Java反射机制

极客学院出版

前言

Java 反射机制可以让我们在编译期(Compile Time)之外的运行期(Runtime)检查类，接口，变量以及方法的信息。反射还可以让我们在运行期实例化对象，调用方法，通过调用 get/set 方法获取变量的值。

Java 反射机制功能强大而且非常实用。目前在互联网上已经有不胜枚举的 Java 反射指南，然而大多数的指南包括 Sun 公司所发布的反射指南中都仅仅只是介绍了一些反射的表面内容以及它的潜能。

在这个系列的文章中，我们会比其他指南更深入的去理解 Java 反射机制，它会阐述 Java 反射机制的基本原理包括如何去使用数组，注解，泛型以及动态代理还有类的动态加载以及类的重载的实现。同时也会向你展示如何实现一些比较有特性的功能，比如从一个类中读取所有的 get/set 方法，或者访问一个类的私有变量以及私有方法。

在这个系列的指南中同时也会说明一些非反射相关的但是令人困惑的问题，比如哪些泛型信息在运行时是有效的，一些人声称所有的泛型信息在运行期都会消失，其实这是不对的。

适用人群

本课程主要适用于希望深入理解并掌握 Java 反射机制的 Java 程序员。

学习前提

在学习本课程之前，我们假定你已经对 Java 语言非常熟悉。

版本信息

书中演示代码基于以下版本：

语言/框架	版本信息
Java	Java 6

目录

1 指南

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gamil.com ）

该系列文章中所描述介绍的是 Java 6 版本的反射机制。

#

Java 反射的例子

下面是一个 Java 反射的简单例子：

```
Method[] methods = MyObject.class.getMethods();

for(Method method : methods){
    System.out.println("method = " + method.getName());
}
```

在这个例子中通过调用 MyObject 类的 class 属性获取对应的 Class 类的对象，通过这个 Class 类的对象获取 MyObject 类中的方法集合。迭代这个方法的集合并且打印每个方法的名字。

本文链接地址: Java Reflection(一):Java反射指南

2

java 类

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gamil.com ）

使用 Java 反射机制可以在运行时期检查 Java 类的信息，检查 Java 类的信息往往是你在使用 Java 反射机制的时候所做的第一件事情，通过获取类的信息你可以获取以下相关的内容：

1. Class 对象
2. 类名
3. 修饰符
4. 包信息
5. 父类
6. 实现的接口
7. 构造器
8. 方法
9. 变量
10. 注解

除了上述这些内容，还有很多的信息你可以通过反射机制获得，如果你想要知道全部的信息你可以查看相应的文档 JavaDoc for java.lang.Class 里面有详尽的描述。

在本节中我们会简短的涉及上述所提及的信息，上述的一些主题我们会使用单独的章节进行更详细的描述，比如这段内容会描述如何获取一个类的所有方法或者指定方法，但是在单独的章节中则会向你展示如何调用反射获得的方法(Method Object)，如何在多个同名方法中通过给定的参数集合匹配到指定的方法，在一个方法通过反射机制调用的时候会抛出那些异常？如何准确的获取 getter/setter 方法等等。本节的内容主要是介绍 Class 类以及你能从Class类中获取哪些信息。

# #

Class 对象

在你想检查一个类的信息之前，你首先需要获取类的 Class 对象。Java 中的所有类型包括基本类型(int, long, float等等)，即使是数组都有与之关联的 Class 类的对象。如果你在编译期知道一个类的名字的话，那么你可以使用如下的方式获取一个类的 Class 对象。

```
Class myObjectClass = MyObject.class;
```

如果你在编译期不知道类的名字，但是你可以在运行期获得到类名的字符串,那么你则可以这么做来获取 Class 对象:

```
String className = ... ;//在运行期获取的类名字符串
Class class = Class.forName(className);
```

在使用 Class.forName() 方法时，你必须提供一个类的全名，这个全名包括类所在的包的名字。例如 MyObject 类位于 com.jenkov.myapp 包，那么他的全名就是 com.jenkov.myapp.MyObject。 如果在调用Class.forName()方法时，没有在编译路径下(classpath)找到对应的类，那么将会抛出ClassNotFoundException。

# #

类名

你可以从 Class 对象中获取两个版本的类名。

通过 getName() 方法返回类的全限定类名（包含包名）：

```
Class aClass = ... //获取Class对象，具体方式可见Class对象小节
String className = aClass.getName();
```

如果你仅仅只是想获取类的名字(不包含包名)，那么你可以使用 getSimpleName()方法:

```
Class aClass = ... //获取Class对象，具体方式可见Class对象小节
String simpleClassName = aClass.getSimpleName();
```

# #

修饰符

可以通过 Class 对象来访问一个类的修饰符， 即public,private,static 等等的关键字，你可以使用如下方法来获取类的修饰符：

```
Class  aClass = ... //获取Class对象，具体方式可见Class对象小节
int modifiers = aClass.getModifiers();
```

修饰符都被包装成一个int类型的数字，这样每个修饰符都是一个位标识(flag bit)，这个位标识可以设置和清除修饰符的类型。 可以使用 java.lang.reflect.Modifier 类中的方法来检查修饰符的类型：

```
Modifier.isAbstract(int modifiers);
Modifier.isFinal(int modifiers);
Modifier.isInterface(int modifiers);
Modifier.isNative(int modifiers);
Modifier.isPrivate(int modifiers);
Modifier.isProtected(int modifiers);
Modifier.isPublic(int modifiers);
Modifier.isStatic(int modifiers);
Modifier.isStrict(int modifiers);
Modifier.isSynchronized(int modifiers);
Modifier.isTransient(int modifiers);
Modifier.isVolatile(int modifiers);
```

#

包信息

可以使用 Class 对象通过如下的方式获取包信息：

```
Class  aClass = ... //获取Class对象，具体方式可见Class对象小节
Package package = aClass.getPackage();
```

通过 Package 对象你可以获取包的相关信息，比如包名，你也可以通过 Manifest 文件访问位于编译路径下 jar 包的指定信息，比如你可以在 Manifest 文件中指定包的版本编号。更多的 Package 类信息可以阅读 java.lang.Package。

# #

父类

通过 Class 对象你可以访问类的父类，如下例：

```
Class superclass = aClass.getSuperclass();
```

可以看到 superclass 对象其实就是一个 Class 类的实例，所以你可以继续在这个对象上进行反射操作。

#

实现的接口

可以通过如下方式获取指定类所实现的接口集合：

```
Class  aClass = ... //获取Class对象，具体方式可见Class对象小节
Class[] interfaces = aClass.getInterfaces();
```

