

11. WordCount详解

11.1 源码跟踪

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD

object sparkWordCount {

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

    // 创建配置文件信息类，setAppName设置应用程序名称
    // setMaster 设置为本地测试模式
    // local[2]：本地用两个线程模拟集群运行任务
    // local：本地用一个线程模拟集群运行任务
    // local[*]：本地用所有空闲的线程模拟集群运行任务

    val conf: SparkConf = new SparkConf()
      .setAppName("SparkWC")
      .setMaster("local[4]")
    // 创建Spark上下文对象，也叫集群入口类，是spark程序的执行入口。
    val sc: SparkContext = new SparkContext(conf)

    // 读取HDFS的数据创建RDD（弹性分布式数据集）
    // val lines: RDD[String] = sc.textFile(args(0))
    val lines: RDD[String] = sc.textFile("hdfs://192.168.56.3:9000/", 1)
    /*
    第一个RDD:HadoopRDD（偏移量，内容）
    第二个RDD：MapPartitionsRDD（内容）
    sc.textFile ==>
    def textFile(
      path: String,
      minPartitions: Int = defaultMinPartitions): RDD[String] = withScope {
    assertNotStopped()
    hadoopFile(path, classOf[TextInputFormat], classOf[LongWritable], classOf[Text],
      minPartitions).map(pair => pair._2.toString).setName(path)
    }

    hadoopFile==>
    def hadoopFile[K, V](
      path: String,
      inputFormatClass: Class[_ <: InputFormat[K, V]],
      keyClass: Class[K],
      valueClass: Class[V],
      minPartitions: Int = defaultMinPartitions): RDD[(K, V)] = withScope {
    assertNotStopped()
    // A Hadoop configuration can be about 10 KB, which is pretty big, so broadcast it.
    val confBroadcast = broadcast(new SerializableConfiguration(hadoopConfiguration))
    val setInputPathsFunc = (jobConf: JobConf) => FileInputFormat.setInputPaths(jobConf, path)
```

```

    new HadoopRDD(
        this,
        confBroadcast,
        Some(setInputPathsFunc),
        inputFormatClass,
        keyClass,
        valueClass,
        minPartitions).setName(path)
}
*/
// 对数据做单词计数
//切分压平
val words: RDD[String] = lines.flatMap(_.split(" "))
/*
第三个RDD: MapPartitionsRDD (内容)
    def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {
        val cleanF = sc.clean(f)
        new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.flatMap(cleanF))
    }
*/
//生成单词和1的组合, 元组
val tuples: RDD[(String, Int)] = words.map((_, 1))
/*
第四个RDD: MapPartitionsRDD (内容)
Return a new RDD by applying a function to all elements of this RDD.
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
}
*/

//按照key进行聚合
val reduced: RDD[(String, Int)] = tuples.reduceByKey(_+_ )
/*
第五个RDD: ShuffledRDD (内容)
    def reduceByKey(func: (V, V) => V): RDD[(K, V)] = self.withScope {
        reduceByKey(defaultPartitioner(self), func)
    }
    def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)] = self.withScope {
        combineByKeyWithClassTag[V]((v: V) => v, func, func, partitioner)
    }
def combineByKeyWithClassTag[C](
    createCombiner: V => C,
    mergeValue: (C, V) => C,
    mergeCombiners: (C, C) => C,.....

*/
//排序
val res: RDD[(String, Int)] = reduced.sortBy(_._2, false)

// 打印结果
// println(res.collect.toBuffer)

// 保存

```

```

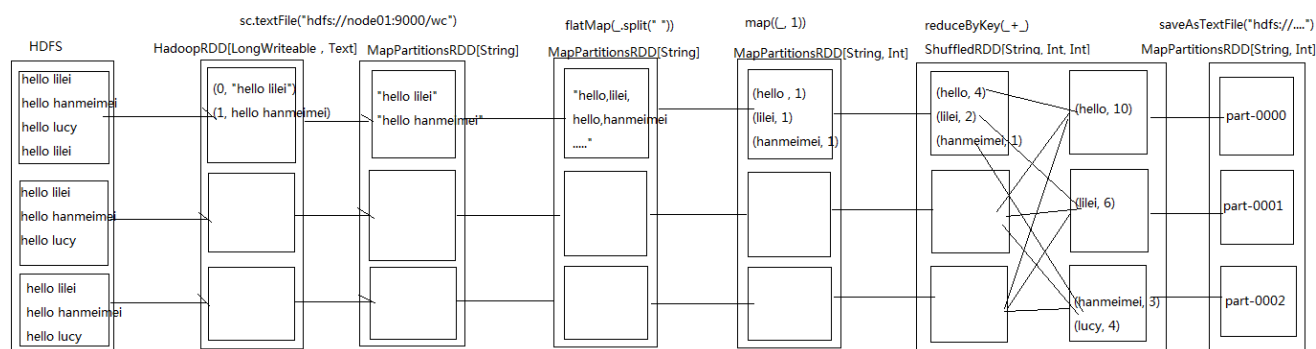
    lines.count()
    // res.saveAsTextFile(args(1))
    //释放资源
    sc.stop()

}

}

```

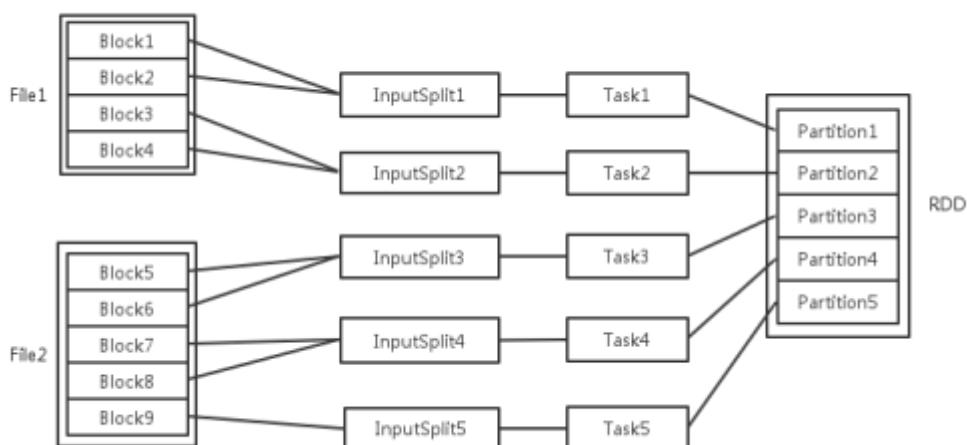
11.2.图解



11.3.分区数与任务数

注意点：

- 1，分区数和任务数是一一对应的；
- 2，shuffle过程会改变分区数；



输入可能以多个文件的形式存储在HDFS上，每个File都包含了很多块，称为Block。当Spark读取这些文件作为输入时，会根据具体数据格式对应的InputFormat进行解析，一般是将若干个Block合并成一个输入分片，称为InputSplit，注意InputSplit不能跨越文件。随后将为这些输入分片生成具体的Task。**InputSplit与Task是一一对应的关系。**随后这些具体的Task每个都会被分配到集群上的某个节点的某个Executor去执行。**每个节点可以起一个或多个Executor。每个Executor由若干core组成，每个Executor的每个core一次只能执行一个Task。每个Task执行的结果就是生成了目标RDD的一个partiton。**

注意: 这里的core是虚拟的core而不是机器的物理CPU核，可以理解为就是Executor的一个工作线程。

而 Task被执行的并发度 = Executor数目 * 每个Executor核数。

至于partition的数目：对于数据读入阶段，例如sc.textFile，输入文件被划分为多少InputSplit就会需要多少初始Task。在Map阶段partition数目保持不变。在Reduce阶段，RDD的聚合会触发shuffle操作，聚合后的RDD的partition数目跟具体操作有关，例如repartition操作会聚合成指定分区数，还有一些算子是可配置的。**RDD在计算的时候，每个分区都会起一个task，所以rdd的分区数目决定了总的task数目。** 申请的计算节点

（Executor）数目和每个计算节点核数，决定了你同一时刻可以并行执行的task。比如的RDD有100个分区，那么计算的时候就会生成100个task，你的资源配置为10个计算节点，每个两2个核，同一时刻可以并行的task数目为20，计算这个RDD就需要5个轮次。如果计算资源不变，你有101个task的话，就需要6个轮次，在最后一轮中，只有一个task在执行，其余核都在空转。如果资源不变，你的RDD只有2个分区，那么同一时刻只有2个task运行，其余18个核空转，造成资源浪费。这就是在spark调优中，增大RDD分区数目，增大任务并行度的做法。

在WordCount的代码中可以调用toDebugString方法来查看整个过程产生的RDD

11.4 webUI和RDD信息打印

注意只有在任务运行过程中可以查看；

<http://192.168.56.3:4040>

打印RDD的信息

