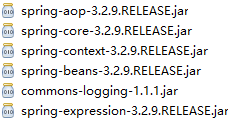
# IOC，DI

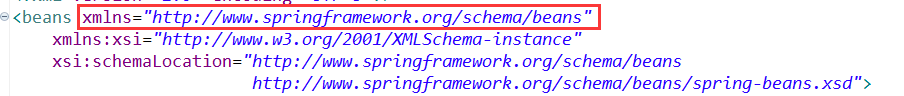
## XML配置

### 准备工作

Jar包

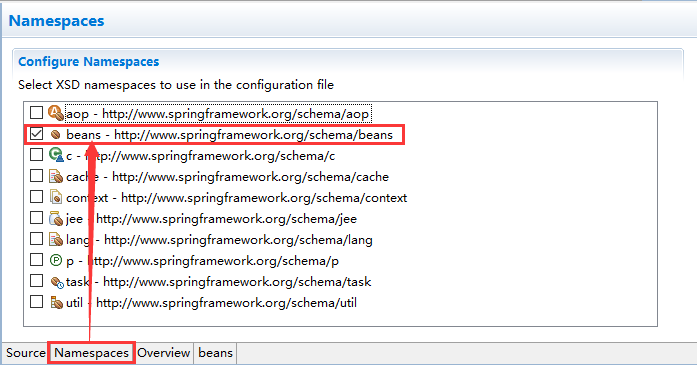


Config.xml文件必须的包含xmlns=<http://www.springframework.org/schema/beans>；如图（1）



图（1）

或者如图（2）配置



图（2）

### Bean定义

#### 依赖Get/Set函数

<bean id=*"Person2"* class=*"com.pojo.Person"*>

<property name=*"name"* value=*"${name}"*></property>

<property name=*"age"* value=*"${age}"*></property>

<property name=*"numbers"*>

<list>

<value>吉利</value>

<value>大众</value>

<value>沃尔沃</value>

</list>

</property>

</bean>

#### 依赖构造函数

<bean id=*"Person"* class=*"com.pojo.Person"*>

<constructor-arg name=*"name"* value=*"jack"*></constructor-arg>

<constructor-arg name=*"age"* value=*"22"*></constructor-arg>

</bean>

#### C命名空间

依赖构造函数且xml中须配置：xmlns:c=*<http://www.springframework.org/schema/c>*

<bean id=*"Person4"* class=*"com.pojo.Person"* c:age=*"19"* c:name=*"赵六"*></bean>

#### P命名空间

依赖get/set且xml中须配置：xmlns:p=*"<http://www.springframework.org/schema/p>"*

<bean id=*"Person3"* class=*"com.pojo.Person"* p:age=*"18"* p:name=*"王五"*></bean>

P命名空间并不支持装配集合（map，list等），为此Spring提供util解决此问题，如下所示：

<util:list id=*"number"*>

<value>吉利</value>

<value>大众</value>

<value>沃尔沃</value>

</util:list>

<bean id=*"Person3"* class=*"com.pojo.Person"* p:age=*"18"* p:name=*"王五"* p:numbers-ref=*"number"*>

</bean>

此外spring util还提供了如下元素

|  |  |
| --- | --- |
| 元素 | 描述 |
| <util:constant> | 引用某个类型的public static域，并将其暴露为bean |
| <util:list> | 创建一个java.util.List类型的bean，其中包含值或引用 |
| <util:map> | 创建一个java.util.Map类型的bean，其中包含值或引用 |
| <util:properties> | 创建一个java.util.Properties类型的bean |
| <util:property-path> | 引用一个bean的属性（或内嵌属性），并将其暴露为bean |
| <util:set> | 创建一个java.util.Set类型的bean，其中包含值或引用 |

### 特殊值注入

除普通使用方式（如下）外，还可以使用util命名空间中的元素

|  |  |
| --- | --- |
| map | 简单值  <map>  <entry key=*"a"* value=*"1"* ></entry>  <entry key=*"b"* value=*"2"* ></entry>  <entry key=*"c"* value=*"3"* ></entry>  </map> |
| 特殊值  <map>  <entry key=*"a"*>  <bean id=*"Stu"* name=*"stu2"* class=*"com.obj.Student"* p:age=*"23"* p:sex=*"true"* p:stuName=*"王五"*></bean>  </entry>  </map> |
| list | 简单值，用list要注意定义类的时候，一定要是list的实现类  <list>  <value>汪峰</value>  <value>邓紫棋</value>  </list> |
| 特殊值：  <list>  <bean name=*"stu1"* class=*"com.obj.Student"* >  <property name=*"stuName"* value=*"张三"*></property>  <property name=*"age"* value=*"18"*></property>  <property name=*"sex"* value=*"true"*></property>  </bean>  </list> |

