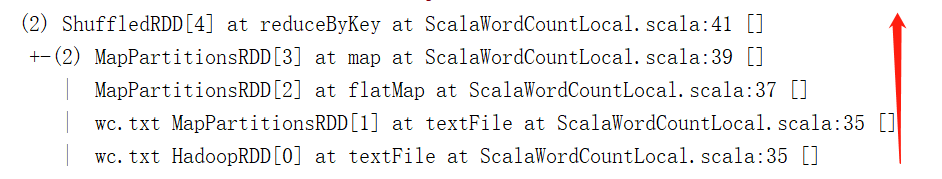
# Spark任务运行机制

## 一个wordcount中产生了几个rdd

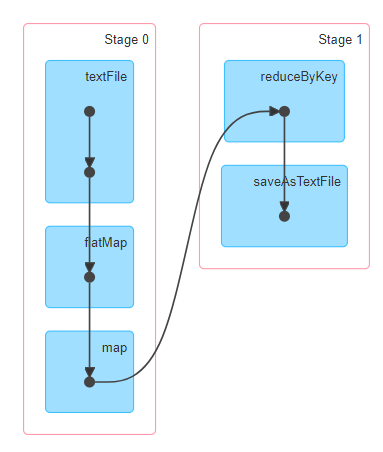
|  |
| --- |
| *// 创建SparkContext* **val** sc: SparkContext = **new** SparkContext(conf) *// 读取数据* **val** file: RDD[String] = sc.textFile(input) *// 切分并压平* **val** words: RDD[String] = file.flatMap(\_.split(**" "**)) *// 组装* **val** wordAndOne: RDD[(String, Int)] = words.map((\_, 1)) *// 分组聚合* **val** result: RDD[(String, Int)] = wordAndOne.reduceByKey(\_ + \_) *// 直接存储到hdfs中* result.saveAsTextFile(output) *// 释放资源* sc.stop() |

问题：

最简版的wordcount，一共产生了几个RDD?



这个wordcount的任务，一共产生了6个RDD



可以通过toDebugString 方法来查看rdd的依赖关系图。

*// 可以查看RDD的依赖关系  
println*(result.toDebugString)

## spark任务运行的数据流向图

|  |
| --- |
| scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/wordcount/input")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/wordcount/input MapPartitionsRDD[1] at textFile at <console>:24  scala> val rdd2 = rdd1.flatMap(\_.split(" ")).map((\_,1))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:26  scala> val rdd3 = rdd2.reduceByKey(\_+\_,2)  rdd3: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:28  scala> rdd3.saveAsTextFile("hdfs://hdp-01:9000/wordcount/output11") |

逻辑图

## 任务调度的流程：

application🡪 job -🡪 stage 🡪 task

## 物理执行流程：

当执行我们自定义程序（main）的时候，并没有真正的执行程序。

Driver会记录rdd之间的依赖关系，每一个rdd传递了什么函数，记录rdd要去哪里读取数据。

当触发action算子的时候，真正开始执行任务。

DAG 🡪 切分stage -🡪 组装Task 🡪 把Task提交给executor去执行。

任务正式的开始按照我们的业务逻辑执行。

# 自定义分区器

## 补充：asInstanceOf isInstanceOf

|  |
| --- |
| **val** c:Any = 100  *// 类型强转* **val** d = c.asInstanceOf[Int] + 3 *println*(d)   **val** str:String = **"nvshen"** *println*(str.isInstanceOf[String]) |

|  |
| --- |
| **object** SubFacTeacher {   *// 取Top3* **val** *topN* = 2   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   *// 1,读数据* **val** file: RDD[String] = sc.textFile(**"f:/mrdata/teacher.log"**)   *// 2，数据预处理 提取出 学科名称 teachername* **val** splitRdd: RDD[((String, String), Int)] = file.map(str => {  *// 协议 host:port URL  // http://bigdata.edu360.cn/laozhang* **val** index: Int = str.lastIndexOf(**"/"**)  *// 获取老师的名称* **val** tName = str.substring(index + 1)   **val** url: URL = **new** URL(str.substring(0, index))  *// 获取hostName* **val** host = url.getHost() *// bigdata.edu360.cn  // 分隔符需要进行转义* **val** split = host.split(**"\\."**)  *// 获取到学科名称* **val** subject = split(0)   *// 把数据组装成嵌套元组* ((subject, tName), 1)  })   **val** subjectArr: Array[String] = splitRdd.map(\_.\_1.\_1).distinct().collect()   *// 分组聚合 2次数据的重新分发  // val result: RDD[((String, String), Int)] = splitRdd.reduceByKey(\_ + \_)  // // 把相同学科的数据分到一个分区中 + mapPartitions  // val parRes: RDD[((String, String), Int)] = result.partitionBy(MyPartitioner(subjectArr))    // 优化的自定义分区器 只有一次数据的shuffle* **val** parRes: RDD[((String, String), Int)] = splitRdd.reduceByKey(*MyPartitioner*(subjectArr), \_ + \_)   **val** finalRes = parRes.mapPartitions(tp => { *// tp ((String, String), Int)  // 要求返回值类型是Iterator* tp.toList.sortBy(-\_.\_2).take(*topN*).iterator  })   *// 思考题？ TreeMap来实现 存储的数据是有序的（按照次数做降序） 只存储 topN条数据* finalRes.foreach(*println*)   sc.stop()  } }  **object** MyPartitioner {  *// 定义一个apply方法* **def** apply(subject: Array[String]): MyPartitioner = **new** MyPartitioner(subject) }  *// 自定义类 分区器* **class** MyPartitioner(subject: Array[String]) **extends** Partitioner {   *// 直接把数组转换为Map 存储的 学科 -- 分区编号* **val** *subMap*: Map[String, Int] = subject.zipWithIndex.toMap   *// 分区的数量 = 学科的数量* **override def** numPartitions: Int = subject.length   *// 根据key来获取分区的编号 实际上 学科的名称 --》 分区编号* **override def** getPartition(key: Any): Int = {   *// 这里的key是什么类型？* **val** (subject, \_) = key.asInstanceOf[(String, String)]  *// asInstanceOf 类型强转 (类型) 变量  // isInstanceOf 类型判断 instanceOf   // 拿学科名称 去获取一个对应的分区编号  subMap*(subject)  } } |

