

# Spark内存管理详解（下）——内存管理



作者 LeonLu (/u/5b15278387a0) + 关注

2017.02.24 23:14\* 字数 3293 阅读 162 评论 2 喜欢 5

(/u/5b15278387a0)



本文最初由IBM developerWorks (<https://www.ibm.com/developerworks/cn>)中国网站发表，其链接为Apache Spark内存管理详解 (<https://www.ibm.com/developerworks/cn/analytics/library/ba-cn-apache-spark-memory-management/index.html>)  
在这里，正文内容分为上下两篇来阐述，上一篇见《Spark内存管理详解（上）——内存分配》(<http://www.jianshu.com/p/3981b14df6b>)

Spark内存管理详解（上）——内存分配

1. 堆内和堆外内存
2. 内存空间分配

Spark内存管理详解（下）——内存管理

3. 存储内存管理
4. 执行内存管理

## 3. 存储内存管理

### 3.1 RDD的持久化机制

弹性分布式数据集（RDD）作为Spark最根本的数据抽象，是只读的分区记录

（Partition）的集合，只能基于在稳定物理存储中的数据集上创建，或者在其他已有的RDD上执行转换（Transformation）操作产生一个新的RDD。转换后的RDD与原始的RDD之间产生的依赖关系，构成了血统（Lineage）。凭借血统，Spark保证了每一个RDD都可以被重新恢复。但RDD的所有转换都是惰性的，即只有当一个返回结果给Driver的行动（Action）发生时，Spark才会创建任务读取RDD，然后真正触发转换的执行。

Task在启动之初读取一个分区时，会先判断这个分区是否已经被持久化，如果没有则需要检查Checkpoint或按照血统重新计算。所以如果一个RDD上要执行多次行动，可以在第一次行动中使用persist或cache方法，在内存或磁盘中持久化或缓存这个RDD，从而在后面的行动时提升计算速度。事实上，cache方法是使用默认的MEMORY\_ONLY的存储级别将RDD持久化到内存，故缓存是一种特殊的持久化。**堆内和堆外存储内存的设计，便可以对缓存RDD时使用的内存做统一的规划和管理**（存储内存的其他应用场景，如缓存broadcast数据，暂时不在本文的讨论范围之内）。

RDD的持久化由Spark的Storage模块<sup>[1]</sup>负责，实现了RDD与物理存储的解耦合。

Storage模块负责管理Spark在计算过程中产生的数据，将那些在内存或磁盘、在本地或远程存取数据的功能封装了起来。在具体实现时Driver端和Executor端的Storage模块构成了主从式的架构，即Driver端的BlockManager为Master，Executor端的BlockManager为Slave。Storage模块在逻辑上以Block为基本存储单位，RDD的每个Partition经过处理



后唯一对应一个Block（BlockId的格式为 rdd\_RDD-ID\_PARTITION-ID）。Master负责整个Spark应用程序的Block的元数据信息的管理和维护，而Slave需要将Block的更新等状态上报到Master，同时接收Master的命令，例如新增或删除一个RDD。

