Spark Executor内存管理

Posted by arganzheng (/about) | November 20, 2018

spark (/tags#spark)

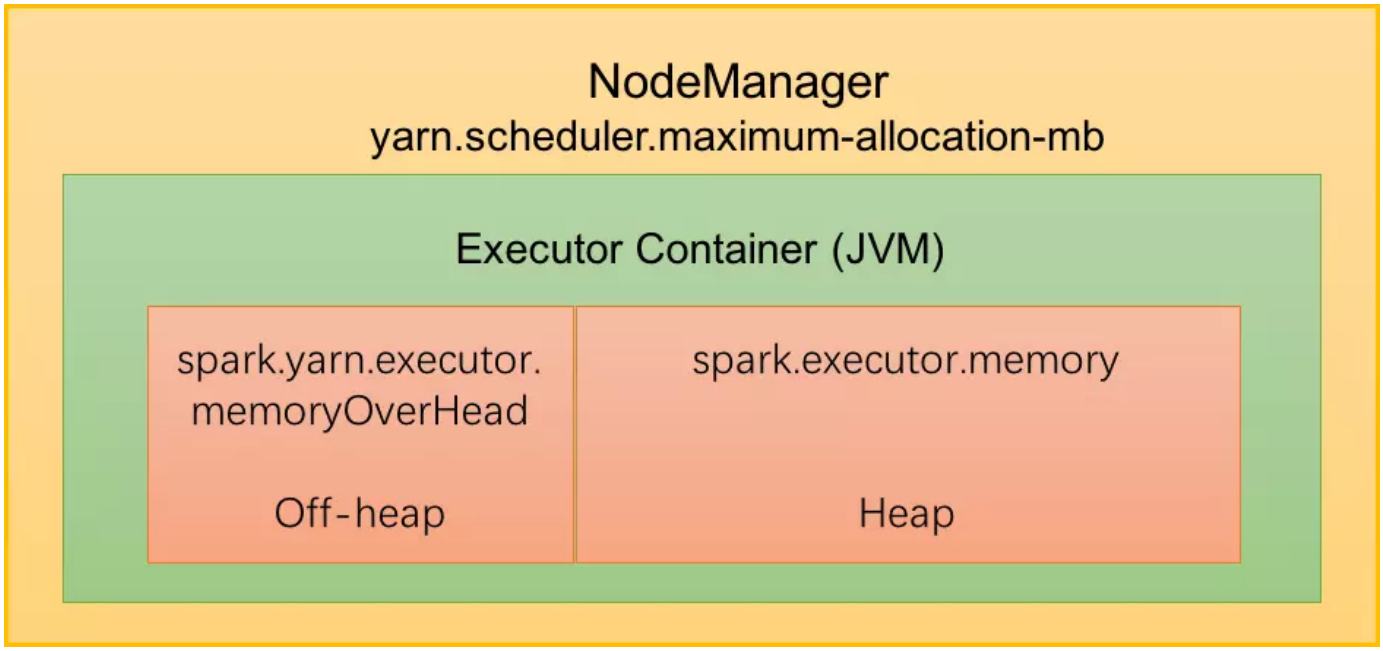
我们都知道 Spark 能够有效的利用内存并进行分布式计算，其内存管理模块在整个系统中扮演着非常重要的角色。为了更好地利用 Spark，深入地理解其内存管理模型具有非常重要的意义，这有助于我们对 Spark 进行更好的调优；在出现各种内存问题时，能够摸清头脑，找到哪块内存区域出现问题。

首先我们知道在执行 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。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。

另外，Spark 1.6 之前使用的是静态内存管理 (StaticMemoryManager) 机制，StaticMemoryManager 也是 Spark 1.6 之前唯一的内存管理器。在 Spark1.6 之后引入了统一内存管理 (UnifiedMemoryManager) 机制，UnifiedMemoryManager 是 Spark 1.6 之后默认的内存管理器，1.6 之前采用的静态管理（StaticMemoryManager）方式仍被保留，可通过配置 `spark.memory.useLegacyMode` 参数启用。这里仅对统一内存管理模块 (UnifiedMemoryManager) 机制进行分析。

Executor内存总体布局

默认情况下，Executor不开启堆外内存，因此整个 Executor 端内存布局如下图所示:



我们可以看到在Yarn集群管理模式中，Spark 以 Executor Container 的形式在 NodeManager 中运行，其可使用的内存上限由 `yarn.scheduler.maximum-allocation-mb` 指定，我们称之为MonitorMemory。

整个Executor内存区域分为两块：

1、JVM堆外内存

大小由 `spark.yarn.executor.memoryOverhead` 参数指定。默认大小为 `executorMemory * 0.10`，with minimum of 384m。

此部分内存主要用于JVM自身，字符串, NIO Buffer（Driect Buffer）等开销。此部分为用户代码及Spark 不可操作的内存，不足时可通过调整参数解决。

> *The amount of off-heap memory (in megabytes) to be allocated per executor. This is memory that accounts for things like VM overheads, interned strings, other native overheads, etc. This tends to grow with the executor size (typically 6-10%).*

2、堆内内存（ExecutorMemory）

大小由 Spark 应用程序启动时的 `-executor-memory` 或 `spark.executor.memory` 参数配置，即JVM最大分配的堆内存（`-Xmx`）。Spark为了更高效的使用这部分内存，对这部分内存进行了逻辑上的划分管理。我们在下面的统一内存管理会详细介绍。

NOTES

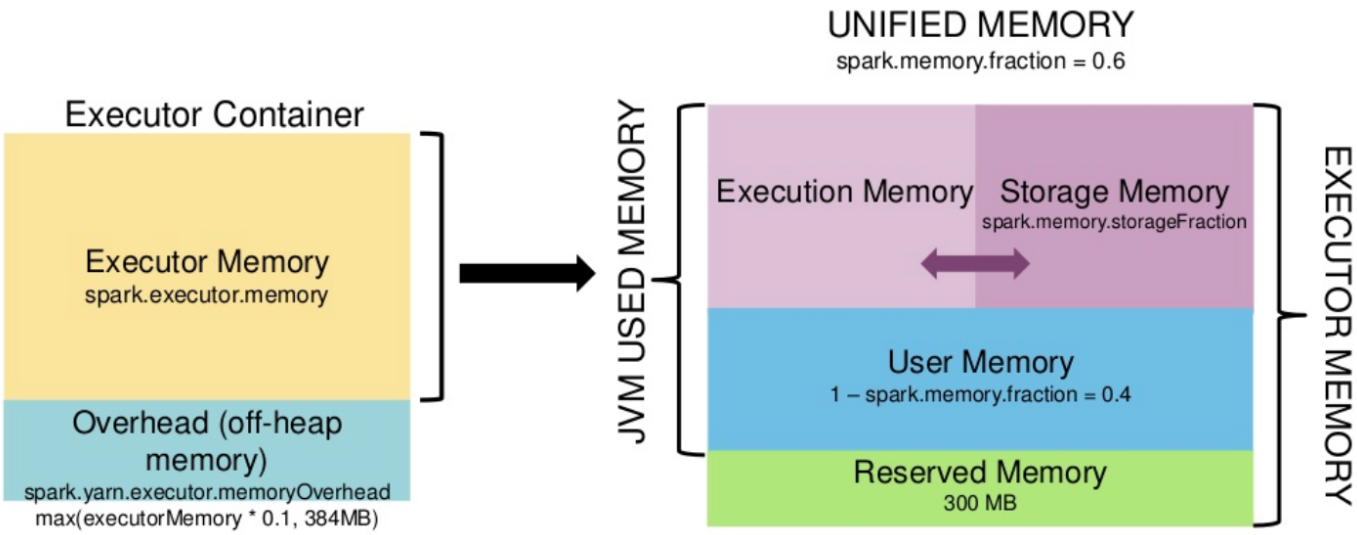
对于Yarn集群，存在: `ExecutorMemory + MemoryOverhead <= MonitorMemory`，若应用提交之时，指定的 ExecutorMemory 与 MemoryOverhead 之和大于 MonitorMemory，则会导致 Executor 申请失败；若运行过程中，实际使用内存超过上限阈值，Executor 进程会被 Yarn 终止掉 (kill)。

