# 一 Spring IoC原理

## 1 什么是Ioc/DI？

IoC的思想：Spring容器来实现相互依赖对象的创建、协调工作。**依赖对象的获取被反转了**。IoC容器：最主要是完成了对象的创建和依赖的管理注入等等。

先从我们自己设计这样一个视角来考虑：所谓控制反转，就是把原先我们代码里面需要实现的对象创建、依赖的代码，反转给容器来帮忙实现。那么必然的我们需要创建一个容器，同时需要一种描述来让容器知道需要创建的对象与对象的关系。这个描述最具体表现就是我们可配置的文件。

对象和对象关系怎么表示？可以用 xml、properties 文件等语义化配置文件表示。描述对象关系的文件存放在哪里？可能是 classpath、filesystem、或者是 URL 网络资源， servletContext 等。

有了配置文件，还需要对配置文件解析。不同的配置文件对对象的描述不一样，如标准的，自定义声明式的，如何统一？ 在内部需要有一个统一的关于对象的定义，所有外部的描述都必须转化成统一的描述定义。如何对不同的配置文件进行解析？需要对不同的配置文件语法，采用不同的解析器。

## [2 Spring IoC体系结构](https://www.cnblogs.com/ITtangtang/p/3978349.html#a2)

### 2.1 BeanFactory

Spring Bean的创建是典型的**工厂模式**，这一系列的Bean工厂，也即IoC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IoC容器的实现，其相互关系如下：



其中**BeanFactory作为最顶层的一个接口类，它定义了IoC容器的基本功能规范**，BeanFactory 有三个子类：ListableBeanFactory、HierarchicalBeanFactory 和AutowireCapableBeanFactory。但是从上图中我们可以发现最终的默认实现类是 DefaultListableBeanFactory，他实现了所有的接口。那为何要定义这么多层次的接口呢？查阅这些接口的源码和说明发现，每个接口都有他使用的场合，它主要是为了区分在 Spring 内部在操作过程中对象的传递和转化过程中，对对象的数据访问所做的限制。例如 ListableBeanFactory 接口表示这些 Bean 是可列表的，而 HierarchicalBeanFactory 表示的是这些 Bean 是有继承关系的，也就是每个Bean 有可能有父 Bean。AutowireCapableBeanFactory 接口定义 Bean 的自动装配规则。这四个接口共同定义了 Bean 的集合、Bean 之间的关系、以及 Bean 行为。

|  |
| --- |
| 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。  Object getBean(String name, Class requiredType) 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  boolean isSingleton(String name) throws NoSuchBeanDefinitionException;  //得到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);  } |

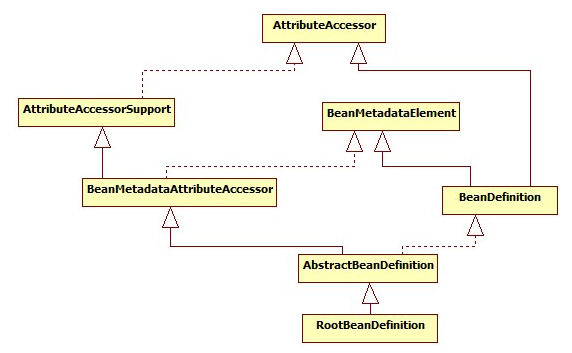
在BeanFactory里只对IoC容器的基本行为作了定义，根本不关心你的bean是如何定义怎样加载的。正如我们只关心工厂里得到什么的产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心。BeanFactory接口定义了IoC容器最基本的形式。在使用容器时，可以使用转义符“&”来得到FactoryBean本身，用来区分通过容器来获取FactoryBean产生的对象和获取FactoryBean本身。在Spring中，所有的Bean都是由BeanFactory来进行管理的，是一个能产生或者修饰对象生成的工厂Bean。

而要知道工厂是如何产生对象的，我们需要看具体的IOC容器实现，Spring提供了许多IoC容器的实现。比如XmlBeanFactory，ClasspathXmlApplicationContext等。其中**XmlBeanFactory**就是针对最基本的IoC容器的实现，构造XmlBeanFactory时，需要指定BeanDefinition的信息来源，这个信息来源需要封装成Spring中的Resource类来给出。

如果说XmlBeanFactory是容器中的屌丝，ApplicationContext应该算容器中的高帅富。**ApplicationContext**是Spring提供的一个高级的IoC容器，它除了能够提供IoC容器的基本功能外，还为用户提供了以下的附加服务。从ApplicationContext接口的实现，我们看出其特点： 1.支持信息源，可以实现国际化。（实现MessageSource接口）；2.访问资源。(实现ResourcePatternResolver接口，这个后面要讲)；3.支持应用事件。(实现ApplicationEventPublisher接口)。FileSystemXmlApplicationContext（作为ApplicationContext的实现）。

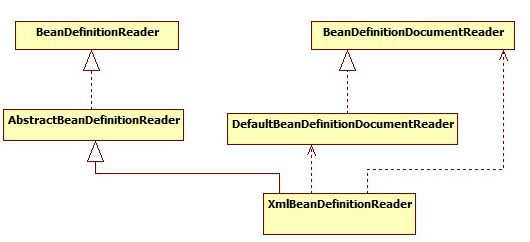
### 2.2 BeanDefinition

Bean对象在Spring实现中是以BeanDefinition来描述的，其继承体系如下。



SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Spring通过定义BeanDefinition来管理基于Spring的应用中的各种对象以及他们之间的相互依赖关系。对于IoC容器来说，BeanDefinition就是对依赖反转模式中管理的对象依赖关系的数据抽象，也是容器实现依赖敢转功能的核心数据结构，依赖反转功能都是围绕对这个BeanDefinition的处理来完成的。

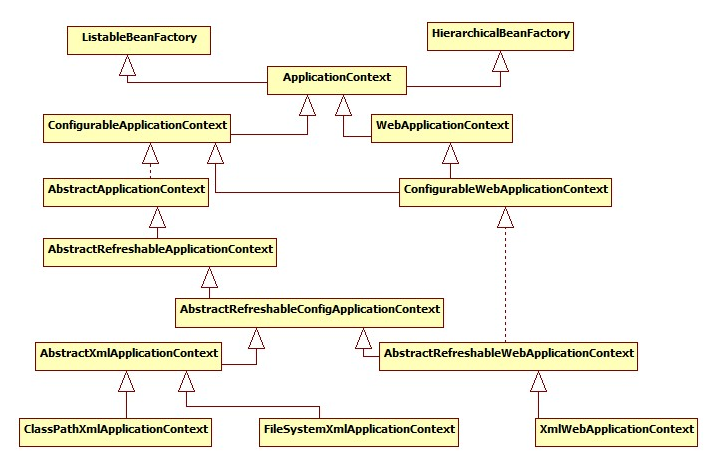
Bean 的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean 的解析主要就是对 Spring 配置文件的解析。这个解析过程主要通过下图中的类完成：



## [3 IoC容器的初始化](https://www.cnblogs.com/ITtangtang/p/3978349.html#a3)

IoC容器的初始化是由refresh()方法来启动的，这个方法标志着IoC容器的正式启动，具体包括**BeanDefinition的Resource定位、载入和注册这三个基本的过程**。第一个过程是Resource定位过程，这个Resource定位指的是BeanDefinition的资源定位，它由ResourceLoader通过统一的Resource接口来完成；第二个过程是BeanDefinition的载入，这个载入过程是把用户定义好的Bean表示成IoC容器内部的数据结构；第三个过程是想IoC容器注册这些BeanDefinition的过程，这个过程是通过调用BeanDefinitionRegistry接口的实现来完成（将BeanDefinition注入到一个HashMap中）。

我们以ApplicationContext为例讲解，ApplicationContext系列容器也许是我们最熟悉的，因为web项目中使用的XmlWebApplicationContext就属于这个继承体系，还有ClasspathXmlApplicationContext等，其继承体系如下图所示：



ApplicationContext允许上下文嵌套，通过保持父上下文可以维持一个上下文体系。对于bean的查找可以在这个上下文体系中发生，首先检查当前上下文，其次是父上下文，逐级向上，这样为不同的Spring应用提供了一个共享的bean定义环境。

### 3.1编程式使用IoC容器

|  |
| --- |
| //根据Xml配置文件创建Resource资源对象，该对象中包含了BeanDefinition的信息  ClassPathResource resource =new ClassPathResource("application-context.xml");  //创建DefaultListableBeanFactory  DefaultListableBeanFactory factory =new DefaultListableBeanFactory();  //创建XmlBeanDefinitionReader读取器，用于载入BeanDefinition。之所以需要BeanFactory作为参数，是因  //为会将读取的信息回调配置给factory  XmlBeanDefinitionReader reader =new XmlBeanDefinitionReader(factory);  //XmlBeanDefinitionReader执行载入BeanDefinition的方法，最后会完成Bean的载入和注册。完成后Bean就  //成功的放置到IOC容器当中，以后我们就可以从中取得Bean来使用  reader.loadBeanDefinitions(resource); |

使用IoC容器时，需要如下几个步骤：

1) 创建IoC配置文件的抽象资源，这个抽象资源包含了BeanDefinition的定义信息;

2) 创建一个BeanFactory，这里使用DefaultListableBeanFactory;

3) 创建一个载入BeanDefinition的读取器，这里使用XmlBeanDefinitionReader来载入XML文件形式的BeanDefinition，通过一个回调配置给BeanFactory;

4)从定义好的资源位置读入配置信息，具体的解析过程由XmlBeanDefinitionReader来完成，完成整个载入和注册Bean定义之后，需要的IoC容器就建立起来了。

### 3.2 BeanDefinition的Resource定位

使用编程式时，定义的Resource不能由DefaultListableBeanFactory直接使用，Spring使用BeanDefinitionReader来对这些信息进行处理。而ApplicationContext中，已经提供了一系列加载不同Resource的读取器的实现。比如FileSystemXMLApplicationContext可以从文件系统载入Resource，ClassPathXMLApplicationContext可以从Class Path载入Resource，XMLWebApplicationContext可以在Web容器中载入Resource。

下面以FileSystemXmlApplicationContext为例，分析Resource定位过程。

FileSystemXmlApplicationContext通过继承AbstractApplicationContext具备了ResourceLoader读入以Resource定义的BeanDefinition的能力。

#### 3.2.1 FileSystemXmlApplicationContext构造函数

|  |
| --- |
| public **FileSystemXmlApplicationContext**(String... configLocations) throws BeansException {  this(configLocations, true, null);  } |

