Java类加载器除了根类加载器之外，其他类加载器都是使用java语言编写的，所以程序员完全可以开发自己的类加载器。当我们运行某个java程序时，该命令将会启动一个java虚拟机进程。两次运行java程序处于两个不同的jvm中，两个jvm之间并不会共享数据

当程序主动使用某个类时，如果该类还没有被加载到内存中，系统会通过**加载、连接、初始化**三个步骤对该类进行初始化，类加载器是将类的class文件读入内存，并为之创建一个java.lang.Class对象，也就是说，当程序中使用任何类的时候，系统都会为之建立一个java.lang.Class对象。

**类的加载**

类的加载由类加载器完成，类加载器通常由jvm提供，这些类加载器也是前面所有程序运行的基础，jvm提供的这些类加载器通常被称为系统加载器，除此之外，开发者可以通过集成ClassLoader基类来创建自己的类加载器。

通过使用不同的类加载器，可以从不同的来源加载类的二进制文件，通常有如下几种来源：

1 从本地文件系统加载class文件，这个前面绝大部分程序的类加载方式

2 从jar包中加载class文件，这种方式也是很常见的

3 通过网络加载class文件

4 把一个java源文件动态编译，并执行加载

类加载器通常无须等到“首次使用“该类时才加载该类，java虚拟机规范允许系统预先加载某些类。

**类的连接**

当类被加载之后，系统为之生成一个对应的Class对象，接着会进入连接阶段，连接阶段负责把类的二进制数据合并到JRE中，类连接又可分为3个阶段

1 验证：验证阶段用于检验被加载的类是否有正确的内部结构，并和其他类协调一致

2 准备：类准备阶段则负责为类的静态Field分配内存，并设置默认初始值

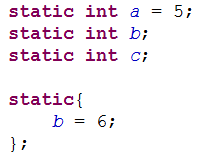
3 解析：将类的二进制数据中的符号引用替换成直接引用

**类的初始化**

在类的初始化阶段，虚拟机负责对类进行初始化，主要是对静态Field进行初始化，在Java类中对静态Field指定初始值有两种方式：

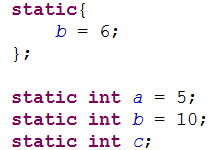
1 声明静态Field时指定初始值

2 使用静态初始化块为静态Field指定初始值





声明变量时指定初始值，静态初始化块都将被当成类的初始化语句，jvm会按这些语句在程序中的排列顺序依次执行它们：





Jvm初始化一个类包含如下几个步骤：

1 假如这个类还没有被加载和连接，则程序先加载并连接该类

2 假设该类的直接父类还没有被初始化，则先初始化其直接父类

3 假如类中有初始化语句，则系统依次执行这些初始化语句

根据步骤2可以知道，jvm最先初始化的总是java.Lang.Object类，当程序主动使用任何一个类时，系统会保证该类以及所有父类（包括直接父类和间接父类）都会被初始化。

