JVM（三）：类加载机制（类加载过程和类加载器）

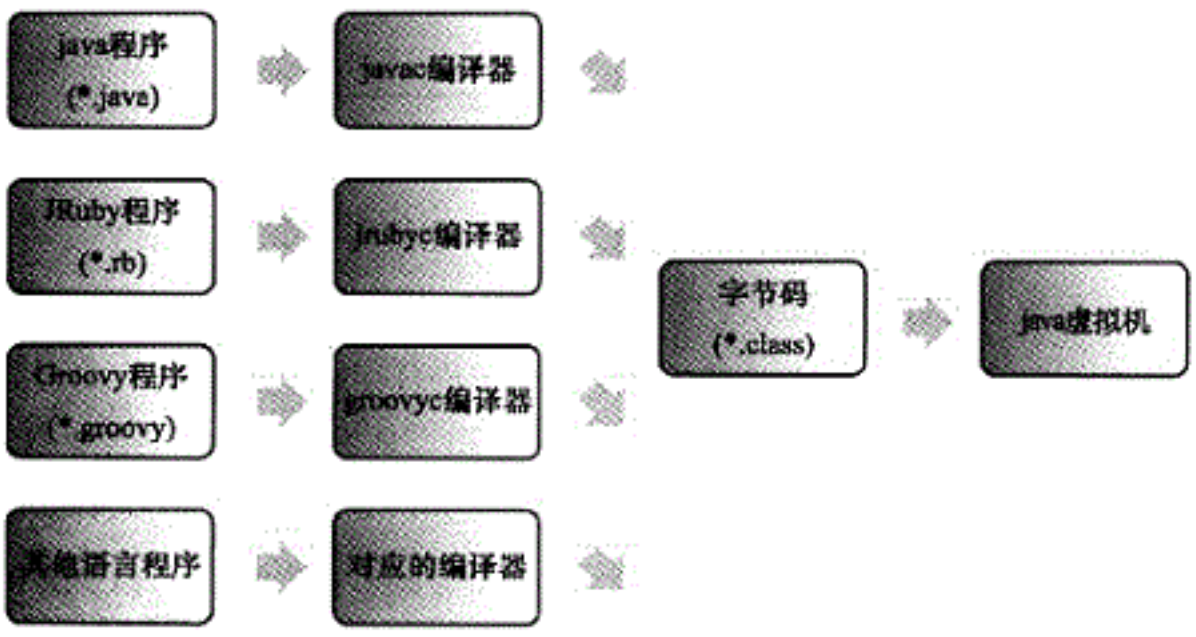
一、为什么要使用类加载器?

Java语言里，类加载都是在程序运行期间完成的，这种策略虽然会令类加载时稍微增加一些性能开销，但是会给java应用程序提供高度的灵活性。例如：

1.编写一个面向接口的应用程序，可能等到运行时再指定其实现的子类；
2.用户可以自定义一个类加载器，让程序在运行时从网络或其他地方加载一个二进制流作为程序代码的一部分；(这个是Android插件化，动态安装更新apk的基础)

