北京圣思园科技有限公 北京已口 mngsiyuan.com http：//www-shenS





深入Java虚拟机(Inside JVM)

主讲人张龙

**All Rights Reserved**

类加載器深入割析

**• Java**虚拟机与程序的生命周期

**-**在如下几种情况下，**Java**虚拟机将结束生 命周期

一执行了 **System.exit()**方法

-程序正常执行结束

-程序在执行过程中遇到了异常或错误而异常终 止

-由于操作系统出现错误而导致**Java**虚拟机进程 终止

**-**加载：查找并加载类的二进制数据

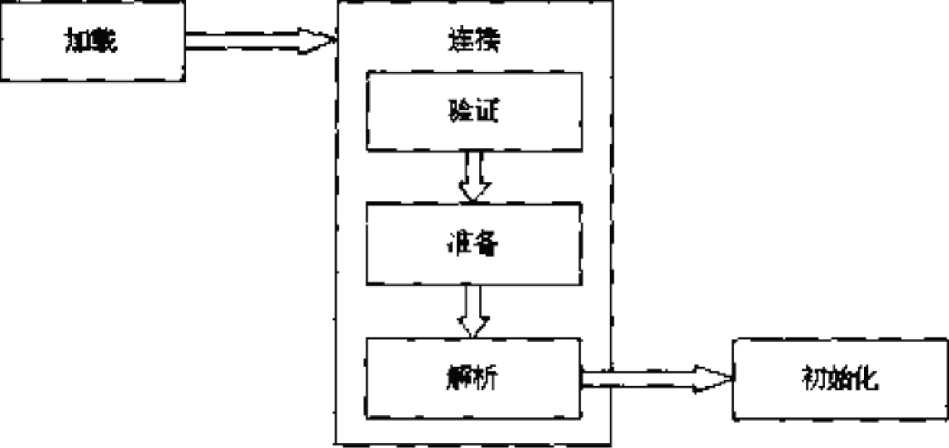
**•**连接

-验证：确保被加载的类的正确性

-准备：为类的**静态变量**分配内存，并将其初始 化为默认值

-解析：**把类中的符号引用转换为直接引用**

**•**初始化：为类的静态变量赋予正确的初始



**• Jav a**程序对类的使用方式可分为两种

-**主动使用**

-**被动使用**

**•**所有的**Java**虚拟机实现必须在每个类或接 口被**Java**程序"首次主动使用”时才初始 化他们

-**Java**虚拟机启动时被标明为启动

**Test**)

•主动使用(六种)

-创建类的实例

-访问某个类或接口的静态变量，或者对该静态 变量赋值

-调用类的静态方法

-反射(如

**Class**・**forName("com.shengsiyuan**・**Test**〃) )

-初始化一个类的子类

**-**除了以上六种情况，其他使用**Java**类的方 式都被看作是对类的被动使用，都不会导 致类的初始化

•类的加载指的是将类的・**class**文件中的二进 制数据读入到内存中，将其放在运行时数 据区的方法区内，然后在堆区创建一个 **java.Iang.Class**对象，用来封装类在方法 区内的数据结构

