首先，我们在开始前提出一个问题：

1.在运行时，对于一个java类，能否知道属性和方法；能否去调用它的任意方法？ 答案是肯定的。

本节所有目录如下：

1. 什么是JAVA的反射机制
2. JDK中提供的Reflection API
3. JAVA反射机制提供了什么功能
   * 获取类的Class对象
   * 获取类的Fields
   * 获取类的Method
   * 获取类的Constructor
   * 新建类的实例  
            Class<T>的函数newInstance  
            通过Constructor对象的方法newInstance
4. 调用类的函数  
            调用private函数
5. 设置/获取类的属性值  
            private属性
6. 动态创建代理类  
            动态代理源码分析
7. JAVA反射Class<T>类型源代码分析
8. JAVA反射原理分析  
           Class文件结构  
           JVM加载类对象，对反射的支持
9. JAVA反射的应用

一、什么是JAVA的反射机制

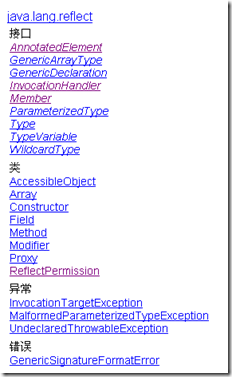
Java反射是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public, static 等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。

Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。

换言之，Java可以加载一个运行时才得知名称的class，获得其完整结构。

二、JDK中提供的Reflection API

Java反射相关的API在包[java.lang.reflect](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)中，JDK 1.6.0的reflect包如下图：

[](http://images.cnblogs.com/cnblogs_com/Quincy/201106/201106191021251045.png)

|  |  |
| --- | --- |
| Member接口 | 该接口可以获取有关类成员（域或者方法）后者构造函数的信息。 |
| AccessibleObject类 | 该类是域(field)对象、方法(method)对象、构造函数(constructor)对象的基础类。  它提供了将反射的对象标记为在使用时取消默认 Java 语言访问控制检查的能力。 |
| Array类 | 该类提供动态地生成和访问JAVA数组的方法。 |
| Constructor类 | 提供一个类的构造函数的信息以及访问类的构造函数的接口。 |
| Field类 | 提供一个类的域的信息以及访问类的域的接口。 |
| Method类 | 提供一个类的方法的信息以及访问类的方法的接口。 |
| Modifier类 | 提供了 static 方法和常量，对类和成员访问修饰符进行解码。 |
| Proxy类 | 提供动态地生成代理类和类实例的静态方法。 |

三、JAVA反射机制提供了什么功能

Java反射机制提供如下功能：

在运行时判断任意一个对象所属的类

在运行时构造任意一个类的对象

在运行时判段任意一个类所具有的成员变量和方法

在运行时调用任一个对象的方法

在运行时创建新类对象

在使用Java的反射功能时，基本首先都要获取类的Class对象，再通过Class对象获取其他的对象。

这里首先定义用于测试的类:

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "view plain) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "copy)

1. **class** Type{
2. **public** **int** pubIntField;
3. **public** String pubStringField;
4. **private** **int** prvIntField;
6. **public** Type(){
7. Log("Default Constructor");
8. }
10. Type(**int** arg1, String arg2){
11. pubIntField = arg1;
12. pubStringField = arg2;
14. Log("Constructor with parameters");
15. }
17. **public** **void** setIntField(**int** val) {
18. **this**.prvIntField = val;
19. }
20. **public** **int** getIntField() {
21. **return** prvIntField;
22. }
24. **private** **void** Log(String msg){
25. System.out.println("Type:" + msg);
26. }
27. }
29. **class** ExtendType **extends** Type{
30. **public** **int** pubIntExtendField;
31. **public** String pubStringExtendField;
32. **private** **int** prvIntExtendField;
34. **public** ExtendType(){
35. Log("Default Constructor");
36. }
38. ExtendType(**int** arg1, String arg2){
39. pubIntExtendField = arg1;
40. pubStringExtendField = arg2;
42. Log("Constructor with parameters");
43. }
45. **public** **void** setIntExtendField(**int** field7) {
46. **this**.prvIntExtendField = field7;
47. }
48. **public** **int** getIntExtendField() {
49. **return** prvIntExtendField;
50. }
52. **private** **void** Log(String msg){
53. System.out.println("ExtendType:" + msg);
54. }
55. }

1、获取类的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。获取类的Class对象有多种方式：

|  |  |
| --- | --- |
| 调用getClass | Boolean var1 = true;  Class<?> classType2 = var1.getClass();  System.out.println(classType2);  输出：class java.lang.Boolean |
| 运用.class 语法 | Class<?> classType4 = Boolean.class;  System.out.println(classType4);  输出：class java.lang.Boolean |
| 运用static method Class.forName() | Class<?> classType5 = Class.forName("java.lang.Boolean");  System.out.println(classType5);  输出：class java.lang.Boolean |
| 运用primitive wrapper classes的TYPE 语法  这里返回的是原生类型，和Boolean.class返回的不同 | Class<?> classType3 = Boolean.TYPE;  System.out.println(classType3);  输出：boolean |

