Don’t call us, we will call you.

Spring的IoC容器

《Spring 揭秘》精选版

王福强 著

InfoQ企业软件开发丛书

免费在线版本

（非印刷免费在线版）

InfoQ 中文站出品

InfoQ中文站
www.infoq.com/cn

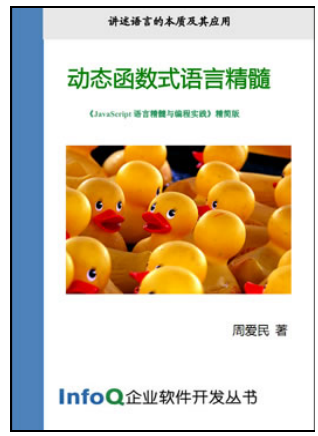
本迷你书主页为

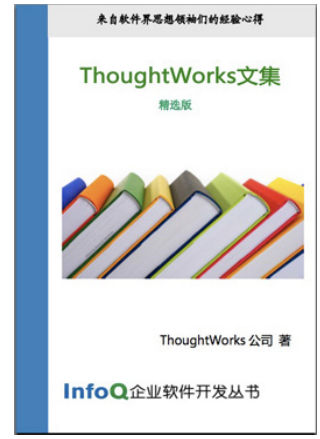
http://infoq.com/cn/minibooks/spring-ioc

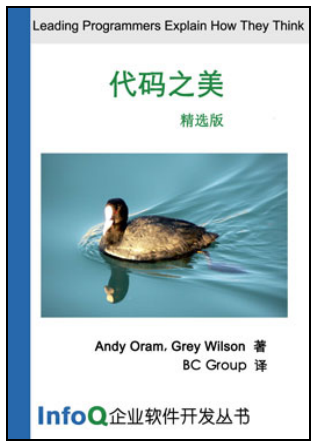
目录

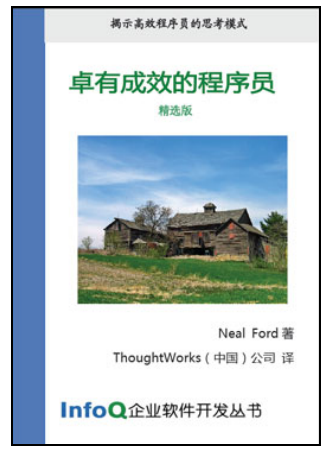
InfoQ企业软件开发丛书

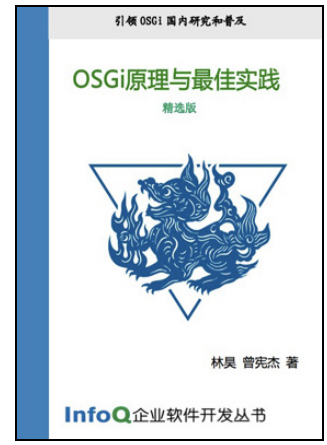
欢迎免费下载











商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com

第1章 Spring框架的由来

本章内容

- Spring之崛起
- Spring框架概述
- Spring大观园

J2EE作为一种企业级应用开发平台，其优异表现是我们有目共睹的。但纵使是最为强大的军队，如果没有一个好的指挥官，不知道如何发挥这支军队的强大战斗力，那这支军队也不可能取得太多辉煌的战果。在J2EE平台的一些早期实践中，就出现了对J2EE平台所提供的各项服务的滥用，将基于J2EE平台的企业级开发带入了窘境。

Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson在其著作*Expert One-On-One J2EE Development and Design*中阐述的部分理念和原型衍生而来。它的最初目的主要是为了简化Java EE的企业级应用开发，相对于过去EJB[①]时代重量级的企业应用开发而言，Spring框架的出现为曾经阴霾的天空带来了灿烂的阳光。

1.1 Spring之崛起

在中世纪的欧洲，当重装骑兵所向披靡时，哪国的军队中如果没有一支重装骑兵真的会让人笑话的，按照电影《大腕》里的一句话说“你都不好意思跟人打招呼”。应该说，在当时的历史/军事环境下，重装骑兵在军队中确实发挥了不可或缺的作用。有时候，一次关键时刻的重装骑兵冲锋就可以奠定战局的胜利。但是，时过境迁，历史的车轮一直在向前缓缓行进，重装骑兵头上的光环也随之渐趋黯淡，其缺点开始显露无遗。

- 重装骑兵代价高昂。一名重装骑兵的装备花费几乎能够武装一小队轻步兵，对于财力不够雄厚的国家来说，维持一支常备的重装骑兵队伍绝非易事。实际上，对于财力雄厚的大国（相当于IT界的IBM、微软）来说，为了减轻财政上的压力，通常也是将这部分花销尽量摊派给贵族。
- 兵种自身限制太多。沉重的盔甲以及一整套装备使得重装骑兵的机动性和灵活性大打折扣，在正式投入战斗之前，重装骑兵需要很长时间的列装和部署，对于瞬息万变的战场形势来说，某些情况下，这等同于自杀。
- 发挥作用的场景有限。纵使各翼军队能够掩护重装骑兵完成部署，但如果战场地形不适合重装骑兵冲锋，那也就无法让他们大显身手，前期的准备或者战斗掩护就更是得不偿失。

当新的战术和兵种能够更加高效地完成作战任务时，依然主打重装骑兵的军队能取得什么样的战

① 此后提到的EJB通常都是指1.*x*和2.*x*版本的EJB，因为EJB3现在已经受Spring框架的影响，也主打基于POJO的轻量级应用解决方案了。

果，轻骑兵的出现已经给了我们一个明确的答案。

当然，我的本意不在讲战争史，我只是想让大家看到，在早期的J2EE平台开发实践过程中，盲目地推崇某一“兵种”，比如EJB，将会招致怎么样的一种命运。历史是相似的，不管是军事发展史，还是我们的J2EE平台企业级应用开发史。君不见当年言必称EJB的盛况吗？这跟中世纪欧洲重装骑兵的显赫是何等地相似。但任何技术或者事物都有其适用的场景，如果用在了不合适的场景下，我们就不得不为滥用而付出相应的代价。EJB是使用J2EE平台各项服务的一种方式，但不是唯一的方式。对于分布式系统来说，使用EJB在某些方面确实可以带给我们很大的收益，但并不是所有的J2EE应用都要用于分布式环境。如果不分场景地滥用EJB，J2EE平台上的这支“重装骑兵”的局限性自然会暴露出来。

- 使用EJB，通常也就意味着需要引入拥有EJB Container的应用服务器（J2EE Application Server）的支持，比如BEA的WebLogic或者IBM的WebSphere，而这往往也就意味着高昂的授权费用。虽然开源的JBoss等应用服务器也提供对EJB Container的支持，但对于商业软件来说，出于产品质量和售后服务等因素考虑，商业产品WebLogic或者WebSphere依然是最佳选择。这跟当年欧洲崇尚重装骑兵相似，都免不了要付出高昂的代价。
- 说到重装骑兵列装和部署的复杂和迟缓，使用EJB开发的企业级应用也存在同样的问题。使用EJB使得应用程序的部署和测试都更加困难，复杂的类加载机制、复杂的部署描述符、过长的开发部署周期等，无疑都增加了开发EJB应用程序的难度。
- 重装骑兵需要合适的战机才能发挥其最大作用，EJB也是如此。只有在分布式的场景中，EJB才会带给我们最大的益处。但是，当大家在崇拜EJB的狂热氛围之下，不分场景地滥用它时，后果自然可想而知了。要么是开发过程缓慢，无法如期交付；要么就是程序交付后性能极差，很难满足客户需求。

当然我们还可以列举更多，但这些已经可以说明问题了。应该说，基于EJB的应用开发实践并非一无是处。尽管其本身在合适的场景下也或多或少地存在一些问题，但不分场景地滥用它，才会导致基于EJB的应用开发声名狼藉。历史的车轮时刻都没有停下，当人们意识到大部分J2EE应用开发初期甚至整个生命周期内都不需要牵扯到分布式架构时，J2EE平台上对应EJB而生的“轻骑兵”自然也该千呼万唤始出来了——倡导轻量级应用解决方案的Spring，就承担了这一历史角色！

Spring倡导一切从实际出发，以实用的态度来选择适合当前开发场景的解决方案。如果不需要用到分布式架构，那就没有必要使用EJB之类的“牛刀”。而大多数的J2EE应用也确实不需要在开发初期或者整个生命周期内引入任何分布式架构。这个时候，采用敏捷、轻量级的开发方案可以收到更好的效果。Spring所倡导的J2EE轻量级应用解决方案顺应天时，自然得以快速崛起……

注意　没有任何一种解决方案是普遍适用的，只有适用于特定场景的解决方案，脱离具体场景来讨论任何解决方案都是脱离实际的表现。Spring并不是要替代EJB，而是给出EJB之外的另一种方案而已，甚至于二者可以是互补的。如果某一场景下EJB依然是最为合适的解决方案，那么我们可以毫不迟疑地使用它；同样地，对于Spring的使用也应该考虑到具体的应用场景，这一点应该是需要我们始终牢记的。当古代的重装骑兵以坦克的形式重新跃上历史舞台时，Java平台上会不会也上演同样的一幕呢？我们拭目以待。

1.2　Spring 框架概述

大气层的包裹为地球创造了适合人类生存的环境。当地球上的资源足够满足人类生存需要时，我

们没必要急于开展星际移民计划，那样造成的人力物力开销会很大。当年盲目倡导EJB架构，就好像从一开始就不顾实际情况而开展星际移民计划，向其他星球移民当然是个好想法，但是否立即有必要如此大动干戈呢？对于大多数的应用程序来说，EJB架构是没有必要的。我们只需要一个大气层包裹的、环境适宜的地球即可。Spring框架所倡导的基于POJO（Plain Old Java Object，简单Java对象）的轻量级开发理念，就是从实际出发，立足于最基础的POJO（就好像我们的地球）。为了能够让这些基础的POJO构建出健壮而强大的应用，Spring框架就好像那包裹地球的大气层一样，为构筑应用的POJO提供了各种服务，进而创造了一套适宜用POJO进行轻量级开发的环境。

从广义上讲，不管Spring框架自发布到现在经过了多少次的版本更迭（从1.*x*到2.0再到2.5），其本质是始终不变的，都是为了提供各种服务，以帮助我们简化基于POJO的Java应用程序开发。Spring框架为POJO提供的各种服务共同组成了Spring的生命之树，如图1-1所示。

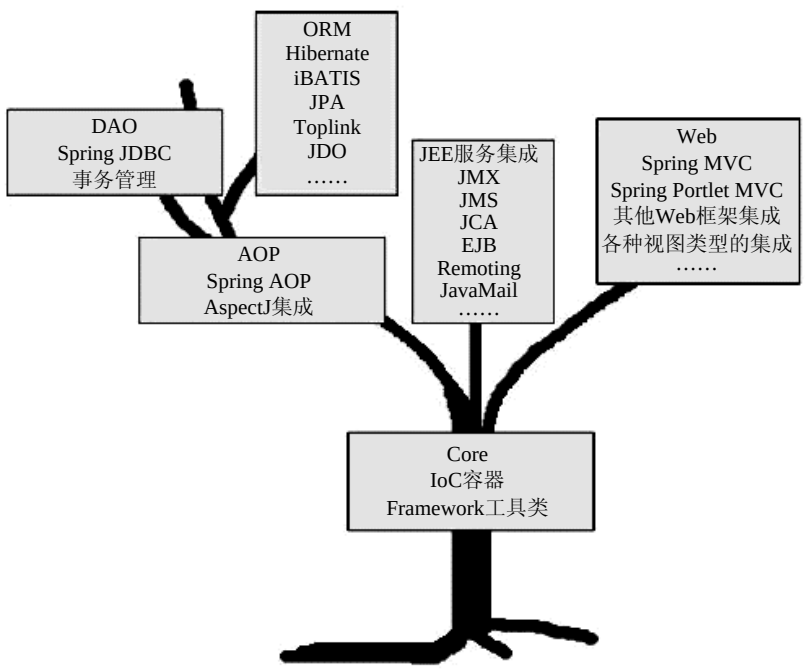


图1-1 Spring框架总体结构

组成整个Spring框架的各种服务实现被划分到了多个相互独立却又相互依赖的模块当中。正如我们在图1-1中所见到的那样，这些模块组成了Spring生命之树的枝和干，说白了也就是它们组成了Spring框架的核心骨架。抓住了这副骨架，也就抓住了Spring框架的学习主线。

整个Spring框架构建在Core核心模块之上，它是整个框架的基础。在该模块中，Spring为我们提供了一个IoC容器（IoC Container）实现，用于帮助我们以依赖注入的方式管理对象之间的依赖关系。对Spring的IoC容器的介绍将成为我们此次Spring之旅的第一站。除此之外，Core核心模块中还包括框架内部使用的各种工具类（如果愿意，我们也可以在框架之外使用），比如Spring的基础IO工具类等，这些基础工具类我们也会在合适的地方介绍。

沿着Spring生命之树往上左序遍历，我们将来到AOP模块。该模块提供了一个轻便但功能强大的AOP框架，让我们可以以AOP的形式增强各POJO的能力，进而补足OOP/OOSD之缺憾。Spring的AOP框架符合AOP Alliance规范，采用Proxy模式构建，与IoC容器相结合，可以充分显示出Spring AOP的强大威力。我们将在了解了Spring的IoC容器的基础上，详细讲述Spring AOP这一部分。

继续上行，Spring框架在Core核心模块和AOP模块的基础上，为我们提供了完备的数据访问和事务管理的抽象和集成服务。在数据访问支持方面，Spring对JDBC API的最佳实践极大地简化了该API的使用。除此之外，Spring框架为各种当前业界流行的ORM产品，比如Hibernate、iBATIS、Toplink、JPA等提供了形式统一的集成支持。Spring框架中的事务管理抽象层是Spring AOP的最佳实践，它直接构建在Spring AOP的基础之上，为我们提供了编程式事务管理和声明式事务管理的完备支持。这些服务极大地简化了日常应用开发过程中的数据访问和事务管理工作。在学习完这两部分内容之后，相信读者将会有切身的体会。

为了简化各种Java EE服务（像JNDI、JMS以及JavaMail等）的使用，Spring框架为我们提供了针对这些Java EE服务的集成服务。在Spring的帮助下，这些Java EE服务现在都变得不再烦琐难用。因为相关的Java EE服务较多，我们将会选择合适的几种介绍Spring框架给予它们的支持。随着航空航天技术的发展，我们现在可以从地球上发送飞船去访问其他星球，使用Spring框架构建的基于POJO的应用程序如果也需要远程访问或者公开一些服务的话，Spring的Remoting框架将帮助它完成这一使命。Spring的Remoting框架和Spring对其他Java EE服务的集成将分别在不同的章节中介绍。

最后要提到的就是Web模块。在该模块中，Spring框架提供了一套自己的Web MVC框架，职责分明的角色划分让这套框架看起来十分地“醒目”。我们将为Spring的Web MVC框架单独开辟一块“领地”进行讲解。在那一部分中，读者可以充分领略Web MVC框架的魅力。Spring的Portlet MVC构建在Spring Web MVC之上，延续了Spring Web MVC的一贯风格。本书不会对其做详细介绍，如果需要，可以参考文献中的有关参考书籍。Spring Web MVC并不排斥现有的其他Web框架，像Struts、WebWork以及JSF等；Spring的Web框架都为它们提供了集成支持。除此之外，像Web开发中可能牵扯的各种视图（View）技术，Spring Web框架更是给予了足够的重视。

就像一棵树必须依赖强大的根基才能生长繁盛一样，Spring框架内的各个模块也是如此。理论上来说，上层的模块需要依赖下层的模块才能正常工作，这就是为什么说这些模块是相互依赖的。不过，近乎处于同一水平线的各个模块之间却可以认为是相互独立的，彼此之间没什么瓜葛。从这个角度看，这些模块之间的相互独立一说也是成立的。

以上就是对整个Spring框架的总体介绍。在开始愉快的Spring旅程之前，我想带大家先逛一逛“Spring大观园”，这样，大家就会发现即将开始的Spring之旅更加值得期待。

注意　不要只将Spring看作是一个IoC容器，也不要只将Spring与AOP挂钩，Spring提供的远比这些东西要多得多。Spring不仅仅是一个简化Java EE开发的轻量级框架，它更应该是一个简化任何Java应用的开发框架。如果你愿意，甚至可以在Java的三个平台上（J2SE、J2EE、J2ME）应用Spring框架。即使当前的Spring框架还不支持相应平台或者相应场景的应用开发，但是只要你掌握了Spring的理念和方法，同样可以让新的“Spring”在相应的场景中发挥作用。

1.3　Spring 大观园

在1995年Java作为一门计算机语言而诞生时，有谁能够想到，短短10多年间，它已经发展成为一

个强大的开发平台？对于Spring框架来说，历史又在重演，而且几乎毫无悬念。

Spring大观园中有一棵参天大树，它得以茁壮成长，主要因为它有一个好的根基，那就是Spring框架。在Spring框架的基础上，Spring家族人丁开始兴旺，不断涌现出一个又一个引人注目的家族成员，包括：

- Spring Web Flow（SWF）[①]。Spring Web Flow构建于Spring Web MVC框架之上，旨在简化拥有复杂用户交互逻辑的Web应用程序的开发。通过Spring Web Flow的扩展支持，可以在基于Spring Web MVC的Web应用程序中以更简单的方式，创建更加复杂的业务交互流程。同时，Spring Web Flow还让Ajax和JSF享受一等公民待遇，所有这些将帮助我们更快更好地满足各种用户的实际需求。
- Spring Web Services[②]。Spring Web Services是一套采用契约优先（Contract-First）开发模式，创建文档驱动（Document-driven）Web服务的Web服务开发框架。它除了对Web服务中涉及的XML的映射关系管理提供了详尽的支持，还与Spring框架以及其他子项目（比如Spring Security）紧密结合，帮助以更加灵活高效的方式打造Web服务应用服务。
- Spring Security（原来的Acegi Security）[③]。Spring Security由原来的Acegi Security发展而来，主要为基于Spring框架的企业级应用程序提供安全解决方案。Spring Security 2.0发布后在原来Acegi Security 1.0的基础上又添加了很多富有吸引力的特性，包括简化配置、面向RESTful请求的安全认证、与Spring Web Flow和Spring Web Services等项目的良好集成等，可以说为基于Spring框架的企业级应用提供了一站式的安全方面的解决方案。
- Spring Dynamic Modules for OSGi Service Platforms[④]。Spring-DM是融合了Spring框架以及OSGi两家优良基因后的产物，它集Spring框架各种服务和OSGi的动态性、模块化等特性于一身，可以帮助我们以一种全新的方式来打造新一代的企业级应用程序。SpringSource Application Platform应用服务器就是构建在Spring-DM之上的。在企业级应用开发领域，Spring-DM或许会掀起另一个浪潮。
- Spring Batch[⑤]。当意识到企业应用中批处理业务所占的市场份额不容小觑之后，Spring Batch开始浮出水面，它是构建在Spring框架之上的一套轻量级批处理开发框架，由SpringSource和埃森哲（Accenture）合力打造。如果你还在为无法找到一款满意的开源批处理开发框架而烦恼，也许Spring Batch会让你的烦恼顷刻间烟消云散。
- Spring Integration[⑥]。Spring Integration面向创建基于Spring开发框架的企业集成（Enterprise Integration）解决方案，对*Enterprise Integration Patterns*[⑦]一书中的企业集成模式提供支持。它在现有Spring框架对企业方案集成的基础上，提出了更高层次的抽象方案，使得业务和集成逻辑得以松散耦合，很好地分离了企业集成过程中的不同关注点。
- Spring LDAP[⑧]。Spring LDAP传承了Spring框架中应用模板方法模式（Template Method Pattern）

① http://www.springframework.org/webflow。
② http://www.springframework.org/spring-ws。
③ http://static.springframework.org/spring-security/site/index.html。
④ http://www.springframework.org/osgi。
⑤ http://www.springframework.org/spring-batch。
⑥ http://www.springframework.org/spring-integration。
⑦ http://www.eaipatterns.com/。
⑧ http://www.springframework.org/ldap。

的优良传统，由最初的LdapTemplate发展演化而来，旨在简化LDAP相关操作。

- Spring IDE[①]。如果读者使用Eclipse平台开发Spring应用程序，结合Spring IDE插件将会使开发更加得心应手。Spring IDE以Eclipse开发平台为中心，想开发人员之所想，包含了各种实用的特性，为使用Eclipse创建基于Spring的应用程序，提供了灵活而强大的开发环境。
- Spring Modules[②]。为了避免Spring框架对各种其他项目的集成和支持造成Spring框架本身的臃肿等一系列问题，Spring Modules将那些可选的工具和附加类库剥离出Spring核心框架，纳入自身进行统一管理。如果在使用Spring框架开发的过程中，发现某些第三方库或工具，在核心框架中不存在的话，可以求助于Spring Modules提供的各种扩展，包括它对ANT、OSWorkflow、Apache OJB，以及低版本的iBatis等第三方库的扩展支持。
- Spring JavaConfig[③]。Spring框架提供的依赖注入支持，最初是使用XML表述依赖注入关系的。在Spring 2.5正式提供了基于注解的依赖注入方式之前，Spring JavaConfig就为Spring框架提出了一套基于注解的依赖注入解决方案，它可以看作是Spring 2.5中基于注解的依赖注入正式方案之外的另一种选择。
- Spring Rich Client[④]。与Eclipse RCP为基于SWT/JFace的GUI应用提供了一套完备的开发框架类似，Spring也为使用Swing进行GUI开发的应用提供了一套开发框架，这就是Spring Rich Client。如果你想在开发Swing应用的过程中同时获得Spring框架的各项支持的话，那Spring Rich Client正是为你而生的。
- Spring .NET[⑤]。Spring框架在Java平台上的成功是有目共睹的，这种成功同样渗透到了.NET平台，Spring .NET就是SpringSource为.NET企业开发平台量身打造的开源应用开发框架。
- Spring BeanDoc[⑥]。Spring BeanDoc可以根据Spring应用程序使用的配置文件中的相应信息，创建对应的文档和图表，帮助我们以更加直观的方式来了解Spring应用程序的整体结构。

这些家族成员全部以Apache Lisence Version 2.0协议发布，共同组成了Spring Projects组合。这一组合对软件开发中的各种需求提供从广度到深度的支持，极大地简化了日常开发工作。而且，因为它们都是以开源形式发布的，所以大部分软件公司都可以从中受益，可以投入更低的成本打造高质量的软件产品。

Spring Projects组合中的开源项目为很多软件开发人员、软件公司带来了益处，但因为是开源，所以这些项目主要靠社区来推动和发展。活跃的开发社区可以为我们带来快速的反馈和支持，但没有任何主体或个人可以保证我们所需要的反馈和支持能够及时有效地得到满足。鉴于这一点，SpringSource（原来的Interface21，即Spring框架的“东家”）在Spring Projects组合的基础上，提供了Spring Portfolio[⑦]产品，SpringSource为Spring Portfolio产品中的各成员提供咨询、培训和支持服务。Spring Portfolio产品由Spring Projects组合中多个成功的企业级开源产品，以及AspectJ等Spring组织外部的优秀开源产品共同组成：

- Spring Framework；

① http://springide.org/。

② https://springmodules.dev.Java.net/。

③ http://www.springframework.org/Javaconfig。

④ http://www.springframework.org/spring-rcp。

⑤ http://www.springframework.net/。

⑥ http://spring-beandoc.sourceforge.net/。

⑦ Portfolio指的是证券投资组合，在这里应该是多种Spring家族相关产品的组合。

- Spring Security（原来的Acegi）；
- Spring Web Flow；
- Spring Web Services；
- Spring Dynamic Modules for the OSGi Service Platform；
- Spring Batch；
- Pitchfork（http://www.springsource.com/pitchfork）；
- AspectJ（http://www.eclipse.org/aspectj/）；
- Spring IDE；
- Spring .NET；
- Spring LDAP；
- Spring Rich Client；
- Spring Integration。

实际上，Spring Portfolio只是SpringSource产品组合之一。为了能够为客户提供更多价值，也为了继续保持Spring家族在Java平台企业级开发中一贯的领先地位，SpringSource积极扩展，为我们带来了更多的企业级开发产品，像构建于Spring Dynamic Modules之上的新一代企业级Java应用服务器SpringSource Application Platform，为企业用户量身打造的SpringSource Enterprise等产品。而且，SpringSource还积极吸纳其他成功开源产品（包括Apache HTTPD[①]、Apache Tomcat[②]、Apache ActiveMQ[③]等）的主要开发人员，以便更好地为这些开源产品的客户提供及时有效的支持服务。

无论从哪一个角度看，整个Spring家族都是富有活力、积极进取的，一旦有新的开发理念或者最佳实践涌现，我们通常会第一时间在Spring家族中发现它们的身影。随着整个Spring平台的发展，我们会看到Spring大观园将愈发地花团锦簇、欣欣向荣。看到这么一幅宏大而美丽的景象，你或许早就热血沸腾，想要马上投入Spring大观园之中。不过，正像Donald J. Trump在*How To Get Rich*一书中所说的那样：“Before the dream lifts you into the clouds, make sure look hard at the facts on the ground.”[④]

要知道，所有的Spring家族成员全部构建于Spring框架基础之上，在我们想要往Spring这棵参天大树更高的地方攀爬之前，实实在在地先去了解Spring框架这一根基，才是当前的首要任务，不是吗？如果读者能够认同这些，那么现在就让我们来开始Spring框架的全景之旅吧！

1.4 小结

本章首先对Spring框架得以迅速崛起的背景做了简短介绍，然后带领读者从总体上了解了Spring框架的构成。按常规思路，在了解了Spring框架的总体结构之后，就应该深入框架内部以进一步了解它的详细内容。但是，为了让大家对Spring框架以及构建在该框架基础之上的整个平台有所认识，本章最后加入了对整个Spring开发平台的总体介绍。

① http://www.covalent.net/supportservices/apache/index.html。

② http://www.covalent.net/supportservices/tomcat/index.html。

③ http://www.covalent.net/supportservices/activemq/index.html。

④ 中文大意是：“在被梦想搞得飘飘然之前，最好先让自己脚踏实地”。——编者注

第 2 章 IoC的基本概念

2

本章内容

- 我们的理念是：让别人为你服务
- 手语，呼喊，还是心有灵犀
- IoC的附加值

2.1 我们的理念是：让别人为你服务

IoC是随着近年来轻量级容器（Lightweight Container）的兴起而逐渐被很多人提起的一个名词，它的全称为Inversion of Control，中文通常翻译为“控制反转”，它还有一个别名叫做依赖注入（Dependency Injection）。好莱坞原则“Don’t call us, we will call you.”[①]恰如其分地表达了“反转”的意味，是用来形容IoC最多的一句话。那么，为什么需要IoC？IoC的具体意义是什么？它到底有什么独到之处？让我们带着这些疑问开始我们的IoC之旅吧。

注意 本章更多的是将IoC和依赖注入看作等同的概念进行讲解。但是，在这一点上可能存在不同的观点，比如*Expert Spring MVC and Web Flow*和*Expert One-on-One J2EE without EJB*等书中都将依赖注入看作是IoC的一种方式。不过，本章暂且忽略这些观点，将IoC和依赖注入等同看待。在读者理解了依赖注入之后，可以再结合其他资料对IoC做进一步的研究。

为了更好地阐述IoC模式的概念，我们引入以下简单场景。

在我经历的FX项目[②]中，经常需要近乎实时地为客户提供外汇新闻。通常情况下，都是先从不同的新闻社订阅新闻来源，然后通过批处理程序定时地到指定的新闻服务器抓取最新的外汇新闻，接着将这些新闻存入本地数据库，最后在FX系统的前台界面显示。

假设我们有一个`FXNewsProvider`类来做以上工作，其代码如代码清单2-1所示。

代码清单2-1 `FXNewsProvider`类的实现

```
public class FXNewsProvider
{
```

① 中文大意是：“你不用找我们，我们会找你的。”——编者注

② FX全称为“Foreign Exchange”，即外汇交易。FX系统通常作为交易的中间商，与上游服务商（比如花旗和莱曼等各大银行）合作，为顾客提供保证金交易服务，顾客只需要交纳很少的保证金就可以进行大额的外汇交易。保证金交易的杠杆可以让顾客“以小博大”， 只要很少的资金，顾客就可以享受到汇率变动所带来的收益（当然，评估方向错误也可能招致损失）。

```
    private IFXNewsListener  newsListener;
    private IFXNewsPersister newPersistener;
    public void getAndPersistNews()
    {
      String[] newsIds = newsListener.getAvailableNewsIds();
      if(ArrayUtils.isEmpty(newsIds))
      {
        return;
      }

      for(String newsId : newsIds)
      {
        FXNewsBean newsBean = newsListener.getNewsByPK(newsId);
        newPersistener.persistNews(newsBean);
        newsListener.postProcessIfNecessary(newsId);
      }
    }
}
```

其中，FXNewsProvider需要依赖IFXNewsListener来帮助抓取新闻内容，并依赖IFXNewsPersister存储抓取的新闻。

假设默认使用道琼斯（Dow Jones）新闻社的新闻，那么我们相应地提供了DowJonesNewsListener和DowJonesNewsPersister两个实现。通常情况下，需要在构造函数中构造IFXNewsProvider依赖的这两个类（以下将这种被其他类依赖的类或对象，简称为“依赖类”、“依赖对象”），如代码清单2-2所示。

代码清单2-2 构造IFXNewsProvider类的依赖类

```
public FXNewsProvider()
{
  newsListener   = new DowJonesNewsListener();
  newPersistener = new DowJonesNewsPersister();
}
```

看，这就是我们通常的做事方式！如果我们依赖于某个类或服务，最简单而有效的方式就是直接在类的构造函数中新建相应的依赖类。这就好比要装修新房，需要用家具，这个时候，根据通常解决对象依赖关系的做法，我们就会直接打造出需要的家具来。不过，通常都是分工明确的，所以，大多数情况下，我们可以去家具广场将家具买回来，然后根据需要装修布置即可。

不管是直接打造家具（通过new构造对象），还是去家具广场买家具（或许是通过Service Locator[①]解决直接的依赖耦合），有一个共同点需要我们关注，那就是，我们都是自己主动地去获取依赖的对象！

可是回头想想，我们自己每次用到什么依赖对象都要主动地去获取，这是否真的必要？我们最终所要做的，其实就是直接调用依赖对象所提供的某项服务而已。只要用到这个依赖对象的时候，它能够准备就绪，我们完全可以不管这个对象是自己找来的还是别人送过来的。对于FXNewsProvider来说，那就是在getAndPersistNews()方法调用newsListener的相应方法时，newsListener能够准备就绪就可以了。如果有人能够在我们需要时将某个依赖对象送过来，为什么还要大费周折地自己去折腾？

实际上，IoC就是为了帮助我们避免之前的“大费周折”，而提供了更加轻松简洁的方式。它的反转，就反转在让你从原来的事必躬亲，转变为现在的享受服务。你想啊，原来还得鞍马劳顿，什么

① J2EE核心模式的一种，主要通过引入中间代理者消除对象间复杂的耦合关系，并统一管理分散的复杂耦合关系。

东西都得自己去拿。现在是用什么，让别人直接送过来就成。所以，简单点儿说，IoC的理念就是，让别人为你服务！在图2-1中，也就是让IoC Service Provider来为你服务！

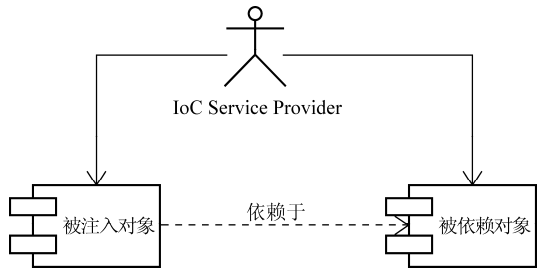


图2-1　IoC的角色

通常情况下，被注入对象会直接依赖于被依赖对象。但是，在IoC的场景中，二者之间通过IoC Service Provider来打交道，所有的被注入对象和依赖对象现在由IoC Service Provider统一管理。被注入对象需要什么，直接跟IoC Service Provider招呼一声，后者就会把相应的被依赖对象注入到被注入对象中，从而达到IoC Service Provider为被注入对象服务的目的。IoC Service Provider在这里就是通常的IoC容器所充当的角色。从被注入对象的角度看，与之前直接寻求依赖对象相比，依赖对象的取得方式发生了反转，控制也从被注入对象转到了IoC Service Provider那里[①]。

其实IoC就这么简单！原来是需要什么东西自己去拿，现在是需要什么东西就让别人送过来。图2-2以两种场景，形象地说明了使用IoC模式前后的差别。

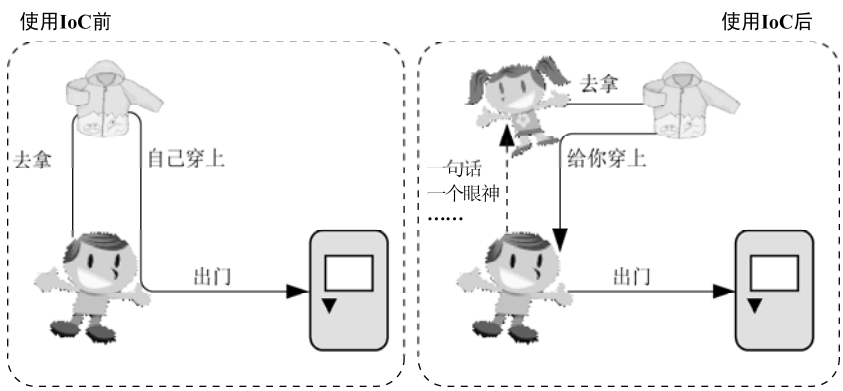


图2-2　使用IoC前后的差别

出门之前得先穿件外套吧？以前，你得自己跑到衣柜前面取出衣服这一依赖对象，然后自己穿上再出门。而现在，你只要跟你的“另一半”使个眼色或说一句“Honey，衣服拿来。”她就会心领神会地到衣柜那里为你取出衣服，然后再给你穿上。现在，你就可以出门了。（此时此刻，你心里肯定窃喜，“有人照顾的感觉真好！”）对你来说，到底哪种场景比较惬意，我想已经不言自明了吧？

① 与之前其他书籍和文章讲解IoC的概念方式不同，本书这里不是从对象解耦的角度来阐述的。为了能让读者将IoC与原来的对象绑定模式做一个对比，我们决定从对象绑定方式的角度来阐述IoC的概念，这样对比可以更加鲜明地表现新概念与老概念的差别。

2.2 手语，呼喊，还是心有灵犀

“伙计，来杯啤酒！”当你来到酒吧，想要喝杯啤酒的时候，通常会直接招呼服务生，让他为你送来一杯清凉解渴的啤酒。同样地，作为被注入对象，要想让IoC Service Provider为其提供服务，并将所需要的被依赖对象送过来，也需要通过某种方式通知对方。

- 如果你是酒吧的常客，或许你刚坐好，服务生已经将你最常喝的啤酒放到了你面前；
- 如果你是初次或偶尔光顾，也许你坐下之后还要招呼服务生，“Waiter,Tsingdao, please.”；
- 还有一种可能，你根本就不知道哪个牌子是哪个牌子，这时，你只能打手势或干脆画出商标图来告诉服务生你到底想要什么了吧！

不管怎样，你终究会找到一种方式来向服务生表达你的需求，以便他为你提供适当的服务。那么，在IoC模式中，被注入对象又是通过哪些方式来通知IoC Service Provider为其提供适当服务的呢？

IoC模式最权威的总结和解释，应该是Martin Fowler的那篇文章“Inversion of Control Containers and the Dependency Injection pattern”，其中提到了三种依赖注入的方式，即构造方法注入（constructor injection）、setter方法注入（setter injection）以及接口注入（interface injection）。下面让我们详细看一下这三种方式的特点及其相互之间的差别。

2.2.1 构造方法注入

顾名思义，构造方法注入，就是被注入对象可以通过在其构造方法中声明依赖对象的参数列表，让外部（通常是IoC容器）知道它需要哪些依赖对象。对于前面例子中的FXNewsProvider来说，只要声明如下构造方法（见代码清单2-3）即可支持构造方法注入。

代码清单2-3 FXNewsProvider构造方法定义

```
public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister)
{
   this.newsListener   = newsListner;
   this.newPersistener = newsPersister;
}
```

IoC Service Provider会检查被注入对象的构造方法，取得它所需要的依赖对象列表，进而为其注入相应的对象。同一个对象是不可能被构造两次的，因此，被注入对象的构造乃至其整个生命周期，应该是由IoC Service Provider来管理的。

构造方法注入方式比较直观，对象被构造完成后，即进入就绪状态，可以马上使用。这就好比你刚进酒吧的门，服务生已经将你喜欢的啤酒摆上了桌面一样。坐下就可马上享受一份清凉与惬意。

2.2.2 setter 方法注入

对于JavaBean对象来说，通常会通过setXXX()和getXXX()方法来访问对应属性。这些setXXX()方法统称为setter方法，getXXX()当然就称为getter方法。通过setter方法，可以更改相应的对象属性，通过getter方法，可以获得相应属性的状态。所以，当前对象只要为其依赖对象所对应的属性添加setter方法，就可以通过setter方法将相应的依赖对象设置到被注入对象中。以FXNewsProvider为例，添加setter方法后如代码清单2-4所示。

代码清单2-4 添加了setter方法声明的FXNewsProvider

```
public class FXNewsProvider
```

```
{
   private IFXNewsListener  newsListener;
   private IFXNewsPersister newPersistener;

   public IFXNewsListener getNewsListener() {
      return newsListener;
   }
   public void setNewsListener(IFXNewsListener newsListener) {
      this.newsListener = newsListener;
   }
   public IFXNewsPersister getNewPersistener() {
      return newPersistener;
   }
   public void setNewPersistener(IFXNewsPersister newPersistener) {
      this.newPersistener = newPersistener;
   }
}
```

这样，外界就可以通过调用setNewsListener和setNewPersistener方法为FXNewsProvider对象注入所依赖的对象了。

setter方法注入虽不像构造方法注入那样，让对象构造完成后即可使用，但相对来说更宽松一些，可以在对象构造完成后再注入。这就好比你可以到酒吧坐下后再决定要点什么啤酒，可以要百威，也可以要大雪，随意性比较强。如果你不急着喝，这种方式当然是最适合你的。

