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

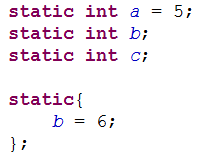
3 解析：将类的二进制数据中的符号引用替换成直接引用

**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：

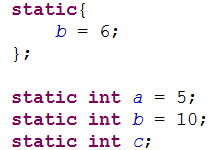
1 声明静态Field时指定初始值

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种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 访问某个类或接口的静态Field，或为该静态Field赋值

4 使用反射方式来强制创建某个类或接口对应的java.lang.Class对象

5 初始化某个类的子类

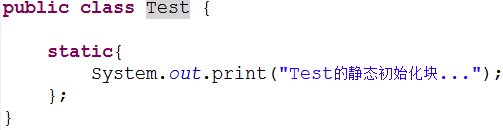
6 直接使用java.exe命令来运行某个主类

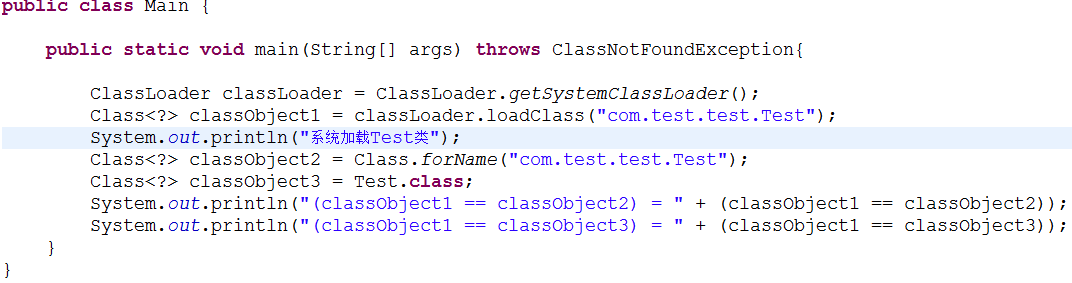
除此之外，下面几种情况需要特别指出：

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。

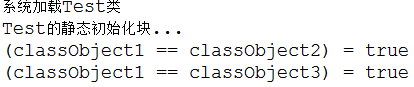
反之，如果final类型的静态Field的值不能在编译时确定下来，则必须等到运行时才可以确定该Field的值，如果通过该类来访问它的静态Filed，则会导致该类被初始化。

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**

Jvm的类加载机制主要有如下三种：

1 全盘负责。所谓全盘负责，就是当一个类加载器负责加载某个class时，该class所依赖和引用的其他class也将由该类加载器负责载入，除非显式使用另外一个类加载器来载入

2 父类委托：所谓父类委托，则是先让parent类加载器视图加载该class，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

3 缓存机制：缓存机制将会保证偶加载过的class都会被缓存，当程序中需要使用某个class时，类加载器先从缓存区中搜寻该class，只有当缓存区中不存在该class对象时，系统才会读取该类对应的二进制数据，并将其转换为class对象，存入缓存区中，这就是为什么修改了class后，必须重新气功jvm，程序所做的修改才会生效的原因。

Jvm中除根类加载器之外的所有类加载器都是ClassLoader子类的实例，开发者可以通过扩展ClassLoader的子类，并重写该ClassLoader所包含的方法来实现自定义的类加载器。

**通过反射查看类信息**

Java程序中的许多对象在运行时都会出现两种类型：编译时类型和运行时类型。例如代码：Person p = new Student()，这行代码将会生成一个p变量，该变量的编译时类型为Person，运行时类型为Student。还有更极端的情况，程序在运行时接受到外部传入的一个对象，该对象的编译死类型是Object，但程序又需要调用该对象运行时类型的方法。

为了解决这些问题，程序需要在运行时发现对象和类的真实信息，为了解决这个问题，有以下两种做法：

1 假设在编译时和运行时都完全知道类型的具体信息，在这种情况下，我们可以直接先使用instanceof运算符进行判断，再利用强制类型转换将其转换成其运行时类型的变量

2 编译时根本无法预知该对象和类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息，这就必须使用反射

**获取class对象**

前面已经介绍过了。每个类被加载之后，系统就会为该类生成一个对应的class对象，通过该Class对象就可以访问到jvm中的这个类。在java程序中获取class对象通常有如下3种方式：

