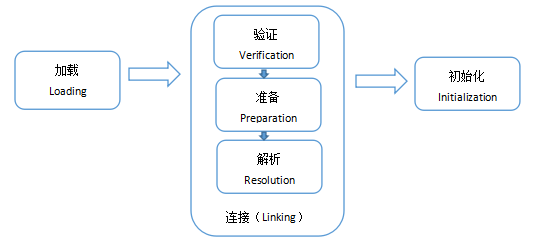
# 第16章 反射

本章将会深入介绍Java类的加载、连接和初始化的知识，并重点介绍Java反射相关的内容，接下来的学习，你可能会感觉比较底层，但是掌握这些底层的运行原理会让读者对Java程序的运行有更好的把握。

## 16.1 类的加载、连接和初始化

当程序主动使用某个类时，如果该类还未被加载到内存中，系统会通过加载、连接、初始化三个步骤来对该类进行初始化，如果没有意外，JVM将会连续完成这三个步骤，所以有时也把这三个步骤统称为类加载或类初始化。



### 16.1.1 类的加载

系统可能在第一次使用某个类时加载该类，但也可能采用预先加载机制来预加载某个类，不管怎样，类的加载必须由类加载器完成，类加载器通常由JVM提供，由JVM提供的这些类加载器通常被称为系统类加载器。除此之外，开发者可以通过继承ClassLoader基类来创建自己的类加载器。

通过使用不同的类加载器，可以从不同来源加载类的二进制数据，通常有如下几种来源：

1）从本地系统直接读取.class文件，这是绝大部分类的加载方法；

2）从zip，jar等归档文件中加载.class文件，这种方式也是很常见的；

3）通过网络下载.class文件或数据

4）从专有数据库中提取.class数据

5）将Java源文件数据上传到服务器中动态编译为.class数据，并执行加载；

但是，不管类的字节码内容从哪里加载，加载的结果都一样，这些字节码内容加载到内存后，都会将这些静态数据转换成方法区的运行时数据结构，然后生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口（即引用地址），所有需要访问和使用类数据只能通过这个Class对象。

### 16.1.2 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，接着将会进入连接阶段，连接阶段将会负责把类的二进制数据合并到JVM的运行状态之中。类连接又可以分为如下三个阶段：

1. 验证：确保加载的类信息符合JVM规范，例如：以cafe开头，没有安全方面的问题
2. 准备：正式为类变量（static）分配内存并设置类变量**默认**初始值的阶段，这些内存都将在方法区中进行分配
3. 解析：虚拟机常量池内的符号引用（常量名）替换为直接引用（地址）的过程

### 16.1.3 类的初始化

类的初始化主要就是对静态的类变量进行初始化：

1. 执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译期自动收集类中所有类变量的显式赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造该类对象的构造器）
2. 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
3. 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步

|  |
| --- |
| **package** com.atguigu.loader.init;  **class** Base{  **private** **static** **int** *a* = *getNum*();  **static**{  ++*a*;  System.***out***.println("(2)a = " + *a*);  }  **static**{  ++*a*;  System.***out***.println("(3)a = " + *a*);  }  **public** **static** **int** getNum(){  System.***out***.println("(1)a = " + *a*);  **return** 1;  }  }  **class** TestClinit **extends** Base{  **private** **static** **int** *b* = *getNum*();  **static**{  ++*b*;  System.***out***.println("(5)b = " + *b*);  }  **static**{  ++*b*;  System.***out***.println("(6)b = " + *b*);  }  **public** **static** **int** getNum(){  System.***out***.println("(4)b = " + *b*);  **return** 1;  }  **public** **static** **void** main(String[] args) {    }  } |
| 以上代码运行结果：  (1)a = 0  (2)a = 2  (3)a = 3  (4)b = 0  (5)b = 2  (6)b = 3 |

虽然类的加载大多数时候和类初始化是一气呵成的，但其实类的加载不一定就会触发类的初始化，当Java程序**首次**通过下面6种方式来使用某个类时，系统就会初始化该类：

* 会发生类的初始化：
  + 当虚拟机启动，先初始化main方法所在的类
  + new一个类的对象
  + 调用该类的静态变量（final的常量除外）和静态方法
  + 使用java.lang.reflect包的方法对类进行反射调用
  + 当初始化一个类，如果其父类没有被初始化，则先会初始化他的父类

|  |
| --- |
| **package** com.atguigu.loader.init;  //当虚拟机启动，先初始化main方法所在的类  **public** **class** A {  **static**{  System.***out***.println("init...A");  }  **public** **static** **void** main(String[] args) {    }  } |
| **package** com.atguigu.loader.init;  //new一个类的对象  **class** B{  **static**{  System.***out***.println("init...B");  }  }  **public** **class** TestB{  **public** **static** **void** main(String[] args) {  **new** B();  }  } |
| **package** com.atguigu.loader.init;  //调用该类的静态变量（final的常量除外）和静态方法  **class** C{  **static**{  System.***out***.println("init...C");  }  **public** **static** **void** test(){  }  }  **public** **class** TestC {  **public** **static** **void** main(String[] args) {  C.*test*();  }  } |
| **package** com.atguigu.loader.init;  //调用该类的静态变量（final的常量除外）和静态方法  **class** C{  **public** **static** **int** *num* = 10;  **static**{  System.***out***.println("init...C");  }  }  **public** **class** TestC {  **public** **static** **void** main(String[] args) {  System.***out***.println(C.*num*);  }  } |
| **package** com.atguigu.loader.init;  //使用java.lang.reflect包的方法对类进行反射调用  **class** D{  **static**{  System.***out***.println("init...D");  }  }  **public** **class** TestD {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  ClassLoader cl = ClassLoader.*getSystemClassLoader*();  cl.loadClass("com.atguigu.loader.D");//该句不会造成类初始化，只是加载类  System.***out***.println("类加载已完成...");  Class.*forName*("com.atguigu.loader.D");//会导致类初始化  }  } |
| **package** com.atguigu.loader.init;  //当初始化一个类，如果其父类没有被初始化，则先会初始化他的父类  **class** EBase{  **static**{  System.***out***.println("父类初始化");  }  }  **public** **class** TestE **extends** EBase {  **static**{  System.***out***.println("子类初始化");  }  **public** **static** **void** main(String[] args) {  }  } |

