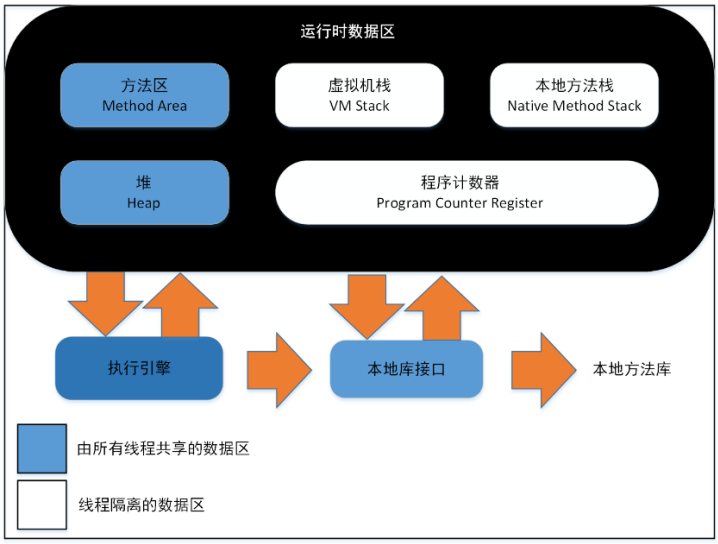


程序计数器（Program Counter Register）是当前线程所执行的**字节码的行号指示器**，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

每条线程都有一个独立的程序计数器，各条线程之间计数器互不影响——“**线程私有**”。

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**，这个计数器值则为**空**。

Java 虚拟机栈是**线程私有**的，它的生命周期与线程相同。虚拟机栈描述的是 **Java 方法执行的内存模型**：**每个方法在执行的同时都会创建一个栈帧**（Stack Frame）用于存储**局部变量表、操作数栈、动态连接、方法出口**等信息。**每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程**。

局部变量表存放了**编译期可知**的各种**基本数据类型**（boolean、byte、char、short、int、float、long、double）、**对象引用**（reference 类型，它不等同于对象本身，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）和 **returnAddress 类型**（指向了一条字节码指令的地址）。

64 位长度的 long 和 double 类型的数据会占用 2 个局部变量空间（Slot），其余的数据类型只占用 1 个。**局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，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**。

如果线程请求的栈深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，如果扩展时无法申请到足够的内存，就会抛出 **OutOfMemoryError** 异常。

本地方法栈（Native Method Stack）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。

变量类型	存储空间（位）	变量类型	存储空间（位）
boolean	未知	int	32
byte	8	float	32
char	16	long	64
short	16	double	64

Java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块。

所有的对象实例以及数组都要在堆上分配，但是随着 JIT（Just In Time）编译器的发展与逃逸分析技术逐渐成熟，**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是那么“绝对”了。

Java 堆可以处于**物理上不连续**的内存空间中，只要**逻辑上是连续**的即可。

-Xmx：最大堆容量　　-Xms：初始堆容量　　-XX:MaxPermSize：最大方法区容量

如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 **OutOfMemoryError** 异常。

方法区（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。

方法区别名：非堆（Non-Heap）、永久代（Permanent Generation）。

方法区**不需要连续的内存**，且可以选择固定大小或者可扩展，还**可以选择不实现垃圾收集**。

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（Runtime Constant Pool）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（Constant Pool Table），用于**存放编译期生成的各种字面量和符号引用**。

除了保存 Class 文件中描述的**符号引用**外，还会把翻译出来的**直接引用**也存储在运行时常量池中。

运行时常量池具备**动态性**，Java 语言并不要求常量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入方法区运行时常量池，运行期间也可能将新的常量放入池中，如 **String 类的 intern()方法**。

当常量池无法再申请到内存时会抛出 **OutOfMemoryError** 异常。

在 JDK1.4 中新加入了 **NIO** 类，引入了一种基于**通道（Channel）**与**缓冲区（Buffer）**的 I/O 方式，**它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作**。

本机直接内存的分配不会受到 Java 堆大小的限制，但会受到本机总内存大小以及处理器寻址空间的限制。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。

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一个类的符号引用，并且检查这个**符号引用**代表的类是否已被**加载**、**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**，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。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，中间放着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**”（Bump the Pointer）。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，那就没有办法简单地进行指针碰撞了，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实例，并更新列表上的记录，这种分配方式称为 “**空闲列表**”（Free List）。选择哪种分配方式由 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。因此，在使用 Serial、ParNew 等带 Compact 过程的收集器时，系统采用的分配算法是指针碰撞，而使用 CMS 这种基于 Mark-Sweep 算法的收集器时，通常采用空闲列表。

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。解决这个问题有两种方案，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 堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer， TLAB）**。哪个线程要分配内存，就在哪个线程的 TLAB 上分配，**只有 TLAB 用完并分配新的 TLAB 时，才需要同步锁定**。虚拟机是否使用 TLAB，可以通过-XX:+/-UseTLAB 参数来设定。

接下来，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**、**对象的哈希码**、**对象的 GC 分代年龄**等信息。这些信息存放在**对象的对象头**（Object Header）之中。根据虚拟机当前的运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置。

在上面的工作都完成之后，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（由字节码中是否跟随 **invokespecial** 指令所决定），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产生出来。

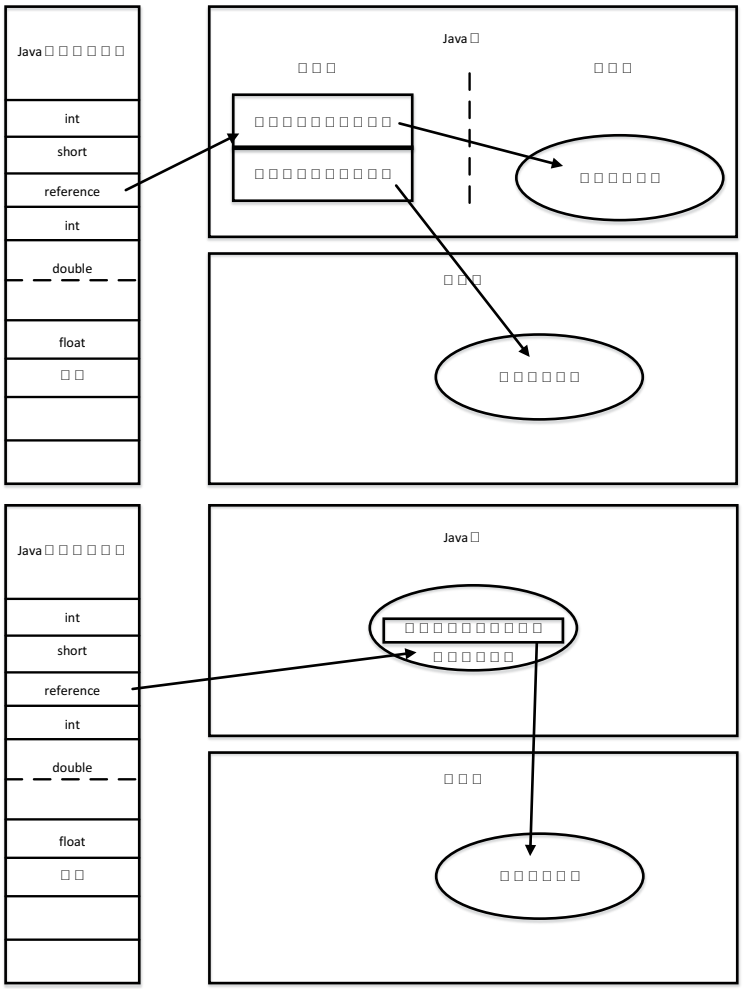
对象在内存中存储的布局可以分为 3 块区域：**对象头**（Object Header）、**实例数据**（Instance Data）和**对齐填充**（Padding）。

对象头包括两部分信息，第一部分用于存储**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，这部分数据的长度在 32 位和 64 位的虚拟机中分别为 32bit 和 64bit，官方称它为 “**Mark Word**”。Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。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。**查找对象的元数据信息并不一定要经过对象本身**（句柄）。**如果对象是一个 Java 数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通 Java 对象的元数据信息确定 Java 对象的大小，但是从数组的元数据中却无法确定数组的大小**。

实例数据部分是对象真正存储的有效信息，也是在程序代码中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，都需要记录起来**。

对齐填充仅仅起着**占位符**的作用，因为对象的大小必须是 8 字节的整数倍。



句柄访问：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。在对象被移动时只会改变句柄的实例数据指针，而 reference 本身不需要修改。

直接指针访问：Java 堆对象的布局中必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象地址。速度更快，节省了一次指针定位的时间开销。

通过参数**-XX:+HeapDumpOnOutOfMemoryError** 可以让虚拟机在出现内存溢出异常时 Dump 出当前的内存堆转储快照以便事后进行分析。

Java 堆内存的 OOM 异常是实际应用中常见的内存溢出异常情况。当出现 Java 堆内存溢出时，异常堆栈信息 “java.lang.OutOfMemoryError” 会跟着进一步提示 “Java heap space”。

要解决这个区域的异常，一般的手段是**先通过内存映像分析工具对 Dump 出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的**，也就是要先分清楚到底是出现了**内存泄漏**（Memory Leak）还是**内存溢出**（Memory Overflow）。

如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄漏对象是通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息及 GC Roots 引用链的信息，就可以比较准确地定位出泄漏代码的位置。

如果不存在泄漏，就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx 与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

String.intern()是一个 Native 方法，它的作用是：如果字符串常量池中已经包含一个等于此 String 对象的字符串，则返回代表池中这个字符串的 String 对象；否则，将此 String 对象包含的字符串添加到常量池中，并且返回此 String 对象的引用。**在 JDK1.6 中，intern()方法会把首次遇到的字符串实例复制到永久代中，返回的也是永久代中这个字符串实例的引用。在 JDK1.7 中，intern()实现不会再复制实例，只是在常量池中记录首次出现的实例引用。**

由 DirectMemory 导致的内存溢出，一个明显的特征是在 Heap Dump 文件中不会看见明显的异常，如果读者发现 OOM 之后 **Dump 文件很小**，而程序中又直接或间接使用了 NIO，那就可以考虑检查一下是不是这方面的原因。

在 Java 语言中，可作为 GC Roots 的对象包括下面几种：（1）**虚拟机栈（栈帧中的本地变量表）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（Java Native Interface）（即一般说的 Native 方法）引用的对象**。

强引用：Object obj = new Object();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。

软引用：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，才会抛出内存溢出异常。

弱引用：当垃圾收集器工作时，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。

虚引用：无法通过虚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得到一个系统通知**。

如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize()方法。当**对象没有覆盖 finalize()方法**，或者 **finalize()方法已经被虚拟机调用过**，虚拟机将这两种情况都视为