实际调用：

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

#### 3.2.2 设置资源加载器和资源定位

在创建FileSystemXmlApplicationContext容器时，构造方法做以下两项重要工作：首先，调用父类容器的构造方法(super(parent)方法)为容器设置好Bean资源加载器。然后，再调用父类**AbstractRefreshableConfigApplicationContext**的setConfigLocations(configLocations)方法设置Bean定义资源文件的定位路径。通过追踪FileSystemXmlApplicationContext的继承体系，发现其父类的父类AbstractApplicationContext中初始化IoC容器所做的主要源码如下：

|  |
| --- |
| public abstract class AbstractApplicationContext extends DefaultResourceLoader  implements ConfigurableApplicationContext, DisposableBean {  //静态初始化块，在整个容器创建过程中只执行一次  static {  //为了避免应用程序在Weblogic8.1关闭时出现类加载异常加载问题，加载IoC容器关闭事件类  ContextClosedEvent.class.getName();  }  //FileSystemXmlApplicationContext调用父类构造方法调用的就是该方法  public AbstractApplicationContext(ApplicationContext parent) {  this.parent = parent;  this.resourcePatternResolver = getResourcePatternResolver();  }  //获取一个Spring Source的加载器用于读入Spring Bean定义资源文件  protected ResourcePatternResolver getResourcePatternResolver() {  // AbstractApplicationContext继承DefaultResourceLoader，也是一个S  //Spring资源加载器，其getResource(String location)方法用于载入资源  return new PathMatchingResourcePatternResolver(this);  }  ……  } |

AbstractApplicationContext构造方法中调用PathMatchingResourcePatternResolver的构造方法创建Spring资源加载器。在设置容器的资源加载器之后，接下来执行setConfigLocations方法通过调用其父类AbstractRefreshableConfigApplicationContext的方法进行对Bean定义资源文件的定位，该方法的源码如下：

|  |
| --- |
| //处理单个资源文件路径为一个字符串的情况  public void setConfigLocation(String location) {  // CONFIG\_LOCATION\_DELIMITERS = ",; /t/n"; ，即多个资源文件路径之间用” ,; /t/n”分隔  setConfigLocations(StringUtils.tokenizeToStringArray(location, CONFIG\_LOCATION\_DELIMITERS));  }  //解析Bean定义资源文件的路径，处理多个资源文件字符串数组  public void setConfigLocations(String[] locations) {  if (locations != null) {  Assert.noNullElements(locations, "Config locations must not be null");  this.configLocations = new String[locations.length];  for (int i = 0; i < locations.length; i++) {  // resolvePath为同一个类中将字符串解析为路径的方法  this.configLocations[i] = resolvePath(locations[i]).trim();  }  }  else {  this.configLocations = null;  }  } |

通过这两个方法的源码我们可以看出，我们既可以使用一个字符串来配置多个Spring Bean定义资源文件，也可以使用字符串数组，多个资源文件路径之间可以是用” ,; /t/n”等分隔。至此，Spring IoC容器在初始化时将配置的Bean定义资源文件定位为Spring封装的Resource。

#### 3.2.3 AbstractApplicationContext的refresh函数载入Bean

**Spring IoC容器对Bean定义资源的载入是从refresh()函数开始的**，refresh()是一个模板方法，refresh()方法的作用是：**在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器**。refresh的作用类似于对IoC容器的重启，在新建立好的容器中对容器进行初始化，对Bean定义资源进行载入。

FileSystemXmlApplicationContext通过调用其父类***AbstractApplicationContext***的refresh()函数启动整个IoC容器对Bean定义的载入过程：

|  |
| --- |
| public void **refresh**() throws BeansException, IllegalStateException {  synchronized (this.startupShutdownMonitor) {  ... ...  //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();**  ... ...  }  } |

refresh()方法主要为IoC容器Bean的生命周期管理提供条件，Spring IoC容器载入Bean定义资源文件从其子类容器的refreshBeanFactory()方法启动，所以整个refresh()中“ConfigurableListableBeanFactory beanFactory =obtainFreshBeanFactory();”这句以后代码的都是注册容器的信息源和生命周期事件，载入过程就是从这句代码启动。refresh()方法的作用是：在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器。refresh的作用类似于对IoC容器的重启，在新建立好的容器中对容器进行初始化，对Bean定义资源进行载入.

AbstractApplicationContext的obtainFreshBeanFactory()方法调用子类容器的refreshBeanFactory()方法，启动容器载入Bean定义资源文件的过程，代码如下：

|  |
| --- |
| protected ConfigurableListableBeanFactory **obtainFreshBeanFactory**() {  //这里使用了委派设计模式，父类定义了抽象的refreshBeanFactory()方法，具体实现调用子类容器的refreshBeanFactory()方法  **refreshBeanFactory();**  ConfigurableListableBeanFactory beanFactory = getBeanFactory();  if (logger.isDebugEnabled()) {  logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);  }  return beanFactory;  } |

AbstractApplicationContext类中只抽象定义了refreshBeanFactory()方法，容器真正调用的是其子类AbstractRefreshableApplicationContext实现的refreshBeanFactory()方法，方法的源码如下：

|  |
| --- |
| protected final void refreshBeanFactory() throws BeansException {  if (hasBeanFactory()) {//如果已经有BeanFactory，销毁并关闭该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);  //调用载入Bean定义的方法，主要这里又使用了一个委派模式，具体的实现调用子类容器  **loadBeanDefinitions(beanFactory);**  synchronized (this.beanFactoryMonitor) {  this.beanFactory = beanFactory;  }  }  ... ...  } |

在这个方法中，先判断BeanFactory是否存在，如果存在则先销毁beans并关闭beanFactory，接着创建DefaultListableBeanFactory，并调用loadBeanDefinitions(beanFactory)装载bean定义。

#### 3.2.4 AbstractRefreshableApplicationContext的createBeanFactory创建IoC容器

|  |
| --- |
| protected DefaultListableBeanFactory createBeanFactory() {  return new **DefaultListableBeanFactory**(getInternalParentBeanFactory());  } |

#### 3.2.5 AbstractRefreshableApplicationContext子类的loadBeanDefinitions方法

AbstractRefreshableApplicationContext中只定义了**抽象**的loadBeanDefinitions方法，容器真正调用的是其子类AbstractXmlApplicationContext对该方法的实现，AbstractXmlApplicationContext的主要源码如下：

|  |
| --- |
| public abstract class AbstractXmlApplicationContext extends AbstractRefreshableConfigApplicationContext {  ……  @Override //实现父类抽象的载入Bean定义方法  protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {  //创建XmlBeanDefinitionReader，即创建Bean读取器，并通过回调设置到容器中去  XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);  //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.setResourceLoader(this);  beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this)); //为Bean读取器设置SAX xml解析器  initBeanDefinitionReader(beanDefinitionReader); //当读取Bean定义的Xml资源文件时，启用Xml的校验机制  **loadBeanDefinitions(beanDefinitionReader); //Bean读取器真正实现加载的方法**  }  //Xml Bean读取器加载Bean定义资源  protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {  Resource[] configResources = getConfigResources(); //获取Bean定义资源的定位  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);**  }  //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  String[] configLocations = getConfigLocations();  if (configLocations != null) {  //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的Bean定义资源  **reader.loadBeanDefinitions(configLocations);**  }  }  //这里又使用了一个委托模式，调用子类的获取Bean定义资源定位的方法  //该方法在ClassPathXmlApplicationContext中进行实现，对于我们  //举例分析源码的FileSystemXmlApplicationContext没有使用该方法  protected Resource[] getConfigResources() {  return null;  } ……  } |

Xml Bean读取器(XmlBeanDefinitionReader)调用其父类AbstractBeanDefinitionReader的 reader.loadBeanDefinitions方法读取Bean定义资源。

由于我们使用FileSystemXmlApplicationContext作为例子分析，因此getConfigResources的返回值为null，因此程序执行reader.loadBeanDefinitions(configLocations)分支,实际上调用XmlBeanDefinitionReader的抽象父类AbstractBeanDefinitionReader的loadBeanDefinitions方法。

#### 3.2.6 AbstractBeanDefinitionReader读取Bean定义资源

|  |
| --- |
| //重载方法，调用下面的loadBeanDefinitions(String, Set<Resource>);方法  public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException {  **return loadBeanDefinitions(location, null);**  }  public int loadBeanDefinitions(String location, Set<Resource> actualResources) throws BeanDefinitionStoreException {  //这里取得ResourceLoader，使用的是DefaultResourceLoader  ResourceLoader resourceLoader = getResourceLoader();  if (resourceLoader == null) {  throw new BeanDefinitionStoreException(… …);  }  //这里对Resource的路径进行解析，得到需要的Resource集合，指向定义好的BeanDefinition信息  if (resourceLoader instanceof ResourcePatternResolver) {  try {  **//调用DefaultResourceLoader的getResource完成具体的Resource定位**  **Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);**  **//委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能**  **int loadCount = loadBeanDefinitions(resources)**;  if (actualResources != null) {  for (Resource resource : resources) {  actualResources.add(resource);  }  }  if (logger.isDebugEnabled()) {  logger.debug("Loaded " + loadCount + " bean definitions from location pattern [" + location + "]");  }  return loadCount;  }  catch (IOException ex) {  throw new BeanDefinitionStoreException(  "Could not resolve bean definition resource pattern [" + location + "]", ex);  }  }  else {  //将指定位置的Bean定义资源文件解析为Spring IoC容器封装的资源  //加载单个指定位置的Bean定义资源文件  **Resource resource = resourceLoader.getResource(location);**  //委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能  int loadCount = loadBeanDefinitions(resource);  if (actualResources != null) {  actualResources.add(resource);  }  if (logger.isDebugEnabled()) {  logger.debug("Loaded " + loadCount + " bean definitions from location [" + location + "]");  }  return loadCount;  }  } |

loadBeanDefinitions(Resource...resources)方法和上面分析的3个方法类似，同样也是调用XmlBeanDefinitionReader的loadBeanDefinitions方法。

从对AbstractBeanDefinitionReader的loadBeanDefinitions方法源码分析可以看出该方法做了以下两件事：

首先，调用资源加载器的获取资源方法resourceLoader.getResource(location)，获取到要加载的资源。

其次，真正执行加载功能是其子类XmlBeanDefinitionReader的loadBeanDefinitions方法。

此时调用的是DefaultResourceLoader中的getSource()方法定位Resource，因为FileSystemXmlApplicationContext本身就是DefaultResourceLoader的实现类，所以此时又回到了FileSystemXmlApplicationContext中来。

#### 3.2.7 资源加载器获取要读入的资源

XmlBeanDefinitionReader通过调用其父类**DefaultResourceLoader**的getResource方法获取要加载的资源，其源码如下：

|  |
| --- |
| public Resource getResource(String location) {  Assert.notNull(location, "Location must not be null");  //这里处理带有classpath标识的Resource  if (location.startsWith(CLASSPATH\_URL\_PREFIX)) {  return new ClassPathResource(location.substring(CLASSPATH\_URL\_PREFIX.length()), getClassLoader());  }  try {  // 这里处理URL标识的Resource定位  URL url = new URL(location);  return new UrlResource(url);  }  catch (MalformedURLException ex) {  }  //如果既不是classpath，也不是URL标识的Resource定位，则调用容器本身的getResourceByPath方法  //这是一个protected方法，默认实现是得到一个C  return getResourceByPath(location);  } |