统一内存管理

Spark 1.6之后引入了统一内存管理，包括了堆内内存 (On-heap Memory) 和堆外内存 (Off-heap Memory) 两大区域，下面对这两块区域进行详细的说明。

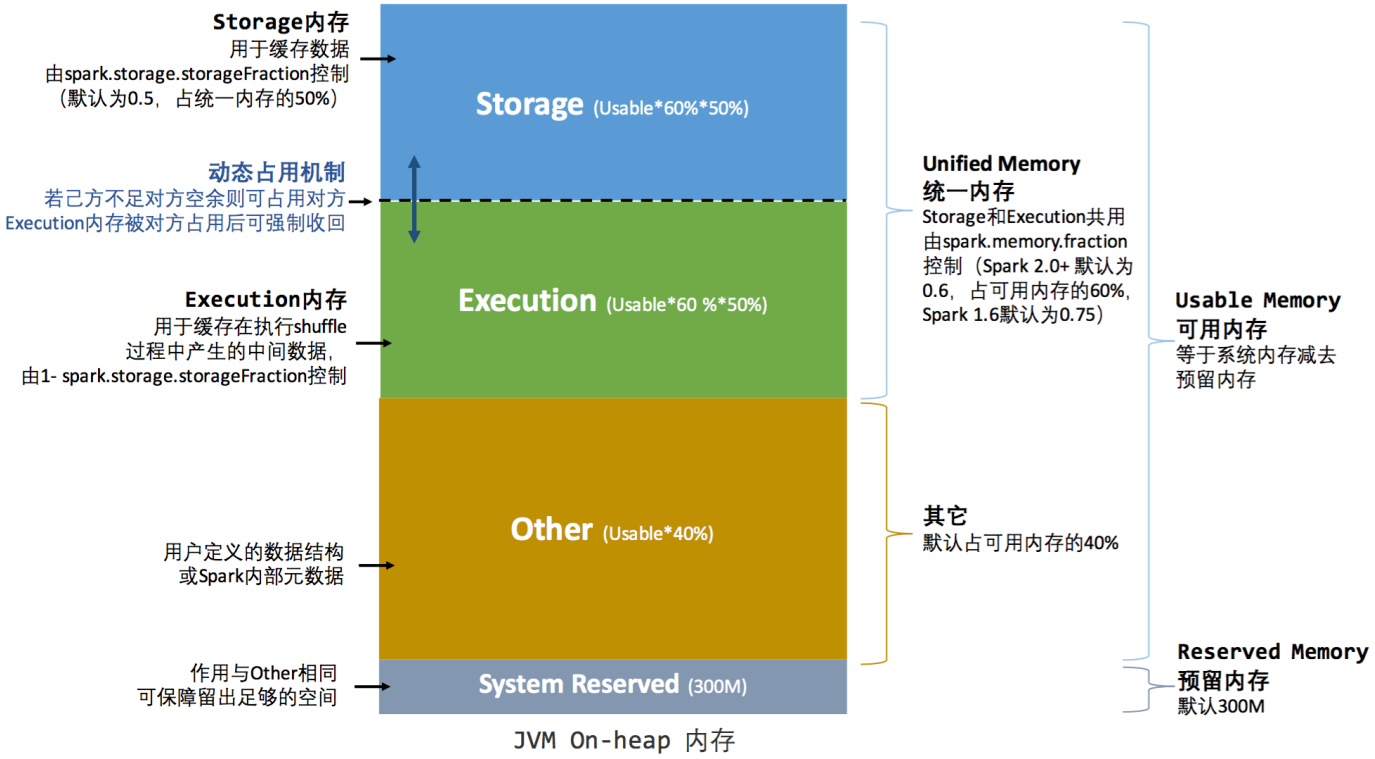
堆内内存 (On-heap Memory)

默认情况下，Spark 仅仅使用了堆内内存。Spark 对堆内内存的管理是一种逻辑上的“规划式”的管理，Executor 端的堆内内存区域在逻辑上被划分为以下四个区域:



1. 执行内存 (Execution Memory) : 主要用于存放 Shuffle、Join、Sort、Aggregation 等计算过程中的临时数据;
2. 存储内存 (Storage Memory) : 主要用于存储 spark 的 cache 数据，例如RDD的缓存、unroll数据;
3. 用户内存（User Memory）: 主要用于存储 RDD 转换操作所需要的数据，例如 RDD 依赖等信息;
4. 预留内存（Reserved Memory）: 系统预留内存，会用来存储Spark内部对象。

下面的图对这个四个内存区域的分配比例做了详细的描述:



1、预留内存 (Reserved Memory)

系统预留内存，会用来存储Spark内部对象。其大小在代码中是写死的，其值等于 300MB，这个值是不能修改的（如果在测试环境下，我们可以通过 `spark.testing.reservedMemory` 参数进行修改）；如果Executor分配的内存小于 1.5 * 300 = 450M 时，Executor将无法执行。

2、存储内存 (Storage Memory)

主要用于存储 spark 的 cache 数据，例如 RDD 的缓存、广播（Broadcast）数据、和 unroll 数据。内存占比为 `UsableMemory * spark.memory.fraction * spark.memory.storageFraction`，Spark 2+ 中，默认初始状态下 Storage Memory 和 Execution Memory 均约占系统总内存的30%（`1 * 0.6 * 0.5 = 0.3`）。在 UnifiedMemory 管理中，这两部分内存可以相互借用，具体借用机制我们下一小节会详细介绍。

3、执行内存 (Execution Memory)

主要用于存放 Shuffle、Join、Sort、Aggregation 等计算过程中的临时数据。内存占比为 `UsableMemory * spark.memory.fraction * (1 - spark.memory.storageFraction)`，Spark 2+ 中，默认初始状态下 Storage Memory 和 Execution Memory 均约占系统总内存的30%（`1 * 0.6 * (1 - 0.5) = 0.3`）。在 UnifiedMemory 管理中，这两部分内存可以相互借用，具体借用机制我们下一小节会详细介绍。

4、其他/用户内存 (Other/User Memory) : 主要用于存储 RDD 转换操作所需要的数据，例如 RDD 依赖等信息。内存占比为 `UsableMemory * (1 - spark.memory.fraction)`，在Spark2+ 中，默认占可用内存的40%（`1 * (1 - 0.6) = 0.4`）。

其中，`usableMemory = executorMemory - reservedMemory`，这个就是 Spark 可用内存。

NOTES

1、为什么设置300M预留内存

统一内存管理最初版本other这部分内存没有固定值 300M 设置，而是和静态内存管理相似，设置的百分比，最初版本占 25%。百分比设置在实际使用中出现了问题，若给定的内存较低时，例如 1G，会导致 OOM，具体讨论参考这里 Make unified memory management work with small heaps (https://issues.apache.org/jira/browse/SPARK-12081)。因此，other这部分内存做了修改，先划出 300M 内存。

