CHAPTER 4

Introducing Inversion of Control
第 4 章　Spring 基础

控制反转介绍

在第一章中，在我们开始讨论控制反转（IoC）时，也许你能回忆起来，它被 Martin Fowler 冠以一个更加贴切的新名称依赖注入（DI）。可是，这也并不是很准确；事实上，DI 只不过是 IoC 的一种形式，尽管你经常能够发现两个概念在应用时可以相互替换。在这一章，我们将会更加详细的考察一下 IoC 和 DI，正式的定义两个概念之间的关系，同时详细的分析 Spring 是如何实现这种概念的。

在此声明，本章中IoC即控制反转，DI即依赖注入，在阐述时我们尽量使用IoC、DI这样的称呼。

在定义两个概念并且分析了Spring与他们的关系后，我们将会探究一下对于Spring实现DI非常重要的那些概念。这一章只介绍Spring的DI的基本实现；我们将会在第5章进一步讨论DI的高级技巧，DI在应用程序设计的上下文（context）中的应用将会在第5章和第11张中展开。明确的说，本章中讲会涵盖如下主题：

控制反转的概念：在这部分中，我们会讨论各种形式的控制反转（IoC），包括依赖注入（Dependency Injection，DI）和依赖查找（Dependency Lookup）。这部分分析了各种IoC方法之间的不同，同时阐述了每种形式的优点和缺点。

Spring中的依赖注入：在这部分中，分析了Spring提供的IoC功能及这些功能是如何实现的。特别的，这部分分析了依赖注入与Spring提供的基于setter和基于构造方法的实现。这部分还首次讨论了BeanFactory 接口，它是Spring框架的核心部分。

Spring BeanFactories的XML配置：本章的最后一部分专注于介绍使用基于XML文件的配置方法来配置BeanFactory。这部分从DI配置的讨论开始，进而分析BeanFactory提供的附加服务，例如Bean继承、生命周期控制、自动装配（autowiring）等。

控制反转和依赖注入

IoC 或者 DI 的核心思想在于提供一个更加简单的机制来规定组件之间的依赖关系（一般涉及到对象间的合作），并且在它们生命周期中对依赖关系进行管理。一个需要特定的依赖的组件一般会涉及到一个依赖对象，在 IoC 的概念中叫做目标（target）。这是很主要的说明，IoC 提供了这样的服务，使一个组件能够在它的整个生命周期中访问它的关联和服务，用这种方法与它的依赖进行交互。总的来说，IoC 能够被分解为两种子类型：依赖注入和依赖查找。这两种子类型在具体实现 IoC 服务的时候被进一步分解。从这个定义上，你能够清晰的看到当我们谈及 DI 的时候总要说 IoC，但是当我们说到 IoC 的时候我们却不一定会说到 DI。

控制反转的类型

你可能想知道为什么会有两种不同的IoC类型，为什么这些类型又会进一步分解为不同的实现。这好像没有一个清晰的答案；的确，这些不同的类型提供灵活的弹性，但是对于我们，IoC看起来更象是新旧概念的一个混合体；这两种不同类型的IoC说明了这个问题。

依赖查找是一种更加传统的方法，第一眼看上去，Java程序员们对它都很熟悉。依赖注入相对新一些，方法还没有完全定型，虽然第一眼看上去有些反常，事实上它比依赖查找更加富有弹性且有用。

在依赖查找风格的IoC中，一个组件必须获得一个依赖的参考，反观在依赖注入风格中，依赖关系由IoC容器从字面意义上注入到组件中。依赖查找有两种类型：依赖托拽和上下文配置依赖查找。依赖注入同样也有两种常见的风格：构造器依赖注入和Setter依赖注入。

> 注意：在本章讨论中，我们并不关心虚构的 IoC 容器如何得到所有不同的依赖，只是在某些方面，为了描述每种机制来展现具体行为过程。

依赖托拽

对于一个Java开发者，依赖托拽是最熟悉的一种IoC。在依赖托拽中，依赖关系是根据需要从一个登记处获取（托拽）下来。任何一个写过访问EJB代码的程序员都使用过依赖托拽。Spring提供了依赖托拽作为从框架管理的组件中重新获取组件的一种机制；你可以在第2章中看到这种实践。Listing4-1展现了一个基于Spring的应用程序中典型的依赖托拽。

