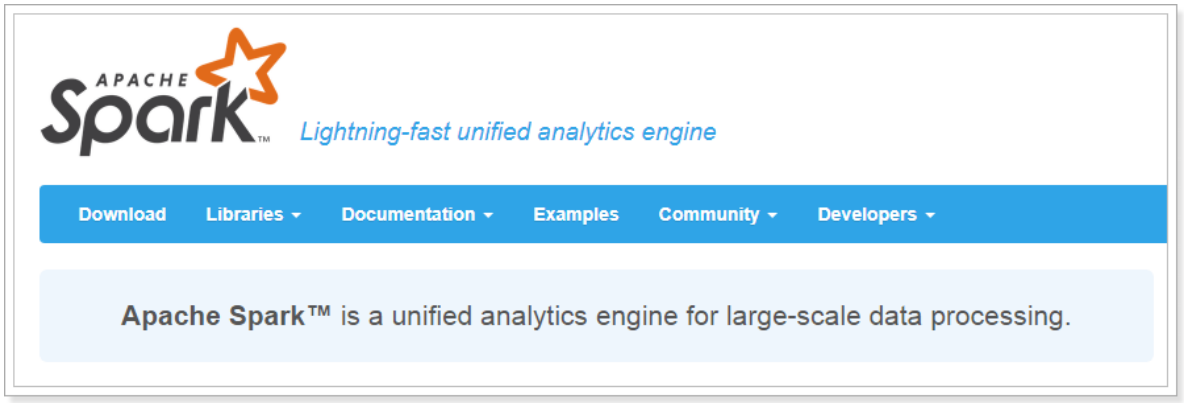
课程介绍

- 了解什么是Spark
- 了解Spark的特点
- 搭建Spark集群
- 了解Spark的角色介绍
- 体验第一个Spark程序
- 编写Spark应用
- 掌握RDD弹性分布式数据集
- 掌握RDD常用的算子操作
- 掌握Spark的任务调度流程

1、Spark概述

1.1、什么是Spark？

官网：http://spark.apache.org/



- 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。
- Spark是UC Berkeley AMP lab (加州大学伯克利分校的AMP实验室) 开发的通用大数据处理框架。
  - A（Algorithms）算法
  - M（Machines）机器
  - P（People）人
  - Spark希望在三者之间进行大规模的集成，并且进行展现运用，将大数据转化为有用的信息。
- Spark 在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。
- 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，逐步形成了大数据处理的一站式解决平台。
- 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。

- Spark 是在 Scala 语言中实现的，它将 Scala 用作其应用程序框架。Spark 和 Scala 能够紧密集成，其中的 Scala 可以像操作本地集合对象一样轻松地操作分布式数据集。

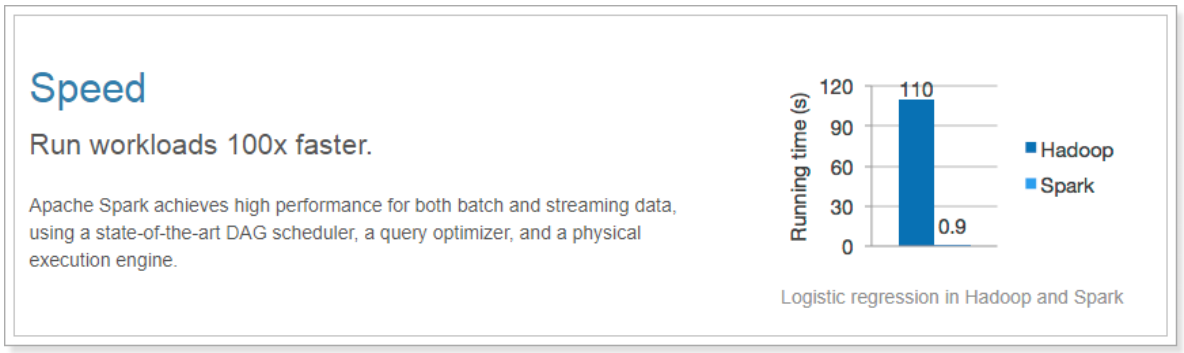
1.2、为什么要学Spark？

- 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MapReduce 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。

1.3、Spark的特点

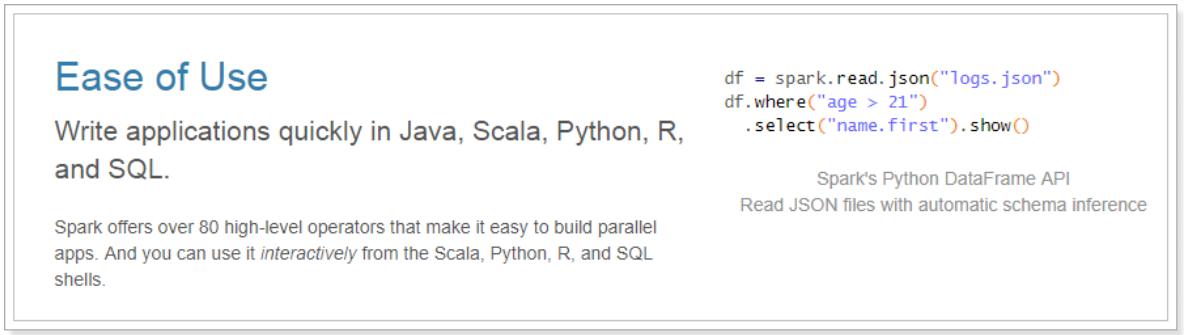
1.3.1、运行速度快

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（有向无环图）执行引擎，可以通过基于内存来高效处理数据流。



