

roki的专栏

□□

乐在自由谷

空间

博客

好友

相册

留言

深入理解Java Proxy机制 收藏

动态代理其实就是java.lang.reflect.Proxy类动态的根据您指定的所有接口生成一个class byte，该class会继承Proxy类，并实现所有你指定的接口（您在参数中传入的接口数组）；然后再利用您指定的classloader将 class byte加载进系统，最后生成这样一个类的对象，并初始化该对象的一些值，如invocationHandler,以即所有的接口对应的Method成员。初始化之后将对象返回给调用的客户端。这样客户端拿到的就是一个实现你所有的接口的Proxy对象。请看实例分析：

一 业务接口类

```
public interface BusinessProcessor {  
    public void processBusiness();  
}
```

二 业务实现类

```
public class BusinessProcessorImpl implements BusinessProcessor {  
    public void processBusiness() {  
        System.out.println("processing business.....");  
    }  
}
```

三 业务代理类

```
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
  
public class BusinessProcessorHandler implements InvocationHandler {  
  
    private Object target = null;  
  
    BusinessProcessorHandler(Object target){  
        this.target = target;  
    }  
  
    public Object invoke(Object proxy, Method method, Object[] args)  
        throws Throwable {
```

用户操作

[留言] [发消息] [加为好友]

订阅我的博客

0 位读者
POWERED BY FEEDSKY

订阅

+ 订阅到 鲜果

+ 订阅到 Google

+ 订阅到 抓虾

rokii的公告

初来乍道！！

文章分类



batch



html.CSS.JAV

ASCRIP



JASE



MYSQL



ORACLE



WebService



搜索引擎



综合

搜索检索

逍遥谷

存档

```

System.out.println("You can do something here before process your business");
Object result = method.invoke(target, args);
System.out.println("You can do something here after process your business");
return result;
}

}

```

四 客户端应用类

```

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Proxy;

public class Test {

    public static void main(String[] args) {
        BusinessProcessorImpl bpimpl = new BusinessProcessorImpl();
        BusinessProcessorHandler handler = new BusinessProcessorHandler(bpimpl);
        BusinessProcessor bp = (BusinessProcessor)Proxy.newProxyInstance(bpimpl.getClass().getClassLoader(), bpimpl.getClass().getInterfaces(), handler);
        bp.processBusiness();
    }
}

```

现在我们看一下打印结果：

```

You can do something here before process your business
processing business.....
You can do something here after process your business

```

通过结果我们就能够很简单的看出Proxy的作用了，它能够在你的核心业务方法前后做一些你所想做的辅助工作，如log日志，安全机制等等。

现在我们来分析一下上面的类的工作原理。

类一二没什么好说的。先看看类三吧。 实现了InvocationHandler接口的invoke方法。其实这个类就是最终Proxy调用的固定接口方法。Proxy不管客户端的业务方法是怎么实现的。当客户端调用Proxy时，它只

会调用InvocationHandler的invoke接口，所以我们的真正实现的方法就必须在invoke方法中去调用。关系如下：

```

BusinessProcessorImpl bpimpl = new BusinessProcessorImpl();

```

2010年01月(1)

2009年10月(2)

2009年09月(2)

2009年08月(3)

2009年06月(3)

2009年04月(1)

2009年02月(1)

2008年12月(2)

2008年04月(1)

2008年03月(6)

```
BusinessProcessorHandler handler = new BusinessProcessorHandler(bpimpl);

BusinessProcessor bp = (BusinessProcessor)Proxy.newProxyInstance(...);

bp.processBusiness()-->invocationHandler.invoke()-->bpimpl.processBusiness();
```

那么bp到底是怎么样一个对象呢。我们改一下main方法看一下就知道了：

```
public static void main(String[] args) {
    BusinessProcessorImpl bpimpl = new BusinessProcessorImpl();
    BusinessProcessorHandler handler = new BusinessProcessorHandler(bpimpl);
    BusinessProcessor bp = (BusinessProcessor)Proxy.newProxyInstance(bpimpl.getClass().getClassLoader(), bpimpl.getClass().getInterfaces(), handler);
    bp.processBusiness();
    System.out.println(bp.getClass().getName());
}
```

输出结果：

You can do something here before process your business
processing business.....

You can do something here after process your business
\$Proxy0

bp原来是个\$Proxy0这个类的对象。那么这个类到底是长什么样子呢？好的。我们再写二个方法去把这个类打印出来看个究竟，是什么三头六臂呢？我们在main下面写如下两个静态方法。

```
public static String getModifier(int modifier){
    String result = "";
    switch(modifier){
        case Modifier.PRIVATE:
            result = "private";
        case Modifier.PUBLIC:
            result = "public";
        case Modifier.PROTECTED:
            result = "protected";
        case Modifier.ABSTRACT :
            result = "abstract";
        case Modifier.FINAL :
            result = "final";
        case Modifier.NATIVE :
            result = "native";
        case Modifier.STATIC :
            result = "static";
        case Modifier.SYNCHRONIZED :
            result = "synchronized";
        case Modifier.STRICT :
```

```

        result = "strict";
    case Modifier.TRANSIENT :
        result = "transient";
    case Modifier.VOLATILE :
        result = "volatile";
    case Modifier.INTERFACE :
        result = "interface";
    }
    return result;
}

public static void printClassDefinition(Class clz){

    String clzModifier = getModifier(clz.getModifiers());
    if(clzModifier!=null && !clzModifier.equals("")){
        clzModifier = clzModifier + " ";
    }
    String superClz = clz.getSuperclass().getName();
    if(superClz!=null && !superClz.equals("")){
        superClz = "extends " + superClz;
    }

    Class[] interfaces = clz.getInterfaces();

    String inters = "";
    for(int i=0; i<interfaces.length; i++){
        if(i==0){
            inters += "implements ";
        }
        inters += interfaces[i].getName();
    }

    System.out.println(clzModifier +clz.getName()+" " + superClz +" " + inters );
    System.out.println("{");

    Field[] fields = clz.getDeclaredFields();
    for(int i=0; i<fields.length; i++){
        String modifier = getModifier(fields[i].getModifiers());
        if(modifier!=null && !modifier.equals("")){
            modifier = modifier + " ";
        }
        String fieldName = fields[i].getName();
        String fieldType = fields[i].getType().getName();
        System.out.println("    "+modifier + fieldType + " "+ fieldName + ";");
    }

    System.out.println();
}

