第 10 章　Java 语言的反射机制

在 Java 运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法？答案是肯定的。这种动态获取类的信息，以及动态调用对象的方法的功能来自于 Java 语言的反射（Reflection）机制。Java 反射机制主要提供了以下功能：

- 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；
- 在运行时判断任意一个类所具有的成员变量和方法；
- 在运行时调用任意一个对象的方法；
- 生成动态代理。

本章首先介绍了 Java Reflection API 的用法，然后介绍了一个远程方法调用的例子，在这个例子中客户端能够远程调用服务器端的一个对象的方法。服务器端采用了反射机制提供的动态调用方法的功能，而客户端则采用了反射机制提供的动态代理功能。

10.1　Java Reflection API 简介

在 JDK 中，主要由以下类来实现 Java 反射机制，这些类都位于 java.lang.reflect 包中。

- Class 类：代表一个类。
- Field 类：代表类的成员变量（成员变量也称为类的属性）。
- Method 类：代表类的方法。
- Constructor 类：代表类的构造方法。
- Array 类：提供了动态创建数组，以及访问数组元素的静态方法。

如例程 10-1 所示 DumpMethods 类演示了 Reflection API 的基本作用，它读取命令行参数指定的类名，然后打印这个类所具有的方法信息：

例程 10-1　DumpMethods.java

```
import java.lang.reflect.*;
public class DumpMethods {
   public static void main(String args[]) throws Exception{
      //加载并初始化命令行参数指定的类
      Class classType = Class.forName(args[0]);
      //获得类的所有方法
      Method methods[] = classType.getDeclaredMethods();
      for(int i = 0; i < methods.length; i++)
         System.out.println(methods[i].toString());
   }
}
```

运行命令“java DumpMethods java.util.Stack”，就会显示 java.util.Stack 类所具有的方法，程序的打印结果如下：

```
public synchronized java.lang.Object java.util.Stack.pop()
public java.lang.Object java.util.Stack.push(java.lang.Object)
public boolean java.util.Stack.empty()
public synchronized java.lang.Object java.util.Stack.peek()
public synchronized int java.util.Stack.search(java.lang.Object)
```

如例程 10-2 所示 ReflectTester 类进一步演示了 Reflection API 的基本使用方法。ReflectTester 类有一个 copy(Object object)方法，这个方法能够创建一个和参数 object 同样类型的对象，然后把 object 对象中的所有属性复制到新建的对象中，并将它返回。

这个例子只能复制简单的 JavaBean，假定 JavaBean 的每个属性都有 public 类型的 getXXX()和 setXXX()方法。

例程 10-2　ReflectTester.java

```
import java.lang.reflect.*;
public class ReflectTester {
   public Object copy(Object object) throws Exception{
      //获得对象的类型
      Class classType=object.getClass();
      System.out.println("Class:"+classType.getName());

      //通过默认构造方法创建一个新的对象
      Object objectCopy=classType.getConstructor(new Class[]{}).
            newInstance(new Object[]{});

      //获得对象的所有属性
      Field fields[]=classType.getDeclaredFields();

      for(int i=0; i<fields.length;i++){
         Field field=fields[i];

         String fieldName=field.getName();
         String firstLetter=fieldName.substring(0,1).toUpperCase();
         //获得和属性对应的 getXXX()方法的名字
         String getMethodName="get"+firstLetter+fieldName.substring(1);
         //获得和属性对应的 setXXX()方法的名字
         String setMethodName="set"+firstLetter+fieldName.substring(1);

         //获得和属性对应的 getXXX()方法
         Method getMethod=classType.getMethod(getMethodName,new Class[]{});
         //获得和属性对应的 setXXX()方法
         Method setMethod=classType.getMethod(setMethodName,new Class[]{field.getType()});

         //调用原对象的 getXXX()方法
         Object value=getMethod.invoke(object,new Object[]{});
         System.out.println(fieldName+":"+value);
         //调用复制对象的 setXXX()方法
         etMethod.invoke(objectCopy,new Object[]{value});
      }
      return objectCopy;
```

```
    }

    public static void main(String[] args) throws Exception{
        Customer customer=new Customer("Tom",21);
        customer.setId(new Long(1));

        Customer customerCopy=(Customer)new ReflectTester().copy(customer);
        System.out.println("Copy information:"+customerCopy.getName()+""+
            customerCopy.getAge());
    }
}

class Customer{                    //Customer 类是一个 JavaBean
    private Long id;
    private String name;
    private int age;

    public Customer(){}
    public Customer(String name,int age){
        this.name=name;
        this.age=age;
    }

    public Long getId(){return id;}
    public void setId(Long id){this.id=id;}

    public String getName(){return name;}
    public void setName(String name){this.name=name;}

    public int getAge(){return age;}
    public void setAge(int age){this.age=age;}
}
```

ReflectTester 类的 copy(Object object)方法依次执行以下步骤。

（1）获得对象的类型：

```
Class classType=object.getClass();
System.out.println("Class:"+classType.getName());
```

在 java.lang.Object 类中定义了 getClass()方法，因此对于任意一个 Java 对象，都可以通过此方法获得对象的类型。Class 类是 Reflection API 中的核心类，它有以下方法。

- getName()：获得类的完整名字。
- getFields()：获得类的 public 类型的属性。
- getDeclaredFields()：获得类的所有属性。
- getMethods()：获得类的 public 类型的方法。
- getDeclaredMethods()：获得类的所有方法。
- getMethod(String name, Class[] parameterTypes)：获得类的特定方法，name 参数指定方法的名字，parameterTypes 参数指定方法的参数类型。
- getConstrutors()：获得类的 public 类型的构造方法。
- getConstrutor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes

参数指定构造方法的参数类型。

- newInstance()：通过类的不带参数的构造方法创建这个类的一个对象。

（2）通过默认构造方法创建一个新的对象：

```
Object objectCopy=classType.getConstructor(new Class[]{}).newInstance(new Object[]{});
```

以上代码先调用 Class 类的 getConstructor()方法获得一个 Constructor 对象，它代表默认的构造方法，然后调用 Constructor 对象的 newInstance()方法构造一个实例。

（3）获得对象的所有属性：

```
Field fields[]=classType.getDeclaredFields();
```

Class 类的 getDeclaredFields()方法返回类的所有属性，包括 public、protected、默认和 private 访问级别的属性。

（4）获得每个属性相应的 getXXX()和 setXXX()方法，然后执行这些方法，把原来对象的属性复制到新的对象中：

```
for(int i=0; i<fields.length;i++){
  Field field=fields[i];

  String fieldName=field.getName();
  String firstLetter=fieldName.substring(0,1).toUpperCase();
  //获得和属性对应的 getXXX()方法的名字
  String getMethodName="get"+firstLetter+fieldName.substring(1);
  //获得和属性对应的 setXXX()方法的名字
  String setMethodName="set"+firstLetter+fieldName.substring(1);

  //获得和属性对应的 getXXX()方法
  Method getMethod=classType.getMethod(getMethodName,new Class[]{});
  //获得和属性对应的 setXXX()方法
  Method setMethod=classType.getMethod(setMethodName,new Class[]{field.getType()});

  //调用原对象的 getXXX()方法
  Object value=getMethod.invoke(object,new Object[]{});
  System.out.println(fieldName+":"+value);
  //调用复制对象的 setXXX()方法
  setMethod.invoke(objectCopy,new Object[]{value});
}
```

以上代码假定每个属性都有相应的 getXXX()和 setXXX()方法，并且在方法名中，“get”和“set”的后面一个字母为大写。例如，Customer 类的 name 属性对应 getName()和 setName()方法。Method 类的 invoke(Object obj,Object args[])方法用于动态执行一个对象的特定方法，它的第一个 obj 参数指定具有该方法的对象，第二个 args 参数指定向该方法传递的参数。

如例程 10-3 所示的 InvokeTester 类的 main()方法中，运用反射机制调用一个 InvokeTester 对象的 add()和 echo()方法。

例程 10-3　InvokeTester.java

```
import java.lang.reflect.*;
public class InvokeTester {
```

```
    public int add(int param1,int param2){
        return param1+param2;
    }
    public String echo(String msg){
        return "echo:"+msg;
    }
    public static void main(String[] args) throws Exception{
        Class classType=InvokeTester.class;
        Object invokeTester=classType.newInstance();

        //调用 InvokeTester 对象的 add()方法
        Method addMethod=classType.getMethod("add",new Class[]{int.class,int.class});
        Object result=addMethod.invoke(invokeTester,
                new Object[]{new Integer(100),new Integer(200)});
        System.out.println((Integer)result);

        //调用 InvokeTester 对象的 echo()方法
        Method echoMethod=classType.getMethod("echo",new Class[]{String.class});
        result=echoMethod.invoke(invokeTester,new Object[]{"Hello"});
        System.out.println((String)result);
    }
}
```

add()方法的两个参数为 int 类型，获得表示 add()方法的 Method 对象的代码如下：

```
Method addMethod=classType.getMethod("add",new Class[]{int.class,int.class});
```

Method 类的 invoke(Object obj,Object args[])方法接收的参数必须为对象，如果参数为基本类型数据，必须转换为相应的包装类型的对象。invoke()方法的返回值总是对象，如果实际被调用的方法的返回类型是基本类型数据，那么 invoke()方法会把它转换为相应的包装类型的对象，再将其返回。

在本例中，尽管 InvokeTester 类的 add()方法的两个参数及返回值都是 int 类型，调用 addMethod 对象的 invoke()方法时，只能传递 Integer 类型的参数，并且 invoke()方法的返回类型也是 Integer 类型，Integer 类是 int 基本类型的包装类：

```
Object result=addMethod.invoke(invokeTester,
                                new Object[]{new Integer(100),new Integer(200)});
System.out.println((Integer)result);   //result 为 Integer 类型
```

java.lang.Array 类提供了动态创建和访问数组元素的各种静态方法。如例程 10-4 所示的 ArrayTester1 类的 main()方法创建了一个长度为 10 的字符串数组，接着把索引位置为 5 的元素设为“hello”，然后再读取索引位置为 5 的元素的值。

例程 10-4　ArrayTester1.java

```
import java.lang.reflect.*;
public class ArrayTester1 {
    public static void main(String args[])throws Exception {
        Class classType = Class.forName("java.lang.String");
        //创建一个长度为 10 的字符串数组
        Object array = Array.newInstance(classType, 10);
        //把索引位置为 5 的元素设为"hello"
        Array.set(array, 5, "hello");
```

```
        //读取索引位置为 5 的元素的值
        String s = (String) Array.get(array, 5);
        System.out.println(s);
    }
}
```

如例程 10-5 所示的 ArrayTester2 类的 main()方法创建了一个 5×10×15 的整型数组，并把索引位置为[3][5][10]的元素的值为设 37。

例程 10-5　ArrayTester2.java

```
import java.lang.reflect.*;
public class ArrayTester2{
    public static void main(String args[]) {
        int dims[] = new int[]{5, 10, 15};
        Object array = Array.newInstance(Integer.TYPE, dims);
        //使 arrayObj 引用 array[3]
        Object arrayObj = Array.get(array, 3);
        Class cls = arrayObj.getClass().getComponentType();
        System.out.println(cls);
        //使 arrayObj 引用 array[3][5]
        arrayObj = Array.get(arrayObj, 5);
        //把元素 array[3][5][10]设为 37
        Array.setInt(arrayObj, 10, 37);
        int arrayCast[][][] = (int[][][]) array;
        System.out.println(arrayCast[3][5][10]);
    }
}
```