FileSystemXmlApplicationContext容器提供了getResourceByPath方法的实现，就是为了处理既不是classpath标识，又不是URL标识的Resource定位这种情况。

|  |
| --- |
| protected Resource getResourceByPath(String path) {  if (path != null && path.startsWith("/")) {  path = path.substring(1);  }  return new FileSystemResource(path); //这里使用文件系统资源对象来定义bean 文件  } |

这样代码就回到了 FileSystemXmlApplicationContext 中来，他提供了FileSystemResource 来完成从文件系统得到配置文件的资源定义。这样，就可以从文件系统路径上对IOC 配置文件进行加载 - 当然我们可以按照这个逻辑从任何地方加载，在Spring 中我们看到它提供的各种资源抽象，比如ClassPathResource, URLResource,FileSystemResource 等来供我们使用。上面我们看到的是定位Resource 的一个过程，而这只是加载过程的一部分。

### 3.3 BeanDefinition的载入和解析

在BeanDefinition定位完成的基础上，就可以通过返回的Resource对象来进行BeanDefinition的载入了。这个载入过程，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。

IoC容器初始化的入口： FileSystemXmlApplicationContext的构造函数调用AbstractApplicationContext. **refresh()**方法实现BeanFactory的更新，MessageSource和PostProcessor的注册；refresh()方法调用基类AbstractApplicationContext.**obtainFreshBeanFactory**()方法；obtainFreshBeanFactory()调用AbstractApplicationContext. **refreshBeanFactory**()方法，这是一个抽象方法，实际调用子类AbstractRefreshableApplicationContext.refreshBeanFactory()方法； refreshBeanFactory调用**createBeanFactory**()创建IoC容器，使用的是DefaultListableBeanFactory，如果已经存在，将已有的容器销毁和关闭，保证refresh之后使用的是新建立起来的IoC容器；refreshBeanFactory调用**loadBeanDefinitions**()启动对BeanDefinition的载入，这是一个抽象方法，实际调用的是AbstractXmlApplicationContext.loadBeanDefinitions()，创建XmlBeanDefinitionReader，并通过回调设置到BeanFactory中去；loadBeanDefinitions()调用loadBeanDefinitions(**XmlBeanDefinitionReader** reader)，调用reader.loadBeanDefinitions(Resource... resources)。

实际使用的IOC容器时DefaultListableBeanFactory，具体的Resource载入在XMLBeanDefinitionReader读入BeanDefinition实现。其他的BeanDefinition由其他的BeanDefinitionReader完成载入。

#### 3.3.1 XmlBeanDefinitionReader加载Bean定义资源

继续回到XmlBeanDefinitionReader的loadBeanDefinitions(Resource …)方法看到代表bean文件的资源定义以后的载入过程。通过源码分析，载入Bean定义资源文件的最后一步是将Bean定义资源转换为Document对象，该过程由documentLoader实现。

|  |
| --- |
| //XmlBeanDefinitionReader加载资源的入口方法  public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {  return loadBeanDefinitions(new EncodedResource(resource)); //将读入的XML资源进行特殊编码处理  }  //这里是载入XML形式BeanDefinition的地方  public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {  .......  try {  InputStream inputStream = encodedResource.getResource().getInputStream(); //将资源文件转为InputStream的IO流  try {  InputSource inputSource = new InputSource(inputStream); //从InputStream中得到XML的解析源  if (encodedResource.getEncoding() != null) {  inputSource.setEncoding(encodedResource.getEncoding());  }  return doLoadBeanDefinitions(inputSource, encodedResource.getResource()); //这里是具体的读取程  } finally {  inputStream.close(); //关闭从Resource中得到的IO流  }  }  .........  }  //从特定XML文件中实际载入Bean定义资源的方法  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, this.entityResolver, this.errorHandler, validationMode, this.namespaceAware);  **//这里是启动对BeanDefinition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);**  }  .......  } |

#### 3.3.2 DocumentLoader将Bean定义资源转换为Document对象

该解析过程调用JavaEE标准的JAXP标准进行处理。

#### 3.3.3 XmlBeanDefinitionReader解析载入的Bean定义资源文件

XmlBeanDefinitionReader类的doLoadBeanDefinitions方法是从特定XML文件中实际载入BeanDefinition资源的方法，该方法在载入Bean定义资源之后将其转换为Document对象，接下来调用registerBeanDefinitions启动Spring IoC容器对Bean定义的解析过程，registerBeanDefinitions方法源码如下：

|  |
| --- |
| //按照Spring的Bean语义要求将Bean定义资源解析并转换为容器内部数据结构  public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {  //得到BeanDefinitionDocumentReader来对xml格式的BeanDefinition解析  BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  int countBefore = getRegistry().getBeanDefinitionCount(); //获得容器中注册的Bean数量  //解析过程入口，这里使用了委派模式，BeanDefinitionDocumentReader只是个接口，  //具体的解析实现过程有实现类DefaultBeanDefinitionDocumentReader完成  **documentReader.registerBeanDefinitions(doc, createReaderContext(resource));**  return getRegistry().getBeanDefinitionCount() - countBefore; //统计解析的Bean数量  } |

Bean定义资源的载入解析分为以下两个过程：首先，通过调用XML解析器将BeanDefinition转换得到Document对象，但是这些Document对象并没有按照Spring的Bean规则进行解析。这一步是载入的过程;其次，在完成通用的XML解析之后，按照Spring的Bean规则对Document对象进行解析。

按照Spring的Bean规则对Document对象解析的过程是在接口BeanDefinitionDocumentReader的实现类DefaultBeanDefinitionDocumentReader中实现的。

#### 3.3.4 DefaultBeanDefinitionDocumentReader对BeanDefinition的Document对象解析

BeanDefinitionDocumentReader接口通过registerBeanDefinitions方法调用其实现类DefaultBeanDefinitionDocumentReader对Document对象进行解析。使用Spring时，在Spring配置文件中可以使用<Import>元素来导入IoC容器所需要的其他资源，Spring IoC容器在解析时会首先将指定导入的资源加载进容器中。使用<Ailas>别名时，Spring IoC容器首先将别名元素所定义的别名注册到容器中。

对于既不是<Import>元素，又不是<Alias>元素的元素，即Spring配置文件中普通的<Bean>元素的解析由BeanDefinitionParserDelegate类的parseBeanDefinitionElement方法来实现。

#### 3.3.5 BeanDefinitionParserDelegate解析BeanDefinition的<Bean>元素

BeanDefinition中的<Import>和<Alias>元素解析在DefaultBeanDefinitionDocumentReader中已经完成，对BeanDefinition中使用最多的<Bean>元素交由BeanDefinitionParserDelegate来解析，其解析实现的源码如下：

|  |
| --- |
| //这里是处理BeanDefinition的地方，具体处理委托给BeanDefinitionParserDelegate来完成  protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {  /\*\* BeanDefinitionHolder是BeanDefinition对象的封装类，封装了BeanDefinition，Bean的名字和别名，用来向IoC注册  \* 得到这个BeanDefinitionHolder，就意味着BeanDefinition是通过BeanDefinitionParserDelegate对XML元素的信息按照Spring  \* 对Bean规则进行解析得到的  \*/  **BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);**  if (bdHolder != null) {  bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);  try {  // 这里是向IoC容器注册解析得到BeanDefinition的地方  BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry());  } catch (BeanDefinitionStoreException ex) {  getReaderContext().error("Failed to register bean definition with name '" +bdHolder.getBeanName() + "'", ele, ex);  }  //在BeanDefinition向IOC容器注册完以后，发送信息  getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));  }  } |

具体的Spring BeanDefinition的解析是在BeanDefinitionParserDelegate中完成的。这个类里包含了对各种Spring Bean定义规则的处理。处理比如id、name、aliase等属性元素时，把这些元素的值从XML文件相应的元素的属性中读取出来以后，设置到生成的BeanDefinitionHolder中去，对于其他元素配置的解析，比如各种Bean的属性配置，通过一个较为复杂的解析过程，这个过程是由parseBeanDefinitionElement来完成的，解析完成以后，会把解析结果放到BeanDefinition对象中并设置到BeanDefinitionHolder中去。

|  |
| --- |
| //解析<Bean>元素的入口  public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {  return parseBeanDefinitionElement(ele, null);  }  //解析Bean定义资源文件中的<Bean>元素，这个方法中主要处理<Bean>元素的id，name 和别名属性  public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition containingBean) {  String id = ele.getAttribute(ID\_ATTRIBUTE); //获取<Bean>元素中的id属性值  String nameAttr = ele.getAttribute(NAME\_ATTRIBUTE); //获取<Bean>元素中的name属性值  List<String> aliases = new ArrayList<String>(); //获取<Bean>元素中的alias属性值  if (StringUtils.hasLength(nameAttr)) { //将<Bean>元素中的所有name属性值存放到别名中  String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, BEAN\_NAME\_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("… …");  }  }  //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) {  checkNameUniqueness(beanName, aliases, ele); //检查<Bean>元素所配置的id、name或者别名是否重复  }  //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，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);  //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("… ..");  }  } catch (Exception ex) {  error(ex.getMessage(), ele);  return null;  }  }  String[] aliasesArray = StringUtils.toStringArray(aliases);  return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);  }  return null; //当解析出错时，返回null  }  //对<Bean>元素中配置的除id、name和别名以外的其他属性数据进行解析，  public AbstractBeanDefinition parseBeanDefinitionElement(Element ele, String beanName, BeanDefinition containingBean) {  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));  parseMetaElements(ele, bd); //对<Bean>元素的meta(元信息)属性解析  parseLookupOverrideSubElements(ele, bd.getMethodOverrides()); //对<Bean>元素的lookup-method属性解析  parseReplacedMethodSubElements(ele, bd.getMethodOverrides()); //对<Bean>元素的replaced-method属性解析  parseConstructorArgElements(ele, bd); //解析<Bean>元素的构造方法设置  parsePropertyElements(ele, bd); //解析<Bean>元素的<property>设置  parseQualifierElements(ele, bd); //解析<Bean>元素的qualifier属性  bd.setResource(this.readerContext.getResource()); //为当前解析的Bean设置所需的资源和依赖对象  bd.setSource(extractSource(ele));  return bd;  } catch (…) {  error("…";  }  } finally {  this.parseState.pop();  }  return null; //解析<Bean>元素出错时，返回null  } |

注意：在解析<Bean>元素过程中没有创建和实例化Bean对象，只是创建了Bean对象的定义类BeanDefinition，将<Bean>元素中的配置信息设置到BeanDefinition中作为记录，当依赖注入时才使用这些记录信息创建和实例化具体的Bean对象。

#### 3.3.6 BeanDefinitionParserDelegate解析<property>元素

BeanDefinitionParserDelegate在解析<Bean>调用parsePropertyElements方法解析<Bean>元素中的<property>属性子元素，解析源码如下：

|  |
| --- |
| //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)) {  //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  refName = ele.getAttribute(LOCAL\_REF\_ATTRIBUTE);  if (!StringUtils.hasLength(refName)) {  //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)) {  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;  }  } |

