#### 反射

class:定义Java类时的关键字.

Class:Java程序中的各个Java类,它们是否属于同一类事物,是不是可以用一个类来描述这类事物呢?这个类的名字就是Class .

有9个预定义的Class对象：8个基本类型+void.class .

这8种基本类型分别是： boolean , char ,byte , short ,int ,long ,float ,double .

数组类型的Class实例对象：Class.isArray .总之，只要是在源程序中出现的类型，都有各自的Class实例对象，例如 int[] ,void ...

Class.forName作用：返回字节码;

返回方式1：这份字节码曾经被加载过,已经在java虚拟机中了,直接返回;

返回方式2：java虚拟机中还没有这份字节码,用类加载器去加载,把加载进来的字节码缓存在虚拟机中,以后要得到这份字节码就不用加载了.

反射就是把Java类中的各种成分映射成相应的Java类.例如，一个Java类用一个Class类的对象来表示，一个类中的组成部分：成员变量、方法、构造方法、包等等信息也用一个个的Java类来表示，就像汽车是一个类，汽车中的发动机、变速箱等等也是一个个的类。表示java类的Class类显然要提供一系列的方法,来获得其中的变量，方法，构造方法，修饰符，包等信息，这些信息就是用相应类的实例对象来表示，他们是Field、Method、Contructor、Package等等。

一个类中的每个成员都可以用相应的反射API类的一个实例对象来表示，通过调用Class类的方法可以得到这些实例对象后，得到这些实例对象后又什么用呢？ 怎么用呢？ 这正是学习和应用反射的要点。

Constructor类代表某个类中的一个构造方法。

1.得到某个类所有的构造方法：

Constructor[] constructors = Class.forName("java.lang.String").getConstructors();

2.得到某一个构造方法：

Constructor constructor =Class.forName("java.lang.String").getConstructor(StringBuffer.class);

//获得构造方法时要用到类型

3.创建实例对象：

通常方式：String str = new String(new StringBuffer("abc"));

反射方式：String str2 = (String)constructor.newInstance(new StringBuffer("abc")); // 调用获得的方法时要用到上面相同类型的实例对象

4.Class.newInstance()方法：

String obj = Class.forName("java.lang.String").newInstance();

该方法内部先得到默认的构造方法，然后用该构造方法创建实例对象。该方法内部的具体是怎样的呢？用到了缓存机制来保存默认构造方法的实例对象。

反射会导致程序性能下降。

Field 代表成员变量 。

Method类代表某个类中的一个成员方法。 //专家模式

1.得到类中的某一个方法：

Method charAt = Class.forName("java.lang.String").getMethod("charAt",int.class);

2.调用方法：

通常方式: System.out.println(str.charAt(1));

反射方式：System.out.println(charAt.invoke(str,1));

3.jdk1.4和Jdk1.5的区别：

jdk1.5: public Object invoke(Object obj, Object... args);

jdk1.4: public Object invoke(Object obj, Object[] args),即按jdk1.4的语法，

需要将一个数组作为参数传递给invoke方法时,数组中的每个元素分别对应被调用方法中的一个参数,所以，调用charAt方法的代码也可以用jdk1.4改写为：

charAt.invoke("str",new Object[]{1})形式.

数组的反射：

具有相同维度和元素类型的数组属于同一个类型，即具有相同的Class实例对象.

代表数组的Class实例对象的getSuperclass()返回父类Object类对应的Class .

基本类型的一维数组可以被当作Object类型使用,不能当作Object[]类型使用；非基本类型的一维数组既可以被当作Object类型使用,也可以当作Object[]类型使用 。

Arrays.asList()方法处理int[] 与 String[]时的差异，见ReflectTest中的示例代码.

java.lang.reflect.Array工具类用于完成对数组的反射操作.

怎么得到一个数组的类型？不能得到，但可以得到数组中元素的类型.

Object[] a = new Object[]{1,"abc"};

a[0].getClass().getName();

反射的作用--->实现框架(Framework)功能.