“没有必要执行”。

如果这个对象被判定为有必要执行 finalize()方法，那么这个对象将会放置在一个叫做 F-Queue 的队列之中，并在稍后由一个虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。**这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束**，这样做的原因是，**如果一个对象在 finalize()方法中执行缓慢，或者发生了死循环，将很可能会导致 F-Queue 队列中其他对象永久处于等待，甚至导致整个内存回收系统崩溃**。finalize()方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可。

任何一个对象的 finalize()方法都只会被系统自动调用一次。

类需要同时满足下面 3 个条件才能算是 “无用的类”：(1) **该类所有的实例都已经被回收，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.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**。

“**标记-清除**” 算法：**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。

复制算法：Eden：Survivor：Survivor=8：1：1。当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行**分配担保**（Handle Promotion）。如果另外一块 Survivor 空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。

为什么要有 Survivor 区：当 Eden 区对象满时，会触发一次 Minor GC，剩余存活的对象会进入 Survivor 区。而如果没有 Survivor 区，剩余存活的对象将会直接进入老年代。这样很快会导致老年代空间不够用，继而触发一次 Major GC（由于发生 Major GC 的同时常常伴随着 Minor GC，也可以看作是发生 Full GC）。由于老年代空间一般是新生代的 2 倍，Major GC 的耗时比起 Minor GC 是十分严重的。如果有了 Survivor 区，就可以保证对象只有在经过 16 次 Minor GC 之后才会进入老年代。

为什么要有两个 Survivor 区：设置两个 Survivor 区最大的好处是解决了**碎片化**。假设现在只有一个 Survivor 区，我们来模拟一下流程：刚刚新建的对象在 Eden 区，当 Eden 区满的时候，触发一次 Minor GC，Eden 区中剩余的存活对象会被转移至 Survivor 区。但是，等到下一次 Eden 区又满的时候，如果只有一个 Survivor 区，那么此时 Survivor 区和 Eden 区都存放有一部分对象，如果此时将 Eden 区的存活对象硬塞到 Survivor 区，很明显这两部分对象所占有的内存是不连续的，也就导致了内存碎片化。

“**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，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

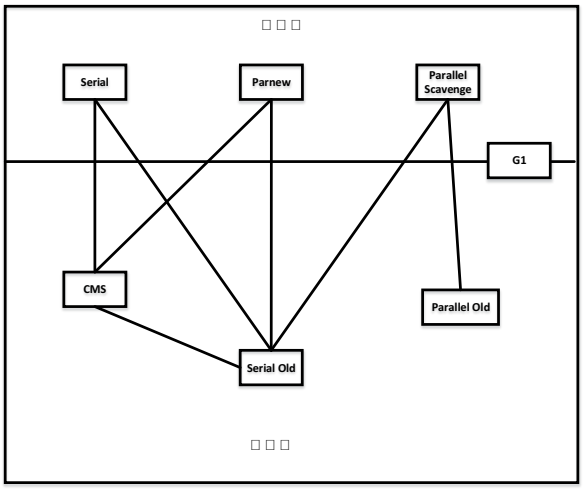
“**分代收集**” 算法：**新生代用复制算法，老年代用 “标记-清理” 或 “标记-整理” 算法**。

枚举根节点时：**Stop The World**。当系统停顿下来之后，并不需要一个不漏地检查完所有执行上下文和全局的引用位置，**虚拟机应当是有办法直接得知哪些地方存放着对象引用**。在 HotSpot 的实现中，是使用一组称为 **OopMap** 的数据结构来达到这个目的的。

HotSpot 没有为每条指令都生成 OopMap，只是在“特定的位置“记录了这些信息，这些位置称为**安全点**（SafePoint），即**程序执行时并非在所有地方都能停顿下来开始 GC，只有在到达安全点时才能暂停**。安全点的选定基本上是以程序“**是否具有让程序长时间执行的特征**”为标准进行选定——因为每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这个原因而过长时间运行，“长时间执行”的最明显特征就是指令序列复用，例如**方法调用**、**循环跳转**、**异常跳转**等，所以具有这些功能的指令才会产生 **SafePoint**。

如何在 GC 发生时让所有线程（这里不包括执行 JNI 调用的线程）都“跑”到最近的安全点上再停顿下来：**抢先式中断**和**主动式中断**。抢先式中断不需要线程的执行代码主动去配合，在 GC 发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。主动式中断是当 GC 需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起，**轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方**。

安全区域是指在一段代码片段之中，引用关系不会发生变化。在这个区域中的任意地方开始 GC 都是安全的。在线程执行到 Safe Region 中的代码时，首先标识自己已经进入了 Safe Region，那样，**当在这段时间里 JVM 要发起 GC 时，就不用管标识为 Safe Region 状态的线程了。在线程要离开 Safe Region 时，它要检查系统是否已经完成了根节点枚举（或者是整个 GC 过程）**，如果完成了，那线程就继续执行，否则它就必须等待直到收到可以安全离开 Safe Region 的信号为止。



Serial 收集器：单线程，它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作。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。**复制算法**。

ParNew 收集器：Serial 收集器的多线程版本，ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器更好的效果。

并行：真正意义上的同时执行。

并发：偏重于多个任务交替执行，而多个任务之间有可能还是串行的。

Parallel Scavenge 收集器：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 收集器的目标是达到一个可控制的吞吐量（Throughput）。**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）**

Serial Old 收集器：Serial 收集器的老年代版本。“**标记-整理**” 算法。

Parallel Old 收集器：多线程，“**标记-整理**” 算法。在注重吞吐量及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

CMS（Concurrent Mark Sweep）分为 4 个步骤：（1）**初始标记**，“Stop The World “，仅仅标记一下 GC Roots 能直接关联到的对象，速度快，单线程。（2）**并发标记**，进行 GC Roots Tracing 的过程，耗时。（3）**重新标记**，“Stop The World “，为了修正并发标记期间因用户程序继续运作而导致标记发生变动的那一部分对象的标记记录，多线程。（4）**并发清除**。

CMS 收集器的缺点：（1）**对 CPU 资源非常敏感**。当 CPU 在 4 个以上时，并发回收时垃圾收集线程不少于 25%的 CPU 资源，并且随着 CPU 数量的增加而下降。但是当 CPU 不足 4 个（譬如 2 个）时，CMS 对用户程序的影响就可能变得很大。（2）CMS 收集器无法处理**浮动垃圾**（Floating Garbage），可能出现“**Concurrent Mode Failure**”失败而导致另一次 Full GC 的产生。由于在垃圾收集阶段用户线程还需要运行，需要预留有足够的内存空间给用户线程使用，因此 **CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提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需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：**临时启用 Serial Old 收集器来重新进行老年代的垃圾收集**。（3）**大量空间碎片的产生**。

G1 收集器的特点：（1）**并行与并发**。（2）**分代收集**。（3）**空间整合**。G1 从整体来看是基于“标记-整理“算法实现的收集器，从局部（两个 Region 之间）来看是基于“复制”算法实现的。（4）**可预测的停顿**。

G1 将整个 Java 堆划分为**多个大小相等的独立区域（Region）**，虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region（不需要连续）的集合。

G1 的每个 Region 都有一个与之对应的 Remembered Set，虚拟机发现程序在对 Reference 类型的数据进行写操作时，会产生一个 **Write Barrier** 暂时中断写操作，检查 Reference 引用的对象是否处于不同的 Region 之中（在分代的例子中就是检查是否老年代中的对象引用了新生代中的对象），如果是，便**通过 CardTable 把相关的引用信息记录到被引用对象所属的 Region 的 Remembered Set 之中**。当进行内存回收时，在 GC 根节点的枚举范围中加入 Remembered Set 即可保证不对全堆扫描也不会有遗漏。

G1 收集器的流程：（1）**初始标记**，“Stop The World”，标记一下 GC Roots 能直接关联到的对象，单线程。（2）**并发标记**，从 GC Roots 开始对堆中的对象进行可达性分析。（3）**最终标记**，“Stop The World”，可并行。（4）**筛选回收**，首先**对各个 Region 的回收价值和成本进行排序**，根据用户所期望的 GC 停顿时间来制定回收计划。

对象优先在 Eden 分配。

大对象直接进入老年代。

长期存活的对象进入老年代。对象在 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。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入老年代，无须等待 MaxTenuringThreshold 中要求的年龄。

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，将尝试着进行一次 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。

新生代 GC（Minor GC）：发生在新生代的垃圾收集动作。

老年代 GC（Major GC / Full GC）：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。Major GC 的速度一般会比 Minor GC 慢 10 倍以上。

JDK 开发团队选择采用 Java 代码来实现监控工具是有特别用意的：当应用程序部署到生产环境后，**无论是直接接触物理服务器还是远程 Telnet 到服务器上都可能会受到限制**。借助 tools.jar 类库里面的接口，我们可以**直接在应用程序中实现功能强大的监控分析工具**。

名称	主要作用
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 Tool，用于收集 HotSpot 虚拟机各方面的运行时数据
jinfo	Configuration Info for Java，显示虚拟机配置信息
jmap	Memory Map for Java，生成虚拟机的内存转储快照（heapdump 文件）
jhat	JVM Heap Dump Browser，用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果
jstack	Stack Trace for Java，显示虚拟机的线程快照

线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如**线程间死锁、死循环、请求外部资源导致的长时间等待**等都是线程长时间停顿的常见原因。

任何一个 Class 文件都对应着唯一一个类或接口的定义信息，但反过来说，类或接口并不一定都得定义在文件里（譬如类或接口也可以通过类加载器直接生成）。

Class 文件是一组以 8 位字节为基础的二进制流，各个数据项目严格按照顺序紧凑地排列在 Class 文件之中，中间没有添加任何分隔符，这使得整个 Class 文件中存储的内容几乎全部是程序运行时的必要数据，没有空隙存在。

根据 Java 虚拟机规范的规定，Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：**无符号数**和**表**。

无符号数属于基本的数据类型，以 **u1**、**u2**、**u4**、**u8** 来分别代表 **1 个字节**、**2 个字节**、**4 个字节**和 **8 个字节**的无符号数，无符号数可以用来描述**数字**、**索引引用**、**数量值**或者**按照 UTF-8 编码构成字符串值**。

表是由多个**无符号数**或者其他**表**作为数据项构成的**复合**数据类型，所有表都习惯性地以“_info“结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上就是一张表。

类型	名称	数量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count - 1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

每个 Class 文件的头 4 个字节称为**魔数**（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的 Class 文件。0x**CAFEBABY**

紧接着魔数的 4 个字节存储的是 Class 文件的版本号 :第 5 和第 6 个字节是**次版本号**（Minor Version），第 7 和第 8 个字节是**主版本号**（Major Version）。

紧接着次主版本号之后的是常量池入口，常量池可以理解为 Class 文件之中的资源仓库，它是 Class 文件结构中与其他项目关联最多的数据类型，也是占用 Class 文件空间最大的数据项目之一，同时它还是在 Class 文件中第一个出现的表类型数据项目。

由于常量池中常量的数量是不固定的，所以在常量池的入口需要放置一项 u2 类型的数据，代表常量池容量计数值（constant_pool_count）。与 Java 语言习惯不一样的是，这个**容**

