Spring学习指南

（第3版）

[印度] J. 夏尔马（J. Sharma） 阿西施·萨林 (Ashish Sarin) 著
周密 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Spring学习指南

（第3版）

[印度] J. 夏尔马（J. Sharma） 阿西施·萨林 (Ashish Sarin) 著
周密 译

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Spring学习指南 : 第3版 / (印) J.夏尔马 (J. Sharma) , (印) 阿西施·萨林 (Ashish Sarin) 著 ; 周密译. -- 北京 : 人民邮电出版社, 2018.7
ISBN 978-7-115-48237-2

Ⅰ. ①S… Ⅱ. ①J… ②阿… ③周… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第067049号

版权声明

◆ 著 [印度] J. 夏尔马（J. Sharma ）
[印度]阿西施·萨林 (Ashish Sarin)
译 周 密
责任编辑 吴晋瑜
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
固安县铭成印刷有限公司印刷

◆ 开本：787×1092 1/16
印张：25.75
字数：834 千字 2018 年 7 月第 1 版
印数：1－2 400 册 2018 年 7 月河北第 1 次印刷

著作权合同登记号 图字：01-2016-8079 号

定价：89.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

内 容 提 要

Spring 框架是以简化 J2EE 应用程序开发为特定目标而创建的，是当前最流行的 Java 开发框架。

本书从介绍 Spring 框架入手，针对 Spring 4.3 和 Java 8 介绍 bean 的配置、依赖注入、定义 bean、基于 Java 的容器、AOP、Spring Data、Spring MVC 等知识，旨在帮助读者更轻松地学习 Spring 框架的方法。

本书适合 Web 开发者和想使用 Spring 的初学者参考，也可供对 Web 开发和 Spring 感兴趣的读者参考。

译 者 序

Spring 框架可以说是当前 Java 开发的事实标准，但是大多数高校教材中并没有涵盖相关内容，这使得很多 Java 开发人员只能在工作中靠口口相传或者自学来了解 Spring 框架，虽然最终可以掌握，但是由于缺乏系统性的指导，难免在花费大量时间之余走很多的弯路。

本书是 Spring 框架的入门指南，兼具系统性和实用性，全面介绍了 Spring 框架的设计思想和模块构成，并针对每个模块都给出了应用场景以及相应的源代码示例，以引导开发者掌握 Spring 框架的使用。

本书适合有一定 Java 基础的学生或者初级开发人员学习，也可供对 Spring 框架掌握不够系统或不了解新版本 Spring 框架功能的资深开发人员参考。

在翻译过程中，我尽量遵循原文意思，力求使译文贴合中文阅读习惯，但碍于自身水平有限，难免会有不尽如人意的地方，还请广大读者不吝指正，谢谢！

最后，感谢人民邮电出版社的各位编辑老师对我的鼓励，感谢推荐我承接翻译工作的高博老师，也感谢在本书翻译过程中给予我理解与支持的家人，我爱你们！

周 密

2018 年 5 月

前　言

如何使用这本书

下载示例项目

本书有许多示例项目，你可以从 GitHub 项目中自行寻找。你可以将示例项目作为单个 ZIP 文件下载，也可以使用 Git 检索出示例项目。

将示例项目导入你的 Eclipse 或者 IntelliJ IDEA IDE

如果你在阅读本书时发现有 IMPORT chapter<chapter-number>/<project name> 这样的标识，那么应该将指定的项目导入你的 Eclipse 或者 IntelliJ IDEA IDE（或者任何其他正在使用的 IDE）。这些示例项目使用 Maven 3.x 版本作为项目的构建工具，因此，你在每个项目中都可以找到一个 pom.xml 文件。在整套源码的根目录中也有一个 pom.xml 文件，该文件是用来一次性构建全部项目的。

通过参考附录 B 可以了解导入和运行示例项目所需的步骤。

参考代码示例

在每个程序示例中都会指明示例项目的名称（使用 Project 标签）和源文件位置（使用 Source location 标签）。如果没有在程序示例中指明 Project 标签和 Source location 标签，你可以认为程序示例中的代码**并不是**从示例项目中摘录出来的，纯粹只是为了帮助你理解。

本书体例

粗体用于强调术语。

`Comic Sans MS` 用于程序示例、Java 代码、XML 中的配置细节和属性文件。

`Comic Sans MS` 用于在程序示例中突出重要的代码或者配置。

注　意

这样的一个标注突出了一个重要的观点或概念。

反馈和问题

你可以在 Google Groups 论坛中向作者发送反馈和问题。

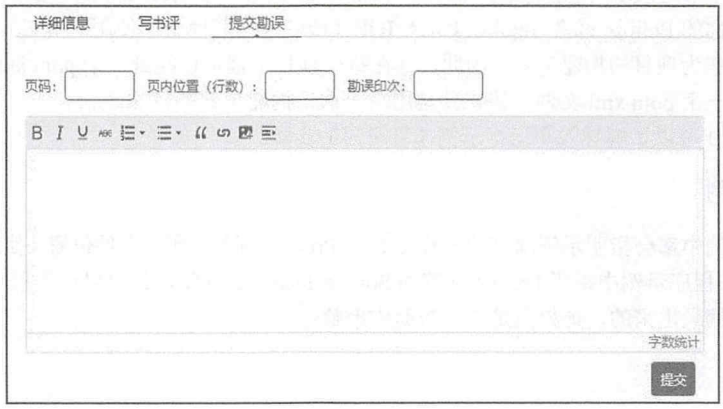
资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，单击“提交勘误”，输入勘误信息，单击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，将赠予您异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。



扫码关注本书

扫描下方二维码，您将会在异步社区微信服务号中看到本书信息及相关的服务提示。

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的 LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区

微信服务号

目　录

第 1 章　Spring 框架简介

1.1　简介

在传统的 Java 企业级应用开发中，创建结构良好、易于维护和易于测试的应用程序是开发者的职责。开发者用各式各样的设计模式来解决这些应用的非业务需求。这不但导致开发者生产效率低下，而且对开发应用的质量造成了不良影响。

Spring 框架（简称 Spring）是 SpringSource 出品的一个用于简化 Java 企业级应用开发的开源的应用程序框架。它提供了开发一个结构良好的、可维护和易于测试的应用所需的基础设施，当使用 Spring 框架时，开发者只需要专注于编写应用的业务逻辑，从而提高了开发者的生产效率。你可以使用 Spring 框架开发独立的 Java 应用程序、Web 应用程序、Applet，或任何其他类型的 Java 应用程序。

本章首先介绍 Spring 框架的模块和它们的优点。Spring 框架的核心是提供了依赖注入（Dependency Injection，DI）机制的控制翻转（Inversion of Control，IoC）容器。本章将介绍 Spring 的 DI 机制以及 IoC 容器，并展示如何使用 Spring 开发一个独立的 Java 应用。在本章的结尾，我们来看一些以 Spring 框架为基础的 SpringSource 项目。有了本章的铺垫，我们可以在后面的章节更深入地探究 Spring 框架。

注　意

在本书中，我们将以一个名为 **MyBank** *的网上银行应用为例，介绍 Spring 框架的功能。*

1.2　Spring 框架的模块

Spring 框架由多个模块组成，它们根据应用开发功能进行分组。表 1-1 列出了 Spring 框架中的各个模块组，并描述了其中一些重要模块组所提供的功能。

表 1-1　　Spring 框架中的各个模块组

模块组	描述
Core container	包含构成 Spring 框架基础的模块。该组中的 spring-core 和 spring-beans 模块提供了 Spring 的 DI 功能和 IoC 容器实现。spring-expressions 模块为在 Spring 应用中通过 **Spring 表达式语言**（见第 6 章）配置应用程序对象提供了支持
AOP and instrumentation	包含支持 AOP（面向切面编程）和类工具模块。The spring-aop 模块提供 Spring 的 AOP 功能，spring-instrument 模块提供了对类工具的支持
Messaging	包含简化开发基于消息的应用的 spring-messaging 模块
Data Access/Integration	包含简化与数据库和消息提供者交互的模块。spring-jdbc 模块简化了用 JDBC 与数据库的交互，spring-orm 模块提供了与 ORM（对象关系映射）框架的集成，如 JPA 和 Hibernate。spring-jms 模块简化了与 JMS 提供者的交互。 此模块组还包含 spring-tx 模块，该模块提供了编程式与声明式事务管理
Web	包含简化开发 Web 和 portlet 应用的模块。spring-web 和 spring-webmvc 模块都是用于开发 Web 应用和 RESTful 的 Web 服务的。spring-websocket 模块支持使用 WebSocket 开发 Web 应用
Test	包含 spring-test 模块，该模块简化了创建单元和集成测试

由表 1-1 可知，Spring 涵盖了企业应用程序开发的各个方面，可以使用 Spring 开发 Web 应用程序、访问数据库、管理事务、创建单元和集成测试等。在设计 Spring 框架模块时，你只需要引入应用程序所需要的模块。例如，在应用程序中使用 Spring 的 DI 功能，只需要引入 **Core container** 组中的模块。看完本书之后，你会发现更多关于 Spring 模块和示例的细节，来展示如何把它们应用在开发工作中。

在 Spring 框架中，JAR 文件的命名惯例如下：

```
spring-<short-module-name>-<spring-version>.jar.
```

其中，**<short-module-name>** Spring 模块的简称，如 aop、beans、context、expressions 等。而 **<spring-version>** 是 Spring 框架的版本。

根据这个命名惯例，Spring 4.3.0.RELEASE 版本中 JAR 文件的名字为 spring-aop-4.3.0.RELEASE.jar、spring-beans-4.3.0.RELEASE.jar 等。

图 1-1 显示了 Spring 模块之间的依赖关系。

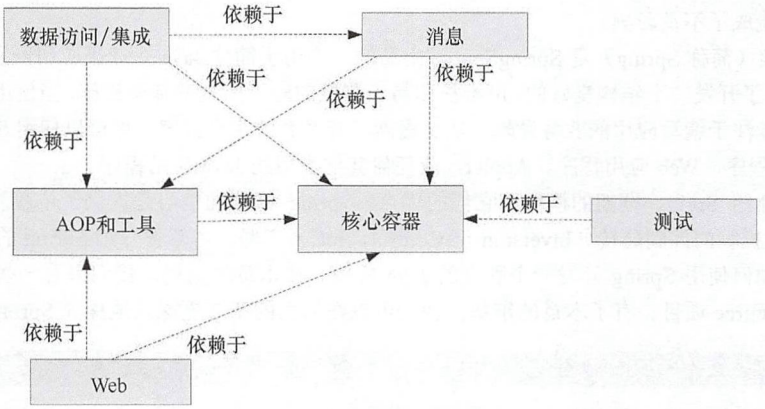


图 1-1 Spring 模块之间的依赖关系

从图 1-1 可知，**Core container** 组所包含的模块是 Spring 框架的中心，其他模块都依赖于它。同等重要的是 AOP and instrumentation 组所包含的模块，因为它们提供了 Spring 框架中其他模块的 AOP 功能。

