代理模式

代理模式的基本概念 就是为其他对象提供一种代理，以控制对这个对象的访问

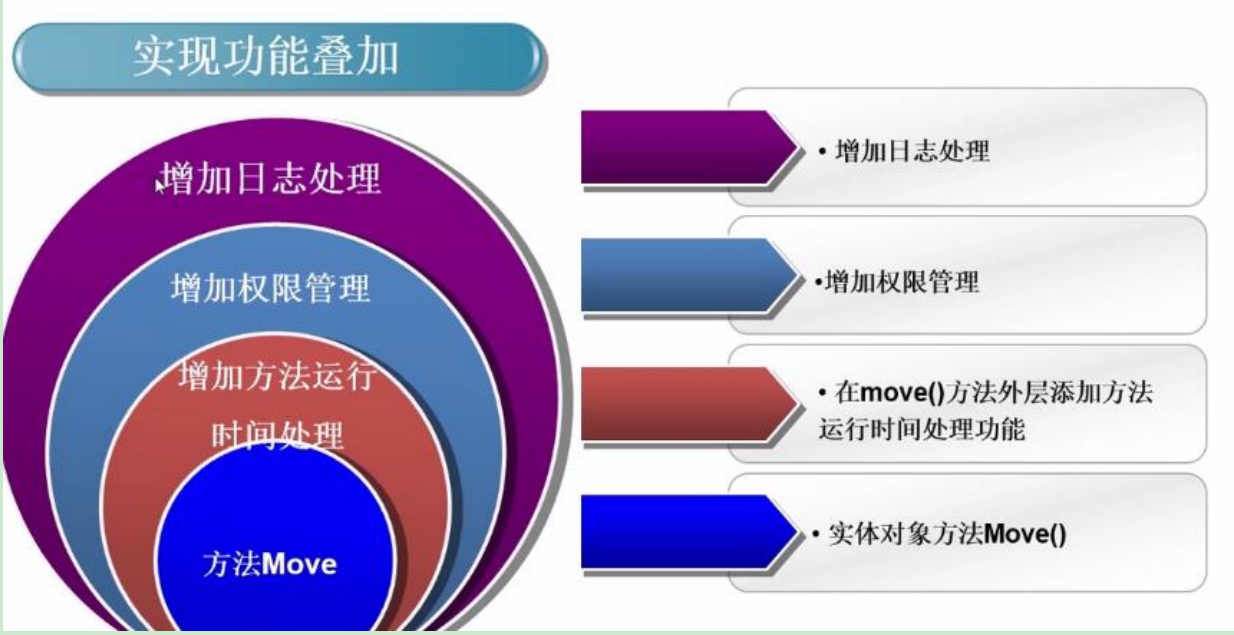
比如 火车站代买票的 比如火车站还可以提供一些额外的服务 代售处不能提供退票的 火车站这个退票的功能去掉了
去掉功能服务 增加额外服务

代理模式定义：
为其他对象提供一种代理以控制对这个对象的访问
代理对象起到中介作用，可去掉功能服务或增加额外的服务

下面是几种常见的代理模式
1 远程代理
为不同地理的对象提供局域网代表对象
2 虚拟代理
根据需要将资源消耗很大的对象进行延迟
真正需要的时候进行创建
3 保护代理
权限控制
4 智能引用代理
提供目标对象以外的一些服务

静态代理
实现方式1，继承的方式，子类重写方法实现父类代理
实现方式2，聚合的方式

关于实行代理模式的的有个网站 http://blog.csdn.net/zuoxiaolong8810/article/details/9026775



聚合比这个继承的方式更好
因为 继承的方式实现的时候，我们假如实现多一个功能，就需要再继承一次原来的类，再多写一次，如果我们变化的话，比如将第一个功能与第二个功能的顺序交换下，就会导致还需要再写一次，
这个是关于为什么要用继承而不用聚合的方式的原因http://blog.csdn.net/dream_broken/article/details/8841894
我大概理解了一下，展现了几个原因，这里用了几个例子 这里多了一个列子，就是组合，关于组合的概念 与聚合的概念，是
在这里体现的是 组合A B A死B死 聚合AB A
死 B不一定死

