链滴

静态代理、动态代理基础

作者：someone36976
原文链接：https://ld246.com/article/1544757534705
来源网站：链滴

静态代理

静态代理也可以看做是代理模式，屏蔽了对真实对象的感知，处理或者调用方法时无需真实对象，只与代理对象交互即可。主要用于由于安全或者其他原因隐藏真实访问对象，或者是为了提升性能对真对象进行延迟加载等操作。

代码实例如下：

- 定义一个接口

```
/**
 * 账户接口
 */
public interface Account {

    /**
     * 查询账户
     */
    void query();

    /**
     * 更新账户
     */
    void update();
}
```

- 实现接口方法

```
public class AccountImpl implements Account {

    static {
```

```
        System.out.println("初始化操作");
    }

    @Override
    public void query() {
        System.out.println("查询账户余额");
    }

    @Override
    public void update() {
        System.out.println("更新账户余额");
    }
}
```

- 定义一个代理对象（请注意代码中 getAccount()方法）

```
public class AccountProxy implements Account {

    private AccountImpl account;

    public AccountImpl getAccount() {
        if (account == null) {
            this.account = new AccountImpl();
        }
        return account;
    }

    @Override
    public void query() {
        System.out.println("查询操作预处理");
        // 调用真正的查询账户方法
        getAccount().query();
        System.out.println("查询操作后回调");
    }

    @Override
    public void update() {
        System.out.println("更新操作预处理");
        // 调用真正的更新账户方法
        getAccount().update();
        System.out.println("更新操作后回调");
    }
}
```

- 调用方法

```
public class Main {
    public static void main(String[] args) {
        AccountProxy accountProxy = new AccountProxy();
        accountProxy.query();
        accountProxy.update();
    }
}
```

动态代理

与静态代理不同，动态代理实在运行时动态生成代理类。不需要像静态代理一样为每一个接口写一个理方法

或者对象，更方便而且利于维护。

动态代理的实现技术有很多，JDK自带动态处理，CGLIB和ASM库等也都提供了相关的处理方法。

JDK

- 定义一个接口

```
public interface TaskService {
    void complete();
}
```

- 实现接口方法

```
public class TaskServiceImpl implements TaskService {

    static {
        System.out.println("初始化操作");
    }

    @Override
    public void complete() {
        System.out.println("完成任务");
    }
}
```

- 代理实现

```
public class TaskProxy implements InvocationHandler {

    private Object target;

    public Object bind(Object target) {
        this.target = target;
        //通过反射机制，创建一个代理类对象实例并返回。用户进行方法调用时使用
        //创建代理对象时，需要传递该业务类的类加载器（用来获取业务实现类的元数据，在包装方法调用真正的业务方法）、接口、handler实现类
        return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result;
        System.out.println("预处理操作");
        result = method.invoke(target, args);
        System.out.println("回调操作");
        return result;
    }
```

原文链接：静态代理、动态代理基础

```
}
```

- 运行

```
public class Main {

    public static void main(String[] args) {

        TaskServiceImpl taskServiceImpl = new TaskServiceImpl();
        TaskProxy proxy = new TaskProxy();
        TaskService taskService = (TaskService) proxy.bind(taskServiceImpl);
        taskService.complete();
    }
}
```

CGLIB

由于该种实现是基于extend的方式，所以对于final类型的方法无法进行代理执行

- 方法定义

```
public class ProcessService {

    public void start() {
        System.out.println("流程启动处理");
    }
}
```

- 代理

```
public class ProcessServiceAgent implements MethodInterceptor {

    private Object target;

    public Object getInstance(Object object) {
        this.target = object;

        Enhancer enhancer = new Enhancer();
        //为加强器指定要代理的业务类（即：为下面生成的代理类指定父类）
        enhancer.setSuperclass(this.target.getClass());
        //设置回调：对于代理类上所有方法的调用，都会调用CallBack，而Callback则需要实现interce
t()方法进行拦
        enhancer.setCallback(this);
        return enhancer.create();
    }

    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodP
oxy) throws Throwable {
        System.out.println("预处理操作");

        methodProxy.invokeSuper(o, objects);

        System.out.println("回调操作");
```

```
        return null;
    }
}
```

- 运行

```
public class Main {

    public static void main(String[] args) {
        ProcessService processService = new ProcessService();
        ProcessServiceAgent processServiceAgent = new ProcessServiceAgent();
        ProcessService instance = (ProcessService) processServiceAgent.getInstance(processServ
ce);
        instance.start();
    }
}
```

其他

常用的应用场景

- 远程代理

```
  <!--也就是为一个对象在不同的地址空间提供局部代表，这样可以隐藏一个对象存在于不同地址空
的事实。比如说 WebService，当我们在应用程序的项目中加入一个 Web 引用，引用一个 WebServi
e，此时会在项目中声称一个 WebReference 的文件夹和一些文件，这个就是起代理作用的，这样可
让那个客户端程序调用代理解决远程访问的问题；-->
```

- 虚拟代理

```
  <!--是根据需要创建开销很大的对象，通过它来存放实例化需要很长时间的真实对象。这样就可以
到性能的最优化，比如打开一个网页，这个网页里面包含了大量的文字和图片，但我们可以很快看到
字，但是图片却是一张一张地下载后才能看到，那些未打开的图片框，就是通过虚拟代里来替换了真
的图片，此时代理存储了真实图片的路径和尺寸；-->
```

- 安全代理

```
  <!--用来控制真实对象访问时的权限。一般用于对象应该有不同的访问权限的时候；-->
```

- 指针引用

```
 <!-- 是指当调用真实的对象时，代理处理另外一些事。比如计算真实对象的引用次数，这样当该对
没有引用时，可以自动释放它，或当第一次引用一个持久对象时，将它装入内存，或是在访问一个实
对象前，检查是否已经释放它，以确保其他对象不能改变它。这些都是通过代理在访问一个对象时附
一些内务处理；-->
```

- 延迟加载

```
  <!--用代理模式实现延迟加载的一个经典应用就在 Hibernate 框架里面。当 Hibernate 加载实体 be
n 时，并不会一次性将数据库所有的数据都装载。默认情况下，它会采取延迟加载的机制，以提高系
的性能。Hibernate 中的延迟加载主要分为属性的延迟加载和关联表的延时加载两类。实现原理是使
代理拦截原有的 getter 方法，在真正使用对象数据时才去数据库或者其他第三方组件加载实际的数
，从而提升系统性能。-->
```

文档参考

- 代理模式原理及实例讲解
- Java动态代理之JDK实现和CGlib实现

原文链接：静态代理、动态代理基础