|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 修订者 | 修订描述 |
| 2018.07.26 | v1 | 林 清 | 初稿 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[1. Spark内存调优 2](#_Toc520484476)

[1.1 Spark Heap内存规划 2](#_Toc520484477)

[1.1.1 静态管理 2](#_Toc520484478)

[1.1.2 动态管理 3](#_Toc520484479)

[1.2 Spark堆外内存 4](#_Toc520484480)

[1.3 GC参数调整 5](#_Toc520484481)

[2. 其他调优手段 5](#_Toc520484482)

[2.1 分区数 5](#_Toc520484483)

[2.2 spark.driver.maxResultSize 6](#_Toc520484484)

[2.3 序列化接口优化 6](#_Toc520484485)

[2.4 数据压缩 6](#_Toc520484486)

# Spark内存调优

## Spark Heap内存规划

Spark中一个Executor对应一个JVM进程，Executor占用的内存分为两部分：ExecutorMemory和MemoryOverhead。

ExecutorMemory是堆区内存，MemoryOverhead是Spark的堆外内存。

下面介绍一下Spark任务的堆区内存机构。

Heap内存大小通过属性**spark.executor.memory**设置，也可以在spark-submit命令时用参数**--executor-memory**设置。

在Spark1.6以前的版本中，heap内存是**静态管理**的，而新版中内存使用**动态管理**方案进行管理。通过配置项**spark.memory.useLegacyMode**可以在两种方式中进行切换。

### 静态管理

静态方案中Spark将一个Executor的Heap空间分为**多个固定大小的内存区域**，并且用作不同用途。通过配置参数中的比例因子，能够调整这些区域的比例。

#### 安全空间预留

**spark.executor.memory x (1-spark.storage.safetyFraction)**

这一部分内存是Spark预留的内存，这一部分内存不会被Spark使用。

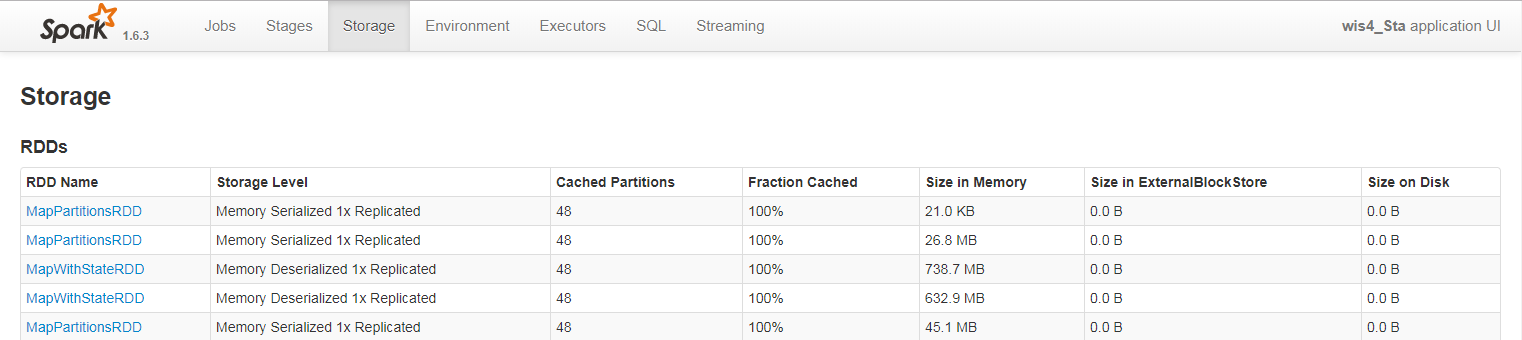
#### 缓存空间 (Storage)