二、类加载的过程

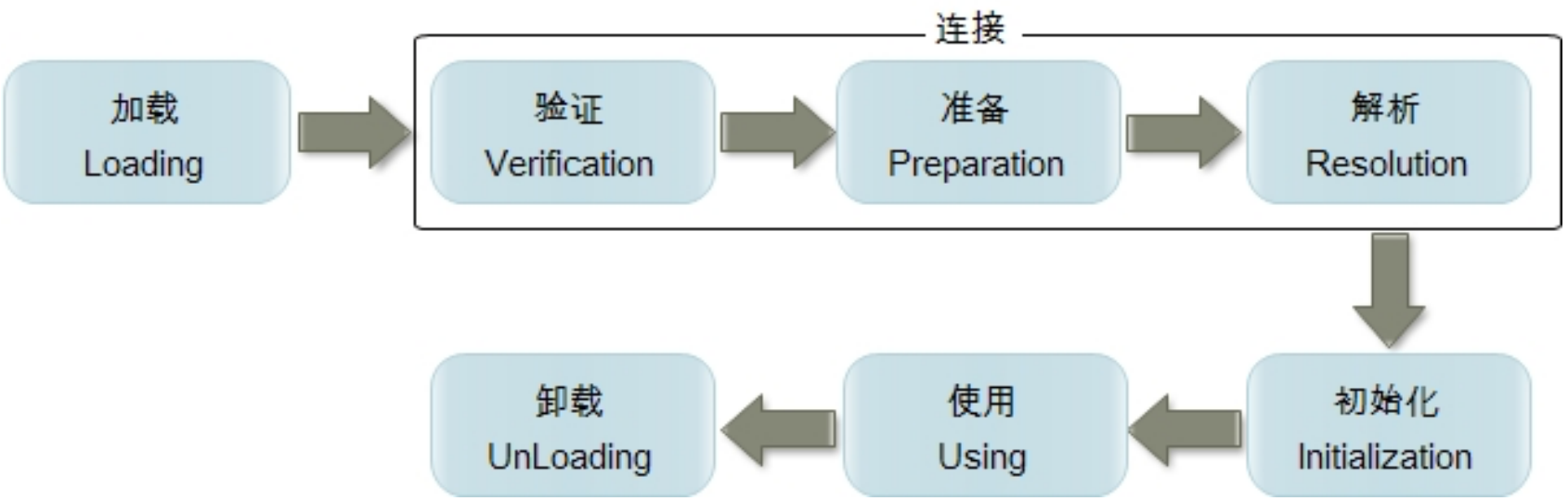
使用java编译器可以把java代码编译为存储字节码的Class文件，使用其他语言的编译器一样可以把程序代码翻译成Class文件，java虚拟机不关心Class的来源是何种语言。如图所示：



在Class文件中描述的各种信息，最终都需要加载到虚拟机中才能运行和使用。那么虚拟机是如何加载这些Class文件的呢?
JVM把描述类数据的字节码.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。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的生命周期包括

了：加载(Loading)、验证(Verification)、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)、卸载(Unloading)七个阶段，其中验证、准备、解析三个部分统称链接。



加载(装载)、验证、准备、初始化和卸载这五个阶段顺序是固定的，类的加载过程必须按照这种顺序开始，而解析阶段不一定；它在某些情况下可以在初始化之后再开始，这是为了运行时动态绑定特性（JIT例如接口只在调用的时候才知道具体实现的是哪个子类）。值得注意的是：这些阶段通常都是互相交叉的混合式进行的，通常会在一个阶段执行的过程中调用或激活另外一个阶段。

1.加载：（重点）

加载阶段是“类加载机制”中的一个阶段，这个阶段通常也被称作“装载”，主要完成：

1.通过“类全名”来获取定义此类的二进制字节流

2.将字节流所代表的静态存储结构转换为方法区的运行时数据结构

3.在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口

相对于类加载过程的其他阶段，加载阶段(准备地说，是加载阶段中获取类的二进制字节流的动作)是开发期可控性最强的阶段，因为加载阶段可以使用系统提供的类加载器(ClassLoader)来完成，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去控制字节流的获取方式。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式有虚拟机实现自行定义，虚拟机并

未规定此区域的具体数据结构。然后在java堆中实例化一个java.lang.Class类的对象，这个对象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：（了解）

验证是链接阶段的第一步，这一步主要的目的是确保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安全。
验证阶段主要包括四个检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。

1.文件格式验证

验证class文件格式规范，例如： class文件是否已魔术0xCAFEBABE开头，主、次版本号是否在当前虚拟机处理范围之内等

2.元数据验证

这个阶段是对字节码描述的信息进行语义分析，以保证起描述的信息符合java语言规范要求。验证点可能包括：这个类是否有父类(除了java.lang.Object之外，所有的类都应当有父类)、这个类是否继承了不允许被继承的类(被final修饰的)、如果这个类的父类是抽象类，是否实现了起父类或接口中要求实现的所有方法。

3.字节码验证

进行数据流和控制流分析，这个阶段对类的方法体进行校验分析，这个阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。如：保证访法体中的类型转换有效，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但不能把一个父类对象赋值给子类数据类型、保证跳转命令不会跳转到方法体以外的字节码命令上。

4.符号引用验证

符号引用中通过字符串描述的全限定名是否能找到对应的类、符号引用类中的类，字段和方法的访问性(private、protected、public、default)是否可被当前类访问。

3.准备：（了解）

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，这些内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的知识点，首先是这时候进行内存分配的仅包括类变量(static 修饰的变量),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。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量定义为:

public static int value = 12;

