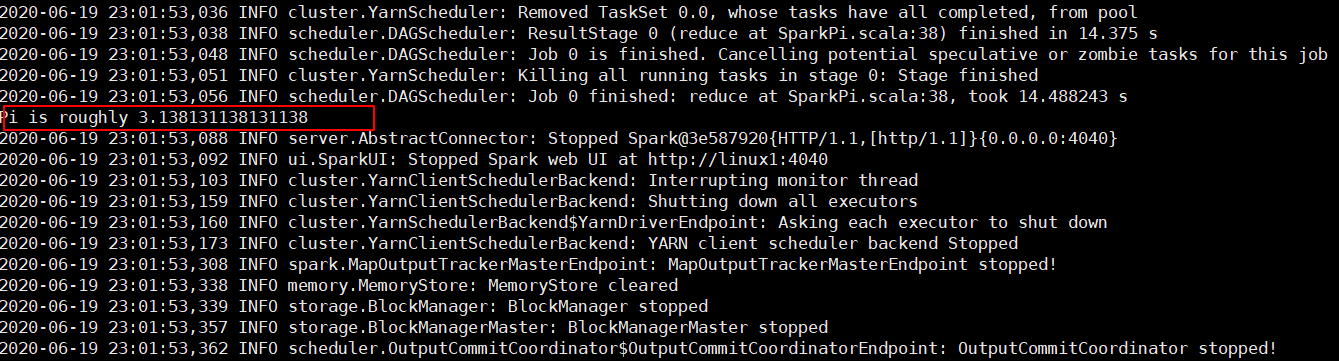
--class org.apache.spark.examples.SparkPi \

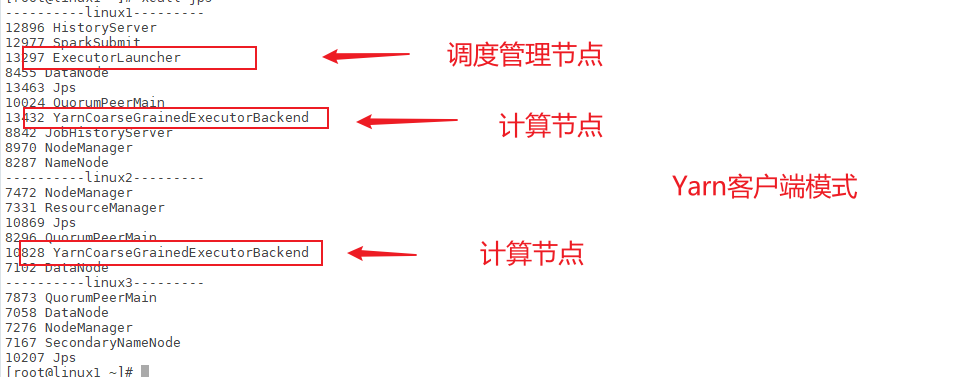
--master yarn \

--deploy-mode client \

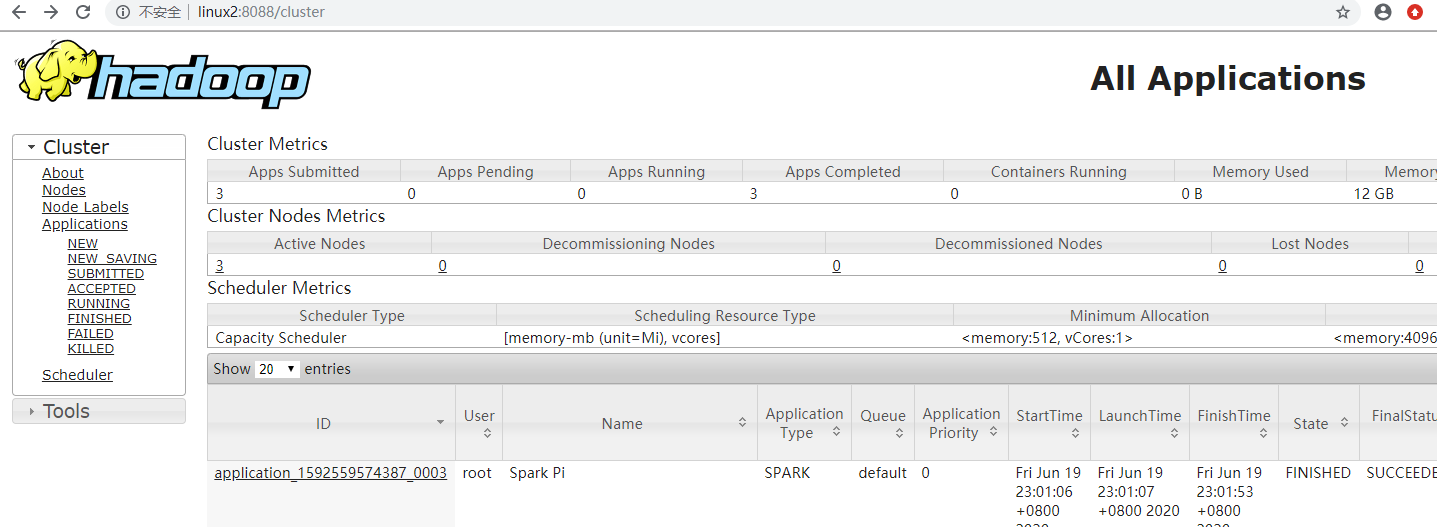
./examples/jars/spark-examples\_2.12-3.0.0.jar \

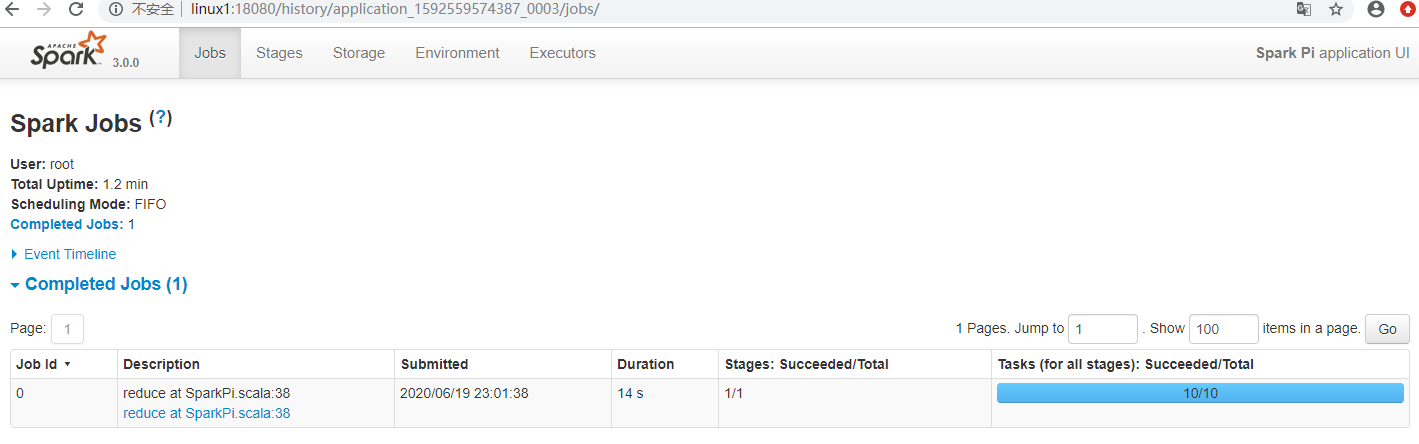
10





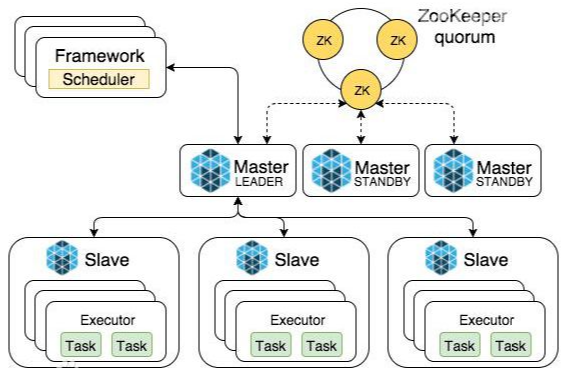
1. Web页面查看日志：http://linux2:8088



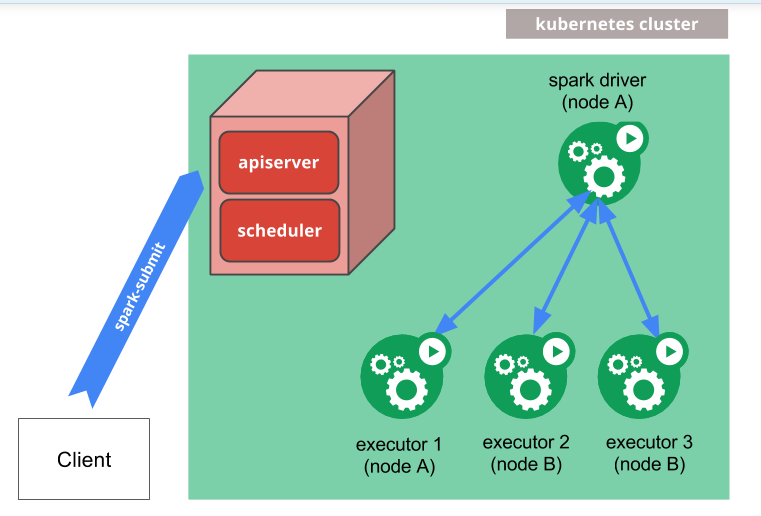


## **3.4 K8S & Mesos模式**

Mesos是Apache下的开源分布式资源管理框架，它被称为是分布式系统的内核,在Twitter得到广泛使用,管理着Twitter超过30,0000台服务器上的应用部署，但是在国内，依然使用着传统的Hadoop大数据框架，所以国内使用Mesos框架的并不多，但是原理其实都差不多，这里我们就不做过多讲解了。



容器化部署是目前业界很流行的一项技术，基于Docker镜像运行能够让用户更加方便地对应用进行管理和运维。容器管理工具中最为流行的就是Kubernetes（k8s），而Spark也在最近的版本中支持了k8s部署模式。这里我们也不做过多的讲解。给个链接大家自己感受一下：https://spark.apache.org/docs/latest/running-on-kubernetes.html



