1. Spring架构和设计理念

Spring可以把对象之间的依赖关系转而用配置文件来管理，也就是依赖注入机制，而这个注入关系由IOC容器管理。

Spring框架中核心组件：Core，Context，Bean；Spring可以说是面向bean编程，Context是舞台，Core是道具，bean就是演员。

Bean包装的是Object；

Context就是要发现每个Bean之间的关系，为他们建立这种关系并且维护关系，所以Context就是一个Bean关系的集合，这个集合又叫IOC容器；

Core就是发现，建立和维护每个Bean之间关系所需要的一些列工具，类似Util

Core组件

Context组件

Bean组件

Bean组件

Bean组件

Bean组件

Bean组件

Bean组件

1. 核心组件详解

2.1 Bean组件

Bean组件位于org.springframework.beans包，主要负责Bean的定义，创建和解析（定义和解析都是Spring内部完成，使用时不用管），Bean的创建用的是典型的工厂模式，它的顶级接口是BeanFactory。

BeanFactory由三个子类ListableBeanFactory（表示这些Bean是可列表的），HierarchicalBeanfactory（表示这些Bean是由继承关系的），AutowireCapableBeanFactory（这个接口定义的Bean的自动装配规则）

Bean的 定义主要由主要由BeanDefinition描述，Bean定义完整描述了Spring配置文件中<bean>节点中的所有信息，Spring解析<bean>节点后就把内容转化成BeanDefinition对象，之后所有操作都是对这个对象进行的

Bean的解析主要对Spring配置文件解析，有XmlBeanDefinitionReader等类

2.2 Context组件

Context在Spring的org.springframework.context包，它实际上是给Spring提供一个运行时的环境，用于保存各个对象的状态。

ApplicationContext是Context的顶级父类，它除了能标识一个应用环境的基本信息外，还继承了其他接口，这些接口主要是扩展Context的功能。ApplicationContext继承了BeanFactory（ListableBeanFactory和HierarchicalBeanfactory），说明了Spring容器中运行的主体对象是Bean，ApplicationContext还继承了ResourceLoader接口，这样ApplicationContext就可以访问外部资源，ApplicationContext还继承了ApplicationEventPublisher，MessageSource。

ApplicationContext的子类主要包含两方面：ConfigurableApplicationContext和WebApplicationContext（可直接访问Servlet，通常很少用），其中ConfigurableApplicationContext表示该Context是可修改的，用户可以在构建Context时，动态添加或修改原有配置信息，它下面有多个子类，最常用的是可更新的Context，即AbstractRefreshableApplicationContext类。

总体来说ApplicationContext必须要完成：标识一个应用环境，通过BeanFactory创建Bean对象，保存对象间关系，捕获各种事件。Context是Spring的IOC容器，整合了Spring大部分基础功能。

2.3 Core组件

Core的一个重要组成部分是把所有资源都抽象成了一个接口Resource，这个接口封装了资源类型，屏蔽了文件类型的不同，Resource接口继承了InputStreamSource接口，这个接口中有getInputStream方法返回InputStream类，所有资源都可以通过InputStream类来获取，所以屏蔽了资源提供者。资源加载通过ResourceLoader完成，屏蔽了所有资源加载者的差异，默认实现类DefaultResourceLoader。

Context把资源的加载，解析和描述工作委托给了ResourcePatternResolver类，它是一个接头人，把资源的加载解析和资源的定义整合在一起方便其他组件使用。

2.4 IOC容器工作机制

1. 创建BeanFactory

IOC容器实际是Context组件结合其他两个组件共同构建了一个Bean关系网，构建的入口就在AbstractApplicationContext类的refresh方法中

public void refesh() throw BeansException, IllegalStateException {

synchronized (this.startupShutdownMonitor) {

prepareRefresh();//为刷新准备新的context

//刷新所有BeanFactory子容器

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

//创建BeanFactory后添加一些Spring本身需要的工具类

prepareBeanFactory(beanFactory);

try {

//注册实现了BeanPostProcessor接口的bean

postProcessBeanFactory(beanFactory);

//初始化和执行BeanFactoryPostProcessor beans

invokeBeanFactoryPostProcessors(beanFactory);

//初始化和执行BeanPostProcessor beans

registerBeanPostProcessors(beanFactory);

//初始化MessageSource

initMessageSource();

//初始化event multicaster

initApplicationEventMulticaster();

//刷新由子类实现的方法

onRefresh();

//检查注册时间

registerListeners();

finishBeanFactoryInitialization(beanFactory);

//执行LifecycleProcessor.onRefresh()和ContextFefreshedEvent事件

finishRefresh();

}

catch (BeansException ex) {

destroyBeans();

cancelRefresh(ex);

throw ex;

