

可以利用类似EL表达式来配置一些属性

<bean class="mycompany.RewardsTestDatabase">

<property name="databaseName"

value="#{systemProperties.databaseName}"/>

<property name="keyGenerator"

value="#{strategyBean.databaseKeyGenerator}"/>

</bean>

This functionality is also available if you prefer to configure your components using annotations:

@Repository

**public** **class** RewardsTestDatabase {

@Value("#{systemProperties.databaseName}")

**public** **void** setDatabaseName(String dbName) { … }

@Value("#{strategyBean.databaseKeyGenerator}")

**public** **void** setKeyGenerator(KeyGenerator kg) { … }

}

可以写一个java类来作为配置文件

@Configuration

<context:component-scan base-package="org.example.config"/>

<util:properties id="jdbcProperties" location="classpath:org/example/config/jdbc.properties"/>

Or you can bootstrap a @Configuration class directly using AnnotationConfigApplicationContext:

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

FooService fooService = ctx.getBean(FooService.class);

fooService.doStuff();

}

You use getBean to retrieve instances of your beans. The ApplicationContext interface has a few other methods for retrieving beans, but ideally your application code should never use them. Indeed, your application code should have no calls to the getBean method at all, and thus no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides for dependency injection for various web framework classes such as controllers and JSF-managed beans.

尽量少使用getBean 而是使用依赖注入

bean的id和name可以不指定,那就只能通过类型来索引它了

<alias name="moduleService" alias="a1"/>

给moduleService起别名

<!-- typed as a java.util.Properties -->

<property name="properties">

<value>

jdbc.driver.className=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mydb

</value>

</property>

idref只的是传id 而不是传实体引用

inner bean 可以没有id和名字

集合

<bean id="moreComplexObject" class="example.ComplexObject">

<*!-- results in a setAdminEmails(java.util.Properties) call --*>

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

<*!-- results in a setSomeList(java.util.List) call --*>

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

<*!-- results in a setSomeMap(java.util.Map) call --*>

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

<*!-- results in a setSomeSet(java.util.Set) call --*>

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

The value of a map key or value, or a set value, can also again be any of the following elements:

bean | ref | idref | list | set | map | props | value | null

bean的init-method方法destroy-method

The following example demonstrates collection merging:

<beans>

<bean id="parent" abstract="true" class="example.ComplexObject">

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.com</prop>

<prop key="support">support@example.com</prop>

</props>

</property>

</bean>

<bean id="child" parent="parent">

<property name="adminEmails">

<!-- the merge is specified on the \*child\* collection definition -->

<props merge="true">

<prop key="sales">sales@example.com</prop>

<prop key="support">support@example.co.uk</prop>

</props>

</property>

</bean>

<beans>

Notice the use of the merge=true attribute on the <props/> element of the adminEmails property of the child bean definition. When the child bean is resolved and instantiated by the container, the resulting instance has an adminEmails Properties collection that contains the result of the merging of the child's adminEmails collection with the parent's adminEmails collection.

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

如果需要指定某个属性为null

需要使用<null/>

而不是value=""; <- 其实是空字符串

使用p命名空间

xmlns:p="http://www.springframework.org/schema/p"

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean name="john-classic" class="com.example.Person">

<property name="name" value="John Doe"/>

<property name="spouse" ref="jane"/>

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person">

<property name="name" value="Jane Doe"/>

</bean>

</beans>

**3.4.2.7 Compound property names**

You can use compound or nested property names when you set bean properties, as long as all components of the path except the final property name are not null. Consider the following bean definition.

<bean id="foo" class="foo.Bar">

<property name="fred.bob.sammy" value="123" />

</bean>

The foo bean has a fred property, which has a bob property, which has a sammy property, and that final sammy property is being set to the value 123. In order for this to work, the fred property of foo, and the bob property of fred must not be null after the bean is constructed, or a NullPointerException is thrown.

