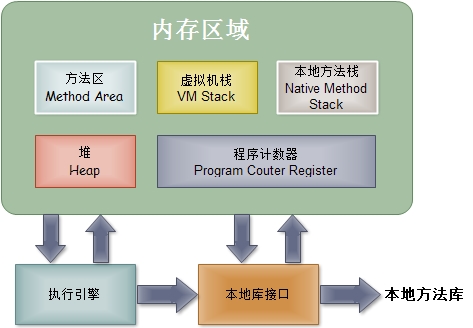
JVM内存分配与垃圾回收

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：

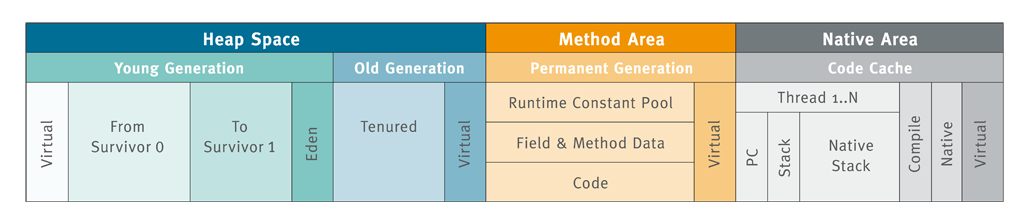
1. 方法区， 所有的类级别的数据都存储在这里，包括静态变量，每个JVM只有一个方法区，并且它是共享资源。类加载器将数据加载到这个数据区中。这个区域也成为“永久代”。
2. 堆区域，所有对象及其对应的实例和数组都存储在这里，每个JVM只有一个堆区域，由于方法和堆区域共享多个线程的内存，因此所存储的数据非线程安全。这个区域也就做GC堆，是虚拟机内存中管理的最大一块内存区域。这个区域被划分为新生代和老年代：新生代存储新创建的对象和尚未进入老年代的对象；老年代存储经过多次新声带GC仍然存活的对象。
3. 堆栈区，JVM为每个线程都会创建一个单独的运行时栈，对于每个方法调用，将在堆栈存储器中产生一个条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

# 方法区和堆区域

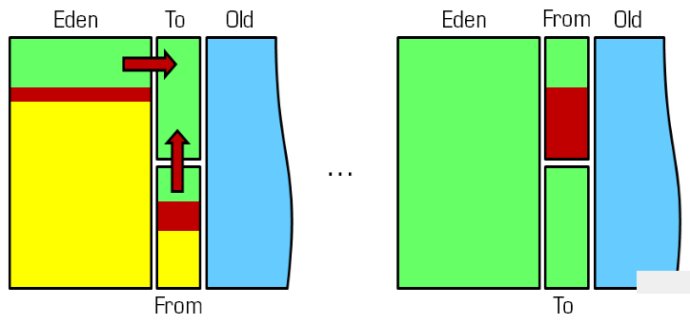
方法区和堆区域是JVM内存分配和垃圾回收的主要区域，内存模型如下图所示：



其中Perm区域为永久代，对应方法区，GC不会在住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。

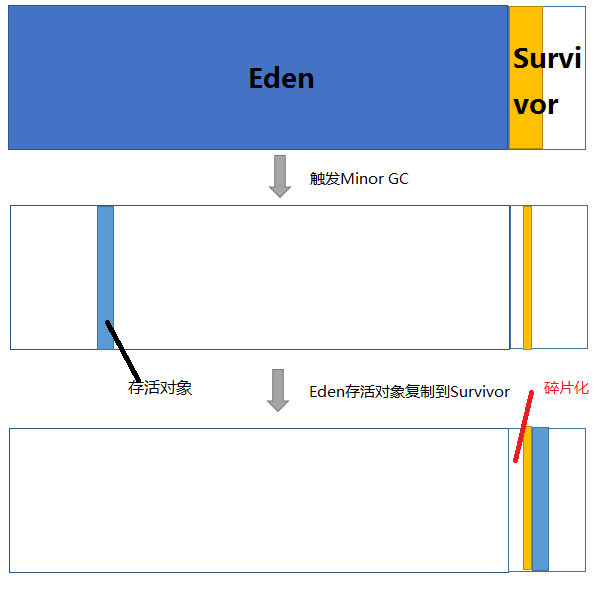
所有的对象在实例化后的整个运行周期内，都被存放在堆内存中，JVM中为了对短周期和长周期对象进行管理，采用分代的策略：

* **年轻代**是所有新对象生成的地方。当年轻代内存空间被用完后，会触发垃圾回收，这个垃圾回收称为Minor GC，这个区域内存相对会比较小，Minor GC会比较频繁。年轻代由分为1个Eden Space和2个Suvivor Space，默认比例为8:1。
* **Eden Space**，在大多数情况下，对象在新生代Eden区中分配，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。Eden Space作为使用空间。
* **Survivor**，对于GC来讲，Survivor相当于保留空间。如果没有Survivor，Edit区每进行一次Minor GC，存活的对象就会被送到老年代，老年代很快被填满，触发Major GC(Full GC)，这个过程执行时间较长，影响程序的执行和相应速度。Survivor存在的意义在于减少Full GC发生。Survivor区又分为From Suvivor 0和To Suvivor 1两个区，存在两个Suvivor区的意义在于解决内存碎片化的问题，如下图所示：



1. 在GC刚开始的时候，对象只存在Eden区和From Survivor区，To Survivor区是空的。
2. 进行GC的时候，Eden区所有存活的对象被复制到To Survivor区，From Survivor根据年龄值判断，当超过一定值（年龄阈值）的对象移动到年老代，没有达到阈值的复制到To Suvivor，经过这次GC后，Eden和From区已经被清空。
3. From Suvivor和To Suvivor区域交换他们的角色，新的To Suvivor是被上次被清空的From Suvivor，这样不管怎样，都保证To Suvivor是空的。
4. Minor GC一直重复2.3过程，直到To Suvivor区被填满，然后将所有对象移动到年老代中。

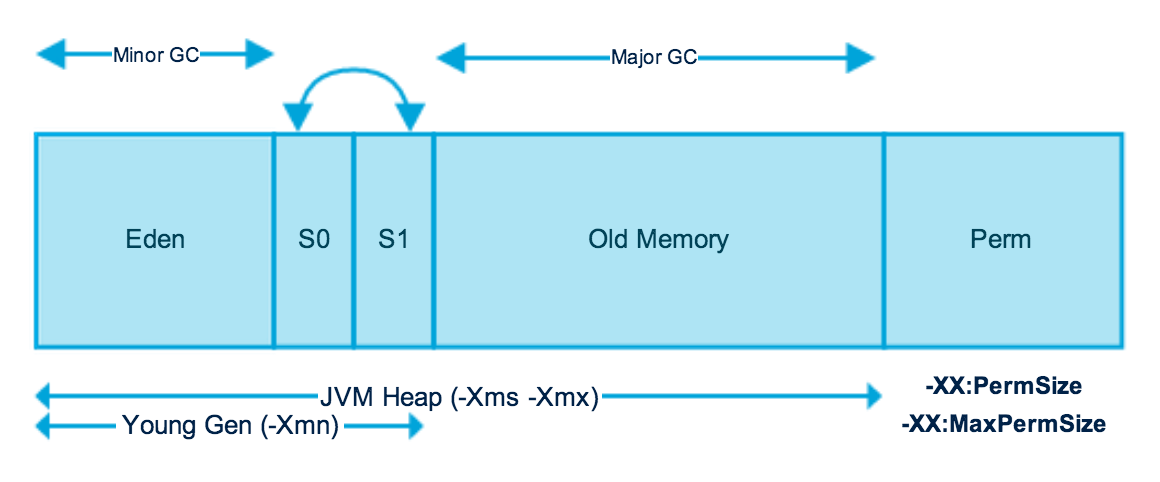
如果只有一个Suvovor，一旦Eden满触发Minor GC，存活的对象移动到Survior区。下一次Eden满了以后，Eden和Survivor各有一些存活对象，此时再把Eden区的存活对象硬放到Survivor区，这两部分对象所占用的内存是不连续的（一部分Survivor中超过年老阈值的对象会移动到永久区），如下图所示：



