**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

以FileSystemXmlApplicationContext为例：

先看类图：

AbstractRefreshableApplicationContext

InitializingBean

BeanNameAware

AbstractRefreshableConfigApplicationContext

AbstractXmlApplicationContext

FileSystemXmlApplicationContext

ConfigurableApplicationContext

DisposableBean

AbstractApplicationContext

ResourceLoader

DefaultResourceLoader

FileSystemXmlApplicationContext通过继承AbstractApplicationContext具备了ResourceLoader读入以Resource定义的BeanDefinition的能力，因为AbstractApplicationContext的基类是DefaultResourceLoader。

FileSystemXmlApplicationContext的具体实现：

|  |
| --- |
| **package** org.springframework.context.support;  **import** org.springframework.beans.BeansException;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.core.io.FileSystemResource;  **import** org.springframework.core.io.Resource;  /\*\*  \*  \*/  **public** **class** FileSystemXmlApplicationContext **extends** AbstractXmlApplicationContext {  /\*\*  \* Create a new FileSystemXmlApplicationContext for bean-style configuration.  \*/  **public** FileSystemXmlApplicationContext() {  }  /\*\*  \* Create a new FileSystemXmlApplicationContext for bean-style configuration.  \* **@param** parent the parent context  \*/  **public** FileSystemXmlApplicationContext(ApplicationContext parent) {  **super**(parent);  }  /\*\*  \* Create a new FileSystemXmlApplicationContext, loading the definitions  \* from the given XML file and automatically refreshing the context.  \* **configLocation 包含的是BeanDefinition所在的文件路径**  \*/  **public** FileSystemXmlApplicationContext(String configLocation) **throws** BeansException {  **this**(**new** String[] {configLocation}, **true**, **null**);  }  /\*\*  \* Create a new FileSystemXmlApplicationContext, loading the definitions  \* from the given XML files and automatically refreshing the context.  \* **@param** configLocations 包含多个BeanDefinition的文件路径  \*/  **public** FileSystemXmlApplicationContext(String... configLocations) **throws** BeansException {  **this**(configLocations, **true**, **null**);  }  /\*\*  \* Create a new FileSystemXmlApplicationContext with the given parent,  \* loading the definitions from the given XML files and automatically  \* refreshing the context.  \* **@param** configLocations array of file paths  \* **@param** parent the parent context 双亲IOC容器  \*/  **public** FileSystemXmlApplicationContext(String[] configLocations, ApplicationContext parent) **throws** BeansException {  **this**(configLocations, **true**, parent);  }  /\*\*  \* Create a new FileSystemXmlApplicationContext, loading the definitions  \* from the given XML files.  \*/  **public** FileSystemXmlApplicationContext(String[] configLocations, **boolean** refresh) **throws** BeansException {  **this**(configLocations, refresh, **null**);  }  /\*\*  \* Create a new FileSystemXmlApplicationContext with the given parent,  \* loading the definitions from the given XML files.  \* **在对象的初始化过程中**，调用refresh函数载入BeanDefinition,它启动了BeanDefinition的载入过程  \*/  **public** FileSystemXmlApplicationContext(String[] configLocations, **boolean** refresh, ApplicationContext parent)  **throws** BeansException {  **super**(parent);  setConfigLocations(configLocations);  **if** (refresh) {  refresh();  }  }  /\*\*  \* Resolve resource paths as file system paths.  \* <p>Note: Even if a given path starts with a slash, it will get  \* interpreted as relative to the current VM working directory.  \* This is consistent with the semantics in a Servlet container.  \* 文件系统中Resource的实现，通过FileSystemResource来得到一个在文件系统中定位的BeanDefinition  \* 该方法是BeanDefinitionReader的loadBeanDefinition中被调用的loadBeanDefinition采用了模板模式，具体  \* 实现实际上是由各个子类来完成的。  \*/  @Override  **protected** Resource getResourceByPath(String path) {  **if** (path != **null** && path.startsWith("/")) {  path = path.substring(1);  }  **return** **new** FileSystemResource(path);  }  } |

在FileSystemApplicationContext中，在其构造函数中，实现了对configuration进行处理的功能，让所有配置在文件系统中的，以XML文件方式存在的BeanDefinition都能够得到有效处理，如实现了getResourceByPath方法，它是为读取Resource服务的。构造函数中的refresh方法来启动容器的初始化。