量计数是从 1 而不是 0 开始的。在 Class 文件格式规范制定之时，设计者将第 0 项常量空出来是有特殊考虑的，这样做的目的在于**满足后面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**，这种情况就可以把索引值置为 0 来标识。Class 文件结构中只有常量池的容量计数是从 1 开始，对于其他集合类型，包括接口索引集合、字段表集合、方法表集合等的容量计数都与一般习惯相同，是从 0 开始的。

常量池中主要存放两大类常量：**字面量**（Literal）和**符号引用**（Symbolic Reference）。**字面量**比较接近于 Java 语言层面的常量概念，如**文本字符串**、**声明为 final 的常量值**等。而**符号引用**则属于编译原理方面的概念，包括了下面三类常量：

（1）**类和接口的全限定名**

（2）**字段的名称和描述符**

（3）**方法的名称和描述符**

Java 代码在进行 Javac 编译时，并不像 C 和 C++那样有“连接”这一步骤，而是在虚拟机加载 Class 文件的时候进行动态拼接。在 Class 文件中不会保存各个方法、字段的最终内存布局信息，因此这些字段、方法的符号引用不经过运行期转换的话无法得到真正的内存入口地址，也就无法直接被虚拟机使用。**当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建时或运行时解析、翻译到具体的内存地址之中。**

常量池中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类型。

由于 Class 文件中的方法、字段等都需要引用 CONSTANT_Utf8_info 型常量来描述名称，所以 **CONSTANT_Utf8_info 型常量的最大长度 64KB 也就是 Java 方法、字段名的最大长度**。

在常量池结束之后，紧接着的两个字节代表**访问标志**（access_flags），这个标志用于识别一些类或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。

类索引、**父类索引**和**接口索引集合**都按顺序排列在访问标志之后，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 CONSTANT_Class_info 类型的常量中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。对于接口索引集合，入口的第一项——u2 类型的数据为接口计数器（interfaces_count），表示索引表的容量。如果该类没有实现任何接口，则该计数器值为 0，后面接口的索引表不再占用任何字节。

字段表（field_info）用于描述接口或者类中声明的变量。字段（field）包括类级变量以及实例级变量，但不包括方法内部声明的局部变量。**ACC_FINAL**、**ACC_VOLATILE** 不能同时选择。接口之中的字段必须有 **ACC_PUBLIC**、**ACC_STATIC**、**ACC_FINAL** 标志，这些都是由 Java 本身的语言规则所决定的。

根据**描述符**规则，**基本数据类型**（byte-B、char-C、boolean-Z、short-S、int-I、float-F、double-D、long-J）以及代表无返回值的 void（V）类型都用一个大写字母来表示，而对象类型则用字符 L 加对象的全限定名来表示。对于**数组类型**，每一维度将使用一个前置的“[”字符来描述，如一个定义为“java.lang.String[ ][ ]”类型的二维数组，将被记录为：“[[Ljava/lang/String”，一个整型数组“int[ ]”将被记录为“[I”。用描述符来**描述方法**时，按照先参数列表，后返回值的顺序描述，参数列表按照参数的严格顺序放在一组“()”之内。如方法 void inc() 的描述符为“()V”，方法 java.lang.String toString() 的描述符为

“()Ljava/lang/String”。

字段表集合中不会列出从超类或者父接口中继承而来的字段，但有可能列出原本 Java 代码之中不存在的字段，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。另外，**在 Java 语言中字段是无法重载的，两个字段的数据类型、修饰符不管是否相同，都必须使用不一样的名称，但是对于字节码来讲，如果两个字段的描述符不一致，那字段重名就是合法的**。

方法里的代码，经过编译器编译成字节码指令后，存放在方法属性表集合中一个名为 “Code” 的属性里面，属性表是 Class 文件格式中最具扩展性的一种数据项目。

如果父类方法在子类中没有被重写（Override），方法表集合中就不会出现来自父类的方法信息，但可能出现由编译器自动添加的方法，如类构造器 “<clinit>()” 方法和实例构造器 “<init>()” 方法。Java 代码的方法特征签名只包括了**方法名称**、**参数顺序**及**参数类型**，而字节码的特征签名还包括**方法返回值**以及**受查异常表**。

如果同时使用 final 和 static 来修饰一个变量，并且这个变量的数据类型是基本类型或者 java.lang.String 的话，就生成 **ConstantValue** 属性来进行初始化，如果这个变量没有被 final 修饰，或者并非基本类型及字符串，则将会选择在<clinit>()方法中进行初始化。

ConstantValue 的属性值只能限于基本类型和 String，因为此属性的属性值只是一个常量池的索引号，由于 **Class 文件格式的常量类型中只有与基本属性和字符串相对应的字面量**，所以计算 ConstantValue 属性想支持别的类型也无能为力。

StackMapTable 会在虚拟机类加载的字节码验证阶段被新类型检查验证器（Type Checker）使用，目的在于**代替以前比较消耗性能的基于数据流分析的类型推导验证器**。新的验证器在同样能保证 Class 文件合法性的前提下，省略了在运行期通过数据流分析去确认字节码的行为逻辑合法性的步骤，而是**在编译阶段将一系列的验证类型（Verification Types）直接记录在 Class 文件之中，通过检查这些验证类型代替了类型推导过程**，从而大幅提升了字节码验证的性能。**StackMapTable 属性中包含零至多个栈映射帧（Stack Map Frames），每个栈映射帧都显式或隐式地代表了一个字节码偏移量，用于表示执行到该字节码时局部变量表和操作数栈的验证类型**。类型检查验证器会通过检查目标方法的局部变量和操作数栈所需要的类型来确定一段字节码指令是否符合逻辑约束。

BootstrapMethods 用于**保存 invokedynamic 指令引用的引导方法限定符**。如果某个类文件结构的常量池中曾经出现过 CONSTANT_InvokeDynamic_info 类型的常量，那么这个类文件的属性表中必须存在一个明确的 BootstrapMethods 属性，另外，即使 CONSTANT_InvokeDynamic_info 类型的常量在常量池中出现过多次，类文件的属性表中最多也只能有一个 BootstrapMethods 属性。

在 Java 虚拟机中，处理异常（catch 语句）不是由字节码指令来实现的，而是采用**异常表**来实现的。

Java 虚拟机可以支持方法级的同步和方法内部一段指令序列的同步，这两种同步结构

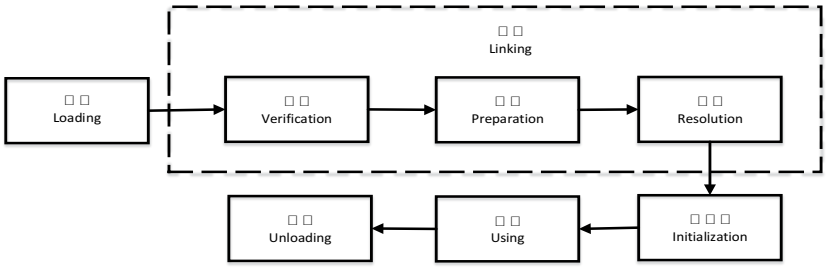
都是使用管程（Monitor）来支持的。

方法级的同步是隐式的，即无须通过字节码指令来控制，它实现在方法调用和返回操作之中。虚拟机可以从方法常量池的方法表结构中的 ACC_SYNCHRONIZED 访问标志得知一个方法是否声明为同步方法。当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置。如果设置了，执行线程就要求先成功持有管程，然后才能执行方法，最后当方法完成（无论是正常完成还是非正常完成）时释放管程。在方法执行期间，执行线程持有了管程，其他任何线程都无法再获取到同一个管程。如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那么这个同步方法所持有的管程将在异常抛到同步方法之外时自动释放。

同步一段指令集序列通常是由 Java 语言中的 synchronized 语句块来表示的，Java 虚拟机的指令集中有 monitorenter 和 monitorexit 两条指令来支持 synchronized 关键字的语义。编译器必须确保无论方法通过何种方式完成，方法中调用过的每条 monitorenter 指令都必须执行其对应的 monitorexit 指令，而无论这个方法是正常结束还是异常结束。

Java 虚拟机的 2 种实现形式：(1) 将输入的 Java 虚拟机代码在加载和执行时翻译成另外一种虚拟机的指令集。(2) 将输入的 Java 虚拟机代码在加载或执行时翻译成宿主机 CPU 的本地指令集（JIT 代码生成技术）。

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。在 Java 语言里，类型的加载、连接和初始化过程都是在程序运行期间完成的。



Java 虚拟机规范严格规定了有且只有 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 条指令的最常见的 Java 代码场景是：使用 new 关键字实例化对象的时候、读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

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

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

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

(5) 当使用 JDK1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandler 实例最后的解析结果是 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，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，所有引用类的方式都不会

触发初始化，称为**被动引用**。

通过子类引用父类的静态字段，不会导致子类初始化。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。

通过数组定义来引用类，不会触发此类的初始化。

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，因此不会触发定义常量的类的初始化。

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，只有在真正使用到父接口（如引用接口中定义的常量）才会初始化。

在**加载**阶段，虚拟机需要完成以下 3 件事情：

（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.lang.Class 对象，作为方法区这个类的各种数据的访问入口**。

验证是连接阶段的第一步，这一阶段的目的是为了确保 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。验证阶段会完成下面 4 个阶段的检验动作：**文件格式验证**、**元数据验证**、**字节码验证**、**符号引用验证**。

文件格式验证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，格式上符合描述一个 Java 类型信息的要求。这阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区中进行存储，所以后面的 3 个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流。

元数据验证的目的是对类的元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息。

字节码验证的目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。如果一个类方法体的字节码没有通过字节码验证，那肯定是有问题的；但如果一个方法体通过了字节码验证，也不能说明其一定就是安全的。由于数据流验证的高复杂性，虚拟机设计团队为了避免过多的时间消耗在字节码验证阶段，在 JDK1.6 之后的 Javac 编译器和 Java 虚拟机中进行了一项优化，给方法体的 Code 属性的属性表中增加了一项名为“**StackMapTable**”的属性，这项属性描述了**方法体中所有的基本块开始时本地变量表和操作栈应有的状态**，在字节码验证期间，就不需要根据程序推导这些状态的合法性，只需要检查 StackMapTable 属性中的记录是否合法即可。

符号引用验证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**，其目的是确保解析动作能正常进行。

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，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。这里所说的初始值“通常情况”下是数据类型的零值，public static int value = 123。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，public static **final** int value = 123。

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（Symbolic References）：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 文件格式之中。

直接引用（Direct References）：可以是**直接指向目标的指针、相对偏移量**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。如果有了直接引用，那引用的目标必定已经在内存中存在。

invokedynamic 指令的目的是用于动态语言支持的，它所对应的引用称为“**动态调用点限定符**”，这里“动态”的含义就是**必须等到程序实际运行到这条指令的时候，解析动作才能进行**。相对的，其余可触发解析的指令都是“静态”的，可以在刚刚完成加载阶段，还没有开始执行代码时就进行解析。

