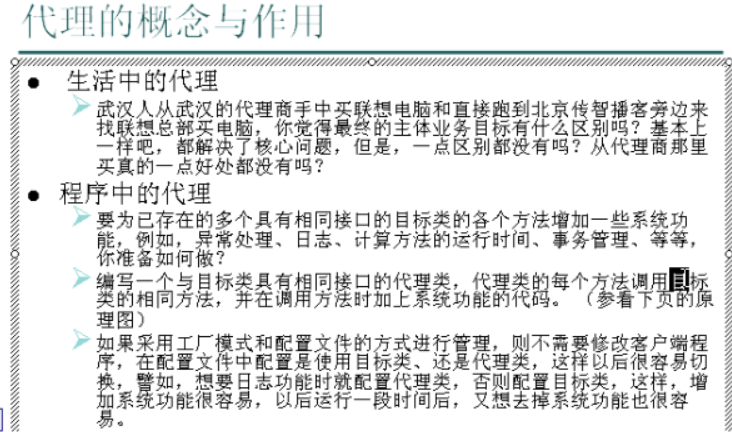
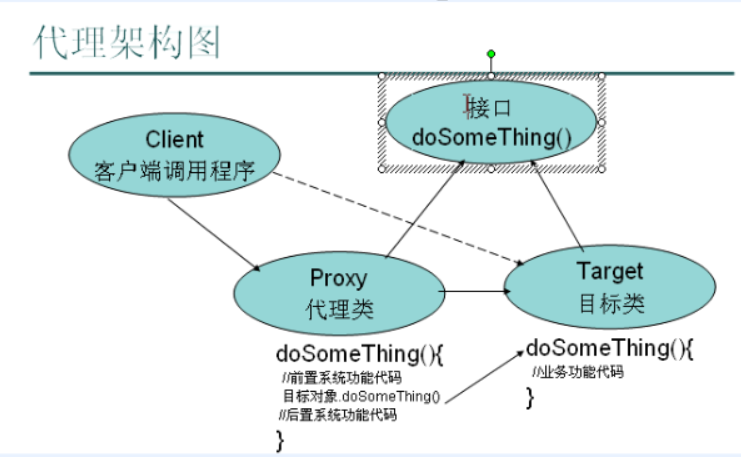
**代理+AOP**

