Spark内存管理详解（上）——内存分配

作者 LeonLu (/u/5b15278387a0) ＋关注
2017.02.24 23:12* 字数 2968 阅读 133 评论 0 喜欢 3
(/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，下一篇见《Spark内存管理详解（下）——内存管理》 (http://www.jianshu.com/p/58288b862030)

引言

Spark作为一个基于内存的分布式计算引擎，其内存管理模块在整个系统中占据着非常重要的角色。理解Spark内存管理的基本原理，有助于更好地开发Spark应用程序和进行性能调优。本文旨在梳理出Spark内存管理的脉络，抛砖引玉，引出读者对这个话题的深入探讨。本文中阐述的原理基于Spark 2.1版本，阅读本文需要读者有一定的Spark和Java基础，了解RDD、Shuffle、JVM等相关概念。

在执行Spark的应用程序时，Spark集群会启动Driver和Executor两种JVM进程，前者为主控进程，负责创建Spark上下文，提交Spark作业（Job），并将作业转化为计算任务（Task），在各个Executor进程间协调任务的调度，后者负责在工作节点上执行具体的计算任务，并将结果返回给Driver，同时为需要持久化的RDD提供存储功能[1]。由于Driver的内存管理相对来说较为简单，本文主要对Executor的内存管理进行分析，下文中的Spark内存均特指Executor的内存。

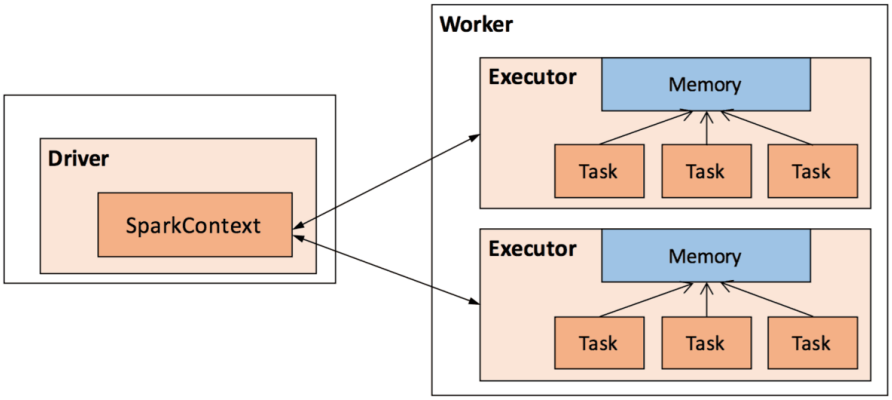

图1 Spark的Driver和Worker

1 堆内和堆外内存

作为一个JVM进程，Executor的内存管理建立在JVM的内存管理之上，Spark对JVM的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。同时，Spark引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用。

