

目录

[第1部分核心实现 3](#_Toc26117548)

[第 1 章 Spring 整体架构和环境搭建 3](#_Toc26117549)

[1.1Spring的整体架构 3](#_Toc26117550)

[1.2环境搭建 5](#_Toc26117551)

[1.3 cglib 和 obienesis 的编译错误解决 6](#_Toc26117552)

[1.4 AspectJ 编译问题解决 6](#_Toc26117553)

[第2章 容器的基本实现 6](#_Toc26117554)

[2.1容器基本用法 7](#_Toc26117555)

[2.2 功能分析 8](#_Toc26117556)

[2.3工程搭建 8](#_Toc26117557)

[2.4 Spring的结构组成 9](#_Toc26117558)

[2.5 容器的基础XmlBeanFactory 13](#_Toc26117559)

[2.5.1 配置文件封装 14](#_Toc26117560)

[2.5.2 加载 Bean 17](#_Toc26117561)

[2.6 获取 XML 的验证模式 21](#_Toc26117562)

[2.7 获取 Document 24](#_Toc26117563)

[2.8 解析及注册 BeanDefinitions 28](#_Toc26117564)

# 第1部分核心实现

* 第l章Spring整体架构和环境搭建
* 第2章容器的基本实现
* 第3章默认标签的解析
* 第4章自定义标签的解析
* 第5章bean的加载
* 第6章容器的功能扩展
* 第7章AOP

## 第 1 章 Spring 整体架构和环境搭建

Spring是于2003年兴起的一个轻量级Java开源框架，由RodJohnson在其著作Expert One-On-OneJ2EE Designand Development中阐述的部分理念和原型衍生而来。

Spring是为了解决企业应用开发的复杂性而创建的，它使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发，从简单性、可测试性和松搞合的角度而言，任何Java应用都可以从Spring中受益。

### 1.1 Spring的整体架构

Spring框架是一个分层架构，它包含一系列的功能要素，并被分为大约20个模块，如图1-1所示。

Spring Framework Runtime

Data Access/Integration

JDBC

ORM

OXM

JMS

Transactions

Web  
(MVC/Remoting)

Servlet

Portiet

Web

AOP

Test

图 1-1 Spring 整体架构图

Aspects

Instrumentation

Core Container

Context

Expression  
Language

Beans

Core

这些模块被总结为以下几部分。

#### 1.CoreContaine

CoreContainer（核心容器）包含有Core、Beans、Context和ExpressionLanguage模块。

Core和Beans模块是框架的基础部分，提供IoC（转控制）和依赖注入特性。这里的基础概念BeanFactory，它提供对Factory模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

* Core模决主要包含Spring框架基本的核心工具类，Spring的其他组件都要用到这个包里的类，Core模块是其他组件的基本核心。当然你也可以在自己的应用系统中使用这些工具类。
* Beans模块是所有应用都妥用到的，它包含访问配直文件、创建和管理bean以及进行

Inversion of Control/Dependency Injection(IoC/DI）操作相关的所有类。

* Context模块构建于Core和Beans模块基础之上，提供了一种类似于JNDI注册器的框架式的对象访问方法。Context模块继承了Beans的特性，为Spring核心提供了大量扩展，添加了对国际化（例如资源绑定）、事件传播、资源加载和对Context的透明创建的支持。Context模块同时也支持J2EE的一些特性，例如EJB、JMX和基础的远程处理。ApplicationContext接口是Context模块的关键。
* Expression Language模块提供了强大的表达式语言，用于在运行时查询和操纵对象。它是JSP2.1规范中定义的unifed expression language的扩展。该语言支设置／获取属性的值，属性的分配，方法的调用，访问数组上下文（accessiong the context of arrays）、容器和索引器、逻辑和算术运算符、命名变量以及从Spring的IoC容器中根据名称检索对象。它也支持list投影、选择和一般的list聚合。

#### 2.DataAccess/Integration

DataAccess/Integration层包含了JDBC、ORM、OXM、JMS和Transaction模块。

* JDBC模块提供了一个JDBC抽象层，它可以消除冗长的JDBC编码和解析数据库厂商特有的错误代码。这个模块包含了Spring对JDBC数据访问进行封装的所有类
* ORM模块为流行的对象－关系映射API，如JPA、JDO、Hibernate、iBatis等，提供了一个交互层。利用ORM封装包，可以混合使用所有Spring提供的特性进行O/R映射，如前边提到的简单声明性事务管理。

Spring框架插入了若干个ORM框架，从而提供了ORM的对象关系工具，其中包括JDO、Hibernate、iBatis SQLMap。所有这些都遵从Spring的通用事务和DAO异常层次结构。

* OXM模块提供了一个对Object/XML映射实现的抽象层，Object/XML映射实现包括JAXB、Castor、XMLBeans、JiBX和XStrearn。
* JMS(Java Messaging Service）模块主要包含了一些制造和消费消息的特性。
* Transaction模块支持编程和声明性的事务管理，这些事务类必须实现特定的接口，并且对所有的POJO都适用。

#### 3.Web

Web上下文模块建立在应用程序上下文模块之上，为基于Web的应用程序提供了上下文。所以，Spring框架支持与Jakarta Struts的集成。Web模块还简化了处理大部分请求以及将请求参数绑定到域对象的工作。Web层包含了Web、Web-Servlet、Web-Struts和Web-Porlet模块，具体说明如下。

* Web模块：提供了基础的面向Web的集成特性。例如，多文件上传、使用servlet listeners初始化IoC容器以及一个面向Web的应用上下文。它还包含Spring远程支持中Web的相关部分。
* Web-Serviet模块web.servlet.jar：该模块包含Spring的model-view-controller(MVC)实现。Spring的MVC框架使得模型范围内的代码和web forms之间能够清楚地分离开来，并与Spring框架的其他特性集成在一起。
* Web-Struts模块：该模块提供了对Struts的支持，使得类在Spring应用中能够与一个典型的Struts Web层集成在一起。注意，该支持在Spring3.0中已被弃用。
* Web-Porlet模块：提供了用于Portlet环境和Web-Servlet模块的MVC的实现。

#### 4.AOP

AOP模块提供了一个符合AOP联盟标准的面向切面编程的实现，它让你可以定义例如方法拦截器和切点，从而将逻辑代码分开，降低它们之间的调合性。利用source-level的元数据功能，还可以将各种行为信息合并到你的代码中，这有点像.Net技术中的attribute概念。

通过配置管理特性，Spring AOP模块直接将面向切面的编程功能集成到了Spring框架中，所以可以很容易地使Spring框架管理的任何对象支持AOP。Spring AOP模块为基于Spring的应用程序中的对象提供了事务管理服务。通过使用Spring AOP，不用依赖EJB组件，就可以将声明性事务管理集成到应用程序中。

* Aspects模块提供了对AspectJ的集成支持。
* Instrumentation模块提供了class instrumentation支持和classloader实现，使得可以在特定的应用服务器上使用。

#### 5.Test

Test模块支持使用JUnit和TestNG对Spring组件进行测试。

### 1.2环境搭建

由于在上一版本中作者对环境搭建描述得比较粗糙，导致非常多的读者询问，作者在此表示歉意，并在这一版本中补充了非常详细的Spring环境搭建流程。如果是第一次接触Spring源码的环境搭建，确实还是比较麻烦的。作者使用的编译器为目前流行的IntelliJIDEA,I阪木为2018旗舰版。Eclipse用户还需要作己揣摩环境搭建方法，这里不再赘述。

#### 1.2.1 源码链接获取

#### 1.2.2 源码下载及 IDEA 导入

### 1.3 cglib 和 obienesis 的编译错误解决

#### 1.3.1 问题发现及原因

#### 1.3.2 问题解决

### 1.4 AspectJ 编译问题解决

#### 1.4.1 问题发现

#### 1.4.2 问题原因

#### 1.4.3 问题解决

第2章容器的基本实现  
源码分析是一件非常煎熬且极具挑战性的任务，你准备好开始战斗了吗？  
在正式开始分析Spring源问之前，我们有必要先来回顾一下Spring中最简单的用法，尽  
管我相信您已经对这个例子非常熟悉了。  
2.1容器基本用法  
bean是Spring中最核心的东西，因为Spring就像是个大水桶，而bean就像是容器中的水，  
水桶脱离了水便也没什么用处了，那么我们先看看bean的定义。

## 第 2 章 容器的基本实现

源码分析是一件非常煎熬且极具挑战性的任务，你准备好开始战斗了吗？

在正式开始分析Spring源问之前，我们有必要先来回顾一下Spring中最简单的用法，尽管我相信您已经对这个例子非常熟悉了。

### 2.1容器基本用法

bean是Spring中最核心的东西，因为Spring就像是个大水桶，而bean就像是容器中的水，水桶脱离了水便也没什么用处了，那么我们先看看bean的定义。

|  |
| --- |
| **public class** MyTestBean {  **private** String **testStr** = **"testStr"**;   **public** String getTestStr() {  **return testStr**;  }   **public void** setTestStr(String testStr) {  **this**.**testStr** = testStr;  } } |

这么看来bean并没有任何特别之处，的确，Spring的目的就是让我们的bean能成为－个纯粹的POJO，这也是Spring所边求的。接下来看看配置文件：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/spring-beans.xsd"**>   <**bean id="myTestBean" class="com.prosayj.springboot.spring源码深度剖析.\_02\_容器的基本实现.MyTestBean"**></**bean**> </**beans**> |

在上面的配置中我们看到了bean的声明方式，尽管Spring中bean的元素定义着N种属性来支撑我们业务的各种应用，但是我们只要声明成这样，基本上就已经可以满足我们的大多数应用了。好了，你可能觉得还有什么,但是，真没了，Spring的人门示例到这里已经结束，我们可以写测试代码测试了。

|  |
| --- |
| @SuppressWarnings(**"deprecation"**) **public class** \_01\_BeanFactoryTest {  @Test  **public void** testSimpleLoad() {  XmlBeanFactory beanFactory = **new** XmlBeanFactory(  **new** ClassPathResource(**"com/prosayj/springboot/spring源码深度剖析/\_02\_容器的基本实现/beanFactoryTest.xml"**));  MyTestBean bean = (MyTestBean) beanFactory.getBean(**"myTestBean"**);  *assertEquals*(**"testStr"**, bean.getTestStr());  } } |

相信聪明的读者会很快看到我们期望的结果：在Eclipse中显示了Grean Bar。直接使用BeanFactory作为容器对于Spring的使用来说并不多见，甚至是甚少使用，因为在企业级的应用中大多数都会使用的Application Context（后续章节我们会介绍它们之间的区别），这里只是用于测试，让读者更快更好地分析Spring的内部原理。

OK，我们又复习了一遍Spring，你是不是会很不屑呢？这样的小例子没有任何挑战性。嗯，确实，这样的使用是过于简单了，但是本书的目的并不是介绍如何使用Spring，而是帮助您更好地了解Spring的内部原理。读者可以自己先想想，上面的一句简单代码都执行了什么样的逻辑呢？这样一句简单代码其实在Spring中执行了太多太多的逻辑，即使作者用半本书的文字也只能介绍它的大致原理。那么就让我们快速地进入分析状态吧。

### 2.2 功能分析

现在我们可以来好好分析一下上面测试代码的功能，来探索上面的测试代码中Spring究竟帮助我们完成了什么工作？不管之前你是否使用过Spring，当然，你应该使用过的，毕竟本书面向的是对Spring有一定使用经验的读者，你都应该能猜出来，这段测试代码完成的功能无非就是以下几点。

* 读取自己直文件beanFactoryTest.xml
* 根据beanFactoryTest.xml中的配置找到对应的类的配直，并实例化。
* 调用实例化后的实例。

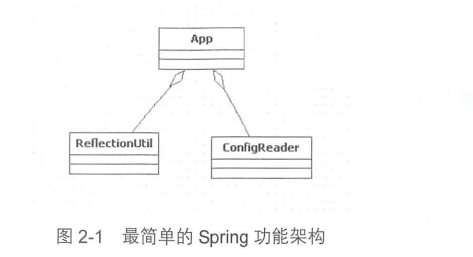
为了更清楚地描述，作者临时画了设计类图，如图2-1所示，如果想完成我们预想的功能，至少需要3个类。

* ConfigReader：用于读取及验证自己直文件。我们妥用配直文件里面的东西，当然首先要做的就是读取，然后放直在内存中。
* ReflectionUtil：用于根据配直文件中的自己直进行反射实例化。比如在上例中beanFactoryTest.xml出现的

<**bean id="myTestBean" class="com.prosayj.springboot.spring.MyTestBean"**></**bean**>

我们就可以根据bean.MyTestBean进行实例化。

* App:用于完成整个逻辑的串联。

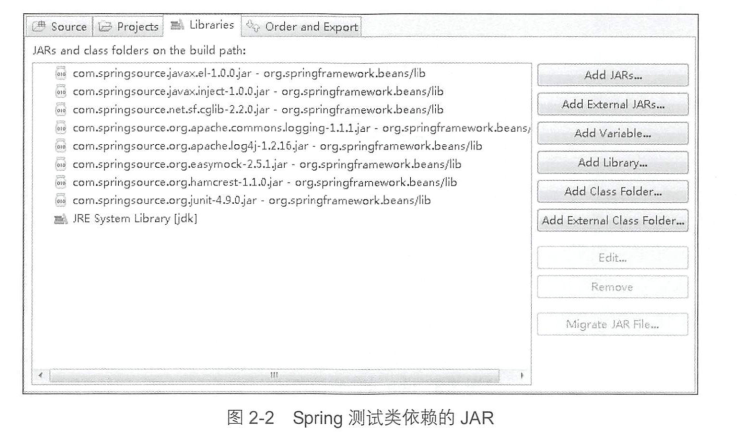


按照原始的思维方式，整个过程无非如此，但是作为一个风靡世界的优秀源代码真的就这么简单吗？

### 2.3工程搭建

不如我们首先大致看看Spring的源码。

在Spring源码中，用于实现上面功能的是org.Springframework.beans.jar，我们看源码的时候要打开这个工程，如果我们只使用上面的功能，那就没有必要引人Spring的其他更多的包，当然Core是必需的，还有些依赖的包如图二2所示。



引人依赖的JAR消除掉所有编译错误后，终于可以看源码了。或许你已经知道了答案，Spring居然用了N多代码实现了这个看似很简单的功能，那么这些代码都是做什么用的呢？

Spring在架构或者编码的时候又是如何考虑的呢？带着疑问，让我们踏上研读Spring源码的征程。

### 2.4 Spring的结构组成

我们首先尝试梳理Spring的框架结构，从全局的角度了解Spring的结构组成。

#### 2.4.1 beans包的层级结构

作者认为阅读源码的最好方法是通过示例跟着操作一遍，虽然有时候或者说大多数时候会被复杂的代码绕来绕去，绕到最后已经不知道自己身在何处了，但是，如果配以UML还是可以搞定的。作者就是按照自己的思路进行分析，并配合必要的UML，希望读者同样可以跟得上思路。

我们先看看整个beans工程的源码结构，如图2-3所示。

* beans包中的各个源码包的功能如下。
* src/main/java用于展现Spring的主要逻辑。
* src/main/resources用于存放系统的配置文件。
* src/test/iava用于对主要逻辑进行单元测试。
* src/test/resources用于存放测试用的配直文件。



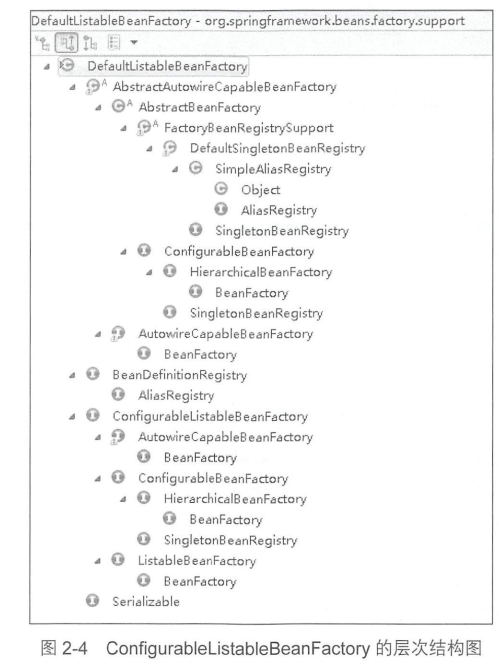
#### 2.4.2 核心类介绍

通过beans工程的结构介绍，我们现在对beans的工程结构有了初步的认识，但是在正式开始源码分析之前，有必要了解Spring中核心的两个类。

##### DefaultlistableBeanFactory

XmlBeanFactory继承向DefaultListableBeanFacto1y，而DefaultListableBeanFactmy是整个bean加载的核心部分，是Spring注册及加载bean的默认实现，而对于XmlBeanFactory与DefaultListableBeanFactory不同的地方其实是在XmlBeanFactory中使用了自定义的XML读取器XmlBeanDefinitionReader，实现了个性化的BeanDefinitionReader读取，DefaultListableBeanFacto1y继承了AbstractAutowireCapableBeanFactory并实现了ConfigurableListableBeanFactory可以及BeanDefinitionRegistry接口。

图2-4是ConfigurableListableBeanFactory的层次结构图，图2-5是相关类图。



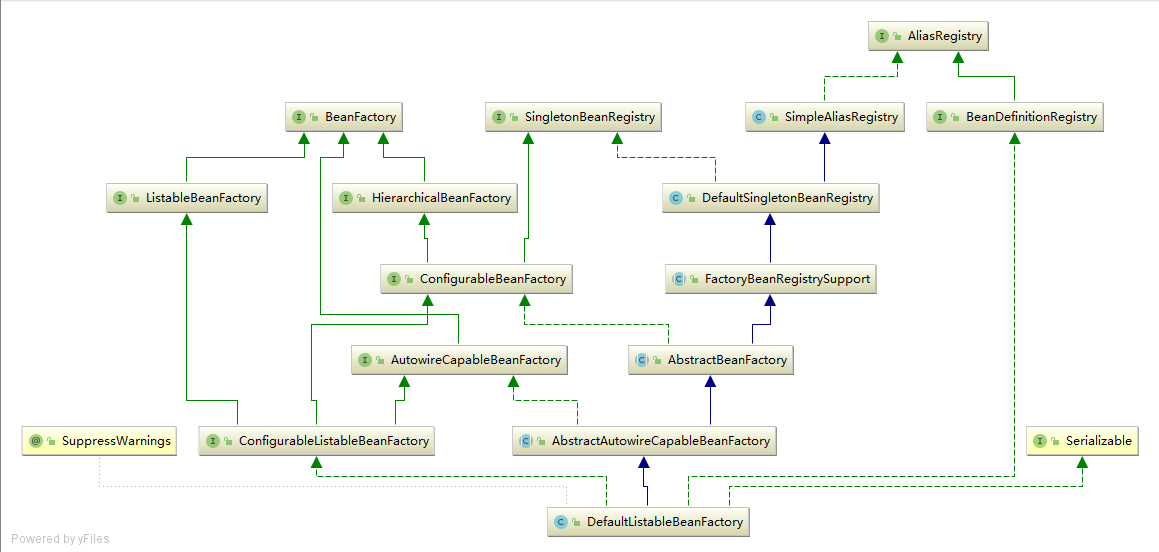
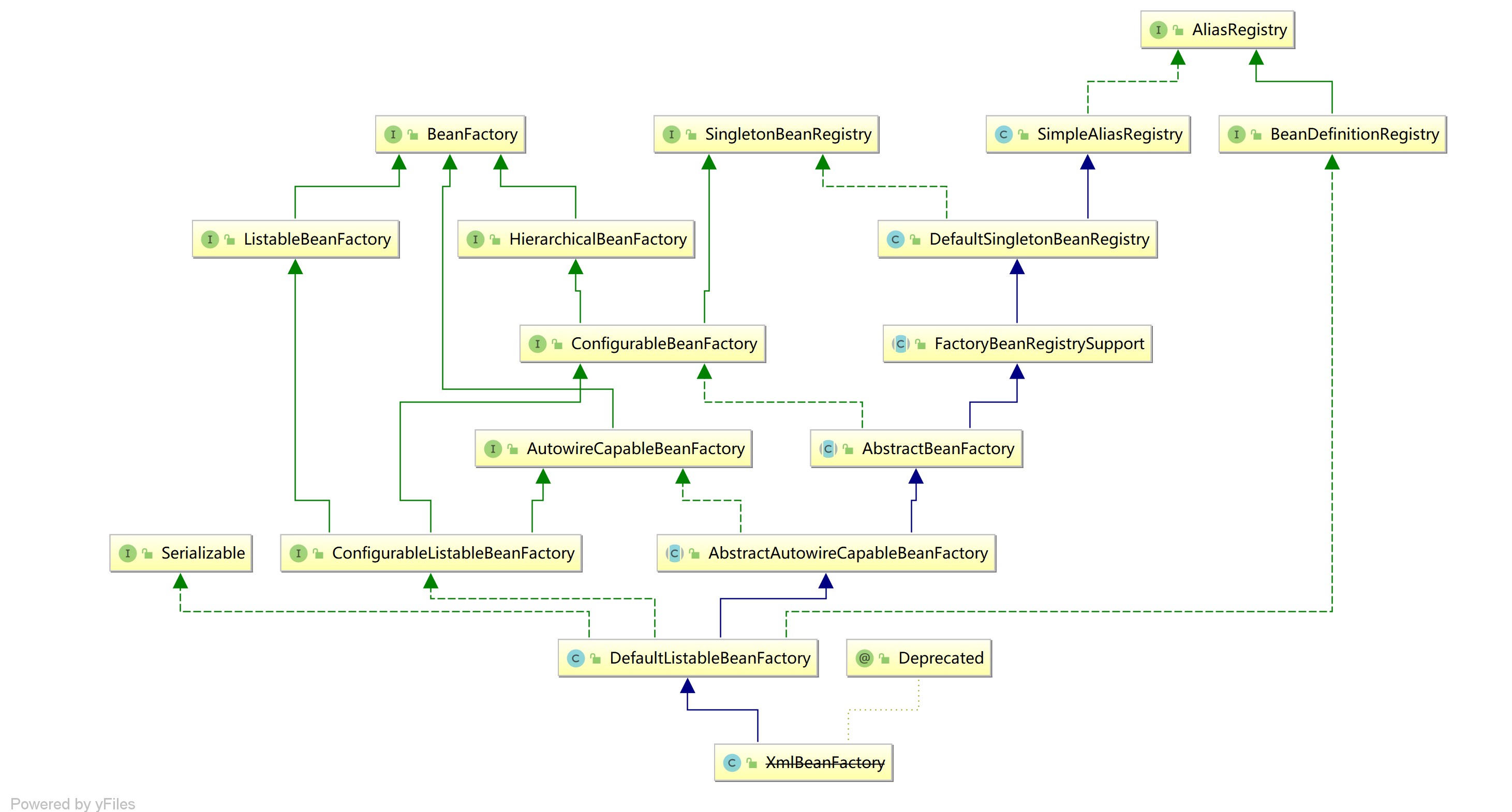


图2-5 容器加载相关类图

从上面的类图以及层次结构阁中，我们可以很清晰地从全局角度了解DefaultListableBeanFactory的脉络。如果读者没有了解过Spring源码可能对上面的类图不是很理解，不过没关系，通过后续的学习，你会逐渐了解每个类的作用。那么，让我们先简单地了解图2-5中各个类的作用。

* AliasRegistrγ：定义对alias的简单增删改等操作。
* SimpleAliasRegistry：主要使用map作为alias的缓存，并对接口AliasRegistry进行实现。
* SingletonBeanRegistry：定义对单例的注册及获取。
* BeanFactory：定义获取bean及bean的各种属性。
* DefauItSingletonBeanRegistry：对接口SingletonBeanRegistry各函数的实现。
* HierarchicalBeanFactory：继承BeanFactory，也就是在BeanFactory定义的功能的基础上增加了对parentFactory的支持。
* BeanDefinitionRegistry：定义对BeanDefinition的各种增删改操作。
* FactoryBeanRegistrySupport：在DefaultSingletonBeanRegistry基础上增加了对FactoryBean的特殊处理功能。
* ConfigurableBeanFactory：提供配直Factory的各种方法。
* ListableBeanFactory：根据各种条件获取bean的配直清单。
* AbstractBeanFactory：综合FactoryBeanRegistrySupport和ConfigurableBeanFactory的功能。
* AutowireCapableBeanFactory：提供创建bean、自动注入、初始化以及应用bean的后处理器。
* AbstractAutowireCapableBeanFactory：综合AbstractBeanFacto1y并对接口AutowireCapableBeanFactory进行实现。
* ConfigurableListableBeanFactory:Beanfactory配直清单，指定忽略类型及接口等。
* DefaultListableBeanFactory：综合上面所有功能，主要是对bean注册后的处理。



