装饰者设计模式以及代理设计模式对比以及详解

**1、装饰者模式与代理模式 (静态代理)**

　在日常开发里面，我们经常需要给某个类的方法增加加某些特定的功能。

       例如：有婴儿，婴儿会吃饭和走动，如以下类

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 package com.scl.designpattern.proxy;

2

3 //婴儿类

4 public class Child implements Human

5 {

6 public void eat()

7 {

8 System.out.println("eat something....");

9 }

10

11 @Override

12 public void run()

13 {

14 System.out.println("Child run very slow");

15 }

16 }

[复制代码](javascript:void(0);)

　　突然有一天，家长发现不行，孩子不能随便吃东西，而且吃饭前一定要洗手。但是孩子太小(被委托方)，不会自己洗手。家长(Client 端)又没法照顾孩子。那简单，找个保姆照顾孩子！ 让保姆类和婴儿类共同实现同一个接口，让保姆类全程管理小孩，同时在家长眼里，只要看到保姆在帮孩子洗手就可以了。于是，有以下内容。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 保姆类

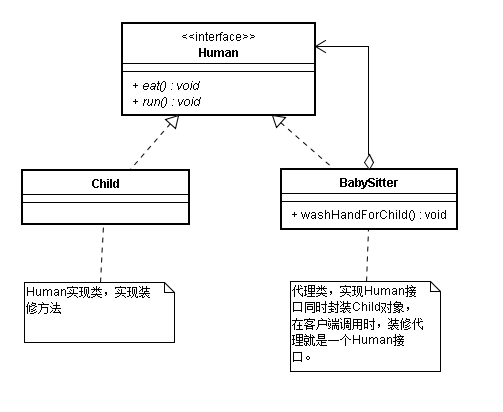
     现在保姆已经有了，孩子也有了，怎么把孩子跟保姆关联起来。让保姆给相应的孩纸洗手。于是保姆类更改如下

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 保姆与婴儿类关联

    好，那么家长就是给孩纸找了个保姆代理，让他附加了一些婴儿做不了事。同时家长也没有强迫孩纸自己学会洗手(不更改Child类)

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 家长客户端代码

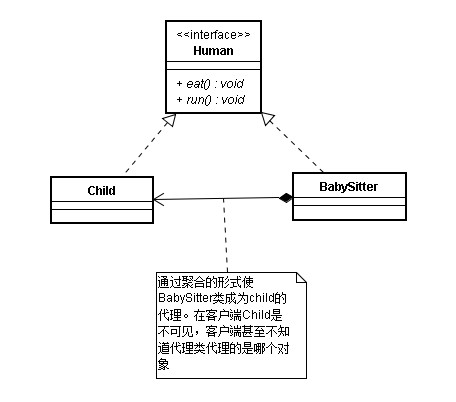
　　以上就是一个简单的装饰模式，来看一下这一块完整的类图。



　　装饰模式的一个很重要特点就是，在客户端可以看到抽象对象的实例，如Human human = new BabySitter(new Child()); 因为装饰模式通过聚合方式，把内容整合到装饰类里面了。

　   装饰者模式能够使用装饰类对抽象对象进行装饰。假如来了个OldMan类手脚不利索。保姆类BabySitter同样能够胜任这个OldMan的饭前洗手操作。

　　例子想了很久，也看了不少别人的博客。发现一个很令人迷惑的问题：为什么我用想的代理模式，确是别人口中的装饰模式？装饰者模式和代理模式有什么区别？。大致的代理模式类图如下：

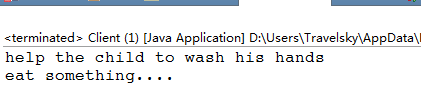


　　由该类图可知，以上BabySitter代码应该如下：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 代理模式下的代理类

