注:

NoteDate:2016-02-13

Idea:2016.1

JDK:1.7.0\_79

参考书籍:Spring in action 4

# (一)、Spring 整体介绍

## 1.Spring的核心DI和AOP：

基于POJO的轻量级和最小侵入性编程；

通过依赖注入和面向接口实现松耦合；

基于切面和惯例进行声明式编程；

通过切面和模板减少样板式代码。

## 2.Spring上下文四种加载方式:

AnnotationConfigApplicationContext：从一个或多个基于Java的配置类中加载

AnnotationConfigWebApplicationContext：从一个或多个基于Java的配置类中加载

ClassPathXmlApplicationContext：从类路径下的一个或多个XML配置文件中加载

FileSystemXmlapplicationcontext：从文件系统下的一个或多个XML配置文件中加载

## 3.Spring bean的生命周期

1．Spring对bean进行实例化；

2．Spring将值和bean的引用注入到bean对应的属性中；

3．如果bean实现了BeanNameAware接口，Spring将bean的ID传递给setBean-Name()方法；

4．如果bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()方法，将

BeanFactory容器实例传入；

5．如果bean实现了ApplicationContextAware接口，Spring将调

用setApplicationContext()方法，将bean所在的应用上下文的引用传入进来；

6．如果bean实现了BeanPostProcessor接口，Spring将调用它们的post-

ProcessBeforeInitialization()方法；

7．如果bean实现了InitializingBean接口，Spring将调用它们的after-

PropertiesSet()方法。类似地，如果bean使用init-method声明了初始化方法，该方法

也会被调用；

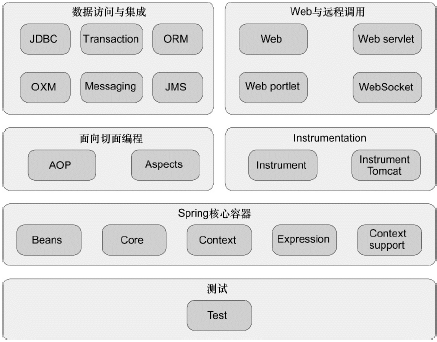
8．如果bean实现了BeanPostProcessor接口，Spring将调用它们的post-

ProcessAfterInitialization()方法；

9．此时，bean已经准备就绪，可以被应用程序使用了，它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．如果bean实现了DisposableBean接口，Spring将调用它的destroy()接口方法。同样，如果bean使用destroy-method声明了销毁方法，该方法也会被调用。

## 4.Spring模块划分



# (二)Spring Bean的装配方式

## 1.自动扫描装配

配合是用Bean定义注解(如:Controller、Component、Service、Repository等)与ComponentScan配合对上述注解定义的Bean进行装配。进行注入的时候使用@Autowired来对上下文中的Bean进行注入

## 2.配置类中@Bean

在配置类中使用@Bean定义方法返回一个Bean，并且会将这个返回的Bean配置到上下文中。例：

@Bean(name = Bus.DEFAULT\_BUS\_ID)  
public SpringBus springBus() {  
 return new SpringBus();  
}

这种方式配置Bean的时候需要完成注入例：

@Bean//将disc注入到CDPlayer中

public CDPlayer cdPlayer() {  
 return new CDPlayer (disc());  
}

## 3.Xml配置:

这种配置方式比较老套,不建议在新的项目中使用

# (三)Spring高级装配

Spring提供一些高级装配方式,如Profile、条件Bean、Bean的作用域、运行时值注入等高级特性。这些高级特性使得Spring的扩展性强，并且Spring boot 中就是使用了这些新特性。进一步简化了JAVA项目的开发。具体情况看学习项目中springAdvancedFeatures类目。下面笔记一些比较重要的新特性。

## 条件注解

条件注解是4.0之后一个重要的新特性,可以根据条件类来判断是否生成该Bean.后面一些新功能,好多都是基于这个特性来实现的.

第一步:

创建条件类.条件类要时间Spring的Condition接口,然后实现matches方法,如果方法返回true,那么指定该条件类的Bean则会在上下文初始化的时候被创建,如果该方法返回的false,那么指定该条件类的Bean则不被创建.

public class LinuxCondition implements Condition{  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 return context.getEnvironment().getProperty("os.name").contains("Linux");  
 }  
}

第二步:

给需要条件创建的Bean指定条件类

@Component  
@Conditional(LinuxCondition.class)  
class LinuxBean implements ConditionServiceInterface{  
 public String showBeanName() {  
 return "Linux";  
 }  
}

Condition接口方法中我们可以通过ConditionContext和AnnotatedTypeMetadata来获取获取一些系统属性.

