目 录

[目 录 I](#_Toc460340733)

[诠释JVM调优 1](#_Toc460340734)

[第1章 JVM内存模型及垃圾收集算法 1](#_Toc460340735)

[1.1 根据Java虚拟机规范，JVM将内存划分为 1](#_Toc460340736)

[1.2 垃圾回收算法 1](#_Toc460340737)

[第2章 内存泄漏及解决方法 2](#_Toc460340738)

[2.1 系统崩溃前的一些现象： 2](#_Toc460340739)

[2.2 生成堆的dump（倾泻）文件 2](#_Toc460340740)

[2.3 分析dump文件 2](#_Toc460340741)

[2.4 分析内存泄漏 3](#_Toc460340742)

[2.5 回归问题 3](#_Toc460340743)

[第3章 性能调优 4](#_Toc460340744)

[3.1 Java线程池（java.util.concurrent.ThreadPoolExecutor） 4](#_Toc460340745)

[3.2 连接池（org.apache.commons.dbcp.BasicDataSource） 5](#_Toc460340746)

[3.3 JVM参数 5](#_Toc460340747)

[3.4 参数说明 7](#_Toc460340748)

[3.5 常见配置汇总 8](#_Toc460340749)

[3.5.1 堆设置 8](#_Toc460340750)

[3.5.2 收集器设置 8](#_Toc460340751)

[3.5.3 垃圾回收统计信息 8](#_Toc460340752)

[3.5.4 并行收集器设置 8](#_Toc460340753)

[3.5.5 并发收集器设置 8](#_Toc460340754)

[3.6 调优方法 9](#_Toc460340755)

[3.7 调优实例 10](#_Toc460340756)

[3.7.1 实例1 10](#_Toc460340757)

[3.7.2 实例2 11](#_Toc460340758)

[3.7.3 实例3 11](#_Toc460340759)

[3.8 调优总结 11](#_Toc460340760)

[3.8.1 年轻代大小选择 11](#_Toc460340761)

[3.8.2 年老代大小选择 12](#_Toc460340762)

[3.8.3 较小堆引起的碎片问题 12](#_Toc460340763)

诠释JVM调优

# JVM内存模型及垃圾收集算法

## 根据Java虚拟机规范，JVM将内存划分为

* New（年轻代）
* Tenured（年老代）
* 永久代（Perm）

其中New和Tenured属于堆内存，堆内存会从JVM启动参数（-Xmx:3G）指定的内存中分配，Perm不属于堆内存，有虚拟机直接分配，但可以通过-XX:PermSize -XX:MaxPermSize 等参数调整其大小。

* 年轻代（New）：年轻代用来存放JVM刚分配的Java对象
* 年老代（Tenured)：年轻代中经过垃圾回收没有回收掉的对象将被Copy到年老代
* 永久代（Perm）：永久代存放Class、Method元信息，其大小跟项目的规模、类、方法的量有关，一般设置为128M就足够，设置原则是预留30%的空间。

**New又分为几个部分**

* Eden：Eden用来存放JVM刚分配的对象
* Survivor1
* Survivro2：两个Survivor空间一样大，当Eden中的对象经过垃圾回收没有被回收掉时，会在两个Survivor之间来回Copy，当满足某个条件，比如Copy次数，就会被Copy到Tenured。显然，Survivor只是增加了对象在年轻代中的逗留时间，增加了被垃圾回收的可能性。

## 垃圾回收算法

垃圾回收算法可以分为三类，都基于标记-清除（复制）算法：

1. Serial算法（单线程）
2. 并行算法
3. 并发算法

JVM会根据机器的硬件配置对每个内存代选择适合的回收算法，比如，如果机器多于1个核，会对年轻代选择并行算法，关于选择细节请参考JVM调优文档。

稍微解释下的是，并行算法是用多线程进行垃圾回收，回收期间会暂停程序的执行，而并发算法，也是多线程回收，但期间不停止应用执行。所以，并发算法适用于交互性高的一些程序。经过观察，并发算法会减少年轻代的大小，其实就是使用了一个大的年老代，这反过来跟并行算法相比吞吐量相对较低。

**还有一个问题是，垃圾回收动作何时执行？**

* 当年轻代内存满时，会引发一次普通GC，该GC仅回收年轻代。需要强调的时，年轻代满是指Eden代满，Survivor满不会引发GC
* 当年老代满时会引发Full GC，Full GC将会同时回收年轻代、年老代
* 当永久代满时也会引发Full GC，会导致Class、Method元信息的卸载