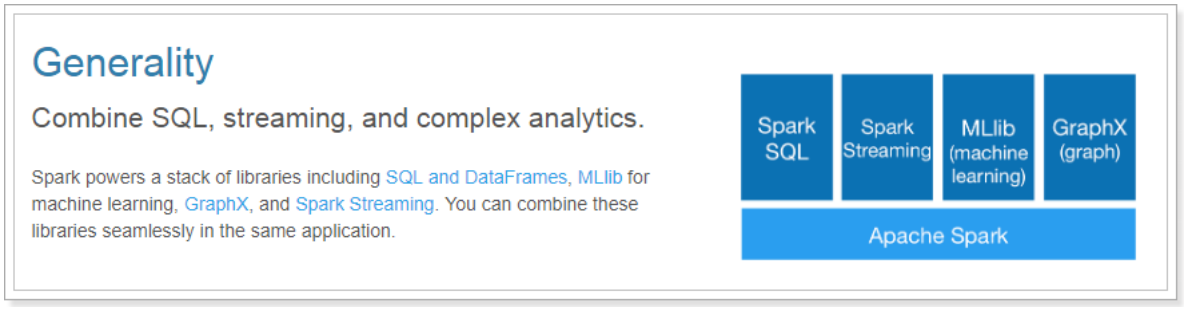
1.3.2、易用性好

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。



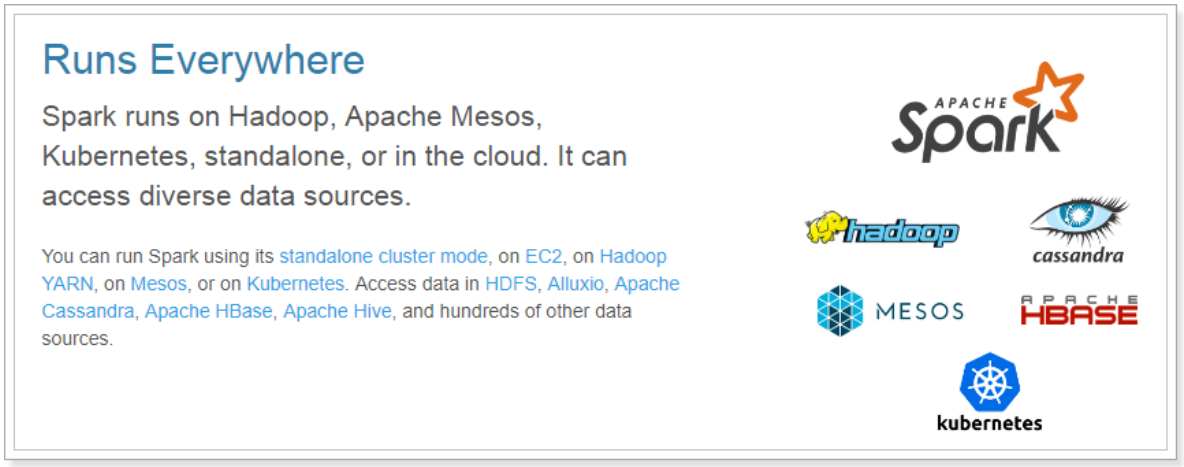
1.3.3、通用性强

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、兼容性强

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。

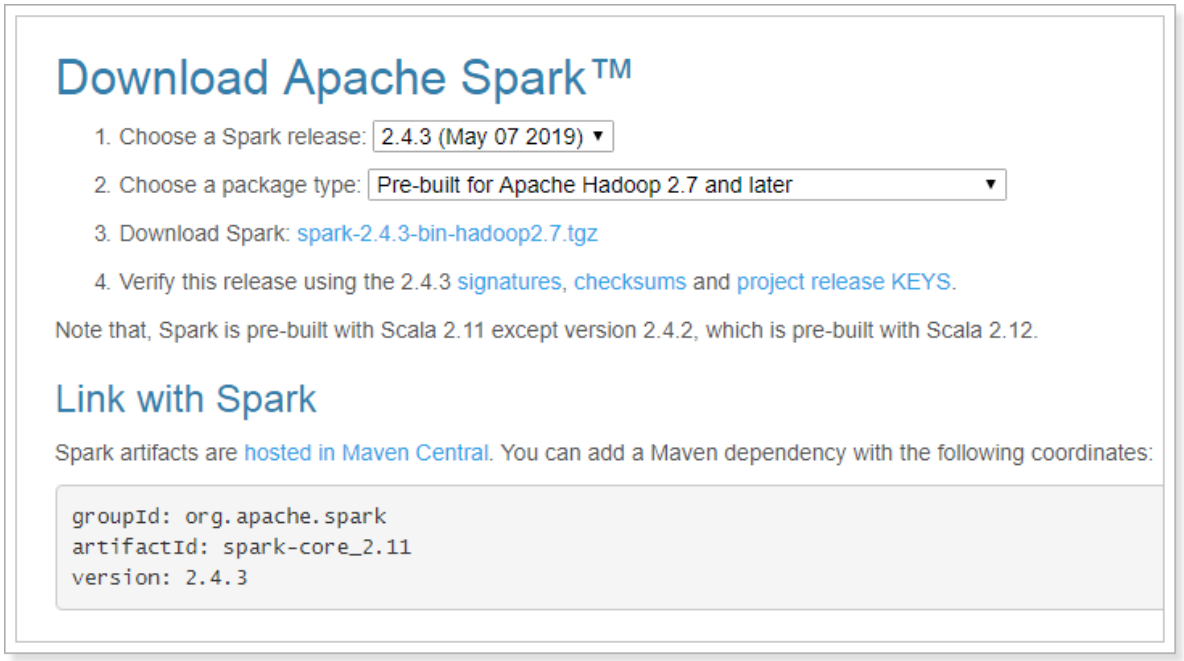


2、搭建Spark集群

Spark的运行环境，可以是在windows上，也可以是运行在linux上，一般情况而言都是运行在linux上的。所以，我们课程也是基于linux来运行的，linux使用的Centos6.5版本。

2.1、下载

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



下载将得到：spark-2.4.3-bin-hadoop2.7.tgz

2.2、环境准备

准备3台linux虚拟机，分别是：

机器地址	节点名称	部署路径
192.168.31.82	node1	/itcast
192.168.31.83	node2	/itcast
192.168.31.84	node3	/itcast

在hosts文件中添加节点的映射：

```
vim /etc/hosts
192.168.31.82 itcast
192.168.31.82 node1 node01
192.168.31.83 node2 node02
192.168.31.84 node3 node03

#关闭防火墙
service iptables stop
#禁止开机启动
chkconfig iptables off
```

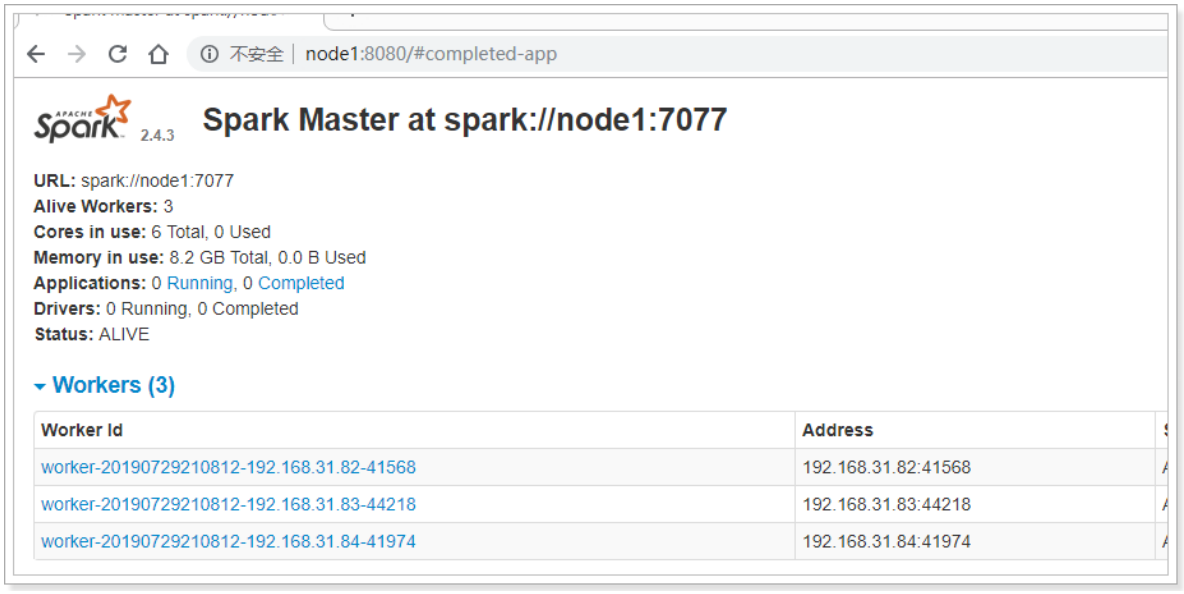
2.3、配置免密登录

```
#生成秘钥，一路回车
ssh-keygen
ssh-copy-id node1
ssh-copy-id node2
ssh-copy-id node3
```