那么变量value在准备阶段过后的初始值为0而不是12，因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把value赋值为12的动作将在初始化阶段才会被执行。

上面所说的“通常情况”下初始值是零值，那相对于一些特殊的情况，如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，建设上面类变量value定义为:

public static final int value = 123;

编译时javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value设置为123。

4.解析：（了解）
解析阶段是虚拟机常量池内的符号引用替换为直接引用的过程。
符号引用：符号引用是一组符号来描述所引用的目标对象，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，引用的目标对象并不一定已经加载到内存中。

直接引用：直接引用可以是直接指向目标对象的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机内存布局实现相关的，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。

虚拟机规范并没有规定解析阶段发生的具体时间，只要求了在执行anewarry、checkcast、getfield、instanceof、invokeinterface、invokespecial、invokestatic、invokevirtual、multianewarray、new、putfield和putstatic这13个用于操作符号引用的字节码指令之前，先对它们使用的符号引用进行解析，

所以虚拟机实现会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

解析的动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。分别对应编译后常量池内的CONSTANT_Class_Info、CONSTANT_Fieldref_Info、CONSTANT_Methodef_Info、CONSTANT_InterfaceMethoder_Info四种常量类型。

1.类、接口的解析

2.字段解析

3.类方法解析

4.接口方法解析

5.初始化：（了解）

类的初始化阶段是类加载过程的最后一步，在准备阶段，类变量已赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。在以下四种情况下初始化过程会被触发执行：

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、读取或设置一个类的静态字段(被final修饰、已在编译器把结果放入常量池的静态字段除外)的时候，以及调用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

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、则需要先出发其父类的初始化

4.jvm启动时，用户指定一个执行的主类(包含main方法的那个类)，虚拟机会先初始化这个类

在上面准备阶段 public static int value = 12; 在准备阶段完成后 value的值为0，而在初始化阶调用了类构造器<clinit>()方法，这个阶段完成后value的值为12。

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

*类构造器<clinit>()方法与类的构造函数(实例构造函数<init>()方法)不同，它不需要显式调用父类构造，虚拟机会保证在子类<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中的第一个执行的<clinit>()方法的类肯定是java.lang.Object。

*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快要优先于子类的变量赋值操作。

*<clinit>()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句，也没有变量赋值的操作，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语句块，但接口与类不太能够的是，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被使用时，父接口才会被初始化。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。

*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和同步，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。如果一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞。

三、类加载器

JVM设计者把类加载阶段中的“通过'类全名'来获取定义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”。

1.类与类加载器

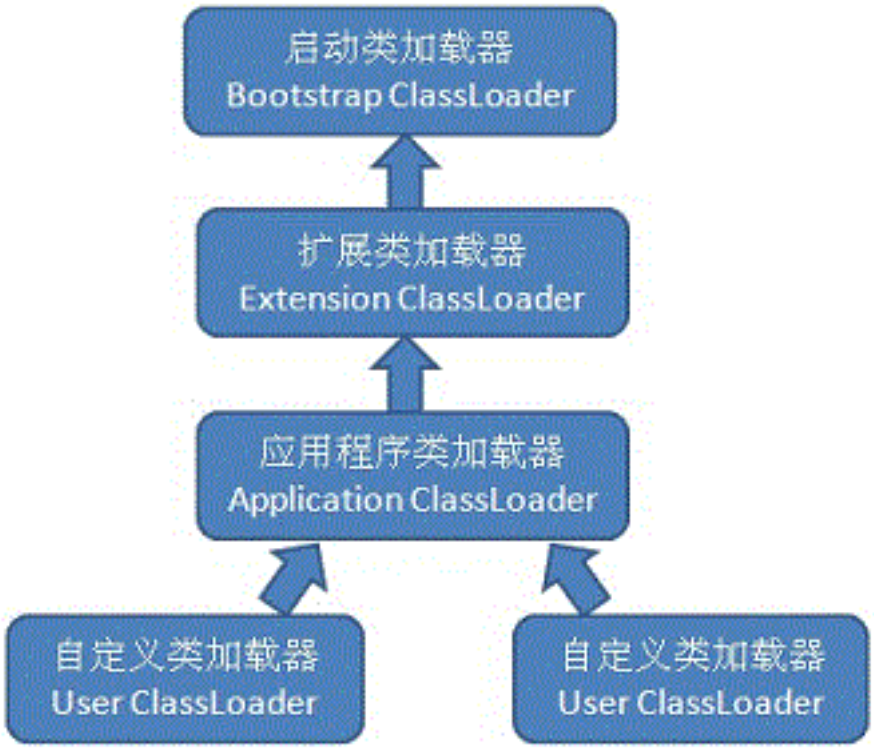
对于任何一个类，都需要由加载它的类加载器和这个类来确立其在JVM中的唯一性。也就是说，两个类来源于同一个Class文件，并且被同一个类加载器加载，这两个类才相等。

