一、DI 注解之 Autowired 和 Value 注解

1、DI 注解

因为之前 XML 配置方式比较繁琐。

```
<bean id="person" class="cn.wolfcode._01_hello.Person">
    <property name="name" value="大罗"/>
</bean>
```

分两类，Spring 的 Autowire，JavaEE 的 Resource，两者作用是一样，**完成属性或字段的注入，注入是 bean（取代 XML property ref 元素）**。而 Spring 的 Value 注解也是完成属性或字段的注入，注入是常量值（取代 XML property value 元素）。

2、Autowired 注解使用

2.1、编写类

```
package cn.wolfcode._04_anno;

public class Dog {
    @Value("黄色")
    private String color;

    @Override
    public String toString() {
        return "Dog [color=" + color + "]";
    }
}
```

```
package cn.wolfcode._04_anno;

public class Person {
    @Autowired
    private Dog dog;

    @Override
    public String toString() {
        return "Person [dog=" + dog + "]";
    }
}
```

2.2、编写配置文件

在 resources 目录下新建 04.anno.xml，配置如下：

```xml
<bean id="dog" class="cn.wolfcode._04_anno.Dog"/>

<bean id="person" class="cn.wolfcode._04_anno.Person"/>

<!-- 配置 DI 注解解析器，但其实可以不配置 -->
<context:annotation-config/>
```

2.3、编写单元测试类

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:04.anno.xml")
public class PersonTest {
    @Autowired
    private Person person;

    @Test
    public void test() {
        System.out.println(person);
    }
}
```

3、Autowired 注解细节

- 可以让 Spring 自动的把属性或字段需要的对象找出来，并注入到属性上或字段上。
- 可以贴在字段或者 setter 方法上面。
- 可以同时注入多个对象。
- 可以注入一些 Spring 内置的重要对象，比如 BeanFactory，ApplicationContext 等。
- 默认情况下 Autowired 注解必须要能找到对应的对象，否则报错。通过 required=false 来避免这个问题：@Autowired(required=false)。
- 第三方程序：Spring3.0 之前需要手动配置 Autowired 注解的解析程序：context:annotation-config/，在 Spring 的测试环境可以不配置，在 Web 开发中换一种配置解决。
- Autowired 注解寻找 bean 的方式：
  - 首先按照依赖对象的类型找，若找到，就是用 setter 或者字段直接注入。
  - 如果在 Spring 上下文中找到多个匹配的类型，再按照名字去找，若没有匹配报错。
  - 可以通过使用 @Qualifier("other") 标签来规定依赖对象按照 bean 的 id 和 类型的组合方式去找。

二、DI 注解之 Resource 注解

1、Resource 注解使用

修改上面代码的 Person 类使用 Resource 注解，再重新测试即可。

```java
package cn.wolfcode._04_anno;

@Component
public class Person {
    @Resource
    private Dog dog;

    @Override
    public String toString() {
        return "Person [dog=" + dog + "]";
    }
}
```

2、Resource 注解细节

- 可以让 Spring 自动的把属性或字段需要的对象找出来，并注入到属性上或字段上。
- 可以贴在字段或者 setter 方法上面。
- 可以注入一些 Spring 内置的重要对象，比如 BeanFactory，ApplicationContext 等。
- Resource 注解必须要能找到对应的对象，否则报错。
- 第三方程序：Spring3.0 之前需要手动配置 Autowired 注解的解析程序：context:annotation-config/，在 Spring 的测试环境可以不配置， Web 开发中换一种配置解决。
- Resource 注解寻找 bean 的方式：
  - 首先按照名字去找，如果找到，就使用 setter 或者字段注入。
  - 若按照名字找不到，再按照类型去找，但如果找到多个匹配类型，报错。
  - 可以直接使用 name 属性指定 bean 的名称（@Resource(name="")）；但若指定的 name，就只能按照 name 去找，若找不到，就不会再按照类型去。

3、Autowired 注解与 Resource 注解选用

都可以，出去后看公司。

三、IoC 注解

1、XML 配置问题

Spring 的 XML 配置文件中还有很多 bean 的配置，那能不能让 Spring 通过什么方式来简化配置呢?