* 不会发生类的初始化：
  + 引用静态常量不会触发此类的初始化（常量在链接阶段就存入调用类的常量池中了）
  + 当访问一个静态域时，只有真正声明这个域的类才会被初始化
    - 当通过子类引用父类的静态变量，不会导致子类初始化
  + 通过数组定义类引用，不会触发此类的初始化

|  |
| --- |
| **package** com.atguigu.loader.init;  //引用静态常量不会触发此类的初始化  **class** NBase{  **public** **static** **final** **int** ***MAX\_VALUE*** = 100;  **static**{  System.***out***.println("父类初始化");  }  }  **class** NSub **extends** NBase{  **static**{  System.***out***.println("子类初始化");  }  }  **public** **class** TestNoInitialize {  **public** **static** **void** main(String[] args) {  System.***out***.println(NSub.***MAX\_VALUE***);  System.***out***.println(NBase.***MAX\_VALUE***);  }  } |
| **package** com.atguigu.loader.init;  //当访问一个静态域时，只有真正声明这个域的类才会被初始化  // 当通过子类引用父类的静态变量，不会导致子类初始化  **class** NBase{  **public** **static** **int** *num* = 10;  **static**{  System.***out***.println("父类初始化");  }  }  **class** NSub **extends** NBase{  **static**{  System.***out***.println("子类初始化");  }  }  **public** **class** TestNoInitialize {  **public** **static** **void** main(String[] args) {  System.***out***.println(NSub.*num*);  }  } |
| //通过数组定义类引用，不会触发此类的初始化  NSub[] arr = **new** NSub[5]; |

## 16.2 类加载器

很多开发人员都遇到过java.lang.ClassNotFoundException或java.lang.NoClassDefError，想要更好的解决这类问题，或者在一些特殊的应用场景，比如需要支持类的动态加载或需要对编译后的字节码文件进行加密解密操作，那么需要你自定义类加载器，因此了解类加载器及其类加载机制也就成了每一个Java开发人员的必备技能之一。

### 16.2.1 四种类加载器

Java的类加载器由如下四种：

1. 引导类加载器（Bootstrap Classloader）：又称为根类加载器

它负责加载Java的核心库（JAVA\_HOME/jre/lib/rt.jar等或sun.boot.class.path路径下的内容），是用原生代码（C/C++）来实现的，并不继承自java.lang.ClassLoder，所以通过Java代码获取引导类加载器对象将会得到null。

1. 扩展类加载器（Extension ClassLoader）

它由sun.misc.Launcher$ExtClassLoader实现，是java.lang.ClassLoader的子类，负责加载Java的扩展库（JAVA\_HOME/jre/ext/\*.jar或java.ext.dirs路径下的内容）。

1. 应用程序类加载器（Application Classloader）

它由sun.misc.Launcher$AppClassLoader实现，是java.lang.ClassLoader的子类，负责加载Java应用程序类路径（classpath、java.class.path）下的内容。

1. 自定义类加载器

开发人员可以通过继承java.lang.ClassLoader类的方式实现自己的类加载器，以满足一些特殊的需求，例如对字节码进行加密来避免class文件被反编译，或者加载特殊目录下的字节码数据。

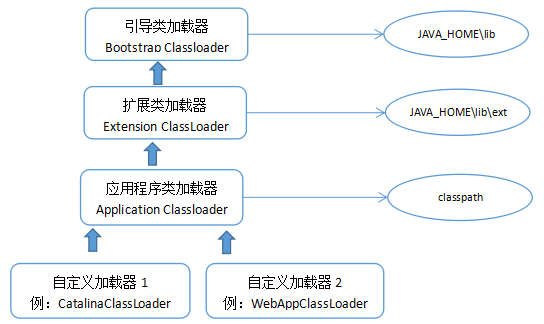
### 16.2.2 经典委托模式

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。

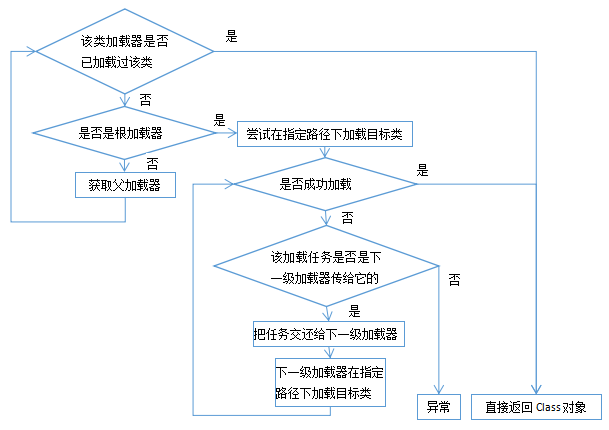
那么，怎么样算是“同一个类”呢？在JVM中，一个类用其全限定类名和其类加载器作为其唯一标识。换句话说，同一个类如果用两个类加载器分别加载，JVM将视为“不同的类”，它们互不兼容。

那么，我们的类加载器在执行类加载任务的时候，如何确保一个类的全局唯一性呢？Java虚拟机的设计者们通过一种称之为“双亲委派模型（Parent Delegation Model）”的委派机制来约定类加载器的加载机制。