框架要解决的核心问题：

我做房子卖给用户住，由用户自己安装门窗和空调，我做的房子就是框架，用户需要使用我的框架，把门窗插入进我提供的框架中。

框架与工具的区别，工具类被用户的类调用，而框架则是调用用户提供的类.

我在写框架（房子）时，你这个用户可能还在上小学，还不会写程序呢？我写的框架程序怎样能调用你以后写的类(门窗)呢？

因为在写程序无法知道要被调用的类名，所以在程序中无法new某个类的实例对象了，而要采用反射方式来做,这样就达到了动态加载用户自定义类的目的.

Java提供的放射机制允许你在运行时动态加载类、查看类信息、生成对象或操作生成的对象。

反射机制的应用实例：

在IDE(集成开发环境,如eclipse)中所提供的方法提示或是类查看工具。Jsp中的JavaBean自动收集请求信息也使用了反射。

Java在真正需要使用一个类时才会加以加载，而不是在程序启动时就加载所有的类。因为大多数的用户都只使用到应用程序的部分资源，在需要某些功能时才加载某些资源，可以让系统的资源运用更有效率（Java本来就是为了资源有限的小型设备而设计的,这样的考虑是必然的）。

enum(属于类的一种)、annotation(属于接口的一种)

Class对象由JVM自动产生，每当一个类被加载时，JVM就自动为其生成一个java.lang.Class对象。

可以通过Object的getClass()方法来取得每一个对象对应的Class对象，或者是通过class常量，如：

String str1 = "abc" ;

/\*\*

\* 如何得到各个字节码对应的实例对象(Class类型)?

\* \*/

Class cls1 = str1.getClass();

Class cls2 = String.**class**;

Class cls3 = Class.*forName*("java.lang.String");

System.*out*.println(cls1 == cls2); //true

System.*out*.println(cls1 == cls3); //true

Java在真正需要类时才会加载类，所谓真正需要通常指的是：

1：使用指定的类生成对象

2：用户指定要加载类时

（如：Class.forName()加载类、使用ClassLoader的loadClass加载类）

使用类名称来声明参考名称并不会导致类的加载。

静态区块：

默认在类第一次被加载时会运行静态区块（说默认的原因，是因为可以设置加载类时不运行静态区块，使用Class生成对象时才运行静态区块）

public class TestClass{

static{

System.out.println(“类被载入”);

}

}

Class的静态forName()方法有两个版本：

1：指定类名称的版本

2：可以指定类名称、加载类时是否运行静态区块、指定类加载器

static Class forName(String name,boolean initialize,ClassLoader loader)

第二个版本将initialize设置为false,这样加载类时并不会立即运行静态区块，而会在

使用类建立对象时才会运行静态区块。

第二个版本的forName()方法会需要一个类加载器，可以通过

Thread.currentThread().getContextClassLoader() 得到主线程的类加载器.

super.getClass()方法调用

下面程序的输出结果是多少？

import java.util.Date;

**public** **class** Test **extends** Date{

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

**new** Test().test();

}

**public** **void** test(){

System.*out*.println(**super**.getClass().getName());

}

}

很奇怪，结果是Test

在test方法中，直接调用getClass().getName()方法，返回的是Test类名

由于getClass()在Object类中定义成了final，子类不能覆盖该方法，所以，在

test方法中调用getClass().getName()方法，其实就是在调用从父类继承的getClass()方法，等效于调用super.getClass().getName()方法，所以，super.getClass().getName()方法返回的也应该是Test。

如果想得到父类的名称，应该用如下代码：

getClass().getSuperClass().getName();

#### 类加载器

什么是类加载器和类加载器的作用？

Java在需要使用类的时候，才会将来加载，Java类的加载是由类加载器来完成的。

Java虚拟机中可以安装多个类加载器，系统默认三个主要类加载器，每个类加载器负责加载特定位置的类：Bootstrap Loader , ExtClassLoader , AppClassLoader 。

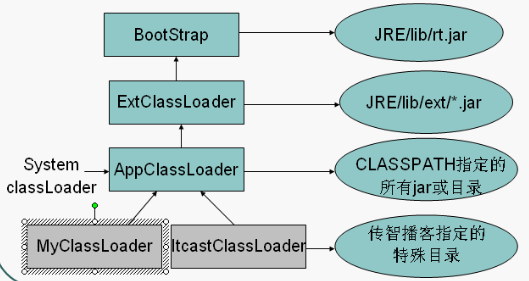
类加载器也是Java类，因为其它是java类的类加载器本身也要被类加载器加载，显然必须有第一个不是java类，这正是BootStrap Loader ， BootStrap Loader通常由C编写而成.

