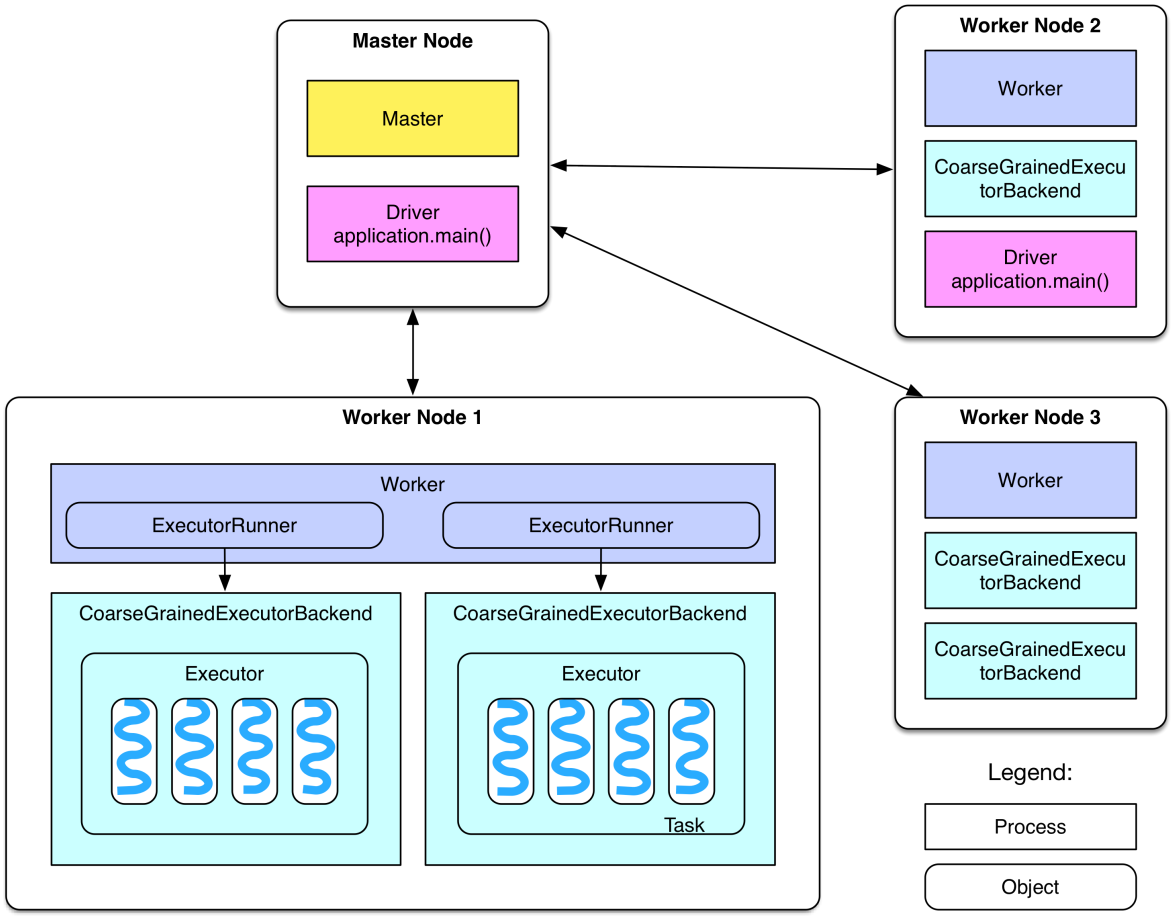
概览

拿到系统后，部署系统是第一件事，那么系统部署成功以后，**各个节点都启动了哪些服务?**

部署图



从部署图中可以看到

- 整个集群分为 Master 节点和 Worker 节点，相当于 Hadoop 的 Master 和 Slave 节点。
- Master 节点上常驻 Master 守护进程，负责管理全部的 Worker 节点。
- Worker 节点上常驻 Worker 守护进程，负责与 Master 节点通信并管理 executors。

