# Spring

## Spring注入方式

### 设置注入

为类的属性设置值，通过set方法注入

### 构造注入

使用类的构造方法，为属性设置值，构造方法的名称必须一直

## 2. Bean

### Bean容器的初始化

#### 文件FileSystemXmlApplicationContext

#### ClassPath ClassPathXmlApplicationContext

#### Web 监听器 ContextLoaderListener

#### Web Servlet ContextLoaderServlet

### Bean的配置项

#### Id 整个IOC容器中的唯一标识

#### Class 具体要实例化的类

#### Scope bean的范围/作用域

#### Constructor arguments 构造器的参数

#### Properties 属性

#### Autowiring mode 自动装配的模式 注入方式

#### Lazy-initialization mode 懒加载模式

#### Initialization/destruction medhod 初始化/销毁方法

### Bean的作用域

#### Singleton 单例，指一个bean容器中只存在一份

#### Prototype 每次请求(每次使用)创建新的实例，destroy方法不生效，请求完成后会被自动销毁

#### Request 每次http请求创建一个实例且仅在当前request内有效

#### Session http请求创建，当前session内有效

#### Global session 多系统单点登录

### Bean的生命周期 [定义-初始化-使用-销毁]

#### Bean的初始化方式

1. 实现org.springframework.beans.factory.InitializingBean

覆盖 afterPropertiesSet()

1. 配置 init-method

#### Bean的销毁方式

1. 实现org.springframework.beans.factory.DisposableBean

覆盖 destroy()

1. 配置 destroy -method

#### 注意:先执行 接口的 方法,后执行配置的方法

### Aware接口

实现Aware接口的bean在被初始化后，可以获取相应的资源

#### ApplicationContextAware

#### ServletContextAware

### Bean自动装配 XML设置default-autowire

1. 自动装配的方式
2. 根据名称(ID) byname
3. 根据类型 byType 根据构造器(构造器参数的类型)

### Resources 根据Spring加载资源文件

1. URL UrlResources 前缀url:
2. classPath classpathResources 前缀classpath:
3. file FileResources 前缀file:
4. none 根据Spring的加载方式来进行加载

使用ResourcesLoader.getResources(“”);

### ClassPath扫描与注解管理

#### 注解实现Bean 的扫描

1. XML开启注解 <context:annotation-config> 支持bean被初始化后属性方法的注解

<context:component-scan base-package=””>(包含前者全部功能)

1. 默认情况下，类被自动发现的条件注解@Compenent @Repository @Service @Controlle

可以修改 扫描过滤器来进行过滤

1. 自定义Bean的命名策略 实现BeanNameGenerator 无参构造函数

name-generator

1. 作用域@Scope 自定义Scope策略实现ScopeMetadataResolver接口无参构造函数

Scope-resolver

#### @Required注解

适用于Bean属性的setter方法，为属性赋值

#### @Autowired

@Autowired 可用于构造器或成员变量 及属性的setter方法 ,可以设置required=true，

每个类只能有一个构造器被标记为required=true

如果required=true,建议使用@Required

可以使用@Autowired注解来注解 Spring的解析依赖性接口 比如ApplicationContext

@Autowired接口 可以为 类型为数组的字段或方法，为属性注入上下文中类型为指定类型的bean如果希望数组有序，可以让Bean实现

[org.springframework.core.Ordered接口或使用@Order](mailto:org.springframework.core.Ordered接口或使用@Order)注解

可以注入key为String的Map

@Autowired是由Spring BeanPostProcessor处理， 不能在BeanPostProcessor或BeanFactoryPostProcessor类型应用这些注解，这些类型必须通过XML或@Bean注解

#### @Qualifier

按类型自动装配可能多个Bean实例的情况，可以使用Spring的@Qualifier注解缩小范围或指定唯一，也可用于指定单独的构造器参数或方法参数

可用于注解集合类型的变量

XML中在Bean内部可以使用<qualifier value=””>

#### @Resource

如果通过名字来进行注解，即使通过@Qualifier指定Bean的名字来实现 也不使用，使用@Resource

#### 属性注解的区别

@Autowired适用于 字段 构造器 多参数的方法 ，可以同时在参数级别使用@Qualifier注解缩小注解范围

@Resource 适用于成员变量

### 基于Java的容器注解

#### @Bean 类似与XML 的bean

@Bean注解 方法返回Bean

在类上注解@Configuration 在返回Bean的方法上注解@Bean 等同与在XML中定义bean

@Bean(name=””) 指定bean的id 支持init-method destroy-method

如果没有指定name 那么将方法的名称注解为bean的ID

#### @ImportResource 和 @Value注解进行资源文件读取

XML方式加载资源文件

<context:property-placeholder location=””>

在XML中使用资源文件的值${jdbc.url}

类 注解加载资源文件

@Configuration

@ImportResource(“………..”);

[属性使用@Value (“${jdbc.url}”)](mailto:属性使用@Value();