从而导致了内存碎片化。

* **年老代**，年老代内中包含了长期存活的对象和经过多次Minor GC后依然存活下来的对象，当年老代内存被占满时进行Major GC（Full GC）。Major GC过程中多有的应用线程都停下来，GC会检查所有存活的对象，因此消耗时间较长，造成应用反应迟钝。

JVM内存相关配置如下图所示：



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

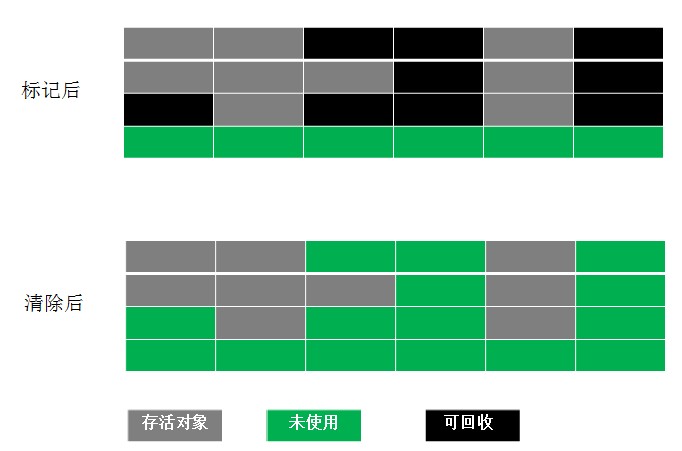
Major GC通常和Full GC是等价的，收集整个GC堆.

## 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）

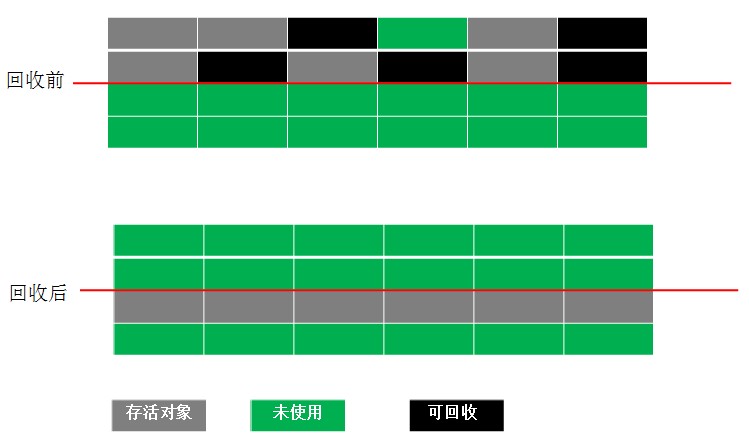
这个是最基础的垃圾回收算法，分为两个阶段：标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间，如下图所示：



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）

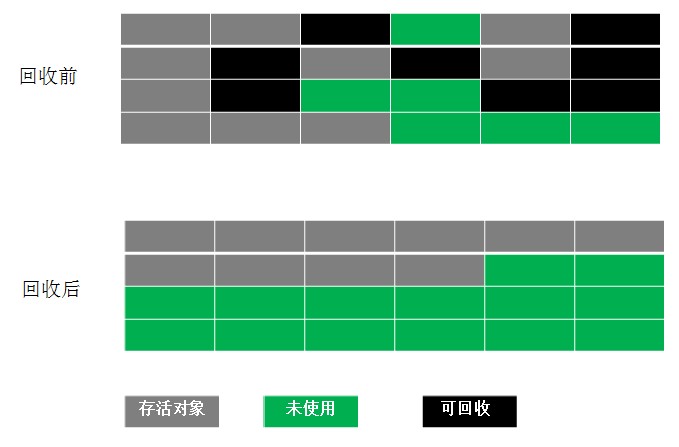
为了解决Mark-Sweep算法内存碎片化的缺陷而被提出来的算法，按内存容量将内存划分为等大小的两块，每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉，如下图：



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

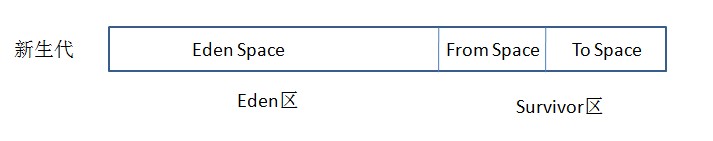
结合以上两个算法，为了避免缺陷，标记阶段和Mark-Sweep算法相同，标记后不是清理对象，而是将存活对象移向内存的一端，然后清除端边界外的内存，示意图如下所示：



### 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。

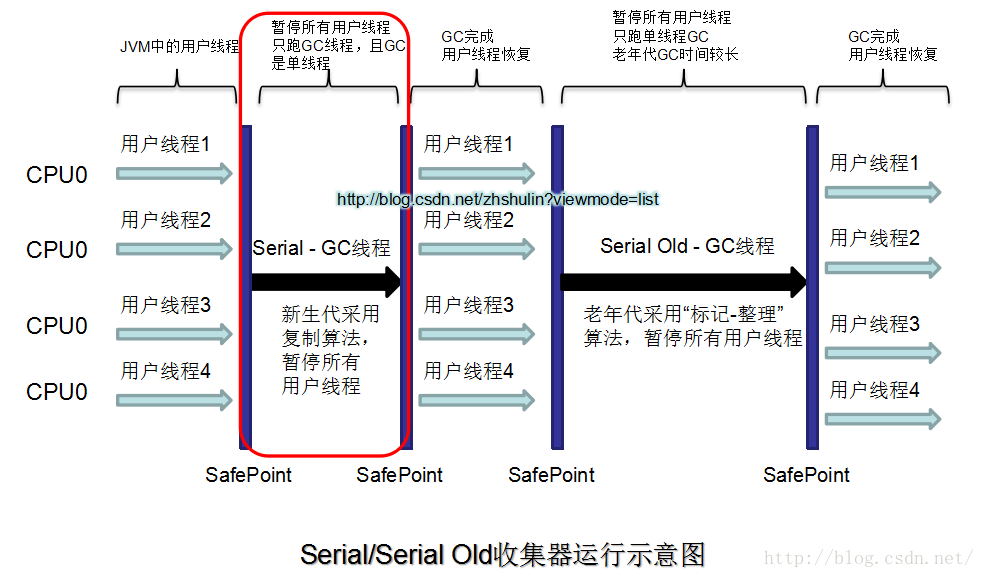
对象的内存分配主要在新生代的Eden Space和Survivor Space的From Space(Survivor目前存放对象的那一块)，少数情况会直接分配到老生代。当新生代的Eden Space和From Space空间不足时就会发生一次GC，进行GC后，Eden Space和From Space区的存活对象会被挪到To Space，然后将Eden Space和From Space进行清理。如果To Space无法足够存储某个对象，则将这个对象存储到老生代。在进行GC后，使用的便是Eden Space和To Space了，如此反复循环。当对象在Survivor区躲过一次GC后，其年龄就会+1。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