2.4、开始安装

```
#创建安装目录，3台机器都要创建
mkdir /itcast

#上传spark-2.4.3-bin-hadoop2.7.tgz到该目录，进行解压
tar -xvf spark-2.4.3-bin-hadoop2.7.tgz
mv spark-2.4.3-bin-hadoop2.7 spark

#修改配置
cd spark/conf
mv spark-env.sh.template spark-env.sh
vim spark-env.sh
#在最上面插入如下信息
export JAVA_HOME=/usr/local/src/java/jdk1.8.0_141  #jdk地址
export SPARK_MASTER_HOST=node1  #指定master的主机名
export SPARK_MASTER_PORT=7077  #master的端口

mv slaves.template slaves
vim slaves
#输入如下内容
node1
node2
node3

#远程拷贝到其它机器
scp -r spark node2:/itcast/
scp -r spark node3:/itcast/

#添加环境变量（3台都添加）
vim /etc/profile
export SPARK_HOME=/itcast/spark
export PATH=$PATH:$SPARK_HOME/bin
export PATH=$PATH:$SPARK_HOME/sbin

source /etc/profile

#启动
cd /itcast/spark/sbin
./start-all.sh
```

访问webui进行查看：http://node1:8080/



2.5、Spark HA 高可用部署

2.5.1、高可用部署说明

Spark Standalone 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：

- 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。
  - 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。
- 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。

2.5.2、基于zookeeper 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：

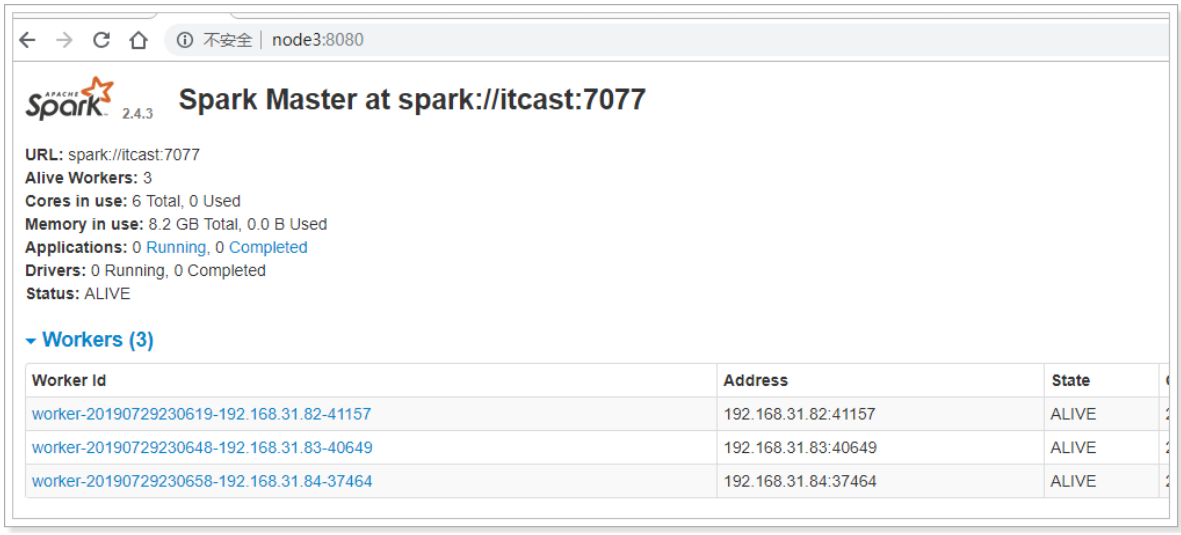
```
#修改hosts文件，增加 192.168.31.81 zk
vim /etc/hosts

#修改spark配置
vim spark-env.sh
export SPARK_MASTER_HOST=node1 #把这个注释掉

#增加ZooKeeper的配置
export SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -
Dspark.deploy.zookeeper.url=zk: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。

启动集群：

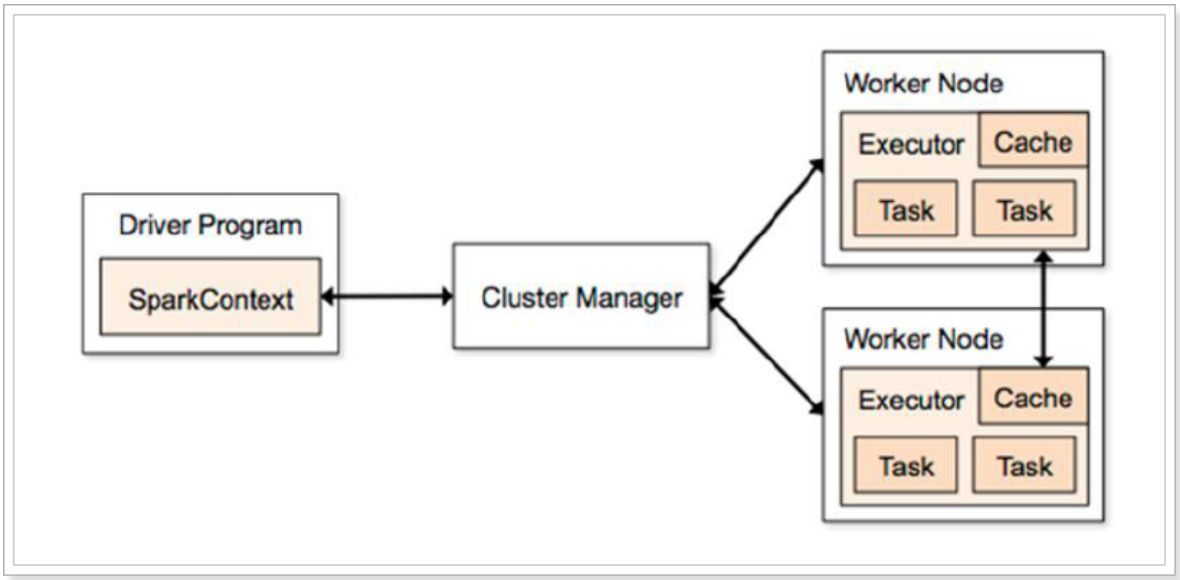
```
#启动HA集群，不能通过start-all.sh的方式启动，会导致找不到master
#首先，将3台机器的master启动起来
./sbin/start-master.sh

#然后，分别启动每个机器上的slave
./start-slave.sh spark://node1:7077  #这里的node1是当前的master，需要通过zk中查看
哪个机器是master