- Driver 官方解释是 “The process running the main() function of the application and creating the SparkContext”。Application 就是用户自己写的 Spark 程序（driver program），比如 WordCount.scala。如果 driver program 在 Master 上运行，比如在 Master 上运行

  ```
  ./bin/run-example SparkPi 10
  ```

  那么 SparkPi 就是 Master 上的 Driver。如果是 YARN 集群，那么 Driver 可能被调度到 Worker 节点上运行（比如上图中的 Worker Node 2）。另外，如果直接在自己的 PC 上运行 driver program，比如在 Eclipse 中运行 driver program，使用

  ```
  val sc = new SparkContext("spark://master:7077", "AppName")
  ```

  去连接 master 的话，driver 就在自己的 PC 上，但是不推荐这样的方式，因为 PC 和 Workers 可能不在一个局域网，driver 和 executor 之间的通信会很慢。

- 每个 Worker 上存在一个或者多个 ExecutorBackend 进程。每个进程包含一个 Executor 对象，该对象持有一个线程

池，每个线程可以执行一个 task。

- 每个 application 包含一个 driver 和多个 executors，每个 executor 里面运行的 tasks 都属于同一个 application。
- 在 Standalone 版本中，ExecutorBackend 被实例化成 CoarseGrainedExecutorBackend 进程。

> 在我部署的集群中每个 Worker 只运行了一个 CoarseGrainedExecutorBackend 进程，没有发现如何配置多个 CoarseGrainedExecutorBackend 进程。（应该是运行多个 applications 的时候会产生多个进程，这个我还没有实验，）
>
> 想了解 Worker 和 Executor 的关系详情，可以参阅 @OopsOutOfMemory 同学写的 Spark Executor Driver资源调度小结 。

- Worker 通过持有 ExecutorRunner 对象来控制 CoarseGrainedExecutorBackend 的启停。

了解了部署图之后，我们先给出一个 job 的例子，然后概览一下 job 如何生成与运行。

Job 例子

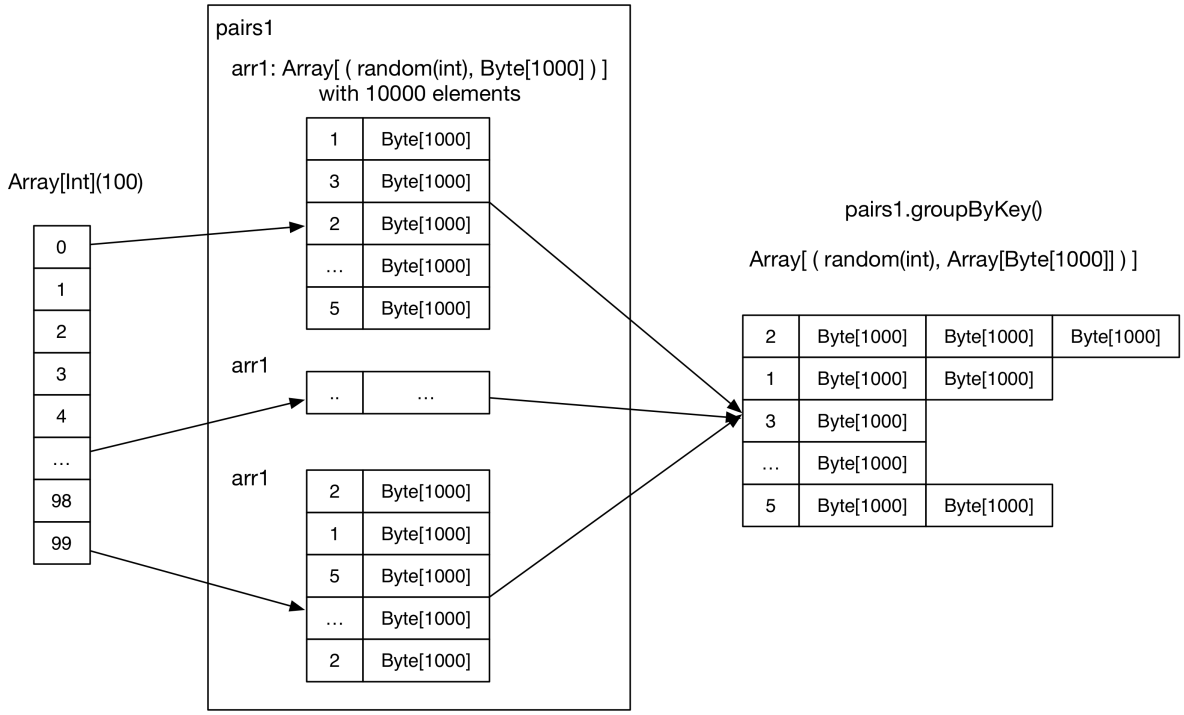
我们使用 Spark 自带的 examples 包中的 GroupByTest，假设在 Master 节点运行，命令是

```
/* Usage: GroupByTest [numMappers] [numKVPairs] [valSize] [numReducers] */

bin/run-example GroupByTest 100 10000 1000 36
```