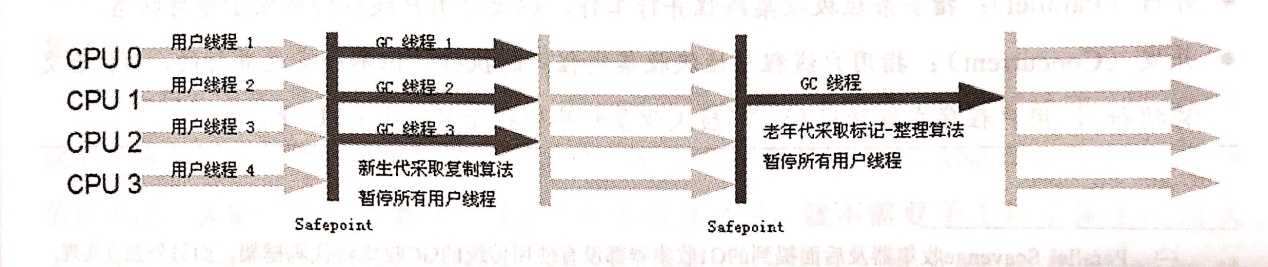
JDK 1.5之前与Parallel Scavenge收集器搭配使用。其作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时启用。

### 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：

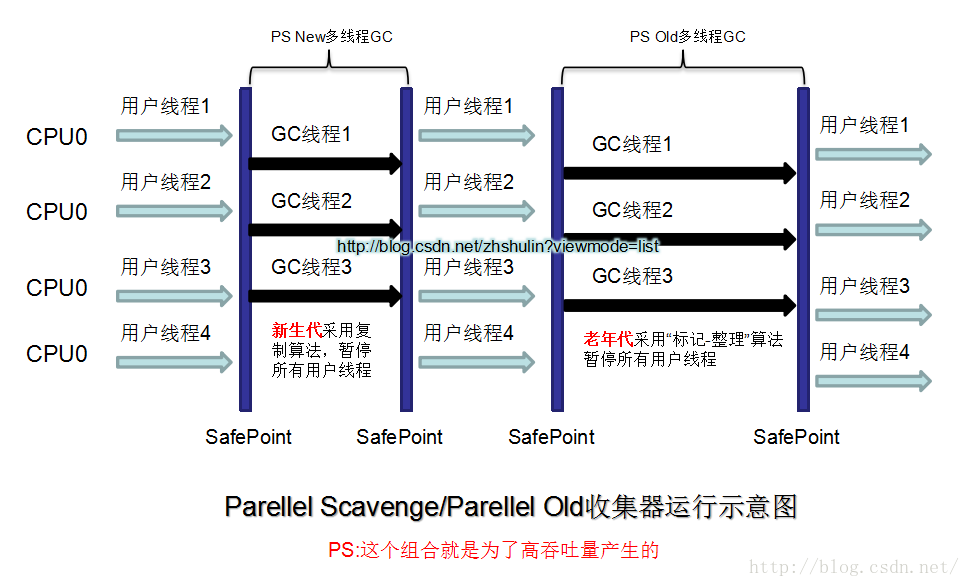


ParNew收集器除了多线程外，其他和Serial差不多，ParNew是许多运行在Server模式下的JVM中首选的垃圾收集器，一个重要的原因是除了Serial，它是唯一可以和CMS(Concurrent Mark Sweep)老年代收集器配合工作。

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：

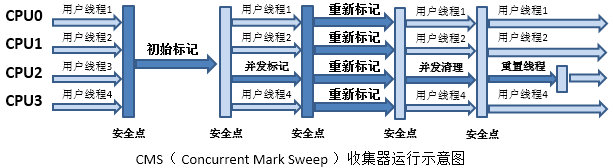


PS收集器正是基于对吞吐量的追求而产生的，目标就是达到一个可控的吞吐量，由于与吞吐量关系密切，PS收集器也被称为吞吐量优先收集器，其提供两个参数量精确控制吞吐量，分别控制最大垃圾收集停顿时间：-XX:MaxGCPauseMillis参数，以及直接设置吞吐量大小的：-XX:GCTimeRatio。GCTimeRatio参数计算规则，比如设置成19，那么允许最大时间就占总时间的5% = 1/(1+19)，默认是99，默认允许最大的GC时间占比为1%。

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。

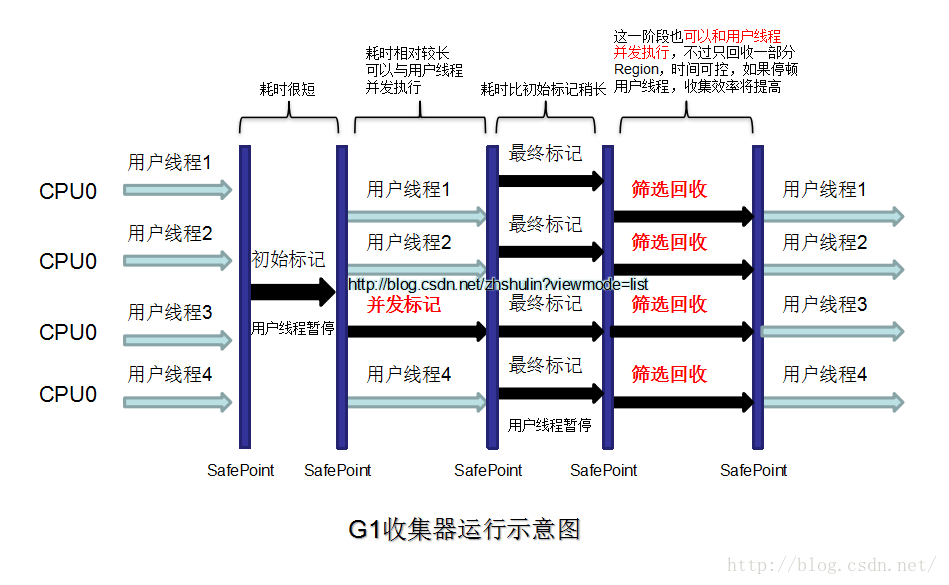
CMS收集器适用于处理很多交互任务的情况，方法区回收一般使用CMS，配置的两个参数：*-XX:+CMSPermGenSweepingEnabled, -XX:+CMSClassUnloadingEnabled*。

