# # day01 [Spark概述,集群安装,简单操作,词频统计应用程序开发]

## ## 今日内容

- Spark概述

- Spark集群安装部署

- Spark集群的高可用部署

- Spark集群之命令的简单使用

- 能够使用Scala完成简单的词频统计的Spark应用程序(掌握)

- 能够使用Java完成简单的词频统计的Spark应用程序(掌握)

- 能够使用Java8新特性(兰姆达表达式)完成简单的词频统计的Spark应用程序(扩展)

## ## 教学目标

- 能够了解Spark相关概念和历史

- 掌握Spark集群的安装和配置

- 掌握Spark集群的高可用部署

- 掌握Spark集群常用的操作

- 掌握使用Scala语言完成词频统计Spark应用程序的开发

- 了解使用Java语言完成词频统计Spark应用程序的开发

- 了解使用Java8新特性完成词频统计Spark应用程序的开发

# 前奏

本次课程主要使用的IT技术生态及版本:

| 技术名称 | 版本号 |

| :----------: | :----------------------: |

| 操作系统 | CentOS6.5 |

| 远程连接工具 | SecureCRT |

| 虚拟工作站 | Vmware Workstation 12 |

| Java | JDK1.8.0\_144 |

| Scala | Scala 2.11.8 |

| Hadoop | Hadoop 2.7.4 |

| HIve | Hive 0.13.1(可选) |

| kafka | kafka\_2.11-0.8.2.1(可选) |

| zookeeper | zookeeper 3.4.5 |

| Spark | Spark2.2.0 |

\*\*注:为了课程能够顺利的进行,建议使用老师提供好的环境,在此基础上进行spark的学习\*\*

# # 第一章 Spark概述

## ## 1.1 什么是Spark

官网:http://spark.apache.org/

![](img/Spark官网介绍.png)



官网介绍,Apache Spark是一个用于大规模数据处理的统一分析引擎,是快如闪电的统一分析引擎.

Spark是UC Berkeley AMP lab(加州大学伯克利分校的AMP实验室)所开源的类似于Hadoop MapReduce的通用并行框架.

它有Hadoop MapReduce所具有的优点,不同是,Spark的输出结果可以保存在内存中,从而不再需要读写HDFS.

但是需要注意的是Spark仅仅是Hadoop MapReduce的替代方案,不能全部替代Hadoop生态系统.

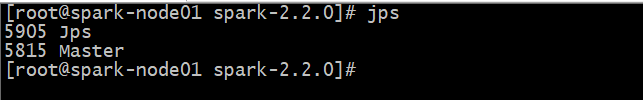
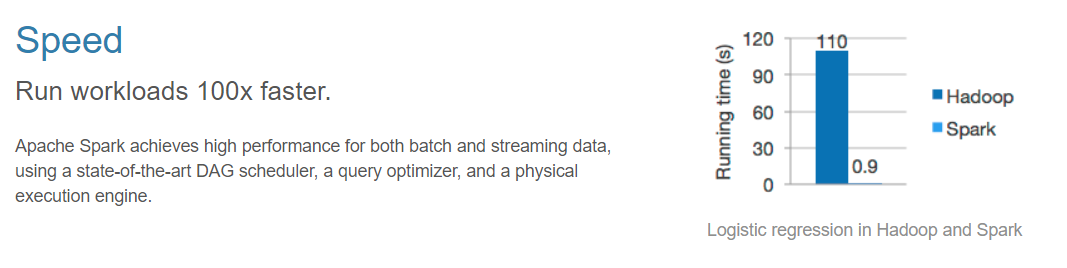
## 1.2 Spark历史

2009 年诞生于加州大学伯克利分校 AMPLab，2010 年开源，2013 年 6 月成为 Apache 孵化项目，2014年2月成为Apache 顶级项目。目前，Spark 生态系统已经发展成为一个包含多个子项目的集合，其中包含 SparkSQL、Spark Streaming、GraphX、MLlib ,Structured Streaming 等子项目，Spark 是基于内存计算的大数据并行计算框架。Spark 基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将Spark 部署在大量廉价硬件之上，形成集群。Spark 得到了众多大数据公司的支持，这些公司包括 Hortonworks、IBM、Intel、Cloudera、MapR、Pivotal、百度、阿里、腾讯、京东、携程、优酷土豆。当前百度的 Spark 已应用于凤巢、大搜索、直达号、百度大数据等业务；阿里利用 GraphX 构建了大规模的图计算和图挖掘系统，实现了很多生产系统的推荐算法；腾讯 Spark 集群达到 8000 台的规模，是当前已知的世界上最大的 Spark 集群。

## 1.3 Spark框架的特点

### ### 1.3.1 速度快

![](img\Spark特点一.png)

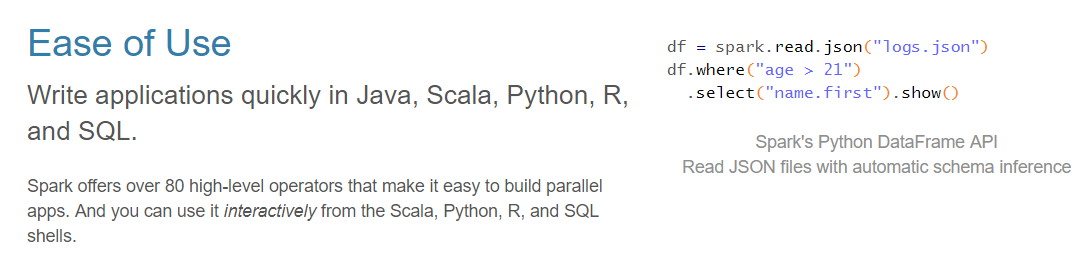


与Hadoop的MapReduce相比,Spark基于内存的运算要快100倍以上,基于硬盘的运算也要快上10倍,

Spark实现了高效的DAG执行引擎,可以通过基于内存来高效处理数据流

### ### 1.3.2 易用性

![](img\Spark特点二\_易用性.png)

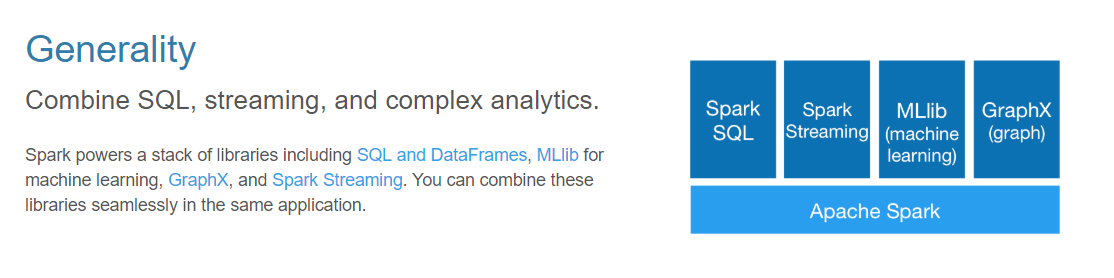


易用性非常好,他可以非常快速的使用Java,Scala,Python,R以及SQL语言编写应用程序.