2.2.3　接口注入

相对于前两种注入方式来说，接口注入没有那么简单明了。被注入对象如果想要IoC Service Provider为其注入依赖对象，就必须实现某个接口。这个接口提供一个方法，用来为其注入依赖对象。IoC Service Provider最终通过这些接口来了解应该为被注入对象注入什么依赖对象。图2-3演示了如何使用接口注入为FXNewsProvider注入依赖对象。

FXNewsProvider为了让IoC Service Provider为其注入所依赖的IFXNewsListener，首先需要实现IFXNewsListenerCallable接口，这个接口会声明一个injectNewsListner方法（方法名随意），该方法的参数，就是所依赖对象的类型。这样，InjectionServiceContainer对象，即对应的IoC Service Provider就可以通过这个接口方法将依赖对象注入到被注入对象FXNewsProvider当中。

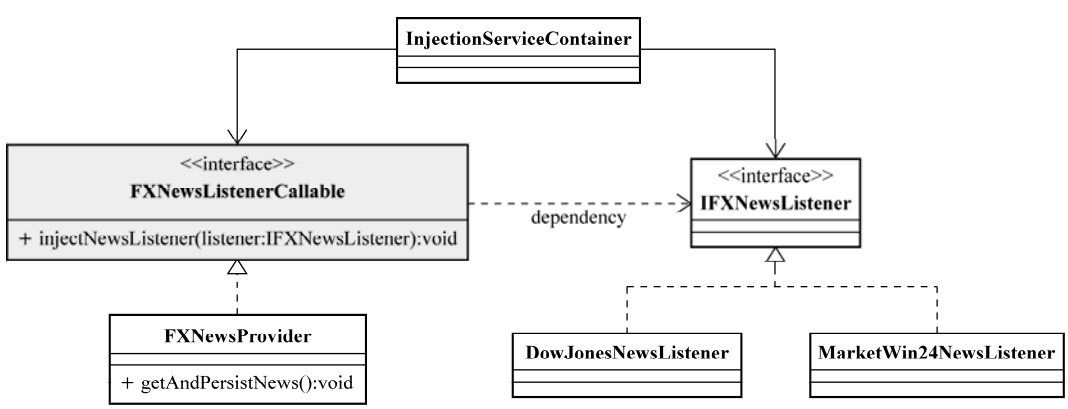


图2-3　使用接口注入的FXNewsProvider

小心 在这种情况下，实现的接口和接口中声明的方法名称都不重要。重要的是接口中声明方法的参数类型，必须是“被注入对象”所依赖对象的类型。

接口注入方式最早并且使用最多的是在一个叫做Avalon的项目中，相对于前两种依赖注入方式，接口注入比较死板和烦琐。如果需要注入依赖对象，被注入对象就必须声明和实现另外的接口。这就好像你同样在酒吧点啤酒，为了让服务生理解你的意思，你就必须戴上一顶啤酒杯式的帽子（如图2-4所示），看起来有点多此一举。

图2-4 只想要一杯啤酒，需要这样嘛

通常情况下，这有些让人不好接受。不过，好在这种方式也可以达到目的。

2.2.4 三种注入方式的比较

- **接口注入**。从注入方式的使用上来说，接口注入是现在不甚提倡的一种方式，基本处于“退役状态”。因为它强制被注入对象实现不必要的接口，带有侵入性。而构造方法注入和setter方法注入则不需要如此。
- **构造方法注入**。这种注入方式的优点就是，对象在构造完成之后，即已进入就绪状态，可以马上使用。缺点就是，当依赖对象比较多的时候，构造方法的参数列表会比较长。而通过反射构造对象的时候，对相同类型的参数的处理会比较困难，维护和使用上也比较麻烦。而且在Java中，构造方法无法被继承，无法设置默认值。对于非必须的依赖处理，可能需要引入多个构造方法，而参数数量的变动可能造成维护上的不便。
- **setter方法注入**。因为方法可以命名，所以setter方法注入在描述性上要比构造方法注入好一些。另外，setter方法可以被继承，允许设置默认值，而且有良好的IDE支持。缺点当然就是对象无法在构造完成后马上进入就绪状态。

综上所述，构造方法注入和setter方法注入因为其侵入性较弱，且易于理解和使用，所以是现在使用最多的注入方式；而接口注入因为侵入性较强，近年来已经不流行了。

2.3 IoC 的附加值

从主动获取依赖关系的方式转向IoC方式，不只是一个方向上的改变，简单的转变背后实际上蕴藏着更多的玄机。要说IoC模式能带给我们什么好处，可能各种资料或书籍中已经罗列很多了。比如不会对业务对象构成很强的侵入性，使用IoC后，对象具有更好的可测试性、可重用性和可扩展性，等等。不过，泛泛而谈可能无法真正地让你深刻理解IoC模式带来的诸多好处，所以，还是让我们从具体的示例入手，来一探究竟吧。

对于前面例子中的`FXNewsProvider`来说，在使用IoC重构之前，如果没有其他需求或变动，不光看起来，用起来也是没有问题的。但是，当系统中需要追加逻辑以处理另一家新闻社的新闻来源时，问题就来了。

突然有一天，客户告诉你，我们又搞定一家新闻社，现在可以使用他们的新闻服务了，这家新闻社叫MarketWin24。这个时候，你该如何处理呢？首先，毫无疑问地，应该先根据MarketWin24的服务接口提供一个MarketWin24NewsListener实现，用来接收新闻；其次，因为都是相同的数据访问逻辑，所以原来的DowJonesNewsPersister可以重用，我们先放在一边不管。最后，就主要是业务处理对象FXNewsProvider了。因为我们之前没有用IoC，所以，现在的对象跟DowJonesNewsListener是绑定的，我们无法重用这个类了，不是吗？为了解决问题，我们可能要重新实现一个继承自FXNewsProvider的MarketWin24NewsProvider，或者干脆重新写一个类似的功能。

而使用IoC后，面对同样的需求，我们却完全可以不做任何改动，就直接使用FXNewsProvider。因为不管是DowJones还是MarketWin24，对于我们的系统来说，处理逻辑实际上应该是一样的：根据各个公司的连接接口取得新闻，然后将取得的新闻存入数据库。因此，我们只要根据MarketWin24的新闻服务接口，为MarketWin24的FXNewsProvider提供相应的MarketWin24NewsListener注入就可以了，见代码清单2-5。

代码清单2-5　构建在IoC之上可重用的FXNewsProvider使用演示

```
FXNewsProvider dowJonesNewsProvider = ➥
new FXNewsProvider(new DowJonesNewsListener(),new DowJonesNewsPersister());
...
FXNewsPrivider marketWin24NewsProvider = ➥
new FXNewsProvider(new MarketWin24NewsListener(),new DowJonesNewsPersister());
...
```

看！使用IoC之后，FXNewsProvider可以重用，而不必因为添加新闻来源去重新实现新的FXNewsProvider。实际上，只需要给出特定的IFXNewsListener实现即可。

随着开源项目的成功，TDD（Test Driven Developement，测试驱动开发）已经成为越来越受重视的一种开发方式。因为保证业务对象拥有良好的可测试性，可以为最终交付高质量的软件奠定良好的基础，同时也拉起了产品质量的第一道安全网。所以对于软件开发来说，设计开发可测试性良好的业务对象是至关重要的。而IoC模式可以让我们更容易达到这个目的。比如，使用IoC模式后，为了测试FXNewsProvider，我们可以根据测试的需求，提供一个MockNewsListener给FXNewsProvider。在此之前，我们无法将对DowJonesNewsListener的依赖排除在外，从而导致难以开展单元测试。而现在，单元测试则可以毫无牵绊地进行，代码清单2-6演示了测试取得新闻失败的情形。

代码清单2-6　测试FXNewsProvider类的相关定义

```
测试新闻取得失败的MockNewsListner定义
public class MockNewsListener implements IFXNewsListener
{
   public String[] getAvailableNewsIds() {
      throw new FXNewsRetrieveFailureException();
   }
   public FXNewsBean getNewsByPK(String newsId) {
      // TODO
      return null;
   }
   public void postProcessIfNecessary(String newsId) {
      // TODO
   }
}
相应的FXNewsProvider的单元测试类
```

```
public class FXNewsProviderTest extends TestCase {
  private FXNewsProvider newsProvider;

  @Override
  protected void setUp() throws Exception {
    super.setUp();
    newsProvider = new FXNewsProvider(new MockNewsListener(),new MockNewsPersister());
  }
  @Override
  protected void tearDown() throws Exception {
    super.tearDown();
    newsProvider = null;
  }
  public void testGetAndPersistNewsWithoutResourceAvailable()
  {
    try
    {
      newsProvider.getAndPersistNews();
      fail("Since MockNewsListener has no news support, ➥
      we should fail to get above.");
    }
    catch(FXNewsRetrieveFailureException e)
    {
      //……
    }
  }
}
```

由此可见，相关资料或书籍提到IoC总会赞不绝口，并不是没有原因的。如果你还心存疑虑，那么自己去验证一下吧！说不定你还可以收获更多。毕竟，实践出真知嘛。

如果要用一句话来概括IoC可以带给我们什么，那么我希望是，IoC是一种可以帮助我们解耦各业务对象间依赖关系的对象绑定方式！

2.4 小结

本章主要介绍了IoC或者说依赖注入的概念，讨论了几种基本的依赖注入方式。还与大家一起探索并验证了IoC所带给我们的部分“附加值”。所以，现在大家应该对IoC或者说依赖注入有了最基本认识。下一章，我们将一起去更深入地了解IoC场景中的重要角色，即IoC Service Provider。

第 3 章

3 掌管大局的IoC Service Provider

本章内容

- IoC Service Provider的职责
- 运筹帷幄的秘密——IoC Service Provider如何管理对象间的依赖关系

虽然业务对象可以通过IoC方式声明相应的依赖，但是最终仍然需要通过某种角色或者服务将这些相互依赖的对象绑定到一起，而IoC Service Provider就对应IoC场景中的这一角色。

IoC Service Provider在这里是一个抽象出来的概念，它可以指代任何将IoC场景中的业务对象绑定到一起的实现方式。它可以是一段代码，也可以是一组相关的类，甚至可以是比较通用的IoC框架或者IoC容器实现。比如，可以通过以下代码（见代码清单3-1）绑定与新闻相关的对象。

代码清单3-1 FXNewsProvider相关依赖绑定代码

```
IFXNewsListener newsListener = new DowJonesNewsListener();
IFXNewsPersister newsPersister = new DowJonesNewsPersister();
FXNewsProvider newsProvider = new FXNewsProvider(newsListener,newsPersister);
newsProvider.getAndPersistNews();
```

这段代码就可以认为是这个场景中的IoC Service Provider，只不过比较简单，而且目的也过于单一罢了。要将系统中几十、几百甚至数以千计的业务对象绑定到一起，采用这种方式显然是不切实际的。通用性暂且不提，单单是写这些绑定代码也会是一种很糟糕的体验。不过，好在现在许多开源产品通过各种方式为我们做了这部分工作。所以，目前来看，我们只需要使用这些产品提供的服务就可以了。Spring的IoC容器就是一个提供依赖注入服务的IoC Service Provider。

3.1 IoC Service Provider 的职责

IoC Service Provider的职责相对来说比较简单，主要有两个：业务对象的构建管理和业务对象间的依赖绑定。

- **业务对象的构建管理**。在IoC场景中，业务对象无需关心所依赖的对象如何构建如何取得，但这部分工作始终需要有人来做。所以，IoC Service Provider需要将对象的构建逻辑从客户端对象[①]那里剥离出来，以免这部分逻辑污染业务对象的实现。
- **业务对象间的依赖绑定**。对于IoC Service Provider来说，这个职责是最艰巨也是最重要的，这是它的最终使命之所在。如果不能完成这个职责，那么，无论业务对象如何的“呼喊”，也不会得到依赖对象的任何响应（最常见的倒是会收到一个NullPointerException）。IoC Service Provider通过结合之前构建和管理的所有业务对象，以及各个业务对象间可以识别的依赖关系，

① 这里指代使用某个对象或者某种服务的对象。如果对象A需要引用对象B，那么A就是B的客户端对象，而不管A处于Service层还是数据访问层。

将这些对象所依赖的对象注入绑定，从而保证每个业务对象在使用的时候，可以处于就绪状态。

3.2 运筹帷幄的秘密——IoC Service Provider 如何管理对象间的依赖关系

前面我们说过，被注入对象可以通过多种方式通知IoC Service Provider为其注入相应依赖。但问题在于，收到通知的IoC Service Provider是否就一定能够完全领会被注入对象的意图，并及时有效地为其提供想要的依赖呢？有些时候，事情可能并非像我们所想象的那样理所当然。

还是拿酒吧的例子说事儿，不管是常客还是初次光顾，你都可以点自己需要的饮料，以任何方式通知服务生都可以。要是侍者经验老道，你需要的任何饮品他都知道如何为你调制并提供给你。可是，如果服务生刚入行又会如何呢？当他连啤酒、鸡尾酒都分不清的时候，你能指望他及时地将你需要的饮品端上来吗？

服务生最终必须知道顾客点的饮品与库存饮品的对应关系，才能为顾客端上适当的饮品。对于为被注入对象提供依赖注入的IoC Service Provider来说，它也同样需要知道自己所管理和掌握的被注入对象和依赖对象之间的对应关系。

IoC Service Provider不是人类，也就不能像酒吧服务生那样通过大脑来记忆和存储所有的相关信息。所以，它需要寻求其他方式来记录诸多对象之间的对应关系。比如：

- 它可以通过最基本的文本文件来记录被注入对象和其依赖对象之间的对应关系；
- 它也可以通过描述性较强的XML文件格式来记录对应信息；
- 它还可以通过编写代码的方式来注册这些对应信息；
- 甚至，如果愿意，它也可以通过语音方式来记录对象间的依赖注入关系（“嗨，它要一个这种类型的对象，拿这个给它”）。

那么，实际情况下，各种具体的IoC Service Provider实现又是通过哪些方式来记录“服务信息”的呢？

我们可以归纳一下，当前流行的IoC Service Provider产品使用的注册对象管理信息的方式主要有以下几种。

3.2.1 直接编码方式

当前大部分的IoC容器都应该支持直接编码方式，比如PicoContainer[①]、Spring、Avalon等。在容器启动之前，我们就可以通过程序编码的方式将被注入对象和依赖对象注册到容器中，并明确它们相互之间的依赖注入关系。代码清单3-2中的伪代码演示了这样一个过程。

代码清单3-2 直接编码方式管理对象间的依赖注入关系

```
IoContainer container = ...;
container.register(FXNewsProvider.class,new FXNewsProvider());
container.register(IFXNewsListener.class,new DowJonesNewsListener());
...
FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.class);
newProvider.getAndPersistNews();
```

通过为相应的类指定对应的具体实例，可以告知IoC容器，当我们要这种类型的对象实例时，请将容器中注册的、对应的那个具体实例返回给我们。

① www.picocontainer.org。

如果是接口注入，可能伪代码看起来要多一些。不过，道理上是一样的，只不过除了注册相应对象，还要将“注入标志接口”与相应的依赖对象绑定一下，才能让容器最终知道是一个什么样的对应关系，如代码清单3-3所演示的那样。

代码清单3-3　直接编码形式管理基于接口注入的依赖注入关系

```
IoContainer container = ...;
container.register(FXNewsProvider.class,new FXNewsProvider());
container.register(IFXNewsListener.class,new DowJonesNewsListener());
...
container.bind(IFXNewsListenerCallable.class, container.get(IFXNewsListener.class));
...
FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.class);
newProvider.getAndPersistNews();
```

通过bind方法将“被注入对象”（由IFXNewsListenerCallable接口添加标志）所依赖的对象，绑定为容器中注册过的IFXNewsListener类型的对象实例。容器在返回FXNewsProvider对象实例之前，会根据这个绑定信息，将IFXNewsListener注册到容器中的对象实例注入到“被注入对象”——FXNewsProvider中，并最终返回已经组装完毕的FXNewsProvider对象。

所以，通过程序编码让最终的IoC Service Provider（也就是各个IoC框架或者容器实现）得以知晓服务的“奥义”，应该是管理依赖绑定关系的最基本方式。

3.2.2　配置文件方式

这是一种较为普遍的依赖注入关系管理方式。像普通文本文件、properties文件、XML文件等，都可以成为管理依赖注入关系的载体。不过，最为常见的，还是通过XML文件来管理对象注册和对象间依赖关系，比如Spring IoC容器和在PicoContainer基础上扩展的NanoContainer，都是采用XML文件来管理和保存依赖注入信息的。对于我们例子中的FXNewsProvider来说，也可以通过Spring配置文件的方式（见代码清单3-4）来配置和管理各个对象间的依赖关系。

代码清单3-4　通过Spring的配置方式来管理FXNewsProvider的依赖注入关系

```
<bean id="newsProvider" class="..FXNewsProvider">
   <property name="newsListener">
      <ref bean="djNewsListener"/>
   </property>
   <property name="newPersistener">
      <ref bean="djNewsPersister"/>
   </property>
</bean>

<bean id="djNewsListener"
   class="..impl.DowJonesNewsListener">
</bean>
<bean id="djNewsPersister"
   class="..impl.DowJonesNewsPersister">
</bean>
```

最后，我们就可以像代码清单3-5所示的那样，通过“newsProvider”这个名字，从容器中取得已经组装好的FXNewsProvider并直接使用。

代码清单3-5　从读取配置文件完成对象组装的容器中获取FXNewsProvider并使用

```
...
container.readConfigurationFiles(...);
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("newsProvider");
newsProvider.getAndPersistNews();
```

3.2.3 元数据方式

这种方式的代表实现是Google Guice，这是Bob Lee在Java 5的注解和Generic的基础上开发的一套IoC框架。我们可以直接在类中使用元数据信息来标注各个对象之间的依赖关系，然后由Guice框架根据这些注解所提供的信息将这些对象组装后，交给客户端对象使用。代码清单3-6演示了使用Guice的相应注解标注后的FXNewsProvider定义。

代码清单3-6 使用Guice的注解标注依赖关系后的FXNewsProvider定义

```
public class FXNewsProvider
{
   private IFXNewsListener  newsListener;
   private IFXNewsPersister newPersistener;
   @Inject
   public FXNewsProvider(IFXNewsListener listener,IFXNewsPersister persister)
   {
      this.newsListener   = listener;
      this.newPersistener = persister;
   }
   ...
}
```

通过@Inject，我们指明需要IoC Service Provider通过构造方法注入方式，为FXNewsProvider注入其所依赖的对象。至于余下的依赖相关信息，在Guice中是由相应的Module来提供的，代码清单3-7给出了FXNewsProvider所使用的Module实现。

代码清单3-7 FXNewsProvider所使用的Module实现

```
public class NewsBindingModule extends AbstractModule
{
   @Override
   protected void configure() {
      bind(IFXNewsListener.class) ➥
      .to(DowJonesNewsListener.class).in(Scopes.SINGLETON);
      bind(IFXNewsPersister.class) ➥
      .to(DowJonesNewsPersister.class).in(Scopes.SINGLETON);
   }
}
```

通过Module指定进一步的依赖注入相关信息之后，我们就可以直接从Guice那里取得最终已经注入完毕，并直接可用的对象了（见代码清单3-8）。

代码清单3-8 从Guice获取并使用最终绑定完成的FXNewsProvider

```
Injector injector = Guice.createInjector(new NewsBindingModule());
FXNewsProvider newsProvider = injector.getInstance(FXNewsProvider.class);
newsProvider.getAndPersistNews();
```

当然，注解最终也要通过代码处理来确定最终的注入关系，从这点儿来说，注解方式可以算作编码方式的一种特殊情况。

3.3 小结

本章就IoC场景中的主要角色IoC Service Provider给出了言简意赅的介绍。讨论了IoC Service Provider的基本职责，以及它常用的几种依赖关系管理方式。

应该说，IoC Service Provider只是为了简化概念而提出的一个一般性的概念。下一章，我们将由一般到特殊，一起深入了解一个特定的IoC Service Provider实现产品，即Spring提供的IoC容器。

第4章 Spring的IoC容器之BeanFactory

本章内容

- 拥有BeanFactory之后的生活
- BeanFactory的对象注册与依赖绑定方式
- BeanFactory的XML之旅
- 容器背后的秘密

我们前面说过，Spring的IoC容器是一个IoC Service Provider，但是，这只是它被冠以IoC之名的部分原因，我们不能忽略的是“容器”。Spring的IoC容器是一个提供IoC支持的轻量级容器，除了基本的IoC支持，它作为轻量级容器还提供了IoC之外的支持。如在Spring的IoC容器之上，Spring还提供了相应的AOP框架支持、企业级服务集成等服务。Spring的IoC容器和IoC Service Provider所提供的服务之间存在一定的交集，二者的关系如图4-1所示。

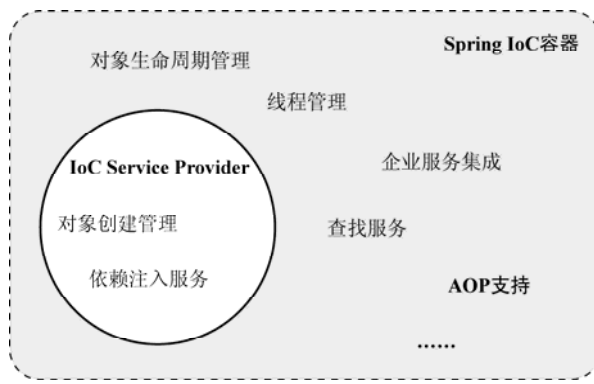


图4-1　Spring的IoC容器和IoC Service Provider之间的关系

注意　本章将主要关注Spring的IoC容器提供的IoC相关支持以及衍生的部分高级特性。而IoC容器提供的其他服务将在后继章节中陆续阐述。

Spring提供了两种容器类型：BeanFactory和ApplicationContext。

- BeanFactory。基础类型IoC容器，提供完整的IoC服务支持。如果没有特殊指定，默认采用延迟初始化策略（lazy-load）。只有当 关，才对该受管对象进行初始化以及依赖注入操作。所以，相对来说，容器启动初期速度较快，所需要的资源有限。对于资源有限，并且功能要求不是很严格的场景，BeanFactory是比较合适的IoC容器选择。
- ApplicationContext。ApplicationContext在BeanFactory的基础上构建，是相对比较高级的容器实现，除了拥有BeanFactory的所有支持，ApplicationContext还提供了其他高级

特性，比如事件发布、国际化信息支持等，这些会在后面详述。ApplicationContext所管理的对象，在该类型容器启动之后，默认全部初始化并绑定完成。所以，相对于BeanFactory来说，ApplicationContext要求更多的系统资源，同时，因为在启动时就完成所有初始化，容器启动时间较之BeanFactory也会长一些。在那些系统资源充足，并且要求更多功能的场景中，ApplicationContext类型的容器是比较合适的选择。

通过图4-2，我们可以对BeanFactory和ApplicationContext之间的关系有一个更清晰的认识。

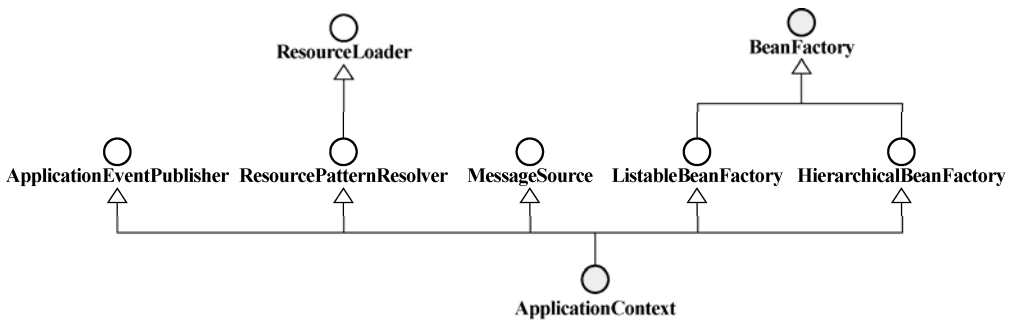


图4-2 BeanFactory和ApplicationContext继承关系

注意 ApplicationContext间接继承自BeanFactory，所以说它是构建于BeanFactory之上的IoC容器。此外，你应该注意到了，ApplicationContext还继承了其他三个接口，它们之间的关系，我们将在第5章中详细说明。

另外，在没有特殊指明的情况下，以BeanFactory为中心所讲述的内容同样适用于ApplicationContext，这一点需要明确一下，二者有差别的地方会在合适的位置给出解释。

BeanFactory，顾名思义，就是生产Bean的工厂。当然，严格来说，这个“生产过程”可能不像说起来那么简单。既然Spring框架提倡使用POJO，那么把每个业务对象看作一个JavaBean对象，或许更容易理解为什么Spring的IoC基本容器会起这么一个名字。作为Spring提供的基本的IoC容器，BeanFactory可以完成作为IoC Service Provider的所有职责，包括业务对象的注册和对象间依赖关系的绑定。

BeanFactory就像一个汽车生产厂。你从其他汽车零件厂商或者自己的零件生产部门取得汽车零件送入这个汽车生产厂，最后，只需要从生产线的终点取得成品汽车就可以了。相似地，将应用所需的所有业务对象交给BeanFactory之后，剩下要做的，就是直接从BeanFactory取得最终组装完成并且可用的对象。至于这个最终业务对象如何组装，你不需要关心，BeanFactory会帮你搞定。

所以，对于客户端来说，与BeanFactory打交道其实很简单。最基本地，BeanFactory肯定会公开一个取得组装完成的对象的方法接口，就像代码清单4-1中真正的BeanFactory的定义所展示的那样。

代码清单4-1 BeanFactory的定义

```
public interface BeanFactory {
   String FACTORY_BEAN_PREFIX = "&";
   Object getBean(String name) throws BeansException;
```

```
    Object getBean(String name, Class requiredType) throws BeansException;
    /**
     * @since 2.5
     */
    Object getBean(String name, Object[] args) throws BeansException;
    boolean containsBean(String name);
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
    /**
     * @since 2.0.3
     */
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
    /**
     * @since 2.0.1
     */
    boolean isTypeMatch(String name, Class targetType) throws NoSuchBeanDefinitionException;
    Class getType(String name) throws NoSuchBeanDefinitionException;
    String[] getAliases(String name);
}
```

上面代码中的方法基本上都是查询相关的方法，例如，取得某个对象的方法（getBean）、查询某个对象是否存在于容器中的方法（containsBean），或者取得某个bean的状态或者类型的方法等。因为通常情况下，对于独立的应用程序，只有主入口类才会跟容器的API直接耦合。

4.1　拥有BeanFactory之后的生活

确切地说，拥有BeanFactory之后的生活没有太大的变化。当然，我的意思是看起来没有太大的变化。到底引入BeanFactory后的生活是什么样子，让我们一起来体验一下吧！

依然“拉拉扯扯的事情”。对于应用程序的开发来说，不管是否引入BeanFactory之类的轻量级容器，应用的设计和开发流程实际上没有太大改变。换句话说，针对系统和业务逻辑，该如何设计和实现当前系统不受是否引入轻量级容器的影响。对于我们的FX新闻系统，我们还是会针对系统需求，分别设计相应的接口和实现类。前后唯一的不同，就是对象之间依赖关系的解决方式改变了。这就是所谓的“拉拉扯扯的事情”。之前我们的系统业务对象需要自己去“拉”（Pull）所依赖的业务对象，有了BeanFactory之类的IoC容器之后，需要依赖什么让BeanFactory为我们推过来（Push）就行了。所以，简单点儿说，拥有BeanFactory之后，要使用IoC模式进行系统业务对象的开发。（实际上，即使不使用BeanFactory之类的轻量级容器支持开发，开发中也应该尽量使用IoC模式。）代码清单4-2演示了FX新闻系统初期的设计和实现框架代码。

代码清单4-2　FX新闻应用设计和实现框架代码

```
1-设计FXNewsProvider类用于普遍的新闻处理
public class FXNewsProvider
{
   ...
}

2-设计IFXNewsListener接口抽象各个新闻社不同的新闻获取方式，并给出相应实现类
public interface IFXNewsListener
{
   ...
}
以及
public class DowJonesNewsListener implements IFXNewsListener
```

```
{
   ...
}

3-设计IFXNewsPersister接口抽象不同数据访问方式，并实现相应的实现类
public interface IFXNewsPersister
{
   ...
}
以及
public class DowJonesNewsPersister implements IFXNewsPersister
{
   ...
}
```

BeanFactory会说，这些让我来干吧。既然使用IoC模式开发的业务对象现在不用自己操心如何解决相互之间的依赖关系，那么肯定得找人来做这个工作。毕竟，工作最终是要有人来做的，大家都不动手，那工作就不能进行了。当BeanFactory说这些事情让它来做的时候，可能没有告诉你它会怎么来做这个事情。不过没关系，稍后我会详细告诉你它是如何做的。通常情况下，它会通过常用的XML文件来注册并管理各个业务对象之间的依赖关系，就像代码清单4-3所演示的那样。

代码清单4-3 使用BeanFactory的XML配置方式实现业务对象间的依赖管理

```
<beans>
   <bean id="djNewsProvider" class="..FXNewsProvider">
      <constructor-arg index="0">
         <ref bean="djNewsListener"/>
      </constructor-arg>
      <constructor-arg index="1">
         <ref bean="djNewsPersister"/>
      </constructor-arg>
   </bean>
   ...
</beans>
```

拉响启航的汽笛。在BeanFactory出现之前，我们通常会直接在应用程序的入口类的main方法中，自己实例化相应的对象并调用之，如以下代码所示：

```
FXNewsProvider newsProvider = new FXNewsProvider();
newsProvider.getAndPersistNews();
```

不过，现在既然有了BeanFactory，我们通常只需将“生产线图纸”交给BeanFactory，让BeanFactory为我们生产一个FXNewsProvider，如以下代码所示：

```
BeanFactory container = ➥
new XmlBeanFactory(new ClassPathResource("配置文件路径"));
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");
newsProvider.getAndPersistNews();
```

或者如以下代码所示：

```
ApplicationContext container = ➥
new ClassPathXmlApplicationContext("配置文件路径");
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");
newsProvider.getAndPersistNews();
```

亦或如以下代码所示：

```
ApplicationContext container = ➥
```

```
new FileSystemXmlApplicationContext("配置文件路径");
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");
newsProvider.getAndPersistNews();
```

这就是拥有BeanFactory后的生活。当然，这只是使用BeanFactory后开发流程的一个概览而已，具体细节请容我慢慢道来。

4.2 BeanFactory的对象注册与依赖绑定方式[①]

BeanFactory作为一个IoC Service Provider，为了能够明确管理各个业务对象以及业务对象之间的依赖绑定关系，同样需要某种途径来记录和管理这些信息。上一章在介绍IoC Service Provider时，我们提到通常会有三种方式来管理这些信息。而BeanFactory几乎支持所有这些方式，很令人兴奋，不是吗？

4.2.1 直接编码方式

其实，把编码方式单独提出来称作一种方式并不十分恰当。因为不管什么方式，最终都需要编码才能“落实”所有信息并付诸使用。不过，通过这些代码，起码可以让我们更加清楚BeanFactory在底层是如何运作的。

下面来看一下我们的FX新闻系统相关类是如何注册并绑定的（见代码清单4-4）。

代码清单4-4　通过编码方式使用BeanFactory实现FX新闻相关类的注册及绑定

```
public static void main(String[] args)
{
   DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
   BeanFactory container = (BeanFactory)bindViaCode(beanRegistry);
   FXNewsProvider newsProvider = ➥
   (FXNewsProvider)container.getBean("djNewsProvider");
   newsProvider.getAndPersistNews();
}

public static BeanFactory bindViaCode(BeanDefinitionRegistry registry)
{
    AbstractBeanDefinition newsProvider = ➥
    new RootBeanDefinition(FXNewsProvider.class,true);
    AbstractBeanDefinition newsListener = ➥
    new RootBeanDefinition(DowJonesNewsListener.class,true);
    AbstractBeanDefinition newsPersister = ➥
    new RootBeanDefinition(DowJonesNewsPersister.class,true);
    // 将bean定义注册到容器中
    registry.registerBeanDefinition("djNewsProvider", newsProvider);
    registry.registerBeanDefinition("djListener", newsListener);
    registry.registerBeanDefinition("djPersister", newsPersister);
// 指定依赖关系
    // 1. 可以通过构造方法注入方式
    ConstructorArgumentValues argValues = new ConstructorArgumentValues();
    argValues.addIndexedArgumentValue(0, newsListener);
    argValues.addIndexedArgumentValue(1, newsPersister);
    newsProvider.setConstructorArgumentValues(argValues);
    // 2. 或者通过setter方法注入方式
```

① 在Spring的术语中，把BeanFactory需要使用的对象注册和依赖绑定信息称为Configuration Metadata。我们这里所展示的，实际上就是组织这些Configuration Metadata的各种方式。因此这个标题才这么长。

```
        MutablePropertyValues propertyValues = new MutablePropertyValues();
        propertyValues.addPropertyValue(new ropertyValue("newsListener",newsListener));
        propertyValues.addPropertyValue(new PropertyValue("newPersistener",newsPersister));
        newsProvider.setPropertyValues(propertyValues);
        // 绑定完成
        return (BeanFactory)registry;
    }
```

BeanFactory只是一个接口，我们最终需要一个该接口的实现来进行实际的Bean的管理，DefaultListableBeanFactory就是这么一个比较通用的BeanFactory实现类。DefaultListableBeanFactory除了间接地实现了BeanFactory接口，还实现了BeanDefinitionRegistry接口，该接口才是在BeanFactory的实现中担当Bean注册管理的角色。基本上，BeanFactory接口只定义如何访问容器内管理的Bean的方法，各个BeanFactory的具体实现类负责具体Bean的注册以及管理工作。BeanDefinitionRegistry接口定义抽象了Bean的注册逻辑。通常情况下，具体的BeanFactory实现类会实现这个接口来管理Bean的注册。它们之间的关系如图4-3所示。

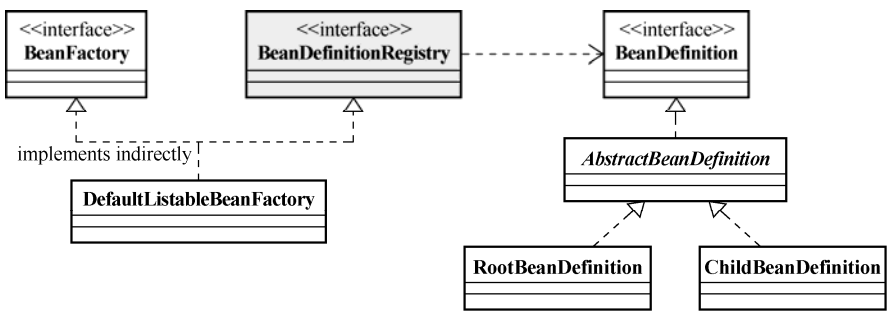


图4-3 BeanFactory、BeanDefinitionRegistry以及DefaultListableBeanFactory的关系

打个比方说，BeanDefinitionRegistry就像图书馆的书架，所有的书是放在书架上的。虽然你还书或者借书都是跟图书馆（也就是BeanFactory，或许BookFactory可能更好些）打交道，但书架才是图书馆存放各类图书的地方。所以，书架相对于图书馆来说，就是它的“BookDefinitionRegistry”。

每一个受管的对象，在容器中都会有一个BeanDefinition的实例（instance）与之相对应，该BeanDefinition的实例负责保存对象的所有必要信息，包括其对应的对象的class类型、是否是抽象类、构造方法参数以及其他属性等。当客户端向BeanFactory请求相应对象的时候，BeanFactory会通过这些信息为客户端返回一个完备可用的对象实例。RootBeanDefinition和ChildBeanDefinition是BeanDefinition的两个主要实现类。

现在，我们再来看这段绑定代码，应该就有“柳暗花明”的感觉了。

- 在main方法中，首先构造一个DefaultListableBeanFactory作为BeanDefinitionRegistry，然后将其交给bindViaCode方法进行具体的对象注册和相关依赖管理，然后通过bindViaCode返回的BeanFactory取得需要的对象，最后执行相应逻辑。在我们的实例里，当然就是取得FXNewsProvider进行新闻的处理。
- 在bindViaCode方法中，首先针对相应的业务对象构造与其相对应的BeanDefinition，使用了RootBeanDefinition作为BeanDefinition的实现类。构造完成后，将这些BeanDefinition注册到通过方法参数传进来的BeanDefinitionRegistry中。之后，因为我们的FXNewsProvider是采用的构造方法注入，所以，需要通过ConstructorArgument-

Values为其注入相关依赖。在这里为了同时说明setter方法注入，也同时展示了在Spring中如何使用代码实现setter方法注入。如果要运行这段代码，需要把setter方法注入部分的4行代码注释掉。最后，以BeanFactory的形式返回已经注册并绑定了所有相关业务对象的BeanDefinitionRegistry实例。

小心 最后一行的强制类型转换是有特定场景的。因为传入的DefaultListableBeanFactory同时实现了BeanFactory和BeanDefinitionRegistry接口，所以，这样做强制类型转换不会出现问题。但需要注意的是，单纯的BeanDefinitionRegistry是无法强制转换到BeanFactory类型的！

4.2.2　外部配置文件方式

Spring的IoC容器支持两种配置文件格式：Properties文件格式和XML文件格式。当然，如果你愿意也可以引入自己的文件格式，前提是真的需要。

采用外部配置文件时，Spring的IoC容器有一个统一的处理方式。通常情况下，需要根据不同的外部配置文件格式，给出相应的BeanDefinitionReader实现类，由BeanDefinitionReader的相应实现类负责将相应的配置文件内容读取并映射到BeanDefinition，然后将映射后的BeanDefinition注册到一个BeanDefinitionRegistry，之后，BeanDefinitionRegistry即完成Bean的注册和加载。当然，大部分工作，包括解析文件格式、装配BeanDefinition之类的工作，都是由BeanDefinitionReader的相应实现类来做的，BeanDefinitionRegistry只不过负责保管而已。整个过程类似于如下代码：

```
BeanDefinitionRegistry beanRegistry = <某个BeanDefinitionRegistry实现类，通常为➥
DefaultListableBeanFactory>;
BeanDefinitionReader   beanDefinitionReader = new BeanDefinitionReaderImpl(beanRegistry);
beanDefinitionReader.loadBeanDefinitions("配置文件路径");
// 现在我们就取得了一个可用的BeanDefinitionRegistry实例
```