### 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：



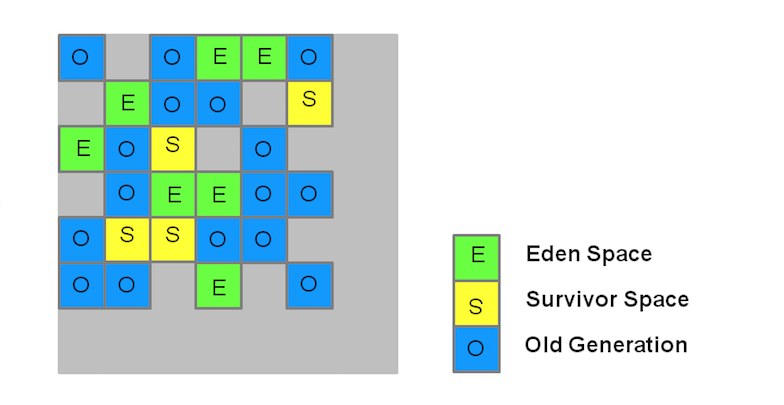
运作过程大致可划分以下几个步骤：

* 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

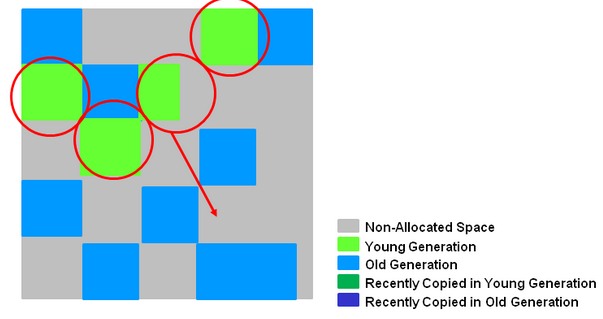
前几个步骤与CMS收集器相似。在筛选回收节点首先对Region的回收价值和成本进行排序，根据用户期望的GC停顿时间来制定回收计划。

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。

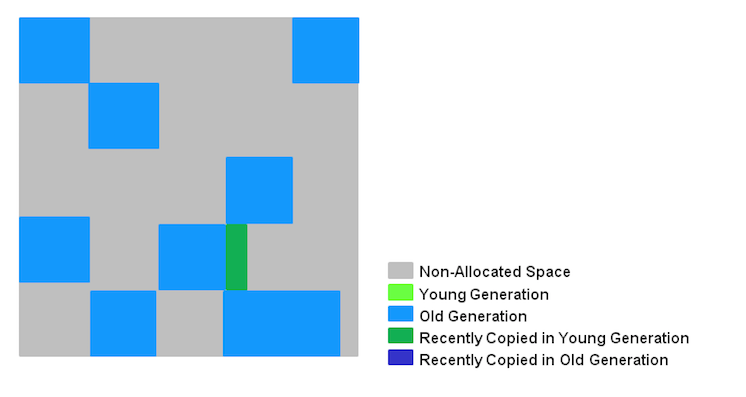
在G1之前其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样，使用G1收集时，Java堆的内存布局与其他收集器就有很大差别，将整个Java堆划分为多个大小相等的独立区域（Region），虽然保留新生代和老生代的概念，但是新生代和老生代不再是物理隔离，它们是一部分Region(不需要连续)的集合，如下图：



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：



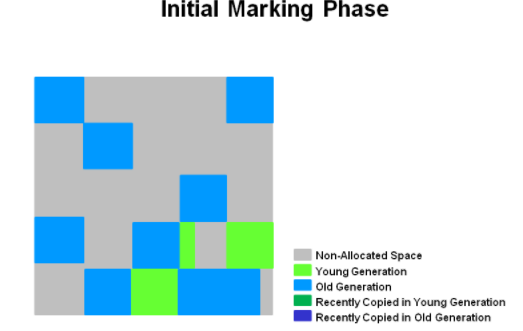
G1新生代收集与ParNew类似，将存活的对象转移到一个或者多个Survivor Regions中，如果存活时间达到阈值，这部分对象就会被提升到老年代，回收后的结果如下图：



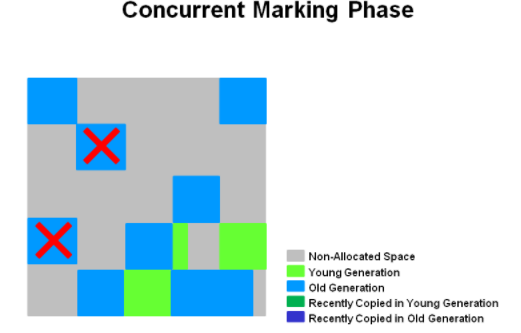
在Young GCs会有STW事件，进行时所有应用程序线程都被暂停，而且多线程并发GC。

和CMS 收集器类似，G1收集器在老生代收集上也尽可能降低暂停时间，下面是老生代收集的步骤：

1. 初始标记阶段，在G1中该操作附带一次年轻代GC，以标记Survivor中有可能引用到的老生代对象的Regions

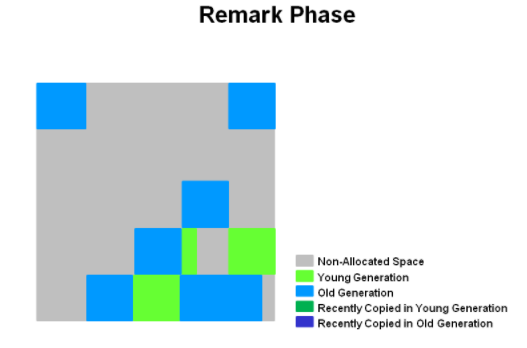


1. 扫描根区域：Root Region Scanning，与应用程序并发执行，扫描Survivor中能够引用到老年代的references，但是必须在Minor GC触发前执行完
2. 并发标记阶段，与应用程序并发执行，在这个对中查找存活对象，该阶段可能被Minor GC中断

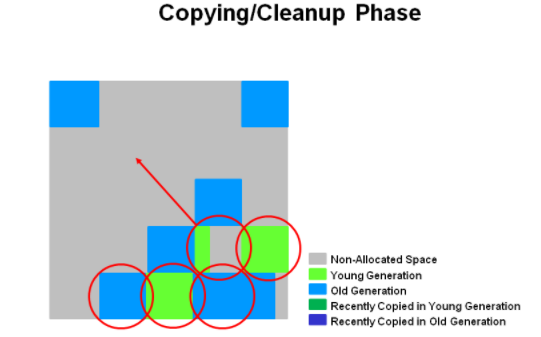


如果发现空region，则标记为X，在Remark阶段remove掉，当然也会计算生命周期。

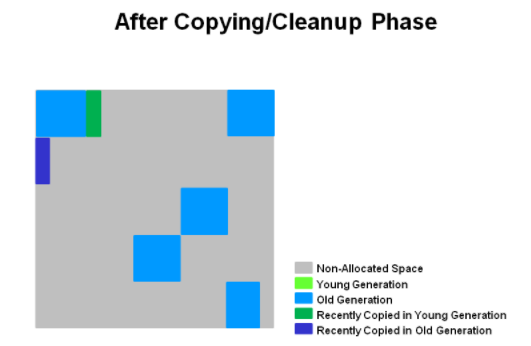
1. Remark，完成堆内存中存活对象的标记，使用SATB(Snapshot-at-the-beginning)算法，比CMS所用算法要快很多，空Region直接被移除并回收，并计算所有区域的活跃度



1. Copying/Cleanup 阶段，依次执行：在含有存活对象和完全空闲区域上进行统计，擦除Remembered Sets，重置空regions并将它们返还给空闲列表



1. After Copying/Cleanup，选择活跃度最低的区域，将拷贝/转到存活对象到新的尚未使用的regions，该阶段被记录在gc log中，只发生年轻代GC Pause(young)，与老年代一起执行则被记录为[GC Pause Mixed]。



