Rays's

Personal Study Note

of

Spring

目 录

[1. 装配Bean 2](#_Toc6519382)

[1.1. 自动化装配 2](#_Toc6519383)

[1.2. 代码配置 2](#_Toc6519384)

[1.3. XML配置 2](#_Toc6519385)

[1.4. 导入和混合 3](#_Toc6519386)

[1.4.1. 代码配置 3](#_Toc6519387)

[1.4.2. xml配置 3](#_Toc6519388)

[2. 高级装配 4](#_Toc6519389)

[2.1. 环境和Profile 4](#_Toc6519390)

[2.1.1. 配置 profile 4](#_Toc6519391)

[2.1.2. 激活 profile 4](#_Toc6519392)

[2.2. 条件化的bean 5](#_Toc6519393)

[2.3. 处理自动装配歧义 5](#_Toc6519394)

[2.4. Bean的作用域 5](#_Toc6519395)

[2.5. 运行时注入 6](#_Toc6519396)

[2.5.1. 注入外部的值 6](#_Toc6519397)

[2.5.2. SpringEL 6](#_Toc6519398)

[3. 面向切面 7](#_Toc6519399)

[3.1. 术语 7](#_Toc6519400)

[3.1.1. 术语 7](#_Toc6519401)

[3.1.2. 动态代理 7](#_Toc6519402)

[3.1.3. SpringAOP 8](#_Toc6519403)

[3.2. 切点选择 8](#_Toc6519404)

[3.2.1. 编写切点 9](#_Toc6519405)

[3.3. 注解创建切面 9](#_Toc6519406)

[3.3.1. 定义切面 9](#_Toc6519407)

[3.3.2. 启用切面 11](#_Toc6519408)

[3.3.3. 环绕通知 11](#_Toc6519409)

[3.3.4. 添加方法 11](#_Toc6519410)

[3.4. XML创建切面 12](#_Toc6519411)

[4. Spring MVC 14](#_Toc6519412)

# 装配Bean

## 自动化装配

自动化装配的两大要素

* 组件扫描（Component Scan）：自动发现Bean
* 自动装配（Autowiring）：自动处理Bean之间的关系

1.组件扫描

@ComponentScan或者<context:component-scan> 开启自动扫描功能，通过basepackage指定扫描的范围，使用 @Component 注解标记需要的Bean对应的类（或者使用JDI规范@Named）。

2.自动装配

@Autowired 注解表明Bean和其他Bean的依赖关系，可以用于构造函数、set方法和直接用于属性（或者使用JDI规范@Inject），Spring在扫描时会按照关系设置属性。