**类初始化的时机**

当java程序首次通过下面6中方式来使用某个类或接口时，系统就会初始化该类或接口：

1 创建类的实例，为某个类创建实例的方式包括：new，使用反射，使用反序列化

2 调用某个类的静态方法

3 访问某各类或几口的静态Field，货位该静态Field赋值

4 使用仿射方式来强制创建某哥类或接口对应的java.lang.Class对象

5 初始化某个类的子类

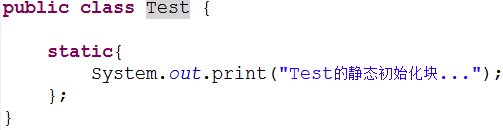
6 直接使用java.exe命令来运行某个主类

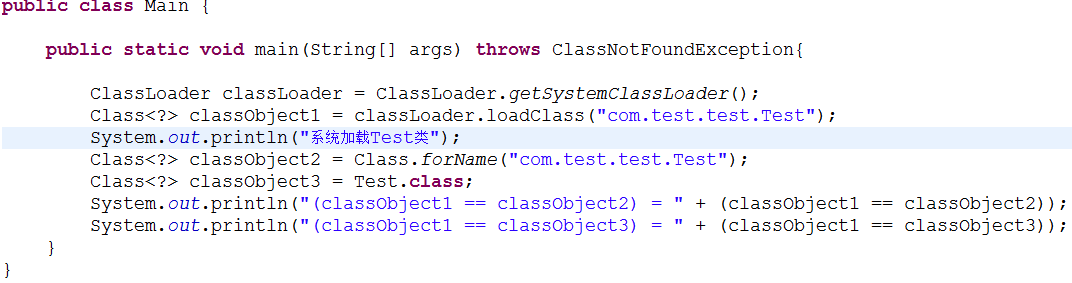
除此之外，下面几种情况需要特别指出：

对于一个final型的静态Field，如果该Field的值在编译时就可以确定下来，那么这个Field相当于“宏变量”，java编译器会在编译时直接把这个Field出现的地方替换成它的值，因此即使程序使用该静态Field，也不会导致该类的初始化。

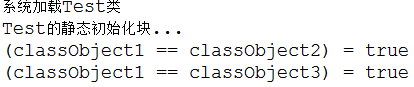
反之，如果final类型的静态Field的值不能在编译时确定下来，则必须等到运行时才可以确定该Field的值，如果通过该类来访问它的静态Filed，则会导致该类被初始化。

当使用ClassLoader类的loadClass()方法来加载某个类时，该方法只是加载该类，并不会执行该类的初始化，使用Class的forName()静态方法才会导致强制初始化该类。





注意在上面的代码中必须添加包名，否则会出现ClassNotFoundException异常



**类加载器**

类加载器负责将.class文件（可能在磁盘上，也可能在网络上）加载到内存中，并为之生成对应的java.lang.Class对象。

类加载器负责加载所有的类，系统为所有被载入内存中的类生成一个java.lang.Class实例，一旦一个类被载入jvm中，同一个类就不会被再次载入了。现在问题是，怎么样才算是“同一个类”？

正如一个对象有一个唯一的标识一样，一个载入jvm的类也有一个唯一的标识，在java中，一个类用其权限定类名（包括包名和类型）作为标识，但是在jvm中，一个类用其权限定类名和其类加载器作为其唯一标识。例如，如果在pg的包中有一个名为Person的类，被类加载器ClassLoader的实例k1负责加载，则该Person类对应的Class对象在JVM中表示为（Person、pg、kl），这意味着两个类加载器加载的同名类是不同的。

当jvm启动时，会形成由三个类加载器组成的初始类加载器层次结构：

1 Bootstarp ClassLoader：根类加载器

2 Extension ClassLoader：扩展类加载器

3 System ClassLoader：系统类加载器

Bootstarp ClassLoader被称为引导（也称为原始或根）类加载器，它负责加载Java的核心类，根加载类非常特殊，它并不是java.lang.ClassLoader的子类，而是由jvm自身实现的。

Extension ClassLoader被称为扩展类加载器，它负责加载JRE的扩展目录（%JAVA\_HOME%jre%lib%ext或者由java.ext.dirs系统属性指定的目录）中jar包的类，通过这种方式，就可以为java扩展核心类以为的新功能，只要我们把自己开发的类打包成jar文件，然后放入%JAVA\_HOME%jre%lib%ext路径即可。

System ClassLoader被称系统（也称为应用）类加载器，它负责在jvm启动时加载来自java命令的-classpath选项、java.class/.path系统属性或CLASSPATH环境变量所指定的jar包和类路径，程序可以通过ClassLoader的静态方法getSystemClassLoader()来获取系统类加载器，如果没有特定指出，则用户自定义的类加载器都以类加载器作为父加载器。

，

**类加载机制**