# Spring

## 注解的实现机制？

<http://www.cnblogs.com/Johness/archive/2013/04/17/3026689.html>

Spring的配置文件为applicationContext.xml文件，在项目启动的时候加载这个配置文件。

Spring需要在配置文件中开启注解

<!-- 开启自动扫描 -->

<context:annotation-config/>

<context:component-scan base-package="com.itxxz" use-default-filters="false">

<context:include-filter type="annotation" expression="org.springframework.stereotype.Service" />

<context:include-filter type="annotation" expression="org.springframework.stereotype.Repository" />

<context:include-filter type="annotation" expression="org.springframework.stereotype.Component" />

</context:component-scan>

需要制定要扫描的包路径

## 什么是spring

Spring的核心是一个轻量级（Lightweight）的容器（Container），它是实现IoC（Inversion of Control）容器和非入侵性（No intrusive）的框架，并提供AOP（Aspect-oriented programming）概念的实现方式；提供对持久层（Persistence）、事物（Transcation）的支持；提供MVC Web框架的实现，并对一些常用的企业服务API（Application Interface）提供一致的模型封装，是一个全方位的应用程序框架（Application Framework），除此之外，对现存的各种框架（Structs、JSF、Hibernate、Ibatis、Webwork等），Spring也提供了与他们相整合的方案。

**轻量级**：sring框架在系统初始化时不用加载所有的服务，只需要加载我们使用的核心服务，这就是轻量级。

**重量级**：对于EJB来说，在系统初始化时需要加载所有的服务。

**非入侵**：允许应用系统自由选择和组装spring框架的各个功能模块，并不强制要求应用系统的类必须继承框架提供的类或者实现接口。

**入侵**：强制必须去继承框架提供的类，这样以后如果想更换框架，那么之前写的代码几乎无法重用。（strust1是入侵的）

**如何实现非入侵？**

1 应用反射机制，通过动态调用的方式来提供各方面的功能，建立核心组件BeanFactory

2 配合使用Spring框架中的BeanWrapper和BeanFactory组件类最终达到对象的实例创建和属性注入

**优点**：允许所开发出来的应用系统能够在不用的环境中自由移植，不需要修改应用系统中的核心功能实现的代码

Spring在ioc容器中是非入侵的，但是在aop 事务管理还是有入侵的，比如需要使用spring的事务管理类，而且spring对jdbc的异常做了封装。

## Spring框架中的主要模块



## 使用spring框架带来的好处

Spring是一个轻量级框架，可以自己选择里面的组件模块。

Spring通过控制反转（IOC）实现了松散耦合，对各个层之间进行了解耦。

Spring支持了面向编程（AOP），将业务逻辑和系统服务分开。

Spring中的容器存储并管理了应用中对象的生命周期和配置。

Spring提供的web框架springmvc。

Spring提供了一个持续的事物管理接口，可以扩展到上至本地事务下至全局事务。

Spring提供了单元测试功能，可以很方便的使用依赖注入来写入测试数据。

## 什么是控制反转和依赖注入

控制反转：以前的做法是由程序代码去创建需要使用的对象，现在由容器进行创建和管理对象。

依赖注入：对象的属性关系由spring管理。

依赖注入是控制反转的基础。

依赖注入和控制反转是对同一件事情的不同描述，从某个方面讲，就是它们描述的角度不同。依赖注入是从应用程序的角度在描述，可以把依赖注入描述完整点：应用程序依赖容器创建并注入它所需要的外部资源；而控制反转是从容器的角度在描述，描述完整点：容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。

依赖注入的四种实现方式：

1 通过setter方法注入

2 构造器注入

3 接口注入：要求组件必须与特定的接口相关联，具有入侵性。

## IOC

Spring中的 org.springframework.beans 包和 org.springframework.context包构成了Spring框架IoC容器的基础。

BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。ApplicationContex接口对BeanFactory（是一个子接口）进行了扩展，在BeanFactory的基础上添加了其他功能，比如与Spring的AOP更容易集成，也提供了处理message resource的机制（用于国际化）、事件传播以及应用层的特别配置，比如针对Web应用的WebApplicationContext。

**org.springframework.beans.factory.BeanFactory** 是Spring IoC容器的具体实现，用来包装和管理前面提到的各种bean。BeanFactory接口是Spring IoC 容器的核心接口。

## BeanFactory和ApplicationContext的区别

**BeanFactory：**是Spring里面最低层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能；

**ApplicationContext：**应用上下文，继承BeanFactory接口，它是Spring的一种更高级的容器，提供了更多的有用的功能；

1) 提供了支持国际化的文本消息

BeanFactory是不支持国际化功能的，因为BeanFactory没有扩展Spring中MessageResource接口。

2) 强大的事件机制。

基本上牵涉到事件(Event)方面的设计，就离不开观察者模式。  
ApplicationContext的事件机制主要通过ApplicationEvent和ApplicationListener这两个接口来提供的，和java swing中的事件机制一样。即当ApplicationContext中发布一个事件的时，所有扩展了ApplicationListener的Bean都将会接受到这个事件，并进行相应的处理。   
  
Spring提供了部分内置事件，主要有以下几种：    
ContextRefreshedEvent ：ApplicationContext发送该事件时，表示该容器中所有的Bean都已经被装载完成，此ApplicationContext已就绪可用   
ContextStartedEvent：生命周期 beans的启动信号    
ContextStoppedEvent: 生命周期 beans的停止信号    
ContextClosedEvent：ApplicationContext关闭事件，则context不能刷新和重启，从而所有的singleton bean全部销毁(因为singleton bean是存在容器缓存中的)   
  
  虽然，spring提供了许多内置事件，但用户也可根据自己需要来扩展spriong中的事物。注意，要扩展的事件都要实现ApplicationEvent接口。

3) 对资源的访问，如URL和文件（ResourceLoader）

http://snkcxy.iteye.com/blog/1913796

 ApplicationContext扩展了ResourceLoader(资源加载器)接口，从而可以用来加载多个Resource，而BeanFactory是没有扩展ResourceLoader

**ClassPathXmlApplicationContext**：从classpath的XML配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中取得。

|  |  |
| --- | --- |
|  | ApplicationContext context = new ClassPathXmlApplicationContext(“bean.xml”); |

**FileSystemXmlApplicationContext** ：由文件系统中的XML配置文件读取上下文。

|  |  |
| --- | --- |
|  | ApplicationContext context = new FileSystemXmlApplicationContext(“bean.xml”); |

**XmlWebApplicationContext**：由Web应用的XML文件读取上下文。

3) 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

4) 消息发送、响应机制（ApplicationEventPublisher）

5) AOP（拦截器）

一般拦截器都是实现HandlerInterceptor接口，其中有三个方法**preHandle、postHandle、afterCompletion**

1. preHandle：执行controller之前执行

2. postHandle：执行完controller，return modelAndView之前执行，主要操作modelAndView的值

3. afterCompletion：controller返回后执行

装载bean的区别：

BeanFactory在启动的时候不会去实例化Bean，只有从容器中拿Bean的时候才会去实例化；

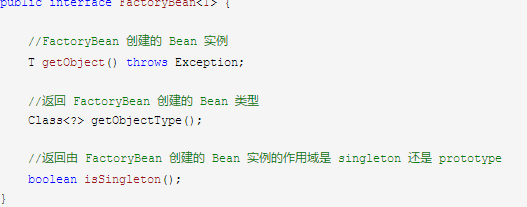
ApplicationContext在启动的时候就把所有的Bean全部实例化了。它还可以为Bean配置lazy-init=true来让Bean延迟实例化；好处是节省了在使用时才去对bean实例化的那部分时间，而且可以在启动时发现spring配置有没有错误。

## BeanFactory和FactoryBean

BeanFactory定义了Spring IOC容器最基本的形式，并提供了IOC容器应遵守的最基本的接口，也是IOC所遵守的最底层和最基本的编程规范。

BeanFactory仅仅是个接口，并不是IOC的具体实现，具体实现有XMLBeanFactory DefaultListableBeanFactory ApplicationContext（接口）

FactoryBean是工厂类接口，可以通过实现该接口定制实例化bean。



BeanFactory是个Factory，一个工厂类，是IOC容器或者对象工厂，所有的bean都是由BeanFactory(Bean容器)进行管理。

FactoryBean是一个能生产或者修饰生成对象的工厂bean，可以在IOC容器中被管理。并不是一个普通的bean，当使用factorybean时，该容器并不会返回factorybean本身，而是返回其生成的对象。（根据该bean的id从beanfactory中获取的实际上是factorybean的getObject对象，如果要获取factoryBean对象，可以在id前面加一个&符号来获取）。它的实现与设计模式中的工厂模式或修饰器模式类似。

## 基于XML的配置

Spring的XML配置方式是使用被Spring命名空间的所支持的一系列的XML标签来实现的。Spring有以下主要的命名空间：context、beans、jdbc、tx、aop、mvc和aso。

在web.xml中配置DispatcherServlet

## 基于java的配置

Spring对Java配置的支持是由@Configuration注解和@Bean注解来实现的。由@Bean注解的方法将会实例化、配置和初始化一个新对象，这个对象将由Spring的IoC容器来管理。@Bean声明所起到的作用与<bean/> 元素类似。被@Configuration所注解的类则表示这个类的主要目的是作为bean定义的资源。被@Configuration声明的类可以通过在同一个类的内部调用@bean方法来设置嵌入bean的依赖关系。



在上面的例子中，com.acme包首先会被扫到，然后再容器内查找被@Component 声明的类，找到后将这些类按照Sring bean定义进行注册。

如果你要在你的web应用开发中选用上述的配置的方式的话，需要用AnnotationConfigWebApplicationContext 类来读取配置文件，可以用来配置Spring的Servlet监听器ContrextLoaderListener或者Spring MVC的DispatcherServlet。



## 基于注解的配置



## BeanWrapper

<http://www.cnblogs.com/mesopotamia/p/5002122.html>

在org.springframework.beans包中，还有两个非常重要的类：BeanWrapper接口及它的实现BeanWrapperImpl。BeanWrapper封装了一个Bean的行为，提供了设置和获得属性值的功能。通过BeanWrapper可以获得Bean的属性和描述、查询只读或者可写属性。

1. 从BeanDefinitionRegistry注册表中取出尚未进行属性配置的BeanDefinition，获取Bean属性的配置信息，
2. 使用属性编辑器对这些配置信息进行转换得到Bean属性的值，
3. 最后对Bean通过反射机制设置属性值。

## Spring流程

1. ResourceLoader加载配置信息，
2. 由BeanDefinitionReader读取并解析<bean>标签，并将<bean>标签的属性都转换为BeanDefinition对应的属性，并注册到BeanDefinitionRegistry注册表中。
3. 容器扫描注册表，通过反射机制获取BeanFactoryPostProcessor类型的工厂后处理器，并用这个工厂后处理器对BeanDefinition进行加工。
4. 取出加工过的BeanDefinition，使用InstantiationStrategy实例化Bean。
5. BeanWrapper结合BeanDefinitionRegistry和PropertyEditorRegistry对Bean的属性赋值。

BeanWrapperImpl完成Bean属性的配置工作之后，接下来还需要Bean后处理器（实现了BeanPostProcessor接口的Bean）继续对Bean实例进行加工，直到装配出一个准备就绪的Bean

