## 基础

@component == @Named Spring扫描到后会创建并管理此Bean

@ComponentScan注解启用了组件扫描，默认扫描与配置类相同的包

@ComponentScan注解== <context:component-scan>XML配置

<context:annotation-config> 和 <context:component-scan>的区别：

<context:annotation-config> 是用于激活那些已经在spring容器里注册过的bean上面的注解，默认会注册AutowiredAnnotationBeanPostProcessor、CommonAnnotationBeanPostProcessor、

PersistenceAnnotationBeanPostProcessor，RequiredAnnotationBeanPostProcessor 这四个bean，

这4个Bean都实现了BeanPostProcessor接口，例如AutowiredAnnotationBeanPostProcessor会在spring启动，也就是Bean实例化的时候去查询带有@Autowired注解的属性、set方法或者构造方法，将对象通过反射注入到带有注解的方法或属性中去，

而<context:component-scan>不仅有<context:annotation-config>所具有的功能外，还有将指定包下的类交给spring管理，会扫描@Component,以下他的子注解@Controller,@Respository,@Service

@Autowired(required=false) == @Inject required=false为null会报空指针

@Configuration 表名此类是一个配置类，包含在spring应用上下文中如何创建bean的细节

@Bean 告诉spring这个方法将返回一个上下文中的bean

@profile(“dev”) 3.2开始可以应用于方法上

@Conditional 4.0后才有（条件化创建bean。用在带有@Bean的方法上）

@Conditional(AA.class) AA implement Condition 实现Condition的matches方法，如果返回true。才会创建此bean

@Primary 可以和@Component 一起用，表示此bean为首选bean（当注入的接口有三个实现类，又没有标示注入id时就会奏效）

@Qualifier(“zfx”) 和@Autowired一起用，表示注入的bean ID为zfx

和@Bean、@Component一起用限定bean ID，默认是将类名首字母小写

@Scope作用于类上表示Bean的作用域 Singleton(默认)、Prototype、Session、Request

@Scope(value=”” ,proxyMode=””);

当单例的Bean中将注入Scope是session或request的bean时候，因为session或request的bean在请求的时候才会有，所以spring会将目标bean的代理注入，等真正调用的时候才会委托给真正的bean。proxyMode的值是ScopedProxyMode.interfaces或者ScopedProxyMode.target\_class 表示基于接口代理或者基于类生成代理（@Scope是类的话只能使用CGLib生成代理）

<aop:scoped-proxy> == @Scope的proxyMode属性 ，默认使用CGLib,可以将 proxy-target-class=”false” 则使用JDK代理，这种方式支持注解方式@Aspect, XML配置方式则使用<aop:config proxy-target-class="false" />来配置

运行时注入外部的值

@Configuration

@PropertySource(“classpath:/aa/bb/app.properties”)

会加载到spring的Enviroment中，所以我们可以

@Autowired

Enviroment env ;

env.getProperty(“xx”)

属性占位符：

XML中，${xx} , java中例如构造参数 @Value(${xx}) 获取properties中的值

使用占位符之前必须配置一个Spring的Bean

PropertySourcesPlaceholderConfigurer（荐，基于Enviroment）或PropertyPlaceholderConfigurer

*PropertySourcesPlaceholderConfigurer* == <context:property-placeholder>

SPel表达式:

字面值：#{1}

引用bean、属性、方法：#{beanId}、#{beanId.name}、#{beanId.say()}

类型：#{T(java.lang.Math.random())}

运算符：#{1\*1}

正则表达式：#{admin.email matches ‘[a-Za-z]’}

## 切面编程

springAOP只支撑方法级别的拦截

spring在运行期把切面织入到spring管理的bean中，直到应用程序需要用到被代理的bean时，spring才会创建代理对象，ApplicationContext从BeanFactory加载所有bean的时候，spring就会创建被代理的对象

SpringAOP使用Aspect切点表达式语言定义切点

例：切点表达式 execution(\* concert.Performance.perform(..))

\*：返回值

concert.Performance：类全路径

perform：方法名

..：参数

@Aspect标注一个POJO类是一个切面。

声明切面中的方法为通知：@After、@AfterReturning、@AfterThrowing、@Around、@Before

切面中声明公共的切点：@PointCut(“execution(\* xxx(..))”)