由于一个类可以实现多个接口，因此 getInterfaces(); 方法返回一个 Class 数组，在 Java 中接口同样有对应的 Class 对象。注意：getInterfaces() 方法仅仅只返回当前类所实现的接口。当前类的父类如果实现了接口，这些接口是不会在返回的 Class 集合中的，尽管实际上当前类其实已经实现了父类接口。

# #

构造器

你可以通过如下方式访问一个类的构造方法：

```
Constructor[] constructors = aClass.getConstructors();
```

更多有关 Constructor 的信息可以访问 Constructors。

# #

方法

你可以通过如下方式访问一个类的所有方法：

```
Method[] method = aClass.getMethods();
```

# #

变量

你可以通过如下方式访问一个类的成员变量：

```
Field[] method = aClass.getFields();
```

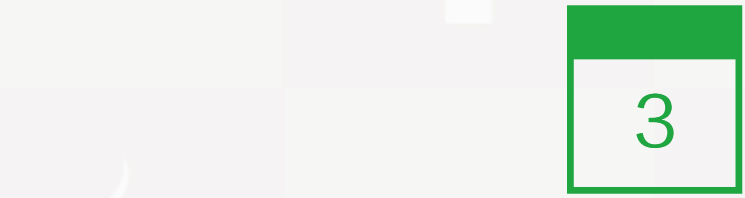
更多有关 Field 的信息可以访问 Fields。

# #

注解

你可以通过如下方式访问一个类的注解：

```
Annotation[] annotations = aClass.getAnnotations();
```



3

Java 构造器

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gamil.com ）

# #

内容索引

1. 获取 Constructor 对象
2. 构造方法参数
3. 利用 Constructor 对象实例化一个类

利用 Java 的反射机制你可以检查一个类的构造方法，并且可以在运行期创建一个对象。这些功能都是通过 java.lang.reflect.Constructor 这个类实现的。本节将深入的阐述 Java Constructor 对象。

#

获取 Constructor 对象

我们可以通 过 Class 对象来获取 Constructor 类的实例：

```
Class aClass = ...//获取Class对象
Constructor[] constructors = aClass.getConstructors();
```

返回的 Constructor 数组包含每一个声明为公有的（ Public ）构造方法。 如果你知道你要访问的构造方法的方法参数类型，你可以用下面的方法获取指定的构造方法，这例子返回的构造方法的方法参数为 String 类型：

```
Class aClass = ...//获取Class对象
Constructor constructor =
  aClass.getConstructor(new Class[]{String.class});
```

如果没有指定的构造方法能满足匹配的方法参数则会抛出：NoSuchMethodException。

# #

构造方法参数

你可以通过如下方式获取指定构造方法的方法参数信息：

```
Constructor constructor = ... //获取Constructor对象
Class[] parameterTypes = constructor.getParameterTypes();
```

#

利用 Constructor 对象实例化一个类

你可以通过如下方法实例化一个类：

```
Constructor constructor = MyObject.class.getConstructor(String.class);
MyObject myObject = (MyObject)
 constructor.newInstance("constructor-arg1");
```

constructor.newInstance()方法的方法参数是一个可变参数列表，但是当你调用构造方法的时候你必须提供精确的参数，即形参与实参必须一一对应。在这个例子中构造方法需要一个 String 类型的参数，那我们在调用 newInstance 方法的时候就必须传入一个 String 类型的参数。

本文链接地址: Java Reflection(三):构造器

4

Java 变量

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gamil.com ）

# #

内容索引

1. 获取 Field 对象
2. 变量名称
3. 变量类型
4. 获取或设置（get/set）变量值

使用 Java 反射机制你可以运行期检查一个类的变量信息(成员变量)或者获取或者设置变量的值。通过使用 java.lang.reflect.Field 类就可以实现上述功能。在本节会带你深入了解 Field 对象的信息。

#

获取 Field 对象

可以通过 Class 对象获取 Field 对象，如下例：

```
Class aClass = ...//获取Class对象
Field[] methods = aClass.getFields();
```

返回的 Field 对象数组包含了指定类中声明为公有的(public)的所有变量集合。 如果你知道你要访问的变量名称，你可以通过如下的方式获取指定的变量：

```
Class  aClass = MyObject.class
Field field = aClass.getField("someField");
```

上面的例子返回的Field类的实例对应的就是在 MyObject 类中声明的名为 someField 的成员变量，就是这样：

```
public class MyObject{
  public String someField = null;
}
```

在调用 getField()方法时，如果根据给定的方法参数没有找到对应的变量，那么就会抛出 NoSuchFieldException。

#

变量名称

一旦你获取了 Field 实例，你可以通过调用 Field.getName()方法获取他的变量名称，如下例：

```
Field field = ... //获取Field对象
String fieldName = field.getName();
```

#

变量类型

你可以通过调用 Field.getType()方法来获取一个变量的类型（如 String, int 等等）

```
Field field = aClass.getField("someField");
Object fieldType = field.getType();
```

#

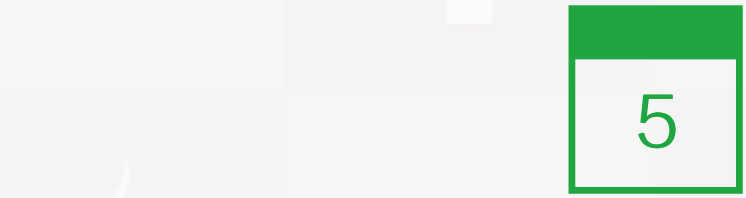
获取或设置（get/set）变量值

一旦你获得了一个 Field 的引用，你就可以通过调用 Field.get()或 Field.set()方法，获取或者设置变量的值，如下例：

```
Class  aClass = MyObject.class
Field field = aClass.getField("someField");

MyObject objectInstance = new MyObject();

Object value = field.get(objectInstance);

field.set(objetInstance, value);
```

传入 Field.get()/Field.set()方法的参数 objetInstance 应该是拥有指定变量的类的实例。在上述的例子中传入的参数是 MyObjec t类的实例，是因为 someField 是 MyObject 类的实例。 如果变量是静态变量的话(public static)那么在调用 Field.get()/Field.set()方法的时候传入 null 做为参数而不用传递拥有该变量的类的实例。(译者注：你如果传入拥有该变量的类的实例也可以得到相同的结果)

