# Apache Spark调优（Tuning Spark）

由于Spark基于内存计算的特性，集群的任何资源都可以成为Spark程序的瓶颈:CPU，网络带宽，或者内存。通常，如果内存容得下数据，瓶颈会是网络带宽。不过有时你同样需要做些优化，例如将RDD以序列化到磁盘，来降低内存占用。这个教程会涵盖两个主要话题：数据序列化，它对网络性能尤其重要并可以减少内存使用，以及内存调优。另外我们也会简述几个小一点的话题。

## 数据序列化

序列化在分布式应用中扮演着重要的角色。序列化对象耗时长，或者占据大量空间的格式，会显著地拖慢计算。通常这会是进行Spark应用调优时你需要做的第一件事。Spark的目标是找到易用性(允许你在操作中使用任何Java类型)以及性能间的平衡。它提供了两个序列化库:

- Java序列化: 默认情况下，Spark使用Java的 ObjectOutputStream 框架序列化对象，可以处理你创建的实现了 java.io.Serializable 接口的所有Class。你也可以通过继承 java.io.Externalizable 来更紧密地控制序列化性能。Java序列化灵活但是通常很慢，而且对于很多Class来说序列化结果比较大。
- Kryo序列化: Spark可以使用Kryo库(第2版本)来进行更快速的序列化。Kryo比起Java序列化显著地更快而且结果空间利用更紧凑(通常是10倍以上)，但为了更好的性能，不支持所有Serializable的类型，而要求你提前注册将要在程序中使用的Class。

通过用SparkConf初始化任务并调用 conf.set(“spark.serializer”, “org.apache.spark.serializer.KryoSerializer”)， 你可以切换到Kryo序列化。这个配置设定的序列化框架不仅用于worker节点间的shuffling，也用于将RDD持续化到磁盘。不将Kryo作为默认序列化框架的唯一原因是它要求特定的注册步骤，但我们推荐在网络资源紧张的情况下使用它。

Spark为包含在 Twitter chill 的 AllScalaRegistar 库中的许多常用的Scala Class自动加载Kryo序列化框架。使用 registerKryoClasses 方法来注册你自定义的Class。

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

Kryo官方文档描述了更多高级的注册选项，比如定制序列化代码。如果对象很大，你或许需要提高 spark.kryoserializr.buffer 配置。这个值需要足够大，可以放得下你需要序列化的最大的对象。最后，如果你不注册你定制的对象，Kryo仍然可以运作，但是它必须在每个对象存储全类名，这是很浪费的。

## 内存调优

内存调优主要有三个方面的考虑:对象使用的内存大小(你可能想要整个数据集都加载到内存)，访问这些对象的成本，还有垃圾回收的消耗(如果你需要大批量地创建和销毁对象)。
默认情况下，Java对象访问快，但每个属性很容易消耗原始数据2-5倍的空间。这是由几个原因造成的:

- 每个不同的Java对象都有一个“对象头”，大约16字节，包含了Class的指针等信息。对于只存储了少量数据的对象（比如只有一个int属性），对象头可能会大于数据。
- Java String对象有比起原始字符串需要多消耗40字节（因为数据以一个Char数组形式存放，要保存长度等额外的数据），并且每个字符是2个字节，因为String类型使用了UTF-16编码。所以一个10字符的字符串很容易就消耗了60字节。
- 普遍的集合类，比如 HashMap 和 LinkedList，使用了链接形式的数据结构，这种数据结构每个成员都有一个“包装器”对象（比如 Map.Entry）。这个对象不仅有一个头部，而且有一个指向下个成员的指针（典型的是8字节）。
- 原生类型的集合通常存储为“盒子”对象，比如 java.lang.Integer。

本章节会由Spark的内存管理概览开始，然后讨论一些特别的策略，这些策略用户可以在他或她的应用中更有效率地使用内存。特别地，我们会描述怎么样确定对象的内存使用和怎么去优化——通过改变你的数据结构，或者通过将数据以序列化方式存储。接下来我们再讨论Spark缓存优化和Java垃圾回收。