现在你对 Spring 所涵盖的应用程序开发有了一些基本的概念，让我们来看看 Spring 的 IoC 容器。

1.3 Spring IoC 容器

一个 Java 应用程序由互相调用以提供应用程序行为的一组对象组成。某个对象调用的其他对象称为它的**依赖项**。例如，如果对象 X 调用了对象 Y 和 Z，那么 Y 和 Z 就是对象 X 的依赖项。DI 是一种设计模式，其中对象的依赖项通常被指定为其构造函数和 setter 方法的参数。并且，这些依赖项将在这些对象创建时注入该对象中。

在 Spring 应用程序中，Spring IoC 容器（也称为 Spring 容器）负责创建应用程序对象并注入它们的依赖项。Spring 容器创建和管理的应用对象称为 **bean**。由于 Spring 容器负责将应用程序对象组合在一起，因此不需要实现诸如工厂或者服务定位器等设计模式来构成应用。因为创建和注入依赖项的不是应用程序的对象，而是 Spring 容器，所以 DI 也称为控制反转（IoC）。

假设 Mybank 应用程序（这是示例应用程序的名称）包含 FixedDepositController 和 FixedDepositService 两个对象，FixedDepositController 对象依赖于 FixedDepositService 对象，如程序示例 1-1 所示。

程序示例 1-1 FixedDepositController 类

```
public class FixedDepositController {
```

```
  private FixedDepositService fixedDepositService;

  public FixedDepositController() {
    fixedDepositService = new FixedDepositService();
  }

  public boolean submit() {
    //-- 保存定期存款明细
    fixedDepositService.save(.....);
  }
}
```

在程序示例 1-1 中， FixedDepositController 的构造函数创建了一个 FixedDepositService 的实例后用于 FixedDepositController 的 submit 方法。因为 FixedDepositController 调用了 FixedDepositService，所以 FixedDepositService 就是 FixedDepositController 的一个依赖项。

若要将 FixedDepositController 配置为一个 Spring bean，首先需要修改在程序示例 1-1 中的 FixedDepositController 类，让它接收 FixedDepositService 依赖作为构造函数参数或者 setter 方法的参数。修改后的 FixedDepositController 类，如程序示例 1-2 所示。

程序示例 1-2 FixedDepositController 类——FixedDepositService 作为构造函数参数传递

```
public class FixedDepositController {
  private FixedDepositService fixedDepositService;

  public FixedDepositController(FixedDepositService fixedDepositService) {
    this.fixedDepositService = fixedDepositService;
  }

  public boolean submit() {
    //--保存定期存款明细
    fixedDepositService.save(.....);
  }
}
```

程序示例 1-2 表明 FixedDepositService 示例现在已经作为构造函数参数传递到 FixedDepositController 实例中。现在的 FixedDepositService 类可以配置为一个 Spring bean。注意，FixedDepositController 类并没有实现或者继承任何 Spring 的接口或者类。

在基于 Spring 的应用程序中，有关应用程序对象及其依赖项的信息都是由**配置元数据**来指定的。Spring IoC 容器读取应用程序的配置元数据来实例化应用程序对象并注入它们的依赖项。程序示例 1-3 展示了一个包含 MyController 和 MyService 两个类的应用的配置元数据（XML 格式）。

程序示例 1-3 配置元数据

```
<beans .....>
  <bean id="myController" class="sample.spring.controller.MyController">
    <constructor-arg ref="myService" />
  </bean>

  <bean id="myService" class="sample.spring.service.MyService"/>
</beans>
```

在程序示例 1-3 中，每个<bean>元素定义了一个由 Spring 容器管理的应用对象，而<constructor-arg>元素指定 MyService 实例作为一个构造函数的参数传递给 MyController。在本章后面的部分将会介绍<bean>元素，而<constructor-arg>元素的介绍会放在第 2 章。

Spring 容器读取应用程序的配置元数据（见程序示例 1-3）后，创建由<bean>元素定义的应用程序并注入它们的依赖项。Spring 容器使用 Java 反射 API 创建应用程序对象并注入其依赖项。图 1-2 总结了 Spring 容器的工作原理。

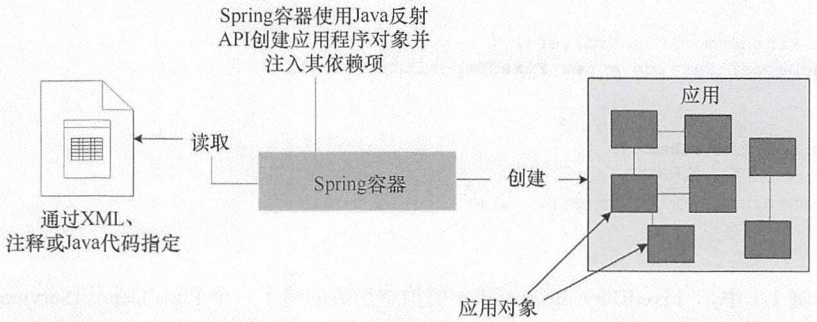


图 1-2 Spring 容器读取应用程序的配置元数据并创建一个配置完整的应用程序

Spring 容器的配置元数据可以通过 XML（见程序示例 1-3）、Java 注释（见第 6 章）以及 Java 代码（见第 7 章）来指定。

由于 Spring 容器负责创建和管理应用程序对象，企业服务（如事务管理、安全性、远程访问等）可以通过 Spring 容器透明地应用到对象上。Spring 容器的这种增强应用程序对象附加功能的能力让我们可以使用简单的 Java 对象（也称为 Plain Old Java Objects，POJO）作为应用对象。对应于 POJO 的 Java 类称作 **POJO 类**，也就是不实现或继承框架特定的接口或类的 Java 类。需要这些 POJO 的企业服务，如事务管理、安全、远程访问等，由 Spring 容器透明地提供。

现在你知道 Spring 容器是如何工作的了，下面再通过几个例子来看看使用 Spring 开发应用的好处。

1.4 使用 Spring 框架的好处

在前面的章节中，我们介绍了 Spring 带来的以下好处。

1）Spring 负责应用程序对象的创建并注入它们的依赖项，简化了 Java 应用程序的组成。

2）Spring 推动了以 POJO 的形式来开发应用程序。

Spring 提供了一个负责样板代码的抽象层，以此简化与以下模块的交互，如 JMS 提供者、JNDI、MBean 服务器、邮件服务器和数据库等。

让我们快速地通过几个例子来更好地理解使用 Spring 开发应用程序有哪些好处。

1. 管理本地和全局事务的一致方法

如果你正在使用 Spring 开发一个需要**事务**的应用程序，那么可以使用 Spring 的**声明式事务管理**来管理事务。

MyBank 应用程序中的 FixedDepositService 类，如程序示例 1-4 所示。

程序示例 1-4 FixedDepositService 类

```
public class FixedDepositService {
  public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }
  public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }
}
```

FixedDepositService 类是用来定义定期存款业务中创建和取回明细方法的 POJO 类，图 1-3 展示了创建一笔新的定期存款的表单。

一位客户在上面的表单中输入了定期存款金额、存期和电子邮箱 ID 信息，并单击 SAVE 按钮来创建一笔新的定期存款，此时会调用在 FixedDepositService 中的 createFixedDeposit 方法（见程序示例 1-4）来创建存款，createFixedDeposit 方法从该客户银行账户中扣除他输入的金额并创建一笔等额的定期

存款。

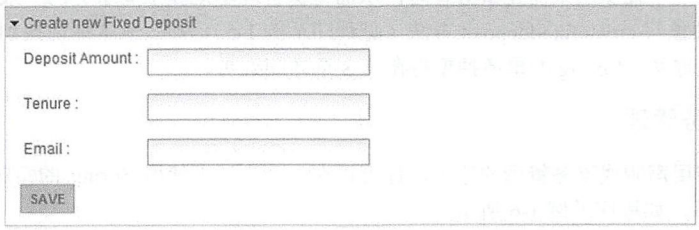


图 1-3 创建定期存款的 HTML 表单

假定关于客户银行余额的信息存在数据表 BANK_ACCOUNT_DETAILS 中，定期存款的明细存在数据表 FIXED_DEPOSIT_DETAILS 中。如果客户创建了一笔金额为 *x* 的定期存款，应该在 BANK_ACCOUNT_DETAILS 表中减去 *x* 并在 FIXED_DEPOSIT_DETAILS 表中插入一条记录来反映这笔新加的定期存款。如果 BANK_ACCOUNT_DETAILS 表没有更新或者新的记录没有插入 FIXED_DEPOSIT_DETAILS 表，这会让系统处于不一致状态。这意味着 createFixedDeposit 方法必须在一个事务中执行。

由 Mybank 应用程序所使用的数据库是一个事务性资源。以传统的方式在一个工作单元中执行一组数据库的修改操作时，先要禁用 JDBC 连接的自动提交模式，然后执行 SQL 语句，最后提交（或回滚）事务。用传统的方式在 createFixedDeposit 方法中管理数据库事务的方法如程序示例 1-5 所示。

程序示例 1-5 以编程方式使用 JDBC 连接对象管理数据库事务

```
import java.sql.Connection;
import java.sql.SQLException;

public class FixedDepositService {
   public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

   public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
          Connection con = ..... ;
          try {
              con.setAutoCommit(false);
              //-- 执行修改数据库表的 SQL 语句
              con.commit();
          } catch(SQLException sqle) {
              if(con != null) {
                  con.rollback();
              }
          }
          .....
      }
}
```

程序示例 1-5 展示了如何在 createFixedDeposit 方法中以编程方式使用 JDBC 连接对象管理数据库事务。这种方式适合只涉及单个数据库的应用场景。具体资源相关的事务，如与 JDBC 连接相关的事务，称为**本地事务**。

当多个事务性资源都有涉及，使用 JTA（Java 事务 API）来管理事务时，例如要在同一个事务中将 JMS 消息发送到消息中间件（一种事务资源）并更新数据库（另一种事务资源），则必须使用一个 JTA 事务管理器管理事务。JTA 事务也称为全局（或分布式）事务。要使用 JTA，需要先从 JNDI 中获取 UserTransaction 对象（这是 JTA API 的一部分），并编程开始和提交（或回滚）事务。

如你所见，可以使用 JDBC 连接（本地事务）或 userTransaction（对于全局事务）对象以编程方式管理事务。但是请注意，本地事务**无法**在全局事务中运行。这意味着如果要在 createFixedDeposit 数据库更新方法（见程序示例 1-5）使之成为 JTA 事务的一部分，则需要修改 createFixedDeposit 方法，用 UserTransaction

对象进行事务管理。

Spring 通过提供一个抽象层来简化事务管理，从而提供管理本地和全局事务的一致方法。这意味着如果用 Spring 的事务抽象写 createfixeddeposit 方法（见程序示例 1-5），那么从本地切换到全局事务管理时不需要修改方法，反之亦然。Spring 的事务抽象将在第 8 章详细说明。

