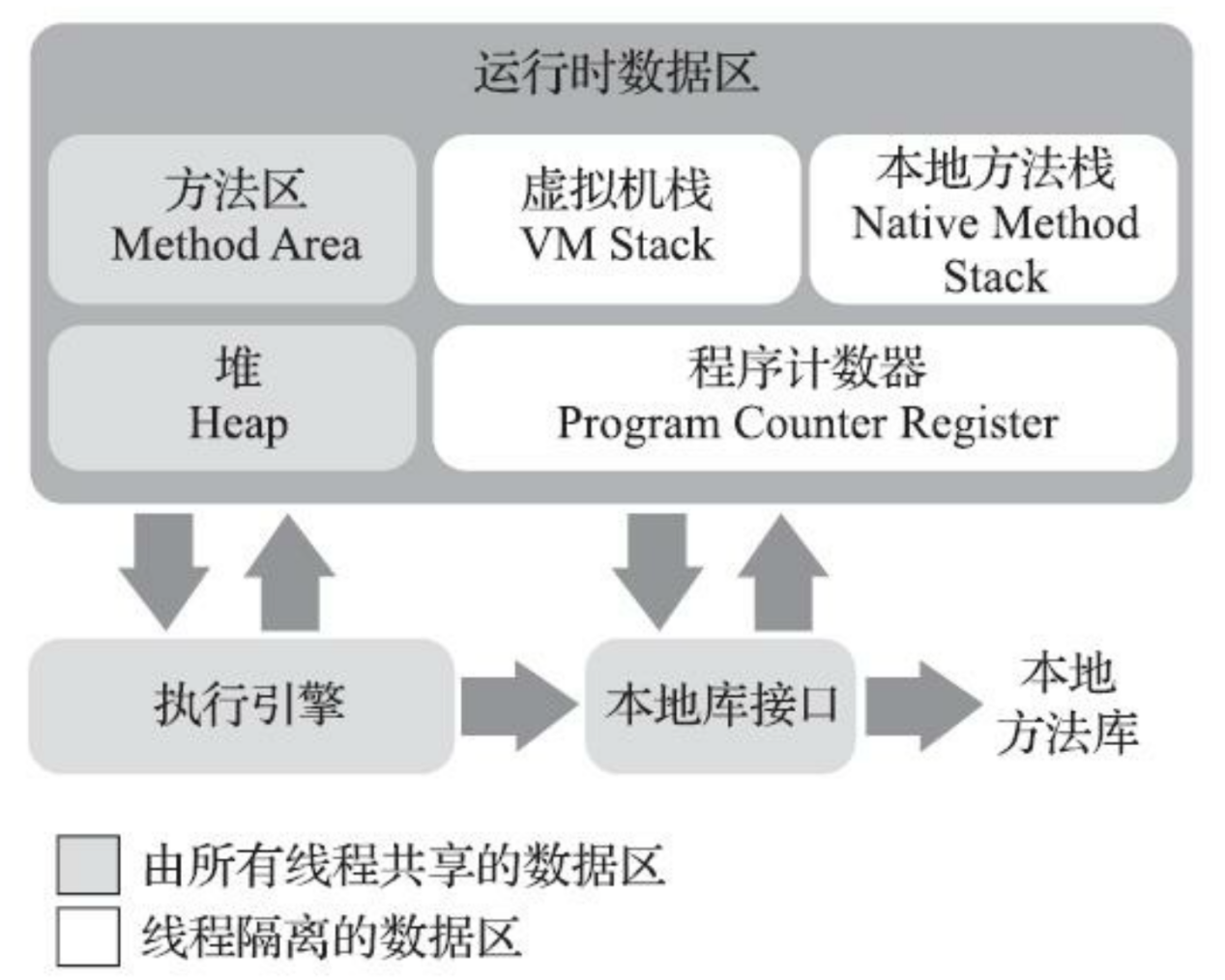
1. **Java内存区域与内存溢出异常**

**1、运行时数据区**

**Java虚拟机把其所管理的内存分为了几个不同的数据区：**1.1 **程序计数器（作用、多线程及线程私有、唯一没有OOM情况）**

程序计数器是当前线程执行的字节码的行号指示器，其工作就是通过改变计数器的值**选取下一条需要执行的字节码指令**，分支、循环、跳转、异常处理等都需要依赖这个计数器完成。

由于Java虚拟机的多线程是通过线程轮流切换、分配处理器执⾏时间的⽅式来实现的，

在任何⼀个确定的时刻，⼀个处理器(对于多核处理器来说是⼀个内核)都只会执⾏⼀条线

程中的指令。因此，**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**，各条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。

执行的是Java方法时，计数器记录的是正在执行的虚拟机字节码指令的地址；执行的是native方法时，计数器记录的值为空，计数器是唯一没有规定任何OOM情况的区域。

**1.2 Java虚拟机栈（线程私有及栈的作用、内部结构、局部变量表结构及内存分配、异常）**

也是**线程私有**的，生命周期和线程相同。描述的是Java方法执行的线程内存模型：每个**方法执行时**虚拟机栈都会创建一个**栈帧**储存**局部变量表**、**操作数栈**、动态连接、方法出口等信息。每个方法从调用到执行完毕的过程对应了一个栈帧从入栈到出栈的过程。

**局部变量表**放了编译期可知的各种Java虚拟机的**基本数据类型**（boolean、byte、char、short、int、float、long、double）、**对象引用**（reference类型，一般是一个指向对象起始地址的引用指针或者是指向一个代表对象的句柄）、**returnAddress**类型（指向一条字节码的地址）。

在局部变量表中以**局部变量槽**（Slot）表示。64位长度的long和double类型数据会占两个变量槽，其余数据类型各占一个。局部变量表所需的内存在**编译时就分配完毕**了，方法运行期间局部变量表大小不会变化。

一般在HotSpot虚拟机栈出现的异常是StackOverflowError，请求深度大于栈允许深度。其他虚拟机栈可能会出现由于申请动态扩展而所需内存不够导致的OOM异常。

**1.3本地方法栈**

主要是为了调用本地方法而非Java方法供虚拟机使用，在调用本地方法栈时技术器的记录的值为空null。

**1.4Java堆（线程共享、存放对象实例、GC内存区域及各区域名称、TLAB提升效率、分配规则小对象可以不连续，大对象必须连续、会出现OOM异常）**

是JVM内存管理中最大的一块，是**线程共享**的区域，主要用来**存放对象实例**，几乎所有对象都在堆中储存。由于即时编译的技术，栈上分配会储存在栈上，所以不是绝对的。

堆是垃圾收集器管理的主要内存区域，被分成了**新生代**（Eden空间、From Survivor空间、To Survivor空间）、**老年代**。

从分配内存角度看，堆可以分成多个线程私有的**分配缓冲区（TLAB）**，当多个线程同时实例化时，不需要加锁也可以保证安全，提升对象分配效率。

Java堆可以处于物理上不连续的内存空间中，但是逻辑上是连续的。但是对于大对象，要求是需要连续的内存空间。

堆是可以通过-Xmx和-Xms分配最大和最小堆内存，内存不足时抛出OOM异常。

**1.5方法区（线程共享、储存信息、实现方式1.6/1.8及其问题、抛出异常OOM）**

**线程共享**的内存区域，用于储存被虚拟机加载的**类型信息、常量、静态变量**、即**时编译器编译后的代码缓存等数据**。

1.8之前方法区的实现方式是永久代，1.8开始实现方式是元空间，使用了堆的一部分内存，但是和堆不是一个定义。

使用永久代可能会有**内存溢出**问题，1.6时放弃了永久代，逐步改为采用本地内存实现方法区，1.7时把**字符串常量池**和**静态变量**移动到了java堆中，1.8完全废弃永久代概念，在本地内存实现元空间代替，把1.7中永久代剩余的内容（主要是类型信息）移动到元空间中。

这部分垃圾收集较少，主要是回收常量池不常用的部分和对类型的卸载。

永久代的**内存溢出**主要集中在动态代理等动态生成类，由于永久代在类加载时就已确定内存大小，因此当动态生成类较多时，类的相关信息过多可能会导致永久代内存溢出。

抛出异常是OOM。

**1.6运行时常量池（方法区一部分、存放字面量和符号引用及部分直接引用、具备动态性，可以在运行期间加入新常量、抛出OOM异常）**

方法区的一部分，Class文件除了有**类的版本、字段、方法、接口**等描述信息外，还有一个**常量池表**，用于存放编译期生成的**各种字面量与符号引用**，这部分内容在**类加载**后存放到方法区的**运行时常量池**中。

除了保存Class文件中描述的符号引用外，也会把由**符号引用**翻译出来的**直接引用**储存在运行时常量池中。

运行时常量池相对Class文件常量池另一个重要特征是具备**动态性**，并不要求常量只能在编译期产生，并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，也可以在运行期间把新的常量放入池中。比如String类的intern()方法。

运行时常量池是方法区一部分，所以抛出异常时是OOM。

**1.7直接内存（属于堆外内存，不受堆大小控制，使用NIO通信方式通过堆中对象引用使用内存）**

并不是虚拟机运行时数据区的一部分，是堆外内存。通过**NIO**类，基于**管道（Channel）和缓冲区（Buffer）**的I/O方式，使用Native库函数直接分配**堆外内存**，通过一个储存在堆里面的**DirectByteBuffer**对象作为这块内存的引用进行操作。**避免了在堆和Native堆中来回复制数据。**

本机直接内存分配不收堆大小控制，但是受本机总内存限制，可能会导致OOM异常。

**2、HotSpot虚拟机对象揭秘**

**2.1对象创建（类加载检查、为对象分配内存、对其分配零值）**

虚拟机遇到new关键字指令时，首先检查这个指令**参数是否能在常量池中定位到一个类的符号引用**，并**检查这个符号引用代表的类是否已被加载、解析、初始化过**，如果没有则执行相应的**类加载**过程。

类加载检查通过后，为新生对象**分配内存**，相当于在堆中划分出一块内存，所需内存大小在类加载完成后可完全确定。

（这里有两种分配内存的方式，一种是**指针碰撞**，使用过的内存放一边，空闲内存另一边，中间有一个指针，分配内存时把指针向空闲内存移动一部分即可。另一种是使用了**空闲列表**，如果堆中内存不规整，需要维护一个列表记录哪些内存块可用，分配时从列表找到一块足够大的内存划分给对象实例，并更新列表。选择哪种方式取决于内存是否规整，即取决于垃圾回收是标记整理还是标记清除。使用Serial、ParNew等回收器，分配算法是指针碰撞，使用CMS回收器时，理论上只能使用空闲列表这种复杂的分配内存方式。）

由于对象创建是很频繁的行为，并发情况下会有安全问题，可能出现给A分配内存，指针还没修改，B同时使用原来的指针分配内存的情况。解决方式第一种是使用**CAS配上失败重试**保证更新操作原子性；第二种方式是把内存分配动作按照线程划分在不同区域中进行，每个线程预先分配一小块内存，称为**本地线程分配缓冲（TLAB）**。哪个线程需要分配内存就在哪个缓冲区分配，不需要加锁，只有缓冲区用完了才会同步锁定。

内存分配完成后，需要**将分配到的内存空间初始化为零值**，如果使⽤了TLAB的话，这⼀项⼯作也可以提前⾄TLAB分配时顺便进⾏。保证对象实例在代码中可以不赋值就可以直接使用。

此时一个新对象已经产生了，但是需要**对其初始化**，Class文件中的**<init>()**方法还没执行，所有字段默认为0，new字段后会接着执行<init>()方法，然后一个可用对象会真正构造出来。

**2.2对象的内存布局（对象头、实例数据、对齐填充）**

对象在堆中储存布局分为：**对象头**、实例数据、对齐填充。



**对象头**有两类信息：一类是**储存对象自身的运行时数据**，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。另一类是**类型指针**，即**对象指向他的类型元数据指针**，通过这个指针**确定该对象是哪个类的实例**。但查找对象的元数据信息不一定要经过对象本身，所以不一定所有虚拟机的对象数据上都保留类型指针。

