# Spark 入门

## 一、 课程目标

目标 1：熟悉Spark 相关概念

目标 2：搭建一个 Spark 集群

目标 3：编写简单的Spark 应用程序

## 二、 Spark 概述

2.1 什么是 Spark （官网：http://spark.apache.org ）

Spark 是一种快速、通用、可扩展的大数据分析引擎，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 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。

2.2 为什么要学Spark

Spark是一个开源的类似于Hadoop MapReduce的通用的并行计算框架，Spark

基于 map reduce 算法实现的分布式计算，拥有 Hadoop MapReduce 所具有的优

点；但不同于MapReduce的是Spark中的Job中间输出和结果可以保存在内存中，

从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需

要迭代的map reduce 的算法。

Spark 是 MapReduce 的替代方案，而且兼容 HDFS、Hive，可融入 Hadoop 的生态

系统，以弥补MapReduce 的不足。

2.3 Spark 特点

2.3.1 快 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。

2.3.2 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。

2.3.4 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。

2.3.5 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。

## 三、 Spark 集群安装

3.1 下载 spark 安装包

下载地址spark官网：http://spark.apache.org/downloads.html

这里我们使用 spark-2.0.2-bin-hadoop2.7版本.

3.2 规划安装目录

/opt/bigdata

3.3 解压安装包

tar-zxvfspark-2.0.2-bin-hadoop2.7.tgz

3.4 重命名目录

mvspark-2.0.2-bin-hadoop2.7spark

3.5 修改配置文件

配置文件目录在/opt/bigdata/spark/conf

vi spark-env.sh 修改文件(先把 spark-env.sh.template 重命名

为spark-env.sh)

#配置java 环境变量

export JAVA\_HOME=/opt/bigdata/jdk1.7.0\_67

#指定spark 老大 Master 的 IP

export SPARK\_MASTER\_HOST=hdp-node-01

#指定spark 老大 Master 的端口

export SPARK\_MASTER\_PORT=7077

vi slaves修改文件(先把slaves.template重命名为slaves)

hdp-node-02

hdp-node-03

3.6 拷贝配置到其他主机

通过scp命令将spark的安装目录拷贝到其他机器上

scp-r/opt/bigdata/sparkhdp-node-02:/opt/bigdata

scp-r/opt/bigdata/sparkhdp-node-03:/opt/bigdata

3.7 配置 spark 环境变量

将spark添加到环境变量,添加以下内容到/etc/profile

export SPARK\_HOME=/opt/bigdata/spark

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

注意最后source/etc/profile 刷新配置

3.7 启动 spark

#在主节点上启动spark

/opt/bigdata/spark/sbin/start-all.sh

3.8 停止 spark

#在主节点上停止spark集群

/opt/bigdata/spark/sbin/stop-all.sh

3.9 spark 的 的 web 界面

正常启动 spark 集群后，可以通过访问 http://hdp-node-01:8080,查看 spark 的 web 界面，

查看相关信息。

## 四、 Spark HA 高可用部署

4.1 高可用部署说明

SparkStandalone集群是Master-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：

（1）基于文件系统的单点恢复(Single-Node Recovery with Local File

System)。

主要用于开发或测试环境。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sparkApplication和worker的注册信息。

（2）基于zookeeper的Standby Masters(Standby Masters with ZooKeeper)。

用于生产模式。其基本原理是通过 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。

4.2 基于 zookeeper 的 的 Spark HA 高可用集群部署

该 HA 方案使用起来很简单，首先需要搭建一个 zookeeper 集群，然后启动

zooKeeper集群，最后在不同节点上启动Master。具体配置如下：

(1)vimspark-env.sh

注释掉exportSPARK\_MASTER\_HOST=hdp-node-01

(2)在spark-env.sh添加SPARK\_DAEMON\_JAVA\_OPTS，内容如下：

export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER

-Dspark.deploy.zookeeper.url=hdp-node-01:2181,hdp-node-02:2181,hdp-node-03:2181

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

参数说明

spark.deploy.recoveryMode：恢复模式（Master 重新启动的模式）

有三种：(1)ZooKeeper (2) FileSystem(3)NONE

spark.deploy.zookeeper.url：ZooKeeper 的Server 地址

spark.deploy.zookeeper.dir：保存集群元数据信息的文件、目录。

包括 Worker，Driver 和 Application。

注意：

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，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。