## 内存管理概览

Spark的内存使用分为两类：执行内存和存储内存。执行内存用于洗牌(shuffle)，连接(join)，排序(sort)和聚合(aggregation)，而存储内存指用于缓存和传输集群内部数据的内存。在Spark中，执行和存储共享同一内存区域（M区）。当不需要使用执行内存时，存储可以占据整个区域，反之亦然。需要的时候执行内存可能会驱逐存储内存，直到所有的存储内存使用降到某个阈值以下（R区）。换句话说，R区是M区的一个子分区，在这个分区内的缓存块不会被驱逐出内存。存储内存不会驱逐执行内存，否则实现就会过于复杂。

这样设计确保了几个重要的特性。第一，不需要缓存空间的应用可以使用整个内存作为执行内存，避免了无谓的内存溢出写到磁盘操作。第二，需要缓存空间的应用可以保留一个最小的存储空间（R区），这个空间内的数据块不会被驱逐出内存。最后，这个机制在不同工作负荷下提供了合理的“盒外”性能，不需要用户研究Spark内部怎么划分内存。

Spark提供了两个相关配置，但一般用户不应该需要调整它们，因为默认值在大多数情况都可以满足要求：

- spark.memory.fraction 表示M区占据整个JVM堆（300MB）的比例，默认为0.6。留下40%的空间给用户数据结构，Spark内部元数据，以及防止过大记录导致OOM的预留空间。
- spark.memory.storageFraction 表示R区占据M区空间的比例，默认为0.6。R区是M区中的存储区域，其中缓存的数据块不会被执行内存抢占。

spark.memory.fraction
的值应该被设置在JVM的老生代或者永生代的空间大小以内。详情参考进阶下文的GC调优讨论。

## 计算内存消耗

计算一个数据集需要的内存大小的最好的方式是，创建一个RDD并把它放进缓存，查看Spark Web界面的”Storage”页面。这个页面会告诉你这个RDD占用了多少内存。如果要估算某个对象的内存消耗，可以使用 SizeEstimator 的 estimate 方法。这无论在试验不同数据布局以减少内存使用的时候，还是在计算一个广播变量对于每个executor堆空间的占用大小的时候，都非常有用。

## 数据结构调优

减少内存消耗的第一个方式就是避免使用增加内存成本的Java特性，比如基于指针的数据结构和包装类。以下是几个具体的做法:

1. 设计数据结构时优先使用对象数组和基本类型，而不是Java和Scala的标准集合(比如HashMap)。Fastutil 库为基本类型提供了方便的集合类，并与Java标准库兼容。
2. 尽量避免包含大量小对象和指针的复杂数据结构。
3. 考虑使用数值类型的id或者枚举类型的key，避免使用字符串作为key。
4. 如果内存不足32GB，设置JVM选项 -XX:+UseCompressedOops
   将指针由默认8字节改为4字节。你可以将这些选项加到 spark-en.sh 中。

## RDD序列化存储

当进行上述优化之后，对象仍旧太大而无法有效存储时，减少内存使用的一个更简单的办法是将它们序列化到磁盘存储，具体办法是使用RDD持续化API的序列化存储级别，比如 MEMORY_ONLY_SER。之后Spark会将每个RDD的分区存为一个大字节数组。这样做唯一的缺点是降低了访问效率，因为需要在访问时进行反序列化。如果你打算以序列化方式缓存数据，我们强烈推荐使用Kryo，因为它序列化的结果比较Java默认序列化小很多(当然也比原始Java对象小)。

