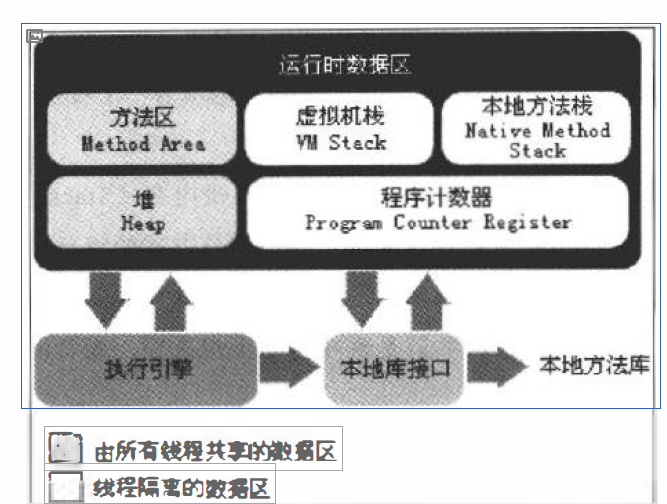
# Jvm相关知识

## Java内存区域与内存溢出异常

### 运行时数据区域



#### 程序计数器

程序计数器是一块较小的内存空间， 它可以毛作是当前线 程所执行的字节码的行号指示器。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。如果线程正在执行的是一个Java方法， 这个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， 这个计数器伯则为空(Undefined)此内存区域 是唯一一个在Jvm虚拟机规范中没有规定任何OutOfMemoryError情况的区域。JVM中多线程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。即在任何时刻，CPU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，各线程之间计数器互不影响，独立存储，该内存为线程私有。

#### Java 虚拟机栈

与程序计数器一样， Jvm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Jvm方法执行的内存模型，每个方法在执行的同时都会创建一个栈帧用于存储局部变量表、 操作数栈、 动态链接、 方法出口等信息。

局部变量表存放了编译期可知的各种基本数据类型，对象引用类型，局部变批表所需的内存空间在编译期间完成分配， 当进人一个方法时． 这个方法需要在帧中分配多大的局部变量空间是完全确定的， 在方法运行期间不会改变局部变量表的大小．。

在jvm虚拟机规范中， 对这个区域规定了两种异常状况, 如果线程请求的栈探度大于虚 拟机所允许的深度， 将抛出StackOverflowError异常：如果虚拟机栈可以动态扩展（当前大部分的jvm, 虚拟机都可动态扩展，只不过Jvm虚拟机中也允许固定长度的虚拟机栈）， 如果扩展时无法申请到足够的内存， 就会t11出OutOfMemoryError异常。

#### 本地方法栈

本地方法栈(Native Method Stock)与虚拟机栈所发挥的作用是非常相似的， 它们之间的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， 而本地方法栈则为 虚拟机使用到的Native方法服务。

#### Java堆

Java堆是被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都在这里分配内存. 这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展与逃逸分析枝术逐渐成熟，栈上分配、 标量替换优化技术将会导致一些微妙的变化发生， 所有的对象都分配在堆上也渐渐变得不那么 “绝对“ 了．

如果在堆中没存内存完成实例分配, 并且堆也无法再扩展时， 将会抛出OutOfMemoryError异常．

#### 方法区

方法区(Method'Area)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。根据Jvm虚拟机规范的规定， 当方法区无法满足内存分配需求时， 将抛出OutOfMemoryError异常

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， 还有一项信息是常量池, 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进人方法区的运行时常量池中存放。

运行时常量池相对于Class文件常量池的另外一个须要特征是具备动态性， Java并不要求常量一定只有编译期才能产生， 也就是并非预置入Class文件中常量池的内容才能进人方法区运行时常量池， 运行期间也可能将新的常量放人池中， 这种特性被开发人员利用得比较多的便是String类的intern()方法．

### 对象的创建、布局与访问

虚拟机遇到一条new指令时， 先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用， 并且检查这个符号引用代表的类是否被加载、 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。

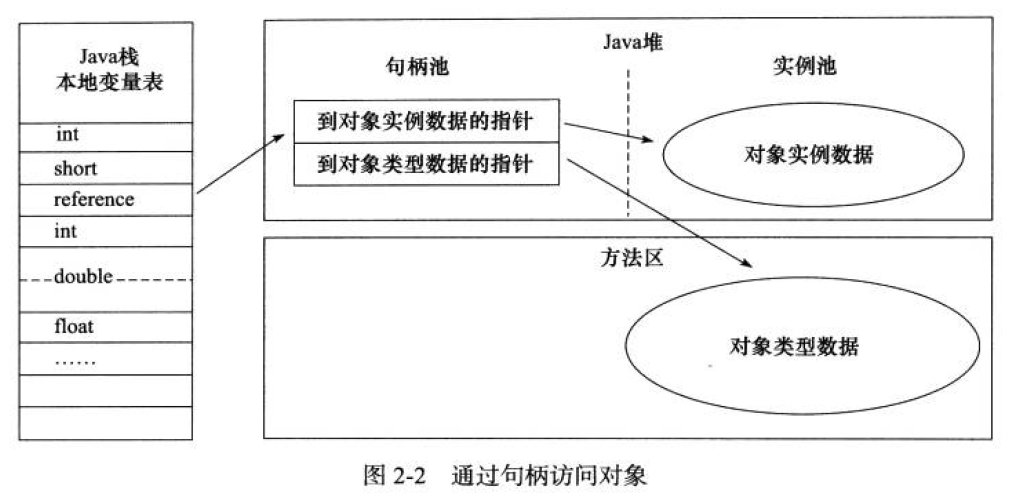
在类加载检查通过后， 接下来虚拟机将为新生对象分配内存． 对象所需内存的大小在类加载完成后便可完全确定。假设Java堆中内存是绝对规整的，所有用过的内存都放在一边， 空闲的内存放在另一边， 中间一个指针作为分界点的指示器， 那所 分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离， 这种分配方式称为 “指针碰撞"；如果Java堆中的内存并不是规整的， 已使用的内存和空闲的内存相互交错，虚拟机就必须维护一个列表 记录上哪些内存块是可用的， 在分配的时候从列表中找到一块足够大的空间划分给对象，这种分配方式称为 “空闲列表"。

