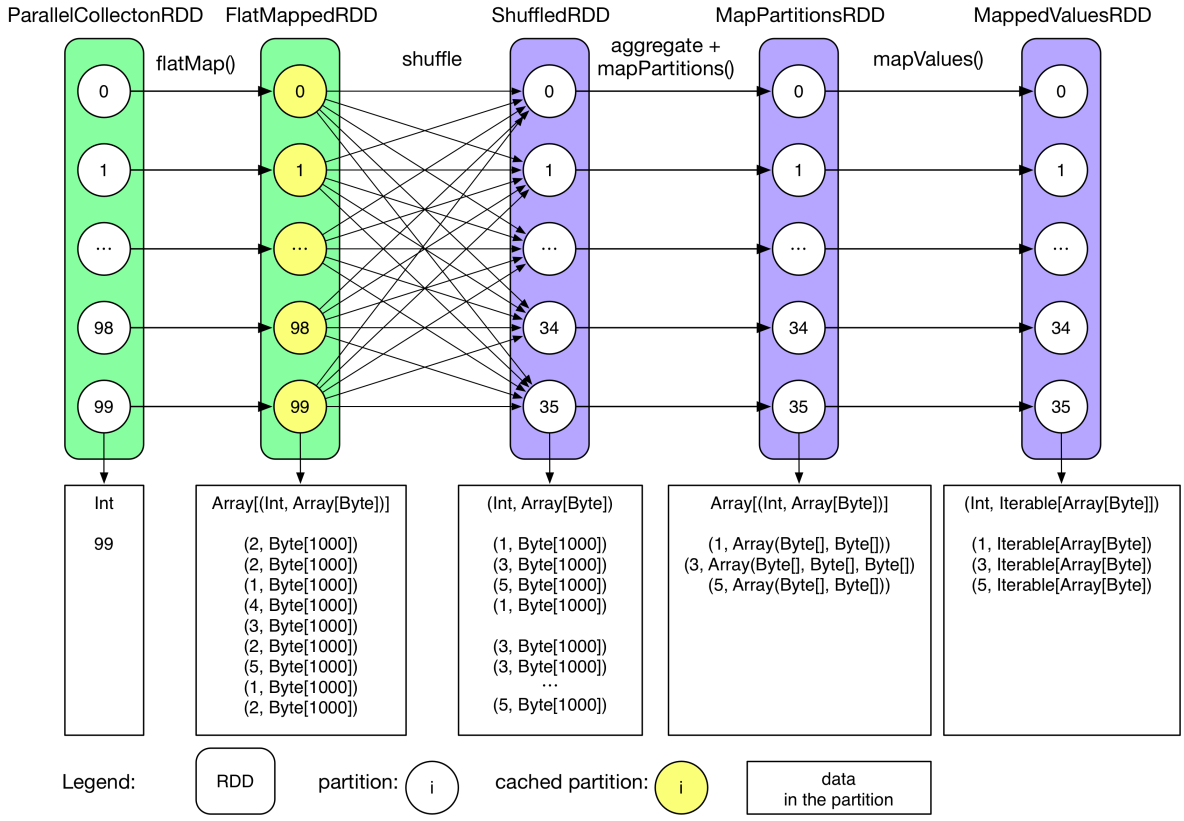
Cache 和 Checkpoint

作为区别于 Hadoop 的一个重要 feature，cache 机制保证了需要访问重复数据的应用（如迭代型算法和交互式应用）可以运行的更快。与 Hadoop MapReduce job 不同的是 Spark 的逻辑/物理执行图可能很庞大，task 中 computing chain 可能会很长，计算某些 RDD 也可能会很耗时。这时，如果 task 中途运行出错，那么 task 的整个 computing chain 需要重算，代价太高。因此，有必要将计算代价较大的 RDD checkpoint 一下，这样，当下游 RDD 计算出错时，可以直接从 checkpoint 过的 RDD 那里读取数据继续算。

