

# 1. Spark 概述

## 1.1. 什么是 Spark (官网: <http://spark.apache.org>)

spark 中文官网: <http://spark.apachecn.org>

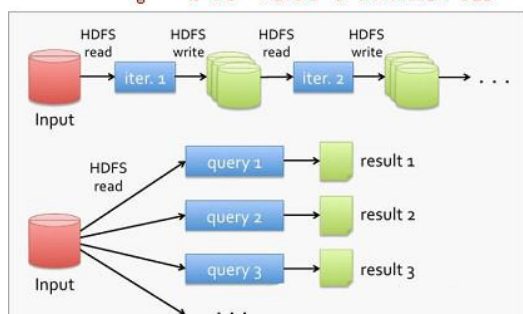
Spark 是一种快速、通用、可扩展的大数据分析引擎, 2009 年诞生于加州大学伯克利分校 AMPLab, 2010 年开源, 2013 年 6 月成为 Apache 孵化项目, 2014 年 2 月成为 Apache 顶级项目。目前, Spark 生态系统已经发展成为一个包含多个子项目的集合, 其中包含 SparkSQL、Spark Streaming、GraphX、MLlib 等子项目, **Spark 是基于内存计算的大数据并行计算框架**。Spark 基于内存计算, 提高了在大数据环境下数据处理的实时性, 同时保证了高容错性和高可伸缩性, 允许用户将 Spark 部署在大量廉价硬件之上, 形成集群。

## 1.2. Spark 和 Hadoop 的比较

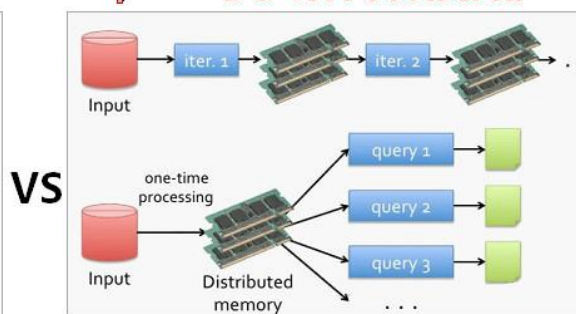
mapreduce 读 - 处理 - 写磁盘 - 读 - 处理 - 写

spark 读 -- 处理 -- 处理 --- (需要的时候) 写磁盘 --- - 写

**Hadoop: 两步计算, 磁盘存储**



**Spark: 多步计算, 内存存储**



Spark 是在借鉴了 MapReduce 之上发展而来的, 继承了其分布式并行计算的优点并改进了 MapReduce 明显的缺陷, (spark 与 hadoop 的差异) 具体如下:

首先, Spark 把中间数据放到内存中, 迭代运算效率高。MapReduce 中计算结果需要落地, 保存到磁盘上, 这样势必会影响整体速度, 而 Spark 支持 DAG 图的分布式并行计算的编程框架, 减少了迭代过程中数据的落地, 提高了处理效率。(延迟加载)

其次, Spark 容错性高。Spark 引进了弹性分布式数据集 RDD (Resilient Distributed Dataset) 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, 如果数据集一部分丢失, 则可以根据“血统”(即允许基于数据衍生过程) 对它们进行重建。另外在 RDD 计算时可以通过 CheckPoint 来实现容错。

最后, Spark 更加通用。mapreduce 只提供了 Map 和 Reduce 两种操作, Spark 提供的数据集操作类型有很多, 大致分为: Transformations 和 Actions 两大类。Transformations 包括 Map、Filter、FlatMap、Sample、GroupByKey、ReduceByKey、Union、Join、Cogroup、MapValues、Sort 等多种操作类型, 同时还提供 Count, Actions 包括 Collect、Reduce、Lookup 和 Save 等操作。

支持的运算平台, 支持的开发语言更多。

spark 4 种开发语言:

scala,java,python,R

总结:

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。

## 1.3. Spark 特点

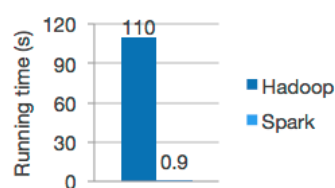
### 1.3.1. 快

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。

#### Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.



Logistic regression in Hadoop and Spark

### 1.3.2. 易用

Spark 支持 Java、Python 和 Scala 和 R 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。

#### Ease of Use

Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python and R shells.

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
           .map(lambda word: (word, 1))
           .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

### 1.3.3. 通用

一站式解决方案   离线处理   实时处理 (streaming)   sql

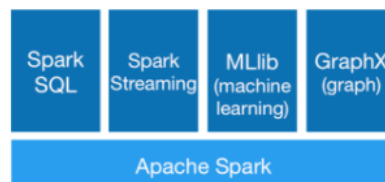
Spark 提供了统一的解决方案。Spark 可以用于批处理、交互式查询 (Spark SQL)、实时流处理 (Spark Streaming)、机器学习 (Spark MLlib) 和图计算 (GraphX)。这些不

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。

## Generality

Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including [SQL and DataFrames](#), [MLlib](#) for machine learning, [GraphX](#), and [Spark Streaming](#). You can combine these libraries seamlessly in the same application.



