# .day02 [Spark核心编程,RDD操作,Spark算子操作]

## 今日反馈

| 1、idea的界面还是调成白色的把，黑色看着不清楚。 2、老师发的虚拟机，哇，难受，还好都调好了。 |

| ------------------------------------------------------------ |

| 老师讲的很棒,喜欢这种研究官网的学习风格 |

| 新老师，非常有干劲，讲的也非常好，逻辑思路清晰，弥补了自己看的时候遗漏的很多细节。 |

| 老师。第二台手动masterq启动报错，Cannot assign requested address: Service 'sparkMaster' failed after 16 retries (starting from 7077)! Consider explicitly setting the appropriate port for the service 'sparkMaster' (for example spark.ui.port for SparkUI) to an available port or increasing spark.port.maxRetries.SPARK\_MASTER\_PORT |

| 老师带我们看官方文档敲代码 真的很棒 点个赞 老师上课手敲的笔记能不能 省略点 你提供的完整笔记都有对应的内容 有些在手敲的可以省略 把更多的时间 放在说明上就好 那个手敲 看着我很着急啊 |

| 老师上课不需要手敲太多笔记，我们需要的仅仅是一份非常详细规整的笔记。不用上课现敲，有点点浪费时间。 老师带我们看官方文档的授课方式很不错，授人以鱼不如授人以渔，点个赞！ |

| 1、最喜欢的老师授课有两种 一种是看官网 一种是看源码 这种讲解很有说服力也很直观 现在老师能带我们从官网学习真的很好 支持一下 2、但是有的讲解不需要做笔记太浪费时间了 只要把注意点记录就行 原有的笔记已经很清楚课下自己看就行没必要做两遍 3、还有sparkTask在spark环境执行和在yarn上执行有什么区别？资源调度有什么不同？ |

| 在master的高可用配置时，node01与node02都开启master时，node01位active，关闭后node02位active，再次启动node01的master时，谁是active，有什么规则吗，我的结果跟老师上课演示的不同。 |

| jackeizhu，我的高可用的alive状态的master进程关闭后在启用，状态是standby，印象中你的变成了alive状态，请问这里面有什么机制吗？老师讲的挺好的，继续加油！！！ |

| 老师讲的非常好,有条理,笔记很认真,真是赞! |

| 老师讲的很好 |

| 预习资料发下 |

| --master local 和 deploy mode client是不是一个意思除了规定线程数 |

| 上课时候主次分明一点就更好了，有些不是很重要东西不必要上课还敲给我们看 |

| 有一个问题：node01已经./\*\*in/start-all.sh ；此时node02中 ./\*\*in/start-master.sh 没法启动 ，得把SPARK\_MASTER\_HOST 中改成node02 ,才能启动；我看了视频，老师你没改怎么可以直接启动的啊？ |

## 今日内容

- 掌握RDD的原理

- Spark核心编程创建RDD

- Spark核心编程算子操作

- Spark核心编程实战

- Spark核心编程\_RDD持久化详解

- Spark核心编程\_RDD的宽窄依赖

## 教学目标

- 理解wordcount程序原理

- 掌握Spark核心编程创建RDD

- 掌握Spark核心编程的算子操作

- 熟练使用算子进行核心编程

- 掌握RDD的持久化机制

- 掌握Spark核心编程\_RDD的宽窄依赖

- 掌握stage的划分

# # 第一章 弹性分布式数据集RDD

## ## 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。

基于内存

弹性的

自动容错的

内存迭代

## ## 1.2 RDD的属性

```scala

/\*\* Internally, each RDD is characterized by five main properties:

\*

\* - A list of partitions

\* - A function for computing each split

\* - A list of dependencies on other RDDs

\* - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)

\* - Optionally, a list of preferred locations to compute each split on (e.g. block locations for

\* an HDFS file)

\*/

```

1) A list of partitions :一个分区（Partition）列表，数据集的基本组成单位。

对于RDD来说，每个分区都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分区个数，如果没有指定，那么就会采用默认值。（比如：读取HDFS上数据文件产生的RDD分区数跟block的个数相等）

2) A function for computing each split ：一个计算每个分区的函数。

Spark中RDD的计算是以分区为单位的，每个RDD都会实现compute函数以达到这个目的。

3) A list of dependencies on other RDDs：一个RDD会依赖于其他多个RDD，RDD之间的依赖关系。

RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

4) Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)：一个Partitioner，即RDD的分区函数（可选项）。

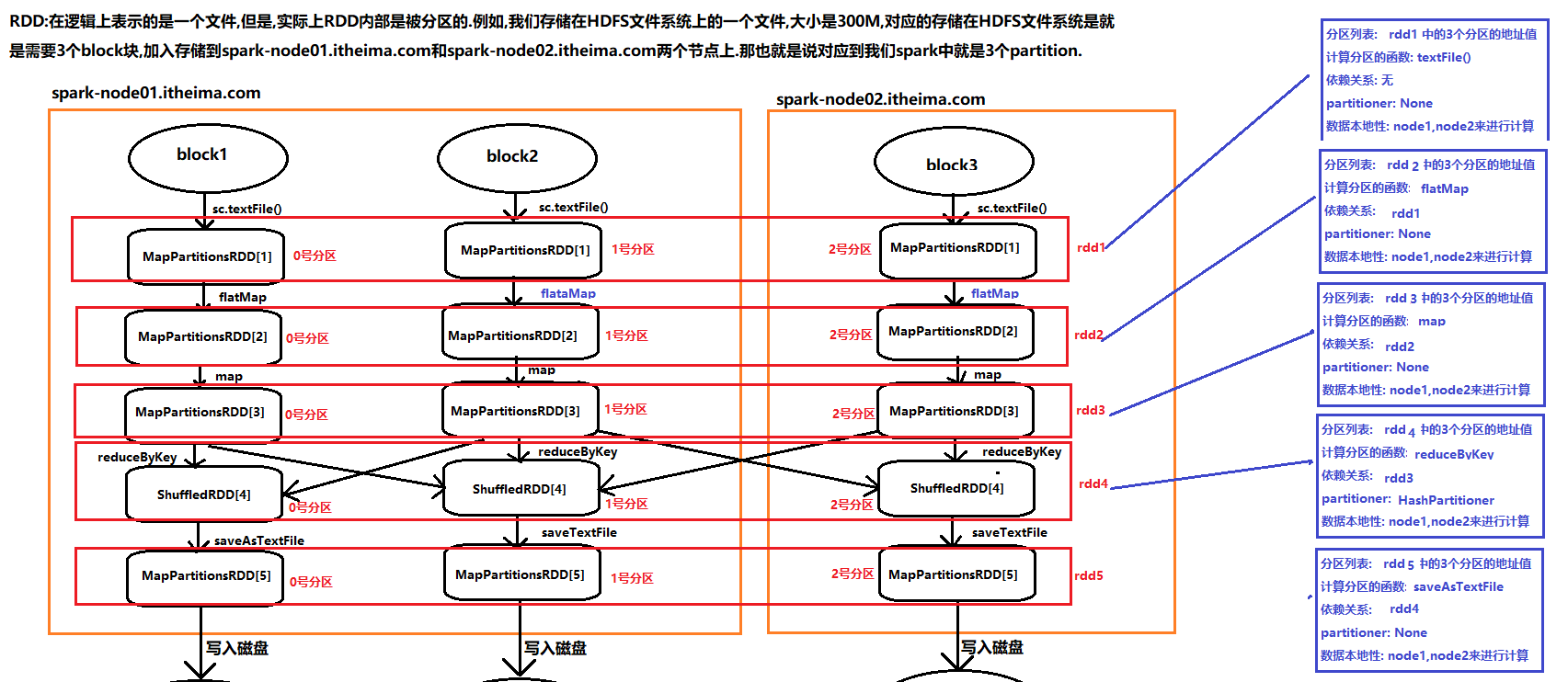
当前Spark中实现了两种类型的分区函数，一个是基于哈希的HashPartitioner，另外一 个是基于范围的RangePartitioner。只有对于key-value的RDD，才会有Partitioner，非key-value的RDD的Parititioner的值是None。Partitioner函数决定了parent RDD Shuffle输出时的分区数量。

5) Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)：一个列表，存储每个Partition的优先位置(可选项)。

对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置（spark进行任务分配的时候尽可能选择那些存有数据的worker节点来进行任务计算）。

## ## 1.3 分布式弹性数据集图解

![](img\RDD原理图解.png)



# # 第二章 Spark核心编程\_创建RDD

## ## 2.1 创建RDD的方式

在学习Spark核心编程的时候,我们需要做的第一件事情就是创建一个初始的RDD,这个RDD通常就代表和包含了Spark应用程序的输入源数据.然后在创建了初始的RDD之后,才可以通过Spark Core提供的transformation算子,对该RDD进行转换,来获取其他的RDD.Spark Core提供了三种创建RDD的方式,主要包括:

1) 使用程序中的集合创建RDD,该种方式主要是用来测试编写的Spark应用程序是否正确

2) 使用本地文件创建RDD,主要用于临时性地处理一些存储了大量数据的文件.