```

```

Method[] methods = clz.getDeclaredMethods();
for(int i=0; i<methods.length; i++){
    Method method = methods[i];

    String modifier = getModifier(method.getModifiers());
    if(modifier!=null && !modifier.equals("")){
        modifier = modifier + " ";
    }

    String methodName = method.getName();

    Class returnClz = method.getReturnType();
    String retrunType = returnClz.getName();

    Class[] clzs = method.getParameterTypes();
    String paraList = "(";
    for(int j=0; j<clzs.length; j++){
        paraList += clzs[j].getName();
        if(j != clzs.length - 1 ){
            paraList += ", ";
        }
    }
    paraList += ")";

    clzs = method.getExceptionTypes();
    String exceptions = "";
    for(int j=0; j<clzs.length; j++){
        if(j==0){
            exceptions += "throws ";
        }

        exceptions += clzs[j].getName();

        if(j != clzs.length - 1 ){
            exceptions += ", ";
        }
    }

    exceptions += ";";

    String methodPrototype = modifier + retrunType+" "+methodName+paraList+
exceptions;

    System.out.println("    "+methodPrototype );

}

```

```
System.out.println("{}");  
}
```

再改写main方法

```
public static void main(String[] args) {  
    BusinessProcessorImpl bpimpl = new BusinessProcessorImpl();  
    BusinessProcessorHandler handler = new BusinessProcessorHandler(bpimpl);  
    BusinessProcessor bp = (BusinessProcessor)Proxy.newProxyInstance(bpimpl.get  
Class().getClassLoader(), bpimpl.getClass().getInterfaces(), handler);  
    bp.processBusiness();  
    System.out.println(bp.getClass().getName());  
    Class clz = bp.getClass();  
    printClassDefinition(clz);  
}
```

现在再看看输出结果：

You can do something here before process your business
processing business.....

You can do something here after process your business

\$Proxy0

\$Proxy0 extends java.lang.reflect.Proxy implements com.tom.proxy.dynamic.Busi
nessProcessor

```
{  
    java.lang.reflect.Method m4;  
    java.lang.reflect.Method m2;  
    java.lang.reflect.Method m0;  
    java.lang.reflect.Method m3;  
    java.lang.reflect.Method m1;  
  
    void processBusiness();  
    int hashCode();  
    boolean equals(java.lang.Object);  
    java.lang.String toString();  
}
```

很明显，Proxy.newProxyInstance方法会做如下几件事：

1，根据传入的第二个参数interfaces动态生成一个类，实现interfaces中的接口，该例中即BusinessProcessor接口的processBusiness方法。并且继承了Proxy类，重写了hashCode,toString,equals等三个方法。具体实现可参看 ProxyGenerator.generateProxyClass(...); 该例中生成了\$Proxy0类

2，通过传入的第一个参数classloder将刚生成的类加载到jvm中。即将\$Proxy0类load

3，利用第三个参数，调用\$Proxy0的\$Proxy0(InvocationHandler)构造函数 创建\$Proxy0的对象，并且用interfaces参数遍历其所有接口的方法，并生成Method对象初始化对象的几个Method成员变量

4，将\$Proxy0的实例返回给客户端。

现在好了。我们再看客户端怎么调就清楚了。

1，客户端拿到的是\$Proxy0的实例对象，由于\$Proxy0继承了BusinessProcessor，因此转化为BusinessProcessor没有任何问题。

BusinessProcessor bp = (BusinessProcessor)Proxy.newProxyInstance(...);

2，bp.processBusiness();

实际上调用的是\$Proxy0.processBusiness();那么\$Proxy0.processBusiness()的实现就是通过InvocationHandler去调用invoke方法啦！


发表于 @ 2009年04月03日 12:34:00 | [评论\(1\)](#) | [举报](#) | [收藏](#)

旧一篇:[java 常用正则表达式](#) | 新一篇:[MySQL 6.0 免安装版配置](#)

[Proxy. Economy.](#)
Traffic Inspector Cuts banners, filters sites, files
[www.trafinsp.com](#)

[笔记本电脑 特价680元](#)
出售特价笔记本电脑 手机等 销售电话姐
[www.kl666888.co.to](#)

[lanyin1111](#) □□□Tuesday, October 13, 2009 10:53:18 □□ □□

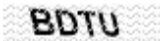
 □□ □□□□ □□

[发表评论](#)

表情:          

评论内容:

用户 名: 匿名用户 [登录](#) [注册](#)

验 证 码:  [重新获得验证码](#)

 **绿森数码 分期付款购物**
www.green3c.com

THINKPAD SL400 2743-2NC电脑
(酷睿双核, 2G内存, 独显)

月付350元

火爆



 招商银行
 交通银行
 中国工商银行
 中国银行

立即订购

www.green3c.com Google 提供的广告

Copyright © rokii

Powered by CSDN Blog