ExtClassLoader , AppClassLoader(也可称为System Loader) 都是java类，分别对应

sun.misc.Launcher$ExtClassLoader 与sun.misc.Launcher$AppClassLoader ，它们都是Launcher(发射装置)中的内部类。

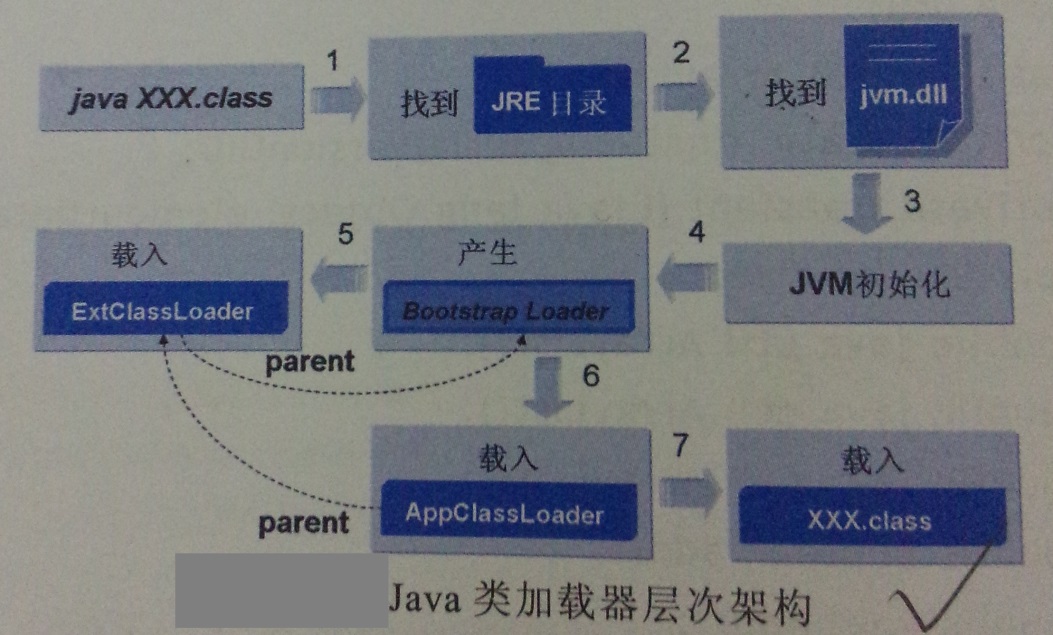
java虚拟机中的所有类加载器采用具有父子关系的树形结构进行组织，在实例化每个类加载器对象时，需要为其指定一个父级类加载器对象或者默认采用系统类加载器为其父级类加载。

类加载器之间的父子关系和管辖范围?



