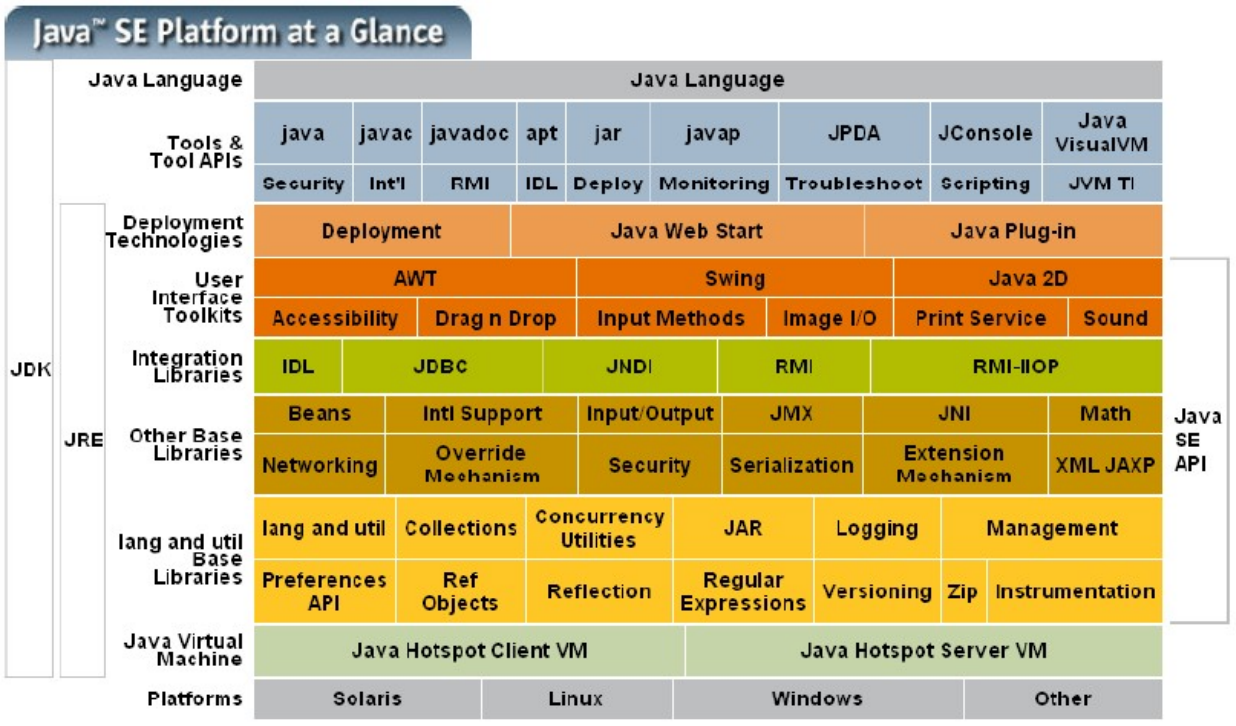
JVM、垃圾回收、内存调优、常见参数

一、什么是JVM

JVM是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

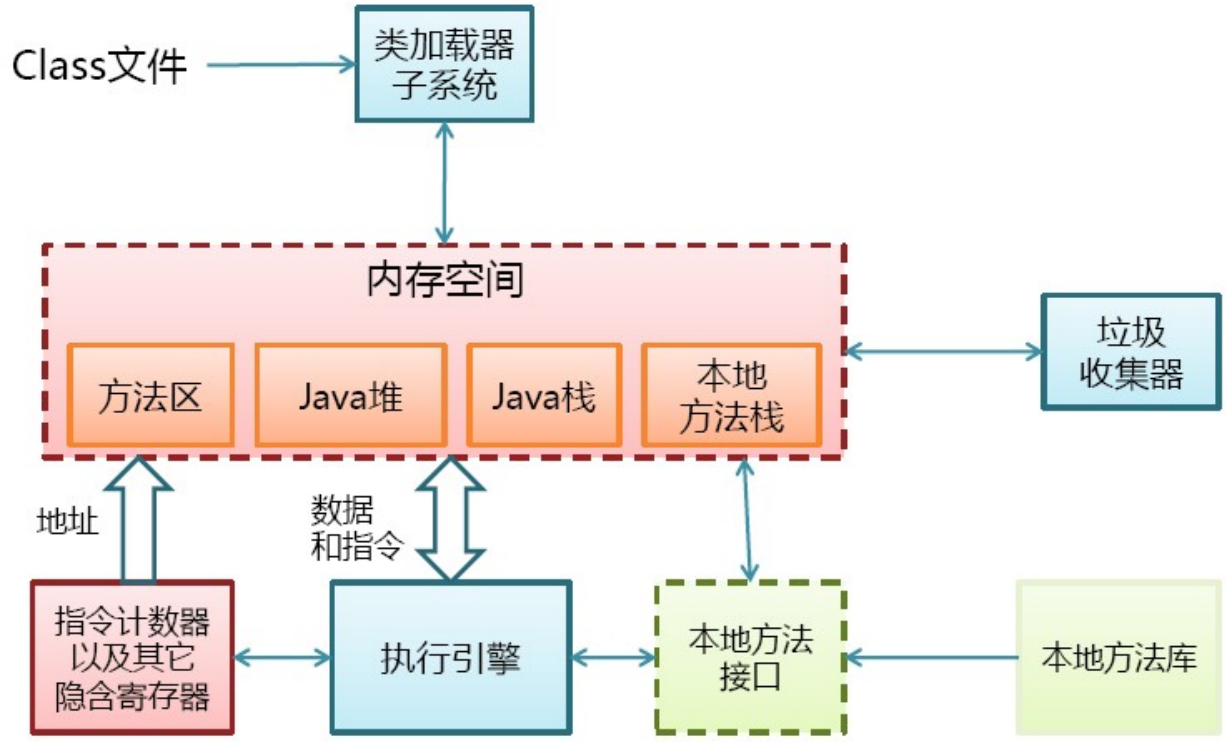
Java语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。这就是Java的能够“一次编译，到处运行”的原因。

