Spring源码分析

# Spring核心设计思想

依赖反转是软件设计中一个重要的思想，工厂模式应用的就是这种思想。解决了对象之间的依赖关系。

spring framework的核心:IOC容器的实现

依赖反转:依赖的对象的获得反转了.

依赖倒置,依赖反转,依赖注入,控制反转,都是一个意思.

在面向对象系统中,对象封装了数据和对数据的处理,对象的依赖关系常常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IOC容器来完成.解耦代码的同时提高代码的可测试性.

依赖控制反转的实现:在spring中IOC容器是实现这个模式的载体,他可以在对象生成或初始化时直接将数据注入到对象中.也可以通过将对象引用注入到对象数据域中的方式来注入对方法调用的依赖.这种依赖注入是可以递归的,对象被逐层注入.把对象的依赖关系有序的建立起来,简化了对象依赖关系的管理.简化了面向对象系统的复杂性.

如何反转对依赖的控制,把控制权从具体业务对象手中转交到平台或者框架中,是降低面向对象系统设计复杂性和提高面向对象系统可测试性的一个有效解决方案，他促进了IOC设计模式的发展,是IOC容器要解决的核心问题。

同时,也是产品化的IOC容器出现的推动力。

spring提供了统一管理依赖的平台框架,而不是每一个对象都有自己的依赖,每次使用某一个对象时,就要创建一个对象.简单的对象还不算麻烦,当一个对象内部有多个数据域需要初始化,或者依赖的它的对象可能多个,或者存在递归或者深层次的调用,或者需要不同的初始化数据时,这时候对象依赖的管理尤为重要.否则后期将无力维护,并且BUG的查找困难,新需要难以开展.

在多种面向对象语言中,都已经使用了这种开发技巧,也就是设计模式.JAVA,C++等.

通过外部配置文件来提高对组件关系的管理水平,并且如果耦合关系需要变动,并不需要重新修改和编译JAVA源代码,符合面向对象的开闭原则,并且提高提高组件系统设计的灵活性,同时,如果结合OSGI的使用特性,还可以提高应用的动态部署能力.另外,在最新版的jdk中已经提供了模块化编程的支持.

spring解决了什么问题？为什么要用spring？

从编程语言来说：在面向对象语言中，对象封装了数据和数据的处理，对象之间相互协作完成业务功能的实现，为了简化对象的依赖关系，简化面向对象的复杂性。诞生了软件设计的一个重要思想，依赖反转（也叫控制反转，依赖倒置）。那么如何实现这种思想是将对象的依赖通过外部框架或容器来完成注入。spring为该设计思想的产品化IOC容器。

从项目管理来说：面向对象之间复杂的依赖关系，导致测试难以开展，BUG的查找困难，后期无力维护，新需求更无法开展。如果不用IOC容器来开发，那么在各业务对象之间依赖前，肯定存在对象的初始化过程，有可能在不同业务下初始化过程不同，可能就会导致数据之间的错乱。

spring提供了统一管理依赖的平台框架,而不是每一个对象都有自己的依赖,每次使用某一个对象时,就要创建一个对象.简单的对象还不算麻烦,当一个对象内部有多个数据域需要初始化,或者依赖的它的对象可能多个,或者存在递归或者深层次的调用,或者需要不同的初始化数据时,这时候对象依赖的管理尤为重要.

另外spring通过外部配置文件来提高对组件关系的管理水平,并且如果耦合关系需要变动,并不需要重新修改和编译JAVA源代码,符合面向对象的开闭原则,并且提高提高组件系统设计的灵活性。

# Spring架构模块

由spring-beans、spring-core、spring-context和spring-expression（Spring Expression Language,SpEL）4个模块组成。

spring-beans和spring-core模块是Spring框架的核心模块，包含了控制反转（InversionofControl,IOC） 和依赖注入（DependencyInjection,DI）。BeanFactory接口是Spring框架中的核心接口，它是工厂模式的具体 实现。BeanFactory使用控制反转对应用程序的配置和依赖性规范与实际的应用程序代码进行了分离。但 BeanFactory容器实例化后并不会自动实例化Bean，只有当Bean被使用时BeanFactory容器才会对该Bean进行实例化与依赖关系的装配。

spring-context模块构架于核心模块之上，他扩展了BeanFactory，为她添加了Bean生命周期控制、框架事件 体系以及资源加载透明化等功能。此外该模块还提供了许多企业级支持，如邮件访问、远程访问、任务调度等， ApplicationContext是该模块的核心接口，她是BeanFactory的超类，与BeanFactory不同， ApplicationContext容器实例化后会自动对所有的单实例Bean进行实例化与依赖关系的装配，使之处于待用状态。

spring-expression模块是统一表达式语言（EL）的扩展模块，可以查询、管理运行中的对象，同时也方便的可以调用对象方法、操作数组、集合等。它的语法类似于传统EL，但提供了额外的功能，最出色的要数函数调用和简单字符串的模板函数。这种语言的特性是基于Spring产品的需求而设计，他可以非常方便地同SpringIOC进行交互。

# Spring IOC容器概述

Spring中主要两个容器系列:

1. 实现BeanFactory 接口的简单容器系列。 容器的基本功能.
2. 实现ApplicationContext 接口的应用上下文容器。 容器的高级形态存在.

应用上下文在简单容器的基础上,增加了面向框架的特性,同时对应用环境做了许多适配.

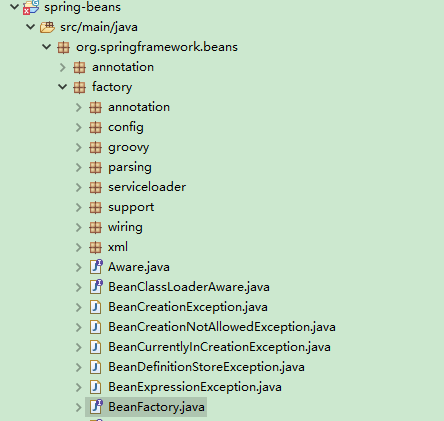
# BeanDefinition概述

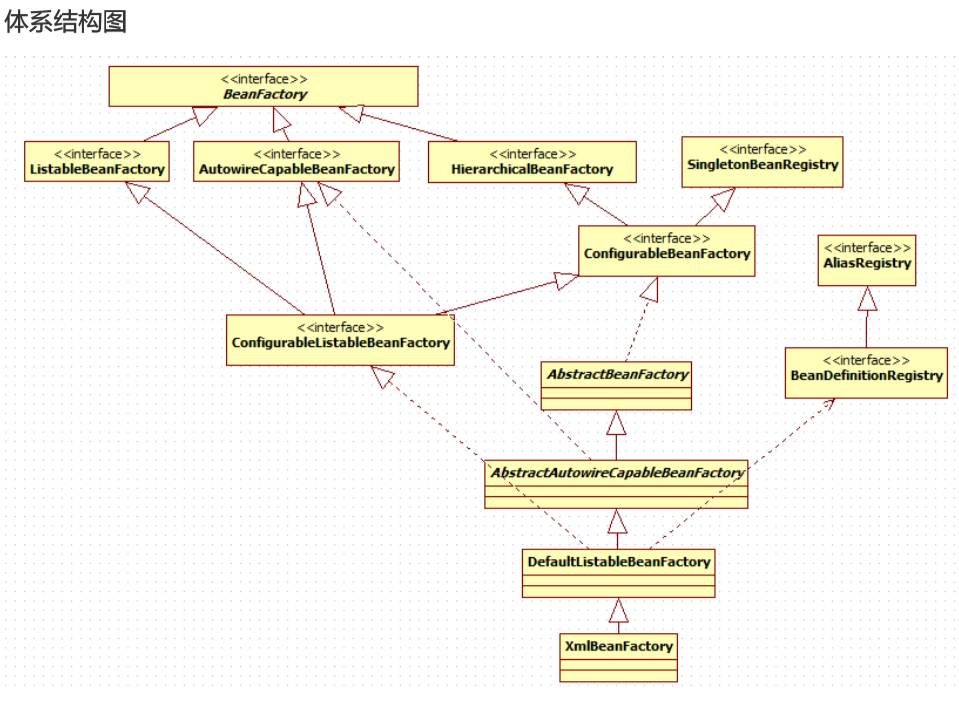
在spring提供的基本IOC容器的接口定义和实现的基础上,spring通过定义BeanDefinition这个数据结构来管理基于spring的应用中的各种对象以及它们的相互依赖关系.

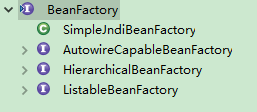
Beandefinition抽象了Bean的定义,是让容器起作用的主要数据类型.相当于bean的元数据。

ICO容器是用来管理对象依赖关系的,那么对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.

# BeanFactory继承体系结构







四级接口继承体系：

1. BeanFactory作为一个主接口不继承任何接口，暂且称为一级接口。

2. AutowireCapableBeanFactory、HierarchicalBeanFactory、ListableBeanFactory 3个子接口继承了它， 进行功能上的增强。这3个子接口称为二级接口。

3. ConfigurableBeanFactory可以被称为三级接口，对二级接口HierarchicalBeanFactory进行了再次增强，它还继承了另一个外来的接口SingletonBeanRegistry

4. ConfigurableListableBeanFactory是一个更强大的接口，继承了上述的所有接口，无所不包，称为四级接口。

这样的设计体系体现了设计模式原则里的接口隔离原则。

子接口简述：

AutowireCapableBeanFactory 自动装载有能力的beanfactory，提供工厂的装配功能。

HierarchicalBeanFactory 分级的beanfactory，提供父容器的访问功能，容器的嵌套。使得容器可以逐级访问的能力。

ListableBeanFactory 可列举的beanfactory，可批量获取bean。提供容器内bean实例的枚举功能.这边不会考虑父容器内的实例.

下面是继承关系的2个抽象类和2个实现类：

1. AbstractBeanFactory作为一个抽象类，实现了三级接口ConfigurableBeanFactory大部分功能。

2. AbstractAutowireCapableBeanFactory同样是抽象类，继承自AbstractBeanFactory，并额外实现了二级接口AutowireCapableBeanFactory。

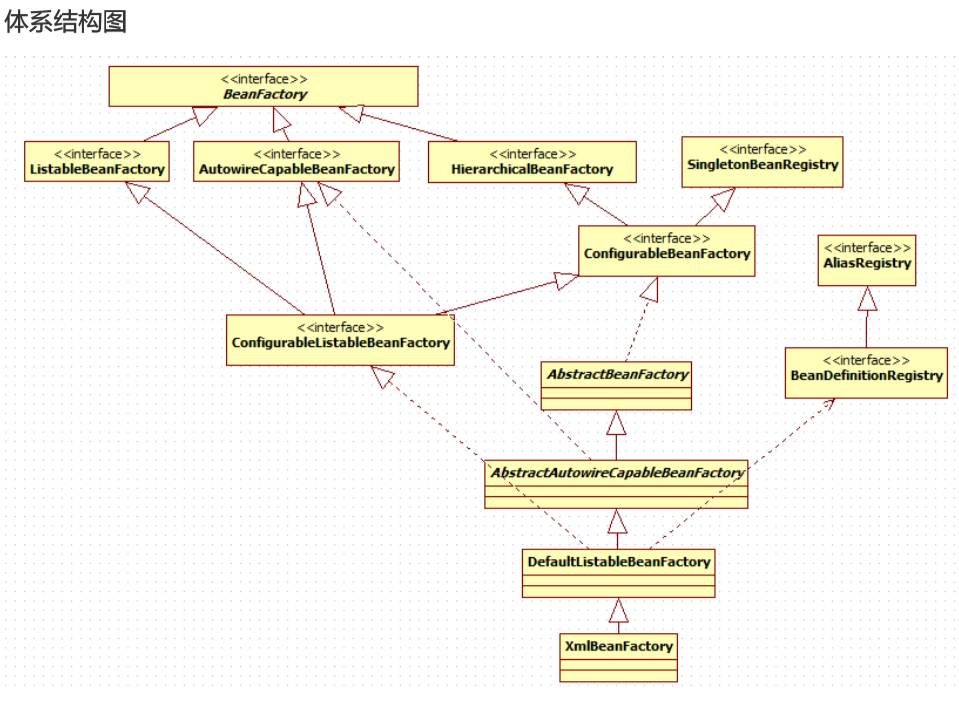
3. DefaultListableBeanFactory继承自AbstractAutowireCapableBeanFactory，实现了最强大的四级接口 ConfigurableListableBeanFactory，并实现了一个外来接口BeanDefinitionRegistry，它并非抽象类。

4. 最后是最强大的XmlBeanFactory，继承自DefaultListableBeanFactory，重写了一些功能，使自己更强大。屌丝容器。产品级的实现。

通过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.

