# 第54课：独家揭秘Spark统一内存管理

1. 传统的Spark内存管理的问题
2. Spark统一内存管理
3. 展望
4. Spark内存由Execution、Storage、Other三部分组成：

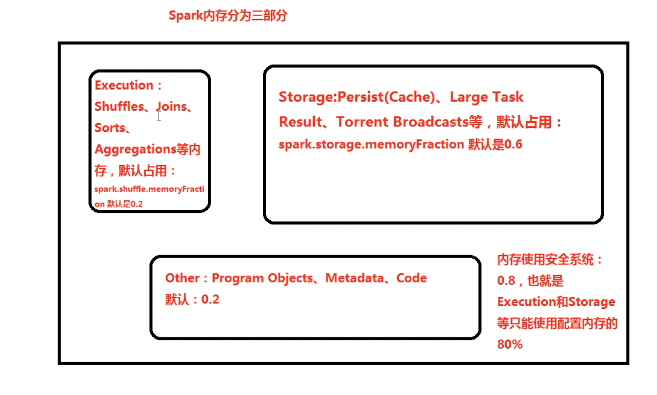
Execution：内存角度做Shuffles、joins、Sorts、Aggregations等，默认占用：0.2（安全范围内），在spark.shuffle.memoryFraction进行设置。决定了在Shuffle过程中占用总内存多少时进行Spill。

Storage:存储层面做Persist（Cache）、Large Task Result、Torrent Broadcast等，默认占用0.6（安全范围内），在spark.storage.memoryFraction进行设置。

Other：存在以上两种空间程序是无法运行的，缺少对象、元数据等，这时Other中就是存储Program Objects、Metadata、Code等，默认占用0.2。

注：由于shuffle过程中占用的内存比预估值大，所以提出了spark.shuffle.safetyFraction进行内存安全的保障，也就是Execution、Storage等使用安全范围内的内存，一般默认值为0.8。spark的性能主要受shuffle的影响，在shuffle的过程中为了保障安全空间，在不断向外界抓取数据的时候为了有缓存的空间，这时可能会进行spill磁盘的IO操作，浪费大量时间。但这时storage可能不进行任何的操作，浪费了大量的内存空间，他本来的作用是加速内存迭代，这时矛盾产生了。也可能导致重新计算、Task可能占满整个内存空间等后果。

组成分块图为：



1. 从Execution的角度分析内存分配，主要有以下几个类：

* ShuffleMemoryManager负责全局计数和内存调度。它根据task当前内存用量决定如何进行分配。
* TaskMemoryManager负责每个task的内存分配和跟踪。它通过page table追踪on-heap内存块，task退出时它若检查到有page未释放则会抛出异常。它使用ExecutorMemoryManager真正执行分配和释放。
* ExecutorMemoryManager负责具体实现，它管理on-heap和off-heap内存，并实现了一个weak reference pool支持跨tasks的空闲页复用。

Task需分配内存

向ShuffleMemoryManager申请内存

Task会向TaskMemoryManager要求分配内存

ExecutorMemoryManager真正分配内存

注：Task向TaskMemoryManager要求分配内存成功后会更新

自己的page table

1. 从Storage的角度分析内存分配：

Storage内存通过BlockManager管理，主要用于缓存RDD的partition，它也被用于torrent broadcast和将Large Task Result数据发送给driver。任何数据都存在Storage level用于指定是存储在内存、磁盘还是off-heap。Storage level也指定是否以序列化的方式进行存储。在纵多Storage level中MEMORY\_AND\_DISK\_SER是最廉价和最安全，内存数据也是以序列化方式存储的，如果需要Spill情况下转移至磁盘时无需额外序列化。

RDD运行时需要空间，获取它的数据是一条条获取，但可能存在空间不够的情况，直接Spill到磁盘，这时只运行内存能够承载的数据。内存中的iterator的过程需转化为数组，但在内存不够的情况下者需Unroll ，用以避免OOM，不断检查内存是否足够。这个内存空间是从Storage中获得。如果当前没有缓存数据，unrolling可占用全部storage空间。为了防止占用过多的空间unrolling内存上限由spark.storage.unrollFraction(默认0.2)配置。

注：如果unroll失败了，BlockManager会直接把unroll对应的数据Spill到磁盘。

1. 在使用legacy的方式下，有如下内存借用规则：

* execution可借用所有空闲的storage内存
* execution也可以收回storage向自己借用的内存
* 但storage已使用的、属于storage的内存，不可被借用
* storage可借用execution空闲内存
* 但storage无法收回已经被execution借用的内存

本方式的类为UnifiedMemoryManager.scala中，其中的注释内容详细的介绍了该方式的所有情况。其中值得注意的是整个准则为计算高于存储。

其中获取内存源码如下：

**object** UnifiedMemoryManager {  
  
 *// Set aside a fixed amount of memory for non-storage, non-execution purposes.  
 // This serves a function similar to `spark.memory.fraction`, but guarantees that we reserve  
 // sufficient memory for the system even for small heaps. E.g. if we have a 1GB JVM, then  
 // the memory used for execution and storage will be (1024 - 300) \* 0.75 = 543MB by default.* **private val** *RESERVED\_SYSTEM\_MEMORY\_BYTES* = 300 \* 1024 \* 1024  
 **def** apply(conf: SparkConf, numCores: Int): UnifiedMemoryManager = {  
 **val** maxMemory = *getMaxMemory*(conf)  
 **new** UnifiedMemoryManager(  
 conf,  
 maxMemory = maxMemory,  
 storageRegionSize =  
 (maxMemory \* conf.getDouble(**"spark.memory.storageFraction"**, 0.5)).toLong,  
 numCores = numCores)  
 }  
 */\*\*  
 \* Return the total amount of memory shared between execution and storage, in bytes.  
 \*/* **private def** getMaxMemory(conf: SparkConf): Long = {  
 **val** systemMemory = conf.getLong(**"spark.testing.memory"**, Runtime.*getRuntime*.maxMemory)  
 **val** reservedMemory = conf.getLong(**"spark.testing.reservedMemory"**,  
 **if** (conf.contains(**"spark.testing"**)) 0 **else** *RESERVED\_SYSTEM\_MEMORY\_BYTES*)  
 **val** minSystemMemory = reservedMemory \* 1.5  
 **if** (systemMemory < minSystemMemory) {  
 **throw new** IllegalArgumentException(**s"System memory $**systemMemory **must "** +  
 **s"be at least $**minSystemMemory**. Please use a larger heap size."**)  
 }  
 **val** usableMemory = systemMemory – reservedMemory//可以使用的内存  
 **val** memoryFraction = conf.getDouble(**"spark.memory.fraction"**, 0.75)  
 (usableMemory \* memoryFraction).toLong//可用的最大内存  
 }  
}