## **3.5 Windows模式**

在同学们自己学习时，每次都需要启动虚拟机，启动集群，这是一个比较繁琐的过程，并且会占大量的系统资源，导致系统执行变慢，不仅仅影响学习效果，也影响学习进度，Spark非常暖心地提供了可以在windows系统下启动本地集群的方式，这样，在不使用虚拟机的情况下，也能学习Spark的基本使用，摸摸哒！



在后续的教学中，为了能够给同学们更加流畅的教学效果和教学体验，我们一般情况下都会采用windows系统的集群来学习Spark。

### **3.5.1 解压缩文件**

将文件spark-3.0.0-bin-hadoop3.2.tgz解压缩到无中文无空格的路径中

### **3.5.2 启动本地环境**

1. 执行解压缩文件路径下bin目录中的spark-shell.cmd文件，启动Spark本地环境



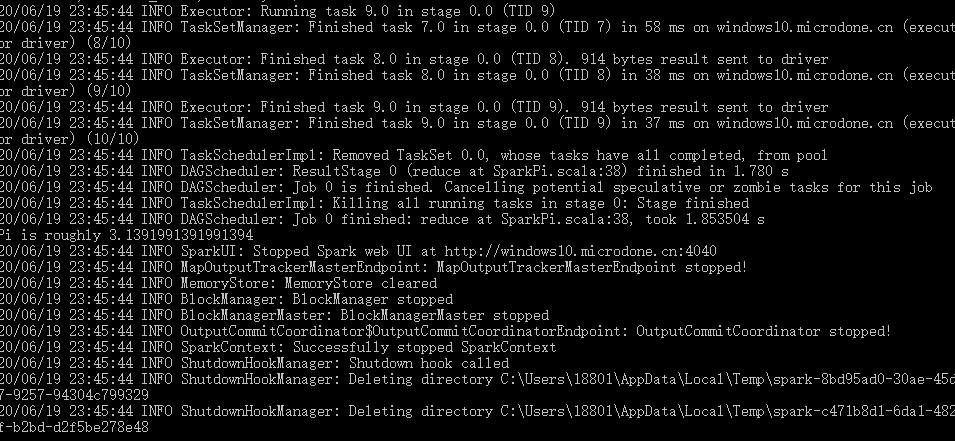
1. 在bin目录中创建input目录，并添加word.txt文件, 在命令行中输入脚本代码



### **3.5.3 命令行提交应用**

在DOS命令行窗口中执行提交指令

spark-submit --class org.apache.spark.examples.SparkPi --master local[2] ../examples/jars/spark-examples\_2.12-3.0.0.jar 10



## **3.6 部署模式对比**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模式 | Spark安装机器数 | 需启动的进程 | 所属者 | 应用场景 |
| Local | 1 | 无 | Spark | 测试 |
| Standalone | 3 | Master及Worker | Spark | 单独部署 |
| Yarn | 1 | Yarn及HDFS | Hadoop | 混合部署 |

## **3.7 端口号**

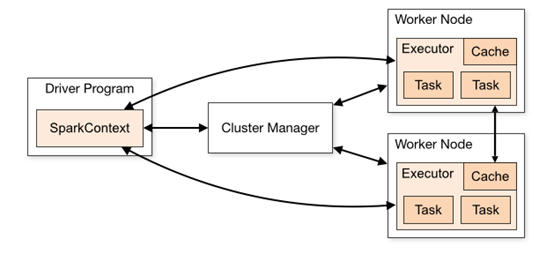
* Spark查看当前Spark-shell运行任务情况端口号：4040（计算）
* Spark Master内部通信服务端口号：7077
* Standalone模式下，Spark Master Web端口号：8080（资源）
* Spark历史服务器端口号：18080
* Hadoop YARN任务运行情况查看端口号：8088

# **Spark运行架构**

## **4.1 运行架构**

Spark框架的核心是一个计算引擎，整体来说，它采用了标准 master-slave 的结构。

如下图所示，它展示了一个 Spark执行时的基本结构。图形中的Driver表示master，负责管理整个集群中的作业任务调度。图形中的Executor 则是 slave，负责实际执行任务。



## **4.2 核心组件**

由上图可以看出，对于Spark框架有两个核心组件：

### **4.2.1 Driver**

Spark驱动器节点，用于执行Spark任务中的main方法，负责实际代码的执行工作。Driver在Spark作业执行时主要负责：

* 将用户程序转化为作业（job）
* 在Executor之间调度任务(task)
* 跟踪Executor的执行情况
* 通过UI展示查询运行情况

实际上，我们无法准确地描述Driver的定义，因为在整个的编程过程中没有看到任何有关Driver的字眼。所以简单理解，所谓的Driver就是驱使整个应用运行起来的程序，也称之为Driver类。

### **4.2.2 Executor**

Spark Executor是集群中工作节点（Worker）中的一个JVM进程，负责在 Spark 作业中运行具体任务（Task），任务彼此之间相互独立。Spark 应用启动时，Executor节点被同时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有Executor节点发生了故障或崩溃，Spark 应用也可以继续执行，会将出错节点上的任务调度到其他Executor节点上继续运行。

Executor有两个核心功能：

* 负责运行组成Spark应用的任务，并将结果返回给驱动器进程
* 它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在Executor进程内的，因此任务可以在运行时充分利用缓存数据加速运算。

### **4.2.3 Master & Worker**

Spark集群的独立部署环境中，不需要依赖其他的资源调度框架，自身就实现了资源调度的功能，所以环境中还有其他两个核心组件：Master和Worker，这里的Master是一个进程，主要负责资源的调度和分配，并进行集群的监控等职责，类似于Yarn环境中的RM, 而Worker呢，也是进程，一个Worker运行在集群中的一台服务器上，由Master分配资源对数据进行并行的处理和计算，类似于Yarn环境中NM。

### **4.2.4 ApplicationMaster**

Hadoop用户向YARN集群提交应用程序时,提交程序中应该包含ApplicationMaster，用于向资源调度器申请执行任务的资源容器Container，运行用户自己的程序任务job，监控整个任务的执行，跟踪整个任务的状态，处理任务失败等异常情况。

说的简单点就是，ResourceManager（资源）和Driver（计算）之间的解耦合靠的就是ApplicationMaster。

## **4.3 核心概念**

### **4.3.1 Executor与Core（核）**

Spark Executor是集群中运行在工作节点（Worker）中的一个JVM进程，是整个集群中的专门用于计算的节点。在提交应用中，可以提供参数指定计算节点的个数，以及对应的资源。这里的资源一般指的是工作节点Executor的内存大小和使用的虚拟CPU核（Core）数量。

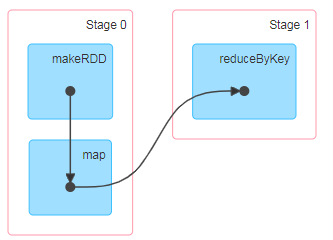
应用程序相关启动参数如下：

|  |  |
| --- | --- |
| 名称 | 说明 |
| --num-executors | 配置Executor的数量 |
| --executor-memory | 配置每个Executor的内存大小 |
| --executor-cores | 配置每个Executor的虚拟CPU core数量 |

### **4.3.2 并行度（**Parallelism**）**

在分布式计算框架中一般都是多个任务同时执行，由于任务分布在不同的计算节点进行计算，所以能够真正地实现多任务并行执行，记住，这里是并行，而不是并发。这里我们将整个集群并行执行任务的数量称之为并行度。那么一个作业到底并行度是多少呢？这个取决于框架的默认配置。应用程序也可以在运行过程中动态修改。

### **4.3.3 有向无环图（**DAG**）**



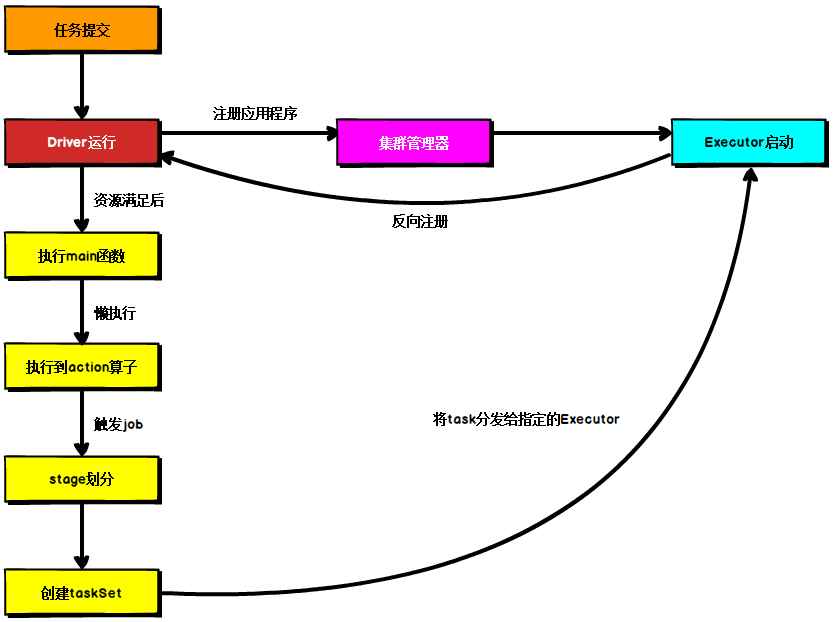
大数据计算引擎框架我们根据使用方式的不同一般会分为四类，其中第一类就是Hadoop所承载的MapReduce,它将计算分为两个阶段，分别为 Map阶段 和 Reduce阶段。对于上层应用来说，就不得不想方设法去拆分算法，甚至于不得不在上层应用实现多个 Job 的串联，以完成一个完整的算法，例如迭代计算。 由于这样的弊端，催生了支持 DAG 框架的产生。因此，支持 DAG 的框架被划分为第二代计算引擎。如 Tez 以及更上层的 Oozie。这里我们不去细究各种 DAG 实现之间的区别，不过对于当时的 Tez 和 Oozie 来说，大多还是批处理的任务。接下来就是以 Spark 为代表的第三代的计算引擎。第三代计算引擎的特点主要是 Job 内部的 DAG 支持（不跨越 Job），以及实时计算。

这里所谓的有向无环图，并不是真正意义的图形，而是由Spark程序直接映射成的数据流的高级抽象模型。简单理解就是将整个程序计算的执行过程用图形表示出来,这样更直观，更便于理解，可以用于表示程序的拓扑结构。

