

30 | 热点问题答疑（3）：Spring框架中的设计模式

2019-07-18 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 07:32 大小 6.91M




在构思这个专栏的时候，回想当时我是如何研究 Tomcat 和 Jetty 源码的，除了理解它们的实现之外，也从中学到了很多架构和设计的理念，其中很重要的就是对设计模式的运用，让我收获到不少经验。而且这些经验通过自己消化和吸收，是可以把它应用到实际工作中去的。

在专栏的热点问题答疑第三篇，我想跟你分享一些我对设计模式的理解。有关 Tomcat 和 Jetty 所运用的设计模式我在专栏里已经有所介绍，今天想跟你分享一下 Spring 框架里的设计模式。Spring 的核心功能是 IOC 容器以及 AOP 面向切面编程，同样也是很多 Web 后端工程师每天都要打交道的框架，相信你一定可以从中吸收到一些设计方面的精髓，帮助你提升设计能力。

简单工厂模式

我们来考虑这样一个场景：当 A 对象需要调用 B 对象的方法时，我们需要在 A 中 new 一个 B 的实例，我们把这种方式叫作硬编码耦合，它的缺点是一旦需求发生变化，比如需要使用 C 类来代替 B 时，就要改写 A 类的方法。假如应用中有 1000 个类以硬编码的方式耦合了 B，那改起来就费劲了。于是简单工厂模式就登场了，简单工厂模式又叫静态工厂方法，其实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

Spring 中的 BeanFactory 就是简单工厂模式的体现，BeanFactory 是 Spring IOC 容器中的一个核心接口，它的定义如下：

 复制代码

```
1 public interface BeanFactory {
2     Object getBean(String name) throws BeansException;
3     <T> T getBean(String name, Class<T> requiredType);
4     Object getBean(String name, Object... args);
5     <T> T getBean(Class<T> requiredType);
6     <T> T getBean(Class<T> requiredType, Object... args);
7     boolean containsBean(String name);
8     boolean isSingleton(String name);
9     boolean isPrototype(String name);
10    boolean isTypeMatch(String name, ResolvableType typeToMatch);
11    boolean isTypeMatch(String name, Class<?> typeToMatch);
12    Class<?> getType(String name);
13    String[] getAliases(String name);
14 }
```

我们可以通过它的具体实现类（比如 ClassPathXmlApplicationContext）来获取 Bean：

 复制代码


```
1 BeanFactory bf = new ClassPathXmlApplicationContext("spring.xml");
2 User userBean = (User) bf.getBean("userBean");
```

从上面代码可以看到，使用者不需要自己来 new 对象，而是通过工厂类的方法 getBean 来获取对象实例，这是典型的简单工厂模式，只不过 Spring 是用反射机制来创建 Bean 的。

工厂方法模式


工厂方法模式说白了其实就是简单工厂模式的一种升级或者说是进一步抽象，它可以应用于更加复杂的场景，灵活性也更高。在简单工厂中，由工厂类进行所有的逻辑判断、实例创建；如果不想在工厂类中进行判断，可以为不同的产品提供不同的工厂，不同的工厂生产不同的产品，每一个工厂都只对应一个相应的对象，这就是工厂方法模式。

Spring 中的 FactoryBean 就是这种思想的体现，FactoryBean 可以理解为工厂 Bean，先来看看它的定义：

 复制代码

```
1 public interface FactoryBean<T> {
2     T getObject();
3     Class<?> getObjectType();
4     boolean isSingleton();
5 }
```


我们定义一个类 UserFactoryBean 来实现 FactoryBean 接口，主要是在 getObject 方法里 new 一个 User 对象。这样我们通过 getBean(id) 获得的是该工厂所产生的 User 的实例，而不是 UserFactoryBean 本身的实例，像下面这样：

 复制代码

```
1 BeanFactory bf = new ClassPathXmlApplicationContext("user.xml");
2 User userBean = (User) bf.getBean("userFactoryBean");
```

单例模式

单例模式是指一个类在整个系统运行过程中，只允许产生一个实例。在 Spring 中，Bean 可以被定义为两种模式：Prototype（多例）和 Singleton（单例），Spring Bean 默认是单例模式。那 Spring 是如何实现单例模式的呢？答案是通过单例注册表的方式，具体来说就是使用了 HashMap。请注意为了方便你阅读，我对代码进行了简化：

