# 第一章















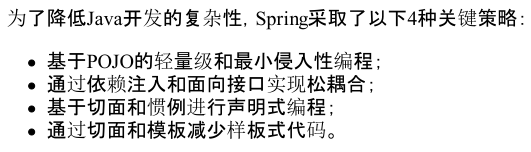












* 为了降低Java开发的复杂性，Spring采取了以下4种关键策略：

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。



引用组件-javaBean





DI的优点：

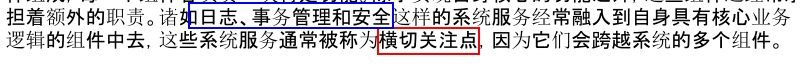
@解耦合：依赖者并不知道被依赖者是一个什么具体的类型，只有在运行时才知道真正依赖的对象是哪一个具体类型的对象。

@便于测试:mock;

:负责对象的创建和组装（wiring）

* Spring通过应用上下文（Application Context）**装载bean的定义并把它们组装起来**。Spring应用上下文全权**负责对象的创建和组装**。Spring自带了多种应用上下文的实现，它们之间主要的区别仅仅在于如何加载配置。





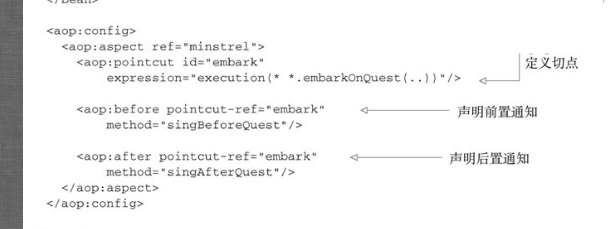


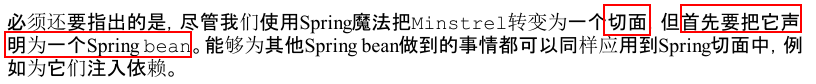


横切面：增强的那些方法写在哪些类里面。（即包含前置后值通知的类）

切点：对哪些方法进行增强（即那些需要添加前置后置通知的方法。）







上图的意思是：切面也需要交给spring来管理。

:使用jdbc进行查询的时候，就会使用样板式代码。

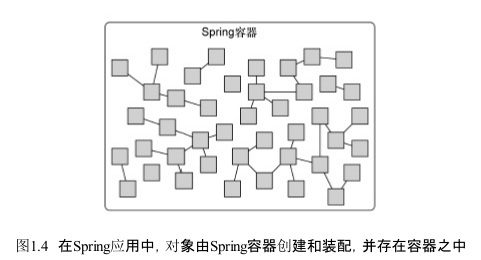


* JdbcTemplate内部封装的是原始jdbc,所以可见它的的crud的速度是相对是比较快的。

### Spring容器

* 定义

：应用中所有bean驻留的地方。



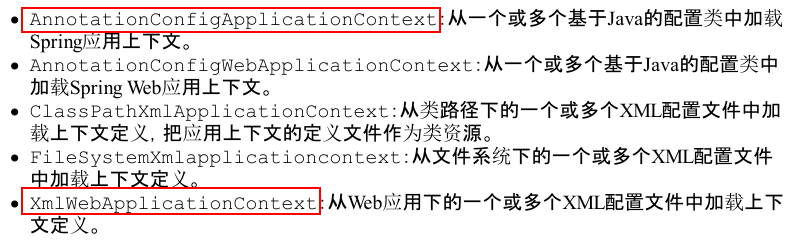
创建，装配，配置，管理，生命周期



* 容器类型

-beanFactory:支持基本的DI

-ApplicationContext

-

* 联系：

ApplicationContext基于BeanFactory构建

* 区别：FileSystemXml…（F）和 ClassPathXml…(C)

FileSystemXml…（F）和 ClassPathXml…(C)的区别在于：

F：会在指定的文件系统中查找xml文件。

C:会在类路径（**包含jar文件**）查找xml文件。



疑问：只能查看xml文件？是不是意味着jar文件下的属性文件也可以使用C？如何读取jar文件中的属性文件?

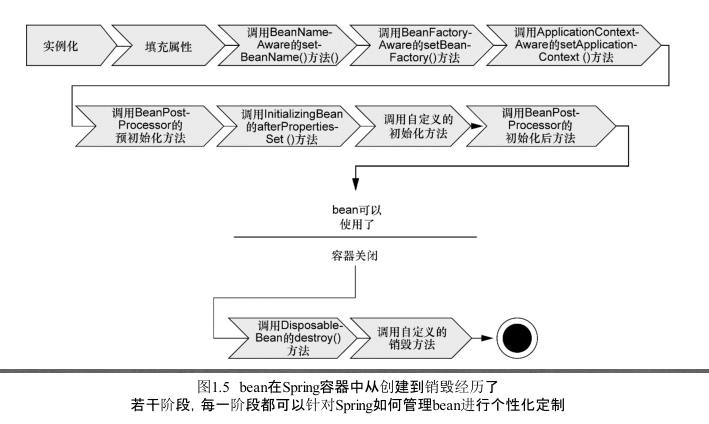
* applicationContext相关文档：

<http://www.cnblogs.com/TTTTT/p/6675557.html> 好文档

* classpathApplicatonContext加载多个文件的写法：详见md

### bean的生命周期

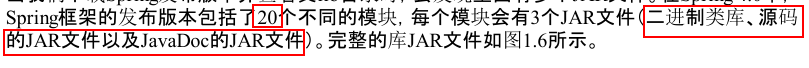
没看懂。。。。。。



疑问：aware，装饰模式？？？

链接：<http://blog.csdn.net/a19881029/article/details/8980503> 装饰模式讲解

### Spring模块







联系：

@所有模块的构建都是基于核心容器的。

@AOP模块：切面基础。

@数据访问和集成





那些更喜欢ORM（Object-Relational Mapping）工具而不愿意直接使用JDBC的开发者,Spring提供了ORM模块。**Spring的ORM模块建立在对DAO的支持之上**，并为多个ORM框架提供了一种构建DAO的简便方式。Spring没有尝试去创建自己的ORM解决方案，而是对许多流行的**ORM框架**进行了集成，包括Hibernate、Java Persisternce API、Java Data Object和iBATIS SQL Maps。**Spring的事务管理支持所有的ORM框架以及JDBC**



数据访问和集成模块同样包含了在JMS（Java Message Service）之上构建的Spring抽象层，它会**使用消息以异步的方式与其他应用集成**。从Spring 3.0开始，本模块还包含**对象到XML映射**的特性



@ Web与远程调用



**MVC框架**：Apache的Struts、JSF、WebWork和Tapestry

Spring远程调用功能集成了RMI（Remote Method Invocation）、Hessian、Burlap、JAX-WS，同时Spring还自带了一个远程调用框架：HTTP invoker。Spring还提供了暴露和使用REST API的良好支持。

@Instrumentation（不用关注）

Spring的Instrumentation模块提供了为JVM添加代理（agent）的功能。具体来讲，它为Tomcat提供

了一个织入代理，能够为Tomcat传递类文件，就像这些文件是被类加载器加载的一样

@测试

Spring为使用JNDI、Servlet和Portlet编写单元测试提供了一系列的**mock**对象实现

疑问：JNDI????