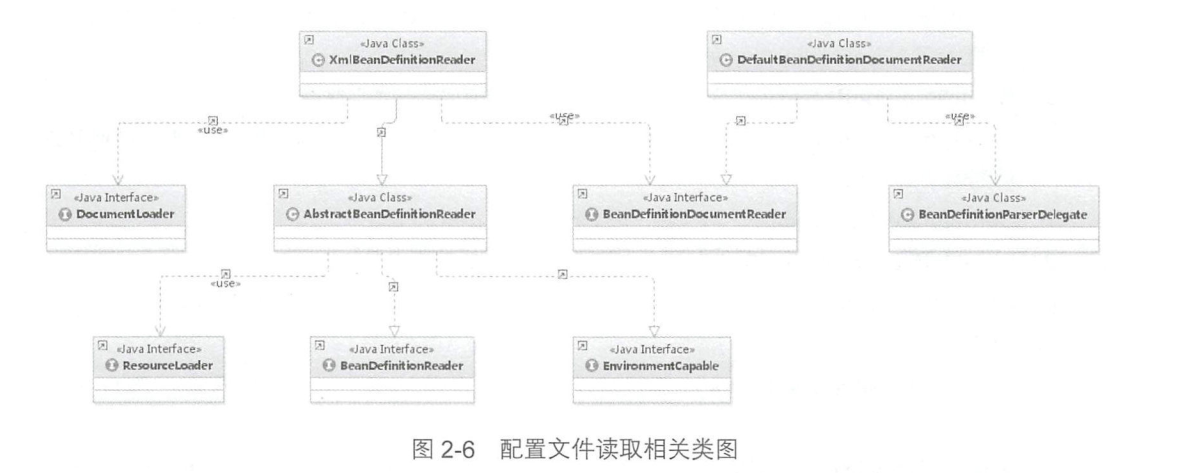
XmlBeanFactory对DefaultListableBeanFactorγ类进行了扩展，主要用于从XML文档中读取BeanDefinition，对于注册及获取bean都是使用从父类DefaultListableBeanFactory继承的方法去实现，而唯独与父类不同的个性化实现就是增加了XmlBeanDefinitionReader类型的reader属性。在XmlBeanFactory中主要使用reader属性对资源文件进行读取和注册。

##### XmlBeanDefinitionReader

XML配置文件的读取是Spring巾重要的功能，因为Spring的大部分功能都是以配置作为切入，点的，那么我们可以从XmlBeanDefinitionReader中梳理一下资源文件读取、解析及注册的大致脉络，首先我们看看各个类的功能。

* ResourceLoader：定义资源加载器，主妥应用于根据给定的资源文件地址返回对应的Resource。
* BeanDefinitionReader：主要定义资源文件读取并转换为BeanDefinition的各个功能。
* EnvironmentCapable：定义获取Environment方法。
* DocumentLoader：定义从资源、文件加载到转换为Document的功能。
* AbstractBeanDefinitionReader：对EnvironmentCapable、BeanDefinitionReader类定义的功能进行实现
* BeanDefinitionDocumentReader：定义读取Docuemnt并注册BeanDefinition功能。
* BeanDefinitionParserDelegate：定义解析Element的各种方法。

经过以上分析，我们可以梳理出整个XML配置文件读取的大致流程，如图2-6所示，在XmlBeanDefinitionReader中主要包含以下几步的处理。



* 通过继承向AbstractBeanDefinitionReader中的方法，来使用ResourLoader将资源文件路径转换为对应的Resource文件。
* 通过DocumentLoaderX才Resource文件进行转换，将Resource文件转换为Document文件。
* 通过实现接口BeanDefinitionDocumentReader的DefaultBeanDefinitionDocumentReader类对Document进行解析，并使用BeanDefinitionParserDelegate对Element进行解析。

### 2.5 容器的基础XmlBeanFactory

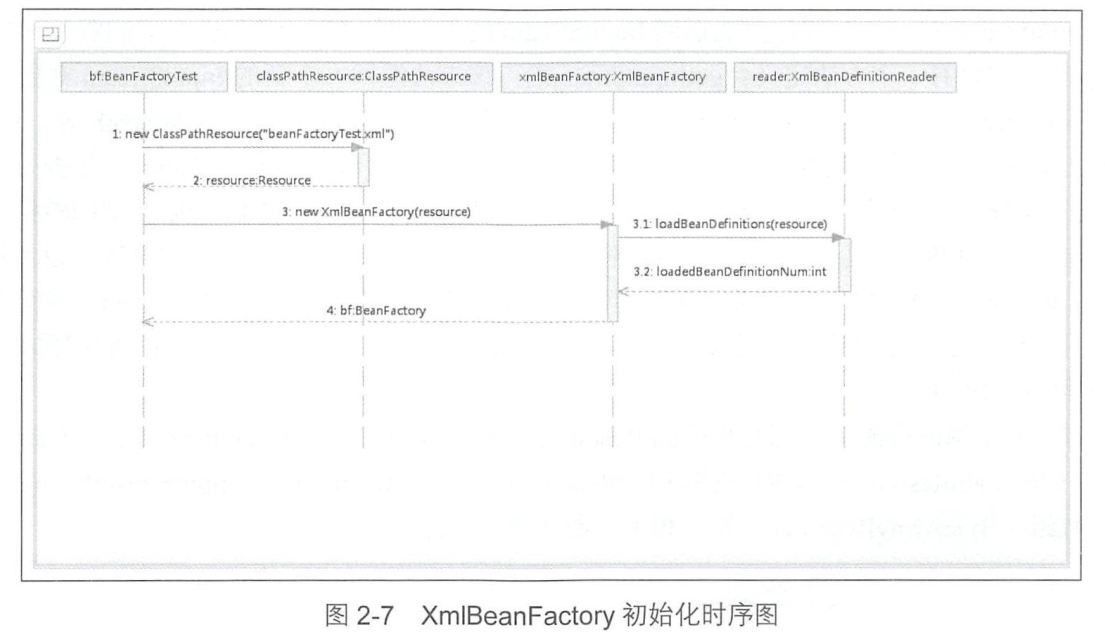
好了，到这里我们已经对Spring的容器功能有了大致的了解，尽管你可能还很迷糊，但是不要紧，接下来我们会详细探索每个步骤的实现。再次重申一下代码，我们接下来要深入分析以下功能的代码实现：

BeanFactory bf = new XmlBeanFactory(new ClassPathResource(”beanFactoryTest.xml”)),

通过XmlBeanFactory初始化时序图（如图2-7所示）我们来看一看上面代码的执行逻辑。

时序图从BeanFactoryTest测试类开始，通过时序图我们可以一目了然地看到整个逻辑处理顺序。在测试的BeanFactoryTest中首先调用ClassPathResource的构造函数来构造Resource资源文件的实例对象，这样后续的资源处理就可以用Resource提供的各种服务来操作了，当我们有了Resource后就可以进行XmlBeanFactory的初始化了。那么Resource资源是如何封装的呢？

#### 图2-7 XmlBeanFactory初始化时序图



#### 2.5.1 配置文件封装

Spring 的配置文件读取是通过 ClassPathResource 进行封装的，如

**new** ClassPathResource(**"com/prosayj/springboot /beanFactoryTest.xml"**) 那么 ClassPathResource 完成了什么功能呢？

在 Java 中，将不同来源的资源抽象成 URL，通过注册不同的 handler（URLStreamHandler）来处理不同来源的资源的读取逻辑，一般 handler 的类型使用不同前缀（协议， Protocol）来识别，如“file:”“http:”“jar:”等，然而 URL 没有默认定义相对 Classpath 或 ServletContext 等资源的 handler，虽然可以注册自己的 URLStreamHandler 来解析特定的 URL 前缀（协议），比如“classpath:”，然而这需要了解 URL 的实现机制，而且 URL 也没有提供基本的方法，如检查当前资源是否存在、检查当前资源是否可读等方法。因而 Spring 对其内部使用到的资源实现了自己的抽象结构： Resource 接口封装底层资源。

|  |
| --- |
| **package** org.springframework.core.io;  **public interface** InputStreamSource {  InputStream getInputStream() **throws** IOException; } |

|  |
| --- |
| **package** org.springframework.core.io; **public interface** Resource **extends** InputStreamSource { **boolean** exists(); **default boolean** isReadable() {**return** exists();} **default boolean** isOpen() {**return false**;} **default boolean** isFile() {**return false**;} URL getURL() **throws** IOException; URI getURI() **throws** IOException; File getFile() **throws** IOException; **default** ReadableByteChannel readableChannel() **throws** IOException {  **return** Channels.*newChannel*(getInputStream());  } **long** contentLength() **throws** IOException; **long** lastModified() **throws** IOException; Resource createRelative(String relativePath) **throws** IOException; String getFilename(); String getDescription(); } |

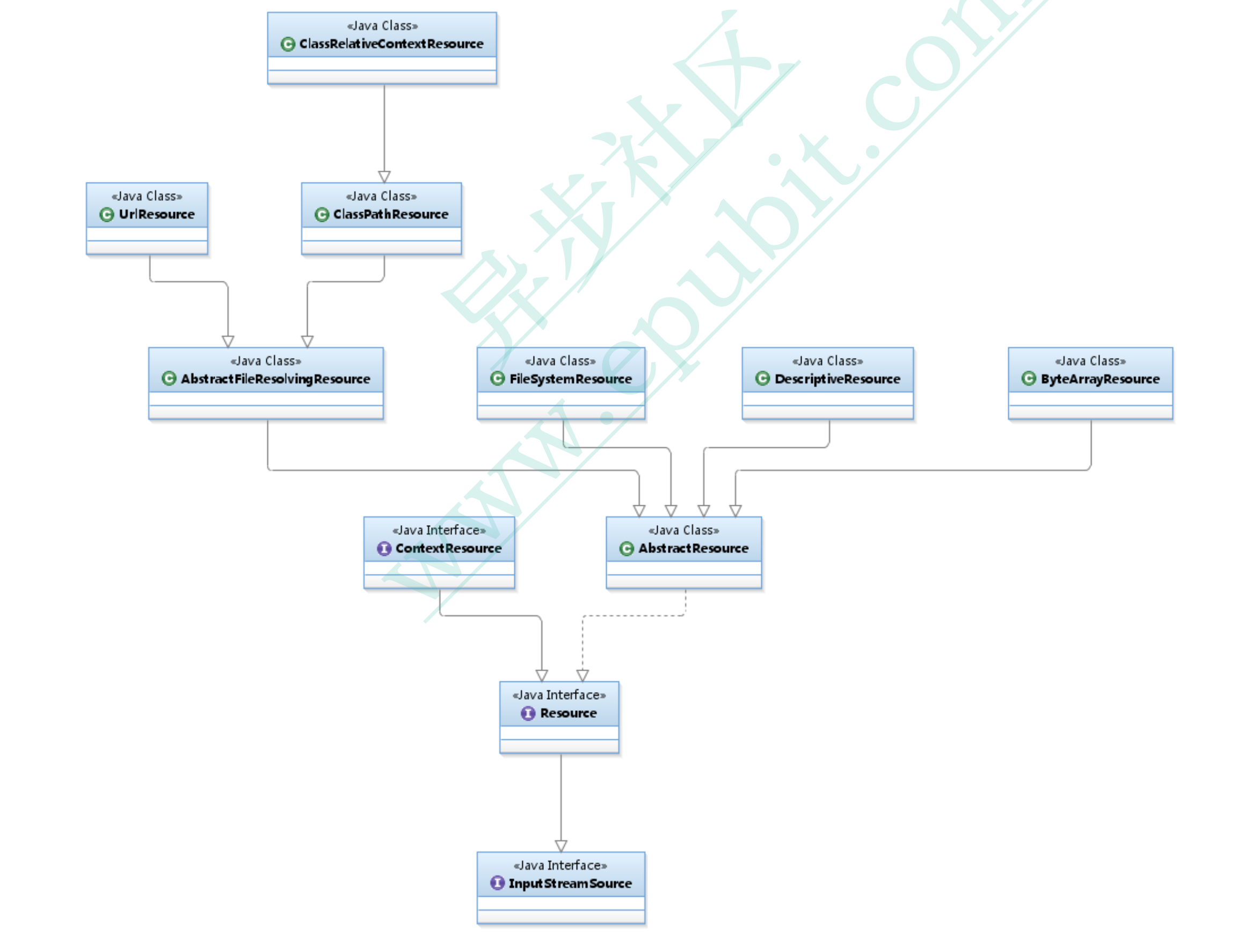
InputStreamSource 封装任何能返回 InputStream 的类，比如 File、 Classpath 下的资源和 Byte Array 等。它只有一个方法定义： getInputStream()，该方法返回一个新的 InputStream 对象。

Resource 接口抽象了所有 Spring 内部使用到的底层资源： File、 URL、 Classpath 等。

* 首先，它定义了 3 个判断当前资源状态的方法：存在性(exists)、可读性(isReadable)、是否处于打开状态（isOpen）。
* 另外， Resource 接口还提供了不同资源到 URL、 URI、 File 类型的转换，以及获取 lastModified 属性、文件名（不带路径信息的文件名， getFilename()）的方法。
* 为了便于操作，Resource 还提供了基于当前资源创建一个相对资源的方法： createRelative()。
* 在错误处理中需要详细地打印出错的资源文件，因而Resource还提供了getDescription()方法用来在错误处理中打印信息。

对不同来源的资源文件都有相应的 Resource 实现：

* 文件（FileSystemResource）、
* Classpath资源（ClassPathResource）、
* URL 资源（UrlResource）、
* InputStream 资源（InputStreamResource）、
* Byte 数组（ByteArrayResource）等。相关类图如图 2-8 所示。



##### 图 2-8 资源文件处理相关类图

在日常的开发工作中，资源文件的加载也是经常用到的，可以直接使用 Spring 提供的类，比如在希望加载文件时可以使用以下代码：

ClassPathResource resource = **new** ClassPathResource(**"beanFactoryTest.xml"**);  
InputStream inputStream = resource.getInputStream();

得到 inputStream 后，我们就可以按照以前的开发方式进行实现了，并且我们可以利用Resource 及其子类为我们提供的诸多特性。

有了 Resource 接口便可以对所有资源文件进行统一处理。至于实现，其实是非常简单的，以 getInputStream 为例，ClassPathResource 中的实现方式便是通过 class 或者 classLoader 提供的底层方法进行调用，而对于 FileSystemResource 的实现其实更简单，直接使用 FileInputStream对文件进行实例化。

|  |
| --- |
| ClassPathResource.java  **public** InputStream getInputStream() **throws** IOException {  InputStream is;  **if** (**this**.**clazz** != **null**) {  is = **this**.**clazz**.getResourceAsStream(**this**.**path**);  }  **else if** (**this**.**classLoader** != **null**) {  is = **this**.**classLoader**.getResourceAsStream(**this**.**path**);  }  **else** {  is = ClassLoader.*getSystemResourceAsStream*(**this**.**path**);  }  **if** (is == **null**) {  **throw new** FileNotFoundException(getDescription() + **" cannot be opened because it does not exist"**);  }  **return** is; } |

|  |
| --- |
| FileSystemResource.java  **public** InputStream getInputStream() **throws** IOException {  **try** {  **return** Files.*newInputStream*(**this**.**filePath**);  }  **catch** (NoSuchFileException ex) {  **throw new** FileNotFoundException(ex.getMessage());  } } |

当通过 Resource 相关类完成了对配置文件进行封装后配置文件的读取工作就全权交给XmlBeanDefinitionReader 来处理了。

了解了 Spring 中将配置文件封装为 Resource 类型的实例方法后，我们就可以继续探寻XmlBeanFactory 的初始化过程了， XmlBeanFactory 的初始化有若干办法， Spring 中提供了很多的构造函数，在这里分析的是使用 Resource 实例作为构造函数参数的办法，代码如下：

|  |
| --- |
| XmlBeanFactory.java  **public** XmlBeanFactory(Resource resource) **throws** BeansException {  **this**(resource, **null**); }  **public** XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) **throws** BeansException {  **super**(parentBeanFactory);// parentBeanFactory为父类BeanFactory用于factory合并，可以为空  **this**.**reader**.loadBeanDefinitions(resource); } |

上面函数中的代码**this**.**reader**.loadBeanDefinitions(resource);才是资源加载的真正实现，也是我们分析的重点之一。我们可以看到时序图中提到的XmlBeanDefinitionReader 加载数据就是在这里完成的，但是在 XmlBeanDefinitionReader 加载数据前还有一个调用父类构造函数初始化的过程：

|  |
| --- |
| **public** AbstractAutowireCapableBeanFactory(@Nullable BeanFactory parentBeanFactory) {  **this**();  setParentBeanFactory(parentBeanFactory); } |

super(parentBeanFactory)，跟踪代码到父类 AbstractAutowireCapableBeanFactory 的构造函数中：

AbstractAutowireCapableBeanFactory.java

public AbstractAutowireCapableBeanFactory() {

super();

ignoreDependencyInterface(BeanNameAware.class);

ignoreDependencyInterface(BeanFactoryAware.class);

ignoreDependencyInterface(BeanClassLoaderAware.class);

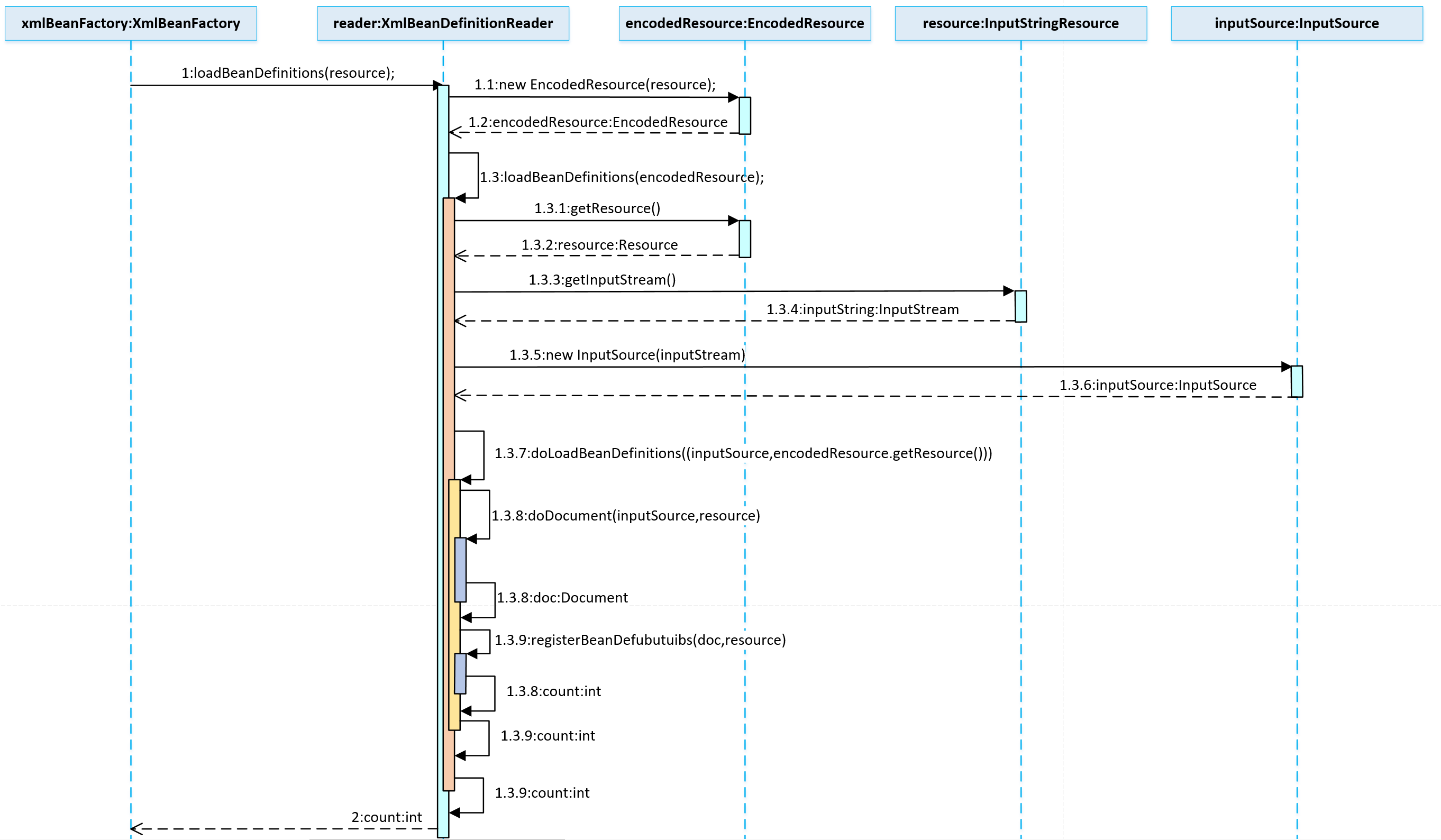
}

这里有必要提及 ignoreDependencyInterface 方法。 ignoreDependencyInterface 的主要功能是忽略给定接口的自动装配功能，那么，这样做的目的是什么呢？会产生什么样的效果呢？

举例来说，当 A 中有属性 B，那么当 Spring 在获取 A 的 Bean 的时候如果其属性 B 还没有初始化，那么 Spring 会自动初始化 B，这也是 Spring 中提供的一个重要特性。但是，某些情况下， B 不会被初始化，其中的一种情况就是 B 实现了 BeanNameAware 接口。 Spring 中是这样介绍的：自动装配时忽略给定的依赖接口，典型应用是通过其他方式解析 Application 上下文注册依赖，类似于 BeanFactory 通过 BeanFactoryAware 进行注入或者 ApplicationContext 通过ApplicationContextAware 进行注入。

#### 2.5.2 加载 Bean

之前提到的在 XmlBeanFactory 构造函数中调用了 XmlBeanDefinitionReader 类型的 reader属性提供的方法**this**.**reader**.loadBeanDefinitions(resource);，而这句代码则是整个资源加载的切入点，我们先来看看这个方法的时序图，如图 2-9 所示。



##### 图 2-9 loadBeanDefinitions 函数执行时序图

看到图 2-9 我们才知道什么叫山路十八弯，绕了这么半天还没有真正地切入正题，比如加载 XML 文档和解析注册 Bean，一直还在做准备工作。我们根据上面的时序图来分析一下这里究竟在准备什么？从上面的时序图中我们尝试梳理整个的处理过程如下。

* 封装资源文件。当进入 XmlBeanDefinitionReader 后首先对参数 Resource 使用EncodedResource 类进行封装。
* 获取输入流。从 Resource 中获取对应的 InputStream 并构造 InputSource。
* 通过构造的 InputSource 实例和 Resource 实例继续调用函数 doLoadBeanDefinitions。

我们来看一下 loadBeanDefinitions 函数具体的实现过程。

|  |
| --- |
| **public int** loadBeanDefinitions(Resource resource) **throws** BeanDefinitionStoreException {  **return** loadBeanDefinitions(**new** EncodedResource(resource)); } |

那么 EncodedResource 的作用是什么呢？通过名称，我们可以大致推断这个类主要是用于对资源文件的编码进行处理的。其中的主要逻辑体现在 getReader()方法中，当设置了编码属性的时候 Spring 会使用相应的编码作为输入流的编码。

|  |
| --- |
| **public** Reader getReader() **throws** IOException {  **if** (**this**.**charset** != **null**) {  **return new** InputStreamReader(**this**.**resource**.getInputStream(), **this**.**charset**);  }  **else if** (**this**.**encoding** != **null**) {  **return new** InputStreamReader(**this**.**resource**.getInputStream(), **this**.**encoding**);  }  **else** {  **return new** InputStreamReader(**this**.**resource**.getInputStream());  } } |