2、 `spark.memory.fraction` 由 0.75 降至 0.6

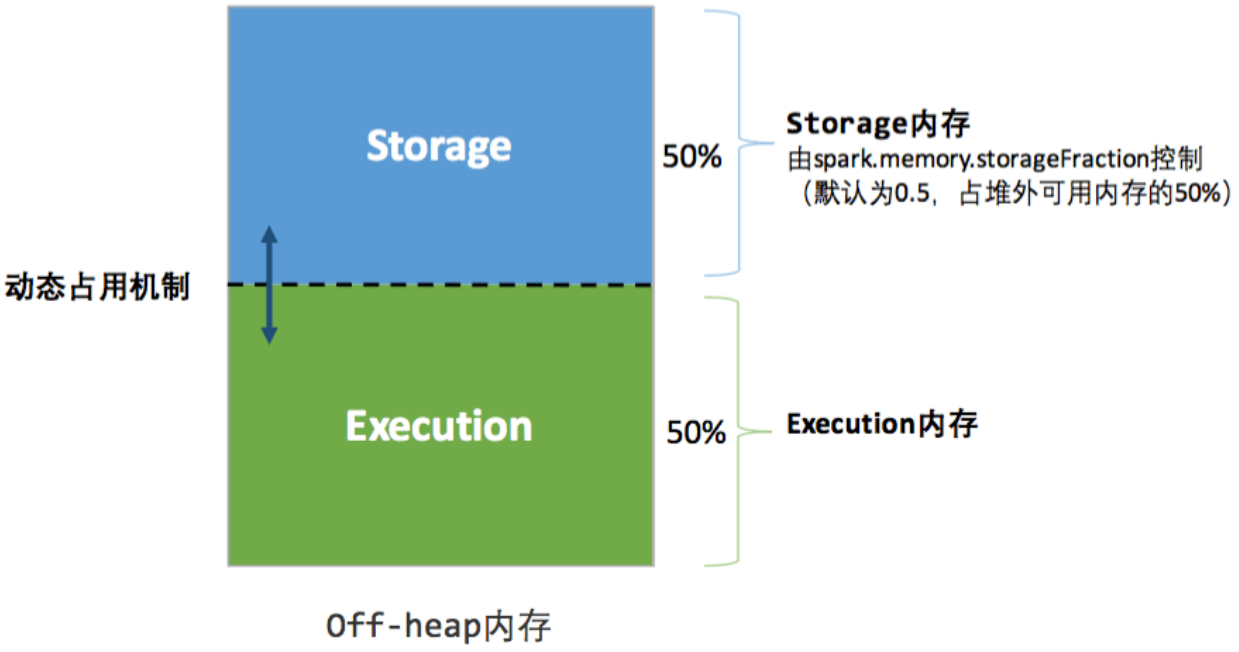
`spark.memory.fraction` 最初版本的值是 0.75，很多分析统一内存管理这块的文章也是这么介绍的，同样的，在使用中发现这个值设置的偏高，导致了 gc 时间过长，spark 2.0 版本将其调整为 0.6，详细谈论参见 Reduce spark.memory.fraction default to avoid overrunning old gen in JVM default config (https://issues.apache.org/jira/browse/SPARK-15796)。

堆外内存 (Off-heap Memory)

Spark 1.6 开始引入了 Off-heap memory (详见SPARK-11389)。这种模式不在 JVM 内申请内存，而是调用 Java 的 unsafe 相关 API 进行诸如 C 语言里面的 `malloc()` 直接向操作系统申请内存。这种方式下 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。另外，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。，缺点是必须自己编写内存申请和释放的逻辑。

默认情况下 `Off-heap` 模式的内存并不启用，我们可以通过 `spark.memory.offHeap.enabled` 参数开启，并由 `spark.memory.offHeap.size` 指定堆外内存的大小，单位是字节（占用的空间划归 JVM OffHeap 内存）。

如果堆外内存被启用，那么 Executor 内将同时存在堆内和堆外内存，两者的使用互补影响，这个时候 Executor 中的 Execution 内存是堆内的 Execution 内存和堆外的 Execution 内存之和，同理，Storage 内存也一样。其内存分布如下图所示：



相比堆内内存，堆外内存只区分 Execution 内存和 Storage 内存：

1、存储内存 (Storage Memory)

内存占比为 `maxOffHeapMemory * spark.memory.storageFraction`，Spark 2+ 中，默认初始状态下 Storage Memory 和 Execution Memory 均约占系统总内存的50%（`1 * 0.5 = 0.5`）。在 UnifiedMemory 管理中，这两部分内存可以相互借用，具体借用机制我们下一小节会详细介绍。

2、执行内存 (Execution Memory)

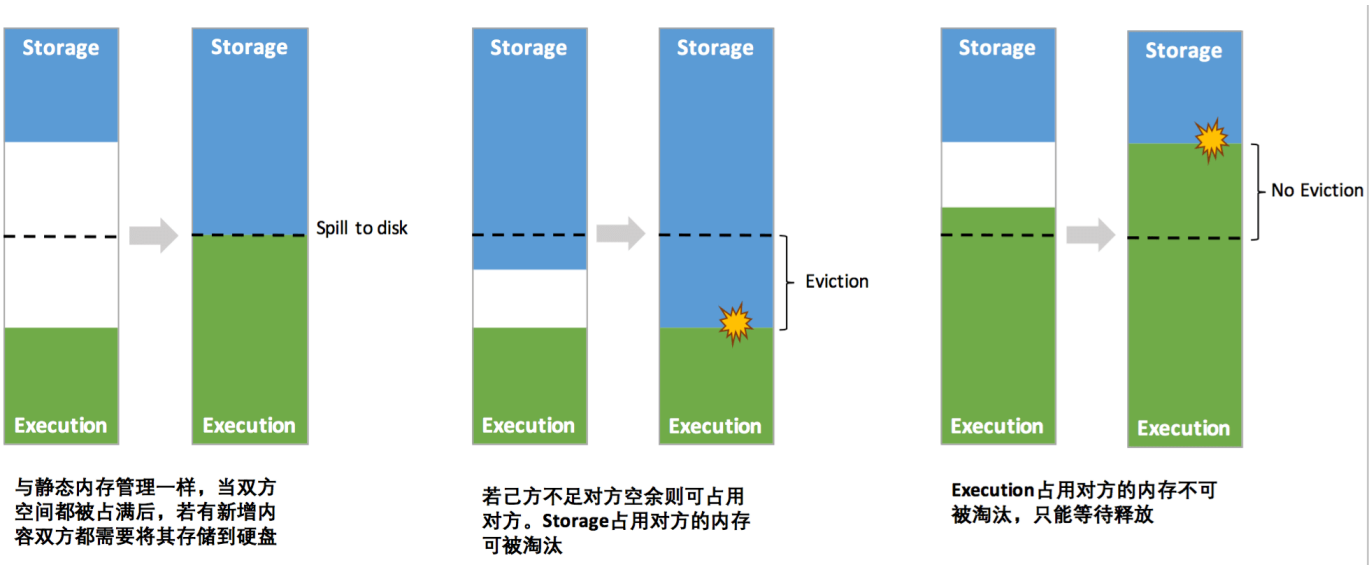
内存占比为 `maxOffHeapMemory * (1 - spark.memory.storageFraction)`，Spark 2+ 中，默认初始状态下 Storage Memory 和 Execution Memory 均约占系统总内存的50%（`1 * (1 - 0.5) = 0.5`）。在 UnifiedMemory 管理中，这两部分内存可以相互借用，具体借用机制我们下一小节会详细介绍。

Execution 内存和 Storage 内存动态占用机制

在 Spark 1.5 之前，Execution 内存和 Storage 内存分配是静态的，换句话说就是如果 Execution 内存不足，即使 Storage 内存有很大空闲程序也是无法利用到的；反之亦然。

