1.内存分区：

包含程序计数器、java 虚拟机栈、本地方法栈、java 堆、方法区、运行时常量池、直接内存

程序计数器：可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作就是通过改变程序计数器的值来选择下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都要依赖这个计数器来完成。

多线程中，为了让线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间互不影响、独立存储，因此这块内存是线程私有的。

当线程正在执行的是一个 Java 方法，这个计数器记录的是在正在执行的虚拟机字节码指令的地址；当执行的是 Native 方法，这个计数器值为空。

此内存区域是唯一一个没有规定任何 OutOfMemoryError 情况的区域。

Java 虚拟机栈：Java 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 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用于存储局部变量表、操作数栈、动态链表、方法出口信息等。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。

局部变量表中存放了编译器可知的各种基本数据类型(boolean、byte、char、short、int、float、long、double)、对象引用和 returnAddress 类型(指向了一条字节码指令的地址)。

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：本地方法栈与虚拟机的作用相似，不同之处在于虚拟机栈为虚拟机执行的 Java 方法服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。有的虚拟机直接把本地方法栈和虚拟机栈合二为一。

会抛出 stackOverflowError 和 OutOfMemoryError 异常。

Java 堆：Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建，此内存区域的唯一目的就是存放对象实例。

Java 堆是垃圾收集器管理的主要区域。由于现在收集器基本采用分代回收算法，所以 Java 堆还可细分为：新生代和老年代。从内存分配的角度来看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区(TLAB)。

Java 堆可以处于物理上不连续的内存空间，只要逻辑上连续的即可。在实现上，既可以实现固定大小的，也可以是扩展的。

如果堆中没有内存完成实例分配，并且堆也无法完成扩展时，将会抛出

OutOfMemoryError 异常。

方法区：方法区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。

相对而言，垃圾收集行为在这个区域比较少出现，但并非数据进了方法区就永久的存在了，这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载，

当方法区无法满足内存分配需要时，将抛出 OutOfMemoryError 异常。

运行时常量池：

是方法区的一部分，它用于存放编译期生成的各种字面量和符号引用。

运行时常量池：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入方法区的运行时常量池中存放。

运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法在申请到内存时会抛出 OutOfMemoryError 异常。

直接内存：直接内存不是虚拟机运行时数据区的一部分，在 NIO 类中引入一种基于通道与缓冲区的 IO 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。

直接内存的分配不会受到 Java 堆大小的限制，但是会受到本机内存大小的限制，所有也可能会抛 OutOfMemoryError 异常。

Native 函数：简单地讲，一个 Native Method 就是一个 java 调用非 java 代码的接口

2.对象的创建

1.创建一个对象通常是需要 new 关键字，当虚拟机遇到一条 new 指令时，首先检查这个指令的参数是否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。如果那么执行相应的类加载过程。

2.类加载检查通过后，虚拟机将为新生对象分配内存。为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配的方式有两种：一种叫指针碰撞，假设 Java 堆中内存是绝对规整的，用过的和空闲的内存各在一边，中间放着一个指针作为分界点的指示器，分配内存就是把那个指针向空闲空间的那边挪动一段与对象大小相等的距离。另一种叫空闲列表：如果 Java 堆中的内存不是规整的，虚拟机就需要维护一个列表，记录哪个内存块是可用的，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。采用哪种分配方式是由 Java 堆是否规整决定的，而 Java 堆是否规整是由所采用的垃圾收集器是否带有压缩整理功能决定的。另外一个需要考虑的问题就是对象创建时的线程安全问题，有两种解决方案：一是对分配内存空间的动作进行同步处理；另一种是吧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(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。

3.内存分配完成后，虚拟机需要将分配到的内存空间初始化为零值。这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就可以直接使用。

4.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等，这些信息存放在对象的对象头中。

5.上面的工作都完成以后，从虚拟机的角度来看一个新的对象已经产生了。但是从 Java 程序的角度，还需要执行 init 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

3.判断对象是否存活算法

1.计数算法

给对象中添加一个应用计数器，每当有一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器为 0 的对象就是不可能在被使用的。

在主流的 Java 虚拟机里面没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决对象之间相互循环引用的问题