•加载**.class**文件的方式

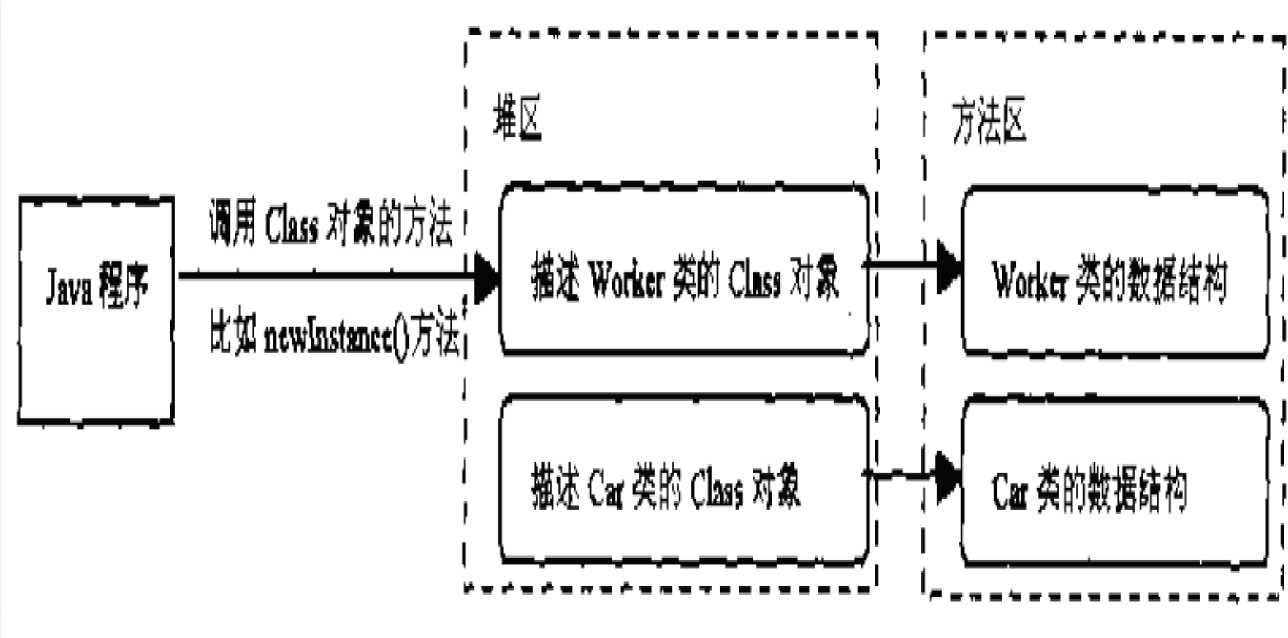
-从本地系统中直接加载

一通过网络下载**.class**文件

一**从zip, jar等归档文件中加载-class文件**

-从专有数据库中提取**.class**文件

一 将Java源文件动态编译为.class文件



-**类的加载的最终产品是位于堆区中的**

Class对象

• **Class**对象封装了类在方法区内的数据结构 ，并且向**Java**程序员提供了访问方法区内 的数据结构的接口

•有两种类型的类加载器

- **Java虚拟机自带的加载器**

•根类加载器(Bootstrap)

•扩展类加载器(Extension)

•系统类加载器(System)

-用户自定义的类加载器

• java.Iang.ClassLoader 的子类

•用户可以定制类的加载方式

**-**类加载器**并不需要**等到某个类被“首次主 动使用”时再加载它

• **JVM**规范允许类加载器在预料某个类将要 被使用时就预先加载它，如果在预先加载 的过程中遇到了 **.class**文件缺失或存在错误 ，类加载器必须在程序首次主动使用该类 时才报告错误（**LinkageError**错误）

•如果这个类一直没有被程序主动使用，那 么类加载器就不会报告错误

•类被加载后，就进入连接阶段。连接就是 将已经读入到内存的类的二进制数据合并 到虚拟机的运行时环境中去。

•类的验证的内容

-类文件的结构检查

-语义检查

-字节码验证

-二进制兼容性的验证

类的验证主要包括以卜内容。

* 类文件的结构检查：确保类文件遵从Java类文件的固定格式。
* 语义检査：确保类本身符合Java语言的语法规定，比如验证final类型的类没 有子类，以及final类型的方法没有被覆盖。
* 字节码验证:确保字节码流可以被Java虚拟机安全地执行。字节码流代表Java 方法（包括静态方法和实例方法），它是由被称做操作码的单字节指令组成的 序列，每一个操作码后都跟着一个或多个操作数。字节码验证步骤会检查每 个操作码是否合法，即是否有着合法的操作数。
* 二进制兼容的验证：确保相互引用的类之间协调一致。例如在Worker类的 goto Work。方法中会调用Car类的nm（）方法。Java虚拟机在验证Worker类时， 会检査在方法区内是否存在Car类的run。方法，假如不存在（当Worker类和 Car类的版本不兼容，就会出现这种问题），就会抛出NoSuchMethodError错误。

