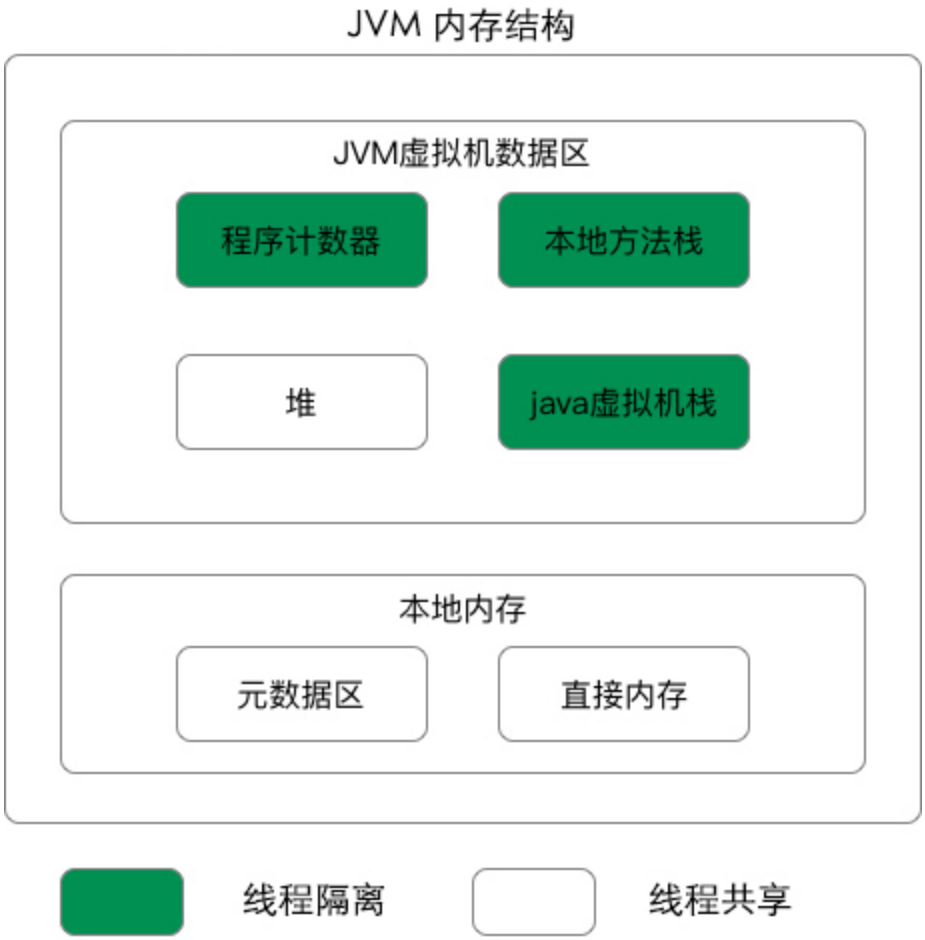
JVM 内存结构

Java 虚拟机的内存空间分为 5 个部分：

- 程序计数器
- Java 虚拟机栈
- 本地方法栈
- 堆
- 方法区



JDK 1.8 同 JDK 1.7 比，最大的差别就是：元数据区取代了永久代。元空间的本质和永久代类似，都是对 JVM 规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元数据空间并不在虚拟机中，而是使用本地内存。

程序计数器（PC 寄存器）

程序计数器的定义

程序计数器是一块较小的内存空间，是当前线程正在执行的那条字节码指令的地址。若当前线程正在执行的是一个本地方法，那么此时程序计数器为 `Undefined` 。

程序计数器的作用

- 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。
- 在多线程情况下，程序计数器记录的是当前线程执行的位置，从而当线程切换回来时，就知道上次线程执行到哪了。

程序计数器的特点

- 是一块较小的内存空间。
- 线程私有，每条线程都有自己的程序计数器。
- 生命周期：随着线程的创建而创建，随着线程的结束而销毁。
- 是唯一一个不会出现 `OutOfMemoryError` 的内存区域。

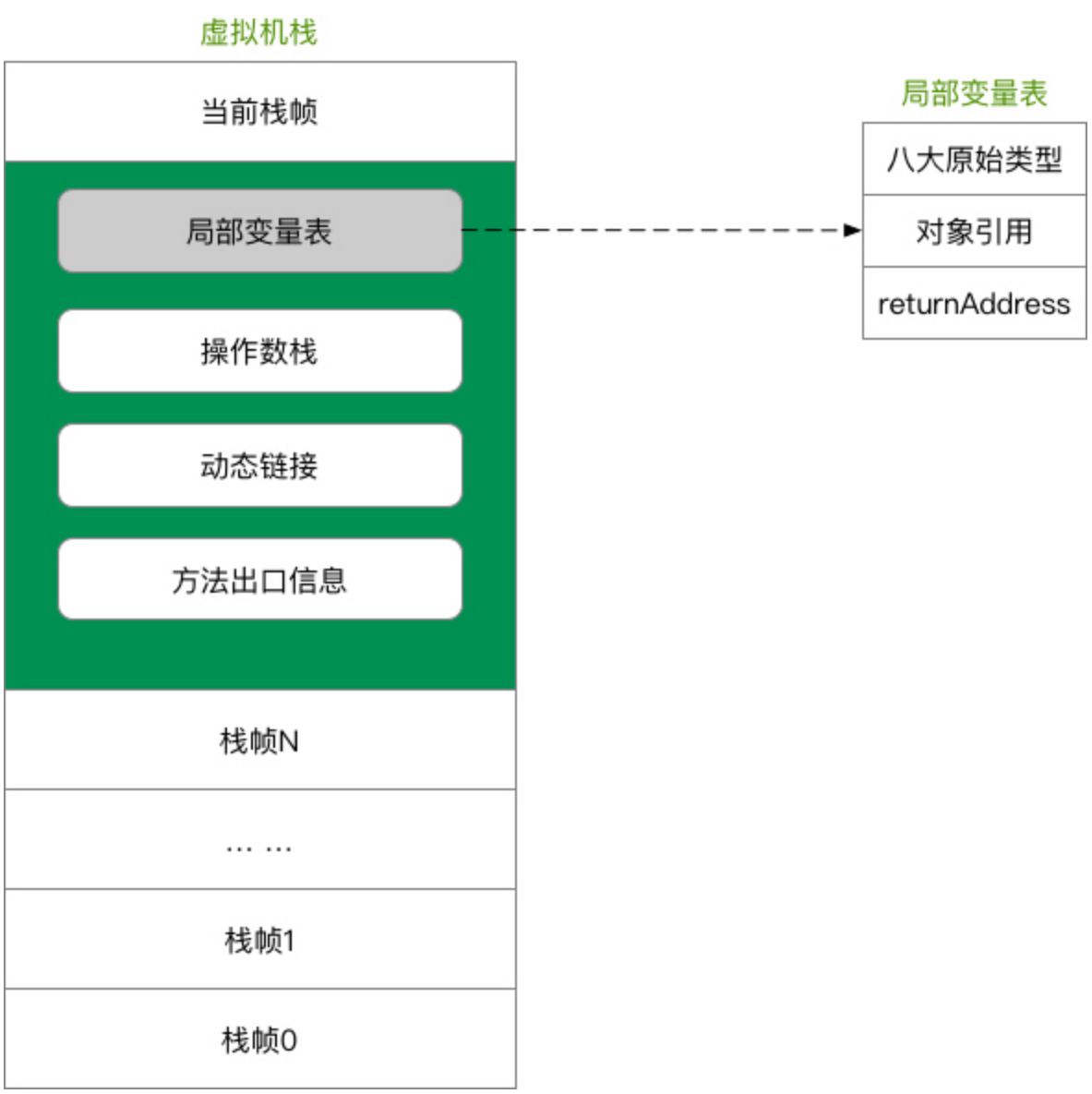
Java 虚拟机栈（Java 栈）

Java 虚拟机栈的定义

Java 虚拟机栈是描述 Java 方法运行过程的内存模型。

Java 虚拟机栈会为每一个即将运行的 Java 方法创建一块叫做“栈帧”的区域，用于存放该方法运行过程中的一些信息，如：

- 局部变量表
- 操作数栈
- 动态链接
- 方法出口信息
- ……



压栈出栈过程

当方法运行过程中需要创建局部变量时，就将局部变量的值存入栈帧中的局部变量表中。

Java 虚拟机栈的栈顶的栈帧是当前正在执行的活动栈，也就是当前正在执行的方法，PC 寄存器也会指向这个地址。只有这个活动的栈帧的本地变量可以被操作数栈使用，当在这个栈帧中调用另一个方法，与之对应的栈帧又会被创建，新创建的栈帧压入栈顶，变为当前的活动栈帧。

