Spring 的本质系列(1) -- 依赖注入

原创： 刘欣

> 前言： Spring 这个轻量级的框架已经成为Web开发事实上的标准， 不少同学建议我写一些Spring相关的技术， 我刚开始是拒绝的， 因为现在网上相关的文章是在太多了。
>
> 后来想想，这些文章更多的关注细节和实现，教大家怎么用， 关注how , 真正讲解why 的还不多， 我觉得我可以给大家分享下**我对Spring本质的感想和体会**，这就是这篇文章的由来， 如果大家喜欢，就继续的写下去。
>
> 希望你对OO,设计模式，单元测试，XML，反射等技术有一定了解， 不了解也没关系，直接看下去就是了 :-)

1 对象的创建

面向对象的编程语言是用类(Class)来对现实世界进行抽象， 在运行时这些类会生成对象(Object)。

当然，单独的一个或几个对象根本没办法完成复杂的业务， 实际的系统是由千千万万个对象组成的，这些对象需要互相协作才能干活，例如对象A调用对象B的方法，那必然会提出一个问题：对象A怎么才能获得对象B的引用呢?

最简单的办法无非是： 当对象A需要使用对象B的时候， **把它给new 出来** ，这也是最常用的办法，java 不就是这么做的? 例如：

```
Apple a = new Apple();
```

后来业务变复杂了， 抽象出了一个水果(Fruit)的类， 创建对象会变成这个样子：

```
Fruit f1 = new Apple();
Fruit f2 = new Banana();
Fruit f3 = ......
```

很自然的，类似下面的代码就会出现：

```
Fruit f = null;
if ("apple".equals(type)) {
    f = new Apple();
} else if("banana".equals(type)) {
    f = new Banana();
}
```

这样的代码如果散落在各处，维护起来将会痛苦不堪， 例如你新加一个水果的类型Orange, 那得找到系统中所有的这些创建Fruit的地方，进行修改， 这绝对是一场噩梦。

解决办法也很简单， 前辈们早就总结好了：工厂模式

```
public class FruitFactory {
    public static Fruit create(String type) {
        if("apple".equals(type)) {
            return new Apple();
        }
        if("banana".equals(type)) {
            return new Banana();
        }
        return null;
    }
}
```

工厂模式，以及其他模式像抽象工厂， Builder模式提供的都是创建对象的方法。

这背后体现的都是"**封装变化**"的思想。

这些模式只是一些最佳实践而已： 起了一个名称、描述一下解决的问题、使用的范围和场景，码农们在项目中还得自己去编码实现他们。

2 解除依赖

我们再来看一个稍微复杂一点， 更加贴近实际项目的例子：

一个订单处理类，它会被定时调用： 查询数据库中订单的处理情况， 必要时给下订单的用户发信。

```
public class OrderProcessor {
    public void process() {
        OrderService oderService = OrderService.getInstance();
        EmailService mailService = EmailService.getInstance();
        List<Order> orders = oderService.getOrders();
        for(Order order:orders) {
            OrderEmail emial = null;
            mailService.sendMail(email);
        }
    }
}
```

看起来也没什么难度， 需要注意的是很多类一起协作了， 尤其是OrderProcessor , 它依赖于

OrderService 和 EmailService这两个服务，它获取依赖的方式就是通过**单例方法**。

如果你想对这个process方法进行单元测试--**这也是很多优秀的团队要求的**-- 麻烦就来了。

首先OrderService 确实会从真正的数据库中取得Order信息，你需要确保数据库中有数据， 数据库连接没问题，实际上如果数据库连接Container（例如Tomcat）管理的， 你没有Tomcat很难建立数据库连接。

其次这个EmailService 真的会对外发邮件， 你可不想对真正的用户发测试邮件，当然你可以修改数据库，把邮件地址改成假的，但那样很麻烦， 并且EmailService 会抛出一堆错误来，很不爽。

所有的这些障碍， 最终会导致脆弱的单元测试： **速度慢， 不可重复，需要手工干预，不能独立运行。**

想克服这些障碍， 一个可行的办法就是不在方法中直接调用OrderService和EmailService的getInstance()方法， 而是把他们通过setter方法传进来。

```
public void setOrderService(OrderService orderService) {
    this.orderService = orderService;
}
public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
}
```