```xml
<bean id="dog1" class="cn.wolfcode._04_anno.Dog">
    <property name="color" value="黄色"/>
</bean>

<bean id="dog2" class="cn.wolfcode._04_anno.Dog">
    <property name="color" value="黑色"/>
</bean>
```

2、IoC 注解

四个注解的功能是相同的，只是用于标注不同类型的类上：

- @Repository：用于标注数据访问组件，即 DAO 实现类上。

- @Service：用于标注业务层实现类上。
- @Controller：用于标注控制层类上（如 SpringMVC 的 Controller）。
- @Component：当不是以上的话，可以使用这个注解进行标注。

3、IoC 注解使用

3.1、修改类

修改上面代码的 Person 类和 Dog 类，使用 Component 注解。

```
package cn.wolfcode._04_anno;

@Component
public class Dog {
    @Value("黄色")
    private String color;

    @Override
    public String toString() {
        return "Dog [color=" + color + "]";
    }
}
```

```
package cn.wolfcode._04_anno;

@Component
public class Person {
    @Autowired
    private Dog dog;

    @Override
    public String toString() {
        return "Person [dog=" + dog + "]";
    }
}
```


@Component 不写 value 值的话，bean 名字就类名首字母小写，上面列子就是 bean 的名字就是 person。

3.2、修改配置文件

在 04.anno.xml，配置如下：

```
<!-- 配置注解的第三方程序  解析 IoC 注解 DI 注解，让这些注解起作用-->
<context:component-scan base-package="cn.wolfcode._04_anno"/>
```

改完配置再测试一下看效果。

四、Scope 和 PostConstruct 以及 PreDestroy 注解

1、问题

比如之前使用 XML 方式配置 bean 的作用域等能不能也使用注解的方式?

2、注解说明

- @Scope：贴在类上，标明 bean 的作用域。
- @PostConstruct：贴在方法上，标明 bean 创建完后调用此方法。
- @PreDestroy：贴在方法上，标明容器销毁时调用此方法。

3、注解使用

3.1、编写类

```
package cn.wolfcode._04_anno;

@Component
@Scope("prototype")
public class MyDataSource {
    public MyDataSource() {
        System.out.println("对象创建");
    }
    public void getConnection() {
        System.out.println("拿到连接");
    }
    @PostConstruct
    public void init() {
        System.out.println("初始化池子");
    }
    @PreDestroy
    public void close() {
        System.out.println("销毁池子");
    }
}
```


3.2、编写配置文件

配置文件内容不变，还是跟上面一样。

3.3、编写单元测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:04.anno.xml")
public class MyDataSourceTest {
    @Autowired
    private MyDataSource dataSource;

    @Test
    public void test() {
        dataSource.getConnection();
    }
}
```

五、练习-使用注解配置模拟用户注册

1、拷贝之前 XML 配置的项目

修改 pom.xml 的项目名为 spring-anno-register，再重新导入。

2、修改 DAO 层代码

```
package cn.wolfcode.dao.impl;

@Repository
public class StudentDAOImpl implements IStudentDAO {
    @Autowired
    private DataSource dataSource;

    @Override
    public void save(String username, String password) throws Exception {
        @Cleanup
        Connection connection = dataSource.getConnection();
        @Cleanup
        PreparedStatement ps = connection.prepareStatement("INSERT INTO
student(username, password) VALUES(?, ?)");
        ps.setString(1, username);
        ps.setString(2, password);
        ps.executeUpdate();
    }
}
```

3、修改业务层代码

```
package cn.wolfcode.service.impl;

@Service
public class StudentServiceImpl implements IStudentService {
    @Autowired
    private IStudentDAO studentDAO;

    @Override
    public void register(String username, String password) throws Exception {
        studentDAO.save(username, password);
    }
}
```

4、修改 applicationContext.xml

```
<!-- 引入 db.properties -->
<context:property-placeholder location="classpath:db.properties"/>

<!-- 配置 DataSource bean -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
    init-method="init" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
```

```
</bean>

<!-- 配置第三方解析程序，让 IoC DI 注解起作用 -->
<context:component-scan base-package="cn.wolfcode"/>
```

六、控制事务繁琐

1、问题

考虑一个应用场景：需要对系统中的某些业务方法做事务管理，拿简单的 save 操作举例。没有加上事务控制的代码如下：

```
public class EmployeeServiceImpl implements IEmployeeService {
    public void save(Employee employee){
        // 保存业务操作
    }
}
```

