# 反射

# 前言

正如其它知识点一样，反射并不是孤立存在的。所以在正式的开始反射的知识点之前，我们需要先铺垫一下Java中类的加载。**类的加载是属于偏底层的知识点**，所以在学习阶段会感觉晦涩难懂，但是掌握这些底层的运行原理却可以让我们对Java的理解更加透彻，使用起来更加游刃有余。因此我们先来看看类的加载。

# 类的加载

类的加载，也叫类的初始化，是将.class文件加载到内存，并为之创建一个java.lang.Class对象的过程。**任何类被使用时系统都会为之建立一个Class对象**。这一过程被具体的分成了三步 ：加载，连接，初始化。

## 加载

加载就是查找并加载类的二进制数据。

系统可能在第一次使用某个类的时候加载该类，也可能采用预加载机制加载某个类。Java虚拟机规范允许系统预先加载某些类。加载工作是由**类加载器**来完成，类加载器通常由JVM提供，这些由JVM提供的类加载器称为系统类加载器，它们是Java程序运行的基础。除此之外，开发者可以通过继承ClassLoader来自定义类加载器。

### 类加载器简介

类加载器负责加载所有的类，它把(磁盘上,网络上).class文件加载到内存中，并为之生成一个对应的java.lang.Class对象。同一个类只会被加载一次，在项目源代码中Java使用全限定类名来区分唯一的一个类，但是在JVM中，使用全限定类名和其类加载器共同区分一个唯一的类。

例如，有xiaoka.Person类，被类加载器ClassLoader的实例C1加载，那么该Person类在JVM中表示为(Person、xiaoka、C1)。这意味着，两个类加载器加载的同名类：(Person、xiaoka、C1)和(Person、xiaoka、C2)是不同的，它们所加载的类也是完全不同，完全不兼容的。

### 类加载器的组成

类加载器的组成有四种，分别是

* Bootstrap ClassLoader 根类加载器
* Extension ClassLoader 扩展类加载器
* Sysetm ClassLoader 系统类加载器
* User-defined ClassLoader用户自定义加载器

JVM启动时，会使用前三个类加载器组成初始类加载器层次结构。下面介绍每种类加载器的作用。

#### Bootstrap ClassLoader

根类加载器，也被称为引导类加载器、启动类加载器、原始类加载器。除了根类加载器，其它三种类加载器都是java.lang.ClassLoder的子类。根类加载器比较特殊，它是由JVM自身实现，所以根类加载器并不是ClassLoader的子类。

根类加载器负责Java核心类库的加载。比如System,String等。在JDK中JRE的lib目录下rt.jar文件中。

#### Extension ClassLoader

扩展类加载器，负责加载JRE扩展目录中的jar包。 这些jar包在JDK中JRE的lib目录下ext目录中，或者在java.ext.dirs系统属性指定的目录。

通过这种方式，就可以为Java扩展核心类以外的新功能。只需要把自己开发的类打成jar包，然后放入到JAVA\_HOME/jre/lib/ext/中即可。

#### Sysetm ClassLoader

系统类加载器也叫应用类加载器，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，或者classpath环境变量所指定的jar包和路径。

使用ClassLoader的静态方法getSystemClassLoader()来获取系统类加载器。

#### User-defined ClassLoader

用户自定义的类加载器，开发者直接用代码实现的类加载器。开发者可以根据需求动态的加载所需要的类。如果没有特别指定，则用户自定义的类加载器都以ClassLoader作为父加载器。

### 类的来源

通过类加载器可以从不同来源加载类的二进制数据，通常类的来源有如下几种：

* 从本地文件系统加载.class文件，这是目前为止绝大部分.class文件的来源
* 从jar包加载.class文件，这种方式也很常见，例如加载数据库的驱动类。JVM可以直接从JAR文件中加载.class文件。
* 通过网络加载.class文件
* 把一个.java源文件动态编译，执行加载

