博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

代理设计模式是在程序开发之中使用最多的设计模式，代理设计模式的核心是有真实业务实现类与代理业务实现类，并且代理类要完成比真实业务更多的处理操作。

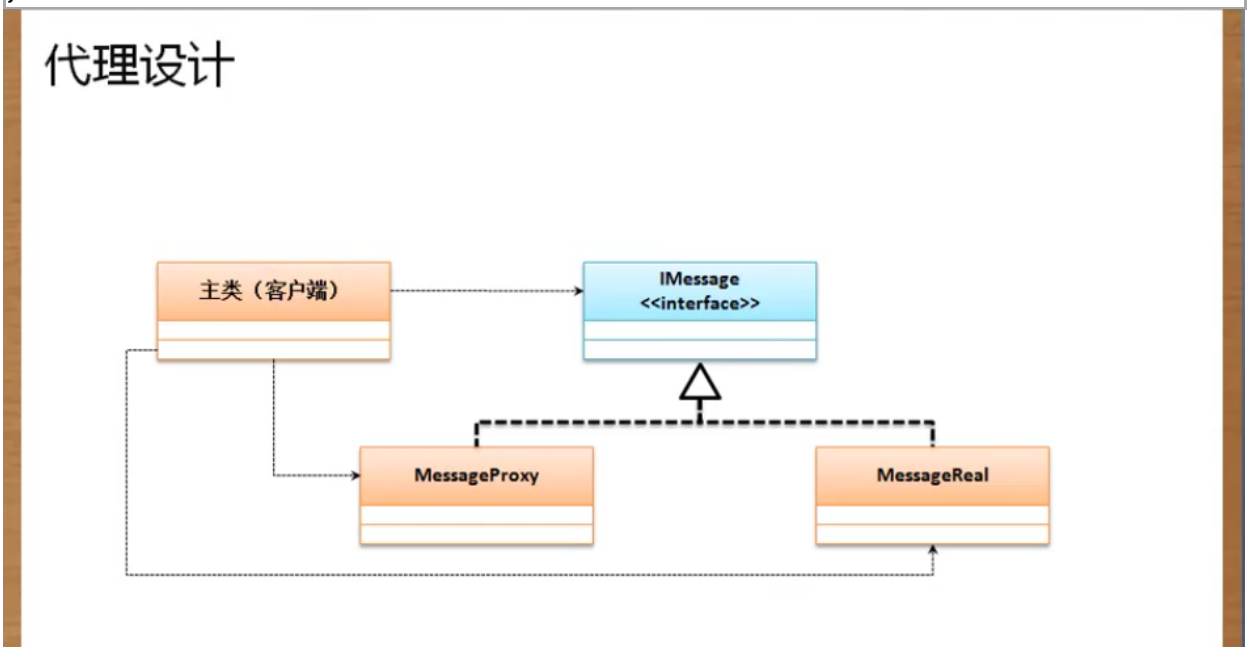
■传统代理设计模式的弊端

所有的代理设计模式如果按照设计要求来讲，必须是基于接口的设计，也就是说需要首先定义出核心接口的组成，下面模拟一个消息发送的代理操作结构。

范例：传统代理设计

```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		IMessage msg = new MessageProxy(new MessageReal()) ;
		msg.send();
	}
}
interface IMessage {		// 传统代理设计必须有接口
	public void send() ; // 业务方法
}
class MessageReal implements IMessage {
	@Override
	public void send() {
		System.out.println("【发送消息】www.mldn.cn");
	}
}
class MessageProxy implements IMessage { // 代理类
	private IMessage message ; // 代理对象，一定是业务接口实例
	public MessageProxy(IMessage message) {
		this.message = message ;
	}
	@Override
	public void send() {
```

```
            if (this.connect()) {
                this.message.send();  // 消息的发送处理
                this.close() ;
            }
        }
        public boolean connect() {
            System.out.println("【消息代理】进行消息发送通道的连接。");
            return true ;
        }
        public void close() {
            System.out.println("【消息代理】关闭消息通道。");
        }
}
```



以上的操作代码是一个最为标准的代理设计，但是如果要进一步的去思考会发现客户端的接口与具体的子类产生耦合问题，所以这样的操作如果从实际的开发来讲最好再引入工厂设计模式进行代理对象的获取。

