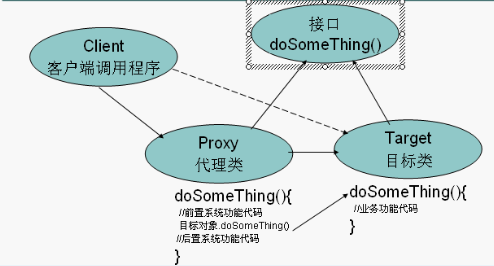
# 代理的概念与作用

* 生活中的代理
* 武汉人从武汉的代理商手中买联想电脑和直接到联想总部买电脑，你觉得最终的业务主体目标有什么区别吗？基本上一样吧，都解决了核心问题，但是，一点区别都没有吗？从代理商那里买真的一点好处都没有吗？
* 程序中的代理
* 要为已存在的多个具有相同接口的目标类的各个方法增加一些系统功能，例如,异常处理、日志、计算方法的运行时间、事务管理、等等，你准备如何做？
* 编写一个与目标类具有相同接口的代理类，代理类的每个方法调用目标类的相同方法，并在调用方法时加上系统功能的代码(参见下面的原理图)。
* 如果采用工厂模式和配置文件的方式进行管理，则不需要修改客户端程序，在配置文件中配置是使用目标类、还是代理类，这样以后很容易切换，譬如，想要日志功能时就配置代理类，否则配置目标类，这样增加系统功能很容易，以后运行一段时间后，又想去掉系统功能也很容易。

# 代理架构图



# 静态代理(Static Proxy)

示例伪码如下：

class X{

void sayHello(){

syso: hello, itcast;

}

}

XProxy{

starttime

X.sayHello();

endtime

}

举个实际的例子来说：

IHello

**public** **interface** IHello {

**public** **void** hello(String name);

}

HelloSpeaker

**public** **class** HelloSpeaker **implements** IHello {

@Override

**public** **void** hello(String name) {

System.*out*.println("hello, "+name);

}

}

HelloProxy

**public** **class** HelloProxy **implements** IHello {

**private** Logger logger = Logger.*getLogger*(**this**.getClass().getName());

**private** IHello helloObject;

**public** HelloProxy(IHello helloObject){

**this**.helloObject = helloObject;

}

@Override

**public** **void** hello(String name) {

//日志服务

log("hello method starts ......");

//执行业务逻辑

helloObject.hello(name);

//日志服务

log("hello method ends....");

}

**private** **void** log(String msg){

logger.log(Level.*INFO*,msg);

}

}

ProxyDemo

**public** **class** ProxyDemo {

**public** **static** **void** main(String[] args){

IHello proxy = **new** HelloProxy(**new** HelloSpeaker());

proxy.hello("Mr Zhang");

}

}

程序中调用执行的是代理对象，构造代理对象时必须给它一个被代理对象，记得在操作取回代理对象市，必须转换操作接口为Ihello接口，看一下实际执行结果：

2014-4-1 0:59:59 cn.ifinance.HelloProxy log

信息: hello method starts ......

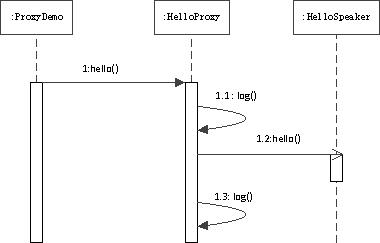
2014-4-1 0:59:59 cn.ifinance.HelloProxy log

信息: hello method ends....

hello, Mr Zhang

代理对象HelloProxy将替代真正的HelloSpeaker来执行hello()，并在其前后加上日志的动作，这使得HelloSpeaker在编写时不必介入日志动作，HelloSpeaker可以专心于它的职责，可以从图解的方式来更进一步看出代理机制的运作流程，

如图（设计模型时序图）所示：



这是静态代理的基本范例，代理对象的一个接口只服务于一种类型的对象，如果要代理的方法很多，势必要为每种方法进行代理，静态代理在程序规模稍大时就无法胜任了，在这里介绍静态代理的目的，是让你了解代理的基本原理。