## 垃圾回收(GC)调优

如果你在程序需要大量新建和销毁RDD操作的时候，JVM垃圾回收会成为一个问题(只是读取一个RDD然后操作多次不会产生这个问题)。Java需要将旧对象驱逐出内存来容纳新的对象，这时它会追踪所有的Java对象，找出其中不在使用的部分。这里的关键是垃圾回收的成本是和Java对象的数量成正比的，所以使用包含少量对象的数据结构(比如整形数组而不是 LinkedList)会显著减少这项成本。一个更好的办法是把对象序列化，就像上面描述的一样:这样每个RDD分区只有一个对象(一个字节数组)。在使用其他优化技巧之前，首先要尝试使用序列化存储来确定问题是不是出在垃圾回收上。
垃圾回收会造成问题的另外一种原因是任务的工作内存(执行任务需要的内存大小)和缓存在节点上的RDD竞争内存。我们接下来会讨论怎么通过控制给RDD分配的空间来减轻这种情况。

### 度量GC的影响

GC调优的第一步是收集数据，包括GC多久发生一次和花在GC上的时间。具体可以通过增加Java参数 -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps。(详情见[配置教程]中的传递Java参数给Spark任务。)下次你的Spark任务执行的时候，你会看见每次GC发生时信息都会打印到worker的日志里面。注意这些日志会在集群的worker节点(在工作目录的stdout文件里)，而不是你的

Driver节点。

进阶GC调优

为了进一步调优GC，我们首先需要理解一些关于JVM内存管理的基本信息:

- Java堆空间分为新生代分区和老年代分区。新生代用于存放短可以生命周期的对象，而老年代则为长时间生存的对象准备。
- 新生代进一步可以分为三个分区[Eden区，Survivor1，Survivor2]。
- GC进程的简要描述:当Eden区满了，一个 minor GC会在Eden区启动，仍然生存的对象会从Eden取和Survivor1区拷贝到Survivor2区。两个Survivor区交换。如果一个对象足够老，或者Survior2满了，它会移到老年代分区。最后当老年代分区接近用满，full GC(全局GC)就会启动。

Spark GC调优的目标是确保只有长期生存的RDD存放在老年代区，而且新生代区足够存放短期生存的对象。这样就避免了启动full
GC来收集任务执行时创建的临时对象。一些可能有帮助的步骤如下:

- 收集GC统计数据，检查是否有过多的GC。如果在任务完成前full GC发生了多次，这意味着没有足够多的可用内存提供给该任务。
- 如果有很多minor GC却没有很多major GC，分配更多内存给Eden区可以改善这个问题。你可以将Eden区的大小调为高于任务所需内存。如果Eden区的小大为E，你可以通过参数 -Xmn=4/3*E 设置新生代的大小。(增大为4/3倍是因为Survivor分区也需要空间。)
- 在打印出来的GC统计信息中，如果老生代接近用满，降低 spark.memory.fraction 以减少用于缓存RDD的空间。缓存少一点对象总比拖慢任务执行要好。或者考虑减小新生代分区的大小也是可以的。这意味着将 -Xmn 调低，如果你已经按照上文做了的话。如果没有的话，尝试改变JVM的NewRatio参数。许多JVM默认将此参数设为2，意味这老年代占据堆大小的2/3。这个值应该要超过 spark.memory.fraction。
- 尝试设置参数 -XX:+UseG1GC 应用G1GC垃圾收集器。在某些情况下，当垃圾回收成为瓶颈时它可以提高性能。注意在堆空间比较大的情况下，使用 –XX:G1HeapRegionSize 参数提高G1分区大小是很重要的。
- 举例，如果你的任务从HDFS读取数据，可以通过读取的数据块大小估算任务使用的内存大小。注意解压后的数据块大小通常是原来大小的2至3倍。所以如果希望给3或4个任务分配工作空间，而且HDFS块大小为128M，我们可以估计Eden区大概需要 43128MB 的空间。
- 更新设置后，监控GC的频率以及使用时间的变化。