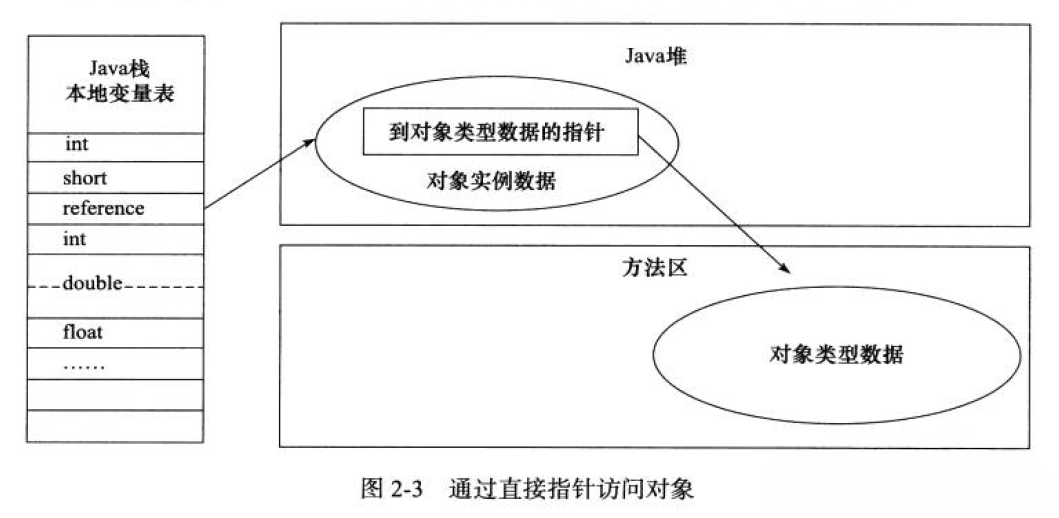
除如何划分可用空间之外， 还有另外一个需要为虑的问题是对象创建在虚拟机中是非常频繁的行为， 仅仅修改一个指针所指向的位置， 在并发情况下也井不是线程安全的， 可能出现正在给对象A分配内存， 指针还没来的及修改， 对象B又同时使用了原来的指针来分配内存。解决方案有两种：一种是对分配内存空间的动作进行同步处理,实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性：另一种是把内存 分配的动作按照线程划分在不同的空间之中进行， 即每个线程在Java,堆中预先分配'---小块内存， 称为本地线程分配缓冲。

内存分配完成后，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、 对象的哈希时、对象的GC分代年龄等信息． 这些信息存放在对象的对象头之中．

在HotS pot虚拟机中 ， 对象在内存中存储的布局可以分为3块区域．对象头(Headed. 实例数据(instance Data)和对齐填充(Padding)。

Hotspot虚拟机的对象头包括两部分信息，第一部分用干存储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。对象头的另外一部分是类型指针， 即对象指向它的类元数据的指针， 虚拟机通过这个指 针来确定这个对象是哪个类的实例。





句柄稳定，直接指针速度快。

## 垃圾回收与内存分配

### 对象已死吗

在堆里面存放若Java世界中几乎所有的对象实例，垃圾收集器在对堆进行回收前，第一 件事情就是要确定这些对象之中哪些还 “存活 ＇， 若， 哪些巳经 “死去“。

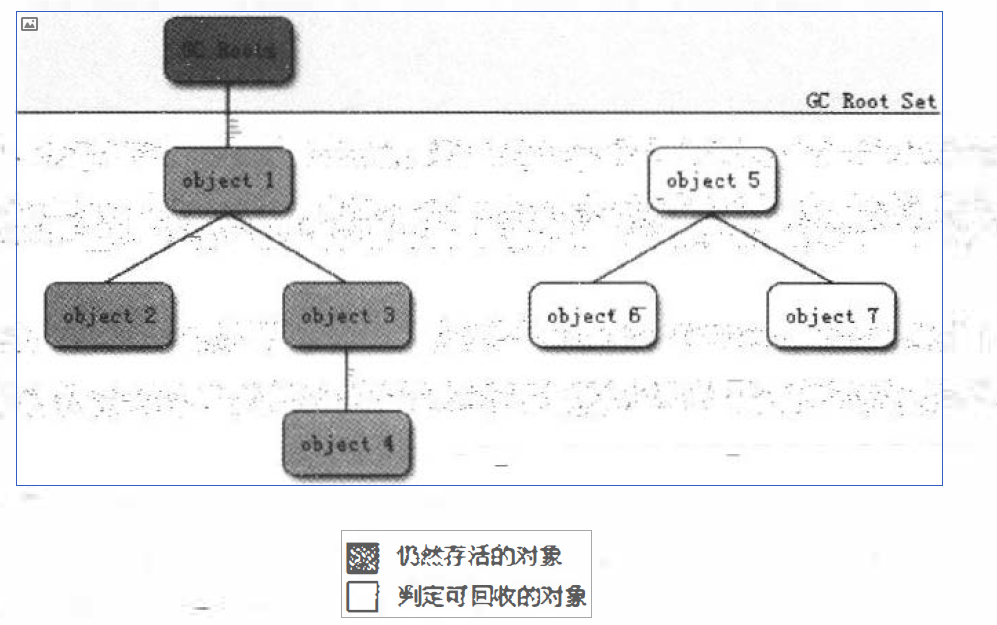
#### 引用计数算法

给对象添加一个引用计数器，每当有一个地方引用他时，计数器值加一，当引用失效时，计数器值减一，任何时刻计数器为0的对象就是不能再使用的。

不能解决循环引用的问题。

#### 可达性分析算法

这个算法的基本思路就是通过一系列的称为"GC Roo!Se"的对象作为起始点， 从这些节点开始向下搜索， 搜索所走过的路径称为引用链, 当一个对象到GC Roots没有任何引用链相连（用图论的话来说， 就是从􀁜GC Roots到这个对象不可达）时， 则证明此对象是不可用的。



在Java中 可作为GC Roots的对象包括下面几种：

1.虚拟机栈（栈帧中的本地变扯表）中引用的对象．

2.方法区中类静态属性引用的对象

3.方法区中常量引用的对象．

4.本地方法栈中JNI (即一般说的Native方法）引用的对象

#### 引用类型

1.强引用就是指在程序代码之中普遍存在的， 类似"Object obj = new Object（）"这类的

引用， 只要强引用还存在垃圾收集器永远不会回收掉被引用的对象．

2. 软引用是用来描述一些还有用但并非必需的对象． 对于软引用关联的对象， 在系统将要发生内存溢出异常之前， 将会把这些对象列进回收范围之中进行第二次回收．如果这次回收还没有足够的内存， 才会抛出内存溢出兄常．

3. 弱引用也是用来描述非必需对象的， 但是它的强度比软引用更弱一些， 被弱引用关联的对象只能生存到下一次垃圾收集发生之前． 当垃圾收渠器工作时， 无论当 前内存是否足够， 都会回收掉只被弱引用关联的对象．

4. 虚引用也称为幽灵引用或者幻影引用， 它是最弱的一种引用关系。 一个对象是否有虚引用的存在， 完全不会对其生存时间构成影响， 也无法通过也引用来取得一个对象实 例． 为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知．

