# Spring学习笔记

## Spring简介

Spring为企业应用开发提供了一个轻量级的解决方案，是一个从实际开发中抽取出来的框架，它完成了大量开发中的通用步骤，留给开发者的仅仅是与特定应用相关的部分，从而大大提高了企业应用的开发效率。

Spring贯穿表现层、业务层、持久层。Spring并不取代那些已有的框架，而是以高度的开放性与他们无缝整合，其包括：基于依赖注入的核心机制、基于AOP的声明式事务管理、与多种持久层技术的整合，以及优秀的Web MVC框架等。

### Spring优点

* 低侵入式设计，代码污染极低。
* 独立于各种应用服务器。基于Spring框架的应用，可以真正实现Write Once，Run Anywhere的承诺。
* Spring的IOC容器降低了业务对象替换的复杂性，提高了组件之间的解耦。
* Spring的AOP支持允许将一些通用任务如安全、事务、日志等进行集中处理，从而提供更好的复用。
* Spring的ORM和DAO提供了与第三方持久层框架的良好整合，并简化了底层数据库的访问。
* Spring的高度开放性，并不强制应用完全依赖于Spring，开发者可以自由选用Spring框架的部分或全部。

### Spring容器（Core Container）

使用Spring框架时，必须使用Spring Core Container（Spring容器），它代表了Spring框架的核心机制。core、beans、context、expression四个包及其子包组成了Spring容器，主要提供Spring IOC容器支持。expression及其子包提供了Spring Expression Language支持。

### Spring 5.0的变化

* Spring 5.0 对JDK的最低要求是Java8，可以在运行时支持Java9.
* Java8反射增强，Spring 5.0 可以对方法的参数进行更高效的访问。
* 核心接口加入了对Java8 接口支持的默认方法。
* 自带通用的日志封装，不在需要额外的common-loggin日志包。
* 引入@Nullable和@NotNull注解来修饰可空的参数以及返回值，避免运行时导致NPE异常。
* 支持使用组件索引来扫描目标组件，比类路径扫描更高效。
* 支持JetBrains Kotlin语言。
* Web支持已经升级为Servlet 3.1 以及更高版本的规范。

总体看，Spring 5.0 主要升级就是全面基于Java8，并在运行时支持Java9和Servlet 3.1 规范，也为核心IoC容器增加了一些注解，并通过组件索引扫描来提升运行效率。

## Spring入门

### Spring下载安装

Spring是一个独立的框架，不需依赖于任何Web服务器或容器，既可以在独立的JavaSE项目中使用，也可在Java Web项目中使用。

下载

登录站点，下载spring-framework-5.0.2.Release-dist.zip压缩文件

解压缩

文件夹中包含几个子文件夹：

* docs：存放Spring的相关文档，包含开发指南、API参考文档。
* libs：存放JAR包，分为三类：
  + 1.Spring框架class文件的JAR包；
  + 2.Spring框架源文件压缩包，以-sources结尾。
  + 3.Spring框架API文档压缩包，以-javadoc结尾。

整个Spring框架由21个模块组成，Spring为每个模块都提供了三个压缩包。

* schemas：包含Spring各种配置文件的XML Schema文档。
* readme.txt、notice.txt、license.txt等说明性文档。

如果需要发布使用了Spring框架的Java Web项目，还需要将Spring框架的JAR包（共21个JAR包）添加到Web应用的WEB-INF路径下。

常用的jar包有5个：

* + aop：开发AOP。
  + beans：处理bean。
  + context：处理Spring上下文。
  + core：Spring核心。
  + expression：Spring表达式。

为了更好的开发Spring应用，可使用Spring官方提供的自动提示Eclipse插件，或直接下载STS IDE开发工具。

下载地址为：

<https://spring.io/tools/sts/all>

下载最新版本spring tool suite安装即可。

### 使用Spring管理Bean

Spring核心容器就是一个超级大工厂，所有对象（包括数据源、Hibernate SessionFactory等基础性资源）都会被当成Spring核心容器管理对象——Spring把容器中的一切对象统称为Bean。

Spring容器中的Bean，与Java Bean不同。Spring对Bean没有任何要求，对于Spring框架而言，一切Java对象都是Bean。

#### 使用Spring管理Bean的基本步骤

#### 1.定义两个简单的类

**public** **class** Axe {

**public** String chop()

{

**return** "使用斧头砍柴";

}

}

**public** **class** Person {

**private** Axe axe;

//setter方法

**public** **void** setAxe(Axe axe)

{

**this**.axe = axe;

}

**public** **void** useAxe()

{

System.*out*.println("我打算去看点柴火");

/\*

\* 调用axe的chop()方法

\* 表名Person对象依赖于axe对象

\* 这种A对象需要调用B对象方法的情形，被称为依赖

\*/

System.*out*.println(axe.chop());

}

}

#### 创建XML配置文件

Spring使用XML配置文件来管理容器中的Bean。Spring对XML配置文件的文件名没有任何要求，可以随意指定。

<?xml version=*"1.0"* encoding=*"GBK"*?>

<!-- Spring配置文件的根元素，使用spring-beans.xsd语义约束 -->

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

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

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

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

<!-- 配置名为person的Bean，其实现类是com.spring.service.Person类 -->

<bean id=*"person"* class=*"com.spring.service.Person"*>

<!-- 控制调动setAxe()方法，将容器中的axeBean作为传入参数 -->

<property name=*"axe"* ref=*"axe"*/>

</bean>

<!-- 配置名为axe的Bean，其实现类是com.spring.service.Axe类 -->

<bean id=*"axe"* class=*"com.spring.service.Axe"*/>

<!-- 配置名为win的Bean，其实现类是javax.swing.JFrame类 -->

<bean id=*"win"* class=*"javax.swing.JFrame"*/>

<!-- 配置名为date的Bean，其实现类是java.util.Date类 -->

<bean id=*"date"* class=*"java.util.Date"*/>

</beans>

配置文件中<bean.../>元素默认以反射方式来调用该类无参数的构造器。

<bean id=*"person"* class=*"com.spring.service.Person"*>

Spring底层会执行以下格式的代码：

String idStr = ...; //解析<bean.../>元素的id属性得到该字符串值为“person”

//解析<bean.../>元素的class属性得到该字符串值为：com.spring.service.Person

String classStr = ...;

Class clazz = Class.forName(classStr);

Object obj = clazz.newInstance();

//container代表Spring容器

container.put(idStr , obj);

#### 3.创建主程序代码

**public** **class** BeanTest {

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

{

/\*

\* ApplicationContext是Spring容器最常用的接口，该接口有如下两个实现类：

\* ClassPathXmlApplicationContext：从类加载路径下搜索配置文件，并根据配置文件来创建Spring容器。

\* FileSystemXmlApplicationContext:从文件系统的相对路径或绝对路径下去搜索配置文件，并根据配置文件来创建Spring容器。

\*/

//创建Spring容器

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml");

/\*

\* Spring容器获取Bean对象主要有如下两个方法：

\* Object getBean(String id)：根据容器中Bean的id来获取指定Bean，获取Bean之后需要进行强制类型转换。（类加载路径对于Java项目而言，总是稳定的。）

\* T getBean(String name,Class<T> requiredType)：根据容器中Bean的id来获取指定Bean，但该方法带一个泛型参数，因此获取Bean之后无需进行强制类型转换。

\*/

//获取id为person的Bean

Person p = ctx.getBean("person", Person.**class**);

//调用useAxe()方法

p.useAxe();

}

}

Spring框架底层通过反射根据<bean.../>的class属性（class属性的值必须是Bean实现类的完整类名（必须带包名），不能是接口，不能是抽象类（除非有特殊配置），否则Spring无法使用反射创建该类的实例），指定的类名创建一个Java对象，并以<bean.../>元素的id属性的值为key，将改对象放入Spring容器中——这个Java对象就成为了Spring容器中的Bean。

<property name=*"axe"* ref=*"axe"*/>

<property.../>子元素通常作为<bean.../>元素的子元素，它驱动Spring在底层以反射执行一次setter方法。其中name属性值决定执行哪个setter方法，而value或ref决定执行setter方法的传入参数。

* 如果传入参数是基本类型及其包装类、则使用ref属性指定传入参数。
* 如果以容器中其他Bean作为传入参数，则使用ref属性指定传入参数。

Bean一旦创建处理，Spring框架会立即根据<property.../>子元素在底层以反射方式执行一次setter方法。这两步先后执行，中间几乎没有任何间隔。

Spring配置文件中<bean.../>元素的作用：默认驱动Spring在底层调用无参的构造器创建对象。

#### ApplicationContext两个实现类

Spring容器最常用的接口ApplicationContext的两个实现类：

* ClassPathXmlApplicationContext：从类加载路径下搜索配置文件，并根据配置文件来创建Spring容器。
* FileSystemXmlApplicationContext：从文件系统的相对路径或绝对路径下与搜索配置文件，并根据配置文件来创建Spring容器。

对Java项目而言，类加载路径总是稳定的，因此通常总是使用ClassPathXmlApplicationContext创建Spring容器。

#### Spring容器获取Bean对象的两种方法

Spring容器获取Bean对象主要有如下两种方法：

* Object getBean(String id)：根据容器中Bean的id来获取指定Bean，获取Bean后需要进行强制类型转换。
* T getBean(String name , Class<T> requiredType)：根据容器中Bean的id来获取指定Bean，但该方法带一个泛型参数，因此获取Bean之后无需进行强制类型转换。

使用Spring框架最大的改变之一是：程序不再使用new关键字调用构造器创建Java对象，所有的Java对象都由Spring容器负责创建。

## Spring的核心机制：依赖注入

### 什么是依赖关系

如A对象需要调用B对象方法的情形，这种情形被Spring称为依赖，即A对象依赖B对象。对于Java应用而言，它们总是由一些互相调用的对象构成的，Spring把这种互相调用的关系称为依赖关系。

Spring框架的核心功能有两个：

* Spring容器作为工厂，负责创建、管理所有的Java对象，这些对象被称为Bean。
* Spring容器管理容器中Bean之间的依赖关系，Spring使用一种被称为“依赖注入”的方式来管理Bean之间的依赖关系。

依赖注入是一种优秀的解耦方式，依赖注入让Spring的Bean以配置文件组织在一起，而不是以硬编码的方式耦合在一起，甚至不需要使用工厂模式。

### 两种角度看依赖注入

传统模式下，当某个Java对象需要调用另一个Java对象的方法时，有如下两种做法：

* 原始做法：调用者主动创建被依赖对象，然后再调用被依赖对象的方法。（此方法必然导致调用者与被依赖对象实现类的硬编码耦合，非常不利于项目升级维护。）
* 简单工厂模式：调用者先找到被依赖对象的工厂，然后主动通过工厂去获取被依赖对象，最后再调用被依赖对象的方法。（此方法使调用组件需要主动通过工厂去获取被依赖对象，这就会带来调用组件与被依赖对象工厂的耦合。）

