# JVM研究

# 什么是JVM

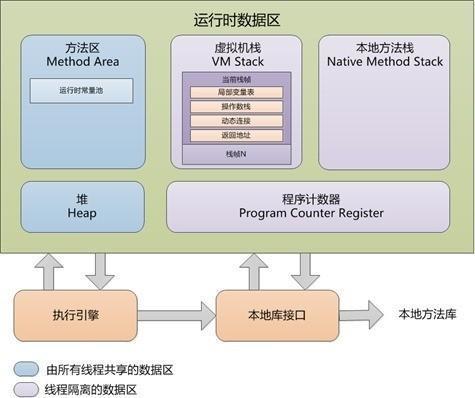
VM是Java Virtual Mechine的缩写。它是一种基于计算设备的规范，是一台虚拟机，即虚构的计算机。

JVM屏蔽了具体操作系统平台的信息（显然，就像是我们在电脑上开了个虚拟机一样），当然，JVM执行字节码时实际上还是要解释成具体操作平台的机器指令的。

通过JVM，Java实现了平台无关性，Java语言在不同平台运行时不需要重新编译，只需要在该平台上部署JVM就可以了。因而能实现一次编译多处运行。

**下文中的内容都是基于Sun HotSpot虚拟机。**

# JVM运行时数据区



JVM运行时数据区

## 程序计数器

程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

Java虚拟机的多线程就是通过线程轮流切换并分配处理器执行时间的方式来实现的，任何一个确定的时刻，一个处理器只会执行一条线程中的指令。所以为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器。

每个线程都有自己单独的程序计数器，互不影响这个属于“线程私有”的内存。如果线程正在执行的是一个Java方法，这个程序计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是**Native方法**，这个计数器值则**为空**。此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

## Java虚拟机栈

Java虚拟机栈也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型，每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。当线程调用Java方法时，虚拟机压入一个新的栈帧到对应线程的虚拟机栈中；当方法返回时，这个栈帧就被从栈中弹出并抛弃。

局部变量表（编译期可知的各种基本数据类型、引用类型和指向一条字节码指令的returnAddress类型）、操作数栈、动态链接、方法出口等信息。 

## 本地方法栈

本地方法栈与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

## Java堆

Java堆（Java Heap）是被所有线程共享的一块区域，所有的对象实例以及数组都要在堆上分配。  
 Java堆是垃圾收集器管理的主要区域。从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以Java堆中还可以细分为：新生代和老年代；新生代再细致一点的有Eden空间、From Survivor空间、To Survivor空间等。  
 如果堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

## 方法区（JDK1.7）

方法区是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

Sun HotSpot虚拟机采用永久代（Permanent Generation）来实现方法区。  
 这个区域的内存回收目标主要是针对常量池和对类型的卸载。当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

注意:方法区在JDK1.8中被废弃，改为Metaspace。

### 运行时常量池

运行时常量池是**方法区**的一部分。CLass文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息的常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

## 元空间（Metaspace , JDK1.8）

JDK 8.HotSpot JVM开始使用本地化的内存存放类的元数据，这个空间叫做元空间（Metaspace）。　其实，移除永久代的工作从JDK1.7就开始了。JDK1.7中，存储在永久代的部分数据就已经转移到了Java Heap或者是 Native Heap。但永久代仍存在于JDK1.7中，并没完全移除，譬如符号引用(Symbols)转移到了native heap；字面量(interned strings)转移到了java heap；类的静态变量(class statics)转移到了java heap。PermSize 和 MaxPermGen 两个参数已经失效，jvm启动时会告警提示。相应增加了-XX:MetaspaceSize和-XX:MaxMetaspaceSize来指定元空间的大小。

## 直接内存

直接内存（DirectMemory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError异常出现。JDK1.4加的NIO中，ByteBuffer有个方法是allocateDirect(intcapacity) ，这是一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括RAM及SWAP区或者分页文件）的大小及处理器寻址空间的限制。服务器管理员配置虚拟机参数时，一般会根据实际内存设置-Xmx等参数信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制（包括物理上的和操作系统级的限制），从而导致动态扩展时出现OutOfMemoryError异常。

# 垃圾回收（GC）

Java 语言的一大特点就是可以进行自动垃圾回收处理，而无需开发人员过于关注系统资源，例如内存资源的释放情况。垃圾回收中，程序计数器和虚拟机栈，本地方法栈3个内存区域是线程私有的，其中栈的内存分配在类结构确定的时候已经确定，所以在垃圾回收过程中不需要考虑。关注堆和方法区。Java 8 移除了方法区，改为了元数据区。

## Java的4种引用

1．强引用

2．软引用（SoftReference）

如果一个对象只具有软引用，那就类似于可有可无生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

3．弱引用（WeakReference）

如果一个对象只具有弱引用，那就类似于可有可物的生活用品。 弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

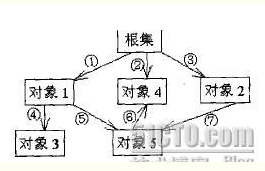
弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4．虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。虚引用的get()方法永远返回 null

**接下来介绍一下，在垃圾回收中，对象的可达性判断：**



在垃圾回收中，一般是对某些引用（比如软、弱、虚引用）或者不可达对象进行内存回收，在判断一个对象是否可达的时候，先从根集（Object对象）开始查找，如果没有路径到达某个对象，那么这个对象就是“不可达”对象，会在垃圾回收的时候被回收。如果有路径可达的情况下，那么就需要判断这个“可达”的强弱程度。

如上图所示，在这个树形的引用链中，箭头的方向代表了引用的方向，所指向的对象是被引用对象。由图可以看出，从根集到一个对象可以由很多条路径。比如到达对象5的路径就有①-⑤，③-⑦两条路径。由此带来了一个问题，那就是某个对象的可达性如何判断:

◆单条引用路径可达性判断:在这条路径中，最弱的一个引用决定对象的可达性

◆多条引用路径可达性判断:几条路径中，最强的一条的引用决定对象的可达性

比如，我们假设图中引用①和③为强引用，⑤为软引用，⑦为弱引用，对于对象5按照这两个判断原则，路径①-⑤取最弱的引用⑤，因此该路径对对象5的引用为软引用。同样，③-⑦为弱引用。在这两条路径之间取最强的引用，于是对象5是一个软可达对象。

## JAVA对象存活判定

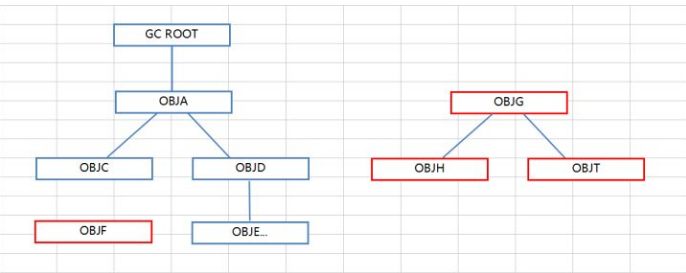
要判断对象是否存活，主要有两种算法：引用计数法和可达性分析算法

### 引用计数法

引用计数法就是给对象加上一个引用计数器，每当对象被 引用一次计数器值就加1，引用时效则减1，计数器为0则表示不会再被使用。但是此算法很难解决对象间相互循环引用的问题。

### 可达性分析算法

主流JVM中判定对象是否存活的算法为可达性分析。

从GC Roots到对象没有任何的引用链存在，则对象不可达。

可以作为RC Roots的对象为：

1.虚拟机栈（栈帧中的本地变量表）中引用的对象。

2.方法区中的类静态变量引用的对象

3.方法区中的常量引用的对象。

4。本地方法栈中JNI（Java Native Interface）引用的方法。

## 生存还是死亡

1. 如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize()方法。当对象没有覆盖 finalize() 方法，或者 finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。如果这个对象被判定为有必要执行 finalize() 方法，那么这个对象将会放置在一个叫做 F-Queue 的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，这样做的原因是，如果一个对象在 finalize() 方法中执行缓慢，或者发生了死循环（更极端的情况），将很可能会导致 F-Queue 队列中其他对象永久处于等待，甚至导致整个内存回收系统崩溃。

2.finalize()方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize() 中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this 关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱, 那基本上它就真的被回收了。  
3.不鼓励使用这种方法来拯救对象。相反，建议尽量避免使用它。它的运行代价高昂，不确定性大，无法保证各个对象的调用顺序。finalize() 能做的所有工作，使用 try-finally 或者其他方式都可以做得更好、更及时。

## 垃圾收集算法（标记的方法）

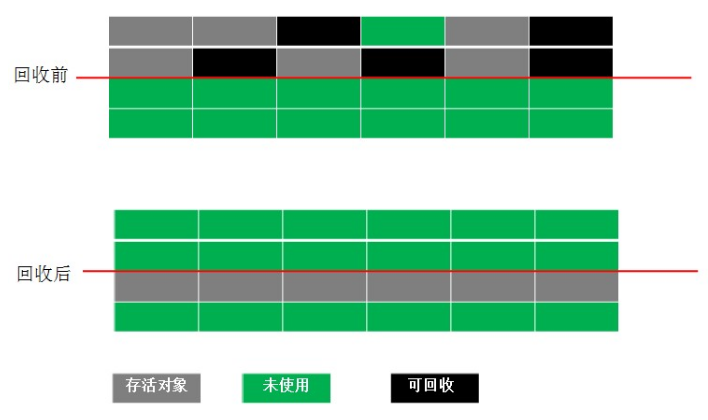
### 1.“标记-清除”（Mark-Sweep）算法



和它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

它的主要缺点有两个：一个是效率问题，标记和清除过程的效率都不高；另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。会导致内存足够却无法分配对象的问题。

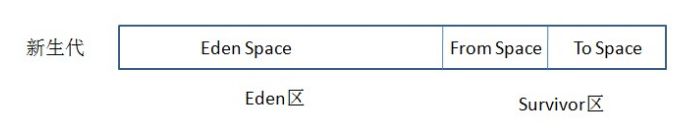
### 2.复制算法



将内存分为（大小相等）两部分，每次只使用其中一块进行内存分配，当内存使用完后，就出发GC，将存活的对象直接复制到另一块空闲的内存中，然后对当前使用的内存块一次性清除所有，然后转到另一块内存进行使用。

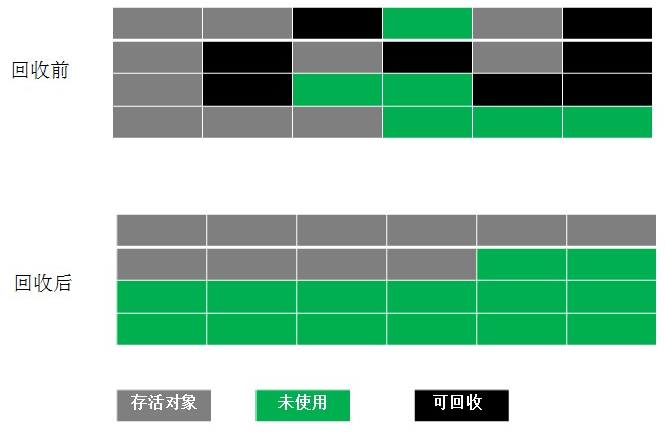
优点：简单，高效。适用于新生代。

缺点：浪费内存，因为每次都有另一块内存空闲着。



现在Hotspot中分为1个Eden和2个Survivor空间。大小比为8：1。使用参数-XX:SurvivorRatio来设置大小比。

### 3. “标记-整理”（Mark-Compact）算法



标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

### 分代收集算法

GC分代的基本假设：绝大部分对象的生命周期都非常短暂，存活时间短。

“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

### 回收方法区

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类

废弃常量：假如一个字符串abc已经进入了常量池中，如果当前系统没有任何一个String对象abc，也就是没有任何Stirng对象引用常量池的abc常量，也没有其他地方引用的这个字面量，这个时候发生内存回收这个常量就会被清理出常量池