引用计数器缺陷的例子：

```
objA = objB
objB = objA
```

互相引用，引用计数器不为 0，回收不了。

2.可达性算法

通过一系列的称为“GC Roots”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时，则证明此对象是不可用的。

4.利用可达性分析算法 GC 怎么判断对象生存还是死亡，经过了几次过滤，每次都做了什么

经历了 2 次标记过程，即 2 次过滤过程。

第一次：如果对象在进行可达性分析后发现没有 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，那么这个对象将会被放置一个叫做 F-Queue 的队列之中，并在稍后由一个由虚拟机自动创建的、低优先级的 Finalizer 线程去执行它。

第二次：GC 将会对 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，如果对象这个时候还没有逃脱，那基本上它就真的被回收了。

finalize()方法：每个对象的 finalize 方法只会执行一次。

5.垃圾收集算法

1.标记-清除算法

算法：分为“标记”和“清除”两个阶段。首先标记出所有需要回收的对象，

在标记完成后统一回收所有标记的对象

缺点：一是效率问题，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回收动作。

2.复制算法

算法：它将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当这一块的内存用完了，就将还存活着的对象复制到另一块上面，然后在把已使用过的内存空间一次清理掉。

优点：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。

缺点：只是这种算法的代价是将内存缩小为了原来的一半。

3.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，而是让所有对象都向一端移动，然后直接清理掉端边界以外的内存。针对老年代的算法。

4.分代收集算法

一般把 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适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制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，就必须使用“标记-清除”或者“标记-整理”算法来进行回收。

6.新生代和老年代

新生代的划分：将内存分为较大的 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 中还存活着的对象一次性地复制到另一块 Survivor 空间上。最后清理掉 Eden 和刚才用过的 Survivor 空间。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8：1，也就是每次新生代中可用内存空间为整个新生代容量的 90%（80%+10%），只有 10%的内存会被“浪费”。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于 10%的对象存活，当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保。

如果另一块 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。

7.垃圾回收器

1. Serial 收集器（JDK1.3.1 之前就有）：新生代收集器

特性：是一个单线程收集器，在他进行垃圾收集时，必须暂停其他所有的工作线程，直到他收集结束（Stop The World）。

优于其他收集器的地方：简单而高效（与其他收集器的单线程比），对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

2. ParNew 收集器：新生代收集器

特性：Serial 收集器的多线程版本

parNew 收集器与 Serial 收集器的比较：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，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证可以超越 Serial 收集器。当然，随着可以使用的 CPU 的数量的增加，它对于 GC 时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多的环境下，可以使用-XX:ParallelGCThreads 参数来限制垃圾收集的线程数。

3. Parallel Scavenge 收集器：新生代收集器（复制算法）

特性：目标是达到一个可控的吞吐量。所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，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），虚拟机总共运行了 100 分钟，其中垃圾收集花

掉了 1 分钟，那吞吐量就是 99%。

Parallel Scavenge 收集器提供了两个参数以及直接设置精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCpauseMillis 参数以及直接设置吞吐量大小的-XX:GCTimeRatio 参数。

Parallel Scavenge 收集器还有一个参数-XX:UseAdaptiveSizePolicy。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小。Eden 与 Survivor 区的比例、晋升老年代对象大小等细节参数，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为 GC 自适应的调节策略。

4. Serial Old 收集器：老年代收集器（标记-整理算法）

是 Serial 收集器的老年代版本，同样为单线程收集器。

5.Parallel Old 收集器：老年代收集器（标记-整理算法）

是多线程和标记-整理算法。

6.CMS 收集器：老年代收集器（标记-清除算法）

一种以获取最短回收停顿时间为目标的收集器。

CMS 收集器的收集步骤：

（1）初始标记

（2）并发标记

（3）重新标记

（4）并发清除

初始标记、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 能直接关联到的对象，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。

CMS 的 3 个明显缺点以及解决方式

（1）CMS 默认启动的回收线程数是（CPU 数量+3）/ 4，也就是并发回收时垃圾线程不少于 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 个时，CMS 对用户程序的影响就可能变得很大，如果本来 CPU 负载就比较大，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了 50%。

解决方法：虚拟机提供了一种称为“增量式并发收集器”，不过实践证明，增量时的 CMS 收集器效果很一般，现已被声明“deprecated”