10.2　在远程方法调用中运用反射机制

假定在 SimpleServer 服务器端创建了一个 HelloServiceImpl 对象，它具有 getTime()和 echo()方法。HelloServiceImpl 类实现了 HelloService 接口。如例程 10-6 和例程 10-7 所示分别是 HelloService 接口和 HelloServiceImpl 类的源程序。

例程 10-6　HelloService.java

```
package remotecall;
import java.util.Date;
public interface HelloService{
    public String echo(String msg);
    public Date getTime();
}
```

例程 10-7　HelloServiceImpl.java

```
package remotecall;
import java.util.Date;
public class HelloServiceImpl implements HelloService{
    public String echo(String msg){
        return "echo:"+msg;
    }
    public Date getTime(){
```

```
        return new Date();
    }
}
```

SimpleClient 客户端如何调用服务器端的 HelloServiceImpl 对象的 getTime()和 echo()方法呢？显然，SimpleClient 客户端需要把调用的方法名、方法参数类型、方法参数值，以及方法所属的类名或接口名发送给 SimpleServer，SimpleServer 再调用相关对象的方法，然后把方法的返回值发送给 SimpleClient。

为了便于按照面向对象的方式来处理客户端与服务器端的通信，可以把它们发送的信息用 Call 类（如例程 10-8 所示）来表示。一个 Call 对象表示客户端发起的一个远程调用，它包括调用的类名或接口名、方法名、方法参数类型、方法参数值和方法执行结果。

例程 10-8　Call.java

```
package remotecall;
import java.io.*;
public class Call implements Serializable{
    private String className;           //表示类名或接口名
    private String methodName;          //表示方法名
    private Class[] paramTypes;         //表示方法参数类型
private Object[] params;                //表示方法参数值

//表示方法的执行结果
//如果方法正常执行，则 result 为方法返回值，如果方法抛出异常，那么 result 为该异常。
    private Object result;

    public Call(){}
    public Call(String className,String methodName,Class[] paramTypes,
                                        Object[] params){
        this.className=className;
        this.methodName=methodName;
        this.paramTypes=paramTypes;
        this.params=params;
    }

    public String getClassName(){return className;}
    public void setClassName(String className){this.className=className;}

    public String getMethodName(){return methodName;}
    public void setMethodName(String methodName){this.methodName=methodName;}

    public Class[] getParamTypes(){return paramTypes;}
    public void setParamTypes(Class[] paramTypes){this.paramTypes=paramTypes;}

    public Object[] getParams(){return params;}
    public void setParams(Object[] params){this.params=params;}

    public Object getResult(){return result;}
    public void setResult(Object result){this.result=result;}

    public String toString(){
        return "className="+className+" methodName="+methodName;
```

```
        }
    }
```

SimpleClient 调用 SimpleServer 端的 HelloServiceImpl 对象的 echo()方法的流程如下。

（1）SimpleClient 创建一个 Call 对象，它包含了调用 HelloService 接口的 echo()方法的信息。

（2）SimpleClient 通过对象输出流把 Call 对象发送给 SimpleServer。

（3）SimpleServer 通过对象输入流读取 Call 对象，运用反射机制调用 HelloServiceImpl 对象的 echo()方法，把 echo()方法的执行结果保存到 Call 对象中。

（4）SimpleServer 通过对象输出流把包含了方法执行结果的 Call 对象发送给 SimpleClient。

（5）SimpleClient 通过对象输入流读取 Call 对象，从中获得方法执行结果。

如例程 10-9 和例程 10-10 所示分别是 SimpleServer 和 SimpleClient 的源程序。

例程 10-9　SimpleServer.java

```
package remotecall;
import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.reflect.*;
public class SimpleServer {
   private Map remoteObjects=new HashMap();          //存放远程对象的缓存

   /** 把一个远程对象放到缓存中 */
   public void register(String className,Object remoteObject){
      remoteObjects.put( className,remoteObject);
   }
   public void service()throws Exception{
      ServerSocket serverSocket = new ServerSocket(8000);
      System.out.println("服务器启动.");
      while(true){
         Socket socket=serverSocket.accept();
         InputStream in=socket.getInputStream();
         ObjectInputStream ois=new ObjectInputStream(in);
         OutputStream out=socket.getOutputStream();
         ObjectOutputStream oos=new ObjectOutputStream(out);

         Call call=(Call)ois.readObject();                  //接收客户发送的 Call 对象
         System.out.println(call);
         call=invoke(call);                                 //调用相关对象的方法
         oos.writeObject(call);                             //向客户发送包含了执行结果的 Call 对象

         ois.close();
         oos.close();
         socket.close();
      }
   }

   public Call invoke(Call call){
```

```
        Object result=null;
        try{
          String className=call.getClassName();
          String methodName=call.getMethodName();
          Object[] params=call.getParams();
          Class classType=Class.forName(className);
          Class[] paramTypes=call.getParamTypes();
          Method method=classType.getMethod(methodName,paramTypes);
          Object remoteObject=remoteObjects.get(className);          //从缓存中取出相关的远程对象
          if(remoteObject==null){
            throw new Exception(className+"的远程对象不存在");
          }else{
            result=method.invoke(remoteObject,params);
          }
        }catch(Exception e){result=e;}

        call.setResult(result);                                      //设置方法执行结果
        return call;
      }

      public static void main(String args[])throws Exception {
        SimpleServer server=new SimpleServer();
        //把事先创建的 HelloServiceImpl 对象加入到服务器的缓存中
        server.register("remotecall.HelloService",new HelloServiceImpl());
        server.service();
      }
    }
```