方法结束后，当前栈帧被移出，栈帧的返回值变成新的活动栈帧中操作数栈的一个操作数。如果没有返回值，那么新的活动栈帧中操作数栈的操作数没有变化。

> 由于Java 虚拟机栈是与线程对应的，数据不是线程共享的，因此不用关心数据一致性问题，也不会存在同步锁的问题。

Java 虚拟机栈的特点

- 局部变量表随着栈帧的创建而创建，它的大小在编译时确定，创建时只需分配事先规定的大小即可。在方法运行过程中，局部变量表的大小不会发生改变。
- Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。
  - StackOverFlowError 若 Java 虚拟机栈的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时，抛出 StackOverFlowError 异常。
  - OutOfMemoryError 若允许动态扩展，那么当线程请求栈时内存用完了，无法再动态扩展时，抛出 OutOfMemoryError 异常。
- Java 虚拟机栈也是线程私有，随着线程创建而创建，随着线程的结束而销毁。

> 出现 **StackOverFlowError** 时，内存空间可能还有很多。

本地方法栈（C 栈）

本地方法栈的定义

本地方法栈是为 JVM 运行 Native 方法准备的空间，由于很多 Native 方法都是用 C 语言实现的，所以它通常又叫 C 栈。它与 Java 虚拟机栈实现的功能类似，只不过本地方法栈是描述本地方法运行过程的内存模型。

栈帧变化过程

本地方法被执行时，在本地方法栈也会创建一块栈帧，用于存放该方法的局部变量表、操作数栈、动态链接、方法出口信息等。

方法执行结束后，相应的栈帧也会出栈，并释放内存空间。也会抛出 StackOverFlowError 和 OutOfMemoryError 异常。

> 如果 **Java** 虚拟机本身不支持 **Native** 方法，或是本身不依赖于传统栈，那么可以不提供本地方法栈。如果支持本地方法栈，那么这个栈一般会在线程创建的时候按线程分配。

堆

堆的定义

堆是用来存放对象的内存空间，几乎所有的对象都存储在堆中。

堆的特点

- 线程共享，整个 Java 虚拟机只有一个堆，所有的线程都访问同一个堆。而程序计数器、Java 虚拟机栈、本地方法栈都是一个线程对应一个。
- 在虚拟机启动时创建。
- 是垃圾回收的主要场所。
- 进一步可分为：新生代(Eden区 From Survior To Survivor)、老年代。

不同的区域存放不同生命周期的对象，这样可以根据不同的区域使用不同的垃圾回收算法，更具有针对性。

堆的大小既可以固定也可以扩展，但对于主流的虚拟机，堆的大小是可扩展的，因此当线程请求分配内存，但堆已满，且内存已无法再扩展时，就抛出 OutOfMemoryError 异常。

> **Java 堆所使用的内存不需要保证是连续的。而由于堆是被所有线程共享的，所以对它的访问需要注意同步问题，方法和对应的属性都需要保证一致性。**

方法区

方法区的定义

Java 虚拟机规范中定义方法区是堆的一个逻辑部分。方法区存放以下信息：

- 已经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

方法区的特点

- 线程共享。 方法区是堆的一个逻辑部分，因此和堆一样，都是线程共享的。整个虚拟机中只有一个方法区。
- 永久代。 方法区中的信息一般需要长期存在，而且它又是堆的逻辑分区，因此用堆的划分方法，把方法区称为“永久代”。
- 内存回收效率低。 方法区中的信息一般需要长期存在，回收一遍之后可能只有少量信息无效。主要回收目标是：对常量池的回收；对类型的卸载。
- Java 虚拟机规范对方法区的要求比较宽松。 和堆一样，允许固定大小，也允许动态扩展，还允许不实现垃圾回收。

运行时常量池

方法区中存放：类信息、常量、静态变量、即时编译器编译后的代码。常量就存放在运行时常量池中。

当类被 Java 虚拟机加载后， .class 文件中的常量就存放在方法区的运行时常量池中。而且在运行期间，可以向常量池中添加新的常量。如 String 类的 intern() 方法就能在运行期间向常量池中添加字符串常量。

直接内存（堆外内存）

直接内存是除 Java 虚拟机之外的内存，但也可能被 Java 使用。

操作直接内存

在 NIO 中引入了一种基于通道和缓冲的 IO 方式。它可以通过调用本地方法直接分配 Java 虚拟机之外的内存，然后通过一个存储在堆中的 `DirectByteBuffer` 对象直接操作该内存，而无须先将外部内存中的数据复制到堆中再进行操作，从而提高了数据操作的效率。

直接内存的大小不受 Java 虚拟机控制，但既然是内存，当内存不足时就会抛出 OutOfMemoryError 异常。

直接内存与堆内存比较

- 直接内存申请空间耗费更高的性能
- 直接内存读取 IO 的性能要优于普通的堆内存。
- 直接内存作用链： 本地 IO -> 直接内存 -> 本地 IO
- 堆内存作用链：本地 IO -> 直接内存 -> 非直接内存 -> 直接内存 -> 本地 IO

> 服务器管理员在配置虚拟机参数时，会根据实际内存设置 `-Xmx` 等参数信息，但经常忽略直接内存，使得各个内存区域总和大于物理内存限制，从而导致动态扩展时出现 `OutOfMemoryError` 异常。

HotSpot 虚拟机对象探秘

对象的内存布局

在 HotSpot 虚拟机中，对象的内存布局分为以下 3 块区域：

- 对象头（Header）
- 实例数据（Instance Data）
- 对齐填充（Padding）

对象头

对象头记录了对象在运行过程中所需要使用的一些数据：

- 哈希码
- 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

对象头可能包含类型指针，通过该指针能确定对象属于哪个类。如果对象是一个数组，那么对象头还会包括数组长度。

实例数据

实例数据部分就是成员变量的值，其中包括父类成员变量和本类成员变量。

对齐填充

用于确保对象的总长度为 8 字节的整数倍。

HotSpot VM 的自动内存管理系统要求对象的大小必须是 8 字节的整数倍。而对象头部分正好是 8 字节的倍数（1 倍或 2 倍），因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

> 对齐填充并不是必然存在，也没有特别的含义，它仅仅起着占位符的作用。

对象的创建过程

类加载检查

虚拟机在解析 `.class` 文件时，若遇到一条 new 指令，首先它会去检查常量池中是否有这个类的符号引用，并且检查这个符号引用所代表的类是否已被加载、解析和初始化过。如果没有，那么必须先执行相应的类加载过程。

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，接下来从堆中划分一块对应大小的内存空间给新的对象。分配堆中内存有两种方式：

- **指针碰撞**
  如果 Java **堆中内存绝对规整**（说明采用的是**“复制算法”**或**“标记整理法”**），空闲内存和已使用内存中间放着一个指针作为分界点指示器，那么分配内存时只需要把指针向空闲内存挪动一段与对象大小一样的距离，这种分配方式称为**“指针碰撞”**。

- **空闲列表**
  如果 Java **堆中内存并不规整**，已使用的内存和空闲内存交错（说明采用的是**标记-清除法**，有碎片），此时没法简单进行指针碰撞， VM 必须维护一个列表，记录其中哪些内存块空闲可用。分配之时从空闲列表中找到一块足够大的内存空间划分给对象实例。这种方式称为**“空闲列表”**。