通过ConditionContext可以获取如下信息:

借助getRegistry()返回的BeanDefinitionRegistry检查bean定义；

借助getBeanFactory()返回的ConfigurableListableBeanFactory检查bean是

否存在，甚至探查bean的属性；

借助getEnvironment()返回的Environment检查环境变量是否存在以及它的值是

什么；

读取并探查getResourceLoader()返回的ResourceLoader所加载的资

## 处理自动装配中的发生歧义的问题.

例如一个接口类有多个实现类,或者在条件Bean中有多个类满足了条件Bean的Condition.在这种情况下进行装配的时候会出现下面异常

No qualifying bean of type [study.springAdvancedFeatures.ConditionServiceInterface] is defined: expected single matching bean but found 2: linuxBean,windowsBean.

如果存在这种的可能,需要在Bean定义的时候添加注解@Primary来指定优先Bean.或者在注入的时候使用Qualifier来指定注入的Bean.的优先级

## Bean的作用域

在默认的情况下Spring应用上下文中的Bean的作用域都是以单例的形式创建的.也就是说,不管给定的Bean被注入到其他Bean多少次,每次注入的都是同一个实例.

Spring定义了多种作用域,可以基于这些作用域创建Bean,包括:

1. 单例(singleton):在这个应用中只会创建Bean的一个实例.
2. 原型(prototype):每次注入或者通过Spring上下文获取的时候都会创建一个新的Bean实例
3. 会话:Web应用中,为每个会话创建一个bean实例.

@Scope(value = WebApplicationContext.*SCOPE\_SESSION*,

proxyMode = ScopedProxyMode.*INTERFACES*)

1. 请求:Web应用中,为每个请求创建一个Bean实例

@Scope(value = WebApplicationContext.*SCOPE\_REQUEST*,

proxyMode = ScopedProxyMode.*INTERFACES*)

在两个Web应用作用域中的设置中有一个属性是proxyMode,这个属性如果使用的是ScopedProxyMdoe.INTERFACE的话,是十分理想的代理模式,这样的话解决了一个问题就是:会话作用域注入到单例的Bean中的时候出现的问题,因为单例Bean在上下文初始化的时候已经创建好了,但是Web作用域的Bean在没有请Web请求的时候是不会创建Bean的,在这种情况下Spring会将代理注入到单例的Bean中,这个代理会暴露与实例一样的方法.当调用单例Bean中调用代理的方法的时候,代理会对其进行懒解析,并将调用委托给作用域内真正的Bean.

## 运行时注入

Spring的装配一部分是为Bean之间进行依赖注入,另一方面是为Bean的属性进行值的注入.为了实现非硬编码的方式为Bean注入值,Spring提供了两种在运行时注入值的方式:

1. 属性占位符(Property placeholder)
2. Spring表达式语言(SpEL)

SpEL拥有很多特性,包括

使用Bean的ID来引用Bean;调用方法和访问对象的属性;