思考题：

如果使用treeMap来实现取topK。

# 展示量和点击量的作业题

|  |
| --- |
| **val** sc: SparkContext = *MySpark*(**this**.getClass.getSimpleName)  *// 读取数据* **val** file: RDD[String] = sc.textFile(**"f:/mrdata/impclick.txt"**) *// 数据的切分 flatMap* **val** splitRdd: RDD[((String, String), (Int, Int))] = file.flatMap(str => {  **val** split = str.split(**","**)  **val** id = split(0)  *// 关键词* **val** keywords: String = split(1)  **val** imp = split(2).toInt  **val** click = split(3).toInt   *// 对关键词组合进行切分 需要转义* **val** splitkeys: Array[String] = keywords.split(**"\\|"**)   **val** result: Array[((String, String), (Int, Int))] = splitkeys.map(keyword => {  *// 组装成 嵌套元组* ((id, keyword), (imp, click))  })  result })  *// 分组聚合* **val** result: RDD[((String, String), (Int, Int))] = splitRdd.reduceByKey((a, b) => {  **val** totalImp = a.\_1 + b.\_1  **val** totalClick = a.\_2 + b.\_2  (totalImp, totalClick) })  *// 结果展示格式的整理* result.map{  **case** ((id,keyword),(imp,click))=> (id,keyword,imp,click) }.foreach(*println*) |

# 依赖关系

## 依赖综述

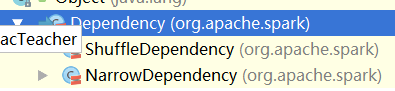
RDD之间的依赖关系。 父子RDD之间的

rdd之间的依赖关系，可以称之为lineage。（血缘 血统）

可以分为两类：

宽依赖（Wide Dependency） 窄依赖 （Narrow Dependency ）

Dependency 在源码中的关系：



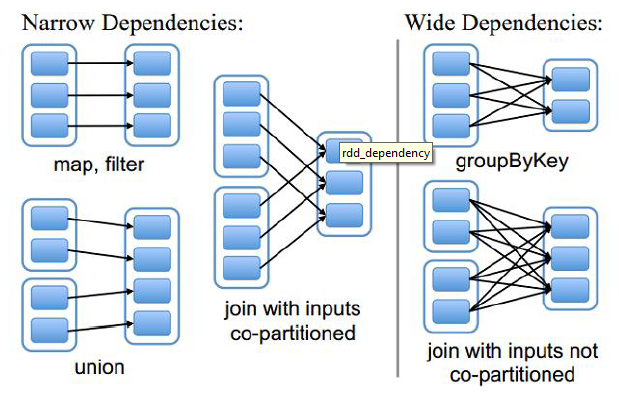
依赖产生的背景：

1. 计算任务，窄依赖 pipeline，流水线作业； 宽依赖，会进行shuffle，效率慢。
2. 失败恢复，窄依赖，失败恢复更高效； 宽依赖，效率低
3. spark Stage（阶段） 阶段的划分标准就是宽依赖。

如何理解：

顺序： 从父RDD到子RDD

**父RDD的一个分区的数据，是给子RDD的一个分区使用，还是所有的分区使用。**



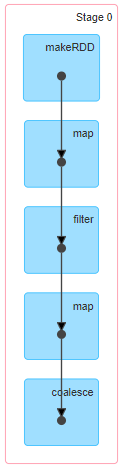
## 窄依赖

父RDD的一个分区的数据，给到子RDD的一个分区使用。

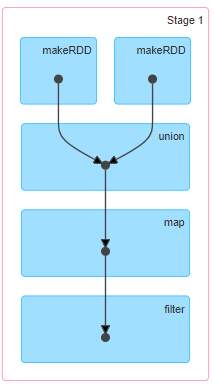
一对一的关系。

map，flatmap，filter,union,coalesce等

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,2,4,5,6,7),3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at <console>:24  scala> val rdd2 = rdd1.map(\_\*3).filter(\_>10).map((\_,1)).coalesce(1)  rdd2: org.apache.spark.rdd.RDD[(Int, Int)] = CoalescedRDD[4] at coalesce at <console>:26  scala> rdd2.partitions.size  res0: Int = 1  scala> rdd2.collect |



