# 引言

Java 编程语言是一种通用的、并发的、面向对象的语言。它的语法类似于 C 和 C ++，但它忽略了许多 C 和 C++ 复杂性、混乱和不安全的语言特性 Java 平台最初是为解决构建联网消费电子设备的软件问题。它被设计为支持多主机架构，并允许软件组件安全交付。为了满足这些要求，编译出来的本地代码必须解决不同网络间传输问题、可以操作各式各样的客户端，并且还要代码在这些客户端上能安全正确地运行。

Java和c++

[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "/home/rui/文档\\x/_blank)、C++都属于高级语言，而计算机能认识执行的只是机器码（即二进制），所以高级语言都必须经过直接或间接的转换成汇编以后，才能运行：

对于C/C++这类高级计算机语言，它们的编译器（例如Unix下的CC命令、Windows下的CL命令）都是可以把源码直接编译成计算机可以认识的机器码，如exe、dll之类的文件，然后直接运行

而Java语言的跨平台性是它最大的特点之一，这就达成了Java程序平台无头性，因此也有一定的牺牲就是多了一个中间过程，先将Java源程序编译成class文件【字节码形式】存储，然后由JVM（Java虚拟机--屏蔽了[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "/home/rui/文档\\x/_blank)层面，直接自己提供虚拟的硬件）加载，然后在执行时再把字节码解释成具体平台上的机器指令

两者各有优缺点，因此通常我们说C/C++语言效率较高，而Java语言平台无关性

****1.什****么是jvm?

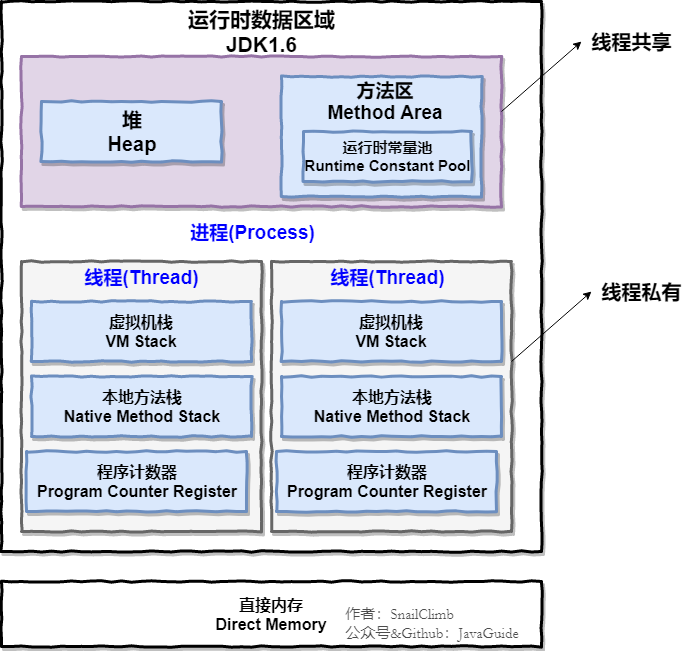
1）jvm是一种用于计算设备的规范，它是一个虚构出来的机器，是通过在实际的计算机上仿真模拟各种功能实现的。  
（2）jvm包含一套字节码指令集，一组寄存器，一个栈，一个垃圾回收堆和一个存储方法域。  
（3）JVM屏蔽了与具体操作系统平台相关的信息，使Java程序只需生成在Java虚拟机上运行的目标代码（字节码）,就可以在多种平台上不加修改地运行。  
JVM在执行字节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行。

# 2,**Java 内存区域详解**

对于 Java 程序员来说，在虚拟机自动内存管理机制下，不再需要像 C/C++程序开发程序员这样为内一个 new 操作去写对应的 delete/free 操作，不容易出现内存泄漏和内存溢出问题。正是因为 Java 程序员把内存控制权利交给 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会是一个非常艰巨的任务。

## **运行时数据区域**

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。JDK. 1.8 和之前的版本略有不同，下面会介绍到。



****线程私有的：****

* 程序计数器
* 虚拟机栈
* 本地方法栈

****线程共享的：****

* 堆
* 方法区
* 直接内存 (非运行时数据区的一部分)

### **程序计数器**

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。****字节码解释器工作时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、线程恢复等功能都需要依赖这个计数器来完。****

另外，****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，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。****