# 重要接口说明



## ListableBeanFactory 可将Bean逐一列出的工厂



源码说明： 3个跟BeanDefinition有关的总体操作。包括BeanDefinition的总数、名字的集合、指定类型的名字的 集合。

这里指出，BeanDefinition是Spring中非常重要的一个类，每个BeanDefinition实例都包含一个类 在Spring工厂中所有属性。

2个getBeanNamesForType重载方法。根据指定类型（包括子类）获取其对应的所有Bean名字。

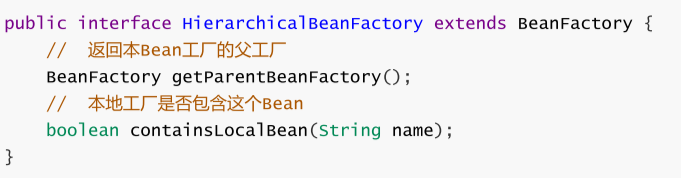
2个getBeansOfType重载方法。根据类型（包括子类）返回指定Bean名和Bean的Map。

正如这个工厂接口的名字所示，这个工厂接口最大的特点就是可以列出工厂可以生产的所有实例。当然，工厂并没有 直接提供返回所有实例的方法，也没这个必要。

它可以返回指定类型的所有的实例。而且你可以通过 getBeanDefinitionNames()得到工厂所有bean的名字，然后根据这些名字得到所有的Bean。

这个工厂接口扩展了 BeanFactory的功能，作为上文指出的BeanFactory二级接口，有9个独有的方法，扩展了跟BeanDefinition的功 能，提供了BeanDefinition、BeanName、注解有关的各种操作。它可以根据条件返回Bean的信息集合， 这就是它名字的由来——ListableBeanFactory

## HierarchicalBeanFactory 分层的Bean工厂



参数说明：

第一个方法返回本Bean工厂的父工厂。这个方法实现了工厂的分层。

第二个方法判断本地工厂是否包含这个Bean（忽略其他所有父工厂）。这也是分层思想的体现。

总结：

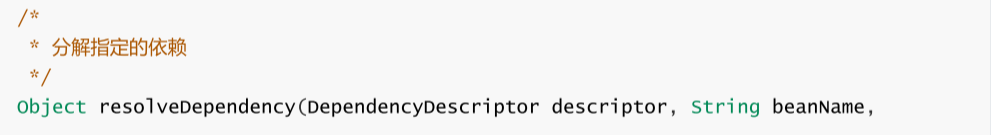
这个工厂接口非常简单，实现了Bean工厂的分层。这个工厂接口也是继承自BeanFacotory，也是一个二级接口，相对于父接口，它只扩展了一个重要的功能——工厂分层。

## AutowireCapableBeanFactory 自动装配的Bean工厂









源码说明：

1. 总共5个静态不可变常量来指明装配策略，其中一个常量被Spring3.0废弃、一个常量表示没有自动装配，另外 3个常量指明不同的装配策略——根据名称、根据类型、根据构造方法。

2. 8个跟自动装配有关的方法，实在是繁杂，具体的意义我们研究类的时候再分辨吧。

3. 2个执行BeanPostProcessors的方法。

4. 2个分解指定依赖的方法

总结：

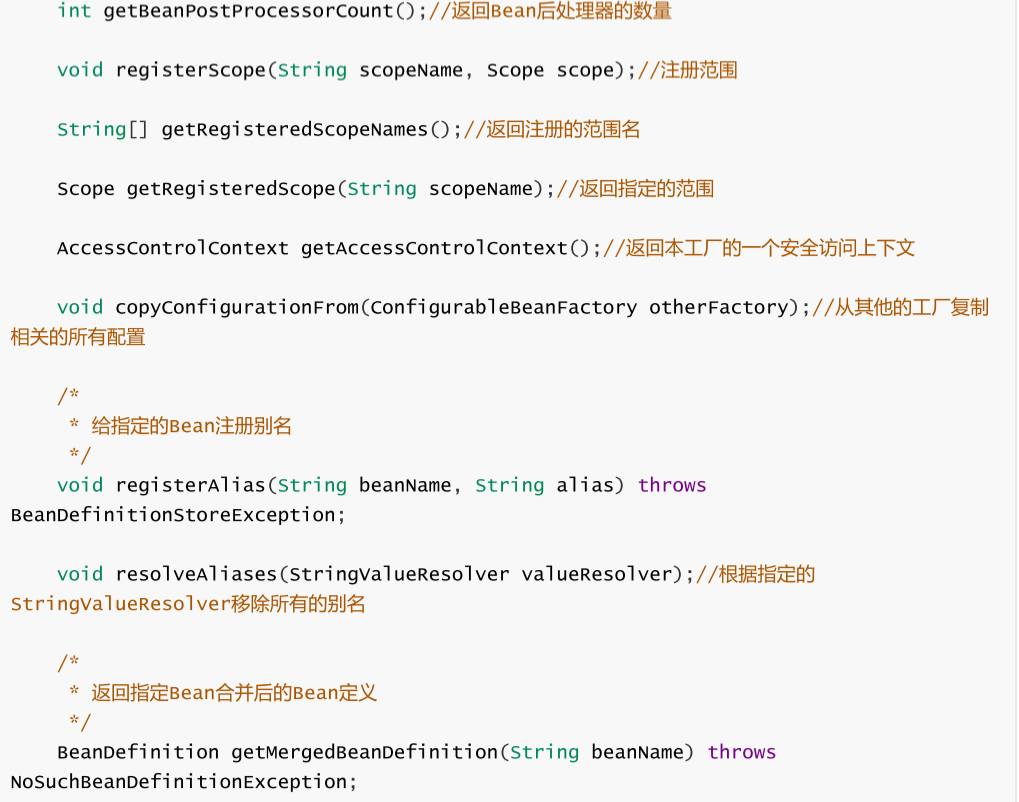
这个工厂接口继承自BeanFacotory，它扩展了自动装配的功能，根据类定义BeanDefinition装配Bean、执行前、 后处理器等。

## ConfigurableBeanFactory 复杂的配置Bean工厂



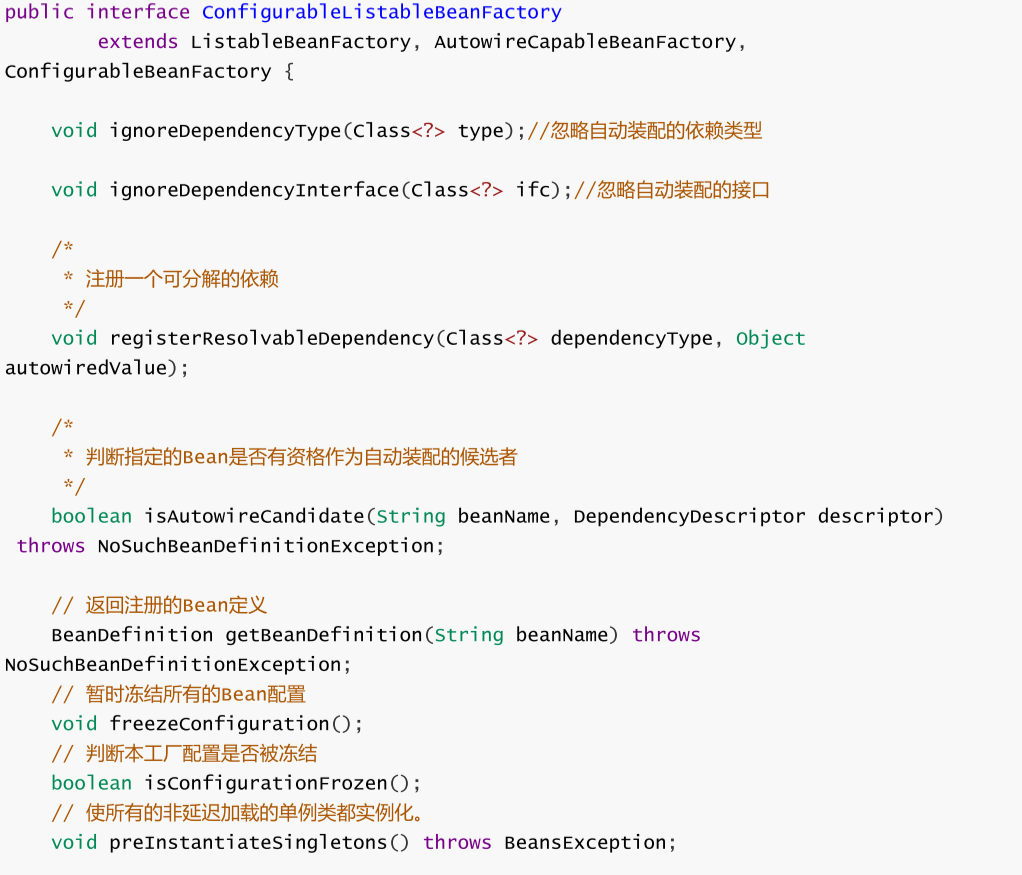








## ConfigurableListableBeanFactory BeanFactory的集大成者



源码说明：

1、2个忽略自动装配的的方法。

2、1个注册一个可分解依赖的方法。

3、1个判断指定的Bean是否有资格作为自动装配的候选者的方法。

4、1个根据指定bean名，返回注册的Bean定义的方法。

5、2个冻结所有的Bean配置相关的方法。

6、1个使所有的非延迟加载的单例类都实例化的方法。

总结： 工厂接口ConfigurableListableBeanFactory 同时继承了3个接口， ListableBeanFactory、 AutowireCapableBeanFactory 和 ConfigurableBeanFactory ，扩展之后，加上自有的这8个方法，这个 工厂接口总共有83个方法，实在是巨大到不行了。这个工厂接口的自有方法总体上只是对父类接口功能的补充，包含了BeanFactory体系目前的所有方法，可以说是接口的集大成者。

BeanDefinitionRegistry

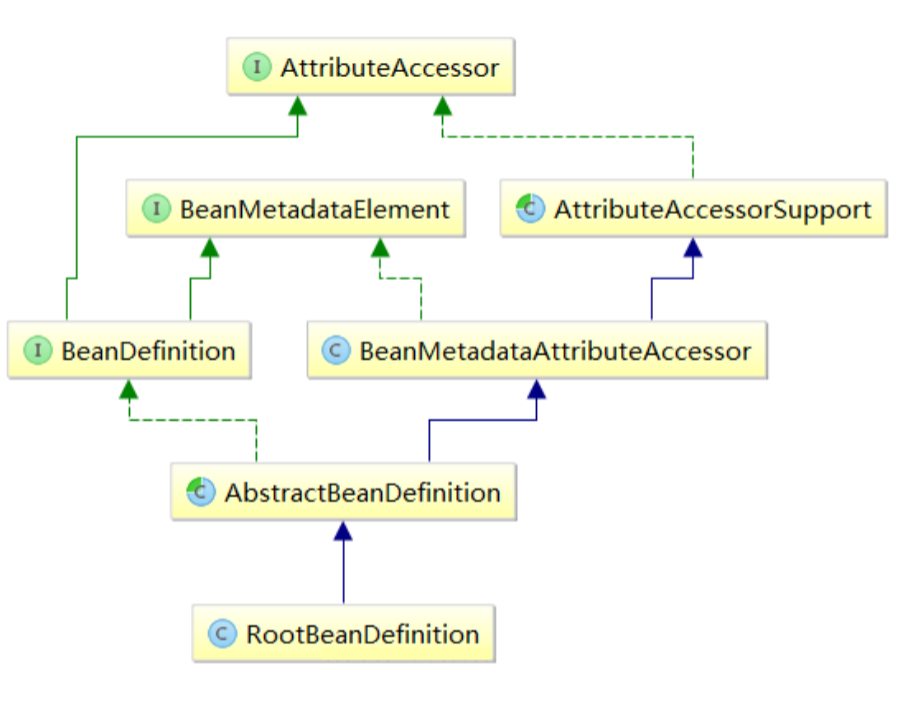
额外的接口,这个接口基本用来操作定义在工厂内部的BeanDefinition的。



BeanDefinition继承体系

体系结构图

SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Bean对象在Spring实现中是以BeanDefinition 来描述的，其继承体系如下：



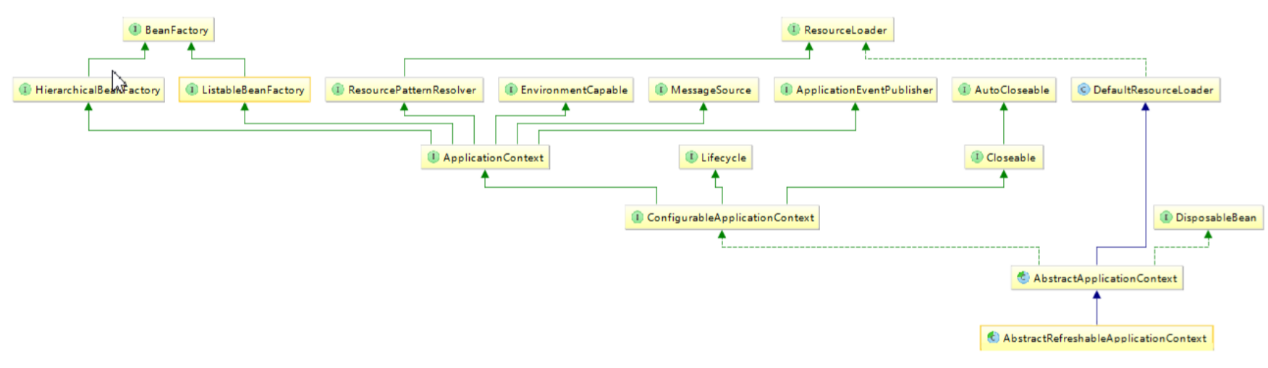
RootBeanDefinition

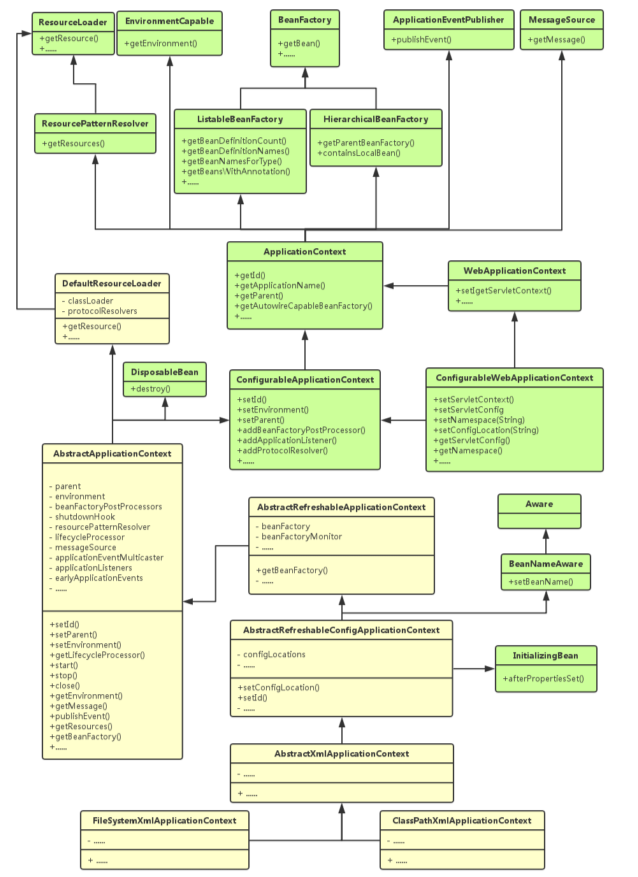
根bean定义表示在运行时在Spring BeanFactory中备份特定bean的合并bean定义。 它可能是从多个相互继承的原始bean定义创建的，通常注册为GenericBeanDefinitions.根bean定义本质上是运行时的“统一”bean定义视图。