初始化阶段是执行类构造器<clinit>()方法的过程。

（1）<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**。

（2）<clinit>()方法与实例构造器<init>()方法不同，它不需要显式地调用父类构造器，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**。因此在虚拟机中第一个被执行的<clinit>()方法的类肯定是 java.lang.Object。

（3）由于父类的<clinit>()方法先执行，也就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。

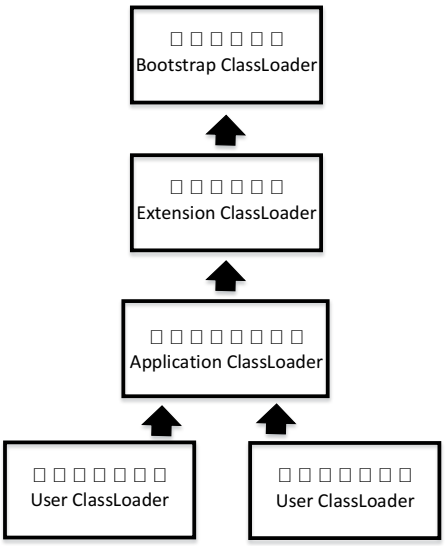
（4）<clinit>()方法对于类或接口来说并不是必需的，**如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>()方法**。

（5）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。只有当父接口中定义的变量使用时，父接口才会初始化**。另外，**接口的实现类在初始化时也一样不会执行接口的<clinit>()方法**。

（6）虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁、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，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。其他线程虽然会被阻塞，但如果执行<clinit>()方法的那条线程退出<clinit>()方法后，其他线程唤醒之后不会再次进入<clinit>()方法。同一个类加载器下，一个类型只会初始化一次**。

虚拟机设计团队把类加载阶段中“通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到 Java 虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“**类加载器**”。

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。



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己无法完成这个请求时，子类加载器才会尝试自己去加载。

使用双亲委派模型来组织类加载之间的关系，有一个显而易见的好处就是 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 java.lang.Object，它存放在 rt.jar 之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 **Object 类在程序的各种类加载器环境中都是同一个类**。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，并放在程序的 ClassPath 中，那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，应用程序也将会变得一片混乱。

双亲委派模型的实现：**先检查是否已经被加载过，若没有加载则调用父加载器的 loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载失败，抛出 ClassNotFoundException 异常后，再调用自己的 findClass()方法进行加载**。

双亲委派模型的第一次“被破坏”发生在双亲委派模型出现之前——JDK1.2 发布之前。由于双亲委派模型在 JDK1.2 之后才被引入，而类加载器和抽象类 java.lang.ClassLoader 则在 JDK1.0 时代就已经存在，面对已经存在的用户自定义类加载器的实现代码，Java 设计者引入双亲委派模型时不得不做出一些妥协。为了向前兼容，JDK1.2 之后的 java.lang.ClassLoader 添加了一个新的 protected 方法 **findClass()**，在此之前，用户去继承 java.lang.ClassLoader 的唯一目的就是为了重写 loadClass()方法，因为虚拟机在进行类加载的时候会调用加载器的私有方法 loadClassInternal()，而这个方法的唯一逻辑就是去调用自己的 loadClass()。JDK1.2 之后已不提倡用户再去覆盖 loadClass()方法，而应当**把自己的类加载逻辑写到 findClass()方法中，在 loadClass()方法的逻辑里如果父类加载失败，则会调用自己的 findClass()方法来完成加载**。

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调用的 API，如果基础类又要调用回用户的代码呢**？

为了解决这个问题，Java 设计团队引入了**线程上下文加载器**（Thread Context

ClassLoader）。这个类加载器可以通过 java.lang.Thread 类的 setContextClassLoader()方法进行设置，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态性的追求而导致的。在 OSGi 环境下，类加载器发展为复杂的**网状**结构。

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）的栈元素。栈帧存储了方法的**局部变量表**、**操作数栈**、**动态连接**和**方法返回地址**等信息。每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

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的影响**，而仅仅取决于**具体的虚拟机实现**。

对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为**当前栈帧**，与这个栈帧相关联的方法称为**当前方法**。执行引擎运行的所有字节码指令都只针对当前栈帧进行操作。

局部变量表（Local Variable Table）是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在 Java 程序编译为 Class 文件时，就在方法的 Code 属性的 max_locals 数据项中确定了该方法所需要分配的局部变量表的最大容量。

局部变量表的容量以变量槽（Variable **Slot**）为最小单位。一个 Slot 可以存放一个 32 位以内的数据类型，Java 中占用 32 位以内的数据类型有 boolean、byte、char、short、int、float、reference 和 returnAddress 8 种类型。对于 reference，虚拟机实现至少需要保证两点：（1）**从此引用中直接或间接地查找到对象在 Java 堆中的数据存放的起始地址索引**。（2）**此引用中直接或间接地查找到对象所属数据类型在方法区中的存储的类型信息**。returnAddress 指向了一条字节码指令的地址。

对于 64 位的数据类型，虚拟机会以高位对齐的方式为其分配两个连续的 Slot 空间。**由于局部变量表建立在线程的堆栈上，是线程私有的数据，无论读写两个连续的 Slot 是否为原子操作，都不会引起数据安全问题**。

虚拟机通过**索引定位**的方式使用局部变量表，索引值的范围是从 0 开始至局部变量表最大的 Slot 数量。如果访问的是 32 位数据类型的变量，索引 n 就代表了使用第 n 个 Slot，如果是 64 位数据类型的变量，则说明会同时使用 n 和 n + 1 两个 Slot。**对于两个相邻的共同存放一个 64 位数据的两个 Slot，不允许采用任何方式单独访问其中的某一个，Java 虚拟机规范中明确要求了如果遇到进行这种操作的字节码序列，虚拟机应该在类加载的校验阶段抛出异常**。

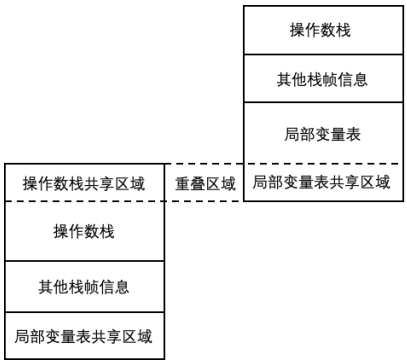
在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果执行的是实例方法，那**局部变量表中第 0 位索引的 Slot 默认是用于传递方法所属对象实例的引用**，在方法中可以通过“this“来访问到这个隐含的参数。其余参数则按照参数表顺序排列，占用从 1 开始的局部变量 Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序

和作用域分配其余的 Slot。

为了尽可能地节省栈空间，局部变量表中的 Slot 是可以重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法体，**如果当前字节码 PC 计数器的值已经超过了某个变量的作用域，那这个变量对应的 Slot 就可以交给其他变量使用**。

操作数栈的每一个元素可以是任意的 Java 数据类型，包括 long 和 double。32 位数据类型所占的栈容量为 1，64 位数据类型所占的栈容量为 2。

操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，在编译程序代码的时候，编译器要严格保证这一点，在类校验阶段的数据流分析中还要再次验证这一点。



在概念模型中，两个栈帧作为虚拟机栈的元素，是完全相互独立的。但在大多虚拟机的实现里都会做一些优化处理，令两个栈帧出现一部分重叠。**让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起**，这样在进行方法调用时就可以共用一部分数据，无须进行额外的参数复制传递。

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。Class 文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池中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或者第一次使用的时候就转化为直接引用，这种转化就称为**静态解析**。另外一部分将在每一次运行期间转化为直接引用，这部分称为**动态连接**。

方法正常退出时，调用者的 PC 计数器的值可以作为返回地址，栈帧中很可能会保存这个计数器值。而**方法异常退出时，返回地址是要通过异常处理器表来确定的**，栈帧中一般不会保存这部分信息。

方法退出的过程实际上就等同于把当前栈帧出栈，因此退出时可能执行的操作有：**恢复上层方法的局部变量表和操作数栈，把返回值（如果有的话）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**。

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入口地址（相当于之前说的直接引用），**需要在类加载时，甚至到运行期间才能确定目标方法的直接引用**。

所有方法调用中的目标方法在 Class 文件里面都是一个常量池中的符号引用，**在类加载的解析阶段，会将其中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：方法在程序真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。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 Java 语言中符合“**编译期可知，运行期不可变**”这个要求的方法，主要包括**静态方法**和**私有方法**两大类。

在 Java 虚拟机里，提供了 5 条方法调用指令：

（1） **invokestatic**：调用静态方法。

（2） **invokespecial**：调用实例构造器<init>方法、私有方法和父类方法。

（3） **invokevirtual**：调用所有的虚方法。

（4） **invokeinterface**：调用接口方法，会在运行时再确定一个实现此接口的对象。

（5） **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，在**此之前的 4 条调用指令，分派逻辑是固化在 Java 虚拟机内部的**，而 **invokedynamic 指令的分派逻辑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。这些方法称为**非虚方法**，与之相反，其他方法称为**虚方法**（除去 final 方法）。虽然 final 方法是使用 invokespecial 指令来调用的，但是由于它无法被覆盖，没有其他版本，所以也无须对方法接收者进行多态选择，又或者说多态选择的结果肯定是唯一的。在 Java 语言规范中明确说明了 **final 方法是一种非虚方法**。

解析调用一定是一个**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**调用则可能是静态的也可能是动态的。

虚拟机**在重载时是通过参数的静态类型而不是实际类型作为判据的**，并且静态类型是编译期可知的，因此，**在编译阶段，Javac 编译器会根据参数的静态类型决定使用哪个重载版本**。

invokevirtual 指令的运行时解析过程：

（1）找到操作数栈顶的第一个元素所指的**对象的实际类型**，记作 C。

（2）如果在类型 C 中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则返回 java.lang.IllegalAccessError 异常。

（3）否则，按照继承关系从下往上依次对 C 的各个父类进行第（2）步的搜索和验证过程。

（4）如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError 异常。

方法的接受者和**方法的参数**统称为**方法的宗量**。Java 是一门**静态多分派，动态单分派**的语言。

虚拟机动态分派的实现：

由于动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在类的方法元数据中搜索适合的目标方法。最常用的“稳定优化”手段就是为类在**方法区**中建立一个**虚方法表**（Virtual Method Table），使用虚方法表索引来代替元数据查找以提高性能。

虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类重写了这个方法，子类方法表中的地址将会替换为指向子类实现版本的入口。

为了程序实现上的方便，**具有相同签名的方法，在父类、子类的虚方法表中都应当具有一样的索引序号**，这样当类型变换时，仅需要变更查找的方法表，就可以从不同的方法虚方法表中按索引转换出所需的入口地址。

JDK1.7 实现了 JSR-292，新加入的 java.lang.invoke 包就是 JSR-292 的一个重要组成部分，这个包的主要目的是在之前单纯依靠符号引用来确定调用的目标方法这种方式以外，提供一种新的动态调用目标方法的机制，称为 **MethodHandle**。**MethodHandle 可以视为对最终调用方法的一个“引用”**。