```
println(res.toDebugString)
```

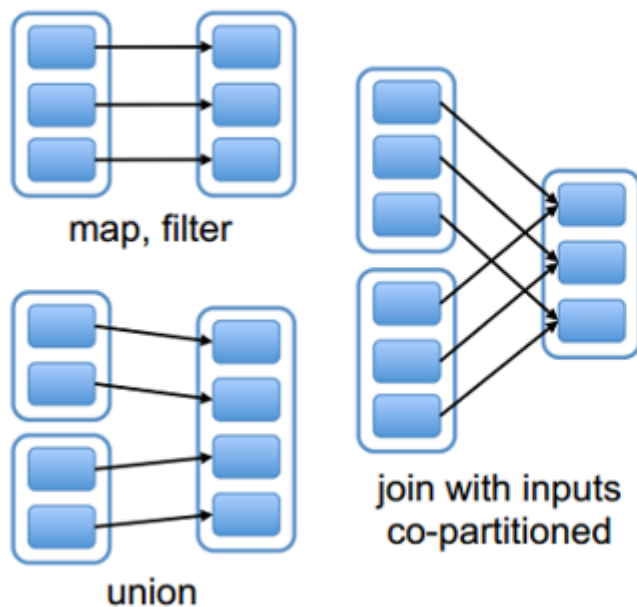
12. RDD的依赖关系

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

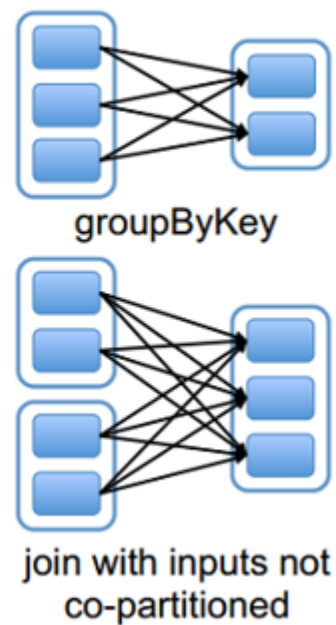
一般有shuffle过程即宽依赖，无shuffle过程就窄依赖

但是窄依赖也有可能产生数据在网络传输。

Narrow Dependencies:



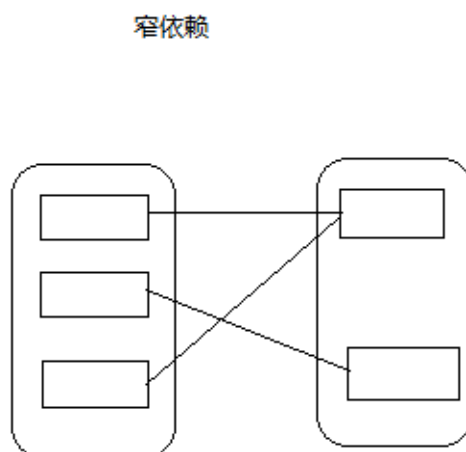
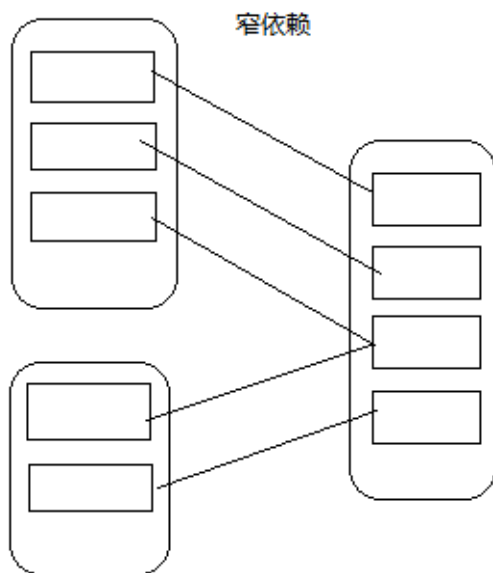
Wide Dependencies:



12.1、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
任务可以在本地执行，不需要shuffle。

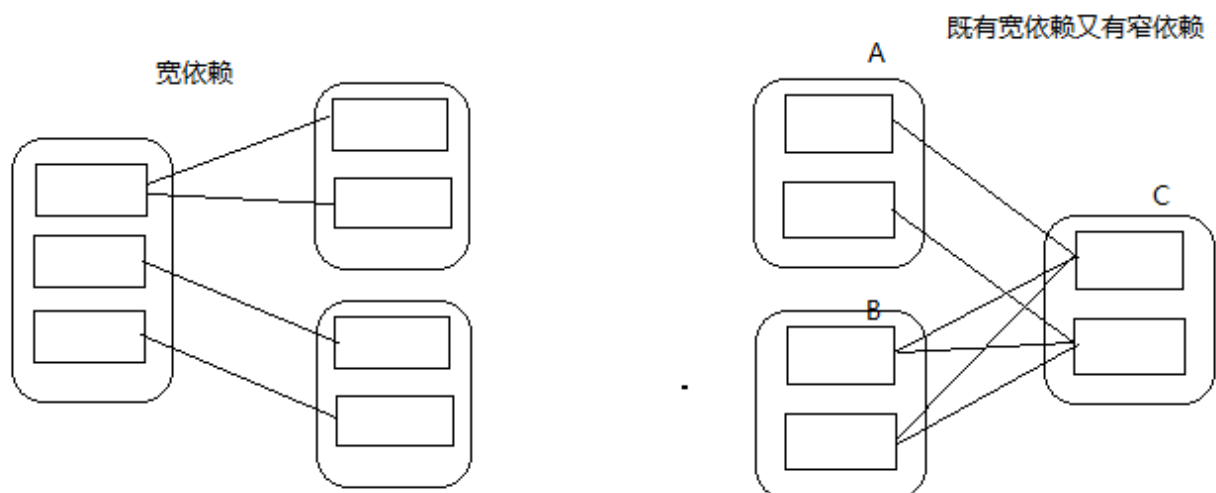
总结：窄依赖我们形象的比喻为**独生子女**



12.2、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 ;
除非父RDD是hash-partitioned, 需要shuffle。

总结：宽依赖我们形象的比喻为**超生**



12.3、常见算子类型

窄依赖：map flatmap filter union sample

宽依赖：groupByKey reduceByKey sortByKey join cartesian

12.4、源码查看

```
//RDD里记录着依赖关系
abstract class RDD[T: ClassTag](
  @transient private var _sc: SparkContext,
  @transient private var deps: Seq[Dependency[_]]
) extends Serializable with Logging {....}
//定义了
abstract class Dependency[T] extends Serializable {
  def rdd: RDD[T]
}....
NarrowDependency
ShuffleDependency
```

13. Lineage

13.1 定义

RDD只支持粗粒度转换，即在大量记录上执行的单个操作。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的操作主要分两类：转换（transformation）和动作（action）。两类函数的主要区别是，转换接受RDD并返回RDD，而动作接受RDD但是返回非RDD。转换采用惰性调用机制，每个RDD记录父RDD转换的方法，这种调用链表称之为血缘（lineage）；而动作调用会直接计算。

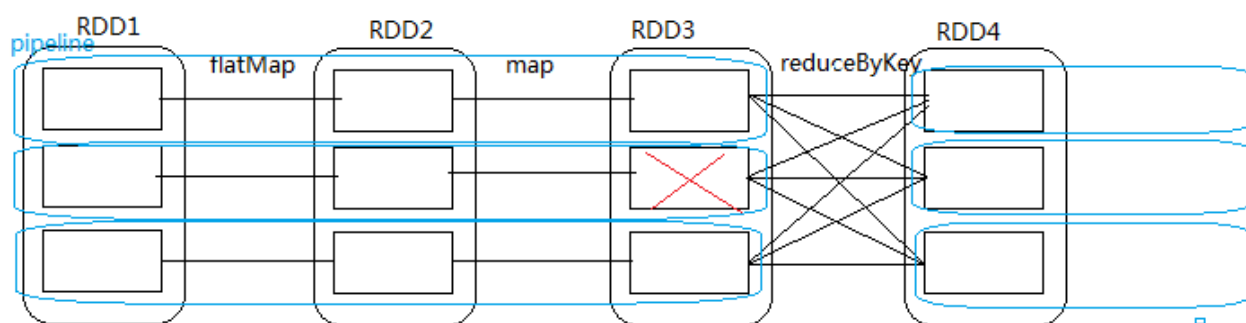
采用惰性调用，通过血缘连接的RDD操作可以管道化（pipeline），管道化的操作可以直接在单节点完成，避免多次转换操作之间数据同步的等待。

使用血缘串联的操作可以保持每次计算相对简单，而不用担心有过多的中间数据，因为这些血缘操作都管道化了，这样也保证了逻辑的单一性，而不用像MapReduce那样，为了尽可能的减少map reduce过程，在单个map reduce中写入过多复杂的逻辑。

13.2 容错

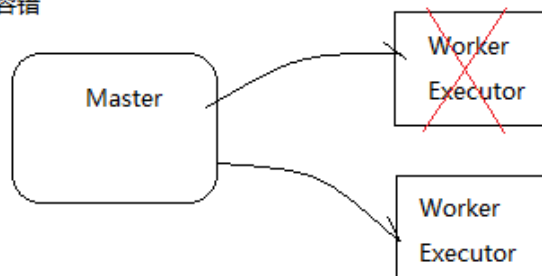
RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

RDD的容错



RDD在计算过程中，如果有一个RDD的分区的数据丢失，RDD的 lineage机制会从父RDD中恢复（重新计算）丢失的数据

集群的容错



Master除了资源分配，还会监控集群在运行任务时的情况，一旦有Worker宕机，Master会重新调度任务。如果Worker并没有宕机，只是子进程Executor出现问题，此时会由Worker重新启动一个新的Executor来计算没有完成的任务

13.3 查看