### 类加载机制

JVM的类加载机制主要有如下三种：

* 全盘负责：当一个类加载器负责加载某个类时，该类所依赖的和引用的其它类也由该类加载器负责加载，除非显示使用另外一个加载器来加载。
* 父类委托：首先这里的父类并不是继承上的父子关系，而是指加载器实例之间的关系（如图所示）。当某个类需要被加载时，先让父加载器尝试加载该类，只有父加载器无法加载该类时，才会尝试从自己的路径中加载该类。



* 缓存机制：缓存机制将会保证所有加载过的类都被缓存。当一个类加载器被请求加载类时，它首先按照顺序在上层加载器、父加载器以及自身的加载器的缓存里检查这个类是否已经存在。简单来说，就是在缓存里查看这个类是否已经被自己加载过了，如果没有的话，继续查找父类的缓存，直到在bootstrap类加载器里也没有找到的话，它就会自己尝试加载这个类。

### 类加载的详细过程

检测此Class是否载入过(即检查缓存区中是否有此Class)

有:则直接返回对应的java.lang.Class对象,加载结束

没有：判断当前类加载器是否具有父类加载器

有：请求父类加载器载入目标类，

载入成功：则返回对应的java.lang.Class对象，加载结束

载入失败：当前类加载器从与此ClassLoader相关的类路径中寻找目标类

寻找到目标类：从文件中载入Class,返回对应的java.lang.Class,加载结束

没有寻找到目标类：抛出异常ClassNotFoundException

没有：这种情况要么父类加载器是根类加载器，要么本身是根类加载器,所以会请求使用根类加载器加载目标类，载入成功，则返回对应的java.lang.Class对象，加载结束。否则抛出异常ClassNotFoundException

### 类加载的三件事情

在加载阶段（可以参考java.lang.ClassLoader的loadClass()方法），虚拟机需要完成以下3件事情：

* 通过一个类的全限定名来获取定义此类的二进制字节流（并没有指明要从一个Class文件中获取，可以从其他渠道，譬如：网络、动态生成、数据库等）；
* 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；
* 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

## 连接

当类被加载之后，系统会生成对应的Class对象，接着将进入连接阶段。连接阶段负责把类的二进制数据合并到JRE中。它具体的分成了三步：验证，准备，解析（这三步统称为连接）

1. 验证

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

1. 准备

负责为类的静态成员变量分配内存，并设置默认初始化值

1. 解析

将类的二进制数据中的符号引用替换为直接引用，即：将引用类型直接替换成指针。

## 初始化

初始化是一个类被加载过程中的最后阶段，该阶段Java虚拟机的主要工作是为类中的静态变量进行初始化赋值。在Java中给静态变量初始化赋值只有两种方式：

* 声明变量时指定初始值，例如 static int a = 23;
* 使用静态代码块给静态变量指定初始值 例如 statc int b; static{b=27;}

## 总结

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。





# 反射

## 反射的概念

JAVA反射机制是在程序运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取信息以及动态调用对象的方法的功能称为java语言的反射机制。

要想解剖一个类,必须先要获取到该类的字节码文件对象【.class文件】。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象。

## 三种获取字节码对象的方式

方式1:

Person p = new Person();

Class c = p.getClass();

通过对象获取字节码文件对象

方式2:

Class c2 = Person.class;

任意数据类型都具备一个class静态属性,看上去要比第一种方式简单.

方式3:

Class c3 = Class.forName("com.xiaoka.Person"); //传入不懂的值，获得不同的Class,体现动态

将类的全限定类名作为字符串传递给Class类中的静态方法forName即可。

刚刚提到，任何数据类型都有一个Class对象，所以

Class c4 = int.class;

System.out.println(c4);//运行结果为”int”。

可见即使基本数据类型不是引用类型，也会有一个对应的字节码文件

## 字节码对象的使用

### 一：获取构造方法

只能获取到public修饰的构造方法

Constructor getConstructor(Class<T>... parameterTypes);

Constructor[] getConstructors();

可以获取到所有声明出来的构造方法,但是如果该构造方法是私有的，那么依然无法在外界直接使用

可以通过setAccessible(true)来设置暴力访问来达到即使私有的构造方法也能在外界访问的目的。

此方法导致了不安全，入侵性很强

Constructor getDeclaredConstructor(Class<T>... parameterTypes);

