JVM GC机制与测试

Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译的程序之需要生成在JVM上运行的目标代码（bytecode），就可以在多种平台上不加修改的运行。在JDK中，编译器将Java文件编译成class文件，然后将class文件输入到jvm中，JVM会加载并执行类文件，JVM架构图如下所示：



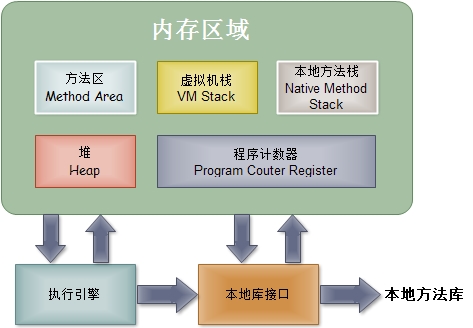
图1. JVM 架构图

如上图所示，JVM分为三个主要子系统：

* 类加载器子系统，将class文件加载到JVM虚拟机中，并对文件数据进行校验、解析和初始化，最终形成JVM可以直接使用的Java对象组件。
* 运行时数据区，JVM将内存分成多个区域：方法区，存储类级别的数据，包括静态变量；堆栈区，线程的单独运行时栈；PC寄存器，每个线程都有单独的PC寄存器，用于保存当前执行的指令地址；本地方法栈，保存本地方法信息；堆区域，所有对象的实例和数组都存储在这个区域。
* 执行引擎，从运行时数据区读取字节码，转换成机器指令，然后通过本地库接口调用物理机器。

# JVM内存分配

内存管理和垃圾回收是JVM中非常关键的点。在程序运行过程当中，会创建大量的对象，其中大部分是短周期的对象，需要频繁的进行垃圾回收以保证无用对象尽早的被释放掉，对于长周期的对象，则不需要频繁进行扫描检测以确保回收。下图是JVM内存区域模型：



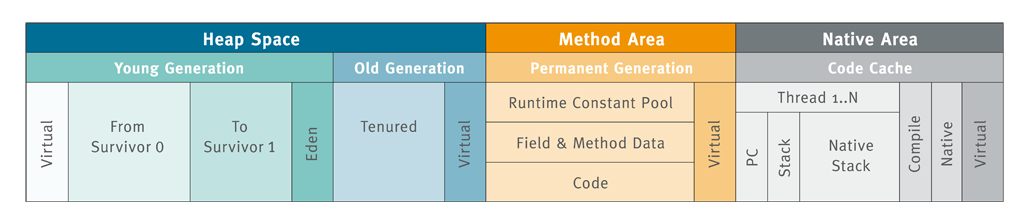
内存数据区分为5个主要区域：

1. 方法区， 所有的类级别的数据都存储在这里，包括静态变量，每个JVM只有一个方法区，并且它是共享资源。类加载器将数据加载到这个数据区中。这个区域也成为“永久代”。
2. 堆区域，所有对象及其对应的实例和数组都存储在这里，每个JVM只有一个堆区域，由于方法和堆区域共享多个线程的内存，因此所存储的数据非线程安全。这个区域也就做GC堆，是虚拟机内存中管理的最大一块内存区域。这个区域被划分为新生代和老年代：新生代存储新创建的对象和尚未进入老年代的对象；老年代存储经过多次新声带GC仍然存活的对象。
3. 堆栈区，JVM为每个线程都会创建一个单独的运行时栈，对于每个方法调用，将在堆栈存储器中产生一个条目，成为堆栈帧。所有局部变量将在堆栈内存中创建，堆栈区域是线程安全的，不是共享资源。堆栈帧分为三个子元素：

* 局部变量数组，与方法相关，涉及局部变量以及将在此存储的相应值得多少
* 操作数堆栈，如果需要执行任何中间操作，那么操作数堆栈充当工作空间来执行操作
* 帧数据，对应于方法的所有符号存储在此处，在任何异常的情况下，捕捉块信息都被保持在帧数据中。

1. PC寄存器，每个线程都有单独的PC寄存器，用于保存当前执行的指令地址，一旦指令执行，PC寄存器将更新到下一条指令
2. 本地方法堆栈，本地方法堆栈保存本地方法信息，对于每个线程，将创建一个单独的本地方法堆栈

在这个5个区域中，方法区和堆区域是JVM内存分配和垃圾回收的主要区域，其该部分内存模型抽象如下图所示：



其中Perm区域为永久代，GC不会在程序运行期间对这个区域进行清理，在JDK 1.8之后，取而代之的是元数据区Metaspace。

所有的对象在实例化后的整个运行周期内，都被存放在堆内存中，JVM中为了对短周期和长周期对象进行管理，Hotspot JVM目前采用分代的策略，将堆区域分成年轻代和老生代两个区域。如果不对堆内存进行区域划分，所有新创建对象和生命周期很长的对象放在一起，随着程序的执行，对内存需要频繁的进行垃圾收集，每次回收都需要遍历所有对象，遍历这些对象花费的时间代价是巨大的，会影响GC效率。