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。

## **4.4 提交流程**

所谓的提交流程，其实就是我们开发人员根据需求写的应用程序通过Spark客户端提交给Spark运行环境执行计算的流程。在不同的部署环境中，这个提交过程基本相同，但是又有细微的区别，我们这里不进行详细的比较，但是因为国内工作中，将Spark引用部署到Yarn环境中会更多一些，所以本课程中的提交流程是基于Yarn环境的。



Spark应用程序提交到Yarn环境中执行的时候，一般会有两种部署执行的方式：Client和Cluster。两种模式主要区别在于：Driver程序的运行节点位置。

### **4.2.1 Yarn Client模式**

Client模式将用于监控和调度的Driver模块在客户端执行，而不是在Yarn中，所以一般用于测试。

* Driver在任务提交的本地机器上运行
* Driver启动后会和ResourceManager通讯申请启动ApplicationMaster
* ResourceManager分配container，在合适的NodeManager上启动ApplicationMaster，负责向ResourceManager申请Executor内存
* ResourceManager接到ApplicationMaster的资源申请后会分配container，然后ApplicationMaster在资源分配指定的NodeManager上启动Executor进程
* Executor进程启动后会向Driver反向注册，Executor全部注册完成后Driver开始执行main函数
* 之后执行到Action算子时，触发一个Job，并根据宽依赖开始划分stage，每个stage生成对应的TaskSet，之后将task分发到各个Executor上执行。

### **4.2.2 Yarn Cluster模式**

Cluster模式将用于监控和调度的Driver模块启动在Yarn集群资源中执行。一般应用于实际生产环境。

* 在YARN Cluster模式下，任务提交后会和ResourceManager通讯申请启动ApplicationMaster，
* 随后ResourceManager分配container，在合适的NodeManager上启动ApplicationMaster，此时的ApplicationMaster就是Driver。
* Driver启动后向ResourceManager申请Executor内存，ResourceManager接到ApplicationMaster的资源申请后会分配container，然后在合适的NodeManager上启动Executor进程
* Executor进程启动后会向Driver反向注册，Executor全部注册完成后Driver开始执行main函数，
* 之后执行到Action算子时，触发一个Job，并根据宽依赖开始划分stage，每个stage生成对应的TaskSet，之后将task分发到各个Executor上执行。

# **Spark核心编程**

Spark计算框架为了能够进行高并发和高吞吐的数据处理，封装了三大数据结构，用于处理不同的应用场景。三大数据结构分别是：

* RDD : 弹性分布式数据集
* 累加器：分布式共享只写变量
* 广播变量：分布式共享只读变量

接下来我们一起看看这三大数据结构是如何在数据处理中使用的。

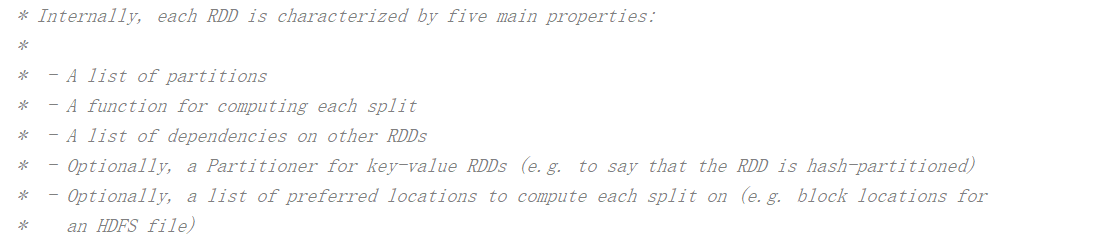
## **5.1 RDD**

### **5.1.1 什么是RDD**

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是Spark中最基本的数据处理模型。代码中是一个抽象类，它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。

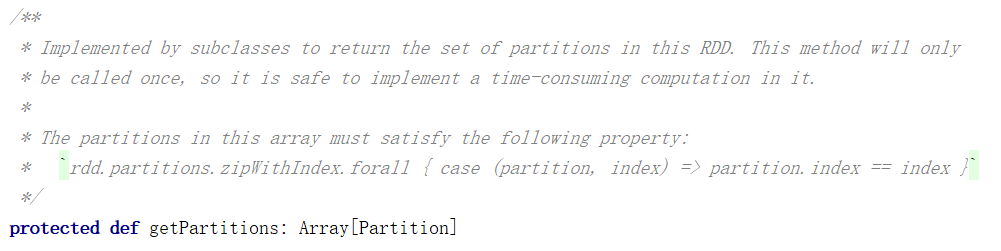
* 弹性
* 存储的弹性：内存与磁盘的自动切换；
* 容错的弹性：数据丢失可以自动恢复；
* 计算的弹性：计算出错重试机制；
* 分片的弹性：可根据需要重新分片。
* 分布式：数据存储在大数据集群不同节点上
* 数据集：RDD封装了计算逻辑，并不保存数据
* 数据抽象：RDD是一个抽象类，需要子类具体实现
* 不可变：RDD封装了计算逻辑，是不可以改变的，想要改变，只能产生新的RDD，在新的RDD里面封装计算逻辑
* 可分区、并行计算

### **5.1.2 核心属性**



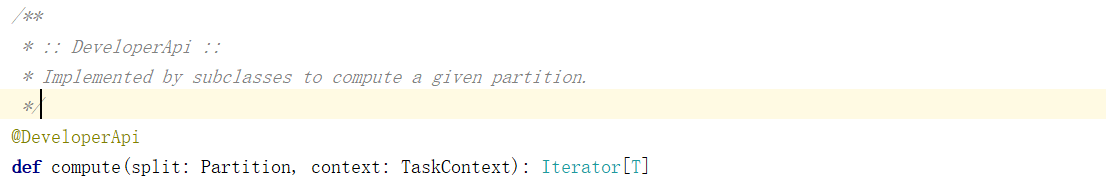
* 分区列表

RDD数据结构中存在分区列表，用于执行任务时并行计算，是实现分布式计算的重要属性。



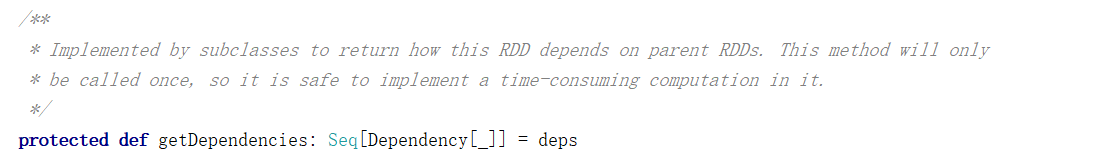
* 分区计算函数

Spark在计算时，是使用分区函数对每一个分区进行计算



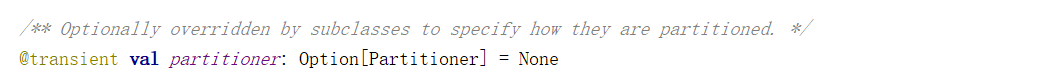
* RDD之间的依赖关系

RDD是计算模型的封装，当需求中需要将多个计算模型进行组合时，就需要将多个RDD建立依赖关系



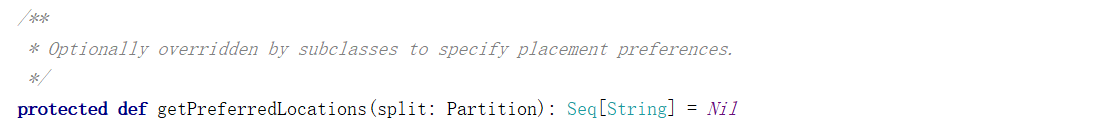
* 分区器（可选）

当数据为KV类型数据时，可以通过设定分区器自定义数据的分区



* 首选位置（可选）

计算数据时，可以根据计算节点的状态选择不同的节点位置进行计算



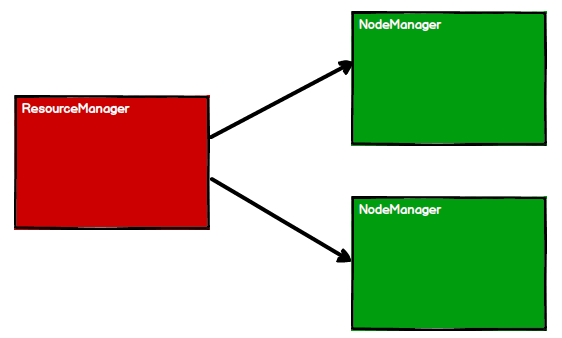
### **5.1.3 执行原理**

从计算的角度来讲，数据处理过程中需要计算资源（内存 & CPU）和计算模型（逻辑）。执行时，需要将计算资源和计算模型进行协调和整合。

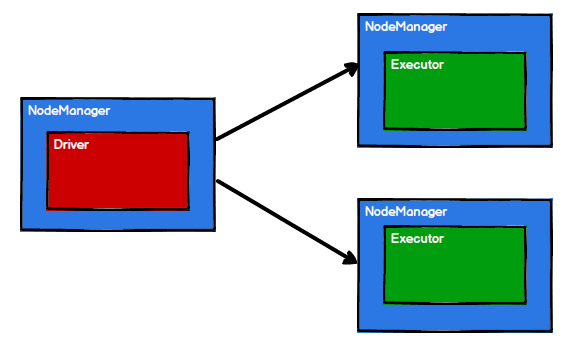
Spark框架在执行时，先申请资源，然后将应用程序的数据处理逻辑分解成一个一个的计算任务。然后将任务发到已经分配资源的计算节点上, 按照指定的计算模型进行数据计算。最后得到计算结果。

RDD是Spark框架中用于数据处理的核心模型，接下来我们看看，在Yarn环境中，RDD的工作原理:

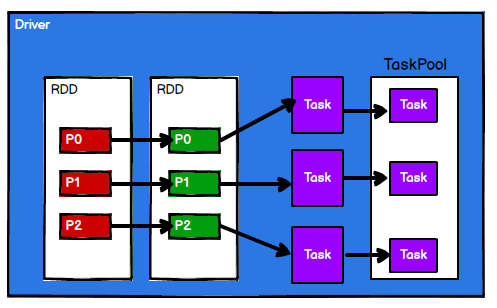
1. 启动Yarn集群环境



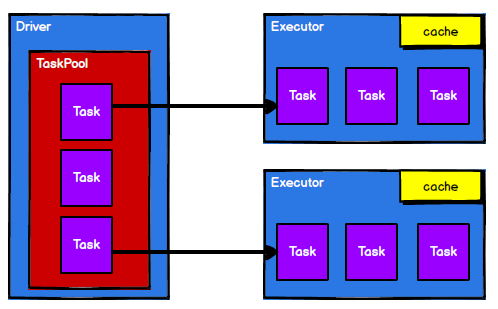
1. Spark通过申请资源创建调度节点和计算节点



1. Spark框架根据需求将计算逻辑根据分区划分成不同的任务



1. 调度节点将任务根据计算节点状态发送到对应的计算节点进行计算



从以上流程可以看出RDD在整个流程中主要用于将逻辑进行封装，并生成Task发送给Executor节点执行计算，接下来我们就一起看看Spark框架中RDD是具体是如何进行数据处理的。