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。

统一内存管理机制，与静态内存管理最大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：



其中最重要的优化在于动态占用机制，其规则如下：

- 程序提交的时候我们都会设定基本的 Execution 内存和 Storage 内存区域（通过 `spark.memory.storageFraction` 参数设置）。我们用 onHeapStorageRegionSize 来表示 `spark.storage.storageFraction` 划分的存储内存区域。这部分内存是不可以被驱逐(Evict)的存储内存（但是如果空闲是可以被占用的）。
- 当计算内存不足时，可以借用 onHeapStorageRegionSize 中未使用部分，且 Storage 内存的空间被对方占用后，需要等待执行内存自己释放，不能抢占。
- 若实际 StorageMemory 使用量超过 onHeapStorageRegionSize，那么当计算内存不足时，可以驱逐并借用 `StorageMemory - onHeapStorageRegionSize` 部分，而 onHeapStorageRegionSize 部分不可被抢占。
- 反之，当存储内存不足时（存储空间不足是指不足以放下一个完整的 Block），也可以借用计算内存空间；但是 Execution 内存的空间被存储内存占用后，是可让对方将占用的部分转存到硬盘，然后“归还”借用的空间。
- 如果双方的空间都不足时，则存储到硬盘；将内存中的块存储到磁盘的策略是按照 LRU 规则进行的。

说明

1、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。可通过配置 `spark.memory.useLegacyMode` 参数启用。

2、 `spark.memory.storageFraction` 是不可被驱逐的内存空间。只有空闲的时候能够被执行内存占用，但是不能被驱逐抢占。

> *Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by spark.memory.fraction. The higher this is, the less working memory may be available to execution and tasks may spill to disk more often. Leaving this at the default value is recommended. For more detail, see this description.*

3、Storage 内存的空间被对方占用后，目前的实现是无法让对方"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；而且 Shuffle 过程产生的文件在后面一定会被使用到，而 Cache 在内存的数据不一定在后面使用。在 Unified Memory Management in Spark 1.6 (http://www.linuxprobe.com/wp-content/uploads/2017/04/unified-memory-management-spark-10000.pdf)中详细讲解了为何选择这种策略，简单总结如下:

1. 数据清除的开销 : 驱逐storage内存的开销取决于 storage level，MEMORY_ONLY 可能是最昂贵的，因为需要重新计算，MEMORY_AND_DISK_SER 正好相反，只涉及到磁盘IO。溢写 execution 内存到磁盘的开销并不昂贵，因为 execution 存储的数据格式紧凑(compact format)，序列化开销低。并且，清除的 storage 内存可能不会被用到，但是，可以预见的是，驱逐的 execution 内存是必然会再被读到内存的，频繁的驱除重读 execution 内存将导致昂贵的开销。
2. 实现的复杂度 : storage 内存的驱逐是容易实现的，只需要使用已有的方法，drop 掉 block。execution 则复杂的多，首先，execution 以 page 为单位管理这部分内存，并且确保相应的操作至少有 one page ，如果把这 one page 内存驱逐了，对应的操作就会处于饥饿状态。此外，还需要考虑 execution 内存被驱逐的情况下，等待 cache 的 block 如何处理。

4、上面说的借用对方的内存需要借用方和被借用方的内存类型都一样，都是堆内内存或者都是堆外内存，不存在堆内内存不够去借用堆外内存的空间。

任务内存管理（Task Memory Manager）

Executor 中任务以线程的方式执行，各线程共享JVM的资源（即 Execution 内存），任务之间的内存资源没有强隔离（任务没有专用的Heap区域）。因此，可能会出现这样的情况：先到达的任务可能占用较大的内存，而后到的任务因得不到足够的内存而挂起。

在 Spark 任务内存管理中，使用 HashMap 存储任务与其消耗内存的映射关系。每个任务可占用的内存大小为潜在可使用计算内存（ 潜在可使用计算内存为: 初始计算内存 + 可抢占存储内存）的 1/2n ~ 1/n，当剩余内存为小于 1/2n 时，任务将被挂起，直至有其他任务释放执行内存，而满足内存下限 1/2n，任务被唤醒。其中 n 为当前 Executor 中活跃的任务树。

比如如果 Execution 内存大小为 10GB，当前 Executor 内正在运行的 Task 个数为5，则该 Task 可以申请的内存范围为 10 / (2 * 5) ~ 10 / 5，也就是 1GB ~ 2GB 的范围。

任务执行过程中，如果需要更多的内存，则会进行申请，如果存在空闲内存，则自动扩容成功，否则，将抛出 OutOffMemroyError。

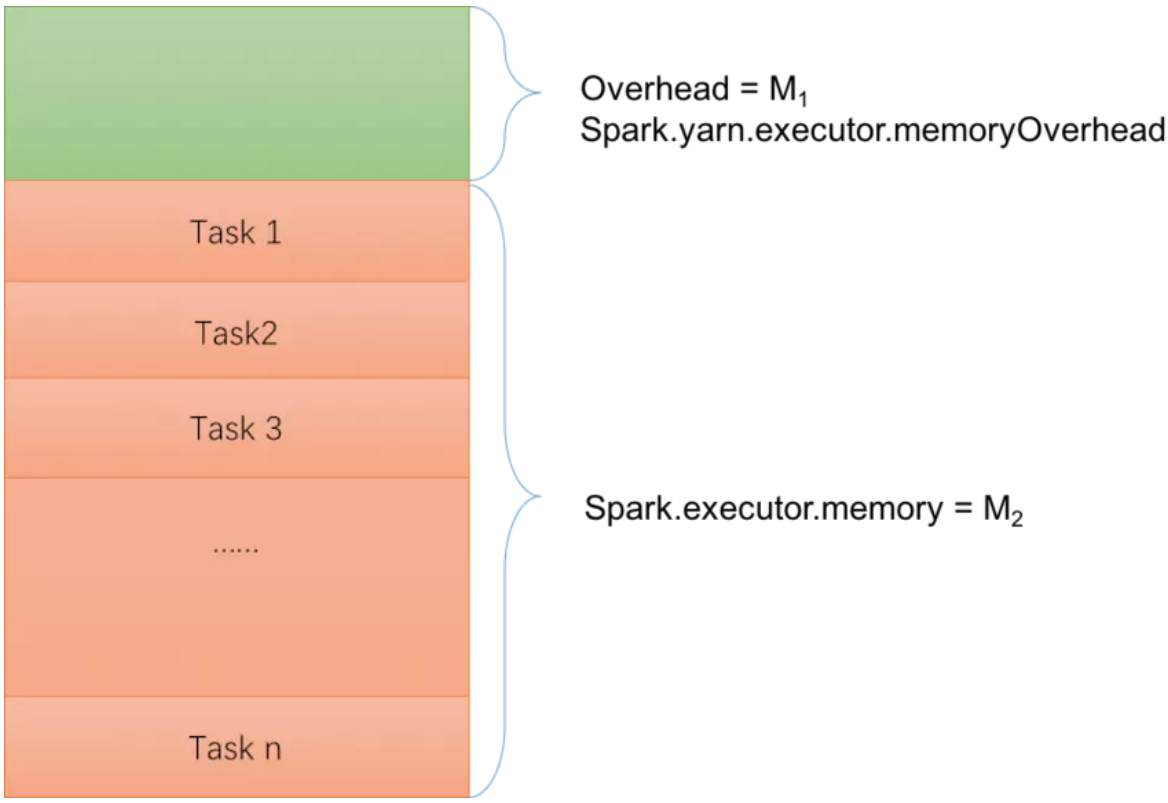
每个 Executor 中可同时运行的任务数由 Executor 分配的 CPU 的核数 N 和每个任务需要的 CPU 核心数 C 决定。其中:

```
N = spark.executor.cores
C = spark.task.cpus
```