按照双亲委派模型的规则，除了引导类加载器之外，程序中的每一个类加载器都应该拥有一个超类加载器，比如：ExtClassLoader的超类加载器是引导类加载器，而AppClassLoader的超类加载器是ExtClassLoader，而自定义类加载器的超类就是AppClassLoader。



那么当一个类加载器接收到一个类加载任务的时候，它并不会立即展开加载，先检测此类是否加载过，即在方法区寻找该类对应的Class对象是否存在，如果存在就是已经加载过了，直接返回该Class对象，否则会将加载任务委派给它的超类加载器去执行，每一层的类加载器都采用相同的方式，直至委派给最顶层的启动类加载器为止，如果超类加载器无法加载委派给它的类时，便会将类的加载任务退回给它的下一级类加载器去执行加载，如果所有的类加载器都加载失败，就会报java.lang.ClassNotFoundException或java.lang.NoClassDefFoundError。



在此大家需要注意，由于Java虚拟机规范并没有要求类加载器的加载机制一定要使用双亲委托模式，只是建议采用这种方式而已。比如在Tomcat中，类加载器所采用的加载机制就和传统的双亲委派模型有一定区别，当缺省的类加载器就接收到一个类的加载任务时，首先会由它自行加载，当它加载失败时，才会将类的加载任务委派给它的超类加载器去执行，这同时也是Servlet规范推荐的一种做法。

说明：数组类型本身并不是由类加载器负责创建，而是由JVM在运行时根据需要而直接创建的，但数组的元素类型仍然需要依靠类加载器去创建。因此，JVM会把数组元素类型的类加载器记录为数组类型的类加载器。

### 16.2.3 java.lang.ClassLoader

ClassLoader 类是一个抽象类，ClassLoader的相关方法：

* public final ClassLoader getParent()：返回委托的父类加载器。一些实现可能使用 null 来表示引导类加载器。
* public static ClassLoader getSystemClassLoader()：返回委托的系统类加载器。
* public Class<?> loadClass(String name)：使用指定的二进制名称（类的全限定名）来加载类。例如：java.lang.String，注意内部类的名称：匿名内部类（外部类的全限定名$编号）、局部内部类（外部类的全限定名$编号+类名）、成员/静态内部类（外部类的全限定名$+类名）。
* protected Class<?> findClass(String name)：使用指定的二进制名称（类的全限定名）来查找类。此方法应该被类加载器的实现重写，该实现按照委托模型来加载类。在通过父类加载器检查所请求的类后，此方法将被 loadClass 方法调用。
* protected final Class<?> findLoadedClass(String name)：返回Class 对象，如果类没有被加载，则返回 null
* protected final Class<?> defineClass(String name,byte[] b,int off,int len)：将一个 byte 数组转换为 Class 类的实例。

自定义类加载器示例代码：

|  |
| --- |
| **package** com.atguigu.loader;  **import** java.io.ByteArrayOutputStream;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileNotFoundException;  **import** java.io.IOException;  **import** java.io.InputStream;  **public** **class** FileClassLoader **extends** ClassLoader{  **private** String rootDir;//指定加载路径    **public** FileClassLoader(String rootDir){  **this**.rootDir = rootDir;  }    @Override  **protected** Class<?> findClass(String name) **throws** ClassNotFoundException {  //首先检查请求的类型是否已经被这个类装载器装载到命名空间中了，如果已经被装载，直接返回；  Class<?> c = findLoadedClass(name);    **if**(c ==**null**){  //委派类加载器请求给父类加载器，如果父类加载器能够完成，则返回父类加载器加载的Class实例；  ClassLoader parent = **this**.getParent();  **try** {  c = parent.loadClass(name);  //加异常处理，父类加载不到，然后自己加载  } **catch** (Exception e) {  }    //调用本类加载器的findClass()方法，试图获取对应的字节码，如果获取的到，则调用defineClass()导入类型到方法区；  //如果获取不到对应的字节码或其他原因失败，则异常，终止加载过程  **if**(c == **null**){  **byte**[] classData = getClassData(name);  **if**(classData == **null**){  **throw** **new** ClassNotFoundException();  }**else**{  c = defineClass(name, classData, 0, classData.length);  }  }  }  **return** c;  }    //把.class文件的内容读取到一个字节数组中  //为什么要读取的字节数组中，因为protected final Class<?> defineClass(String name,byte[] b,int off,int len)  **private** **byte**[] getClassData(String name) {  String path = rootDir + File.***separator*** + name.replace(".", File.***separator***)+".class";  InputStream is = **null**;  ByteArrayOutputStream baos = **null**;  **try** {  is = **new** FileInputStream(path);  baos =**new** ByteArrayOutputStream();  **byte**[] buffer = **new** **byte**[1024];  **int** len;  **while**((len = is.read(buffer))!=-1){  baos.write(buffer, 0, len);  }  **return** baos.toByteArray();  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }**finally**{  **try** {  **if**(is!=**null**){  is.close();  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  **return** **null**;  }  } |
| **package** com.atguigu.loader;  **public** **class** TestFileClassLoader {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  FileClassLoader fsc = **new** FileClassLoader("D:/atguigu/code");  Class<?> uc = fsc.loadClass("com.atguigu.UserManager");  System.***out***.println(uc);    Class<?> sc = fsc.loadClass("java.lang.String");  System.***out***.println(sc);  System.***out***.println(sc.getClassLoader());//null,因为委托给父类加载器...一直到引导类加载器  }  } |

### 16.2.4 使用类加载器加载资源文件

ClassLoader类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个Java类，即java.lang.Class类的一个实例。除此之外，ClassLoader还负责加载Java应用所需的资源，如图像文件和配置文件等。

