**SPRING学习笔记**

# Spring IoC容器

## 依赖注入

哪些方面的控制被返回转了？

依赖对象的获取被反转了。

业务逻辑的实现是通过两个或两个以上的类的合作来实现的，这就使得每个对象都需要与其合作的对象（即其所依赖的对象）的引用，如果获取引用的这个过程要靠对象自身实现，这将导致代码耦合。

即由 对象 <<=>> 对象 转为 对象 <<=>> 容器 <<=>> 对象

## POJO

POJO（Plain Ordinary Java Object）普通JAVA类， 其中有一些属性及其getter setter方法的类,没有业务逻辑。

POJO与JavaBean区别：

**pojo的格式用于数据的临时传递，它只能装在数据，作为数据存储的载体，而不具有业务逻辑处理的能力。**

**javabean虽然数据的获取与pojo一样，但是javabean当中可以有其它的方法。**

**JavaBean 是一种JAVA语言写成的可重用组件。它的方法命名，构造及行为必须符合特定的约定：**

1.这个类必须有一个公共的缺省构造函数。

2.这个类的属性使用getter和setter来访问，其他方法遵从标准命名规范。

3.这个类应是可序列化的，实现Serializable接口。

## ****3. 注入方式****

接口注入：

setter注入：

构造器注入：

setter与构造器注入是主要注入方式。

## 4. 容器系列

容器是指一系列功能各异的容器产品，就好比水桶一样，有大小不同，材料不同等。

spring分为两个容器系列：

1. BeanFactory : 简单容器系列，实现了容器的最基本功能。
2. ApplicationContext ：应用上下文，容器的高级形态，具备许多面向框架的特性，同时对应用环境作了许多适配。

### 4.1 BeanFactory (Bean工厂)

**定义了容器的最基本的功能规范**，以水桶为例，**装水**是水桶的最基本功能。然后在此基础上可以根据不同的用户需求，设计各式各样的水桶。即所有的容器都必须满足BeanFactory这个基本接口的定义。

### 4.2 BeanDefinition (Bean定义)

管理Spring应用中的各种对象及其之间的依赖关系。容器是管理对象之间的依赖关系的，而这些依赖关系是通过对数据进行抽象来实现的，BeanDefinition就是对象依赖关系的数据抽象。就如同水桶中的水。

### 4.3 Spring容器的接口设计图

ApplicationEventPublisher

MessageSource

ResourceLoader

**ApplicationContext**

BeanFactory

AutowireCapableBeanFactory

ListableBeanFactory

ThemeSource

WebApplicationContext

ConfigurableApplicationContext

HierarchicalBeanFactory

ConfigurableBeanFactory

对接口关系可以从两条主线来分析：

1. BeanFactory 到HierarchicalBeanFactory(Hierarchical:层次，分层）再到ConfigurableBeanFactory接口。

BeanFactory定义了容器的基本功能；

HierarchicalBeanFactory增加了双亲的管理功能（即getParentBeanFactory()方法）；

ConfigurableBeanFactory定义了对BeanFactory的配置功能。

1. BeanFactory 到 ListableBeanFactory 到 ApplicationContext 到 WebApplicationContext 或 ConfigurableApplicationContext接口。

以上主要是接口关系，而具体的IoC容器都是实现DefaultListableBeanFactory实现的，它实现了ConfigurableBeanFactory，从而成为一个简单的IoC容器的实现。XmlBeanFactory和ApplicationContext都是在DefaultListableBeanFactory的基础上扩展实现的。