上面代码构造了一个有编码（encoding）的 InputStreamReader。当构造好 encodedResource对象后，再次转入了可复用方法 loadBeanDefinitions(new EncodedResource(resource))。这个方法内部才是真正的数据准备阶段，也就是时序图所描述的逻辑：

|  |
| --- |
| **public int** loadBeanDefinitions(EncodedResource encodedResource) **throws** BeanDefinitionStoreException {  Assert.*notNull*(encodedResource, **"EncodedResource must not be null"**);  **if** (**logger**.isTraceEnabled()) {  **logger**.trace(**"Loading XML bean definitions from "** + encodedResource);  }  //通过属性来记录已经加载的资源  Set<EncodedResource> currentResources = **this**.**resourcesCurrentlyBeingLoaded**.get();  **if** (currentResources == **null**) {  currentResources = **new** HashSet<>(4);  **this**.**resourcesCurrentlyBeingLoaded**.set(currentResources);  }  **if** (!currentResources.add(encodedResource)) {  **throw new** BeanDefinitionStoreException(  **"Detected cyclic loading of "** + encodedResource + **" - check your import definitions!"**);  }  **try** {  //从 encodedResource 中获取已经封装的 Resource 对象并再次从 Resource 中获取其中的 inputStream  InputStream inputStream = encodedResource.getResource().getInputStream();  **try** {  //InputSource 这个类并不来自于 Spring，它的全路径是 org.xml.sax.InputSource  InputSource inputSource = **new** InputSource(inputStream);  **if** (encodedResource.getEncoding() != **null**) {  inputSource.setEncoding(encodedResource.getEncoding());  }  //真正进入了逻辑核心部分  **return** doLoadBeanDefinitions(inputSource, encodedResource.getResource());  }  **finally** {  inputStream.close();//关闭输入流  }  }  **catch** (IOException ex) {  **throw new** BeanDefinitionStoreException(  **"IOException parsing XML document from "** + encodedResource.getResource(), ex);  }  **finally** {  currentResources.remove(encodedResource);  **if** (currentResources.isEmpty()) {  **this**.**resourcesCurrentlyBeingLoaded**.remove();  }  } } |

我们再次整理数据准备阶段的逻辑，首先对传入的 resource 参数做封装，目的是考虑到Resource 可能存在编码要求的情况，其次， 通过 SAX 读取 XML 文件的方式来准备 InputSource对象，最后将准备的数据通过参数传入真正的核心处理部分 doLoadBeanDefinitions(inputSource, encodedResource.getResource())。

|  |
| --- |
| **protected int** doLoadBeanDefinitions(InputSource inputSource, Resource resource)  **throws** BeanDefinitionStoreException {  **try** {  Document doc = doLoadDocument(inputSource, resource);  **int** count = registerBeanDefinitions(doc, resource);  **if** (**logger**.isDebugEnabled()) {  **logger**.debug(**"Loaded "** + count + **" bean definitions from "** + resource);  }  **return** count;  }  **catch** (BeanDefinitionStoreException ex) {  **throw** ex;  }  **catch** (SAXParseException ex) {  **throw new** XmlBeanDefinitionStoreException(resource.getDescription(),  **"Line "** + ex.getLineNumber() + **" in XML document from "** + resource + **" is invalid"**, ex);  }  **catch** (SAXException ex) {  **throw new** XmlBeanDefinitionStoreException(resource.getDescription(),  **"XML document from "** + resource + **" is invalid"**, ex);  }  **catch** (ParserConfigurationException ex) {  **throw new** BeanDefinitionStoreException(resource.getDescription(),  **"Parser configuration exception parsing XML from "** + resource, ex);  }  **catch** (IOException ex) {  **throw new** BeanDefinitionStoreException(resource.getDescription(),  **"IOException parsing XML document from "** + resource, ex);  }  **catch** (Throwable ex) {  **throw new** BeanDefinitionStoreException(resource.getDescription(),  **"Unexpected exception parsing XML document from "** + resource, ex);  } } |

|  |
| --- |
| **protected** Document doLoadDocument(InputSource inputSource, Resource resource) **throws** Exception {  **return this**.**documentLoader**.loadDocument(inputSource, getEntityResolver(), **this**.**errorHandler**,  getValidationModeForResource(resource), isNamespaceAware()); } |

在上面冗长的代码中假如不考虑异常类的代码，其实只做了三件事，这三件事的每一件都必不可少。

* 获取对 XML 文件的验证模式。
* 加载 XML 文件，并得到对应的 Document。
* 根据返回的 Document 注册 Bean 信息。

这 3 个步骤支撑着整个 Spring 容器部分的实现，尤其是第 3 步对配置文件的解析，逻辑非常的复杂，我们先从获取 XML 文件的验证模式讲起。

### 2.6 获取 XML 的验证模式

了解 XML 文件的读者都应该知道 XML 文件的验证模式保证了 XML 文件的正确性，而比较常用的验证模式有两种： DTD 和 XSD。它们之间有什么区别呢？

#### 2.6.1 DTD 与 XSD 区别

DTD（Document Type Definition）即文档类型定义，是一种 XML 约束模式语言，是 XML文件的验证机制，属于 XML 文件组成的一部分。 DTD 是一种保证 XML 文档格式正确的有效方法，可以通过比较 XML 文档和 DTD 文件来看文档是否符合规范，元素和标签使用是否正确。一个 DTD 文档包含：元素的定义规则，元素间关系的定义规则，元素可使用的属性，可使用的实体或符号规则。

要使用 DTD 验证模式的时候需要在 XML 文件的头部声明，以下是在 Spring 中使用 DTD声明方式的代码：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//Spring//DTD BEAN 2.0//EN" "http://www.Springframework.org/

dtd/ Spring-beans-2.0.dtd">

<beans>

... ...

</beans>

而以 Spring 为例，具体的 Spring-beans-2.0.dtd 部分如下：

<!ELEMENT beans (

description?,

(import | alias | bean)\*

)>

<!ATTLIST beans default-lazy-init (true | false) "false">

<!ATTLIST beans default-merge (true | false) "false">

<!ATTLIST beans default-autowire (no | byName | byType | constructor|autodetect)"no">

<!ATTLIST beans default-dependency-check (none | objects | simple | all) "none">

<!ATTLIST beans default-init-method CDATA #IMPLIED>

<!ATTLIST beans default-destroy-method CDATA #IMPLIED>

... ...

XML Schema 语言就是 XSD（XML Schemas Definition）。 XML Schema 描述了 XML 文档的结构。可以用一个指定的 XML Schema 来验证某个 XML 文档，以检查该 XML 文档是否符合其要求。文档设计者可以通过 XML Schema 指定 XML 文档所允许的结构和内容，并可据此检查 XML 文档是否是有效的。 XML Schema 本身是 XML 文档，它符合 XML 语法结构。可以用通用的 XML 解析器解析它。

在使用 XML Schema 文档对 XML 实例文档进行检验，除了要声明名称空间外

（xmlns=http://www.Springframework.org/schema/beans），还必须指定该名称空间所对应的 XML Schema 文

档的存储位置。通过 schemaLocation 属性来指定名称空间所对应的 XML Schema 文档的存储位置，它包含两个部分，一部分是名称空间的 URI，另一部分就是该名称空间所标识的 XML Schema文件位置或 URL 地址（xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.

Springframework.org/schema/beans/Spring-beans.xsd）。

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd"**>   <**bean id="myTestBean" class="com.prosayj.springboot.spring源码深度剖析.\_02\_容器的基本实现.MyTestBean"**></**bean**> </**beans**> |

Spring-beans-3.0.xsd 部分代码如下：

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.Springframework.org/schema/beans"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.Springframework.org/schema/beans">

<xsd:import namespace="http://www.w3.org/XML/1998/namespace"/>

<xsd:annotation>

<xsd:documentation><![CDATA[

... ...

]]></xsd:documentation>

</xsd:annotation>

<!-- base types -->

<xsd:complexType name="identifiedType" abstract="true">

<xsd:annotation>

<xsd:documentation><![CDATA[

The unique identifier for a bean. The scope of the identifier is the enclosing bean factory.

]]></xsd:documentation>

</xsd:annotation>

<xsd:attribute name="id" type="xsd:ID">

<xsd:annotation>

<xsd:documentation><![CDATA[

The unique identifier for a bean.

]]></xsd:documentation>

</xsd:annotation>

</xsd:attribute>

</xsd:complexType>

... ...

</xsd:schema>

我们只是简单地介绍一下 XML 文件的验证模式的相关知识，目的在于让读者对后续知识的理解能有连续性，如果对 XML 有兴趣的读者可以进一步查阅相关资料。

#### 2.6.2 验证模式的读取

了解了 DTD 与 XSD 的区别后我们再去分析 Spring 中对于验证模式的提取就更容易理解了。通过之前的分析我们锁定了 Spring 通过 getValidationModeForResource 方法来获取对应资源的的验证模式。

|  |
| --- |
| **protected int** getValidationModeForResource(Resource resource) {  **int** validationModeToUse = getValidationMode();  //如果手动指定了验证模式则使用指定的验证模式  **if** (validationModeToUse != ***VALIDATION\_AUTO***) {  **return** validationModeToUse;  }  //如果未指定则使用自动检测  **int** detectedMode = detectValidationMode(resource);  **if** (detectedMode != ***VALIDATION\_AUTO***) {  **return** detectedMode;  }  *// Hmm, we didn't get a clear indication... Let's assume XSD,  // since apparently no DTD declaration has been found up until  // detection stopped (before finding the document's root tag).* **return *VALIDATION\_XSD***; } |

方法的实现其实还是很简单的，无非是如果设定了验证模式则使用设定的验证模式（可以通过对调用 XmlBeanDefinitionReader 中的 setValidationMode 方法进行设定），否则使用自动检测的方式。而自动检测验证模式的功能是在函数 detectValidationMode 方法中实现的，在detectValidationMode 函数中又将自动检测验证模式的工作委托给了专门处理类 XmlValidationModeDetector，调用了 XmlValidationModeDetector 的 validationModeDetector 方法，具体代码如下：

|  |
| --- |
| **protected int** detectValidationMode(Resource resource) {  **if** (resource.isOpen()) {  **throw new** BeanDefinitionStoreException(  **"Passed-in Resource ["** + resource + **"] contains an open stream: "** +  **"cannot determine validation mode automatically. Either pass in a Resource "** +  **"that is able to create fresh streams, or explicitly specify the validationMode "** +  **"on your XmlBeanDefinitionReader instance."**);  }  InputStream inputStream;  **try** {  inputStream = resource.getInputStream();  }  **catch** (IOException ex) {  **throw new** BeanDefinitionStoreException(  **"Unable to determine validation mode for ["** + resource + **"]: cannot open InputStream. "** +  **"Did you attempt to load directly from a SAX InputSource without specifying the "** +  **"validationMode on your XmlBeanDefinitionReader instance?"**, ex);  }  **try** {  **return this**.**validationModeDetector**.detectValidationMode(inputStream);  }  **catch** (IOException ex) {  **throw new** BeanDefinitionStoreException(**"Unable to determine validation mode for ["** +  resource + **"]: an error occurred whilst reading from the InputStream."**, ex);  } } |

|  |
| --- |
| XmlValidationModeDetector.java  **public int** detectValidationMode(InputStream inputStream) **throws** IOException {  *// Peek into the file to look for DOCTYPE.* BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(inputStream));  **try** {  **boolean** isDtdValidated = **false**;  String content;  **while** ((content = reader.readLine()) != **null**) {  content = consumeCommentTokens(content);  **if** (**this**.**inComment** || !StringUtils.*hasText*(content)) {  **continue**;  }  **if** (hasDoctype(content)) {  isDtdValidated = **true**;  **break**;  }  **if** (hasOpeningTag(content)) {  *// End of meaningful data...* **break**;  }  }  **return** (isDtdValidated ? ***VALIDATION\_DTD*** : ***VALIDATION\_XSD***);  }  **catch** (CharConversionException ex) {  *// Choked on some character encoding...  // Leave the decision up to the caller.* **return *VALIDATION\_AUTO***;  }  **finally** {  reader.close();  } } |

只要我们理解了 XSD 与 DTD 的使用方法，理解上面的代码应该不会太难， Spring 用来检测验证模式的办法就是判断是否包含 DOCTYPE，如果包含就是 DTD，否则就是 XSD。

### 2.7 获取 Document

经过了验证模式准备的步骤就可以进行 Document 加载了，同样 XmlBeanFactoryReader 类对于文档读取并没有亲力亲为，而是委托给了 DocumentLoader 去执行，这里的 DocumentLoader是个接口，而真正调用的是 DefaultDocumentLoader，解析代码如下：

|  |
| --- |
| DefaultDocumentLoader.java  **public** Document loadDocument(InputSource inputSource, EntityResolver entityResolver,  ErrorHandler errorHandler, **int** validationMode, **boolean** namespaceAware) **throws** Exception {  DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode, namespaceAware);  **if** (***logger***.isTraceEnabled()) {  ***logger***.trace(**"Using JAXP provider ["** + factory.getClass().getName() + **"]"**);  }  DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);  **return** builder.parse(inputSource); } |

对于这部分代码其实并没有太多可以描述的，因为通过 SAX 解析 XML 文档的套路大致都差不多， Spring 在这里并没有什么特殊的地方，同样首先创建 DocumentBuilderFactory，再通过 DocumentBuilderFactory 创建 DocumentBuilder，进而解析 inputSource 来返回 Document 对象。对此感兴趣的读者可以在网上获取更多的资料。这里有必要提及一下 EntityResolver，对于参数entityResolver，传入的是通过 getEntityResolver()函数获取的返回值，如下代码：

|  |
| --- |
| **protected** EntityResolver getEntityResolver() {  **if** (**this**.**entityResolver** == **null**) {  *// Determine default EntityResolver to use.* ResourceLoader resourceLoader = getResourceLoader();  **if** (resourceLoader != **null**) {  **this**.**entityResolver** = **new** ResourceEntityResolver(resourceLoader);  }  **else** {  **this**.**entityResolver** = **new** DelegatingEntityResolver(getBeanClassLoader());  }  }  **return this**.**entityResolver**; } |

那么， EntityResolver 到底是做什么用的呢？

#### 2.7.1 EntityResolver 用法

在 loadDocument 方法中涉及一个参数 EntityResolver，何为 EntityResolver？官网这样解释:如果 SAX 应用程序需要实现自定义处理外部实体，则必须实现此接口并使用 setEntityResolver方法向 SAX 驱动器注册一个实例。也就是说，对于解析一个 XML， SAX 首先读取该 XML 文档上的声明，根据声明去寻找相应的 DTD 定义，以便对文档进行一个验证。默认的寻找规则，即通过网络（实现上就是声明的 DTD 的 URI 地址）来下载相应的 DTD 声明，并进行认证。下载的过程是一个漫长的过程，而且当网络中断或不可用时，这里会报错，就是因为相应的 DTD

声明没有被找到的原因。

EntityResolver 的作用是项目本身就可以提供一个如何寻找 DTD 声明的方法，即由程序来实现寻找 DTD 声明的过程，比如我们将 DTD 文件放到项目中某处，在实现时直接将此文档读取并返回给 SAX 即可。这样就避免了通过网络来寻找相应的声明。

首先看 entityResolver 的接口方法声明：

|  |
| --- |
| **public abstract** InputSource resolveEntity (String publicId, String systemId) **throws** SAXException, IOException; |

这里，它接收两个参数 publicId 和 systemId，并返回一个 inputSource 对象。这里我们以特定配置文件来进行讲解。

1．如果我们在解析验证模式为 XSD 的配置文件，代码如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.Springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.Springframework.org/schema/beans http://www.springframework.org/schema/beans/Spring-beans.xsd"**>  ... ... </**beans**> |

读取到以下两个参数。

* publicId： null
* systemId： <http://www.springframework.org/schema/beans/Spring-beans.xsd>

2．如果我们在解析验证模式为 DTD 的配置文件，代码如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* **<!DOCTYPE beans PUBLIC "-//Spring//DTD BEAN 2.0//EN" "http://www.Springframework. org/ dtd/Spring-beans-2.0.dtd"*>*** <**beans**>  ... ... </**beans**> |

读取到以下两个参数。

* publicId： -//Spring//DTD BEAN 2.0//EN
* systemId： <http://www.springframework.org/dtd/Spring-beans-2.0.dtd>

之前已经提到过，验证文件默认的加载方式是通过 URL 进行网络下载获取，这样会造成延迟，用户体验也不好，一般的做法都是将验证文件放置在自己的工程里，那么怎么做才能将这个 URL 转换为自己工程里对应的地址文件呢？我们以加载 DTD 文件为例来看看 Spring 中是如何实现的。根据之前 Spring 中通过 getEntityResolver()方法对 EntityResolver 的获取，我们知道， Spring 中使用 DelegatingEntityResolver 类为 EntityResolver 的实现类， resolveEntity 实现方法如下：

|  |
| --- |
| DelegatingEntityResolver.java  **public** InputSource resolveEntity(String publicId, @Nullable String systemId) **throws** SAXException, IOException {  **if** (systemId != **null**) {  **if** (systemId.endsWith(***DTD\_SUFFIX***)) {  //如果是 dtd 从这里解析  **return this**.**dtdResolver**.resolveEntity(publicId, systemId);  }  **else if** (systemId.endsWith(***XSD\_SUFFIX***)) {  //通过调用 META-INF/Spring.schemas 解析  **return this**.**schemaResolver**.resolveEntity(publicId, systemId);  }  }  **return null**; } |

我们可以看到，对不同的验证模式， Spring 使用了不同的解析器解析。这里简单描述一下原理，比如加载 DTD类型的 BeansDtdResolver的 resolveEntity是直接截取 systemId最后的 xx.dtd然后去当前路径下寻找，而加载 XSD 类型的 PluggableSchemaResolver 类的 resolveEntity 是默认到 META-INF/Spring.schemas 文件中找到 systemid 所对应的 XSD 文件并加载。

|  |
| --- |
| BeansDtdResolver.java  **public** InputSource resolveEntity(String publicId, @Nullable String systemId) **throws** IOException {  **if** (***logger***.isTraceEnabled()) {  ***logger***.trace(**"Trying to resolve XML entity with public ID ["** + publicId +  **"] and system ID ["** + systemId + **"]"**);  }  **if** (systemId != **null** && systemId.endsWith(***DTD\_EXTENSION***)) {  **int** lastPathSeparator = systemId.lastIndexOf(**'/'**);  **int** dtdNameStart = systemId.indexOf(***DTD\_NAME***, lastPathSeparator);  **if** (dtdNameStart != -1) {  String dtdFile = ***DTD\_NAME*** + ***DTD\_EXTENSION***;  **if** (***logger***.isTraceEnabled()) {  ***logger***.trace(**"Trying to locate ["** + dtdFile + **"] in Spring jar on classpath"**);  }  **try** {  Resource resource = **new** ClassPathResource(dtdFile, getClass());  InputSource source = **new** InputSource(resource.getInputStream());  source.setPublicId(publicId);  source.setSystemId(systemId);  **if** (***logger***.isTraceEnabled()) {  ***logger***.trace(**"Found beans DTD ["** + systemId + **"] in classpath: "** + dtdFile);  }  **return** source;  }  **catch** (IOException ex) {  **if** (***logger***.isDebugEnabled()) {  ***logger***.debug(**"Could not resolve beans DTD ["** + systemId + **"]: not found in classpath"**, ex);  }  }  }  }  *// Use the default behavior -> download from website or wherever.* **return null**; } |

|  |
| --- |
| **PluggableSchemaResolver.java**  **public** InputSource resolveEntity(String publicId, @Nullable String systemId) **throws** IOException {  **if** (***logger***.isTraceEnabled()) {  ***logger***.trace(**"Trying to resolve XML entity with public id ["** + publicId +  **"] and system id ["** + systemId + **"]"**);  }   **if** (systemId != **null**) {  String resourceLocation = getSchemaMappings().get(systemId);  **if** (resourceLocation != **null**) {  Resource resource = **new** ClassPathResource(resourceLocation, **this**.**classLoader**);  **try** {  InputSource source = **new** InputSource(resource.getInputStream());  source.setPublicId(publicId);  source.setSystemId(systemId);  **if** (***logger***.isTraceEnabled()) {  ***logger***.trace(**"Found XML schema ["** + systemId + **"] in classpath: "** + resourceLocation);  }  **return** source;  }  **catch** (FileNotFoundException ex) {  **if** (***logger***.isDebugEnabled()) {  ***logger***.debug(**"Could not find XML schema ["** + systemId + **"]: "** + resource, ex);  }  }  }  }  **return null**; } |

### 2.8 解析及注册 BeanDefinitions

当把文件转换为 Document 后，接下来的提取及注册 bean 就是我们的重头戏。继续上面的分析，当程序已经拥有 XML 文档文件的 Document 实例对象时，就会被引入下面这个方法。

|  |
| --- |
| XmlBeanDefinitionReader.java  **public int** registerBeanDefinitions(Document doc, Resource resource) **throws** BeanDefinitionStoreException {  //使用 DefaultBeanDefinitionDocumentReader 实例化 BeanDefinitionDocumentReader  //在实例化 BeanDefinitionReader 时候会将 BeanDefinitionRegistry 传入，默认使用继承自  DefaultListableBeanFactory 的子类  BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  //记录统计前 BeanDefinition 的加载个数  **int** countBefore = getRegistry().getBeanDefinitionCount();  //将环境变量设置其中以后，加载及注册 bean  documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  **return** getRegistry().getBeanDefinitionCount() - countBefore;//记录本次加载的 BeanDefinition 个数  } |

其中的参数 doc 是通过上一节 loadDocument 加载转换出来的。在这个方法中很好地应用了面向对象中单一职责的原则，将逻辑处理委托给单一的类进行处理，而这个逻辑处理类就是BeanDefinitionDocumentReader。 BeanDefinitionDocumentReader 是一个接口，而实例化的工作是在createBeanDefinitionDocumentReader()中完成的，而通过此方法， BeanDefinitionDocumentReader 真正的类型其实已经是

DefaultBeanDefinitionDocumentReader 了，进入 DefaultBeanDefinitionDocumentReader 后，发现这个方法的重要目的之一就是提取 root，以便于再次将 root 作为参数继续BeanDefinition 的注册。

|  |
| --- |
| **public void** registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  **this**.**readerContext** = readerContext;  doRegisterBeanDefinitions(doc.getDocumentElement()); } |

经过艰难险阻，磕磕绊绊，我们终于到了核心逻辑的底部doRegisterBeanDefinitions(Element root) ，至少我们在这个方法中看到了希望。如果说以前一直是 XML 加载解析的准备阶段，那么 doRegisterBeanDefinitions 算是真正地开始进行解析了，我们期待的核心部分真正开始了。

|  |
| --- |
| **protected void** doRegisterBeanDefinitions(Element root) {  *// Any nested <beans> elements will cause recursion in this method. In  // order to propagate and preserve <beans> default-\* attributes correctly,  // keep track of the current (parent) delegate, which may be null. Create  // the new (child) delegate with a reference to the parent for fallback purposes,  // then ultimately reset this.delegate back to its original (parent) reference.  // this behavior emulates a stack of delegates without actually necessitating one.*  //专门处理解析  BeanDefinitionParserDelegate parent = **this**.**delegate**;  **this**.**delegate** = createDelegate(getReaderContext(), root, parent);   **if** (**this**.**delegate**.isDefaultNamespace(root)) {  //处理 profile 属性  String profileSpec = root.getAttribute(***PROFILE\_ATTRIBUTE***);  **if** (StringUtils.*hasText*(profileSpec)) {  String[] specifiedProfiles = StringUtils.*tokenizeToStringArray*(  profileSpec, BeanDefinitionParserDelegate.***MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS***);  *// We cannot use Profiles.of(...) since profile expressions are not supported  // in XML config. See SPR-12458 for details.* **if** (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {  **if** (**logger**.isDebugEnabled()) {  **logger**.debug(**"Skipped XML bean definition file due to specified profiles ["** + profileSpec +  **"] not matching: "** + getReaderContext().getResource());  }  **return**;  }  }  }  //解析前处理，留给子类实现  preProcessXml(root);  parseBeanDefinitions(root, **this**.**delegate**);  //解析后处理，留给子类实现  postProcessXml(root);  **this**.**delegate** = parent; } |

通过上面的代码我们看到了处理流程，首先是对 profile 的处理，然后开始进行解析，可是当我们跟进 preProcessXml(root)或者 postProcessXml(root)发现代码是空的，既然是空的写着还有什么用呢？就像面向对象设计方法学中常说的一句话，一个类要么是面向继承的设计的，要么就用 final 修饰。在