# 代理的概念与作用



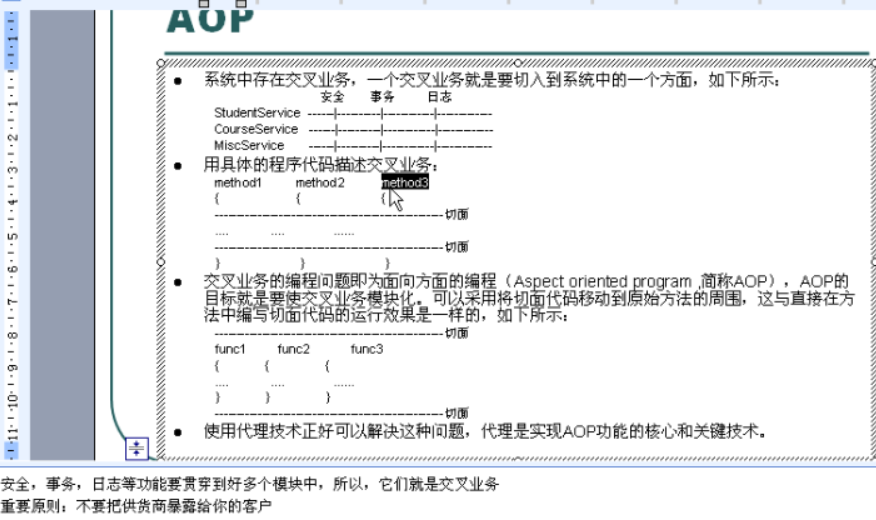
# 代理架构图



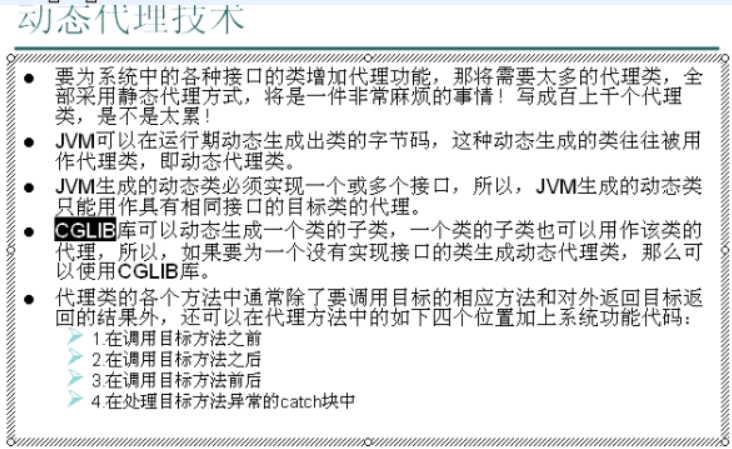
# 面向切面编程AOP：

系统的多个模块存在交叉业务（比如安全检验、事务管理、日志记录等），多个模块中相同的交叉业务连在一起就形成了系统的一个切面，AOP就是将交叉业务模块化（对象化：封装成对象），使之可以独立管理，降低和逻辑业务的耦合性。