自动连线

autowire="byName",还有其他类型比如byType

给某个bean加上autowire-candidate="false"

则该bean不能作为自动连线的候选者(除非类型是byName)

<bean id="moduleService" class="sp.module1.ModuleService" autowire="byType">

</bean>

<bean id="module1Object1" class="sp.module1.Module1Object1" autowire-candidate="false">

<property name="name" value="xzc"/>

</bean>

第二个bean不能被自动连线

当使用byType的时候

给某个bean加上primary="true" 可以使它成为主要的候选者

在spring内部配置bean的过程叫做装配

考虑到某个情况

一个singleton bean A,一个prototype bean B

A的某个方法method(B b)[也可以不带参数,但是还是需要新的B]每次都需要一个新的B

这时候肯定不能简单的用属性将B注入A了(否则每次都是同一个B)

一种又优雅的解决方案是

A去实现ApplicationContextAware 然后每次都使用 getbean("b")获得一个新的B

另一种比较好一点的方法是

在A里声明一个方法//可以是抽象的,那样的话整个类A都要抽象

比如叫做 public abstract b createB();

然后在配置文件里

<bean id="a" class="sp.module1.Lei1">

<lookup-method name="createB" bean="b"/>

</bean>

<bean id="b" class="sp.module1.Lei2" scope="prototype">// prototype别忘记了

</bean>

则会使用cglib动态生成一个子类,重写createB方法

返回值就是 id=b的bean

本质上好像差不多啊?

任意方法替换

<bean id="lei1" class="sp.module1.Lei1">

<replaced-method name="createLei2" replacer="repl"/>

//利用repl将createLei2方法替换掉

</bean>

<bean id="repl" class="sp.module1.MethodReplacer1"/>

<bean id="lei2" class="sp.module1.Lei2" scope="prototype">

</bean>

repl类的定义

**public** **class** MethodReplacer1 **implements** MethodReplacer ,ApplicationContextAware{

**public** Object reimplement(Object obj, Method method, Object[] args) **throws** Throwable {

**return** applicationContext.getBean( "lei2" );

}

**private** ApplicationContext applicationContext;

**public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {

**this**.applicationContext=applicationContext;

}

}

replaced-method下可以指定arg-type类判断重载

You can use one or more contained <arg-type/> elements within the <replaced-method/> element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match java.lang.String:

java.lang.String

String

Str

使用bean的parent属性可以继承parent的配置

注意如果指定了不存在的属性就会出异常

当有大量的相同配置的时候可以写一个基本的bean

然后其他的bean都去继承它的配置,注意只是继承了它的配置,跟类继承无关(注意如果指定了不存在的属性就会出异常)

**Table 3.4. Bean scopes**

| **Scope** | **Description** |
| --- | --- |
| [singleton](file:///E:\Java\Java%20EE\libs%20and%20other\libs\Spring\spring-framework-3.0.0.RELEASE\docs\spring-framework-reference\html\beans.html#beans-factory-scopes-singleton) | Scopes a single bean definition to a single object instance per Spring IoC container. |
| [prototype](file:///E:\Java\Java%20EE\libs%20and%20other\libs\Spring\spring-framework-3.0.0.RELEASE\docs\spring-framework-reference\html\beans.html#beans-factory-scopes-prototype) | Scopes a single bean definition to any number of object instances. |
| [request](file:///E:\Java\Java%20EE\libs%20and%20other\libs\Spring\spring-framework-3.0.0.RELEASE\docs\spring-framework-reference\html\beans.html#beans-factory-scopes-request) | Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext. |
| [session](file:///E:\Java\Java%20EE\libs%20and%20other\libs\Spring\spring-framework-3.0.0.RELEASE\docs\spring-framework-reference\html\beans.html#beans-factory-scopes-session) | Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| [global session](file:///E:\Java\Java%20EE\libs%20and%20other\libs\Spring\spring-framework-3.0.0.RELEASE\docs\spring-framework-reference\html\beans.html#beans-factory-scopes-global-session) | Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext. |

util空间下的集合bean

几个接口InitializingBean DisposableBean 对应init-method destroy-method 和 [PostConstruct and @PreDestroy annotations](file:///E:\Java\Java%20EE\libs%20and%20other\libs\Spring\spring-framework-3.0.0.RELEASE\docs\spring-framework-reference\html\beans.html#beans-postconstruct-and-predestroy-annotations).