DefaultBeanDefinitionDocumentReader 中并没有用 final 修饰，所以它是面向继承而设计的。这两个方法正是为子类而设计的，如果读者有了解过设计模式，可以很快速地反映出这是模版方法模式，如果继承自

DefaultBeanDefinitionDocumentReader 的子类需要在 Bean 解析前后做一些处理的话，那么只需要重写这两个方法就可以了。

#### 2.8.1 profile 属性的使用

我们注意到在注册 Bean 的最开始是对 PROFILE\_ATTRIBUTE 属性的解析，可能对于我们来说， profile 属性并不是很常用。让我们先了解一下这个属性。分析 profile 前我们先了解下 profile 的用法，官方示例代码片段如下：

|  |
| --- |
| <**beans xmlns="http://www.Springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**  **xmlns:jdbc=**[**http://www.Springframework.org/schema/jdbc**](http://www.Springframework.org/schema/jdbc)  **xmlns:jee="http://www.springframework.org/schema/jee"  xsi:schemaLocation="..."**>  ... ...  <**beans profile="dev"**>  ... ...  </**beans**>  <**beans profile="production"**>  ... ...  </**beans**> </**beans**> |

集成到 Web 环境中时，在 web.xml 中加入以下代码：

|  |
| --- |
| <**context-param**>  <**param-name**>Spring.profiles.active</**param-name**>  <**param-value**>dev</**param-value**> </**context-param**> |

有了这个特性我们就可以同时在配置文件中部署两套配置来适用于生产环境和开发环境，这样可以方便的进行切换开发、部署环境，最常用的就是更换不同的数据库。

了解了 profile 的使用再来分析代码会清晰得多，首先程序会获取 beans 节点是否定义了profile 属性，如果定义了则会需要到环境变量中去寻找，所以这里首先断言 environment 不可能为空，因为 profile 是可以同时指定多个的，需要程序对其拆分，并解析每个 profile 是都符合环境变量中所定义的，不定义则不会浪费性能去解析。

#### 2.8.2 解析并注册 BeanDefinition

处理了 profile 后就可以进行 XML 的读取了，跟踪代码进入 parseBeanDefinitions(root,this.delegate)。

|  |
| --- |
| **DefaultBeanDefinitionDocumentReader.java**  **protected void** parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  //对 beans 的处理  **if** (delegate.isDefaultNamespace(root)) {  NodeList nl = root.getChildNodes();  **for** (**int** i = 0; i < nl.getLength(); i++) {  Node node = nl.item(i);  **if** (node **instanceof** Element) {  Element ele = (Element) node;  **if** (delegate.isDefaultNamespace(ele)) {  //对 beans 的处理  parseDefaultElement(ele, delegate);  }  **else** {  //对 bean 的处理  delegate.parseCustomElement(ele);  }  }  }  }  **else** {  delegate.parseCustomElement(root);  } } |

上面的代码看起来逻辑还是蛮清晰的，因为在 Spring 的 XML 配置里面有两大类 Bean 声明，一个是默认的，如：

<**bean id="test" class="test.TestBean"**/>

另一类就是自定义的，如：

<**tx:annotation-driven**/>

而两种方式的读取及解析差别是非常大的，如果采用 Spring 默认的配置， Spring 当然知道该怎么做，但是如果是自定义的，那么就需要用户实现一些接口及配置了。对于根节点或者子节点如果是默认命名空间的话则采用 parseDefaultElement 方法进行解析，否则使用delegate.parseCustomElement 方法对自定义命名空间进行解析。而判断是否默认命名空间还是自定义命名空间的办法其实是使用 node.getNamespaceURI()获取命名空间，并与 Spring 中固定的命名空间 <http://www.springframework.org/schema/beans>进行比对。如果一致则认为是默认，否则就认为是自定义。而对于默认标签解析与自定义标签解析我们将会在下一章中进行讨论。

## 第 5 章 bean 的加载

经过前面的分析 ， 我们终于结束了对 XML 配置文件的解析，接下来将会面临更大的挑战，就是对 bean 加载的探索。 bean 加载的功能实现远比 bean 的解析要复杂得多，同样，我们还是以本书开篇的示例为基础 ，对于加载 bean 的功能，在 Spring中的调用方式为：

MyTestBean bean = (MyTestBean) beanFactory.getBean(**"myTestBean"**);

这句代码实现了什么样的功能呢？ 我们可以先快速体验一下 Spring 中代码是如何实现的。

|  |
| --- |
| **public** Object getBean(String name) **throws** BeansException {  **return** doGetBean(name, **null**, **null**, **false**); } |

|  |
| --- |
| **protected** <T> T doGetBean(**final** String name, @Nullable **final** Class<T> requiredType,  @Nullable **final** Object[] args, **boolean** typeCheckOnly) **throws** BeansException {  //提取对应的beanName  **final** String beanName = transformedBeanName(name);  Object bean;  */\*\*  \* 检查缓存中或者实例工厂中是否有对应的实例  \* 会为什么首先会使用这段代码呢？  \* 因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，  \* Spring创建bean的原则是不等bean创建完成就将创建bean的ObjectFactory提早曝光  \* 也就是将ObjctFactory加入到缓存中，一旦下一个bean创建的时候要依赖上一个bean则直接使用ObjectFactory  \*/*  *// Eagerly check singleton cache for manually registered singletons.  // 直接尝试从缓存获取或者singletonFactories中的ObjectFactory中获取。*  Object sharedInstance = getSingleton(beanName);  **if** (sharedInstance != **null** && args == **null**) {  **if** (**logger**.isTraceEnabled()) {  **if** (isSingletonCurrentlyInCreation(beanName)) {  **logger**.trace(**"Returning eagerly cached instance of singleton bean '"** + beanName +  **"' that is not fully initialized yet - a consequence of a circular reference"**);  }  **else** {  **logger**.trace(**"Returning cached instance of singleton bean '"** + beanName + **"'"**);  }  }  //返回对应的实例，有时候存在诸如BeanFactory的情况下并不是直拨返回实例本身而是返回指定方法返回的实例  bean = getObjectForBeanInstance(sharedInstance, name, beanName, **null**);  }   **else** {  *// Fail if we're already creating this bean instance:  // We're assumably within a circular reference.*  *//只有在单例情况才会尝试解决循环依赖，原型模式情况下，如果存在*  *//A中有B属性，B中有A属性，那么当依赖注入的时候，就会产生当A还未创建完的候因为*  *//对于B的创边再次返回创A，造成循环依赖，也就是下面的情况*  *//isPrototypeCurrentlyInCreation(beanName)为true* **if** (isPrototypeCurrentlyInCreation(beanName)) {  **throw new** BeanCurrentlyInCreationException(beanName);  }   *// Check if bean definition exists in this factory.* BeanFactory parentBeanFactory = getParentBeanFactory();  // 如果beanDefinitionMap中也就是在所有已加载的类张不包括beanName则尝试从parentBeanFactory中检测。  **if** (parentBeanFactory != **null** && !containsBeanDefinition(beanName)) {  *// Not found -> check parent.*  *//递归到BeanFactory中寻找* String nameToLookup = originalBeanName(name);  **if** (parentBeanFactory **instanceof** AbstractBeanFactory) {  **return** ((AbstractBeanFactory) parentBeanFactory).doGetBean(  nameToLookup, requiredType, args, typeCheckOnly);  }  **else if** (args != **null**) {  *// Delegation to parent with explicit args.* **return** (T) parentBeanFactory.getBean(nameToLookup, args);  }  **else if** (requiredType != **null**) {  *// No args -> delegate to standard getBean method.* **return** parentBeanFactory.getBean(nameToLookup, requiredType);  }  **else** {  **return** (T) parentBeanFactory.getBean(nameToLookup);  }  }  //如果不是仅仅做类型检查则是创建bean，这里要进行记录  **if** (!typeCheckOnly) {  markBeanAsCreated(beanName);  }   **try** {  //将存储XML配置文件的GernericBeanDefinition抓换为RootBeanDefinition,如果指定BeanName是子Bean的话同时会合并父类的相关属性。  **final** RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);  checkMergedBeanDefinition(mbd, beanName, args);   *// Guarantee initialization of beans that the current bean depends on.* String[] dependsOn = mbd.getDependsOn();  //若存在依赖则需要递归实例化依赖的bean  **if** (dependsOn != **null**) {  **for** (String dep : dependsOn) {  **if** (isDependent(beanName, dep)) {  **throw new** BeanCreationException(mbd.getResourceDescription(), beanName,  **"Circular depends-on relationship between '"** + beanName + **"' and '"** + dep + **"'"**);  }  //缓存依赖调用  registerDependentBean(dep, beanName);  **try** {  getBean(dep);  }  **catch** (NoSuchBeanDefinitionException ex) {  **throw new** BeanCreationException(mbd.getResourceDescription(), beanName,  **"'"** + beanName + **"' depends on missing bean '"** + dep + **"'"**, ex);  }  }  }   *// Create bean instance.* //实例化依赖的bean后便可以实例化mbd本身了singleton模式的创建  **if** (mbd.isSingleton()) {  sharedInstance = getSingleton(beanName, () -> {  **try** {  **return** createBean(beanName, mbd, args);  }  **catch** (BeansException ex) {  *// Explicitly remove instance from singleton cache: It might have been put there  // eagerly by the creation process, to allow for circular reference resolution.  // Also remove any beans that received a temporary reference to the bean.* destroySingleton(beanName);  **throw** ex;  }  });  bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  }   **else if** (mbd.isPrototype()) {  //prototype模式的创建(new)  *// It's a prototype -> create a new instance.* Object prototypeInstance = **null**;  **try** {  beforePrototypeCreation(beanName);  prototypeInstance = createBean(beanName, mbd, args);  }  **finally** {  afterPrototypeCreation(beanName);  }  bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  }   **else** {  // 指定scope 上实例化bean  String scopeName = mbd.getScope();  **final** Scope scope = **this**.**scopes**.get(scopeName);  **if** (scope == **null**) {  **throw new** IllegalStateException(**"No Scope registered for scope name '"** + scopeName + **"'"**);  }  **try** {  Object scopedInstance = scope.get(beanName, () -> {  beforePrototypeCreation(beanName);  **try** {  **return** createBean(beanName, mbd, args);  }  **finally** {  afterPrototypeCreation(beanName);  }  });  bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);  }  **catch** (IllegalStateException ex) {  **throw new** BeanCreationException(beanName,  **"Scope '"** + scopeName + **"' is not active for the current thread; consider "** +  **"defining a scoped proxy for this bean if you intend to refer to it from a singleton"**,  ex);  }  }  }  **catch** (BeansException ex) {  cleanupAfterBeanCreationFailure(beanName);  **throw** ex;  }  }  // 检查需要的类型是否符合bean的实际类型  *// Check if required type matches the type of the actual bean instance.* **if** (requiredType != **null** && !requiredType.isInstance(bean)) {  **try** {  T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);  **if** (convertedBean == **null**) {  **throw new** BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  }  **return** convertedBean;  }  **catch** (TypeMismatchException ex) {  **if** (**logger**.isTraceEnabled()) {  **logger**.trace(**"Failed to convert bean '"** + name + **"' to required type '"** +  ClassUtils.*getQualifiedName*(requiredType) + **"'"**, ex);  }  **throw new** BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  }  }  **return** (T) bean; } |

仅从代码量上就能看出来 bean 的加载经历了一个相当复杂的过程，其中涉及各种各样的考虑。 相信读者细心阅读上面的代码，并参照部分代码注释，是可以粗略地了解整个Spring加载bean的过程。对于加载过程中所涉及的步骤大致如下。

1. 转换对应 beanName

或许很多人不理解转换对应 beanNarne 是什么意思，传人的参数 name 不就是 beanNarne吗？其实不是，这里传入的参数可能是别名，也可能是 FactoryBean ，所以需要进行一系列的解析，这些解析内容包括如下内容 。

* 去除FactoryBean 的修饰符，也就是如果name=”&aa”，那么会首先去除＆而使name=”aa”。
* 取指定 alias 所表示的最终 beanName，例如别名 A 指向名称为 B 的 bean 则返回 B;若别名 A 指向别名 B，别名B又指向名称为 C 的 bean 则返回 C。

1. 尝试从缓存中加载单例：

单例在 Spring 的同一个容器内只会被创建一次，后续再获取 bean ，就直接从单例缓存中获取了。 当然这里也只是尝试加载，首先尝试从缓存中加载，如果加载不成功则再次尝试从singletonFactories中加载。 因为在创建单例 bean 的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，在Spring中创建 bean 的原则是不等bean创建完成就会将创建 bean的 ObjectFactory 提早曝光加入到缓存中， 一旦下一个 bean 创建时候需要依赖上一个bean则直接使用ObjectFactory(后面章节会对循环依赖重点讲解)。

1. bean 的实例化

如果从缓存中得到了bean的原始状态，则需要对 bean 进行实例化。 这里有必要强调一下，缓存中记录的只是最原始的bean状态， 并不一定是我们最终想要的bean。 举个例子，假如我们需要对工厂bean进行处理，那么这里得到的其实是工厂 bean 的初始状态，但是我们真正需要的是工厂bean定义的 factory-method 方法中返回的bean，而getObjectForBeanlnstance 就是完成这个工作的，后续会详细讲解。

1. 原型模式的依赖检查

只有在单例情况下才会尝试解决循环依赖，如果存在A中有B的属性，B中有A的属性，那么当依赖注入的时候，就会产生当A还未创建完的时候因为对于B的创建再次返回创建A,造成循环依赖，也就是情况：isPrototypeCurrentlyInCreation(beanName)为true

1. 检测parentBeanFactory

从代码上看，如果缓存没有数据的话直接转到父类工厂上去加载了，这是为什么呢？

可能读者忽略了一个很重要的判断条件： parentBeanFactory !=null && !containsBean Definition (beanName), parentBeanFactory != null。parentBeanFactory如果为空，则其他一切都是浮云，这个没什么说的，但是  
!containsBeanDefinition(beanName）就比较重要了，它是在检测如果当前加载的 XML 配置文件中不包含 beanName 所对应的配置，就只能到 parentBeanFactory 去尝试下了，然后再去递归的调用getBean 方法 。

1. 将存储 XML 配置文件的 GernericBeanDefinition 转换为 RootBeanDefinition

因为从 XML 配置文件中读取到的bean信息是存储在GernericBeanDefinition中的，但是所有的bean后续处理都是针对于 RootBeanDefinition 的 ，所以这里需要进行一个转换，转换的同时如果父类bean不为空的话，则会一并合并父类的属性。

1. 寻找依赖

因为 bean 的初始化过程中很可能会用到某些属性，而某些属性很可能是动态配置的，并且配置成依赖于其他的 bean ，那么这个时候就有必要先加载依赖的bean，所以，在Spring的加载顺寻中，在初始化某一个bean的时候首先会初始化这个bean所对应的依赖。

1. 针对不同的scope 进行 bean 的创建

我们都知道，在Spring中存在着不同的scope ，其中默认的是singleton ，但是还有些其他的配置诸如 prototype、request之类的。 在这个步骤中， Spring 会根据不同的配置进行不同的初始化策略。

1. 类型转换

程序到这里返回bean后已经基本结束了，通常对该方法的调用参数requiredType是为空的，但是可能会存在这样的情况，返回的bean其实是个 String ，但是 requiredType 却传入Integer类型，那么这时候本步骤就会起作用了，它的功能是将返回的 bean 转换为 requiredType 所指定的类型。 当然，String 转换为 Integer 是最简单的一种转换，在 Spring 中提供了各种各样的转换器，用户也可以自己扩展转换器来满足需求。

经过上面的步骤后 bean 的加载就结束了，这个时候就可以返回我们所需要的bean了，

图 5-1 直观地反映了整个过程。 其中最重要的就是步骤8，针对不同的 scope 进行 bean 的创建，你会看到各种常用的 Spring 特性在这里的实现。

在细化分析各个步骤提供的功能前，我们有必要先了解，下FactoryBean的用法 。

### 5.1 FactorvBean 的使用

一般情况下，Spring通过反射机制利用bean的class属性指定实现类来实例化bean。 在某些情况下，实例化bean过程比较复杂，如果按照传统的方式，则需要在< bean >中提供大盘的配置信息，配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。Spring为此提供了一个 org.Springframework.bean.factory. FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化bean的逻辑。

FactoryBean接口对于 Spring 框架来说占有重要的地位， Spring自身就提供了70多个FactoryBean的实现。它们隐藏了实例化一些复杂 bean 的细节，给上层应用带来了便利。从 Spring3.0 开始， FactoryBean 开始支持泛型，即接口声明改为FactoryBean<T＞的形式：

|  |
| --- |
| **package** org.springframework.beans.factory;  **import** org.springframework.lang.Nullable;**public interface** FactoryBean<T> {  T getObject() **throws** Exception;  Class<?> getObjectType(); **default boolean** isSingleton() {  **return true**;  } } |

在该接口中还定义了以下 3 个方法。

* T getObject()：返回由 FactoryBean 创建的 bean 实例，如果 isSingleton()返回 true ，则该实例会放到 Spring 容器中单实例缓存池中。
* **boolean** isSingleton()：返回由FactoryBean创建的bean 实例的作用域是 singleton 还是prototype。
* Class<?> getObjectType()：返回 FactoryBean 创建的 bean 类型 。

当配置文件中＜bean＞的 class 属性配置的实现类是 FactoryBean 时，通过 getBean()方法返回的不是 FactoryBean本身，而是 FactoryBean#getObject( )方法所返回的对象，相当于FactoryBean#getObject()代理了 getBean()方法 。 例如：如果使用传统方式配置下面 Car 的＜bean>时·， Car 的每个属性分别对应一个<property>元素标签。

|  |
| --- |
| **public class** Car{  **private int maxSpeed**;  **private** String **brand**;  **private double price**;  *//get set* } |

如果用 FactoryBean 的方式实现就会灵活一些 ，下例通过逗号分割符的方式一次性地为 Car的所有属性指定配置值：

|  |  |
| --- | --- |
| **public class** CarFactoryBean **implements** FactoryBean<Car> {  **private** String **carInfo**;  @Override  **public** Car getObject() **throws** Exception {  Car car = **new** Car();  String[] infos = **carInfo**.split(**","**);  car.setBrand(infos[0]);  car.setMaxSpeed(Integer.*valueOf*(infos[1]));  car.setPrice(Double.*valueOf*(infos[2]));  **return** car;  }  @Override  **public** Class<?> getObjectType() {  **return** Car.**class**;  }  @Override  **public boolean** isSingleton() {  **return false**;  }  **public** String getCarInfo() {  **return carInfo**;  }  **public void** setCarInfo(String carInfo) {  **this**.**carInfo** = carInfo;  } } |  |

有了这个 CarFactoryBean 后，就可以在配置文件中使用下面这种向定义的配置方式配置

Car Bean 了：

|  |
| --- |
| <**bean id="car"  class="com.prosayj.springboot.spring源码深度剖析.第5章\_bean的加载.CarFactoryBean"  carInfo="超级跑车,400,2000000"**></**bean**> |

当调用 getBean（”car”）时， Spring 通过反射机制发现 Car FactoryBean 实现了 FactoryBean的接口，这时 Spring 容器就调用接口方法 CarFactoryBean#getObject（）方法返回。 如果希望获取CarFactoryBean 的实例,则需要在使用 getBean(beanName） 方法时在 beanName 前显示的加上”＆”前缀，例如 getBean（”＆car”）。

### 5.2 缓存中获取单例 bean

介绍过 FactoryBean 的用法后，我们就可以了解 bean 加载的过程了 。 前面已经提到过，单

例在 Spring 的同一个容器内只会被创建一次，后续再获取 bean 直接从单例缓存中获取，当然

这里也只是尝试加载，首先尝试从缓存中加载，然后再次尝试尝试从 singletonFactories 中加载。

因为在创建单例 bean 的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，

Spring 创建 bean 的原则是不等 bean 创建完成就会将创建 bean 的 ObjectFactory 提早曝光加入到

缓存中，一旦下一个 bean 创建时需要依赖上个 bean ，则直接使用 ObjectFactory 。

public Ob] ect gets 工 ngleton (String beanNarne) {

／／参数 true 设I'f:标识允许早期依赖

return getSinglet。n (beanNarne , t r ue );

protected Ob] ect getSingleton(String beanNarne , boolean allowEarlyReference)

／／检盒缓存 tI·t是干吁存在实仿lj

Object singletonObject = this . s 工ηq letonOb] ects . get(beanNarne) ;

if (singletonObject == null) {

／／如果为空 . 9!Jj锁定全局变盎并进行处；四

synchronized (this.singletonOb] ects) {

／／如果此 bean J正在 ）111载则不处础V且 94 第 5 章 bean 的加载

singletonObject = this .earlySingletonOb] ects . get(beanName) ;

if (singletonObject == null 晶晶 allowEarlyReference) {

I I ;'j某些方法记~捉前初始化的时候则会调川：J addSi 呵 le ton Factory 方法将对应的

/ /Obj ect F actory 初始化策附存 fi"ME si n gletonFactor 工 es

ObjectFactory singletonFactory = this 5 工 ngletonFactories . get

(beanName) ;

if {singletonFactory 1 = null) {

／／调rn预先设定的 getObject 方法

singletonObject = singletonFactory .getObject() ;

／／ 记录在级存 rJ1 , earlySingletonOb〕 ects 丰II s 工 ngleto 口 Fact 。 ries 厅:Jf;

t his earl ySingle tonObjects .put (beanName , singletonObject );

this . singletonFact 。 ries remove(beanName) ;

returη （ singletonObject 1 = NULL\_OBJECT ? singletonOb] ect : null );

这个方法因为涉及循环依赖的检测，以及涉及很多变量的记录存取，所以让很多读者摸不

着头脑 。 这个方法首先尝试从 singletonO同时ts 里面获取实例，如果获取不到再从 early SingletonObjects 里面获取，如果还获取不到，再尝试从 singletonFactories 里面获取 beanName 对应的

ObjectFactory ，然后调用这个 ObjectFactory 的 getObject 来创建 bean ，并放到 early SingletonObjects 里面去 ，并且从 singletonFacotories 里面 remove 掉这个 ObjectFactorγ ，而对于后续的所

有内存操作都只为了循环依赖检测时候使用，也就是在 allowEarlyReference 为阳巳的情况下才

会使用 。

这里涉及用于存储 bean 的不同的 map ，可能让读者感到崩溃，简单解释如下 。

s in g l eto nObjects ：用于保存 Bea1训 ame 和创建 bean 实例之间的关系， bean name 一＞ bean

mstance 。

singleton.Factories ：用于保存 BeanName 和创建 bean 的工厂之间的关系， bean name 一＞

ObjectFactory 。

earlySingletonObjects ：也是保存 BeanName 和创建 bean 实例之间的关系，与

singletonO均 ects 的不同之处在于，当一个单例 bean 被放到这里面后，那么当 bean 还

在创建过程中，就可以通过 getBean 方法获取到了，其目的是用来检测循环引用 。

0 registeredSingletons ：用来保存当前所有巳注册的 bean。

5.3 从 bea目的实例中获取对象

在 getBean 方法中， getObjectForBeanlnstance 是个高频率使用的方法，无论是从缓存巾

获得 bean 还是根据不同的 scope 策略加载 bean 。 总之，我们得到 bean 的实例后要做的第一

步就是调用这个方法来检测一下正确性，其实就是用于检测当前 bean 是否是 FactoryBean

类型的 bean ，如果是，那么需要调用该 bean 对应的 FactoryBean 实例中的 getObject（）作为

制V且 5 .3 从 bean 的实例中获取对象 95

返回值 。

无论是从缓存仁｜二1获取到的 bean 还是通过不同的 scope 策略加载的 bean 都只是最原始的 bean

状态，并不一定是我们最终想要的 bean。举个例子，假如我们需要对工厂 bean 进行处理，那

么这里得到的其实是工厂 bean 的初始状态，但是我们真正需要的是工厂 bean 中定义的

factory-method 方法中退回的 bean ，而 getObjectForBeaninstance 方法就是完成这个工作的 。

protected Object getObjectForBeaninstance(

Object beaninstance, Str工ng name , String beanName , RootBeanDefinit 工口口 πlbd) {

／／如果J肯定的 name 是T.厂丰If（＜以＆为阶级） 且 bean Instance 义不是 FactoryBean 类型则验证不通过

工 f ( Bea 口 F ac toryUτi ls isFactoryDe reference(n ame ) ＆品 ’（ beaninstance instanceof