通过这种方式，你的单元测试就可以构造一个假的OrderService 和假的EmailService 了。

例如OrderService 的冒牌货可以是MockOrderService , 它可以返回你想要的任何Order 对象， 而不是从数据库取。

MockEmailService 也不会真的发邮件， 而是把代码中试图发的邮件保存下来， 测试程序可以检查是否正确。

你的测试代码可能是这样的：

```
public void testOrderProcess() {
    OrderProcessor op = new OrderProcessor();
    op.setOrderService(new MockOrderService());
    op.setEmailService(new MockEmailService());
    op.process();
}
```

当然， 有经验的你马上就会意识到： 需要把OrderService 和 EmailService 变成 接口或者抽象类， 这样才可以把Mock对象传进来。

这其实也遵循了面向对象编程的另外一个要求： **对接口编程， 而不是对实现编程。**

3 Spring 依赖注入

啰啰嗦嗦说了这么多， 快要和Spring扯上关系了。

上面的代码其实就是实现了一个依赖的注入，把两个冒牌货注入到业务类中(通过set方法)， 这个注入的过程是在一个测试类中通过代码完成的。

既然能把冒牌货注入进去， 那毫无疑问，肯定也能把一个正经的类安插进去， 因为setter 方法接受的是接口，而不是具体类。

```
OrderProcessor op = new OrderProcessor();
op.setOrderService(new OrderServiceImpl());
op.setEmailService(new EmailServiceImpl());
op.process();
```

用这种方式来处理对象之间的依赖， 会强迫你对接口编程， 好处显而易见。

随着系统复杂度的增长， 这样的代码会越来越多， 最后也会变得难于维护。

能不能把各个类之间的依赖关系统一维护呢?

能不能把系统做的更加灵活一点，用声明的方式而不是用代码的方式来描述依赖关系呢?

肯定可以， 在Java 世界里，如果想描述各种逻辑关系， XML是不二之选：

```
<beans>
    <bean id="order-processor" class="com.coderising.OrderProcessor">
        <property name="orderService" ref="order-service">
        <property name="emailService" ref="email-service">
    </bean>
    <bean id="order-service" class="com.coderising.OrderServiceImpl">
    </bean>
    <bean id="email-service" class="com.coderising.EmailServiceImpl">
    </bean>
</beans>
```