2.双亲委派模型

从虚拟机的角度来说，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），该类加载器使用C++语言实现，属于虚拟机自身的一部分。另外一种就是所有其它的类加载器，这些类加载器是由Java语言实现，独立于JVM外部，并且全部继承自抽象类java.lang.ClassLoader。

从Java开发人员的角度来看，大部分Java程序一般会使用到以下三种系统提供的类加载器：
1)启动类加载器（Bootstrap ClassLoader）：负责加载JAVA_HOME\lib目录中并且能被虚拟机识别的类库到JVM内存中，如果名称不符合的类库即使放在lib目录中也不会被加载。该类加载器无法被Java程序直接引用。
2)扩展类加载器（Extension ClassLoader）：该加载器主要是负责加载JAVA_HOME\lib\，该加载器可以被开发者直接使用。
3)应用程序类加载器（Application ClassLoader）：该类加载器也称为系统类加载器，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该类加载器，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。

我们的应用程序都是由这三类加载器互相配合进行加载的，我们也可以加入自己定义的类加载器。这些类加载器之间的关系如下图所示：



如上图所示的类加载器之间的这种层次关系，就称为类加载器的双亲委派模型（Parent Delegation Model）。该模型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。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。

双亲委派模型的工作过程为：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，每一个层次的加载器都是如此，因此所有的类加载请求都会传给顶层的启动类加载器，只有当父加载器反馈自己无法完成该加载请求（该加载器的搜索范围中没有找到对应的类）时，子加载器才会尝试自己去加载。

使用这种模型来组织类加载器之间的关系的好处是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java.lang.Object类，无论哪个类加载器去加载该类，最终都是由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。否则的话，如果不使用该模型的话，如果用户自定义一个java.lang.Object类且存放在classpath中，那么系统中将会出现多个Object类，应用程序也会变得很混乱。如果我们自定义一个rt.jar中已有类的同名Java类，会发现JVM可以正常编译，但该类永远无法被加载运行。
在rt.jar包中的java.lang.ClassLoader类中，我们可以查看类加载实现过程的代码，具体源码如下：

通过上面代码可以看出，双亲委派模型是通过loadClass()方法来实现的，根据代码以及代码中的注释可以很清楚地了解整个过程其实非常简单：先检查是否已经被加载过，如果没有则调用父加载器的loadClass()方法，如果父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载器加载失败，则先抛出ClassNotFoundException，然后再调用自己的findClass()方法进行加载。

3.自定义类加载器

若要实现自定义类加载器，只需要继承java.lang.ClassLoader 类，并且重写其findClass()方法即可。java.lang.ClassLoader 类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class 类的一个实例。除此之外，ClassLoader 还负责加载 Java 应用所需的资源，如图像文件和配置文件等，ClassLoader 中与加载类相关的方法如下：

方法	说明
getParent()	返回该类加载器的父类加载器。
loadClass(String name)	加载名称为 二进制名称为name 的类，返回的结果是 java.lang.Class 类的实例。
findClass(String name)	查找名称为 name 的类，返回的结果是 java.lang.Class 类的实例。
findLoadedClass(String name)	查找名称为 name 的已经被加载过的类，返回的结果是 java.lang.Class 类的实例。
resolveClass(Class<?> c)	链接指定的 Java 类。

注意：在JDK1.2之前，类加载尚未引入双亲委派模式，因此实现自定义类加载器时常常重写loadClass方法，提供双亲委派逻辑，从JDK1.2之后，双亲委派模式已经被引入到类加载体系中，自定义类加载器时不需要在自己写双亲委派的逻辑，因此不鼓励重写loadClass方法，而推荐重写findClass方法。