### 1.3.4. 兼容性

Spark 可以非常方便地与其他开源产品进行融合。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。

## Runs Everywhere

Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

You can run Spark using its [standalone cluster mode](#), on [EC2](#), on Hadoop YARN, or on [Apache Mesos](#). Access data in [HDFS](#), [Cassandra](#), [HBase](#), [Hive](#), [Tachyon](#), and any Hadoop data source.



## 2. spark 的部署模式:

- 1, local            本地模式, 只要机器上有 spark 的安装包, 仅仅是用于测试  
不写 master    -- master local 一个线程 local[2] local[\*] 模拟使用多个线程
- 2, Standalone    spark 自带的集群模式            --master spark://hdp-01:7077
- 3, yarn            把 spark 任务, 运行在 yarn    --master yarn
- 4, mesos           把 spark 任务, 运行在 mesos 资源调度平台上    --master mesos

# 3. Spark 集群安装

## 3.1. 安装

**注意:** 安装 spark 时, 无需安装 scala

jdk 必须要 jdk1.8+

### 3.1.1. 下载 Spark 安装包

#### Download Apache Spark™

1. Choose a Spark release:  选择spark的版本
  2. Choose a package type:  选择spark对应的hadoop版本
  3. Choose a download type:
  4. Download Spark: [spark-2.2.0-bin-hadoop2.7.tgz](#)
  5. Verify this release using the [2.2.0 signatures and checksums](#) and [project release KEYS](#).
- Note: Starting version 2.0, Spark is built with Scala 2.11 by default. Scala 2.10 users should download the Spark source package and build with Scala 2.10 support.

上传 spark-安装包到 Linux 上

解压安装包到指定位置

```
# tar -zxvf spark-2.2.0-bin-hadoop2.7.tgz -C apps/
```

Spark 安装包目录结构:

bin	可执行脚本
conf	配置文件
data	示例程序使用数据
examples	示例程序
jars	依赖 jar 包
LICENSE	
licenses	
NOTICE	
python	pythonAPI
R	R 语言 API
README.md	
RELEASE	
sbin	集群管理命令

### 3.1.2. 机器部署

准备 4 台 Linux 服务器, 安装好 JDK, 最低要求 2 台

hdp-01 192.168.8.11

hdp-02 192.168.8.12

hdp-03 192.168.8.13

hdp-04 192.168.8.14

master: hdp-01

workers: hdp-02 hdp-03 hdp-04

### 3.1.3. 部署 standalone 集群

确保集群中各节点的防火墙是关闭的。

查看防火墙状态

```
# service iptables status
```

关闭防火墙

```
# service iptables stop
```

永久关闭防火墙

```
# chkconfig iptables off
```

确保主节点到各从节点的免密登录配置好了

从 Master 节点到 worker 节点的免密登录

在 master 机器上执行：

```
# ssh-keygen
```

```
# for i in 2 3 4; do ssh-copy-id hdp-0$i; done
```

进入到 Spark 安装目录

```
# cd apps/ spark-2.2.0-bin-hadoop2.7
```

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

```
# cd conf/
```

```
# mv spark-env.sh.template spark-env.sh
```

```
# vim spark-env.sh
```

在该配置文件中添加如下配置

```
export JAVA_HOME=/usr/local/jdk
export SPARK_MASTER_HOST=hdp-01
export SPARK_MASTER_PORT=7077
```

保存退出

重命名并修改 slaves.template 文件

```
# mv slaves.template slaves
# vim slaves
```

**思考?** 为什么要修改 slaves 配置文件

在该文件中添加子节点所在的位置 (Worker 节点)

```
hdp-02
hdp-03
hdp-04
```

保存退出

将配置好的 Spark 文件夹拷贝到其他节点上

单独拷贝:

```
# scp -r /root/apps/spark-2.2.0-bin-hadoop2.7/ hdp-02:/root/apps/
```

批量拷贝:

```
# cd /root/apps
# for i in {2..4};do scp -r spark-2.2.0-bin-hadoop2.7 hdp-0$i:$PWD ;done
```

Spark 集群配置完毕, 目前是 1 个 Master, 3 个 Worker, 在 Master(hdp-01)上启动 Spark 集群



### 3.1.4. 启停操作

单独启动 master (在 master 安装节点上) :

```
# start-master.sh
```

启动众 worker(在 Master 所在节点上执行)

```
# start-slaves.sh
```

这里获取的是 slaves 文件中的主机名

分别停止:

```
# stop-slaves.sh
```

```
# stop-master.sh
```

批量脚本启动:

```
# start-all.sh
```

停止:

```
# stop-all.sh
```

为了能方便使用, 配置一下环境变量:

```
export SPARK_HOME=/root/apps/spark-2.2.0-bin-hadoop2.7
```

```
export JAVA_HOME=/usr/local/jdk
export HADOOP_HOME=/root/apps/hadoop
export SCALA_HOME=/usr/local/scala
export SPARK_HOME=/root/apps/spark-2.2.0-bin-hadoop2.7
export ZK_HOME=/root/apps/zookeeper-3.4.6
export PATH=.:$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin:$JAVA_HOME/bin:$HADOOP_HOME/bin:$SCALA_HOME/bin:$ZK_HOME/bin
```

配置环境变量的注意事项:

hadoop/sbin 的目录和 spark/sbin 可能会有命令冲突:

```
start-all.sh stop-all.sh
```


解决方案:

1, 把其中一个框架的 sbin 从环境变量中去掉;