在准备阶段，Java虚拟机为类的静态变量分配内存，并设置默认的初始值。例如 对于以下Sample类，在准备阶段，将为ini类型的静态变量a分配4个字节的内存空 间，并且赋予默认值0，为long类型的静态变量b分配8个字节的内存空间，并且赋 予默认值0。

public class Sanple( private static int a=l; public static long b;

static(

b=：2 ；



IM

在解析阶段，Java虚拟机会把类的二进制数据中的符号引用替换为直接引用。例 如在Worker类的goto Work。方法中会引用Car类的run()方法。

public void gotoWork()(

//这段代码在**Worker**超二进(瞄鼻中表示为符轲用

**car.runO;**

在Worker类的二进制数据中,包含了一个对Car类的nm()方法的符号引用，它由run() 方法的全名和相关描述符组成。在解析阶段，Java虚拟机会把这个符号引用替换为一个指 针，该指针指向Car类的run()方法在方法区内的内存位置，这个指针就是直接引用。

在初始化阶段，Java虚拟机执行类的初始化语句，为类的静态变量賦予初始值。 在程序中，静态变量的初始化有两种途径：(1)在静态变量的声明处进行初始化；(2) 在静态代码块中进行初始化。例如在以下代码中，静态变量a和b都被显式初始化， 而静态变量c没有被显式初始化，它将保持默认值0。

public class Sanple（ private static int a=l; public static long b； public static long c；

static（ b=2； ）

静态变量的声明语句，以及静态代码块都被看做类的初始化语句，Java虚拟机会 按照初始化语句在类文件中的先后顺序来依次执行它们。例如当以下Sample类被初始 化后，它的静态变量a的取值为4。

public class Sanple(

static int a=l;

static{ a=2; }

staticf a=4; }

public static void main(String args[])(

System.out .println("a="+a)； 〃打印a=4

}

} .

•类的初始化步骤

（1） 假如这个类还没有被加载和连接，那就先进行加载和连接。

（2） 假如类存在直接的父类，并且这个父类还没有被初始化，那就先初始化直接 的父类。

（3） 假如类中存在初始化语句，那就依次执行这些初始化语句。

-**Java**虚拟机启动时被标明为启动

**Test**)

•主动使用(**六种**)

-创建类的实例

-访问某个类或接口的静态变量，或者对该静态 变量赋值

-调用类的静态方法

-反射(如

**Class**・**forName("com.shengsiyuan**・**Test**〃) )

-初始化一个类的子类

-**除了上述六种情形，其他使用Java类的方 式都被看作是被动使用,不会导致类的初 始化**

（3）当Java虚拟机初始化一个类时，要求它的所有父类都已经被初始化，但是 这条规则并不适用于接口。

* 在初始化一个类时，并不会先初始化它所实现的接口。
* 在初始化一个接口时，并不会先初始化它的父接口。

因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化。只有当 程序首次使用特定接口的静态变量时，才会导致该接口的初始化。

•只有当程序访问的静态变量或静态方法确 实在当前类或当前接口中定义时，才可以 认为是对类或接口的主动使用

•调用**ClassLoade「**类的**loadClass**方法加载 一个类，并不是对类的主动使用，不会导 致类的初始化。

类加载器用来把类加载到Java虚拟机中。从JDK 1.2版本开始，类的加载过程釆 用父亲委托机制，这种机制能更好地保证Java平台的安全。在此委托机制中，除了 Java 虚拟机自带的根类加载器以外，其余的类加载器都有且只有一个父加载器。当Java程 序请求加载器loader 1加载Sample类时,loader 1首先委托自己的父加载器去加载Sample 类，若父加载器能加载，则由父加载器完成加载任务，否则才由加载器loaderl本身加

Java虚拟机自带了以下几种加载器。

* 根(Bootstrap)类加载器：该加载器没有父加载器。它负责加载虚拟机的核心 类库，如 java.lang.\* 等。例如从例程 10-4(Sample.java)可以看出,java.lang.Object 就是由根类加载器加载的。根类加载器从系统属性sun.boot.class.path所指定的 目录中加载类库。根类加载器的实现依赖于底层操作系统，属于虚拟机的实现 的一部分，它并没有继承javaJang.ClassLoader类。
* 扩展(Extension)类加载器：它的父加载器为根类加载器。它从java.ext.dirs 系统属性所指定的目录中加载类库，或者从JDK的安装目录的jre\lib\ext子目 录(扩展目录)下加载类库，如果把用户创建的JAR文件放在这个目录下， 也会自动由扩展类加载器加载。扩展类加载器是纯Java类，是 java.lang.ClassLoader 类的子类。
* 系统(System)类加载器：也称为应用类加载器，它的父加载器为扩展类加 载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中加 载类，它是用户自定义的类加载器的默认父加载器。系统类加载器是纯Java 类，是 java.lang.ClassLoader 类的子类。