（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着，伴随着，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程之后，CMS 无法当次收集中处理掉他们，只好留待下一次 GC 时再清理掉。这一部分垃圾就是浮动垃圾）

解决方法：因此 CMS 收集器不像其他收集器那样等老年代几乎填满了在进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在 JDK1.5 的默认设置下，CMS 收集器当老年代使用了 68%的空间后就会被激活，这是一个保守的设置，如果在应用中老年代增长不是很快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction 的值来提高触发百分比，以降低内存回收次数从而获取更好的性能。在 JDK1.6 中，CMS 收集器启动值已经提升至 92%。要是 CMS 运行期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：临时启动 Serial Old 收集器来重新进行老年代的垃圾收集。这样停顿时间就很长了。

（3）由于“标记-清除”算法的实现会出现大量的碎片，会给大对象分配带来很大的麻烦。往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次 Full GC。

解决方法：CMS 提供了一个-XX:+UseCMSCompactAtFullCollection 开关参数（默认就是开启的），用于 CMS 收集器顶不住要进行 FullGC 时开启内存碎片的合并整理过程，内存整理的过程时无法并发的，空间碎片问题没有了，但停顿时间不得不变长。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction，这个参数是用于设置执行多少次不压缩的 Full GC 后，跟着来一次压缩的（默认值为 0，表示每次进入 Full GC 时都进行碎片整理）

7.G1 收集器：新生代和老年代收集器（整体看是标记-整理算法，局部看是复制算法）
G1 收集器的特点：
并行与并发：G1 能够充分利用多 CPU、多核环境下的硬件优势，使用多个 CPU 来缩短 Stop-The-World 停顿时间，部分其他收集器原本需要停顿 java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方法让 java 程序继续执行
分代收集：与其他收集器一样，分代概念在 G1 中依然得以保留。虽然 G1 可以不需要其他收集器配合就能独立完成整个 GC 堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次 GC 的旧对象以获取更好的收集效果
空间整合：与 CMS 的“标记-清理”算法不同，G1 从整体来看是基于“标记-整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着 G1 运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象是不会因为无法找到连续内存空间而提前触发下一次 GC
可预测的停顿：这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS

共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒，这几乎已经是实时 java 的垃圾回收器的特征了。

G1 收集器的内存划分：

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在是物理隔离的了，它们都是一部分 Region（不需要连续）的集合。

G1 收集器是如何建立可预测的停顿时间模型：

它可以有计划地避免在整个 java 堆中进行全区域的垃圾收集。G1 跟踪各个 Region 里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收最大的 Region（这也就是 Garbage-First 名称的来由）。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限的时间内可以获取尽可能高的收集效率。

G1 收集器出现的问题以及解决方法：

Region 不可能是孤立的。一个对象分配在某个 Region 中的，它并非只能被本 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？这个问题其实并非在 G1 中才有，只是在 G1 中更加突出而已。

解决方法：在 G1 收集器中，Region 之间的对象引用以及其他收集器中的新生代与老年代之间的对象引用，虚拟机都是使用 Remembered Set 来避免全堆扫描的。G1 中每个 Region 都有一个与之相对应的 Remembered Set，虚拟机发现程序在对 Reference 类型的数据记进行写操作时，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 根节点的枚举范围加入 Remembered Set 即可保证不对全堆扫描也不会遗漏。

G1 收集器的收集步骤：

（1）初始标记
（2）并发标记
（3）最终标记
（4）塞选回收

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，并且修改 TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的 Region 中创建对象，这阶段需要停顿线程，但耗时很短。并发标记阶段时

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。

8.类文件结构即 class 结构

Class 文件是一组以 8 位字节为基础单位的二进制流，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。

魔数与 Class 文件的版本：每个 Class 文件的头 4 个字节称为魔数，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。第 5 和第 6 个字节是次版本号，第 7 和第 8 个字节是主版本号。

虚拟机必须拒绝执行超过其版本号的 Class 文件。

一个 Class 文件中的二进制文件：

CA FE BA BE 00 00 00 32 00 16 07 00 02 01 00 1D

6F 72 67 2F 66 6E 69 .....