同时结合声明式预编译和动态代理技术，方便的增加系统的业务功能。

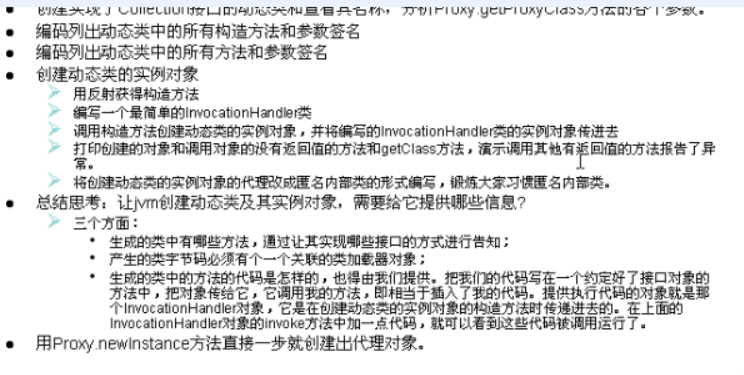


# 动态代理技术

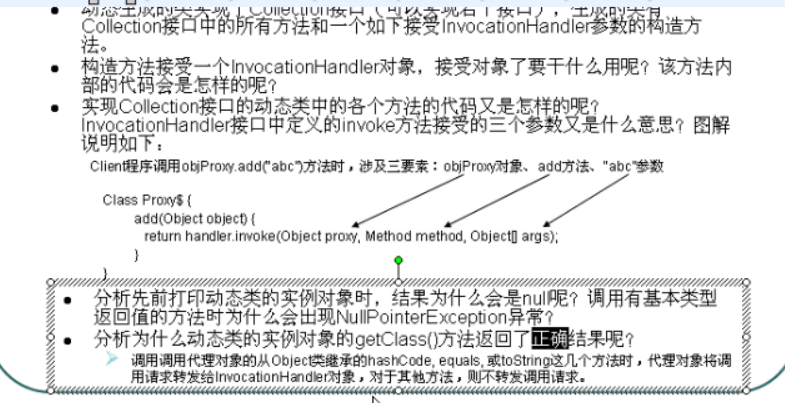


* 5 最终通知，相当于finally中的语句

## 实现JDK动态代理



## 深入分析InvocationHandler的invoke方法



调用基本类型返回值得方法时出现NullPointerException异常？

是因为当时没有在InvocationHandler中加入目标类对象，且返回值为null，而null往基本类型int转换时，无法转换，从而抛出异常。

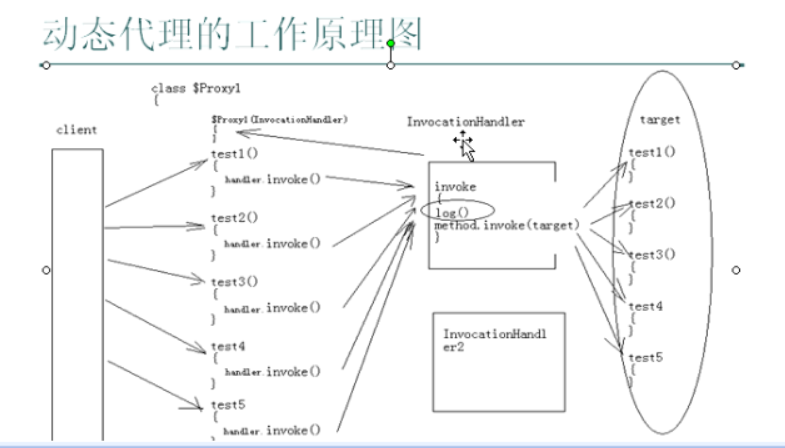
调用代理对象从Oject类继承的方法时：

1 如果是继承的hashCode、equals、toString 这三个方法，代理对象将调用请求转发给InvocationHandler对象的ivoke方法

2 对于其他从Object中继承的方法，则不转发调用请求，而是调用Proxy自己的方法。

（比如：proxy.getClass( )----> 则会调用代理类自己的这个方法，而不是去invoke中）

## 动态代理的工作原理图



代理对象

InvocationHandler

Target

## AOP底层原理：将系统功能模块，传入invoke中



# 手写Spring AOP功能和BeanFactory实现

AOP的实现依赖于动态代理。

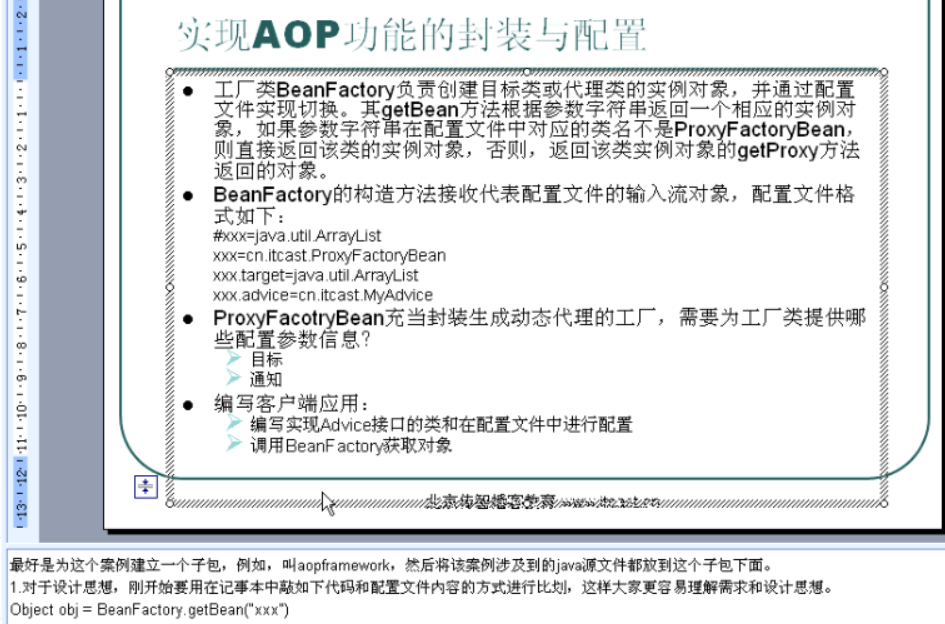
我们将系统功能封装成对象，并将其作为参数传入代理对象中，然后在代理对象中调用这个系统功能对象的方法。

所以，只需要向代理生成类中传入 目标类 和 系统功能对象（Advice通知），即可实现。

至于，具体传入什么目标类和Advice，可以通过配置文件设置，进而实现解耦。

设计思路：

|  |
| --- |
| 1 创建BeanFactory 通过配置文件config.properties 获取bean的路径名，在getBean方法中利用反射获取其实例instance  1.1 对instance进行判断，是普通bean，还是ProxyFactoryBean  1.2 普通bean，直接返回  1.3ProxyFactoryBean，则为其设置属性Target Advice ，然后调用getProxy()方法获取代理对象并返回  2 创建ProxyFactoryBean 该bean是用于创建代理对象  2.1设置属性Object target; 目标对象 Advice advice;通知对象  2.2在InvocationHandler的invoke中调用 目标类方法，和通知类的系统功能方法增强目标类。  3 创建Advice 通知接口 和 MyAdvice实现类  3.1 设置通知的位置：前置通知 后置通知 环绕通知 异常抛出通知 最终通知 （都是方法声明，具体内容由其实现类补充）  3.2 创建实现类MyAdvice并覆盖Advice中自己需要的接口  4 创建目标类    5 配置文件config.properties内容格式如下：beanName=xxx.xx  #普通测试的bean  #xxx=java.util.ArrayList  #代理工厂bean  xxx=MySpringAop.ProxyFactoryBean  #代理的目标类  xxx.target=java.util.ArrayList  #代理中的通知类  xxx.advice=MySpringAop.MyAdvice  6 测试类  6.1 创建BeanFactory 并传入配置文件的相对路径  6.2调用其getBean方法获取实例对象 |
|  |