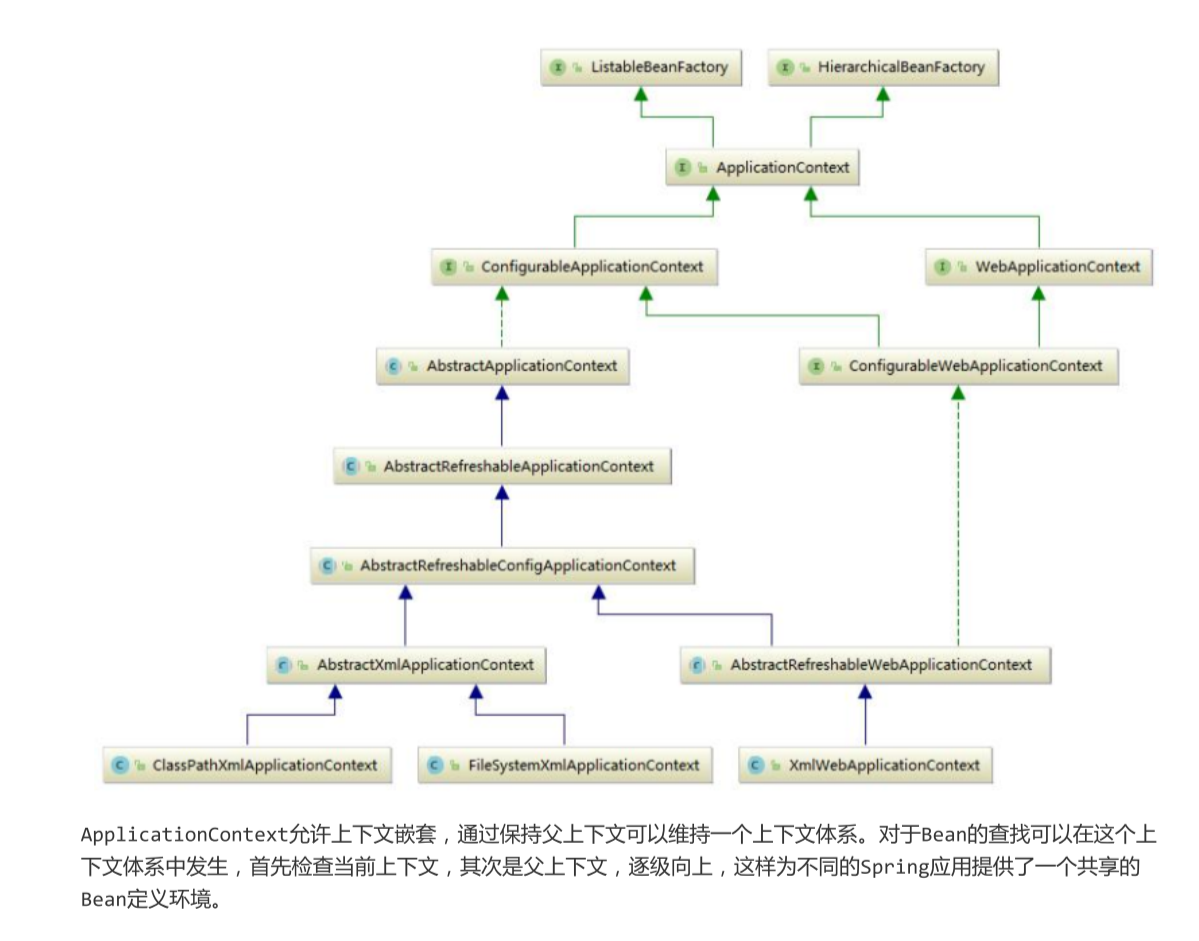
AbstractBeanDefinition

具体的，完善的BeanDefinition类的基类，分解GenericBeanDefinition，RootBeanDefinition和ChildBeanDefinition的公共属性。

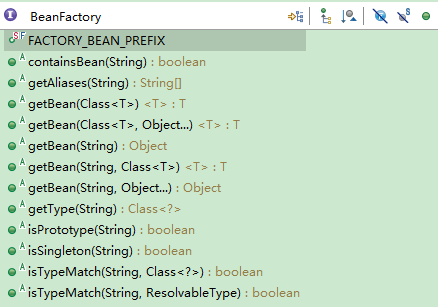
ApplicationContext继承体系



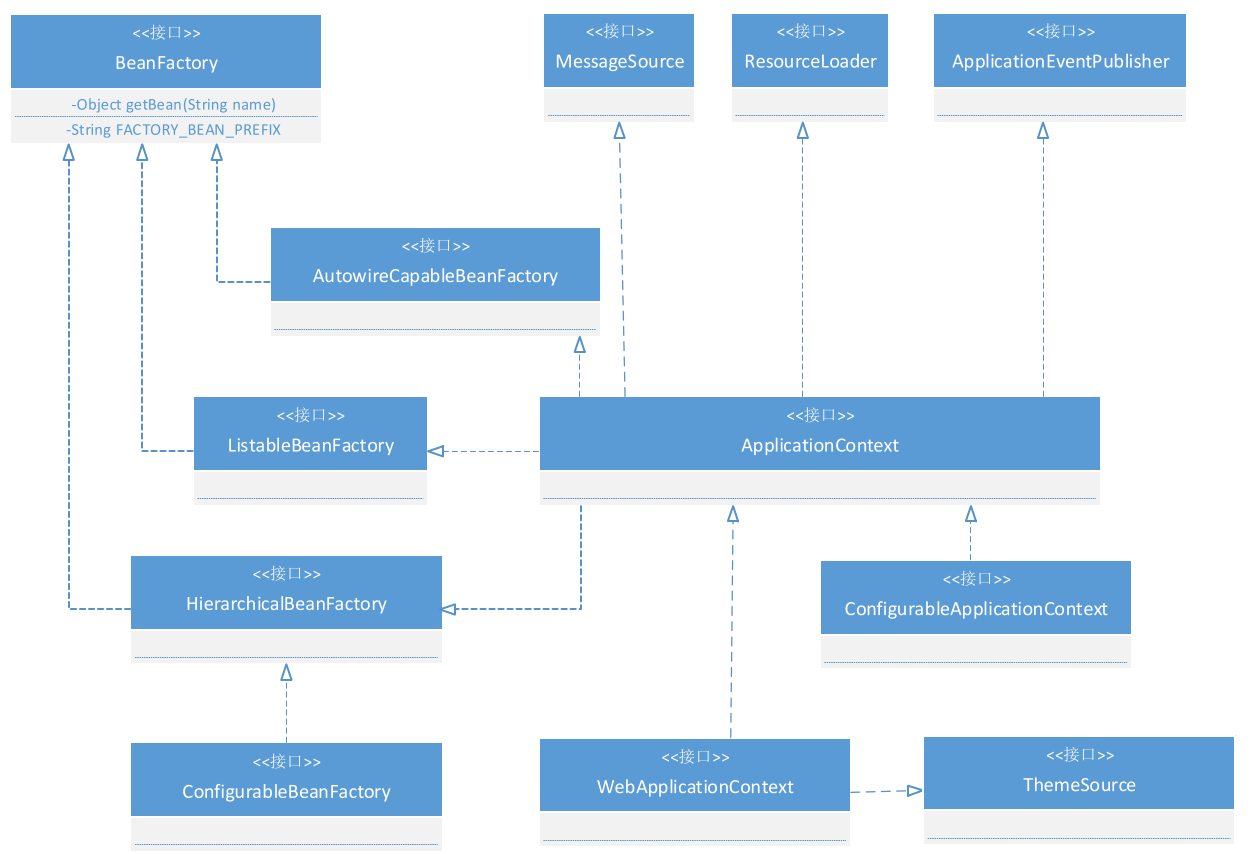




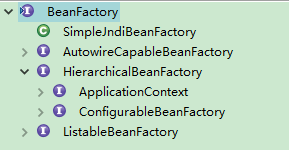
# BeanFactory接口的内部结构



# IOC容器两大设计路线



## 第一条设计路线：BeanFactory基本实现



BeanFactory 🡪 HierarchicalBeanFactory 🡪 ConfigurableBeanFactory

* HierarchicalBeanFactory 具备双亲IOC容器的管理功能.getParentBeanFactory
* ConfigurableBeanFactory 具备设置双亲IOC容器 setParentBeanFactory

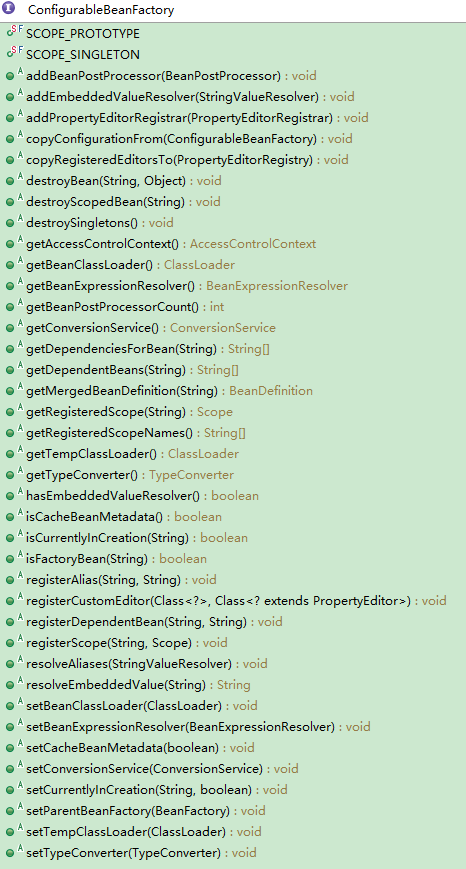
HierarchicalBeanFactory：分级工厂

在继承了BeanFactory基本接口之后，增加了getParentBeanFactory的接口功能，使BeanFactory具备了双亲IOC容器的配置功能。



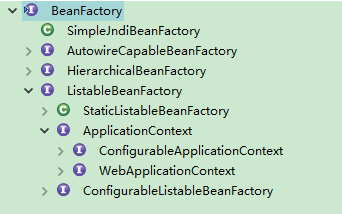
ConfigurableBeanFactory：可配置工厂

配置接口由大多数bean工厂实现。 除了org.springframework.beans.factory.BeanFactory接口中的bean factoryclient方法之外，还提供了配置bean工厂的功能。



设计路线: ConfigurableBeanFactory和HierarchicalBeanFactory的产品设计思想为 可配置,可管理.强调IOC容器的功能性实现.

## 第二条设计路线：ApplicationContext高级功能



从ListableBeanFactory 开始都在context包下。

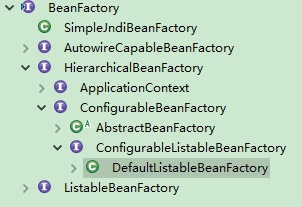
BeanFactory 🡪 ListableBeanFactory 🡪 ApplicationContext 🡪 WebApplicationContext

* ListableBeanFactory 可将Bean逐一列出的工厂，返回都是bean的集合。细化BeanFactory的接口功能. 其中defaultListableBeanFactory默认实现的中间产品.可以基于该产品进行扩展开发.
* ApplicationContext 强调对应用环境的适配,在框架中实际使用的是WebApplicationContext

ApplicationContext 通过继承BeanFactory的ListableBeanFactory HierarchicalBeanFactory AutowireCapableBeanFactory 来获取基本功能,还通过继承MessageSource, ApplicationEventPublisher, ResourcePatternResolver 来获取高级功能.

设计路线: ApplicationContext 可直接使用, 强调产品性

## DefaultListableBeanFactory具体IOC容器实现



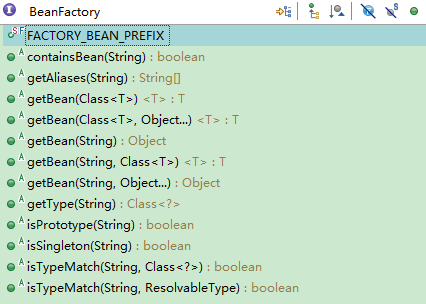
DefaultListableBeanFactory 实现了 ConfigurableBeanFactory 从而成为一个简单的IOC容器实现.

其他IOC容器,比如XmlBeanFactory 都是在DefaultListableBeanFactory 基础上扩展的.

XmlBeanFactory 已过时。从Spring 3.1开始，支持DefaultListableBeanFactory和XmlBeanDefinitionReader

对于高级需求，请考虑使用XmlBeanDefinitionReader中的DefaultListableBeanFactory。后者允许从多个XML资源中读取，并且在其实际的XML解析行为中具有高度可配置性。

# BeanFactory 的应用场景

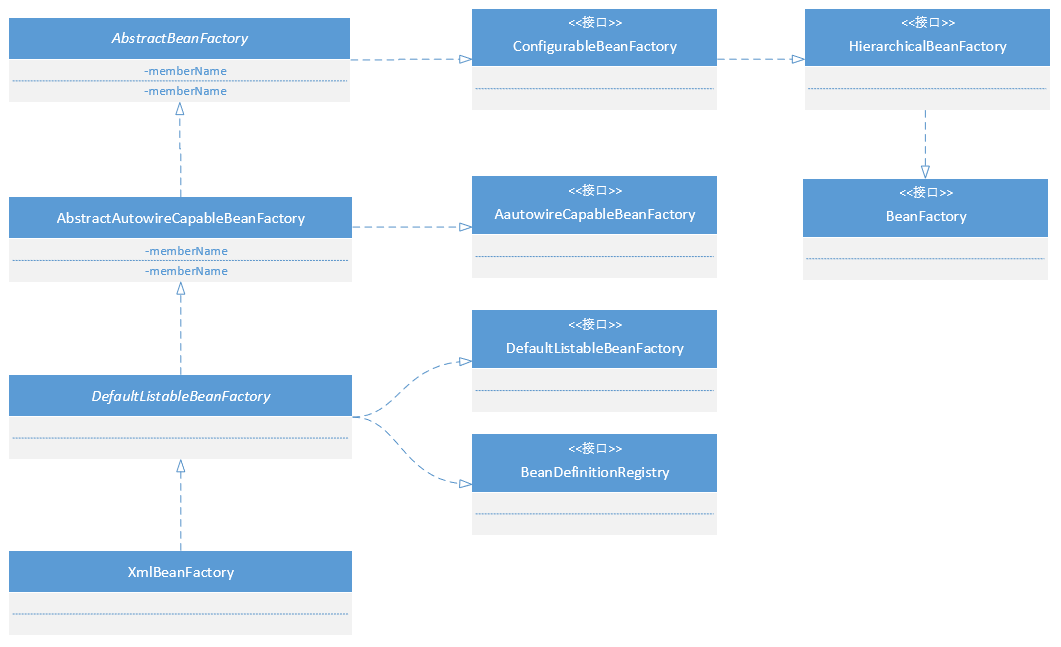


String FACTORY\_BEAN\_PREFIX = "&";

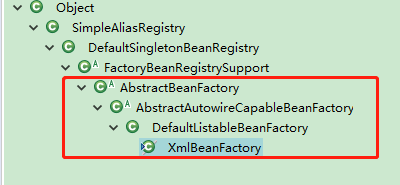
用来区分通过容器来获取BeanFactory产生的对象和获取BeanFactory本身