从 Java 语言的角度来看，MethodHandle 的使用方法与 Reflection 有众多相似之处，不过，它们还是有以下这些区别：

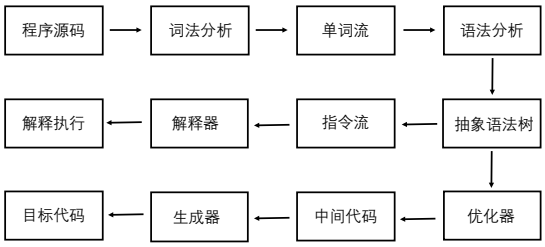
（1）从本质上讲，Reflection 和 MethodHandle 机制都是在模拟方法调用，但 **Reflection 是在模拟 Java 代码层次的方法调用**，而 **MethodHandle 是在模拟字节码层次的方法调用**。在 MethodHandles.lookup 中的 3 个方法——**findStatic()、findVirtual()、findSpecial()**正是为了对应于 invokestatic、invokevirtual & invokeinterface 和 invokespecial 这几条字节码指令的执行权限校验行为，而这些底层细节是在使用 Reflection API 时不需要关心的。

（2）Reflection 中的 java.lang.reflect.Method 对象远比 MethodHandle 机制中的 java.lang.invoke.MethodHandle 对象所包含的信息多。前者是方法在 Java 一端的全面映像，包含了方法的签名、描述符以及方法属性表中各种属性的 Java 端表示方式，还包含执行权限等的运行期信息。而后者仅仅包含与执行该方法相关的信息。用通俗地话来讲，**Reflection 是重量级，而 MethodHandle 是轻量级**。

（3）由于 MethodHandle 是对字节码的方法指令调用的模拟，所以理论上**虚拟机在这方面所做的各种优化（如方法内联），在 MethodHandle 上也应当可以采用类似思路去支持**。而通过反射去调用方法则不行。

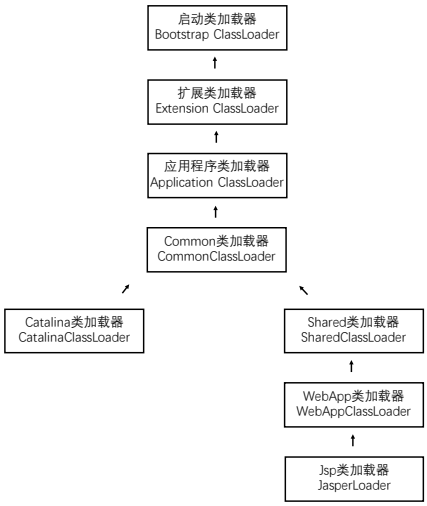
Reflection API 的设计目标是只为 Java 语言服务的，而 **MethodHandle 则设计成可服务于所有 Java 虚拟机之上的语言，其中也包括 Java 语言**。

每一处含有 invokedynamic 指令的位置都称作“**动态调用点**”，这条指令的第一个参数不再是代表方法符号引用的 CONSTANT_Methodref_info 常量，而是为 JDK1.7 新加入的 **CONSTANT_InvokeDynamic_info** 常量，从这个常量可以得到 3 项信息：**引导方法**（Bootstrap Method，此方法存放在新增的 BootstrapMethods 属性中）、**方法类型**（MethodType）和**名称**。**引导方法是有固定的参数，并且返回值是 java.lang.invoke.CallSite 对象，这个代表真正要执行的目标方法调用**。根据 CONSTANT_InvokeDynamic_info 常量中提供的信息，虚拟机可以找到并且执行引导方法，从而获得一个 CallSite 对象，最终调用要执行的目标方法。



基于栈的指令集主要的优点就是**可移植**，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。缺点是执行速度相对来说会**缓慢**一些。

栈顶缓存：把最常用的操作映射到寄存器中避免直接内存访问。



在 Tomcat 目录结构中，有 3 组目录（“/common/*”、“/server/*” 和 “/shared/”）可以存放 Java 类库，另外还可以加上 Web 应用程序自身的目录 “/WEB-INF/*”，一共 4 组，把 Java 类库放置在这些目录中的含义分别如下。

（1）放置在/common 目录中：类库可被 Tomcat 和所有 Web 应用程序共同使用。

（2）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。

（3）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。

（4）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。

WebApp 类加载器和 Jsp 类加载器通常会存在多个实例，**每一个 Web 应用程序对应一个 WebApp 类加载器**，**每一个 JSP 文件对应一个 Jsp 类加载器**。

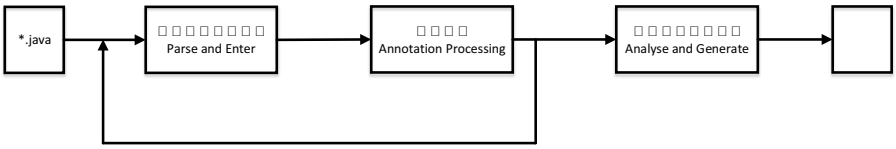
如果有 10 个 Web 应用程序都是用 Spring 来进行组织和管理的话，可以把 Spring 放到 Common 或 Shared 目录下让这些程序共享。Spring 要对用户程序的类进行管理，自然要能访问到用户程序的类，而用户的程序显然是放在/WebApp/WEB-INF 目录中的，那么被 CommonClassLoader 或 SharedClassLoader 加载的 Spring 如何访问并不在其加载范围内的用户程序呢？**线程上下文类加载器**。

在 OSGi（Open Service Gateway Initiative）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**平级模块之间的依赖**，一个模块里只有被 Export 过的 Package

才可能被外界访问，其他的 Package 和 Class 将会被隐藏起来。基于 OSGi 的程序很可能可以实现模块级的热插拔功能，当程序升级更新或调试出错时，可以只停用、重新安装然后启用程序的其中一部分。

JDK 动态代理：最后会调用 sun.misc.**ProxyGenerator.generateProxyClass()**方法来完成生成字节码的动作。

许多新生的 Java 语法特性，都是靠编译器的 “语法糖” 来实现，而不是依赖虚拟机的底层改进来支持，Java 中即时编译器在运行期的优化过程对于程序运行来说更重要，而前端编译器在编译期的优化过程对于程序编码来说关系更加密切。



编译过程大致分为 3 个过程，分别是：(1) **解析与填充符号表过程**。(2) **插入式注解处理器的注解处理过程**。(3) **语义分析与字节码生成过程**。

解析步骤包括了经典程序编译原理中的**词法分析**和**语法分析**两个过程。词法分析是将源代码的字符流转变为**标记**（Token）集合，单个字符是程序编写过程的最小元素，而标记则是编译过程的最小元素，**关键字**、**变量名**、**字面量**、**运算符**都可以成为标记。语法分析是根据 Token 序列构造抽象语法树的过程，**抽象语法树**（Abstract Syntax Tree）是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构，例如**包**、**类型**、**修饰符**、**运算符**、**接口**、**返回值**甚至**代码注释**都可以是一个语法结构。经过这个步骤之后，编译器就基本不会再对源码文件进行操作了，**后续的操作都建立在抽象语法树之上**。

完成了词法分析和语法分析之后，下一步就是**填充符号表**的过程。符号表（Symbol Table）是一组**符号地址**和**符号信息**构成的表格。符号表中所登记的信息在编译的不同阶段都要用到。**在语义分析中，符号表所登记的内容将用于语义检查**（如检查一个名字的使用和原先的说明是否一致）**和产生中间代码**。**在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据**。

JDK1.6 提供了一组插入式注解处理器的标准 API 在编译期间对注解进行处理，我们可以把它看作是一组编译器的插件，在这些插件里面，可以读取、修改、添加抽象语法树中的任意元素。**如果这些插件在处理注解期间对语法树进行了修改，编译器将回到解析及填充符号表的过程重新处理，直到所有插入式注解处理器都没有再对语法树进行修改为止**。

语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的。而**语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查**，如进行类型审查。

语义分析过程分为**标注检查**以及**数据及控制流分析**两个步骤。

标注检查步骤检查的内容诸如**变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配**等。在标注检查中，还有一个重要的动作称为**常量折叠**，在代码里面定义“a=1+2”比起直接定义“a=3”，并不会增加程序运行期哪怕一个 CPU 指令的运算量。

数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如**程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了**等问题。编译时期的数据及控制流分析与类加载时的数据及控制流分析的目的基本上是一致的，但校验范围有所区别，有一些校验项只有在编译期或运行期才能进行。

局部变量与字段（实例变量、类变量）是有区别的，它在常量池中没有 CONSTANT_Fieldref_info 符号引用，自然就没有访问标志（Access_Flags）的信息，甚至可能连名称都不会保留下来，自然在 Class 文件中不可能知道一个局部变量是不是声明为 final 了。因此，**将局部变量声明为 final，对运行期是没有影响的，变量的不变性仅仅由编译器在编译期间保障**。

虚拟机在运行时不支持变长参数、自动装箱/拆箱等语法，它们在编译阶段还原回简单的基础语法结构，这个过程称为**解语法糖**。

字节码生成是 Javac 编译过程的最后一个阶段，不仅仅把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。例如，实例构造器<init>()和类构造器<clinit>()就是在这个阶段添加到语法树之中的，这两个构造器的产生过程实际上是一个代码收敛的过程，编译器会把语句块（对于实例构造器而言是“{}”块，对于类构造器而言是“static{}”块）、变量初始化（实例变量和类变量）、调用父类的实例构造器（仅仅是实例构造器，<clinit>()方法中无须调用父类的<clinit>()方法，虚拟机会自动保证父类构造器的执行）等操作收敛到<init>()和<clinit>()方法之中，并且保证一定是按**先执行父类的实例构造器，然后初始化变量，最后执行语句块**的顺序进行。

Java 语言中的泛型只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此，对于运行期的 Java 语言来说，ArrayList<Integer>与 ArrayList<String>即是同一个类，所以泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为**类型擦除**，基于这种方法实现的泛型称为伪泛型。

方法重载要求方法具备不同的特征签名，返回值并不包含在方法的特征签名之中，所以返回值不参与重载选择，但是在 Class 文件格式之中，只要描述符不是完全一致的两个方法就可以共存。也就是说，两个方法如果有相同的名称和特征签名，但返回值不同，那它们也可以合法地共存于一个 Class 文件中。

Signature 的作用是存储一个方法在**字节码层面的特征签名**，这个属性中保存的参数类型并不是原生类型，而是包括了**参数化类型**的信息。

在 Java 代码中的方法特征签名只包括了**方法名称**、**参数顺序**及**参数类型**，而在字节码中的特征签名还包括**方法返回值**及**受查异常表**。

擦除法所谓的擦除，仅仅是**对方法的 Code 属性中的字节码进行擦除**，实际上**元数据还是保留了泛型信息**，这也是**我们能通过反射手段取得参数化类型**的根本依据。