#最后就可以通过停止master的方式进行测试了
```



3、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。

3.1、Spark架构



3.2、架构说明

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。

- Driver Program ：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。
- Cluster Manager：指的是在集群上获取资源的外部服务。目前有三种类型：
  - Standalone: spark 原生的资源管理，由Master 负责资源的分配。
  - Apache Mesos:与hadoop MR 兼容性良好的一种资源调度框架。
  - Hadoop Yarn: 主要是指Yarn 中的ResourceManager。
- Worker Node：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。
- 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。

4、体验 Spark 程序

4.1、执行第一个spark 程序

4.1.1、普通模式提交任务

```
#进入到bin目录
./run-example SparkPi 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。
```

```
19/04/15 08:29:31 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
19/04/15 08:29:31 INFO DAGScheduler: ResultStage 0 (reduce at SparkPi.scala:38) finished in 0.526 s
19/04/15 08:29:31 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38, took 0.574569 s
Pi is roughly 3.142895142895143
19/04/15 08:29:31 INFO SparkUI: Stopped Spark web UI at http://itcast:4040
19/04/15 08:29:31 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
19/04/15 08:29:31 INFO MemoryStore: MemoryStore cleared
19/04/15 08:29:31 INFO BlockManager: BlockManager stopped
```

4.1.2、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。

```
#进入到bin目录
./run-example --master spark://node1:7077,node2:7077,node3:7077 SparkPi 10
```

测试：将node1的masterkill掉，发现还是可以正常执行的。

4.2、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。

> **需求：读取本地文件，实现文件内的单词计数。**

```
vim /itcast/word.txt
#输入以下内容
hello world
hello spring
hello mvc
hello spark
```

```
./spark-shell

#输入如下命令：
sc.textFile("file:///itcast/word.txt").flatMap(_.split("
")).map((_,1)).reduceByKey(_+_).collect

#结果：
res2: Array[(String, Int)] = Array((hello,4), (mvc,1), (world,1), (spark,1),
(spring,1))
```

代码解释：

- **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，file:// 读取本地文件
- **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。

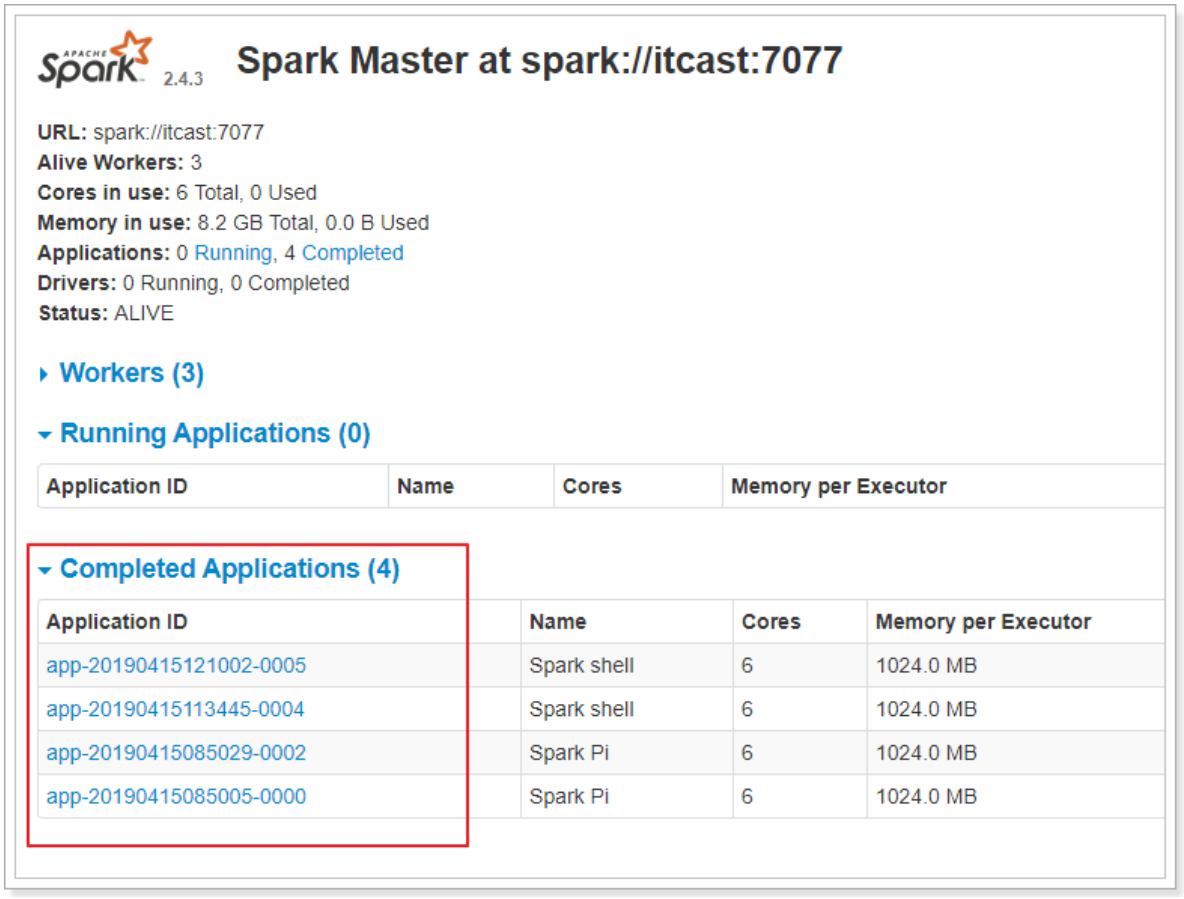
4.3、集群运行

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。

```
#指定集群master
./spark-shell --master spark://node2:7077

#计算完成后，将结果写入到本地磁盘中
```

```
sc.textFile("file:///itcast/word.txt").flatMap(_.split("
")).map((_,1)).reduceByKey(_+_).saveAsTextFile("file:///itcast/wc")