在Java中，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定其在java虚拟机中的唯一性，即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之下才有意义，否则，即使这两个类来源于同一个Class类文件，只要加载它的类加载器不相同，那么这两个类必定不相等(这里的相等包括代表类的Class对象的equals()方法、isAssignableFrom()方法、isInstance()方法和instanceof关键字的结果)。例子代码如下：

类加载器双亲委派模型是从JDK1.2以后引入的，并且只是一种推荐的模型，不是强制要求的，因此有一些没有遵循双亲委派模型的特例：（了解）

⑴.在JDK1.2之前，自定义类加载器都要覆盖loadClass方法去实现加载类的功能，JDK1.2引入双亲委派模型之后，loadClass方法用于委派父类加载器进行类加载，只有父类加载器无法完成类加载请求时才调用自己的findClass方法进行类加载，因此在JDK1.2之前的类加载的loadClass方法没有遵循双亲委派模型，因此在JDK1.2之后，自定义类加载器不推荐覆盖loadClass方法，而只需要覆盖findClass方法即可。

⑵.双亲委派模式很好地解决了各个类加载器的基础类统一问题，越基础的类由越上层的类加载器进行加载，但是这个基础类统一有一个不足，当基础类想要调用回下层的用户代码时无法委派子类加载器进行类加载。为了解决这个问题JDK引入了ThreadContext线程上下文，通过线程上下文的setContextClassLoader方法可以设置线程上下文类加载器。

JavaEE只是一个规范，sun公司只给出了接口规范，具体的实现由各个厂商进行实现，因此JNDI，JDBC,JAXB等这些第三方的实现库就可以被JDK的类库所调用。线程上下文类加载器也没有遵循双亲委派模型。

⑶.近年来的热码替换，模块热部署等应用要求不用重启java虚拟机就可以实现代码模块的即插即用，催生了OSGi技术，在OSGi中类加载器体系被发展为网状结构。OSGi也没有完全遵循双亲委派模型。

4.动态加载Jar && ClassLoader 隔离问题

动态加载Jar：

Java 中动态加载 Jar 比较简单，如下：

表示加载 libs 下面的 jar1.jar，其中 parentLoader 就是上面1中的 parent，可以为当前的 ClassLoader。

ClassLoader 隔离问题：

大家觉得一个运行程序中有没有可能同时存在两个包名和类名完全一致的类?
JVM 及 Dalvik 对类唯一的识别是 ClassLoader id + PackageName + ClassName，所以一个运行程序中是有可能存在两个包名和类名完全一致的类的。并且如果这两个”类”不是由一个 ClassLoader 加载，是无法将一个类的示例强转为另外一个类的，这就是 ClassLoader 隔离。 如 Android 中碰到如下异常

当碰到这种问题时可以通过 instance.getClass().getClassLoader(); 得到 ClassLoader，看 ClassLoader 是否一样。

加载不同 Jar 包中公共类：

现在 Host 工程包含了 common.jar, jar1.jar, jar2.jar，并且 jar1.jar 和 jar2.jar 都包含了 common.jar，我们通过 ClassLoader 将 jar1, jar2 动态加载进来，这样在 Host 中实际是存在三份 common.jar，如下图：

https://farm4.staticflickr.com/3872/14301963930_2f0f0fe8aa_o.png

我们怎么保证 common.jar 只有一份而不会造成上面3中提到的 ClassLoader 隔离的问题呢，其实很简单，在生成 jar1 和 jar2 时把 common.jar 去掉，只保留 host 中一份，以 host ClassLoader 为 parentClassLoader 即可。

最后：

一道面试题

能不能自己写个类叫java.lang.System?

答案：通常不可以，但可以采取另类方法达到这个需求。
解释：为了不让我们写System类，类加载采用委托机制，这样可以保证爸爸们优先，爸爸们能找到的类，儿子就没有机会加载。而System类是Bootstrap加载器加载的，就算自己重写，也总是使用Java系统提供的System，自己写的System类根本没有机会得到加载。

但是，我们可以自己定义一个类加载器来达到这个目的，为了避免双亲委托

机制，这个类加载器也必须是特殊的。由于系统自带的三个类加载器都加载特定目录下的类，如果我们自己的类加载器放在一个特殊的目录，那么系统的加载器就无法加载，也就是最终还是由我们自己的加载器加载。