如果对象是**数组**，对象头还要有一块**记录数组长度的数据**，如果长度不确定，无法通过元数据信息推断数组大小。

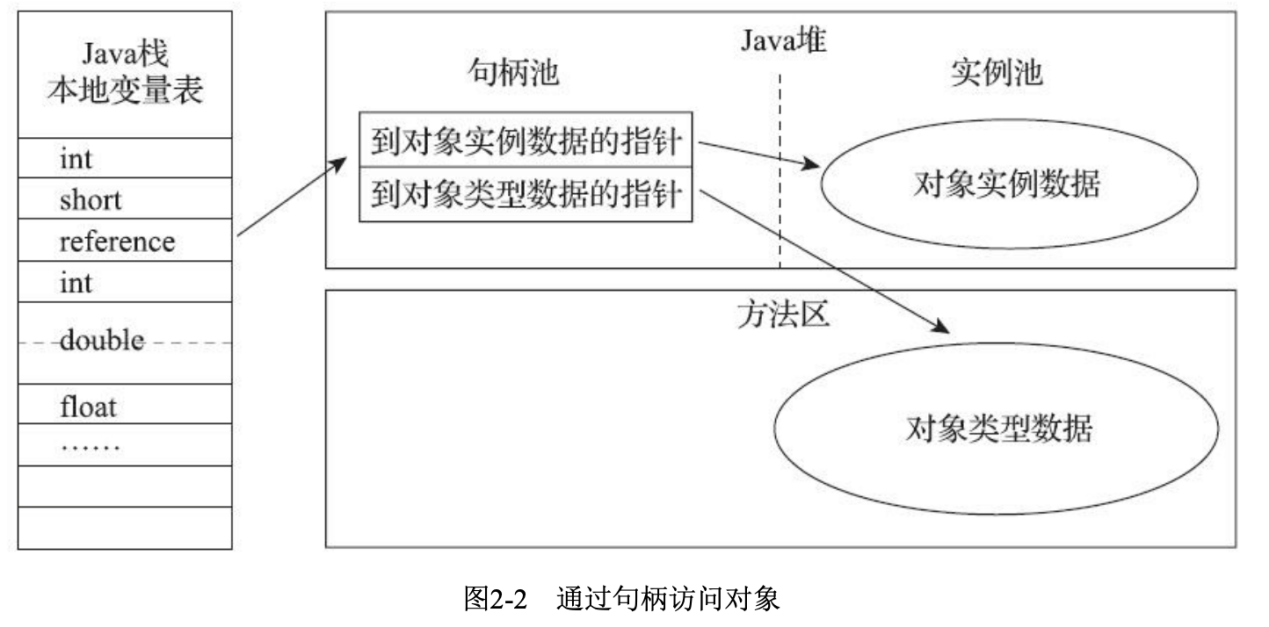
**实例数据**部分是对象储存信息的位置，即我们在程序代码⾥⾯所定义的各种类型的**字段内容**，⽆论是从⽗类继承下来的，还是在⼦类中定义的字段都必须记录起来。**相同宽度的字段总是被分配到⼀起存放**，在满⾜这个前提条件的情况下，**在⽗类中定义的变量会出现在⼦类之前**。

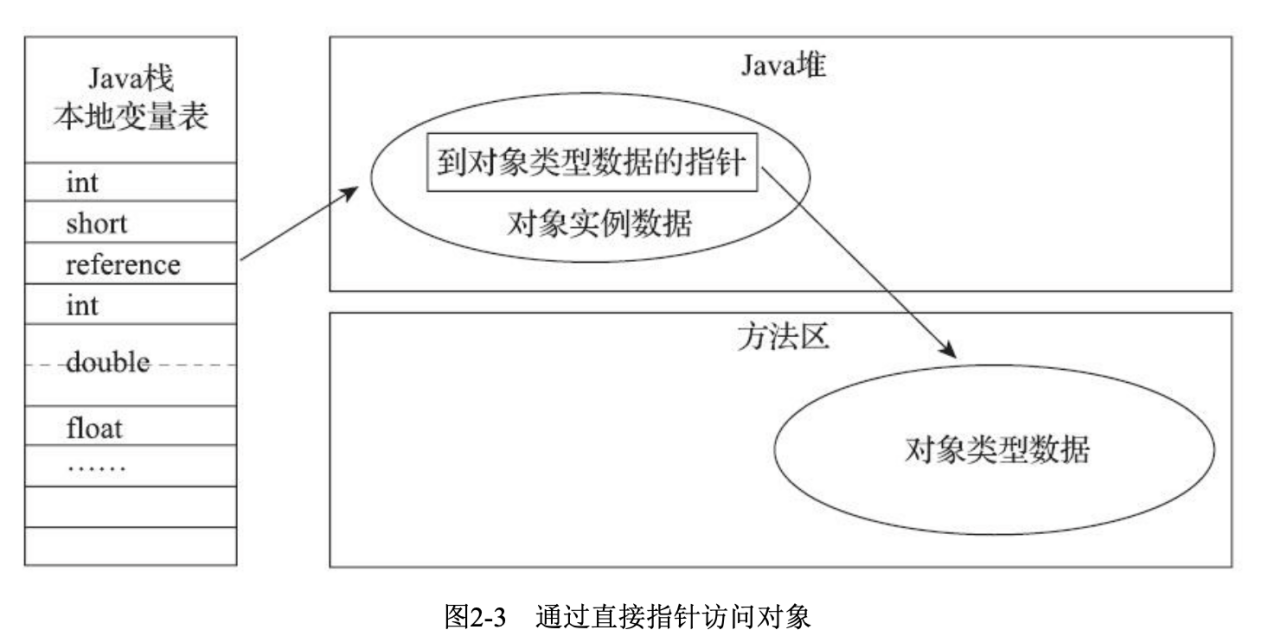
**对齐填充**起到了占位符的作用虚拟机自动内存管理系统要求任何对象大小都是8字节的整数倍，对象头部分已经设计成**8字节**的倍数，如果实例数据部分没有对齐就需要对齐填充补全。

**2.3对象的访问定位**

对象会通过栈上的**reference**数据操作堆上的具体对象，reference是一个**指向对象的引用**，主流访问方式有**使用句柄和直接指针**两种。

通过**句柄**访问对象，堆中会划分一块内存作为**句柄池**，reference指向**句柄地址**，句柄池中包含了**对象实例数据**和**对象类型数据**的地址。



通过直接指针访问对象，reference中存储的**直接就是对象地址**，如果只是访问对象本身的话，就不需要多⼀次间接访问的开销。

使用**句柄**的好处是对象被移动（垃圾收集时移动对象是很普遍的）时**只会改变句柄池中的实例数据指针，栈中reference指向的句柄地址不需要修改**。

使用**直接指针**的好处是**速度更快，减少一次指针定位的时间开销**，对象访问在Java中很频繁，所以HotSpot使用这种方式的对象访问。

1. **垃圾收集器与内存分配策略**
2. **概述**

**哪些内存需要回收？什么时候回收？何时回收？**

**堆和方法区**有不确定性：**一个接口多个实现类需要的内存可能不一样，一个方法执行的不同条件分支所需的内存也可能不一样**，**只有运行期间才能确定程序创建的对象及其个数**。垃圾回收就是关注这部分动态分配内存的管理。

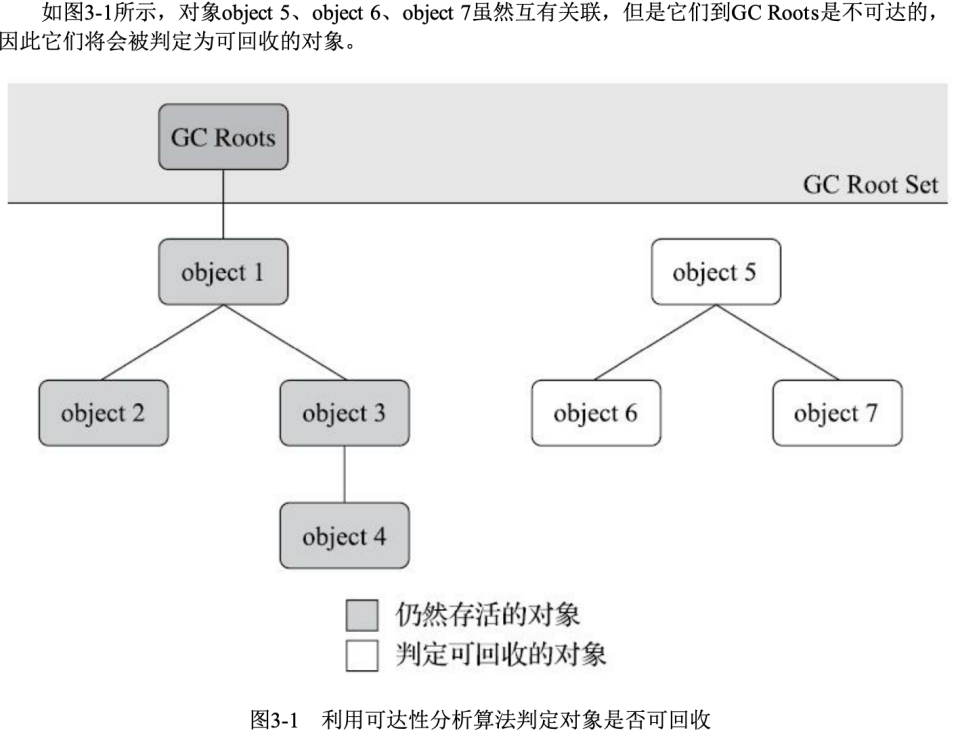
1. **对象死亡判断**

**2.1引用计数法**

在对象中添加一个**计数器**，有一个地方引用这个对象时，计数器+1，反之则-1；计数器为0的就是没有被使用的对象就可以回收了。但是会有**循环引用**的风险，即A引用B，B引用A但是计数器不为0，无法回收导致内存泄漏。

**2.2可达性分析算法**

基本思路是通过一系列称为**GC Roots**的根对象作为起始节点集，从这些节点开始遍历向下搜索引用的对象，搜索的路径称为**“引用链”**，如果某个对象没有和任何引用链相连，则说明该对象没有被使用不可达。



**可以作为GC Roots的对象包括以下：**

**栈中引用的对象**，如当前正在运行的方法用到的参数、局部变量、临时变量。

**方法区中类静态属性引用的对象**，如Java类的引用类型静态变量。

**方法区中常量引用的对象，**如字符串常量池的引用。

**本地方法栈中Native方法引用的对象**。

**虚拟机内部的引用**，如基本数据类型对应的Class对象，一些异常对象（如空指针异常、OOM错误），还有系统类加载器。

**被同步锁持有的对象。**

反应虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等。

**2.3引用**

四种引用：强引用、软引用、弱引用、虚引用。

**强引用**：正常new出来的对象。

**软引用**：用软引用关联的对象，初次GC时不会回收，当再次OOM之前，会二次回收软引用的对象，如果回收了软引用仍然没有足够内存才会抛出内存异常。用SoftReference类实现软引用。

**弱引用**：弱引用关联的对象只能存活到下次GC，GC会把弱引用的对象进行回收，用WeakReference类实现弱引用。

**虚引用**：为一个对象设置虚引用后，在这个对象被回收时会受到系统通知。

**2.4对象的濒死挣扎（两次标记）**

即使在可达性分析算法中判定为不可达的对象，也不是“⾮死不可”的，这时候它们暂时还处于**“缓刑”**阶段，要真正宣告⼀个对象死亡，⾄少要经历**两次标记过程**:

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**，随后进⾏⼀次筛选，筛选的条件是**此对象是否有必要执⾏finalize()⽅法**。假如对象没有覆盖finalize()⽅法，或者finalize()⽅法已经被虚拟机调⽤过，那么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确有必要执⾏finalize()⽅法，那么该对象将会被放置在⼀个名为**F-Queue的队列**之中，并在稍后由⼀条**由虚拟机⾃动建⽴的、低调度优先级的Finalizer线程**去执⾏它们的finalize()⽅法。finalize()⽅法是对象逃脱死亡命运的最后⼀次机会，稍后收集器将对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，那基本上它就真的要被回收了。**(这种⾃救的机会只有⼀次，因为⼀个对象的finalize()⽅法最多只会被系统⾃动调⽤⼀次)**

**2.5回收方法区（从常量池清理废弃常量和对不再使用的类进行卸载，第二种主要用于反射、动态代理等情况，此时需要虚拟机具备类型卸载能力，不会对方法区造成过大内存压力）**

方法区垃圾回收成果往往低于70%~99%，主要回收：**废弃的常量和需要卸载的类型**。

**常量池**和回收堆中的对象类似，假如“Java”进入了**常量池**，但是当前系统没有字符串对象的值是“Java”，没有引用“Java”常量的字符串对象，且虚拟及其他地方也没有引用这个字面量。此时内存回收且垃圾回收期认为需要，那么这个常量会被清理出常量池。常量池中其他**类（接口）、方法、字段的符号引用**也类似。

判断**是否是不再使用的类**的条件比较苛刻，需同时满足以下三个条件：

**该类所有的实例都已经被回收**，也就是Java堆中不存在该类及其任何派⽣⼦类的实例。

**加载该类的类加载器已经被回收**，这个条件除⾮是经过精⼼设计的可替换类加载器的场景，如OSGi、JSP的重加载等，否则通常是很难达成的。

**该类对应的java.lang.Class对象没有在任何地⽅被引⽤**，⽆法在任何地⽅通过反射访问该类的⽅法。

1. **垃圾搜集算法**

**3.1分代收集理论**

**建立在两个分代假说之上：**

**弱分代假说：大多数对象都是朝生夕灭的。**

**强分代假说：度过垃圾回收次数越多的对象越难消亡。**

把分代收集理论具体放到现在的商⽤Java虚拟机⾥，设计者⼀般⾄少会把Java堆划分为**新⽣代**(Young Generation)和**⽼年代**(Old Generation)两个区域。顾名思义，在新⽣代中，每次垃圾收集时都发现有⼤批对象死去，⽽每次回收后存活的少量对象，将会逐步晋升到⽼年代中存放。