FactoryBean) ) (

throw new BeanisN。 tAFactoryExcept1on(transformedBeanName(name) , beaninstance

getClass()) ;

／／现在我们有了个 bean 的实 fY1J. 这个实例可能会是正 ~？，·（1甘 bean TI.X:.fi·J; FactoryBear飞

!! MMU主 FactoryBean J.li;们攸 HJ它创 ill实例，但是如果川户但要 1＇！：按~xll旦工厂叉’例而不是士厂的

//getOb 〕 ect 方法开l应的实例那么传人的 name 应该Jm人 1'iti ~级品

if ( 1 (beaninstance instanceof FactoryBean ) I I BeanFactoryUtils . Is Factory

Dereference (name)) (

return beaninstance;

11 )1114主 FactoryBea n

Ob] ect ob] ect = null ;

if (mbd == null) {

／／尝试从缓了F小 ｝111载 bean

。b 〕 ect = getCachedOb] ectForFactoryBean (beanName );

if (ob] ect == null) {

／／到这里己经 WI硕J J:il迫 bean Instance 一定是 FactoryBean 炎型

FactoryBean ＜勺＞ factory = (FactoryBean ＜。＞） beaninstance ;

llcontainsBeanDefinition 检测 beanDefin 工 tionMap 'I ’也就是在所有已经加载的类 1cJ·1检测

／凡是否定义 beanName

if (mbd == null 晶晶 ccntainsBeanDefinition (beanName )) {

／／将布fi~ XML 配？1＇文们的 GernericBeanDefinition 转换为 RootBeanDefinition ，如

II ＊指定 Bean Name i庄子 Bean 的话同时会合并父类的丰11 ）巳M性

mbd = getMergedLocalBeanDefinition(beanName) ,

／／是否是用户定义的而不是应川程序本身定义的

boolean synthetic = (mbd != null && mbd . isSynthetic( ));

ob] ect = getObjectFr。mFact。ryBean (factory , beanName , 1synthetic) ;

return ob"i ect ;

从上面的代码来看，其实这个方法并没有什么重要的信息，大多是些辅助代码以及一些功

能性的判断，而 真正 的核心代码却 委托给了 getObjectFrom FactoryBean ，我们来 看看

getObj ectForBeanInstance 中的所做的 T作 。

1. 对 Factory Bean 正确性的验证 。

前96 第 5 章 bean 的加载

2. 对非 Factory Bean 不做任何处理 。

3. 对 bean 进行转换 。

4. 将从 Factory 中解析 bean 的工作委托给 getObjectFromFactoryBean o

protected Object getObjectFromFactoryBean ( Fact。 ryBean factory , String beanName , Boolean

shouldPostProcess ) {

Process );

NULL OBJECT)) ;

／／如果是单例模式

if (factory .isS 1口qleton () && containsSingleton (beanName )) {

synchronized (getSingletonMutex ()) {

else (

Object object ~ this . factoryBeanOb] ectCache get (beanName ) ;

if （。bject == nu.Cl ) {

。bject = d。GetObjectFr。mFact。ryBean( factory , beanName , shouldPost

this . factoryBeanOb] ectCache .put (beanName, (ob] ect 1 = null ? ob] ect

return (object 1 = NULL OBJECT ? object null) ;

return d。GetObjectFr。mFact。ryBean( factory , beanName , shou~dPostProcess );

很遗憾，在这个代码中我们还是没有看到想要看到的代码，在这个方法里只做了一件事情，

就是返回的 bean 如果是单例的，那就必须要保证全局唯一 ，同时，也因为是单例的，所以不

必重复创建，可以使用缓存来提高性能，也就是说已经加载过就要记录下来以便于下次复用，

否则的话就直接获取了 。

在 doG巳tO均ectFromFactoryBean 方法巾我们终于看到了我们想要看到的并法，也就是 0战ject =

factory.getO均 ect（），是的，就是这句代码，我们的历程犹如剥洋葱一样， 一层一层的直到最内

部的代码实现，虽然很简单 。

private Object doGetObjectFromFactoryBean(

final FactoryBean factory , final String beanName, final boolean shouldPostProcess )

throws BeanCreationException {

Object object ;

try {

I /',/,i'l!l:权限验 iiE

工 f (System.getSecu r 工 t yManager () 1 ＝ ηu il) {

Action< Object> {) {

AccessControlContext acc = getAccessContr。lContext ();

try {

。bject = AccessController . doPrivileged (new PrivilegedExcept 工 0口

public Object run () throws Exception {

return factory . getObject ();

) , acc) ;

catch (PrivilegedActionExcept 工 on pae) {

throw pae . getExceptio口 （）；

明5.3 从 bea n 的实例中获取对象 97

else {

111'[-ttiJ，＼）川 get Ob 〕 ect 方法

。bject = fact。ry.getObject();

catch (FactoryBeanN。EIni tializedExcept:ion ex) {

throw new BeanCurrentlyinCreationException (beanName , ex toString()) ;

catch (Throwable ex) {

throw new BeanCreationExcept l。 n(beanName ，” Fac t。 ryBean threw exception 。n

。bJect creation”, ex );

工 f （。bject == null && isSinqletonCurrent:lyinCreation (beanName)) {

throw new BeanCurrentlyinCreationException (

bean Name ，” FactoryBeaηwh 工 ch 工 s currently in creation returned

null from getObject " ) ;

工 f （。bject != null && shouldPostProcess) {

try {

I I i)f;Jfl) Object Factory 的后处理 （，~

。 bject = p。 stPr。 cessObjectFr。mFact。 ryBean (object, beanName) ,

catch (Throwable ex ) {

throw new BeanCreationException (beanName , ” Post-processing of the

FactoryBean ' s object failed”, ex );

return ob] ect ,

上面我们已经讲述了 FactoryBean 的调用方法，如果 bean 声明为 FactoryBean 类型，则当

提取 bean fl才提取的并不是 FactoryBean ，而是 FactoryBean 中对应的 getO战ject 方法返回的 bean,

而 doGetObjectFromFactoryBean 正是实现这个功能的 。 但是，我们看到在上面的方法中除了调

用。同巳ct = factory.getObject（）得到我们想要的结果后并没有直接返回，而是接下来又做了些后

处理的操作，这个又是做什么用的呢？于是我们跟踪进入 AbstractAutowireCapableBeanFactory

类的 postProcessObjectFromF actoryBean 方法：

I Abstra州towireCapa协BeanFactory.java ' I

pr。 tected Ob] ect p。s tProcessOb 〕 ectFrcmFactoryBean ( Ob 〕 ect 。bject , String beanName) {

return applyBeanP。stProcessorsAfterinitialization (object, beanl\ame) ;

public Object applyBeanP。st: Pr。cess。rsAfterinitialization(Object existingBean , String beanName)

throws BeansException {

Object result = existingBean ;

for (BeanPostProcessor beanProcessor getBeanPostProcessors ()) {

result = beanPr。cessor . postProcessAfterinitializat i。n(result , beanName) ;

制98 第 5 章 bean 的加载

if (result == null) (

return result ;

return result ;

对于后处理器的使用我们还未过多接触，后续章节会使用大量篇幅介绍，这里，我们只需

了解在 Spring 获取 bean 的规则 中有这样一条 ： 尽可能保证所有 bean 初始化后都会调用注册的

BeanPostProc巳ssor 的 postProcessA负erlnitialization 方法进行处理，在实际开发过程中大可以针

对此特性设计自己的业务逻辑。

5.4 获取单例

之前我们讲解了从缓存中获取单例的过程，那么， 如果缓存中不存在已经加载的单例 b巳an

就需要从头开始 bean 的加载过程了，而 Spring 中使用 getSingleton 的重载方法实现 bean 的加

载过程 。

public Ob] ect getSingleton(String beanName , ObjectFactory singletonFact 。 ry) (

Assert . notNull(beanName,”’ beanName ’ must not be null ” );

／／全局变 iJ:~l.i耍同步

synchronized (this . singletonOb] ects ) (

／／首先检查对应的 bean 是否已经力II载过 ， jJ;J为 s 工 ngleton 模式其实就是复用 以创建的 bean .

／／所以这一步是必须的

Object singletonObject = this . singletonObjects . get(beanName) ;

／／如果为空才可以进行 singleto 的 bean 的初始化

if (singletonObject == null) (

if (this . singletonsCurrentlyinDestruction) (

throw new BeanCreationNotAllowedException(beanName ,

” Singleton bean creation not allowed while the singletons

of this factory are in destructi 。n ” ＋

” ( Do not request a bean from a BeanFactory 工n a destroy

method implementation !) ” ) ,

if ( logger . 工 SDebugEnabled () ) (

logger .debug ("Creating shared instance of singleton bean ’ ” + beant\ame

+ ”’”);

bef。reSinglet。nCreati 。n ( beanName ),

b。。 lean recordSuppress edExceptions = (this suppressedExceptions == null ) ;

if (recordSuppressedExcepti 。ns) (

this . suppressedExceptions = new LinkedHashSet<Exception>() ;

try (

／／初始化 bean

singlet 。nob 〕 ect = singlet。nFact。ry. getObj ect () ;

catch (BeanCreationException ex) {

if ( recordSuppressedExcepti。ns) {

for (Exception suppressedException : this . suppressedExceptions ) (

陆U5.4 获取单例 99

ex . addRelatedCause(suppressedException) ;

throw ex ,

f 工口a lly !

if (recordSuppressedExceptio口 s ) {

this . suppressedExceptions = null ,

afterSinglet。nCreati。n (beanName ),

／／加入级存

addSinglet。n ( beanName , singletonObject );

return (singletonObject 1= NULL OBJECT? singletonObject : null) ;

上述代码中其实是使用了 回调方法， 使得程序可以在单例创建的前后做一些准备及处理操

作，而真正的获取单例 b巳an 的方法其实’并不是在此方法中实现的， 其实现逻辑是在 0性ctFactory

类型的实例 singletonFactory 巾实现的 。 而这些准备及处理操作包括如下内容 。

I. 检查缓存是否 已经力［J 载过 。

2. 若没有加载，则记录 beanName 的正在加载状态 。

3 . 加载单例前记录加载状态 。

可能你会觉得 beforeSingl etonCreation 方法是个空实现，里面没有任何逻辑，但其实不是 ，

这个函数中做了 一个很重要的操作：记录加载状态，也就是通过 this .singletonsCurrentlyln”

Creation.add(beanName）才有一 当前正要创建的 bean 记录在缓存 中，这样便可以对循环依赖进行

检测 。

protected voic beforeSingletonCreation(String bea 口 Name) {

if ('this . 工nCreationCheckExclusions . contains(beanName) ＆品＇ this . singletons

CurrentlyinCreation . add(beanName)) {

throw new BeanCurrentlyinCreat 工 onException ( beanName) ;

4. 通过调用参数传入的 ObjectFactory 的个体 0均 ect 方法实例化 bean 。

5 . 加载单例后 的处理方法调用 。

同步骤 3 的记录加载状态相似， 当 bean 加载结束后需要移除缓存中对该 bean 的正在加载

状态的记录 。

protected vo 工 ci afterSingletonCreation (String beanName) (

if ( 1this . inCreat工onCheckExclusions . contains(beanName) ＆ ιl th工s . singletons

CurrentlyinCreat 工 on. remove (bea:iName)) {

throw 口 ew IllegalStateException ( ” S ingleton ’” + beanName + ”’ l sn ’ t

currently in creation” ) ;

6. 将结果记录至缓存并删除加载 bean 过程 中所记录 的各种辅助状态 。

目100 第 5 章 bean 的加载

protected void addSinglet。n(String beanName , Object s 工 ngletonObject) {

synchronized (th 工 s . singletonOb〕 ects) {

this . s 工 ngletonObjects . put (beanName , ( s 工 ngletonObject 1= null ? singletonObject

NULL OBJECT)) ,

thi s singletonFact 。r工 es . remove(beanName) ,

th 工 s . earlyS 工 ngletonObjects . remove(beanName) ;

this . registeredSinglet 。n s . add (beanName) ;

7. 返回处理结果 。

虽然我们已经从外部了解了加载 bean 的逻辑架构，但现在我们还并没有开始对 bean 加载

功能的探索，之前提到过， bean 的加载逻辑其实是在传入的 ObjectFactory 类型的参数

s ingletonFactory 中定义的，我们反推参数的获取，得到如下代码 ：

sharedinstance = getSingleton(beanName , new ObjectFact。ry<Ob 〕 ect> () {

public Object getOb] ect () throws BeansException {

try {

return createBean (beanName, mbd, args) ;

catch (BeansException ex) {

destroys 工 ngleton(beanName) ;

thr 。w ex ;

ObjectFactory 的核心部分其实只是调用了 createBean 的方法，所以我们还需要到 createBean

方法中追寻真理。

5.5 准备创建 bean

我们不可能指望在一个函数中完成一个复杂的逻辑，而且我们跟踪了这么多 Spring 代码，

经历了这么多函数，或多或少也发现了一些规律 ： 一个真正干活的函数其实是以 do 开头的，

比如 doGetObjectFromFactoryBean ；而给我们错觉的函数，比如 getObjectFromFactoryBean ，其

实只是从全局角度去做些统筹的工作 。 这个规则对于 createBean 也不例外，那么让我们看看在

createBean 函数中做了｜那些准备工作 。

pr 。 tected Ob] ect createBean(final String beanName , final RootBeanDefin 工 t ion mbd , final

Object [] args ) throws BeanCreat 工 onException {

if (logger . isDebugEnabled () ) {

logger . debug （ ” Creating 工 nsta n ce of bean '" + beanName + ””’),

I l<fJ!i定 class ，根据设置的 class 属性或者根据 className 来解析 Class

res 。 lveBeanClass(mbd , beanName) ;

／／验 iiE及准备也i盖的方法

try {

mbd. prepareMeth。dOverrides 门，

前5 .5 准备创建 bean

catch (BeanDefinit l。nValidationExceptior: ex) (

throw new BeanDefinitionStoreExcept i。n(mbd getRes 。urceDescription() ,

beanName ，” Validation 。 f me th 。d 。 verrides failed” , ex) ;

try {

／／给 BeanPostProcessors 一个机会来返回代用来抖代ft1U1<J:;i~Wu

Ob] ect bean = res 。lveBef。re 工 nstantiation (beanName , mbd) ,

工 f (bean 1= null) {

return bean ;

catch (Throwable ex) {

throw new BeanCreationException(mbd . getResourceDescription() , beanName ,

”BeanPostProcessor before instantiation of bean failed ”, ex) ;

Object beanlnstance = d。CreateBean (beanName, mbd , args) ;

if (logger . isDebugEnabled ()) {

logger . debug (” Finished creating instance of bean ’” + beanName + ”’”);

return beanlnstance ,

从代码中我们可以总结出函数完成的具体步骤及功能。

I. 根据设置的 class 属性或者根据 classl'如ne 来解析 Cl ass 。

2 对 ovetTide 属性进行标记及验证 。

101

很多读者可能会不知道这个方法的作用，因为在 Spring 的配置里面根本就没有诸如

override-method 之类的配置， 那么这个方法到底是干什么用的呢 ？

其实在 Spring 中确实没有 override-method 这样的配置，但是如果读过前面的部分，可能

会有所发现，在 Spring 配置中是存在 lookup-method 和 replace-method 的，而这两个配置的力n

载其实就是将配置统一存放在 BeanDefinition 中的 methodOverrides 属性里，而这个函数的操作

其实也就是针对于这两个配置的 。

3. 应用初始化前的后处理器，解析指定 bean 是否存在初始化前的短路操作 。

4. 创建 bean o

我们首先查看下对 override 属性标记及验证的逻辑实现。

5.5.1 处理 override 属性

查看游、码中 AbstractBeanDefinition 类的 prepareMethodOveηides 方法 ：

public void prepareMethodOverrides() throws BeanDefinitionValidationExcept 工 on {

II Check that lookup methods exists

MethodOverrides meth。dOverr ides = getMethodOverrides () ;

if ( 'methodOverrides . isEmpty ()) (

for (MethodOverride mo : methodOverrides . getOverrides ()) {

prepareMeth。dOverride <mo) ;

明102 第 5 章 bean 的加载

protected void prepareMethodαrerride (Method伪erride mo) throws BeanCefinitionValidationExceptioη ｛

／／获取对j虫类小对应方法名 的个数

int count = ClassUtils . getMethodCountforName(getBear丁Class() , mo . getMethodName()) ;

if (c。 unt == 0) {

throw new BeanDefinit 工 onValidatio口Exception (

” Invalid method override no method with name ’” + mo getMethodName() +

”’ 。n class [ ” + getBeanClassName() + ” ] ” ) ;

else if (count == 1) {

／／标记 MethodOverride 'fg｛米被被盖 ， 避免参数类型检盒的开销 。

m。 . setOverloaded(false);

通过以上两个函数的代码你能体会到它所要实现的功能吗？之前反复提到过，在 Spring 配

置中存在 lookup-method 和 replace-method 两个配置功能，而这两个配置的加载其实就是将配置

统一存放在 BeanDefinition 中的 methodOverrides 属性里，这两个功能实现原理其实是在 bean 实

例化的时候如果检测到存在 methodOverrides 属性，会动态地为当前 bean 生成代理并使用对应

的拦截器为 bean 做增强处理，相关逻辑实现在 bean 的实例化部分详细介绍 。

但是，这里要提到的是，对于方法的匹配来讲，如果一个类中存在若干个重载方法，那么，

在函数调用及增强的时候还需要根据参数类型进行匹配，来最终确认当前调用的到底是哪个函

数 。 但是， Spring 将一部分匹配工作在这里完成了，如果当前类中的方法只有一个，那么就设

置重载眩方法没有被重载，这样在后续调用的时候便可以直接使用找到的方法，而不需要进行

方法的参数匹配验证了，而且还可以提前对方法存在性进行验证，正可谓一箭双雕。

5.5.2 实例化的前置处理

在真正调用 doCreate 方法创建 bean 的实例前使用了这样一个方法 reso lveBeforelnstantiation

(beanName, mbd）对 BeanDefinigiton 中的属性做些前置处理。当然，无论其中是再有相应的逻

辑实现我们都可以理解，因为真正逻辑实现前后留有处理函数也是可扩展的一种体现，但是，

这并不是最重要的，在函数中还提供了一个短路判断，这才是最为关键的部分。

if (bean != null) {

return bean ;

当经过前置处理后返回的结果如果不为空，那么会直接略过后续的 bean 的创建而直接返

回结果 。 这一特性虽然很容易被忽略，但是却起着至关重要的作用，我们熟知的 AOP 功能就

是基于这里的判断的 。

protected Ob] ect resolveBeforeinstantiat 工 on (String beanName , RootBeanDefinition mbd) {

Object bean = null ;

／／如果尚未被解析

if ( 1Boolean . FALSE .equals (mbd . beforeinstantiationResolved)) {

II Make sure bean class is actually resolved at th 工s po工口 t.

盼5.5 准备创建 bean 103

if (mbd . hasBeanClass () && 1mbd isSynthetic () && hasinstantiationAware

BeanPostPr。 cessors ()) {

bean = applyBeanP。stPr。cess。rsBef。re工nstantiati。n (mbd . getBeanClass() ,

beanName ) ;

if (bean ’ = null ) (

bean = applyBeanP。stPr。cess。rs且fterinitializati。n (bean , beanName) ,

mbd . beforeinstantiat 工 onResolved = (bean 1 = null );

return bean ,

此方法中最吸引我们的无疑是两个方法 applyBeanPostProcessorsBeforelnstantiation 以及

applyBeanPostProcessorsAfterInitialization 。 两个方法实现的非常简单，无非是对后处理器中的

所有 lnstantiationAwareBeanPostProcessor 类型的后处理器进行 postProcessBeforelnstantiation 方

法和 BeanPostProcessor 的 postProcessAfterInitialization 方法的调用 。

1 . 实例化前的后处理器应用

bea n 的实例化前调用，也就是将 A b sractBeanDefinition 转换为 BeanWrapper 前的处理 。 给

子类一个修改 BeanDefinition 的机会，也就是说当程序经过这个方法后， bean 可能已经不是我

们认为的 bean 了 ， 而是或许成为了一个经过处理的代理 bean ，可能是通过 cglib 生成的，也可

能是通过其他技术生成的 。 这在第 7 章中会详细介绍，我们只需要知道，在 bean 的实例化前

会调用后处理器的方法进行处理。

protected Object applyBeanPostProcessorsBeforeinstant 工at工on ( Class beanClass , Stri ng beanName )

τhrows BeansException {

f o r (BeanPostProcessor bp : getBeanPostProcesso rs ()) {

PostProcessor ) bp ;

if (bp instanceof Instant工ati。nAwareBeanP。stPr。cess。r ) {

InstantiationAwareBeanPostProcessor ibp = (Instan t 工a tion AwareBean

Object result = ibp .p。stPr。cessBef。re工nstantiati。n (beanCla s s , bea nName );

if (result 1= null) {

return result ;

return nul l,

2 . 实例化后的后处理器应用

在讲解从缓存中获取单例 bean 的时候就提到过， Spring 中的规则是在 bean 的初始化后尽

可能保证将注册的后处理器的 postProcessA丘erlnitialization 方法应用到该 bean 中，因为如果返

回的 bean 不为空， 那么便不会再次经历普通 bean 的创建过程，所以只能在这里应用后处理器

的 postProcessAfterinitial ization 方法。

明10韭 第 5 章 bean 的加载

publ 工C Object applyBeanPostProcessorsAfterIηitialization(Object existingBean , String beanName)

throws BeansException (

Object result = exist 工 ngBean ;

for ( BeanP。stPr。cess。r beanPr。cess 。τ ： getBeanPostProcessors ()) (

result = beanProcessor .p。stPr。cessAfterinitializati。n (result , beanName);

if (result == null) (

return result ;

return result ;

5.6 循环依赖

实例化 bean 是一个非常复杂的过程， 而其中 比较难以理解的就是对循环依赖的解决，不

管之前读者有没有对循环依赖方面的研究，这里有必要先对此知识点稍作回顾 。

5.6.1 什么是循环依赖

循环依赖就是循环引用，就是两个或多个 bean 相互之间的持有对方，比如 Circ leA 引用

C ircleB , CircleB 引用 CircleC, C ircl eC 引用 CircleA ，则它们最终反映为一个环 。 此挝、不是循

环调用，循环调用是方法之间的环调用， 如罔 5-2 所示 。

<lava Class•

G TestA

。 testB TestB

」一

~ a()

Q getTestB ()

9 setTestB (]

...use-.