### **5.1.4 基础编程**

#### 5.1.4.1 RDD创建

在Spark中创建RDD的创建方式可以分为四种：

1. **从集合（内存）中创建RDD**

从集合中创建RDD，[Spark](https://www.iteblog.com/archives/tag/spark/" \t "_blank)主要提供了两个方法：parallelize和makeRDD

val sparkConf =

new SparkConf().setMaster("local[\*]").setAppName("spark")

val sparkContext = new SparkContext(sparkConf)

val rdd1 = sparkContext.parallelize(

List(1,2,3,4)

)

val rdd2 = sparkContext.makeRDD(

List(1,2,3,4)

)

rdd1.collect().foreach(println)

rdd2.collect().foreach(println)

sparkContext.stop()

从底层代码实现来讲，makeRDD方法其实就是parallelize方法

def makeRDD[T: ClassTag](

seq: Seq[T],

numSlices: Int = defaultParallelism): RDD[T] = withScope {

parallelize(seq, numSlices)

}

1. **从外部存储（文件）创建RDD**

由外部存储系统的数据集创建RDD包括：本地的文件系统，所有Hadoop支持的数据集，比如HDFS、HBase等。

val sparkConf =

new SparkConf().setMaster("local[\*]").setAppName("spark")

val sparkContext = new SparkContext(sparkConf)

val fileRDD: RDD[String] = sparkContext.textFile("input")

fileRDD.collect().foreach(println)

sparkContext.stop()

1. **从其他RDD创建**

主要是通过一个RDD运算完后，再产生新的RDD。详情请参考后续章节

1. **直接创建RDD（new）**

使用new的方式直接构造RDD，一般由Spark框架自身使用。

#### 5.1.4.2 RDD并行度与分区

默认情况下，Spark可以将一个作业切分多个任务后，发送给Executor节点并行计算，而能够并行计算的任务数量我们称之为并行度。这个数量可以在构建RDD时指定。记住，这里的并行执行的任务数量，并不是指的切分任务的数量，不要混淆了。

val sparkConf =

new SparkConf().setMaster("local[\*]").setAppName("spark")

val sparkContext = new SparkContext(sparkConf)

val dataRDD: RDD[Int] =

sparkContext.makeRDD(

List(1,2,3,4),

4)

val fileRDD: RDD[String] =

sparkContext.textFile(

"input",

2)

fileRDD.collect().foreach(println)

sparkContext.stop()

* 读取内存数据时，数据可以按照并行度的设定进行数据的分区操作，数据分区规则的Spark核心源码如下：

def positions(length: Long, numSlices: Int): Iterator[(Int, Int)] = {

(0 until numSlices).iterator.map { i =>

val start = ((i \* length) / numSlices).toInt

val end = (((i + 1) \* length) / numSlices).toInt

(start, end)

}

}

* 读取文件数据时，数据是按照Hadoop文件读取的规则进行切片分区，而切片规则和数据读取的规则有些差异，具体Spark核心源码如下

public InputSplit[] getSplits(JobConf job, int numSplits)

throws IOException {

long totalSize = 0; // compute total size

for (FileStatus file: files) { // check we have valid files

if (file.isDirectory()) {

throw new IOException("Not a file: "+ file.getPath());

}

totalSize += file.getLen();

}

long goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);

long minSize = Math.max(job.getLong(org.apache.hadoop.mapreduce.lib.input.

FileInputFormat.SPLIT\_MINSIZE, 1), minSplitSize);

...

for (FileStatus file: files) {

...

if (isSplitable(fs, path)) {

long blockSize = file.getBlockSize();

long splitSize = computeSplitSize(goalSize, minSize, blockSize);

...

}

protected long computeSplitSize(long goalSize, long minSize,

long blockSize) {

return Math.max(minSize, Math.min(goalSize, blockSize));

}

#### 5.1.4.3 RDD转换算子

RDD根据数据处理方式的不同将算子整体上分为Value类型、双Value类型和Key-Value类型

* **Value类型**

##### map

* 函数签名

def map[U: ClassTag](f: T => U): RDD[U]

* 函数说明

将处理的数据逐条进行映射转换，这里的转换可以是类型的转换，也可以是值的转换。

val dataRDD: RDD[Int] = sparkContext.makeRDD(List(1,2,3,4))

val dataRDD1: RDD[Int] = dataRDD.map(

num => {

num \* 2

}

)

val dataRDD2: RDD[String] = dataRDD1.map(

num => {

"" + num

}

)

* 小功能：从服务器日志数据apache.log中获取用户请求URL资源路径

##### mapPartitions

* 函数签名

def mapPartitions[U: ClassTag](

f: Iterator[T] => Iterator[U],

preservesPartitioning: Boolean = false): RDD[U]

* 函数说明

将待处理的数据以分区为单位发送到计算节点进行处理，这里的处理是指可以进行任意的处理，哪怕是过滤数据。

val dataRDD1: RDD[Int] = dataRDD.mapPartitions(

datas => {

datas.filter(\_==2)

}

)

* 小功能：获取每个数据分区的最大值

**** 思考一个问题：map和mapPartitions的区别？

* **数据处理角度**

Map算子是分区内一个数据一个数据的执行，类似于串行操作。而mapPartitions算子是以分区为单位进行批处理操作。

* **功能的角度**

Map算子主要目的将数据源中的数据进行转换和改变。但是不会减少或增多数据。MapPartitions算子需要传递一个迭代器，返回一个迭代器，没有要求的元素的个数保持不变，所以可以增加或减少数据

* **性能的角度**

Map算子因为类似于串行操作，所以性能比较低，而是mapPartitions算子类似于批处理，所以性能较高。但是mapPartitions算子会长时间占用内存，那么这样会导致内存可能不够用，出现内存溢出的错误。所以在内存有限的情况下，不推荐使用。使用map操作。

**完成比完美更重要**

##### mapPartitionsWithIndex

* 函数签名

def mapPartitionsWithIndex[U: ClassTag](

f: (Int, Iterator[T]) => Iterator[U],

preservesPartitioning: Boolean = false): RDD[U]

* 函数说明

将待处理的数据以分区为单位发送到计算节点进行处理，这里的处理是指可以进行任意的处理，哪怕是过滤数据，在处理时同时可以获取当前分区索引。

val dataRDD1 = dataRDD.mapPartitionsWithIndex(

(index, datas) => {

datas.map(index, \_)

}

)

* 小功能：获取第二个数据分区的数据

##### flatMap

* 函数签名

def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]

* 函数说明

将处理的数据进行扁平化后再进行映射处理，所以算子也称之为扁平映射

val dataRDD = sparkContext.makeRDD(List(

List(1,2),List(3,4)

),1)

val dataRDD1 = dataRDD.flatMap(

list => list

)

* 小功能：将List(List(1,2),3,List(4,5))进行扁平化操作

##### glom

* 函数签名

def glom(): RDD[Array[T]]

* 函数说明

将同一个分区的数据直接转换为相同类型的内存数组进行处理，分区不变

val dataRDD = sparkContext.makeRDD(List(

1,2,3,4

),1)

val dataRDD1:RDD[Array[Int]] = dataRDD.glom()

* 小功能：计算所有分区最大值求和（分区内取最大值，分区间最大值求和）

##### groupBy

* 函数签名

def groupBy[K](f: T => K)(implicit kt: ClassTag[K]): RDD[(K, Iterable[T])]

* 函数说明

将数据根据指定的规则进行分组, 分区默认不变，但是数据会被打乱重新组合，我们将这样的操作称之为shuffle。极限情况下，数据可能被分在同一个分区中

一个组的数据在一个分区中，但是并不是说一个分区中只有一个组

val dataRDD = sparkContext.makeRDD(List(1,2,3,4),1)

val dataRDD1 = dataRDD.groupBy(

\_%2

)

* 小功能：将List("Hello", "hive", "hbase", "Hadoop")根据单词首写字母进行分组。
* 小功能：从服务器日志数据apache.log中获取每个时间段访问量。
* 小功能：WordCount。

##### filter

* 函数签名

def filter(f: T => Boolean): RDD[T]

* 函数说明

将数据根据指定的规则进行筛选过滤，符合规则的数据保留，不符合规则的数据丢弃。

当数据进行筛选过滤后，分区不变，但是分区内的数据可能不均衡，生产环境下，可能会出现数据倾斜。

val dataRDD = sparkContext.makeRDD(List(

1,2,3,4

),1)

val dataRDD1 = dataRDD.filter(\_%2 == 0)

* 小功能：从服务器日志数据apache.log中获取2015年5月17日的请求路径

##### sample

* 函数签名

def sample(

withReplacement: Boolean,

fraction: Double,

seed: Long = Utils.random.nextLong): RDD[T]

* 函数说明

根据指定的规则从数据集中抽取数据

val dataRDD = sparkContext.makeRDD(List(

1,2,3,4

),1)

// 抽取数据不放回（伯努利算法）

// 伯努利算法：又叫0、1分布。例如扔硬币，要么正面，要么反面。

// 具体实现：根据种子和随机算法算出一个数和第二个参数设置几率比较，小于第二个参数要，大于不要

// 第一个参数：抽取的数据是否放回，false：不放回

// 第二个参数：抽取的几率，范围在[0,1]之间,0：全不取；1：全取；

// 第三个参数：随机数种子

val dataRDD1 = dataRDD.sample(false, 0.5)

// 抽取数据放回（泊松算法）

// 第一个参数：抽取的数据是否放回，true：放回；false：不放回

// 第二个参数：重复数据的几率，范围大于等于0.表示每一个元素被期望抽取到的次数

// 第三个参数：随机数种子

val dataRDD2 = dataRDD.sample(true, 2)

****思考一个问题：有啥用，抽奖吗？

##### distinct

* 函数签名

def distinct()(implicit ord: Ordering[T] = null): RDD[T]

def distinct(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]

* 函数说明

将数据集中重复的数据去重

val dataRDD = sparkContext.makeRDD(List(

1,2,3,4,1,2

),1)

val dataRDD1 = dataRDD.distinct()

val dataRDD2 = dataRDD.distinct(2)