## 代码配置

Spring可以直接使用Java代码来进行配置。

1.启动代码配置

为某个类添加注解@Configuration，表明某个类是配置类。

2.配置Bean

编写方法创建并返回Bean，为方法添加@Bean注解

@bean

A bean1(){ return new A(); }

3.配置注入

在配置Bean的方法中，加入对其他Bean的依赖即可有两种方式：

一种是配置方法依赖其他Bean的配置方法

B bean2() { return new B(bean1());}

一种是配置方法依赖其他的Bean的参数传入

B bean2(A a) { return new B(a); }

## XML配置

新的Spring项目应该尽量不再使用xml配置，了解xml配置主要用于维护旧项目。

1.规范

Spring的配置xml文件以<beans>元素为根，并且需要在xml顶部声明多个xml模式（XSD）文件。

2.配置<bean>

使用 <bean> 元素配置Bean

3.配置依赖

<constructor-arg> 或 -c 标签配置构造器依赖

<property> 或 -p 配置属性依赖

## 导入和混合

### 代码配置

@Import 引入其他配置类

@ImportResource 引入配置文件

### xml配置

<Import > 元素引入其他配置文件

<bean > 元素直接将配置类作为Bean配置

# 高级装配

## 环境和Profile

对于同一个bean，希望在不同的场景下使用不同的配置方案，可以使用profile功能。

常见的场景就是数据源的配置，就算在普通情况下，也会有生产和测环境的区别，这时候数据源会希望使用不同的地址、用户名等。

### 配置 profile

@Profile注解，可以用于类、配置方法表明配置的生效范围。

如 @Profile("dev") 表明在 dev profile 激活后生效。

在xml中使用 <beans> 元素的 profile 可以配置生效范围。也可以再 <beans> 中嵌套 <beans> 而不用创建多个文件来配置生效范围。

### 激活 profile

Spring 依赖两个属性来确定当前的 profile：

spring.profiles.active和spring.profiles.default

优先使用 active 若该值没有设置则使用default，若default也没有设置，则只创建不包含 profile 的bean。

这两个属性的设置方式有多种

* 作为DispatcherServlet的初始化参数；
* 作为Web应用的上下文参数；
* 作为JNDI条目；
* 作为环境变量；
* 作为JVM的系统属性；
* 在集成测试类上，使用@ActiveProfiles注解设置。

可以同时激活多个profile，一般用于多个profile彼此无关的场景可能需要。

## 条件化的bean

@Conditional 注解用于特殊场景，只有当某个条件符合时才创建某个Bean。该注解使用 Condition 接口进行条件比对。

public interface Condition {

boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);

}

任何实现该接口的方法都可以用于条件比对，只有matches方法返回true时，才创建bean。matches方法的参数 ConditionContext 也是一个接口，大致方法如下：

* 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所加载的资源；
* 借助getClassLoader()返回的ClassLoader加载并检查类是否存在。

AnnotatedTypeMetadata 也是一个接口，可以用来检查要配置的bean是否包含其他注解。在Spring 4 之后的profile 的代码也重构为基于Conditional来创建。

## 处理自动装配歧义

Spring在处理自动装配时，若没有无歧义的选项时，会抛出NoUniqueBeanDefinitionException，

@Primary 用于Bean 定义，标记该Bean是首选Bean

@Qualifier 用于注入定义，标记将注入的值首选某个Bean

## Bean的作用域

默认情况下Spring的所有的Bean均为单例模式。

Spring定义的作用域有：

单例：整个应用中，只创建一个

原型：每次注入或获取均创建新的对象。

会话：web应用中，每个会话中只有一个。

请求：web应用中，每次请求只有一个。

使用 @Scope注解，并配置ConfigurableBeanFactory中的常量，可以设置Bean的作用域。

## 运行时注入

### 注入外部的值

1. 使用 @PropertySource 声明外部属性文件，使用 Environment 类来检索属性值。

2. 属性占位符

在配置中使用 @Value("${ ... }") 表明该处是一个属性的值

然后 配置 PropertyPlaceholderConfigurer bean或PropertySourcesPlaceholderConfigurer 来处理处理占位符。

### SpringEL

略

# 面向切面

## 术语

### 术语

面向切面：希望程序在某处做某事，将这些期望统一管理则形成切面。

* **通知Advice**，就是希望做的事情，Spring切面中有5种通知：

1. 前置、2. 后置、3. 返回、4. 异常、5. 环绕

从定义可以看出Spring的通知和Method是密切相关的，实际上就是在方法执行过程中寻找时机完成功能。

* **连接点Join Point**，可以运行通知的时机
* **切点PointCut**，需要运行通知的时机，就是某处
* **引入**，向现有的类添加新方法或属性。
* **织入**，实现新的切面功能，必然要对已经存在的代码进行修改，不过这种修改是模式化的。织入大致可以在以下几个阶段：

编译器：编译类的时候，加入切面的代码，这时候需要特殊的编译器。

类加载期：加载类的时候，加入切面的代码，需要特殊的加载器（ClassLoader）

运行期：应用运行的时候，加入切面的代码，一般是通过动态代理实现。Spring就是基于该方式。

### 动态代理

代理是一种设计模式， java为代理提供了动态代理的语法支持。

假设一个类Impl实现了接口Intf，现在要实现代理Impl，可以有两种方式：

1. 编写类IntfProxy，添加成员Impl并实现接口Intf。然后在每个Intf方法实现中，使用Impl调用对应的方法。直接创建类IntfProxy对象，就得到了Impl的代理，这就是静态代理。

2. 编写类IntfHandler实现InvocationHandler接口，添加成员Impl。在invoke方法中使用method.invoke(implObj, args)来调用Impl的方法。使用Proxy. newProxyInstance创建Impl的代理对象，这就是动态代理。

