类加载的全过程：
加载 -> 验证 -> 准备 -> 解析 -> 初始化 一共五个阶段

1.加载过程：
加载和类加载不同，加载阶段，虚拟机要做的三件事情：
(1) 通过一个类的全限定名来获取定义此类的二进制流
(2) 将这个字节流所代表的静态存储结构转化称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，作为方法区对这个类的各种数据的访问入口

普通对象加载:
(1) 从zip包中读取，如jar、war、ear包等
(2) 从网络中获取
(3) 运行时计算生成，常见的就是动态带了，在java.lang.refelect.Proxy中，就是通过ProxyGenerator.gerneratеProxyClass来为特定结构生成代理类的二进制流
(4) 由其他文件生成，典型的就是jsp
(5) 从数据库读取，这种场景比较少见
加载阶段是开发人员可控性最强的（一个非数组类的加载），因为加载阶段既可以通过系统提供的引导类进行也可以使用用户自定义的类加载器来完成，用户可以通过自己实现的类加载器控制字节流的获取方式（类加载器的laodClass方法）
数组类的加载不是通过加载器，而是有java虚拟机直接创建：
(1) 如果是数组的组件类型是引用类型，那就递归采用定义的加载过程去加载组件，数组类将会在加载该类的类加载器的类名称空间上被标识
(2) 如果数组的组件类型不是引用类型，java虚拟机将会把数组类标记为与引导类加载器关联
(3) 数组类的可见性与他的组件类型可见性一致，如果组件类型不是引用类型，那数组类的可见性将会默认为public
加载完成后，虚拟机外部的二进制流就会按照虚拟机所需要的格式存储在方法区之中，然后在内存中定义一个java.lang.Class类的对象（并没规定在堆上，对于hotspot虚拟机，Class对象比较特殊，它虽然是对象，但是存放在方法区中），这个对象将作为程序访问方法区中的这些类型数据的外部接口

2. 验证阶段

验证阶段大致分为: 文件格式校验、元数据校验、字节校验、符号引用校验
校验阶段是一个非常重要但是非必要的阶段，如果所运行的全部代码都已经反复使用和验证过，那么实施阶段可以考虑使用-Xverify:none参数来关闭大部分的类实施措施，来缩短加载时间

3. 准备阶段

准备阶段是正式为类变量分配内容并设置类变量初始化值的阶段。这些变量所使用的内存都将在方法去中进行分配。这里分配的变量仅仅类变量（static），而非实例变量，实例变量需要与对象实例化是一起分配到堆内存上的。其次这里的初始值“通常情况”下是对数据类型的零值
eg: public static int val = 123
变量val在准备阶段后的初始值是0而不是123，因为还没有执行任何java代码，把vla赋值123需要pubstatic指令宝贝程序编译或才行，所以val = 123的动作在初始化阶段才会执行 【类初始化五个阶段的最后一个】
通常情况赋零值，那相对的特殊情况是如果类字段的字段属性表中存在ConstantValue属性，你们准备解读那的变量val就会被初始化为ConstantValue属性的值
eg: public static fina lint val = 123
编译时javac将会为val生成constantValue属性，在准备阶段虚拟就就会根据ConstantValue的设置键val赋值为123

4. 解析阶段

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。符号引用在class文件的数据结构中被多次提到，在class文件中已Constant_Class_info, Constant_FieldRef_info，Constant_Method_ref等类型的常量出现
符号引用：符号引用以一组符号来描述所引用的目标，符号引用可以是任何形式上的字面量、只要使用时能无歧义的定位到目标即可。符号引用的字面量形式明确定义在Java虚拟机规范的class文件格式中
直接引用：直接引用可以是直接指向目标的指针、相对偏移量或一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在
对于同一个符号引用进行多次解析请求是常见的事情，除了invokedynamic指令外，虚拟机实现了可以对第一次解析的结果进行缓存（在 运行时常量池中记录直接引用、并把常量标识为已解析状态）从而避免解析动作重复进行

解析动作主要是针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用上

(1) 类或接口解析

a. 如果C不是一个数组类型，那虚拟机讲会把代表N的全限定名传递给D的类加载器去加载这个类

b. 如果c是一个数组，并且数组的元素类型为对象，那将会按照第一点的规整加载数组元素类型

c. 如果上面步骤没有任何异常，那么C在虚拟机中实际已经成为一个有效的类或接口了，但在解析完成之前还要对符号引用进行验证，如果没有对C的访问权限，就会抛出IllegalAcessError

(2) 字段解析

要解析一个为被解析过的字段引用，首先将会对字段内class_index项中的索引的Constant_Class_info符号进行解析，也就是字段所属类或接口的解析。解析类或接口失败，都会导致解析字段失败。

a. 如果c本身就包括了简单名称和字段描述符都与目标想匹配的字段，则返回这个字段的直接引用

b. 否则，如果c中实现了接口，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，则返回这个字段的直接引用

c. 否则如果c不是object的话，将会按照继承方式从下往上递归搜索其父类，如果其父类中包含了简单名称和字段描述符的都与目标相匹配的字段，则返回这个字段的直接引用

d. 如果查找失败，则抛出NoSuchFieldError

(3) 类方法解析

和字段解析相似

5. 初始化阶段

类初始化阶段是类加载过程的最后一步，前面类加载过程中，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之外，其余动作完全由虚拟机主导和控制，而到了初始化阶段，才是真正意义上开始执行java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，则根据程序员通过程序制定主观的计划去初始化类变量和其他资源，初始化极端就是执行类构造器<cinit>()方法，<cinit>()和<init>()不同

(1) <cinit>()方法是由编译器自动收集类 中的所有变量的赋值动作和静态代码块

中的语句合并产生的，编译器收集的顺序是由语句在源代码中出现的顺序决定的，静态代码块中只能访问你到定义在静态语句块之前的变量，定义在其之后的变量，可以在静态代码块中赋值，但是不能访问

(2) <cinit>()方法与类的构造函数<init>()不同，它不需要显示调用父类构造器，虚拟机会保证在子类的<cinit>()方法的执行之前，父类的<cinit>()已经执行完毕，所以虚拟机中第一个执行的<cinit>()方法的肯定是Object类

(3) 由于父类的<cinit>()方法先执行，也就意味着父类中定义的静态语句块要优于子类变量的赋值

(4) <cinit>()方法对于类或者接口来说并不需要的，如果一个类中没有静态语句块，也没有相对变量的赋值操作，那编译器不会为这个类生成<cinit>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<cinit>犯法，但是接口与类不同，执行接口的<cinit>方法不需先执行父接口的<cinit>方法

(6) 虚拟机保证一个类的<cinit>方法在多线程的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，那么只有一个线程会去执行<cinit>方法，其他的线程都需要阻塞等待，直到程序的<cinit>执行完成