2. 声明式事务管理

Spring 提供了使用**声明式事务管理**的选项，你可以在一个方法上使用 Spring 的@Transactional 注解并让 Spring 来处理事务，如程序示例 1-6 所示。

程序示例 1-6 使用@Transactional 注解

```
import org.springframework.transaction.annotation.Transactional;

public class FixedDepositService {
  public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

  @Transactional
  public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }
}
```

程序示例 1-6 表明，FixedDepositService 类没有实现任何接口或继承任何 Spring 特定的类以得到 Spring 的事务管理能力。Spring 框架透明地通过@Transactional 注解为 createFixedDeposit 方法提供事务管理功能。这说明 Spring 是一个非侵入性的框架，因为它不需要应用对象依赖于 Spring 特定的类或接口。由于事务管理是由 Spring 接管的，因此不需要直接使用事务管理 API 来管理事务。

3. 安全

对于任何 Java 应用程序来说，安全都是一个重要的方面。Spring Security 是一个 SpringSourc 置于 Spring 框架顶层的项目，它提供了身份验证和授权功能，可以用来保护 Java 应用程序。下面以 3 个在 Mybank 应用程序中认证过的用户角色为例进行说明，即 LOAN_CUSTOMER、SAVINGS_ACCOUNT_CUSTOMER 和 APPLICATION_ADMIN。调用 FixedDepositService 类（见示例 1-6）中 createFixedDeposit 方法的客户必须是相关的 SAVINGS_ ACCOUNT_CUSTOMER 或者拥有 APPLICATION_ADMIN 角色。而使用 Spring Security 时，你可以通过在 createFixedDeposit 方法上添加 Spring Security 的 @Secured 注解来轻松地解决这个问题， 如程序示例 1-7 所示。

程序示例 1-7 使用@Secured 注解的 createFixedDeposit 方法

```
import org.springframework.transaction.annotation.Transactional;
import org.springframework.security.access.annotation.Secured;

public class FixedDepositService {
        public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

        @Transactional
        @Secured({ "SAVINGS_ACCOUNT_CUSTOMER", "APPLICATION_ADMIN" })
        public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }
}
```

如果用@Secured 给一个方法加注解，安全特性将被 Spring Security 框架透明地应用到该方法上。程序示例 1-7 表明，为了实现方法级别的安全，你无须继承或实现任何 Spring 特定类或接口，而且不需要在业务方法中写安全相关的代码。

我们将在第 16 章详细讨论 Spring Security 框架。

4. JMX（Java 管理扩展）

Spring 对 JMX 的支持可以让你非常简单地将 JMX 技术融合到应用程序中。

假设 Mybank 应用程序的定期存款功能应该只在每天早上 9 点到下午 6 点的时间段提供给客户。为了满足这个要求，需要在 FixedDepositService 中增加一个变量，以此作为一个标志表明定期存款服务是否活跃。程序示例 1-8 显示了使用活跃变量的 FixedDepositService 类。

程序示例 1-8 使用活跃变量的 FixedDepositService 类

```
public class FixedDepositService {
   private boolean active;

  public FixedDepositDetails getFixedDepositDetails( ..... ) {
        if(active) { ..... }
  }
  public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
        if(active) { ..... }
  }
  public void activateService() {
        active = true;
  }
  public void deactivateService() {
        active = false;
  }
}
```

程序示例 1-8 表明，FixedDepositService 类中加了一个名为 active 的变量。如果 active 变量的值为 true，getFixedDepositDetails 和 createFixedDeposit 方法将按照预期工作。如果 active 变量的值为 false，getFixedDepositDetails 和 createFixedDeposit 方法将抛出一个异常，表明定期存款服务当前不可用。activateService 和 deactivateService 方法分别将 active 变量的值置为 true 和 false。

那么，谁调用 activateService 和 deactivateService 方法呢？假设有一个名为 Bank App Scheduler 的调度应用程序，分别在上午 9:00 和下午 6:00 执行 activateservice 和 deactivateservice 方法。Bank App Scheduler 应用使用 JMX （Java 管理扩展） API 与 FixedDepositService 实例远程交互。

Bank App Scheduler 使用 JMX 改变 FixedDepositService 中 active 变量的值，你需要将 FixedDepositService 实例在一个可被管理的 bean（或者称为 MBean）服务器上注册为一个 MBean，并将 FixedDepositService 中的 activateService 和 deactivateService 方法暴露为 JMX 操作方法。在 Spring 中，你可以通过在一个类上添加 Spring 的@ManagedResource 注释来将一个类的实例注册到 MBean 服务器上，并且可以使用 Spring 的@ManagedOperation 注释将该类的方法暴露为 JMX 操作方法。

在程序示例 1-9 中展示了使用@ManagedResource 和@ManagedOperation 注释将 FixedDepositService 类的实例注册到 MBean 服务器，并将 activateService 和 deactivateService 方法暴露为 JMX 操作方法。

程序示例 1-9 使用 Spring JMX 支持的 FixedDepositService 类

```
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(objectName = "fixed_deposit_service:name=FixedDepositService")
public class FixedDepositService {
  private boolean active;

  public FixedDepositDetails getFixedDepositDetails( ..... ) {
    if(active) { ..... }
  }
  public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
    if(active) { ..... }
```

```
    }

    @ManagedOperation
    public void activateService() {
      active = true;
    }

    @ManagedOperation
    public void deactivateService() {
      active = false;
    }
}
```

程序示例 1-9 表明 FixedDepositService 类将它的实例注册到 MBean 服务器并暴露它的方法为 JMX 操作方法时并没有直接使用 JMX API。

5. JMS（Java 消息服务）

Spring 的 JMS 支持简化了从 JMS 提供者发送和接收消息。

在 MyBank 应用程序中，当客户通过电子邮件提交一个接收其定期存款明细的请求时，FixedDepositService 将请求的明细发送到 JMS 消息中间件（比如 ActiveMQ），而请求随后由消息侦听器处理。Spring 通过提供一个抽象层来简化与 JMS 提供者的交互。程序示例 1-10 展示了 FixedDepositService 类如何通过 Spring 的 JmsTemplate 将请求的明细发送到 JMS 提供者。

程序示例 1-10 发送 JMS 消息的 FixedDepositService 类

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;

public class FixedDepositService {
  @Autowired
  private transient JmsTemplate jmsTemplate;
  .....
  public boolean submitRequest(Request request) {
      jmsTemplate.convertAndSend(request);
  }
}
```

在程序示例 1-10 中，FixedDepositService 定义了一个 JmsTemplate 类型的变量，这个变量使用了 Spring 的@Autowired 注释。现在，你可以认为@Autowired 注释提供了一个 JmsTemplate 实例。这个 JmsTemplate 实例知道 JMS 消息发送的目的地。如何配置这个 JmsTemplate 的细节会在第 10 章介绍。FixedDepositService 类的 submitRequest 方法调用了 JmsTemplate 的 convertAndSend 方法，把请求的明细（由 submitRequest 方法的 Request 参数表示）作为一个 JMS 消息发送到 JMS 提供者。

这也再一次表明，如果使用 Spring 框架向 JMS 提供者发送消息，并不需要直接处理 JMS API。

6. 缓存

Spring 的缓存抽象提供了在应用程序中使用缓存的一致方法。

使用缓存解决方案来提高应用程序的性能是很常见的。MyBank 应用使用一个缓存产品以提高读取定期存款明细操作的性能。Spring 框架通过抽象缓存相关的逻辑来简化与不同缓存解决方案的交互。

程序示例 1-11 展示了 FixedDepositService 类的 getFixedDepositDetails 方法使用 Spring 的缓存抽象功能来缓存定期存款明细。

程序示例 1-11 将定期存款明细缓存的 FixedDepositService 类

```
import org.springframework.cache.annotation.Cacheable;
```

```
public class FixedDepositService {

  @Cacheable("fixedDeposits")
  public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

  public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }
}
```

在程序示例 1-11 中，Spring 的@Cacheable 注解表明由 getFixedDepositDetails 方法返回的定期存款明细将被缓存起来，如果使用同样的参数来调用 getFixedDepositDetails 方法，getFixedDepositDetails 方法并不会实际运行，而是直接返回缓存中的定期存款明细。这表明，如果使用 Spring 框架，则不需要在类中编写与缓存相关的逻辑。Spring 的缓存抽象在第 10 章中详细介绍。

在这一部分中，我们看到 Spring 框架通过透明地向 POJO 提供服务的方式简化了企业应用程序的开发，从而将开发者从底层 API 的细节中解放出来。Spring 还提供了与各种标准框架，如 Hibernate、Quartz、JSF、Struts 和 EJB 等的简单集成，使得 Spring 成为企业应用程序开发的理想选择。

现在，我们已经看到了一些使用 Spring 框架的好处，下面来看如何开发一个简单的 Spring 应用程序。

1.5 一个简单的 Spring 应用程序

在这一部分，我们来关注一个使用 Spring 的 DI 功能的简单的 Spring 应用程序。在一个应用程序中使用 Spring 的 DI 功能，需要遵循以下步骤：

1）确定应用程序对象及其依赖关系；

2）根据步骤 1 中确定的应用程序对象创建 POJO 类；

3）创建描述应用程序对象及其依赖项的配置元数据；

4）创建一个 Spring IoC 容器的实例并将配置元数据传递给它；

5）从 Spring IoC 容器实例中访问应用程序对象。

现在让我们来看看上述步骤在 Mybank 应用程序中是如何体现的。

1. 确定应用程序对象及其依赖关系

前面讨论过，Mybank 应用程序展示了创建一笔定期存款的表单（见图 1-3）。图 1-4 的时序图显示了当用户提交表单时出现的应用程序对象（以及它们之间的交互）：

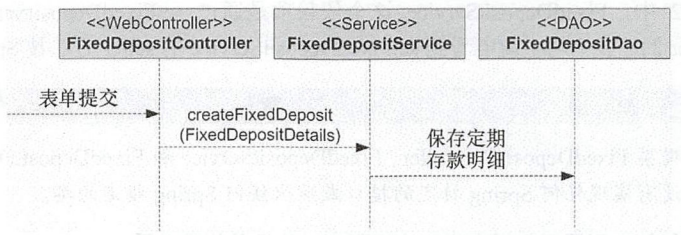


图 1-4 MyBank 的应用程序对象及其依赖项

在图 1-4 所示的时序图中，FixedDepositController 代表当这个表单提交时接受请求的 WebController，而 FixedDepositDetails 对象包含定期存款明细，FixedDepositController 调用 FixedDepositService （服务层对象）的 createFixedDeposit 方法。然后，FixedDepositService 调用 FixedDepositDao 对象（数据访问对象）来把定期存款明细保存到应用程序的数据存储区。因此，我们可以从图中理解 FixedDepositService 是 FixedDepositController 对象的依赖项，而 FixedDepositDao 是 FixedDepositService 对象的依赖项。

含 义

chapter 1/ch01-bankapp-xml （这个项目是一个使用 Spring DI 功能的简单的 Spring 应用程序。执行项目中 BankApp 类中的 main 方法即可运行应用程序）。

2. 根据确定的应用程序对象创建 POJO 类

一旦已经确定了应用程序对象，下一步就是根据这些应用程序对象创建 POJO 类。ch01-bankapp-xml 项目中已经包含了对应于 FixedDepositController、FixedDepositService 和 FixedDepositDao 这些应用程序对象的 POJO 类。ch01-bankapp-xml 项目是一个使用 Spring DI 功能的简化版 Mybank 应用程序。你可以将 ch01-bankapp-xml 项目导入 IDE 中，接下来我们来看这个项目中包含的文件。

在 1.3 节中，我们讨论了把依赖项作为构造函数参数或作为 setter 方法参数传递给应用程序对象。程序示例 1-12 展示了一个 FixedDepositService 的实例（FixedDepositController 的依赖项）是如何作为一个 setter 方法的参数传递给 FixedDepositController 类的。

程序示例 1-12 FixedDepositController 类