而且 Spark 支持交互式的Python 和 Scala 的 shell，可以非常方便地在这些 shell 中使用 Spark 集群来验证解决问题的方法。

### ### 1.3.3 通用性

![](img\Spark特点三\_通用性.png)



Spark 提供了统一的解决方案。

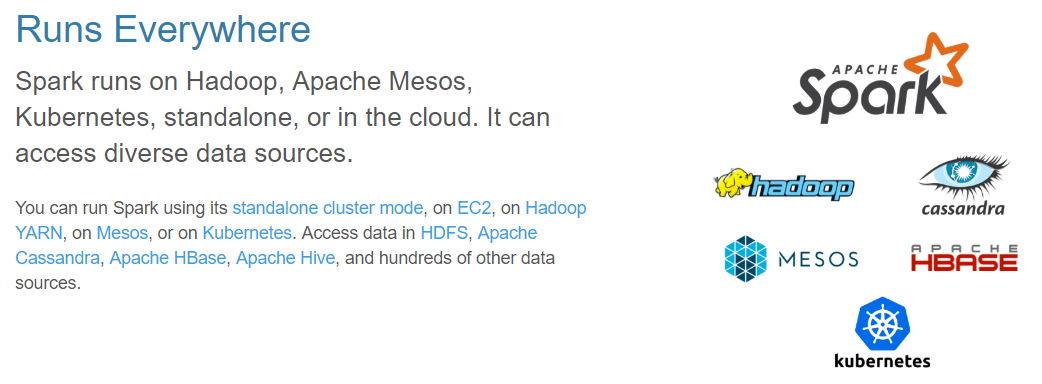
Spark 可以用于批处理、交互式查询（Spark SQL）、实时流处理（Spark Streaming）、机器学习（Spark MLlib）和图计算（GraphX）。

这些不同类型的处理都可以在同一个应用中无缝使用。

Spark统一的解决方案非常具有吸引力，毕竟任何公司都想用统一的平台去处理遇到的问题，减少开发和维护的人力成本和部署平台的物力成本。

### ### 1.3.4 兼容性

![](img\Spark特点四\_兼容性.png)



Spark 可以非常方便地与其他的开源产品进行融合。比如，Spark 可以使用Hadoop 的 YARN 和 Apache Mesos 作为它的资源管理和调度器，

并且可以处理所有 Hadoop 支持的数据，包括 HDFS、HBase 和 Cassandra 等。

这对于已经部署 Hadoop 集群的用户特别重要，因为不需要做任何数据迁移就可以使用 Spark的强大处理能力。

Spark 也可以不依赖于第三方的资源管理和调度器，它实现了Standalone 作为其内置的资源管理和调度框架，这样进一步降低了 Spark 的使用

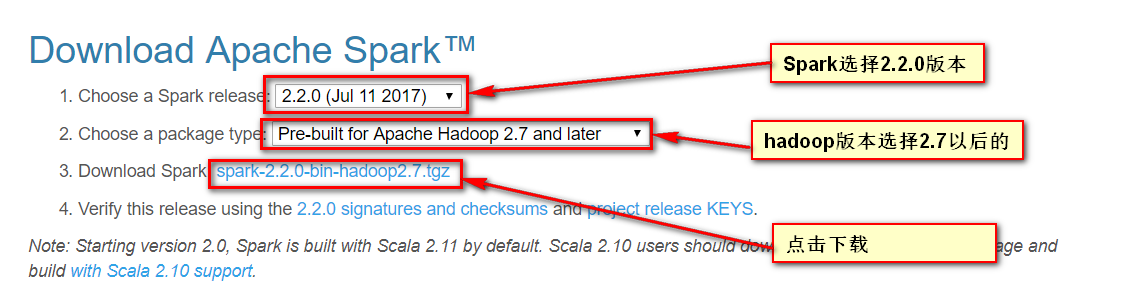
门槛，使得所有人都可以非常容易地部署和使用 Spark。此外，Spark 还提供了在EC2 上部署 Standalone 的 Spark 集群的工具。

# 第二章 Spark集群安装

## ## 2.1 下载Spark安装包

下载Spark安装包地址:http://spark.apache.org/downloads.html

![](img\Spark安装包下载.png)



\*\*注:本次学习过程中我们选用比较新的spark版本spark2.2.0\*\*

上述方式获取Spark安装包的方式是使用官方编译好的安装包来进行安装的.

官方一般只提供有限的几个匹配hadoop版本的编译包.所以,如果将来你需要特意匹配你自己的hadoop版本,这个时候就需要你自己去编译spark源码.

具体的编译步骤如下:构建步骤参照官网:http://spark.apache.org/docs/2.2.0/building-spark.htmlApache

Spark是基于Maven构建的,构建Spark的时候,Maven的版本需要在3.3.9以上,需要的Java版本是Java8以上.

在编译源码之前,需要修改一个配置文件spark2.2.0/dev/make-distribution.sh,需要在配置文件中编辑如下内容:

```shell

VERSION=2.2.0

SCALA\_HOME=your scala version in your machine

SPARK\_HADOOP\_VERSION=your hadoop version in your machine

SPARK\_HIVE=1

```

以上配置,可以不必配置.但是,如果不配置,它会在系统中寻找,非常耗时.指定之后,就可以让编译跑的更加快.

\*\*注:手动编译需要联网,并且在网络情况比较好的情况下,2个小时内可以编译成功.在编译的过程中回出现各种各样的问题,但是,不要怕,有什么问题解决就是了.最后一定可以成功的.\*\*

## ## 2.2 规划安装

### ### 2.2.1 目录规划

| 目录名称 | 说明 |

| :------------: | :---------------------: |

| /opt/modules | 主要是用来安装各种组件 |

| /opt/jars | 放置我们编写的一个jar包 |

| /opt/datas | 放置一些数据文件 |

| /opt/softwares | 放置各种组件的安装包 |

### ### 2.2.2 节点规划

| spark-node01.itheima.com | spark-node02.itheima.com | spark-node03.itheima.com |

| :----------------------: | :----------------------: | :----------------------: |

| Master | Worker | Worler |

## ## 2.3 解压安装包

tar -zxvf spark-2.2.0-bin-hadoop2.7.tgz -C /opt/modules/

执行上述命令将压缩包进行解压,并对解压后的文件夹进行冲命名

mv spark-2.2.0-bin-hadoop2.7/ spark-2.2.0

## 2.4 修改配置文件

Spark的配置文件的位置在$SPARK\_HOME/conf/下

### ### 2.4.1 修改spark-env.sh

- 首先将spark-env.sh.template这个文件重新命名为spark-env.sh

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

- 其次修改spark-env.sh这个配置文件,配置如下内容

- vim spark.env.sh

```shell

#设置JAVA\_HOME目录

JAVA\_HOME=/opt/modules/jdk1.8.0\_144

#设置SCALA\_HOME目录

SCALA\_HOME=/opt/modules/scala-2.11.8

#设置SPARK主机的地址

SPARK\_MASTER\_HOST=spark-node01.itheima.com

#设置SPARK主机的端口地址

SPARK\_MASTER\_PORT=7077

#设置worker节点的内存大小

SPARK\_WORKER\_MEMORY=1g

#设置HDFS文件系统的配置文件的位置

export HADOOP\_CONF\_DIR=/opt/modules/hadoop-2.7.4/etc/hadoop

```

### ### 2.4.2 修改slaves配置文件

- 首先将slaves.template这个文件重新命名为slaves

- mv slaves.template slaves

- 然后修改slaves配置文件中的内容,在里面添加worker节点的地址

- vim slaves

```shell

#配置从节点的地址

spark-node02.itheima.com

spark-node03.itheima.com

```

### ### 2.4.3 配置Spark环境变量

- 打开/etc/profile

- vim /etc/profile

- 在里面编辑如下内容

```shell

#配置Spark环境变量

export SPARK\_HOME=/opt/modules/spark-2.2.0

export PATH=$SPARK\_HOME/bin:$PATH

export SPARK\_HOME PATH

```

## ## 2.5 将spark安装包和配置文件拷贝只其他节点

使用scp命令,将spark的安装目录拷贝只远程其他节点

```powershell

#拷贝spark安装目录道node02机器上

scp -r spark-2.2.0/ spark-node02.itheima.com:$PWD

#拷贝spark安装目录到node03机器上

scp -r spark-2.2.0/ spark-node03.itheima.com:$PWD

#拷贝/etc/profile至node02机器上

scp /etc/profile spark-node02.itheima.com:/etc/

#拷贝/etc/profile至node03机器上

scp /etc/profile spark-node03.itheima.com:/etc/

```

\*\*注:在每一台机器上使用source /etc/profile 让配置生效\*\*

## ## 2.6 启动Spark

在主节点上启动Spark,进入$SPARK\_HOME目录,执行如下命令

```shell

./sbin/start-all.sh

```

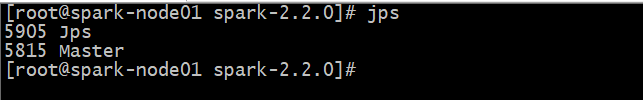
## ## 2.7 验证Spark集群是否启动成功

### ### 2.7.1 使用jps命令查看进程

如果在节点1有master进程,在节点2和节点3上有worker进程,说明Spark集群启动成功

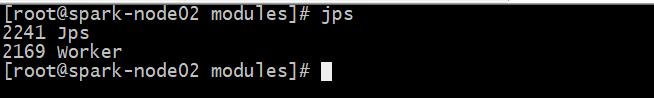
主节点如下:

![](img\spark集群节点1.png)

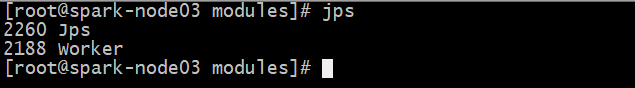


从节点如下:

![](img\spark集群节点2.png)



![](img\spark集群节点3.png)

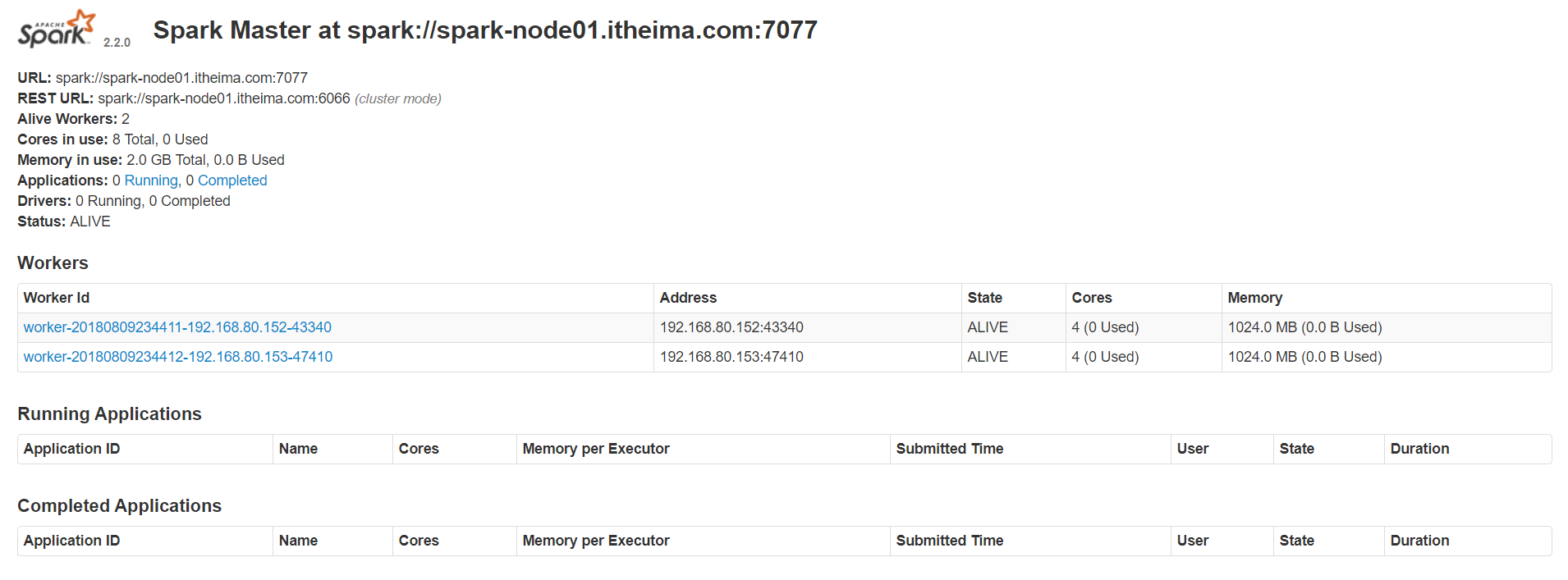


### ### 2.7.2 访问Spark集群的WEB UI界面,查看各个节点状态

访问地址:http://spark-node01.itheima.com:8080/

界面如下:

![](img\Spark集群的WEB UI界面.png)



### ### 2.7.3 使用spark-shell测试spark集群是否启动成功

在任意一个节点上的Spark安装目录执行如下命令

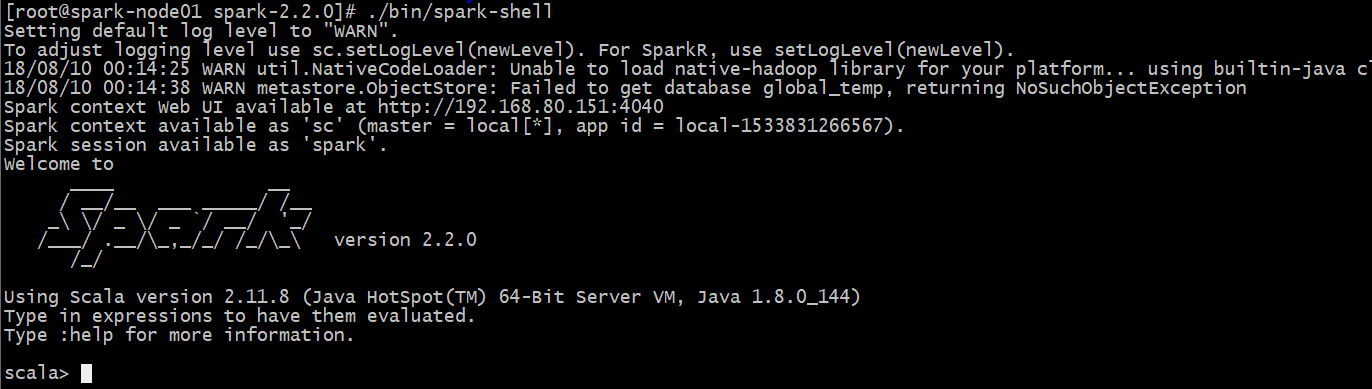
```shell

./bin/spark-shell

```

启动成功之后,界面如下:

![](img/spark-shell界面.png)



至此,我们就验证了整个Spark集群是否成功启动.

\*\*注:如果spark-env.sh配置文件中配置了SPARK\_HADOOP\_CONF这个配置选项,在使用spark-shell进行验证spark集群是否启动成功的时候,需要提前开启HDFS文件系统\*\*

## ## 2.8 停止Spark集群

在Spark的安装目录下使用如下命令

```shell

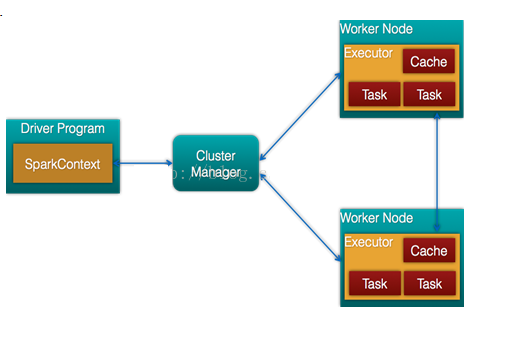
./sbin/stop-all.sh

```

# # 第三章 Spark的高可用部署

## ## 3.1 Spark架构原理图

![](img/04\_Spark架构.png)



## ## 3.2 高可用部署方案说明

Spark Standalone集群是Mater-Slaves架构的集群模式,和大部分的Master-Slaves 结构集群一样，存在着 Master 单点故障的问题。如何解决这个单点故障的问题，Spark 提供了两种方案：

### ### 3.2.1 基于文件系统的单点恢复

此种单点恢复方案主要是用于开发和测试环境,当Spark提供目录保存Spark Application和Worker的注册信息,并将他们的恢复状态写入该目录中,这时,一旦Master发生故障,就可以通过重新启动Master进程(sbin/start-master.sh),恢复已运行的Spark Application和Worker的注册信息.此种方案是手动HA,生产环境中不推荐使用这种方式

### ### 3.2.2 基于zookeeper的Standby Master(Standby Masters with Zookeeper)

此种单点恢复方案主要用于生产模式,基本原理就是通过zookeeper来选举一个Master,其他的Master处于Standby状态.将Spark集群连接到同一个zookeeper实例并启动多个Master,利用zookeeper提供的选举和状态保存功能,可以使一个Master被选举成活着的Master,而其他Master处于Standby状态,如果现任Master死去,另外一个Master会通过选举产生,并恢复到旧的Master状态,然后恢复调度.整个恢复过程可能要1-2分钟.(2n+1) n是zookeeper可以挂掉的节点数目

