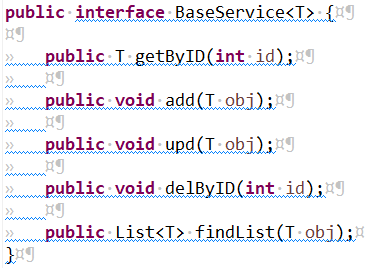
# Spring中的设计模式

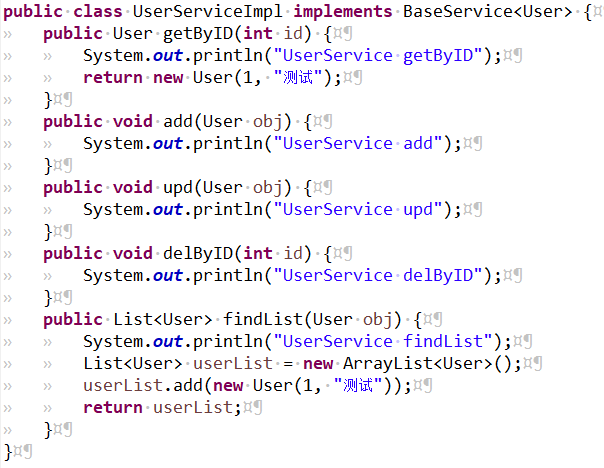
## 静态工厂

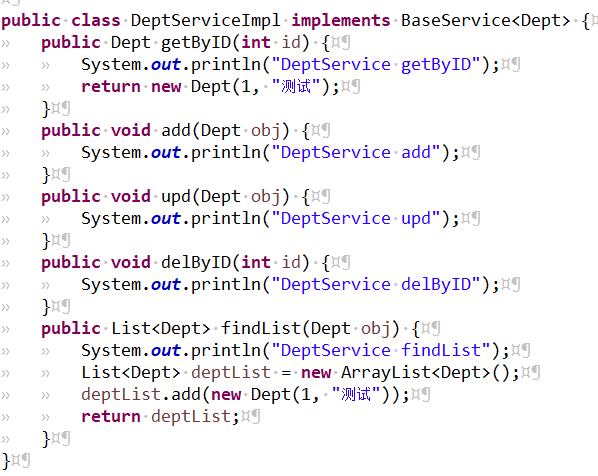
这个其实不属于GOF23 模式，但是在Spring中，IOC的根本BeanFactory就是这个，还是要学习一下。我们看代码。

设计一个Service接口，有增删改查查列表5个方法：

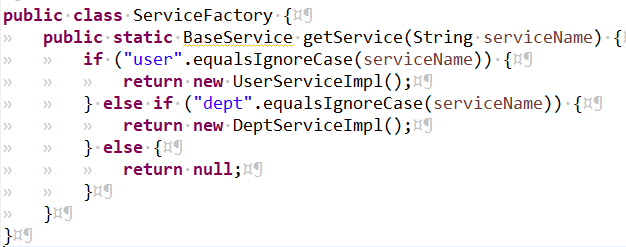


有两个Service实现类：用户、部门：

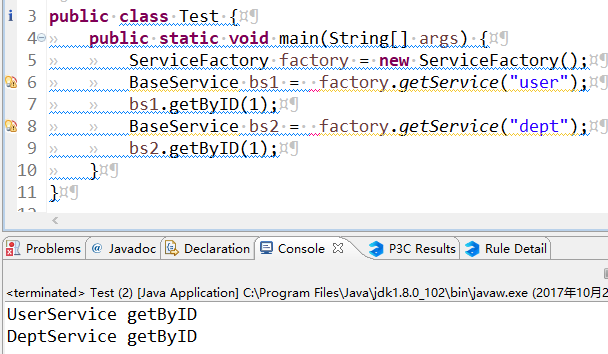




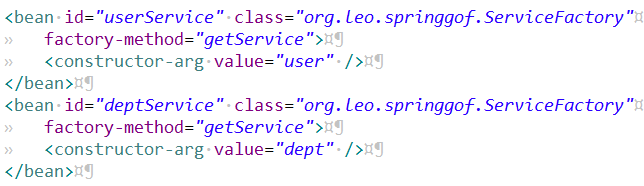
静态工厂类如下，根据传入的唯一参数去决定去创建哪个Service：



测试类：



在Spring里，其实我们可以用XML配置要创建的对象，比如：



然后通过ClassPathXmlApplicationContext去加载这个配置，然后通过getBean()方法去获取实例。

这种模式，我之前在接手一个蛮久远的项目代码时见过，那个项目很传统，多是增删改查。Service里的方法很固定，所以当时的架构就用这种方式来生成Service实例的。