### Spring Portfolio

@Spring Web Flow

基于流程的会话式web应用（购物车）

访问Spring Web Flow的主页（<http://projects.spring.io/spring-webflow/>）

@Spring Web Service

虽然核心的Spring框架提供了将Spring bean以声明的方式发布为Web Service的功能，但是这些

服务是基于一个具有争议性的架构（拙劣的契约后置模型）之上而构建的。这些服务的契约由bean的接口来决定。 Spring Web Service提供了契约优先的Web Service模型，服务的实现都是

为了满足服务的契约而编写。

可以浏览站点[http://docs.spring.io/spring- ws/site/](http://docs.spring.io/spring-%20ws/site/)

@ Spring Security

<http://projects.spring.io/spring-security/>

@ Spring Integration（整合）

Spring Integration提供了多种通用应用集成模式的Spring声明式风格实现。

<http://projects.spring.io/spring-integration/>

我推荐Mark Fisher、Jonas Partner、Marius Bogoevici和Iwein Fuld编写的《Spring Integration

in Action》（Manning，2012，[www.manning.com/fisher/](http://www.manning.com/fisher/)）；

@Spring Batch

当我们需要**对数据进行大量操作**时，没有任何技术可以比批处理更胜任这种场景。如果需要开发一个批处理应用，你可以通过Spring Batch，使用Spring强大的面向POJO的编程模型。

Arnaud Cogoluegnes、Thierry Templier、Gary Gregory和Olivier Bazoud编写的《Spring Batch in Action》（Manning，2012，[www.manning.com/templier/](http://www.manning.com/templier/)）

<http://projects.spring.io/spring-batch/>

@Spring Data

Spring Data使得在Spring中使用任何数据库都变得非常容易。

@Spring Social

社交网络是互联网领域中新兴的一种潮流，越来越多的应用正在融入社交网络网站，例如

Facebook或者Twitter。如果对此感兴趣，你可以了解一下Spring Social，这是Spring的一个社交

网络扩展模块。不过，Spring Social并不仅仅是tweet和好友。尽管名字是这样，但Spring Social更多的是**关注连接（connect）**，而不是社交（social）。它能够帮助你通过REST API连接Spring应用，其中有些Spring应用可能原本并没有任何社交方面的功能目标。

Spring如何连接Facebook或Twitte

[https://spring.io/guides/gs/accessing- facebook/](https://spring.io/guides/gs/accessing-%20facebook/)

<https://spring.io/guides/gs/accessing-twitter/>

@Spring Mobile

移动应用是另一个引人瞩目的软件开发领域。智能手机和平板设备已成为许多用户首选的客

户端。Spring Mobile是Spring MVC新的扩展模块，用于支持移动Web应用开发。

@Spring for Android

与Spring Mobile相关的是Spring Android项目。这个新项目，旨在通过Spring框架为开发基于

Android设备的本地应用提供某些简单的支持。最初，这个项目提供了Spring RestTemplate

的一个可以用于Android应用之中的版本。它还能与Spring Social协作，使得原生应用可以通过

REST API进行社交网络的连接。

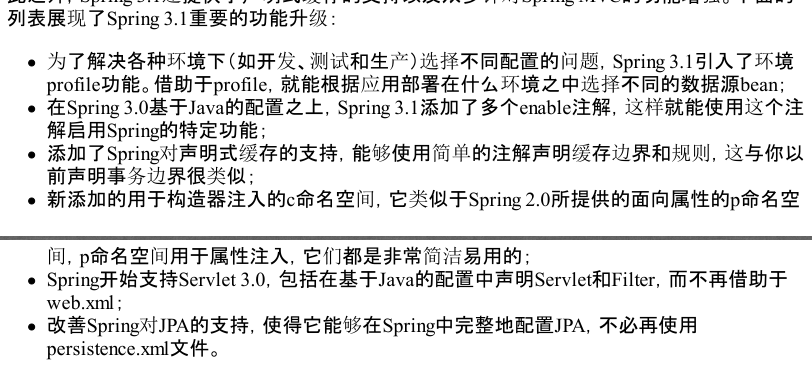
<http://projects.spring.io/spring-android/>

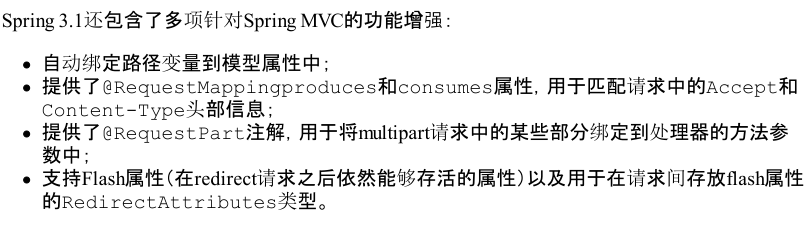
@Spring Boot

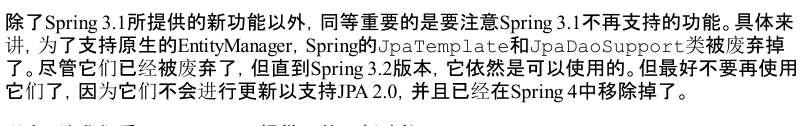
Spring极大地简化了众多的编程任务，减少甚至消除了很多样板式代码，

Spring Boot大量依赖于自动配置技术，消除大部分（在很多场景中，甚至是全部）Spring配置。它还提供了多个Starter项目，不管你使用Maven还是Gradle，这都能减少Spring工程构建文件的大小。