以上的代理设计模式为静态代理设计，这种静态代理的特点在于：一个代理类只为一个接口服务，那么如果说现在准备出了3000个业务接口，则按照此种做法就意味着需要编写3000个代理类，并且这3000个代理类的操作形式类似。

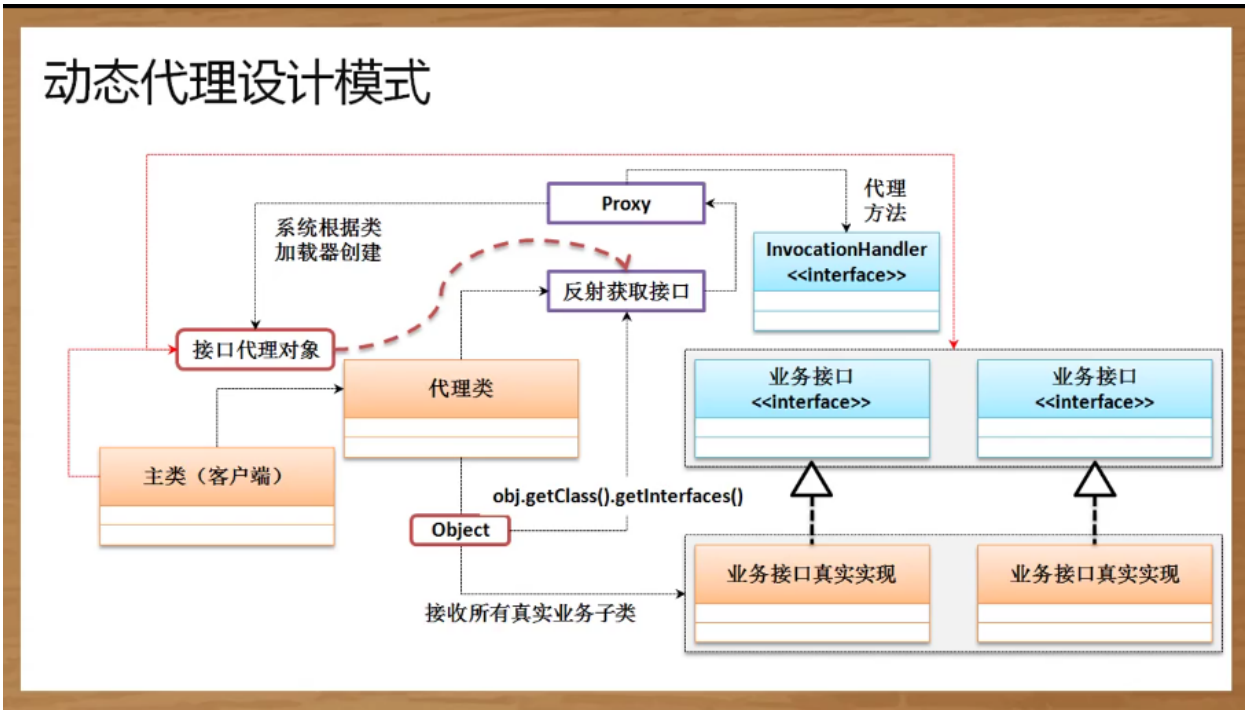
所以现在需要解决的问题在于：如何可以让一个代理类满足于所有的业务接口操作要求。

■动态代理设计模式

通过静态代理设计模式的缺陷可以发现，最好的做法是为所有功能一致的业务操作接口提供有统一的代理处理操作，而这就可以通过动态代理机制来实现，但是在动态代理机制里面需要考虑到如下几点问题：

·不管是动态代理类还是静态代理类都一定要接收真实业务实现子类对象；

·由于动态代理类不在于某一个具体的接口进行捆绑，所以应该可以动态获取类的接口信息；



在进行动态代理实现的操作之中，首先需要关注的就是一个InvocationHandler接口，这个接口规定了代理方法的执行。

```
public interface InvocationHandler {
        /**
         * 代理方法调用，代理主题类里面执行的方法最终都是此方法
         * @param proxy     要代理的对象
         * @param method 要执行的接口方法名称
         * @param args       传递的参数
         * @return       某一个方法的返回值
         * @throws Throwable    方法调用时出现的错误继续向上抛出
         */
         public Object invoke(Object proxy, Method method, Object[] args)
                        throws Throwable;

}
```