无用的类：

1.该类所有的实例都已经被回收，就是Java堆中不存在该类的任何实例

2.加载该类的ClassLoader已经被回收

3.该类对用的java.lang.Class对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法

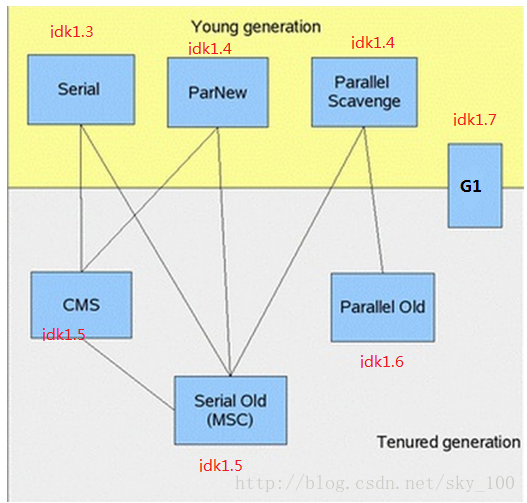
## 垃圾收集器

使用

windows： java -XX:+PrintFlagsFinal -version | findstr :

Linux ：java -XX:+PrintFlagsFinal -version | grep :

查看使用的垃圾收集器



新生代：

1.Serial：

只使用一个线程进行垃圾回收。使用复制算法。产生STW。

参数控制： -XX:+UseSerialGC

适用于Client模式。

2.ParNew:

Serial的多线程版本。（复制算法）

使用与Server模式。

参数控制：-XX:+UserParNewGC

      -XX:ParallelGCThread 限制线程数量

3.Parallel Scavenge：

关注吞吐量。也被成为"吞吐量优先"收集器（Throughput）

只能和Parallel Old或者Serial Old配合使用。无法和CMS配合。

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)

参数控制： -XX:+UseParallelGC

老年代：

1.Serial Old：

Serial的老年代版本。单线程。主要配合PS使用。还有作为CMS的后备方案，在并发收集出现Concurrent Mode Failure时使用。

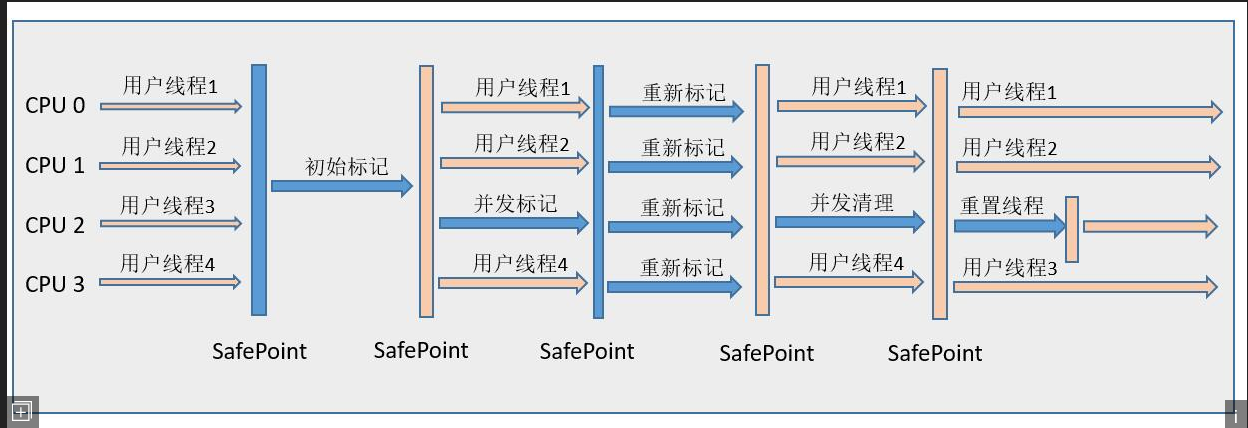
2.Parallel Old：

PS的老年代版本。配合PS使用。主要用于CPU资源敏感和注重吞吐量的场景。

3.CMS

ConcurrentMarkSweep

关注收集时的停顿时间。



目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其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。

从名字（包含“Mark Sweep”）上就可以看出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来说要更复杂一些。

整个过程分为4个步骤，包括：

    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标记阶段稍长一些，但远比并发标记的时间短。

由于

缺点：产生大量空间碎片、并发阶段会降低吞吐量（多线程并发收集，CPU资源占用多）。

          产生浮动垃圾。因为并发清理阶段用户线程仍在执行。因此需要预留老年代空间给线程使用。所以CMS无法等老年代被填满再收集。否则会出现Concurrent Mode Failure导致另一次Full GC。