BeanDefinition的载入过程，见AbstractBeanDefinitionReader类中的loadBeanDefinitions()方法：

|  |
| --- |
| **public** **abstract** **class** AbstractRefreshableApplicationContext **extends** AbstractApplicationContext {  **private** Boolean allowBeanDefinitionOverriding;  **private** Boolean allowCircularReferences;  /\*\* Bean factory for this context \*/  **private** DefaultListableBeanFactory beanFactory;  /\*\* Synchronization monitor for the internal BeanFactory \*/  **private** **final** Object beanFactoryMonitor = **new** Object();  /\*\*  \* Create a new AbstractRefreshableApplicationContext with no parent.  \*/  **public** AbstractRefreshableApplicationContext() {  }  /\*\*  \* Create a new AbstractRefreshableApplicationContext with the given parent context.  \* **@param** parent the parent context  \*/  **public** AbstractRefreshableApplicationContext(ApplicationContext parent) {  **super**(parent);  }  /\*\*  \* Set whether it should be allowed to override bean definitions by registering  \* a different definition with the same name, automatically replacing the former.  \* If not, an exception will be thrown. Default is "true".  \* **@see** org.springframework.beans.factory.support.DefaultListableBeanFactory#setAllowBeanDefinitionOverriding  \*/  **public** **void** setAllowBeanDefinitionOverriding(**boolean** allowBeanDefinitionOverriding) {  **this**.allowBeanDefinitionOverriding = allowBeanDefinitionOverriding;  }  /\*\*  \* Set whether to allow circular references between beans - and automatically  \* try to resolve them.  \* <p>Default is "true". Turn this off to throw an exception when encountering  \* a circular reference, disallowing them completely.  \* **@see** org.springframework.beans.factory.support.DefaultListableBeanFactory#setAllowCircularReferences  \*/  **public** **void** setAllowCircularReferences(**boolean** allowCircularReferences) {  **this**.allowCircularReferences = allowCircularReferences;  }  /\*\*  \* This implementation performs an actual refresh of this context's underlying  \* bean factory, shutting down the previous bean factory (if any) and  \* initializing a fresh bean factory for the next phase of the context's lifecycle.  \*/  @Override  **protected** **final** **void** refreshBeanFactory() **throws** BeansException {  **if** (hasBeanFactory()) { // 如果已经有容器，销毁并关闭容器该BeanFactory  destroyBeans();  closeBeanFactory();  }  //  **try** {  //创建IoC容器  DefaultListableBeanFactory beanFactory = createBeanFactory();  beanFactory.setSerializationId(getId());  //对IoC容器进行定制化，如设置启动参数，开启注解的自动装配等  customizeBeanFactory(beanFactory);  //载入Bean定义  loadBeanDefinitions(beanFactory);  **synchronized** (**this**.beanFactoryMonitor) {  **this**.beanFactory = beanFactory;  }  }  **catch** (IOException ex) {  **throw** **new** ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);  }  }  @Override  **protected** **void** cancelRefresh(BeansException ex) {  **synchronized** (**this**.beanFactoryMonitor) {  **if** (**this**.beanFactory != **null**)  **this**.beanFactory.setSerializationId(**null**);  }  **super**.cancelRefresh(ex);  }  @Override  **protected** **final** **void** closeBeanFactory() {  **synchronized** (**this**.beanFactoryMonitor) {  **this**.beanFactory.setSerializationId(**null**);  **this**.beanFactory = **null**;  }  }  /\*\*  \* Determine whether this context currently holds a bean factory,  \* i.e. has been refreshed at least once and not been closed yet.  \*/  **protected** **final** **boolean** hasBeanFactory() {  **synchronized** (**this**.beanFactoryMonitor) {  **return** (**this**.beanFactory != **null**);  }  }  @Override  **public** **final** ConfigurableListableBeanFactory getBeanFactory() {  **synchronized** (**this**.beanFactoryMonitor) {  **if** (**this**.beanFactory == **null**) {  **throw** **new** IllegalStateException("BeanFactory not initialized or already closed - " +  "call 'refresh' before accessing beans via the ApplicationContext");  }  **return** **this**.beanFactory;  }  }  /\*\*  \* Create an internal bean factory for this context.  \* Called for each {@link #refresh()} attempt.  \* <p>The default implementation creates a  \* {@link org.springframework.beans.factory.support.DefaultListableBeanFactory}  \* with the {@linkplain #getInternalParentBeanFactory() internal bean factory} of this  \* context's parent as parent bean factory. Can be overridden in subclasses,  \* for example to customize DefaultListableBeanFactory's settings.  \* **@return** the bean factory for this context  \* 这里就是在上下文中创建DefaultListableBeanFactory的地方，而getInternalParentBeanFactory的具体实现  \* 可以。  \*/  **protected** DefaultListableBeanFactory createBeanFactory() {  **return** **new** DefaultListableBeanFactory(getInternalParentBeanFactory());  }  /\*\*  \* Customize the internal bean factory used by this context.  \* Called for each {@link #refresh()} attempt.  \* <p>The default implementation applies this context's  \* {@linkplain #setAllowBeanDefinitionOverriding "allowBeanDefinitionOverriding"}  \* and {@linkplain #setAllowCircularReferences "allowCircularReferences"} settings,  \* if specified. Can be overridden in subclasses to customize any of  \* {@link DefaultListableBeanFactory}'s settings.  \*/  **protected** **void** customizeBeanFactory(DefaultListableBeanFactory beanFactory) {  **if** (**this**.allowBeanDefinitionOverriding != **null**) {  beanFactory.setAllowBeanDefinitionOverriding(**this**.allowBeanDefinitionOverriding);  }  **if** (**this**.allowCircularReferences != **null**) {  beanFactory.setAllowCircularReferences(**this**.allowCircularReferences);  }  beanFactory.setAutowireCandidateResolver(**new** QualifierAnnotationAutowireCandidateResolver());  }  /\*\*  \* Load bean definitions into the given bean factory, typically through  \* delegating to one or more bean definition readers.  \* **@param** beanFactory the bean factory to load bean definitions into  \* **@throws** BeansException if parsing of the bean definitions failed  \* **@throws** IOException if loading of bean definition files failed  \* 这里是使用载入Bean定义的地方  \*  \*/  **protected** **abstract** **void** loadBeanDefinitions(DefaultListableBeanFactory beanFactory)  **throws** BeansException, IOException;  } |

以上过程就如同用水桶打水时已经找到水源了。接下来就是打水的过程了。

### 4.10 BeanDefinition的载入和解析

这个过程相当于把定义的BeanDefinition在IOC容器中转化成一个Spring内部表示的数据结构的过程。这些数据在容器中通过一个HashMap来保持和维护。

先从DefaultListableBeanFactory的设计开始，看IOC容器是怎样完成BeanDefinition的载入的。

IOC容器初始化的入口，refresh方法，最初在FileSystemXmlApplicationContext的构造函数中被调用的，标志着初始化的开始，这些初始化的数据就是BeanDefinition数据。refresh方法的具体实现，见AbstractApplicationContext类。