## 年轻代（Young Generation）

年轻代是所有新对象生成的地方，当年轻代内存被用完后，会触发垃圾回收，这个垃圾回称为为Minor GC，主要相关配置参数：

-XX:NewRatio，新生代和老生代的比例，默认为2，表示年轻代与老生代的比值为1:2

-XX:Newsize，年轻代的初始值大小

-XX:Maxnewsize，年轻代的最大值大小

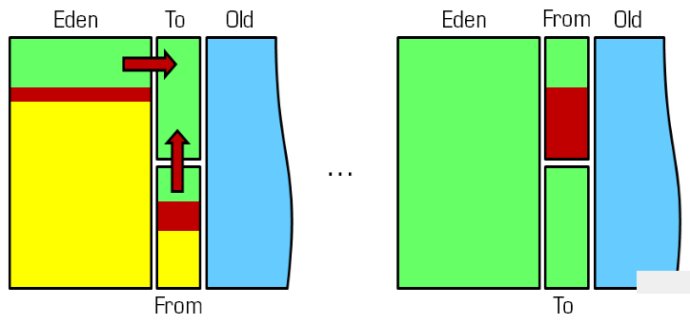
这个区域内存相对比较小，所有Minor GC会比较频繁，年轻代分为1个Eden和2个Survivor区域，默认比例为8:1，在大多数的情况下，对象在Eden区中分配，当Eden区没有区域没有足够的区间分配时，JVM触发一次Minor GC，将Eden区和其中一块Survivor 0区内存活的对象放入另一块Survivor 1区域，在Minor GC期间将超过存活阈值的对象放入老生代，对年轻代这么划分的目的有两个：

**1) 避免频繁Full GC**，如果堆区域仅分为新生代和老生代，Eden每进行一次Minor GC，那么存活的对象就会被送到老年代，老年代会很快填满，然后触发Full GC，这个过程执行时间比较长，影响程序的执行和相应的速度。因此在新生代中加入一个Survivor区域，在Minor GC过程中根据对象年龄值决定在内存中的去向，相关参数：

-XX:MaxTenuringThreshold

没有达到阈值的对象复制到Survivor区。

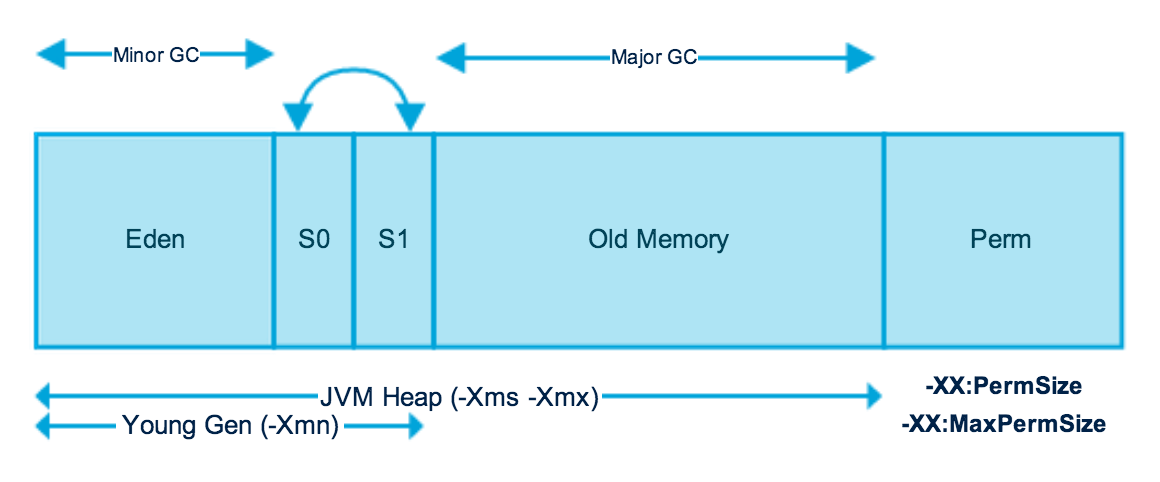
2）**解决内存碎片化问题**，如果只有一个Survivor区，一旦Eden满就触发Minor GC，存活的对象到Survivor区，多次Minor GC会在Survivor中占用多个区域，但是在Minor GC的过程中部分对象会移动到老生代中，因此每个区域中都有空闲区域，造成内存碎片。为了解决这个问题，将Survivor区分为From Survivor 0和To Survivor 1两个内存区：



在Minor GC时，Eden区和From Survivor 0未到年龄阈值的对象移动到To Survivor 1区，然后From Survivor 0和To Survivor 1交换角色。

## 年老代（Old Generation）

在年老代中，包含了长期存活的对象和经过多次Minor GC后仍然存放下来的对象。当年老代被占满时进行Full GC，在Full GC的过程中多数应用线程都停下来，然后检查所有存活的对象，这个过程消耗时间较长，造成应用反应迟钝。JVM内存的划分及对象配置，如下图所示：



JVM整个内存大小=年轻代大小+年老代大小+持久代大小。

* -Xms，初始堆内存，等价于-XX:InitialHeapSize，示例：-Xms2G
* -Xmx，最大堆内存，等价于-XX:MaxHeapSize，最好将-Xms和-Xmx设置一样大，减少申请内存的消耗
* -Xmn，设置年轻代的大小
* -XX:NewRatio，设置老生代与新生代的比例
* -XX:SurvivorRatio，Eden区与Survivor区的大小比例，默认是8
* -XX:PermSize，永久代初始大小
* -XX:MaxPermSize，永久代最大值，永久代的大小并会被包括到使用参数-Xmx设置的堆内存大小中，两者是独立的。
* -Xss，设置每个线程的堆栈大小，相同物理内存下，减少这个值能生成更多的线程。

# 垃圾回收

垃圾回收机制是Java语言的一个优势，在JVM中垃圾回收由GC(守护进程)来实现，其根据内存使用情况自动运行。由于GC需要消耗一定资源和时间，采用分代的方式进行对象收集，按照新生代、旧生代对对象进行收集，尽可能的缩短GC对应用造成的暂停。GC对垃圾的回收分为：

1. **Partitial GC，不收集整个GC堆得模式**

* Minor GC（Young GC），当Eden区没有足够的区间区分配时，JVM发起一次Minor GC，将Eden区和其中一块Survivor 0区内尚存活的对象放入另一块Survivor 1区域
* Old GC，当年老代内存被占满时进行Old GC，清理老生代，只有CMS的Concurrent Collection是这个模式。
* Mixed GC，收集整个young gen以及部分old gen GC，目前只有G1有这个模式。