GroupByTest 具体代码如下

```
package org.apache.spark.examples

import java.util.Random

import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._

/**
  * Usage: GroupByTest [numMappers] [numKVPairs] [valSize] [numReducers]
  */
object GroupByTest {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("GroupBy Test")
    var numMappers = 100
    var numKVPairs = 10000
    var valSize = 1000
    var numReducers = 36

    val sc = new SparkContext(sparkConf)

    val pairs1 = sc.parallelize(0 until numMappers, numMappers).flatMap { p =>
      val ranGen = new Random
      var arr1 = new Array[(Int, Array[Byte])](numKVPairs)
      for (i <- 0 until numKVPairs) {
        val byteArr = new Array[Byte](valSize)
        ranGen.nextBytes(byteArr)
        arr1(i) = (ranGen.nextInt(Int.MaxValue), byteArr)
      }
      arr1
    }.cache
    // Enforce that everything has been calculated and in cache
    pairs1.count

    println(pairs1.groupByKey(numReducers).count)

    sc.stop()
  }
}
```

阅读代码后，用户头脑中 job 的执行流程是这样的：



具体流程很简单，这里来估算下 data size 和执行结果：

1. 初始化 SparkConf()。
2. 初始化 numMappers=100, numKVPairs=10,000, valSize=1000, numReducers= 36。
3. 初始化 SparkContext。这一步很重要，是要确立 driver 的地位，里面包含创建 driver 所需的各种 actors 和 objects。
4. 每个 mapper 生成一个 `arr1: Array[(Int, Byte[])]`，length 为 numKVPairs。每一个 Byte[] 的 length 为 valSize，Int 为随机生成的整数。`Size(arr1) = numKVPairs * (4 + valSize) = 10MB`，所以 `Size(pairs1) = numMappers * Size(arr1) =1000MB`。这里的数值计算结果都是*约等于*。
5. 每个 mapper 将产生的 arr1 数组 cache 到内存。
6. 然后执行一个 action 操作 count()，来统计所有 mapper 中 arr1 中的元素个数，执行结果是 `numMappers * numKVPairs = 1,000,000`。这一步主要是为了将每个 mapper 产生的 arr1 数组 cache 到内存。
7. 在已经被 cache 的 paris1 上执行 groupByKey 操作，groupByKey 产生的 reducer （也就是 partition） 个数为 numReducers。理论上，如果 hash(Key) 比较平均的话，每个 reducer 收到的 record 个数为 `numMappers * numKVPairs / numReducer = 27,777`，大小为 `Size(pairs1) / numReducer = 27MB`。
8. reducer 将收到的 `<Int, Byte[]>` records 中拥有相同 Int 的 records 聚在一起，得到 `<Int, list(Byte[], Byte[], ..., Byte[])>`。
9. 最后 count 将所有 reducer 中 records 个数进行加和，最后结果实际就是 pairs1 中不同的 Int 总个数。