****从上面的介绍中我们知道程序计数器主要有两个作用：****

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

****注意：程序计数器是唯一一个不会出现 OutOfMemoryError 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。****

### **Java 虚拟机栈**

****与程序计数器一样，Java 虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型，每次方法调用的数据都是通过栈传递的。****

****Java 内存可以粗糙的区分为堆内存（Heap）和栈内存 (Stack),其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分。**** （实际上，Java 虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。）

****局部变量表主要存放了编译器可知的各种数据类型****（boolean、byte、char、short、int、float、long、double）、****对象引用****（reference 类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

****Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。****

* ****StackOverFlowError：**** 若 Java 虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 StackOverFlowError 异常。
* ****OutOfMemoryError：**** 若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出 OutOfMemoryError 异常。

Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

### **本地方法栈**

和虚拟机栈所发挥的作用非常相似，区别是： ****虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。**** 在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 StackOverFlowError 和 OutOfMemoryError 两种异常。

### **堆**

Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。****此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。****

Java 堆是垃圾收集器管理的主要区域，因此也被称作****GC 堆（Garbage Collected Heap）****.从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代：再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。****进一步划分的目的是更好地回收内存，或者更快地分配内存。****

### **方法区**

方法区与 Java 堆一样，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。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 ****Non-Heap（非堆）****，目的应该是与 Java 堆区分开来。

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、接口等描述信息外，还有**常量池信息（用于存放编译期生成的各种字面量和符号引用）**

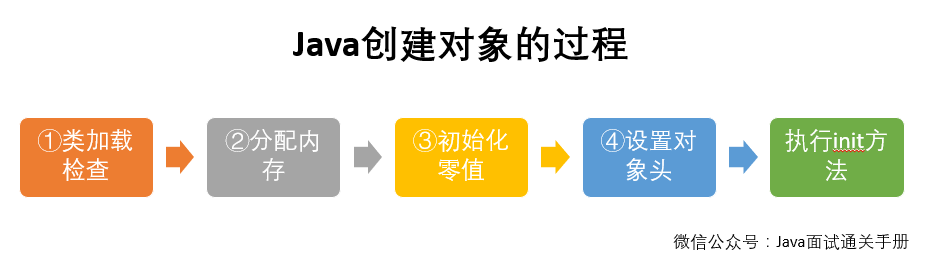
既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

## **HotSpot 虚拟机对象探秘**

通过上面的介绍我们大概知道了虚拟机的内存情况，下面我们来详细的了解一下 HotSpot 虚拟机在 Java 堆中对象分配、布局和访问的全过程。

### **3.1 对象的创建**

下图便是 Java 对象的创建过程，我建议最好是能默写出来，并且要掌握每一步在做什么。



#### **Step1:类加载检查**

虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

#### **Step2:分配内存**

在****类加载检查****通过后，接下来虚拟机将为新生对象****分配内存****。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。****分配方式****有 ****“指针碰撞”**** 和 ****“空闲列表”**** 两种，****选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定****。

#### **Step3:初始化零值**

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

#### **Step4:设置对象头**

初始化零值完成之后，****虚拟机要对对象进行必要的设置****，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、对象的 GC 分代年龄等信息。 ****这些信息存放在对象头中。**** 另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

#### **Step5:执行 init 方法**

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚开始，<init> 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 <init> 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

### **对象的访问定位**

建立对象就是为了使用对象，我们的 Java 程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式有虚拟机实现而定，目前主流的访问方式有****①使用句柄****和****②直接指针****两种：

****句柄：**** 如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；

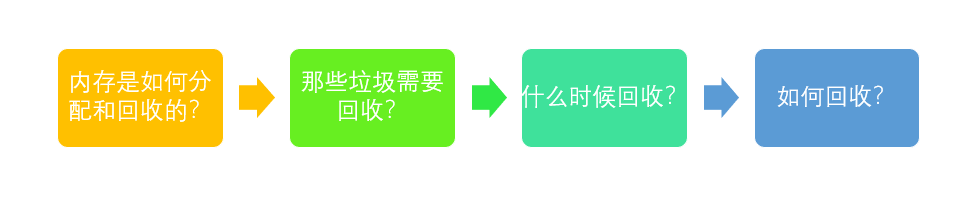
****String 对象的两种创建方式：****

String str1 = "abcd";//先检查字符串常量池中有没有"abcd"，如果字符串常量池中没有，则创建一个，然后 str1 指向字符串常量池中的对象，如果有，则直接将 str1 指向"abcd""；