、 五、 Spark 角色介绍

Spark是基于内存计算的大数据并行计算框架。因为其基于内存计算，比

Hadoop 中 MapReduce 计算框架具有更高的实时性，同时保证了高效容错性和可

伸缩性。从 2009 年诞生于 AMPLab 到现在已经成为 Apache 顶级开源项目，并成

功应用于商业集群中，学习Spark就需要了解其架构。

Spark架构图如下：

Spark架构使用了分布式计算中 master-slave 模型，master 是集群中含

有master进程的节点，slave是集群中含有worker进程的节点。

DriverProgram：运⾏main函数并且新建SparkContext的程序。

Application：基于 Spark 的应用程序，包含了 driver 程序和集群上的

executor。

ClusterManager：指的是在集群上获取资源的外部服务。目前有三种类型

（1）Standalone:spark原生的资源管理，由Master负责资源的分配

（2）ApacheMesos:与hadoopMR兼容性良好的一种资源调度框架

（3）HadoopYarn:主要是指Yarn中的ResourceManager

WorkerNode：集群中任何可以运行Application 代码的节点，在 Standalone

模式中指的是通过slaves文件配置的Worker节点，在Spark on Yarn 模式

下就是NodeManager节点

Executor：是在一个worker node 上为某应⽤启动的⼀个进程，该进程负责

运⾏行任务，并且负责将数据存在内存或者磁盘上。每个应⽤都有各自独立

的executor。

Task：被送到某个executor上的工作单元。

## 五、 识 初识 Spark 程序

5.1 执行第一个 spark 程序

普通模式提交任务：

bin/spark-submit \

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

--master spark:// hdp-node-01 :7077 \

--executor-memory 1G \

--total-executor-cores 2 \

examples/jars/spark-examples\_2.11-2.0.2.jar \

10

该算法是利用蒙特·卡罗算法求圆周率 PI，通过计算机模拟大量的随机数，

最终会计算出比较精确的π。

高可用模式提交任务：

在高可用模式下，因为涉及到多个 Master，所以对于应用程序的提交就有了

一点变化，因为应用程序需要知道当前的 Master 的 IP 地址和端口。这种 HA 方

案处理这种情况很简单，只需要在SparkContext指向一个Master列表就可以了，

如spark://host1:port1,host2:port2,host3:port3，应用程序会轮询列表，找

到活着的Master。

bin/spark-submit \

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

--master spark://hdp-node-01:7077,hdp-node-02:7077,hdp-node-03:7077 \

--executor-memory 1G \

--total-executor-cores 2 \

examples/jars/spark-examples\_2.11-2.0.2.jar \

10

5.2 启动 Spark-Shell

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编写spark程序。

5.2.1 运行 spark-shell --master local[N] 读取本地文件

单机模式：通过本地N个线程跑任务，只运行一个SparkSubmit进程。

（1）需求

读取本地文件，实现文件内的单词计数。本地文件words.txt内容如下：

hellome

helloyou

helloher

（2）运行 spark-shell --master local[2]

观察启动的进程：

（3）编写 scala 代码

sc.textFile("file:///root///words.txt").flatMap(\_.split("

