# The IoC container

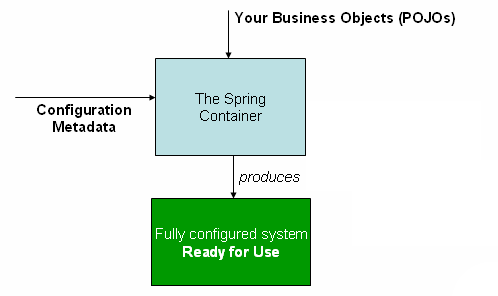
## Introduction to the Spring IoC container and beans(介绍Spring IOC容器和beans)

org.springframework和org.springframework.context是IOC容器的基础的集成包。

[BeanFactory](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html)接口提供了管理所有对象类型的配置和原理. [ApplicationContext](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html) 是 BeanFactory.的接口，它很容易集成到Spring AOP、消息订阅、甚至WebApplicationContext。

## Container overview(容器预览)

org.springframework.context.ApplicationContext代表着整个IOC容器，ApplicationContext做为接口，一般的实现主要有[ClassPathXmlApplicationContext](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) 或者 [FileSystemXmlApplicationContext](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html)，下图就显示，在ApplicationContext作用下，你应用的类在捆绑配置的原数据类型后，产生你系那个要的。



### Configuration metadata

这里介绍下采用XML配置

<?xml version="1.0" encoding="UTF-8"?>

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

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

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

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

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

*<!-- collaborators and configuration for this bean go here -->*

</bean>

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

*<!-- collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions go here -->*

</beans>

### Instantiating a container

 Spring IoC container非常直接容易，只需告知资源所在路径，

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {"services.xml", "daos.xml"});

<?xml version="1.0" encoding="UTF-8"?>

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

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

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

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

*<!-- services -->*

<bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">

<property name="accountDao" ref="accountDao"/>

<property name="itemDao" ref="itemDao"/>

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions for services go here -->*

</beans>

为了简化，也可以直接采用多个xml导入

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

### Using a container

当你拿到ApplicationContext后，可以通过下面方法T getBean(String name, Class<T> requiredType)，检索到实体Bean类

*// create and configure beans*

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {"services.xml", "daos.xml"});

*// retrieve configured instance*

PetStoreService service = context.getBean("petStore", PetStoreService.**class**);

*// use configured instance*

List<String> userList = service.getUsernameList();

## Bean overview(Bean预览)

### Naming beans

Xml配置时，id必须唯一性

在bean定义的外面取别名

<alias name="fromName" alias="toName"/>

### Instantiating beans

**构建构造函数的bean**

<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>

**通过工作方法来构建bean**

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

**public** **class** ClientService {

**private** **static** ClientService clientService = **new** ClientService();

**private** ClientService() {}

**public** **static** ClientService createInstance() {

**return** clientService;

}

}

## Dependencies

### Dependency injection

基于构造函数的DI

**public** **class** SimpleMovieLister {

*// the SimpleMovieLister has a dependency on a MovieFinder*

**private** MovieFinder movieFinder;

*// a constructor so that the Spring container can inject a MovieFinder*

**public** SimpleMovieLister(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// business logic that actually uses the injected MovieFinder is omitted...*

}

基于构造参数类型

<beans>

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

</bean>

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

</beans>

基于成员变量类型

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

基于索引值

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

基于get/set的方式

<bean id="exampleBean" class="examples.ExampleBean">

*<!-- setter injection using the nested ref element -->*

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

*<!-- setter injection using the neater ref attribute -->*

<property name="beanTwo" ref="yetAnotherBean"/>

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

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

### Dependencies and configuration in detail

#### 直接引用

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

*<!-- results in a setDriverClassName(String) call -->*

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>

<property name="username" value="root"/>

<property name="password" value="masterkaoli"/>

</bean>

#### 或者使用p-namespace标签

<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.xsd">

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:driverClassName="com.mysql.jdbc.Driver"