返回Bean的方法注解@Bean 可以使用属性值

#### @Bean和@Scope

默认@Bean是单类的

@Scope可以设置代理方式 proxyMode=ScopeProxyMode.XXXX

#### 基于泛型的自动装配

Spring可以根据泛型来自动装配

#### Spring对于JSR支持的说明

@Resource支持属性 或 setter方法 可以使用name属性 ，如果没有name 将根据属性属性名 或setter 方法名得出

Spring还支持JSR的初始化回调@PostConstruct 和销毁回调 @PreDestroy 前提是

CommonAnnotationBeanPostPrecessor在Spring ApplicationContext中已注册

JSR330标准 需要引入 javax.inject的支持

@Inject 等效于 @Autowired

@Named 等效于Component 可以指定name 与@Qualifier

## AOP

### 什么是AOP / 实现方式

Aspect Oriented Programming 面相切面编程

通过 预编译方式 运行期动态代理 实现程序功能的统一维护的一种技术

主要功能 日志记录 性能统计 安全控制 事务控制 异常处理

切面是和功能垂直的

预编译 --AspetJ

运行期间动态代理 --JDK动态代理 CGlib动态代理 Spring AOP JbossAOP

### 配置AOP aspect切面 及切面处理类

<aop:config>

<aop:assept id="myAsseptAopInvokeClass" class=""></aop:assept>

</aop:config>

### 配置poincut 切入点



<aop:config>

<aop:assept id="myAsseptAopInvokeClass" class="">

<aop:pointcut id="" expression="execution(\* com.rzp.\*.service.\*.\*(..))"></aop:pointcut>

</aop:assept>

</aop:config>

### 配置通知 Advice

#### 基本通知

<aop:config>

<aop:assept id="myAsseptAopInvokeClass" class="">

<aop:pointcut id="cut" expression="execution(\* com.rzp.\*.service.\*.\*(..))" />

<!--方法执行前执行-->

<aop:before method="before" pointcut-ref="cut"/>

<!--方法返回前执行-->

<aop:after-returning method="afterReturn" pointcut-ref="cut"/>

<!--方法抛出异常后执行，如果抛出一次，则不执行 after-returning -->

<aop:after-throwing method="before" pointcut-ref="cut"/>

<!--方法在返回前最后执行，不论是否有异常都会执行-->

<aop:after method="before" pointcut-ref="cut"/>

</aop:assept>

</aop:config>

#### 环绕通知

Around advice 环绕通知

<aop:around method="around" pointcut-ref="cut"/>

方法的第一个参数必须为 ProceedingJoinPoint 类型

public Object around(ProceedingJoinPoint pj) throw Exception {

Object retVal=null;

try{

retVal =pj.proceed();

}catch(Exception e){

e.printStackTrace();

}

return retVal;

}

### 配置declare-parents

<aop:config>

<aop:aspect id=*"aop"* ref=*"myLogAOP"*>

<aop:declare-parents types-matching=*"com.rzp.sping.\*"*

implement-interface=*"com.rzp.sping.Foo"*

default-impl=*"com.rzp.sping.FooImpl"* />

</aop:aspect>

</aop:config>

为所有匹配 type-matching的类 添加一个默认的接口，和默认的实现

只支持 单例模式

### 配置aop: advisor 通常用来与事务管理结合

<aop:config>

<aop:pointcut id=*"pcut"* expression=*"execution(\* com.rzp..\*(..))"*/>

<aop:advisor advice-ref=*"tx-ad"* pointcut-ref=*"pcut"* />

</aop:config>

<bean id=*"tx-ad"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*></bean>

<tx:advice id=*"tx-ad"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<tx:method name=*"c\*"* propagation=*"REQUIRED"* />

</tx:attributes>

</tx:advice>

## Spring支持@Aspect

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

@Aspect并不能通过注解类的 扫描路径来 被发现需要配合使用@Component 并不会代理自己

### @Pointcut 定义切面

@Pointcut("execution(...)")

返回值必须为void

可以引用切入点的方法来引用切入点，

看可以通过&& || ! 进行切入点的组合

### @Before("pointCut()")

@Before("pointCut(foo)")//获取传递参数

**public** **void** before(Foo foo){}//获取切面注解

@After("pointCut()&& @Annotation(ann)")

**public** **void** after(Ann ann){}

@AfterReturning(pointcut="pointCut()",returning="returnValue")

**public** **void** afterReturning(Object returnValue){ }

@AfterThrowing(pointcut="pointCut()",throwing="ex")

**public** **void** afterThrowing(RuntimeException ex){ }

@Around("pointCut()")

**public** Object round(ProceedingJoinPoint pjp) **throws** Throwable

{

**return** pjp.proceed();

}

@DeclareParents(value="pointCut(foo)",defaultImpl=FooImpl.**class**) //都实现Foo接口

**public** **static** Foo *foo*;