```
Project - ch01-bankapp-xml
Source location - src/main/java/sample/spring/chapter01/bankapp

package sample.spring.chapter01.bankapp;
.....
public class FixedDepositController {
  .....
   private FixedDepositService fixedDepositService;
  .....
   public void setFixedDepositService(FixedDepositService fixedDepositService) {
    logger.info("Setting fixedDepositService property");
    this.fixedDepositService = fixedDepositService;
  }
  .....
  public void submit() {
    fixedDepositService.createFixedDeposit(new FixedDepositDetails( 1, 10000,
      365, "someemail@something.com"));
  }
  .....
}
```

在程序示例 1-12 中，FixedDepositService 这个依赖项是通过 setFixedDepositService 方法被传递给 FixedDepositController 的。我们马上就能看到 setFixedDepositService 的 setter 方法被 Spring 调用。

注 意

如果观察 FixedDepositController、FixedDepositService 和 FixedDepositDao 类，你会发现这几个类都没有实现任何 Spring 特定的接口或继承任何 Spring 指定的类。

现在让我们看看如何在配置元数据中指定应用程序对象及其依赖关系。

3. 创建配置元数据

我们在 1.3 节中了解到，Spring 容器读取指定了应用程序对象及其依赖项的配置元数据，将应用程序对象实例化并注入它们的依赖项。在本节中，我们将首先介绍配置元数据中包含的其他信息，然后深入研究如何用 XML 方式指定配置元数据。

配置元数据指定应用程序所需的企业服务（如事务管理、安全性和远程访问）的信息。例如，如果想让 Spring 来管理事务，你需要在配置元数据中配置对 Spring 的 PlatformTransactionManager 接口的一个实

现。PlatformTransactionManager 实现负责管理事务（更多关于 Spring 的事务管理功能详见第 8 章）。

如果应用程序和消息中间件（如 ActiveMQ）、数据库（如 MySQL）、电子邮件服务器等进行交互，那么这些简化了与外部系统交互的 Spring 的特定对象也是在配置元数据中定义的。例如，如果应用程序需要向 ActiveMQ 发送或接收 JMS 消息，你可以在配置元数据中配置 Spring 的 JmsTemplate 类来简化和 ActiveMQ 的交互。在程序示例 1-10 中可以看到，如果使用 JmsTemplate 向 JMS 提供者发送消息，你不需要处理低级别的 JMS API（更多关于 Spring 对与 JMS 提供者交互的支持详见第 10 章）。

你可以通过 XML 文件或者通过 POJO 类中的注解将配置元数据提供给 Spring 容器。从 Spring 3.0 版本开始，你也可以通过在 Java 类上添加 Spring 的@Configuration 注解来将配置元数据提供给 Spring 容器。在本节中，我们将介绍如何通过 XML 方式指定配置元数据。在第 6 章和第 7 章中，我们将分别介绍如何通过 POJO 类中的注解和通过对 Java 类的@Configuration 注解来配置元数据。

通过创建一个包含应用程序对象及其依赖项信息的**应用程序上下文 XML** 文件，可以按照 XML 格式将配置元数据提供给应用程序。程序示例 1-13 呈现了一个应用程序上下文 XML 文件的大体样式。下面的 XML 展示了 MyBank 应用程序的应用程序上下文 XML 文件由 FixedDepositController、FixedDepositService 以及 FixedDepositDao（见图 1-4 以了解这些对象如何相互作用）等对象组成。

程序示例 1-13 applicationContext.xml——MyBank 的应用程序上下文 XML 文件

```
Project - ch01-bankapp-xml
Source location - src/main/resources/META-INF/spring

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="controller"
            class="sample.spring.chapter01.bankapp.FixedDepositController">
       <property name="fixedDepositService" ref="service" />
    </bean>

   <bean id="service" class="sample.spring.chapter01.bankapp.FixedDepositService">
     <property name="fixedDepositDao" ref="dao" />
   </bean>

   <bean id="dao" class="sample.spring.chapter01.bankapp.FixedDepositDao"/>
</beans>
```

以下是关于应用程序上下文 XML 文件的要点。

1）在 spring-beans.xsd schema（也被称为 Spring 的 bean schema）中定义的<beans>元素是应用程序上下文的 XML 文件的根元素。spring-beans.xsd schema 在 Spring 框架发布的 spring-beans-4.3.0.RELEASE.jar JAR 包中。

2）每个<bean>元素配置一个由 Spring 容器管理的应用程序对象。在 Spring 框架的术语中，一个<bean>元素代表一个 bean 定义。Spring 容器创建的基于 bean 定义的对象称为一个 bean。id 特性指定 bean 的唯一名称，class 特性指定 bean 的完全限定类名。还可以使用<bean>元素的 name 特性来指定 bean 的别名。在 MyBank 应用程序中，FixedDepositController、FixedDepositService 和 FixedDepositDao 为应用程序对象，因此我们有三个<bean>元素——每个应用程序对象对应一个<bean>元素。由于 Spring 容器管理着由<bean>元素配置的应用程序对象，Spring 容器也就需要承担创建并注入它们的依赖关系的责任。不需要直接创建由<bean>元素定义的应用程序对象实例，而是应该从 Spring 容器中获取它们。在本节后面的部分，我们将介绍如何获取由 Spring 容器管理的应用程序对象。

3）没有和 MyBank 应用程序中的 FixedDepositDetails 域对象相对应的<bean>元素。这是因为域对象通常不是由 Spring 容器管理的，它们由应用程序所使用的 ORM 框架（如 Hibernate）创建，或者通过使用 new

运算符以编程方式创建它们。

4）<property>元素指定由<bean>元素配置的 bean 的依赖项（或者配置属性）。<property> 元素对应于 bean 类中的 JavaBean 风格的 setter 方法，该方法由 Spring 容器调用以设置 bean 的依赖关系（或配置属性）。

现在让我们来介绍一下如何通过 setter 方法注入依赖项。

4. 通过 setter 方法注入依赖项

为了理解如何通过在 bean 类中定义的 setter 方法注入依赖，我们再来观察一下 MyBank 应用程序中的 FixedDepositController 类。

程序示例 1-14 FixedDepositController 类

```
Project - ch01-bankapp-xml
Source location - src/main/java/sample/spring/chapter01/bankapp

package sample.spring.chapter01.bankapp;

import org.apache.log4j.Logger;

public class FixedDepositController {
  private static Logger logger = Logger.getLogger(FixedDepositController.class);

  private FixedDepositService fixedDepositService;

