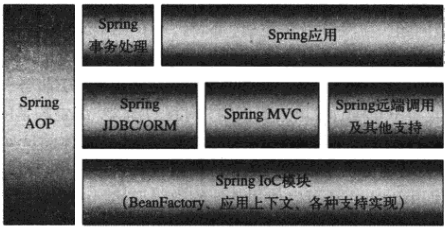
**1.Spring的整体架构**



Spring IoC：包含了最基本的IoC容器的BeanFactory的**接口与实现。**BeanFactory系列容器只实现了最基本的IoC容器的功能；ApplicationContext应用上下文，作为容器的高级形态存在，增加了许多面向框架的特性，包括国际化和应用支持。

Spring Aop：集成AspectJ作为AOP的一个特定实现，同时还在JVM动态代理和CGLIB的基础上，实现了AOP框架；声明式事务，日志处理，权限判断。

Spring MVC：以DispatcherServlet为核心，实现了MVC模式，包括怎样与Web容器集成，Web请求的拦截、转发、处理和ModelAndView数据的返回。

Spring JDBC/ORM：对JDBC进行封装，使得通过JDBC完成的对数据库的操作更加方便，还提供了JdbcTemplate作为模板类，封装了基本数据库的操作方法。还提供了许多ORM工具的封装，例如HibernateTemplate等。

Spring事务处理：通过Spring AOP实现自身功能增强的典型模块。

Spring 远端调用：通过Spring的封装从Spring应用到Spring应用之间的端到端的调用。在这个过程中，通过Spring的封装，为应用屏蔽了各种通信和调用的细节，从而可以使用不同的远端调用来实现，HTTP调用器已经传统的RMI调用。

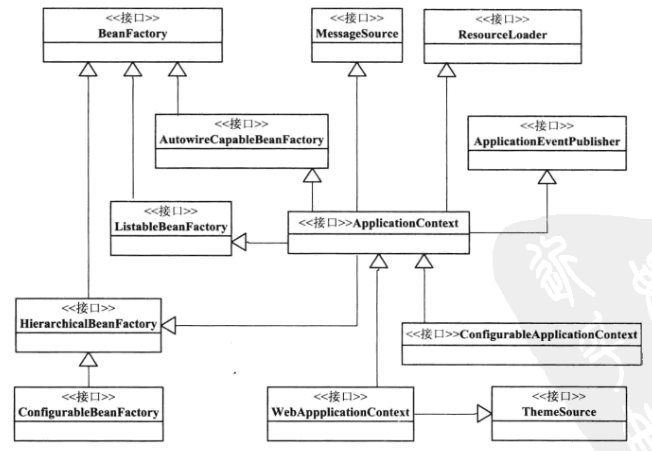
Spring 应用：通过这个模块使得Spring应用能够简洁的容纳第三方的技术实现，丰富了Spring的应用功能。

**2.Spring IoC容器的概述**

控制反转，控制权的反转，控制权从具体业务对象手中转到IoC容器中；如果相关对象的引用或依赖关系的管理由具体对象来完成，会导致代码的高度耦合。对象的依赖关系往往体现在对数据和方法的依赖上，这些依赖关系可以通过把对象的依赖注入交给框架或者IoC容器来完成，这种控制权的转移能够降低耦合提高可测试性。

控制反转的实现有很多种方式，可以在对象初始化时，就进行数据注入，也可以通过对象引用注入到数据域中实现。对象之间的相互依赖关系由IoC容器进行管理，并由IoC容器完成对象的注入。

**2.1IoC容器的设计**



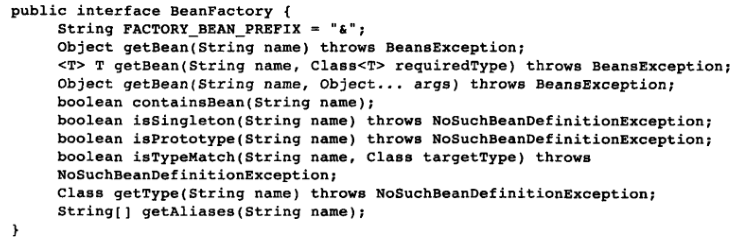
①BeanFactory-HierachicalBeanFactory-ConfigurableBeanFactory，HierachicalBeanFactory继承了BeanFactory，增加了getParentBeanFactory的接口功能，ConfigurableBeanFactory接口，主要定义了一些对BeanFactory的配置功能。

②BeanFactory-ListableBeanFactory-ApplicationContext-ConfigurableApplicationContext，ListableBeanFactory细化了好多BeanFactory的接口功能，如定义了getBeanDefinitionNames接口方法，ApplicationContext通过集成MessageSource、ResourceLoader、ApplicationEventPublisher接口，添加了许多对高级容器的特性支持。

BeanFactory与FactoryBean，BeanFactory是IoC容器或者对象工厂，所有的Bean都是由BeanFactory进行管理的。FactoryBean是一个能够产生对象的工厂Bean，**使用&可以得到FactoryBean本身，不会得到FactoryBean产生的对象**。