2, 改名 `hadoop/sbin/start-all.sh` --> `start-all-hdp.sh`

启动后执行 `jps` 命令, 主节点上有 Master 进程, 其他子节点上有 Worker 进程,

登录 Spark 管理界面查看集群状态 (主节点): <http://hdp-01:8080/>

**Spark Master at spark://hdp-01:7077**

URL: spark://hdp-01:7077  
REST URL: spark://hdp-01:6066 (cluster mode)  
Alive Workers: 4  
Cores in use: 4 Total, 0 Used  
Memory in use: 10.9 GB Total, 0.0 B Used  
Applications: 0 Running, 0 Completed  
Drivers: 0 Running, 0 Completed  
Status: ALIVE

**Workers**

Worker Id	Address	State	Cores	Memory
<a href="#">worker-20171008003426-192.168.8.13-55939</a>	192.168.8.13:55939	ALIVE	1 (0 Used)	2.7 GB (0.0 B Used)
<a href="#">worker-20171008003447-192.168.8.12-40770</a>	192.168.8.12:40770	ALIVE	1 (0 Used)	2.7 GB (0.0 B Used)
<a href="#">worker-20171008005334-192.168.8.11-35328</a>	192.168.8.11:35328	ALIVE	1 (0 Used)	2.7 GB (0.0 B Used)
<a href="#">worker-20171011081514-192.168.8.14-40911</a>	192.168.8.14:40911	ALIVE	1 (0 Used)	2.7 GB (0.0 B Used)

查看机器内存的命令: `free -m`

默认情况下: spark 会占用机器上的所有 cores,

memory 呢, 会默认的使用 ram - 1G

默认配置:

<http://spark.apache.org/docs/latest/spark-standalone.html>

SPARK_WORKER_CORES	Total number of cores to allow Spark applications to use on the machine (default: all available cores).
SPARK_WORKER_MEMORY	Total amount of memory to allow Spark applications to use on the machine, e.g. 1000m, 2g (default: total memory minus 1 GB); note that each application's <i>individual</i> memory is configured using its <code>spark.executor.memory</code> property.

到此为止, Spark 集群安装完毕。

## 4. 执行 Spark 程序

使用 `spark-shell` 命令和 `spark-submit` 命令来提交 spark 任务。

当执行测试程序，使用 spark-shell，spark 的交互式命令行

提交 spark 程序到 spark 集群中运行时，spark-submit

## 4.1. 执行第一个 spark 示例程序

```
spark-submit --class org.apache.spark.examples.SparkPi  
/root/apps/spark/examples/jars/spark-examples_2.11-2.2.0.jar 100
```

该算法是利用蒙特·卡罗算法求 PI(圆周率)

spark 任务提交的方式：

```
spark-submit --master spark://hdp-01:7077 --class xxx.SparkPi /root/xx.jar 输  
入输出参数
```

怎么用：

提交正式任务，或者有 jar 包，使用 spark-submit ；本地测试，选用 spark-shell

## 4.2. 启动 Spark Shell

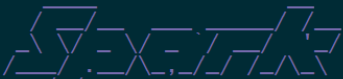
spark-shell 用命令行的方式提交任务到集群的一个客户端。spark-shell 是 Spark 自带的交互式 Shell 程序，方便用户进行交互式编程，用户可以在该命令行下用 scala 编写 spark 程序。

## 4.2.1. 启动 spark shell

直接启动 spark-shell 默认使用的是 local 模式，和 spark 集群无关

只要把 spark 安装包解压了，就可以运行 local 模式

```
[root@hdp-01 ~]# spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
17/10/08 04:38:18 WARN NativeCodeLoader: Unable to load native-hadoop library for
ng builtin-java classes where applicable
17/10/08 04:38:20 WARN SparkContext: Use an existing SparkContext, some configura
ect.
Spark context web UI available at http://192.168.8.11:4040
Spark context available as 'sc' (master = local[*], app id = local-1507408699750)
Spark session available as 'spark'.
Welcome to

 version 2.0.2
```

local 模式没有指定 master 地址，仅在本机启动一个进程 (SparkSubmit)，没有与集群建立联系。但是也可以正常启动 spark shell 和执行 spark shell 中的程序

指定集群模式启动：

hdfs://hdp-01:9000

spark 的协议 URI: **spark://hdp-01:7077**

# spark-shell --master

```
[root@hdp-04 ~]# spark-shell --master spark://hdp-01:7077
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel)
18/01/20 00:08:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your pl
using builtin-java classes where applicable
18/01/20 00:08:58 WARN ObjectStore: Failed to get database global_temp, returning NoSuchO
ption
Spark context web UI available at http://192.168.8.14:4040
Spark context available as 'sc' (master = spark://hdp-01:7077, app id = app-2018010305101
Spark session available as 'spark'.
```

在 webUI 界面，可以查看到正在运行的程序：

## Running Applications

Application ID	Name	Cores	Memory per Node
app-20171008054602-0000 (kill)	Spark shell	4	1024.0 MB