通过对上述源码的分析，我们可以了解在Spring配置文件中，<Bean>元素中<property>元素的相关配置是如何处理的： a. ref被封装为指向依赖对象一个引用。b.value配置都会封装成一个字符串类型的对象。c.ref和value都通过“解析的数据类型属性值.setSource(extractSource(ele));”方法将属性值/引用与所引用的属性关联起来。

在方法的最后对于<property>元素的子元素通过parsePropertySubElement 方法解析。

#### 3.3.7 解析<property>元素的子元素

在BeanDefinitionParserDelegate类中的parsePropertySubElement方法对<property>中的子元素解析，源码如下：

|  |
| --- |
| //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)) {  //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  refName = ele.getAttribute(LOCAL\_REF\_ATTRIBUTE);  if (!StringUtils.hasLength(refName)) {  //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)) {  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;  }  } |

通过上述源码分析，我们明白了在Spring配置文件中，对<property>元素中配置的Array、List、Set、Map、Prop等各种集合子元素的都通过上述方法解析，生成对应的数据对象，比如ManagedList、ManagedArray、ManagedSet等，这些Managed类是Spring对象BeanDefiniton的数据封装，对集合数据类型的具体解析有各自的解析方法实现，解析方法的命名非常规范，一目了然，我们对<list>集合元素的解析方法进行源码分析，了解其实现过程。

#### 3.3.8 解析<list>子元素

在BeanDefinitionParserDelegate类中的parseListElement方法就是具体实现解析<property>元素中的<list>集合子元素。

|  |
| --- |
| //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());  target.setSource(extractSource(collectionEle));  //设置集合目标数据类型  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);  //节点不是description节点  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));  }  }  } |

经过对Spring Bean定义资源文件转换的Document对象中的元素层层解析，Spring IoC现在已经将XML形式定义的Bean定义资源文件转换为Spring IoC所识别的数据结构——BeanDefinition，它是Bean定义资源文件中配置的POJO对象在Spring IoC容器中的映射，我们可以通过AbstractBeanDefinition为入口，让IoC容器进行索引、查询和操作。通过Spring IoC容器对Bean定义资源的解析后，IoC容器大致完成了管理Bean对象的准备工作，即初始化过程，但是最为重要的依赖注入还没有发生，现在在IoC容器中BeanDefinition存储的**只是一些静态信息**，接下来需要向容器注册Bean定义信息才能全部完成IoC容器的初始化过程。

### 3.4 解析过后的BeanDefinition在IoC容器中的注册

#### 3.4.1 DefaultListableBeanFactory向IoC容器注册解析后的BeanDefinition

接下来回到我们第3步中分析DefaultBeanDefinitionDocumentReader对Bean定义转换的Document对象解析的流程中，在其parseDefaultElement方法中完成对Document对象的解析后得到封装BeanDefinition的BeanDefinitionHold对象，然后调用BeanDefinitionReaderUtils的registerBeanDefinition方法向IoC容器注册解析的Bean，BeanDefinitionReaderUtils的注册的源码如下：

|  |
| --- |
| //将解析的BeanDefinitionHold注册到容器中  public static void registerBeanDefinition(BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)  throws BeanDefinitionStoreException {  //获取解析的BeanDefinition的名称  String beanName = definitionHolder.getBeanName();  //向IoC容器注册BeanDefinition  registry.**registerBeanDefinition**(beanName, definitionHolder.getBeanDefinition());  //如果解析的BeanDefinition有别名，向容器为其注册别名  String[] aliases = definitionHolder.getAliases();  if (aliases != null) {  for (String aliase : aliases) {  registry.registerAlias(beanName, aliase);  }  }  } |

**当调用BeanDefinitionReaderUtils向IoC容器注册解析的BeanDefinition时，真正完成注册功能的是DefaultListableBeanFactory。**

#### 3.4.2 DefaultListableBeanFactory向IoC容器注册解析后的BeanDefinition

**DefaultListableBeanFactory中使用一个HashMap的集合对象存放IoC容器中注册解析的BeanDefinition**，向IoC容器注册的主要源码如下：

|  |
| --- |
| //存储注册的俄BeanDefinition  private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<String, BeanDefinition>();  //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");  if (beanDefinition instanceof AbstractBeanDefinition) { //校验解析的BeanDefiniton  try {  ((AbstractBeanDefinition) beanDefinition).validate();  } catch (BeanDefinitionValidationException ex) {  throw new BeanDefinitionStoreException(… …);  }  }  **//注册的过程中需要线程同步，以保证数据的一致性**  **synchronized (this.beanDefinitionMap) {**  **Object 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(… …);  } 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("… …");  }  }  } else { //IoC容器中没有已经注册同名的Bean，按正常注册流程注册  this.beanDefinitionNames.add(beanName);  this.frozenBeanDefinitionNames = null;  }  //把beanName作为Map的key，把BeanDefinition作为value存到IOC容器的BeanDefinitionMap中  this.beanDefinitionMap.put(beanName, beanDefinition);  resetBeanDefinition(beanName); //重置所有已经注册过的BeanDefinition的缓存  }  } |

在DefaultListableBeanFactory中实现了BeanDefinitionRegistry接口，这个接口的实现完成BeanDefinition向容器的注册，即把解析得到的BeanDefinition设置到HashMap中。注意，如果遇到同名的BeanDefinition，进行处理的时候需要依据allowBeanDefinitionOverriding的配置来完成。

至此，Bean定义资源文件中配置的Bean被解析过后，已经注册到IoC容器中，被容器管理起来，真正完成了IoC容器初始化所做的全部工作。现 在IoC容器中已经建立了整个Bean的配置信息，这些BeanDefinition信息已经可以使用，并且可以被检索，IoC容器的作用就是对这些注册的Bean定义信息进行处理和维护。这些的注册的Bean定义信息是IoC容器控制反转的基础，正是有了这些注册的数据，容器才可以进行依赖注入。

### 3.20 总结

现在通过上面的代码，总结一下IOC容器初始化的基本步骤：

(1) 初始化的入口在容器实现中的 refresh()调用来完成；

(2) 对 BeanDefinition载入 IOC 容器使用的方法是 loadBeanDefinition,其中的大致过程如下：通过 ResourceLoader 来完成资源文件位置的定位，DefaultResourceLoader 是默认的实现，同时上下文本身就给出了 ResourceLoader 的实现，可以从类路径，文件系统, URL 等方式来定为资源位置。如果是 XmlBeanFactory作为 IOC 容器，那么需要为它指定 bean 定义的资源，也就是说 bean 定义文件时通过抽象成 Resource 来被 IOC 容器处理的，容器通过 BeanDefinitionReader来完成定义信息的解析和 Bean 信息的注册,往往使用的是XmlBeanDefinitionReader 来解析 bean 的 xml 定义文件 - 实际的处理过程是委托给 BeanDefinitionParserDelegate 来完成的，从而得到 bean 的定义信息，这些信息在 Spring 中使用 BeanDefinition 对象来表示 - 这个名字可以让我们想到loadBeanDefinition,RegisterBeanDefinition 这些相关的方法 - 他们都是为处理 BeanDefinitin 服务的， 容器解析得到 BeanDefinitionIoC 以后，需要把它在 IOC 容器中注册，这由 IOC 实现 BeanDefinitionRegistry 接口来实现。注册过程就是在 IOC 容器内部维护的一个HashMap 来保存得到的 BeanDefinition 的过程。这个 HashMap 是 IoC 容器持有 bean 信息的场所，以后对 bean 的操作都是围绕这个HashMap 来实现的.

(3) 然后我们就可以通过 BeanFactory 和 ApplicationContext 来享受到 Spring IOC 的服务了,在使用 IOC 容器的时候，我们注意到除了少量粘合代码，绝大多数以正确 IoC 风格编写的应用程序代码完全不用关心如何到达工厂，因为容器将把这些对象与容器管理的其他对象钩在一起。基本的策略是把工厂放到已知的地方，最好是放在对预期使用的上下文有意义的地方，以及代码将实际需要访问工厂的地方。 Spring 本身提供了对声明式载入 web 应用程序用法的应用程序上下文,并将其存储在ServletContext 中的框架实现。具体可以参见以后的文章

在使用 Spring IOC 容器的时候我们还需要区别两个概念:

Beanfactory 和FactoryBean，其中 BeanFactory 指的是 IOC 容器的编程抽象，比如 ApplicationContext， XmlBeanFactory 等，这些都是 IOC 容器的具体表现，需要使用什么样的容器由客户决定,但 Spring 为我们提供了丰富的选择。 FactoryBean 只是一个可以在 IOC而容器中被管理的一个 bean,是对各种处理过程和资源使用的抽象,Factory bean 在需要时产生另一个对象，而不返回 FactoryBean本身,我们可以把它看成是一个抽象工厂，对它的调用返回的是工厂生产的产品。所有的 Factory bean 都实现特殊的org.springframework.beans.factory.FactoryBean 接口，当使用容器中 factory bean 的时候，该容器不会返回 factory bean 本身,而是返回其生成的对象。Spring 包括了大部分的通用资源和服务访问抽象的 Factory bean 的实现，其中包括:对 JNDI 查询的处理，对代理对象的处理，对事务性代理的处理，对 RMI 代理的处理等，这些我们都可以看成是具体的工厂,看成是SPRING 为我们建立好的工厂。也就是说 Spring 通过使用抽象工厂模式为我们准备了一系列工厂来生产一些特定的对象,免除我们手工重复的工作，我们要使用时只需要在 IOC 容器里配置好就能很方便的使用了

## 4 IOC容器的依赖注入

### 4.1 依赖注入发生的时间

当Spring IoC容器完成了Bean定义资源的定位、载入和解析注册以后，IoC容器中已经管理类Bean定义的相关数据，但是此时IoC容器还没有对所管理的Bean进行依赖注入，依赖注入在以下两种情况发生：

(1).用户第一次通过getBean方法向IoC容索要Bean时，IoC容器触发依赖注入。

(2).当用户在Bean定义资源中为<Bean>元素配置了lazy-init属性，即让容器在解析注册Bean定义时进行预实例化，触发依赖注入。

BeanFactory接口定义了Spring IoC容器的基本功能规范，是Spring IoC容器所应遵守的最底层和最基本的编程规范。BeanFactory接口中定义了几个getBean方法，就是用户向IoC容器索取管理的Bean的方法，我们通过分析其子类的具体实现，理解Spring IoC容器在用户索取Bean时如何完成依赖注入。下面从DefaultListableBeanFactory的基类AbstractBeanFactory入手看看getBean的实现。