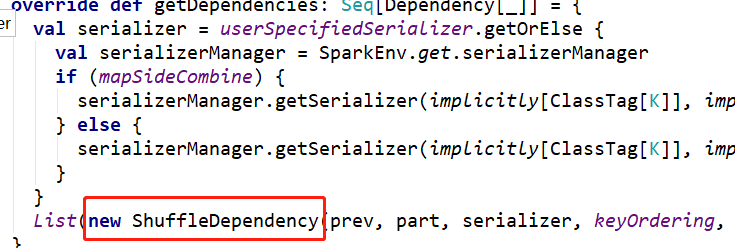
|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,2,4,5,6,7),3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at makeRDD at <console>:24  scala> val rdd2 = sc.makeRDD(List(11,13,15),3)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at makeRDD at <console>:24  scala> val rdd3 = rdd1 union rdd2  rdd3: org.apache.spark.rdd.RDD[Int] = UnionRDD[7] at union at <console>:28  scala> rdd3.partitions.size  res2: Int = 6  scala> val rdd4 = rdd3.map(\_\*3).filter(\_>20)  rdd4: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at filter at <console>:30  scala> rdd4.collect |



## 宽依赖

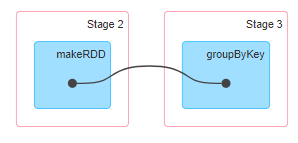
**父RDD的一个分区的数据，给到子RDD的所有分区使用。** 会有数据的shuffle。

对应着源码中的ShuffleDendency:



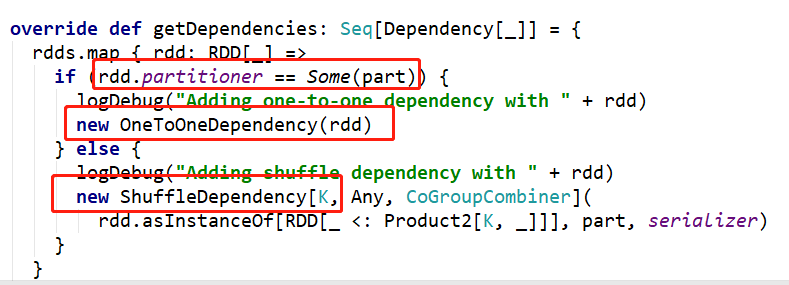
reduceBykey join distinct repartition 等都是宽依赖。

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(("shouji",4999),("shoulei",10000),("shoubiao",2000),("shoukao",2999)),3)  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[11] at makeRDD at <console>:24  scala> val rdd2 = rdd1.groupByKey()  rdd2: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[12] at groupByKey at <console>:26  scala> rdd2.collect |



### 特殊的join算子：

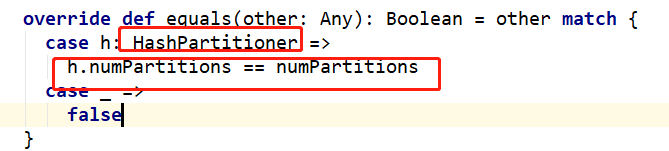
join算子在源码中获取依赖关系：



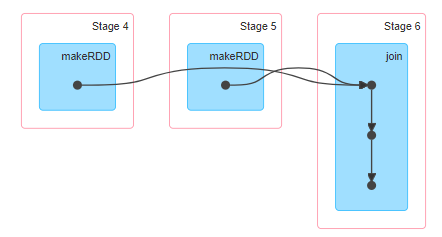
是否是窄依赖的条件：

1. 参与join的两个rdd的分区器是否是HashPartitioner
2. 如果是，再看rdd的分区数量是否一致。

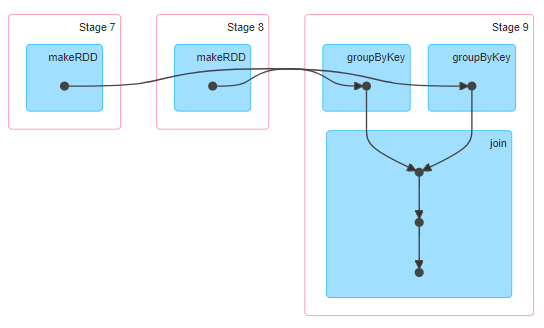
都满足，那么这个join就是窄依赖。



|  |
| --- |
| scala> val rdd2 = sc.makeRDD(List(("shouji",1000),("shoulei",10),("shoubiao",200),("shoukao",3999)),3)  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[13] at makeRDD at <console>:24  scala> val rdd3 = rdd1 join rdd2  rdd3: org.apache.spark.rdd.RDD[(String, (Int, Int))] = MapPartitionsRDD[16] at join at <console>:28  scala> rdd3.collect |



|  |
| --- |
| scala> val rdd4 = rdd1.groupByKey(3) join rdd2.groupByKey(3)  rdd4: org.apache.spark.rdd.RDD[(String, (Iterable[Int], Iterable[Int]))] = MapPartitionsRDD[21] at join at <console>:28  scala> rdd4.collect |