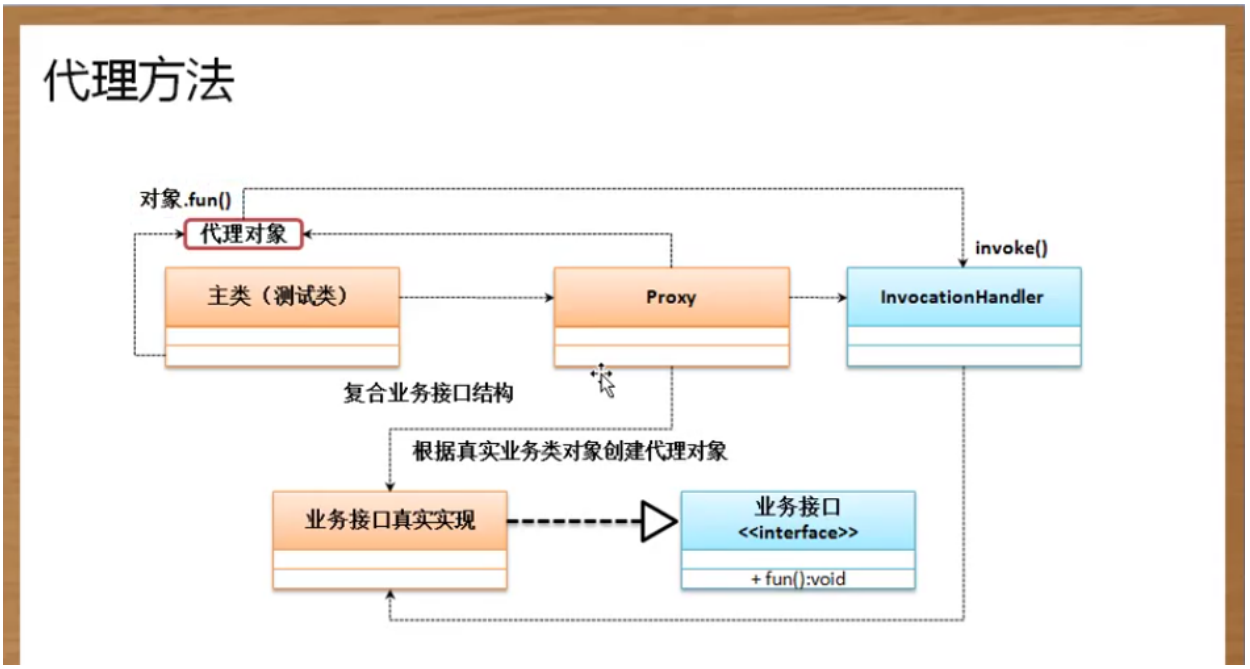
在进行动态代理设计的时候对于动态对象的创建是由JVM底层完成的，此时主要依靠的是java.lang.reflect.Proxy程序类，而这个程序类之中只提供有一个核心方法：

·代理对象：public static Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h);

|- ClassLoader loader：获取当前真实主体类的ClassLoader；

|- Class<?>[] interfaces：代理是围绕接口进行的，所以一定要获取真实主体类的接口信息；

|- ,InvocationHandler h：代理处理的方法；



范例：实现动态代理机制

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

public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		IMessage msg = (IMessage) new MLDNProxy().bind(new MessageReal()) ;
		msg.send();
	}
}
class MLDNProxy implements InvocationHandler {
	private Object target ; // 保存真实业务对象
	/**
	 * 进行真实业务对象与代理业务对象之间的绑定处理
	 * @param target 真实业务对象
	 * @return Proxy生成的代理业务对象
	 */
	public Object bind(Object target) {
		this.target = target ;
		return Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(), this);
	}
	public boolean connect() {
		System.out.println("【消息代理】进行消息发送通道的连接。");
		return true ;
	}
	public void close() {
		System.out.println("【消息代理】关闭消息通道。");
	}
	@Override
	public Object invoke(Object pro, Method method, Object[] args) throws Throwable {
		System.out.println("****** 【执行方法：】" + method);
		Object returnData = null ;
```