## ## 3.3 Spark HA高可用集群部署

Spark HA高可用集群部署使用起来很简单,首先需要搭建一个Zookeeper集群,然后启动Zookeeper集群,最后在不同的节点上启动Master

### ### 3.3.1 修改spark-env.sh配置文件

在spark-env.sh配置文件中,注释掉单点主机Master地址,然后添加Spark HA高可用部署的地址

```shell

#设置SPARK主机的地址 注释掉此项

#export SPARK\_MASTER\_HOST=spark-node01.itheima.com

#添加Spark高可用HA部署

export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=spark-node01.itheima.com:2181,spark-node02.itheima.com:2181,spark-node03.itheima.com:2181 -Dspark.deploy.zookeeper.dir=/spark"

```

参数说明:

| 参数名称 | 参数说明 |

| :------------------------: | :----------------------------------------------------------: |

| spark.deploy.recoveryMode | 恢复模式(Master重新启动的模式),主要有三种:1) zookeeper 2) FileSystem 3) NONE |

| spark.deploy.zookeeper.url | zookeeper的Server地址 |

| spark.deploy.zookeeper.dir | 保存Spark集群元数据的文件,目录.包括Worker,Driver和Application |

### 3.3.2 将配置文件发送至远程节点

在节点的spark安装目录下的conf目录执行如下命令

```shell

#将spark-env.sh拷贝至节点02机器上

scp spark-env.sh spark-node02.itheima.com:$PWD

#将spark-env.sh拷贝至节点03机器上

scp spark-env.sh spark-node03.itheima.com:$PWD

```

### ### 3.3.3 验证Spark HA高可用

在普通模式下启动spark集群,只需要在主机上面执行start-all.sh就可以了.在高可用模式下启动Spark集群,首先需要在任意一台节点上启动start-all.sh命令.然后在另外一台节点上单独启动master.命令start-master.sh在节点一执行命令如下

```shell

./sbin/start-all.sh

```

在节点二执行命令如下

```shell

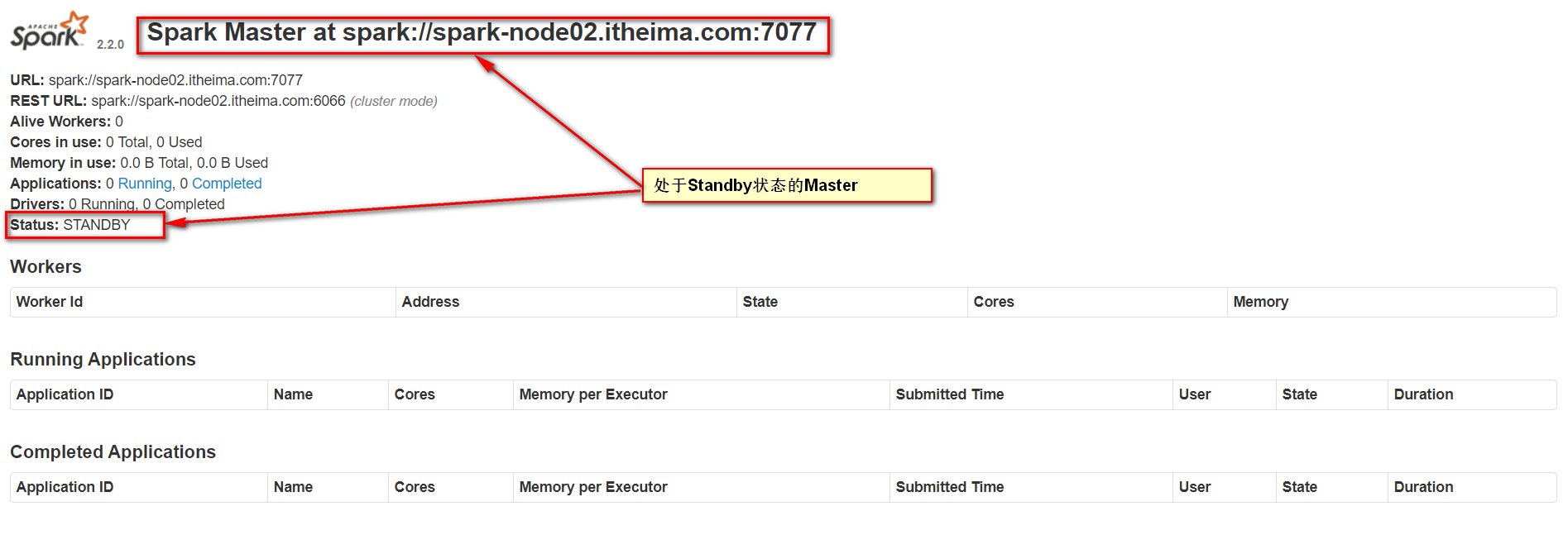
./sbin/start-master.sh

```

访问节点一和节点二的WEB UI界面

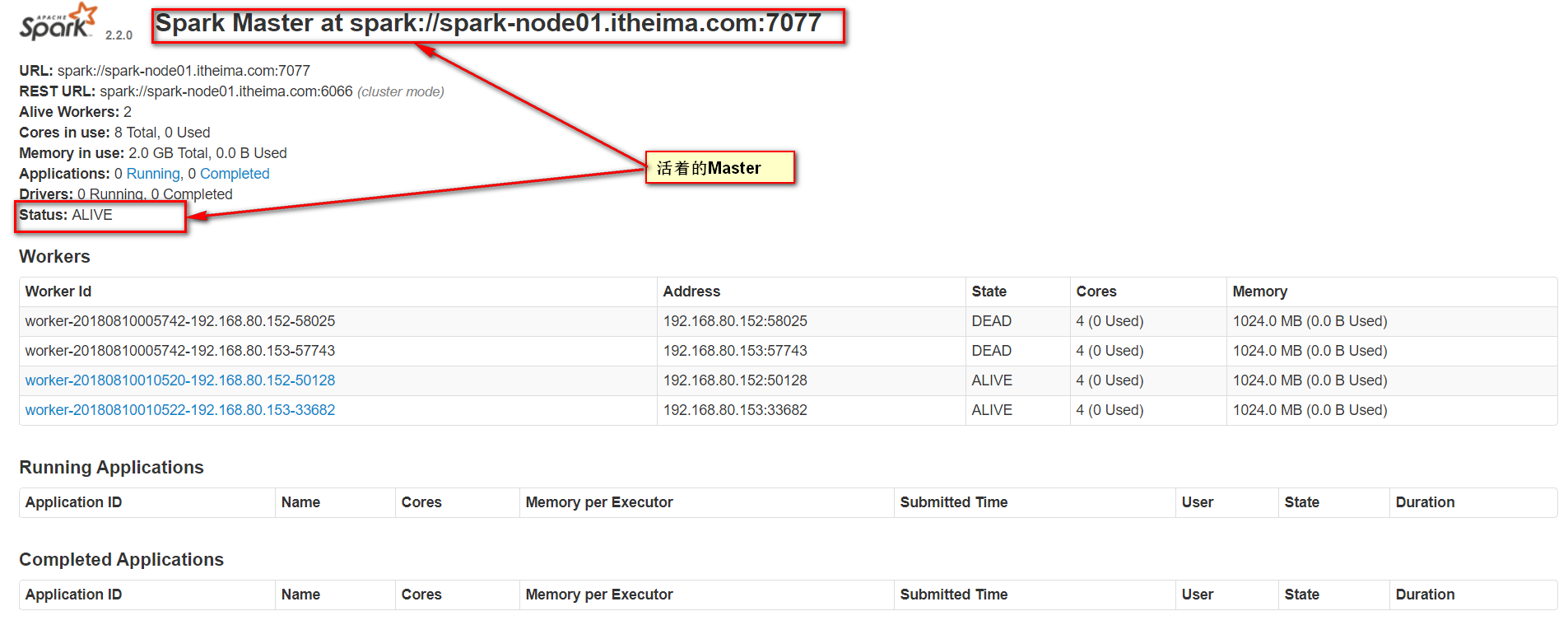
访问地址1:http://spark-node01.itheima.com:8080/

