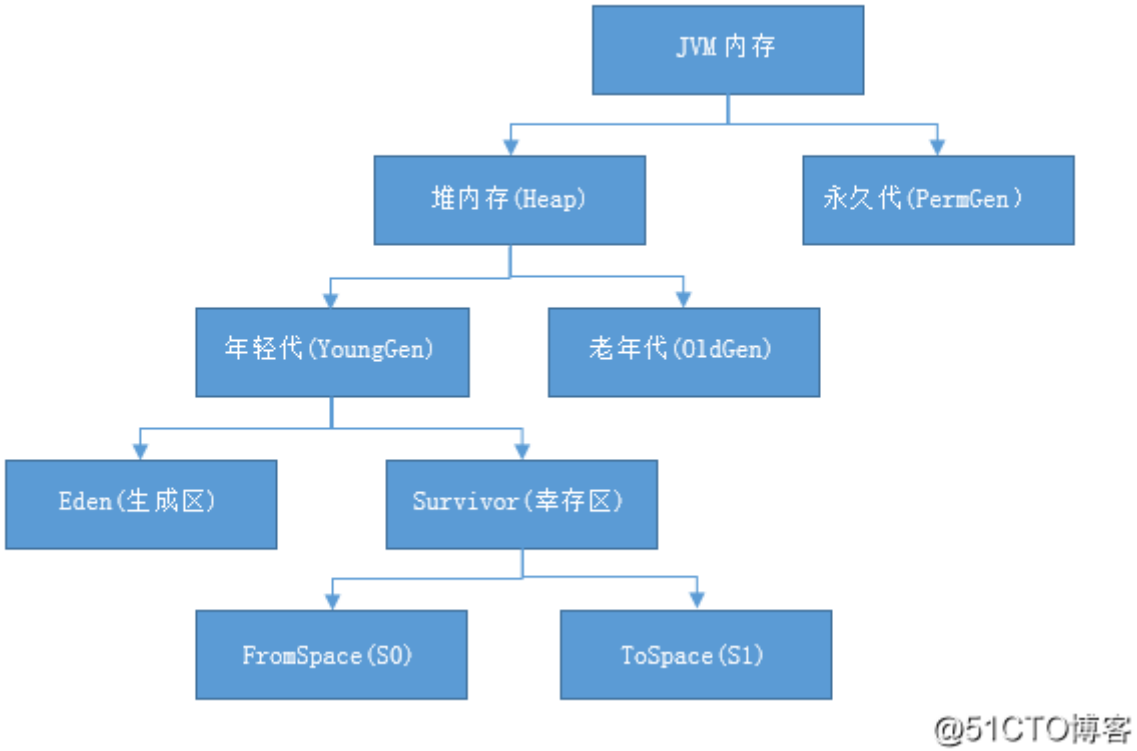
JVM堆内存(heap)详解

很好的一篇文章，转载了http://blog.51cto.com/lizhenliang/2164876?wx=

JAVA堆内存管理是影响性能主要因素之一。
堆内存溢出是JAVA项目非常常见的故障，在解决该问题之前，必须先了解下JAVA堆内存是怎么工作的。

先看下JAVA堆内存是如何划分的，如图：



1. JVM内存划分为堆内存和非堆内存，堆内存分为年轻代（Young Generation）、老年代（Old Generation），非堆内存就一个永久代（Permanent Generation）。
2. 年轻代又分为Eden和Survivor区。Survivor区由FromSpace和ToSpace组成。Eden区占大容量，Survivor两个区占小容量，默认比例是8:1:1。
3. 堆内存用途：存放的是对象，垃圾收集器就是收集这些对象，然后根据GC算法回收。
4. 非堆内存用途：永久代，也称为方法区，存储程序运行时长期存活的对象，比如类的元数据、方法、常量、属性等。

在JDK1.8版本废弃了永久代，替代的是元空间（MetaSpace），元空间与永久代上类似，都是方法区的实现，他们最大区别是：元空间并不在JVM中，而是使用本地内存。
元空间有注意有两个参数：

- MetaspaceSize： 初始化元空间大小，控制发生GC阈值
- MaxMetaspaceSize： 限制元空间大小上限，防止异常占用过多物理内存

为什么移除永久代?

移除永久代原因：为融合HotSpot JVM与JRockit VM（新JVM技术）而做出的改变，因为JRockit没有永久代。
有了元空间就不再会出现永久代OOM问题了!

分代概念

新生成的对象首先放到年轻代Eden区，当Eden空间满了，触发Minor GC，存活下来的对象移动到Survivor0区，Survivor0区满后触发执行Minor GC，Survivor0区存活对象移动到Suvivor1区，这样保证了一段时间内总有一个survivor区为空。经过多次Minor GC仍然存活的对象移动到老年代。
老年代存储长期存活的对象，占满时会触发Major GC=Full GC，GC期间会停止所有线程等待GC完成，所以对响应要求高的应用尽量减少发生Major GC，避免响应超时。
Minor GC： 清理年轻代
Major GC： 清理老年代
Full GC： 清理整个堆空间，包括年轻代和永久代
所有GC都会停止应用所有线程。

为什么分代?