**另一个问题是，何时会抛出OutOfMemoryException（内存空间不足/内存泄露），并不是内存被耗空的时候才抛出**

* JVM98%的时间都花费在内存回收
* 每次回收的内存小于2%

满足这两个条件将触发OutOfMemoryException，这将会留给系统一个微小的间隙以做一些Down之前的操作，比如手动打印Heap Dump（堆栈）。

# 内存泄漏及解决方法

## 系统崩溃前的一些现象：

* 每次垃圾回收的时间越来越长，由之前的10ms延长到50ms左右，FullGC的时间也有之前的0.5s延长到4、5s
* FullGC的次数越来越多，最频繁时隔不到1分钟就进行一次FullGC
* 年老代的内存越来越大并且每次FullGC后年老代没有内存被释放

之后系统会无法响应新的请求，逐渐到达OutOfMemoryError的临界值。

## 生成堆的dump（倾泻）文件

通过JMX的MBean生成当前的Heap(堆栈）信息，大小为一个3G（整个堆的大小）的hprof文件，如果没有启动JMX可以通过Java的jmap命令来生成该文件。

## 分析dump文件

下面要考虑的是如何打开这个3G的堆信息文件，显然一般的Window系统没有这么大的内存，必须借助高配置的Linux。当然我们可以借助X-Window把Linux上的图形导入到Window。我们考虑用下面几种工具打开该文件：

1. 1. Visual VM
2. 2. IBM HeapAnalyzer
3. 3. JDK 自带的Hprof工具

使用这些工具时为了确保加载速度，建议设置最大内存为6G。使用后发现，这些工具都无法直观地观察到内存泄漏，Visual VM虽能观察到对象大小，但看不到调用堆栈；HeapAnalyzer虽然能看到调用堆栈，却无法正确打开一个3G的文件。因此，我们又选用了Eclipse专门的静态内存分析工具：Mat。

## 分析内存泄漏

通过Mat我们能清楚地看到，哪些对象被怀疑为内存泄漏，哪些对象占的空间最大及对象的调用关系。针对本案，在ThreadLocal中有很多的JbpmContext实例，经过调查是JBPM的Context没有关闭所致。

另外，通过Mat或JMX我们还可以分析线程状态，可以观察到线程被阻塞在哪个对象上，从而判断系统的瓶颈。

## 回归问题

**Q1：为什么崩溃前垃圾回收的时间越来越长？**

A：根据内存模型和垃圾回收算法，垃圾回收分两部分：内存标记、清除（复制），标记部分只要内存大小固定时间是不变的，变的是复制部分，因为每次垃圾回收都有一些回收不掉的内存，所以增加了复制量，导致时间延长。所以，垃圾回收的时间也可以作为判断内存泄漏的依据

**Q2：为什么Full GC的次数越来越多？**

A：因此内存的积累，逐渐耗尽了年老代的内存，导致新对象分配没有更多的空间，从而导致频繁的垃圾回收

**Q3:为什么年老代占用的内存越来越大？**

A：因为年轻代的内存无法被回收，越来越多地被Copy到年老代

# 性能调优

除了上述内存泄漏外，我们还发现CPU长期不足3%，系统吞吐量不够，针对8core×16G、64bit的Linux服务器来说，是严重的资源浪费。

在CPU负载不足的同时，偶尔会有用户反映请求的时间过长，我们意识到必须对程序及JVM进行调优。从以下几个方面进行：

1. **线程池**：解决用户响应时间长的问题
2. **连接池**
3. **JVM启动参数**：调整各代的内存比例和垃圾回收算法，提高吞吐量
4. **程序算法**：改进程序逻辑算法提高性能

## Java线程池（java.util.concurrent.ThreadPoolExecutor）

大多数JVM6上的应用采用的线程池都是JDK自带的线程池，之所以把成熟的Java线程池进行罗嗦说明，是因为该线程池的行为与我们想象的有点出入。Java线程池有几个重要的配置参数：

* corePoolSize：核心线程数（最新线程数）
* maximumPoolSize：最大线程数，超过这个数量的任务会被拒绝，用户可以通过RejectedExecutionHandler接口自定义处理方式
* keepAliveTime：线程保持活动的时间
* workQueue：工作队列，存放执行的任务

Java线程池需要传入一个Queue参数（workQueue）用来存放执行的任务，而对Queue的不同选择，线程池有完全不同的行为：

* SynchronousQueue：一个无容量的等待队列，一个线程的insert操作必须等待另一线程的remove操作，采用这个Queue线程池将会为每个任务分配一个新线程
* LinkedBlockingQueue ：无界队列，采用该Queue，线程池将忽略 maximumPoolSize参数，仅用corePoolSize的线程处理所有的任务，未处理的任务便在LinkedBlockingQueue中排队
* ArrayBlockingQueue：有界队列，在有界队列和 maximumPoolSize的作用下，程序将很难被调优：更大的Queue和小的maximumPoolSize将导致CPU的低负载；小的Queue和大的池，Queue就没起动应有的作用。