例程 10-10　SimpleClient.java

```
package remotecall;
import java.io.*;
import java.net.*;
import java.util.*;
public class SimpleClient {
  public void invoke()throws Exception{
    Socket socket = new Socket("localhost",8000);
    OutputStream out=socket.getOutputStream();
    ObjectOutputStream oos=new ObjectOutputStream(out);
    InputStream in=socket.getInputStream();
    ObjectInputStream ois=new ObjectInputStream(in);

    //Call call=new Call("remotecall.HelloService","getTime",
        new Class[]{},new Object[]{});
    Call call=new Call("remotecall.HelloService","echo",
                  new Class[]{String.class},new Object[]{"Hello"});
    oos.writeObject(call);                                           //向服务器发送 Call 对象
    call=(Call)ois.readObject();                                     //接收包含了方法执行结果的 Call 对象
    System.out.println(call.getResult());

    ois.close();
    oos.close();
    socket.close();
  }
  public static void main(String args[])throws Exception {
```

```
            new SimpleClient().invoke();
        }
    }
```

先运行命令“java remotecall.SimpleServer”，再运行命令“java remotecall.SimpleClient”，SimpleClient 端将打印“echo:Hello”。该打印结果是服务器端执行 HelloServiceImpl 对象的 echo()方法的返回值。如图 10-1 所示显示了 SimpleClient 与 SimpleServer 的通信过程。

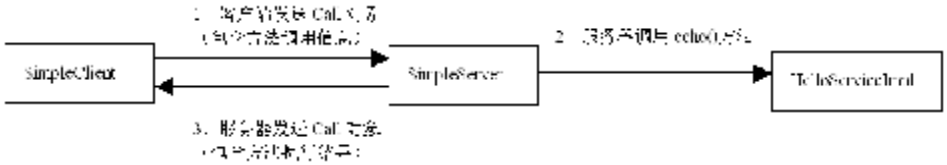

图 10-1　SimpleClient 与 SimpleServer 的通信过程

10.3　代理模式

代理模式是常用的 Java 设计模式，它的特征是代理类与委托类有同样的接口，如图 10-2 所示。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。

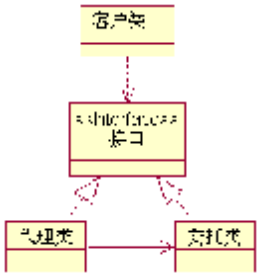

图 10-2　代理模式

按照代理类的创建时期，代理类可分为两种。

- 静态代理类：由程序员创建或由特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class 文件就已经存在了。
- 动态代理类：在程序运行时，运用反射机制动态创建而成。

10.3.1　静态代理类

如图 10-3 所示，HelloServiceProxy 类（如例程 10-13 所示）是代理类，HelloServiceImpl 类（如例程 10-12 所示）是委托类，这两个类都实现了 HelloService 接口（如例程 10-11 所示）。其中 HelloServiceImpl 类是 HelloService 接口的真正实现者，而 HelloServiceProxy 类是通过调用 HelloServiceImpl 类的相关方法来提供特定服务的。HelloServiceProxy 类的 echo()方法和 getTime()方法会分别调用被代理的 HelloServiceImpl 对象的 echo()方法和 getTime()方法，并且在方法调用前后都会执行一些简单的打印操作。由此可见，代理类可以为委托类预处理消息、把消息转发给委托类和事后处理消息等。

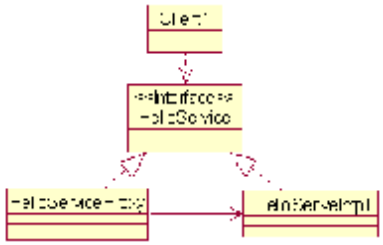

图 10-3　HelloServiceProxy 类是 HelloService 的代理类

例程 10-11　HelloService.java

```
package proxy;
import java.util.Date;
public interface HelloService{
   public String echo(String msg);
   public Date getTime();
}
```

例程 10-12　HelloServiceImpl.java

```
package proxy;
import java.util.Date;
public class HelloServiceImpl implements HelloService{
   public String echo(String msg){
      return "echo:"+msg;
   }
   public Date getTime(){
      return new Date();
   }
}
```

例程 10-13　HelloServiceProxy.java

```
package proxy;
import java.util.Date;
public class HelloServiceProxy implements HelloService{
   private HelloService helloService;                    //表示被代理的 HelloService 实例

   public HelloServiceProxy(HelloService helloService){
      this.helloService=helloService;
   }

   public void setHelloServiceProxy(HelloService helloService){
      this.helloService=helloService;
   }

   public String echo(String msg){
      System.out.println("before calling echo()");       //预处理
      String result=helloService.echo(msg);              //调用被代理的 HelloService 实例的 echo()方法
      System.out.println("after calling echo()");        //事后处理
      return result;
   }
   public Date getTime(){
      System.out.println("before calling getTime()");    //预处理
```

```
        Date date=helloService.getTime();                    //调用被代理的 HelloService 实例的 getTime()方法
        System.out.println("after calling getTime()");       //事后处理
        return date;
    }
}
```

在 Client1 类（如例程 10-14 所示）的 main()方法中，先创建了一个 HelloServiceImpl 对象，又创建了一个 HelloServiceProxy 对象，最后调用 HelloServiceProxy 对象的 echo()方法。

例程 10-14　Client1.java

```
package proxy;
public class Client1{
    public static void main(String args[]){
        HelloService helloService=new HelloServiceImpl();
        HelloService helloServiceProxy=new HelloServiceProxy(helloService);
        System.out.println(helloServiceProxy.echo("hello"));
    }
}
```

运行 Client1 类，打印结果如下：

```
before calling echo()
after calling echo()
echo:hello
```