下面贴关于静态代理的代码

```
public class Eat implements EatAction {


@Override
public void EatFood(){
```

```
        System.out.println("开始吃啦");
         try {
             Thread.sleep(new Random().nextInt(1000));
} catch (InterruptedException e) {
             e.printStackTrace();
}
        System.out.println("吃完了");

}
```

```
public interface EatAction {
void EatFood();
}
```

```
public class EatLook  implements EatAction{
private  EatAction eatAction;

    public EatLook(EatAction eatAction) {
this.eatAction = eatAction;
}

@Override
public void EatFood() {
         System.out.println("听说你吃好东西了，我来看看");
eatAction.EatFood();
System.out.println("吃你妹啊，吃完了！");
}
```

```
public class EatTime implements  EatAction{

private  EatAction eatAction;

    public EatTime(EatAction eatAction) {
this.eatAction = eatAction;
}



@Override
public void EatFood() {
long start= System.currentTimeMillis();
eatAction.EatFood();
         long end= System.currentTimeMillis();
System.out.println("吃的时间...."+(end-start));
}
```

```
public static void main(String[] args) {
    Eat eat=new Eat();
EatLook eatLook=new EatLook(eat);
EatTime eatTime=new EatTime(eatLook);
eatTime.EatFood();
```

这个静态代理的重点是
这就类似Spring中的AOP以及 Filter wep中的拦截器，过滤器，
我们定义了一个规范的接口，下面的功能都会继承了这个接口，然后我们在第一个类实现一个功能，
在第二个类中 我们就开始定义了一个构造器，构造外面传来的参数，就是这个接口类型的，就是将第一个类传过来，
实现它的功能，在这个功能上，我们加上自己的功能，再加一个类，也是继承这个接口，都是一个类型的，然后我们还是按这个步骤，构造这个接口类型的构造器将第二个类传递过来加上我们自己的功能。

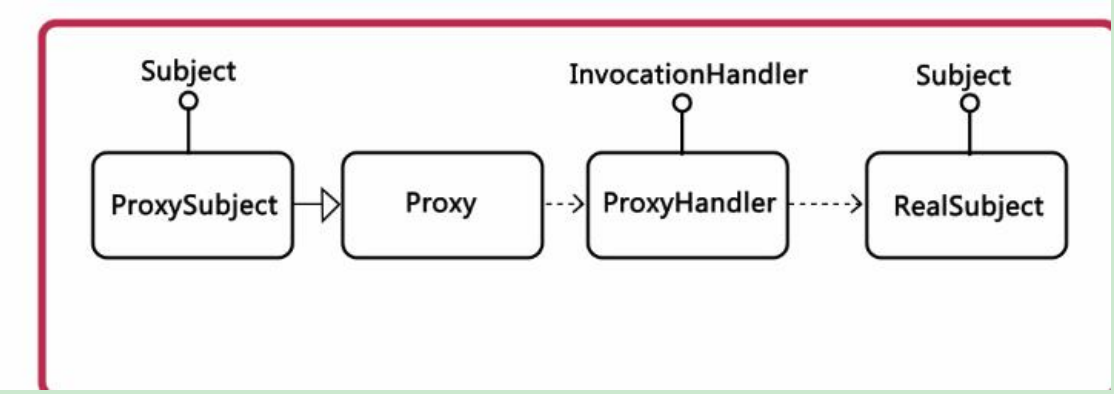
关于代理
就是通过构造器，传进原有的功能，在这个原有的功能上，实现我们自己的功能，而原来的功能则不需要new了，我们可以直接new这个新的功能的这个类 代理是什么？代理就是提供一种代理访问，而不是直接调用这个类本身。我大概就是理解到这了

代理模式的作用是：**为其他对象提供一种代理以控制对这个对象的访问**
在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