Constructor[] getDeclaredConstructors();

示例代码：

**public** **class** Person {

**private** String name;

**int** age;

**public** String address;

**public** Person() {

**super**();

}

**public** Person(String name) {

**super**();

**this**.name = name;

}

**public** Person(**int** age) {

**super**();

**this**.age = age;

}

**public** Person(String name, **int** age, String address) {

**super**();

**this**.name = name;

**this**.age = age;

**this**.address = address;

}

**public** **class** Test {

**public** **static** **void** main(String[] args) **throws** NoSuchMethodException, SecurityException, InstantiationException,

IllegalAccessException, IllegalArgumentException, InvocationTargetException {

// 获取字节码对象

Class p = Person.**class**;

//通过字节码对象获取Person类的构造方法 根据参数获取不同的构造方法

// 获取空参构造

Constructor kongCanGouZhao = p.getConstructor();

// 设置暴力访问，使得私有构造也可以在外界被访问

kongCanGouZhao.setAccessible(**true**);

// 获取参数为一个int类型的带参构造

Constructor intConstructor = p.getConstructor(**int**.**class**);

// 获取三个参数的构造方法

Constructor constructor = p.getConstructor(String.**class**, **int**.**class**, String.**class**);

// 使用构造方法创建对象

Object newInstance = kongCanGouZhao.newInstance();

**if** (newInstance **instanceof** Person) {

Person p1 = (Person) newInstance;

System.***out***.println(p1);

}

Object newInstance2 = intConstructor.newInstance(23);

**if** (newInstance2 **instanceof** Person) {

Person p2 = (Person) newInstance2;

System.***out***.println(p2);

}

Object newInstance3 = constructor.newInstance("小钻风", 23, "上海·浦东");

**if** (newInstance3 **instanceof** Person) {

Person p3 = (Person) newInstance3;

System.***out***.println(p3);

}

}

}

### 二：获取普通方法

public Method[] getMethods() 获取公共方法，包含父类继承的方法

public Method[] getDeclaredMethods() 获取本类中声明的方法[包括本类private,不包括父类任何方法]

public Method getMethod(String name,Class... parameterTypes)

public Method getDeclaredMethod(String name,Class... parameterTypes)

public Object invoke(Object obj,Object... args)

示例代码

//创建字节码文件对象

Class<Person> clazz = Person.class;

//获取普通方法

Method[] methods = clazz.getMethods();

Method[] declaredMethods = clazz.getDeclaredMethods();

//获取具体的普通方法

Method method = clazz.getDeclaredMethod("method");

Method method2 = clazz.getDeclaredMethod("method",int.class);

//调用方法对象的方法用P也可以，用newInstance也可以。

Person p = new Person();

Constructor<Person> declaredConstructor = clazz.getDeclaredConstructor();

Person newInstance = declaredConstructor.newInstance();

method.invoke(newInstance);

method2.setAccessible(true);

//newInstance对象来调用method（int）方法，实际参数是100

method2.invoke(newInstance, 100);

### 三：获取成员属性[字段/域]

public Field getDeclaredField(String name) 反射获取字段对象

public void set(Object obj, Object value) 给某个对象的某个属性赋某个值

示例代码

//创建字节码文件对象

Class clazz = Class.forName("cn.xiaoka.Person");

//反射字段

Field declaredField = clazz.getDeclaredField("sex");

//调用字段对象的方法，修改某个对象的该字段的值

Person p = new Person("小钻风", "男");

declaredField.setAccessible(true);

declaredField.set(p, "女");

### 四：反射习题

绕过ArrayList<Integer>的一个对象的泛型检查，在这个集合中添加一个字符串数据。只需要将add方法使用反射的方式调用即可。

原因：泛型检查存在擦除泛型的动作(即编译器认识泛型，而虚拟机不认识泛型)，真正在运行时，仍然是泛型位置使用的是Object

示例代码

ArrayList<Integer> arrayList = new ArrayList<>();

arrayList.add(10);

//arrayList.add("abc");

//使用反射，将add方法反射

Class clazz = arrayList.getClass();