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 代理模式下的客户端

 装饰者模式和代理模式的最后运行的结果都是一样的，显示如下



 由代理模式代码可知，客户端不关心代理类了哪个类。但代码控制了客户端对委托类的访问。客户端代码表现为 Human human = new BabySitter( );

 所以资料上都说了，装饰模式主要是强调对类中代码的拓展，而代理模式则偏向于委托类的访问限制。两者都一样拥有抽象角色（接口）、真实角色（委托类）、代理类 。

 由于代理类实现了抽象角色的接口，导致代理类无法通用。如有天，一个有钱人养了只小猩猩，他要一个保姆在猩猩吃东西前，帮猩猩洗手....保姆根本不懂猩猩的特性（跟猩猩类不是同一类型的，保姆属于Human类，而猩猩可能属于Animal类型。），但洗手这个方法是不变的，用水洗。能不能找一个代理说既可以照看人吃饭前洗手也可以照看猩猩吃饭前洗手？

用机器人吧... (编不下去了）。要实现这种功能，必须让代理类与特定的接口分离。在代理的时候能够了解每个委托的特性，这就有可能了。这时候动态代理就出现了。

**2、动态代理**

**2.1 基于继承接口类的动态代理**

    如静态代理的内容所描述的，静态代理受限于接口的实现。动态代理就是通过使用反射，动态地获取抽象接口的类型，从而获取相关特性进行代理。因动态代理能够为所有的委托方进行代理，因此给代理类起个通用点的名字HealthHandle。先看保姆类可以变成什么样。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 由保姆类转化而来的动态代理类

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 动态代理模式下的客户端代码

　　首先，使用动态代理

       1. 必须实现InvocationHandler接口，订立一个契约标识该类为一个动态代理执行类。

       2. InvocationHandler接口内有一实现方法签名如下： public Object invoke(Object proxy, Method method, Object[] args) 。使用时需要重写这个方法

       3. 获取代理类，需要使用 Proxy.newProxyInstance(Clas loader, Class<?>[] interfaces, InvocationHandler h) 这个方法去获取Proxy对象（Proxy类类型的实例，非Child类类型实例）。

　　  先看获取代理对象签名解析如下：

//Clas loader : 类的加载器

　 //Class<?>[] interfaces : 委托类实现的接口，保证代理类返回的是同一个实现接口下的类型，保持代理类与抽象角色行为的一致

//invocationHandler, 该类最重要的一环。一个设计模式：策略模式.即告诉代理类，代理类遇到某个委托类的方法时该调用哪个类下的invoke方法。

Proxy.newProxyInstance(Class loader, Class<?>[] interfaces, InvocationHandler h)

　　再看实现InvocationHandler方法下的签名 public Object invoke(Object proxy, Method method, Object[] args)

[复制代码](javascript:void(0);)

//第一个参数为Proxy类类型实例，如匿名的$proxy实例

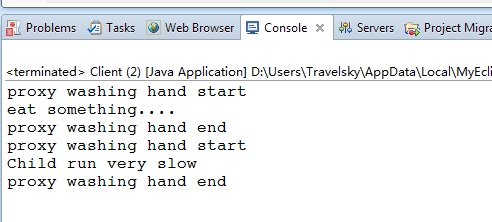
　 //第二个参数为委托类的方法对象//第三个参数为委托类的方法参数

　 //返回类型为委托类某个方法的执行结果

public Object invoke(Object proxy, Method method, Object[] args)  
　　  
 总的来说这个方法就是动态获取委托类里面的方法，在调用委托类的方法时在这个方法内进行拓展。 通过上面的Proxy.newProxyInstance方法，告诉底层代码，该去哪个类里面执行invoke方法。

[复制代码](javascript:void(0);)

　　由上面的描述，在客户端代码的调用后，结果如下：



     由此可见，在代理类对系统下每个方法的调用时，都会去调用invocationHandler里面的invoke方法. 包括里面的eat和run方法，可见受作用的范围是多广：每个方法都受代理类作用了。由例子可以看出，如果要实现我们想要的方法上面添加特定的代理，可以通过invoke方法里面的方法反射获取method对象方法名称即可实现（但这样会产生硬编码）。修改可如下：