代码清单：

|  |
| --- |
| **public** **void** refresh() **throws** BeansException, IllegalStateException {  **synchronized** (**this**.startupShutdownMonitor) {  // Prepare this context for refreshing.  // 调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识  prepareRefresh();  // Tell the subclass to refresh the internal bean factory.  // 告诉子类启动refreshBeanFactory()方法，Bean定义资源文件的载入从  // 子类的refreshBeanFactory()方法启动  ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();  // Prepare the bean factory for use in this context.  // 为BeanFactory配置容器特性，例如类加载器、事件处理器等  prepareBeanFactory(beanFactory);  **try** {  // 为容器的某些子类指定特殊的BeanPost事件处理器  // 设置BeanFactroy的后置处理  postProcessBeanFactory(beanFactory);    // 调用BeanFactory的后处理器，这些后处理器是在Bean定义中向容器注册的  invokeBeanFactoryPostProcessors(beanFactory);    // 注册Bean的后处理器，在Bean创建过程中调用。  registerBeanPostProcessors(beanFactory);  // Initialize message source for this context.  // 初始化上下文中的信息源，和国际化相关.  initMessageSource();  // Initialize event multicaster for this context.  // 初始化容器事件机制  initApplicationEventMulticaster();  // Initialize other special beans in specific context subclasses.  // 初始化其他特殊的Bean  onRefresh();  // Check for listener beans and register them.  // 检查监听Bean并将这些Bean向容器注册.  registerListeners();  // Instantiate all remaining (non-lazy-init) singletons.  // 初始化所有的单态Bean.  finishBeanFactoryInitialization(beanFactory);  // Last step: publish corresponding event.  //发布容器的生命周期事件，结束refresh过程  finishRefresh();  }  **catch** (BeansException ex) {  **if** (logger.isWarnEnabled()) {  logger.warn("Exception encountered during context initialization - "  + "cancelling refresh attempt: " + ex);  }  // Destroy already created singletons to avoid dangling resources.  // 销毁已创建的单态Bean  destroyBeans();  // Reset 'active' flag.  // 取消refresh操作，重置容器的状态标识.  cancelRefresh(ex);  // Propagate exception to caller.  **throw** ex;  }  }  } |

进入到AbstractRefreshableApplicationContext的refreshBeanFactory方法，此处创建了BeanFactory，在创建IOC容器前，如果已有容器存在，那么要把已有的容器销毁和关闭，保证在refresh以后使用的是新建立起来的IOC容器，由此看来，这个refresh如同重启容器。

以下为refreshBeanFactory方法：

|  |
| --- |
| **protected** **final** **void** refreshBeanFactory() **throws** BeansException {  **if** (hasBeanFactory()) { // 如果已经有容器，销毁容器中的bean，关闭容器  destroyBeans();  closeBeanFactory();  }  **try** {  //创建IoC容器  DefaultListableBeanFactory beanFactory = createBeanFactory();  beanFactory.setSerializationId(getId());  //对IoC容器进行定制化，如设置启动参数，开启注解的自动装配等  customizeBeanFactory(beanFactory);  //调用载入Bean定义的方法  loadBeanDefinitions(beanFactory);  **synchronized** (**this**.beanFactoryMonitor) {  **this**.beanFactory = beanFactory;  }  }  **catch** (IOException ex) {  **throw** **new** ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);  }  } |

此处的loadBeanDefinitions()实际上是一个抽象方法，那么实际的载入过程发生在哪里呢？看看loacBeanDefinitions在AbstractRefreshableApplicationContext的子类AbstractXmlApplicationContext中的实现，在这里初始化了读取器XmlBeanDefinitionReader，然后把这个读取吕在IOC容器中设置好，最后是启动读取器来完成BeanDefinition在IOC容器中的载入。

AbstractXmlApplicationContext类中的loadBeanDefinitions方法：

|  |
| --- |
| **protected** **void** loadBeanDefinitions(DefaultListableBeanFactory beanFactory)  **throws** BeansException, IOException {  // Create a new XmlBeanDefinitionReader for the given BeanFactory.  // 创建XmlBeanDefinitionReader，即创建Bean读取器，并通过回调设置到容器中去，容器使用该读取器读取Bean定义资源  XmlBeanDefinitionReader beanDefinitionReader = **new** XmlBeanDefinitionReader(  beanFactory);  // Configure the bean definition reader with this context's  // resource loading environment.  // 为Bean读取器设置Spring资源加载器，AbstractXmlApplicationContext的祖先父类AbstractApplicationContext继承DefaultResourceLoader，因此，容器本身也是一个资源加载器  beanDefinitionReader.setEnvironment(**this**.getEnvironment());  beanDefinitionReader.setResourceLoader(**this**);  //为Bean读取器设置SAX xml解析器  beanDefinitionReader.setEntityResolver(**new** ResourceEntityResolver(**this**));  // Allow a subclass to provide custom initialization of the reader,  // then proceed with actually loading the bean definitions.  //当Bean读取器读取Bean定义的Xml资源文件时，启用Xml的校验机制  // 这是启动Bean定义信息载入的过程  initBeanDefinitionReader(beanDefinitionReader);  //Bean读取器真正实现加载的方法  loadBeanDefinitions(beanDefinitionReader);  }  **protected** **void** loadBeanDefinitions(XmlBeanDefinitionReader reader)  **throws** BeansException, IOException {  // 获取Bean定义资源的定位  Resource[] configResources = getConfigResources();  **if** (configResources != **null**) {  //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位 的Bean定义资源  reader.loadBeanDefinitions(configResources);  }  //以String形式获得配置文件的位置  String[] configLocations = getConfigLocations();  **if** (configLocations != **null**) {  //如果子类中获取的Bean定义资源定位为空，则获取FileSystemXmlApplicationContext构造方法中setConfigLocations方法设置的资源  reader.loadBeanDefinitions(configLocations);  }  } |

接着就是loadBeanDefinitions调用的地方，首先得到BeanDefinition信息的Resource定位，然后直接调用XmlBeanDefinitionReader来读取，具体的载入过程是委托给BeanDefinitionReader完成的。因为这里的BeanDefinition是通过XML文件定义的，所以这里使用XmlBeanDefinitionReader来载入BeanDefinition到容器中。而实际使用的容器是DefaultListableBeanFactory. 如果使用了其它的BeanDefinition方式，就需要使用其它种类的BeanDefinitionReader来完成数据的载入工作。载入工作是在reader的loadBeanDefinitions中开始进行的。参见AbstractBeanDefinitionReader中的

loadBeanDefinitions方法：

|  |
| --- |
| **public** **int** loadBeanDefinitions(Resource... resources) **throws** BeanDefinitionStoreException {  // 如果为空，则停止载入  Assert.*notNull*(resources, "Resource array must not be null");  // 载入过程会遍历整个Resource数组所包含的BeanDefinition信息  **int** counter = 0;  **for** (Resource resource : resources) {  counter += loadBeanDefinitions(resource);  }  **return** counter;  } |