宽依赖，是切分stage的标准。

### 关于join算子：

1. 只要是new CogroupRDD
2. join算子得到的rdd的分区数量

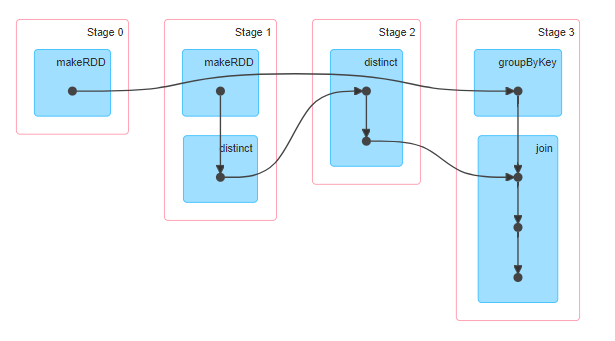
分区数量的结论：

1. 普通的rdd1和普通的rdd2 的join（没有co-partitioned）， 以分区数量多的一个rdd的分区为标准。 eg： rdd1 3 rdd2 5 join之后的rdd的分区数量 = 5
2. 如果有一个rdd1是co-partitioned,另一个是普通的rdd2, 分区数量 = rdd1的分区数量
3. 如果两个rdd都是co-partitioned ,以多的分区数量为准。（同结论1）

验证：

|  |
| --- |
| scala> val rdd2 = sc.makeRDD(List(("shouji",1000),("shoulei",10),("shoubiao",200),("shoukao",3999)),3)  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[22] at makeRDD at <console>:24  scala> val rdd1 = sc.makeRDD(List(("shouji",4999),("shoulei",10000),("shoubiao",2000),("shoukao",2999)),5)  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[23] at makeRDD at <console>:24  scala> val rdd3 = rdd2 join rdd1  rdd3: org.apache.spark.rdd.RDD[(String, (Int, Int))] = MapPartitionsRDD[26] at join at <console>:28  scala> rdd3.partitions.size  res7: Int = 5  scala> val rdd3 = rdd2.groupByKey(3) join rdd1  rdd3: org.apache.spark.rdd.RDD[(String, (Iterable[Int], Int))] = MapPartitionsRDD[30] at join at <console>:28  scala> rdd3.partitions.size  res8: Int = 3  scala> val rdd3 = rdd2.groupByKey(3) join rdd1.groupByKey(6)  rdd3: org.apache.spark.rdd.RDD[(String, (Iterable[Int], Iterable[Int]))] = MapPartitionsRDD[35] at join at <console>:28  scala> rdd3.partitions.size  res9: Int = 6 |

|  |
| --- |
| val rdd1 = sc.makeRDD(List(("shouji",4999),("shoulei",10000),("shoubiao",2000),("shoukao",2999)),3)  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at makeRDD at <console>:24  scala> val rdd2 = sc.makeRDD(List(("shouji",1000),("shoulei",10),("shoubiao",200),("shoukao",3999)),3)  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[1] at makeRDD at <console>:24  scala> val rdd4 = rdd1.distinct(3) join rdd2.groupByKey(3)  rdd4: org.apache.spark.rdd.RDD[(String, (Int, Iterable[Int]))] = MapPartitionsRDD[8] at join at <console>:28  scala> rdd4.collect |