  public FixedDepositController() {
    logger.info("initializing");
  }

  public void setFixedDepositService(FixedDepositService fixedDepositService) {
    logger.info("Setting fixedDepositService property");
    this.fixedDepositService = fixedDepositService;
  }
  .....
}
```

程序示例 1-14 表明 FixedDepositController 类中声明了一个类型为 FixedDepositService、名称为 fixedDepositService 的实例变量。这个 fixedDepositService 变量由 setFixedDepositService 方法设定——一种针对 fixedDepositService 变量的 JavaBean 风格的 setter 方法。这是一个基于 setter 方法的 DI 示例，其中的 setter 方法满足依赖项。

图 1-5 描述了在 applicationContext.xml 文件中对 FixedDepositController 类的 bean 定义（见程序示例 1-13）。

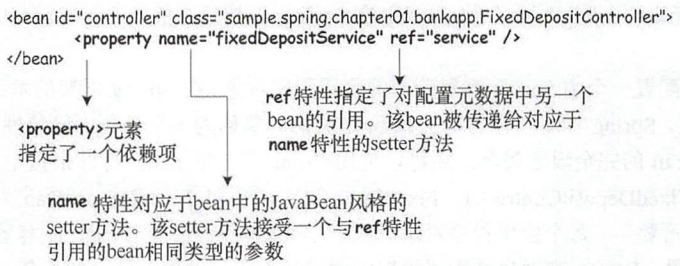


图 1-5 使用<property> 元素定义依赖项

前文的 bean 定义表明，FixedDepositController bean 通过<property>元素定义了它对于 FixedDepositService bean 的依赖。Spring 容器在 bean 创建时会调用 bean 类中 JavaBean 风格的 setter 方法，该方法与<property>元素的 name 特性对应。<property>元素的引用特性标识可以分辨需要创建具体哪个

Spring bean 的实例。引用特性的值必须与配置元数据中的<bean>元素的 id 特性值（或由 name 特性指定的名称之一）匹配。

在图 1-5 中，<property>元素的 name 特性的值为 fixedDepositService，这意味着<property>元素对应于 FixedDepositController 类（见程序示例 1-14）中 setFixedDepositService 的 setter 方法。由于<property>元素的 ref 特性的值是 service，因此<property>元素是指 id 特性的值为 service 的<bean>元素。现在，id 特性的值为 service 的<bean>元素是 FixedDepositService bean（见程序示例 1-13）。Spring 容器创建了一个 FixedDepositService 类（一个依赖项）的实例，并将 FixedDepositService 实例作为调用 FixedDepositController（一个依赖对象）的 setFixedDepositService 方法（一个用于 fixedDepositService 变量的 JavaBean 风格的 setter 方法）的参数。

在 FixedDepositController 应用程序对象的上下文中，图 1-6 总结了<property>元素的 name 和 ref 特性的用途。

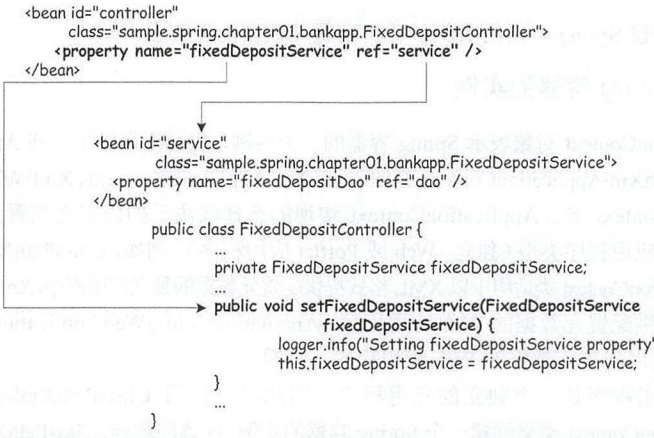


图 1-6 <property>元素的 name 特性对应于一个满足 bean 依赖关系的 JavaBean 风格的 setter 方法，ref 特性指的是另一个 bean

图 1-6 显示了 name 特性的 fixedDepositService 值对应于 FixedDepositController 类的 setFixedDepositService 方法，ref 特性的 service 值对应于 id 值为 service 的 bean。

图 1-7 总结了 Spring 容器如何根据 MyBank 应用程序的 applicationContext.xml 文件（见程序示例 1-13）提供的配置元数据创建 bean 并注入它们的依赖项。该图显示了 Spring IoC 容器创建 FixedDepositController、FixedDepositService 和 FixedDepositDao bean 并注入其依赖项的步骤顺序。在尝试创建 bean 之前，Spring 容器读取并验证由 applicationContext.xml 文件提供的配置元数据。由 Spring 容器创建 bean 的顺序取决于它们在 applicationContext.xml 文件中的定义顺序。Spring 容器确保在调用 setter 方法之前完全配置了一个 bean 的依赖关系。例如，FixedDepositController bean 依赖于 FixedDepositService bean，因此，Spring 容器会在调用 FixedDepositController bean 的 setFixedDepositService 方法之前配置好 FixedDepositService bean。

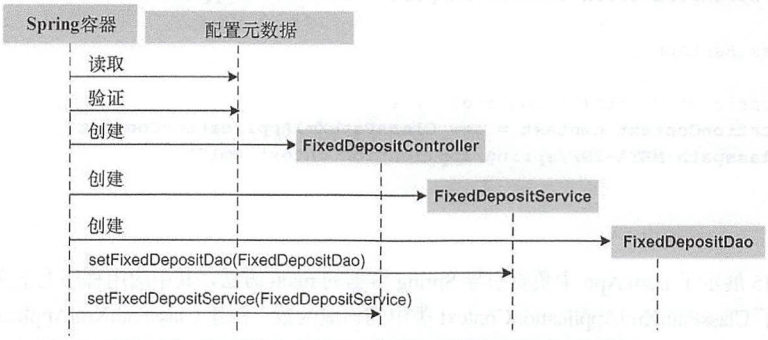


图 1-7 Spring IoC 容器创建 bean 并注入其依赖关系的顺序

注 意

通过一个 bean 的名称（id 特性的值）或类型（class 特性的值）或由 bean 类实现的接口来引用 bean 定义是相当普遍的。例如，你可以将“FixedDepositController bean”称为“控制器 bean”。而且，如果 FixedDepositController 类实现了 FixedDepositControllerIntf 接口，那么可以将“FixedDepositController bean”称为“FixedDepositControllerIntf bean”。

到目前为止，我们已经看到的这些 bean 定义指示 Spring 容器调用 bean 类的无参数构造函数来创建 bean 实例，并使用基于 setter 的 DI 来注入依赖关系。在第 2 章中，我们将介绍指示 Spring 容器通过类中定义的工厂方法创建 bean 实例的 bean 定义。另外，我们将介绍如何通过构造函数参数注入依赖关系（也称为基于构造函数的 DI）。

现在来看看如何创建 Spring 容器的实例，并将配置元数据传递给它。

5. 创建一个 Spring 容器的实例

Spring 的 ApplicationContext 对象表示 Spring 容器的一个实例。Spring 提供了一些 ApplicationContext 接口的内置实现，如 ClassPathXmlApplicationContext、FileSystemXmlApplicationContext、XmlWebApplicationContext、XmlPortletApplicationContext 等。ApplicationContext 实现的选择取决于如何定义配置元数据（使用 XML、注释或 Java 代码）以及应用程序类型（独立、Web 或 Portlet 应用程序）。例如，ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext 类适用于以 XML 格式提供配置元数据的独立应用程序，XmlWebApplicationContext 适用于以 XML 格式提供配置元数据的 Web 应用程序，AnnotationConfigWebApplicationContext 适用于通过 Java 代码以编程方式提供配置元数据的 Web 应用程序，等等。

由于 MyBank 应用程序是一个独立的应用程序，因此可以使用 ClassPathXmlApplicationContext 或 FileSystemXmlApplicationContext 类来创建一个 Spring 容器的实例。应该注意到，ClassPathXmlApplicationContext 类从指定的类路径位置加载应用程序上下文 XML 文件，FileSystemXmlApplicationContext 类从文件系统上的指定位置加载应用程序上下文 XML 文件。

MyBank 应用程序中的 BankApp 类展示了使用 ClassPathXmlApplicationContext 类创建一个 Spring 容器的实例（见程序示例 1-15）。

程序示例 1-15 BankApp 类

```
Project - ch01-bankapp-xml
Source location - src/main/java/sample/spring/chapter01/bankapp

package sample.spring.chapter01.bankapp;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class BankApp {
  .....
  public static void main(String args[]) {
     ApplicationContext context = new ClassPathXmlApplicationContext(
        "classpath:META-INF/spring/applicationContext.xml");
     .....
  }
}
```

程序示例 1-15 展示了 BankApp 中负责引导 Spring 容器的 main 方法，其中应用程序上下文 XML 文件的类路径位置传递给了 ClassPathXmlApplicationContext 类中的构造函数。创建 ClassPathXmlApplicationContext 实例的结果是在应用程序上下文 XML 文件中创建的那些 bean 都是单个范围并被设置为预实例化的。在第 2 章中，我们将讨论 bean 的范围，以及使用 Spring 容器预实例化或者延迟实例化 bean 的含义。现在，你可以

假设在 MyBank 应用程序的 applicationContext.xml 文件中定义的 bean 是 singleton 范围的，并设置为预实例化。这意味着在创建 ClassPathXmlApplicationContext 的实例时，在 applicationContext.xml 文件中定义的 bean 也会被创建。

现在我们已经看到如何创建一个 Spring 容器的实例，下面来看如何从 Spring 容器中检索 bean 实例。

6. 从 Spring 容器访问 bean

通过<bean>元素定义的应用程序对象由 Spring 容器创建和管理。可以通过调用 ApplicationContext 接口的 getBean 方法来访问这些应用程序对象的实例。

程序示例 1-16 展示了 BankApp 类的 main 方法，它从 Spring 容器中检索 FixedDepositController bean 实例并调用其方法。

程序示例 1-16 BankApp 类

```
Project - ch01-bankapp-xml
Source location - src/main/java/sample/spring/chapter01/bankapp

package sample.spring.chapter01.bankapp;

import org.apache.log4j.Logger;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class BankApp {
  private static Logger logger = Logger.getLogger(BankApp.class);

  public static void main(String args[]) {
     ApplicationContext context = new ClassPathXmlApplicationContext(
      "classpath:META-INF/spring/applicationContext.xml");

       FixedDepositController fixedDepositController =
           (FixedDepositController) context.getBean("controller");
       logger.info("Submission status of fixed deposit : " + fixedDepositController.submit());
       logger.info("Returned fixed deposit info : " + fixedDepositController.get());
  }
}
```

首先调用 ApplicationContext 的 getBean 方法，从 Spring 容器中检索 FixedDepositController bean 的一个实例，然后调用 FixedDepositController bean 的 submit 和 get 方法。要从 Spring 容器检索其实例的 bean 的名称是传递给 getBean 方法的参数。传递给 getBean 方法的 bean 的名称必须是要检索的 bean 的 id 或 name 特性的值。如果没有指定名称的 bean 注册到 Spring 容器中，getBean 方法将抛出异常。

在程序示例 1-16 中，要配置 FixedDepositController 实例，我们没有以编程方式创建 FixedDepositService 的实例并将其设置在 FixedDepositController 实例上，也没有创建一个 FixedDepositDao 的实例并将其设置在 FixedDepositService 实例上。这是因为创建依赖项并将它们注入依赖对象中的任务是由 Spring 容器处理的。

如果进入 ch01-bankapp-xml 项目并执行 BankApp 类的 main 方法，将在控制台上看到以下输出内容。