从Java平台的逻辑结构上来看，我们可以从下图来了解JVM：



wKioL1Xs9MLDQRHuAAHLIb4NrvA724.gif

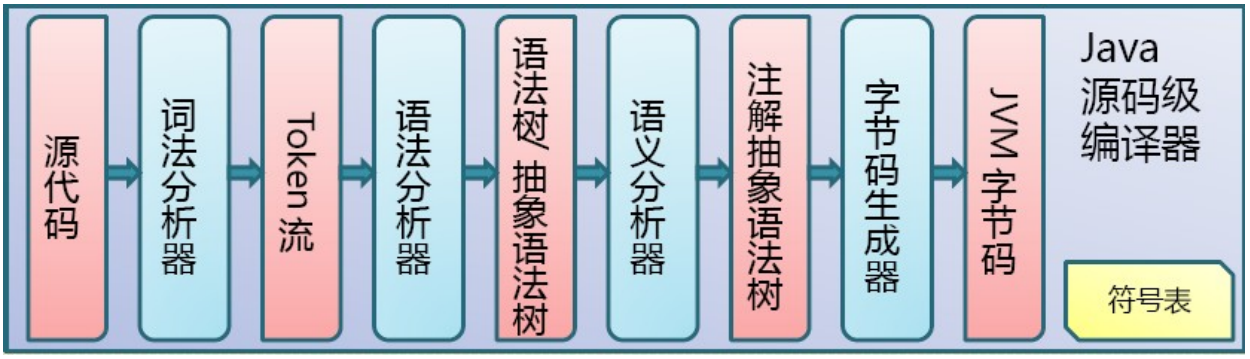
从上图能清晰看到Java平台包含的各个逻辑模块，也能了解到JDK与JRE的区别，对于JVM自身的物理结构，我们可以从下图鸟瞰一下：