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

## 4.2.2. 在 spark shell 中编写 WordCount 程序

1.首先启动 hdfs

2.向 hdfs 上传一个文件到 hdfs://hdp-01:9000/wordcount/input/a.txt

3.在 spark shell 中用 scala 语言编写 spark 程序

```
scala> sc.textFile("hdfs://hdp-01:9000/wordcount/input/")
```

spark 是懒加载的，所以这里并没有真正执行任务。可使用 collect 方法快速查看数据。

lazy 执行的，只有调用了 action 方法，才正式开始运行。

```
scala> sc.textFile("hdfs://hdp-01:9000/wordcount/input/").flatMap(_.
```

```
split("")).map(_._1).reduceByKey(_ + _).sortBy(_._2,false).collect
```

**注意：**这些 flatMap, map 等方法是 RDD 上的方法，要区分于原生的 scala 方法。

和原生 scala 的方法名称有的相同，但属于不通的类的方法，底层实现完全不一致。

原生的方法：对单机的数组或集合进行操作。

RDD 上的方法：

RDD 是 spark 的计算模型，RDD 上有很多的方法，这些方法通常称为算子，主要有两类算

子，一类是 transform，一类是 action，transform 是懒加载的。

```
scala>sc.textFile("hdfs://hdp-01:9000/wordcount/input/").flatMap(_.split("
")).map((_,1)).reduceByKey(_+_).saveAsTextFile("hdfs://hdp-01:9000/wordcount/out
spark1")
```

#### 4.使用 hdfs 命令查看结果

```
# hadoop fs -ls /wordcount/outspark1
```

说明：

sc 是 SparkContext 对象，该对象是提交 spark 程序的入口

textFile(hdfs://hdp-01:9000/wordcount/intput/a.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://hdp-01:9000/outspark1")将结果写入到 hdfs 中

spark 中的方法很多，这些方法统称为算子。一共有两类算子（transform，action）

spark 是懒加载的，transform 方法并不会立即执行，只有当程序遇到 action 的时候才会

被执行。collect 算子是一个 action

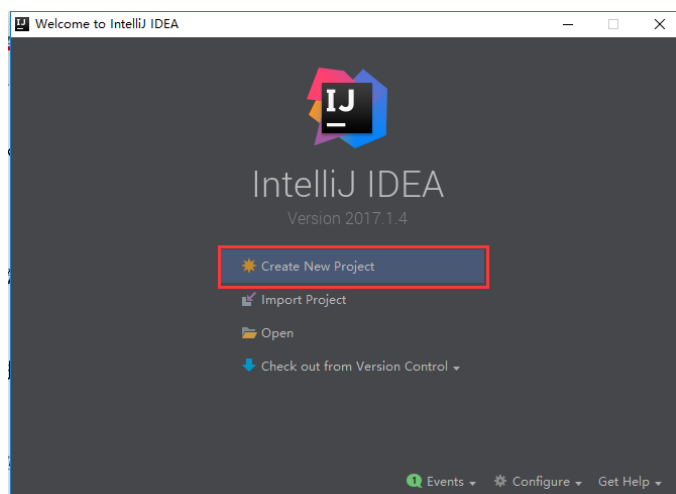
collect: 收集数据到本地

## 4.3. 在 IDEA 中编写 WordCount 程序

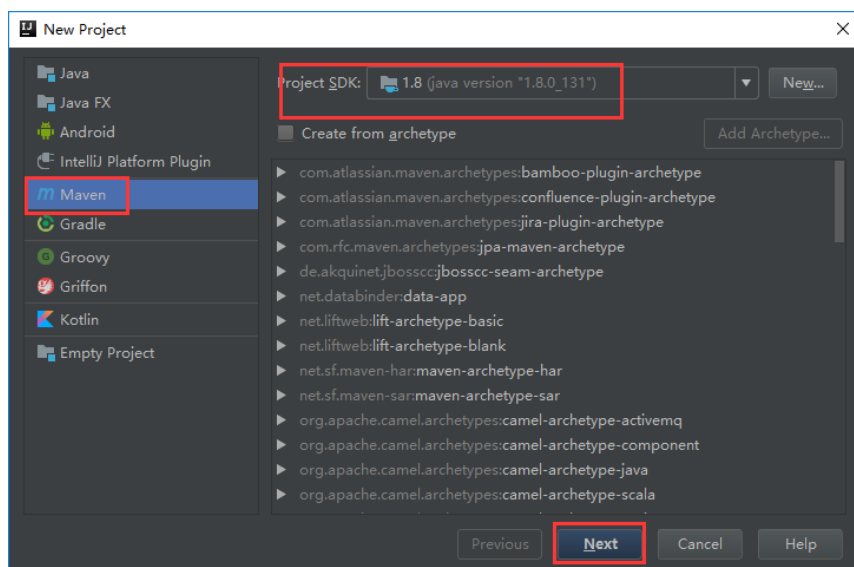
spark shell 仅在测试和验证我们的程序时使用的较多，在生产环境中，通常会在 IDE 中开发程序，然后打成 jar 包，然后提交到集群，最常用的是创建一个 Maven 项目，利用 Maven 来管理 jar 包的依赖。