****思考一个问题：如果不用该算子，你有什么办法实现数据去重？

##### coalesce

* 函数签名

def coalesce(numPartitions: Int, shuffle: Boolean = false,

partitionCoalescer: Option[PartitionCoalescer] = Option.empty)

(implicit ord: Ordering[T] = null)

: RDD[T]

* 函数说明

根据数据量缩减分区，用于大数据集过滤后，提高小数据集的执行效率

当spark程序中，存在过多的小任务的时候，可以通过coalesce方法，收缩合并分区，减少分区的个数，减小任务调度成本

val dataRDD = sparkContext.makeRDD(List(

1,2,3,4,1,2

),6)

val dataRDD1 = dataRDD.coalesce(2)

****思考一个问题：我想要扩大分区，怎么办？

##### repartition

* 函数签名

def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]

* 函数说明

该操作内部其实执行的是coalesce操作，参数shuffle的默认值为true。无论是将分区数多的RDD转换为分区数少的RDD，还是将分区数少的RDD转换为分区数多的RDD，repartition操作都可以完成，因为无论如何都会经shuffle过程。

val dataRDD = sparkContext.makeRDD(List(

1,2,3,4,1,2

),2)

val dataRDD1 = dataRDD.repartition(4)

****思考一个问题：coalesce和repartition区别？

##### sortBy

* 函数签名

def sortBy[K](

f: (T) => K,

ascending: Boolean = true,

numPartitions: Int = this.partitions.length)

(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]

* 函数说明

该操作用于排序数据。在排序之前，可以将数据通过f函数进行处理，之后按照f函数处理的结果进行排序，默认为升序排列。排序后新产生的RDD的分区数与原RDD的分区数一致。中间存在shuffle的过程

val dataRDD = sparkContext.makeRDD(List(

1,2,3,4,1,2

),2)

val dataRDD1 = dataRDD.sortBy(num=>num, false, 4)

* **双Value类型**

##### intersection

* 函数签名

def intersection(other: RDD[T]): RDD[T]

* 函数说明

对源RDD和参数RDD求交集后返回一个新的RDD

val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))

val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.intersection(dataRDD2)

****思考一个问题：如果两个RDD数据类型不一致怎么办？

##### union

* 函数签名

def union(other: RDD[T]): RDD[T]

* 函数说明

对源RDD和参数RDD求并集后返回一个新的RDD

val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))

val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.union(dataRDD2)

****思考一个问题：如果两个RDD数据类型不一致怎么办？

##### subtract

* 函数签名

def subtract(other: RDD[T]): RDD[T]

* 函数说明

以一个RDD元素为主，去除两个RDD中重复元素，将其他元素保留下来。求差集

val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))

val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.subtract(dataRDD2)

****思考一个问题：如果两个RDD数据类型不一致怎么办？

##### zip

* 函数签名

def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]

* 函数说明

将两个RDD中的元素，以键值对的形式进行合并。其中，键值对中的Key为第1个RDD中的元素，Value为第2个RDD中的相同位置的元素。

val dataRDD1 = sparkContext.makeRDD(List(1,2,3,4))

val dataRDD2 = sparkContext.makeRDD(List(3,4,5,6))

val dataRDD = dataRDD1.zip(dataRDD2)

****思考一个问题：如果两个RDD数据类型不一致怎么办？

****思考一个问题：如果两个RDD数据分区不一致怎么办？

****思考一个问题：如果两个RDD分区数据数量不一致怎么办？

* **Key - Value类型**

##### partitionBy

* 函数签名

def partitionBy(partitioner: Partitioner): RDD[(K, V)]

* 函数说明

将数据按照指定Partitioner重新进行分区。Spark默认的分区器是HashPartitioner

val rdd: RDD[(Int, String)] =

sc.makeRDD(Array((1,"aaa"),(2,"bbb"),(3,"ccc")),3)

import org.apache.spark.HashPartitioner

val rdd2: RDD[(Int, String)] =

rdd.partitionBy(new HashPartitioner(2))

****思考一个问题：如果重分区的分区器和当前RDD的分区器一样怎么办？

****思考一个问题：Spark还有其他分区器吗？

****思考一个问题：如果想按照自己的方法进行数据分区怎么办？

****思考一个问题：哪那么多问题？

##### reduceByKey

* 函数签名

def reduceByKey(func: (V, V) => V): RDD[(K, V)]

def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]

* 函数说明

可以将数据按照相同的Key对Value进行聚合

val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val dataRDD2 = dataRDD1.reduceByKey(\_+\_)

val dataRDD3 = dataRDD1.reduceByKey(\_+\_, 2)

* 小功能：WordCount

##### groupByKey

* 函数签名

def groupByKey(): RDD[(K, Iterable[V])]

def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]

def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]

* 函数说明

将数据源的数据根据key对value进行分组

val dataRDD1 =

sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val dataRDD2 = dataRDD1.groupByKey()

val dataRDD3 = dataRDD1.groupByKey(2)

val dataRDD4 = dataRDD1.groupByKey(new HashPartitioner(2))

****思考一个问题：reduceByKey和groupByKey的区别？

**从shuffle的角度**：reduceByKey和groupByKey都存在shuffle的操作，但是reduceByKey可以在shuffle前对分区内相同key的数据进行预聚合（combine）功能，这样会减少落盘的数据量，而groupByKey只是进行分组，不存在数据量减少的问题，reduceByKey性能比较高。

**从功能的角度**：reduceByKey其实包含分组和聚合的功能。groupByKey只能分组，不能聚合，所以在分组聚合的场合下，推荐使用reduceByKey，如果仅仅是分组而不需要聚合。那么还是只能使用groupByKey

* 小功能：WordCount

##### aggregateByKey

* 函数签名

def aggregateByKey[U: ClassTag](zeroValue: U)(seqOp: (U, V) => U,

combOp: (U, U) => U): RDD[(K, U)]

* 函数说明

将数据根据不同的规则进行分区内计算和分区间计算

val dataRDD1 =

sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val dataRDD2 =

dataRDD1.aggregateByKey(0)(\_+\_,\_+\_)

* 取出每个分区内相同key的最大值然后分区间相加

// TODO : 取出每个分区内相同key的最大值然后分区间相加

// aggregateByKey算子是函数柯里化，存在两个参数列表

// 1. 第一个参数列表中的参数表示初始值

// 2. 第二个参数列表中含有两个参数

// 2.1 第一个参数表示分区内的计算规则

// 2.2 第二个参数表示分区间的计算规则

val rdd =

sc.makeRDD(List(

("a",1),("a",2),("c",3),

("b",4),("c",5),("c",6)

),2)

// 0:("a",1),("a",2),("c",3) => (a,10)(c,10)

// => (a,10)(b,10)(c,20)

// 1:("b",4),("c",5),("c",6) => (b,10)(c,10)

val resultRDD =

rdd.aggregateByKey(10)(

(x, y) => math.max(x,y),

(x, y) => x + y

)

resultRDD.collect().foreach(println)

****思考一个问题：分区内计算规则和分区间计算规则相同怎么办？（WordCount）

##### foldByKey

* 函数签名

def foldByKey(zeroValue: V)(func: (V, V) => V): RDD[(K, V)]

* 函数说明

当分区内计算规则和分区间计算规则相同时，aggregateByKey就可以简化为foldByKey

val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val dataRDD2 = dataRDD1.foldByKey(0)(\_+\_)

##### combineByKey

* 函数签名

def combineByKey[C](

createCombiner: V => C,

mergeValue: (C, V) => C,

mergeCombiners: (C, C) => C): RDD[(K, C)]

* 函数说明

最通用的对key-value型rdd进行聚集操作的聚集函数（aggregation function）。类似于aggregate()，combineByKey()允许用户返回值的类型与输入不一致。

小练习：将数据List(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98))求每个key的平均值

val list: List[(String, Int)] = List(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98))

val input: RDD[(String, Int)] = sc.makeRDD(list, 2)

val combineRdd: RDD[(String, (Int, Int))] = input.combineByKey(

(\_, 1),

(acc: (Int, Int), v) => (acc.\_1 + v, acc.\_2 + 1),

(acc1: (Int, Int), acc2: (Int, Int)) => (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2)

)

****思考一个问题：reduceByKey、foldByKey、aggregateByKey、combineByKey的区别？

reduceByKey: 相同key的第一个数据不进行任何计算，分区内和分区间计算规则相同

foldByKey: 相同key的第一个数据和初始值进行分区内计算，分区内和分区间计算规则相同

aggregateByKey：相同key的第一个数据和初始值进行分区内计算，分区内和分区间计算规则可以不相同

combineByKey:当计算时，发现数据结构不满足要求时，可以让第一个数据转换结构。分区内和分区间计算规则不相同。

##### sortByKey

* 函数签名

def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length)

: RDD[(K, V)]

* 函数说明

在一个(K,V)的RDD上调用，K必须实现Ordered接口(特质)，返回一个按照key进行排序的

val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(true)

val sortRDD1: RDD[(String, Int)] = dataRDD1.sortByKey(false)

* 小功能：设置key为自定义类User

##### join

* 函数签名

def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]

* 函数说明

在类型为(K,V)和(K,W)的RDD上调用，返回一个相同key对应的所有元素连接在一起的(K,(V,W))的RDD

val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1, "a"), (2, "b"), (3, "c")))

val rdd1: RDD[(Int, Int)] = sc.makeRDD(Array((1, 4), (2, 5), (3, 6)))

rdd.join(rdd1).collect().foreach(println)

****思考一个问题：如果key存在不相等呢？

##### leftOuterJoin

* 函数签名

def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]

* 函数说明

类似于SQL语句的左外连接

val dataRDD1 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val dataRDD2 = sparkContext.makeRDD(List(("a",1),("b",2),("c",3)))

val rdd: RDD[(String, (Int, Option[Int]))] = dataRDD1.leftOuterJoin(dataRDD2)

##### cogroup

* 函数签名

def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]

* 函数说明

在类型为(K,V)和(K,W)的RDD上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的RDD

val dataRDD1 = sparkContext.makeRDD(List(("a",1),("a",2),("c",3)))

val dataRDD2 = sparkContext.makeRDD(List(("a",1),("c",2),("c",3)))

val value: RDD[(String, (Iterable[Int], Iterable[Int]))] =

dataRDD1.cogroup(dataRDD2)