这里调用 的是loadBeanDefinitions(Resource res)方法，但这个方法没有在这里实现，具体实现是在XmlBeanDefinitionReader中。在读取器中，得到代表XML文件的Resource， 以获取IO流并解析XML文档，解析工作是由BeanDefinitionParserDelegate来完成。具体如下：

|  |
| --- |
| **// 调用入口**  **public** **int** loadBeanDefinitions(Resource resource) **throws** BeanDefinitionStoreException {  // 将读入的XML资源进行特殊编码处理  **return** loadBeanDefinitions(**new** EncodedResource(resource));  }  /\*\*  \* Load bean definitions from the specified XML file.  \* **@param** encodedResource the resource descriptor for the XML file,  \* allowing to specify an encoding to use for parsing the file  \* **@return** the number of bean definitions found  \* **@throws** BeanDefinitionStoreException in case of loading or parsing errors  \* 这里是载入XML形式BeanDefinition文件方法  \*/  **public** **int** loadBeanDefinitions(EncodedResource encodedResource) **throws** BeanDefinitionStoreException {  Assert.*notNull*(encodedResource, "EncodedResource must not be null");  **if** (logger.isInfoEnabled()) {  logger.info("Loading XML bean definitions from " + encodedResource.getResource());  }  Set<EncodedResource> currentResources = **this**.resourcesCurrentlyBeingLoaded.get();  **if** (currentResources == **null**) {  currentResources = **new** HashSet<EncodedResource>(4);  **this**.resourcesCurrentlyBeingLoaded.set(currentResources);  }  **if** (!currentResources.add(encodedResource)) {  **throw** **new** BeanDefinitionStoreException(  "Detected cyclic loading of " + encodedResource + " - check your import definitions!");  }  // 得到XML文件  **try** {  //将资源文件转为InputStream的IO流  InputStream inputStream = encodedResource.getResource().getInputStream();  **try** {  //从InputStream中得到XML的解析源  InputSource inputSource = **new** InputSource(inputStream);  **if** (encodedResource.getEncoding() != **null**) {  inputSource.setEncoding(encodedResource.getEncoding());  }  //这里是具体的读取过程  **return** doLoadBeanDefinitions(inputSource, encodedResource.getResource());  }  **finally** {  //关闭从Resource中得到的IO流  inputStream.close();  }  }  **catch** (IOException ex) {  **throw** **new** BeanDefinitionStoreException(  "IOException parsing XML document from " + encodedResource.getResource(), ex);  }  **finally** {  currentResources.remove(encodedResource);  **if** (currentResources.isEmpty()) {  **this**.resourcesCurrentlyBeingLoaded.remove();  }  }  }  //具体的读取过程可以在doLoadBeanDefinitions方法中找到，这是从特定的XML文件中实际载入BeanDefinition的地方。  **protected** **int** doLoadBeanDefinitions(InputSource inputSource, Resource resource)  **throws** BeanDefinitionStoreException {  **try** {  **int** validationMode = getValidationModeForResource(resource);  //将XML文件转换为DOM对象，解析过程由documentLoader实现  Document doc = **this**.documentLoader.loadDocument(  inputSource, getEntityResolver(), **this**.errorHandler, validationMode, isNamespaceAware());  //这里是启动对Bean定义解析的详细过程，该解析过程会用到Spring的Bean配置规则  **return** registerBeanDefinitions(doc, resource);  }  **catch** (BeanDefinitionStoreException ex) {  **throw** ex;  }  **catch** (SAXParseException ex) {  **throw** **new** XmlBeanDefinitionStoreException(resource.getDescription(),  "Line " + ex.getLineNumber() + " in XML document from " + resource + " is invalid", ex);  }  **catch** (SAXException ex) {  **throw** **new** XmlBeanDefinitionStoreException(resource.getDescription(),  "XML document from " + resource + " is invalid", ex);  }  **catch** (ParserConfigurationException ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "Parser configuration exception parsing XML from " + resource, ex);  }  **catch** (IOException ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "IOException parsing XML document from " + resource, ex);  }  **catch** (Throwable ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "Unexpected exception parsing XML document from " + resource, ex);  }  } |

那么Spring的BeanDefinition是怎样按照Spring的Bean语义要求进行解析并转化为容器内部数据结构的，这个过程是在registerBeanDefinitions(doc, resource)中完成的。具体过程是由BeanDefinitionDocumentReader来完成的，同时还对载入的Bean的数量进行了统计。

|  |
| --- |
| **public** **int** registerBeanDefinitions(Document doc, Resource resource) **throws** BeanDefinitionStoreException {  //得到BeanDefinitionDocumentReader来对xml格式的BeanDefinition解析  BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  documentReader.~~setEnvironment~~(getEnvironment());  //获得容器中注册的Bean数量  **int** countBefore = getRegistry().getBeanDefinitionCount();  //解析过程入口  documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  //统计解析的Bean数量  **return** getRegistry().getBeanDefinitionCount() - countBefore;  } |

BeanDefinition的载入分为两部分，先通过调用XML的解析器得到document对象，但这些对象并没有按照Spring的Bean规则进行解析。在完成通用解析后，才是按照Spring的Bean规则进行解析的地方，这个过程是在documentReader中实现的。这里使用的documentReader是默认设置好的DefaultBeanDefinitionDocumentReader, 它的创建是在后面的方法中完成的，然后再完成BeanDefinition的处理，处理的结果由BeanDefinitionHolder对象来持有。它除了持在BeanDefinition对象外，还持有其他与BeanDefinition的使用相关的信息，如Bean的名字，别名等。这个解析过程是由BeanDefinitionParserDelegate来实现（具体在processBeanDefinition方法中实现）的，这个解析是与Spring对BeanDefinition的配置规则 紧密相关的。