}

}

}

这段代码是构建IOC容器的步骤：

1. 构建BeanFactory，便于产生所需的Bean

2. 注册可能感兴趣的事件

3. 创建Bean实例对象

4. 触发被监听的事件

首先是创建和配置BeanFactory，这里是refresh，即刷新配置，如果BeanFactory存在就更新，如果不存在就新建

//更新BeanFactory的代码，实现ConfigurableApplicationContext类里的方法

protected final void refreshBeanFactory() throws BeanException {

if (hasBeanFactory()) {

destoryBeans();

closeBeanFactory();

}

try {

DefaultListableBeanFactory beanFactory = createBeanFactory();

beanFactory.setSerialization(getId());

customizeBeanFactory(beanFactory);

//把用户定义的数据结构转换为IOC容器中的特定数据结构

loadBeanDefinitions(beanFactory);

synchronized (this.beanFactoryMonitor) {

this,beanFactory = beanFactory;

}

}

catch (IOException ex) {

throw XXX

}

}

//注册实现了BeanPostProcessor接口的bean

postProcessBeanFactory(beanFactory);

//初始化和执行BeanFactoryPostProcessor beans

invokeBeanFactoryPostProcessors(beanFactory);

//初始化和执行BeanPostProcessor beans

registerBeanPostProcessors(beanFactory);

这段对Spring功能扩展起了至关重要的作用，invokeBeanFactoryPostProcessors主要是获取实现BeanFactoryPostProcessor接口的子类，并执行它的postProcessBeanFactory方法

registerBeanPostProcessors方法可以获取用户定义的实现了BeanPostProcessor接口的子类，并把它们注册到BeanFactory对象中的beanPostProcessors变量中

在BeanPostProcessor中声明了两个方法：postProcessBeforeInitialization和postProcessAfterInitialization，分别用于在Bean对象初始化时执行用户自定义操作

后面几行是注册监听，message等

创建Bean实例和Bean关系网，从finishBeanFactoryInitialization方法开始，Bean的实例化是在BeanFactory中发生的，其中有一个非常重要的Bean，FactoryBean，Spring的很多扩展跟这个Bean有关

它是个工厂Bean，可以产生Bean实例的Bean，一个类继承FactoryBean用户可以自己定义产生实例对象的方法，只需要实现其getObject方法

Spring必须要先构建IOC容器，ApplicationContext.xml是IOC容器的默认配置文件。

1. AOP

AOP是基于动态代理实现的，实现基础是JDK的java.lang.reflect包有个Proxy类，它是构造代理类的入口，这个类通过newProxyInstance方法建立代理对象，这个方法需要三个参数：

ClassLoader，用于加载代理类的Loader类

Interfaces：被代理的接口

InvocationHandler：用于执行除了被代理接口方法之外的用户自定义操作，用户调用目标方法被代理到Invocationhandler类中唯一方法invoke方法中

AOP的代理类继承了FactoryBean的ProxyFactoryBean，FactoryBean的特别就是它允许自定义创建对象的方法。

AOP创建代理对象基本步骤：调用getObject对象，初始化AdvisorChain，将interceptor对象包装成Advisor对象，把这些类放到拦截器链中，调用getProxy产生代理对象。

Spring调用拦截器步骤：执行AOP代理对象invoke方法，获取目标方法上的拦截器，执行拦截的invoke方法，执行前置方法，下一个拦截器执行invoke方法，执行前置方法，执行目标方法，执行后置方法

Spring还支持CGLIB代理

1. 代理模式

静态代理：

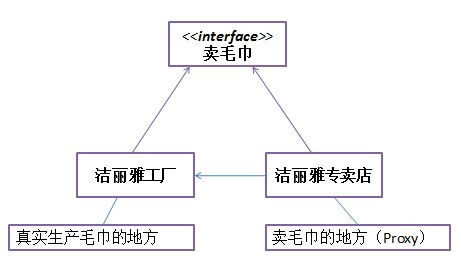
**1.初识代理模式**

　　生活中大家一定遇到这样的情况了：比如说我要买一条毛巾，不妨就是洁丽雅的吧，那一般人的话我应该不是去洁丽雅的工厂里面直接去买吧，而是我们在洁丽雅专卖店或者什么超市啊，这些地方购买，这些地方实际上就是洁丽雅毛巾的代理。这其实和我们OO中的代理模式是很相似的。

　　一个it人士，上网应该是经常的事了吧，那么总会有这样的情况，打开一个网页，文字先出现，而那些比较大的资源，例如图片要等等才会显示出来，这是为什么呢？？实际上这其中采用的就是代理模式。

**2.代理模式的一个简单的例子**

就说上面的毛巾的例子吧，将有如下类图：



消费者虽然买的是洁丽雅工厂生产的毛巾，确实通过洁丽雅专卖店买到的毛巾，并不是直接去厂家拿货，这就是代理模式。看看代码吧：

//定义了接口

interface SaleTowelIntertace

{

void sellTowel();