#### 5.1.4.4 案例实操

1. 数据准备

agent.log：时间戳，省份，城市，用户，广告，中间字段使用空格分隔。

1. 需求描述

统计出每一个省份**每个广告被点击数量**排行的Top3

1. 需求分析
2. 功能实现

#### 5.1.4.5 RDD行动算子

##### reduce

* 函数签名

def reduce(f: (T, T) => T): T

* 函数说明

聚集RDD中的所有元素，先聚合分区内数据，再聚合分区间数据

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 聚合数据

val reduceResult: Int = rdd.reduce(\_+\_)

##### collect

* 函数签名

def collect(): Array[T]

* 函数说明

在驱动程序（Driver）中，以数组Array的形式返回数据集的所有元素

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 收集数据到Driver

rdd.collect().foreach(println)

##### count

* 函数签名

def count(): Long

* 函数说明

返回RDD中元素的个数

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回RDD中元素的个数

val countResult: Long = rdd.count()

##### first

* 函数签名

def first(): T

* 函数说明

返回RDD中的第一个元素

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回RDD中元素的个数

val firstResult: Int = rdd.first()

println(firstResult)

##### take

* 函数签名

def take(num: Int): Array[T]

* 函数说明

返回一个由RDD的前n个元素组成的数组

vval rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 返回RDD中元素的个数

val takeResult: Array[Int] = rdd.take(2)

println(takeResult.mkString(","))

##### takeOrdered

* 函数签名

def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]

* 函数说明

返回该RDD排序后的前n个元素组成的数组

val rdd: RDD[Int] = sc.makeRDD(List(1,3,2,4))

// 返回RDD中元素的个数

val result: Array[Int] = rdd.takeOrdered(2)

##### aggregate

* 函数签名

def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U

* 函数说明

分区的数据通过初始值和分区内的数据进行聚合，然后再和初始值进行分区间的数据聚合

val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4), 8)

// 将该RDD所有元素相加得到结果

//val result: Int = rdd.aggregate(0)(\_ + \_, \_ + \_)

val result: Int = rdd.aggregate(10)(\_ + \_, \_ + \_)

##### fold

* 函数签名

def fold(zeroValue: T)(op: (T, T) => T): T

* 函数说明

折叠操作，aggregate的简化版操作

val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4))

val foldResult: Int = rdd.fold(0)(\_+\_)

##### countByKey

* 函数签名

def countByKey(): Map[K, Long]

* 函数说明

统计每种key的个数

val rdd: RDD[(Int, String)] = sc.makeRDD(List((1, "a"), (1, "a"), (1, "a"), (2, "b"), (3, "c"), (3, "c")))

// 统计每种key的个数

val result: collection.Map[Int, Long] = rdd.countByKey()

##### save相关算子

* 函数签名

def saveAsTextFile(path: String): Unit

def saveAsObjectFile(path: String): Unit

def saveAsSequenceFile(

path: String,

codec: Option[Class[\_ <: CompressionCodec]] = None): Unit

* 函数说明

将数据保存到不同格式的文件中

// 保存成Text文件

rdd.saveAsTextFile("output")

// 序列化成对象保存到文件

rdd.saveAsObjectFile("output1")

// 保存成Sequencefile文件

rdd.map((\_,1)).saveAsSequenceFile("output2")

##### foreach

* 函数签名

def foreach(f: T => Unit): Unit = withScope {

val cleanF = sc.clean(f)

sc.runJob(this, (iter: Iterator[T]) => iter.foreach(cleanF))

}

* 函数说明

分布式遍历RDD中的每一个元素，调用指定函数

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

// 收集后打印

rdd.map(num=>num).collect().foreach(println)

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

// 分布式打印

rdd.foreach(println)

#### 5.1.4.6 RDD序列化

1. 闭包检查

从计算的角度, 算子以外的代码都是在Driver端执行, 算子里面的代码都是在Executor端执行。那么在scala的函数式编程中，就会导致算子内经常会用到算子外的数据，这样就形成了闭包的效果，如果使用的算子外的数据无法序列化，就意味着无法传值给Executor端执行，就会发生错误，所以需要在执行任务计算前，检测闭包内的对象是否可以进行序列化，这个操作我们称之为闭包检测。Scala2.12版本后闭包编译方式发生了改变

1. 序列化方法和属性

从计算的角度, 算子以外的代码都是在Driver端执行, 算子里面的代码都是在Executor端执行，看如下代码：

object serializable02\_function {

def main(args: Array[String]): Unit = {

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.创建一个RDD

val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello spark", "hive", "atguigu"))

//3.1创建一个Search对象

val search = new Search("hello")

//3.2 函数传递，打印：ERROR Task not serializable

search.getMatch1(rdd).collect().foreach(println)

//3.3 属性传递，打印：ERROR Task not serializable

search.getMatch2(rdd).collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

class Search(query:String) extends Serializable {

def isMatch(s: String): Boolean = {

s.contains(query)

}

// 函数序列化案例

def getMatch1 (rdd: RDD[String]): RDD[String] = {

//rdd.filter(this.isMatch)

rdd.filter(isMatch)

}

// 属性序列化案例

def getMatch2(rdd: RDD[String]): RDD[String] = {

//rdd.filter(x => x.contains(this.query))

rdd.filter(x => x.contains(query))

//val q = query

//rdd.filter(x => x.contains(q))

}

}

1. Kryo序列化框架

参考地址: https://github.com/EsotericSoftware/kryo

Java的序列化能够序列化任何的类。但是比较重（字节多），序列化后，对象的提交也比较大。Spark出于性能的考虑，Spark2.0开始支持另外一种Kryo序列化机制。Kryo速度是Serializable的10倍。当RDD在Shuffle数据的时候，简单数据类型、数组和字符串类型已经在Spark内部使用Kryo来序列化。

注意：即使使用Kryo序列化，也要继承Serializable接口。

object serializable\_Kryo {

def main(args: Array[String]): Unit = {

val conf: SparkConf = new SparkConf()

.setAppName("SerDemo")

.setMaster("local[\*]")

// 替换默认的序列化机制

.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

// 注册需要使用 kryo 序列化的自定义类

.registerKryoClasses(Array(classOf[Searcher]))

val sc = new SparkContext(conf)

val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello atguigu", "atguigu", "hahah"), 2)

val searcher = new Searcher("hello")

val result: RDD[String] = searcher.getMatchedRDD1(rdd)

result.collect.foreach(println)

}

}

case class Searcher(val query: String) {

def isMatch(s: String) = {

s.contains(query)

}

def getMatchedRDD1(rdd: RDD[String]) = {

rdd.filter(isMatch)

}

def getMatchedRDD2(rdd: RDD[String]) = {

val q = query

rdd.filter(\_.contains(q))

}

}

#### 5.1.4.7 RDD依赖关系

1. **RDD 血缘关系**

RDD只支持粗粒度转换，即在大量记录上执行的单个操作。将创建RDD的一系列Lineage（血统）记录下来，以便恢复丢失的分区。RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

val fileRDD: RDD[String] = sc.textFile("input/1.txt")

println(fileRDD.toDebugString)

println("----------------------")

val wordRDD: RDD[String] = fileRDD.flatMap(\_.split(" "))

println(wordRDD.toDebugString)

println("----------------------")

val mapRDD: RDD[(String, Int)] = wordRDD.map((\_,1))

println(mapRDD.toDebugString)

println("----------------------")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(\_+\_)

println(resultRDD.toDebugString)

resultRDD.collect()

1. **RDD 依赖关系**

这里所谓的依赖关系，其实就是两个相邻RDD之间的关系

val sc: SparkContext = new SparkContext(conf)

val fileRDD: RDD[String] = sc.textFile("input/1.txt")

println(fileRDD.dependencies)

println("----------------------")

val wordRDD: RDD[String] = fileRDD.flatMap(\_.split(" "))

println(wordRDD.dependencies)

println("----------------------")

val mapRDD: RDD[(String, Int)] = wordRDD.map((\_,1))

println(mapRDD.dependencies)

println("----------------------")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(\_+\_)

println(resultRDD.dependencies)

resultRDD.collect()

1. **RDD 窄依赖**

窄依赖表示每一个父(上游)RDD的Partition最多被子（下游）RDD的一个Partition使用，窄依赖我们形象的比喻为独生子女。

class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd)

1. **RDD 宽依赖**

宽依赖表示同一个父（上游）RDD的Partition被多个子（下游）RDD的Partition依赖，会引起Shuffle，总结：宽依赖我们形象的比喻为多生。

class ShuffleDependency[K: ClassTag, V: ClassTag, C: ClassTag](

@transient private val \_rdd: RDD[\_ <: Product2[K, V]],

val partitioner: Partitioner,

val serializer: Serializer = SparkEnv.get.serializer,

val keyOrdering: Option[Ordering[K]] = None,

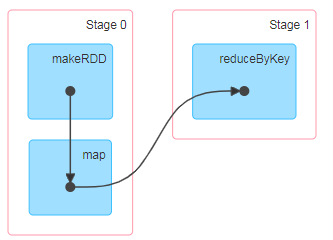
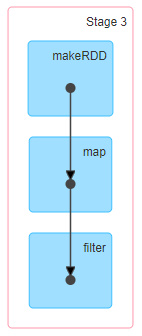
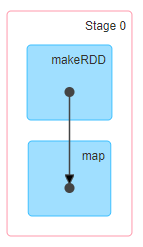
val aggregator: Option[Aggregator[K, V, C]] = None,

val mapSideCombine: Boolean = false)

extends Dependency[Product2[K, V]]

1. **RDD 阶段划分**

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。例如，DAG记录了RDD的转换过程和任务的阶段。

1. **RDD 阶段划分源码**

try {

// New stage creation may throw an exception if, for example, jobs are run on a

// HadoopRDD whose underlying HDFS files have been deleted.

finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)

} catch {

case e: Exception =>

logWarning("Creating new stage failed due to exception - job: " + jobId, e)

listener.jobFailed(e)

return

}

……

private def createResultStage(

rdd: RDD[\_],

func: (TaskContext, Iterator[\_]) => \_,

partitions: Array[Int],

jobId: Int,

callSite: CallSite): ResultStage = {

val parents = getOrCreateParentStages(rdd, jobId)

val id = nextStageId.getAndIncrement()

val stage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)

stageIdToStage(id) = stage

updateJobIdStageIdMaps(jobId, stage)

stage

}

……

private def getOrCreateParentStages(rdd: RDD[\_], firstJobId: Int): List[Stage] = {

getShuffleDependencies(rdd).map { shuffleDep =>

getOrCreateShuffleMapStage(shuffleDep, firstJobId)