很多人会问：为什么上面例子中的日志消息都出现在“hello, Mr Zhang”之前，要直到，日志的动作是由另一个线程来进行的，日志动作并不影响（或介入）程序流程，范例的重点在于日志动作确实有执行，而不是在于消息在主控台出现的顺序，日志动作也不是用来预测程序执行顺序之用。

# 动态代理(Dynamic proxy)

* 要为系统中的各种接口的类增加代理功能，那就需要太多的代理类，全部采用静态代理(Statoc proxy)方式，将是一件非常麻烦的事情！写成百上千个代理类，是不是太累！

在JDK1.3之后加入了可协助开发动态代理功能的API,你不必为特定的对象与方法编写特定的代理对象。使用动态代理，可以使得一个处理者（Handler）服务于各个对象。首先一个处理者的类设计必须实现java.lang.reflect.InvocationHandler接口。

* JVM可以在运行期动态生成出类的字节码，这种动态生成的类往往被用作代理类，即动态代理类。
* JVM生成的动态类必须实现一个或多个接口，所以，JVM生成的动态类只能用作具有相同接口的目标类的代理。
* CGLIB库可以动态生成一个类的子类，一个类的子类也可以用作该类的代理，所以，如果要为一个没有实现接口的类生成动态代理类，那么可以使用CGLIB库
* 代理类的各个方法中通常除了要调用目标的相应方法和对外返回目标的返回结果外，还可以在代理方法中的如下四个位置加上系统功能代码：
* 1.在调用目标方法之前
* 2.在调用目标方法之后
* 3.在调用目标方法前后
* 4.在处理目标方法异常的catch块中

**public** **interface** IHello {

**public** **void** hello(String name);

}

**public** **class** HelloSpeaker **implements** IHello {

@Override

**public** **void** hello(String name) {

System.*out*.println("hello, "+name);

}

}

**public** **class** LogHandler **implements** InvocationHandler {

**private** Logger logger = Logger.*getLogger*(**this**.getClass().getName());

**private** Object delegate;

**public** Object bind(Object delegate) {

**this**.delegate = delegate;

**return** Proxy.*newProxyInstance*(delegate.getClass().getClassLoader(),

delegate.getClass().getInterfaces(), **this**);

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

Object result = **null**;

log("method starts ... " + method);

result = method.invoke(delegate, args);

logger.log(Level.*INFO*, "method ends ..." + method);

**return** result;

}

**private** **void** log(String message) {

logger.log(Level.*INFO*, message);

}

}

**public** **class** DynamicProxyDemo {

**public** **static** **void** main(String[] args) {

LogHandler logHandler = **new** LogHandler();

IHello helloProxy = (IHello)logHandler.bind(**new** HelloSpeaker());

helloProxy.hello("Mr Zhang");

}

}

执行结果如下：

2014-4-1 2:52:48 cn.ifinance.LogHandler log

信息: method starts ... public abstract void cn.ifinance.IHello.hello(java.lang.String)

2014-4-1 2:52:48 cn.ifinance.LogHandler invoke

信息: method ends ...public abstract void cn.ifinance.IHello.hello(java.lang.String)

hello, Mr Zhang

LogHandler不再服务于特定对象与接口，而HelloSpeaker也不用插入任何有关日志的动作，它不用意识到日志动作的存在。

使用代理对象将日志等与业务逻辑无关的动作或任务提取出来，设计成一个服务对象，像之前范例中的HelloProxy或是LogHandler,这样的对象称之为切面（Aspect）.