1. **Full GC，对整个堆进行GC，当出现以下情况触发Full GC**

* System.gc方法调用
* 老生代空间不足
* 永生区空间不足
* CMS GC时出现promotion failed和concurrent mode failure一次。Concurrent Mode Failure时在执行CMS GC的过程中同时又对象要放入老生代，而老生代空间不足造成的。
* 堆中分配很大的对象，该对象直接放入老生代，但是老生代没有足够的连续空间放置该对象

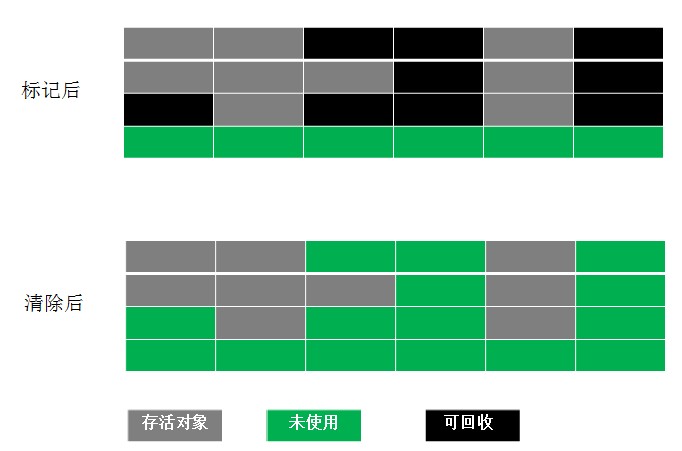
Major GC通常和Full GC是等价的，收集整个GC堆.

## 2.1垃圾回收算法

在JVM规范中并没有明确GC的运作方式，可以采用不同的方式去实现垃圾回收器，下面是几种常见的GC算法。

### 2.1.1 标记-清除算法（Mark-Sweep）

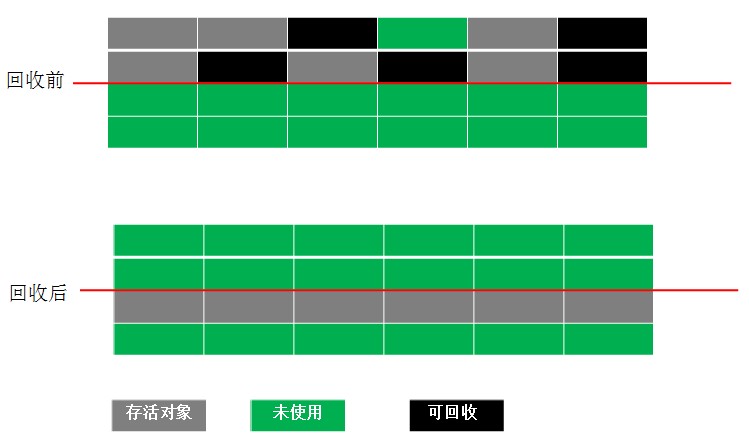
这个是最基础的垃圾回收算法，分为两个阶段：标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间，如下图所示：



从图中可以发现，该算法最大的问题是内存碎片化严重，可能会发生大对象找不到可利用空间的问题。

### 2.1.2 复制算法（Copying）

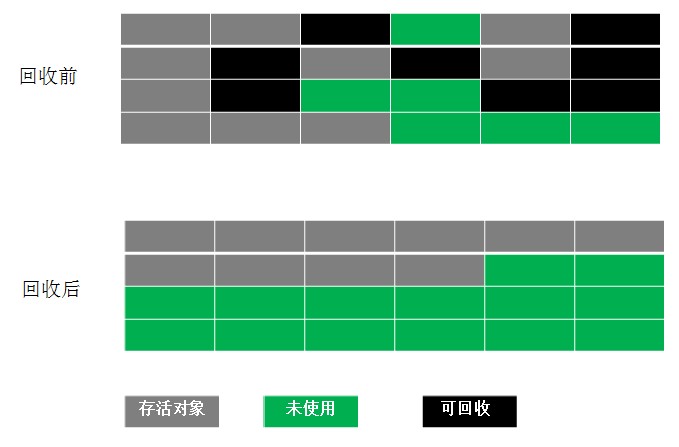
为了解决Mark-Sweep算法内存碎片化的缺陷而被提出来的算法，按内存容量将内存划分为等大小的两块，每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉，如下图：



算法实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原来的一半，且存活对象增多的话，Copying算法的效率会大大降低。

### 2.1.2 标记-整理算法

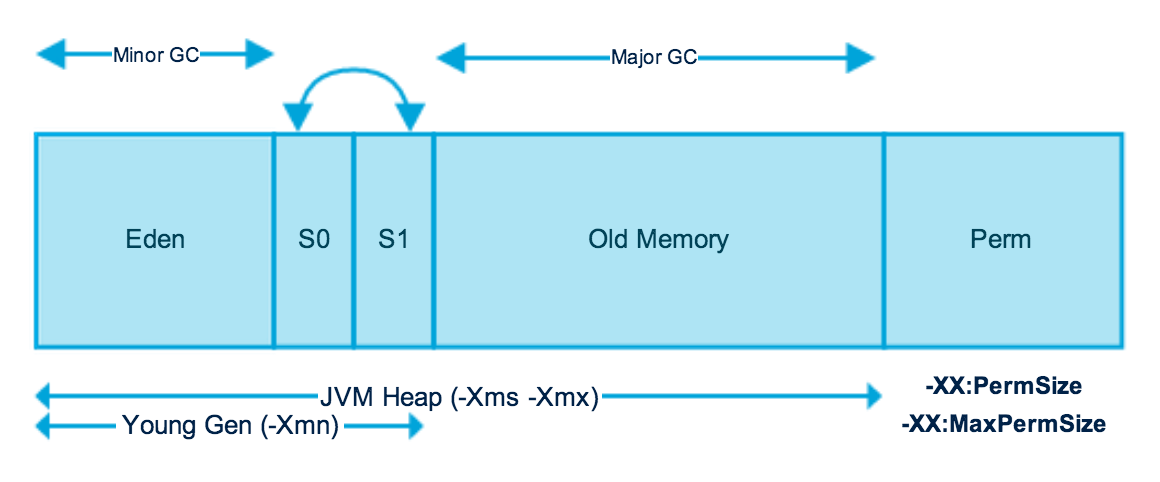
结合以上两个算法，为了避免缺陷，标记阶段和Mark-Sweep算法相同，标记后不是清理对象，而是将存活对象移向内存的一端，然后清除端边界外的内存，示意图如下所示：