诸如标记-清除、标记-复制、标记-整理等算法都是始于分代收集理论。

分代后会出现**跨代引用**问题，即新生代对象可能被老年代引用，为了找到这个存活对象**，不得不额外遍历老年代**所有对象来确保可达性分析结果的正确性，反过来也是一样。遍历老年代会**导致内存回收的性能降低**，因此添加了第三个假说：

**跨代引用假说：跨代引用相对同代引用占极少数，存在互相引用关系的两个对象倾向于同时生存或消亡。**

依据这个假说，不再为少量的跨代引用扫描遍历老年代，只需要**在新生代建立一个全局数据结构“记忆集**”，**把老年代划分成若干小块，标识出老年代哪一块内存会存在跨代引用**。这样当发生**Minor GC**时，只有**包含了跨代引用的小块内存**才会被加入GC Roots进行扫描。这样会增加一部分运行时开销，但是相对遍历老年代会好很多。

**部分收集(**Partial GC): 指⽬标不是完整收集整个Java堆的垃圾收集，其中⼜分为:

新⽣代收集(Minor GC/Young GC): 指⽬标只是新⽣代的垃圾收集。

⽼年代收集(Major GC/Old GC): 指⽬标只是⽼年代的垃圾收集。⽬前只有**CMS**收集器会有单独收集⽼年代的⾏为。

混合收集(Mixed GC): **指⽬标是收集整个新⽣代以及部分⽼年代的垃圾收集**。⽬前只有

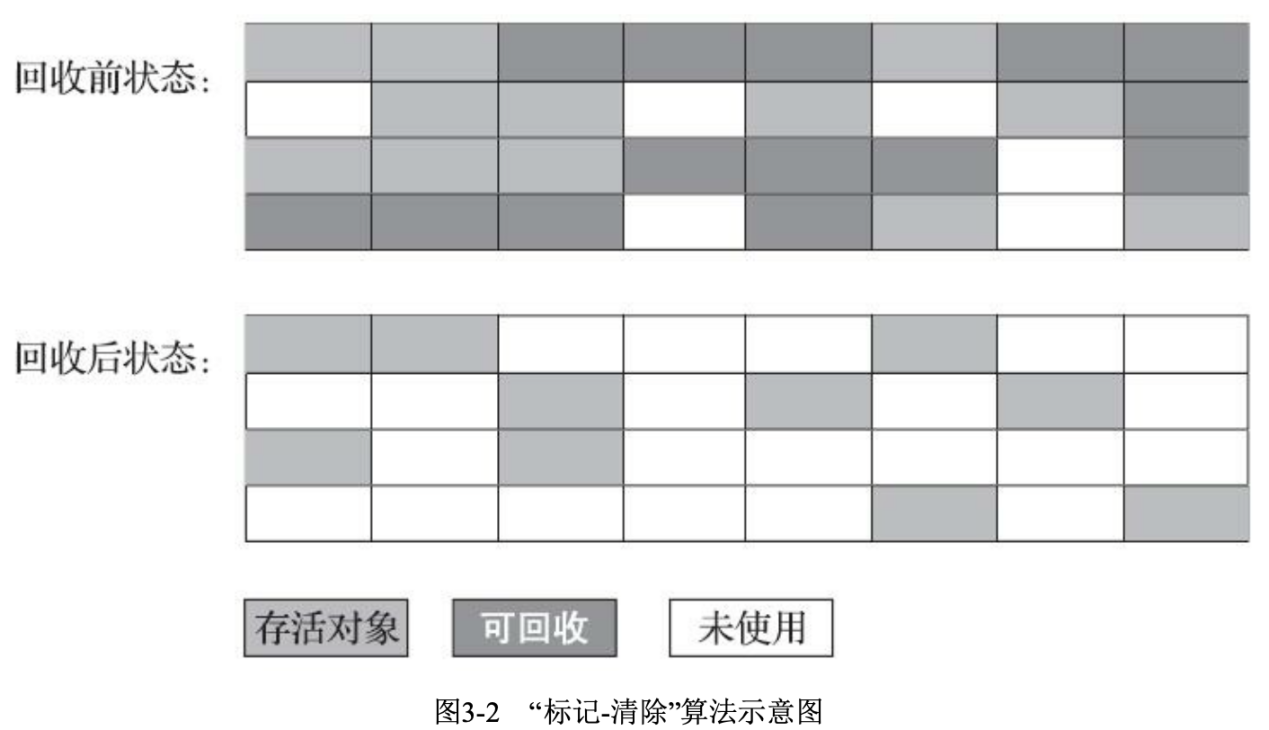
G1收集器会有这种⾏为。

·**整堆收集**(Full GC): 收集整个Java堆和⽅法区的垃圾收集。

**3.2标记-清除算法**

首先**标记需要回收的对象，标记完成后统一回收**，或者标记存货对象，标记完成后回收未标记对象。

两个缺点：1、**执行效率不稳定**，堆中大部分对象是需要回收的，此时必须大量标记并清除，执行效率随对象数量的增加而降低。

2、**会产生内存碎片**，清除后不整理会产生大量不连续内存碎片，如果分配对象过大会导致无法分配从而触发一次GC。

**3.3标记-复制算法（多用于新生代）**

**将内存划分成大小相等的两块，每次只使用一半内存，当这块内存满了，就标记存活的对象复制到另一块内存，然后清除掉当前内存所有对象。**

如果内存**对象中大部分是存活的**会复制大量存活对象导致**内存开销很大**，但是多数内存需要回收时性能会好。并且不会有内存碎片问题。缺点是**浪费了一半的内存**。

新生代对象中**98%**不会熬过第一轮收集，所以把新生代分为**Eden区、From Survivor区和To Survivor区**，三者占比为**8:1:1**，仅浪费了**10%**的内存，并且在每次垃圾回收存活对象不超过总内存10%的前提下设置的占比。

如果**存活对象超过10%**，使Survivor区没有空间存放上次新生代收集的存活对象，会通过**内存的分配担保机制**直接进入老年代。

**3.4标记-整理算法（多用于老年代）**

标记所有存活对象，并不对可回收对象清理，而让所有存活对象**向一端移动**，然后直接**清理掉边界以外的内存**。

如果移动存活对象，尤其是在老年代这种每次回收都有大量对象的存活区域，**移动存活对象并更新所有引用这些对象的地方**会是一种极为**负重的操作**，并且这种对象移动操作必须**全程暂停用户应用程序才能进行**，即**Stop The World**。

如果像标记-清除那样不考虑移动和整理存活对象，空间碎片化问题只能依赖**更复杂的内存分配器和内存访问器**。**内存访问**是用户程序**最频繁操作**，这个环节增加了负担会**影响应用程序吞吐量**。

基于以上两点，从整个程序吞吐量看，**移动对象更划算**。即使不移动对象使收集器效率提升一些，但因**内存分配和访问**相比**垃圾收集频率高**的多，这部分耗时增加，总吞吐量仍然是下降的。

Parallel Old收集器是基于标记-整理算法的，关注**吞吐量**；CMS收集器基于标记-清除算法的，关注**延迟**。并且CMS大多是标记清除算法，暂时容忍碎片，碎片化程度影响对象分配时，会进行一次标记整理算法。

**4 HotSpot算法细节实现**

**4.1根节点枚举**

可作为GC Roots的节点主要在**全局性引用**（常量或类静态属性）与**执行上下文**（栈帧中本地变量表）中。

根节点枚举不需要一个不漏的检查完所有**执行上下文和全局的引用位置**，而是有办法直接得到哪些地方存放着对象引用的。使用一组称为**OopMap**的数据结构来达到这个⽬的。类加载完成后， HotSpot就会把对象内什么偏移量上是什么类型的数据计算出来，在即时编译过程中，也会在特定的位置记录下栈⾥和寄存器⾥哪些位置是引⽤。这样收集器在扫描时就可以直接得知这些信息了，并不需要真正⼀个不漏地从⽅法区等GC Roots开始查找。

**OopMap相当于记录了GC Roots的根节点，可以直接通过获得OopMap得到，就不用搜索这些节点了。**

**4.2安全点**

在OopMap协助下，可以迅速完成GC Roots枚举。由于OopMap过多会**增加内存压力**，因此会在一些**特定位置**记录这个位置的OopMap，这个特定位置就是**安全点**。**强制线程在到达安全点后才能暂停**。**安全点选定既不能太少使收集器等待时间过长，也不能过多增加运行时内存负荷**。安全点一般设定在方法调用、循环跳转、异常跳转等**指令序列复用**逻辑时，这时候**程序可能会长时间执行**，因此在这种位置设置安全点。

为了让所有线程都能跑到最近的安全点，采用**主动式中断**：当垃圾收集需要中断线程时，不直接对线程操作，仅简单**设置标志位**，各线程执行时主动轮询这个标志，发现中断标志为真时就在最近安全点**主动挂起**。**安全点和中断标志是重合的**。

**4.3安全区域**

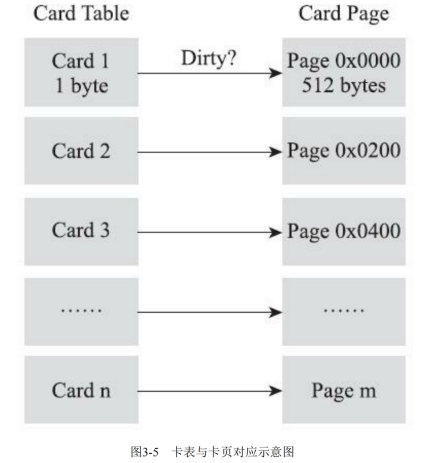
**在程序Sleep或者Blocked时无法响应虚拟机中断请求，因此无法走到安全点中断挂起**，因此需要安全区域。

安全区域就是**扩展拉伸的安全点**，在这个区域任意地方开始垃圾收集都是安全的。用户线程进入安全区域后会标识自己进入了安全区域，线程要离开时，如果完成了根节点枚举，顺利离开，如果没完成要等待其他线程完成后收到离开安全区域信号后才能离开。

**4.4记忆集和卡表（解决跨代引用的实现方式及具体数据结构）**

**记忆集是⼀种⽤于记录从⾮收集区域指向收集区域的指针集合的抽象数据结构**。卡表是记忆集的实现，相当于HashMap和Map的关系。

**卡精度**: 每个记录精确到⼀块内存区域，该区域内有对象含有跨代指针。

**卡表**的每一个元素都对应着其表示的内存区域中一块特定大小的内存块，这个内存块叫**卡页**。

⼀个卡⻚的内存中通常包含不⽌⼀个对象，只要卡⻚内有⼀个(或更多)对象的字段存在着跨代指针，那就将对应卡表的数组元素的值标识为1，称为这个元素变脏(Dirty)，没有则标识为0。在垃圾收集发⽣时，只要筛选出卡表中变脏的元素，就能轻易得出哪些卡⻚内存块中包含跨代指针，把它们加⼊GC Roots中⼀并扫描。

**卡表就是相当于一个把老年代分成了几个区域，每个卡表有512字节内存的卡页，一个卡页包含不止一个对象，如果卡页中对象因跨代引用存在跨代指针，就把卡页相应区域从0变为1，称为变脏。垃圾收集时只要筛选出变脏区域中的元素就可以得出哪些对象跨代引用，把他们加入GC Roots一块扫描。**

**4.4写屏障**

在HotSpot虚拟机⾥是通过写屏障技术**维护卡表状态的**。写屏障可以看作在虚拟机层⾯对“引⽤类型字段赋值”这个动作的**AOP切⾯**，在引⽤对象赋值时会产⽣⼀个环形(Around)通知，供程序执⾏额外的动作，也就是说赋值的前后都在写屏障的覆盖范畴内。在赋值前的部分的写屏障叫作**写前屏障**(Pre-Write Barrier)，在赋值后的则叫作**写后屏障(**PostWrite Barrier)。HotSpot虚拟机的许多收集器中都有使⽤到写屏障，但直⾄G1收集器出现之前，其他收集器都只⽤到了**写后屏障**。