### 4.3.1. scalaAPI 的 wordcount

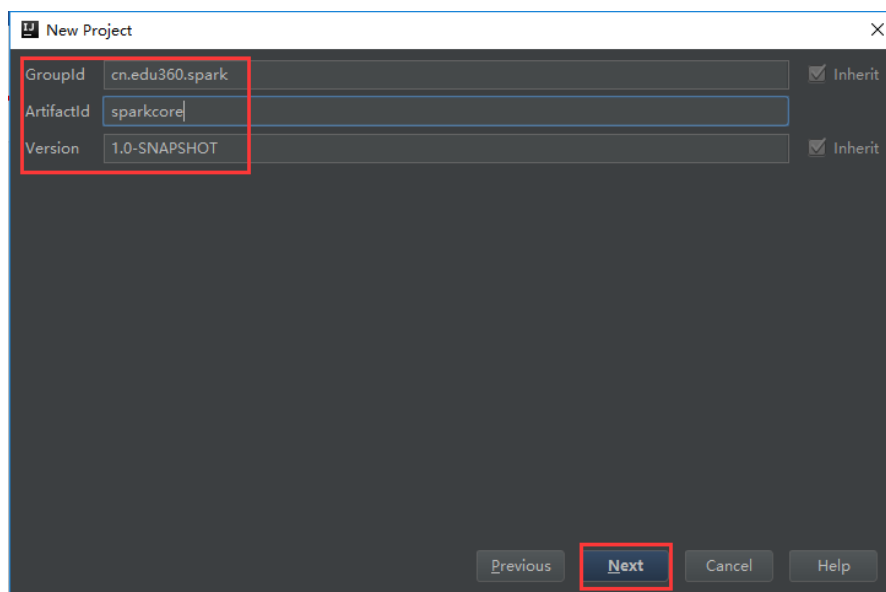
#### 1. 创建一个项目



#### 2. 选择 Maven 项目，然后点击 next

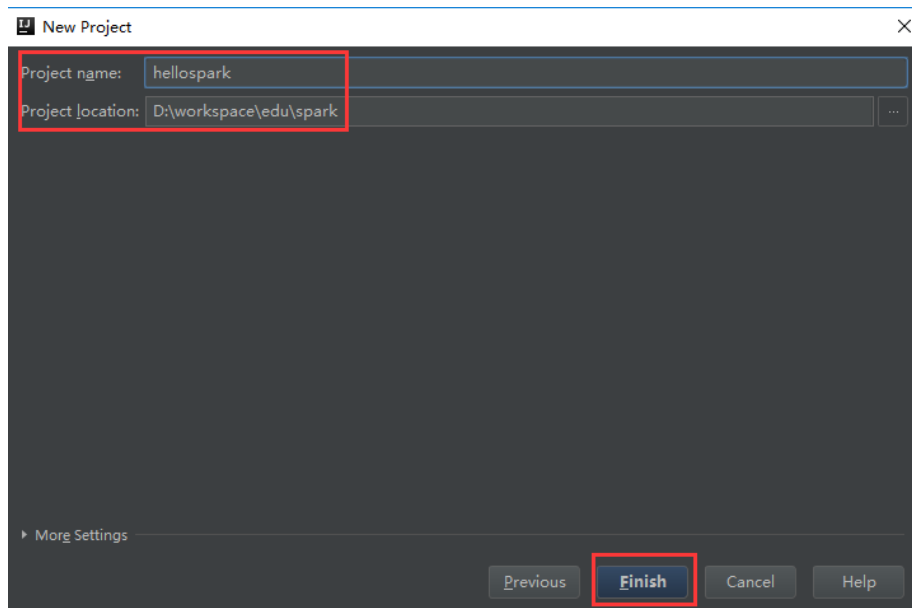


3.填写 maven 的 GAV, 然后点击 next

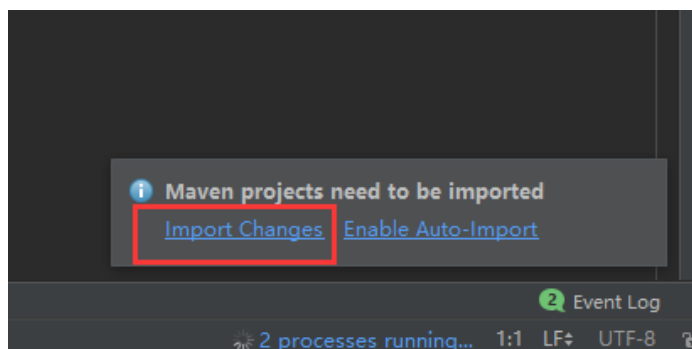


4.填写项目名称, 然后点击 finish





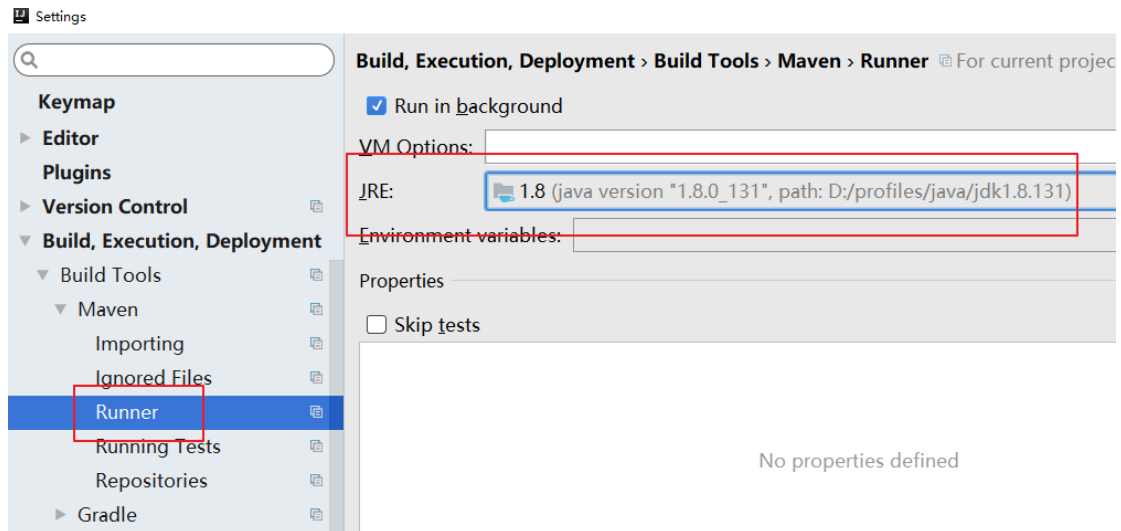
5.创建好 maven 项目后, 点击 Import Changes 手动导入, 点击 Enable Auto-Import 可自动导入