加载类路径下（例如：src下）jdbc.properties资源文件的示例代码：

|  |
| --- |
| username=root  password=123456  url=jdbc:mysql://localhost:3306/test |
| **package** com.atguigu.loader;  **import** java.io.IOException;  **import** java.util.Properties;  **public** **class** TestLoaderProperties {  **public** **static** **void** main(String[] args) {  Properties pro = **new** Properties();  **try** {  pro.load(ClassLoader.*getSystemResourceAsStream*("jdbc.properties"));  System.***out***.println(pro);  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

如果发布在tomcat中web应用，使用如上代码是无法加载src类路径下的资源文件的，src下的资源文件会随着类被发布到web应用的WEB-INF\classes目录下，而tomcat使用自定义类加载器加载该路径下的内容。那么就需要通过该路径下的类先获取到该自定义加载器对象，然后再调用getSystemResourceAsStream(String name)去加载。例如：

|  |
| --- |
| Properties pro = **new** Properties();  **try** {  ClassLoader loader = DBUtils.**class**.getClassLoader();  InputStream is = loader.getResourceAsStream("druid.properties");  pro.load(is);  System.***out***.println(pro);  } **catch** (IOException e) {  e.printStackTrace();  } |

## 16.3 通过反射查看类信息

Java程序中，所有的对象都有两种类型：编译时类型和运行时类型，而很多时候对象的编译时类型和运行时类型不一致。

例如：我们某些变量或形参的类型是Object类型，但是程序确需要调用该对象运行时类型的方法，该方法不是Object中方法，那么如何解决呢？

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，我们有两种方法：

第一种是在编译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符将其转换成运行时类型的变量即可。

第二种是编译时根本无法预知该对象和类的真实信息，程序只能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。

因为加载完类之后，在堆内存的方法区中就产生了一个Class类型的对象（一个类只有一个Class对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：反射。

### 16.3.1 java.lang.Class

#### 1、哪些类型有Class对象

Class 类的实例表示正在运行的 Java 应用程序中的类和接口。事实上，所有类型都可以表示为Class的实例对象。

（1）class：外部类，内部类

（2）interface：接口

（3）[]：数组，所有具有相同元素类型和维数的数组共享同一个Class 对象

（4）enum：枚举

（5）annotation：注解@interface

（6）primitive type：8种基本数据类型

（7）void

#### 2、获得Class对象

Java程序中可以通过以下四种方式获得Class对象：

1. 类型名.class：适用于编译期间已知的任意类型；
2. 调用任意对象的getClass()方法，可以获取该对象的运行时类型的Class对象；
3. 使用Class类的forName(String name)静态方法，该方法需要传入一个字符串参数，该值是某个类的全限定名（全限定名即完整的包.类型名）；该方法适用于除了数组以外的任意引用数据类型；
4. 调用类加载对象的loadClass(String name)该方法需要传入一个字符串参数，该值是某个类的全限定名（全限定名即完整的包.类型名）；

方式（1）只适用于编译器间已知的类型，如果某个类型编译期间是已知的，优先考虑这种方式，代码更安全，效率更高；另外基本数据类型和也只能通过该方式获得Class对象；如果某个类型编译期间未知，我们只能通过某种方式获取该类型的全名称的字符串形式，那么就只能选择（3）和（4）了，但是该方法在运行期间仍然无法加载该类的话，会报ClassNotFoundException。对于基本数据类型和数组类型，无法通过第（3）和（4）方式获取。

示例代码：获取Class对象的四种方式

|  |
| --- |
| @Test  **public** **void** test() **throws** ClassNotFoundException{  Class<?> c1 = String.**class**;  Class<?> c2 = "hello".getClass();  Class<?> c3 = Class.*forName*("java.lang.String");  Class<?> c4 = ClassLoader.*getSystemClassLoader*().loadClass("java.lang.String");  System.***out***.println(c1 == c2);  System.***out***.println(c1 == c3);  System.***out***.println(c1 == c4);  } |
| @Test  **public** **void** test1(){  Class<?> c1 = **int**.**class**;//基本数据类型  Class<?> c2 = **void**.**class**;//void类型    Class<?> c3 = String.**class**;//类  Class<?> c4 = Object.**class**;//类  Class<?> c5 = Class.**class**;//类  Class<?> c6 = Comparable.**class**;//接口  //只要元素类型与维度一样，就是同一个Class  Class<?> c7 = **int**[].**class**;  **int**[] arr1 = **new** **int**[5];  **int**[] arr2 = **new** **int**[10];  System.***out***.println(arr1.getClass() == c7);  System.***out***.println(arr2.getClass() == c7);  Class c9 = String[].**class**;  Class c10 = **int**[][].**class**;  System.***out***.println(c7 == c9);  System.***out***.println(c7 == c10);    Class c11 = Override.**class**;//注解  Class c12 = ElementType.**class**;//枚举  } |

### 16.3.2 从Class中获取信息

Class类提供了大量实例方法来获取该Class对象所对应类的详细信息，Class类大致包含如下几种方法，下面每种方法都可能包含多个重载的版本。

例如：包、修饰符、类名、父类、父接口、注解，及成员（属性、构造器、方法）等

反射相关的API主要是java.lang.Class和java.lang.reflect包的内容。

#### 1、获取某个类的加载器

public ClassLoader getClassLoader()：返回该类的类加载器。有些实现可能使用 null 来表示引导类加载器。如果此对象表示一个基本类型或 void，则返回 null。