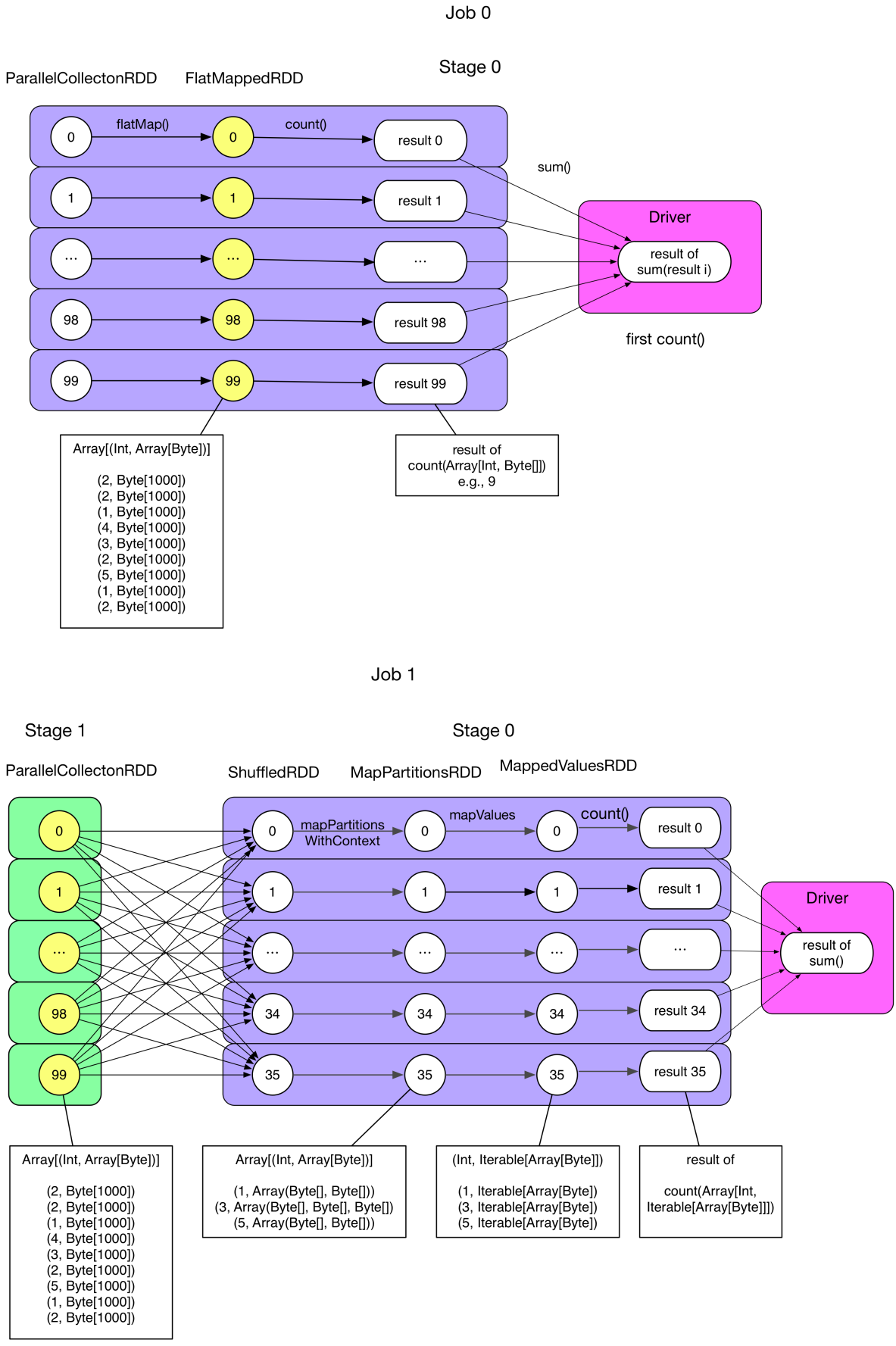
Cache 机制

回到 Overview 提到的 GroupByTest 的例子，里面对 FlatMappedRDD 进行了 cache，这样 Job 1 在执行时就直接从 FlatMappedRDD 开始算了。可见 cache 能够让重复数据在同一个 application 中的 jobs 间共享。

逻辑执行图：



物理执行图：



问题：哪些 **RDD** 需要 **cache**?