修改源代码，加上事务控制之后：

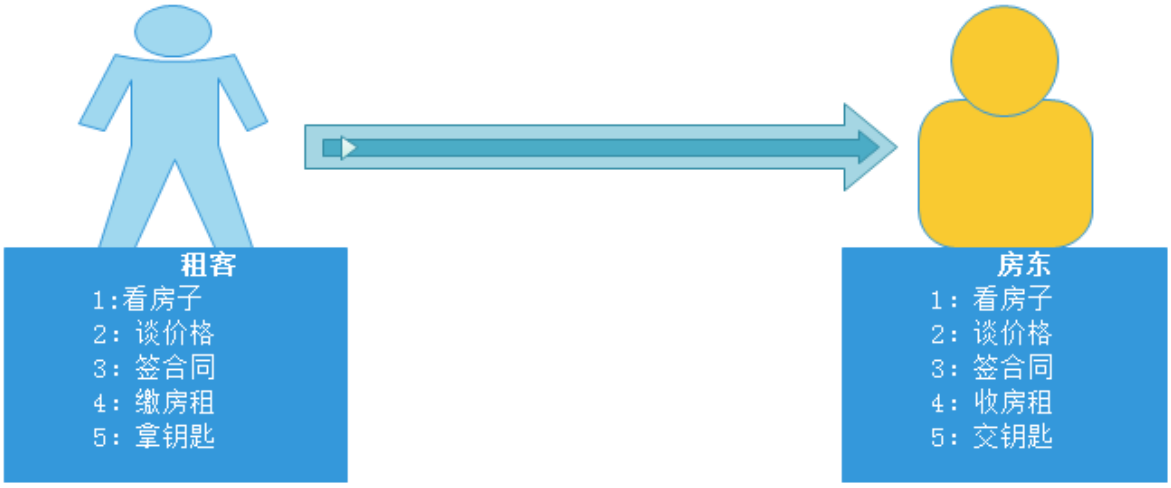
```
public class EmployeeServiceImpl implements IEmployeeService {
    public void save(Employee employee){
        // 打开资源
        // 开启事务
        try {
            // 保存业务操作
            // 提交事务
        }catch (Exception e){
            // 回滚事务
        }finally{
            // 释放资源
        }
    }
}
```


上述问题：在我们的业务层中每一个业务方法都得处理事务（繁琐的 try-catch），这样设计上存在两个很严重问题：

- 代码结构重复：在开发中不要重复代码，重复就意味着维护成本增大；
- 责任不分离：业务方法只需要关心如何完成该业务功能，不需要去关系事务管理、日志管理、权限管理等等。