![](img/Spark HA\_1.png)



访问地址2:http://spark-node02.itheima.com:8080/

![](img\Spark HA\_2.png)



接下来,我们将节点1上的Master进程手动杀死它,执行命令

```shell

kill -9 master进程号

```

然后观察节点二的WEB UI界面,这个恢复过程可能需要1到2分钟,需要稍微等待一下

刷新节点1的WEB UI界面,如下所示

![](img\HA\_3.png)



刷新节点2的WEB UI界面,恢复的过程需要1到2分钟

![](img\HA\_4.png)



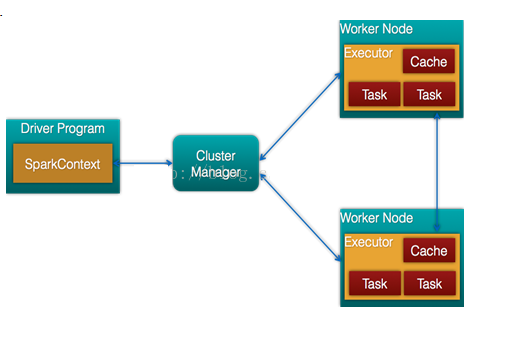
# 第四章 Spark角色介绍

## ## 4.1 Spark角色概述

Spark架构采用了分布式计算中的Master-Slave模型。Master是对应集群中的含有Master进程的节点，Slave是集群中含有Worker进程的节点。Master作为整个集群的控制器，负责整个集群的正常运行；Worker相当于是计算节点，接收主节点命令与进行状态汇报；Executor负责任务的执行；Client作为用户的客户端负责提交应用，Driver负责控制一个应用的执行

Spark集群部署后，需要在主节点和从节点分别启动Master进程和Worker进程，对整个集群进行控制。在一个Spark应用的执行过程中，Driver和Worker是两个重要角色。Driver程序是应用逻辑执行的起点，负责作业的调度，即Task任务的分发，而多个Worker用来管理计算节点和创建Executor并行处理任务。在执行阶段，Driver会将Task和Task所依赖的file和jar序列化后传递给对应的Worker机器，同时Executor对相应数据分区的任务进行处理。

![](img\04\_Spark架构.png)



# 第五章 认识Spark应用程序

## ## 5.1 执行第一个Spark应用程序

### ### 5.1.1 命令说明

第一个Spark应用程序,我们采用官方提供的计算圆周率的一个example来进行实验,从官网上来看,我们在SPARK\_HOME目录下输入如下命令:

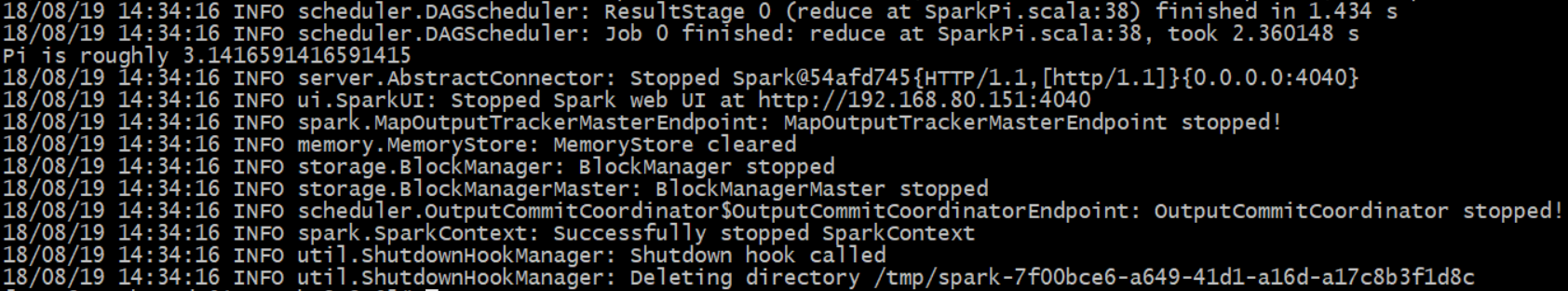
```shell

./bin/run-example SparkPi 10

```

执行结果如下:

![](img\求圆周率.png)



上述的命令,在底层其实调用的是一个spark-submit命令,这个命令会将Spark应用程序提交到Spark集群上去运行

```shell

./bin/spark-submit --class org.apache.spark.examples.SparkPi --executor-memory 1G --total-executor-cores 2 examples/jars/spark-examples\_2.11-2.2.0.jar 10

```

### 5.1.2 参数说明

spark-submit提交应用的命令模板:

```shell

./bin/spark-submit \

--class <main-class> \

--master <master-url> \

--deploy-mode <deploy-mode> \

--conf <key>=<value> \

... # other options

<application-jar> \

[application-arguments]

```

参数说明

| 参数名称 | 参数描述 |

| :---------------------: | :-------------------------------------------: |

| --class | spark应用程序的入口 |

| --master | 集群的主机url |

| --deploy-mode | 部署模式,可以部署在cluster,本地(默认的是本地) |

| --conf | key-value格式的spark配置信息 |

| application-jar | spark应用程序的jar包的位置 |

| [application-arguments] | 如果需要的话,可以传入参数到main方法中 |

## ## 5.2 spark-submit使用

### ### 5.2.1 在本地运行spark application

```shell

./bin/spark-submit --class org.apache.spark.examples.SparkPi --master local[2] --executor-memory 1G --total-executor-cores 2 examples/jars/spark-examples\_2.11-2.2.0.jar 10

```

### ### 5.2.2 在Spark集群上运行spark application

```shell

#本地

./bin/spark-submit --class org.apache.spark.examples.SparkPi --deploy-mode client --master spark://spark-node01.itheima.com:7077 --executor-memory 1G --total-executor-cores 2 examples/jars/spark-examples\_2.11-2.2.0.jar 10

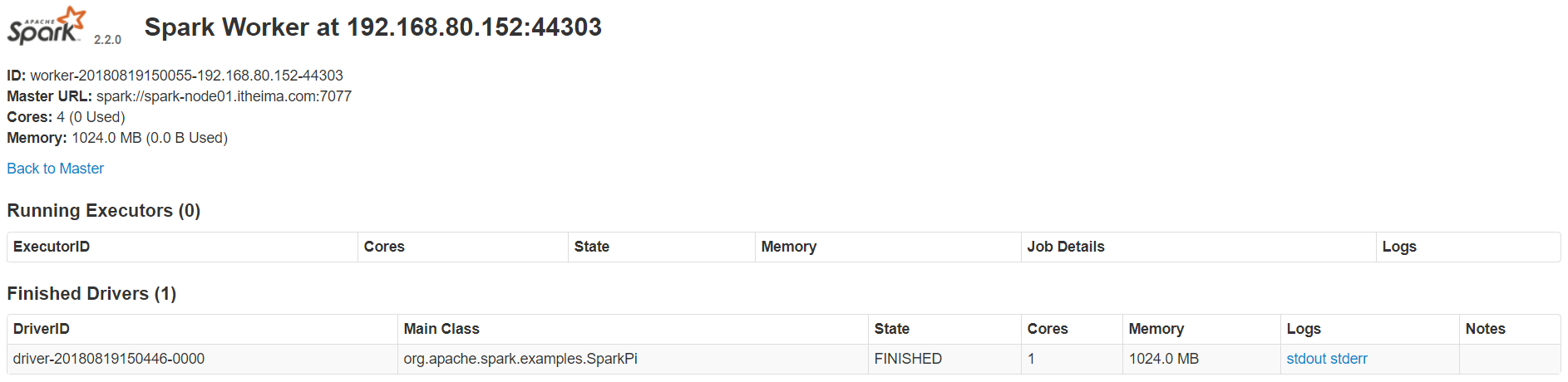
#集群

./bin/spark-submit --class org.apache.spark.examples.SparkPi --deploy-mode cluster --master spark://spark-node01.itheima.com:7077 --executor-memory 1G --total-executor-cores 2 examples/jars/spark-examples\_2.11-2.2.0.jar 10