除了以上虚拟机自带的加载器以外，用户还可以定制 自己的类加载器(User-defined Class Loader)。Java 提供了 抽象类java.lang.ClassLoader,所有用户自定义的类加载器 应该继承ClassLoader类

F1 、

根类加蛇置

**,t '**

扩展类加裁器

*-\_ 丿*

系読类加載**88**

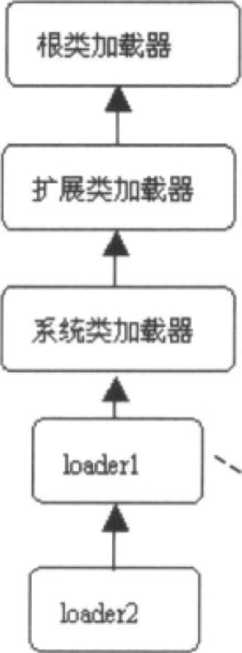


用户自定义类加載88

在父亲委托机制中，各个加载器按照父子关系形成了树形结构，除了根类加载器 以外，其余的类加载器都有且只有一个父加载器

加载

**Sample** 类



Class sanpleClass=loader2. loadClass ("Sanple");

loader?首先从自己的命名空间中查找Sample类是否己经被加载，如果已经加载, 就直接返回代表Sample类的Class对象的引用。

如果Sample类还没有被加载，loader?首先请求loader 1代为加载，loaderl再请求 系统类加载器代为加载，系统类加载器再请求扩展类加载器代为加载，扩展类加载器 再请求根类加载器代为加载。若根类加载器和扩展类加载器都不能加载，则系统类加 载器尝试加载，若能加载成功，则将Sample类所对应的Class对象的引用返回给 loaderl, loaderl再将引用返回给loader?,从而成功将Sample类加载进虚拟机。若系 统类加载器不能加载Sample类，则loaderl尝试加载Sample类，若loaderl也不能成 功加载，则loader2尝试加载。若所有的父加载器及loader2本身都不能加载，则抛出 ClassNotFoundException 异常。

**若右一个类加载器能成功加载**Sample**类，那 么这个类加载器被称为定义类加载器，所有能成功 返回**Class**对象的引用的类加载器（包括定义类加 载器）都被称为初始类加载器** 1

**假设**

loader 1 **实际加载了** Sample **类，则** loader 1 **为** Sample **类的定义类加载器，**loader2**和**loader 1**为**Sample**类 的初始类加载器。**

**需要指出的是，加载器之间的父子关系实际上指 的是加载器对象之间的包装关系，而不是类之间的继 承关系。一对父子加载器可能是同一个加载器类的两 个实例，也可能不是。在子加载器对象中包装了一个 父加载器对象。例如以下**loader 1**和**loader2**都是** MyClassLoade 1**■类的实例，并且** loader?**包装了** loader 1, loader 1**是**loader2**的父加载器。**

ClassLoader loader 1 = new btyClassLoader ()；

//W loaderl 林为 loader2

ClassLoader loader2 = new MyClassIxoader (loaderl)；

父亲委托机制的优点是能够提高软件系统的安全性。因为在此机制下，用户自定义 的类加载器不可能加载应该由父加载器加载的可靠类，从而防止不可靠甚至恶意的代码 代替由父加载器加载的可靠代码。例如，java.lang.Object类总是由根类加载器加载，其 他任何用户自定义的类加载器都不可能加载含有恶意代码的java.lang.Object类

命名空间

每个类加载器都有自己的命名空间，命名空间由该加载器及所有父加载器所加载 的类组成。在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两 个类；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两 个类’