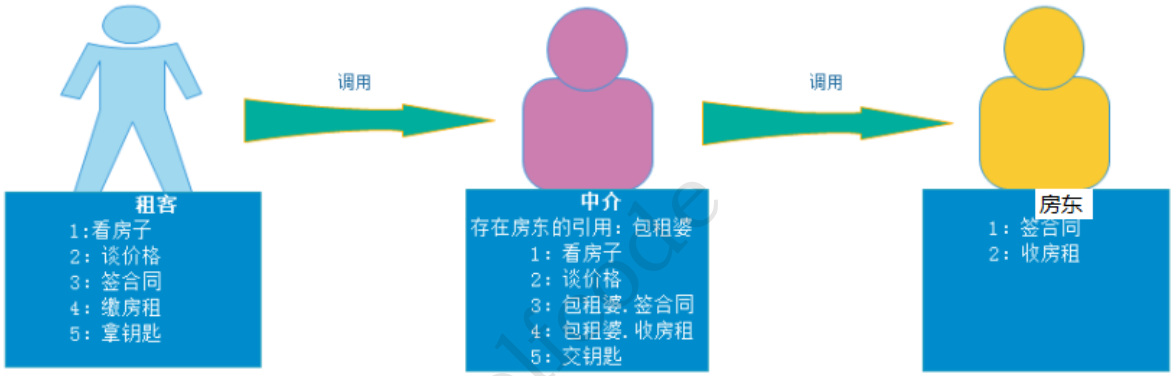
2、租房案例

租房情况一：直接联系房东：



问题：此时若有人来整房东，派很多人来找房东假租房，这会导致房东一天到晚都忙且没收获。带来这个问题就是：重复，且责任不分离，其实房东最关系的就是签合同和收房租。

租房情况二：通过中介：



后面我们就借鉴上面生活的例子的思想来之前事务繁琐的问题。

七、代理模式

1、好处

客户端直接使用的都是代理对象，不知道真实对象是谁，此时代理对象可以在客户端和真实对象之间起到中介的作用。

- 代理对象完全包含真实对象，客户端使用的都是代理对象的方法，和真实对象没有直接关系；
- 代理模式的职责：把不是真实对象该做的事情从真实对象上撇开—职责分离。

2、分类

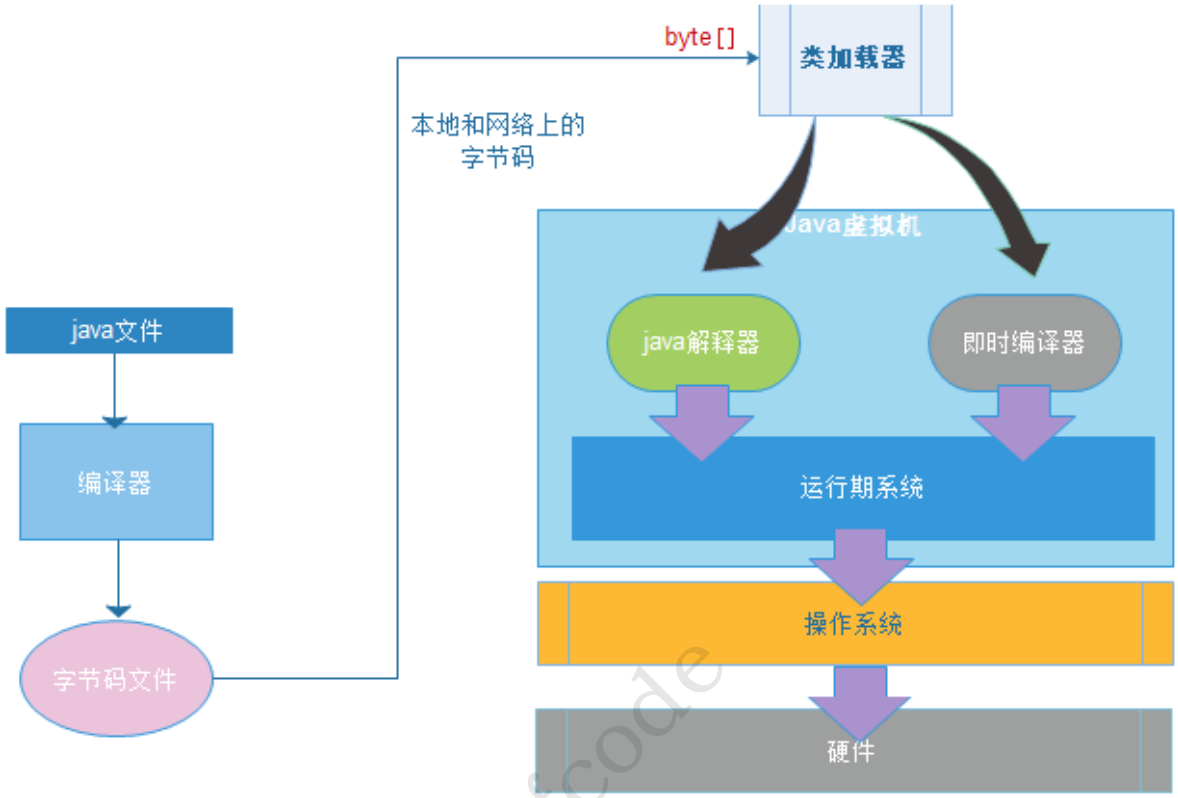
- 静态代理：在程序运行前就已经存在代理类的字节码文件，代理对象和真实对象的关系在运行前就确定了。（即代理类及对象要我们自己创建）
- 动态代理：代理类是在程序运行期间由 JVM 通过反射等机制动态的生成的，所以不存在代理类的字节码文件，动态生成字节码对象，代理对象和真实对象的关系是在程序运行时期才确定的。（即代理类及对象不要我们自己创建）

3、动态代理实现方式

- 针对真实类有接口使用 JDK 动态代理；
- 针对真实类没实现接口使用 CGLIB 或 Javassist 组件。

4、动态代理实现机制

由于 JVM 通过字节码的二进制信息加载类的，若我们在运行期系统中，遵循 Java 编译系统组织 .class 文件的格式和结构，生成相应的二进制数据，然后再把这个二进制数据加载转换成对应的类。如此，就完成了在代码中动态创建一个类的能力了。