由此每个 Executor 的最大任务并行度可表示为： `TP = N / C` 。

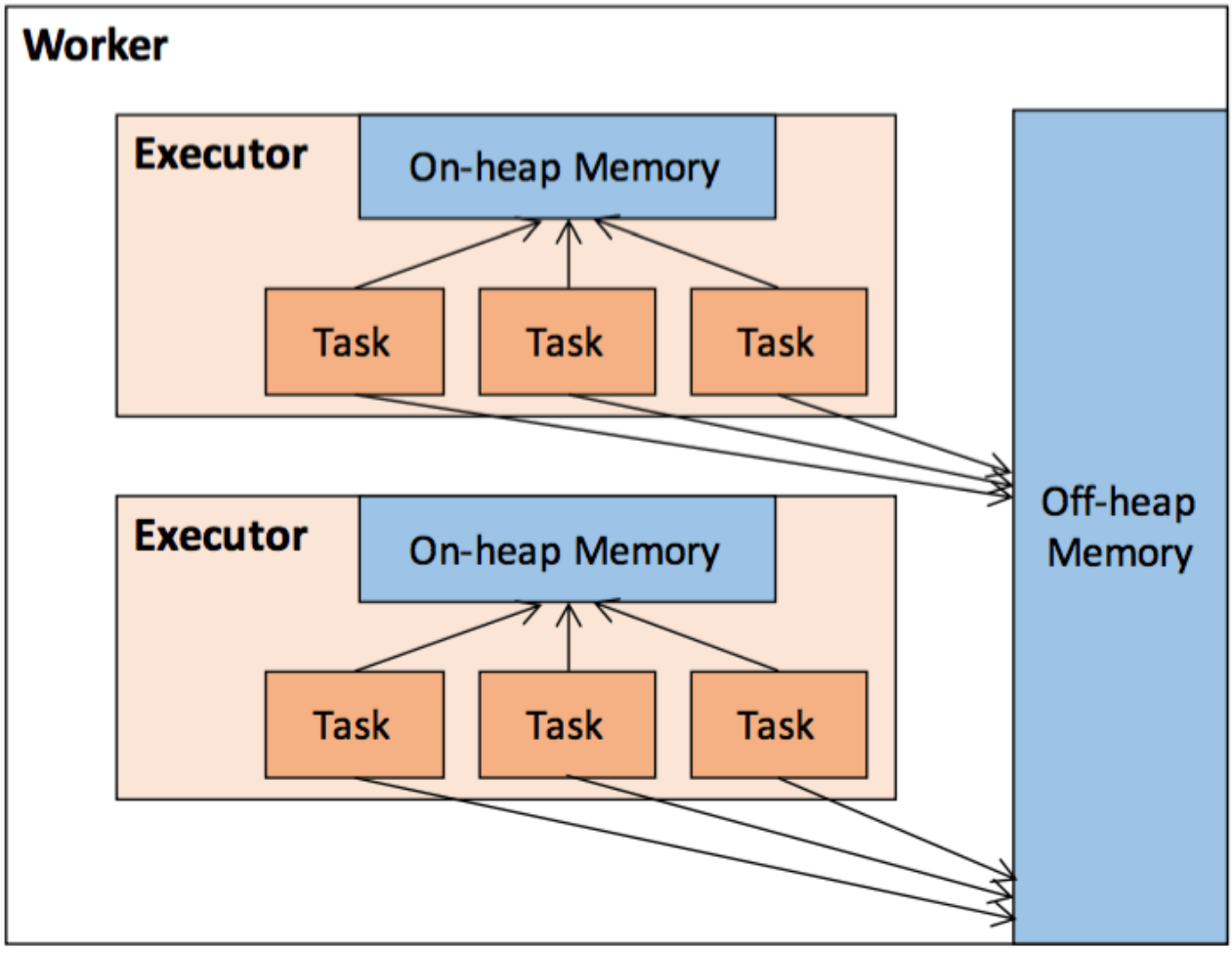
其中，C 值与应用类型有关，大部分应用使用默认值 1 即可，因此，影响 Executor 中最大任务并行度（最大活跃task数）的主要因素是 N。

依据 Task 的内存使用特征，前文所述的 Executor 内存模型可以简单抽象为下图所示模型：



其中，Executor 向 yarn 申请的总内存可表示为：`M = M1 + M2` 。

如果考虑堆外内存则大概是如下结构：



一个示例

为了更好的理解上面堆内内存和堆外内存的使用情况，这里给出一个简单的例子。

只用了堆内内存

现在我们提交的 Spark 作业关于内存的配置如下： `--executor-memory 18g`

由于没有设置 `spark.memory.fraction` 和 `spark.memory.storageFraction` 参数，我们可以看到 Spark UI 关于 Storage Memory 的显示如下：

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	[redacted]	Active	0	0.0 B / 429.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	l-[redacted].qunar.com:55955	Active	0	0.0 B / 10.1 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	l-[redacted].qunar.com:44908	Active	0	0.0 B / 10.1 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

上图很清楚地看到 Storage Memory 的可用内存是 10.1GB，这个数是咋来的呢？根据前面的规则，我们可以得出以下的计算：

```
systemMemory = spark.executor.memory
reservedMemory = 300MB
usableMemory = systemMemory - reservedMemory
StorageMemory= usableMemory * spark.memory.fraction * spark.memory.storageFraction
```

如果我们把数据代进去，得出以下的结果：

```
systemMemory = 18Gb = 19327352832 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 19327352832 - 314572800 = 19012780032
StorageMemory = usableMemory * spark.memory.fraction * spark.memory.storageFraction
              = 19012780032 * 0.6 * 0.5 = 5703834009.6 = 5.312109375GB
```

和上面的 10.1GB 对不上啊。为什么呢？这是因为 Spark UI 上面显示的 Storage Memory 可用内存其实等于 Execution 内存和 Storage 内存之和，也就是 `usableMemory * spark.memory.fraction` ：

```
StorageMemory = usableMemory * spark.memory.fraction
              = 19012780032 * 0.6 = 11407668019.2 = 10.62421GB
```

还是不对，这是因为我们虽然设置了 `--executor-memory 18g` ，但是 Spark 的 Executor 端通过 `Runtime.getRuntime.maxMemory` 拿到的内存其实没这么大，只有 17179869184 字节，所以 systemMemory=17179869184，然后计算的数据如下：

```
systemMemory = 17179869184 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 16865296384
StorageMemory= usableMemory * spark.memory.fraction
             = 16865296384 * 0.6 = 9.42421875 GB
```

我们通过将上面的 16865296384 * 0.6 字节除于 1024 * 1024 * 1024 转换成 9.42421875 GB，和 UI 上显示的还是对不上，这是因为 Spark UI 是通过除于 1000 * 1000 * 1000 将字节转换成 GB，如下：

```
systemMemory = 17179869184 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 16865296384
StorageMemory = usableMemory * spark.memory.fraction
              = 16865296384 * 0.6 字节 =  16865296384 * 0.6 / (1000 * 1000 * 1000) = 10.1GB
```

现在终于对上了。

具体将字节转换成 GB 的计算逻辑如下(core 模块下面的 /core/src/main/resources/org/apache/spark/ui/static/utils.js)：

```
functionformatBytes(bytes, type) {
    if(type !=='display')returnbytes;
    if(bytes == 0)return'0.0 B';
    vark = 1000;
    vardm = 1;
    varsizes = ['B','KB','MB','GB','TB','PB','EB','ZB','YB'];
    vari = Math.floor(Math.log(bytes) / Math.log(k));
    returnparseFloat((bytes / Math.pow(k, i)).toFixed(dm)) +' '+ sizes[i];
}
```

我们设置了 `--executor-memory 18g` ，但是 Spark 的 Executor 端通过 `Runtime.getRuntime.maxMemory` 拿到的内存其实没这么大，只有 17179869184 字节，这个数据是怎么计算的?