在G1收集器中，Region之间的对象引用以及其他收集器中的新生代和老年代之间的对象引用都是使用Remembered Set来避免扫描全堆。G1中每个Region都有一个与之对应的Remembered Set, VM发现程序对Reference类型数据进行写操作时, 会产生一个Write Barrier暂时中断写操作, 检查Reference引用的对象是否处于不同的Region中(在分代例子中就是检查是否老年代中的对象引用了新生代的对象), 如果是, 便通过CardTable把相关引用信息记录到被引用对象所属的Region的Remembered Set中. 当内存回收时, 在GC根节点的枚举范围加入Remembered Set即可保证不对全局堆扫描也不会有遗漏。

对G1的使用配置参数如下：

* -XX:+UseG1GC
* -XX:MaxGCPauseMillis=50，暂停时间目标
* -XX:GCPauseIntervalMillis=200，暂停间隔目标
* -XX:+G1YoungGenSize=512m，年轻代大小
* -XX:SurvivorRatio=9，幸存区比例

### 2.2.6 GC机制的总结

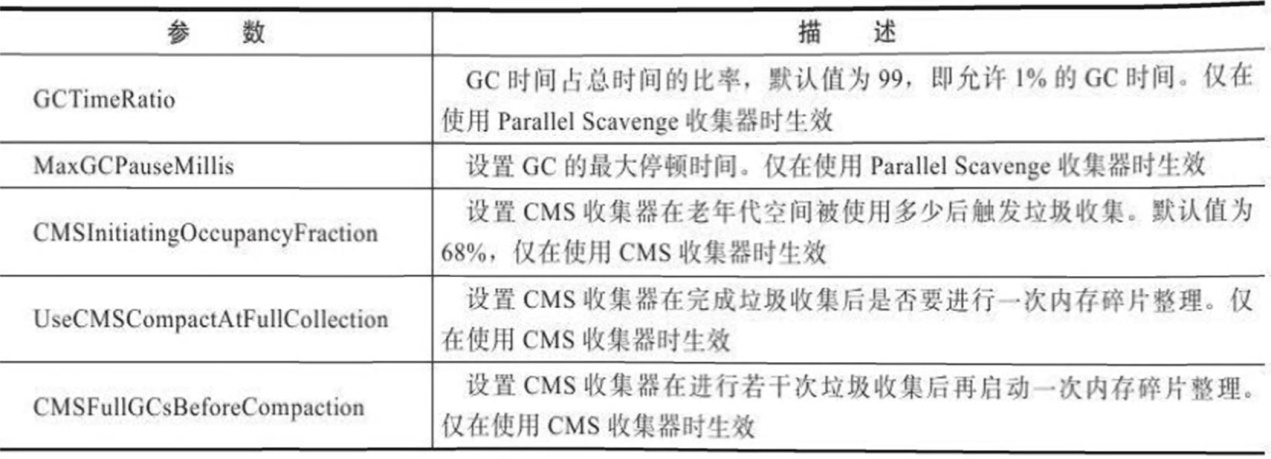
以上各种GC机制需要组合使用，指定方式如下所示：



以上介绍的集中GC，从收集器着重点上可以进行如下分类：

* 串行垃圾收集器，最基本的垃圾收集器，进行垃圾回收时会STW
* 吞吐量垃圾收集器，Parallel Scavenge垃圾收集器，达到一个可控制的吞吐量。
* 并发低停顿收集器（低延迟垃圾收集器），CMS，只有ParNew能与CMS收集器配合，通过参数-XX:+UseConcMarkSweepGC来设置，默认使用ParNew。CMS垃圾收集器以获取最短回收停顿时间为目标。
* 垃圾首先收集器（G1），独立管理这个GC堆，包括新生代和老生代，不需要与其他收集器搭配。在实现低停顿的同时实现高吞吐量。可配置参数：-XX:+UseG1GC

其他参数：



http://blog.csdn.net/tjiyu/article/details/53983650

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html#t5

# JVM GC性能监控工具

在JDK中提供了多种监控工具

## 3.1 jstat，虚拟机统计信息监控工具

jstat用于监视虚拟机各种运行状态信息的命令行工具，可以显示本地或远程虚拟机进程中的类装载、内存、GC及JIT编译等运行数据，只提供纯文本控制台服务，一般是运行期间定位虚拟机性能问题的首先工具，jstat命令格式：

*jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]*

对于命令格式中的VMID是虚拟机pid号，如果是远程虚拟机进程，那么VMID的格式应当是：

*[protocol:][//]lvmid[@hostname[:port]/servername]*

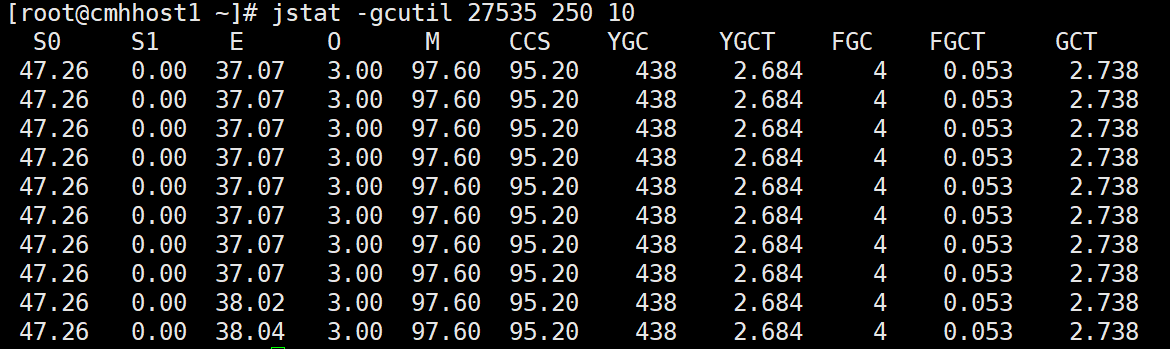
参数interval和count代表查询间隔和次数，如果省略这两个参数，说明只查询一次，例如每250毫秒查询一次进程2764 GC收集情况，一共查询20次，命令如下：

*jstat -gc 27535 250 20*

选项option代表用户希望查询的虚拟机信息，主要分为3类：类状态、垃圾回收，运行期编译状况

|  |  |
| --- | --- |
| 选项 | 作用 |
| -class | 监视类装载、卸载数量以及类装载所耗费的时间 |
| -gc | 监视Java堆状况，包括Eden区、两个Suvivor区、老生代、永久代等容量、已用空间、GC时间合计等信息 |
| -gccapacity | 监视内容与-gc基本相同，但输出主要关注Java堆各个区域使用到的最大和最小空间 |
| -gcutil | 监视内容与-gc基本相同，但输出主要关注已使用空间占总空间的百分比 |
| -gcnew | 监视新生代的GC状况 |
| -gcnewcapacity | 与-gcnew相同，输出主要关注使用到的最大，最小空间 |
| -gcold | 监视老生代的GC状况 |
| -gcoldcapacity | 与-gcold相同，输出主要关注使用到的最大，最小空间 |
| -gcpermcapacity | 输出永久代使用到的最大，最小空间 |
| -compiler | 输出JIT编译便器过的方法和耗时的信息 |
| -printcompilation | 输出已经被JIT编译的方法 |