3) 使用HDFS文件创建RDD,此种方式在生产环境中是最常用的处理方式.

## ## 2.2 并行化集合创建RDD

### ### 2.2.1 使用Java语言完成并行化集合创建RDD

```java

public class ParallelizeConnection {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("ParallelizeConnection")

.setMaster("local[2]");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:要通过并行化集合的方式创建RDD,那么就调用SparkContext以及其子类的parallelize()方法

List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

JavaRDD<Integer> numberRDD = sc.parallelize(numbers);

//4:执行reduce算子操作,相当于先进行1+2=3,然后再用3+3=6,然后再用6 + 4 = 10......以此类推

Integer sum = numberRDD.reduce(new Function2<Integer, Integer, Integer>() {

@Override

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

return v1+v2;

}

});

//输出累加的和

System.out.println("1~10之间的累加和: "+sum);

//5:关闭JavaSparkContext

sc.close();

}

}

```

主要是调用parallelize方法,将一个已知的集合转换成RDD

### ### 2.2.2 使用Scala语言完成并行化集合创建RDD

```scala

object ParallelizeConnectionScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("ParallelizeConnection")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:创建一个已经存在的集合

val numbers = Array(1,2,3,4,5,6,7,8,9,10)

//4:调用parallelize方法完成并行化创建RDD

val numberRDD: RDD[Int] = sc.parallelize(numbers)

//5:进行累加求和操作

val sum: Int = numberRDD.reduce((a, b)=>a+b)

//输出累加的和

println("1~10之间的累加和: "+sum)

//关闭sc

sc.stop()

}

}

```

## ## 2.3 使用本地文件和HDFS创建RDD

Spark是支持使用任何Hadoop支持的存储系统上的文件创建RDD的,比如说HDFS,Cassandra,Hbase以及本地文件.通过调用SparkContext的textFile()方法,可以针对本地文件或HDFS文件创建RDD

注意事项:

1) 如果是针对本地文件的话,如果是在windows上本地测试,windows上有一份文件即可,如果是在spark集群上针对linux本地文件,那么需要将文件拷贝到所有worker节点上.

2) Spark默认会为HDFS文件的每一个block创建一个partition,但是也可以通过textFile()的第二个参数手动设置分区数量,只能比block数量多,不可以比block数量少.

### ### 2.3.1 使用本地文件创建RDD(Java版本)

需求:使用本地文件创建RDD,案例:统计文本文件字数

```java

public class CreateRddFromLocalFileWithJava {

public static void main(String[] args) {

//1:首先创建一个SparkConf对象

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("CreateRddFromLocalFileWithJava")

.setMaster("local");

//2:根据SparkConf对象创建SparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:读取本地文件,得到初始的RDD

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

//4:获取每一行内容的长度,其实就是每一行内容的字数

JavaRDD<Integer> lineLength = lines.map(line->line.length());

//5:对每一行的字数进行求和,第一行的字数+第二行的字数=和,和+第三行的字数,以此类推,得到总的字数

Integer count = lineLength.reduce((a,b)->a+b);

//6:将总的字数输出到控制台

System.out.println("这个文件中的总的字数为:"+count);

//7:释放资源

sc.stop();

}

}

```

### ### 2.3.2 使用HDFS文件创建RDD(Java版本)

需求:使用HDFS文件创建RDD,案例:统计文本文件字数

```java

/\*\*

\* 使用HDFS创建RDD,案例:统计文本文件字数

\* @author JackieZhu

\* @date 2018-08-12 17:17:50

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class HDFSFileWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("HDFSFileWithJava");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:使用SparkContext以及其子类的textFile()方法,针对本地文件创建RDD

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

//4:统计文本文件内的字数

JavaRDD<Integer> lineLength = lines.map(new Function<String, Integer>() {

@Override

public Integer call(String v1) throws Exception {

return v1.length();

}

});

//5:统计文件出现的个数

Integer count = lineLength.reduce(new Function2<Integer, Integer, Integer>() {

@Override

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

return v1+v2;

}

});

//6:输出结果

System.out.println("文件总字数是: "+count);

//7:关闭SparkContext

sc.close();

}

}

```

将上述代码打包成jar包,然后使用spark-submit提交到spark集群中去运行.编写spark-submit提交脚本

```shell

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

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

--driver-memory 1G \

--executor-memory 1G \

--executor-cores 3 \

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

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

```

### ### 2.3.3 使用本地文件创建RDD(Scala版本)

需求:使用本地文件创建RDD,统计文件中单词得个数

```scala

/\*\*

\* @author JackieZhu

\* @note 2018-08-13 08:59:08

\* @desc 我是传智JackieZhu 我是不一样的JackieZhu

\*/

object LocalFileWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("LocalFileWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:读取本地文件

val lines: RDD[String] = sc.textFile("f:\\data\\words.txt")

//4:统计每一行得字数长度

val count = lines.map(line=>line.length).reduce((a,b)=>a+b)

//5:输出结果

println("文件中出现得单词数为:"+count)

//6:关闭资源

sc.stop()

}

}

```

### ### 2.3.4 使用HDFS文件创建RDD(Scala版本)

```java

/\*\*

\* @author JackieZhu

\* @note 2018-08-13 09:24:31

\* @desc 我是传智JackieZhu 我是不一样的JackieZhu

\*/

object HDFSFileWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("HDFSFileWithScala")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:读取HDFS文件

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

//4:统计每一行得字数长度

val count = lines.map(line=>line.length).reduce((a,b)=>a+b)

//5:输出结果

println("文件中出现得单词数为:"+count)

//6:关闭资源

sc.stop()

}

}

```

编写spark-submit提交任务脚本

```shell

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

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

--driver-memory 1G \

--executor-memory 1G \

--executor-cores 3 \

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

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

```

## ## 2.4 总结

SparkContext中得textFile()方法,除了可以针对上述几种普通得文件创建RDD之外,还有一些特殊得方法来创建RDD得.

1) SparkContext.wholeTextFiles()方法,可以针对一个目录中得大量小文件,返回<filename,fileContent>组成得pair,作为一个PairRDD,而不是普通得RDD.普通得textFile()返回得RDD中,每个元素就是文件中得一行文本.

2) SparkContext.sequenceFile\[K,V]\()方法,可以针对SequenceFile创建RDD,K和V泛型类型就是SequenceFile得key和value得类型.K和V要求必须是Hadoop得序列化类型,比如IntWritable,Text等

3) SparkContext.hadoopRDD()方法,对于Hadoop得自定义输入类型,可以创建RDD,该方法接收JobConf,InputFormatClass,Key和Value得Class

4) SparkContext.objectFile()方法,可以针对之前调用RDD.saveAsObjectFile()创建得对象序列化得文件,反序列化文件中得数据,并创建一个RDD.

## ## 3.1 RDD操作简介

Spark支持两种类型的操作,transformation以及action.

| 操作描述 | 操作说明 |

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

| transformation | 该操作是从已有的数据集创建新的数据集. |

| action | 该操作主要是对数据集运行计算后,将运算得到的结果返回给驱动程序 |

例如,map就是一个transformation操作,它用于将已有的RDD的每个元素传入一个自定义的函数,并获取一个新的元素,然后将所有的新元素组成一个新的RDD.而reduce就是一中action操作,它用于对RDD中的所有元素进行聚合操作,并获取一个最终的结果,然后返回给Driver程序.