对值进行算术、关系和逻辑运算；正则表达式匹配；

集合操作。

# (四)Spring AOP

## 1.面向切面的编程的关键术语

通知：通知是指切面完成的工作目标，有五种通知

前置通知（Before）：在目标方法被调用之前调用通知功能；

后置通知（After）：在目标方法完成之后调用通知，

返回通知（After-returning）：在目标方法成功执行之后调用通知；

异常通知（After-throwing）：在目标方法抛出异常后调用通知；

环绕通知（Around）：在被通知的方法调用之前和调用之后执行自定义的行为。

切面：切面是通知和切点的结合，是什么，何时，何处完成指定功能。

织入：Spring是在运行期织入的，AOP容器会为目标对象动态地创建一个代理对象。

## 2.Spring对AOP的支持

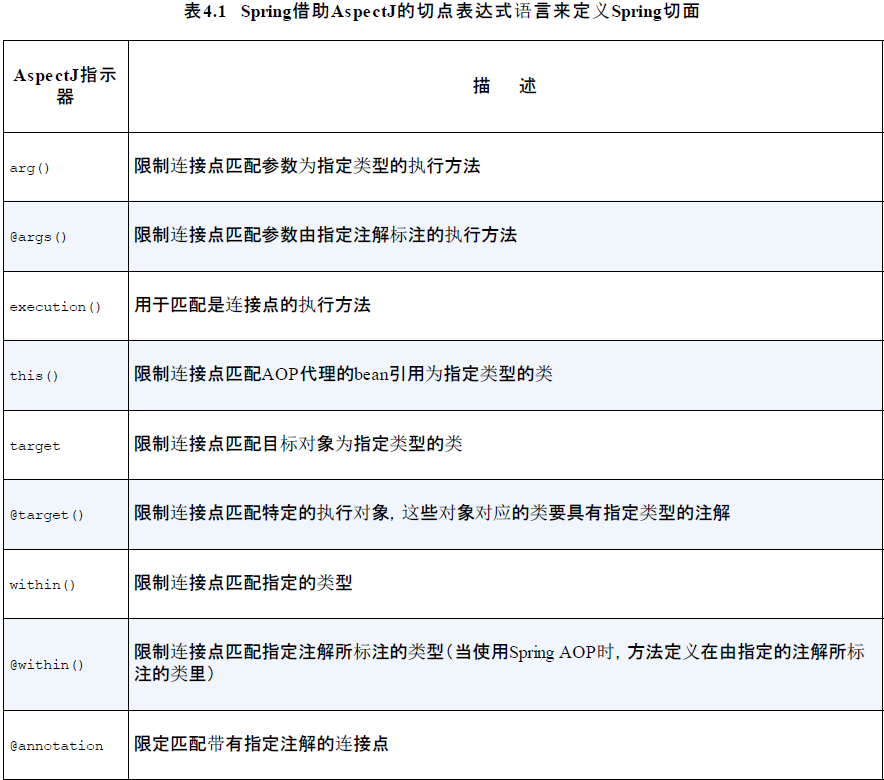
Spring AOP基于动态代理,所有Spring只支持方法连接点.通过在代理类中包裹切面,Spring在运行期把切面织入到Spring管理的Bean中.当代理拦截到方法调用时,在调用目标Bean方法前会执行切面逻辑.知道应用需要被代理的Bean时,Spring才创建代理对象.

Spring提供了四种类型的AOP

1. 基于代理的经典Spring AOP
2. 纯POJO切面
3. @AspectJ注解驱动的切面
4. 注入式AspectJ切面(适用于Spring各个版本)

## 3.通过切点来选择连接点

Spring借助AspectJ的切点表达式语言来定义Spring切面.



在上述指示器中,只有execution指示器是实际执行匹配的,而其他的指示器都是用来限制匹配的.因此上面的execution指示是最重要的.

Spring使用AspectJ注解来声明通知方法



## 4.编写切点

@Around("execution(\* study.springAdvancedFeatures.aop.Performance.perform(..))")  
public void performancePointCut(ProceedingJoinPoint jp) {  
 try {  
 System.*out*.println("@Around-beforePerformance");  
 jp.proceed();//如果这个方法不运行的话被通知的方法会被阻塞  
 System.*out*.println("@Around-afterPerformance");  
 } catch (Throwable e) {  
 System.*out*.println("@Around-PerformanceFault");  
 }  
}

# (五)事务管理

## 1.spring对事务管理的支持

Spring支持编程式事务管理和声明式事务管理两种方式.

编程式事务管理时使用TransactionTemplate或者直接使用底层的PlatformTransactionManager.对于编程式事务管理,spring推荐使用TransactionTemplate.

Spring 的事务管理其实就是通过aop来实现的.其本质是对方法前后进行拦截,然后在目标方法开始之前创建或者加入一个事务,在执行完目标方法之后,根据执行情况提交或者回滚事务.

显然的,经过对比,声明式事务管理优于编程式事务管理,着也是spring提倡的非入侵式的开发方式.唯一不足的是声明式事务管理最细力度只能作用到方法级别.无法像编程式事务管理可以作用到代码块级别.

## 2.spring的事务特性

Spring的事务管理策略类都继org.springframework.transaction.platformTransactionManager

事务隔离级别:是指若干个并发的事务之间的隔离程度

事务传播行为:是指如果在当前事务之前,一个事务上下文已经存在,此时有若干选项可以指定一个事务方法的执行行为.