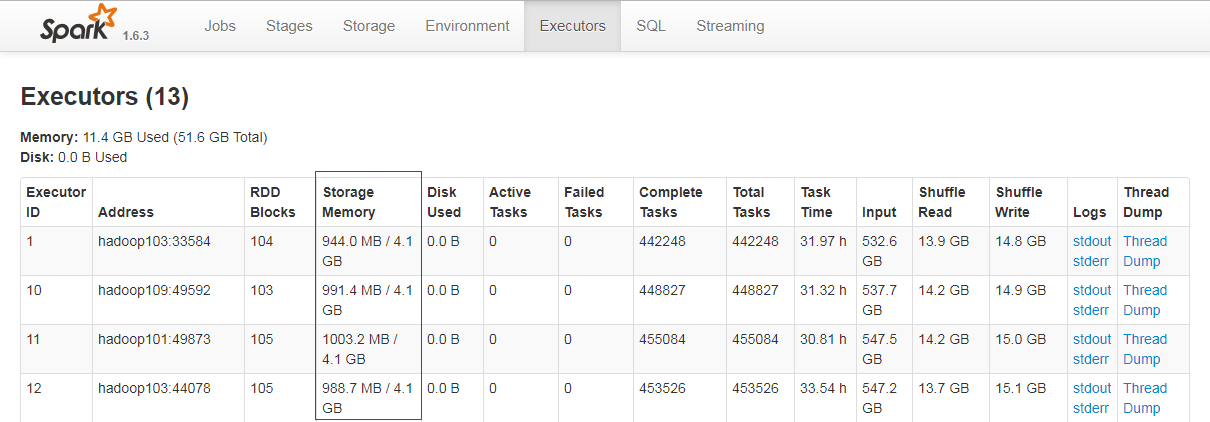
**spark.executor.memory x spark.storage.safetyFraction x spark.storage.memoryFraction**

Spark任务缓存RDD数据 时使用，占比通过spark.storage.memoryFraction配置，默认为0.6；

当Spark使用完了缓存空间时，应用根据用户代码中的持久化策略，Spark可能将之前的数据spill到磁盘，或者直接删除。

一个Executor任务缓存了多少数据、哪些数据都可以在Spark UI中查看，如下图：





#### **Unroll 空间**

**spark.executor.memory x spark.storage.safetyFraction x spark.storage.memoryFraction x spark.storage.unrollFraction**

spark.storage.unrollFraction**的默认大小为0.2**

这一部分内存使用被Spark用于序列化和反序列化的。

#### **Shuffle 空间**

**spark.executor.memory x spark.shuffle.memoryFraction x spark.shuffle.safteyFraction**

spark.shuffle.safteyFraction的大小为0.8不可配置，**spark.shuffle.memoryFraction**默认大小为0.2。

如果按照上述默认配置，每个Spark中用户用于运行代码的内存大小实际上只有：

**(1-0.6-0.2) x 0.9 x spark.executor.memory= spark.executor.memory x 18%**

### 动态管理

Spark1.5以上版本提供了内存动态管理的配置，将内存划分成下面的三个部分：

#### Reserved Memory

这部分内存是预留给系统使用，是固定不变的。

在1.6.0默认为300MB（RESERVED\_SYSTEM\_MEMORY\_BYTES = 300 \* 1024 \* 1024）。这一部分内存不计算在spark execution和storage中。

除了重新编译spark和spark.testing.reservedMemory配置项，Reserved Memory是不可以改变的，spark.testing.reservedMemory不推荐使用在实际运行环境中。

如果executor的大小小于1.5 \* Reserved Memory = 450MB ，那么就会报 “please use larger heap size”的错误。

#### User Memory

写 Spark 程序中产生的临时数据或者是自己维护的一些数据结构也需要给予它一部份的存储空间，这是程序运行时用户可以主导的空间，叫用户操作空间。

它占用的空间是：

**(Java Heap - Reserved Memory) x （1-spark.memory.fraction）**

**spark.memory.fraction**默认是0.6，可以有参数供调优，这样设计可以让用户操作时所需要的空间与系统框架运行时所需要的空间分离开。

#### Spark Memory

系统框架运行时需要使用的空间，这是从两部份构成的分别是 Storage Memeory 和 Execution Memory。前者用来进行RDD缓存，后者用来Shuffle缓存。

Storage 和 Execution (Shuffle) 采用了 Unified 的方式共同使用一个内存区域

**(Heap Size - 300MB) x spark.memory.fraction**

默认情况下 Storage 和 Execution 各占该空间的 50%。

Storgae 和 Execution 在适当时候可以借用彼此的 Memory，需要注意的是，当 Execution 空间不足而且 Storage 空间也不足的情况下，Storage 空间如果曾经使用了超过 Unified 默认的 50% 空间的话则超过部份会被强制 drop 掉一部份数据来解决 Execution 空间不足的问题 (注意：drop 后数据会不会丢失主要是看你在程序设置的 storage\_level 来决定你是 Drop 到那里，可能 Drop 到磁盘上)，这是因为执行(Execution) 比缓存 (Storage) 是更重要的事情。

## Spark堆外内存

Spark的堆外内存称为Memory-Overhead是JVM进程中除Java堆以外占用的空间大小，包括方法区（永久代）、Java虚拟机栈、本地方法栈、JVM进程本身所用的内存、直接内存（DirectMemory）等。通过**spark.yarn.executor.memoryOverhead**设置，单位MB。