wKiom1Xs8uyjeKCKAADr2P7KwZ4547.gif

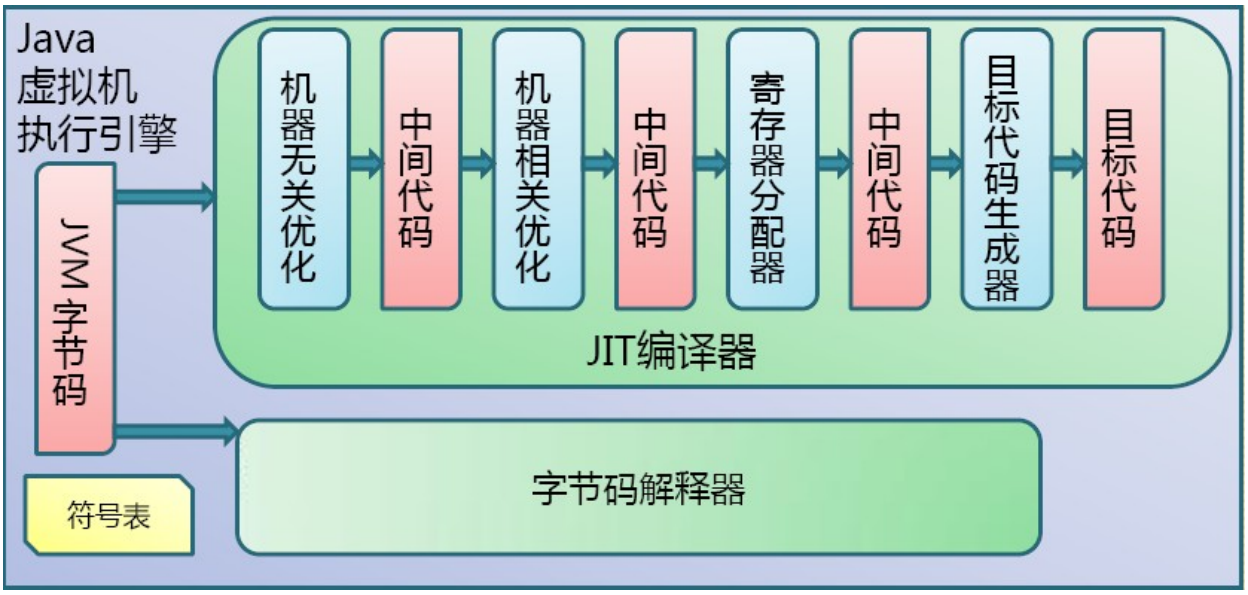
二、JAVA代码编译和执行过程

Java代码编译是由Java源码编译器来完成，流程图如下所示：



wKiom1Xs84Kz3tLNAAD0t06TroI238.gif

Java字节码的执行是由JVM执行引擎来完成，流程图如下所示：



wKiom1Xs85uxOF5mAAFJNP-fzhg385.gif

ava代码编译和执行的整个过程包含了以下三个重要的机制：

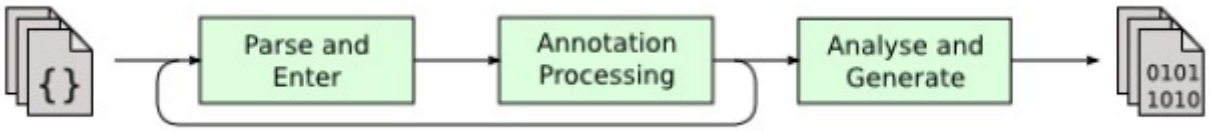
- Java源码编译机制
- 类加载机制
- 类执行机制

Java源码编译机制

Java 源码编译由以下三个过程组成：

- 分析和输入到符号表
- 注解处理
- 语义分析和生成class文件

流程图如下所示：



wKiom1Xs9FPAesSjAAAs7C8hkWs833.gif

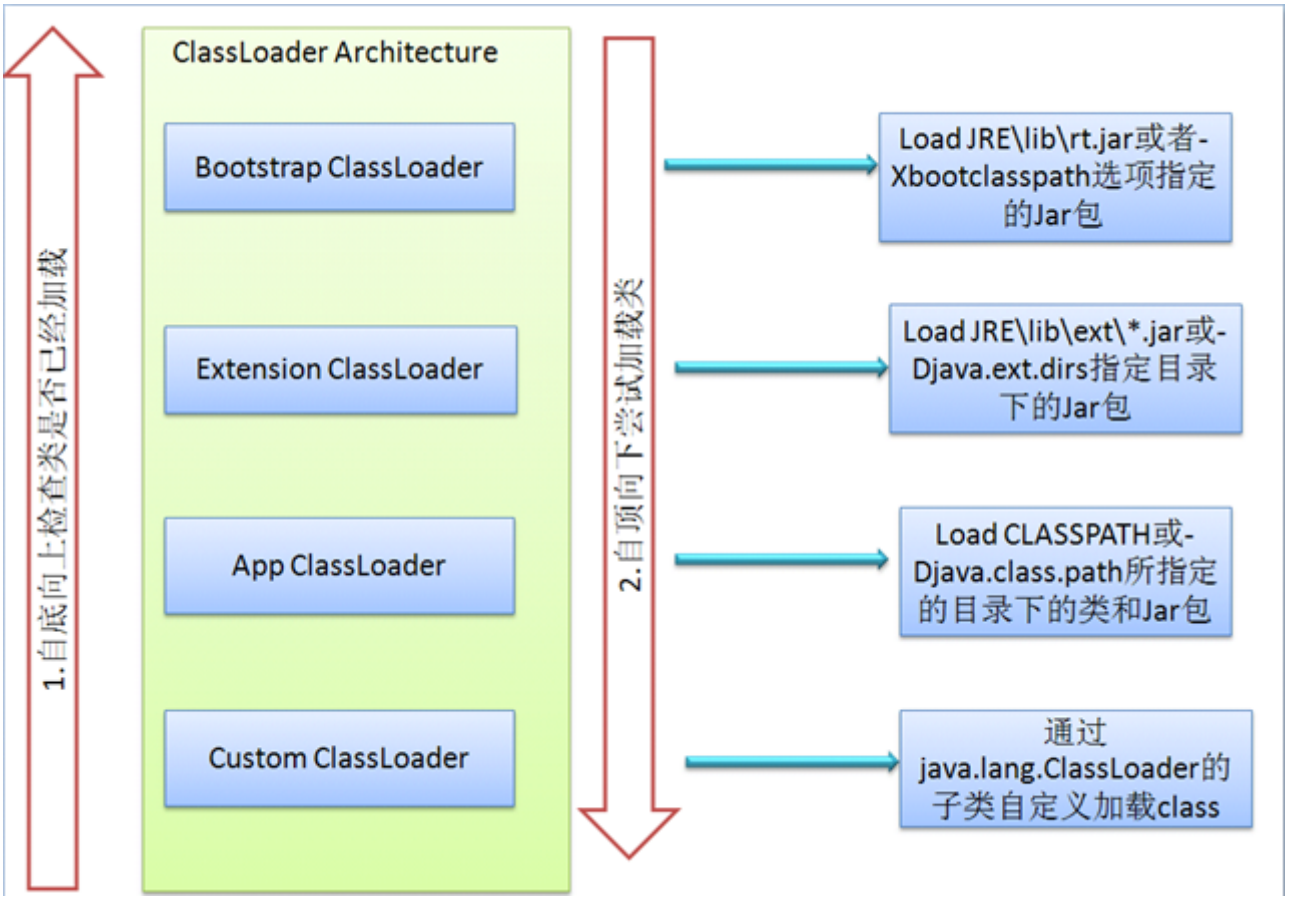
最后生成的class文件由以下部分组成：

- 结构信息。包括class文件格式版本号及各部分的数量与大小的信息
- 元数据。对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池
- 方法信息。对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号

信息

类加载机制

JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



wKiom1Xs9HazvE-rAALSsJvKx8U794.gif

1）Bootstrap ClassLoader

负责加载$JAVA_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

2）Extension ClassLoader

负责加载java平台中扩展功能的一些jar包，包括$JAVA_HOME中jre/lib/*.jar或-Djava.ext.dirs指定目录下的jar包

3）App ClassLoader

负责记载classpath中指定的jar包及目录中class

4）Custom ClassLoader

属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视

为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

类执行机制

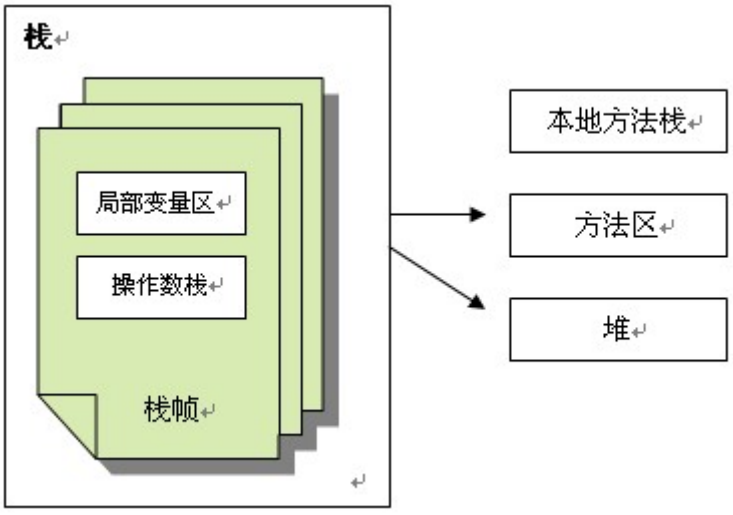
JVM是基于栈的体系结构来执行class字节码的。线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。栈的结构如下图所示：