```

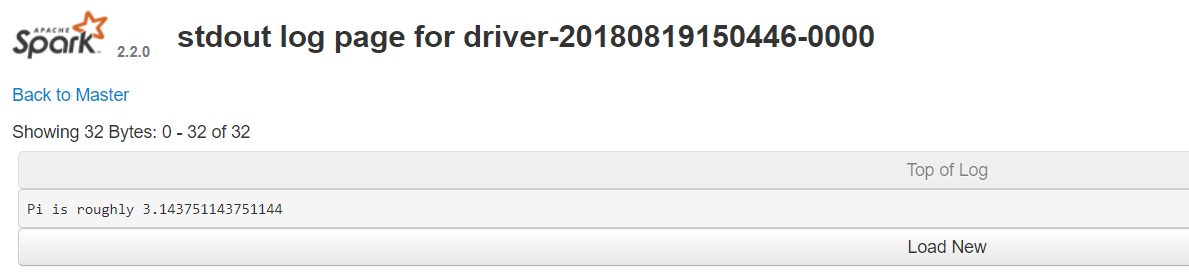
\*\*注:需要提前启动好spark集群,需要我们连接一个指定的集群.让我们能够将Spark应用程序提交到集群上去运行.\*\*

![](img\Spark的worker节点.png)



点击stdouot可以看的到我们的计算结果

![](img\stdout.png)



### ### 5.2.3 在Yarn集群上运行spark application

```shell

#首先导出Hadoop的配置环境

export HADOOP\_CONF\_DIR=/usr/local/tools/hadoop-2.7.4/etc/hadoop

#提交任务

./bin/spark-submit --class org.apache.spark.examples.SparkPi --master yarn --deploy-mode cluster --executor-memory 6G --num-executors 4 examples/jars/spark-examples\_2.11-2.2.0.jar 10

```

\*\*注:需要首先启动yarn集群\*\*

### ### 5.2.4 高可用模式提交

```shell

./bin/spark-submit --class org.apache.spark.examples.SparkPi --master spark://spark-node01.itheima.com:7077 --executor-memory 1G --total-executor-cores 2 examples/jars/spark-examples\_2.11-2.2.0.jar 10

```

## ## 5.3 使用

spark-shell是scala sehll的升级版本,比scala shell的功能强大.通过spark shell你可以运行spark程序,是一种比较好的学习Spark框架的方式.启动命令如下:

```shell

./bin/spark-shell --master local[2]

```

--master:用来指定分布式集群的主节点的地址,或者local是运行在本地的一个线程,或者local[n]是运行在本地的n个线程.我们测试的时候应该多多的使用local.local参数说明如下所示

| 参数 | 说明 |

| :---------------: | :----------------------------------------------------------: |

| local | 使用本地的一个线程 |

| local[k] | 使用本地的k个线程(在理想情况下,将其设置为计算及上的核心数) |

| local[\*] | 使用与计算机逻辑核心数(CPU的个数\*每个CPU的核心数)相同的线程 |

| spark://HOST:PORT | 连接到给定的Spark集群上运行 |

| yarn | 以客户端或者集群的形式连接到Yarn集群,具体使用client还是cluster模式,取决于--deploy-mode这个值.集群的位置将会通过HADOOP\_CONF\_DIR 或者 YARN\_CONF\_DIR 这两个变量的值获得. |

### ### 5.3.1 读取本地文件进行词频统计

```scala

sc.textFile("file:///opt/datas/words.txt").flatMap(line=>line.split(" ")).map(word=>(word,1)).reduceByKey((a,b)=>a+b).collect

//例如 (hello,1) (hello,1),(hello,1) 1+1=2 2+1=3

```

\*\*注:读取本地文件,需要在最前面加上file:本地文件使用的是file协议\*\*

### ### 5.3.2 读取HDFS系统上面的文件进行词频统计

```scala

sc.textFile("hdfs://spark-node01.itheima.com:9000/words.txt").flatMap(line=>line.split(" ")).map(word=>(word,1)).reduceByKey((a,b)=>a+b).collect

```

\*\*注意:需要在spark-env.sh配置文件中添加HADOOP\_CONF\_DIR这个配置,这样saprk集群才可以读取HDFS文件上的内容\*\*

### ### 5.3.3 spark-shell连接指定spark集群

```shell

./bin/spark-shell --master spark://spark-node01.itheima.com:7077 --executor-memory 1g --total-executor-cores 2

```

指定具体的master地址,也就是要将提交的任务交给worker节点中的Executor去执行,主节点master不负责计算任务.只负责调度.

```scala

//词频统计,并将结果存入HDFS文件系统中

sc.textFile("hdfs://spark-node01.itheima.com:9000/words.txt").flatMap(line=>line.split(" ")).map(word=>(word,1)).reduceByKey((a,b)=>a+b).saveAsTextFile("/wc/output")

```

# # 第六章 完成词频统计的Spark应用程序的开发

## ## 6.1 使用Java语言完成词频统计应用程序开发

### ### 6.1.1 环境准备

| 环境名称 | 环境版本 |

| :------: | :--------------------: |

| maven | apache-maven-3.3.9 |

| JDK | jdk1.8.0\_141 |

| Scala | Scala 2.11.8 |

| IDE | IntelliJ IDEA Ultimate |

### ### 6.1.2 新建maven工程

略

### ### 6.1.3 配置pom.xml

```xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.itheima</groupId>

<artifactId>SaprkCloud4</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<!--定义版本号-->

<properties>

<hadoop.version>2.7.4</hadoop.version>

<scala.binary.version>2.11</scala.binary.version>

<spark.version>2.2.0</spark.version>

<scala.version>2.11.8</scala.version>

</properties>

<dependencies>

<!--Scala依赖库-->

<dependency>

<groupId>org.scala-lang</groupId>

<artifactId>scala-library</artifactId>

<version>${scala.version}</version>

</dependency>

<!--spark核心依赖库-->

<dependency>

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

<artifactId>spark-core\_${scala.binary.version}</artifactId>

<version>${spark.version}</version>

</dependency>

<!--mysql数据库依赖-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.18</version>

</dependency>

<!--hadoop版本库-->

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>${hadoop.version}</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<sourceDirectory>src/main/scala</sourceDirectory>

<testSourceDirectory>src/test/scala</testSourceDirectory>

<plugins>

<plugin>

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

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

<version>3.2.2</version>

<executions>

<execution>

<goals>

<goal>compile</goal>

<goal>testCompile</goal>

</goals>

<configuration>

<args>

<arg>-dependencyfile</arg>

<arg>${project.build.directory}/.scala\_dependencies</arg>

</args>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

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

<artifactId>maven-shade-plugin</artifactId>

<version>2.3</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<filters>

<filter>

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*.RSA</exclude>

</excludes>

</filter>

</filters>

<transformers>

<transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

<mainClass></mainClass>

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

<!--配置maven编译插件-->

<plugin>

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

<artifactId>maven-compiler-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