**4.5并发可达性分析:**

**三色标记：**

**白色**：对象尚未被垃圾收集器访问过

**灰色**：对象被垃圾收集器访问过但是对象至少存在一个引用还没被扫描过。

**黑色**：对象及其所有引用都被扫描过。黑色不可能直接指向白色对象。

并发条件下，可能**把原本消亡的的对象错误标记为存活**，导致**浮动垃圾**，可以在下次垃圾收集回收。

如果**把原本存活的对象错误标记为消亡**，会导致程序错误，不能容忍。

解决方案：**增量更新和原始快照**

我们要解决并发扫描时的对象消失问题，只需破坏这两个条件的任意⼀个即可。由此分

别 产⽣了两种解决⽅案:**增量更新**(Incremental Update)和**原始快照**(Snapshot At The

Beginning， SATB ) 。

**增量更新**要破坏的是第⼀个条件，当⿊⾊对象插⼊新的指向⽩⾊对象的引⽤关系时，就将

这个新插⼊的引⽤记录下来，等**并发扫描结束之后，再将这些记录过的引⽤关系中的⿊⾊**

**对象为根，重新扫描⼀次。**这可以简化理解为，⿊⾊对象⼀旦新插⼊了指向⽩⾊对象的引

⽤之后，它就变回灰⾊对象了。

**原始快照**要破坏的是第⼆个条件，当灰⾊对象要删除指向⽩⾊对象的引⽤关系时，就将这

个要删除的引⽤记录下来，在并发扫描结束之后，再将这些记录过的引⽤关系中的灰⾊对

象为根，重新扫描⼀次。这也可以简化理解为，⽆论引⽤关系删除与否，都会按照刚刚开

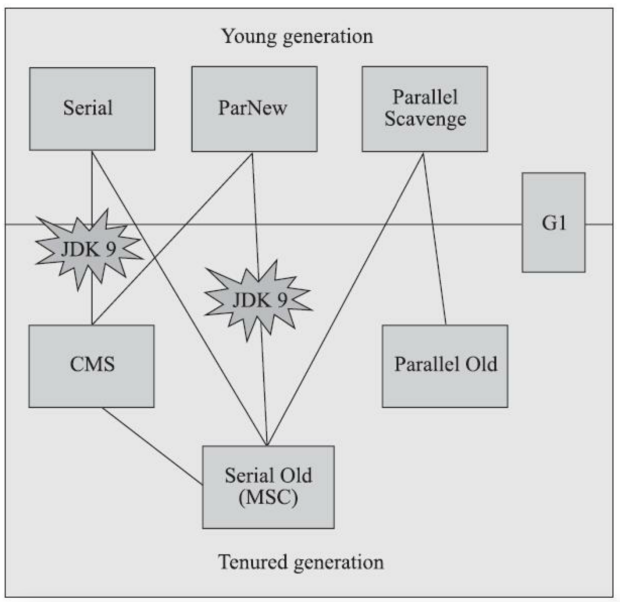
始扫描那⼀刻的对象图快照来进⾏搜索。

以上⽆论是对引⽤关系记录的插⼊还是删除，虚拟机的记录操作都是通过**写屏障**实现的。

在HotSpot虚拟机中，增量更新和原始快照这两种解决⽅案都有实际应⽤，譬如，**CMS**

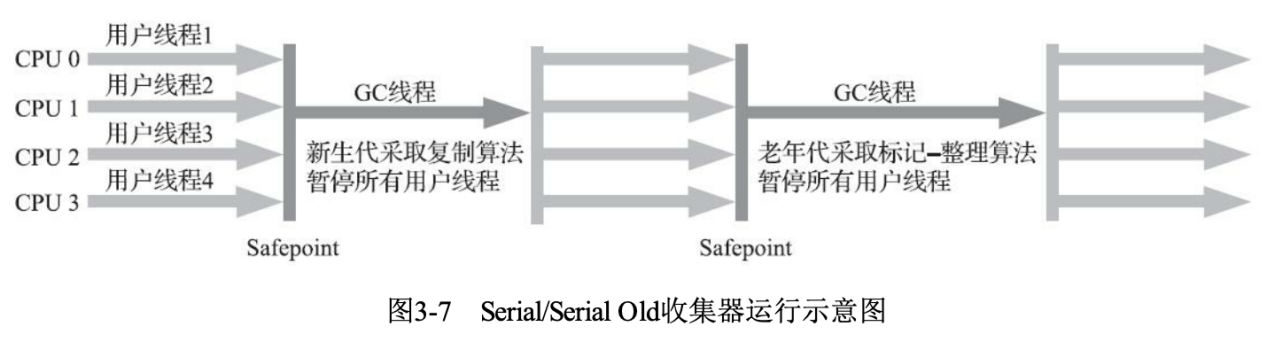
**是基于增量更新来做并发标记的，G1、Shenandoah则是⽤原始快照来实现**。

**5 经典垃圾收集器**



**5.1Serial收集器（单线程、标记-复制）**

**在垃圾收集时必须暂停其他用户线程，用单线程进行垃圾回收。**



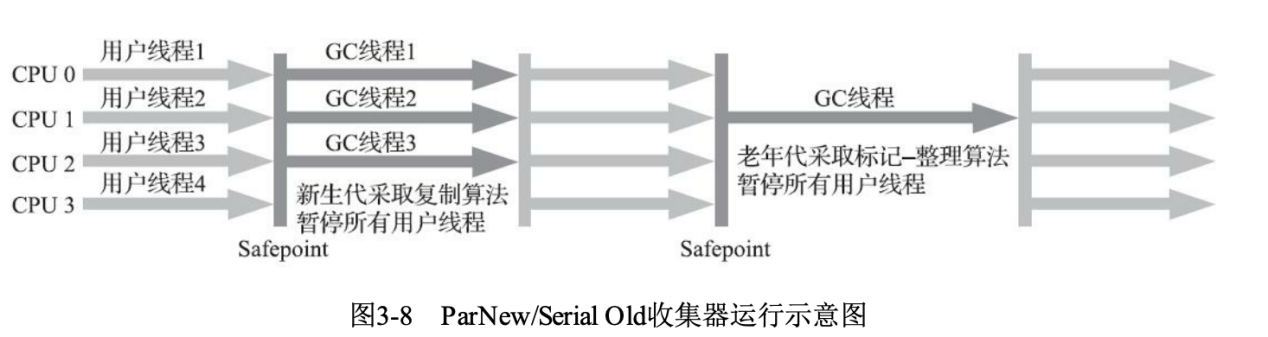
Stop The World 是在用户不可知不可控的情况下强行停掉用户线程。

对于内存受限的环境，Serial是所有收集器中内存消耗最小的。对于单核处理器，Serial没有线程交互的开销，因此效率更高。

Serial虽然基本淘汰了，但是仍然是运行在客户端模式下的虚拟机的很好的选择。

**5.2ParNew收集器（Serial的多线程并行版本、标记-复制）**

用**多线程并行**进行垃圾回收，其他同Serial。



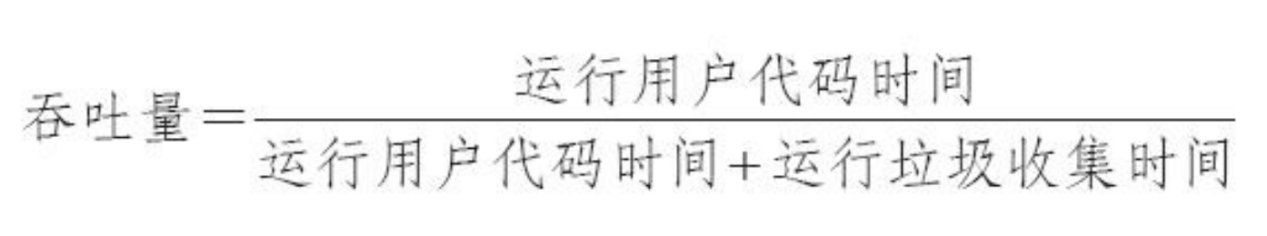
是运行在服务端下虚拟机的**新生代垃圾回收器**，只有**ParNew**才能和**CMS**回收器配合工作。

G1回收器出现后，ParNew合并入CMS，成为CMS专门处理新生代的回收器。

ParNew在处理**CPU核心多**的情况下性能好。

**并行和并发**：**并行**是指多条垃圾回收线程同时协同工作，此时用户线程处于等待状态。**并发**是指垃圾回收线程和用户线程同时运行，但是垃圾回收线程占用了一部分系统资源，程序吞吐量受到影响。

**5.3 Parallel Scavenge收集器（新生代收集器、标记-复制、吞吐量优先）**

Parallel Scavenge 和ParNew很相似，都是**并行收集、基于标记-复制算法**，但是此收集器目标是达到一个可控制的**吞吐量**，CMS等收集器关注的是缩短垃圾收集时的**停顿时间**。

停顿时间越短越适合**需要与用户交互或需要保证服务响应质量**的程序，提升用户体验。

吞吐量越高越可以**高效利用处理器资源**，尽快完成运算任务，适用于**后台运算不需要太多交互**的分析任务。

Parallel Scavenge 提供了两个参数精确控制吞吐量：

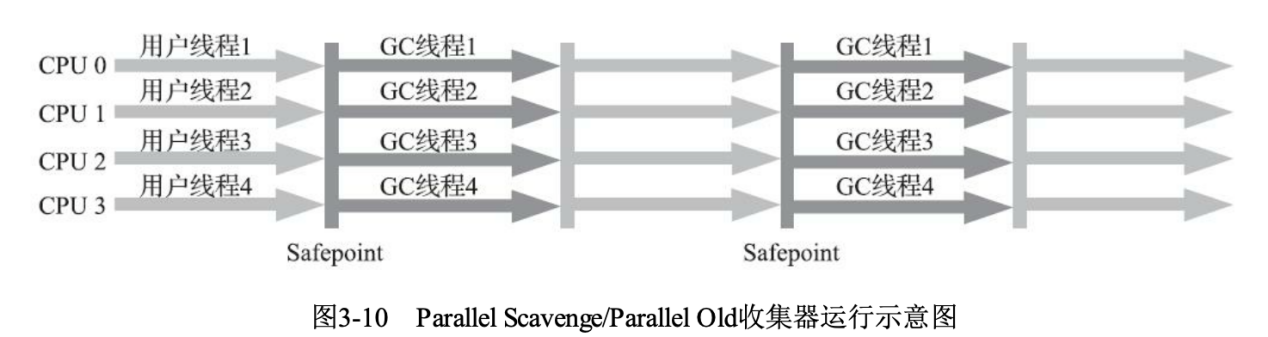
**-XX：MaxGCPauseMills**：尽力保证**内存回收时间**不超过此设定值

但是垃圾收集**停顿时间缩短**是**牺牲吞吐量和新生代空间**作为交换，回收300M新生代肯定比回收500M快，但是相应的垃圾收集更频繁，之前10s收集一次，现在5s收集一次，只不过每次收集间隔时间都缩短了，但是吞吐量下降了。

**-XX：GCTimeRatio**：默认值99，表示用户希望虚拟机消耗在GC上的时间不超过程序运行时间的1/(1+N)。

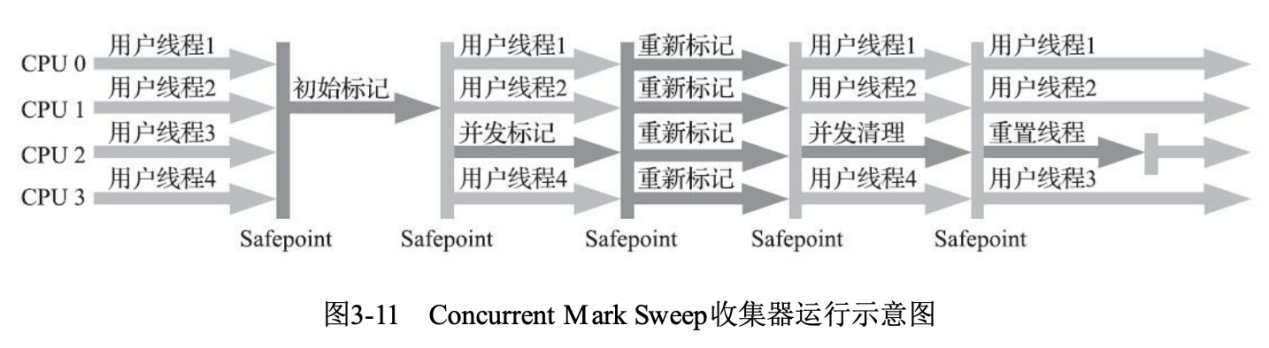
**5.4Serial Old 收集器（单线程、标记-整理）**

用作与Parallel Scavenge搭配使用或者作为CMS收集器失败时的后备预案。

**5.5 Parallel Old收集器（老年代收集器、多线程并行、标记整理、吞吐量优先、配合Parallel Scavenge收集器使用、jdk1.8默认收集器）**

**5.6 CMS收集器（标记-清除、多线程并发收集器、用于老年代、获取最短停顿时间为目标、三次标记+一次清除）**

收集过程：1、**初始标记2、并发标记3、重新标记4、并发清除**



初始标记和重新标记都要**STW**，并发标记和并发清除都是**和用户线程并发进行**。

**初始标记**：只是标记所有GC Roots直接关联到的对象，速度很快。

**并发标记**：从GC Roots的直接关联对象向下遍历标记整个对象图，和用户线程并发进行。

**重新标记**：修正并发标记期间，因用户程序进行导致标记产生变动的那部分对象的标记记录。

**并发清除**：清除掉没有被标记的死亡对象，不需要移动存活对象，所以这个过程和用户线程可以并发进行。

整个回收过程耗时最长的**并发标记和并发清除**是和**用户线程并发进行**的，所以**CMS整体内存回收与用户线程一起并发执行的**。

CMS优点：**并发收集、低停顿**

缺点：在**并发阶段不会停掉用户线程但会占用一部分线程**导致线程处理性能降低，程序变慢，**降低吞吐量**。如果cpu核心数>=4个影响不大，小于四个CMS对程序影响会很大。