但是静态代理的缺点来了，如果我们想增加一个功能，就需要新写一个类，来实现新的功能，但是这样会导致业务层的大量增加代码。

这个时候我们还有一种方式 动态代理
动态产生代理，实现不同类，不同方法的代理

JDK的动态代理



InvocationHandler 这个接口只定义一个方法 就是invoke，有三个参数，第一个是代理类，第二个是被代理的方法 第三个是这个方法的参数

Proxy ：proxy表示最终生成的代理类对象 ，返回后的代理类可以被当做被代理类来使用（可使用被代理类的在接口中声明的方法）

Proxy的三个参数：

```
loader 类加载器
interface 实现接口
handler我们要实现的代理类 InvocationHandler
```

下面贴代码：

```
public static void main(String[] args) {
   Car car=new Car();
InvocationHandler handler=new TimeHandler(car);
Class<?> cls=car.getClass();
/*loader 类加载器
    interface 实现接口
    *handler我们要实现的代理类 InvocationHandler
    *
    * */
Moveable moveable= (Moveable) Proxy.newProxyInstance(cls.getClassLoader(),cls.getInt
erfaces(),handler);
moveable.move();
}
```

```
public class TimeHandler implements InvocationHandler {
private  Object target;

    public TimeHandler(Object target) {
        super();
        this.target = target;
}
/*
     参数：proxy被代理的对象
     参数：method被代理的对象的方法
     参数：args被代理对象的方法的参数
     返回值是Object对象
     就是Object方法的返回值
    * */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        long starttime = System.currentTimeMillis();
        System.out.println("汽车开始行使、、" + starttime);
        method.invoke(target);
        long endtime = System.currentTimeMillis();
         System.out.println("汽车结束行使、、" + endtime + "汽车行使的时间" + (endtime - s
tarttime));

        return null;
}
```

关键在于Handler

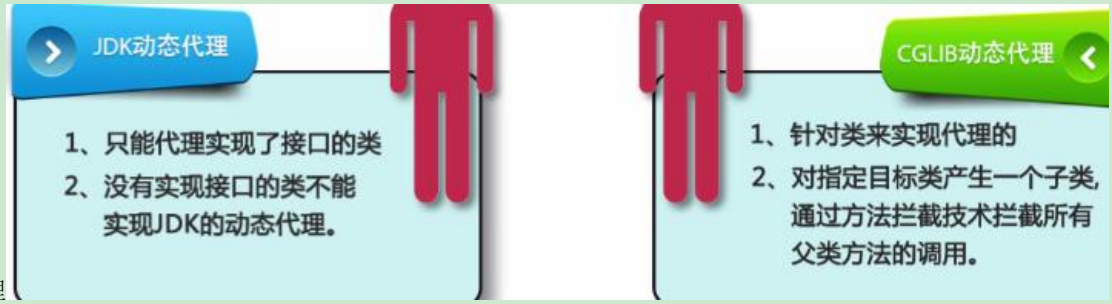
1 在于一个实现接口InvocationHandler的类，它必须实现invoke方法

2 创建被代理的类以及接口

3 调用Proxy的静态方法，创建一个代理类

```
newProxyInstance(ClassLoader loader,Class[]
interfaces, InvocationHandler  h)
```

4 通过代理调用方法

CGLIB动态代理 

这种采用的是继承的方式

```
public class CglibProxy  implements MethodInterceptor {

private Enhancer enhancer = new Enhancer();

    public Object getProxy(Class clazz){
      //设置创建子类的类
      enhancer.setSuperclass(clazz);
      enhancer.setCallback(this);

      return enhancer.create();
}

/**
     * 拦截所有目标类方法的调用
     * obj   目标类的实例
     * m    目标方法的反射对象
     * args   方法的参数
     * proxy代理类的实例
     */
@Override
public Object intercept(Object obj, Method m, Object[] args,
        MethodProxy proxy) throws Throwable {
        System.out.println("日志开始...");
        //代理类调用父类的方法
        proxy.invokeSuper(obj, args);
        System.out.println("日志结束...");
        return null;
}
```

```
public class Train {

public  void move() {
       System.out.println("火车行驶中。。。");
}
```

```
public static void main(String[] args) {
  CglibProxy proxy=new CglibProxy();
  Train t = (Train)proxy.getProxy(Train.class);
  t.move();
```