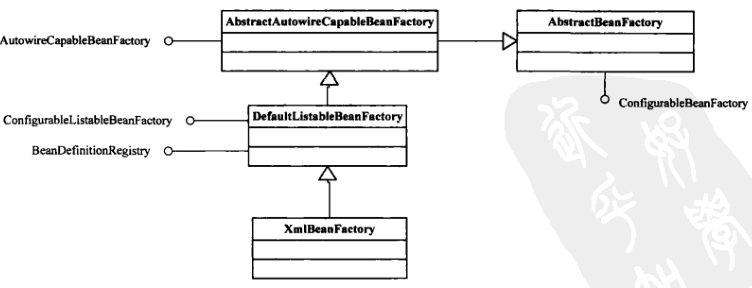
**2.1.1BeanFactory的应用场景**

BeanFactory的getBean方法，可以取得IoC容器中管理的Bean，通过指定名字来索引，还可以带Bean的类型，或者prototype类型的参数。

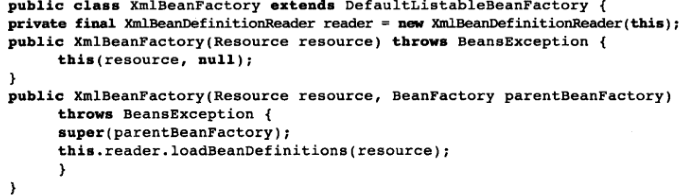


**2.1.2 BeanFactory容器的设计原理**

以XmlBeanFactory为例，简单说明IoC容器的设计原理

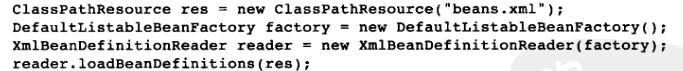


只提供最基本的IoC容器的功能。XmlBeanFactory继承了DefaultListableBeanFactory这个类，DefaultListableBeanFactory是经常需要用到的一个IoC的实现，ApplicationContext就会使用到该类，它包含了基本IoC容器所具有的重要功能。XmlBeanFactory在DefaultListableBeanFactory的基础上增加了可以读取xml文件定义的BeanDefinition的功能。实现过程如下



XmlBeanFactory中初始化了一个XmlBeanDefinitionReader对象，该对象那个可以处理xml方式定义的BeanDefinition。构造XmlBeanFactory的时候，需要指定BeanDefinition的信息来源（某个xml文件），而这个信息来源需要封装成Resource类（用来封装IO的类），例如ClassPathResource res = new ClassPathResource(“beans.xml”)；将resource作为构造参数传入到XmlBeanFactory的构造器中，调用loadBeanDefinition方法，从Resource中载入BeanDefinition。

**理解编程方式使用IoC容器**



①创建IoC配置文件的抽象资源Resource，包含了BeanDefinition的定义

②创建一个BeanFactory，这里使用DefaultListableBeanFactory

③创建一个载入BeanFactory的读取器，使用XmlBeanDefinitionReader来载入xml文件形式的BeanDefinition。

④调用loadBeanDefinition，完成载入信息操作，载入和注册bean后，就可以使用IoC容器了。

**2.1.3ApplicationContext的应用场景**

ApplicationContext是一个高级形态意义的IoC容器，提供了一些附加功能

①继承了MessageSource接口，支持国际化实现

②继承了ResourceLoader接口，可以从不同地方得到Bean定义资源

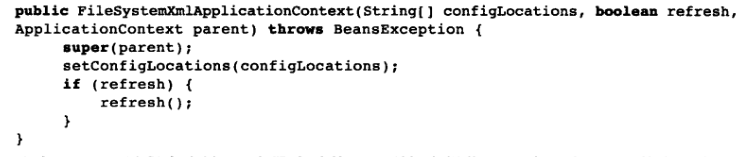
③继承了ApplicationEventPublisher接口，引入了事件机制，与Bean生命周期结合为Bean的管理提供便利。

**2.1.4 ApplicationContext容器的设计原理**

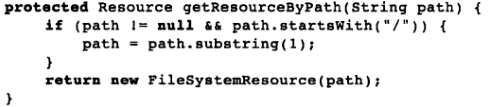
以FileSystemXmlApplicationContext的实现为例

ApplicationContext主要功能已经在AbstractXmlApplicationContext中实现了，FileSystemXmlApplicationContext只需要实现与其自身设计相关的功能即可

①如果直接使用FileSystemXmlApplicationContext，对于实例化ApplicationContext的支持，同时启动IoC容器的refresh过程



②怎样从文件系统中加载xml的bean定义资源



从xml中得到BeanDefinition的抽象资源，然后返回Resource

**2.2 IoC容器的初始化过程**

IoC的初始化过程是由refresh来启动，这个方法标志着IoC容器的正式启动。启动过程包括BeanDefinition的Resource定位、BeanDefinition的载入和BeanDefinition的注册。

Spring把三个过程分开，并使用不同模块来完成，通过这样的设计方式，可以让用户更加灵活地对这三个过程进行裁剪和扩展。

①Resource定位，由ResourceLoader通过统一的Resource接口来完成，可以使用上述getResourceByPath完成，也可以使用ClassPathResource来完成。

②BeanDefinition载入，用户定义好的Bean表示成IoC容器内部的数据结构，而这个容器的内部数据结构就是BeanDefinition。

③注册BeanDefinition，调用BeanDefinitionRegistry接口实现。IoC容器内部将BeanDefinition注入到一个HashMap中去，通过HashMap持有这些BeanDefinition

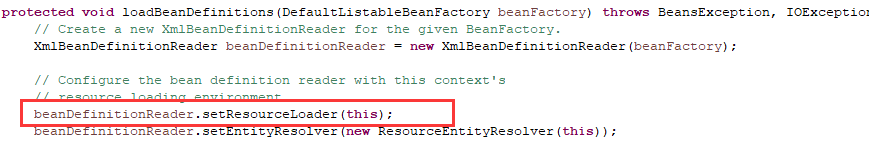
**注意：一般来说初始化和依赖注入是两个分开的过程，依赖注入是在第一次调用getBean时进行的，即默认的lazyinit属性是true，有个例外，对bean设置了lazyinit属性为false，并且是singleton之后，那么这个Bean的依赖注入就会在容器初始化时就预先完成了。如果是prototype类型，那么即使lazyinit是false，那么也不会在初始化就进行依赖注入，详情见DefaultListableBeanFactory中的preInstantiateSingletons()方法**

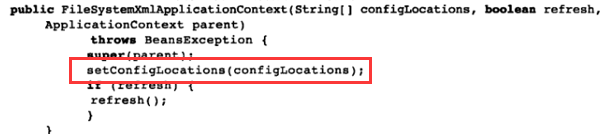
**2.2.1 Resource定位**

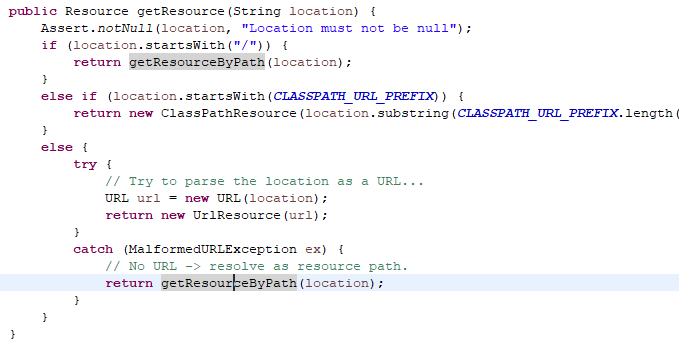
以FileSystemXmlApplicationContext为例，通过继承AbstractApplicationContext具备了ResourceLoader从Resource中读入BeanDefinition的能力；