下面是jstat gcutil监控的内存状态的例子：



各监控参数如下表所示：

|  |  |
| --- | --- |
| 结果 | 含义 |
| S0 | 新生代S0的已使用的百分比 |
| S1 | 新生代S1的已使用的百分比 |
| E | 新生代Eden区域使用的百分比 |
| O | 堆上的老生代使用空间的百分比 |
| M | 元数据区已使用空间的百分比 |
| YGC | 新生代的GC次数，也就是Minor GC次数 |
| YCGT | 新生代GC所用的时间 |
| FGC | Full GC次数 |
| FGCT | Full GC所用的时间 |
| GCT | 垃圾回收的总时间 |
| CCS | 类指针压缩空间使用率 |

## 3.2 VisualVM :多合一故障处理工具

Visual VM是目前随JDK发布的功能最强大的运行监视和故障处理工具，其除了提供运行监视、故障处理外，还提供了很多其他方面的功能，如性能分析（Profiling），功能如下：

* 显示虚拟机进程以及进程的配置、环境信息（jps,jinfo）
* 监视应用程序的CPU、GC、堆、方法区以及线程的消息（jstack,jstat）
* Dump以及分析堆转储快照（jmap,jhat）
* 方法级的程序运行性能分析，找出被调用最多、运行时间最长的方法
* 离线程序快照，收集程序的运行时配置、线程dump、内存dump等信息建立一个快照，可以将快照发送给开发者进行Bug反馈
* 其他plugins无线可能性

### 3.2.1 VisualVM监控远程JVM

VisualVM打开后就显示本地JVM，但是监控远程JVM需要进行一些额外的配置，目前支持的两种监控方式：jstatd方法和基于JMX方法。

1. jstatd方法，利用后台的RMI守护进程来实现对远程JVM的监控，方法如下：

* 创建监控策略文件，在{JDK\_HOME}/bin下，创建jstatd.all.policy文件，内容如下：

*grant codebase "file:${java.home}/../lib/tools.jar" {*

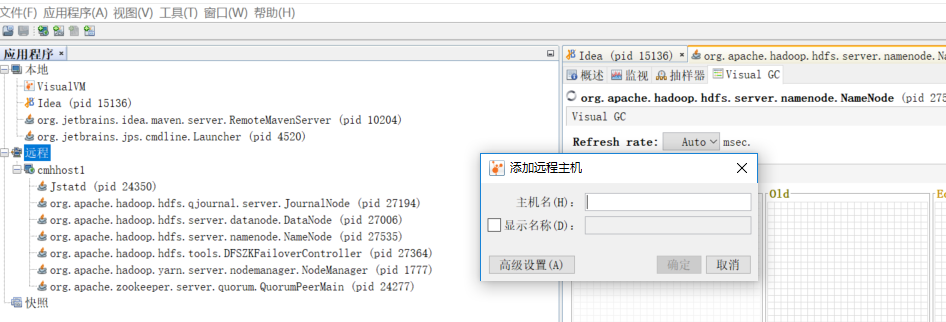
*permission java.security.AllPermission;*

*};*

* 在要监控的主机上，启动jstatd进程，要确认/etc/hosts配置，命令如下：

*jstatd -J-Djava.security.policy=jstatd.all.policy -J-Djava.rmi.server.hostname=10.139.4.82*

* 在监控VisualVM添加远程主机后，自动显示监控进程情况，但是在jstatd监控方式下，Sampler,MBeans和JConsole是不能启用的



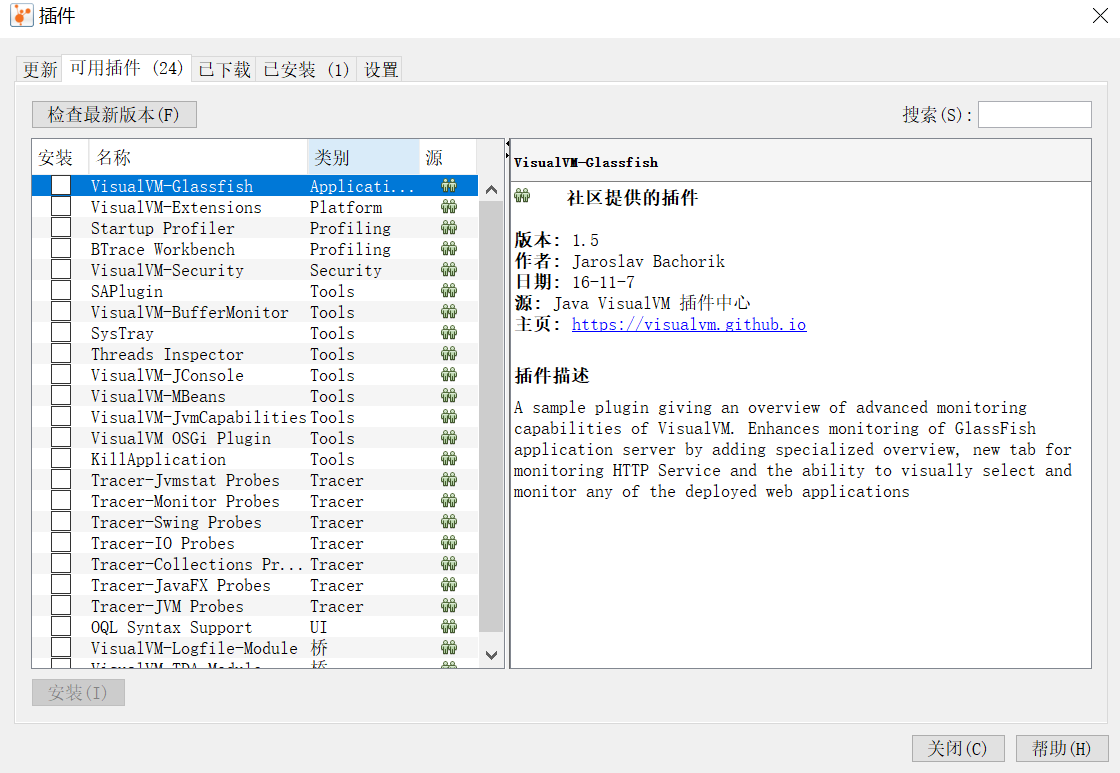
另外一种方法JMX不再介绍。

### 3.2.2 安装插件

在VisualVM中，可以根据自己的工作需要安装合适的插件，在有网络的环境下，操作如下:

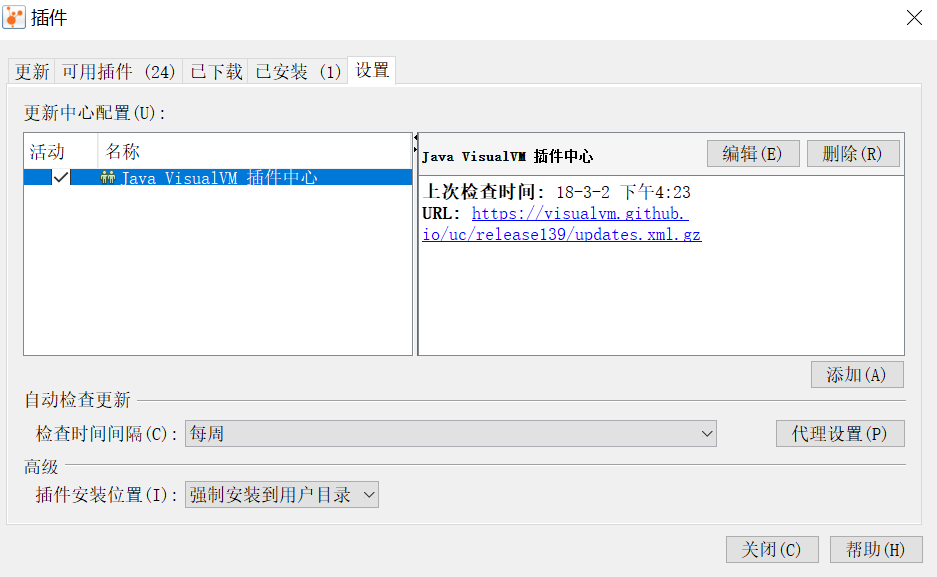
*工具 —> 插件菜单*

可以看到弹出的可用插件，其中列举了VisualVM可以使用的插件，如下图所示：



选中插件后，可在右边窗口显示插件的基本信息，如开发者、版本、功能描述等。

注意：在设置中配置visualvm插件的网址：



目前插件都放在github上进行托管，url为：

[*https://visualvm.github.io/uc/release139/updates.xml.gz*](https://visualvm.github.io/uc/release139/updates.xml.gz)

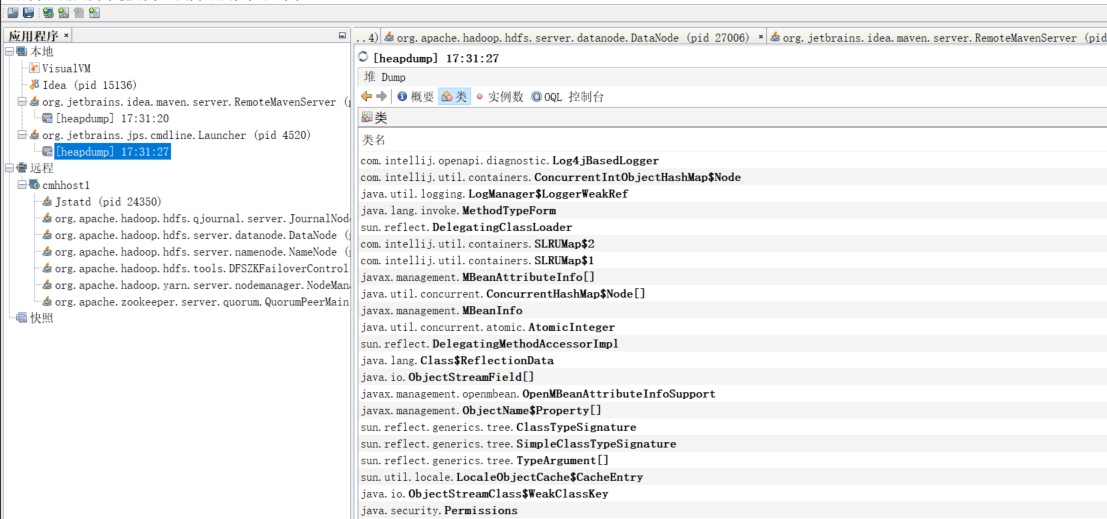
选择插件后，点击安装后重启VisualVM即可生效。

### 3.2.3 生成、浏览堆转储快照

在VisualVM中生成dump文件有两种方式，可以执行下列任一操作：

* 在”应用程序”窗口右键单击应用程序节点，然后选择“堆dump”
* 在”应用程序”窗口中双击应用程序节点以打开应用程序标签，然后在监视标签中单击”堆dump”

生成dump文件后，应用程序页签在该堆的应用程序下增加一个以[heapdump]开头的字节点，并且在主页签中打开了该转储快照，如下图所示：



如果需要把dump文件保存或者发送出去，在heapdump节点上右击”另存为”裁断，否则Visual VM关闭后生成的dump文件被当做临时文件删除掉。要打开一个已经存在的dump文件，通过文件菜单中的装入功能，选择硬盘上的dump文件即可。

但是不能对远程JVM进行dump文件。

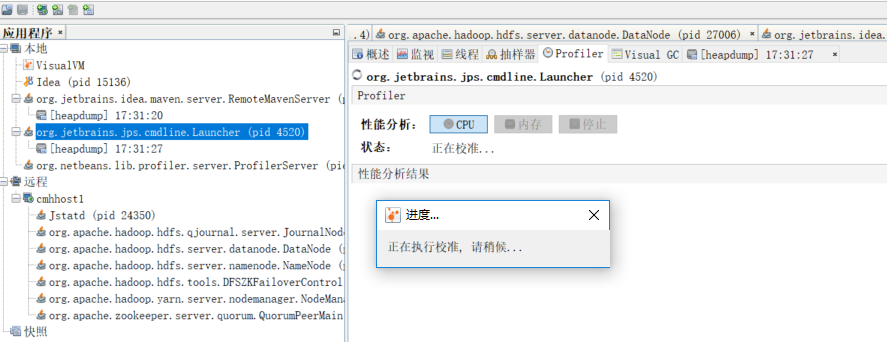
### 3.2.4 分析程序性能

在Porfile页签中，VisualVM提供了程序运行期间方法级的CPU执行时间分析以及内存分析，做Profiling分析堆程序运行性能有比较大的影响，所以一般不在生产环境中使用这个功能。

