

Динамическая связь интерфейсов и реализации в JDK 1.3

Алексей Мирошкин

В JDK 1.3 в пакете `java.lang.reflect` (который и сам по себе достаточно интересен) появились класс `Proxy` и интерфейс `InvocationHandler`. Рассмотрим некоторые из возможностей, которые они могут нам предложить. В первую очередь, изучим применение этого API для организации динамической реализации интерфейсов. Звучит ужасно, поэтому попробуем расшифровать.

Предположим, существует класс (`ObjectImpl`), реализующий некоторую логику, и существует некоторый интерфейс (`MyInterface`).

```
public class ObjectImpl
{
    public void operation ()
    {
        System.out.println
            ("You call object's method");
    }
}

public interface MyInterface
{
    public void operation ();
}
```

Как видим, класс и интерфейс не связаны, предположим, писались в разное время разными людьми. Но семантически класс `ObjectImpl` является ничем иным, как реализацией интерфейса `MyInterface` и хотелось бы сохранить эту семантику в нашем приложении, т.е. вызывать методы класса через ссылку на интерфейс. Особенно это актуально при работе с `framework` и сторонней библиотекой классов. `Framework`

Динамическая связь интерфейсов и реализации в JDK 1.3

обычно предоставляет набор интерфейсов, который необходимо реализовать, а в библиотеке классов может находиться подходящая реализация. В этом случае обычно необходимо написать класс-оболочку (wrapper) , который бы включал объект `ObjectImpl` и реализовывал бы интерфейс `MyInterface` делегированием.

```
public class MyInterfaceImpl implements MyInterface
{
    private ObjectImpl impl = null;

    public MyInterfaceImpl (ObjectImpl impl)
    {
        this.impl = impl;
    }

    public void operation ()
    {
        impl.operation();
    }
}
```

Недостаток очевиден — усложнение приложения за счет лишних классов, а главное, для каждой пары интерфейс-реализация приходится писать свой код.

Здесь приходит на помощь **Dynamic proxy API**, не только решая описанную выше проблему, но привнося в приложение изрядную динамичность и гибкость кода.

Оказывается, в JDK 1.3 мы можем динамически связывать интерфейсы и их реализацию.

В разрабатываемом приложении необходимо создать прокси объект, указав интерфейсы, предоставляемые этим объектом, загрузчик классов, а также обработчик вызовов.

На последнем объекте стоит остановиться подробнее. Это — реализация интерфейса `java.lang.reflection.InvocationHandler`. Он содержит единственный метод:

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

Динамическая связь интерфейсов и реализации в JDK 1.3

Когда в приложении вызывается метод через интерфейсный прокси-объект, JVM перенаправляет вызов в предоставленный нами обработчик (посредством создания объекта класса с забавным именем, например, \$Proxy0, который и выполняет за нас грязную работу по оборачиванию методов интерфейса). Нам остается лишь аккуратно реализовать метод `invoke`, заставив его делать то что надо — в данном случае перенаправить вызов.

Реализация обработчика в данном примере построена таким образом, чтобы предоставить должный уровень работоспособности и при этом не скрывать идею за деталями реализации.

```
import java.lang.reflect.*;

public class MyHandler implements InvocationHandler
{
    private Object impl;

    public MyHandler(Object o)
    {
        impl = o;
    }

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

    throws Throwable
    {
        Method myMethod = null;
        // get arguments types
        try {
            Class types[] = null;
            if (args== null) {
                types = new Class[0];
            } else {
                types = new Class[args.length];
                for (int i=0; i < args.length; i++) {
                    types[i]= args[i].getClass();
                }
            }
        }
    }
}
```

```
    }

    try {
        // find implementation method
        myMethod = impl.getClass().getMethod
            (meth.getName(), types);
    }
    catch (Exception e){ throw
        new RuntimeException
            ("Method " + meth.getName()+ " not
            implemented ");
    }
    // invoke
    return myMethod.invoke (impl, args);
}
catch (InvocationTargetException e) {
    throw e.getTargetException();
}
}
```

Смысл этого кода достаточно прост — необходимо, исходя из информации о методе интерфейса, найти соответствующий ему метод реализации. В этом примере предполагается, что имена методов и параметры в интерфейсе и в классе реализации совпадают, но несложно предусмотреть схему, где через конфигурационный файл возможно установить это соответствие иным образом.

Теперь можно создать тестовое приложение:

```
import java.lang.reflect.*;

public class Test
{
    public static void main (String[] args)
    {
        ObjectImpl obj = new ObjectImpl();

        MyInterface inter =
            (MyInterface) Proxy.newProxyInstance
                ( obj.getClass().getClassLoader(),
```

Динамическая связь интерфейсов и реализации в JDK 1.3

```
        new Class[] {MyInterface.class},
        new MyHandler (obj)
    );

    inter.operation();
}
}
```

Как видим, мы вызываем методы объекта через ссылку на интерфейс, хотя между ними нет никакой зависимости.

Хочу выразить благодарность Душкину Роману за помощь в редактировании текста.