wKiom1Xs9MHg3bYVAAAmhLAOiik099.gif

三、JVM内存管理和垃圾回收

JVM内存组成结构

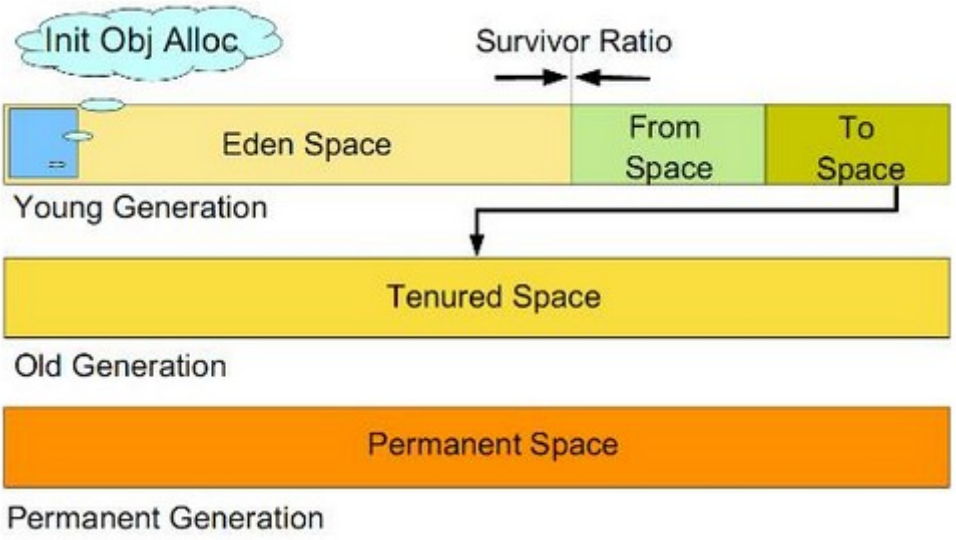
JVM栈由堆、栈、本地方法栈、方法区等部分组成，结构图如下所示：



wKiom1Xs9aWhN3OEAABJZEwuJ-c836.gif

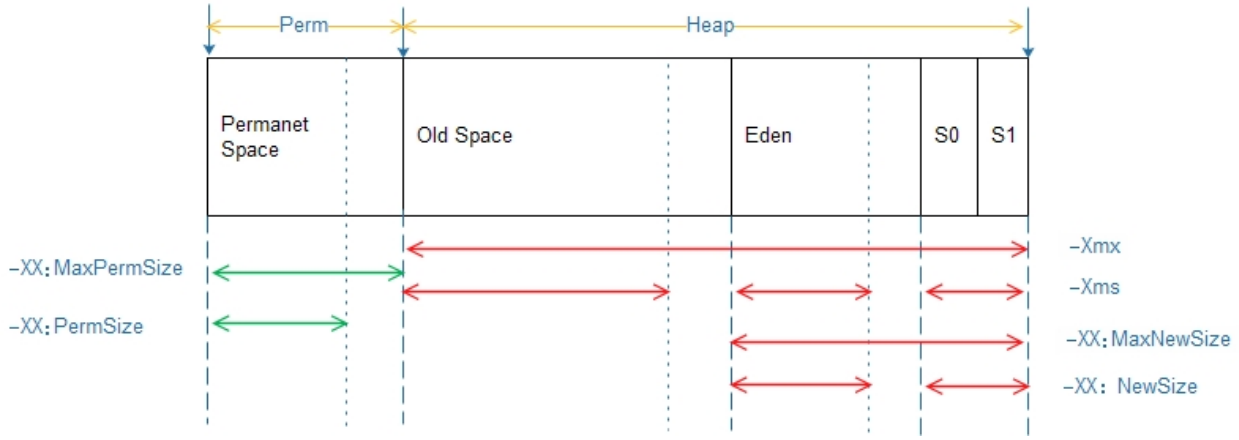
1）堆

所有通过new创建的对象的内存都在堆中分配，堆的大小可以通过-Xmx和-Xms来控制。堆被划分为新生代和旧生代，新生代又被进一步划分为Eden和Survivor区，最后Survivor由From Space和To Space组成，结构图如下所示：



wKiom1Xs9ceyZWP0AAB-_k0lkUo882.gif

- 新生代。新建的对象都是用新生代分配内存，Eden空间不足的时候，会把存活的对象转移到Survivor中，新生代大小可以由-Xmn来控制，也可以用-XX:SurvivorRatio来控制Eden和Survivor的比例
- 旧生代。用于存放新生代中经过多次垃圾回收仍然存活的对象
- 持久带（Permanent Space）实现方法区，主要存放所有已加载的类信息，方法信息，常量池等等。可通过-XX:PermSize和-XX:MaxPermSize来指定持久带初始化值和最大值。Permanent Space并不等同于方法区，只不过是Hotspot JVM用Permanent Space来实现方法区而已，有些虚拟机没有Permanent Space而用其他机制来实现方法区。



wKiom1Xs-fLx4MWrAAEPniOGR34183.jpg

- -Xmx:最大堆内存,如：-Xmx512m
- -Xms:初始时堆内存,如：-Xms256m
- -XX:MaxNewSize:最大年轻区内存
- -XX:NewSize:初始时年轻区内存.通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。实际可用空间为 = Eden + 1 个 Survivor，即 90%
- -XX:MaxPermSize:最大持久带内存
- -XX:PermSize:初始时持久带内存
- -XX:+PrintGCDetails。打印 GC 信息
- -XX:NewRatio 新生代与老年代的比例，如 -XX:NewRatio=2，则新生代占整个堆空间的1/3，老年代占2/3
- -XX:SurvivorRatio 新生代中 Eden 与 Survivor 的比值。默认值为 8。即 Eden 占新生代空间的 8/10，另外两个 Survivor 各占 1/10

2）栈

每个线程执行每个方法的时候都会在栈中申请一个栈帧，每个栈帧包括局部变量区和操作数栈，用于存放此次方法调用过程中的临时变量、参数和中间结果。

-xss:设置每个线程的堆栈大小. JDK1.5+ 每个线程堆栈大小为 1M，一般来说如果栈不是很深的话， 1M 是绝对够用了的。

3）本地方法栈

用于支持native方法的执行，存储了每个native方法调用的状态

4）方法区

存放了要加载的类信息、静态变量、final类型的常量、属性和方法信息。JVM用持久代（Permanet Generation）来存放方法区，可通过-XX:PermSize和-XX:MaxPermSize来指定最小值和最大值

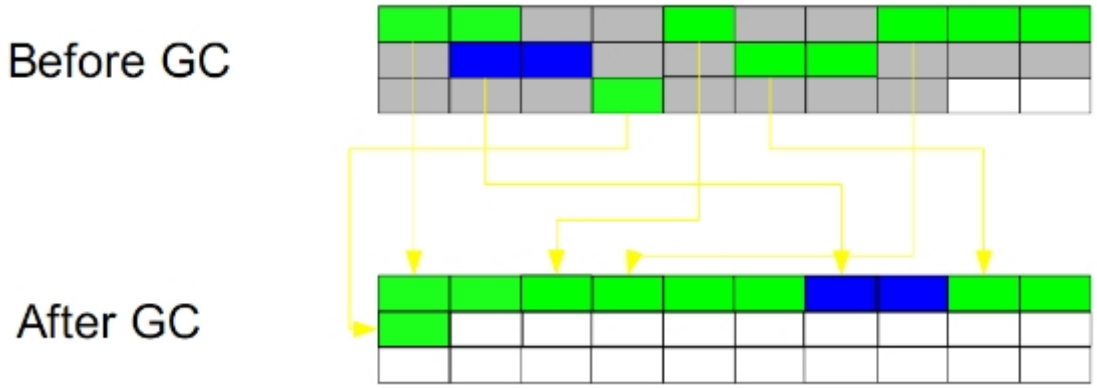
垃圾回收按照基本回收策略分

引用计数（Reference Counting）:

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为0的对象。此算法最致命的是无法处理循环引

用的问题。

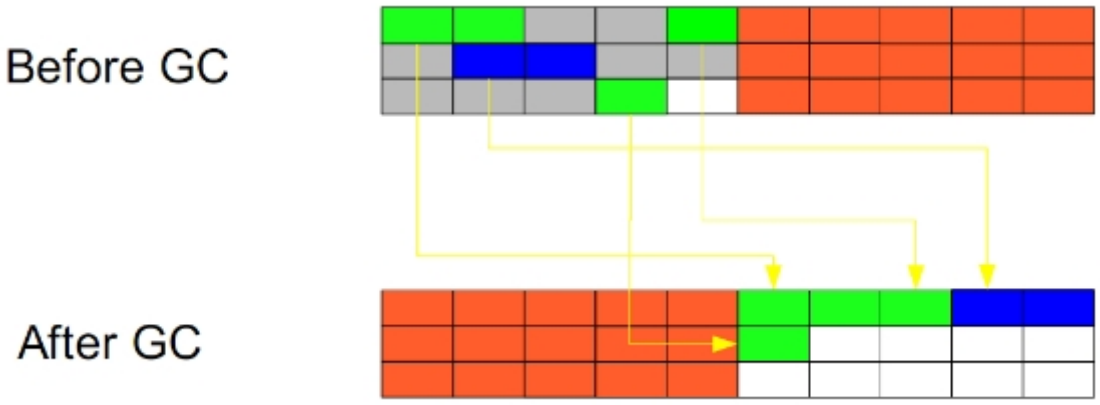
标记-清除（Mark-Sweep）：



wKiom1Xs_uqCNFJgAACkJWszH9Y986.jpg

此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

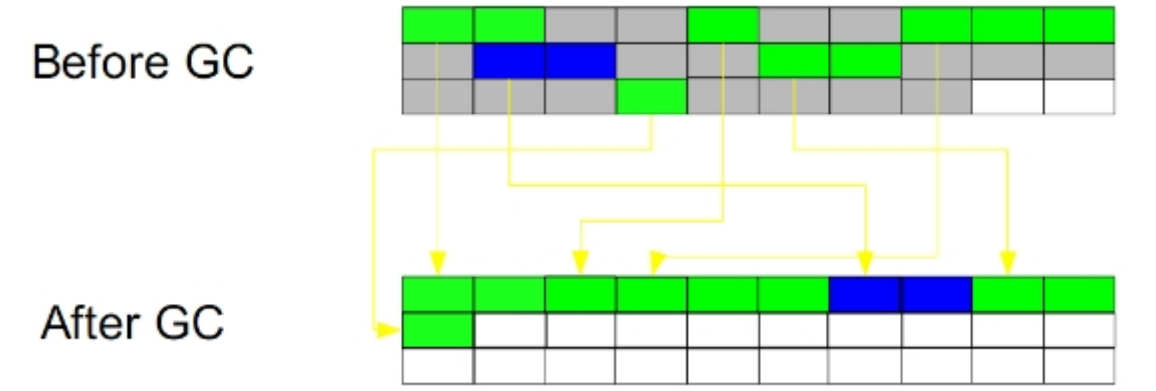
复制（Copying）：



wKioL1XtAUXT2TQ6AACmPxPQHwc598.jpg

此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

标记-整理（Mark-Compact）：



wKioL1XtAXfDO_2zAACkJWszH9Y289.jpg

此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

JVM分别对新生代和旧生代采用不同的垃圾回收机制

新生代的GC：

新生代通常存活时间较短，因此基于Copying算法来进行回收，所谓Copying算法就是扫描出存活的对象，并复制到一块新的完全未使用的空间中，对应于新生代，就是在Eden和From Space或To Space之间copy。新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在新生代区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。当连续分配对象时，对象会逐渐从eden到survivor，最后到旧生代。

在执行机制上JVM提供了串行GC（Serial GC）、并行回收GC（Parallel Scavenge）和并行GC（ParNew）

1）串行GC

在整个扫描和复制过程采用单线程的方式来进行，适用于单CPU、新生代空间较小及对暂停时间要求不是非常高的应用上，是client级别默认的GC方式，可以通过-XX:+UseSerialGC来强制指定

2）并行回收GC

在整个扫描和复制过程采用多线程的方式来进行，适用于多CPU、对暂停时间要求较短的应用上，是server级别默认采用的GC方式，可用-XX:+UseParallelGC来强制指定，用-XX:ParallelGCThreads=4来指定线程数

3）并行GC

与旧生代的并发GC配合使用

旧生代的GC：

旧生代与新生代不同，对象存活的时间比较长，比较稳定，因此采用标记（Mark）算法来进行回收，所谓标记就是扫描出存活的对象，然后再进行回收未被标记的对象，回收后对用空出的空间要么进行合并，要么标记出来便于下次进行分配，总之就是要减少内存碎片带来的效率损耗。在执行机制上JVM提供了串行GC（Serial MSC）、并行GC（parallel MSC）和并发GC（CMS），具体算法细节还有待进一步深入研究。

以上各种GC机制是需要组合使用的，指定方式由下表所示：

指定方式	新生代GC方式	旧生代GC方
-XX:+UseSerialGC	串行GC	串行GC
-XX:+UseParallelGC	并行回收GC	并行GC
-XX:+UseConeMarkSweepGC	并行GC	并发GC
-XX:+UseParNewGC	并行GC	串行GC
-XX:+UseParallelOldGC	并行回收GC	并行GC
-XX:+ UseConeMarkSweepGC -XX:+UseParNewGC	串行GC	并发GC
不支持的组合	1、-XX:+UseParNewGC -XX:+UseParallelOldGC 2、-XX:+UseParNewGC -XX:+UseSerialGC	

四、JVM内存调优

首先需要注意的是在对JVM内存调优的时候不能只看操作系统级别Java进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况，因为GC过后这个值是不会变化的，因此内存调优的时候要更多地使用JDK提供的内存查看工具，比如JConsole和Java VisualVM。

对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

旧生代空间不足

调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象

Pemanet Generation空间不足

增大Perm Gen空间，避免太多静态对象

统计得到的GC后晋升到旧生代的平均大小大于旧生代剩余空间

控制好新生代和旧生代的比例

System.gc()被显示调用

垃圾回收不要手动触发，尽量依靠JVM自身的机制

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，下面来看看各部分比例不良设置会导致什么后果

1）新生代设置过小

一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

2）新生代设置过大

一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加

一般说来新生代占整个堆1/3比较合适

3）Survivor设置过小

导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4）Survivor设置过大

导致eden过小，增加了GC频率

另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收

由内存管理和垃圾回收可知新生代和旧生代都有多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM提供两种较为简单的GC策略的设置方式

1）吞吐量优先

JVM以吞吐量为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，来达到吞吐量指标。这个值可由-XX:GCTimeRatio=n来设置

2）暂停时间优先

JVM以暂停时间为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，尽量保证每次GC造成的应用停止时间都在指定的数值范围内完成。这个值可由-XX:MaxGCPauseRatio=n来设置

最后汇总一下JVM常见配置

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。

我自己生产环境配置如下，tomcat是多实例的

```
CATALINA_OPTS="-Xms1024m -Xmx1024m -XX:NewRatio=4 -XX:PermSize=192m -XX:XX:SurvivorRatio=4"
```