//这里的参数不能写Integer.class，这是因为根本没有这个方法，所有的泛型实际上都是Object

Method declaredMethod = clazz.getDeclaredMethod("add", Object.class);

declaredMethod.invoke(arrayList, 10);

declaredMethod.invoke(arrayList, "abc");

System.out.println(arrayList);

# 附录

## 动态语言的概念

所谓动态语言是指，程序在运行期间可以改变程序结构或者变量类型。典型的动态语言有Python,Ruby,JavaScript。而C,C++和Java都不是动态语言。

虽然Java并不是动态语言，但是Java却有一定的动态性。利用Java的反射机制，字节码操作等方法就能够获得类似动态语言的特征。因此Java可以称之为”准动态语言”。Java的动态性让编程变得更加灵活。

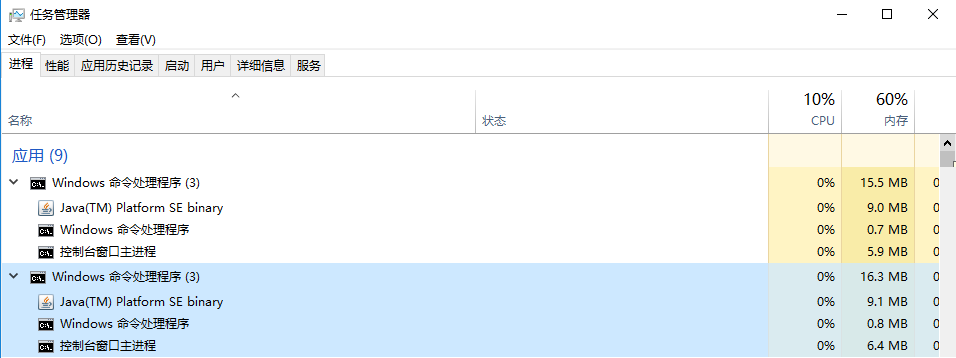
## JVM和类

每一个Java程序启动的时候都会开启一个Java虚拟机进程，这个Java程序中所有的所有线程，所有变量都处于该Java虚拟机进程中，它们都使用该Java虚拟机进程的内存区。

也就是所程序A会开启一个Java虚拟机进程，程序B也会开启一个Java虚拟机进程，而这两个Java虚拟机进程之间的内存是独立的，并不会共享数据。

如图所示 ：

我们通过控制台运行两个Java程序(带main方法的Java类)，发现每一个Java程序都会独立的开启一个进程。



接下来，再使用代码测试两个Java虚拟机进程之间的内存不共享 ：

一：创建Java工程

略

二：创建Share类

**public** **class** Share {

**public** **static** **int** *share* = 0;

**public** **static** List<Integer> *list* = **new** ArrayList<>();

}

三：创建Demo01.java

**public** **class** Demo01 {

**public** **static** **void** main(String[] args) **throws** Exception {

**while** (**true**) {

Thread.*currentThread*().*sleep*(1000);

Share.*share*++;

Share.*list*.add(Share.*share*);

System.***out***.println("Share.share :"+Share.*share*);

System.***out***.println("Share.list.size() : "+Share.*list*.size());

}

}

}

四：创建Demo02.java

**public** **class** Demo02 {

**public** **static** **void** main(String[] args) **throws** Exception {

**while** (**true**) {

Thread.currentThread().sleep(1000);

System.**out**.println("Share.share :"+Share.share);

System.**out**.println("Share.list.size() : "+Share.list.size());

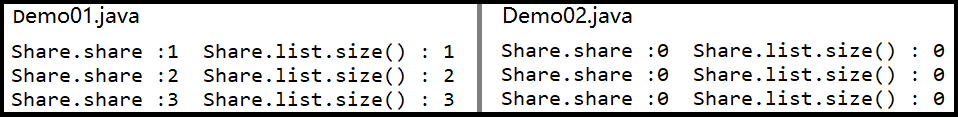
}

}

}

五：运行Demo01.java和Demo02.java

查看控制台打印，发现，两个Java虚拟机进程(分别由Demo01和Demo02开启)间内存不是共享的。



当Java程序出现以下情况的时候，Java虚拟机进程终止，Java程序结束。

* Java程序运行到最后正常结束
* Java程序中运行到调用了System.exit()或者Runtime.getRuntime().exit()代码处，结束程序
* Java程序运行中遇到了未捕获的异常或者遇到了错误，程序结束
* Java程序所在平台强制结束了Java虚拟机进程