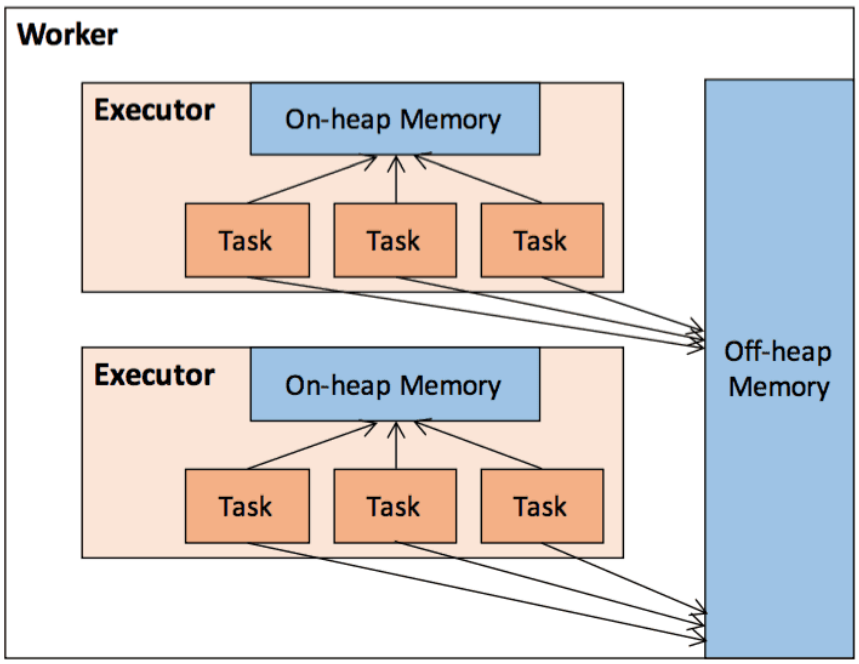

图2 堆外和堆内内存

1.1 堆内

堆内内存的大小，由Spark应用程序启动时的 `–executor–memory` 或 `spark.executor.memory` 参数配置。Executor内运行的并发任务共享JVM堆内内存，这些任务在缓存RDD和广播（Broadcast）数据时占用的内存被规划为存储（Storage）内存，而这些任务在执行Shuffle时占用的内存被规划为执行（Execution）内存，剩余的部分不做特殊规划，那些Spark内部的对象实例，或者用户定义的Spark应用程序中的对象实例，均占用剩余的空间。不同的管理模式下，这三部分占用的空间大小各不相同（下面第2小节介绍）。

Spark对堆内内存的管理是一种逻辑上的“规划式”的管理，因为对象实例占用内存的申请和释放都由JVM完成，Spark只能在申请后和释放前**记录**这些内存：

- **申请内存**：
  - Spark在代码中new一个对象实例

- JVM从堆内内存分配空间，创建对象并返回对象引用
- Spark保存该对象的引用，记录该对象占用的内存

- **释放内存**：
  - Spark记录该对象释放的内存，删除该对象的引用
  - 等待JVM的垃圾回收机制释放该对象占用的堆内内存

我们知道，JVM的对象可以以序列化的方式存储，序列化的过程是将对象转换为二进制字节流，本质上可以理解为将非连续空间的链式存储转化为连续空间或块存储，在访问时则需要进行序列化的逆过程——反序列化，将字节流转化为对象，序列化的方式可以节省存储空间，但增加了存储和读取时候的计算开销。

对于Spark中序列化的对象，由于是字节流的形式，其占用的内存大小可直接计算，而对于非序列化的对象，其占用的内存是通过周期性地采样近似估算而得，即并不是每次新增的数据项都会计算一次占用的内存大小，这种方法降低了时间开销但是有可能误差较大，导致某一时刻的实际内存有可能远远超出预期[2]。此外，在被Spark标记为释放的对象实例，很有可能在实际上并没有被JVM回收，导致实际可用的内存小于Spark记录的可用内存。所以Spark并不能准确记录实际可用的堆内内存，从而也就无法完全避免内存溢出（OOM, Out of Memory）的异常。

虽然不能精准控制堆内内存的申请和释放，但Spark通过对存储内存和执行内存各自独立的规划管理，可以决定是否要在存储内存里缓存新的RDD，以及是否为新的任务分配执行内存，在一定程度上可以提升内存的利用率，减少异常的出现。

1.2 堆外

为了进一步优化内存的使用以及提高Shuffle时排序的效率，Spark引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，存储经过序列化的二进制数据。利用JDK Unsafe API（从Spark 2.0开始，在管理堆外的存储内存时不再基于Tachyon，而是与堆外的执行内存一样，基于JDK Unsafe API实现[3]），Spark可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的GC扫描和回收，提升了处理性能。堆外内存可以被精确地申请和释放，而且序列化的数据占用的空间可以被精确计算，所以相比堆内内存来说降低了管理的难度，也降低了误差。