使用&时获取BeanFactory本身

# BeanFactory 容器设计原理

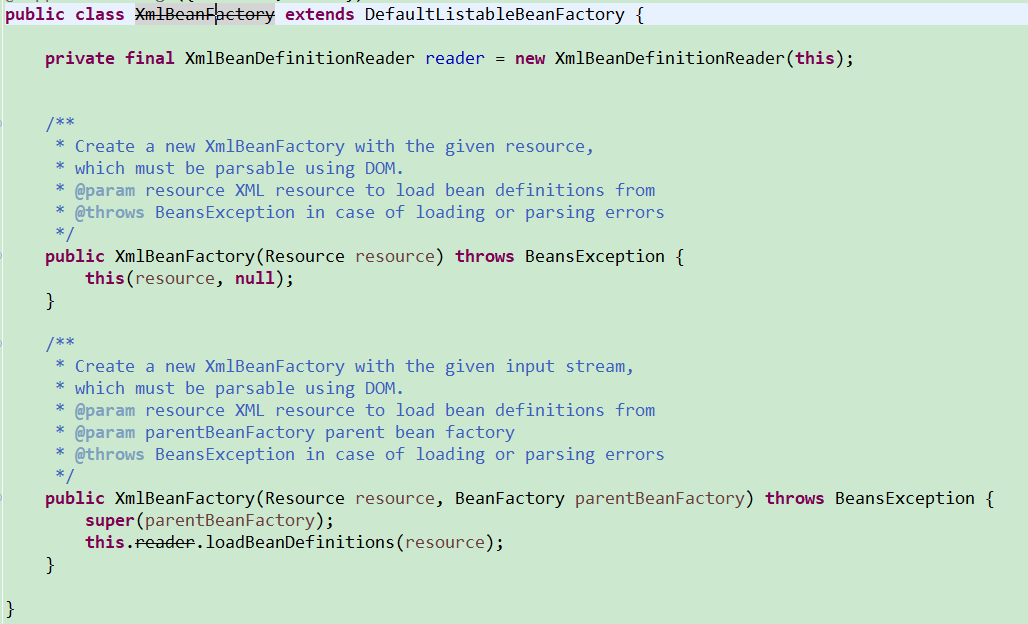


以XmlBeanFactory 为例进行说明



可以看出来, 在spring中 抽象类 为接口的下一层实现.

其中DefaultListableBeanFactory 作为一个默认的功能完整的IOC容器来使用。包含了基本IOC容器所具有的的重要功能。XmlBeanFactory通过继承DefaultListableBeanFactory，获取到了他的基本功能。并在其基础之上添加了读取解析XML新的功能。



如何实现XML读取功能?

通过以下接口和类来组合实现:

* XmlBeanDefinitionReader XML 文件处理
* Resource IO流操作封装接口

构造器初始化XmlBeanFactory时,传入Resource 接口

XmlBeanDefinitionReader初始化后调用reader.loadBeanDefinitions(resource);

载入beandefinitions ，完成IOC容器初始化。就可以直接使用该容器。

编程实现DefaultListableBeanFactory:

回调的核心:就是回调方将本身的this传递给对方.

|  |
| --- |
| ClassPathResource res = new ClassPathResource(“beans.xml”);  DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);  Reader.loadBeanDefinitions(res); |

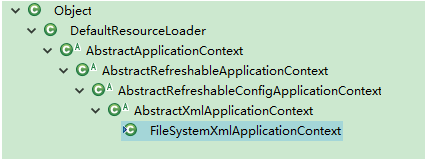
1. 创建IOC配置文件的抽象资源,这个抽象资源包含了BeanDefinition的定义信息
2. 创建一个BeanFactory , 这里使用DefaultListableBeanFactory
3. 创建一个载入BeanDefinition的读取器，这里使用XMLBeanDefinitionReader来载入XML文件形式的BeanDefinition，通过一个回调配置给BeanFactory
4. 从定义好的资源位置读入配置信息，具体的解析过程由XmlBeanDefinitionReader来完成。完成整个载入和注册Bean定义之后，需要的IOC容器就建立起来了。

# ApplicationContext 的应用场景

1. 支持不同的信息源，国际化的支持，多语言版本的应用提供服务。MessageSource。
2. 访问资源。ResourceLoader、Resource、ResourcePatternResolver。一般通过继承DefaultResourceLoader的子类，因为DefaultResourceLoader是AbstractApplicationContext的父类。
3. 支持应用事件。继承了接口ApplicationEventPublisher，在上下文中引入事件机制。这些事件和Bean的生命周期的结合为Bean的管理提供了便利。

# ApplicationContext容器的设计原理

以FileSystemXmlApplicationContext为例讲解

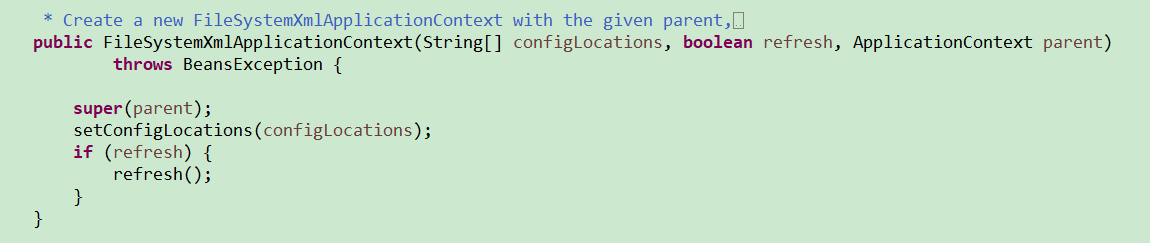


FileSystemXmlApplicationContext中ApplicationContext应用上下文的主要功能已经在FileSystemXmlApplicationContext的基类AbstractXmlApplicationContext中实现。所以作为一个具体的应用上下文，只需要实现和它自身设计相关的两个功能即可。

功能1：

容器的初始化和启动由以下方法来判定

这个refresh()过程会牵涉IOC容器启动的一系列复杂操作，对于不同的容器实现，这些操作都是类似的，因此在基类中将它们封装好。所以，我们在FileSystemXmlApplicationContext的设计中看到的只是一个简单的调用。



功能2：

怎么从文件系统中加载XML的Bean定义资源有关。

通过从文件系统中加载XML这个过程，可以为在文件系统中读取以XML形式存在的BeanDefinition做准备，因为不同的应用上下文实现对应着不同的读取BeanDefinition的方式，在FileSystemXmlApplication 的代码实现如下



通过调用这个方法，可以得到FileSystemResource的资源定位

# IOC容器的初始化过程

IOC容器的初始化由具体的产品化容器调用自己的父类AbstractApplicationContext中refresh()方法来启动，这个方法标志着IOC容器的正式启动。

这个启动包括BeanDefinition的Resource定位，BeanDefinition载入和注册BeanDefinition三个基本过程。

Spring将三个基本过程分开，并使用不同的模块来完成。通过这样的设计方式，可以让用户灵活的对三个过程进行裁剪或扩展，定义出最合适自己的IOC容器的初始化过程。

在IOC容器的初始化过程中，是不包含Bean依赖注入的实现。

在Spring IOC的设计中，Bean定义的载入和依赖注入是两个独立的过程。

依赖注入一般发生在应用第一次通过getBean向容器索取Bean的时候。

但有一个例外是，设置Bean定义信息中lazying属性。

## BeanDefinition的资源定位

这个定位指的是BeanDefinition的资源定位，它由ResourceLoader通过统一的Resource接口来完成，这个Resources对各种形式的BeanDefinition的使用都提供了统一的接口。比如说FlieSystemResource和ClassPathResource

## BeanDefinition的载入

用户定义好的Bean表示成IOC容器内部的数据结构，而这个容器内部的数据结构就是Beandefinition。

Beandefinition就是POJO对象在IOC容器中的抽象，通过这个Beandefinition定义的数据结构，使IOC容器能够方便地对POJO对象也就是Bean进行管理。

## Beandefinition的注册

向IOC容器中注册Beandefinition，这个过程是通过调用BeandefinitionRegistry接口的实现来完成的。把载入过程中解析得到的Beandefinition向IOC容器进行注册。在IOC容器内部将Beandefinition注入到一个HashMap中，IOC容器就是通过这个HashMap来持有这些Beandefinition数据的。

# Beandefinition的Resources定位

1. ClassPathResource

以编程方式使用DefaultListableBeanFactory时，首先定义一个Resource来定位容器使用的

Beandefinition。这时使用的是ClassPathResource，这意味着Spring会在类路径中寻找以文件形式存在的Beandefinition信息。

1. BeandefinitionReader

这里定义的Resource并不能由DefaultListableBeanFactory直接使用，spring通过BeandefinitionReader来对这些信息进行处理。

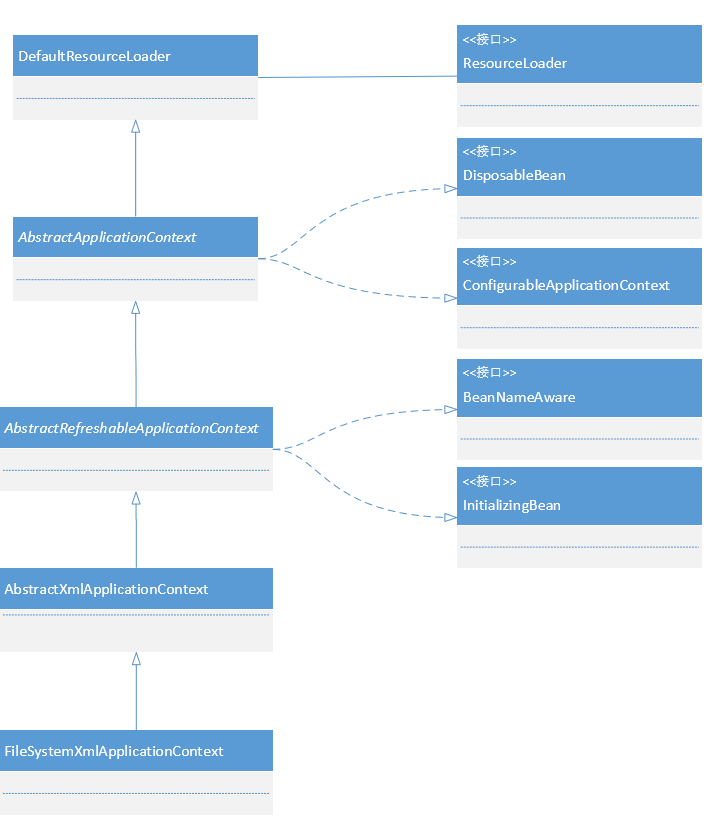
在Application中已经为我们提供了加载不同resource的实现Reader，而DefaultListableBeanFactory只是一个基本的IOC容器，需要为它配置特定的读取器 才能完成这些功能。

3. ResourceLoader

以FileSystemXmlApplicationContext为例，它通过继承AbstractApplicationContext，而AbstractApplicationContext继承了ResourceLoader，就具备了ResourceLoader读入以Resource定义的Beandefinition的能力。

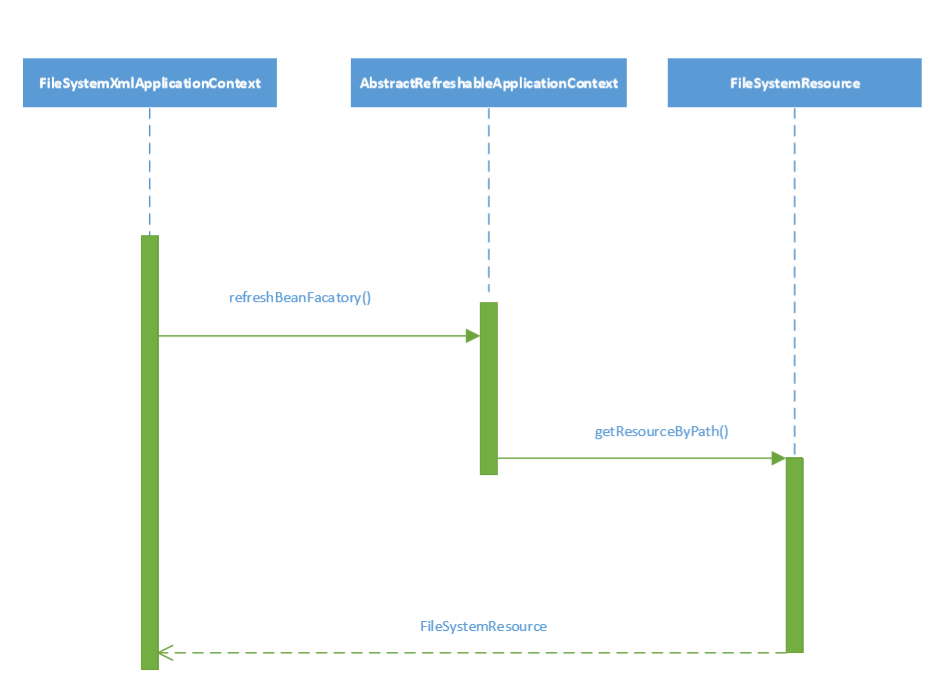
以此类推：

FileSystemXmlApplicationContext，ClassPathXmlApplicationContext以及XmlWebApplicationContext 分别提供不同放入Resource读入功能。

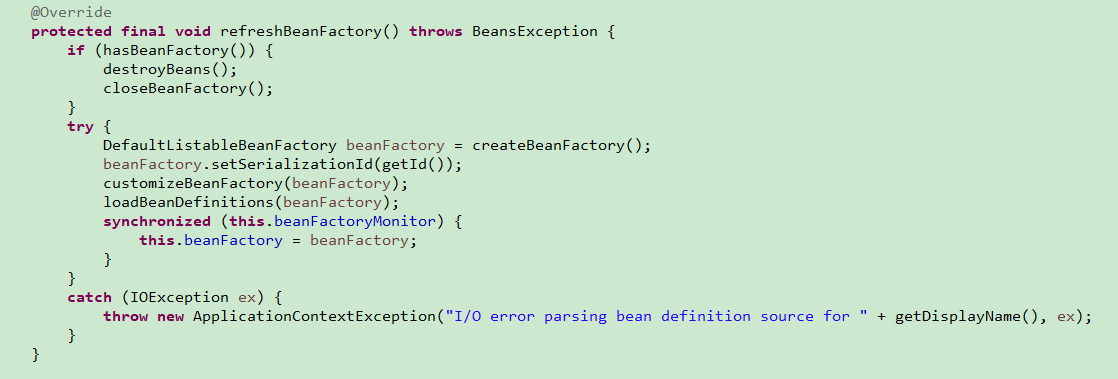


FileSystemXmlApplicationContext是在什么地方定义了BeanDefinition的BeanDefinitionReader,从而完成BeanDefinition信息的读入？