本文链接地址: Java Reflection(四):变量



5

Java 方法

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gamil.com ）

# #

内容索引

1. 获取 Method 对象
2. 方法参数以及返回类型
3. 通过 Method 对象调用方法

使用 Java 反射你可以在运行期检查一个方法的信息以及在运行期调用这个方法，通过使用 java.lang.reflect.Method 类就可以实现上述功能。在本节会带你深入了解 Method 对象的信息。

#

获取 Method 对象

可以通过 Class 对象获取 Method 对象，如下例：

```
Class aClass = ...//获取Class对象
Method[] methods = aClass.getMethods();
```

返回的 Method 对象数组包含了指定类中声明为公有的(public)的所有变量集合。

如果你知道你要调用方法的具体参数类型，你就可以直接通过参数类型来获取指定的方法，下面这个例子中返回方法对象名称是“doSomething”，他的方法参数是 String 类型：

```
Class  aClass = ...//获取Class对象
Method method = aClass.getMethod("doSomething", new Class[]{String.class});
```

如果根据给定的方法名称以及参数类型无法匹配到相应的方法，则会抛出 NoSuchMethodException。 如果你想要获取的方法没有参数，那么在调用 getMethod()方法时第二个参数传入 null 即可，就像这样：

```
Class  aClass = ...//获取Class对象
Method method = aClass.getMethod("doSomething", null);
```

#

方法参数以及返回类型

你可以获取指定方法的方法参数是哪些：

```
Method method = ... //获取Class对象
Class[] parameterTypes = method.getParameterTypes();
```

你可以获取指定方法的返回类型：

```
Method method = ... //获取Class对象
Class returnType = method.getReturnType();
```

#

通过 Method 对象调用方法

你可以通过如下方式来调用一个方法：

```
//获取一个方法名为doSomesthing，参数类型为String的方法
Method method = MyObject.class.getMethod("doSomething", String.class);
Object returnValue = method.invoke(null, "parameter-value1");
```

传入的 null 参数是你要调用方法的对象，如果是一个静态方法调用的话则可以用 null 代替指定对象作为 invoke()的参数，在上面这个例子中，如果 doSomething 不是静态方法的话，你就要传入有效的 MyObject 实例而不是 null。 Method.invoke(Object target, Object … parameters)方法的第二个参数是一个可变参数列表，但是你必须要传入与你要调用方法的形参一一对应的实参。就像上个例子那样，方法需要 String 类型的参数，那我们必须要传入一个字符串。

本文链接地址: Java Reflection(五):方法

6

Java 访问器

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gamil.com ）

使用 Java 反射你可以在运行期检查一个方法的信息以及在运行期调用这个方法，使用这个功能同样可以获取指定类的 getters 和 setters，你不能直接寻找 getters 和 setters，你需要检查一个类所有的方法来判断哪个方法是 getters 和 setters。

首先让我们来规定一下 getters 和 setters 的特性：

#

Getter

Getter 方法的名字以 get 开头，没有方法参数，返回一个值。

#

Setter

Setter 方法的名字以 set 开头，有一个方法参数。

setters 方法有可能会有返回值也有可能没有，一些 Setter 方法返回 void，一些用来设置值，有一些对象的 setter 方法在方法链中被调用（译者注：这类的 setter 方法必须要有返回值），因此你不应该妄自假设 setter 方法的返回值，一切应该视情况而定。

下面是一个获取 getter方法和 setter方法的例子：

查看源代码打印帮助

```
</pre>
<pre class="codeBox">public static void printGettersSetters(Class aClass){
  Method[] methods = aClass.getMethods();

  for(Method method : methods){
    if(isGetter(method)) System.out.println("getter: " + method);
    if(isSetter(method)) System.out.println("setter: " + method);
  }
}

public static boolean isGetter(Method method){
  if(!method.getName().startsWith("get"))      return false;
  if(method.getParameterTypes().length != 0)   return false;
  if(void.class.equals(method.getReturnType()) return false;
  return true;
}

public static boolean isSetter(Method method){
  if(!method.getName().startsWith("set")) return false;
  if(method.getParameterTypes().length != 1) return false;
  return true;
}</pre>
<pre>
```

本文链接地址: Java Reflection(六):Getters and Setters

7

Java 私有变量和私有方法

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gamil.com ）

# #

内容索引

1. 访问私有变量
2. 访问私有方法

在通常的观点中从对象的外部访问私有变量以及方法是不允许的，但是 Java 反射机制可以做到这一点。使用这个功能并不困难，在进行单元测试时这个功能非常有效。本节会向你展示如何使用这个功能。

注意：这个功能只有在代码运行在单机 Java 应用(standalone Java application)中才会有效,就像你做单元测试或者一些常规的应用程序一样。如果你在 Java Applet 中使用这个功能，那么你就要想办法去应付 SecurityManager 对你限制了。但是一般情况下我们是不会这么做的，所以在本节里面我们不会探讨这个问题。

#

访问私有变量

要想获取私有变量你可以调用 Class.getDeclaredField(String name)方法或者 Class.getDeclaredFields()方法。

Class.getField(String name)和 Class.getFields()只会返回公有的变量，无法获取私有变量。下面例子定义了一个包含私有变量的类，在它下面是如何通过反射获取私有变量的例子：

```
public class PrivateObject {

  private String privateString = null;

  public PrivateObject(String privateString) {
    this.privateString = privateString;
  }
}
```

```
PrivateObject privateObject = new PrivateObject("The Private Value");

Field privateStringField = PrivateObject.class.
          getDeclaredField("privateString");

privateStringField.setAccessible(true);

String fieldValue = (String) privateStringField.get(privateObject);
System.out.println("fieldValue = " + fieldValue);
```

这个例子会输出”fieldValue = The Private Value”，The Private Value 是 PrivateObject 实例的 privateString 私有变量的值，注意调用 PrivateObject.class.getDeclaredField(“privateString”)方法会返回一个私有变量，这个方法返回的变量是定义在 PrivateObject 类中的而不是在它的父类中定义的变量。 注意 privateStringField.setAccessible(true)这行代码，通过调用 setAccessible()方法会关闭指定类 Field 实例的反射访问检查，这行代码执行之后不论是私有的、受保护的以及包访问的作用域，你都可以在任何地方访问，即使你不在他的访问权限作用域之内。但是你如果你用一般代码来访问这些不在你权限作用域之内的代码依然是不可以的，在编译的时候就会报错。