如果需要引用**配置文件**中的值，最好使用${……}

import引入其他bean.xml配置文件

<import resource="classpath:**com/obj/Beans.xml**" />红字体替换成src目录下的文件路径+文件全名称

## 使用Maven配置jar包

## $与#

#{}：spring用#符号来支持SpEL运算，支持一下运算操作。

|  |  |
| --- | --- |
| 比较运算 | value=*"#{22 > 3}"* |
| 算术运算 | value=*"#{10-2}"* |
| 三元运算 | value=*"#{22 > 3 ? 10:1}"* |
| 字符串运算 | value=*"#{'abcdefg'.toUpperCase().indexOf('E')}"* |
| 调用实例方法 | value=*"#{bo1.getTime()}"* |

${}：spring引用外部资源（一般指properties文件）中的内容。

注：${}和#{}联用，相互并不影响。如：value=*"#{'${name}'.toUpperCase()}"*

## 注解配置

|  |  |  |
| --- | --- | --- |
| 注解名 | 解释 | 备注 |
| @Autowired | spring会自动寻找名称一样的bean装配进来  required=false 允许null值  required=true 不允许null值 |  |
| @Configuration | 配置Bean，相当于xml配置中的<Beans></Beans>标签，具备动态代理功能 |  |
| @ComponentScan | Spring会扫描这个包以及这个包下所有的子包，查找带有@Component注解的类加入bean容器。使用了此注解，xml中就可以不用配置  <context:component-scan base-package=*""*></context:component-scan>了 |  |
| @Component("…") | 组件注解（通用注解），applicationContext.xml中需开启扫描配置（在引号中写上 包路径）  <context:component-scan base-package=*""*></context:component-scan>  @Controller：controller层，@RestController（推荐使用） = @ ResponseBody + @Controller  @Service：service层  @Respository：实现层即Dao层 | 详细信息见注解详解 |
| @Bean | @Bean (name = "bean名称", initMethod = "初始化方法", destroyMethod = "销毁前执行方法")引用第三方的对象对当前对象的装配（比如jar包中的类） |  |
| @Scope | @Scope(value = "", proxyMode=ScopedProxyMode.No)：供spring判断被注释的Bean生命周期（proxyMode=ScopedProxyMode.No一般不常用）  singleton：ConfigurableBeanFactory. SCOPE\_SINGLETON  prototype：ConfigurableBeanFactory. SCOPE\_PROTOTYPE  request：WebApplicationContext.SCOPE\_REQUEST  session：WebApplicationContext.SCOPE\_SESSION |  |
| @ImportResource | @ImportResource("classpath:…")：Javaconfig引入xml配置 |  |

@Qulifier指定注入的bean的名称，常和@Autowired搭配使用，这样就不需要将属性名和注入值的名称一样了；如：

@Autowired(required = true)

**@Qualifier("student2")//注入student2的Student Bean**

private Student stu;

@Resource(name="…")指定对象的名称注入，相当于@Autowired+@Qualifier。[推荐使用]

## 注解详解

@Autowired(required = false/true)

|  |  |
| --- | --- |
| required | 默认为true。如果上下文没有匹配的bean，spring会默认抛出异常，屏蔽异常改用false |
|  |  |

@Configuration(value="…", proxyBeanMethods="…")：

显式配置bean，写在类名上，常在spring套用第三方库时使用。

|  |  |
| --- | --- |
| value |  |
| proxyBeanMethods |  |

@Bean：创建类的实例

|  |  |
| --- | --- |
| value | 默认为方法的名称（首字母小写）。加上value属性，则套用value中的值作为bean ID。  @Bean( value="student2")  **public** Student student() {  **return** **new** Student("jack",19);  } |
| autowire |  |

@ ComponentScan(

value="…",

basePackages={"…","…","…",…},

basePackageClasses={…})：扫描器

|  |  |
| --- | --- |
| value | 需要明确设置基础包的情况下，使用value属性指定包路径向spring说明扫描哪一个具体的包下带有@Component注释的类 |
| basePackages | 以包路径通知spring扫描哪些包，如  @ComponentScan(basePackages= {"com.scanner","com.scanner2"}) |
| basePackageClasses | 以包中所含的类通知spring扫描包含这些类的包，如  @ComponentScan(basePackageClasses= {Snippet.**class**,CreateBeansFactory.**class**}) |
|  |  |
|  |  |