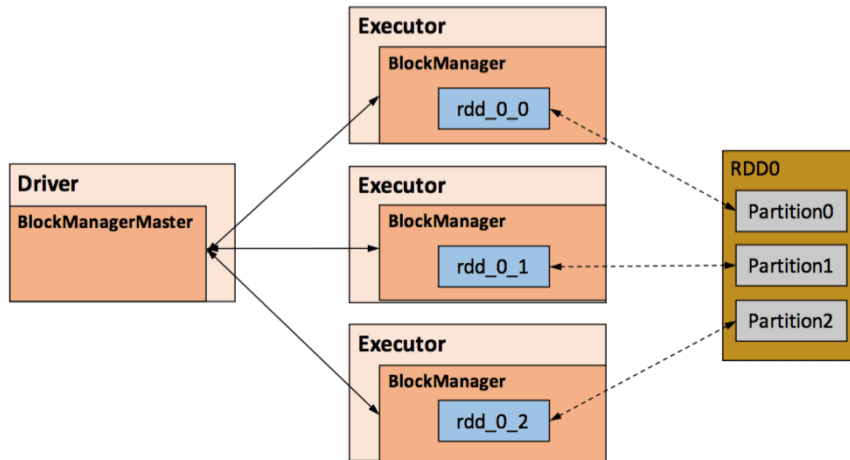


图1 Storage模块示意图

在对RDD持久化时，Spark规定了MEMORY\_ONLY、MEMORY\_AND\_DISK等7种不同的存储级别 (<http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence>)，而存储级别是以下5个变量的组合<sup>[2]</sup>：

```
class StorageLevel private {
  private var _useDisk: Boolean, //磁盘
  private var _useMemory: Boolean, //这里其实是指堆内存
  private var _useOffHeap: Boolean, //堆外内存
  private var _deserialized: Boolean, //是否为非序列化
  private var _replication: Int = 1 //副本个数
}
```

通过对数据结构的分析，可以看出存储级别从三个维度定义了RDD的Partition（同时也是Block）的存储方式：

- 存储位置：磁盘 / 堆内存 / 堆外内存。如MEMORY\_AND\_DISK是同时在磁盘和堆内存上存储，实现了冗余备份。OFF\_HEAP则是只在堆外内存存储，目前选择堆外内存时不能同时存储到其他位置。
- 存储形式：Block缓存到存储内存后，是否为非序列化的形式。如MEMORY\_ONLY是非序列化方式存储，OFF\_HEAP是序列化方式存储。
- 副本数量：大于1时需要远程冗余备份到其他节点。如DISK\_ONLY\_2需要远程备份1个副本。

### 3.2 RDD缓存的过程

RDD在缓存到存储内存之前，Partition中的数据一般以迭代器（Iterator ([http://www.scala-lang.org/docu/files/collections-api/collections\\_43.html](http://www.scala-lang.org/docu/files/collections-api/collections_43.html))）的数据结构来访问，这是Scala语言中一种遍历数据集的方法。通过Iterator可以获取分区中每一条序列化或者非序列化的数据项(Record)，这些Record的对象实例在逻辑上占用了JVM堆内存的other部分的空间，同一Partition的不同Record的空间并不连续。

RDD在缓存到存储内存之后，Partition被转换成Block，Record在堆内或堆外存储内存中占用一块连续的空间。将Partition由不连续的存储空间转换为连续存储空间的过程，Spark称之为“展开”（Unroll）。Block有序列化和非序列化两种存储格式，具体以哪种方式取决于该RDD的存储级别。非序列化的Block以一种DeserializedMemoryEntry的数据结构定义，用一个数组存储所有的Java对象，序列化的Block则以SerializedMemoryEntry的数据结构定义，用字节缓冲区（ByteBuffer）来存储二进制数



据。每个Executor的Storage模块用一个链式Map结构（LinkedHashMap）来管理堆内和堆外存储内存中所有的Block对象的实例<sup>[6]</sup>，对这个LinkedHashMap新增和删除间接记录了内存的申请和释放。

因为不能保证存储空间可以一次容纳Iterator中的所有数据，当前的计算任务在Unroll时要向MemoryManager申请足够的Unroll空间来临时占位，空间不足则Unroll失败，空间足够时可以继续进行。对于序列化的Partition，其所需的Unroll空间可以直接累加计算，一次申请。而非序列化的Partition则要在遍历Record的过程中依次申请，即每读取一条Record，采样估算其所需的Unroll空间并进行申请，空间不足时可以中断，释放已占用的Unroll空间。如果最终Unroll成功，当前Partition所占用的Unroll空间被转换为正常的缓存RDD的存储空间，如下图2所示。