}.toList

}

……

private[scheduler] def getShuffleDependencies(

rdd: RDD[\_]): HashSet[ShuffleDependency[\_, \_, \_]] = {

val parents = new HashSet[ShuffleDependency[\_, \_, \_]]

val visited = new HashSet[RDD[\_]]

val waitingForVisit = new Stack[RDD[\_]]

waitingForVisit.push(rdd)

while (waitingForVisit.nonEmpty) {

val toVisit = waitingForVisit.pop()

if (!visited(toVisit)) {

visited += toVisit

toVisit.dependencies.foreach {

case shuffleDep: ShuffleDependency[\_, \_, \_] =>

parents += shuffleDep

case dependency =>

waitingForVisit.push(dependency.rdd)

}

}

}

parents

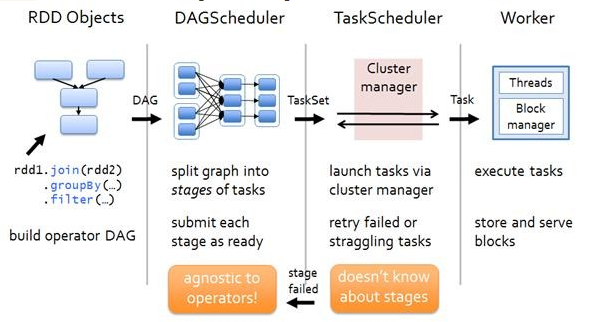
}

1. **RDD 任务划分**

RDD任务切分中间分为：Application、Job、Stage和Task

* Application：初始化一个SparkContext即生成一个Application；
* Job：一个Action算子就会生成一个Job；
* Stage：Stage等于宽依赖(ShuffleDependency)的个数加1；
* Task：一个Stage阶段中，最后一个RDD的分区个数就是Task的个数。

注意：Application->Job->Stage->Task每一层都是1对n的关系。



1. **RDD 任务划分源码**

val tasks: Seq[Task[\_]] = try {

stage match {

case stage: ShuffleMapStage =>

partitionsToCompute.map { id =>

val locs = taskIdToLocations(id)

val part = stage.rdd.partitions(id)

new ShuffleMapTask(stage.id, stage.latestInfo.attemptId,

taskBinary, part, locs, stage.latestInfo.taskMetrics, properties, Option(jobId),

Option(sc.applicationId), sc.applicationAttemptId)

}

case stage: ResultStage =>

partitionsToCompute.map { id =>

val p: Int = stage.partitions(id)

val part = stage.rdd.partitions(p)

val locs = taskIdToLocations(id)

new ResultTask(stage.id, stage.latestInfo.attemptId,

taskBinary, part, locs, id, properties, stage.latestInfo.taskMetrics,

Option(jobId), Option(sc.applicationId), sc.applicationAttemptId)

}

}

……

val partitionsToCompute: Seq[Int] = stage.findMissingPartitions()

……

override def findMissingPartitions(): Seq[Int] = {

mapOutputTrackerMaster

.findMissingPartitions(shuffleDep.shuffleId)

.getOrElse(0 until numPartitions)

}

#### 5.1.4.8 RDD持久化

1. **RDD Cache缓存**

RDD通过Cache或者Persist方法将前面的计算结果缓存，默认情况下会把数据以缓存在JVM的堆内存中。但是并不是这两个方法被调用时立即缓存，而是触发后面的action算子时，该RDD将会被缓存在计算节点的内存中，并供后面重用。

// cache操作会增加血缘关系，不改变原有的血缘关系

println(wordToOneRdd.toDebugString)

// 数据缓存。

wordToOneRdd.cache()

// 可以更改存储级别

//mapRdd.persist(StorageLevel.MEMORY\_AND\_DISK\_2)

存储级别

object StorageLevel {

val NONE = new StorageLevel(false, false, false, false)

val DISK\_ONLY = new StorageLevel(true, false, false, false)

val DISK\_ONLY\_2 = new StorageLevel(true, false, false, false, 2)

val MEMORY\_ONLY = new StorageLevel(false, true, false, true)

val MEMORY\_ONLY\_2 = new StorageLevel(false, true, false, true, 2)

val MEMORY\_ONLY\_SER = new StorageLevel(false, true, false, false)

val MEMORY\_ONLY\_SER\_2 = new StorageLevel(false, true, false, false, 2)

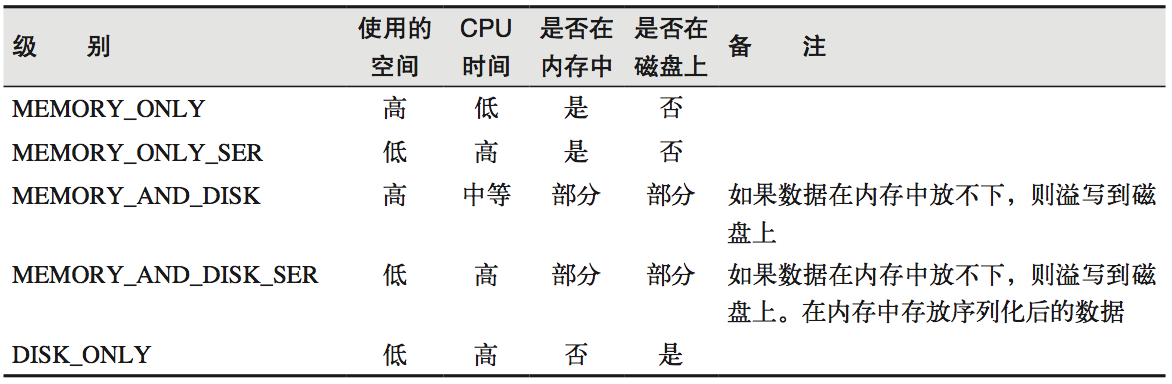
val MEMORY\_AND\_DISK = new StorageLevel(true, true, false, true)

val MEMORY\_AND\_DISK\_2 = new StorageLevel(true, true, false, true, 2)

val MEMORY\_AND\_DISK\_SER = new StorageLevel(true, true, false, false)

val MEMORY\_AND\_DISK\_SER\_2 = new StorageLevel(true, true, false, false, 2)

val OFF\_HEAP = new StorageLevel(true, true, true, false, 1)



缓存有可能丢失，或者存储于内存的数据由于内存不足而被删除，RDD的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于RDD的一系列转换，丢失的数据会被重算，由于RDD的各个Partition是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部Partition。

Spark会自动对一些Shuffle操作的中间数据做持久化操作(比如：reduceByKey)。这样做的目的是为了当一个节点Shuffle失败了避免重新计算整个输入。但是，在实际使用的时候，如果想重用数据，仍然建议调用persist或cache。

1. **RDD CheckPoint检查点**

所谓的检查点其实就是通过将RDD中间结果写入磁盘

由于血缘依赖过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果检查点之后有节点出现问题，可以从检查点开始重做血缘，减少了开销。

对RDD进行checkpoint操作并不会马上被执行，必须执行Action操作才能触发。

// 设置检查点路径

sc.setCheckpointDir("./checkpoint1")

// 创建一个RDD，读取指定位置文件:hello atguigu atguigu

val lineRdd: RDD[String] = sc.textFile("input/1.txt")

// 业务逻辑

val wordRdd: RDD[String] = lineRdd.flatMap(line => line.split(" "))

val wordToOneRdd: RDD[(String, Long)] = wordRdd.map {

word => {

(word, System.currentTimeMillis())

}

}

// 增加缓存,避免再重新跑一个job做checkpoint

wordToOneRdd.cache()

// 数据检查点：针对wordToOneRdd做检查点计算

wordToOneRdd.checkpoint()

// 触发执行逻辑

wordToOneRdd.collect().foreach(println)

1. **缓存和检查点区别**

1）Cache缓存只是将数据保存起来，不切断血缘依赖。Checkpoint检查点切断血缘依赖。

2）Cache缓存的数据通常存储在磁盘、内存等地方，可靠性低。Checkpoint的数据通常存储在HDFS等容错、高可用的文件系统，可靠性高。

3）建议对checkpoint()的RDD使用Cache缓存，这样checkpoint的job只需从Cache缓存中读取数据即可，否则需要再从头计算一次RDD。

#### 5.1.4.9 RDD分区器

Spark目前支持Hash分区和Range分区，和用户自定义分区。Hash分区为当前的默认分区。分区器直接决定了RDD中分区的个数、RDD中每条数据经过Shuffle后进入哪个分区，进而决定了Reduce的个数。

* 只有Key-Value类型的RDD才有分区器，非Key-Value类型的RDD分区的值是None
* 每个RDD的分区ID范围：0 ~ (numPartitions - 1)，决定这个值是属于那个分区的。

1. **Hash分区**：对于给定的key，计算其hashCode,并除以分区个数取余

class HashPartitioner(partitions: Int) extends Partitioner {

require(partitions >= 0, s"Number of partitions ($partitions) cannot be negative.")

def numPartitions: Int = partitions

def getPartition(key: Any): Int = key match {

case null => 0

case \_ => Utils.nonNegativeMod(key.hashCode, numPartitions)

}

override def equals(other: Any): Boolean = other match {

case h: HashPartitioner =>

h.numPartitions == numPartitions

case \_ =>

false

}

override def hashCode: Int = numPartitions

}

1. **Range分区**：将一定范围内的数据映射到一个分区中，尽量保证每个分区数据均匀，而且分区间有序

class RangePartitioner[K : Ordering : ClassTag, V](

partitions: Int,

rdd: RDD[\_ <: Product2[K, V]],

private var ascending: Boolean = true)

extends Partitioner {

// We allow partitions = 0, which happens when sorting an empty RDD under the default settings.

require(partitions >= 0, s"Number of partitions cannot be negative but found $partitions.")

private var ordering = implicitly[Ordering[K]]

// An array of upper bounds for the first (partitions - 1) partitions

private var rangeBounds: Array[K] = {

...