创建BeanDefinitionDocumentReader:

|  |
| --- |
| **protected** BeanDefinitionDocumentReader createBeanDefinitionDocumentReader() {  **return** BeanDefinitionDocumentReader.**class**.cast(BeanUtils.*instantiateClass*(**this**.documentReaderClass));  }  // 在得到documentReader以后，为具体的Spring Bean 的解析过程准备好了数据  // 这里是处理BeanDefinition的地方，具体的处理委托给BeanDefinitionParserDelegate来完成，ele对应在BeanDefinition中定义的XML元素。  **protected** **void** processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {  // BeanDefinitionHolder是对BeanDefinition的封装，即BeanDefinition、Bean的名字、别名的封装类  //对Document对象中<Bean>元素的解析由BeanDefinitionParserDelegate实现  BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);    **if** (bdHolder != **null**) {  bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);  **try** {  // Register the final decorated instance.  //向Spring IoC容器注册解析得到的Bean定义，这是Bean定义向IoC容器注册的入口  BeanDefinitionReaderUtils.*registerBeanDefinition*(bdHolder, getReaderContext().getRegistry());  }  **catch** (BeanDefinitionStoreException ex) {  getReaderContext().error("Failed to register bean definition with name '" +  bdHolder.getBeanName() + "'", ele, ex);  }  // Send registration event.  //在完成向Spring IoC容器注册解析得到的Bean定义之后，发送注册事件  getReaderContext().fireComponentRegistered(**new** BeanComponentDefinition(bdHolder));  }  } |

BeanDefinitionParserDelegate类包含了对各种Spring Bean定义规则的处理，如常见的<bean></bean>元素以及id，name，aliase等元素属性。将这些值从xml文件中读出以后设置到BeanDefinitionHolder中去，这些是简单的解析。对于其他元素配置的解析，如各种bean的属性配置是一个较为复杂的过程 ，是通过parseBeanDefinitionElement来完成，并将解析结果放到BeanDefinitionHolder中去。代码如下：

|  |
| --- |
| **public** BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition containingBean) {  //获取<Bean>元素中的id属性值  String id = ele.getAttribute(***ID\_ATTRIBUTE***);  //获取<Bean>元素中的name属性值  String nameAttr = ele.getAttribute(***NAME\_ATTRIBUTE***);  //获取<Bean>元素中的alias属性值  List<String> aliases = **new** ArrayList<String>();  //将<Bean>元素中的所有name属性值存放到别名中  **if** (StringUtils.*hasLength*(nameAttr)) {  String[] nameArr = StringUtils.*tokenizeToStringArray*(nameAttr, ***MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS***);  aliases.addAll(Arrays.*asList*(nameArr));  }  String beanName = id;  //如果<Bean>元素中没有配置id属性时，将别名中的第一个值赋值给beanName  **if** (!StringUtils.*hasText*(beanName) && !aliases.isEmpty()) {  beanName = aliases.remove(0);  **if** (logger.isDebugEnabled()) {  logger.debug("No XML 'id' specified - using '" + beanName +  "' as bean name and " + aliases + " as aliases");  }  }  //检查<Bean>元素所配置的id或者name的唯一性，containingBean标识<Bean>  //元素中是否包含子<Bean>元素  **if** (containingBean == **null**) {  //检查<Bean>元素所配置的id、name或者别名是否重复  checkNameUniqueness(beanName, aliases, ele);  }  //详细对<Bean>元素中配置的Bean定义进行解析的地方  AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);  **if** (beanDefinition != **null**) {  **if** (!StringUtils.*hasText*(beanName)) {  **try** {  **if** (containingBean != **null**) {  //如果<Bean>元素中没有配置id、别名或者name，且没有包含子  //<Bean>元素，为解析的Bean生成一个唯一beanName并注册  beanName = BeanDefinitionReaderUtils.*generateBeanName*(  beanDefinition, **this**.readerContext.getRegistry(), **true**);  }  **else** {  //如果<Bean>元素中没有配置id、别名或者name，且包含了子  //<Bean>元素，为解析的Bean使用别名向IoC容器注册  beanName = **this**.readerContext.generateBeanName(beanDefinition);  // Register an alias for the plain bean class name, if still possible,  // if the generator returned the class name plus a suffix.  // This is expected for Spring 1.2/2.0 backwards compatibility.  //为解析的Bean使用别名注册时，为了向后兼容  //Spring1.2/2.0，给别名添加类名后缀  String beanClassName = beanDefinition.getBeanClassName();  **if** (beanClassName != **null** &&  beanName.startsWith(beanClassName) && beanName.length() > beanClassName.length() &&  !**this**.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {  aliases.add(beanClassName);  }  }  **if** (logger.isDebugEnabled()) {  logger.debug("Neither XML 'id' nor 'name' specified - " +  "using generated bean name [" + beanName + "]");  }  }  **catch** (Exception ex) {  error(ex.getMessage(), ele);  **return** **null**;  }  }  String[] aliasesArray = StringUtils.*toStringArray*(aliases);  **return** **new** BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);  }  //当解析出错时，返回null  **return** **null**;  } |

BeanDefinition可以看成是对<bean>定义的抽象，如常见的init-method, destory-method, factory-method， beanClass, description, lazyinit等等。对Bean定义的相关处理，如对元素attribute值的处理，对元素属性值的处理，对构造函数设置的处理等。对BeanDefinition元素处理代码如下：