运行时包

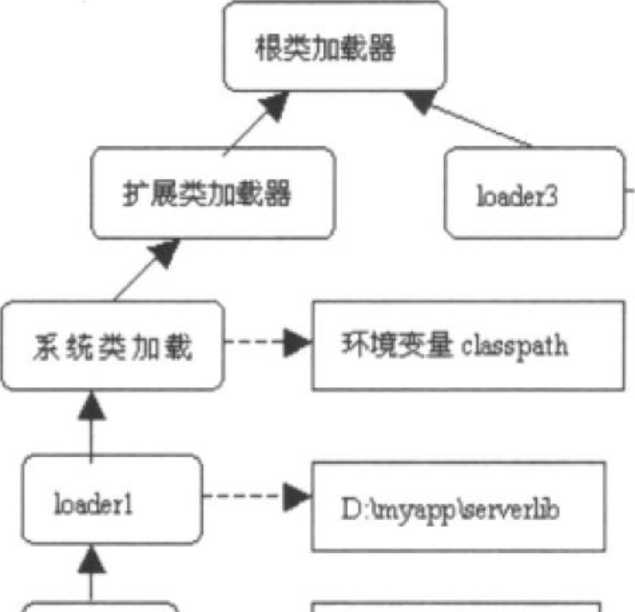
由同一类加载器加载的属于相同包的类组成了运行时包。决定两个类是不是属于 同一个运行时包，不仅要看它们的包名是否相同，还要看定义类加载器是否相同。只 有属于同一运行时包的类才能互相访问包可见（即默认访问级别）的类和类成员。这 样的限制能避免用户自定义的类冒充核心类库的类，去访问核心类库的包可见成员。 假设用户自己定义了一个类java.lang.Spy,并由用户自定义的类加载器加载，由于 java.lang.Spy和核心类库java.lang.\*由不同的加载器加载，它们属于不同的运行时包， 所以java.lang.Spy不能访问核心类库java.lang包中的包可见成员。

要创建用户自己的类加载器，只需要扩展java.lang.ClassLoader类，然后覆盖它的 findClass(String name)方法即可，该方法根据参数指定的类的名字，返回对应的Class 对象的引用。

A DAmyappVchentlib

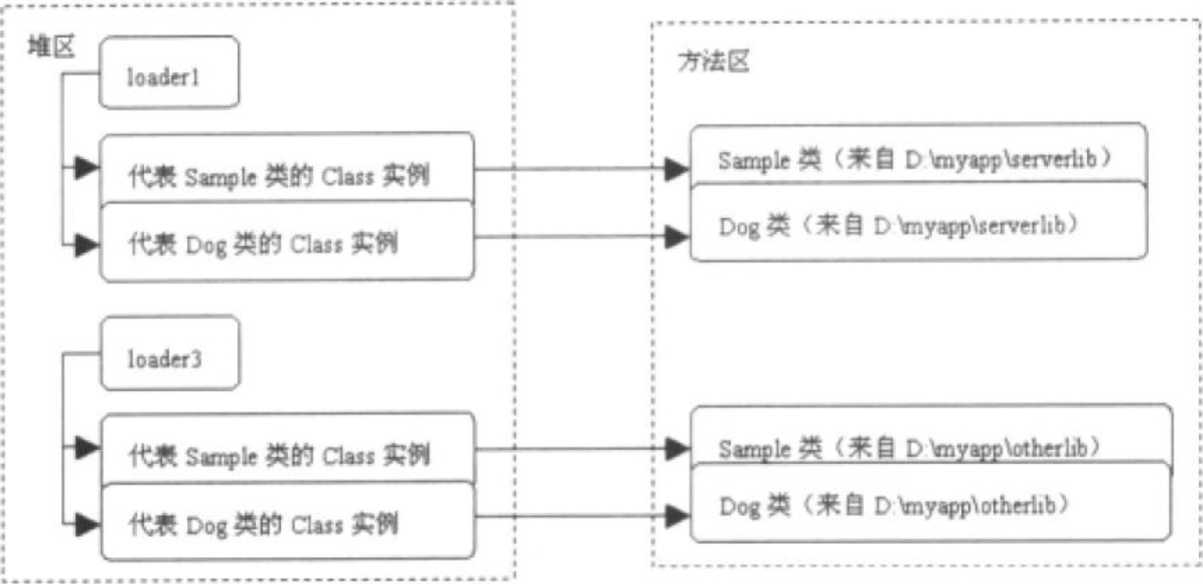
loads r2

D hnyappk)therli)



