1. 认识Spark

1.1. Spark简介

看官网http://spark.apache.org/了解，主要是两点spark定义，spark特性

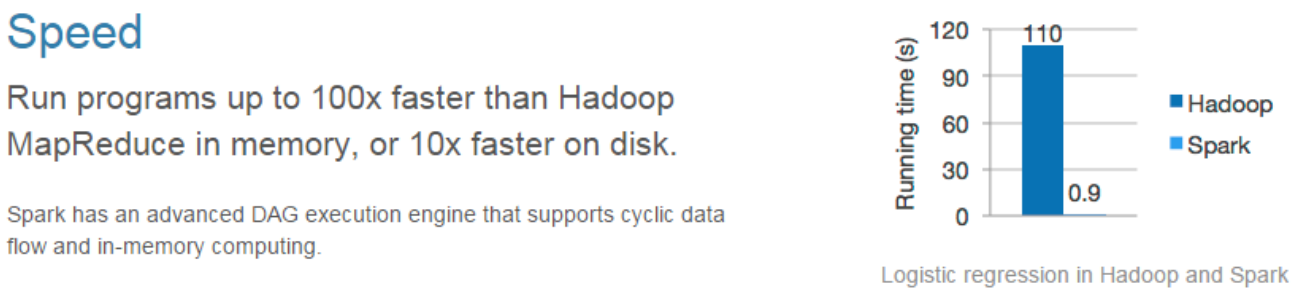
Apache Spark是专为大规模数据处理而设计的快速通用的计算引擎

Apache Spark 是一个快速的, 多用途的集群计算系统。 它提供了 Java, Scala, Python 和 R 的高级 API，以及一个支持通用的执行图计算的优化过的引擎. 它还支持一组丰富的高级工具, 包括使用 SQL 处理结构化数据处理的 Spark SQL, 用于机器学习的 MLlib, 用于图计算的 GraphX, 以及 Spark Streaming。

1.2. Spark的特点

1.2.1 速度快

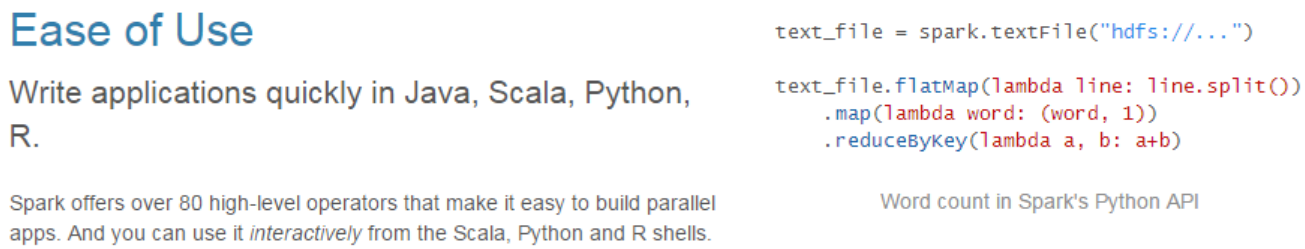
与Hadoop的MapReduce相比，Spark基于内存的运算要快100倍以上，基于硬盘的运算也要快10倍以上。Spark实现了高效的DAG执行引擎，可以通过基于内存来高效处理数据流。



Logistic regression in Hadoop and Spark

1.2.2. 易用

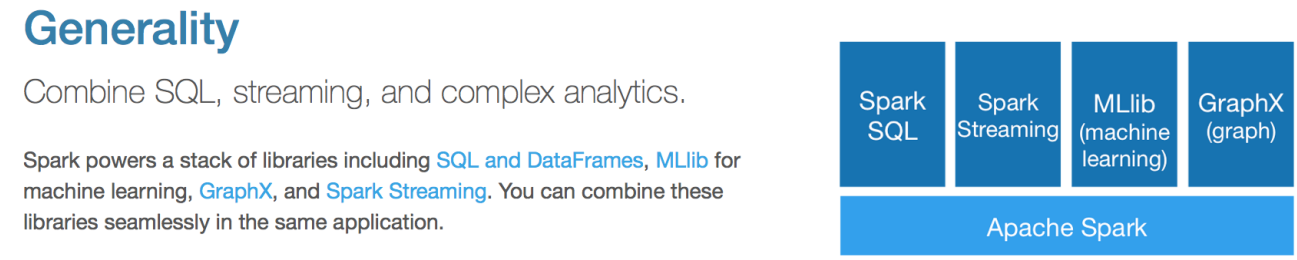
Spark支持Java、Python和Scala的API，还支持超过80种高级算法，使用户可以快速构建不同的应用。而且Spark支持交互式的Python和Scala的shell，可以非常方便地在这些shell中使用Spark集群来验证解决问题的方法。



Word count in Spark's Python API

1.2.3. 通用

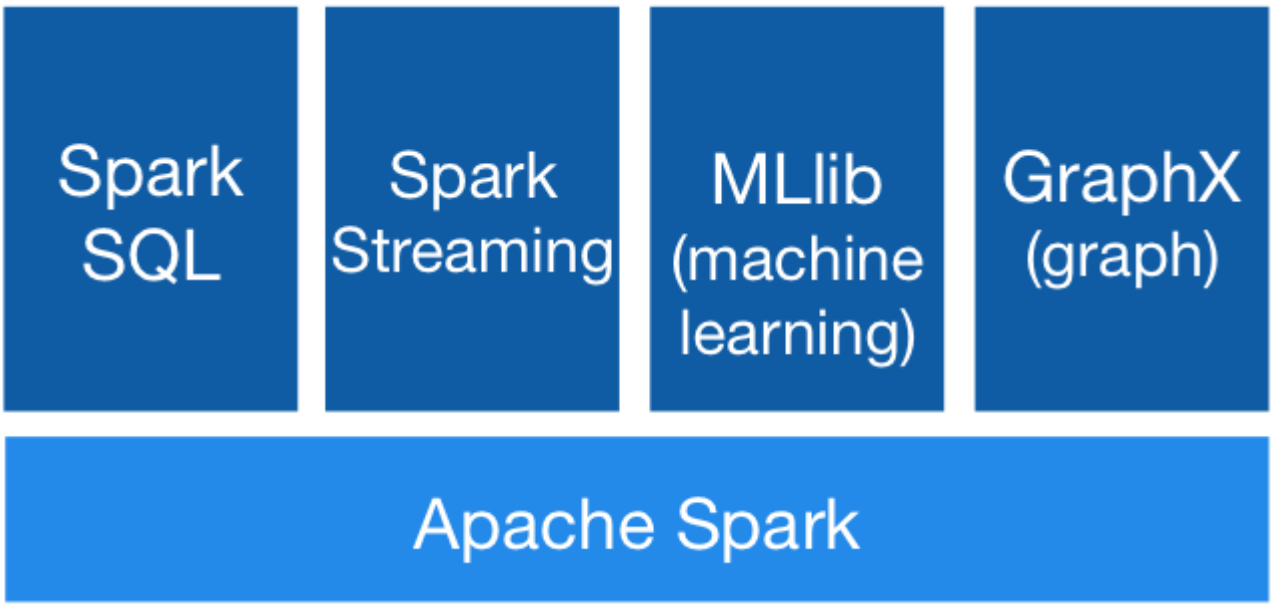
Spark提供了统一的解决方案。Spark可以用于批处理、交互式查询（Spark SQL）、实时流处理（Spark Streaming）、机器学习（Spark MLlib）和图计算（GraphX）。这些不同类型的处理都可以在同一个应用中无缝使用。Spark统一的解决方案非常具有吸引力，毕竟任何公司都想用统一的平台去处理遇到的问题，减少开发和维护的人力成本和部署平台的物力成本。



1.2.4. 兼容性

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，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。

1.3. Spark核心组件



Spark Core：

Spark Core是整个BDAS生态系统的核心组件，是一个分布式的大数据处理框架。Spark Core提供了多种资源调度管理，通过内存计算、有向无环图等机制保证分布式计算的快速，并引入了RDD的抽象保证数据的高容错性，其重要性描述如下：

- Spark Core提供了多种运行模式，不仅可以使用自身运行模式处理任务，如本地模式、Standalone，而且可以使用第三方资源调度来处理任务，如YARN、MESOS等。
- Spark Core 提供了有向无环图的分布式计算框架，并提供内存机制来支持多次迭代计算或者数据共享，大大减少迭代计算的之间读取数据的开销，这对于需要进行对此迭代的数据挖掘和分析性能有极大提升，另外在任务处理过程中移动计算而非移动数据，RDD Partition可以就近读取分布式文件系统中数据块到各个节点内存中进行计算。
- 在Spark中引入了RDD的抽象，它是分布在一组节点中的只读对象集合，这些集合是弹性的，如果数据一部分丢失，可以根据"血统"进行重建，保证了数据的高容错性。

Spark Streaming: Spark Streaming提供的处理引擎和RDD编程模型可以同时进行批处理和流处理。Spark Streaming使用的是将流数据离散化处理，通过该方式能够进行秒级一下的数据批处理。它使用DStream，简单来说就是一个弹性分布式数据集（RDD）系列，处理实时数据。