将对象根据存活概率进行分类，对存活时间长的对象，放到固定区，从而减少扫描垃圾时间及GC频率。针对分类进行不同的垃圾回收算法，对算法扬长避短。

为什么survivor分为两块相等大小的幸存空间?

主要为了解决碎片化。如果内存碎片化严重，也就是两个对象占用不连续的内存，已有的连续内存不够新对象存放，就会触发GC。

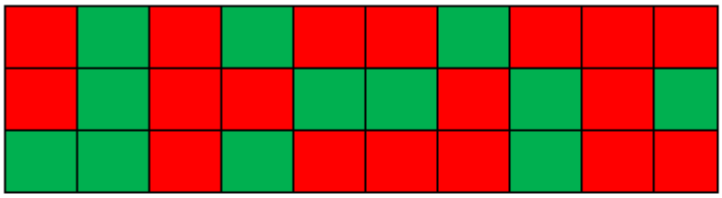
JVM堆内存常用参数

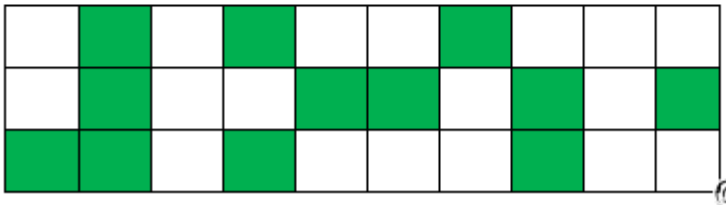
参数	描述
-Xms	堆内存初始大小，单位m、g
-Xmx（MaxHeapSize）	堆内存最大允许大小，一般不要大于物理内存的80%
-XX:PermSize	非堆内存初始大小，一般应用设置初始化200m，最大1024m就够了
-XX:MaxPermSize	非堆内存最大允许大小
-XX:NewSize（-Xns）	年轻代内存初始大小
-XX:MaxNewSize（-Xmn）	年轻代内存最大允许大小，也可以缩写
-XX:SurvivorRatio=8	年轻代中Eden区与Survivor区的容量比例值，默认为8，即8:1
-Xss	堆栈内存大小

垃圾回收算法（GC，Garbage Collection）

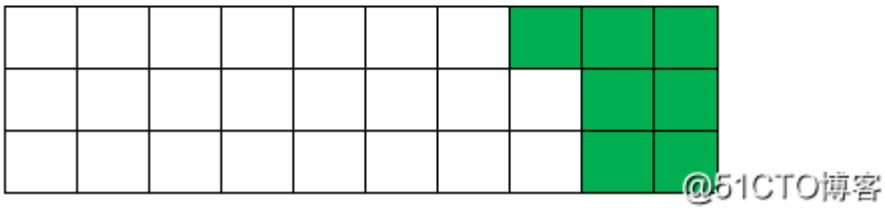
红色是标记的非活动对象，绿色是活动对象。

- **标记-清除（Mark-Sweep）**
  GC分为两个阶段，标记和清除。首先标记所有可回收的对象，在标记完成后统一回收所有被标记的对象。同时会产生不连续的内存碎片。碎片过多会导致以后程序运行时需要分配较大对象时，无法找到足够的连续内存，而不得已再次触发GC。

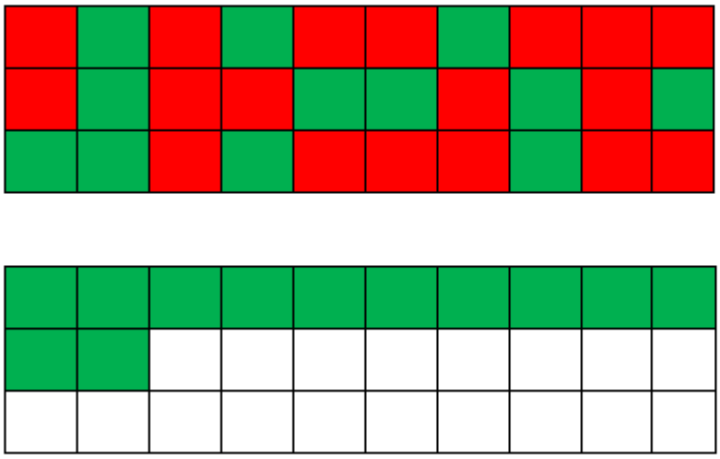




- **复制（Copy）**
  将内存按容量划分为两块，每次只使用其中一块。当这一块内存用完了，就将存活的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。这样使得每次都是对半个内存区回收，也不用考虑内存碎片问题，简单高效。缺点需要两倍的内存空间。



- **标记-整理（Mark-Compact）**
  也分为两个阶段，首先标记可回收的对象，再将存活的对象都向一端移动，然后清理掉边界以外的内存。此方法避免标记-清除算法的碎片问题，同时也避免了复制算法的空间问题。
  一般年轻代中执行GC后，会有少量的对象存活，就会选用复制算法，只要付出少量的存活对象复制成本就可以完成收集。而老年代中因为对象存活率高，没有额外过多内存空间分配，就需要使用标记-清理或者标记-整理算法来进行回收。