#### 控制反转（Inversion of Control , IoC）

从调用者角度来看，使用Spring框架之后，调用者无需主动获取被依赖对象，调用者只要被动接受Spring容器为调用者的成员变量赋值即可，调用者获取被依赖对象的方式由原来的主动获取，变成了被动接受——这就是由Rod Johnson提出的控制反转。

其实，反转的是获取对象的方式。

#### 依赖注入（Dependency Injection）

从Spring容器的角度来看，Spring容器负责将被依赖对象赋值给调用者的成员变量——相当于为调用者注入它以来的实例，因此Martine Fowler将这种方式称为依赖注入。

因为Spring将被依赖对象注入给了调用者，所以调用者无需主动获取被依赖对象，只要被动等待Spring容器注入即可。由此可见，控制反转和依赖注入其实是同一个行为的两种表达，只是描述的角度不同而已。

### 依赖注入三种方式

依赖注入通常有如下两种：

* 设值注入：IoC容器使用成员变量的setter方法来注入被依赖对象。
* 构造注入：IoC容器使用构造器来注入被依赖对象。
* P命名空间注入：Ioc容器使用P命名空间方法来注入被依赖对象。

#### 设值注入

Bean与Bean之间的依赖关系由Spring管理，Spring采用setter方法为目标Bean注入所依赖的Bean，这种方式被称为设值注入。

<property name=*"axe"* ref=*"axe"*/>

Spring会自动检测每个<bean.../>定义里的<property.../>元素定义，Spring会在调用默认的构造器创建Bean实例之后，立即调用对应的setter方法为Bean的成员变量注入值。

每个Bean的id属性是该Bean的唯一标识，程序通过id属性值访问Bean，Spring容器也通过Bean的id属性值管理Bean与Bean之间的依赖。

设值注入是指IoC容器通过成员变量的setter方法来注入依赖对象。这种注入方式简单、直观。

Spring推荐面向接口编程，可以更好地让规范和实现分离从而提供更好的解耦。不管是调用者，还是被依赖对象，都应该为之定义接口，程序应该面向它们的接口，而不是面向实现类编程，以便程序后期的升级、维护。

在配置文件中，Spring配置Bean实例通常会指定两个属性：

* id：指定该Bean的唯一标识，Spring根据id属性值来管理Bean，程序通过id属性值来访问该Bean实例。
* class：指定该Bean的实现类，此处不可再用接口，必须使用实现类，Spring容器会使用XML解析器读取该属性值，并利用反射来创建该实现类的实例。

#### 构造注入

利用构造器来设置依赖关系的方式，被称为构造注入。

通俗来说，就是驱动Spring在底层以反射方式执行带参数的构造器，当执行带参数的构造器时，就可利用构造器参数对成员变量执行初始化——这就是构造注入的本质。

使用<constructor-arg.../>子元素，每一个<constructor-arg.../>代表一个构造器参数，如果<bean.../>元素包含N个<constructor-arg.../>子元素，就会驱动Spring调用带N个参数的构造器来创建对象。

配置<constructor-arg.../>元素时，可以指定一个index属性，用于指定该构造参数值将作为第几个构造参数值。

为更明确指定数据类型，Spring允许制定一个type属性。<value.../>元素也可指定type属性，用于确定该属性值的数据类型。

**注意：**

无论是String还是Int/short/long，在赋值时都是 value="值" ，

因此建议 此种情况 需要配合 name\type进行区分。

**在ioc中定义bean的前提：该bean的类 必须提供了 无参构造。**

#### P命名空间注入

在声明中添加如下：

xmlns:p=<http://www.springframework.org/schema/p>

即可使用P命名空间注入，注入方式如下：

在bean属性中添加： p:stuAge=*"2"*

简单类型：

p:属性名="属性值"

引用类型（除了String外）：

p:属性名-ref="引用的id"

**注意：**

多个 p赋值的时候 要有空格。

自动装配（只适用于 ref类型 ）

**约定优于配置**

自动装配：

<bean...class="org.Spring.entity.Course" autowire="byName|byType|constructor|no" >

byName: 自动寻找：其他bean的id值=该Course类的属性名，本质上就是byId。

byType: 其他bean的类型(class) 是否与 该Course类的ref属性类型一致（注意，此种方式 必须满足：当前Ioc容器中 只能有一个Bean满足条件）。

constructor：其他bean的类型(class) 是否与 该Course类的构造方法参数 的类型一致；此种方式的本质就是byType。

可以在头文件中一次性将该IOC容器的所有bean统一设置成自动装配：

<beans xmlns="http://www.springframework.org/schema/beans"...default-autowire="byName">

**注意：**

**自动装配虽然可以减少代码量，但是会降低程序的可读性，使用时需要谨慎。**

#### 特殊值的注入问题

##### value与<value>注入方式的区别：

|  |  |  |
| --- | --- | --- |
|  | 使用子元素<value>注入 | 而使用value属性注入 |
| 参数值位置 | 写在首尾标签（<value></value>）的中间(不加双引号) | 写在value的属性值中（必须加双引号） |
| type属性 | 有（可选）  可以通过type属性指定数据类型 | 无 |
| 参数值包含特殊字符（<， &）时的处理方法 | 两种处理方法。  **一、使用<![CDATA[ ]]>标记**  二、使用XML预定义的实体引用 | 一种处理方法。即使用XML预定义的实体引用 |

其中，XML预定义的实体引用，如表所示。

|  |  |
| --- | --- |
| 实体引用 | 表示的符号 |
| &lt; | < |
| &amp; | & |
| &gt; | > |

##### 处理空值问题：

当对对象类型赋值null时：

<property name=*"stuName"* value=*""*></property>

或：

<property name=*"stuName"*>

<null/>

</property>

#### 两种注入方式的对比

区别在于：

设值注入是先通过无参数的构造器创建一个Bean实例，然后调用对应的setter方法注入依赖关系。

构造注入则直接调用有参数的构造器，当Bean实例创建完成后，已经完成了依赖关系的注入。

设值注入具有如下优点：

* 与传统JavaBean写法更相似，程序开发人员更容易理解，接受，通过setter方法设定依赖关系更加直观、自然。
* 对于复杂的依赖关系，如果采用构造注入，会导致构造器过于臃肿，难以阅读。Spring在创建Bean实例时，需要同时实例化其依赖的全部实力，因而导致性能下降。
* 在某些成员变量可选的情况下，多参数构造器更加笨重。

构造注入具有如下优点：

* 构造注入可以在构造器中决定依赖关系的注入顺序，优先依赖的优先注入。采用构造注入，可以在代码中清晰地决定注入顺序。
* 对于依赖关系无需变化的Bean，构造注入更有用处。无需担心后续代码对依赖关系产生破坏。
* 依赖关系只能在构造器中设定，则只有组件的创建者才能改变组建的依赖关系。对组件的调用者而言，组件内部的依赖关系完全透明，更符合高内聚的原则。

建议采用以设值注入为主，构造注入为辅的注入策略。对于依赖关系无需改变的注入，尽量采用构造注入；而其他依赖关系的注入，则考虑设值注入。

#### **使用注解定义bean**

通过注解的形式将bean以及相应的属性值 放入ioc容器：

<context:component-scan base-package="org.Spring.dao"></context:component-scan>

Spring在启动的时候，会根据base-package在 该包中扫描所有类，查找这些类是否有注解@Component("studentDao"),如果有，则将该类 加入spring Ioc容器。

@Component细化：

Dao层注解：@Repository

service层注解：@Service

控制器层注解：@Controller

## 使用Spring容器

使用Spring框架之后的两个主要改变是：

* 程序无需使用new调用构造器去创建对象。所有的Java对象都可以交给Spring容器去创建。
* 当调用者需要调用被依赖对象的方法时，调用者无需主动获取被依赖对象，只要等待Spring容器注入即可。

Spring IoC容器三个基本要点：

* 应用程序的各组件面向接口编程。面向接口编程可以将组件之间的耦合关系提升到接口层次，从而有利于项目后期的扩展。
* 应用程序的各组件不再有程序主动创建，而是有Spring容器来负责产生并初始化。
* Spring采用配置文件或注解来管理Bean的实现类、依赖关系，Spring容器则根据配置文件或注解，利用反射来创建实例，并为之注入依赖关系。

Spring有两个核心接口：BeanFactory和ApplicationContext，其中ApplicationContext是BeanFactory的子接口。它们都可代表Spring容器，Spring容器是生成Bean实例的工厂，并管理容器中的Bean。在基于Spring的JavaEE应用中，所有组件都被当成Bean处理，包括数据源、Hibernate的SessionFactory、事务管理器等。

Java程序面向接口编程，无需关心Bean实例的实现类；但Spring容器负责创建Bean实例，故Spring配置文件必须指定Bean实例的实现类。

Spring容器最基本的接口就是BeanFactory。BeanFactory负责配置、创建、管理Bean，它有一个子接口：ApplicationContext，也被称为Spring的上下文。Spring容器还负责管理Bean与Bean之间的依赖关系。

### 使用BeanFactory

BeanFactory接口包含如下几个基本方法：

* boolean containsBean(String name)：判断Spring容器中是否包含id为name的Bean实例。
* <T> T getBean(Class<T> requiredType)：获取Spring容器中属于requiredType类型的、唯一的Bean实例。
* Object getBean(String name)：返回容器id为name的Bean实例。
* <T> T getBean(String name , Class requiredType)：返回容器中id为name，并且类型为requiredType的Bean。
* Class<?> getType(String name)：返回容器中id为name的Bean实例的类型。

调用者只需使用getBean()方法即可获得指定Bean的引用，无需关心Bean的实例化过程。Bean实例的创建，初始化以及依赖关系的注入都由Spring容器完成。

**BeanFactory常用的实现类是DefaultListableBeanFactory。**

创建Spring容器的实例时，必须提供Spring容器管理的Bean的详细配置信息。Spring的配置信息通常采用XML配置文件来设置。因此，创建BeanFactory实例时，应该提供XML配置文件作为参数。XML配置文件通常使用Resource对象传入。

Resource接口是Spring提供的资源访问接口，通过使用该接口，Spring能以简单、透明的方式访问磁盘、类路径以及网络上的资源。

大部分JavaEE应用，可在启动Web应用时，自动加载ApplicationContext实例，接受Spring管理的Bean无需知道ApplicationContext的存在，一样可以利用ApplicationContext的管理。

对于独立的应用程序，可通过如下方法来实例化BeanFactory。

//搜索类加载路径下的beans.xml文件创建Resource对象

Resource isr = **new** ClassPathResource("beans.xml");

//搜索文件系统的当前路径下的beans.xml文件创建Resource对象

Resource isr = **new** FileSystemResource("beans.xml");