p:url="jdbc:mysql://localhost:3306/mydb"

p:username="root"

p:password="masterkaoli"/>

</beans>

#### 关联到其它bean

*<!-- in the parent context -->*

<bean id="accountService" class="com.foo.SimpleAccountService">

*<!-- insert dependencies as required as here -->*

</bean>

*<!-- in the child (descendant) context -->*

<bean id="accountService" <!-- bean name is the same as the parent bean -->

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target">

<ref parent="accountService"/> *<!-- notice how we refer to the parent bean -->*

</property>

*<!-- insert other configuration and dependencies as required here -->*

</bean>

#### 内敛bean

<bean id="outer" class="...">

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

#### Collections

<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>

#### 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>

#### Null and empty string values

<bean class="ExampleBean">

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

</bean>

<bean class="ExampleBean">

<property name="email">

<null/>

</property>

</bean>

#### XML shortcut with the p-namespace

<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.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>

#### XML shortcut with the c-namespace

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

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

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

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

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

<bean id="bar" **class**="x.y.Bar"/>

<bean id="baz" **class**="x.y.Baz"/>

<!-- traditional declaration -->

<bean id="foo" **class**="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

<constructor-arg value="foo@bar.com"/>

</bean>

<!-- c-namespace declaration -->

<bean id="foo" **class**="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com"/>

</beans>

### Using depends-on

<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">

<property name="manager" ref="manager" />

</bean>

<bean id="manager" class="ManagerBean" />

<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />

### 5.4.4 Lazy-initialized beans

lazy-initialized bean告诉IOC容器当它被第一次请求才被加载，而非像其它的bean在启动时加载完成。

通过lazy-init参数来控制

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>

也可以通过beans元素上来决定

<beans default-lazy-init="true">

*<!-- no beans will be pre-instantiated... -->*

</beans>

### 5.4.6 Method injection

#### Lookup method injection

**public** **abstract** **class** CommandManager {

**public** Object process(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

*<!-- a stateful bean deployed as a prototype (non-singleton) -->*

<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">

*<!-- inject dependencies here as required -->*

</bean>

*<!-- commandProcessor uses statefulCommandHelper -->*

<bean id="commandManager" class="fiona.apple.CommandManager">

<lookup-method name="createCommand" bean="command"/>

</bean>

#### Arbitrary method replacement(函数任意取代)

**public** **class** MyValueCalculator {

**public** String computeValue(String input) {

*// some real code...*

}

*// some other methods...*

}

**/\*\***

**\* meant to be used to override the existing computeValue(String)**

**\* implementation in MyValueCalculator**

**\*/**

**public** **class** ReplacementComputeValue **implements** MethodReplacer {

**public** Object reimplement(Object o, Method m, Object[] args) **throws** Throwable {

*// get the input value, work with it, and return a computed result*

String input = (String) args[0];

...

**return** ...;

}

}

<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">

*<!-- arbitrary method replacement -->*

<replaced-method name="computeValue" replacer="replacementComputeValue">

<arg-type>String</arg-type>

</replaced-method>

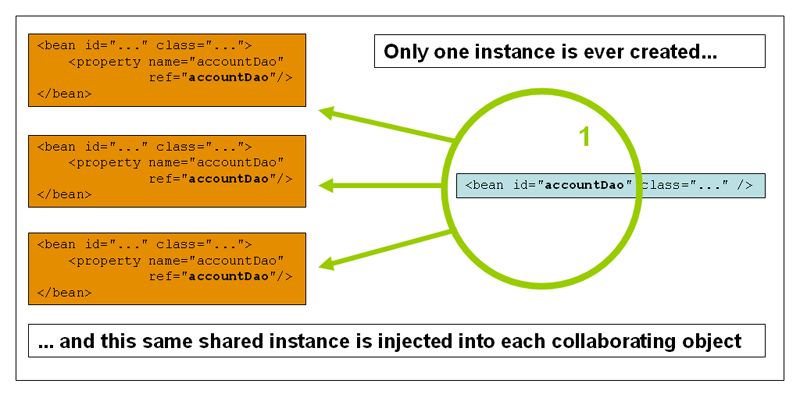
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>