```
toDebugString()  
dependencies
```

```
scala> val rdd1 = sc.parallelize(List("tom","kate","jerry","mary","candy","willam"))
```

```
rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[13] at parallelize at
```

```

<console>:27

scala> val rdd2 = rdd1.map(_.length)
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[14] at map at <console>:29

scala> val rdd3 = rdd1.zip(rdd2)
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = ZippedPartitionsRDD2[15] at zip at <console>:31

scala> var rdd4 = rdd3.reduceByKey(_+_ )
rdd4: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[16] at reduceByKey at <console>:33

scala> rdd4.dependencies
res12: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.ShuffleDependency@3aa6a6d8)

scala> rdd4.toDebugString
res13: String =
(4) ShuffledRDD[16] at reduceByKey at <console>:33 []
+- (4) ZippedPartitionsRDD2[15] at zip at <console>:31 []
    | ParallelCollectionRDD[13] at parallelize at <console>:27 []
    | MapPartitionsRDD[14] at map at <console>:29 []
    | ParallelCollectionRDD[13] at parallelize at <console>:27 []

```

13.4 相关问题

1：从lineage看spark高效的原因：

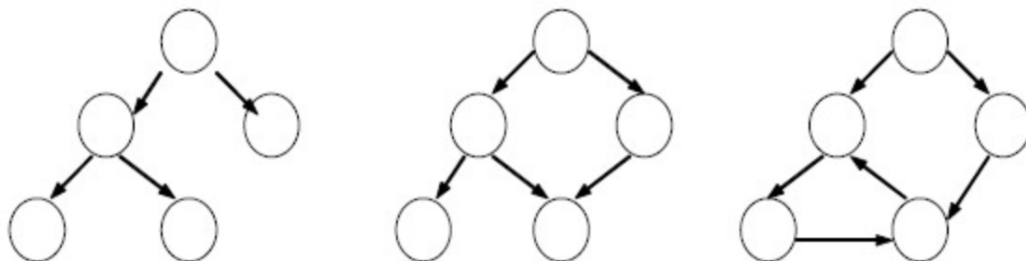
Spark官方提供的数据是RDD在某些场景下，计算效率是Hadoop的20X。这个数据是否有水分，我们先不追究，但是RDD效率高的由一定机制保证的：

- RDD数据只读，不可修改。如果需要修改数据，必须从父RDD转换（transformation）到子RDD。所以，在容错策略中，RDD没有数据冗余，而是通过RDD父子依赖（血缘）关系进行重算实现容错。
- RDD数据在内存中，多个RDD操作之间，数据不用落地到磁盘上，避免不必要的I/O操作。
- RDD存放的数据可以是java对象，所以避免的不必要的对象序列化和反序列化。
- 总而言之，RDD高效的主要因素是尽量避免不必要的操作和牺牲数据的操作精度，用来提高计算效率。

14. DAG的生成

14.1 有向无环图

DAG(Directed Acyclic Graph)叫做有向无环图，是一种不包含有向环的有向图。



DAG(Directed Acyclic Graph)叫做有向无环图

一个action触发一个（runjob方法）job，每一个job对应一个有向无环图。

14.2 DAG的创建

DAG描述多个RDD的转换过程，任务执行时，可以按照DAG的描述，执行真正的计算；

DAG是有边界的：开始（通过sparkcontext创建的RDD），结束（触发action，调用runjob就是一个完整的DAG形成了，一旦触发action，就形成了一个完整的DAG）；

一个RDD描述了数据计算过程中的一个环节，而一个DAG包含多个RDD，描述了数据计算过程中的所有环节；

一个spark application可以包含多个DAG，取决于具体有多少个action。

14.3 任务的划分

原始的RDD通过一系列的转换就形成了DAG，根据RDD之间的依赖关系的不同将DAG划分成不同的Stage，对于窄依赖，partition的转换处理在Stage中完成计算。对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完后，才能开始接下来的计算，因此宽依赖是划分Stage的依据。

RDD任务的切分

Application（应用）其实就是用spark-submit提交的程序。比方说spark examples中的计算pi的SparkPi。一个application通常包含三部分：从数据源（比方说HDFS）取数据形成RDD，通过RDD的transformation和action进行计算，将结果输出到console或者外部存储（比方说collect收集输出到console）。



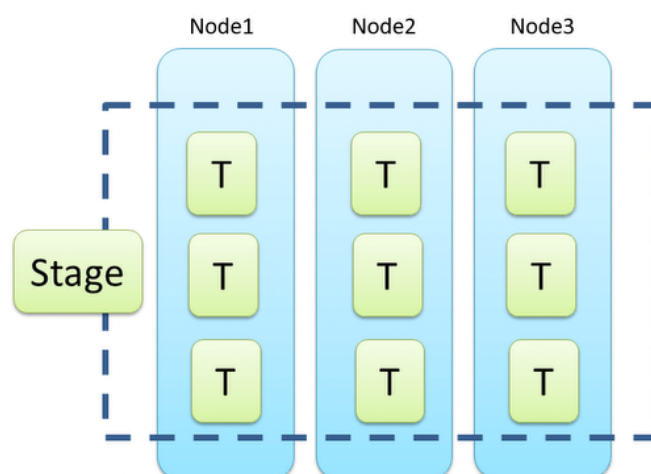
Job Spark中的Job和MR中Job不一样不一样。MR中Job主要是Map或者Reduce Job。而Spark的Job其实很好区别，一个action算子就算一个Job。



Stage概念是spark中独有的。一般而言一个Job会切换成一定数量的stage。各个stage之间按照顺序执行。至于stage是怎么切分的，首选需要知道spark中的窄依赖和宽依赖的概念。其实很好区分，看一下父RDD中的数据是否进入不同的子RDD，如果只进入到一个子RDD则是窄依赖，否则就是宽依赖。宽依赖和窄依赖的边界就是stage的划分点。

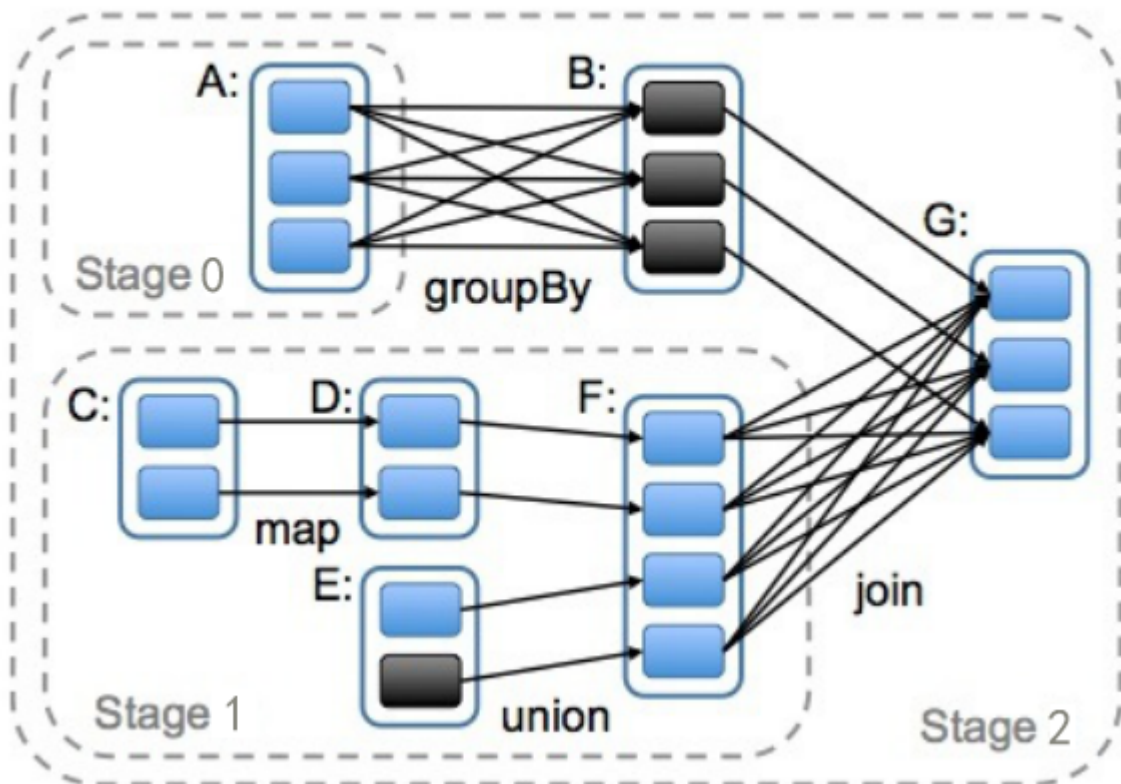


Task是Spark中最新的执行单元。RDD一般是带有partitions的，每个partition的在一个executor（一个在q特）上的执行可以任务是一个Task。



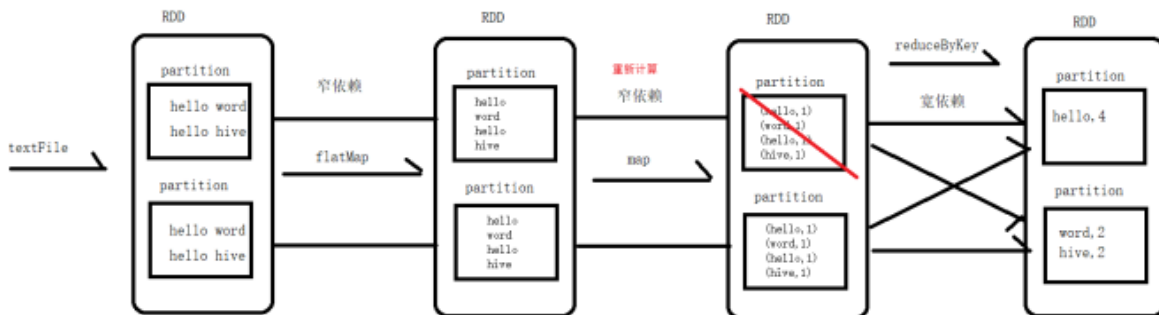
原始的RDD通过一系列的转换就形成了DAG，根据RDD之间的依赖关系的不同将DAG划分成不同的Stage

先从G开始向前知道B和F是父RDD B是窄依赖 F是宽依赖 B和A之间是窄依赖所以A划分出来 F在向前找有D E C 所以是段 G整个个是一段



对于窄依赖，partition的转换处理在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。