以上的接口系统以BeanFactory和ApplicationContext为核心。

### 4.4 BeanFactory

|  |
| --- |
| **package** org.springframework.beans.factory;  **import** org.springframework.beans.BeansException;  /\*\*  \* spring bean 容器的根接口  \* 定义了获取bean对象的方法.  \* 判断bean对象是单例的还是多例的,获取指定bean名称的bean的别名.获取bean对象的类型.  \*/  **public** **interface** BeanFactory {  /\*\*  \* 对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成的对象，  \* 如果需要得到工厂本身，需要转义  \*/  String ***FACTORY\_BEAN\_PREFIX*** = "&";  /\*\*  \* 根据bean的名字，获取在IOC容器中得到bean实例  \*/  Object getBean(String name) **throws** BeansException;  /\*\*  \* 根据bean的名字和Class类型来得到bean实例，增加了类型安全验证机制。  \*/  <T> T getBean(String name, Class<T> requiredType) **throws** BeansException;  /\*\*  \* 返回唯一匹配给定对象类型(如果有的话)的bean实例。  \*/  <T> T getBean(Class<T> requiredType) **throws** BeansException;  /\*\*  \*  \*/  Object getBean(String name, Object... args) **throws** BeansException;  /\*\*  \* 提供对bean的检索，看看是否在IOC容器有这个名字的bean  \*/  **boolean** containsBean(String name);  /\*\*  \* 根据bean名字得到bean实例，并同时判断这个bean是不是单例，**对于Singleton属性，可以在BeanDefinition中指定**。  \*/  **boolean** isSingleton(String name) **throws** NoSuchBeanDefinitionException;  /\*\*  \* 查询指定名称的Bean是否是propertype类型，也可以在BeanDefinition中指定。  \*/  **boolean** isPrototype(String name) **throws** NoSuchBeanDefinitionException;  /\*\*  \* 指定名称的Bean是否是特定的Class类型。  \*/  **boolean** isTypeMatch(String name, Class<?> targetType) **throws** NoSuchBeanDefinitionException;  /\*\*  \* 得到指定名称的bean实例的Class类型  \*/  Class<?> getType(String name) **throws** NoSuchBeanDefinitionException;  /\*\*  \* 得到bean的别名，如果根据别名检索，那么其原名也会被检索出来  \*/  String[] getAliases(String name);  } |

BeanFactory提供的是最基本的IoC容器功能。而DefaultListableBeanFactory、XmlBeanFactory、ApplicationContext都是容器体系中的具体的容器产品。

1. “&”：BeanFactory接口中的常量***FACTORY\_BEAN\_PREFIX***，作用是用来获取FactoryBean本身，例如myJndiObject是一个FactoryBean，那么使用&myJndiObject得到的是FactoryBean，而不是myJndiObject这个由FactoryBean产生出来的对象。
2. getBean（）：获取IOC容器中管理的Bean，是通过指定名字来索引的，如果在获取Bean时对Bean的类型进行检查，则使用带有参数的getBean方法。

### 4.5 XmlBeanFactory的实现分析

XmlBeanFactory继承了DefaultListableBeanFactory类，后者包含了基本IOC容器所具有的重要功能，可以说是容器系列中的一个基本产品。在Spring中，实际上是把DefaultListableBeanFactory作为一个默认的功能完整的IOC容器来使用的。XmlBeanFactory在继承它的同时增加了新的功能。

XmlBeanFactory是一个可以读取xml文件方式定义的BeanDefinition的IOC容器，那么xml信息读取是如何实现的，在XmlBeanFactory中，初始化了一个XmlBeanDefinitionReader对象，由它来完成的。

AutowireCapableBeanFactory

ConfigurableListableBeanFactory

BeanDefinitionRegistry

DefaultlistableBeanFactory

AbstractAutowireCapableBeanFactory

AbstractBeanFactory

ConfigurableBeanFactory

XmlBeanFactory

|  |
| --- |
| **package** org.springframework.beans.factory.xml;  **import** org.springframework.beans.BeansException;  **import** org.springframework.beans.factory.BeanFactory;  **import** org.springframework.beans.factory.support.DefaultListableBeanFactory;  **import** org.springframework.core.io.Resource;  /\*\*  \*  \*/  @Deprecated  @SuppressWarnings({"serial", "all"})  **public** **class** ~~XmlBeanFactory~~ **extends** DefaultListableBeanFactory {  /\*\*  \* 完成xml文件读取功能  \*/  **private** **final** XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(**this**);  /\*\*  \* Create a new XmlBeanFactory with the given resource,  \* which must be parsable using DOM.  \* **@param** resource XML resource to load bean definitions from  \* **@throws** BeansException in case of loading or parsing errors  \*/  **public** XmlBeanFactory(Resource resource) **throws** BeansException {  **this**(resource, **null**);  }  /\*\*  \* Create a new XmlBeanFactory with the given input stream,  \* which must be parsable using DOM.  \* **@param** resource XML resource to load bean definitions from  \* **@param** parentBeanFactory parent bean factory  \* **@throws** BeansException in case of loading or parsing errors  \*/  **public** XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) **throws** BeansException {  **super**(parentBeanFactory);  **this**.~~reader~~.loadBeanDefinitions(resource);  }  } |