`Runtime.getRuntime.maxMemory` 是程序能够使用的最大内存，其值会比实际配置的执行器内存的值小。这是因为内存分配池的堆部分划分为 Eden，Survivor 和 Tenured 三部分空间，而这里面一共包含了两个 Survivor 区域，而这两个 Survivor 区域在任何时候我们只能用到其中一个，所以我

们可以使用下面的公式进行描述：

```
ExecutorMemory = Eden + 2 * Survivor + Tenured
Runtime.getRuntime.maxMemory = Eden + Survivor + Tenured
```

上面的 17179869184 字节可能因为你的 GC 配置不一样得到的数据不一样，但是上面的计算公式是一样的。

用了堆内和堆外内存

现在如果我们启用了堆外内存，情况会怎样呢？我们的内存相关配置如下：

```
spark.executor.memory           18g
spark.memory.offHeap.enabled    true
spark.memory.offHeap.size       10737418240
```

从上面可以看出，堆外内存为 10GB，现在 Spark UI 上面显示的 Storage Memory 可用内存为 20.9GB，如下：

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	[redacted]	Active	0	0.0 B / 11.2 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	l-[redacted]qunar.com:47825	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	l-[redacted].qunar.com:43763	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

其实 Spark UI 上面显示的 Storage Memory 可用内存等于堆内内存和堆外内存之和，计算公式如下：

堆内:

```
systemMemory = 17179869184 字节
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 16865296384
totalOnHeapStorageMemory = usableMemory * spark.memory.fraction
                         = 16865296384 * 0.6 = 10119177830
```

堆外:

```
totalOffHeapStorageMemory = spark.memory.offHeap.size = 10737418240
```

总 Storage 内存:

```
StorageMemory = totalOnHeapStorageMemory + totalOffHeapStorageMemory
              = (10119177830 + 10737418240) 字节
              = (20856596070 / (1000 * 1000 * 1000)) GB
              = 20.9 GB
```

Executor内存参数调优

1. Executor JVM Used Memory Heuristic

现象：配置的executor内存比实际使用的JVM最大使用内存还要大很多。

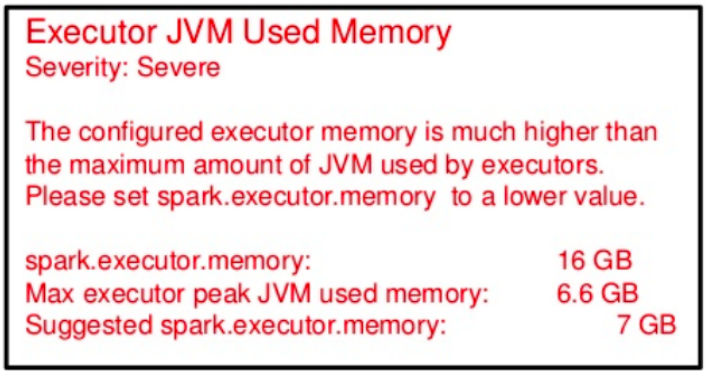
原因：这意味着 executor 内存申请过多了，实际上并不需要使用这么多内存。

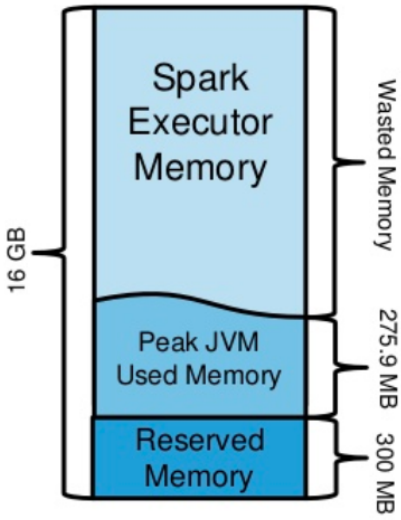
解决方案：将 `spark.executor.memory` 设置为一个比较小的值。

例如：

```
spark.executor.memory : 16 GB
Max executor peak JVM used memory : 6.6 GB

Suggested spark.executor.memory : 7 GB
```





2. Executor Unified Memory Heuristic

现象：分配的统一内存 (`Unified Memory = Storage Memory + Execution Memory`) 比 executor 实际使用的统一内存大的多。

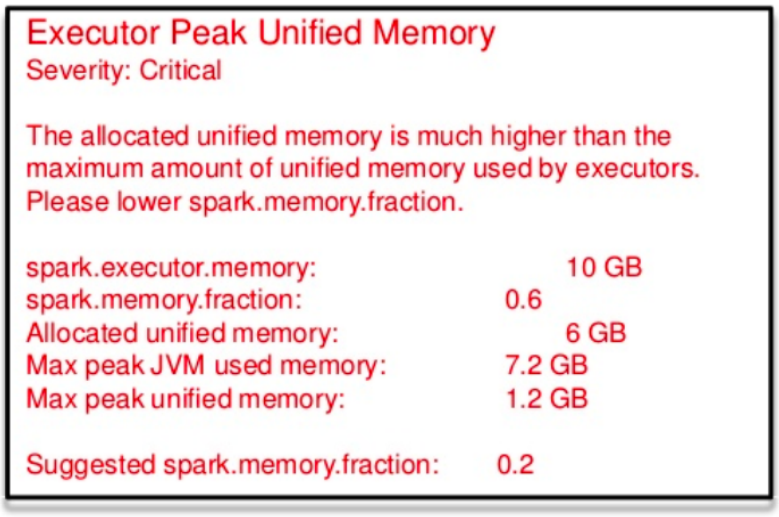
原因：这意味着不需要这么大的统一内存。

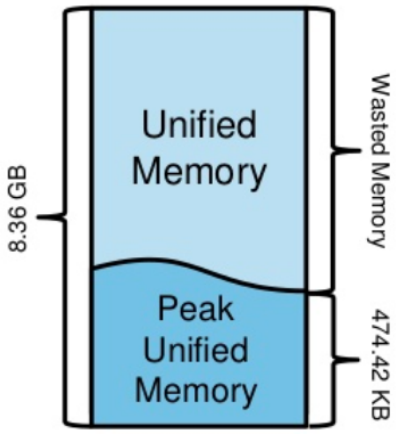
解决方案：降低 `spark.memory.fraction` 的比例。

例如：

```
spark.executor.memory : 10 GB
spark.memory.fraction : 0.6
Allocated unified memory : 6 GB
Max peak JVM userd memory : 7.2 GB
Max peak unified memory : 1.2 GB

Suggested spark.memory.fraction : 0.2
```





3. Executor OOM类错误 （错误代码 137、143等）

该类错误一般是由于 Heap（M2）已达上限，Task 需要更多的内存，而又得不到足够的内存而导致。因此，解决方案要从增加每个 Task 的内存使用量，满足任务需求 或 降低单个 Task 的内存消耗量，从而使现有内存可以满足任务运行需求两个角度出发。因此有如下解决方案:

法一：增加单个task的内存使用量

- 增加最大 Heap值，即上图中 M2 的值，使每个 Task 可使用内存增加。
- 降低 Executor 的可用 Core 的数量 N , 使 Executor 中同时运行的任务数减少，在总资源不变的情况下，使每个 Task 获得的内存相对增加。当然，这会使得 Executor 的并行度下降。可以通过调高 `spark.executor.instances` 参数来申请更多的 executor 实例（或者通过 `spark.dynamicAllocation.enabled` 启动动态分配），提高job的总并行度。