//创建默认的BeanFactory容器

DefaultListableBeanFactory beanFactory = **new** DefaultListableBeanFactory();

//让默认的BeanFactory容器加载isr对应的XML配置文件

**new** XmlBeanDefinitionReader(beanFactory).loadBeanDefinitions(isr);

### 使用ApplicationContext

ApplicationContext是BeanFactory的子接口，因此功能更强大。对于大部分JavaEE应用来说，用它做为Spring容器更方便。其常用实现类是：FileSystemXmlApplicationContext、ClassPathXmlApplicationContext和AnnotationConfigWebApplicationContext。如果在Web应用中使用Spring容器，则通常有XmlWebApplicationContext、AnnotationConfigWebApplicationContext两个实现类。

如果应用需要加载多个配置文件来创建Spring容器，则应该采用BeanFactory子接口ApplicationContext来创建BeanFactory实例。

//以类加载路径下的beans.xml、service.xml文件创建ApplicationContext

ApplicationContext appContext = **new** ClassPathXmlApplicationContext("beans.xml","serviece.xml");

//以文件系统的相对路径或绝对路径下的beans.xml、service.xml文件创建ApplicationContext

ApplicationContext appContext = **new** FileSystemXmlApplicationContext("beans.xml","serviece.xml");

由于ApplicationContext本身就是BeanFactory的子接口，因此ApplicationContext完全可以作为Spring容器来使用，而且功能更强。如果有需要，也可以把ApplicationContext实例赋给BeanFactory变量。

大部分时候，都不会使用BeanFactory实例作为Spring容器，而是使用ApplicationContext实例作为Spring容器来使用，因此Spring容器也成为Spring上下文。ApplicationContext作为BeanFactory的子接口，增强了BeanFactory的功能。

ApplicationContext包括BeanFactory的全部功能，因此**建议优先使用ApplicationContext**。除非对于某些内存非常关键的应用，才考虑使用BeanFactory。

ApplicationContext允许以声明式方式操作容器，无需手动创建。可利用如ContextLoader的支持类，在Web应用启动时自动创建ApplicationContext。也可采用编程方式创建ApplicationContext。

ApplicationContext额外功能包括：

* ApplicationContext默认会预初始化所有的Singleton Bean，也可以通过配置取消预初始化。
* ApplicationContext继承MessageSource接口，因此提供国际化支持。
* 资源访问，如URL和文件。
* 事件机制。
* 同时加载多个配置文件。
* 以声明式方式启动并创建Spring容器。

当ApplicationContext容器初始化完成后，容器会自动初始化所有的Singleton Bean。包括调用构造器创建该Bean的实例，并根据<property.../>元素执行setter方法。这意味着系统在前期创建ApplicationContext时将有较大的系统开销，但一旦ApplicationContext初始化完成，程序后面获取singleton Bean实例时将拥有较好的性能。

为了阻止Spring容器预初始化容器中的singleton Bean，可以为<bean.../>元素指定lazy-init=“true”，那么即使使用ApplicationContext作为Spring容器，spring也不会预初始化该singleton Bean。

### ApplicationContext的国际化支持

ApplicationContext接口继承了MessageSource接口，因此具有国际化功能。下面是MessageSource接口中定义的两个用于国际化的方法：

* String getMessage(String code,Object[] args,Locate loc)
* String getMessage(String code,Object[] args,String default ,Locale loc)

ApplicationContext正式通过这两个方法来完成国际化的，当程序创建ApplicationContext容器时，Spring自动查找配置文件中名为messageSource的Bean实例，一旦找到这个Bean实例，上述两个方法的调用就被委托给messageSource Bean。如果没有该Bean，ApplicationContext会查找其父容器中的messageSource Bean；如果找到，它将作为messageSource Bean使用。

如果无法找到messageSource Bean，系统将会创建一个空的StaticMessageSource Bean，该Bean能接受上述两个方法的调用。

在Spring中配置messageSource Bean时通常使用ResourceBundleMessageSource类。

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

<!-- 驱动Spring调用MessageSource Bean的setBasenames()方法，

该方法需要一个数组参数，使用list元素配置多个数组元素 -->

<property name=*"basenames"*>

<list>

<value>message</value>

<!-- 如果有多个资源文件，全部列在此处 -->

</list>

</property>

</bean>

该Bean实例制定了一份国际化资源文件，其baseName是message。

然后给出如下两份资源文件：

第一份为美式英语的资源文件，文件名：message\_en\_US.properties。

hello=welcome,**{0}**

now=now is : **{0}**

第二份为简体中文的资源文件，文件名：message\_zh\_CN.properties。

hello=欢迎你,**{0}**

now=现在时间是：**{0}**

注：Java 9支持使用UTF-8字符集保存国际化资源文件，这种国际化资源文件可以包含非西欧字符。

主程序调用：

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml");

//使用getMessage()方法获取本地化消息

//Locale的getDefault()方法返回计算机环境默认的Locale。

String hello = ctx.getMessage("hello", **new** String[] {"孙悟空"},Locale.*getDefault*(Locale.Category.***FORMAT***));

String now = ctx.getMessage("now", **new** Object[] {**new** Date()},Locale.*getDefault*(Locale.Category.***FORMAT***));

System.***out***.println(hello);

System.***out***.println(now);

Spring国际化支持，其实是建立在Java程序国际化的基础之上的，其核心思路都是讲程序中需要实现国际化的信息写入资源文件，而代码中仅仅使用相应的各信息的Key。

### ApplicationContext的事件机制

ApplicationContext的事件机制是观察者设计模式的实现。

Spring的事件框架有如下两个重要成员：

* ApplicationEvent：容器事件，必须由ApplicationContext发布。
* ApplicationListener：监听器，可由容器中的任何监听器Bean担任。

Spring的事件机制与所有的事件机制都基本相似，它们都需要由事件源、事件和事件监听器组成，只是此处的事件源是ApplicationContext，且事件必须由Java程序显式触发。

ApplicationContext

激发监听器

发布ApplicationEvent事件

程序

ApplicationListener

onApplicationEvent()方法被触发，ApplicationEvent将作为该方法的参数

图：Spring容器的事件机制示意图

只要一个Java类继承了ApplicationEvent基类，那该对象就可以作为Spring容器的容器事件。

**import** org.springframework.context.ApplicationEvent;

**public** **class** EmailEvent **extends** ApplicationEvent{

**private** String address ;

**private** String text ;

**public** EmailEvent(Object source) {

**super**(source);

}

//初始化全部成员变量的构造器

**public** EmailEvent(Object source , String address , String text) {

**super**(source);

**this**.address = address ;

**this**.text = text ;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** String getText() {

**return** text;

}

**public** **void** setText(String text) {

**this**.text = text;

}

}

容器事件的监听器类必须实现ApplicationListener接口，实现该接口必须实现如下方法。

* onApplicationEvent(ApplicationEvent event)：每当容器内发生任何事件时，此方法都能被触发。

**import** org.du.app.event.EmailEvent;

**import** org.springframework.context.ApplicationEvent;

**import** org.springframework.context.ApplicationListener;

**public** **class** EmailNotifier **implements** ApplicationListener{

@Override

**public** **void** onApplicationEvent(ApplicationEvent evt) {

//只处理EmailEvent,模拟发送email通知

**if**(evt **instanceof** EmailEvent) {

EmailEvent emailEvent = (EmailEvent)evt;

System.***out***.println("需要发送邮件的接收地址" + emailEvent.getAddress());

System.***out***.println("需要发送邮件的正文" + emailEvent.getText());

}**else** {

//其他时间不做任何处理

System.***out***.println("其他时间：" + evt);

}

}

}

<!-- 配置监听器 -->

<bean class=*"org.du.app.listener.EmailNotifier"*></bean>

为Spring容器注册事件监听器，不需要想AWT变成那样采用代码进行编程，只要进行简单配置即可。只要在Spring中配置一个实现了ApplicationListener接口的Bean，Spring容器就会把这个Bean当成容器事件的事件监听器。

当系统创建Spring容器，加载Spring容器时会自动触发容器事件，容器事件监听器可以监听到这些事件。

除此之外，程序可以调用ApplicationContext的pulishEvent()方法来主动触发容器事件。

**import** org.du.app.event.EmailEvent;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** SpringTest {

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

ApplicationContext context = **new** ClassPathXmlApplicationContext("beans.xml");

//创建一个ApplicationEvent对象

EmailEvent ele = **new** EmailEvent("test" ,

"Spring\_test@163.com" , "this is a test");

//发布容器事件

context.publishEvent(ele);

}

}

**注意：**

设立监听器需要引入spring-aop-5.0.2.RELEASE.jar包，进行切面编程。

**执行结果：**

其他事件：org.springframework.context.event.ContextRefreshedEvent[source=org.springframework.context.support.ClassPathXmlApplicationContext@5fa7e7ff: startup date [Thu Jan 24 16:47:42 CST 2019]; root of context hierarchy]

需要发送邮件的接收地址 Spring\_test@163.com

需要发送邮件的正文 this is a test

**监听器不仅监听到程序所触发的事件，也监听到容器内置的时间。**

如果Bean希望发布容器事件，则该Bean必须先获得对Spring容器的引用。为了让Bean获得对Spring容器的引用，可让Bean类实现ApplicationContextAware或BeanFactoryAware接口。

**Spring提供如下几个内置事件：**

* ContextRefreshedEvent：ApplicationContext容器初始化或刷新触发该事件。此处的初始化是指，所有的Bean被成功加载，后处理的Bean被检测并激活，所有的Singleton Bean被预实例化，ApplicationContext容器已就绪可用。
* ContextStartedEvent：当使用COnfigurableApplicationContext（ApplicationContext的子接口）接口的start()方法启动ApplicationContext容器时触发该事件。容器管理生命周期的Bean实例将获得一个指定的启动信号，这在经常需要停止后重新启动的场合比较常见。
* ContextCloseEvent：当使用ConfigurableApplication（ApplicationContext的子接口）接口的close()方法关闭ApplicationContext容器时触发该事件。
* ContextStoppedEvent：当使用ConfigurableApplicationContext（ApplicationContext的子接口）接口的stop()方法使ApplicationContext停止时触发该事件。此处的“停止”意味着容器管理声明周期的Bean实例将获得一个指定的停止信号，被停止的Spring容器可再次调用start()方法重新启动。
* RequestHandledEvent：Web相关的事件，只能应用于使用DispatcherServlet的Web应用中。在使用Spring作为前端的MVC控制器时，当Spring处理用户请求结束后，系统会自动触发该事件。
* Spring 4之后还新增了SessionConnectedEvent、SessionConnectEvent、SessionDisconnectEvent这三个事件，它们都用于为Spring新增的WebSocket功能服务。