## 5.5 Bean scopes

|  |  |
| --- | --- |
| **Scope** | **Description** |
| [**singleton**](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#beans-factory-scopes-singleton) | **默认值 Spring IoC container定义单一bean** |
| **prototype** | **Spring IoC container定义人艺术的bean** |
| **Request** | **完整的HTTP request的生命周期请求** |
| **Session** | **HTTP Session** |
| **Global session** | **global HTTP Session** |
| **Application** | **ServletContext上下文的生命周期** |

### 5.5.1 The singleton scope



<bean id="accountService" class="com.foo.DefaultAccountService"/>

*<!-- the following is equivalent, though redundant (singleton scope is the default) -->上述两种方式相同*

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

### 5.5.2 The prototype scope

对有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域，DAO通常不会持有任何会话状态，因此应该使用singleton作用域

### 5.5.5 Custom scopes

通过实现org.springframework.beans.factory.config.Scope的接口

<?xml version="1.0" encoding="UTF-8"?>

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

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

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

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

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

http://www.springframework.org/schema/aop

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

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

<property name="scopes">

<map>

<entry key="thread">

<bean class="org.springframework.context.support.SimpleThreadScope"/>

</entry>

</map>

</property>

</bean>

<bean id="bar" class="x.y.Bar" scope="thread">

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

<aop:scoped-proxy/>

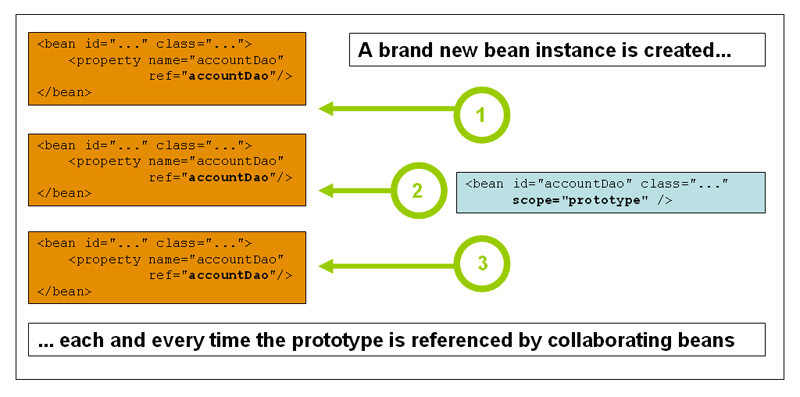
</bean>

<bean id="foo" class="x.y.Foo">

<property name="bar" ref="bar"/>

</bean>

</beans>



## 5.6 Customizing the nature of a bean

### 5.6.1 Lifecycle callbacks(生命周期回调)

管理容器的生命周期，可以通过Spring InitializingBean 和 DisposableBean的afterPropertiesSet()和destroy()去初始化到破坏的整个生命周期的实现。

#### Initialization callbacks

在所有成员到位后，org.springframework.beans.factory.InitializingBean就可以开始初始化工作，它只提供了唯一的方法

void afterPropertiesSet() throws Exception;

Spring虽然可以通过InitializingBean完成一个bean初始化后对这个bean的回调，但是这种方式要求bean实现 InitializingBean接口。一但bean实现了InitializingBean接口，那么这个bean的代码就和Spring耦合到一起 了。通常情况下我不鼓励bean直接实现InitializingBean，可以使用Spring提供的init-method的功能来执行一个bean 子定义的初始化方或者[@PostConstruct](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#beans-postconstruct-and-predestroy-annotations)