则这个 Class 文件的

魔数为 0xCAFEBABE

版本号为：00 00 00 32　　45.0

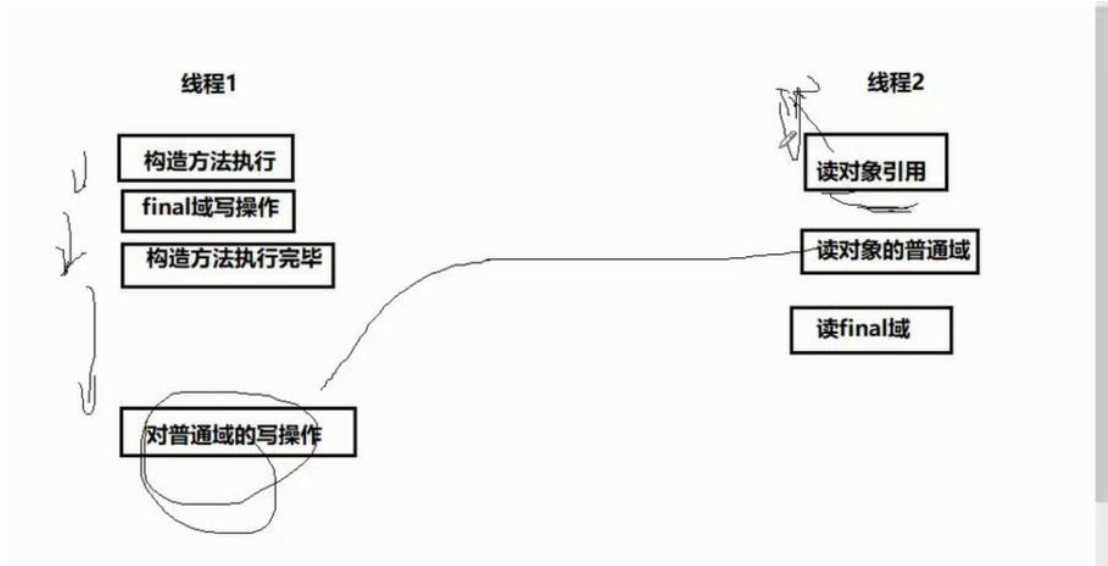
9.Final

写 final 域的重排序规则：

写 final 域的重排序的规则禁止把 final 域的写 重排序到构造方法之外。

Final 域在多线程中较普通变量的优势；

因为 final 域中的变量不会被 java 虚拟机重排序到构造方法之外，所以在另一个线程读取 final 域中的变量时一定不会出错，而普通变量在重排序就有可能被重排序到构造方法之外去，所以在另一个线程中读取普通变量时有时会读取不到正确的值。



读 final 域的重排序规则
在一个线程中，初次读对象引用和初次读对象所包含的 final 域，java 内存模型禁止处理器重排序这两个操作。

这样就保证了，在读取完对象后，final 域的初始化也能完成了。

Final 域为抽象类型
在构造方法内对一个 final 引用的对象的成员域的写入，与随后在构造方法外把这个构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

10.类加载机制

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，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。

类加载使用的是懒加载策略：用的时候才去加载。

类加载的时机：

加载
链接（验证、准备、解析）
初始化
使用
卸载

加载：当加载的时候将字节码加载进来
链接：链接不是等待加载完在进行连接的，而是开始加载了，也就可以链接了，只有加载完毕之后链接才能完毕。
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（被 final 修饰、已在编译器把结果放入常量池的静态字段除外），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，用户需要制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。
不被初始化的例子：
通过子类引用父类的静态字段，子类不会被初始化，父类被初始化了
通过数组定义来引用类
调用类的常量

类加载过程：

加载---->验证---->准备---->解析---->初始化
　　|<----------连接--------->|
加载过程：
（1）通过一个了类的全限定名来获取定义此类的二进制流
（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 Class 对象（这个 Class 对象不会放在堆中，而是放在方法运行区），作为这个类的各种数据的访问入口
加载源：文件（class 文件，jar 文件）、网络、计算生成一个二进制流（$Proxy）、由其他文件生成（JSP）、数据库中

验证过程：

为什么需要验证：因为二进制字节码文件不一定都是由 java 编译而成的，有可能是手写的

编码，或者是在加载的过程中，不是由 class 文件加载过来的，由网上加载被别人截获进行串改等等！
验证是连接的第一步，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，需要校验一下内容。
文件格式验证（李：魔数-版本）
元数据验证（语意的校验）
字节码验证
符号引用验证（确保解析动作可以正常执行的，根据常量池中的一个变量是否有正确的指向了某一个类，如果错误会报错 noSecseError）