要开始分析，选择“CPU”和内存按钮中的一个，然后切换到应用程序对程序进行操作，VisualVM会记录到这段时间应用程序执行过的方法：

* CPU分析，会统计每个方法的执行次数，执行耗时
* 内存分析，统计每个方法关联的对象数及这些对象占用的空间

分析结束后，点击停止按钮结束监控过程，如下图所示：

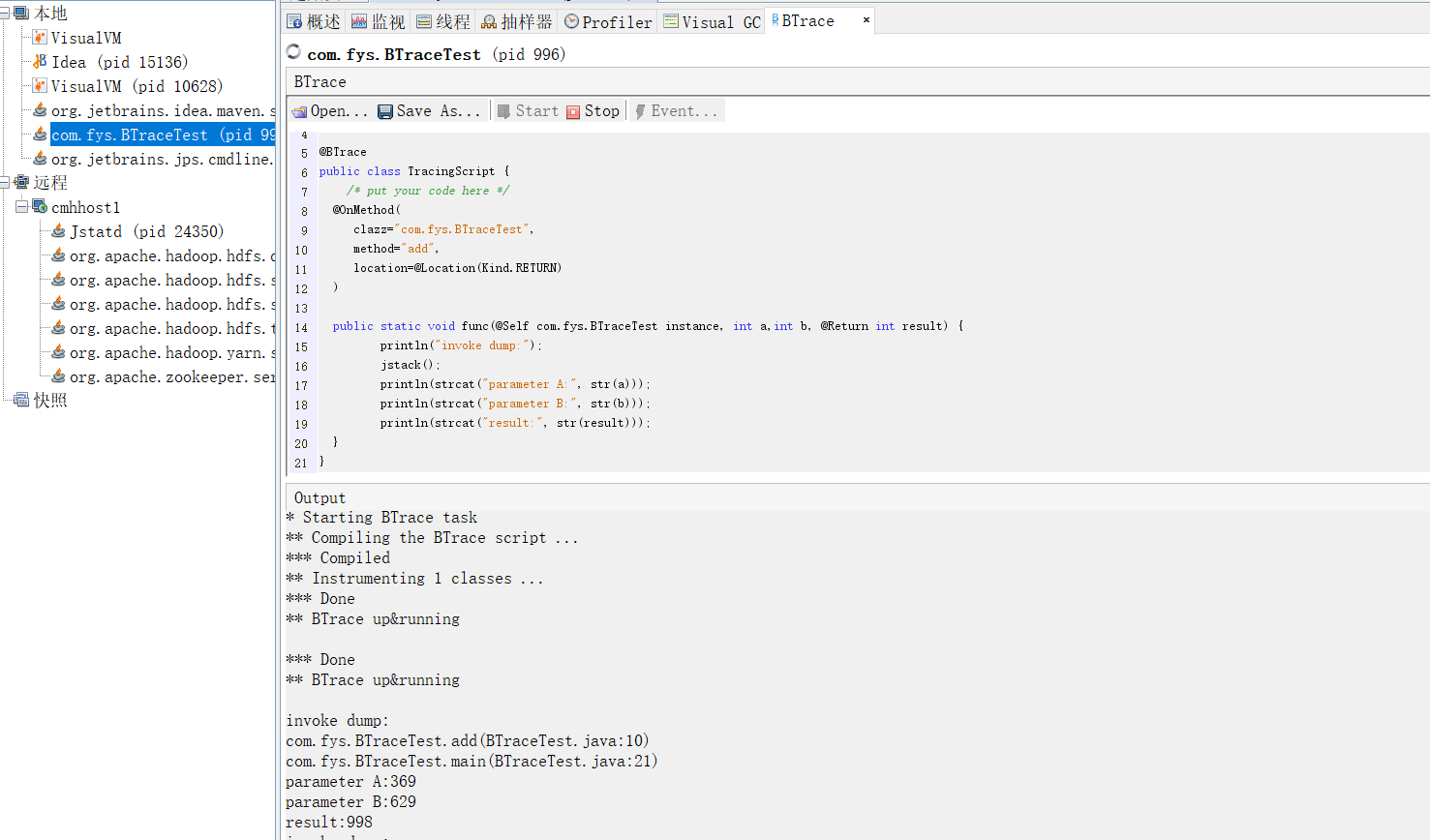


开发者可以对Eclipse IDE操作进行录制和分析，读者分析自己的应用程序时，可以根据实际的业务的复杂程度与方法的时间、调用次数做比较，找到有价值的方法。

### 3.2.5 BTrace动态日志跟踪

BTrace可以在不停止目标程序运行的前提下，通过HotSpot虚拟机的HotSwap技术动态加入原本不存在的调试代码。可以将程序中排查错误的一些信息，如方法参数、返回值等，在开发时没有打印到日志中。

在VisualVM中安装BTrace插件后，在应用程序面板右键要调试的程序，出现Trace Application菜单，点击进入Btrace面板。这个面板看起来像一个简单的Java程序开发环境，如下图所示：



测试代码：

*package com.fys;*

*import java.io.BufferedReader;*

*import java.io.IOException;*

*import java.io.InputStreamReader;*

*public class BTraceTest {*

*public int add(int a, int b) {*

*return a + b;*

*}*

*public static void main(String[] args) throws IOException {*

*BTraceTest test = new BTraceTest();*

*BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));*

*for(int i = 0; i < 10; i++) {*

*reader.readLine();*

*int a = (int) Math.round(Math.random() \* 1000);*

*int b = (int) Math.round(Math.random() \* 1000);*

*System.out.println(test.add(a,b));*

*}*

*}*

*}*

程序运行后，在Visual VM中打开该程序的监视，在BTraace页签填充TracingScript内容，如下所示：

*@BTrace*

*public class TracingScript {*

*/\* put your code here \*/*

*@OnMethod(*

*clazz="com.fys.BTraceTest",*

*method="add",*

*location=@Location(Kind.RETURN)*

*)*

*public static void func(@Self com.fys.BTraceTest instance, int a,int b, @Return int result) {*

*println("invoke dump:");*

*jstack();*

*println(strcat("parameter A:", str(a)));*

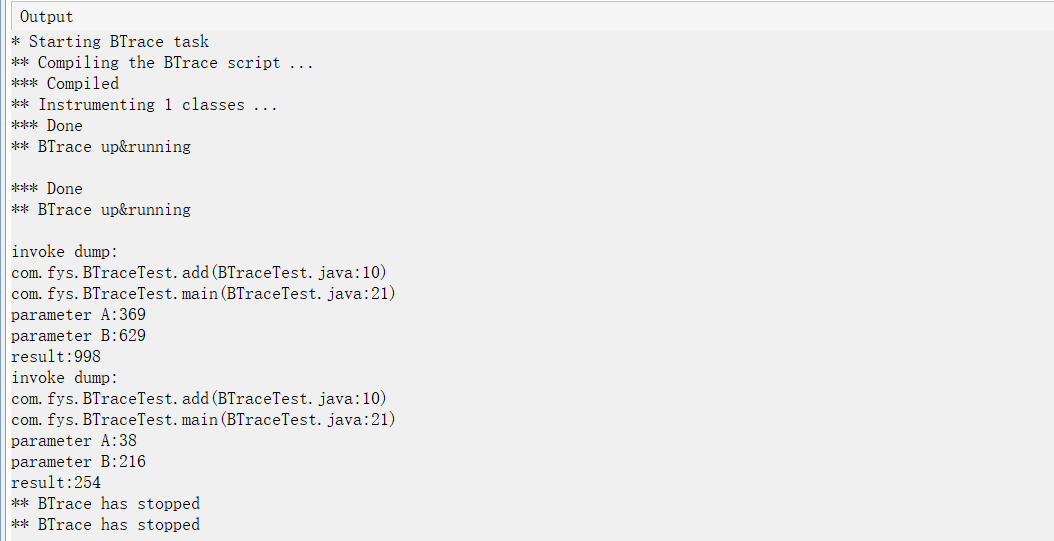
*println(strcat("parameter B:", str(b)));*

*println(strcat("result:", str(result)));*

*}*

*}*

点击”Start”按钮后，编译完成，可见Output面板中程序”BTrace Code successfule deployed”的字样。程序运行过程中在Output面板中输出入下调试信息：



BTrace的用法还有呢多，打印调用堆栈、参数、返回值只是最基本的应用，还可以使用BTrace进行性能监视，定位连接泄露和内存泄露、解决多线程竞争问题等例子。

参考文献：

https://www.ibm.com/developerworks/cn/java/j-lo-visualvm/

http://www.infoq.com/cn/news/2010/08/VisualVM1.3

http://www.knowsky.com/967679.html

http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

http://www.importnew.com/23035.html

http://www.importnew.com/15820.html

http://ifeve.com/under-the-hood-runtime-data-areas-javas-memory-model/

http://blog.sina.com.cn/s/blog\_b4e45e910102xmle.html

一步步优化JVM: http://blog.csdn.net/zhoutao198712/article/details/7842500

http://www.blogjava.net/paulwong/archive/2014/06/16/414812.html

http://blog.csdn.net/coslay/article/details/48918715

http://www.infoq.com/cn/articles/jdk7-garbage-first-collector

测试程序：

https://github.com/chewiebug/GCViewer