题目：java 的反射机制

问题：

在运行时，对一个 JAVA 类，能否知道属性和方法；能否调用它的任意方法？

答案是可以的，JAVA 提供一种反射机制可以实现。

目录

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 中，JDK 1.6.0的 reflect 包如下图：

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：

View Code?

```
1  classType{
2      publicintpubIntField;
3      publicString pubStringField;
4      privateintprvIntField;
5
6      publicType(){
7          Log("Default Constructor");
8      }
9
10     Type(intarg1, String arg2){
11         pubIntField = arg1;
12         pubStringField = arg2;
13
14         Log("Constructor with parameters");
15     }
16
17     publicvoidsetIntField(intval) {
18         this.prvIntField = val;
19     }
20     publicintgetIntField() {
21         returnprvIntField;
22     }
23
24     privatevoidLog(String msg){
25         System.out.println("Type:"+ msg);
26     }
27 }
28
```

```
29 classExtendTypeextendsType{
30     publicintpubIntExtendField;
31     publicString pubStringExtendField;
32     privateintprvIntExtendField;
33 
34     publicExtendType(){
35         Log("Default Constructor");
36     }
37 
38     ExtendType(intarg1, String arg2){
39         pubIntExtendField = arg1;
40         pubStringExtendField = arg2;
41 
42         Log("Constructor with parameters");
43     }
44 
45     publicvoidsetIntExtendField(intfield7) {
46         this.prvIntExtendField = field7;
47     }
48     publicintgetIntExtendField() {
49         returnprvIntExtendField;
50     }
51 
52     privatevoidLog(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 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
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
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
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

View Code?

```
Class<?> classType = ExtendType.class;

// 使用 getFields 获取属性
Field[] fields = classType.getFields();
for(Field f : fields)
{
    System.out.println(f);
}

System.out.println();

// 使用 getDeclaredFields 获取属性
fields = classType.getDeclaredFields();
for(Field f : fields)
{
    System.out.println(f);
}
```

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 getMethod(String name, Class<?>... 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[] 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 getDeclaredMethod(String name,Class<?>... 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[] 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

View Code?

```
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  }
8
9  System.out.println();
```

```
10
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<T>	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

getConstructor(Class<?>... parameterTypes)	
public Constructor<?>[] 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<T> getDeclaredConstructor(Class<?>... 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<?>[] 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

View Code?

```
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  }
7
8  System.out.println();
9
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 }
16
17 输出：
18 publiccom.quincy.ExtendType()
19
20 publiccom.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
```

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。

View Code?

```
Class<?> classType = ExtendType.class;
Object inst = classType.newInstance();
Method logMethod =
classType.<STRONG>getDeclaredMethod</STRONG>("Log",
String.class);
logMethod.invoke(inst,"test");

输出：
Type:Default Constructor
ExtendType:Default Constructor
<FONT color=#ff0000>Class com.quincy.ClassT can not access a member
ofclasscom.quincy.ExtendType with modifiers"private"</FONT>

<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>
Class<?> classType = ExtendType.class;
Object inst = classType.newInstance();
Method logMethod = classType.getDeclaredMethod("Log",
String.class);
```

```
<FONT color=#ff0000>logMethod.setAccessible(true);</FONT>
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

View Code?

```
Class<?> classType = ExtendType.class;
Object inst = classType.newInstance();
Field intField = classType.getField("pubIntExtendField");
intField.<STRONG>setInt</STRONG>(inst,100);
    intvalue = intField.<STRONG>getInt</STRONG>(inst);
```

View Code?

```
1
```

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
==========================================================
```

本篇文章依旧采用小例子来说明，因为我始终觉的，案例驱动是最好的，要不然只看理论的话，看了也不懂，不过建议大家在看完文章之后，在回过头去看看理论，会有更好的理解。

下面开始正文。

【案例1】通过一个对象获得完整的包名和类名

```
?
    packageReflect;
1
2
    /**
3
     * 通过一个对象获得完整的包名和类名
4
     * */
5
    classDemo{
6
        //other codes...
7
    }
8
9
    classhello{
10
        publicstaticvoidmain(String[] args) {
11
            Demo demo=newDemo();
12
            System.out.println(demo.getClass().getName());
13
        }
14
    }
15
```