#

访问私有方法

访问一个私有方法你需要调用 Class.getDeclaredMethod(String name, Class[] parameterTypes)或者 Class.getDeclaredMethods() 方法。 Class.getMethod(String name, Class[] parameterTypes)和 Class.getMethods()方法，只会返回公有的方法，无法获取私有方法。下面例子定义了一个包含私有方法的类，在它下面是如何通过反射获取私有方法的例子：

```
public class PrivateObject {

  private String privateString = null;

  public PrivateObject(String privateString) {
    this.privateString = privateString;
  }

  private String getPrivateString(){
    return this.privateString;
  }
}
```

```
PrivateObject privateObject = new PrivateObject("The Private Value");

Method privateStringMethod = PrivateObject.class.
        getDeclaredMethod("getPrivateString", null);

privateStringMethod.setAccessible(true);

String returnValue = (String)
        privateStringMethod.invoke(privateObject, null);

System.out.println("returnValue = " + returnValue);
```

这个例子会输出”returnValue = The Private Value”，The Private Value 是 PrivateObject 实例的 getPrivateString()方法的返回值。 PrivateObject.class.getDeclaredMethod(“privateString”)方法会返回一个私有方法，这个方法是定义在 PrivateObject 类中的而不是在它的父类中定义的。 同样的，注意 Method.setAccessible(true)这行代码，通过调用 setAccessible()方法会关闭指定类的 Method 实例的反射访问检查，这行代码执行之后不论是私有的、受保护的以及包访问的作用域，你都可以在任何地方访问，即使你不在他的访问权限作用域之内。但是你如果你用一般代码来访问这些不在你权限作用域之内的代码依然是不可以的，在编译的时候就会报错。

本文链接地址: Java Reflection(七):私有变量和私有方法

8

Java 注解

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gmail.com ）

# #

内容索引

1. 什么是注解
2. 类注解
3. 方法注解
4. 参数注解
5. 变量注解

利用 Java 反射机制可以在运行期获取 Java 类的注解信息。

#

什么是注解

注解是 Java 5 的一个新特性。注解是插入你代码中的一种注释或者说是一种元数据（meta data）。这些注解信息可以在编译期使用预编译工具进行处理（pre-compiler tools），也可以在运行期使用 Java 反射机制进行处理。下面是一个类注解的例子：

```
@MyAnnotation(name="someName",  value = "Hello World")
public class TheClass {
}
```

在 TheClass 类定义的上面有一个@MyAnnotation 的注解。注解的定义与接口的定义相似，下面是MyAnnotation注解的定义：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)

public @interface MyAnnotation {
  public String name();
  public String value();
}
```

在 interface 前面的@符号表名这是一个注解，一旦你定义了一个注解之后你就可以将其应用到你的代码中，就像之前我们的那个例子那样。 在注解定义中的两个指示@Retention(RetentionPolicy.RUNTIME)和@Target(ElementType.TYPE)，说明了这个注解该如何使用。 @Retention(RetentionPolicy.RUNTIME)表示这个注解可以在运行期通过反射访问。如果你没有在注解定义的时候使用这个指示那么这个注解的信息不会保留到运行期，这样反射就无法获取它的信息。 @Target(ElementType.TYPE) 表示这个注解只能用在类型上面（比如类跟接口）。你同样可以把Type改为Field或者Method，或者你可以不用这个指示，这样的话你的注解在类，方法和变量上就都可以使用了。 关于 Java 注解更详细的讲解可以访问 Java Annotations tutorial。

# #

类注解

你可以在运行期访问类，方法或者变量的注解信息，下是一个访问类注解的例子：

```
 Class aClass = TheClass.class;
Annotation[] annotations = aClass.getAnnotations();

for(Annotation annotation : annotations){
    if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("name: " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
    }
}
```

你还可以像下面这样指定访问一个类的注解：

```
Class aClass = TheClass.class;
Annotation annotation = aClass.getAnnotation(MyAnnotation.class);

if(annotation instanceof MyAnnotation){
    MyAnnotation myAnnotation = (MyAnnotation) annotation;
    System.out.println("name: " + myAnnotation.name());
    System.out.println("value: " + myAnnotation.value());
}
```

#

方法注解

下面是一个方法注解的例子：

```
public class TheClass {
  @MyAnnotation(name="someName",  value = "Hello World")
  public void doSomething(){}
}
```

你可以像这样访问方法注解：

```
Method method = ... //获取方法对象
Annotation[] annotations = method.getDeclaredAnnotations();

for(Annotation annotation : annotations){
    if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("name: " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
    }
}
```

你可以像这样访问指定的方法注解：

```
Method method = ... // 获取方法对象
Annotation annotation = method.getAnnotation(MyAnnotation.class);

if(annotation instanceof MyAnnotation){
    MyAnnotation myAnnotation = (MyAnnotation) annotation;
    System.out.println("name: " + myAnnotation.name());
    System.out.println("value: " + myAnnotation.value());
}
```

# #

参数注解

方法参数也可以添加注解，就像下面这样：

```
public class TheClass {
  public static void doSomethingElse(
      @MyAnnotation(name="aName", value="aValue") String parameter){
  }
}
```

你可以通过 Method对象来访问方法参数注解：

```
Method method = ... //获取方法对象
Annotation[][] parameterAnnotations = method.getParameterAnnotations();
Class[] parameterTypes = method.getParameterTypes();

int i=0;
for(Annotation[] annotations : parameterAnnotations){
  Class parameterType = parameterTypes[i++];

  for(Annotation annotation : annotations){
    if(annotation instanceof MyAnnotation){
      MyAnnotation myAnnotation = (MyAnnotation) annotation;
      System.out.println("param: " + parameterType.getName());
      System.out.println("name : " + myAnnotation.name());
      System.out.println("value: " + myAnnotation.value());
    }
  }
}
```

需要注意的是 Method.getParameterAnnotations()方法返回一个注解类型的二维数组，每一个方法的参数包含一个注解数组。

#

变量注解

下面是一个变量注解的例子：

```
public class TheClass {

  @MyAnnotation(name="someName",  value = "Hello World")
  public String myField = null;
}
```

你可以像这样来访问变量的注解：