关于读入器的配置，可以在FileSystemXmlApplicationContext的基类AbstractRefreshableApplicationContext中看看它时怎么实现的。

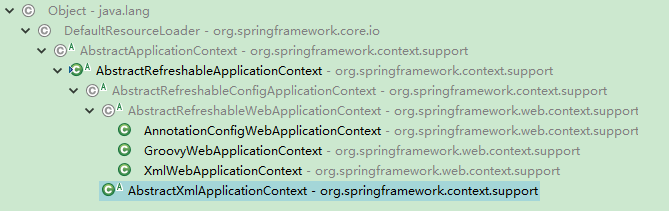


AbstractRefreshableApplicationContext的refreBeanFactory方法的实现，这个refrershBeanFactory被FileSystemXmlFactoryContext构造函数中的refresh调用。



通过createBeanFactory构建了一个IOC容器供ApplicationContext使用。这个容器就是我们前面提到的DefaultListableBeanFactory，同时，AbstractRefreshableApplicationContext启动了LoadBeanDefinitions来载入BeanDefinition。

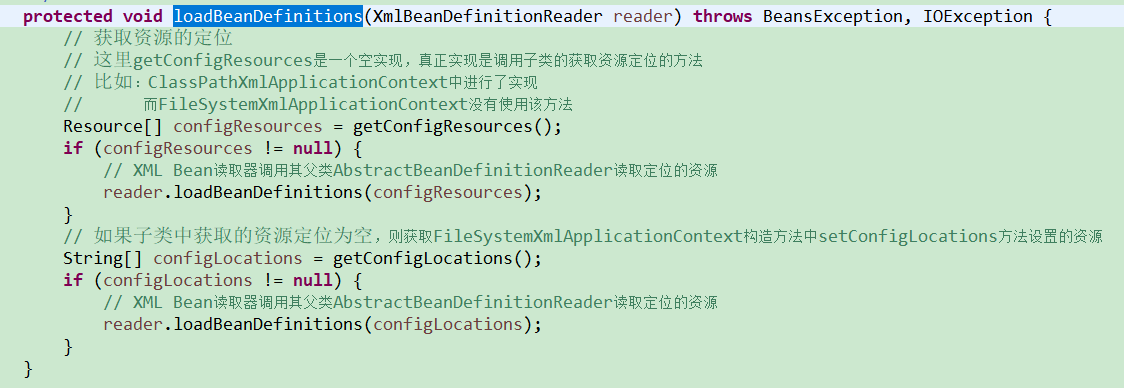
AbstractRefreshableApplicationContext的LoadBeanDefinitions是个抽象方法，具体有子类来实现。



可以看出抽象类AbstractXmlApplicationContext下有该方法的实现，



在该AbstractXmlApplicationContext的重写方法中有同名的处理函数



模板方法：

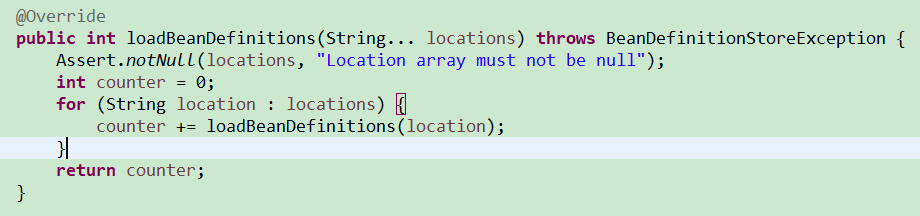
Resource[] configResources = getConfigResources();

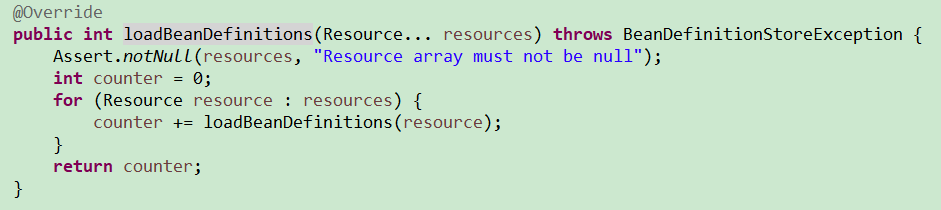
中的getConfigResources是本类AbstractXmlApplicationContext中的一个空实现，用来给子类实现的。子类实现该接口后，AbstractXmlApplicationContext调用，就可以加载到资源文件。

String[] configLocations = getConfigLocations();

中的getConfigLocations()是AbstractRefreshableConfigApplicationContext中的空实现。也是用来给子类实现的。子类实现该接口后，AbstractXmlApplicationContext就可以调用，就可以加载到资源文件。 因为AbstractXmlApplicationContext是AbstractRefreshableConfigApplicationContext的子类。

计算导入数量





具体的载入过程，是在XMLBeanDefinitionReader的基类中AbstractBeanDefinitionReader中可以看到这个载入过程的具体实现。代码如下



这里有两个重要的方法：

## Resource resource = resourceLoader.getResource(location);

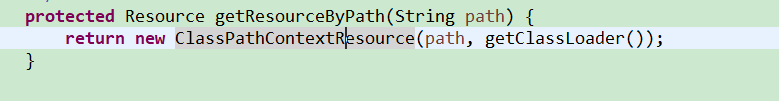
在AbstractBeanDefinitionReader的loadBeanDefinitions的方法中，Resource resource = resourceLoader.getResource(location);

是使用ResourceLoader接口的子类实现来完成的。

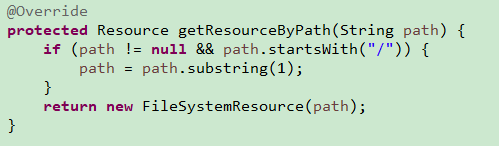
默认实现DefaultResourceLoader的代码实现如下



最后在异常中返回调用getResourceByPath(location);

默认实现是

一般在DefaultResourceLoader的子类中FileSystemXmlApplicationContext中的重写实现



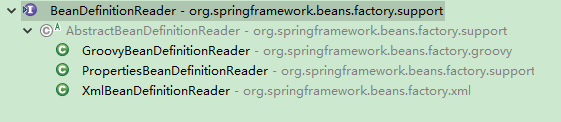
返回的是Resource接口，所以，根据不同需要可以返回不同的Resource子类

## int loadCount = loadBeanDefinitions(resources);

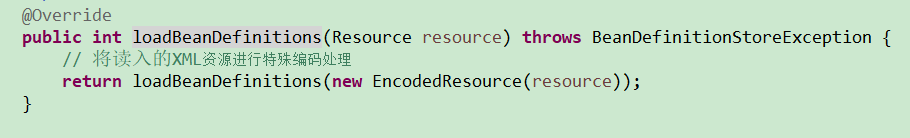
在上面这个方法中，会调用BeanDefinitionReader接口中的loadBeanDefinitions方法

C:\Users\roseonly\AppData\Local\Temp\1564740183(1).png

BeanDefinitionReader的子类实现有



在XmlBeanDefinitionReader实现中



上面方法调用下面



return doLoadBeanDefinitions(inputSource, encodedResource.getResource());

是具体的读取过程



完成定位。接下来就是载入和注册工作。

# Beandefinition的载入和解析

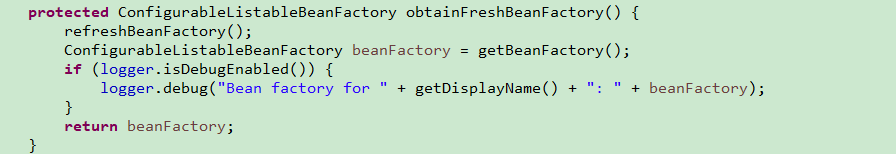
载入的过程就相当于把定义的BeanDefinition在IOC容器中转化成一个Spring内部表示的数据结构的过程。

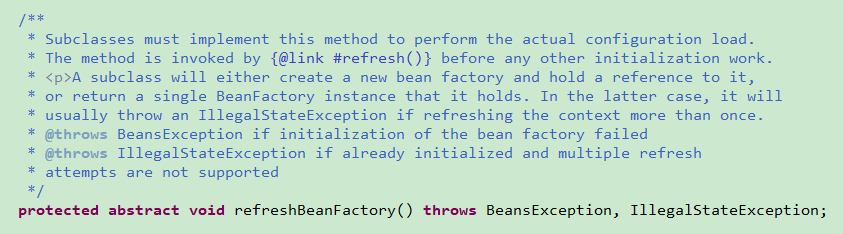
IOC容器对Bean的管理和依赖注入功能的实现，是通过对其持有的BeanDefinition进行各种相关的操作来完成的。

对容器的启动来说，refresh是一个重要的方法。该方法在AbstractApplicationContext类中找到，FilesystemXmlApplicationContext的基类。它详细描述了整个ApplicationContext的初始化过程。如BeanFactory的更新，MessageSource的PostProcess的注册。这里看起来更像是对ApplicationContext进行初始化的模板或执行提纲。

AbstractApplicationContext代码如下：





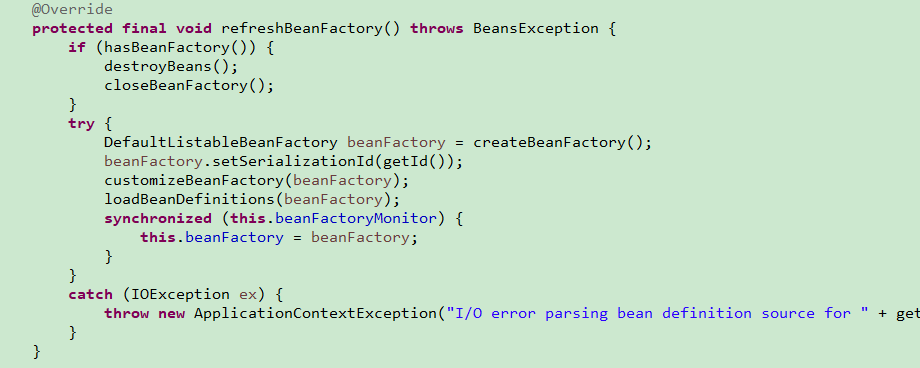


该抽象方法的中文注释说明：

子类必须实现此方法以执行实际的配置加载。该方法在任何其他初始化工作之前由refresh（）调用。

子类将创建一个新的bean工厂并保存对它的引用，或者返回它所拥有的单个BeanFactory实例。 在后一种情况下，如果多次刷新上下文，它通常会抛出IllegalStateException。

AbstractApplicationContext子类AbstractRefreshableApplicationContext的实现代码如下：

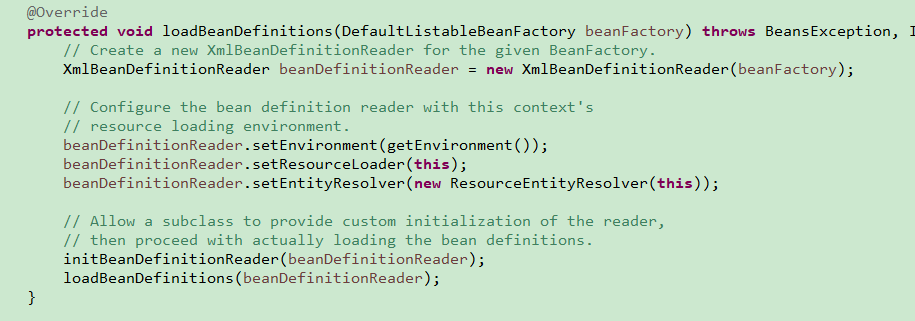


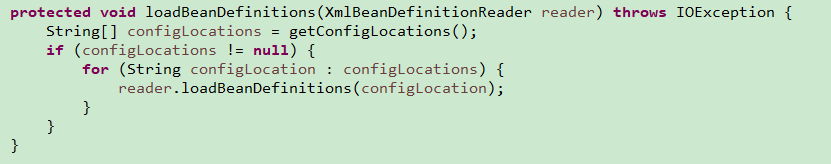
AbstractRefreshableApplicationContext中载入beandefintion接口

C:\Users\roseonly\AppData\Local\Temp\1562573934(1).png

AbstractRefreshableApplicationContext的子类

XmlWebApplicationContext的实现代码如下





AbstractBeanDefinitionReader的代码实现如下

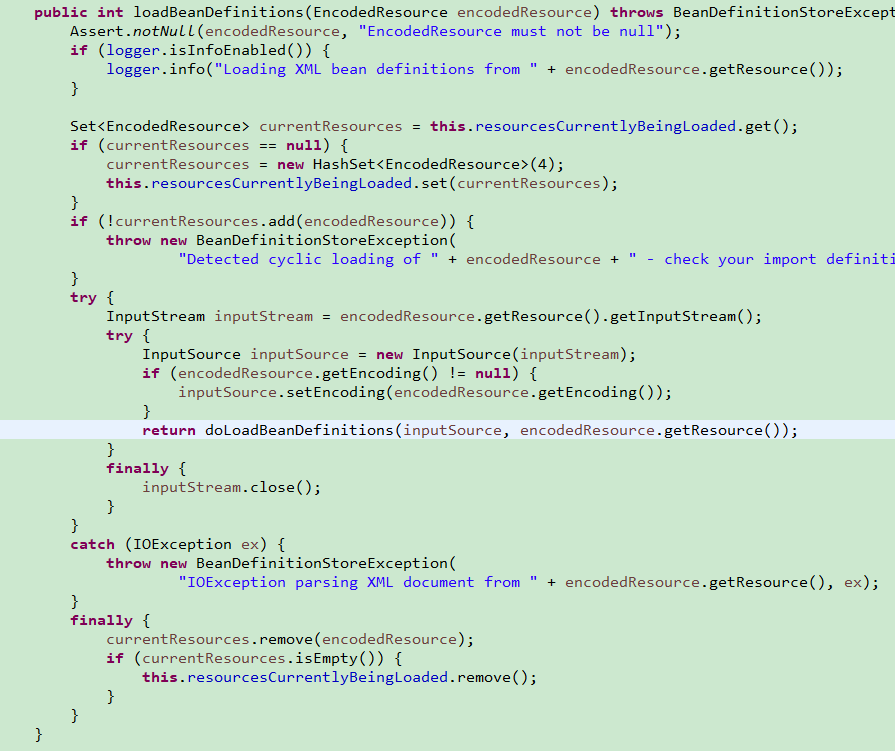


AbstractBeanDefinitionReader实现了BeanDefinitionReader接口的loadBeanDefinitions(Resource resource)方法