法二： 降低单个Task的内存消耗量

降低单个Task的内存消耗量可从配置方式和调整应用逻辑两个层面进行优化:

一、配置方式

减少每个 Task 处理的数据量，可降低 Task 的内存开销，在 Spark 中，每个 partition 对应一个处理任务 Task，因此，在数据总量一定的前提下，可以通过增加 partition 数量的方式来减少每个 Task 处理的数据量，从而降低 Task 的内存开销。针对不同的 Spark 应用类型，存在不同的 partition 配置参数 :

```
P = spark.default.parallism (非SQL应用)
P = spark.sql.shuffle.partition (SQL 应用)
```

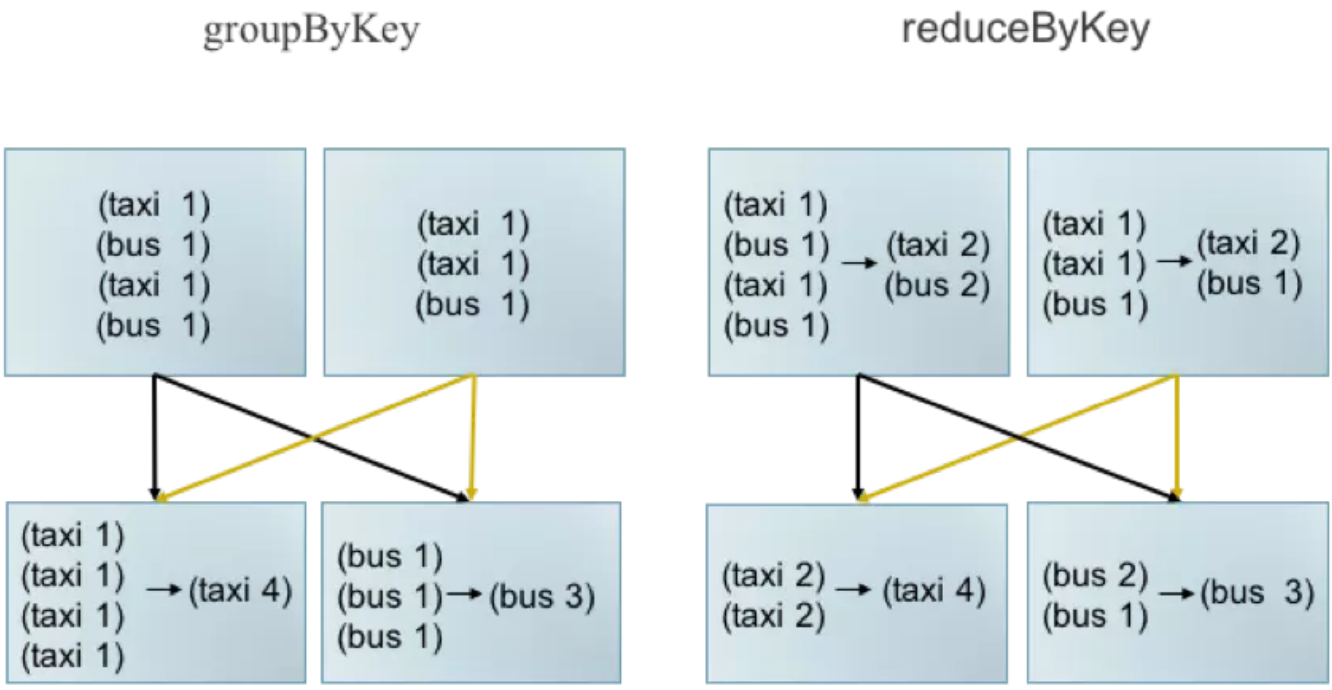
通过增加 P 的值，可在一定程度上使 Task 现有内存满足任务运行。注: 当调整一个参数不能解决问题时，上述方案应进行协同调整。

二、调整应用逻辑

Executor OOM 一般发生 Shuffle 阶段，该阶段需求计算内存较大，且应用逻辑对内存需求有较大影响，下面举例就行说明:

1、选择合适的算子，如 groupByKey 转换为 reduceByKey

一般情况下，groupByKey 能实现的功能使用 reduceByKey 均可实现，而 ReduceByKey 存在 Map 端的合并，可以有效减少传输带宽占用及 Reduce 端内存消耗。

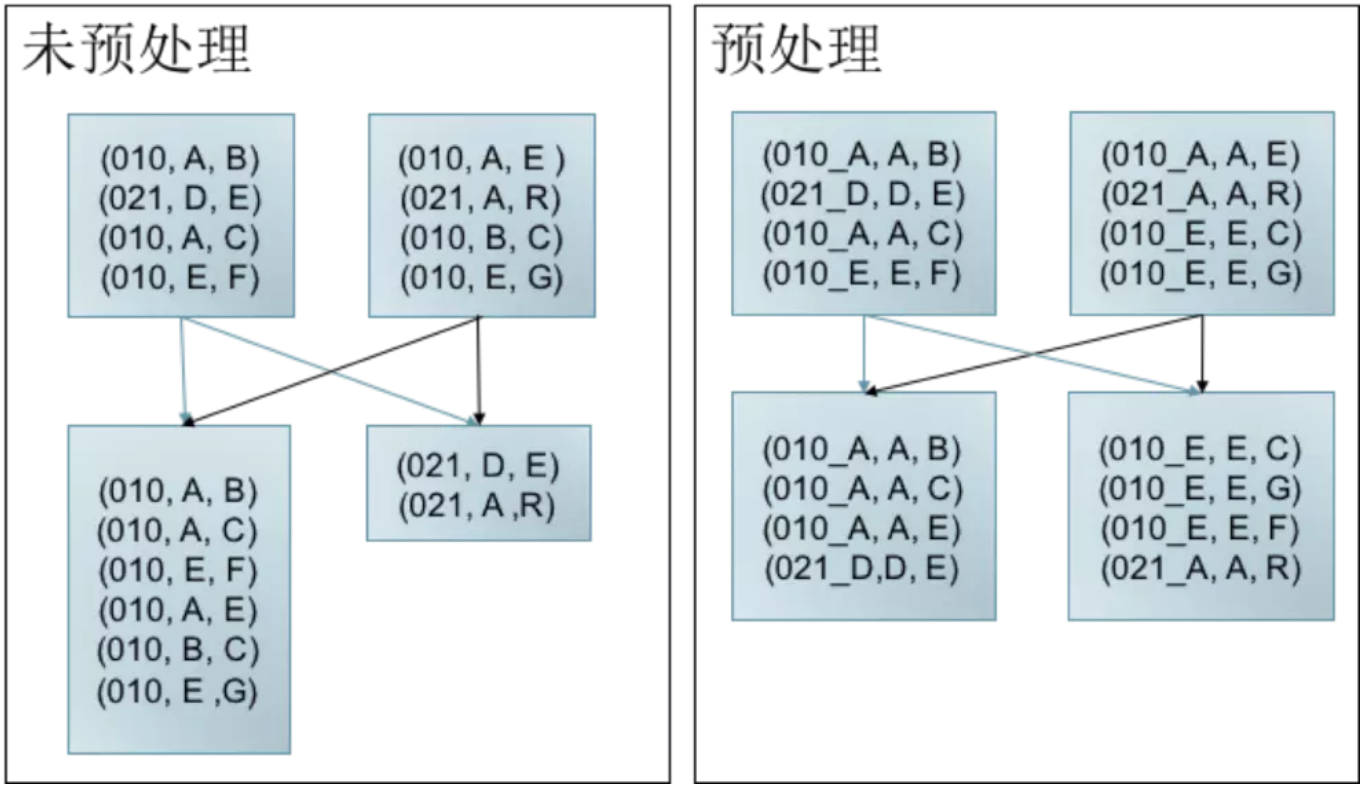


2、避免数据倾斜 (data skew)