```
Field field = ... //获取方法对象</pre>
<pre>Annotation[] annotations = field.getDeclaredAnnotations();

for(Annotation annotation : annotations){
 if(annotation instanceof MyAnnotation){
 MyAnnotation myAnnotation = (MyAnnotation) annotation;
 System.out.println("name: " + myAnnotation.name());
 System.out.println("value: " + myAnnotation.value());
 }
}
```

你可以像这样访问指定的变量注解：

```
Field field = ...//获取方法对象</pre>
<pre>
Annotation annotation = field.getAnnotation(MyAnnotation.class);

if(annotation instanceof MyAnnotation){
 MyAnnotation myAnnotation = (MyAnnotation) annotation;
 System.out.println("name: " + myAnnotation.name());
 System.out.println("value: " + myAnnotation.value());
}
```

本文链接地址: Java Reflection(八):注解

9

Java 泛型

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gmail.com ）

# #

内容索引

1. 运用泛型反射的经验法则
2. 泛型方法返回类型
3. 泛型方法参数类型
4. 泛型变量类型

我常常在一些文章以及论坛中读到说 Java 泛型信息在编译期被擦除（erased）所以你无法在运行期获得有关泛型的信息。其实这种说法并不完全正确的，在一些情况下是可以在运行期获取到泛型的信息。这些情况其实覆盖了一些我们需要泛型信息的需求。在本节中我们会演示一下这些情况。

#

运用泛型反射的经验法则

下面是两个典型的使用泛型的场景： 1、声明一个需要被参数化（parameterizable）的类/接口。 2、使用一个参数化类。

当你声明一个类或者接口的时候你可以指明这个类或接口可以被参数化， java.util.List 接口就是典型的例子。你可以运用泛型机制创建一个标明存储的是 String 类型 list，这样比你创建一个 Object 的l ist 要更好。

当你想在运行期参数化类型本身，比如你想检查 java.util.List 类的参数化类型，你是没有办法能知道他具体的参数化类型是什么。这样一来这个类型就可以是一个应用中所有的类型。但是，当你检查一个使用了被参数化的类型的变量或者方法，你可以获得这个被参数化类型的具体参数。总之：

你不能在运行期获知一个被参数化的类型的具体参数类型是什么，但是你可以在用到这个被参数化类型的方法以及变量中找到他们，换句话说就是获知他们具体的参数化类型。 在下面的段落中会向你演示这类情况。

#

泛型方法返回类型

如果你获得了 java.lang.reflect.Method 对象，那么你就可以获取到这个方法的泛型返回类型信息。如果方法是在一个被参数化类型之中（译者注：如 T fun()）那么你无法获取他的具体类型，但是如果方法返回一个泛型类（译者注：如 List fun()）那么你就可以获得这个泛型类的具体参数化类型。你可以在“Java Reflection: Methods”中阅读到有关如何获取Method对象的相关内容。下面这个例子定义了一个类这个类中的方法返回类型是一个泛型类型：

```
public class MyClass {

protected List<String> stringList = ...;

public List<String> getStringList(){
  return this.stringList;
}
}
```

我们可以获取 getStringList()方法的泛型返回类型，换句话说，我们可以检测到 getStringList()方法返回的是 List 而不仅仅只是一个 List。如下例：

```
Method method = MyClass.class.getMethod("getStringList", null);

Type returnType = method.getGenericReturnType();

if(returnType instanceof ParameterizedType){
    ParameterizedType type = (ParameterizedType) returnType;
    Type[] typeArguments = type.getActualTypeArguments();
    for(Type typeArgument : typeArguments){
        Class typeArgClass = (Class) typeArgument;
        System.out.println("typeArgClass = " + typeArgClass);
    }
}
```

这段代码会打印出 “typeArgClass = java.lang.String”，Type[]数组typeArguments 只有一个结果 - 一个代表 java.lang.String 的 Class 类的实例。Class 类实现了 Type 接口。

#

泛型方法参数类型

你同样可以通过反射来获取方法参数的泛型类型，下面这个例子定义了一个类，这个类中的方法的参数是一个被参数化的 List：

```
public class MyClass {
  protected List<String> stringList = ...;

  public void setStringList(List<String> list){
    this.stringList = list;
  }
}
```

你可以像这样来获取方法的泛型参数：

```
method = Myclass.class.getMethod("setStringList", List.class);

Type[] genericParameterTypes = method.getGenericParameterTypes();

for(Type genericParameterType : genericParameterTypes){
    if(genericParameterType instanceof ParameterizedType){
        ParameterizedType aType = (ParameterizedType) genericParameterType;
        Type[] parameterArgTypes = aType.getActualTypeArguments();
        for(Type parameterArgType : parameterArgTypes){
            Class parameterArgClass = (Class) parameterArgType;
            System.out.println("parameterArgClass = " + parameterArgClass);
        }
    }
}
```

这段代码会打印出”parameterArgType = java.lang.String”。Type[]数组 parameterArgTypes 只有一个结果 – 一个代表 java.lang.String 的 Class 类的实例。Class 类实现了Type接口。

#

泛型变量类型

同样可以通过反射来访问公有（Public）变量的泛型类型，无论这个变量是一个类的静态成员变量或是实例成员变量。你可以在“Java Reflection: Fields”中阅读到有关如何获取 Field 对象的相关内容。这是之前的一个例子，一个定义了一个名为 stringList 的成员变量的类。

```
public class MyClass {
  public List<String> stringList = ...;
}


Field field = MyClass.class.getField("stringList");