### 让Bean获取Spring容器

在某些特殊的情况下，Bean需要实现某个功能（比如该Bean需要输出国际化消息，或者该Bean需要向Spring容器发布事件·····），但该功能必须借助于Spring容器才能实现，此时就必须让该Bean获取它所在的Spring容器，然后借助于Spring容器来实现该功能。

为了让Bean获取它所在的Spring容器，可以让该Bean实现BeanFactoryAware接口，BeanFactoryAware接口里只有一个方法。

* setBeanFactory(BeanFactory beanFactory)：该方法有一个参数beanFactory，该参数指向创建它的BeanFactory。

该方法将由Spring调用，Spring调用该方法时会将Spring容器作为参数传入该方法。与该接口类似的还有BeanNameAware、ResourceLoaderAware接口，这些接口里面都会提供类似的setter方法，这些方法也由Spring负责调用。

与BeanFactoryAware接口类似的有ApplicationContextAware接口，实现该接口的Bean需要实现setApplicationContext(ApplicationContext applicationContext)方法——该方法也是由Spring来调用的。当Spring容器调用该方法时，它会把自身作为参数传入该方法。

**import** java.util.Locale;

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

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.ApplicationContextAware;

**public** **class** Person **implements** ApplicationContextAware{

//用成员变量保存它所在的ApplicationContext容器

**private** ApplicationContext ctx;

/\*

\* Spring容器会检测容器中所有的Bean，如果发现某个Bean实现了ApplicationContextAware接口，

\* Spring容器会在创建该Bean之后，自动调用该方法，调用该方法时，会将容器本身作为参数传给该方法。

\*/

@Override

**public** **void** setApplicationContext(ApplicationContext arg0) **throws** BeansException {

// **TODO** Auto-generated method stub

**this**.ctx = ctx ;

}

**public** **void** sayHi(String name) {

System.***out***.println(ctx.getMessage("hello", **new** String[]{name}, Locale.*getDefault*(Locale.Category.***FORMAT***)));

}

}

上面的Person类实现了ApplicationContextAware接口，并实现了该接口提供的setApplicationContextAware()方法。

Spring容器会检测容器中所有的Bean，如果发现某个Bean实现了ApplicationContextAware接口，Spring容器会在创建该Bean之后，自动调用该方法，调用该方法时，会将容器本身作为参数传给该方法。

将该Bean部署在Spring容器中：

<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"*>

<!-- 加载容器国际化所需要的语言资源文件 -->

<bean id=*"messageSource"*

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

<property name=*"basenames"*>

<list>

<value>message</value>

</list>

</property>

</bean>

<!--

Spring容器会检测容器中所有的Bean，如果发现某个Bean实现了

ApplicationContextAware接口，Spring容器会在创建该Bean之后，

自动调用该Bean的setApplicationContext()方法，调用该方法时，

会将容器本身作为参数传给该方法。

-->

<bean id=*"person"* class=*"org.du.app.service.Person"*></bean>

</beans>

**import** org.du.app.service.Person;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** SpringTest {

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

ApplicationContext context = **new** ClassPathXmlApplicationContext("beans.xml");

Person p = context.getBean("person",Person.**class**);

p.sayHi("孙悟空");

}

}

## Spring容器中的Bean

对于开发者来说，开发者使用Spring框架主要是做两件事：1.开发Bean；2.配置Bean。

对于Spring框架来说，它要做的就是根据配置文件来创建Bean实例，并调用Bean实例的方法完成“依赖注入”——这就是所谓IOC的本质。

其实Spring框架的本质就是，通过XML配置来驱动Java代码，这样就可以把原本由Java代码管理的耦合关系，提取到XML配置文件中管理，这就实现了系统中各组件的解耦，有利于后期的升级和维护。

### Bean的基本定义和Bean的别名

<beans…/>元素是Spring配置文件的根元素，该元素可指定如下属性：

* default-lazy-init：指定该<beans…/>元素下配置的所有Bean默认的延迟初始化行为。
* default-merge：指定该<beans…/>元素下配置的所有Bean默认的merge行为。
* default-autowire：指定该<beans…/>元素下配置的所有Bean默认的自动装配行为。
* default-autowire-candidates：指定该<beans…/>元素下配置的所有Bean默认是否作为自动装配的候选Bean。
* default-init-method：指定该<beans…/>元素下配置的所有Bean默认的初始化方法。
* default-destroy-method：指定该<beans…/>元素下配置的所有Bean默认的回收方法。

<beans…/>元素下所能制定的属性都可以在每个<bean…/>子元素中指定——将属性名去掉default即可。

定义Bean时，通常需要指定两个属性：

* id：确定该Bean的唯一标识，容器对Bean的管理、访问，以及该Bean的依赖关系，都通过该属性完成。Bean的id属性在Spring容器中应该是唯一的。
* class：指定该Bean的具体实现类，这里不能是接口。Spring容器必须知道创建Bean的实现类，而不能是接口。在通常情况下，Spring会直接使用new关键字创建该Bean的实例，因此这里必须提供Bean实现类的类名。

**注：**

在一些特殊情况下，Spring会采用其他方式创建Bean实例，例如工厂方法等，则可能不再需要class属性。

除了可以为<bean…/>元素指定一个id属性之外，还可以为<bean…/>元素指定name属性，用于为Bean实例指定别名。

指定别名有两种方式：

* 定义<bean…/>元素时通过name属性指定别名：如果需要为Bean实例指定多个别名，则可以在name属性中使用逗号、冒号或空格来分割多个别名，后面通过任一别名即可访问该Bean实例。
* 通过<alias…/>元素为已有的Bean指定别名。

在一些极端情况下，程序无法在定义Bean时就指定所有的别名，而是需要在其他地方为一个已经存在的Bean实例指定别名，则可使用<alias…/>元素来完成，该元素可指定如下两个属性。

* name：该属性指定一个Bean实例的标识名，表明将为该Bean实例指定别名。
* alias：指定一个别名。

例如：

<bean id=*"person"* class=*"..."* name=*"#abd,@123,abc\*"*/>

<alias name=*"person"* alias=*"jack"*/>

<alias name=*"jack"* alias=*"jackee"*/>

### 容器中Bean的作用域

当通过Spring容器创建一个Bean实例时，不仅可以完成Bean实例的实例化，还可以为Bean指定特定的作用域。

Spring支持如下6种作用域：

* singleton：单例模式，在整个SpringIOC容器中，singleton作用域的Bean将只生成一个实例。
* prototype：每次通过容器的getBean()方法获取prototype作用域的Bean时，都将产生一个新的Bean实例。
* request：对于一次HTTP请求，request作用域的Bean将只生成一个实例，这意味着在同一次HTTP会话内，程序每次请求该Bean，得到的总是同一个实例。只有在Web应用中使用Spring时，该作用域才真正有效。
* session：对于一次HTTP会话，session作用域的Bean将只生成一个实例，这意味着，在同一次HTTP会话内，程序每次请求该Bean，得到的总是同一个实例，只有在Web应用中使用Spring时，该作用域才真正有效。
* application：对应整个Web应用，该Bean只生成一个实例。这意味着，在整个Web应用内，程序每次请求该Bean时，得到的总是同一个实例。只有在Web应用中使用Spring时，该作用域才真正有效。
* websocket：在整个WebSocket的通信过程中，该Bean只生成一个实例。只有在Web应用中使用Spring时，该作用域才真正有效。

比较常用singleton和prototype两种作用域，对于singleton作用域的Bean，每次请求该Bean都将获得相同的实例。容器负责跟踪Bean实例的状态，负责维护Bean实例的生命周期行为。

注：

如果不指定Bean的作用域，Spring默认使用singleton作用域。

Spring配置文件通过scope属性指定Bean的作用域，该属性可以接受singleton、prototype、request、session和globalSession五个值。

**例：**

<bean id=*"person"* class=*"org.du.app.service.Person"* scope=*"prototype"*/>

session作用域与request作用域类似，区别在于：

request作用域的Bean对于每次HTTP请求有效，而session作用域的Bean则对于每次HTTP Session有效。

request和session作用域只有在web应用中才有效，并且必须在Web应用中增加额外配置才会生效。为了让request和session两个作用域生效，必须将HTTP请求对象绑定到为该请求提供服务的线程上，这使得具有request和session两个作用域的Bean实例能够在后面的调用链中被访问到。

在Web应用的web.xml文件中增加如下Listener配置，该Listener负责使request作用域生效。  
<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

一旦在web.xml中增加了如上配置，程序就可以在Spring配置文件中使用request或session作用域了。

**注：**

1.如果web应用直接使用Spring MVC作为MVC框架，即用SpringDispatcherServlet或DispatcherPorlet来拦截所有用户请求，则无需这些额外的配置。

2.Spring5不仅可以为Bean指定已经存在的6个作用域，还支持自定义作用域。

### 配置依赖

不管是设值注入还是构造注入，都是为Bean的依赖，接受Spring容器管理，依赖关系的值要么使一个确定的值，要么是Spring容器其他Bean的引用。

通常不建议使用配置文件管理Bean的基本类型的属性值；

通常只是用配置文件管理容器中Bean与Bean之间的依赖关系。

对于singleton作用域的Bean，如果没有强行取消其预初始化行为，系统会在创建Spring容器时预初始化所有的singleton Bean，与此同时，该Bean所依赖的Bean也被一起实例化。

BeanFactory与ApplicationContext实例化容器中Bean的时机不同：前者等到程序需要Bean实例时才创建Bean；后者在容器创建ApplicationContext时，会预初始化容器中所有的singleton Bean。

**注：**

因为采用ApplicationContext作为Spring容器，创建容器时会同时创建容器中所有singleton作用域的Bean，因此可能需要更多的系统开销。但一旦创建成功，应用后面响应的速度更快，因此，对于普通的JavaEE应用，推荐使用ApplicationContext作为Spring容器。

创建BeanFactory时不会立即创建Bean实例，所以有可能程序可以正常地创建BeanFactory实例，但当请求Bean实例时依然抛出异常，创建Bean实例或注入它的依赖关系时出现错误。配置错误的延迟出现，也会给系统引入不安全因素。

### 设置普通属性值

<value…/>元素用于指定基本类型及其包装类、字符串类型的参数值，Spring使用XML解析器来解析出这些数据，然后利用java.beans.PropertyEditor完成类型转换：从Java.lang.String类型转换为所需的参数值类型。

### 配置合作者Bean

如果需要为Bean设置的属性值时容器中的另一个Bean实例，则应该使用<ref…/>元素。使用<ref…/>元素时可以指定一个bean属性，该属性用于引用容器中其它Bean实例的id属性值。

通过为<property…/>元素增加ref属性一样可以将容器中另一个Bean作为调用setter方法的参数。

<constructor-arg…/>元素也可以增加ref属性，从而指定将容器中另一个Bean作为构造器参数。

<bean id=*"person"* class=*"org.du.app.service.Person"*>

<property name=*"stuName"* ref=*"teacher"*>

<ref bean=*"teacher"*/>

</property>

</bean>

### 使用自动装配注入合作者Bean

**约定优于配置。**

Spring能自动装配Bean与Bean之间的依赖关系，即无需使用ref显示指定依赖Bean，而是由Spring容器检查XML配置文件内容，通过某种规则，为调用者Bean注入被依赖的Bean。

