代理模式

简单来说就是 **我们使用代理对象来代替对真实对象(real object)的访问，这样就可以在不修改原目标对象的前提下，提供额外的功能操作，扩展目标对象的功能。**

代理模式的主要作用就是扩展目标对象的功能，比如在目标对象的某个方法执行前后，可以增加一些自定义方法操作

举个例子：新娘找来了自己的姨妈来代替自己处理新郎的提问，新娘收到的提问都是**经过姨妈处理过滤之后**的。姨妈在这里就可以看作是代理你的代理对象，代理的行为（方法）是接收和回复新郎的提问。

静态代理

在原有子类的基础上增添功能。缺点是需要实现所有子类，一旦父类增添新的方法，子类也需要做出相应的更改，增加维护成本

动态代理

本篇演示，通过反射机制将当前线程的ClassLoader抽取出来，对需要的方法进行增强操作

相比于静态代理来说，动态代理更加灵活。我们不需要针对每个目标类都单独创建一个代理类，并且也不需要我们必须实现接口，我们可以直接代理实现类(*CGLIB* 动态代理机制)。

从 JVM 角度来说，动态代理是在运行时动态生成类字节码，并加载到 JVM 中的。

说到动态代理，Spring AOP、RPC 框架应该是两个不得不提的，它们的实现都依赖了动态代理。

动态代理在我们日常开发中使用的相对较少，但是在框架中的几乎是必用的一门技术。学会了动态代理之后，对于我们理解和学习各种框架的原理也非常有帮助。

就 Java 来说，动态代理的实现方式有很多种，比如 **JDK 动态代理**、**CGLIB 动态代理**等等。

JDK 动态代理机制

介绍

在 Java 动态代理机制中 `InvocationHandler` 接口和 `Proxy` 类是核心。

`Proxy` 类中使用频率最高的方法是：`newProxyInstance()` ，这个方法主要用来生成一个代理对象。

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

这个方法一共有 3 个参数：

1. **loader** :类加载器，用于加载代理对象。
2. **interfaces** : 被代理类实现的一些接口；
3. **h** : 实现了 `InvocationHandler` 接口的对象；

要实现动态代理的话，还必须需要实现`InvocationHandler` 来自定义处理逻辑。 当我们的**动态代理对象调用一个方法时，这个方法的调用就会被转发到实现`InvocationHandler` 接口类的`invoke` 方法来调用。**

```
public interface InvocationHandler {

    /**
     * 当你使用代理对象调用方法的时候实际会调用到这个方法
     */
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

`invoke()` 方法有下面三个参数：

1. **proxy** :动态生成的代理类
2. **method** : 与代理类对象调用的方法相对应
3. **args** : 当前 method 方法的参数

也就是说：**你通过`Proxy` 类的`newProxyInstance()` 创建的代理对象在调用方法的时候，实际会调用到实现`InvocationHandler` 接口的类的`invoke()`方法。** 你可以在`invoke()` 方法中自定义处理逻辑，比如在方法执行前后做什么事情。

JDK 动态代理类使用步骤

1. 定义一个接口及其实现类；
2. 自定义 `InvocationHandler` 并重写`invoke`方法，在 `invoke` 方法中我们会调用原生方法（被代理类的方法）并自定义一些处理逻辑；
3. 通过 `Proxy.newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)` 方法创建代理对象；

代码示例

这样说可能会有点空洞和难以理解，我上个例子，大家感受一下吧！

1.定义发送短信的接口

```
public interface SmsService {
    String send(String message);
}
```

2.实现发送短信的接口

```
public class SmsServiceImpl implements SmsService {
    public String send(String message) {
        System.out.println("send message:" + message);
        return message;
    }
}
```

3.定义一个 JDK 动态代理类

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * @author shuang.kou
 * @createTime 2020年05月11日 11:23:00
 */
public class DebugInvocationHandler implements InvocationHandler {
    /**
     * 代理类中的真实对象
     */
    private final Object target;

    public DebugInvocationHandler(Object target) {
        this.target = target;
    }


    public Object invoke(Object proxy, Method method, Object[] args) throws InvocationTargetException, IllegalAccessException {
        //调用方法之前，我们可以添加自己的操作
        System.out.println("before method " + method.getName());
        Object result = method.invoke(target, args);
        //调用方法之后，我们同样可以添加自己的操作
        System.out.println("after method " + method.getName());
        return result;
    }
}
```