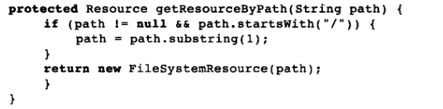


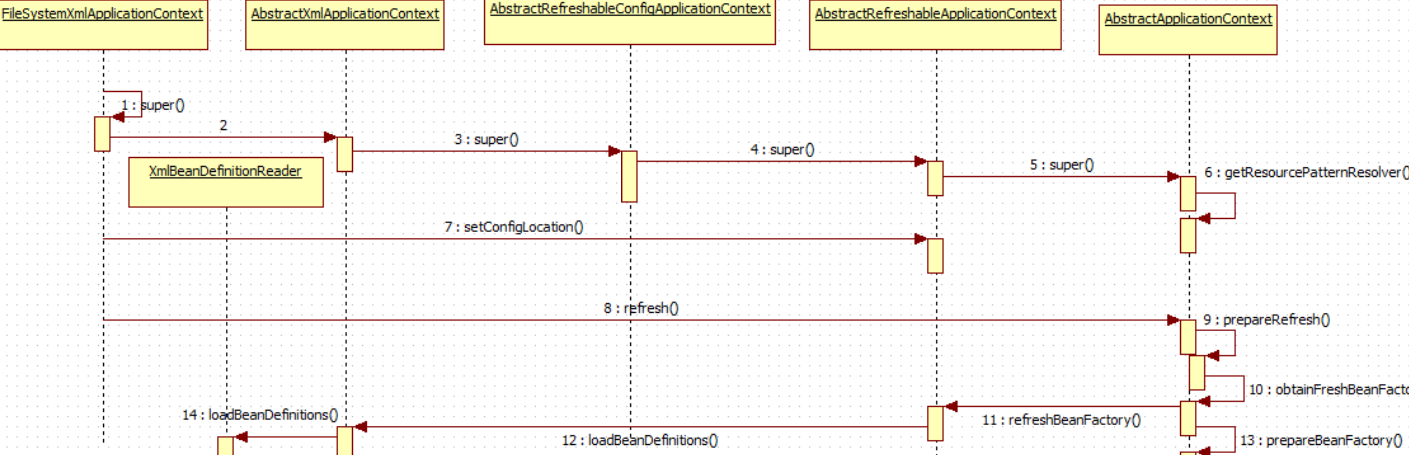
getResourceBypath方法用于BeanDefinition的定位，**构造器中的refresh方法用于实现BeanDefinition的载入过程**。FileSystemXmlApplicationContext的refresh方法调用AbstractRefreshableApplicationContext的refreshBeanFactory方法。refreshBeanFactory通过createBeanFactory构建了一个IoC容器供ApplicationContext使用，就是defaultlistableBeanfactory，然后调用loadBeanDefinitions来载入BeanDefinition，方法参数就是defaultlistableBeanfactory；该过程实际上调用的是AbstractApplicationContext里面的loadBeanDefinitions，然后初始化了XmlBeanDefinitionReader，调用loadBeanDefinitions(XmlBeanDefinitionReader reader)方法，里面有获取getConfigLocations，之后就会在**AbstractBeanDefinitionReader**里面的loadBeanDefinitions(String location, Set<Resource> actualResources)方法getResourceLoader，此时get到的ResourceLoader就是FileSystemXmlApplicationContext，然后就会调用ResourceLoader的getResource方法，由于FileSystemXmlApplicationContext继承了DefaultResourceLoader，所以调用其中的getResource方法；如下。



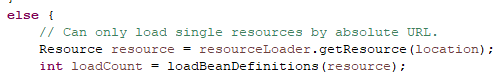






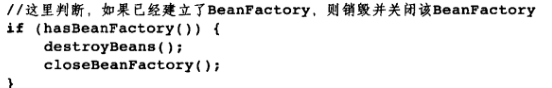


由于FileSystemXMLApplicationContext设定了configLocation，所以用的是reader.loadBeanDefinitions(configLocations);这个方法，此方法在getResource得到Resource之后就会进行BeanDefinition的载入

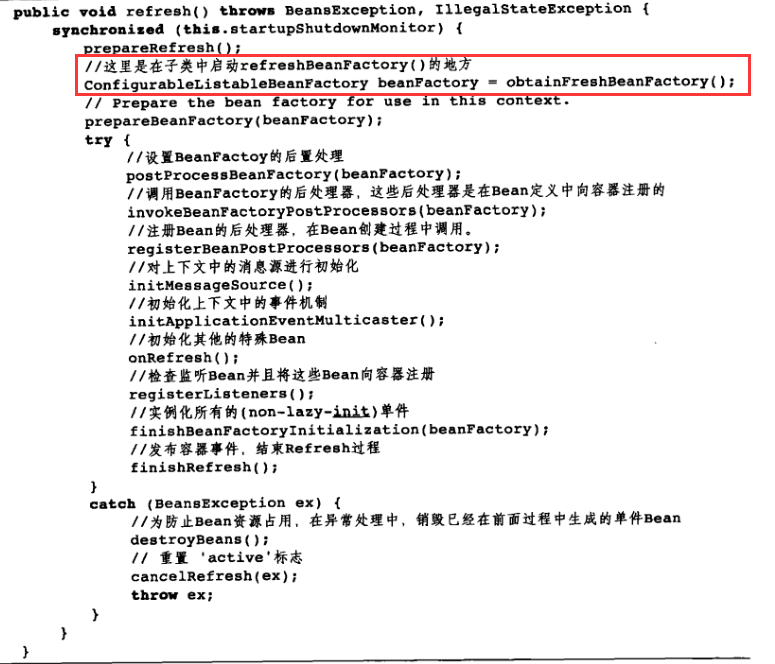
由于AbstractBeanDefinitionReader没有实现loadBeanDefinitions(resource);因此调用xmlBeanDefinitionReader类中的loadBeanDefinitions(resource);进行载入；