我们现在用SSH、SSM框架的时候，比如我们要在Controller里引用Service，应该多是用@Autowired去注入多个Service，使用的时候类似userService.getByID(1)，deptService.getByID(1)这样。

而在那个项目中基本都是service.getByID(1)这种。

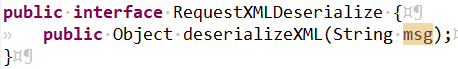
## 适配器模式

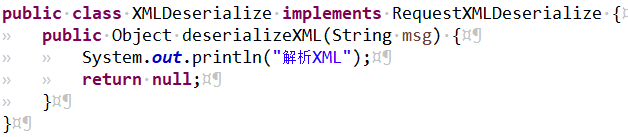
假设有这种情况，根据协议，第三方系统发送XML格式的消息给我们，我们解析XML转成实体类，然后进行处理。反序列化由专门的类进行负责。

后来觉得XML传输有点费空间，大家商量把XML改为JSON，如果不考虑设计模式，我们可以粗暴一点，直接再新搞一个反序列化JSON的类和方法，然后整个工程中所有用到原来的反序列化类的地方，全部改成新的反序列化类和方法，这么搞是不是工程量有点大啊。

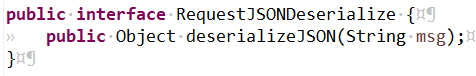
为了不修改客户端（或者叫调用方）的代码，我们在被调用方可以用适配器模式。

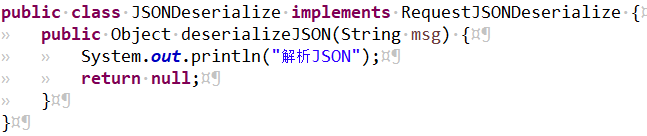
XML反序列化接口与实现类：



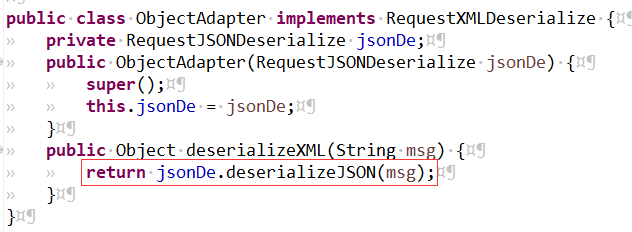


JSON反序列化接口与实现类：



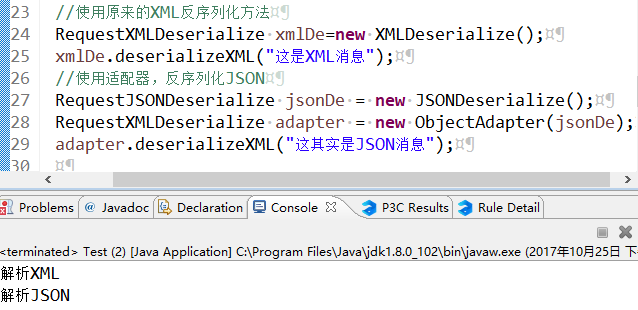


对象适配器：

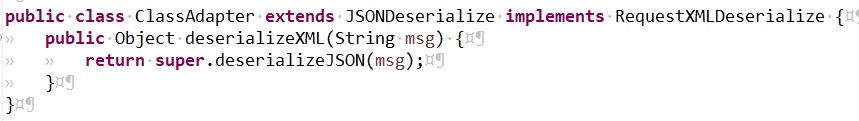


代码看起来很简单，就是在原来的XML反序列化接口前加了一层，李代桃僵，移花接木，不再使用原来XML的反序列化方法，取而代之的是JSON反序列化方法。