Spark Core： Spark SQL可以通过JDBC API将Spark数据集暴露出去，而且还可以用传统的BI和可视化工具在Spark数据上执行类似SQL的查询。

Spark MLlib: MLlib是一个可扩展的Spark机器学习库，由通用的学习算法和工具组成，包括二元分类、线性回归、聚类、协同过滤、梯度下降以及底层优化原语。用于机器学习和统计等场景

Spark GraphX: GraphX在Spark基础上提供了一站式的数据解决方案，可以高效地完成图计算的完整流水作业。GraphX是用于图计算和并行图计算的新的（alpha）Spark API。通过引入弹性分布式属性图（Resilient Distributed Property Graph），一种顶点和边都带有属性的有向多重图，扩展了Spark RDD。

1.4. Spark谁在用

http://spark.apache.org/powered-by.html着重介绍一下国内的公司，如阿里，腾讯，百度等

1.5. 怎么学Spark

官网:http://spark.apache.org/

examples:https://github.com/apache/spark

mail list

meet up

源码

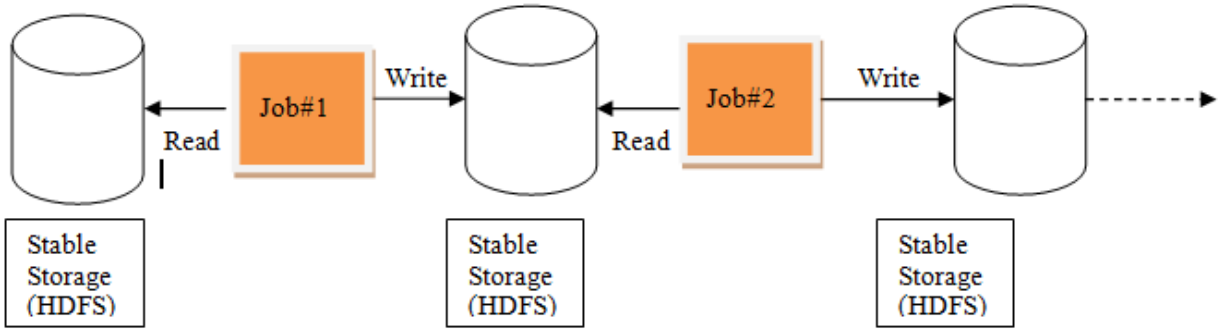
https://databricks.com/

spark survey

源码编译

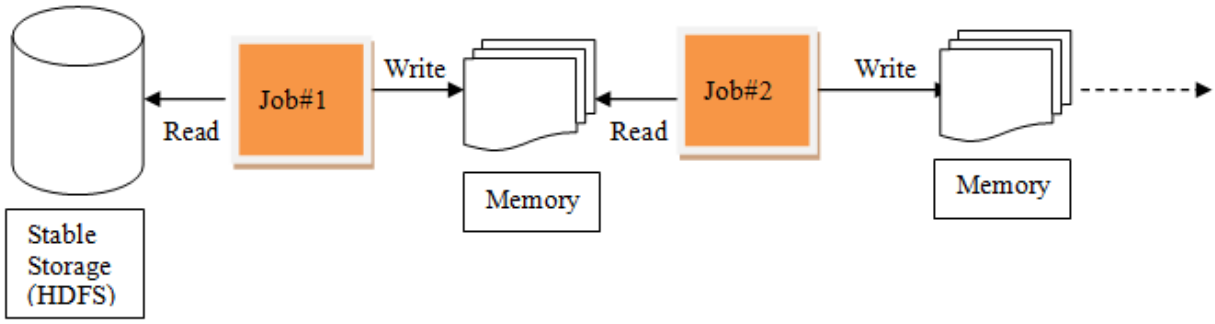
1.6. Spark和MR对比

MR中的迭代:



MR中要完成数据处理，需要写多个MR程序并且要反复的读取磁盘中文件

Spark中的迭代:



Spark中处理任务是使用多个job在连续在内存中

1、spark把运算的中间数据存放在内存，迭代计算效率更高；mapreduce的中间结果需要落地，需要保存到磁盘，这样必然会有磁盘io操做，影响性能。

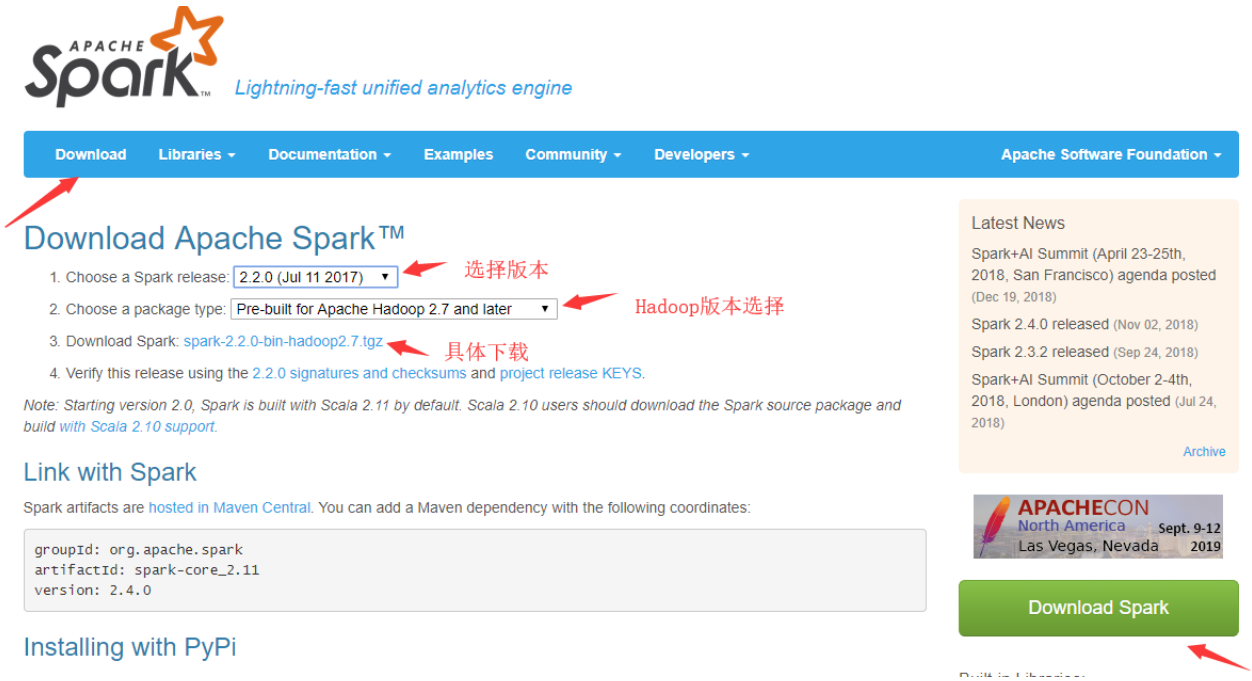
2、spark容错性高，它通过弹性分布式数据集RDD来实现高效容错，RDD是一组分布式的存储在节点内存中的只读性质的数据集，这些集合是弹性的，某一部分丢失或者出错，可以通过整个数据集的计算流程的血缘关系来实现重建；mapreduce的话容错可能只能重新计算了，成本较高。

3、spark更加通用，spark提供了transformation和action这两大类的多个功能api，另外还有流式处理sparkstreaming模块、图计算GraphX等等；mapreduce只提供了map和reduce两种操作，流计算以及其他模块的支持比较缺乏。

4、spark框架和生态更为复杂，首先有RDD、血缘lineage、执行时的有向无环图DAG、stage划分等等，很多时候spark作业都需要根据不同业务场景的需要进行调优已达到性能要求；mapreduce框架及其生态相对较为简单，对性能的要求也相对较弱，但是运行较为稳定，适合长期后台运行。

2，spark集群

2.0 Saprk安装包下载



另外的两种下载方式:

http://archive.apache.org/dist

https://github.com/apache/spark

2.1 Spark运行模式

Local 多用于本地测试，如在eclipse，idea中写程序测试等。

Standalone是Spark自带的一个资源调度框架，它支持完全分布式。

Yarn Hadoop生态圈里面的一个资源调度框架，Spark也是可以基于Yarn来计算的。

Mesos资源调度框架。

2.2 集群搭建

2.2.1 standalone模式

环境预准备，多台机器互通互联，免密登录，时间同步，

首先是怎么选择版本，包括怎么选择Hadoop版本，怎么选择Spark版本，Spark版本之间的差异是什么,怎么选择Scala，怎么选择Java

首先先搭建环境三步（standalone）：

解压安装

上传安装包并解压到指定目录

```
tar -xvf  spark-2.2.0-bin-hadoop2.7.tar -C /usr/local
```

进入到Spark安装目录

```
cd /usr/local/spark-2.2.0-bin-hadoop2.7
```

配置

进入conf目录，重命名并修改spark-env.sh.template文件

```
cd conf/
mv spark-env.sh.template spark-env.sh
vi spark-env.sh
```

在该配置文件中添加如下配置

```
export JAVA_HOME=/usr/java/jdk1.8.0_181
export SPARK_MASTER_IP=centos2
export SPARK_MASTER_PORT=7077
```