```
wordCount: sc.textFile().flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_)
```



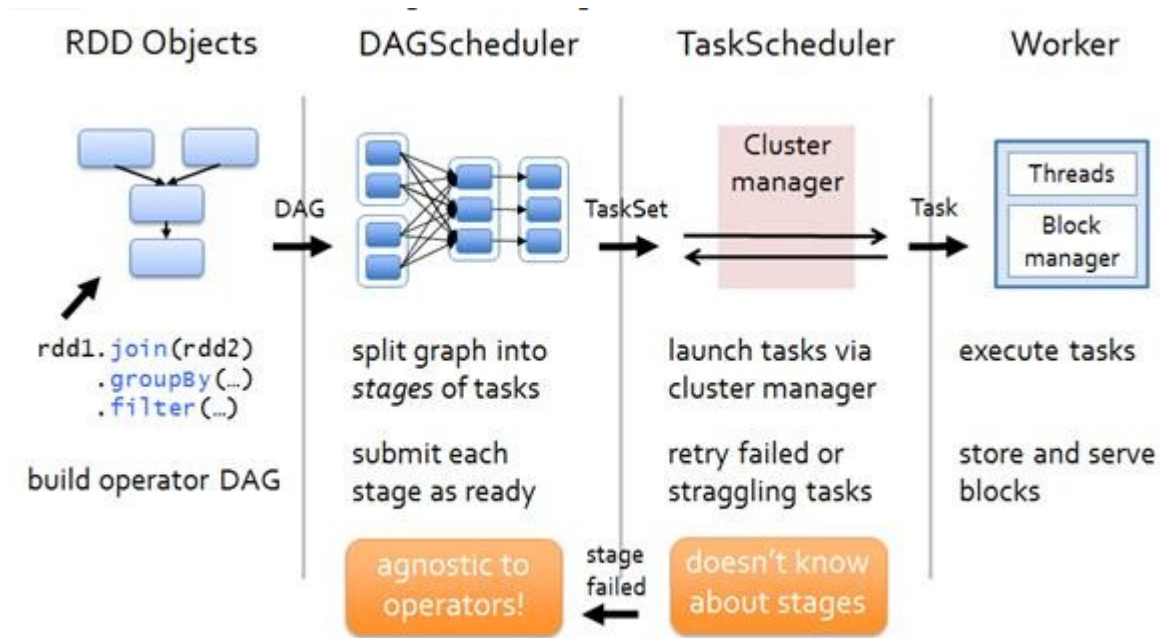
RDD在计算的过程, 如果有一个RDD的分区信息丢失, 还RDD会先判断是否进行缓存, 如果缓存了, 则直接读取缓存中的数据, 如果没有缓存, 就判断时候做过checkpoint, 如果没有做过checkpoint, 则从父RDD的分区重新开始计算, 其他分区都不用重新计算, 这样就保证容错性, 提高运行效率(Lineage)

集群:

在任务计算过程, 如果某个Executor宕掉了, 会有Worker重新启动一个新的Executor来计算完成剩余任务,

如果某个worker宕掉了, 此时Master不会重新启动这个worker, 会把宕掉worker没有完成的任务重新分配给其他worker进行计算, 这个过程和Lineage没有关, 这个是集群容错

15. 任务生成和提交的四个阶段



四个步骤：

1，构建DAG

用户提交的job将首先被转换成一系列RDD并通过RDD之间的依赖关系构建DAG,然后将DAG提交到调度系统；

DAG描述多个RDD的转换过程，任务执行时，可以按照DAG的描述，执行真正的计算；

DAG是有边界的：开始（通过sparkcontext创建的RDD），结束（触发action，调用runjob就是一个完整的DAG形成了，一旦触发action，就形成了一个完整的DAG）；

一个RDD描述了数据计算过程中的一个环节，而一个DAG包含多个RDD，描述了数据计算过程中的所有环节；