会被重复使用的（但不能太大）。

问题：用户怎么设定哪些 RDD 要 cache?

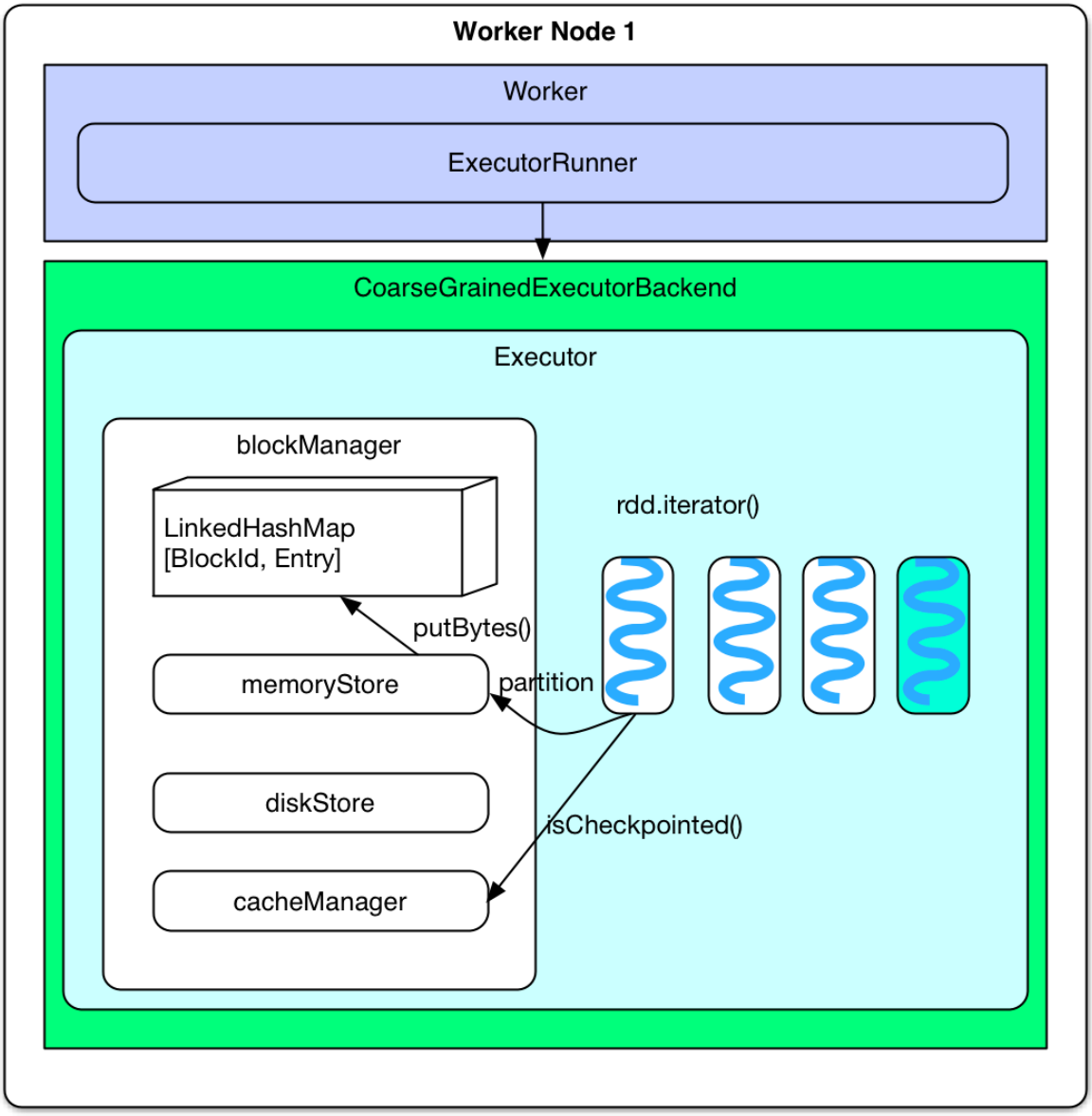
因为用户只与 driver program 打交道，因此只能用 rdd.cache() 去 cache 用户能看到的 RDD。所谓能看到指的是调用 transformation() 后生成的 RDD，而某些在 transformation() 中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，比如 reduceByKey() 中会生成的 ShuffledRDD、MapPartitionsRDD 是不能被用户直接 cache 的。