### 4.2 AbstractBeanFactory通过getBean向IoC容器获取被管理的Bean

**AbstractBeanFactory**的getBean相关方法的源码如下

|  |
| --- |
| //获取IoC容器中指定名称的Bean  public Object getBean(String name) throws BeansException {  return doGetBean(name, null, null, false); //doGetBean才是真正向IoC容器获取被管理Bean的过程  }    //**真正实现向IoC容器获取Bean的功能，也是触发依赖注入功能的地方**  @SuppressWarnings("unchecked")  protected <T> T doGetBean( final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)  throws BeansException {  final String beanName = transformedBeanName(name); //如果指定的是别名，将别名转换为规范的Bean名称  Object bean;  //**先从缓存中取是否已经有被创建过的单态类型的Bean，整个IoC容器中只创建一次，不需要重复创建**  Object sharedInstance = getSingleton(beanName);  if (sharedInstance != null && args == null) {  **//IoC容器创建单态模式Bean实例对象**  … …  **//获取给定Bean的实例对象，主要是完成FactoryBean的相关处理**  **//注意：BeanFactory是管理容器中Bean的工厂，而FactoryBean是创建创建对象的工厂Bean**  bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);  } else {  **//缓存没有正在创建的单态模式Bean**  … …  **//对IoC容器中是否存在指定名称的BeanDefinition进行检查，首先检查是否能在当前的BeanFactory中**  **//获取的所需要的Bean，如果不能则委托当前容器的父级容器去查找，**  **//如果还是找不到则沿着容器的继承体系向父级容器查找**  BeanFactory parentBeanFactory = getParentBeanFactory();  //当前容器的父级容器存在，且当前容器中不存在指定名称的Bean  if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {  String nameToLookup = originalBeanName(name); //解析指定Bean名称的原始名称  if (args != null) {  return (T) parentBeanFactory.getBean(nameToLookup, args); //委派父级容器根据指定名称和显式的参数查找  } else {  return parentBeanFactory.getBean(nameToLookup, requiredType); //委派父级容器根据指定名称和类型查找  }  }  if (!typeCheckOnly) { //创建的Bean是否需要进行类型验证，一般不需要  markBeanAsCreated(beanName); //向容器标记指定的Bean已经被创建  }  //根据指定Bean名称获取BeanDefinition  final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);  checkMergedBeanDefinition(mbd, beanName, args);  **//获取当前Bean所有依赖Bean，这样会触发getBean的递归调用，直到取得一个没有任何依赖的bean为止**  String[] dependsOn = mbd.getDependsOn();  if (dependsOn != null) { //如果当前Bean有依赖Bean  for (String dependsOnBean : dependsOn) {  getBean(dependsOnBean); //递归调用getBean方法，获取当前Bean的依赖Bean  registerDependentBean(dependsOnBean, beanName); //把被依赖Bean注册给当前依赖的Bean  }  }  if (mbd.isSingleton()) { //通过调用createBean方法创建单态模式Bean的实例对象  sharedInstance = getSingleton(beanName, new ObjectFactory() {  public Object getObject() throws BeansException {  try {  //创建一个指定Bean实例对象，如果有父级继承，则合并子类和父类的定义  return createBean(beanName, mbd, args);  } catch (BeansException ex) {  destroySingleton(beanName);  throw ex;  }  }  });  bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd); //获取给定Bean的实例对象  }  //IoC容器创建原型模式Bean实例对象  else if (mbd.isPrototype()) {  Object prototypeInstance = null; //原型模式(Prototype)是每次都会创建一个新的对象  try {  //回调beforePrototypeCreation方法，默认的功能是注册当前创//建的原型对象  beforePrototypeCreation(beanName);  prototypeInstance = createBean(beanName, mbd, args); //创建指定Bean对象实例  } finally {  afterPrototypeCreation(beanName);  }  bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  } else {  String scopeName = mbd.getScope();  final Scope scope = this.scopes.get(scopeName);  //Bean定义资源中没有配置生命周期范围，则Bean定义不合法  if (scope == null) {  throw new IllegalStateException("No Scope registered for scope '" + scopeName + "'");  }  try {  Object scopedInstance = scope.get(beanName, new ObjectFactory() {  public Object getObject() throws BeansException {  beforePrototypeCreation(beanName);  try {  return createBean(beanName, mbd, args);  } finally {  afterPrototypeCreation(beanName);  }  }  });  bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd); //获取给定Bean的实例对象  } catch (IllegalStateException ex) {  throw new BeanCreationException(… …);  }  }  }  **//对创建的Bean实例对象进行类型检查，如果没有问题，就返回这个新创建的bean，包含了依赖关系的bean**  if (requiredType != null && bean != null && !requiredType.isAssignableFrom(bean.getClass())) {  throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  }  return (T) bean;  } |

通过上面对向IoC容器获取Bean方法的分析，我们可以看到在Spring中，如果Bean定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果Bean定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。

上面的源码只是定义了根据Bean定义的模式，采取的不同创建Bean实例对象的策略，**具体的Bean实例对象的创建过程由实现了ObejctFactory接口的匿名内部类的createBean方法完成，ObejctFactory使用委派模式，具体的Bean实例创建过程交由其实现类AbstractAutowireCapableBeanFactory完成**，我们继续分析AbstractAutowireCapableBeanFactory的createBean方法的源码，理解其创建Bean实例的具体实现过程。

### 4.3 AbstractAutowireCapableBeanFactory创建Bean实例对象

重点来说，getBean是依赖注入的起点，之后会调用createBean。这个过程中，bean对象会依据BeanDefinition定义的要求生成，在AbstractAutowireCapableBeanFactory中实现了这个createBean，不但生成了需要的bean，还对bean初始化进行了处理，比如实现了在BeanDefinition中的init-method属性定义，bean后置处理器等。

|  |
| --- |
| //创建Bean实例对象  protected Object **createBean**(final String beanName, final RootBeanDefinition mbd, final Object[] args) throws BeanCreationException {  … …  **//判断需要创建的Bean是否可以实例化，即是否可以通过当前的类加载器加载**  resolveBeanClass(mbd, beanName);  //校验和准备Bean中的方法覆盖  try {  mbd.prepareMethodOverrides();  } catch (BeanDefinitionValidationException ex) {  throw new BeanDefinitionStoreException(… …);  }  try {  **//如果Bean配置了初始化前和初始化后的处理器，则试图返回一个需要创建Bean的代理对象**  Object bean = resolveBeforeInstantiation(beanName, mbd);  if (bean != null) {  return bean;  }  } catch (Throwable ex) {  throw new BeanCreationException(… …);  }  Object beanInstance = **doCreateBean**(beanName, mbd, args);  **//创建Bean的入口**  if (logger.isDebugEnabled()) {  logger.debug("Finished creating instance of bean '" + beanName + "'");  }  return beanInstance;  }    protected Object **doCreateBean**(final String beanName, final RootBeanDefinition mbd, final Object[] args) { //真正创建Bean的方法  BeanWrapper instanceWrapper = null; //封装被创建的Bean对象  if (mbd.isSingleton()){ **//如果是Singleton，先把缓存中的同名bean清除**  instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);  }  if (instanceWrapper == null) {  instanceWrapper = **createBeanInstance**(beanName, mbd, args); //创建实例对象  }  final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);  Class beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null); //获取实例化对象的类型  **synchronized** (mbd.postProcessingLock) { //调用PostProcessor后置处理器  if (!mbd.postProcessed) {  applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  mbd.postProcessed = true;  }  }  //向容器中缓存单态模式的Bean对象，以防循环引用  boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&  isSingletonCurrentlyInCreation(beanName));  if (earlySingletonExposure) {  if (logger.isDebugEnabled()) {  logger.debug("… …");  }  addSingletonFactory(beanName, new ObjectFactory() {  public Object getObject() throws BeansException {  return getEarlyBeanReference(beanName, mbd, bean);  }  });  }    **//Bean对象的初始化，依赖注入在此触发 .**这个exposedObject在初始化完成之后返回作为依赖注入完成后的Bean  Object exposedObject = bean;  try {  //将Bean实例对象封装，并且Bean定义中配置的属性值赋值给实例对象  populateBean(beanName, mbd, instanceWrapper);  if (exposedObject != null) {  exposedObject = initializeBean(beanName, exposedObject, mbd); //初始化Bean对象  }  } catch (Throwable ex) {  if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {  throw (BeanCreationException) ex;  } else {  throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);  }  }  if (earlySingletonExposure) {  Object earlySingletonReference = getSingleton(beanName, false); //获取指定名称的已注册的单态模式Bean对象  if (earlySingletonReference != null) {  if (exposedObject == bean) { //根据名称获取的以注册的Bean和正在实例化的Bean是同一个  exposedObject = earlySingletonReference; //当前实例化的Bean初始化完成  }  //当前Bean依赖其他Bean，并且当发生循环引用时不允许新创建实例对象  else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {  String[] dependentBeans = getDependentBeans(beanName);  Set<String> actualDependentBeans = new LinkedHashSet<String>(dependentBeans.length);  for (String dependentBean : dependentBeans) { //获取当前Bean所依赖的其他Bean  if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) { //对依赖Bean进行类型检查  actualDependentBeans.add(dependentBean);  }  }  if (!actualDependentBeans.isEmpty()) {  throw new BeanCurrentlyInCreationException(");  }  }  }  }  try { //注册完成依赖注入的Bean  registerDisposableBeanIfNecessary(beanName, bean, mbd);  } catch (BeanDefinitionValidationException ex) {  throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);  }  return exposedObject;  } |

通过对方法源码的分析，我们看到具体的依赖注入实现在以下两个方法中：

(1).createBeanInstance：生成Bean所包含的java对象实例。

(2).populateBean ：对Bean属性的依赖注入进行处理。

### 4.4 createBeanInstance方法创建Bean的java实例对象

在 **createBeanInstance**中生成了Bean所包含的Java对象，这个对象的生成有很多种不同的方式，可以通过工厂方法生成，也可以通过容器的autowire特性生成，具体有相关的BeanDefinition来指定。

在createBeanInstance方法中，根据指定的初始化策略，使用静态工厂、工厂方法或者容器的自动装配特性生成java实例对象，创建对象的源码如下：

|  |
| --- |
| //创建Bean的实例对象  protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) {  Class beanClass = resolveBeanClass(mbd, beanName); //检查确认Bean是可实例化的  //使用工厂方法对Bean进行实例化  if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {  throw new BeanCreationException();  }  if (mbd.getFactoryMethodName() != null) {  return instantiateUsingFactoryMethod(beanName, mbd, args); //调用工厂方法实例化  }  boolean resolved = false; //使用容器的自动装配方法进行实例化  boolean autowireNecessary = false;  if (args == null) {  synchronized (mbd.constructorArgumentLock) {  if (mbd.resolvedConstructorOrFactoryMethod != null) {  resolved = true;  autowireNecessary = mbd.constructorArgumentsResolved;  }  }  }  if (resolved) {  if (autowireNecessary) {  //配置了自动装配属性，使用容器的自动装配实例化  //容器的自动装配是根据参数类型匹配Bean的构造方法  return autowireConstructor(beanName, mbd, null, null);  } else {  return instantiateBean(beanName, mbd); **//使用默认的无参构造方法实例化**  }  }  //使用Bean的构造方法进行实例化  Constructor[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);  if (ctors != null || mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_CONSTRUCTOR ||  mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {  //使用容器的自动装配特性，调用匹配的构造方法实例化  return autowireConstructor(beanName, mbd, ctors, args);  }  return **instantiateBean**(beanName, mbd); //使用默认的无参构造方法实例化  } |
| //使用默认的无参构造方法实例化Bean对象  protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {  try {  Object beanInstance;  final BeanFactory parent = this;  if (System.getSecurityManager() != null) { //获取系统的安全管理接口，JDK标准的安全管理API  //这里是一个匿名内置类，根据实例化策略创建实例对象  beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {  public Object run() {  return getInstantiationStrategy().instantiate(mbd, beanName, parent);  }  }, getAccessControlContext());  } else { //将实例化的对象封装起来  beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);  }  BeanWrapper bw = new BeanWrapperImpl(beanInstance);  initBeanWrapper(bw);  return bw;  } catch (Throwable ex) {  throw new BeanCreationException();  }  } |

