Spring 容器AOP的实现原理——动态代理

IOC负责将对象动态的注入到容器，从而达到一种需要谁就注入谁，什么时候需要就什么时候注入的效果，可谓是招之则来，挥之则去。

而AOP呢，它实现的就是容器的另一大好处了，就是可以让容器中的对象都享有容器中的公共服务。

那么容器是怎么做到的呢？它怎么就能让在它里面的对象自动拥有它提供的公共性服务呢？答案就是我们今天要讨论的内容——动态代理。

动态代理其实并不是什么新鲜的东西，学过设计模式的人都应该知道代理模式，代理模式是一种静态代理，而动态代理就是利用反射和动态编译将代理模式变成动态的。原理跟动态注入一样，代理模式在编译的时候就已经确定代理类将要代理谁，而动态代理在运行的时候才知道自己要代理谁。

Spring的动态代理有两种：一是JDK的动态代理；另一个是cglib动态代理（通过修改字节码来实现代理）。

今天咱们主要讨论JDK动态代理的方式。
JDK的代理方式主要就是通过**反射跟动态编译**来实现的，
下面咱们就通过代码来看看它具体是怎么实现的。

假设我们要对下面这个用户管理进行代理：

```
//用户管理接口
package com.tgb.proxy;

public interface UserMgr {
    void addUser();
    void delUser();
}

//用户管理的实现
package com.tgb.proxy;

public class UserMgrImpl implements UserMgr {

    @Override
    public void addUser() {
        System.out.println("添加用户.....");
    }

    @Override
    public void delUser() {
        System.out.println("删除用户.....");
    }

}
```

按照代理模式的实现方式，肯定是用一个代理类，让它也实现UserMgr接口，然后在其内部声明一个UserMgrImpl，然后分别调用addUser和delUser方法，并在调用前后加上我们需要的其他操作。但是这样很显然都是写死的，我们怎么做到动态呢？别急，接着看。我们知道，要实现代理，那么我们的代理类跟被代理类都要实现同一接口，但是动态代理的话我们根本不知道我们将要代理谁，也就不知道我们要实现哪个接口，那么要怎么办呢？我们只有知道要代理谁以后，才能给出相应的代理类，那么我们何不等知道要代理谁以后再去生成一个代理类呢？想到这里，我们好像找到了解决的办法，就是动态生成代理类！

这时候我们亲爱的反射又有了用武之地，我们可以写一个方法来接收被代理类，这样我们就可以通过反射知道它的一切信息——包括它的类型、它的方法等等（如果你不知道怎么得到，请先去看看我写的反射的博客《反射一》《反射二》）。

JDK动态代理的两个核心分别是InvocationHandler和Proxy，
下面我们就用简单的代码来模拟一下它们是怎么实现的：

InvocationHandler接口：

```
package com.tgb.proxy;

import java.lang.reflect.Method;

public interface InvocationHandler {
    public void invoke(Object o, Method m);
}
```

实现动态代理的关键部分，通过Proxy动态生成我们具体的代理类：

```
package com.tgb.proxy;

import java.io.File;
import java.io.FileWriter;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.net.URL;
import java.net.URLClassLoader;
import javax.tools.JavaCompiler;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;
import javax.tools.JavaCompiler.CompilationTask;

public class Proxy {
    /**
     *
     * @param infce 被代理类的接口
     * @param h 代理类
     * @return
     * @throws Exception
     */
    public static Object newProxyInstance(Class infce, InvocationHandler h) throws
```

```
Exception {
        String methodStr = "";
        String rt = "\r\n";

        //利用反射得到infce的所有方法，并重新组装
        Method[] methods = infce.getMethods();
        for(Method m : methods) {
            methodStr += "    @Override" + rt +
                         "    public  "+m.getReturnType()+" " + m.getName() + "() {" +
rt +
                         "        try {" + rt +
                         "        Method md = " + infce.getName() +
".class.getMethod(\"" + m.getName() + "\");" + rt +
                         "        h.invoke(this, md);" + rt +
                         "        }catch(Exception e) {e.printStackTrace();}" + rt +
                         "    }" + rt ;
        }

        //生成Java源文件
        String srcCode =
            "package com.tgb.proxy;" +  rt +
            "import java.lang.reflect.Method;" + rt +
            "public class $Proxy1 implements " + infce.getName() + "{" + rt +
            "    public $Proxy1(InvocationHandler h) {" + rt +
            "        this.h = h;" + rt +
            "    }" + rt +
            "    com.tgb.proxy.InvocationHandler h;" + rt +
            methodStr + rt +
            "}";

        String fileName =
            "d:/src/com/tgb/proxy/$Proxy1.java";
        File f = new File(fileName);
        FileWriter fw = new FileWriter(f);
        fw.write(srcCode);
        fw.flush();
        fw.close();

        //将Java文件编译成class文件
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        StandardJavaFileManager fileMgr = compiler.getStandardFileManager(null, null,
null);
        Iterable units = fileMgr.getJavaFileObjects(fileName);
        CompilationTask t = compiler.getTask(null, fileMgr, null, null, null, units);
        t.call();
        fileMgr.close();
```

```
        //加载到内存，并实例化
        URL[] urls = new URL[] {new URL("file:/" + "d:/src/")};
        URLClassLoader ul = new URLClassLoader(urls);
        Class c = ul.loadClass("com.tgb.proxy.$Proxy1");

        Constructor ctr = c.getConstructor(InvocationHandler.class);
        Object m = ctr.newInstance(h);

        return m;
    }

}
```