}

def numPartitions: Int = rangeBounds.length + 1

private var binarySearch: ((Array[K], K) => Int) = CollectionsUtils.makeBinarySearch[K]

def getPartition(key: Any): Int = {

val k = key.asInstanceOf[K]

var partition = 0

if (rangeBounds.length <= 128) {

// If we have less than 128 partitions naive search

while (partition < rangeBounds.length && ordering.gt(k, rangeBounds(partition))) {

partition += 1

}

} else {

// Determine which binary search method to use only once.

partition = binarySearch(rangeBounds, k)

// binarySearch either returns the match location or -[insertion point]-1

if (partition < 0) {

partition = -partition-1

}

if (partition > rangeBounds.length) {

partition = rangeBounds.length

}

}

if (ascending) {

partition

} else {

rangeBounds.length - partition

}

}

override def equals(other: Any): Boolean = other match {

...

}

override def hashCode(): Int = {

...

}

@throws(classOf[IOException])

private def writeObject(out: ObjectOutputStream): Unit = Utils.tryOrIOException {

...

}

@throws(classOf[IOException])

private def readObject(in: ObjectInputStream): Unit = Utils.tryOrIOException {

...

}

}

#### 5.1.4.10 RDD文件读取与保存

Spark的数据读取及数据保存可以从两个维度来作区分：文件格式以及文件系统。

文件格式分为：text文件、csv文件、sequence文件以及Object文件；

文件系统分为：本地文件系统、HDFS、HBASE以及数据库。

* **text文件**

// 读取输入文件

val inputRDD: RDD[String] = sc.textFile("input/1.txt")

// 保存数据

inputRDD.saveAsTextFile("output")

* **sequence文件**

SequenceFile文件是[Hadoop](http://lib.csdn.net/base/hadoop" \t "_blank" \o "Hadoop知识库)用来存储二进制形式的key-value对而设计的一种平面文件(Flat File)。在SparkContext中，可以调用sequenceFile[keyClass, valueClass](path)。

// 保存数据为SequenceFile

dataRDD.saveAsSequenceFile("output")

// 读取SequenceFile文件

sc.sequenceFile[Int,Int]("output").collect().foreach(println)

* **object对象文件**

对象文件是将对象序列化后保存的文件，采用Java的序列化机制。可以通过objectFile[T: ClassTag](path)函数接收一个路径，读取对象文件，返回对应的RDD，也可以通过调用saveAsObjectFile()实现对对象文件的输出。因为是序列化所以要指定类型。

// 保存数据

dataRDD.saveAsObjectFile("output")

// 读取数据

sc.objectFile[Int]("output").collect().foreach(println)

## **5.2 累加器**

### **5.2.1 实现原理**

累加器用来把Executor端变量信息聚合到Driver端。在Driver程序中定义的变量，在Executor端的每个Task都会得到这个变量的一份新的副本，每个task更新这些副本的值后，传回Driver端进行merge。

### **5.2.2 基础编程**

#### 5.2.2.1 系统累加器

val rdd = sc.makeRDD(List(1,2,3,4,5))

// 声明累加器

var sum = sc.longAccumulator("sum");

rdd.foreach(

num => {

// 使用累加器

sum.add(num)

}

)

// 获取累加器的值

println("sum = " + sum.value)

#### 5.2.2.2 自定义累加器

// 自定义累加器

// 1. 继承AccumulatorV2，并设定泛型

// 2. 重写累加器的抽象方法

class WordCountAccumulator extends AccumulatorV2[String, mutable.Map[String, Long]]{

var map : mutable.Map[String, Long] = mutable.Map()

// 累加器是否为初始状态

override def isZero: Boolean = {

map.isEmpty

}

// 复制累加器

override def copy(): AccumulatorV2[String, mutable.Map[String, Long]] = {

new WordCountAccumulator

}

// 重置累加器

override def reset(): Unit = {

map.clear()

}

// 向累加器中增加数据 (In)

override def add(word: String): Unit = {

// 查询map中是否存在相同的单词

// 如果有相同的单词，那么单词的数量加1

// 如果没有相同的单词，那么在map中增加这个单词

map(word) = map.getOrElse(word, 0L) + 1L

}

// 合并累加器

override def merge(other: AccumulatorV2[String, mutable.Map[String, Long]]): Unit = {

val map1 = map

val map2 = other.value

// 两个Map的合并

map = map1.foldLeft(map2)(

( innerMap, kv ) => {

innerMap(kv.\_1) = innerMap.getOrElse(kv.\_1, 0L) + kv.\_2

innerMap

}

)

}

// 返回累加器的结果 （Out）

override def value: mutable.Map[String, Long] = map

}

## **5.3 广播变量**

### **5.3.1 实现原理**

广播变量用来高效分发较大的对象。向所有工作节点发送一个较大的只读值，以供一个或多个Spark操作使用。比如，如果你的应用需要向所有节点发送一个较大的只读查询表，广播变量用起来都很顺手。在多个并行操作中使用同一个变量，但是 Spark会为每个任务分别发送。

### **5.3.2 基础编程**

val rdd1 = sc.makeRDD(List( ("a",1), ("b", 2), ("c", 3), ("d", 4) ),4)

val list = List( ("a",4), ("b", 5), ("c", 6), ("d", 7) )

// 声明广播变量

val broadcast: Broadcast[List[(String, Int)]] = sc.broadcast(list)

val resultRDD: RDD[(String, (Int, Int))] = rdd1.map {

case (key, num) => {

var num2 = 0

// 使用广播变量

for ((k, v) <- broadcast.value) {

if (k == key) {

num2 = v

}

}

(key, (num, num2))

}

}

# **Spark案例实操**

在之前的学习中，我们已经学习了Spark的基础编程方式，接下来，我们看看在实际的工作中如何使用这些API实现具体的需求。这些需求是电商网站的真实需求，所以在实现功能前，咱们必须先将数据准备好。



上面的数据图是从数据文件中截取的一部分内容，表示为电商网站的用户行为数据，主要包含用户的4种行为：搜索，点击，下单，支付。数据规则如下：

* 数据文件中每行数据采用下划线分隔数据
* 每一行数据表示用户的一次行为，这个行为只能是4种行为的一种
* 如果搜索关键字为null,表示数据不是搜索数据
* 如果点击的品类ID和产品ID为-1，表示数据不是点击数据
* 针对于下单行为，一次可以下单多个商品，所以品类ID和产品ID可以是多个，id之间采用逗号分隔，如果本次不是下单行为，则数据采用null表示
* 支付行为和下单行为类似

详细字段说明：

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 字段名称 | 字段类型 | 字段含义 |
| 1 | date | String | 用户点击行为的日期 |
| 2 | user\_id | Long | 用户的ID |
| 3 | session\_id | String | Session的ID |
| 4 | page\_id | Long | 某个页面的ID |
| 5 | action\_time | String | 动作的时间点 |
| 6 | search\_keyword | String | 用户搜索的关键词 |
| 7 | click\_category\_id | Long | 某一个商品品类的ID |
| 8 | click\_product\_id | Long | 某一个商品的ID |
| 9 | order\_category\_ids | String | 一次订单中所有品类的ID集合 |
| 10 | order\_product\_ids | String | 一次订单中所有商品的ID集合 |
| 11 | pay\_category\_ids | String | 一次支付中所有品类的ID集合 |
| 12 | pay\_product\_ids | String | 一次支付中所有商品的ID集合 |
| 13 | city\_id | Long | 城市 id |

样例类：

//用户访问动作表

case class UserVisitAction(

date: String,//用户点击行为的日期

user\_id: Long,//用户的ID

session\_id: String,//Session的ID

page\_id: Long,//某个页面的ID

action\_time: String,//动作的时间点

search\_keyword: String,//用户搜索的关键词

click\_category\_id: Long,//某一个商品品类的ID

click\_product\_id: Long,//某一个商品的ID

order\_category\_ids: String,//一次订单中所有品类的ID集合

order\_product\_ids: String,//一次订单中所有商品的ID集合

pay\_category\_ids: String,//一次支付中所有品类的ID集合

pay\_product\_ids: String,//一次支付中所有商品的ID集合

city\_id: Long

)//城市 id

## **6.1 需求1：Top10热门品类**



### **6.1.1 需求说明**

品类是指产品的分类，大型电商网站品类分多级，咱们的项目中品类只有一级，不同的公司可能对热门的定义不一样。我们按照每个品类的点击、下单、支付的量来统计热门品类。

鞋 点击数 下单数 支付数

衣服 点击数 下单数 支付数

电脑 点击数 下单数 支付数

例如，综合排名 = 点击数\*20%+下单数\*30%+支付数\*50%

本项目需求优化为：先按照点击数排名，靠前的就排名高；如果点击数相同，再比较下单数；下单数再相同，就比较支付数。

### **6.1.2 实现方案一**

#### 6.1.2.1 需求分析

分别统计每个品类点击的次数，下单的次数和支付的次数：

（品类，点击总数）（品类，下单总数）（品类，支付总数）

#### 6.1.2.2 需求实现

### **6.1.3 实现方案二**

#### 6.1.3.1 需求分析

一次性统计每个品类点击的次数，下单的次数和支付的次数：

（品类，（点击总数，下单总数，支付总数））

#### 6.1.3.2 需求实现

### **6.1.4 实现方案三**

#### 6.1.4.1 需求分析

使用累加器的方式聚合数据

#### 6.1.4.2 需求实现

## **6.2 需求2：Top10热门品类中每个品类的Top10活跃Session统计**

#### 6.2.1 需求说明

在需求一的基础上，增加每个品类用户session的点击统计

### **6.2.2 需求分析**

### **6.2.3 功能实现**

## **6.3 需求3：页面单跳转换率统计**

### **6.3.1 需求说明**

1）页面单跳转化率

计算页面单跳转化率，什么是页面单跳转换率，比如一个用户在一次 Session 过程中访问的页面路径 3,5,7,9,10,21，那么页面 3 跳到页面 5 叫一次单跳，7-9 也叫一次单跳，那么单跳转化率就是要统计页面点击的概率。

比如：计算 3-5 的单跳转化率，先获取符合条件的 Session 对于页面 3 的访问次数（PV）为 A，然后获取符合条件的 Session 中访问了页面 3 又紧接着访问了页面 5 的次数为 B，那么 B/A 就是 3-5 的页面单跳转化率。



2）统计页面单跳转化率意义

产品经理和运营总监，可以根据这个指标，去尝试分析，整个网站，产品，各个页面的表现怎么样，是不是需要去优化产品的布局；吸引用户最终可以进入最后的支付页面。

数据分析师，可以此数据做更深一步的计算和分析。

企业管理层，可以看到整个公司的网站，各个页面的之间的跳转的表现如何，可以适当调整公司的经营战略或策略。

### **6.3.2 需求分析**

### **6.3.3 功能实现**