### 2.1.4 分代收集算法（Generational Collection）

分代收集算法是目前大部分JVM采用的方法，核心方法是根据对象存活的不同生命周期将内存分为不同的域，一般情况下将GC堆分为老生代和新生代。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

目前大部分JVM的GC对于新生代都采取Copying算法，因为新生代每次垃圾回收都要回收大部分对象，即要复制的比较少，但通常并不是按照1:1来划分新生代，而是一块较大的Eden空间和两个较小的Survivor空间（From Space和To Space），每次使用Eden空间和其中一块Survivor空间，当回收时将该两块空间中还存活的对象复制到另一块Survivor空间中：



而老生代因为每次只回收少量对象，因此采用Mark-Compact算法。

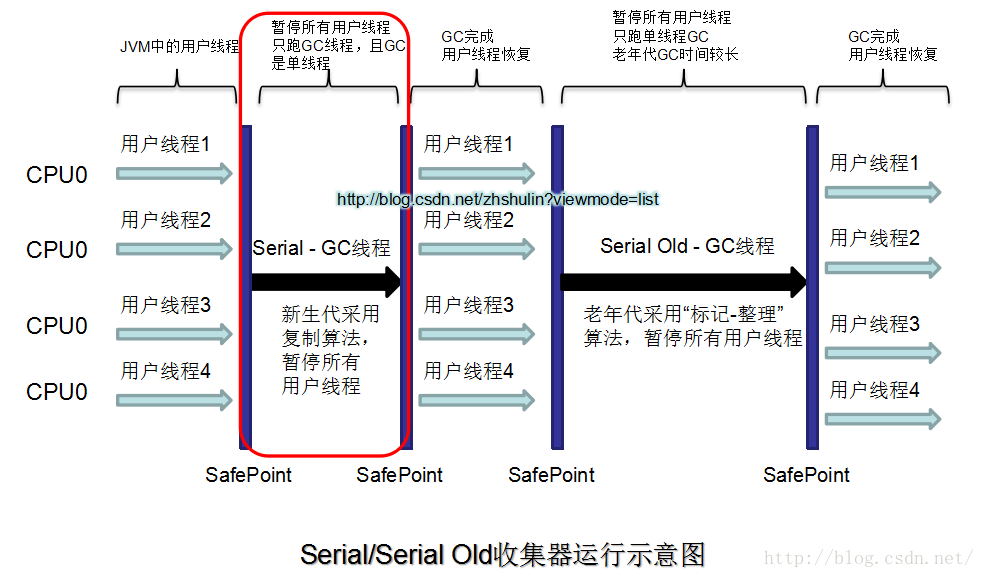
对象的内存分配主要在新生代的Eden Space和Survivor Space的From Space(Survivor目前存放对象的那一块)，少数情况会直接分配到老生代。当新生代的Eden Space和From Space空间不足时就会发生一次GC，进行GC后，Eden Space和From Space区的存活对象会被挪到To Space，然后将Eden Space和From Space进行清理。如果To Space无法足够存储某个对象，则将这个对象存储到老生代。在进行GC后，使用的便是Eden Space和To Space了，如此反复循环。当对象在Survivor区躲过一次GC后，其年龄就会+1。默认情况下年龄到达15的对象会被移到老生代中。

## 2.2 垃圾收集器

垃圾回收算法是垃圾收集器的理论基础，而垃圾收集器是其实现，下面是HotSpot虚拟机的几种垃圾收集器。

### 2.2.1 Serial/Serial Old收集器

Serial收集器是最基本、最古老的收集器，这个收集器是一个单线程收集器，用它进行收集的时候，必须暂停所有用户线程，使用标记-整理算法：



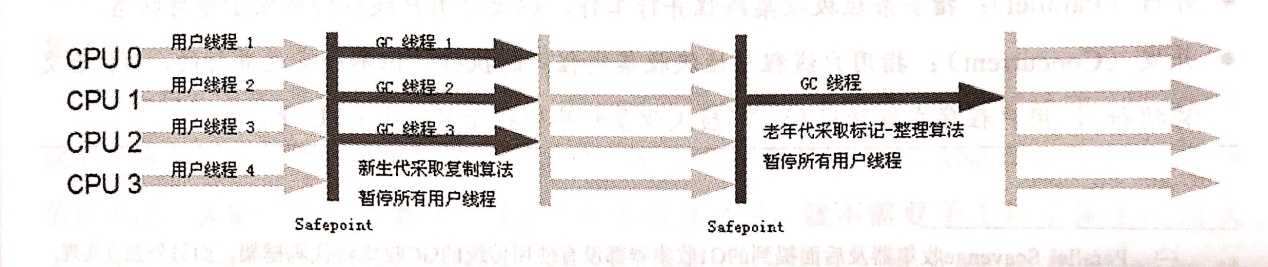
JDK 1.5之前与Parallel Scavenge收集器搭配使用。其作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时启用。