一个spark application可以包含多个DAG，取决于具体有多少个action。

2，DAGScheduler将DAG切分stage（切分依据是shuffle），将stage中生成的task以taskset的形式发送给TaskScheduler

为什么要切分stage？

一个复杂是业务逻辑（将多台机器上具有相同属性的数据聚合到一台机器上:shuffle）

如果有shuffle，那么就意味着前面阶段产生结果后，才能执行下一个阶段，下一个阶段的计算依赖上一个阶段的数据

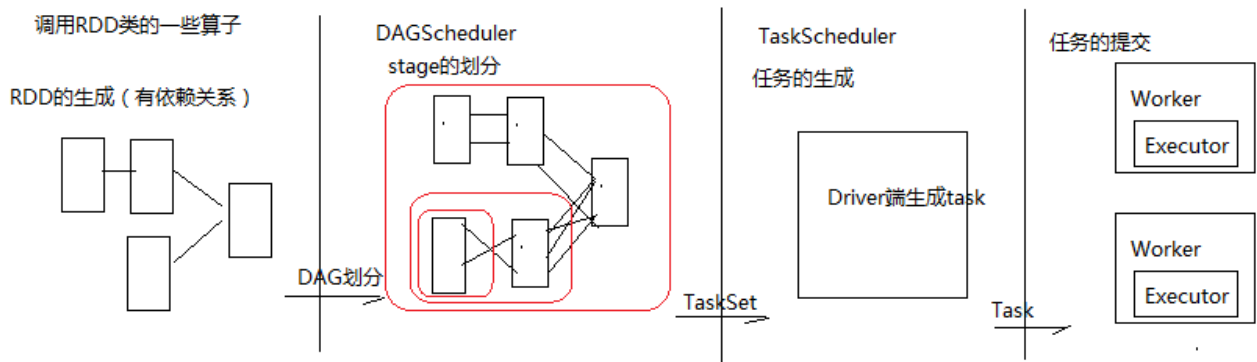
在同一个stage中，会有多个算子，可以合并到一起，我们很难”

称其为pipeline（流水线，严格按照流程、顺序执行）

3，TaskScheduler 调度task（根据资源情况将task调度到Executors）

4，Executors接收task，然后将task交给线程池执行。

RDD的生成、stage切分、task的生成、任务提交



1, stage构建:

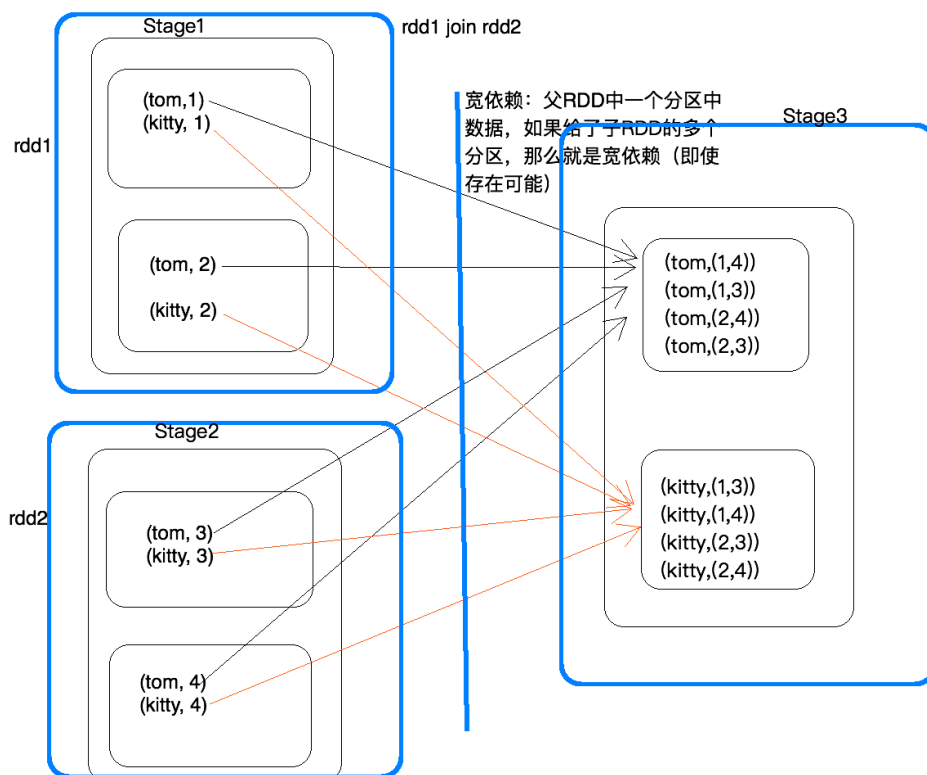
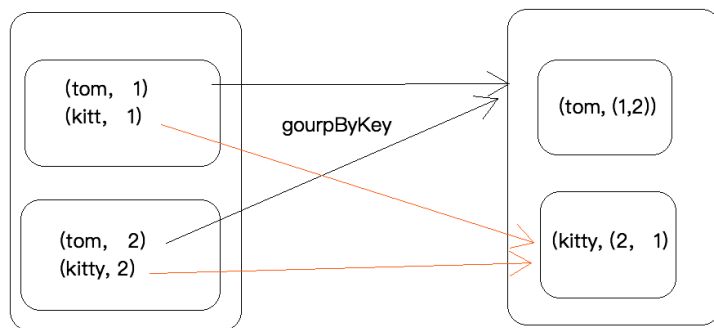
Job中所有stage的提交过程包括反向驱动与正向提交

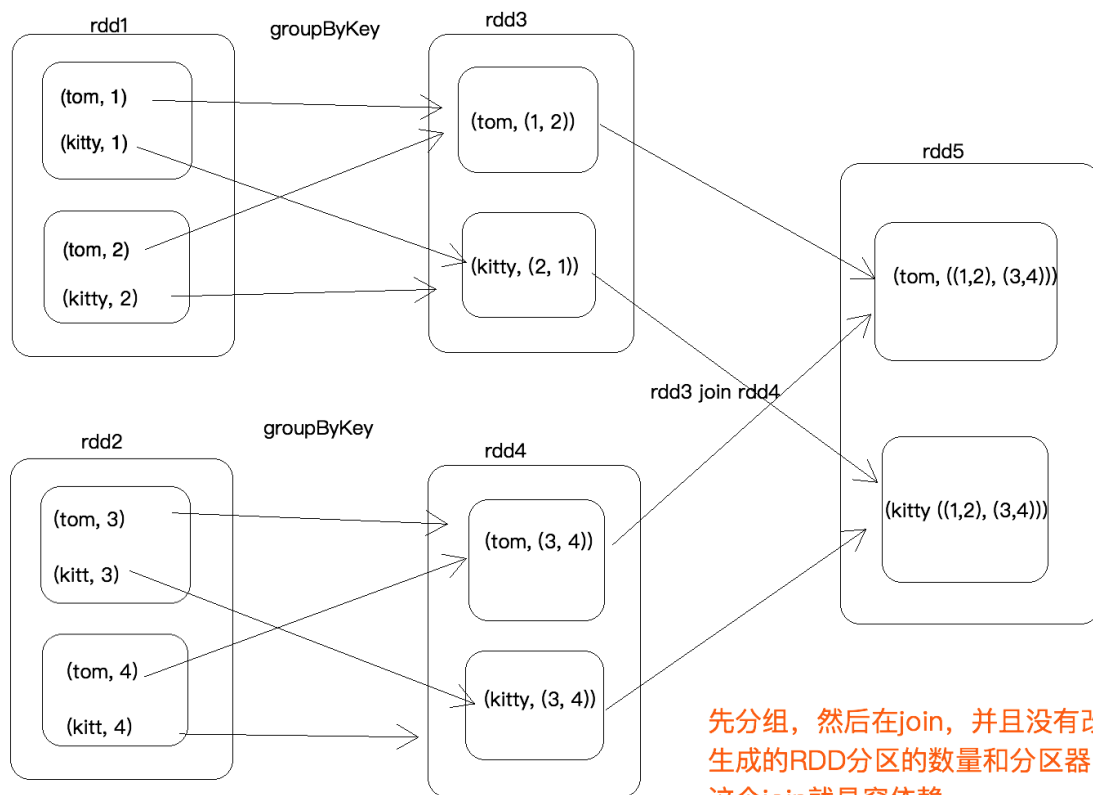
所谓反向驱动,就是从最下游的resultstage开始,由resultstage驱动所有的父stage执行,父stage又驱动它的父stage,直到最上游的stage。正向提交,就是前代stage先于后代stage将task提交给taskscheduler.

2,stage划分,有shuffle可能就发生宽依赖