Spring的自动装配可通过<beans…/>元素的default-autowire属性指定，该属性对配置文件中所有的Bean起作用；也可通过<bean…/>元素的autowire属性指定，该属性只对该Bean起作用。

自动装配可以减少配置文件的工作量，但降低了依赖关系的透明性和清晰性。

autowire、default-autowire属性可以接受如下值：

* no：不使用自动装配。这是默认的配置，在较大的部署环境中不鼓励改变这个配置。
* byName：根据setter方法名进行自动装配。
* byType：根据setter方法的形参类型来自动装配。
* constructor：与byType类似，区别是用于自动装配构造器的参数。
* autodetect：Spring容器根据Bean内部结构，自行决定使用constructor或byType策略。

在某些情况下，程序希望将某些Bean排除在自动装配之外，不作为Spring自动装配策略的候选者，此时可设置autowire-candidate属性，通过为<bean…/>元素设置autowire-candidate=”false”，即可将该Bean排除在自动装配之外。除此之外，还可以在<beans…/>元素中指定default-autowire-candidates属性将一批Bean排除在自动装配之外。default-autowire-candidates属性的值允许使用模式字符串，例如指定default-autowire-candidates=”\*abc”，则所有以abc结尾的Bean都将被排除在自动装配之外。

### 注入嵌套Bean

如果某个Bean所依赖的Bean不想被Spring容器直接访问到，则可以使用嵌套Bean。

把<bean…/>配置成<property…/>或<constructor-arg…/>的子元素，那么该<bean…/>元素配置的Bean仅仅作为setter注入、构造注入的参数，这种Bean就是嵌套Bean。由于容器不能获取嵌套Bean，因此它不需要指定id属性。

<bean id=*"person"* class=*"org.du.app.service.Person"*>

<property name=*"bean"*>

<bean class=*"org.du.app.service.Teacher"*/>

</property>

</bean>

嵌套Bean提高了程序的内聚性，但降低了程序的灵活性，只有在完全确定无需通过Spring容器访问某个Bean实例时，才考虑使用嵌套Bean来配置该Bean。

### 注入集合值

如果需要调用形参类型为集合的setter方法，或调用形参类型为集合的构造器，则可使用集合元素<list…/>、<set…/>、<map…/>和<props…/>分别来设置类型为List、Set、Map和Properties的集合参数值。

<list…/>、<key…/>、<set…/>元素可接受如下子元素：

<list>

<value>基本数据类型或字符串类型值</value>

<bean>指定一个嵌套Bean实例</bean>

<ref bean=*"指定集合元素时容器中的另一个Bean实例"*/>

<list>指定集合元素又是集合</list>

<set>指定集合元素又是集合</set>

<map>指定集合元素又是集合</map>

<props>指定集合元素又是集合</props>

</list>

<props…/>元素用于配置Properties类型的参数值，Properties类型是一种特殊的类型，其key和value都只能是字符串。

<prop key=*"hello"*>你好</prop>

<map…/>元素配置Map参数值时比较复杂。

<map>

<entry>

<key>

<value>hello</value>

</key>

<value>你好</value>

</entry>

</map>

从Spring 2.0开始，Spring IOC容器将支持集合的合并，子Bean中的集合属性可以从其父Bean的集合属性继承和覆盖而来。也就是说，自Bean的集合属性的最终值时父Bean、子Bean合并后的最终结果，而且子Bean集合中的元素也可以覆盖父Bean集合中对应的元素。

<!-- 将父Bean定义成抽象Bean -->

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

<!-- 定义Properties类型的集合属性 -->

<property name=*"adminEmails"*>

<props>

<prop key=*"administrator"*>administrator@test.com</prop>

<prop key=*"support"*>support@test.com</prop>

</props>

</property>

</bean>

<!-- 使用parent属性指定该Bean继承了parent Bean -->

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

<property name=*"adminEmails"*>

<!-- 指定该集合属性支持合并 -->

<props merge=*"true"*>

<prop key=*"sales"*>sales@test.com</prop>

<prop key=*"support"*>master@test.com</prop>

</props>

</property>

</bean>

上面配置片段中child Bean继承了parent Bean，并为<props…/>元素指定了merge=”true”，这将会把parent Bean的集合属性合并到child Bean中；当进行合并时，由child Bean再次配置了名为support的属性，所以该属性会覆盖parent Bean中的配置定义。

从JDK 1.5以后，Java可以使用泛型指定集合元素的类型，则Spring可通过反射来获取集合元素类型，这样Spring的类型转换器也会起作用了。

### 组合属性

Spring还支持组合属性的方式。例如使用配置文件形如foo.bar.name的属性设置参数值。

为Bean的组合属性设置参数值时，除最后一个属性之外，其他属性值都不允许为null,否则将引发NullPointerException异常。。

<property name=*"person.name"* value=*"孙悟空"* />

**例如：**

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

<property name=*"foo.bar.x.y"* value=*"xxx"*/>

</bean>

上面的组合属性注入相当于让Spring执行如下代码：

a.getFoo().getBar().getX().setY("xxx");

### Spring的Bean和JavaBean

Spring容器对Bean没有特殊要求，甚至不要求该Bean像标准的JavaBean——必须为每个属性提供对应的getter和setter方法。Spring中的Bean时Java实例、Java组件；而传统的Java应用中JavaBean通常作为DTO（数据传输对象），用来封装值对象，在各层之间传递数据。

虽然Spring对Bean没有特殊要求，但依然建议Spring中的Bean应满足如下几个原则：

* 尽量为每个Bean实现类提供无参数的构造器。
* 接受构造注入的Bean，则应提供对应的、带参数的构造函数。
* 接受设值注入的Bean，则应提供对应的setter方法，并不要求提供对应的getter方法。

传统的JavaBean和Spring中的Bean存在如下区别：

* 用处不同：传统的JavaBean更多是作为之对象传递参数；Spring的Bean用处几乎无所不包，任何应用组件都被称为Bean。
* 写法不同：传统的JavaBean作为之对象，要求每个属性都提供getter和setter方法；但Spring的Bean只需为接受设值注入的属性提供setter方法即可。
* 生命周期不同：传统的JavaBean作为值对象传递，不接受任何容器管理其生命周期；Spring中的Bean由Spring管理其生命周期行为。

## Spring提供的Java配置管理

Spring为不喜欢XML的人提供了一种选择：如果不喜欢使用XML来管理Bean，以及Bean之间的依赖关系，Spring允许开发者使用Java类进行配置管理。

<?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"*>

<!-- 配置chinese实例，其实现类是Chinese -->

<bean id=*"Chinese"* class=*"org.app.service.impl.Chinese"*>

<!-- 驱动Spring执行setAxe()方法，以容器中id为stoneAxe的Bean为参数 -->

<property name=*"axe"* ref=*"stoneAxe"*/>

<!-- 驱动Spring执行setName()方法，以字符串"孙悟空"为参数 -->

<property name=*"name"* value=*"孙悟空"*/>

</bean>

<!-- 配置stoneAxe实例，其实现类是StoneAxe -->

<bean id=*"stoneAxe"* class=*"org.app.service.impl.StoneAxe"*/>

</beans>

上面的XML配置文件可以替换如下的Java配置类：

@Configuration

**public** **class** AppConfig {

//相当于定义一个名为personName的变量，其值为"孙悟空"

@Value("孙悟空") String personName;

//配置一个Bean：Chinese

@Bean(name="chinese")

**public** Person Person()

{

Chinese p = **new** Chinese();

p.setAxe(stoneAxe());

p.setName(personName);

**return** p;

}

//配置Bean：stoneAxe

@Bean(name="stoneAxe")

**public** Axe stoneAxe()

{

**return** **new** StoneAxe();

}

}

上面的配置文件中使用了Java配置类的三个常用注解：

* @Configuraton：用于修饰一个Java配置类。
* @Bean：用于修饰一个方法，将方法的返回值定义成容器中的一个Bean。
* @Value：用于修饰一个Field，用于为该Field配置一个值，相当于配置一个变量。

一旦使用了Java配置类来管理Spring容器中的Bean及其依赖关系，此时就需要使用如下方式来创建Spring容器：

//创建Spring容器

ApplicationContext context = **new**

AnnotationConfigApplicationContext(AppConfig.**class**);

使用Java配置类时，还有如下常用的注解：

* @Import：修饰一个Java配置类，用于向当前Java配置类中导入其他Java配置类。
* @Scope：用于修饰一个方法，指定该方法对应的Bean的生命域。
* @Lazy：用于修饰一个方法，指定该方法对应的Bean是否需要延迟初始化。
* @DependsOn：用于修饰一个方法，指定在初始化该方法对应的Bean之前初始化制定的Bean。

Spring提供@Configuration和@Bean并不是为了完全取代XML配置，只希望作为一种补充。在目前多数项目中，要么完全使用XML配置方式管理Bean的配置，要么使用以注解为主、XML为辅的配置方式管理Bean的配置，想要完全放弃XML配置还是很难的。

如果以XML为主，Java配置为辅，则在XML文件中配置：

<!-- 加载Java配置类 -->

<bean class=*"org.app.config.AppConfig"*/>

如果以Java配置为主，XMl配置为辅，则在Java中配置：

@Configuration

//导入XML配置

@ImportResource("classpath:/beans.xml")

**public** **class** AppConfig {

......