【运行结果】：Reflect.Demo

添加一句：所有类的对象其实都是 Class 的实例。

【案例2】实例化 Class 类对象

```
?

   packageReflect;
1
   classDemo{
2
       //other codes...
3
   }
4

5
   classhello{
6
       publicstaticvoidmain(String[] args) {
7
           Class<?> demo1=null;
8
           Class<?> demo2=null;
9
           Class<?> demo3=null;
10
           try{
11
               //一般尽量采用这种形式
12
               demo1=Class.forName("Reflect.Demo");
13
           }catch(Exception e){
14
               e.printStackTrace();
15
           }
16
           demo2=newDemo().getClass();
17
           demo3=Demo.class;
18

19
           System.out.println("类名称   "+demo1.getName());
20
           System.out.println("类名称   "+demo2.getName());
21
           System.out.println("类名称   "+demo3.getName());
22

23
       }
24
   }
25
```

【运行结果】：

类名称   Reflect.Demo

类名称　Reflect.Demo

类名称　Reflect.Demo

【案例3】通过 Class 实例化其他类的对象

通过无参构造实例化对象

```
?
1  packageReflect;
2
3  classPerson{
4
5      publicString getName() {
6          returnname;
7      }
8      publicvoidsetName(String name) {
9          this.name = name;
10     }
11     publicintgetAge() {
12         returnage;
13     }
14     publicvoidsetAge(intage) {
15         this.age = age;
16     }
17     @Override
18     publicString toString(){
19         return"["+this.name+"  "+this.age+"]";
20     }
21     privateString name;
22     privateintage;
23 }
24
25 classhello{
26     publicstaticvoidmain(String[] args) {
```

```
27          Class<?> demo=null;
28          try{
29              demo=Class.forName("Reflect.Person");
30          }catch(Exception e) {
31              e.printStackTrace();
32          }
33          Person per=null;
34          try{
35              per=(Person)demo.newInstance();
36          }catch(InstantiationException e) {
37              // TODO Auto-generated catch block
38              e.printStackTrace();
39          }catch(IllegalAccessException e) {
40              // TODO Auto-generated catch block
41              e.printStackTrace();
42          }
43          per.setName("Rollen");
44          per.setAge(20);
45          System.out.println(per);
46      }
47  }
```

【运行结果】:

```
[Rollen  20]
```

但是注意一下，当我们把 Person 中的默认的无参构造函数取消的时候，比如自己定义只定义一个有参数的构造函数之后，会出现错误：

比如我定义了一个构造函数：

```
?
1  publicPerson(String name,intage) {
2          this.age=age;
3          this.name=name;
```

```
4      }
```

然后继续运行上面的程序，会出现：

```
java.lang.InstantiationException: Reflect.Person
    at java.lang.Class.newInstance0(Class.java:340)
    at java.lang.Class.newInstance(Class.java:308)
    at Reflect.hello.main(hello.java:39)
Exception in thread "main"java.lang.NullPointerException
    at Reflect.hello.main(hello.java:47)
```

所以大家以后再编写使用 Class 实例化其他类的对象的时候，一定要自己定义无参的构造函数

【案例】通过 Class 调用其他类中的构造函数 （也可以通过这种方式通过 Class 创建其他类的对象）

```
?
1  packageReflect;
2
3  importjava.lang.reflect.Constructor;
4
5  classPerson{
6
7      publicPerson() {
8
9      }
10     publicPerson(String name){
11         this.name=name;
12     }
13     publicPerson(intage){
14         this.age=age;
15     }
16     publicPerson(String name,intage) {
```

```
17          this.age=age;
18          this.name=name;
19      }
20      publicString getName() {
21          returnname;
22      }
23      publicintgetAge() {
24          returnage;
25      }
26      @Override
27      publicString toString(){
28          return"["+this.name+"  "+this.age+"]";
29      }
30      privateString name;
31      privateintage;
32  }
33
34  classhello{
35      publicstaticvoidmain(String[] args) {
36          Class<?> demo=null;
37          try{
38              demo=Class.forName("Reflect.Person");
39          }catch(Exception e) {
40              e.printStackTrace();
41          }
42          Person per1=null;
43          Person per2=null;
44          Person per3=null;
45          Person per4=null;
46          //取得全部的构造函数
47          Constructor<?> cons[]=demo.getConstructors();
48          try{
```

```
49              per1=(Person)cons[0].newInstance();
50              per2=(Person)cons[1].newInstance("Rollen");
51              per3=(Person)cons[2].newInstance(20);
52              per4=(Person)cons[3].newInstance("Rollen",20);
53          }catch(Exception e){
54              e.printStackTrace();
55          }
56          System.out.println(per1);
57          System.out.println(per2);
58          System.out.println(per3);
59          System.out.println(per4);
60      }
61  }
```