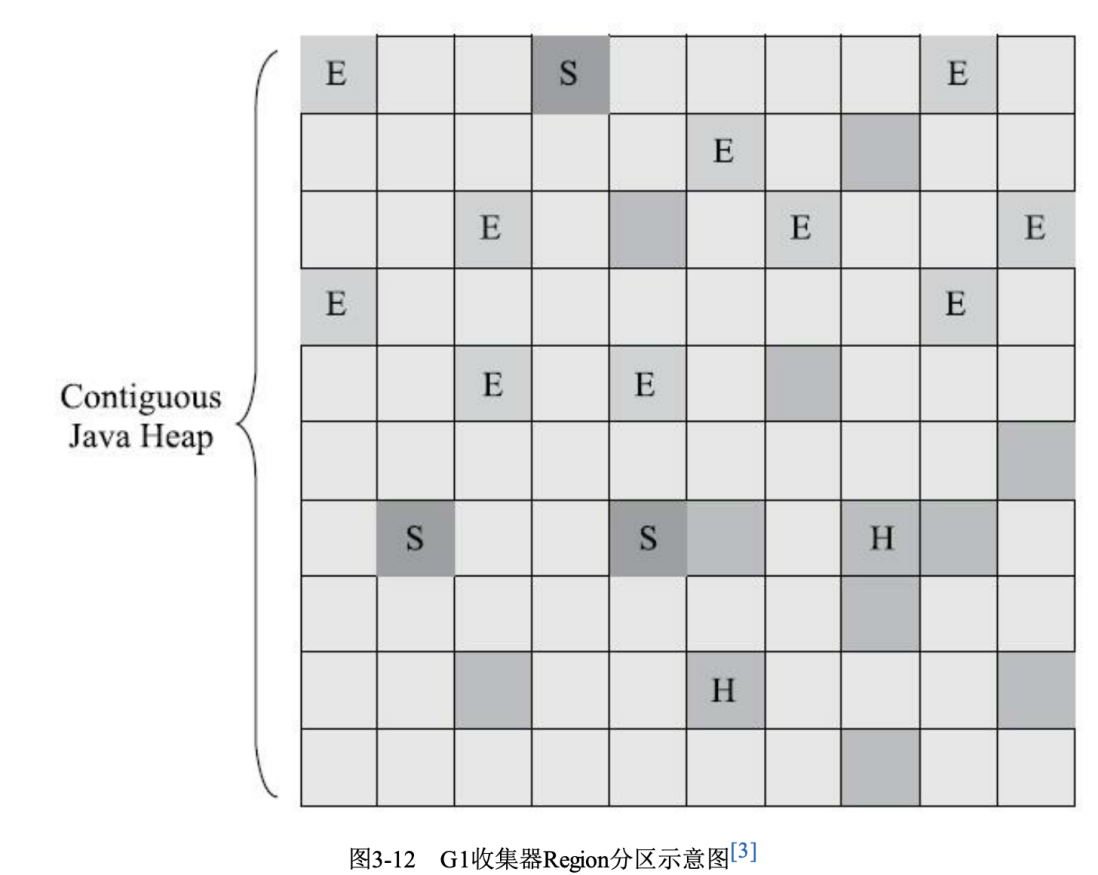
然后，由于CMS收集器**⽆法处理“浮动垃圾**”(FloatingGarbage)，有可能出现“Concurrent Mode Failure”失败进⽽导致另⼀次完全“Stop The World”的Full GC的产⽣。在CMS的并发标记和并发清理阶段，⽤户线程是还在继续运⾏的，程序在运⾏⾃然就还会伴随有新的垃圾对象不断产⽣，但这⼀部分垃圾对象是出现在标记过程结束以后，CMS⽆法在当次收集中处理掉它们，只好留待下⼀次垃圾收集时再清理掉。这⼀部分垃圾就称为“浮动垃圾”。同样也是由于在垃圾收集阶段⽤户线程还需要持续运⾏，那就还需要预留⾜够内存空间提供给⽤户线程使⽤，因此CMS收集器不能像其他收集器那样等待到⽼年代⼏乎完全被填满了再进⾏收集，必须预留⼀部分空间供并发收集时的程序运作使⽤。要是**CMS运⾏期间预留的内存⽆法满⾜程序分配新对象的需要，就会出现⼀次“并发失败”**(Concurrent Mode Failure)，这时候虚拟机将不得不启动后备预案: 冻结⽤户线程的执⾏，**临时启⽤Serial Old收集器来重新进⾏⽼年代的垃圾收集**， 但这**样停顿时间就很⻓了**。所以参数-XX: CMSInitiatingOccupancyFraction设置得太⾼将会很容易导致⼤量的并发失败产⽣，性能反⽽降低，⽤户应在⽣产环境中根据实际应⽤情况来权衡设置。

还有一个缺点就是CMS基于标记清除算法，会**产生内存碎片**，导致有空间但是无法盛放对象从而Full GC。此虚拟机设计者们还提供了另外⼀个参数-XX: CMSFullGCsBefore - Compaction(此参数从JDK 9开始废弃)，这个参数的作⽤是要求CMS收集器在执⾏过若⼲次(数量由参数值决定)不整理空间的Full GC之后，下⼀次进⼊Full GC前会先进⾏碎⽚整理(默认值为0，表示每次进⼊Full GC时都进⾏碎⽚整理)。

**5.7 G1收集器（局部标记-复制，整体标记-整理，JDK9取代Parallel组合成为默认收集器，Mixed GC模式、在延迟可控的情况下尽可能高吞吐量）**

面向服务端应用的垃圾收集器。面向堆内存任何部分组成回收集。哪块内存垃圾数多，回收收益最大就回收哪块内存，而不是回收全部垃圾。

G1把堆划分成**多个大小相等的独立区域Region**，每个区域根据需要扮演**Eden、Survivor或者老年代**空间。还有一类**Humongous区域用来储存大对象**，对象大小超过一个Region一半了就认定为大对象，把Humongous作为老年代看待。

G1每次回收**以Region为最小单位**回收，让G1收集器跟踪每个Region里垃圾堆积的价值大小，在**后台维护优先级列表**，优先处理回收收益最大的Region。

G1解决**跨代引用**：使用**记忆集**，本质上是一种**哈希表**，Key是别的Region起始地址，Value是集合，储存的元素是卡表索引号，这种双向卡表结构比之前的实现起来更复杂，并且由于Region数量比传统收集器分代数量多，因此G1收集器对内存负担更大，G1要耗费大约堆容量10~20%的额外内存。

**在并发标记阶段如何保证收集线程与⽤户线程互不⼲扰地运⾏**？这⾥⾸先要解决的是**⽤户线程改变对象引⽤关系**时，必须**保证其不能打破原本的对象图结构**，CMS收集器采⽤**增量更新算法**实现，⽽G1收集器则是通过**原始快照(SATB)算法**来实现的。

此外，垃圾收集对⽤户线程的影响还体现在回收过程中新创建对象的内存分配上，程序要继续运⾏就肯定会持续有新对象被创建，G1为每⼀个Region设计了**两个名为TAMS(Top at Mark Start)的指针**，**把Region中的⼀部分空间划分出来⽤于并发回收过程中的新对象分配，**并发回收时**新分配的对象地址**都必须要**在这两个指针位置以上**。G1收集器默认在这个地址以上的对象是被隐式标记过的，即默认它们是存活的，不纳⼊回收范围。与CMS中的“Concurrent Mode Failure”失败会导致Full GC类似，如果内存回收的速度赶不上内存分配的速度，G1收集器也要被迫冻结⽤户线程执⾏，导致Full GC⽽产⽣⻓时间“Stop The World”。

如果我们不去计算⽤户线程运⾏过程中的动作(如使⽤写屏障维护记忆集的操作)，**G1收集器的运作过程**⼤致可划分为以下四个步骤:

**初始标记**(Initial Marking): 仅仅只是标记⼀下GC Roots能直接关联到的对象，并且**修改**

**TAMS指针**的值，让下⼀阶段⽤户线程并发运⾏时，能正确地在可⽤的Region中分配新对

象。这个阶段需要停顿线程，但耗时很短，⽽且是借⽤进⾏Minor GC的时候同步完成

的，所以G1收集器在这个阶段实际并没有额外的停顿。

**并发标记(**Concurrent Marking): 从GC Root开始对堆中对象进⾏可达性分析，递归扫

描整个堆⾥的对象图，找出要回收的对象，这阶段耗时较⻓，但可与⽤户程序并发执⾏。

当对象图扫描完成以后，还要**重新处理SATB记录下的在并发时有引⽤变动的对象。**

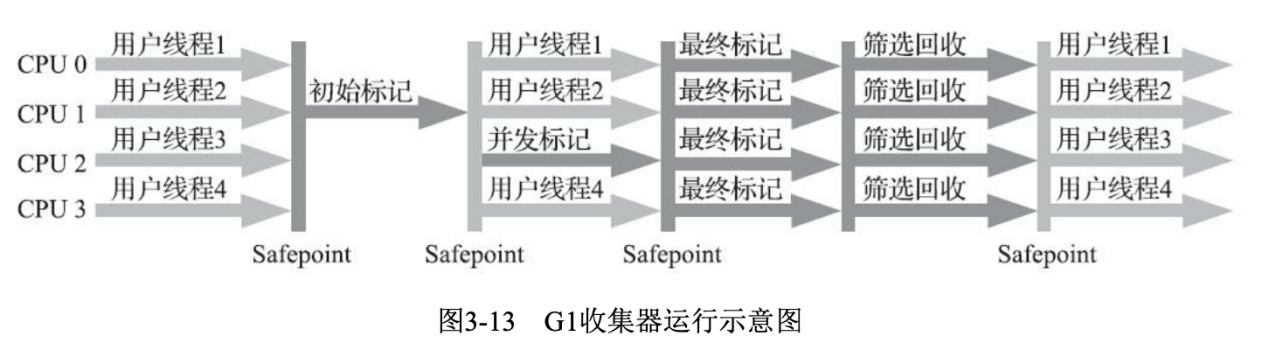
**最终标记**(Final Marking): 对⽤户线程做**另⼀个短暂的暂停**，⽤于**处理并发阶段结束后仍**

**遗留下来的最后那少量的SATB**记录。

**筛选回收**(Live Data Counting and Evacuation): 负责更新Region的统计数据，**对各个**

**Region的回收价值和成本进⾏排序**，**根据⽤户所期望的停顿时间来制定回收计划**，可以

⾃由选择任意多个Region构成回收集，然后**把决定回收的那⼀部分Region的存活对象复制到空的Region中，再清理掉整个旧Region的全部空间**。这⾥的操作涉及存活**对象的移**

**动**，是必须**暂停⽤户线程**，由多条收集器线程**并⾏**完成的。

相⽐CMS，G1的**优点**有很多，暂且不论可以**指定最⼤停顿时间**、**分Region的内存布局**、**按收益动态确定回收集**这些创新性设计带来的红利，单从最传统的算法理论上看，G1也更有发展潜⼒。

与CMS 的“标记-清除”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，但从**局部**(两个Region之间)上看⼜是基于**“标记-复制”**算法实现，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**，垃圾收集完成之后能提供规整的可⽤内存。

⽐起CMS，G1的**弱项**也可以列举出不少，如在⽤户程序运⾏过程中，G1⽆论是为了垃圾收集产⽣的内存占⽤(Footprint)还是程序运⾏时的**额外执⾏负载**(Overload)都要**⽐CMS要⾼**。

**⽬前在⼩内存应⽤上CMS的表现⼤概率仍然要会优于G1，⽽在⼤内存应⽤上G1则⼤多能**

**发挥其优势**，这个优劣势的Java堆容量平衡点通常在6GB⾄8GB之间。

**6垃圾收集器权衡**

衡量收集器的三个指标：**吞吐量、延迟、内存占用**

**应⽤程序的主要关注点是什么?**

**数据分析、科学计算**等需要计算快的任务，选择吞吐量大的。

**SLA应用**和**一些对用户交互页面响应时间有要求的**，选择停顿时间少的，延迟是关注点。

一些**客户端应用**，单核CPU的，垃圾回收的内存占用是不可忽视的。

**运⾏应⽤的基础设施如何?**

譬如硬件规格，要涉及的系统架构是x86-32/64、SPARC还是ARM /Aarch64; 处理器的数量多少，分配内存的⼤⼩; 选择的操作系统是Linux、Solaris还是Windows等。

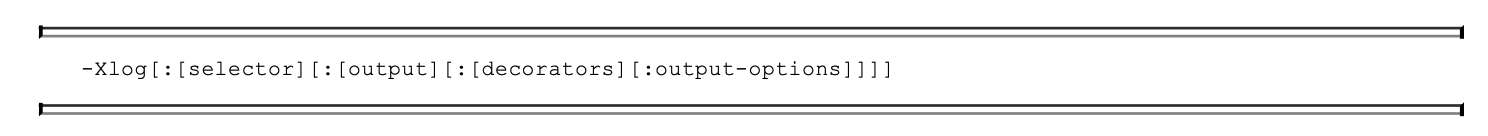
**使⽤JDK的发⾏商是什么? 版本号是多少?**

是ZingJDK/Zulu、OracleJDK、Open-JDK、OpenJ9抑或是其他公司的发⾏版?该JDK对应了《Java虚拟机规范》的哪个版本?