<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

#### Destruction callbacks

当容器被回收的时候，通过实现org.springframework.beans.factory.DisposableBean接口也会被回调

**void** destroy() **throws** Exception;

同样的道理，也不推荐使用DisposableBean，而是使用destroy-method或者[@PreDestroy](http://docs.spring.io/spring-framework/docs/4.1.x/spring-framework-reference/htmlsingle/#beans-postconstruct-and-predestroy-annotations)

<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

#### Default initialization and destroy methods

默认的初始化方法和破坏方式，分别为default-init-method和default-destroy-method

<beans default-init-method="init">

<bean id="blogService" class="com.foo.DefaultBlogService">

<property name="blogDao" ref="blogDao" />

</bean>

</beans>

## 5.7 Bean definition inheritance(Bean继承关系定义)

通过parent 变量值，来指明父子类的继承关系，如果类不能够明确地指明类，需要使用abstract

<bean id="inheritedTestBean" abstract="true"

class="org.springframework.beans.TestBean">

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

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

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

**parent="inheritedTestBean"** init-method="initialize">

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

*<!-- the age property value of 1 will be inherited from parent -->*

</bean>

## 5.8 Container Extension Points

Spring IOC容器提供更多插件帮你扩展实现更多bean操作

### 5.8.1 Customizing beans using a BeanPostProcessor(用BeanPostProcessor进行定制配置)

接口作用是：如果我们需要在Spring容器完成Bean的实例化，配置和其他的初始化后添加一些自己的逻辑处理，我们就可以定义一个或者多个BeanPostProcessor接口的实现。

如果你在实现BeanPostProcessor接口的同时，我也建议Ordered 接口也实现

**package** scripting;

**import** org.springframework.beans.factory.config.BeanPostProcessor;

**import** org.springframework.beans.BeansException;

**public** **class** InstantiationTracingBeanPostProcessor **implements** BeanPostProcessor {

*// simply return the instantiated bean as-is*

**public** Object postProcessBeforeInitialization(Object bean,

String beanName) **throws** BeansException {

**return** bean; *// we could potentially return any object reference here...*

}

**public** Object postProcessAfterInitialization(Object bean,

String beanName) **throws** BeansException {

System.out.println("Bean *'" + beanName + "*' created : " + bean.toString());

**return** bean;

}

}

<?xml version="1.0" encoding="UTF-8"?>

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

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

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

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

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

http://www.springframework.org/schema/lang

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

<lang:groovy id="messenger"

script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">

<lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>

</lang:groovy>

*<!--*

*when the above bean (messenger) is instantiated, this custom*

*BeanPostProcessor implementation will output the fact to the system console*

*-->*

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>

打印：

Bean *messenger* created : org.springframework.scripting.groovy.GroovyMessenger@272961

org.springframework.scripting.groovy.GroovyMessenger@272961

### 5.8.2 Customizing configuration metadata with a BeanFactoryPostProcessor

#### Example: the Class name substitution PropertyPlaceholderConfigurer

使用Properties 格式在外面配置文件，然后通过PropertyPlaceholderConfigurer进行替换

标准的Properties 格式

jdbc.driverClassName=org.hsqldb.jdbcDriver

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

jdbc.username=sa

jdbc.password=root

PropertyPlaceholderConfigurer的配置方式

<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>

PropertyPlaceHolderConfigurer不仅仅在你指定的Porperties文件中查找属性， 如果它在其中没有找到你想使用的属性，它还会在Java的系统properties中查找。 这个行为能够通过设置配置中的systemPropertiesMode 属性来定制。这个属性有三个值， 一个让配置总是覆盖，一个让它永不覆盖，一个让它仅在properties文件中找不到的时候覆盖。 请参考 PropertiesPlaceholderConfigurer的JavaDoc获得更多信息。

* never (0):从不检查系统配置
* Fallback (1):如果指定配置文件中没有属性，则检查
* override (2):系统覆盖

### 5.8.3 Customizing instantiation logic with a FactoryBean

* Object getObject()，如果你要获取具体实例对象，需要getObject(&mybean)，而非getObject(mybean)，获取到是mybean的factorybean
* Boolean issingle()
* Class getObjectType()