`invoke()` 方法: 当我们的动态代理对象调用原生方法的时候，最终实际上调用到的是 `invoke()` 方法，然后 `invoke()` 方法代替我们去调用了被代理对象的原生方法。

4.获取代理对象的工厂类

```
public class JdkProxyFactory {
    public static Object getProxy(Object target) {
        return Proxy.newProxyInstance(
                target.getClass().getClassLoader(), // 目标类的类加载
                target.getClass().getInterfaces(),  // 代理需要实现的接口，可指定多个
                new DebugInvocationHandler(target)   // 代理对象对应的自定义InvocationHandler
        );
    }
}
```

`getProxy()`：主要通过`Proxy.newProxyInstance()`方法获取某个类的代理对象

5.实际使用

```
SmsService smsService = (SmsService) JdkProxyFactory.getProxy(new SmsServiceImpl());
smsService.send("java");
```

运行上述代码之后，控制台打印出：

```
before method send
send message:java
after method send
```

静态代理和动态代理的对比

1. **灵活性**：动态代理更加灵活，不需要必须实现接口，可以直接代理实现类，并且可以不需要针对每个目标类都创建一个代理类。另外，静态代理中，接口一旦新增加方法，目标对象和代理对象都要进行修改，这是非常麻烦的！
2. **JVM 层面**：静态代理在编译时就将接口、实现类、代理类这些都变成了一个个**实际的 class 文件**。而动态代理是在**运行时动态生成类字节码，并加载到 JVM 中的**

什么是代理模式

代理模式有点像老大和小弟，也有点像分销商。主要解决的是问题是为某些资源的访问、对象的类的易用操作上提供方便使用的代理服务。而这种设计思想的模式经常会出现在我们的系统中，或者你用到过的组件中，它们都提供给你一种非常简单易用的方式控制原本你需要编写很多代码的进行使用的服务类。

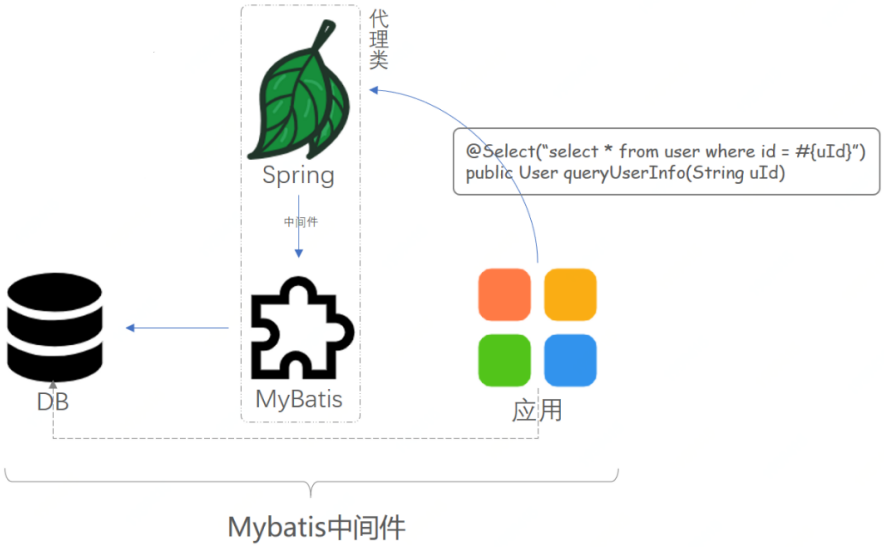
类似这样的场景可以想到；

1. 你的数据库访问层面经常会提供一个较为基础的应用，以此来减少应用服务扩容时不至于数据库连接数暴增。
2. 使用过的一些中间件例如；RPC框架，在拿到jar包对接口的描述后，中间件会在服务启动的时候生成对应的代理类，当调用接口的时候，实际是通过代理类发出的socket信息进行通过。