Listing 4-1. Spring 中的依赖托拽

```
public static void main(String[] args) throws Exception {
   // get the bean factory
   BeanFactory factory = getBeanFactory();
   MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
   mr.render();
}
```

这种类型的IoC不仅流行于J2EE应用中，在从注册处获得依赖关系的JNDI查找（JNDI lookups）中也得到广泛应用，它在使用Spring的许多环境中也起着关键的作用。

上下文配置依赖查找

上下文配置依赖查找（CDL）也类似，一些地方与依赖托拽相仿。但是在CDL中，查找是在容器管理的资源中进行的，而不是从一个集中的注册处，同时它一般在规定点执行。CDL通过类似Listing 4-2中的形式来工作，组件实现一个特定的接口。

Listing 4-2. 提供 CDL 的组件接口

```
package com.apress.prospring.ch4;
public interface ManagedComponent {
    public void performLookup(Container container);
}
```

通过实现这个接口，组件通知容器它需要获取依赖关系。当容器准备好传递依赖到组件中去，它会依次调用各个组件的performLookup()方法。组件这个时候就能够通过容器的接口来查找依赖关系，就像Listing 4-3中的例子。

Listing 4-3. 在 CDL 中获取依赖关系

```
package com.apress.prospring.ch4;
public class ContextualizedDependencyLookup implements ManagedComponent {
    private Dependency dep;
    public void performLookup(Container container) {
      this.dep = (Dependency) container.getDependency("myDependency");
   }
}
```

构造器依赖注入

构造器依赖注入就是在组件的构造器处提供依赖关系的注入。这种组件声明一个构造器或者一组构造器从构造参数中获取依赖关系，IoC 容器会在实例化它的时候将依赖关系传送给它，例如 Listing 4-4 的形式。

Listing 4-4. 构造器依赖注入

```
package com.apress.prospring.ch4;
public class ConstructorInjection {
    private Dependency dep;

   public ConstructorInjection(Dependency dep) {
      this.dep = dep;
   }
}
```

Setter 依赖注入

在Setter依赖注入中，IoC容器通过JavaBean形式的方法将组件的依赖关系注入到组件中。组件

的Setter方法将一组依赖关系暴露给IoC容器，并受之控制。Listing 4-5展示了一个典型的基于Setter依赖注入的组件。

Listing 4-5. Setter 依赖注入

```
package com.apress.prospring.ch4;
public class SetterInjection {
   private Dependency dep;

   public void setMyDependency(Dependency dep) {
      this.dep = dep;
   }
}
```

在容器中，依赖需要通过setMyDependency()方法来暴露出来，它是myDependency遵循JavaBeans风格命名的方法。实际上，setter注入是最广泛使用的注入机制，它也是最易于实现的IoC机制中之一。

注入 对 查找

在IoC中选择应用哪种风格，是“ 注入 ”还是“ 查找 ”，一般来说这并不难抉择。大部分情况下，IoC的类型一般委由你所使用的容器来决定。例如，如果你在使用EJB2.0，那么你就必须使用J2EE容器提供的“ 查找 ”风格的IoC。在Spring中，除了初始化时的bean查找，你的组件和它们之间的依赖关系都可以通过“ 注入 ”风格的IoC来连接到一起。

> 注意：当你使用 Spring 时，你可以在访问 EJB 资源时不必进行明确的查找操作。Spring 可以在查找风格的 IoC 与注入风格的 IoC 系统之间扮演一个适配器的角色，这样你就可以完全通过注入方式来管理所有资源了。

问题在于：如果非要做个选择，你会是用哪种方式，注入还是查找？大部分情况下当然会选择注入。如果你看看Listing 4-4与4-5的代码，可以清楚地看到应用注入对你的代码没有任何影响。反观依赖托拽的代码，它必须实际包含一个对注册处（registry）的引用并且要与之交互来获取依赖关系，使用上下文配置依赖查找（CDL），需要你的类实现一个特定的接口然后手动查找所有的依赖关系。如果你使用注入，你的类所需要做的就是允许依赖通过构造器或者setters注入。