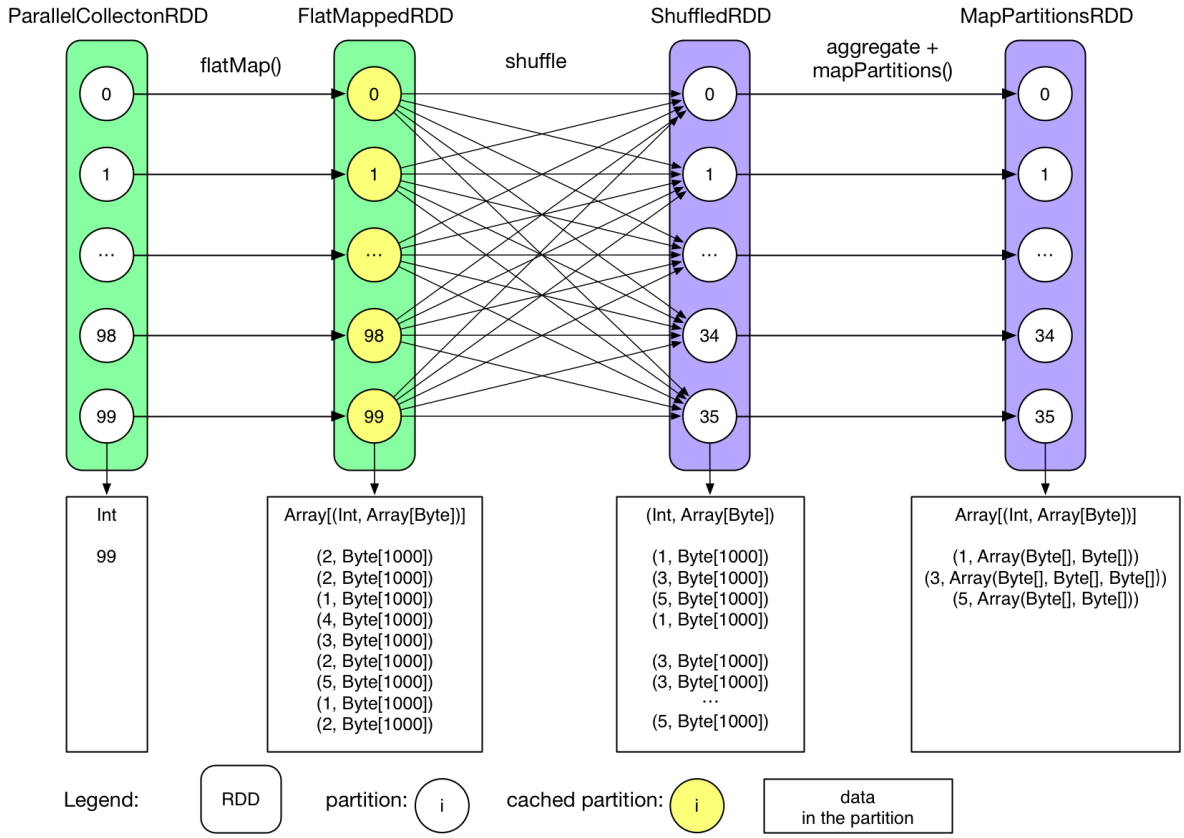
Job 逻辑执行图

Job 的实际执行流程比用户头脑中的要复杂，需要先建立逻辑执行图（或者叫数据依赖图），然后划分逻辑执行图生成 DAG 型的物理执行图，然后生成具体 task 执行。分析一下这个 job 的逻辑执行图：

使用 `RDD.toDebugString` 可以看到整个 logical plan （RDD 的数据依赖关系）如下

```
MapPartitionsRDD[3] at groupByKey at GroupByTest.scala:51 (36 partitions)
  ShuffledRDD[2] at groupByKey at GroupByTest.scala:51 (36 partitions)
    FlatMappedRDD[1] at flatMap at GroupByTest.scala:38 (100 partitions)
      ParallelCollectionRDD[0] at parallelize at GroupByTest.scala:38 (100 partitions)
```