 复制代码

```
1 public class DefaultSingletonBeanRegistry {
2
3     // 使用了线程安全容器 ConcurrentHashMap，保存各种单实例对象
4     private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, (
5
```

```
6     protected Object getSingleton(String beanName) {
7         // 先到 HashMap 中拿 Object
8         Object singletonObject = singletonObjects.get(beanName);
9
10        // 如果没拿到通过反射创建一个对象实例，并添加到 HashMap 中
11        if (singletonObject == null) {
12            singletonObjects.put(beanName,
13                                Class.forName(beanName).newInstance());
14        }
15
16        // 返回对象实例
17        return singletonObjects.get(beanName);
18    }
19 }
```

上面的代码逻辑比较清晰，先到 HashMap 去拿单实例对象，没拿到就创建一个添加到 HashMap。

代理模式

所谓代理，是指它与被代理对象实现了相同的接口，客户端必须通过代理才能与被代理的目标类进行交互，而代理一般在交互的过程中（交互前后），进行某些特定的处理，比如在调用这个方法前做前置处理，调用这个方法后做后置处理。代理模式中有下面几种角色：

抽象接口：定义目标类及代理类的共同接口，这样在任何可以使用目标对象的地方都可以使用代理对象。

目标对象：定义了代理对象所代表的目标对象，专注于业务功能的实现。

代理对象：代理对象内部含有目标对象的引用，收到客户端的调用请求时，代理对象通常不会直接调用目标对象的方法，而是在调用之前和之后实现一些额外的逻辑。

代理模式的好处是，可以在目标对象业务功能的基础上添加一些公共的逻辑，比如我们想给目标对象加入日志、权限管理和事务控制等功能，我们就可以使用代理类来完成，而没必要修改目标类，从而使得目标类保持稳定。这其实是开闭原则的体现，不要随意去修改别人已经写好的代码或者方法。


代理又分为静态代理和动态代理两种方式。静态代理需要定义接口，被代理对象（目标对象）与代理对象（Proxy）一起实现相同的接口，我们通过一个例子来理解一下：

```
1 // 抽象接口
2 public interface IStudentDao {
3     void save();
4 }
5
6 // 目标对象
7 public class StudentDao implements IStudentDao {
8     public void save() {
9         System.out.println(" 保存成功 ");
10    }
11 }
12
13 // 代理对象
14 public class StudentDaoProxy implements IStudentDao{
15     // 持有目标对象的引用
16     private IStudentDao target;
17     public StudentDaoProxy(IStudentDao target){
18         this.target = target;
19     }
20
21     // 在目标功能对象方法的前后加入事务控制
22     public void save() {
23         System.out.println(" 开始事务 ");
24         target.save();// 执行目标对象的方法
25         System.out.println(" 提交事务 ");
26     }
27 }
28
29 public static void main(String[] args) {
30     // 创建目标对象
31     StudentDao target = new StudentDao();
32
33     // 创建代理对象，把目标对象传给代理对象，建立代理关系
34     StudentDaoProxy proxy = new StudentDaoProxy(target);
35
36     // 执行的是代理的方法
37     proxy.save();
38 }
```

而 Spring 的 AOP 采用的是动态代理的方式，而动态代理就是指代理类在程序运行时由 JVM 动态创建。在上面静态代理的例子中，代理类（StudentDaoProxy）是我们自己定义好的，在程序运行之前就已经编译完成。而动态代理，代理类并不是在 Java 代码中定义的，而是在运行时根据我们在 Java 代码中的“指示”动态生成的。那我们怎么“指示”JDK 去动态地生成代理类呢？


在 Java 的 `java.lang.reflect` 包里提供了一个 `Proxy` 类和一个 `InvocationHandler` 接口，通过这个类和这个接口可以生成动态代理对象。具体来说有如下步骤：

1. 定义一个 `InvocationHandler` 类，将需要扩展的逻辑集中放到这个类中，比如下面的例子模拟了添加事务控制的逻辑。

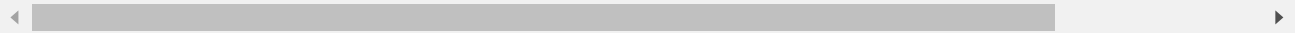
 复制代码

```
1 public class MyInvocationHandler implements InvocationHandler {
2
3     private Object obj;
4
5     public MyInvocationHandler(Object obj){
6         this.obj=obj;
7     }
8
9     @Override
10    public Object invoke(Object proxy, Method method, Object[] args)
11        throws Throwable {
12
13        System.out.println(" 开始事务 ");
14        Object result = method.invoke(obj, args);
15        System.out.println(" 开始事务 ");
16
17        return result;
18    }
19 }
```