#wc目录
-rw-r--r--. 1 root root 28 4月  15 12:13 part-00000
-rw-r--r--. 1 root root 21 4月  15 12:13 part-00001
-rw-r--r--. 1 root root  0 4月  15 12:13 _SUCCESS
[root@itcast wc]# cat part-00000 part-00001
(hello,4)
(mvc,1)
(world,1)
(spark,1)
(spring,1)
```

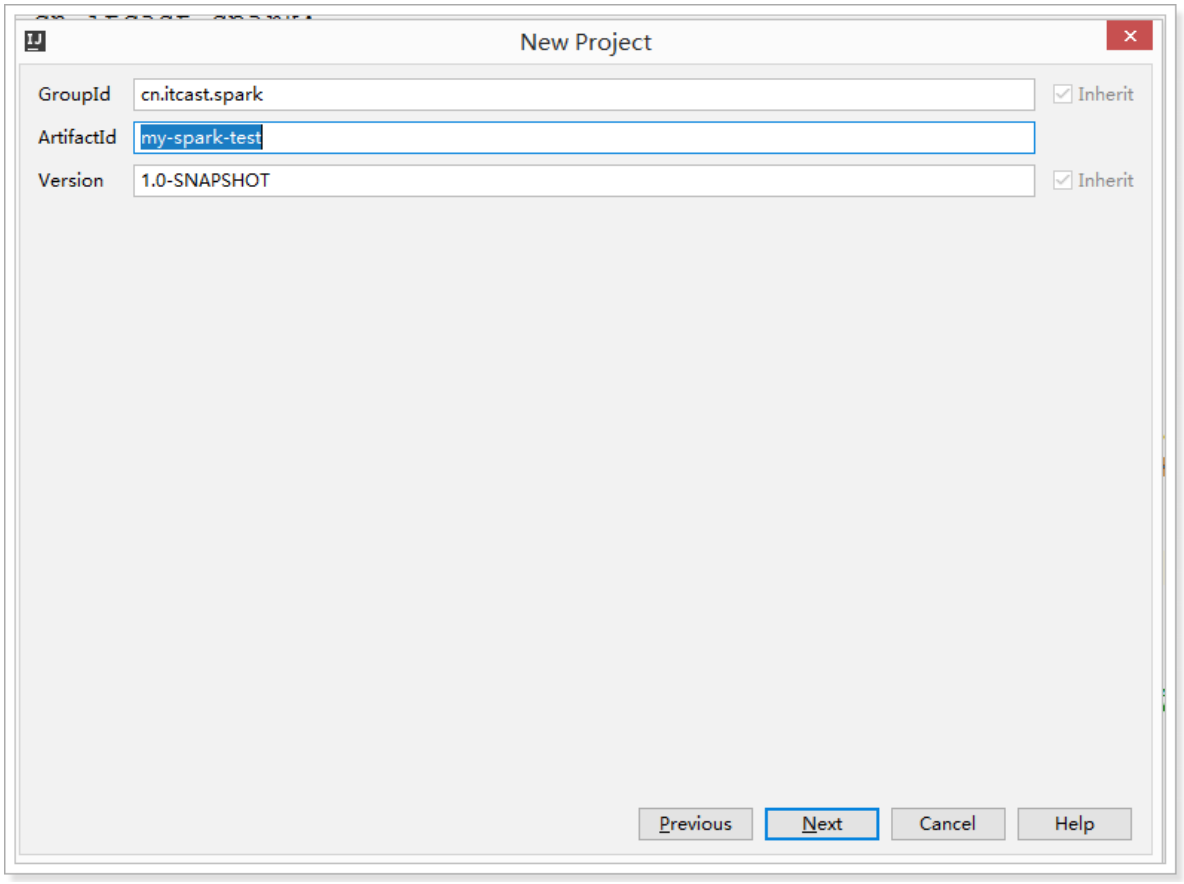
可以看到完成的应用：



5、编写Spark应用

> 说明：编写Spark应用，官方推荐使用的是Scala语言，由于本课程中不涉及到Scala语言的语法讲解，所以，我们将使用java语言进行编写应用，如熟悉Scala语言的同学请使用Scala编写。

5.1、创建工程



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>cn.itcast.spark</groupId>
    <artifactId>my-spark-test</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.12</artifactId>
            <version>2.4.3</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- java编译插件 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
```

```
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

5.2、编写WordCount程序

实现：

```
package cn.itcast.spark;

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.Int;
import scala.Tuple2;

import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public class WordCountApp {

    public static void main(String[] args) {

        // spark的配置
        SparkConf sparkConf = new SparkConf()
                .setAppName("WordCountApp")
                .setMaster("local[*]"); //本地模式，并且使用和cpu的内核数相同的线程数进行执行

        // 定义上下文对象,它是程序的入口
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        // 读取文件
        JavaRDD<String> fileRdd = jsc.textFile("F://code//word.txt");

        // 压扁操作，并且按照空格分割
        JavaRDD<String> flatMapRdd = fileRdd.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public Iterator<String> call(String s) throws Exception {
                String[] ss = s.split(" ");
                return Arrays.asList(ss).iterator();
            }
        });

        // 对单词做计数
        JavaPairRDD<Object, Integer> mapToPairRdd = flatMapRdd.mapToPair(new PairFunction<String, Object, Integer>() {
            @Override
```

```
            public Tuple2<Object, Integer> call(String s) throws Exception
{
                return new Tuple2<>(s, 1);
            }
        });

        // 对相同的单词做相加操作
        JavaPairRDD<Object, Integer> reduceByKeyRdd =
mapToPairRdd.reduceByKey(new Function2<Integer, Integer, Integer>() {
            @Override
            public Integer call(Integer v1, Integer v2) throws Exception {
                return v1 + v2;
            }
        });

        // 执行计算
        List<Tuple2<Object, Integer>> collect = reduceByKeyRdd.collect();
        for (Tuple2<Object, Integer> obj : collect) {
            System.out.println(obj._1() + "出现的次数为："+ obj._2());
        }


        // 关闭，释放资源
        jsc.stop();
    }
}

```

优化之后：

```
package cn.itcast.spark;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import scala.Tuple2;

import java.util.Arrays;
import java.util.List;

public class WordCountApp {

    public static void main(String[] args) {
        //设置spark 的配置文件信息
        SparkConf sparkConf = new SparkConf()
                .setAppName("WordCountApp")
                .setMaster("local[*]");
        //构建sparkcontext 上下文对象，它是程序的入口,所有计算的源头
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        //读取文件
        JavaRDD<String> linesJavaRDD = jsc.textFile("F://code//word.txt");

        //对文件中每一行单词进行压平切分,并且按照空格切分，对相同单词做汇总
        List<Tuple2<String, Integer>> list = linesJavaRDD.flatMap(s -> {

```

```
            String[] words = s.split(" ");
            return Arrays.asList(words).iterator();
        }).mapToPair(s -> new Tuple2<String, Integer>(s, 1))
                .reduceByKey((v1, v2) -> v1 + v2)
                .collect();

        // 循环打印单词出现的次数
        list.forEach(tuple2 -> System.out.println(tuple2._1() + "出现次数: "
+ tuple2._2()));

        // 停止应用
        jsc.stop();

    }

}

```

5.3、打包在集群中运行

5.3.1、修改pom文件

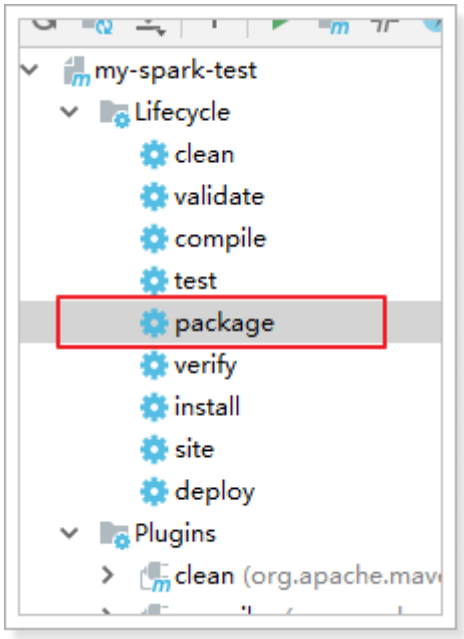
```
   <dependencies>
       <dependency>
           <groupId>org.apache.spark</groupId>
           <artifactId>spark-core_2.12</artifactId>
           <version>2.4.3</version>
           <scope>provided</scope>
       </dependency>
   </dependencies>
```

5.3.2、修改代码