**2.2.2 BeanDefinition的载入和解析**

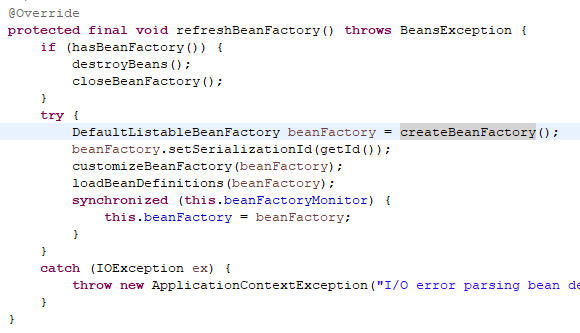
Refresh方法的实现，是在AbstractApplicationContext中，代码如下，BeanFactory的更新，此方法在AbstractRefreshableApplicationContext类的refreshBeanFactory方法



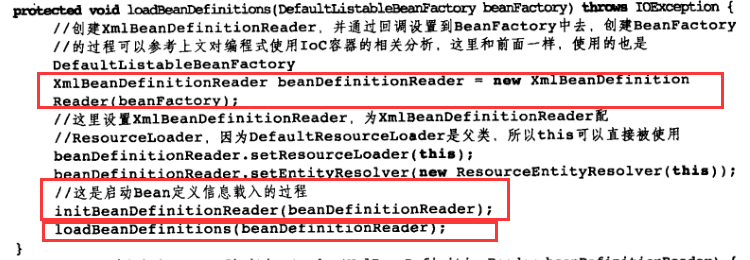
如果有旧的beanFactory，就销毁和关闭并且创建新的BeanFactory；MessageSource和PostProcessor的注册等。



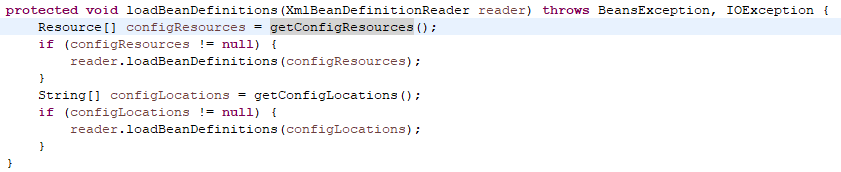
真正BeanDefinition的载入过程在refreshBeanFactory方法中



loadBeanDefinitions是一个抽象方法，在AbstractXmlApplicationContext中实现，在该方法中，初始化了XmlBeanDefinitionReader，然后把这个读取器在IoC容器中设置好，在loadBeanDefinitions(beanDefinitionReader);调用reader.loadBeanDefinitions(configLocations)完成BeanDefinition的载入



然后就是loadBeanDefinitions调用，首先得到Resource定位，具体由子类进行，此时由于FileSystemXmlApplicationContext不是ClassPathXmlApplicationContext，所以configResources返回null，进行getConfigLocation载入。



reader.loadBeanDefinitions(configLocations)在AbstractBeanDefinitionReader中，然后进行资源的定位以及载入；

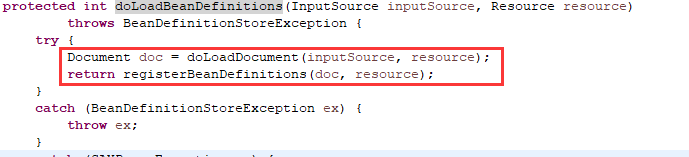




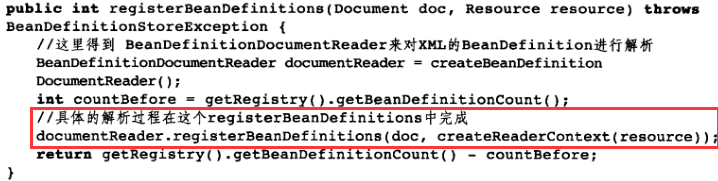
上面已经分析过，得到resource之后，进行真正的载入过程，此时的loadBeanDefinitions(resource)在XmlBeanDefinitionReader中，

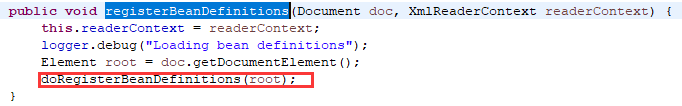


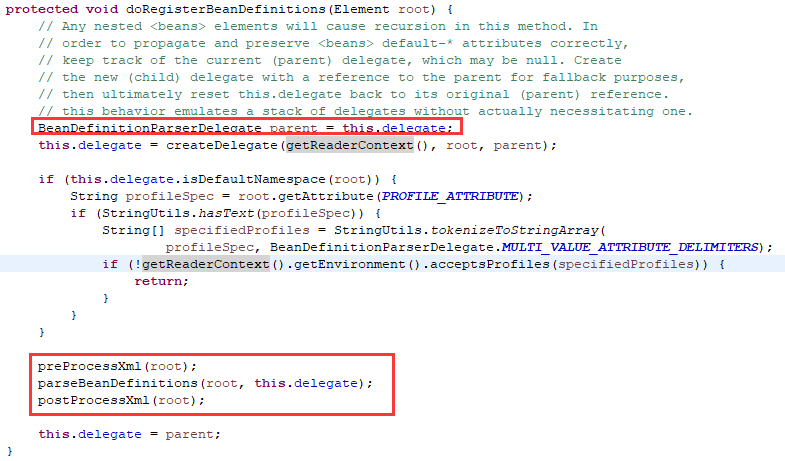
得到输入流之后，就可以采用doLoadBeanDefinitions方法进行xml文件解析