```
INFO  sample.spring.chapter01.bankapp.FixedDepositController - initializing
INFO  sample.spring.chapter01.bankapp.FixedDepositService - initializing
INFO  sample.spring.chapter01.bankapp.FixedDepositDao - initializing
INFO  sample.spring.chapter01.bankapp.FixedDepositService - Setting fixedDepositDao property
INFO sample.spring.chapter01.bankapp.FixedDepositController - Setting fixedDepositService property
INFO  sample.spring.chapter01.bankapp.BankApp - Submission status of fixed deposit : true
INFO  sample.spring.chapter01.bankapp.BankApp - Returned fixed deposit info : id :1, deposit amount :
10000.0, tenure : 365, email : someemail@something.com
```

由上面的输出可知，Spring 容器将所有在 MyBank 应用程序的 applicationContext.xml 文件中定义的 bean

都创建了一个实例。此外，Spring 容器使用基于 setter 的 DI 将 FixedDepositService 的实例注入 FixedDepositController 实例中，并将 FixedDepositDao 的实例注入 FixedDepositService 实例中。

下面来看在 Spring 框架之上构建的一些框架。

1.6 建立在 Spring 之上的框架

虽然 SpringSource 有许多以 Spring 框架作为基础的框架，但我们将介绍一些广泛流行的框架。有关更全面的框架列表以及有关单个框架的更多详细信息，建议读者自行访问 SpringSource 网站查找。

构建在 Spring 框架之上的 SpringSource 框架的高级概述见表 1-2。

表 1-2 构建在 Spring 框架之上的 SpringSource 框架的高级概述

框架	描述
Spring Security	企业应用认证和授权框架。你需要在应用程序上下文 XML 文件中配置几个 bean，以将身份验证和授权功能合并到应用程序中
Spring Data	提供一致的编程模型来与不同类型的数据库进行交互。例如，你可以使用它与非关系数据库（如 MongoDB 或 Neo4j）进行交互，还可以通过它来使用 JPA 访问关系数据库
Spring Batch	如果应用程序需要批量处理，则使用此框架
Spring Integration	为应用程序提供企业应用程序集成（EAI）功能
Spring Social	如果应用程序需要与社交媒体网站（如 Facebook 和 Twitter）进行交互，那么你将发现此框架非常有用

由于表 1-1 中提到的框架构建在 Spring 框架之上，所以在使用任何这些框架之前，请确保它们与你正在使用的 Spring 框架版本兼容。

1.7 小结

在本章中，我们介绍了使用 Spring 框架的好处。我们还研究了一个简单的 Spring 应用程序，它展示了如何在 xml 格式中指定配置元数据，如何创建 Spring 容器实例并从中获取 bean。在下一章中，我们将讨论 Spring 框架的一些基础概念。

第 2 章　Spring 框架基础

2.1　简介

在上一章中，我们看到 Spring 容器调用 bean 类的无参数构造函数来创建一个 bean 实例，而基于 setter 的 DI 用于设置 bean 依赖关系。在本章中，我们将进一步介绍以下内容：

- Spring 支持“面向接口编程”的设计方法；
- 使用静态和实例工厂方法创建 bean ；
- 基于构造函数的 DI，用于将 bean 依赖关系作为构造函数参数传递；
- 将简单的 String 值作为参数传递给构造函数和 setter 方法；
- bean 的作用域。

Spring 如何通过支持“面向接口编程”的设计方法来提高应用程序的可测试性是本章首先要介绍的内容。

2.2　面向接口编程的设计方法

在 1.5 节中，我们看到一个依赖于其他类的 POJO 类包含了对其依赖项的具体类的引用。例如，FixedDepositController 类包含对 FixedDepositService 类的引用，FixedDepositService 类包含对 FixedDepositDao 类的引用。如果这个依赖于其他类的类直接引用其依赖项的具体类，则会导致类之间的紧密耦合。这意味着如果要替换其依赖项的其他实现，则需要更改这个依赖于其他类的类本身。

我们知道 Java 接口定义了其实现类应遵循的契约。因此，如果一个类依赖于其依赖项实现的接口，那么当替换不同的依赖项实现时，类不需要改变。一个类依赖于由其依赖项所实现的接口的应用程序设计方法称为“面向接口编程”。这种设计方法使得依赖类与其依赖项之间松耦合。由依赖项类实现的接口称为依赖接口。

和“面向类编程”相比，“面向接口编程”是更加良好的设计实践，图 2-1 中的类图表明 ABean 类依赖于 BBean 接口而不是 BBeanImpl 类（BBean 接口的实现）。

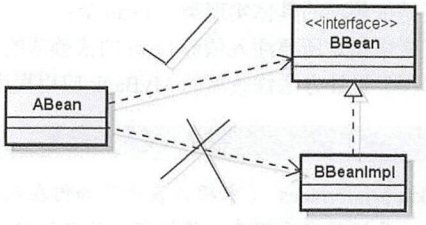


图 2-1　和“面向类编程”相比，“面向接口编程”是更加良好的设计实践

图 2-2 中的类图显示了 FixedDepositService 类如何使用“面向接口编程”的设计方法轻松地切换与数据库交互的策略。

在图 2-2 中，FixedDepositJdbcDao 单纯地使用 JDBC，而 FixedDepositHibernateDao 使用 Hibernate ORM 进行数据库交互。如果 FixedDepositService 直接依赖于 FixedDepositJdbcDao 或 FixedDepositHibernateDao，当需要切换与数据库交互的策略时，则需要在 FixedDepositService 类中进行必要的更改。FixedDepositService 依赖于 FixedDepositJdbcDao 和 FixedDepositHibernateDao 类实现 FixedDepositDao 接口（依赖接口）。现在，通过使用单纯的 JDBC 或 Hibernate ORM 框架，你可以向 FixedDepositService 实例提供 FixedDepositJdbcDao 或 FixedDepositHibernateDao 的实例。

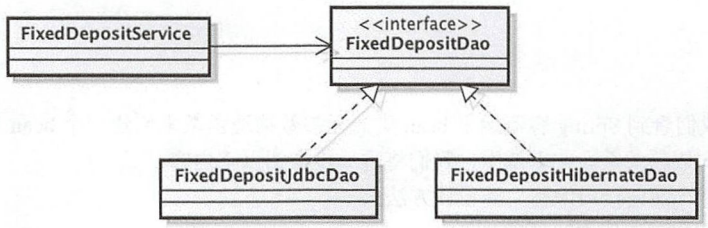


图 2-2 FixedDepositService 依赖于 FixedDepositDao 接口，由 FixedDepositJdbcDao 和 FixedDepositHibernateDao 类实现

由于 FixedDepositService 依赖于 FixedDepositDao 接口，因此将来可以支持其他数据库交互策略。如果决定使用 iBATIS（现在更名为 MyBatis）持久性框架进行数据库交互，那么可以使用 iBATIS，而不需要对 FixedDepositService 类进行任何更改，只需创建一个实现 FixedDepositDao 接口的 FixedDepositIbatisDao 类，并将 FixedDepositIbatisDao 的实例提供给 FixedDepositService 实例。

现在来看看“面向接口编程”是如何提高依赖类的可测试性的。

提高依赖类的可测试性

在图 2-2 中，FixedDepositService 类保留了对 FixedDepositDao 接口的引用。FixedDepositJdbcDao 和 FixedDepositHibernateDao 是 FixedDepositDao 接口的具体实现类。现在，为了简化 FixedDepositService 类的单元测试，我们可以把原来对具体数据库操作的实现去掉，用一个实现了 FixedDepositDao 接口但是不需要数据库的代码来代替。

如果 FixedDepositService 类直接引用 FixedDepositJdbcDao 或 FixedDepositHibernateDao 类，那么测试 FixedDepositService 类则需要设置数据库以进行测试。这表明通过对依赖接口的模拟依赖类实现，你可以减少针对单元测试的基础设施设置的工作量。

现在来看看 Spring 如何在应用程序中支持“面向接口编程”的设计方法。

Spring 对“面向接口编程”设计方法的支持

要在 Spring 应用程序中使用“面向接口编程”的设计方法，你需要执行以下操作：

- 创建引用依赖接口，而不是依赖项的具体实现类的 bean 类；
- 定义<bean>元素，并在元素中指定所要注入依赖 bean 的依赖项的具体实现类。

现在来看看根据“面向接口编程”设计方法修改后的 MyBank 应用程序。

含 义

chapter 2/ch02-bankapp-interfaces （本项目展示了如何在创建 Spring 应用程序中应用“面向接口编程”设计方法。要运行应用程序，请执行本项目的 BankApp 类的 main 方法）。

使用“面向接口编程”设计方法的 MyBank 应用程序

图 2-3 所示的类图描述了根据“面向接口编程”设计方法修改后的 MyBank 应用程序。

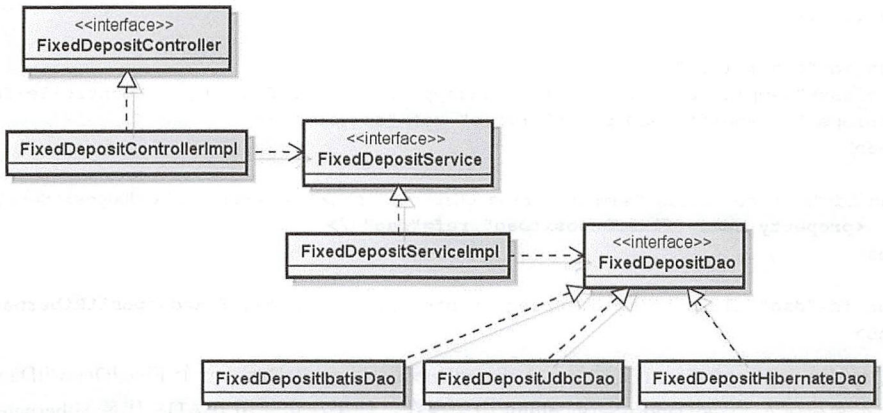


图 2-3 使用“面向接口编程”设计方法的 MyBank 应用程序

图 2-3 显示了一个类依赖于依赖项实现的接口，而不依赖于具体的依赖项实现类。例如，FixedDepositControllerImpl 类依赖于 FixedDepositService 接口，FixedDepositServiceImpl 类依赖于 FixedDepositDao 接口。

程序示例 2-1 展示了基于图 2-3 设计的 FixedDepositServiceImpl 类。

程序示例 2-1 FixedDepositService 类

```
Project - ch02-bankapp-interfaces
Source location - src/main/java/sample/spring/chapter02/bankapp

package sample.spring.chapter02.bankapp;

public class FixedDepositServiceImpl implements FixedDepositService {
  private FixedDepositDao fixedDepositDao;
  .....
  public void setFixedDepositDao(FixedDepositDao fixedDepositDao) {
    this.fixedDepositDao = fixedDepositDao;
  }

  public FixedDepositDetails getFixedDepositDetails(long id) {
    return fixedDepositDao.getFixedDepositDetails(id);
  }

  public boolean createFixedDeposit(FixedDepositDetails fdd) {
    return fixedDepositDao.createFixedDeposit(fdd);
  }
}
```

在程序示例 2-1 中，FixedDepositServiceImpl 类包含对 FixedDepositDao 接口的引用。要注入到 FixedDepositServiceImpl 实例中的 FixedDepositDao 具体实现，则在应用程序上下文 XML 文件中指定。如图 2-3 所示，可以注入以下 FixedDepositDao 接口的具体实现：FixedDepositIbatisDao、FixedDepositJdbcDao 和 FixedDepositHibernateDao。

程序示例 2-2 展示了将 FixedDepositHibernateDao 注入到 FixedDepositServiceImpl 中的 applicationContext.xml 文件。

程序示例 2-2 applicationContext.xml MyBank 的应用程序上下文 XML 文件

```
Project - ch02-bankapp-interfaces
Source location - src/main/resources/META-INF/spring

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

```
<beans .....>

  <bean id="controller"
      class="sample.spring.chapter02.bankapp.controller.FixedDepositControllerImpl">
     <property name="fixedDepositService" ref="service" />
  </bean>

  <bean id="service" class="sample.spring.chapter02.bankapp.service.FixedDepositServiceImpl">
       <property name="fixedDepositDao" ref="dao" />
  </bean>

  <bean id="dao" class="sample.spring.chapter02.bankapp.dao.FixedDepositHibernateDao"/>
</beans>
```

上述的 applicationContext.xml 文件显示了 FixedDepositHibernateDao（一个 FixedDepositDao 接口的实现）的一个实例被注入 FixedDepositServiceImpl 中。现在，如果决定使用 iBATIS 代替 Hibernate 进行持久化，那么所需要做的就是将 dao bean 定义的 class 特性修改为 FixedDepositIbatisDao 类的完全限定名。

到目前为止，我们已经介绍了 bean 定义的示例，Spring 容器通过调用 bean 类的无参构造函数来创建 bean 实例。下面来了解 Spring 容器如何使用静态或实例工厂方法来创建 bean 实例。

2.3 使用静态和实例工厂方法创建 Spring bean

Spring 容器可以创建和管理任何类的实例，而不管类是否提供无参数构造函数。在 2.4 节中，我们将介绍在构造函数中可以接受一个或多个参数的 bean 类的定义。如果现有的 Java 应用程序用工厂类来创建对象实例，那么仍然可以使用 Spring 容器来管理由这些工厂创建的对象。

现在来介绍一下 Spring 容器如何调用类的静态或实例工厂方法来管理返回的对象实例。

1. 通过静态工厂方法实例 bean

图 2-3 展示了如何使用 FixedDepositHibernateDao、FixedDepositIbatisDao 和 FixedDepositJdbcDao 类实现 FixedDepositDao 接口。程序示例 2-3 中的 FixedDepositDaoFactory 类定义了一个静态工厂的方法，该静态方法根据传入的参数来创建和返回 FixedDepositDao 实例。