新建slaves文件

```
vi slaves
```

在该文件中添加子节点所在的位置（Worker节点）

```
centos1
centos0
```

将配置好的Spark分发到其他节点上相同目录

```
scp -r spark-2.2.0-bin-hadoop2.7/ centos1:$PWD
scp -r spark-2.2.0-bin-hadoop2.7/ centos0:$PWD
```

Spark集群配置完毕，目前是1个Master，2个Work。

可能出现的错误

```
java.net.NoRouteToHostException: No route to host(关闭防火墙，service iptables stop)
jps not found this command(java 配置问题)
```

启动

在node01上启动Spark集群

```
/usr/local/spark-2.2.0-bin-hadoop2.7/sbin/start-all.sh
```

启动后执行jps命令，会看到主节点上有Master进程，其他子节点上有Work进程

注意

浏览脚本文件start-all.sh

- Start all spark daemons.
- Starts the master on this node.
- Starts a worker on each node specified in conf/slaves

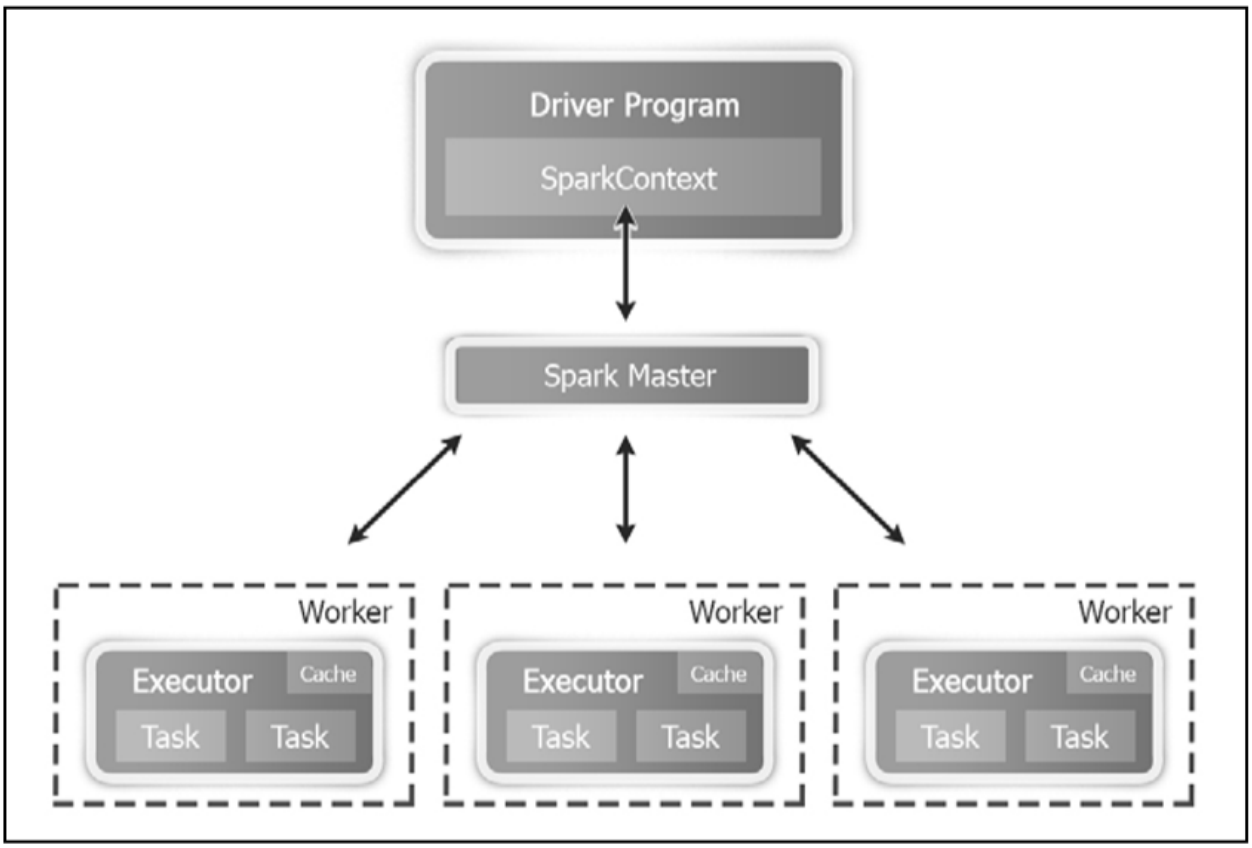
执行了 start-master.sh和start-slaves.sh

测试（两种jps和spark管理界面）

登录Spark管理界面查看集群状态（主节点）：http://node01:8080/

至此，Spark集群安装完成

该集群架构图如下：



2.2.2 Spark高可用模式

关于Spark集群的单点故障的解决，需要借助zookeeper，并且启动至少两个Master节点来实现高可用，配置如下：

Spark集群规划：node01，node02为Master；node03，node04，node05为Worker；

安装配置zookeeper集群，并启动zookeeper集群；

停止spark所有服务，修改配置文件spark-env.sh，在该配置文件中删掉SPARK_MASTER_IP并添加如下配置，

其中zk1,zk2,zk3为zk的hosts列表

```
export SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER-
Dspark.deploy.zookeeper.url=zk1,zk2,zk3-Dspark.deploy.zookeeper.dir=/spark"
```

1.在node01节点上修改slaves配置文件内容指定worker节点

2.在node01上执行**sbin/start-all.sh**脚本，然后在node02上执行**sbin/start-master.sh**启动第二个Master

2.2.3 yarn模式

配置：

修改spark-env.sh

```
export HADOOP_CONF_DIR = /opt/apps/hadoop/etc/hadoop
exprot YARN_CONF_DIR = /opt/apps/hadoop/etc/hadoop
```

yarn-client模式

yarn-cluster模式

There are two deploy modes that can be used to launch Spark applications on YARN. In `cluster` mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application. In `client` mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

Unlike other cluster managers supported by Spark in which the master's address is specified in the `--master` parameter, in YARN mode the ResourceManager's address is picked up from the Hadoop configuration. Thus, the `--master` parameter is `yarn`.

To launch a Spark application in `cluster` mode:

```
$ ./bin/spark-submit --class path.to.your.Class --master yarn --deploy-mode cluster [options] <app jar> [app options]
```

For example:

```
$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 4g \
    --executor-memory 2g \
    --executor-cores 1 \
    --queue thequeue \
    examples/jars/spark-examples*.jar \
    10
```

The above starts a YARN client program which starts the default Application Master. Then SparkPi will be run as a child thread of Application Master. The client will periodically poll the Application Master for status updates and display them in the console. The client will exit once your application has finished running. Refer to the "Debugging your Application" section below for how to see driver and executor logs.

To launch a Spark application in `client` mode, do the same, but replace `cluster` with `client`. The following shows how you can run `spark-shell` in `client` mode:

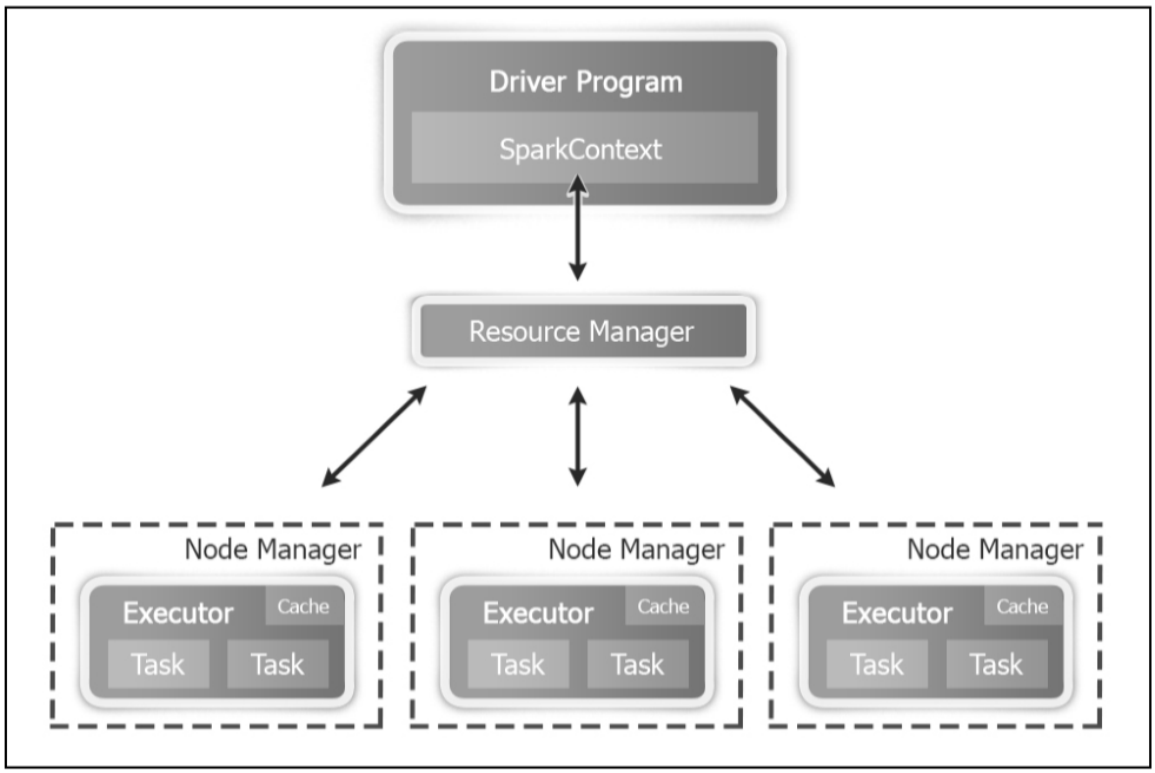
```
$ ./bin/spark-shell --master yarn --deploy-mode client
```