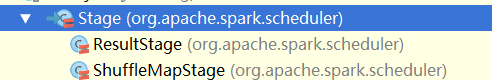
## 依赖和stage

宽依赖会切分stage，一系列的窄依赖都会在一个stage中。

Stage：阶段。

job中有2个shuffle类的算子，得到几个stage呢？ 2次 + 1

stage有两种类别： ResultStage，ShuffleMapStage



ResultStage只有一个。

ShuffleMapStage 会有多个。

一个job由 一个ResultStage + 多个ShuffleMapStage组成。

## 依赖和容错

窄依赖，错误恢复快； 宽依赖，错误恢复慢

如果是窄依赖，只需要拿父rdd的对应分区的数据，+ 函数的业务逻辑 = 新的结果数据，恢复快

宽依赖： 必须拿父RDD中的所有的分区数据 + 业务逻辑 = 新的结果数据， 恢复慢

Driver: 会记录 rdd的依赖 ，rdd的操作的函数 等

executor端挂掉了。

Driver会把运行在挂掉机器上的任务，重新分发到其他的executor中去运行。

重试的次数： 3次

只是底层的机制。

新的提升任务运行效率的算子： 缓存/持久化 checkpoint等

## DAG的生成

DAG(Directed Acyclic Graph)叫做**有向无环图**，指任意一条边有方向，且不存在环路的图。

图的构成： 点 + 关系

点： RDD

关系： Dependency

这个图，在哪一个步骤生成的？

当触发action算子的时候，DAG图就生成了。

DAG 🡪 切分stage

DAG的起始边界：

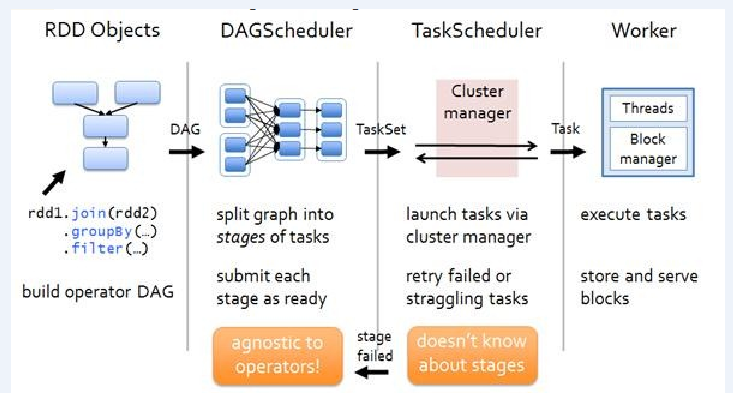
起点在哪里？ 第一个rdd

终点在哪里？ 触发action。

每一个rdd都会在其被调用的时候，重新计算。 cache

# spark任务运行的基本机制

## spark任务运行的机制图

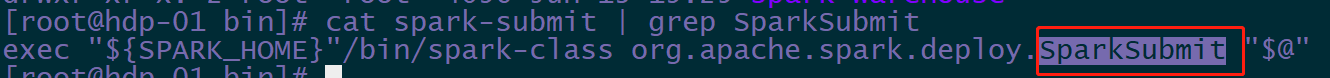


1. 构建DAG，切分stage，组装task，都是在Driver端完成的。
2. Executor仅仅负责任务的执行。

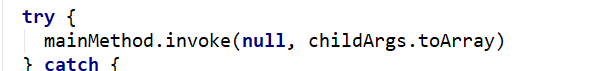
## 任务的提交：

spark-submit –master spark://hdp-01:7077 –class xx.WordCount xxx.jar input output

生成了一个SparkSubmit的进程。



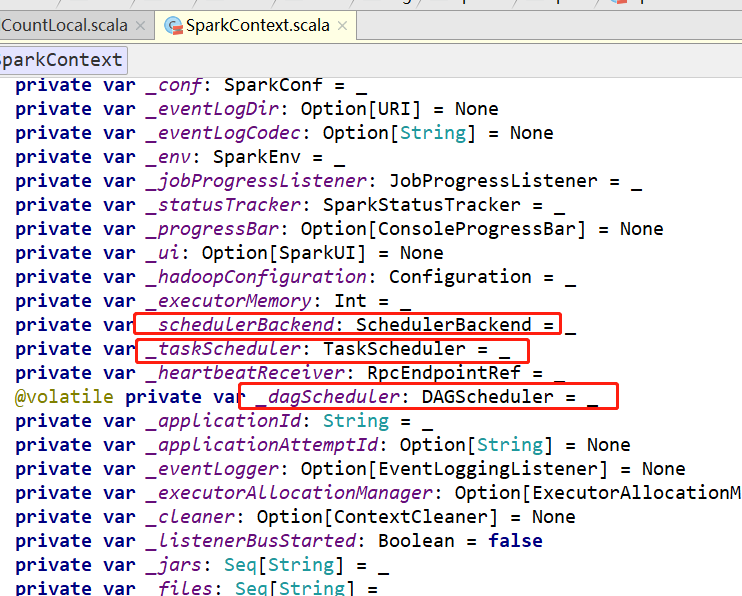
SparkSubmit的main方法中，会通过反射，调用我们自己定义的main方法所在的类。



这样，我们的自己写的业务逻辑，是否就开始被处理。

## 自己的main方法的代码被执行

在main方法中，创建了一个SparkContext的实例。



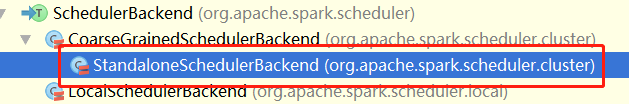
在创建SparkContext的时候，做了非常多的初始化的工作，任务执行需要的所有的实例对象，都是在这里创建好的。