BeanFactory代码

|  |
| --- |
| **package** MySpringAop;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.util.Properties;  /\*\*  \* BeanFacotry  \* 用于通过配置文件生产对象实例  \* **@author** Bang  \*  \*/  **public** **class** BeanFactory {    // 获取配置文件  Properties props = **new** Properties();  // 构造器传入配置文件路径 并加载配置文件  **public** BeanFactory(String configPath) {  InputStream ips = BeanFactory.**class**.getResourceAsStream(configPath);  **try** {  props.load(ips);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  /\*\*  \* 通过beanName获取实例对象  \* 1 判断是否为代理工厂bean，如果是代理工厂bean，则利用代理工厂bean的方法获取代理对象  \* 2 如果不是代理工厂bean，则直接通过反射创建实例对象返回  \* **@param** beanName  \* **@return**  \*/  **public** Object getBean(String beanName) {  Object result = **null**;  **try** {  // 通过beanName获取路径名  String className = props.getProperty(beanName);  // 通过反射创建对象  Object newInstance = Class.*forName*(className).newInstance();  // 判断该对象是否为ProxyFactoryBean  **if**(newInstance **instanceof** ProxyFactoryBean) {  //强转为ProxyFactoryBean  ProxyFactoryBean proxyFactoryBean = (ProxyFactoryBean)newInstance;  // 初始化proxyFactoryBean的属性 target advice  Advice advice = (Advice)Class.*forName*(props.getProperty(beanName+".advice")).newInstance();  Object target = Class.*forName*(props.getProperty(beanName+".target")).newInstance();  proxyFactoryBean.setAdvice(advice);  proxyFactoryBean.setTarget(target);  // 通过方法获取代理对象  result = proxyFactoryBean.getProxy();  }**else** {  // 如果不是ProxyFactoryBean，则直接返回bean  result = newInstance;  }  } **catch** (Exception e) {  e.printStackTrace();  }  **return** result;  }  } |

ProxyFactoryBean代码

|  |
| --- |
| **package** MySpringAop;  **import** java.lang.reflect.InvocationHandler;  **import** java.lang.reflect.Method;  **import** java.lang.reflect.Proxy;  /\*\*  \* 用户创建代理对象  \* **@author** Bang  \*  \*/  **public** **class** ProxyFactoryBean {  // 目标对象  **private** Object target;  // 通知  **private** Advice advice;      **public** Object getTarget() {  **return** target;  }  **public** **void** setTarget(Object target) {  **this**.target = target;  }  **public** Advice getAdvice() {  **return** advice;  }  **public** **void** setAdvice(Advice advice) {  **this**.advice = advice;  }  /\*\*  \* 获取代理对象  \* **@return**  \*/  **public** Object getProxy() {  **return** Proxy.*newProxyInstance*(  target.getClass().getClassLoader(), // 目标类加载器  target.getClass().getInterfaces(), // 目标类实现接口  **new** InvocationHandler() { // 调用处理器  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  Object result = **null**;  **try** {  // 前置通知  advice.beforeMethod();  result = method.invoke(target, args);  // 后置通知  advice.afterMethod();  } **catch** (Exception e) {  // 异常通知  advice.throwMethod();  }**finally** {  // 最终通知  advice.finallyMethod();  **return** result;  }  }  });    }  } |

Advice通知类接口

|  |
| --- |
| **package** MySpringAop;  **public** **interface** Advice {  // 前置通知  **void** beforeMethod();  // 后置通知  **void** afterMethod();  // 异常抛出通知  **void** throwMethod();  // 最终通知  **void** finallyMethod();  } |

MyAdvice通知接口具体实现类

|  |
| --- |
| **package** MySpringAop;  **public** **class** MyAdvice **implements** Advice{  @Override  **public** **void** beforeMethod() {  // **TODO** Auto-generated method stub  System.***out***.println("前置通知。。。。。。。。");  }  @Override  **public** **void** afterMethod() {  // **TODO** Auto-generated method stub  System.***out***.println("后置通知。。。。。。。。");  }  @Override  **public** **void** throwMethod() {  // **TODO** Auto-generated method stub  System.***out***.println("异常通知。。。。。。。。");  }  @Override  **public** **void** finallyMethod() {  // **TODO** Auto-generated method stub  System.***out***.println("最终通知。。。。。。。。");  }  } |

配置文件config.properties

|  |
| --- |
| #普通测试的bean  #xxx=java.util.ArrayList  #代理工厂bean  xxx=MySpringAop.ProxyFactoryBean  #代理的目标类  xxx.target=java.util.ArrayList  #代理中的通知类  xxx.advice=MySpringAop.MyAdvice |

测试类

|  |
| --- |
| **package** MySpringAop;  **public** **class** MyAopTest {  **public** **static** **void** main(String[] args) {  // 创建bean工厂  BeanFactory factory = **new** BeanFactory("config.properties");  // 获取对象  Object bean = factory.getBean("xxx");  System.***out***.println(bean.getClass().getName());    }  } |