# 分析JVM动态生成的类

* 创建实现了Collection接口的动态类和查看其名称，分析Proxy.getProxyClass方法的各个参数。
* 编码列出动态类中的所有构造方法和参数签名
* 编码列出动态类中的所有方法和参数签名
* 创建动态类的实例对象
* 用反射获得构造方法
* 编写一个最简单的InvocationHandler类
* 调用构造方法创建动态类的实例对象，并将编写的InvocationHandler类的实例对象传进去
* 打印创建的对象和调用对象的没有返回值的方法和getClass方法，演示调用其它有返回值的方法报告了异常。
* 将创建动态类的实例对象的代理改成匿名内部类的形式编写，锻炼大家习惯匿名内部类。
* 总结思考： 让jvm创建动态类，需要给它提供哪些信息？

三个方面：

* 生成的类中有哪些方法，通过让其实现哪些接口的方式进行告知;
* 产生的类字节码必须有一个关联的类加载器对象；
* 生成的类中的方法的代码是怎样的，也得由我们提供。把我们的代码写在一个约定好了接口对象的方法中，把对象传给它，它调用我的方法，即相当于插入了我的代码。提供执行代码的对象就是那个InvocationHandler对象，它是在创建动态类的实例对象的构造方法时传递进去的。在上面的Invoccation对象的invoke方法中加一点代码，就可以看到这些代码被调用了。
* 用Proxy.newInstance方法直接一步就创建出代理对象。

Advice.java

**import** java.lang.reflect.Method;

**public** **interface** Advice {

**void** beforeMethod(Method method); // 默认为public

**void** afterMethod(Method method);

}

MyAdvice.java

**import** java.lang.reflect.Method;

**public** **class** MyAdvice **implements** Advice {

**long** beginTime = 0;

@Override

**public** **void** beforeMethod(Method method) {

System.*out*.println("到itcast学习了！");

beginTime = System.*currentTimeMillis*();

}

@Override

**public** **void** afterMethod(Method method) {

System.*out*.println("从itcast毕业上班了！");

**long** endTime = System.*currentTimeMillis*();

System.*out*.println(method.getName()+" cost time :" + (endTime-beginTime));

}

}

ProxyTest.java

**public** **class** ProxyTest {

/\*\*

\* jvm动态生成类

\*/

**public** **static** **void** main(String[] args) **throws** Exception {

Class clazzProxy1 = Proxy.*getProxyClass*(

Collection.**class**.getClassLoader(), Collection.**class**);

System.*out*.println(clazzProxy1.getName());

System.*out*.println("-----begin constructors list------");

/\*\*

\* $Proxy0() $Proxy0(InvocationHandler,int)

\* \*/

// StringBuilder在单线程下效率高点，StringBuffer考虑到了线程安全，在多线程情况下用

Constructor[] constructors = clazzProxy1.getConstructors();

**for** (Constructor constructor : constructors) {

String name = constructor.getName();

StringBuilder sBuilder = **new** StringBuilder(name);

sBuilder.append('(');

Class[] clazzParams = constructor.getParameterTypes();

**for** (Class clazzParam : clazzParams) {

sBuilder.append(clazzParam.getName()).append(',');

}

**if** (clazzParams != **null** && clazzParams.length != 0)

sBuilder.deleteCharAt(sBuilder.length() - 1);

sBuilder.append(')');

System.*out*.println(sBuilder);

}

System.*out*.println("-----begin methods list------");

Method[] methods = clazzProxy1.getMethods();

**for** (Method method : methods) {

String name = method.getName();

StringBuilder sBuilder = **new** StringBuilder(name);

sBuilder.append('(');

Class[] clazzParams = method.getParameterTypes();

**for** (Class clazzParam : clazzParams) {

sBuilder.append(clazzParam.getName()).append(',');

}

**if** (clazzParams != **null** && clazzParams.length != 0)

sBuilder.deleteCharAt(sBuilder.length() - 1);

sBuilder.append(')');

System.*out*.println(sBuilder);

}

System.*out*.println("-----begin create instance object list------");

// Object obj = clazzProxy1.newInstance();

// //会调用不带参数的构造方法，但clazzProxy1并没有无参的构造方法

Constructor constructor = clazzProxy1

.getConstructor(InvocationHandler.**class**);

// 内部类

**class** MyInvocationHandler1 **implements** InvocationHandler {

@Override

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

// **TODO** Auto-generated method stub

**return** **null**;

}

}

Collection proxy1 = (Collection) constructor

.newInstance(**new** MyInvocationHandler1());