事务超时:一个事务所允许执行的最长时间,当超过这个时间限制,事务还没有完成,则自动回滚事务

## 3.@Transactional注解

@Transactional的属性

| **属性** | **类型** | **描述** |
| --- | --- | --- |
| value | String | 可选的限定描述符，指定使用的事务管理器 |
| propagation | enum: Propagation | 可选的事务传播行为设置 |
| isolation | enum: Isolation | 可选的事务隔离级别设置 |
| readOnly | boolean | 读写或只读事务，默认读写 |
| timeout | int (in seconds granularity) | 事务超时时间设置 |
| rollbackFor | Class对象数组，必须继承自Throwable | 导致事务回滚的异常类数组 |
| rollbackForClassName | 类名数组，必须继承自Throwable | 导致事务回滚的异常类名字数组 |
| noRollbackFor | Class对象数组，必须继承自Throwable | 不会导致事务回滚的异常类数组 |
| noRollbackForClassName | 类名数组，必须继承自Throwable | 不会导致事务回滚的异常类名字数组 |

@Transactional的用法

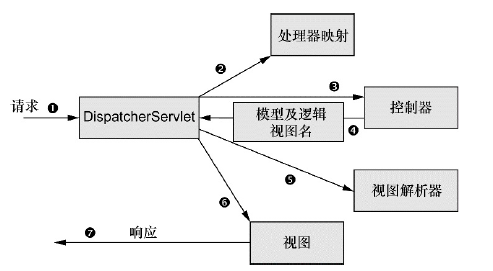
@Transactional 可以作用于接口、接口方法、类以及类方法上。当作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。

虽然 @Transactional 注解可以作用于接口、接口方法、类以及类方法上，但是 Spring 建议不要在接口或者接口方法上使用该注解，因为这只有在使用基于接口的代理时它才会生效。另外， @Transactional 注解应该只被应用到 public 方法上，这是由 Spring AOP 的本质决定的。如果你在 protected、private 或者默认可见性的方法上使用 @Transactional 注解，这将被忽略，也不会抛出任何异常。

默认情况下，只有来自外部的方法调用才会被AOP代理捕获，也就是，类内部方法调用本类内部的其他方法并不会引起事务行为，即使被调用方法使用@Transactional注解进行修饰。

# (六)Spring MVC

## 1.SpringMVC请求信息流:



由SpringMVC的请求信息流可以清晰看出,SpringMVC的所有功能都是由DIspatcherServlet来控制和管理的.

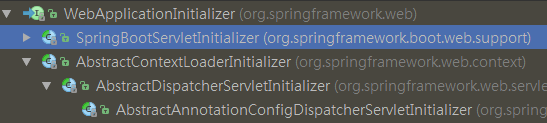
## 2.将DispatcherServlet部署到Servlet容器

按照传统的Servlet规范中,像SpringMVC中的DispatcherServlet这样的Servlet会配置在Web.xml文件.中然后再一起部署到war包下.进而让容器解析.xml文件来部署Servlet.

借助于Servlet3规范和Spring3.1的功能增强.我将可用使用java配置的方式将SpringMVC的核心Servlet-DispatcherServlet配置到容器中.而不再需要使用web.xml文件.

在Servlet 3.0环境中,容器会在类目中查找实现了ServletContainerInitializer接口的类,如果发现了这个接口的实现类,就会用它来配置Servlet容器.

而Spring提供了这个ServletContainerInitializer接口的实现类SpringServletContainerInitializer,这个实现类,反过来会在类路径中查找实现WebApplicationInitializer的类,并将配置的任务交给这些类来完成.Spring3.2为了方便用户使用引用了一个便利的实现AbstractAnnotationConfigDispatcherServletInitializer这个基础配置类.我们可以编写我们的配置类的时候继承这个Spring编写好的基础配置类.



从这个基础配置类中可以看出这个基础类多重继承了AbstractDispatcherServletInitializer和AbstractContextLoaderinitializer.然后观察AbstractDispatcherServletInitializer的onStartup方法后会发现它在配置DispatcherServlet之前会启动ContextLoaderinitializer应用上下文,所以它会自动地配置DispatcherServlet和Spring应用上下文,Spring上下文会位于应用程序的Servlet上下文之中.

当DispatcherServlet启动的时候它会创建Spring应用上下文,并且加载配置类中所声明的Bean.一般情况下我们希望DispatcherServlet加载包含Web组件的bean,如控制器、视图解析器已经处理器映射等.

但是在Spring Web应用中,通常还会有另外一个应用上下文,这个上下文是由ContextLoaderListener创建的,而这些Bean通常是驱动应用后端的中间层和数据层组件

例子:继承AbstractAnnotationConfigDispatcherServletInitializer并且重写三个抽象方法

//由ContextLoaderListener加载应用上下文  
protected Class<?>[] getRootConfigClasses() {  
 return new Class<?>[] {SpringAdvancedFeaturesContext.class};  
}  
//加载Web相关配置文件,并且引入Web组件,如控制器、视图解析器、映射  
protected Class<?>[] getServletConfigClasses() {  
 return new Class<?>[] {SpringMVCContext.class};  
}  
//指定DispatcherServlet的映射地址  
protected String[] getServletMappings() {  
 return new String[] {"/"};  
}

## 3.额外配置

1.对DispatcherServlet进行额外配置.在AbstractAnnotationConfigDispatcherServletInitializer将DispatcherServlet注册到Servlet容器后,会调用customizeRegistration(),并将Servlet注册后得到的Registration.Dynamic registration传递进来.通过重写customizeRegistration()方法,我们可以对DispatcherServlet进行额外的配置.