当执行loader?.loadClass("SampleH)时，先由它上层的所有父加载器尝试加载 Sample 类。loader 1 从 D:\myapp\serverlib 目录下成功地加载 了 Sample 类，因此 loader 1 是Sample类的定义类加载器，loader 1和loader2是Sample类的初始类加载器。

当执行loader3.1oadClassC'Sample")时，先由它上层的所有父加载器尝试加载 Sample类。loader3的父加载器为根类加载器，它无法加载Sample类，接着loader3从 D:\my叩p'olherlib目录卜成功地加载了 Sample类，因此loader3是Sample类的定义类 加载器及初始类加载器。



在loader 1和loader3各自的命名空间中都存在Sample类和Dog类

(2)在Sample类中主动使用了 Dog类，当执行Sample类的构造方法中的new Dog() 语句时，Java虚拟机需要先加载Dog类，到底用哪个类加载器加载呢？从步骤(1)的 打印结果可以看出，加载Sample类的loader 1还加载了 Dog类,Java虚拟机会用Sample 类的定义类加载器去加载Dog类，加载过程也同样采用父亲委托机制。为了验证这一 点，可以把D:\myapp\serverlib目录下的Dog.class文件删除，然后在D:\myapp\syslib 目录下存放一个Dog.class文件，此时程序的打印结果为：

Sanple is loaded by loaderl

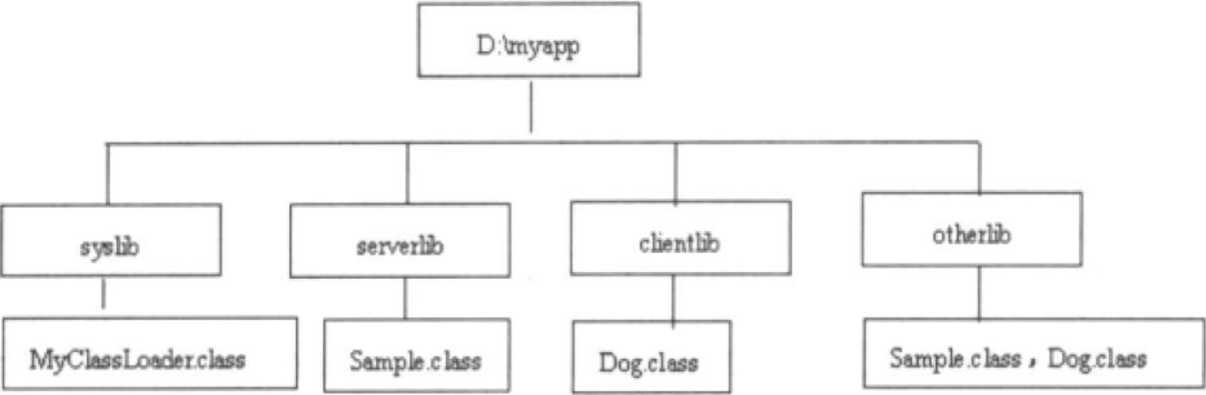
Dog is loaded toy sun .misc. Launcher$^pClassLoadereid6096

Sanple is loaded by loader3

Dog is loaded by loader3

由此可见，当由loader 1加载的Sample类首次主动使用Dog类时，Dog类由系统 类加载器加载。如果把D:\myapp\serverlib和D:\myapp\syslib目录下的Dog.class文件 都删除，然后在D:\myapp\clientlib目录下存放一个Dog.class文件，此时的目录结构参 见图10-8。当由loader 1加载的Sample类首次主动使用Dog类时，由于loader 1及它 的父加载器都无法加载Dog类，因此test(loader?)方法会抛出ClassNotFoundExceptionc

**创建用户自定义的类加载器**



同一个命名空间内的类是相互可见的。

子加载器的命名空间包含所有父加载器的命名空间。因此由子加载器加载的 类能看见父加载器加载的类。例如系统类加载器加载的类能看见根类加载器 加载的类。

由父加载器加载的类不能看见子加载器加载的类。

如果两个加载器之间没有直接或间接的父子关系，那么它们各自加载的类相 互不可见。



•**修改MyClassLoader类的源代码**

下面把 Sample.class 和 Dog.class 仅仅拷贝到 D:\myapp\serverlib 目录下

MyClassLoader类由系统类加载器加载，而Sample类由loader 1类加载，因此 MyClassLoader 类看不见 Sample 类。在 MyClassLoader 类的 main。方法中使用 Sample 类，会导致 NoClassDefFoundError 错误。