```

### ### 6.1.4 编写Java程序

```java

public class WordCountJavaLocal {

public static void main(String[] args) {

//1:创建SparkConf对象,配置Spark应用程序得基本信息

SparkConf sparkConf = new SparkConf().setAppName("WordCountJavaLocal").setMaster("local[2]");

//2: 创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(sparkConf);

//3:要针对输入源(hdfs文件,本地文件),创建一个初始得RDD

JavaRDD<String> lines = sc.textFile("F:\\data\\words.txt");

//4: 对初始RDD进行transformation操作,也就是一些计算操作

JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {

@Override

public Iterator<String> call(String line) throws Exception {

String[] split = line.split(" ");

return Arrays.asList(split).iterator();

}

});

//5:接着,需要将每一个单词,映射为(单词,1)得这种格式

JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String, Integer>() {

@Override

public Tuple2<String, Integer> call(String word) throws Exception {

return new Tuple2<String,Integer>(word,1);

}

});

//6: 接着,需要以单词作为key,统计每个单词出现得次数

JavaPairRDD<String, Integer> wordCounts = pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {

@Override

public Integer call(Integer v1, Integer v2) throws Exception {

return v1 + v2;

}

});

//7:到这里为止，我们通过几个Spark算子操作，已经统计出了单词的次数,将结果打印到控制台

wordCounts.foreach(new VoidFunction<Tuple2<String, Integer>>() {

@Override

public void call(Tuple2<String, Integer> wordCount) throws Exception {

System.out.println(wordCount.\_1+" 出现了 "+wordCount.\_2+" 次");

}

});

sc.close();

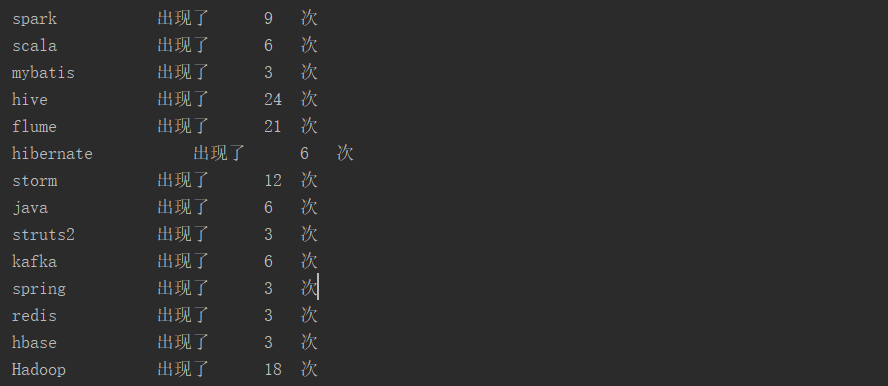
}

}

```

### ### 6.1.5 运行结果

![](img\WordCountLocalJava运行结果.png)



### ### 6.1.6 WordCount程序说明

```java

SparkConf sparkConf = new SparkConf().setAppName("WordCountJavaLocal").setMaster("local[2]");

```

编写Spark程序首先需要创建一个SparkConf对象,这个对象包含了spark应用程序的一些信息.从上述代码可以看出,setMaster可以设置Spark应用程序需要连接的Spark集群的master节点的url,但是,如果设置为local,则代表的是在本地进行测试实验.setAppName()其实是用来设置集群UI页面的显示名称的

```java

JavaSparkContext sc = new JavaSparkContext(sparkConf);

```

第二步就是需要创建一个SparkContext对象,SparkContext对象会告知Spark应用程序如何访问一个集群.也就是说 SparkContext对象是所有Spark应用程序的入口.无论我们是使用java,scala,还是python编写,都必须要有一个SparkContext对象.它的主要作用,包括初始化Spark应用程序所需的一些核心组件,包括调度器(DAGScheduler,TaskScheduler),还会去到Spark Master节点上进行注册,等等.总而言之,SparkContext是Spark应用程序中最最重要的一个对象.

在Spark中，编写不同类型的Spark应用程序，使用的SparkContext是不同的.如下所示:

| 程序开发 | SparkContext对象类型 |

| :-------------: | :--------------------: |

| scala | SparkContext |

| Java | JavaSparkContext |

| Spark SQL | SQLContext/HiveContext |

| Spark Streaming | StreamingContext |

\*\*注:在Spark2.0之后,Spark框架将所有的入口全部整合到了一个新的对象,这个新的对象是所有一切Spark应用程序的入口,这个对象是SparkSession,后期我们会对其进行讲解\*\*

```java

JavaRDD<String> lines = sc.textFile("F:\\data\\words.txt");

```

针对输入源(hdfs文件,本地文件 等等),创建一个初始的RDD,输入源中的数据会被打散,分配到RDD的每个partition中,从而形成一个初始的分布式的数据集.在SparkContext中,用于根据文件类型的输入源创建RDD的方法,叫做textFile()方法,在Java中,创建的普通的RDD,都叫做JavaRDD.我们这里的lines这个RDD,其实就是文件里的每一行数据.

```java

JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {

@Override

public Iterator<String> call(String line) throws Exception {

String[] split = line.split(" ");

return Arrays.asList(split).iterator();

}

});

```

接下来就是对RDD进行transformation操作,也就是一些计算操作,通常操作会通过创建Function,并配合RDD的map,flatMap等算子来执行.Function是一个接口,如果实现比较简单,则创建指定Function的匿名内部类,但是,如果Function实现比较复杂,则会单独创建这个接口的实现类.而且Function接口是一个函数式接口,可以使用Java的新特性(lamdba表达式)来实现业务.

这里是将原始的RDD中的每一行数据拆分成单个的单词,调用flatMap方法,这个方法中需要一个FlatMapFunction的对象,这个FlatMapFunction,有两个泛型参数,分别代表了输入和输出类型.在我们编写的程序中,输入的数据类型肯定是String,因为是一行一行的文本,输出也是String,因为是每一行的文本.

flatMap算子的作用,其实就是,将RDD的一个元素,给拆分成一个或多个元素.

```java

JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String, Integer>() {

@Override

public Tuple2<String, Integer> call(String word) throws Exception {

return new Tuple2<String,Integer>(word,1);

}

});

```

我们需要将每一个单词,映射为(单词,1)的这种格式,因为只有这样,后面才能根据单词作为key,来进行每个单词的出现次数的累加.mapToPair,其实就是将每个元素,映射为一个(v1,v2)这样的Tuple2类型的元素.这里的Tuple2是scala类型,包含了两个值,mapToPair这个算子,要求的是与PairFunction配合使用,第一个泛型参数代表了输入类型,第二个和第三个泛型参数,代表的输出的Tuple2的第一个值和第二个值得类型.JavaPairRDD得两个泛型参数,分别代表了tuple元素得第一个值和第二个值得类型.

```java

JavaPairRDD<String, Integer> wordCounts = pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {

@Override

public Integer call(Integer v1, Integer v2) throws Exception {

return v1 + v2;

}

});

```

这一步需要我们以单词作为key,统计每个单词出现得次数,这个地方需要用到一个算子操作reduceByKey,对每个key对应得value,都进行reduce操作.

例如JavaPairRDD中得几个元素,分别为(spark,1) (spark,1) (spark,1) (hadoop,1).

reduce操作,相当于是把第一个值和第二个值进行计算,然后在将结果与第三个值进行计算.例如上述例子中得spark,就相当于1 + 1 = 2,然后在世 2 + 1 = 3;最后返回得JavaPairRDD中得元素,也是tuple,但是第一个值就是每个key,第二个值就是key得value,reduce之后得结果就相当于每个单词出现得次数.

```java

wordCounts.foreach(new VoidFunction<Tuple2<String, Integer>>() {

@Override

public void call(Tuple2<String, Integer> wordCount) throws Exception {

System.out.println(wordCount.\_1+"\t\t\t出现了\t\t"+wordCount.\_2+"\t次");

}

});

```

到这里,我们已经完成了词频统计得所有算子操作,也就是说,我们已经完成了词频统计,但是,之前我们得所有算子操作所使用得flatMap,mapToPair,ReduceByKey这种操作,都叫做transformation操作,这种操作都是懒加载得,不会主动执行得,必须等我调用action操作得时候,才可以触发这些算子操作.