1. Properties配置格式的加载

Spring提供了org.springframework.beans.factory.support.PropertiesBeanDefinitionReader类用于Properties格式配置文件的加载，所以，我们不用自己去实现BeanDefinitionReader，只要根据该类的读取规则，提供相应的配置文件即可。

对于FXNews系统的业务对象，我们采用如下文件内容（见代码清单4-5）进行配置加载。

代码清单4-5　Properties格式表达的依赖注入配置内容

```
djNewsProvider.(class)=..FXNewsProvider
# ----------通过构造方法注入的时候-------------
djNewsProvider.$0(ref)=djListener
djNewsProvider.$1(ref)=djPersister
# ----------通过setter方法注入的时候---------
# djNewsProvider.newsListener(ref)=djListener
# djNewsProvider.newPersistener(ref)=djPersister

djListener.(class)=..impl.DowJonesNewsListener

djPersister.(class)=..impl.DowJonesNewsPersister
```

这些内容是特定于Spring的PropertiesBeanDefinitionReader的，要了解更多内容，请参照

Spring的API参考文档。我们可以很容易地看明白代码清单4-5中的配置内容所要表达的意思。

- djNewsProvider作为beanName，后面通过.(class)表明对应的实现类是什么，实际上使用djNewsProvider.class=...的形式也是可以的，但Spring 1.2.6之后不再提倡使用，而提倡使用.(class)的形式。其他两个类的注册，djListener和djPersister，也是相同的道理。
- 通过在表示beanName的名称后添加.$[number]后缀的形式，来表示当前beanName对应的对象需要通过构造方法注入的方式注入相应依赖对象。在这里，我们分别将构造方法的第一个参数和第二个参数对应到djListener和djPersister。需要注意的一点，就是$0和$1后面的(ref)，(ref)用来表示所依赖的是引用对象，而不是普通的类型。如果不加(ref)，PropertiesBeanDefinitionReader会将djListener和djPersister作为简单的String类型进行注入，异常自然不可避免啦。
- FXNewsProvider采用的是构造方法注入，而为了演示setter方法注入在Properties配置文件中又是一个什么样子，以便于你更全面地了解基于Properties文件的配置方式，我们在下面增加了setter方法注入的例子，不过进行了注释。实际上，与构造方法注入最大的区别就是，它不使用数字顺序来指定注入的位置，而使用相应的属性名称来指定注入。newsListener和newPersistener恰好就是我们的FXNewsProvider类中所声明的属性名称。这印证了之前在比较构造方法注入和setter方法注入方式不同时提到的差异，即构造方法注入无法通过参数名称来标识注入的确切位置，而setter方法注入则可以通过属性名称来明确标识注入。与在Properties中表达构造方法注入一样，同样需要注意，如果属性名称所依赖的是引用对象，那么一定不要忘了(ref)。

当这些对象之间的注册和依赖注入信息都表达清楚之后，就可以将其加载到BeanFactory而付诸使用了。而这个加载过程实际上也就像我们之前总体上所阐述的那样，代码清单4-6中的内容再次演示了类似的加载过程。

代码清单4-6 加载Properties配置的BeanFactory的使用演示

```
public static void main(String[] args)
{
   DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
   BeanFactory container = (BeanFactory)bindViaPropertiesFile(beanRegistry);
   FXNewsProvider newsProvider =
(FXNewsProvider)container.getBean("djNewsProvider");
   newsProvider.getAndPersistNews();
}

public static BeanFactory bindViaPropertiesFile(BeanDefinitionRegistry registry)
{
   PropertiesBeanDefinitionReader reader =
new PropertiesBeanDefinitionReader(registry);
   reader.loadBeanDefinitions("classpath:../../binding-config.properties");
   return (BeanFactory)registry;
}
```

基于Properties的加载方式就是这么简单，所有的信息配置到Properties文件即可，不用再通过冗长的代码来完成对象的注册和依赖绑定。这些工作就交给相应的BeanDefinitionReader来做吧！哦，我的意思是，让给PropertiesBeanDefinitionReader来做。

注意　Spring提供的PropertiesBeanDefinitionReader是按照Spring自己的文件配置规则进行加载的，而同样的道理，你也可以按照自己的规则[①]来提供相应的Properties配置文件。只不过，现在需要实现你自己的“PropertiesBeanDefinitionReader”来读取并解析。这当然有“重新发明轮子”之嫌，不过，如果你只是想试验一下，也可以尝试哦。无非就是按照自己的规则把各个业务对象信息读取后，将编码方式的代码改造一下放到你自己的“PropertiesBeanDefinitionReader”而已。

2. XML配置格式的加载

XML配置格式是Spring支持最完整，功能最强大的表达方式。当然，一方面这得益于XML良好的语意表达能力；另一方面，就是Spring框架从开始就自始至终保持XML配置加载的统一性。同Properties配置加载类似，现在只不过是转而使用XML而已。Spring 2.*x*之前，XML配置文件采用DTD（Document Type Definition）实现文档的格式约束。2.*x*之后，引入了基于XSD（XML Schema Definition）的约束方式。不过，原来的基于DTD的方式依然有效，因为从DTD转向XSD只是“形式”上的转变，所以，后面的大部分讲解还会沿用DTD的方式，只有必要时才会给出特殊说明。

如果FX新闻系统对象按照XML配置方式进行加载的话，配置文件内容如代码清单4-7所示。

代码清单4-7　FX新闻系统相关类对应XML格式的配置内容

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" ➥
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
   <bean id="djNewsProvider" class="..FXNewsProvider">
     <constructor-arg index="0">
       <ref bean="djNewsListener"/>
     </constructor-arg>
     <constructor-arg index="1">
       <ref bean="djNewsPersister"/>
     </constructor-arg>
   </bean>

   <bean id="djNewsListener" class="..impl.DowJonesNewsListener">
   </bean>
   <bean id="djNewsPersister" class="..impl.DowJonesNewsPersister">
   </bean>
</beans>
```

我想这段内容不需要特殊说明吧，应该比Properties文件的内容要更容易理解。如果想知道这些内容背后的更多玄机，往后看吧！

有了XML配置文件，我们需要将其内容加载到相应的BeanFactory实现中，以供使用，如代码清单4-8所示。

代码清单4-8　加载XML配置文件的BeanFactory的使用演示

```
public static void main(String[] args)
{
```

① 也就是，如果我们不满足于Spring的默认规则，可以实现自己喜欢的规则表达方式。

```
    DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
    BeanFactory container = (BeanFactory)bindViaXMLFile(beanRegistry);
    FXNewsProvider newsProvider = ➥
    (FXNewsProvider)container.getBean("djNewsProvider");
    newsProvider.getAndPersistNews();
}

public static BeanFactory bindViaXMLFile(BeanDefinitionRegistry registry)
{
    XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(registry);
    reader.loadBeanDefinitions("classpath:../news-config.xml");
    return (BeanFactory)registry;
    // 或者直接
    //return new XmlBeanFactory(new ClassPathResource("../news-config.xml"));
}
```

与为Properties配置文件格式提供PropertiesBeanDefinitionReader相对应，Spring同样为XML格式的配置文件提供了现成的BeanDefinitionReader实现，即XmlBeanDefinitionReader。XmlBeanDefinitionReader负责读取Spring指定格式的XML配置文件并解析，之后将解析后的文件内容映射到相应的BeanDefinition，并加载到相应的BeanDefinitionRegistry中（在这里是DefaultListableBeanFactory）。这时，整个BeanFactory就可以放给客户端使用了。

除了提供XmlBeanDefinitionReader用于XML格式配置文件的加载，Spring还在DefaultListableBeanFactory的基础上构建了简化XML格式配置加载的XmlBeanFactory实现。从以上代码最后注释掉的一行，你可以看到使用了XmlBeanFactory之后，完成XML的加载和BeanFactory的初始化是多么简单。

注意 当然，如果你愿意，就像Properties方式可以扩展一样，XML方式的加载同样可以扩展。虽然XmlBeanFactory基本上已经十分完备了，但如果出于某种目的，XmlBeanFactory或者默认的XmlBeanDefinitionReader所使用的XML格式无法满足需要的话，你同样可以通过扩展XmlBeanDefinitionReader或者直接实现自己的BeanDefinitionReader来达到自定义XML配置文件加载的目的。Spring的可扩展性为你服务！

4.2.3 注解方式

可能你没有注意到，我在提到BeanFactory所支持的对象注册与依赖绑定方式的时候，说的是BeanFactory“几乎”支持IoC Service Provider可能使用的所有方式。之所以这么说，有两个原因。

- 在Spring 2.5发布之前，Spring框架并没有正式支持基于注解方式的依赖注入；
- Spring 2.5发布的基于注解的依赖注入方式，如果不使用classpath-scanning功能的话，仍然部分依赖于“基于XML配置文件”的依赖注入方式。

另外，注解是Java 5之后才引入的，所以，以下内容只适用于应用程序使用了Spring 2.5以及Java 5或者更高版本的情况之下。

如果要通过注解标注的方式为FXNewsProvider注入所需要的依赖，现在可以使用@Autowired以及@Component对相关类进行标记。代码清单4-9演示了FXNews相关类使用指定注解标注后的情况。

代码清单4-9 使用指定注解标注后的FXNews相关类

```
@Component
public class FXNewsProvider
```

```
{
   @Autowired
   private IFXNewsListener  newsListener;
   @Autowired
   private IFXNewsPersister newPersistener;

   public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister)
   {
      this.newsListener   = newsListner;
      this.newPersistener = newsPersister;
   }
   ...
}

@Component
public class DowJonesNewsListener implements IFXNewsListener
{
   ...
}

@Component
public class DowJonesNewsPersister implements IFXNewsPersister
{
   ...
}
```

@Autowired是这里的主角，它的存在将告知Spring容器需要为当前对象注入哪些依赖对象。而@Component则是配合Spring 2.5中新的classpath-scanning功能使用的。现在我们只要再向Spring的配置文件中增加一个“触发器”，使用@Autowired和@Component标注的类就能获得依赖对象的注入了。代码清单4-10给出的正是针对这部分功能的配置内容。

代码清单4-10　配置使用classpath-scanning功能

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xmlns:tx="http://www.springframework.org/schema/tx" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd ➥
http://www.springframework.org/schema/tx ➥
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
<context:component-scan base-package="cn.spring21.project.base.package"/>
</beans>
```

<context:component-scan/>会到指定的包（package）下面扫描标注有@Component的类，如果找到，则将它们添加到容器进行管理，并根据它们所标注的@Autowired为这些类注入符合条件的依赖对象。

在以上所有这些工作都完成之后，我们就可以像通常那样加载配置并执行当前应用程序了，如以下代码所示：

```
public static void main(String[] args)
{
   ApplicationContext ctx = new ClassPathXmlApplicationContext("配置文件路径");
   FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("FXNewsProvider");
```

```
    newsProvider.getAndPersistNews();
}
```

本章最后将详细讲解Spring 2.5新引入的“基于注解的依赖注入”。当前的内容只是让我们先从总体上有一个大概的印象，所以，不必强求自己现在就完全理解它们。

注意 Google Guice是一个完全基于注解方式、提供依赖注入服务的轻量级依赖注入框架，可以从Google Guice的站点获取有关这个框架的更多信息。

4.3 BeanFactory的XML之旅

XML格式的容器信息管理方式是Spring提供的最为强大、支持最为全面的方式。从Spring的参考文档到各Spring相关书籍，都是按照XML的配置进行说明的，这部分内容可以让你充分领略到Spring的IoC容器的魅力，以致于我们也不得不带你初次或者再次踏上Spring的XML之旅。

4.3.1 <beans>和<bean>

所有使用XML文件进行配置信息加载的Spring IoC容器，包括BeanFactory和ApplicationContext的所有XML相应实现，都使用统一的XML格式。在Spring 2.0版本之前，这种格式由Spring提供的DTD规定，也就是说，所有的Spring容器加载的XML配置文件的头部，都需要以下形式的DOCTYPE声明：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" ➥
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   ...
</beans>
```

从Spring 2.0版本之后，Spring在继续保持向前兼容的前提下，既可以继续使用DTD方式的DOCTYPE进行配置文件格式的限定，又引入了基于XML Schema的文档声明。所以，Spring 2.0之后，同样可以使用代码清单4-11所展示的基于XSD的文档声明。

代码清单4-11 基于XSD的Spring配置文件文档声明

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:util="http://www.springframework.org/schema/util" ➥
xmlns:jee="http://www.springframework.org/schema/jee" ➥
xmlns:lang="http://www.springframework.org/schema/lang" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xmlns:tx="http://www.springframework.org/schema/tx" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/util ➥
http://www.springframework.org/schema/util/spring-util-2.0.xsd ➥
http://www.springframework.org/schema/jee ➥
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd ➥
http://www.springframework.org/schema/lang ➥
http://www.springframework.org/schema/lang/spring-lang-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd ➥
```

```
    http://www.springframework.org/schema/tx ➥
    http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    </beans>
```

不过，不管使用哪一种形式的文档声明，实际上限定的元素基本上是相同的。让我们从最顶层的元素开始，看一下这两种文档声明都限定了哪些元素吧！

所有注册到容器的业务对象，在Spring中称之为Bean。所以，每一个对象在XML中的映射也自然而然地对应一个叫做<bean>的元素。既然容器最终可以管理所有的业务对象，那么在XML中把这些叫做<bean>的元素组织起来的，就叫做<beans>。多个<bean>组成一个<beans>很容易理解，不是吗？

1. <beans>之唯我独尊

<beans>是XML配置文件中最顶层的元素，它下面可以包含0或者1个<description>和多个<bean>以及<import>或者<alias>，如图4-4所示。

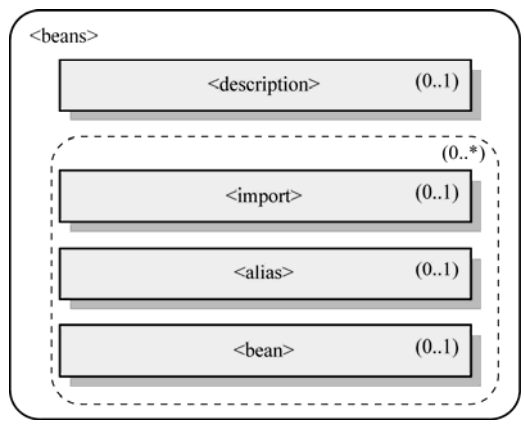


图4-4　<beans>与下一层元素的关系

<beans>作为所有<bean>的“统帅”，它拥有相应的属性（attribute）对所辖的<bean>进行统一的默认行为设置，包括如下几个。

- **default-lazy-init**。其值可以指定为true或者false，默认值为false。用来标志是否对所有的<bean>进行延迟初始化。
- **default-autowire**。可以取值为no、byName、byType、constructor以及autodetect。默认值为no，如果使用自动绑定的话，用来标志全体bean使用哪一种默认绑定方式。
- **default-dependency-check**。可以取值none、objects、simple以及all，默认值为none，即不做依赖检查。
- **default-init-method**。如果所管辖的<bean>按照某种规则，都有同样名称的初始化方法的话，可以在这里统一指定这个初始化方法名，而不用在每一个<bean>上都重复单独指定。
- **default-destroy-method**。与default-init-method相对应，如果所管辖的bean有按照某种规则使用了相同名称的对象销毁方法，可以通过这个属性统一指定。

注意　当然，如果你不清楚上面这些默认的属性具体有什么用，那也不必着急。在看完对<bean>的讲解之后，再回头来看，就会明了多了。给出这些信息，是想让你知道，如果在某

个场景下需要对大部分<bean>都重复设置某些行为的话，可以回头看一下，利用<beans>是否可以减少这种不必要的工作。

2. **<description>、<import>和<alias>**

之所以把这几个元素放到一起讲解，是因为通常情况下它们不是必需的。不过，了解一下也没什么不好，不是吗？

- <description>

可以通过<description>在配置的文件中指定一些描述性的信息。通常情况下，该元素是省略的。当然，如果愿意，<description>随时可以为我们效劳。

- <import>

通常情况下，可以根据模块功能或者层次关系，将配置信息分门别类地放到多个配置文件中。在想加载主要配置文件，并将主要配置文件所依赖的配置文件同时加载时，可以在这个主要的配置文件中通过<import>元素对其所依赖的配置文件进行引用。比如，如果A.xml中的<bean>定义可能依赖B.xml中的某些<bean>定义，那么就可以在A.xml中使用<import>将B.xml引入到A.xml，以类似于<import resource="B.xml"/>的形式。

但是，这个功能在我看来价值不大，因为容器实际上可以同时加载多个配置，没有必要非通过一个配置文件来加载所有配置。不过，或许在有些场景中使用这种方式比较方便也说不定。

- <alias>

可以通过<alias>为某些<bean>起一些“外号”（别名），通常情况下是为了减少输入。比如，假设有个<bean>，它的名称为dataSourceForMasterDatabase，你可以为其添加一个<alias>，像这样<alias name="dataSourceForMasterDatabase" alias="masterDataSource"/>。以后通过dataSourceForMasterDatabase或者masterDataSource来引用这个<bean>都可以，只要你觉得方便就行。

4.3.2 孤孤单单一个人

哦，错了，是孤孤单单一个Bean。每个业务对象作为个体，在Spring的XML配置文件中是与<bean>元素一一对应的。窥一斑而知全豹，只要我们了解单个的业务对象是如何配置的，剩下的就可以“依葫芦画瓢”了。所以，让我们先从最简单的单一对象配置开始吧！如下代码演示了最基础的对象配置形式：

```
<bean id="djNewsListener" class="..impl.DowJonesNewsListener">
</bean>
```

- id属性

通常，每个注册到容器的对象都需要一个唯一标志来将其与“同处一室”的“兄弟们”区分开来，就好像我们每一个人都有一个身份证号一样（重号的话就比较麻烦）。通过id属性来指定当前注册对象的beanName是什么。这里，通过id指定beanName为djNewsListener。实际上，并非任何情况下都需要指定每个<bean>的id，有些情况下，id可以省略，比如后面会提到的内部<bean>以及不需要根据beanName明确依赖关系的场合等。

除了可以使用id来指定<bean>在容器中的标志，还可以使用name属性来指定<bean>的别名（alias）。比如，以上定义，我们还可以像如下代码这样，为其添加别名：

```
<bean id="djNewsListener"
```

```
    name="/news/djNewsListener,dowJonesNewsListener"
    class="..impl.DowJonesNewsListener">
</bean>
```

与id属性相比，name属性的灵活之处在于，name可以使用id不能使用的一些字符，比如/。而且还可以通过逗号、空格或者冒号分割指定多个name。name的作用跟使用<alias>为id指定多个别名基本相同：

```
<alias name="djNewsListener" alias="/news/djNewsListener"/>
<alias name="djNewsListener" alias="dowJonesNewsListener"/>
```

- class属性

每个注册到容器的对象都需要通过<bean>元素的class属性指定其类型，否则，容器可不知道这个对象到底是何方神圣。在大部分情况下，该属性是必须的。仅在少数情况下不需要指定，如后面将提到的在使用抽象配置模板的情况下。

4.3.3　Help Me, Help You[①]

在大部分情况下，你不太可能选择单独“作战”，业务对象也是；各个业务对象之间会相互协作来更好地完成同一使命。这时，各个业务对象之间的相互依赖就是无法避免的。对象之间需要相互协作，在横向上它们存在一定的依赖性。而现在我们就是要看一下，在Spring的IoC容器的XML配置中，应该如何表达这种依赖性。

既然业务对象现在都符合IoC的规则，那么要了解的表达方式其实也很简单，无非就是看一下构造方法注入和setter方法注入通过XML是如何表达的而已。那么，让我们开始吧！

1. 构造方法注入的XML之道

按照Spring的IoC容器配置格式，要通过构造方法注入方式，为当前业务对象注入其所依赖的对象，需要使用<constructor-arg>。正常情况下，如以下代码所示：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <constructor-arg>
    <ref bean="djNewsListener"/>
  </constructor-arg>
  <constructor-arg>
    <ref bean="djNewsPersister"/>
  </constructor-arg>
</bean>
```

对于<ref>元素，稍后会进行详细说明。这里你只需要知道，通过这个元素来指明容器将为djNewsProvider这个<bean>注入通过<ref>所引用的Bean实例。这种方式可能看起来或者编写起来不是很简洁，最新版本的Spring也支持配置简写形式，如以下代码所示：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <constructor-arg ref="djNewsListener"/>
  <constructor-arg ref="djNewsPersister"/>
</bean>
```

简洁多了不是嘛？其实，无非就是表达方式不同而已，实际达到的效果是一样的。

有些时候，容器在加载XML配置的时候，因为某些原因，无法明确配置项与对象的构造方法参数列表的一一对应关系，就需要请<constructor-arg>的type或者index属性出马。比如，对象存在多

① 这句话是*Warcraft*中女巫的一句台词，这里用这句话来类比多个<bean>之间的关系：互相依赖，互相帮助以完成同一目标。

个构造方法，当参数列表数目相同而类型不同的时候，容器无法区分应该使用哪个构造方法来实例化对象，或者构造方法可能同时传入最少两个类型相同的对象。

- type属性

假设有一个对象定义如代码清单4-12所示。

代码清单4-12 随意声明的一个业务对象定义

```
public class MockBusinessObject {
   private String dependency1;
   private int    dependency2;

   public MockBusinessObject(String dependency)
   {
      this.dependency1 = dependency;
   }

   public MockBusinessObject(int dependency)
   {
      this.dependency2 = dependency;
   }
   ...

   @Override
   public String toString() {
      return new ToStringBuilder(this) ➥
      .append("dependency1", dependency1) ➥
      .append("dependency2", dependency2).toString();
   }
}
```

该类声明了两个构造方法，分别都只是传入一个参数，且参数类型不同。这时，我们可以进行配置，如以下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
   <constructor-arg>
      <value>111111</value>
   </constructor-arg>
</bean>
```

如果从BeanFactory取得该对象并调用toString()查看的话，我们会发现Spring调用的是第一个构造方法，因为输出是如下内容：

```
..MockBusinessObject@f73c1[dependency1=111111,dependency2=0]
```

但是，如果我们想调用的却是第二个传入int类型参数的构造方法，又该如何呢？可以使用type属性，通过指定构造方法的参数类型来解决这一问题，配置内容如下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
   <constructor-arg type="int">
      <value>111111</value>
   </constructor-arg>
</bean>
```

现在，我们得到了自己想要的对象实例，如下的控制台输出信息印证了这一点：

```
..MockBusinessObject@f73c1[dependency1=<null>,dependency2=111111]
```

- index属性

当某个业务对象的构造方法同时传入了多个类型相同的参数时，Spring又该如何将这些配置中的信息与实际对象的参数一一对应呢？好在，如果配置项信息和对象参数可以按照顺序初步对应的话，

Spring还是可以正常工作的，如代码清单4-13所示。

代码清单4-13　随意声明的一个业务对象定义

```
public class MockBusinessObject {
   private String dependency1;
   private String dependency2;

   public MockBusinessObject(String dependency1,String    dependency2)
   {
      this.dependency1 = dependency1;
      this.dependency2 = dependency2;
   }
   ...

   @Override
   public String toString() {
      return new ToStringBuilder(this) ➥
      .append("dependency1", dependency1) ➥
      .append("dependency2", dependency2).toString();
   }
}
```

并且，配置内容如以下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
   <constructor-arg value="11111"/>
   <constructor-arg value="22222"/>
</bean>
```

那么，我们可以得到如下对象：

```
..MockBusinessObject@1ef8cf3[dependency1=11111,dependency2=22222]
```

但是，如果要让“11111”作为对象的第二个参数，而将“22222”作为第一个参数来构造对象，又该如何呢？好！可以颠倒配置项，如以下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
   <constructor-arg value="22222"/>
   <constructor-arg value="11111"/>
</bean>
```

不过，还有一种方式，那就是像如下代码所示的那样，使用index属性：

```
<bean id="mockBO" class="..MockBusinessObject">
   <constructor-arg index="1" value="11111"/>
   <constructor-arg index="0" value="22222"/>
</bean>
```

这时，同样可以得到想要的对象实例，以下控制台输出表明了这一点：

```
..MockBusinessObject@ecd7e[dependency1=22222,dependency2=11111]
```

注意 index属性的取值从0开始，与一般的数组下标取值相同。所以，指定的第一个参数的index应该是0，第二个参数的index应该是1，依此类推。

2. setter方法注入的XML之道

与构造方法注入可以使用<constructor-arg>注入配置相对应，Spring为setter方法注入提供了<property>元素。

<property>有一个name属性（attribute），用来指定该<property>将会注入的对象所对应的实例变量名称。之后通过value或者ref属性或者内嵌的其他元素来指定具体的依赖对象引用或者值，如以下代码所示：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <property name="newsListener">
    <ref bean="djNewsListener"/>
  </property>
  <property name="newPersistener">
    <ref bean="djNewsPersister"/>
  </property>
</bean>
```

当然，如果只是使用<property>进行依赖注入的话，请确保你的对象提供了默认的构造方法，也就是一个参数都没有的那个。

以上配置形式还可以简化为如下形式：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <property name="newsListener" ref="djNewsListener"/>
  <property name="newPersistener" ref="djNewsPersister"/>
</bean>
```

使用<property>的setter方法注入和使用<constructor-arg>的构造方法注入并不是水火不容的。实际上，如果需要，可以同时使用这两个元素：

```
<bean id="mockBO" class="..MockBusinessObject">
  <constructor-arg value="11111"/>
  <property name="dependency2" value="22222"/>
</bean>
```

当然，现在需要MockBusinessObject提供一个只有一个String类型参数的构造方法，并且为dependency2提供了相应的setter方法。代码清单4-14演示了符合条件的一个业务对象定义。

代码清单4-14 随意声明的一个同时支持构造方法注入和setter方法注入的对象定义

```
public class MockBusinessObject {
  private String dependency1;
  private String dependency2;

  public MockBusinessObject(String dependency)
  {
    this.dependency1 = dependency;
  }

  public void setDependency2(String dependency2) {
    this.dependency2 = dependency2;
  }
  ...
}
```

3. <property>和<constructor-arg>中可用的配置项

之前我们看到，可以通过在<property>和<constructor-arg>这两个元素内部嵌套<value>或者<ref>，来指定将为当前对象注入的简单数据类型或者某个对象的引用。不过，为了能够指定多种注入类型，Spring还提供了其他的元素供我们使用，这包括bean、ref、idref、value、null、list、set、map、props。下面我们来逐个详细讲述它们。

提示　以下涉及的所有内嵌元素，对于<property>和<constructor-arg>都是通用的。

(1) <value>。可以通过value为主体对象注入简单的数据类型，不但可以指定String类型的数据，而且可以指定其他Java语言中的原始类型以及它们的包装器（wrapper）类型，比如int、Integer等。容器在注入的时候，会做适当的转换工作（我们会在后面揭示转换的奥秘）。你之前已经见过如何使用<value>了，不过让我们通过如下代码来重新认识一下它：

```
<constructor-arg>
   <value>111111</value>
</constructor-arg>
<property name="attributeName">
   <value>222222</value>
</property>
```

当然，如果愿意，你也可以使用如下的简化形式（不过这里的value是以上一层元素的属性身份出现）：

```
<constructor-arg value="111111"/>
<property name="attributeName" value="222222"/>
```

需要说明的是，<value>是最“底层”的元素，它内部不能再嵌套使用其他元素了。

(2) <ref>。使用ref来引用容器中其他的对象实例，可以通过ref的local、parent和bean属性来指定引用的对象的beanName是什么。代码清单4-15演示了ref及其三个对应属性的使用情况。

代码清单4-15　<ref>及其local、parent和bean属性的使用

```
constructor-arg>
   <ref local="djNewsPersister"/>
</constructor-arg>
或者
<constructor-arg>
   <ref parent="djNewsPersister"/>
</constructor-arg>
或者
<constructor-arg>
   <ref bean="djNewsPersister"/>
</constructor-arg>
```

local、parent和bean的区别在于：

- local只能指定与当前配置的对象在同一个配置文件的对象定义的名称(可以获得XML解析器的id约束验证支持)；
- parent则只能指定位于当前容器的父容器中定义的对象引用；

注意　BeanFactory可以分层次（通过实现HierarchicalBeanFactory接口），容器A在初始化的时候，可以首先加载容器B中的所有对象定义，然后再加载自身的对象定义，这样，容器B就成为了容器A的父容器，容器A可以引用容器B中的所有对象定义：

```
BeanFactory parentContainer = new XmlBeanFactory(new ClassPathResource("父容器配置文件路径"));
BeanFactory childContainer  = new XmlBeanFactory(new ClassPathResource("子容器配置文件路径"),parentContainer);
```

childContainer中定义的对象，如果通过parent指定依赖，则只能引用parentContainer中的对象定义。

- bean则基本上通吃，所以，通常情况下，直接使用bean来指定对象引用就可以了。

<ref>的定义为<!ELEMENT ref EMPTY>，也就是说，它下面没有其他子元素可用了，别硬往人家肚子里塞东西哦。

(3) <idref>。如果要为当前对象注入所依赖的对象的名称，而不是引用，那么通常情况下，可以使用<value>来达到这个目的，使用如下形式：

```
<property name="newsListenerBeanName">
   <value>djNewsListener</value>
</property>
```

但这种场合下，使用idref才是最为合适的。因为使用idref，容器在解析配置的时候就可以帮你检查这个beanName到底是否存在，而不用等到运行时才发现这个beanName对应的对象实例不存在。毕竟，输错名字的问题很常见。以下代码演示了idref的使用：

```
<property name="newsListenerBeanName">
   <idref bean="djNewsListener"/>
</property>
```

这段配置跟上面使用<value>达到了相同的目的，不过更加保险。如果愿意，也可以通过local而不是bean来指定最终值，不过，bean比较大众化哦。

(4) 内部<bean>。使用<ref>可以引用容器中独立定义的对象定义。但有时，可能我们所依赖的对象只有当前一个对象引用，或者某个对象定义我们不想其他对象通过<ref>引用到它。这时，我们可以使用内嵌的<bean>，将这个私有的对象定义仅局限在当前对象。对于FX新闻系统的DowJonesNewsListener而言，实际上只有道琼斯的FXNewsProvider会使用它。而且，我们也不想让其他对象引用到它。为此完全可以像代码清单4-16这样，将它配置为内部<bean>的形式。

代码清单4-16 内部<bean>的配置演示

```
<bean id="djNewsProvider" class="..FXNewsProvider">
   <constructor-arg index="0">
      <bean class="..impl.DowJonesNewsListener">
      </bean>
   </constructor-arg>
   <constructor-arg index="1">
      <ref bean="djNewsPersister"/>
   </constructor-arg>
</bean>
```

这样，该对象实例就只有当前的djNewsProvider可以使用，其他对象无法取得该对象的引用。

注意 因为就只有当前对象引用内部<bean>所指定的对象，所以，内部<bean>的id不是必须的。当然，如果你愿意指定id，那也是无所谓的。如下所示：

```
<constructor-arg index="0">
   <bean id="djNewsListener" class="..impl.DowJonesNewsListener">
   </bean>
</constructor-arg>
```

内部<bean>的配置只是在位置上有所差异，但配置项上与其他的<bean>是没有任何差别的。也就是说，<bean>内嵌的所有元素，内部<bean>的<bean>同样可以使用。如果内部<bean>对应的对象还依赖于其他对象，你完全可以像其他独立的<bean>定义一样为其配置相关依赖，没有任何差别。

(5) <list>。<list>对应注入对象类型为java.util.List及其子类或者数组类型的依赖对象。

通过<list>可以有序地为当前对象注入以collection形式声明的依赖。代码清单4-17给出了一个使用<list>的实例演示。

代码清单4-17 使用<list>进行依赖注入的对象定义以及相关配置内容

```
public class MockDemoObject
{
   private List param1;
   private String[] param2;
   ...
   // 相应的setter和getter方法
   ...
}

配置类似于
<property name="param1">
   <list>
      <value> something</value>
      <ref bean="someBeanName"/>
      <bean class="..."/>
   </list>
</property>
<property name="param2">
   <list>
      <value>stringValue1</value>
      <value>stringValue2</value>
   </list>
</property>
```

注意，<list>元素内部可以嵌套其他元素，并且可以像param1所展示的那样夹杂配置。但是，从好的编程实践来说，这样的处理并不恰当，除非你真的知道自己在做什么！（以上只是出于演示的目的才会如此配置）。

(6) <set>。如果说<list>可以帮你有序地注入一系列依赖的话，那么<set>就是无序的，而且，对于set来说，元素的顺序本来就是无关紧要的。<set>对应注入Java Collection中类型为java.util.Set或者其子类的依赖对象。代码清单4-18演示了通常情况下<set>的使用场景。

代码清单4-18 使用<set>进行依赖注入的对象定义以及相关配置内容

```
public class MockDemoObject
{
   private Set valueSet;
   // 必要的setter和getter方法
   ...
}
配置类似于
<property name="valueSet">
   <set>
      <value> something</value>
      <ref bean="someBeanName"/>
      <bean class="..."/>
      <list>
         ...
      </list>
   </set>
</property>
```

例子就是例子，只是为了给你演示这个元素到底有多少能耐。从配置上来说，这样多层嵌套、多

元素混杂配置是完全没有问题的。不过，各位在具体编程实践的时候可要小心了。如果你真的想这么夹杂配置的话，不好意思，估计ClassCastException会很愿意来亲近你，而这跟容器或者配置一点儿关系也没有。

(7) <map>。与列表（list）使用数字下标来标识元素不同，映射（map）可以通过指定的键（key）来获取相应的值。如果说在<list>中混杂不同元素不是一个好的实践（方式）的话，你就应该求助<map>。<map>与<list>和<set>的相同点在于，都是为主体对象注入Collection类型的依赖，不同点在于它对应注入java.util.Map或者其子类类型的依赖对象。代码清单4-19演示了<map>的通常使用场景。

代码清单4-19 使用<map>进行依赖注入的对象定义以及相关配置内容

```
public class MockDemoObject
{
   private Map mapping;
   // 必要的setter和getter方法
   ...
}
配置类似于
<property name="valueSet">
   <map>
     <entry key="strValueKey">
       <value>something</value>
     </entry>
     <entry>
       <key>objectKey</key>
       <ref bean="someObject"/>
     </entry>
     <entry key-ref="lstKey">
       <list>
         ...
       </list>
     </entry>
     ...
   </map>
</property>
```

对于<map>来说，它可以内嵌任意多个<entry>，每一个<entry>都需要为其指定一个键和一个值，就跟真正的java.util.Map所要求的一样。

- 指定entry的键。可以使用<entry>的属性——key或者key-ref来指定键，也可以使用<entry>的内嵌元素<key>来指定键，这完全看个人喜好，但两种方式可以达到相同的效果。在<key>内部，可以使用以上提到的任何元素来指定键，从简单的<value>到复杂的Collection，只要映射需要，你可以任意发挥。
- 指定entry对应的值。<entry>内部可以使用的元素，除了<key>是用来指定键的，其他元素可以任意使用，来指定entry对应的值。除了之前提到的那些元素，还包括马上就要谈到的<props>。如果对应的值只是简单的原始类型或者单一的对象引用，也可以直接使用<entry>的value或者value-ref这两个属性来指定，从而省却多敲入几个字符的工作量。

注意 key属性用于指定通常的简单类型的键，而key-ref则用于指定对象的引用作为键。

所以，如果你不想敲那么些字符，可以像代码清单4-20所展示的那样使用<map>进行依赖注入的

配置。

代码清单4-20　简化版的<map>配置使用演示

```
public class MockDemoObject
{
   private Map mapping;
   // 必要的setter和getter方法
   ...
}
配置类似于
<property name="valueSet">
   <map>
     <entry key="strValueKey" value="something"/>
     <entry key-ref="" value-ref="someObject"/>
     <entry key-ref="lstKey">
       <list>
         ...
       </list>
     </entry>
     ...
   </map>
</property>
```

(8) <props>。<props>是简化后了的<map>，或者说是特殊化的map，该元素对应配置类型为java.util.Properties的对象依赖。因为Properties只能指定String类型的键（key）和值，所以，<props>的配置简化很多，只有固定的格式，见代码清单4-21。

代码清单4-21　使用<props>进行依赖注入的场景演示

```
public class MockDemoObject
{
   private Properties emailAddrs;
   // 必要的setter和getter方法
   ...
}
配置类似于
<property name="valueSet">
   <props>
     <prop key="author">fujohnwang@gmail.com</prop>
     <prop key="support">support@spring21.cn</prop>
     ...
   </props>
</property>
```

每个<props>可以嵌套多个<prop>，每个<prop>通过其key属性来指定键，在<prop>内部直接指定其所对应的值。<prop>内部没有任何元素可以使用，只能指定字符串，这个是由java.util.Properties的语意决定的。

(9) <null/>。最后一个提到的元素是<null/>，这是最简单的一个元素，因为它只是一个空元素，而且通常使用到它的场景也不是很多。对于String类型来说，如果通过value以这样的方式指定注入，即<value></value>，那么，得到的结果是""，而不是null。所以，如果需要为这个string对应的值注入null的话，请使用<null/>。当然，这并非仅限于String类型，如果某个对象也有类似需求，请不要犹豫。代码清单4-22演示了一个使用<null/>的简单场景。

代码清单4-22 使用<null/>进行依赖注入的简单场景演示

```
public class MockDemoObject
{
   private String param1;
   private Object param2;
   // 必要的setter和getter方法
   ...
}
配置为
<property name="param1">
   <null/>
</property>
<property name="param2">
   <null/>
</property>
实际上就相当于
public class MockDemoObject
{
   private String param1=null;
   private Object param2=null;
   // 必要的setter和getter方法
   ...
}
虽然这里看起来没有太大意义!
```

4. depends-on

通常情况下，可以直接通过之前提到的所有元素，来显式地指定bean之间的依赖关系。这样，容器在初始化当前bean定义的时候，会根据这些元素所标记的依赖关系，首先实例化当前bean定义所依赖的其他bean定义。但是，如果某些时候，我们没有通过类似<ref>的元素明确指定对象A依赖于对象B的话，如何让容器在实例化对象A之前首先实例化对象B呢？

考虑以下所示代码：

```
public class SystemConfigurationSetup
{
   static
   {
      DOMConfigurator.configure("配置文件路径");
      // 其他初始化代码
   }
   ...
}
```