如图 10-4 所示显示了 Client1 调用 HelloServiceProxy 类的 echo()方法的时序图。

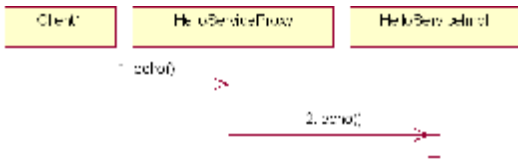

图 10-4　Client1 调用 HelloServiceProxy 类的 echo()方法的时序图

例程 10-13 的 HelloServiceProxy 类的源代码是由程序员编写的，在程序运行前，它的.class 文件就已经存在了，这种代理类称为静态代理类。

10.3.2　动态代理类

与静态代理类对照的是动态代理类，动态代理类的字节码在程序运行时由 Java 反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为 Java 反射机制可以生成任意类型的动态代理类。java.lang.reflect 包中的 Proxy 类和 InvocationHandler 接口提供了生成动态代理类的能力。

Proxy 类提供了创建动态代理类及其实例的静态方法。

（1）getProxyClass()静态方法负责创建动态代理类，它的完整定义如下：

```
public static Class<?> getProxyClass(ClassLoader loader, Class<?>[] interfaces)
```

```
                                                    throws IllegalArgumentException
```

参数 loader 指定动态代理类的类加载器，参数 interfaces 指定动态代理类需要实现的所有接口。

（2）newProxyInstance()静态方法负责创建动态代理类的实例，它的完整定义如下：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,
                                       InvocationHandler handler) throws IllegalArgumentException
```

参数 loader 指定动态代理类的类加载器，参数 interfaces 指定动态代理类需要实现的所有接口，参数 handler 指定与动态代理类关联的 InvocationHandler 对象。

以下两种方式都创建了实现 Foo 接口的动态代理类的实例：

```
                    /****  方式一  ****/
//创建 InvocationHandler 对象
InvocationHandler handler = new MyInvocationHandler(...);
//创建动态代理类
Class proxyClass = Proxy.getProxyClass(
    Foo.class.getClassLoader(), new Class[] { Foo.class });
//创建动态代理类的实例
Foo foo = (Foo) proxyClass.
    getConstructor(new Class[] { InvocationHandler.class }).
    newInstance(new Object[] { handler });

                    /****  方式二  ****/
//创建 InvocationHandler 对象
InvocationHandler handler = new MyInvocationHandler(...);
//直接创建动态代理类的实例
Foo foo = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                       new Class[] { Foo.class },
                                       handler);
```

由 Proxy 类的静态方法创建的动态代理类具有以下特点：

- 动态代理类是 public、final 和非抽象类型的；
- 动态代理类继承了 java.lang.reflect.Proxy 类；
- 动态代理类的名字以“$Proxy”开头；
- 动态代理类实现 getProxyClass()和 newProxyInstance()方法中参数 interfaces 指定的所有接口；
- Proxy 类的 isProxyClass(Class<?> cl)静态方法可用来判断参数指定的类是否为动态代理类。只有通过 Proxy 类创建的类才是动态代理类；
- 动态代理类都具有一个 public 类型的构造方法，该构造方法有一个 InvocationHandler 类型的参数。

由 Proxy 类的静态方法创建的动态代理类的实例具有以下特点：

- 假定变量 foo 是一个动态代理类的实例，并且这个动态代理类实现了 Foo 接口，那么“foo instanceof Foo”的值为 true。把变量 foo 强制转换为 Foo 类型是合法的：

```
(Foo) foo        //合法
```

- 每个动态代理类实例都和一个 InvocationHandler 实例关联。Proxy 类的

getInvocationHandler(Object proxy)静态方法返回与参数 proxy 指定的代理类实例所关联的 InvocationHandler 对象。

- 假定 Foo 接口有一个 amethod()方法，那么当程序调用动态代理类实例 foo 的 amethod()方法时，该方法会调用与它关联的 InvocationHandler 对象的 invoke()方法。

InvocationHandler 接口为方法调用接口，它声明了负责调用任意一个方法的 invoke()方法：

```
Object invoke(Object proxy,Method method,Object[] args) throws Throwable
```

参数 proxy 指定动态代理类实例，参数 method 指定被调用的方法，参数 args 指定向被调用方法传递的参数，invoke()方法的返回值表示被调用方法的返回值。

如图 10-5 所示，HelloServiceProxyFactory 类（如例程 10-15 所示）的 getHello-ServiceProxy()静态方法负责创建实现了 HelloService 接口的动态代理类的实例。

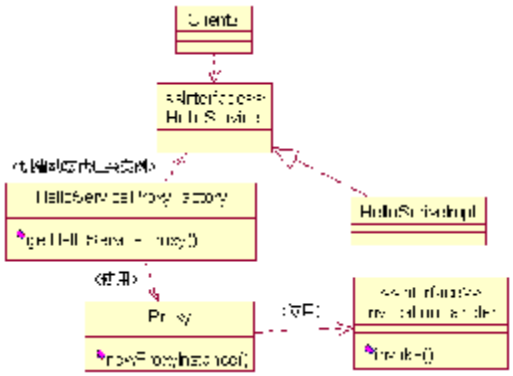

图 10-5　HelloServiceProxyFactory 类创建动态代理类实例

例程 10-15　HelloServiceProxyFactory.java

```
package proxy;
import java.lang.reflect.*;
public class HelloServiceProxyFactory {
   /**  创建一个实现了 HelloService 接口的动态代理类的实例
     *   参数 helloService 引用被代理的 HelloService 实例
     */
public static HelloService getHelloServiceProxy(final HelloService helloService){
   //创建一个实现了 InvocationHandler 接口的匿名类的实例
      InvocationHandler handler=new InvocationHandler(){
         public Object invoke(Object proxy,Method method,Object args[])throws Exception{
            System.out.println("before calling "+method);                    //预处理
            Object result=method.invoke(helloService,args);
//调用被代理的 HelloService 实例的方法
            System.out.println("after calling "+method);                     //事后处理
            return result;
         }
      };