实现方法，第一个开始实现一个继承与MethodInterceptor的接口，

实现其中的一个intercept方法，这个方法主要就是负责处理我们需要处理的类，我们可以在这里加上我们自己的业务逻辑

同时，我们还需要实现一个Enhancer的类，这个类是实现生成代理的类，然后返回出去，接受到，然后直接实现方法，就完成了

下面是一个关于动态代理的 例子：

实现功能：

通过Proxy的newProxyInstance返回代理对象

1 需要声明一段源码（动态产生代理）

2 编译源码（JDK ComPiler API），产生新的类（代理类）

3 将这个类load到内存中当中，产生一个新的对象（代理对象）

4 return 代理对象

下面这就是源码：

```
//这个方法的最终目的是产生一个代理类  这个就是我们要生成
public static Object newProxyInstance(Class infce) throws Exception {
        String rt = "\r\n";
        String methodStr = "";
        for (Method m : infce.getMethods()) {
            methodStr += "    @Override\n" + rt +
"    public void " + m.getName() + "() {\n" + rt +
"\n" + rt +
"        long starttime = System.currentTimeMillis();\n" + rt +
"        System.out.println(\"汽车开始行使、、\" + starttime);\n" + rt +
"        m." + m.getName() + "();\n" + rt +
"        long endtime = System.currentTimeMillis();\n" + rt +
"        System.out.println(\"汽车结束行使、、\" + endtime + \"汽车行使的时间\" + (endtime
 - starttime));\n" + rt +
"\n" + rt +
"    }";
}
        String str = "package com.imocc.proxy;\n" +
"\n" + rt +
"/**\n" + rt +
" * Created by Administrator on 2016/5/27.\n" +
" */\n" + rt +
"public class $Proxy0 implements " + infce.getName() + " {" + rt +
"\n" + rt +
"    private  " + infce.getName() + " m;\n" +
"    public  $Proxy0() {\n" +
"\n" + rt +
"    }\n" + rt +
"\n" + rt +
"    public $Proxy0(" + infce.getName() + " m) {\n" +
"        super();\n" + rt +
"      this.m=m;\n" + rt +
"    }\n" + rt +
"\n" + rt +
                methodStr +
"}\n";
//        产生代理类的java文件
String filanname = System.getProperty("user.dir") + "/bin/com/imooc/proxy/$Proxy0.ja
va";
//        这是个当前我们要生成这个类的路径
File file = new File(filanname);
//         生成我们需要的类
```

```
FileUtils.writeStringToFile(file, str);

//编译
        //拿到编译器
JavaCompiler complier = ToolProvider.getSystemJavaCompiler();
//文件管理者
StandardJavaFileManager fileMgr =
                complier.getStandardFileManager(null, null, null);
//获取文件
Iterable units = fileMgr.getJavaFileObjects(filanname);
//编译任务
CompilationTask t = complier.getTask(null, fileMgr, null, null, null, units);
//进行编译
t.call();
fileMgr.close();

//        load到内存
ClassLoader cl=ClassLoader.getSystemClassLoader();
Class c = cl.loadClass("com.imooc.proxy.$Proxy0");
Constructor ctr = c.getConstructor(infce);
        return ctr.newInstance(new Car());
```

动态代理类的源码是在程序运行期间由JVM根据反射等机制动态的生成，所以不存在代理类的字节码文件。代理类和委托类的关系是在程序运行时确定

这就是我们在内部动态改的代码 这个没用实现代理

```
packag..
import..
import..
public class $ProxyN implements infce.getclass(){
    private handler h;

 public $ProxyN(){
    this.h=h;
}
for(循环接口所有的方法){
  move(){
       try{

      Method md=....;//具体方法名    在这里 我们第一个我们自己写的模板，这边我们直接定死了，第二个才是接口，第三个才是这个h   也就是代理 handler
        h.invoke （this,md） //实际执行的是h的方法 然后由h代理 通过反射加载
}catch(Exception e){
  e.printStackTrace();
}

}

}
```