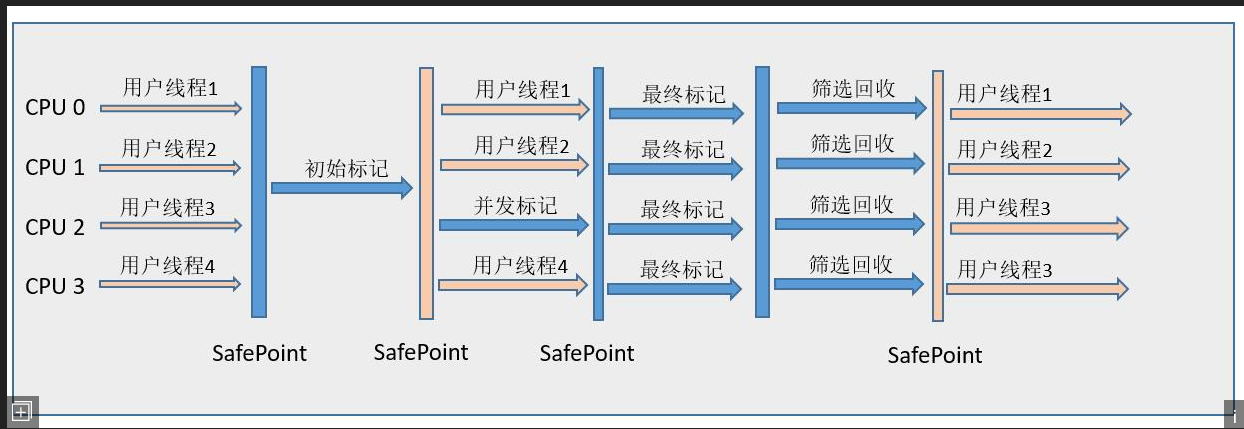
参数控制：-XX:+UseConcMarkSweepGC  使用CMS收集器

          -XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长

          -XX:+CMSFullGCsBeforeCompaction  设置进行几次不压缩的Full GC后，进行一次带碎片整理的GC。

          -XX:ParallelCMSThreads  设定CMS的线程数量（一般情况为可用(CPU数量+3)/4）

G1收集器作用与新生代和年老代



## GC日志

# JVM类加载机制

JVM把描述类数据的字节码.Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的java类型，这就是虚拟机的类加载机制。

（注意：下面所说的初始化是指类初始化，而类初始化完成后实例对象也要初始化，这里2个概念是不同的）

Java类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using) 和 卸载(Unloading)七个阶段。其中准备、验证、解析3个部分统称为连接（Linking），如图所示：



　　加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）。以下陈述的内容都以HotSpot为基准。特别需要注意的是，类的加载过程必须按照这种顺序按部就班地“开始”，而不是按部就班的“进行”或“完成”，因为这些阶段通常都是相互交叉地混合式进行的，也就是说通常会在一个阶段执行的过程中调用或激活另外一个阶段。

## 类初始化时机

那么，什么情况下虚拟机需要开始初始化一个类呢？这在虚拟机规范中是有严格规定的，虚拟机规范指明 有且只有 五种情况必须立即对类进行初始化（而这一过程自然发生在加载、验证、准备之后）：

1) 遇到new、getstatic、putstatic或invokestatic这四条字节码指令（注意，newarray指令触发的只是数组类型本身的初始化，而不会导致其相关类型的初始化，比如，new String[]只会直接触发String[]类的初始化，也就是触发对类[Ljava.lang.String的初始化，而直接不会触发String类的初始化）时，如果类没有进行过初始化，则需要先对其进行初始化。生成这四条指令的最常见的Java代码场景是：

使用new关键字实例化对象的时候；

读取或设置一个类的静态字段（被final修饰，已在编译器把结果放入常量池的静态字段除外）的时候；

调用一个类的静态方法的时候。

　2) 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

　3) 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

　4) 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

　5) 当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getstatic,REF\_putstatic,REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

## 类加载器

JVM设计者把类加载阶段中的“通过'类全名'来获取定义此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。

1.类与类加载器

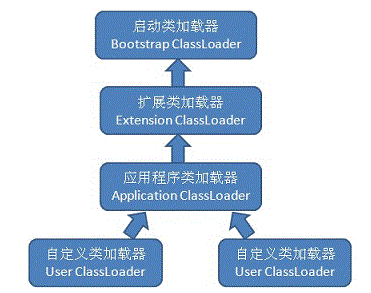
对于任何一个类，都需要由加载它的类加载器和这个类来确立其在JVM中的唯一性。也就是说，两个类来源于同一个Class文件，并且被同一个类加载器加载，这两个类才相等。

2.双亲委派模型

从虚拟机的角度来说，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），该类加载器使用C++语言实现，属于虚拟机自身的一部分。另外一种就是所有其它的类加载器，这些类加载器是由Java语言实现，独立于JVM外部，并且全部继承自抽象类java.lang.ClassLoader。

从Java开发人员的角度来看，大部分Java程序一般会使用到以下三种系统提供的类加载器：  
1)启动类加载器（Bootstrap ClassLoader）：负责加载JAVA\_HOME\lib目录中并且能被虚拟机识别的类库到JVM内存中，如果名称不符合的类库即使放在lib目录中也不会被加载。该类加载器无法被Java程序直接引用。  
2)扩展类加载器（Extension ClassLoader）：该加载器主要是负责加载JAVA\_HOME\lib\ext，该加载器可以被开发者直接使用。  
3)应用程序类加载器（Application ClassLoader）：该类加载器也称为系统类加载器，它负责加载用户类路径（Classpath）上所指定的类库，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

我们的应用程序都是由这三类加载器互相配合进行加载的，我们也可以加入自己定义的类加载器。这些类加载器之间的关系如下图所示：



如上图所示的类加载器之间的这种层次关系，就称为类加载器的双亲委派模型（Parent Delegation Model）。该模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。  
双亲委派模型的工作过程为：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的加载器都是如此，因此所有的类加载请求都会传给顶层的启动类加载器，只有当父加载器反馈自己无法完成该加载请求（该加载器的搜索范围中没有找到对应的类）时，子加载器才会尝试自己去加载。  
使用这种模型来组织类加载器之间的关系的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如java.lang.Object类，无论哪个类加载器去加载该类，最终都是由启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。否则的话，如果不使用该模型的话，如果用户自定义一个java.lang.Object类且存放在classpath中，那么系统中将会出现多个Object类，应用程序也会变得很混乱。如果我们自定义一个rt.jar中已有类的同名Java类，会发现JVM可以正常编译，但该类永远无法被加载运行。