### ### 3.1.1 transformation操作

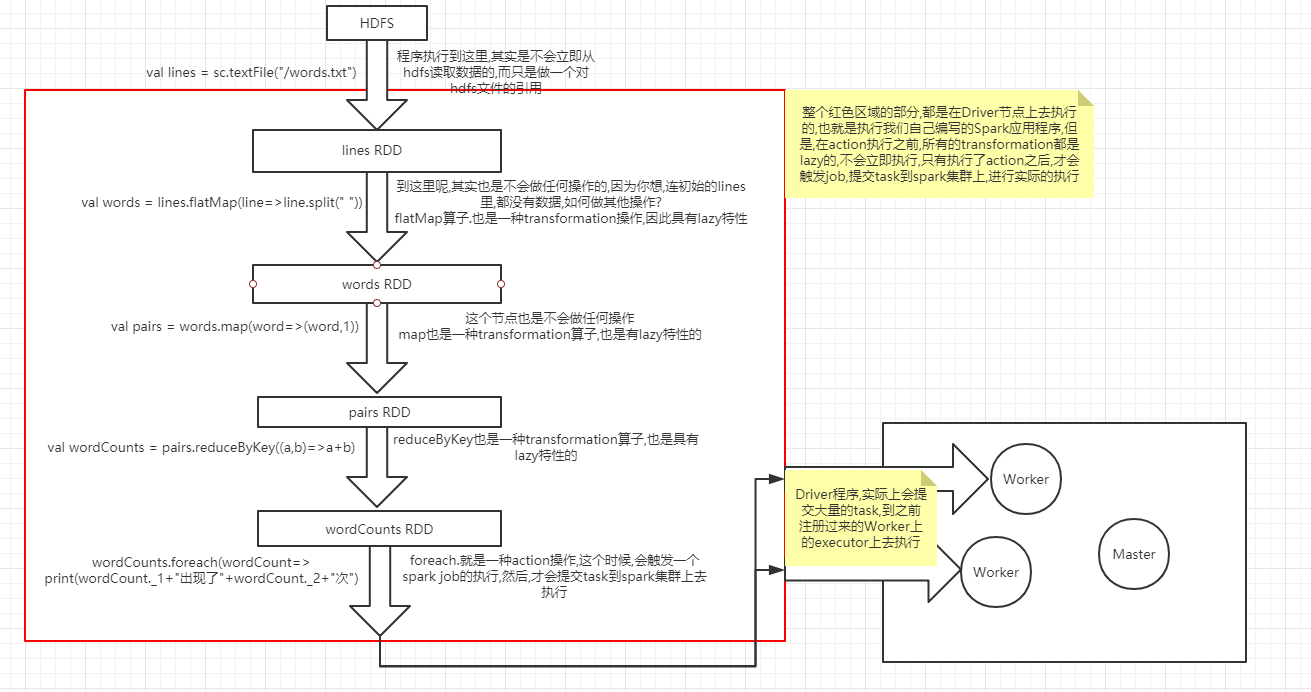
Spark中的所有transformation操作的特点是lazy,lazy特性指的是,如果一个spark应用中只定义了transformation操作,那么即使你执行该应用,这些操作也不会执行.也就是说,transformation是不会触发spark程序的执行的,他们只是记录了对RDD所做的操作,但是不会自发的执行.只有当transformation之后,接着执行了一个action操作,那么所有的transformation才会执行.Spark通过这种lazy特性,来进行底层的spark应用执行的优化,避免产生过多中间结果.

### ### 3.1.2 action操作

action操作执行,会触发一个spark job的运行,从而触发这个action之前所有的transformation的执行,这是action的特性.

### ### 3.1.3 transformation原理图解

![](img\transformation操作原理图解.png)



## ## 3.2 常用transformation和action介绍

### ### 3.2.1 transfotmation操作介绍

| 操作名称 | 操作描述 |

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

| map | 将RDD中的每一个元素传入自定义函数,获取一个新的元素,然后用新的元素组成新的RDD |

| filter | 对RDD中的每个元素进行判断,如果返回true,则保留,返回false,则过滤掉 |

| flatMap | 与map类似,但是对每个元素都可以返回一个或多个新元素 |

| groupByKey | 根据key进行分组,每个key对应一个iterable<value> |

| reduceByKey | 对每个key对应的value进行reduce操作 |

| sortByKey | 对每个key对应的value进行排序操作 |

| join | 对两个包含<key,value>对的RDD进行join操作,每个key join上的pair,都会传入自定义函数进行处理 |

| cogroup | 同join,但是每个key对应的iterable<value>都会传入自定义函数来进行处理 |

### ### 3.2.2 action操作介绍

| 操作名称 | 操作描述 |

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

| reduce | 将RDD中的所有元素进行聚合操作.第一个和第二个元素聚合,值与第三个元素聚合,值与第四个元素聚合,以此类推. |

| collect | 将RDD中所有元素获取到本地客户端 |

| count | 获取RDD元素总数 |

| take(n) | 获取RDD中前n个元素 |

| saveAsTextFile | 将RDD元素保存到文件中,对每个元素调用toString()方法 |

| countByKey | 对每个key对应的值进行count计数 |

| foreach | 遍历RDD中的每个元素. |

### ### 3.2.3 算子操作开发实战

#### #### 3.2.3.1 map,filter

##### Java语言

```java

public class MapFilterWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("MapFilterWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

//3:准备集合

List<Integer> numbers = Arrays.asList(5, 6, 4, 7, 3, 8, 2, 9, 1, 10);

//4:并行化集合创建初始RDD

JavaRDD<Integer> numbersRDD = sc.parallelize(numbers);

//5:对numbersRDD里的每一个元素乘2然后排序

JavaRDD<Integer> sortByRDD = numbersRDD.map(number -> number \* 2).sortBy(num -> num, true, 1);

//6:过滤出大于等于5的元素

JavaRDD<Integer> filter = sortByRDD.filter(num -> num > 5);

//7:将元素以数组的形式打印到控制台

List<Integer> collect = filter.collect();

for (Integer integer : collect) {

System.out.println(integer);

}

//8:释放资源

sc.stop();

}

}

```

##### Scala语言

```scala

object MapFilterWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("MapFilterWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:准备集合

val numbers: List[Int] = List(5, 6, 4, 7, 3, 8, 2, 9, 1, 10)

//4:并行化集合获取RDD

val numbersRDD: RDD[Int] = sc.parallelize(numbers)

//5:对numbersRDD里的每一个元素乘2然后排序

val sortByRDD = numbersRDD.map(\_ \* 2).sortBy(x => x, true)

//过滤出大于等于5的元素

val resultRDD = sortByRDD.filter(\_ >= 5)

//将元素以数组的方式在客户端显示

val collect: Array[Int] = resultRDD.collect

println(collect.toBuffer)

//释放资源

sc.stop()

}

}

```

#### #### 3.2.3.2 flatMap

##### Java语言

```java

public class FlatMapWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("MapFilterWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

//3:准备集合

List<String> linesStr = Arrays.asList("hello world", "hello you", "hello spark hadoop hive hadoop");

//4:并行化集合,创建初始RDD

JavaRDD<String> lines = sc.parallelize(linesStr);

//5:切分每一行数据

JavaRDD<String> words = lines.flatMap(line -> Arrays.asList(line.split(" ")).iterator());

//6:将结果收集到一个集合中

List<String> collect = words.collect();

//打印结果

System.out.println(collect);

//6:释放资源

sc.stop();

}

}

```

##### Scala语言

```scala

object FlatMapWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("FlatMapWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:准备集合

val linesStr: Array[String] = Array("hello world", "hello you", "hello spark hadoop hive hadoop")

//4:并行化创建初始RDD

val lines: RDD[String] = sc.parallelize(linesStr)

//5:切分每一行

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

//6:将单词收集到一个数组中

val result: Array[String] = words.collect()

//7:结果打印到控制台

result.foreach(word=>println(word))

//8:释放资源

sc.stop()

}

}