Public void zfx(){}

在方法上应用切点：

@Before(“zfx()”)

Public void before(){xxxxx}

Aspect需要配置之后才会自动代理：

@Configuration

@EnableAspectJAutoProxy //启动Aspect自动代理

@ComponentScan

Public class Config{

@Bean

Xxxxx 返回切面类的方法

}考虑；‘,l

@EnableAspectAutoProxy == <aop:aspect-autoproxy>

Spring只是使用@Aspect作为创建切面的指导，切面依然是基于代理的。

因为Aspect切面是由AspectJ在运行时创建的，不是由spring创建管理的，但是所有的AspectJ切面都提供了一个静态的aspectOf()方法，该方法返回切面的一个单例，此时这个切面就相当于一个bean交给spring管理，我们也可以将其他bean注入到此切面中，例如：

<bean class=’com.Hello’ factory-method=’aspectOf’ >

<property name=’’ ref=’’></ property>

</bean>

<aop:declare-parents> 通过切面引入新功能

## Spring持久化

## springMVC

ServletContainerInitializer

SpringServletContainerInitializer

WebApplicationInitializer

我们希望DispatcherServlet加载包含Web组件的bean，如控制器、试图解析器以及处理器映射，而ContextLoaderListener要加载应用中其他bean。这些bean通常是驱动应用后端的中间层和数据层组件

springMvc的配置类

@Configuration

@EnableWebMvc

Public class WebConfig{

}

//接收参数名称

Public String spittle(@RequestParam(“spittleId”)long spittleId){

}

//属性占位符

Public String spittle(@PathVariable(“spittleId”)long spittleId , Model model){

}

## <bean id=’id’ name=’alias’ />

从定义来看，name或id如果指定它们中的一个时都作为“标识符”，那为什么还要有id和name同时存在呢？这是因为  
当使用基于XML的配置元数据时，在XML中id是一个真正的XML id属性，因此当其他的定义来引用这个id时就体现出id  
的好处了，可以利用XML解析器来验证引用的这个id是否存在，从而更早的发现是否引用了一个不存在的bean，而使  
用name，则可能要在真正使用bean时才能发现引用一个不存在的bean。

## Bean的实例化方式

1，

<bean name="bean2" class="cn.javass.spring.chapter2.HelloImpl2">  
<!-- 指定构造器参数 -->  
<constructor-arg index="0" value="Hello Spring!"/>

</bean>,

2，

<!-- 使用静态工厂方法 -->

<bean id="bean3" class="cn. HelloApiStaticFactory" factorymethod="newInstance">

<constructor-arg index="0" value="Hello Spring!"/><!-- 方法参数 -->

</bean>

3，

使用实例工厂方法实例化Bean，使用这种方式不能指定class属性，此时必须使用factory-bean属性来指定工厂Bean，factory-method属性指定实例化Bean的方法，而且使用实例工厂方法允许指定方法参数，方式和使用构造器方  
式一样

<!—1、定义实例工厂Bean -->

<bean id="beanInstanceFactory" class="cn.javass.spring.chapter2.HelloApiInstanceFactory"/>

<!—2、使用实例工厂Bean创建Bean -->

<bean id="bean4" factory-bean="beanInstanceFactory" factory-method="newInstance">

<constructor-arg index="0" value="Hello Spring!" />

</bean>

## Spring解决循环依赖

1. 构造器循环依赖无法解决，只能抛出BeanCurrentlyInCreationException错误，正在构造的bean会放在“当前创建Bean池”中。
2. Setter循环依赖：

具体步骤如下：

1、Spring容器创建单例“circleA” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory ”用于

返回一个提前暴露一个创建中的Bean，并将“circleA” 标识符放到“当前创建Bean池”；然后进行setter注入

“circleB”；

2、Spring容器创建单例“circleB” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于

返回一个提前暴露一个创建中的Bean，并将“circleB” 标识符放到“当前创建Bean池”，然后进行setter注入

“circleC”；

3、Spring容器创建单例“circleC” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory ”用于

返回一个提前暴露一个创建中的Bean，并将“circleC” 标识符放到“当前创建Bean池”，然后进行setter注入

“circleA”；进行注入“circleA”时由于提前暴露了“ObjectFactory”工厂从而使用它返回提前暴露一个创建中的

Bean；

4、最后在依赖注入“circleB”和“circleA”，完成setter注入。

对于“singleton”作用域Bean，可以通过“setAllowCircularReferences(false);”来禁用循环引用

## depends-on

depends-on是指指定Bean初始化及销毁时的顺序，使用depends-on属性指定的Bean要先初始化完毕后才初始化当前Bean，由于只有“singleton”Bean能被Spring管理销毁，所以当指定的Bean都是“singleton”时，使用depends-on属性指定的Bean要在指定的Bean之后销毁。如用在文件读取之前要先初始化加载文件的类

## 自动装配

自动装配就是指由Spring来自动地注入依赖对象，无需人工参与。

目前Spring3.0支持“no”、“byName ”、“byType”、“constructor”四种自动装配，默认是“no”指不支  
持自动装配的

自动装配的好处是减少构造器注入和setter注入配置，减少配置文件的长度。自动装配通过配置<bean>标签的“autowire”属性来改变自动装配方式。

如果ByType匹配到两个适合的bean候选者，有两种情况可以确定候选者，第一是在自动匹配的候选者加上属性autowire-candidate="false" ，将此bean踢出出去，第二种方式是在候选bean上加primary="true" 属性，将此bean作为候选的首选注入bean。