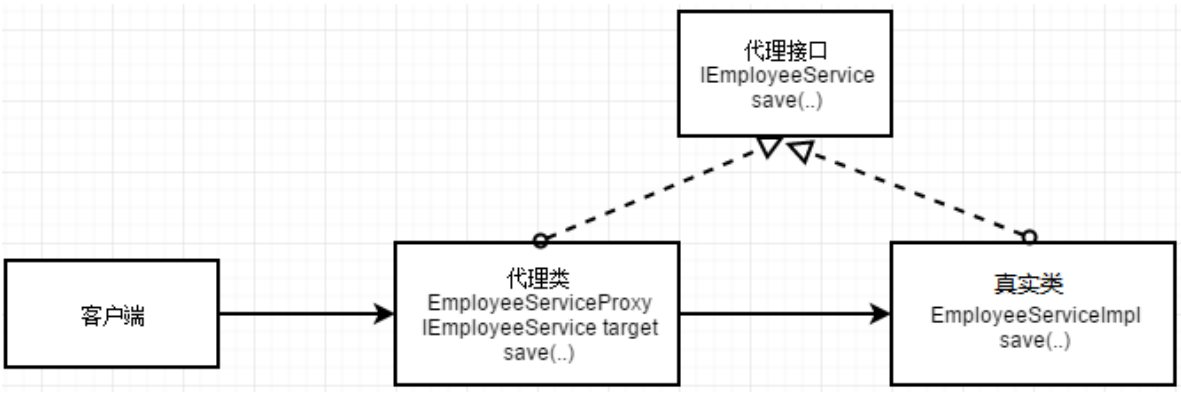


八、静态代理

1、需求

给业务类的方法增加模拟的事务。

2、类体系图



3、代码实现

新建一个 Maven 项目 static-proxy。

3.1、设置编译版本及添加依赖

```xml
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.8.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.0.8.RELEASE</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

3.2、编写业务接口及实现

```java
package cn.wolfcode.service;

public interface IEmployeeService {
    void save(String username, String password);
}
```

```java
package cn.wolfcode.service.impl;

// 房东
// 真实类，其对象为真实对象
public class EmployeeServiceImpl implements IEmployeeService {
    @Override
    public void save(String username, String password) {
        System.out.println("保存：" + username + ":" + password);
    }
}
```

3.3、编写代理类

```java
package cn.wolfcode.service.impl;

// 中介
// 代理类，其对象为代理对象
public class EmployeeServiceProxy implements IEmployeeService {

    // 房东引用，真实对象引用
```

```
    private IEmployeeService target;
    public void setTarget(IEmployeeService target) {
        this.target = target;
    }

    private MyTransactionManager tx;
    public void setTx(MyTransactionManager tx) {
        this.tx = tx;
    }

    @Override
    public void save(String username, String password) {
        try {
            tx.begin();
            // 保存的业务操作,找房东来做，找真实对象来做
            target.save(username, password);
            tx.commit();
        }catch (Exception e) {
            e.printStackTrace();
            tx.rollback();
        }
    }
}
```

3.4、编写事务模拟类


```
package cn.wolfcode.tx;

public class MyTransactionManager {
    public void begin() {
        System.out.println("开启事务");
    }
    public void commit() {
        System.out.println("提交事务");
    }
    public void rollback() {
        System.out.println("回滚事务");
    }
}
```

3.5、编写 applicationContext.xml

在 resources 目录新建 applicationContext.xml，配置如下：

```xml
<!-- 配置事务管理器对象 -->
<bean id="tx" class="cn.wolfcode.tx.MyTransactionManager"/>

<!-- 配置代理对象 -->
<bean id="employeeServiceProxy"
class="cn.wolfcode.service.impl.EmployeeServiceProxy">
    <property name="target">
        <!-- 配置真实对象，藏起来 -->
        <bean class="cn.wolfcode.service.impl.EmployeeServiceImpl"/>
    </property>
    <property name="tx" ref="tx"></property>
</bean>
```

3.6、编写单元测试类

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class EmployeeServiceTest {
    @Autowired
    private IEmployeeService proxy;

    @Test
    public void testSave() {
        // proxy 为代理对象
        proxy.save("罗老师", "666");
    }
}
```


4、优缺点

4.1、优点

- 业务类只需要关注业务逻辑本身，保证了业务类的重用性。
- 把真实对象隐藏起来了，保护真实对象。

4.2、缺点

- 代理对象的某个接口只服务于某一种类型的对象，也就是为每个真实类创建一个代理类，比如项目还有其他业务类呢。
- 若需要代理的方法很多，则要为每一种方法都进行代理处理。
- 若接口增加一个方法，除了所有实现类需要实现这个方法外，代理类也需要实现此方法。