系统中所有需要日志记录的类，都需要在这些类使用之前首先初始化log4j。那么，就会非显式地依赖于SystemConfigurationSetup的静态初始化块。如果ClassA需要使用log4j，那么就必须在bean定义中使用depends-on来要求容器在初始化自身实例之前首先实例化SystemConfigurationSetup，以保证日志系统的可用，如下代码演示的正是这种情况：

```
<bean id="classAInstance" class="...ClassA" depends-on="configSetup"/>

<bean id="configSetup" class="SystemConfigurationSetup"/>
```

举log4j在静态代码块（static block）中初始化的例子在实际系统中其实不是很合适，因为通常在应用程序的主入口类初始化日志就可以了。这里主要是给出depends-on可能的使用场景，大部分情况下，是那些拥有静态代码块初始化代码或者数据库驱动注册之类的场景。

如果说ClassA拥有多个类似的非显式依赖关系，那么，你可以在ClassA的depends-on中通过逗号分割各个beanName，如下代码所示：

```
<bean id="classAInstance" class="...ClassA" depends-on="configSetup,configSetup2,..."/>

<bean id="configSetup" class="SystemConfigurationSetup"/>
<bean id="configSetup2" class="SystemConfigurationSetup2"/>
```

5. **autowire**

除了可以通过配置明确指定bean之间的依赖关系，Spirng还提供了根据bean定义的某些特点将相互依赖的某些bean直接自动绑定的功能。通过<bean>的autowire属性，可以指定当前bean定义采用某种类型的自动绑定模式。这样，你就无需手工明确指定该bean定义相关的依赖关系，从而也可以免去一些手工输入的工作量。

Spring提供了5种自动绑定模式，即no、byName、byType、constructor和autodetect，下面是它们的具体介绍。

- no

容器默认的自动绑定模式，也就是不采用任何形式的自动绑定，完全依赖手工明确配置各个bean之间的依赖关系，以下代码演示的两种配置是等效的：

```
<bean id="beanName" class="..."/>

或者

<bean id="beanName" class="..." autowire="no"/>
```

- byName

按照类中声明的实例变量的名称，与XML配置文件中声明的bean定义的beanName的值进行匹配，相匹配的bean定义将被自动绑定到当前实例变量上。这种方式对类定义和配置的bean定义有一定的限制。假设我们有如下所示的类定义：

```
public class Foo
{
   private Bar emphasisAttribute;
   ...
   // 相应的setter方法定义
}
public class Bar
{
   ...
}
```

那么应该使用如下代码所演示的自动绑定定义，才能达到预期的目的：

```
<bean id="fooBean" class="...Foo" autowire="byName">
</bean>
<bean id="emphasisAttribute" class="...Bar">
</bean>
```

需要注意两点：第一，我们并没有明确指定fooBean的依赖关系，而仅指定了它的autowire属性为byName；第二，第二个bean定义的id为emphasisAttribute，与Foo类中的实例变量名称相同。

- byType

如果指定当前bean定义的autowire模式为byType，那么，容器会根据当前bean定义类型，分析其相应的依赖对象类型，然后到容器所管理的所有bean定义中寻找与依赖对象类型相同的bean定义，然

后将找到的符合条件的bean自动绑定到当前bean定义。

对于byName模式中的实例类Foo来说，容器会在其所管理的所有bean定义中寻找类型为Bar的bean定义。如果找到，则将找到的bean绑定到Foo的bean定义；如果没有找到，则不做设置。但如果找到多个，容器会告诉你它解决不了“该选用哪一个”的问题，你只好自己查找原因，并自己修正该问题。所以，byType只能保证，在容器中只存在一个符合条件的依赖对象的时候才会发挥最大的作用，如果容器中存在多个相同类型的bean定义，那么，不好意思，采用手动明确配置吧！

指定byType类型的autowire模式与byName没什么差别，只是autowire的值换成byType而已，可以参考如下代码：

```
<bean id="fooBean" class="...Foo" autowire="byType">
</bean>

<bean id="anyName" class="...Bar">
</bean>
```

- constructor

byName和byType类型的自动绑定模式是针对property的自动绑定，而constructor类型则是针对构造方法参数的类型而进行的自动绑定，它同样是byType类型的绑定模式。不过，constructor是匹配构造方法的参数类型，而不是实例属性的类型。与byType模式类似，如果找到不止一个符合条件的bean定义，那么，容器会返回错误。使用上也与byType没有太大差别，只不过是应用到需要使用构造方法注入的bean定义之上，代码清单4-23给出了一个使用construtor模式进行自动绑定的简单场景演示。

代码清单4-23 construtor类型自动绑定的使用场景演示

```
public class Foo
{
   private Bar bar;
   public Foo(Bar arg)
   {
      this.bar = arg;
   }
   ...
}
相应配置为
<bean id="foo" class="...Foo" autowire="constructor"/>

<bean id="bar" class="...Bar">
</bean>
```

- autodetect

这种模式是byType和constructor模式的结合体，如果对象拥有默认无参数的构造方法，容器会优先考虑byType的自动绑定模式。否则，会使用constructor模式。当然，如果通过构造方法注入绑定后还有其他属性没有绑定，容器也会使用byType对剩余的对象属性进行自动绑定。

小心

- 手工明确指定的绑定关系总会覆盖自动绑定模式的行为。
- 自动绑定只应用于“原生类型、String类型以及Classes类型以处”的对象类型，对“原生类型、String类型和Classes类型”以及“这些类型的数组”应用自动绑定是无效的。

自动绑定与手动明确绑定

自动绑定和手动明确绑定各有利弊。自动绑定的优点有如下两点。

(1) 某种程度上可以有效减少手动敲入配置信息的工作量。

(2) 某些情况下，即使为当前对象增加了新的依赖关系，但只要容器中存在相应的依赖对象，就不需要更改任何配置信息。

自动绑定的缺点有如下几点。

(1) 自动绑定不如明确依赖关系一目了然。我们可以根据明确的依赖关系对整个系统有一个明确的认识，但使用自动绑定的话，就可能需要在类定义以及配置文件之间，甚至各个配置文件之间来回转换以取得相应的信息。

(2) 某些情况下，自动绑定无法满足系统需要，甚至导致系统行为异常或者不可预知。根据类型（byType）匹配进行的自动绑定，如果系统中增加了另一个相同类型的bean定义，那么整个系统就会崩溃；根据名字（byName）匹配进行的自动绑定，如果把原来系统中相同名称的bean定义类型给换掉，就会造成问题，而这些可能都是在不经意间发生的。

(3) 使用自动绑定，我们可能无法获得某些工具的良好支持，比如Spring IDE。

通常情况下，只要有良好的XML编辑器支持，我不会介意多敲那几个字符。起码自己可以对整个系统的行为有完全的把握。当然，任何事物都不绝对，只要根据相应场景找到合适的就可以。

噢，对了，差点儿忘了！作为所有<bean>的统帅，<beans>有一个default-autowire属性，它可以帮我们省去为多个<bean>单独设置autowire属性的麻烦，default-autowire的默认值为no，即不进行自动绑定。如果想让系统中所有的<bean>定义都使用byType模式的自动绑定，我们可以使用如下配置内容：

```
<beans default-autowire="byType">
   <bean id="..." class="..."/>
   ...
</beans>
```

6. **dependency-check**

我们可以使用每个<bean>的dependency-check属性对其所依赖的对象进行最终检查，就好像电影里每队美国大兵上战场之前，带队的军官都会朝着士兵大喊“检查装备，check，recheck”是一个道理。该功能主要与自动绑定结合使用，可以保证当自动绑定完成后，最终确认每个对象所依赖的对象是否按照所预期的那样被注入。当然，并不是说不可以与平常的明确绑定方式一起使用。

该功能可以帮我们检查每个对象某种类型的所有依赖是否全部已经注入完成，不过可能无法细化到具体的类型检查。但某些时候，使用setter方法注入就是为了拥有某种可以设置也可以不设置的灵活性，所以，这种依赖检查并非十分有用，尤其是在手动明确绑定依赖关系的情况下。

与军官会让大兵检查枪支弹药和防弹衣等不同装备一样，可以通过dependency-check指定容器帮我们检查某种类型的依赖，基本上有如下4种类型的依赖检查。

- none。不做依赖检查。将dependency-check指定为none跟不指定这个属性等效，所以，还是不要多敲那几个字符了吧。默认情况下，容器以此为默认值。
- simple。如果将dependency-check的值指定为simple，那么容器会对简单属性类型以及相关的collection进行依赖检查，对象引用类型的依赖除外。
- object。只对对象引用类型依赖进行检查。

- all。将simple和object相结合，也就是说会对简单属性类型以及相应的collection和所有对象引用类型的依赖进行检查。

总地来说，控制得力的话，这个依赖检查的功能我们基本可以不考虑使用。

7. **lazy-init**

延迟初始化（lazy-init）这个特性的作用，主要是可以针对ApplicationContext容器的bean初始化行为施以更多控制。与BeanFactory不同，ApplicationContext在容器启动的时候，就会马上对所有的“singleton的bean定义”[①]进行实例化操作。通常这种默认行为是好的，因为如果系统有问题的话，可以在第一时间发现这些问题，但有时，我们不想某些bean定义在容器启动后就直接实例化，可能出于容器启动时间的考虑，也可能出于其他原因的考虑。总之，我们想改变某个或者某些bean定义在ApplicationContext容器中的默认实例化时机。这时，就可以通过<bean>的lazy-init属性来控制这种初始化行为，如下代码所示：

```
<bean id="lazy-init-bean" class="..." lazy-init="true"/>
<bean id="not-lazy-init-bean" class="..."/>
```

这样，ApplicationContext容器在启动的时候，只会默认实例化not-lazy-init-bean而不会实例化lazy-init-bean。

当然，仅指定lazy-init-bean的lazy-init为true，并不意味着容器就一定会延迟初始化该bean的实例。如果某个非延迟初始化的bean定义依赖于lazy-init-bean，那么毫无疑问，按照依赖决计的顺序，容器还是会首先实例化lazy-init-bean，然后再实例化后者，如下代码演示了这种相互牵连导致延迟初始化失败的情况：

```
<bean id="lazy-init-bean" class="..." lazy-init="true"/>

<bean id="not-lazy-init-bean" class="...">
   <property name="propName">
      <ref bean="lazy-init-bean"/>
   </property>
</bean>
```

虽然lazy-init-bean是延迟初始化的，但因为依赖它的not-lazy-init-bean并不是延迟初始化，所以lazy-init-bean还是会被提前初始化，延迟初始化的良好打算“泡汤”。如果我们真想保证lazy-init-bean一定会被延迟初始化的话，就需要保证依赖于该bean定义的其他bean定义也同样设置为延迟初始化。在bean定义很多时，好像工作量也不小哦。不过不要忘了，<beans>可是所有<bean>的统领啊，让它一声令下吧！如代码清单4-24所演示的，在顶层由<beans>统一控制延迟初始化行为即可。

代码清单4-24 通过<beans>设置统一的延迟初始化行为

```
<beans default-lazy-init="true">
   <bean id="lazy-init-bean" class="..."/>

   <bean id="not-lazy-init-bean" class="...">
      <property name="propName">
         <ref bean="lazy-init-bean"/>
      </property>
   </bean>
   ...
</beans>
```

① 对于singleton的概念，参考4.3.5节。

这样我们就不用每个<bean>都设置一遍，省事儿多了不是吗？

4.3.4　继承？我也会！

除了单独存在的bean以及多个bean之间的横向依赖关系，我们也不能忽略“纵向上”各个bean之间的关系。确切来讲，我其实是想说“类之间的继承关系”。不可否认，继承可是在面向对象界声名远扬啊。

假设我们某一天真的需要对FXNewsProvider使用继承进行扩展，那么可能会声明如下代码所示的子类定义：

```
class SpecificFXNewsProvider extends FXNewsProvider
{
   private IFXNewsListener  newsListener;
   private IFXNewsPersister newPersistener;
   ...
}
```

实际上，我们想让该子类与FXNewsProvider使用相同的IFXNewsPersister，即DowJonesNewsPersister，那么可以使用如代码清单4-25所示的配置。

代码清单4-25　使用同一个IFXNewsPersister依赖对象的FXNewsProvider和SpecificFXNewsProvider配置内容

```
<bean id="superNewsProvider" class="..FXNewsProvider">
   <property name="newsListener">
      <ref bean="djNewsListener"/>
   </property>
   <property name="newPersistener">
      <ref bean="djNewsPersister"/>
   </property>
</bean>

<bean id="subNewsProvider" class="..SpecificFXNewsProvider">
   <property name="newsListener">
      <ref bean="specificNewsListener"/>
   </property>
   <property name="newPersistener">
      <ref bean="djNewsPersister"/>
   </property>
</bean>
```

但实际上，这种配置存在冗余，而且也没有表现两者之间的纵向关系。所以，我们可以引入XML中的bean的“继承”配置，见代码清单4-26。

代码清单4-26　使用继承关系配置的FXNewsProvider和SpecificFXNewsProvider

```
<bean id="superNewsProvider" class="..FXNewsProvider">
   <property name="newsListener">
      <ref bean="djNewsListener"/>
   </property>
   <property name="newPersistener">
      <ref bean="djNewsPersister"/>
   </property>
</bean>

<bean id="subNewsProvider" parent="superNewsProvider"
   class="..SpecificFXNewsProvider">
   <property name="newsListener">
```

```
      <ref bean="specificNewsListener"/>
    </property>
  </bean>
```

我们在声明subNewsProvider的时候，使用了parent属性，将其值指定为superNewsProvider，这样就继承了superNewsProvider定义的默认值，只需要将特定的属性进行更改，而不要全部又重新定义一遍。

parent属性还可以与abstract属性结合使用，达到将相应bean定义模板化的目的。比如，我们还可以像代码清单4-27所演示的这样声明以上类定义。

代码清单4-27 使用模板化配置形式配置FXNewsProvider和SpecificFXNewsProvider

```
<bean id="newsProviderTemplate" abstract="true">
  <property name="newPersistener">
    <ref bean="djNewsPersister"/>
  </property>
</bean>

<bean id="superNewsProvider" parent="newsProviderTemplate"
  class="..FXNewsProvider">
  <property name="newsListener">
    <ref bean="djNewsListener"/>
  </property>
</bean>

<bean id="subNewsProvider" parent="newsProviderTemplate"
  class="..SpecificFXNewsProvider">
  <property name="newsListener">
    <ref bean="specificNewsListener"/>
  </property>
</bean>
```

newsProviderTemplate的bean定义通过abstract属性声明为true，说明这个bean定义不需要实例化。实际上，这就是之前提到的可以不指定class属性的少数场景之一（当然，同时指定class和abstract="true"也是可以的）。该bean定义只是一个配置模板，不对应任何对象。superNews-Provider和subNewsProvider通过parent指向这个模板定义，就拥有了该模板定义的所有属性配置。当多个bean定义拥有多个相同默认属性配置的时候，你会发现这种方式可以带来很大的便利。

另外，既然这里提到abstract，对它就多说几句。容器在初始化对象实例的时候，不会关注将abstract属性声明为true的bean定义。如果你不想容器在初始化的时候实例化某些对象，那么可以将其abstract属性赋值true，以避免容器将其实例化。对于ApplicationContext容器尤其如此，因为默认情况下，ApplicationContext会在容器启动的时候就对其管理的所有bean进行实例化，只有标志为abstract的bean除外。

4.3.5 bean 的 scope

BeanFactory除了拥有作为IoC Service Provider的职责，作为一个轻量级容器，它还有着其他一些职责，其中就包括对象的生命周期管理。

本节主要讲述容器中管理的对象的scope这个概念。多数中文资料在讲解bean的scope时喜欢用“作用域”这个名词，应该还算贴切吧。不过，我更希望告诉你scope这个词到底代表什么意思，至于你怎么称呼它反而不重要。

scope用来声明容器中的对象所应该处的限定场景或者说该对象的存活时间，即容器在对象进入其相应的scope之前，生成并装配这些对象，在该对象不再处于这些scope的限定之后，容器通常会销毁

这些对象。打个比方吧！我们都是处于社会（容器）中，如果把中学教师作为一个类定义，那么当容器初始化这些类之后，中学教师只能局限在中学这样的场景中；中学，就可以看作中学教师的scope。

Spring容器最初提供了两种bean的scope类型：singleton和prototype，但发布2.0之后，又引入了另外三种scope类型，即request、session和global session类型。不过这三种类型有所限制，只能在Web应用中使用。也就是说，只有在支持Web应用的ApplicationContext中使用这三个scope才是合理的。

我们可以通过使用<bean>的singleton或者scope属性来指定相应对象的scope，其中，scope属性只能在XSD格式的文档声明中使用，类似于如下代码所演示的形式：

```
DTD:
<bean id="mockObject1" class="...MockBusinessObject" singleton="false"/>
XSD:
<bean id="mockObject2" class="...MockBusinessObject" scope="prototype"/>
```

让我们来看一下容器提供的这几个scope是如何限定相应对象的吧！

1. singleton

配置中的bean定义可以看作是一个模板，容器会根据这个模板来构造对象。但是要根据这个模板构造多少对象实例，又该让这些构造完的对象实例存活多久，则由容器根据bean定义的scope语意来决定。标记为拥有singleton scope的对象定义，在Spring的IoC容器中只存在一个实例，所有对该对象的引用将共享这个实例。该实例从容器启动，并因为第一次被请求而初始化之后，将一直存活到容器退出，也就是说，它与IoC容器“几乎”拥有相同的“寿命”。

图4-5是Spring参考文档中所给出的singleton的bean的实例化和注入语意演示图例，或许可以更形象地说明问题。

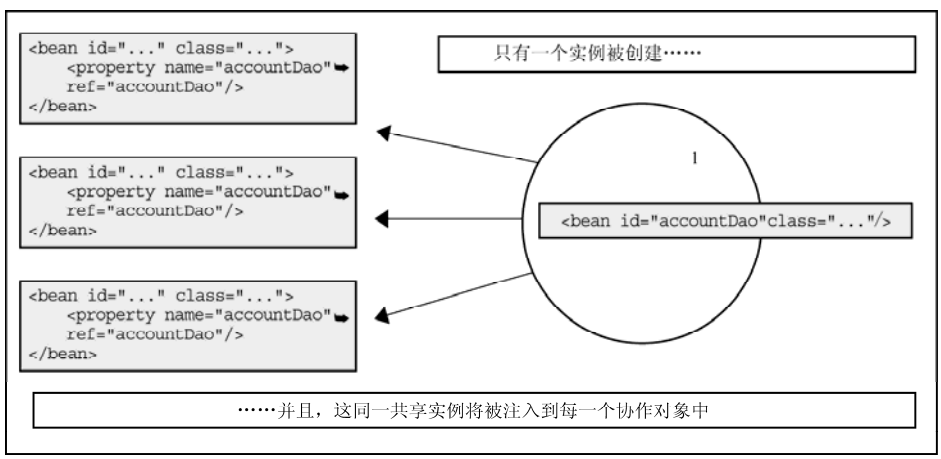


图4-5　singleton scope

需要注意的一点是，不要因为名字的原因而与GoF[①]所提出的Singleton模式相混淆，二者的语意是不同的：标记为singleton的bean是由容器来保证这种类型的bean在同一个容器中只存在一个共享实例；而Singleton模式则是保证在同一个Classloader中只存在一个这种类型的实例。

可以从两个方面来看待singleton的bean所具有的特性。

- **对象实例数量**。singleton类型的bean定义，在一个容器中只存在一个共享实例，所有对该类型

① Gang of Four，*Design Patterns: Elements of Reusable Object Software*一书的四位作者的称号。

bean的依赖都引用这一单一实例。这就好像每个幼儿园都会有一个滑梯一样，这个幼儿园的小朋友共同使用这一个滑梯。而对于该幼儿园容器来说，滑梯实际上就是一个singleton的bean。

- **对象存活时间**。singleton类型bean定义，从容器启动，到它第一次被请求而实例化开始，只要容器不销毁或者退出，该类型bean的单一实例就会一直存活。

通常情况下，如果你不指定bean的scope，singleton便是容器默认的scope，所以，下面三种配置形式实际上达成的是同样的效果：

```
<!-- DTD or XSD -->
<bean id="mockObject1" class="...MockBusinessObject"/>
<!-- DTD -->
<bean id="mockObject1" class="...MockBusinessObject" singleton="true"/>
<!-- XSD -->
<bean id="mockObject1" class="...MockBusinessObject" scope="singleton"/>
```

2. prototype

针对声明为拥有prototype scope的bean定义，容器在接到该类型对象的请求的时候，会每次都重新生成一个新的对象实例给请求方。虽然这种类型的对象的实例化以及属性设置等工作都是由容器负责的，但是只要准备完毕，并且对象实例返回给请求方之后，容器就不再拥有当前返回对象的引用，请求方需要自己负责当前返回对象的后继生命周期的管理工作，包括该对象的销毁。也就是说，容器每次返回给请求方一个新的对象实例之后，就任由这个对象实例“自生自灭”了。

让我们继续幼儿园的比喻，看看prototype在这里应该映射到哪些事物。儿歌里好像有句“排排坐，分果果”，我们今天要分苹果咯！将苹果的bean定义的scope声明为prototype，在每个小朋友领取苹果的时候，我们都是分发一个新的苹果给他。发完之后，小朋友爱怎么吃怎么吃，爱什么时候吃什么时候吃。但是，吃完后要记得把果核扔到果皮箱哦！ 而如果你把苹果的bean定义的scope声明为singleton会是什么情况呢？如果第一个小朋友比较谦让，那么他可能对这个苹果只咬一口，但是下一个小朋友吃多少就不知道了。当吃得只剩一个果核的时候，下一个来吃苹果的小朋友肯定要哭鼻子的。

所以，对于那些请求方不能共享使用的对象类型，应该将其bean定义的scope设置为prototype。这样，每个请求方可以得到自己对应的一个对象实例，而不会出现上面“哭鼻子”的现象。通常，声明为prototype的scope的bean定义类型，都是一些有状态的，比如保存每个顾客信息的对象。

从Spring 参考文档上的这幅图片（见图4-6），你可以再次了解一下拥有prototype scope的bean定义，在实例化对象并注入依赖的时候，它的具体语意是个什么样子。

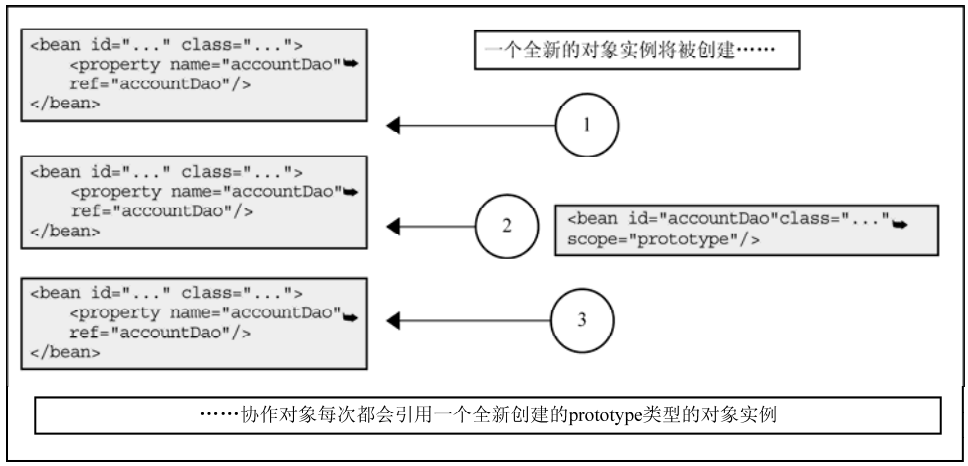


图4-6　prototype scope

你用以下形式来指定某个bean定义的scope为prototype类型，效果是一样的：

```
<!-- DTD -->
<bean id="mockObject1" class="...MockBusinessObject" singleton="false"/>
<!-- XSD -->
<bean id="mockObject1" class="...MockBusinessObject" scope="prototype"/>
```

3. request、session和global session

这三个scope类型是Spirng 2.0之后新增加的，它们不像之前的singleton和prototype那么“通用”，因为它们只适用于Web应用程序，通常是与XmlWebApplicationContext共同使用，而这些将在第6部分详细讨论。不过，既然它们也属于scope的概念，这里就简单提几句。

注意　只能使用scope 属性才能指定这三种“bean的scope类型”。也就是说，你不得不使用基于XSD文档声明的XML配置文件格式。

- request

request通常的配置形式如下：

```
<bean id="requestProcessor" class="...RequestProcessor" scope="request"/>
```

Spring 容器，即XmlWebApplicationContext会为每个HTTP请求创建一个全新的RequestProcessor对象供当前请求使用，当请求结束后，该对象实例的生命周期即告结束。当同时有10个HTTP请求进来的时候，容器会分别针对这10个请求返回10个全新的RequestProcessor对象实例，且它们之间互不干扰。从不是很严格的意义上说，request可以看作prototype的一种特例，除了场景更加具体之外，语意上差不多。

- session

对于Web应用来说，放到session中的最普遍的信息就是用户的登录信息，对于这种放到session中的信息，我们可使用如下形式指定其scope为session：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

Spring容器会为每个独立的session创建属于它们自己的全新的UserPreferences对象实例。与

request相比，除了拥有session scope的bean的实例具有比request scope的bean可能更长的存活时间，其他方面真是没什么差别。

- global session

还是userPreferences，不过scope对应的值换一下，如下所示：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

global session只有应用在基于portlet的Web应用程序中才有意义，它映射到portlet的global范围的session。如果在普通的基于servlet的Web应用中使用了这个类型的scope，容器会将其作为普通的session类型的scope对待。

4. 自定义scope类型

在Spring 2.0之后的版本中，容器提供了对scope的扩展点，这样，你可以根据自己的需要或者应用的场景，来添加自定义的scope类型。需要说明的是，默认的singleton和prototype是硬编码到代码中的，而request、session和global session，包括自定义scope类型，则属于可扩展的scope行列，它们都实现了org.springframework.beans.factory.config.Scope接口，该接口定义如下：

```
public interface Scope {

  Object get(String name, ObjectFactory objectFactory);

  Object remove(String name);

  void registerDestructionCallback(String name, Runnable callback);

  String getConversationId();
}
```

要实现自己的scope类型，首先需要给出一个Scope接口的实现类，接口定义中的4个方法并非都是必须的，但get和remove方法必须实现。我们可以看一下http://www.jroller.com/eu/entry/implementing_efficinet_id_generator中提到的一个ThreadScope的实现（见代码清单4-28）。

代码清单4-28 自定义的ThreadScope的定义

```
public class ThreadScope implements Scope {

  private final ThreadLocal threadScope = new ThreadLocal() {
      protected Object initialValue() {
        return new HashMap();
      }
    };

  public Object get(String name, ObjectFactory objectFactory) {
    Map scope = (Map) threadScope.get();
    Object object = scope.get(name);
    if(object==null) {
      object = objectFactory.getObject();
      scope.put(name, object);
    }
    return object;
  }

  public Object remove(String name) {
    Map scope = (Map) threadScope.get();
    return scope.remove(name);
  }
```

```
    public void registerDestructionCallback(String name, Runnable callback) {
    }
    ...
}
```

更多Scope相关的实例，可以参照同一站点的一篇文章“More fun with Spring scopes”（http://jroller.com/eu/entry/more_fun_with_spring_scopes），其中提到PageScope的实现。

有了Scope的实现类之后，我们需要把这个Scope注册到容器中，才能供相应的bean定义使用。通常情况下，我们可以使用ConfigurableBeanFactory的以下方法注册自定义scope：

```
void registerScope(String scopeName, Scope scope);
```

其中，参数scopeName就是使用的bean定义可以指定的名称，比如Spring框架默认提供的自定义scope类型request或者session。参数scope即我们提供的Scope实现类实例。

对于以上的ThreadScope，如果容器为BeanFactory类型（当然，更应该实现ConfigurableBeanFactory），我们可以通过如下方式来注册该Scope：

```
Scope threadScope = new ThreadScope();
beanFactory.registerScope("thread",threadScope);
```

之后，我们就可以在需要的bean定义中直接通过“thread”名称来指定该bean定义对应的scope为以上注册的ThreadScope了，如以下代码所示：

```
<bean id="beanName" class="..." scope="thread"/>
```

除了直接编码调用ConfigurableBeanFactory的registerScope来注册scope，Spring还提供了一个专门用于统一注册自定义scope的BeanFactoryPostProcessor实现（有关BeanFactoryPostProcessor的更多细节稍后将详述），即org.springframework.beans.factory.config.CustomScopeConfigurer。对于ApplicationContext来说，因为它可以自动识别并加载BeanFactoryPostProcessor，所以我们就可以直接在配置文件中，通过这个CustomScopeConfigurer注册来ThreadScope（如代码清单4-29所示）。

代码清单4-29　使用CustomScopeConfigurer注册自定义scope

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="thread" value="com.foo.ThreadScope"/>
    </map>
  </property>
</bean>
```

在以上工作全部完成之后，我们就可以在自己的bean定义中使用这个新增加到容器的自定义scope“thread”了，如下代码演示了通常情况下“thread”自定义scope的使用：

```
<bean id="beanName" class="..." scope="thread">
  <aop:scoped-proxy/>
</bean>
```

由于<aop:scoped-proxy/>涉及Spring AOP相关知识，这里不会详细讲述。需要注意的是，使用了自定义scope的bean定义，需要该元素来为其在合适的时间创建和销毁相应的代理对象实例。对于request、session和global session来说，也是如此。

4.3.6　工厂方法与 FactoryBean

在强调“面向接口编程”的同时，有一点需要注意：虽然对象可以通过声明接口来避免对特定接口实现类的过度耦合，但总归需要一种方式将声明依赖接口的对象与接口实现类关联起来。否则，只依赖一个不做任何事情的接口是没有任何用处的。假设我们有一个像代码清单4-30所声明的Foo类，它声明了一个BarInterface依赖。

代码清单4-30 依赖于某一BarInterface接口的Foo类定义

```
public class Foo
{
   private BarInterface barInstance;
   public Foo()
   {
      // 我们应该避免这样做
      // instance = new BarInterfaceImpl();
   }
   // ...
}
```

如果该类是由我们设计并开发的，那么还好说，我们可以通过依赖注入，让容器帮助我们解除接口与实现类之间的耦合性。但是，有时，我们需要依赖第三方库，需要实例化并使用第三方库中的相关类，这时，接口与实现类的耦合性需要其他方式来避免。

通常的做法是通过使用工厂方法（Factory Method）模式，提供一个工厂类来实例化具体的接口实现类，这样，主体对象只需要依赖工厂类，具体使用的实现类有变更的话，只是变更工厂类，而主体对象不需要做任何变动。代码清单4-31演示了这种做法。

代码清单4-31 使用了工厂方法模式的Foo类可能定义

```
public class Foo
{
   private BarInterface barInterface;
   public Foo()
   {
      // barInterface = BarInterfaceFactory.getInstance();
      // 或者
      // barInterface = new BarInterfaceFactory().getInstance();
   }
   ...
}
```

针对使用工厂方法模式实例化对象的方式，Spring的IoC容器同样提供了对应的集成支持。我们所要做的，只是将工厂类所返回的具体的接口实现类注入给主体对象（这里是Foo）。

注意 有关工厂方法模式的信息，可以参考设计模式方面的书籍或者网上有关资源。

1. 静态工厂方法（Static Factory Method）

假设某个第三方库发布了BarInterface，为了向使用该接口的客户端对象屏蔽以后可能对BarInterface实现类的变动，同时还提供了一个静态的工厂方法实现类StaticBarInterface-Factory，代码如下：

```
public class StaticBarInterfaceFactory
{
   public static BarInterface getInstance()
   {
```

```
        return new BarInterfaceImpl();
    }
}
```

为了将该静态工厂方法类返回的实现注入Foo，我们使用以下方式进行配置（通过setter方法注入方式为Foo注入BarInterface的实例）：

```
<bean id="foo" class="...Foo">
    <property name="barInterface">
        <ref bean="bar"/>
    </property>
</bean>

<bean id="bar" class="...StaticBarInterfaceFactory" factory-method="getInstance"/>
```

class指定静态方法工厂类，factory-method指定工厂方法名称，然后，容器调用该静态方法工厂类的指定工厂方法（getInstance），并返回方法调用后的结果，即BarInterfaceImpl的实例。也就是说，为foo注入的bar实际上是BarInterfaceImpl的实例，即方法调用后的结果，而不是静态工厂方法类（StaticBarInterfaceFactory）。我们可以实现自己的静态工厂方法类返回任意类型的对象实例，但工厂方法类的类型与工厂方法返回的类型没有必然的相同关系。

某些时候，有的工厂类的工厂方法可能需要参数来返回相应实例，而不一定非要像我们的getInstance()这样没有任何参数。对于这种情况，可以通过<constructor-arg>来指定工厂方法需要的参数，比如现在StaticBarInterfaceFactory需要其他依赖来返回某个BarInterface的实现，其定义可能如下：

```
public class StaticBarInterfaceFactory
{
    public static BarInterface getInstance(Foobar foobar)
    {
        return new BarInterfaceImpl(foobar);
    }
}
```

为了让包含方法参数的工厂方法能够预期返回相应的实现类实例，我们可以像代码清单4-32所演示的那样，通过<constructor-arg>为工厂方法传入相应参数。

代码清单4-32 使用<constructor-arg>调用含有参数的工厂方法

```
<bean id="foo" class="...Foo">
    <property name="barInterface">
        <ref bean="bar"/>
    </property>
</bean>

<bean id="bar" class="...StaticBarInterfaceFactory" factory-method="getInstance">
    <constructor-arg>
        <ref bean="foobar"/>
    </constructor-arg>
</bean>

<bean id="foobar" class="...FooBar"/>
```

唯一需要注意的就是，针对静态工厂方法实现类的bean定义，使用<constructor-arg>传入的是工厂方法的参数，而不是静态工厂方法实现类的构造方法的参数。（况且，静态工厂方法实现类也没有提供显式的构造方法。）

2. 非静态工厂方法（Instance Factory Method）

既然可以将静态工厂方法实现类的工厂方法调用结果作为bean注册到容器中，我们同样可以针对基于工厂类实例的工厂方法调用结果应用相同的功能，只不过，表达方式可能需要稍微变一下。

现在为BarInterface提供非静态的工厂方法实现类，该类定义如下代码所示：

```
public class NonStaticBarInterfaceFactory
{
   public BarInterface getInstance()
   {
     return new BarInterfaceImpl();
   }
   ...
}
```

因为工厂方法为非静态的，我们只能通过某个NonStaticBarInterfaceFactory实例来调用该方法（哦，错了，是容器来调用），那么也就有了如下的配置内容：

```
<bean id="foo" class="...Foo">
   <property name="barInterface">
     <ref bean="bar"/>
   </property>
</bean>

<bean id="barFactory" class="...NonStaticBarInterfaceFactory"/>

<bean id="bar" factory-bean="barFactory" factory-method="getInstance"/>
```

NonStaticBarInterfaceFactory是作为正常的bean注册到容器的，而bar的定义则与静态工厂方法的定义有些不同。现在使用factory-bean属性来指定工厂方法所在的工厂类实例，而不是通过class属性来指定工厂方法所在类的类型。指定工厂方法名则相同，都是通过factory-method属性进行的。

如果非静态工厂方法调用时也需要提供参数的话，处理方式是与静态的工厂方法相似的，都可以通过<constructor-arg>来指定方法调用参数。

3. FactoryBean

FactoryBean是Spring容器提供的一种可以扩展容器对象实例化逻辑的接口，请不要将其与容器名称BeanFactory相混淆。FactoryBean，其主语是Bean，定语为Factory，也就是说，它本身与其他注册到容器的对象一样，只是一个Bean而已，只不过，这种类型的Bean本身就是生产对象的工厂（Factory）。

当某些对象的实例化过程过于烦琐，通过XML配置过于复杂，使我们宁愿使用Java代码来完成这个实例化过程的时候，或者，某些第三方库不能直接注册到Spring容器的时候，就可以实现org.springframework.beans.factory.FactoryBean接口，给出自己的对象实例化逻辑代码。当然，不使用FactoryBean，而像通常那样实现自定义的工厂方法类也是可以的。不过，FactoryBean可是Spring提供的对付这种情况的“制式装备”[①]哦！

要实现并使用自己的FactoryBean其实很简单，org.springframework.beans.factory.FactoryBean只定义了三个方法，如以下代码所示：

```
public interface FactoryBean {
```

① 制式装备通常指正规军使用的标准装备。

```
    Object getObject() throws Exception;
    Class getObjectType();
    boolean isSingleton();
}
```

getObject()方法会返回该FactoryBean“生产”的对象实例，我们需要实现该方法以给出自己的对象实例化逻辑；getObjectType()方法仅返回getObject()方法所返回的对象的类型，如果预先无法确定，则返回null；isSingleton()方法返回结果用于表明，工厂方法（getObject()）所“生产”的对象是否要以singleton形式存在于容器中。如果以singleton形式存在，则返回true，否则返回false；

如果我们想每次得到的日期都是第二天，可以实现一个如代码清单4-33所示的FactoryBean。

代码清单4-33 NextDayDateFactoryBean的定义代码

```
import org.joda.time.DateTime;
import org.springframework.beans.factory.FactoryBean;

public class NextDayDateFactoryBean implements FactoryBean {

    public Object getObject() throws Exception {
        return new DateTime().plusDays(1);
    }

    public Class getObjectType() {
        return DateTime.class;
    }

    public boolean isSingleton() {
        return false;
    }

}
```

很简单的实现，不是嘛？

要使用NextDayDateFactoryBean，只需要如下这样将其注册到容器即可：

```
<bean id="nextDayDateDisplayer" class="...NextDayDateDisplayer">
    <property name="dateOfNextDay">
        <ref bean="nextDayDate"/>
    </property>
</bean>

<bean id="nextDayDate" class="...NextDayDateFactoryBean">
</bean>
```

配置上看不出与平常的bean定义有何不同，不过，只有当我们看到NextDayDateDisplayer的定义的时候，才会知道FactoryBean的魔力到底在哪。NextDayDateDisplayer的定义如下：