初始化

分配完内存后，为对象中的成员变量赋上初始值，设置对象头信息，调用对象的构造函数方法进行初始化。

至此，整个对象的创建过程就完成了。

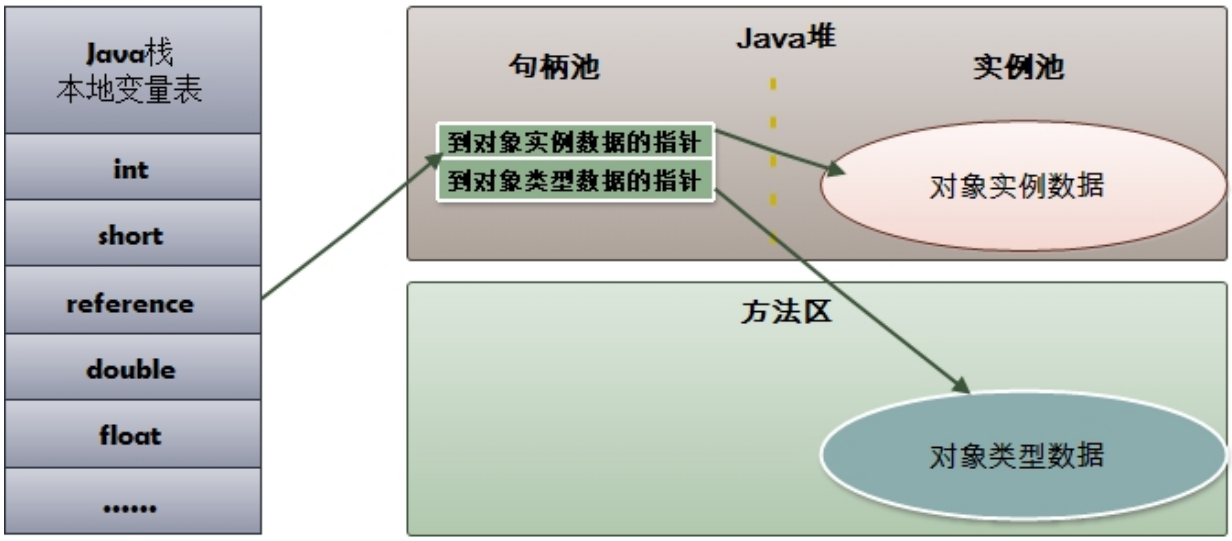
对象的访问方式

所有对象的存储空间都是在堆中分配的，但是这个对象的引用却是在堆栈中分配的。也就是说在建立一个对象时两个地方都分配内存，在堆中分配的内存实际建立这个对象，而在堆栈中分配的内存只是一个指向这个堆对象的指针（引用）而已。 那么根据引用存放的地址类型的不同，对象有不同的访问方式。

句柄访问方式

堆中需要有一块叫做“句柄池”的内存空间，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。

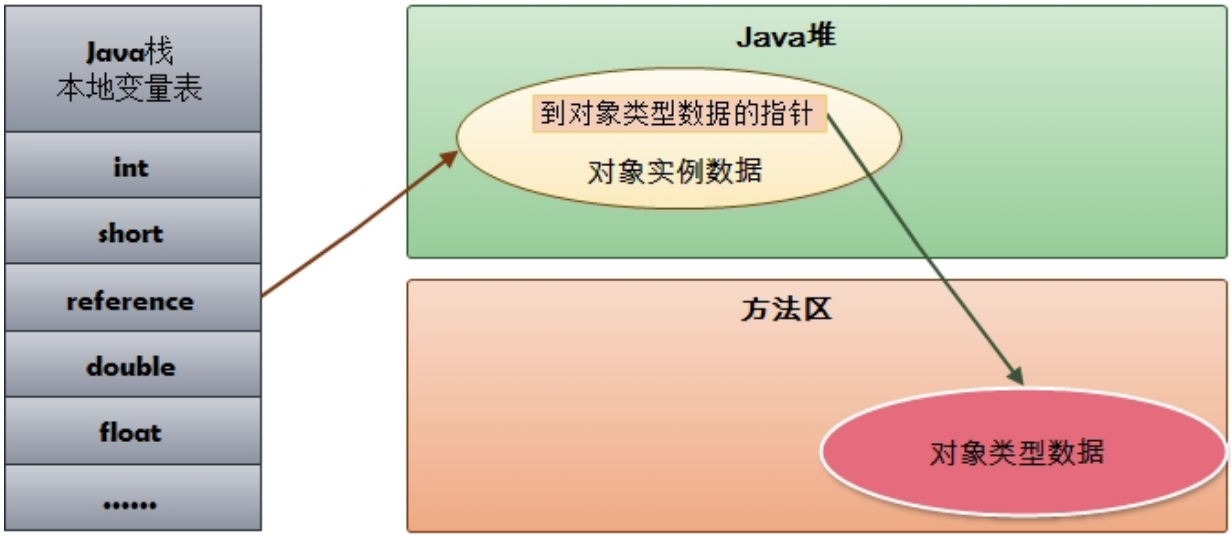
引用类型的变量存放的是该对象的句柄地址（reference）。访问对象时，首先需要通过引用类型的变量找到该对象的句柄，然后根据句柄中对象的地址找到对象。



句柄方式访问对象

直接指针访问方式

引用类型的变量直接存放对象的地址，从而不需要句柄池，通过引用能够直接访问对象。但对象所在的内存空间需要额外的策略存储对象所属的类信息的地址。



直接指针方式访问对象

需要说明的是，HotSpot 采用第二种方式，即直接指针方式来访问对象，只需要一次寻址操作，所以在性能上比句柄访问方式快一倍。但像上面所说，它需要额外的策略来存储对象在方

法区中类信息的地址。

垃圾收集策略与算法

程序计数器、虚拟机栈、本地方法栈随线程而生，也随线程而灭；栈帧随着方法的开始而入栈，随着方法的结束而出栈。这几个区域的内存分配和回收都具有确定性，在这几个区域内不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了。

而对于 Java 堆和方法区，我们只有在程序运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的正是这部分内存。

判定对象是否存活

若一个对象不被任何对象或变量引用，那么它就是无效对象，需要被回收。

引用计数法

在对象头维护着一个 counter 计数器，对象被引用一次则计数器 +1；若引用失效则计数器 -1。当计数器为 0 时，就认为该对象无效了。

引用计数算法的实现简单，判定效率也很高，在大部分情况下它都是一个不错的算法。但是主流的 Java 虚拟机里没有选用引用计数算法来管理内存，主要是因为它很难解决对象之间循环引用的问题。

> 举个栗子👉对象 objA 和 objB 都有字段 instance，令 objA.instance = objB 并且 objB.instance = objA，由于它们互相引用着对方，导致它们的引用计数都不为 0，于是引用计数算法无法通知 GC 收集器回收它们。

可达性分析法

所有和 GC Roots 直接或间接关联的对象都是有效对象，和 GC Roots 没有关联的对象就是无效对象。

GC Roots 是指：

- Java 虚拟机栈（栈帧中的本地变量表）中引用的对象
- 本地方法栈中引用的对象
- 方法区中常量引用的对象
- 方法区中类静态属性引用的对象

GC Roots 并不包括堆中对象所引用的对象，这样就不会有循环引用的问题。

引用的种类

判定对象是否存活与“引用”有关。在 JDK 1.2 以前，Java 中的引用定义很传统，一个对象只有被引用或者没有被引用两种状态，我们希望能描述这一类对象：当内存空间还足够时，则保留在内存中；如果内存空间在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为了以下四种。不同的引用类型，主要体现的是对象不同的可达性状态 `reachable` 和垃圾收集的影响。

强引用（Strong Reference）

类似 "Object obj = new Object()" 这类的引用，就是强引用，只要强引用存在，垃圾收集器永远不会回收被引用的对象。但是，如果我们**错误地保持了强引用**，比如：赋值给了 static 变量，那么对象在很长一段时间内不会被回收，会产生内存泄漏。

软引用（Soft Reference）

软引用是一种相对强引用弱化一些的引用，可以让对象豁免一些垃圾收集，只有当 JVM 认为内存不足时，才会去试图回收软引用指向的对象。JVM 会确保在抛出 OutOfMemoryError 之前，清理软引用指向的对象。软引用通常用来**实现内存敏感的缓存**，如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