## 类的加载

### 加载父类而不加载子类

对于静态字段，只有直接定义这个字段的类才会被加载，因此通过其子类来引用父类中定义的静态字段，只会触发父类的加载而不会触发子类的加载

一：创建Java工程

略

二：创建父类Animal.java

**public** **class** Animal {

**public** **static** String *name* = "Animal";

**static** {

System.***out***.println("loading Animal.class");

}

}

三：创建子类Cat.java

**public** **class** Cat **extends** Animal{

**static** {

System.***out***.println("loading Cat.class");

}

}

四：创建测试类Test.java

**public** **class** Test {

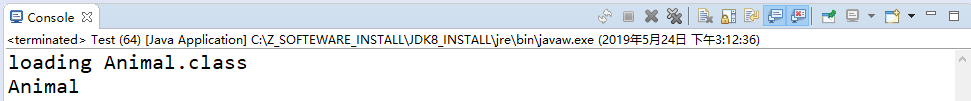
**public** **static** **void** main(String[] args) {

System.***out***.println(Cat.*name*);

}

}

五：查看控制台输出



可以看到子类Cat.class并没有被加载，而是加载了父类Animal.class。

### 类不被加载

对于final类型的静态变量，如果该变量的值在编译期就可以确定下来，那么该静态变量相当于”宏变量”。Java编译器会在编译期直接将该变量出现的地方替换成它的值。因此即使程序使用该静态变量，也不会导致该类的初始化。

总结：使用在编译器就能够确定值的静态常量不会初始化类

示例代码

**public** **class** Share {

**static** {

System.***out***.println("loading Share");

}

**public** **static** **final** **int** ***share*** = 23;

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

System.***out***.println(Share.***share***);

}

}

运行测试类，可以发现Share类没有被加载，而将关键字final去掉以后运行，则能看到加载Share类的输出语句。之所以会出现这种现象是因为，程序在使用的时候，并没有使用该静态变量，而是直接使用了常量。

反之，如果该静态常量的值在编译期间不能够确定，必须在运行时才能够确定，这个时候如果访问它，则会导致类的初始化。

示例代码

**public** **class** Share {

**static** {

System.***out***.println("loading Share");

}

**public** **static** **final** String ***CUR\_TIME*** = System.*currentTimeMillis*() + "";

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

System.***out***.println(Share.***CUR\_TIME***);

}

}

这种情况之所以会导致Share类的初始化是因为Share类的CUR\_TIME的值必须在运行时才能确定，所以Share类必须保留对于CRU\_TIME变量的引用，这个时候CRE\_TIME其实就称为了静态变量，而使用类的静态变量则会初始化类。

### Class.forName()和loadClass()

ClassLoader类的loadClass()方法只负责加载类，并不会执行类的初始化。使用Class.forName()方法才会导致强制初始化类。

示例代码

**public** **class** Share {

**static** {

System.***out***.println("loading Share");

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) **throws** Exception {

ClassLoader classLoader = ClassLoader.*getSystemClassLoader*();

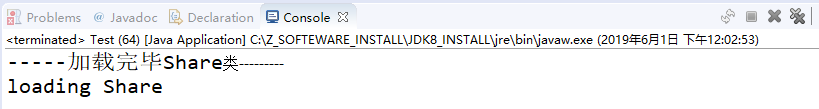
classLoader.loadClass("com.reflect.Share");

System.***out***.println("-----加载完毕Share类---------");

Class.*forName*("com.reflect.Share");

}

}



运行测试类发现classLoader.loadClass()函数根本不会初始化类，只会加载类。

## 获得根类加载器所加载的核心类库

**public** **class** BootStrapClassLoaderTest {

**public** **static** **void** main(String[] args) {

URL[] urls = sun.misc.Launcher.*getBootstrapClassPath*().getURLs();