Spark applications on YARN run in two modes:

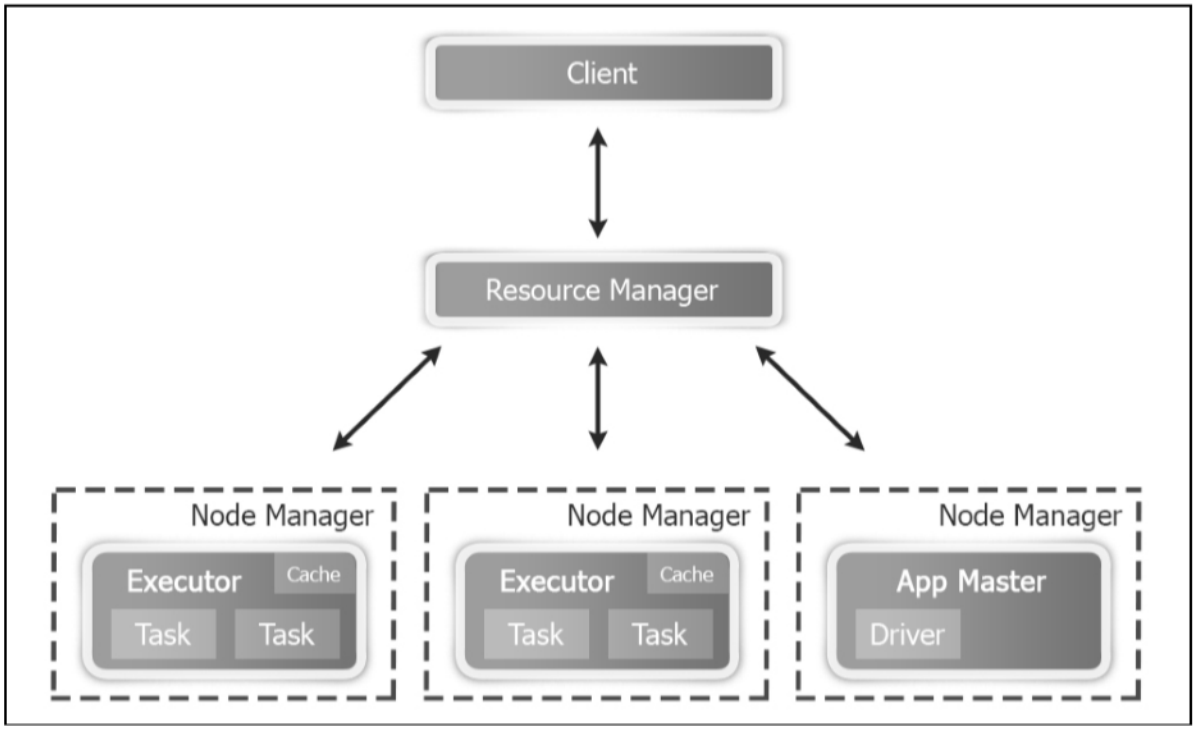
- `yarn-client`: Spark Driver runs in the client process outside of YARN cluster, and `ApplicationMaster` is only used to negotiate resources from ResourceManager
- `yarn-cluster`: Spark Driver runs in `ApplicationMaster` spawned by `NodeManager` on a slave node

The `yarn-cluster` mode is recommended for production deployments, while the `yarn-client` mode is good for development and debugging when you would like to see immediate output. There is no need to specify Spark master in either mode as it's picked from the Hadoop configuration, and the master parameter is either `yarn-client` or `yarn-cluster`.

The following figure shows how Spark is run with YARN in the client mode:



The following figure shows how Spark is run with YARN in the cluster mode:



In the YARN mode, the following configuration parameters can be set:

```
--num-executors: Configure how many executors will be allocated
--executor-memory: RAM per executor
--executor-cores: CPU cores per executor
```

2.2.4 配置Job History Server

在运行Spark应用程序的时候，driver会提供一个webUI给出应用程序的运行信息，但是该webUI随着应用程序的完成而关闭端口，也就是 说，Spark应用程序运行完后，将无法查看应用程序的历史记录。Spark history server就是为了应对这种情况而产生的，通过配置，Spark应用程序在运行完应用程序之后，将应用程序的运行信息写入指定目录，而Spark history server可以将这些运行信息装载并以web的方式供用户浏览。

```
1.启动HDFS
start-dfs.sh
创建directory目录
hdfs dfs -mkdir /directory

2.进入到spark安装目录conf目录下
cd /opt/software/spark-2.2.0-bin-hadoop2.7/conf

3.将spark-default.conf.template复制为spark-default.conf
mv spark-defaults.conf.template spark-defaults.conf
vi  spark-defaults.conf在文件的末尾添加
spark.eventLog.enabled           true  开启日志
spark.eventLog.dir               hdfs://hadoop01:8020/directory 存储路径

spark.eventLog.compress          true 是否压缩
```

参数描述：
spark.eventLog.dir：Application在运行过程中所有的信息均记录在该属性指定的路径下
spark.eventLog.compress 这个参数设置history-server产生的日志文件是否使用压缩，true为使用，false为不使用。这个参数务可以成压缩哦，不然日志文件岁时间积累会过

4.修改spark-env.sh文件，添加如下配置
export SPARK_HISTORY_OPTS="-Dspark.history.ui.port=4000 -Dspark.history.retainedApplications=10
-Dspark.history.fs.logDirectory=hdfs://hadoop01:8020/directory"

spark.history.ui.port=4000 调整WEBUI访问的端口号为4000
spark.history.fs.logDirectory=hdfs://hadoop01:8020/directory 配置了该属性后，在start-history-server.sh时就无需再显式的指定路径，Spark History Server页面只展示该指定路径下的信息
spark.history.retainedApplications=10 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。

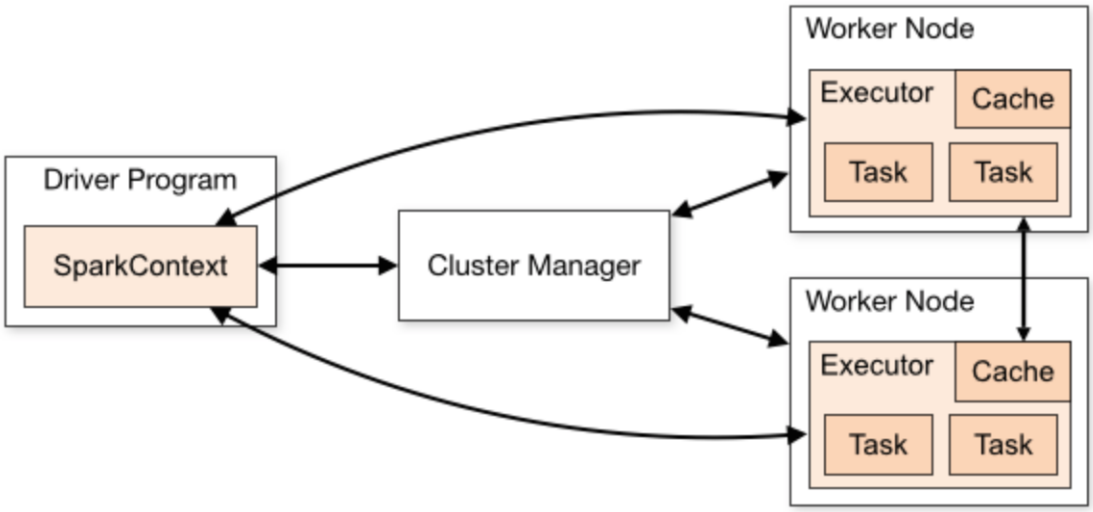
5.配置完成后分发文件到相应节点
scp -r ./spark-env.sh root@hadoop02:$PWD
scp -r ./spark-defaults.conf root@hadoop02:$PWD

ps:最好不要是用IE内核的浏览器不然效果是显示不出来的
   启动的时候是
   start-all.sh start-history-server.sh

2.3 集群架构(standalone模式)

介绍spark任务是如何在集群中执行，以及涉及到的组件。

spark集群架构图：



sparkContext

sparkContext在Spark应用程序的执行过程中起着主导作用，它负责与程序和spark集群进行交互，包括申请集群资源、创建RDD、accumulators及广播变量等。

SparkContext是Spark的入口，相当于应用程序的main函数。

目前在一个JVM进程中可以创建多个SparkContext，但是只能有一个active级别的。如果你需要创建一个新的SparkContext实例，必须先调用stop方法停掉当前active级别的SparkContext实例。