九、JDK 动态代理

1、需求

给业务类的方法增加模拟的事务。

2、JDK 动态代理 API

2.1、java.lang.reflect.Proxy

Java 动态代理机制生成的所有动态代理类的父类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象。主要方法：public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler hanlder)

- 方法职责：为指定类加载器、一组接口及调用处理器生成动态代理类实例。
- 参数：
  - loader ： 类加载器，一般传递真实对象的类加载器；
  - interfaces：代理类需要实现的接口；
  - handler：代理执行处理器，说人话就是生成代理对象帮你要做什么。
- 返回：创建的代理对象。

2.2、java.lang.reflect.InvocationHandler

主要方法：public Object invoke(Object proxy, Method method, Object[] args)。

- 方法职责：负责集中处理动态代理类上的所有方法调用，让使用者自定义做什么事情， 对原来方法增强（加什么功能）。
- 参数：
  - proxy ： 生成的代理对象；
  - method：当前调用的真实方法对象；
  - args ： 当前调用方法的实参。
- 返回：真实方法的返回结果。

3、代码实现

3.1、拷贝静态代理项目

- 修改项目名为 jdk-dynamic-proxy，重新导入；
- 删除自己手写的代理类，配置文件报错顺便解决。

3.2、编写 TransactionInvocationHandler 类

该实现 InvocationHandler 接口，实现 invoke 方法，实现增强操作。

```
package cn.wolfcode.handler;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import cn.wolfcode.tx.MyTransactionManager;

public class TransactionInvocationHandler implements InvocationHandler {

    private Object target;
    public void setTarget(Object target) {
        this.target = target;
    }
    public Object getTarget() {
        return target;
    }

    private MyTransactionManager tx;
    public void setTx(MyTransactionManager tx) {
        this.tx = tx;
    }
```

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("到此一游");
        Object retVal = null;
        try {
            tx.begin(); // 调用事务管理器对象的方法
            retVal= method.invoke(target, args); // 调用真实对象的方法
            tx.commit(); // 调用事务管理器对象的方法
        }catch (Exception e) {
            tx.rollback(); // 调用事务管理器对象的方法
        }
        return retVal; // 返回方法调用的结果
    }
}
```

3.3、修改 applicationContext.xml

配置 TransactionInvocationHandler、MyTransactionManager、EmployeeServiceImpl，让 Spring 帮我们创建这些对象组装依赖。

```
<!-- 配置事务管理器对象 -->
<bean id="tx" class="cn.wolfcode.tx.MyTransactionManager"/>

<!-- 配置处理器执行器对象 -->
<bean id="transactionInvocationHandler"
class="cn.wolfcode.handler.TransactionInvocationHandler">
    <property name="target">
        <bean class="cn.wolfcode.service.impl.EmployeeServiceImpl"/>
    </property>
    <property name="tx" ref="tx"/>
</bean>
```


3.4、修改单元测试类

注入类型 InvocationHandler 的 bean，在测试方法中手动使用 Proxy 中的方法创建代理对象，调用代理对象的方法。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class EmployeeServiceTest {
    @Autowired
    private TransactionInvocationHandler transactionInvocationHandler;

    @Test
    public void testSave() {
        // 动态生成代理类，并创建代理对象
        IEmployeeService proxy = (IEmployeeService)Proxy.newProxyInstance(

 transactionInvocationHandler.getTarget().getClass().getClassLoader(),
            // 获取真实类实现的接口，生成的代理类也是实现什么的接口
            transactionInvocationHandler.getTarget().getClass().getInterfaces(),
            // 通过这个参数告诉 API 生成代理对象具体做什么
            transactionInvocationHandler);
        proxy.save("罗老师", "666");
```

```
    }

}
```

4、优缺点

4.1、优点

- 对比静态代理，发现不需手动地提供那么多代理类。

4.2、缺点

- 真实对象必需实现接口（JDK 动态代理特有）；
- 动态代理的最小单位是类（类中某些方法都会被处理），如果只想拦截一部分方法，可以在 invoke 方法中对要执行的方法名进行判断；
- 对多个真实对象进行代理的话，若使用 Spring 的话配置太多了，要手动创建代理对象，用起来麻烦。