}

## 创建Bean的3中方式

Spring支持使用如下方式来创建Bean：

* 调用构造器创建Bean。
* 调用静态工厂方法创建Bean。
* 调用实例工厂方法创建Bean。

使用实例工厂方法创建Bean实例，以及使用子Bean方法创建Bean实例时，都可以不指定class属性。

### 使用构造器创建Bean实例

如果不采用构造注入，Spring底层会调用Bean类的无参数构造器来创建实例，因此要求该Bean类提供无参数的构造器。在这种情况下，class元素时必须的（除非采用继承），class属性的值就是Bean实例的实现类。Spring容器将使用默认的构造器来创建Bean实例，Spring对Bean实例的左右属性执行默认初始化，即所有基本类型的值初始化为0或false；所有引用类型的值初始化为null。接下来，BeanFactory会根据配置文件决定依赖关系，先实例化被依赖的Bean 实例，然后为Bean注入依赖关系，最后将一个完整的Bean实例返回给程序。

如果采用构造注入，则要求配置文件为<bean…/>元素添加<constructor-arg…/>子元素，每个<constructor-arg…/>子元素配置一个构造器参数。Spring容器将使用带对应参数的构造器来创建Bean实例，Spring调用构造器传入的参数即可用于初始化Bean的实例变量，最后也将一个完整的Bean实例返回给程序。

### 使用静态工厂方法创建Bean

使用静态工厂方法创建Bean实例时，class属性也必须指定，但此时class属性并不是指定Bean实例的实现类，而是静态工厂类，Spring通过该属性知道由哪个工厂类来创建Bean实例。使用factory-method属性来指定静态工厂方法，Spring将调用静态工厂方法（可能包含一组参数）返回一个Bean实例，一旦获得了指定Bean实例，Spring后面的处理步骤与采用普通方法创建的Bean实例则完全一样。

采用静态工厂方法创建Bean实例时，<bean…/>元素需要指定如下两个属性：

* class：该属性的值为静态工厂类的类名。
* factory-method：该属性指定静态工厂方法来生产Bean实例。

如果静态工厂方法需要参数，则使用<constructor-arg…/>元素传入。

**public** **interface** Being {

**public** **void** testBeing();

}

**public** **class** Dog **implements** Being{

**private** String msg;

//msg的setter方法

**public** **void** setMsg(String msg) {

**this**.msg = msg;

}

//实现接口必须实现的testBeing()方法

@Override

**public** **void** testBeing() {

System.***out***.println(msg + ",狗爱啃骨头！");

}

}

**public** **class** Cat **implements** Being{

**private** String msg;

//msg的setter方法

**public** **void** setMsg(String msg) {

**this**.msg = msg;

}

//实现接口必须实现的testBeing()方法

@Override

**public** **void** testBeing() {

System.***out***.println(msg + ",猫喜欢吃鱼！");

}

}

**public** **class** BeingFactory {

//返回Being实例的静态工厂方法

//arg参数决定返回那个Being类的实例

**public** **static** Being getBeing(String arg) {

//调用此静态方法的参数为dog，则返回Dog实例

**if**(arg.equalsIgnoreCase("dog")) {

**return** **new** Dog();

}**else** {

//否则返回Cat实例

**return** **new** Cat();

}

}

}

<?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"*>

<!--

下面配置驱动Spring调用BeingFactory的静态getBeing()方法来创建Bean，

该bean元素包含的constructor-arg元素用于为静态工厂方法指定参数，

因此这段配置会驱动Spring以反射方式来执行如下代码：

dog = org.app.factory.BeingFactory.getBeing("dog");

-->

<bean id=*"dog"* class=*"org.app.factory.BeingFactory"*

factory-method=*"getBeing"*>

<!-- 配置静态方法的参数 -->

<constructor-arg value=*"dog"*/>

<!-- 驱动Spring以"我是狗"为参数来执行Dog的setMsg()方法 -->

<property name=*"msg"* value=*"我是狗"*/>

</bean>

<!--

下面这段配置会驱动Spring以反射方式来执行如下代码：

dog = org.app.factory.BeingFactory.getBeing("cat");

-->

<bean id=*"cat"* class=*"org.app.factory.BeingFactory"*

factory-method=*"getBeing"*>

<!-- 配置静态方法的参数 -->

<constructor-arg value=*"cat"*/>

<!-- 驱动Spring以"我是猫"为参数来执行Cat的setMsg()方法 -->

<property name=*"msg"* value=*"我是猫"*/>

</bean>

</beans>

**public** **class** SpringTest {

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

// 以类加载路径下的beans.xml配置文件创建Spring容器

ApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

Being b1 = context.getBean("dog" , Being.**class**);

b1.testBeing();

Being b2 = context.getBean("cat" , Being.**class**);

b2.testBeing();

}

}

使用静态工厂方法创建实例时必须提供工厂类，工厂类包含产生实例的静态工厂方法。通过静态工厂方法创建实例时需要对配置文件进行如下改变。

* class属性的值不再是Bean实例的实现类，而是生成Bean实例的静态工厂类。
* 使用factory-method属性指定创建Bean实例的静态工厂方法。
* 如果静态工厂方法需要参数，则使用<constructor-arg…/>元素指定静态工厂方法的参数。

指定Spring使用静态工厂方法来创建Bean实例时，Spring将先解析配置文件，并根据配置文件指定的信息，通过反射调用静态工厂类的静态工厂方法，将该静态工厂方法的返回值作为Bean实例。在这个过程中，Spring不再负责创建Bean实例，Bean实例是由用户提供的静态工厂类负责创建的。

当静态工厂方法创建了Bean实例后，Spring依然可以管理该Bena实例的依赖关系，包括为其注入所需的依赖Bean、管理其生命周期等。

### 调用实例工厂方法创建Bean

实例工厂方法与静态工厂方法只有一点不同：调用静态工厂方法只需使用工厂类即可，而调用实例工厂方法则需要工厂实例。所以配置实例工厂方法与配置静态方法基本相似，只有一点区别：配置静态工厂方法使用class指定静态工厂类，而配置实例工厂方法使用factory-bean指定工厂实例。

采用实例工厂方法创建Bean的<bean…/>元素时需要制定如下两个属性：

* factory-bean：该属性的值为工厂Bean的id。
* factory-method：该属性指定实例工厂的工厂方法。

与静态工厂方法相似，如果需要在调用实例工厂时传入参数，则使用<constructor-arg…/>元素指定参数值。

**public** **interface** Person {

//定义一个打招呼的方法

**public** String sayHello(String name);

//定义一个告别的方法

**public** String sayGoodBye(String name);

}

**public** **class** American **implements** Person{

@Override

**public** String sayHello(String name) {

**return** name + ",Hello!";

}

@Override

**public** String sayGoodBye(String name) {

**return** name + ",Good Bye!";

}

}

**public** **class** Chinese **implements** Person{

@Override

**public** String sayHello(String name) {

**return** name + "，您好！";

}

@Override

**public** String sayGoodBye(String name) {

**return** name + "，下次再见！";

}

}

**public** **class** PersonFactory {

**public** Person getPerson(String ethnic) {

**if**(ethnic.equalsIgnoreCase("chinese")) {

**return** **new** Chinese();

}**else** {

**return** **new** American();

}

}

}

<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"*>

<!-- 配置工厂Bean，该Bean负责产生其他Bean实例 -->

<bean id=*"personFactory"* class=*"org.app.factory.PersonFactory"*/>

<!-- 下面配置驱动Spring调用personFactory Bean的getPerson()方法来创建Bean -->

<bean id=*"chinese"* factory-bean=*"personFactory"*

factory-method=*"getPerson"*>

<!-- 配置实例工厂方法的参数 -->

<constructor-arg value=*"chinese"*/>

</bean>

<bean id=*"american"* factory-bean=*"personFactory"*

factory-method=*"getPerson"*>

<constructor-arg value=*"american"*/>

</bean>

</beans>

**public** **class** SpringTest {

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

ApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

Person p1 = context.getBean("chinese",Person.**class**);

System.***out***.println(p1.sayHello("孙悟空"));

System.***out***.println(p1.sayGoodBye("猪八戒"));

Person p2 = context.getBean("american" , Person.**class**);

System.***out***.println(p2.sayHello("孙悟空"));

System.***out***.println(p2.sayGoodBye("猪八戒"));

}

}

**调用实例工厂方法与调用静态工厂方法创建Bean的区别如下：**

* 配置实例工厂方法创建Bean，必须将实例工厂配置成Bean实例；而配置静态工厂方法创建Bean，则无需配置工厂Bean。
* 配置实例工厂方法创建Bean，必须使用factory-bean属性确定工厂Bean；而配置静态工厂方法创建Bean，则使用class元素确定静态工厂类。

**调用实例工厂方法与调用静态工厂方法创建Bean的相同之处如下：**

* 都需要使用factory-method属性指定产生Bean实例的工厂方法。
* 工厂方法如果需要参数，都使用<constructor-arg…/>元素指定参数值。
* 普通的设值注入，都是用<property…/>元素确定参数值。

## 高级配置依赖关系

Spring允许将Bean实例的所有成员变量，甚至基本类型的成员变量都通过配置文件来指定值，这种方式提供了很好地解耦，但如果将基本类型的成员变量值也通过配置文件指定，虽然可以很好地解耦，但大大降低了程序的可读性，因此，滥用依赖注入也会引起一些问题。

通常建议是：组件与组件之间的耦合，采用依赖注入管理；但基本类型的成员变量值，应该直接在代码中设置。对于组件之间的耦合关系，通过使用控制反转，代码变得非常清晰。

因此Bean无需管理依赖关系，而是由容器提供注入，Bean无需知道这些实例在哪里，以及它们具体的实现。

Spring框架的本质是，开发者在Spring配置文件中使用XML元素进行配置，实际驱动Spring执行相应的代码，例如：

* 使用<bean.../>元素，实际启动Spring执行无参数或有参数的构造器，或者调用工厂方法创建Bean。
* 使用<property.../>元素，实际驱动Spring执行一次setter方法。
* 调用getter方法：使用propertyPathFactoryBean。
* 访问类或对象的Field值：使用FieldRetrievingFactoryBean。
* 调用普通方法：使用MethodInvokingFactoryBean。

换一个角度看Spring框架：Spring框架的功能是什么？它可以让开发者无需书写Java 代码就可以进行Java编程，当开发者XML采用合适语法进行配置之后，Spring就可以通过反射在底层执行任意的Java代码。

## 基于XML Schema的简化配置方法

从Spring2.0开始，Spring允许使用基于XML Schema的配置方式来简化Spring的配置文件。

### 使用p:命名空间简化配置

p:命名空间主要用于简化设值注入，它甚至不需要特定的Schema定义，它直接存在于Spring内核中。当导入p:命名空间后，就可以直接在<bean.../>元素中使用属性来驱动执行setter方法。

配置方式：

1、导入XML Schema里的p:命名空间：

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

1. 直接使用属性配置执行设值注入，例如：

<bean id=*"chinese"* class=*"org.crazyit.app.service.impl.Chinese"*

p:age=*"29"* p:axe-ref=*"stoneAxe"*/>

因为axe设值注入的参数需要引用容器中另一个已存在的Bean实例，故在axe后增加了“-ref”后缀，这个后缀指定该值不是一个具体的值，而是对另一个Bean的引用。

**注：**

使用p:命名空间没有标准的XML格式灵活，如果某个Bean的属性名是以“-ref”结尾的，那么采用p:命名空间定义时就会发生冲突，而采用标准的XML格式定义则不会出现这种问题。