就是这样 我们实际执行的时候样子，我只是随便写下，表示下意思 这个接口不是定死的，是我们传参导致出现的
我们拿到我们需要代理的类，然后我们需要内部改动，然后再生成 关键就在于动态生成，动态，传过来的时候，我们代理的比如是car，其中我们获取到了car的模板，然后我们想给它加上方法，关于这个方法，就是我们的精髓了，我们是定义一个接口，这个接口就是我们定义的方法的地方，就是用上Java反射机制动态生成，这个时候，我们需要一个代理对象，也就是handler，我们可以在这里加上我们自己的业务逻辑，然后我们最终返回的是这个handler,

```
@SuppressWarnings("unchecked")
public static Object newProxyInstance(Class infce,InvocationHandler h) throws Except
ion{
    String rt = "\r\n";
String methodStr = "";
    for(Method m : infce.getMethods()){
        methodStr += "    @Override" + rt +
```

```
"    public void " + 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 +
"    }" ;
}//加上我们获取到的方法 也就是接口定义的所有方法

    String str =
"package com.imooc.proxy;" + rt +
"import java.lang.reflect.Method;" + rt +
"import com.imooc.proxy.InvocationHandler;" +  rt+
"public class $Proxy0 implements " + infce.getName() + " {" + rt +
"    public $Proxy0(InvocationHandler h) {" + rt +
"        this.h = h;" + rt +
"    }" + rt +
"  private InvocationHandler h;" + rt+
                    methodStr + rt +
"}" ;
//产生代理类的java文件
String filename = System.getProperty("user.dir") +"/bin/com/imooc/proxy/$Proxy0.java
";
File file = new File(filename);
FileUtils.writeStringToFile(file, str);

//编译
    //拿到编译器
JavaCompiler complier = ToolProvider.getSystemJavaCompiler();
//文件管理者
StandardJavaFileManager fileMgr =
            complier.getStandardFileManager(null, null, null);
//获取文件
Iterable units = fileMgr.getJavaFileObjects(filename);
//编译任务
JavaCompiler.CompilationTask t = complier.getTask(null, fileMgr, null, null, null, u
nits);
//进行编译
t.call();
fileMgr.close();

//load 到内存
ClassLoader cl = ClassLoader.getSystemClassLoader();
Class c = cl.loadClass("com.imooc.proxy.$Proxy0");

Constructor ctr = c.getConstructor(InvocationHandler.class);
    return ctr.newInstance(h);
```

```
public class TimeHandler implements  InvocationHandler {

private  Object target;


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

@Override
public void invoke(Object o, Method m) {
try {
long starttime = System.currentTimeMillis();
System.out.println("汽车开始行驶....");
m.invoke(target);//这是通过反射来加载的类方法
long endtime = System.currentTimeMillis();
System.out.println("汽车结束行驶....  汽车行驶时间: "
+ (endtime - starttime) + "毫秒! ");
} catch (Exception e) {
            e.printStackTrace();
}

    }
}
```

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

}
```

这是代码
这里面有一个for循环获取到所有接口中声明的方法 我们可以获取到这里接口所有的方法，然后

动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）

但是这种，方式，这个业务逻辑是写死的，所以我们可以写一个事务处理器，用来专门对方法来处理。

代理模式就是去掉了某些功能，或者添加某些功能，进行对原来的对象上增加一些额外的服务，如果我们调用源码的

某些方法，但是我们不能修改源码，我们就可以用动态代理来代理方法

与静态代理类对照的是动态代理类，动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。java.lang.reflect 包中的Proxy类和InvocationHandler 接口提供了生成动态代理类的能力。

这种的好处在就是在于代理类，静态代理的代理是定死的，动态代理的代理是活的，我们真正在乎的不是接口与这个我们要被代理的类之间的优化，而是
代理类的方法的复用