```
public class NextDayDateDisplayer
{
    private DateTime dateOfNextDay;
    // 相应的setter方法
    // ...
}
```

看到了嘛？NextDayDateDisplayer所声明的依赖dateOfNextDay的类型为DateTime，而不是NextDayDateFactoryBean。也就是说FactoryBean类型的bean定义，通过正常的id引用，容器返回

的是FactoryBean所“生产”的对象类型，而非FactoryBean实现本身。

如果一定要取得FactoryBean本身的话，可以通过在bean定义的id之前加前缀&来达到目的。代码清单4-34展示了获取FactoryBean本身与获取FactoryBean“生产”的对象之间的差别。

代码清单4-34 使用&获取FactoryBean的实例演示

```
Object nextDayDate = container.getBean("nextDayDate");
assertTrue(nextDayDate instanceof DateTime);

Object factoryBean = container.getBean("&nextDayDate");
assertTrue(factoryBean instanceof FactoryBean);
assertTrue(factoryBean instanceof NextDayDateFactoryBean);

Object factoryValue = ((FactoryBean)factoryBean).getObject();
assertTrue(factoryValue instanceof DateTime);

assertNotSame(nextDayDate, factoryValue);
assertEquals(((DateTime)nextDayDate).getDayOfYear(),((DateTime)factoryValue).getDayOfYear());
```

Spring容器内部许多地方了使用FactoryBean。下面是一些比较常见的FactoryBean实现，你可以参照FactoryBean的Javadoc以了解更多内容。

- JndiObjectFactoryBean
- LocalSessionFactoryBean
- SqlMapClientFactoryBean
- ProxyFactoryBean
- TransactionProxyFactoryBean

4.3.7 偷梁换柱之术

在学习以下内容之前，先提一下有关bean的scope的使用“陷阱”，特别是prototype在容器中的使用，以此引出本节将要介绍的Spring容器较为独特的功能特性：方法注入（Method Injection）以及方法替换（Method Replacement）。

我们知道，拥有prototype类型scope的bean，在请求方每次向容器请求该类型对象的时候，容器都会返回一个全新的该对象实例。为了简化问题的叙述，我们直接将FX News系统中的FXNewsBean定义注册到容器中，并将其scope设置为prototype。因为它是有状态的类型，每条新闻都应该是新的独立个体；同时，我们给出MockNewsPersister类，使其实现IFXNewsPersister接口，以模拟注入FXNewsBean实例后的情况。这样，我们就有了代码清单4-35所展示的类声明和相关配置。

代码清单4-35 MockNewsPersister的定义以及相关配置

```
public class MockNewsPersister implements IFXNewsPersister {
  private FXNewsBean newsBean;

  public void persistNews(FXNewsBean bean) {
    persistNewes();
  }
  public void persistNews()
  {
    System.out.println("persist bean:"+getNewsBean());
  }
  public FXNewsBean getNewsBean() {
    return newsBean;
  }
```

```
    public void setNewsBean(FXNewsBean newsBean) {
      this.newsBean = newsBean;
    }
}
配置为
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
   <property name="newsBean">
     <ref bean="newsBean"/>
   </property>
</bean>
```

当多次调用MockNewsPersister的persistNews时，你猜会得到什么结果？如下代码可以帮助我们揭开答案：

```
BeanFactory container = new XmlBeanFactory(new ClassPathResource(".."));
MockNewsPersister persister = (MockNewsPersister)container.getBean("mockPersister");
persister.persistNews();
persister.persistNews();
输出:
persist bean:..domain.FXNewsBean@1662dc8
persist bean:..domain.FXNewsBean@1662dc8
```

从输出看，对象实例是相同的，而这与我们的初衷是相悖的。因为每次调用persistNews都会调用getNewsBean()方法并返回一个FXNewsBean实例，而FXNewsBean实例是prototype类型的，因此每次不是应该输出不同的对象实例嘛？

好了，问题实际上不是出在FXNewsBean的scope类型是否是prototype的，而是出在实例的取得方式上面。虽然FXNewsBean拥有prototype类型的scope，但当容器将一个FXNewsBean的实例注入MockNewsPersister之后，MockNewsPersister就会一直持有这个FXNewsBean实例的引用。虽然每次输出都调用了getNewsBean()方法并返回一个FXNewsBean的实例，但实际上每次返回的都是MockNewsPersister持有的容器第一次注入的实例。这就是问题之所在。换句话说，第一个实例注入后，MockNewsPersister再也没有重新向容器申请新的实例。所以，容器也不会重新为其注入新的FXNewsBean类型的实例。

知道原因之后，我们就可以解决这个问题了。解决问题的关键在于保证getNewsBean()方法每次从容器中取得新的FXNewsBean实例，而不是每次都返回其持有的单一实例。

1. 方法注入

Spring容器提出了一种叫做方法注入（Method Injection）的方式，可以帮助我们解决上述问题。我们所要做的很简单，只要让getNewsBean方法声明符合规定的格式，并在配置文件中通知容器，当该方法被调用的时候，每次返回指定类型的对象实例即可。方法声明需要符合的规格定义如下：

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

也就是说，该方法必须能够被子类实现或者覆写，因为容器会为我们要进行方法注入的对象使用Cglib动态生成一个子类实现，从而替代当前对象。既然我们的getNewsBean()方法已经满足以上方法声明格式，剩下唯一要做的就是配置该类，配置内容如下所示：

```
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
   <lookup-method name="getNewsBean" bean="newsBean"/>
</bean>
```

通过<lookup-method>的name属性指定需要注入的方法名，bean属性指定需要注入的对象，当getNewsBean方法被调用的时候，容器可以每次返回一个新的FXNewsBean类型的实例。所以，这个时候，我们再次检查执行结果，输出的实例引用应该是不同的：

```
persist bean:..domain.FXNewsBean@18aaa1e
persist bean:..domain.FXNewsBean@a6aeed
```

哇噢，很帅不是吗？

注意 FXNewsBean的取得实际上可以在相应方法中按需要自行实例化，而不一定非要注册到容器中，从容器中获取。我们只是为了引入prototype的使用“陷阱”以及方法注入功能，才将FXNewsBean以prototype类型注册到容器中供使用。当然，如果愿意你也可以以这种方式使用。在最终输出的结果中，对象引用的数字不一定就是上面的那样。因为每次注入的实例是不同的，所以对应实例的数字也可能不同。在此只需要关注每次同时输出的结果是否相同即可说明问题。

2. 殊途同归

除了使用方法注入来达到“每次调用都让容器返回新的对象实例”的目的，还可以使用其他方式达到相同的目的。下面给出其他两种解决类似问题的方法，供读者参考。

- 使用BeanFactoryAware接口

我们知道，即使没有方法注入，只要在实现getNewsBean()方法的时候，能够保证每次调用BeanFactory的getBean("newsBean")，就同样可以每次都取得新的FXNewsBean对象实例。现在，我们唯一需要的，就是让MockNewsPersister拥有一个BeanFactory的引用。

Spring框架提供了一个BeanFactoryAware接口，容器在实例化实现了该接口的bean定义的过程中，会自动将容器本身注入该bean。这样，该bean就持有了它所处的BeanFactory的引用。BeanFactoryAware的定义如下代码所示：

```
public interface BeanFactoryAware {
   void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

我们让MockNewsPersister实现该接口以持有其所处的BeanFactory的引用，这样MockNewsPersister的定义如代码清单4-36所示。

代码清单4-36 实现了BeanFactoryAware接口的MockNewsPersister及相关配置

```
public class MockNewsPersister implements IFXNewsPersister,BeanFactoryAware {
   private BeanFactory beanFactory;

   public void setBeanFactory(BeanFactory bf) throws BeansException {
      this.beanFactory = bf;
   }
   public void persistNews(FXNewsBean bean) {
      persistNews();
   }
   public void persistNews()
   {
      System.out.println("persist bean:"+getNewsBean());
   }
   public FXNewsBean getNewsBean() {
      return beanFactory.getBean("newsBean");
```

```
    }
}
配置简化为
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
</bean>
```

如此，可以预见到，输出的结果将与我们所预期的相同：

```
persist bean:..domain.FXNewsBean@121cc40
persist bean:..domain.FXNewsBean@1e893df
```

实际上，方法注入动态生成的子类，完成的是与以上类似的逻辑，只不过实现细节上不同而已。

- 使用ObjectFactoryCreatingFactoryBean

ObjectFactoryCreatingFactoryBean是Spring提供的一个FactoryBean实现，它返回一个ObjectFactory实例。从ObjectFactoryCreatingFactoryBean返回的这个ObjectFactory实例可以为我们返回容器管理的相关对象。实际上，ObjectFactoryCreatingFactoryBean实现了BeanFactoryAware接口，它返回的ObjectFactory实例只是特定于与Spring容器进行交互的一个实现而已。使用它的好处就是，隔离了客户端对象对BeanFactory的直接引用。

现在，我们使用ObjectFactory取得FXNewsBean的实例，代码清单4-37给出了对应这种方式的MockNewsPersister实现声明。

代码清单4-37　使用ObjectFactory的MockNewsPersister定义

```
public class MockNewsPersister implements IFXNewsPersister {
   private ObjectFactory newsBeanFactory;

   public void persistNews(FXNewsBean bean) {
      persistNews();
   }
   public void persistNews()
   {
      System.out.println("persist bean:"+getNewsBean());
   }
   public FXNewsBean getNewsBean() {
      return newsBeanFactory.getObject();
   }
   public void setNewsBeanFactory(ObjectFactory newsBeanFactory) {
      this.newsBeanFactory = newsBeanFactory;
   }
}
```

有了以上的类定义之后，我们应该为MockNewsPersister注入相应的ObjectFactory，这也正是ObjectFactoryCreatingFactoryBean闪亮登场的时候，代码清单4-38给出了对应的配置内容。

代码清单4-38　使用ObjectFactoryCreatingFactoryBean的相关配置

```
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="newsBeanFactory" class="org.springframework.beans.factory.config. ➥
ObjectFactoryCreatingFactoryBean">
   <property name="targetBeanName">
      <idref bean="newsBean"/>
   </property>
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
```

```
    <property name="newsBeanFactory">
      <ref bean="newsBeanFactory"/>
    </property>
</bean>
```

看，真有效！

```
persist bean:..domain.FXNewsBean@ecd7e
persist bean:..domain.FXNewsBean@1d520c4
```

提示 也可以使用ServiceLocatorFactoryBean来代替ObjectFactoryCreatingFactoryBean，该FactoryBean可以让我们自定义工厂接口，而不用非要使用Spring的ObjectFactory。可以参照该类定义的Javadoc取得更多信息，Javadoc中有详细的实例，足够让你了解该类的使用和功能。

3. 方法替换

与方法注入只是通过相应方法为主体对象注入依赖对象不同，方法替换更多体现在方法的实现层面上，它可以灵活替换或者说以新的方法实现覆盖掉原来某个方法的实现逻辑。基本上可以认为，方法替换可以帮助我们实现简单的方法拦截功能。要知道，我们现在可是在不知不觉中迈上了AOP的大道哦！

假设某天我看FXNewsProvider不爽，想替换掉它的getAndPersistNews方法默认逻辑，这时，我就可以用方法替换将它的原有逻辑给替换掉。

小心 这里只是为了演示方法替换（Method Replacement）的功能，不要真的这么做。要使用也要用在好的地方，对吧？

首先，我们需要给出org.springframework.beans.factory.support.MethodReplacer的实现类，在这个类中实现将要替换的方法逻辑。假设我们只是简单记录日志，打印简单信息，那么就可以给出一个类似代码清单4-39所示的MethodReplacer实现类。

代码清单4-39 FXNewsProviderMethodReplacer类的定义

```
public class FXNewsProviderMethodReplacer implements MethodReplacer {

  private static final transient Log logger = ➥
  LogFactory.getLog(FXNewsProviderMethodReplacer.class);

  public Object reimplement(Object target, Method method, Object[] args) ➥
  throws Throwable {
    logger.info("before executing method["+method.getName()+➥
    "] on Object["+target.getClass().getName()+"].");

    System.out.println("sorry,We will do nothing this time.");

    logger.info("end of executing method["+method.getName()+➥
    "] on Object["+target.getClass().getName()+"].");
    return null;
  }
}
```

有了要替换的逻辑之后，我们就可以把这个逻辑通过<replaced-method>配置到FXNewsProvider的bean定义中，使其生效，配置内容如代码清单4-40所示。

代码清单4-40　FXNewsProvider中使用方法替换的相关配置

```
<bean id="djNewsProvider" class="..FXNewsProvider">
   <constructor-arg index="0">
      <ref bean="djNewsListener"/>
   </constructor-arg>
   <constructor-arg index="1">
      <ref bean="djNewsPersister"/>
   </constructor-arg>
   <replaced-method name="getAndPersistNews" replacer="providerReplacer">
   </replaced-method>
</bean>

<bean id="providerReplacer" class="..FXNewsProviderMethodReplacer">
</bean>

<!--其他bean配置-->
...
```

现在，你猜调用FXNewsProvider的getAndPersistNews方法后，会得到什么结果？输出结果如下所示：

```
771 [main] INFO ..FXNewsProviderMethodReplacer
   - before executing method[getAndPersistNews]
     on Object[..FXNewsProvider$$EnhancerByCGLIB$$3fa709d3].
sorry,We will do nothing this time.
771 [main] INFO ..FXNewsProviderMethodReplacer
   - end of executing method[getAndPersistNews]
     on Object[..FXNewsProvider$$EnhancerByCGLIB$$3fa709d3].
```

我们把FXNewsProvider的getAndPersistNews方法逻辑给完全替换掉了。现在该方法基本上什么也没做，哇……

最后需要强调的是，这种方式刚引入的时候执行效率不是很高。而且，当你充分了解并应用Spring AOP之后，我想你也不会再回头求助这个特色功能。不过，怎么说这也是一个选择，场景合适的话，为何不用呢？

哦，如果要替换的方法存在参数，或者对象存在多个重载的方法，可以在<replaced-method>内部通过<arg-type>明确指定将要替换的方法参数类型。祝“替换”愉快！

4.4　容器背后的秘密

子曰：学而不思则罔。除了了解Spring的IoC容器如何使用，了解Spring的IoC容器都提供了哪些功能，我们也应该想一下，Spring的IoC容器内部到底是如何来实现这些的呢？虽然我们不太可能“重新发明轮子”，但是，如图4-7（该图摘自Spring官方参考文档）所示的那样，只告诉你“Magic Happens Here”，你是否就能心满意足呢？

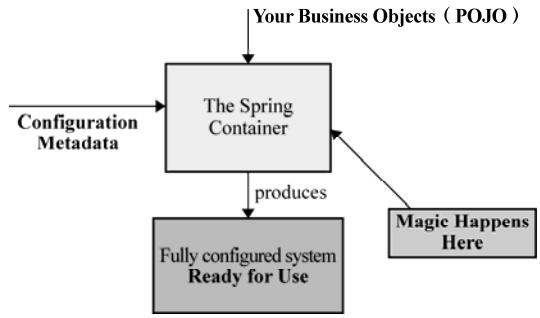


图4-7 即将揭示的奥秘所在

好，如果你的答案是“不”（我当然认为你说的是“不想一直被蒙在鼓里”），那么就随我一起来探索一下这个“黑匣子”里面到底有些什么……

4.4.1 “战略性观望”

Spring的IoC容器所起的作用，就像图4-7所展示的那样，它会以某种方式加载Configuration Metadata（通常也就是XML格式的配置信息），然后根据这些信息绑定整个系统的对象，最终组装成一个可用的基于轻量级容器的应用系统。

Spring的IoC容器实现以上功能的过程，基本上可以按照类似的流程划分为两个阶段，即容器启动阶段和Bean实例化阶段，如图4-8所示。

Spring的IoC容器在实现的时候，充分运用了这两个实现阶段的不同特点，在每个阶段都加入了相应的容器扩展点，以便我们可以根据具体场景的需要加入自定义的扩展逻辑。

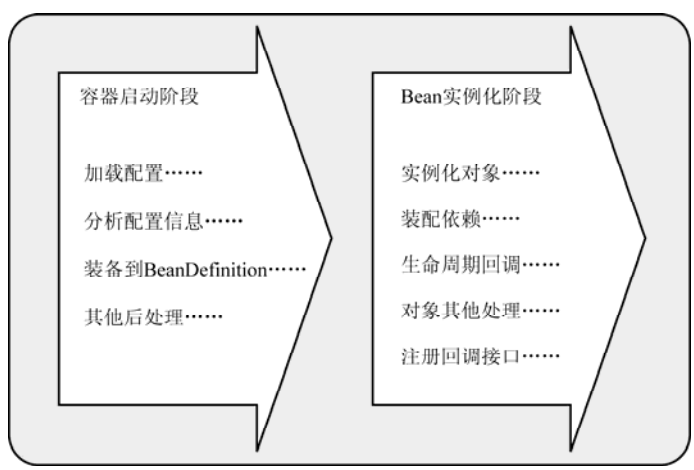


图4-8 容器功能实现的各个阶段

1. 容器启动阶段

容器启动伊始，首先会通过某种途径加载Configuration MetaData。除了代码方式比较直接，在大部分情况下，容器需要依赖某些工具类（`BeanDefinitionReader`）对加载的Configuration MetaData

进行解析和分析，并将分析后的信息编组为相应的BeanDefinition，最后把这些保存了bean定义必要信息的BeanDefinition，注册到相应的BeanDefinitionRegistry，这样容器启动工作就完成了。图4-9演示了这个阶段的主要工作。

图4-9　XML配置信息到BeanDefinition的映射

总地来说，该阶段所做的工作可以认为是准备性的，重点更加侧重于对象管理信息的收集。当然，一些验证性或者辅助性的工作也可以在这个阶段完成。

2. Bean实例化阶段

经过第一阶段，现在所有的bean定义信息都通过BeanDefinition的方式注册到了BeanDefinitionRegistry中。当某个请求方通过容器的getBean方法明确地请求某个对象，或者因依赖关系容器需要隐式地调用getBean方法时，就会触发第二阶段的活动。

该阶段，容器会首先检查所请求的对象之前是否已经初始化。如果没有，则会根据注册的BeanDefinition所提供的信息实例化被请求对象，并为其注入依赖。如果该对象实现了某些回调接口，也会根据回调接口的要求来装配它。当该对象装配完毕之后，容器会立即将其返回请求方使用。如果说第一阶段只是根据图纸装配生产线的话，那么第二阶段就是使用装配好的生产线来生产具体的产品了。

4.4.2　插手“容器的启动”

Spring提供了一种叫做BeanFactoryPostProcessor的容器扩展机制。该机制允许我们在容器实例化相应对象之前，对注册到容器的BeanDefinition所保存的信息做相应的修改。这就相当于在容器实现的第一阶段最后加入一道工序，让我们对最终的BeanDefinition做一些额外的操作，比如修改其中bean定义的某些属性，为bean定义增加其他信息等。

如果要自定义实现BeanFactoryPostProcessor，通常我们需要实现org.springframework.beans.factory.config.BeanFactoryPostProcessor接口。同时，因为一个容器可能拥有多个BeanFactoryPostProcessor，这个时候可能需要实现类同时实现Spring的org.springframework.core.Ordered接口，以保证各个BeanFactoryPostProcessor可以按照预先设定的顺序执行（如果顺序紧要的话）。但是，因为Spring已经提供了几个现成的BeanFactoryPostProcessor实现类，所以，大多时候，我们很少自己去实现某个BeanFactoryPostProcessor。其中，org.springframework.beans.factory.config.PropertyPlaceholderConfigurer和org.springframework.beans.factory.config.Property OverrideConfigurer是两个比较常用的BeanFactoryPostProcessor。另外，为了处理配置文件中的数据类型与真正的业务对象所定义的数据类型转换，Spring还允许我们通过org.springframework.beans.factory.config.CustomEditorConfigurer来注册自定义的PropertyEditor以补助容器中默认的PropertyEditor。可以参考BeanFactoryPostProcessor的Javadoc来了解更多其实现子类的情况。

我们可以通过两种方式来应用BeanFactoryPostProcessor，分别针对基本的IoC容器BeanFactory和较为先进的容器ApplicationContext。

对于BeanFactory来说，我们需要用手动方式应用所有的BeanFactoryPostProcessor，代码清单4-41演示了具体的做法。

代码清单4-41 手动装配BeanFactory使用的BeanFactoryPostProcessor

```
// 声明将被后处理的BeanFactory实例
ConfigurableListableBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("..."));
// 声明要使用的BeanFactoryPostProcessor
PropertyPlaceholderConfigurer propertyPostProcessor = new PropertyPlaceholderConfigurer();
propertyPostProcessor.setLocation(new ClassPathResource("..."));
// 执行后处理操作
propertyPostProcessor.postProcessBeanFactory(beanFactory);
```

如果拥有多个BeanFactoryPostProcessor，我们可以添加更多类似的代码来应用所有的这些BeanFactoryPostProcessor。

对于ApplicationContext来说，情况看起来要好得多。因为ApplicationContext会自动识别配置文件中的BeanFactoryPostProcessor并应用它，所以，相对于BeanFactory，在ApplicationContext中加载并应用BeanFactoryPostProcessor，仅需要在XML配置文件中将这些BeanFactoryPostProcessor简单配置一下即可。只要如代码清单4-42所示，将相应BeanFactoryPostProcessor实现类添加到配置文件，ApplicationContext将自动识别并应用它。

代码清单4-42 通过ApplicationContext使用BeanFactoryPostProcessor

```
...
<beans>
   <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
      <property name="locations">
         <list>
            <value>conf/jdbc.properties</value>
            <value>conf/mail.properties</value>
         </list>
      </property>
   </bean>
   ...
</beans>
```

下面让我们看一下Spring提供的这几个BeanFactoryPostProcessor实现都可以完成什么功能。

1. **PropertyPlaceholderConfigurer**

通常情况下，我们不想将类似于系统管理相关的信息同业务对象相关的配置信息混杂到XML配置文件中，以免部署或者维护期间因为改动繁杂的XML配置文件而出现问题。我们会将一些数据库连接信息、邮件服务器等相关信息单独配置到一个properties文件中，这样，如果因系统资源变动的话，只需要关注这些简单properties配置文件即可。

PropertyPlaceholderConfigurer允许我们在XML配置文件中使用占位符（PlaceHolder），并将这些占位符所代表的资源单独配置到简单的properties文件中来加载。以数据源的配置为例，使用了PropertyPlaceholderConfigurer之后（这里沿用代码清单4-42的配置内容），可以在XML配置文件中按照代码清单4-43所示的方式配置数据源，而不用将连接地址、用户名和密码等都配置到XML中。

代码清单4-43 使用了占位符的数据源配置

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
```

```
    <property name="url">
      <value>${jdbc.url}</value>
    </property>
    <property name="driverClassName">
      <value>${jdbc.driver}</value>
    </property>
    <property name="username">
      <value>${jdbc.username}</value>
    </property>
    <property name="password">
      <value>${jdbc.password}</value>
    </property>
    <property name="testOnBorrow">
      <value>true</value>
    </property>
    <property name="testOnReturn">
      <value>true</value>
    </property>
    <property name="testWhileIdle">
      <value>true</value>
    </property>
    <property name="minEvictableIdleTimeMillis">
      <value>180000</value>
    </property>
    <property name="timeBetweenEvictionRunsMillis">
      <value>360000</value>
    </property>
    <property name="validationQuery">
      <value>SELECT 1</value>
    </property>
    <property name="maxActive">
      <value>100</value>
    </property>
  </bean>
```

如果你使用过Ant或者Velocity等工具，就会发现${property}之类的表达很熟悉。现在，所有这些占位符所代表的资源，都放到了jdbc.properties文件中，如下所示：

```
jdbc.url=jdbc:mysql://server/MAIN?useUnicode=true&amp;characterEncoding=ms932&amp; ➥
failOverReadOnly=false&amp;roundRobinLoadBalance=true
jdbc.driver=com.mysql.jdbc.Driver
jdbc.username=your username
jdbc.password=your password
```

基本机制就是之前所说的那样。当BeanFactory在第一阶段加载完成所有配置信息时，BeanFactory中保存的对象的属性信息还只是以占位符的形式存在，如${jdbc.url}、${jdbc.driver}。当PropertyPlaceholderConfigurer作为BeanFactoryPostProcessor被应用时，它会使用properties配置文件中的配置信息来替换相应BeanDefinition中占位符所表示的属性值。这样，当进入容器实现的第二阶段实例化bean时，bean定义中的属性值就是最终替换完成的了。

PropertyPlaceholderConfigurer不单会从其配置的properties文件中加载配置项，同时还会检查Java的System类中的Properties，可以通过setSystemPropertiesMode()或者setSystemPropertiesModeName()来控制是否加载或者覆盖System相应Properties的行为。PropertyPlaceholderConfigurer提供了SYSTEM_PROPERTIES_MODE_FALLBACK、SYSTEM_PROPERTIES_MODE_NEVER和SYSTEM_PROPERTIES_MODE_OVERRIDE三种模式。默认采用的是SYSTEM_PROPERTIES_ MODE_FALLBACK，即如

果properties文件中找不到相应配置项，则到System的Properties中查找，我们还可以选择不检查System的Properties或者覆盖它。更多信息请参照PropertyPlaceholderConfigurer的Javadoc文档。

2. **PropertyOverrideConfigurer**

PropertyPlaceholderConfigurer可以通过占位符，来明确表明bean定义中的property与properties文件中的各配置项之间的对应关系。如果说PropertyPlaceholderConfigurer做的这些是“明事”的话，那相对来说，PropertyOverrideConfigurer所做的可能就有点儿“神不知鬼不觉”了。

可以通过PropertyOverrideConfigurer对容器中配置的任何你想处理的bean定义的property信息进行覆盖替换。这听起来比较抽象，我们还是给个例子吧！比如之前的dataSource定义中，maxActive的值为100，如果我们觉得100不合适，那么可以通过PropertyOverrideConfigurer在其相应的properties文件中做如下所示配置，把100这个值给覆盖掉，如将其配置为200：

```
dataSource.maxActive=200
```

这样，当容器实例化对象的时候，该dataSource对象对应的maxActive值就是200，而不是原来XML配置中的100。也就是说，PropertyOverrideConfigurer的properties文件中的配置项，覆盖掉了原来XML中的bean定义的property信息。但这样的活动，只看XML配置的话，你根本看不出哪个bean定义的哪个property会被覆盖替换掉，只有查看PropertyOverrideConfigurer指定的properties配置文件才会了解。基本上，这种覆盖替换对于bean定义来说是透明的。

如果要对容器中的某些bean定义的property信息进行覆盖，我们需要按照如下规则提供一个PropertyOverrideConfigurer使用的配置文件：

```
beanName.propertyName=value
```

也就是说，properties文件中的键是以XML中配置的bean定义的beanName为标志开始的（通常就是id指定的值），后面跟着相应被覆盖的property的名称，比如上面的maxActive。

下面是针对dataSource定义给出的PropertyOverrideConfigurer的propeties文件配置信息：

```
# pool-adjustment.properties
dataSource.minEvictableIdleTimeMillis=1000
dataSource.maxActive=50
```

这样，当按照如下代码，将PropertyOverrideConfigurer加载到容器之后，dataSource原来定义的默认值就会被pool-adjustment.properties文件中的信息所覆盖：

```
<bean class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
   <property name="location" value="pool-adjustment.properties"/>
</bean>
```

pool-adjustment.properties中没有提供的配置项将继续使用原来XML配置中的默认值。

当容器中配置的多个PropertyOverrideConfigurer对同一个bean定义的同一个property值进行处理的时候，最后一个将会生效。

配置在properties文件中的信息通常都以明文表示，PropertyOverrideConfigurer的父类PropertyResourceConfigurer提供了一个protected类型的方法convertPropertyValue，允许子类覆盖这个方法对相应的配置项进行转换，如对加密后的字符串解密之后再覆盖到相应的bean定义中。当然，既然PropertyPlaceholderConfigurer也同样继承了PropertyResourceConfigurer，我们也可以针对PropertyPlaceholderConfigurer应用类似的功能。

3. **CustomEditorConfigurer**

其他两个BeanFactoryPostProcessor都是通过对BeanDefinition中的数据进行变更以达到某种目的。与它们有所不同，CustomEditorConfigurer是另一种类型的BeanFactoryPostProcessor实现，它只是辅助性地将后期会用到的信息注册到容器，对BeanDefinition没有做任何变动。

我们知道，不管对象是什么类型，也不管这些对象所声明的依赖对象是什么类型，通常都是通过XML（或者properties甚至其他媒介）文件格式来配置这些对象类型。但XML所记载的，都是String类型，即容器从XML格式的文件中读取的都是字符串形式，最终应用程序却是由各种类型的对象所构成。要想完成这种由字符串到具体对象的转换（不管这个转换工作最终由谁来做），都需要这种转换规则相关的信息，而CustomEditorConfigurer就是帮助我们传达类似信息的。

Spring内部通过JavaBean的PropertyEditor来帮助进行String类型到其他类型的转换工作。只要为每种对象类型提供一个PropertyEditor，就可以根据该对象类型取得与其相对应的PropertyEditor来做具体的类型转换。Spring容器内部在做具体的类型转换的时候，会采用JavaBean框架内默认的PropertyEditor搜寻逻辑，从而继承了对原生类型以及java.lang.String.java.awt.Color和java.awt.Font等类型的转换支持。同时，Spring框架还提供了自身实现的一些PropertyEditor，这些PropertyEditor大部分都位于org.springframework. beans.propertyeditors包下。以下是这些Spring提供的部分PropertyEditor的简要说明。

- StringArrayPropertyEditor。该PropertyEditor会将符合CSV格式的字符串转换成String[]数组的形式，默认是以逗号（，）分隔的字符串，但可以指定自定义的字符串分隔符。ByteArrayPropertyEditor、CharArrayPropertyEditor等都属于类似功能的PropertyEditor，参照Javadoc可以取得相应的详细信息。
- ClassEditor。根据String类型的class名称，直接将其转换成相应的Class对象，相当于通过Class.forName(String)完成的功效。可以通过String[]数组的形式传入需转换的值，以达到与提供的ClassArrayEditor同样的目的。
- FileEditor。Spring提供的对应java.io.File类型的PropertyEditor。同属于对资源进行定位的PropertyEditor还有InputStreamEditor、URLEditor等。
- LocaleEditor。针对java.util.Locale类型的PropertyEditor，格式可以参照LocaleEditor和Locale的Javadoc说明。
- PatternEditor。针对Java SE 1.4之后才引入的java.util.regex.Pattern的PropertyEditor，格式可以参照java.util.regex.Pattern类的Javadoc。

以上这些PropertyEditor，容器通常会默认加载使用，所以，即使我们不告诉容器应该如何对这些类型进行转换，容器同样可以正确地完成工作。但当我们需要指定的类型没有包含在以上所提到的PropertyEditor之列的时候，就需要给出针对这种类型的PropertyEditor实现，并通过CustomEditorConfigurer告知容器，以便容器在适当的时机使用到适当的PropertyEditor。

- 自定义PropertyEditor

通常情况下，对于Date类型，不同的Locale、不同的系统在表现形式上存在不同的需求。如系统这个部分需要以yyyy-MM-dd的形式表现日期，系统那个部分可能又需要以yyyyMMdd的形式对日期进行转换。虽然可以使用Spring提供的CustomDateEditor，不过为了能够演示自定义PropertyEditor的详细流程，在此我们有必要“重新发明轮子”！

下面是对自定义PropertyEditor实现的简单介绍。

给出针对特定对象类型的PropertyEditor实现

假设需要对yyyy/MM/dd形式的日期格式转换提供支持。虽然可以直接让PropertyEditor实现类去实现java.beans.PropertyEditor接口，不过，通常情况下，我们可以直接继承java.beans.PropertyEditorSupport类以避免实现java.beans.PropertyEditor接口的所有方法。就好像这次，我们仅仅让DatePropertyEditor完成从String到java.util.Date的转换，只需要实现setAsText(String)方法，而其他方法一概不管。该自定义PropertyEditor类定义如代码清单4-44所示。

代码清单4-44 DatePropertyEditor定义

```
public class DatePropertyEditor extends PropertyEditorSupport {
   private String datePattern;

   @Override
   public void setAsText(String text) throws IllegalArgumentException {
      DateTimeFormatter dateTimeFormatter = DateTimeFormat.forPattern(getDatePattern());
      Date dateValue = dateTimeFormatter.parseDateTime(text).toDate();
      setValue(dateValue);
   }
   public String getDatePattern() {
      return datePattern;
   }
   public void setDatePattern(String datePattern) {
      this.datePattern = datePattern;
   }
}
```

如果仅仅是支持单向的从String到相应对象类型的转换，只要覆写方法setAsText(String)即可。如果想支持双向转换，需要同时考虑getAsText()方法的覆写。

通过CustomEditorConfigurer注册自定义的PropertyEditor

如果有类似于DateFoo这样的类对java.util.Date类型的依赖声明，通常情况下，会以代码清单4-45所示的形式声明并将该类配置到容器中。

代码清单4-45 DateFoo的定义声明以及相关配置

```
类声明类似于
public class DateFoo {
   private Date date;

   public Date getDate() {
      return date;
   }
   public void setDate(Date date) {
      this.date = date;
   }
}
配置类似于
<bean id="dateFoo" class="...DateFoo">
   <property name="date">
      <value>2007/10/16</value>
   </property>
</bean>
```

但是，默认情况下，Spring容器找不到合适的PropertyEditor将字符串“2007/10/16”转换成对

象所声明的java.util.Date类型。所以，我们通过CustomEditorConfigurer将刚实现的DatePropertyEditor注册到容器，以告知容器按照DatePropertyEditor的形式进行String到java.util.Date类型的转换工作。

如果使用的容器是BeanFactory的实现，比如XmlBeanFactory，就需要通过编码手动应用CustomEditorConfigurer到容器，类似如下形式：

```
XmlBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("..."));
//
CustomEditorConfigurer ceConfigurer = new CustomEditorConfigurer();
Map customerEditors = new HashMap();
customerEditors.put(java.util.Date.class, new DatePropertyEditor());
ceConfigurer.setCustomEditors(customerEditors);
//
ceConfigurer.postProcessBeanFactory(beanFactory);
```

但如果使用的是ApplicationContext相应实现，因为ApplicationContext会自动识别BeanFactoryPostProcessor并应用，所以只需要在相应配置文件中配置一下，如代码清单4-46所示。

代码清单4-46　使用CustomEditorConfigurer注册自定义DatePropertyEditor到容器

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
   <property name="customEditors">
      <map>
         <entry key="java.util.Date">
            <ref bean="datePropertyEditor"/>
         </entry>
      </map>
   </property>
</bean>

<bean id="datePropertyEditor" class="...DatePropertyEditor">
   <property name="datePattern">
      <value>yyyy/MM/dd</value>
   </property>
</bean>
```

Spring 2.0之前通常是通过CustomEditorConfigurer的customEditors属性来指定自定义的PropertyEditor。2.0之后，比较提倡使用propertyEditorRegistrars属性来指定自定义的PropertyEditor。不过，这样我们就需要再多做一步工作，就是给出一个org.springframework.beans.PropertyEditorRegistrar的实现。这也很简单，代码清单4-47给出了相应的实例。

代码清单4-47　DatePropertyEditorRegistrar定义

```
public class DatePropertyEditorRegistrar implements PropertyEditorRegistrar {
   private PropertyEditor propertyEditor;

   public void registerCustomEditors(PropertyEditorRegistry peRegistry) {
      peRegistry.registerCustomEditor(java.util.Date.class, getPropertyEditor());
   }

   public PropertyEditor getPropertyEditor() {
      return propertyEditor;
   }

   public void setPropertyEditor(PropertyEditor propertyEditor) {
      this.propertyEditor = propertyEditor;
```

```
    }
}
```

这样，2.0之后所提倡的注册自定义PropertyEditor的方式，如代码清单4-48所示。

代码清单4-48 通过CustomEditorConfigurer的propertyEditorRegistrars注册自定义PropertyEditor

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="propertyEditorRegistrars">
    <list>
      <ref bean="datePropertyEditorRegistrar"/>
    </list>
  </property>
</bean>

<bean id="datePropertyEditorRegistrar" class="...DatePropertyEditorRegistrar">
  <property name="propertyEditor">
    <ref bean="datePropertyEditor"/>
  </property>
</bean>