2、获取类的Fields

可以通过反射机制得到某个类的某个属性，然后改变对应于这个类的某个实例的该属性值。JAVA 的Class<T>类提供了几个方法获取类的属性。

|  |  |
| --- | --- |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)getField([String](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank) 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 |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)[] 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 |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)getDeclaredField([String](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)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 |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)[] 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 |

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "view plain) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "copy)

1. Class<?> classType = ExtendType.**class**;
3. // 使用getFields获取属性
4. Field[] fields = classType.getFields();
5. **for** (Field f : fields)
6. {
7. System.out.println(f);
8. }
10. System.out.println();
12. // 使用getDeclaredFields获取属性
13. fields = classType.getDeclaredFields();
14. **for** (Field f : fields)
15. {
16. System.out.println(f);
17. }

输出：

public int com.quincy.ExtendType.pubIntExtendField

public java.lang.String com.quincy.ExtendType.pubStringExtendField

public int com.quincy.Type.pubIntField

public java.lang.String com.quincy.Type.pubStringField

public int com.quincy.ExtendType.pubIntExtendField

public java.lang.String com.quincy.ExtendType.pubStringExtendField

private int com.quincy.ExtendType.prvIntExtendField

可见getFields和getDeclaredFields区别：

getFields返回的是申明为public的属性，包括父类中定义，

getDeclaredFields返回的是指定类定义的所有定义的属性，不包括父类的。

3、获取类的Method

通过反射机制得到某个类的某个方法，然后调用对应于这个类的某个实例的该方法

Class<T>类提供了几个方法获取类的方法。

|  |  |
| --- | --- |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)getMethod([String](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank) name,[Class](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<?>... parameterTypes) | 返回一个 Method 对象，  它反映此 Class 对象所表示的类或接口的指定公共成员方法 |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)[] getMethods() | 返回一个包含某些 Method 对象的数组，  这些对象反映此 Class 对象所表示的类或接口  （包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）  的公共 member 方法 |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)getDeclaredMethod([String](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)name,[Class](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<?>... parameterTypes) | 返回一个 Method 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方法 |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)[] getDeclaredMethods() | 返回 Method 对象的一个数组，  这些对象反映此 Class 对象表示的类或接口声明的所有方法，  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 |

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "view plain) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "copy)

1. // 使用getMethods获取函数
2. Class<?> classType = ExtendType.**class**;
3. Method[] methods = classType.getMethods();
4. **for** (Method m : methods)
5. {
6. System.out.println(m);
7. }
9. System.out.println();
11. // 使用getDeclaredMethods获取函数
12. methods = classType.getDeclaredMethods();
13. **for** (Method m : methods)
14. {
15. System.out.println(m);
16. }

输出：

public void com.quincy.ExtendType.setIntExtendField(int)

public int com.quincy.ExtendType.getIntExtendField()

public void com.quincy.Type.setIntField(int)

public int com.quincy.Type.getIntField()

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

private void com.quincy.ExtendType.Log(java.lang.String)

public void com.quincy.ExtendType.setIntExtendField(int)

public int com.quincy.ExtendType.getIntExtendField()

4、获取类的Constructor

通过反射机制得到某个类的构造器，然后调用该构造器创建该类的一个实例

Class<T>类提供了几个方法获取类的构造器。

|  |  |
| --- | --- |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<[T](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)> getConstructor([Class](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<?>... parameterTypes) | 返回一个 Constructor 对象，  它反映此 Class 对象所表示的类的指定公共构造方法 |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<?>[] getConstructors() | 返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造方法 |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<[T](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)> getDeclaredConstructor([Class](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<?>... parameterTypes) | 返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法 |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)<?>[] getDeclaredConstructors() | 返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。它们是公共、保护、默认（包）访问和私有构造方法 |

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "view plain) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "copy)

1. // 使用getConstructors获取构造器
2. Constructor<?>[] constructors = classType.getConstructors();
3. **for** (Constructor<?> m : constructors)
4. {
5. System.out.println(m);
6. }
8. System.out.println();
10. // 使用getDeclaredConstructors获取构造器
11. constructors = classType.getDeclaredConstructors();
12. **for** (Constructor<?> m : constructors)
13. {
14. System.out.println(m);
15. }
17. 输出：
18. **public** com.quincy.ExtendType()
20. **public** com.quincy.ExtendType()
21. com.quincy.ExtendType(**int**,java.lang.String)