**for** (URL url : urls) {

System.***out***.println(url.toExternalForm());

}

}

}

上面程序编译可能会遇到警告或者错误，无需理会，直接执行即可。

控制台打印结果

file:/绝对路径 /java\_home路径/JRE/lib/resources.jar

file:/ 绝对路径/java\_home路径/JRE/lib/rt.jar

file:/ 绝对路径/java\_home路径/JRE/lib/sunrsasign.jar

file:/ 绝对路径/java\_home路径/JRE/lib/jsse.jar

file:/ 绝对路径/java\_home路径/JRE/lib/jce.jar

file:/ 绝对路径/java\_home路径/JRE/lib/charsets.jar

file:/ 绝对路径/java\_home路径/JRE/lib/jfr.jar

file:/ 绝对路径/java\_home路径/JRE/classes

通过打印结果，可以明白为什么可以在系统中使用System,String等这些类，因为这些核心类已经被加载到了JVM的内存中。其实这些类就在file:/C:/Z\_SOFTEWARE\_INSTALL/JDK8\_INSTALL/JRE/lib/rt.jar包中，可以直接通过解压缩软件在硬盘中看到。

## 获得JVM的类加载器

**public** **class** GainClassLoader {

**public** **static** **void** main(String[] args) **throws** Exception {

//获得系统类加载器

ClassLoader systemClassLoader = ClassLoader.*getSystemClassLoader*();

System.***out***.println(systemClassLoader);

/\*

\* 获得系统类加载器的加载路径：通常由CLASSPATH环境变量指定

\* 如果没有指定CLASSPATH环境变量，则默认以当前路径作为系统类加载器的加载路径

\*

\*/

Enumeration<URL> resources = systemClassLoader.getResources("");

**while** (resources.hasMoreElements()) {

URL element = resources.nextElement();

System.***out***.println(element);

}

//获得系统类加载器的父加载器：扩展类加载器

ClassLoader extensionLoader = systemClassLoader.getParent();

System.***out***.println(extensionLoader);

//获得扩展类加载器的加载路径

String property = System.*getProperty*("java.ext.dirs");

System.***out***.println(property);

//获得扩展类加载器的父加载器：根类加载器

ClassLoader classLoader = extensionLoader.getParent();

System.***out***.println(classLoader);

}

}

程序的运行结果可以看到：

扩展类加载器的父类加载器是null，这是因为根类加载器不是Java实现的，根类加载器并没有继承ClassLoader。虽然如此，但是我们依然可以明确，扩展类加载器的父类加载器是根类加载器。

系统类加载器是sun.misc.Launcher$AppClassLoader，

扩展类加载器是sun.misc.Launcher$ExtClassLoader。

AppClassLoader和ExtClassLoader都是java.net.URLClassLoader的子类。

URLClassLoader是ClassLoader的子类，所以除了根类加载器，其它类加载器都是ClassLoader的子类。

## 自定义类加载器

### 定义类加载器类

一：创建MyClassLoader.java,继承ClassLoader

**public** **class** MyClassLoader **extends** ClassLoader {}

二：创建读取文件的方法

**public** **class** MyClassLoader **extends** ClassLoader {

// 读取一个文件的内容到字节数组

**private** **byte**[] getBytes(String fileName) **throws** Exception {

File file = **new** File(fileName);

**long** length = file.length();

**byte**[] bytes = **new** **byte**[(**int**) length];

FileInputStream inputStream = **new** FileInputStream(file);

**int** read = inputStream.read(bytes);

**if** (read != length) {

**throw** **new** IOException("无法读取全部文件");

}

**return** bytes;

}

}

三：创建编译.java文件的方法

**public** **class** MyClassLoader **extends** ClassLoader {

**private** **byte**[] getBytes(String fileName) **throws** Exception{…}

// 定义编译指定Java文件的方法

**private** **boolean** compileFile(String javaFile) **throws** Exception {

System.***out***.println("--正在编译Java文件--" + javaFile);

// 调用系统javac命令

Process process = Runtime.*getRuntime*().exec("javac " + javaFile);

**try** {

// 其它线程都等待这个线程完成

process.waitFor();