[复制代码](javascript:void(0);)

1 @Override

2 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable

3 {

4 Object methodObject = null;

5 if (method.getName().equals("eat"))

6 {

7 System.out.println("proxy washing hand start");

8 methodObject = method.invoke(proxyTarget, args);

9 System.out.println("proxy washing hand end");

10 }

11 else

12 {  
 //不使用第一个proxy参数作为参数，否则会造成死循环

13 methodObject = method.invoke(proxyTarget, args);

14 }

15 return methodObject;

16 }

[复制代码](javascript:void(0);)

    由此，我们可以猜想出来动态代理在开发中的用法：

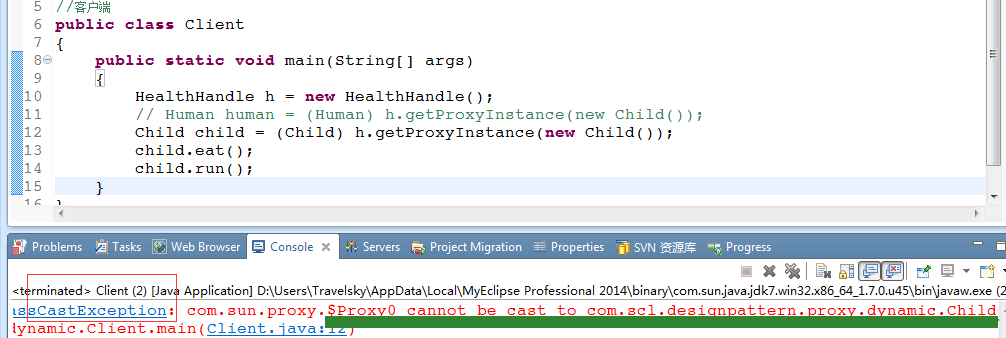
　　1. 监控程序下执行某个方法的耗时、性能及日志的编写

　　2. 监控程序方法的权限，不再单一地实行Controler→ServiceImpl→DaoImpl这种纵向编程。增加了在同一层面间横向编程的可能(切面编程:AOP)

　 另外有几点必须指出的内容：

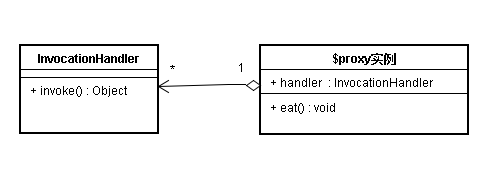
   1. 基于接口类的动态代理模式，必须具备抽象角色、委托类、代理三个基本角色。委托类和代理类必须由抽象角色衍生出来，否则无法使用该模式

   2.在使用Invoke方法的时候，需要对method的可访问性进行加工吗？不需要，动态代理模式最后返回的是具有抽象角色（顶层接口）的对象。在委托类内被private或者protected关键修饰的方法将不会予以调用，即使允许调用。也无法在客户端使用代理类转换成子类接口，对方法进行调用。



 3. 在invoke方法内为什么不使用第一个参数进行执行回调。

　　　　在客户端使用getProxyInstance（new Child( )）时，JDK会返回一个$proxy的实例，实例内有InvokecationHandler对象及动态继承下来的方法eat。客户端调用了eat方法，有如下操作：①JDK先查找$proxy实例内的handler对象 ②执行handler内的invoke方法。 根据public Object invoke(Object proxy, Method method, Object[] args)这个方法签名第一个参数proxy就是对应着下图的$proxy实例。如果在invoke内使用method.invoke(proxy,args) ,会出现这样一条方法链，eat→invoke→eat→invoke...，最终导致堆栈溢出。