#### 2、获取包名和类型名

public Package getPackage()：获取此类的包。然后可以通过Package实例对象的getName()获取包名。

public String getName()：以 String 的形式返回此 Class 对象所表示的实体（类、接口、数组类、基本类型或 void）名称。

|  |
| --- |
| @Test  **public** **void** test1(){  //java.lang.String  System.***out***.println(String.**class**.getName());  //int  System.***out***.println(**int**.**class**.getName());  //[I  System.***out***.println(**int**[].**class**.getName());  //[[[I  System.***out***.println(**int**[][][].**class**.getName());  //[Ljava.lang.Object;  System.***out***.println(Object[].**class**.getName());  } |

如果此类对象表示的是非数组类型的引用类型，则返回该类的二进制名称，即包.类名。

如果此类对象表示一个基本类型或 void，则返回该基本类型或 void 所对应的 Java 语言关键字相同的字符串名称。

如果此类对象表示一个数组类，则名字的内部形式为：表示该数组嵌套深度的一个或多个 '[' 字符加元素类型名。元素类型名的编码如下：

|  |  |
| --- | --- |
| Element | Type Encoding |
| byte | B |
| short | S |
| int | I |
| long | L |
| float | F |
| double | D |
| boolean | Z |
| char | C |
| class or interface | Lclassname; |

#### 3、获取类型修饰符

public int getModifiers()：返回此类或接口以整数编码的 Java 语言修饰符。

修饰符由 Java 虚拟机的 public、protected、private、final、static、abstract 和 interface 对应的常量组成；它们应当使用 Modifier 类的方法来解码。

如果底层类是数组类，则其 public、private 和 protected 修饰符与其组件类型的修饰符相同。如果此 Class 表示一个基本类型或 void，则其 public 修饰符始终为 true，protected 和 private 修饰符始终为 false。如果此对象表示一个数组类、一个基本类型或 void，则其 final 修饰符始终为 true，其接口修饰符始终为 false。该规范没有给定其他修饰符的值。

|  |
| --- |
| @Test  **public** **void** test2(){  Class<?> clazz = String.**class**;  **int** mod = clazz.getModifiers();  System.***out***.println(Modifier.*toString*(mod));//public final  System.***out***.println(Modifier.*isPublic*(mod));//true  } |

#### 4、获取父类或父接口

public Class<? super T> getSuperclass()：返回表示此 Class 所表示的实体（类、接口、基本类型或 void）的超类的 Class。如果此 Class 表示 Object 类、一个接口、一个基本类型或 void，则返回 null。如果此对象表示一个数组类，则返回表示该 Object 类的 Class 对象。

public Class<?>[] getInterfaces()：确定此对象所表示的类或接口实现的接口。如果此对象表示一个类，则返回值是一个数组，它包含了表示该类所实现的所有接口的对象。数组中接口对象顺序与此对象所表示的类的声明的 implements 子句中接口名顺序一致。如果此对象表示一个接口，则该数组包含表示该接口扩展的所有接口的对象。数组中接口对象顺序与此对象所表示的接口的声明的 extends 子句中接口名顺序一致。 如果此对象表示一个不实现任何接口的类或接口，则此方法返回一个长度为 0 的数组。如果此对象表示一个基本类型或 void，则此方法返回一个长度为 0 的数组。

|  |
| --- |
| @Test  **public** **void** test3(){  System.***out***.println(Integer.**class**.getSuperclass());//Number  System.***out***.println(**int**.**class**.getSuperclass());//null  System.***out***.println(Runnable.**class**.getSuperclass());//null  System.***out***.println(**int**[].**class**.getSuperclass());//Object  System.***out***.println(String[].**class**.getSuperclass());//Object  } |
| @Test  **public** **void** test4(){  Class<?> clazz = String.**class**;  Class<?>[] interfaces = clazz.getInterfaces();  **for** (Class<?> inter : interfaces) {  System.***out***.println(inter);  }  } |

#### 5、获取内部类或外部类信息

public Class<?>[] getClasses()：返回所有公共内部类和内部接口。包括从超类继承的公共类和接口成员以及该类声明的公共类和接口成员。

public Class<?>[] getDeclaredClasses()：返回 Class 对象的一个数组，这些对象反映声明为此 Class 对象所表示的类的成员的所有类和接口。包括该类所声明的公共、保护、默认（包）访问及私有类和接口，但不包括继承的类和接口。

public Class<?> getDeclaringClass()：如果此 Class 对象所表示的类或接口是一个内部类或内部接口，则返回它的外部类或外部接口，否则返回null。

|  |
| --- |
| @Test  **public** **void** test5(){  Class<?> clazz = Map.**class**;  Class<?>[] inners = clazz.getDeclaredClasses();  **for** (Class<?> inner : inners) {  System.***out***.println(inner);  }    Class<?> ec = Map.Entry.**class**;  Class<?> outer = ec.getDeclaringClass();  System.***out***.println(outer);  } |

#### 6、获取属性

四个方法用于访问Class对应类所包含的属性（Field）：

public Field[] getFields()：返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段。返回数组中的元素没有排序，也没有任何特定的顺序。包括继承的公共字段。

public Field getField(String name)：返回一个 Field 对象，它反映此 Class 对象所表示的类或接口的指定公共成员字段。包括继承的公共字段。name 参数是一个 String，用于指定所需字段的简称。

public Field[] getDeclaredFields()：返回 Field 对象的一个数组，这些对象反映此 Class 对象所表示的类或接口所声明的所有字段。包括公共、保护、默认（包）访问和私有字段，但不包括继承的字段。返回数组中的元素没有排序，也没有任何特定的顺序。

public Field getDeclaredField(String name)：返回一个 Field 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明字段。

|  |
| --- |
| @Test  **public** **void** test6(){  Class<?> clazz = String.**class**;  Field[] fields = clazz.getDeclaredFields();  **for** (Field field : fields) {  **int** mod = field.getModifiers();  Class<?> type = field.getType();  String name = field.getName();  System.***out***.print(Modifier.*toString*(mod)+"\t");  System.***out***.println(type.getName()+"\t" + name);  }  } |