弱引用（Weak Reference）

弱引用的**强度比软引用更弱**一些。当 JVM 进行垃圾回收时，**无论内存是否充足，都会回收**只被弱引用关联的对象。

虚引用（Phantom Reference）

虚引用也称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。它仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制，比如，通常用来做所谓的 Post-Mortem 清理机制。

回收堆中无效对象

对于可达性分析中不可达的对象，也并不是没有存活的可能。

判定 finalize() 是否有必要执行

JVM 会判断此对象是否有必要执行 finalize() 方法，如果对象没有覆盖 finalize() 方法，或者 finalize() 方法已经被虚拟机调用过，那么视为“没有必要执行”。那么对象基本上就真的被回收了。

如果对象被判定为有必要执行 finalize() 方法，那么对象会被放入一个 F-Queue 队列中，虚拟机会以较低的优先级执行这些 finalize()方法，但不会确保所有的 finalize() 方法都会执行结束。如果 finalize() 方法出现耗时操作，虚拟机就直接停止指向该方法，将对象清除。

对象重生或死亡

如果在执行 finalize() 方法时，将 this 赋给了某一个引用，那么该对象就重生了。如果没有，那么就会被垃圾收集器清除。

> **任何一个对象的 finalize() 方法只会被系统自动调用一次，如果对象面临下一次回收，它的 finalize() 方法不会被再次执行，想继续在 finalize() 中自救就失效了。**

回收方法区内存

方法区中存放生命周期较长的类信息、常量、静态变量，每次垃圾收集只有少量的垃圾被清除。方法区中主要清除两种垃圾：

- 废弃常量
- 无用的类

判定废弃常量

只要常量池中的常量不被任何变量或对象引用，那么这些常量就会被清除掉。比如，一个字符串 "bingo" 进入了常量池，但是当前系统没有任何一个 String 对象引用常量池中的 "bingo" 常量，也没有其它地方引用这个字面量，必要的话，"bingo"常量会被清理出常量池。

判定无用的类

判定一个类是否是“无用的类”，条件较为苛刻。

- 该类的所有对象都已经被清除
- 加载该类的 ClassLoader 已经被回收
- 该类的 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。

> 一个类被虚拟机加载进方法区，那么在堆中就会有一个代表该类的对象：**java.lang.Class**。这个对象在类被加载进方法区时创建，在方法区该类被删除时清除。

垃圾收集算法

学会了如何判定无效对象、无用类、废弃常量之后，剩余工作就是回收这些垃圾。常见的垃圾收集算法有以下几个：

标记-清除算法

标记的过程是：遍历所有的 `GC Roots` ，然后将所有 `GC Roots` 可达的对象**标记为存活的对象**。

清除的过程将遍历堆中所有的对象，将没有标记的对象全部清除掉。与此同时，清除那些被标记过的对象的标记，以便下次的垃圾回收。

这种方法有两个**不足**：

- 效率问题：标记和清除两个过程的效率都不高。
- 空间问题：标记清除之后会产生大量不连续的内存碎片，碎片太多可能导致以后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（新生代）

为了解决效率问题，“复制”收集算法出现了。它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。当这一块内存用完，需要进行垃圾收集时，就将存活者的对象复制到另一块上面，然后将第一块内存全部清除。这种算法有优有劣：

- 优点：不会有内存碎片的问题。
- 缺点：内存缩小为原来的一半，浪费空间。

为了解决空间利用率问题，可以将内存分为三块： Eden、From Survivor、To Survivor，比例是 8:1:1，每次使用 Eden 和其中一块 Survivor。回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才使用的 Survivor 空间。这样只有 10% 的内存被浪费。

但是我们无法保证每次回收都只有不多于 10% 的对象存活，当 Survivor 空间不够，需要依赖其他内存（指老年代）进行分配担保。

分配担保

为对象分配内存空间时，如果 Eden+Survivor 中空闲区域无法装下该对象，会触发 MinorGC 进行垃圾收集。但如果 Minor GC 过后依然有超过 10% 的对象存活，这样存活的对象直接通过分配担保机制进入老年代，然后再将新对象存入 Eden 区。

标记-整理算法（老年代）

标记：它的第一个阶段与**标记/清除算法**是一模一样的，均是遍历 `GC Roots` ，然后将存活的对象标记。

整理：移动所有**存活的对象**，且按照内存地址次序依次排列，然后将末端内存地址以后的内存全部回收。因此，第二阶段才称为整理阶段。

这是一种老年代的垃圾收集算法。老年代的对象一般寿命比较长，因此每次垃圾回收会有大量对象存活，如果采用复制算法，每次需要复制大量存活的对象，效率很低。

分代收集算法

根据对象存活周期的不同，将内存划分为几块。一般是把 Java 堆分为新生代和老年代，针对各个年代的特点采用最适当的收集算法。

- 新生代：复制算法
- 老年代：标记-清除算法、标记-整理算法

HotSpot 垃圾收集器

HotSpot 虚拟机提供了多种垃圾收集器，每种收集器都有各自的特点，虽然我们要对各个收集器进行比较，但并非为了挑选出一个最好的收集器。我们选择的只是对具体应用最合适的收集器。

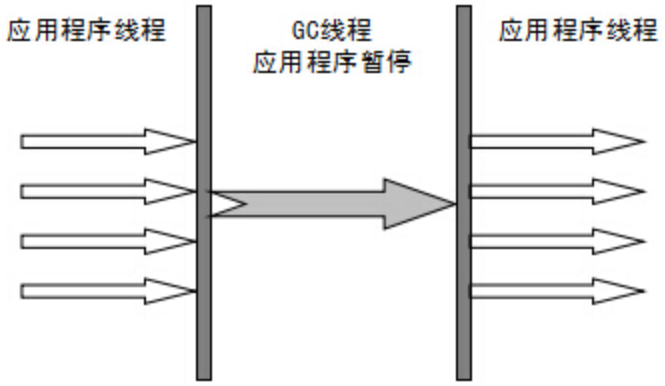
新生代垃圾收集器

Serial 垃圾收集器（单线程）

只开启**一条** GC 线程进行垃圾回收，并且在垃圾收集过程中停止一切用户线程(Stop The World)。

一般客户端应用所需内存较小，不会创建太多对象，而且堆内存不大，因此垃圾收集器回收时间短，即使在这段时间停止一切用户线程，也不会感觉明显卡顿。因此 Serial 垃圾收集器**适合客户端**使用。

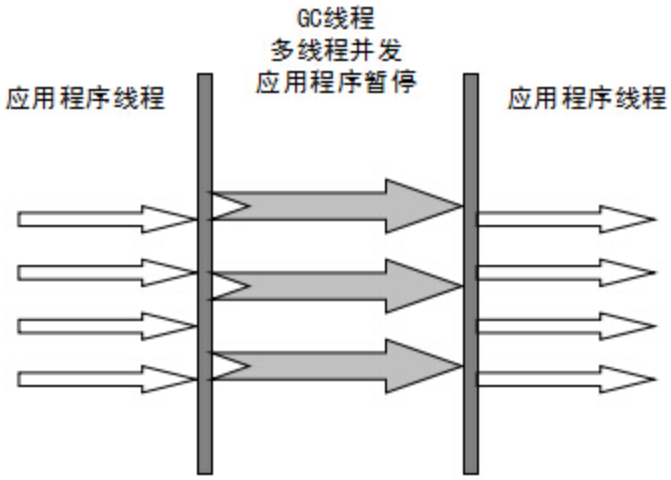
由于 Serial 收集器只使用一条 GC 线程，避免了线程切换的开销，从而简单高效。



ParNew 垃圾收集器（多线程）

ParNew 是 Serial 的多线程版本。由多条 GC 线程并行地进行垃圾清理。但清理过程依然需要 Stop The World。

ParNew 追求“**低停顿时间**”,与 Serial 唯一区别就是使用了多线程进行垃圾收集，在多 CPU 环境下性能比 Serial 会有一定程度的提升；但**线程切换需要额外的开销**，因此在单 CPU 环境中表现不如 Serial。