经过对上面的代码分析，我们可以看出，对使用工厂方法和自动装配特性的Bean的实例化相当比较清楚，调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于**我们最常使用的默认无参构造方法就需要使用相应的初始化策略(JDK的反射机制或者CGLIB)来进行初始化了**，在方法getInstantiationStrategy().instantiate中就具体实现类使用初始策略实例化对象。

### 4.5 SimpleInstantiationStrategy类使用默认的无参构造方法创建Bean实例化对象

在使用默认的无参构造方法创建Bean的实例化对象时，方法getInstantiationStrategy().instantiate调用了SimpleInstantiationStrategy类中的实例化Bean的方法。通过上面的代码分析，我们看到了如果Bean有方法被覆盖了，则使用JDK的反射机制进行实例化，否则，使用CGLIB进行实例化。

instantiateWithMethodInjection方法调用SimpleInstantiationStrategy的子类CglibSubclassingInstantiationStrategy使用CGLIB来进行初始化。CGLIB是一个常用的字节码生成器的类库，它提供了一系列API实现java字节码的生成和转换功能。我们在学习JDK的动态代理时都知道，JDK的动态代理只能针对接口，如果一个类没有实现任何接口，要对其进行动态代理只能使用CGLIB。

### 4.6 populateBean方法对Bean属性的依赖注入

在第3步的分析中我们已经了解到Bean的依赖注入分为以下两个过程：

(1).createBeanInstance：生成Bean所包含的java对象实例。

(2).populateBean ：对Bean属性的依赖注入进行处理。

第4、5步中我们已经分析了容器初始化生成Bean所包含的Java实例对象的过程，现在我们继续分析生成对象后，Spring IoC容器是如何将Bean的属性依赖关系注入Bean实例对象中并设置好的，属性依赖注入的代码如下：

|  |
| --- |
| //将Bean属性设置到生成的实例对象上  2 protected void populateBean(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw) {  3 //这里取的在BeanDefinition中设置的property值，即设置的属性值  4 PropertyValues pvs = mbd.getPropertyValues();  6 if (bw == null) { //实例对象为null  7 if (!pvs.isEmpty()) { //属性值不为空  9 throw new BeanCreationException( );  11 } else {  13 return; //实例对象为null，属性值也为空，不需要设置属性值，直接返回  15 }  16 }  17 //在设置属性之前调用Bean的PostProcessor后置处理器  18 boolean continueWithPropertyPopulation = true;  19 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  20 for (BeanPostProcessor bp : getBeanPostProcessors()) {  21 if (bp instanceof InstantiationAwareBeanPostProcessor) {  22 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  23 if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {  24 continueWithPropertyPopulation = false;  25 break;  26 }  27 }  28 }  29 }  30 if (!continueWithPropertyPopulation) {  31 return;  32 }  33  **//依赖注入开始，首先处理autowire自动装配的注入**  34 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME ||  35 mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {  36 MutablePropertyValues newPvs = new MutablePropertyValues(pvs);  37 //**对autowire自动装配的处理，根据Bean名称自动装配注入**  38 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME) {  39 autowireByName(beanName, mbd, bw, newPvs);  40 }  41 //**根据Bean类型自动装配注入**  42 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {  43 autowireByType(beanName, mbd, bw, newPvs);  44 }  45 pvs = newPvs;  46 }  47 boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();  49 boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY\_CHECK\_NONE);  51 if (hasInstAwareBpps || needsDepCheck) {  53 PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw);  54 if (hasInstAwareBpps) {  55 for (BeanPostProcessor bp : getBeanPostProcessors()) {  56 if (bp instanceof InstantiationAwareBeanPostProcessor) {  57 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  59 pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  60 if (pvs == null) {  61 return;  62 }  63 }  64 }  65 }  66 if (needsDepCheck) {  67 checkDependencies(beanName, mbd, filteredPds, pvs); //为要设置的属性进行依赖检查  69 }  70 }  72 **applyPropertyValues**(beanName, mbd, bw, pvs);  **//对属性进行注入**  73 }  74  **//解析并注入依赖属性的过程**  75 protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {  76 if (pvs == null || pvs.isEmpty()) {  77 return;  78 }  80 MutablePropertyValues mpvs = null; //封装属性值  81 List<PropertyValue> original;  82 if (System.getSecurityManager()!= null) {  83 if (bw instanceof BeanWrapperImpl) {  84 ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext()); //设置安全上下文，JDK安全机制  86 }  87 }  88 if (pvs instanceof MutablePropertyValues) {  89 mpvs = (MutablePropertyValues) pvs;  90 if (mpvs.isConverted()) { //属性值已经转换  92 try {  94 bw.setPropertyValues(mpvs); //为实例化对象设置属性值  95 return;  96 } catch (BeansException ex) {  98 throw new BeanCreationException();  100 }  101 }  103 original = mpvs.getPropertyValueList(); //获取属性值对象的原始类型值  104 } else {  106 original = Arrays.asList(pvs.getPropertyValues());  107 }  108 TypeConverter converter = getCustomTypeConverter(); //获取用户自定义的类型转换  110 if (converter == null) {  111 converter = bw;  112 }  113 //创建一个Bean定义属性值解析器，将Bean定义中的属性值解析为Bean实例对象的实际值  115 BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converter);  116 //为属性的解析值创建一个拷贝，将拷贝的数据注入到实例对象中  117 List<PropertyValue> deepCopy = new ArrayList<PropertyValue>(original.size());  118 boolean resolveNecessary = false;  119 for (PropertyValue pv : original) {  121 if (pv.isConverted()) { //属性值不需要转换  122 deepCopy.add(pv);  123 } else { //属性值需要转换  126 String propertyName = pv.getName();  127 //原始的属性值，即转换之前的属性值  128 Object originalValue = pv.getValue();  129 //转换属性值，例如将引用转换为IoC容器中实例化对象引用  130 Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);  131 //转换之后的属性值  132 Object convertedValue = resolvedValue;  133 //属性值是否可以转换  134 boolean convertible = bw.isWritableProperty(propertyName) &&  135 !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);  136 if (convertible) {  137 //使用用户自定义的类型转换器转换属性值  138 convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);  139 }  140 //存储转换后的属性值，避免每次属性注入时的转换工作  141 if (resolvedValue == originalValue) {  142 if (convertible) {  143 //设置属性转换之后的值  144 pv.setConvertedValue(convertedValue);  145 }  146 deepCopy.add(pv);  147 }  148 //属性是可转换的，且属性原始值是字符串类型，且属性的原始类型值不是  149 //动态生成的字符串，且属性的原始值不是集合或者数组类型  150 else if (convertible && originalValue instanceof TypedStringValue &&  151 !((TypedStringValue) originalValue).isDynamic() &&  152 !(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {  153 pv.setConvertedValue(convertedValue);  154 deepCopy.add(pv);  155 }  156 else {  157 resolveNecessary = true;  158 //重新封装属性的值  159 deepCopy.add(new PropertyValue(pv, convertedValue));  160 }  161 }  162 }  163 if (mpvs != null && !resolveNecessary) {  164 //标记属性值已经转换过  165 mpvs.setConverted();  166 }  167 //进行属性依赖注入  168 try {  169 bw.setPropertyValues(new MutablePropertyValues(deepCopy));  170 }  171 catch (BeansException ex) {  172 throw new BeanCreationException(  173 mbd.getResourceDescription(), beanName, "Error setting property values", ex);  174 }  } |

分析上述代码，我们可以看出，对属性的注入过程分以下两种情况：

(1).属性值类型不需要转换时，不需要解析属性值，直接准备进行依赖注入。

(2).属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

对属性值的解析是在BeanDefinitionValueResolver类中的resolveValueIfNecessary方法中进行的，对属性值的依赖注入是通过bw.setPropertyValues方法实现的，在分析属性值的依赖注入之前，我们先分析一下对属性值的解析过程

### 6.7 BeanDefinitionValueResolver解析属性值

当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个Bean实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由resolveValueIfNecessary方法实现。

通过上面的代码分析，我们明白了Spring是如何将引用类型，内部类以及集合类型等属性进行解析的，属性值解析完成后就可以进行依赖注入了，依赖注入的过程就是Bean对象实例设置到它所依赖的Bean对象属性上去，在第7步中我们已经说过，依赖注入是通过bw.setPropertyValues方法实现的，该方法也使用了委托模式，在BeanWrapper接口中至少定义了方法声明，依赖注入的具体实现交由其实现类BeanWrapperImpl来完成。

### 6.8 BeanWrapperImpl对Bean属性的依赖注入

BeanWrapperImpl类主要是对容器中完成初始化的Bean实例对象进行属性的依赖注入，即把Bean对象设置到它所依赖的另一个Bean的属性中去。

通过对上面注入依赖代码的分析，我们已经明白了Spring IoC容器是如何将属性的值注入到Bean实例对象中去的：

(1).对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

(2).对于非集合类型的属性，大量使用了JDK的反射和内省机制，通过属性的getter方法(reader method)获取指定属性注入以前的值，同时调用属性的setter方法(writer method)为属性设置注入后的值。看到这里相信很多人都明白了Spring的setter注入原理。

### 6.9 总结

至此Spring IoC容器对Bean定义资源文件的定位，载入、解析和依赖注入已经全部分析完毕，现在Spring IoC容器中管理了一系列靠依赖关系联系起来的Bean，程序不需要应用自己手动创建所需的对象，Spring IoC容器会在我们使用的时候自动为我们创建，并且为我们注入好相关的依赖，这就是Spring核心功能的控制反转和依赖注入的相关功能。

在Bean的创建和对象依赖注入的过程那个，需要依据BeanDefinition中的信息来递归地完成依赖注入。从上面的几个递归过程中可以看到，这些递归都是以getBean为入口的。一个递归是在上下文体系中查找需要的bean和创建bean的递归ongoing；另一个递归是在依赖注入时，通过递归调用容器的getBean方法，得到当前bean的依赖bean，同时也触发对依赖bean的创建和注入。在对bean的属性进行依赖注入时，解析的过程也是一个递归的过程，这样，依据依赖关系，一层一层地完成bean的创建和注入，知道最后完成当前bean的创建。