@Component( value = "…" )：组件

|  |  |
| --- | --- |
| value | 若不加入value属性，默认情况下spring将套用被修饰的类的“类名”作为bean ID（类名的首字母将小写），加入value属性后spring以value值为ID |
|  |  |

## Bean生命周期

创建🡪使用🡪销毁

|  |  |
| --- | --- |
| bean作用范围 | |
| scope=”request”请求 | 每次HTTP request请求都会创建一个新的Bean，适用于Web |
| scope=”session”会话 | 一个HTTP Session共享一个Bean，不同Session用不同的Bean，适用于Web |
| scope=”prototype”原型 | 每次从容器中调用Bean时，都返回一个新的实例，即每次调用getBean()时相当于执行newBean()操作 |
| scope=”singleton”单例 | 在spring IOC容器仅存在一个Bean实例——单例；默认 |

## 引入Xml配置文件

eg: <import resource=*"classpath:bean.xml"*/>

<import resource="classpath:文件路径"/> 引入。随着应用的不断发展，bean配置势必会变得更加繁杂，这些bean若是放到一个文件中管理就会越来越繁琐。很多项目会将bean拆分成多个分类，每个类一个个bean文件；多个bean文件到最后还是需要合并到一起的，所用的就是上述的标签（如下图）。

数据库配置

工具类配置

实体

配置

其他

配置

applicationConfig.xml

import

如果实体配置文件需要使用工具配置文件中定义的bean中的属性或就是这个bean做为value，也可以直接使用，而不用引入。

## 引入Properties配置文件

格式：key:value 或 key=value（注：字符串不需要打引号也可以识别，换行不需要打“;”）

name:jack

age:10

引入：applicationContext.xml文件中加入

<context:property-placeholder location="classpath:文件路径"/>，使用格式${key}。

# AOP

通知：就是你想要的功能，也就是所说的身份验证，日志等功能的具体实现。

连接点：spring允许你使用通知的地方，基本每个方法的前后，环绕（执行方法前执行一次，执行方法后执行一次），或是抛出异常时，这些地方都是连接点。

切入点：基于上面说的连接点的基础上来解释切入点，你的一个类中如果有10个方法，那么连接点就会有40个了吧，但是你并不希望所有方法都使用通知，只使用其中几个方法在调用前后或者抛出异常后调用通知干点什么事情，那么就用切入点来标识需要用得着的连接点。

切面：切入点和通知的结合，看！就没有连接点啥事~那就是一个利于理解的名词

引入：允许我们向现有的类添加新方法属性，这不就是把切面用到目标类中了嘛

织入；把切面应用到目标创建新代理对象的过程，共3种方式。

目标：被通知的对象，也就是使用切面的类

AOP依赖jar包：aspectjrt.jar，aspectjweaver.jar，aopalliance-1.0.jar，spring-aop，spring-aspect

支持以下的通知

前置通知，后置通知，后置异常通知，后置返回通知，环绕通知

例子如下：

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.After;

**import** org.aspectj.lang.annotation.AfterReturning;

**import** org.aspectj.lang.annotation.AfterThrowing;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

**import** org.springframework.stereotype.Component;

/\*\*

\* 通知

\* \*/

@Component

@Aspect

**public** **class** AspectPro {

//前置通知

@Before(value ="execution(\* com.aop.User.add(..) )")//被加强的方法

**public** **void** before() {

System.***out***.println("before...");

}

//后置通知

@After(value = "execution(\* com.aop.User.add(..) )")

**public** **void** after() {

System.***out***.println("after...");

}

//后置返回通知

@AfterReturning(value = "execution(\* com.aop.User.add(..) )")

**public** **void** afterReturning() {

System.***out***.println("afterReturning...");

}

//后置异常通知

@AfterThrowing(value = "execution(\* com.aop.User.add(..) )")

**public** **void** afterThrowing() {

System.***out***.println("afterThrowing:");

}

//环绕通知

@Around(value = "execution(\* com.aop.User.delete(..) )")

**public** **void** around(ProceedingJoinPoint point) **throws** Throwable {

System.***out***.println("around 执行前");

point.proceed();

System.***out***.println("around 执行后");

}

}

抽取相同切入点

@Component

@Aspect

**public** **class** AspectPro {

@Pointcut(value = "execution(\* com.aop.User.add(..) )") //切入点

**public** **void** Pointcut() {}

//前置通知

@Before(value ="Pointcut()")//套用切入点

**public** **void** before() {

System.***out***.println("before...");