System.*out*.println(proxy1); // 相当于

// System.out.println(proxy1.toString());

System.*out*.println(proxy1 == **null**);

proxy1.clear(); // 无返回值，不报错；会调用invoke方法,clear()无返回值(void)，而上面invoke方法的返回值为null,代表的就是无返回值。

// proxy1.size(); //有返回值，会报错；会调用invoke方法，而上面invoke方法的返回值为null，所以报错！

// 内部类的另一种写法

Collection proxy2 = (Collection) constructor

.newInstance(**new** InvocationHandler() {

@Override

**public** Object invoke(Object proxy, Method method,

Object[] args) **throws** Throwable {

// **TODO** Auto-generated method stub

**return** **null**;

}

});

/\*//jvm生成动态类后直接创建其实例对象

Collection proxy3 = (Collection) Proxy.newProxyInstance(

Collection.class.getClassLoader(),

new Class[] { Collection.class },

new InvocationHandler() {

ArrayList target = new ArrayList(); //写在这儿,proxy3.size()输出3

@Override

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

//ArrayList target = new ArrayList(); //写在这儿,proxy3.size()输出0

long beginTime = System.currentTimeMillis();

Object retVal = method.invoke(target, args);

long endTime = System.currentTimeMillis();

System.out.println(method.getName()+" cost time :" + (endTime-beginTime));

return retVal;

// return method.invoke(proxy, args);会造成死循环

}

});

Object obj = proxy3.add("zxx"); // obj也即是public Object invoke(...){...}方法的返回值

proxy3.add("lhm");

proxy3.add("bxd");

System.out.println(proxy3.size());

// 在代理实例上的 java.lang.Object 中声明的 hashCode、equals 或 toString 方法的调用

// 将按照与编码和指派接口方法调用相同的方式进行编码，并被指派到调用处理程序的 invoke 方法，如上所述。

// 传递到 invoke 的 Method 对象的声明类是 java.lang.Object。

// 代理类不重写从 java.lang.Object 继承的代理实例的其他公共方法，

// 所以这些方法的调用行为与其对 java.lang.Object 实例的操作一样。

System.out.println(proxy3.getClass().getName());\*/

//对上面注释的代码进行重构

**final** ArrayList target = **new** ArrayList(); //方法内部，提供给内部类使用的变量必须是常量

Collection proxy3 = (Collection)*getProxy*(target,**new** MyAdvice());

Object obj = proxy3.add("zxx"); // obj也即是public Object invoke(...){...}方法的返回值

proxy3.add("lhm");

proxy3.add("bxd");

System.*out*.println(proxy3.size());

System.*out*.println(proxy3.getClass().getName());

}

**private** **static** Object getProxy(**final** Object target,**final** Advice advice) {

Object proxy3 = Proxy.*newProxyInstance*(

target.getClass().getClassLoader(),

/\*new Class[] { Collection.class },\*/

target.getClass().getInterfaces(),

**new** InvocationHandler() {

@Override

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

advice.beforeMethod(method);

Object retVal = method.invoke(target, args);

advice.afterMethod(method);

**return** retVal;

}

});

**return** proxy3;

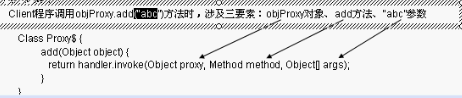
}

}

# 猜想分析JVM动态生成的类的内部代码

* 动态生成的类实现了Collection接口(可以实现若干接口)，生成的类有Collection接口中的所有方法和一个如下接受InvocationHandler参数的构造方法。
* 构造方法接受一个InvocationHandler对象，接受对象要干什么用呢？该方法内部的代码会是怎样的呢？
* 实现Collection接口的动态类中的各个方法的代码又是怎样的呢？