程序示例 2-3 FixedDepositDaoFactory 类

```
public class FixedDepositDaoFactory {
  private FixedDepositDaoFactory() { }

   public static FixedDepositDao getFixedDepositDao(String daoType, ...) {
    FixedDepositDao fixedDepositDao = null;

    if("jdbc".equalsIgnoreCase(daoType)) {
        fixedDepositDao = new FixedDepositJdbcDao();
    }
    if("hibernate".equalsIgnoreCase(daoType)) {
        fixedDepositDao = new FixedDepositHibernateDao();
    }
    .....
    return fixedDepositDao;
  }
}
```

如程序示例 2-3 所示，FixedDepositDaoFactory 类定义了一个 getFixedDepositDao 静态方法，该方法根据 daoType 参数的值创建并返回 FixedDepositJdbcDao、FixedDepositHibernateDao 或 FixedDepositIbatisDao 类的实例。

在程序示例 2-4 中，FixedDepositDaoFactory 类的 bean 定义指示 Spring 容器调用 FixedDepositDaoFactory 的 getFixedDepositDao 方法，以获取 FixedDepositJdbcDao 类的实例。

程序示例 2-4 FixedDepositDaoFactory 类的 bean 定义

```
<bean id="dao" class="sample.spring.FixedDepositDaoFactory"
        factory-method="getFixedDepositDao">
    <constructor-arg index=" 0" value="jdbc"/>
    ...
</bean>
```

在上述 bean 定义中，class 特性指定了定义静态工厂方法的类的完全限定名称。factory-method 特性指定了 Spring 容器调用的获取 FixedDepositDao 对象实例的静态工厂方法的名称。<constructor-arg>元素在 Spring 的 bean schema 中定义，用于传递构造函数的参数以及静态和实例工厂方法的参数。index 特性指的是构造函数中，也可以是静态或实例工厂方法的参数的位置。在上述 bean 定义中，index 特性值为 0 意味着<constructor-arg>元素为 getFixedDepositDao 工厂方法的第一个参数（即 daoType），而 value 特性指定了参数值。如果工厂方法接受多个参数，则需要为每个参数定义一个<constructor-arg>元素。

需要着重注意的是，调用 ApplicationContext 的 getBean 方法来获取 dao bean（见程序示例 2-4）将会调用 FixedDepositDaoFactory 的 getFixedDepositDao 工厂方法。这意味着调用 getBean（“dao”）返回由 getFixedDepositDao 工厂方法创建的 FixedDepositDao 实例，而不是 FixedDepositDaoFactory 类的实例。

现在我们已经看到创建了一个 FixedDepositDao 实例的工厂类的配置，程序示例 2-5 将展示如何将 FixedDepositDao 的实例注入 FixedDepositServiceImpl 类中。

程序示例 2-5 注入由静态工厂方法创建的对象

```
<bean id="service" class="sample.spring.chapter02.bankapp.FixedDepositServiceImpl">
    <property name=“fixedDepositDao" ref="dao" />
</bean>

<bean id="dao" class="sample.spring.chapter02.basicapp.FixedDepositDaoFactory"
     factory-method="getFixedDepositDao">
     <constructor-arg index=" 0" value="jdbc"/>
</bean>
```

在程序示例 2-5 中，<property>元素将 FixedDepositDaoFactory 的 getFixedDepositDao 工厂方法返回的 FixedDepositDao 实例注入 FixedDepositServiceImpl 实例中。如果将上面显示的 FixedDepositServiceImpl 类的 bean 定义与程序示例 2-2 中所示的 bean 定义进行对比，你会发现它们完全相同。这表明，无论 Spring 容器如何（使用无参数构造函数或静态工厂方法）创建 bean 实例，bean 的依赖项都会以相同的方式指定。

现在来介绍一下 Spring 容器是如何通过调用实例工厂方法来将 bean 实例化的。

2. 通过实例工厂方法实例化 bean

程序示例 2-6 展示了 FixedDepositDaoFactory 类，它定义了用于创建和返回 FixedDepositDao 实例的实例工厂方法。

程序示例 2-6 FixedDepositDaoFactory 类

```
public class FixedDepositDaoFactory {
  public FixedDepositDaoFactory() {
  }

  public FixedDepositDao getFixedDepositDao(String daoType, ...) {
    FixedDepositDao fixedDepositDao = null;

    if("jdbc".equalsIgnoreCase(daoType)) {
        fixedDepositDao = new FixedDepositJdbcDao();
    }
```

```
    if("hibernate".equalsIgnoreCase(daoType)) {
        fixedDepositDao = new FixedDepositHibernateDao();
    }
    .....
    return fixedDepositDao;
  }
}
```

如果类定义了一个实例工厂方法，则该类必须定义一个 public 构造函数，以便 Spring 容器可以创建该类的实例。在程序示例 2-6 中，FixedDepositDaoFactory 类定义了一个 public 无参构造函数。FixedDepositDaoFactory 的 getFixedDepositDao 方法是一个创建并返回 FixedDepositDao 实例的实例工厂方法。

程序示例 2-7 展现了如何指示 Spring 容器调用 FixedDepositDaoFactory 的 getFixedDepositDao 方法来获取 FixedDepositDao 的一个实例。

程序示例 2-7 调用 FixedDepositDaoFactory 的 getFixedDepositDao 方法的配置

```
<bean id="daoFactory" class="sample.spring.chapter02.basicapp.FixedDepositDaoFactory" />

<bean id="dao" factory-bean="daoFactory" factory-method="getFixedDepositDao">
    <constructor-arg index="0" value="jdbc"/>
</bean>

<bean id="service" class="sample.spring.chapter02.bankapp.FixedDepositServiceImpl">
   <property name="fixedDepositDao" ref="dao" />
</bean>
```

在程序示例 2-7 中，FixedDepositDaoFactory 类（包含实例工厂方法的类）被配置为常规的 Spring bean，并且使用单独的<bean>元素来配置实例工厂方法的详细信息。要配置实例工厂方法的详细信息，请使用<bean>元素的 factory-bean 和 factory-method 特性。factory-bean 特性是指定义实例工厂方法的 bean、factory-method 特性指定实例工厂方法的名称。在程序示例 2-7 中，<property>元素将 FixedDepositDaoFactory 的 getFixedDepositDao 工厂方法返回的 FixedDepositDao 实例注入 FixedDepositServiceImpl 实例中。

与 static 工厂方法一样，可以使用<constructor-arg>元素将参数传递给实例工厂方法。注意，在程序示例 2-7 中，调用 ApplicationContext 的 getBean 方法获取 dao bean 将会导致调用 FixedDepositDaoFactory 的 getFixedDepositDao 工厂方法。

下面介绍如何设置由静态和实例工厂方法创建的 bean 的依赖项。

注入由工厂方法创建的 bean 的依赖项

可以将 bean 依赖项作为参数传递给工厂方法，也可以使用基于 setter 的 DI 来注入由静态或实例工厂方法返回的 bean 实例的依赖项。

用于定义 databaseInfo 特性的 FixedDepositJdbcDao 类如程序示例 2-8 所示。

程序示例 2-8 FixedDepositJdbcDao 类

```
public class FixedDepositJdbcDao {
  private DatabaseInfo databaseInfo;
  .....
  public FixedDepositJdbcDao() { }

  public void setDatabaseInfo(DatabaseInfo databaseInfo) {
    this. databaseInfo = databaseInfo;
  }
  .....
}
```

在程序示例 2-8 中，databaseInfo 表示通过 setDatabaseInfo 方法赋值的 FixedDepositJdbcDao 类的依赖项。

FixedDepositDaoFactory 类定义了一个负责创建和返回 FixedDepositJdbcDao 类的实例的工厂方法，如程序示例 2-9 所示。

程序示例 2-9 FixedDepositDaoFactory 类

```
public class FixedDepositDaoFactory {
  public FixedDepositDaoFactory() {
  }

  public FixedDepositDao getFixedDepositDao(String daoType) {
    FixedDepositDao fixedDepositDao = null;

    if("jdbc".equalsIgnoreCase(daoType)) {
        fixedDepositDao = new FixedDepositJdbcDao();
    }
    if("hibernate".equalsIgnoreCase(daoType)) {
        fixedDepositDao = new FixedDepositHibernateDao();
    }
    .....
    return fixedDepositDao;
  }
}
```

在程序示例 2-9 中，getFixedDepositDao 方法是用于创建 FixedDepositDao 实例的实例工厂方法。如果 daoType 参数的值为 jdbc，则 getFixedDepositDao 方法将创建一个 FixedDepositJdbcDao 的实例。请注意，getFixedDepositDao 方法没有设置 FixedDepositJdbcDao 实例的 databaseInfo 特性。

正如我们在程序示例 2-7 中看到的，bean 定义指示 Spring 容器通过调用 FixedDepositDaoFactory 类的 getFixedDepositDao 实例工厂方法来创建 FixedDepositJdbcDao 的实例，如程序示例 2-10 所示。

程序示例 2-10 调用 FixedDepositDaoFactory 的 getFixedDepositDao 方法的配置

```
<bean id="daoFactory" class="FixedDepositDaoFactory" />

<bean id="dao" factory-bean="daoFactory" factory-method="getFixedDepositDao">
    <constructor-arg index="0" value="jdbc"/>
</bean>
```

dao bean 定义指示 Spring 容器调用 FixedDepositDaoFactory 的 getFixedDepositDao 方法，该方法创建并返回 FixedDepositJdbcDao 的实例。但是，FixedDepositJdbcDao 的 databaseInfo 特性并没有设置。如果需要设置 databaseInfo 特性，可以在 getFixedDepositDao 方法返回的 FixedDepositJdbcDao 实例上执行基于 setter 的 DI，如程序示例 2-11 所示。

程序示例 2-11 调用 FixedDepositDaoFactory 的 getFixedDepositDao 方法并设置返回的 FixedDepositJdbcDao 实例的 databaseInfo 特性的配置

```
<bean id="daoFactory" class="FixedDepositDaoFactory" />

<bean id="dao" factory-bean="daoFactory" factory-method="getFixedDepositDao">
    <constructor-arg index="0" value="jdbc"/>
    <property name="databaseInfo" ref="databaseInfo"/>
</bean>

<bean id="databaseInfo" class="DatabaseInfo" />
```

在程序示例 2-11 的 bean 定义中，<property>元素用于设置由 getFixedDepositDao 实例工厂方法返回的 FixedDepositJdbcDao 实例的 databaseInfo 特性。

注 意

与实例工厂方法一样，可以使用<property>元素将依赖关系注入静态工厂方法返回的 bean 实例中。

2.4 基于构造函数的 DI

在 Spring 中，依赖注入是通过将参数传递给 bean 的构造函数和 setter 方法来实现的。我们在前面的章节中介绍过，通过 setter 方法注入依赖的 DI 技术称为基于 setter 的 DI。在本节中，我们将介绍依赖项作为构造函数参数传递的 DI 技术（又称为基于构造函数的 DI）。