### 使用c:命名空间简化配置

c:命名空间主要用于简化构造注入，导入c:命名空间之后，可以直接使用属性来配置构造器参数。

使用c:指定构造器参数的格式为：c:构造器参数名=“值”或c：构造器参数名-ref=“其它Bean的id”。

配置方式：

1. 导入XML Schema里的c:命名空间：

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

1. 直接使用属性配置执行构造注入：

<bean id=*"chinese"* class=*"org.crazyit.app.service.impl.Chinese"*

c:age=*"29"* c:axe-ref=*"stoneAxe"*/>

Spring还支持一种通过索引来配置构造器参数的方式。上面的bean也可以改写为如下形式：

<bean id=*"chinese"* class=*"org.crazyit.app.service.impl.Chinese"*

c:\_1=*"29"* c:\_0-ref=*"stoneAxe"*/>

在这种方式下，c:\_N中的N代表第几个构造参数。

如果希望根据构造参数的名称来配置构造注入，则可使用java.beans包的@ConstructorProperties注解。

### 使用util：命名空间简化配置

在Spring框架解压缩包的schema\util\路径下包含有util：命名空间的XML Schema文件，为了使用util命令空间的元素，必须先在Spring配置文件中导入最新的spring-util.xsd，也就是需要在Spring配置文件中增加如下配置片段：

xmlns:util=*"http://www.springframework.org/schema/util"*

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

http://www.springframework.org/schema/uitl/spring-util.xsd

在util Schema下提供了如下几个元素：

* constant：该元素用于获取指定类的静态Field的值。它是FieldRetrievingFactoryBean的简化配置。
* property-path：该元素用于获取指定对象的getter方法的返回值。它是propertyPathFactoryBean的简化配置。
* list：该元素用于定义一个List Bean，支持使用<value.../>、<ref.../>、<bean.../>等子元素来定义List集合元素。使用该标签支持如下三个属性：
  + id：该属性指定定义一个名为id的List Bean实例。
  + list-class：该属性指定Spring使用哪个List实现类来创建Bean实例。默认使用ArrayList作为实现类。
  + scope：指定该List Bean实例的作用域。
* set：该元素用于定义一个Set Bean，支持使用<value.../>、<ref.../>、<bean.../>等子元素来定义Set集合元素。使用该标签支持如下三个属性：
  + id：该属性指定定义一个名为id的Set Bean实例。
  + set-class：该属性指定Spring使用哪个Set实现类来创建Bean实例。默认使用HashSet作为实现类。
  + scope：指定该Set Bean实例的作用域。
* map：该元素用于定义一个Map Bean，支持使用<entry.../>来定义Map的key-value对。使用该标签支持如下三个属性：
  + id：该属性指定定义一个名为id的Map Bean实例。
  + map-class：该属性指定Spring使用哪个Map实现类来创建Bean实例。默认使用HashMap作为实现类。
  + scope：指定该Map Bean实例的作用域。
* properties：钙元素用于加载一份资源文件，并根据加载的资源文件创建一个properties Bean实例。使用该标签可指定如下几个属性：
  + id：该属性指定定义一个名为id的properties Bean实例。
  + location：指定资源文件位置。
  + scope：指定该properties Bean实例的作用域。

除此之外，关于Spring其他常用的简化Schema简要说明：

* spring-aop.xsd：用于简化Spring AOP配置的Schema。
* spring-jee.xsd：用于简化Spring的Java EE配置的Schema。
* spring-jms.xsd：用于简化Spring关于JMS配置的Schema。
* spring-lang.xsd：用于简化Spring动态语言配置的Schema。
* Spring-tx.xsd：用于简化Spring事务配置的Schema。

## Spring表达式语言（SpEL）

Spring表达式语言（简称SpEL）是一种与JSP2的EL功能类似的表达式语言，它可以在运行时查询和操作对象图。与JSP2的EL相比，SpEL功能更加强大，甚至支持方法调用和基本字符串模板函数。

SpEL可以独立于Spring容器使用，也可以在注解或XML配置中使用，这样可以充分利用SpEL简化Spring的Bean配置。

### 使用Expression接口进行表达式求值

Spring的SpEL提供了三个接口对表达式进行计算、求值：

#### SpEL提供的三个接口

* ExpressionParser：该接口的实例负责解析一个SpEL表达式，返回一个Expression对象。
* Expression：该接口的实例代表一个表达式。
* EvaluationContext：代表计算表达式值的上下文。当SpEL表达式中含有变量时，程序将需要使用该API来计算表达式的值。

#### Expression实例

Expression实例代表一个表达式，它包含了如下方法用于计算，得到表达式的值。

* Object getValue()：计算表达式的值。
* <T> T getValue(Class<T> desiredResultType)：计算表达式的值，而且尝试该表达式的值当成desiredResultType类型处理。
* Object getValue(EvaluationContext context)：使用指定的EvaluationContext来计算表达式的值。
* <T> T getValue(EvaluationContext context,Class<T> desriedResultType)：使用指定的EvaluationContext来计算表达式的值，而且尝试将该表达式的值当成desiredResultType类型处理。
* Object getValue(Object rootObject)：以rootObject作为表达式的root对象来计算表达式的值。
* <T> T getValue(Object rootObject,Class<T> desiredResultType)：以rootObject作为表达式的root对象来计算表达式的值，而且尝试将该表达式的值当成desiredResultType类型处理。

**public** **class** SpELTest {

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

//创建一个ExpressionParser对象，用于解析表达式

ExpressionParser parser = **new** SpelExpressionParser();

//最简单的字符串表达式

Expression exp = parser.parseExpression("'HelloWorld'");

System.***out***.println("'HelloWorld'的结果是：" + exp.getValue());

//调用方法的表达式

exp = parser.parseExpression("'HelloWorld'.concat('!')");

System.***out***.println("'HelloWorld'.concat('!')的结果是：" + exp.getValue());

//调用对象的getter方法

exp = parser.parseExpression("'HelloWorld'.bytes");

System.***out***.println("'HelloWorld'.bytes的结果是：" + exp.getValue());

//访问对象的属性(相当于HelloWorld.getBytes().length)

exp = parser.parseExpression("'HelloWorld'.bytes.length");

System.***out***.println("'HelloWorld'.bytes.length的结果是：" + exp.getValue());

//使用构造器来创建对象

exp = parser.parseExpression("new String('helloworld')" + ".toUpperCase()");

System.***out***.println("new String('helloworld')" + ".toUpperCase()的结果是：" +

exp.getValue(String.**class**));

Person p = **new** Person(1,"孙悟空",**new** Date());

exp = parser.parseExpression("name");

//以指定对象作为root来计算表达式的值

//相当于调用person.name表达式的值

System.***out***.println("以person为root,name表达式的值是：" + exp.getValue( p , String.**class**));

exp = parser.parseExpression("name=='孙悟空'");

StandardEvaluationContext ctx = **new** StandardEvaluationContext();

//将p设置为Context的root对象

ctx.setRootObject(p);

//以指定Context来计算表达式的值

System.***out***.println(exp.getValue(ctx, Boolean.**class**));

List<Boolean> list = **new** ArrayList<Boolean>();

(**true**);

EvaluationContext ctx2 = **new** StandardEvaluationContext();

//将list设置成EvaluationContext的一个变量

ctx2.setVariable("list", list);

//修改list变量的第一个元素的值

parser.parseExpression("#list[0]"). setValue(ctx2, "false");

//list集合的第一个元素被改变

System.***out***.println("list集合的第一个元素为：" + parser.parseExpression("#list[0]").getValue(ctx2));

}

}

上面代码使用ExpressionParser多次解析不同类型的表达式，ExpressionParser调动parserExpression()方法将返回一个Expression实例（表达式对象）。程序调用Expression对象的getValue()方法即可计算该表达式的值。

EvaluationContext代表SpEL计算表达式的值“上下文”，这个对象可以包含多个对象，但只能有一个root对象。

为了往EvaluationContext里放入对象，可以调用该对象的方法：

* setVariable(String name,Object value)，向EvaluationContext中放入value对象，该对象名为name，为了在SpEL中访问EvaluationContext中指定的对象，应采用与OGNL类似的格式：

#name

StandatdEvaluationContext提供了如下方法来设置root对象：

* setRootObject(Object rootObject)

在SpEL中访问root对象的属性时，可以省略root对象的前缀，例如：

foo.bar //访问rootObject的foo属性的bar属性

使用Expression对象计算表达式的值的时候，也可以直接指定root对象：

exp.getValue(person,String.class) //以person对象为root对象计算表达式的值。

#### Bean定义中的表达式语言支持

SpEL允许在Bean定义中使用SpEL，在XML配置文件和注解中都可以使用SpEL。在XML配置文件和注解中使用SpEL时，在表达式外面增加#{}包围即可。

### SpEL语法详述

#### 直接量表达式

直接量表达式就是在表达式中使用Java语言支持的直接量，包括字符串、日期、数值、boolean值和null。

#### 在表达式中创建数组

SpEL表达式直接支持使用静态初始化、动态初始化两种语法来创建数组。

//创建一个数组

exp = parser.parseExpression("new String[]{'java','Struts','Spring'}");

System.***out***.println(exp.getValue());

//创建一个二维数组

exp = parser.parseExpression("new int[2][4]");

System.***out***.println(exp.getValue());

#### 在表达式中创建List集合

SpEL直接使用如下语法来创建List集合：

{ele1 , ele2 , ele3 ......}

#### 在表达式中访问List、Map等集合元素

在SpEL中访问List集合的元素，可以使用如下语法格式：

List[index]

在SpEL中访问Map集合的元素，可以使用如下语法格式：

map[key]

List<String> l = **new** ArrayList<String>();

l.add("java");

l.add("Spring");

Map<String , Double> map = **new** HashMap<String , Double>();

map.put("java", 80.0);

map.put("Spring", 89.0);

ctx.setVariable("list", l);

ctx.setVariable("map", map);

System.***out***.println(parser.parseExpression("#list[1]").getValue(ctx));

System.***out***.println(parser.parseExpression("#map['java']").getValue(ctx));

#### 调用方法

在SpEL中调用方法与在Java代码中调用方法没有任何区别。

#### 算术、比较、逻辑、赋值、三目等运算符

在SpEL中使用赋值运算符的功能比较强大，这种赋值可以改变表达式所引用的实际对象。

#### 类型运算符

SpEL提供了一个特殊的运算符：T()，这个运算符告诉SpEL将该运算符内的字符串当成“类”处理，避免Spring对其进行其他的解析。尤其是调用某个类的静态方法时，T()运算符尤其有用。

在表达式中使用某个类时，推荐使用该类的全限定类名，但如果只写类名，不写包名，SpEL也可以尝试处理，SpEL使用StandardTypeLocator去定位这些类，默认会在java.lang包下找这些类。T()运算符在使用java.lang包下的类时可以省略包名，但是用其他包下的所有类时应使用全限定类名。

//调用Math的静态方法

System.***out***.println(parser.parseExpression("T(java.lang.Math).random()").getValue());

//调用Math的静态方法

System.***out***.println(parser.parseExpression("T(System).getProperty('os.name')").getValue());

#### 调用构造器

SpEL允许在表达式中直接使用new来调用构造器，这种调用可以创建一个Java对象。

//创建对象

System.***out***.println(parser.parseExpression("new javax.swing.JFrame('测试')" + ".setVisable('true')").getValue());