根据经验我们认为GC调优的效果取决于具体应用和可用内存大小。网上有更多关于调优参数的描述，不过是高层的调优，以管理full
GC发生的频率来帮助减轻GC成本。Executor的GC调优可以通过设置任务配置中的 spark.executor.extraJavaOptions 来指定。

## 其他考虑因素

### 并行度

除非你为每个操作设置足够高的并行度，否则集群资源不会被充分利用。Spark自动根据文件大小决定启动多少个map任务(虽然你可以通过可选参数或者SparkContext.textFile等方法控制)。对于分布式的reduce操作，比如 groupByKey 和 reduceByKey，并行度是最大父RDD的分区数。你可以将并行度作为第二参数传递给Spark(见spark.PairRDDFunctions文档)，或者设置配置属性 spark.default.parallelism来改变默认值。总的来说，我们推荐集群内每个CPU执行2至3个任务。

## Reduce任务的内存使用

有时发生内存溢出错误并不是因为内存放不下RDD，而是其中一个task处理的数据集太大了，比如在 groupByKey 操作中就可能出现这种情况。Spark的shuffle操作(sortByKey， groupByKey，reduceByKey， join等等)为了进行分组操作，在每个task中都新建了一个哈希表。最简单的办法是提高并行度，然后每个task的输入都会变小。Spark执行时长200ms的短任务很有效率，因为它在许多task中复用了一个executor JVM，并且每个task的启动成本都较低，所以你可以安全地将并行度提高到集群cpu核数以上。

## 广播大变量

使用SparkContext中的广播函数可以显著减小每个序列化task的大小，还有在集群上启动任务的成本。如果任务需要从Driver程序获取大对象(比如静态的扫表)，你可以考虑将这个对象转变为广播变量。Spark将每个task序列化后的大小打印在master上，你可以根据这个来判断task是不是太大。通常来说task大于20KB就可能需要优化。

## 数据本地性

数据本地性对Spark任务的性能有重要影响。如果数据和处理它的代码在一起，计算会快一些。不过如果数据和代码是分开的，那么其中一个必须移动到另外一个那里。一般来说，移动序列化后的代码比移动一个数据块要快，因为代码远比数据小。Spark围绕数据本地性的一般原则来建立调度策略。
数据本地性指数据离对应的代码多近。有几个基于数据当前位置的本地性，从近到远如下:

- PROCESS_LOCAL(进程本地) 数据就在代码所在的JVM里，这是最好的数据本地性。
- NODE_LOCAL(节点本地) 数据在同一个节点上。例如在同一个节点的HDFS上，或者在同一节点的另外一个executor上。这比PROCESS_LOCAL稍慢，因为数据要跨进程传输。
- NO_PREF(无偏好) 数据在所有节点的访问都一样快，没有本地性偏好。
- RACK_LOCAL(机架本地)
  数据在同一个机架的节点上。数据在相同机架的不同节点上所以需要网络传输。
- ANY(任何) 数据在网络上其他地方，但不在同一机架上。

Spark倾向于以最好的数据本地性调度任务，但并不总能做到。在所有空闲的executor上都没有未处理数据的情况下，Spark将本地性要求放低。有两个方法：a)等待某个在工作的CPU空闲下来，可以在同一节点启动新的任务。b)马上在远端的节点启动新任务，但是需要将数据传输过去。

Spark典型的处理是等一会，希望有CPU资源释放。一旦等待超时，它开始将数据移动到远端的空闲CPU。不同级别间的等待时长可以分别设置或者同一设置。详情见配置页面的 spark.locality 参

数。如果任务执行时间长或数据的本地性差，你应该调高这些时长，不过默认值同样也可以工作。

## 总结

这是一个简短的教程，指出了调优Spark时你应该知道的主要的关注点–最重要的是，数据序列化和内存调优。对于大多数程序来说，切换到Kryo序列化和将数据序列化存储会解决大部分常见的性能问题。欢迎在Spark邮件列表提出关于其他调优经验的问