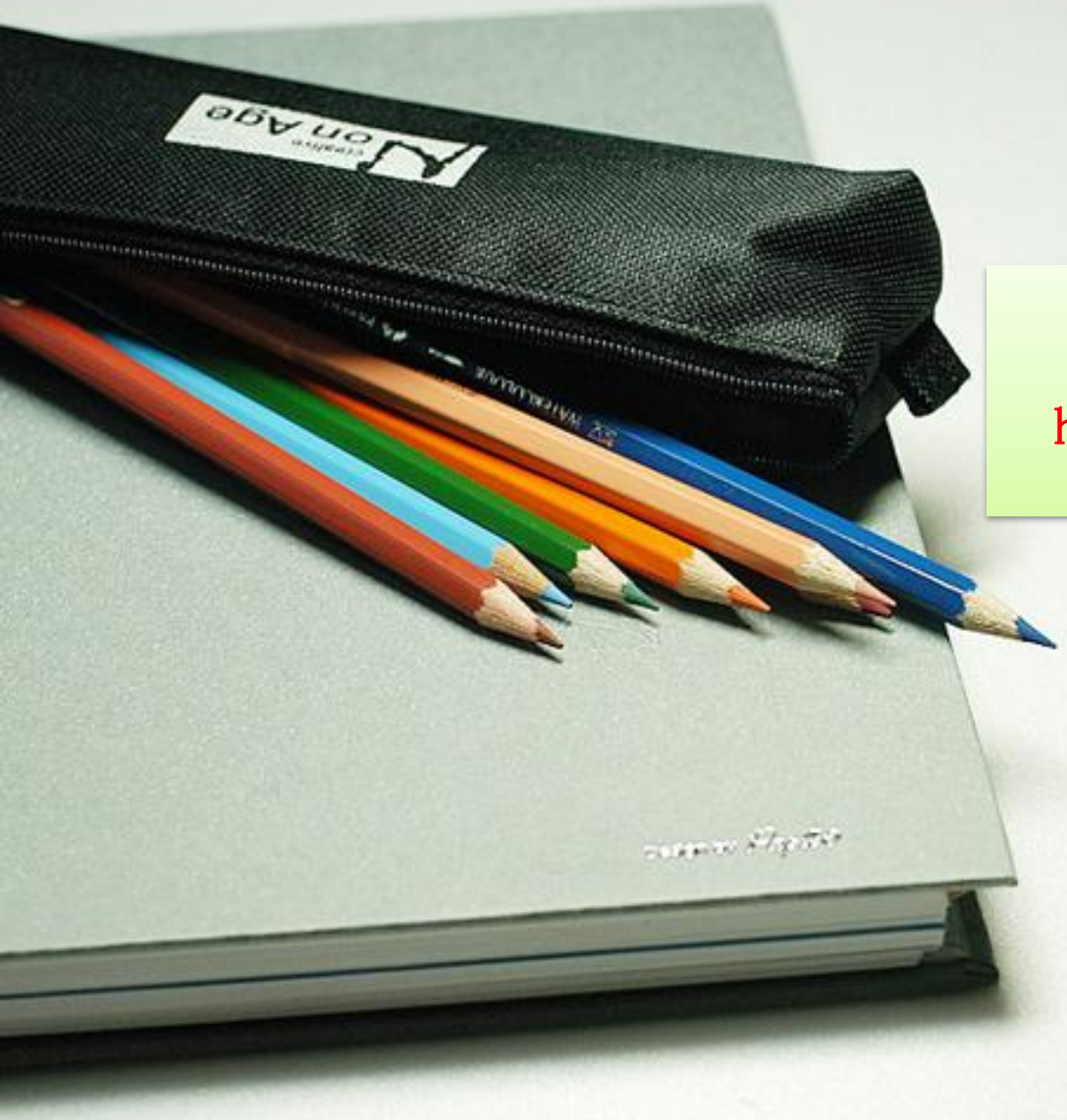


北京圣思园科技有限公司
http://www.shengsiyuan.com

主讲人：张龙

深入Java虚拟机（Inside JVM）

主讲人　张龙

All Rights Reserved

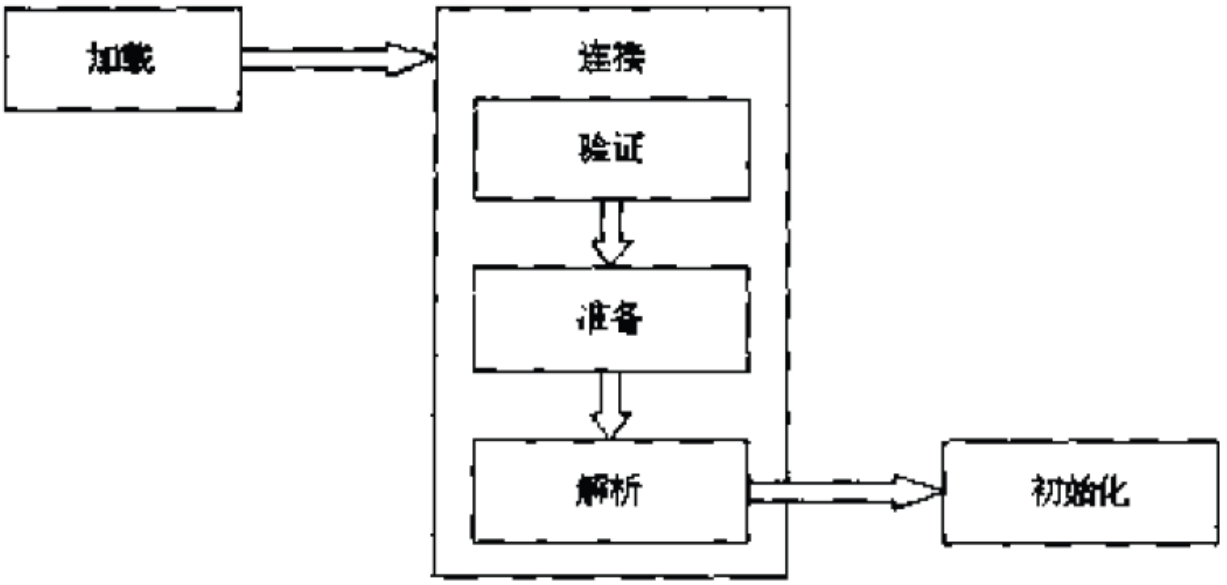
类加载器深入剖析

- 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
  - 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

类的加载、连接与初始化

- 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值**

类的加载、连接与初始化



类的加载、连接与初始化

- Java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

类的加载、连接与初始化

- 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
  - Java虚拟机启动时被标明为启动类的类（Java Test）