### 2.2.2 ParNew

ParNew收集器是Serial收集器的多线程版本，除了使用多线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSize

Threshold、-XX:HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器一致，在实现上，这两种收集器共用了相当多的代码。

ParNew收集器的工作过程如下图：

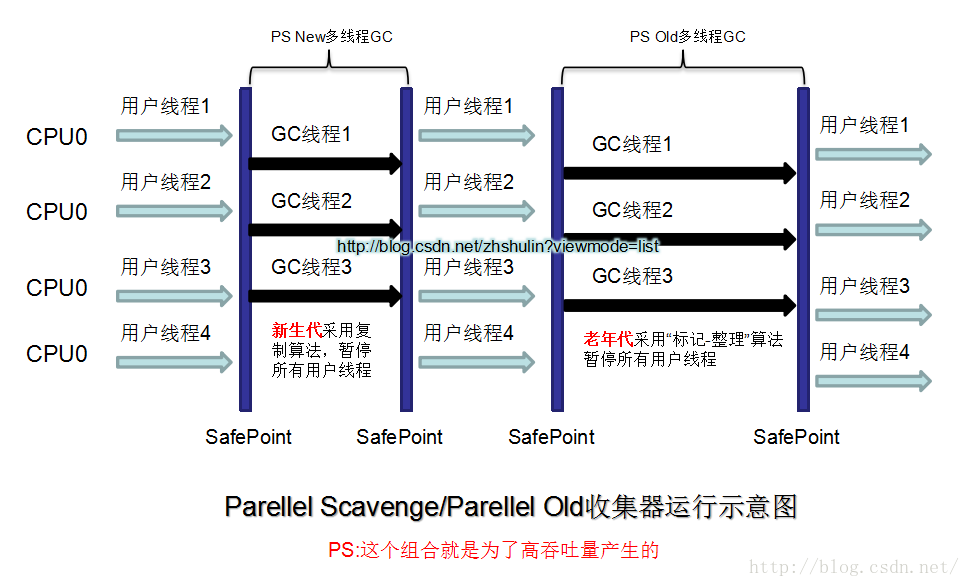


ParNew收集器除了多线程外，其他和Serial差不多，ParNew是许多运行在Server模式下的JVM中首选的垃圾收集器，一个重要的原因是除了Serial，它是唯一可以和CMS(Concurrent Mark Sweep)老年代收集器配合工作。

ParNew在单CPU环境中收集效果下不如Serial收集器，但是随着CPU增加，对于GC时系统资源的有效利用还是有好处的。它默认开启的收集线程数与CPU的数量相同，在CPU非常多的环境下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

### 2.2.3 Parallel Scavenge/Parallel Old

Parallel Scavenge收集器，简称PS收集器，和ParNew收集器一样是一个多线程的并行新生代垃圾收集器，一样采用复制算法，但是PS收集器的目标是达到一个可控制的吞吐量（CPU用于运行用户代码的时间与CPU消耗时间的比值），适合用于后台计算而不需要太多交互的任务：

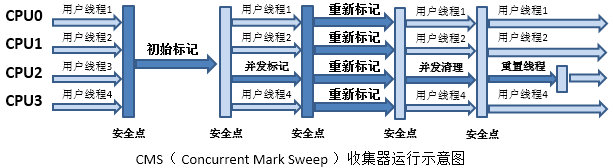


PS收集器正是基于对吞吐量的追求而产生的，目标就是达到一个可控的吞吐量，由于与吞吐量关系密切，PS收集器也被称为吞吐量优先收集器，其提供两个参数量精确控制吞吐量，分别控制最大垃圾收集停顿时间：-XX:MaxGCPauseMillis参数，以及直接设置吞吐量大小的：-XX:GCTimeRatio。GCTimeRatio参数计算规则，比如设置成19，那么允许最大时间就占总时间的5% = 1/(1+19)，默认是99，默认允许最大的GC时间占比为1%。

将其中一个参数设置好座位JVM的优化目标，那么JVM就可以根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大吞吐量，自适应调节策略是PS收集器与ParNew收集器的一个重要区别。

### 2.2.4 CMS收集器

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或B/S系统的服务端上，这类应用重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。



CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器更复杂一些，整个过程分为四个阶段：

* 初始标记，CMS Initial mark
* 并发标记，CMS concurrent mark
* 重新标记，CMS remark
* 并发清除，CMS concurrent sweep

其中，初始标记、重新标记这两个步骤仍然需要Stop The World。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快。并发标记阶段就是进行GC Roots Tracing的过程，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而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时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，从总体上来说CMS收集器的内存回收过程是与用户线程一起并发执行的，从上图中可以清楚看到CMS收集器的运作步骤中并发和需要停顿的时间。

CMS收集器无法处理浮动垃圾（Floating Garbage）可能出现的Concurrent Mode Failure失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行，伴随程序运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在再次收集中处理掉它们，只好留在下一次GC时清理，这一部分垃圾成为浮动垃圾。由于在垃圾收集阶段用户线程还需要运行，那么还需要预留足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供给并发收集时的程序运作使用。

在JDK 1.5默认设置下，CMS收集器当老年代使用68%空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当提供参数：

*-XX:CMSInitiatingOccupancyFraction*

的值来提高触发百分比，一次降低内存回收次数从而获取更好的性能。在JDK 1.6中，CMS收集器的自动阈值已经提升至92%。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次Concurrent Mode Failure失败，这时虚拟机将启动后备方案，临时启用Serial Old收集器重新进行老年代的垃圾收集，这样停顿就很长，所以上面的参数设置的太高很容易导致Concurrent Mode Failure失败，性能反而降低。