## 5.9 Annotation-based container configuration(注解的容器配置)

Spring中并未提到注解和XML哪种方式比较好，只是谈到要统一风格，反正spring提供这两种方式，因人而异。

@Autowired是根据类型进行自动装配的。如果当Spring上下文中存在不止一个MovieCatalog类型的bean时，就会抛出BeanCreationException异常;如果Spring上下文中不存在MovieCatalog类型的bean，也会抛出BeanCreationException异常。我们可以使用@Qualifier配合@Autowired来解决这些问题。如下：

**public** **class** MovieRecommender {

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

*@Autowired*

**public** **void** prepare(**@Qualifier("main")**MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

**this**.movieCatalog = movieCatalog;

**this**.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

### 5.9.1 @Required

RequiredAnnotationBeanPostProcessor是一个Spring bean后处理器，它检查带有@Required注解的所有Bean属性是否设置。 Bean后处理器是一类特殊的Spring bean，能够在每个Bean初始化之前执行附加的操作。为了启用这个bean后处理器进行属性检查，必须在Spring IoC容器中注册它。注意，这个bean后处理器只能检查属性是否已经设置，而不能检查属性是否非空。

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Required*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

  为了要求Spring检查序列生成器上所有带有@Required注解的bean属性是否已经设置，必须在IoC容器中注册一个RequiredAnnotationBeanPostProcessor实例。 如果打算使用Bean工厂，就必须通过API注册这个Bean后处理器，否则只能在应用程序上下文中声明。

 如果任何带有@Required的属性未设置，Bean后处理器将抛出一个BeanInitializationException异常。

### 5.9.2 @Autowired

这个是最为传统的方式，可以配置一个或者多个，举个比较非常规的例子

当不能确定 Spring 容器中一定拥有某个类的 Bean 时，可以在需要自动注入该类 Bean 的地方可以使用 @Autowired(required = **false**)，这等于告诉 Spring：在找不到匹配 Bean 时也不报错。

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Autowired(required=false)*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

### 5.9.3 Fine-tuning annotation-based autowiring with qualifiers

**public** **class** MovieRecommender {

*@Autowired*

**@Qualifier("main")**

**private** MovieCatalog movieCatalog;

*// ...*

}

当我们的有相同的bean时候，可以通过结合@Qualifier注解进行配置，指定唯一的bean

### 5.9.4 Using generics as autowiring qualifiers

*@Autowired*

**private** Store<String> s1; *// <String> qualifier, injects the stringStore bean*

*@Autowired*

**private** Store<Integer> s2; *// <Integer> qualifier, injects the integerStore bean*

### 5.9.6 @Resource

Spring也支持 JSR-250 @Resource

@Resource和@Autowired两者都是做bean的注入使用。

其实@Resource并不是Spring的注解，他的包是javax.annotation.Resource 需要导入。但是Spring支持该注解的注入。

共同点：两者都可以写在字段和setter方法上。两者如果都写在字段上，就不需要写写setter方法。

不同点如下：

先来说一说@Autowired

    @Autowired为Spring提供的注解，

    需导入Package:org.springframework.beans.factory.annotation.Autowired;

    只按照byType 注入。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Autowired  private UserDao  userDao;//用于字段上  @Autowired  public void setUserDao(UserDao userDao) {//用于属性的setter方法上       this.userDao= userDao;  } |

    @Autowired注解是按类型装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它required属性为false。如果我们想使用按名称装配，可以结合@Qualifier注解一起使用。如下：

[?](http://my.oschina.net/u/216467/blog/205951)

|  |  |
| --- | --- |
| 1  2 | @Autowired  @Qualifier("userDao")  private PersonDao  personDao; |