下面通过一个例子比较一下在基于 setter 的 DI 和基于构造函数的 DI 技术中指定 bean 依赖项的区别。

1. 回顾基于 setter 的 DI

在基于 setter 的 DI 中，<property>元素用于指定 bean 依赖项。假设 MyBank 应用程序包含一个 PersonalBankingService 服务，该服务允许客户检索银行账户对账单、检查银行账户明细、更新联系电话、更改密码和联系客户服务。PersonalBankingService 类使用 JmsMessageSender（用于发送 JMS 消息）、EmailMessageSender（用于发送电子邮件）和 WebServiceInvoker（用于调用外部 Web 服务）对象来完成其预期功能。程序示例 2-12 展示了 PersonalBankingService 类。

程序示例 2-12 PersonalBankingService 类

```
public class PersonalBankingService {
  private JmsMessageSender jmsMessageSender;
  private EmailMessageSender emailMessageSender;
  private WebServiceInvoker webServiceInvoker;
  .....
  public void setJmsMessageSender(JmsMessageSender jmsMessageSender) {
    this.jmsMessageSender = jmsMessageSender;
  }

  public void setEmailMessageSender(EmailMessageSender emailMessageSender) {
    this.emailMessageSender = emailMessageSender;
  }

  public void setWebServiceInvoker(WebServiceInvoker webServiceInvoker) {
    this.webServiceInvoker = webServiceInvoker;
  }
  .....
}
```

在程序示例 2-12 中，PersonalBankingService 类的每个依赖项（JmsMessageSender、EmailMessageSender 和 WebServiceInvoker）都定义了一个 setter 方法。

PersonalBankingService 类为其依赖项定义了 setter 方法，因此使用了基于 setter 的 DI，如程序示例 2-13 所示。

程序示例 2-13 PersonalBankingService 类的 bean 定义及其依赖项

```
<bean id="personalBankingService" class="PersonalBankingService">
  <property name="emailMessageSender" ref="emailMessageSender" />
  <property name="jmsMessageSender" ref="jmsMessageSender" />
  <property name="webServiceInvoker" ref="webServiceInvoker" />
</bean>

<bean id="jmsMessageSender" class="JmsMessageSender">
  .....
</bean>
<bean id="webServiceInvoker" class="WebServiceInvoker" />
  .....
```

```
  </bean>
  <bean id="emailMessageSender" class="EmailMessageSender" />
    .....
  </bean>
```

在 PersonalBankingService bean 的定义中，为 PersonalBankingService 类的每个依赖项都指定了一个<property>元素。

下面介绍如何使用基于构造函数的 DI 来对 PersonalBankingService 类建模。

2. 基于构造函数的 DI

在基于构造函数的 DI 中，bean 的依赖项作为参数传递给 bean 类的构造函数。程序示例 2-14 展示了一个 PersonalBankingService 类的修改版本，其构造函数接收 JmsMessageSender、EmailMessageSender 和 WebServiceInvoker 对象。

程序示例 2-14 PersonalBankingService 类

```
public class PersonalBankingService {
  private JmsMessageSender jmsMessageSender;
  private EmailMessageSender emailMessageSender;
  private WebServiceInvoker webServiceInvoker;
  .....
  public PersonalBankingService(JmsMessageSender jmsMessageSender,
    EmailMessageSender emailMessageSender,
    WebServiceInvoker webServiceInvoker) {

    this.jmsMessageSender = jmsMessageSender;
    this.emailMessageSender = emailMessageSender;
    this.webServiceInvoker = webServiceInvoker;
  }
  .....
}
```

PersonalBankingService 类的构造函数的参数代表 PersonalBankingService 类的依赖项。程序示例 2-15 展示了如何通过<constructor-arg>元素来提供这些依赖项。

程序示例 2-15 PersonalBankingService 的 bean 定义

```
<bean id="personalBankingService" class="PersonalBankingService">
  <constructor-arg index="0" ref="jmsMessageSender" />
  <constructor-arg index="1" ref="emailMessageSender" />
  <constructor-arg index="2" ref="webServiceInvoker" />
</bean>

<bean id="jmsMessageSender" class="JmsMessageSender">
  .....
</bean>
<bean id="webServiceInvoker" class="WebServiceInvoker" />
  .....
</bean>
<bean id="emailMessageSender" class="EmailMessageSender" />
  .....
</bean>
```

在程序示例 2-15 中，<constructor-arg>元素指定了传递给 PersonalBankingService 实例的构造函数参数的详细信息。index 特性指定了构造函数参数中的索引： 如果 index 特性值为 0，则表示<constructor-arg>元素对应于第一个构造函数参数；如果 index 特性值为 1，则表示<constructor-arg>元素对应于第二个构造函数参数，以此类推。如果构造函数参数与继承无关，则不需要指定 index 特性。例如，如果 JmsMessageSender、WebServiceInvoker 和 EmailMessageSender 是不同的对象，则不需要指定 index 特性。与< property >元素的

情况一样，<constructor-arg>元素的 ref 特性用于传递对 bean 的引用。

下面介绍如何结合使用基于构造函数的 DI 以及基于 setter 的 DI。

基于构造函数和基于 setter 的 DI 机制的结合使用

如果 bean 类需要结合使用基于构造函数的 ID 机制和基于 setter 的 DI 机制，则可以使用<constructor-arg>和<property>元素的组合来注入依赖关系。

程序示例 2-16 展示了 PersonalBankingService 类的一个版本，其依赖项作为参数注入构造函数和 setter 方法中。

程序示例 2-16 PersonalBankingService 类

```
public class PersonalBankingService {
  private JmsMessageSender jmsMessageSender;
  private EmailMessageSender emailMessageSender;
  private WebServiceInvoker webServiceInvoker;
  .....
  public PersonalBankingService(JmsMessageSender jmsMessageSender,
         EmailMessageSender emailMessageSender) {
    this.jmsMessageSender = jmsMessageSender;
    this.emailMessageSender = emailMessageSender;
  }

  public void setWebServiceInvoker(WebServiceInvoker webServiceInvoker) {
    this.webServiceInvoker = webServiceInvoker;
  }
  .....
}
```

在 PersonalBankingService 类中，jmsMessageSender 和 emailMessageSender 依赖项作为构造函数注入，而 webServiceInvoker 依赖关系通过 setWebServiceInvoker setter 方法注入。以下 bean 定义表明，<constructor-arg>和<property>元素用于注入 PersonalBankingService 类的依赖项，如程序示例 2-17 所示。

程序示例 2-17 基于构造函数和基于 setter 两种 DI 机制的结合

```
<bean id="dataSource" class="PersonalBankingService">
    <constructor-arg index="0" ref="jmsMessageSender" />
    <constructor-arg index="1" ref="emailMessageSender" />
    <property name="webServiceInvoker" ref="webServiceInvoker" />
</bean>
```

可以看到，<property>和<constructor-arg>元素用于传递依赖项（对其他 bean 的引用）到 setter 方法和构造函数中。也可以使用这些元素来传递 bean 所需的配置信息（单纯的 String 值）。

2.5 将配置详细信息传递给 bean

以 EmailMessageSender 类为例，在该类中需要使用电子邮件服务器地址、用户名和密码三项来对连接电子邮件服务器进行身份验证。可以使用<property>元素设置 EmailMessageSender bean 的属性，如程序示例 2-18 所示。

程序示例 2-18 EmailMessageSender 类和相应的 bean 定义

```
public class EmailMessageSender {
  private String host;
  private String username;
  private String password;
  .....
```

```
  public void setHost(String host) {
    this.host = host;
  }

  public void setUsername(String username) {
    this.username = username;
  }

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

```
  <bean id="emailMessageSender" class="EmailMessageSender">
       <property name="host" value="smtp.gmail.com"/>
       <property name="username" value="myusername"/>
       <property name="password" value="mypassword"/>
  </bean>
```

在程序示例 2-18 中，我们已经使用<property>元素来设置 EmailMessageSender bean 的主机（host）、用户名（username）和密码（password）特性。由 name 特性标识的 bean 特性的 String 值通过 value 特性来指定。主机、用户名和密码特性表示 EmailMessageSender bean 所需的配置信息。在第 3 章中，我们将看到如何设置原始类型（如 int、long 等）、集合类型（如 java.util.List、java.util.Map 等）和自定义类型（如地址）的特性。

程序示例 2-19 展示了将配置信息（如主机、用户名和密码）作为构造函数参数接收的 EmailMessageSender 类（以及相应的 bean 定义）的修改版本。

程序示例 2-19　EmailMessageSender 类和相应的 bean 定义

```
public class EmailMessageSender {
  private String host;
  private String username;
  private String password;
   .....
   public EmailMessageSender(String host, String username, String password) {
     this.host = host;
     this.username = username;
     this.password = password;
   }
   .....
}
```

```
   <bean id="emailMessageSender" class="EmailMessageSender">
        <constructor-arg index="0" value="smtp.gmail.com"/>
        <constructor-arg index="1" value="myusername"/>
        <constructor-arg index="2" value="mypassword"/>
   </bean>
```

在程序示例 2-19 中，<constructor-arg>元素用于传递 EmailMessageSender bean 所需的配置详细信息。由 index 特性标识的构造函数参数的 String 值通过 value 特性来指定。

到目前为止，我们已经看到，<constructor-arg>元素用于注入 bean 依赖项，并为 String 类型构造函数参数传递值。在第 3 章中，我们将看到如何为原始类型（如 int、long 等），集合类型（如 java.util.List、java.util.Map 等）以及自定义类型（如地址）的构造函数参数指定值。

现在我们已经了解了如何指示 Spring 容器创建 bean 并执行 DI，接下来介绍 bean 的各种作用域。

2.6 bean 的作用域

你可能需要指定一个 bean 的范围，以控制所创建的 bean 实例是否可以共享（singleton 范围），还是每

有关此电子书试读版的说明

本人可以帮助你找到你要的高清 PDF 电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融等等。

质量都很清晰，为方便读者阅读观看，每本 100°/。都带可跳转的书签索引和目录。

一般情况下，出版半年左右就会有 PDF 极个别的书出 PDF 时间要长一些

如看到试读版信息.说明已经有完整版，需求完整版即可联系我。
请添加 QQ 312082710 和微信312082710

或扫描微信二维码添加

PDF 代找说明：

本人已经帮助了上万人找到了他们需要的 **PDF** ，其实网上有很多 **PDF**,

大家如果在网上不到的话，可以联系我**QQ** 312082710或微信 312082710

大部分我都可以找到，而且每本 100%带书签索引目录。
因 **PDF** 电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

提供各种书籍的高清 **PDF** 电子版代找服务，如果你找不到自己想要的书的 **PDF** 电子版，我们 可以帮您找到，如有需要，请联系 **QQ** 2028969416 微信 2028969416 备用 **QQ** 202896416
若以上联系方式失效，您可通过以下电子邮件获取有效联系方式。邮箱：312082710@qq.com

若您没有 **QQ** 通讯工具，请点击下面链接与客服取得联系。&

https://item.taobao.com/item.htm?id=565681036651

声明：本人只提供代找服务，每本 100%索弓|书签和目录，因寻找和后期制作 **pdf** 电子书有一定难度
仅收取代找费用。如因 **PDF** 产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的 **PDF** 而已。