```
            if (this.connect()) {
                returnData = method.invoke(this.target, args) ;
                this.close() ;
            }
            return returnData;
        }
}
interface IMessage {       // 传统代理设计必须有接口
        public void send() ; // 业务方法
}
class MessageReal implements IMessage {
        @Override
        public void send() {
            System.out.println("【发送消息】www.mldn.cn");
        }
}
```

如果认真观察系统中提供的Proxy.newProxyInstance()方法会发现该方法会使用大量的底层机制来进行代理对象的动态创建，所有的代理类是符合所有相关功能需求的操作功能类，它不再代表具体的接口，这样在处理的时候就必须依赖于类加载器与接口进行代理对象的伪造。

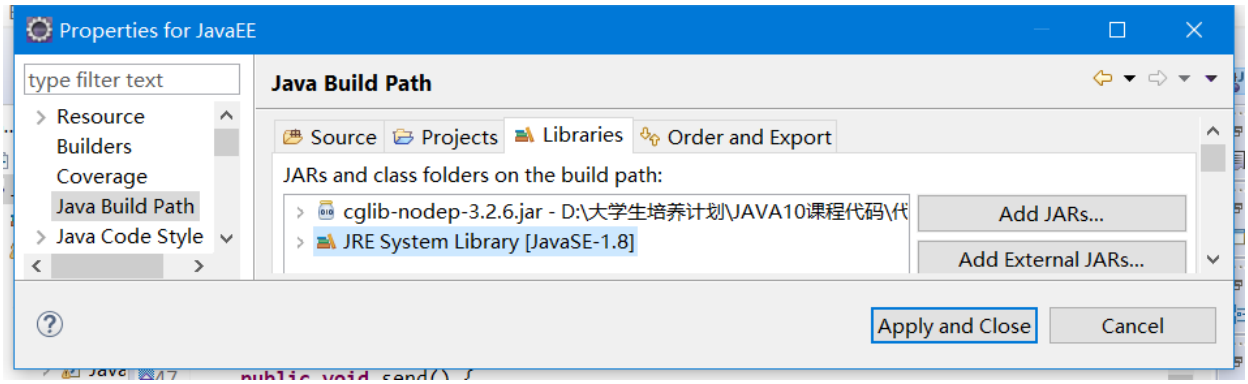
■CGLIB实现代理设计模式

从Java的官方来讲已经明确的要求了如果要想实现代理设计模式，那么一定是基于接口的应用，所以在官方给出的Proxy类创建对象时都需要传递该对象的所有接口信息：

```
Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(), this);
```

但是这个时候有一部分的开发者就认为不应该强迫性的基于接口来实现代理设计，所以一些开发者就开发出了一个CGLIB的开发包，利用这个开发包就可以实现基于类的代理设计模式。

1、CGLIB是一个第三方的程序包，需要单独在Eclipse之中进行配置，现在程序包的路径为：D:\jar-lib\cglib-nodep-3.2.6.jar，那么需要打开Eclipse项目属性安装第三方开发包：



2、编写程序类，该类不实现任何借口

```
class Message {
```

```
    public void send() {
        System.out.println("【发送消息】www.mldn.cn");
    }
}
```

3、利用CGLIB编写代理类，但是这个代理类需要做一个明确，此时相当于使用了类的形式实现了代理设计的处理，所以该代理设计需要通过CGLIB来生成代理对象，定义一个代理类：

```
class MLDNProxy implements MethodInterceptor {     // 拦截器配置
    private Object target ; // 保存真实主题对象
    public MLDNProxy(Object target) {
        this.target = target ;      // 保存真实主体对象
    }
    @Override
    public Object intercept(Object proxy, Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {
        Object returnData = null ;
        if (this.connect()) {
            returnData = method.invoke(this.target, args) ;
            this.close() ;
        }
        return returnData;
    }
    public boolean connect() {
        System.out.println("【消息代理】进行消息发送通道的连接。");
        return true ;
    }
    public void close() {
        System.out.println("【消息代理】关闭消息通道。");
    }
}
```

4、此时如果要想创建代理类对象，则就必须进行一系列的CGLIB处理。

```
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        Message realObject = new Message() ; // 真实主体对象
        Enhancer enhancer = new Enhancer() ; // 负责代理操作的程序类
        enhancer.setSuperclass(realObject.getClass()); // 假定一个父类
        enhancer.setCallback(new MLDNProxy(realObject)); // 设置代理类
        Message proxyObject = (Message) enhancer.create() ;// 创建代理对象
        proxyObject.send();
    }
}
```

在进行代理设计模式定义的时候除了可以使用接口之外，也可以不受接口的限制而实现基于类的代理设计，但是如果从正常的设计角度来讲，强烈建议还是基于接口的设计会比较合理。