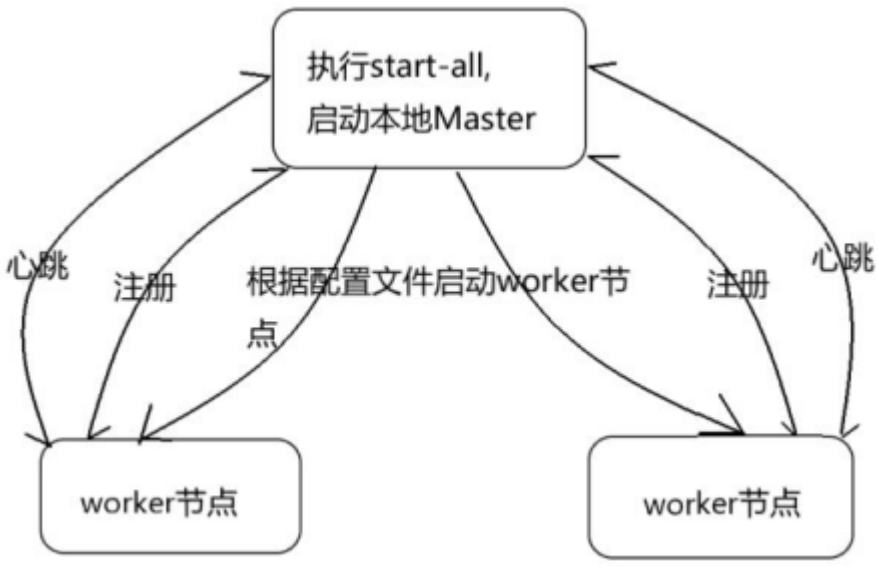
注意点：（1）不同的Spark应用程序对应该不同的Executor，这些Executor在整个应用程序执行期间都存在并且Executor中可以采用多线程的方式执行Task。这样做的好处是，各个Spark应用程序的执行是相互隔离的（主要意思是两个方面，不同应用程序任务调度是独立的，不同应用程序的任务跑在不同JVM上）。除Spark应用程序向外部存储系统写数据进行数据交互这种方式外，各Spark应用程序（sparkContext实例）间无法进行数据共享。（2）Spark对于其使用的集群资源管理器没有感知能力，只要它能对Executor进行申请并通信即可。这意味着不管使用哪种资源管理器（如Mesos/Yarn），其执行流程都是不变的。这样Spark可以不同的资源管理器进行交互。（3）Spark应用程序(驱动程序)在整个执行过程中要与Executors进行来回通信。（4）Driver端负责Spark应用程序任务的调度，因此最好Driver应该靠近Worker节点，最好在一个局域网。

与集群相关概念

The following table summarizes terms you'll see used to refer to cluster concepts:

Term	Meaning
Application	User program built on Spark. Consists of a *driver program* and *executors* on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. `save`, `collect`); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

2.4 spark集群启动流程



1，启动Master进程

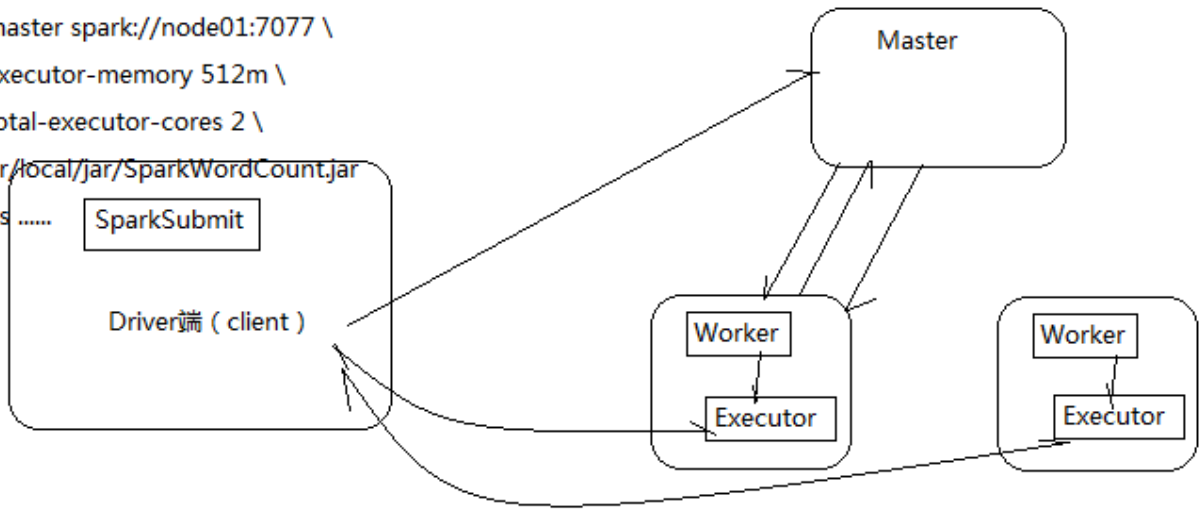
2，Master开始解析conf目录的slaves配置文件，找到相应的Worker节点，开始启动Worker进程

3，Worker进程开始向Master发送注册信息

4，Master接受到Worker的注册信息后并保存到内存和磁盘里，然后向Worker发送注册成功信息

5，Worker开始和Master建立心跳，Master每次接收到心跳后更新WorkerInfo的最后一次心跳时间。

```
$Spark/bin/spark-submit \
--class com.qf.gp1703.day09.SparkWordCount \
--master spark://node01:7077 \
--executor-memory 512m \
--total-executor-cores 2 \
/usr/local/jar/SparkWordCount.jar
args ……
```



Spark集群启动流程：

1、启动Master进程

2、Master开始解析conf目录的slaves配置文件，找到启动Worker的相应节点，开始启动Worker进程

3、Worker进程开始向Master发送注册信息

4、Master接收到Worker的注册信息后并保存到内存和磁盘里，
接下来Master向Worker发送注册成功信息（MasterURL）

5、Worker接收到Master的URL后，开始和Master建立心跳，Master每次接收到心跳后会更新workerInfo的最后一次心跳时间

Spark任务提交流程：

1、Driver端首先启动SparkSubmit进程，启动后开始和Master进行通信，此时创建了一个非常重要的对象（SparkContext），接着向Master发送任务信息

2、Master接收到任务信息后，开始资源调度，此时会和所有的Worker进行通信，找到比较空闲的Worker，
并通知Worker来拿取任务和启动相应的Executor

3、Executor启动后，开始与Driver进行反向注册，接下来Driver开始把任务发送给相应的Executor，
Executor开始计算任务

2.5 集群任务部署

介绍spark-submit

- spark-submit定义：

应用程序部署工具bin/spark-submit，可以完成将Spark应用程序在local、Standalone、YARN、Mesos等集群上的快捷部署。

- 查看所有spark-submit参数：

```
./spark-submit
```

```
[root@centos0 bin]# ./spark-submit
Usage: spark-submit [options] <app jar | python file> [app arguments]
Usage: spark-submit --kill [submission ID] --master [spark://...]
Usage: spark-submit --status [submission ID] --master [spark://...]

Options:
  --master MASTER_URL         spark://host:port, mesos://host:port, yarn, or local.
  --deploy-mode DEPLOY_MODE   Whether to launch the driver program locally ("client") or
                              on one of the worker machines inside the cluster ("cluster")
                              (Default: client).
  --class CLASS_NAME          Your application's main class (for Java / Scala apps).
  --name NAME                 A name of your application.
  --jars JARS                 Comma-separated list of local jars to include on the driver
                              and executor classpaths.
  --packages                  Comma-separated list of maven coordinates of jars to include
                              on the driver and executor classpaths. Will search the local
                              maven repo, then maven central and any additional remote
                              repositories given by --repositories. The format for the
                              coordinates should be groupId:artifactId:version.
  --exclude-packages          Comma-separated list of groupId:artifactId, to exclude while
                              resolving the dependencies provided in --packages to avoid
                              dependency conflicts.
  --repositories              Comma-separated list of additional remote repositories to
                              search for the maven coordinates given with --packages.
  --py-files PY_FILES         Comma-separated list of .zip, .egg, or .py files to place
                              on the PYTHONPATH for Python apps.
  --files FILES               Comma-separated list of files to be placed in the working
                              directory of each executor.

  --conf PROP=VALUE           Arbitrary Spark configuration property.
  --properties-file FILE      Path to a file from which to load extra properties. If not
                              specified, this will look for conf/spark-defaults.conf.

  --driver-memory MEM         Memory for driver (e.g. 1000M, 2G) (Default: 1024M).
  --driver-java-options       Extra Java options to pass to the driver.
  --driver-library-path       Extra library path entries to pass to the driver.
  --driver-class-path         Extra class path entries to pass to the driver. Note that
                              jars added with --jars are automatically included in the
                              classpath.

  --executor-memory MEM       Memory per executor (e.g. 1000M, 2G) (Default: 1G).

  --proxy-user NAME           User to impersonate when submitting the application.
```

spark-submit参数以及参数解读

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --executor-memory 20G \
  --total-executor-cores <number of cores> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

spark-submit使用（例子演示）

提交Spark提供的利用蒙特·卡罗算法求π的例子，其中100这个参数是计算因子

```
/usr/local/spark-2.2.0-bin-hadoop2.7/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://centos2:7077 \
/usr/local/spark-2.2.0-bin-hadoop2.7/examples/jars/spark-examples_2.11-2.2.0.jar
100
```