3. 另外像我们常用的`MyBatis`，基本是定义接口但是不需要写实现类，就可以对`xml`或者自定义注解里的`sql`语句进行增删改查操作。

案例场景模拟

MyBatis-Plus中间件的代理生成部分



代理模式实现

接下来会使用代理类模式来模拟实现一个Mybatis中对类的代理过程，也就是**只需要定义接口，就可以关联到方法注解中的`sql`语句完成对数据库的操作。**

这里需要注意一些知识点；

1. `BeanDefinitionRegistryPostProcessor`，spring的接口类用于处理对bean的定义注册。
2. `GenericBeanDefinition`，定义bean的信息，在mybatis-spring中使用到的是；`ScannedGenericBeanDefinition`略有不同。
3. `FactoryBean`，用于处理bean工厂的类，这个类非常常见

工程代码

项目树：

```
bantanger-demo-design-10-00
└── src
    ├── main
    |   ├── java
    |   |   └── com.bantanger-demo-design
    |   |       ├── agent
    |   |       |   ├── MapperFactoryBean.java
    |   |       |   ├── RegisterBeanFactory.java
    |   |       |   └── Select.java
    |   |       └── IUserDao.java
    |   └── resources
    |       └── spring-config.xml
```

```
    └── test
        └── java
            └── org.itstack.demo.test
                └── ApiTest.java
```

自定义注解：

不明白的可以查一下

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Select {

    /**
     * 当注解类使用的属性名是value是，对其赋值可以不指定属性名称而直接写上属性接口
     * 除了value以外的变量名，在使用注解时都要使用name=value的方式进行赋值
     * @return
     */
    String value() default "";

}
```

定义接口：

模拟实现MyBatis的注解实现方式

```
public interface IUserDao {

    @Select("select userName from user where id = # {uId}")
    String queryUserInfo(String uId);

}
```

代理类：

负责接管调用方法，调用方法实际生成的实例是动态的代理对象

```
public class MapperFactoryBean<T> implements FactoryBean<T> {

    private Logger logger =
LoggerFactory.getLogger(MapperFactoryBean.class);

    private Class<T> mapperInterface; // 接口信息

    public MapperFactoryBean(Class<T> mapperInterface) {
        this.mapperInterface = mapperInterface;
    }

    /**
```

```
     * 实现FactoryBean的getObject方法，它能提供bean对象
     * 在这个方法中提供类的代理和模拟对sql的处理
     *
     * @return
     * @throws Exception
     */
    @Override
    public T getObject() throws Exception {
        // 执行时机：客户端调用IUserDao接口的方法时，被代理类自动接管，增强功能
        // 本质：实际函数调用时，被InvocationHandler拦截，使用重写invoke执行实际函数
        InvocationHandler handler = (proxy, method, args) -> {
            Select select = method.getAnnotation(Select.class); // 通过反射得到注解的内容
            logger.info("SQL: {}", select.value().replace("#{uId}", args[0].toString()));
            return args[0] + ", bantanger 爱奋斗的小鲨鱼，执行SQL完毕";
        };

        // 执行时机：通过getBean得到接口bean时执行
        // 本质：利用代理工厂得到代理类实例，返回给实际调用
        return (T) Proxy.newProxyInstance(
                this.getClass().getClassLoader(),
                new Class[]{mapperInterface},
                handler
        );
    }

    /**
     * 提供对象类型反馈
     * @return
     */
    @Override
    public Class<?> getObjectType() {
        return mapperInterface;
    }