在rt.jar包中的java.lang.ClassLoader类中，我们可以查看类加载实现过程的代码，具体源码如下：

protected synchronized Class loadClass(String name, boolean resolve)

throws ClassNotFoundException {

// 首先检查该name指定的class是否有被加载

Class c = findLoadedClass(name);

if (c == null) {

try {

if (parent != null) {

// 如果parent不为null，则调用parent的loadClass进行加载

c = parent.loadClass(name, false);

} else {

// parent为null，则调用BootstrapClassLoader进行加载

c = findBootstrapClass0(name);

}

} catch (ClassNotFoundException e) {

// 如果仍然无法加载成功，则调用自身的findClass进行加载

c = findClass(name);

}

}

if (resolve) {

resolveClass(c);

}

return c;

}

通过上面代码可以看出，双亲委派模型是通过loadClass()方法来实现的，根据代码以及代码中的注释可以很清楚地了解整个过程其实非常简单：先检查是否已经被加载过，如果没有则调用父加载器的loadClass()方法，如果父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载器加载失败，则先抛出ClassNotFoundException，然后再调用自己的findClass()方法进行加载。

## 破坏双亲委派模型

1. 第一次破坏是在jdk2之前，用户自定义的类加载器都是重写Classloader中的loadClass方法,这样就导致每个自定义的类加载器其实是在使用自己的loadClass方法中的加载机制来进行加载,这种模式当然是不符合双亲委派机制的，也是无法保证同一个类在jvm中的唯一性的，那么为了保证及时是由不同的类加载器(哪怕是用户自定义的类加载器加载)也是唯一的，java官方在Classloader中添加了findClass方法,用户只需要重新这个findClass方法，在loadClass方法的逻辑里，如果父类加载失败的时候，才会调用自己的findClass方法来完成类加载，这样就完成了符合双亲委派机制。
2. 第二次的破坏是类似于jndi，jdbc这种服务，因为这种服务需要回调用户的代码，但是对于父类加载器而言是不认识用户的代码的。那么这时候java团队使用了一个不太优雅的设计：线程上下文类加载器。这个类加载器可以通过Thread类的setContextClassLoader方法进行设置,如果创建线程时还未设置，它就从父线程继承一个，如果在应用全局范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器。利用这个线程上下文类加载器,jdni去加载需要的spi代码，也就是父类请求子类的加载器去加载。
3. 第三次的破坏是因为用户对于程序的动态性追求，诸如：代码热替换，模块热部署。  
   这时候就诞生了诸如jigsaw和osgi。对于现在的业界来讲，osgi赢得了java模块化的主导权，成为目前业界模块化的标准。而Osgi模块话的关键是他自己的类加载机制：每个程序模块(bundle)都有自己的类加载器，需要更换程序(bundle)的时候，连同类加载器一起替换，以实现代码的热部署。

# 常用JVM监控工具

在JDK目录下的bin目录里有许多JDK提供的JVM监控工具。

## JPS：虚拟机进程状况工具

JDK的很多小工具的名字都参考了UNIX命令的命名方式，jps（JVM Process Status）是其中的典型。除了名字像UNIX的ps命令外，它的功能也和ps命令类似：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一ID（Local Virtual Machine Identifier,LVMID）。虽然功能比较单一，但它是使用最高的JDK命令行工具，因为其他的JDK工具大多需要输入它查询到的LVMID来确定要监控的是哪一个虚拟机进程。

**jps命令格式：**

jps [ options ] [ hostid ]

**jps工具主要选项**

|  |  |
| --- | --- |
| **选    项** | **作            用** |
| -q | 只输出LVMID，省略主类的名称 |
| -m | 输出虚拟机进程启动时传递给主类main()函数的参数 |
| -l | 输出主类的全名，如果进程执行的是jar包，输出jar包路径 |
| -v | 输出虚拟机进程启动时的JVM参数 |

示例：



## Jstat：虚拟机统计信息监控工具 jstat（JVM Statistics Monitoring Tool）使用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程（需要远程主机提供RMI支持）虚拟机进程中的类信息、内存、垃圾收集、JIT编译等运行数据，在没有GUI，只提供了纯文本控制台环境的服务器上，它将是运行期间定位虚拟机性能问题的首选工具。

**jstat命令格式**

jstat [ option vmid [ interval [ s | ms ] [ count ] ] ]

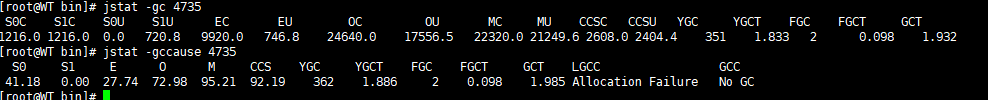
这个VMID，对于本地虚拟机进程而言，VMID和LVMID是一致的。参数interval和count分别表示查询间隔和次数，如果省略这两个参数，说明只查询一次，假设需要每250毫秒查询一次进程2764的垃圾收集情况，一共查询20次，那命令应当是：

**jstat -gc 2764 250 20**

**jstat主要工具选项**

|  |  |
| --- | --- |
| **选       项** | **作              用** |
| -class | 监视类装载、卸载数量、总空间以及类装载所耗费的时间 |
| -gc | 监视Java堆状况，包括Eden区、两个Survivor区、、老年代、永久带等的容量、已用空间、GC时间合计等信息 |
| -gccapacity | 监视内容基本与-gc相同，但输出主要关注Java堆各个区域使用到的最大、最小空间 |
| -gcutil | 监视内容基本与-gc相同，但输出主要关注已使用的空间占总空间的百分比 |
| -gccause | 与-gcutil功能一样，但是会额外输出导致上一次GC产生的原因 |
| -gcnew | 监视新生代GC状况 |
| -gcnewcapacity | 监视内容基本与-gcnew相同，但输出主要关注使用到的最大、最小空间 |
| -gcold | 监视老年代GC状况 |
| -gcoldcapacity | 监视内容基本与-gcold相同，但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编译器编译过的方法、耗时等信息 |
| -printcompilation | 输出已经被JIT编译的方法 |