【运行结果】:

```
[null  0]
[Rollen  0]
[null  20]
[Rollen  20]
```

【案例】

返回一个类实现的接口：

```
?
1  packageReflect;
2
3  interfaceChina{
4      publicstaticfinalString name="Rollen";
5      publicstatic  int age=20;
6      publicvoidsayChina();
7      publicvoidsayHello(String name,intage);
8  }
```

```
9
10 classPersonimplementsChina{
11     publicPerson() {
12
13     }
14     publicPerson(String sex){
15         this.sex=sex;
16     }
17     publicString getSex() {
18         returnsex;
19     }
20     publicvoidsetSex(String sex) {
21         this.sex = sex;
22     }
23     @Override
24     publicvoidsayChina(){
25         System.out.println("hello ,china");
26     }
27     @Override
28     publicvoidsayHello(String name,intage){
29         System.out.println(name+"  "+age);
30     }
31     privateString sex;
32 }
33
34 classhello{
35     publicstaticvoidmain(String[] args) {
36         Class<?> demo=null;
37         try{
38             demo=Class.forName("Reflect.Person");
39         }catch(Exception e) {
40             e.printStackTrace();
```

```
41          }
42          //保存所有的接口
43          Class<?> intes[]=demo.getInterfaces();
44          for(inti =0; i < intes.length; i++) {
45              System.out.println("实现的接口    "+intes[i].getName());
46          }
47      }
48  }
```

【运行结果】:

实现的接口 Reflect.China

（注意，以下几个例子，都会用到这个例子的 Person 类，所以为节省篇幅，此处不再粘贴 Person 的代码部分，只粘贴主类 hello 的代码）

【案例】: 取得其他类中的父类

```
?
   classhello{
1      publicstaticvoidmain(String[] args) {
2          Class<?> demo=null;
3          try{
4              demo=Class.forName("Reflect.Person");
5          }catch(Exception e) {
6              e.printStackTrace();
7          }
8          //取得父类
9          Class<?> temp=demo.getSuperclass();
10         System.out.println("继承的父类为:    "+temp.getName());
11     }
12 }
13
```

【运行结果】

继承的父类为： java.lang.Object

【案例】：获得其他类中的全部构造函数

这个例子需要在程序开头添加 import java.lang.reflect.*;

然后将主类编写为：

```
?
   classhello{
1
       publicstaticvoidmain(String[] args) {
2
           Class<?> demo=null;
3
           try{
4
               demo=Class.forName("Reflect.Person");
5
           }catch(Exception e) {
6
               e.printStackTrace();
7
           }
8
           Constructor<?>cons[]=demo.getConstructors();
9
           for(inti =0; i < cons.length; i++) {
10
               System.out.println("构造方法：  "+cons[i]);
11
           }
12
       }
13
   }
14
```

【运行结果】：

构造方法： public Reflect.Person()

构造方法： public Reflect.Person(java.lang.String)

但是细心的读者会发现，上面的构造函数没有 public 或者 private 这一类的修饰符

下面这个例子我们就来获取修饰符

```
?
1  classhello{
```

```
2      publicstaticvoidmain(String[] args) {
3          Class<?> demo=null;
4          try{
5              demo=Class.forName("Reflect.Person");
6          }catch(Exception e) {
7              e.printStackTrace();
8          }
9          Constructor<?>cons[]=demo.getConstructors();
10         for(inti =0; i < cons.length; i++) {
11             Class<?> p[]=cons[i].getParameterTypes();
12             System.out.print("构造方法:   ");
13             intmo=cons[i].getModifiers();
14             System.out.print(Modifier.toString(mo)+" ");
15             System.out.print(cons[i].getName());
16             System.out.print("(");
17             for(intj=0;j<p.length;++j){
18                 System.out.print(p[j].getName()+" arg"+i);
19                 if(j<p.length-1){
20                     System.out.print(",");
21                 }
22             }
23             System.out.println("){}");
24         }
25     }
26 }
```