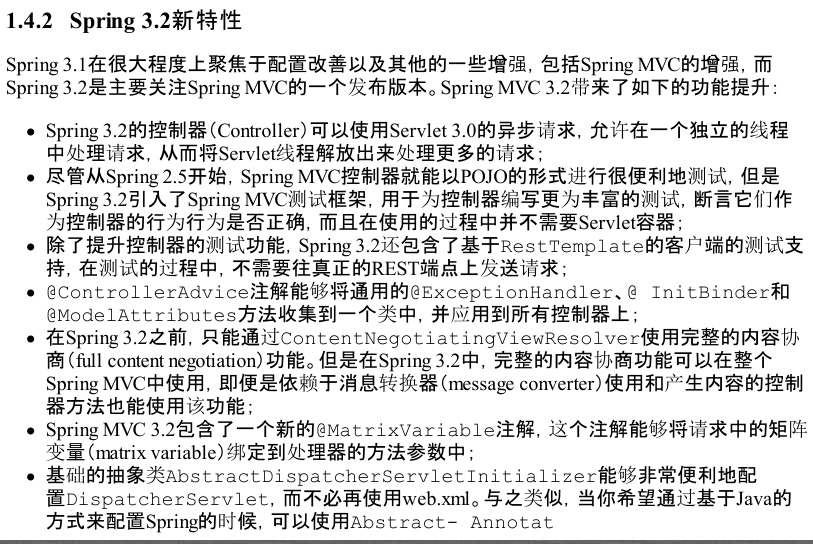
### Spring3.1新特性

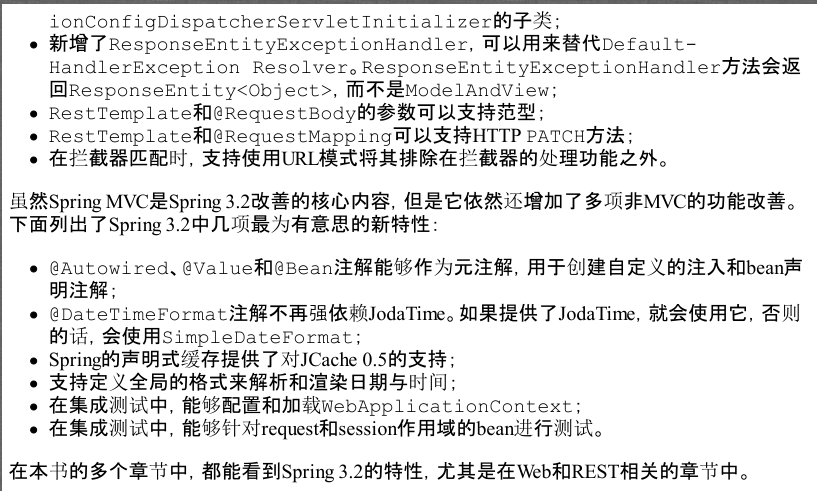




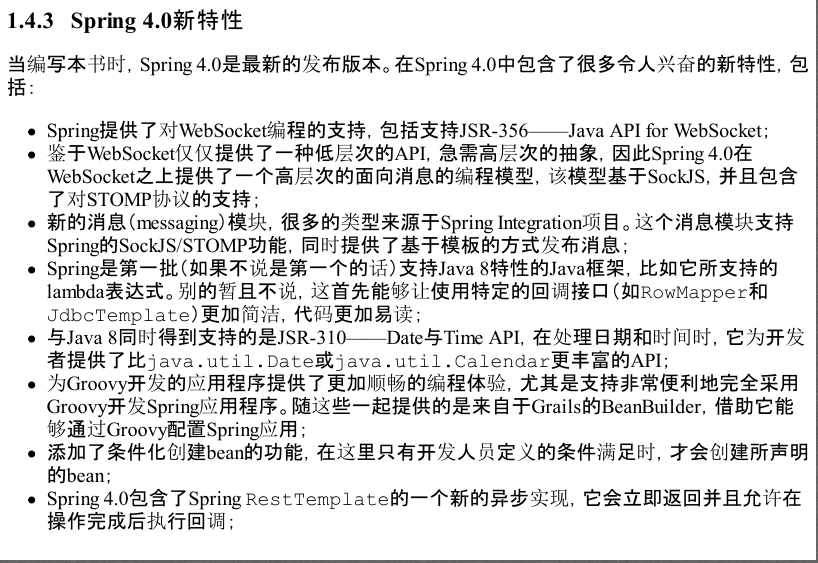


### Spring3.2新特性





### Spring4新特性



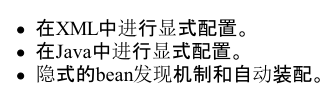
# 第二章

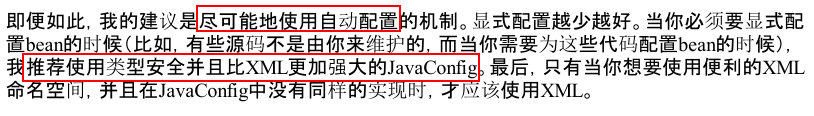
在Spring中，对象无需自己查找或创建与其所关联的其他对象。相反，容器负责把需要相互协作的对象引用赋予各个对象。



## 装配Bean

三种装配机制：





1. 将bean交给Spring:@component
2. 组件扫描：@ComponentScan；。
3. 自动装配：@autowired

重点：

1. @ComponentScan注解，这个注解能够在Spring中启用组件扫描（默认是不启用的）

2. 使用XML来启用组件扫描:<context:component-scan>

3. 如果没有其他配置的话，@ComponentScan默认会扫描与配置类相同的包。因此Spring将会扫描这个包以及这个包下的所有子包，查找带有@Component注解的类。

4.

## 自动化装配

1. @Component组件未指明name属性时候，Spring默认给bean的命名：类名首字母小写。Eg.类Student.class 默认id为student
2. @Component（“student”）等效于@Named(“student”)为bean自定义name。
3. @ComponentScan声明多个基础包

说明：第一种直接说明包路径；第二种说明包路径下的任意一个类或者接口（第一种存在类型不安全的问题。推荐第二种）

@Configuration

//@ComponentScan(basePackages={"net.shopin.charator1.bean","net.shopin.charator1.bean1"}) //声明多个基础包

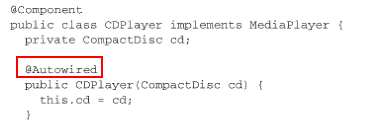
@ComponentScan(basePackageClasses={BeanScan.**class**,Bean1Scan.**class**})//作用等效于上面。只是取得是包路径下的某一个类。

**public** **class** ApplicationConfig4 {

}

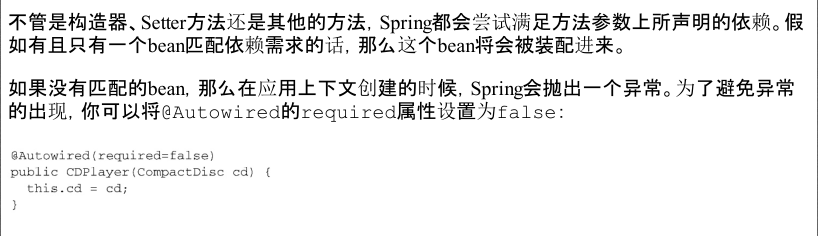
1. @autowired使用位置

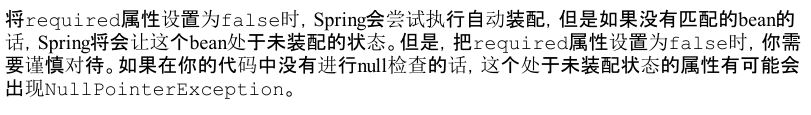
# 构造器上&&方法上：@Autowired(required=false)

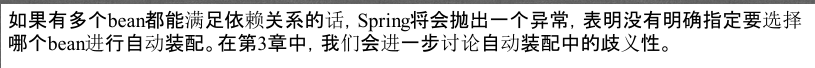


说明：一旦在方法上使用@autowired,spring回去查找方法所以来的参数bean,一旦没有，spring会报错。为了不报错，可以添加属性

Required=false;但是设置为false后，一旦spring没有找到依赖的bean，那么这个bean（参数）处于未装配状态，即null，可能会已发空指针异常。





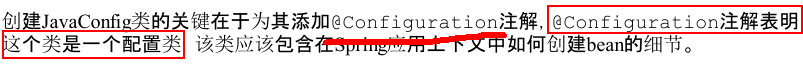


#

(JSR-330)

## JavaConfig显示配置

使用场景：当使用三方类库，不可能在三方类的源码上添加@Component,@autowired，这个时候想讲这个类个spring管理，只能通过显示配置的方式（java,xml）;java代码配置方式优点：类型安全，对重构友好.



*// 采用java配置。*

@Configuration *//声明这是一个配置类*

**public** **class** **JavaConfig** {

*//@Bean显示声明bean交给spring管理，方法返回类型即此bean的实例，默认bean的id为方法名，使用name属性修改默认bean的Id.*

@Bean(name="classicCD")

**public** CD **getClassicCD**(){

**return** **new** ClassicCD();

}

@Bean(name="cdPlayer")

**public** CDPlayer **getCDPlayer**(){

**return** **new** CDPlayer(getClassicCD());

*//注意这里：虽然调用的是方法getClassicCD（），但情况并非完全如此。因为getClassicCD ()方法上添加了@Bean注解，Spring将会拦截所有对它的调用，并确保直接返回该方法所创建的bean，而不是每次都对其进行实际的调用*

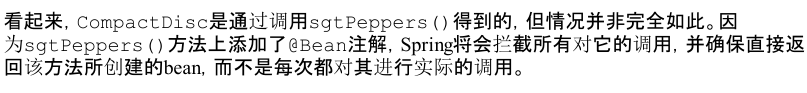
}

}

说明：

# @Bean注解，表明这个方法会创建一个bean实例，同时将其注册的spring应用的上下文中。

# 在JavaConfig中装配bean的最简单方式就是引用创建bean的方法



详情见markdown: “javaConfig的方式去将bean交给spring管理”

# 证明spring管理的bean**默认是单例**(涉及什么时候使用)

说明：CDPlayer这个类型因为声明了两次,那么在使用这个类型的时候，必须要指明id。如果只说明一次，那么在使用的时候，名称可以随便定义。

详情见markdown: “证明spring管理的bean默认是单例（singlton）[javaConfig管理的bean]”

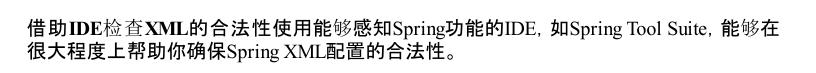
# 带有@Bean注解的方法可以采用任何必要的Java功能来产生bean实例。构造器和Setter方法只是@Bean方法的两个简单样例。这里所存在的可能性仅仅受到Java语言的限制。

## 通过XML装配bean

1. Xml配置文件声明bean，class属性使用全限定类名，如果没有指明id,默认名字就是全限定类型+“#n”.

Eg。<bean class=“net.shopin.xyy.Student”/>,默认的id为“net.shopin.xyy.Student#0”，当出现同类型的bean，是#号后数字自增。