      Class classType=HelloService.class;
      return (HelloService)Proxy.newProxyInstance(classType.getClassLoader(),
```

```
                                                    new Class[]{classType},
                                                    handler);
        }//# getHelloServiceProxy()
    }
```

如例程 10-16 所示的 Client2 类先创建了一个 HelloServiceImpl 实例，然后创建了一个动态代理类实例 helloServiceProxy，最后调用动态代理类实例的 echo()方法。

例程 10-16　Client2.java

```
package proxy;
public class Client2{
    public static void main(String args[]){
        HelloService helloService=new HelloServiceImpl();
        HelloService helloServiceProxy=
                    HelloServiceProxyFactory.getHelloServiceProxy(helloService);
        System.out.println("动态代理类的名字为"
                                +helloServiceProxy.getClass().getName());
        System.out.println(helloServiceProxy.echo("Hello"));
    }
}
```

运行 Client2，打印结果如下：

```
动态代理类的名字为$Proxy0
before calling public abstract java.lang.
        String proxy.HelloService.echo(java.lang.String)
after calling public abstract java.lang.
        String proxy.HelloService.echo(java.lang.String)
echo:Hello
```

从以上打印结果看出，动态代理类的名字为$Proxy0。如图 10-6 所示显示了 Client2 调用动态代理类$Proxy0 的实例 helloServiceProxy 的 echo()方法的时序图。

图 10-6　Client2 调用动态代理类$Proxy0 的 echo()方法的时序图

10.3.3　在远程方法调用中运用代理类

如图 10-7 所示，SimpleClient 客户端通过 HelloService 代理类来调用 SimpleServer 服务器端的 HelloServiceImpl 对象的方法。客户端的 HelloService 代理类也实现了 HelloService 接口，这可以简化 SimpleClient 客户端的编程。对于 SimpleClient 客户端而言，与远程服务器的通信的细节被封装到 HelloService 代理类中。SimpleClient 客户端可以按照以下方式调用远程服务器上的 HelloServiceImpl 对象的方法：

```
//创建 HelloService 代理类的对象
HelloService helloService1=new HelloServiceProxy(connector);
```

```
//通过代理类调用远程服务器上的 HelloServiceImpl 对象的方法
System.out.println(helloService1.echo("hello"));
```

从以上程序代码可以看出，SimpleClient 客户端调用远程对象的方法的代码与调用本地对象的方法的代码很相似，由此可以看出，代理类简化了客户端的编程。

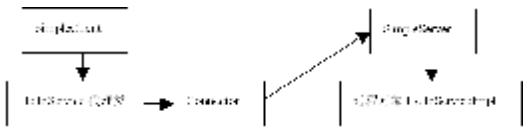

图 10-7　SimpleClient 通过 HelloService 代理类调用远程对象的方法

Connector 类负责建立与远程服务器的连接，以及接收和发送 Socket 对象。如例程 10-17 所示是 Connector 类的源程序。

例程 10-17　Connector.java

```
package proxy1;
import java.io.*;
import java.net.*;
import java.util.*;
public class Connector {
   private String host;
   private int port;
   private Socket skt;
   private InputStream is;
   private ObjectInputStream ois;
   private OutputStream os;
   private ObjectOutputStream oos;

   public Connector(String host,int port)throws Exception{
       this.host=host;
       this.port=port;
       connect(host,port);
   }

   public void send(Object obj)throws Exception{                    //发送对象
      oos.writeObject(obj);
   }
   public Object receive() throws Exception{                        //接收对象
      return ois.readObject();
   }
   public void connect()throws Exception{                           //建立与远程服务器的连接
      connect(host,port);
   }
   public void connect(String host,int port)throws Exception{       //建立与远程服务器的连接
      skt=new Socket(host,port);
      os=skt.getOutputStream();
      oos=new ObjectOutputStream(os);
      is=skt.getInputStream();
      ois=new ObjectInputStream(is);
   }
   public void close(){                                             //关闭连接
      try{
```

```
            }finally{
                try{
                    ois.close();
                    oos.close();
                    skt.close();
                }catch(Exception e){
                    System.out.println("Connector.close: "+e);
                }
            }
        }
    }
```

HelloService 代理类有两种创建方式：一种方式是创建一个 HelloServiceProxy 静态代理类，如例程 10-18 所示；还有一种方式是创建 HelloService 的动态代理类，如例程 10-19 所示 ProxyFactory 类的静态 getProxy()方法就负责创建 HelloService 的动态代理类，并且返回它的一个实例。

例程 10-18　HelloServiceProxy.java（静态代理类）

```
package proxy1;
import java.util.Date;
public class HelloServiceProxy implements HelloService{
    private String host;
    private int port;

    public HelloServiceProxy(String host,int port){
        this.host=host;
        this.port=port;
    }
    public String echo(String msg)throws RemoteException{
        Connector connector=null;
        try{
            connector=new Connector(host,port);
            Call call=new Call("proxy1.HelloService","echo",
                                new Class[]{String.class},new Object[]{msg});
            connector.send(call);
            call=(Call)connector.receive();
            Object result=call.getResult();
            if(result instanceof Throwable)
                throw new RemoteException((Throwable)result);          //把异常都转换为 RemoteException
            else
                return (String)result;
        }catch(Exception e){
            throw new RemoteException(e);                              //把异常都转换为 RemoteException
        }finally{if(connector!=null)connector.close();}
    }

    public Date getTime()throws RemoteException{
        Connector connector=null;
        try{
            connector=new Connector(host,port);
            Call call=new Call("proxy1.HelloService","getTime",new Class[]{},new Object[]{});
            connector.send(call);
            call=(Call)connector.receive();
```