其实我们的要求很简单，希望线程池能跟连接池一样，能设置最小线程数、最大线程数，当最小数<任务<最大数时，应该分配新的线程处理；当任务>最大数时，应该等待有空闲线程再处理该任务。

但线程池的设计思路是，任务应该放到Queue中，当Queue放不下时再考虑用新线程处理，如果Queue满且无法派生新线程，就拒绝该任务。设计导致“先放等执行”、“放不下再执行”、“拒绝不等待”。所以，根据不同的Queue参数，要提高吞吐量不能一味地增大maximumPoolSize。

当然，要达到我们的目标，必须对线程池进行一定的封装，幸运的是ThreadPoolExecutor中留了足够的自定义接口以帮助我们达到目标。我们封装的方式是：

* 以SynchronousQueue作为参数，使maximumPoolSize发挥作用，以防止线程被无限制的分配，同时可以通过提高maximumPoolSize来提高系统吞吐量
* 自定义一个RejectedExecutionHandler，当线程数超过maximumPoolSize时进行处理，处理方式为隔一段时间检查线程池是否可以执行新Task，如果可以把拒绝的Task重新放入到线程池，检查的时间依赖keepAliveTime的大小。

## 连接池（org.apache.commons.dbcp.BasicDataSource）

在使用org.apache.commons.dbcp.BasicDataSource的时候，因为之前采用了默认配置，所以当访问量大时，通过JMX观察到很多Tomcat线程都阻塞在BasicDataSource使用的Apache ObjectPool的锁上，直接原因当时是因为BasicDataSource连接池的最大连接数设置的太小，默认的BasicDataSource配置，仅使用8个最大连接。

我还观察到一个问题，当较长的时间不访问系统，比如2天，DB上的Mysql会断掉所以的连接，导致连接池中缓存的连接不能用。为了解决这些问题，我们充分研究了BasicDataSource，发现了一些优化的点：

* Mysql默认支持100个链接，所以每个连接池的配置要根据集群中的机器数进行，如有2台服务器，可每个设置为60
* initialSize：参数是一直打开的连接数
* minEvictableIdleTimeMillis：该参数设置每个连接的空闲时间，超过这个时间连接将被关闭
* timeBetweenEvictionRunsMillis：后台线程的运行周期，用来检测过期连接
* maxActive：最大能分配的连接数
* maxIdle：最大空闲数，当连接使用完毕后发现连接数大于maxIdle，连接将被直接关闭。只有initialSize < x < maxIdle的连接将被定期检测是否超期。这个参数主要用来在峰值访问时提高吞吐量。
* initialSize是如何保持的？经过研究代码发现，BasicDataSource会关闭所有超期的连接，然后再打开initialSize数量的连接，这个特性与minEvictableIdleTimeMillis、timeBetweenEvictionRunsMillis一起保证了所有超期的initialSize连接都会被重新连接，从而避免了Mysql长时间无动作会断掉连接的问题。

## JVM参数

在JVM启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置JVM会工作的很好，但对一些配置很好的Server和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标：

* GC的时间足够的小
* GC的次数足够的少
* 发生Full GC的周期足够的长

前两个目前是相悖的，要想GC时间小必须要一个更小的堆，要保证GC次数足够少，必须保证一个更大的堆，我们只能取其平衡。

（1）针对JVM堆的设置一般，可以通过-Xms -Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值

（2）年轻代和年老代将根据默认的比例（1：4）分配堆内存，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小

（3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

* + 更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC
  + 更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少Full GC的频率
  + 如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 （B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：4。但应该给年老代至少预留1/3的增长空间

（4）在配置较好的机器上（比如多核、大内存），可以为年老代选择并行收集算法： -XX:+UseParallelOldGC ，默认为Serial收集

（5）线程堆栈的设置：每个线程默认会开启1M的堆栈，用于存放栈帧、调用参数、局部变量等，对大多数应用而言这个默认值太了，一般256K就足用。理论上，在内存不变的情况下，减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

（6）可以通过下面的参数打Heap Dump信息

* -XX:HeapDumpPath
* -XX:+PrintGCDetails
* -XX:+PrintGCTimeStamps
* -Xloggc:/usr/aaa/dump/heap\_trace.txt