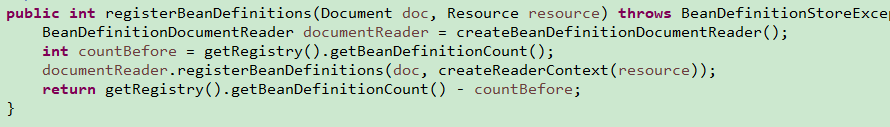
XmlBeanDefinitionReader重写了AbstractBeanDefinitionReader抽象类的loadBeanDefinitions(Resource resource)方法

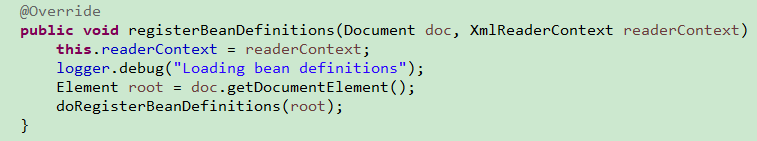
XmlBeanDefinitionReader的loadBeanDefinitions(Resource resource)方法实现：



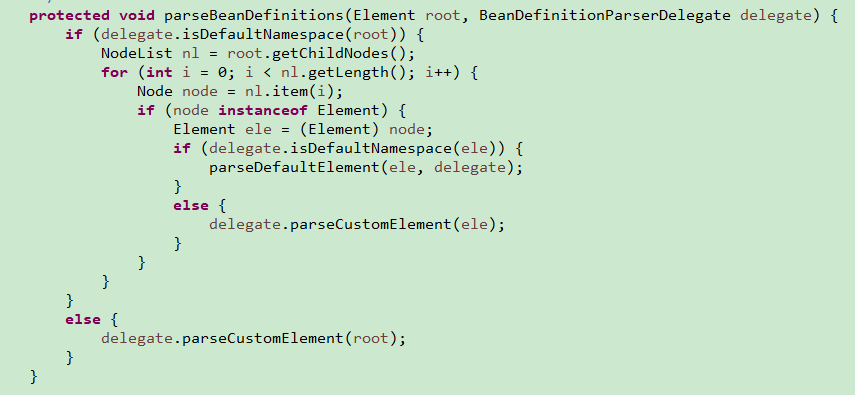


从这里开始 Beandefinition的注册



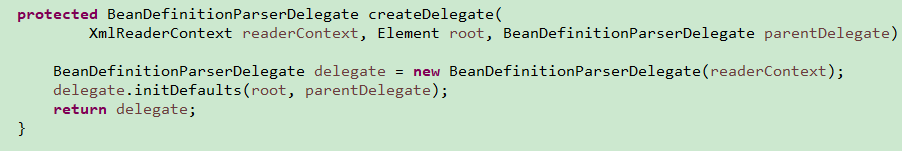
DefaultBeanDefinitionDocumentReader





BeanDefinitionParserDelegate的创建和初始化

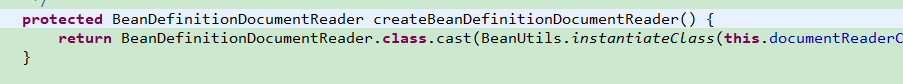
按照Spring的Bean定义规则来对这个XML的文档树进行解析，这个解析是交给BeanDefinitionParserDelegate来完成的



Spring的BeanDefinition是怎么按照Spring的Bean语义要求进行解析并转化成容器内部数据结构的，这个过程是在XmlBeanDefinitionReader的registerBeanDefinitions（）方法中完成的。

BeanDefintion的载入分成两个部分，

1. 首先通过调用XML的解析器得到document对象，但这些document对象并没有按照Spring的Bean规则进行解析。在完成通用的XML解析以后，才是按照spring的Bean规则进行解析的地方，这个按照Spring的Bean规则进行解析的过程是在documentReader中实现的。这里使用的documentReader是默认设置好的DefaultBeanDefinitionDocumentReader.这个DefaultBeanDefinitionDocumentReader的创建是通过



This.documentReaderClass=DefaultBeanDefinitionDocumentReader.class;

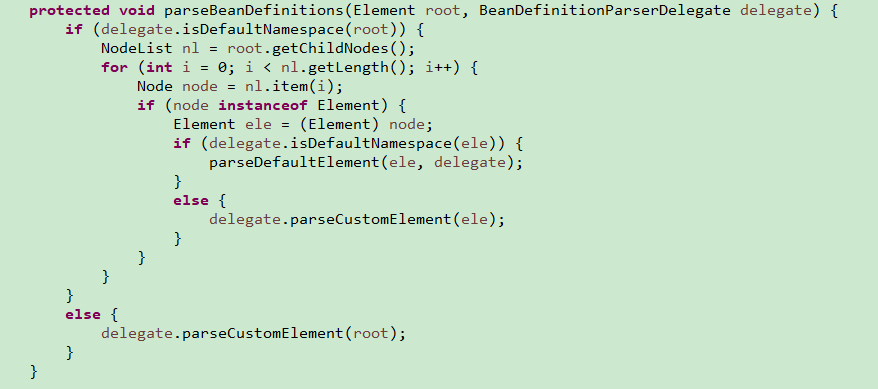
C:\Users\roseonly\AppData\Local\Temp\1562578803(1).png

来完成的。

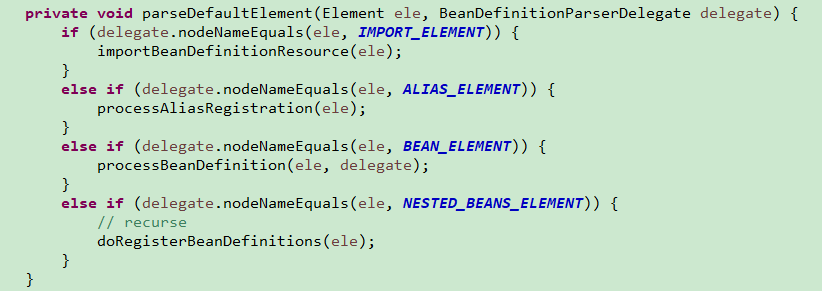
1. 完成BeanDefinition的处理，处理的结果由BeanDefinitionHolder对象来持有。

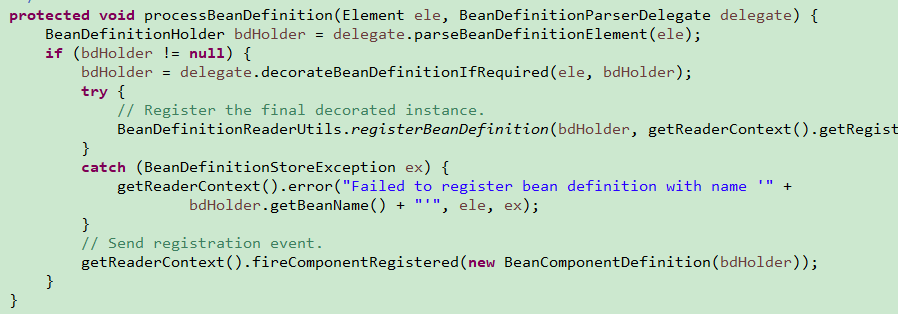
BeanDefinitionHolder除了持有BeanDefinition对象外，还持有其他与BeanDefinition的使用相关的信息，比如Bean的名字，别名集合。BeanDefinitionHolder的生成是通过对Document文档树的内容进行解析来完成的。 可以看到这个解析过程是由BeanDefinitionParserDelegate的processBeanDefinition方法中实现的，同时 这个解析是与Spring对BeanDefinition的配置规则紧密相关的。

DefaultBeanDefinitionDocumentReader的parseBeanDefinitions方法

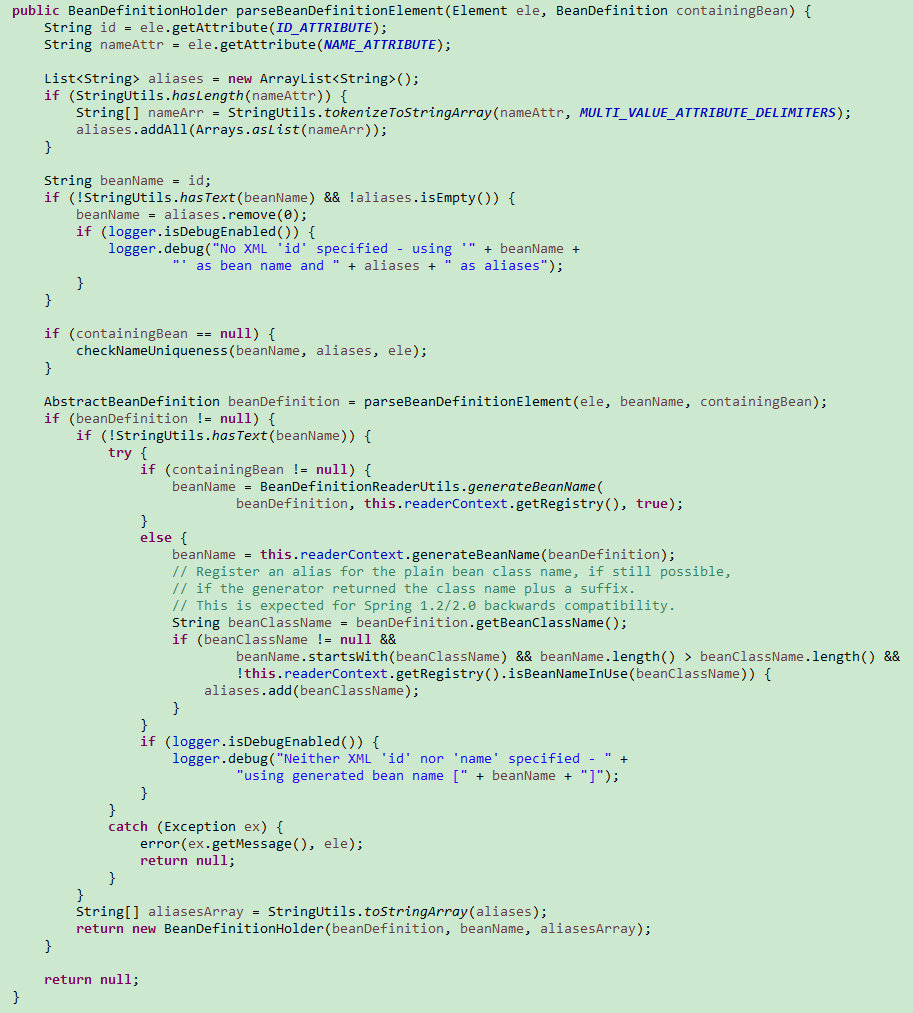


parseDefaultElement(ele, delegate);



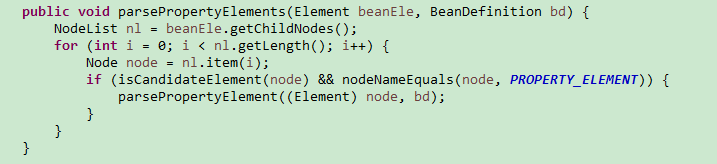


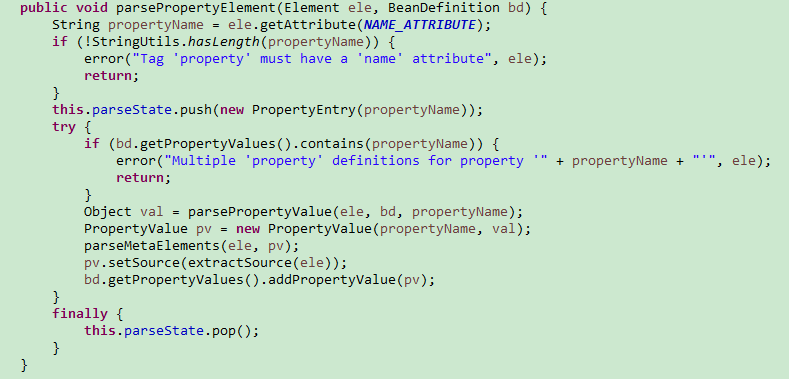
Bean属性配置，通过一个较为复杂的解析过程来完成，这个过程是由parseBeanDefinitionELement来完成的。解析完成以后，会把解析结果放到Beandefinition对象中并设置到BeandefinitionHolder中去。

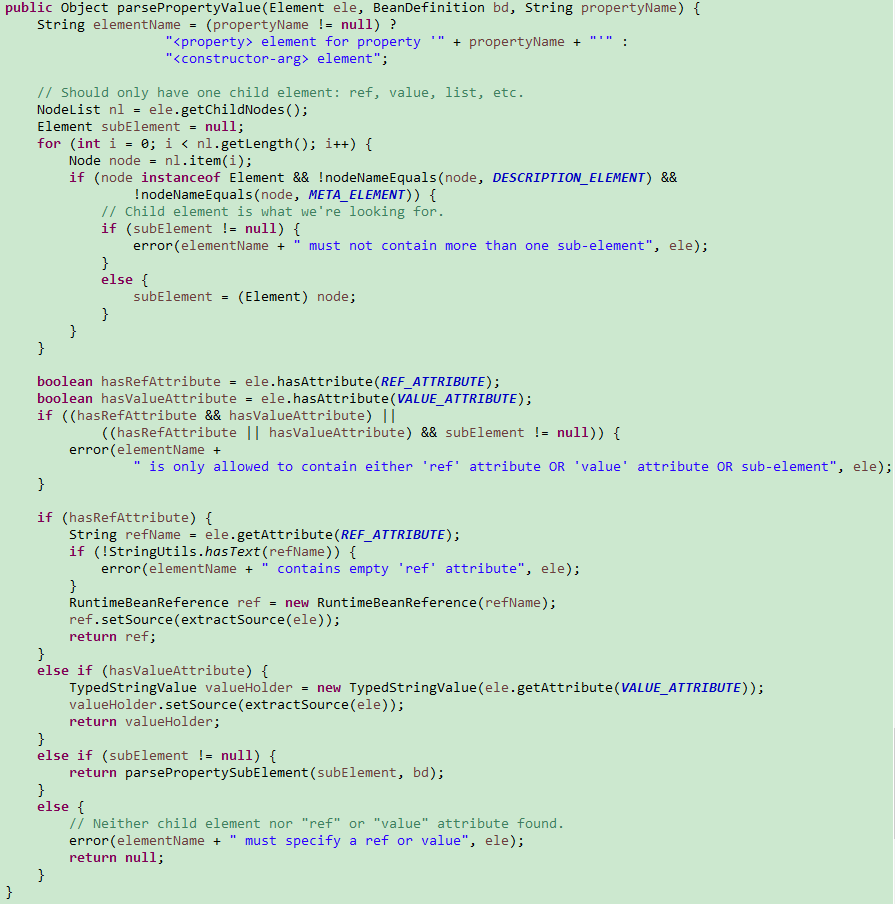




举例说明对property的解析，来完成对整个Beandefinition载入过程的分析，对属性值的处理会被封装到PropertyValue对象并设置到BeanDefinition对象中去。