```

#### #### 3.2.3.3 union&intersection&distinct

##### Java语言

```java

//求并集,交集,去重

public class Union\_Distinct\_IntersectionWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("MapFilterWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

sc.setLogLevel("WARN");

//3:准备集合

List<Integer> list1 = Arrays.asList(5, 6, 4, 3);

List<Integer> list2 = Arrays.asList(1, 2, 3, 4);

//4:并行化集合,获取初始RDD

JavaRDD<Integer> rdd1 = sc.parallelize(list1);

JavaRDD<Integer> rdd2 = sc.parallelize(list2);

//5:求并集

JavaRDD<Integer> rdd3 = rdd1.union(rdd2);

//6:求交集

JavaRDD<Integer> rdd4 = rdd1.intersection(rdd2);

List<Integer> c3 = rdd3.collect();

System.out.println(c3);

//去重

List<Integer> c4 = rdd3.distinct().collect();

System.out.println(c4);

List<Integer> c5 = rdd4.collect();

System.out.println(c5);

}

}

```

##### Scala语言

```scala

object Union\_Distinct\_IntersectionWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("FlatMapWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

sc.setLogLevel("WARN")

//3:准备集合

val list1: List[Int] = List(5, 6, 4, 3)

val list2: List[Int] = List(1, 2, 3, 4)

//4:并行化集合,获取初始RDD

val rdd1: RDD[Int] = sc.parallelize(list1)

val rdd2: RDD[Int] = sc.parallelize(list2)

//求并集

val rdd3 = rdd1.union(rdd2)

//求交集

val rdd4 = rdd1.intersection(rdd2)

val c3 = rdd3.collect()

//去重

val c4 = rdd3.distinct.collect

val c5 = rdd4.collect

println(c3.toBuffer)

println(c4.toBuffer)

println(c5.toBuffer)

//释放资源

sc.stop()

}

}

```

#### #### 3.2.3.4 join&groupByKey

##### Java语言

```java

public class Join\_GroupByKeyWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("Join\_GroupByKeyWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

sc.setLogLevel("WARN");

//3:准备集合

List<Tuple2<String, Integer>> list1 = Arrays.asList(

new Tuple2<String, Integer>("tom", 1),

new Tuple2<String, Integer>("jerry", 3),

new Tuple2<String, Integer>("kitty", 2)

);

List<Tuple2<String, Integer>> list2 = Arrays.asList(

new Tuple2<String, Integer>("jerry", 2),

new Tuple2<String, Integer>("tom", 1),

new Tuple2<String, Integer>("shuke", 2)

);

//并行化集合获取初始RDD

JavaPairRDD<String, Integer> rdd1 = sc.parallelizePairs(list1);

JavaPairRDD<String, Integer> rdd2 = sc.parallelizePairs(list2);

//求join

JavaPairRDD<String, Tuple2<Integer, Integer>> rdd3 = rdd1.join(rdd2);

System.out.println(rdd3.collect());

//求并集

JavaPairRDD<String, Integer> rdd4 = rdd1.union(rdd2);

System.out.println(rdd4.collect());

JavaPairRDD<String, Iterable<Integer>> rdd5 = rdd4.groupByKey();

System.out.println(rdd5.collect());

}

}

```

##### Scala语言

```scala

object Join\_GroupByKeyWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("FlatMapWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

sc.setLogLevel("WARN")

//3:准备集合

val list1: List[(String, Int)] = List(("tom", 1), ("jerry", 3), ("kitty", 2))

val list2: List[(String, Int)] = List(("jerry", 2), ("tom", 1), ("shuke", 2))

//4:并行化创建初始RDD

val rdd1: RDD[(String, Int)] = sc.parallelize(list1)

val rdd2: RDD[(String, Int)] = sc.parallelize(list2)

//求join

val rdd3 = rdd1.join(rdd2)

println(rdd3.collect.toBuffer)

//求并集

val rdd4 = rdd1 union rdd2

println(rdd4.collect.toBuffer)

//按key进行分组

val rdd5=rdd4.groupByKey

println(rdd5.collect.toBuffer)

}

}

```

#### #### 3.2.3.5 cogroup

##### Java语言

```java

public class CogroupWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("Join\_GroupByKeyWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

sc.setLogLevel("WARN");

//3:准备集合

List<Tuple2<String, Integer>> list1 = Arrays.asList(

new Tuple2<String, Integer>("tom", 1),

new Tuple2<String, Integer>("tom", 2),

new Tuple2<String, Integer>("jerry", 3),

new Tuple2<String, Integer>("kitty", 2)

);

List<Tuple2<String, Integer>> list2 = Arrays.asList(

new Tuple2<String, Integer>("jerry", 2),

new Tuple2<String, Integer>("tom", 1),

new Tuple2<String, Integer>("jim", 2)

);

//并行化

JavaPairRDD<String, Integer> rdd1 = sc.parallelizePairs(list1);

JavaPairRDD<String, Integer> rdd2 = sc.parallelizePairs(list2);

//

JavaPairRDD<String, Tuple2<Iterable<Integer>, Iterable<Integer>>> rdd3 = rdd1.cogroup(rdd2);

System.out.println(rdd3.collect());

}

}

```

##### Scala语言

```scala

object CogroupWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("FlatMapWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

sc.setLogLevel("WARN")

val list1 = List(("tom", 1), ("tom", 2), ("jerry", 3), ("kitty", 2))

val list2 = List(("jerry", 2), ("tom", 1), ("jim", 2))

//并行化集合

val rdd1: RDD[(String, Int)] = sc.parallelize(list1)

val rdd2 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("jim", 2)))

//cogroup

val rdd3 = rdd1.cogroup(rdd2)

//注意cogroup与groupByKey的区别

println(rdd3.collect.toBuffer)

}

}

```

#### #### 3.2.3.6 reduce

##### Java语言

```java

public class ReduceWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("Join\_GroupByKeyWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

sc.setLogLevel("WARN");

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

JavaRDD<Integer> rdd1 = sc.parallelize(numbers);

Integer count = rdd1.reduce((a, b) -> a + b);

System.out.println(count);

}

}

```

##### Scala语言

```scala

object ReduceWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("FlatMapWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

sc.setLogLevel("WARN")

val rdd1 = sc.parallelize(List(1, 2, 3, 4, 5)) //1+2=3 3+3=6 6+4=10 10 + 5= 15

//reduce聚合

val rdd2 = rdd1.reduce((a,b)=>a+b)

//rdd2.collect

println(rdd2)

}

}

```

#### #### 3.2.3.7 reduceByKey&sortByKey

##### Java语言

```java

public class ReduceByKey\_SortByKeyWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("Join\_GroupByKeyWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

sc.setLogLevel("WARN");

//3:准备集合

List<Tuple2<String, Integer>> list1 = Arrays.asList(

new Tuple2<String, Integer>("tom", 1),

new Tuple2<String, Integer>("jerry", 3),

new Tuple2<String, Integer>("kitty", 2),

new Tuple2<String, Integer>("shuke", 1)

);

List<Tuple2<String, Integer>> list2 = Arrays.asList(

new Tuple2<String, Integer>("jerry", 2),

new Tuple2<String, Integer>("tom", 3),

new Tuple2<String, Integer>("shuke", 2),

new Tuple2<String, Integer>("kitty", 5)

);

//4:并行化集合,创建初始RDD

JavaPairRDD<String, Integer> rdd1 = sc.parallelizePairs(list1);

JavaPairRDD<String, Integer> rdd2 = sc.parallelizePairs(list2);

//5:rdd1和rdd2进行并集

JavaPairRDD<String, Integer> rdd3 = rdd1.union(rdd2);

//6:安装key进行聚合

JavaPairRDD<String, Integer> rdd4 = rdd3.reduceByKey((a, b) -> a + b);

//7:将rdd4收集到一个集合中

System.out.println(rdd4.collect());

//8:按照value的值进行降序排序

JavaRDD<Tuple2> result = rdd4.map(value -> new Tuple2(value.\_2, value.\_1)).sortBy(value -> value.\_1, false, 1).map(value -> new Tuple2(value.\_2, value.\_1));

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

}

}