|  |
| --- |
| /\*\*  \* Parse the bean definition itself, without regard to name or aliases. May return  \* {@code null} if problems occurred during the parsing of the bean definition.  \* //详细对<Bean>元素中配置的Bean定义其他属性进行解析，由于上面的方法中已经对  \* //Bean的id、name和别名等属性进行了处理，该方法中主要处理除这三个以外的其他属性数据  \*/  **public** AbstractBeanDefinition parseBeanDefinitionElement(  Element ele, String beanName, BeanDefinition containingBean) {  //记录解析的<Bean>  **this**.parseState.push(**new** BeanEntry(beanName));  //这里只读取<Bean>元素中配置的class名字，然后载入到BeanDefinition中去  //只是记录配置的class名字，不做实例化，对象的实例化在依赖注入时完成  String className = **null**;  **if** (ele.hasAttribute(***CLASS\_ATTRIBUTE***)) {  className = ele.getAttribute(***CLASS\_ATTRIBUTE***).trim();  }  **try** {  String parent = **null**;  //如果<Bean>元素中配置了parent属性，则获取parent属性的值  **if** (ele.hasAttribute(***PARENT\_ATTRIBUTE***)) {  parent = ele.getAttribute(***PARENT\_ATTRIBUTE***);  }  //根据<Bean>元素配置的class名称和parent属性值创建BeanDefinition  //为载入Bean定义信息做准备  AbstractBeanDefinition bd = createBeanDefinition(className, parent);  //对当前的<Bean>元素中配置的一些属性进行解析和设置，如配置的单态(singleton)属性等  parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);  //为<Bean>元素解析的Bean设置description信息  bd.setDescription(DomUtils.*getChildElementValueByTagName*(ele, ***DESCRIPTION\_ELEMENT***));  //对<Bean>元素的meta(元信息)属性解析  parseMetaElements(ele, bd);  //对<Bean>元素的lookup-method属性解析  parseLookupOverrideSubElements(ele, bd.getMethodOverrides());  //对<Bean>元素的replaced-method属性解析  parseReplacedMethodSubElements(ele, bd.getMethodOverrides());  //解析<Bean>元素的构造方法设置  parseConstructorArgElements(ele, bd);  //解析<Bean>元素的<property>设置  parsePropertyElements(ele, bd);  //解析<Bean>元素的qualifier属性  parseQualifierElements(ele, bd);  //为当前解析的Bean设置所需的资源和依赖对象  bd.setResource(**this**.readerContext.getResource());  bd.setSource(extractSource(ele));  **return** bd;  }  // 下面这些异常是在配置Bean出现问题时经常会见到的  **catch** (ClassNotFoundException ex) {  error("Bean class [" + className + "] not found", ele, ex);  }  **catch** (NoClassDefFoundError err) {  error("Class that bean class [" + className + "] depends on not found", ele, err);  }  **catch** (Throwable ex) {  error("Unexpected failure during bean definition parsing", ele, ex);  }  **finally** {  **this**.parseState.pop();  }  //解析<Bean>元素出错时，返回null  **return** **null**;  } |

上面是具体生成BeanDefinition的地方，例如对property进行解析如下：

|  |
| --- |
| /\*\*  \* Parse property sub-elements of the given bean element.  \* //解析<Bean>元素中的<property>子元素  \*/  **public** **void** parsePropertyElements(Element beanEle, BeanDefinition bd) {  //获取<Bean>元素中所有的子元素  NodeList nl = beanEle.getChildNodes();  **for** (**int** i = 0; i < nl.getLength(); i++) {  Node node = nl.item(i);  //如果子元素是<property>子元素，则调用解析<property>子元素方法解析  **if** (isCandidateElement(node) && nodeNameEquals(node, ***PROPERTY\_ELEMENT***)) {  parsePropertyElement((Element) node, bd);  }  }  }  /\*\*  \* Parse a property element.  \* //解析<property>元素  \*/  **public** **void** parsePropertyElement(Element ele, BeanDefinition bd) {  //获取<property>元素的名字  String propertyName = ele.getAttribute(***NAME\_ATTRIBUTE***);  **if** (!StringUtils.*hasLength*(propertyName)) {  error("Tag 'property' must have a 'name' attribute", ele);  **return**;  }  **this**.parseState.push(**new** PropertyEntry(propertyName));  **try** {  //如果一个Bean中已经有同名的property存在，则不进行解析，直接返回。  //即如果在同一个Bean中配置同名的property，则只有第一个起作用  **if** (bd.getPropertyValues().contains(propertyName)) {  error("Multiple 'property' definitions for property '" + propertyName + "'", ele);  **return**;  }  //解析获取property的值  Object val = parsePropertyValue(ele, bd, propertyName);  //根据property的名字和值创建property实例  PropertyValue pv = **new** PropertyValue(propertyName, val);  //解析<property>元素中的属性  parseMetaElements(ele, pv);  pv.setSource(extractSource(ele));  bd.getPropertyValues().addPropertyValue(pv);  }  **finally** {  **this**.parseState.pop();  }  }  /\*\*  \* Get the value of a property element. May be a list etc.  \* Also used for constructor arguments, "propertyName" being null in this case.  \* //解析获取property值  \*/  **public** Object parsePropertyValue(Element ele, BeanDefinition bd, String propertyName) {  String elementName = (propertyName != **null**) ?  "<property> element for property '" + propertyName + "'" :  "<constructor-arg> element";  // Should only have one child element: ref, value, list, etc.  //获取<property>的所有子元素，只能是其中一种类型:ref,value,list等  NodeList nl = ele.getChildNodes();  Element subElement = **null**;  **for** (**int** i = 0; i < nl.getLength(); i++) {  Node node = nl.item(i);  //子元素不是description和meta属性  **if** (node **instanceof** Element && !nodeNameEquals(node, ***DESCRIPTION\_ELEMENT***) &&  !nodeNameEquals(node, ***META\_ELEMENT***)) {  // Child element is what we're looking for.  **if** (subElement != **null**) {  error(elementName + " must not contain more than one sub-element", ele);  }  **else** { ////当前<property>元素包含有子元素  subElement = (Element) node;  }  }  }  //判断property的属性值是ref还是value，不允许既是ref又是value  **boolean** hasRefAttribute = ele.hasAttribute(***REF\_ATTRIBUTE***);  **boolean** hasValueAttribute = ele.hasAttribute(***VALUE\_ATTRIBUTE***);  **if** ((hasRefAttribute && hasValueAttribute) ||  ((hasRefAttribute || hasValueAttribute) && subElement != **null**)) {  error(elementName +  " is only allowed to contain either 'ref' attribute OR 'value' attribute OR sub-element", ele);  }  //如果属性是ref，创建一个ref的数据对象RuntimeBeanReference，这个对象封装了ref信息  **if** (hasRefAttribute) {  String refName = ele.getAttribute(***REF\_ATTRIBUTE***);  **if** (!StringUtils.*hasText*(refName)) {  error(elementName + " contains empty 'ref' attribute", ele);  }  //一个指向运行时所依赖对象的引用  RuntimeBeanReference ref = **new** RuntimeBeanReference(refName);  //设置这个ref的数据对象是被当前的property对象所引用  ref.setSource(extractSource(ele));  **return** ref;  }  //如果属性是value，创建一个value的数据对象TypedStringValue，这个对象封装了value信息  **else** **if** (hasValueAttribute) {  //一个持有String类型值的对象  TypedStringValue valueHolder = **new** TypedStringValue(ele.getAttribute(***VALUE\_ATTRIBUTE***));  //设置这个value数据对象是被当前的property对象所引用  valueHolder.setSource(extractSource(ele));  **return** valueHolder;  }  //如果当前<property>元素还有子元素  **else** **if** (subElement != **null**) {  //解析<property>的子元素  **return** parsePropertySubElement(subElement, bd);  }  **else** {  // Neither child element nor "ref" or "value" attribute found.  //propery属性中既不是ref，也不是value属性，解析出错返回null  error(elementName + " must specify a ref or value", ele);  **return** **null**;  }  } |