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

初始化顺序

* Methods annotated with @PostConstruct
* afterPropertiesSet() as defined by the InitializingBean callback interface
* A custom configured init() method

Destroy methods are called in the same order:

* Methods annotated with @PreDestroy
* destroy() as defined by the DisposableBean callback interface
* A custom configured destroy() method

BeanNameAware

<bean id="inheritedTestBeanWithoutClass" abstract="true">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBeanWithoutClass" init-method="initialize">

<property name="name" value="override"/>

<*!-- age will inherit the value of 1 from the parent bean definition--*>

</bean>

连class都不指定,需要指定为abstract 然后就可以用作parent

placeholder

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations" value="classpath:com/foo/jdbc.properties"/>

</bean>

<bean id="dataSource" destroy-method="close"

class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

The actual values come from another file in the standard Java Properties format:

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql:*//production:9002*

jdbc.username=sa

jdbc.password=root

Therefore, the string ${jdbc.username} is replaced at runtime with the value 'sa' and similarly for other placeholder values that match to keys in the property file. The PropertyPlaceholderConfigurer checks for placeholders in most locations of a bean definition and the placeholder prefix and suffix can be customized.

With the context namespace introduced in Spring 2.5, it is possible to configure property placeholders with a dedicated configuration element. You can provide multiple locations as a comma-separated list in the location attribute.

<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>

<bean id="lei2" class="sp.module1.Lei2" scope="prototype" parent="lei1" p:name="${msg}">

<property name="age" value="23" />

</bean>

<context:property-placeholder location="classpath:test.properties" />//可以用逗号分割

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<value>classpath:com/foo/strategy.properties</value>

</property>

<property name="properties">

<value>custom.strategy.class=com.foo.DefaultStrategy</value>

</property>

</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>

<context:property-override location="classpath:over.properties"/>



感觉场景可能是这样的:一个挺庞大的配置文件,不想直接修改

就可以用这招覆盖掉旧值

FactoryBean 用于产生bean的工厂bean

getBean("&myBean") returns the FactoryBean instance itself.

否则getBean("myBean");

容器判断mybean是一个FactoryBean,

就会实际返回它的getObject方法返回的对象

注解

在使用Spring框架中@Autowired标签时默认情况下使用 @Autowired 注释进行自动注入时，Spring 容器中匹配的候选 Bean 数目必须有且仅有一个。当找不到一个匹配的 Bean 时，Spring 容器将抛出

BeanCreationException 异常，并指出必须至少拥有一个匹配的 Bean。

Spring 允许我们通过 @Qualifier 注释指定注入 Bean 的名称，这样歧义就消除了，可以通过下面的方法解决异常。

@Qualifier("XXX") 中的 XX是 Bean 的名称，所以 @Autowired 和 @Qualifier 结合使用时，自动注入的策略就从 byType 转变成 byName 了。

@Autowired 可以对[成员变量](http://baike.baidu.com/view/684821.htm)、方法以及[构造函数](http://baike.baidu.com/view/411124.htm)进行注释，而 @Qualifier 的标注对象是成员变量、方法入参、构造函数入参。

有时候也可以用@Resource代替

@Resource(name="lei2")

@Autowired

默认有required=true

Spring 2.5 introduces further stereotype annotations: @Component, @Service, and @Controller. @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also possible that @Repository, @Service, and @Controller may carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice. Similarly, as stated above, @Repository is already supported as a marker for automatic exception translation in your persistence layer.

You can disable the registration of AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor by including the annotation-config attribute with a value of false.

自定义包扫描的过滤机制

### 3.10.3 Using filters to customize scanning

By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components. However, you can modify and extend this behavior simply by applying custom filters. Add them as include-filter or exclude-filter sub-elements of the component-scan element. Each filter element requires the type and expression attributes. The following table describes the filtering options.