示例：



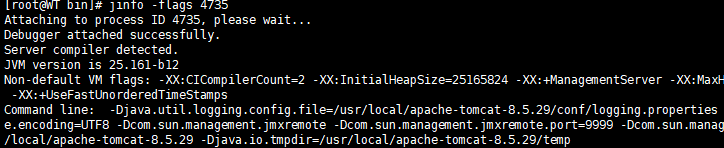
## jinfo：Java配置信息工具

jinfo（Configuration Info for Java）的作用是实时地查看和调整虚拟机各项参数。使用jps命令的-v可以查看虚拟机启动时显式指定的参数列表，但如果想知道未被显式指定的参数的系统默认值，可以使用jinfo的-flag选项进行查询，jinfo还可以使用-sysprops选项把虚拟机进程的System.getProperties()的内容打印出来。

**jinfo命令格式**

jinfo [ option ] pid

示例：



## jmap：Java内存映像工具

jmap（Memory Map for Java）命令用于生成堆转储快照。如果不使用jmap命令，要想获取Java堆转储，可以使用“-XX:+HeapDumpOnOutOfMemoryError”参数，可以让虚拟机在OOM异常出现之后自动生成dump文件，Linux命令下可以通过kill -3发送进程退出信号也能拿到dump文件。

jmap的作用并不仅仅是为了获取dump文件，它还可以查询finalize执行队列、Java堆和永久代的详细信息，如空间使用率、当前使用的是哪种收集器等。和jinfo一样，jmap有不少功能在Windows平台下也是受限制的，除了生成dump文件的-dump选项和用于查看每个类的实例、空间占用统计的-histo选项在所有操作系统都提供之外，其余选项都只能在Linux和Solaris系统下使用。

**jmap命令格式**

jmap [ option ] vmid

**jmap工具主要选项**

|  |  |
| --- | --- |
| **选             项** | **作          用** |
| -dump | 生成Java堆转储快照。格式为-dump:[live, ]format=b,file=<filename>，其中live自参数说明是否只dump出存活的对象 |
| -finalizerinfo | 显示在F-Queue中等待Finalizer线程执行finalize方法的对象。只在Linux和Solaris系统下有效 |
| -heap | 显示Java堆详细信息，如使用哪种收集器、参数配置、分代状况等。只在Linux和Solaris系统下有效 |
| -histo | 显示堆中对象统计信息，包括类、实例数量、合计容量 |
| -permstat | 以ClassLoader为统计口径显示永久代内存状态。只在Linux和Solaris系统下有效 |
| -F | 当虚拟机进行对-dump选项没有响应时，可使用这个选项强制生成dump快照。只在Linux和Solaris系统下有效 |

示例：



## jstack：Java堆栈跟踪工具

jstack（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的主要是定位线程长时间出现停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的原因。线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者在等待些什么资源。

**jstack命令格式**

jstack [ option ] vmid

**jstack主要工具选项**

|  |  |
| --- | --- |
| **选    项** | **作             用** |
| -F | 当正常输出的请求不被响应时，强制输出线程堆栈 |
| -l | 除堆栈外，显示关于锁的附加信息 |
| -m | 如果调用到本地方法的时候，可以显示C/C++的堆栈 |

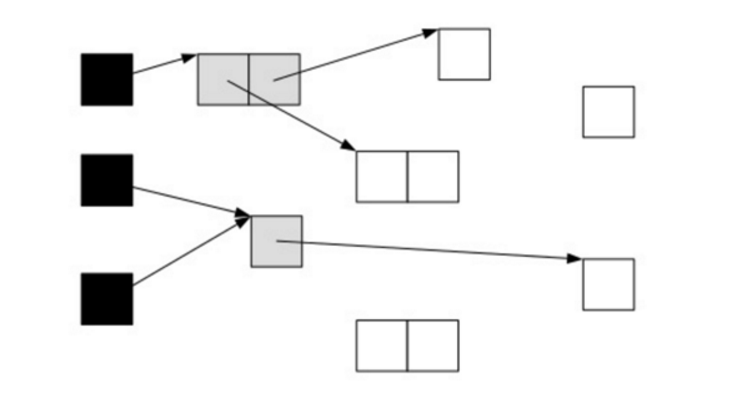
# 标记算法

## 三色标记

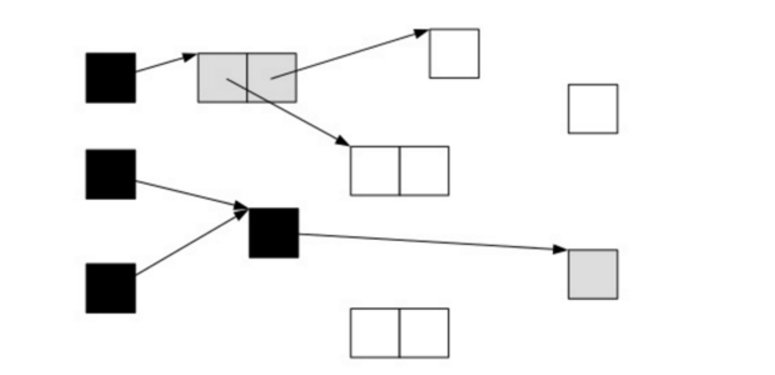
它是描述追踪式回收器的一种有用的方法，利用它可以推演回收器的正确性。 首先，我们将对象分成三种类型的。

* 黑色:根对象，或者该对象与它的子对象都被扫描
* 灰色:对象本身被扫描,但还没扫描完该对象中的子对象
* 白色:未被扫描对象，扫描完成所有对象之后，最终为白色的为不可达对象，即垃圾对象