问题：driver program 设定 rdd.cache() 后，系统怎么对 RDD 进行 cache?

先不看实现，自己来想象一下如何完成 cache：当 task 计算得到 RDD 的某个 partition 的第一个 record 后，就去判断该 RDD 是否要被 cache，如果要被 cache 的话，将这个 record 及后续计算的到的 records 直接丢给本地 blockManager 的 memoryStore，如果 memoryStore 存不下就交给 diskStore 存放到磁盘。

实际实现与设想的基本类似，区别在于：将要计算 RDD partition 的时候（而不是已经计算得到第一个 record 的时候）就去判断 partition 要不要被 cache。如果要被 cache 的话，先将 partition 计算出来，然后 cache 到内存。cache 只使用 memory，写磁盘的话那就叫 checkpoint 了。

调用 rdd.cache() 后， rdd 就变成 persistRDD 了，其 StorageLevel 为 MEMORY_ONLY。persistRDD 会告知 driver 说自己是需要被 persist 的。



如果用代码表示：

```
rdd.iterator()
=> SparkEnv.get.cacheManager.getOrCompute(thisRDD, split, context, storageLevel)
=> key = RDDBlockId(rdd.id, split.index)
=> blockManager.get(key)
=> computedValues = rdd.computeOrReadCheckpoint(split, context)
```