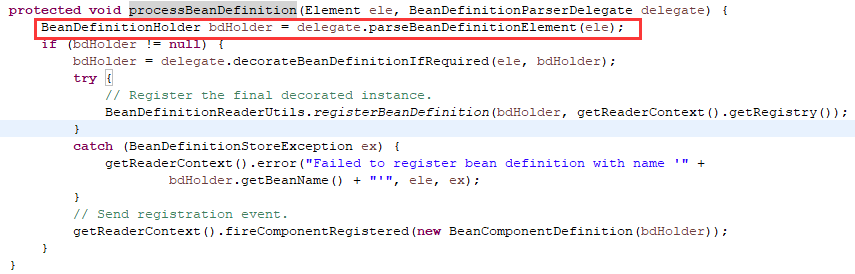


获取Document对象，采用SAX解析xml；Spring的BeanDefinition是怎么样按照Spring的Bean语义要求进行解析并转化为内部数据结构的，这个过程由registerBeanDefinitions完成；





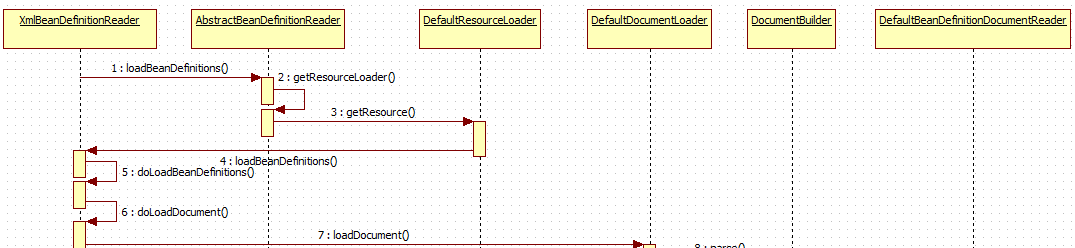
parseBeanDefinition到下面的方法

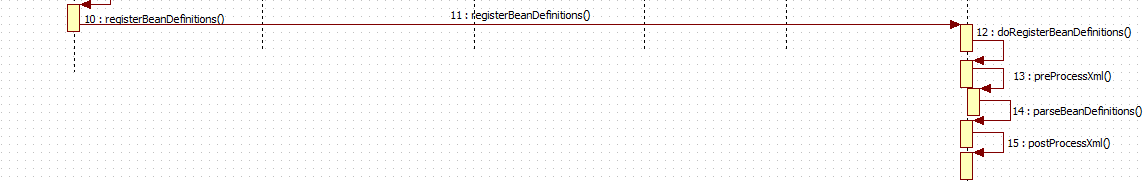


创建一个BeanDefinitionDocumentReader，默认是DefaultBeanDefinitionDocumentReader，使用BeanDefinitionParserDelegate进行解析Document对象，解析后的结果



放在BeanDefinition中，并将beanName,aliase作为参数保存到BeanDefinitionHolder中。





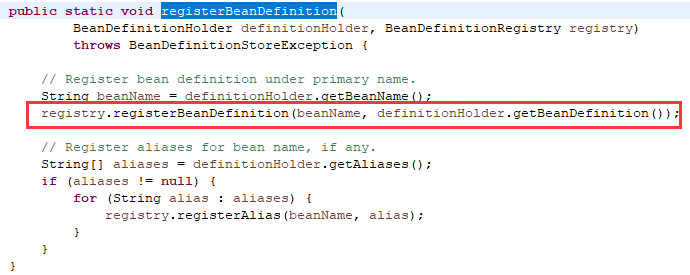
一层一层调用解析，这样xml中定义的BeanDefinition就被载入到IoC容器中了，并在容器中建立了数据的映射。这个时候容器还没有完全起作用，要完全发挥容器作用，还需要进行注册

**2.2.3 BeanDefinition在IoC中的注册**

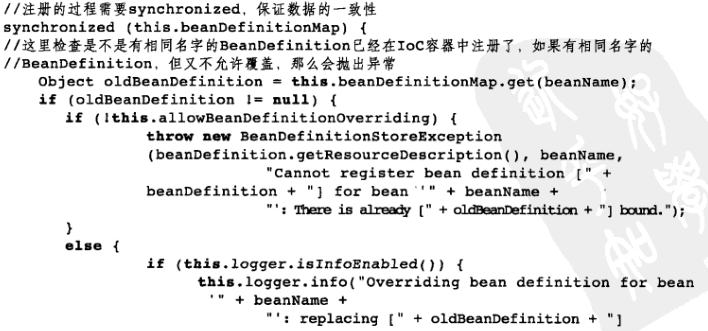
载入和解析过程，已经将用户定义的BeanDefinition信息在IoC容器中建立起了自己的数据结构，但是还需要进行注册才能使用。在DefaultListableBeanFactory中，通过一个ConcurrentHashMap来持有载入的BeanDefinition

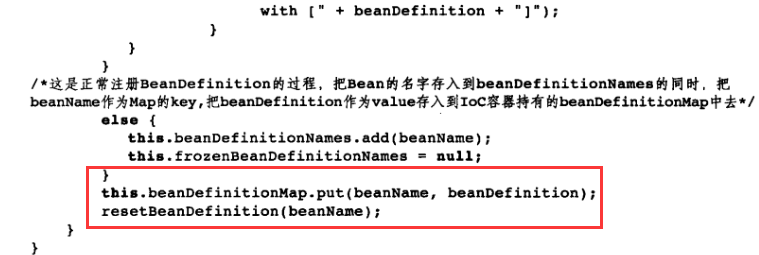


在xmlBeanDefinitionReader传入的DefaultListableBeanFactory，就是设定AbstractBeanDefinitionReader中的registry为DefaultListableBeanFactory。因此在DefaultBeanDefinitionDocumentReader的processBeanDefinition方法中BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry());就是用来注册BeanDefinition；