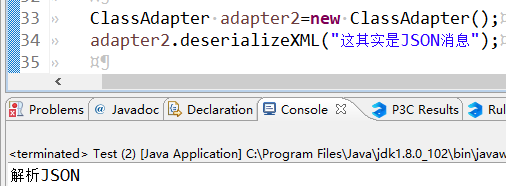
调用方法：



接着还有类适配器：

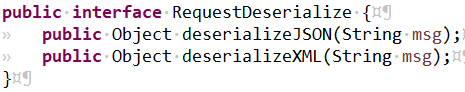


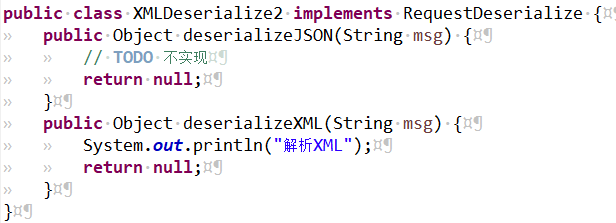
调用方法：

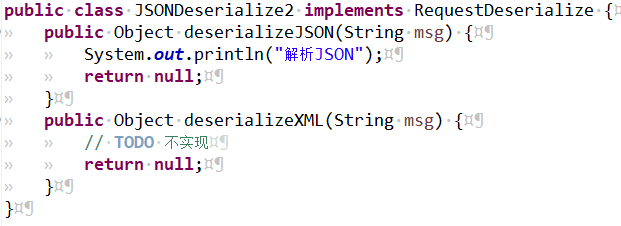


以上两种只是实现了对原有方法的替换，如果在工作中想保留多个方法，可以采用如下方法。

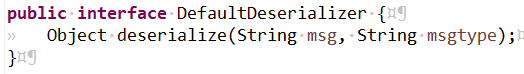
具体实现类与接口：

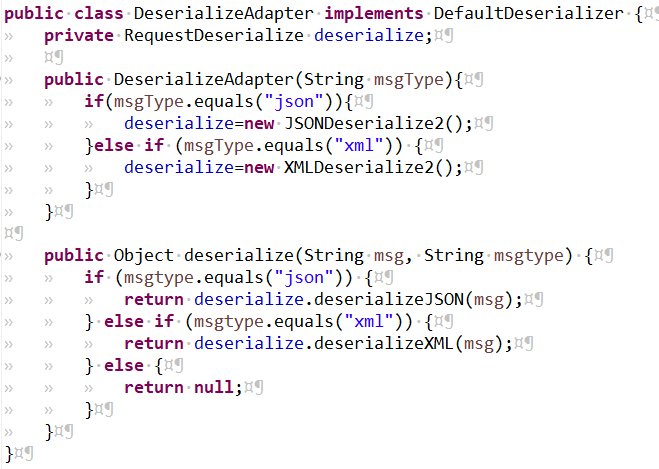




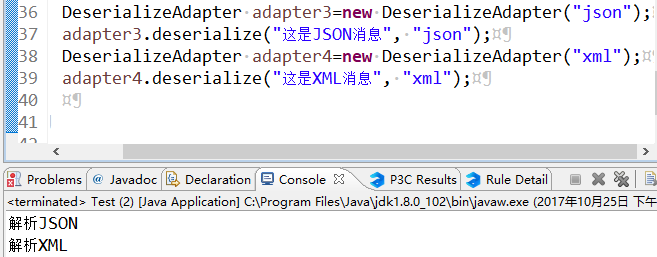


适配器实现类与接口：





调用方法：



如上图，这样不管被调用的方法如何变，只要把“json”、“xml”放在配置文件里，调用方引用即可，没什么代码要改。

在Spring里，我们用到的AOP中的Advice用的就是适配器模式。将各种类型的Advice通过适配器模式，封装成对应的拦截器类型，返回给容器。

在这里多说两句。

相信很多程序员对设计模式是心情比较复杂的，觉得这东西在实际工作中极少用到，但是有时候去面试的时候，设计模式又肯定被问到。

这个就让人很纠结了。

我说一下我的理解，抛砖引玉吧。

首先，如果你写代码是为了一个项目，我觉得设计模式这些确实不重要，重要的是对需求的理解，对业务的设计。因为一个项目基本上是针对一个公司或单位，不管你对这个项目做多少次升级，在技术结构上的改动是很小的，多是一些业务上的变动——而这些变动是那种大的、翻天覆地的变动的几率也是很小的。就拿上面我举的例子来讲，你能想象之前我们制定了消息是XML的设计，今天刚做完发版，明天客户就要求改成JSON，后天改成protobuf这种情况么？