Parallel Scavenge 垃圾收集器（多线程）

Parallel Scavenge 和 ParNew 一样，都是多线程、新生代垃圾收集器。但是两者有巨大的不同点：

- Parallel Scavenge：追求 CPU 吞吐量，能够在较短时间内完成指定任务，因此适合没有交互的后台计算。
- ParNew：追求降低用户停顿时间，适合交互式应用。

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)

追求高吞吐量，可以通过减少 GC 执行实际工作的时间，然而，仅仅偶尔运行 GC 意味着每当 GC 运行时将有许多工作要做，因为在此期间积累在堆中的对象数量很高。单个 GC 需要花更多的时间来完成，从而导致更高的暂停时间。而考虑到低暂停时间，最好频繁运行 GC 以便更快速完成，反过来又导致吞吐量下降。

- 通过参数 -XX:GCTimeRadio 设置垃圾回收时间占总 CPU 时间的百分比。
- 通过参数 -XX:MaxGCPauseMillis 设置垃圾处理过程最久停顿时间。
- 通过命令 -XX:+UseAdaptiveSizePolicy 开启自适应策略。我们只要设置好堆的大小和 MaxGCPauseMillis 或 GCTimeRadio，收集器会自动调整新生代的大小、Eden 和 Survivor 的比例、对象进入老年代的年龄，以最大程度上接近我们设置的 MaxGCPauseMillis 或 GCTimeRadio。

老年代垃圾收集器

Serial Old 垃圾收集器（单线程）

Serial Old 收集器是 Serial 的老年代版本，都是单线程收集器，只启用一条 GC 线程，都适合客户端应用。它们唯一的区别就是：Serial Old 工作在老年代，使用“标记-整理”算法；Serial 工作在新生代，使用“复制”算法。

Parallel Old 垃圾收集器（多线程）

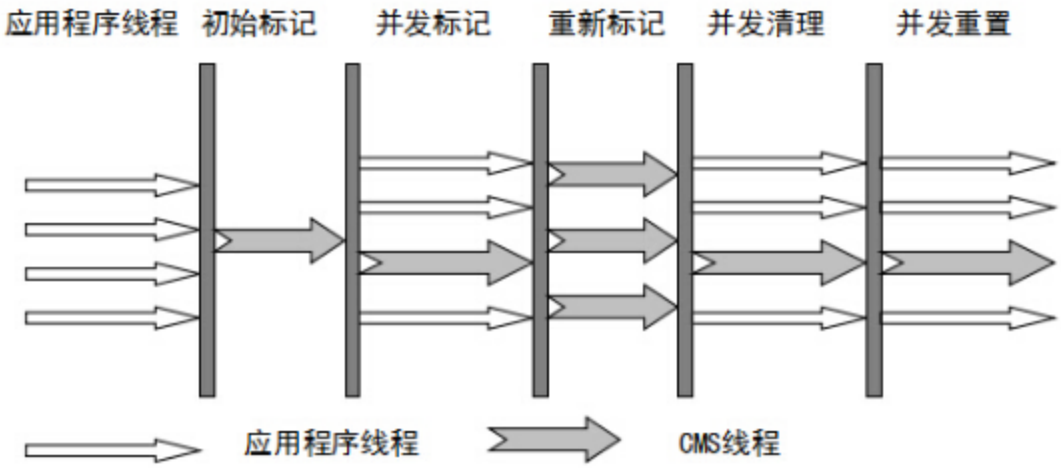
Parallel Old 收集器是 Parallel Scavenge 的老年代版本，追求 CPU 吞吐量。

CMS 垃圾收集器

CMS(Concurrent Mark Sweep，并发标记清除)收集器是以获取最短回收停顿时间为目标的收集器（追求低停顿），它在垃圾收集时使得用户线程和 GC 线程并发执行，因此在垃圾收集过程中用户也不会感到明显的卡顿。

- 初始标记：Stop The World，仅使用一条初始标记线程对所有与 GC Roots 直接关联的对象进行标记。
- 并发标记：使用多条标记线程，与用户线程并发执行。此过程进行可达性分析，标记出所有废弃对象。速度很慢。
- 重新标记：Stop The World，使用多条标记线程并发执行，将刚才并发标记过程中新出现的废弃对象标记出来。
- 并发清除：只使用一条 GC 线程，与用户线程并发执行，清除刚才标记的对象。这个过程非常耗时。

并发标记与并发清除过程耗时最长，且可以与用户线程一起工作，因此，**总体上说**，CMS 收集器的内存回收过程是与用户线程**一起并发执行**的。



CMS 的缺点：

- 吞吐量低
- 无法处理浮动垃圾，导致频繁 Full GC
- 使用“标记-清除”算法产生碎片空间

对于产生碎片空间的问题，可以通过开启 -XX:+UseCMSCompactAtFullCollection，在每次 Full GC 完成后都会进行一次内存压缩整理，将零散在各处的对象整理到一块。设置参数 -XX:CMSFullGCsBeforeCompaction告诉 CMS，经过了 N 次 Full GC 之后再进行一次内存整理。

G1 通用垃圾收集器

G1 是一款面向服务端应用的垃圾收集器，它没有新生代和老年代的概念，而是将堆划分为一块块独立的 Region。当要进行垃圾收集时，首先估计每个 Region 中垃圾的数量，每次都从垃圾回收价值最大的 Region 开始回收，因此可以获得最大的回收效率。

从整体上看， G1 是基于“标记-整理”算法实现的收集器，从局部（两个 Region 之间）上看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。

这里抛个问题👇
一个对象和它内部所引用的对象可能不在同一个 Region 中，那么当垃圾回收时，是否需要扫描整个堆内存才能完整地进行一次可达性分析？

并不！每个 Region 都有一个 Remembered Set，用于记录本区域中所有对象引用的对象所在的区域，进行可达性分析时，只要在 GC Roots 中再加上 Remembered Set 即可防止对整个堆内存进行遍历。

如果不计算维护 Remembered Set 的操作，G1 收集器的工作过程分为以下几个步骤：

- 初始标记：Stop The World，仅使用一条初始标记线程对所有与 GC Roots 直接关联的对象进行标记。
- 并发标记：使用一条标记线程与用户线程并发执行。此过程进行可达性分析，速度很慢。
- 最终标记：Stop The World，使用多条标记线程并发执行。
- 筛选回收：回收废弃对象，此时也要 Stop The World，并使用多条筛选回收线程并发执行。

内存分配与回收策略

对象的内存分配，就是在堆上分配（也可能经过 JIT 编译后被拆散为标量类型并间接在栈上分配），对象主要分配在新生代的 Eden 区上，少数情况下可能直接分配在老年代，**分配规则不固定**，取决于当前使用的垃圾收集器组合以及相关的参数配置。

以下列举几条最普遍的内存分配规则，供大家学习。

对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 区中分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。

👇**Minor GC** vs **Major GC/Full GC**：

- Minor GC：回收新生代（包括 Eden 和 Survivor 区域），因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。
- Major GC / Full GC: 回收老年代，出现了 Major GC，经常会伴随至少一次的 Minor GC，但这并非绝对。Major GC 的速度一般会比 Minor GC 慢 10 倍 以上。

> **在 JVM 规范中，Major GC 和 Full GC 都没有一个正式的定义，所以有人也简单地认为 Major GC 清理老年代，而 Full GC 清理整个内存堆。**

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，如很长的字符串或数据。

一个大对象能够存入 Eden 区的概率比较小，发生分配担保的概率比较大，而分配担保需要涉及大量的复制，就会造成效率低下。

虚拟机提供了一个 -XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代分配，这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存复制。（还记得吗，新生代采用复制算法回收垃圾）

长期存活的对象将进入老年代

JVM 给每个对象定义了一个对象年龄计数器。当新生代发生一次 Minor GC 后，存活下来的对象年龄 +1，当年龄超过一定值时，就将超过该值的所有对象转移到老年代中去。

使用 -XXMaxTenuringThreshold 设置新生代的最大年龄，只要超过该参数的新生代对象都会被转移到老年代中去。

动态对象年龄判定

如果当前新生代的 Survivor 中，相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄 >= 该年龄的对象就可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄。