## ContextLoaderListener

在启动web容器时，会自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener接口，在web.xml配置这个监听器，启动容器时，就会默认执行它实现的方法。

在ContextLoaderListener中关联了ContextLoader这个类，所以整个加载配置过程由ContextLoader来完成。

1. **<listener>**
2. **<listener-class>**
3. org.springframework.web.context.ContextLoaderListener
4. **</listener-class>**
5. **</listener>**
6. **<context-param>**
7. **<param-name>**contextConfigLocation**</param-name>**
8. **<param-value>**classpath:conf/spring/applicationContext.xml**</param-value>**
9. **</context-param>**

实现了接口ServletContextListener,也就是说他必须实现contextDestroyed, contextInitialized这两个方法



Web容器调用contextInitialized方法初始化ContextLoaderListener，

在此方法中，ContextLoaderListener通过调用继承自**ContextLoader的initWebApplicationContext方法实例化Spring Ioc容器**。

在实例化spring ioc容器过程中，最主要的两个方法是createWebApplicationContext和configureAndRefreshWebApplicationContext方法。

createWebApplicationContext方法用于返回XmlWebApplicationContext实例，即web环境下的spring ioc容器。

configureAndRefreshWebApplicationContext用于配置XmlWebApplicationContext，读取web.xml中通过contextConfigLocation标签指定的xml文件，实例化xml文件中配置的bean，并在上一步实例化的容器中进行注册。

完成上述两步之后，springmvc会将XmlWebApplicationContext实例以属性的方式注册到ServletContext中。

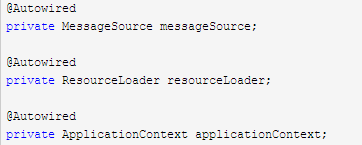
## Spring中常用的接口和类

### ApplicationContextAware接口

     当一个类需要获取ApplicationContext实例时，可以让该类实现ApplicationContextAware接口。



通过@Autowired注解可以自动装配一些常用对象实例：



### ApplicationEvent抽象类

当需要创建自定义事件时，可以新建一个继承自ApplicationEvent抽象类的类。

### ApplicationListener接口

当需要监听自定义事件时，可以新建一个实现ApplicationListener接口的类，并将该类配置到Spring容器中。重写onApplicationEvent(ApplicationEvent event)方法。

### BeanNameAware接口

     当bean需要获取自身在容器中的id/name时，可以实现BeanNameAware接口。

实现BeanNameAware接口，接口中就一个方法setBeanName()

### BeanFactoryAware

让Bean获取配置他们的BeanFactory的引用。

实现BeanFactoryAware接口，其中只有一个方法即setFactory(BeanFactory factory)。使用上与BeanNameAware接口无异，只不过BeanFactoryAware注入的是个工厂，BeanNameAware注入的是个Bean的名字。

### InitializingBean接口

      当需要在bean的全部属性设置成功后做些特殊的处理，可以让该bean实现InitializingBean接口。效果等同于bean的init-method属性的使用或者@PostContsuct注解的使用。三种方式的执行顺序：先注解，然后执行InitializingBean接口中定义的方法，最后执行init-method属性指定的方法。

### DisposableBean接口

当需要在bean销毁之前做些特殊的处理，可以让该bean实现DisposableBean接口。效果等同于bean的destroy-method属性的使用或者@PreDestory注解的使用。三种方式的执行顺序：先注解，然后执行DisposableBean接口中定义的方法，最后执行destroy-method属性指定的方法

### BeanPostProcessor接口

<http://www.iteye.com/topic/1122859>

BeanPostProcessor，是bean的后置处理器，是一个监听器，可以监听容器触发的事件。把它向IOC容器注册以后，使得容器中管理的bean具备接收IOC容器事件回调的能力 。

当需要对受管bean进行预处理时，可以新建一个实现BeanPostProcessor接口的类，并将该类配置到Spring容器中。

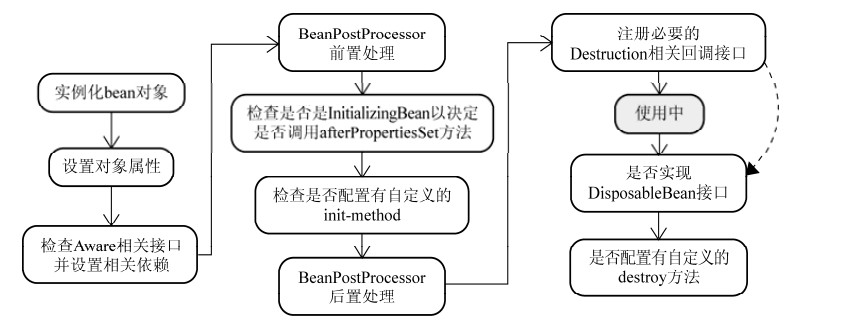
实现BeanPostProcessor接口时，需要实现以下两个方法“

postProcessBeforeInitialization在受管bean的初始化动作之前调用

postProcessAfterInitialization在受管bean的初始化动作之后调用

容器中的每个Bean在创建时都会恰当地调用它们。

如果我们需要在spring容器完成bean的实例化 配置或其他的初始化前后添加一些自己的逻辑处理,就可以定义一个或者多个BeanPostProcessor接口的实现，然后注册到容器中。





### BeanFactoryPostProcessor接口

当需要对Bean工厂进行预处理时，可以新建一个实现BeanFactoryPostProcessor接口的类，并将该类配置到Spring容器中。

## Spring bean的生命周期

<http://www.cnblogs.com/zrtqsk/p/3735273.html>

<http://blog.csdn.net/qq_33642117/article/details/51924653>

生命周期执行的过程如下:  
1)spring对bean进行实例化,默认bean是单例  
2)spring对bean进行依赖注入  
3)如果bean实现了BeanNameAware接口,spring将bean的id传给setBeanName()方法  
4)如果bean实现了BeanFactoryAware接口,spring将调用setBeanFactory方法,将BeanFactory实例传进来  
5)如果bean实现了ApplicationContextAware()接口,spring将调用setApplicationContext()方法将应用上下文的引用传入  
6) 如果bean实现了BeanPostProcessor接口,spring将调用它们的postProcessBeforeInitialization接口方法  
7) 如果bean实现了InitializingBean接口,spring将调用它们的afterPropertiesSet接口方法,类似的如果bean使用了init-method属性声明了初始化方法,改方法也会被调用  
8)如果bean实现了BeanPostProcessor接口,spring将调用它们的postProcessAfterInitialization接口方法  
9)此时bean已经准备就绪,可以被应用程序使用了,他们将一直驻留在应用上下文中,直到该应用上下文被销毁  
10)若bean实现了DisposableBean接口,spring将调用它的distroy()接口方法。同样的,如果bean使用了destroy-method属性声明了销毁方法,则该方法被调用

Spring bean factory 负责管理在spring容器中被创建的bean的生命周期。Bean的生命周期由两组回调（call back）方法组成。

1. 初始化之后调用的回调方法。
2. 销毁之前调用的回调方法。

Spring框架提供了以下四种方式来管理bean的生命周期事件：

* InitializingBean和DisposableBean回调接口
* 针对特殊行为的其他Aware接口
* Bean配置文件中的Custom init()方法和destroy()方法
* @PostConstruct和@PreDestroy注解方式

**简单的说就是：**

**对于单例，spring在启动时会去读取xml文件，创建实例对象。在创建对象的时候会先调用构造器，然后调用init-method属性值中的方法。在对象被销毁的时候，会调用destroy-method属性值中所指定的方法。Lazy-init=true可以让这个对象在第一次被访问时被创建。**

**对于非单例，spring在读取xml文件时，不会去创建对象，在每一次访问这个对象时，spring容器才会去创建并调用init-method属性值中的方法，在对象销毁时，spring不会帮我们调用任何方法，因为非单例，spring在创建完毕之后就不会管理这个对象了。**

## Spring bean的作用域

1. singleton：这种bean范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个bean的实例，单例的模式由bean factory自身来维护。
2. prototype：原形范围与单例范围相反，为每一个bean请求提供一个实例。
3. request：在请求bean范围内会每一个来自客户端的网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。
4. Session：与请求范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
5. global-session：global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。

全局作用域与Servlet中的session作用域效果相同

## Spring内部bean

就是在一个bean的内部再装配一个bean。那么此时内部bean是不能被其他bean使用的。可以通过setter注入属性和构造方法注入构造参数的方式实现。



## Spring中单例bean是线程安全的么

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。

最浅显的解决办法就是将多态bean的作用域由**“singleton**”变更为“**prototype**”。

## Spring中注入集合

Spring提供了以下四种集合类的配置元素：

* **<list>** :   该标签用来装配可重复的list值。
* **<set>** :    该标签用来装配没有重复的set值。
* **<map>**:   该标签可用来注入键和值可以为任何类型的键值对。
* **<props>** : 该标签支持注入键和值都是字符串类型的键值对。

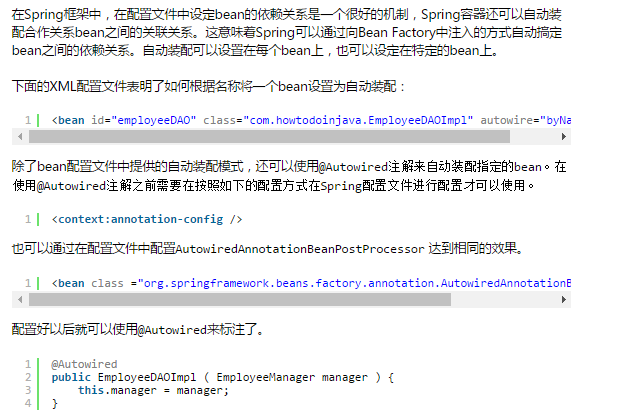


## Spring中注入一个java.util.Properties



## Spring的自动装配及分类

装配:spring容器把bean组装到一起,前提是容器需要知道bean的依赖关系,通过依赖注入将他们装配到一起.