## 5 IoC容器的高级特性

### 5.1 Spring IoC容器的lazy-init属性实现预实例化

通过前面我们对IoC容器的实现和工作原理分析，我们知道IoC容器的初始化过程就是对Bean定义资源的定位、载入和注册，此时容器对Bean的依赖注入并没有发生，依赖注入主要是在应用程序第一次向容器索取Bean时，通过getBean方法的调用完成。

当Bean定义资源的<Bean>元素中配置了lazy-init属性时，容器将会在初始化的时候对所配置的Bean进行预实例化，Bean的依赖注入在容器初始化的时候就已经完成。这样，当应用程序第一次向容器索取被管理的Bean时，就不用再初始化和对Bean进行依赖注入了，直接从容器中获取已经完成依赖注入的现成Bean，可以提高应用第一次向容器获取Bean的性能。

下面我们通过代码分析容器预实例化的实现过程：

(1) refresh()

在refresh方法中ConfigurableListableBeanFactorybeanFactory = obtainFreshBeanFactory();启动了Bean定义资源的载入、注册过程，而finishBeanFactoryInitialization方法是对注册后的Bean定义中的预实例化(lazy-init=false，Spring默认就是预实例化，即为true)的Bean进行处理的地方。

(2) finishBeanFactoryInitialization处理预实例化Bean

当Bean定义资源被载入IoC容器之后，容器将Bean定义资源解析为容器内部的数据结构BeanDefinition注册到容器中，AbstractApplicationContext类中的finishBeanFactoryInitialization方法对配置了预实例化属性的Bean进行预初始化过程。

ConfigurableListableBeanFactory是一个接口，其preInstantiateSingletons方法由其子类DefaultListableBeanFactory提供。

(3) DefaultListableBeanFactory对配置lazy-init属性单态Bean的预实例化

通过对lazy-init处理源码的分析，我们可以看出，如果设置了lazy-init属性，则容器在完成Bean定义的注册之后，会通过getBean方法，触发对指定Bean的初始化和依赖注入过程，这样当应用第一次向容器索取所需的Bean时，容器不再需要对Bean进行初始化和依赖注入，直接从已经完成实例化和依赖注入的Bean中取一个线程的Bean，这样就提高了第一次获取Bean的性能。

### 5.2 FactoryBean的实现

在Spring中，有两个很容易混淆的类：BeanFactory和FactoryBean。

BeanFactory：Bean工厂，是一个工厂(Factory)，我们Spring IoC容器的最顶层接口就是这个BeanFactory，它的作用是管理Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean：工厂Bean，是一个Bean，作用是产生其他bean实例。通常情况下，这种bean没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他bean实例。通常情况下，bean无须自己实现工厂模式，Spring容器担任工厂角色；但少数情况下，容器中的bean本身就是工厂，其作用是产生其它bean实例。

当用户使用容器本身时，可以使用转义字符”&”来得到FactoryBean本身，以区别通过FactoryBean产生的实例对象和FactoryBean对象本身。在BeanFactory中通过如下代码定义了该转义字符：

StringFACTORY\_BEAN\_PREFIX = "&";

如果myJndiObject是一个FactoryBean，则使用&myJndiObject得到的是myJndiObject对象，而不是myJndiObject产生出来的对象。

**(1). AbstractBeanFactory的getBean方法调用FactoryBean**

在前面我们分析Spring Ioc容器实例化Bean并进行依赖注入过程的源码时，提到在getBean方法触发容器实例化Bean的时候会调用AbstractBeanFactory的doGetBean方法来进行实例化的过程。

在上面获取给定Bean的实例对象的getObjectForBeanInstance方法中，会调用FactoryBeanRegistrySupport类的getObjectFromFactoryBean方法，该方法实现了Bean工厂生产Bean实例对象。

Dereference(解引用)：一个在C/C++中应用比较多的术语，在C++中，”\*”是解引用符号，而”&”是引用符号，解引用是指变量指向的是所引用对象的本身数据，而不是引用对象的内存地址。

**(2). AbstractBeanFactory生产Bean实例对象**

从上面的源码分析中，我们可以看出，BeanFactory接口调用其实现类的getObject方法来实现创建Bean实例对象的功能。

**(3).工厂Bean的实现类getObject方法创建Bean实例对象**

FactoryBean的实现类有非常多，比如：Proxy、RMI、JNDI、ServletContextFactoryBean等等，FactoryBean接口为Spring容器提供了一个很好的封装机制，具体的getObject有不同的实现类根据不同的实现策略来具体提供。其他的Proxy，RMI，JNDI等等，都是根据相应的策略提供getObject的实现。

### 5.3 BeanPostProcessor后置处理器的实现

BeanPostProcessor后置处理器是Spring IoC容器经常使用到的一个特性，这个Bean后置处理器是一个监听器，可以监听容器触发的Bean声明周期事件。后置处理器向容器注册以后，容器中管理的Bean就具备了接收IoC容器事件回调的能力。

BeanPostProcessor的使用非常简单，只需要提供一个实现接口BeanPostProcessor的实现类，然后在Bean的配置文件中设置即可。

(1).BeanPostProcessor的源码如下

这两个回调的入口都是和容器管理的Bean的生命周期事件紧密相关，可以为用户提供在Spring IoC容器初始化Bean过程中自定义的处理操作。

(2).AbstractAutowireCapableBeanFactory类对容器生成的Bean添加后置处理器

BeanPostProcessor后置处理器的调用发生在Spring IoC容器完成对Bean实例对象的创建和属性的依赖注入完成之后，在对Spring依赖注入的源码分析过程中我们知道，当应用程序第一次调用getBean方法(lazy-init预实例化除外)向Spring IoC容器索取指定Bean时触发Spring IoC容器创建Bean实例对象并进行依赖注入的过程，其中真正实现创建Bean对象并进行依赖注入的方法是AbstractAutowireCapableBeanFactory类的doCreateBean方法。从上面的代码中我们知道，为Bean实例对象添加BeanPostProcessor后置处理器的入口的是initializeBean方法。

(3).initializeBean方法为容器产生的Bean实例对象添加BeanPostProcessor后置处理器：

同样在AbstractAutowireCapableBeanFactory类中，initializeBean方法实现为容器创建的Bean实例对象添加BeanPostProcessor后置处理器。BeanPostProcessor是一个接口，其初始化前的操作方法和初始化后的操作方法均委托其实现子类来实现，在Spring中，BeanPostProcessor的实现子类非常的多，分别完成不同的操作，如：AOP面向切面编程的注册通知适配器、Bean对象的数据校验、Bean继承属性/方法的合并等等，我们以最简单的AOP切面织入来简单了解其主要的功能。

(4).AdvisorAdapterRegistrationManager在Bean对象初始化后注册通知适配器

AdvisorAdapterRegistrationManager是BeanPostProcessor的一个实现类，其主要的作用为容器中管理的Bean注册一个面向切面编程的通知适配器，以便在Spring容器为所管理的Bean进行面向切面编程时提供方便。其他的BeanPostProcessor接口实现类的也类似，都是对Bean对象使用到的一些特性进行处理，或者向IoC容器中注册，为创建的Bean实例对象做一些自定义的功能增加，这些操作是容器初始化Bean时自动触发的，不需要认为的干预。

### 5.4 Spring IoC容器autowiring实现原理

Spring IoC容器提供了两种管理Bean依赖关系的方式：

a.显式管理：通过BeanDefinition的属性值和构造方法实现Bean依赖关系管理。

b．autowiring：Spring IoC容器的依赖自动装配功能，不需要对Bean属性的依赖关系做显式的声明，只需要在配置好autowiring属性，IoC容器会自动使用反射查找属性的类型和名称，然后基于属性的类型或者名称来自动匹配容器中管理的Bean，从而自动地完成依赖注入。

通过对autowiring自动装配特性的理解，我们知道容器对Bean的自动装配发生在容器对Bean依赖注入的过程中。在前面对Spring IoC容器的依赖注入过程源码分析中，我们已经知道了容器对Bean实例对象的属性注入的处理发生在AbstractAutoWireCapableBeanFactory类中的populateBean方法中，我们通过程序流程分析autowiring的实现原理：

(1). AbstractAutoWireCapableBeanFactory对Bean实例进行属性依赖注入：

应用第一次通过getBean方法(配置了lazy-init预实例化属性的除外)向IoC容器索取Bean时，容器创建Bean实例对象，并且对Bean实例对象进行属性依赖注入，AbstractAutoWireCapableBeanFactory的populateBean方法就是实现Bean属性依赖注入的功能.

(2).Spring IoC容器根据Bean名称或者类型进行autowiring自动依赖注入：

通过上面的源码分析，我们可以看出来通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是DefaultSingletonBeanRegistry类的registerDependentBean方法。

(3).DefaultSingletonBeanRegistry的registerDependentBean方法对属性注入：

通过对autowiring的源码分析，我们可以看出，autowiring的实现过程：

a.对Bean的属性迭代调用getBean方法，完成依赖Bean的初始化和依赖注入。

b.将依赖Bean的属性引用设置到被依赖的Bean属性上。

c.将依赖Bean的名称和被依赖Bean的名称存储在IoC容器的集合中。

Spring IoC容器的autowiring属性自动依赖注入是一个很方便的特性，可以简化开发时的配置，但是凡是都有两面性，自动属性依赖注入也有不足，首先，Bean的依赖关系在配置文件中无法很清楚地看出来，对于维护造成一定困难。其次，由于自动依赖注入是Spring容器自动执行的，容器是不会智能判断的，如果配置不当，将会带来无法预料的后果，所以自动依赖注入特性在使用时还是综合考虑。

## 6 总结

为了说明Spring的实现原理，分析了包括 IoC容器和上下文的基本工作原理、容器的初始化过程、依赖注入的实现。总的来说，关于容器的基本原理，可以大致整理出以下几个方面：

**1) BeanDefinition的定位**

在初始化IoC容器的过程中，首先需要定位到这些有效的Bean定义信息，这里Spring使用Resource接口来统一这些Bean定义信息，而这个定位由ResourceLoader来完成。如果使用上下文，ApplicationContext本身就为客户提供了定位的功能。因为上下文本身就是DefaultResourceLoader的子类，如果使用基本的BeanFactory最为IoC容器，需要做的额外工作就是为BeanFactory指定相应的Resource来完成Bean信息的定位。