【运行结果】:

```
构造方法:   public Reflect.Person(){}
构造方法:   public Reflect.Person(java.lang.String  arg1){}
```

有时候一个方法可能还有异常，呵呵。下面看看：

```
?
```

```
1  classhello{
2      publicstaticvoidmain(String[] args) {
3          Class<?> demo=null;
4          try{
5              demo=Class.forName("Reflect.Person");
6          }catch(Exception e) {
7              e.printStackTrace();
8          }
9          Method method[]=demo.getMethods();
10         for(inti=0;i<method.length;++i){
11             Class<?> returnType=method[i].getReturnType();
12             Class<?> para[]=method[i].getParameterTypes();
13             inttemp=method[i].getModifiers();
14             System.out.print(Modifier.toString(temp)+" ");
15             System.out.print(returnType.getName()+"  ");
16             System.out.print(method[i].getName()+" ");
17             System.out.print("(");
18             for(intj=0;j<para.length;++j){
19                 System.out.print(para[j].getName()+" "+"arg"+j);
20                 if(j<para.length-1){
21                     System.out.print(",");
22                 }
23             }
24             Class<?> exce[]=method[i].getExceptionTypes();
25             if(exce.length>0){
26                 System.out.print(") throws ");
27                 for(intk=0;k<exce.length;++k){
28                     System.out.print(exce[k].getName()+" ");
29                     if(k<exce.length-1){
30                         System.out.print(",");
31                     }
32                 }
```

```
33            }else{
34                System.out.print(")");
35            }
36            System.out.println();
37        }
38    }
39 }
```

【运行结果】:

```
public java.lang.String  getSex ()
public void  setSex (java.lang.String arg0)
public void  sayChina ()
public void  sayHello (java.lang.String arg0,int arg1)
public final native void  wait (long arg0) throwsjava.lang.InterruptedException
public final void  wait () throwsjava.lang.InterruptedException
public final void  wait (long arg0,int arg1)
throwsjava.lang.InterruptedException
public boolean  equals (java.lang.Object arg0)
public java.lang.String  toString ()
public native int  hashCode ()
public final native java.lang.Class  getClass ()
public final native void  notify ()
public final native void  notifyAll ()
```

【案例】接下来让我们取得其他类的全部属性吧，最后我讲这些整理在一起，也就是通过 class 取得一个类的全部框架

?

```
1  classhello {
2      publicstaticvoidmain(String[] args) {
3          Class<?> demo =null;
4          try{
5              demo = Class.forName("Reflect.Person");
6          }catch(Exception e) {
7              e.printStackTrace();
8          }
9          System.out.println("===============     本     类     属     性
10 ========================");
11         // 取得本类的全部属性
12         Field[] field = demo.getDeclaredFields();
13         for(inti =0; i < field.length; i++) {
14             // 权限修饰符
15             intmo = field[i].getModifiers();
16             String priv = Modifier.toString(mo);
17             // 属性类型
18             Class<?> type = field[i].getType();
19             System.out.println(priv +" "+ type.getName() +" "
20                    + field[i].getName() +";");
21         }
22         System.out.println("=============== 实 现 的 接 口 或 者 父 类 的 属 性
23 ========================");
24         // 取得实现的接口或者父类的属性
25         Field[] filed1 = demo.getFields();
26         for(intj =0; j < filed1.length; j++) {
27             // 权限修饰符
28             intmo = filed1[j].getModifiers();
29             String priv = Modifier.toString(mo);
30             // 属性类型
31             Class<?> type = filed1[j].getType();
32             System.out.println(priv +" "+ type.getName() +" "
```

```
33                  + filed1[j].getName() +";");
34          }
        }
    }
```

【运行结果】:

```
==============本类属性=======================
private java.lang.String sex;
==============实现的接口或者父类的属性=======================
public static final java.lang.String name;
public static final int age;
```