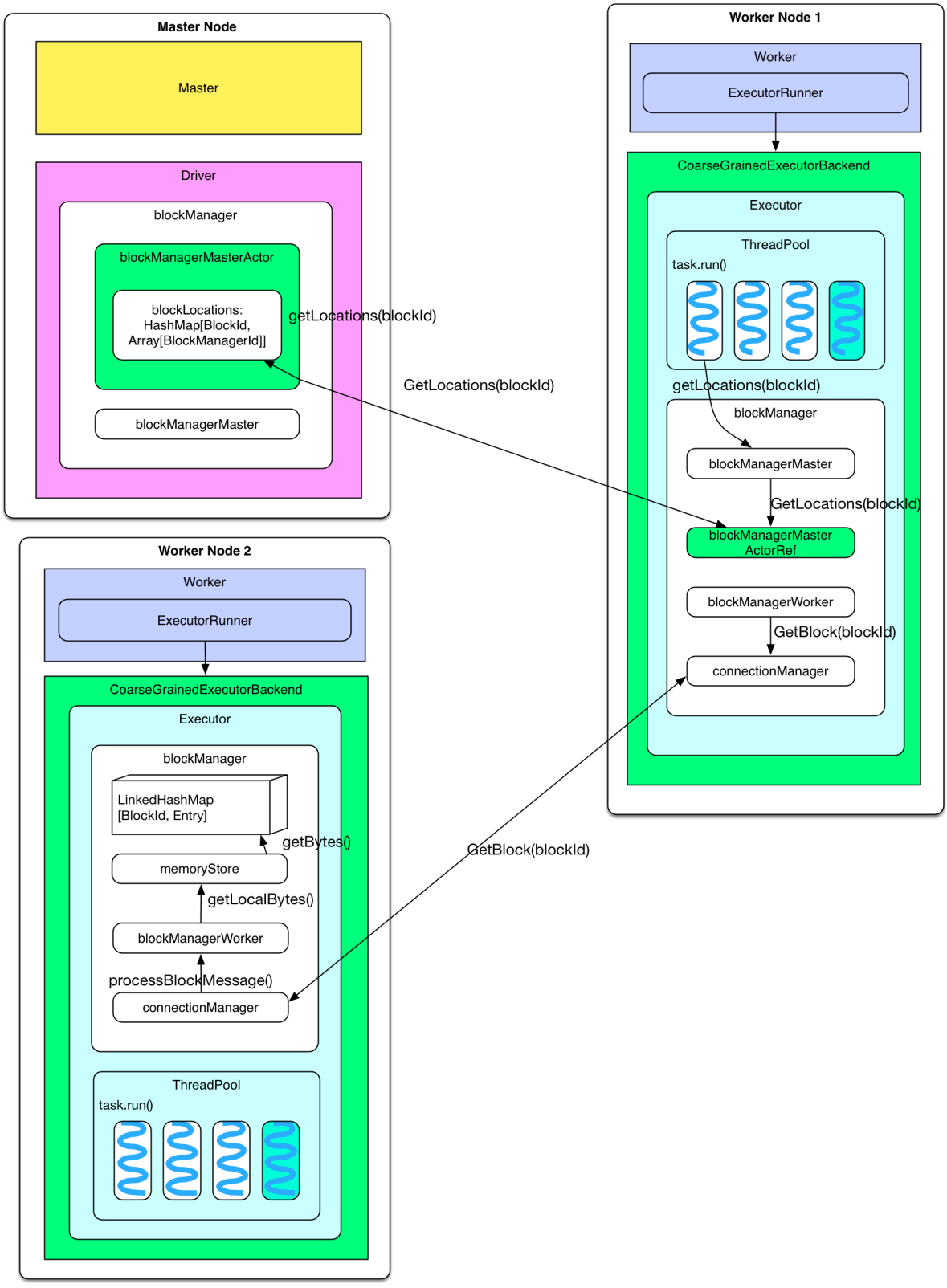
```
      if (isCheckpointed) firstParent[T].iterator(split, context)
      else compute(split, context)
=> elements = new ArrayBuffer[Any]
=> elements ++= computedValues
=> updatedBlocks = blockManager.put(key, elements, tellMaster = true)
```

当 rdd.iterator() 被调用的时候，也就是要计算该 rdd 中某个 partition 的时候，会先去 cacheManager 那里领取一个 blockId，表明是要存哪个 RDD 的哪个 partition，这个 blockId 类型是 RDDBlockId（memoryStore 里面可能还存放有 task 的 result 等数据，因此 blockId 的类型是用来区分不同的数据）。然后去 blockManager 里面查看该 partition 是不是已经被 checkpoint 了，如果是，表明以前运行过该 task，那就不用计算该 partition 了，直接从 checkpoint 中读取该 partition 的所有 records 放到叫做 elements 的 ArrayBuffer 里面。如果没有被 checkpoint 过，先将 partition 计算出来，然后将其所有 records 放到 elements 里面。最后将 elements 交给 blockManager 进行 cache。

blockManager 将 elements（也就是 partition） 存放到 memoryStore 管理的 LinkedHashMap[BlockId, Entry] 里面。如果 partition 大于 memoryStore 的存储极限（默认是 60% 的 heap），那么直接返回说存不下。如果剩余空间也许能放下，会先 drop 掉一些早先被 cached 的 RDD 的 partition，为新来的 partition 腾地方，如果腾出的地方够，就把新来的 partition 放到 LinkedHashMap 里面，腾不出就返回说存不下。注意 drop 的时候不会去 drop 与新来的 partition 同属于一个 RDD 的 partition。drop 的时候先 drop 最早被 cache 的 partition。（说好的 LRU 替换算法呢？）