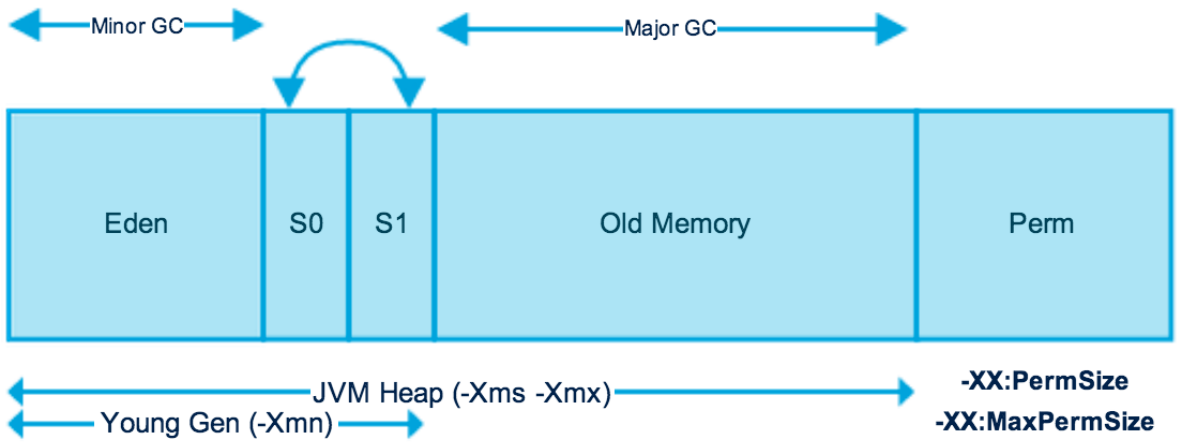
连接池

```
<Connector port="8080" protocol="HTTP/1.1"
               maxHttpHeaderSize="8192"
               maxThreads="1000"
               minSpareThreads="100"
               maxSpareThreads="1000"
               enableLookups="false"
               connectionTimeout="10000"
               URIEncoding="utf-8"
               acceptCount="1000"
               redirectPort="8443"
               disableUploadTimeout="true"/>
```

调优续：

要了解Java垃圾收集机制，先理解JVM内存模式是非常重要的。今天我们将会了解JVM内存的各个部分、如何监控以及垃圾收集调优。

Java（JVM）内存模型



正如你从上面的图片看到的，JVM内存被分成多个独立的部分。广泛地说，JVM堆内存被分为两部分——年轻代（Young Generation）和老年代（Old Generation）。

年轻代

年轻代是所有新对象产生的地方。当年轻代内存空间被用完时，就会触发垃圾回收。这个垃圾回收叫做Minor GC。年轻代被分为3个部分——Enden区和两个Survivor区。

年轻代空间的要点：

- 大多数新建的对象都位于Eden区。
- 当Eden区被对象填满时，就会执行Minor GC。并把所有存活下来的对象转移到其中一个survivor区。
- Minor GC同样会检查存活下来的对象，并把它们转移到另一个survivor区。这样在一段时间内，总会有一个空的survivor区。
- 经过多次GC周期后，仍然存活下来的对象会被转移到年老代内存空间。通常这是在年轻代有资格提升到年老代前通过设定年龄阈值来完成的。

年老代

年老代内存里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。通常会在老年代内存被占满时进行垃圾回收。老年代的垃圾收集叫做Major GC。Major GC会花费更多的时间。

Stop the World事件

所有的垃圾收集都是“Stop the World”事件，因为所有的应用线程都会停下来直到操作完成（所以叫“Stop the World”）。

因为年轻代里的对象都是一些临时（short-lived ）对象，执行Minor GC非常快，所以应用不会受到（“Stop the World”）影响。

由于Major GC会检查所有存活的对象，因此会花费更长的时间。应该尽量减少Major GC。因为Major GC会在垃圾回收期间让你的应用反应迟钝，所以如果你有一个需要快速响应的应用发生多次Major GC，你会看到超时错误。

垃圾回收时间取决于垃圾回收策略。这就是为什么有必要去监控垃圾收集和对垃圾收集进行调优。从而避免要求快速响应的应用出现超时错误。

永久代

永久代或者“Perm Gen”包含了JVM需要的应用元数据，这些元数据描述了在应用里使用的类和方法。注意，永久代不是Java堆内存的一部分。

永久代存放JVM运行时使用的类。永久代同样包含了Java SE库的类和方法。永久代的对象在full GC时进行垃圾收集。

方法区

方法区是永久代空间的一部分，并用来存储类型信息（运行时常量和静态变量）和方法代码和构造函数代码。

内存池

如果JVM实现支持，JVM内存管理会为创建内存池，用来为不变对象创建对象池。字符串池就是内存池类型的一个很好的例子。内存池可以属于堆或者永久代，这取决于JVM内存管理的实现。

运行时常量池

运行时常量池是每个类常量池的运行时代表。它包含了类的运行时常量和静态方法。运行时常量池是方法区的一部分。

Java栈内存

Java栈内存用于运行线程。它们包含了方法里的临时数据、堆里其它对象引用的特定数据。你可以阅读栈内存和堆内存的区别。

Java 堆内存开关

Java提供了大量的内存开关（参数），我们可以用它来设置内存大小和它们的比例。下面是一些常用的开关：

VM 开关	VM 开关描述
-Xms	设置JVM启动时堆的初始化大小。
-Xmx	设置堆最大值。
-Xmn	设置年轻代的空间大小，剩下的为老年代的空间大小。
-XX:PermGen	设置永久代内存的初始化大小。
-XX:MaxPermGen	设置永久代的最大值。
-XX:SurvivorRatio	提供Eden区和survivor区的空间比例。比如，如果年轻代的大小为10m并且VM开关是-XX:SurvivorRatio=2，那么将会保留5m内存给Eden区和每个Survivor区分配2.5m内存。默认比例是8。
-XX:NewRatio	提供年老代和年轻代的比例大小。默认值是2。

大多数时候，上面的选项已经足够使用了。但是如果你还想了解其他的选项，那么请查看JVM选项官方网页。

Java垃圾回收

Java垃圾回收会找出没用的对象，把它从内存中移除并释放出内存给以后创建的对象使用。Java程序语言中的一个最大优点是自动垃圾回收，不像其他的程序语言那样需要手动分配和释放内存，比如C语言。

垃圾收集器是一个后台运行程序。它管理着内存中的所有对象并找出没被引用的对象。所有的这些未引用的对象都会被删除，回收它们的空间并分配给其他对象。

一个基本的垃圾回收过程涉及三个步骤：

1. 标记：这是第一步。在这一步，垃圾收集器会找出哪些对象正在使用和哪些对象不在使用。
2. 正常清除：垃圾收集器清会除不在使用的对象，回收它们的空间分配给其他对象。
3. 压缩清除：为了提升性能，压缩清除会在删除没用的对象后，把所有存活的对象移到一起。这样可以提高分配新对象的效率。

简单标记和清除方法存在两个问题：

1. 效率很低。因为大多数新建对象都会成为“没用对象”。
2. 经过多次垃圾回收周期的对象很有可能在以后的周期也会存活下来。

上面简单清除方法的问题在于Java垃圾收集的分代回收的，而且在堆内存里有年轻代和年老代两个区域。我已经在上面解释了Minor GC和Major GC是怎样扫描对象，以及如何把对象从一个分代空间移到另外一个分代空间。