【案例】其实还可以通过反射调用其他类中的方法:

```
?
1  classhello {
2      publicstaticvoidmain(String[] args) {
3          Class<?> demo =null;
4          try{
5              demo = Class.forName("Reflect.Person");
6          }catch(Exception e) {
7              e.printStackTrace();
8          }
9          try{
10             //调用 Person 类中的 sayChina 方法
11             Method method=demo.getMethod("sayChina");
12             method.invoke(demo.newInstance());
13             //调用 Person 的 sayHello 方法
14             method=demo.getMethod("sayHello", String.class,int.class);
```

```
15            method.invoke(demo.newInstance(),"Rollen",20);
16
17        }catch(Exception e) {
18            e.printStackTrace();
19        }
20    }
21 }
```

【运行结果】:

```
hello ,china
Rollen  20
```

【案例】调用其他类的 set 和 get 方法

```
?
1  classhello {
2      publicstaticvoidmain(String[] args) {
3          Class<?> demo =null;
4          Object obj=null;
5          try{
6              demo = Class.forName("Reflect.Person");
7          }catch(Exception e) {
8              e.printStackTrace();
9          }
10         try{
11          obj=demo.newInstance();
12         }catch(Exception e) {
13             e.printStackTrace();
14         }
15         setter(obj,"Sex","男",String.class);
16         getter(obj,"Sex");
```

```
17      }
18
19      /**
20       * @param obj
21       *            操作的对象
22       * @param att
23       *            操作的属性
24       * */
25      publicstaticvoidgetter(Object obj, String att) {
26          try{
27              Method method = obj.getClass().getMethod("get"+ att);
28              System.out.println(method.invoke(obj));
29          }catch(Exception e) {
30              e.printStackTrace();
31          }
32      }
33
34      /**
35       * @param obj
36       *            操作的对象
37       * @param att
38       *            操作的属性
39       * @param value
40       *            设置的值
41       * @param type
42       *            参数的属性
43       * */
44      publicstaticvoidsetter(Object obj, String att, Object value,
45              Class<?> type) {
46          try{
47              Method method = obj.getClass().getMethod("set"+ att, type);
48              method.invoke(obj, value);
```

```
49          }catch(Exception e) {
50              e.printStackTrace();
51          }
52      }
53  }// end class
```

【运行结果】:

男

【案例】通过反射操作属性

```
?
   classhello {
1
       publicstaticvoidmain(String[] args)throwsException {
2
           Class<?> demo =null;
3
           Object obj =null;
4
5
           demo = Class.forName("Reflect.Person");
6
           obj = demo.newInstance();
7
8
           Field field = demo.getDeclaredField("sex");
9
           field.setAccessible(true);
10
           field.set(obj,"男");
11
           System.out.println(field.get(obj));
12
       }
13
   }// end class
14
```

【案例】通过反射取得并修改数组的信息:

```
?
   importjava.lang.reflect.*;
1  classhello{
2      publicstaticvoidmain(String[] args) {
3          int[] temp={1,2,3,4,5};
4          Class<?>demo=temp.getClass().getComponentType();
5          System.out.println("数组类型： "+demo.getName());
6          System.out.println("数组长度  "+Array.getLength(temp));
7          System.out.println("数组的第一个元素： "+Array.get(temp,0));
8          Array.set(temp,0,100);
9          System.out.println(" 修 改 之 后 数 组 第 一 个 元 素 为 ：
10 "+Array.get(temp,0));
11     }
12 }
```

【运行结果】:

数组类型： int

数组长度 5

数组的第一个元素：1

修改之后数组第一个元素为： 100

【案例】通过反射修改数组大小

```
?
1  classhello{
2      publicstaticvoidmain(String[] args) {
3          int[] temp={1,2,3,4,5,6,7,8,9};
```

```
4          int[] newTemp=(int[])arrayInc(temp,15);
5          print(newTemp);
6          System.out.println("====================");
7          String[] atr={"a","b","c"};
8          String[] str1=(String[])arrayInc(atr,8);
9          print(str1);
10     }
11
12     /**
13      * 修改数组大小
14      * */
15     publicstaticObject arrayInc(Object obj,intlen){
16         Class<?>arr=obj.getClass().getComponentType();
17         Object newArr=Array.newInstance(arr, len);
18         intco=Array.getLength(obj);
19         System.arraycopy(obj,0, newArr,0, co);
20         returnnewArr;
21     }
22     /**
23      * 打印
24      * */
25     publicstaticvoidprint(Object obj){
26         Class<?>c=obj.getClass();
27         if(!c.isArray()){
28             return;
29         }
30         System.out.println("数组长度为:  "+Array.getLength(obj));
31         for(inti =0; i < Array.getLength(obj); i++) {
32             System.out.print(Array.get(obj, i)+" ");
33         }
34     }
35 }
```

【运行结果】:

```
数组长度为： 15
1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 ====================
数组长度为： 8
a b c null null null null null
```

动态代理

【案例】首先来看看如何获得类加载器：

```
?
  classtest{
1
2 }
3 classhello{
4     publicstaticvoidmain(String[] args) {
5         test t=newtest();
6         System.out.println("          类          加          载          器
7 "+t.getClass().getClassLoader().getClass().getName());
8     }
9 }
```

【程序输出】:

```
类加载器  sun.misc.Launcher$AppClassLoader
```

其实在 java 中有三种类类加载器。

1）Bootstrap ClassLoader 此加载器采用 c++编写，一般开发中很少见。

2）Extension ClassLoader 用来进行扩展类的加载，一般对应的是 jre\lib\ext 目录中的类

3）AppClassLoader 加载 classpath 指定的类，是最常用的加载器。同时也是 java 中默认的加载器。

如果想要完成动态代理，首先需要定义一个 InvocationHandler 接口的子类，已完成代理的具体操作。

```
?
1  packageReflect;
2  importjava.lang.reflect.*;
3
4  //定义项目接口
5  interfaceSubject {
6      publicString say(String name,intage);
7  }
8
9  // 定义真实项目
10 classRealSubjectimplementsSubject {
11     @Override
12     publicString say(String name,intage) {
13         returnname +"  "+ age;
14     }
15 }
16
17 classMyInvocationHandlerimplementsInvocationHandler {
18     privateObject obj =null;
19
20     publicObject bind(Object obj) {
21         this.obj = obj;
22         returnProxy.newProxyInstance(obj.getClass().getClassLoader(),
23 obj
24                 .getClass().getInterfaces(),this);
25     }
26
```

```
27     @Override
28     publicObject invoke(Object proxy, Method method, Object[] args)
29             throwsThrowable {
30         Object temp = method.invoke(this.obj, args);
31         returntemp;
32     }
33 }
34
35 classhello {
36     publicstaticvoidmain(String[] args) {
37         MyInvocationHandler demo =newMyInvocationHandler();
38         Subject sub = (Subject) demo.bind(newRealSubject());
39         String info = sub.say("Rollen",20);
40         System.out.println(info);
41     }
   }
```

【运行结果】:

```
Rollen  20
```

类的生命周期

在一个类编译完成之后，下一步就需要开始使用类，如果要使用一个类，肯定离不开 JVM。在程序执行中 JVM 通过装载，链接，初始化这3个步骤完成。

类的装载是通过类加载器完成的，加载器将 .class 文件的二进制文件装入 JVM 的方法区，并且在堆区创建描述这个类的 java.lang.Class 对象。用来封装数据。但是同一个类只会被类装载器装载以前