问题：**cached RDD 怎么被读取?**

下次计算（一般是同一 application 的下一个 job 计算）时如果用到 cached RDD，task 会直接去 blockManager 的 memoryStore 中读取。具体地讲，当要计算某个 rdd 中的 partition 时候（通过调用 rdd.iterator()）会先去 blockManager 里面查找是否已经被 cache 了，如果 partition 被 cache 在本地，就直接使用 blockManager.getLocal() 去本地 memoryStore 里读取。如果该 partition 被其他节点上 blockManager cache 了，会通过 blockManager.getRemote() 去其他节点上读取，读取过程如下图。



获取 cached partitions 的存储位置：partition 被 cache 后所在节点上的 blockManager 会通知 driver 上的 blockMangerMasterActor 说某 rdd 的 partition 已经被我 cache 了，这个信息会存储在 blockMangerMasterActor 的 blockLocations: HashMap中。等到 task 执行需要 cached rdd 的时候，会调用 blockManagerMaster 的 getLocations(blockId) 去询问某 partition 的存储位置，这个询问信息会发到 driver 那里，driver 查询 blockLocations 获得位置信息并将信息送回。

读取其他节点上的 cached partition：task 得到 cached partition 的位置信息后，将 GetBlock(blockId) 的请求通过 connectionManager 发送到目标节点。目标节点收到请求后从本地 blockManager 那里的 memoryStore 读取 cached partition，最后发送回来。

Checkpoint

问题：哪些 **RDD** 需要 **checkpoint**?

运算时间很长或运算量太大才能得到的 RDD，computing chain 过长或依赖其他 RDD 很多的 RDD。 实际上，将 ShuffleMapTask 的输出结果存放到本地磁盘也算是 checkpoint，只不过这个 checkpoint 的主要目的是去 partition 输出数据。

问题：什么时候 **checkpoint**?

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。**也就是说需要 checkpoint 的 RDD 会被计算两次。因此，在使用 rdd.checkpoint() 的时候，建议加上 rdd.cache()**，这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。其实 Spark 提供了 rdd.persist(StorageLevel.DISK_ONLY) 这样的方法，相当于 cache 到磁盘上，这样可以做到 rdd 第一次被计算得到时就存储到磁盘上，但这个 persist 和 checkpoint 有很多不同，之后会讨论。

问题：**checkpoint** 怎么实现?

RDD 需要经过 [Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed] 这几个阶段才能被 checkpoint。

Initialized： 首先 driver program 需要使用 rdd.checkpoint() 去设定哪些 rdd 需要 checkpoint，设定后，该 rdd 就接受 RDDCheckpointData 管理。用户还要设定 checkpoint 的存储路径，一般在 HDFS 上。

marked for checkpointing：初始化后，RDDCheckpointData 会将 rdd 标记为 MarkedForCheckpoint。

checkpointing in progress：每个 job 运行结束后会调用 finalRdd.doCheckpoint()，finalRdd 会顺着 computing chain 回溯扫描，碰到要 checkpoint 的 RDD 就将其标记为 CheckpointingInProgress，然后将写磁盘（比如写 HDFS）需要的配置文件（如 core-site.xml 等）broadcast 到其他 worker 节点上的 blockManager。完成以后，启动一个 job 来完成 checkpoint（使用 `rdd.context.runJob(rdd, CheckpointRDD.writeToFile(path.toString, broadcastedConf))` ）。

checkpointed：**job 完成 checkpoint 后，将该 rdd 的 dependency 全部清掉，并设定该 rdd 状态为 checkpointed。然后，为该 rdd 强加一个依赖，设置该 rdd 的 parent rdd 为 CheckpointRDD**，该 CheckpointRDD 负责以后读取在文件系统上的 checkpoint 文件，生成该 rdd 的 partition。