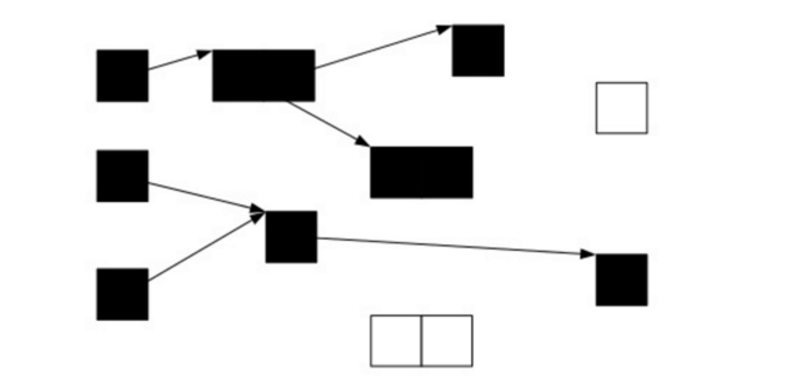
当GC开始扫描对象时，按照如下图步骤进行对象的扫描：

根对象被置为黑色，子对象被置为灰色。

继续由灰色遍历,将已扫描了子对象的对象置为黑色。



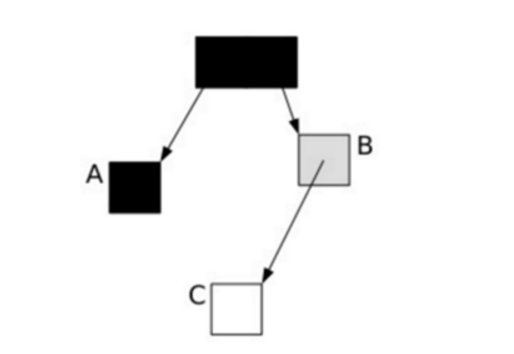
遍历了所有可达的对象后，所有可达的对象都变成了黑色。不可达的对象即为白色，需要被清理。



## 保证回收对象的正确性

在标记过程中，应用程序也在运行，那么对象的指针就有可能改变。这样的话，我们就会遇到一个问题：对象丢失问题。

我们看下面一种情况，当垃圾收集器扫描到下面情况时：

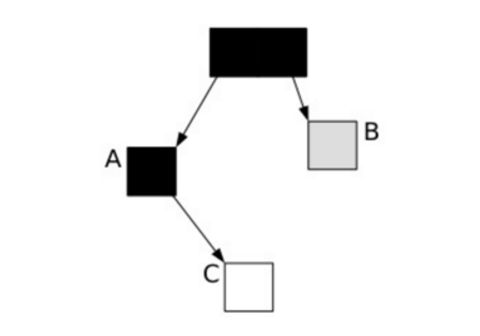


这时候应用程序执行了以下操作：

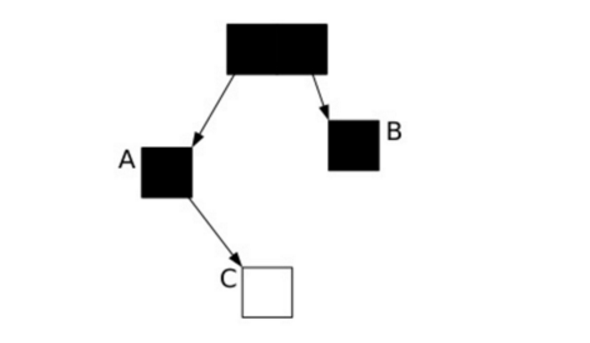
A.c=C

B.c=null

这样，对象的状态图变成如下情形：



这时候垃圾收集器再标记扫描的时候就会下图成这样：



很显然，此时C是白色，被认为是垃圾需要清理掉，显然这是不合理的。那么我们如何保证应用程序在运行的时候，GC标记的对象不丢失呢？有如下两种可行的方式：

1. 在插入的时候记录对象（CMS）
   1. 采用增量更新（Incremental update），每次对象变更会插入写屏障（强制将信息刷入主存）
   2. 在写屏障中如果发现有白色对象被赋值到黑色对象里面，则标记为灰色
2. 在删除的时候记录对象（G1）
   1. 采用SATB（snapshot-at-the-beginning），开始标记时候生成快照，记录所有存活对象
   2. 并发标记的时候，记录所有变更的对象，然后在写屏障中把所有旧的引用所指向的对象都变成非白的（旧的引用指开始快照里面的旧对象）

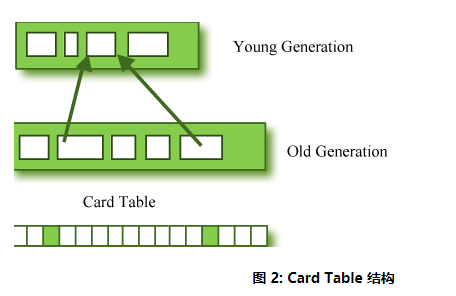
# RememberSet

如果老年代引用年轻代，年轻代如何回收？扫描整个老年代？

1. 在老年代中有一块区域用来记录指向新生代的引用（RememberSet，底层使用CardTable实现）
2. 当年轻代回收时，直接扫描RS而不用扫描所有的老年代，如果年轻代有对象被引用，则不删除

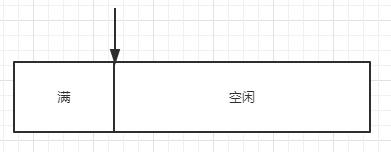
卡表是个单字节数组，每一位只能放0/1，且每位表示老年代一个4K的空间，当该位为1，表示这个4K空间有引用年轻代对象（会记录引用的年轻代的物理地址），为0，则没有。

在写卡表的时候，先设置写屏障，将各个线程中最新的数据刷新到主存，（获取最新的变更信息）然后检查赋值的引用对象是否在年轻代，如果是，则卡表该位置为1，否则为0.