在默认情况下堆外内存并不启用，可通过配置 `spark.memory.offHeap.enabled` 参数启用，并由 `spark.memory.offHeap.size` 参数设定堆外空间的大小。除了没有other空间，堆外内存与堆内内存的划分方式相同，所有运行中的并发任务共享存储内存和执行内存。

1.3 接口

Spark为存储内存和执行内存的管理提供了统一的接口—— `MemoryManager` ，同一个Executor内的任务都调用这个接口的方法来申请或释放内存，同时在调用这些方法时都需要指定内存模式（MemoryMode），这个参数决定了是在堆内还是堆外完成这次操作。MemoryManager的具体实现上，Spark 1.6之后默认为统一管理（Unified Memory Manager (https://github.com/apache/spark/blob/v2.1.0/core/src/main/scala/org/apache/spark/memory/UnifiedMemoryManager.scala)）方式，1.6之前采用的静态管理（Static Memory Manager (https://github.com/apache/spark/blob/v2.1.0/core/src/main/scala/org/apache/spark/memory/StaticMemoryManager.scala)）方式仍被保留，可通过配置 `spark.memory.useLegacyMode` 参数启用。两种方式的区别在于对空间分配的方式，下面分别对这两种方式进行介绍。

2 内存空间分配

2.1 静态内存管理

堆内

在静态内存管理机制下，存储内存、执行内存和其他内存三部分的大小在Spark应用程序运行期间是固定的，但用户可以在应用程序启动前进行配置，堆内内存的分配如图3所示：

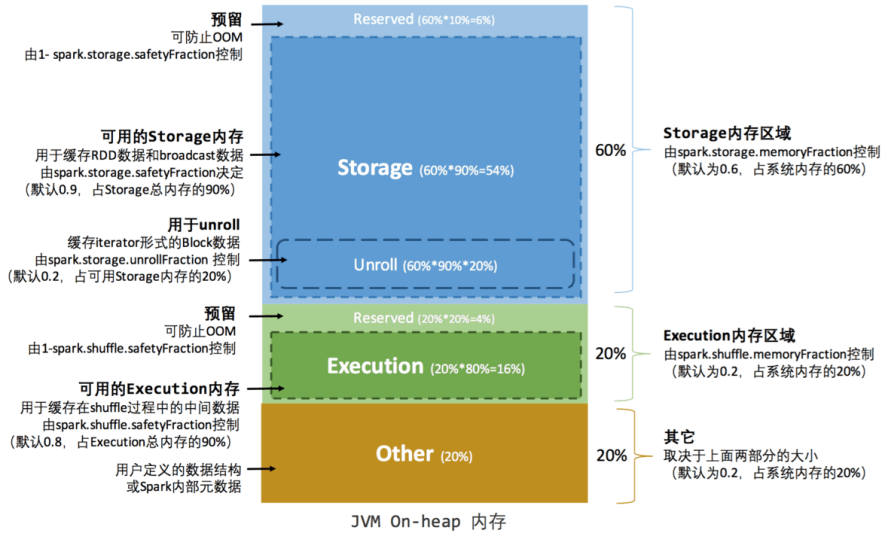

图3 静态内存管理图示——堆内

可以看到，可用的堆内内存的大小需要按照下面的方式计算：

```
可用的存储内存 = systemMaxMemory * spark.storage.memoryFraction * spark.storage.safety
可用的执行内存 = systemMaxMemory * spark.shuffle.memoryFraction * spark.shuffle.safety
```

其中 `systemMaxMemory` 取决于当前JVM堆内内存的大小，最后可用的执行内存或者存储内存要在此基础上与各自的 `memoryFraction` 参数和 `safetyFraction` 参数相乘得出。上述计算公式中的两个 `safetyFraction` 参数，其意义在于在逻辑上预留出 `1-safetyFraction` 这么一块保险区域，降低因实际内存超出当前预设范围而导致OOM的风险（上文提到，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时Spark并没有区别对待，和“其它内存”一样交给了JVM去管理。