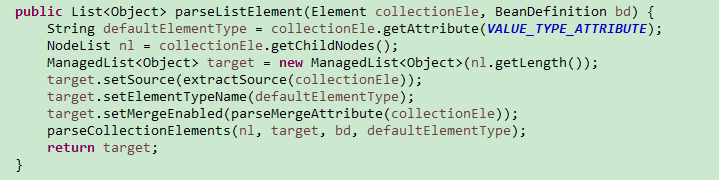


parseArrayElement,parseListElement,parseSetElement,parseMapElement,parsePropELement对应着不同类型的数据解析，同时这些具体的解析方法在BeanDefinitionParserDelegate类中也都能找到。

对ProPerty元素的解析代码实现如下



List的属性配置是怎么被解析的，依然是在BeanDefinitionParserDelegate中，返回的是一个List对象。这个list是Spring定义的ManagedList,作为封装List这类配置定义的数据封装。



经过这样逐层解析，我们在XML文件中定义的BeanDefinition就被整个载入到了IOC容器中，并在容器中建立了数据映射。在IOC容器中建立了对应的数据结构，或者说可以看成是POJO对象在IOC容器中的抽象，这些数据结构可以以AbstractBeandefinition为入口，让IOC容器执行索引，查询和操作。经过以上的载入过程，IOC容器大致完成了管理Beand对象的数据准备工作，

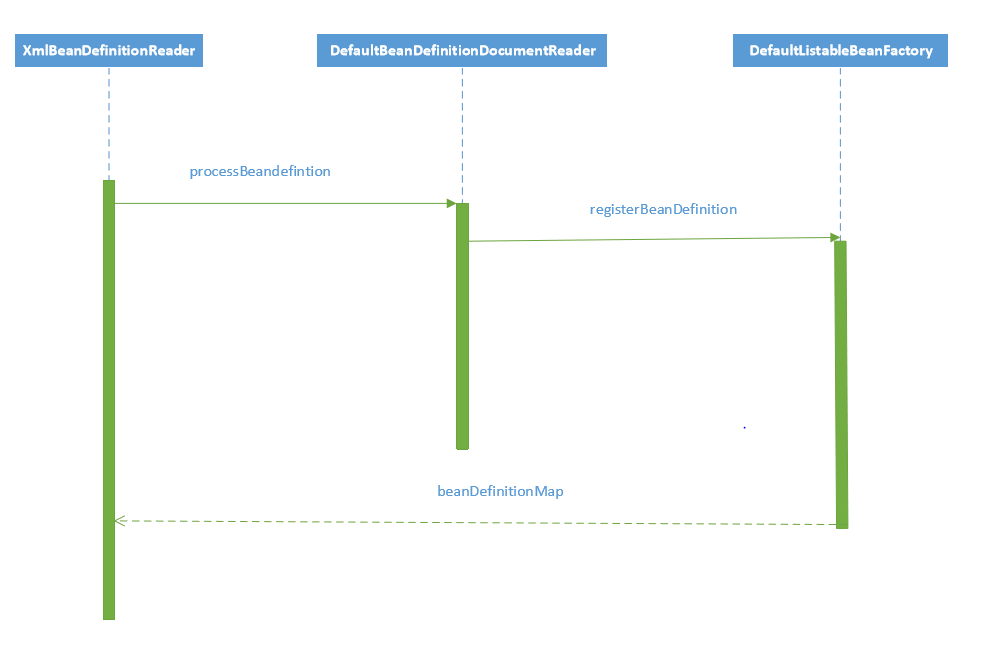
现在IOC容器Beandefinition中存在的还只是一些静态的配置信息。严格地说，这时候的容器还没有完全起作用，要完全发挥容器的作用，还需完成数据向容器的注册。

# Beandefinition的注册

在DefaultListableBeanFactory中，是通过一个HashMap来持有载入的Beandefinition的，这个HashMap的定义在DefaultListableBeanFactory中可以看到



将解析得到的BeanDefinition向IOC容器中的BeanDefinitionMap注册的过程是在载入BeanDefinition完成后进行的。



把解析得到的BeanDefinition设置到hashMap中去。需要注意的是，如果遇到同名的BeanDefinition，进行处理的时候需要依据allowBeanDefintionOverriding的配置来完成。

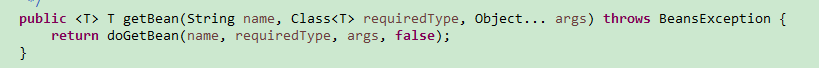


完成了BeanDefinition的注册，就完成了IOC容器的初始化过程，此时，在使用的IOCrongqi defaultListableBeanFactory中已经建立整个Bean的配置信息，而且这些BeanDefintion已经可以被容器使用了，它们都在beandefinitionMap里检索和使用，。容器的作用就是对这个信息进行处理和维护。

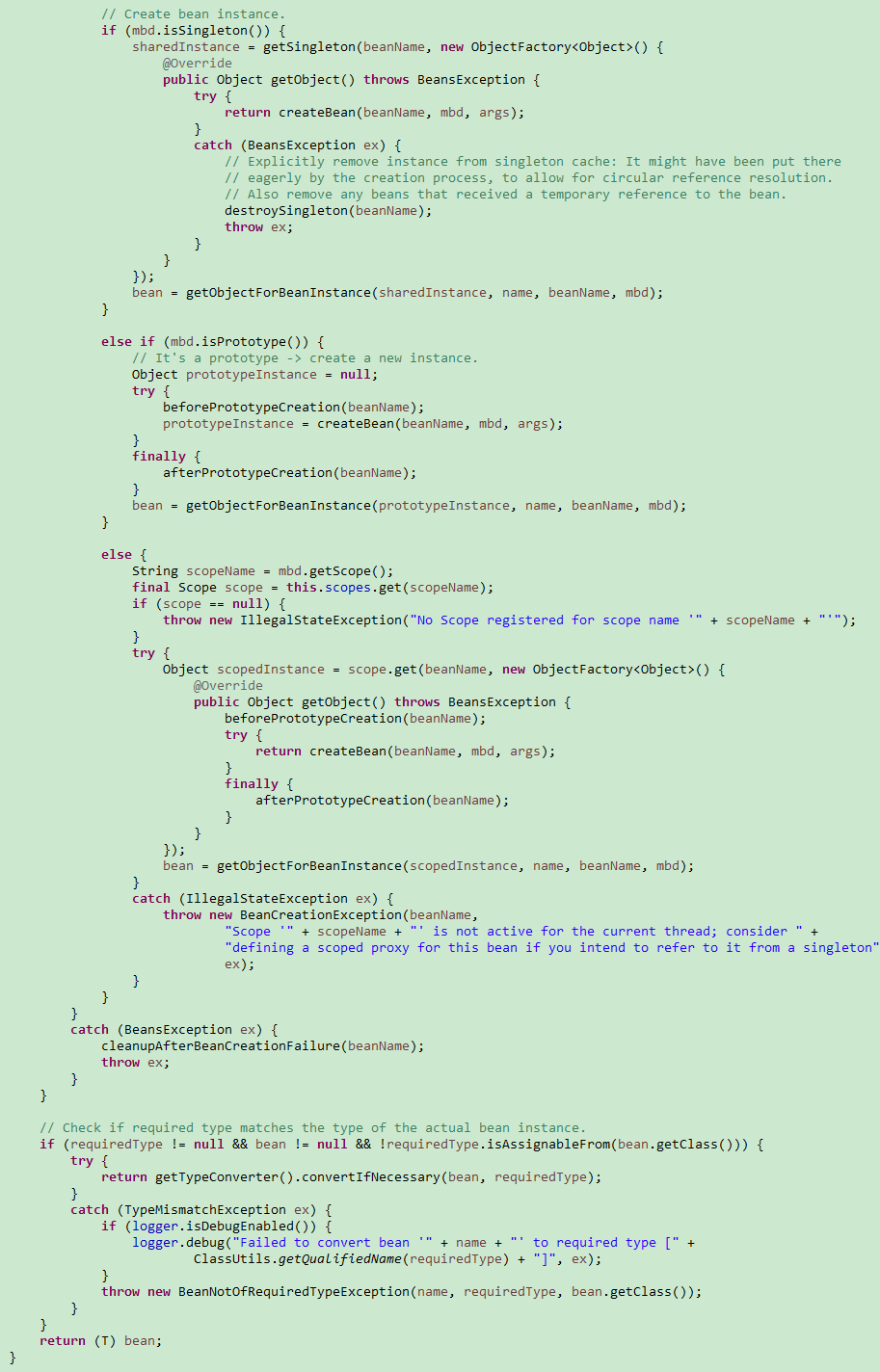
# IOC容器的依赖注入

从DefaultListableBeanFactory的基类AbstractBeanFactory来看依赖注入的实现









getBean是依赖注入的起点， 之后会调用createBean，下面通过createBean代码来了解这个实现过程。在这个过程，Bean对象会根据BeanDefinition定义的要求生成。在AbstractAutiwireCapableBeanFactory中实现了这个createBean,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init-method属性定义，Bean后置处理器等。

注：从spring2.5开始，spring一开始都是使用GenericBeanDefinition类保存Bean的相关信息，在需要时，在将其转换为其他的BeanDefinition类型

是运行时使用的Bean视图，即spring会使用RootBeanDefinition初始化Bean

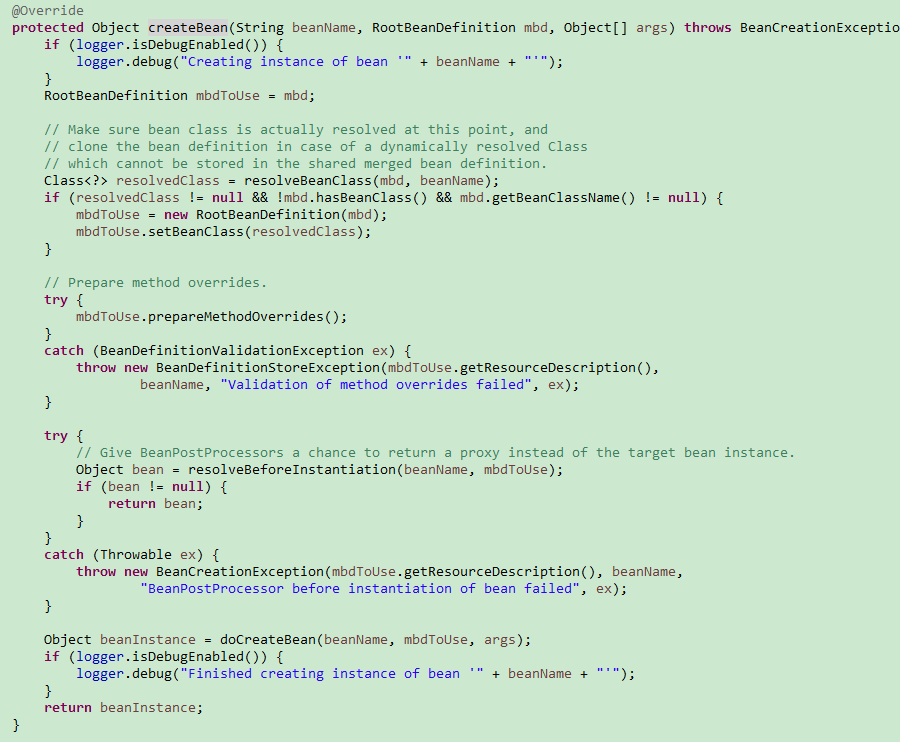
在源码注释中，有这么一句：It might have been created from multiple original bean definitions that inherit from each other，spring会依据多个具有继承关系的BeanDefinition（实际上是GenericBeanDefinition类型）来创建RootBeanDefinition，换句话说，在多继承体系中，RootBeanDefinition代表的是当前初始化类的父类的BeanDefinition

RootBeanDefinition可用于没有继承关系的Bean的创建

RootBeanDefinition，ChildBeanDefinition，GenericBeanDefinition均继承了AbstractBeanDefiniton,其中BeanDefinition是配置文件<bean>元素标签在容器中内部表示形式。<bean>元素标签拥有class、scope、lazy-init等配置属性，BeanDefinition则提供了相应的beanClass、scope、lazyInit属性，BeanDefinition和<bean>中的属性是一一对应的。其中RootBeanDefinition是最常用的实现类，它对应一般性的<bean>元素标签，GenericBeanDefinition是自2.5以后新加入的bean文件配置属性定义类，是一站式服务类。在配置文件中可以定义父<bean>和子<bean>，父<bean>用RootBeanDefinition表示，而子<bean>用ChildBeanDefiniton表示，而没有父<bean>的<bean>就使用RootBeanDefinition表示。AbstractBeanDefinition对两者共同的类信息进行抽象。

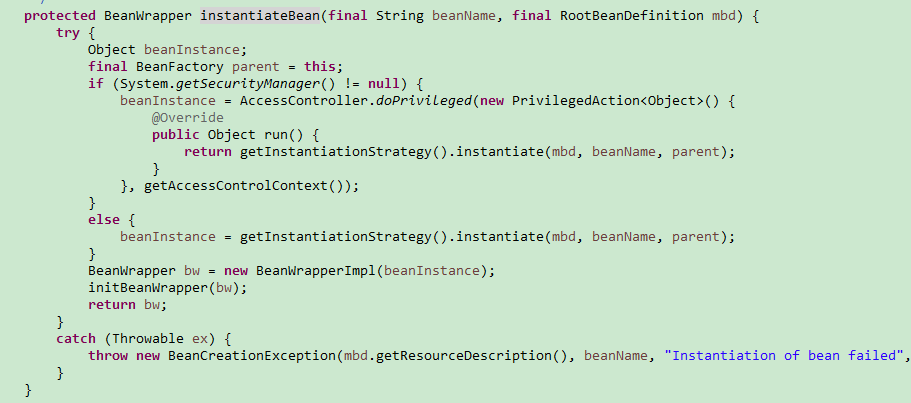
Spring通过BeanDefinition将配置文件中的<bean>配置信息转换为容器的内部表示，并将这些BeanDefiniton注册到BeanDefinitonRegistry中。Spring容器的BeanDefinitionRegistry就像是Spring配置信息的内存数据库，主要是以map的形式保存，后续操作直接从BeanDefinitionRegistry中读取配置信息。一般情况下，BeanDefinition只在容器启动时加载并解析，除非容器刷新或重启，这些信息不会发生变化，当然如果用户有特殊的需求，也可以通过编程的方式在运行期调整BeanDefinition的定义。