Java垃圾回收类型

这里有五种可以在应用里使用的垃圾回收类型。仅需要使用JVM开关就可以在我们的应用里启用垃圾回收策略。让我们一起来逐一了解：

1. Serial GC（-XX:+UseSerialGC）：Serial GC使用简单的标记、清除、压缩方法对年轻代和年老代进行垃圾回收，即Minor GC和Major GC。Serial GC在client模式（客户端模式）很有用，比如在简单的独立应用和CPU配置较低的机器。这个模式对占有内存较少的应用很管用。
2. Parallel GC（-XX:+UseParallelGC）：除了会产生N个线程来进行年轻代的垃圾收集外，Parallel GC和Serial GC几乎一样。这里的N是系统CPU的核数。我们可以使用 -XX:ParallelGCThreads=n 这个JVM选项来控制线程数量。并行垃圾收集器也叫throughput收集器。因为它使用了多CPU加快垃圾回收性能。Parallel GC在进行年老代垃圾收集时使用单线程。
3. Parallel Old GC（-XX:+UseParallelOldGC）：和Parallel GC一样。不同之处，Parallel Old GC在年轻代垃圾收集和年老代垃圾回收时都使用多线程收集。
4. 并发标记清除（CMS）收集器（-XX:+UseConcMarkSweepGC)：CMS收集器也被称为短暂停顿并发收集器。它是对年老代进行垃圾收集的。CMS收集器通过多线程并发进行垃圾回收，尽量减少垃圾收集造成的停顿。CMS收集器对年轻代进行垃圾回收使用的算法和Parallel收集器一样。这个垃圾收集器适用于不能忍受长时间停

顿要求快速响应的应用。可使用 -XX:ParallelCMSThreads=n JVM选项来限制CMS收集器的线程数量。

5. G1垃圾收集器（-XX:+UseG1GC) G1（Garbage First）：垃圾收集器是在Java 7后才可以使用的特性，它的长远目标时代替CMS收集器。G1收集器是一个并行的、并发的和增量式压缩短暂停顿的垃圾收集器。G1收集器和其他的收集器运行方式不一样，不区分年轻代和年老代空间。它把堆空间划分为多个大小相等的区域。当进行垃圾收集时，它会优先收集存活对象较少的区域，因此叫“Garbage First”。你可以在Oracle Garbage-FIrst收集器文档找到更多详细信息。

Java垃圾收集监控

我们可以使用命令行和图形工具来监控监控应用垃圾回收。例如，我使用Java SE下载页中的一个demo来实验。

如果你想使用同样的应用，可以到Java SE下载页面下载JDK 7和JavaFX演示和示例。我使用的示例应用是Java2Demo.jar，它位于 jdk1.7.0_55/demo/jfc/Java2D 目录下。这只是一个可选步骤，你可以运行GC监控命令监控任何Java应用。

我打开演示应用使用的命令是：

```
1  pankaj@Pankaj:~/Downloads/jdk1.7.0_55/demo/jfc/Java2D$ java -Xmx120m -Xm
   Java2Demo.jar
```

jsat

可以使用jstat命令行工具监控JVM内存和垃圾回收。标准的JDK已经附带了jstat，所以不需要做任何额外的事情就可以得到它。

要运行jstat你需要知道应用的进程id，你可以使用 ps -eaf | grep java 命令获取进程id。

```
1  pankaj@Pankaj:~$ ps
   -eaf | grep
   Java2Demo.jar
2
   501 9582 11579 0 9:48PM ttys000 0:21.66 /usr/bin/java
   -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseG1
3
   501 14073 14045 0 9:48PM ttys002 0:00.00 grep
   Java2Demo.jar
```

从上面知道，我的Java应用进程id是9582。现在可以运行jstat命令了，就像下面展示的一样：

```
1  pankaj@Pankaj:~$ jstat -gc 9582 1000
2
   S0C S1C S0U S1U EC EU OC OU PC PU YGC YGCT FGC FGCT GCT
3
   1024.0 1024.0 0.0 0.0 8192.0 7933.3 42108.0 23401.3 20480.0 19990.9 157
   1.381 1.654
4
```

```
5          1024.0 1024.0 0.0 0.0 8192.0 8026.5 42108.0 23401.3 20480.0 19990.9 157 1.381 1.654
6          1024.0 1024.0 0.0 0.0 8192.0 8030.0 42108.0 23401.3 20480.0 19990.9 157 1.381 1.654
7          1024.0 1024.0 0.0 0.0 8192.0 8122.2 42108.0 23401.3 20480.0 19990.9 157 1.381 1.654
8          1024.0 1024.0 0.0 0.0 8192.0 8171.2 42108.0 23401.3 20480.0 19990.9 157 1.381 1.654
9          1024.0 1024.0 48.7 0.0 8192.0 106.7 42108.0 23401.3 20480.0 19990.9 158 1.381 1.656
           1024.0 1024.0 48.7 0.0 8192.0 145.8 42108.0 23401.3 20480.0 19990.9 158 1.381 1.656
```