<laνa Class、

G TestC

a testA : TestA

e c[)

e getTe筑A()

e oetTe处A()

'8"USe•

图 5-2 循环依赖

•lava Class•

G TestB

c一testC Testc

一一 ----1

Q b ()

& getT<stC ()

e setTe这c ()

循环调用是无法解决的 除非有终结条件 再则就是死循环 ，最终导致内存溢出错误。

5.6.2 Spring 如何解决循环依赖

Spr ing 容器循环依赖包括构造器循环依赖和 se忧er 循环依赖，那 S pring 容器如何解决循环

盼V且 5.6 循环依赖 105

依赖呢？ 首先让我们来定义循环引用类 ：

public class TestA {

private TestB testB ;

public void a() (

testB . b ();

public TestB getTestB ( ) (

return testB ;

public v:>id setTestB (TestB testB) {

this . testB = testB ;

public class TestB (

private TestC testC ;

public void b () (

testC . c ();

publ 工 C TestC getTestC () {

retu:::n testC ;

p ublic void setTestC (TestC testC ) {

this . testc = testC ;

public class TestC {

pr 工 vate TestA testA ;

public void c () (

testA . a () ;

public TestA getTestA () {

return testA ;

public void setTestA (TestA testA ) (

th 工 s . testA = testA ;

在 Spring 中将循环依赖的处理分成了 3 种情况 。

V且 106 第 5 章 bean 的加载

1.构造器循环依赖

表示通过构造器注入构成的循环依赖， 此依赖是无法解决的 ，只能抛出 BeanCurrentlylnCreationException 异常表示循环依赖 。

如在创建 TestA 类时， 构造器需要 TestB 类，那将去创建 TestB ， 在创建 TestB 类时又发现

需要 TestC 类， 则又去创建 TestC ， 最终在创建 TestC 时发现又需要 TestA ，从而形成一个环，

没办法创建。

Spring 容器将每一个正在创建的 bean 标识符放在一个“当前创建 bean 池”中 ， be an 标识

柏：在创建过程中将一直保持在这个池中，因此如果在创建 bean 过程中发现自己已经在“当前

创建 bean 池” 里时，将抛出 BeanCurrentlylnCreationException 异常表示循环依赖；而对于创建

完毕的 bean 将从“当前创建 bean 池”中清除掉 。

我们通过一个直观的测试用例来进行分析 。

I. 创建配置文件 。

<be an id=” testA” class= "com . bean TestA”>

<constructor- arg index ＝ ” 。 ” ref ＝ ” testB ” ／〉

</be an>

<bea n id=”testB” class=” com . bean TestB”>

<constructor arg 工 ndex ＝ ” 。 ” ref＝ ” testC ” ／〉

</bean>

<be an id=” tesi::C ” class=” com .bea n . TestC”>

<constructor- arg index ＝ ” 。 ” ref ＝ ” testA ” ／〉

</bea n>

2. 创建测试用例 。

@Te s t (expected = BeanCurrentlyinCreacionException . class)

pub l 工 C 飞10id testCircleByConstructor () throws Throwable {

try {

new ClassPathXmlApplicationContext ( ” test . xml ” );

} catch (Exception e} {

／／ 因为要在创建 testC 时抛出；

Throwable el = e . getCause(} . getCause () . getCause() ;

throw el ;

针对以上代码的分析如下 。

Spring 容器创建“ testA ” bean ，首先去“当前创 建 bean 池” 查找是否当前 bean 正在

创建，如果没发现，则继续准备其需妥的构造器参数“t巳stB

放到 “当前创建 bean 池 ，，

。

Spring 容器创建“ testB ” bean ， 首先去“当前创建 bean 池”查找是否当前 bean 正在

创 建 ，如果没发现，则继 续准备其需妥的构造器参数“testC ” ，并将“testB”标识符

放到“当前创建 bean 池” 。

Spring 容器创建“testC ” bean ， 首先去“当前创建 bean 池”查找是否当前 bean 正在

创 建，如果没发现 ，则 继续准备其需妥的构造器参数“testA”，并将“testC”标识符

U局 5.6 循环依赖 107

放圭1J “当前创建 bean 池” 。

到此为止 Spring 容器要去创建“testA” bean ，发现该 bean 标识符在“当前创建 bean

池”中，因为表示循环依赖，抛出 BeanCurrentlyInCreationExceptiona

2. sette「循环依赖

表示通过 se忧er 注入方式构成的循环依赖 。 对于 S巳tter 注入造成的依赖是通过 Spring 容器

提前暴露刚完成构造器注入但未完成其他步骤（如 setter 注入）的 bean 来完成的，而且只能解

决单例作用域的 bean 循环依赖 。 通过提前暴露一个单例工厂方法，从而使其他 bean 能引用到

该 bean ，如下代码所示 ：

addSingletonFactory (beanName , new 0:0] ectFactory() {

public Object getob;ect () throws BeansException {

return getEarlyBeanReference(beanName , mbd , bean) ;

具体步骤如下 。

l. Spring 容器创建单例“ testA ” bean ，首先根据元参构造器创建 bean ，并暴露一个

“0均巳ctFactory”用于返回一个提前暴露一个创建中的 b巳a n ，并将“ testA”标识符放到“当前

创建 bean 池”， 然后进行 setter 注入“testB ” 。

2. Spring 容器创建单例 “ testB ” bean ，首先根据无参构造器创建 bean ，并暴露一个

“ ObjectFactory”用于返回一个提前暴露一个创建中的 bean ，并将“ testB ”标识符放到“当前

创建 bean 池”，然后进行 setter 注入“ circle ” 。

3. Spring 容器创建单例“ testC ” bean ，首先根据元参构造器创建 bean ，并暴露一个

“ ObjectFactory”用于返回一个提前暴露一个创建中的 bean ，并将“ testC ”标识符放到“当前

创建 bean 池”，然后进行 setter 注入“testA”。 进行注入“testA”时由于提前暴露了“0问jectFactory”

工厂，从而使用它返回提前暴露一个创建中的 bean 。

4. 最后在依赖注入 “ testB ”和“ testA ”，完成 setter 注入 。

3. prototype 范围的依赖处理

对于“prototype”作用域 bean, Spring 容器无法完成依赖注入，因为 Spring 容器不进行缓

存“prototype”作用域的 bean ，因此无法提前暴露一个创建中的 bean o 示例如下：

I. 创建配置文件 。

<bean id=” testA” c l ass=” com . bean . CircleA" scope=”prototype”>

<property name=” tests” ref=”tests”/>

</bean>

<bea n id=” tes t B" class=” com bean . Circles” scope=”prototype”>

<property name=”testC'' ref=”testC”/>

</bean>

<bean i d=”te stC” class=”com. bean . CircleC” scope=”prototype”>

<property name=” testA” r ef=” testA”/>

</bean>

V且 108 第 5 章 bean 的加载

2. 创建测试用例 。

@Test (expected = BeanCurrentlyinCreationException class )

public void testCircleBySetterAndPrototype () throws Throwab 二 e (

try {

ClassPathXmlApplicationContext ct x =new ClassPa t hXmlApplicationContext (

” testPrototype xml ” ),

System . out . pr1ntln (ct x getBean (” testA " ));

) catch ( Except 工。n e ) {

Throwable el = e getCause () getCause () .getCause () ;

throw e l,

对于“ singleton” 作用域 bean ，可以通过“民tAllowCircularReferences(false ） ；”来禁用循环

引用 。

感谢互联网时代，让我可以方便地获取我想要的各种信息 ， 当初我刚开始学习的时候 ， 一

直纠结于这里错综复杂的逻辑 ， 幸好我看到了一篇文章解开了我心中的疑惑 。 在此， 感谢原作

者 ， 并将原文与大家分享 ， 帮助大家更好的理解 Spring 的依赖， 大家可以从 h即 ：／／www.ifiym.

com/index.php/code/20120828000 I .html 来获取原文 。

5.1 创建 bean

介绍了循环依赖以及 Spring 中的循环依赖的处理方式后 ，我们继续 4 . 5 节的内容。当经历

过 resolveBeforelnstantiation 方法后，程序有 两个选择 ，如果创 建 了代 理 或 者 说重写 了

InstantiationAwareBeanPostProcessor 的 postProcessBeforelnstantiation 方法并在方法 postProcessBeforelnstantiation 中改变了 bean ， 则直接返回就可以 了 ， 否则需要进行常规 bean 的创建 。 而

这常规 bean 的创建就是在 doCreateBean 中完成的 。

protected Object doCreateBean (final Str ing beanName , fi nal RootBeanDefinition mbd , f工na l

Object[] args) {

II Instan t 工ate the bean .

BeanWrapper instanceWrapper = null ;

工 f (mbd . is Singleton () ) {

1nstanceWrapper = this . factoryBeanin s tanceCache remove (beanName ) ,

if (instanceWrapper == null ) (

／／根据指定 bean 使用对应的策略创建新的实 例， 如 · 工厂方法 、构造局数自 ;IJ注入 、简单初始化

instanceWrapper = createBeaninstance (beanName , mbd , args ),

final Object bean = (instanceWrapper 1= null 节 instanceWrapper . getWr apped instance ()

口ull ) ,

Class beanType = (instanceWrapper !=null ? instanceWrapper getWrappedClass ()

null ) ;

/I Allow post - p rocessors to modify the merged bea n definition .

synchronized (mbd . pos t Proces s ingLock ) (

工 f ( 'mbd . postProce ssed) (

／／应川j MergedBea nDefi ni tion PostProces s or

V且 5.7 创建 bea n 109

applyMergedBeanDefini t 工 onPostProcessors(mbd , beanType , beanName) ;

mbd p ostProcessed = true ;

／女

会是干吁切要提与l l版光 ． 吁＇·例 ι允许1i'l:ff、依赖ι旦~rl'rf bean 正在创建巾，检测循环依赖

\*/

油。lean earlySingletonE碍。sure = (mbd.isSingleton （）“吐白 . all。，weircularRef自由出S 晶晶

isSinglet。nCurrently工nCreati。n(beanName)),

工 f (earlySingletonExposure) {

if (logger . isDebugEnabled ()) {

logger . debug (” Eagerly caching bean ’” + beanName +

to allow for resolving poten t 工 al circular references " );

／／为滥免后WJ循环依赖，可以在 bean 初始化完成前将创建实例的 Ob 〕 ectFactory 1Jll人工厂

addSingletonFactory (beanName , new Object Factory () {

public Object getObject() throws BeansExcept 工 on {

／／对 bean l每一次依赖引用，主2!.'应用 SmartinstantiationAware BeanPost

//Processor

／／其中我们熟知的 AOP 就是在这里将 adv 工 ce 动态织人 beaη 巾，若没有则直接返回

//bean ，不做任何处理

retur 口 getEarlyBeanReference(beanName , mbd , bean) ;

,

} )

II Initialize the bean instance .

Object exposedObject = bean ;

try (

／／芹l bean 进行填充 ， 将各个属性｛i~I i注入，其中，可能存在依赖于；w;他 bean 的属性 ， 』)lj会递归 :fJJ始

／／依赖 bean

p。pulateBean (beanName , mbd ， 工 nstanceWrapper);

工 f (exposedObject 1 = null) {

I I i用汀l初始化方法，比如 in it - method

exposedObject ＝ 二nitializeBean (beanName , exposedObject , mbd) ;

catch (Throwable ex) {

工 f (ex instanceof BeanCreationExcept 工 on 晶晶 beanName.equals( ( (BeanCreatioη

Exceptioη ） ex) .g eτBea 口 Name ())) {

throw (BeanCreationException) ex ;

else {

throw new BeanCreationException(mbd . getResourc eDescription() , beanName ,

” I nitialization of bean failed ", ex) ;

if (earlySingletonExposure) {

Object earlySingletonReference = getSingleton(beanName , false) ;

llearlySingletonReference 只有在检测到有循环依赖的忻况下才会不为空

if (earlySingletonReference 1= null) {

V且 110 第 5 章 bean 的加载

／／如果 exposedOb 〕 ect 没有在初始化方法巾被改变，也就是没有被增强

if (exposedObject == bean) {

exposedOb] ect = earlySingle t onReference ;

)else if { ' this . al l。wRawin 〕 ectionDespiteWrapp 工 ng 晶晶 hasDependentBean

(beanName) ) {

String[] dependentBeans = getDependentBeans (beanName) ;

Set<String> actualDependentBeans ＝口ew LinkedHashSet<String>

(dependentBeans length ) ;

for (String dependentBean : dependentBeans) {

／／检 i9!1J依赖

if （’ removeSingletor丁I fCreatedF。rTypeCheckOnly (depe'.1dentBean ) ) {

actualDependentBeans . add(dependentBean) ;

/\*

＊ 因为 bean 创建后；民所依赖的 bean 一定是已经创建的，

女 actualDependentBeans 不为空则表示当前 bean {iij kj；后其依赖的 bean 却 没有没

全部创建完，也就是说存在循环依赖

食／

if ( 1actualDependentBeans . i sEmpty ()) {

~hrow new BeanCurrentlyinCreationException(beanName ,

” Bean with name ’ ” + beanName + ”’ has been in ] ected

into other beans [ ” +

StringUtils . collectionToCommaDelimitedString

(actualDependentBeans) +

” ] in its raw version as part of a circular reference ,

but has eve ntu a lly be e n " +

” wrapped . This rieans that said other beans do not use

the final version of the " +

” bean Th is is often t he result of over- eager type

matching - consider using ” +

” ’ getBeanNamesOfType ' with the ’ a llowEagerinit ' flag

turned off , for example ” );

II Regi ster bean as disposable .

t ry {

／／根据 scopse 注册 bean

registerDisp。sableBeanifNecessary (beanName , bean , mbd) ;

catch (BeanDef 工n it 工 onVa l 工 dati 。nException ex ) {

throw new BeanCreationE x ception(mbd . getResourceDescr 工 pt l。n() , beanName , ”

Invalid destruct i on signature”, ex );

return exposedObject ;

尽管日志与异常的内容非常重要，但是在阅读源码的时候似乎大部分人都会直接忽略掉。

在此不深入探讨日志及异常的设计，我们看看整个函数的概要思路。

V且 5.7 创建 bean 111

I. 如果是单例则需要首先清除缓存 。

2. 实例化 bean ，将 Bean.Definition 转换为 BeanWrapper 。

转换是一个复杂的过程，但是我们可以尝试概括大致的功能，如下所示 。

如果存在工厂方法则使用工厂方法进行初始化 。

\,) 一个类有 多个构造函数，每个构造函数都有不同的 参数，所以需要根据参数锁定构造

函数并进行初始化 。

如果既不存在工厂方法也不存在带有参数的构造函数，则使用默认的构造函数进行

bean 的实例 f匕 。

3. MergedBean.Defin itionPostProcessor 的应用 。

bean 合并后的处理， A utowired 注解正是通过此方法实现诸如类型的预解析 。

4. 依赖处理 。

在 Spring 中会有循环依赖的情况，例如，当 A 中含有 B 的属性，而 B 中又含有 A 的属性

时就会构成一个循环依赖，此时如果 A 和 B 都是单例，那么在 Spring 中的处理方式就是当创

建 B 的时候，涉及自动注入 A 的步骤｜｜寸，并不是直接去再次创建 A ，而是通过放入缓存中的

ObjectFactory 来创建实例，这样就解决了循环依赖的问题 。

5. 属性填充 。 将所有属性填充至 bean 的实例中 。

6 . 循环依赖检查 。

之前有提到过，在 Sping 中解决循环依赖只对单例有效，而对于 prototype 的 bean, Spring

没有好的解决办法，唯一要做的就是抛出异常 。 在这个步骤里面会检测已经加载的 bean 是否

已经出现了依赖循环，并判断是再需要抛出异常 。

7. 注册 Disposab l eBean 。

如果配置了 destroy-method ，这里需要注册以便于在销毁时候调用 。

8. 完成创建井返回 。

可以看到上面的步骤非常的繁琐，每一步骤都使用了大量的代码来完成其功能，最复杂

也是最难以理解的当属循环依赖的处理，在真正进入 doCreateBean 前我们有必要先了解下循

环依赖 。

5.7.1 创建 bean 的实例

当 我们了解了循环依赖以后就可以深入分析创建 bean 的每一个步骤了，首先我们从

createBeaninstance 开始 。

protected BeanWrapper createBeaninstance (String beanNarne , RootBea n Defin i t 工 on rnbd ,

Object[] args ) {

／／解析 class

Cla ss beanCl ass = resolveBeanClas s (rnbd, beanNarne ),

i f (beanClass 1= nul lιι 1 Mod 工 fi e r . isP ublic { beanClas s getModifiers{)) && 1rnbd .

isNonPubl i cAccessAllowed ()) {

V且 112 第 5 章 bean 的加载

throw new BeanCrea t 工 onException(rnbd getResourceDescription() , beanNarne ,

” Bean class isn ' t public , and non - public access not allowed : ” +

beanClass . getNarne()) ;

／／ ~n;+!T厂方法不为空则使jTJ T厂方法初始化策略

if (!nbd . getFactoryMethodNarne () ! = null) {

return instantiateUsingFact。ryMeth。d (beanNarne, rnbd , args) ;

II Shortcut when re creat 工ng the same bean

boolean resolved = false ,

boolean au t。wireNecessary = false ;

工 f (args == null ) {

synchronized (rnbd.co口 structorArgurnentLock) {

／／一个类有多个构造函数，每个构造商数都有不同的参数 ， 所以惊iJJ书前需要先根据参数锁定构

／／边的数或对应的t厂方法

工 f (rnbd . re sol vedConstructorOrFactoryMethod ! = null) {

resolved = true ,

au tow 工 reNecessary = rnbd . constructorArgurnentsResolved ,

／／如果已经 lffll析过则使川 fl)~析好的构；；！；：两数方法不＇［／；；要手1次锁定

if (resolved) {

if (autow工reNecessary) {

／／构边读i数向 i9J 注入

return aut。wireC。nstruct。r {beanNarne , rnbd , null , null) ,

else {

／／使 HJ默认构造民数构造

return instantiateBean (beanNarne, rnbd) ,

／／苟安根据参数解析构选民数

Constructor[] ctors = determiηeConstructorsFrornBeanPostProcess 。rs(beanClass ,

beanName) ;

if (ctors 1= null I I

rnbd . getResolvedAutow 工 reMode() == RootBeanDefinition . AUTOWIRE

CONSTRUCTOR I I

rnbd . hasConstructorArgurnentValues () I I ' ObjectUtils isEmpty {args)) {

／／构造函数自动注入

return aut。wireC。nstruct。r (beanName , rnbd , ctors , args) ;

／／使用默认构造的数构造

return 二nstant工 ateBean (beanNarne, rnbd) ;

虽然代码中实例化的细节非常复杂 ，但是在 createBeanlntance 方法中我们还是可以清晰地

看到实例化的逻辑的 。

1. 如果在 RootB eanDefinition 中存在 factoryMethodN ame 属性，或者说在配置文件中配置

了 factory-method ，那么 Spring 会尝试使用 instantiateUsingFa ctoryMethod(beanNam巳， mbd, args）方法

5 .7 创建 bean 113

根据 RootBeanDe mition 中的配置生成 bean 的实例 。

2 . 解析构造函数并进行构造函数的实例化 。 因为一个 bean 对应的类中可能会有多个构造

函数，而每个构造函数的参数不同， Spring 在根据参数及类型去判断最终会使用哪个构造函数

进行实例化。 但是，判断的过程是个比较消耗性能的步骤，所以采用缓存机制，如果已经解析过则

不需要重复解析而是直接从 RootBeanDefinition 中的属性 resoIvedConstructorOrFactoryMethod 缓存

的值去取，否则需要再次解析， 并将解析的结果添加至 RootBeanDefinition 中的属性

resolvedConstructorOrFactoryMethod 中 。

1. autowireConstructor

对于实例的创建 Spring 中分成了两种情况，一种是通用的实例化，另一种是带有参数的实例

化。 带有参数的实例化过程相当复杂，因为存在着不确定性，所以在判断对应参数上做了大量工作。

public BeanWrapper autowireConstructor(

final String beanName, final R。otBeanDefinition mbd , Constructor[]

chosenCtors , final Object[] explicitArgs) {

BeanWrapperimpl bw =new BeanWrapperimpl() ;

this . beanFactory . initBeanWrapper(bw) ;

Constructor construct 。rT。 Use = null,

ArgumentsHolder argsHolderToUse = null ;

Ob] ect[] argsToUse =null;

//explic 工 tArgs 通过 get Bean 方法传人

／／如果 get Bean 方法调用的时候指定方法参数那么直接使用

if (expl 工citArgs 1= null) {

argsT。 Use = explicitArgs ;

)else {

argsToResolve) ;

／／如果在 g et Bean 方法时候没有指定则尝试从配置文件中解析

Ob] ect[] argsToResolve = null;

／／尝试从缓存中获垠

synchronized (mbd . constructorArgumentL。ck) {

construct。rToUse = (Constructor) mbd.res。 lvedC。nstructorOrFactoryMeth。d ;

if (c。nstructorToUse != null && mbd . c。nstructorArgumentsResolved) (

／／从缓存中取

argsT。Use = mbd.resolvedConstructorArguments ;

if (argsToUse == null) {

／／配置的构造商数参数

argsToRes。 lve = mbd . preparedConstructorArguments ;

／／如果缓存中存在

if (argsT。Resolve != null) {

／／解析参数类型， 如给定方法的构造函数 A( int , int ） 则通过此方法后就会把配置中的

／／（ ” 1 ”，” l ” ）转换为 (1 , 1)

／／级存巾的值可能是原始值也可能是最终值

argsToUse = resolvePreparedArguments(beanName , mbd , bw, c。nstruct。rToUse ,

114 第 5 章 bean 的加载

／／没有被缓存

if (constructorToUse == null) {

II Need to resolve the constructor .

boolean autowiring = (chosenCt。 rs '= null I I

mbd . getResolvedAutowireM。de() == Ro。 tBeanDefinition . AUTOWIRE\_

CONSTRUCTOR) ;

c。 nstruct 。 rArgumentValues resolvedValues = null ;

int minNrOfArgs ;

if (explicitArgs 1= null ) {

m工 nNrOfArgs = explicitArgs . length;

}else {

／／提取配置文件中的配置的构造函数参数

ConstructorArgumentValues cargs = mbd . getConstructorArgumentValues();

／／用于承载解析后的构造函数参数的值

resolvedValues =new ConstructorArgumentValues();

／／能解析到的参数个数

minNrOfArgs = resolveConstructorArguments(beanName , mbd , bw , cargs,

resolvedValues) ;

II Take specified constructors , if any.

Constructor[] candidates = chosenCtors ;

if (candidates == null} {

getConstructors() } ;

beanClass . getName() +

+ ” l failed”, ex );

Class beanClass = mbd . getBeanClass() ;

try {

candidates = (mbd . isNonPublicAccessAllowed() ?

beanClass . getDeclaredConstructors () : beanClass

catch (Throwable ex) {

throw new BeanCreationException (mbd.getResourceDescription () , beanName ,

” Resolution of declared constructors on bean Class [ ” ?