1. 什么时候需要使用id属性：这个bean可能需要按照名称被引用时，最好给这个bean命名。



1. Xml配置的缺点：

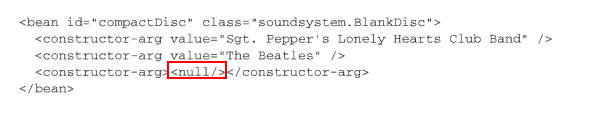
# 凡是需要给spring管理的bean都需要再xml中进行配置，很繁琐。

# 声明bean,class属性即使有误，编译时期也不会报错。Spring的XML配置并不能从编译期的类型检查中受益。

#



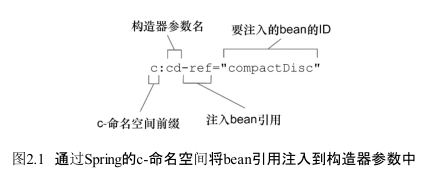
1. bean装配的方式。
   1. 配置文件（xml）
   2. JavaConfig
   3. 自动装配
2. 依赖注入的方式
   1. 构造器注入
      * 1. <constructor-arg>元素【ref,value,list,set,数组】



<null/>将null传递给构造器。

* + - 1. 使用Spring 3.0所引入的c-命名空间【value: c:\_title;c:\_0;c:\_】【ref: c:\_student-ref; c:\_0-ref;c:\_-ref.】

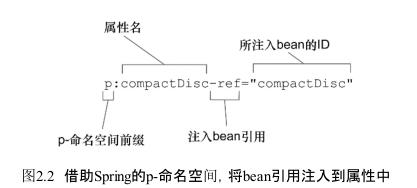
Spring为<constructor-arg>元素提供了c-命名空间作为替代方案



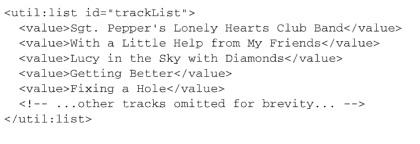
* 1. Setter（方法参数。）
     + 1. <property>



* + - 1. Spring提供了更加简洁的p-命名空间，作为<property>元素的替代方案【】



* + - 1. <util:list>



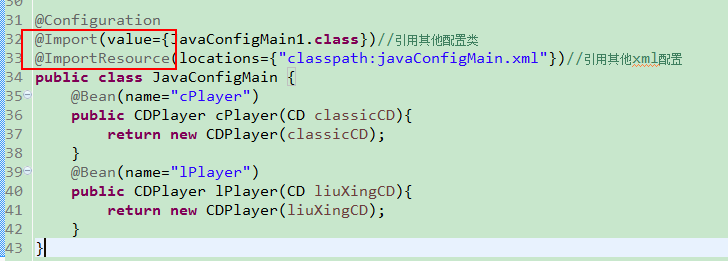


* 1. 接口

## javaConfig和xml配置混合使用

### javaConfig中使用xml配置

* + - 1. @import
      2. @importResource



### Xml配置中使用javaConfig

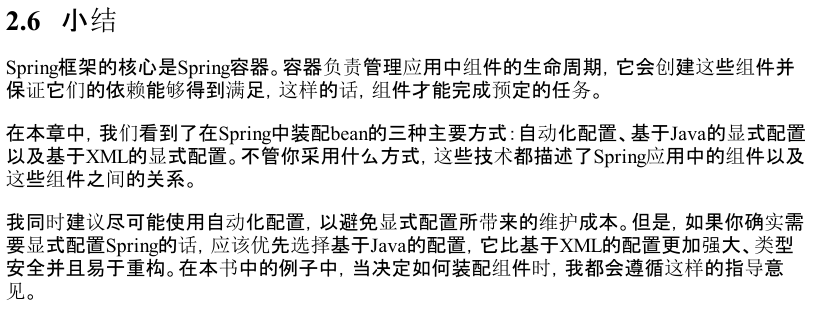
1. 将javaConfig配置类作为一个bean写到xml文件中

详情见markdown: “xml配置中使用javaConfig”

作者建议：不管是javaConfig还是xml配置，推荐有一个根配置类或者是根配置文件。



疑问：@Configuration注解是什么作用：？？ http://www.tuicool.com/articles/M3MVr2



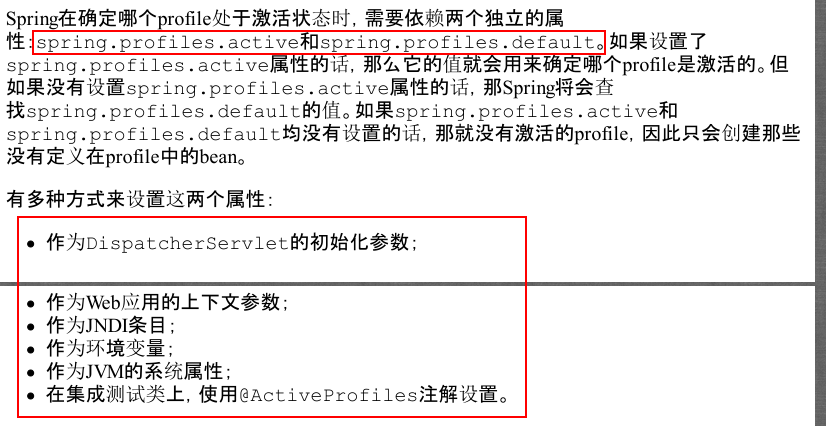
# 第三章

## 高级装配

### 多种环境的profile

1. 借助Maven的profile
2. Spring 的profile Bean
3. 在Spring 3.1中，只能在类级别上使用@Profile注解。不过，从Spring 3.2开始，你也可以在方法级别上使用@Profile注解，与@Bean注解一同使用。这样的话，就能将这两个bean的声明放到同一个配置类之中.
4. 没有指定profile的bean始终都会被创建，与激活哪个profile没有关系
5. 在Java配置中，可以使用@Profile注解指定某个bean属于哪一个profile。
6. @Profile注解应用在了类级别上。它会告诉Spring这个配置类中的bean只有在dev profile激活时才会创建。如果dev profile没有激活的话，那么带有@Bean注解的方法都会被忽略掉。

激活profile的方式：



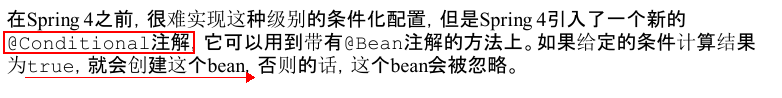
1. 当设置spring.profiles.active以后，至于spring.profiles.default置成什么值就已经无所谓了；系统会优先使用spring.profiles.active中所设置的profile.
2. 你可能已经注意到了，在spring.profiles.active和spring.profiles.default中，profile使用的都是复数形式。这意味着你可以同时激活多个profile，这可以通过列出多个profile名称，并**以逗号分隔**来实现。
3. 测试时启用profile: 注解@ActiveProfiles指定测试时候激活哪一个profile.