#### 生存与死亡

即使在可达性分析竹法中不可达的对象， 也并非是 “非死不可＂ 的， 这时候它们暂时处于缓刑阶段），真正宜告一个对象死亡， 至少要经历两次标记过程：如果对象在进行可达性分析后没有发现与gc roots链接的引用链，那它将会被笫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耍执行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-Quouo的队列之中， 井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．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-Quouo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，否则就真的被回收了。

#### 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。回收废弃常量与回收 ""堆中的对象非常类似． 以常量池中字面量的回收为例， 假如一个字符串“abc” 已经进入了常量池中． 但是当前系统没有任何一个String对象是叫做"abc” 的， 换句话说， 就是没 有任何String对象引用常量池中的"abc"常量， 也没有其他地方引用了这个字面量， 如果这时发生内存回收， 而且必要的话 这个"abc“ 常量会被清理出常量池。

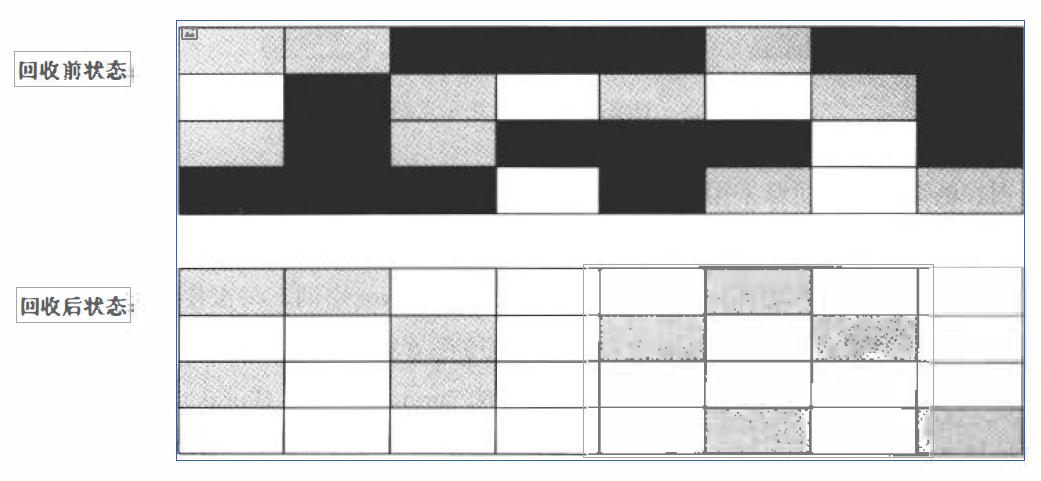
判定无用的类：

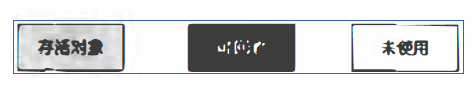
1. 该类所有的实例都巳经被回收， 也就是Java堆中不存在该类的任何实例．
2. 加载该类的ClassLoader已经被回收
3. 该类对应的Java.long.Class, 对象没有在任何地方被引用． 无法在任何地方通过反射访问该类的方法．

### 垃圾收集算法

#### 标记－清除算法

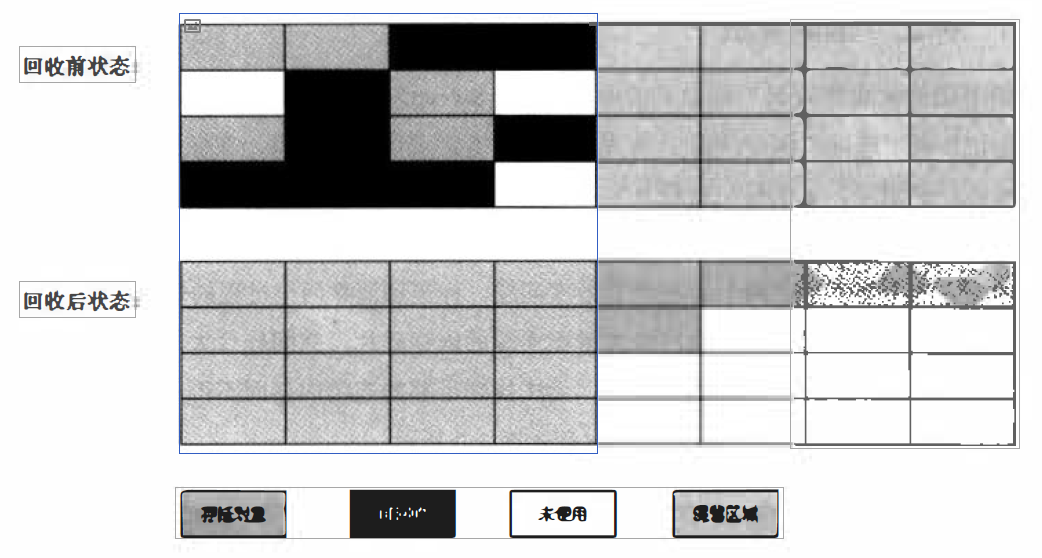
算法分 标记” 和 “清除“ 2个阶段，首先标记出所有需安回收的对象， 在标记完成后统一回收。它的标记过程在前一节讲述对象标记判定时已经介绍过了。它的主要不足有两个， 一个是效率问题， 标记和清除两个过程的效率都不高；另一个是空间问题， 标记清除之后会产生大量不连续的内存碎片， 空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时， 无法找到足够的连续内存而不得不提前触发另一次 垃圾收集动作．





#### 复制算法

将可用内存按容量划分为大小相等的两块．每次只使用其中的一块．当这一块的内存用完了，就将还存活的对象复制到另外一块上面．然后再把已使用过的内存空间一次清理掉．这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可．实现简单，运行高效．只是这种算法的代价是将内存缩小为了 原来的一半，未免太高了一点．