假设某个直接⾯向⽤户提供服务的B/S系统准备选择垃圾收集器，⼀般来说**延迟时间**是这类应⽤的主要关注点。

**7虚拟机及垃圾回收器日志**

HotSpot所有功能的⽇志都收归到了**“-Xlog”**参数上:



命令⾏中最关键的参数是选择器(Selector)，它由**标签(Tag)和⽇志级别(Level)**共同组成。⽇志级别从低到⾼，共有**Trace，Debug，Info，Warning，Error，Of**六种级别，⽇志级别决定了输出信息的详细程度，默认级别为**Info。**

另外，还可以使⽤**修饰器(Decorator)**来要求每⾏⽇志输出都附加上额外的内容，⽀持附

加 在⽇志⾏上的信息包括:

·time:当前⽇期和时间。

**·uptime:虚拟机启动到现在经过的时间，以秒为单位。**

·timemillis:当前时间的毫秒数，相当于System.currentTimeMillis()的输出。

·uptimemillis:虚拟机启动到现在经过的毫秒数。

·timenanos:当前时间的纳秒数，相当于System.nanoTime()的输出。

·uptimenanos:虚拟机启动到现在经过的纳秒数。

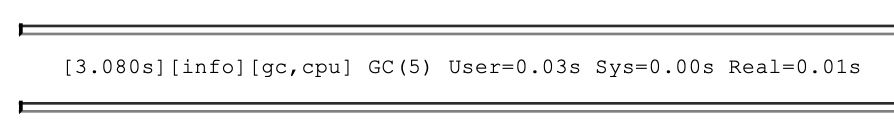
·pid:进程ID。

·tid:线程ID。

**·level:⽇志级别。**

**·tags: ⽇志输出的标签集。**

如果不指定 ，默认值是uptime、level、tags这三个，此时⽇志输出类似于以下形式:



**8内存分配和垃圾回收策略**

**8.1对象优先在Eden区分配**

**当Eden区没有足够空间分配时，虚拟机会发起一次Minor GC**

**8.2大对象直接进入老年代**

大对象是指**需要大量连续内存空间**的Java对象。虚拟机提供**-XX：PretenureSizeThreshold**参数，**指定大于该设置值的对象直接进入老年代**，避免在Eden区和两个Survivor区之间来回复制，产生大量内存复制操作。

**8.3长期存活对象进入老年代**

虚拟机给每个对象设定了一个**年龄Age计数器，储存在对象头**，对象每熬过一次MinorGC并留在Survivor中时，age+1，age增加到15时晋升到老年代，**-XX:MaxTenuringThreshold**可以改变这个默认值15

**8.4动态对象年龄判定**

在survivor空间中**低于或等于某个年龄的所有对象大小总和大于Survivor空间的一半**，大于这个年龄的所有对象就可以直接进入老年代，无序等到age到15.

**8.5空间分配担保（即Minor GC后存活对象太多，只能移动到老年代了）**

在发⽣Minor GC之前，虚拟机必须先检查**⽼年代最⼤可⽤的连续空间是否⼤于新⽣代所有对象总空间**，如果这个条件成⽴，那这⼀次Minor GC可以确保是安全的。如果不成⽴，则虚拟机会先查看**-XX: HandlePromotionFailure**参数的设置值是否**允许担保失败(**Handle Promotion Failure); 如果允许，那会继续**检查⽼年代最⼤可⽤的连续空间是否⼤于历次晋升到⽼年代对象的平均⼤⼩**，**如果⼤于，将尝试进⾏⼀次Minor GC**，尽管这次Minor GC是有⻛险的; **如果⼩于**，或者-XX: HandlePromotionFailure设置不允许冒险，那这时就要改为进⾏⼀次**Full GC**。

新⽣代使⽤复制收集算法，但为了内存利⽤率， 只使⽤其中⼀个Survivor空间来作为轮换备份，因此当**出现⼤量对象在Minor GC后仍然存活的情况**——最极端的情况就是内存回收后新⽣代中所有对象都存活，需要**⽼年代进⾏分配担保**，把Survivor⽆法容纳的对象**直接送⼊⽼年代**，这与⽣活中贷款担保类似。⽼年代要进⾏这样的担保，前提是**⽼年代本身还有容纳这些对象的剩余空间**，但⼀共有多少对象会在这次回收中活下来在实际完成内存回收之前是⽆法明确知道的，所以只能取之前每⼀次回收晋升到⽼年代对象容量的平均⼤⼩作为经验值，与**⽼年代的剩余空间进⾏⽐较**，决定**是否进⾏Full GC**来让⽼年代腾出更多空间。

JDK 6 Update 24之后的规则变为**只要⽼年代的连续空间⼤于新⽣代对象总⼤⼩或者历次晋升的平均⼤⼩，就会进⾏Minor GC，否则将进⾏Full GC**。(不再使⽤-XX: HandlePromotionFailure参数了)

**第4、5章 JVM监控和故障处理及内存调优**

**jvm问题排查和调优**：

**jps**主要⽤来输出JVM中运⾏的进程状态信息。

**jstat**命令可以⽤于持续观察虚拟机内存中各个分区的使⽤率以及GC的统计数据

**jmap可**以⽤来查看堆内存的使⽤详情。

**jstack**可以⽤来查看Java进程内的线程堆栈信息。 jstack是个⾮常好⽤的⼯具，结合应

⽤⽇志可以迅速定位到问题线程。

**Java性能分析⼯具:**

jdk会⾃带JMC(JavaMissionControl)⼯具。可以分析本地应⽤以及连接远程ip使⽤。提

供了实时分析线程、内存，CPU、GC等信息的可视化界⾯。

**JVM内存调优:**

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。

使⽤JDK提供的内存查看⼯具，⽐如JConsole和Java VisualVM。 1）监控GC的状态，使⽤各种JVM⼯具，查看当前⽇志，并且分析当前堆内存快照和gc⽇志，根据实际的情况看是否需要优化。

2）通过JMX的MBean或者Java的jmap⽣成当前的Heap信息，并使⽤Visual VM或者

Eclipse⾃带的Mat分析dump⽂件

3）如果参数设置合理，没有超时⽇志，GC频率GC耗时都不⾼则没有GC优化的必要，如

果GC时间超过1秒或者频繁GC，则必须优化

4）调整GC类型和内存分配，使⽤1台和多台机器进⾏测试，进⾏性能的对⽐。再做修

改，最后通过不断的试验和试错，分析并找到最合适的参数

**调优命令:**

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

• jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

• jstat，JVM statistics Monitoring是⽤于监视虚拟机运⾏时状态信息的命令，它可

以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运⾏数据。

• jmap，JVM Memory Map命令⽤于⽣成heap dump⽂件

• jhat，JVM Heap Analysis Tool命令是与jmap搭配使⽤，⽤来分析jmap⽣成的

dump，jhat内置了⼀个微型的HTTP/HTML服务器，⽣成dump的分析结果后，可

以在浏览器中查看

• jstack，⽤于⽣成java虚拟机当前时刻的线程快照。 • jinfo，JVM Configuration info 这个命令作⽤是实时查看和调整虚拟机运⾏参数。

**调优⼯具:**

常⽤调优⼯具分为两类, jdk⾃带监控⼯具：jconsole 和 jvisualvm，第三⽅有：

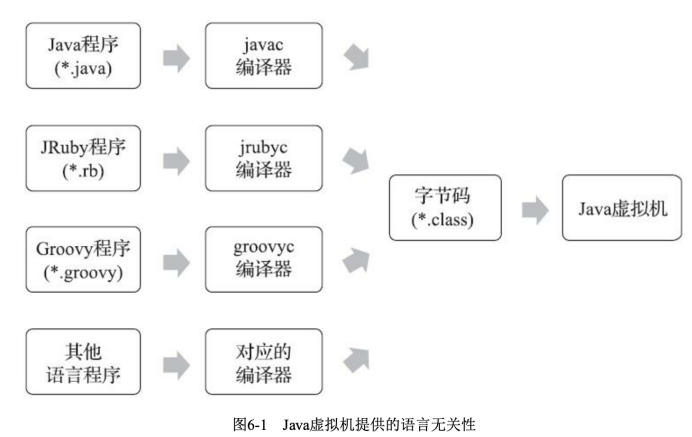
MAT(Memory Analyzer Tool)、GChisto。 • jconsole，Java Monitoring and Management Console是从java5开始，在JDK中

⾃带的java监控和管理控制台，⽤于对JVM中内存，线程和类等的监控

**第6章 类⽂件结构**

**1、概述**

Java中**的各种语法、关键字、常量变量和运算符号**的语义最终会**转换成字节码指令**组合表达出来。

.java文件首先会被编译成**二进制字节码文件**，然后被虚拟机类加载后转换成操作系统识别的机器码运行。

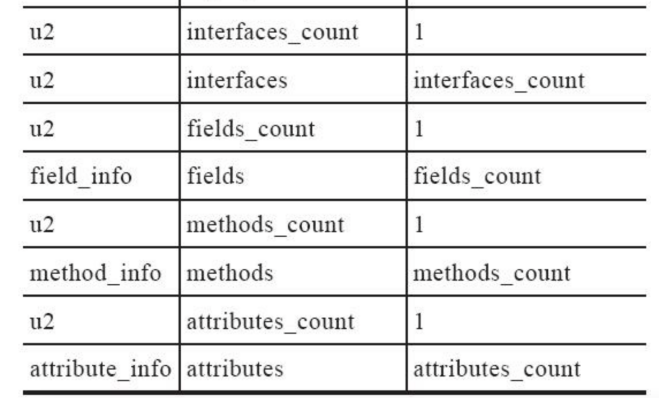
1. **Class类文件结构**

任何一个类文件都对应一个**唯一的接口或者类的定义信息**，但是类或者接口不一定都要定义在文件里，可以**动态生成**直接送入类加载器中，完全不需要以磁盘文件形式存在。

Class文件是**以字节为基础的二进制字节流**，紧凑排列没有任何分隔符。采用一种类似C语言结构体的伪结构储存数据，两种数据类型：**无符号数和表**。

**无符号数**属于基本数据类型：以u1、u2、u4、u8分别代表1/2/4/8字节的无符号数，可以用来描述数字、索引引用、数量值或按照UTF-8编码构成字符串值。

**表**是多个无符号数或其他表作为数据项构成的数据类型，都习惯性以“\_info”结尾，表示一类具有相同层次关系的数据，**比如常量池就是一类表**。每个表之前都会用一个计数器记录这个表中数据的数量。如下是各数据项具体含义：



1. **魔数和Class文件版本**

Class头4个字节称为**魔数**，确定这个文件是否是Class文件被虚拟机接受，Java魔数是0xCAFEBABE（咖啡宝贝）。

紧接着4个字节是**Class文件版本号**：5、6字节是次版本号，7、8字节是主版本号。Jdk1.2到12版本之间没使用过次版本号，固定位0,JDK12之后如果Class文件中使用了该版本JDK尚未列入正式特性清单中的预览功能，次版本号要标识为65535，使虚拟机在加载类文件时能区分出来。

1. **常量池**

接在主次版本号之后的是**常量池入口**，需要放置一项u2类型数据代表常量池的**容量计数值**（constant\_pool\_count）。从1计数而不是从0，是为了后面如果某些指向常量池的索引值的数据在特定情况下需要表达“**不引用任何一个常量池项目**”的含义时，可以把索引值设置为0来表示。

常量池主要放两大类常量：**字面量和符号引用**

**字面量**接近于Java语言层面的**常量**概念：如文本字符串、被声明为final的常量值等。

**符号引用**主要包含以下：

（主要是各种名字，在解析阶段再把名字转成具体地址，即符号引用 -> 直接引用）

被模块导出或者开放的**包(Package)**

**类和接⼝**的全限定名(Fully Qualified Name)

**字段的名称和描述符(Descriptor)**

**⽅法的名称和描述符**

**⽅法句柄和⽅法类型动态调⽤点和动态常量**

**为了保证动态加载**的特性，Class文件不会保存各个方法、字段最终在内存的布局信息，这些方法、字段的符号引用**不经过JVM运行期转换成直接引用无法得到真正的内存入口地址**。当虚拟机做类加载时，会从常量池得到**符号引用**，在**解析**阶段转化为**直接引用**指向对应的内存入口地址。