6.配置 Maven 的 pom.xml

详见 pom.xml 文件

maven 的编译 jdk 版本设置:



## 7.新建一个 scala object

## 8.编写 spark 程序

```
object WordCount {
  def main(args: Array[String]): Unit = {
    if(args.length!=2){
      println("cn.edu360.sparkcore.WordCount <input> <output>")
      sys.exit(1)
    }
    val Array(input,output) = args
    /**
     * flatMap map 都是 rdd 上的方法
     * scala 的 api 中, 也有 flatMap map 方法
     * 仅仅是名称一样而已, 一个属于 RDD, 一个属于本地集合
     * 在操作 RDD 的时候, 是不是和本地集合一样的。
     * 使用 spark 来运行程序, 不需要再指定 main 方法了。
     */
    // 配置参数
    val conf = new SparkConf()
    // spark 程序执行的入口 SparkContext
    val sc: SparkContext = new SparkContext(conf)
    // 1, 读取文件
    val data: RDD[String] = sc.textFile(input)
    // 切分
    val lines: RDD[String] = data.flatMap(_.split(" "))
    // 组装
    val wordWithOne: RDD[(String, Int)] = lines.map((_, 1))
    val arr:Array[Int] = Array(1,3,5).map(_*10)
    // 分组聚合
    val key: RDD[(String, Int)] = wordWithOne.reduceByKey(_ + _) // ((a,b)=> a+b)
```

```

// 可选: 排序 倒序排序
val result: RDD[(String, Int)] = key.sortBy(t => -t._2)
//    key.sortBy(t=> t._2,false)
// 写文件 到 hdfs 中
key.saveAsTextFile(output)
// 释放资源
sc.stop()
}
}

```

### 4.3.2. JAVA API 的 wordcount

```

public class JavaWordCount {
    public static void main(String[] args) {
        if(args.length != 2){// 快捷键 sou  psvm
            System.out.println("cn.edu360.sparkcore.JavaWordCount <input> <output>");
            System.exit(1);
        }
        //spark 程序 SparkContext
        SparkConf conf = new SparkConf();
        JavaSparkContext sc = new JavaSparkContext(conf);
        // 获取数据
        JavaRDD<String> data = sc.textFile(args[0]);
        // 切分 输入 类型 ---》 输出类型
        JavaRDD<String> lines = data.flatMap(new FlatMapFunction<String, String>() {
            // 调用每一条数据, 进行处理
            @Override
            public Iterator<String> call(String s) throws Exception {
                // s : hello spark
                // 把数据 String [] 转换成 iterator  Arrays.asList().iterator()
                return Arrays.asList(s.split(" ")).iterator();
            }
        });
        // 组装 hello ---> (hello,1)
        // 3 个参数类型, 输入数据类型 返回值类型 (String, Integer)
        JavaPairRDD<String, Integer> wordwithOne = lines.mapToPair(new
        PairFunction<String, String, Integer>() {
            @Override
            public Tuple2<String, Integer> call(String word) throws Exception {
                return new Tuple2<>(word, 1);
            }
        })
    }
}

```

```
});

// 分组聚合 reduceByKey( +_ ) redeceByKey((a,b)=>a+b) (hello,5)
// 3 个参数类型
JavaPairRDD<String, Integer> resultt = wordwithOne.reduceByKey(new
Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer v1, Integer v2) throws Exception {
        return v1 + v2;
    }
});

// 排序
/** javaAPI 不可以指定排序的规则 sortByKey
 * 先把数据 k-v 互换 , 然后再调用 sortByKey 的方法, 然后再互换回去
 */
JavaPairRDD<Integer, String> swapedResult = resultt.mapToPair(new
PairFunction<Tuple2<String, Integer>, Integer, String>() {
    @Override
    public Tuple2<Integer, String> call(Tuple2<String, Integer> tp) throws Exception {
//        new Tuple2<>(tp._2,tp._1)
// k- v 互换
        return tp.swap();
    }
});

// sortByKey 默认是升序
JavaPairRDD<Integer, String> sortedResult = swapedResult.sortByKey(false);
// 得到使用 javaAPI 计算的 wordcount
JavaPairRDD<String, Integer> finalResult = sortedResult.mapToPair(new
PairFunction<Tuple2<Integer, String>, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(Tuple2<Integer, String> tp) throws Exception {
        return tp.swap();
    }
});

// 写文件
finalResult.saveAsTextFile(args[1]);
// 释放资源
sc.stop();
}
}
```