}

//毛巾的生产类

class TowelProduce : SaleTowelIntertace

{

string ConsumerName;

TowelProduce(string consumer\_name)

{

ConsumerName = consumer\_name;

}

void sellTowel()

{

Console.WriteLine("毛巾卖给了{0}",ConsumerName);

}

}

//卖毛巾的代理类

class TowelSaleProxy : SaleTowelIntertace

{

TowelProduce tp;

TowelSaleProxy(string consumner\_name)

{

tp = new TowelProduce(consumer\_name);

}

void sellTowel()

{

tp.sellTowel();

}

}

看了这段代码，可能会想，那么这个代理有什么用呢？似乎看不到具体有什么好处啊！！其实这样的主要好处是真正购买毛巾的客户和厂家并不存在联系，即减小了他们之间的耦合性。这样说似乎还是有点抽象了，而这又是一个关键的问题。所以……

**3.代理模式的意义**

　　代理模式到底好处在哪里呢？？

　　那先要说一下代理模式中的三种角色了。

**抽象角色：**声明真实对象和代理对象的共同接口。  
　　**代理角色：**代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象　　　　　　　　可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。  
　　**真实角色：**代理角色所代表的真实对象，是我们最终要引用的对象。

　　代理模式的一个好处就是对外部提供统一的接口方法，而代理类在接口中实现对真实类的附加操作行为，从而可以在不影响外部调用情况下，进行系统扩展。也就是说，我要修改真实角色的操作的时候，尽量不要修改他，而是在外部在“包”一层进行附加行为，即代理类。例如：接口A有一个接口方法operator()，真实角色：RealA实现接口A，则必须实现接口方法operator()。客户端Client调用接口A的接方法operator()。现在新需求来了，需要修改RealA中的operator()的操作行为。怎么办呢？如果修改RealA就会影响原有系统的稳定性，还要重新测试。这是就需要代理类实现附加行为操作。创建代理ProxyA实现接口A，并将真实对象RealA注入进来。ProxyA实现接口方法operator()，另外还可以增加附加行为，然后调用真实对象的operator()。从而达到了“对修改关闭，对扩展开放”，保证了系统的稳定性。我们看客户端Client调用仍是接口A的接口方法operator()，只不过实例变为了ProxyA类了而已。也就是说代理模式实现了ocp原则。

**4.什么时候使用代理模式**

　　当我们需要使用的对象很复杂或者需要很长时间去构造，这时就可以使用代理模式(Proxy)。例如：如果构建一个对象很耗费时间和计算机资源，代理模式(Proxy)允许我们控制这种情况，直到我们需要使用实际的对象。一个代理(Proxy)通常包含和将要使用的对象同样的方法，一旦开始使用这个对象，这些方法将通过代理(Proxy)传递给实际的对象。 一些可以使用代理模式(Proxy)的情况：

　　一个对象，比如一幅很大的图像，需要载入的时间很长。

　　一个需要很长时间才可以完成的计算结果，并且需要在它计算过程中显示中间结果

　　一个存在于远程计算机上的对象，需要通过网络载入这个远程对象则需要很长时间，特别是在网络传输高峰期。

　　一个对象只有有限的访问权限，代理模式(Proxy)可以验证用户的权限

　　代理模式(Proxy)也可以被用来区别一个对象实例的请求和实际的访问，例如：在程序初始化过程中可能建立多个对象，但并不都是马上使用，代理模式(Proxy)可以载入需要的真正的对象。这是一个需要载入和显示一幅很大的图像的程序，当程序启动时，就必须确定要显示的图像，但是实际的图像只能在完全载入后才可以显示！这时我们就可以使用代理模式(Proxy)。

动态代理（AOP用的）

动态代理（Dynamic Proxy）：相比前一篇文章所实现的静态代理，动态代理具有更强的灵活性，因为它不用在我们设计实现的时候就指定某一个代理类来代理哪一个被代理对象，我们可以把这种指定延迟到程序运行时由JVM来实现。

我们知道，所谓代理，就是需要代理类和被代理类有相同的对外接口或者说成服务，所以代理类一般都必须实现了所有被代理类已实现的接口，因为接口就是制定了一系列对外服务的标准。

正因为动态代理有这样灵活的特性，所以我们在设计动态代理类（DynamicProxy）时不用显式地让它实现与真实主题类（RealSubject）相同的接口（interface），而是把这种实现推迟到运行时。