空间分配担保

JDK 6 Update 24 之前的规则是这样的：
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，Minor GC 可以确保是安全的； 如果不成立，则虚拟机会查看 HandlePromotionFailure 值是否设置为允许担保失败， 如果是，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，那此时也要改为进行一次 Full GC。

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。

通过清除老年代中废弃数据来扩大老年代空闲空间，以便给新生代作担保。

这个过程就是分配担保。

👇总结一下有哪些情况可能会触发 JVM 进行 Full GC。

1. System.gc() 方法的调用
   此方法的调用是建议 JVM 进行 Full GC，注意这**只是建议而非一定**，但在很多情况下它会触发 Full GC，从而增加 Full GC 的频率。通常情况下我们只需要让虚拟机自己去管理内存即可，我们可以通过 -XX:+ DisableExplicitGC 来禁止调用 System.gc()。

2. 老年代空间不足
   老年代空间不足会触发 Full GC操作，若进行该操作后空间依然不足，则会抛出如下错误：

```
java.lang.OutOfMemoryError: Java heap space
```

3. 永久代空间不足

   JVM 规范中运行时数据区域中的方法区，在 HotSpot 虚拟机中也称为永久代（Permanet Generation），存放一些类信息、常量、静态变量等数据，当系统要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，会触发 Full GC。如果经过 Full GC 仍然回收不了，那么 JVM 会抛出如下错误信息：

   ```
   java.lang.OutOfMemoryError: PermGen space
   ```

4. CMS GC 时出现 promotion failed 和 concurrent mode failure

   promotion failed，就是上文所说的担保失败，而 concurrent mode failure 是在执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足造成的。

5. 统计得到的Minor GC晋升到旧生代的平均大小大于老年代的剩余空间

JVM 性能调优

在高性能硬件上部署程序，目前主要有两种方式：

- 通过 64 位 JDK 来使用大内存；
- 使用若干个 32 位虚拟机建立逻辑集群来利用硬件资源。

使用 64 位 JDK 管理大内存

堆内存变大后，虽然垃圾收集的频率减少了，但每次垃圾回收的时间变长。 如果堆内存为14 G，那么每次 Full GC 将长达数十秒。如果 Full GC 频繁发生，那么对于一个网站来说是无法忍受的。

对于用户交互性强、对停顿时间敏感的系统，可以给 Java 虚拟机分配超大堆的前提是有把握把应用程序的 Full GC 频率控制得足够低，至少要低到不会影响用户使用。

可能面临的问题：

- 内存回收导致的长时间停顿；
- 现阶段，64位 JDK 的性能普遍比 32 位 JDK 低；
- 需要保证程序足够稳定，因为这种应用要是产生堆溢出几乎就无法产生堆转储快照（因为要产生超过 10GB 的 Dump 文件），哪怕产生了快照也几乎无法进行分析；
- 相同程序在 64 位 JDK 消耗的内存一般比 32 位 JDK 大，这是由于指针膨胀，以及数据类型对齐补白等因素导致的。

使用 32 位 JVM 建立逻辑集群

在一台物理机器上启动多个应用服务器进程，每个服务器进程分配不同端口， 然后在前端搭建一个负载均衡器，以反向代理的方式来分配访问请求。

考虑到在一台物理机器上建立逻辑集群的目的仅仅是为了尽可能利用硬件资源，并不需要关心状态保留、热转移之类的高可用性能需求， 也不需要保证每个虚拟机进程有绝对的均衡负载，因此使用无 Session 复制的亲合式集群是一个不错的选择。 我们仅仅需要保障集群具备亲合性，也就是均衡器按一定的规则算法（一般根据 SessionID 分配） 将一个固定的用户请求永远分配到固定的一个集群节点进行处理即可。

可能遇到的问题：

- 尽量避免节点竞争全局资源，如磁盘竞争，各个节点如果同时访问某个磁盘文件的话，很可能导致 IO 异常；
- 很难高效利用资源池，如连接池，一般都是在节点建立自己独立的连接池，这样有可能导致一些节点池满了而另外一些节点仍有较多空余；
- 各个节点受到 32 位的内存限制；
- 大量使用本地缓存的应用，在逻辑集群中会造成较大的内存浪费，因为每个逻辑节点都有一份缓存，这时候可以考虑把本地缓存改成集中式缓存。

调优案例分析与实战

场景描述

一个小型系统，使用 32 位 JDK，4G 内存，测试期间发现服务端不定时抛出内存溢出异常。 加入 -XX:+HeapDumpOnOutOfMemoryError（添加这个参数后，堆内存溢出时就会输出异常日志）， 但再次发生内存溢出时，没有生成相关异常日志。

分析

在 32 位 JDK 上，1.6G 分配给堆，还有一部分分配给 JVM 的其他内存，直接内存最大也只能在剩余的 0.4G 空间中分出一部分， 如果使用了 NIO，JVM 会在 JVM 内存之外分配内存空间，那么就要小心“直接内存”不足时发生内存溢出异常了。

直接内存的回收过程

直接内存虽然不是 JVM 内存空间，但它的垃圾回收也由 JVM 负责。

垃圾收集进行时，虚拟机虽然会对直接内存进行回收， 但是直接内存却不能像新生代、老年代那样，发现空间不足了就通知收集器进行垃圾回收， 它只能等老年代满了后 Full GC，然后“顺便”帮它清理掉内存的废弃对象。 否则只能一直等到抛出内存溢出异常时，先 catch 掉，再在

catch 块里大喊 “System.gc()”。 要是虚拟机还是不听，那就只能眼睁睁看着堆中还有许多空闲内存，自己却不得不抛出内存溢出异常了。

类文件结构

JVM 的“无关性”

谈论 JVM 的无关性，主要有以下两个：

- 平台无关性：任何操作系统都能运行 Java 代码
- 语言无关性： JVM 能运行除 Java 以外的其他代码

Java 源代码首先需要使用 Javac 编译器编译成 .class 文件，然后由 JVM 执行 .class 文件，从而程序开始运行。

JVM 只认识 .class 文件，它不关心是何种语言生成了 .class 文件，只要 .class 文件符合 JVM 的规范就能运行。 目前已经有 JRuby、Jython、Scala 等语言能够在 JVM 上运行。它们有各自的语法规则，不过它们的编译器 都能将各自的源码编译成符合 JVM 规范的 .class 文件，从而能够借助 JVM 运行它们。

> **Java 语言中的各种变量、关键字和运算符号的语义最终都是由多条字节码命令组合而成的， 因此字节码命令所能提供的语义描述能力肯定会比 Java 语言本身更加强大。因此，有一些 Java 语言本身无法有效支持的语言特性，不代表字节码本身无法有效支持。**

Class 文件结构

Class 文件是二进制文件，它的内容具有严格的规范，文件中没有任何空格，全都是连续的 0/1。Class 文件 中的所有内容被分为两种类型：无符号数、表。

- 无符号数 无符号数表示 Class 文件中的值，这些值没有任何类型，但有不同的长度。u1、u2、u4、u8 分别代表 1/2/4/8 字节的无符号数。
- 表 由多个无符号数或者其他表作为数据项构成的复合数据类型。

Class 文件具体由以下几个构成:

- 魔数
- 版本信息
- 常量池
- 访问标志
- 类索引、父类索引、接口索引集合

- 字段表集合
- 方法表集合
- 属性表集合

魔数

Class 文件的头 4 个字节称为魔数，用来表示这个 Class 文件的类型。

Class 文件的魔数是用 16 进制表示的“CAFE BABE”，是不是很具有浪漫色彩？

> 魔数相当于文件后缀名，只不过后缀名容易被修改，不安全，因此在 Class 文件中标识文件类型比较合适。

版本信息

紧接着魔数的 4 个字节是版本信息，5-6 字节表示次版本号，7-8 字节表示主版本号，它们表示当前 Class 文件中使用的是哪个版本的 JDK。

高版本的 JDK 能向下兼容以前版本的 Class 文件，但不能运行以后版本的 Class 文件，即使文件格式并未发生任何变化，虚拟机也必需拒绝执行超过其版本号的 Class 文件。