动态代理和静态代理的区别在于，动态代理不需要预先实现被代理对象的接口，一个动态代理的调用可以为任何接口的对象提供代理支持。

### SpringAOP

Spring的AOP是基于动态代理完成的，java 中动态代理的核心功能就是拦截方法调用，因此SpringAOP也是在方法拦截的基础上完成。

切面实际上是需要添加的功能，表现在代码中也只是一个方法。因此Spring的AOP的过程就是，先有业务方法，然后有切面方法，然后确定切面方法是在业务方法执行前、执行后或者异常后来执行。

基于动态代理实现的过程是这样的，业务Bean作为被代理对象，切面成为代理，对业务Bean的方法调用都会被代理拦截。代码实现上Spring使用ReflectiveMethodInvocation接管方法调用，然后递归执行proceed方法，每层递归会执行不同的Invocation，而前置、后置等通知的区别就在于是递归前还是递归后执行。



## 切点选择

前面已经介绍了连接点和切点，在Spring中切点实际上就是方法。Spring的切点指示器是AspectJ的一个子集，具体如下：

|  |  |
| --- | --- |
| 指示器 | 描述 |
| args() | 指定切点的参数类型 |
| @args() | 指定切点的参数的类型包含指定注解 |
| execution() | 指定切点方法 |
| this() | 指定切点代理引用的类，用于Bean未实现接口的情况，Spring会使用CGLib来实现代理 |
| target() | 指定切点所在的对象类，用于Bean实现了接口的情况，Spring会使用动态代理 |
| @target() | 指定切点所在的对象的类需要包含的注解 |
| within() | 指定切点所在的类或者包 |
| @within() | 指定切点所在类需要包含的注解 |
| @annotation | 指定切点方法的需要包含的注解 |
| bean() | 指定切点所在的bean的id或名称 |

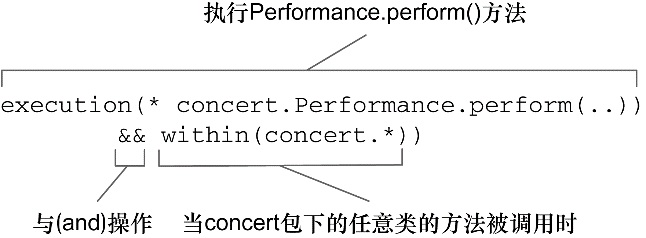
### 编写切点

编写切点的过程就是使用指示器指定切点方法的过程。Spring中可以使用&&,||和！逻辑运算符来对指示器进行逻辑运算，也可以使用 and， or 和 not。

只是器内部值是一种类似于正则表达式的模糊字符串，例如execution指示器可以这样来指定：



在所有指示器中，只有execution是直接指定方法，其他的指示器主要是在execution的基础上做二次筛选，因此可以结合其他指示器，对切点做进一步的限制，如：



## 注解创建切面

### 定义切面

使用@Aspect标注可以创建一个切面，使用具体的注解可以指定切点和通知，由于通知的不同，指定通知的方式也有所不同。具体的通知指定方式有以下几种：

|  |  |
| --- | --- |
| 注解 | 通知 |
| @After | 指定通知在方法结束或异常后调用 |
| @AfterReturning | 指定通知在方法结束后调用 |
| @AfterThrowing | 指定通知在方法异常后调用 |
| @Around | 指定通知环绕切点方法 |
| @Before | 指定通知在目标方法之前调用 |

使用不同的注解指定通知执行的时机，使用指示器指定通知覆盖的范围，Spring会根据这些配置为需要的Bean创建代理。如一个Dog类实现了Pet接口，其中有一个play()方法。然后有一个PetMaster的类，他希望自己的宠物在玩耍时不要受伤，因此他会格外关注宠物的玩耍的情况，这时候就可以创建一个关注宠物玩耍的切面。

@Aspect

class PetMaster {

@Before("execution(\*\* com.rays.entity.Pet.play(..))")

public void beforePlay(){

// watch pet play

}

}

这样Spring会为Dog的bean创建一个代理，代理中在执行play方法前，会先执行PetMaster Bean的beforePlay方法。如果要为一个方法创建多个通知，那么不可避免的会重复的配置多个雷同的指示器表达式，这样对于Spring来说也将有一个冗余的解析过程。可以使用@Pointcut注解来指定切点，然后在其他的通知表达式中直接使用切点配置方法即可。