#### 7、获取构造器

四个方法用于访问Class对应的类所包含的构造器（Constructor）：

public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)：构造器名称无需指定，因为它和类名一致。parameterTypes 参数是 Class 对象的一个数组，它按声明顺序标识构造方法的形参类型。 如果此 Class 对象表示非静态上下文中声明的内部类，则形参类型作为第一个参数包括显示封闭的实例。

public Constructor<?>[] getDeclaredConstructors()：它们是公共、保护、默认（包）访问和私有构造方法。

public Constructor<T> getConstructor(Class<?>... parameterTypes)：指定公共构造方法

public Constructor<?>[] getConstructors()：所有公共构造方法

|  |
| --- |
| **public** **class** TestConstructor{  @Test  **public** **void** test7() **throws** Exception{  Class<?> clazz = Outer.**class**;  Constructor<?> constructor = clazz.getDeclaredConstructor();  System.***out***.println(constructor);//无参构造    Class<?> c = Outer.Inner.**class**;  //因为Inner是非静态的内部类，所以它的构造器，默认第一个形参是外部类的实例对象  Constructor<?> cs = c.getDeclaredConstructor(Outer.**class**);  System.***out***.println(cs);  }  }  **class** Outer{  **class** Inner{    }  } |
| @Test  **public** **void** test8(){  Class<?> clazz = String.**class**;  Constructor<?>[] constructors = clazz.getConstructors();  **for** (Constructor<?> constructor : constructors) {  **int** mod = constructor.getModifiers();  String name = constructor.getName();  Class<?>[] parameterTypes = constructor.getParameterTypes();  System.***out***.print(Modifier.*toString*(mod)+"\t" + name + "(");  System.***out***.println(Arrays.*toString*(parameterTypes)+")");  }  } |

#### 8、获取方法

四个方法用于访问Class对应的类所包含的方法（Method）：

public Method getDeclaredMethod(String name,Class<?>... parameterTypes)：name 参数是一个 String，它指定所需方法的简称，parameterTypes 参数是 Class 对象的一个数组或0~n个Class对象，它按声明顺序标识该方法的**形参类型**。如果是无参方法，那么parameterTypes 可以不传或者传null。因为可能存在重载的方法，所以在一个类中唯一确定一个方法，需要方法名和形参类型列表。

public Method[] getDeclaredMethods()：包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。

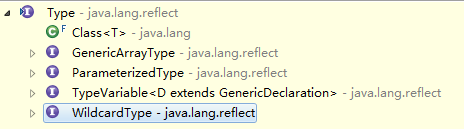
public Method getMethod(String name,Class<?>... parameterTypes)：指定的公共成员方法。包括继承的公共方法。

public Method[] getMethods()：所有公共成员方法。包括继承的公共方法。

|  |
| --- |
| @Test  **public** **void** test9(){  Class<?> clazz = String.**class**;  Method[] methods = clazz.getMethods();  **for** (Method method : methods) {  **int** mod = method.getModifiers();  Class<?> returnType = method.getReturnType();  String name = method.getName();  Class<?>[] parameterTypes = method.getParameterTypes();  System.***out***.print(Modifier.*toString*(mod)+"\t" + returnType + "\t" + name + "(");  System.***out***.println(Arrays.*toString*(parameterTypes)+")");  }  } |

#### 9、获取泛型父类

JDK1.5引入的泛型，为了通过反射操作这些泛型，新增了ParameterizedType，GenericArrayType，TypeVariable和WildcardType几种类型来代表不能被归一到Class中的类型但是又和原始类型齐名的类型。



而在Class类、Field类、Method类等API中增加了很多关于获取泛型信息的方法，例如在Class类中就有很多，其中有一个获取泛型父类的方法：

public Type getGenericSuperclass()：返回表示此 Class 所表示的实体（类、接口、基本类型或 void）的直接**超类的 Type**。

示例代码：

|  |
| --- |
| **package** com.atguigu.reflect;  **import** java.lang.reflect.ParameterizedType;  **import** java.lang.reflect.Type;  **import** java.lang.reflect.TypeVariable;  **public** **class** TestGenericSuperClass {  **public** **static** **void** main(String[] args) {  Class<?> c = Base.**class**;  TypeVariable<?>[] typeParameters = c.getTypeParameters();  **for** (TypeVariable<?> typeVariable : typeParameters) {  System.***out***.println(typeVariable + "，上限：" + typeVariable.getBounds()[0]);  }    Class<Sub> clazz = Sub.**class**;  Type gs = clazz.getGenericSuperclass();    ParameterizedType gt = (ParameterizedType)gs;  Type[] types = gt.getActualTypeArguments();  **for** (Type type : types) {  System.***out***.println(type);  }  }  }  **class** Base<T **extends** Number>{    }  **class** Sub **extends** Base<Integer>{    } |

#### 10、获取注解信息

可以通过反射API，获得相关的注解信息。

public Annotation[] getAnnotations() ：返回此元素上存在的所有注释。

public Annotation[] getDeclaredAnnotations()：获取某元素上存在的所有注释。该方法将忽略继承的注释。

public <T extends Annotation> T getAnnotation(Class<T> annotationClass)：如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null。