最重要的3个：

DAGScheduler ： 负责切分stage，提交stage，把TaskSet提交给TaskScheduler去调度。

TaskScheduler： 负责调度task , 把task 发送给executor去执行。

SchedulerBackend： 负责通信



StandaloneSchedulerBackend: 负责和executor进行 通信

Driver会对我们的业务逻辑程序，执行解析，记录rdd之间的依赖关系，记录rdd读取数据的位置，记录算子传递的函数等等。

但是，这里并不会真正的去执行我们的任务。

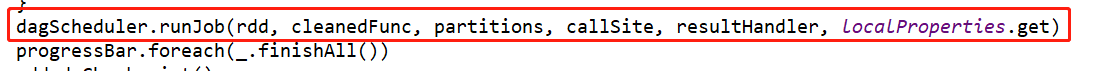
当driver解析到action算子的时候，任务才开始触发提交。

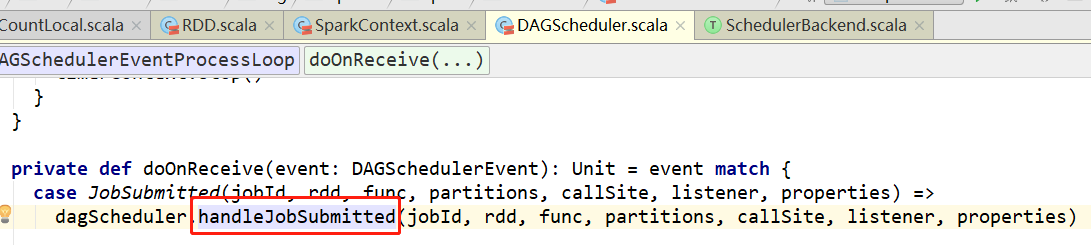
DAG图已经生成了。（点 rdd + 边 依赖关系）

## 程序解析完成，进行切分stage，组装task

当程序解析到action算子的时候，开始触发任务的提交。

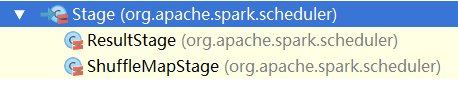
层层的调用之后，





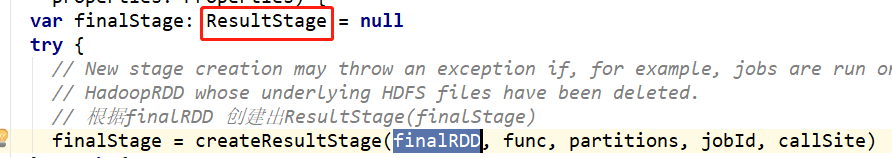
在handleJobSubmitted 方法中，开始进行stage的切分了。

## 开始进行stage的切分



首先，根据finalRDD，就是DAG图中的最后一个rdd。

创建一个finalStage：ResultStage =



从finalRDD开始，向前遍历，如果发现rdd之间的依赖关系是窄依赖，就把父rdd加入到当前的stage中；如果发现rdd之间的依赖关系是宽依赖，就进行切分stage。生成一个新的stage，继续往前遍历。

如果发现依赖关系是窄依赖，继续把父rdd加入到当前的stage中，如果发现是宽依赖，继续切分stage。

继续前面的步骤 。。。。

当发现某一个rdd没有任务的父依赖关系了，结束当前的stage。

每一个stage都是有边界的：

一个stage的起点，要么是DAG中第一个RDD，要么发生了shuffle，产生了一个新的stage。

一个stage的终点： 要么有宽依赖，进行stage的切分， 要么是一个DAG中的最后一个rdd。

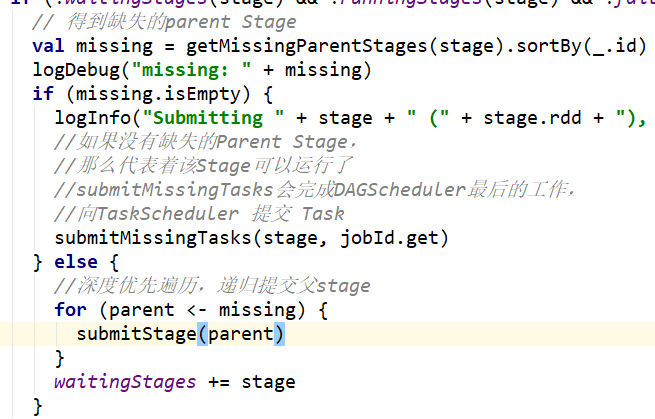
有几个stage:

宽依赖的次数 + 1

## 开始提交stage：



从finalStage开始提交，会深度优先遍历，是否还有父stage没有提交，如果有，就先提交父stage，最终，没有父依赖的stage优先提交。



开始提交某一个stage，会对这个stage进行生成task。

task： 是spark任务执行的最小的单位。

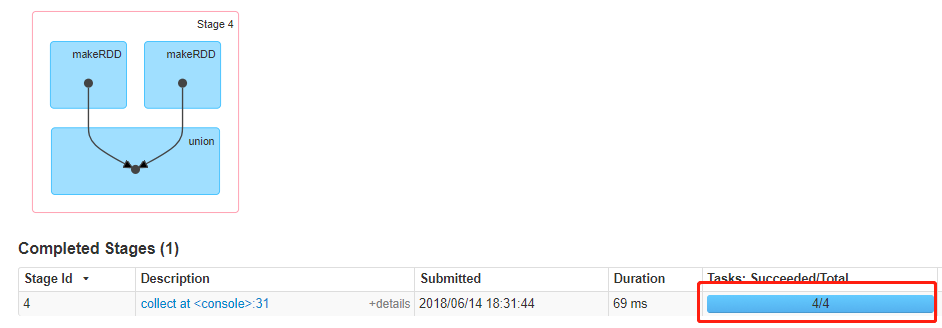
一个stage中的，一组分区的业务逻辑的综合。

一个stage中，有多少task呢？有多少分区，就有多少的task。

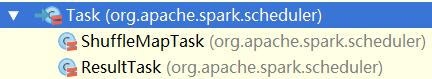
**标准： 一个stage中，最后一个rdd有几个分区，stage就有几个task。**

union：

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,5),2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[9] at makeRDD at <console>:24  scala> val rdd2 = sc.makeRDD(List(2,34,6),2)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at makeRDD at <console>:24  scala> val rdd3 = rdd1 union rdd2  rdd3: org.apache.spark.rdd.RDD[Int] = UnionRDD[11] at union at <console>:28  scala> rdd3.partitions.size  res1: Int = 4  scala> rdd3.collect |