使用依赖，你可以自由的使用完全与IoC容器解耦的类，可以手动设定各个独立对象之间的协作关系。反之，在查找的方式下，你的类总是依赖于容器定义的特定类和接口。查找的另一个缺点是很难脱离容器来测试你的代码。使用注入，测试你的组件将变得非常轻松，因为你只需要通过适当的构造器和setter来设定它们之间的依赖关系。

> 注意：关于通过 Spring 和依赖注入来测试的详细讨论包括在附录 A 中。

基于查找的解决方案要比基于注入的方案更复杂。虽然我们并不害怕复杂性，但是添加大量无用的复杂机制到你的程序中并以此作为依赖关系管理的核心工作，我们认为这样是错误的。

暂且不论这些缺点，最大的问题是选择注入来代替查找将会使你的生活更加轻松。如果使用注入你可以节省大量的代码，并且你写的代码将会非常简单，甚至，一般来说这些代码可以通过IDE自动生成。你会注意到所有的演示注入的代码都是被动的，这样它们就不需要主动尝试完成任务；你会发现在注入的代码中最激动人心的是对象被保存在一个统一的地方——那里不会产生什么错误！被动的代码比主动的更容易维护，因为那里很少会产生错误。可以思考一下下面这个例子的代码Listing 4-3：

```
    public void performLookup(Container container) {
      this.dep = (Dependency) container.getDependency("myDependency");
  }
    public void performLookup(Container container) {
      this.dep = (Dependency) container.getDependency("myDependency");
  }
```

在这段代码中，很多地方可能出问题：依赖的关键词（key）可能改变，容器的实例可能为null，或者返回的依赖可能是错误的形式。我们参考这段代码是因为这里有太多活动的部分，因为有很多地方可能发生变数。使用查找也许可以将你的应用程序的组件解耦，添加进来的代码用来将这些组件一一连接起来去完成一些有用得任务，但是这些代码提升了应用的复杂性。

Setter 注入 对 构造器注入

现在我们已经确定哪种IoC方法更加优越，我们还需要选择使用setter注入还是构造器注入。构造器注入在你需要实例化存在依赖的类之前就使用这些组件的时候特别有用。很多容器，包括Spring，在使用setter注入的时候提供了一种检测依赖关系是否已经定义的机制，但是，如果使用构造器依赖注入时你却在容器无法察觉的情况下声明（assert）需要的依赖关系。

Setter注入在很多不同情况下都有用。如果这个组件对容器暴露了他的依赖关系，但是也愿意自己提供一个默认依赖关系，那么setter注入一般是这种情况下最好的实现方法。Setter注入的另一个优点是它允许以来关系被声明为借口，虽然这并不像一开始想象的那么有用。想象一个典型的商业逻辑接口，它包括一个商业逻辑方法defineMeaningOfLife()。如果，对于这个方法你追加定义一个setter注入，例如setEncylopedia()，那样就以为这你要求所有的具体实现都必须使用或者至少知道对encyclopedia的依赖。事实上你并不需要定义这些setter依赖注入，现有的任何IoC容器，包括Spring，都可以与应用了商业逻辑接口的组件一同工作，仍然对实现了这些接口的类提供依赖关系。一个使用这种方法的例子可以简单阐明这个问题。思考一下Listing 4-6里面的商业接口。

Listing 4-6. Oracle 接口

```
package com.apress.prospring.ch4;
public interface Oracle {
    public String defineMeaningOfLife();
}
```

注意商业逻辑没有定义任何依赖注入的 setters 方法。这个接口在 Listing 4-7 中会被实现。

Listing 4-7. 实现 Oracle 接口

```
package com.apress.prospring.ch4;
public class BookwormOracle implements Oracle {
    private Encyclopedia enc;

    public void setEncyclopedia(Encyclopedia enc) {
       this.enc = enc;
   }
    public String defineMeaningOfLife() {
       return "Encyclopedias are a waste of money - use the Internet";
   }
}
```