Type genericFieldType = field.getGenericType();

if(genericFieldType instanceof ParameterizedType){
    ParameterizedType aType = (ParameterizedType) genericFieldType;
    Type[] fieldArgTypes = aType.getActualTypeArguments();
    for(Type fieldArgType : fieldArgTypes){
        Class fieldArgClass = (Class) fieldArgType;
        System.out.println("fieldArgClass = " + fieldArgClass);
    }
}
```

这段代码会打印出”fieldArgClass = java.lang.String”。Type[]数组 fieldArgClass 只有一个结果 - 一个代表 java.lang.String 的 Class 类的实例。Class 类实现了 Type 接口。

本文链接地址: Java Reflection(九):泛型

10

Java 数组

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gmail.com ）

# #

内容索引

1. java.lang.reflect.Array
2. 创建一个数组
3. 访问一个数组
4. 获取数组的 Class 对象
5. 获取数组的成员类型

利用反射机制来处理数组会有点棘手。尤其是当你想要获得一个数组的 Class 对象，比如 int[] 等等。本节会讨论通过反射机制创建数组和如何获取数组的 Class 对象。

注意：在阅读 Eyal Lupu 的博客文章“Two Side Notes About Arrays and Reflection”之后对本文的内容做了更新。目前这个版本参考了这篇博文里面的内容。

#

java.lang.reflect.Array

Java 反射机制通过 java.lang.reflect.Array 这个类来处理数组。不要把这个类与 Java 集合套件（Collections suite）中的 java.util.Arrays 混淆， java.util.Arrays 是一个提供了遍历数组，将数组转化为集合等工具方法的类。

#

创建一个数组

Java 反射机制通过 java.lang.reflect.Array 类来创建数组。下面是一个如何创建数组的例子：

```
int[] intArray = (int[]) Array.newInstance(int.class, 3);
```

这个例子创建一个 int 类型的数组。Array.newInstance()方法的第一个参数表示了我们要创建一个什么类型的数组。第二个参数表示了这个数组的空间是多大。

#

访问一个数组

通过 Java 反射机制同样可以访问数组中的元素。具体可以使用Array.get(…)和Array.set(…)方法来访问。下面是一个例子：

```
int[] intArray = (int[]) Array.newInstance(int.class, 3);

Array.set(intArray, 0, 123);
Array.set(intArray, 1, 456);
Array.set(intArray, 2, 789);

System.out.println("intArray[0] = " + Array.get(intArray, 0));
System.out.println("intArray[1] = " + Array.get(intArray, 1));
System.out.println("intArray[2] = " + Array.get(intArray, 2));
```

这个例子会输出：

```
intArray[0] = 123
intArray[1] = 456
intArray[2] = 789
```

# #

获取数组的 Class 对象

在我编写 Butterfly DI Container 的脚本语言时，当我想通过反射获取数组的 Class 对象时遇到了一点麻烦。如果不通过反射的话你可以这样来获取数组的 Class 对象：

```
Class stringArrayClass = String[].class;
```

如果使用 Class.forName()方法来获取 Class 对象则不是那么简单。比如你可以像这样来获得一个原生数据类型（primitive）int 数组的 Class 对象：

```
Class intArray = Class.forName("[I");
```

在 JVM 中字母 I 代表 int 类型，左边的‘[’代表我想要的是一个int类型的数组，这个规则同样适用于其他的原生数据类型。 对于普通对象类型的数组有一点细微的不同：

```
Class stringArrayClass = Class.forName("[Ljava.lang.String;");
```

注意‘[L’的右边是类名，类名的右边是一个‘;’符号。这个的含义是一个指定类型的数组。 需要注意的是，你不能通过 Class.forName()方法获取一个原生数据类型的 Class 对象。下面这两个例子都会报 ClassNotFound Exception：

```
Class intClass1 = Class.forName("I");
Class intClass2 = Class.forName("int");
```

我通常会用下面这个方法来获取普通对象以及原生对象的 Class 对象：

```
public Class getClass(String className){
  if("int" .equals(className)) return int .class;
  if("long".equals(className)) return long.class;
  ...
  return Class.forName(className);
}
```

一旦你获取了类型的 Class 对象，你就有办法轻松的获取到它的数组的 Class 对象，你可以通过指定的类型创建一个空的数组，然后通过这个空的数组来获取数组的 Class 对象。这样做有点讨巧，不过很有效。如下例：

```
Class theClass = getClass(theClassName);
Class stringArrayClass = Array.newInstance(theClass, 0).getClass();
```

这是一个特别的方式来获取指定类型的指定数组的Class对象。无需使用类名或其他方式来获取这个 Class 对象。 为了确保 Class 对象是不是代表一个数组，你可以使用 Class.isArray()方法来进行校验：

```
Class stringArrayClass = Array.newInstance(String.class, 0).getClass();
System.out.println("is array: " + stringArrayClass.isArray());
```

#

获取数组的成员类型

一旦你获取了一个数组的 Class 对象，你就可以通过 Class.getComponentType()方法获取这个数组的成员类型。成员类型就是数组存储的数据类型。例如，数组 int[]的成员类型就是一个 Class 对象 int.class。String[]的成员类型就是 java.lang.String 类的 Class 对象。 下面是一个访问数组成员类型的例子：

```
String[] strings = new String[3];
Class stringArrayClass = strings.getClass();
Class stringArrayComponentType = stringArrayClass.getComponentType();
System.out.println(stringArrayComponentType);
```

下面这个例子会打印“java.lang.String”代表这个数组的成员类型是字符串。

本文链接地址: Java Reflection(十):数组

11

Java 动态代理

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gmail.com ）

# #

内容索引

1. 创建代理
2. InvocationHandler 接口

#

常见用例

1. 数据库连接以及事物管理
2. 单元测试中的动态 Mock 对象
3. 自定义工厂与依赖注入（DI）容器之间的适配器
4. 类似 AOP 的方法拦截器

利用Java反射机制你可以在运行期动态的创建接口的实现。 java.lang.reflect.Proxy 类就可以实现这一功能。这个类的名字（译者注：Proxy 意思为代理）就是为什么把动态接口实现叫做动态代理。动态的代理的用途十分广泛，比如数据库连接和事物管理（transaction management）还有单元测试时用到的动态 mock 对象以及 AOP 中的方法拦截功能等等都使用到了动态代理。

#

创建代理

你可以通过使用 Proxy.newProxyInstance()方法创建动态代理。 newProxyInstance()方法有三个参数：1、类加载器（ ClassLoader ）用来加载动态代理类。 2、一个要实现的接口的数组。 3、一个 InvocationHandler 把所有方法的调用都转到代理上。 如下例：

```
InvocationHandler handler = new MyInvocationHandler();
MyInterface proxy = (MyInterface) Proxy.newProxyInstance(
                    MyInterface.class.getClassLoader(),
                    new Class[] { MyInterface.class },
                    handler);
```

在执行完这段代码之后，变量 proxy 包含一个 MyInterface 接口的的动态实现。所有对 proxy 的调用都被转向到实现了 InvocationHandler 接口的 handler 上。有关 InvocationHandler 的内容会在下一段介绍。

#

InvocationHandler 接口

在前面提到了当你调用 Proxy.newProxyInstance()方法时，你必须要传入一个 InvocationHandler 接口的实现。所有对动态代理对象的方法调用都会被转向到 InvocationHandler 接口的实现上，下面是 InvocationHandler 接口的定义：