用图表示就是：



> 需要注意的是 data in the partition 展示的是每个 partition 应该得到的计算结果，并不意味着这些结果都同时存在于内存中。

根据上面的分析可知：

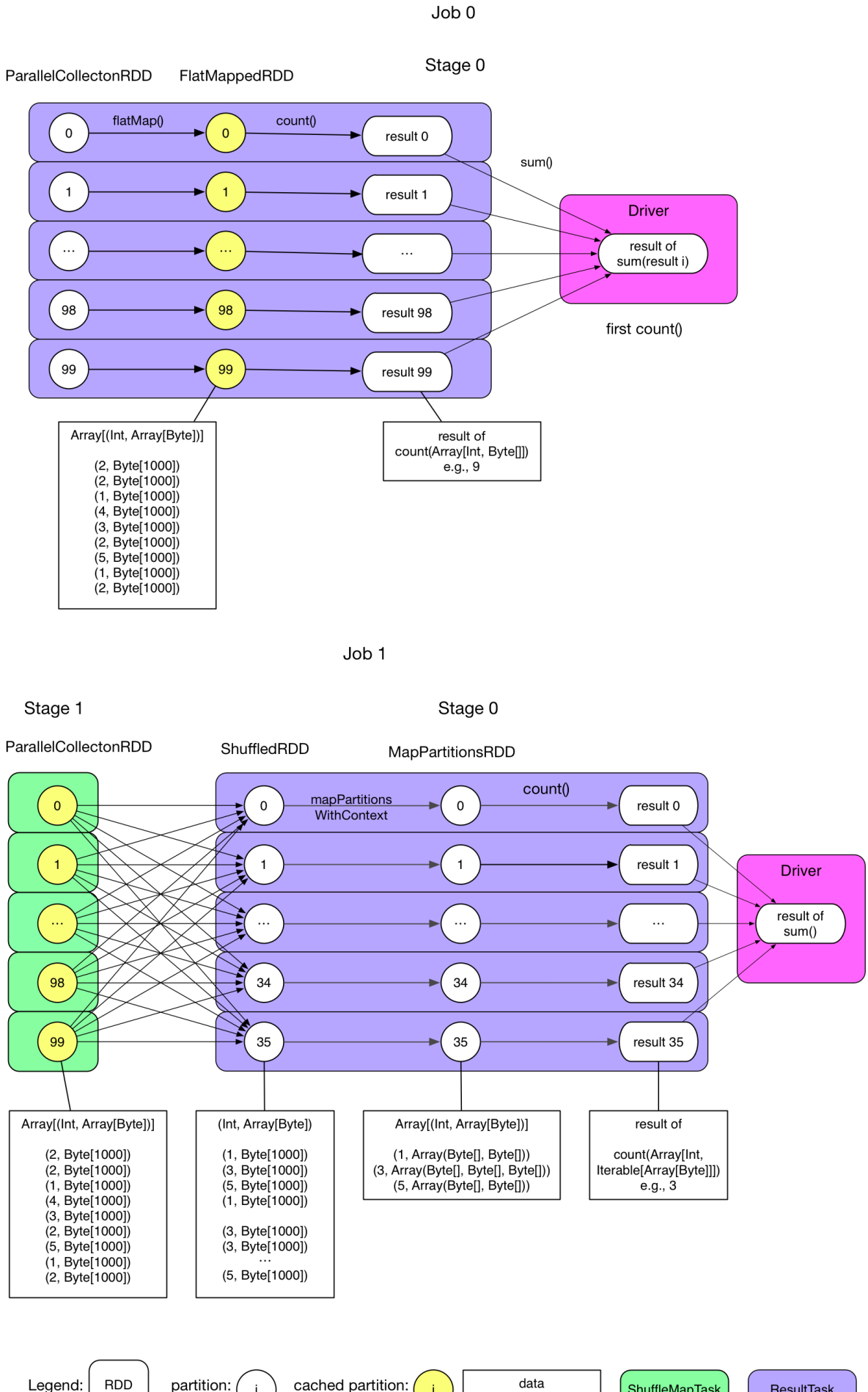
- 用户首先 init 了一个0-99 的数组： `0 until numMappers`
- parallelize() 产生最初的 ParrallelCollectionRDD，每个 partition 包含一个整数 i。
- 执行 RDD 上的 transformation 操作（这里是 flatMap）以后，生成 FlatMappedRDD，其中每个 partition 包含一个 Array[(Int, Array[Byte])]。
- 第一个 count() 执行时，先在每个 partition 上执行 count，然后执行结果被发送到 driver，最后在 driver 端进行 sum。
- 由于 FlatMappedRDD 被 cache 到内存，因此这里将里面的 partition 都换了一种颜色表示。
- groupByKey 产生了后面两个 RDD，为什么产生这两个在后面章节讨论。
- 如果 job 需要 shuffle，一般会产生 ShuffledRDD。该 RDD 与前面的 RDD 的关系类似于 Hadoop 中 mapper 输出数据与 reducer 输入数据之间的关系。
- MapPartitionsRDD 里包含 groupByKey() 的结果。
- 最后将 MapPartitionsRDD 中的 每个value（也就是Array[Byte]）都转换成 Iterable 类型。
- 最后的 count 与上一个 count 的执行方式类似。