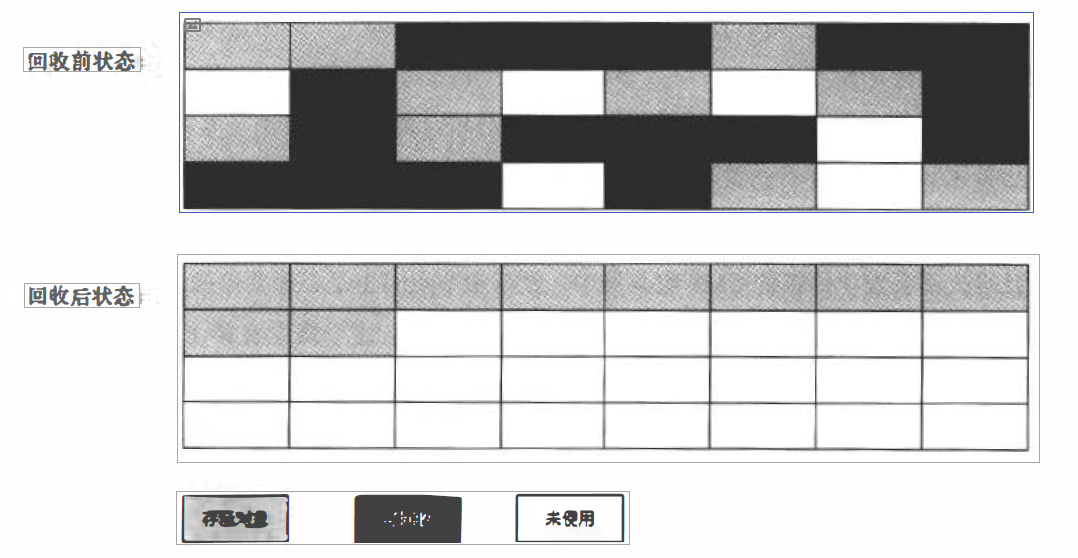


现在的商业虚拟机都采用这种收集算法来回收新生代。将内存分为一块较大的Eden空间和两块较小的Survivor, 空间， 每次使用Eden和其中一 块Survivor。当回收时， 将Eden和Survivor,中还存活的对象一次性地复制到另外一块Survivor, 空间上， 最后清理悼Eden和刚才用过的Survivor, 空间. HotSpot虚拟机默认Eden 和Survivor, 的大小比例8：1. ,

#### 标记－整理算法

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低，更关键的 是如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所打对象都存活的极端情况，所以在老年代一般不能立接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理“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，而是让所打存活的对象都向一端移动，然后立搂清理掉端边界以外的内存。



当前商业虚拟机的垃圾收集机制采用 “分代收集" (Genm1>onal Collecuon)算法． 这种算法没有什么新的思想． 只是根据对象存活周期的不同将内存划分为儿块． 一般是把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还当的收集方法． 在新生代中每次垃圾收集时都发现有大批对象死去， 只有少数存活， 那就选用复制算法， 只需要付出少让存活对象的复利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"标记一清理” 或 “标记一整理"法来进行回收。

### Hotspot算法实现

#### 枚举根节点

可达性分析对执行时问的敏感体现在GC停频上，因为这项分析工作必须在一 个能确保一致性的快照中进行，不可以出现分析过程中对象引用关系还在不断变化的情况， 该点不满足的话分析结果准确性就无法得到保证。这点是并致GC进行时必须停校所有 Java执行线程(Sun将这件事情称为"Stop-The-World")的其中一个重要原因。， 所以执行系统停顿下来后，并不需耍一个不漏地检查在完所有执行上下文和全局引用位置。使用OopMap来达到这个目的。在类加载完成后，就会在特点位置记录下栈和寄存器哪些位置事引用。

#### 安全点

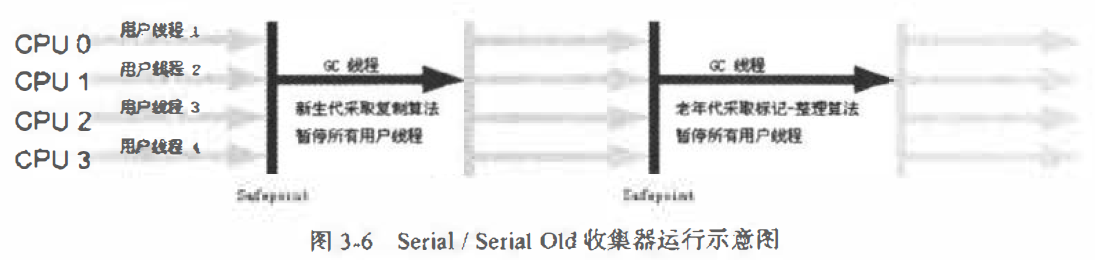
OopMap内容变化的指令非常多， 如果为每一条指令都生成对应的OopMap， 那将会需要大量的额外空间， 这样GC的空间成本将会变得很高。HotSpot也的确没有为每条指令都生成OopMap, 前面已经提到， 只是在 “特定的位置＂ 记载了这些信息， 这些位置称为安全点, 即程序执行时并非在所有地方都能停预下来开始GC, 只有在到达安全点时才能暂停。安全点的选定既不能太少以致于让GC等待时间太长， 也不能过于频繁以致于过分增大运行时的负荷．

### 垃圾收集器

#### Serial 与Serial Old收集器

这个收集器是一个单线程的收菜器， 但它的 "单线程 ” 的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作， 更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

Serial Old是Serial收集器的老年代版本 ， 它同祥是一个单线程收织器 ， 使用“标记一 整理 ” 算法．



#### ParNew收集器

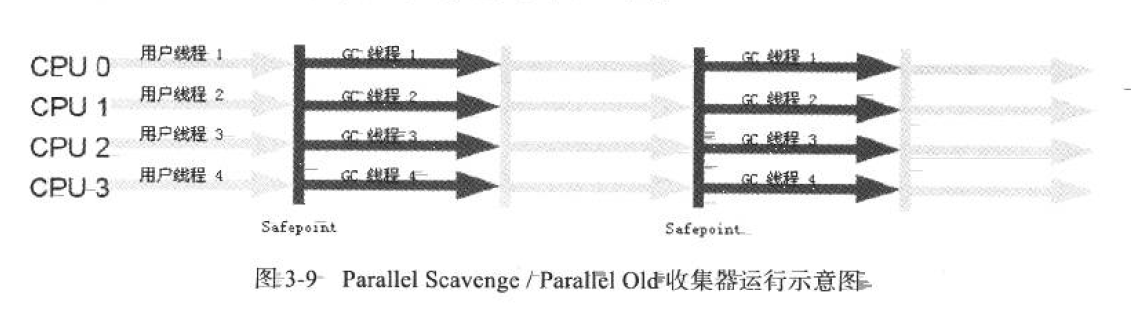
ParNew收集器其实就是serial收集器的多线程版本。使用多条线程进行垃圾收集。



#### Parallel Scavenge与parallel old收集器

Parallel Scavenge收集器是一个新生代收集器， 它也是使用复制算法的收集器． 又是并行的多线程收集器，特别之处在于，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间， 而Parallel Scavenge, 收集器的目标则是达到一个可控割的吞吐量，

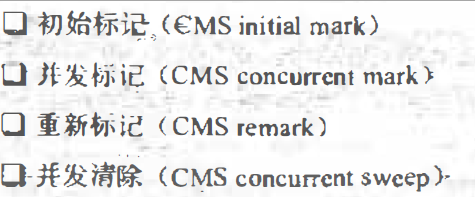
即吞吐量＝运行用户代码时间／（运行用户代码时间＋ 垃圾收集时间）。除此之外，还具有自适应调节策略。