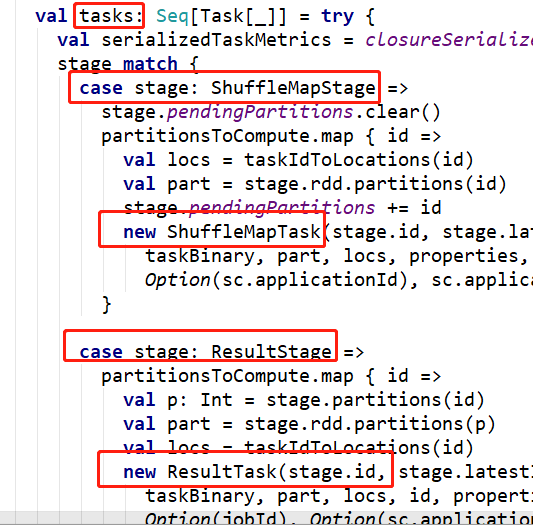


Task依然分为两类？



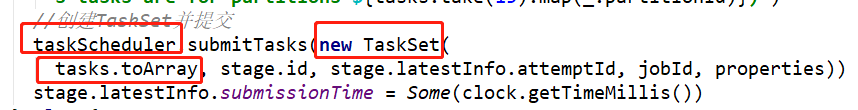
ShuffleMapStage ---🡪 ShuffleMapTask

ResultStage ----- > ResultTask



一个stage中，可能会有多个task。

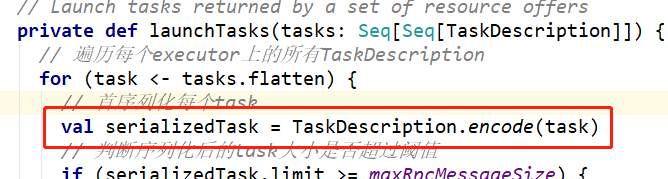
DAGScheduler中，会把一个stage的所有的task，组装成TaskSet，然后交给TaskScheduler去调度。



## TaskScheduler调度task

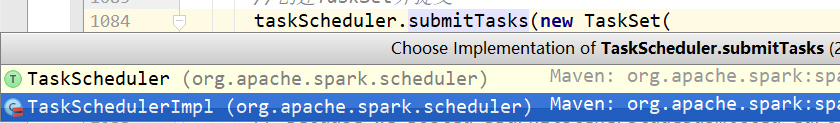
把taskSet解析，得到一个一个的task。

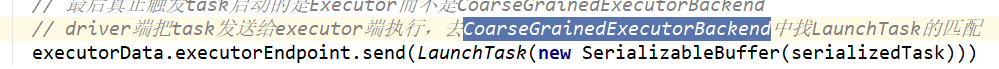
把task进行序列化，然后发送给executor去执行。



TaskScheduler 是一个特质，真正工作的是它的实现类，TaskScheudlerImpl。

ctrl + alt + b 看实现类的方法。





taskScheduler:

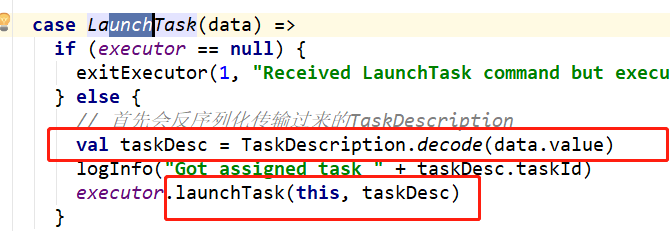
序列化task，然后提交给executor去执行。

把落后的task进行重新的提交。

## executor开始执行task

首先，接收TaskScheduler发送的task。

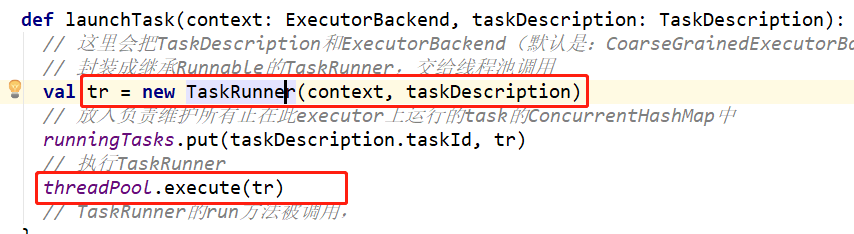
把序列化的task进行反序列化。



创建一个TaskRunner(task) , TaskRunner 是线程 ，run方法。

把taskRunner 放到线程池中去等待被调度（executor） 1cores – 处理一个task

一旦有资源，taskRunner的run方法就会被调用。



run 方法中，最后是Task的runTask方法被执行。

根据具体的task的类型，运行相应task的runTask方法。

如果是ShuffleMapTask: -🡪 runTask

ResultTask --🡪 runTask

到此为止，spark任务才开始真正的以task为单位，分布式的运行在各个executor中。

# Master和Worker的启动：

Spark 1.x 版本，底层的通信机制，使用的akka，actor

spark1.6 🡪 spark2.x akka废弃了。 netty

akka,在传递大数据的时候，会有性能问题。

akka，要求版本必须一致

规范：

Actor: -🡪 RpcEndpoint

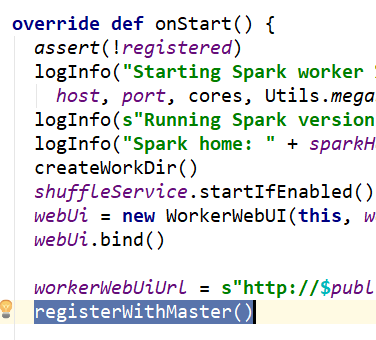
ActorSystem: --🡪 RpcEnv

ActorRef: -🡪 **RpcEndpointRef**

receive方法 ---🡪 receive

preStart方法 ---🡪 onStart方法。

在worker的onStart方法中，向Master进行注册：



# spark任务运行的4大调度

## Application

应用程序，

spark-submit 或者spark-shell，就会产生一个application。

## job

任务。

一个application 对应着一到多个job。

当调用action类的算子的时候，产生job。 一个action算子，产生一个job。

特殊的算子： sortBy,zipWithIndex, checkpoint 2个job。

## stage

阶段。

一个job，会切分成多个stage。

stage的数量 = shuffle算子 的次数 + 1

ShuffleMapStage ResultStage

## task

spark任务运行的最小单位。

一个stage中，有一到多个task。 stage中最后一个rdd的分区数量 = 当前stage中 task的数量

一个job 会有多少个task呢？

job中的 所有的stage的task数量之和。

# 几个数值

1. 读取数据的时候，数据有几个block块 ，RDD就有几个分区。
2. stage中的task的数量， 每一个stage中最后一个rdd的分区来决定的。
3. 写到hdfs中的文件的数量，ResultStage的task数量（= 最后一个rdd的分区的数量）
4. 问，一个job，能同时执行的task的数量？