```
package cn.itcast.spark;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import scala.Tuple2;

import java.util.Arrays;

public class WordCountApp {

    public static void main(String[] args) {
        SparkConf sparkConf = new SparkConf()
                .setAppName("WordCountApp");

        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        // 文件从参数中获取
        JavaRDD<String> linesJavaRDD = jsc.textFile(args[0]);

        // 输出路径也是从参数中获取
        linesJavaRDD.flatMap(s -> {
            String[] words = s.split(" ");
```

```
            return Arrays.asList(words).iterator();
        }).mapToPair(s -> new Tuple2<String, Integer>(s, 1))
                .reduceByKey((v1, v2) -> v1 + v2).saveAsTextFile(args[1]);

        jsc.stop();

    }

}

```

5.3.3、maven打包

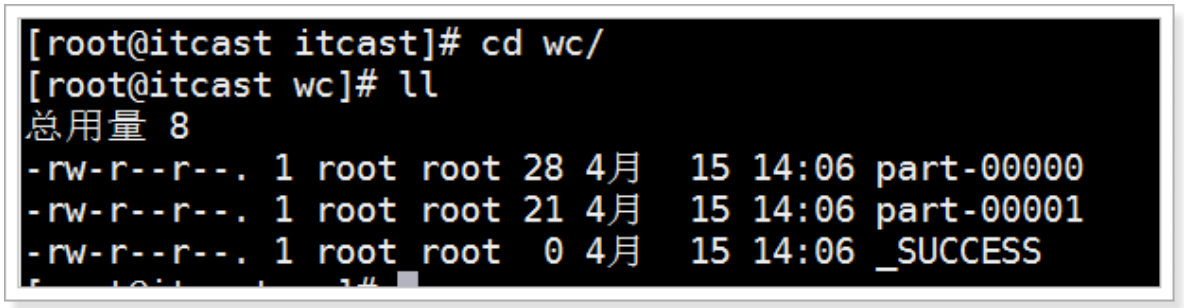


打包完成后会得到my-spark-test-1.0-SNAPSHOT.jar文件，将该文件上传到/itcast目录下。

5.3.4、提交任务到集群

```
./spark-submit  --master spark://node2:7077  --class
cn.itcast.spark.WordCountApp /itcast/my-spark-test-1.0-SNAPSHOT.jar
/itcast/word.txt /itcast/wc
```

执行结果：



6、弹性分布式数据集RDD

6.1、RDD概述

6.1.1、什么是RDD

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是Spark中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将数据缓存在内存中，后续的查询能够重用这些数据，这极大地提升了查询速度。

- Dataset：一个数据集合，用于存放数据的。
- Distributed：RDD中的数据是分布式存储的，可用于分布式计算。
- Resilient：RDD中的数据可以存储在内存中或者磁盘中。

6.1.2、为什么会产生RDD？

- 传统的MapReduce虽然具有自动容错、平衡负载和可拓展性的优点，但是其最大缺点是采用非循环式的数据流模型，使得在迭代计算中要进行大量的磁盘IO操作。RDD正是解决这一缺点的抽象方法。
- RDD是Spark提供的最重要的抽象的概念，它是一种具有容错机制的特殊集合，可以分布在集群的节点上，以函数式编程来操作集合，进行各种并行操作。可以把RDD的结果数据进行缓存，方便进行多次重用，避免重复计算。

6.2、RDD的类型

Spark中的操作大致可以分为四类操作，分别是创建操作、转换操作、控制操作和行为操作。

- 创建操作：用于RDD创建工作，RDD的创建只有两种方法：
  - 一种是来自于内存集合和外部存储系统
  - 通过转换操作生成的RDD
- 转换操作：将RDD通过一定的操作变换成新的RDD
  - 比如：RDD通过flatMap操作后得到新的RDD
- 控制操作：进行RDD持久化，可以让RDD按不同的存储策略保存在磁盘或者内存中。
- 行为操作：能够触发Spark的运行操作，比如：collect的操作就是行为操作。

6.3、RDD常用的算子操作

这里介绍一些常用的算子操作，更多的参数资料中的《SparkRDD函数详解.doc》

6.3.1、map

map是对RDD中的每个元素都执行一个指定的函数来产生一个新的RDD。 任何原RDD中的元素在新RDD中都有且只有一个元素与之对应。

```
1  package cn.itcast.spark;
2
3  import org.apache.spark.SparkConf;
4  import org.apache.spark.api.java.JavaRDD;
5  import org.apache.spark.api.java.JavaSparkContext;
6  import org.apache.spark.api.java.function.Function;
7  import org.junit.Test;
8
9  import java.util.Arrays;
10 import java.util.List;
11
12 public class RddOperation {
13
14     @Test
15     public void testMap() {
16         SparkConf sparkConf = new SparkConf()
```

```
                .setAppName("RddOperation")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // 生成RDD
        JavaRDD<Integer> rdd = jsc.parallelize(Arrays.asList(1, 2, 3, 4,
5));
        // map操作
        JavaRDD<Object> rdd2 = rdd.map(v1 -> v1 * 2);

        //收集数据
        List<Object> list = rdd2.collect();
        for (Object o : list) {
            System.out.println(o);
        }
    }

}
```

6.3.2、filter

filter 是对RDD中的每个元素都执行一个指定的函数来过滤产生一个新的RDD。 任何原RDD中的元素在新RDD中都有且只有一个元素与之对应。

```
@Test
    public void testFilter() {
        SparkConf sparkConf = new SparkConf()
                .setAppName("RddOperation")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // 生成RDD
        JavaRDD<Integer> rdd = jsc.parallelize(Arrays.asList(1, 2, 3, 4,
5));
        // 过滤操作，只保留大于3的数
        JavaRDD<Integer> rdd2 = rdd.filter(v1 -> v1 >3);

        //收集数据
        List<Integer> list = rdd2.collect();
        for (Integer o : list) {
            System.out.println(o);
        }
    }
```

6.3.3、flatMap

与map类似，区别是原RDD中的元素经map处理后只能生成一个元素，而原RDD中的元素经flatmap处理后可生成多个元素来构建新RDD。 举例：对原RDD中的每个元素x产生y个元素（从1到y，y为元素x的值）

```
    @Test
    public void testFlatMap() {
        SparkConf sparkConf = new SparkConf()
                .setAppName("RddOperation")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // 生成RDD
```

```
        JavaRDD<String> rdd = jsc.parallelize(Arrays.asList("hello world","hello spark"));
        // 数据压扁操作
        JavaRDD<String> rdd2 = rdd.flatMap(s -> Arrays.asList(s.split(" ")).iterator());