## ## 6.2 将Java开发得WordCount程序部署到Spark集群上去运行

### ### 6.2.1 修改源代码

将Java开发得Spark应用程序部署到Spark集群上去运行,我们需要修改两个地方,第一个地方就是将SparkConf得setMaster()方法给删除掉,我们可以自己来指定需要连接得Spark集群.

```java

SparkConf sparkConf = new SparkConf().setAppName("WordCountJavaCluster");

```

​

第二个就是我们这次针对得不是本地文件,我们可以通过指定参数得形式我告诉程序读取哪个地方得文件,这个文件位置我们可以通过在程序中获取得到.

```java

//我们不再是针对得是本地文件,修改为动态读取

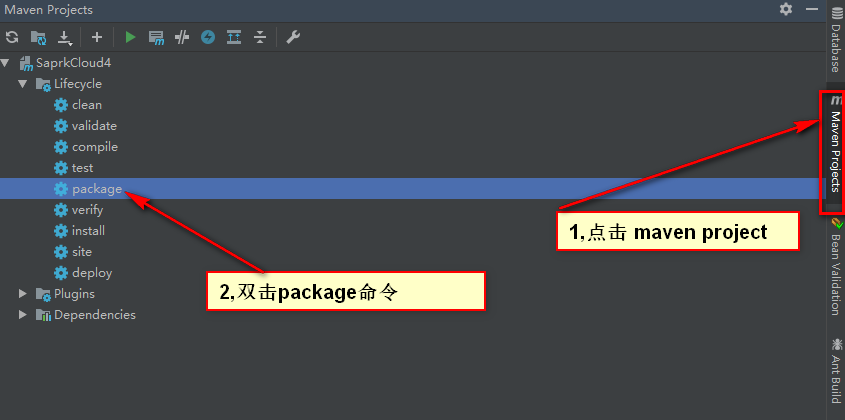
JavaRDD<String> lines = sc.textFile(args[0]);

```

### ### 6.2.2 将WordCountJavaCluster程序打成jar包

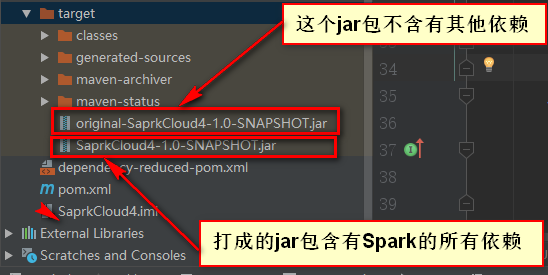
执行package命令:

![](img/package打包.png)



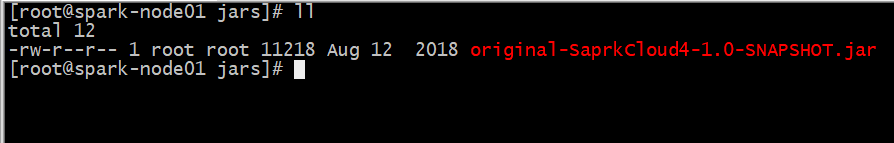
执行上述命令,会将编写的Spark应用程序打成jar包,会生成两个不同的jar包

![](img\jar包说明.png)



我们在将jar包提交到Spark集群去运行的时候,只需要将原始的Jar包提交上去即可,即original-SaprkCloud4-1.0-SNAPSHOT.因为在Spark集群中,Spark的运行环境已经有了.我们将这个jar包上传至Spark集群所在的机器上.

![](img\上传jar包.png)



### ### 6.2.3 编写spark-submit脚本

接下来我们需要编写spark-submit脚本

```shell

/opt/modules/spark-2.2.0/bin/spark-submit \

--class com.itheima.spark.core.WordCountJavaCluster \

--driver-memory 1G \

--executor-memory 1G \

--executor-cores 3 \

/opt/jars/original-SaprkCloud4-1.0-SNAPSHOT.jar \

hdfs://spark-node01.itheima.com:9000/words.txt \

```

### ### 6.2.4 spark-submit执行脚本参数

| 参数名称 | 参数描述 |

| :-------------------------------------------------: | :------------------------: |

| /opt/modules/spark-2.2.0/bin/spark-submit | spark脚本执行的命令 |

| --class com.itheima.spark.core.WordCountJavaCluster | 指定spark-submit执行的主类 |

| --driver-memory 1G | 指定驱动程序运行的内存大小 |

| --executor-memory 1G | 指定执行器内存的大小 |

| --executor-cores 3 | 执行器的核心数 |

| /opt/jars/original-SaprkCloud4-1.0-SNAPSHOT.jar | jar包的位置 |

| hdfs://spark-node01.itheima.com:9000/words.txt | 文件的位置 |

### ### 6.2.5 执行spark-submit脚本

```shell

#首先将wordcount\_java.sh赋予可执行权限

chmod 777 wordcount\_java.sh

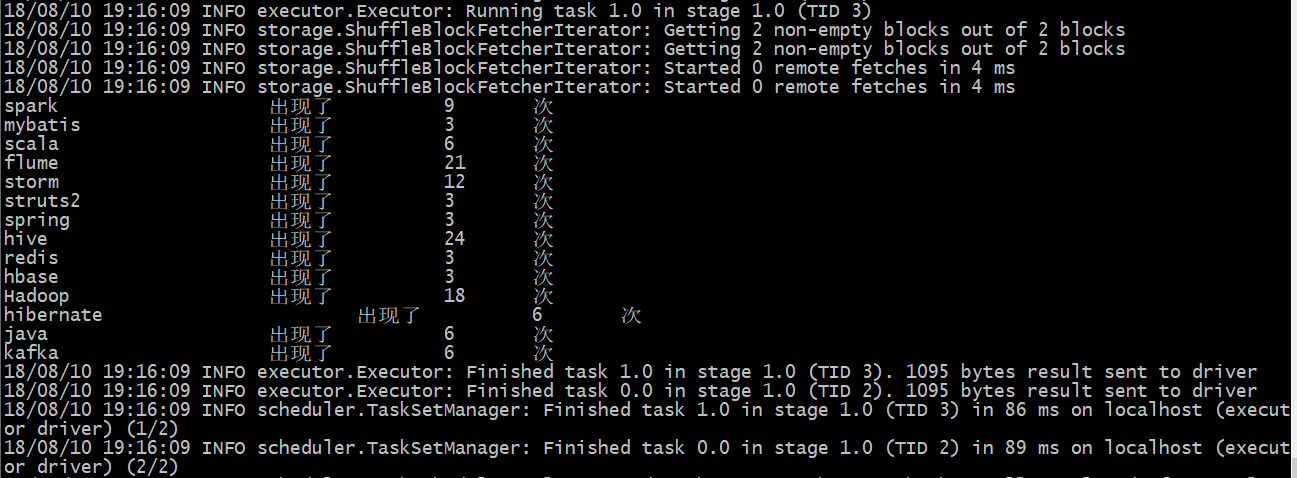
#执行wordcount\_java.sh脚本

./wordcount\_java.sh

```

### ### 6.2.6 运行结果

![](img\spark-submit执行结果.png)



## ## 6.3 使用Scala语言完成词频统计应用程序开发

### ### 6.3.1 编写scala程序

```scala

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

object WordCountScalaLocal {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf().setAppName("WordCountScalaLocal")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:读取文件

val lines: RDD[String] = sc.textFile(args(0))

//4:将每一行数据切分为单词

val words: RDD[String] = lines.flatMap(line=>line.split(" "))

//5:将出现的每一个单词记为1

val pairs: RDD[(String, Int)] = words.map(word=>(word,1))

//6:统计相同的单词出现的次数

val result: RDD[(String, Int)] = pairs.reduceByKey((a, b)=>a+b)

//7:遍历输出所有的结果

result.foreach(wordCount=>{

println(wordCount.\_1+"\t\t\t出现了\t\t\t"+wordCount.\_2+"\t\t次")

})

//8:释放资源

sc.stop()

}

}

```

### ### 6.3.2 将程序进行打包

略

### ### 6.3.3 编写spark-submit脚本

```shell

/opt/modules/spark-2.2.0/bin/spark-submit \

--class com.itheima.spark.core.WordCountScalaLocal \

--driver-memory 1G \

--executor-memory 1G \

--executor-cores 3 \

/opt/jars/scala/original-SaprkCloud4-1.0-SNAPSHOT.jar \

hdfs://spark-node01.itheima.com:9000/words.txt \

```

### ### 6.3.4 执行结果

略

## ## 6.4 使用spark-shell开发词频统计程序

### ### 6.4.1 启动spark-shell

spark-shell主要用来进行测试,在SPARK\_HOME目录下执行命令:

```shell

./bin/spark-shell

```

### ### 6.4.2 编写spark程序

在spark-shell界面输入:paste命令,进入多行输入模式

```scala

import org.apache.spark.rdd.RDD

val lines: RDD[String] = sc.textFile("hdfs://spark-node01.itheima.com:9000/words.txt")

val words: RDD[String] = lines.flatMap(line=>line.split(" "))

val pairs: RDD[(String, Int)] = words.map(word=>(word,1))

val result: RDD[(String, Int)] = pairs.reduceByKey((a, b)=>a+b)

result.saveAsTextFile("/1")

```

将上述代码复制粘贴到spark-shell命令行中去执行