示例代码：

|  |
| --- |
| **package** com.atguigu.reflect;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.RetentionPolicy;  **public** **class** TestAnnotation {  **public** **static** **void** main(String[] args) {  Class<?> clazz = MyClass.**class**;  MyAnnotation my = clazz.getAnnotation(MyAnnotation.**class**);  System.***out***.println(my.value());  }  }  @MyAnnotation  **class** MyClass{    }  @Retention(RetentionPolicy.***RUNTIME***)  **@interface** MyAnnotation{  String value() **default** "尚硅谷";  } |

提示：要想通过反射获取到某个注解的信息，该注解声明时必须加@Retention(RetentionPolicy.***RUNTIME***)元注解，表明滞留注解信息到运行时。

## 16.4 使用反射生成并操作对象

### 16.4.1 使用反射创建对象

通过反射来生成对象有如下两种方式：

方式一：使用Class对象的newInstance()方法来创建该Class对象对应类的实例，这种方式要求该Class对象的对应类有默认构造器，而执行newInstance()方法时实际上是利用默认构造器来创建该类的实例。

方式二：先使用Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance(Object... args)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某个类的指定构造器来创建实例。

通过第一种方式来创建对象是比较常见的情形，因为在很多JavaEE框架中都需要根据配置文件信息来创建实例对象，从配置文件读取的只是某个类的字符串类名，程序就需要根据该字符串来创建对应的实例，就必须使用反射。

使用两种方式创建实例对象的示例代码：

|  |
| --- |
| **package** com.atguigu.reflect;  **import** java.lang.reflect.Constructor;  **import** org.junit.Test;  **public** **class** TestNewInstance {  @Test  **public** **void** test1() **throws** Exception{  Class<?> clazz = Class.*forName*("com.atguigu.reflect.Student");  Object obj = clazz.newInstance();  System.***out***.println(obj);  }    @Test  **public** **void** test2() **throws** Exception{  Class<?> clazz = Class.*forName*("com.atguigu.reflect.Student");  Constructor<?> constructor = clazz.getDeclaredConstructor(String.**class**);  Object obj = constructor.newInstance("佟刚");  System.***out***.println(obj);  }  }  **class** Student{  **private** String name;  **public** Student(String name) {  **super**();  **this**.name = name;  }  **public** Student() {  **super**();  }  @Override  **public** String toString() {  **return** "Student [name=" + name + "]";  }  } |

### 16.4.2 获取或设置某个对象的属性值

通过Class对象的getFields()等方法可以获取该类所包括的全部Field（属性）或指定Field。而Field类除了提供获取属性的修饰符、属性类型、属性名等方法外，还提供了如下两组方法来访问属性：

public xxx getXxx(Object obj)：获取obj对象该Field的属性值。此处的Xxx对应8种基本数据类型，如果该属性的类型是引用数据类型，则直接使用get(Object obj)方法。

public void setXxx(Object obj,Xxx value)：设置obj对象该Field的属性值为value。此处的Xxx对应8种基本数据类型，如果该属性的类型是引用数据类型，则直接使用set(Object obj, Object value)方法。

public void setAccessible(boolean flag)启动和禁用访问安全检查的开关。

* 值为true则指示反射的对象在使用时应该取消Java语言访问检查。
  + 提高反射的效率。如果代码中必须用反射，而该句代码需要频繁的被调用，那么请设置为true.
  + 使得原本无法访问的私有成员也可以访问
* 值为false则指示反射的对象应该实施Java语言访问检查。

获取或设置某个对象的属性值示例代码：

|  |
| --- |
| **package** com.atguigu.reflect;  **import** java.lang.reflect.Field;  **public** **class** TestField {  **public** **static** **void** main(String[] args)**throws** Exception {  Class<?> clazz = Class.*forName*("com.atguigu.reflect.Circle");  Object obj = clazz.newInstance();  Field field = clazz.getDeclaredField("radius");  field.setAccessible(**true**);  field.set(obj, 1.2);  Object value = field.get(obj);  System.***out***.println(value);  }  }  **class** Circle{  **private** **double** radius;  } |

### 16.4.3 调用方法

当获得某个类对应的Class对象后，就可以通过该Class对象的getMethods()等方法获取全部方法或指定方法。每个Method对象对应一个方法，获得Method对象后，程序就可以通过该Method对象的invoke方法来调用对应方法。

示例代码：

|  |
| --- |
| **package** com.atguigu.reflect;  **import** java.lang.reflect.Method;  **public** **class** TestMethod {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> clazz = Class.*forName*("com.atguigu.reflect.Utils");  Object obj = clazz.newInstance();  Method method = clazz.getMethod("check", String.**class**,String.**class**);  Object value = method.invoke(obj, "tong","666");  System.***out***.println(value);  }  }  **class** Utils{  **public** **boolean** check(String user,String password){  **if**("admin".equals(user) && "123".equals(password)){  **return** **true**;  }**else**{  **return** **false**;  }  }  } |

### 16.4.5 操作数组

在java.lang.reflect包下还提供了一个Array类，Array对象可以代表所有的数组。程序可以通过使用Array类来动态的创建数组，操作数组元素等。

Array类提供了如下几个方法：

public static Object newInstance(Class<?> componentType, int... dimensions)：创建一个具有指定的组件类型和维度的新数组。

public static void setXxx(Object array,int index,xxx value)：将array数组中[index]元素的值修改为value。此处的Xxx对应8种基本数据类型，如果该属性的类型是引用数据类型，则直接使用set(Object array,int index, Object value)方法。

public static xxx getXxx(Object array,int index,xxx value)：将array数组中[index]元素的值返回。此处的Xxx对应8种基本数据类型，如果该属性的类型是引用数据类型，则直接使用get(Object array,int index)方法。