堆外

堆外的空间分配较为简单，存储内存、执行内存的大小同样是固定的，如图4所示：

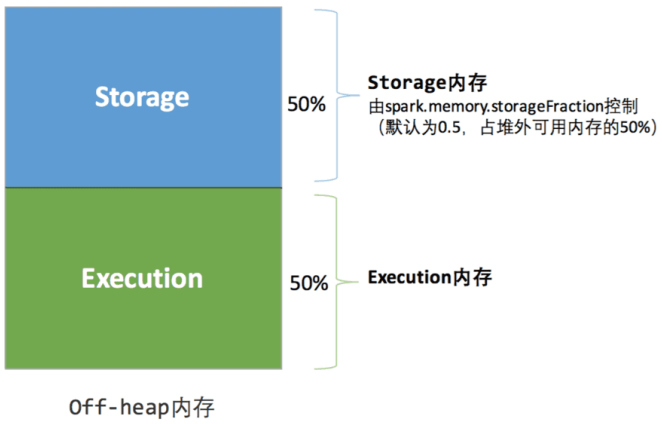

图4 静态内存管理图示——堆外

可用的执行内存和存储内存占用的空间大小直接由参数 `spark.memory.storageFraction` 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。

静态内存管理机制实现起来较为简单，但如果用户不熟悉Spark的存储机制，或没有根据具体的数据规模和计算任务或做相应的配置，很容易造成“一半海水，一半火焰”的局面，即存储内存和执行内存中的一方剩余大量的空间，而另一方却早早被占满，不得不淘汰或移出旧的内容以存储新的内容。由于新的内存管理机制的出现，这种方式目前已经很少有开发者使用，出于兼容旧版本的应用程序的目的，Spark仍然保留了它的实现。