 再说说@Resource

@Resource默认按 byName 自动注入,由J2EE提供。

    需导入Package:  javax.annotation.Resource

    @Resource有两个中重要的属性：name和type ，而Spring将@Resource注解的name属性解析为bean的

    名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用

    type属性时则使用 byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。

[?](http://my.oschina.net/u/216467/blog/205951)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Resource(name=“userDao”)  private UserDao  userDao;//用于字段上  @Resource(name=“userDao”)  public void setUserDao(UserDao userDao) {//用于属性的setter方法上       this.userDao= userDao;  } |

    注：最好是将@Resource放在setter方法上。

    @Resource装配顺序

　 (1). 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常;

　 (2). 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常;

　 (3). 如果指定了type，则从上下文中找到类型匹配的唯一bean进行装配，找不到或者找到多个，都会抛出异常;

　 (4). 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配；

    @Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入。

# 5.15  Additional Capabilities of the ApplicationContext(额外功能)

* 通过实现MessageSource接口,可以进行本地化*i18n-style*
* *通过实现*ResourceLoader 接口,能够进入资源文件
* *通过*ApplicationListener 接口发布，通过ApplicationEventPublisher 使用
* *通过*HierarchicalBeanFactory 接口，可以分层加载上下文文件，父子级联的Ioc容器接口，“子容器”可以通过接口访问“父容器”，比如我们做Web开发（MVC架构）时一般控制层（C）的Bean由“子容器”管理，而Service层的Bean由“父容器”管理，这样控制层的Bean可以通过接口访问“父容器”中的Service Bean，而Service Bean不能访问存在于“子容器”中的控制层Bean

### 5.15.1 Internationalization using MessageSource（国际化使用messagesource）

Spring提供2种MessageSource接口，分别为ResourceBundleMessageSource 和 StaticMessageSource

StaticMessageSource 很少被采用，除非动态把信息加载到信息源中

这里简单介绍下使用ResourceBundleMessageSource 的例子

<beans>

<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>format</value>

<value>exceptions</value>

<value>windows</value>

</list>

</property>

</bean>

</beans>

假设你有3种资源在你的classpath下面，分别为format、exceptions、windows，假设其中内容为

# in format.properties

message=Alligators rock!

# in exceptions.properties

argument.required=The {0} argument is required.

通过下面方式

**public** **static** **void** main(String[] args) {

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("message", null, "Default", null);

System.out.println(message);

}

输出结果：Alligators rock!

<beans>

*<!-- this MessageSource is being used in a web application -->*

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="exceptions"/>

</bean>

*<!-- lets inject the above MessageSource into this POJO -->*

<bean id="example" class="com.foo.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

**public** **class** Example {

**private** MessageSource messages;

**public** **void** setMessages(MessageSource messages) {

**this**.messages = messages;

}

**public** **void** execute() {

String message = **this**.messages.getMessage("argument.required",

**new** Object [] {"userDao"}, "Required", null);

System.out.println(message);

}

}

输出结果：The userDao argument is required.

### 5.15.2 Standard and Custom Events（标准和定制的事件）

在处理ApplicationContext 的时候，是通过ApplicationEvent 和ApplicationListener 接口。

|  |  |
| --- | --- |
| Event（事件） | Explanation（解释） |
| ContextRefreshedEvent | 针对于ApplicationContext 初始化或者刷新的时候；像ConfigurableApplicationContext的refresh() XmlWebApplicationContext 支持热部署，GenericApplicationContext不支持 |
| ContextStartedEvent | 针对于ApplicationContext 开始时候，像调用ConfigurableApplicationContext的start()方法时 |
| ContextStoppedEvent | 针对于ApplicationContext 结束时候，像调用ConfigurableApplicationContext的stop()方法时 |
| ContextClosedEvent | 所有singleton都被破坏了，针对于ApplicationContext 关闭时候，像调用ConfigurableApplicationContext的close()方法时 |
| RequestHandledEvent | 该事件只针对于web应用，当http请求结束时候触发 |