详情见markdown: “注解@Profile实现不同环境”

## 条件化bean—@Conditional



//表明这个bean只有满足StudentExistCondition要求的条件才会注册到Spring中。

@Conditional(StudentExistCondition.**class**)

@Component(value = "student")

**public** **class** Student {

**private** String name = "11";

**private** Integer age = 2;

**public** String getName() {

**return** name;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Student [name=" + name + ", age=" + age + "]";

}

}

//此类必须实现org.springframework.context.annotation.Condition

public class StudentExistCondition implements Condition {

Integer num = 8;

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

if(num ==8){

return true;

}

return false;

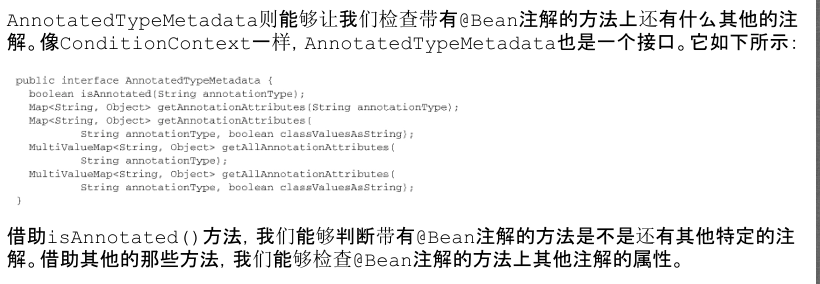
}

}

**ConditionContext接口**

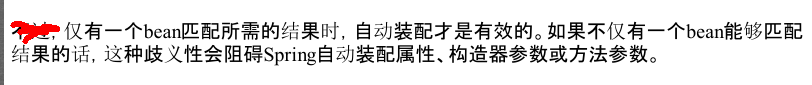


**AnnotatedTypeMetadata接口**



详情见markdown: “条件化bean—@Conditional”

## 自动装配的歧义性



1. @primary:设置bean为首选bean.(如果一个类型的多个实现类都设置了primary,那么spring就会报错。【@Componet,@Bean，xml中bean的属性primary=”true”】
2. @Qualifier(“classicCD”):指明了id。(在没有其他限定符的情况下,bean的限定符默认是和id一致的。)

使用的地方：

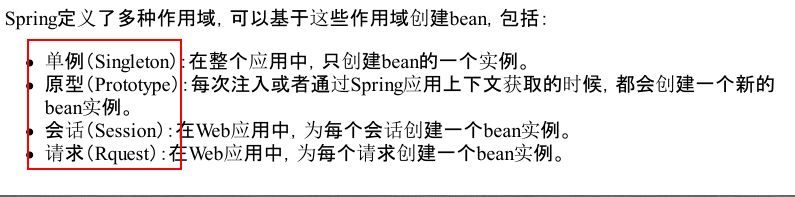
* + - * 1. 限定符默认值：和id一致。
        2. 和@autowired组合使用，指明依赖的bean的限定符
        3. 和@Component组合使用，给bean自定义限定符(自动扫描)【自定义限定符】
        4. 和@Bean组合使用，给bean自定义限定符（javaconfig）【自定义限定符】
        5. 作用于自定义的注解上，解决@qualifier注解不可以重复出现在一个条目上的问题。自定义注解 可以理解为给bean标注特性或者特征，在需要注入的时候，通过制定限定符注解从而找到bean(通过特性特征找到bean)【自定义限定符注解】

详情见markdown: “限定符—@Qualifier-自定义限定符”,” 限定符—@Qualifier-自定义限定符注解”

## Bean作用域（默认Singleton）







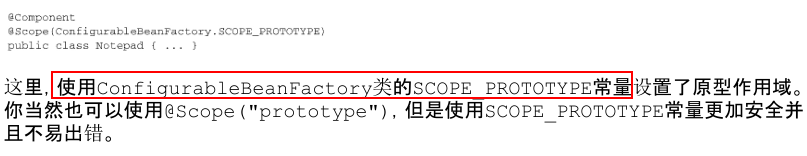
\* **@see** ConfigurableBeanFactory#SCOPE\_PROTOTYPE 原型作用域。

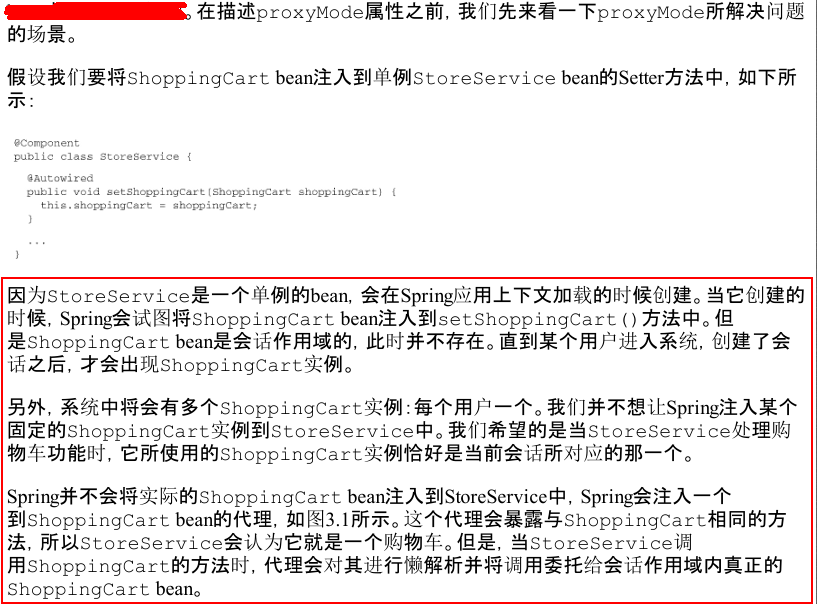
\* **@see** ConfigurableBeanFactory#SCOPE\_SINGLETON 单例作用域。

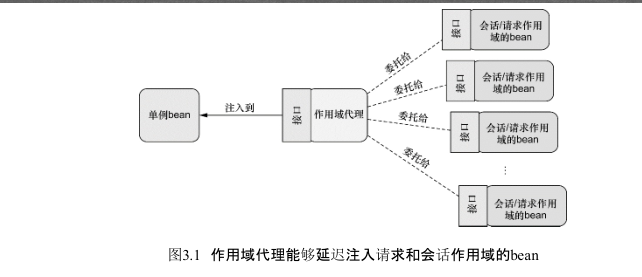
\* **@see** org.springframework.web.context.WebApplicationContext#SCOPE\_REQUEST 在请求范围内是单例的

\* **@see** org.springframework.web.context.WebApplicationContext#SCOPE\_SESSION 在会话范围内是单例的，即会话范围内是共享的。

@Scope注解







方法一：Java代码配置作用域代理：

/\*\*

\* scope:定义bean的作用域.

\* value取值：直接看源码。

\* proxyMode作用：因为这个bean实在会话范围内才会创建的，但是如果这个bean被注入到单例的bean中（单例bean是在spring容器加载的时候就已经创建了），这个时候

\* 就无法找到这个bean.怎么办？实际上，加载的时候，这个类没有必要创建，这个时候就需要借助代理了。在创建单例bean的时候，注入本bean的代理类，在会话范围内调用代理

\* 类的时候，再去调用目标类。而这里的proxyMode指明的就是代理的方式：基于接口的；或者基于类的。（主要看目标类有没有时间接口，如果实现了接口，那么就基于接口，否则就

\* 使用cglib的基于目标类的）

\*/

@Component

@Scope(value=WebApplicationContext.***SCOPE\_SESSION***,proxyMode=ScopedProxyMode.***TARGET\_CLASS***)//

**public** **class** ScopeBeanOfSession {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return** "ScopeBean [name=" + name + "]";

}

}

方法二：Xml方式配置作用域代理：

<!-- xml方式设置作用域代理 -->

<bean id=*"scopeXmlOfSession"* class=*"net.shopin.charactor3.scope.ScopeXmlOfSession"* scope=*"session"*>

<aop:scoped-proxy proxy-target-class=*"true"*/>

</bean>

测试代码：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations={"classpath:application\_scope.xml"})

**public** **class** ScopeTest {

@Autowired

ScopeBeanOfSession sos;

@Test

**public** **void** testProxyScope(){

System.***out***.println(sos.getClass().getName());

}

}

返回结果：

net.shopin.charactor3.scope.ScopeBeanOfSession$$EnhancerBySpringCGLIB$$dd0ddb9a

很明显，返回结果是一个代理类！！！

## 运行时值注入

### 属性占位符(Property placeholder)

1. 核心组件：

**@PropertySource** ：指定属性文件的位置。【一般用于配置类上】

**Environment**：通过这个对象获取属性值。【javaconfig方式注册bean的时候，bean的创建可能需要属性文件定义的具体属性值，通过这个对象去获取具体的值。】

例子: 详情见markdown: “运行时值注入-属性占位符(Property placeholder)-PropertyResource&Environment”