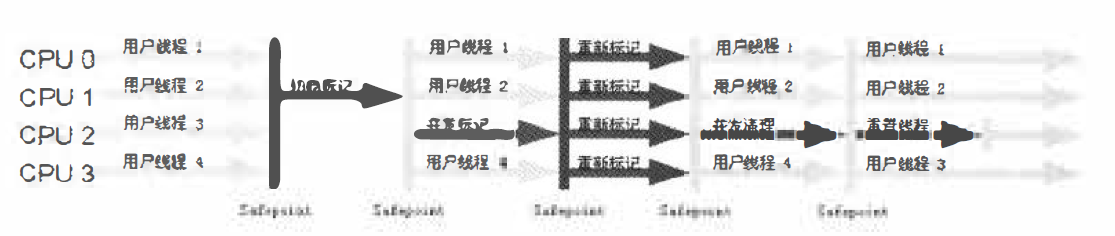
#### CMS收集器

CMS收集器是一钟获取最短回收停预时时为目标的收集器。．CMS收粲器是基于 “标记-一清除")算法实现的。

整个过程分为4个阶段：



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。



优点：并发收集，低停顿

缺点：

1. CMS收集器对CPU资源非常敏感。Cpu越少资源吞吐量越低。
2. CMS收集器无法处理浮动垃圾。由于CMS 并发清理阶段用户线程还在运行, 伴随程序运行自然就还会有新的垃圾不断产生， 这一部分垃圾出现在标记过桯之后 ， CMS无法在当次收集中处理掉它们。
3. 产生大量的空间碎片。

#### G1收集器

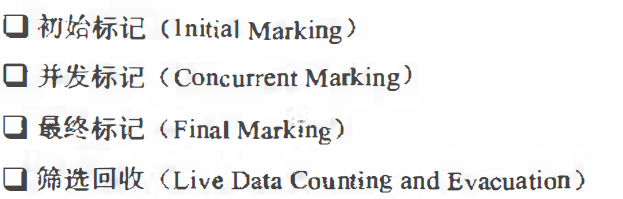
该收集器具有空间整合的特点：

将整个内存空间分区（Region），垃圾回收载Region内进行。并在后台维护一个优先级列表，优先级高的Region先进行回收。

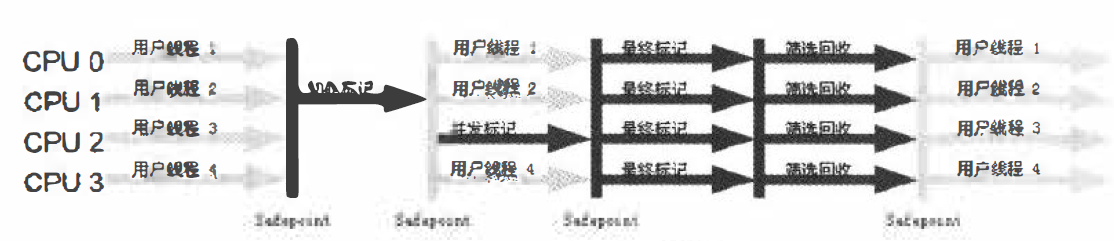
另外由于Region不可能是孤立的． 一个对象分配在某个Region中， 它井非只能被本Region中的其他对象引用， 而是可以与整个Java堆任意的对象发生引用关系。 那在做可达性判定确定对象是否存活的时候， 岂不是还得扫描整个Java堆才能保证准确性吗？

GI中每个Region都有一个与 之对应的RememberedSet，虚拟机发现程序在对Reference类型的数据进行写操作时，检查Reference引用的对象是否处于不同的Region之 中如果是，把相关引用信息，记录到被引用对象所属的Region的RememberedSet之中。 当进行内存回收时， 在GC根节点的枚举范围中加入RememberedSet即可保证不对全堆扫描也不会有遗漏．

执行过程：



初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象；并发标记阶段是从GC Root 开始对堆中对象进行可达性分析， 找出存活的对象， 这阶段耗时较长， 但可与用户程序并发执行；而最终标记阶段则是为了修正在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标记记录， 虚拟机将这段时间对象变化记录在线程Remembered Set Logs里面， 最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中。



### 内存分配与回收策略

#### 对象优先在Eden分配

对象优先在新生代Eden区中分配，当Edmo区没有足够空间进行分配时，虚拟机拟机将发起一次Minor GC。

#### 大对象直接进入老年代

大对象是指，需耍大量连续内存空间的Java对象。

#### 长期存活的对象将进入老年代

如果对象在Eden出生并经过第一次Minor GC后仍然存 活 并且能被survivor容纳的话， 将被移动到survivor空间中， 并且对象年龄设为1。在survivor区中每＂熬过“ 一次Minor GC, 年龄就增加1岁 ，当它的年龄增加到一定程度（默认为15岁）， 就将会被晋升到老年代中．

#### 动态对象年龄判定

为了能更好地适应不同程序的内存状况， 虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代， 如果在Survivor, 空间中相同年龄所有对象大小的 总和大于Survivor,, 空间的一半， 年龄大于或等于该年龄的对象就可以直接进人老年代。

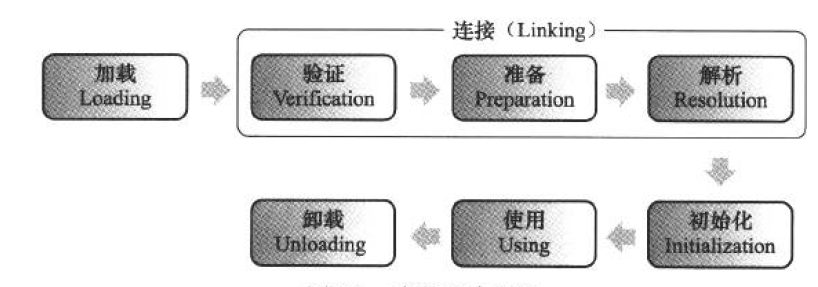
#### 空间分配担保

在发生Minor GC之前， 虚拟机会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， 那么Minor GC可以确保是安全的， 如果不成立， 则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。 如果允许， 那么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， 将尝试看进行一次Minor GC, 尽管这次Minor GC是有风险的， 如果小于， 或者 HandlePromotionFailure设置不允许冒险， 那这时也要改为进行一次Full GC。

## 类加载

### 类加载的时机

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， 验证, 准备、解析、初始化. 使用和卸载7个阶段． 其中验证、 准备、 解析3个部分统称为连接。



什么情况下需要开始类加载过程的第一个阶段， 加载? java虚拟机规范中并没有进行强制约束， 这点可以交给虚拟机的具体实现来自由把握．但是对于初始化阶段， 虚拟机规范则 是严格见定了有且只有5种情况必须立即对类进行 “初始化 ”。