常量池每一项常量都是一个表，JDK13中有17中不同类型的常量，表结构起始第一位是一个u1类型的**标志位（tag）**，代表当前常量属于哪种常量类型，用于**区分常量类型**。

**5、访问标志:**

在常量池结束之后，紧接着的**2个字节**代表**访问标志(access\_flags)**，这个标志⽤于**识别**

**⼀些类或者接⼝层次的访问信息**，包括: 这个Class是类还是接⼝; 是否定义为public类型;

是否定义为abstract类型; 如果是类的话，是否被声明为final;等等。

**第7章 虚拟机类加载机制**

1、概述

虚拟机把Class文件加载到内存，并对数据进行校验、解析和初始化，形成可以被虚拟机直接使用的Java类型，这个过程称为虚拟机类加载机制。

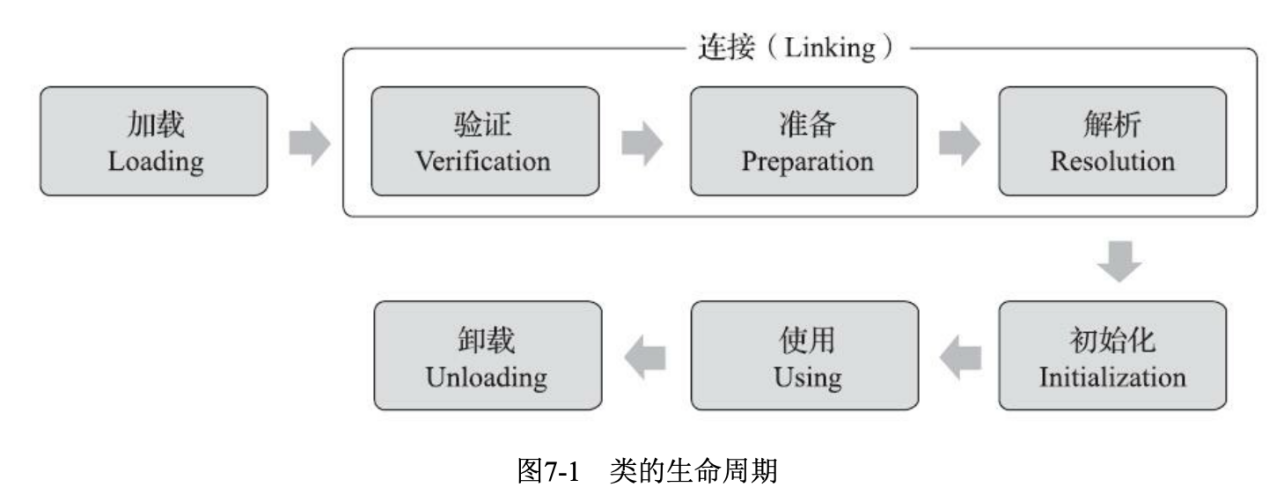
类型加载连接初始化都是在运行期间完成的，会增加性能开销，但是提供了极大地扩展性。依赖运行期动态加载和动态连接实现AOP、反射等功能。用户可以通过自定义类加载器选择从网络上或者其他地方获得Class二进制流字节码作为程序代码一部分。

1. 类加载时机

⼀个类型从被加载到虚拟机内存中开始，到卸载出内存为⽌，它的整个⽣命周期将会经历

**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 ，其中验证、准

备、解析三个部分统称为连接(Linking)。

为了**支持Java语言动态绑定的特性**，**解析阶**段在某些情况下可以在**初始化之后**开始。

严格规定了有且只有六种情况必须立即对类进行“**初始化**”：

1. 遇到**new、getstatic、putstatic或invokestatic**这四条字节码指令时，如果类型没有进⾏过初始化，则需要先触发其初始化阶段。能够⽣成这四条指令的典型Java代码场景有:

使⽤new关键字**实例化对象**的时候。

**读取或设置**⼀个类型的**静态字段**(被final修饰、**已在编译期把结果放⼊常量池的静态字段除外**)**(常量在编译阶段会存⼊调⽤类的常量池中，本质上没有直接引⽤到定义常量的类，**

**因此不会触发定义常量的类的初始化**)的时候。

**调⽤**⼀个类型的**静态⽅法**的时候。

2)使⽤java.lang.reflect包的⽅法对类型进⾏**反射调⽤**的时候，如果类型没有进⾏过初始

化，则需要先触发其初始化。

3)当**初始化类**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JDK 7新加⼊的动态语⾔⽀持时，如果⼀个java.lang.invoke.MethodHandle实例

最后的解 析结果为REF\_getStatic、REF\_p utStatic、REF\_invokeStatic、REF\_newInvokeSpecial四种类型的⽅法句柄，并且这个⽅法句柄对应的类没有进⾏过初始化，则需要先触发其初始化。

6)当⼀个接⼝中定义了JDK 8新加⼊的**默认⽅法(被default关键字修饰的接⼝⽅法)**时，如

果有**这个接⼝的实现类发⽣了初始化，那该接⼝要在其之前被初始化**。

对于这六种会触发类型进⾏初始化的场景，《Java虚拟机规范》中使⽤了⼀个⾮常强烈的限定语 ——“**有且只有**”，这六种场景中的⾏为称为对⼀个类型进⾏**主动引⽤**。除此之外，所有引⽤类型的⽅式都不会触发初始化，称为**被动引⽤**。

对于**静态字段**，只有**直接定义这个字段的类**才会被初始化，因此**通过其⼦类来引⽤⽗类中**

**定义的静态字段**，**只会触发⽗类的初始化⽽不会触发⼦类的初始化**。

**接⼝初始化:**

接口也有初始化过程，但是接口中不能有**static代码块**，编译期仍然会为接口生成**<clinit>()类构造器**，用于**初始化接口中定义的成员变量**。

**接口和类的主要区别**在于前面六种中的一种：**类初始化时要求其父类必须初始化。但是接口初始化不要求其父类接口全部初始化，真正用到父类接口时（如引用接口中定义的常量）才会初始化**。

1. **类加载过程**

**3.1加载**

**1)通过⼀个类的全限定名来获取定义此类的⼆进制字节流。**

**2)将其转化为⽅法区的运⾏时数据结构。**

**3)在内存中⽣成⼀个代表这个类的java.lang.Class对象，作为⽅法区这个类的各种数据的访问⼊⼝。**

相对于类加载过程的其他阶段，**⾮数组类型**的**加载阶段**(准确地说，是**加载阶段中获取类的⼆进制字节流的动作**)是开发⼈员可控性最强的阶段。加载阶段既可以使⽤Java虚拟机⾥内置的**引导类加载器**来完成，也可以由⽤户**⾃定义的类加载器**去完成，开发⼈员通过定义⾃⼰的类加载器去控制字节流的获取⽅式(**重写⼀个类加载器的findClass()或loadClass()⽅法)**，实现根据⾃⼰的想法来赋予应⽤程序获取运⾏代码的动态性。

对于**数组类**⽽⾔，情况就有所不同，数组类本身**不通过类加载器创建**，它是由Java虚拟机**直接在内存中动态构造**出来的。但数组类与类加载器仍然有很密切的关系，因为数组类的**元素类型(Element Type，指的是数组去掉所有维度的类型)**最终还是要靠类加载器来完成加载，⼀个数组类(下⾯简称 为C)创建过程遵循以下规则:

如果数组的**组件类型(Component Type，指的是数组去掉⼀个维度的类型，注意和前⾯**

**的元素类型区分开来)**是**引⽤类型**，那就递归采⽤本节中定义的加载过程去加载这个组件

类型，数组C将被标识在加载该组件类型的类加载器的类名称空间上。

如果数组的**组件类型**不是**引⽤类型(例如int[]数组的组件类型为int)**，Java虚拟机将会把数组C标记为与引导类加载器关联。

**数组类**的**可访问性与它的组件类型的可访问性⼀致**，如果组件类型**不是引⽤类型**，它的数

组类的可访问性将默认为**public**，可被所有的类和接⼝访问到。

**3.2验证**

验证是**确保Class文件的字节流中包含的信息符合约束要求**，不会危害虚拟机安全。

主要验证四部分：**文件格式认证、元数据验证、字节码验证、符号引用验证**。

1.**⽂件格式验证**

第⼀阶段要**验证字节流是否符合Class⽂件格式的规范**，并且能被当前版本的虚拟机处理。这⼀阶段可能包括下⾯这些验证点:

·是否以**魔数**0xCAFEBABE开头。

·**主、次版本**号是否在当前Java虚拟机接受范围之内。

·常量池的常量中是否有不被⽀持的常量类型(**检查常量tag标志**)。

·**指向常量的各种索引值**中是否有指向不存在的常量或不符合类型的常量。

·CONSTANT\_Utf8\_info型的常量中是否**有不符合UTF-8编码**的数据。

·Class⽂件中各个部分及⽂件本身是否有被删除的或附加的其他信息。

第⼀阶段的⽬的是**保证输⼊的字节流能正确地解析并存储于⽅法区之内**，这阶段的验证**基于⼆进制字节流**进⾏的，**后⾯的三个验证阶段**全部是**基于⽅法区的存储结构**上进⾏的，**不会再直接读取、操作字节流了**。

2.**元数据验证**

第⼆阶段是对**字节码描述的信息进⾏语义分析**，以保证其描述的信息符合《Java语⾔规范》的要求，这个阶段可能包括的验证点如下:

·这个类**是否有⽗类**(除了java.lang.Object之外，所有的类都应当有⽗类)。

·这个类的**⽗类是否继承了不允许被继承的类**(被final修饰的类)。

·如果这个类不是抽象类，**是否实现了其⽗类或接⼝之中要求实现的所有⽅法**。

·类中的**字段、⽅法是否与⽗类产⽣⽭盾**(例如覆盖了⽗类的final字段，或者出现不符合规

则的⽅法重载，例如⽅法参数都⼀致，但返回值类型却不同等)。

第⼆阶段的主要⽬的是**对类的元数据信息进⾏语义校验**，保证不存在与《Java语⾔规

范》定义相悖的元数据信息。

**3.字节码验证**

第三阶段是整个验证过程中最复杂的⼀个阶段，主要⽬的是**通过数据流分析和控制流分析，确定程序语义是合法的、符合逻辑的**。在第⼆阶段对元数据信息中的数据类型校验完毕以后，这阶段就要对**类的⽅法体(Class⽂件中的Code属性)**进⾏校验分析，保证被校验类的⽅法在运⾏时不会做出危害虚拟机安全的⾏为，例如:

·**保证任意时刻操作数栈的数据类型与指令代码序列都能配合⼯作**，例如不会出现类似于

“在操作栈放置了⼀个int类型的数据，使⽤时却按long类型来加载⼊本地变量表中”这样的情况。

·**保证任何跳转指令都不会跳转到⽅法体以外的字节码指令上**。

·**保证⽅法体中的类型转换总是有效的，**例如可以把⼀个⼦类对象赋值给⽗类数据类型，这是安全的，但是把⽗类对象赋值给⼦类数据类型，甚⾄把对象赋值给与它毫⽆继承关系、完全不相⼲的⼀个数据类型，则是危险和不合法的。

**4.符号引⽤验证**

最后⼀个阶段的校验⾏为**发⽣在虚拟机将符号引⽤转化为直接引⽤的时候**，这个转化动作将在连接的第三阶段——**解析阶段中发⽣**。符号引⽤验证可以看作是对类⾃身以外(常量池中的各种符号引⽤)的各类信息进⾏匹配性校验，通俗来说就是，**该类是否缺少或者被禁⽌访问它依赖的某些外部 类、⽅法、字段等资源**。本阶段通常需要校验下列内容:

·符号引⽤中通过字符串描述的全限定名**是否能找到对应的类**。

·在指定类中**是否存在符合⽅法的字段描述符及简单名称所描述的⽅法和字段**。

·符号引⽤中的**类、字段、⽅法的可访问性**(private、protected、public、<package>)是否

可被当前类访问。

**3.3准备**

正式**为类中定义的变量（静态变量，被static修饰的变量）分配内存并设置类变量初始值阶段**。这个阶段**不包括实例变量**，实例变量随着**new实例化**分配在堆中。

通常情况是为静态变量设置为0值，但是如果用**final**定义了一个值，会在此时直接赋值。public static final int value = 123;

**3.4解析**

**虚拟机将常量池的符号引用替换为直接引用**。

**符号引用**：用符号来描述目标的引用，符号引用的目标**不一定已被加载**

**直接引用**：指向目标的指针，直接引用的目标**一定已被加载到内存**

如果符号引用指向一个未被加载的类，或者未被加载类的字段或方法，那么**解析将触发这个类的加载**（但未必触发这个类的链接以及初始化）

**3.5初始化（为静态变量按照程序员只管指定的代码来赋值）**

**初始化阶段虚拟机才真正开始执行类中编写的Java代码**

**为静态变量按照程序员只管指定的代码来赋值**或者说**初始化阶段就是执行类构造器<clinit>()方法的过程。**

<clinit>()是Java编译器的自动生成物，<clinit>()方法是编译器收集了**类中的所有静态变量赋值动作和静态代码块中的语句**合并成的，先后顺序就是行号