```
val rdd0 = sc.parallelize(List(("tom",1),("kitty",1),("tom",2),("kitty","2")))
sc.groupByKey()
scala> def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String] = {
  |   iter.map(x => index + ":" + x)
  | }
myfunc: (index: Int, iter: Iterator[Int])Iterator[String]

scala> x.mapPartitionsWithIndex(myfunc).collect()
res6: Array[String] = Array(0:1, 0:2, 0:3, 1:4, 1:5, 1:6, 2:7, 2:8, 2:9, 2:10)
```





先分组，然后在join，并且没有改变新生成的RDD分区数量和分区器，那么这个join就是窄依赖

DAG Schedule源码分析

1. DAG Schedule源码分析，首先找到RDD，然后找到SparkContext里面的runjob这个方法，然后点击1832行的dagSchedule.runjob，然后进入到这个DAG Schedule这个类，找到611行的submitJob点进去，找到583行，这个是进行stage划分的入口，点入eventprocessLoop，在点入New DAG这个类中，这个类里面有一个很重要的方法，在1607行handleJobSubmitted，点进去这个方法，这个方法就是生成stage的具体执行步骤，然后呢，我们找到finalStage，他表示我们创建stage的最后一个stage，因为我们stage是倒序推到的。

接下来就是开始stage的源码分析：

第一步（833-837）：使用触发job的最后一个RDD，创建stage

第二步（845）：（ActiveJob这个里面就是封装的Job参数，用来表示已经激活的Job，就是被DAGScheduler接受处理的Job）用创建好的stage，创建一个Job（什么意思呢，就是说，这个Job的最后一个stage，当然就是我们的finalStage了）

第三步（856）：之前呢，这里有一个本地运行模式的判断，不过现在修改了，就没有了，直接进行Job提交了，然后这一块呢，就是说将job加入到内存缓存中。

第四步（861）：submitStage方法，提交finalStage，这一块呢，就跟之前咱们讲过的stage划分的倒推模式一样了，就是会提交最后一个stage，然后点入进去。这个方法里面做了很重要的一件事，前面的判断无关紧要，不需要关注，主要看917行，它调用了getMissingParentStages方法，去获取当前这个stage的父stage，点进去这个方法，我们看到，这个就是真正正的stage划分算法的具体实现。

（458-461）首先往栈中，推入了stage最后一个RDD，然后进行while循环，循环内部实现的是，对stage的最后一个RDD调用自己内部定义的visit方法，然后是不是就进入这个439行的方法了啊，

上面不用看，直接看重要的，444行，这一块开始遍历RDD的依赖了吧，（451）如果说是窄依赖，那么将依赖的RDD方法栈中，那么这轮循环是不是调用完了啊，那么我们下面是不是已经将第一个stage，push完了啊，那么这个时候还是窄依赖，那么他是不是继续push循环啊，又一次调用这个方法了吧，那这个时候，我们发现这个进来的RDD是宽依赖

（446），那么就开始使用宽依赖的那个RDD，创建一个

Stage，我们来看一下这个方法，这个方法呢就是创建宽依赖的Stage，他会根据这个方法将isShuffleMap设置为

true，默认最后一个stage，不是shuffleMap Stage，但是finalStage之前所有的stage，都是shuffleMap Stage，这说明如果是宽依赖，他会为他创建一个新的stage，他会将新的stage放入missing中，那么这个时候，窄依赖是不是就不会继续循环了啊，以为被push掉了，那么这个新的Stage会被返回，这块总结一下。

这个方法的意思，就是说，对一个stage，如果他的最后一个RDD的所有依赖，都是窄依赖，那么就不会创建任何新的stage，但是，只要发现这个stage的RDD是宽依赖，那么就用宽依赖的那个RDD创建一个新的stage，然后立即将这个新的stage返回。

接下来呢看另外一个方法，因为stage划分算法，是两个方法和一起完成的，（搜索handleJobSubmitted，点入submitStage，这个方法）由submitStage()方法和getMissingStages()方法共同组成

接下来的这个方法（919-926）这个方法其实是反复递归调用的，直到最初的stage，它没有了父stage了，那么，此时，就会去提交第一个stage，stage0，然后其余的stage，此时全部都在waitingstage里面，我们看一下（923），递归调用submit方法，去提交父stage，这里就是递归调用，就是stage的划分算法推动者和精髓，（926）下面这个就是将stage放入到waitingstage等待执行的stage队列中。

前面我们这个stage已经划分完成了，接下来DAG还会做一个操作，也就是说，Tasks进行划分和提交

点进921行submitMissingTasks，这个方法会提交stage，为stage创建一批task，task数量与partition数量相等，

1.（941）首先获取我们要创建的task数量（partition数量）

2.（1021）为stage创建指定数量的task，（给每一个task计算最佳位置，get每一个partition创建一个task，然后根据shuffleMapstage创建shuffleMapTask）

3.（1031）如果前面的stage不是shuffleMapStage，那么就是resultStage，那么会创建resultTask。

最佳位置算法其实说白了，就是从stage的最后一个RDD开始，去找哪一个RDD的partition是被cache了，或者checkpoint了，那么，task最佳位置，就是缓存的/checkpoint的partition的位置，因为这样的话，task就在那个节点上执行，不需要计算之前的RDD了。

16. RDD的缓存

Spark速度非常快的原因之一，就是不同操作中可以在内存中持久化或缓存多个数据集。当持久化某个RDD后，每一个节点都将把计算的分片结果保存在内存中，并在对此RDD或衍生出的RDD进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD相关的持久化和缓存，是Spark最重要的特征之一。可以说，缓存是Spark构建迭代式算法和快速交互式查询的关键。

（如果一个有持久化数据的节点发生故障，Spark 会在需要用到缓存的数据时重算丢失的数据分区。如果希望节点故障的情况不会拖累我们的执行速度，也可以把数据备份到多个节点上。）

16.1、使用缓存

RDD通过persist方法或cache方法可以将前面的计算结果缓存，但是并不是这两个方法被调用时立即缓存，而是触发后面的action时，该RDD将会被缓存在计算节点的内存中，并供后面重用。

通过调用cache()或者persist()实现：

```
scala> val rdd1 = sc.parallelize(1 to 10)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:27

scala> val rdd2 = rdd1.map(_._toString+"_"+System.currentTimeMillis)
rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at map at <console>:29

scala> val rdd3 = rdd1.map(_._toString+"_"+System.currentTimeMillis)
rdd3: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at map at <console>:29
```

```

scala> rdd2.cache
res5: rdd2.type = MapPartitionsRDD[4] at map at <console>:29

scala> rdd2.collect
res6: Array[String] = Array(1_1538115708131, 2_1538115708132, 3_1538115708133, 4_1538115708133,
5_1538115708133, 6_1538115708130, 7_1538115708130, 8_1538115708136, 9_1538115708136,
10_1538115708137)

scala> rdd2.collect
res7: Array[String] = Array(1_1538115708131, 2_1538115708132, 3_1538115708133, 4_1538115708133,
5_1538115708133, 6_1538115708130, 7_1538115708130, 8_1538115708136, 9_1538115708136,
10_1538115708137)

scala> rdd2.collect
res8: Array[String] = Array(1_1538115708131, 2_1538115708132, 3_1538115708133, 4_1538115708133,
5_1538115708133, 6_1538115708130, 7_1538115708130, 8_1538115708136, 9_1538115708136,
10_1538115708137)

scala> rdd3.collect
res9: Array[String] = Array(1_1538115718874, 2_1538115718874, 3_1538115718876, 4_1538115718876,
5_1538115718876, 6_1538115718873, 7_1538115718873, 8_1538115718875, 9_1538115718875,
10_1538115718875)

scala> rdd3.collect
res10: Array[String] = Array(1_1538115722726, 2_1538115722726, 3_1538115722725, 4_1538115722725,
5_1538115722725, 6_1538115722723, 7_1538115722723, 8_1538115722731, 9_1538115722731,
10_1538115722731)

scala> rdd3.collect
res11: Array[String] = Array(1_1538115724673, 2_1538115724673, 3_1538115724673, 4_1538115724673,
5_1538115724673, 6_1538115724669, 7_1538115724669, 8_1538115724676, 9_1538115724676,
10_1538115724676)

```

缓存提高速度示例：

```

scala> sc.textFile("hdfs://centos0:9000/names/")
res0: org.apache.spark.rdd.RDD[String] = hdfs://centos0:9000/names/ MapPartitionsRDD[1] at
textFile at <console>:28

scala> res0.count
res2: Long = 1924665

scala> res0.count
res3: Long = 1924665

scala> res0.cache
res4: res0.type = hdfs://centos0:9000/names/ MapPartitionsRDD[1] at textFile at <console>:28

scala> res0.count
res5: Long = 1924665

```




```
scala> res0.count
res6: Long = 1924665

scala> res0.count
res7: Long = 1924665
```

Completed Jobs (5)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total
4	count at <console>:30	2018/09/28 17:28:26	0.5 s	1/1
3	count at <console>:30	2018/09/28 17:28:23	0.6 s	1/1
2	count at <console>:30	2018/09/28 17:28:16	5 s	1/1
1	count at <console>:30	2018/09/28 17:27:36	2 s	1/1
0	count at <console>:30	2018/09/28 17:26:03	5 s	1/1

 1.6.3	Jobs	Stages	Storage	Environment	Executors	SQL	Spark shell application UI
---	------	--------	---------	-------------	-----------	-----	----------------------------

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
hdfs://centos0:9000/names/	Memory Deserialized 1x Replicated	138	100%	125.5 MB	0.0 B	0.0 B

16.2、比较cache和persist

cache底层调用的也是persist

cache()只能使用默认的缓存级别

persist()可以自定义缓存级别

```
/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)

/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def cache(): this.type = persist()
```

16.3、缓存级别