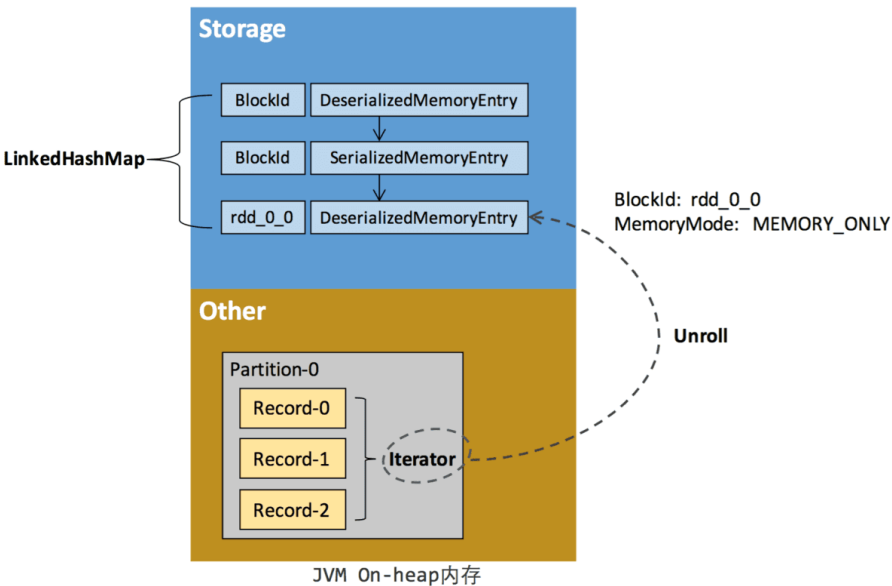


图2 Spark Unroll示意图

在《Spark内存管理详解（上）——内存分配》(<http://www.jianshu.com/p/3981b14df76b>)的图3和图5中可以看到，在静态内存管理时，Spark在存储内存中专门划分了一块Unroll空间，其大小是固定的，统一内存管理时则没有对Unroll空间进行特别区分，当存储空间不足是会根据动态占用机制进行处理。

3.3 淘汰和落盘

由于同一个Executor的所有的计算任务共享有限的存储内存空间，当有新的Block需要缓存但是剩余空间不足且无法动态占用时，就要对LinkedHashMap中的旧Block进行淘汰（Eviction），而被淘汰的Block如果其存储级别中同时包含存储到磁盘的要求，则要对其进行落盘（Drop），否则直接删除该Block。  
存储内存的淘汰规则为：

- 被淘汰的旧Block要与新Block的MemoryMode相同，即同属于堆外或堆内内存
- 新旧Block不能属于同一个RDD，避免循环淘汰
- 旧Block所属RDD不能处于被读状态，避免引发一致性问题
- 遍历LinkedHashMap中Block，按照最近最少使用（LRU）的顺序淘汰，直到满足新Block所需的内存空间。其中LRU是LinkedHashMap的特性。

落盘的流程则比较简单，如果其存储级别符合\_useDisk为true的条件，再根据其\_deserialized判断是否是序列化的形式，若是则对其进行序列化，最后将数据存储到磁盘，在Storage模块中更新其信息。

4. 执行内存管理



## 4.1 多任务间的分配

Executor内运行的任务同样共享执行内存，Spark用一个HashMap结构保存了任务到内存耗费的映射。每个任务可占用的执行内存大小的范围为  $1/2N \sim 1/N$ ，其中N为当前Executor内正在运行的任务的个数。每个任务在启动之时，要向MemoryManager请求申请最少为 $1/2N$ 的执行内存，如果不能被满足要求则该任务被阻塞，直到有其他任务释放了足够的执行内存，该任务才可以被唤醒。

## 4.2 Shuffle的内存占用

执行内存主要用来存储任务在执行Shuffle时占用的内存，Shuffle是按照一定规则对RDD数据重新分区的过程，我们来看Shuffle的Write和Read两阶段对执行内存的使用：

- Shuffle Write
  - 若在map端选择普通的排序方式，会采用ExternalSorter进行外排，在内存中存储数据时主要占用堆内执行空间。
  - 若在map端选择Tungsten的排序方式，则采用ShuffleExternalSorter直接对以序列化形式存储的数据排序，在内存中存储数据时可以占用堆外或堆内执行空间，取决于用户是否开启了堆外内存以及堆外执行内存是否足够。
- Shuffle Read
  - 在对reduce端的数据进行聚合时，要将数据交给Aggregator处理，在内存中存储数据时占用堆内执行空间。
  - 如果需要进行最终结果排序，则要将再次将数据交给ExternalSorter处理，占用堆内执行空间。

在ExternalSorter和Aggregator中，Spark会使用一种叫AppendOnlyMap的哈希表在堆内执行内存中存储数据，但在Shuffle过程中所有数据并不能都保存到该哈希表中，当这个哈希表占用的内存会进行周期性地采样估算，当其大到一定程度，无法再从MemoryManager申请到新的执行内存时，Spark就会将其全部内容存储到磁盘文件中，这个过程被称为溢存(Spill)，溢存到磁盘的文件最后会被归并(Merge)。