包装类的“==”运算在不遇到算术运算的情况下不会自动拆箱，以及它们 **equals()方法不处理数据转型的关系**。

编译器并非一个个地编译 Java 文件，而是**将所有编译单元的语法树顶级节点输入到待处理列表后再进行编译**，因此各个文件之间能够互相提供符号信息。

Java 程序最初是通过解释器进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁时，就会把这些代码认定为“**热点代码**”（Hot Spot Code）。为了提高热点代码的执行效率，在运行时，虚拟机会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器（**JIT 编译器**）。

解释器与编译器两者各有优势：当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。当程序运行环境中内存资源限制较大（如部分嵌入式系统中），可以使用解释执行节约内存，反之可以使用编译执行来提升效率。同时，解释器还可以作为编译器激进优化时的一个“**逃生门**”，让编译器根据概率选择一些大多数时候都能提升运行速度的优化手段，当激进优化的假设不成立，如**加载了新类后类型继承结构出现变化**、出现“**罕见陷阱**”（Uncommon Trap）时可以通过**逆优化**（Deoptimization）退回到解释状态继续执行。

HotSpot 虚拟机中内置了两个即时编译器，分别称为 Client Compiler 和 Server Compiler，或者称为 **C1 编译器**和 **C2 编译器**（也叫 Opto 编译器）。目前主流的 HotSpot 虚拟机中，默认采用解释器与其中一个编译器直接配合的方式工作，程序使用哪个编译器，取决于虚拟机运行的模式，HotSpot 虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用“-client”或“-server”参数去强制指定虚拟机运行在 Client 模式或 Server 模式。

由于即时编译器编译本地代码需要占用程序运行时间，要编译出优化程度更高的代码，所花费的时间可能更长；而且想要编译出优化程度更高的代码，**解释器可能还要替编译器收集性能监控信息**，这对解释执行的速度也有影响。为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot 虚拟机还会逐渐启用**分层编译**的策略：

第 0 层，程序解释执行，解释器不开启性能监控功能（Profiling），可触发第 1 层编译。

第 1 层，也称为 C1 编译，将字节码编译为本地代码，进行**简单、可靠的优化**，如有必要将加入性能监控的逻辑。

第 2 层（或 2 层以上），也称为 C2 编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

实施分层编译后，Client Compiler 和 Server Compiler 将会同时工作，许多代码都可能会被多次编译，用 Client Compiler 获取更高的编译速度，用 Server Compiler 来获取更好的编译质量，在解释执行的时候也无须再承担收集性能监控信息的任务。

在运行过程中会被即时编译器编译的“热点代码”有两类：(1) **被多次调用的方法**。(2) **被多次执行的循环体**。

对于多次调用的方法，由于是由方法调用触发的编译，因此编译器理所当然地会以整个

方法作为编译对象，这种编译也是虚拟机中**标准的 JIT 编译方式**。对于多次执行的循环体，尽管编译动作是由循环体所触发的，但编译器仍然会以整个方法（而不是单独的循环体）作为编译对象。这种编译方式因为编译发生在方法执行过程之中，因此形象地称为**栈上替换**（On Stack Replacement，简称为 **OSR 编译**，即**方法栈帧还在栈上，方法就被替换了**）。

判断一段代码是不是热点代码，是不是需要触发即时编译，这样的行为称为**热点探测**（Hot Spot Detection），目前主要的热点探测判定方式有两种，分别如下：

基于**采样**的热点探测（Sample Based Hot Spot Detection）：采用这种方法的虚拟机会**周期性地检查各个线程的栈顶**，如果发现某个方法经常出现在栈顶，那这个方法就是“热点方法”。基于采样的热点探测的好处是实现简单、高效，还可以很容易地获取方法调用关系（将调用堆栈展开即可），缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。

基于**计数器**的热点探测（Counter Based Hot Spot Detection）：采用这种方法的虚拟机会**为每个方法（甚至是代码块）建立计数器，统计方法的执行次数**，如果执行次数超过一定的阈值就认为它是“热点方法”。这种统计方法实现起来麻烦一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对来说更加精确和严谨。

在 HotSpot 虚拟机中使用的是基于计数器的热点探测方法，它为每个方法准备了两类计数器：**方法调用计数器**（Invocation Counter）和**回边计数器**（Back Edge Counter）。在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，**当计数器超过阈值溢出了，就会触发 JIT 编译**。

当一个方法被调用时，会**先检查该方法是否存在被 JIT 编译过的版本**，如果存在，则优先使用编译后的本地代码来执行。**如果不存在已被编译过的版本，则将此方法的调用计数器值加 1，然后判断方法调用计数器与回边计数器之和是否超过方法调用计数器的阈值**。如果已超过阈值，那么将会向即时编译器提交一个该方法的代码编译请求。

如果不做任何设置，执行引擎并不会同步等待编译请求的完成，而是继续进入解释器按照解释方式执行字节码，直到提交的请求被编译器编译完成。当编译工作完成之后，这个方法的调用入口地址就会被系统自动改写成新的，下一次调用该方法时就会使用已编译的版本。

如果不做任何设置，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的**执行频率**，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器的**衰减**（Counter Decay），而这段时间就称为此方法统计的**半衰周期**（Counter Half Life Time）。**进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的**。

回边计数器的作用是统计一个方法中循环体代码执行的次数，**在字节码中遇到控制流向后跳转的指令称为“回边”（Back Edge）**。建立回边计数器统计的目的是为了触发 **OSR 编译**。

当解释器遇到一条回边指令时，会先查找将要执行的代码片段是否有已经编译好的版本，如果有，它将会优先执行已编译的大门，否则就把回边计数器的值加 1，然后**判断方法调用计数器与回边计数器值之和是否超过回边计数器的阈值**。当超过阈值的时候，将会提交一个 OSR 请求，并且把回边计数器的值降低一些，以便继续在解释器中执行循环，等待编译器输出编译结果。

与方法计数器不同，回边计数器没有计数热度衰减的过程，因此这个计数器统计的就是该方法循环执行的绝对次数。**当计数器溢出的时候，它还会把方法计数器的值也调整到溢出状态**，这样下次再进入该方法的时候就会执行**标准编译过程**。

方法内联的重要性要高于其他优化措施，它的主要目的有两个，一是**去除方法调用的成本（如建立栈帧等）**，二是**为其他优化建立良好的基础**，方法内联膨胀之后可以便于在更大范围上采取后续的优化手段，从而获取更好的优化效果。因此，各种**编译器一般都会把方法内联优化放在优化序列的最靠前位置**。

公共子表达式消除：如果一个表达式 E 已经计算过了，并且从先前的计算到现在 E 中所有变量的值都没有发生变化，那么 E 的这次出现就成为了公共子表达式。对于这种表达式，没有必要花时间再对它进行计算，只需要直接用前面计算过的表达式结果代替 E 就可以了。

方法内联的优化行为看起来很简单，不过是把目标方法的代码“复制”到发起调用的方法之中，避免发生真实的方法调用而已。但实际上 Java 虚拟机中的内联过程远远没有那么简单，因为如果不是即时编译器做了一些特别的努力，按照经典编译原理的优化理论，大多数的 Java 方法都无法进行内联。

只有使用 invokespecial 指令调用的私有方法、实例构造器、父类方法以及使用 invokestatic 指令进行调用的静态方法才是在编译期进行解析的，除了上述 4 种方法之外，其他的 Java 方法调用都需要在运行时进行方法接收者的多态选择，并且都有可能存在多于一个版本的方法接收者（最多再**除去被 final 修饰的方法**这种特殊情况，尽管它使用 invokevirtual 指令调用，但也是非虚方法，Java 语言规范中明确说明了这点），简而言之，Java 语言中默认的实例方法是虚方法。对于一个虚方法，编译器做内联的时候根本无法确定应该使用哪个方法版本。

为了解决虚方法的内联问题，Java 虚拟机设计团队想了很多办法，首先是引入了一种名为“**类型继承关系分析**”（Class Hierarchy Analysis，CHA）的技术，这是**一种基于整个应用程序的类型分析技术，它用于确定在目前已加载的类中，某个接口是否有多于一种的实现，某个类是否存在子类、子类是否为抽象类等信息**。

编译器在进行内联时，如果是非虚方法，那么直接进行内联就可以了，这时候的内联是有稳定前提保障的。如果遇到虚方法，则会**向 CHA 查询此方法在当前程序下是否有多个目标版本可供选择，如果查询结果只有一个版本，那也可以进行内联，不过这种内联就属于激进优化**，需要预留一个“**逃生门**”，称为**守护内联**。如果程序的后续执行过程中，虚拟机一直没有加载到会令这个方法的接收者的继承关系发生变化的类，那这个内联优化的代码就可以一直使用下去。但**如果加载了导致继承关系发生变化的新类，那就需要抛弃已经编译的代码，退回到解释状态执行，或者重新进行编译**。

如果向 CHA 查询出来的结果是有多个版本的目标方法可供选择，则编译器还将会进行最后一次努力，使用**内联缓存**（Inline Cache）来完成方法内联，这是一个建立在目标方法正常入口之前的缓存，它的工作原理大致是：**在未发生方法调用之前，内联缓存状态为空，当第一次调用发生后，缓存记录下方法接收者的版本信息，并且每次进行方法调用时都比较接收者版本，如果以后进来的每次调用的方法接收者版本都是一样的，那这个内联还可以一直用下去。如果发生了方法接收者不一致的情况，就说明程序真正使用了虚方法的多态特性，这时才会取消内联，查找虚方法表进行方法分派**。

逃逸分析（Escape Analysis）是目前 Java 虚拟机中比较前沿的优化技术，它与类型继承关系分析一样，并不是直接优化代码的手段，而是**为其他优化手段提供依据的分析技术**。逃逸分析的基本行为就是**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**。甚至还有可能**被外部线**

程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，称为**线程逃逸**。

如果能证明一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问到这个对象，则可能为这个变量进行一些高效的优化，如下所示。

栈上分配（Stack Allocation）：如果确定一个对象不会逃逸出方法之外，那让这个对象在栈上分配内存将会是一个很不错的主意，**对象所占用的内存空间就可以随栈帧出栈而销毁**。在一般应用中，不会逃逸的局部对象所占的比例很大，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集系统的压力就会小很多。

同步消除（Synchronization Elimination）：线程同步本身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，那这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以消除掉。

标量替换（Scalar Replacement）：**标量**（Scalar）是指一个数据已经无法再分解成更小的数据表示了，Java 虚拟机中的原始数据类型（**int、long 等数据类型及 reference 类型**等）都不能再进一步分解，它们就可以称为标量。相对的，如果一个数据可以继续分解，那它就称作**聚合量**（Aggregate），**Java 中的对象**就是最典型的聚合量。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**就叫做**标量替换**。如果**逃逸分析能证明一个对象不会被外部访问，并且这个对象可以被拆散的话**，那**程序真正执行的时候将可能不创建这个对象，而改为直接创建它的若干个被这个方法使用到的成员变量来代替**。将对象拆分后，除了可以**让对象的成员变量在栈上分配和读写**之外，还可以为后续进一步的优化手段创建条件。