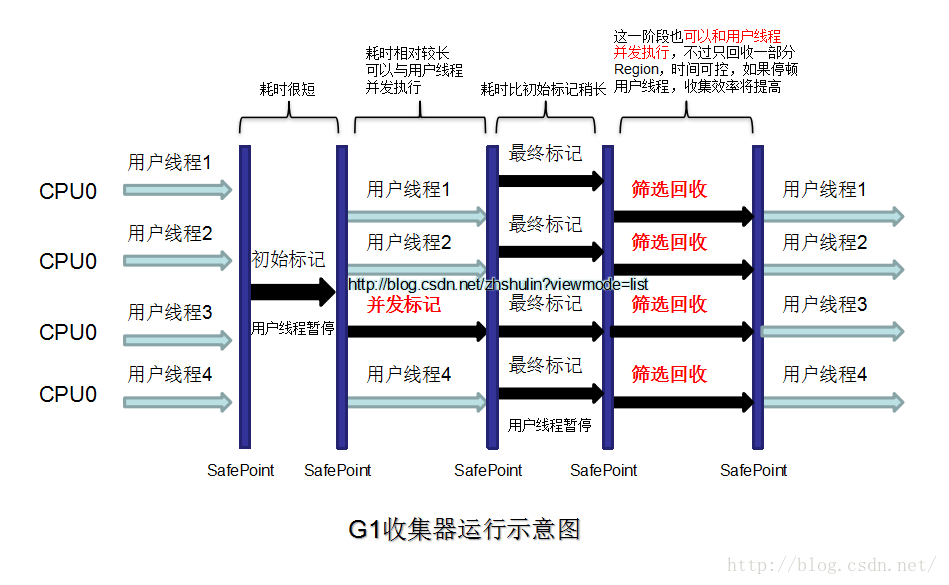
CMS收集器适用于处理很多交互任务的情况，方法区回收一般使用CMS，配置的两个参数：*-XX:+CMSPermGenSweepingEnabled, -XX:+CMSClassUnloadingEnabled*。

### 2.2.5 G1收集器

G1(Garbage-First)收集器是当前收集器技术最前沿的成果，直到JDK 7U4后才达到足够成熟的商用程度，其是面向服务端应用的垃圾收集器，其使命是可以替换掉CMS收集器，与其他GC收集器相比，具备以下特点：

* 并行与并发，G1能充分利用多CPU，多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿的时间，部分其他收集器原本需要停顿的Java线程进行GC动作，G1收集器仍然可以通过并发的方式让Java程序继续进行
* 分代收集，虽然G1可以不需要与其他收集器配置就能独立管理整个GC堆，但是它能够采用不同的方式去处理新创建的对象和已经存活一段时间，熬过多次GC的旧对象以获取更好的收集效果。
* 空间整合，与CMS的标记-清理算法不同，G1从整体上看是基于标记-整理算法实现的收集器，从局部（两个Region之间）上来看是基于复制算法来实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能够提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC
* 可预测的停顿，这是G1相对于CMS的另一个大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿之外，还建立可预测的停顿时间模型，让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不超过N毫秒。

下图是G1收集器运行示意图：



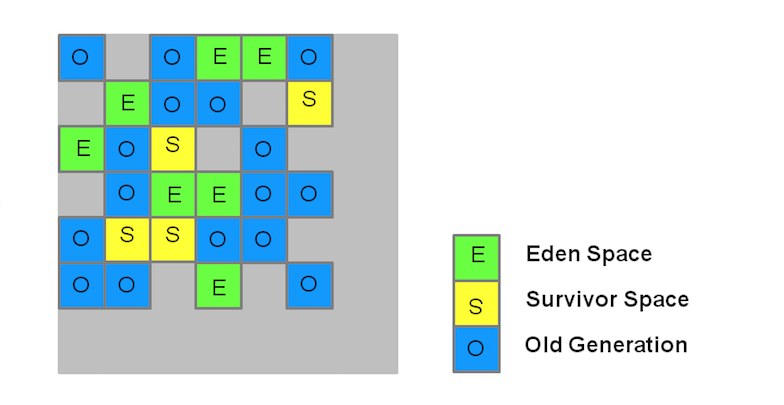
运作过程大致可划分以下几个步骤：

* 初始标记，Initial Marking
* 并发标记，Concurrent Marking
* 最终标记，Final Marking
* 筛选回收，Live Data Counting and Evacuation

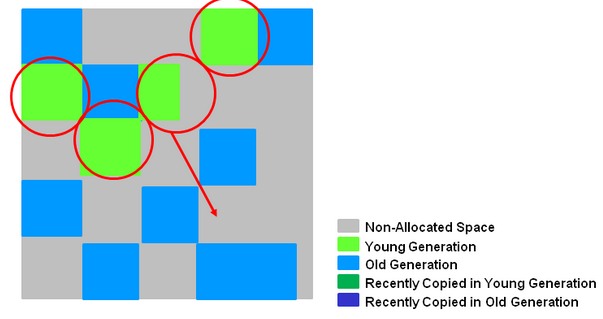
前几个步骤与CMS收集器相似。在筛选回收节点首先对Region的回收价值和成本进行排序，根据用户期望的GC停顿时间来制定回收计划。