少写几个ref

1. **no：**这是Spring框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在bean定义中用标签明确的设置依赖关系。
2. **byName：**该选项可以根据bean名称设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的名称自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。
3. **byType：**该选项可以根据bean类型设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的类型自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。
4. **constructor：**造器的自动装配和byType模式类似，但是仅仅适用于与有构造器相同参数的bean，如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。
5. **autodetect：**该模式自动探测使用构造器自动装配或者byType自动装配。首先，首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在bean内部没有找到相应的构造器或者是无参构造器，容器就会自动选择byTpe的自动装配方式。

## 如何开启基于注解的自动装配

要使用 @Autowired，需要注册 AutowiredAnnotationBeanPostProcessor，可以有以下两种方式来实现：

1 引入配置文件中的<bean>下引入 <context:annotation-config>

<beans>

    <context:annotation-config />

</beans>

2 在bean配置文件中直接引入AutowiredAnnotationBeanPostProcessor

<beans>

    <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

</beans>

## @Required注解

在产品级别的应用中，IoC容器可能声明了数十万了bean，bean与bean之间有着复杂的依赖关系。设值注解方法的短板之一就是验证所有的属性是否被注解是一项十分困难的操作。可以通过在<bean>中设置“**dependency-check**”来解决这个问题。

在应用程序的生命周期中，你可能不大愿意花时间在验证所有bean的属性是否按照上下文文件正确配置。或者你宁可验证某个bean的特定属性是否被正确的设置。即使是用“**dependency-check**”属性也不能很好的解决这个问题，在这种情况下，你需要使用@Required 注解。

@ Required 注解应用于bean属性的setter方法上，只会检查属性是否已经设置而不会测试属性是否为空。



检查bean的属性是否被严格的设置。dependency-check”（依赖检查）属性将会非常有用。它默认为“none”，不进行依赖检查。“simple”会核对所有的原始类型和String类型的属性。“objects”只做对象间的关联检查（包括集合）。“all”会检查所有的属性，包括“simple”和“objects”。

## @Autowired注解

@Autowired注解可以像@Required注解、构造器一样被用于在bean的设值方法上自动装配bean的属性，一个参数或者带有任意名称或带有多个参数的方法。

比如，可以在设值方法上使用@Autowired注解来替代配置文件中的 <property>元素。当Spring容器在setter方法上找到@Autowired注解时，会尝试用**byType** 自动装配。如果有多个，则会再按照byName方式比对，如果还有多个，则报异常。

当然我们也可以在构造方法上使用@Autowired 注解。带有@Autowired 注解的构造方法意味着在创建一个bean时将会被自动装配，即便在配置文件中使用<constructor-arg> 元素。

@Autowired也可以手动指定按照byName方式注入，使用@Qualifier标签，例如：

@Autowired ()

@Qualifier ( "beanName" )

@Resource标签是按照byName方式注入的

## @Resource注解

@Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入，而@Resource默认按 byName自动注入罢了。@Resource有两个属性是比较重要的，分是name和type，Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。

@Resource装配顺序   
　　1. 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常   
　　2. 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常   
　　3. 如果指定了type，则从上下文中找到类型匹配的唯一bean进行装配，找不到或者找到多个，都会抛出异常   
　　4. 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配；

## @Qualifier注解

当使用@Autowired注解时，如果此时匹配的类型有多个，则会按照byName去查找。但是如果还会有多个，就会报出异常。使用@Qualifier注解，可以可以按照指定的名称去匹配。



## 构造方法注入和setter方法注入区别

设置注入：

* 设值注入需要该Bean包含这些属性的setter方法
* 与传统的JavaBean的写法更相似，程序开发人员更容易理解、接收。
* 对于复杂的依赖关系，如果采用构造注入，会导致构造器臃肿，难以阅读。Spring在创建Bean实例时，需要同时实例化器依赖的全部实例，因而导致性能下降。而使用设值注入，则能避免这些问题
* 尤其是在某些属性可选的情况况下，多参数的构造器显得更加笨重

构造注入：

* 构造注入需要该Bean包含带有这些属性的构造器
* 构造注入可以在构造器中决定依赖关系的注入顺序，优先依赖的优先注入。例如，组件中其他依赖关系的注入，常常要依赖于DataSrouce的注入。采用构造注入，可以在代码中清晰的决定注入顺序。
* 对于依赖关系无需变化的Bean，构造注入更有用处。因为没有Setter方法，所有的依赖关系全部在构造器内设定。因此，无需担心后续的代码对依赖关系产生破坏。
* 依赖关系只能在构造器中设定，则只有组件的创建者才能改变组件的依赖关系。对组件的调用者而言，组件内部的依赖关系完全透明，更符合高内聚的原则

更重要的是对于循环依赖的解决。

## Spring中循环依赖问题及解决

<http://jinnianshilongnian.iteye.com/blog/1415278>

**spring**容器循环依赖包括构造器循环依赖和setter循环依赖。

Spring容器能够顺利的实例化以构造函数注入方式配置bean的前提：bean构造函数参数引用的对象必须已经准备就绪。如果两个bean都采用构造注入，并通过构造函数参数引用了对方，就会发生类似线程死锁的循环依赖问题。

1 构造器循环依赖问题

通过构造器注入构成的循环依赖，这种依赖是无法解决的。只能抛出BeanCurrentlyIn CreationException异常表示循环依赖。

比如，在创建A类时，构造器需要B类，将去创建B类，在创建B类时，发现又需要A类，就去创建A类…从而形成了循环依赖，没办法创建。

Spring容器将每一个正在创建的bean标识符放在一个“当前创建bean池”中，bean标识符在创建过程中会一直保持在这个池中，因为如果在创建bean过程中发现自己已经在“当前创建bean池”中，会抛出BeanCurrentlyIn CreationException异常。对于创建完毕的bean，将从当前创建bean池中清除掉。

2 setter循环依赖

通过setter注入方式构成的循环依赖，是通过spring容器提前暴露刚完成构造器注入但未完成其他步骤（setter注入）的bean来完成的。但是只能解决单例作用域的bean循环依赖。通过提前暴露一个单例工厂方法，从而使其他bean能引用到该bean，如下代码所示：

1. addSingletonFactory(beanName, new ObjectFactory() {
2. public Object getObject() throws BeansException {
3. return getEarlyBeanReference(beanName, mbd, bean);
4. }
5. });

Spring容器创建A时，首先根据无参构造器创建bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的bean，并将A的标识符放到当前创建bean中，然后进行setter注入B类。

Spring容器创建B类时，首先根据无参构造器创建bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的bean，并将B类标识符放到当前创建bean池中，然后进行setter注入A，由于提前暴露了“ObjectFactory”工厂，从而使用它返回提前暴露一个创建中的bean。

完成了依赖注入。

对于非单例（prototype）作用域的bean，spring容器无法完成依赖注入。因为spring容器不会进行缓存prototype作用域的bean，因此无法提前暴露一个创建中的bean。

对于"singleton"作用域bean，可以通过"setAllowCircularReferences(false）；"来禁用循环引用。

## Spring缓存bean

入口方法：

Object sharedInstance = getSingleton(beanName);

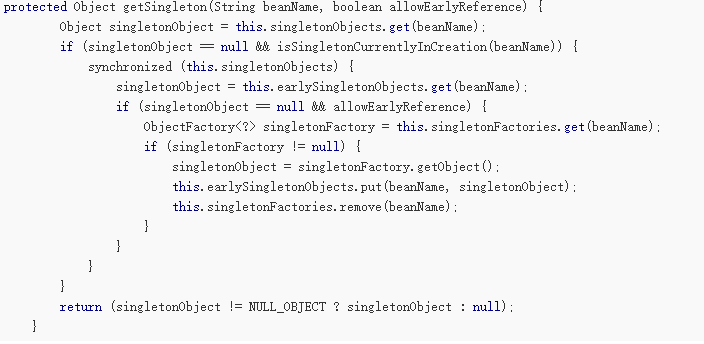
@Override

public Object getSingleton(String beanName) {

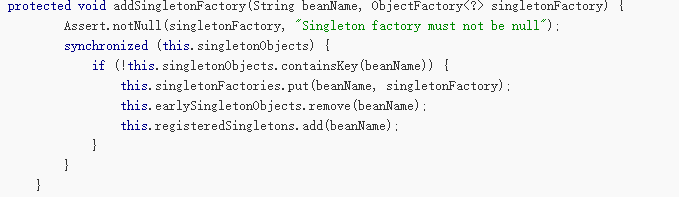
return getSingleton(beanName, true);

}

真正的实现：



Spring单例bean在spring容器中只会创建一次，之后在获取bean时，会首先尝试从缓存中加载bean，首先从singletonObjects中获取，singletonObjects中存储的是BeanName和Bean instance的映射关系，如果为空，并且该bean正在创建(isSingletonCurrentlyCreation)，则尝试从earlySingleObjects中获取。如果为空，则从singletonFactories中获取对应的singletonFactory。因为spring在创建单例bean时，可能存在循环依赖问题，为了解决循环依赖问题，spring采取了一种将bean的实例提前暴露加入到缓存中。提前暴露bean实例到缓存的时机是在bean实例创建（调用构造方法）之后，初始化bean实例（属性注入）之前。在创建bean实例doCreateBean中，该方法调用了DefaultSingletonRegitry类的



将允许提前暴露的bean提前加入到singletonFactories中，这样在创建依赖时可以避免循环依赖问题。

从singletonFactories中获取bean后，会将其存储到earlySingletonObjects中，然后从singletonfactories中移除该bean，之后获取bean时，就直接从earlySingletonObjects中获取。

SingletonObjects：存储的是完全实例化bean name到bean instance的映射

EarlySingletonObjects:存储的是提前保留的bean name到bean instance的映射（主要是为了解决循环依赖设置的）

SingletonFactories:用于保存beanName和创建bean的工厂之间的关系

RegisteredSingletons:用于保存当前所有已经注册的bean

## Spring框架中的事件机制

<http://blog.csdn.net/blueboz/article/details/49949573>

<http://xls9577087.iteye.com/blog/2121752>

事件驱动模型采用的是观察者模式或者发布-订阅模式：

1 是一种对象间的一对多的关系，如信号灯是目标（1），行人注视着信号灯（多方）

2 当目标发送改变（发布）时，观察者就可以接收到改变

3 观察者如何处理，目标无需干涉，所以就松散耦合了它们之间的关系。

我们可以自己把普通的java事件操作定义为bean集成到bean容器中，也可以使用spring提供的事件支持。

Spring的ApplicationContext 提供了支持事件和代码中监听器的功能。

**ApplicationEvent：**抽象类，继承自EventObject，同时是spring的application中事件的父类，需要被自定义的事件继承。

**ApplicationListener：**继承自EventListener，spring的application中的监听器必须实现的接口，需要被自定义的监听器实现其onApplicationEvent方法

 ApplicationEventPublisherAware：在spring的context中希望能发布事件的类必须实现的接口，该接口中定义了设置ApplicationEventPublisher的方法，由ApplicationContext调用并设置。在自己实现的ApplicationEventPublisherAware子类中，需要有ApplicationEventPublisher属性的定义。

ApplicationEventPublisher：spring的事件发布者接口，定义了发布事件的接口方法publishEvent。**因为ApplicationContext实现了该接口，**因此spring的ApplicationContext实例具有发布事件的功能(publishEvent方法在AbstractApplicationContext中有实现)。在使用的时候，只需要把ApplicationEventPublisher的引用定义到ApplicationEventPublisherAware的实现中，spring容器会完成对ApplicationEventPublisher的注入。

在spring的bean配置中，因为事件是由事件源发出的，不需要注册为bean由spring容器管理。所以在spring的配置中，只需配置自定义的ApplicationEventListener和publisherAware(即实现了ApplicationEventPublisherAware接口的发布类)，而对于ApplicationEventPublisher的管理和注入都由容器来完成。

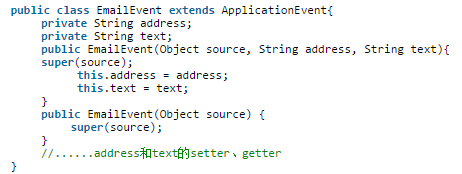
------------------------------------------------------------------------------------------------------------------------------

ApplicationContext事件机制是观察者设计模式的实现，通过ApplicationEvent类和ApplicationListener接口，可以实现ApplicationContext事件处理。如果容器中有一个ApplicationListener Bean，每当ApplicationContext发布ApplicationEvent时，ApplicationListener Bean将自动被触发。

Spring的事件框架有如下两个重要的成员：

* ApplicationEvent：容器事件，必须由ApplicationContext发布
* ApplicationListener：监听器，可由容器中的任何监听器Bean担任