        //收集数据
        List<String> list = rdd2.collect();
        for (String o : list) {
            System.out.println(o);
        }
    }
```

6.3.4、mapPartitions

mapPartitions是map的一个变种。map的输入函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区，也就是把每个分区中的内容作为整体来处理的。

```
    @Test
    public void testMapPartitions (){
        SparkConf sparkConf = new SparkConf()
                .setAppName("RddOperation")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // 生成RDD，分3个区存储
        JavaRDD<Integer> rdd = jsc.parallelize(Arrays.asList(1,2,3,4,5,6,7,8,9,10),3);
        // 分区操作数据
        JavaRDD<Integer> rdd2 = rdd.mapPartitions(integerIterator -> {
            List<Integer> result = new ArrayList<>();
            while (integerIterator.hasNext()){
                Integer i = integerIterator.next();
                result.add(i);
            }
            System.out.println("分区数据：" + result);
            return result.iterator();
        });

        //收集数据
        List<Integer> list = rdd2.collect();
        list.forEach(integer ->  System.out.println(integer));
    }
```

6.3.5、mapToPair

此函数会对一个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

```
@Test
    public void testMapToPair (){
        SparkConf sparkConf = new SparkConf()
                .setAppName("RddOperation")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // 生成RDD
```

```
        JavaRDD<String> rdd = jsc.parallelize(Arrays.asList("hello world","hello spark"));

        // 按照空格分割
        JavaRDD<String> rdd1 = rdd.flatMap(s -> Arrays.asList(s.split(" ")).iterator());

        // 把每一个单词的出现数量标记为1
        JavaPairRDD<Object, Object> rdd2 = rdd1.mapToPair(s -> new Tuple2<>(s, 1));

        //收集数据
        List<Tuple2<Object, Object>> list = rdd2.collect();
        for (Tuple2<Object, Object> o : list) {
            System.out.println(o);
        }
    }
```

6.3.6、reduceByKey

顾名思义，reduceByKey就是对元素为KV对的RDD中Key相同的元素的Value进行reduce，因此，Key相同的多个元素的值被reduce为一个值，然后与原RDD中的Key组成一个新的KV对。

```
    @Test
    public void testReduceByKey (){
        SparkConf sparkConf = new SparkConf()
                .setAppName("RddOperation")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // 生成RDD
        JavaRDD<String> rdd = jsc.parallelize(Arrays.asList("hello world","hello spark"));

        // 按照空格分割
        JavaRDD<String> rdd1 = rdd.flatMap(s -> Arrays.asList(s.split(" ")).iterator());

        // 把每一个单词的出现数量标记为1
        JavaPairRDD<Object, Integer> rdd2 = rdd1.mapToPair(s -> new Tuple2<>(s, 1));

        // 按照key相同的数进行相加操作
        JavaPairRDD<Object, Integer> rdd3 = rdd2.reduceByKey((v1, v2) -> v1 + v2);

        //收集数据
        List<Tuple2<Object, Integer>> list = rdd3.collect();
        for (Tuple2<Object, Integer> o : list) {
            System.out.println(o);
        }
    }
```

6.3.7、coalesce

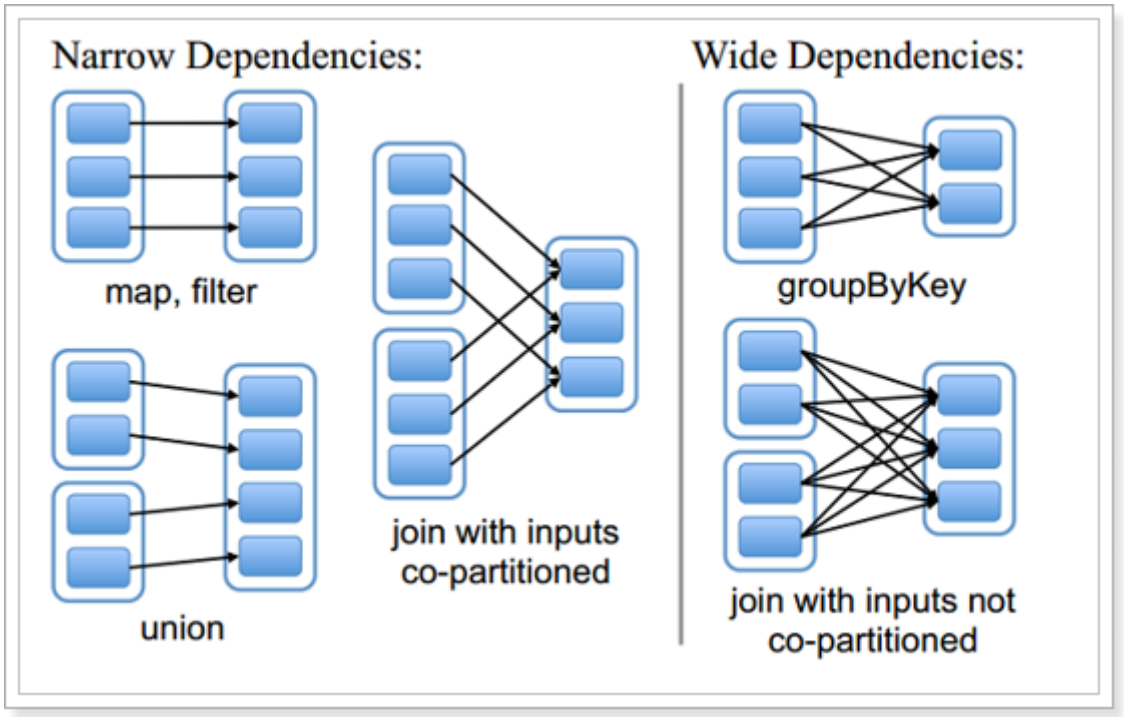
该函数用于将RDD进行重分区，使用HashPartitioner。第一个参数为重分区的数目，第二个为是否进行shuffle，默认为false;

```
    @Test
    public void testCoalesce (){
        SparkConf sparkConf = new SparkConf()
                .setAppName("RddOperation")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        // 生成RDD
        JavaRDD<Integer> rdd =
jsc.parallelize(Arrays.asList(1,2,3,4,5,6,7,8,9,10));

        System.out.println(rdd.getNumPartitions()); //4
        // 重新分区，shuffle：是否重新分配存储，如果分区数大于原有分区数，就需要设置为
true
        JavaRDD<Integer> rdd2 = rdd.coalesce(2, false);
        System.out.println(rdd2.getNumPartitions()); //2