} **catch** (Exception e) {

e.printStackTrace();

}

// 获取javac线程的退出值

**int** exitValue = process.exitValue();

// 返回编译是否成功

**return** exitValue == 0;

}

}

四：重写findClass方法

**public** **class** MyClassLoader **extends** ClassLoader {

**private** **byte**[] getBytes(String fileName) **throws** Exception {…}

**private** **boolean** compileFile(String javaFile) **throws** Exception {…}

@Override

**protected** Class<?> findClass(String name) **throws** ClassNotFoundException {

Class clazz = **null**;

System.***out***.println(name);

String realPath = name.replaceAll("/.", "//");

System.***out***.println(realPath);

String javaFile = realPath + ".java";

String classFile = realPath + ".class";

System.***out***.println(javaFile);

System.***out***.println(classFile);

File javaF = **new** File(javaFile);

File classF = **new** File(classFile);

//如果.java文件存在，.class文件不存在，重新编译

//如果.java文件最后修改时间大于.class文件，重新编译

**if** (javaF.exists() && (!classF.exists()) || javaF.lastModified() > classF.lastModified()) {

**try** {

**boolean** compileFile = compileFile(javaFile);

**if** (!compileFile || !classF.exists()) {

**throw** **new** ClassNotFoundException("class not found exception :" + javaFile);

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

**if** (classF.exists()) {

**try** {

//将编译的.class文件读取成二进制字节数组

**byte**[] bytes = getBytes(classFile);

//将字节数组转换成Class对象，该函数是父类被final重写的

clazz = defineClass(name, bytes, 0, bytes.length);

} **catch** (Exception e) {

e.printStackTrace();

}

}

**if** (clazz == **null**) {

**throw** **new** ClassNotFoundException(name);

}

**return** clazz;

}

五：创建main方法

**public** **class** MyClassLoader **extends** ClassLoader {

**private** **byte**[] getBytes(String fileName) **throws** Exception {…}

**private** **boolean** compileFile(String javaFile) **throws** Exception {…}

**protected** Class<?> findClass(String name) **throws** ClassNotFoundException {…}

**public** **static** **void** main(String[] args) **throws** Exception {

**if**(args.length < 1){

System.***out***.println("缺少目标类，请按如下格式运行java源文件");

System.***out***.println("java MyClassLoader ClassName");

}

//第一个参数是需要运行的类

String targetClass = args[0];

//其它参数是运行目标类时的参数

String[] tempArgs = **new** String[args.length-1];

//将这些参数复制到数组中

System.*arraycopy*(args,1,tempArgs,0,tempArgs.length);

Demo04\_MyClassLoader classLoader = **new** Demo04\_MyClassLoader();

//加载需要运行的类

Class<?> clazz = classLoader.loadClass(targetClass);

Method method = clazz.getMethod("main", (**new** String[0]).getClass());

Object[] tempArgsObj = {tempArgs};

method.invoke(**null**, tempArgsObj);

}

}

### 定义目标类

**public** **class** Hello {

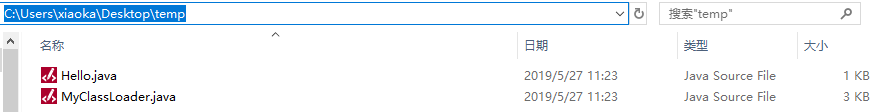
**public** **static** **void** main(String[] args) {

System.***out***.println("+++++Hello World+++++++");