实际上，Spring的事件机制与所有时间机制都基本相似，它们都需要事件源、事件和事件监听器组成。只是此处的事件源是ApplicationContext，且事件必须由Java程序显式触发。下面的程序将演示Spring容器的事件机制。程序先定义了一个ApplicationEvent类，其对象就是一个Spring容器事件。



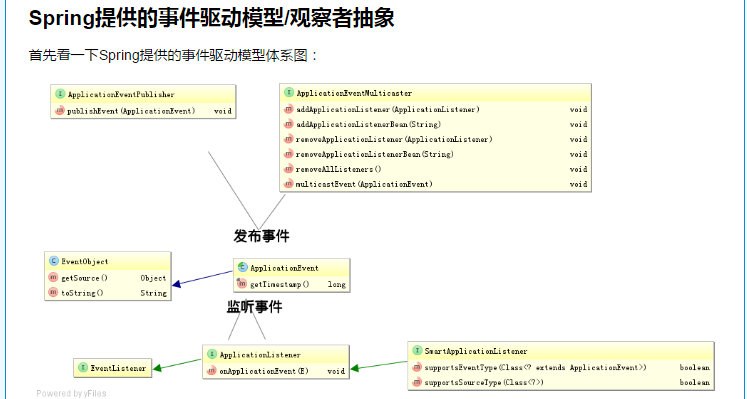


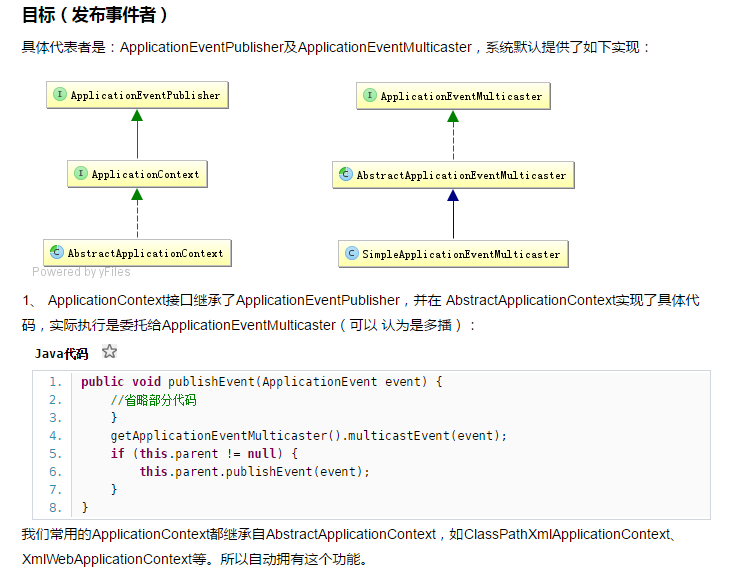
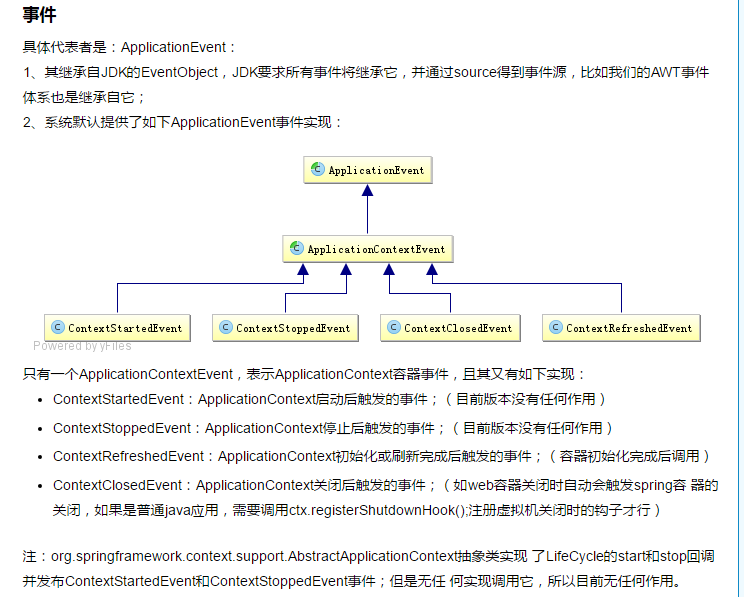


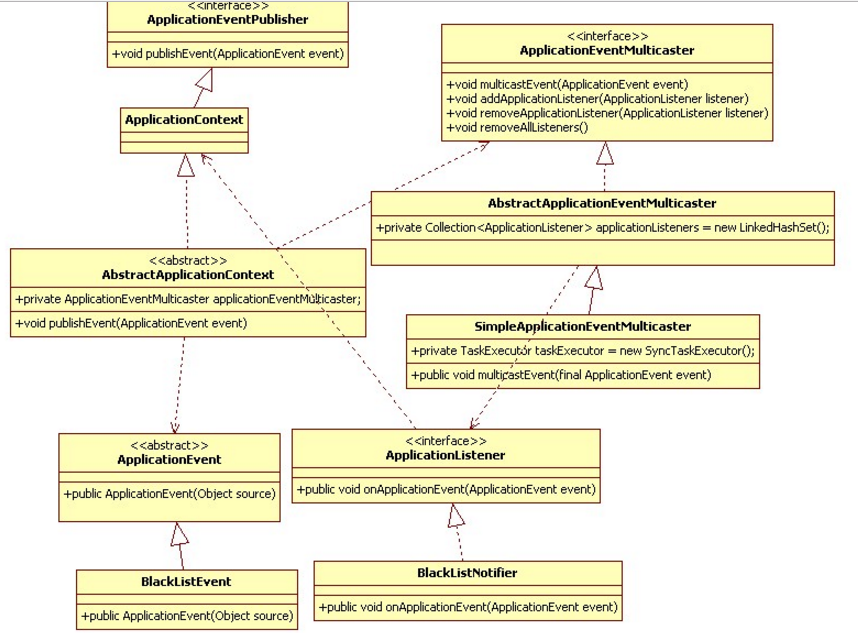
上述是使用applicationContext作为事件源，来发布某个事件的。也可以自己定义一个事件源用来发布事件，需要实现ApplicationEventPublisherAware接口。这样该类就可以有发布事件的功能了。

也可以









* **ApplicationListener 就是我们的 Observer，需要到容器中注册。他要关心他所关心的ApplicationEvent 。一般有如下代码：if (event instanceof BlackListEvent) {}**
* **ApplicationEventMulticaster是我们的SUBJECT一个代理。他会管理我们的 ApplicationListener 。**
* **ApplicationEvent 是事件，它就是媒介，充当介质的作用。**



Spring提供如下几个内置事件：

* ContextRefreshedEvent：上下文更新事件。ApplicationContext容器初始化或刷新时触发该事件。此处的初始化是指：所有的Bean被成功装载，后处理Bean被检测并激活，所有Singleton Bean 被预实例化，ApplicationContext容器已就绪可用
* ContextStartedEvent：上下文开始事件。当使用ConfigurableApplicationContext(ApplicationContext的子接口）接口的start()方法启动ApplicationContext容器时触发该事件。容器管理声明周期的Bean实例将获得一个指定的启动信号，这在经常需要停止后重新启动的场合比较常见
* ContextClosedEvent：上下文关闭事件。当使用ConfigurableApplicationContext接口的close()方法关闭ApplicationContext时触发该事件。容器关闭时，其管理的单例bean全部被销毁。
* ContextStoppedEvent：上下文停止事件。当使用ConfigurableApplicationContext接口的stop()方法使ApplicationContext容器停止时触发该事件。此处的停止，意味着容器管理生命周期的Bean实例将获得一个指定的停止信号，被停止的Spring容器可再次调用start()方法重新启动
* RequestHandledEvent：请求处理事件。请求处理结束后触发。Web相关事件，只能应用于使用DispatcherServlet的Web应用。在使用Spring作为前端的MVC控制器时，当Spring处理用户请求结束后，系统会自动触发该事件。

## FileSystemResource和ClassPathResource

在FileSystemResource 中需要给出spring-config.xml文件在你项目中的相对路径或者绝对路径。在ClassPathResource中spring会在ClassPath中自动搜寻配置文件，所以要把ClassPathResource 文件放在ClassPath下。

如果将spring-config.xml保存在了src文件夹下的话，只需给出配置文件的名称即可，因为src文件夹是默认。

**简而言之，ClassPathResource在环境变量中读取配置文件，FileSystemResource在配置文件中读取配置文件**

## Spring框架中用到了哪些设计模式

<http://www.cnblogs.com/yuefan/p/3763898.html>

<http://blog.csdn.net/adoocoke/article/details/8286902>

代理模式——在AOP和remoting中用的比较多

单例模式——在spring配置文件中定义的bean默认是单例

模板方法——解决代码重复问题，如果HibernateTemplate

前端控制器——spring提供了DispatcherServlet来对请求进行分发

工厂模式——BeanFactory用来创建对象的实例

依赖注入——贯穿于BeanFactory/ApplicationContext接口的核心理念

观察者模式——事件监听机制

装饰器模式——BeanWrapper，获取bean的属性和行为

## Spring对hibernate的操作方式

继承HibernateDaoSupport类:提供一个AOP拦截器

或者组合HibernateTemplate：控制反转 callback

用spring的sessionfactory调用localsessionfactory。

配置hibernate sessionfactory

继承hibernatedaosupport

在aop支持的事务中装配

## spring支持的事务管理

<http://blog.csdn.net/z69183787/article/details/8199932>

<http://blog.csdn.net/trigl/article/details/50968079>

https://www.ibm.com/developerworks/cn/education/opensource/os-cn-spring-trans/

* **编程式事务管理**：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。
* **声明式事务管理：**这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

**一般选择声明式事务，因为它对应用代码影响最小，更符合一个无侵入的轻量级容器的思想。**

**事务指的是逻辑上的一组操作，这组操作要么全部成功，要么全部失败。例子：银行转账。**

**四大特性：原子性 一致性 隔离性 持久性**

## Spring对hibernate的事务管理

在没有使用spring时，hibernate每次进行一个操作，都需要先开启一个事务，然后进程数据操作，然后提交事务，关闭事务。Hibernate默认的事务自动提交是false，需要手动提交事务。可以改为自动提交，在hibernate.cfg.xml中修改：

<property name="hibernate.connection.autocommit">true</property>

使用spring整合后，我们不再是每次都去拿session进行数据操作，也不需要每次都开启事务，提交事务。Spring提供了hibernate template，直接使用这个类提供的数据操作方法就可以操作数据。前提是使用spring去管理hibernate的配置文件，不使用hibernate.cfg.xml文件，也就是说是在spring的配置文件中进行数据源一些的配置。Spring对事务的提交机制是默认提交的。可以把spring设置为手动提交：

<property name="defaultAutoCommit">

<value>false</value>

</property>

纵然我们把它的事务提交方式设置为自动，它可以进行数据操作，但是这样并不满足我们实际的业务需求，因为有时候在我保存一个数据之后，我希望他能继 续保存另一条数据，我希望在保存完两条或者多条之后一起进行事务提交，这样即使出错，我们可以回滚，取保数据的一致性，要么都成功要么都失败，这时候我们 就不能每保存完一条数据之后事务就自动提交，因为这样它们不在同一个事务当中，我们不能保证数据的一致行。所以这时候我们就需要手动的来配置我们的事务， 这就需要用到Spring为Hibernate提供的事务管理机制，Spring提供的事务管理可以分为两类：编程式的和声明式的，编程式，其实就是在代 码里面来控制，像Hibernate操作数据一样，开启事务，提交事务，这种方式有一定的局限性，所以我们一般是用声明式来配置我们的事务。