常量池

版本信息之后就是常量池，常量池中存放两种类型的常量：

- 字面值常量

字面值常量就是我们在程序中定义的字符串、被 final 修饰的值。

- 符号引用

符号引用就是我们定义的各种名字：类和接口的全限定名、字段的名字和描述符、方法的名字和描述符。

常量池的特点

- 常量池中常量数量不固定，因此常量池开头放置一个 u2 类型的无符号数，用来存储当前常量池的容量。
- 常量池的每一项常量都是一个表，表开始的第一位是一个 u1 类型的标志位（tag），代表当前这个常量属于哪种常量类型。

常量池中常量类型

类型	tag	描述
CONSTANT_utf8_info	1	UTF-8编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	标识方法类型
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

对于 CONSTANT_Class_info（此类型的常量代表一个类或者接口的符号引用），它的二维表结构如下：

类型	名称	数量
u1	tag	1
u2	name_index	1

tag 是标志位，用于区分常量类型；name_index 是一个索引值，它指向常量池中一个 CONSTANT_Utf8_info 类型常量，此常量代表这个类（或接口）的全限定名，这里 name_index 值若为 0x0002，也即是指向了常量池中的第二项常量。

CONSTANT_Utf8_info 型常量的结构如下：

类型	名称	数量
u1	tag	1
u2	length	1
u1	bytes	length

tag 是当前常量的类型；length 表示这个字符串的长度；bytes 是这个字符串的内容（采用缩略的 UTF8 编码）

访问标志

在常量池结束之后，紧接着的两个字节代表访问标志，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；是否被 abstract/final 修饰。

类索引、父类索引、接口索引集合

类索引和父类索引都是一个 u2 类型的数据，而接口索引集合是一组 u2 类型的数据的集合，Class 文件中由这三项数据来确定类的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名。

由于 Java 不允许多重继承，所以父类索引只有一个，除了 java.lang.Object 之外，所有的 Java 类都有父类，因此除了 java.lang.Object 外，所有 Java 类的父类索引都不为 0。一个类可能实现了多个接口，因此用接口索引集合来描述。这个集合第一项为 u2 类型的数据，表示索引表的容量，接下来就是接口的名字索引。

类索引和父类索引用两个 u2 类型的索引值表示，它们各自指向一个类型为 CONSTANT_Class_info 的类描述符常量，通过该常量总的索引值可以找到定义在 CONSTANT_Utf8_info 类型的常量中的全限定名字符串。

字段表集合

字段表集合存储本类涉及到的成员变量，包括实例变量和类变量，但不包括方法中的局部变量。

每一个字段表只表示一个成员变量，本类中的所有成员变量构成了字段表集合。字段表结构如下：

类型	名称	数量	说明
u2	access_flags	1	字段的访问标志，与类稍有不同
u2	name_index	1	字段名字的索引
u2	descriptor_index	1	描述符，用于描述字段的数据类型。 基本数据类型用大写字母表示； 对象类型用“L 对象类型的全限定名”表示。
u2	attributes_count	1	属性表集合的长度

u2	attributes	attributes_count	属性表集合，用于存放属性的额外信息，如属性的值。

> 字段表集合中不会出现从父类（或接口）中继承而来的字段，但有可能出现原本 Java 代码中不存在的字段，譬如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。

方法表集合

方法表结构与属性表类似。

volatile 关键字 和 transient 关键字不能修饰方法，所以方法表的访问标志中没有 ACC_VOLATILE 和 ACC_TRANSIENT 标志。

方法表的属性表集合中有一张 Code 属性表，用于存储当前方法经编译器编译后的字节码指令。

属性表集合

每个属性对应一张属性表，属性表的结构如下：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

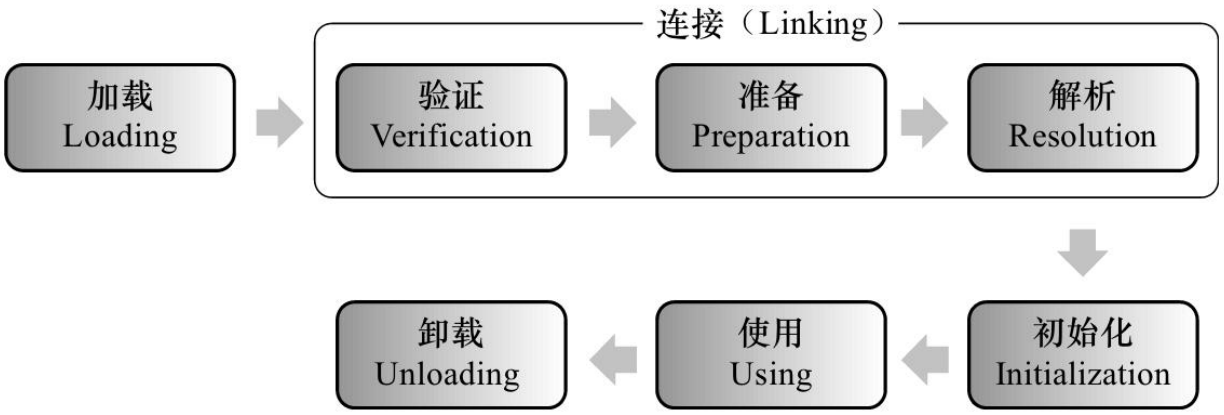
类加载的时机

类的生命周期

类从被加载到虚拟机内存开始，到卸载出内存为止，它的整个生命周期包括以下 7 个阶段：

- 加载
- 验证
- 准备
- 解析
- 初始化
- 使用
- 卸载

验证、准备、解析 3 个阶段统称为连接。



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（注意是“开始”，而不是“进行”或“完成”），而解析阶段则不一定：它在某些情况下可以在初始化后再开始，这是为了支持 Java 语言的运行时绑定。

类加载过程中“初始化”开始的时机

Java 虚拟机规范没有强制约束类加载过程的第一阶段（即：加载）什么时候开始，但对于“初始化”阶段，有着严格的规定。有且仅有 5 种情况必须立即对类进行“初始化”：

- 在遇到 new、putstatic、getstatic、invokestatic 字节码指令时，如果类尚未初始化，则需要先触发其初始化。
- 对类进行反射调用时，如果类还没有初始化，则需要先触发其初始化。
- 初始化一个类时，如果其父类还没有初始化，则需要先初始化父类。
- 虚拟机启动时，用于需要指定一个包含 main() 方法的主类，虚拟机会先初始化这个主类。
- 当使用 JDK 1.7 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_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类还没初始化，则需要先触发其初始化。

这 5 种场景中的行为称为对一个类进行**主动引用**，除此之外，其它所有引用类的方式都不会触发初始化，称为**被动引用**。

被动引用演示 Demo

Demo1

```java
/**
 * 被动引用 Demo1：
```

```
 * 通过子类引用父类的静态字段，不会导致子类初始化。
 *
 * @author ylb
 *
 */
class SuperClass {
    static {
        System.out.println("SuperClass init!");
    }

    public static int value = 123;
}

class SubClass extends SuperClass {
    static {
        System.out.println("SubClass init!");
    }
}

public class NotInitialization {

    public static void main(String[] args) {
        System.out.println(SubClass.value);
        // SuperClass init!
    }

}
```

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。

Demo2

java

```
/**
 * 被动引用 Demo2:
 * 通过数组定义来引用类，不会触发此类的初始化。
 *
 * @author ylb
 *
 */
```

```
public class NotInitialization {

    public static void main(String[] args) {
        SuperClass[] superClasses = new SuperClass[10];
    }

}
```

这段代码不会触发父类的初始化，但会触发“[L 全类名”这个类的初始化，它由虚拟机自动生成，直接继承自 java.lang.Object，创建动作由字节码指令 newarray 触发。

Demo3

java

```
/**
 * 被动引用 Demo3:
 * 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触
 *
 * @author ylb
 *
 */
class ConstClass {
    static {
        System.out.println("ConstClass init!");
    }

    public static final String HELLO_BINGO = "Hello Bingo";

}

public class NotInitialization {

    public static void main(String[] args) {
        System.out.println(ConstClass.HELLO_BINGO);
    }

}
```