String str2 = new String("abcd");//堆中创建一个新的对象

# **JVM 垃圾回收**

* 如何判断对象是否死亡（两种方法）。
* 简单的介绍一下强引用、软引用、弱引用、虚引用（虚引用与软引用和弱引用的区别、使用软引用能带来的好处）。
* 如何判断一个常量是废弃常量
* 如何判断一个类是无用的类
* 垃圾收集有哪些算法，各自的特点？
* HotSpot 为什么要分为新生代和老年代？
* 常见的垃圾回收器有那些？
* 介绍一下 CMS,G1 收集器。
* Minor Gc 和 Full GC 有什么不同呢？



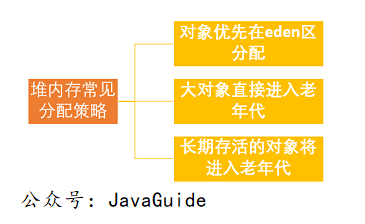
Java 的自动内存管理主要是针对对象内存的回收和对象内存的分配。同时，Java 自动内存管理最核心的功能是 ****堆**** 内存中对象的分配与回收。

Java 堆是垃圾收集器管理的主要区域，因此也被称作****GC 堆（Garbage Collected Heap）****.从垃圾回收的角度，由于现在收集器基本都采用**分代垃圾收集算法**，所以 Java 堆还可以细分为：新生代和老年代：再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。****进一步划分的目的是更好地回收内存，或者更快地分配内存。****



上图所示的 eden 区、s0("From") 区、s1("To") 区都属于新生代，tentired 区属于老年代。

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s1("To")，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。



* ****新生代 GC（Minor GC）****:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
* ****老年代 GC（Major GC/Full GC）****:指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