声明式事务配置主要分以下几步：

1、声明式事务配置

* 配置事务管理器；
* 事务的传播特性；
* 那些类那些方法使用事务。



我们在配置事务的时候，我们一般是把事务边界设置到service层，也就是你的业务逻辑层，因为我们很多时候都是在我们的业务逻辑层来完成我们一 些列的数据操作，如果放到Dao数据层，其粒度太小了。另外，如果我们把事务配置在业务逻辑层的话，对我们的二级缓存也是有好处的，这个大家以后实际操作 的时候会发现。

2、编写业务逻辑方法

这时候我们就可以在我们业务逻辑层用HibernateTemplate里面提供的数据操作方法来编写我们的业务逻辑方法了，当然我们的方法必须要 是以我们事务配置里面配置的一样，用save，delete，update，get做我们的方法的开头。需要注意的是，默认情况下运行期异常才会回滚（包 括继承了RuntimeException子类），普通异常是不会滚的。

最后我们顺便总结一下事务的几种传播特性：

1. PROPAGATION\_REQUIRED: 如果存在一个事务，则支持当前事务。如果没有事务则开启；

2. PROPAGATION\_SUPPORTS: 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行；

3. PROPAGATION\_MANDATORY: 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常；

4. PROPAGATION\_REQUIRES\_NEW: 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起；

5. PROPAGATION\_NOT\_SUPPORTED: 总是非事务地执行，并挂起任何存在的事务；

6. PROPAGATION\_NEVER: 总是非事务地执行，如果存在一个活动事务，则抛出异常；

7. PROPAGATION\_NESTED：如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按TransactionDefinition.PROPAGATION\_REQUIRED 属性执行。

## AOP

http://blog.csdn.net/moreevan/article/details/11977115/

AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

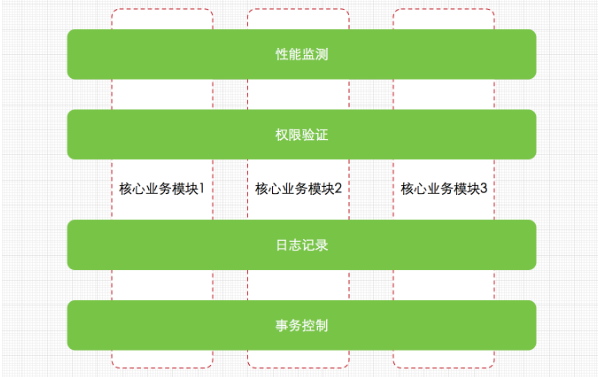
## Spring AOP和Aspectj

<http://blog.csdn.net/javazejian/article/details/56267036>

<http://blog.csdn.net/a128953ad/article/details/50509437>

<http://www.oschina.net/translate/comparative_analysis_between_spring_aop_and_aspectj>

AOP面向切面编程。AOP将软件系统分为核心关注点和横切关注点，核心关注点就是业务功能处理的部分，而横切关注点是与业务逻辑无关的部分，比如事务操作、权限认证以及日志输出等，横切关注点的特点是散布于系统的各个地方，经常出现在核心关注点的周围。AOP将这些横切关注点提取出来并封装，以在核心关注点的执行前后来执行这些横切关注点。这样做大大提高了代码的复用性，减少了代码冗余和代码混乱。



核心模块只需要关注自己相关的业务，当需要外围的业务（日志 权限 事务等）时，这些外围业务会自动应用到核心模块中。

实现AOP比较流行的两种的技术是AspectJ和Spring AOP。

### AspectJ

#### 基本知识点

例子：使用AspectJ技术切入该类

public class HelloWord {

public void sayHello(){

System.out.println("hello world !");

}

public static void main(String args[]){

HelloWord helloWord =new HelloWord();

helloWord.sayHello();

}

}

编写AspectJ类 关键词为aspect

public aspect MyAspectJDemo {

/\*\*

\* 定义切点,日志记录切点

\*/

pointcut recordLog():call(\* HelloWord.sayHello(..));

/\*\*

\* 定义切点,权限验证(实际开发中日志和权限一般会放在不同的切面中,这里仅为方便演示)

\*/

pointcut authCheck():call(\* HelloWord.sayHello(..));

/\*\*

\* 定义前置通知!

\*/

before():authCheck(){

System.out.println("sayHello方法执行前验证权限");

}

/\*\*

\* 定义后置通知

\*/

after():recordLog(){

System.out.println("sayHello方法执行后记录日志");

}

}

运行结果：

sayHello方法执行前验证权限

hello world !

sayHello方法执行后记录日志

AspectJ是java实现的AOP框架，能够对java代码进行AOP编译（一般在编译期进行）。让java代码具有AspectJ的AOP功能（当然需要特殊的编译器），可以这样说AspectJ是目前实现AOP框架中最成熟，功能最丰富的语言，更幸运的是，AspectJ与java程序完全兼容，几乎是无缝关联。

上述的aspect关键字定义了一个类，这个类就是一个切面，在切面内部使用了pointcut定义了两个切点，一个用于权限验证，一个用于日志记录，而所谓的切点就是那些需要应用切面的方法，如需要在sayHello方法执行前后进行权限验证和日志记录，那么就需要捕捉该方法，而pointcut就是定义这些需要捕捉的方法（常常是不止一个方法的），这些方法也称为目标方法，最后还定义了两个通知，通知就是那些需要在目标方法前后执行的函数，如before()即前置通知在目标方法之前执行，即在sayHello()方法执行前进行权限验证，另一个是after()即后置通知，在sayHello()之后执行，如进行日志记录。

切点的定义语法：关键字为pointcut，定义切点，后面跟着函数名称，最后编写匹配表达式，此时函数一般使用call()或者execution()进行匹配。

pointcut 函数名 : 匹配表达式

通知的定义语法：

[返回值类型] 通知函数名称(参数) [returning/throwing 表达式]：连接点函数(切点函数){   
函数体   
}

通知的类型：

* before 目标方法执行前执行，前置通知
* after 目标方法执行后执行，后置通知
* after returning 目标方法返回时执行 ，后置返回通知
* after throwing 目标方法抛出异常时执行 异常通知
* around 在目标函数执行中执行，可控制目标函数是否执行，环绕通知

案例如下：

/\*\*

\* 定义前置通知

\*

\* before(参数):连接点函数{

\* 函数体

\* }

\*/

before():authCheck(){

System.out.println("sayHello方法执行前验证权限");

}

/\*\*

\* 定义后置通知

\* after(参数):连接点函数{

\* 函数体

\* }

\*/

after():recordLog(){

System.out.println("sayHello方法执行后记录日志");

}

/\*\*

\* 定义后置通知带返回值

\* after(参数)returning(返回值类型):连接点函数{

\* 函数体

\* }

\*/

after()returning(int x): get(){

System.out.println("返回值为:"+x);

}

/\*\*

\* 异常通知

\* after(参数) throwing(返回值类型):连接点函数{

\* 函数体

\* }

\*/

after() throwing(Exception e):sayHello2(){

System.out.println("抛出异常:"+e.toString());

}

/\*\*

\* 环绕通知 可通过proceed()控制目标函数是否执行

\* Object around(参数):连接点函数{

\* 函数体

\* Object result=proceed();//执行目标函数

\* return result;

\* }

\*/

Object around():aroundAdvice(){

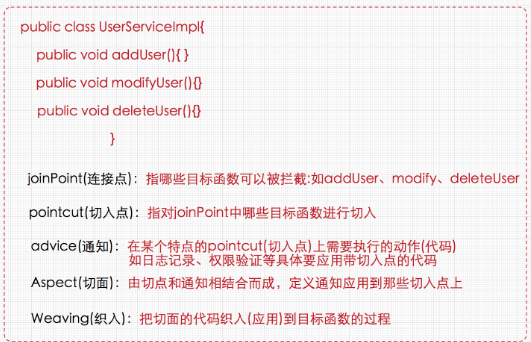
System.out.println("sayAround 执行前执行");

Object result=proceed();//执行目标函数

System.out.println("sayAround 执行后执行");

return result;

}



#### 原理

织入：将aspect(切面)应用到目标函数(类)的过程。

分为动态织入和静态织入。

动态织入的方式是在运行时将要增强的代码织入到目标类中，往往通过动态代理来实现。如Java JDK的动态代理（Proxy 底层通过反射实现），或者CGLIB的动态代理（底层通过继承实现），Spring AOP采用的基于运行时增强的代理技术。

AspectJ采用的静态织入的方式，AspectJ主要采用的编译期织入，在这个期间使用AspectJ的ajc编译器把aspect类编译成class字节码后，在java目标类编译时织入，即先编译aspect类再编译目标类。

ajc编译器可以识别aspect语法的编译器，采用java语言编写。ajc编译器也可以编译java文件。

上述案例的反编译的代码如下：

public class HelloWord {

public HelloWord() {

}

public void sayHello() {

System.out.println("hello world !");

}

public static void main(String[] args) {

HelloWord helloWord = new HelloWord();

HelloWord var10000 = helloWord;

try {

//MyAspectJDemo 切面类的前置通知织入

MyAspectJDemo.aspectOf().ajc$before$com\_zejian\_demo\_MyAspectJDemo$1$22c5541();

//目标类函数的调用

var10000.sayHello();

} catch (Throwable var3) {

MyAspectJDemo.aspectOf().ajc$after$com\_zejian\_demo\_MyAspectJDemo$2$4d789574();

throw var3;

}

//MyAspectJDemo 切面类的后置通知织入

MyAspectJDemo.aspectOf().ajc$after$com\_zejian\_demo\_MyAspectJDemo$2$4d789574();

}

}

当然除了编译期织入，还存在链接期(编译后)织入，即将aspect类和java目标类同时编译成字节码文件后，再进行织入处理，这种方式比较有助于已编译好的第三方jar和Class文件进行织入操作。

### Spring AOP

Spring AOP注重过的是与IOC容器的结合，并以此来解决横切业务的问题。Spring AOP并没有提供完整的AOP功能，Spring AOP采用动态代理技术来构建Spring AOP的内部机制（动态织入），这是与AspectJ（静态织入）根本区别。Spring2.0后采用了和AspectJ一样的注解，并没有使用AspectJ的编译器，底层是动态代理技术实现。

案例：

//接口类

public interface UserDao {

int addUser();

void updateUser();

void deleteUser();

void findUser();

}

@Repository

public class UserDaoImp implements UserDao {

@Override

public int addUser() {

System.out.println("add user ......");

return 6666;

}

@Override

public void updateUser() {

System.out.println("update user ......");

}

@Override

public void deleteUser() {

System.out.println("delete user ......");

}

@Override

public void findUser() {

System.out.println("find user ......");

}

}

@Aspect

public class MyAspect {

/\*\*

\* 前置通知

\*/

@Before("execution(\* com.zejian.spring.springAop.dao.UserDao.addUser(..))")

public void before(){

System.out.println("前置通知....");

}

/\*\*

\* 后置通知

\* returnVal,切点方法执行后的返回值

\*/

@AfterReturning(value="execution(\* com.zejian.spring.springAop.dao.UserDao.addUser(..))",returning = "returnVal")

public void AfterReturning(Object returnVal){

System.out.println("后置通知...."+returnVal);