```
            Object result=call.getResult();
            if(result instanceof Throwable)
              throw new RemoteException((Throwable)result);          //把异常都转换为 RemoteException
            else
              return (Date)result;
          }catch(Exception e){
            throw new RemoteException(e);                          //把异常都转换为 RemoteException
          }finally{if(connector!=null)connector.close();}
        }
      }
```

例程 10-19　ProxyFactory.java（负责创建动态代理类及其实例）

```
package proxy1;
import java.lang.reflect.*;
public class ProxyFactory {
  public static Object getProxy(final Class classType,final String host,final int port){
    InvocationHandler handler=new InvocationHandler(){
      public Object invoke(Object proxy,Method method,Object args[])
                                                                  throws Exception{
        Connector connector=null;
        try{
          connector=new Connector(host,port);
          Call call=new Call(classType.getName(),
                             method.getName(),method.getParameterTypes(),args);
          connector.send(call);
          call=(Call)connector.receive();
          Object result=call.getResult();
          if(result instanceof Throwable)
            throw new RemoteException((Throwable)result);          //把异常都转换为 RemoteException
          else
            return result;
        }finally{if(connector!=null)connector.close();}
      }
    };

    return Proxy.newProxyInstance(classType.getClassLoader(),
                                  new Class[]{classType},
                                  handler);
  }
}
```

无论 HelloService 的静态代理类还是动态代理类，都通过 Connector 类来发送和接收 Call 对象。ProxyFactory 工厂类的 getProxy()方法的第一个参数 classType 指定代理类实现的接口的类型，如果参数 classType 的取值为 HelloService.class，那么 getProxy()方法就创建 HelloService 动态代理类的实例。如果参数 classType 的取值为 Foo.class，那么 getProxy()方法就创建 Foo 动态代理类的实例。由此可见，getProxy()方法可以创建任意类型的动态代理类的实例，并且它们都具有调用被代理的远程对象的方法的能力。

如果使用静态代理方式，那么对于每一个需要代理的类，都要手工编写静态代理类的源代码；如果使用动态代理方式，那么只要编写一个动态代理工厂类，它就能自动创建各种类型的动态代理类，从而大大简化了编程，并且提高了软件系统的可扩展

性和可维护性。

如例程 10-20 所示的 SimpleClient 类的 main()方法中，分别通过静态代理类和动态代理类去访问 SimpleServer 服务器上的 HelloServiceImpl 对象的各种方法。

例程 10-20　SimpleClient.java

```
package proxy1;
import java.io.*;
import java.net.*;
import java.util.*;

public class SimpleClient {
   public static void main(String args[])throws Exception {
      //创建静态代理类实例
      HelloService helloService1=new HelloServiceProxy("localhost",8000);
      System.out.println(helloService1.echo("hello"));
      System.out.println(helloService1.getTime());

      //创建动态代理类实例
      HelloService helloService2=
            (HelloService)ProxyFactory.getProxy(HelloService.class,"localhost",8000);
      System.out.println(helloService2.echo("hello"));
      System.out.println(helloService2.getTime());
   }
}
```

先运行命令“java proxy1.SimpleServer”，再运行命令“java proxy1.SimpleClient”，SimpleClient 端的打印结果如下：

```
echo:hello
Thu Nov 02 10:54:48 CST 2006
echo:hello
Thu Nov 02 10:54:49 CST 2006
```

proxy1.SimpleServer 类的源程序与本章 10.2 节的例程 10-9 的 remotecall.SimpleServer 类相同。如图 10-8 和图 10-9 所示分别显示了 SimpleClient 通过 HelloService 静态代理类和动态代理类访问远程 HelloServiceImpl 对象的 echo()方法的时序图。

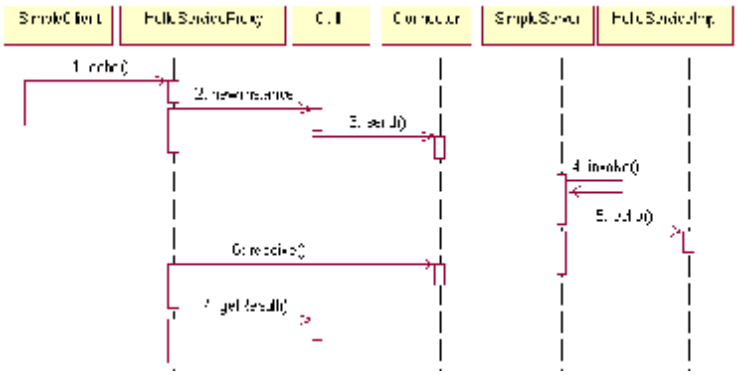

图 10-8　SimpleClient 通过 HelloService 静态代理类（HelloServiceProxy）访问远程对象的方法的时序图

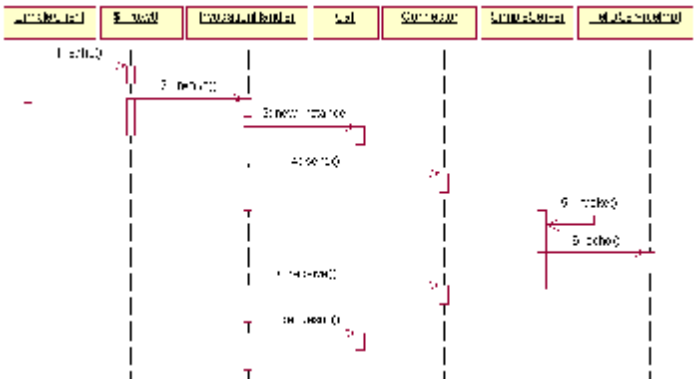

图 10-9　SimpleClient 通过 HelloService 动态代理类（$Proxy0）访问远程对象的时序图

10.4　小结

Java 反射机制是 Java 语言的一个重要特性。考虑实现一个 newInstance(String className)方法，它的作用是根据参数 className 指定的类名，通过该类的不带参数的构造方法创建这个类的对象，并将其返回。如果不运用 Java 反射机制，必须在 newInstance()方法中罗列参数 className 所有可能的取值，然后创建相应的对象：