```
/*
useDisk : 使用硬盘 ( 外存 )
useMemory : 使用内存
useOffHeap : 使用堆外内存, 这是Java虚拟机里面的概念, 堆外内存意味着把内存对象分配在Java虚拟机的堆以外的内存, 这些内存直接受操作系统管理 ( 而不是虚拟机 )。这样做的结果就是能保持一个较小的堆, 以减少垃圾收集对应用的影响。
deserialized : 反序列化, 其逆过程序列化 ( Serialization ) 是java提供的一种机制, 将对象表示成一连串的字节; 而反序列化就表示将字节恢复为对象的过程。序列化是对象永久化的一种机制, 可以将对象及其属性保存起来, 并能在反序列化后直接恢复这个对象
replication : 备份数 ( 在多个节点上备份 )
*/
class StorageLevel private(
  private var _useDisk: Boolean,
```

```

    private var _useMemory: Boolean,
    private var _useOffHeap: Boolean,
    private var _deserialized: Boolean,
    private var _replication: Int = 1)
  extends Externalizable {

    // TODO: Also add fields for caching priority, dataset ID, and flushing.
    private def this(flags: Int, replication: Int) {
      this((flags & 8) != 0, (flags & 4) != 0, (flags & 2) != 0, (flags & 1) != 0, replication)
    }

    def this() = this(false, true, false, false) // For deserialization

    def useDisk: Boolean = _useDisk
    def useMemory: Boolean = _useMemory
    def useOffHeap: Boolean = _useOffHeap
    def deserialized: Boolean = _deserialized
    def replication: Int = _replication

    assert(replication < 40, "Replication restricted to be less than 40 for calculating hash codes")

    if (useOffHeap) {
      require(!useDisk, "Off-heap storage level does not support using disk")
      require(!useMemory, "Off-heap storage level does not support using heap memory")
      require(!deserialized, "Off-heap storage level does not support deserialized storage")
      require(replication == 1, "Off-heap storage level does not support multiple replication")
    }

    override def clone(): StorageLevel = {
      new StorageLevel(useDisk, useMemory, useOffHeap, deserialized, replication)
    }

    override def equals(other: Any): Boolean = other match {
      case s: StorageLevel =>
        s.useDisk == useDisk &&
        s.useMemory == useMemory &&
        s.useOffHeap == useOffHeap &&
        s.deserialized == deserialized &&
        s.replication == replication
      case _ =>
        false
    }

    def isValid: Boolean = (useMemory || useDisk || useOffHeap) && (replication > 0)

    def toInt: Int = {
      var ret = 0
      if (_useDisk) {
        ret |= 8
      }
      if (_useMemory) {
        ret |= 4
      }
    }
  }

```

```

    }
    if (_useOffHeap) {
        ret |= 2
    }
    if (_deserialized) {
        ret |= 1
    }
    ret
}

override def writeExternal(out: ObjectOutput): Unit = Utils.tryOrIOException {
    out.writeByte(toInt)
    out.writeByte(_replication)
}

override def readExternal(in: ObjectInput): Unit = Utils.tryOrIOException {
    val flags = in.readByte()
    _useDisk = (flags & 8) != 0
    _useMemory = (flags & 4) != 0
    _useOffHeap = (flags & 2) != 0
    _deserialized = (flags & 1) != 0
    _replication = in.readByte()
}

@throws(classOf[IOException])
private def readResolve(): Object = StorageLevel.getCachedStorageLevel(this)

override def toString: String = {
    s"StorageLevel($useDisk, $useMemory, $useOffHeap, $deserialized, $replication)"
}

override def hashCode(): Int = toInt * 41 + replication

def description: String = {
    var result = ""
    result += (if (useDisk) "Disk " else "")
    result += (if (useMemory) "Memory " else "")
    result += (if (useOffHeap) "ExternalBlockStore " else "")
    result += (if (deserialized) "Deserialized " else "Serialized ")
    result += s"${replication}x Replicated"
    result
}
}

/**
 * Various [[org.apache.spark.storage.StorageLevel]] defined and utility functions for creating
 * new storage levels.
 */
object StorageLevel {
    val NONE = new StorageLevel(false, false, false, false)
    val DISK_ONLY = new StorageLevel(true, false, false, false)
    val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)

    val MEMORY_ONLY = new StorageLevel(false, true, false, true)

```

```

val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
val OFF_HEAP = new StorageLevel(false, false, true, false)

```

级 别	使用的 空间	CPU 时间	是否在 内存中	是否在 磁盘上	备 注
MEMORY_ONLY	高	低	是	否	
MEMORY_ONLY_SER	低	高	是	否	
MEMORY_AND_DISK	高	中等	部分	部分	如果数据在内存中放不下，则溢写到磁盘上
MEMORY_AND_DISK_SER	低	高	部分	部分	如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据
DISK_ONLY	低	高	否	是	

16.4、缓存释放

cache数据已经使用完毕，后面有新的action，也有需要重复使用的RDD，这个时候最好释放已经占用的内存，使用unpersist()方法，参数决定改方法是否阻塞。

同时结合webUI确认是否释放了占用的内存。