InvocationHandler接口中定义的invoke方法接受的三个参数又是什么意思？图解说明如下：



* 分析先前打印动态类的实例对象时，结果为什么会是null呢？调用有基本类型返回值的方法时为什么会出现NullPointerException异常？
* 分析为什么动态类的实例对象的getClass()方法返回了正确结果呢？
* 调用代理对象的从Object类继承的hashCode、equals 或 toString 这几个方法时，代理对象将请求转发给InvocationHandler对象，对于其他方法，则不转发调用请求。

动态代理类部分内部代码:

InvocationHandler handler;

public $Proxy0(InvocationHandler handler){

this.handler = handler ;

}

int size(){

return handler.invoke(this,this.getClass().getMethod(“size”),null);

}

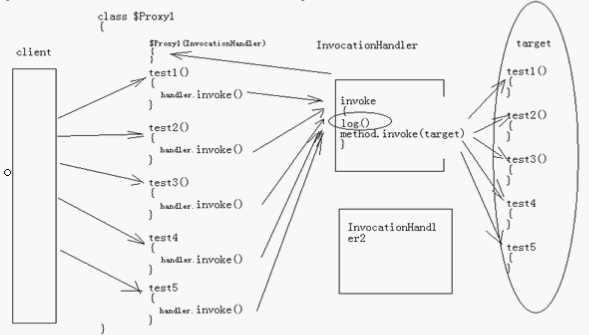
void clear(){

return handler.invoke(this,this.getClass().getMethod(“clear”),null);

}

# 让动态生成的类成为目标类的代理

* 分析动态代理的工作原理图



* 怎样将目标类传进去？
* 直接在InvocationHandler实现类中创建目标类的实例对象，可以看运行效果和加入日志代码，但没有实际意义。
* 为InvocationHandler实现类注入目标类的实例对象，不能采用匿名内部类的形式了。
* 让匿名的InvocationHandler实现类访问外面方法中的目标类实例对象的final类型的引用变量。
* 将创建代理的过程改为一种更优雅的方式，eclipse重构出一个getProxy方法绑定接收目标同时返回代理对象，让调用者更懒惰、更方便，调用甚至不用接触任何代理的API.
* 将系统功能代码模块化，即将切面代码也改为通过参数形式提供，怎样把要执行的系统功能代码以参数形式提供？
* 把要执行的代码装到一个对象的某个方法里，然后把这个对象作为参数传递，接收者只要调用这个对象的方法，即等于执行了外界提供的代码！
* 为bind方法增加一个Advice参数。

# AOP

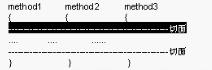
在一个服务的流程中插入与业务逻辑无关的系统服务逻辑（例如 Logging、Security），这样的逻辑称为Cross-cutting concerns , 将Crossing-cutting concerns 独立出来设计为一个对象，这样的特殊对象称之为Aspect ， Aspect-oriented programming 着重在Aspect的设计上以及与应用程序的织入(Weave).

AOP跟OOP(Object-oriented programming)并不互相抵触，它们是可以相辅相成的两个设计模型，Spring AOP是实现AOP的一种技术，而Spring AOP也是Spring中一些子框架或子功能所依赖的核心。

* 系统中存在交叉业务，一个交叉业务就是要切入到系统中的一个方面，如下所示：



* 用具体的程序代码描述交叉业务：



* 交叉业务的编程问题即为面向方面编程(AOP)，AOP的目标就是要使交叉业务模块化。可以采用将切面代码移动到原始方法的周围，这与直接在方法中编写切面代码的运行效果是一样的，如下所示：



* 使用代理技术正好可以解决这种问题，代理是实现AOP功能的核心和关键技术。

注：安全、事务、日志等功能要贯穿到好多个模块中，所以他们就是交叉业务。

重要原则：不要把供货商暴露给你的客户。

# 实现AOP功能的封装与配置

* 工厂类BeanFactory负责创建目标类或代理类的实例对象，并通过配置文件实现切换。其中getBean方法根据参数字符串返回一个相应的实例对象，如果参数字符串在配置文件中对应的类名不是ProxyFactoryBean，则直接返回该类的实例对象，否则，返回该类实例对象的getProxy方法返回的对象。
* BeanFactory的构造方法接收代表配置文件的输入流对象，配置文件格式如下：