类的加载、连接与初始化

- 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

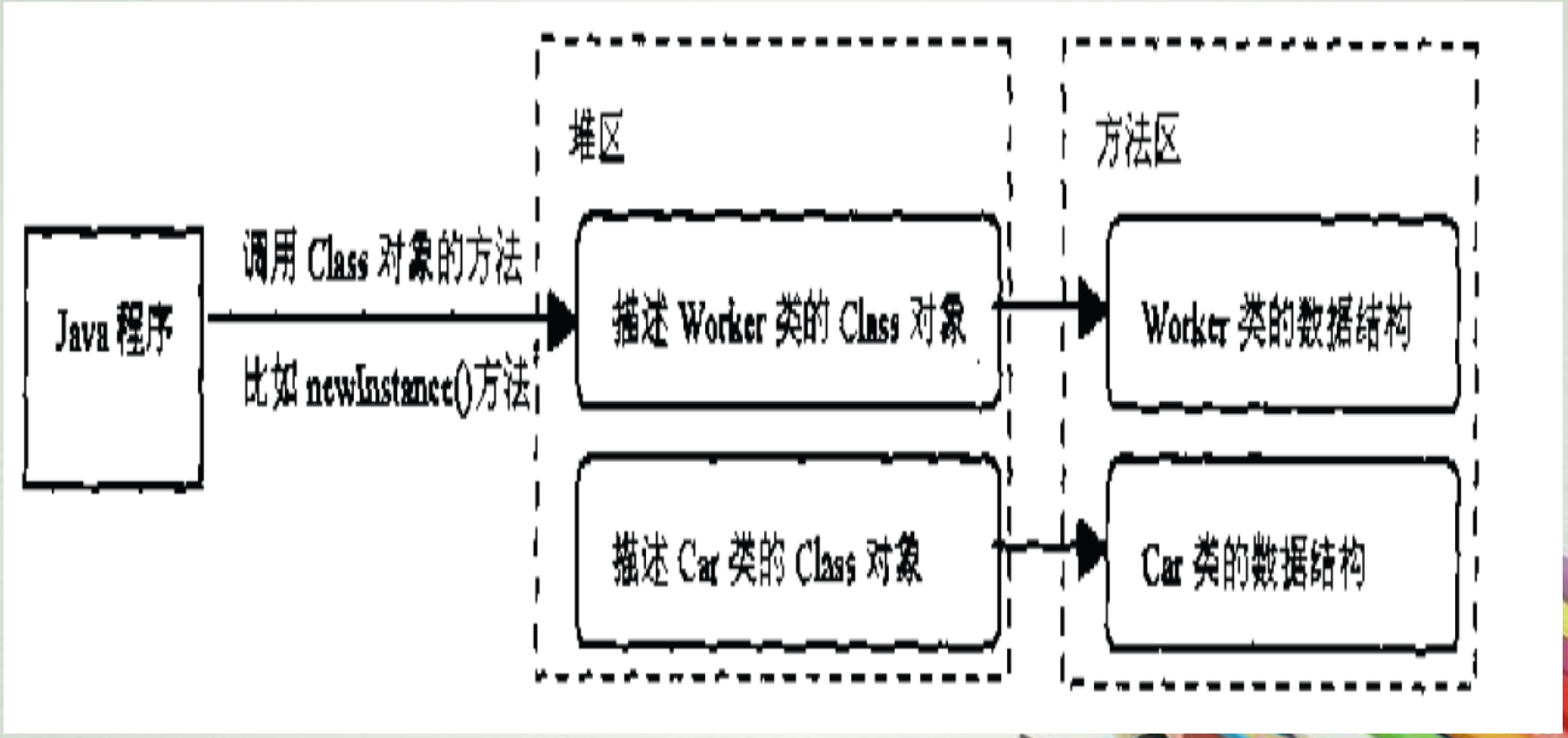
类的加载

- 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.lang.Class对象，用来封装类在方法区内的数据结构

类的加载

- 加载.class文件的方式
  - 从本地系统中直接加载
  - 通过网络下载.class文件
  - **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
  - 将Java源文件动态编译为.class文件

类的加载


堆区
方法区
Java 程序
调用 Class 对象的方法
比如 newInstance()方法
描述 Worker 类的 Class 对象
描述 Car 类的 Class 对象
Worker 类的数据结构
Car 类的数据结构

类的加载

- **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

类的加载

- 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.lang.ClassLoader的子类**
    - 用户可以定制类的加载方式

类的加载

- 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

类的加载

- 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

类的验证

- 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

- 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

类的验证主要包括以下内容。

- 类文件的结构检查：确保类文件遵从 Java 类文件的固定格式。
- 语义检查：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 gotoWork()方法中会调用 Car 类的 run()方法。Java 虚拟机在验证 Worker 类时，会检查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。

类的准备

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 int 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 Sample{
    private static int a=1;
    public static long b;

    static{
        b=2;
    }
    ...
}
```

类的解析

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 gotoWork()方法中会引用 Car 类的 run()方法。

```
public void gotoWork(){
    car.run();  //这段代码在Worker类的二进制数据中表示为符号引用
}
```

在 Worker 类的二进制数据中，包含了一个对 Car 类的 run()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。

类的初始化

在初始化阶段，Java 虚拟机执行类的初始化语句，为类的静态变量赋予初始值。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 Sample{
    private static int a=1;          //在静态变量的声明处进行初始化
    public static long b;
    public static long c;

    static{
        b=2;                         //在静态代码块中进行初始化
    }
    ...
}
```