<bean id="datePropertyEditor" class="...DatePropertyEditor">
  <property name="datePattern">
    <value>yyyy/MM/dd</value>
  </property>
</bean>
```

要是还有其他扩展类型的PropertyEditor，可以在propertyEditorRegistrars的<list>中一并指定。

4.4.3 了解 bean 的一生

在已经可以借助于BeanFactoryPostProcessor来干预Magic实现的第一个阶段（容器启动阶段）的活动之后，我们就可以开始探索下一个阶段，即bean实例化阶段的实现逻辑了。

容器启动之后，并不会马上就实例化相应的bean定义。我们知道，容器现在仅仅拥有所有对象的BeanDefinition来保存实例化阶段将要用的必要信息。只有当请求方通过BeanFactory的getBean()方法来请求某个对象实例的时候，才有可能触发Bean实例化阶段的活动。BeanFactory的getBean方法可以被客户端对象显式调用，也可以在容器内部隐式地被调用。隐式调用有如下两种情况。

- 对于BeanFactory来说，对象实例化默认采用延迟初始化。通常情况下，当对象A被请求而需要第一次实例化的时候，如果它所依赖的对象B之前同样没有被实例化，那么容器会先实例化对象A所依赖的对象。这时容器内部就会首先实例化对象B，以及对象 A依赖的其他还没有被实例化的对象。这种情况是容器内部调用getBean()，对于本次请求的请求方是隐式的。
- ApplicationContext启动之后会实例化所有的bean定义，这个特性在本书中已经多次提到。但ApplicationContext在实现的过程中依然遵循Spring容器实现流程的两个阶段，只不过它会在启动阶段的活动完成之后，紧接着调用注册到该容器的所有bean定义的实例化方法getBean()。这就是为什么当你得到ApplicationContext类型的容器引用时，容器内所有对象已经被全部实例化完成。不信你查一下类org.springframework.context.support.AbstractApplicationContext的refresh()方法。

之所以说getBean()方法是有可能触发Bean实例化阶段的活动，是因为只有当对应某个bean定义的getBean()方法第一次被调用时，不管是显式的还是隐式的，Bean实例化阶段的活动才会被触发，

第二次被调用则会直接返回容器缓存的第一次实例化完的对象实例（prototype类型bean除外）。当getBean()方法内部发现该bean定义之前还没有被实例化之后，会通过createBean()方法来进行具体的对象实例化，实例化过程如图4-10所示。

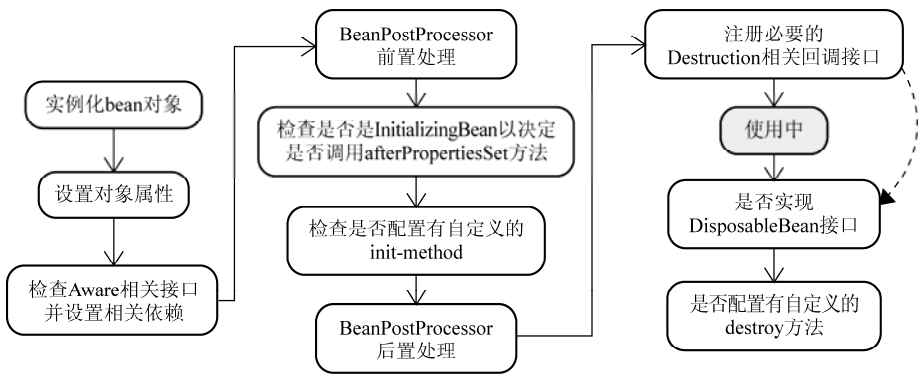


图4-10　Bean的实例化过程

Spring容器将对其所管理的对象全部给予统一的生命周期管理，这些被管理的对象完全摆脱了原来那种“new完后被使用，脱离作用域后即被回收”的命运。下面我们将详细看一看现在的每个bean在容器中是如何走过其一生的。

提示　可以在org.springframework.beans.factory.support.AbstractBeanFactory类的代码中查看到getBean()方法的完整实现逻辑，可以在其子类org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory的代码中一窥createBean()方法的全貌。

1. Bean的实例化与BeanWrapper

容器在内部实现的时候，采用“策略模式（Strategy Pattern）”来决定采用何种方式初始化bean实例。通常，可以通过反射或者CGLIB动态字节码生成来初始化相应的bean实例或者动态生成其子类。

org.springframework.beans.factory.support.InstantiationStrategy定义是实例化策略的抽象接口，其直接子类SimpleInstantiationStrategy实现了简单的对象实例化功能，可以通过反射来实例化对象实例，但不支持方法注入方式的对象实例化。CglibSubclassingInstantiationStrategy继承了SimpleInstantiationStrategy的以反射方式实例化对象的功能，并且通过CGLIB的动态字节码生成功能，该策略实现类可以动态生成某个类的子类，进而满足了方法注入所需的对象实例化需求。默认情况下，容器内部采用的是CglibSubclassingInstantiationStrategy。

容器只要根据相应bean定义的BeanDefintion取得实例化信息，结合CglibSubclassingInstantiationStrategy以及不同的bean定义类型，就可以返回实例化完成的对象实例。但是，返回方式上有些“点缀”。不是直接返回构造完成的对象实例，而是以BeanWrapper对构造完成的对象实例进行包裹，返回相应的BeanWrapper实例。

至此，第一步结束。

BeanWrapper接口通常在Spring框架内部使用，它有一个实现类org.springframework.beans.BeanWrapperImpl。其作用是对某个bean进行“包裹”，然后对这个“包裹”的bean进行操作，比如

设置或者获取bean的相应属性值。而在第一步结束后返回BeanWrapper实例而不是原先的对象实例，就是为了第二步“设置对象属性”。

BeanWrapper定义继承了org.springframework.beans.PropertyAccessor接口，可以以统一的方式对对象属性进行访问；BeanWrapper定义同时又直接或者间接继承了PropertyEditorRegistry和TypeConverter接口。不知你是否还记得CustomEditorConfigurer？当把各种PropertyEditor注册给容器时，知道后面谁用到这些PropertyEditor吗？对，就是BeanWrapper！在第一步构造完成对象之后，Spring会根据对象实例构造一个BeanWrapperImpl实例，然后将之前CustomEditorConfigurer注册的PropertyEditor复制一份给BeanWrapperImpl实例（这就是BeanWrapper同时又是PropertyEditorRegistry的原因）。这样，当BeanWrapper转换类型、设置对象属性值时，就不会无从下手了。

使用BeanWrapper对bean实例操作很方便，可以免去直接使用Java反射API（Java Reflection API）操作对象实例的烦琐。来看一段代码（见代码清单4-49），之后我们就会更加清楚Spring容器内部是如何设置对象属性的了！

代码清单4-49 使用BeanWrapper操作对象

```
Object provider = Class.forName("package.name.FXNewsProvider").newInstance();
Object listener = Class.forName("package.name.DowJonesNewsListener").newInstance();
Object persister = Class.forName("package.name.DowJonesNewsPersister").newInstance();

BeanWrapper newsProvider = new BeanWrapperImpl(provider);
newsProvider.setPropertyValue("newsListener", listener);
newsProvider.setPropertyValue("newPersistener", persister);

assertTrue(newsProvider.getWrappedInstance() instanceof FXNewsProvider);
assertSame(provider, newsProvider.getWrappedInstance());
assertSame(listener, newsProvider.getPropertyValue("newsListener"));
assertSame(persister, newsProvider.getPropertyValue("newPersistener"));
```

我想有了BeanWrapper的帮助，你不会想直接使用Java反射API来做同样事情的。代码清单4-50演示了同样的功能，即直接使用Java反射API是如何实现的（忽略了异常处理相关代码）。

代码清单4-50 直接使用Java反射API操作对象

```
Object provider = Class.forName("package.name.FXNewsProvider").newInstance();
Object listener = Class.forName("package.name.DowJonesNewsListener").newInstance();
Object persister = Class.forName("package.name.DowJonesNewsPersister").newInstance();

Class providerClazz = provider.getClass();
Field listenerField = providerClazz.getField("newsListener");
listenerField.set(provider, listener);
Field persisterField = providerClazz.getField("newsListener");
persisterField.set(provider, persister);

assertSame(listener, listenerField.get(provider));
assertSame(persister, persisterField.get(provider));
```

如果你觉得没有太大差别，那是因为没有看到紧随其后的那些异常（exception）还有待处理！

2. 各色的Aware接口

当对象实例化完成并且相关属性以及依赖设置完成之后，Spring容器会检查当前对象实例是否实现了一系列的以Aware命名结尾的接口定义。如果是，则将这些Aware接口定义中规定的依赖注入给

当前对象实例。

这些Aware接口为如下几个。

- org.springframework.beans.factory.BeanNameAware。如果Spring容器检测到当前对象实例实现了该接口，会将该对象实例的bean定义对应的beanName设置到当前对象实例。
- org.springframework.beans.factory.BeanClassLoaderAware。如果容器检测到当前对象实例实现了该接口，会将对应加载当前bean的Classloader注入当前对象实例。默认会使用加载org.springframework.util.ClassUtils类的Classloader。
- org.springframework.beans.factory.BeanFactoryAware。在介绍方法注入的时候，我们提到过使用该接口以便每次获取prototype类型bean的不同实例。如果对象声明实现了BeanFactoryAware接口，BeanFactory容器会将自身设置到当前对象实例。这样，当前对象实例就拥有了一个BeanFactory容器的引用，并且可以对这个容器内允许访问的对象按照需要进行访问。

以上几个Aware接口只是针对BeanFactory类型的容器而言，对于ApplicationContext类型的容器，也存在几个Aware相关接口。不过在检测这些接口并设置相关依赖的实现机理上，与以上几个接口处理方式有所不同，使用的是下面将要说到的BeanPostProcessor方式。不过，设置Aware接口这一步与BeanPostProcessor是相邻的，把这几个接口放到这里一起提及，也没什么不可以的。

对于ApplicationContext类型容器，容器在这一步还会检查以下几个Aware接口并根据接口定义设置相关依赖。

- org.springframework.context.ResourceLoaderAware。ApplicationContext实现了Spring的ResourceLoader接口（后面会提及详细信息）。当容器检测到当前对象实例实现了ResourceLoaderAware接口之后，会将当前ApplicationContext自身设置到对象实例，这样当前对象实例就拥有了其所在ApplicationContext容器的一个引用。
- org.springframework.context.ApplicationEventPublisherAware。ApplicationContext作为一个容器，同时还实现了ApplicationEventPublisher接口，这样，它就可以作为ApplicationEventPublisher来使用。所以，当前ApplicationContext容器如果检测到当前实例化的对象实例声明了ApplicationEventPublisherAware接口，则会将自身注入当前对象。
- org.springframework.context.MessageSourceAware。ApplicationContext通过MessageSource接口提供国际化的信息支持，即I18n（Internationalization）。它自身就实现了MessageSource接口，所以当检测到当前对象实例实现了MessageSourceAware接口，则会将自身注入当前对象实例。
- org.springframework.context.ApplicationContextAware。如果ApplicationContext容器检测到当前对象实现了ApplicationContextAware接口，则会将自身注入当前对象实例。

3. BeanPostProcessor

BeanPostProcessor的概念容易与BeanFactoryPostProcessor的概念混淆。但只要记住BeanPostProcessor是存在于对象实例化阶段，而BeanFactoryPostProcessor则是存在于容器启动阶段，这两个概念就比较容易区分了。

与BeanFactoryPostProcessor通常会处理容器内所有符合条件的BeanDefinition类似，BeanPostProcessor会处理容器内所有符合条件的实例化后的对象实例。该接口声明了两个方法，分别在两个不同的时机执行，见如下代码定义：

```
public interface BeanPostProcessor
```

```
{
   Object postProcessBeforeInitialization(Object bean, String beanName) throws ➥
   BeansException;
   Object postProcessAfterInitialization(Object bean, String beanName) throws ➥
   BeansException;
}
```

postProcessBeforeInitialization()方法是图4-10中BeanPostProcessor前置处理这一步将会执行的方法，postProcessAfterInitialization()则是对应图4-10中BeanPostProcessor后置处理那一步将会执行的方法。BeanPostProcessor的两个方法中都传入了原来的对象实例的引用，这为我们扩展容器的对象实例化过程中的行为提供了极大的便利，我们几乎可以对传入的对象实例执行任何的操作。

通常比较常见的使用BeanPostProcessor的场景，是处理标记接口实现类，或者为当前对象提供代理实现。在图4-10的第三步中，ApplicationContext对应的那些Aware接口实际上就是通过BeanPostProcessor的方式进行处理的。当ApplicationContext中每个对象的实例化过程走到BeanPostProcessor前置处理这一步时，ApplicationContext容器会检测到之前注册到容器的ApplicationContextAwareProcessor这个BeanPostProcessor的实现类，然后就会调用其postProcessBeforeInitialization()方法，检查并设置Aware相关依赖。ApplicationContextAwareProcessor的postProcessBeforeInitialization()代码很简单明了，见代码清单4-51。

代码清单4-51 postProcessBeforeInitialization方法定义

```
public Object postProcessBeforeInitialization(Object bean, String beanName) throws ➥
BeansException {
   if (bean instanceof ResourceLoaderAware) {
      ((ResourceLoaderAware) bean).setResourceLoader(this.applicationContext);
   }
   if (bean instanceof ApplicationEventPublisherAware) {
      ((ApplicationEventPublisherAware) bean).setApplicationEventPublisher ➥
      (this.applicationContext);
   }
   if (bean instanceof MessageSourceAware) {
      ((MessageSourceAware) bean).setMessageSource(this.applicationContext);
   }
   if (bean instanceof ApplicationContextAware) {
      ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
   }
   return bean;
}
```

除了检查标记接口以便应用自定义逻辑，还可以通过BeanPostProcessor对当前对象实例做更多的处理。比如替换当前对象实例或者字节码增强当前对象实例等。Spring的AOP则更多地使用BeanPostProcessor来为对象生成相应的代理对象，如org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator。我们将在Spring AOP部分详细介绍该类和AOP相关概念。

BeanPostProcessor是容器提供的对象实例化阶段的强有力的扩展点。为了进一步演示它的强大威力，我们有必要实现一个自定义的BeanPostProcessor。

- 自定义**BeanPostProcessor**

假设系统中所有的IFXNewsListener实现类需要从某个位置取得相应的服务器连接密码，而且系统中保存的密码是加密的，那么在IFXNewsListener发送这个密码给新闻服务器进行连接验证的时

候，首先需要对系统中取得的密码进行解密，然后才能发送。我们将采用BeanPostProcessor技术，对所有的IFXNewsListener的实现类进行统一的解密操作。

(1) 标注需要进行解密的实现类

为了能够识别那些需要对服务器连接密码进行解密的IFXNewsListener实现，我们声明了接口PasswordDecodable，并要求相关IFXNewsListener实现类实现该接口。PasswordDecodable接口声明以及相关的IFXNewsListener实现类定义见代码清单4-52。

代码清单4-52　PasswordDecodable接口声明以及相关的IFXNewsListener实现类

```
public interface PasswordDecodable {
   String getEncodedPassword();
   void setDecodedPassword(String password);
}

public class DowJonesNewsListener implements IFXNewsListener,PasswordDecodable {
   private String password;

   public String[] getAvailableNewsIds() {
      // 省略
   }

   public FXNewsBean getNewsByPK(String newsId) {
      // 省略
   }

   public void postProcessIfNecessary(String newsId) {
      // 省略
   }

   public String getEncodedPassword() {
      return this.password;
   }

   public void setDecodedPassword(String password) {
      this.password = password;
   }

}
```

(2) 实现相应的BeanPostProcessor对符合条件的Bean实例进行处理

我们通过PasswordDecodable接口声明来区分将要处理的对象实例[①]，当检查到当前对象实例实现了该接口之后，就会从当前对象实例取得加密后的密码，并对其解密。然后将解密后的密码设置回当前对象实例。之后，返回的对象实例所持有的就是解密后的密码，逻辑如代码清单4-53所示。

代码清单4-53　用于解密的自定义BeanPostProcessor实现类

```
public class PasswordDecodePostProcessor implements BeanPostProcessor {

   public Object postProcessAfterInitialization(Object object, String beanName)
        throws BeansException {
      return object;
   }
   public Object postProcessBeforeInitialization(Object object, String beanName)
        throws BeansException {
      if(object instanceof PasswordDecodable)
```

① 如果有其他方式可以区分将要处理的对象实例，那么声明类似的标记接口（Marker Interface）就不是必须的。

```
        {
          String encodedPassword = ((PasswordDecodable)object).getEncodedPassword();
          String decodedPassword = decodePassword(encodedPassword);
          ((PasswordDecodable)object).setDecodedPassword(decodedPassword);
        }
        return object;
      }
      private String decodePassword(String encodedPassword) {
        // 实现解码逻辑
        return encodedPassword;
      }
}
```

(3) 将自定义的BeanPostProcessor注册到容器

只有将自定义的BeanPostProcessor实现类告知容器，容器才会在合适的时机应用它。所以，我们需要将PasswordDecodePostProcessor注册到容器。

对于BeanFactory类型的容器来说，我们需要通过手工编码的方式将相应的BeanPostProcessor注册到容器，也就是调用ConfigurableBeanFactory的addBeanPostProcessor()方法，见如下代码：

```
ConfigurableBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource(...));
beanFactory.addBeanPostProcessor(new PasswordDecodePostProcessor());
...
// getBean();
```

对于ApplicationContext容器来说，事情则方便得多，直接将相应的BeanPostProcessor实现类通过通常的XML配置文件配置一下即可。ApplicationContext容器会自动识别并加载注册到容器的BeanPostProcessor，如下配置内容将我们的PasswordDecodePostProcessor注册到容器：

```
<beans>
   <bean id="passwordDecodePostProcessor" class="package.name.PasswordDecodePostProcessor">
      <!--如果需要，注入必要的依赖-->
   </bean>

   ...
</beans>
```

合理利用BeanPostProcessor这种Spring的容器扩展机制，将可以构造强大而灵活的应用系统。

提示 实际上，有一种特殊类型的BeanPostProcessor我们没有提到，它的执行时机与通常的BeanPostProcessor不同。

org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor接口可以在对象的实例化过程中导致某种类似于电路“短路”的效果。实际上，并非所有注册到Spring容器内的bean定义都是按照图4-10的流程实例化的。在所有的步骤之前，也就是实例化bean对象步骤之前，容器会首先检查容器中是否注册有InstantiationAwareBeanPostProcessor类型的BeanPostProcessor。如果有，首先使用相应的InstantiationAwareBeanPostProcessor来构造对象实例。构造成功后直接返回构造完成的对象实例，而不会按照“正规的流程”继续执行。这就是它可能造成“短路”的原因。

不过，通常情况下都是Spring容器内部使用这种特殊类型的BeanPostProcessor做一些动态对象代理等工作，我们使用普通的BeanPostProcessor实现就可以。这里简单提及一下，目的是让大家有所了解。

4. InitializingBean和init-method

org.springframework.beans.factory.InitializingBean是容器内部广泛使用的一个对象生命周期标识接口，其定义如下：

```
public interface InitializingBean {
   void afterPropertiesSet() throws Exception;
}
```

该接口定义很简单，其作用在于，在对象实例化过程调用过“BeanPostProcessor的前置处理”之后，会接着检测当前对象是否实现了InitializingBean接口，如果是，则会调用其afterPropertiesSet()方法进一步调整对象实例的状态。比如，在有些情况下，某个业务对象实例化完成后，还不能处于可以使用状态。这个时候就可以让该业务对象实现该接口，并在方法afterPropertiesSet()中完成对该业务对象的后续处理。

虽然该接口在Spring容器内部广泛使用，但如果真的让我们的业务对象实现这个接口，则显得Spring容器比较具有侵入性。所以，Spring还提供了另一种方式来指定自定义的对象初始化操作，那就是在XML配置的时候，使用<bean>的init-method属性。

通过init-method，系统中业务对象的自定义初始化操作可以以任何方式命名，而不再受制于InitializingBean的afterPropertiesSet()。如果系统开发过程中规定：所有业务对象的自定义初始化操作都必须以init()命名，为了省去挨个<bean>的设置init-method这样的烦琐，我们还可以通过最顶层的<beans>的default-init-method统一指定这一init()方法名。

一般，我们是在集成第三方库，或者其他特殊的情况下，才会需要使用该特性。比如，ObjectLab提供了一个外汇系统交易日计算的开源实现——ObjectLabKit，系统在使用它提供的DateCalculator时，封装类会通过一个自定义的初始化方法来为这些DateCalculator提供计算交易日所需要排除的休息日信息。代码清单4-54给出了封装类的部分代码。

代码清单4-54　DateCalculator封装类定义

```
public class FXTradeDateCalculator {
    public  static final DateTimeFormatter FRONT_DATE_FORMATTER = ➥
    DateTimeFormat.forPattern("yyyyMMdd");
    private static final Set<LocalDate> holidaySet = ➥
    new HashSet<LocalDate>();
    private static final String holidayKey = "JPY";
    private SqlMapClientTemplate sqlMapClientTemplate;

    public FXTradeDateCalculator(SqlMapClientTemplate sqlMapClientTemplate)
    {
        this.sqlMapClientTemplate = sqlMapClientTemplate;
    }

    public void setupHolidays()
    {
        List holidays = getSystemHolidays();
        if(!ListUtils.isEmpty(holidays))
        {
            for(int i=0,size=holidays.size();i<size;i++)
            {
                String holiday = (String)holidays.get(i);
                LocalDate date = ➥
                FRONT_DATE_FORMATTER.parseDateTime(holiday).toLocalDate();
```

```
                holidaySet.add(date);
            }
        }
        LocalDateKitCalculatorsFactory ➥
        .getDefaultInstance().registerHolidays(holidayKey, holidaySet);
    }
    public DateCalculator<LocalDate> getForwardDateCalculator()
    {
        return LocalDateKitCalculatorsFactory ➥
            .getDefaultInstance() ➥
            .getDateCalculator(holidayKey, HolidayHandlerType.FORWARD);
        }

    public DateCalculator<LocalDate> getBackwardDateCalculator()
    {
        return LocalDateKitCalculatorsFactory ➥
            .getDefaultInstance() ➥
            .getDateCalculator(holidayKey, HolidayHandlerType.BACKWARD);
    }
    public List getSystemHolidays()
    {
        return getSqlMapClientTemplate() ➥
        .queryForList("CommonContext.holiday", null);
    }
}
```

为了保证getForwardDateCalculator()和getBackwardDateCalculator()方法返回的DateCalculator已经将休息日考虑进去，在这两个方法被调用之前，我们需要setupHolidays()首先被调用，以保证将休息日告知DateCalculator，使它能够在计算交易日的时候排除掉这些休息日的日期。因此，我们需要在配置文件中完成类似代码清单4-55所示的配置，以保证在对象可用之前，setupHolidays()方法会首先被调用。

代码清单4-55 使用init-method保证封装类的初始化方法得以执行

```
<beans>
  <bean id="tradeDateCalculator" class="FXTradeDateCalculator" ➥
  init-method="setupHolidays">
    <constructor-arg>
      <ref bean="sqlMapClientTemplate"/>
    </constructor-arg>
  </bean>

  <bean id="sqlMapClientTemplate" ➥
  class="org.springframework.orm.ibatis.SqlMapClientTemplate">
    ...
  </bean>

  ...
</beans>
```

当然，我们也可以让FXTradeDateCalculator实现InitializingBean接口，然后将setupHolidays()方法的逻辑转移到afterPropertiesSet()方法。不过，相对来说还是采用init-method的方式比较灵活，并且没有那么强的侵入性。

可以认为在InitializingBean和init-method中任选其一就可以帮你完成类似的初始化工作。除非……，除非你真的那么“幸运”，居然需要在同一个业务对象上按照先后顺序执行两个初始化方

法。这个时候，就只好在同一对象上既实现InitializingBean的afterPropertiesSet()，又提供自定义初始化方法啦！

5. **DisposableBean与destroy-method**

当所有的一切，该设置的设置，该注入的注入，该调用的调用完成之后，容器将检查singleton类型的bean实例，看其是否实现了org.springframework.beans.factory.DisposableBean接口。或者其对应的bean定义是否通过<bean>的destroy-method属性指定了自定义的对象销毁方法。如果是，就会为该实例注册一个用于对象销毁的回调（Callback），以便在这些singleton类型的对象实例销毁之前，执行销毁逻辑。

与InitializingBean和init-method用于对象的自定义初始化相对应，DisposableBean和destroy-method为对象提供了执行自定义销毁逻辑的机会。

最常见到的该功能的使用场景就是在Spring容器中注册数据库连接池，在系统退出后，连接池应该关闭，以释放相应资源。代码清单4-56演示了通常情况下使用destroy-method处理资源释放的数据源注册配置。

代码清单4-56　使用了自定义销毁方法的数据源配置定义

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
   <property name="url">
      <value>${jdbc.url}</value>
   </property>
   <property name="driverClassName">
      <value>${jdbc.driver}</value>
   </property>
   <property name="username">
      <value>${jdbc.username}</value>
   </property>
   <property name="password">
      <value>${jdbc.password}</value>
   </property>
   ...
</bean>
```

不过，这些自定义的对象销毁逻辑，在对象实例初始化完成并注册了相关的回调方法之后，并不会马上执行。回调方法注册后，返回的对象实例即处于使用状态，只有该对象实例不再被使用的时候，才会执行相关的自定义销毁逻辑，此时通常也就是Spring容器关闭的时候。但Spring容器在关闭之前，不会聪明到自动调用这些回调方法。所以，需要我们告知容器，在哪个时间点来执行对象的自定义销毁方法。

对于BeanFactory容器来说。我们需要在独立应用程序的主程序退出之前，或者其他被认为是合适的情况下（依照应用场景而定），如代码清单4-57所示，调用ConfigurableBeanFactory提供的destroySingletons()方法销毁容器中管理的所有singleton类型的对象实例。

代码清单4-57　使用ConfigurableBeanFactory的destroySingletons()方法触发销毁对象行为

```
public class ApplicationLauncher
{
   public static void main(String[] args) {
   BasicConfigurator.configure();
   BeanFactory container = new XmlBeanFactory(new ClassPathResource("..."));
   BusinessObject bean = (BusinessObject)container.getBean("...");
```

```
    bean.doSth();
    ((ConfigurableListableBeanFactory)container).destroySingletons();
    // 应用程序退出，容器关闭
    }
}
```

如果不能在合适的时机调用destroySingletons()，那么所有实现了DisposableBean接口的对象实例或者声明了destroy-method的bean定义对应的对象实例，它们的自定义对象销毁逻辑就形同虚设，因为根本就不会被执行！

对于ApplicationContext容器来说。道理是一样的。但AbstractApplicationContext为我们提供了registerShutdownHook()方法，该方法底层使用标准的Runtime类的addShutdownHook()方式来调用相应bean对象的销毁逻辑，从而保证在Java虚拟机退出之前，这些singtleton类型的bean对象实例的自定义销毁逻辑会被执行。当然AbstractApplicationContext注册的shutdownHook不只是调用对象实例的自定义销毁逻辑，也包括ApplicationContext相关的事件发布等，代码清单4-58演示了该方法的使用。

代码清单4-58 使用registerShutdownHook()方法注册并触发对象销毁逻辑回调行为

```
public class ApplicationLauncher
{
   public static void main(String[] args) {
      BasicConfigurator.configure();
      BeanFactory container = new ClassPathXmlApplicationContext("...");
      ((AbstractApplicationContext)container).registerShutdownHook();
      BusinessObject bean = (BusinessObject)container.getBean("...");
      bean.doSth();
      // 应用程序退出，容器关闭
   }
}
```

同样的道理，在Spring 2.0引入了自定义scope之后，使用自定义scope的相关对象实例的销毁逻辑，也应该在合适的时机被调用执行。不过，所有这些规则不包含prototype类型的bean实例，因为prototype对象实例在容器实例化并返回给请求方之后，容器就不再管理这种类型对象实例的生命周期了。

至此，bean走完了它在容器中“光荣”的一生。

4.5 小结

Spring的IoC容器主要有两种，即BeanFactory和ApplicationContext。本章伊始，首先对这两种容器做了总体上的介绍，然后转入本章的重点，也就是Spring的BeanFactory基础容器。

我们从对比使用BeanFactory开发前后的差别开始，阐述了BeanFactory作为一个具体的IoC Service Provider，它是如何支持各种对象注册以及依赖关系绑定的。XML自始至终都是Spring的IoC容器支持最完善的Configuration Metadata提供方式。所以，我们接着从XML入手，深入挖掘了BeanFactory（以及ApplicationContext）的各种潜力。

对于充满好奇心的我们，不会只停留在会使用BeanFactory进行开发这一层面。所以，最后我们又一起探索了BeanFactory（当然，也是ApplicationContext）实现背后的各种奥秘。BeanFactory是Spring提供的基础IoC容器，但并不是Spring提供的唯一IoC容器。ApplicationContext构建于BeanFactory之上，提供了许多BeanFactory之外的特性。下一章，我们将一起走入Application-Context的世界。

第5章 Spring IoC容器 ApplicationContext

本章内容

- 统一资源加载策略
- 国际化信息支持
- 容器内部事件发布
- 多配置模块加载的简化

作为Spring提供的较之BeanFactory更为先进的IoC容器实现，ApplicationContext除了拥有BeanFactory支持的所有功能之外，还进一步扩展了基本容器的功能，包括BeanFactoryPostProcessor、BeanPostProcessor以及其他特殊类型bean的自动识别、容器启动后bean实例的自动初始化、国际化的信息支持、容器内事件发布等。真是“青出于蓝而胜于蓝”啊！

Spring为基本的BeanFactory类型容器提供了XmlBeanFactory实现。相应地，它也为ApplicationContext类型容器提供了以下几个常用的实现。

- org.springframework.context.support.FileSystemXmlApplicationContext。在默认情况下，从文件系统加载bean定义以及相关资源的ApplicationContext实现。
- org.springframework.context.support.ClassPathXmlApplicationContext。在默认情况下，从Classpath加载bean定义以及相关资源的ApplicationContext实现。
- org.springframework.web.context.support.XmlWebApplicationContext。Spring提供的用于Web应用程序的ApplicationContext实现，我们将在第六部分更多地接触到它。

更多实现可以参照org.springframework.context.ApplicationContext接口定义的Javadoc，这里不再赘述。

第4章中说明了ApplicationContext所支持的大部分功能。下面主要围绕ApplicationContext较之BeanFactory特有的一些特性展开讨论，即国际化（I18n）信息支持、统一资源加载策略以及容器内事件发布等。

5.1 统一资源加载策略

要搞清楚Spring为什么提供这么一个功能，还是从Java SE提供的标准类java.net.URL说起比较好。URL全名是Uniform Resource Locator（统一资源定位器），但多少有些名不副实的味道。

首先，说是统一资源定位，但基本实现却只限于网络形式发布的资源的查找和定位工作，基本上只提供了基于HTTP、FTP、File等协议（sun.net.www.protocol包下所支持的协议）的资源定位功能。虽然也提供了扩展的接口，但从一开始，其自身的“定位”就已经趋于狭隘了。实际上，资源这个词的范围比较广义，资源可以任何形式存在，如以二进制对象形式存在、以字节流形式存在、以文件形式存在等；而且，资源也可以存在于任何场所，如存在于文件系统、存在于Java应用的Classpath中，甚至存在于URL可以定位的地方。

其次，从某些程度上来说，该类的功能职责划分不清，资源的查找和资源的表示没有一个清晰的界限。当前情况是，资源查找后返回的形式多种多样，没有一个统一的抽象。理想情况下，资源查找完成后，返回给客户端的应该是一个统一的资源抽象接口，客户端要对资源进行什么样的处理，应该由资源抽象接口来界定，而不应该成为资源的定位者和查找者同时要关心的事情。

所以，在这个前提下[①]，Spring提出了一套基于org.springframework.core.io.Resource和org.springframework.core.io.ResourceLoader接口的资源抽象和加载策略。

5.1.1 Spring中的Resource

Spring框架内部使用org.springframework.core.io.Resource接口作为所有资源的抽象和访问接口，我们之前在构造BeanFactory的时候已经接触过它，如下代码：

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("..."));
...
```

其中ClassPathResource就是Resource的一个特定类型的实现，代表的是位于Classpath中的资源。

Resource接口可以根据资源的不同类型，或者资源所处的不同场合，给出相应的具体实现。Spring框架在这个理念的基础上，提供了一些实现类（可以在org.springframework.core.io包下找到这些实现类）。

- ByteArrayResource。将字节（byte）数组提供的数据作为一种资源进行封装，如果通过InputStream形式访问该类型的资源，该实现会根据字节数组的数据，构造相应的ByteArrayInputStream并返回。
- ClassPathResource。该实现从Java应用程序的ClassPath中加载具体资源并进行封装，可以使用指定的类加载器（ClassLoader）或者给定的类进行资源加载。
- FileSystemResource。对java.io.File类型的封装，所以，我们可以以文件或者URL的形式对该类型资源进行访问，只要能跟File打的交道，基本上跟FileSystemResource也可以。
- UrlResource。通过java.net.URL进行的具体资源查找定位的实现类，内部委派URL进行具体的资源操作。
- InputStreamResource。将给定的InputStream视为一种资源的Resource实现类，较为少用。可能的情况下，以ByteArrayResource以及其他形式资源实现代之。

如果以上这些资源实现还不能满足要求，那么我们还可以根据相应场景给出自己的实现，只需实现org.springframework.core.io.Resource接口就是了。代码清单5-1给出了该接口的定义。

代码清单5-1 Resource接口定义

```
public interface Resource extends InputStreamSource {

  boolean exists();

  boolean isOpen();

  URL getURL() throws IOException;

  File getFile() throws IOException;
```

① 只是个人看法，至于说SpringTeam基于如何考虑，我也不知，或许只是给出了URL之外的另一种选择，也可能是“重新发明一套更好用的轮子”，我姑且说之，你姑且听之，或者用之……

```
    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();

}

public interface InputStreamSource {
   InputStream getInputStream() throws IOException;
}
```

该接口定义了7个方法，可以帮助我们查询资源状态、访问资源内容，甚至根据当前资源创建新的相对资源。不过，要真想实现自定义的Resource，倒是真没必要直接实现该接口，我们可以继承org.springframework.core.io.AbstractResource抽象类，然后根据当前具体资源特征，覆盖相应的方法就可以了。什么？让我给个实现的例子？算了吧，目前我还没碰到这样的需求。呵呵！要真的碰上了，你只要知道有这么“一出儿”就行了。

5.1.2 ResourceLoader，“更广义的URL”

资源是有了，但如何去查找和定位这些资源，则应该是ResourceLoader的职责所在了。org.springframework.core.io.ResourceLoader接口是资源查找定位策略的统一抽象，具体的资源查找定位策略则由相应的ResourceLoader实现类给出。我想，把ResourceLoader称作统一资源定位器或许才更恰当一些吧！ResourceLoader定义如下：

```
public interface ResourceLoader {
   String CLASSPATH_URL_PREFIX = ResourceUtils.CLASSPATH_URL_PREFIX;
   Resource getResource(String location);
   ClassLoader getClassLoader();
}
```

其中最主要的就是Resource getResource(String location);方法，通过它，我们就可以根据指定的资源位置，定位到具体的资源实例。

1. 可用的ResourceLoader

- DefaultResourceLoader

ResourceLoader有一个默认的实现类，即org.springframework.core.io.DefaultResourceLoader，该类默认的资源查找处理逻辑如下。

(1) 首先检查资源路径是否以classpath:前缀打头，如果是，则尝试构造ClassPathResource类型资源并返回。

(2) 否则，(a) 尝试通过URL，根据资源路径来定位资源，如果没有抛出MalformedURLException，则会构造UrlResource类型的资源并返回；(b)如果还是无法根据资源路径定位指定的资源，则委派getResourceByPath(String) 方法来定位，DefaultResourceLoader 的getResourceByPath(String)方法默认实现逻辑是，构造ClassPathResource类型的资源并返回。

在这个基础上，让我们来看一下DefaultResourceLoader的行为是如何反应到程序中的吧！代码清单5-2给出的代码片段演示了DefaultResourceLoader的具体行为。

代码清单5-2 DefaultResourceLoader使用演示

```
ResourceLoader resourceLoader = new DefaultResourceLoader();
```

```
Resource fakeFileResource = resourceLoader.getResource("D:/spring21site/README");
assertTrue(fakeFileResource instanceof ClassPathResource);
assertFalse(fakeFileResource.exists());

Resource urlResource1 = resourceLoader.getResource("file:D:/spring21site/README");
assertTrue(urlResource1 instanceof UrlResource);

Resource urlResource2 = resourceLoader.getResource("http://www.spring21.cn");
assertTrue(urlResource2 instanceof UrlResource);

try{
   fakeFileResource.getFile();
   fail("no such file with path["+fakeFileResource.getFilename()+"] exists in classpath");
}
catch(FileNotFoundException e){
   //
}
try{
   urlResource1.getFile();
}
catch(FileNotFoundException e){
   fail();
}
```

尤其注意fakeFileResource资源的类型，并不是我们所预期的FileSystemResource类型，而是ClassPathResource类型，这是由DefaultResourceLoader的资源查找逻辑所决定的。如果最终没有找到符合条件的相应资源，getResourceByPath(String)方法就会构造一个实际上并不存在的资源并返回。而指定有协议前缀的资源路径，则通过URL能够定位，所以，返回的都是UrlResource类型。

- FileSystemResourceLoader

为了避免DefaultResourceLoader在最后getResourceByPath(String)方法上的不恰当处理，我们可以使用org.springframework.core.io.FileSystemResourceLoader，它继承自DefaultResourceLoader，但覆写了getResourceByPath(String)方法，使之从文件系统加载资源并以FileSystemResource类型返回。这样，我们就可以取得预想的资源类型。代码清单5-3中的代码将帮助我们验证这一点。

代码清单5-3 使用FileSystemResourceLoader

```
public void testResourceTypesWithFileSystemResourceLoader()
{
    ResourceLoader resourceLoader = new FileSystemResourceLoader();
    Resource fileResource = resourceLoader.getResource("D:/spring21site/README");
    assertTrue(fileResource instanceof FileSystemResource);
    assertTrue(fileResource.exists());

    Resource urlResource = resourceLoader.getResource("file:D:/spring21site/README");
    assertTrue(urlResource instanceof UrlResource);
}
```

FileSystemResourceLoader在ResourceLoader家族中的兄弟FileSystemXmlApplicationContext，也是覆写了getResourceByPath(String)方法的逻辑，以改变DefaultResourceLoader的默认资源加载行为，最终从文件系统中加载并返回FileSystemResource类型的资源。

2. ResourcePatternResolver——批量查找的ResourceLoader

ResourcePatternResolver是ResourceLoader的扩展，ResourceLoader每次只能根据资源路径返回确定的单个Resource实例，而ResourcePatternResolver则可以根据指定的资源路径匹配模式，每次返回多个Resource实例。接口org.springframework.core.io.support.ResourcePatternResolver定义如下：

```
public interface ResourcePatternResolver extends ResourceLoader {
   String CLASSPATH_ALL_URL_PREFIX = "classpath*:";
   Resource[] getResources(String locationPattern) throws IOException;
}
```

ResourcePatternResolver在继承ResourceLoader原有定义的基础上，又引入了Resource[] getResources(String)方法定义，以支持根据路径匹配模式返回多个Resources的功能。它同时还引入了一种新的协议前缀classpath*:，针对这一点的支持，将由相应的子类实现给出。

ResourcePatternResolver最常用的一个实现是org.springframework.core.io.support.PathMatchingResourcePatternResolver，该实现类支持ResourceLoader级别的资源加载，支持基于Ant风格的路径匹配模式（类似于**/*.suffix之类的路径形式），支持ResourcePatternResolver新增加的classpath*:前缀等，基本上集所有技能于一身。

在构造PathMatchingResourcePatternResolver实例的时候，可以指定一个ResourceLoader，如果不指定的话，则PathMatchingResourcePatternResolver内部会默认构造一个DefaultResourceLoader实例。PathMatchingResourcePatternResolver内部会将匹配后确定的资源路径，委派给它的ResourceLoader来查找和定位资源。这样，如果不指定任何ResourceLoader的话，PathMatchingResourcePatternResolver在加载资源的行为上会与DefaultResourceLoader基本相同，只存在返回的Resource数量上的差异。如下代码表明了二者在资源加载行为上的一致性：