**Table 3.5. Filter Types**

| **Filter Type** | **Example Expression** | **Description** |
| --- | --- | --- |
| annotation | org.example.SomeAnnotation | An annotation to be present at the type level in target components. |
| assignable | org.example.SomeClass | A class (or interface) that the target components are assignable to (extend/implement). |
| aspectj | org.example..\*Service+ | An AspectJ type expression to be matched by the target components. |
| regex | org\.example\.Default.\* | A regex expression to be matched by the target components class names. |
| custom | org.example.MyTypeFilter | A custom implementation of the org.springframework.core.type .TypeFilter interface. |

The following example shows the XML configuration ignoring all @Repository annotations and using "stub" repositories instead.

<beans>

<context:component-scan base-package="org.example">

<context:include-filter type="regex" expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

|  |  |
| --- | --- |
| [Note] | **Note** |
| You can also disable the default filters by providing use-default-filters="false" as an attribute of the <component-scan/> element. This will in effect disable automatic detection of classes annotated with @Component, @Repository, @Service, or @Controller. |

一般很少用到

### 3.10.5 Naming autodetected components

When a component is autodetected as part of the scanning process, its bean name is generated by the BeanNameGenerator strategy known to that scanner. By default, any Spring stereotype annotation (@Component, @Repository, @Service, and @Controller) that contains a name value will thereby provide that name to the corresponding bean definition.

|  |  |
| --- | --- |
| [Note] | **Note** |
| JSR 330's @Named annotation can be used as a means to both detect components and to provide them with a name. This behavior is enabled automatically if you have the JSR 330 JAR on the classpath. |

If such an annotation contains no name value or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. For example, if the following two components were detected, the names would be myMovieLister and movieFinderImpl:

@Service("myMovieLister")

public class SimpleMovieLister {

// ...

}

@Repository

public class MovieFinderImpl implements MovieFinder {

// ...

}

|  |  |
| --- | --- |
| [Note] | **Note** |
| If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the [BeanNameGenerator](http://static.springframework.org/spring/docs/3.0.x/javadoc-api/org/springframework/beans/factory/support/BeanNameGenerator.html) interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner: |

<beans>

<context:component-scan base-package="org.example"

name-generator="org.example.MyNameGenerator" />

</beans>

As a general rule, consider specifying the name with the annotation whenever other components may be making explicit references to it. On the other hand, the auto-generated names are adequate whenever the container is responsible for wiring.

可以自定义命名机制

## 3.11 Java-based container configuration

### 3.11.1 Basic concepts: @Configuration and @Bean

The central artifact in Spring's new Java-configuration support is the @Configuration-annotated class. These classes consist principally of @Bean-annotated methods that define instantiation, configuration, and initialization logic for objects to be managed by the Spring IoC container.

Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The simplest possible @Configuration class would read as follows:

@Configuration

public class AppConfig {

@Bean

public void MyService myService() {

return new MyServiceImpl();

}

}

For those more familiar with Spring <beans/> XML, the AppConfig class above would be equivalent to:

<beans>

<bean id="myService" class="com.acme.services.MyServiceImpl"/>

</beans>

As you can see, the @Bean annotation plays the same role as the <bean/> element. The @Bean annotation will be discussed in depth in the sections below. First, however, we'll cover the various ways of creating a spring container using Java-based configuration.

自定义属性编辑器(类型转换器)

**public** **class** AddressEditor **extends** PropertyEditorSupport {

**private** String splitChar = ",";

**public** **void** setAsText(String text) **throws** IllegalArgumentException {

**if** (text == **null**)

**return**;

String[] ss = text.split( "," );

**if** (ss.length != 2)

**return**;

Address a = **new** Address();

a.setSheng( ss[0] );

a.setShi( ss[1] );

setValue( a );

}

**public** String getSplitChar() {

**return** splitChar;

}

**public** **void** setSplitChar(String splitChar) {

**this**.splitChar = splitChar;

}

}

<!-- 自定义属性编辑器 -->

<bean id="customEditorConfigurer" class="org.springframework.beans.factory.config.CustomEditorConfigurer">

<property name="customEditors">

<map>

<entry key="sp.module2.Address">

<bean class="sp.module2.eidtor.AddressEditor" />

</entry>

</map>

</property>

</bean>