示例代码：

|  |
| --- |
| **package** com.atguigu.reflect;  **import** java.lang.reflect.Array;  **public** **class** TestArray {  **public** **static** **void** main(String[] args) {  Object arr = Array.*newInstance*(String.**class**, 5);  Array.*set*(arr, 0, "尚硅谷");  Array.*set*(arr, 1, "佟刚");  System.***out***.println(Array.*get*(arr, 0));  System.***out***.println(Array.*get*(arr, 1));  System.***out***.println(Array.*get*(arr, 2));  }  } |

## 16.5 动态代理

在Java的java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口，通过使用这个类和接口可以生成JDK动态代理类或动态代理对象。

Proxy提供用于创建动态代理类和代理对象的静态方法，它也是所有动态代理类的父类。如果我们在程序中为一个或多个接口动态地生成实现类，就可以使用Proxy来创建动态代理类或它们的实例。

* public static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)：创建一个动态代理类所对应的Class对象
* public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)：直接创建一个动态代理对象。第一个参数，被代理类的类加载器对象，第二个参数，被代理类实现的接口们，第三个参数，代理类代理工作处理器对象。

InvocationHandler接口，有一个invoke方法需要实现，该invoke方法中三个参数分别是proxy，代表动态代理对象，method，代表正在执行的方法，args，代表执行代理对象的方法时传入的实参。

案例需求：我需要在所有的Dog，Person，Bird等接口的实现类所有实现方法加上统计方法的执行时间，并要求不修改这些实现类的代码。

接口们：

|  |
| --- |
| **package** com.atguigu.proxy;  **public** **interface** Dog {  **void** bark();  **void** run();  } |
| **package** com.atguigu.proxy;  **public** **interface** Person {  **void** study();  **void** think();  } |
| **package** com.atguigu.proxy;  **public** **interface** Bird {  **void** jump();  **void** fly();  } |

实现类们：

|  |
| --- |
| **package** com.atguigu.proxy;  **public** **class** TibetanMastiff **implements** Dog{  @Override  **public** **void** bark() {  System.***out***.println("藏獒在叫");  }  @Override  **public** **void** run() {  System.***out***.println("藏獒在跑");  }  } |
| **package** com.atguigu.proxy;  **public** **class** Chinese **implements** Person{  @Override  **public** **void** study() {  System.***out***.println("中国人在学习");  }  @Override  **public** **void** think() {  System.***out***.println("中国人在思考");  }  } |
| **package** com.atguigu.proxy;  **public** **class** Magpie **implements** Bird{  @Override  **public** **void** jump() {  System.***out***.println("喜鹊在跳来跳去");  }  @Override  **public** **void** fly() {  System.***out***.println("喜鹊飞来了");  }  } |

代理类处理器，必须实现InvocationHandler接口。

|  |
| --- |
| **package** com.atguigu.proxy;  **import** java.lang.reflect.InvocationHandler;  **import** java.lang.reflect.Method;  **public** **class** TimeInvocationHandler **implements** InvocationHandler{  **private** Object target;    **public** TimeInvocationHandler(Object target) {  **super**();  **this**.target = target;  }  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  System.***out***.println("开始计时...");  **long** start = System.*currentTimeMillis*();    Object returnValue = method.invoke(target, args);    **long** end = System.*currentTimeMillis*();  System.***out***.println("方法执行时间：" + (end-start));  **return** returnValue;  }  } |

动态创建代理对象

|  |
| --- |
| **package** com.atguigu.proxy;  **import** java.lang.reflect.Proxy;  **import** org.junit.Test;  **public** **class** TestProxy {    @Test  **public** **void** test1(){  TibetanMastiff target = **new** TibetanMastiff();  Class<?> clazz = TibetanMastiff.**class**;  TimeInvocationHandler handler = **new** TimeInvocationHandler(target);  Dog dog = (Dog) Proxy.*newProxyInstance*(clazz.getClassLoader(), clazz.getInterfaces(), handler);  dog.bark();  dog.run();  }    @Test  **public** **void** test2(){  Chinese target = **new** Chinese();  Class<?> clazz = Chinese.**class**;  TimeInvocationHandler handler = **new** TimeInvocationHandler(target);  Person person = (Person) Proxy.*newProxyInstance*(clazz.getClassLoader(), clazz.getInterfaces(), handler);  person.study();  person.think();  }    @Test  **public** **void** test3(){  Magpie target = **new** Magpie();  Class<?> clazz = Magpie.**class**;  TimeInvocationHandler handler = **new** TimeInvocationHandler(target);  Bird bird = (Bird) Proxy.*newProxyInstance*(clazz.getClassLoader(), clazz.getInterfaces(), handler);  bird.jump();  bird.fly();  }  } |

动态代理对象需要实现一个或多个接口的方法时，它是如何知道怎么实现这些方法的呢？实际上它实现的方法体，其实就是执行InvocationHandler对象的invoke()方法。

不难发现，使用动态代理可以非常灵活的实现解耦合，这种动态代理在Spring框架体系的AOP（Aspect Orient Program，即面向切面编程）里被称为AOP代理，AOP代理可替代目标对象，AOP代理包含了目标对象的全部方法。但AOP代理中的方法与目标对象的方法存在差异：AOP代理里的方法可以在执行目标方法之前、后插入一下通用处理。

## 16.6 本章小结

本章详细介绍了Java反射的相关知识，本章对于普通Java学习者来说，确实显得有点深入，并且会感觉不太实用。但随着学习的深入，知识的积累，当读者需要开发出更多基础的、适应性更广的的、灵活性更强的代码时，就会想到使用反射知识了。本章从类的加载、初始化开始介绍，深入介绍了Java类加载器的原理和机制。本章重点介绍了Class、Field、Constructor、Method等，包括动态创建Java实例和动态操作对象的属性和方法。本章还介绍了在Spring框架中核心使用的动态代理原理。