查看

webUI（查看网页UI变化）

JPS（查看各结点任务执行前后进程变化）

（任务执行后会有资源释放）

再执行

```
/usr/local/spark-1.6.3-bin-hadoop2.6/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://centos0:7077 \
--executor-memory 1G
--total-executor-cores 2 \
/usr/local/spark-1.6.3-bin-hadoop2.6/lib/spark-examples-1.6.3-hadoop2.6.0.jar\
100
```

观察前后资源分配变化

问题：

提交一个spark程序到spark集群，会产生那些进程？

1，spark-submit（driver）提交任务，任务调度

2，Executor，执行计算任务

提交任务如何指定多个master？

```
/usr/local/spark-1.6.3-bin-hadoop2.6/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://node01:7077,node02:7077 \
--executor-memory 1G
--total-executor-cores 2 \
/usr/local/spark-1.6.3-bin-hadoop2.6/lib/spark-examples-1.6.3-hadoop2.6.0.jar\
100
```

错误：

WARN TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient memory

解决：

可能原因如下：

1.因为提交任务的节点不能和spark工作节点交互，因为提交完任务后提交任务节点上会起一个进程，展示任务进度，大多端口为4044，工作节点需要反馈进度给该该端口，所以如果主机名或者IP在hosts中配置不正确。所以检查下主机名和ip是否配置正确

检查 centos0:8080下total core是否为0

2.有可能是内存不足

检查内存

conf.set("spark.executor.memory", "3000m")

Make sure to set SPARK_LOCAL_IP andSPARK_MASTER_IP.

查看8080端口，确保一些workers保持Alive状态，确保 some cores 是可利用的。

3.driver在不停地工作，一直处于running状态，而从前端8080页面又可看出程序的状态一直是WAITING。submit所在的shell输出driver信息后便退出了，目前还未得到程序运行结果，这显然是因为该程序处于WAITING状态，还未运行。而导致这一结果，应该跟日志的输出有关。值得一提的是，到目前为止，出了更改了对master ip和port的配置以及worker的配置外，其他深入的配置还需进一步研究。

3. Spark Shell介绍

3.1 Spark Shell 是什么

Spark Shell也是一个客户端。

spark-shell是Spark自带的交互式Shell程序，方便用户进行交互式编程，用户可以在该命令行下用Scala或者Python编写Spark程序。Spark Shell程序一般用作Spark程序测试练习来用。Spark Shell属于Spark的特殊应用程序，我们可以在这个特殊的应用程序中提交应用程序。

3.2 Spark Shell启动模式

spark-shell启动有两种模式：local模式和cluster模式。

1）local模式

```
# local模式 ，本地执行，没有提交任务到集群
# local模式仅在本机启动一个SparkSubmit进程，没有与集群建立联系。
# 在spark UI上查找不到应用程序的提交。
/usr/local/spark-1.6.3-bin-hadoop2.6/bin/spark-shell
```

2) cluster模式

```
# cluster模式
/usr/local/spark-1.6.3-bin-hadoop2.6/bin/spark-shell \
--master spark://node01:7077 \
--executor-memory 1g \
--total-executor-cores 2
```

```
参数说明：
--master spark://node01:7077 指定Spark集群的Master的地址
--executor-memory 1g 指定每个worker可用内存为1G
--total-executor-cores 2 指定整个集群使用的cup核数为2个
注：spark-shell中已经默认将SparkContext和SQLContext对象初始化为sc和sqlContext。如果需要用到，则直接应用sc和sqlContext即可
```

注意两种模式的区别：

1.看webui,一个显示没有应用程序运行，一个显示有

2，看各台jps，是否有executor启动

3.3 使用spark shell实现WordCount

1.首先启动hdfs

2.向hdfs上传一个文件到hdfs://node01:9000/words.txt

文件内容为：

```
hello scala
hello java
hello scala
```

3.在spark shell中用Scala语言编写spark程序

```
sc.textFile("hdfs://node01:9000/words.txt").flatMap(_.split(""))
.map((_,1)).reduceByKey(_+_).saveAsTextFile("hdfs://node01:9000/out")
```

4.使用hdfs命令查看结果

```
hdfs dfs -ls hdfs://node01:9000/out/p*
```

说明：
sc是SparkContext对象，该对象是提交spark程序的入口
textFile(hdfs://node01:9000/words.txt)是hdfs中读取数据
flatMap(_.split(""))先map再压平
map((_,1))将单词和1构成元组
reduceByKey(_+_)按照key进行reduce，并将value累加
saveAsTextFile("hdfs://node01:9000/out")将结果写入到hdfs中

4. 编写spark应用程序

4.0 通过IDEA创建Spark工程

对工程中的pom.xml文件配置

```
 <!-- 声明公有的属性 -->
<properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <encoding>UTF-8</encoding>
        <scala.version>2.11.8</scala.version>
        <spark.version>2.2.0</spark.version>
        <hadoop.version>2.7.1</hadoop.version>
        <scala.compat.version>2.11</scala.compat.version>
    </properties>
<!-- 声明并引入公有的依赖 -->
    <dependencies>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>${scala.version}</version>
        </dependency>
        <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>${hadoop.version}</version>
        </dependency>
    </dependencies>

<!-- 配置构建信息 -->
    <build>
        <!-- 资源文件夹 -->
        <sourceDirectory>src/main/scala</sourceDirectory>
        <!-- 声明并引入构建的插件 -->
        <plugins>
             <!-- 用于编译Scala代码到class -->
            <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>
                <!-- 程序打包 -->
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.4.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                             <!-- 过滤掉以下文件，不打包 ：解决包重复引用导致的打包错误-->
                            <filters>
                                <filter><artifact>*:*</artifact>
                                    <excludes>
                                        <exclude>META-INF/*.SF</exclude>
                                        <exclude>META-INF/*.DSA</exclude>
                                        <exclude>META-INF/*.RSA</exclude>
                                    </excludes>
                                </filter>
                            </filters>
                            <transformers>
                                <!-- 打成可执行的jar包 的主方法入口-->
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                    <mainClass></mainClass>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>

    </build>
```

4.1 在IDEA中用Scala实现WordCount

```
`import org.apache.spark.rdd.RDD`
`import org.apache.spark.{SparkConf, SparkContext}`

`object SparkWC {`

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

        // 创建配置文件信息类，setAppName设置应用程序名称
        // setMaster 设置为本地测试模式
        // local[2]：本地用两个线程模拟集群运行任务
        // local：本地用一个线程模拟集群运行任务
        // local[*]：本地用所有空闲的线程模拟集群运行任务
        val conf: SparkConf = new SparkConf()
          .setAppName("SparkWC")
          .setMaster("local")
        // 创建Spark上下文对象，也叫集群入口类
        val sc: SparkContext = new SparkContext(conf)

        // 读取HDFS的数据
        val lines: RDD[String] = sc.textFile(args(0))
    //    val lines: RDD[String] = sc.parallelize(Array("hello tom", "hello jerry hello jerry",
"hello hello"))

        // 对数据做单词计数
        val words: RDD[String] = lines.flatMap(_.split(" "))
        val tuples: RDD[(String, Int)] = words.map((_, 1))
        val reduced: RDD[(String, Int)] = tuples.reduceByKey(_+_)
        val res: RDD[(String, Int)] = reduced.sortBy(_._2, false)

        // 打印结果
        // println(res.collect.toBuffer)

        // 保存
        res.saveAsTextFile(args(1))

        sc.stop()

      }

`}`
```