” ] from ClassLoader [ ” + beanClass . getClassLοader ()

／／排序给定的构造函数， public 构造函数优先参数数量降序、斗 I: public 构造商数参数数量降序

ll.utowireUtils sortConstructors (candidates};

int minTypeDiffWeight = Integer . MAX\_VALUE ;

Set<Constructor> ambiguousConstructors = null ;

List<Except 工 on> causes = null ;

for (int i = 0 ； 工＜ candidates . length J 工＋＋）｛

Coηstructor<?> candidate= candidates[i] ;

Class[] paramTypes = candidate . getParameterTypes() ;

if (constructorToUse 1= null && argsToUse length > paramTypes.length) {

／／如果已经找到选用的构造函数或者币\i耍的参数个数小子当前的构造函数参数个数则终止，

／／因为已经按照参数个数降序排列

5.7 创建 bean

break;

if (paramTypes . length < minNrOfArgs) (

／／参数个数不相等

continue ;

ArgumentsHolder argsHolde r ;

if (resolvedValues != null) {

／／有参数则根据值构造对应参数类型的参数

try (

String[] paramNames = null ;

if (constructorPropertiesAnn。tationAvailable) (

／／注释上获取参数名称

115

paramNames = ConstructorPropertiesChecker . evaluateAnn。tati。n

(candidate , paramTypes . length);

if (paramNames == null) (

／／获取参数名称探索然

ParameterNameDiscoverer pnd = this .beanFactory. getParameter

NameDiscoverer() ;

if (pnd ’ = null) {

／／获取指定构造函数的参数名称

paramNames = pnd . getParameterNames(candidate) ;

／／根据名粉和数据类型创建参数持有者

argsHolder = createArgumentArray(

beanName , mbd , resolvedValues , bw , paramTypes,

paramNames , candidate , autowir 工 ng) ;

catch (Unsatisf 工 edDependencyExcept ion ex) (

’” + beanName + ”’ : ” + ex);

if (this .beanFactory . logger.isTraceEnabled()) {

this . beanFactory . logger.trace(

’『 Ignoring constructor [ ” + cand 工 date ＋ ” ］。 f bean

if (i == candidates . length - 1 && constructorToUse == null) (

if (causes != null) I

f。 r (Exception cause causes) {

this .beanFactory . onSuppressedException (cause );

throw ex ;

else I

II Swallow and try next constructor.

if (causes == null) (

causes= new LinkedList<Exception>();

causes . add(ex) ;

continue ;116 第 5 章 bean 的加载

)else {

工 f {paramTypes . length ! = explici tArgs. length) {

continue ;

／／构造函数没有参数的情况

argsHolder ＝口ew ArgumentsHolder(explicitArgs) ;

／／探测是否有不确定性的构造函数存在 ， 例如不同构造函数的参数为父子关系

int typeDiffWeight = (mbd.isLen 工entConstructorResolution () ?

argsHolder . getTypeDifferenceWeight (paramTypes) : argsHolder.

getAssignabilityWeight (paramTypes));

／／如果它代表着当前最接近的匹配则选择作为构造函数

if (typeDiffWeight < minTypeDiffWeight) {

constructorToUse = candidate ;

argsHolderToUse = argsHolder ;

argsToUse = argsHolder.arguments ;

minTypeDiffWeight = typeDiffWeight;

ambiguousConstructors = null ;

)else if (constructorToUse != null && typeDiffWeight = minTypeDiffWeight) {

if (ambiguousConstructors ＝＝口ull ) {

ambiguousConstructors =new LinkedHashSet<Constructor>();

ambiguousConstructors.add(constructorToUse);

ambiguousConstructors.add(candidate);

if (constructorToUse == null) {

thr。w new BeanCreationException(mbd.getResourceDescription() , beanName ,

” Could not resolve match 工 ng constructor ” +

” (hint : spec工 fy index/type/name arguments for simple

parameters to avoid type ambiguities )”) ;

)else if (ambiguousConstructors 1= null && 1mbd.isLen工entConstructor Resolution()) {

throw new BeanCreationException (mbd . getResourceDescription() , beanName ,

” Ambiguous constructor rr田tches found in bean ' " + beanName + ",” +

” (hint : spec 工 fy index/type/name arguments for simple

parameters to avoid type ambiguities ): ” +

ambiguousco口 s tructors);

if (expl 工 citArgs == null) {

／／将解析的构造函数加入缓存

argsHolderToUse . storeCache(mbd , constructorT。Use) ;

try {

Ob] ect beaninstance ;

if (System . getSecur工tyManager () 1= null) {

final Constructor ctorToUse = constructorT。Use;5.7 创建 bea n 117

final Object[] argumentsToUs e = argsT。Use;

悦目Instance = AccessController .doPrivileged (new PrivilegedAction< Object>() {

publ i c Ob] ect run () {

return beanFactory.getinstantiationStrategy() . instantiate(

πlbd , beanName , beanFactory, ctorToUse , argumentsT。Use) ;

) , beanFactory .getAccessControlContext ());

else {

beaninstance = this . beanFact。ry . getinstantiationStrategy{) . instant 工ate(

mbd, beanName , this . beanFactory , constructorToUse , argsT。Use) ;

an e c

＋」

中 m

rι？I＆

r ab pe pa en

(

ea an nC we

nu ←」

入 M

HHHγI牟

巾1N d

2

巾J Nr

品业户口‘ ，

』止 aw

’H

zb

书Z H

HM 卡、 n

MNuer

户J SU

BW刷l· ←L

/

we

lafJ

’Dr

catch (Throwable ex ) [

throw new BeanCreationException(mbd . getResourceDescription() , beanName,”

Instantiation of bean failed”, ex );

逻辑很复杂 ， 函数代码量很大，不知道你是否坚持读完了整个函数并理解了整个功能呢？

这里要先吐个槽，作者觉得这个函数的写法完全不符合 Spring 的一贯风格，如果你一直跟随作

者的分析思路到这里，相信你或多或少对 Spring 的编码风格有所了解， Spring 的一贯做法是

将复杂的逻辑分解，分成 N 个小函数的嵌套，每一层都是对下一层逻辑的总结及概要，这样使

得每一层的逻辑会变得简单容易理解 。 在上面的函数中，包含着很多的逻辑实现，作者觉得至

少应该将逻辑封装在不同函数中而使得在 autowir巳Cons位uctor 中的逻辑清晰明了 。

我们总览一下整个函数，其实现的功能考虑了 以下几个方面 。

l. 构造函数参数的确定 。

根据 巳xplicitArgs 参数判断 。

如果传人的参数 explicitArgs 不为空，那边可以直接确定参数，因为 explicitArgs 参数是在

调用 Bean 的时候用户指定的，在 BeanFactory 类中存在这样的方法 ：

Object getBean(String name , Object .. args) thr。ws BeansExcept1 。n ;

在获取 bean 的时候，用户不但可以指定 bean 的名称还可以指定 bean 所对应类的构造函数

或者工厂方法的方法参数，主要用于静态工厂方法的调用，而这里是需要给定完全匹配的参数

的，所以，便可以判断，如果传人参数 explicitArgs 不为空，则可以确定构造函数参数就是它 。

缓存 中 获取。

除此之外，确定参数的办法如果之前已经分析过 ， 也就是说构造函数参数已经记录在缓存

中，那么便可以直接拿来使用 。 而且，这里要提到的是，在缓存中缓存的可能是参数的最终类

型也可能是参数的初始类型 ，例如：构造函数参数要求的是 int 类型，但是原始的参数值可能

是 String 类型的“ l ” ，那么 即使在缓存中得到了参数，也需要经过类型转换器的过滤以确保参

数类型与对应的构造函数参数类型完全对应。118 第 5 章 bean 的加载

配直文件获取。

如果不能根据传人的参数 explicitArgs 确定构造函数的参数也无法在缓存中得到相关信息，

那么只能开始新一轮的分析了 。

分析从获取配置文件中配置的构造函数信息开始，经过之前的分析，我们知道， Spring 中

配置文件中的信息经过转换都会通过 BeanDefinition 实例承载，也就是参数 mbd 中包含，那么

可以通过调用 mbd.getConstructorArgumentValues（）来获取配置的构造函数信息 。 有了配置中的

信息便可以获取对应的参数值信息了，获取参数值的信息包括直接指定值，如：直接指定构造

函数中某个值为原始类型 String 类型，或者是一个对其他 bean 的引用，而这一处理委托给

resoIveConstructorArguments 方法，并返回能解析到的参数的个数。

2. 构造函数的确定 。

经过了第一步后已经确定了构造函数的参数，接下来的任务就是根据构造函数参数在所有

构造函数中锁定对应的构造函数，而匹配的方法就是根据参数个数匹配，所以在匹配之前需要

先对构造函数按照 public 构造函数优先参数数量降序、 三I~ public 构造函数参数数量降序 。 这样

可以在遍历的情况下迅速判断排在后面的构造函数参数个数是否符合条件 。

由 于在配置文件中并不是唯一限制使用参数位置索引的方式去创建，同样还支持指定参数

名称进行设定参数值的情况，如＜constructor-arg name＝”aa怡，那么这种情况就需要首先确定构

造函数中的参数名称 。

获取参数名称可以有两种方式，一种是通过注解的方式直接获取 ， 另一种就是使用 Spring

中提供的工具类 ParameterNameDiscoverer 来获取。 构造函数、参数名称、参数类型、参数值

都确定后就可以锁定构造函数以及转换对应的参数类型了 。

3. 根据确定的构造函数转换对应 的参数类型 。

主要是使用 Spring 中提供的类型转换器或者用户提供的自定义类型转换器进行转换 。

4 . 构造函数不确定性的验证 。

当然，有时候即使构造函数、参数名称、参数类型 、参数值都确定后也不一定会直接锁定

构造函数，不同构造函数的参数为父子关系，所以 Spring 在最后又做了一次验证。

5. 根据实例化策略以及得到的构造函数及构造函数参数实例化 Bean 。 后面章节中将进

行讲解 。

2. instantiateBean

经历了带有参数的构造函数的实例构造，相信你会非常轻松愉快地理解不带参数的构造函

数的实例化过程 。

protected BeanWrapper instantiateBean(final String beanNarne , final R。otBean

Def 工 ni tion rnbd) {

try {

Object beaninstance ;

final BeanFactory parent = this ;

if (Systern .getSecurityManager() !=null ) (5.7 创建 bean 119

beaninstance = AccessController doPrivileged(new PrivilegedActi 。n

<Object> ( ) (

public Object run () (

return getinstantiationStrategy () . instantiate （口剧， beanNarr晤， parent) ;

}, getAccessControlContext () ) ;

else (

beaninstance = getinstant工ationStrategy () . instantiate (mbd, beanName , parent) ;

BeanWrapper bw =new BeanWrapperirnpl (beanin s tance );

initBeanWrapper(bw) ;

retur 口 b w;

catch (Throwable ex) {

throw new BeanCrea t 工 onException(rnbd . getResourceDescription() , beanNarne,”

Instantiation of bean fa 工 led”, ex) ;

你会发现，此方法并没有什么实质性的逻辑，带有参数的实例构造中， Spring 把精力都放

在了构造函数以及参数的匹配上，所以如果没有参数的话那将是非常简单的一件事 ，直接调用

实例化策略进行实例化就可以了 。

3. 实例化策略

实例化过程中反复提到过实例化策略，那这又是做什么用的呢？ 其实 ， 经过前面的分析，

我们已经得到了足以实例化的所有相关信息，完全可以使用最简单的反射方法直接反射来构造

实例对象，但是 Spring 却并没有这么做。

I Simp怡In阳tiationStr叫y.java I

public Ob] ect insta口 tiate(RootBeanDefinit 工 on beanDefinition , String beanNarne ,

BeanFactory owner) {

／／如果有需要在i盖或者动态替换的方法则 当然需要使用 cg lib 进行动态代理，因为可以在创建代理的同时

／／将动态方法织人类中

／／但是如果没有需要动态改变得方法 ， 为了方便直接反射就可以了

if (beanDefinition . getMethodOverrides () isErnpty()) {

OrFactor yMethod ;

is an interface "),

Constructor<?> constructorToUse ;

synchronized ( beanDefinitio口 . const r uctorArgurnentLock) {

constructorT。Use = ( Constructor ＜。＞） beanDefinition . resolvedConstructor

if ( construct。rT。 Use == null) {

final Class clazz = beanDef 工 nition . getBeanClass() ;

if (clazz . i s Interface ()} {

throw new BeaninstantiationException(claz z ，” Spec 工 fied class

try {

if (System. getSecurityManager () ! = null) {

constructorToUse = AccessCo n trolle r . doP r 工 vi l eged ( new

Privile gedExceptionAction<Construct 。 r> () {120 第 5 章 bean 的加载

publ 工 C Constructor run() thr 。ws Excepti 。n {

ret urn clazz .getDeclaredConstructor ( (Class [] ) null) ;

.,

] )

e lse (

constructoL'l'oUse = clazz . getDeclaredConstruct。r ((Class []) null );

beanDefinition resolvedConstructorDrFact。ryMethod = constructoL'l'oUse,

catch (Excepti 。n e x ) (

thr。w new Bean l nstantiati 。nExcepti 。n ( cla z z ，” N。 default

c。 nstructor found”, ex ),

return BeanUtils .i nstantiateC l ass ( const r uctorT。 Use ) ;

}else {

II Must gener ate CGLIB subclass.

retur n instantiateWithMeth。dinjecti。n ( oeanDefinitio n, beanNarne , ow口er ) ;

CglibSubclass1nginstantiat1 。nStrategy. Java

pu b l 工 C Ob 〕 ect 工 nstantiate ( Constructor ctor , Ob] e c t [] arg s) {

Enhancer enhancer = new Enhancer() ;

enhancer setSuperclass(this . beanDefinit 工口口 . getBeanClass () ) ;

enhancer . setCallbackFilter {new CallbackF工 lte r irnpl () ) ;

enhancer . setCallbacks (new Callback ( ] {

NoOp . INSTANCE ,

new LookupOverrideMethodi nterceptor (),

new ReplaceOverrideMethodinterceptor ()

.,

} )

retur n (ctor == null ) ?

enhancer.create ()

enhancer . create (ctor . getPararneterTypes( ) , args) ;

看了上面两个函数后似乎我们已经感受到了 Spring 的 良苦用心以及为了能更方便地使用

Spring 而做了大量的工作 。 程序中，首先判断如果 beanDefinition.getMethodOverridesO为空也就是

用户没有使用 replace 或者 lookup 的配置方法，那么直接使用反射的方式，简单快捷，但是如果使

用了这两个特性，在直接使用反射的方式创建实例就不妥了，因为需要将这两个配置提供的功能切

入进去，所以就必须要使用动态代理的方式将包含两个特性所对应的逻辑的拦截增强器设置进去，

这样才可以保证在调用方法的时候会被相应的拦截器增强，返回值为包含拦截器的代理实例。

对于拦截器的处理方法非常简单，不再详细介绍，如果读者有兴趣，可以仔细研读第 7 章

中关于 AOP 的介绍，对动态代理方面的知识会有更详细地介绍 。

5.7.2 记录创建 bean 的 ObjectFactory

在 doCreate 函数中有这样一段代码 ：5.7 创建 bean 121

b。clea n earlySing工et。nExp。 sure = (mbd . isSi ngleton () 晶晶 this .allowCircularRefe r ences 晶晶

isSingletonCurrentlyinCreation (beanName) );

if (earlySingletonE xp。su r e ) (

if (logger .isDebugEnabled () ) {

logger .debug (” Eagerly caching bean ’” + beanName +

to allow for resolving potential c工 rcular references ” ) ;

／／为避免后期循环依赖，可以在 bean 初始化完成前将创建实例的 Object Fact 。 r y 加入工厂

addSingletonFactory (beanName , new Ob] ectFactory () {

public Ob] ect getOb] ect () throws Beans Ex cept 工 on (

／／对 bean 再一次依赖引用 ， 主要应用 Sma rtinstantiationA旧re BeanPost Processor,

／／其中我们熟知的 AOP 就是在这里将 advice 7;/J态伊、人 beaη 中， 若没有则直接返回

//b ea口，不做任何处理

return getEarlyBeanReference(beanName , mbd , bean );

.,

} )

这段代码不是很复杂，但是很多人不是太理解这段代码的作用，而且，这段代码仅从此函

数中去理解也很难弄懂其中的含义 ，我们需要从全局的角度去思考 Spring 的依赖解决办法。

earlySingletonExposure ： 从字面的意思理解就是提早曝光的单例，我们暂不定义它的

学名叫什么，我们感兴趣的是有哪些条件影响这个值。

mbd.isSingleton（）：没有太 多可以解释的，此 RootBeanDefinition 代表的是否是羊例 。

this.allowCircularReferences ：是否九许循环依赖，很抱歉，并没有找到在配直文件 中

如何配直，但是在 AbstractRefreshableApplicationContext 中提供了设直函数，可以通

过硬编码的方式进行设直或者可以通过自定义命名空间进行配直，其中硬编码的方式

代码如下 。

ClassPathXr吐且pplicationContext bf = new ClassPath沽nlAppl 工cationContext (” aspectTest .xml” ) ;

bf .setAl l。wBea nDefi n itio n Ove r ri d ing ( false );

isSingletonCurrentlylnCreation(beanName）：该 bean 是否在创建中 。 在 Spring 中，会有

个专门的属性默认为 DefaultSingletonBeanRegistry 的 singletonsCurrentlyInCreation 来记

录 bean 的加载状态，在 bean 开始创建前会将 beanName 记录在属性中，在 bean 创建

结束后会将 beanName 从属性中移除。 那么我们跟随代码一路走来可是对这个属性的

记录并没有多少印象，这个状态是在哪里记录的呢？不同 scope 的记录位直并不一样，

我们以 singleton 为例，在 singleton 下记录属性的函数是在 DefaultSingletonBeanRegistry

类的 public Object getSingleton(String beanName, ObjectFactory singletonFactory） 函数

的 beforeSingletonCreation(beanName）和 afterSingletonCreation(beanName）中，在这两段函

数 中分别 this.singletonsCurrentlyInCreation.add(beanName）与 this . singletonsCurrentlyinCreation.remove(beanName）来进行状态的记录与移除 。

经过以上分析我们了解变量 earlySingletonExposure 是否是单例 、 是否允许循环依赖 、 是否

对应的 bean 正在创建的条件的综合 。 当这 3 个条件都满足时会执行 addS ingletonFactory 操作，

那么加入 SingletonFactory 的作用是什么呢？ 又是在什么时候调用呢？122 第 5 章 bean 的加载

我们还是以简单的 AB 循环依赖为例 ， 类 A 中含有属性类 B ，而类 B 中又会含有属性类 A,

那么初始化 beanA 的过程如图 5 -3 所示 。

开始创建 b阻四记录 bean!也meA)

populatell 田证填充属性）

addSingletonFactory

开始创建 b国n(记录 b国创町eB )

结束创建 bear(移除 b四iNameA)

addSin国 etonFactory

populateB 田式填充属性〕

结束创建 b国n(移除 b自iNameB)

图 5-3 处理循环依赖

图 5 -3 中展示了创建 beanA 的流程，图 中我们看到 ，在创建 A 的时候首先会记录类 A 所

对应的 beanName ，并将 beanA 的创建工厂加入缓存中 ，而在对 A 的属性填充也就是调用 populate

方法的时候又会再一次的对 B 进行递归创建。 同样的，因为在 B 中同样存在 A 属性，因此在

实例化 B 的的 popul ate 方法中又会再次地初始化 B ，也就是图形的最后，调用 getBean(A） 。 关

键是在这里，有心的同学可以去找找这个代码的实现方式，我们之前已经讲过，在这个函数中

并不是直接去实例化 A ，而是先去检测缓存中是否有已经创建好的对应的 bean ，或者是否已经创

建好的 Obj ectFactory ，而此时对于 A 的 Obj ectFactory 我们早已经创建，所以便不会再去向后

执行 ， 而是直接调用 Obj ectFactory 去创建 A。 这里最关键的是 Obj ectFactory 的实现。

addSingletonFactory(beanNarne, new ObjectFactory() {

public Object getObject( ) throws BeansException {

／／对 bean 再二次依赖引用，主要应用 SrnartinstantiationAware BeanPost Process。r ,

／／其中我们熟知的 AOP 就是在这里将 advice z;IJ态织人 bean 中， 若没有则直接返 回

//bean ，不做任何处理

return getEarlyBeanReference (beanNarne , rnbd , bean ) ;

其中 getEarIyBeanReference 的代码如下 ：

protected Ob] ect getEarlyBeanReference(String beanNarne , RootBeanDefinition rnbd, Object bean) {

Ob] ect exposedObject = bean ;

工 f (bean 1= null && 1rnbd . isSynthe tic() && hasinstantiationAwareBeanPostProcessors

) ( ) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof SrnartinstantiationAwareBeanPostProcessor) {

SrnartinstantiacionAwareBeanPostProcessor ibp =

( Srnartinstant 工 ationAwareBeanPostProcessor) bp ;V且 5 .7 创建 bean 123

exposedObject ~ ibp . getEarlyBeanReference(exposedObject , beanName) ;

if (exposedOb] ect == null) {

return exposedObject ;

return exposedOb] ect ;

在 getEarlyBeanReference 函数中并没有太多的逻辑处理，或者说除了后处理器的调用外没

有别的处理工作，根据以上分析， 基本可以理清 Spring 处理循环依赖的解决办法， 在 B 中创

建依赖 A 时通过 Obj ectFactory 提供的实例化方法来中断 A 中的属性填充 ， 使 B 中持有的 A

仅仅是刚刚初始化并没有填充任何属性的 A ， 而这正初始化 A 的步骤还是在最开始创建 A 的

时候进行的，但是因为 A 与 B 中的 A 所表示的属性地址是一样的 ， 所以在 A 中创建好的属性

填充自然可以通过 B 中的 A 获取，这样就解决了循环依赖的问题。

5.7.3 属性注入

在了解循环依赖的时候 ， 我们 曾经反复提到了 populateBean 这个函数 ， 也多少了解了这个

函数的主要功能就是属性填充，那么究竟是如何实现填充的呢？

protected void populateBean(String beanName , AbstractBeanDefinition mbd , BeanWrapper bw) {

PropertyValues pvs = mbd . getPropertyValues () ;

if (bw == null) {

if ( !pvs . isEmpty () ) {

throw new BeanCreationException(

mbd.getResourceDescription() , beanName ， ”Cann。t apply property

values to nul l 工nstance” ) ;

else {

／／没有可填充的属性

re tu z口 ；

／／给 Instant 工 atio础wareBeanPostProcessors 最后一次机会在属性设置前来改变 bean

／／如 ·可以用来支持属性注入的类型

boolean continueWithPropertyPopulation = true ;

if ( 'mbd isSynthet工C () && hasinstantiat 工onAwareBeanPostProcessors () ) {

for (BeanPostPr。cessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp =

(Instant 工 ationAwareBean PostProcessor) bp ;

／／返回值为是杏继续填充 bean

if (1ibp . postProcessAfterinstantiation(bw . getWrappedinstance() ,

beanName)) {

con t 工 nueWithPr。pertyPopulatio口＝ false ;V且 124 第 5 章 be an 的 加载

break ;

／／如果后处理器发出停止填充命令则终止后续的执行

if I' c。n tinueWithPropertyP。pulation ) (

return ,

if ( mbd . getResolvedAutow工 reMode( ) == RootBeanDefinitio口 . AUTOWIRE\_BY\_NAME I I

mbd. getResolvedAutowireMode () == RootBeanDef1n1tion .AUTOWIRE BY TYPE) {

MutablePr。pertyValues newPvs =new MutablePropertyValues (pvs ) ;

II Add property values based on autowire by name if applicable .

／／根据名称自动注入

if (mbd .getResolvedAutowireMode( ) == RootBeanDefinition .AUTOWIRE BY NAME) (

aut。wireByName (beanName , mbd , bw , newPvs);

II Add property values based on autowire by type 工f applicable .

／／根据类型自动注人

if (mbd .getResolvedAutowireMode () == RootBeanDefinition .AUTOWIRE BY TYPE) (

aut。wireByType ( beanName , mbd , bw , newPvs );

pvs = newPvs;

／／后处理然已经初始化

b。 olean hasinstAwareBpps = hasinstantiationAwareBeanPostProcessors( ) ;

／／需要依赖检查

boolean needsDepCheck = (mbd . getDependencyCheck () ' = Ro。 tBeanDefinition .