2.2 统一内存管理

Spark 1.6之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域，如图5和图6所示

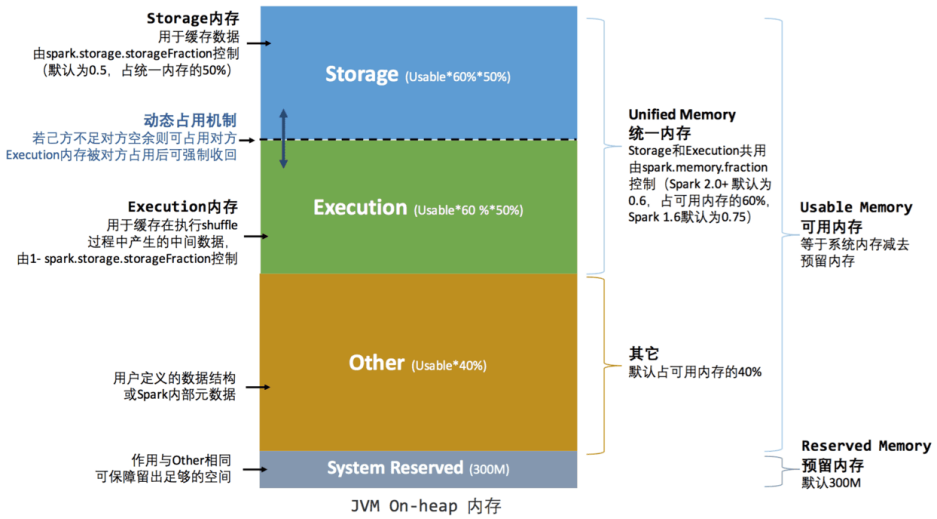

图5 统一内存管理图示——堆内

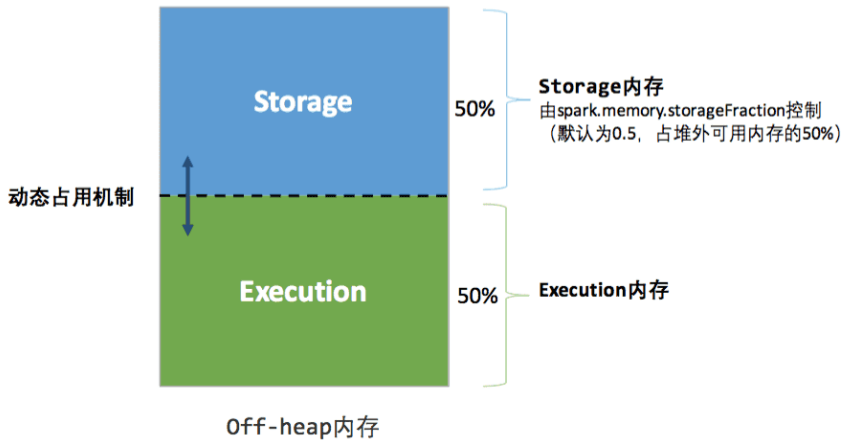

图6 统一内存管理图示——堆外

动态占用机制的规则如下：

- 设定基本的存储内存和执行内存区域（`spark.storage.storageFraction` 参数），该设定确定了双方各自拥有的空间的范围
- 双方的空间都不足时，则存储到硬盘；若己方空间不足而对方空余时，可借用对方的空间；（存储空间不足是指不足以放下一个完整的Block）
- 执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后“归还”借用的空间
- 存储内存的空间被对方占用后，无法让对方“归还”，因为需要考虑Shuffle过程中的很多因素，实现起来较为复杂[4]

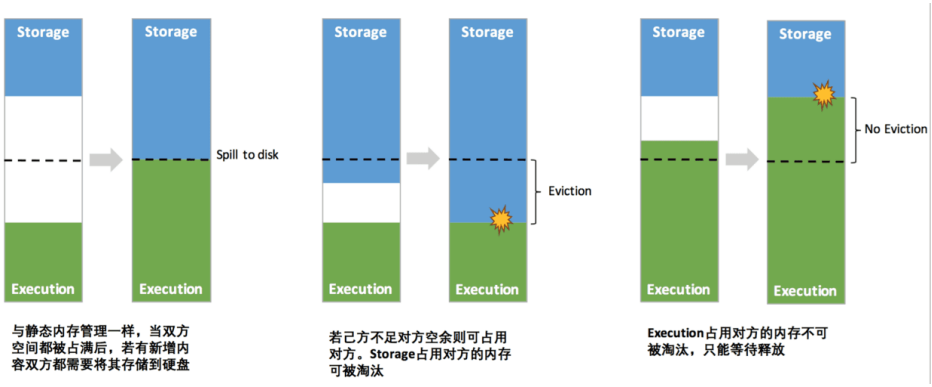

图7 动态占用机制图解

小结

凭借统一内存管理机制，Spark在一定程度上提高了堆内和堆外内存资源的利用率，降低了开发者维护Spark内存的难度，但并不意味着开发者可以高枕无忧。譬如，所以如果存储内存的空间太大或者说缓存的数据过多，反而会导致频繁的全量垃圾回收，降低任务执行时的性能，因为缓存的RDD数据通常都是长期驻留内存的[5]。所以要想充分发挥Spark的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

参考文献

1. Spark Cluster Mode Overview (http://spark.apache.org/docs/latest/cluster-overview.html)
2. Spark Sort Based Shuffle内存分析 (http://www.jianshu.com/p/c83bb237caa8)
3. Spark OFF_HEAP (http://www.jianshu.com/p/c6f6d4071560)
4. Unified Memory Management in Spark 1.6 (https://issues.apache.org/jira/secure/attachment/12765646/unified-memory-management-spark-10000.pdf)
5. Tuning Spark: Garbage Collection Tuning (http://spark.apache.org/docs/latest/tuning.html#garbage-collection-tuning)
6. Spark Architecture (https://0x0fff.com/spark-architecture/)