“constructor”：通过设置Bean定义属性autowire="constructor"，功能和“byType”功能一样，根据类型注入构造器参数，只是用于构造器注入方式

数组类型、集合的setter属性自动装备分两种情况：1，属性是接口类型的如List<Hello> list 会将所有Hello类型的bean注入到list中，2，属性是具体实现类型的如ArrayList<Hello> list 将选择一个类型为ArrayList的bean注入，而不是所有Hello类型的bean都会注入

### 依赖检查

依赖检查：用于检查Bean定义的属性都注入数据了，不管是自动装配的还是配置方式注入的都能检查，如果没有注入数据将报错，从而提前发现注入错误，只检查具有setter方法的属性。现在spring已经不推荐使用这种方式，用使用spring-beans-2.5.xsd才能看到此标签，现在推荐使用@Required 注解方式

dependency-check="objects" none，objects，simple，all

objects：检查除基本类型外的依赖对象

simple：对基本类型进行依赖检查，包括数组类型，其他依赖不报错

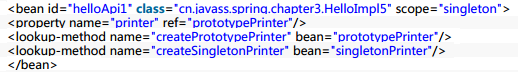
all：对所以类型进行依赖检查

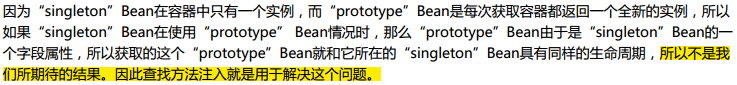
### 方法注入

所谓方法注入其实就是通过配置方式覆盖或拦截指定的方法，通常通过代理模式实现。Spring提供两种方法注入：查找方法注入和方法替换注入。

#### 查找方法注入

<lookup-method name="方法名" bean="bean名字"/>配置；其中name属性指定方法名，bean属性指定方法需返回的Bean





#### 替换方法注入

也叫“MethodReplacer”注入，和查找注入方法不一样的是，他主要用来替换方法体。通过  
首先定义一个MethodReplacer接口实现



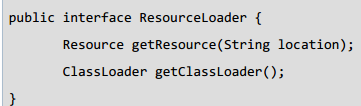


## Resource接口

Spring提供的resource实现ByteArrayResource、InputStreamResource 、FileSystemResource 、UrlResource 、ClassPathResource、ServletContextResource、VfsResource等等。。

### ResourceLoader接口

ResourceLoader接口用于返回Resource对象；其实现可以看作是一个生产Resource的工厂类。



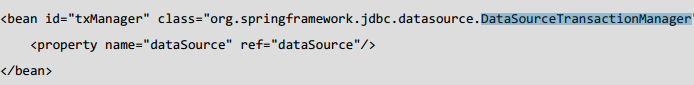
ResourceLoader有一个默认实现类DefaultResourceLoader，AbstractApplicationContext是DefaultResourceLoader子类，所以ApplicationContext的子类都可以用此来加载资源。加载资源使用前缀来区分getResource方法返回具体的哪个Resource实现

ResourceLoaderAware是一个标记接口，用于通过ApplicationContext上下文注入ResourceLoader

## Spring声明式事务

事务管理器的父类 PlatformTransactionManager，

最经常使用的事务管理器



### Xml配置事务管理

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="save\*" propagation="REQUIRED" isolation="READ\_COMMITTED"/>

<tx:method name="\*" propagation="REQUIRED" isolation="READ\_COMMITTED" read-only="true"/>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut id="serviceMethod" expression="execution(\* cn..chapter9.service..\*.\*(..))"/>

<aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice"/>

</aop:config>

<tx:advice>：事务通知定义，用于指定事务属性，其中“transaction-manager”属性指定事务管理器，  
并通过< tx:attributes >指定具体需要拦截的方法；  
• <tx:method name="save\*">：表示将拦截以save开头的方法，被拦截的方法将应用配置的事务属性：  
propagation="REQUIRED"表示传播行为是Required，isolation="READ\_COMMITTED"表示隔离级别是  
提交读；  
• <tx:method name="\*">：表示将拦截其他所有方法，被拦截的方法将应用配置的事务属性：  
propagation="REQUIRED"表示传播行为是Required，isolation="READ\_COMMITTED"表示隔离级别是  
提交读，read-only="true"表示事务只读；  
• <aop:config>：AOP相关配置：  
• <aop:pointcut/>：切入点定义，定义名为"serviceMethod"的aspectj切入点，切入点表达式  
为"execution(\* cn..chapter9.service..\*.\*(..))"表示拦截cn包及子包下的chapter9. service包及子包下的任何  
类的任何方法；  
• <aop:advisor>：Advisor定义，其中切入点为serviceMethod，通知为txAdvice。

### @Transactional事务管理

Xml中配置<tx:annotation-driven transaction-manager="txManager"/>，然后在方法上加注解@Transactional就会开启事务