}

/\*\*

\* 环绕通知

\* @param joinPoint 可用于执行切点的类

\* @return

\* @throws Throwable

\*/

@Around("execution(\* com.zejian.spring.springAop.dao.UserDao.addUser(..))")

public Object around(ProceedingJoinPoint joinPoint) throws Throwable {

System.out.println("环绕通知前....");

Object obj= (Object) joinPoint.proceed();

System.out.println("环绕通知后....");

return obj;

}

/\*\*

\* 抛出通知

\* @param e

\*/

@AfterThrowing(value="execution(\* com.zejian.spring.springAop.dao.UserDao.addUser(..))",throwing = "e")

public void afterThrowable(Throwable e){

System.out.println("出现异常:msg="+e.getMessage());

}

/\*\*

\* 无论什么情况下都会执行的方法

\*/

@After(value="execution(\* com.zejian.spring.springAop.dao.UserDao.addUser(..))")

public void after(){

System.out.println("最终通知....");

}

}

Spring AOP中定义五种注解类型的通知函数，分别是前置通知@Before、后置通知@AfterReturning、环绕通知@Around、异常通知@AfterThrowing、最终通知@After。

注解通知上使用execution关键字定义的切点表达式，即指明该通知要应用的目标函数，当只有一个execution参数时，value属性可以省略，当含两个以上的参数，value必须注明，如存在返回值时。当然除了把切点表达式直接传递给通知注解类型外，还可以使用@pointcut来定义切点匹配表达式。

如果要使用Spring AOP的aspectJ功能时，需要启动aspectJ的注解功能：

<aop:aspectj-autoproxy />

### 两者区别

**织入时期不同**

Spring AOP采用的动态织入，AspectJ采用的静态织入。（静态织入指的是在编译期就织入。动态织入分为动和静两种，静指的是织入过程只在第一次调用时执行，动指的是根据代码运行的中间状态来决定如何操作，每次调用Target时都执行。）

**使用对象不同**

Spring AOP的通知是基于该对象是Spring Bean对象，而AspectJ对象可以在任何java对象上应用通知。

Spring AOP：如果你想要在通过this对象调用的方法上应用通知，那么你必须使用currentProxy对象，并调用其上的相应方法;于此相似，如果你想要在某对象的方法上应用通知，那么你必须使用与该对象相应的Spring bean

   AspectJ：使用AspectJ的一个间接局限是，因为AspectJ通知可以应用于POJO之上，它有可能将通知应用于一个已配置的通知之上。对于一个你没有注意到这方面问题的大范围应用的通知，这有可能导致一个无限循环。

也就是说最主要的区别是 1 spring aop是动态织入，aspectj是静态织入

2 spring aop是为了和ioc紧密结合，只能用于spring bean对象上，而aspectj可以用于基于普通java对象的模块化。

如何选择？应该明确在应用横切关注点时（事务管理 日志），需要处理的是spring bean还是pojo。如果正在开发新的应用，那么spring aop可以使用，但是如果正在维护一个没有使用spring框架的应用，那么使用aspectj。

例子 将日志功能作为一个通知advice加入到应用中，用于追踪程序的流程，那么该通知advice就只能应用在spring beans 的连接点joinpoint之上。

<bean id="myServices" class="com.ashutosh.MyServicesImpl " />

<aop:config>

<aop:aspect id="loggingAspect" ref="logging">

<aop:around method="log" pointcut="execution(public \* \*(..))"/>

</aop:aspect>

</aop:config -->

ApplicationContext beans =newClassPathXmlApplicationContext("appbeans.xml");

MyServices myServices = (MyServices) beans.getBean("myServices");

myServices.service(); // Logging advice applied here

但是当在service方法中调用同一个类的其他方法时，如果没有使用代理对象，那么日志通知就不会应用到该方法的调用上。

public void service() {

performOperation();// Logging advice not going to apply here

}

如果想通过this对象调用的方法上应用通知，那么必须使用currentProxy对象，并调用其上的相应方法。

public void service() {

// Logging advice going to apply here

((MyServices) AopContext.currentProxy()).performOperation();

}

于此相似，如果你想要在某对象的方法上应用通知，那么你必须使用与该对象相应的Spring bean。

public void service() {

MyObject obj = new MyObject();

Obj.performOperation();// Logging advice not going to apply here

}

如果你想要应用该通知，那么上述代码必须修改为如下形式。

public void service() {

MyObject obj = new MyObject();

Obj.performOperation();// Logging advice not going to apply here

ApplicationContext beans =newClassPathXmlApplicationContext("appbeans.xml");

MyObject obj =(MyObject) beans.getBean("myObject");

obj.performOperation()// Logging advice applied here

}

于此不同，使用“AspectJ”你可以在任何Java对象上应用通知，而不需要在任何文件中创建或配置任何bean。

另一个需要考虑的因素是，你是希望在编译期间进行织入(weaving)，还是编译后(post-compile)或是运行时(run-time)。Spring只支持运行时织入。如果你有多个团队分别开发多个使用Spring编写的模块（导致生成多个jar文件，例如每个模块一个jar文件），并且其中一个团队想要在整个项目中的所有Spring bean（例如，包括已经被其他团队打包了的jar文件）上应用日志通知（在这里日志只是用于加入横切关注点的举例），那么通过配置该团队自己的Spring配置文件就可以轻松做到这一点。之所以可以这样做，就是因为Spring使用的是运行时织入。

<!—Configuration -->

<bean id="myServices" class="com.ashutosh.MyServicesImpl " />

<aop:config>

<aop:aspect id="loggingAspect" ref="logging">

<aop:around method="log" pointcut="execution(public \* \*(..))"/>

</aop:aspect>

</aop:config -->

如果你使用AspectJ想要做到同样的事情，你也许就需要使用acj（AspectJ编译器）重新编译所有的代码并且进行重新打包。否则，你也可以选择使用AspectJ编译后(post-compile)或载入时(load-time)织入。

因为Spring基于代理模式（使用CGLIB），它有一个使用限制，即无法在使用final修饰的bean上应用横切关注点。因为代理需要对Java类进行继承，一旦使用了关键字final，这将是无法做到的。

例如，在Spring bean MyServicesImpl上使用关键字final，并配置一个“execution(public \* \*(..))”这样的切入点，将导致运行时异常(exception)，因为Spring不能为MyServicesImpl生成代理。

// Configuration file

<bean id="myServices" class="com.ashutosh.MyServicesImpl" />

//Java file

public final classMyServicesImpl {

---

}

在这种情况下，你也许会考虑使用AspectJ，其支持编译期织入且不需要生成代理。

于此相似，在static和final方法上应用横切关注点也是无法做到的。因为Spring基于代理模式。如果你在这些方法上配置通知，将导致运行时异常，因为static和final方法是不能被覆盖的。在这种情况下，你也会考虑使用AspectJ，因为其支持编译期织入且不需要生成代理。

## Jdk和cglib动态代理区别

Jdk动态代理只能对实现了接口的类生成代理，因为代理类需要实现目标类的接口。

Cglib是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法实现增强。因为是继承，所以该类或者方法不能声明为final，final阻止继承和多态。

1. /\*\*
2. \*
3. \* JDK动态代理类
4. \*
5. \*
6. \*/
7. public class JDKProxy implements InvocationHandler {
9. private Object targetObject;//需要代理的目标对象
11. public Object newProxy(Object targetObject) {//将目标对象传入进行代理
12. this.targetObject = targetObject;
13. return Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),
14. targetObject.getClass().getInterfaces(), this);//返回代理对象
15. }
17. public Object invoke(Object proxy, Method method, Object[] args)//invoke方法
18. throws Throwable {
19. checkPopedom();//一般我们进行逻辑处理的函数比如这个地方是模拟检查权限
20. Object ret = null;      // 设置方法的返回值
21. ret  = method.invoke(targetObject, args);       //调用invoke方法，ret存储该方法的返回值
22. return ret;
23. }
25. private void checkPopedom() {//模拟检查权限的例子
26. System.out.println(".:检查权限  checkPopedom()!");
27. }
28. }
29. /\*\*
30. \* CGLibProxy动态代理类的实例
31. \*
32. \*
33. \*/
34. public class CGLibProxy implements MethodInterceptor {
36. private Object targetObject;// CGLib需要代理的目标对象
38. public Object createProxyObject(Object obj) {
39. this.targetObject = obj;
40. Enhancer enhancer = new Enhancer();
41. enhancer.setSuperclass(obj.getClass());
42. enhancer.setCallback(this);
43. Object proxyObj = enhancer.create();
44. return proxyObj;// 返回代理对象
45. }
47. public Object intercept(Object proxy, Method method, Object[] args,
48. MethodProxy methodProxy) throws Throwable {
49. Object obj = null;
50. if ("addUser".equals(method.getName())) {// 过滤方法
51. checkPopedom();// 检查权限
52. }
53. obj = method.invoke(targetObject, args);
54. return obj;
55. }
57. private void checkPopedom() {
58. System.out.println(".:检查权限  checkPopedom()!");
59. }
60. }
61. public class Client {
63. public static void main(String[] args) {
65. UserManager userManager = (UserManager) new CGLibProxy()
66. .createProxyObject(new UserManagerImpl());
67. System.out.println("-----------CGLibProxy-------------");
68. userManager.addUser("tom", "root");
69. System.out.println("-----------JDKProxy-------------");
70. JDKProxy jdkPrpxy = new JDKProxy();
71. UserManager userManagerJDK = (UserManager) jdkPrpxy
72. .newProxy(new UserManagerImpl());
73. userManagerJDK.addUser("tom", "root");
74. }
76. }

1 Jdk动态代理生成的代理类和委托类实现了相同的接口

2 cglib动态代理中生成的字节码更加复杂，生成的代理类是委托类的子类，且不能处理被final关键字修饰的方法

3 jdk采用反射机制调用委托类的方法，cglib采用类似索引的方式直接调用委托类方法

## Jdk和cglib代理类的应用

Spring aop中可以使用jdk动态代理和cglib代理。

Hibernate的懒加载中，使用的是cglib代理，不需要被代理的对象实现一个接口。在load时候，cglib会生成一个数据库实体类的代理类，此代理类开始的时候里面的属性值是null，只有当真正请求数据时，如user.getName时，这个代理类才会去数据库中查询出具体的值（通过cglib的回调机制），然后加载到代理类对应的属性中，最后返回给调用者。

## Cdlib动态代理原理

<http://blog.csdn.net/zghwaicsdn/article/details/50957474>

http://www.jianshu.com/p/13aa63e1ac95

Cglib可以为没有实现接口的类提供代理，当要代理的类没有实现接口或者为了更好的性能，cglib是一个不错的选择。

Cglib原理：动态生成一个要代理类的子类，子类会重写代理类的所有非final方法。在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。它比使用java反射的jdk动态代理要快。

Cglib底层：使用字节码处理框架ASM，来转换字节码并生成新的类。

缺点：对于final的类或者方法，无法进行代理。（基于继承）

Jdk的动态代理是通过反射类Proxy以及InvocationHandler回调接口实现的。

Jdk要求动态代理的类必须实现一个接口，这在实际中具有一定局限性，而且反射的效率并不高。

Cglib不需要代理类实现接口，cglib底层采用asm字节码生成框架，使用字节码技术生成代理类，比使用java反射效率要高。（但是在高版本jdk中，java反射的性能已经提升很好，比cglib差不了多少）