G1是一款面向服务端应用的收集器，主要目标用于配备多颗CPU服务器治理大内存。

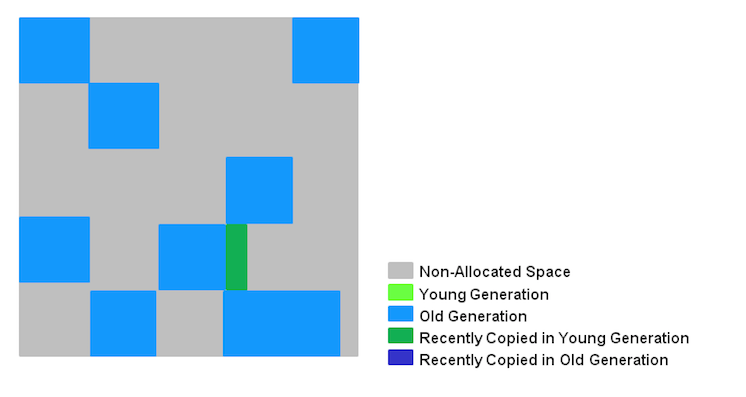
在G1之前其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样，使用G1收集时，Java堆的内存布局与其他收集器就有很大差别，将整个Java堆划分为多个大小相等的独立区域（Region），虽然保留新生代和老生代的概念，但是新生代和老生代不再是物理隔离，它们是一部分Region(不需要连续)的集合，如下图：



每块区域既有可能属于O区，也可能属于Y区，因此不需要一次对整个老年代/新生代进行回收。而是当线程并发寻找可回收的对象时，有些区块包含可回收的对象要比其他区块多。虽然在清理这些区块时G1仍然需要暂停应用线程，但可以用相对较少的时间优先回收垃圾较多的Region。新生代收集如下：



G1新生代收集与ParNew类似，将存活的对象转移到一个或者多个Survivor Regions中，如果存活时间达到阈值，这部分对象就会被提升到老年代，回收后的结果如下图：



在Young GCs会有STW事件，进行时所有应用程序线程都被暂停，而且多线程并发GC。

# GC测试

GC优化的目标是提高其GC性能，主要包括两方面：

* 吞吐量（Throughput）

使用-XX:GCTimeRatio参数指定可接受的GC时间占比（目标吞吐量），吞吐量计算公式为：

*吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）*

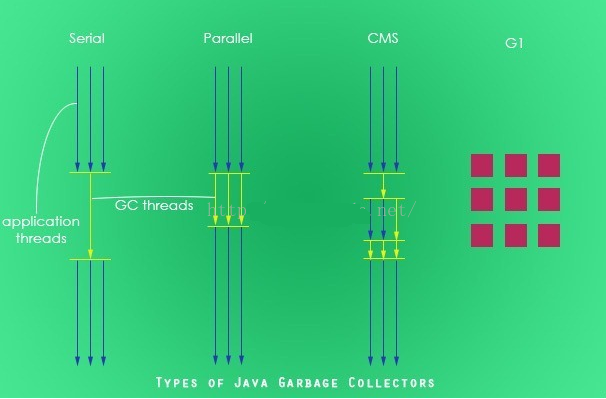
高吞吐量即减少垃圾收集时间，让用户代码获得更长的运行时间

* 应用停顿及延迟时间

包括GC最大的暂停时间，GC统计变量：平均暂停时间、标准差等。其他更复杂的统计（软实时目标）：

1. V%，表示测试过程中，软实时目标失败的概率，目标失败表示GC时间超过允许的最大GC时间，配置参数：-XX:MaxGCPauseMillis。
2. avgV%，表示在所有实际GC时间超标的时间片段里，实际GC时间超过最大的GC时间的平均百分比，实际GC时间减去允许最大GC时间，再除以总时间片段。
3. wV%，表示在测试结果最差的时间片段里，实际GC时间占用执行时间的百分比

下面对Java HotSpotVM中四种垃圾收集器进行性能测试：



1. Serial 收集器，单线程收集器，适用于单CPU主机及虚拟机内存不打的应用中，启用参数：-XX:+UseSerialGC
2. Parallel收集器，多线程收集器，在串行基础上采用多线程方式进行GC，弥补串行收集的不足，缩短停顿时间，-XX:+UseParallelOldGC
3. 并发收集器，多线程进行垃圾收集，在垃圾收集的时候不需要暂停应用，包括两种收集算法

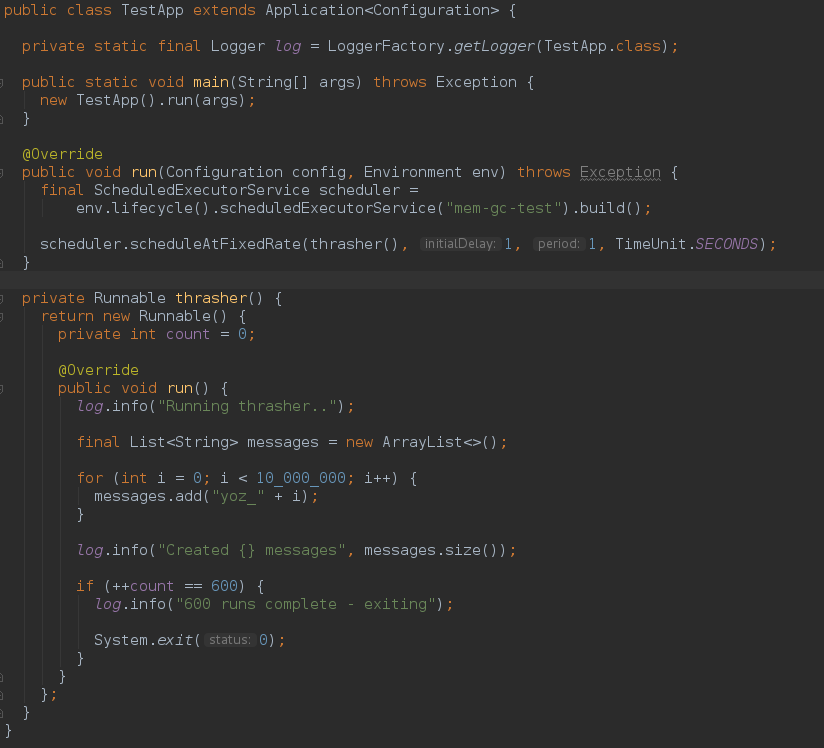
* CMS（Concurrent Mark Sweep），基于标记-清除算法，使用多线程算法区扫描堆（标记）并对发现的未使用的对象进行回收（清除）。初始标记、重新标记两个步骤需要STW：初始标记这是标记一下GC Roots能直接关联到的对象；重新标记，为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录。缺点是会产生大量空间碎片。-XX:+UseConcMarkSweepGC
* G1，基于标准-清除算法，针对多处理大容量内存的服务器端的垃圾收集器，目标是实现高吞吐量的同时，尽可能的满足收集暂停时间的要求。G1使用暂停时间预测模型使得暂停时间控制在用户指定的暂停时间内，并根据用户指定的暂停时间来选择合适的区域回收内存。-XX:+UseG1GC