}

//后置通知

@After(value = "Pointcut()")//套用切入点

**public** **void** after() {

System.***out***.println("after...");

}

通知优先级：如果一个方法被多个方法增强，就得对增强方法的优先级配置，使用@Order，value从0开始。value值越小，优先级越高，越先执行。

第一个切面类

@Component

@Order(value = 0)

@Aspect

**public** **class** AspectPro {

//前置通知0

@Before(value ="Pointcut()")//被加强的方法

**public** **void** before() {

System.***out***.println("before...");

}

}

第二个切面类

@Component

@Order(value = 1)

@Aspect

**public** **class** AspectPro {

//前置通知1

@Before(value ="Pointcut()")//被加强的方法

@Order(value = 0)

**public** **void** before1() {

System.***out***.println("before1...");

}

}

拦截传入参数

//前置通知0

@Before(value ="Pointcut()")//被加强的方法

**public** **void** before(JoinPoint joinPoint) {

System.***out***.println("before...");

}

//后置通知

@After(value = "Pointcut()")

**public** **void** after(JoinPoint point) {

Object[] objects = point.getArgs();

**for** (Object object : objects) {

System.***out***.println(object.toString());

}

System.***out***.println("after...");

}

Spring自身支持了继承接口实现AOP

|  |  |  |  |
| --- | --- | --- | --- |
| 注释 | 接口 | 实现方法 | 注意 |
| 环绕 | AfterReturningAdvice | //arg0 切点参数值，这些参数值是按照切点参数顺序排列的  invoke(MethodInvocation arg0) | 1. arg0.process（）为执行切点，此段代码前写前置程序，此段代码后执行后置程序。 2. getArguments()获取切点参数，用object[]接收 |
| 前置 | MethodBeforeAdvice | //method切点,args:切点参数值  before(Method method, Object[] args, Object target) |  |
| 后置 | AfterReturningAdvice | //returnValue:,method:切点，args:切点参数值  afterReturning(Object returnValue, Method method, Object[] args, Object target) |  |
| 异常 | IntroductionInterceptor | invoke(MethodInvocation invocation)  implementsInterface(Class<?> intf) |  |

Spring也支持AspectJ方式编写接口

## XML配置

|  |  |  |
| --- | --- | --- |
| <bean id=*"cut"* class=*"com.aop.Cut"*></bean>  <bean id=*"person"* class=*"com.aop.Person"*></bean>  <aop:config>  <aop:pointcut expression=*"execution(\* com.aop.Person.\*())"* id=*"cf"* />  <aop:aspect ref=*"cut"*>  <!--method:配置的aspect中的方法名（切面），pointcut-ref：切点-->  <aop:before method=*"before"* pointcut-ref=*"cf"*/>  <aop:after method=*"after"* pointcut-ref=*"cf"* />  <aop:around method=*"around"* pointcut-ref=*"cf"*/>  <aop:after-returning method=*"afterRunning"* pointcut-ref=*"cf"*/>  <aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"cf"*/>  </aop:aspect>  </aop:config>  </beans> | | |
| <aop:pointcut> | 切点 | **—** |
| <aop:aspect> | 切面配置 | **—** |
| <aop:before> | 前置通知 |
| <aop:after> | 后置通知：无论是否异常都会执行 |
| <aop:after-returning> | 后置通知（切点无异常时调用） |
| <aop:after-throwing> | 后置通知（切点异常时调用） |
| <aop:around> | 环绕通知，切点异常后不执行通知后面的代码 |

注意：环绕通知方法必须加参数ProceedingJoinPoint point用以获取切点方法，并用point.proceed()执行切点方法【环绕通知还需加throws Throwable将异常抛出】，不太支持使用xml配置开发AOP。

## 注解配置

|  |  |  |
| --- | --- | --- |
| @Aspect | 加在切面类上 |  |
| @Before(value="…") | 前置通知value值  execution(public void com.aop.Person.save()) |
| @After(value="…") | 后置通知：无论是否异常都会执行 |
| @Around (value="…") | 后置通知（切点无异常时调用） |
| @AfterReturning(value="…") | 后置通知（切点异常时调用） |
| @AfterThrowing (value="….") | 环绕通知，切点异常后不执行通知后面的代码 |

【注】applicationContext.xml中需开启织入配置<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

# 整合Mybaties

# 整合Redis

准备工作