### 被代理的类

1. **public** **class** TargetObject {
2. **public** String method1(String paramName) {
3. **return** paramName;
4. }
6. **public** **int** method2(**int** count) {
7. **return** count;
8. }
10. **public** **int** method3(**int** count) {
11. **return** count;
12. }
14. @Override
15. **public** String toString() {
16. **return** "TargetObject []"+ getClass();
17. }
18. }

### 拦截器

定义一个拦截器。在调用目标方法时，CGLib会回调MethodInterceptor接口方法拦截，来实现你自己的代理逻辑，类似于JDK中的InvocationHandler接口。

1. **public** **class** TargetInterceptor **implements** MethodInterceptor{
3. /\*\*
4. \* 重写方法拦截在方法前和方法后加入业务
5. \* Object obj为目标对象
6. \* Method method为目标方法
7. \* Object[] params 为参数，
8. \* MethodProxy proxy CGlib方法代理对象
9. \*/
10. @Override
11. **public** Object intercept(Object obj, Method method, Object[] params,
12. MethodProxy proxy) **throws** Throwable {
13. System.out.println("调用前");
14. Object result = proxy.invokeSuper(obj, params);
15. System.out.println(" 调用后"+result);
16. **return** result;
17. }

20. }

参数：Object为由CGLib动态生成的代理类实例，Method为上文中实体类所调用的被代理的方法引用，Object[]为参数值列表，MethodProxy为生成的代理类对方法的代理引用。

返回：从代理实例的方法调用返回的值。

其中，proxy.invokeSuper(obj,arg)：

调用代理类实例上的proxy方法的父类方法（即实体类TargetObject中对应的方法）

在这个示例中，只在调用被代理类方法前后各打印了一句话，当然实际编程中可以是其它复杂逻辑。

### 生成动态代理类

1. **public** **class** TestCglib {
2. **public** **static** **void** main(String args[]) {
3. Enhancer enhancer =**new** Enhancer();
4. enhancer.setSuperclass(TargetObject.**class**);
5. enhancer.setCallback(**new** TargetInterceptor());
6. TargetObject targetObject2=(TargetObject)enhancer.create();
7. System.out.println(targetObject2);
8. System.out.println(targetObject2.method1("mmm1"));
9. System.out.println(targetObject2.method2(100));
10. System.out.println(targetObject2.method3(200));
11. }
12. }

这里Enhancer类是CGLib中的一个字节码增强器，它可以方便的对你想要处理的类进行扩展，以后会经常看到它。

首先将被代理类TargetObject设置成父类，然后设置拦截器TargetInterceptor，最后执行enhancer.create()动态生成一个代理类（字节码技术创建子类实例），并从Object强制转型成父类型TargetObject。

最后，在代理类上调用方法.

代理对象的生成过程是由Enhancer类实现，步骤如下：

1 生成代理类Class的二进制字节码

2 通过Class.forName加载二进制字节码，生成Class对象

3 通过反射机制获取实例构造，并初始化代理类对象

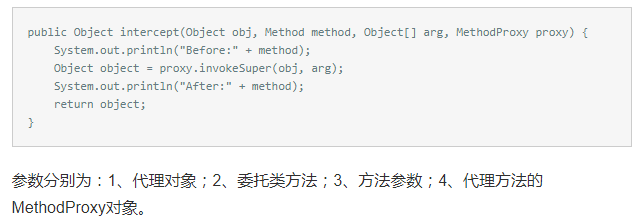
Enhancer是CGLIB的字节码增强器，方便对类进行扩展，内部调用。GeneratorStrategy.generate方法生成代理类的字节码，通过以下方式可以生成class文件。

通过Cglib生成的字节码比jdk实现来说要显得复杂：

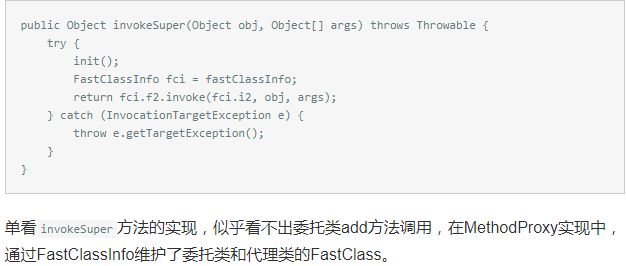
1 代理类UserService$$EnhancerByCGLIB$$394dddeb继承了委托类UserService，且委托类的final方法不能被代理。

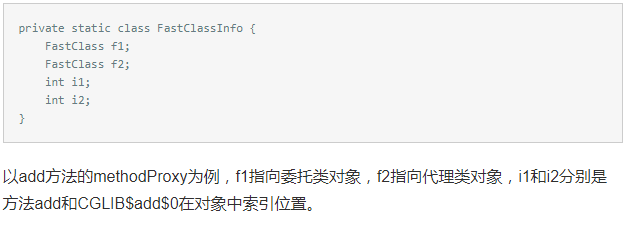
2 代理类为每个委托方法都生成两个方法。以add方法为例，一个是重写的add方法，一个是cglib$add$0方法，该方法直接调用委托类的add方法。

3 当执行代理对象的add方法时，会先判断是否存在实现了MethodInterceptor接口对象cglib$CALLBACK\_0,如果存在，则调用MethodInterceptor对象的intercept方法。



4 每个被代理的方法都对应一个MethodProxy对象。Method.invokeSuper方法最终调用委托类的add方法，实现如下：





## Java反射性能问题

错误的使用方法是每次需要获取Class对象时都使用Class.forName方法，或者需要调用Class对象上的方法时都调用。

getDeclaredMethod(String name,Class<?> ..params)或者getMethod(String name,Class<?> .. params)方法获取method对象，再调用其上的invoke(Object obj,Object.. args)方法。

这里有两个容易造成性能损耗的地方。

Class.forName调用会在整个类路径下搜索，频繁调用影响性能。

Class对象上的getDeclaredMethod (String, Class<?>...)或getMethod(String, Class<?>...)方法的调用会执行Method对象在Class对象上的搜索。有时还使用getMethods()方法获取Method数组，然后执行搜索任务，实际上getMethods()还会执行方法对象的集体Copy比直接使用(String, Class<?>...)或getMethod(String, Class<?>...)方法还要消耗时间及空间。

java反射是要解析字节码，将内存中的对象进行解析，包括了一些动态类型，所以JVM无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多！  
  
反射：  
JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制

反射是在运行期进行调用的，这样编译器就无法对其进行一些优化操作。

## Java反射优化

使用缓存。将需要反射的中间件存储下来。不管使用什么方法获取Class对象上的Method对象，返回的都是Method对象的copy对象。将这些对象缓存起来，下次使用。

对method field constructor对象做缓存

JVM对java反射做了一层优化。在jdk1.4之前，method.invoke方法的调用需要经过一个JNI的本地调用。每一次调用至少需要两次传输：java—native—java。运行时虚拟机需要解析方法签名、验证传参的类型是否正确。这些过程导致java反射调用比较慢。

在jdk1.4之后，method.invoke方法如果调用次数超过15次，那么就会使用动态字节码生成技术。会生成一个特殊的java类用于调用给定的方法。这个类实现了MethodAccessor接口。

动态字节码生成技术速度更快的原因：1 不会有JNI本地调用的瓶颈

2 不需要每次调用都去解析方法签名，因为通过反射调用的方法都有自己的MethodAccessor

3 可以继续优化等。这些MethodAccessors可以被JIT优化，比如内联优化、常量传输等。

## Cglib FastClass机制

Jdk动态代理的拦截对象是通过反射机制来调用被拦截方法，反射效率比较低。

Cglib采用了FastClass机制来实现对拦截方法的调用。FastClass机制就是对一个类的方法建立索引，通过索引来直接调用相应的方法，避免了反射调用，这就是fast。

例子：

1 定义原类

Class Test {

Public void f() {

System.out.println(“f method”);

}

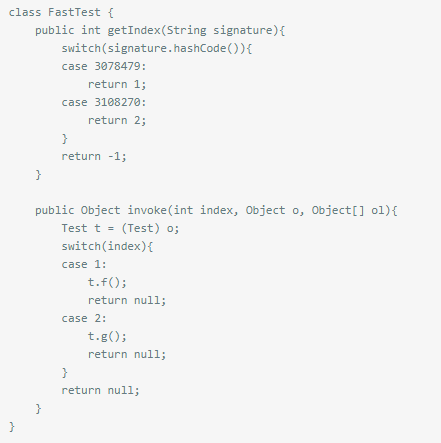
Public void g(){

System.out.println(“g method”);

}

}

2 定义fast类



在FastTest中有两个方法，getIndex会对Test类的每个方法根据hash建立索引，invoke根据指定的索引直接调用目标方法，避免了反射调用，所以当调用methodProx.invokeSuper方法时，实际上是调用代理类的CGLB$add$0方法，CGLIB$add$0直接调用了委托类的add方法。

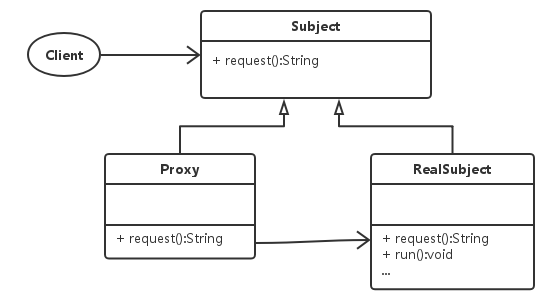
## Spring AOP为什么用两种代理方式？

基于反射机制的java动态代理只能代理实现某个接口的类，有一定的局限性。

而Cglib动态代理需要导入ASM包，对于简单的有接口的代理使用JDK动态代理，这样可以少导入一个包。

## 静态代理和动态代理

静态代理：比如黄牛就相当于火车站的代理。在代码实现中相当于为一个委托对象realSubject提供一个代理对象proxy，通过proxy调用realSubject的部分功能，并添加一些额外的业务处理，同时可以屏蔽realSubject中未开放的接口。



1 realSubject是委托类，proxy是代理类

2 Subject是委托类和代理类的接口

3 request()是委托类和代理类的共同方法

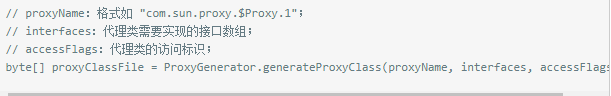
静态代理中，一个委托类对应一个代理类，代理类在编译期间就已经确定了。

动态代理：代理类并不是在java代码中实现，而是在运行期生成。相比静态代理，动态代理可以很方便的对委托类的方法进行统一的处理，如添加方法调用次数 添加日志功能。

动态代理分为jdk动态代理 cglib动态代理。

代理对象的生成过程是由Proxy类的newProxyInstance方法实现。分为3个步骤：

1 ProxyGenerator.generateProxyClass方法负责生成代理类的字节码



2 native方法Proxy.defineClass0负责字节码加载的实现，并返回对应的class对象



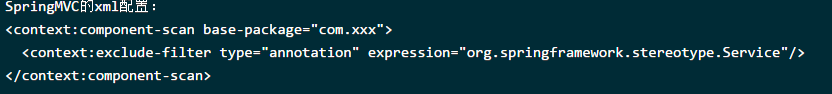
3 利用clazz.newInstance反射机制生成代理类的对象。

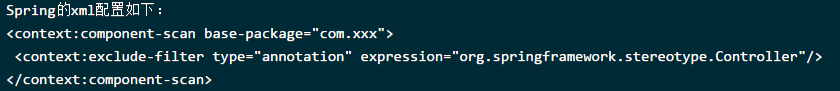
## Spring AOP不生效

（详解见spring事务文档）

排查过程： 1 查看配置 2 断点调试：spring源码断点调试，在调用controller方法时，controller的实例被jdk进行动态代理了 3 spring默认的是动态代理