AbstractAutowireCapableBeanFactory





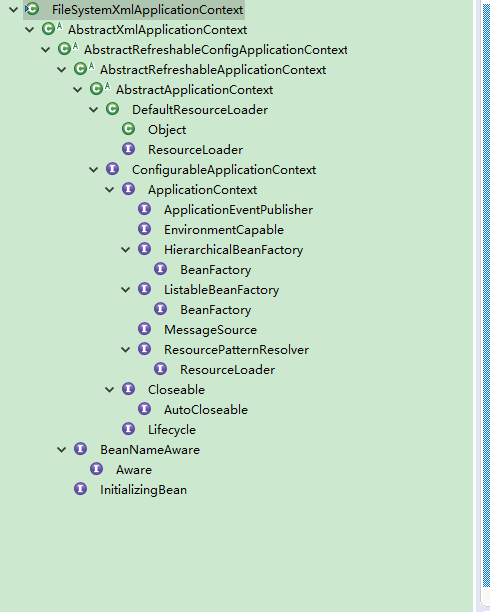






# 详细源码分析

## FileSystemXmlApplicationContext



我们可以通过多种构造函数来创建FileSystemXmlApplicationContext容器，可以指定他的父容器，可以指定他的资源路径。

当创建该容器的对象实例时，构造函数被调用。

最终调用

public FileSystemXmlApplicationContext(

String[] configLocations, boolean refresh, @Nullable ApplicationContext parent)

throws BeansException {

configLocations为资源路径。Refresh用来标记该容器是否需要刷新。不传的话默认为true。

Parent为父容器，可以为null值。

该方法分为三步：

1.首先调用父容器的构造方法初始化。super(parent);

2.设置资源文件。setConfigLocations(configLocations);

3.刷新容器。refresh();

第二步，会调用它的上二级的父类AbstractRefreshableConfigApplicationContext方法来处理。

将configLocations的定义设置到该抽象类的成员变量中。

第三步，会调用它的上四级的父类AbstractApplicationContext的方法来处理

在refresh();中，分12步完成。

1. 容器刷新前的准备，设置上下文状态，获取属性，验证必要的属性等

prepareRefresh();

2. 创建容器，加载解析资源文件，加载并注册bean

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

1. 对刚创建的IoC容器进行一些预处理（设置一些公共属性），为BeanFactory配置容器特性，配置标准的beanFactory，设置ClassLoader，设置SpEL表达式解析器，添加忽略注入的接口，添加bean，添加bean后置处理器等

prepareBeanFactory(beanFactory);

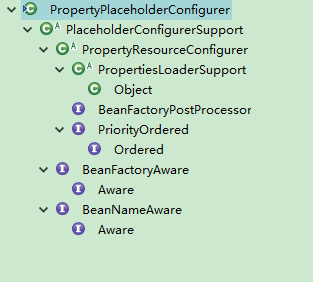
1. 允许在上下文子类中对bean工厂进行后处理，为容器的某些子类指定特殊的BeanPost事件处理器 。模板方法，此时，所有的beanDefinition已经加载，但是还没有实例化。允许在子类中对beanFactory进行扩展处理。比如添加ware相关接口自动装配设置，添加后置处理器等，是子类扩展prepareBeanFactory(beanFactory)的方法

postProcessBeanFactory(beanFactory);

1. 实例化并调用所有注册的BeanFactory的后置处理器BeanFactoryPostProcessor，也就是调用BeanFactoryPostProcessor后置处理器对BeanDefinition处理（实现接口BeanFactoryPostProcessor的bean，在beanFactory标准初始化之后执行）。

invokeBeanFactoryPostProcessors(beanFactory);

例如：PropertyPlaceholderConfigurer(处理占位符)



6．为BeanFactory注册BeanPost事件处理器. BeanPostProcessor是Bean后置处理器，用于监听容器触发的事件 。实例化和注册beanFactory中扩展了BeanPostProcessor的bean。

registerBeanPostProcessors(beanFactory);

例如：

AutowiredAnnotationBeanPostProcessor(处理被@Autowired注解修饰的bean并注入)

RequiredAnnotationBeanPostProcessor(处理被@Required注解修饰的方法)

CommonAnnotationBeanPostProcessor(处理@PreDestroy、@PostConstruct、@Resource等多个注解的作用)等。

7. 初始化一些消息源（比如处理国际化的i18n等消息源）

initMessageSource();

8. 初始化应用事件广播器

initApplicationEventMulticaster();

9.模板方法，在容器刷新的时候可以自定义逻辑，不同的Spring容器做不同的事情。初始化一些特殊的bean，调用子类的某些特殊Bean初始化方法

onRefresh();

10. 注册一些监听器，广播early application events

registerListeners();

11. 实例化剩余的单例bean（非懒加载方式）

finishBeanFactoryInitialization(beanFactory);

比如invokeBeanFactoryPostProcessors方法中根据各种注解解析出来的类，在这个时候都会被初始化。

实例化的过程各种BeanPostProcessor开始起作用。

12. 完成刷新时，需要发布对应的事件，初始化容器的生命周期事件处理器，并发布容器的生命周期事件

finishRefresh();

refresh做完之后需要做的其他事情。

清除上下文资源缓存（如扫描中的ASM元数据）

初始化上下文的生命周期处理器，并刷新（找出Spring容器中实现了Lifecycle接口的bean并执行start()方法）。

发布ContextRefreshedEvent事件告知对应的ApplicationListener进行响应的操作

第一步： 受保护，本类方法内

1.设定启动时间，设定关闭状态，设定激活状态。

2. 在上下文环境中初始化任何占位符属性源，例如web容器中的spring的Profile测试与正式环境的区分。初始化placeholder属性

initPropertySources方法主要是调用了AbstractEnvironment的getPropertySources方法获取了之前SpringApplication的prepareEnvironment方法中getOrCreateEnvironment方法准备的各种环境变量及配置并用于初始化ServletPropertySources。具体的servletContextInitParams这些是在环境对象初始化时由各集成级别Environment的customizePropertySources方法中初始化的。

3.查看标示为必填的属性信息是否都有了

4. 允许收集早期的ApplicationEvents，在多播器可用时发布.

第二步： 受保护，本类方法内

1. 创建IoC容器（DefaultListableBeanFactory）并返回。

2. 加载解析XML文件（最终存储到Document对象中）。

3. 读取Document对象，并完成BeanDefinition的加载和注册工作。

在obtainFreshBeanFactory方法中，AbstractApplicationContext会调用自己的抽象方法refreshBeanFactory();，该抽象方法可以由自己的子类AbstractRefreshableApplicationContext来实现。

在子类AbstractRefreshableApplicationContext的refreshBeanFactory()内部实现为：

1. 如果之前有IoC容器，则销毁

调用的是父类的公共方法

1. 创建IoC容器并返回，也就是DefaultListableBeanFactory

DefaultListableBeanFactory beanFactory = createBeanFactory();

1. 指定序列化ID，如果需要的话，让这个BeanFactory从id反序列化到BeanFactory对象
2. 定制BeanFactory，设置工厂的属性：

1. 是否允许覆盖同名称的不同定义的对象

2. 是否允许bean之间存在循环依赖

customizeBeanFactory(beanFactory);

1. 加载BeanDefinition。

loadBeanDefinitions(beanFactory);

1. 使用全局变量记录BeanFactory类实例。

synchronized (this.beanFactoryMonitor) {

this.beanFactory = beanFactory;

}

refresh()和refreshBeanFactory（）这两个方法放在AbstractApplicationContext的原因：

方便子类的调用。属于子类公共的方法。

refreshBeanFactory由类去自定义实现。模板方法

refreshBeanFactory方法由AbstractRefreshableApplicationContext来实现。又提取了一个抽象类。在AbstractRefreshableApplicationContext中创建了DefaultListableBeanFactory的一个典型工厂。

loadBeanDefinitions（beanFactory）；加载BeanDefinitions

这个方法由FileSystemXmlApplicationContext的上一级父类来实现。

虽然加载的方式不用，但其实是一个流程。通过传入默认工厂实现，来回调。



获取资源的定位有个判断

ClassPathXmlApplicationContext中进行了实现

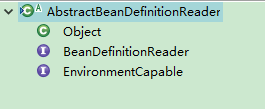
Resource[] configResources = getConfigResources();

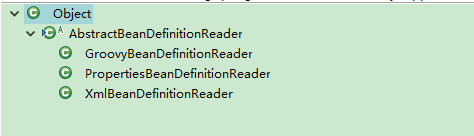
获取不为空

FileSystemXmlApplicationContext没有使用该方法，所以为空，当子类中获取的资源定位为空，则获取FileSystemXmlApplicationContext构造方法中setConfigLocations方法设置的资源

AbstractRefreshableConfigApplicationContext

专门用来处理刷新工厂时配置加载的抽象类



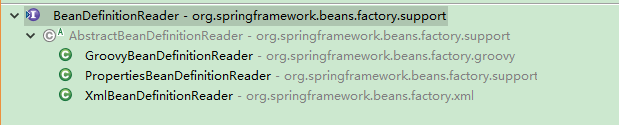


AbstractBeanDefinitionReader 实现了BeanDefinitionReader接口

**public** **int** loadBeanDefinitions(Resource... resources)

通过resources获取和通过string获取不一样的。

还是要通过子类来自己实现如何加载单个Resource



# 其他

## Java中关于initcause的用法说明

initCause()这个方法就是对异常来进行包装的，目的就是为了出了问题的时候能够追根究底。因为一个项目，越往底层，可能抛出的异常类型会用很多，如果你在上层想要处理这些异常，你就需要挨个的写很多catch语句块来捕捉异常，这样是很麻烦的。如果我们对底层抛出的异常捕获后，抛出一个新的统一的异常，会避免这个问题。但是直接抛出一个新的异常，会让最原始的异常信息丢失，这样不利于排查问题。举个例子，在底层会出现一个A异常，然后在中间代码层捕获A异常，对上层抛出一个B异常。如果在中间代码层不对A进行包装，在上层代码捕捉到B异常后就不知道为什么会导致B异常的发生，但是包装以后我们就可以用getCause()方法获得原始的A异常。这对追查BUG是很有利的。

|  |
| --- |
| class A  {  try  {  ....  }  catch(AException a)  {  throw new BException();  }  }  ...  class B  {  try  {  ...  }  catch(BException b)  {  //这时候你需要去看b异常式什么问题导致的，你在A类里面  //没有对AException进行包装，所以你无法知道是A导致的B  }  } |

例子

|  |
| --- |
| **package** com.dada.carpool;  **public** **class** test {  **public** **static** **void** main(String[] args) **throws** Throwable {  **try** {  *Exception1*();  }  **catch**(Exception e) {  System.***out***.println(e);  }  }    **public** **static** **void** Exception1()**throws** amitException{  **try** {  *Exception2*();  }  **catch**(otherException e) {  amitException a1 = **new** amitException();    // initializes the cause of this throwable to the specified value.  // a1.initCause(e);  **throw** a1;  }  }    **public** **static** **void** Exception2() **throws** otherException {  **throw** **new** otherException();  }  }  **class** amitException **extends** Throwable {  amitException() {  **super**("This is my Exception....");  }  }  **class** otherException **extends** Throwable {  otherException(){  **super**("This is any other Exception....");  }  } |

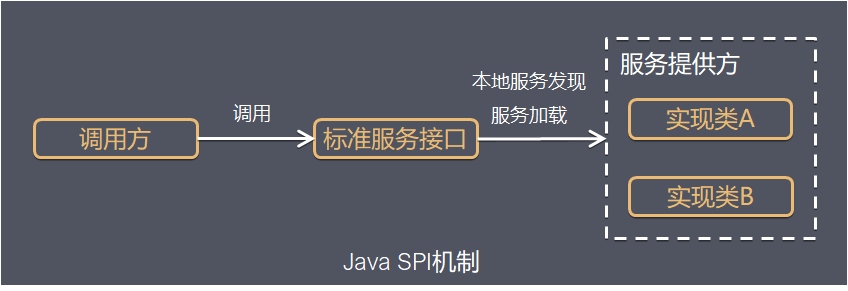
如果包装以后：

|  |
| --- |
| class A  {  try  {  ...  }  catch(AException a)  {  BException b = new BEexception()；  b.initCause(a);  throw b;  }  }  ...  class B  {  try  {  ...  }  catch(BException b)  {  //什么导致了b呢？  b.getCause()；//得到导致B异常的原始异常  }  } |

BeanDefinitionDocumentReader 负责解析生成的document对象

## java SPI

SPI全称Service Provider Interface，是Java提供的一套用来被第三方实现或者扩展的API，它可以用来启用框架扩展和替换组件。



Java SPI 实际上是“**基于接口的编程＋策略模式＋配置文件**”组合实现的动态加载机制。

系统设计的各个抽象，往往有很多不同的实现方案，在面向的对象的设计里，一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。  
Java SPI就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似IOC的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。所以SPI的核心思想就是**解耦**。

其中driver是DriverInfo类中的的Driver类型的成员变量，而Driver(java.sql.Driver)是个接口，所以点进去的connect方法只是个Driver接口里面定义的方法，并没有实现类，实现类哪里去了？？？(虽然后来在com.mysql.fabric.jdbc包里碰巧找到了，但不知道原理咋回事)。

Java Class.cast方法

我们来看一下JDK中这个方法的源码！

   @SuppressWarnings("unchecked")

    public T cast(Object obj) {

        if (obj != null && !isInstance(obj))

            throw new ClassCastException(cannotCastMsg(obj));

        return (T) obj;

    }

其中return (T) obj;就是强制转换成你所需要的类型，相当于你自己写（假设有A和B两个类，B继承了A类）：

A  a = (A) b1;

cast方法不过是给你做了一些安全措施而已

**这个泛型T就是自动判断你要转换成什么类型，这也是一个安全措施！**

委托模式Delegate

就是代理模式Poxy