在构造XmlBeanFactory这个IOC容器时，要指定BeanDefinition的信息来源，这个来源由Resource类给出，Resource是Spring用来封装IO操作的类，例如BeanDefinition信息是以xml形式存在，那么可以使用像ClassPathResource res = new ClassPathResource(“bean.xml”)构造出Resourc类对象，然后将其作为构造函数传给XmlBeanFactory， 这样IOC容器就可以方便的定义到BeanDefinition信息来完成对容器的初始化。通过XmlBeanDefinitionReader的loadBeanDefinitions()方法的调用，完成从Resourc中载入BeanDefinition的过程。

编程式使用以上的IOC容器：

|  |
| --- |
| ClassPathResource resource = **new** ClassPathResource("bean.xml");//加载资源  DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();//初始化beanFactory  //创建一个BeanDefinition读取器  XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory); reader.loadBeanDefinitions(resource);//从定义好的资源位置读入配置信息 |

### 4.6 ApplicationContext

这是一个高级形态意义的IOC容器。在BeanFactory的基础上添加的附加功能如下：

1. 支持不同的信息源。扩展了MessageSource接口，可以支持国际化的实现，为多语言开发提供服务。
2. 访问资源。体现在ResourceLoader和Resource的支持上，可以从不同地方得到Bean定义资源。
3. 支持应用事件。继承了接口ApplicationEventPublisher，在上下文中引入了事件机制，这些事件和Bean的生命周期的结合为Bean的管理提供了便利。
4. 在ApplicationContext中提供的附加服务使得基本IOC容器的功能更丰富。所以在开发时建议用其作为IOC容器的基本形式。

**4.7 FileSystemXmlApplicationContext实现分析**

此分析用来说明ApplicationContext的设计原理。

FileSystemXmlApplicationContext的主要功能已在AbstractXmlApplicationContext中实现了，而FileSystemXmlApplicationContext作为一个具体的容器，只需要实现与其自身相关的两个功能。

一是如果直接使用它，对于实例化这个应用上下文的支持，同时启动IOC容器的refresh()过程。这个refresh会牵涉IOC容器启动的一系列复杂操作，同时，对于不同的容器实现，过程是类似的。因此在基类中将它们封装好。

|  |
| --- |
| **public** FileSystemXmlApplicationContext(String[] configLocations, **boolean** refresh, ApplicationContext parent)  **throws** BeansException {  **super**(parent);  setConfigLocations(configLocations);  **if** (refresh) {  refresh();  }  } |

二是与FileSystemXmlApplicationContext设计具体相关的功能，即怎样加载xml，即在文件中读取xml。

|  |
| --- |
| **protected** Resource getResourceByPath(String path) {  **if** (path != **null** && path.startsWith("/")) {  path = path.substring(1);  }  **return** **new** FileSystemResource(path);  } |

**4.8 IOC容器的初始化过程**

简单说：IOC容器的初始化是由**refresh()**方法来启动的，这标志着IOC容器的正式启动。这个启动包括三个过程：BeanDefinition的Resource定位、载入和注册三个基本过程。

第一个过程：Resource定位过程。由ResourceLoader通过统一的Resource接口完成，对各种形式的BeanDefinition的使用提供了统一接口。如在文件系统中使用FileSystemResource来定位，在类路径中使用ClassPathResource来定位。如同用水桶装水要先把水找到一样。

第二个过程：BeanDefinition载入。通过BeanDefinition定义的数据结构对Bean对象进行管理。

第三个过程：向IOC容器注册这些BeanDefinition。这个过程是调用BeanDefinitionRegistry接口的实现来完成的。即将解析得到的BeanDefinition向IOC容器进行注册，即注入到一个HashMap中去。

注意：这个过程中不包含Bean依赖注入的实现。在Spring IOC的设计中，Bean定义的载入与依赖注入是两个独立的过程，依赖注入一般发生在应用第一次通过getBean向容器索取Bean的时候。但是如果在使用IOC时进行了预实例化配置（即Bean定义的lazyinit属性），则对容器初始化过程作一个微小的控制，即这个Bean的依赖注入在IOC容器初始化时就预先完成了，而不是等到第一次getBean时才会触发。

### 4.9 BeanDefinition的Resource定位