1. 遇到new、 getstatic、putstatic或,invokestatic这4条字节码指令时， 如果类没有进行过初始化， 则需要先触发其初始化． 生成这4条指令的常见的代码场景是 使用new 关键下实例化对象的时候、 读取或设置一个类的静态字段（被final修饰， 已在编译期把结果放人常量1池的静态字段除外）的时候， 以及调用一个类的静态方法的时候。
2. 使用java.long.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，则要先触发其初始化。
3. 如果初始化一个类时发现其父类没有初始化，先初始化其父类。
4. 虚拟机启动时，先初始化主类（main方法）

注意：

1.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；

2.通过数组定义来引用类，不会触发此类的初始化；

3.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类的初始化。

### 类加载的过程

#### 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. 将这个字节流所代表的静态存储结构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.long.Class对象， 作为方法区这个类的各种数据的访问入口。

动态代理技术就是在运行时计算生成二进制字节流。

#### 验证

为了确保Class文件的字节流中包式的信息符合当前虚拟机的要求， 并且不会危害虚拟机自身的安全。验证阶段直接决定了java虚拟机是否能够承受恶意代码的攻击。该过程主要包括，文件格式验证，元数据验证，字节码验证，符号引用验证。

#### 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段， 这些变量所使用的内存都将在方法区中进行分配．这时候进行内存分配的仅包括类变量（被"static""修饰的变员）， 而不包括实例变量． 实例变量将会在对象实例化时随对象一起分配在Java堆中． 其次， 这里所说的初始化"通常情况“ 下 是数据类型的零值。

Public static int value = 123；

那变量value在准备阶段过后的初始值为0而不是123，在初始化阶段才会执行123操作。

上面提到 ， 在 “通常情况“ 下初始值是零， 那相对的会有一些 “特殊情况＂ ．如果 类字段的字段属性表中存在ConstantValue属性， 那在准备阶段变量"'value"'就会被初始化为ConstantValue属性所指定的值 ， 假设上而类变杜value的定义变为

Public static final int value = 123；

Value值被安排为123.

#### 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用以一组符号来描述所引用的目标， 符号可以是任何形式的字面量， 只要使用时能无歧义地定位到目标即可，符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中，各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用可以是直接指向目标的指针、 相对偏移社或是一个能间接定位到目标的句柄． 直接引用是和虚拟机实现的内存布局相关的 ， 同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同． 如果有了且接引用，那引用的目标必定已经在内存中存在。

#### 初始化

初始化阶段是执行类构造器<clinit>()方法的过程。

<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块 (static{}块）中的语句合并产生的， 编译器收集的顺序是由语句在源文件中出现的顺序所决定的， 静态语句块中只能访问到定义在静态语句块之前的变量， 定义在它之后的变量， 在前面的静态语句块可以赋值，但是不能访问。

虚拟机保证在<clint>()之前所有父类的<clinit>()都执行完毕。

### 类加载器

#### 类与类加载器

类加载器虽然只用于实现类的加载动作， 但它在Java程序中起到的作用却远远不限于类加载阶段． 对于任意一个类， 都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性， 每一个类加载器， 都拥有一个独立的类名称空间． 这句话可以表达得更通俗一些： 比较两个类是否 ”相等“， 只有在这两个类是由同一个类加载器加载的前提下才有意义， 否则， 即使这两个类来源于同一个Class文件， 被同一个虚拟机加载， 只要加载它们的类加载器不同， 那这两个类就必定不相等。

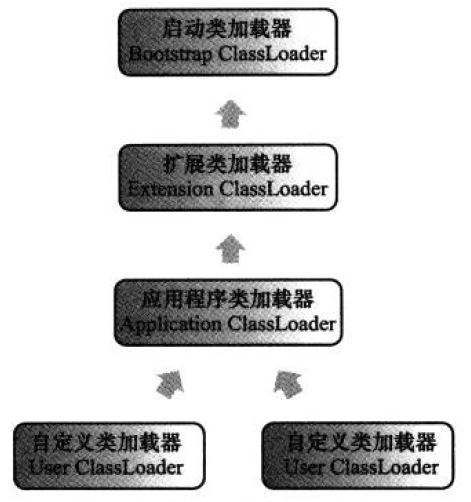
#### 双亲委派模型

从java开发人员角度来讲，类加载器主要有三种。

启动类加载器(Bootstrap ClassLoader), 前面已经介绍过， 这个类将器负责将存放在<JAVA\_HOME>\lib目录。

扩展类加载器（Extension ClassLoder），它负责加载<JAVA\_HOME>\lib\ext目录中的内容。

应用程序类加载器（Application ClassLoader），它负责加载用户类路径上所指定的类库。



双亲委派模型的工作过程是．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，每一个层次的类加载器都是如此， 因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈 自已无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

这种模型可以让基础类比较稳定。如java.lang.Object。

#### 破坏双亲委派模型

此处省去第一次“破坏”。

双亲委派模型的第二次 ” 被破坏” 是由这个模型自身的缺陷所导致的， 双亲委派很好地解决了各个类加载器的基础类的统一问题（越基础的类由越上层的加载器进行加载）， 基础类之所以称为 “基础＂，是因为它们总是作为被用户代码州用的API, 但往往没有绝对的完美， 如果基础类又要调用回用户的代码， 那该怎么办？（如数据库链接）

为了解决这个问题，Java设计团队只好引入了一个不太优雅的设计： 线程上下文类加载器(Thread Context ClassLoader），如果创建线程时还未设置，它将会从父线程中继承一个 如果在应用程序的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。

双亲委派模型的第三次 “被破坏“是由于用户对程序动念性的追求而导致的， 这里所说的”动态性“指的是当前一些非常 “ 热门 ” 的名词： 代码热替换，代码热部署。

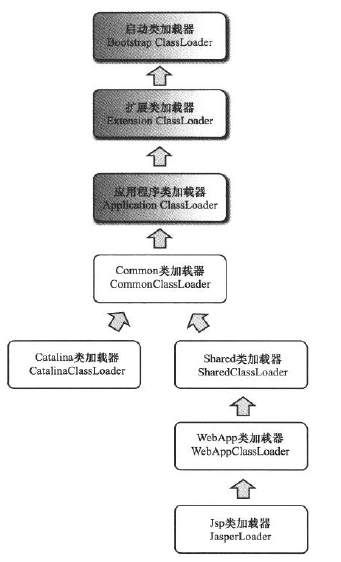
OSGI实现模块化热部署的关键则是它定义的类加载器机制的实现， 每一个程序模块(OSGI中称为Bundlel)都有一个自己的类加载器， 当需要更换一个Bundle时，就把Bundle􀃣连同类加载器一起换掉以实现代码的热替换。

### 案例分析

#### Tomcat

一个功能健全的Web服务器，要解决如下几个问题：

1. 部署在问一个服务器上的两个Web应用程序所使用的Java类库可以实现相互隔离。
2. 部署在同一个服务器上的两个Web应用程序所使用的Java类库可以互相共享。
3. 服务器需要尽可能地保证自身的安全不受部署的Web应用程序影响。