```

##### Scala语言

```scala

object ReduceByKey\_SortByKeyWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("ReduceByKey\_SortByKeyWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

sc.setLogLevel("WARN")

//3:准备集合

val list1: List[(String, Int)] = List(("tom", 1), ("jerry", 3), ("kitty", 2), ("shuke", 1))

val list2: List[(String, Int)] = List(("jerry", 2), ("tom", 3), ("shuke", 2), ("kitty", 5))

//4:并行化集合,创建初始RDD

val rdd1: RDD[(String, Int)] = sc.parallelize(list1)

val rdd2: RDD[(String, Int)] = sc.parallelize(list2)

val rdd3: RDD[(String, Int)] = rdd1.union(rdd2)

//按key进行聚合

val rdd4 = rdd3.reduceByKey((a,b)=>a+b)

println(rdd4.collect.toBuffer)

//按value的降序排序

val rdd5 = rdd4.map(t => (t.\_2, t.\_1)).sortByKey(false).map(t => (t.\_2, t.\_1))

println(rdd5.collect.toBuffer)

}

}

```

#### #### 3.2.3.8 repartition&coalesce

\*\*注意：repartition可以增加和减少rdd中的分区数，coalesce只能减少rdd分区数，增加rdd分区数不会生效。\*\*

##### Java语言

```java

public class Repartition\_CoalesceWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("Join\_GroupByKeyWithJava")

.setMaster("local");

//2:创建JavaSparkContext

JavaSparkContext sc = new JavaSparkContext(conf);

sc.setLogLevel("WARN");

//3:准备集合

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

JavaRDD<Integer> rdd1 = sc.parallelize(numbers, 3);

//4:查看分区大小

System.out.println(rdd1.partitions().size());

//5:利用repartition改变rdd1的分区大小

System.out.println(rdd1.repartition(2).partitions().size());

System.out.println(rdd1.repartition(4).partitions().size());

//利用coalesce改变rdd1分区数

//减少分区

System.out.println(rdd1.coalesce(2).partitions().size());

}

}

```

##### Scala语言

```scala

object Repartition\_CoalesceWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("Repartition\_CoalesceWithScala")

.setMaster("local")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

sc.setLogLevel("WARN")

//3:准备集合

val rdd1 = sc.parallelize(1 to 10,3)

//利用repartition改变rdd1分区数

//减少分区

println(rdd1.repartition(2).partitions.size)

//增加分区

println(rdd1.repartition(4).partitions.size)

//利用coalesce改变rdd1分区数

//减少分区

println(rdd1.coalesce(2).partitions.size)

}

}

```

# # 第四章 RDD编程实战

## ## 4.1 访问的PV

### ### 4.1.1 Java版本

```java

/\*\*

\* @author JackieZhu

\* @date 2018-08-23 09:44:19

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class PVCountWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("PVCountWithJava")

.setMaster("local[2]");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:读取日志文件

JavaRDD<String> lines = sc.textFile("f:\\data\\access.log");

//4:将每一行记录进行统计

JavaPairRDD<String, Integer> pvWithOne = lines.mapToPair(new PairFunction<String, String, Integer>() {

@Override

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

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

}

});

//5:对每一行记录进行统计输出

JavaPairRDD<String, Integer> totalPv = pvWithOne.reduceByKey(new Function2<Integer, Integer, Integer>() {

@Override

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

return v1 + v2;

}

});

//6:将结果进行打印

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

@Override

public void call(Tuple2<String, Integer> tuple2) throws Exception {

System.out.println(tuple2.\_1+"的页面访问量为:"+tuple2.\_2);

}

});

//7:释放资源

sc.stop();

}

}

```

### ### 4.1.2 Scala版本

```scala

/\*\*

\* @author JackieZhu

\* @note 2018-08-23 09:52:43

\* @desc 我是传智JackieZhu 我是不一样的JackieZhu

\*/

object PVCountWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("PVCountWithScala")

.setMaster("local[2]")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:读取数据

val lines: RDD[String] = sc.textFile("f:\\data\\access.log")

//4:对每一行数据进行统计

val pvWithOne: RDD[(String, Int)] = lines.map(line=>("pv",1))

//5:对所有的pv进行聚合统计输出

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

//6:遍历所有的内容

lineCounts.foreach(lineCount=>println(lineCount.\_1+"的访问量为:"+lineCount.\_2))

//7:释放资源

sc.stop()

}

}

```

### ### 4.1.3 lamdba版本

```java

/\*\*

\* @author JackieZhu

\* @date 2018-08-23 09:44:19

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class PVCountWithLamdba {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("PVCountWithLamdba")

.setMaster("local[2]");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:读取日志文件

JavaRDD<String> lines = sc.textFile("f:\\data\\access.log");

//4:将每一行记录进行统计

JavaPairRDD<String, Integer> pvWithOne = lines.mapToPair(s->new Tuple2<String,Integer>("pv",1));

//5:对每一行记录进行统计输出

JavaPairRDD<String, Integer> totalPv = pvWithOne.reduceByKey((a,b)->a+b);

//6:将结果进行打印

totalPv.foreach(tuple2 -> System.out.println(tuple2.\_1+"的访问量为:"+tuple2.\_2));

//7:释放资源

sc.stop();

}

}

```

## ## 4.2 访问的UV

### ### 4.2.1 Java版本

```java

/\*\*

\* @author JackieZhu

\* @date 2018-08-23 10:02:47

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class UVCountWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("UVCountWithJava")

.setMaster("local[2]");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:读取日志文件

JavaRDD<String> lines = sc.textFile("f:\\data\\access.log");

//4:对每一行数据切分.

JavaRDD<String[]> words = lines.map(new Function<String, String[]>() {

@Override

public String[] call(String line) throws Exception {

return line.split(" ");

}

});

//5:获取所有的ip地址

JavaRDD<String> ips = words.map(new Function<String[], String>() {

@Override

public String call(String[] v1) throws Exception {

return v1[0];

}

});

//6:对ip地址进行去重,输出最后的格式为("UV",1)

JavaPairRDD<String, Integer> uvAndOne = ips.distinct().mapToPair(new PairFunction<String, String, Integer>() {

@Override

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

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

}

});

//7:统计输出所有的结果

JavaPairRDD<String, Integer> totalUV = uvAndOne.reduceByKey(new Function2<Integer, Integer, Integer>() {

@Override

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

return v1 + v2;

}

});

//8:输出所有的结果

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

@Override

public void call(Tuple2<String, Integer> tuple2) throws Exception {

System.out.println(tuple2.\_1+"的独立访客量为:"+tuple2.\_2);

}

});

//9:释放资源

sc.stop();

}

}

```

### ### 4.2.2 Scala版本

```scala

/\*\*

\* @author JackieZhu

\* @note 2018-08-23 10:19:54

\* @desc 我是传智JackieZhu 我是不一样的JackieZhu

\*/

object UVCountWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("PVCountWithScala")

.setMaster("local[2]")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:读取数据

val lines: RDD[String] = sc.textFile("f:\\data\\access.log")

//4:对每一行数据进行分割,并获取ip地址

val ips: RDD[String] = lines.map(line=>line.split(" ")).map(x=>x(0))

//5:对ip地址进行去重,输出格式为("UV",1)

val uvAndOne: RDD[(String, Int)] = ips.distinct().map(x=>("UV",1))

//6:聚合输出

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

//7:遍历输出所有的结果

totalUV.foreach(x=>println(x.\_1+"独立访客量为:"+x.\_2))

//8:释放资源

sc.stop()

}

}

```

### ### 4.2.3 Lamdba版本

```java

/\*\*

\* @author JackieZhu

\* @date 2018-08-23 10:02:47

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class UVCountWithLamdba {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("UVCountWithLamdba")

.setMaster("local[2]");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:读取日志文件

JavaRDD<String> lines = sc.textFile("f:\\data\\access.log");

//4:对每一行数据切分.

JavaRDD<String[]> words = lines.map(line->line.split(" "));

//5:获取所有的ip地址

JavaRDD<String> ips = words.map(x->x[0]);

//6:对ip地址进行去重,输出最后的格式为("UV",1)

JavaPairRDD<String, Integer> uvAndOne = ips.distinct().mapToPair(x->new Tuple2<>("UV",1));

//7:统计输出所有的结果

JavaPairRDD<String, Integer> totalUV = uvAndOne.reduceByKey((a,b)->a+b);

//8:输出所有的结果

totalUV.foreach(tuple2 -> System.out.println(tuple2.\_1+"独立访客量为:"+tuple2.\_2));

//9:释放资源

sc.stop();

}

}