默认情况下，spark.yarn.executor.memoryOverhead的大小按照下面的方式决定：

MEMORY\_OVERHEAD\_FACTOR = 0.07

MEMORY\_OVERHEAD\_MIN = 384

min（MEMORY\_OVERHEAD\_FACTOR\*spark.executor.memory，MEMORY\_OVERHEAD\_MIN）

默认情况下，spark单个Executor占用的内存资源为Spark堆外内存和Heap内存，当堆外内存超出限制时会产生OOM，使Yarn直接杀死容器，这时候没有任何异常。

## GC参数调整

调整Excutor和Driver的GC性能的方式和普通JVM应用类似。

通过spark.driver.extraJavaOptions和spark.executor.extraJavaOptions，可以配置executor和driver的JVM内存。默认配置下，Spark使用的GC配置为CMS，调优时可以考虑使用性能更优的G1。

进行GC调优时可以打印出GC日志，添加以下参数：

–conf spark.driver.extraJavaOptions="-XX:+PrintGCDetails -XX:+PrintGCTimeStamps "

-conf spark.executor.extraJavaOptions="-XX:+PrintGCDetails -XX:+PrintGCTimeStamps"

添加上面的参数后，GC日志可以在JobHistory看到：



# 其他调优手段

## 分区数

Spark任务的输入在绝大多数场景下存在分区的概念（如输入是HDFS文件时，默认一个Block看做一个分区）。默认情况下一个分区的数据在一个Task线程中执行，Task线程由Executor启动占用Executor的资源。通过spark.default.parallelism来强制指定任务的分区数量，如果没有指定spark.default.parallelism，Spark会根据输入自动划分分区。

当Spark任务的输入数据总量一定时，如果分区数量过小，那么单个Executor需要处理数据量会增多，使Executor的压力过大。如果分区数量过多，Executor会频繁启停小任务影响，并且增加Driver的负载。

## spark.driver.maxResultSize

spark.driver.maxResultSize参数表示Driver能够接收的Executor任务结果集大小的上限，当Executor返回值超过Driver的内存大小，会造成Driver的OOM，导致退出进程退出。

默认时，**spark.driver.maxResultSize**大小为1GB。如果超出该大小，Driver会出现以下异常。



## 序列化接口优化

在Spark的架构中，在网络中传递的或者缓存在内存、硬盘中的对象需要进行序列化操作。通过**Java序列化（默认的序列化方式）**形成一个二进制字节数组，大大减少了数据在内存、硬盘中占用的空间，减少了网络数据传输的开销，并且可以精确的推测内存使用情况，降低GC频率。以下场景的处理的数据，均进行过序列化操作

1、分发给Executor上Task的数据分片；

2、需要缓存的RDD（前提是使用序列化方式缓存）；

3、广播变量；

4、Shuffle过程中的数据缓存；

5、使用receiver方式接收的流数据缓存；

6、算子函数中使用的外部变量；

Spark默认使用Java序列化，但是Java序列化的性能比较差，同时序列化后占用的存储空间也比较大。官方推荐，使用Kryo的序列化库代替Java序列化。

从Spark 2.0开始，简单类型、简单类型数组、字符串类型的Shuffling RDDs，默认使用Kryo序列化方式。使用Kryo序列化时能够调优的配置包括以下：

spark.kryoserializer.buffer = 64K （序列化缓存的初始大小）

spark.kryoserializer.buffer.max = 64m （序列化缓存的大小上限）

spark.kryo.unsafe = false （使用不安全的序列化工具，可以提高IO性能）

## 数据压缩

为了节省网络、内存的开销，Spark允许对RDD缓存、**广播变量、混洗数据压缩。**

**Spark默认使用**lz4压缩算法、同时支持lzf、snappy。lzf有比较高的压缩比，但是同时对CPU资源的消耗也比较高。Snappy和lz4的压缩速度由于lzf。

可以通过以下配置项，对应的变量进行数据压缩：

spark.broadcast.compress = true

spark.shuffle.compress = true

spark.shuffle.spill.compress = true

spark.rdd.compress = true

选择合适的压缩算法和序列化算法可以显著节省网络、内存开销，但是会加大CPU的负载。因此，对于IO密集型任务应当开启数据压缩，反之CPU密集任务应当关闭数据压缩。

## 网络参数调优

未完待续~~~~