DEPENDENCY CHECK NONE) ;

if (hasinstAwareBpps I I needsDepCheck) (

E'ropertyDescriptor[J filteredPds = filterPropertyDescriptors ForDependency

Check(bw) ;

if (hasinstAwareBpps) (

for (BeanPostProcessor bp : getBeanPostProcessors ()) (

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBean

PostProcessor ) bp ;

／／对所有需要依赖检查的属性进行后处理

pvs = ibp . postProcessPropertyValues(pvs , filteredPds , bw .

getWrappedinstance() , beanName) ;

if (pvs == null) (

return ;

if (needsDepCheck) (V且 5 .7 创建 bean 125

／／依赖检查， 对应 depends - o n 属性 ， 3 . 0 已经弃用此属性

checkDependencies (beanName , mbd , filteredPds , pvs );

／／将属性应用到 bean 中

applyPr。pertyValues (beanName , mbd , b w, pvs) ;

在 populateBean 函数中提供了这样的处理流程 。

1. InstantiationAwareBeanPostProcessor 处理器的 postProcessAfterinstantiation 函数的应用 ，

此函数可以控制程序是否继续进行属性填充。

2. 根据注人类型（ byName/byTyp e ），提取依赖的 bean ，并统一存入 PropertyValues 中 。

3. 应用 InstantiationAwareBeanPostProcessor 处理器的 postProcessPrope町Values 方法 ， 对属性

获取完毕填充前对属性的再次处理，典型应用是 RequiredAnnotationBeanPostProcessor 类中对

属性的验证。

4. 将所有 PropertyValue s 中 的属性填充至 BeanWrapper 中 。

在上 面 的 步骤 中 有 几个地方是我们 比较感兴趣的，它们 分别是依赖注入（ autow ireByName/autowireByType ） 以及属性填充 ， 那么 ， 接下来进一步分析这几个功能的实现细节 。

1. autowireByName

上文提到根据注人类型（ byName/ byType ），提取依赖的 bean ，并统一存入 PropertyValues

中，那么我们首先了解下 byName 功能是如何实现的 。

p r otected 飞ra i d autowireByName (

String beanName , AbstractBeanDefiniti on mbd, BeanWr apper bw , Mutable Property

Values pvs ) (

／／寻找 bw 中需要依赖注人的属性

String[] propertyNames = unsat 工 sfiedNonSimpleProperties(mbd , bw );

f。 r (St r i ng p r opertyName : propertyNames ) (

if (containsBean (propertyName )) (

／／递归初始化相关的 bean

Obj e ct bean = getBean (propertyName );

pvs . a dd (propertyName , bean) ;

／／注册依赖

regi sterDepe ndentBean( propertyName , beanName) ;

if (logger . isDebugEnabled ()) (

logger .debug (”Added autowiring by name fr。m bean name ’ ” + beanName +

”’ via property ’” + propertyName + ”’ to bean named E ” +

p ropert y Name + ”’ ”) ,

else (

if (logger . isTraceEnabled () ) (

logger . tra ce (” Not autowiring property ’” + propertyName ＋ ” t 。f

bea n ’” + bea nName +

”’ by name · n。 matching bean found " ) ;V且 126 第 5 章 bea n 的加载

如果读者之前了解了 autowire 的使用方法 ，相信理解这个函数的功能不会太困难 ，无非是

在传人的参数 pvs 中找出已经加载的 bean ， 并递归实例化 ，进而加入到 pvs 中 。

2. autowireByType

autow ireByType 与 autowireByName 对于我们理解与使用来说复杂程度都很相似 ，但是其

实现功能的复杂度却完全不一样。

protected void autowireByType(

String beanNarne , AbstractBeanDef工 nition rnbd , BeanWrapper bw, MutablePr。perty

Values pvs ) {

Typ e C 。nverter converter= getCustornTypeConverter () ;

if (converter == null ) {

converter = bw ;

Set<String> autowiredBeanNarnes =new LinkedHashSet<String> (4 ) ;

／／寻找 bw 中需要依赖注入的属性

String[] propertyNarnes = unsatisfiedNonSirnpleProperties (rnbd , bw ) ;

for (String propertyNarne propertyNarnes) (

try {

PropertyDescriptor pd = bw . getPropertyDescriptor (propertyNarne) ;

II D。n ’ t try autowi r工 ng by type for type Ob] ect neve r makes sense ,

II even if it technically i s a unsatisfied , non simple property .

if ( 10b] ect . clas s . equals (pd . getPropertyType () ) ) (

／／探测指定属性的 set 方法

MethodPararneter rnethodPararn = BeanUtils .getWriteMethod Pararneter (pd);

boolean eager= 1Priority0rdered .class ． 工sAssignableFrorn (bw . get

WrappedClass() );

DependencyDescr i ptor desc = new AutowireByTypeDependencyDescriptor

(rnethodPararn , eager) ;

／／解析指定 bean Name 的属性所匹配的值 ， 并把解析到的属性名称存储在

llautowiredB ean Narne s 中 ， 当属性存在多个封装 b ean H忖 rl :

ll@Autowired private List<A> aList ， 将会找到所有匹配 A 类型的 bean 并将

／／其注入

Ob] ect autowiredArgurnent = resolveDependency (desc , beanNarne ,

autowiredBeanNarnes , converter ) ;

i f (autowiredArgurnent ' = null ) {

pvs . add(propertyNarne , autow 工 redArgurnent) ;

f。 r (String autowiredBeanNarne : aut 。wiredBeanNarnes) {

／／注册依赖

regi s terDependentBean (autowiredBeanNarne , beanNarne ) ,

if (l ogger i s DebugEnabled ()) {

l ogg e r . de bug (” Autowiring by type from bea n name ’” +

beanName + ” ’ via property ’ ” +

propertyName + ”’ t o bean named ’ ” +

autowiredBeanName + ””’),

V且 5 .7 创建 bean 127

autowiredBeanNames clear( ),

catch ( BeansExceptioηex) {

throw new Unsatisf iedDepe口dencyException ( mbd . getResourceDescript 工on() ,

beanName , p r opertyName , ex );

实现根据名称自动匹配的第一步就是寻找 bw 中需要依赖注入的属性，同样对于根据类型

自动匹配的实现来讲第一步也是寻找 bw 中需要依赖注入的属性，然后遍历这些属性并寻找类

型匹配的 bean ， 其中最复杂的就是寻找类型匹配的 bean 。 同时， Spring 中提供了对集合的类型

注入的支持，如使用注解的方式：

@Au tow ired

private List<Test> tes ts ;

Spring 将会把所有与 Test 匹配的类型找出来并注入到 tests 属性中，正是由于这一因素，

所以在 autowireByType 函数中，新建了局部遍历 autowiredBeanNames ，用于存储所有依赖的

bean ，如果只是对非集合类的属性注入来说，此属性并无用处 。

对于寻找类型匹配的逻辑实现封装在了 resolveDependency 函数中 。

| D伽 ltli由协BeanFa伽叩java J

public Ob] ect resolveDependency(DependencyDescriptor descr 工 ptor , St E工 ng bea nName ,

Set<String> autowiredBeanNames , TypeConverter typeConverter ) throws Bean s

Exception {

descriptor － 工 nitParameterNameDiscovery ( getParameterNameDiscoverer ());

工 f (descriptor getDependencyType () . equals (Object Factory . class) ) {

//ObjectFactory 类注入的特殊处理

return new DependencyObjectFactory (descriptor , beanName) ;

else if (descriptor . getDependencyType () .equals (] avaxinjectProviderClass) ) {

//javaxin 〕 ectProviderClass 类注入的特殊处理

return new DependencyProviderFactory () . createDependencyProvider ( descript。r ,

beanName) ;

else {

! l:illi用处理逻辑

return d。Res。lveDependency ( descriptor , descriptor getDependeηcyType (),

beanName, a u tow 工 redBeanNames , typeConverte r);

protected Ob] ect doResolveDepeηde n cy ( Depe n dencyDescript。r descriptor , Class<?> type ,

String beanName ,

Set<String> autowiredBeanNames , TypeConverter typeConve r ter ) throws Be ans

Except 工 on {

制

V且 128 第 5 章 bean 的加载

/\*

食用于支持 Spr i ng 中新增的注解＠ Value

去／

Object value = getAutow工 reCandidateResolver() . getSuggestedValue(descr 工 ptor);

工 f (value 1= null) {

if (value instanceof String) (

String strVal = resolveEmbeddedValue((String ) value) ;

BeanDefinition bd = (beanName 1= null && containsBean (beanName) ? get

MergedBeanDefinit L。n (beanName ) : null );

value = evaluateBeanDe f 工n工 t 工 onString(strVal , bd) ;

TypeConverter converter = (typeC。n verter '= null ? typeConverter

getTypeConverter());

return converter . convertifNecessary (value, type);

／／如果解析器没有成功解析， 则需要考虑各种情况

／／属性是数组类型

if (type.isArray()) {

Class<?> c。mp。nentType = type .getC。mpone ntType ();

／／根据属性类型找到 bean Fa cotry 中所有类型的匹配 bean ,

／／返回值的构成为 key＝匹配的 beanName , value=beanName 对应的实例化后的 bean （通过

//getBean(beanName ）返 回 ）

Map<String , Object> matchingBeans = findAutowireCa ndidate s(beanName ,

c。mpoηentType , descriptor) ;

if (matchingBeans. isEmpty ()) {

／／如果 au tow ire 的 require 属性为 true 而找到的匹配项却为空则 只能抛出异常

if (descriptor.isRequired()) {

raiseNoSuchBeanDe f 工n itionException (componentType , ” array o f ” +

componentType . getName() , descriptor) ;

return null ;

if (autowiredBeanNames != null) {

autowiredBeanNames . addAll (matchingBeans . keySet ());

TypeConverter converter = (typeConverter 1= null ? typeConverter

getTypeConverter());

／／通过转换器将 bean 的值转换为对应的 type 类型

return converter . convertifNecessary(matchingBeans . values () , type ) ;

／／属性是 Collection 类型

else if (Collecti 。n.class.isAssignableFrom(type) && type.isinterface()) {

Class< ?> elementType = descriptor . getCollectionType ();

if (elementType == null) {

if (descr工ptor isRequired ()) {

throw new FatalBeanException ( ” No element type declared for

collection [ ” + type . getName() + ” ] ” ) ;

return null ;

Map<String , Object> matchingBeans = findAutowireCand工dates(beanName ,

elementType , descriptor) ;

销

U局 5 . 7 创 建 be an 129

if (matchingBeans. 工sEmpty ()) {

if (descr工ptor. isRequired ()) (

ra 工 seNoSuchBeanOefiniti 。 nException(elementType ，” collecti 。n of ”

+ elementType.getName() , descriptor) ;

return null;

if (autowiredBeanNames ！~ null) {

autowiredBeanNames . addAll(matchingBeans . keySet()) ;

TypeConverter converter = (typeConverter !=null ? typeConverter

getTypeConverter()) ;

return converter . convertifNecessary(match 工口gBeans . values() , type );

／／属性是 Map 类型

else if (Map . class.isAss 工 gnableFrom (type) && type . isinterface ()) (

Class<?> keyType = descriptor . getMapKeyType() ;

if (keyType ==null I I 1String.class . isAssignableFrom(keyType)) {

if (descriptor . isRequired()) {

throw new FatalBeanException (" Key type [ ” + keyType + ” J of

map [ ” + type. get Name() +

” ] must be ass 工 gnable to [java.lang . String ] ” );

return null ;

Class<?> valueType = descriptor . getMapValueType() ;

if (valueType == null) (

if (descript。 r . isRequired ()) {

thr 。w new FatalBeanException (” No value type declared for map [ ” +

type . getName() + ” ] ” );

return null;

Map<String , Object> matchingBeans = findAu t。wireCandidates(beanName ,

valueType , descriptor) ;

if (match 工ngBeans . isEmpty () ) (

if (descr 工ptor . isRequired () ) (

raiseNoSuchBeanDefinitionExcepti 。n(valueType ，” map with value type

’ + valueType.getName() , descriptor);

return null ;

if (autow工redBeanNames != null) {

autowiredBeanNames . addAll(matchingBeans keySet ());

return match 工 ngBeans ;

)else {

Map<St r 工 ng, Object> matchingBeans = f 工ndAutowireCandidates ( beanName , type ,

descriptor) ;

工 f (match 工 ngBeans . isEmpty ()) (

if (descriptor. isRequired ()) (

raiseNoSuchBeanDefinitionException(type,””, descriptor) ;

return null ;

制

U局 130 第 5 章 bean 的力日载

if (match 工ngBeans . s 工 ze () > 1) (

String pr i maryBe anName = determinePrimaryCand工date ( matchingBeans ,

des er 工ptor) ;

if (pr1maryBeanName == null) {

throw new NoSuchBeanDefinitionExcept L。n(type ，” expected single

matching bean but f。und ” ＋

matchingBeans . size() + ”:” + matchingBeans keySet ()) ;

if (autow1redBeanNames ! = null) (

autowiredBeanNames . add(primaryBeanName) ;

return matchingBeans get (primaryBeanName) ;

／／已经可 以确定只有一个匹配项

Map . Entry<String , Object> e口 try = matchingBeans .entrySet () . iterator （）.口ext() ;

if (autowiredBeanNames != null) {

autowiredBeanNames . add(entry . getKey ()) ;

returr飞 entr y. getValue ();

寻找类型的匹配执行顺序时，首先尝试使用解析器进行解析，如果解析器没有成功解析，

那么可能是使用默认的解析器没有做任何处理， 或者是使用了自 定义的解析器，但是对于集

合等类型来说并不在解析范围之内，所以再次对不同类型进行不同情况的处理，虽说对于不

同类型处理方式不一致，但是大致的思路还是很相似的，所以函数中只对数组类型进行了详

细地注释 。

3. applyP 「ope「tyValues

程序运行到这里，已经完成了对所有注入属性的获取，但是获取的属性是以PropertyValues

形式存在的，还并没有应用到已经实例化的 bean 中 ， 这一工作是在 applyPropertyValues 中 。

protected void applyPropertyValues( String beanName , BeanDefin工 tion mbd , BeanWrapper bw ,

PropertyVa l ues pvs) (

if (pvs == null I I pvs . isEmpty ()) (

return ;

Mutable Pr。pertyValues mpvs = null;

List<PropertyValue> original ;

if (System . getSecur 工tyManager () ! = null) (

if (bw instanceof BeanWrapperimpl ) (

( (BeanWrapperimpl) bw) setSecurityContext (getAccessControlContext ());

工 f (pvs instanceof MutablePropertyValues) (

mpvs = (MutablePropertyValues) pvs ;

／／如果 mpvs 中的值已经被转换为对应的类型那么可 以直接设置到 beanwappe r 中

目

U局 5 .7 创建 bean 131

if (mpvs . isConverted ()) {

II Shortcut · use the pre-converted values as- is .

try {

bw . setPr。pertyValues(mpvs);

return ,

catch (Bea口 sExcept 工 on ex) {

throw new BeanCreationException(

mbd . getResourceDescription( ), beanName , ” Error set t 工 ng

pr 。perty values ”, ex) ;

original = mpvs . getPr。pertyValueList ( ) ;

)else {

／／如果 pvs 并不是使用 MutablePropertyValues 封装的类型，那么直接使用原始的属性获取方法

original= Arrays . asList (pvs . getPropertyValues()) ;

TypeConverter converter= getCustomTypeConverter() ;

if (converter == null) {

converter = bw ;

／／获取对应的解析器

BeanDefiniti。nValueRes。lver valueResolver =new BeanDefinitionValueRe solver(t h 工 s ,

beanName , mbd , converter) ;

II Create a deep copy , resolving any references for values .

List<PropertyValue> deepCopy =new ArrayList<PropertyValue> (o r iginal .size {));

b。olean resolveNecessary = false ,

／／遍历属性，将属性转换为对应类的对应属性的类型

for (PropertyValue pv : orig 工 nal) {

if {pv . isConverted ()) {

deepCopy . add(pv) ;

)else {

St r工 ng propertyName = pv . getName() ;

Object or 工q 工 nalValue = pv .getValue() ;

Object resolvedValue = valueResolver .re s。lveValueifNecessary ( pv ,

。 rig 工 nalValue) ;

Object convertedValue = res 。 lvedValue ;

boolean convertible= bw . isWritableProperty(propertyName) &&

' PropertyAccessorUtils .isNestedOrindexedProperty (propertyName ) ;

if (convertible) {

convertedValue = convertForProperty(res 。 lvedValue , propertyName ,

bw , converter) ;

if (resolvedValue == originalValue) {

if (convertible) {

pv . setC。 nvertedValue(convertedValue) ;

deepCopy . add(pv) ;

else if (convertible ι ＆ or 工 gin alValue instanceof TypedStringValue 晶晶

1 ( (TypedStringValue ) original Value) . is Dynamic () &&

1 (convertedValue instanceof Collect 工 on 11 Ob] ectUtils.isArray

(convertedValue) )) {

制132

values ”, ex),

第 5 章 bean 的加载

pv . setConvertedValue (convertedValue );

deepCopy.add(pv) ;

else (

resolveNecessary = true ;

deepCopy . add(new PropertyValue(pv , convertedValue)) ;

if (mpvs != null && !res。lveNecessary) {

mpvs setConverted() ;

try {

bw.setPropertyValues(new MutablePropertyValues(deepCopy));

catch (BeansException ex ) {

throw new BeanCreat 工 onException(

mbd . getResourceDescr 工 ption() , beanName,”Error se t ting property

5.7.4 初始化 bean

大家应该记得在 bean 配置时 bean 中有一个 init-method 的属性，这个属性的作用是在 bean

实例化前调用 init-method 指定的方法来根据用户业务进行相应的实例化。 我们现在就已经进入

这个方法了，首先看一下这个方法的执行位置 ， Spring 中程序已经执行过 bean 的实例化，并且

进行了属性的填充，而就在这时将会调用用户设定的初始化方法。

protected Ob] ect initializeBean(final String beanName , final Ob] ect bean, RootBean Definit工on

mbd ) {

if (System . getSecurityManager () != null ) {

AccessController.doPrivileged(new Pr 工 vilegedAction<Object> () {

publ 工 C Object run () {

inv。keAwareMeth。ds( beanName, bean );

return null ,

) , getAccessControlContext( ));

else {

／／对特殊的 bean 处理 ： Aware 、 BeanClassLoaderAware 、 BeanFactoryAware

inv。keAwareMeth。ds (beanName, bean) ;

Object wrappedBea口＝ bean;

if (mbd == null I I 1mbd. isSynthetic () ) {

／／应用后处理器

wrappedBean = apply』a且P。stl'rocess。rsBef。reinitialization (wrappedBean , beanName);

try {

目

5.7 创建 bean

／／激活用户自定义的工 nit 方法

inv。ke 工nitMeth。ds (beanName , wrappedBean , mbd );

catch (Throwable ex) {

throw new BeanCreationException(

(mbd 1= null ? mbd.getResourceDescript l。 n() : null) ,

bean Name,”I nvocation of init method failed”, ex) ;

if (mbd ==null 11 1mbd.isSynthetic()) {

／／后处理器应用

133

wrappedBean = applyBeanPostpr。cess。rsAfter工nitializat工。n ( wrappedBean , beanName) ;

return wrappedBean;

虽然说此函数的主要目 的是进行客户设定的初始化方法的调用，但是除此之外还有些其他

必要的工作。

1. 激活 Aware 方法

在分析其原理之前，我们先了解一下 Aware 的使用。 Spring 中提供一些 Aware 相关接口，比如

BeanFactoryAware 、 ApplicationContextAware 、 ResourceLoaderAware 、 ServletContextAware 等，实现这

些 Aware 接口的 bean 在橱刀始之后，可以取得一些相对应的资源，例如实现 BeanFactoryAware 的 bean

在初始后， Spring 容器将会注入 BeanFactory 的实例，而实现 ApplicationContextAware 的 bean ，在

bean 被初始后，将会被注入 ApplicationContext 的实例等。我们首先通过示例方法来了解一下 Aware

的使用。

1. 定义普通 bean 。

public class Hello {

public 飞roid say () {

System . out . println （ ” hell。 ” ）；

2. 定义 BeanFactoryAware 类型的 bean 。

public class Test implements BeanFactoryAware {

private BeanFactory beanFactory ;

／／声明 bean 的时候 Spring 会自 i;IJ注入 Bean Factory

自 Override

public void setBeanFactory(BeanFact。ry beanFactory) throws BeansExcept 工 on {

this . beanFactory = beanFactory ;

public void testAware () {

／／通过 hello 这个 bean id 从 beanFactory 获取实例

Hello hello= (Hello) beanFactory.getBean （ ” hell。 ” ） ，

hello . say();

制134

第 5 章 bean 的加载

3. 使用 main 方法测试 。

public static vo 工d main (String[] s) (

Applicati。nContext ctx = new ClassPathXmlApplicati。nContext （ ” appl1cat1。nContext . xml ” ）；

Test test = (Test ) ctx . ge tBean (” test ” ) ;

test . testAware() ;

运行测试类，控制 台输出：

hel l。

按照上面的方法我们可 以获取到 Spring 中 BeanFactory ， 并且可以根据 BeanFactory 获取

所有 bean ， 以及进行相关设置 。 当然还有其他 Aware 的使用方法都大同小异 ，看一下 Spring

的实现方式， 相信读者便会使用了 。

pr 工 vate void invokeAwareMethods(final String beanName , f 工 nal Obj e ct bean ) {

Fact ory . t his ) ;

if (bean instanceof Aware ) {

if (bean instance of BeanNameAware ) {

(( BeanNa meAwa re ) bean ) . setBeanName (beanName ) ;

if (bean 工nstanceof BeanClassL。aderAware ) {

( (BeanClassLoaderAware ) bean ) . setBeanClassLoader (ge tBeanClassLoader ());

i f (bea n instance of BeanFact。ryAware ) {

((BeanFacto ryAware ) bean) setBeanFactory ( Abst ractAut 。wire CapableBean

代码简单得已经没有什么好说的了 。 读者可以自己尝试使用别的 Aware ， 者IS比较简单。

2 . 处理器的应用

BeanPostProcessor 相信大家都不陌生，这是 Spring 中开放式架构中一个必不可少的亮点，

给用户 充足的权限去更改或者扩展 Spring ，而除 了 BeanPostProcessor 外还有很多其他的

PostProcessor ， 当然大部分都是 以此为基础， 继承自 BeanPostProcessor 。 BeanPostProcessor 的

使用位置就是这里，在调用客户 自 定义初始化方法前以及调用自定义初始化方法后分别会调用

BeanPostProces s or 的 postProcessBeforelnitialization 和 postProcessAfterlnitia li zation 方法 ，使用

户可以根据自己的业务需求进行响应的处理。

pub l 工 C Ob j ect applyBeanPostProcessorsBe forei n it i ali zat i on (Object exis t ingBean , Str 工 ng

beanName )

thr。ws BeansEx c ept 工 o n {

Ob] ect r esult = existingBean ;

for (Bea nPo s tProcessor bea nProcesso r : getBeanPostPr ocess ors ()) {

result = beanPr ocesso r .p。stPr。cessBef。 re工ni tializati。n ( result , bear>Name );

if (result == null ) {

return resul L

return result ;

前5 .7 创建 bean 135

public Object applyBeanPostProcessorsAfterinitializati 。n (Ob ject existingBean , String

beanName)

throws BeansExceptio口 ｛

Ob] ect result = existingBean ;

for (BeanPostProcessor beanProces sor getBeanPostProcessors ()) {

result = beanProcessor. p。stPr。ces sAfterinitializati。n ( result , beanName) ;

if (result == null) {

return result ;

return result ;

3. 激活自定义的 in it 方法

客户定制的初始化方法除了我们熟知的使用配置 init-method 外，还有使 自定义的 bean 实

现 InitializingBean 接口，并在 afterPropertiesSet 中实现自己 的初始化业务逻辑。

init - method 与 afterPropertiesSet 都是在初始化 b巳an 时执行，执行顺序是 afterPropertiesSet

先执行，而 init-method 后执行。

在 invokelnitMethods 方法中就实现了这两个步骤的初始化方法调用 。

protected void invokeinitMethods (String beanName, final Ob] ect bean, RootBeanDefinit工on mbd)

throws Throwable {

／／首先会检查是否是 InitializingBean ， 如果是的话宿要调用 afterPropertiesSet 方法

boolean isinitializingBean = (bean instance。f Ini t 工a l 工 Z ingBean) ;

工 f (isinitializingBean && (mbd == null 11 ' mbd . isExternally ManagedinitMethod

( ” afterPropertiesSet ” ))){

beanName + ””’);

<Object>() {

工 f (logger.isDebugEnabled()) {

logger . debug （ ” Invok工 ng afterPropertiesSet () on bean with name ’” +

if (System . getSecurityManager () != null) {

try {

AccessController .doPrivileged(new PrivilegedExceptionAct1on

public Object run () throws Exception (

( (InitializingBean) bean) . afterPropertiesSet ();

return null ;

}, getAccessControlContext ()) ;

catch (PrivilegedActionException pae} {

throw pae . getException ();

}else (

／／属性初始化后的处理

( (I n 工 t ializingBean) bean) . afterPropertiesSet() ;

盼136 第 5 章 bean 的力日载

if (rnbd ' = null) {

String initMeth。dName = mbd . get IηitMeth。dName () ;

if (initMethodName != null && ' (isinitializingBean && "afterPropertiesSet ” -

equals (initMethodName)) &&

!mbd.isExternallyManagedinitMethod (initMethodName ) ) {

／／调用 自定义初始化方法

invokeCustominitMethod (beanName , bean , mbd ),

5.7.5 注册 DisposableBean

Spring 中不但提供了对于初始化方法的扩展人 口 ， 同样也提供了销毁方法的扩展入口，对

于销毁方法的扩展，除了我们熟知的配置属性 destroy-method 方法外 ，用户还可以注册后处理

器 DestructionAwareBeanPostProcessor 来统一处理 bean 的销毁方法，代码如下 ：

p r otected v。 id registerDisposableBeanifNecessary (String beanName , Object bean ,

RootBeanDef in 工 t ion mbd) (

() : null) ;

() , ace )) ;

AccessControlContext ace = (System. getSecurityManager () '= null ? getAccessControlContext

if (!mbd . isPrototype () 晶晶 requires Destruction (bean , mbd) ) {

if (rnbd. isSingleton ()) {

／舍

食单例模式下注册需要销毁的 bean ， 此方法中会处理实现 DisposableBean 的 bean ,

女并且对所有的 bean 使用 DestructionAwareBeanPostProcessors 处型

\* DisposableBean Destruct 工。nAwareBeanPostProcessors ,

女／

registerDisp。sableBean ( beanName ,

)else {

/\*

new DisposableBeanAdapter {bean, beanNar回， mbd , getBeanPostPr。cessors

食自定义 scope 的处理

\*/

Scope scope= this . scopes . get {mbd . getScope ());

if (SC。pe == null) {

throw new IllegalStateExcept 工on (” No Sc。pe registered for scope ’”

+ mbd . getScope() + ”’” ) ;

scope . registerDestructi。nCallback ( beanName ,

new DisposableBeanAdapter ( be缸飞， beanName , mbd, getBeanPostProcessor s

( ) , acc)) ;