十、JDK 动态代理原理

1、获取代理类字节码文件

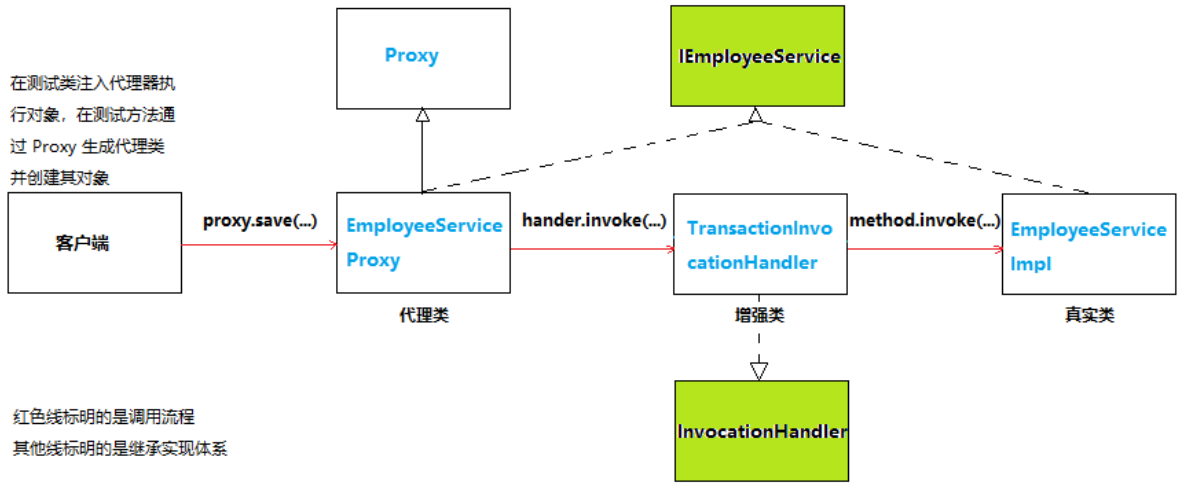
在包路径 cn.wolfcode.service.impl 下新建一个类 DynamicProxyClassGenerator，拷贝如下代码：

```
import java.io.FileOutputStream;
import sun.misc.ProxyGenerator;

@SuppressWarnings("restriction")
public class DynamicProxyClassGenerator {
    public static void main(String[] args) throws Exception {
        generateClassFile(EmployeeServiceImpl.class, "EmployeeServiceProxy");
    }
    public static void generateClassFile(Class<?> targetClass, String proxyName)
throws Exception {
        // 根据类信息和提供的代理类名称，生成字节码
        byte[] classFile = ProxyGenerator.generateProxyClass(proxyName,
targetClass.getInterfaces());
        String path = targetClass.getResource(".").getPath();
        System.out.println(path);
        FileOutputStream out = null;
        // 保留到硬盘中
        out = new FileOutputStream(path + proxyName + ".class");
        out.write(classFile);
        out.close();
    }
}
```