1. **@Value**的使用场景

获取属性文件的中具体的属性值：

@javaConfig方式的bean，可以通过Environment对象获取属性值。（@PropertyResource指明属性文件位置）

@xml声明的bean，可以通过占位符获取“${…}”,但是需要设置属性文件的位置(<context:property-placeholder location>)。

@如果通过自动扫描注入的bean，又需要获取属性文件的内容，怎么办？？。这个时候，就可以借助@Value了。

@Value的两种使用方法：

[A.@Value(${jdbc.username})](mailto:A.@Value($%7bjdbc.username%7d)) ------->依赖于PropertySourcesPlaceholderConfigurer【能够基于Spring Environment

及其属性源来解析占位符】

[B.@Value(#{configProperties[‘jdbc.driver’]})- ------](mailto:B.@Value(#{configProperties['jdbc.driver']})-  ------)>依赖于PropertiesFactoryBean和SpEL表达式

例子:

详情见markdown:

“@Value@Value-PropertiesFactoryBean（使用SpEL）”

”@Value-PropertySourcesPlaceholderConfigurer”

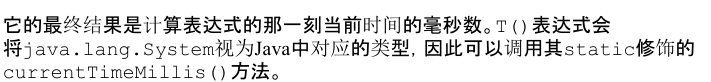
### SpEL(Spring Expression Language)Spring表达式语言

1. 格式：#{}

Eg.#{student.name}--🡪获取id为student的bean的name属性值。

Eg.#{systemProperties[‘jdbc.username’]} ----🡪通过systemProperties对象引用系统属性。

Eg.



Eg.

1.整数，浮点数，Boolean,string ---🡪字面量eg.#{1}；#{8.9}，#{true};#{‘Hello’}

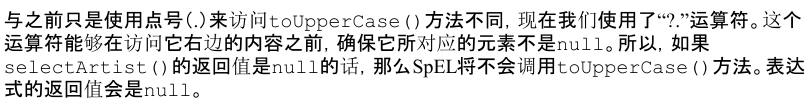
2.引用bean的id----🡪#{student}表明已经获取到id为student的bean对象

3.获取对象中的属性或者方法---🡪#{student.name}表明获取到id为student的bean的name属性的值。

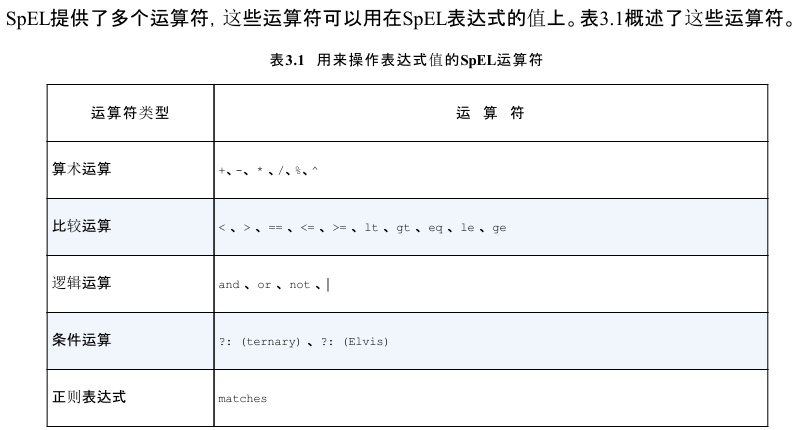
---🡪#{student.toString()}表明调用id为student的bean的toString()方法。

--🡪#{student.toString().toUpperCase()}也可以再次调用方法

4.”?.”表示先判断问好左边的是否null，如果为null,则不调用右边的方法，同时表达式返回null.



1. T(类的权限定名)可以获取到对象的静态方法和常量。Eg.#{T(java.util.Math).PI}；#{T(System).currentTimeMillies()}
2. SpEL中的运算符



^ 乘方。

字符串的“+”即拼接字符串。

Eg.#{student.age==18} ；#{student.age eq 18}

1. 三元运算符

Eg.#{student.score>80?’优秀’:’合格’}；如果学生分数大于80，那么就是“优秀”，否则为‘合格’

#特殊情况（又称Elvis运算符）：#{student.score?:’合格’}；判断学生分数是否为空，如果是null,则返回‘合格’。

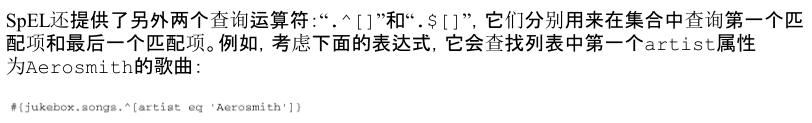
解释一下这种情况，

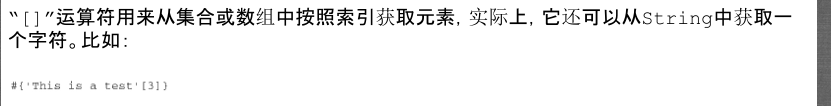
如上面讨论的那样，？具有判断是否为null的作用，当发现为null就不执行右边的了，所以如果

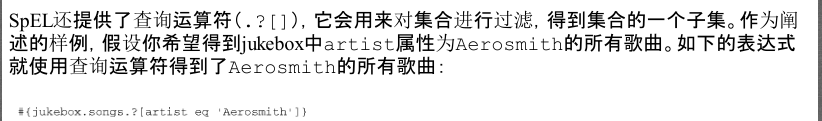
为null,那么就不执行冒号左边的了，直接执行冒号右边的表达式了。

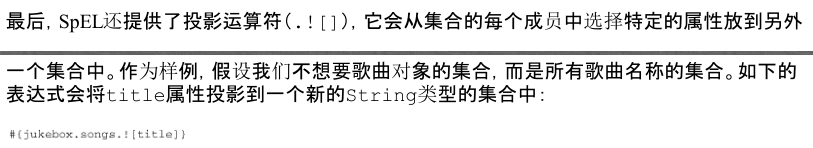
1. 正则表达式

Eg. #{student.email matches ‘([a-zA-Z0-9\_-])+@([a-zA-Z0-9\_-])+(.[a-zA-Z0-9\_-])+’ }

1. 集合([])
2. 查询运算符（.?[……]）
3. 



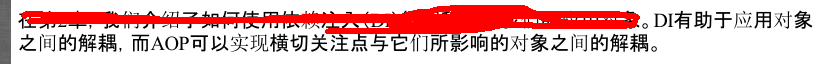




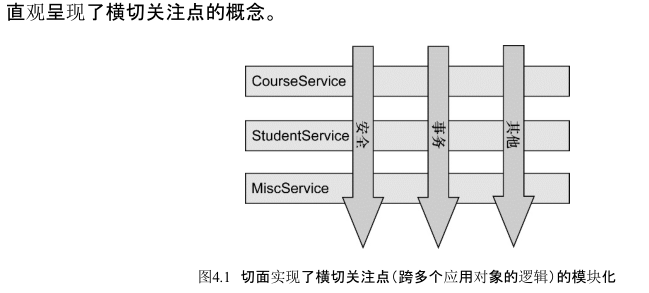
# 第四章

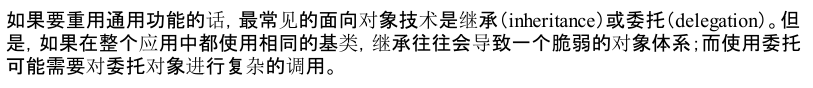
## What:AOP(面向切面编程)