调用的是DefaultListableBeanFactory的registerBeanDefinition(String, BeanDefinition)方法

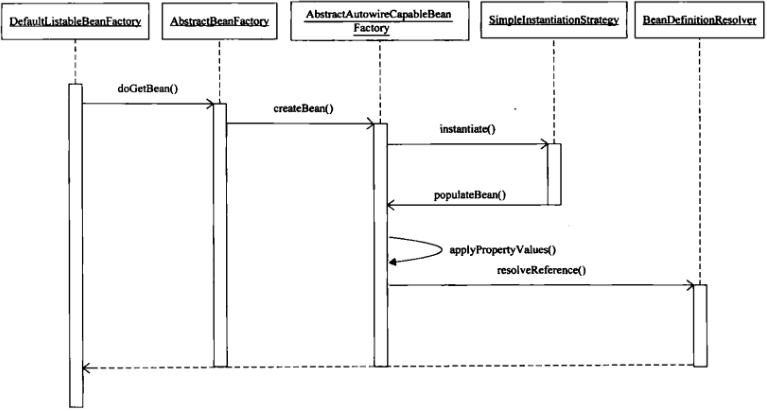




注册之后，BeanDefinition就可以被使用了。

**2.3 依赖注入**

依赖注入过程是用户第一次调用getBean时触发的，例外就是lazy-init属性的预实例化，后面会讲。GetBean调用完之后会触发doGetBean；就是用于依赖注入；getBean-dogetBean-createbean；



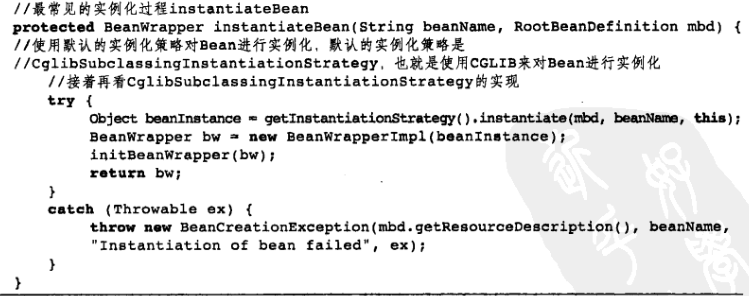
AbstractBeanFactory中的dogetBean方法会调用AbstractAutowireCapableBeanFactory的CreateBean方法；该方法生成了需要的bean，还对bean初始化进行了处理，定义后置处理器，定义init-method属性等；createBean中调用了doCreateBean方法



该方法中主要就是createBeanInstance和populateBean方法，createBeanInstance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。



使用不同的方式进行bean的实例化



使用cglib进行bean实例化，cglib是一个常用的字节码生成器的类库，提供了生成和转换字节码的功能。Cglib如何生成bean对象需要看simpleInstantiationStrategy。提供了两种实例化对象的方法，一种是BeanUtils，使用JVM的反射功能，一种使用cglib生成



实例化对象后，需要为依赖关系进行处理，即使用populateBean方法，在AbstractAutowireCapableBeanFactory中；

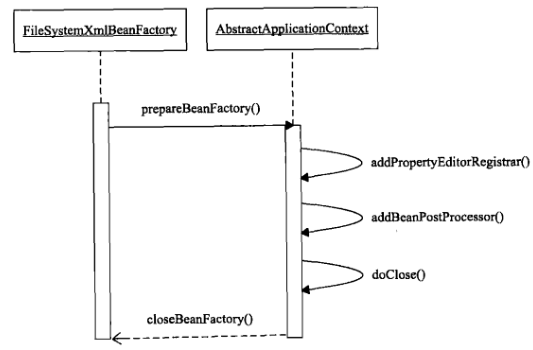
|  |
| --- |
| **protected** **void** populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {  **获取BeanDefinition的property属性值**  PropertyValues pvs = mbd.getPropertyValues();  **if** (bw == **null**) {  **if** (!pvs.isEmpty()) {  **throw** **new** BeanCreationException(  mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");  }  **else** {  // Skip property population phase for null instance.  **return**;  }  }  **boolean** continueWithPropertyPopulation = **true**;  **开始依赖注入过程，先处理autowired的注入**  **if** (mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_NAME*** ||  mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_TYPE***) {  MutablePropertyValues newPvs = **new** MutablePropertyValues(pvs);  **对autowired注入的处理，可以根据名字或者类型来完成bean注入**  **if** (mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_NAME***) {  autowireByName(beanName, mbd, bw, newPvs);  }  **if** (mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_TYPE***) {  autowireByType(beanName, mbd, bw, newPvs);  }  pvs = newPvs;  }  **boolean** hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();  **boolean** needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.***DEPENDENCY\_CHECK\_NONE***);  **if** (hasInstAwareBpps || needsDepCheck) {  PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);  **if** (hasInstAwareBpps) {  **for** (BeanPostProcessor bp : getBeanPostProcessors()) {  **if** (bp **instanceof** InstantiationAwareBeanPostProcessor) {  InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  **if** (pvs == **null**) {  **return**;  }  }  }  }  **if** (needsDepCheck) {  checkDependencies(beanName, mbd, filteredPds, pvs);  }  }  **然后对属性进行注入**  applyPropertyValues(beanName, mbd, bw, pvs);  } |
| **通过applyPropertyvalues了解具体的对属性行进行解析然后注入的过程**  **protected** **void** applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {  **if** (pvs == **null** || pvs.isEmpty()) {  **return**;  }  MutablePropertyValues mpvs = **null**;  List<PropertyValue> original;  **if** (System.*getSecurityManager*() != **null**) {  **if** (bw **instanceof** BeanWrapperImpl) {  ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());  }  }  **if** (pvs **instanceof** MutablePropertyValues) {  mpvs = (MutablePropertyValues) pvs;  **if** (mpvs.isConverted()) {  // Shortcut: use the pre-converted values as-is.  **try** {  bw.setPropertyValues(mpvs);  **return**;  }  **catch** (BeansException ex) {  **throw** **new** BeanCreationException(  mbd.getResourceDescription(), beanName, "Error setting property values", ex);  }  }  original = mpvs.getPropertyValueList();  }  **else** {  original = Arrays.*asList*(pvs.getPropertyValues());  }  TypeConverter converter = getCustomTypeConverter();  **if** (converter == **null**) {  converter = bw;  }    **这个BeanDefinitionValueResolver对BeanDefinition的解析是在这个valueResolver中完成的**  BeanDefinitionValueResolver valueResolver = **new** BeanDefinitionValueResolver(**this**, beanName, mbd, converter);  **为解析之后的属性值创建一个副本，副本数据将会被注入到bean中**  // Create a deep copy, resolving any references for values.  List<PropertyValue> deepCopy = **new** ArrayList<PropertyValue>(original.size());  **boolean** resolveNecessary = **false**;  **for** (PropertyValue pv : original) {  **if** (pv.isConverted()) {  deepCopy.add(pv);  }  **else** {  String propertyName = pv.getName();  Object originalValue = pv.getValue();  **根据需要解析的内容，originalValue和propertyName判断是reference还是list还是其他集合**  Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);  Object convertedValue = resolvedValue;  **boolean** convertible = bw.isWritableProperty(propertyName) &&  !PropertyAccessorUtils.*isNestedOrIndexedProperty*(propertyName);  **if** (convertible) {  convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);  }  // Possibly store converted value in merged bean definition,  // in order to avoid re-conversion for every created bean instance.  **if** (resolvedValue == originalValue) {  **if** (convertible) {  pv.setConvertedValue(convertedValue);  }  deepCopy.add(pv);  }  **else** **if** (convertible && originalValue **instanceof** TypedStringValue &&  !((TypedStringValue) originalValue).isDynamic() &&  !(convertedValue **instanceof** Collection || ObjectUtils.*isArray*(convertedValue))) {  pv.setConvertedValue(convertedValue);  deepCopy.add(pv);  }  **else** {  resolveNecessary = **true**;  deepCopy.add(**new** PropertyValue(pv, convertedValue));  }  }  }  **if** (mpvs != **null** && !resolveNecessary) {  mpvs.setConverted();  }  **这里是依赖注入发生的地方，会在beanWrapperImpl中完成**  // Set our (possibly massaged) deep copy.  **try** {  bw.setPropertyValues(**new** MutablePropertyValues(deepCopy));  }  **catch** (BeansException ex) {  **throw** **new** BeanCreationException(  mbd.getResourceDescription(), beanName, "Error setting property values", ex);  }  } |