```

/**
 * Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.
 *
 * @param blocking Whether to block until all blocks are deleted.
 * @return This RDD.
 */
def unpersist(blocking: Boolean = true): this.type = {
  logInfo("Removing RDD " + id + " from persistence list")
  sc.unpersistRDD(id, blocking)
  storageLevel = StorageLevel.NONE
  this
}

```

如果不手动删除缓存，系统如何处理 Spark 会自动监视每个节点上的缓存使用情况，并使用 least-recently-used (LRU) 的方式来丢弃旧数据分区。

17. checkpoint

Spark中对于数据的保存除了持久化操作之外，还提供了一种检查点的机制，检查点（本质是通过将RDD写入Disk做检查点）是为了通过lineage做容错的辅助，lineage过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果之后有节点出现问题而丢失分区，从做检查点的RDD开始重做Lineage，就会减少开销。检查点通过将数据写入到HDFS文件系统实现了RDD的检查点功能。

q1 比较cache 和 checkpoint

cache 和 checkpoint 是有显著区别的，缓存把 RDD 计算出来然后放在内存中，但是RDD 的依赖链（相当于数据库中的redo 日志），也不能丢掉，当某个点某个 executor 宕了，上面cache 的RDD就会丢掉，需要通过 依赖链重放计算出来，不同的是，checkpoint 是把 RDD 保存在 HDFS中，是多副本可靠存储，所以依赖链就可以丢掉了，就斩断了依赖链，是通过复制实现的高容错。

q2 检查点的选择

如果存在以下场景，则比较适合使用检查点机制：

- 1) DAG中的Lineage过长，如果重算，则开销太大（如在PageRank中）。
- 2) 在宽依赖上做Checkpoint获得的收益更大。

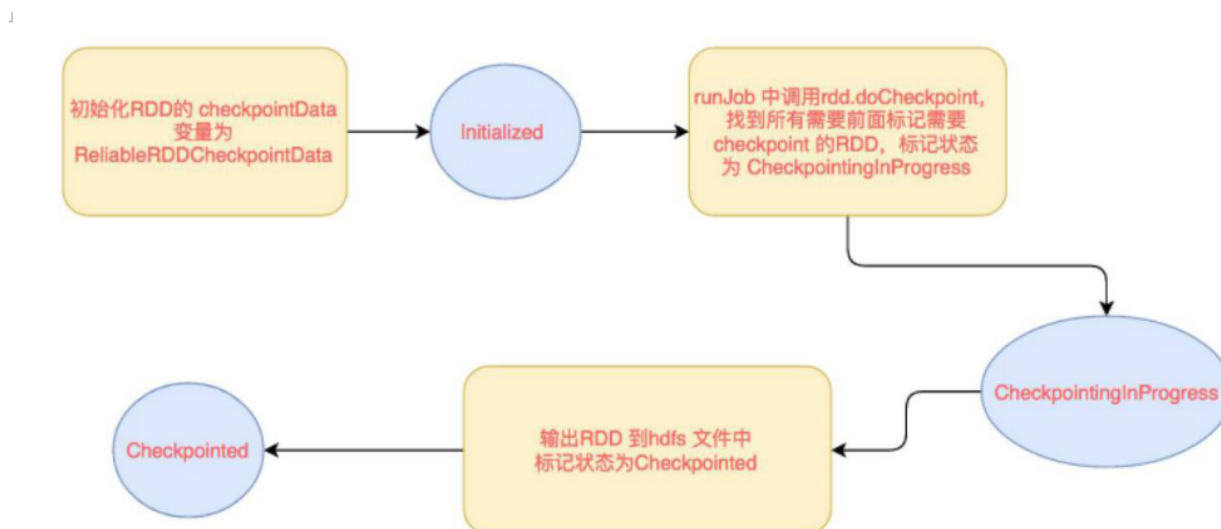
为当前RDD设置检查点。该函数将会创建一个二进制的文件，并存储到checkpoint目录中，该目录是用 [SparkContext.setCheckpointDir\(\)](#) 设置的。在checkpoint的过程中，该RDD的所有依赖于父RDD中的信息将全部被移出。对RDD进行checkpoint操作并不会马上被执行，必须执行Action操作才能触发。

17.1 checkpoint 写流程

RDD checkpoint 过程中会经过以下几个状态，

[Initialized → marked for checkpointing → checkpointing in progress → checkpointed]

转换流程如下



- 1) data.checkpoint 这个函数调用中，设置的目录中，所有依赖的 RDD 都会被删除，函数必须在 job 运行之前调用执行，强烈建议 RDD 缓存在内存中（又提到一次，千万要注意哟），否则保存到文件的时候需要从头计算。初始化RDD的 checkpointData 变量为 ReliableRDDCheckpointData。这时候标记为 Initialized 状态，

2) 在所有 job action 的时候，runJob 方法中都会调用 rdd.doCheckpoint，这个会向前递归调用所有的依赖的 RDD，看看需不需要 checkpoint。需要需要 checkpoint，然后调用 checkpointData.get.checkpoint()，里面标记状态为 CheckpointingInProgress，里面调用具体实现类的 ReliableRDDCheckpointData 的 doCheckpoint 方法，

3) doCheckpoint -> writeRDDToCheckpointDirectory，注意这里会把 job 再运行一次，如果已经 cache 了，就可以直接使用缓存中的 RDD 了，就不需要重头计算一遍了（怎么又说了一遍），这时候直接把 RDD，输出到 hdfs，每个分区一个文件，会先写到一个临时文件，如果全部输出完，进行 rename，如果输出失败，就回滚 delete。

4) 标记状态为 Checkpointed，markCheckpointed方法中清除所有的依赖，怎么清除依赖的呢，就是把 RDD 变量的强引用 设置为 null，垃圾回收了，会触发 ContextCleaner 里面监听清除实际 BlockManager 缓存中的数据

17.2 checkpoint 读流程

如果一个 RDD 我们已经 checkpoint 了那么是什么时候用呢，checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉，是一直存在的，也就是说可以被下一个 driver program 使用。比如 spark streaming 挂掉了，重启后就可以使用之前 checkpoint 的数据进行 recover，当然在同一个 driver program 也可以使用。我们讲下在同一个 driver program 中是怎么使用 checkpoint 数据的。

如果一个 RDD 被 checkpoint 了，如果这个 RDD 上有 action 操作时候，或者回溯的这个 RDD 的时候，这个 RDD 进行计算的时候，里面判断如果已经 checkpoint 过，对分区和依赖的处理都是使用的 RDD 内部的 checkpointRDD 变量。

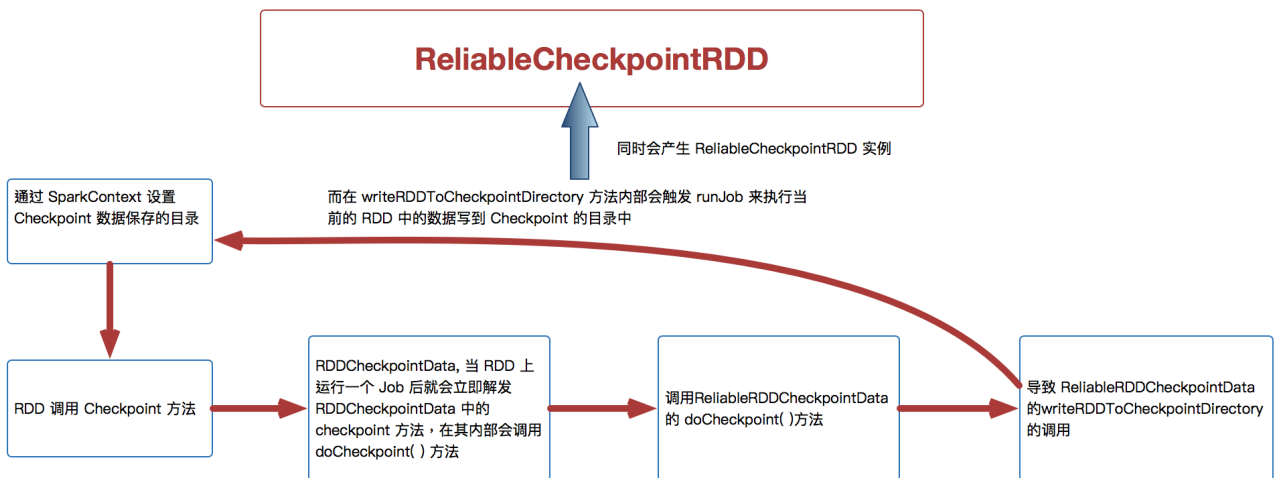
具体细节如下，

如果一个 RDD 被 checkpoint 了，那么这个 RDD 中对分区和依赖的处理都是使用的 RDD 内部的 checkpointRDD 变量，具体实现是 ReliableCheckpointRDD 类型。这个是在 checkpoint 写流程中创建的。依赖和获取分区方法中先判断是否已经 checkpoint，如果已经 checkpoint 了，就斩断依赖，使用 ReliableCheckpointRDD，来处理依赖和获取分区。如果没有，才往前回溯依赖。依赖就是没有依赖，因为已经斩断了依赖，获取分区数据就是读取 checkpoint 到 hdfs 目录中不同分区保存下来的文件。

Checkpoint 到底是什么和需要用 Checkpoint 解决什么问题：

1. 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) 或者是具体的 Transformation 产生的 **RDD 本身计算特别复杂和耗时**(例如计算时常超过 1 个小时)，可能业务比较复杂，此时我们必需考虑对计算结果的持久化。
2. Spark 是擅长**多步骤迭代**，同时擅长基于 Job 的复用。这个时候如果曾经可以对计算的过程进行复用，就可以极大的提升效率。因为有时候有共同的步骤，就可以免却重复计算的时间。
3. 如果采用 **persists** 把数据在内存中的话，虽然最快速但是也是最不可靠的；如果放在磁盘上也不是完全可靠的，例如磁盘会损坏，系统管理员可能会清空磁盘。
4. Checkpoint 的产生就是为了相对而言更加可靠的持久化数据，在 Checkpoint 可以指定把数据放在本地并且是多副本的方式，但是在正常生产环境下放在 HDFS 上，这就天然的借助 HDFS 高可靠的特征来完成最大化的**可靠的持久化数据的方式**。
5. Checkpoint 是为了**最大程度保证绝对可靠的复用 RDD** 计算数据的 Spark 的高级功能，通过 Checkpoint 我们通过把数据持久化到 HDFS 上来保证数据的最大程度的安任性
6. Checkpoint 就是针对整个 RDD 计算链条中**特别需要数据持久化的环节**(后面会反覆使用当前环节的 RDD) 开始基于 HDFS 等的**数据持久化复用策略**，通过对 RDD 启动 Checkpoint 机制来**实现容错和高可用**；

Checkpoint 运行原理图



17.3 案例练习

```
scala> sc.setCheckpointDir("hdfs://centos0:9000/ck20180929")

scala> val lines = sc.textFile("hdfs://centos0:9000/names/")
lines: org.apache.spark.rdd.RDD[String] = hdfs://centos0:9000/names/ MapPartitionsRDD[1] at
textFile at <console>:27

scala> val tuples = lines.map(item=>(item.split(",")(0),item.split(",")(2).toInt))
tuples: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[2] at map at <console>:29

scala> val reduced = tuples.reduceByKey(_+_ )
reduced: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[3] at reduceByKey at <console>:31

scala> val sorted = reduced.sortBy(_._2, false)
sorted: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[8] at sortBy at <console>:33

scala> sorted.checkpoint

scala> sorted.isCheckpointed
res2: Boolean = false

scala> sorted.take(3)
res3: Array[(String, Int)] = Array((James,5173828), (John,5137142), (Robert,4834915))

scala> sorted.isCheckpointed
res4: Boolean = true

scala> sorted.take(3)
res5: Array[(String, Int)] = Array((James,5173828), (John,5137142), (Robert,4834915))

scala> sorted.getCheckpointFile
res6: Option[String] = Some(hdfs://centos0:9000/ck20180929/6a572c8e-d27a-404a-a399-
db3bc7653255/rdd-8)
```