**2) 容器的初始化**

这个过程的入口是在refresh中实现的，这个refresh相当于容器的初始化函数。在初始化过程中，具体包括BeanDefinition的Resource定位、载入和注册这三个基本的过程。第一个过程是Resource定位过程，这个Resource定位指的是BeanDefinition的资源定位，它由ResourceLoader通过统一的Resource接口来完成；第二个过程是BeanDefinition的载入，这个载入过程是把用户定义好的Bean表示成IoC容器内部的数据结构，由BeanDefinitionReader来完成BeanDefinition的读取、解析和IoC容器内部BeanDefinition的建立。第三个过程是想IoC容器注册这些BeanDefinition的过程，这个过程是通过调用BeanDefinitionRegistry接口的实现来完成，以后的IoC容器对bean的管理和操作就是通过这些BeanDefinition来完成的。

**3) 依赖注入**

在容器初始化完成以后，IoC容器的使用就准备好了，但这是只是在IoC容器内部建立了BeanDefinition，具体的依赖关系还没有注入。在第一次想IoC容器请求Bean时，IoC容器对相关的Bean依赖关系进行注入。如果需要提前注入，可以通过lazy-init属性进行与实例化，起到提前完成依赖注入的控制作用。在依赖注入完成之后，可以通过getBean来取得Bean，这些Bean不是简单的Java对象，而是已经包含了对象之间依赖关系的Bean。

**Spring Bean的创建是典型的工厂模式。**

### 6.1 相关概念

**1) IoC容器**

**BeanFactory**作为最顶层的一个接口类，它定义了IoC容器的基本功能规范。BeanFactory里只对IoC容器的基本行为作了定义，根本不关心你的bean是如何定义怎样加载的。在Spring中，所有的Bean都是由BeanFactory来进行管理的。

**FactoryBean**：在使用容器时，可以使用转义符“&”来得到FactoryBean本身，用来区分通过容器来获取FactoryBean产生的对象和获取FactoryBean本身。FactoryBean不是简单的Bean，是一个能产生或者修饰对象生成的工厂Bean，**它的实现与设计模式中的工厂模式和修饰器模式类似。**

**2) ApplicationContext**

ApplicationContext是Spring提供的一个高级的IoC容器，它除了提供IoC容器的基本功能外，还为用户提供了以下的附加服务。1.支持信息源，可以实现国际化。（实现MessageSource接口）； 2.访问资源。(实现ResourcePatternResolver接口，这个后面要讲)； 3.支持应用事件。(实现ApplicationEventPublisher接口)。

**3) BeanDefinition**

Bean对象在Spring实现中是以BeanDefinition来描述的. Spring通过定义BeanDefinition来管理基于Spring的应用中的各种对象以及他们之间的相互依赖关系。

### 6.2 BeanFactory设计原理

**DefaultListableBeanFactory作为一个默认的功能完整的IoC容器使用**。

使用IoC容器时，需要如下几个步骤：

创建IoC配置文件的抽象资源，这个抽象资源包含了BeanDefinition的定义信息；

创建一个BeanFactory，这里使用DefaultListableBeanFactory。

创建一个载入BeanDefinition的读取器，这里使用XMLBeanDefinitionReader，通过回调配置给BeanFactory。

从定义好的资源位置读取配置信息，具体的解析过程由XMLBeanDefinitionReader来完成，完成载入和注册之后，IoC容器就建立起来了。

### 6.3 IoC容器的初始化

IoC容器的初始化是由refresh方法来启动的，这个启动包括：BeanDefinition的Resource定位、载入和注册三个基本过程。

# 二 Spring Aop原理

## 1 什么是AOP

AOP（Aspect Oriented Programming），即面向切面编程。

AOP利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。Aspect：是一种新的模块化机制，分离关注点使解决特定领域问题的代码从业务逻辑中独立出来。使用AOP后，不仅可以讲这些重复的代码抽取出来单独维护，在需要使用时统一调用，还可以为如何使用这些公共代码提供丰富灵活的手段。

### 1.1 相关概念

**横切关注点**：对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点；

**切面(Aspect)**：切面就是对横切关注点的抽象。官方的抽象定义为“一个关注点的模块化，这个关注点可能

会横切多个对象”。PointCut + Advice 形成了切面Aspect。

**连接点(Joinpoint)：**被拦截到的点, 在Spring中连接点指的就是被拦截到的方法。

**切入点(Pointcut)：**对连接点进行拦截的定义。AOP框架允许开发者指定切入点：例如使用正则表达式。

**通知(Advice)：**在特定的连接点，AOP框架执行的动作。各种类型的通知包括“around”、“before”和“throws”。

**目标对象(Target Object)：**被一个或者多个切面所通知的对象。

**AOP代理(AOP Proxy)：**在Spring AOP中有两种代理方式，JDK动态代理和CGLIB代理。

### 1.2 通知(Advice)类型

**前置通知(Before advice)：**在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。

ApplicationContext中在<aop:aspect>里面使用<aop:before>元素进行声明。

**后通知(After advice)：**当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

ApplicationContext中在<aop:aspect>里面使用<aop:after>元素进行声明。

**返回后通知(After return advice)：**在某连接点**正常完成**后执行的通知，不包括抛出异常的情况。ApplicationContext中在<aop:aspect>里面使用<after-returning>元素进行声明。

**环绕通知(Around advice)：**包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext中在<aop:aspect>里面使用<aop:around>元素进行声明。

**抛出异常后通知(After throwing advice)：** 在方法抛出异常退出时执行的通知。 ApplicationContext中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。使用反射机制完成。

**切入点表达式** ：如execution(\* com.spring.service.\*.\*(..))。

### 1.3 静态代理

|  |
| --- |
| public interface IPerson {  public void doSomething();  } |
| public class Person implements IPerson {  public void doSomething(){  System.out.println("I want wo sell this house");  }  } |
| public class PersonProxy {  private IPerson iPerson;  private final static Logger logger = LoggerFactory.getLogger(PersonProxy.class);  public PersonProxy(IPerson iPerson) {  this.iPerson = iPerson;  }  public void doSomething() {  logger.info("Before Proxy");  iPerson.doSomething();  logger.info("After Proxy");  }  public static void main(String[] args) {  PersonProxy personProxy = new PersonProxy(new Person());  personProxy.doSomething();  }  } |

### 1.4 动态代理

|  |
| --- |
| public class PersonProxy implements InvocationHandler{  private Object delegate;  private final Logger logger = LoggerFactory.getLogger(this.getClass();  public Object bind(Object delegate) {  this.delegate = delegate;  return Proxy.newProxyInstance(delegate.getClass().getClassLoader(), delegate.getClass().getInterfaces(), this);  }  public Object **invoke**(Object proxy, Method method, Object[] args) throws Throwable {  Object result = null;  try {  logger.info("Before Proxy");  result = method.invoke(delegate, args);  logger.info("After Proxy");  } catch (Exception e) {  throw e;  }  return result;  }  public static void main(String[] args) {  PersonProxy personProxy = new PersonProxy();  IPerson iperson = (IPerson) personProxy.bind(new Person());  iperson.doSomething();  }  } |

它的好处理时可以为我们生成任何一个接口的代理类，并将需要增强的方法织入到任意目标函数。但它仍然具有一个局限性，就是只有实现了接口的类，才能为其实现代理。

### 1.5 CGLIB

|  |
| --- |
| public class PersonProxy implements MethodInterceptor {  private Object delegate;  private final Logger logger = LoggerFactory.getLogger(this.getClass());  public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  logger.info("Before Proxy");  Object result = methodProxy.invokeSuper(method, args);  logger.info("After Proxy");  return result;  }  public static Person getProxyInstance() {  Enhancer enhancer = new Enhancer();  enhancer.setSuperclass(Person.class);    enhancer.setCallback(new PersonProxy());  return (Person) enhancer.create();  }  } |

CGLIB解决了动态代理的难题，它通过生成目标类子类的方式来实现来实现代理，而不是接口，规避了接口的局限性。当然CGLIB也具有局限性，对于无法生成子类的类（final类），肯定是没有办法生成代理子类的。

## 2 如何使用Spring AOP

配置可以通过xml文件来进行，大概有四种方式：

1.配置ProxyFactoryBean，显式地设置advisors, advice, target等

2.配置AutoProxyCreator，如以前一样使用定义的bean，但是从容器中获得的其实已经是代理对象

3.通过<aop:config>来配置

**4.通过<aop: aspectj-autoproxy>来配置，使用AspectJ的注解来标识通知及切入点C:\Users\Laura\AppData\Local\Temp\SGPicFaceTpBq\7000\B4097234.png**

## 3 Spring AOP的设计与实现

在Spring AOP实现中，使用的核心技术是：**动态代理**。

JDK中已经实现了这个Proxy模式，在使用时，还需要为代理对象设计一个回调方法，这个回调方法，如果在JDK中实现，需要实现InvocationHandler接口，实现invoke方法，invoke方法的第一个参数是代理对象实例，第二个参数是Method方法对象。

|  |
| --- |
| public interface InvocationHandler {  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable;  } |

为了让AOP起作用，需要完成一系列过程

## 3 Spring AOP代理对象的生成

### 3.1 设计原理

**Spring提供了两种方式来生成代理对象: JDKProxy和Cglib**，具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。

ProxyConfig看成是一个数据基类，这个数据基类为ProxyFactoryBean这样的子类提供了配置属性，AdvisedSupport的实现中，封装了AOP对通知和通知器的相关操作。**但对于具体的AOP代理对象的创建，AdvisedSupport把它交给它的子类们去完成**。

AspectJProxyFactory起到集成Spring和AspectJ的作用；ProxyFactoryBean和ProxyFactory都提供了AOP功能的封装，只是ProxyFactoryBean可以在IoC容器中完成声明式配置，而ProxyFactory需要编程式地使用。

### 3.2代理对象生成了，那切面是如何织入的？

我们知道**InvocationHandler接口是JDK动态代理**的核心，**生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法**。而通过JdkDynamicAopProxy的签名我们可以看到这个类其实也实现了InvocationHandler，下面我们就通过分析这个类中实现的invoke()方法来具体看下Spring AOP是如何织入切面的。

主流程可以简述为：获取可以应用到此方法上的通知链（Interceptor Chain）,如果有,则应用通知,并执行joinpoint; 如果没有,则直接反射执行joinpoint。而这里的关键是通知链是如何获取的以及它又是如何执行的。

首先，通知链是通过Advised.getInterceptorsAndDynamicInterceptionAdvice()这个方法来获取的，实际的获取工作其实是由AdvisorChainFactory. getInterceptorsAndDynamicInterceptionAdvice()这个方法来完成的，获取到的结果会被缓存【从提供的配置实例config中获取advisor列表,遍历处理这些advisor.如果是IntroductionAdvisor,则判断此Advisor能否应用到目标类targetClass上.如果是PointcutAdvisor,则判断此Advisor能否应用到目标方法method上.将满足条件的Advisor通过AdvisorAdaptor转化成Interceptor列表返回.】这个方法执行完成后，Advised中配置能够应用到连接点或者目标类的Advisor全部被转化成了MethodInterceptor。

接下来我们再看下得到的拦截器链是怎么起作用的。如果得到的拦截器链为空，则直接反射调用目标方法，否则创建MethodInvocation，调用其proceed方法，触发拦截器链的执行。

## 4 总结