大致过程：第一个调用getBean的时候，getBean会调用AbstractBeanFactory的createBean方法；createBean方法会调用doCreateBean方法得到一个Bean的引用；doCreateBean方法中定义了一个BeanWrapper用来持有创建出来bean对象，主要有两个过程，一是实例化bean，二是设置bean的依赖关系；createInstance方法中有两种方法对bean进行实例化，通过BeanUtils，使用jvm反射，一种是使用cglib生成；实例化对象之后，就需要设置bean的依赖关系；populateBean方法中先取得载入解析过程的property的值；先处理autowire的注入，根据bean的名字，类型等；然后处理property的注入；调用applyPropertyValues方法；方法中定义了一个BeanDefinitionValueResolver，设定一个需要注入的属性值的副本deepcopy，遍历属性值，使用解析器BeanDefinitionValueResolver进行属性值的解析；然后添加到deepcopy中。

上述的过程是为解析准备条件，真正把bean对象设置到另外一个依赖bean是在bw.setPropertyValues(new MutablePropertyValues(deepCopy));具体是在子类中实现了注入过程；通过几个地方的递归将，集合map，list和非集合的属性值注入到bean中，然后返回到BeanWapper，最终得到注入完成的Bean的实例。

**2.4 容器其他相关特性的设计与实现**

2.4.1 ApplicationContext和Bean的初始化及销毁





在容器要关闭的时候，也需要完成一系列的工作，这些工作在doClose方法中完成。在这个方法中，先发出容器关闭信号，然后将Bean逐个关闭，最后关闭容器自身。

容器的实现是通过IoC管理Bean的生命周期实现的。

Bean的生命周期：

①bean的初始化和实例化

②设置bean的属性，依赖注入

③应用可以通过IoC容器使用bean

④当容器关闭时，调用bean的销毁方法

Bean的初始化方法调用在initializeBean中实现：该方法在AbstractAutowireCapableBeanFactory类中

|  |
| --- |
| **protected** Object initializeBean(**final** String beanName, **final** Object bean, RootBeanDefinition mbd) {  **if** (System.*getSecurityManager*() != **null**) {  AccessController.*doPrivileged*(**new** PrivilegedAction<Object>() {  @Override  **public** Object run() {  invokeAwareMethods(beanName, bean);  **return** **null**;  }  }, getAccessControlContext());  }  **else** {  invokeAwareMethods(beanName, bean);  }  Object wrappedBean = bean;  **if** (mbd == **null** || !mbd.isSynthetic()) {  wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  }  **try** {  invokeInitMethods(beanName, wrappedBean, mbd);  }  **catch** (Throwable ex) {  **throw** **new** BeanCreationException(  (mbd != **null** ? mbd.getResourceDescription() : **null**),  beanName, "Invocation of init method failed", ex);  }  **if** (mbd == **null** || !mbd.isSynthetic()) {  wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  }  **return** wrappedBean;  } |

在调用invokeInitMethods方法进行初始化之前，会先调用一系列的aware接口实现，把相关的BeanClassLoader,BeanFactory注入到Bean中。如果有后置处理器BeanPostProcessor，就将其与bean进行关联，调用postProcessBeforeInitialization()方法；接着才会调用invokeInitMethods的调用，方法中还会调用afterPropertiesSet回调方法，最后判断是否配置了initMethod，如果有通过invokeCustomInitMethod调用，该方法中首先需要得到bean定义的initMethod，然后通过jdk反射得到Method对象，然后使用invoke调用该初始化方法。如果有后置处理器BeanPostProcessor，在调用postProcessAfterInitialization方法，接下来就可以使用Bean了；关闭过程由容器调用

|  |
| --- |
| **protected** **void** doClose() {  **boolean** actuallyClose;  **synchronized** (**this**.activeMonitor) {  actuallyClose = **this**.active && !**this**.closed;  **this**.closed = **true**;  }  **if** (actuallyClose) {  **if** (logger.isInfoEnabled()) {  logger.info("Closing " + **this**);  }  **try** {  // Publish shutdown event.  publishEvent(**new** ContextClosedEvent(**this**));  }  **catch** (Throwable ex) {  logger.warn("Exception thrown from ApplicationListener handling ContextClosedEvent", ex);  }  // Stop all Lifecycle beans, to avoid delays during individual destruction.  **try** {  getLifecycleProcessor().onClose();  }  **catch** (Throwable ex) {  logger.warn("Exception thrown from LifecycleProcessor on context close", ex);  }  // Destroy all cached singletons in the context's BeanFactory.  destroyBeans();  // Close the state of this context itself.  closeBeanFactory();  // Let subclasses do some final clean-up if they wish...  onClose();  **synchronized** (**this**.activeMonitor) {  **this**.active = **false**;  }  }  } |