对property子元素的解析过程，Array、List、Set、Map、Prop等各种元素都会在这里进行解析，生成对应的数据对象，如ManagedList、ManagedSet、ManagedArray等，这些Managed类是Spring对具体的BeanDefinition的数据封装。下面以Property的元素解析为例来说明：

|  |
| --- |
| /\*\*  \* Parse a value, ref or collection sub-element of a property or  \* constructor-arg element.  \* **@param** ele subelement of property element; we don't know which yet  \* **@param** defaultValueType the default type (class name) for any  \* {@code &lt;value&gt;} tag that might be created  \* //解析<property>元素中ref,value或者集合等子元素  \*/  **public** Object parsePropertySubElement(Element ele, BeanDefinition bd, String defaultValueType) {  //如果<property>没有使用Spring默认的命名空间，则使用用户自定义的规则解析内嵌元素  **if** (!isDefaultNamespace(ele)) {  **return** parseNestedCustomElement(ele, bd);  }  //如果子元素是bean，则使用解析<Bean>元素的方法解析  **else** **if** (nodeNameEquals(ele, ***BEAN\_ELEMENT***)) {  BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);  **if** (nestedBd != **null**) {  nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);  }  **return** nestedBd;  }  //如果子元素是ref，ref中只能有以下3个属性：bean、local、parent  **else** **if** (nodeNameEquals(ele, ***REF\_ELEMENT***)) {  // A generic reference to any name of any bean.  //获取<property>元素中的bean属性值，引用其他解析的Bean的名称  //可以不再同一个Spring配置文件中，具体请参考Spring对ref的配置规则  String refName = ele.getAttribute(***BEAN\_REF\_ATTRIBUTE***);  **boolean** toParent = **false**;  **if** (!StringUtils.*hasLength*(refName)) {  //获取<property>元素中的local属性值，引用同一个Xml文件中配置  //的Bean的id，local和ref不同，local只能引用同一个配置文件中的Bean  // A reference to the id of another bean in the same XML file.  refName = ele.getAttribute(***LOCAL\_REF\_ATTRIBUTE***);  **if** (!StringUtils.*hasLength*(refName)) {  // A reference to the id of another bean in a parent context.  //获取<property>元素中parent属性值，引用父级容器中的Bean  refName = ele.getAttribute(***PARENT\_REF\_ATTRIBUTE***);  toParent = **true**;  **if** (!StringUtils.*hasLength*(refName)) {  error("'bean', 'local' or 'parent' is required for <ref> element", ele);  **return** **null**;  }  }  }  //没有配置ref的目标属性值  **if** (!StringUtils.*hasText*(refName)) {  error("<ref> element contains empty target attribute", ele);  **return** **null**;  }  //创建ref类型数据，指向被引用的对象  RuntimeBeanReference ref = **new** RuntimeBeanReference(refName, toParent);  //设置引用类型值是被当前子元素所引用  ref.setSource(extractSource(ele));  **return** ref;  }  //如果子元素是<idref>，使用解析ref元素的方法解析  **else** **if** (nodeNameEquals(ele, ***IDREF\_ELEMENT***)) {  **return** parseIdRefElement(ele);  }  //如果子元素是<value>，使用解析value元素的方法解析  **else** **if** (nodeNameEquals(ele, ***VALUE\_ELEMENT***)) {  **return** parseValueElement(ele, defaultValueType);  }  //如果子元素是null，为<property>设置一个封装null值的字符串数据  **else** **if** (nodeNameEquals(ele, ***NULL\_ELEMENT***)) {  // It's a distinguished null value. Let's wrap it in a TypedStringValue  // object in order to preserve the source location.  TypedStringValue nullHolder = **new** TypedStringValue(**null**);  nullHolder.setSource(extractSource(ele));  **return** nullHolder;  }  //如果子元素是<array>，使用解析array集合子元素的方法解析  **else** **if** (nodeNameEquals(ele, ***ARRAY\_ELEMENT***)) {  **return** parseArrayElement(ele, bd);  }  //如果子元素是<list>，使用解析list集合子元素的方法解析  **else** **if** (nodeNameEquals(ele, ***LIST\_ELEMENT***)) {  **return** parseListElement(ele, bd);  }  //如果子元素是<set>，使用解析set集合子元素的方法解析  **else** **if** (nodeNameEquals(ele, ***SET\_ELEMENT***)) {  **return** parseSetElement(ele, bd);  }  //如果子元素是<map>，使用解析map集合子元素的方法解析  **else** **if** (nodeNameEquals(ele, ***MAP\_ELEMENT***)) {  **return** parseMapElement(ele, bd);  }  //如果子元素是<props>，使用解析props集合子元素的方法解析  **else** **if** (nodeNameEquals(ele, ***PROPS\_ELEMENT***)) {  **return** parsePropsElement(ele);  }  //既不是ref，又不是value，也不是集合，则子元素配置错误，返回null  **else** {  error("Unknown property sub-element: [" + ele.getNodeName() + "]", ele);  **return** **null**;  }  } |