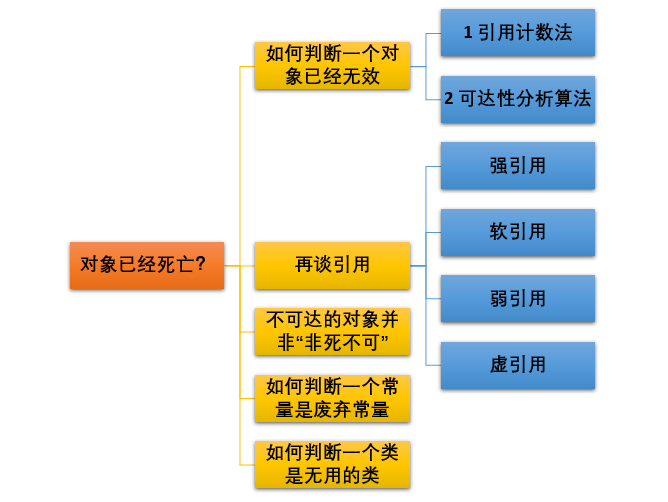
### **大对象直接进入老年代**

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。

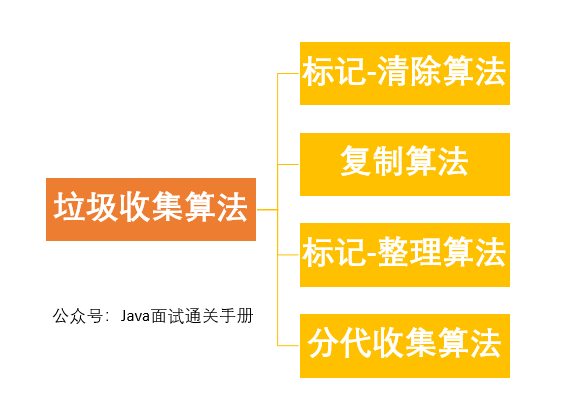
****为什么要这样呢？****

为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

### **长期存活的对象将进入老年代**



## **垃圾收集算法**



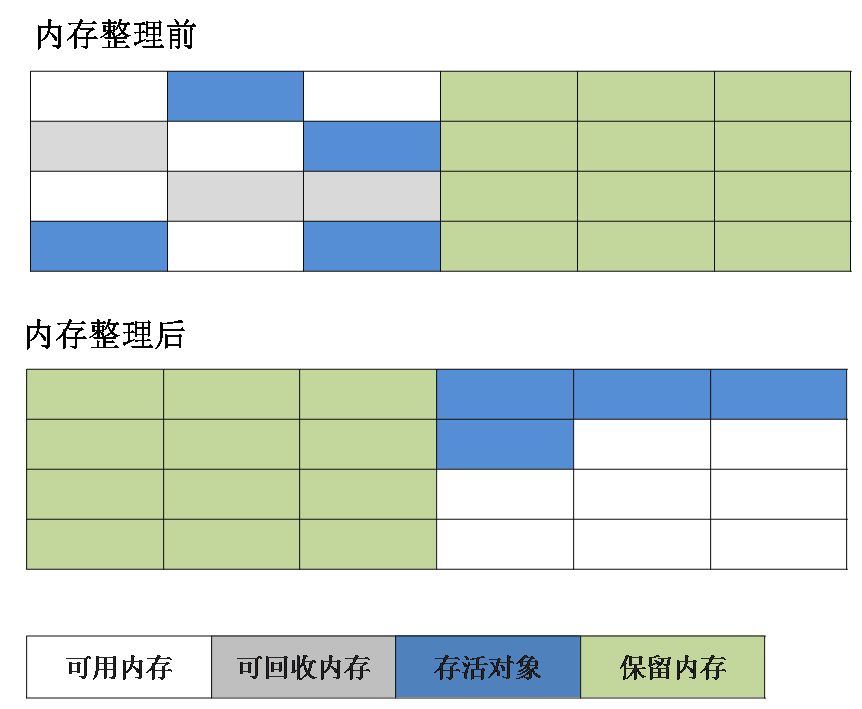
### **标记-清除算法**

算法分为“标记”和“清除”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所有被标记的对象。它是最基础的收集算法，效率也很高，但是会带来两个明显的问题：

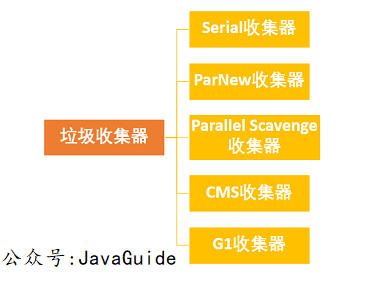
1. ****效率问题****
2. ****空间问题（标记清除后会产生大量不连续的碎片）****

### **复制算法**

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。



## **垃圾收集器**

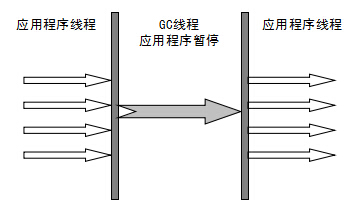


****如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。****

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为知道现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，****我们能做的就是根据具体应用场景选择适合自己的垃圾收集器****。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的 HotSpot 虚拟机就不会实现那么多不同的垃圾收集器了。

### **Serial 收集器**

Serial（串行）收集器收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的 ****“单线程”**** 的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（ ****"Stop The World"**** ），直到它收集结束。

****新生代采用复制算法，老年代采用标记-整理算法。**** [](https://camo.githubusercontent.com/aba41c5c08ea9884554b9a69ea69c7ceeebc83ff/687474703a2f2f6d792d626c6f672d746f2d7573652e6f73732d636e2d6265696a696e672e616c6979756e63732e636f6d2f31382d382d32372f34363837333032362e6a7067)

虚拟机的设计者们当然知道 Stop The World 带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

### **ParNew 收集器**

****ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。****

****并行和并发概念补充：****

****并行（Parallel）**** ：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

****并发（Concurrent）****：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

### **Parallel Scavenge 收集器**

Parallel Scavenge 收集器类似于 ParNew 收集器。 ****那么它有什么特别之处呢？****

-XX:+UseParallelGC

使用 Parallel 收集器+ 老年代串行

-XX:+UseParallelOldGC

使用 Parallel 收集器+ 老年代并行

****Parallel Scavenge 收集器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。****

### **4.6 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。它而非常符合在注重用户体验的应用上使用。****

****CMS（Concurrent Mark Sweep）收集器是 HotSpot 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。****

从名字中的****Mark Sweep****这两个词可以看出，CMS 收集器是一种 ****“标记-清除”算法****实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

* ****初始标记：**** 暂停所有的其他线程，并记录下直接与 root 相连的对象，速度很快 ；
* ****并发标记：**** 同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
* ****重新标记：**** 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行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的时间稍长，远远比并发标记阶段时间短
* ****并发清除：**** 开启用户线程，同时 GC 线程开始对为标记的区域做清扫。

### **G1 收集器**

****G1 (Garbage-First) 是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征.****

被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备一下特点：

* ****并行与并发****：G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。
* ****分代收集****：虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。
* ****空间整合****：与 CMS 的“标记--清理”算法不同，G1 从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
* ****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 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内。

G1 收集器的运作大致分为以下几个步骤：

* ****初始标记****
* ****并发标记****
* ****最终标记****
* ****筛选回收****

****G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来)****。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 GF 收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。