垃圾收集器

- **串行收集器（Serial）**
  比较老的收集器，单线程。收集时，必须暂停应用的工作线程，直到收集结束。
- **并行收集器（Parallel）**
  多条垃圾收集线程并行工作，在多核CPU下效率更高，应用线程仍然处于等待状态。
- **CMS收集器（Concurrent Mark Sweep）**
  CMS收集器是缩短暂停应用时间为目标而设计的，是基于标记-清除算法实现，整个过程分为4个步骤，包括：
  - 初始标记（Initial Mark）
  - 并发标记（Concurrent Mark）
  - 重新标记（Remark）
  - 并发清除（Concurrent Sweep）

其中，初始标记、重新标记这两个步骤仍然需要暂停应用线程。初始标记只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段是标记可回收对象，而重新标记阶段则是为了修正并发标记期间因用户程序继续运作导致标记产生变动的那一部分对象的标记记录，这个阶段暂停时间比初始标记阶段稍长一点，但远比并发标记时间段。
由于整个过程中消耗最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，CMS收集器内存回收与用户一起并发执行的，大大减少了暂停时间。

- **G1收集器（Garbage First）**
  G1收集器将堆内存划分多个大小相等的独立区域（Region），并且能预测暂停时间，能预测原因它能避免对整个堆进行全区收集。G1跟踪各个Region里的垃圾堆积价值大小（所获得空间大小以及回收所需时间），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region，从而保证了再有限时间内获得更高的收集效率。
  G1收集器工作工程分为4个步骤，包括：
  - 初始标记（Initial Mark）
  - 并发标记（Concurrent Mark）
  - 最终标记（Final Mark）
  - 筛选回收（Live Data Counting and Evacuation）

初始标记与CMS一样，标记一下GC Roots能直接关联到的对象。并发标记从GC Root开始标记存活对象，这个阶段耗时比较长，但也可以与应用线程并发执行。而最终标记也是为了修正在并发标记期间因用户程序继续运作而导致标记产生变化的那一部分标记记录。最后在筛选回收阶段对各个Region回收价值和成本进行排序，根据用户所期望的GC暂停时间来执行回收。

垃圾收集器参数

参数	描述
-XX:+UseSerialGC	串行收集器
-XX:+UseParallelGC	并行收集器
-XX:+UseParallelGCThreads=8	并行收集器线程数，同时有多少个线程进行垃圾回收，一般与CPU数量相等
-XX:+UseParallelOldGC	指定老年代为并行收集
-XX:+UseConcMarkSweepGC	CMS收集器（并发收集器）
-XX:+UseCMSCompactAtFullCollection	开启内存空间压缩和整理，防止过多内存碎片
-XX:CMSFullGCsBeforeCompaction=0	表示多少次Full GC后开始压缩和整理，0表示每次Full GC后立即执行压缩和整理
-XX:CMSInitiatingOccupancyFraction=80%	表示老年代内存空间使用80%时开始执行CMS收集，防止过多的Full GC
-XX:+UseG1GC	G1收集器
-XX:MaxTenuringThreshold=0	在年轻代经过几次GC后还存活，就进入老年代，0表示直接进入老年代

为什么会堆内存溢出?

在年轻代中经过GC后还存活的对象会被复制到老年代中。当老年代空间不足时，JVM会对老年代进行完全的垃圾回收（Full GC）。如果GC后，还是无法存放从Survivor区复制过来的对象，就会出现OOM（Out of Memory）。

OOM（Out of Memory）异常常见有以下几个原因：
1）老年代内存不足：java.lang.OutOfMemoryError:Javaheapspace
2）永久代内存不足：java.lang.OutOfMemoryError:PermGenspace
3）代码bug，占用内存无法及时回收。
OOM在这几个内存区都有可能出现，实际遇到OOM时，能根据异常信息定位到哪个区的内存溢出。
可以通过添加个参数-XX:+HeapDumpOnOutMemoryError，让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后分析。

熟悉了JAVA内存管理机制及配置参数，下面是对JAVA应用启动选项调优配置：

```
1. JAVA_OPTS="-server -Xms512m -Xmx2g -XX:+UseG1GC -XX:SurvivorRatio=6 -
   XX:MaxGCPauseMillis=400 -XX:G1ReservePercent=15 -XX:ParallelGCThreads=4
   -XX:
```

```
2. ConcGCThreads=1 -XX:InitiatingHeapOccupancyPercent=40 -
   XX:+PrintGCDetails  -XX:+PrintGCTimeStamps -Xloggc:../logs/gc.log"
```

- 设置堆内存最小和最大值，最大值参考历史利用率设置
- 设置GC垃圾收集器为G1
- 启用GC日志，方便后期分析

小结

- 选择高效的GC算法，可有效减少停止应用线程时间。
- 频繁Full GC会增加暂停时间和CPU使用率，可以加大老年代空间大小降低Full GC，但会增加回收时间，根据业务适当取舍。