验证过程可以在执行程序时取消掉，不用去验证了。

准备过程：

准备阶段正式为类变量分配内存并设置变量的初始值(初始化一个默认值)。这些变量使用的内存都将在方法区中进行分配。

这里的初始值并非我们制定的值，而是其默认的值，但是如果被 fianl 修饰，那么在这个过程中，常量值会被一同指定，指定为指定的值。

Int 0
Boolean false
Float 0.0
Char ‘0’
抽象数据类型 null

```
Class hello{
    Public static int a = 10;
}
```

解析过程：

解析阶段时虚拟机将常量池中的符号引用替换为直接引用的过程。

类或者接口的解析
字段解析
类方法解析
接口方法解析

类或者接口的解析：

要把一个类或者接口的符号引用解析为直接引用，需要以下三个步骤：

1. 如果该符号引用不是一个数组类型，那么虚拟机将会把该符号代表的全限定名称传递给类加载器去加载这个类。这个过程由于涉及验证过程所以可能会触发其他相关类的加载
2. 如果该符号引用是一个数组类型，并且该数组的元素类型是对象。我们知道符号引用是存在方法区的常量池中的，该符号引用的描述符会类似”[java/lang/Integer”的形式，将会按照上面的规则进行加载数组元素类型，如果描述符如前面假设的形式，需要加载的元素类型就是 java.lang.Integer ,接着由虚拟机将会生成一个代表此数组对象的直接引用
3. 如果上面的步骤都没有出现异常，那么该符号引用已经在虚拟机中产生了一个直接引用，但是在解析完成之前需要对符号引用进行验证，主要是确认当前调用这个符号引用的类是否具有访问权限，如果没有访问权限将抛出 java.lang.IllegalAccess 异常

字段解析:

对字段的解析需要首先对其所属的类进行解析，因为字段是属于类的，只有在正确解析得到其类的正确的直接引用才能继续对字段的解析。对字段的解析主要包括以下几个步骤：

1. 如果该字段符号引用就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，解析结束
2. 否则，如果在该符号的类实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果在接口中包含了简单名称和字段描述符都与目标相匹配的字段，那么久直接返回这个字段的直接引用，解析结束
3. 否则，如果该符号所在的类不是 Object 类的话，将会按照继承关系从下往上递归搜索其父类，如果在父类中包含了简单名称和字段描述符都相匹配的字段，那么直接返回这个字段的直接引用，解析结束
4. 否则，解析失败，抛出 java.lang.NoSuchFieldError 异常

如果最终返回了这个字段的直接引用，就进行权限验证，如果发现不具备对字段的访问权限，将抛出 java.lang.IllegalAccessError 异常

类方法解析:

进行类方法的解析仍然需要先解析此类方法的类，在正确解析之后需要进行如下的步骤：

1. 类方法和接口方法的符号引用是分开的，所以如果在类方法表中发现 class_index（类中方法的符号引用）的索引是一个接口，那么会抛出 java.lang.IncompatibleClassChangeError 的异常
2. 如果 class_index 的索引确实是一个类，那么在该类中查找是否有简单名称和描述符都与目标字段相匹配的方法，如果有的话就返回这个方法的直接引用，查找结束
3. 否则，在该类的父类中递归查找是否具有简单名称和描述符都与目标字段相匹配的字段，如果有，则直接返回这个字段的直接引用，查找结束
4. 否则，在这个类的接口以及它的父接口中递归查找，如果找到的话就说明这个方法是一个抽象类，查找结束，返回 java.lang.AbstractMethodError 异常
5. 否则，查找失败，抛出 java.lang.NoSuchMethodError 异常

如果最终返回了直接引用，还需要对该符号引用进行权限验证，如果没有访问权限，就抛出 java.lang.IllegalAccessError 异常

接口方法解析:

同类方法解析一样，也需要先解析出该方法的类或者接口的符号引用，如果解析成功，就进行下面的解析工作：

1. 如果在接口方法表中发现 class_index 的索引是一个类而不是一个接口，那么也会抛出 java.lang.IncompatibleClassChangeError 的异常
2. 否则，在该接口方法的所属的接口中查找是否具有简单名称和描述符都与目标字段相匹配的方法，如果有的话就直接返回这个方法的直接引用。
3. 否则，在该接口以及其父接口中查找，直到 Object 类，如果找到则直接返回这个方法的直接引用
4. 否则，查找失败

接口的所有方法都是 public，所以不存在访问权限问题

初始化过程：

初始化是类加载的最后一步，前面类加载的过程中除了在加载阶段用户应用程序可以通过自定义类加载器参与外，其余动作完全由虚拟机主导与控制。到了初始化阶段，才是真正执行类中定义的 java 程序代码。

在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据开发者通过程序控制制定的主观计划去初始化类变量和其他资源。

初始化阶段时执行类构造器<clinit>()方法的过程。

<clinit>()方法介绍：

```
public class Demo{
    static {
        i = 0;
        System.out.println(i);
    }
    static int i = 1;
}
```

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，在前面的语句块中可以赋值，但不能访问。

```
public 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);
    }
}
```

2.子类的<clinit>()在执行之前，虚拟机保证父类的先执行完毕，因此在赋值前父类 static 已经执行，因此结果为 2

3.接口也有变量要赋值，也会生成<clinit>()，但不需要先执行父类的<clinit>()方法。只有父类接口中定义的变量使用是才会初始化。

4.如果多个线程同时初始化一个类，只有一个线程会执行这个类的<clinit>()方法，其他线程等待执行完毕。如果方法执行时间过长，那么就会造成多个线程阻塞。

11.类加载器

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

只有被同一个类加载器加载的类才可能会相等。相同的字节码被不同的类加载器加载的类不相等。

类加载器的分类

启动类加载器：

由 C++实现，是虚拟机的一部分，用于加载 javahome 下的 lib 目录下的类

扩展类加载器：

加载 javahome 下/lib/ext 目录中的类

应用程序类加载器（用的最多的）：

加载用户类路径上的所指定的类库

自定义类加载器：

定义一个类，继承 ClassLoader

重写 loadClass 方法

实例化 Class 对象

自定义类加载器的优势：
1.类加载器是 java 语言的一项创新，也是 java 语言流行的重要原因之一，它最初的设计是为了满足 java Applet 的需求而开发出来的。
2.高度的灵活性
3.通过自定义类加载器可以实现热部署
4.代码加密

一个自定义类加载器的代码：

```
import java.io.InputStream;

public class ClassLoaderDemo {
    public static void main(String[] args) throws Exception {
        ClassLoader mycl = new ClassLoader() {
            @Override
            public Class<?> loadClass(String name) throws
ClassNotFoundException {
                String fileName = name.substring(name.lastIndexOf(".") + 1)
+ ".class";

                InputStream ins = getClass().getResourceAsStream(fileName);

                if(ins == null) {
                    return super.loadClass(name);
                }
                try {
                     byte[] buff = new byte[ins.available()];
                     ins.read(buff);

                     return defineClass(name, buff, 0, buff.length);
                } catch (Exception e) {
                     throw new ClassNotFoundException();
                }

            }
        };

        Object c = mycl.loadClass("jvm.ClassLoaderDemo").newInstance();

        System.out.println(c.getClass());
    }
}
```

这些类加载器是如何协同工作的呢

1.从 jdk1.2 开始，java 虚拟机规范推荐开发者使用双亲委派模式（ParentsDelegation Model）进行类加载，其加载过程如下：
（1）如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器器完成
（2）每一层的类加载器都把类加载请求委派给父类加载器，直到所有类加载请求都应该传递给顶层的启动类加载器。
（3）如果顶层的启动类加载器无法完成加载请求，子类加载器尝试去加载，如果连最初发起类加载请求的类加载器也无法完成加载请求时，将会抛出 ClassNotFoundException，而不再调用其子类加载器去进行类加载
2.双亲委派模式的类加载机制的优点是 java 类它的类加载器一起具备了一种带优先级的层次关系，越是基础的类，越是被上层的类加载器进行加载，保证了 java 程序的稳定运行。

启动类加载器（Bootstrap ClassLoader）<--扩展类加载器（Extension ClassLoader）<--应用程序类加载器（Application ClassLoader）<--自定义类加载器（User ClassLoader）