```

## ## 4.3 访问的topN

### ### 4.3.1 Java版本

```java

/\*\*

\* @author JackieZhu

\* @date 2018-08-23 10:28:48

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class TopNWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("UVCountWithJava")

.setMaster("local[2]");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:读取日志文件

JavaRDD<String> lines = sc.textFile("f:\\data\\access.log");

//4:将每一行数据作为输入,按照空格对每一行数据进行切分

JavaRDD<String[]> splitRDD = lines.map(new Function<String, String[]>() {

@Override

public String[] call(String v1) throws Exception {

return v1.split(" ");

}

});

//5:对splitRDD中的每一行数据进行切分

JavaRDD<String[]> filterRDD = splitRDD.filter(new Function<String[], Boolean>() {

@Override

public Boolean call(String[] v1) throws Exception {

return v1.length > 10 && !v1[10].equals("\"-\"");

}

});

//6:将过滤后的元素转换成一个元组

JavaPairRDD<String, Integer> pairs = filterRDD.mapToPair(new PairFunction<String[], String, Integer>() {

@Override

public Tuple2<String, Integer> call(String[] strings) throws Exception {

return new Tuple2<>(strings[10], 1);

}

});

//7:对pairs元素进行聚合统计以及排序

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

@Override

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

return v1 + v2;

}

});

//8:对聚合统计的结果进行位置调换

JavaPairRDD<Integer, String> reverse = reduceByKey.mapToPair(new PairFunction<Tuple2<String, Integer>, Integer, String>() {

@Override

public Tuple2<Integer, String> call(Tuple2<String, Integer> tuple2) throws Exception {

return new Tuple2<>(tuple2.\_2, tuple2.\_1);

}

});

//9:对调换之后的结果进行排序

JavaPairRDD<Integer, String> sortByKey = reverse.sortByKey(false);

//10:将排序之后的结果进行调换

JavaPairRDD<String, Integer> result = sortByKey.mapToPair(new PairFunction<Tuple2<Integer, String>, String, Integer>() {

@Override

public Tuple2<String, Integer> call(Tuple2<Integer, String> tuple2) throws Exception {

return new Tuple2<>(tuple2.\_2, tuple2.\_1);

}

});

//11:取出前5名

List<Tuple2<String, Integer>> take = result.take(5);

//12 输出结果

for (Tuple2<String, Integer> tuple : take) {

System.out.println(tuple.toString());

}

//13:释放资源

sc.stop();

}

}

```

### ### 4.3.2 Scala版本

```scala

/\*\*

\* @author JackieZhu

\* @note 2018-08-23 10:48:56

\* @desc 我是传智JackieZhu 我是不一样的JackieZhu

\*/

object TopNWithScala {

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

//1:创建SparkConf对象

val conf: SparkConf = new SparkConf()

.setAppName("PVCountWithScala")

.setMaster("local[2]")

//2:创建SparkContext对象

val sc: SparkContext = new SparkContext(conf)

//3:读取数据

val lines: RDD[String] = sc.textFile("f:\\data\\access.log")

//4:对每一行数据进行分割

val mapRdd: RDD[Array[String]] = lines.map(line=>line.split(" "))

//5:对切分后的数据进行过滤

val filterRdd: RDD[Array[String]] = mapRdd.filter(x=>x.length>10).filter(x=>x(10)!="\"-\"")

//6:将过滤后的每一行数据输出为一个元组

val urlAndOne: RDD[(String, Int)] = filterRdd.map(x=>(x(10),1))

//7:对url进行聚合统计

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

//8:对urlCounts进行排序

val result: RDD[(String, Int)] = urlCounts.sortBy(x=>x.\_2,false)

//9:取出前5

val finalResult = result.take(5)

//10 输出所有的结果

finalResult.foreach(x=>println(x))

//11 释放资源

sc.stop()

}

}

```

### ### 4.3.3 Lamdba版本

```java

/\*\*

\* @author JackieZhu

\* @date 2018-08-23 11:01:36

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class TopNWithLamdba {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("TopNWithLamdba")

.setMaster("local[2]");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:读取日志文件

JavaRDD<String> lines = sc.textFile("f:\\data\\access.log");

//4:将每一行数据作为输入,按照空格对每一行数据进行切分

JavaRDD<String[]> splitRDD = lines.map(x->x.split(" "));

//5:对splitRDD中的每一行数据进行切分

JavaRDD<String[]> filterRDD = splitRDD.filter(x->x.length>10 && !x[10].equals("\"-\""));

//6:将过滤后的元素转换成一个元组

JavaPairRDD<String, Integer> pairs = filterRDD.mapToPair(arr->new Tuple2<>(arr[10],1));

//7:对pairs元素进行聚合统计以及排序

JavaPairRDD<String, Integer> reduceByKey = pairs.reduceByKey((a,b)->a+b);

//8:对聚合统计的结果进行位置调换

JavaPairRDD<Integer, String> reverse = reduceByKey.mapToPair(tuple2 -> new Tuple2<>(tuple2.\_2,tuple2.\_1));

//9:对调换之后的结果进行排序

JavaPairRDD<Integer, String> sortByKey = reverse.sortByKey(false);

//10:将排序之后的结果进行调换

JavaPairRDD<String, Integer> result = sortByKey.mapToPair(tuple2 -> new Tuple2<>(tuple2.\_2,tuple2.\_1));

//11:取出前5名

List<Tuple2<String, Integer>> take = result.take(5);

//12 输出结果

for (Tuple2<String, Integer> tuple : take) {

System.out.println(tuple.toString());

}

//13:释放资源

sc.stop();

}

}

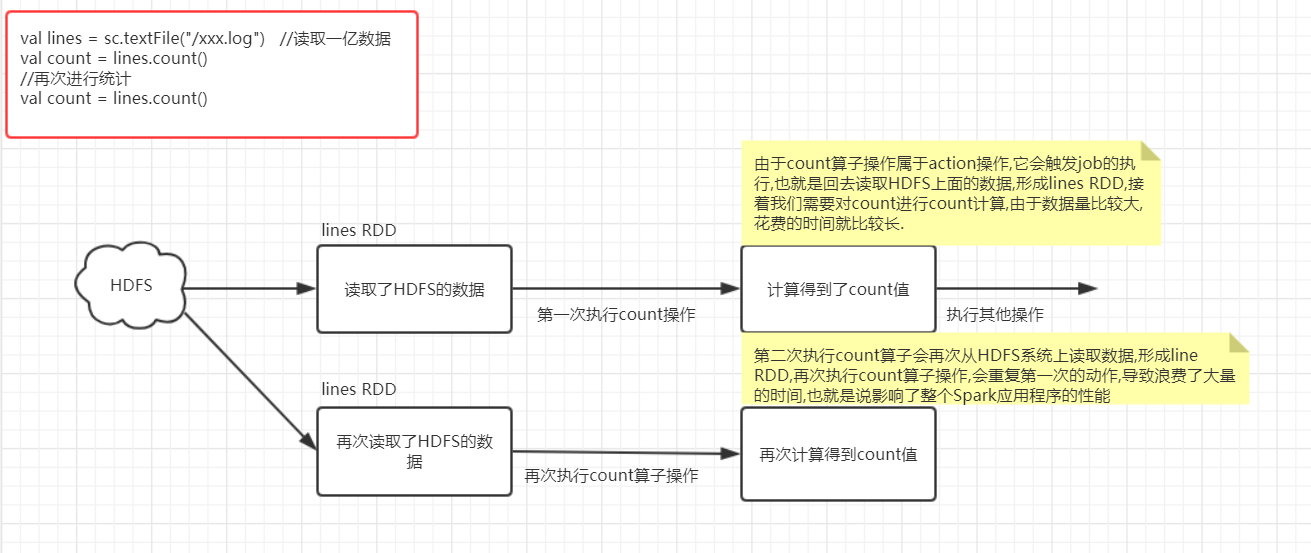
```