4.2 在IDEA中用Java实现WordCount

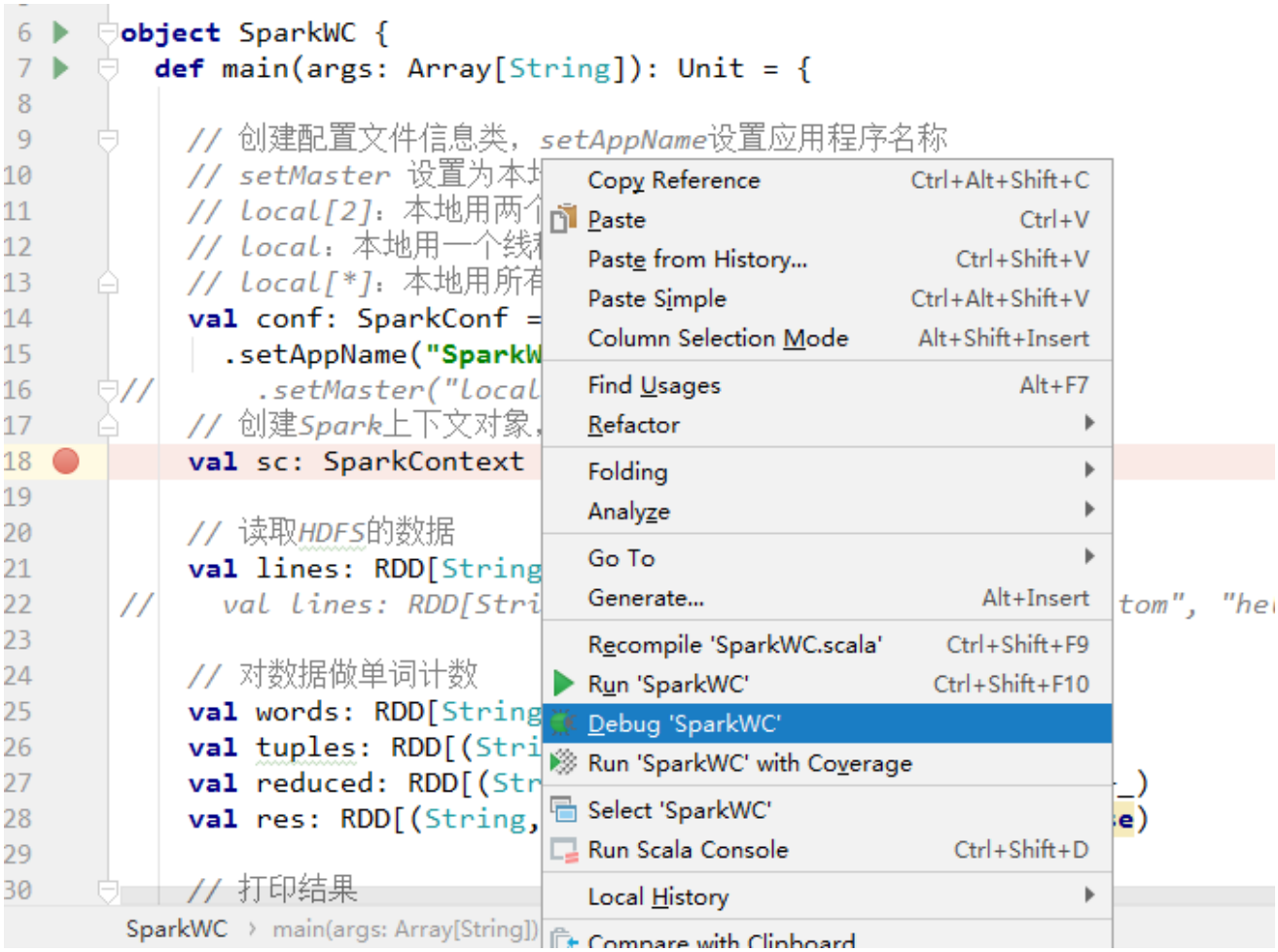
```
`import org.apache.spark.SparkConf;`
```

```
`import org.apache.spark.api.java.JavaPairRDD;`
`import org.apache.spark.api.java.JavaRDD;`
`import org.apache.spark.api.java.JavaSparkContext;`
`import org.apache.spark.api.java.function.FlatMapFunction;`
`import org.apache.spark.api.java.function.Function2;`
`import org.apache.spark.api.java.function.PairFunction;`
`import scala.Tuple2;`
`import java.util.Arrays;`

`public class JavaWC {`
    public static void main(String[] args) {
        // 配置信息类
        final SparkConf conf = new SparkConf()
                .setAppName("JavaWc");
                //.setMaster("local[2]");
        // 上下文对象
        final JavaSparkContext jsc = new JavaSparkContext(conf);

        // 获取数据
        final JavaRDD<String> lines = jsc.textFile(args[0]);

        // 切分
        final JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public Iterable<String> call(String s) throws Exception {
                return Arrays.asList(s.split(" "));
            }
        });

        // 把每个单词生成一个个元组
        final JavaPairRDD<String, Integer> pairRDD = words.mapToPair(
                new PairFunction<String, String, Integer>() {
            @Override
            public Tuple2<String, Integer> call(String s) throws Exception {
                return new Tuple2<String, Integer>(s, 1);
            }
        });

        // 聚合
        final JavaPairRDD<String, Integer> reduced = pairRDD.reduceByKey(new Function2<Integer,
Integer, Integer>() {
            @Override
            public Integer call(Integer v1, Integer v2) throws Exception {
                return v1 + v2;
            }
        });

        // Java并没有提供sortBy算子，如果需要一value来进行排序时，此时需要把数据反转一下，
        // 排序完成后，再反转回来
        final JavaPairRDD<Integer, String> swaped = reduced.mapToPair(new
PairFunction<Tuple2<String, Integer>, Integer, String>() {
            @Override

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

```
                return tup.swap();
            }
        });



        // 排序
        final JavaPairRDD<Integer, String> sorted = swaped.sortByKey(false);

        // 把数据再反转回来
        final JavaPairRDD<String, Integer> res = sorted.mapToPair(new
PairFunction<Tuple2<Integer, String>, String, Integer>() {

            @Override
            public Tuple2<String, Integer> call(Tuple2<Integer, String> tup) throws Exception {
                return tup.swap();
            }
        });

`//        System.out.println(reduced.collect());`
        res.saveAsTextFile(args[1]);
        jsc.stop();

    }

`}`
```

4.3 IDEA中代码调试

首先在要调试的应用程序的类中打上断点，在main方法中右键点选Debug 'SparkWC'选项，或Alt+Shift+F9

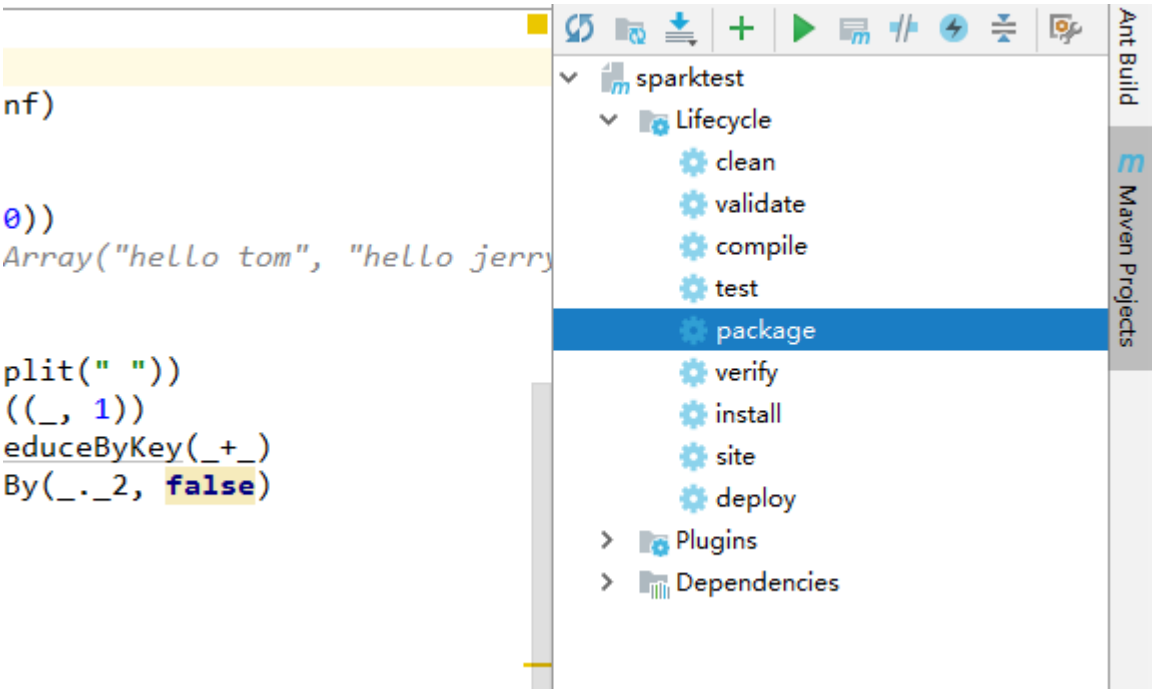


4.4 打包运行

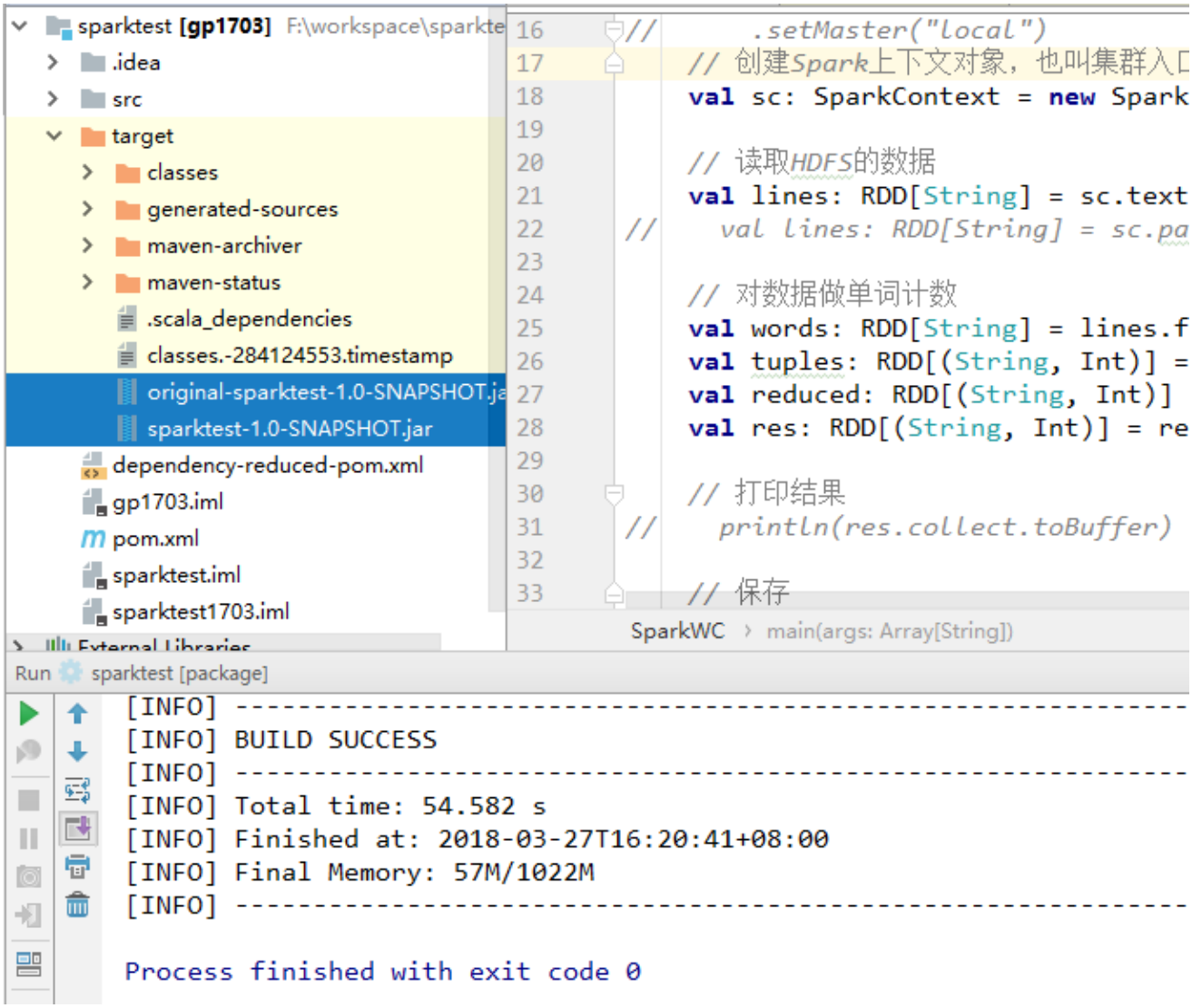
使用Maven打包：首先修改pom.xml中的main class