")).map((\_,1)).reduceByKey(\_+\_).collect

代码说明：

sc：Spark-Shell 中已经默认将 SparkContext 类初始化为对象 sc。用户代码如

果需要用到，则直接应用sc即可。

textFile:读取数据文件

flatMap:对文件中的每一行数据进行压平切分,这里按照空格分隔。

map:对出现的每一个单词记为1（word，1）

reduceByKey:对相同的单词出现的次数进行累加

collect:触发任务执行，收集结果数据。

（4）观察结果：

5.2.2 运行 spark-shell --master local[N] 读取 HDFS 上数据

(1)、整合spark 和 HDFS，修改配置文件

在spark-env.sh ，添加 HADOOP\_CONF\_DIR 配置，指明了 hadoop 的配置文件

后，默认它就是使用的hdfs 上的文件

export HADOOP\_CONF\_DIR=/opt/bigdata/hadoop-2.6.4/etc/hadoop

(2)、再启动启动hdfs，然后重启 spark 集群

(3)、向hdfs 上传一个文件到 hdfs://hdp-node-01:9000/words.txt

(4)、在spark shell 中用 scala 语言编写 spark 程序

sc.textFile("/words.txt").flatMap(\_.split("

")).map((\_,1)).reduceByKey(\_+\_).collect

5.2.3 运行 spark-shell 指定具体的 master 地址

（1）需求:

spark-shell运行时指定具体的 master 地址，读取 HDFS 上的数据，做单词

计数，然后将结果保存在HDFS上。

（2）执行启动命令：

spark-shell \

--master spark://hdp-node-01:7077 \

--executor-memory 1g \

--total-executor-cores 2

参数说明：

--master spark://hdp-node-01:7077 指定 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

注意：

如果启动 spark shell 时没有指定 master 地址，但是也可以正常启动 spark shell

和执行spark shell 中的程序，其实是启动了 spark 的 local 模式，该模式仅在本机

启动一个进程，没有与集群建立联系。

（2）编写scala代码

sc.textFile("/words.txt").flatMap(\_.split("

")).map((\_,1)).reduceByKey(\_+\_).saveAsTextFile("/wc")

saveAsTextFile:保存结果数据到文件中

（3）查看hdfs上结果

5.3 在 在 IDEA 中编写 WordCount 程序

spark-shell 仅在测试和验证我们的程序时使用的较多，在生产环境中，通常

会在IDEA 中编写程序，然后打成 jar 包，最后提交到集群。最常用的是创建一个

Maven 项目，利用 Maven 来管理 jar 包的依赖。

（1）.创建一个项目

（2）.选择 Maven 项目，然后点击 next

（3）.填写 maven 的 GAV，然后点击 next

（4）填写项目名称，然后点击finish

（5）.创建好 maven 项目后，点击 Enable Auto-Import

（6）配置Maven 的 pom.xml

<properties>

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

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

<spark.version>2.0.2</spark.version>

</properties>

<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>

<build>

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

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

<plugins>

<plugin>

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

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

<version>3.2.0</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>

</plugins>

</build>

（7）添加src/main/scala 和 src/test/scala，与 pom.xml 中的配置保持一致

（8）新建一个scala class，类型为 Object

（9）.编写 spark 程序

e package cn.test.spark

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

t import org.apache.spark.rdd.RDD

t object WordCount {

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

//设置 spark的配置文件信息

l val sparkConf: SparkConf = w new SparkConf().setAppName("WordCount")

//构建 sparkcontext 上下文对象，它是程序的入口,所有计算的源头

l val sc: SparkContext = w new SparkContext(sparkConf)

//读取文件

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

//对文件中每一行单词进行压平切分

l val words: RDD[String] = file.flatMap(\_.split(" "))

//对每一个单词计数为 1 转化为(单词，1)

l val wordAndOne: RDD[(String, Int)] = words.map(x=>(x,1))

//相同的单词进行汇总 前一个下划线表示累加数据，后一个下划线表示新数据

l val result: RDD[(String, Int)] = wordAndOne.reduceByKey(\_+\_)

//保存数据到 HDFS

result.saveAsTextFile(args(1))

sc.stop()