类的初始化

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 Sample{
  static int a=1;
  static{ a=2; }
  static{ a=4; }
  public static void main(String args[]){
    System.out.println("a="+a);  //打印a=4
  }
}
```

类的初始化

- 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。

类的初始化时机

- 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
  - Java虚拟机启动时被标明为启动类的类（Java Test）

类的初始化时机

- 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

类的初始化时机

（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。

类的初始化时机

- 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

类的初始化时机

- 调用ClassLoader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

类加载器用来把类加载到 Java 虚拟机中。从 JDK 1.2 版本开始，类的加载过程采用父亲委托机制，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 loader1 加载 Sample 类时，loader1 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 loader1 本身加载 Sample 类。

类加载器

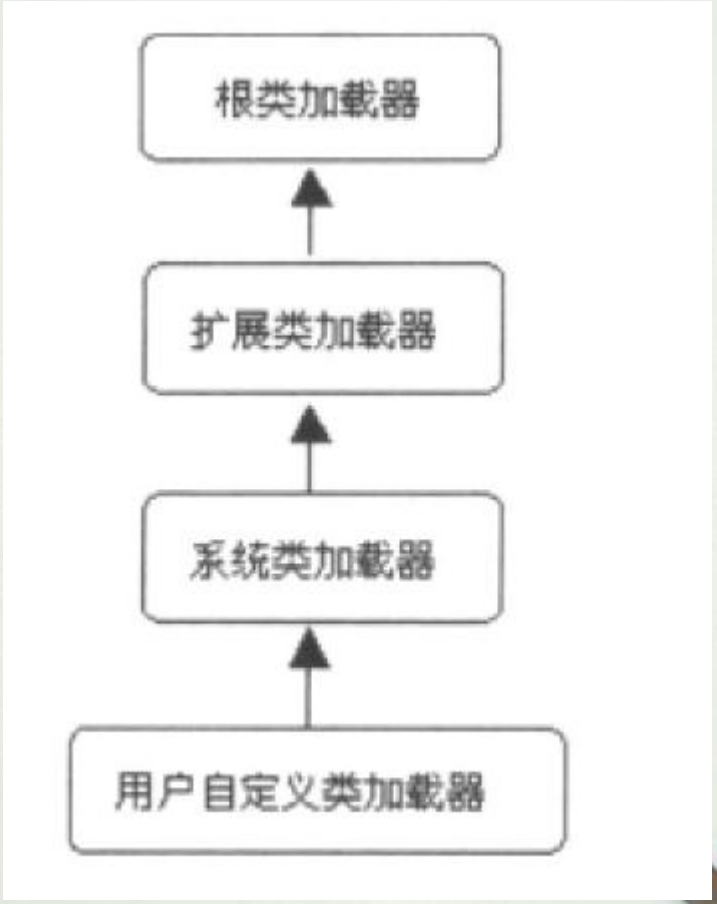
Java 虚拟机自带了以下几种加载器。

- 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 java.lang.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。

类加载器

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

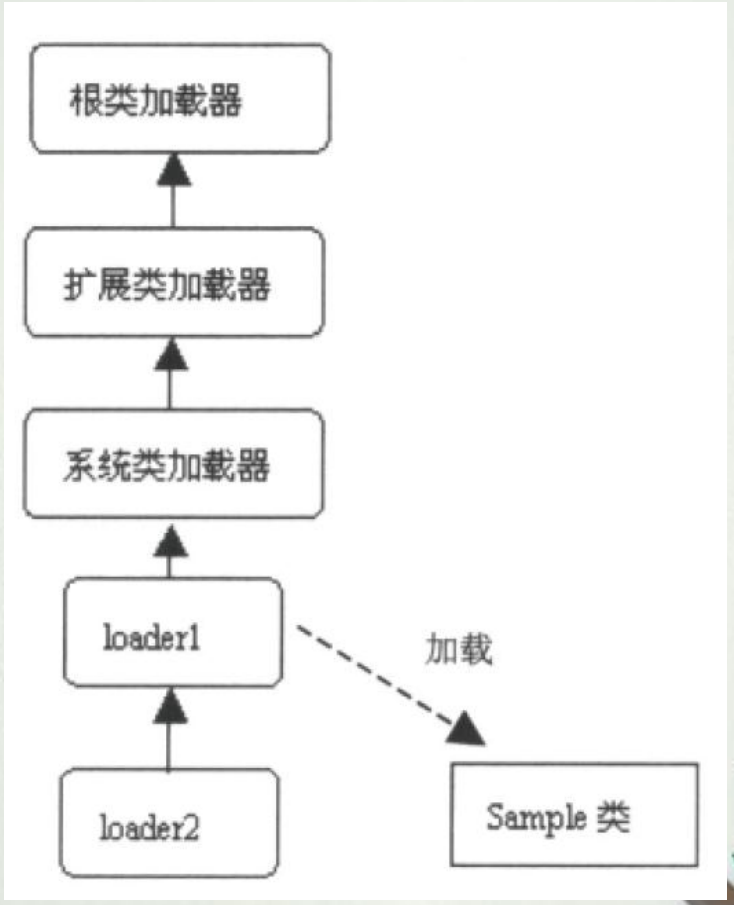
类加载器



类加载的父委托机制

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

类加载的父委托机制



类加载的父委托机制