2. 使用 `Proxy` 的 `newProxyInstance` 方法动态的创建代理对象：

 复制代码

```
1 public static void main(String[] args) {
2     // 创建目标对象 StudentDao
3     IStudentDao stuDAO = new StudentDao();
4
5     // 创建 MyInvocationHandler 对象
6     InvocationHandler handler = new MyInvocationHandler(stuDAO);
7
8     // 使用 Proxy.newProxyInstance 动态的创建代理对象 stuProxy
9     IStudentDao stuProxy = (IStudentDao)
10    Proxy.newProxyInstance(stuDAO.getClass().getClassLoader(), stuDAO.getClass().getInterf
11
12    // 动用代理对象的方法
13    stuProxy.save();
14 }
```



上面的代码实现和静态代理一样的功能，相比于静态代理，动态代理的优势在于可以很方便地对代理类的函数进行统一的处理，而不用修改每个代理类中的方法。

Spring 实现了通过动态代理对类进行方法级别的切面增强，我来解释一下这句话，其实就是动态生成目标对象的代理类，并在代理类的方法中设置拦截器，通过执行拦截器中的逻辑增强了代理方法的功能，从而实现 AOP。

本期精华

今天我和你聊了 Spring 中的设计模式，我记得我刚毕业那会儿，拿到一个任务时我首先考虑的是怎么把功能实现了，从不考虑设计的问题，因此写出来的代码就显得比较稚嫩。后来随着经验的积累，我会有意识地去思考，这个场景是不是用个设计模式会更高大上呢？以后重构起来是不是会更轻松呢？慢慢我也就形成一个习惯，那就是用优雅的方式去实现一个系统，这也是每个程序员需要经历的过程。

今天我们学习了 Spring 的两大核心功能 IOC 和 AOP 中用到的一些设计模式，主要有简单工厂模式、工厂方法模式、单例模式和代理模式。而代理模式又分为静态代理和动态代理。JDK 提供实现动态代理的机制，除此之外，还可以通过 CGLIB 来实现，有兴趣的同学可以理解一下它的原理。

课后思考

注意到在 newProxyInstance 方法中，传入了目标类的加载器、目标类实现的接口以及 MyInvocationHandler 三个参数，就能得到一个动态代理对象，请你思考一下 newProxyInstance 方法是如何实现的。

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它分享给你的朋友。

深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 29 | 比较：Jetty如何实现具有上下文信息的责任链？

下一篇 31 | Logger组件：Tomcat的日志框架及实战

精选留言 (8)

写留言



-W.LI-

2019-07-18

之前硬着头皮看过这个源码，中间有一步weakcache印象深刻。弱引用缓存，先从缓存中取，没取到获取字节码，校验魔术版本号之类的，最后反射实现。反射效率不高也是这个原因导致的，要获取源文件校验准备初始化(轻量级累加载一次)。

展开



3



QQ怪

2019-07-18

动态代理的思路便是动态生成一个新类，先通过传入的classLoader生成对应Class对象，后通过反射获取构造函数对象并生成代理类实例，jdk动态代理是通过接口实现的，但很多

类是没有针对接口设计的，但是我们知道可以通过拼接字节码生成新的类的自由度是十分大的，所以cglib框架就是通过拼接字节码来实现非接口类的代理。

展开 ▾



👍 1



QQ怪

2019-07-18

老师这次加餐面试必问题

展开 ▾



👍 1



nightmare

2019-07-19

老师能讲一下spring这么解决循环依赖的吗

展开 ▾

作者回复: Spring只解决了单例bean通过setXxx或者@Autowired进行循环依赖：

<https://blog.csdn.net/qq924862077/article/details/73926268>

其他场景可以想办法绕过：<https://www.baeldung.com/circular-dependencies-in-spring>



nightmare

2019-07-19

如果是基于反射实现的，那增强的业务这么植入的

展开 ▾



z.l

2019-07-18

老师，工厂方法模式没看懂。另外DefaultSingletonBeanRegistry的getBean方法的实现存在线程安全问题吧？虽然用了ConcurrentHashMap,但是if (singletonObject == null)存在竞态条件，可能有2个线程同时判断为true，最后产生了2个对象实例。应该用putIfAbsent方法。

展开 ▾

作者回复: 汗颜，代码简化的过程中去掉了线程安全的部分，线程安全的版本应该是这样的：

```
//如果没拿到通过反射创建一个对象实例，并添加到HashMap中
if (singletonObject == null) {
    synchronized(singletonObjects){
        if(singletonObjects.get(beanName) == null){
            singletonObjects.put(beanName,Class.forName(beanName).newInstance());
        }
    }
}
```



许童童

2019-07-18

工厂方法模式说的就是抽象工厂模式吧

展开 ▾



阿青，你学到了吗

2019-07-18

基于反射实现的

展开 ▾