jstat命令的最后一个参数是每个输出的时间间隔。每隔一秒就会打印出内存和垃圾收集数据。

让我们一起来对每一列的意义进行逐一了解：

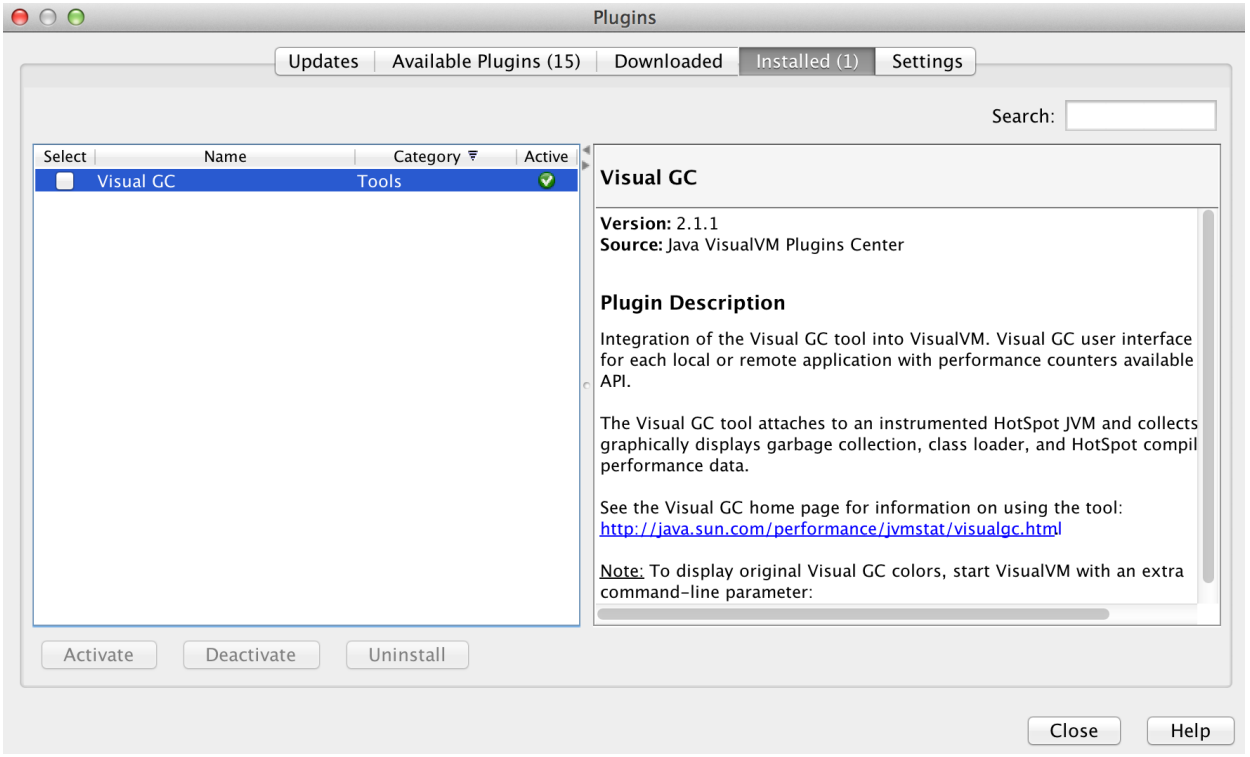
- S0C和S1C：这一列展示了Survivor0和Survivor1区的当前大小（单位KB）。
- S0U和S1U：这一列展示了当前Survivor0和Survivor1区的使用情况（单位KB）。注意：无论任何时候，总会有一个Survivor区是空着的。
- EC和EU：这些列展示了Eden区当前空间大小和使用情况（单位KB）。注意：EU的大小一直在增大。而且只要大小接近EC时，就会触发Minor GC并且EU将会减小。
- OC和OU：这些列展示了年老代当前空间大小和当前使用情况（单位KB）。
- PC和PU：这些列展示了Perm Gen（永久代）当前空间大小和当前使用情况（单位KB）。
- YGC和YGCT：YGC这列显示了发生在年轻代的GC事件的数量。YGCT这列显示了在年轻代进行GC操作的累计时间。注意：在EU的值由于minor GC导致下降时，同一行的YGC和YGCT都会增加。
- FGC和FGCT：FGC列显示了发生Full GC事件的次数。FGCT显示了进行Full GC操作的累计时间。注意：相对于年轻代的GC使用时间，Full GC所用的时间长很多。
- GCT：这一列显示了GC操作的总累计时间。注意：总累计时间是YGCT和FGCT两列所用时间的总和（GCT=YGCT+FGCT）。

jstat的优点，我们同样可以在没有GUI的远程服务器上运行jstat。注意：我们是通过 -Xmn10m 选项来指定S0C、S1C和EC的总和为10m的。

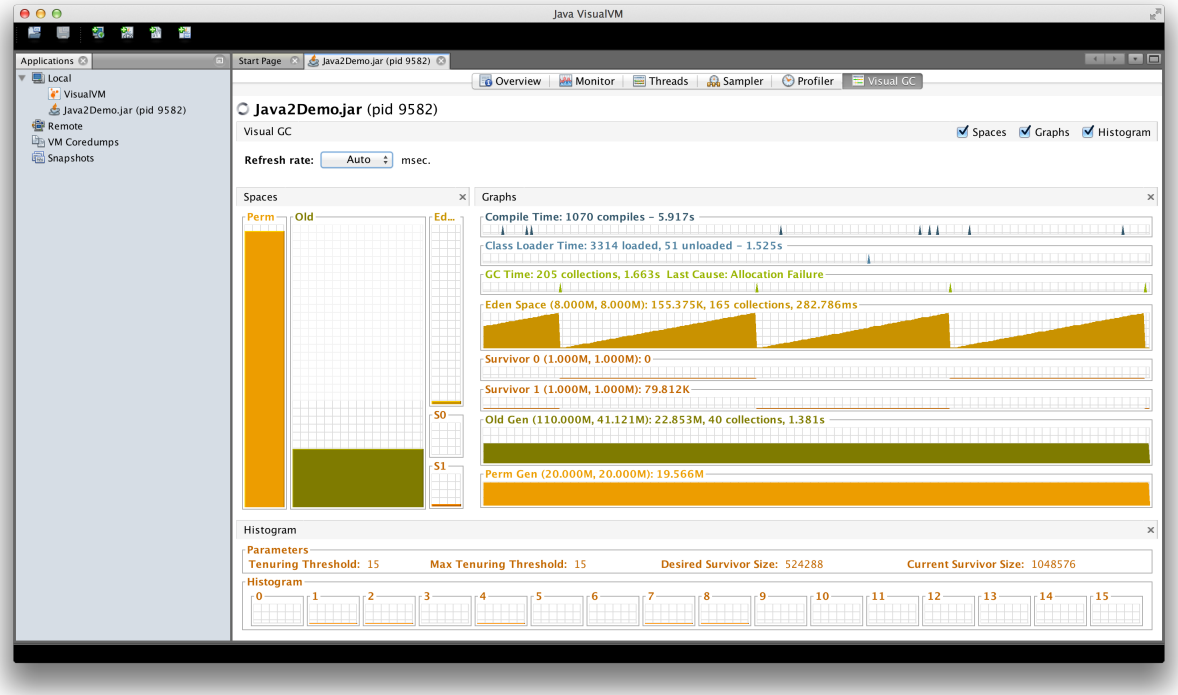
Java VisualVM及Visual GC插件

如果你想在GUI里查看内存和GC，那么可以使用jvisualvm工具。Java VisualVM同样是JDK的一部分，所以你不需要单独去下载。

在终端运行jvisualvm命令启动Java VisualVM程序。一旦启动程序，你需要从Tools->Plugins选项安装Visual GC插件，就像下面图片展示的。



安装完Visual GC插件后，从左边栏打开应用并把视角转到Visual GC部分。你将会得到关于JVM内存和垃圾收集详情，如下图所示。



Java垃圾回收调优

Java垃圾回收调优应该是提升应用吞吐量的最后一个选择。在你发现应用由于长时间垃圾回收导致了应用性能下降、出现超时的时候，应该考虑Java垃圾收集调优。

如果你在日志里看到 java.lang.OutOfMemoryError: PermGen space错误，那么可以尝试使用 -XX:PermGen 和 -XX:MaxPermGen JVM选项去监控并增加Perm Gen内存空间。你也可以尝试使用-XX:+CMSClassUnloadingEnabled并查看使用CMS垃圾收集器的执行性能。

如果你看到了大量的Full GC操作，那么你应该尝试增大老年代的内存空间。

全面垃圾收集调优要花费大量的努力和时间，这里没有一尘不变的硬性调优规则。你需要去尝试不同的选项并且对这些选项进行对比，从而找出最适合自己应用的方案。
这就是所有的Java内存模型和垃圾回收内容。希望对你理解JVM内存和垃圾收集过程有所帮助。