java程序启动与加载类的顺序图：

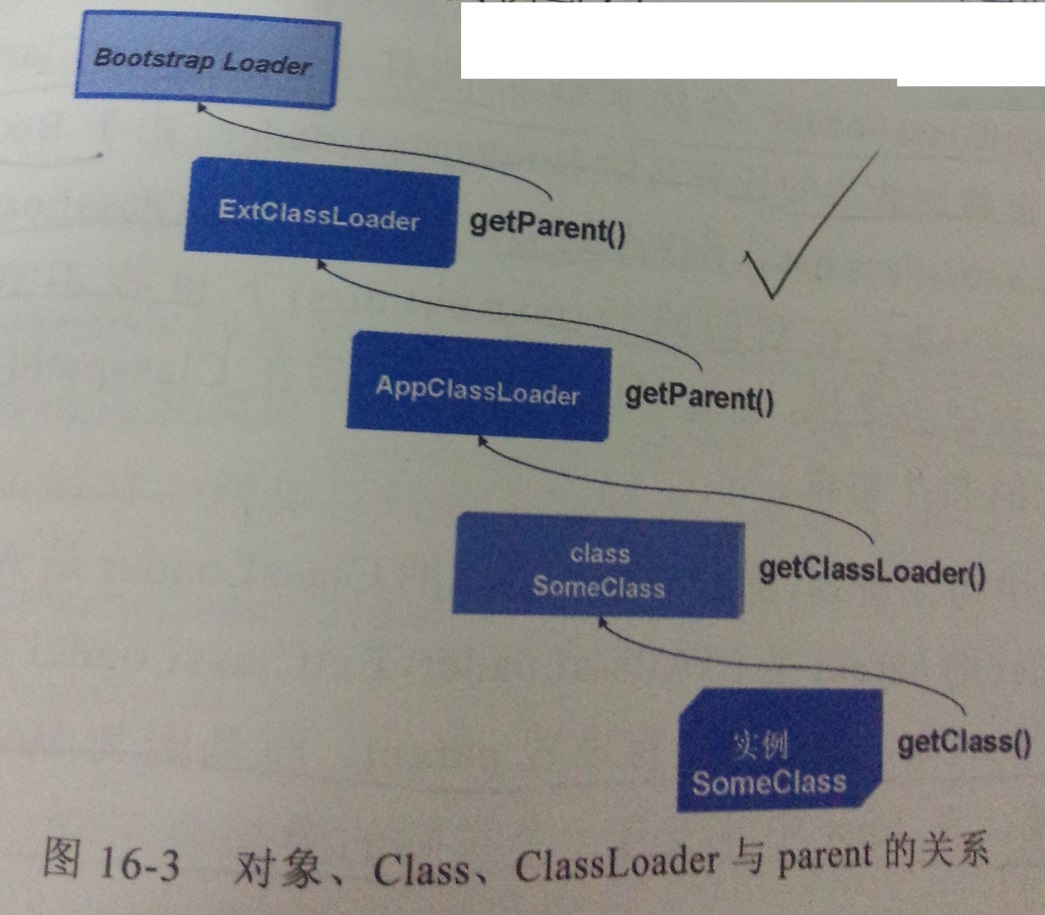
当在命令行模式下执行java XXX.class 指令后，java运行程序会尝试找到JRE安装的所在目录，然后寻找jvm.dll(默认是在JRE目录下bin\client目录中),接着启动JVM并进行初始化动作，产生Bootstrap Loader , Bootstrap Loader 会加载 Extended Loader ,并设置 Extended Loader 的 parent 为 Bootstrap Loader . Bootstrap Loader 会加载 System Loader ,并将System Loader的parent 设置为 Extended Loader.