通过下面参数可以控制OutOfMemoryError时打印堆的信息

* -XX:+HeapDumpOnOutOfMemoryError

请看一下一个时间的Java参数配置：（服务器：Linux 64Bit，8Core×16G）

JAVA\_OPTS="$JAVA\_OPTS -server -Xms3G -Xmx3G -Xss256k -XX:PermSize=128m -XX:MaxPermSize=128m -XX:+UseParallelOldGC -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/aaa/dump -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:/usr/aaa/dump/heap\_trace.txt -XX:NewSize=1G -XX:MaxNewSize=1G"

经过观察该配置非常稳定，每次普通GC的时间在10ms左右，Full GC基本不发生，或隔很长很长的时间才发生一次

通过分析dump文件可以发现，每个1小时都会发生一次Full GC，经过多方求证，只要在JVM中开启了JMX服务，JMX将会1小时执行一次Full GC以清除引用，关于这点请参考附件文档。

## 参数说明

-server -Xmx3g -Xms3g -XX:MaxPermSize=128m

-XX:NewRatio=2 # eden/old 的比例

-XX:SurvivorRatio=8 # s/e的比例

-XX:+UseParallelGC

-XX:ParallelGCThreads=8

-XX:+UseParallelOldGC # 这个是JAVA 6出现的参数选项

-XX:LargePageSizeInBytes=128m # 内存页的大小， 不可设置过大，会影响Perm的大小

-XX:+UseFastAccessorMethods # 原始类型的快速优化

-XX:+DisableExplicitGC # 关闭System.gc()

-Xss # 是线程栈的大小

另外 -Xss 是线程栈的大小， 这个参数需要严格的测试，一般小的应用，如果栈不是很深， 应该是128k够用的，不过，我们的应用调用深度比较大，还需要做详细的测试。这个选项对性能的影响比较大。**建议使用256K的大小**。

## 常见配置汇总

### 堆设置

-Xms：初始堆大小

-Xmx：最大堆大小

-XX：NewSize=n：设置年轻代大小

-XX：NewRatio=n：设置年轻代和年老代的比值。如：为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX：SurvivorRatio=n：年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

-XX：MaxPermSize=n：设置持久代大小

### 收集器设置

-XX：+UseSerialGC：设置串行收集器

-XX：+UseParallelGC：设置并行收集器

-XX：+UseParalledlOldGC：设置并行年老代收集器

-XX：+UseConcMarkSweepGC：设置并发收集器

### 垃圾回收统计信息

-XX：+PrintGC

-XX：+PrintGCDetails

-XX：+PrintGCTimeStamps

-Xloggc：filename

### 并行收集器设置

-XX：ParallelGCThreads=n：设置并行收集器收集时使用的CPU数。并行收集线程数。

-XX：MaxGCPauseMillis=n：设置并行收集最大暂停时间

-XX：GCTimeRatio=n：设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

### 并发收集器设置

-XX：+CMSIncrementalMode：设置为增量模式。适用于单CPU情况。

-XX：ParallelGCThreads=n：设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

## 调优方法

一切都是为了这一步，**调优**，在调优之前，我们需要记住下面的**原则**：

1、多数的Java应用不需要在服务器上进行GC优化；

2、多数导致GC问题的Java应用，都不是因为我们参数设置错误，而是代码问题；

3、在应用上线之前，先考虑将机器的JVM参数设置到最优（最适合）；

4、减少创建对象的数量；

5、减少使用全局变量和大对象；

6、GC优化是到最后不得已才采用的手段；

7、在实际使用中，分析GC情况优化代码比优化GC参数要多得多；

GC优化的目的有两个（<http://www.360doc.com/content/13/0305/10/15643_269388816.shtml> ）：

**1、将转移到老年代的对象数量降低到最小；**

**2、减少full GC的执行时间；**

为了达到上面的目的，一般地，你需要做的事情有：

1、减少使用全局变量和大对象；

2、调整新生代的大小到最合适；

3、设置老年代的大小为最合适；

4、选择合适的GC收集器；

在上面的4条方法中，用了几个“合适”，那究竟什么才算合适，一般的，请参考上面“收集器搭配”和“启动内存分配”两节中的建议。但这些建议不是万能的，需要根据您的机器和应用情况进行发展和变化，实际操作中，可以将两台机器分别设置成不同的GC参数，并且进行对比，选用那些确实提高了性能或减少了GC时间的参数。