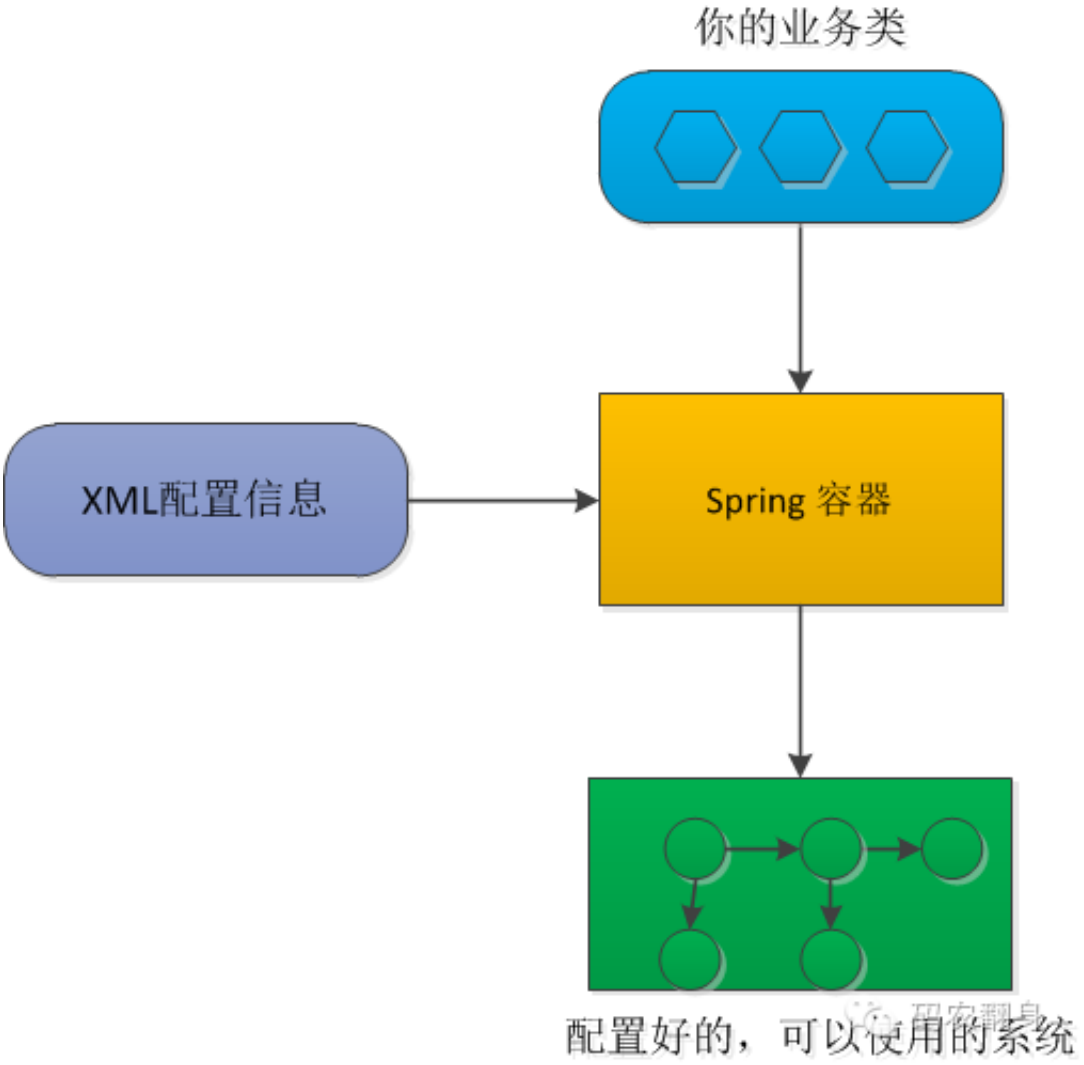
这个xml 挺容易理解的， 但是仅仅有它还不够， 还缺一个解析器（假设叫做XmlAppContext）来解析，处理这个文件，基本过程是：

0. 解析xml, 获取各种元素
1. 通过**Java反射**把各个bean 的实例创建起来： com.coderising.OrderProcessor , OrderServiceImpl, EmailServiceImpl.
2. 还是通过**Java反射**调用OrderProcessor的两个方法：setOrderService(....) 和 setEmailService(...) 把orderService , emailService 实例 注入进去。

应用程序使用起来就简单了：

```
XmlAppContext ctx = new XmlAppContext("c:\bean.xml");
OrderProcessor op = (OrderProcessor) ctx.getBean("order-processor");
op.process();
```

其实Spring的处理方式和上面说的非常类似， 当然Spring 处理了更多的细节，例如不仅仅是setter方法注入， 还可以构造函数注入，init 方法， destroy方法等等， 基本思想是一致的。



既然对象的创建过程和装配过程都是Spring做的， 那Spring 在这个过程中就可以玩很多把戏了， 比如对你的业务类做点字节码级别的增强， 搞点AOP什么的， 这都不在话下了。

4 IoC vs DI

“不要给我们打电话，我们会打给你的(don‘t call us, we‘ll call you)”这是著名的好莱坞原则。

在好莱坞，把简历递交给演艺公司后就只有回家等待。由演艺公司对整个娱乐项目完全控制，演员只能被动式的接受公司的差使,在需要的环节中，完成自己的演出。

这和软件开发有一定的相似性， 演员们就像一个个Java Object, 最早的时候自己去创建自己所依赖的对象， 有了演艺公司（Spring容器）的介入，所有的依赖关系都是演艺公司搞定的， 于是控制就翻转了

Inversion of Control, 简称IoC。

但是IoC这个词不能让人更加直观和清晰的理解背后所代表的含义， 于是Martin Flower先生就创造了一个新词 : 依赖注入 (Dependency Injection，简称DI), 是不是更加贴切一点?