```
Class sampleClass=loader2.loadClass("Sample");
```

loader2 首先从自己的命名空间中查找 Sample 类是否已经被加载，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，loader2 首先请求 loader1 代为加载，loader1 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 loader1，loader1 再将引用返回给 loader2，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 loader1 尝试加载 Sample 类，若 loader1 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。

类加载的父委托机制

若有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

类加载的父委托机制

假设

loader1 实际加载了 Sample 类，则 loader1 为 Sample 类的定义类加载器，loader2 和 loader1 为 Sample 类的初始类加载器。

类加载的父委托机制

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 loader1 和 loader2 都是MyClassLoader类的实例，并且loader2包装了loader1，loader1 是 loader2 的父加载器。

类加载的父委托机制

```
ClassLoader loader1 = new MyClassLoader();

//参数loader1将作为loader2的父加载器
ClassLoader loader2 = new MyClassLoader (loader1);
```

类加载的父委托机制

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。

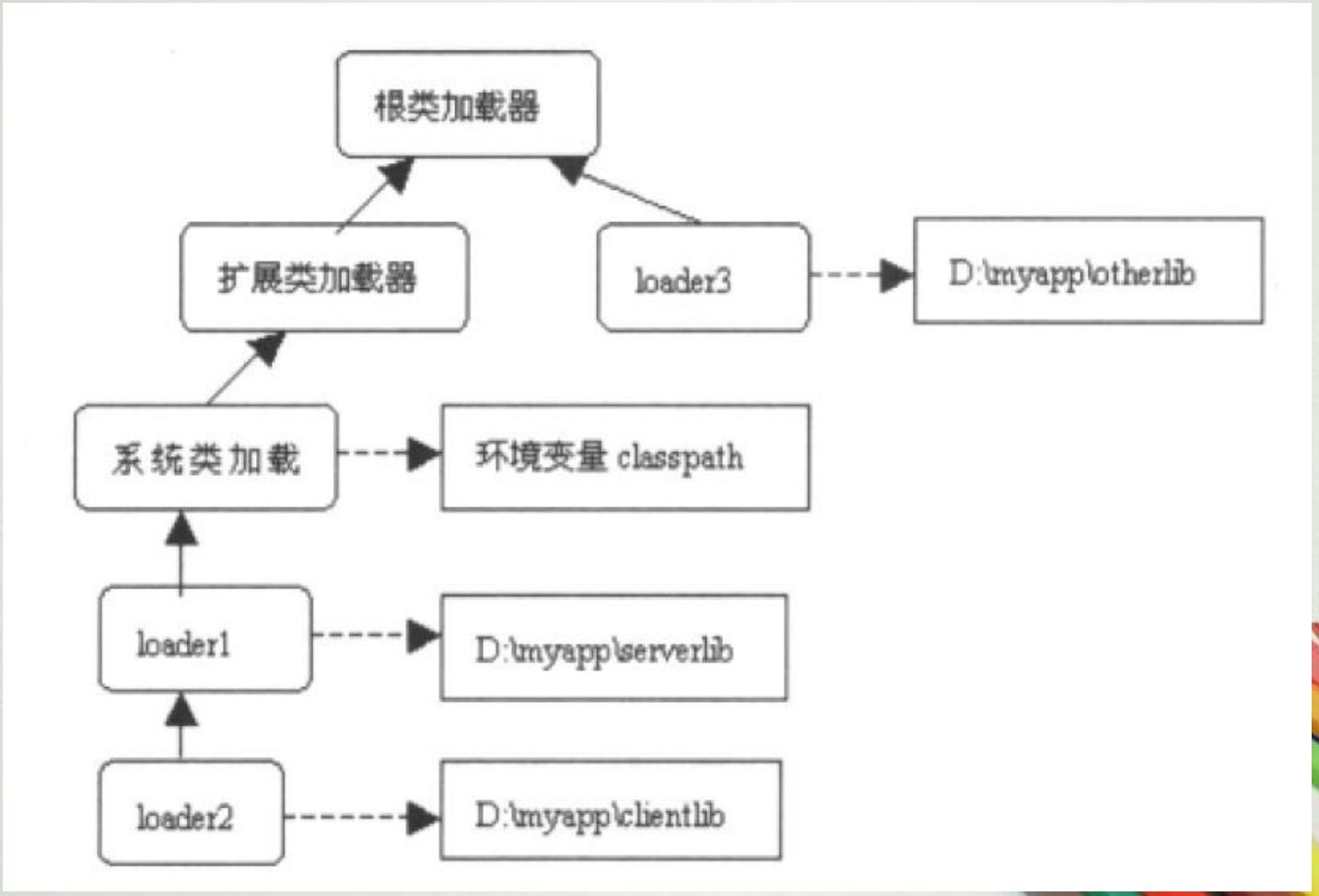
运行时包

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。

创建用户自定义的类加载器

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。

创建用户自定义的类加载器

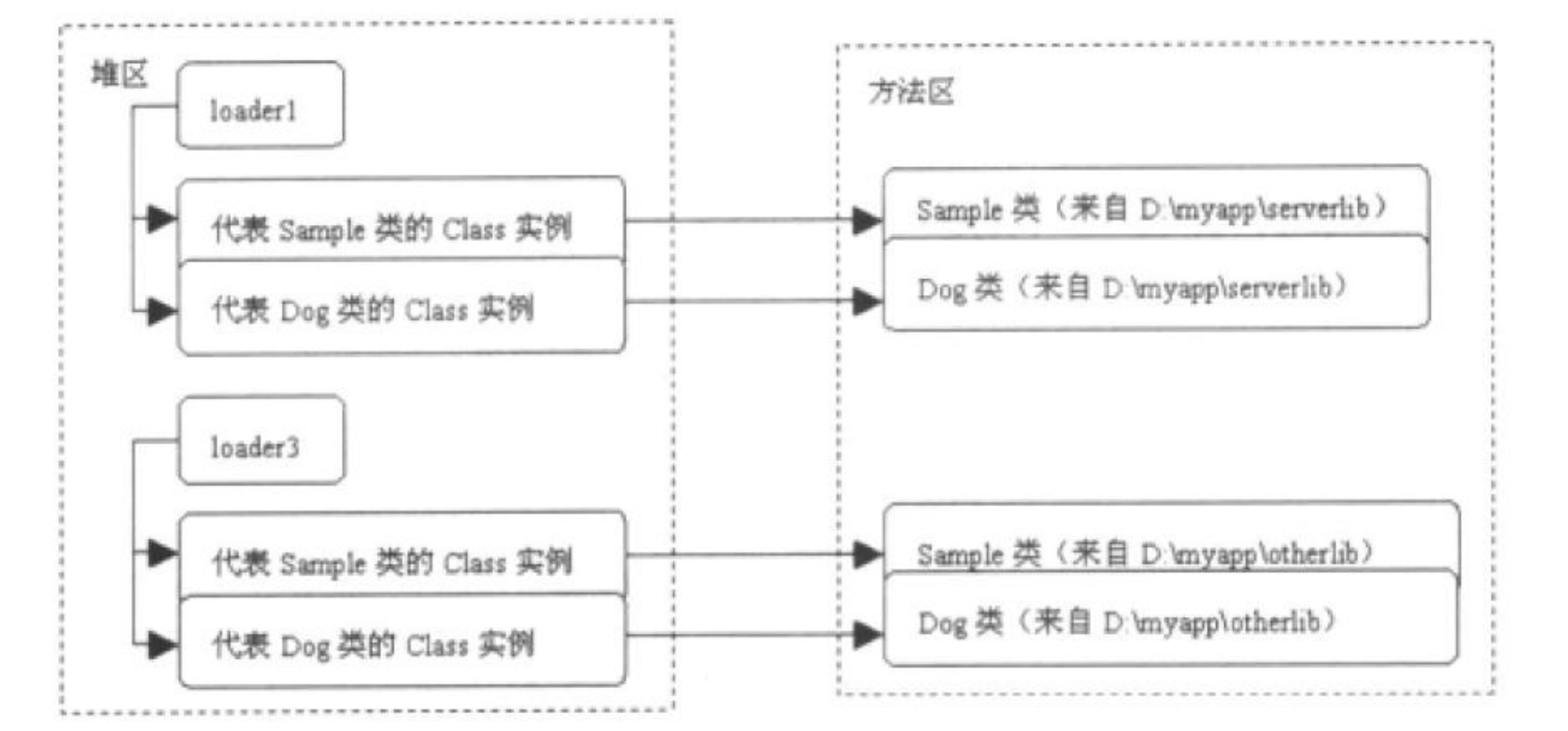


创建用户自定义的类加载器

当执行 loader2.loadClass("Sample")时，先由它上层的所有父加载器尝试加载 Sample 类。loader1 从 D:\myapp\serverlib 目录下成功地加载了 Sample 类，因此 loader1 是 Sample 类的定义类加载器，loader1 和 loader2 是 Sample 类的初始类加载器。