编译通过之后，常量存储到 NotInitialization 类的常量池中，NotInitialization 的 Class 文件中并没有 ConstClass 类的符号引用入口，这两个类在编译成 Class 之后就没有任何联系了。

接口的加载过程

接口加载过程与类加载过程稍有不同。

当一个类在初始化时，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，当真正用到父接口的时候才会初始化。

类加载的过程

类加载过程包括 5 个阶段：加载、验证、准备、解析和初始化。

加载

加载的过程

“加载”是“类加载”过程的一个阶段，不能混淆这两个名词。在加载阶段，虚拟机需要完成 3 件事：

- 通过类的全限定名获取该类的二进制字节流。
- 将二进制字节流所代表的静态结构转化为方法区的运行时数据结构。
- 在内存中创建一个代表该类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

获取二进制字节流

对于 Class 文件，虚拟机没有指明要从哪里获取、怎样获取。除了直接从编译好的 .class 文件中读取，还有以下几种方式：

- 从 zip 包中读取，如 jar、war等
- 从网络中获取，如 Applet
- 通过动态代理技术生成代理类的二进制字节流
- 由 JSP 文件生成对应的 Class 类
- 从数据库中读取，如 有些中间件服务器可以选择把程序安装到数据库中来完成程序代码在集群间的分发。

“非数组类”与“数组类”加载比较

- 非数组类加载阶段可以使用系统提供的引导类加载器，也可以由用户自定义的类加载器完成，开发人员可以通过定义自己的类加载器控制字节流的获取方式（如重写一个类加载器的 loadClass() 方法）

- 数组类本身不通过类加载器创建，它是由 Java 虚拟机直接创建的，再由类加载器创建数组中的元素类。

注意事项

- 虚拟机规范未规定 Class 对象的存储位置，对于 HotSpot 虚拟机而言，Class 对象比较特殊，它虽然是对象，但存放在方法区中。
- 加载阶段与连接阶段的部分内容交叉进行，加载阶段尚未完成，连接阶段可能已经开始了。但这两个阶段的开始时间仍然保持着固定的先后顺序。

验证

验证的重要性

验证阶段确保 Class 文件的字节流中包含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。

验证的过程

- 文件格式验证 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，验证点如下：
  - 是否以魔数 0XCAFEBABE 开头
  - 主次版本号是否在当前虚拟机处理范围内
  - 常量池是否有不被支持的常量类型
  - 指向常量的索引值是否指向了不存在的常量
  - CONSTANT_Utf8_info 型的常量是否有不符合 UTF8 编码的数据
  - ……
- 元数据验证 对字节码描述信息进行语义分析，确保其符合 Java 语法规范。
- 字节码验证 本阶段是验证过程中最复杂的一个阶段，是对方法体进行语义分析，保证方法在运行时不会出现危害虚拟机的事件。
- 符号引用验证 本阶段发生在解析阶段，确保解析正常执行。

准备

准备阶段是正式为类变量（或称“静态成员变量”）分配内存并设置初始值的阶段。这些变量（不包括实例变量）所使用的内存都在方法区中进行分配。

初始值“通常情况下”是数据类型的零值（0, null...），假设一个类变量的定义为：

java

```
public static int value = 123;
```

那么变量 value 在准备阶段过后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 Java 方法。

存在“特殊情况”：如果类字段的字段属性表中存在 ConstantValue 属性，那么在准备阶段 value 就会被初始化为 ConstantValue 属性所指定的值，假设上面类变量 value 的定义变为：

java

```
public static final int value = 123;
```

那么在准备阶段虚拟机会根据 ConstantValue 的设置将 value 赋值为 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。

初始化

类初始化阶段是类加载过程的最后一步，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中可以赋值，但不能访问。如下方代码所示：

java

```
public class Test {
    static {
        i = 0;  // 给变量赋值可以正常编译通过
        System.out.println(i);  // 这句编译器会提示“非法向前引用”
    }
    static int i = 1;
}
```

<clinit>() 方法不需要显式调用父类构造器，虚拟机会保证在子类的 <clinit>() 方法执行之前，父类的 <clinit>() 方法已经执行完毕。

由于父类的 <clinit>() 方法先执行，意味着父类中定义的静态语句块要优先于子类的变量赋值操作。如下方代码所示：

```java
static class Parent {
    public static int A = 1;
    static {
        A = 2;
    }
}

static class Sub extends Parent {
    public static int B = A;
}

public static void main(String[] args) {
    System.out.println(Sub.B); // 输出 2
}
```

<clinit>() 方法不是必需的，如果一个类没有静态语句块，也没有对类变量的赋值操作，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代码块，但接口也需要通过 <clinit>() 方法为接口中定义的静态成员变量显式初始化。但接口与类不同，接口的 <clinit>() 方法不需要先执行父类的 <clinit>() 方法，只有当父接口中定义的变量使用时，父接口才会初始化。

虚拟机会保证一个类的 <clinit>() 方法在多线程环境中被正确加锁、同步。如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的 <clinit>() 方法。

类加载器

类与类加载器

判断类是否“相等”

任意一个类，都由**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，都有一个独立的类名称空间。

因此，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

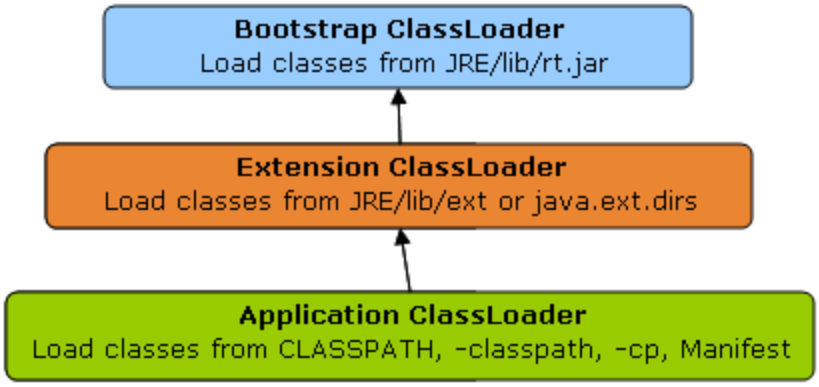
器不同，那么这两个类就必定不相等。

这里的“相等”，包括代表类的 Class 对象的 equals() 方法、isInstance() 方法的返回结果，也包括使用 instanceof 关键字做对象所属关系判定等情况。

加载器种类

系统提供了 3 种类加载器：

- 启动类加载器（Bootstrap ClassLoader）： 负责将存放在 `<JAVA_HOME>\lib` 目录中的，并且能被虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。
- 扩展类加载器（Extension ClassLoader）： 负责加载 `<JAVA_HOME>\lib\ext` 目录中的所有类库，开发者可以直接使用扩展类加载器。
- 应用程序类加载器（Application ClassLoader）： 由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，所以一般也称它为“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这个类加载器，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。



当然，如果有必要，还可以加入自己定义的类加载器。

双亲委派模型

什么是双亲委派模型

双亲委派模型是描述类加载器之间的层次关系。它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。（父子关系一般不会以继承的关系实现，而是以组合关系来复用父加载器的代码）

工作过程

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己无法完成这个加载请求（找不到所需的类）时，子加载器才会尝试自己去加载。

在 java.lang.ClassLoader 中的 loadClass() 方法中实现该过程。

为什么使用双亲委派模型

像 java.lang.Object 这些存放在 rt.jar 中的类，无论使用哪个类加载器加载，最终都会委派给最顶端的启动类加载器加载，从而使得不同加载器加载的 Object 类都是同一个。

相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 java.lang.Object 的类，并放在 classpath 下，那么系统将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无法保证。

公众号

Doocs 技术社区旗下唯一公众号「**Doocs开源社区**」，欢迎扫码关注，**专注分享技术领域相关知识及业内最新资讯**。当然，也可以加我个人微信（备注：GitHub），拉你进技术交流群。

	
公众平台-	个人微信-