```
                <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
                            <mainClass>com.qf.gp1701.sparktest.day07.SparkWC</mainClass>
                        </transformer>
                    </transformers>
                </configuration>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>

</project>
```

点击IDEA右侧的Maven Projects选项，先双击clean清除以前生成的包，再双击package进行打包



选择编译成功的jar包，并将该jar上传到Spark集群中的某个节点上，会生成两个jar包，一个是不带引用的包，一个是带引用的包



启动hdfs和Spark集群

```
/usr/local/hadoop-2.6.4/sbin/start-dfs.sh

/usr/local/spark-1.6.3-bin-hadoop2.6/sbin/start-all.sh
```

使用spark-submit命令提交Spark应用（注意参数的顺序）

```
/usr/local/spark-1.6.3-bin-hadoop2.6/bin/spark-submit\

--class com.qf.spark.WordCount \

--master spark://node01:7077 \

--executor-memory 1G \

--total-executor-cores 2 \

/root/spark-mvn-1.0-SNAPSHOT.jar \

hdfs://node01:9000/wc \

hdfs://node01:9000/out
```

查看程序执行结果

hdfs dfs -cat hdfs://node01:9000/out/part-00000

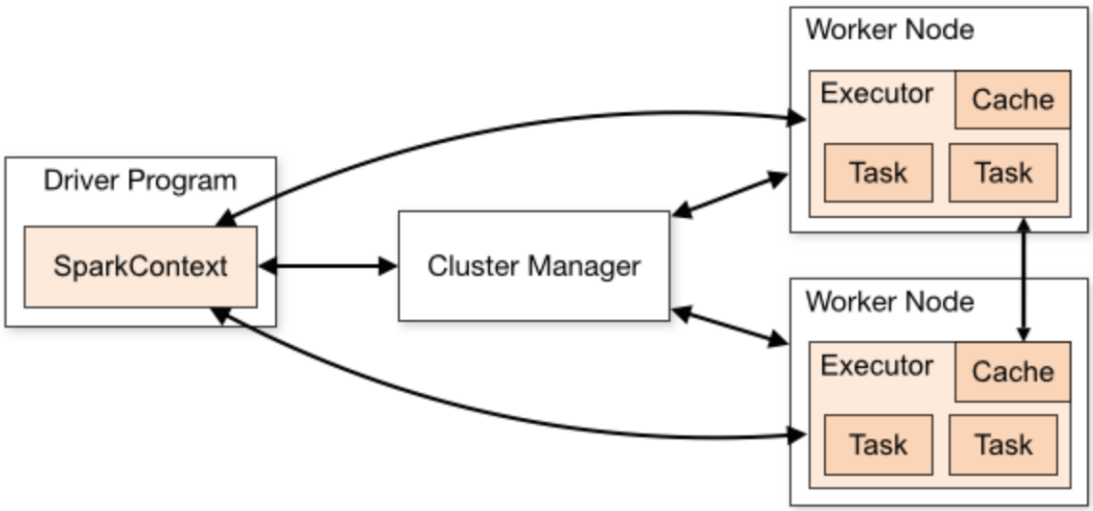
5. Spark任务提交流程

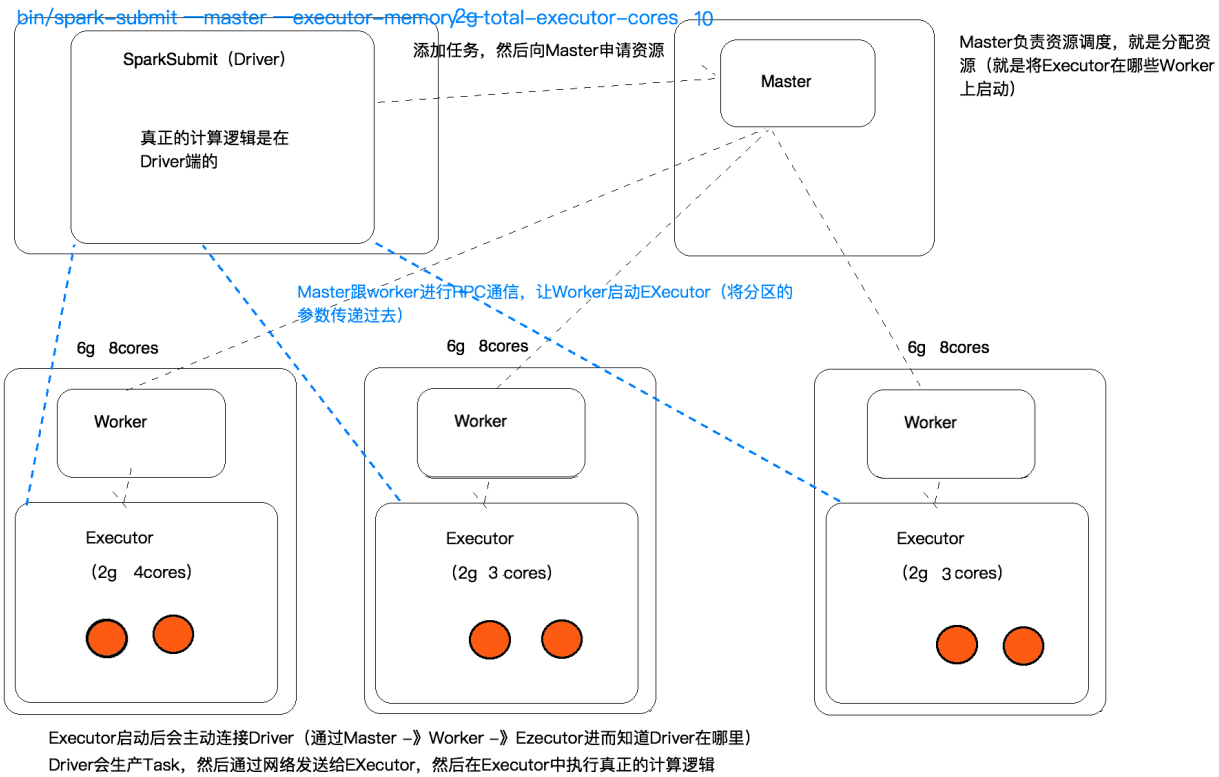
Spark任务提交流程：

1）Driver端首先启动SparkSubmit进程，启动后开始于Master进行通信，此时创建了一个非常重要的对象（SparkContext)，接着向Master发送任务信息；

2）Master接收到任务信息后，开始资源调度，此时会和所有的Worker进行通信，找到比较空闲的Worker，并通知Worker来取任务和启动相应的Executor；

3) Executor启动后，开始与Driver进行反向注册，接下来Driver开始把任务发送给相应的Executor，Executor开始计算任务。





问题：

worker为什么要重新启动Executor执行任务？

在不同进程执行不同的应用程序任务，某一个task失败，不会影响其他任务的执行；

程序的并行执行。