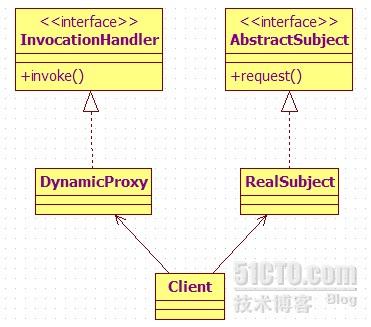
为了能让DynamicProxy类能够在运行时才去实现RealSubject类已实现的一系列接口并执行接口中相关的方法操作，需要让DynamicProxy类实现JDK自带的java.lang.reflect.InvocationHandler接口，该接口中的invoke()方法能够让DynamicProxy实例在运行时调用被代理类的“对外服务”，即调用被代理类需要对外实现的所有接口中的方法，也就是完成对真实方法的调用，Java帮助文档中称这些真实方法为处理程序。

按照上面所述，我们肯定必须先把被代理类RealSubject已实现的所有interface都加载到JVM中，不然JVM怎么能够找到这些方法呢？明白了这个道理，那么我们就可以创建一个被代理类的实例，获得该实例的类加载器ClassLoader。

所谓的类加载器ClassLoader，就是具有某个类的类定义，即类的内部相关结构（包括继承树、方法区等等）。

更重要的是，动态代理模式可以使得我们在**不改变原来已有的代码结构**的情况下，对原来的“真实方法”进行**扩展、增强其功能**，并且可以**达到控制被代理对象的行为的目的**。请详看下面代码中的DynamicProxy类，其中必须实现的invoke()方法在调用被代理类的真实方法的前后都可进行一定的特殊操作。这是动态代理最明显的优点。

虽然都是根据自己看了书之后的理解说了这么多，不知道能不能让人明白，这里先给出动态代理的类图吧，如下：

[](http://img1.51cto.com/attachment/201007/175427339.jpg)

具体代码实现如下：

1. **import** java.lang.reflect.InvocationHandler;
2. **import** java.lang.reflect.Method;
3. **import** java.lang.reflect.Proxy;
5. //抽象主题类,这里不能用abstract抽象类，一定要是interface
6. **interface** AbstractSubject {
7. **public** **abstract** **void** request();
8. }
10. // 真实主题类，即被代理类
11. **class** RealSubject **implements** AbstractSubject {
12. **public** **void** request() {
13. System.out.println("RealSubject's request() ...");
14. }
15. }
17. // 动态代理类，实现InvocationHandler接口
18. **class** DynamicProxy **implements** InvocationHandler {
20. // 被代理类的实例
21. Object obj = **null**;
23. // 将被代理者的实例传进动态代理类的构造函数中
24. **public** DynamicProxy(Object obj) {
25. **this**.obj = obj;
26. }
28. /\*\*
29. \* 覆盖InvocationHandler接口中的invoke()方法
30. \*
31. \* 更重要的是，动态代理模式可以使得我们在不改变原来已有的代码结构
32. \* 的情况下，对原来的“真实方法”进行扩展、增强其功能，并且可以达到
33. \* 控制被代理对象的行为，下面的before、after就是我们可以进行特殊
34. \* 代码切入的扩展点了。
35. \*/
36. **public** Object invoke(Object proxy, Method method, Object[] args)
37. **throws** Throwable {
38. /\*
39. \* before ：doSomething();
40. \*/
41. Object result = method.invoke(**this**.obj, args);
43. /\*
44. \* after : doSomething();
45. \*/
46. **return** result;
47. }
48. }
50. // 测试类
51. **public** **class** Client {
52. **public** **static** **void** main(String[] args) {
54. // 被代理类的实例
55. AbstractSubject realSubject = **new** RealSubject();
57. // 获得被代理类的类加载器，使得JVM能够加载并找到被代理类的内部结构，以及已实现的interface
58. ClassLoader loader = realSubject.getClass().getClassLoader();
60. // 获得被代理类已实现的所有接口interface,使得动态代理类的实例
61. Class<?>[] interfaces = realSubject.getClass().getInterfaces();
63. // 用被代理类的实例创建动态代理类的实例，用于真正调用处理程序
64. InvocationHandler handler = **new** DynamicProxy(realSubject);
66. /\*
67. \* loader : 被代理类的类加载器
68. \* interfaces ：被代理类已实现的所有接口，而这些是动态代理类要实现的接口列表
69. \* handler ： 用被代理类的实例创建动态代理类的实例，用于真正调用处理程序
70. \*
71. \* return ：返回实现了被代理类所实现的所有接口的Object对象，即动态代理，需要强制转型
72. \*/
73. //获得代理的实例
74. AbstractSubject proxy = (AbstractSubject) Proxy.newProxyInstance(
75. loader, interfaces, handler);
77. proxy.request();
78. //打印出该代理实例的名称
79. System.out.println(proxy.getClass().getName());
80. }
81. }

测试结果：

|  |
| --- |
| RealSubject's request() ...  DesignPattern.proxy.dynamicProxy.$Proxy0 |