## 3.1 测试程序及运行

测试程序使用Dropwizard框架，旨在为Web应用所需要的功能提供高性能、可靠的实现，程序执行命令如下：

*java -Xmx2g -Xloggc:gc\_serial.log -XX:+UseSerialGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps com.fys.TestApp server*

源码如下：



使用GCViewer对生成的GC日志进行处理，生成统计数据，命令如下：

*./run\_gcviewer.sh serial*

## 3.2 测试结果

统计结果如下所示:

表格1：-Xmx2g，执行时间5分钟，每轮写入消息条数：5.000,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GC算法 | Serial | Parallel | CMS | G1 |
| 占用总量 | 1945 | 1928 | 1944 | 2048 |
| 平均暂停 | 0.28353 | 0.10062 | 0.20142 | 0.0886 |
| 最小暂停 | 0.00437 | 0.00166 | 0.00489 | 0.00154 |
| 最大暂停 | 1.28869 | 1.41787 | 3.34579 | 1.01982 |
| 总暂停时间 | 119.65 | 101.82 | 110.78 | 138.13 |
| 吞吐量 | 80% | 84.00% | 81.53% | 76.98% |

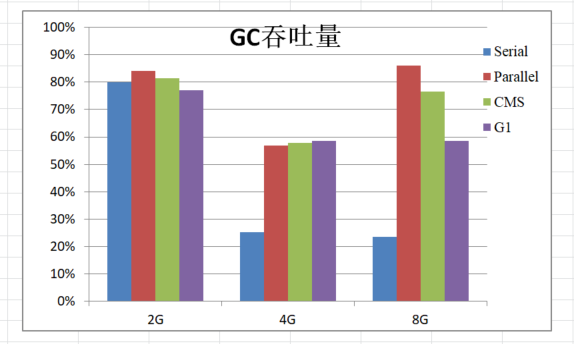
表格2：-Xmx4g，执行时间10分钟，每轮写入消息条数：10,000,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GC算法 | Serial | Parallel | CMS | G1 |
| 占用总量 | 2286 | 3641 | 3959 | 4096 |
| 平均暂停 | 0.81596 | 0.24596 | 0.4146 | 0.11451 |
| 最小暂停 | 0.0331 | 0.0049 | 0.00191 | 0.00166 |
| 最大暂停 | 3.06851 | 2.60713 | 3.41563 | 0.68087 |
| 总暂停时间 | 709.07 | 259 | 254.9 | 249.51 |
| 吞吐量 | 25% | 56.85% | 51.94% | 58.44% |

表格3：-Xmx8g，执行时间20分钟，每轮写入条数：10,000,000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GC算法 | Serial | Parallel | CMS | G1 |
| 占用总量 | 2288 | 4218 | 8000 | 8192 |
| 平均暂停 | 0.83424 | 0.20114 | 0.40625 | 0.17414 |
| 最小暂停 | 0.02332 | 0.01496 | 0.00194 | 0.00254 |
| 最大暂停 | 2.69648 | 3.91097 | 8.22612 | 0.97166 |
| 总暂停时间 | 1449 | 166.75 | 279.9 | 497.34 |
| 吞吐量 | 24% | 86.11% | 76.67% | 58.58% |

**1）吞吐量的测试结果统计图如下：**



Parallel GC的吞吐量在几轮测试中都是最高的，G1在内存越大的情况下吞吐量会有所下降。

**2）GC Pause统计结果汇总**

|  |  |
| --- | --- |
|  |  |
|  | |

**3）其他**

CMS收集器无法处理浮动垃圾（Floating Garbage）可能出现的Concurrent Mode Failure失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行，伴随程序运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在再次收集中处理掉它们，只好留在下一次GC时清理，这一部分垃圾成为浮动垃圾。由于在垃圾收集阶段用户线程还需要运行，那么还需要预留足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供给并发收集时的程序运作使用。

在JDK 1.5默认设置下，CMS收集器当老年代使用68%空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当提供参数：

*-XX:CMSInitiatingOccupancyFraction*

的值来提高触发百分比，一次降低内存回收次数从而获取更好的性能。在JDK 1.6中，CMS收集器的自动阈值已经提升至92%。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次Concurrent Mode Failure失败，这时虚拟机将启动后备方案，临时启用Serial Old收集器重新进行老年代的垃圾收集，这样停顿就很长，所以上面的参数设置的太高很容易导致Concurrent Mode Failure失败，性能反而降低。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GC算法 | Serial | Parallel | CMS | G1 |
| 占用总量 | 1945 | 1928 | 1944 | 2048 |
| 平均暂停 | 0.28353 | 0.10062 | 0.20142 | 0.0886 |
| 最小暂停 | 0.00437 | 0.00166 | 0.00489 | 0.00154 |
| 最大暂停 | 1.28869 | 1.41787 | 3.34579 | 1.01982 |
| 总暂停时间 | 119.65 | 101.82 | 110.78 | 138.13 |
| 吞吐量 | 80% | 84.00% | 81.53% | 76.98% |