链接就是把二进制数据组装为可以运行的状态。

链接分为校验，准备，解析这3个阶段

校验一般用来确认此二进制文件是否适合当前的 JVM（版本），

准备就是为静态成员分配内存空间，。并设置默认值

解析指的是转换常量池中的代码作为直接引用的过程，直到所有的符号引用都可以被运行程序使用（建立完整的对应关系）

完成之后，类型也就完成了初始化，初始化之后类的对象就可以正常使用了，直到一个对象不再使用之后，将被垃圾回收。释放空间。

当没有任何引用指向 Class 对象时就会被卸载，结束类的生命周期

将反射用于工厂模式

先来看看，如果不用反射的时候，的工厂模式吧：

http://www.cnblogs.com/rollenholt/archive/2011/08/18/2144851.html

```
?
1  /**
2   * @author Rollen-Holt 设计模式之 工厂模式
3   */
4
5  interfacefruit{
6      publicabstractvoideat();
7  }
8
9  classAppleimplementsfruit{
10     publicvoideat(){
11         System.out.println("Apple");
12     }
13 }
14
15 classOrangeimplementsfruit{
16     publicvoideat(){
17         System.out.println("Orange");
18     }
19 }
20
21 // 构造工厂类
```

```
22 // 也就是说以后如果我们在添加其他的实例的时候只需要修改工厂类就行了
23 classFactory{
24     publicstaticfruit getInstance(String fruitName){
25         fruit f=null;
26         if("Apple".equals(fruitName)){
27             f=newApple();
28         }
29         if("Orange".equals(fruitName)){
30             f=newOrange();
31         }
32         returnf;
33     }
34 }
35 classhello{
36     publicstaticvoidmain(String[] a){
37         fruit f=Factory.getInstance("Orange");
38         f.eat();
39     }
40
41 }
```

这样，当我们在添加一个子类的时候，就需要修改工厂类了。如果我们添加太多的子类的时候，改的就会很多。

现在我们看看利用反射机制：

```
?
1  packageReflect;
2
3  interfacefruit{
4      publicabstractvoideat();
5  }
6
```

```
7  classAppleimplementsfruit{
8      publicvoideat(){
9          System.out.println("Apple");
10     }
11 }
12
13 classOrangeimplementsfruit{
14     publicvoideat(){
15         System.out.println("Orange");
16     }
17 }
18
19 classFactory{
20     publicstaticfruit getInstance(String ClassName){
21         fruit f=null;
22         try{
23             f=(fruit)Class.forName(ClassName).newInstance();
24         }catch(Exception e) {
25             e.printStackTrace();
26         }
27         returnf;
28     }
29 }
30 classhello{
31     publicstaticvoidmain(String[] a){
32         fruit f=Factory.getInstance("Reflect.Apple");
33         if(f!=null){
34             f.eat();
35         }
36     }
37 }
```

现在就算我们添加任意多个子类的时候，工厂类就不需要修改。

上面的爱吗虽然可以通过反射取得接口的实例，但是需要传入完整的包和类名。而且用户也无法知道一个接口有多少个可以使用的子类，所以我们通过属性文件的形式配置所需要的子类。

下面我们来看看： 结合属性文件的工厂模式

首先创建一个 fruit.properties 的资源文件，

内容为：

```
?
  apple=Reflect.Apple
1 orange=Reflect.Orang
2 e
```

然后编写主类代码：

```
?
1  packageReflect;
2
3  importjava.io.*;
4  importjava.util.*;
5
6  interfacefruit{
7      publicabstractvoideat();
8  }
9
10 classAppleimplementsfruit{
11     publicvoideat(){
12         System.out.println("Apple");
13     }
14 }
```

```
15
16 classOrangeimplementsfruit{
17     publicvoideat(){
18         System.out.println("Orange");
19     }
20 }
21
22 //操作属性文件类
23 classinit{
24     publicstaticProperties        getPro()throwsFileNotFoundException,
25 IOException{
26         Properties pro=newProperties();
27         File f=newFile("fruit.properties");
28         if(f.exists()){
29             pro.load(newFileInputStream(f));
30         }else{
31             pro.setProperty("apple","Reflect.Apple");
32             pro.setProperty("orange","Reflect.Orange");
33             pro.store(newFileOutputStream(f),"FRUIT CLASS");
34         }
35         returnpro;
36     }
37 }
38
39 classFactory{
40     publicstaticfruit getInstance(String ClassName){
41         fruit f=null;
42         try{
43             f=(fruit)Class.forName(ClassName).newInstance();
44         }catch(Exception e) {
45             e.printStackTrace();
46         }
```

```
47        returnf;
48    }
49 }
50 classhello{
51    publicstaticvoidmain(String[]      a)throwsFileNotFoundException,
52 IOException{
53        Properties pro=init.getPro();
54        fruit f=Factory.getInstance(pro.getProperty("apple"));
55        if(f!=null){
56            f.eat();
57        }
      }
   }
```

【运行结果】：Apple