#xxx=java.util.ArrayList

xxx=cn.itcast.day3.aopframework.ProxyFactoryBean

xxx.advice=cn.itcast.day3.MyAdvice

xxx.target=java.util.ArrayList

* ProxyFactoryBean充当封装生成动态代理的工厂，需要为工厂类提供哪些配置参数信息？
* 目标
* 通知
* 编写客户端应用：
* 编写实现Advice接口的类和在配置文件中进行配置
* 调用BeanFactory获取对象

代码示例：

Advice.java

**import** java.lang.reflect.Method;

**public** **interface** Advice {

**void** beforeMethod(Method method); // 默认为public

**void** afterMethod(Method method);

}

MyAdvice.java

**import** java.lang.reflect.Method;

**public** **class** MyAdvice **implements** Advice {

**long** beginTime = 0;

@Override

**public** **void** beforeMethod(Method method) {

System.*out*.println("到itcast学习了！");

beginTime = System.*currentTimeMillis*();

}

@Override

**public** **void** afterMethod(Method method) {

System.*out*.println("从itcast毕业上班了！");

**long** endTime = System.*currentTimeMillis*();

System.*out*.println(method.getName()+" cost time :" + (endTime-beginTime));

}

}

config.properties

#xxx=java.util.ArrayList

xxx=cn.itcast.day3.aopframework.ProxyFactoryBean

xxx.advice=cn.itcast.day3.MyAdvice

xxx.target=java.util.ArrayList

ProxyFactoryBean.java

**import** java.lang.reflect.InvocationHandler;

**import** java.lang.reflect.Method;

**import** java.lang.reflect.Proxy;

**import** cn.itcast.day3.Advice;

**public** **class** ProxyFactoryBean {

**private** Advice advice;

**private** Object target;

**public** Advice getAdvice() {

**return** advice;

}

**public** **void** setAdvice(Advice advice) {

**this**.advice = advice;

}

**public** Object getTarget() {

**return** target;

}

**public** **void** setTarget(Object target) {

**this**.target = target;

}

**public** Object getProxy() {

Object proxy3 = Proxy.*newProxyInstance*(

target.getClass().getClassLoader(),

/\*new Class[] { Collection.class },\*/

target.getClass().getInterfaces(),

**new** InvocationHandler() {

@Override

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

advice.beforeMethod(method);

Object retVal = method.invoke(target, args);

advice.afterMethod(method);

**return** retVal;

}

});

**return** proxy3;

}

}

BeanFactory.java

**public** **class** BeanFactory {

Properties props = **new** Properties();

**public** BeanFactory(InputStream ips){

**try** {

props.load(ips);

} **catch** (IOException e) {

e.printStackTrace();

}

}

**public** Object getBean(String name){

String className = props.getProperty(name);

Object bean = **null**;

**try** {

Class clazz = Class.*forName*(className);

bean = clazz.newInstance();

} **catch** (Exception e) {

e.printStackTrace();

}

**if**(bean **instanceof** ProxyFactoryBean){

Object proxy = **null**;

ProxyFactoryBean proxyFactoryBean = (ProxyFactoryBean)bean;

**try** {

Advice advice = (Advice)Class.*forName*(props.getProperty(name+".advice")).newInstance();

Object target = Class.*forName*(props.getProperty(name+".target")).newInstance();

proxyFactoryBean.setAdvice(advice);

proxyFactoryBean.setTarget(target);

proxy = proxyFactoryBean.getProxy();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** proxy;

}

**return** bean;

}

}

AopFrameworkTest.java

**public** **class** AopFrameworkTest {

**public** **static** **void** main(String[] args) {

InputStream ips = AopFrameworkTest.**class**.getResourceAsStream("config.properties");

Object bean = **new** BeanFactory(ips).getBean("xxx");

System.*out*.println(bean.getClass().getName());

//((Collection)bean).clear();

}

}