#### 变量

SpEL允许通过EvaluationContext来使用变量，该对象包含了一个setVariable(String name,Object value)方法，用于设置一个变量。

一旦设置变量，就可以在SpEL中通过#name来访问该变量。

SpEL中有两个特殊的变量：

* #this：引用SpEL当前正在计算的对象。
* #root：引用SpEL的EvaluationContext的root对象。

#### 自定义函数

SpEl允许开发者开发自己的函数，通过StandardEvaluationContext的如下方法即可在SpEL中注册自定义函数：

* registerFunction(String name,Method m)：将m方法注册成自定义函数，该函数名称为name。

自定义函数用处不大。

#### Elvis运算符

Elvis运算符只是三木运算符的特殊写法：

name != null ? name : “newVal”

SpEL允许将上面写法简写为：

name ?:”newVal”

#### 安全导航操作

在SpEL中使用如下语句可能导致NullPointerException:

foo.bar

如果root对象的foo属性本身已经是null，那么上面表达式尝试访问foo属性的bar属性时自然就会引发异常。

为了避免异常发生，SpEL支持如下用法：

foo?.bar

如果foo属性为null，计算结果直接返回null，而不会引发NullPointerException异常。

#### 集合选择

SpEL允许直接对集合进行选择操作，这种操作可以根据指定表达式对集合元素进行筛选，只有符合条件的集合元素才会被选择出来。

SpEL集合选择的语法格式如下：

collection.?[condition\_expr]

在上面的语法格式中，condition\_expr是一个根据集合元素定义的表达式，只有当该表达式返回true时，对应的集合元素才会被筛选出来。

EvaluationContext ec = **new** StandardEvaluationContext();

ec.setVariable("mylist",l);

Expression expr = parser.parseExpression("#mylist.?[length()>1]");

System.***out***.println(expr.getValue(ec));

当操作List集合时，condition\_expr中访问的每个属性、方法都是以集合元素为主调的；

当操作Map集合时，需要显式地用key引号MapEntry的key，用value引用Map Entry的value。

#### 集合投影

SeEL允许对集合进行投影运算，投影运算将依次迭代每个集合元素，迭代时将根据指定表达式对几何元素进行计算得到一个新的结果，一次奖每个结果收集成新的集合，这个集合将作为投影运算的结果。

SpEL投影运算的语法格式为：

conllection.![condition\_expr]

#### 表达式模板

表达式模板有点类似于带占位符的国际化消息。

表达式模板的本质是对“直接量表达式”的扩展，它允许在“直接量表达式”中插入一个或多个#{expr},#{expr}将会被动态计算出来。

使用ExpressionParser解析字符串模板时需要传入一个TemplateParserContext参数，该TemplateParserContext实现了ParserContext接口，它用于为表达式解析传入一些额外的信息。

## Spring两种后处理器

Spring框架提供了很好地扩展性，除了可以与各种第三方框架良好整合外，其IoC容器也允许开发者进行扩展，这种扩展甚至无需事先BeanFactory或ApplicationContext接口，而是允许通过两个后处理器对IoC容器进行扩展，Spring提供了两种常用的后处理器，这两种后处理器可以后处理IoC容器本身，或者对容器中所有Bean进行后处理。

* Bean后处理器：这种后处理器会对容器中的Bean进行后处理，对Bean进行额外加强。
* 容器后处理器：这种后处理器对IoC容器进行后处理，用于增强容器功能。

### Bean后处理器

如果需要对容器中某一批Bean进行通用的增强处理，则可以考虑使用Bean后处理器。

Bean后处理器是一种特殊的Bean，这种特殊的Bean并不对外提供服务，它甚至无需id属性，它会在Bean实例创建成功之后，对Bean进行进一步的增强处理，这种Bean被称为Bean后处理器。

Bean后处理器必须实现BeanPostProcessor接口，BeanPostProcessor接口包含如下两种方法：

* Object postProcessBeforeInitialization(Object bean,String name) throws BeansException：该方法第一个参数是系统即将进行后处理的Bean实例，第二个参数是该Bean的配置id。
* Object postprocessAfterInitialization(Object bean,String name) throws BeansException：该方法第一个参数是系统即将进行后处理的Bean实例，第二个参数是该Bean的配置id。

实现该接口的Bean后处理器必须实现这两个方法，这两个方法会对容器的Bean进行后处理，会在目标Bean初始化之前、初始化之后分别被回调，这两个方法用于对容器中的Bean实例进行增强处理。

Bean后处理器是对IoC容器一种极好的扩展，Bean后处理器可以对容器中的Bean进行后处理，而开发者决定到底要对Bean进行怎样的后处理。Spring容器负责把各Bean创建出来，Bean后处理器可以一次对每个Bean进行某种修改、增强，从而可以对容器中的Bean集中增加某种功能。

Spring提供了两个常用的后处理器：

* BeanNameAutoProxyCreator：根据Bean实例的name属性，创建Bean实例的代理。
* DefaultAdvisorAutoProxyCreator：根据提供的Advisor，对容器中的所有Bean实例创建代理。

这两个Bean后处理器都用于根据容器中配置的拦截器，创建代理Bean，代理Bean就是对目标Bean进行增强，在目标Bean的基础上进行修改得到的新Bean。

### 容器后处理器

容器后处理器负责处理容器本身。

容器后处理器必须事先BeanFactoryPostProcessor接口。实现该接口必须实现如下一个方法：

* postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)

实现该方法的方法体就是对Spring容器进行的处理，这种处理可以对Spring容器进行自定义扩展，当然也可以对Spring容器不进行任何处理。

实现BeanFactoryPostProcessor接口的容器后处理器不仅可以对BeanFactory执行后处理，也可以对ApplicationContext容器执行后处理，容器后处理器还可以用来注册额外的属性编辑器。

Spring以提供如下几个常用的容器后处理器。

* PropertyPlaceholderConfigurer：属性占位符配置器。
* PropertyOverrideConfigurer：重写占位符配置器。
* CustomAutowireConfigurer：自定义自动装配的配置器。
* CustomScopeConfigurer：自定义作用域的配置器。

为了给容器后处理器指定order属性，则要求容器后处理器必须事先Ordered接口，因此在事先BeanFactoryPostProcessor时，就应当考虑事先Ordered接口。

#### 属性占位符配置器

Spring提供了PropertyPlaceholderConfigurer,它是一个容器后处理器，负责读取Properties属性文件里的属性值，并将这些属性值设置成Spring配置文件的数据。

也可以使用简化配置<context:property-placeholder.../>元素

<context:property-placeholder location=”classpath:db.properties”/>

#### 重写占位符配置器

propertyOverrideConfigurer是Spring提供的另一个容器后处理器，其属性文件指定的信息可以直接覆盖Spring配置文件中的元数据。

当XML配置文件和属性文件指定的元数据不一致时，属性文件的信息取胜、

使用PropertyOverrideConfigurer的属性文件，每条属性应保持如下格式：

beanId.property=value

beanId是属性占位符试图覆盖的Bean的id，property是试图覆盖的属性名。

也可以使用简化配置<context:property-override.../>元素

<context:property-override location=”classpath:db.properties”/>

## 资源访问

Spring把文件、二进制流等统称为资源。在官方提供的标准API中，资源访问通常有java.net.URL和文件IO来完成，如果需要访问来自网络的资源，则通常会选择URL类。

URL类可以处理一些常规的资源访问问题，但依然不能很好地满足所有底层资源访问的需要。

Spring改进了Java资源访问的策略，Spring为资源访问提供了一个Resource接口，该接口提供了更强的资源访问能力，Spring本身大量使用了Resource来访问底层资源。

Resource本身是一个接口，是具体资源访问策略的抽象，也是所有资源访问类所实现的接口。Resource接口主要提供如下几个方法：

* getInputStream()：定位并打开资源，访问资源对应的输入流。每次调用都返回新的输入流，调用者必须负责关闭输入流。
* exists()：返回Resource所指向的资源是否存在。
* isOpen()：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束时应该显示关闭，以防止资源泄漏。
* getDescription()：返回资源的描述信息，用于资源处理出错时输出该信息，通常是全限定文件名或实际URL。
* getFile：返回资源对应的File对象。
* getURL：返回资源对应的URL对象。

最后两个方法通常无需使用，仅在通过简单方式访问无法实现时，Resource才提供传统的资源访问功能。

Resource接口本身没有提供访问任何底层资源的实现逻辑，针对不同的底层资源，Spring将会提供不同的Resource实现类，不同的实现类负责不同的资源访问逻辑。

Spring的Resource设计是一种典型的策略模式，通过使用Resource接口，客户端程序可以在不同的资源访问策略之间自由切换。

Resource不仅可以在Spring的项目中使用，也可以直接作为资源访问的工具类使用。使用Resource接口会让代码与Spring接口耦合在一起，但是这种耦合只是部分工具集的耦合，不会造成太大的代码污染。

### Resource实现类

Resource接口是Spring资源访问的接口，具体的资源访问由该接口的实现类完成。

Spring提供了Resource接口的大量实现类。

* UrlResource：访问网络资源的实现类。
* ClassPathResource：访问类加载路径里资源的实现类。
* FileSystemResource：访问文件系统里资源的实现类。
* ServletContextResource：访问相对于ServletContext路径下的资源的实现类。
* InputStreamResource：访问输入流资源的实现类。
* ByteArrayResource：访问字节数组资源的实现类。

#### 访问网络资源

访问网络资源通过UrlResource类实现，UrlResource是java.net.URL类的包装，主要用于访问通过URL类访问的资源对象。URL资源通常应该提供标准的协议前缀。

例如：

file：用于访问文件系统；

http：用于通过HTTP协议访问资源；

ftp：用于通过FTP协议访问资源等。

UrlResource类实现了Resource接口，对Resource的全部方法提供了实现，完全支持Resource的全部API。

## Spring的AOP

AOP(Aspect Orient Programming)，面向切面编程，作为面向对象编程的一种补充。AOP和OOP互为补充，作为面向对象编程将程序分解成各个层次的对象，而面向切面编程将程序运行过程分解成各个切面。

可以理解为：面向对象编程是从静态角度考虑程序结构，而面向切面编程则是从动态角度考虑程序运行过程。

### 为什么需要AOP

AOP专门用于处理系统中分布于各个模块（不同方法）中的交叉关注点的问题，在JavaEE应用中，常常通过AOP来处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等，AOP已经成为一种非常常用的解决方案。

### 使用AspectJ实现AOP

AspectJ是一个基于Java语言的AOP框架，提供了强大的AOP功能，其主要包括两个部分：一个部分定义了如何表达、定义AOP编程中的语法规范，通过这套语法规范，可以方便地使用AOP来解决Java语言中存在的交叉关注点的问题；另一个部分是工具部分，包括编译器、调试工具等。