Java 虚拟机的即时编译器与 C/C++的静态编译器相比，可能会由于下列这些原因而导致输出的本地代码有一些劣势：

第一，**因为即时编译器运行占用的是用户程序的运行时间，具有很大的时间压力，它能提供的优化手段也严重受制于编译版本**。如果编译速度不能达到要求，那用户将在启动程序或程序的某部分察觉到重大延迟，这点使得即时编译器不敢随便引入大规模的优化技术，而编译的时间成本在静态优化编译器中并不是主要的关注点。

第二，**Java 语言是动态的类型安全语言，这就意味着需要由虚拟机来确保程序不会违反语言语义或访问非结构化内存**。从实现层面上看，这就意味着虚拟机必须频繁地进行动态检查，如实例方法访问时检查空指针、数组元素访问时检查上下界范围、类型转换时检查继承关系等。对于这类程序代码没有明确写出的检查行为，尽管编译器会努力进行优化，但总体上仍然要消耗不少的运行时间。

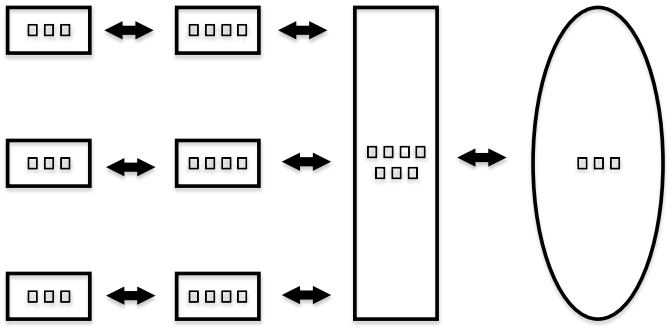
第三，Java 语言中虽然没有 virtual 关键字，但是使用虚方法的频率却远远大于 C/C++语言，这意味着**运行时对方法接收者进行多态选择的频率要远远大于 C/C++语言**，也意味着即时编译器在进行一些优化时的难度要远大于 C/C++的静态优化编译器。

第四，Java 语言是可以**动态扩展**的语言，**运行时加载新的类可能改变程序类型的继承关系，这使得很多全局的优化都难以进行**，因为编译器无法看见程序的全貌，许多全局的优化措施都只能以激进优化的方式来完成，编译器不得不时刻注意并随着类型的变化而在运行时撤销或重新进行一些优化。

第五，**Java 语言中对象的内存分配都是在堆上进行的，只有方法中的局部变量才能在栈上分配**。而 C/C++的对象则有多种内存分配方式，既可能在堆上分配，又可能在栈上分配，如果可以在栈上分配线程私有的对象，将减轻内存回收的压力。另外，C/C++主要由用户程序代码来回收分配的内存，这就不存在无用对象筛选的过程，因此效率上也比垃圾收集机制要高。

Java 编译器的一个红利是由它的动态性所带来的，由于 C/C++编译器所有优化都在编译期完成，**以运行期性能监控为基础的优化措施**它都无法进行，如**调用频率预测**（Call

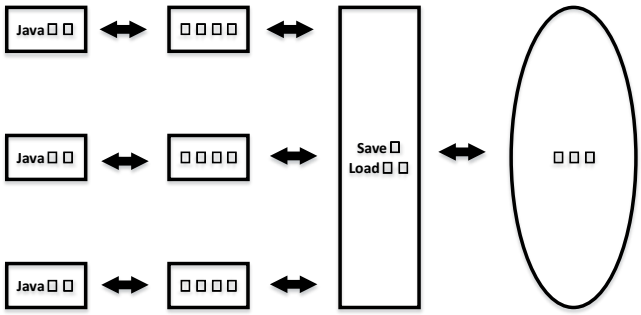
Frequency Prediction)、**分支频率预测**（Branch Frequency Prediction)、**裁剪未被选择的分支**（Untaken Branch Pruning）等，这些都会成为 Java 语言独有的性能优势。



由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了。

缓存一致性：在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存，当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致，如果真的发生这种情况，那同步回到主内存时以谁的缓存数据为准呢？

为了使得处理器内部的运算单元能尽量被充分利用，处理器可能会对输入代码进行**乱序执行**优化，处理器会在计算之后将乱序执行的结果重组，**保证该结果与顺序执行的结果是一致的**，但并不保证程序中各个语句计算的先后顺序与输入代码中的顺序一致，因此，如果存在一个计算任务依赖另外一个计算任务的中间结果，那么其顺序性并不能靠代码的先后顺序来保证。



Java 虚拟机规范试图定义一种 Java 内存模型（Java Memory Model， JMM）来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。

如果局部变量是一个 reference 类型，它引用的对象在 Java 堆中可被各个线程共享，但是 reference 本身在 Java 栈的局部变量表中，它是线程私有的。

Java 内存模型规定了所有的变量都存储在**主内存**中。每条线程还有自己的**工作内存**，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，**线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方**

工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

lock	作用于主内存的变量，它把一个变量标识为一条线程独占的状态。
unlock	作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
read	作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的 load 动作使用。
load	作用于工作内存的变量，它把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。
use	作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时会执行这个操作。
assign	作用于工作内存中的变量，它把一个执行引擎收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
store	作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的 write 操作使用。
write	作用于主内存的变量，它把 store 操作从工作内存中得到的变量的值放入主内存的变量中。

由于 volatile 变量只能保证可见性，在不符合以下两条规则的运算场景中，我们仍然要通过加锁来保证原子性。(1) **运算结果并不依赖变量的当前值**，或者能够**确保只有单一的线程修改变量的值**。(2) **变量不需要与其他的状态变量共同参与不变约束**。

内存屏障：重排序时不能把后面的指令重排序到内存屏障之前。

volatile 变量的特殊规则：(1) **每次使用 V 前都必须从主内存刷新最新的值，用于保证能看见其他线程对变量 V 所做的修改后的值**。(2) **每次修改 V 后都必须立刻同步回主内存中，用于保证其他线程可以看到自己对变量 V 所做的修改**。(3) **volatile 修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同**。

long 和 double 的非原子性协定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作划分为两次 32 位的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性。

Java 内存模型是围绕着在并发过程中如何处理**原子性**、**可见性**和**有序性**这 3 个特征来建立的。

原子性：由 Java 内存模型来直接保证的原子性变量操作包括 read、load、assign、use、store、write，我们大致可以认为**基本数据类型的访问读写是具备原子性的**。

如果应用场景需要一个更大范围的原子性保证，Java 内存模型还提供了 lock 和 unlock 操作来满足这种需求，尽管虚拟机未把 lock 和 unlock 操作直接开放给用户使用，但是却提供了更高层次的字节码指令 **monitorenter** 和 **monitorexit** 来隐式地使用这两个操作，这两个字节码指令反映到 Java 代码中就是同步块——**synchronized** 关键字，因此在 synchronized 块之间的操作也具备原子性。

可见性：当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。**Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性的**，无论是普通变量还是 volatile 变量都是如此，普通变量与 volatile 变量的区别是，**volatile 的特殊规则保证了新值能立即同步到主内存，以及每次用前立即从主内存刷新**。因此，**volatile 保证了多线程操作时变量的可见性**，而普通变量则不能保证这一点。

除了 volatile 之外，Java 还有 2 个关键字能实现可见性，即 **synchronized** 和 **final**。同步块的可见性是由“**对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存中**”这条规则获得的，而 final 关键字的可见性是指：**被 final 修饰的字段在构造器中一旦初始化完成，并且构造器没有把“this”的引用传递出去，那在其他线程中就能看见 final 字段的值**。

有序性：如果在本线程内观察，所有的操作都是有序的（线程内表现为串行的语义）；如果在一个线程中观察另一个线程，所有的操作都是无序的（指令重排序、工作内存与主内存同步延迟）。

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 关键字本身就包含了禁止指令重排序的语义，而 synchronized 则是由“**一个变量在同一时刻只允许一条线程对其进行 lock 操作**”这条规则获得的。

先行发生是 Java 内存模型中定义的**两项操作之间的偏序关系**，如果说操作 A 先行发生于操作 B，其实就是说在发生操作 B 之前，**操作 A 产生的影响能被操作 B 观察到**，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等。

happens-before 原则：

（1）**程序次序**规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说，应该是**控制流顺序**而不是程序代码顺序，因为要考虑分支、循环等结构。

（2）**管程锁定**规则：**一个 unlock 操作先行发生于后面对同一个锁的 lock 操作**。这里必须强调的是同一个锁，而“后面”是指时间上的先后顺序。

（3）**volatile 变量**规则：**对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作**，这里的“后面”同样指时间上的先后顺序。

（4）**线程启动**规则：**Thread 对象的 start()方法先行发生于此线程的每一个动作**。

（5）**线程终止**规则：**线程中的所有操作都先行发生于对此线程的终止检测**，我们可以通过 Thread.join()方法结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。

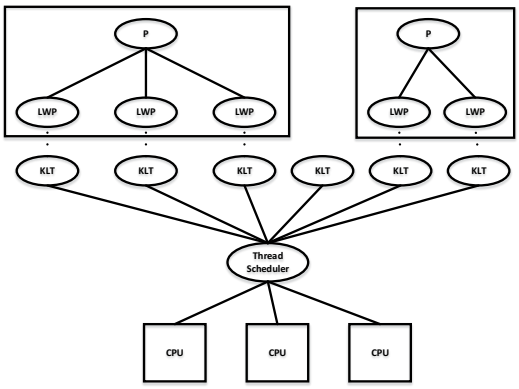
（6）**线程中断**规则：**对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生**，可以通过 Thread.interrupted()方法检测到是否有中断发生。

（7）**对象终结**规则：**一个对象的初始化完成（构造函数执行结束）先行发生于它的 finalize()方法的开始**。

（8）**传递性**：**如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C**。

一个操作“时间上的先发生”不代表这个操作会是“先行发生”；一个操作“先行发生”也不能代表这个操作是“时间上的先发生”。**时间先后顺序与先行发生原则之间基本没有太大的关系**。

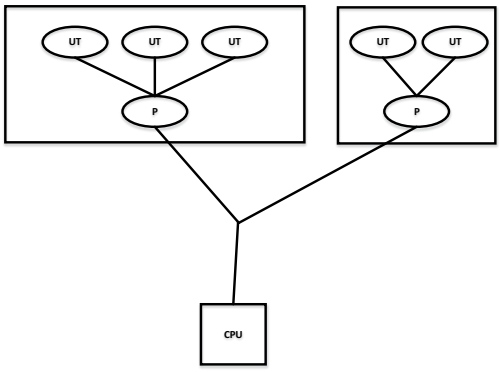
实现线程主要有 3 种方式：使用**内核线程**实现、使用**用户线程**实现和使用**用户线程加轻量级进程混合**实现。



内核线程（Kernel-Level Thread，KLT）就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器对线程进行调度，并负责将线程任务映射到各个处理器上。