真正熟练的使用GC调优，是建立在多次进行GC监控和调优的实战经验上的，进行监控和调优的一般步骤为：

1. **监控GC的状态**

使用各种JVM工具，查看当前日志，分析当前JVM参数设置，并且分析当前堆内存快照和gc日志，根据实际的各区域内存划分和GC执行时间，觉得是否进行优化；

1. **分析结果，判断是否需要优化**

如果各项参数设置合理，系统没有超时日志出现，GC频率不高，GC耗时不高，那么没有必要进行GC优化；如果GC时间超过1-3秒，或者频繁GC，则必须优化；

注：**如果满足下面的指标，则一般不需要进行GC**：

Minor GC执行时间不到50ms；

Minor GC执行不频繁，约10秒一次；

Full GC执行时间不到1s；

Full GC执行频率不算频繁，不低于10分钟1次；

1. **调整GC类型和内存分配**

如果内存分配过大或过小，或者采用的GC收集器比较慢，则应该优先调整这些参数，并且先找1台或几台机器进行beta，然后比较优化过的机器和没有优化的机器的性能对比，并有针对性的做出最后选择；

1. **不断的分析和调整**

通过不断的试验和试错，分析并找到最合适的参数

1. **全面应用参数**

如果找到了最合适的参数，则将这些参数应用到所有服务器，并进行后续跟踪。

## 调优实例

上面的内容都是纸上谈兵，下面我们以一些真实例子来进行说明。

### 实例1

笔者昨日发现部分开发测试机器出现异常：

java.lang.OutOfMemoryError: GC overhead limit exceeded

这个异常代表：GC为了释放很小的空间却耗费了太多的时间，其原因一般有两个

1. 堆太小
2. 有死循环或大对象

笔者首先排除了第2个原因，因为这个应用同时是在线上运行的，如果有问题，早就挂了。所以怀疑是这台机器中堆设置太小；

使用ps -ef |grep "java" 查看，发现：

E:\youdaodir\kuangzhanmoyu1@163.com\a86b50e234794fca8c03a9006d3ee18e\e78aced45e01.png

该应用的堆区设置只有768m，而机器内存有2g，机器上只跑这一个java应用，没有其他需要占用内存的地方。另外，这个应用比较大，需要占用的内存也比较多；

笔者通过上面的情况判断，只需要改变堆中各区域的大小设置即可，于是改成下面的情况：

E:\youdaodir\kuangzhanmoyu1@163.com\1d5eaf9ce54b43d7b8c70b64d2387155\08300587d5e4.png

跟踪运行情况发现，相关异常没有再出现；

### 实例2

<http://www.360doc.com/content/13/0305/10/15643_269388816.shtml>

一个服务系统，**经常出现卡顿，分析原因，发现Full GC时间太长**：

jstat -gcutil:

S0 S1 E O P YGC YGCT FGC FGCT GCT

12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993

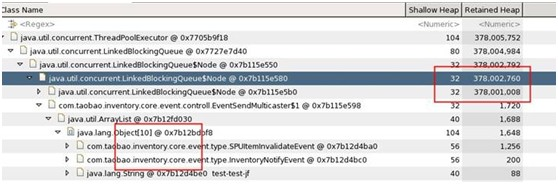
分析上面的数据，发现Young GC执行了54次，耗时2.047秒，每次Young GC耗时37ms，在正常范围，**而Full GC执行了5次，耗时6.946秒，每次平均1.389s，数据显示出来的问题是：Full GC耗时较长**，分析该系统的是指发现，NewRatio=9，也就是说，新生代和老生代大小之比为1:9，这就是问题的原因：

1. 新生代太小，导致对象提前进入老年代，触发老年代发生Full GC；
2. 老年代较大，进行Full GC时耗时较大；

优化的方法是调整NewRatio的值，调整到4，发现Full GC没有再发生，只有Young GC在执行。这就是把对象控制在新生代就清理掉，没有进入老年代（这种做法对一些应用是很有用的，但并不是对所有应用都要这么做）

### 实例3

一应用在性能测试过程中，发现内存占用率很高，Full GC频繁，使用sudo -u admin -H jmap -dump:format=b,file=文件名.hprof pid 来dump内存，生成dump文件，并使用Eclipse下的mat差距进行分析，发现：



从图中可以看出，这个线程存在问题，队列LinkedBlockingQueue所引用的大量对象并未释放，导致整个线程占用内存高达378m，此时通知开发人员进行代码优化，将相关对象释放掉即可。

## 调优总结

### 年轻代大小选择

* **响应时间优先的应用**：**尽可能设大，直到接近系统的最低响应时间限制**（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。
* **吞吐量优先的应用**：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

### 年老代大小选择

* **响应时间优先的应用**：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑**并发会话率**和**会话持续时间**等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：
* 并发垃圾收集信息
* 持久代并发收集次数
* 传统GC信息
* 花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

* **吞吐量优先的应用**：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

### 较小堆引起的碎片问题

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

* -XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。
* -XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩。

**参考资料：**

<http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>