JVM会通过**加锁**来确保类构造方法<clinit>()方法仅被执行一次

虚拟机保证子类<clinit>()方法执行前，父类的<clinit>()已经执行完毕。

<clinit>()⽅法与类的构造函数(即在虚拟机视⻆中的实例构造器<init>()⽅法)不同，它不需要显式地调⽤⽗类构造器，Java虚拟机会保证**在⼦类的<clinit>()⽅法执⾏前，⽗类的<clinit>()⽅法已经执⾏完毕**。因此在Java虚拟机中第⼀个被执⾏的<clinit>()⽅法的类型肯定是**java.lang.Object**。

·<clinit>()⽅法对于类或接⼝来说并不是必需的，如果⼀个类中**没有静态语句块**，也**没有对变量的赋值操作**，那么编译器可以不为这个类⽣成<clinit>()⽅法。

**接⼝**中不能使⽤静态语句块，但仍**然有变量初始化的赋值操**作，因此接⼝与类⼀样都会⽣成 <clinit>()⽅法。但接⼝与类不同的是，执⾏**接⼝的<clinit>()⽅法不需要先执⾏⽗接⼝的<clinit>()⽅法**， 因为只有当⽗接⼝中定义的变量被使⽤时，⽗接⼝才会被初始化。此外，**接⼝的实现类在初始化时也⼀样不会执⾏接⼝的<clinit>()⽅法**。

**同⼀个类加载器下，⼀个类型只会被初始化⼀次。**

**4类加载器**

虚拟机团队把加载阶段的“**通过一个类的全限定名获取描述该类的二进制字节流**”这个动作放到**虚拟机外部**实现，让程序决定如何获取所需的类，**实现这个动作的代码称为类加载器**。

任何一个类都需要由**类本身和加载这个类的类加载器共同决定**其在虚拟机中的**唯一性**，相同的Class文件被不同的类加载器加载，这就不是相同的类。

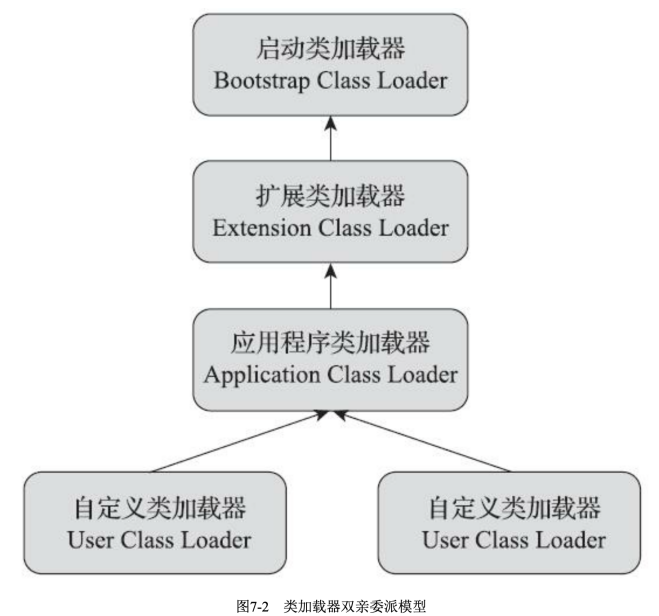
**4.1双亲委派模型：**

Java一直保持着**三层类加载器+双亲委派模型**。

**启动类加载器（Bootstrap Class Loader）**：负责**加载**存放在**<JAVA\_HOME>\lib**目录，或被-Xbootclasspath参数指定的路径存放的类，将这些类加载进内存中。C++实现的顶层类加载器。

**扩展类加载器（Extension Class Loader）**：负责**加载<JAVA\_HOME>\lib\ext目录**中，或被java.ext.dirs系统变量所指定的路径中所有的类库。 用户可以将具有通用性的类库放在ext目录下扩展Java SE功能。

**应用类加载器（Application Class Loader）**：负责**加载用户类路径（ClassPath）上所有类库**，应用程序中没有自定义类加载器，**默认使用这个应用类加载器**。

**自定义类加载器**：由用户定义，可以在自定义中加载其他的Class文件来源。

**双亲委派工作流程**：类加载器收到加载请求后，从**下向上**一层层把**请求委派**给父类加载器，再**从上向下**依次**尝试加载**，父类加载器无法加载时再一层层向下由子类加载器尝试加载。

**双亲委派优点**：系统中定义的Class文件通过双亲委派可以保证加载Class文件的类加载器不会改变，保证加载后是同一个类。**保证了Java程序的稳定和安全**。

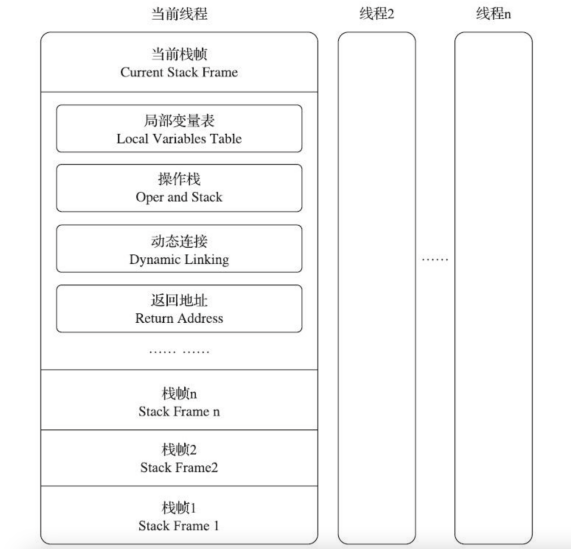
**第8章 虚拟机字节码执⾏引擎**

**1.概述**

执行引擎执行字节码时，会有**解释执行**（通过解释器执行）和**编译执行**（通过即时编译器产生本地代码执行）两种执行方式。所有执行引擎输入输出都是一致的：输入是**字节码二进制流**，处理过程是字节码解析执行的等效过程，输出是执行结果（**机器码**）。

1. **运行时栈帧结构**

栈帧是虚拟机栈**进行方法调用**的数据结构，栈帧储存**局部变量表、操作数栈、动态连接和方法返回地址**等信息。一个方法调用到结束，对应着一个栈帧的**入栈到出栈**。

栈帧所需**局部变量表的大小、操作数栈的深度在编译期就决定了**，并写入了**方法区的Code属性**中。一个栈帧需要分配多少内存不会受到程序运行期变量数据影响。

1. **局部变量表（存放方法参数和内部局部变量）**

用于**存放方法参数和方法内部定义的局部变量**。在编译期就在方法的Code属性的max\_locals数据项中确定了该方法需要分配的局部变量表的**最大容量**。

局部变量表以**变量槽**为最小单位，每个变量槽都能盛放一个**boolean、byte、char、short、int、float、reference或returnAddress**类型的数据。这8种类型最大使用**32位物理内存**储存，变量槽可以盛放不超过32位储存空间的类型数据，其大小可以**根据数据类型大小改变**，不需要固定为32位。

**Reference**表示对一个对象实例的引用，通过这个引用：

1. 根据引用直接或间接的**查找**到对象**在Java堆**中数据存放**的起始地址**或索引。
2. 根据引用直接或间接地**查找**到对象所属数据类型**在方法区中储存的类型信息**。

对于**64位数据类型的long和double**，会使用**两个连续的变量槽，**并且在访问时**不允许分开访问**，否则会抛出异常。

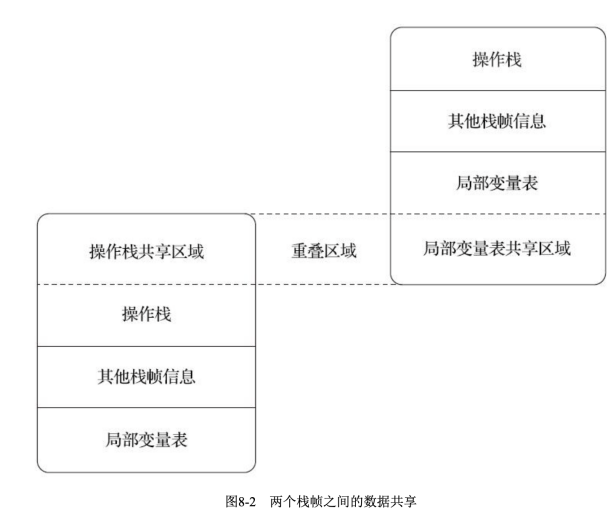
一个**方法调用**时，会生成一个**栈帧**，栈帧中局部变量表会完成**实参到形参的传递**，即**为方法参数赋值**。

为了**节省栈帧消耗的内存空间**，变量槽是**可复用的**，如果PC计数器中的值**超出了某个变量的作用域**，那么储存这个变量的变量槽可以**复用盛放其他变量**。如果一个局部变量定义了但是没赋初始值，是不能使用的，这就是局部变量必须要赋值的原因。

1. **操作数栈**

先进后出的栈，和局部变量表一样编译期就确定了其最大深度。用来**做方法中的算术运算或者调用其他方法时通过操作数栈进行参数传递。**

虚拟机优化了**两个栈帧之间会有重叠，**让下面栈帧部分操作数栈和上面栈帧部分局部变量表重叠，**节约空间**同时在方法调用时可以直接使用操作数栈的数据作为上面栈帧局部变量表的值，**无序额外的参数复制传递**。



1. **动态连接**

用来**指向运行时常量池该栈帧所属方法的引用**，从而支持方法调用过程中的动态连接。在常量池存在大量符号引用，方法调用指令就是以常量池里指向方法的符号引用作为参数，这些符号引用**一部分会在类加载阶段变为直接引用（静态解析）**，**一部分在运行期间转换为直接引用（动态连接）**。

1. **方法返回地址**

方法执行完成后，**返回值会传给上一个栈帧**，此栈帧出栈，**正常调用结束**。

或者**方法执行时出现异常**，**不会给上层栈帧返回值**。

**PC计数器的值作为返回地址**

退出时可能执行的操作：恢复上层方法的局部变量表和操作数栈，把返回值压入调用者栈帧的操作数栈中，调整PC计数器的值以指向方法调用指令后面的一条指令等。

**8.2方法调用**

方法调用**不是执行方法内部代码**，其唯一作用就是**确定调用哪个方法**。方法调用的**目标方法**在Class文件里储存的只是常量池中**的符号引用**，而不是直接引用，**提高了Java动态扩展能力**。

1. **解析**

**把常量池中方法的符号引用的一部分转化成直接引用**，调用这部分方法时在编译阶段就确定下来了，是不会动态扩展的部分。

能在编译阶段就确定下来不会改变的，只有**静态方法和私有方法**了，不可能通过继承或者其他方式重写这两种方法，因此可以编译阶段就可确定。

在Java虚拟机⽀持以下5条⽅法调⽤字节码指令，分别是:

**invokestatic**。⽤于**调⽤静态⽅法**。

**invokespecial**。⽤于**调⽤实例构造器<init>()⽅法、私有⽅法和⽗类中的⽅法**。

**invokevirtual**。⽤于**调⽤所有的虚⽅法。**

**invokeinterface**。⽤于调⽤接⼝⽅法，会在运⾏时再确定⼀个实现该接⼝的对象。

invokedynamic。先在运⾏时动态解析出调⽤点限定符所引⽤的⽅法，然后再执⾏该⽅法。前⾯4条调⽤指令，分派逻辑都固化在Java虚拟机内部，⽽invokedynamic指令的分派逻辑是由⽤户设定的引导⽅法来决定的。

从指令可以看出，invokestatic和invokespecial都可在解析阶段确定唯一调用版本，符合条件的是**静态方法、私有方法、实例构造器、父类方法**四种，加上**被final修饰的方法**（被invokevirtual调用），这五类方法**在类加载的解析阶段把符号引用直接解析为直接引用**。统称为**非虚方法**。

**2.分派（最典型的表现形式：重载）**

2.1**静态分派**

如Human man = new Man();

**Human是静态类型，Man是实际类型，是运行时才能确定的。**

虚拟机在**重载时是通过参数的静态类型作为判断依据的**。所有依赖静态类型决定方法执行版本的分派动作称为**静态分派**。

**2.2动态分派（典型表现：多态的重写）**

由**invokevirtual**调用的是**虚方法**。

1. 调用**invokevirtual**指令时首先会找到**操作数栈顶第一个元素所指向对象**的**实际类型**C
2. 如果在C中找到了与常量中描述符和简单名称都相同的方法，进行**权限校验**，都**通过**了就**返回这个方法的直接引用**。

3、否则，按照继承关系**从下往上依次对C的各个父类搜索验证**。

4、如果没找到合适的方法，抛出AbstractMethodError异常。

**重写的本质就是根据方法接收者的实际类型选择方法版本。**这种在运行期根据实际类型确定方法执行版本的分派过程称为**动态分派**。