但是如果你是为一个产品写代码，那么作为一个有经验的老程序员，在很多地方你不仅要考虑业务，也要考虑设计模式。

上面反序列化的例子其实就是我在工作中遇到的。那是我们的一个产品，有个接口需要跟第三方对接，但是每个客户所对应的接口方是不一样的。我们的产品又不是那种牛X哄哄的产品，我们公司也不是那种牛X哄哄的公司。所以这个接口的协议很多时候要跟接口方一起商定。

一开始我也是经验不足，就没考虑多种协议这种事，原来订下XML，但是后来各个接口方有要求用这的，有要求用那的。

写反序列化的代码简单，加个类而已，但是各处调用原反序列化方法的代码都要改啊，这很要人命的，我得维护多少版代码？

如果我对适配器模式不了解，那么也就只能忍了。

好在后来对这一块做了改造（参照第三种方式），以后再有新的反序列化方式，也就是增加实现类，改一改Adapter，把配置文件中协议的类型改了即可。

## 模板方法模式

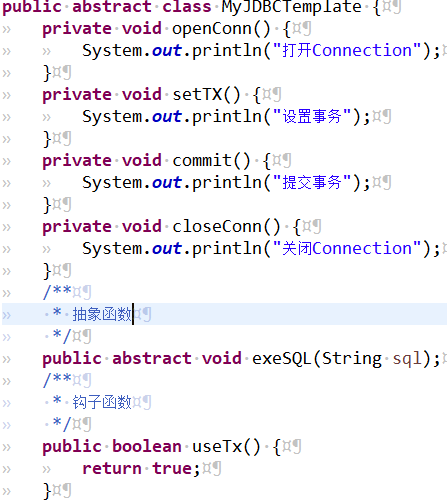
这个其实最容易理解，也是最经常见到的，比如我们在Spring中最常见的就是JdbcTemplate、HibernateTemplate，结合简单的代码来谈谈。

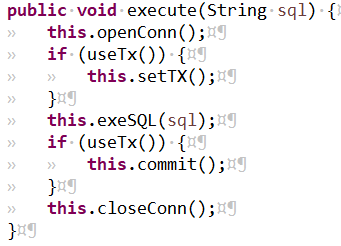
我们把执行SQL语句的步骤简化如下：

|  |
| --- |
| 1. 打开Connection 2. 设置事务 3. 执行SQL 4. 提交事务 5. 关闭Connection |

这个顺序没法省，也没法打乱。大家也看出来了，如果我们每执行一次SQL，都要按这种顺序写代码，代码重复的太多了。模板方法模式的解决方案如下。

模板类：





没啥难度，看execute()这个方法的代码应该就能很好理解了。

这里要多说一下钩子方法，也就是useTx()。

我们执行SQL语句，有的是需要启用事务的（增删改），有的则不需要（查），但是抽象类是不知道哪个SQL用哪个不用的，这个选择权就只能交给了具体的实现类。

useTX()方法并不是执行SQL所必须的步骤，但是它却可以将各个步骤进行挂钩。

你想用事务，将useTX()返回true，execute()就会执行事务相关的方法——通过useTX()这个钩子，把事务方法挂了上去。

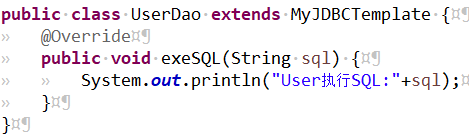
你不想用事务，将useTX()返回false，execute()就不执行事物相关的方法——通过useTX()这个钩子，把事务方法从原方法上脱了钩。

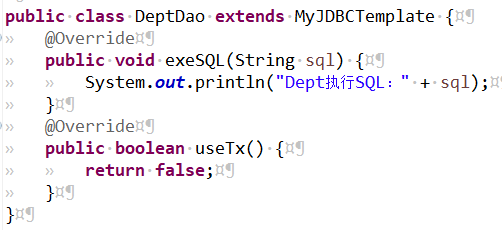
以前那种老式火车，各个车厢是不是用钩子连接的？



而所谓的钩子方法，在抽象类中，我们可以将其设置为空代码，也可以像上面的代码一样，给个默认的实现。那么在子类中我们就可以根据实际情况去决定是否要重写这个方法，从而根据这个方法，来影响具体的步骤。

实现类：





测试结果：