# **3,Java 类加载机制**

**JVM将类加载过程分为三个步骤：装载（Load），链接（Link）和初始化(Initialize)。**1) 装载：查找并加载类的二进制数据

2) 链接：

验证：确保被加载类的正确性；

准备：为类的静态变量分配内存，并将其初始化为默认值；

解析：把类中的符号引用转换为直接引用；

1. 初始化：为类的静态变量赋予正确的初始值

**2.2 类的初始化**

类什么时候才被初始化：

      1）创建类的实例，也就是new一个对象

　　2）访问某个类或接口的静态变量，或者对该静态变量赋值

　　3）调用类的静态方法

　　4）反射（Class.forName("com.lyj.load")）

　　5）初始化一个类的子类（会首先初始化子类的父类）

　　6）JVM启动时标明的启动类，即文件名和类名相同的那个类

只有这6中情况才会导致类的类的初始化。类的初始化步骤：

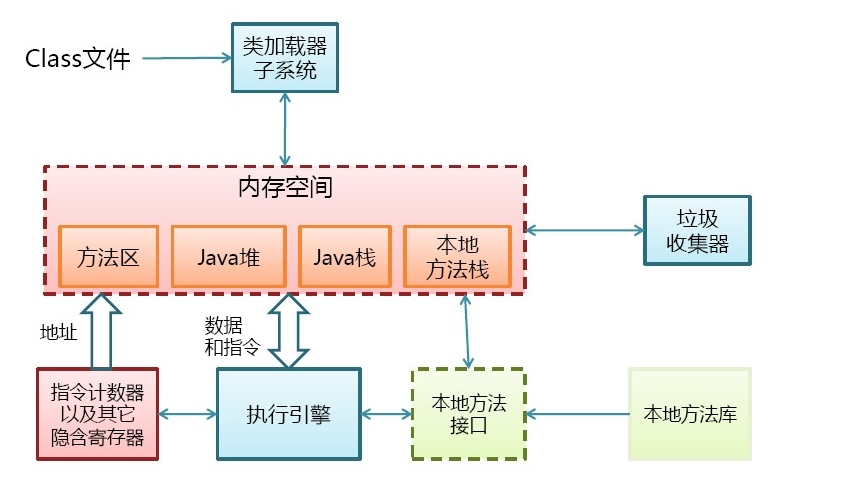
        1）如果这个类还没有被加载和链接，那先进行加载和链接

        2）假如这个类存在直接父类，并且这个类还没有被初始化（注意：在一个类加载器中，类只能初始化一次），那就初始化直接的父类（不适用于接口）

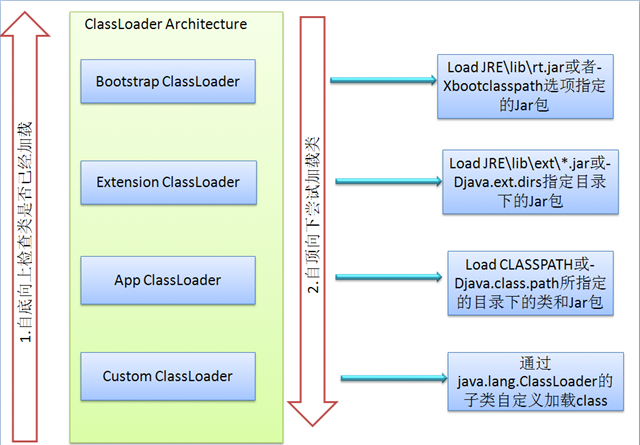
        3) 假如类中存在初始化语句（如static变量和static块），那就依次执行这些初始化语句。

**类的加载**

      类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个这个类的java.lang.Class对象，用来封装类在方法区类的对象。看下面2图



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：



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/ext/\*.jar或-Djava.ext.dirs指定目录下的jar包

3）App ClassLoader

　　负责加载classpath中指定的jar包及目录中class

4）Custom ClassLoader

JVM的类加载机制主要有如下3种。

全盘负责：所谓全盘负责，就是当一个类加载器负责加载某个Class时，该Class所依赖和引用其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入。

双亲委派：所谓的双亲委派，则是先让父类加载器试图加载该Class，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类。通俗的讲，就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父加载器，依次递归，如果父加载器可以完成类加载任务，就成功返回；只有父加载器无法完成此加载任务时，才自己去加载。

缓存机制。缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区中搜寻该Class，只有当缓存区中不存在该Class对象时，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓冲区中。这就是为很么修改了Class后，必须重新启动JVM，程序所做的修改才会生效的原因。

双亲委派机制，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每个儿子都很懒，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己才想办法去完成。