2、通过反编译工具查看字节码文件

使用 IDEA 就可以反编译。观察：save 方法，发现底层其实依然在执行 InvocationHandler 中的 invoke 方法。

3、类体系及调用流程



十一、CGLIB 动态代理

1、JDK 动态代理的问题

JDK 动态代理要求真实类必须实现接口。而 CGLIB 与 JDK 动态代理不同是，真实类不用实现接口，生成代理类的代码不一样且代理类会继承真实类。

2、CGLIB 动态代理 API

2.1、org.springframework.cglib.proxy.Enhancer

类似 JDK 中 Proxy，用来生成代理类创建代理对象的。

2.2、org.springframework.cglib.proxy.InvocationHandler

类似 JDK 中 InvocationHandler，让使用者自定义做什么事情，对原来方法增强。

3、代码实现

3.1、拷贝 JDK 动态代理项目

修改项目名为 jdk-cglib-proxy，重新导入。

3.2、修改 TransactionInvocationHandler 类

修改其实现接口为 org.springframework.cglib.proxy.InvocationHandler，其他不变。

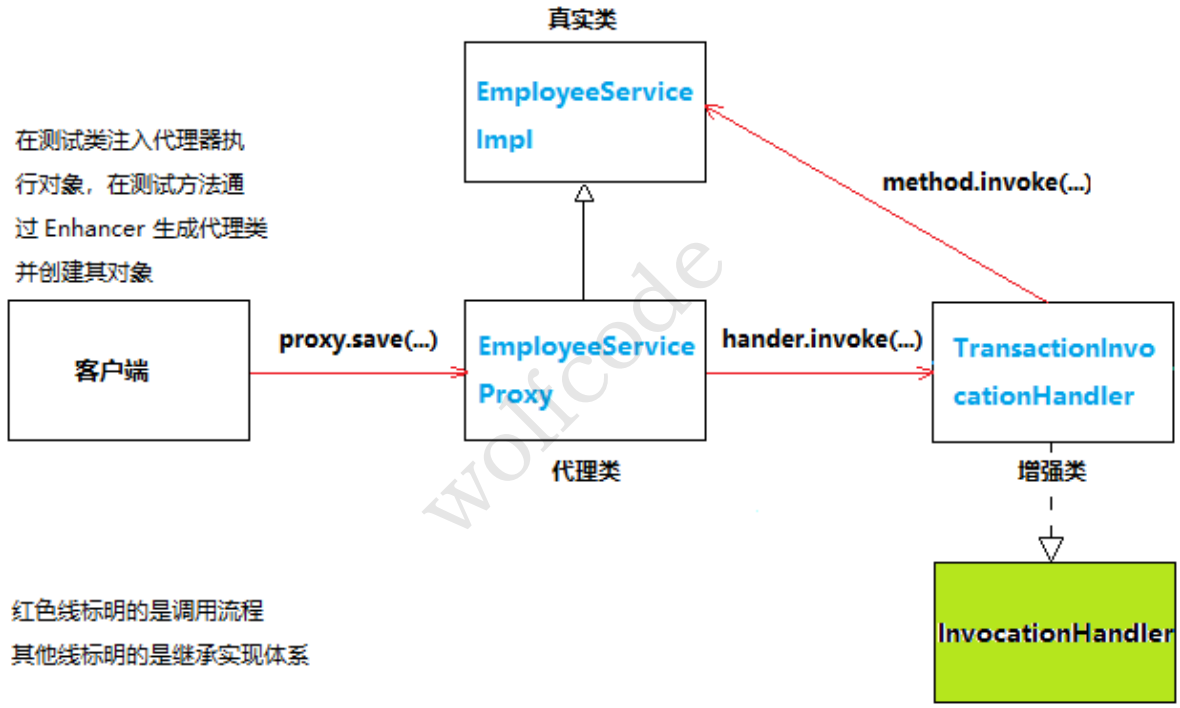
3.3、修改单元测试类

改用 Enhancer API 来生成代理类创建代理对象。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class EmployeeServiceTest {

    @Autowired
    private TransactionInvocationHandler transactionInvocationHandler;
```

```
    @Test
    public void testSave() {
        Enhancer enhancer = new Enhancer();
        // 设置生成代理类继承的类

enhancer.setSuperclass(transactionInvocationHandler.getTarget().getClass());
        // 设置生成代理类对象对象，要做什么事情
        enhancer.setCallback(transactionInvocationHandler);
        // 生成代理类，创建代理对象
        EmployeeServiceImpl proxy = (EmployeeServiceImpl)enhancer.create();
        proxy.save("罗老师", "666");
    }

}
```

4、类体系及调用流程



十二、动态代理总结

1、JDK 动态代理总结

- Java 动态代理是使用 java.lang.reflect 包中的 Proxy 类与 InvocationHandler 接口这两个来完成的。
- 要使用 JDK 动态代理，真实类必须实现接口。
- JDK 动态代理将会拦截所有 pubic 的方法（因为只能调用接口中定义的方法），这样即使在接口中增加了新的方法，不用修改代码也会被拦截。
- 动态代理的最小单位是类（类中某些方法都会被处理），如果只想拦截一部分方法，可以在 invoke 方法中对要执行的方法名进行判断。
- **代理类与真实类共同实现一个接口的。**

2、CGLIB 动态代理总结

- CGLIB 可以生成真实类的子类，并重写父类非 final 修饰符的方法。

- 要求类不能是 final 的，要拦截的方法要是非 final、非 static、非 private 的。
- 动态代理的最小单位是类（类中某些方法都会被处理），如果只想拦截一部分方法，可以在 invoke 方法中对要执行的方法名进行判断。
- **代理类是继承真实类。**

3、选用

JDK 动态代理是基于实现接口的，CGLIB 和 Javassit 是基于继承真实类的。

从性能上考虑：Javassit > CGLIB > JDK。

选用如下：

- 若真实类实现了接口，优先选用 JDK 动态代理。（因为会产生更加松耦合的系统，也更符合面向接口编程）
- 若真实类没有实现任何接口，使用 Javassit 和 CGLIB 动态代理。