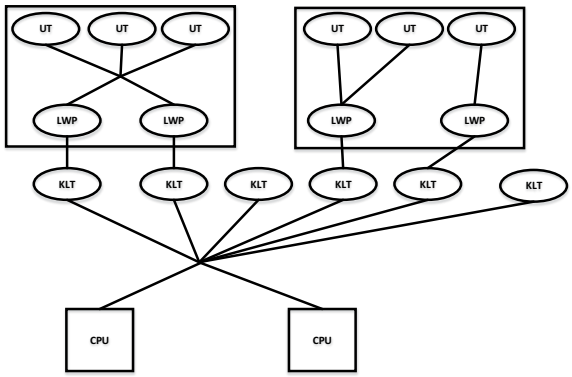
程序一般不会去直接使用内核线程，而是去使用内核线程的一种高级接口——**轻量级进程**（Light Weight Process，LWP），轻量级进程就是我们通常意义上所讲的线程。这种轻量级进程与内核线程之间 1：1 的关系称为**一对一的线程模型**。

由于内核线程的支持，每个轻量级进程都成为一个独立的调度单元，即使有一个轻量级进程在系统调用中阻塞了，也不会影响整个进程继续工作，但是轻量级进程具有它的局限性：首先，**由于是基于内核线程实现的，所以各种线程操作，如创建、析构及同步，都需要进行系统调用。而系统调用的代价相对较高，需要在用户态和内核态中来回切换**。其次，**每个轻量级进程都需要有一个内核线程的支持，因此轻量级进程需要消耗一定的内核资源**，因此一个系统支持轻量级进程的数量是有限的。



广义上，一个线程只要不是内核线程，就是用户线程。狭义上，**用户线程**指的是**完全建立在用户空间的线程库上，系统内核不能感知线程存在的实现**。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。如果程序实现得当，这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也可以支持规模更大的线程数量。

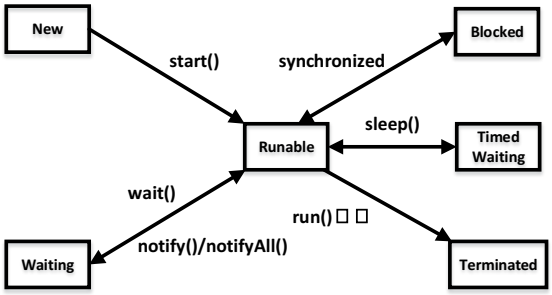
使用用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援，所有的线程操作都需要用户程序自己处理，实现复杂。



内核线程与用户线程一起使用。在这种混合实现下，既存在用户线程，也存在轻量级进程。**用户线程还是完全建立在用户空间中**，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发。而**操作系统提供支持的轻量级进程则作为用户线程和内核线程之间的桥梁**，这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级进程来完成，大大降低了整个进程完全被阻塞的风险。

对于 SUN JDK 来说，它的 Windows 版和 Linux 版都是使用一对一的线程模型实现的，**一条 Java 线程就映射到一条轻量级进程之中**，因为 Windows 和 Linux 系统提供的线程模型就是一对一的。

线程调度方式有两种，分别是**协同式线程调度**和**抢占式线程调度**。



新建（New）：创建后尚未启动的线程处于这种状态。
运行（Runnable）：Running：正在执行；Runnable：正在等待着 CPU 为它分配执行时间。
无限期等待（Waiting）：没有设置 Timeout 参数的 Object.wait()方法；没有设置 Timeout 参数的 Thread.join()方法。
限期等待（Timed Waiting）：Thread.sleep(10)方法；设置了 Timeout 参数的 Object.wait()方法；设置了 Timeout 参数的 Thread.join()方法。
阻塞（Blocked）：等待着获取一个排它锁。
结束（Terminated）：已终止的线程的状态。

线程安全的定义：**当多个线程访问一个对象时，如果不用考虑这些线程运行时环境下的调度和执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果**。

不可变的对象一定是线程安全的。Java 语言中，**如果共享数据是一个基本数据类型，那么只要在定义时使用 final 关键字修饰它就可以保证它是不可变的。如果共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行**。保证对象行为不影响自己状态的途径有很多种，其中最简单的就是**把对象中带有状态的变量都声明为 final**，这样在构造函数之后，它就是不可变的。

String，Long、Double 等数值包装类型，BigInteger、BlgDecimal 等大数据类型都是不可变的。而 AtomicInteger 和 AtomicLong 并非不可变。

绝对的线程安全——在 Java API 中标注自己是线程安全的类，大多数都不是绝对的线程安全。两个线程，一个往 vector 中获取值，一个将 vector 中的值删除，会出现数组越界异常。

相对线程安全——保证**对这个对象单独的操作是线程安全**的，我们在调用的时候不需要做额外的保障措施，但是**对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性**。如 Vector、HashTable、Collections 的 synchronizedCollection()方法包装的集合。

线程兼容，如 ArrayList 和 HashMap 等。

线程对立：无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。Thread 类的 suspend()和 resume()方法。

互斥同步。在 Java 中，最基本的互斥同步手段就是 synchronized 关键字，synchronized 关键字经过编译之后，**会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令**，这两个字节码都需要一个 reference 类型的参数来指明要锁定和解锁的对象。如果 Java 程序中的 synchronized 明确指定了对象参数，那就是这个对象的 reference；如果没有明确指定，那就**根据 synchronized 修饰的是实例方法还是类方法，去取对应的对象实例或 Class 对象来作为锁对象**。Java 的线程是映射到操作系统的原生线程之上的，**如果要阻塞或唤醒一个线程，都需要操作系统来帮忙完成，这就需要从用户态转换到核心态中，因此状态转换需要耗费很多的处理器时间**。对于代码简单的同步块，状态转换消耗的时间有可能比用户代码执行的时间还要长。所以 synchronized 是 Java 语言中一个重量级的操作。

除了 synchronized 之外，我们还可以使用 java.util.concurrent 包中的重入锁 ReentrantLock 来实现同步，在基本用法上，ReentrantLock 与 synchronized 很相似，他们都具备一样的线程重入特性，只是代码写法上有点区别，一个表现为 API 层面的互斥（lock()和 unlock()方法配合 try/finally 语句块来完成），另一个表现为原生语法层面的互斥锁。不过，相比 synchronized，ReentrantLock 增加了一些高级功能，主要有以下 3 项：**等待可中断**、**可实现公平锁**，以及**锁可以绑定多个条件**。

等待可中断是指当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情，可中断特性对处理执行时间非常长的同步块很有帮助。

公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；而非公平锁则不保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但可以通过带有布尔值的构造函数要求使用公平锁。

锁绑定多个条件是指一个 ReentrantLock 对象可以同时绑定多个 Condition 对象，而在 synchronized 中，锁对象的 wait()和 notify()或 notifyAll()方法可以实现一个隐含的条件，如果要和多于一个的条件关联时，就不得不额外地添加一个锁，而 ReentrantLock 则无须这样做，只需要多次调用 newCondition()方法即可。

非阻塞同步。随着**硬件指令集的发展**，可以使用基于冲突检测的乐观并发策略。通俗地说，就是先进行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再采取其他的补偿措施（最常见的补偿措施就是不断地重试，直到成功为止）。

硬件保证一个从语义上看起来需要多次操作的行为只通过一条处理器指令就能完成：**测试并设置**（Test-and-Set）、**获取并增加**（Fetch-and-Increment）、**交换**（Swap）、**比较并交换**（Compare-and-Swap，CAS）、**加载链接/条件存储**（Load-Linked/Store-Conditional，LL/SC）。

CAS 指令需要有 3 个操作数，分别是内存位置（在 Java 中可以简单理解为变量的内存地址，用 V 表示）、旧的预期值（用 A 表示）和新值（用 B 表示）。CAS 指令执行时，当且仅当 V 符合旧预期值 A 时，处理器用新值 B 更新 V 的值，否则它就不执行更新，但是无论是否更新了 V 值，都会返回 V 的旧值，上述的处理过程是一个原子操作。

在 JDK1.5 之后，Java 程序中才可以使用 CAS 操作，该操作由 sun.misc.Unsafe 类里面的 compareAndSwapInt()和 compareAndSwapLong()等几个方法包装提供，虚拟机在内部对这些方法做了特殊处理，即时编译出来的结果就是一条平台相关的处理器 CAS 指令，没有方法调用的过程，或者可以认为是无条件内联进去了。

为了解决 ABA 问题，java.util.concurrent 提供了一个带有标记的原子引用类“AtomicStampedReference”它可以通过控制变量值的版本来保证 CAS 的正确性。

无同步方案。

可重入代码：可以在代码执行的任何时刻中断它，转而去执行另外一段代码（包括递归调用它本身），而在控制权返回后，原来的程序不会出现任何错误。**所有的可重入代码都是线程安全的，但并非所有的线程安全的代码都是可重入的。如果一个方法，它的返回结果是可以预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的**。

线程本地存储。大部分使用消费队列的架构模式（如“生产者-消费者”模式）都会将产品的消费过程尽量在一个线程中消费完，其中最重要的一个应用实例就是经典 Web 交互模型中的“**一个请求对应一个服务器线程**”（Thread-pre-Request），这种处理方式使得很多 Web 服务端应用都可以使用线程本地存储来解决线程安全问题。

每一个线程的 Thread 对象中都有一个 ThreadLocalMap 对象，这个对象存储了一组以 ThreadLocal.threadLocalHashCode 为键，以本地线程变量为值的 K-V 值对，ThreadLocal 对象就是当前线程的 ThreadLocalMap 的访问入口，每一个 ThreadLocal 对象都包含了一个独一无二的 **threadLocalHashCode** 值，使用这个值就可以在线程 K-V 值对中找回对应的本地线程变量。

自旋锁：如果**物理机器有一个以上的处理器**，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程“稍等一下”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让线程执行一个忙循环（自旋）。

自旋等待本身虽然避免了线程切换的开销，但它是要**占用处理器时间**的。如果锁被占用的时间很短，自旋等待的效果就会非常好，反之，如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，而不会做任何有用的工作，反而会带来性能上的浪费。

自适应自旋意味着自旋的时间不再固定了，而是**由前一次在同一个锁上的自旋时间及**

锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免处理器资源浪费。

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于**逃逸分析的数据支持**，如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁自然就无须进行。

锁粗化：如果虚拟机探测到有一串零碎的操作都对同一个对象加锁，就会把加锁同步的范围扩展（粗化）到整个操作序列的外部。

轻量级锁：在**没有多线程竞争**的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。轻量级锁能提升同步性能的依据是“对于绝大部分的锁，在整个同步周期内都是不存在竞争的“。**如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥量的开销，但如果存在锁竞争，除了互斥量的开销外，还额外发生 CAS 操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢**。

如果说轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那**偏向锁**就是**在无竞争的情况下把整个同步都消除掉，连 CAS 操作都不做了**。偏向锁的“偏”就是偏心的“偏”，这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。