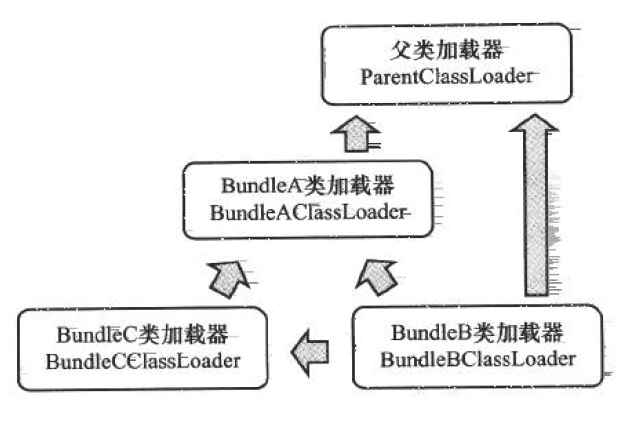
放置在/ common目录中类库可被Tomcat和所有的Web应用程序共同使用．

放置在/server目录中类库可被Tomcat使用，对所有的Web应用程序都不可见

放置在/shared目录中类库可被所有的Web应用程序共同使用， 但对Tomcat自己不可见。

放置在/WebApp/WEB-INF目录中类库仅仅可以被此Web应用程序使用， 对Tomcat和其他Web应用程序都不可见。

#### OSGi



OSGI, 中的每个模块（称为Bundle)与普通的Java类库区别并不太大， 两者一般都以JAR格式进行封装， 井且内部存储的都是JavaPackage和Class. 但是一个Bundle可以声明 它所依赖的JavaPackage (通过lmport-Package描述）， 也可以声明它允许导出发布的Java Package (通过Export-Package描述）， 在OSGi里面， Bundle之间的依赖关系从传统的上 层模块依赖底层模块转变为平级栈块之间的依赖（至少外观上如此）， 而且类库的可见性能得到非常精确的控制， 一个侠块里只有被Export过的Package才可能由外界访问， 其他的Package和Class将会隐藏起来． 除了更精确的模块划分和可见性控制外， 引入OSGI, 的另一个重要理由是 ， 基于OSGi的程序很可能（只是很可能， 并不是一定会）可以实现模块级的热插拔功能。

OSGI的具体规则是：

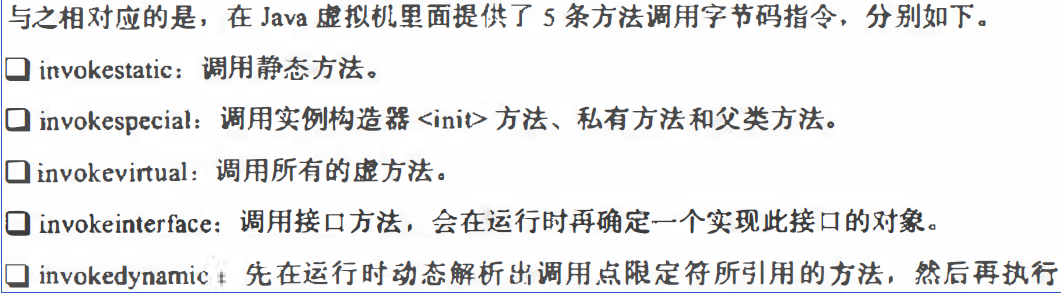
某个Bundle声明了一个它依赖的Package, 如果有其他Bundle声明发布了这个Package, 那么所有对这个Package的类 加载动作都会委派给发布它的Bundle类加载器去完成

## 方法调用（java多态分析）

方法调用并不等同于方法执行， 方法调用阶段唯一的任务就是确定被调用方法的版本。Class文件的编译过程中不包含传统编译中的连接步骤，一切方法调用在Class文件里面存储的都只是符号引用， 而不是方法在实际运行时内存布局中的人口地址。这个特性给Java带来了更强大的动态扩展能力， 但也使得Java方法调用过程变得相对复杂起来， 需要在类加载期间， 甚至到运行期间才能确定目标方法的直接引用．

### 解析

所有方法调用中的目标方法在CIass文件里面都是一个常员池中的符号引用， 在类加载的解析阶段， 会将其中的一部分符号引用转化为直接引用。这种解析能成立的前提是． 方法在程序其正运行之前就有一个可确定的调用版本， 并且这个方法的调用版本在运行期是不可改变的． 换句话说， 调用目标在程序代码写好、 编译器进行编译时就必须确定下来． 这类方法的调用称为解析。



在此之前的4条调用指令， 分派逻辑是固化在'"'虚拟机内部的， 而 , invokedynamic指令的分派逻组是由用户所设定的引导方法决定的。

只要能被invokestatic和invokespecial指令调用的方法， 都可以在解析阶段中确定唯一 的调用版本， 符合这个条件的有静态方法、 私有方法、 实例构造器、 父类方法4类， 它们在 类加载的时候就会把符号引用解析为该方法的直接引用。

解析调用一定是个静态的过程， 在编译期间就完全确定， 在类装载的解析阶段就会把涉及的符号引用全部转变为可确定的直接引用，不会延迟到运行期再去完成。而分派 (Dispatch)调用则可能是静态的也可能是动态的，根据分派依据的宗量数。可分为单分派和多分派。 这两类分派方式的两两组合就构成了静态单分派、 静态多分派、 动态单分派、 动态多分派4种分派组合情况。

### 分派

#### 静态分派

有三个类，Human，Man，Women。其中后两个是前两个的子类。

Human man = new Man();

Man = new Woman();