# TLAB

## 什么是指针碰撞（bump-the-pointer）



基于连续的物理内存来分配新空间，所以每次要创建新对象时，只需要判断eden剩余的空间是否满足新对象的大小，大大加快了内存分配速度。

## 什么是TLAB（Thread-Local Allocation Buffers）

JVM在内存新生代Eden Space中开辟了一小块线程私有的区域，称作TLAB（Thread-local allocation buffer）。默认设定为占用Eden Space的1%。在Java程序中很多对象都是小对象且用过即丢，它们不存在线程共享也适合被快速GC，所以对于小对象通常JVM会优先分配在TLAB上，并且TLAB上的分配由于是线程私有所以没有锁开销。因此在实践中分配多个小对象的效率通常比分配一个大对象的效率要高。

也就是说，Java中每个线程都会有自己的缓冲区称作TLAB（Thread-local allocation buffer），每个TLAB都只有一个线程可以操作，TLAB结合bump-the-pointer技术可以实现快速的对象分配，而不需要任何的锁进行同步，也就是说，在对象分配的时候不用锁住整个堆，而只需要在自己的缓冲区分配即可。

## 对象分配过程

1. 对象在TLAB中进行分配，如果空间不足则进入2
2. 在Eden区加锁（堆所有线程共享，不加锁会导致多个线程同时读取到当前地址为空，然后开始分配对象），然后分配对象
3. 如果Eden区放不下，则进行Young GC
4. Young GC后还放不下就放到老年代
5. 如果老年代放不下，会进行Old GC
6. 如果GC后还放不下，堆内存溢出

备注：尽量避免创建大对象，默认大对象会直接分配到老年代，这样会导致老年代过早执行GC，也容易内存溢出

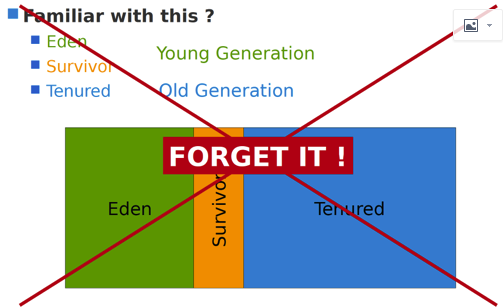
# G1垃圾回收器

首先，G1的设计原则就是简单可行的性能调优

开发人员仅仅需要声明以下参数即可：

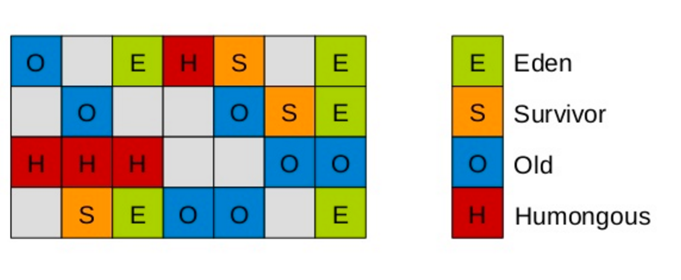
1. -XX:+UseG1GC -Xmx32g -XX:MaxGCPauseMillis=200
   1. 其中-XX:+UseG1GC为开启G1垃圾收集器，-Xmx32g 设计堆内存的最大内存为32G，
   2. -XX:MaxGCPauseMillis=200设置GC的最大暂停时间为200ms。如果我们需要调优，在内存大小一定的情况下，我们只需要修改最大暂停时间即可。

其次，G1将新生代，老年代的物理空间划分取消了。



## G1对jvm的内存划分

G1算法将堆划分为若干个区域（Region），它仍然属于分代收集器。不过，这些区域的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或者Survivor空间。老年代也分成很多区域，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有cms内存碎片问题的存在了。

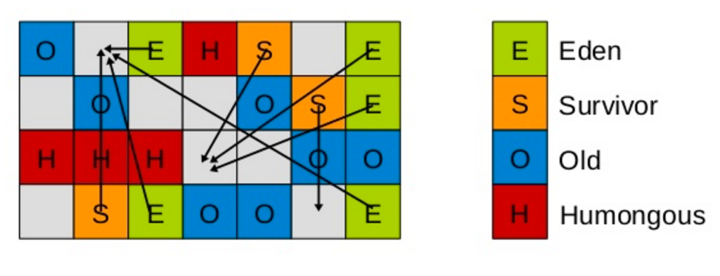


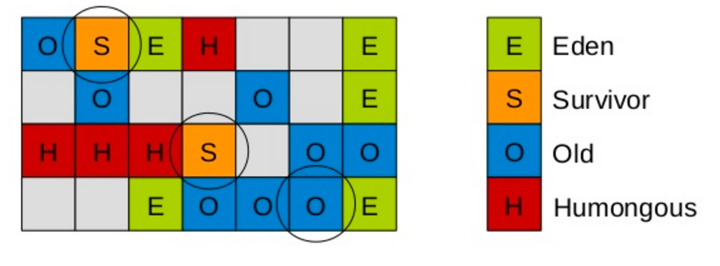
分区描述：

1. Eden：伊甸区
2. Survivor：存活区，Eden在GC后，把存活对象放入该区
3. Old：年老区，若干次GC后，还存活的对象移入该区
4. Humongous：存放巨型对象（对象容量大于一个region的50%）
   1. 如果一个H放不下，则jvm会找几个连续的H区来存放

## G1的年轻代回收

1. Eden区满后，存活对象会移入Survivor区（初始状态，所有Survivor都为空）
2. Eden区再次满后，Eden+Survivor区都会GC，然后存活对象放入新的Survivor区
3. 上面第二步如果Survivor区放不下，则一部分对象会加入到Old区





## G1的MIX回收