5、新建类的实例

通过反射机制创建新类的实例，有几种方法可以创建

|  |  |
| --- | --- |
| 调用无自变量ctor | 1、调用类的Class对象的newInstance方法，  该方法会调用对象的默认构造器，如果没有默认构造器，会调用失败.  Class<?> classType = ExtendType.class;  Object inst = classType.newInstance();  System.out.println(inst);  输出：  Type:Default Constructor  ExtendType:Default Constructor  [com.quincy.ExtendType@d80be3](mailto:com.quincy.ExtendType@d80be3" \t "http://blog.csdn.net/yongjian1092/article/details/_blank)    2、调用默认Constructor对象的newInstance方法  Class<?> classType = ExtendType.class;  Constructor<?> constructor1 = classType.getConstructor();  Object inst = constructor1.newInstance();  System.out.println(inst);  输出：  Type:Default Constructor  ExtendType:Default Constructor  com.quincy.ExtendType@1006d75 |
| 调用带参数ctor | 3、调用带参数Constructor对象的newInstance方法  Constructor<?> constructor2 =  classType.getDeclaredConstructor(int.class, String.class);  Object inst = constructor2.newInstance(1, "123");  System.out.println(inst);  输出：  Type:Default Constructor  ExtendType:Constructor with parameters  com.quincy.ExtendType@15e83f9 |

6、调用类的函数

通过反射获取类Method对象，调用Field的Invoke方法调用函数。

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "view plain) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "copy)

1. Class<?> classType = ExtendType.**class**;
2. Object inst = classType.newInstance();
3. Method logMethod = classType.<strong>getDeclaredMethod</strong>("Log", String.**class**);
4. logMethod.invoke(inst, "test");
6. 输出：
7. Type:Default Constructor
8. ExtendType:Default Constructor
9. <font color="#ff0000">Class com.quincy.ClassT can not access a member of **class** com.quincy.ExtendType with modifiers "private"</font>
11. <font color="#ff0000">上面失败是由于没有权限调用**private**函数，这里需要设置Accessible为**true**;</font>
12. Class<?> classType = ExtendType.**class**;
13. Object inst = classType.newInstance();
14. Method logMethod = classType.getDeclaredMethod("Log", String.**class**);
15. <font color="#ff0000">logMethod.setAccessible(**true**);</font>
16. logMethod.invoke(inst, "test");

7、设置/获取类的属性值

通过反射获取类的Field对象，调用Field方法设置或获取值

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "view plain) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451" \o "copy)

1. Class<?> classType = ExtendType.**class**;
2. Object inst = classType.newInstance();
3. Field intField = classType.getField("pubIntExtendField");
4. intField.<strong>setInt</strong>(inst, 100);
5. **int** value = intField.<strong>getInt</strong>(inst);

四、动态创建代理类

代理模式：代理模式的作用=为其他对象提供一种代理以控制对这个对象的访问。

代理模式的角色：

抽象角色：声明真实对象和代理对象的共同接口

代理角色：代理角色内部包含有真实对象的引用，从而可以操作真实对象。

真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

动态代理：

|  |  |
| --- | --- |
| java.lang.reflect.Proxy | Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类 |
| InvocationHandler | 是代理实例的调用处理程序 实现的接口，每个代理实例都具有一个关联的调用处理程序。对代理实例调用方法时，将对方法调用进行编码并将其指派到它的调用处理程序的 invoke 方法。 |

动态Proxy是这样的一种类:

它是在运行生成的类，在生成时你必须提供一组Interface给它，然后该class就宣称它实现了这些interface。你可以把该class的实例当作这些interface中的任何一个来用。当然，这个Dynamic Proxy其实就是一个Proxy，它不会替你作实质性的工作，在生成它的实例时你必须提供一个handler，由它接管实际的工作。

在使用动态代理类时，我们必须实现InvocationHandler接口

步骤：

1、定义抽象角色

public interface Subject {

public void Request();

}

2、定义真实角色

public class RealSubject implements Subject {

@Override

public void Request() {

// TODO Auto-generated method stub

System.out.println("RealSubject");

}

}

3、定义代理角色

public class DynamicSubject implements InvocationHandler {

private Object sub;

public DynamicSubject(Object obj){

this.sub = obj;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

// TODO Auto-generated method stub

System.out.println("Method:"+ method + ",Args:" + args);

method.invoke(sub, args);

return null;

}

}

4、通过Proxy.newProxyInstance构建代理对象

RealSubject realSub = new RealSubject();

InvocationHandler handler = new DynamicSubject(realSub);

Class<?> classType = handler.getClass();

Subject sub = (Subject)Proxy.newProxyInstance(classType.getClassLoader(),

realSub.getClass().getInterfaces(), handler);

System.out.println(sub.getClass());

5、通过调用代理对象的方法去调用真实角色的方法。

sub.Request();

输出：

class $Proxy0 新建的代理对象，它实现指定的接口

Method:public abstract void DynamicProxy.Subject.Request(),Args:null

RealSubject 调用的真实对象的方法