Data Skew 是指任务间处理的数据量存大较大的差异。

如左图所示，key 为 010 的数据较多，当发生 shuffle 时，010 所在分区存在大量数据，不仅拖慢 Job 执行（Job 的执行时间由最后完成的任务决定）。 而且导致 010 对应 Task 内存消耗过多，可能导致 OOM。

而右图，经过预处理（加盐，此处仅为举例说明问题，解决方法不限于此）可以有效减少 Data Skew 导致的问题。



NOTE

上述举例仅为说明调整应用逻辑可以在一定程序上解决OOM问题，解决方法不限于次。

4. Execution Memory Spill Heuristic

现象: 在 stage 3 发现执行内存溢出。Shuffle read bytes 和 spill 分布均匀。这个 stage 有 200 个 tasks。

原因: 执行内存溢出，意味着执行内存不足。跟上面的 OOM 错误一样，只是执行内存不足的情况下不会报 OOM 而是会将数据溢出到磁盘。但是整个性能很难接受。

解决方案: 同 3。

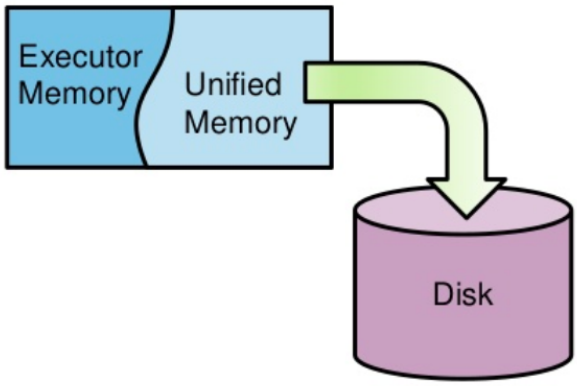
Execution Memory Spill
Severity: Severe

Execution memory spill has been detected in stage 3. Shuffle read bytes and spill are evenly distributed. There are 200 tasks for this stage. Please increase spark.sql.shuffle.partitions, or modify the code to use more partitions, or reduce the number of executor cores.

spark.executor.memory	10 GB
spark.executor.cores	3
spark.executor.instances	300

Stage 3:

Median shuffle read bytes:	954 MB
Max shuffle read bytes:	955 MB
Median shuffle write bytes:	359 MB
Max shuffle write bytes:	388 MB
Median memoryBytesSpilled:	1.2 GB
Max memoryBytesSpilled:	1.2 GB
Num tasks:	200



4. Executor GC Heuristic

现象: Executor 花费很多时间在 GC。

原因: 可以通过 `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps` 查看 GC 情况

解决方案: Garbage Collection Tuning (https://spark.apache.org/docs/2.1.0/tuning.html#garbage-collection-tuning)

5. Beyond … memory, killed by yarn.

出现该问题原因是由于实际使用内存上限超过申请的内存上限而被 Yarn 终止掉了, 首先说明 Yarn 中 Container 的内存监控机制：

- Container 进程的内存使用量 : 以 Container 进程为根的进程树中所有进程的内存使用总量。
- Container 被杀死的判断依据 : 进程树总内存（物理内存或虚拟内存）使用量超过向 Yarn 申请的内存上限值，则认为该 Container 使用内存超量，可以被“杀死”。

因此，对该异常的分析要从是否存在子进程两个角度出发。

1、不存在子进程

根据 Container 进程杀死的条件可知，在不存在子进程时，出现 killed by yarn 问题是于由 Executor(JVM) 进程自身内存超过向 Yarn 申请的内存总量 M 所致。由于未出现上一节所述的 OOM 异常，因此可判定其为 M1 (Overhead) 不足，依据 Yarn 内存使用情况有如下两种方案:

法一、如果，M (`spark.executor.memory`) 未达到 Yarn 单个 Container 允许的上限时，可仅增加 M1（ `spark.yarn.executor.memoryOverhead` ），从而增加 M；如果，M 达到 Yarn 单个 Container 允许的上限时，增加 M1，降低 M2。

注意二者之各要小于 Container 监控内存量，否则伸请资源将被 yarn 拒绝。

法二、减少可用的 Core 的数量 N，使并行任务数减少，从而减少 Overhead 开销

2、存在子进程

Spark 应用中 Container 以 Executor（JVM进程）的形式存在，因此根进程为 Executor 对应的进程，而 Spark 应用向Yarn申请的总资源 `M = M1 + M2` ，都是以 Executor(JVM) 进程（非进程树）可用资源的名义申请的。申请的资源并非一次性全量分配给 JVM 使用，而是先为 JVM 分配初始值，随后内存不足时再按比率不断进行扩容，直致达到 Container 监控的最大内存使用量 M。当 Executor 中启动了子进程（如调用 shell 等）时，子进程占用的内存（记为 S）就被加入 Container 进程树，此时就会影响 Executor 实际可使用内存资源（Executor 进程实际可使用资源变为: `M - S` ），然而启动 JVM 时设置的可用最大资源为 M，且 JVM 进程并不会感知 Container 中留给自己的使用量已被子进程占用，因此，当 JVM 使用量达到 `M - S` ，还会继续开劈内存空间，这就会导致 Executor 进程树使用的总内存量大于 M 而被 Yarn 杀死。

典形场景有:

1. PySpark（Spark已做内存限制，一般不会占用过大内存）
2. 自定义Shell调用

其解决方案分别为:

1) PySpark场景:

- 如果，M 未达到 Yarn 单个 Container 允许的上限时，可仅增加 M1 ，从而增加 M；如果，M 达到 Yarn 单个 Container 允许的上限时，增加 M1，降低 M2。
- 减少可用的 Core 的数量 N，使并行任务数减少，从而减少 Overhead 开销

2) 自定义 Shell 场景:（OverHead 不足为假象）

调整子进程可用内存量 (通过单机测试，内存控制在 Container 监控内存以内，且为 Spark 保留内存等留有空间）。

推荐阅读

1. Apache Spark 内存管理详解 (https://www.ibm.com/developerworks/cn/analytics/library/ba-cn-apache-spark-memory-management/index.html?ca=drs-&utm_source=tuicool&utm_medium=referral)
2. Spark on Yarn之Executor内存管理 (https://www.jianshu.com/p/10e91ace3378)
3. Apache Spark 统一内存管理模型详解 (https://toutiao.io/posts/9x4hj5/preview)
4. Spark 内存管理之UnifiedMemoryManager (https://blog.csdn.net/u011564172/article/details/71170151)
5. Tuning Spark-Memory Management Overview (https://spark.apache.org/docs/2.1.0/tuning.html#memory-management-overview)

PREVIOUS
SPARK RDD **(/SPARK-RDD.HTML)**

NEXT
SPARK数据倾斜及其解决方案 **(/SPARK-DATA-SKEW.HTML)**

YOU MIGHT ALSO LIKE

26 Dec 2018 » 创建Hadoop FileSystem报Provider org.apache.hadoop.fs.azure.NativeAzureFileSystem not a subtype异常 (http://arganzheng.life/provider-org-apache-hadoop-fs-azure-nativeazurefilesystem-not-a-subtype.html)
24 Nov 2018 » Spark数据倾斜及其解决方案 (http://arganzheng.life/spark-data-skew.html)
12 Nov 2018 » Spark RDD (http://arganzheng.life/spark-rdd.html)