```
ResourcePatternResolver resourceResolver = new PathMatchingResourcePatternResolver();
Resource fileResource = resourceResolver.getResource("D:/spring21site/README");
assertTrue(fileResource instanceof ClassPathResource);
assertFalse(fileResource.exists());
...
```

不过，可以通过传入其他类型的ResourceLoader来替换PathMatchingResourcePatternResolver内部默认使用的DefaultResourceLoader，从而改变其默认行为。比如，可以如代码清单5-4所示，使用FileSystemResourceLoader替换默认的DefaultResourceLoader，从而使得PathMatchingResourcePatternResolver的行为跟使用FileSystemResourceLoader一样。

代码清单5-4　替换DefaultResourceLoader后的PathMatchingResourcePatternResolver

```
public void testResourceTypesWithPathMatchingResourcePatternResolver()
{
   ResourcePatternResolver resourceResolver = new PathMatchingResourcePatternResolver();
   Resource fileResource = resourceResolver.getResource("D:/spring21site/README");
   assertTrue(fileResource instanceof ClassPathResource);
   assertFalse(fileResource.exists());

   resourceResolver = new PathMatchingResourcePatternResolver(new ➥
   FileSystemResourceLoader());
   fileResource = resourceResolver.getResource("D:/spring21site/README");
   assertTrue(fileResource instanceof FileSystemResource);
   assertTrue(fileResource.exists());
}
```

3. 回顾与展望

现在我们应该对Spring的统一资源加载策略有了一个整体上的认识，就如图5-1所示。

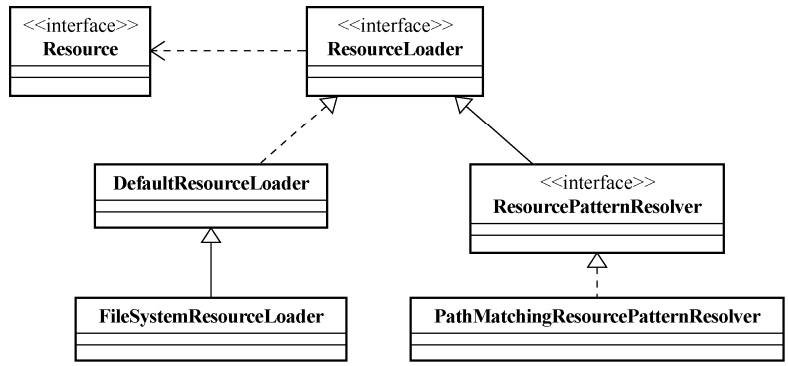


图5-1 Resource和ResourceLoader类层次图

虽然现在看来比较“单薄”，不过，稍后，我们就会发现情况并非如此了。

5.1.3 ApplicationContext与ResourceLoader

说是讲ApplicationContext的统一资源加载策略，到目前为止却一直没有涉及任何ApplicationContext相关的内容，不知道你是否开始奇怪了呢？实际上，我是有意为之，就是不想让各位因为过多关注ApplicationContext，却忽略了事情的本质。

如果回头看一下图4-2，就会发现，ApplicationContext继承了ResourcePatternResolver，当然就间接实现了ResourceLoader接口。所以，任何的ApplicationContext实现都可以看作是一个ResourceLoader甚至ResourcePatternResolver。而这就是ApplicationContext支持Spring内统一资源加载策略的真相。

通常，所有的ApplicationContext实现类会直接或者间接地继承org.springframework.context.support.AbstractApplicationContext，从这个类上，我们就可以看到ApplicationContext与ResourceLoader之间的所有关系。AbstractApplicationContext继承了DefaultResourceLoader，那么，它的getResource(String)当然就直接用DefaultResourceLoader的了。剩下需要它“效劳”的，就是ResourcePatternResolver的Resource[]getResources (String)，当然，AbstractApplicationContext也不负众望，当即拿下。AbstractApplicationContext类的内部声明有一个resourcePatternResolver，类型是ResourcePatternResolver，对应的实例类型为PathMatchingResourcePatternResolver。之前我们说过PathMatchingResourcePatternResolver构造的时候会接受一个ResourceLoader，而AbstractApplicationContext本身又继承自DefaultResourceLoader，当然就直接把自身给“贡献”了。这样，整个ApplicationContext的实现类就完全可以支持ResourceLoader或者ResourcePatternResolver接口，你能说ApplicationContext不支持Spring的统一资源加载吗？说白了，ApplicationContext的实现类在作为ResourceLoader或者ResourcePatternResolver时候的行为，完全就是委派给了PathMatchingResourcePatternResolver和DefaultResourceLoader来做。图5-2给出了AbstractApplicationContext与ResourceLoader和ResourcePatternResolver之间的类层次关系。

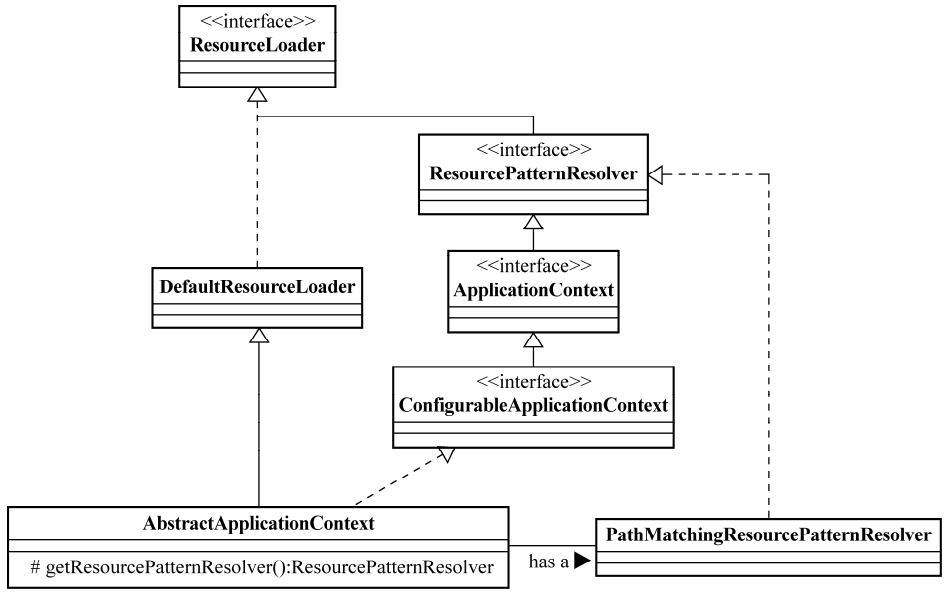


图5-2　AbstractApplicationContext作为ResourceLoader和ResourcePatternResolver

有了这些做前提，让我们看看作为ResourceLoader或者ResourcePatternResolver的ApplicationContext，到底因此拥有了何等神通吧！

1. 扮演ResourceLoader的角色

既然ApplicationContext可以作为ResourceLoader或者ResourcePatternResolver来使用，那么，很显然，我们可以通过ApplicationContext来加载任何Spring支持的Resource类型。与直接使用ResourceLoader来做这些事情相比，很明显，ApplicationContext的表现过于“谦虚”了。代码清单5-5演示的正是“大材小用”后的ApplicationContext。

代码清单5-5　以ResourceLoader身份登场的ApplicationContext

```
ResourceLoader resourceLoader = new ClassPathXmlApplicationContext("配置文件路径");
// 或者
// ResourceLoader resourceLoader = new FileSystemXmlApplicationContext("配置文件路径");

Resource fileResource = resourceLoader.getResource("D:/spring21site/README");
assertTrue(fileResource instanceof ClassPathResource);
assertFalse(fileResource.exists());

Resource urlResource2 = resourceLoader.getResource("http://www.spring21.cn");
assertTrue(urlResource2 instanceof UrlResource);
```

我想这样的使用场景，你一定比我先猜到，不是吗？

2. ResourceLoader类型的注入

在大部分情况下，如果某个bean需要依赖于ResourceLoader来查找定位资源，我们可以为其注入容器中声明的某个具体的ResourceLoader实现，该bean也无需实现任何接口，直接通过构造方法注入或者setter方法注入规则声明依赖即可，这样处理是比较合理的。不过，如果你不介意你的bean定义依赖于Spring的API，那不妨考虑用一下Spring提供的便利。

4.4.3节中曾经提到几个对ApplicationContext特定的Aware接口，这其中就包括ResourceLoaderAware和ApplicationContextAware接口。

假设我们有类定义如代码清单5-6所示。

代码清单5-6 依赖于ResourceLoader的实例类

```
public class FooBar {
   private ResourceLoader resourceLoader;

   public void foo(String location)
   {
      System.out.println(getResourceLoader().getResource(location).getClass());
   }

   public ResourceLoader getResourceLoader() {
      return resourceLoader;
   }
   public void setResourceLoader(ResourceLoader resourceLoader) {
      this.resourceLoader = resourceLoader;
   }
}
```

该类出于什么目的要依赖于ResourceLoader，我们暂且不论，要为其注入什么样的ResourceLoader实例才是我们当下该操心的事情。姑且先给它注入DefaultResourceLoader。这样也就有了如下配置：

```
<bean id="resourceLoader" class="org.springframework.core.io.DefaultResourceLoader">
</bean>

<bean id="fooBar" class="...FooBar">
   <property name="resourceLoader">
      <ref bean="resourceLoader"/>
   </property>
</bean>
```

不过，ApplicationContext容器本身就是一个ResourceLoader，我们为了该类还需要单独提供一个resourceLoader实例就有些多于了，直接将当前的ApplicationContext容器作为ResourceLoader注入不就行了？而ResourceLoaderAware和ApplicationContextAware接口正好可以帮助我们做到这一点，只不过现在的FooBar需要依赖于Spring的API了。不过，在我看来，这没有什么大不了，因为我们从来也没有真正逃脱过依赖（这种依赖也好，那种依赖也罢）。

现在，修改我们的FooBar定义，让其实现ResourceLoaderAware或者ApplicationContextAware接口，修改后的定义如代码清单5-7所示。

代码清单5-7 实现了ResourceLoaderAware或者ApplicationContextAware接口的实例类

```
public class FooBar implements ResourceLoaderAware{
   private ResourceLoader resourceLoader;

   public void foo(String location)
   {
   System.out.println(getResourceLoader().getResource(location).getClass());
   }

   public ResourceLoader getResourceLoader() {
      return resourceLoader;
   }
```

```
    public void setResourceLoader(ResourceLoader resourceLoader) {
      this.resourceLoader = resourceLoader;
    }
}
public class FooBar implements ApplicationContextAware{
    private ResourceLoader resourceLoader;

    public void foo(String location)
    {
    System.out.println(getResourceLoader().getResource(location).getClass());
    }

    public ResourceLoader getResourceLoader() {
      return resourceLoader;
    }
    public void setApplicationContext(ApplicationContext ctx) ➥
    throws BeansException {
    this.resourceLoader = ctx;
    }
}
```

剩下的就是直接将一个FooBar配置到bean定义文件即可，如下所示：

```
<bean id="fooBar" class="...FooBar">
</bean>
```

哇，简洁多了不是嘛？现在，容器启动的时候，就会自动将当前ApplicationContext容器本身注入到FooBar中，因为ApplicationContext类型容器可以自动识别Aware接口。

当然，如果应用场景仅使用ResourceLoader类型即可满足需求，那么，还是使用ResourceLoaderAware比较合适，ApplicationContextAware相对来说过于宽泛了些（当然，使用也未尝不可）。

3. Resource类型的注入

我们之前讲过，容器可以将bean定义文件中的字符串形式表达的信息，正确地转换成具体对象定义的依赖类型。对于那些Spring容器提供的默认的PropertyEditors无法识别的对象类型，我们可以提供自定义的PropertyEditor实现并注册到容器中，以供容器做类型转换的时候使用。默认情况下，BeanFactory容器不会为org.springframework.core.io.Resource类型提供相应的PropertyEditor，所以，如果我们想注入Resource类型的bean定义，就需要注册自定义的PropertyEditor到BeanFactory容器。不过，对于ApplicationContext来说，我们无需这么做，因为ApplicationContext容器可以正确识别Resource类型并转换后注入相关对象。

假设有一个XMailer类，它依赖于一个模板来提供邮件发送的内容，我们声明模板为Resource类型，那么，最终的XMailer定义也就如代码清单5-8所示。

代码清单5-8　依赖于Resource的XMailer类定义

```
public class XMailer {
    private Resource template;

    public void sendMail(Map mailCtx)
    {
      // String mailContext = merge(getTemplate().getInputStream(),mailCtx);
      //...
    }

    public Resource getTemplate() {
      return template;
```

```
    }

    public void setTemplate(Resource template) {
      this.template = template;
    }
}
```

该类定义与平常的bean定义没有什么差别，我们直接在配置文件中以String形式指定template所在位置，ApplicatonContext就可以正确地转换类型并注入依赖，配置内容如下：

```
<bean id="mailer" class="...XMailer">
  <property name="template" value="..resources.default_template.vm"/>
  ...
</bean>
```

至于这里面的奥秘，估计你也猜个八九不离十了。

ApplicationContext启动伊始，会通过一个org.springframework.beans.support.ResourceEditorRegistrar来注册Spring提供的针对Resource类型的PropertyEditor实现到容器中，这个PropertyEditor叫做org.springframework.core.io.ResourceEditor。这样，ApplicationContext就可以正确地识别Resource类型的依赖了。至于ResourceEditor怎么实现我就不用说了吧？你想啊，把配置文件中的路径让ApplicationContext作为ResourceLoader给你定位一下不就得了。

注意 如果应用对象需要依赖一组Resource，与ApplicationContext注册了ResourceEditor类似，Spring提供了org.springframework.core.io.support.ResourceArrayPropertyEditor实现，我们只需要通过CustomEditorConfigurar告知容器即可。

4. 在特定情况下，ApplicationContext的Resource加载行为

特定的ApplicationContext容器实现，在作为ResourceLoader加载资源时，会有其特定的行为。我们下面主要讨论两种类型的ApplicationContext容器，即ClassPathXmlApplicationContext和FileSystemXmlApplicationContext。其他类型的ApplicationContext容器，会在稍后章节中提到。

我们知道，对于URL所接受的资源路径来说，通常开始都会有一个协议前缀，比如file:、http:、ftp:等。既然Spring使用UrlResource对URL定位查找的资源进行了抽象，那么，同样也支持这样类型的资源路径，而且，在这个基础上，Spring还扩展了协议前缀的集合。ResourceLoader中增加了一种新的资源路径协议——classpath:，ResourcePatternResolver又增加了一种——classpath*:。这样，我们就可以通过这些资源路径协议前缀，明确地告知Spring容器要从classpath中加载资源，如下所示：

```
// 代码中使用协议前
ResourceLoader resourceLoader = new ➥
FileSystemXmlApplicationContext("classpath:conf/container-conf.xml");
// 配置中使用协议前缀
<bean id="..." class="...">
  <property name="...">
    <value>classpath:resource/template.vm</value>
  </property>
</bean>
```

classpath*:与classpath:的唯一区别就在于，如果能够在classpath中找到多个指定的资源，则返回多个。我们可以通过这两个前缀改变某些ApplicationContext实现类的默认资源加载行为。

ClassPathXmlApplicationContext和FileSystemXmlApplicationContext在处理资源加载的

默认行为上有所不同。当ClassPathXmlApplicationContext在实例化的时候，即使没有指明classpath:或者classpath*:等前缀，它会默认从classpath中加载bean定义配置文件，以下代码中演示的两种实例化方式效果是相同的：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
以及
ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:conf/appContext.xml");
```

而FileSystemXmlApplicationContext则有些不同，如果我们像如下代码那样指定conf/appContext.xml，它会尝试从文件系统中加载bean定义文件：

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("conf/appContext.xml");
```

不过，我们可以像如下代码所示，通过在资源路径之前增加classpath:前缀，明确指定FileSystemXmlApplicationContext从classpath中加载bean定义的配置文件：

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

这时，FileSystemXmlApplicationContext就是从Classpath中加载配置，而不是从文件系统中加载。也就是说，它现在对应的是ClassPathResource类型的资源，而不是默认的FileSystemResource类型资源。FileSystemXmlApplicationContext之所以如此，是因为它与org.springframework.core.io.FileSystemResourceLoader一样，也覆写了DefaultResourceLoader的getResourceByPath(String)方法，逻辑跟 FileSystemResourceLoader一模一样。

当实例化相应的ApplicationContext时，各种实现会根据自身的特性，从不同的位置加载bean定义配置文件。当容器实例化并启动完毕，我们要用相应容器作为ResourceLoader来加载其他资源时，各种ApplicationContext容器的实现类依然会有不同的表现。

对于ClassPathXmlApplicationContext来说，如果我们不指定路径之前的前缀，它也不会像资源路径所表现的那样，从文件系统加载资源，而是像实例化时候的行为一样，从Classpath中加载这种没有路径前缀的资源。如类似如下指定的资源路径，ClassPathXmlApplicationContext依然尝试从Classpath加载：

```
<bean id="..." class="...">
   <property name="..." value="conf/appContext.xml"/>
</bean>
```

如果当前容器类型为FileSystemXmlApplicationContext，事情则会像预想的那样进行，FileSystemXmlApplicationContext将从文件系统中给我们加载该文件。但是，就跟实例化时可以通过classpath:前缀覆盖掉FileSystemXmlApplicationContext的默认加载行为一样，我们也可以在这个时候用classpath:前缀强制指定FileSystemXmlApplicationContext从Classpath中加载该文件，如以下代码所示：

```
<bean id="..." class="...">
   <property name="..." value="classpath:conf/appContext.xml"/>
</bean>
```

去掉配置中的classpath:前缀，FileSystemXmlApplicationContext默认从文件系统加载资源。

小心　即使在FileSystemXmlApplicationContext实例化启动时，通过classpath:前缀强制让它从Classpath中加载bean定义文件，但这也仅限于容器的实例化并加载bean定义文件这个特

定阶段。容器实例化并启动后，作为ResourceLoader来加载资源，如果不是每个地方都使用classpath:前缀，强制FileSystemXmlApplicationContext从Classpath中加载资源，FileSystemXmlApplicationContext还会默认从文件系统中加载资源。

如果细化下去，这部分内容还有许多，如通配符加载的行为、FileSystemResource的特定行为等。这里不做赘述，更多相关特性，请参照Spring参考文档。

5.2 国际化信息支持（I18n[①] MessageSource）

全世界有很多不同的国家和地区，每个国家或者地区都使用各自的语言文字。在当今全球化的信息大潮中，要让我们的应用程序可以供全世界不同国家和地区的人们使用，应用程序就必须支持它所面向的国家和地区的语言文字，为不同的国家和地区的用户提供他们各自的语言文字信息。所以，要向全世界推广，应用程序的国际化信息支持自然是势在必行。

5.2.1 Java SE 提供的国际化支持

程序的国际化不是三言两语可以讲清楚的，它涉及许多的内容，如货币形式的格式化、时间的表现形式、各国家和地区的语言文字等。要全面了解Java中的I18n，建议参考O'Reilly出版的*Java Internationalization*。我们这里主要是简单地介绍基本概念，以便你可以对Spring中的国际化信息支持有更好的了解。

对于Java中的国际化信息处理，主要涉及两个类，即java.util.Locale和java.util.ResourceBundle。

1. Locale

不同的Locale代表不同的国家和地区，每个国家和地区在Locale这里都有相应的简写代码表示，包括语言代码以及国家代码，这些代码是ISO标准代码。如，Locale.CHINA代表中国，它的代码表示为zh_CN；Locale.US代表美国地区，代码表示为en_US；而美国和英国等都属于英语地区，则可以使用Locale.ENGLISH来统一表示，这时代码只有语言代码，即en。

Locale类提供了三个构造方法，它们的定义如下：

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

这样我们就可以根据相应的语言代码和国家代码来构造相应的Locale，如以下所示：

```
Locale china = new Locale("zh","CN");
相当于
Locale.CHINA
```

常用的Locale都提供有静态常量，不用我们自己重新构造。一些不常用的Locale的则需要根据相应的国家和地区以及语言来进行构造。有了Locale，我们的应用程序就可以通过它来判别如何为不同的国家和地区的用户提供相应的信息。

2. ResourceBundle

ResourceBundle用来保存特定于某个Locale的信息（可以是String类型信息，也可以是任何类型的对象）。通常，ResourceBundle管理一组信息序列，所有的信息序列有统一的一个basename，然后特定的Locale的信息，可以根据basename后追加的语言或者地区代码来区分。比如，我们用一组

① Internationalization，单词的开头字母I和结尾字母n中间有18个字母，所以，通常简写为I18n。

properties文件来分别保存不同国家地区的信息，可以像下面这样来命名相应的properties文件：

```
messages.properties
messages_zh.properties
messages_zh_CN.properties
messages_en.properties
messages_en_US.properties
...
```

其中，文件名中的messages部分称作ResourceBundle将加载的资源的basename，其他语言或地区的资源在basename的基础上追加Locale特定代码。

每个资源文件中都有相同的键来标志具体资源条目，但每个资源内部对应相同键的资源条目内容，则根据Locale的不同而不同。如下代码片段演示了两个不同的资源文件内容的对比情况：

```
# messages_zh_CN.properties文件中
menu.file=文件({0})
menu.edit=编辑
...

# messages_en_US.properties文件中
menu.file=File({0})
menu.edit=Edit
...
```

注意　按照规定，properties文件内容是以ISO-8859-1编码的，所以，实际上message_zh_CN.properties中各个键对应的内容是不应该以中文提供的，应该使用native2ascii或者类似的相关工具进行转码，这里如此举例，只是为了更好地说明差别。

有了ResourceBundle对应的资源文件之后，我们就可以通过ResourceBundle的getBundle(String baseName, Locale locale)方法取得不同Locale对应的ResourceBundle，然后根据资源的键取得相应Locale的资源条目内容。

通过结合ResourceBundle和Locale，我们就能够实现应用程序的国际化信息支持。

5.2.2　MessageSource与ApplicationContext

Spring在Java SE的国际化支持的基础上，进一步抽象了国际化信息的访问接口，也就是org.springframework.context.MessageSource，该接口定义如下：

```
public interface MessageSource {

   String getMessage(String code, Object[] args, String defaultMessage, Locale locale);

   String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessageException;

   String getMessage(MessageSourceResolvable resolvable, Locale locale) throws ➥
   NoSuchMessage zException;

}
```

通过该接口，我们统一了国际化信息的访问方式。传入相应的Locale、资源的键以及相应参数，就可以取得相应的信息，再也不用先根据Locale取得ResourceBundle，然后再从ResourceBundle查询信息了。对MessageSource所提供的三个方法的简单说明如下。

- String getMessage(String code, Object[] args, String defaultMessage, Locale locale)。根据传入的资源条目的键（对应方法声明中的code参数）、信息参数以及Locale来

查找信息，如果对应信息没有找到，则返回指定的defaultMessage。

- String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessageException。与第一个方法相同，只不过，因为没有指定默认信息，当对应的信息找不到的情况下，将抛出NoSuchMessageException异常。
- String getMessage(MessageSourceResolvable resolvable, Locale locale) throws NoSuchMessageException。使用MessageSourceResolvable对象对资源条目的键、信息参数等进行封装，将封住了这些信息的MessageSourceResolvable对象作为查询参数来调用以上方法。如果根据MessageSourceResolvable中的信息查找不到相应条目内容，将抛出NoSuchMessageException异常。

现在我们知道，ApplicationContext除了实现了ResourceLoader以支持统一的资源加载，它还实现了MessageSource接口，那么就跟ApplicationContext因为实现了ResourceLoader而可以当作ResourceLoader来使用一样，ApplicationContext现在也是一个MessageSource了。

在默认情况下，ApplicationContext将委派容器中一个名称为messageSource的MessageSource接口实现来完成MessageSource应该完成的职责。如果找不到这样一个名字的MessageSource实现，ApplicationContext内部会默认实例化一个不含任何内容的StaticMessageSource实例，以保证相应的方法调用。所以通常情况下，如果要提供容器内的国际化信息支持，我们会添加如代码清单5-9类似的配置信息到容器的配置文件中。

代码清单5-9 ApplicationContext容器内使用的messageSource的配置实例

```
<beans>
  <bean id="messageSource" class="org.springframework.context.support. ➥
  ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>messages</value>
        <value>errorcodes</value>
      </list>
    </property>
  </bean>
  ...
</beans>
```

有了这些，我们就可以通过ApplicationContext直接访问相应Locale对应的信息，如下所示：

```
ApplicationContext ctx = ...;
String fileMenuName = ctx.getMessage("menu.file", new Object[]{"F"}, Locale.US);
String editMenuName = ctx.getMessage("menu.file", null, Locale.US);
assertEquals("File(F)", fileMenuName);
assertEquals("Edit", editMenuName);
```

1. 可用的MessageSource实现

Spring提供了三种MessageSource的实现，即StaticMessageSource、ResourceBundleMessageSource和ReloadableResourceBundleMessageSource。

- org.springframework.context.support.StaticMessageSource。MessageSource接口的简单实现，可以通过编程的方式添加信息条目，多用于测试，不应该用于正式的生产环境。
- org.springframework.context.support.ResourceBundleMessageSource。基于标准的java.util.ResourceBundle而实现的MessageSource，对其父类AbstractMessageSource的行为进行了扩展，提供对多个ResourceBundle的缓存以提高查询速度。同时，对于参数化的信息和非参数化信息的处理进行了优化，并对用于参数化信息格式化的MessageFormat实

例也进行了缓存。它是最常用的、用于正式生产环境下的MessageSource实现。

- org.springframework.context.support.ReloadableResourceBundleMessageSource。同样基于标准的java.util.ResourceBundle而构建的MessageSource实现类，但通过其cacheSeconds属性可以指定时间段，以定期刷新并检查底层的properties资源文件是否有变更。对于properties资源文件的加载方式也与ResourceBundleMessageSource有所不同，可以通过ResourceLoader来加载信息资源文件。使用ReloadableResourceBundleMessageSource时，应该避免将信息资源文件放到classpath中，因为这无助于ReloadableResourceBundleMessageSource定期加载文件变更。更多信息参照该类的Javadoc。

这三种实现都可以独立于容器并在独立运行（Standalone形式）的应用程序中使用，而并非只能依托ApplicationContext才可使用。代码清单5-10为我们演示了这三种MessageSource的简单使用。

代码清单5-10　三种MessageSource实现类的简单使用演示

```
StaticMessageSource messageSource = new StaticMessageSource();
messageSource.addMessage("menu.file", Locale.US, "File");
messageSource.addMessage("menu.edit", Locale.US, "Edit");
assertEquals("File(F)", messageSource.getMessage("menu.file", new Object[]{"F"}, Locale.US));
assertEquals("Edit", messageSource.getMessage("menu.edit", null,"Edit", Locale.US));

ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
messageSource.setBasenames(new String[]{"conf/messages"});// 从 classpath加载资源文件
assertEquals("File(F)", messageSource.getMessage("menu.file", new Object[]{"F"}, Locale.US));

ReloadableResourceBundleMessageSource messageSource = new ➥
ReloadableResourceBundleMessageSource();
messageSource.setBasenames(new String[]{"file:conf/messages"}); // 从文件系统加载资源文件
assertEquals("File(F)", messageSource.getMessage("menu.file", new Object[]{"F"},➥
Locale.US));
```

之前提到，ApplicationContext需要其配置文件中有一个名称为messageSource的MessageSource实现，自然就是以上的三选一了。

至此，我们有了图5-3。

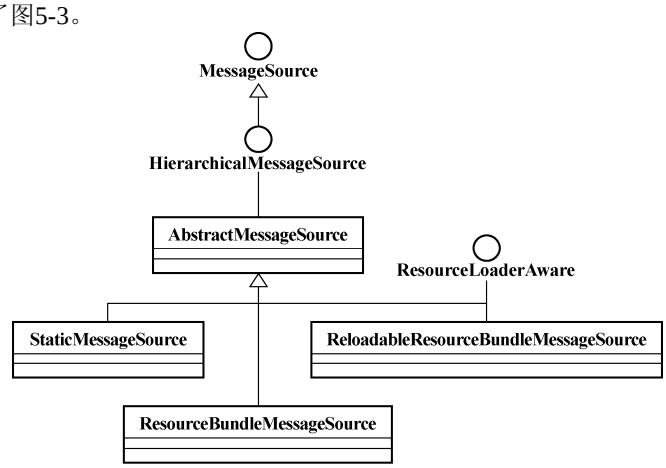


图5-3　MessageSource类层次结构

如果说以上三种MessageSource的实现还是不能满足你的要求，那么直接继承Abstract-

MessageSource，然后覆写几个方法就行了，甚至可以直接实现MessageSource接口，如果你的需求真的那么特别。

2. MessageSourceAware和MessageSource的注入

ApplicationContext启动的时候，会自动识别容器中类型为MessageSourceAware的bean定义，并将自身作为MessageSource注入相应对象实例中。如果某个业务对象需要国际化的信息支持，那么最简单的办法就是让它实现MessageSourceAware接口，然后注册到ApplicationContext容器。不过这样一来，该业务对象对ApplicationContext容器的依赖性就太强了，显得容器具有较强的侵入性。

而实际上，如果真的某个业务对象需要依赖于MessageSource的话，直接通过构造方法注入或者setter方法注入的方式声明依赖就可以了。只要配置bean定义时，将ApplicationContext容器内部的那个messageSource注入该业务对象即可。假设我们有一个通用的Validator数据验证类，它需要通过MessageSource来返回相应的错误信息，那么可以为其声明一个MessageSource依赖，然后将ApplicationContext中的那个已经配置好的messageSource注入给它。代码清单5-11给出了该类的定义以及相关注入配置。

代码清单5-11 依赖于MessageSource的Validator类定义以及相关注入配置

```
public class Validator
{
   private MessageSource messageSource;

   public ValidateResult validate(Object target)
   {
      // 执行相应验证逻辑
      // 如果有错误，通过messageSource.getMessage(...)获取相应信息并放入验证结果对象中
      // 返回验证结果（return result）
   }
   public MessageSource getMessageSource()
   {
      return messageSource;
   }
   public void setMessageSource(MessageSource msgSource)
   {
      this.messageSource = msgSource;
   }
   // ...
}

<beans>
<bean id="messageSource" ➥
class="org.springframework.context.support.ResourceBundleMessageSource">
   <property name="basenames">
      <list>
         <value>messages</value>
         <value>errorcodes</value>
      </list>
   </property>
</bean>

<bean id="validator" class="...Validator">
   <property name="messageSource" ref="messageSource"/>
</bean>
...
</beans>
```

与通常的依赖注入没有什么区别，不是吗？

既然MessageSource可以独立使用，那为什么还让ApplicationContext实现该接口呢？在独立运行的应用程序（Standalone Application）中，就如我们上面这些应用场景所展示的那样，直接使用MessageSource的相应实现类就行了。不过在Web应用程序中，通常会公开ApplicationContext给视图（View）层，这样，通过标签（tag）就可以直接访问国际化信息了。我们将在第六部分更多接触相关内容，至此，ApplicationContext对国际化信息的支持功能的讲解告一段落。

5.3　容器内部事件发布

Spring的ApplicationContext容器提供的容器内事件发布功能，是通过提供一套基于Java SE标准自定义事件类而实现的。为了更好地了解这组自定义事件类，我们可以先从Java SE的标准自定义事件类实现的推荐流程说起。

5.3.1　自定义事件发布

Java SE提供了实现自定义事件发布（Custom Event publication）功能的基础类，即java.util.EventObject类和java.util.EventListener接口。所有的自定义事件类型可以通过扩展EventObject来实现，而事件的监听器则扩展自EventListener。下面让我们看一下要实现一套自定义事件发布类的架构，应该如何来做。

给出自定义事件类型（define your own event object）。为了针对具体场景可以区分具体的事件类型，我们需要给出自己的事件类型的定义，通常做法是扩展java.util.EventObject类来实现自定义的事件类型。我们此次定义的自定义事件类型见代码清单5-12。

代码清单5-12　针对方法执行事件的自定义事件类型定义

```
public class MethodExecutionEvent extends EventObject {
   private static final long serialVersionUID = -71960369269303337L;
   private String methodName;

   public MethodExecutionEvent(Object source) {
      super(source);
   }
   public MethodExecutionEvent(Object source,String methodName)
   {
      super(source);
      this.methodName = methodName;
   }
   public String getMethodName() {
      return methodName;
   }
   public void setMethodName(String methodName) {
      this.methodName = methodName;
   }
}
```

我们想对方法的执行情况进行发布和监听，所以，就声明了一个MethodExecutionEvent类型，它继承自EventObject，当该类型的事件发布之后，相应的监听器即可对该类型的事件进行处理。如果需要，自定义事件类可以根据情况提供更多信息，不用担心自定义事件类的“承受力”。

实现针对自定义事件类的事件监听器接口（define custom event listener）。自定义的事件监听器需要在合适的时机监听自定义的事件，如刚声明的MethodExecutionEvent，我们可以在方法开始执行的时候发布该事件，也可以在方法执行即将结束之际发布该事件。相应地，自定义的事件监听器

需要提供方法对这两种情况下接收到的事件进行处理。代码清单5-13给出了针对MethodExecutionEvent的事件监听器接口定义。

代码清单5-13 自定义事件监听器MethodExecutionEventListener定义

```
public interface MethodExecutionEventListener extends EventListener {
   /**
    * 处理方法开始执行的时候发布的MethodExecutionEvent事件
    */
   void onMethodBegin(MethodExecutionEvent evt);
   /**
    * 处理方法执行将结束时候发布的MethodExecutionEvent事件
    */
   void onMethodEnd(MethodExecutionEvent evt);
}
```

事件监听器接口定义首先继承了java.util.EventListener,然后针对不同的事件发布时机提供相应的处理方法定义，最主要的就是，这些处理方法所接受的参数就是MethodExecutionEvent类型的事件。也就是说，我们的自定义事件监听器类只负责监听其对应的自定义事件并进行处理，如果什么事件它都要处理，那么非忙死不可。有了事件监听器接口定义，还必须根据时机需求提供相应的实现，只有接口定义可是干不了什么事情的啊！出于简化，我们仅给出一个简单的实现定义，见代码清单5-14。

代码清单5-14 自定义事件监听器具体实现类SimpleMethodExecutionEventListener的定义

```
public class SimpleMethodExecutionEventListener implements ➥
MethodExecutionEventListener {

   public void onMethodBegin(MethodExecutionEvent evt) {
      String methodName = evt.getMethodName();
      System.out.println("start to execute the method["+methodName+"].");
   }

   public void onMethodEnd(MethodExecutionEvent evt) {
      String methodName = evt.getMethodName();
      System.out.println("finished to execute the method["+methodName+"].");
   }
}
```

组合事件类和监听器，发布事件。有了自定义事件和自定义事件监听器，剩下的就是发布事件，然后让相应的监听器监听并处理事件了。通常情况下，我们会有一个事件发布者（EventPublisher），它本身作为事件源，会在合适的时点，将相应事件发布给对应的事件监听器。代码清单5-15给出了针对MethodExecutionEvent的事件发布者类的定义。

代码清单5-15 MethodExeuctionEventPublisher时间发布者类定义

```
public class MethodExeuctionEventPublisher {

   private List<MethodExecutionEventListener> listeners = new ➥
   ArrayList<MethodExecutionEventListener>();

   public void methodToMonitor()
   {
      MethodExecutionEvent event2Publish = ➥
      new MethodExecutionEvent(this,"methodToMonitor");
      publishEvent(MethodExecutionStatus.BEGIN,event2Publish);
```