当执行 loader3.loadClass("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:\myapp\otherlib 目录下成功地加载了 Sample 类，因此 loader3 是 Sample 类的定义类加载器及初始类加载器。

创建用户自定义的类加载器



在 loader1 和 loader3 各自的命名空间中都存在 Sample 类和 Dog 类

创建用户自定义的类加载器

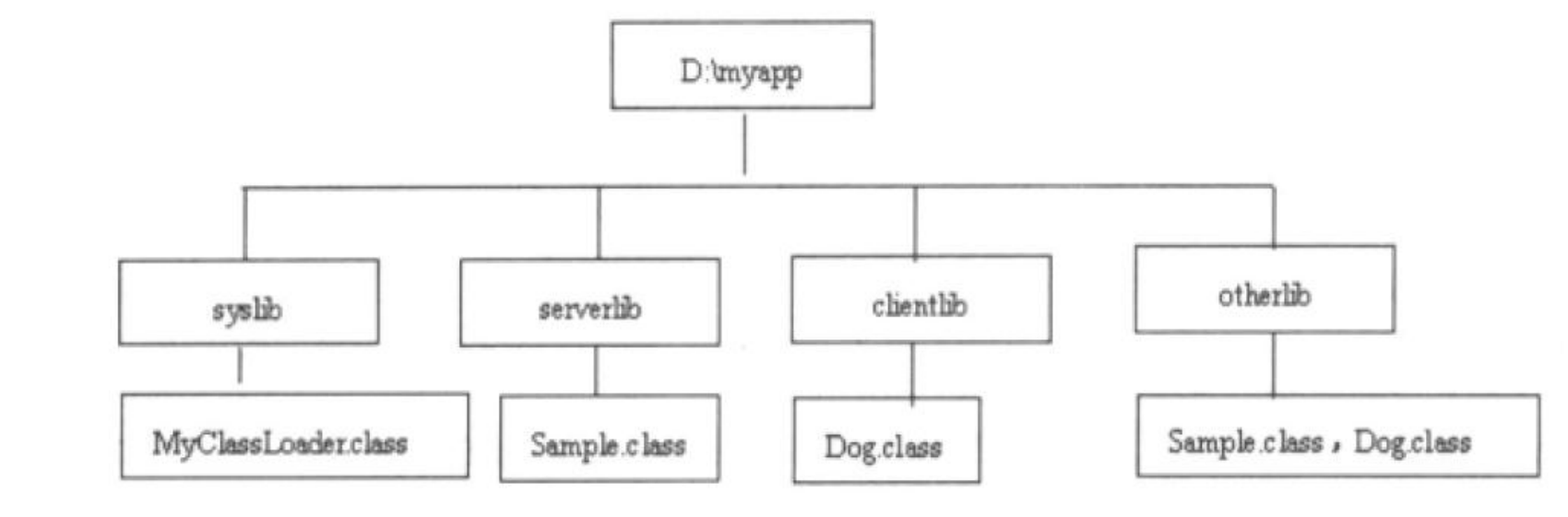
（2）在 Sample 类中主动使用了 Dog 类，当执行 Sample 类的构造方法中的 new Dog() 语句时，Java 虚拟机需要先加载 Dog 类，到底用哪个类加载器加载呢？从步骤（1）的打印结果可以看出，加载 Sample 类的 loader1 还加载了 Dog 类，Java 虚拟机会用 Sample 类的定义类加载器去加载 Dog 类，加载过程也同样采用父亲委托机制。为了验证这一点，可以把 D:\myapp\serverlib 目录下的 Dog.class 文件删除，然后在 D:\myapp\syslib 目录下存放一个 Dog.class 文件，此时程序的打印结果为：