Shuffle Write阶段中用到的Tungsten是Databricks公司提出的对Spark优化内存和CPU使用的计划<sup>[4]</sup>，解决了一些JVM在性能上的限制和弊端。Spark会根据Shuffle的情况来自动选择是否采用Tungsten排序。Tungsten采用的页式内存管理机制建立在MemoryManager之上，即Tungsten对执行内存的使用进行了一步的抽象，这样在Shuffle过程中无需关心数据具体存储在堆内还是堆外。每个内存页用一个MemoryBlock来定义，并用Object obj 和 long offset 这两个变量统一标识一个内存页在系统内存中的地址。堆内的MemoryBlock是以long型数组的形式分配的内存，其obj 的值为是这个数组的对象引用，offset 是long型数组的在JVM中的初始偏移地址，两者配合使用可以定位这个数组在堆内的绝对地址；堆外的MemoryBlock是直接申请到的内存块，其obj 为null，offset 是这个内存块在系统内存中的64位绝对地址。Spark用MemoryBlock巧妙地堆内和堆外内存页统一抽象封装，并用页表(pageTable)管理每个Task申请到的内存页。

Tungsten页式管理下的所有内存用64位的逻辑地址表示，由页号和页内偏移量组成：

1. 页号：占13位，唯一标识一个内存页，Spark在申请内存页之前要先申请空闲页号。
2. 页内偏移量：占51位，是在使用内存页存储数据时，数据在页内的偏移地址。

有了统一的寻址方式，Spark可以用64位逻辑地址的指针定位到堆内或堆外的内存，整个Shuffle Write排序的过程只需要对指针进行排序，并且无需反序列化，整个过程非常高效，对于内存访问效率和CPU使用效率带来了明显的提升<sup>[5]</sup>。

## 小结



Spark的存储内存和执行内存有着截然不同的管理方式：对于存储内存来说，Spark用一个LinkedHashMap来集中管理所有的Block，Block由需要缓存的RDD的Partition转化而成；而对于执行内存，Spark用AppendOnlyMap来存储Shuffle过程中的数据，在Tungsten排序中甚至抽象为页式内存管理，开辟了全新的JVM内存管理机制。

结束语

Spark的内存管理是一套复杂的机制，且Spark的版本更新比较快，笔者水平有限，难免有叙述不清、错误的地方，若读者有好的建议和更深的理解，还望不吝赐教。

参考文献

- 1. 《Spark技术内幕：深入解析Spark内核架构于实现原理》——第8章 Storage模块详解 (https://book.douban.com/subject/26649141/)
- 2. Spark存储级别的源码 (https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/storage/StorageLevel.scala)
- 3. Spark Sort Based Shuffle内存分析 (http://www.jianshu.com/p/c83bb237caa8)
- 4. Project Tungsten: Bringing Apache Spark Closer to Bare Metal (https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html)
- 5. Spark Tungsten-sort Based Shuffle 分析 (http://www.jianshu.com/p/d328c96aebfd)
- 6. 探索Spark Tungsten的秘密 (https://github.com/hustnn/TungstenSecret/tree/master)
- 7. Spark Task 内存管理 (on-heap&off-heap) (http://www.jianshu.com/p/8f9ed2d58a26);

Big Data (/nb/9130292) 举报文章 © 著作权归作者所有



LeonLu (/u/5b15278387a0)

写了 19236 字，被 42 人关注，获得了 57 个喜欢

(/u/5b15278387a0)

+ 关注

IT圈不知名青年司机，了解大数据起步手法，擅长后端路段的平稳驾驶，熟悉代码的保养和维修。

如果觉得我的文章对您有用，请随意赞赏。您的支持将鼓励我继续创作！

赞赏支持

喜欢 (/sign\_in?utm\_source=desktop&utm\_medium=not-signed-in-like-button)

5

更多分享

(http://cwb.assets.jianshu.io/notes/images/951303C




登录 (/sign\_in?utm\_source=desktop&utm\_medium=not-signed-in-comment-form)

发表评论

2条评论

只看作者

按喜欢排序 按时间正序 按时间倒序



dream2017 (/u/9cf090796e34)

2楼 · 2017.03.02 05:40

(/u/9cf090796e34)