理论上，task的最大的并行度， = stage的task的数量

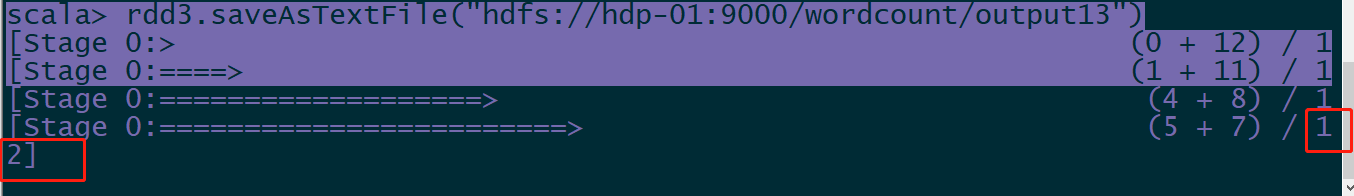
实际上，能并行执行的task的数量 = application使用的cores

某一个stage中，task 有20个，然后application cores 12个

最多，可以同时执行12个task。 剩下的task数量，什么时候有资源，才会被执行。

## 测试：

|  |
| --- |
| scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/b.txt")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/b.txt MapPartitionsRDD[1] at textFile at <console>:24  scala> rdd1.partitions.size  res0: Int = 12  scala> val rdd2 = rdd1.repartition(20)  rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at repartition at <console>:26  scala> rdd2.partitions.size  res1: Int = 20  scala> val rdd3 = rdd2.flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_)  rdd3: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[8] at reduceByKey at <console>:28  scala> rdd3.saveAsTextFile("hdfs://hdp-01:9000/wordcount/output13") |

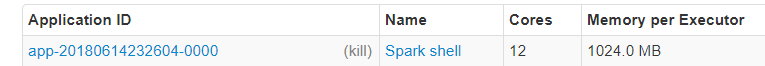


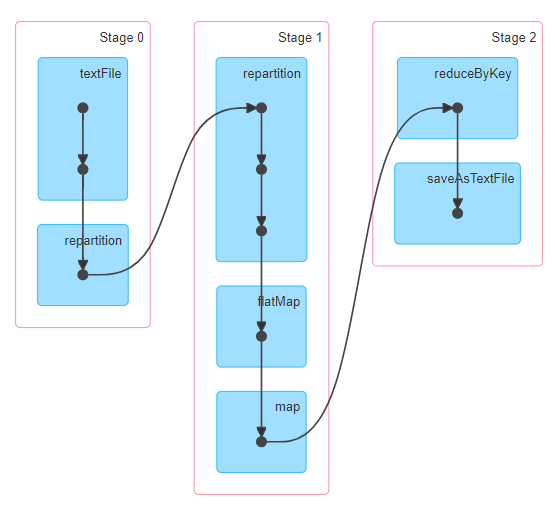
(1 + 11) / 12

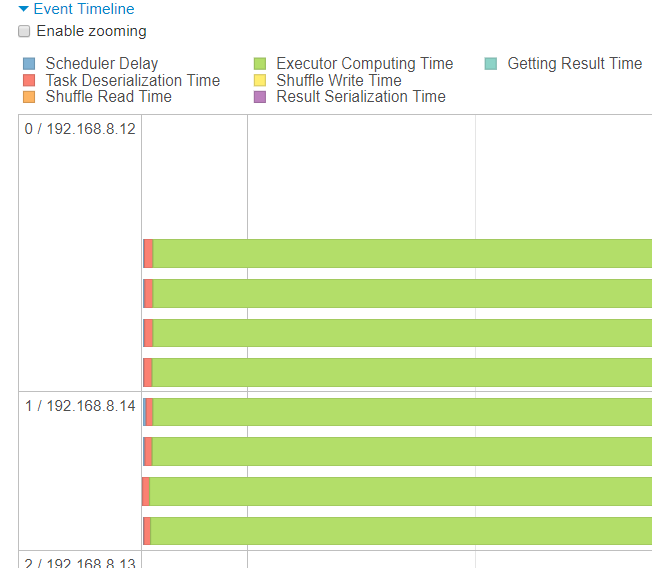
12: 表示当前的stage 的task的数量

1： 已经完成的task的数量

11： 未完成的task的数量。







# 今日重点：

宽窄依赖

spark任务的提交：

spark-submit -🡪 task

4种调度

思考题： 自定义分区器 使用 treeMap实现

作业题：

1. 根据ip地址计算归属地，统计归属地出现的次数，把结果数据写入到mysql中

2，url加密 匹配