# # 第五章 RDD持久化(缓存)详解

## ## 5.1 RDD持久化原理图解

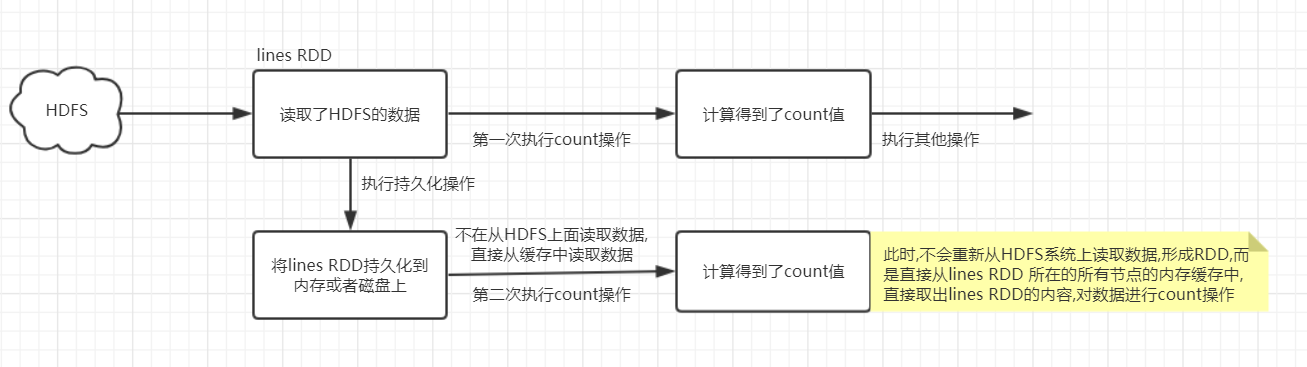
### ### 5.1.1 不使用RDD持久化导致的问题图解

![](img\不使用RDD持久化的问题的原理.png)



### ### 5.1.2 使用RDD持久化原理图解

![](img\RDD持久化原理图解.png)



### ### 5.1.3 RDD持久化原理总结

Spark非常重要的一个功能特性就是可以将RDD持久化在内存中。当对RDD执行持久化操作时，每个节点都会将自己操作的RDD的partition持久化到内存中，并且在之后对该RDD的反复使用中，直接使用内存缓存的partition。这样的话，对于针对一个RDD反复执行多个操作的场景，就只要对RDD计算一次即可，后面直接使用该RDD，而不需要反复计算多次该RDD。

巧妙使用RDD持久化，甚至在某些场景下，可以将spark应用程序的性能提升10倍。对于迭代式算法和快速交互式应用来说，RDD持久化，是非常重要的。

要持久化一个RDD，只要调用其cache()或者persist()方法即可。在该RDD第一次被计算出来时，就会直接缓存在每个节点中。而且Spark的持久化机制还是自动容错的，如果持久化的RDD的任何partition丢失了，那么Spark会自动通过其源RDD，使用transformation操作重新计算该partition。

cache()和persist()的区别在于，cache()是persist()的一种简化方式，cache()的底层就是调用的persist()的无参版本，同时就是调用persist(MEMORY\_ONLY)，将数据持久化到内存中。如果需要从内存中清除缓存，那么可以使用unpersist()方法。

Spark自己也会在shuffle操作时，进行数据的持久化，比如写入磁盘，主要是为了在节点失败时，避免需要重新计算整个过程。

## ## 5.2 RDD持久化练习

```java

/\*\*

\* RDD持久化

\* @author JackieZhu

\* @date 2018-08-15 15:31:01

\* @description 我是传智JackieZhu, 我是不一样的JackieZhu

\*/

public class PersistWithJava {

public static void main(String[] args) {

//1:创建SparkConf对象

SparkConf conf = new SparkConf()

.setAppName("PersistWithJava")

.setMaster("local");

//2:创建JavaSparkContext对象

JavaSparkContext sc = new JavaSparkContext(conf);

//3:使用SparkContext以及其子类的textFile()方法,针对本地文件创建RDD

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

//4:统计words.txt的行数

long beginTime = System.currentTimeMillis();

long count = lines.count();

System.out.println(count);

long endTime = System.currentTimeMillis();

System.out.println("总共花费了时间为:"+(endTime-beginTime)+"毫秒");

beginTime = System.currentTimeMillis();

count = lines.count();

System.out.println(count);

endTime = System.currentTimeMillis();

System.out.println("总共花费了时间为:"+(endTime-beginTime)+"毫秒");

}

}

```

## ## 5.3 RDD持久化策略

RDD持久化是可以选择不同的策略的,比如可以将RDD持久化在内存中,持久化到内存中,持久化到磁盘上,使用序列化的方式持久化,多持久化的数据进行多路复用.只要在调用persist()时传入对应的Storagel Level即可.

| 持久化级别 | 说明 |

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

| MEMORY\_ONLY | 以非序列化的Java对象的方式持久化在JVM内存中.如果内存无法完全存储RDD所有的partition,那么哪些没有持久化的partition就会在下一次需要使用它的时候,重新呗计算. |

| MEMORY\_AND\_DISK | 总体同上,但是当某些partition无法存储在内存中时,会持久化到磁盘中,下次需要使用这些partition时,需要从磁盘上读取 |

| MEMORY\_ONLY\_SER | 总体同第一个,但是会使用Java序列化方式,将Java对象序列化后进行持久化,可以减少内存开销,但是需要进行反序列化,因此会加大CPU开销 |

| MEMORY\_AND\_DISK\_SER | 同MEMORY\_AND\_DSK。但是使用序列化方式持久化Java对象。 |

| DISK\_ONLY | 使用非序列化Java对象的方式持久化，完全存储到磁盘上。 |

| MEMORY\_ONLY\_2 | 如果是尾部加了2的持久化级别，表示会将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可。 |

| MEMORY\_AND\_DISK\_2 | 如果是尾部加了2的持久化级别，表示会将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可。 |

## ## 5.4 RDD持久化策略选择

Spark提供的多种持久化级别，主要是为了在CPU和内存消耗之间进行取舍。下面是一些通用的持久化级别的选择建议：

1、优先使用MEMORY\_ONLY，如果可以缓存所有数据的话，那么就使用这种策略。因为纯内存速度最快，而且没有序列化，不需要消耗CPU进行反序列化操作。

2、如果MEMORY\_ONLY策略，无法存储的下所有数据的话，那么使用MEMORY\_ONLY\_SER，将数据进行序列化进行存储，纯内存操作还是非常快，只是要消耗CPU进行反序列化。

3、如果需要进行快速的失败恢复，那么就选择带后缀为\_2的策略，进行数据的备份，这样在失败时，就不需要重新计算了。

4、能不使用DISK相关的策略，就不用使用，有的时候，从磁盘读取数据，还不如重新计算一次。

# # 第六章 RDD的依赖关系

## ## 6.1 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/shuffle dependency）

## ## 6.2 窄依赖

窄依赖:英文全名,Narrow Dependency.什么样的情况,叫做窄依赖啊?一个RDD,对它的父RDD,只有简单的一对一的依赖关系.也就是说,RDD的每个partition,仅仅依赖于父RDD中的一个partition,父RDD和子RDD的partition之间的对应关系,是一对一的.这种情况下,是简单的RDD之间的依赖关系,也被称之为窄依赖

## ## 6.3 宽依赖

宽依赖 Shuffle Dependency.也就是说,每一个父RDD的partition中的数据,都可能会传输一部分,到下一个RDD的每个partition中,此时就会出现,父RDD和子RDD的partition之间,具有交互错综复杂的关系,那么这种情况,就叫做两个RDD之间的宽依赖.同事,他们之间发生的操作,就是Shuffle.