可以看到逻辑执行图描述的是 job 的数据流：job 会经过哪些 transformation()，中间生成哪些 RDD 及 RDD 之间的依赖关系。

Job 物理执行图

逻辑执行图表示的是数据上的依赖关系，不是 task 的执行图。在 Hadoop 中，用户直接面对 task，mapper 和 reducer 的职责分明：一个进行分块处理，一个进行 aggregate。Hadoop 中 整个数据流是固定的，只需要填充 map() 和 reduce() 函数即可。Spark 面对的是更复杂的数据处理流程，数据依赖更加灵活，很难将数据流和物理 task 简单地统一在一起。因此 Spark 将数据流和具体 task 的执行流程分开，并设计算法将逻辑执行图转换成 task 物理执行图，转换算法后面的章节讨论。

针对这个 job，我们先画出它的物理执行 DAG 图如下：



可以看到 GroupByTest 这个 application 产生了两个 job，第一个 job 由第一个 action（也就是 `pairs1.count`）触发产生，分析一下第一个 job：

- 整个 job 只包含 1 个 stage。
- Stage 0 包含 100 个 ResultTask。
- 每个 task 先计算 flatMap，产生 FlatMappedRDD，然后执行 action() 也就是 count()，统计每个 partition 里 records 的个数，比如 partition 99 里面只含有 9 个 records。
- 由于 pairs1 被声明要进行 cache，因此在 task 计算得到 FlatMappedRDD 后会将其包含的 partitions 都 cache 到 executor 的内存。
- task 执行完后，driver 收集每个 task 的执行结果，然后进行 sum()。
- job 0 结束。

第二个 job 由 `pairs1.groupByKey(numReducers).count` 触发产生。分析一下该 job：

- 整个 job 包含 2 个 stage。
- Stage 1 包含 100 个 ShuffleMapTask，每个 task 负责从 cache 中读取 pairs1 的一部分数据并将其进行类似 Hadoop 中 mapper 所做的 partition，最后将 partition 结果写入本地磁盘。
- Stage 0 包含 36 个 ResultTask，每个 task 首先 shuffle 自己要处理的数据，边 fetch 数据边进行 aggregate 以及后续的 mapPartitions() 操作，最后进行 count() 计算得到 result。
- task 执行完后，driver 收集每个 task 的执行结果，然后进行 sum()。
- job 1 结束。

可以看到物理执行图并不简单。与 MapReduce 不同的是，Spark 中一个 application 可能包含多个 job，每个 job 包含多个 stage，每个 stage 包含多个 task。**怎么划分 job，怎么划分 stage，怎么划分 task 等等问题会在后面的章节介绍。**

Discussion

到这里，我们对整个系统和 job 的生成与执行有了概念，而且还探讨了 cache 等特性。 接下来的章节会讨论 job 生成与执行涉及到的系统核心功能，包括：

1. 如何生成逻辑执行图
2. 如何生成物理执行图
3. 如何提交与调度 Job
4. Task 如何生成、执行与结果处理
5. 如何进行 shuffle
6. cache机制
7. broadcast 机制