**2.2 基于子类的动态代理**

　 假设目前有个项目，A公司写了一个Computer类已经整合到某一个jar包里面，这个类是个普通类没有任何抽象角色(没有实现任何接口)。但是B公司需要对Computer类内的方法进行加强、拓展。那B公司怎么办？目前只有两个方法：1、拿A公司代码反编译，然后修改A公司里面这个类的方法。 2、拿A公司代码反编译，让A公司Computer类进行抽象化，然后B公司使用基于接口的动态代理对方法进行拓展。很明显，这两个方法都是违背了面向对象设计的思想的 （①高内聚低耦合 ②开闭原则[OCP]:对拓展开放，对修改关闭）  
　 在这种情况下基于子类的动态代理产生了。基于子类的动态代理不是由JDK内附功能。需要引入第三方的开发包：[CGlib](http://download.csdn.net/detail/YSOLA4/3057412)，使用这个子类的代理需要引入cglib-nodep.jar  
 CGLib在代码上的使用基本和基于接口的动态代理一样，使用方法极为相似，仅表现在CGLib代理时使用的是方法拦截器MethodInterceptor。代码使用如下：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 package com.scl.designpattern.proxy.dynamic.cglib;

2

3 //婴儿类

4 public class Child

5 {

6 public void eat()

7 {

8 System.out.println("eat something....");

9 }

10

11 public void run()

12 {

13 System.out.println("Child run very slow");

14 }

15

16 protected void breath()

17 {

18 System.out.println("Child breath slowly");

19 }

20 }

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 package com.scl.designpattern.proxy.dynamic.cglib;

2

3 import java.lang.reflect.Method;

4

5 import net.sf.cglib.proxy.Enhancer;

6 import net.sf.cglib.proxy.MethodInterceptor;

7 import net.sf.cglib.proxy.MethodProxy;

8

9 public class HealthHandle implements MethodInterceptor

10 {

11

12 private Object proxyTarget;

13

14 public Object getProxyInstance(Object target)

15 {

16 this.proxyTarget = target;

17

18 return Enhancer.create(target.getClass(), target.getClass().getInterfaces(), this);

19 }

20

21 @Override

22 public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable

23 {

24 Object methodResult = null;

25 if (method.getName().equals("eat"))

26 {

27 System.out.println("wash hand before eat");

28 methodResult = method.invoke(proxyTarget, args);

29 methodProxy.invokeSuper(proxy, args);

30 System.out.println("wash hand after eat");

31 }

32 else

33 {

34 methodResult = method.invoke(proxyTarget, args);

35 }

36 return methodResult;

37 }

38 }

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 客户端

   源码Interceptor签名如下

[复制代码](javascript:void(0);)

1 /\*\*

2 \* All generated proxied methods call this method instead of the original method.

3 \* The original method may either be invoked by normal reflection using the Method object,

4 \* or by using the MethodProxy (faster). 官方建议使用MethodProxy对方法进行调用

5 \* @param obj "this", the enhanced object 跟接口代理一样，第一个参数为Enhanced对象实例

6 \* @param method intercepted Method

7 \* @param args argument array; primitive types are wrapped

8 \* @param proxy used to invoke super (non-intercepted method); may be called

9 \* as many times as needed

10 \* @throws Throwable any exception may be thrown; if so, super method will not be invoked

11 \* @return any value compatible with the signature of the proxied method. Method returning void will ignore this value.

12 \* @see MethodProxy

13 \*/

14 public Object intercept(Object obj, java.lang.reflect.Method method, Object[] args,MethodProxy proxy) throws Throwable;

[复制代码](javascript:void(0);)

另外要补充的两点：

1. 方法拦截器对protected修饰的方法可以进行调用

2. 官方推荐在对委托方法进行调用时使用MethodProxy对方法进行调用。这样有两个好处：①官方说速度比较快 ②在intercept内调用委托类方法时不用保存委托对象引用

# 转载于：https://www.cnblogs.com/doucheyard/p/5737366.html