        //收集数据
        List<Integer> list = rdd2.collect();
        for (Integer o : list) {
            System.out.println(o);
        }
    }
```

6.4、Spark任务调度

6.4.1、RDD的依赖关系

RDD和它依赖的父RDD的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。



- 窄依赖
  - 窄依赖指的是每一个父RDD的Partition最多被子RDD的一个Partition使用。
  - **总结：窄依赖我们形象的比喻为独生子女**
- 宽依赖
  - 宽依赖指的是多个子RDD的Partition会依赖同一个父RDD的Partition

    - 总结：宽依赖我们形象的比喻为超生
- Lineage（血统）
    - RDD只支持粗粒度转换，即只记录单个块上执行的单个操作。
    - 将创建RDD的一系列Lineage（即血统）记录下来，以便恢复丢失的分区。RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

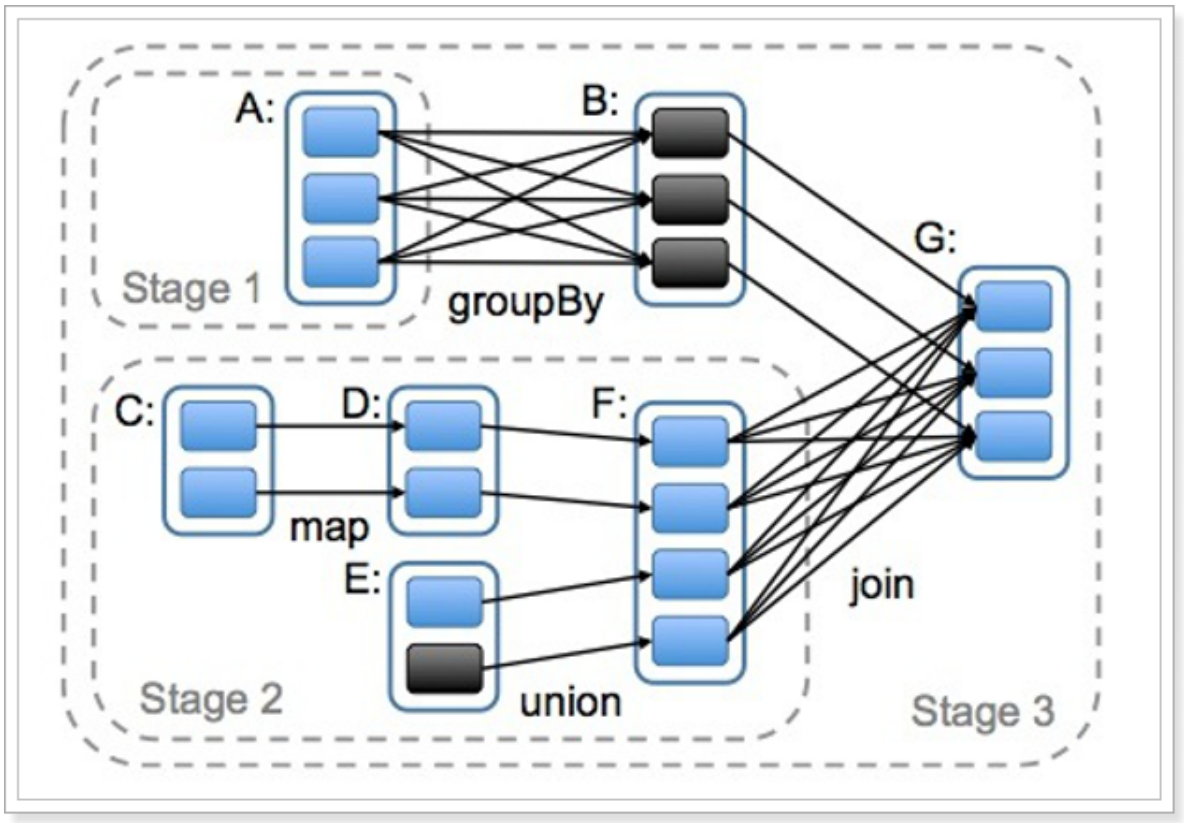
6.4.2、DAG

DAG(Directed Acyclic Graph)叫做有向无环图，原始的RDD通过一系列的转换就形成了DAG，根据RDD之间依赖关系的不同将DAG划分成不同的Stage(调度阶段)。

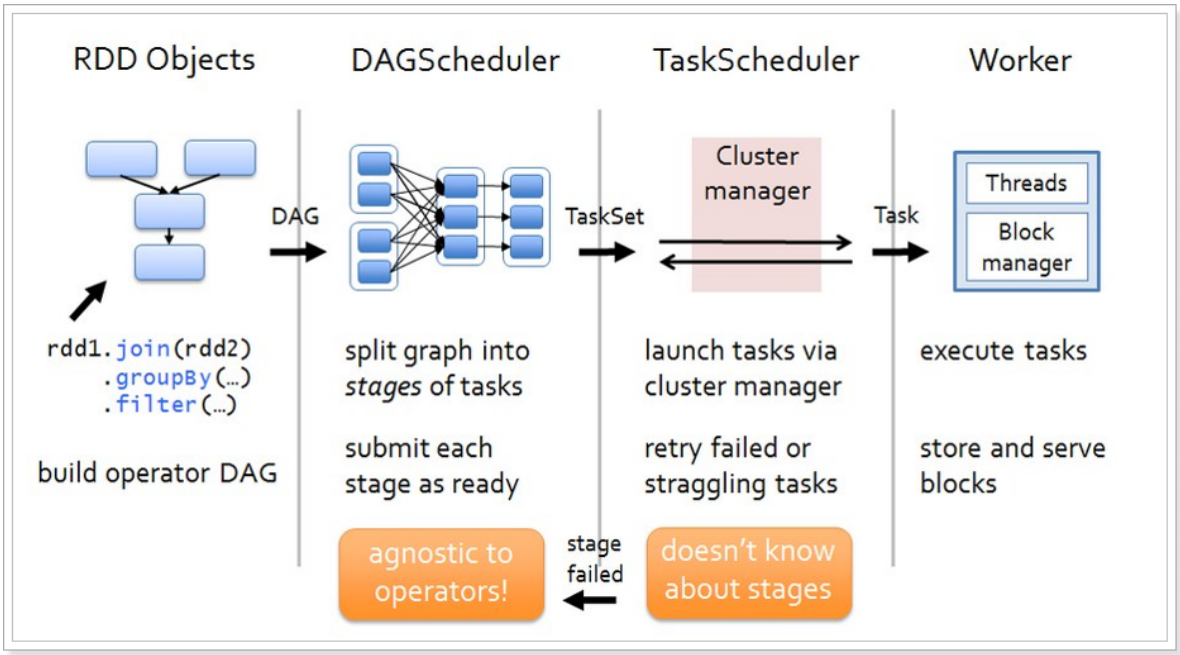
对于窄依赖，partition的转换处理在一个Stage中完成计算。

对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算。

因此宽依赖是划分Stage的依据。



6.4.3、 任务调度流程



说明：

- 各个RDD之间存在着依赖关系，这些依赖关系就形成有向无环图DAG；
- DAGScheduler对这些依赖关系形成的DAG进行Stage划分，划分的规则很简单，从后往前回溯，遇到窄依赖加入本stage，遇见宽依赖进行Stage切分，完成了Stage的划分。
- DAGScheduler基于每个Stage生成TaskSet,并将TaskSet提交给TaskScheduler。
- TaskScheduler 负责具体的task调度,最后在Worker节点上启动task。