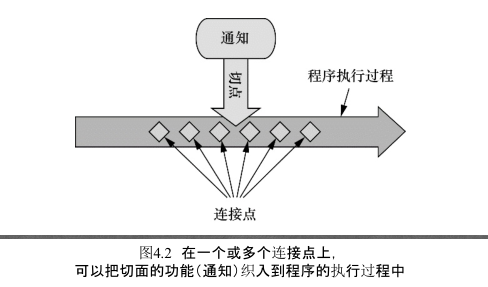


下图直观呈现横切关注点的定义：





### 专业术语



切面（aspect）：横切关注点被模块化为特殊的类，这个类就是切面。

横切关注点（crossing-cutting corcon）：分散在各个应用中多处相似的功能，就是横切关注点。

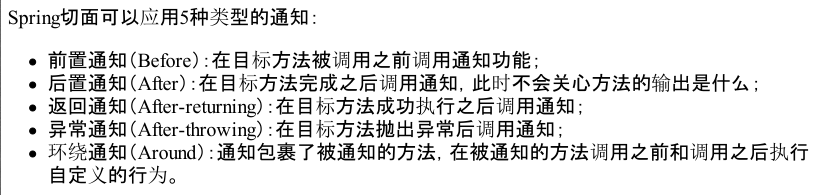
定义切面的好处：

@将所有的横切关注点移动到一处，不用分散在应用功能的各处，便于管理，也比较直观。

@服务类只需要关注核心代码，关注业务逻辑，而无需关心其他逻辑。



通知（advice）:执行什么+什么时候执行。





连接点（join point）：可以使用通知的各个方法，就是连接点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。

切点（pointcut）：定义实际需要执行通知的位置。（正则）

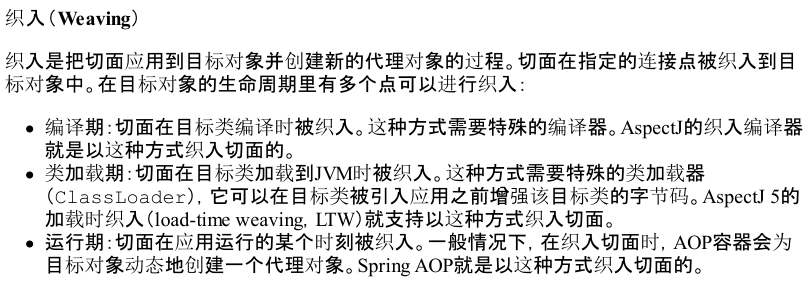


切面(aspect):通知（执行什么+何时执行）+切点（何处执行）=切面

引入（introduction）:向类中添加新方法和新属性。？？？

织入（weaving）：

（运行期：spring aop）

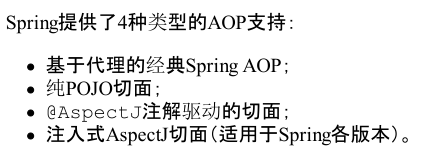


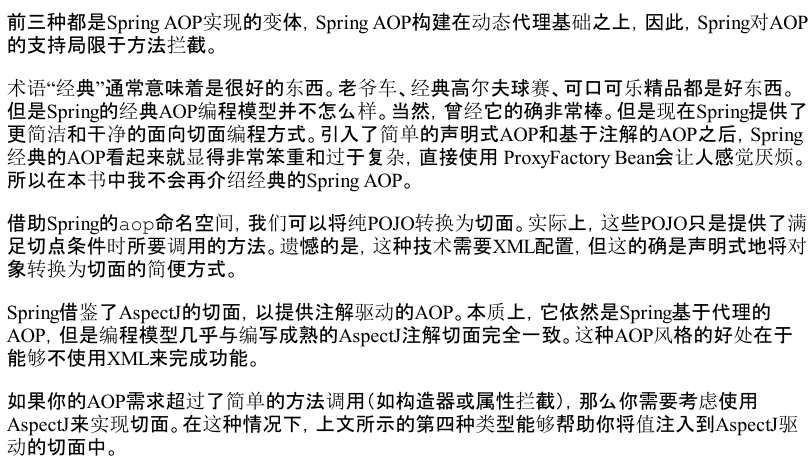
总结：

* 通知包含了需要用于多个应用对象的横切行为；
* 连接点是程序执行过程中能够应用通知的所有点；
* 切点定义了通知被应用的具体位置（在哪些连接点）。
* 其中关键的概念是切点定义了哪些连接点会得到通知。

### spring aop的局限性

书籍《spring实战4》章节“4.1.2　Spring对AOP的支持”建议多看几遍。





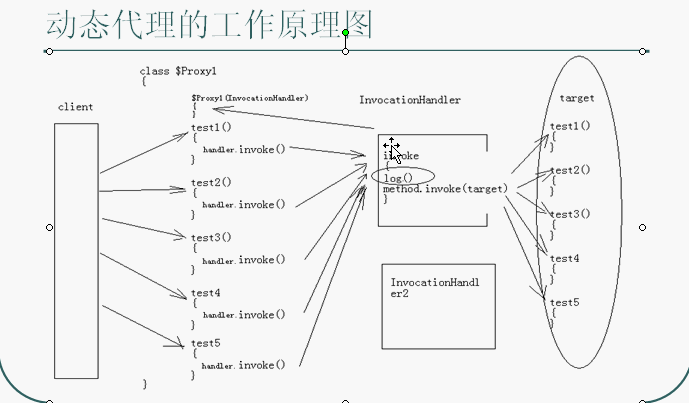
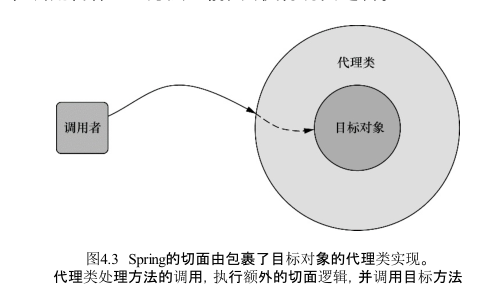
Spring Aop基于动态代理的基础之上，所以spring对于aop的支持，只能够局限于方法拦截。Spring aop无法支持在构造器或者属性上使用切面，也就是：（构造器）当一个bean创建时，我们无法添加额外功能；（属性）当一个属性被修改时，我们也无法拦截。如果需要使用这些，需要借助AspectJ

@ Spring所创建的通知都是用标准的Java类编写的

@AspectJ最初是以Java语言扩展的方式实现的。

@Spring在运行时通知对象

@Spring只支持方法级别的连接点。



声明式AOP

基于注解的AOP

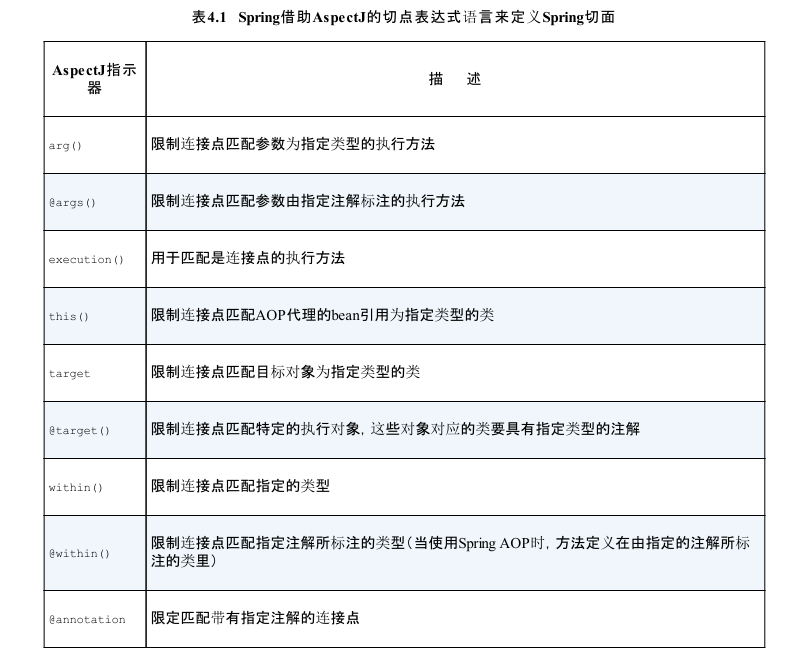
经典的spring aop

## 编写切点

http://sishuok.com/forum/posts/list/281.html







注意：

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。这说明execution指示器是我们在编写切点定义时最主要使用的指示器。在此基础上，我们使用其他指示器来限制所匹配的切点。

与：&&(and)

或：|| （or）

非：!(not)