有意思的是我在 driver program 里 checkpoint 了两个 rdd，结果只有一个（下面的 result）被 checkpoint 成功，pairs2 没有被 checkpoint，也不知道是 bug 还是故意只 checkpoint 下游的 RDD：

```
val data1 = Array[(Int, Char)]((1, 'a'), (2, 'b'), (3, 'c'),
    (4, 'd'), (5, 'e'), (3, 'f'), (2, 'g'), (1, 'h'))
val pairs1 = sc.parallelize(data1, 3)

val data2 = Array[(Int, Char)]((1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'))
val pairs2 = sc.parallelize(data2, 2)

pairs2.checkpoint

val result = pairs1.join(pairs2)
result.checkpoint
```

问题：怎么读取 **checkpoint** 过的 **RDD**?

在 runJob() 的时候会先调用 finalRDD 的 partitions() 来确定最后会有多个 task。rdd.partitions() 会去检查（通过 RDDCheckpointData 去检查，因为它负责管理被 checkpoint 过的 rdd）该 rdd 是会否被 checkpoint 过了，如果该 rdd 已经被 checkpoint 过了，直接返回该 rdd 的 partitions 也就是 Array[Partition]。

当调用 rdd.iterator() 去计算该 rdd 的 partition 的时候，会调用 computeOrReadCheckpoint(split: Partition) 去查看该 rdd 是否被 checkpoint 过了，如果是，就调用该 rdd 的 parent rdd 的 iterator() 也就是 CheckpointRDD.iterator()，CheckpointRDD 负责读取文件系统上的文件，生成该 rdd 的 partition。**这就解释了为什么那么 trickly 地为 checkpointed rdd 添加一个 parent CheckpointRDD**。

问题：**cache** 与 **checkpoint** 的区别?

关于这个问题，Tathagata Das 有一段回答: There is a significant difference between cache and checkpoint. Cache materializes the RDD and keeps it in memory and/or disk（其实只有 memory）. But the lineage（也就是 computing chain） of RDD (that is, seq of operations that generated the RDD) will be remembered, so that if there are node failures and parts of the cached RDDs are lost, they an be regenerated. However, **checkpoint saves the RDD to an HDFS file**

and actually forgets the lineage completely. This is allows long lineages to be truncated and the data to be saved reliably in HDFS (which is naturally fault tolerant by replication).

深入一点讨论，rdd.persist(StorageLevel.DISK_ONLY) 与 checkpoint 也有区别。前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉（**话说怎么 remove checkpoint 过的 RDD**？），是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 dirver program 使用。

Discussion

Hadoop MapReduce 在执行 job 的时候，不停地做持久化，每个 task 运行结束做一次，每个 job 运行结束做一次（写到 HDFS）。在 task 运行过程中也不停地在内存和磁盘间 swap 来 swap 去。 可是讽刺的是，Hadoop 中的 task 太傻，中途出错需要完全重新运行，比如 shuffle 了一半的数据存放到了磁盘，下次重新运行时仍然要重新 shuffle。Spark 好的一点在于尽量不去持久化，所以使用 pipeline，cache 等机制。用户如果感觉 job 可能会出错可以手动去 checkpoint 一些 critical 的 RDD，job 如果出错，下次运行时直接从 checkpoint 中读取数据。唯一不足的是，checkpoint 需要两次运行 job。

Example

貌似还没有发现官方给出的 checkpoint 的例子，这里我写了一个：

```
package internals

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object groupByKeyTest {

   def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("GroupByKey").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setCheckpointDir("/Users/xulijie/Documents/data/checkpoint")

    val data = Array[(Int, Char)]((1, 'a'), (2, 'b'),
                                  (3, 'c'), (4, 'd'),
                                  (5, 'e'), (3, 'f'),
                                  (2, 'g'), (1, 'h')
                                 )
    val pairs = sc.parallelize(data, 3)

    pairs.checkpoint
    pairs.count

    val result = pairs.groupByKey(2)

    result.foreachWith(i => i)((x, i) => println("[PartitionIndex " + i + "] " + x))

    println(result.toDebugString)
   }
}
```