当两个不同命名空间内的类相互不可见时，可釆用Java反射机制来访问对方实例 的属性和方法。如果把MyClassLoader类的main。方法替换为如下代码：

如果把D:\myapp\serverlib目录下的Sample.class和Dog.class删除，再把这两个文 件拷贝到D:\myapp\syslib目录下，然后运行例程10-8中的main()方法，也能正常运行。 此时MyClassLoader类和Sample类都由系统类加载器加载，由于它们位于同一个命名 空间内，因此相互可见。

当Sample类被加载、连接和初始化后，它的生命周期就开始了。当代表Sample 类的Class对象不再被引用，即不可触及时，Class对象就会结束生命周期，Sample类 在方法区内的数据也会被卸载，从而结束Sample类的生命周期。由此可见，一个类何 时结束生命周期，取决于代表它的Class对象何时结束生命周期

由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被 卸载。前面已经介绍过，Java虚拟机自带的类加载器包括根类加载器、扩展类加载器 和系统类加载器。Java虚拟机本身会始终引用这些类加载器，而这些类加载器则会始 终引用它们所加载的类的Class对象，因此这些Class对象始终是可触及的。

**由用户自定义的类加载器所加载的类是可以被卸载的**

把 Sample.class 和 Dog.class 拷贝至U D:\myapp\serveHib 目录下，然后把 MyClassLoader类的main()方法替换为如下代码：

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

MyClassLoader loaderl = new MyClassLoader (■ loaderl"); 〃① loader 1. set Path ("D: \\nr/app\\serverlib\\ •); //②

Class objClass = loaderl.loadclass(•Sample")； //③ System.out.printIn("objClass's hashcode is ,+objClass.hashCode()); //④ Object obj=objClass.newlnstance() ； //(§)

loaderl ^null ； //(§) objClasss：null; 〃⑦ obj=null; 〃⑧

loaderl = new MyClassLoader (" loaderl ,); 〃⑨ objClass = loaderl. loadclass ("Sanple'); 〃⑩ System.out.printIn("objClass'a hashcode is •+objClass.hashCode(>);

运行以上程序时，Sample类由loader 1加载。在类加载器的内部实现中，用一个 Java集合来存放所加载类的引用。另一方面，一个Class对象总是会引用它的类加载器, 调用Class对象的getClassLoaderO方法，就能获得它的类加载器。由此可见，代表Sample 类的Class实例与loader 1之间为双向关联关系。

一个类的实例总是引用代表这个类的Class对象。在Objecl类中定义了 getClass() 方法，这个方法返回代表对象所属类的Class对象的引用。此外，所有的Java类都有 —静态属性class»它引用代表这个类的Class对象，例如：

**当程序执行完第⑤步时，引用变量与对象之间的引用关系如图**

:! !方法区

I \_\_ • •

**1 loScrll** 用賣■ **\*—► MyCM**•侦**Scr** 对譲

**objClaw**引間支■ 代表**s**鏡的**＞ + Simple**案的二話■散**1**■皓榆

**obj**引用变■ **: • Sample**药象

从图10-9可以看出,loaderl变量和obj变量间接引用代表Sample类的Class对象, 而objClass变量则直接引用它。

当程序执行完第⑧步时，所有的引用变量都置为null,此时Sample对象结束生命 周期，MyClassLoader对象结束生命周期，代表Sample类的Class对象也结束生命周 期，Sample类在方法区内的二进制数据被卸载。

当程序执行完第⑩步时,Sample类又重新被加载，在Java虚拟机的堆区会生成一 个新的代表Sample类的Class实例。

以上程序的打印结果如下：

objClass \* s hashCode is **7434986**

Sanple is loaded by loaderl

Dog is loaded by loaderl objClass,8 hashcode is **4923951**

从以上打印结果可以看出，程序两次打印objClass变量引用的Class对象的哈希码, 得到的数值不同，因此objClass变量两次引用不同的Class对象，可见在Java虚拟机 的生命周期中，对Sample类先后加载了两次。