### 4.3.3. JAVALambda 的 wordcount

```
public class JavaLambdawC {
    public static void main(String[] args) {
        if(args.length !=2){// 快捷键 sou psvm
            System.out.println("cn.edu360.sparkcore.JavaWordCount <input> <output>");
            System.exit(1);
        }
        //spark 程序 SparkContext
        SparkConf conf = new SparkConf();
        JavaSparkContext sc = new JavaSparkContext(conf);
        // 获取数据
        JavaRDD<String> data = sc.textFile(args[0]);
        // 分割并压平
        JavaRDD<String> lines = data.flatMap(t -> Arrays.asList(t.split("
))).iterator());
        // 组装
        JavaPairRDD<String, Integer> wordwithOne = lines.mapToPair(t -> new
        Tuple2<String, Integer>(t, 1));
        // 分组聚合
        JavaPairRDD<String, Integer> result = wordwithOne.reduceByKey((a, b) -> a + b);
        // 排序 先 k- v 互换
        JavaPairRDD<Integer, String> swapedResult = result.mapToPair(t -> t.swap());
        // 再排序
        JavaPairRDD<Integer, String> sortedResult = swapedResult.sortByKey(false);
        JavaPairRDD<String, Integer> finalRes = sortedResult.mapToPair(t -> t.swap());
        // 结果数据写入到hdfs 中
        finalRes.saveAsTextFile(args[1]);
        // 释放资源
        sc.stop();
    }
}
```

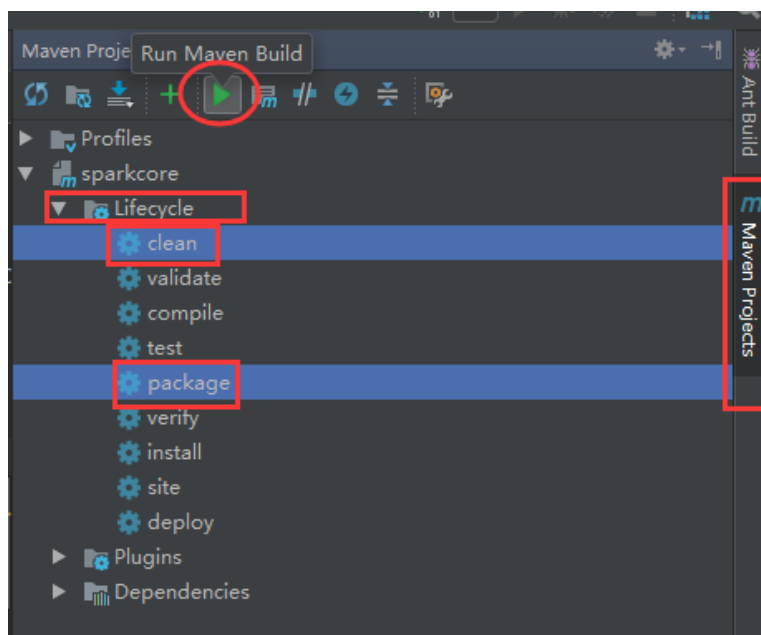
### 4.3.4. local 模式运行 spark 程序

```
// 配置参数
val conf = new SparkConf()
// 设置master 为local 模式 本地模式: local local[*] local[2]
conf.setMaster("local[*]")
conf.setAppName(WordCount.getClass.getSimpleName)
```

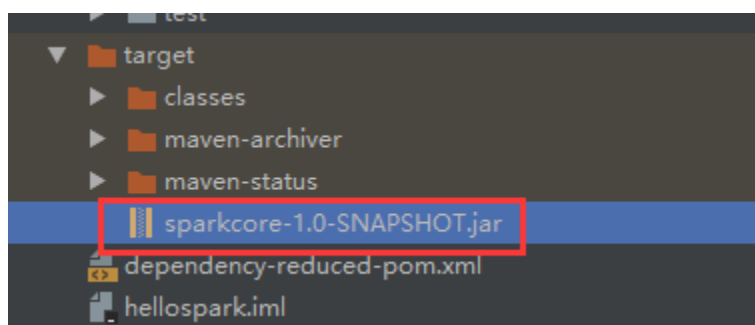
## 4.4. 打包并上传到集群

点击 idea 右侧的 Maven Project 选项

点击 Lifecycle, 选择 clean 和 package, 然后点击 **Run Maven Build**



9. 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 (任意节点即可)



```
[root@hdp-04 ~]# ll
total 16
drwxr-xr-x. 4 root root 4096 Oct 11 16:14 apps
drwxr-xr-x. 3 root root 4096 Oct 7 08:00 dfs
-rw-r--r--. 1 root root 7867 Oct 14 2017 sparkcore-1.0-SNAPSHOT.jar
```

确保启动了 hdfs 集群和 spark 集群

```
# hdfs 启动 (在 namenode 节点上)

# /root/apps/hadoop/sbin/start-dfs.sh

# spark 启动 (在 master 节点上)

# start-all.sh
```