SpringMVC和Spring整合的时候，SpringMVC的springmvc.xml文件中配置扫描包，不要包含service的注解，Spring的applicationContext.xml文件中配置扫描包时，不要包含controller的注解。





原理：Spring MVC启动时的配置文件，包含组件扫描 url映射以及设置freemarker参数，让spring不扫描带有@Service注解的类。因为springmvc.xml与applicationContext.xml不是同时加载，如果不进行这样的设置，那么spring就会将所有带@Service注解的类都扫描到容器中，等加载applicationContext.xml时，会因为容器中已经存在Service类，使得不会对Service进行代理，直接导致的结果是在applicationContext中的事务不起作用，发生异常时，无法对数据进行回滚。（在事务注册到Spring容器之前，就已经初始化了所有的bean，获取该bean就不会通过代理了，自然就没有事务）、

在spring的配置文件中不扫描controller类，因为这些类已经随容器启动，在springmvc.xml中扫描过一遍了。

另解：spring配置文件的加载顺序，先加载springmvc配置文件，再加载spring配置文件，事务一般会在spring配置文件中进行配置，如果此时在加载springmvc配置文件时，将Service层的类也注册了，但是此时事务还没加载，也就导致了后面的事务无法成功注入到Service中。所以将对Service层的扫描放在Spring配置文件或者其他文件中。

ContextLoaderListener创建的是ApplicationContext上下文，而ContextLoaderServlet创建的是WebApplicationContext，后者继承了前者，是父子关系。

不论什么bean只要存活在ApplicationContext上下文，都能被WebApplicationContext引用。

service dao层组件的xml配置应该由ContextLoaderListener加载。

controller层组件的xml配置应该由ContextLoaderServlet加载。

## Aspect切面

AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。比如，一个日志模块可以被称作日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在Spring AOP中，切面通过带有@Aspect注解的类实现。

## **在Spring AOP 中，关注点和横切关注的区别**

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

## 连接点

连接点代表一个应用程序的某个位置，在这个位置我们可以插入一个AOP切面，它实际上是个应用程序执行Spring AOP的位置。

Spring中只支持方法类型的连接点。

## 切入点

是一个或者一组连接点，通知会在这些位置执行。对连接点进行拦截的定义

## 通知

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过SpringAOP框架触发的代码段。

Spring切面可以应用五种类型的通知：

* **before**：前置通知，在一个方法执行前被调用。
* **after:**在方法执行之后调用的通知，无论方法执行是否成功。
* **after-returning:**仅当方法成功完成后执行的通知。
* **after-throwing:**在方法抛出异常退出时执行的通知。
* **around:**在方法执行之前和之后调用的通知。

前置、后置、异常、最终、环绕通知五类

spring aop对目标对象的增强实际上是通过拦截器完成的。

## 织入

将切面应用到目标对象并导致代理对象创建的过程。织入可以在编译 加载 或者 运行时完成。

在AOP中，有一个目标类，就是我们要调用的类，或者说是在这个类的哪些方法上执行增强的类。 增强类，需要实现AOP前置 后置 环绕 异常 最终通知，就是定义了一系列增强的功能。 测试类：要为其进行代理，所以要有AOP的代理工厂。

流程：当调用AOP目标类中某个方法时，spring容器会为目标类生成一个代理类，（jdk代理，该类必须实现了接口）（CGLIB代理，代理类中融合了原类和增强类），当访问目标类的方法时，会先访问增强类，然后再到目标类。

## 引入

在不修改代码的前提下，引入可以在**运行期**为类动态地添加一些方法或字段

## 目标对象

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（advised）对象。

## 代理

代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。

## **有几种不同类型的自动代理**

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

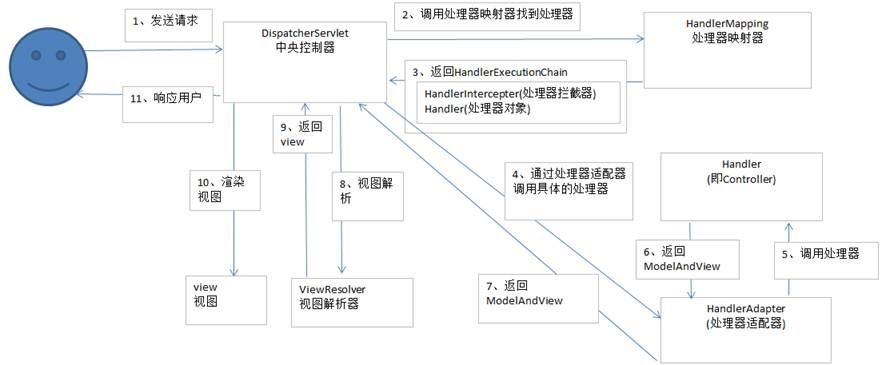
Metadata autoproxying

# Springmvc

## **什么是Spring的MVC框架？**

Spring 配备构建Web 应用的全功能MVC框架。Spring可以很便捷地和其他MVC框架集成，如Struts，Spring 的MVC框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

## 简述springmvc的工作流程



1 用户发送请求到达前端控制器DispatcherServlet

2 前端控制器收到请求后，会调用处理器映射器HandlerMapping

3 处理器映射器找到具体的处理器handler，生成处理器对象及处理器拦截器（如果有则生成），返回给前端控制器

4 前端控制器调用处理器适配器HandlerAdapter（也叫后端控制器 Controller）去执行handler

5 controller执行完毕之后，给适配器返回ModelAndView

6 处理器适配器将ModelAndView返回给前端 控制器

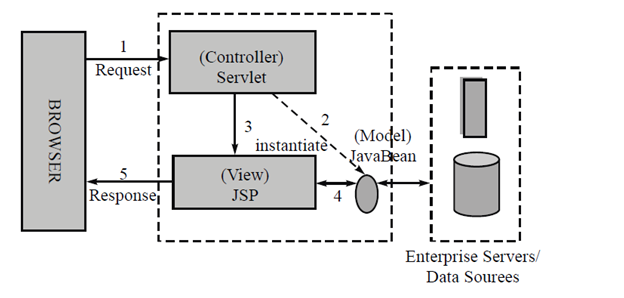
7 前端控制器将ModelAndView传给视图解析器（ViewReslover）进行解析（根据逻辑视图名解析成真正的视图jsp）

8 视图解析器解析完毕后向前端控制器返回具体的view

9 DispatcherServlet根据view进行视图渲染（即将模型数据填充到request域中）

10 前端控制器向用户返回结果

## MVC架构模式



MVC架构将应用程序的输入 处理和输出分开。三个核心部件：模型 视图 控制器

视图（view）：与用户交互的界面。MVC允许应用程序可以处理很多不同的视图。

通常，视图是依据模型的数据来建立的。

模型（Model）：表示数据和业务规则。模型与数据格式无关，这样一个模型能为多个视图提供数据。应用模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性。

应用程序中用于处理数据逻辑的部分，通常负责在数据库中存取数据。

控制器（Controller）：接受用户的输入并调用模型和视图完成用户的需求。当点击链接或者提交表单时，控制器本身不输出任何东西或者做任何处理。它只是接收请求并决定调用哪个模型构件去处理请求，然后确定用哪个视图来显示模型处理返回的数据。

通常负责从视图中读取数据，控制用户输入，并向模型发送数据。

MVC处理过程：控制器接受用户的请求，并决定应该调用哪个模型来处理数据，然后模型用业务逻辑来处理用户的请求并返回数据，最后控制器用相应的视图格式化模型返回的数据，并通过表现层呈现给用户。

MVC架构优点：

1 采用三层架构，解耦，降低了层与层之间的依赖

2 开发人员只需要去关注整个架构中的其中某一层

3 利于代码的复用

MVC架构缺点：

1 增加了系统结构和实现的复杂性。对于简单的界面，严格遵循MVC，使模型 视图与控制器分离，会增加架构的复杂性，并可能产生过多的更新操作，降低运行效率

2 视图与控制器之间过于紧密的连接。视图与控制是相互分离，但是确实联系紧密的部件。视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。

3 视图对模型数据的低效率的访问。依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对于未变化数据的不必要的频繁访问，也将损害操作性能。

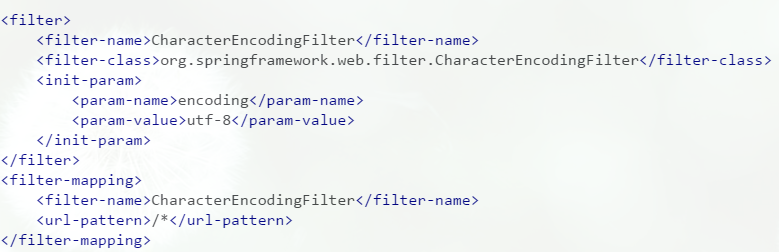
## MVC设计模式？框架？

框架通常是代码重用，而设计模型是设计重用。架构介于两者之间，部分代码重用，部分设计重用。设计模式是对在某种环境中反复出现的问题以及解决该问题的方案的描述，它比框架更抽象。

## 如何解决POST GET请求中文乱码问题

对于post请求：

在web.xml中，进行过滤器配置：CharacterEncodingFilter



对于get请求：

1 修改tomcat配置文件 编码与工程中编码一致



2 对参数进行重新编码



ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码

## Spingmvc和struts2的区别

1 springmvc的入口是一个servlet即前端控制器，而struts2的入口是一个filter过滤器

2 springmvc是基于方法开发的，是方法级别的拦截，一个方法对应于request上下文，方法跟url对应，传递参数是通过方法的形参，可以单例或多例。strust2是基于类开发，是类级别的拦截，一个类对应于request上下文，传递参数是通过类的属性（模型驱动和属性驱动），只能设计为多例。

所以从架构本身上springmvc更加容易实现restful api。因为struts2中的一个方法可以对应一个url，但是其类属性是被所有方法共享的。这也就无法使用注解或者其他方式标识其所属的方法了。

3 struts2采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器将request对象的内容进行解析成方法形参，将响应数据和页面封装成ModelAndView对象，最后将模型数据通过request对象传输到页面上。Jsp页面上默认使用jstl。

4 springmvc中集成了AJAX，使用比较方便，只需使用注解@RequestBody和@ResponseBody。

而struts2拦截器集成了AJAX，在action中处理时一般必须安装插件或者自己写代码集成进去，不方便。

5 Struts2更加符合OOP的编程思想，springmvc是在servlet上扩展。

## **DispatcherServlet**

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

## **WebApplicationContext**

WebApplicationContext 继承了ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

## **Spring MVC框架的控制器**

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

## **@Controller 注解**

该注解表明该类扮演控制器的角色，Spring不需要你继承任何其他控制器基类或引用Servlet API。

## **@RequestMapping 注解**

该注解是用来映射一个URL到一个类或一个特定的方处理法上。