    /**
     * 返回单例对象
     * @return
     */
    @Override
    public boolean isSingleton() {
        return FactoryBean.super.isSingleton();
    }
}
```

- 如果你有阅读过mybatis源码，是可以看到这样的一个类；`MapperFactoryBean`，这里我们也模拟一个这样的类，在里面实现我们对代理类的定义。
- 通过继承`FactoryBean`，提供bean对象，也就是方法；`T getObject()`。
- 在方法`getObject()`中提供类的代理以及模拟对sql语句的处理，这里包含了用户调用dao层方法时候的处理逻辑。
- 还有最上面我们提供构造函数来透传需要被代理类，`Class<T> mapperInterface`，在mybatis中也是使用这样的方式进行透传。
- 另外`getObjectType()`提供对象类型反馈，以及`isSingleton()`返回类是单例的。

将Bean注册到spring中：

```
public class RegisterBeanFactory implements
BeanDefinitionRegistryPostProcessor {

    /**
     * 将对象交给spring容器管理，可以很方便的获取到代理bean
     *
     * spring中将bean注册的过程
     * @param registry
     * @throws BeansException
     */
    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
registry) throws BeansException {

        GenericBeanDefinition beanDefinition = new GenericBeanDefinition();
        beanDefinition.setBeanClass(MapperFactoryBean.class); // 定义交给
spring管理的bean属于哪一个类
        beanDefinition.setScope("singleton"); //
        beanDefinition.getConstructorArgumentValues() // 获取这个bean的构造函
数值(mapperInterface)
                .addGenericArgumentValue(IUserDao.class); // 将值传递到接口
IUserDao中

        BeanDefinitionHolder definitionHolder =
                new BeanDefinitionHolder(beanDefinition, "userDao");
        BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder,
registry);

    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
configurableListableBeanFactory) throws BeansException {

    }
}
```

- 这里我们将代理的bean交给spring容器管理，也就可以非常方便让我们可以获取到代理的bean。这部分是spring中关于一个bean注册过程的源码。
- `GenericBeanDefinition`，用于定义一个bean的基本信息`setBeanClass(MapperFactoryBean.class);`，也包括可以透传给构造函数信息`addGenericArgumentValue(IUserDao.class);`
- 最后使用`BeanDefinitionReaderUtils.registerBeanDefinition`，进行bean的注册，也就是注册到`DefaultListableBeanFactory`中。

配置文件spring-config：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
       default-autowire="byName">

    <bean id="userDao"
class="org.itstack.demo.design.agent.RegisterBeanFactory"/>

</beans>
```

- 接下来在配置文件中添加我们的bean配置，在mybatis的使用中一般会配置扫描的dao层包，这样就可以减少这部分的配置。

测试类实现：

```java
class RegisterBeanFactoryTest {

    private Logger logger =
LoggerFactory.getLogger(MapperFactoryBean.class);

    @Test
    public void test_IUserDao() {
        BeanFactory beanFactory = new
ClassPathXmlApplicationContext("spring-config.xml");
        IUserDao userDao = beanFactory.getBean("userDao", IUserDao.class);
        String res = userDao.queryUserInfo("100001");
        logger.info("测试结果：{}", res);
    }

}
```

总结

- 关于这部分代理模式的讲解我们采用了开发一个关于`mybatis-spring`中间件中部分核心功能来体现代理模式的强大之处，所以涉及到了一些关于代理类的创建以及spring中bean的注册这些知识点，可能在平常的业务开发中都是很少用到的，但是在中间件开发中确实非常常见的操作。
- 代理模式除了开发中间件外还可以是对服务的包装，物联网组件等等，让复杂的各项服务变为**轻量级调用、缓存使用**。你可以理解为你家里的电灯开关，我们不能操作220v电线的人肉连接，但是可以使用开关，避免触电。
- 代理模式的设计方式可以让代码更加整洁、干净易于维护，虽然在这部分开发中额外增加了很多类也包括了自己处理bean的注册等，但是这样的中间件复用性极高也更加智能，可以非常方便的扩展到各个服务应用中。

参考：

静态代理和动态代理

JavaGuide-java代理模式详解

Java自定义注解中关于string[] value() default {};的理解

小傅哥重学java设计模式-代理模式

代理 3 动态代理