对象、Class 、ClassLoader 与 Parent 的关系图：

类加载器在java中是以java.lang.ClassLoader类型存在，每一个类被加载后，都会有一个

Class的实例来代表，而每个Class的实例都会记得自己是由哪个ClassLoader 加载的。

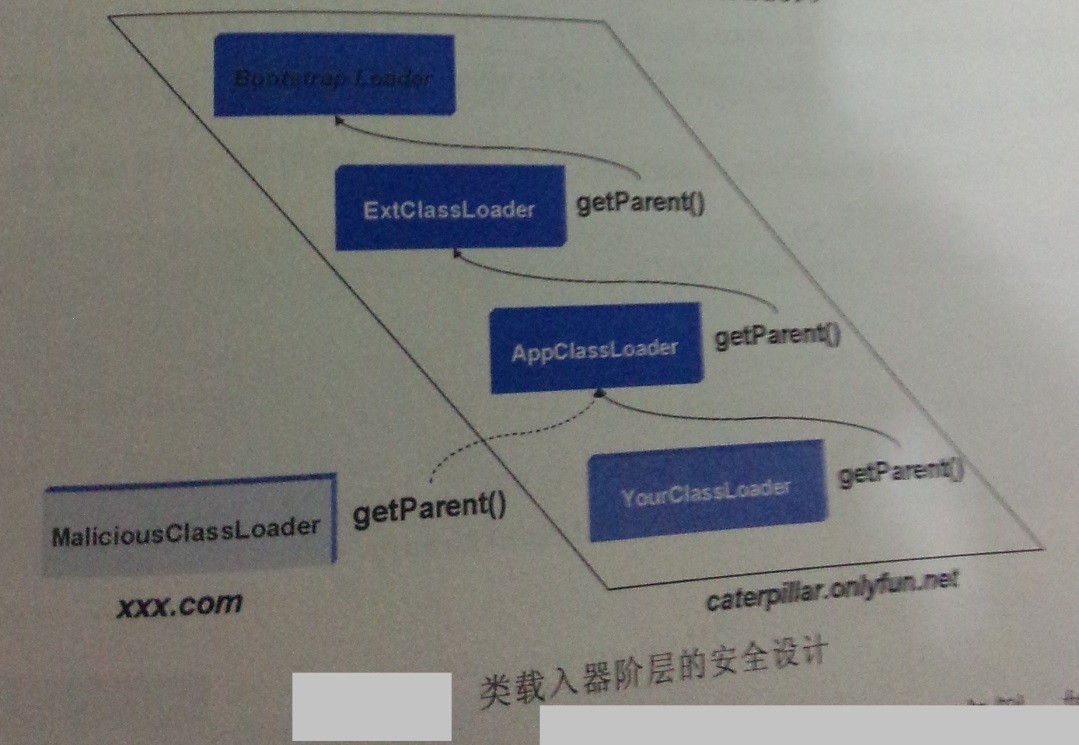


类载入器阶层的安全设计:

Java的类加载器层次架构除了可以达到动态加载类的目的之外，还有着安全上的考虑。因为每次寻找类时都是委托parent开始寻找，所以除非有人可以侵入你的计算机，置换掉标准的JavaSE API与您自己安装的延伸包，否则不能通过编写自己的类加载器来载入恶意类，以置换掉标准 Java SE API与您自己安装的延伸包。

由于每次的类载入是由子ClassLoader 委托父 ClassLoader 先尝试加载，但父ClassLoader看不到子 ClassLoader , 所以同一层次的子 ClassLoader 不会被误用 ，从而避免加载错误类的可能性，如下图，若从YourClassLoader 来加载类 ， 类加载器不会看到

MaliciousClassLoader .



由同一个ClassLoader加载的类文件，会只有一份Class实例(字节码) 。如果同一个类文件由两个不同的ClassLoader载入，则会有两份不同的字节码，注意，如果有两个不同的ClassLoader搜索同一个类，而在parent的AppClassLoader搜索路径中可以找到指定的类，则Class实例就只会有一个，因为两个不同的ClassLoader都是在委托父ClassLoader时找到该类的。如果父ClassLoader找不到，而是由各自ClassLoader搜索到，则Class的实例会有两份。

类加载器的委托机制?

* 当Java虚拟机要加载一个类时，到底派出哪个类加载器去加载呢？
* 首先当前线程的类加载器去加载线程中的第一个类。
* 如果类A中引用了类B，Java虚拟机将使用加载类A的类加载器来加载类B。
* 还可以直接调用ClassLoader.loadClass()方法来指定某个类加载器去加载某个类。
* 每个类加载器加载类时，又先委托给其上级类加载器
* 当所有祖宗类加载器没有加载到类，回到发起者类加载器，还加载不了，则抛

ClassNotFoundException,不是再去找发起者类加载器的儿子，因为没有getChild方法，

即使有，那有多个儿子，找哪一个呢？

* 对着类加载器的层次结构图和委托加载原理，解释先前ClassLoaderTest输出成jre/lib/ext目录下的itcast.jar包后，运行结果为ExtClassLoader的原因。

每个ClassLoader本身只能分别加载特定位置和目录中的类，但它们可以委托给其它的类加载器去加载类，这就是类加载器的委托模式。类加载器一级级委托到BootStrap类加载器，当BootStrap无法加载当前所要加载的类时，然后才一级级回退到子孙类装载器去进行真正的加载。当回退到最初的类加载器时，如果它自己也不能完成类的装载，那就应报告ClassNotFoundException异常。

有一道面试题，能不能自己写个类叫java.lang.System ？

能写，但是没有意义，因为类加载器采用了委托机制，这样可以保证爸爸们优先，也就是总是使用爸爸们能找到的类，这样总是使用java系统提供的System .