@Aspect

class PetMaster {

@Pointcut("execution(\*\* com.rays.entity.Pet.play(..))")

public void petPlay(){}

@Before("petPlay ()")

public void beforePlay(){

// watch pet play

}

}

### 启用切面

只是做了3.3.1中的操作并不能使切面生效，首先为了让Spring能够知道这个切面的存在，需要将PetMaster也配置成一个Bean，然后需要在配置中其中AOP功能。在配置类上使用注解@EnableAspectJAutoProxy或者在xml中使用<aop:aspectj-autoproxy>可以启用SpringAOP配置。

### 环绕通知

其他的通知中，我们只需要编写通知功能就能够起作用，环绕通知则需要多做一些操作。使用@Around可以创建环绕通知，但是必然的，我们需要在环绕通知内指定目标方法在何时调用（或者不调用）。Spring使用ProceedingJoinPoint来表示目标方法调用。因此环绕通知方法必须接受一个ProceedingJoinPoint类型的参数jp，然后在通知内需要的时候，使用jp.proceed()类调用目标方法。

@Around("execution(\*\* com.rays.entity.Pet.play(..))")

public void aroundPlay(ProceedingJoinPoint pjp) {

try {

Out.p("Another watching begin");

pjp.proceed();

Out.p("Another watching end");

} catch (Throwable e) {

e.printStackTrace();

}

}

### 添加方法

对于已经实现的Dog类，我们需要为其添加Listener接口的listen方法。常规的方式是修改Dog类，实现Listener接口，然后添加的新的方法。使用Spring可以在不修改原来类的情况下，直接为原来类的Bean添加的方法。

首先配置一个切面

@Aspect

@Component

public class DogListener {

@DeclareParents(value="com.rays.entity.Dog+",

defaultImpl=CommonListener.class)

public static Listener dogListener;

}

使用@DeclareParents注解将Listener的功能引入到Dog的bean中去。

* **value**属性指定哪些类型的Bean要引入新的功能。
* **defaultImpl**指定要引入的具体功能（实现了这些功能的类）。
* **@ DeclareParents**本身标记的静态属性指定了需要引入的接口。

最后，和之前的切面一样，将切面配置为一个Bean以实现切面功能。在代码中就可以将dog bean 作为一个Listener来使用。

Pet p1 = (Pet) context.getBean("dog");

Listener l = (Listener)p1;

l.listener();

## XML创建切面

Spring的aop命名空间中，提供多个元素用来在xml中声明切面，如下：

|  |  |
| --- | --- |
| AOP配置元素 | 用途 |
| <aop:config> | 顶层的AOP配置元素 |
| <aop:advisor> | 定义通知 |
| <aop:before><aop:after>  <aop:after-returning>  <aop:after-throwing><aop:around> | 具体通知 |
| <aop:aspect> | 定义切面 |
| <aop:pointcut> | 定义切点 |
| <aop:declare-parents> | 引入额外接口方法 |

xml的配置和注解配置略有不同，注解本身所在的配置也是一种配置信息，而xml的配置则没有了这些优势。如果要通过xml指定一个切面，那么必须显示的标注切面Bean的id。然后在指定通知时，无论何种通知，都要指定切点和通知方法。