}

}

（10）.使用 Maven 打包：

点击idea右侧的MavenProject选项

点击Lifecycle,选择package，然后点击RunMavenBuild

（11）.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

（12）.首先启动 hdfs和Spark 集群

启动hdfs

/opt/bigdata/hadoop-2.6.4/sbin/start-dfs.sh

启动spark

/opt/bigdata/spark/sbin/start-all.sh

(13).使用 spark-submit 命令提交 Spark 应用（注意参数的顺序）

spark-submit\

--classcn.test.spark.WordCount\

--masterspark://hdp-node-01:7077\

--executor-memory1g\

--total-executor-cores2\

/root/spark-1.0-SNAPSHOT.jar\

/words.txt\

/spark\_out

这里通过spark-submit提交任务到集群上。用的是spark的Standalone模式

Standalone 模式是 Spark 内部默认实现的一种集群管理模式，这种模式是通过

集群中的Master来统一管理资源。

1) 查看Spark 的 web管理界面

地址：192.168.200.160:8080

2) 查看HDFS 上的结果文件

hdfsdfs-cat/spark\_out/part\*

(hello,3)

(me,1)

(you,1)

(her,1)

5.4 使用 java 语言写 编写 spark wordcount 程序

packagecn.itcast;

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;

/\*\*

\* java 代码实现 spark 的 WordCount

\*/

public class WordCountJava {

public static void main(String[] args) {

// todo:1、构建 sparkconf,设置配置信息

SparkConf sparkConf = new

SparkConf().setAppName("WordCount\_Java").setMaster("local[2]");

// todo:2、构建 java 版的 sparkContext

JavaSparkContext sc = new JavaSparkContext(sparkConf);

// todo:3、读取数据文件

JavaRDD<String> dataRDD= sc.textFile("d:/data/words1.txt");

// todo:4、对每一行单词进行切分

JavaRDD<String> wordsRDD = dataRDD.flatMap(new

FlatMapFunction<String, String>() {

@Override

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

String[] words = s.split(" ");

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

}

});

// todo:5、给每个单词计为 1

//Spark 为包含键值对类型的 RDD 提供了一些专有的操作。这些 RDD 被称为

PairRDD。

// mapToPair 函数会对一个 RDD 中的每个元素调用f 函数，其中原来 RDD 中的

每一个元素都是 T 类型的，

// 调用 f 函数后会进行一定的操作把每个元素都转换成一个<K2,V2>类型的对象,

其中 Tuple2 为多元组

JavaPairRDD<String, Integer> wordAndOnePairRDD =

wordsRDD.mapToPair(new PairFunction<String, String,Integer>(){

@Override

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

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

}

});

// todo:6、相同单词出现的次数累加

JavaPairRDD<String, Integer> resultJavaPairRDD =

wordAndOnePairRDD.reduceByKey(new Function2<Integer,Integer, Integer>() {

@Override

public Integer call(Integer v1, Integerv2) throws Exception {

return v1 + v2;

}

});

// todo:7、反转顺序

JavaPairRDD<Integer, String> reverseJavaPairRDD =

resultJavaPairRDD.mapToPair(new PairFunction<Tuple2<String, Integer>, Integer,

String>() {

@Override

public Tuple2<Integer, String> call(Tuple2<String, Integer>tuple)

throws Exception {

return new Tuple2<Integer, String>(tuple.\_2, tuple.\_1);

}

});

// todo:8、把每个单词出现的次数作为 key，进行排序，并且在通过 mapToPair

进行反转顺序后输出

JavaPairRDD<String, Integer> sortJavaPairRDD =

reverseJavaPairRDD.sortByKey(false).mapToPair(new

PairFunction<Tuple2<Integer, String>, String, Integer>() {

@Override

public Tuple2<String, Integer> call(Tuple2<Integer, String>tuple)

throws Exception {

return new Tuple2<String, Integer>(tuple.\_2,tuple.\_1);

//或者使用 tuple.swap() 实现位置互换，生成新的 tuple;

}

});

// todo:执行输出

System. out .println(sortJavaPairRDD.collect());

// todo:关闭 sparkcontext

sc.stop();

}

}