在tomcat的web应用程序中，都是由webapp自己的类加载器先自己加载WEB-INF/classess目录中的类，然后才委托上级的类加载器加载，如果我们在tomcat的web应用程序中写一个java.lang.String，这时候Servlet程序加载的就是我们自己写的java.lang.String，但是这么干就会出很多潜在的问题，原来所有用了java.lang.String类的都将出现问题。

编写自己的类加载器？

* 知识讲解
* 自定义的类加载器必须继承ClassLoader
* loadClass方法与findClass方法
* defineClass方法
* 编程步骤
* 编写一个对文件内容进行简单加密的程序
* 编写一个自己的类装载器，可实现对加密过的类进行装载和解密
* 编写一个程序调用类加载器加载类，在源程序中不能用该类名定义引用变量，因为编译无法识别这个类。程序中可以除了使用ClassLoader.load方法之外，还可以使用设置线程的上下文类加载器或者系统类加载器，然后再使用Class.forName().
* 试验步骤
* 对不带包名的class文件进行加密，加密结果存放到另一个目录，例如：

java MyClassLoader MyTest.class F:\itcast

* 运行被加载类的程序，结果能够正常加载，但打印出来的类加载器名称为AppClassLoader
* 用加密后的类文件替换CLASSPATH环境下的类文件，再执行上一步操作就出问题了，错误说明是AppClassLoader类装载器装载失败
* 删除CLASSPATH环境下的类文件，再执行上一步操作就没问题了。
* 原理分析

模板方法设计模式。

父类--->loadClass/findClass/defineClass--->得到.Class文件中的内容转换成字节码

子类1（自己干) 不能覆盖loadClas,需要覆盖findClass

子类2（自己干）

例如，应用程序可以创建一个网络类加载器，从服务器中下载类文件。示例代码如下所示：

ClassLoader loader = new NetworkClassLoader(host, port);

Object main = loader.loadClass("Main", true).newInstance(); . . .

网络类加载器子类必须定义方法 findClass 和 loadClassData，以实现从网络加载类。下载组成该类的字节后，它应该使用方法 defineClass 来创建类实例。示例实现如下：

class NetworkClassLoader extends ClassLoader {

String host;

int port;

public Class findClass(String name) {

byte[] b = loadClassData(name);

return defineClass(name, b, 0, b.length);

}

private byte[] loadClassData(String name) {

// load the class data from the connection

. . .

}

}

代码示例1：

/\*\*

测试当前类的类加载器

\*/

**public** **class** ClassLoaderTest {

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

System.*out*.println(

ClassLoaderTest.**class**.getClassLoader().getClass().getName()

); // AppClassLoader

/\*\*System类是由BootStrap类加载器加载的，BootStrap是由c/c++编写的，

所以调用getClassLoader()的返回值为null\*/

System.*out*.println(System.**class**.getClassLoader()); // null ；

ClassLoader loader = ClassLoaderTest.**class**.getClassLoader();

**while**(loader!=**null**){

System.*out*.println(loader.getClass().getName());

loader = loader.getParent();

}

System.*out*.println(loader); // null

}

}

代码示例2：

/\*\*

自定义类加载器试验

\*/

**package** cn.itcast.day2;

/\*\*

\* 作为被自定义加载器 加载的测试类

\* \*/

**public** **class** ClassLoaderAttachment **extends** Date{

@Override

**public** String toString() {

**return** "hello,itcast";

}

}

**package** cn.itcast.day2;

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

/\*\*

\* 运行时：

\* 右键 ---> Run As ---> Run Configurations... --->

\* 在 Arguments 的选项卡中 输入以下内容，然后运行.