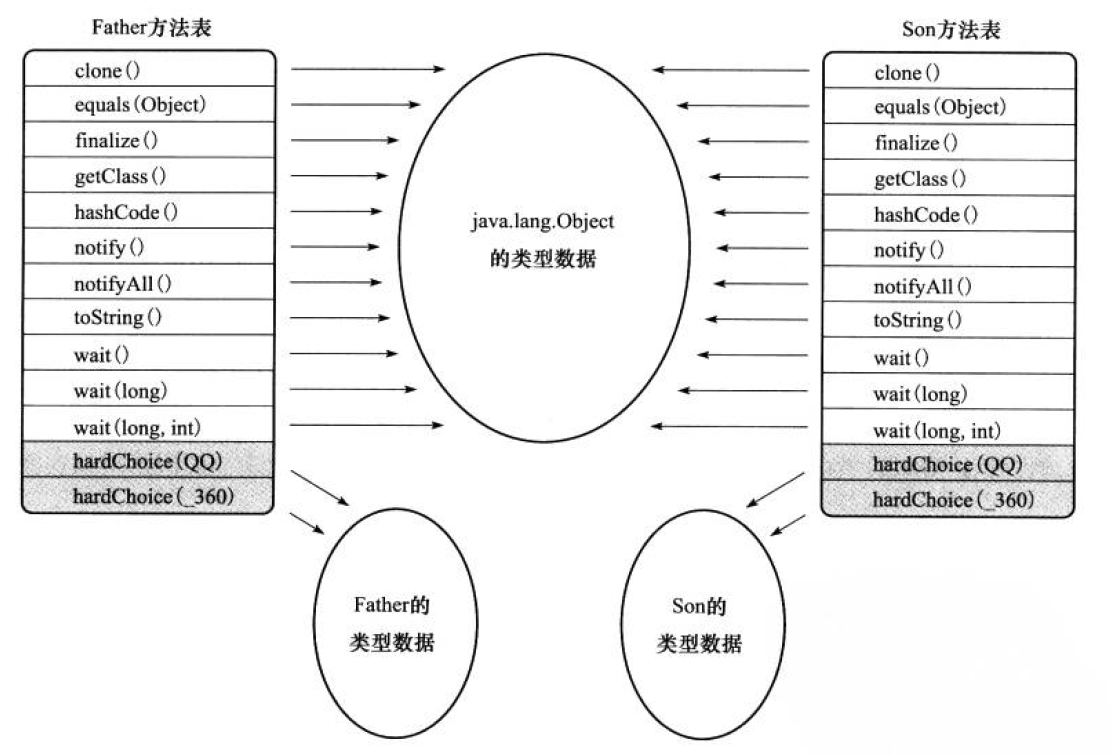
其中Human称为变量的静态类型，Man称为变量的实际类型。静态类型的变化仅仅在使用时发生， 变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的； 而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是仕么。

虚拟机在重载时，根据参数的静态类型而不是实际类型作为判定依据。依赖静态类型来定位方法执行版本的分派动作称为静态分派。典型应用时方法重载。

#### 动态分派

动态分派的典型应用—重写。与静态分派不同的是，静态分派是接收者已经确定，根据参数类型（变量的静态类型）选择方法，动态分派的第一步是先确定接收者。确定接收者的过程是在运行期根据实际类型确定的。

虚拟机通过查询虚方法表的方式来实现动态分派，如果子类没有重写父类发方法，则会返回父类的方法入口。



## Java对象的分配

### **分配策略**

* 栈上分配
  + 线程私有小对象
  + 无逃逸
  + 支持标量替换
  + 无需调整（虚拟机自动优化，无需调优）
* 线程本地分配TLAB（Thread Local Allocation Buffer）
  + 占用eden，默认1%，仍在堆上申请，用作线程专用
  + 多线程的时候不用竞争（加锁）eden就可以申请空间（同步消除），提高效率
  + 小对象
  + 无需调整
* 老年代
  + 大对象（大数组、长字符串）
* eden
  + new普通对象

如果JVM启动了逃逸分析，那么new一个对象时，首先会尝试在栈上分配，如果分配不了，则会尝试在线程本地分配，如果栈上分配与线程本地分配均分配失败的话，则会先判断该对象是否为大对象，如果是大对象，则在老年代分配内存，否则到新生代的eden区分配。

### **逃逸分析**

逃逸分析是一种为其他优化手段提供依据的分析技术，其基本行为是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，称为方法逃逸；也有可能被其外部线程访问到，如赋值给类变量或者可以在其他线程中访问的实例变量，称为线程逃逸。 如果一个对象不会逃逸到方法或者线程之外，则可以对这个对象进行一些高效的优化：

栈上分配Stack Allocation：如果一个对象不会逃逸到方法之外，那么可以让这个对象在栈上分配内存，以提高执行效率，对象所占内存会随着栈帧出栈而销毁。在一般应用中，无逃逸的局部变量对象所占的比例较大，如果能使用栈上分配，那么大量的对象就会随着方法的结束而自动销毁，GC压力减小很多。

同步消除SynchronizationElimination：线程同步是一个相对耗时的过程，如果逃逸分析能够确定一个变量不会逃逸出线程，无法被其他线程访问，那么该变量的读写不存在竞争关系，即可以消除掉对这个变量的同步措施。

标量替换：

* 标量：指的是一个数据已经无法再分解成更小的数据来表示了，Java虚拟机的原始数据类型（int,float等数值类型以及reference类型）都不能再进行进一步的分解
* 聚合量：相对于标量，如果一个数据可继续分解，则可以称作聚合量，Java对象是典型的聚合量。
* 如果把一个Java对象拆散，根据程序访问的情况，将其使用到的成员变量恢复原始类型来访问，这过程成为标量替换
* 如果逃逸分析可以确定一个对象不会被外部访问，且这个对象可以被拆散，那程序真正执行的时候，可以不创建这个对象，而是直接创建它的成员变量来替换这个对象。将对象拆分后，可以在栈上分配内存。

## JIT编译器

JIT编译器是JVM的核心。它对于程序性能的影响最大。CPU只能执行汇编代码或者二进制代码，所有程序都需要被翻译成它们，然后才能被CPU执行。C++以及Fortran这类编译型语言都会通过一个静态的编译器将程序编译成CPU相关的二进制代码。PHP以及Perl这列语言则是解释型语言，只需要安装正确的解释器，它们就能运行在任何CPU之上。当程序被执行的时候，程序代码会被逐行解释并执行。

**编译型语言的优缺点：**

速度快：因为在编译的时候它们能够获取到更多的有关程序结构的信息，从而有机会对它们进行优化。

适用性差：它们编译得到的二进制代码往往是CPU相关的，在需要适配多种CPU时，可能需要编译多次。

**解释型语言的优缺点：**

适应性强：只需要安装正确的解释器，程序在任何CPU上都能够被运行。

速度慢：因为程序需要被逐行翻译，导致速度变慢。同时因为缺乏编译这一过程，执行代码不能通过编译器进行优化。

Java的做法是找到编译型语言和解释性语言的一个中间点：

Java代码会被编译：被编译成Java字节码，而不是针对某种CPU的二进制代码；Java代码会被解释：Java字节码需要被java程序解释执行，此时，Java字节码被翻译成CPU相关的二进制代码。JIT编译器的作用：在程序运行期间，将Java字节码编译成平台相关的二进制代码。正因为此编译行为发生在程序运行期间，所以该编译器被称为Just-In-Time编译器。

### Hotspot

HotSpot VM名字也体现了JIT编译器的工作方式。在VM开始运行一段代码时，并不会立即对它们进行编译。在程序中，总有那么一些“热点”区域，该区域的代码会被反复的执行。而JIT编译器只会编译这些“热点”区域的代码。

总结：

Java综合了编译型语言和解释性语言的优势。

Java会将类文件编译成为Java字节码，然后Java字节码会被JIT编译器选择性地编译成为CPU能够直接运行的二进制代码。

将Java字节码编译成二进制代码后，性能会被大幅度提升。