```
Sample is loaded by loader1
Dog is loaded by sun.misc.Launcher$AppClassLoader@1d6096
Sample is loaded by loader3
Dog is loaded by loader3
```

创建用户自定义的类加载器

由此可见，当由 loader1 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 loader1 加载的 Sample 类首次主动使用 Dog 类时，由于 loader1 及它的父加载器都无法加载 Dog 类，因此 test(loader2)方法会抛出 ClassNotFoundException。

创建用户自定义的类加载器



不同类加载器的命名空间关系

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。

不同类加载器的命名空间关系

- 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

不同类加载器的命名空间关系

MyClassLoader 类由系统类加载器加载，而 Sample 类由 loader1 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，可采用 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：

不同类加载器的命名空间关系

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。

类的卸载

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

类的卸载

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

类的卸载

把 Sample.class 和 Dog.class 拷贝到 D:\myapp\serverlib 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 loader1 = new MyClassLoader("loader1");                    //①
  loader1.setPath("D:\\myapp\\serverlib\\");                               //②

  Class objClass = loader1.loadClass("Sample");                            //③
  System.out.println("objClass's hashCode is "+objClass.hashCode());       //④
  Object obj=objClass.newInstance();                                       //⑤

  loader1=null;                                                            //⑥
  objClass=null;                                                           //⑦
  obj=null;                                                                //⑧

  loader1 = new MyClassLoader("loader1");                                  //⑨
  objClass = loader1.loadClass("Sample");                                  //⑩
  System.out.println("objClass's hashCode is "+objClass.hashCode());
}
```