2.对Web容器进行额外的Servlet和Filter配置.当我们想往Web容器注册其他组件.可以配置一个新的初始化器就可以的.最简单的方式就是实现SpringWebApplicationInitializer接口.

# (七)Spring Security

## 1.介绍:

Spring Security充分利用DI、AOP和Servlet规范中的Filter能够在Web请求级别和方法调用级别处理身份认证和授权。

## 2.特性

Require authentication to every URL in your application

Generate a login form for you

Allow the user with the **Username** *user* and the **Password** *password* to authenticate with form based authentication

Allow the user to logout

[CSRF attack](https://en.wikipedia.org/wiki/Cross-site_request_forgery) prevention

[Session Fixation](https://en.wikipedia.org/wiki/Session_fixation) protection

Security Header integration

[HTTP Strict Transport Security](https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security) for secure requests

[X-Content-Type-Options](https://msdn.microsoft.com/en-us/library/ie/gg622941(v=vs.85).aspx) integration

Cache Control (can be overridden later by your application to allow caching of your static resources)

[X-XSS-Protection](https://msdn.microsoft.com/en-us/library/dd565647(v=vs.85).aspx) integration

X-Frame-Options integration to help prevent [Clickjacking](https://en.wikipedia.org/wiki/Clickjacking" \t "_top)

Integrate with the following Servlet API methods

[HttpServletRequest#getRemoteUser()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getRemoteUser())

[HttpServletRequest.html#getUserPrincipal()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getUserPrincipal())

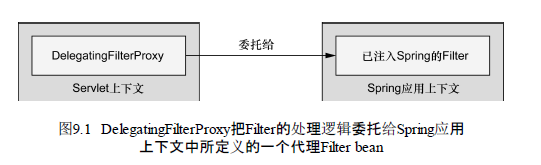
[HttpServletRequest.html#isUserInRole(java.lang.String)](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#isUserInRole(java.lang.String))

[HttpServletRequest.html#login(java.lang.String, java.lang.String)](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#login(java.lang.String,%20java.lang.String))

[HttpServletRequest.html#logout()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#logout())

## 3.快速配置和原理

由于Spring Security 是借助一些列的Servlet Filter来提供各种安全功能.Spring提供了一个基于代理模式的特殊Servlet Filter:DelegatingFilterProxy. 这个代理类,把Filter的处理逻辑委托为Spring应用上下文中所定义的一个代理Filter bean.

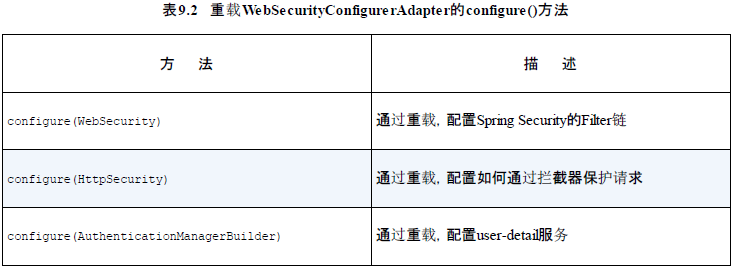


在使用这些安全功能的时候我们不需要显式的声明任何Filter来提供安全功能.可以引用Spring的Java配置方案.添加@EnableWebSecurity注解来启动Web安全功能.并且需要配置在一个实现了WebSecurityConfigurer或者继承了WebSecurityConfigurerAdapter的Bean中.

@Configuration  
@EnableWebSecurity  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

}

当我们的配置是继承了ConfigureAdapter来实现的时候,我们可以重载这个父类的三个configure()方法来配置Web安全性.



## 4.配置用户存储信息

常用的四中用户存储方式有

1.In-Memory Authentication : 配置内存用户存储

2.JDBC Authentication : 通过连接数据库进行用户存储

3.LDAP Authentication : 通过连接微软LDAP服务器进行用户存储

4.UserDetailsService : 通过service层的服务获取用户存储