## 4.5. 提交任务

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

```
spark-submit --master spark://hdp-01:7077 --class cn.edu360.spark.WordCount
sparkcore-1.0-SNAPSHOT.jar hdfs://hdp-01:9000/wordcount/input
hdfs://hdp-01:9000/wordcount/output
```

```
[root@hdp-04 ~]# spark-submit --master spark://hdp-01:7077 --executor-memory 1g --total-executor-cores 3 --class cn.edu360.spark.WordCount sparkcore-1.0-SNAPSHOT.jar hdfs://hdp-01:9000/wordcount/input hdfs://hdp-01:9000/wordcount/output
```

可以分多行写:

```
spark-submit \

--class cn.edu360.spark.WordCount \

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

/root/sparkcore-1.0-SNAPSHOT.jar \

hdfs://hdp-01:9000/wordcount/input \

hdfs://hdp-01:9000/wordcount/output
```

任务执行命令的基本套路:

```
# spark-submit 任务提交参数 --class 程序的 main 方法 jar 包 main 的参数列表
```

查看程序执行过程:

在 web 页面查看程序运行状态: <http://hdp-01:8080>

使用 jps 命令查看进程信息

查看 hdfs 文件结果

```
hdfs dfs -cat hdfs://hdp-01:9000/output/part-00000
```

可以直接通过 spark-submit 查看所有的参数配置:

```
root@hdp-01 ~]# 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://...]
Usage: spark-submit run-example [options] example-class [example args]

Options:
  --master MASTER_URL             spark://host:port, mesos://host:port, ya
  --deploy-mode DEPLOY_MODE       whether to launch the driver program loc
```

## 4.6. spark 任务常用参数说明

**--master spark://hdp-01:7077** 指定 Master 的地址

**--executor-memory 2g** 指定**每个** executor 可用内存为 2G ( 512m) 默认是 1024mb

**--total-executor-cores 2** 指定运行任务使用的 cup 核数为 2 个

**--name "appName"** 指定程序运行的名称

**--executor-cores 1** 指定每一个 executor 可用的内存

**--jars xx.jar** 程序额外使用的 jar 包

注意: 如果 worker 节点的内存不足, 那么在启动 spark-shell 的时候, 就不能为 executor

分配超出 worker 可用的内存容量, 大家根据自己 worker 的容量进行分配任务资源。



如果使用配置—`executor-cores`,超过了每个 worker 可以的 `cores`, 任务处于等待状态。

如果使用—`total-executor-cores` ,即使超过可以的 `cores`, 默认使用所有的。以后当集群其他的资源释放之后, 就会被该程序所使用。

如果内存或单个 `executor` 的 `cores` 不足, 启动 `spark-submit` 就会报错, 任务处于等待状态, 不能正常执行。

```
17/10/30 15:32:15 INFO DAGScheduler: Submitting 4 missing tasks from ShuffleMapStage 0 (MapPartitionsRDD[3] at map at WordCount.scala:45) (first 15 tasks are for partitions Vector(0, 1, 2, 3))
17/10/30 15:32:15 INFO TaskSchedulerImpl: Adding task set 0.0 with 4 tasks
17/10/30 15:32:30 WARN TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources
```

#### Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-2018012225842-0011	(kill) wordcount	0	3.0 GB	2018/01/22 22:58:42	root	WAITING	16 s

## 4.7. spark 集群各角色简介

常驻进程: Master 进程 Worker 进程

当我们提交 `spark` 任务的时候 (`spark-shell` ,`spark-submit`)

会生成了一个 `Applications`, 默认会占用所有 `Worker` 的 `cores`, 每一个默认占用了 1g 内存。

可在启动时指定参数。

#### Workers

Worker Id	Address	State	Cores	Memory
worker-20171225193953-192.168.8.14-38285	192.168.8.14:38285	ALIVE	4 (4 Used)	2.7 GB (1024.0 MB Used)
worker-20171227012544-192.168.8.13-43401	192.168.8.13:43401	ALIVE	4 (4 Used)	2.7 GB (1024.0 MB Used)
worker-20171227012607-192.168.8.12-55235	192.168.8.12:55235	ALIVE	4 (4 Used)	2.7 GB (1024.0 MB Used)

#### Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20171225233435-0001	(kill) swc	12	1024.0 MB	2017/12/25 23:34:35	root	RUNNING	0.9 s

通过 `jps` 命令, 可以查看到

在执行 `spark-submit` 的节点上, 有 `spark-submit(dirver)`进程,

```
6353 QuorumPeerMain
9457 SparkSubmit
9560 Jps
5978 NodeManager
6989 DataNode
9134 worker
```

在任务执行的节点上 (worker 节点上), 有 CoarseGrainedExecutorBackend (executor) 进程。

```
1200 NodeManager
4443 CoarseGrainedExecutorBackend
4475 Jps
3933 worker
1631 DataNode
```

然后, 当我们的任务执行完毕之后, 这两个进程都会退出了。

这一个超长的进程通常叫做 Executor, 被 worker 进程启动。真正负责任务的运行。

提交任务的节点, 通常称为 Driver

当执行 spark-shell 时, 会携带参数, 并向 master 发送任务请求, master 在接收到请求之后, 会根据客户端需要的任务资源, 选择出合适的 Worker 节点, 然后向 worker 发送任务指令, 接收到任务之后, worker 会启动一个 executor 进程, executor 启动后, 会等待分配计算任务, 那然后呢, executor 会向 driver 通信, 当有 driver 任务要执行时, 任务就会分发到 executor 上, 然后并行执行。