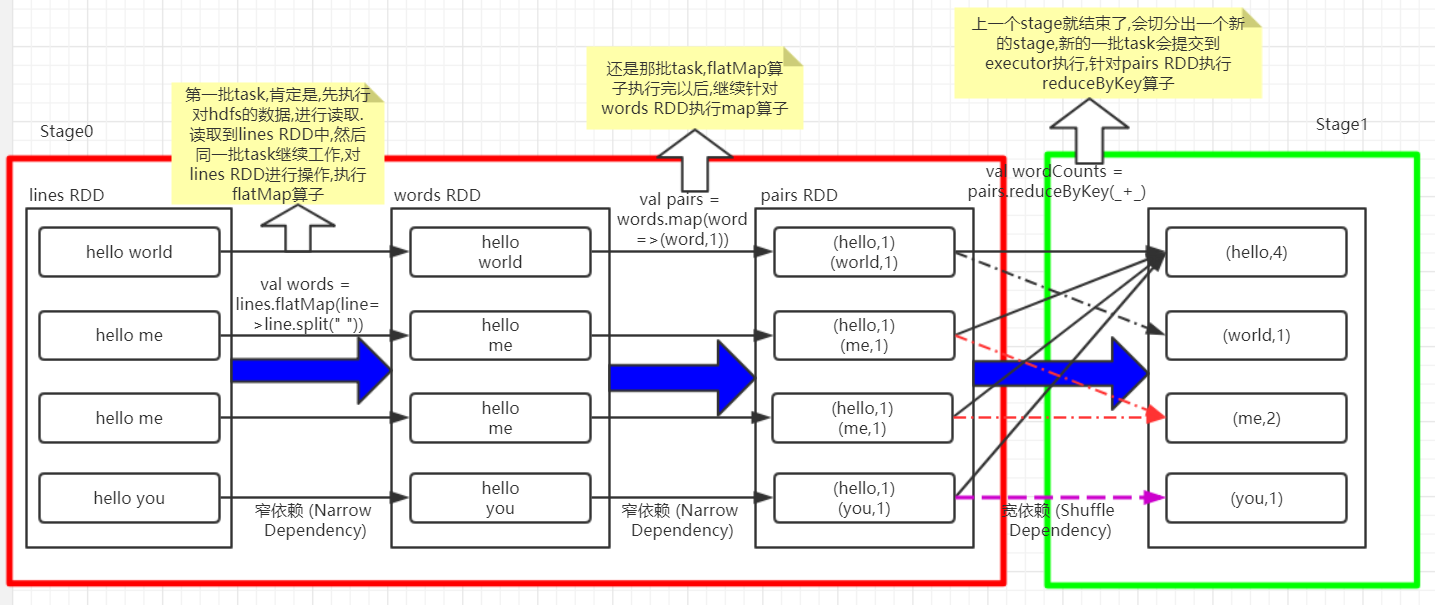
## ## 6.4 Lineage(血统)

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。

## ## 6.5 DAG的生成

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。

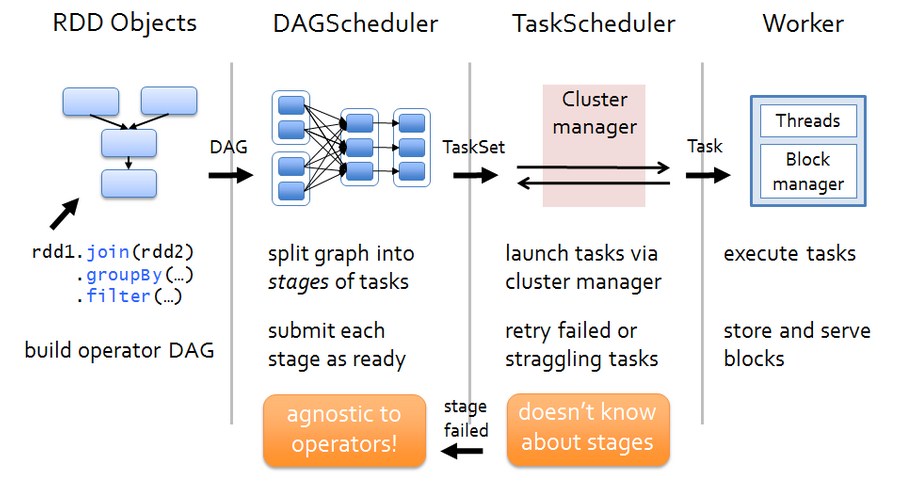
![](img\宽窄依赖.png)



# # 第七章 Spark任务调度

## ## 7.1 Spark任务调度流程图

![](img\Spark任务调度流程图.jpg)



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。

## ## 7.2 DAGScheduler介绍

- DAGScheduler对DAG有向无环图进行Stage划分。

- 记录哪个RDD或者Stage输出被物化（缓存），通常在一个复杂的shuffle之后，通常物化一下(cache、persist)，方便之后的计算。

- 重新提交shuffle输出丢失的stage（stage内部计算出错）给TaskScheduler

- 将Taskset传给底层调度器

- spark-cluster TaskScheduler

- yarn-cluster YarnClusterScheduler

- yarn-client YarnClientClusterScheduler

## ## 7.3 TaskScheduler介绍

- 为每一个TaskSet构建一个TaskSetManager 实例管理这个TaskSet 的生命周期

- 数据本地性决定每个Task最佳位置

- 提交 taskset( 一组task) 到集群运行并监控

- 推测执行，碰到计算缓慢任务需要放到别的节点上重试

- 重新提交Shuffle输出丢失的Stage给DAGScheduler

# # 第八章 RDD容错机制之checkpoint

## ## 8.1 checkpoint简介

- Spark 在生产环境下经常会面临transformation的RDD非常多（例如一个Job中包含1万个RDD）或者具体transformation的RDD本身计算特别复杂或者耗时（例如计算时长超过1个小时），这个时候就要考虑对计算结果数据持久化保存；

- Spark是擅长多步骤迭代的，同时擅长基于Job的复用，这个时候如果能够对曾经计算的过程产生的数据进行复用，就可以极大的提升效率；

- 如果采用persist把数据放在内存中，虽然是快速的，但是也是最不可靠的；如果把数据放在磁盘上，也不是完全可靠的！例如磁盘会损坏，系统管理员可能清空磁盘。

- Checkpoint的产生就是为了相对而言更加可靠的持久化数据，在Checkpoint的时候可以指定把数据放在本地，并且是多副本的方式，但是在生产环境下是放在HDFS上，这就天然的借助了HDFS高容错、高可靠的特征来完成了最大化的可靠的持久化数据的方式；假如进行一个1万个算子操作，在9000个算子的时候persist，数据还是有可能丢失的，但是如果checkpoint，数据丢失的概率几乎为0。

## ## 8.2 checkpoint原理机制

1. 当RDD使用cache机制从内存中读取数据，如果数据没有读到，会使用checkpoint机制读取数据。此时如果没有checkpoint机制，那么就需要找到父RDD重新计算数据了，因此checkpoint是个很重要的容错机制。checkpoint就是对于一个RDD chain（链）如果后面需要反复使用某些中间结果RDD，可能因为一些故障导致该中间数据丢失，那么就可以针对该RDD启动checkpoint机制，使用checkpoint首先需要调用sparkContext的setCheckpoint方法，设置一个容错文件系统目录，比如hdfs，然后对RDD调用checkpoint方法。之后在RDD所处的job运行结束后，会启动一个单独的job来将checkpoint过的数据写入之前设置的文件系统持久化，进行高可用。所以后面的计算在使用该RDD时，如果数据丢失了，但是还是可以从它的checkpoint中读取数据，不需要重新计算。

2. persist或者cache与checkpoint的区别在于,前者持久化只是将数据保存在BlockManager中但是其lineage是不变的，但是后者checkpoint执行完后，rdd已经没有依赖RDD，只有一个checkpointRDD，checkpoint之后，RDD的lineage就改变了。persist或者cache持久化的数据丢失的可能性更大，因为可能磁盘或内存被清理，但是checkpoint的数据通常保存到hdfs上，放在了高容错文件系统。

```scala

//设置checkpoint的目录

sc.setCheckpointDir("./checkpoint")

//将结果持久化到checkpoint中

wordCounts.checkpoint()

```