}

}

至此，所需要的类创建完毕，如图示

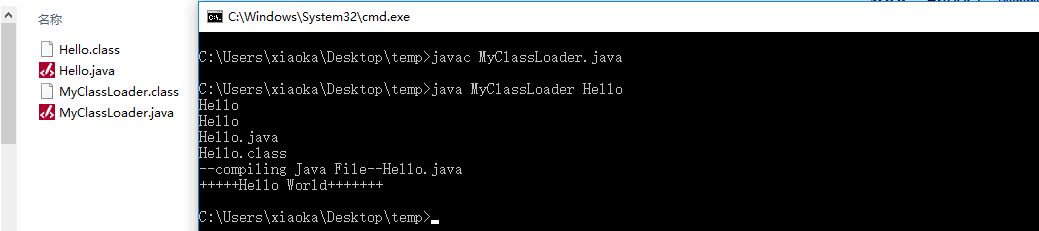


### 测试运行

第一步：编译MyClassLoader.java

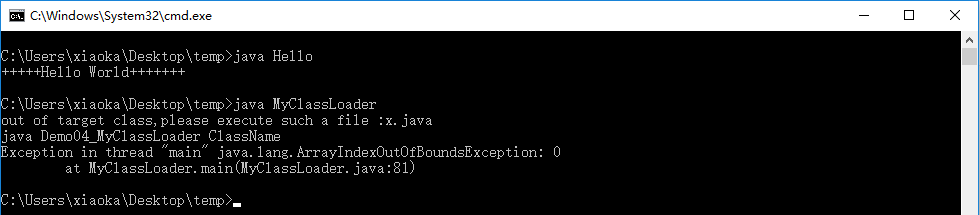


第二步：运行MyClassLoader



将Hello作为参数给MyClassLoader，它对Hello.java进行了编译，和运行。

可以再次测试，编译出来的Hello.class可以正常的运行，而运行MyClassLoader不加参数则会抛异常。



## 类的初始化

### 类的初始化时机

类初始化是类加载过程的最后一个阶段，到初始化阶段，才真正开始执行类中的 Java 程序代码。虚拟机规范严格规定了有且只有四种情况必须立即对类进行初始化：

* 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类还没有进行过初始化，则需要先触发其初始化生成这四条指令最常见的 Java 代码场景是：
  + 创建类的实例对象：包括使用new关键字创建对象，使用反射创建对象，或者使用反序列化的方式创建对象。
  + 调用类的静态方法
  + 访问类静态变量或者为该类的静态变量赋值
  + 访问接口的静态变量
* 使用反射方式创建类或者接口对应的java.lang.Class对象，如果类还没有进行过初始化，则需要先触发其初始化。即：Class.froName(“target”)将导致target类的初始化
* 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
* 当虚拟机启动时，用户需要指定一个要执行的主类（包含main方法的类），虚拟机会先初始化该主类。

### 类的初始化顺序

**声明静态变量并赋值，以及静态代码块都会被当做类的初始化语句**。对于这两种初始化语句的执行顺序，JVM是按照这些语句在源码中的排列顺序来执行的。

示例代码一：

**public** **class** Test {

**static** **int** *k* = 23;

**static** {

*k* = 27;

}

**public** **static** **void** main(String[] args) {

System.***out***.println(Test.*k*);//27

}

}

示例代码二

**public** **class** Test {

**static** {

*k* = 27;

}

**static** **int** *k* = 23;

**public** **static** **void** main(String[] args) {

System.***out***.println(Test.*k*);//23

}

}

### 子父类的初始化顺序

现有如下继承关系

class A extends B

class B extends C

类的初始化顺序是：永远先初始化父类，即，当需要A的时候，一定会保证C最先被初始化完成。具体过程如下：

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. 假如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. 假如该类中有初始化语句，则系统依次执行这些初始化语句

当执行第2步时，系统对该类的直接父类也执行1-3步，依次类推，所以JVM最先初始化的总是java.lang.Object。从这里可以看出来，类的加载和初始化是一个典型的递归的过程。当程序主动使用任何一个类的时，系统会保证该类以及所有的父类（包括直接父类和间接父类）都会被初始化。

## 静态加载&动态加载

Java初始化一个类的时候可以用new 操作符来初始化，也可通过Class.forName的方式来得到一个Class类型的实例，然后通过这个Class类型的实例的newInstance来初始化。

我们把前者叫做JAVA的静态加载，把后者叫做动态加载。

静态加载的时候如果在运行环境中找不到要初始化的类,抛出的是NoClassDefFoundError,它在JAVA的异常体系中是一个Error。

动态态加载的时候如果在运行环境中找不到要初始化的类,抛出的是ClassNotFoundException,它在JAVA的异常体系中是一个checked异常,在写代码的时候就需要catch。