这个类的主要功能就是，根据被代理对象的信息，动态组装一个代理类，生成$Proxy1.java文件，然后将其编译成$Proxy1.class。
这样我们就可以在运行的时候，根据我们具体的被代理对象生成我们想要的代理类了。
这样一来，我们就不需要提前知道我们要代理谁。也就是说，你想代理谁，想要什么样的代理，我们就给你生成一个什么样的代理类。
然后，在客户端我们就可以随意的进行代理了。

```
package com.tgb.proxy;

public class Client {
    public static void main(String[] args) throws Exception {
        UserMgr mgr = new UserMgrImpl();

        //为用户管理添加事务处理
        InvocationHandler h = new TransactionHandler(mgr);
        UserMgr u = (UserMgr)Proxy.newProxyInstance(UserMgr.class,h);

        //为用户管理添加显示方法执行时间的功能
        TimeHandler h2 = new TimeHandler(u);
        u = (UserMgr)Proxy.newProxyInstance(UserMgr.class,h2);

        u.addUser();
        System.out.println("\r\n==========华丽的分割线==========\r\n");
        u.delUser();
    }
}
```

运行结果：

```
开始时间:2014年-07月-15日 15时:48分:54秒
开启事务.....
添加用户.....
提交事务.....
结束时间:2014年-07月-15日 15时:48分:57秒
耗时：3秒
```

```
==========华丽的分割线==========
```

```
开始时间:2014年-07月-15日 15时:48分:57秒
开启事务.....
删除用户.....
提交事务.....
结束时间:2014年-07月-15日 15时:49分:00秒
耗时：3秒
```

这里我写了两个代理的功能，一个是事务处理，一个是显示方法执行时间的代理，当然都是非常简单的写法，只是为了说明这个原理。当然，我们可以想Spring那样将这些AOP写到配置文件，因为之前那篇已经写了怎么通过配置文件注入了，这里就不重复贴了。到这里，你可能会有一个疑问：你上面说，只要放到容器里的对象，都会有容器的公共服务，我怎么没看出来呢？好，那我们就继续看一下我们的代理功能：

事务处理：

```
package com.tgb.proxy;

import java.lang.reflect.Method;

public class TransactionHandler implements InvocationHandler {

    private Object target;

    public TransactionHandler(Object target) {
        super();
        this.target = target;
    }

    @Override
    public void invoke(Object o, Method m) {
        System.out.println("开启事务.....");
        try {
            m.invoke(target);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("提交事务.....");
    }

}
```

从代码中不难看出，我们代理的功能里没有涉及到任何被代理对象的具体信息，这样有什么好处呢？这样的好处就是将代理要做的事情跟被代理的对象完全分开，这样一来我们就可以在代理和被代理之间随意的进行组合了。也就是说同一个功能我们只需要一个。同样的功能只有一个，那么这个功能不就是公共的功能吗？不管容器中有多少给对象，都可以享受容器提供的服务了。这就是容器的好处。

不知道我讲的够不够清楚，欢迎大家积极交流、讨论。