再看看List这样的属性配置是怎样被解析的，依然是在BeanDefinitionParserDelegate中，返回的是一个List对象，即ManagedList，看代码：

|  |
| --- |
| /\*\*  \* Parse a list element.  \* //解析<list>集合子元素  \*/  **public** List parseListElement(Element collectionEle, BeanDefinition bd) {  //获取<list>元素中的value-type属性，即获取集合元素的数据类型  String defaultElementType = collectionEle.getAttribute(***VALUE\_TYPE\_ATTRIBUTE***);  //获取<list>集合元素中的所有子节点  NodeList nl = collectionEle.getChildNodes();  //Spring中将List封装为ManagedList  ManagedList<Object> target = **new** ManagedList<Object>(nl.getLength());  //设置集合目标数据类型  target.setSource(extractSource(collectionEle));  target.setElementTypeName(defaultElementType);  target.setMergeEnabled(parseMergeAttribute(collectionEle));  //具体的<list>元素解析  parseCollectionElements(nl, target, bd, defaultElementType);  **return** target;  }  //具体解析<list>集合元素，<array>、<list>和<set>都使用该方法解析  **protected** **void** parseCollectionElements(  NodeList elementNodes, Collection<Object> target, BeanDefinition bd, String defaultElementType) {  //遍历集合所有节点  **for** (**int** i = 0; i < elementNodes.getLength(); i++) {  Node node = elementNodes.item(i);  **if** (node **instanceof** Element && !nodeNameEquals(node, ***DESCRIPTION\_ELEMENT***)) {  //将解析的元素加入集合中，递归调用下一个子元素  target.add(parsePropertySubElement((Element) node, bd, defaultElementType));  }  }  } |

经过这样逐层解析，在XML文件中定义的BeanDefinition就被整个载入到了IOC容器中，并在容器中建立了数据映射。这些数据结构可以以AbstractBeanDefinition为入口，让IOC容器进行索引，查询等操作。以上大致完成了管理Bean对象的数据准备工作，但是重要的依赖注入还没有发生，现在IOC容器BeanDefinition中存在的还只是一些静态的配置信息，要想容器发挥作用，还需要完成数据向容器的注册。

### 4.11 BeanDefinition在IOC容器中的注册

在DefaultListableBeanFactory中是通过一个HashMap来持有载入的BeanDefinition的，这个HashMap的定义在DefaultListableBeanFactory中的定义：

|  |
| --- |
| //存储注册的BeanDefinition  **private** **final** Map<String, BeanDefinition> beanDefinitionMap  = **new** ConcurrentHashMap<String, BeanDefinition>(64); |

注册的调用过程

|  |
| --- |
| DefaultBeanDefinitionDocumentReader  DefaultListableBeanFactory  XmlBeanDefinitionReader  processBeanDefinition()  registerBeanDefinition()    beanDefinitionMap |

在DefaultListableBeanFactory中实现了BeanDefinitionRegistry的接口，这个接口的实现完成了BeanDefinition向容器的注册，即把解析得到的BeanDefinition设置到hashMap中去。需要注意的是，如果遇到同名的BeanDefinition，进行处理的进候要依据allowBeanDefinitionOverriding的配置来完成，具体代码如下：

|  |
| --- |
| //向IoC容器注册解析的BeanDefiniton  **public** **void** registerBeanDefinition(String beanName, BeanDefinition beanDefinition)  **throws** BeanDefinitionStoreException {  Assert.*hasText*(beanName, "Bean name must not be empty");  Assert.*notNull*(beanDefinition, "BeanDefinition must not be null");  //校验解析的BeanDefiniton  **if** (beanDefinition **instanceof** AbstractBeanDefinition) {  **try** {  ((AbstractBeanDefinition) beanDefinition).validate();  }  **catch** (BeanDefinitionValidationException ex) {  **throw** **new** BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,  "Validation of bean definition failed", ex);  }  }  BeanDefinition oldBeanDefinition;  //注册的过程中需要线程同步，以保证数据的一致性  **synchronized** (**this**.beanDefinitionMap) {  oldBeanDefinition = **this**.beanDefinitionMap.get(beanName);  //检查是否有同名的BeanDefinition已经在IoC容器中注册，如果已经注册，  //并且不允许覆盖已注册的Bean，则抛出注册失败异常  **if** (oldBeanDefinition != **null**) {  **if** (!**this**.allowBeanDefinitionOverriding) {  **throw** **new** BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,  "Cannot register bean definition [" + beanDefinition + "] for bean '" + beanName +  "': There is already [" + oldBeanDefinition + "] bound.");  }  **else** {//如果允许覆盖，则同名的Bean，后注册的覆盖先注册的  **if** (**this**.logger.isInfoEnabled()) {  **this**.logger.info("Overriding bean definition for bean '" + beanName +  "': replacing [" + oldBeanDefinition + "] with [" + beanDefinition + "]");  }  }  }  //IoC容器中没有已经注册同名的Bean，按正常注册流程注册，把Bean的名字存入到beanDefinitionNames的同时，把beanName作为Map的key,把beanDefinition作为value存入到IOC容器持有的beanDefinitionMap中去。  **else** {  **this**.beanDefinitionNames.add(beanName);  **this**.frozenBeanDefinitionNames = **null**;  }  **this**.beanDefinitionMap.put(beanName, beanDefinition);  }  **if** (oldBeanDefinition != **null** || containsSingleton(beanName)) {  //重置所有已经注册过的BeanDefinition的缓存  resetBeanDefinition(beanName);  }  } |

完成了BeanDefinition的注册，就完成了IOC容器的初始化过程。BeanDefinition都被存在benaDefinitionMap里被检索和使用，容器的作用就是对这些信息进行处理和维护，这些信息是容器建立依赖反转的基础。