Java动态代理与静态代理

六尺帐篷 (/u/f8e9b1c246f1)
2017.03.04 16:19 字数 1071 阅读 171 评论 0 喜欢 13
(/u/f8e9b1c246f1)

编辑文章 (/writer#/notebooks/10012174/notes/9774971)

我们先看一个简单的例子，当我们需要程序中加入方法执行的日志信息的时候，很显然我们最容易想到的实现方法，就是在方法前后插入日志记录信息。

```
import java.util.logging.*;

public class HelloSpeaker {
    private Logger logger =
            Logger.getLogger(this.getClass().getName());

    public void hello(String name) {
        // 方法執行開始時留下日誌
        logger.log(Level.INFO, "hello method starts....");
        // 程式主要功能
        System.out.println("Hello, " + name);
        // 方法執行完畢前留下日誌
        logger.log(Level.INFO, "hello method ends....");
    }
}
```

然后这种实现方式有明显的不足，这种切入式的代码（Cross-cutting），会使得HelloSpeaker拥有了本该不属于他的职责，要在hello的同时记录日志。

试想一下，如果程序中的代码到处都是这种日志需求，那么我们的就必须在到处都加上这些日志代码，想必那是很大的工作量，而且当我们需要修改密码的时候，将会变得更加复杂，维护起来变得困难，所以我们自然想到封装，由于很多对象都需要日志记录这种需求，我们何不把日志行为分离出来。

这时候就可以代理模式解决这个问题，代理又分为**静态代理（Static proxy）**和**动态代理（Dynamic proxy）**

静态代理

在静态代理模式中，代理与被代理对象必须实现同一个接口，代理专注于实现日志记录需求，并在合适的时候，调用被代理对象，这样被代理对象就可以专注于执行业务逻辑。

改进上面那个例子
首先定义一个接口

- IHello.java

```
package Reflection;

public interface IHello {
    public void hello(String name);

}
```

然后专注于业务逻辑实现HelloSpeaker实现上面这个接口：

- HelloSpeaker.java

```
package Reflection;

public class HelloSpeaker implements IHello {

    @Override
    public void hello(String name) {
        System.out.println("Hello, " + name);

    }

}
```

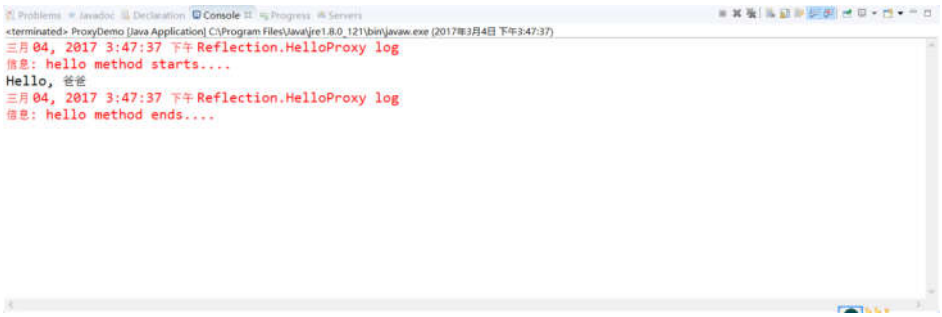
可以看到，在这个类中没有日志记录的代码，其只需要专注于实现业务功能，而记录日志的工作则可以交给代理对象来实现，代理对象也要实现Ihello接口：

- HelloProxy.java

```
package Reflection;

import java.util.logging.*;

public class HelloProxy implements IHello {
    private Logger logger =
            Logger.getLogger(this.getClass().getName());

    private IHello helloObject;

    public HelloProxy(IHello helloObject) {
        this.helloObject = helloObject;
    }

    public void hello(String name) {
        // 日誌服務
        log("hello method starts....");

        // 執行商務邏輯
        helloObject.hello(name);

        // 日誌服務
        log("hello method ends....");
    }

    private void log(String msg) {
        logger.log(Level.INFO, msg);
    }
}
```

我们可以看到在hello方法的实现中，前后插入了日志记录的方法。
下面我们就测试一下

```
public class ProxyDemo {
    public static void main(String[] args) {
        IHello proxy =
            new HelloProxy(new HelloSpeaker());
        proxy.hello("Justin");
    }
}
```



Paste_Image.png

程序中执行hello方法的是代理对象，实例化代理对象的时候，必须传入被代理对象，而且声明代理对象的时候，必须使用代理对象和被代理对象共同实现的接口，以便实现多态。

代理对象将代理真正执行hello方法的被代理对象来执行hello,并在执行的前后加入日志记录的操作这样就可以使业务代码专注于业务实现。

这就是静态代理

动态代理

jdk1.3加入了动态代理相关的API，从上面静态代理的例子我们知道，静态代理，需要为被代理对象和方法实现撰写特定的代理对象，显然这样做并不灵活，我们希望可以有一个公用的代理，可以动态的实现对不同对象的代理，这就需要利用到反射机制和动态代理机制。
在动态代理中，一个handler可以代理服务各种对象，首先，每一个handler都必须继承实现java.lang.reflect.InvocationHandler接口，下面具体实例说明，依然是上面那个记录日志的例子

- LogHandler.java

```
package Reflection;

import java.util.logging.*;
import java.lang.reflect.*;

public class LogHandler implements InvocationHandler {
    private Logger logger =
            Logger.getLogger(this.getClass().getName());

    private Object delegate;

    public Object bind(Object delegate) {
        this.delegate = delegate;
        return Proxy.newProxyInstance(
                            delegate.getClass().getClassLoader(),
                            delegate.getClass().getInterfaces(),
                            this);
    }

    public Object invoke(Object proxy, Method method,
                         Object[] args) throws Throwable {
        Object result = null;

        try {
            log("method starts..." + method);

            result = method.invoke(delegate, args);

            logger.log(Level.INFO, "method ends..." + method);
        } catch (Exception e){
            log(e.toString());
        }

        return result;
    }

    private void log(String message) {
        logger.log(Level.INFO, message);
    }
}
```

具体来说就是使用Proxy.newProxyInstance()静态方法new一个代理对象出来，底层会使用反射机制，建立代理对象的时候，需要传入被代理对象的class，以及被代理对象的所实现的接口，以及代理方法调用的调用程序 InvocationHandler，即实现InvocationHandler接口的对象。这个对象会返回一个指定类指定接口，指定InvocationHandler的代理类实例，这个实例执行方法时，每次都会调用InvocationHandler的invoke方法，invoke方法会传入被代理对象的方法与方法参数，实际方法的执行会交给method.invoke().所以我们就可以在其前后加上日志记录的工作。

接下来我们就来测试一下，使用logHandler的bind方法来绑定代理对象：

```
package Reflection;

import java.lang.reflect.Proxy;

public class ProxyDemo {

    public static void main(String[] args) {


            LogHandler logHandler  = new LogHandler();

            IHello helloProxy =
                    (IHello) logHandler.bind(new HelloSpeaker());
            helloProxy.hello("baba");

    }

}
```

Java (/nb/10012174)

六尺帐篷 (/u/f8e9b1c246f1)
写了 245418 字，被 15240 人关注，获得了 1429 个喜欢
(/u/f8e9b1c246f1)