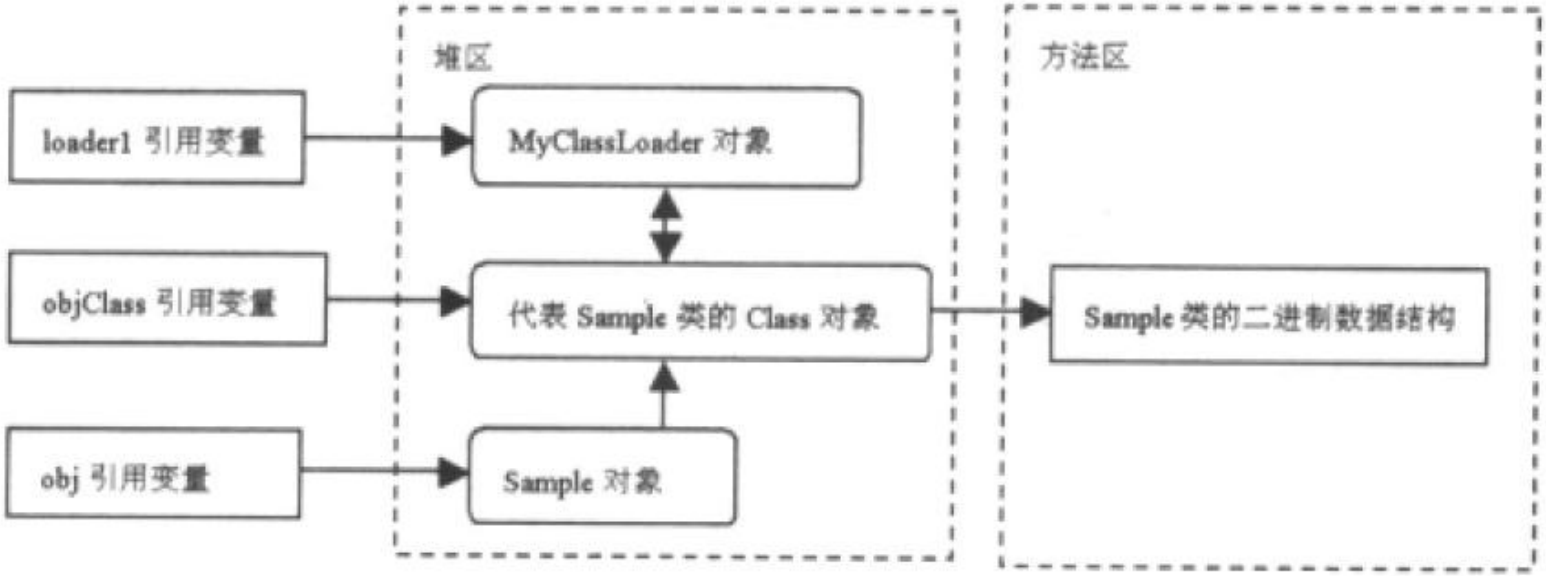
类的卸载

运行以上程序时，Sample 类由 loader1 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 getClassLoader()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 loader1 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 Object 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 类都有一个静态属性 class，它引用代表这个类的 Class 对象，例如：

类的卸载

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



类的卸载

从图 10-9 可以看出，loader1 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。

类的卸载

以上程序的打印结果如下：

```
objClass's hashCode is 7434986
Sample is loaded by loader1
Dog is loaded by loader1
objClass's 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。