```
public interface InvocationHandler{
  Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

下面是它的实现类的定义：

```
public class MyInvocationHandler implements InvocationHandler{

  public Object invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    //do something "dynamic"
  }
}
```

传入 invoke()方法中的 proxy 参数是实现要代理接口的动态代理对象。通常你是不需要他的。

invoke()方法中的 Method 对象参数代表了被动态代理的接口中要调用的方法，从这个 method 对象中你可以获取到这个方法名字，方法的参数，参数类型等等信息。关于这部分内容可以查阅之前有关 Method 的文章。

Object 数组参数包含了被动态代理的方法需要的方法参数。注意：原生数据类型（如int，long等等）方法参数传入等价的包装对象（如Integer， Long等等）。

#

常见用例

动态代理常被应用到以下几种情况中

- 数据库连接以及事物管理
- 单元测试中的动态 Mock 对象
- 自定义工厂与依赖注入（DI）容器之间的适配器
- 类似 AOP 的方法拦截器

#

数据库连接以及事物管理

Spring 框架中有一个事物代理可以让你提交/回滚一个事物。它的具体原理在 Advanced Connection and Transaction Demarcation and Propagation 一文中有详细描述，所以在这里我就简短的描述一下，方法调用序列如下：

```
web controller --> proxy.execute(...);
  proxy --> connection.setAutoCommit(false);
  proxy --> realAction.execute();
    realAction does database work
  proxy --> connection.commit();
```

#

单元测试中的动态 Mock 对象

Butterfly Testing工具通过动态代理来动态实现桩（stub），mock 和代理类来进行单元测试。在测试类A的时候如果用到了接口 B，你可以传给 A 一个实现了 B 接口的 mock 来代替实际的 B 接口实现。所有对接口B的方法调用都会被记录，你可以自己来设置 B 的 mock 中方法的返回值。 而且 Butterfly Testing 工具可以让你在 B 的 mock 中包装真实的 B 接口实现，这样所有调用 mock 的方法都会被记录，然后把调用转向到真实的 B 接口实现。这样你就可以检查B中方法真实功能的调用情况。例如：你在测试 DAO 时你可以把真实的数据库连接包装到 mock 中。这样的话就与真实的情况一样，DAO 可以在数据库中读写数据，mock 会把对数据库的读写操作指令都传给数据库，你可以通过 mock 来检查 DAO 是不是以正确的方式来使用数据库连接，比如你可以检查是否调用了 connection.close()方法。这种情况是不能简单的依靠调用 DAO 方法的返回值来判断的。

#

自定义工厂与依赖注入（DI）容器之间的适配器

依赖注入容器 Butterfly Container 有一个非常强大的特性可以让你把整个容器注入到这个容器生成的 bean 中。但是，如果你不想依赖这个容器的接口，这个容器可以适配你自己定义的工厂接口。你仅仅需要这个接口而不是接口的实现，这样这个工厂接口和你的类看起来就像这样：

```
public interface IMyFactory {
  Bean   bean1();
  Person person();
  ...
}


public class MyAction{

  protected IMyFactory myFactory= null;

  public MyAction(IMyFactory factory){
    this.myFactory = factory;
  }

  public void execute(){
    Bean bean = this.myFactory.bean();
    Person person = this.myFactory.person();
  }

}
```

当 MyAction 类调用通过容器注入到构造方法中的 IMyFactory 实例的方法时，这个方法调用实际先调用了 IContainer.instance()方法，这个方法可以让你从容器中获取实例。这样这个对象可以把 Butterfly Container 容器在运行期当成一个工厂使用，比起在创建这个类的时候进行注入，这种方式显然更好。而且这种方法没有依赖到 Butterfly Container 中的任何接口。

#

类似 AOP 的方法拦截器

Spring 框架可以拦截指定 bean 的方法调用，你只需提供这个 bean 继承的接口。Spring 使用动态代理来包装 bean。所有对 bean 中方法的调用都会被代理拦截。代理可以判断在调用实际方法之前是否需要调用其他方法或者调用其他对象的方法，还可以在 bean 的方法调用完毕之后再调用其他的代理方法。

本文链接地址: Java Reflection(十一):动态代理

12

Java 动态类加载与重载

原文地址

作者: Jakob Jenkov 译者:叶文海（ yewenhai@gmail.com ）

# #

内容索引

1. 类加载器
2. 类加载体系
3. 类加载
4. 动态类加载
5. 动态类重载
6. 自定义类重载
7. 类加载/重载示例

Java 允许你在运行期动态加载和重载类，但是这个功能并没有像人们希望的那么简单直接。这篇文章将阐述在 Java 中如何加载以及重载类。 你可能会质疑为什么 Java 动态类加载特性是 Java 反射机制的一部分而不是 Java 核心平台的一部分。不管怎样，这篇文章被放到了 Java 反射系列里面而且也没有更好的系列来包含它了。

#

类加载器

所有 Java 应用中的类都是被 java.lang.ClassLoader 类的一系列子类加载的。因此要想动态加载类的话也必须使用 java.lang.ClassLoader 的子类。

一个类一旦被加载时，这个类引用的所有类也同时会被加载。类加载过程是一个递归的模式，所有相关的类都会被加载。但并不一定是一个应用里面所有类都会被加载，与这个被加载类的引用链无关的类是不会被加载的，直到有引用关系的时候它们才会被加载。

# #

类加载体系

在 Java 中类加载是一个有序的体系。当你新创建一个标准的 Java 类加载器时你必须提供它的父加载器。当一个类加载器被调用来加载一个类的时候，首先会调用这个加载器的父加载器来加载。如果父加载器无法找到这个类，这时候这个加载器才会尝试去加载这个类。

# #

类加载

类加载器加载类的顺序如下： 1、检查这个类是否已经被加载。 2、如果没有被加载，则首先调用父加载器加载。 3、如果父加载器不能加载这个类，则尝试加载这个类。

当你实现一个有重载类功能的类加载器，它的顺序与上述会有些不同。类重载不会请求的他的父加载器来进行加载。在后面的段落会进行讲解。

#

动态类加载

动态加载一个类十分简单。你要做的就是获取一个类加载器然后调用它的 loadClass()方法。下面是个例子：

```
public class MainClass {

  public static void main(String[] args){

    ClassLoader classLoader = MainClass.class.getClassLoader();

    try {
       Class aClass = classLoader.loadClass("com.jenkov.MyClass");
       System.out.println("aClass.getName() = " + aClass.getName());
    } catch (ClassNotFoundException e) {
       e.printStackTrace();
    }

}
```