```
        // 执行实际的方法逻辑
        // ...
        publishEvent(MethodExecutionStatus.END,event2Publish);
    }

    protected void publishEvent(MethodExecutionStatus status, ➥
    MethodExecutionEvent methodExecutionEvent) {
        List<MethodExecutionEventListener> copyListeners = ➥
        new ArrayList<MethodExecutionEventListener>(listeners);
        for(MethodExecutionEventListener listener:copyListeners)
        {
            if(MethodExecutionStatus.BEGIN.equals(status))
                listener.onMethodBegin(methodExecutionEvent);
            else
                listener.onMethodEnd(methodExecutionEvent);
        }
    }

    public void addMethodExecutionEventListener(MethodExecutionEventListener listener)
    {
        this.listeners.add(listener);
    }
    public void removeListener(MethodExecutionEventListener listener)
    {
        if(this.listeners.contains(listener))
            this.listeners.remove(listener);
    }
    public void removeAllListeners()
    {
        this.listeners.clear();
    }

    public static void main(String[] args) {
        MethodExeuctionEventPublisher eventPublisher = ➥
        new MethodExeuctionEventPublisher();
        eventPublisher.addMethodExecutionEventListener(new ➥
        SimpleMethodExecutionEventListener());
        eventPublisher.methodToMonitor();
    }
}
```

我们的事件发布者关注的主要有两点。

具体时点上自定义事件的发布。方法methodToMonitor()是事件发布的源头，MethodExeuction-EventPublisher在该方法开始和即将结束的时候，分别针对这两个时点发布MethodExecutionEvent事件。具体实现上，每个时点发布的事件会通过MethodExecutionEventListener的相应方法传给注册的监听者并被处理掉。在实现中，需要注意到，为了避免事件处理期间事件监听器的注册或移除操作影响处理过程，我们对事件发布时点的监听器列表进行了一个安全复制（safe-copy）。另外，事件的发布是顺序执行，所以为了能够不影响处理性能，事件监听器的处理逻辑应该尽量简短。

自定义事件监听器的管理。MethodExeuctionEventPublisher类提供了与事件监听器的注册和移除相关的方法，这样，客户端可以根据情况决定是否需要注册或者移除某个事件监听器。这里容易出现问题的情况是，如果没有提供remove事件监听器的方法，那么注册的监听器实例会一直被MethodExeuctionEventPublisher引用，即使已经过期了或者废弃不用了，也依然存在于MethodExeuctionEventPublisher的监听器列表中。这会导致隐性的内存泄漏，在任何事件监听器的处理上都可能出现这种问题。

整个Java SE中标准的自定义事件实现就是这个样子，基本上涉及三个角色，即自定义的事件类型、自定义的事件监听器和自定义的事件发布者，关系如图5-4所示。

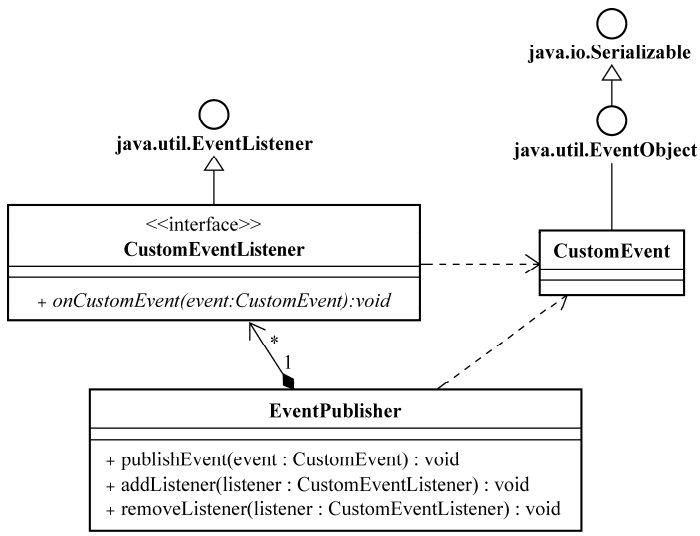


图5-4 自定义事件发布类结构图

5.3.2 Spring 的容器内事件发布类结构分析

Spring 的 ApplicationContext 容器内部允许以 org.springframework.context.ApplicationEvent 的形式发布事件，容器内注册的 org.springframework.context.ApplicationListener类型的bean定义会被ApplicationContext容器自动识别，它们负责监听容器内发布的所有ApplicationEvent类型的事件。也就是说，一旦容器内发布ApplicationEvent及其子类型的事件，注册到容器的ApplicationListener就会对这些事件进行处理。

我想你已经猜到是怎么回事了。

- ApplicationEvent

Spring容器内自定义事件类型，继承自java.util.EventObject，它是一个抽象类，需要根据情况提供相应子类以区分不同情况。默认情况下，Spring提供了三个实现。

- ContextClosedEvent：ApplicationContext容器在即将关闭的时候发布的事件类型。
- ContextRefreshedEvent：ApplicationContext容器在初始化或者刷新的时候发布的事件类型。
- RequestHandledEvent：Web请求处理后发布的事件，其有一子类ServletRequestHandledEvent提供特定于Java EE的Servlet相关事件。

- ApplicationListener

ApplicationContext容器内使用的自定义事件监听器接口定义，继承自java.util.EventListener。ApplicationContext容器在启动时，会自动识别并加载EventListener类型bean定义，一旦容器内有事件发布，将通知这些注册到容器的EventListener。

- ApplicationContext

还记得ApplicationContext的定义吧？除了之前的ResourceLoader和MessageSource，ApplicationContext接口定义还继承了ApplicationEventPublisher接口，该接口提供了void publishEvent(ApplicationEvent event)方法定义。不难看出，ApplicationContext容器现在担当的就是事件发布者的角色。

虽然ApplicationContext继承了ApplicationEventPublisher接口而担当了事件发布者的角色，但是在具体实现上，与之前提到的自定义事件实现流程有些许差异，且让我一一道来……

ApplicationContext容器的具体实现类在实现事件的发布和事件监听器的注册方面，并没事必躬亲，而是把这些活儿转包给了一个称作org.springframework.context.event.ApplicationEventMulticaster的接口。该接口定义了具体事件监听器的注册管理以及事件发布的方法，但接口终归是接口，还得有具体实现。ApplicationEventMulticaster有一抽象实现类——org.springframework.context.event.AbstractApplicationEventMulticaster，它实现了事件监听器的管理功能。出于灵活性和扩展性考虑，事件的发布功能则委托给了其子类。org.springframework.context.event.SimpleApplicationEventMulticaster是Spring提供的AbstractApplicationEventMulticaster的一个子类实现，添加了事件发布功能的实现。不过，其默认使用了SyncTaskExecutor进行事件的发布。与我们给出的样例事件发布者实现一样，事件是同步顺序发布的。为了避免这种方式可能存在的性能问题，我们可以为其提供其他类型的TaskExecutor实现类（TaskExecutor的概念将在后面详细介绍）。

因为ApplicationContext容器的事件发布功能全部委托给了ApplicationEventMulticaster来做，所以，容器启动伊始，就会检查容器内是否存在名称为applicationEventMulticaster的ApplicationEventMulticaster对象实例。有的话就使用提供的实现，没有则默认初始化一个SimpleApplicationEventMulticaster作为将会使用的ApplicationEventMulticaster。这样，整个Spring容器内事件发布功能实现结构图就有了，如图5-5所示。

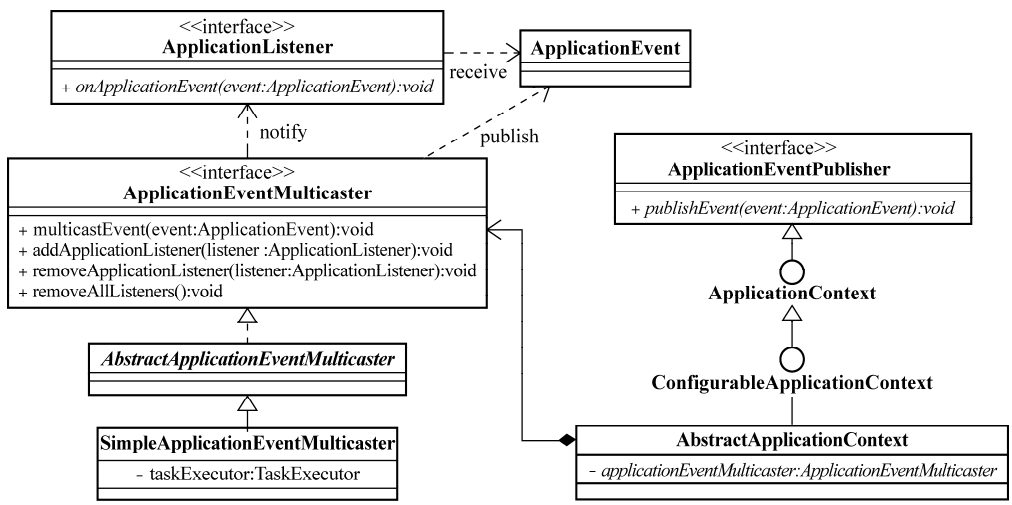


图5-5　Spring容器内事件发布实现类图

5.3.3　Spring 容器内事件发布的应用

Spring的ApplicationContext容器内的事件发布机制，主要用于单一容器内的简单消息通知和处理，并不适合分布式、多进程、多容器之间的事件通知。虽然可以通过Spring的Remoting支持，“曲折一点”来实现较为复杂的需求，但是难免弊大于利，失大于得。其他消息机制处理较复杂场景或许更合适。所以，我们应该在合适的地点、合适的需求分析的前提下，合理地使用Spring提供的ApplicationContext容器内的事件发布机制。

要让我们的业务类支持容器内的事件发布，需要它拥有ApplicationEventPublisher的事件发布支持。所以，需要为其注入ApplicationEventPublisher实例。可以通过如下两种方式为我们的业务对象注入ApplicationEventPublisher的依赖。

- 使用ApplicationEventPublisherAware接口。在ApplicationContext类型的容器启动时，会自动识别该类型的bean定义并将ApplicationContext容器本身作为ApplicationEventPublisher注入当前对象，而ApplicationContext容器本身就是一个ApplicationEventPublisher。
- 使用ApplicationContextAware接口。既然ApplicationContext本身就是一个ApplicationEventPublisher，那么通过ApplicationContextAware几乎达到第一种方式相同的效果。

下面，我们把之前的MethodExecutionEvent相关类改装一下，也好看看改装成使用容器内的事件发布到底是个什么样子。

1. MethodExecutionEvent的改装

因为ApplicationListener只通过void onApplicationEvent(ApplicationEvent event)这一个事件处理方法来处理事件，所以现在要在事件类中尽量保存必要的信息。改装后的MethodExecutionEvent类定义如代码清单5-16所示。

代码清单5-16 改装后的MethodExecutionEvent类定义

```
public class MethodExecutionEvent extends ApplicationEvent {
   private static final long serialVersionUID = -71960369269303337L;
   private String methodName;
   private MethodExecutionStatus methodExecutionStatus;

   public MethodExecutionEvent(Object source) {
      super(source);
   }
   public MethodExecutionEvent(Object source,String methodName, ➥
   MethodExecutionStatus methodExecutionStatus)
   {
      super(source);
      this.methodName = methodName;
      this.methodExecutionStatus = methodExecutionStatus;
   }
   public String getMethodName() {
      return methodName;
   }
   public void setMethodName(String methodName) {
      this.methodName = methodName;
   }
   public MethodExecutionStatus getMethodExecutionStatus() {
      return methodExecutionStatus;
   }
   public void setMethodExecutionStatus(MethodExecutionStatus methodExecutionStatus) {
      this.methodExecutionStatus = methodExecutionStatus;
```

```
    }
}
```

2. **MethodExecutionEventListener**

我们的MethodExecutionEventListener不再是接口，而是具体的ApplicationListener实现类。因为ApplicationListener已经取代了MethodExecutionEventListener原来的角色，所以，改装后的MethodExecutionEventListener定义如下：

```
public class MethodExecutionEventListener implements ApplicationListener {

   public void onApplicationEvent(ApplicationEvent evt) {
      if(evt instanceof MethodExecutionEvent)
      {
         // 执行处理逻辑
      }
   }
}
```

3. **MethodExeuctionEventPublisher改造**

MethodExeuctionEventPublisher改造后如代码清单5-17所示。

代码清单5-17　改造后的MethodExeuctionEventPublisher定义

```
public class MethodExeuctionEventPublisher implements ApplicationEventPublisherAware {
   private ApplicationEventPublisher eventPublisher;

   public void methodToMonitor()
   {
      MethodExecutionEvent beginEvt = new ➥
      MethodExecutionEvent(this,"methodToMonitor",MethodExecutionStatus.BEGIN);
      this.eventPublisher.publishEvent(beginEvt);
      // 执行实际方法逻辑
      // ...
      MethodExecutionEvent endEvt = new ➥
      MethodExecutionEvent(this,"methodToMonitor",MethodExecutionStatus.END);
      this.eventPublisher.publishEvent(endEvt);
   }

   public void setApplicationEventPublisher(ApplicationEventPublisher appCtx) {
      this.eventPublisher = appCtx;
   }
}
```

现在，直接使用注入的eventPublisher来发布事件，而不用自己实现事件发布逻辑了。需要注意的就是，我们实现了ApplicationEventPublisherAware接口（当然，ApplicationContextAware也是可以的）。

4. **注册到ApplicationContext容器**

最后一步工作就是将MethodExeuctionEventPublisher和MethodExecutionEventListener注册到ApplicationContext容器中。当MethodExeuctionEventPublisher的methodToMonitor方法被调用时，事件即被发布。配置如下所示：

```
<bean id="methodExecListener" class="...MethodExecutionEventListener">
</bean>

<bean id="evtPublisher" class="...MethodExeuctionEventPublisher">
</bean>
```

整个改造就此宣告结束！

提示 你可能觉得我们的实例没有任何实际意义。不过，我想提醒的是，如果你尝试在每次发布事件的时候，将当前系统时间或者其他信息也通过MethodExecutionEvent传给具体的ApplicationListener处理，情况是否有所改观呢？你完全可以通过这样的方式来监控系统性能了！要知道，完全可以在这个的基础上往简单的AOP迈进哦！

Spring的容器内事件发布机制初步看来无法脱离容器单独使用。不过，要想做，也不是不可以的，只不过是直接使用ApplicationEventMulticaster接口进行事件发布而已。就提这一句吧，你如果有兴趣可以自己尝试一下。

5.4 多配置模块加载的简化

实际上，这也不算ApplicationContext比较突出的特色功能，只是相对于BeanFactory来说，在这一点上做得更好罢了。

我们知道，在使用Spring的IoC轻量级容器进行实际开发的过程中，为了避免出现整个团队因某个资源独占而无法并行、高效地完成工作等问题，通常会将整个系统的配置信息按照某种关注点进行分割，使得关注点逻辑良好地划分到不同的配置文件中，如按照功能模块或者按照系统划分的层次等。这样，在加载整个系统的bean定义时，就需要让容器同时读入划分到不同配置文件的信息。相对于BeanFactory来说，ApplicationContext大大简化了这种情况下的多配置文件的加载工作。

假设在文件系统中存在多个Spring的配置文件，它们所在路径如下所示：

```
{user.dir}/conf/dao-tier.springxml
{user.dir}/conf/view-tier.springxml
{user.dir}/conf/business-tier.springxml
...
```

通过ApplicationContext，我们只要以String[]形式传入这些配置文件所在的路径，即可构造并启动容器，如代码清单5-18所示。

代码清单5-18 使用ApplicationContext加载多个配置文件

```
String[] locations = new String[]{ "conf/dao-tier.springxml", ➥
"conf/view-tier.springxml", "conf/business-tier.springxml"};
ApplicationContext container = new FileSystemXmlApplicationContext(locations);
// 或者
ApplicationContext container = new ClassPathXmlApplicationContext(locations);
...

甚至于使用通配符
ApplicationContext container = new FileSystemXmlApplicationContext("conf/**/*.springxml");
...
```

而使用BeanFactory来加载这些配置，则需要动用过多的代码，如以下代码所示：

```
BeanFactory parentFactory = new XmlBeanFactory➥
(new FileSystemResource("conf/dao-tier.springxml"));
BeanFactory subFactory = new XmlBeanFactory➥
(new FileSystemResource("conf/view-tier.springxml"),parentFactory);
BeanFactory subsubFactory = new XmlBeanFactory➥
(new FileSystemResource("conf/business-tier.springxml"),subFactory);
...
```

```
BeanFactory container = new XmlBeanFactory(new FileSystemResource("..."),sub...Factory);
...
```

当然，我只是故意给出了一个比较损的对比。实际上，如果通过在某一个主配置文件中使用<import>分别加载其余的配置文件，然后容器就可以通过加载这个主配置文件，来加载其他的配置文件了。但使用<import>的问题，在于需要时刻关注主配置文件与其他配置文件的一致性。

除了可以批量加载配置文件之外，ClassPathXmlApplicationContext还可以通过指定Classpath中的某个类所处位置来加载相应配置文件，配置文件分布结构如下（例子来自Spring参考文档）：

```
com/
   foo/
     services.xml
     daos.xml
     MessengerService.class
```

ClassPathXmlApplicationContext可以通过MessengerService类在Classpath中的位置定位配置文件，而不用指定每个配置文件的完整路径名，如以下代码所示：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
   new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

读者可以参考各ApplicationContext实现类的Javadoc取得更多相关信息。

5.5 小结

ApplicationContext是Spring在BeanFactory基础容器之上，提供的另一个IoC容器实现。它拥有许多BeanFactory所没有的特性，包括统一的资源加载策略、国际化信息支持、容器内事件发布以及简化的多配置文件加载功能。本章对ApplicationContext的这些新增特性进行了详尽的阐述。希望读者在学习完本章内容之后，对每一种特性的来龙去脉都能了如指掌。

第 6 章

Spring IoC容器之扩展篇

本章内容

- Spring 2.5的基于注解的依赖注入
- Spring 3.0展望

6.1 Spring 2.5 的基于注解的依赖注入

Spring 2.5提供的基于注解的依赖注入功能延续了Spring框架内在IoC容器设计与实现上的一致性。除了依赖关系的“表达”方式上的不同，底层的实现机制基本上保持一致。如果我们已经从Spring的IoC容器的XML之旅中成功走过来，那么在体验基于注解的依赖注入的过程中，一定会发现许多似曾相识的身影。你瞧，基于XML配置方式的自动绑定功能，就是我们再次邂逅的第一位老朋友……

6.1.1 注解版的自动绑定（@Autowired）

1. 从自动绑定（autowire）到@Autowired

在使用依赖注入绑定FXNews相关实现类时，为了减少配置量，我们可以采用Spring的IoC容器提供的自动绑定功能，如下所示：

```
<beans default-autowire="byType">
   <bean id="newsProvider"    class="..FXNewsProvider" autowire="byType"/>
   <bean id="djNewsListener"  class="..DowJonesNewsListener"/>
   <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

可以通过default-autowire来指定默认的自动绑定方式，也可以通过每个bean定义上的autowire来指定每个bean定义各自的自动绑定方式，它们都是触发容器对相应对象给予依赖注入的标志。而将自动绑定的标志用注解来表示时，也就得到了基于注解的依赖注入，或者更确切地称为基于注解的自动绑定。

@Autowired是基于注解的依赖注入的核心注解，它的存在可以让容器知道需要为当前类注入哪些依赖。比如可以使用@Autowired对FXNewsProvider类进行标注，以表明要为FXNewsProvider注入的依赖。代码清单6-1给出了标注后的情况。

代码清单6-1 使用@Autowired标注后的FXNewsProvider

```
public class FXNewsProvider
{
   private IFXNewsListener  newsListener;
   private IFXNewsPersister newPersistener;
```

```
    @Autowired
    public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister)
    {
      this.newsListener   = newsListner;
      this.newPersistener = newsPersister;
    }
    ...
}
```

与原有的byType类型的自动绑定方式类似，@Autowired也是按照类型匹配进行依赖注入的，只不过，它要比byType更加灵活，也更加强大。@Autowired可以标注于类定义的多个位置，包括如下几个。

域（Filed）或者说属性（Property）。不管它们声明的访问限制符是private、protected还是public，只要标注了@Autowired，它们所需要的依赖注入需求就都能够被满足，如下所示：

```
public class FXNewsProvider
{
  @Autowired
  private IFXNewsListener  newsListener;
  @Autowired
  private IFXNewsPersister newPersistener;
  ...
}
```

构造方法定义（Constructor）。标注于类的构造方法之上的@Autowired，相当于抢夺了原有自动绑定功能中“constructor”方式的权利，它将根据构造方法参数类型，来决定将什么样的依赖对象注入给当前对象。从最初的代码示例中，我们可以看到标注于构造方法之上的@Autowired的用法。

方法定义（Method）。@Autowired不仅可以标注于传统的setter方法之上，而且还可以标注于任意名称的方法定义之上，只要该方法定义了需要被注入的参数。代码清单6-2给出了一个标注于这种任意名称方法之上的@Autowired使用示例代码。

代码清单6-2　标注于方法之上的@Autowired代码示例

```
public class FXNewsProvider
{
  private IFXNewsListener  newsListener;
  private IFXNewsPersister newPersistener;

  @Autowired
  public void setUp(IFXNewsListener  newsListener,IFXNewsPersister newPersistener)
  {
    this.newsListener   = newsListener;
    this.newPersistener = newPersistener;
  }
  ...
}
```

现在，虽然可以随意地在类定义的各种合适的地方标注@Autowired，希望这些被@Autowired标注的依赖能够被注入，但是，仅将@Autowired标注于类定义中并不能让Spring的IoC容器聪明到自己去查看这些注解，然后注入符合条件的依赖对象。容器需要某种方式来了解，哪些对象标注了@Autowired，哪些对象可以作为可供选择的依赖对象来注入给需要的对象。在考虑使用什么方式实现这一功能之前，我们先比较一下原有的自动绑定功能与使用@Autowired之后产生了哪些差别。

使用自动绑定的时候，我们将所有对象相关的bean定义追加到了容器的配置文件中，然后使用default-autowire或者autowire告知容器，依照这两种属性指定的绑定方式，将容器中各个对象绑

定到一起。在使用@Autowired之后，default-autowire或者autowire的职责就转给了@Autowired，所以，现在，容器的配置文件中就只剩下了一个个孤伶伶的bean定义，如下所示：

```
<beans>
   <bean id="newsProvider"    class="..FXNewsProvider"/>
   <bean id="djNewsListener"  class="..DowJonesNewsListener"/>
   <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

为了给容器中定义的每个bean定义对应的实例注入依赖，可以遍历它们，然后通过反射，检查每个bean定义对应的类上各种可能位置上的@Autowired。如果存在的话，就可以从当前容器管理的对象中获取符合条件的对象，设置给@Autowired所标注的属性域、构造方法或者方法定义。整个逻辑如代码清单6-3中的原型代码所示。

代码清单6-3 容器遍历@Autowired并进行依赖注入的原型代码示例

```
Object[] beans = ...;
for(Object bean:beans)
{
   if(autowiredExistsOnField(bean))
   {
     Field f = getQulifiedField(bean));
     setAccessiableIfNecessary(f);
     f.set(getBeanByTypeFromContainer());
   }
   if(autowiredExistsOnConstructor(bean))
   {
     ...
   }
   if(autowiredExistsOnMethod(bean))
   {
     ...
   }
}
```

看到以上的原型代码所要完成的功能以及我们的设想，你一定想到了，我们可以提供一个Spring的IoC容器使用的BeanPostProcessor自定义实现，让这个BeanPostProcessor在实例化bean定义的过程中，来检查当前对象是否有@Autowired标注的依赖需要注入。org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor就是Spring提供的用于这一目的的BeanPostProcessor实现。所以，很幸运，我们不用自己去实现它了。

将FXNews相关类定义使用@Autowired标注之后，只要在IoC容器的配置文件中追加AutowiredAnnotationBeanPostProcessor就可以让整个应用开始运作了，如下所示：

```
<beans>
   <bean class="org.springframework.beans.factory.annotation. ➥
   AutowiredAnnotationBeanPostProcessor"/>

   <bean id="newsProvider"    class="..FXNewsProvider"/>
   <bean id="djNewsListener"  class="..DowJonesNewsListener"/>
   <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

当然，这需要我们使用ApplicationContext类型的容器，否则还得做点儿多余的准备工作。

注意　看着依赖注入相关的信息，一半分散在Java源代码中（@Autowired标注的信息），一半依然留在XML配置文件里，你心里一定觉得很不爽。实际上，我也是，这不是折腾人吗？不过，别急，让我们先解决眼前的另一个问题，稍后再回过头来看看怎么进一步统一这两片国土。

2. @Qualifier的陪伴

@Autowired是按照类型进行匹配，如果当前@Autowired标注的依赖在容器中只能找到一个实例与之对应的话，那还好。可是，要是能够同时找到两个或者多个同一类型的对象实例，又该怎么办呢？我们自己当然知道应该把具体哪个实例注入给当前对象，可是，IoC容器并不知道，所以，得通过某种方式告诉它。这时，就可以使用@Qualifier对依赖注入的条件做进一步限定，使得容器不再迷茫。

@Qualifier实际上是byName自动绑定的注解版，既然IoC容器无法自己从多个同一类型的实例中选取我们真正想要的那个，那么我们不妨就使用@Qualifier直接点名要哪个好了。假设FXNewsProvider使用的IFXNewsListener有两个实现，一个是DowJonesNewsListener，一个是ReutersNewsListener，二者相关配置如下：

```
<beans>
   <bean class="org.springframework.beans.factory.annotation. ➥
   AutowiredAnnotationBeanPostProcessor"/>

   <bean id="newsProvider"    class="..FXNewsProvider"/>
   <bean id="djNewsListener"  class="..DowJonesNewsListener"/>
   <bean id="reutersNewsListner" class="..ReutersNewsListener"/>
   <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

如果我们想让FXNewsProvider使用ReutersNewsListener，那么就可以在FXNewsProvider的类定义中使用@Qualifier指定这一选择结果，如下：

```
public class FXNewsProvider
{
   @Autowired
   @Qualifier("reutersNewsListner")
   private IFXNewsListener  newsListener;
   @Autowired
   private IFXNewsPersister newPersistener;
   ...
}
```

以上我们使用的是标注于属性域的@Autowired进行依赖注入。如果使用@Autowired来标注构造方法或者方法定义的话，同样可以使用@Qualifier标注方法参数来达到限定注入实例的目的。代码清单6-4给出的正是标注于方法参数之上的@Qualifier的使用示例。

代码清单6-4　标注于方法参数之上的@Qualifier

```
public class FXNewsProvider
{
   ...

   @Autowired
   public void setUp(@Qualifier("reutersNewsListner") IFXNewsListener ➥
   newsListener,IFXNewsPersister newPersistener)
   {
      this.newsListener   = newsListener;
      this.newPersistener = newPersistener;
```

```
    }
    ...
}
```

除此之外，@Qualifier还可以用于标注注解类型，这主要用于自定义@Qualifier的场合。有关自定义@Qualifier的内容我们这里就不做赘述了，Spring 2.5的参考文档中有详细的介绍。

6.1.2 @Autowired之外的选择——使用JSR250 标注依赖注入关系

Spring 2.5提供的基于注解的依赖注入，除了可以使用Spring提供的@Autowired和@Qualifier来标注相应类定义之外，还可以使用JSR250的@Resource和@PostConstruct以及@PreDestroy对相应类进行标注，这同样可以达到依赖注入的目的。

@Resource与@Autowired不同，它遵循的是byName自动绑定形式的行为准则，也就是说，IoC容器将根据@Resource所指定的名称，到容器中查找beanName与之对应的实例，然后将查找到的对象实例注入给@Resource所标注的对象。同样的FXNewsProvider，如若使用@Resource进行标注以获取依赖注入的话，类似如下的样子：

```
public class FXNewsProvider
{
   @Resource(name="djNewsListener")
   private IFXNewsListener  newsListener;
   @Resource(name="djNewsPersister")
   private IFXNewsPersister newPersistener;
   ...
}
```

JSR250规定，如果@Resource标注于属性域或者方法之上的话，相应的容器将负责把指定的资源注入给当前对象，所以，除了像我们这样直接在属性域上标注@Resource，还可以在构造方法或者普通方法定义上标注@Resource，这与@Autowired能够存在的地方大致相同。

确切地说，@PostConstruct和@PreDestroy不是服务于依赖注入的，它们主要用于标注对象生命周期管理相关方法，这与Spring的InitializingBean和DisposableBean接口，以及配置项中的init-method和destroy-method起到类似的作用。代码清单6-5给出了可能使用这两个注解的示例代码。

代码清单6-5 使用@PostConstruct和@PreDestroy标注对象的生命周期管理方法

```
public class LifecycleEnabledClass
{
   @PostConstruct
   public void setUp()
   {
      ...
   }
   @PreDestroy
   public void destroy()
   {
      ...
   }
}
```

如果想某个方法在对象实例化之后被调用，以做某些准备工作，或者想在对象销毁之前调用某个方法清理某些资源，那么就可以像我们这样，使用@PostConstruct和@PreDestroy来标注这些方法。当然，是使用@PostConstruct和@PreDestroy，还是使用Spring的InitializingBean和Disposable-

Bean接口，或者init-method和destroy-method配置项，可以根据个人的喜好自己决定。

天上永远不会掉馅饼，我们只是使用@Resource或者@PostConstruct和@PreDestroy标注了相应对象，并不能给该对象带来想要的东西。所以，就像@Autowired需要AutowiredAnnotationBeanPostProcessor为它与IoC容器牵线搭桥一样，JSR250的这些注解也同样需要一个BeanPostProcessor帮助它们实现自身的价值。这个BeanPostProcessor就是org.springframework.context.annotation.CommonAnnotationBeanPostProcessor，只有将CommonAnnotationBeanPostProcessor添加到容器，JSR250的相关注解才能发挥作用，通常如下添加相关配置即可：

```
<beans>
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>

<bean id="newsProvider"    class="..FXNewsProvider"/>
<bean id="djNewsListener"  class="..DowJonesNewsListener"/>
<bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

既然不管是@Autowired还是@Resource都需要添加相应的BeanPostProcessor到容器，那么我们就可以在基于XSD的配置文件中使用一个<context:annotation-config>配置搞定以上所有的BeanPostProcessor配置，如代码清单6-6所示。

代码清单6-6　使用<context:annotation-config/>激活注解的相关功能

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xmlns:p="http://www.springframework.org/schema/p" ➥
xsi:schemaLocation=" ➥
http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd"> ➥

   <context:annotation-config/>

   <bean id="newsProvider"    class="..FXNewsProvider"/>
   <!--其他bean定义-->
   ...
</beans>
```

<context:annotation-config>不但帮我们把AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor注册到容器，同时还会把PersistenceAnnotationBeanPostProcessor和RequiredAnnotationBeanPostProcessor一并进行注册，可谓一举四得啊！

注意　Spring提供的@Autowired加上@Qualifier和JSR250提供的@Resource等注解属于两个派系。如果要实现依赖注入的话，使用一个派别的注解就可以了。当然，既然<context:annotation-config>对两个派系都提供了BeanPostProcessor的支持，混合使用也是没有问题的，只要别造成使用上的混乱就行。

6.1.3　将革命进行得更彻底一些（classpath-scanning 功能介绍）

好了，该来解决让我们不爽的那个问题了。到目前为止，我们还是需要将相应对象的bean定义，

一个个地添加到IoC容器的配置文件中。与之前唯一的区别就是，不用在配置文件中明确指定依赖关系了（改用注解来表达了嘛）。既然使用注解来表达对象之间的依赖注入关系，那为什么不搞的彻底一点儿，将那些几乎“光秃秃”的bean定义从配置文件中彻底消灭呢？OK，我们想到了，Spring开发团队也想到了，classpath-scanning的功能正是因此而诞生的！

使用相应的注解对组成应用程序的相关类进行标注之后，classpath-scanning功能可以从某一项层包（base package）开始扫描。当扫描到某个类标注了相应的注解之后，就会提取该类的相关信息，构建对应的BeanDefinition，然后把构建完的BeanDefinition注册到容器。这之后所发生的事情就不用我说了，既然相关的类已经添加到了容器，那么后面BeanPostProcessor为@Autowired或者@Resource所提供的注入肯定是有东西拿咯！

classpath-scanning功能的触发是由<context:component-scan>决定的。按照如下代码，在XSD形式（也只能是XSD形式）的配置文件中添加该项配置之后，classpath-scanning功能立即开启：

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
<context:component-scan base-package="org.spring21"/>
</beans>
```

现在<context:component-scan>将遍历扫描org.spring21路径下的所有类型定义，寻找标注了相应注解的类，并添加到IoC容器。

提示 如果要扫描的类定义存在于不同的源码包下面，也可以为base-package指定多个以逗号分隔的扫描路径。需要的话，不要犹豫！

<context:component-scan>默认扫描的注解类型是@Component。不过，在@Component语义基础上细化后的@Repository、@Service和@Controller也同样可以获得<context:component-scan>的青睐。@Component的语义更广、更宽泛，而@Repository、@Service和@Controller的语义则更具体。所以，同样对于服务层的类定义来说，使用@Service标注它，要比使用@Component更为确切。对于其他两种注解也是同样道理，我们暂且使用语义更广的@Component来标注FXNews相关类，以便摆脱每次都要向IoC容器配置添加bean定义的苦恼。使用@Component标注后的FXNews相关类见代码清单6-7。

代码清单6-7 使用@Component标注后的FXNews相关类定义

```
@Component
public class FXNewsProvider
{
   @Autowired
   private IFXNewsListener  newsListener;
   @Autowired
   private IFXNewsPersister newPersistener;
   ...
}

@Component("djNewsListener")
public class DowJonesNewsListener implements IFXNewsListener
```

```
{
   ...
}

@Component
public class DowJonesNewsPersister implements IFXNewsPersister
{
   ...
}
```

<context:component-scan>在扫描相关类定义并将它们添加到容器的时候，会使用一种默认的命名规则，来生成那些添加到容器的bean定义的名称（beanName）。比如DowJonesNewsPersister通过默认命名规则将获得dowJonesNewsPersister作为bean定义名称。如果想改变这一默认行为，就可以像以上DowJonesNewsListener所对应的@Component那样，指定一个自定义的名称[①]。

现在，除了<context:component-scan>是唯一需要添加到IoC容器的配置内容，所有的工作都可以围绕着使用注解的Java源代码来完成了。如果现在加载配置文件，启动FXNewProvider来处理外汇新闻的话，我们可以得到预期的运行效果[②]，运行的代码如下所示：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("../conf.xml");
FXNewProvider provider = (FXNewProvider)ctx.getBean("FXNewsProvider");
provider.getAndPersistNews();
```

你或许会觉得有些诧异，因为我们并没有使用<context:annotation-config>甚至直接将相应的BeanPostProcessor添加到容器中，而FXNewsProvider怎么会获得相应的依赖注入呢？这个得怪<context:component-scan>“多管闲事”，它同时将AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor一并注册到了容器中，所以，依赖注入的需求得以满足。如果你不喜欢，非要自己通过<context:annotation-config>或者直接添加相关BeanPostProcessor的方式来满足@Autowired或者@Resource的需求，可以将<context:component-scan>的annotation-config属性值从默认的true改为false。不过，我想没有太好的理由非要这么做吧？

<context:component-scan>的扫描行为可以进一步定制，默认情况下它只关心@Component、@Repository、@Service和@Controller四位大员，但我们可以丰富这一范围，或者对默认的扫描结果进行过滤以排除某些类，<context:component-scan>的嵌套配置项可以帮我们达到这一目的。代码清单6-8演示了<context:component-scan>部分嵌套配置项的使用。

代码清单6-8　<context:component-scan>部分嵌套配置项的使用示例

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/ schema/context/spring-context-2.5.xsd">
<context:component-scan base-package="org.spring21">
```

① 另一种方法就是使用自定义实现的BeanNameGenerator，通过<context:component-scan>的name-generator属性指定我们自己的BeanNameGenerator实现类来替换掉默认的BeanNameGenerator，也可以改变默认的bean定义名称生成规则。

② 注意，我们通过FXNewsProvider作为bean定义名称来获取FXNewsProvider的实例，对于开头都是大写的类名来说，bean定义名称实际上就相当于类名，这与默认的命名生成规则有些小小的差异。

```
        <context:include-filter type="annotation" ➥
        expression="cn.spring21.annotation.FXService"/>
        <context:exclude-filter type="aspectj" expression=".."/>
    </context:component-scan>
</beans>
```

include-filter和exclude-filter可以使用的type类型有annotation、assignable、regex和aspectj四种。它们的更多信息可以参考最新的Spring 2.5参考文档。上例中，我们增加了@FXService作为新的被扫描注解对象，并使用aspectj表达式排除某些扫描结果。

注意 有关基于注解的依赖注入和classpath-scanning功能的更多细节不能尽述，可以对照Spring 2.5的参考文档做进一步的认识。

6.2 Spring 3.0展望

Java 5是Java平台发展史上的一个里程碑，它为Java平台带来了诸如泛型（Generics）[①]、注解等新的特性。随着时间推移，Java 5将逐渐流行并成为各种Java应用所需要的“最低”配置。Spring 3.0的发布将顺应这种形势，对现有的API进行全线升级。无疑，现有代码库（codebase）将接受一个大手术。届时，Spring框架将会“脱筋换骨”，以全新的面貌展现在我们的面前。

从发布的Spring 3.0 M2版本中，我们可以发现许多类都已经泛型化，像FactoryBean、ActionListener等类。

```
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<? extends T> getObjectType();
    boolean isSingleton();
}

public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {
    void onApplicationEvent(E event);

}
```

当然，Java 5风格的API升级并不限于泛型这一点，根据Spring团队博客上的介绍，他们还有可能在Spring 3.0中追加“基于注解的工厂方法（Annotation-based Factory method）”。虽然“基于注解的工厂方法”这一功能还未发布，不过我们可以先设想一下，如果要我们来实现，应该如何做呢？

首先，Spring 3.0将追加一个注解定义，用来标注相应的工厂方法。我们暂且假设该注解定义如下：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface FactoryMethod {

}
```

@FactoryMethod看起来有些“单薄”。不过没关系，它只是为了帮助我们构建一个代码原型，以便说明问题。在有了这一注解之后，我们就可以标注相应的工厂方法，比如：

```
public class AnyFactory {
```

① generics有的翻译成“模板”，有的翻译成“泛型”。如果硬要翻译的话，我倾向于“泛型”，因为“模板”已经有了其他的含义，很容易引起混淆以及交流障碍。

```
    @FactoryMethod
    public <T> T create(Class<T> clazz)
    {
      // 创建并返回相应的实例
      try {
        return clazz.newInstance();
      } catch (InstantiationException e) {
        //...
        return null;
      } catch (IllegalAccessException e) {
        //...
        return null;
      }
    }
}
```

但是，只使用@FactoryMethod标注相应方法，Spring并不会知道这个被标注的方法就是一个工厂方法。我们还需要通过某种方式通知Spring说，“标注了@FactoryMethod的方法定义都是需要特殊对待的工厂方法”，或者“如果哪个类中有方法定义被标注了@FactoryMethod，那么应该对这个类做一些特殊处理”。在有了classpath-scanning这一“基础设施”之后，要实现这样的需求只需要在classpath-scanning期间添加相应的处理逻辑即可。要完成这一工作可能有如下几种选择。

- classpath-scanning的时候，在对标注了@Component或者其他注解的类进行处理的同时，也稍带检查一下当前类是否也标注了@FactoryMethod。如果是，则进一步将工厂方法相关信息添加到相应的BeanDefinition。
- 向容器中新添加一个BeanFactoryPostProcessor实现类，由这个BeanFactoryPostProcessor来统一做@FactoryMethod的相关处理。不过，这需要org.springframework.beans.factory.config.BeanFactoryPostProcessor接口开放更多一些的接口。在Spring 3.0 M2的Javadoc中可以发现这样的迹象，所以这条路应该也可以走通。

还可以尝试其他思路。或者等Spring 3.0发布之后自己验证一下是不是这么回事……

6.3　小结

Spring最初并不支持基于注解的依赖注入方式。所以，在Spring 2.5中引入这一依赖注入方式的时候，肯定要在维护整个框架设计与实现的一致性和引入这种依赖注入方式对整个框架的冲击之间做出权衡。最终的结果我们已经看到了，Spring 2.5中引入的基于注解的依赖注入从整体上保持了框架内的一致性，同时又提供了足够的基于注解的依赖注入表达能力。我想，最初的决定和最终的效果都是令人满意的。虽然我们还会部分地依赖于容器的配置文件，但通过20%的工作却可以带来80%的效果，这本身已经是最好的结果了。

不过，从实际开发角度看，如果非要使用完全基于注解的依赖注入的话，或许会遇到一些过不去的坎儿。比如，对于第三方提供的类库，肯定没法给其中的相关类标注@Component之类的注解。这时，我们可以结合使用基于配置文件的依赖注入方式。毕竟，基于XML的依赖注入方式是Spring提供的最基本、也最为强大的表达方式了！

到目前为止，我们已经结束了Spring IoC容器的旅程。接下来的第三部分将带领读者探索Spring AOP框架的精彩世界。

时刻关注企业软件开发领域的变化与创新

架构师

www.infoq.com/cn/architect

每月8号出版

InfoQ中文站
www.infoq.com/cn

商务合作：sales@cn.infoq.com
读者反馈/投稿：editors@cn.infoq.com