```
public Object newInstance(String className) throws Exception{
    if(className.equals("HelloService1"))
        return new HelloService1();
    if(className.equals("HelloService2"))
        return new HelloService2();
    if(className.equals("HelloService3"))
        return new HelloService3();
    if(className.equals("HelloService4"))
        return new HelloService4();

    ...

    if(className.equals("HelloService1000"))
        return new HelloService1000();
}
```

以上程序代码很冗长，而且可维护性差。如果在以后软件的升级版本中去除了一个 HelloService4 类，或者增加了一个 HelloService1001 类，都需要修改以上 newInstance()方法。

如果运用反射机制，就可以简化程序代码，并且提高软件系统的可维护性和可扩展性：

```
public Object newInstance(String className) throws Exception{
    Class classType=Class.forName(className);
    return classType.newInstance();
}
```

Java 反射机制在服务器程序和中间件程序中得到了广泛运用。在服务器端，往往需要根据客户的请求，动态调用某一个对象的特定方法。此外，有一种 ORM（Object-Relation Mapping，对象-关系映射）中间件能够把任意一个 JavaBean 持久化到关系数据库中。在 ORM 中间件的实现中，运用 Java 反射机制来读取任意一个 JavaBean 的所有属性，或者给这些属性赋值。在作者的另一本书《精通 Hibernate：Java 对象持久化技术详解》中阐述了 Java 反射机制在 Hibernate（一种 ORM 中间件）的实现中的运用。

在 JDK 类库中，主要由以下类来实现 Java 反射机制，这些类都位于 java.lang.reflect 包中。

- Class 类：代表一个类。
- Field 类：代表类的属性。
- Method 类：代表类的方法。
- Constructor 类：代表类的构造方法。
- Array 类：提供了动态创建数组，以及访问数组元素的静态方法。
- Proxy 类和 InvocationHandler 接口：提供了生成动态代理类及其实例的方法。

本章还介绍了 Java 反射机制、静态代理模式和动态代理模式在远程方法调用中的运用。本章以 SimpleClient 客户调用 SimpleServer 服务器上的 HelloServiceImpl 对象的方法为例，探讨了实现远程方法调用的一些技巧。本书第 11 章介绍的 RMI 框架是 JDK 类库提供的一个现成的完善的远程方法调用框架。即使程序员不了解这个框架本身的实现细节，也能使用这个框架。不过，熟悉框架本身的实现原理，可以帮助程序员更娴熟地运用 RMI 框架。本章对实现远程方法调用所作的初步探讨，有助于程序员去进一步探索 RMI 框架本身的实现原理。

10.5 练习题

1．假定 Tester 类有如下 test 方法：

```
public int test(int p1, Integer p2)
```

以下哪段代码能正确地动态调用一个 Tester 对象的 test 方法？（单选）

A.

```
Class classType=Tester.class;
Object tester=classType.newInstance();
Method addMethod=classType.getMethod("test",new Class[]{int.class,int.class});
Object result=addMethod.invoke(tester,
                                    new Object[]{new Integer(100),new Integer(200)});
```

B.

```
Class classType=Tester.class;
Object tester=classType.newInstance();
Method addMethod=classType.getMethod("test",new Class[]{int.class,int.class});
int result=addMethod.invoke(tester,
                                    new Object[]{new Integer(100),new Integer(200)});
```

C.

```
Class classType=Tester.class;
Object tester=classType.newInstance();
Method addMethod=classType.getMethod("test",new Class[]{int.class,Integer.class});
Object result=addMethod.invoke(tester,
                                    new Object[]{new Integer(100),new Integer(200)});
```

D.

```
Class classType=Tester.class;
Object tester=classType.newInstance();
Method addMethod=classType.getMethod("test",new Class[]{int.class,Integer.class});
Integer result=addMethod.invoke(tester,
                                    new Object[]{new Integer(100),new Integer(200)});
```

2．以下哪些方法在 Class 类中定义？（多选）

A．getConstructors()　　B．getPrivateMethods()　　C．getDeclaredFields()

D．getImports()　　E．setField()

3．以下哪些说法正确？（多选）

A．动态代理类与静态代理类一样，必须由开发人员编写源代码，并编译成.class 文件

B．代理类与被代理类具有同样的接口

C．java.lang.Exception 类实现了 java.io.Serializable 接口，因此 Exception 对象可以被序列化后在网络上传输

D．java.lang.reflect 包中的 Proxy 类提供了创建动态代理类的方法

4．以下哪些属于动态代理类的特点？（多选）

A．动态代理类是 public、final 和非抽象类型的

B．动态代理类继承了 java.lang.reflect.Proxy 类

C．动态代理类实现了 getProxyClass()或 newProxyInstance()方法中参数 interfaces 指定的所有接口

D．动态代理类可以继承用户自定义的任意类

E．动态代理类都具有一个 public 类型的构造方法，该构造方法有一个 InvocationHandler 类型的参数

5．在本章 10.3.3 节（在远程方法调用中运用代理类）介绍的例子中，Connector 类位于服务器端还是客户端？（单选）

A．服务器端　　B．客户端

6．在本章 10.3.3 节（在远程方法调用中运用代理类）介绍的例子中，HelloServiceProxy 类位于服务器端还是客户端？（单选）

A．服务器端　　B．客户端

7．运用本章介绍的动态代理机制，重新实现第 1 章的 EchoServer 服务器与 EchoClient 客户，具体实现方式参照本章 10.3.3 节（在远程方法调用中运用代理类）所介绍的例子。

答案：1．C　2．AC　3．BCD　4．ABCE　5．B　6．B