如之前的PetMaster的配置可以如下

<aop:config ref="petMaster">

<aop:before pointcut="execution(xxx)" method="beforePlay">

<aop:config>

也可以使用定义切点的方法指定切点

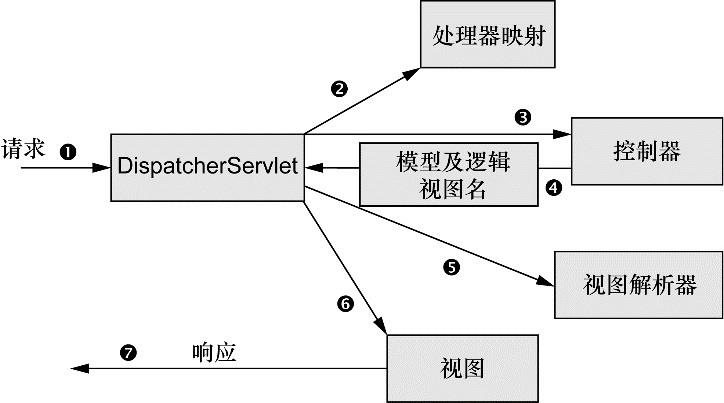
<aop:pointcut id="p1" expression="xx"/>

<aop:before pointcut-ref="p1" xx />

# Spring MVC

Spring MVC是基于servlet的一个MVC框架，其使用DispatchServlet转发所有请求。

## 请求过程



通过配置请求映射到Spring的DispatchServlet，然后编写控制器和视图等模块，Spring将自动的让请求在这些模块中依次处理。请求大致经过的流程有以下几个步骤：

1.请求到达容器，容器转交请求给DispatchSetvlet

2.DispatchSetvler查询handler mapping确定本次请求的控制器

3.转交请求到具体的控制器，等候其处理

4.控制器处理完毕提供模型和逻辑视图名

5.视图解析器根据逻辑视图名解析出最终实际的视图

6.转交模型到最终视图，这个视图可以是JSP也可以是其他技术实现。

7.最终视图根据模型中的资料渲染最终视图并返回响应报文。

## 搭建Spring MVC

### 代码配置

SpringMVC的核心是DispatchServlet，配置Spring的第一步是通过配置，将需要使用Sping处理的请求映射到DispatchServlet。在Servlet3之前，容器只会从web.xml中获取servlet的映射配置，因此只能使用xml文件来配置DispatchServlet。在Servlet3规范中，容器会在类路径中查找配置类，这个配置类需要实现ServletContainerInitializer接口，Spring实现了这个接口并且将配置工作再次转交给实现了WebApplicationInitializer的类来处理配置，Spring同时提供了一个基础实现AbstractAnnotationConfigDispatcherServletInitializer类。总之，借助于此，自定义的程序可以继承以上配置类，进而通过代码来配置DispatchServlet映射和一些其他的配置。

继承的配置类，主要需要实现三个方法：

1. getServletMappings() ： 用于配置哪些请求会映射到DispatchServlet

2. getServletConfigClasses()：用于配置web组件的配置、如视图解析器、控制器、处理器映射等。

3. getRootConfigClasses()：用于配置其他Bean，如一些服务对象、数据层组件等

一个配置类的例子如下：

public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class<?>[] {RootConfig.class};

}

@Override

protected Class<?>[] getServletConfigClasses() { // 指定 servlet 配置类

return new Class<?>[] {WebConfig.class};

}

@Override

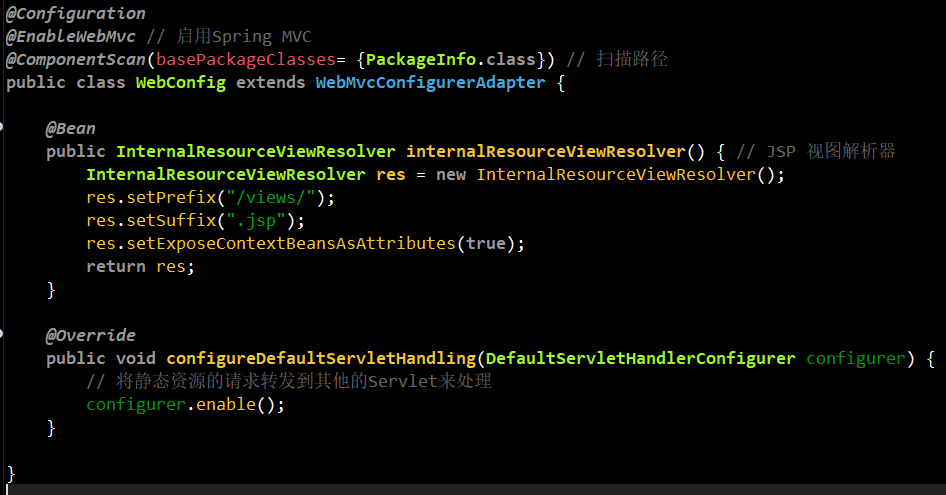
protected String[] getServletMappings() { // 映射 DispatchServlet

return new String[] {"/"};

}

}

### 启用SpringMVC



启用SpingMVC组件的相关配置如图。