1 使用Class类的forName(String clazzname)静态方法，该方法的字符串参数是某个类的全限定类名（必须添加完成包名）

2 调用某个类的class属性来获取该类对应的Class对象

3 调用某个对象的getClass()方法，该方法是java.lang.Object类中的一个方法，该方法返回该对象所属类对应的Class对象

一旦获得了某个类所对象的Class对象之后，程序就可以调用Class对象的方法来获得该对象和该类的真实信息了。

**从Class中获取信息**

1 获取构造器：

Constructor<T> getConstructor(Class<?>… parameterTypes):返回此Class对象对应类的指定public构造器

Constructor<?>[] getConstructors():返回此Class对象对应类的所有public构造器

Constructor<T> getDeclaredConstructor(Class<?>… parameterTypes):返回Class对象对应类的指定构造器，与构造器的访问权限无关

Constructor<?>[] getDeclaredConstructors():返回此Class对象对应类的所有构造器，与构造器的访问权限无关

还有依次可以获去Class对应类所包含的方法Method、所有Field、所有Annotation、内部类、外部类、接口、包名、类名

/\*\*

\* **@throws** NoClassDefFoundError, NoSuchMethodException

\* **@return** void

\* \*/

@SuppressWarnings(value="all")

**public** **static** **void** showClassInformation() **throws** NoClassDefFoundError, NoSuchMethodException{

Class<?> clazz = ~~ClassTest~~.**class**;

Constructor[] ctors = **null**;

System.*out*.println("全部构造函数");

ctors = clazz.getDeclaredConstructors();

**for**(Constructor ctor: ctors){

System.*out*.println(ctor);

}

System.*out*.println("全部public构造函数");

ctors = clazz.getConstructors(); // 获取所有的public构造函数

**for**(Constructor ctor: ctors){

System.*out*.println(ctor);

}

System.*out*.println("全部方法");

Method[] methods = clazz.getDeclaredMethods();

**for**(Method method: methods){

System.*out*.println(method);

}

System.*out*.println("全部public方法");

methods = clazz.getMethods();

**for**(Method method: methods){

System.*out*.println(method);

}

System.*out*.println("带一个字符串为参数的info方法");

System.*out*.println(clazz.getMethod("info", String.**class**));

System.*out*.println("全部Annotation");

Annotation[] annotations = clazz.getAnnotations();

**for**(Annotation annotation: annotations){

System.*out*.println(annotation);

}

System.*out*.println("@SuppressWarnings上的注解");

System.*out*.println(clazz.getAnnotation(SuppressWarnings.**class**));

System.*out*.println("全部内部类");

Class<?>[] innerClasses = clazz.getDeclaredClasses();

**for**(Class<?> innerClass: innerClasses){

System.*out*.println(innerClass);

}

}

}

全部构造函数

private com.test.test.ClassTest()

public com.test.test.ClassTest(java.lang.String)

全部public构造函数

public com.test.test.ClassTest(java.lang.String)

全部方法

public void com.test.test.ClassTest.showClassInformation()

public void com.test.test.ClassTest.info()

public void com.test.test.ClassTest.info(java.lang.String)

全部public方法

public void com.test.test.ClassTest.showClassInformation()

public void com.test.test.ClassTest.info()

public void com.test.test.ClassTest.info(java.lang.String)

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

带一个字符串为参数的info方法

public void com.test.test.ClassTest.info(java.lang.String)

全部Annotation

@java.lang.Deprecated()

@SuppressWarnings上的注解

null

全部内部类

class com.test.test.ClassTest$Inner

注意:虽然我们定义ClassTest类时使用了@SuppressWarning注释，但程序运行时无法分析出该类型包含的该注释，这是因为@SuppressWarnings使用了@Retengion(value=SOURCE)修饰，这表明@SuppressWarnings只能保存在源代码级别上，而通过classTest.class获取该类的运行时Class对象，所以程序无法访问到@SuppressWarnings注释

**使用反射操作并生成对象**

使用反射来生成对象有如下两种方式：

1 使用Class对象的newInstance方法来创建该Class对象对应类的实例，这种方法要求该Class对象的对应类有默认构造器，而执行newInstance方法实际上是利用默认构造器来创建该类的实例**（即通过调用Class的newInstance方法）**

2 先使用Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance方法来创建该Class对象对应类的实例，通过这种方式可以选择使用指定的构造器来创建实例**（即通过调用Constructor的newInstance方法）**

~~ClassTest~~ object = **null**;

Class<~~ClassTest~~> clazz = ~~ClassTest~~.**class**;

**try**{

// 首先通过Class创建对象

object = (~~ClassTest~~)clazz.newInstance();