@Pointcut:切点（切点表达式）

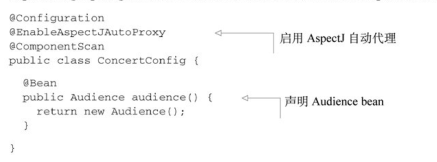
@AspectJ：切面

如何开启AspectJ切面？？

1.JavaConfig:

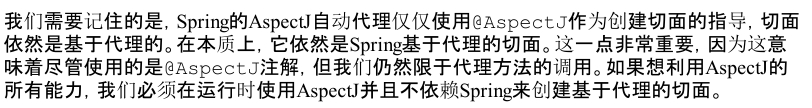
使用@EnableAspectJ-AutoProxy注解启用自动代理功能

@Bean显示声明切面bean,交给spring管理



xml: <aop:aspectj-autoproxy>





# 零散知识点：

## 和@Bean一起使用的注解

@Profile 指明此bean在哪个profile中注册到容器中。【基于@conditional】

@Conditional 指明此bean在什么条件下，才会注册到容器中。

@Scope:指明此bean的作用域。（默认单例）

@Bean javaConfig方式声明Bean,将bean交给spring管理。

@Component 需要依赖于@Component-scan，组件扫描寻找@Component的类，交给spring管理。

1. Java不允许在同一个条目上重复出现相同类型的多个注解。

补充：Java 8允许出现重复的注解，只要这个注解本身在定义的时候带有@Repeatable注解就可以。不过，Spring的@Qualifier注解并没有在定义时添加@Repeatable注解。

|  |  |  |  |
| --- | --- | --- | --- |
| Java依赖注入规范 | spring注解 | Xml文件 | 作用 |
| @inject | @autowired |  | 自动装配 |
| @Named | @Component | <bean> | 给bean命名 |
|  | @Scope(proxyMode=ScopedProxyMode.Interfaces | <bean id=*""* class=*"net.shopin.charactor3.scope.ScopeXmlOfSession"* scope=*"session"*>  <aop:scoped-proxy proxy-target-class=*"false"*/>  </bean> | 设置作用域代理:如左例，设置bean在会话范围是有效的，同时代理的方式是基于接口方式的。 |

1. **需要被注入依赖对象的bean需要提供无参构造**：因为setter方式的自动装配过程是这样的：

先调用此bean的通过调用无参构造器或无参static工厂方法实例化bean之后，调用该bean的setter方法，即可实现*基于setter*的DI。

（http://blog.csdn.net/bluetjs/article/details/52057849）

## 注解相关

1. 注解是可以继承的！！
2. **元注解（meta-annotation）：**

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：  
　　　　1.@Target,  
　　　　2.@Retention,  
　　　　3.@Documented,  
　　　　4.@Inherited

## 作者推荐阅读：

* DI: Dhanji R. Prasanna的《DependencyInjection》，该著作覆盖了依赖注入的所有内容。
* Spring Integration（整合）:Mark Fisher、Jonas Partner、Marius Bogoevici和Iwein Fuld编写的《Spring Integration in Action》（Manning，2012，[www.manning.com/fisher/](http://www.manning.com/fisher/)）
* Spring Bathch: Arnaud Cogoluegnes、Thierry Templier、Gary Gregory和Olivier Bazoud编写的《Spring Batch in Action》（Manning，2012，[www.manning.com/templier/](http://www.manning.com/templier/)）
* AspectJ和AspectJ切点表达式语言：Ramniva Laddad编写的《AspectJ in Action》第二版（Manning，2009，[www.manning.com/laddad2/](http://www.manning.com/laddad2/)）

# 其他

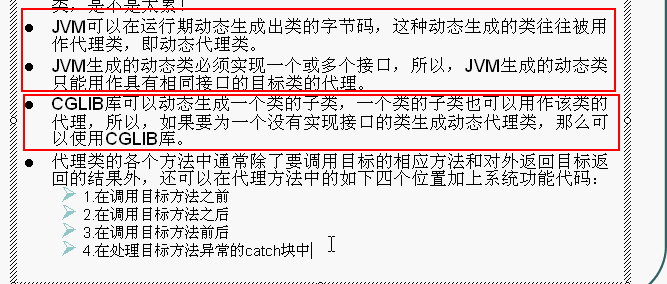
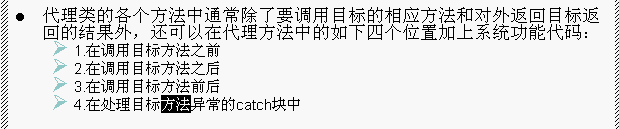
## 动态代理

动态代理：

详情见本地代码: “E:\eclipse\_file\_1\DesignMode”

动态的创建代理类并且动态的调用目标类的方法。需要给某一些类添加一些系统功能，这些功能非常相似，需要添加在方法前或者后的，这部分系统功能抽离出来，即Advice,而动态代理就在代理类在去调用目标类方法前后去调用相应的advice.这就是动态代理，即给目标添加添加功能，而只有在运行时期，才可以真正知道目标类是谁。

子类可以继承父类中的方法，这样就可以获取到Method.

1. 
2. 

Proxy.getProxyClass([ClassLoader](mk:@MSITStore:G:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/ClassLoader.html) loader, [Class](mk:@MSITStore:G:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html)<?>... interfaces)；代理类会实现给定的接口。

说明：针对interfaces,所有非公共接口必须位于同一包中；否则，该代理类将不可能实现所有的接口，无论它在哪一个包中定义(先记着，不明白。)

1. 动态代理执行过程。

Class Proxy{

Private Invocationhandler ih;

Public proxy(Invocationhandler ih){

This.ih = ih;

}

Public void add(String args){

Ih.invoke(this,addMethod,args);

}

}

**public** **class** ProxyHandler **implements** InvocationHandler {

**private** Object targetClass;

**public** ProxyHandler(Object targetClass) {

**super**();

**this**.targetClass = targetClass;

}

@Override

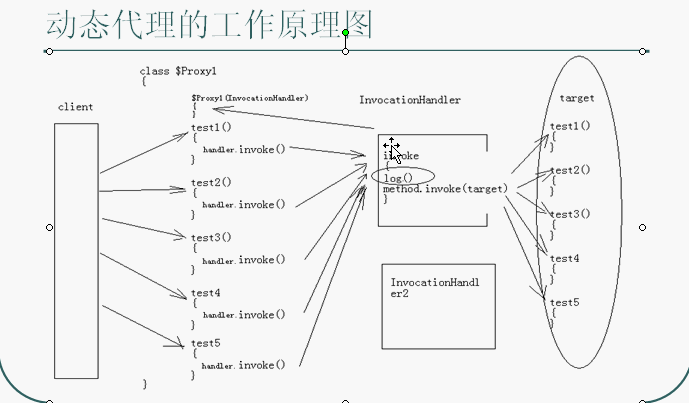
**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

**return** method.invoke(targetClass, args);

}

}

调用Proxy.add(args)——> 调用invocationHandler.invoke(..);——>重点：invoke内部：调用的是目标对象相应的方法。



面向切面编程：将日志功能，系统功能封装成为了一个对象，将切面的代码以面向对象的方式进行封装。【上图中的log部分代码写到一个对象中去，将对象传给invocationHandler，ih直接调用对象中的方法，也即添加了方法，就不是硬编码了。嘻嘻】

1. 匿名内部类需要访问外部的变量，需要把变量定义为final，因为两个的生命周期不一样。
2. Stingbuilder和Stringbuffer的区别：

StringBuffer: 线程安全，速度相对慢。（使用场景：如果字符串变量是类的属性，即共享参数，需要考虑安全。）

StringBuilder: 线程不安全，速度相对快。（使用场景: 如果字符串变量位于方法中，即单线程操作，没有安全问题。）

速度：StringBuilder>StringBuffer>String

1. 反射

ProxyClass.newInstance()----🡪这个调用的是不带参数的构造方法。

2017.5.23

访问路径：https://app.shopin.cn/cms/act....