\* E:\wks\_study\javaenhance\bin\cn\itcast\day2\ClassLoaderAttachment.class itcastlib

\*/

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

String srcPath = args[0];

String destDir = args[1];

FileInputStream fis = **new** FileInputStream(srcPath);

//windows下分隔符

String destFileName = srcPath.substring(srcPath.lastIndexOf("\\")+1);

String destPath = destDir+"\\"+destFileName;

FileOutputStream fos = **new** FileOutputStream(destPath);

*cypher*(fis,fos);

fis.close();

fos.close();

}

**private** **static** **void** cypher(InputStream ips, OutputStream ops)

**throws** Exception {

**int** b = -1;

**while** ((b = ips.read()) != -1) {

/\*\*

\* 异或运算^(只要两个条件同时为真或假，其结果都为假;其它情况为真)，

\* 会先将两个条件进行转换，转换成二进制数据后，再进行运算

\* \*/

ops.write(b ^ 0xff);

}

}

**private** String classDir;

@Override

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

System.*out*.println("name---"+name);

String classFileName = classDir + "\\" + name.substring(name.lastIndexOf(".")+1) + ".class"; //如果传入的类名带包名，则将包名去掉

System.*out*.println(classFileName);

**try** {

FileInputStream fis = **new** FileInputStream(classFileName);

ByteArrayOutputStream bos = **new** ByteArrayOutputStream();

*cypher*(fis,bos);

**byte**[] bytes = bos.toByteArray();

fis.close();

bos.close();

**return** ~~defineClass~~(bytes, 0, bytes.length);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** **super**.findClass(name);

}

**public** MyClassLoader() {

}

**public** MyClassLoader(String classDir) {

**this**.classDir = classDir;

}

}

**package** cn.itcast.day2;

**public** **class** ClassLoaderTest {

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

//有包名的类不能调用无包名的类

/\*\*

\* 在运行MyClassLoader的main方法后，

\* 会在itcastlib目录下生成ClassLoaderAttachment类加密后的.class文件

\* 将该文件替换E:\wks\_study\javaenhance\bin\cn\itcast\day2目录下的ClassLoaderAttachment.class

\* 这样上面的打印语句编译时就会报错，因为.class文件是加过密的，现在我们采用自定义的类加载器解决这个错误

\*

\*类加载时，会先去找父类加载器进行加载，如果父类加载器找不到，才由自身的类加载器进行加载

\* 当加载cn.itcast.day2包下的ClassLoaderAttachment.class时，

\* 如果cn.itcast.day2包下存在该类，父类加载器就会加载加密后的ClassLoaderAttachment.class，加载时就会出错，

\* 如果将cn.itcast.day2包下的ClassLoaderAttachment.class删除，父类加载器找不到该类，这样加载任务重新交给自定义的类加载器，这样就不会报错.

\* \*/

Class clazz = **new** MyClassLoader("itcastlib").loadClass("ClassLoaderAttachment");

//ClassLoaderAttachment d1 = (ClassLoaderAttachment)clazz.newInstance(); //因为加过密的类内容不合法，所以编译时就会报错

Date d1 = (Date)clazz.newInstance(); //因为加过密的类内容不合法，所以编译时就会报错，采用ClassLoaderAttachment类的父类Date可以规避编译问题

System.*out*.println(d1);

System.*out*.println(d1.getClass().getClassLoader().getClass().getName());

// //cn.itcast.day2.MyClassLoader

/\*\*

\*加载类时也可以指定包名 ， classpath路径下如果有cn.itcast.day2.ClassLoaderAttachment这个类，并且该类没有被加密，那么根据类加载委托机制，

\*所以下面代码最终会由AppClassLoader成功加载

\*\*/

Class clazz2 = **new** MyClassLoader("itcastlib").loadClass("cn.itcast.day2.ClassLoaderAttachment");

Date d2 = (Date)clazz2.newInstance();

System.*out*.println(d2); //hello,itcast

System.*out*.println(d2.getClass().getClassLoader().getClass().getName());

//sun.misc.Launcher$AppClassLoader

}

}