#

动态类重载

动态类重载有一点复杂。Java 内置的类加载器在加载一个类之前会检查它是否已经被加载。因此重载一个类是无法使用 Java 内置的类加载器的，如果想要重载一个类你需要手动继承 ClassLoader。

在你定制 ClassLoader 的子类之后，你还有一些事需要做。所有被加载的类都需要被链接。这个过程是通过 ClassLoader.resolve()方法来完成的。由于这是一个 final 方法，因此这个方法在 ClassLoader 的子类中是无法被重写的。resolve()方法是不会允许给定的 ClassLoader 实例链接一个类两次。所以每当你想要重载一个类的时候你都需要使用一个新的 ClassLoader 的子类。你在设计类重载功能的时候这是必要的条件。

#

自定义类重载

在前面已经说过你不能使用已经加载过类的类加载器来重载一个类。因此你需要其他的 ClassLoader 实例来重载这个类。但是这又带来了一些新的挑战。

所有被加载到 Java 应用中的类都以类的全名（包名 + 类名）作为一个唯一标识来让 ClassLoader 实例来加载。这意味着，类 MyObject 被类加载器 A 加载，如果类加载器 B 又加载了 MyObject 类，那么两个加载器加载出来的类是不同的。看看下面的代码：

```
MyObject object = (MyObject)
   myClassReloadingFactory.newInstance("com.jenkov.MyObject");
```

MyObject 类在上面那段代码中被引用，它的变量名是 object。这就导致了 MyObject 这个类会被这段代码所在类的类加载器所加载。

如果 myClassReloadingFactory 工厂对象使用不同的类加载器重载 MyObject 类，你不能把重载的 MyObject类的实例转换（cast）到类型为 MyObject 的对象变量。一旦 MyObject 类分别被两个类加载器加载，那么它就会被认为是两个不同的类，尽管它们的类的全名是完全一样的。你如果尝试把这两个类的实例进行转换就会报 ClassCastException。 你可以解决这个限制，不过你需要从以下两个方面修改你的代码： 1、标记这个变量类型为一个接口，然后只重载这个接口的实现类。 2、标记这个变量类型为一个超类，然后只重载这个超类的子类。

请看下面这两个例子：

```
MyObjectInterface object = (MyObjectInterface)
   myClassReloadingFactory.newInstance("com.jenkov.MyObject");


MyObjectSuperclass object = (MyObjectSuperclass)
   myClassReloadingFactory.newInstance("com.jenkov.MyObject");
```

只要保证变量的类型是超类或者接口，这两个方法就可以正常运行，当它们的子类或是实现类被重载的时候超类跟接口是不会被重载的。

为了保证这种方式可以运行你需要手动实现类加载器然后使得这些接口或超类可以被它的父加载器加载。当你的类加载器加载 MyObject 类时，超类 MyObjectSuperclass 或者接口 MyObjectSuperclass 也会被加载，因为它们是 MyObject 的依赖。你的类加载器必须要代理这些类的加载到同一个类加载器，这个类加载器加载这个包括接口或者超类的类。

#

类加载/重载示例

光说不练假把式。让我们看看一个简单的例子。下面这个例子是一个类加载器的子类。注意在这个类不想被重载的情况下它是如何把对一个类的加载代理到它的父加载器上的。如果一个类被它的父加载器加载，这个类以后将不能被重载。记住，一个类只能被同一个 ClassLoade r实例加载一次。 就像我之前说的那样，这仅仅是一个简单的例子，通过这个例子会向你展示类加载器的基本行为。这并不是一个可以让你直接用于设计你项目中类加载器的模板。你自己设计的类加载器应该不仅仅只有一个，如果你想用来重载类的话你可能会设计很多加载器。并且你也不会像下面这样将需要加载的类的路径硬编码（ hardcore ）到你的代码中。

```
public class MyClassLoader extends ClassLoader{

   public MyClassLoader(ClassLoader parent) {
      super(parent);
   }

   public Class loadClass(String name) throws ClassNotFoundException {
      if(!"reflection.MyObject".equals(name))
            return super.loadClass(name);

      try {
         String url = "file:C:/data/projects/tutorials/web/WEB-INF/" +
                     "classes/reflection/MyObject.class";
         URL myUrl = new URL(url);
         URLConnection connection = myUrl.openConnection();
         InputStream input = connection.getInputStream();
         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
         int data = input.read();

         while(data != -1){
            buffer.write(data);
            data = input.read();
         }

         input.close();

         byte[] classData = buffer.toByteArray();

         return defineClass("reflection.MyObject",
               classData, 0, classData.length);
```

```
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }

}
```

下面是使用 MyClassLoader 的例子：

```
public static void main(String[] args) throws
    ClassNotFoundException,
    IllegalAccessException,
    InstantiationException {

    ClassLoader parentClassLoader = MyClassLoader.class.getClassLoader();
    MyClassLoader classLoader = new MyClassLoader(parentClassLoader);
    Class myObjectClass = classLoader.loadClass("reflection.MyObject");

    AnInterface2      object1 =
            (AnInterface2) myObjectClass.newInstance();

    MyObjectSuperClass object2 =
            (MyObjectSuperClass) myObjectClass.newInstance();

    //create new class loader so classes can be reloaded.
    classLoader = new MyClassLoader(parentClassLoader);
    myObjectClass = classLoader.loadClass("reflection.MyObject");

    object1 = (AnInterface2)      myObjectClass.newInstance();
    object2 = (MyObjectSuperClass) myObjectClass.newInstance();

}
```

下面这个就是被加载的 reflection.MyObject 类。注意它既继承了一个超类并且也实现了一个接口。这样做仅仅是为了通过例子演示这个特性。在你自定义的情况下你可能仅会实现一个类或者继承一两个接口。

```
public class MyObject extends MyObjectSuperClass implements AnInterface2{
    //... body of class ... override superclass methods
    //    or implement interface methods
}
```

本文链接地址: Java Reflection(十二):动态类加载与重载

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/java-reflection/