}**catch**(Exception e){

// 如果发生异常则通过构造函数创建对象

Constructor constructor = **null**;

**try**{

constructor = clazz.getConstructor((Class[])**null**);

**constructor.setAccessible(true);**

}**catch**(NoSuchMethodException ex){

ex.printStackTrace();

}

**try**{

object = (~~ClassTest~~)constructor.newInstance((Class[])**null**);

}**catch**(Exception ex1){

ex1.printStackTrace();

}

}

System.*out*.println(object);

object.info();

在上面的代码中如果有无参数的构造器，但是是私有的则创建不成功，如果没有无参数的构造器，则也会出现异常，在正常有无参数构造器且为public的情况下：



**同时我们也可以采用指定的构造函数进行反射**

~~ClassTest~~ object = **null**;

Class<~~ClassTest~~> clazz = ~~ClassTest~~.**class**;

**try**{

Constructor constructor = clazz.getConstructor(String.**class**);

object = (~~ClassTest~~)constructor.newInstance("text");

}**catch**(Exception e){

e.printStackTrace();

}

System.*out*.println(object);

object.info();



**调用方法**

当获得某个类对应的Class对象后，就可以通过该Class对象获取方法，每个Method对象对应一个方法，获得Method对象后，程序就可通过该Method来调用它对应的方法，在Method里包含一个invoke方法

**try**{

~~ClassTest~~ classTest = **new** ClassTest();

Method method = clazz.getMethod("info", String.**class**);

**method.invoke(classTest, "this");**

}**catch**(Exception e){

e.printStackTrace();

}

当通过Method的invoke方法来调用对应的方法时，java会要求程序必须有调用该方法的权限，如果程序确实需要调用某个对象的private方法，则可以先调用Method对象的如下方法：

setAccessuvke(Boolean flag)：将Method对象的accessible设置为指定的boolean值。值为true，表示该Method在使用时应该取消java语言的访问权限检查；值为false，则表示该Method在使用时要实施java语言的访问检查

实际上，setAccessible方法并不属于Method，而是属于它的父类AccessibleObject。因此Method、Constructor、Field都可以调用该方法，从而实现通过反射来调用private方法、private构造器和private属性，也就是说，我们可以通过调用该方法来取消访问权限检查，通过反射即可访问private成员。下面是例子：

~~ClassTest~~ object = **null**;

Class<~~ClassTest~~> clazz = ~~ClassTest~~.**class**;

**try**{

Constructor constructor = clazz.**getConstructor**(String.**class**);

constructor.setAccessible(**true**);

object = (~~ClassTest~~)constructor.newInstance("text");

}**catch**(Exception e){

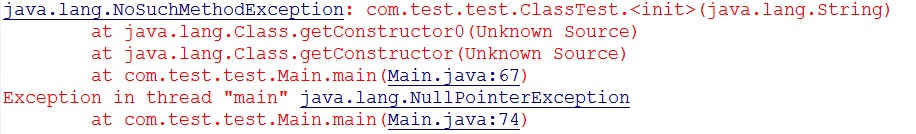
e.printStackTrace();

}

System.*out*.println(object);

object.info();

在上面的例子中ClassTest的带String参数的构造函数是private的。



修改上面的代码如下：

~~ClassTest~~ object = **null**;

Class<~~ClassTest~~> clazz = ~~ClassTest~~.**class**;

**try**{

Constructor constructor = clazz.**getDeclaredConstructor**(String.**class**);

constructor.setAccessible(**true**);

object = (~~ClassTest~~)constructor.newInstance("text");

}**catch**(Exception e){

e.printStackTrace();

}

System.*out*.println(object);

object.info();



**注意：getDeclaredConstructor能返回指定参数的构造函数，getConstructor只能返回指定参数的public构造函数**

**访问属性值**

**public** **class** People {

**private** String name;

**private** **int** age;

**public** People(){

}

**public** People(String name, **int** age){

**this**.name = name;

**this**.age = age;

}

**public** **void** show(){

System.*out*.println("name = " + name + ", age = " + age);

}

}

Main函数：

**try**{

People object = **new** People();

Class<?> clazz = People.**class**;

Field field = (Field)clazz.**getDeclaredField**("age");

**field.setAccessible(true);**

field.set(object, 100);

field = (Field)clazz.**getDeclaredField**("name");

**field.setAccessible(true);**

field.set(object, "Li lei");

object.show();

}**catch**(Exception e){

e.printStackTrace();

}