容器关闭时，调用destroyBeans方法，删除单例bean，最终会调用destroyBean方法以及destroy方法；

1：DisposableBean的destroy()

在容器关闭时，如果Bean类有实现org.springframework.beans.factory.DisposableBean接口，则执行他的destroy()方法

2：Bean定义文件中定义destroy-method

在容器关闭时，可以在Bean定义文件中使用"destroy-method"属性设定方法名称，例如：

如果有以上设定的话，则进行至这个阶段时，就会执行destroy()方法。

**2.4.2 lazy-init属性和预实例化**

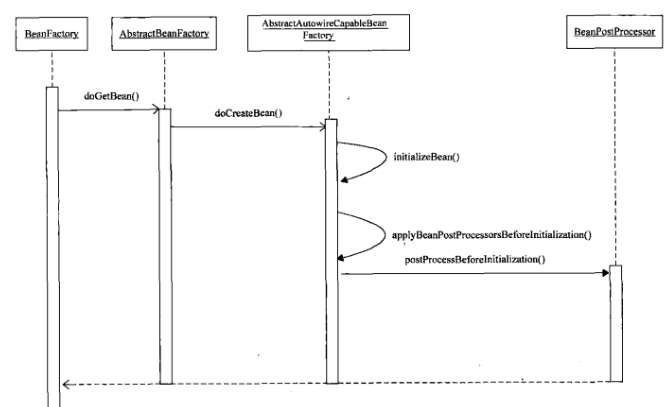
设置bean的lazy-init属性来控制预实例化过程，使得初始化容器时完成bean的依赖注入

Refresh方法中的finishBeanFactoryInitialization方法里面，会调用BeanFactory的preInstantiateSingleton方法，

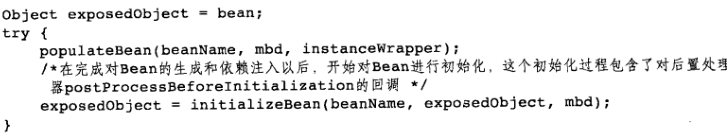


**2.4.3 BeanPostProcessor的实现**

**BeanPostProcessor**是使用IoC容器时经常遇到一个特性，这个Bean的后置处理器是一个监听器，它可以监听容器触发的事件。将它向IoC容器注册后，容器中管理bean具备了接收IoC容器事件回调的能力。BeanPostProcessor只需要通过设置一个具体的后置处理器来实现。具体的后置处理器需要实现BeanPostProcessor，实现两个接口方法postProcessBeforeInitialization和postProcessAfterInitialization，在初始化前后提供回调入口，参数是bean的实例化对象和bean的名字；

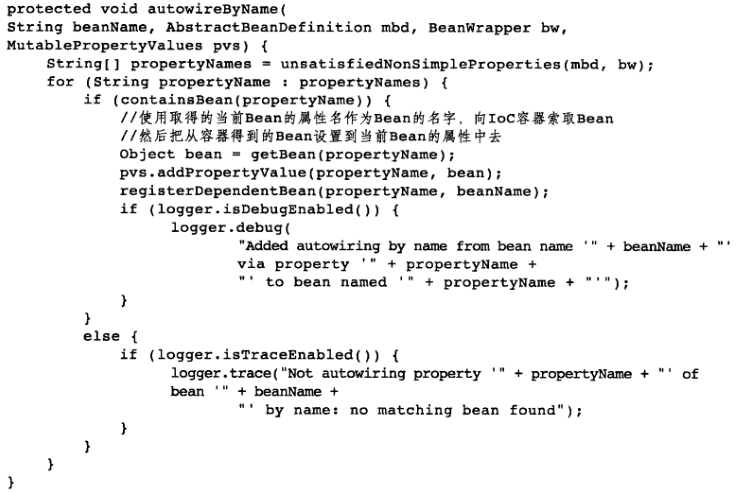


在依赖注入的时候，populateBean方法之后会调用initializeBean；initializeBean方法里面就会调用两个回调方法；



**2.4.4 autowire的实现**

populateBean的实现中，处理一般bean之前，先对autowire属性进行处理，如果当前的bean配置了auto\_by\_name/auto\_by\_type属性，那么调用相应的autowireByName和autowireByType方法。以autowireByName为例，首先获取需要注入bean的bean的属性名，然后遍历属性名，查找容器中是否含有该bean，如果有，就从容器中取得该bean实例，然后获取到属性名对应的bean，此时这个bean由于进行getBean方法，会触发进行依赖注入，这样得到的bean就是已经完成注入的依赖bean，再将这个依赖bean注入到当前bean的属性中。



**2.4.5 Bean对IoC容器的感知**

如果需要在bean中直接对IoC容器进行操作，需要在bean中设定对容器的感知，通过特定的aware接口来完成

BeanNameAware，可以在bean中得到它在容器中的实例名称

BeanFactoryAware，可以在bean中得到bean所在的Ioc容器，直接在bean中使用Ioc容器的服务

ApplicationContextAware，在bean中得到bean所在应用上下文；

在设定bean的属性之后，调用初始化回调方法之前，spring会调用aware接口中的setter方法，以ApplicationContextAware为例



setApplicationContext是一个回调函数，在bean中通过实现这个函数，可以在容器中回调该aware接口方法时使注入的ApplicationContext引用在bean中保存下来。



通过initializeBean就会调用postProcessBeforeInitialization，然后将相关的容器通过aware接口注入，这样bean在需要使用的时候，就可以通过aware进行使用IoC容器；