如你所见，BookwormOracle不只实现了Oracle接口，同时还定义了setter依赖注入。Spring在处理这类结构的时候异常舒适，完全不需要在商务逻辑接口里定义依赖关系。使用接口来定义依赖的能力一般是setter注入所经常吹捧的，但是实际上，你应该努力保持setters注入的应用于商务逻辑接口保持独立。除非你非常肯定所有的特定商务逻辑接口的实现都需要与之对应的依赖关系，否则，最好让每个具体的实现定义它自己的以来关系，保证商务逻辑接口里只包括商务逻辑的方法。

虽然你不应该总是在商务逻辑接口中放置setters来注入依赖关系，但是在商务逻辑接口中放置setters和getters方法作为配置参数是一个很好的想法，这样setter注入就变成有价值的工具了。我们认为配置参数是一种特殊的依赖关系。当然你的组件依赖于配置数据，但是配置数据显然不同于你见过的其他依赖关系。我们马上就会讨论他们的不同，不过现在先思考一下Listing 4-8中的商业逻辑接口。

Listing 4-8. NewsletterSender 接口

```
package com.apress.prospring.ch4;
public interface NewsletterSender {
    public void setSmtpServer(String smtpServer);
   public String getSmtpServer();

   public void setFromAddress(String fromAddress);
   public String getFromAddress();

   public void send();
}
```

NewsletterSender接口是用来定义通过e-mail来发送一组新闻通讯的类。Send()方法是唯一的商务逻辑方法，但是可以看到我们在接口中定义了两个JavaBean属性。我们为什么这样做呢，我们不是刚刚说过不应该在商务逻辑接口中定义以来关系么？原因在于这些值,包括SMTP服务器地址和发送e-mails的客户端的地址，实际上没有依赖关系；相反的，它们的具体配置将会影响到所有的NewsletterSender接口的具体实现的运作。Spring的依赖注入能力为应用程序组件的外部配置提供了完美的解决方案。这里问题在于：配置参数和其它种类的依赖之间的区别在哪里呢？大部分情况下，你可以清晰的看出一个依赖是否应该以配置参数的方式封装，如果你不缺定，看看配置参数下面这三个特征：

1. 配置参数是被动的。在Listing 4-8中的NewsletterSender那个例子中，SMTP服务器的参数是一个被动依赖关系的范例。被动的依赖关系不是直接用来完成一个操作的；取而代之，他们是用来在

内部使用或者由其它的依赖关系来完成实际操作。对比在第二章中的MessageRenderer那个例子中，MessageProvider的依赖不是被动的——它通过必须通过一个需要MessageRenderer的功能来完成这个工作。

2. 配置参数一般是消息，而不是其它组件。我们这里是说配置参数一般是一些消息片断，组件需要它们来完成工作。显然SMTP服务器就是一个NewsletterSender所需要的消息片断，但是，MessageProvider则确实是MessageRenderer正确完成功能所必须的另外一个组件。

3. 配置参数一般是简单的值或者是一组简单值的集合。这实际上是1和2所述问题的一个副产品，但是配置参数的确一般是简单的值。在Java中意味着它们是原生的（或者对应的封装类）或者是String再或是这些值的集合。简单值通常是被动的。这意味着你除了操作它本身的数据外，无法对String做其它的事情；而且你基本上知识拿这些值作为消息使用——例如，某一个int值代表监听的网络接口的端口号，或者某个String代表一个e-mail程序发送信息要访问的SMTP服务器。当决定是否要在商务逻辑接口中定义配置选项的时候，同时需要考虑一下这个配置参是对接口的所有实现都适用还是只对一个有用。例如，在实现NewsletterSender接口的时候，显然所有的实现都需要知道发送e-mails所用的SMTP服务器。然而，我们会选择将是否发送受保护e-mail的标志从商业逻辑接口中剥离，因为并不是所有的e-mail应用程序接口（APIs）都能够识别它，设想有大量的实现都不会考虑安全问题才是正确的。

> 注意：回忆一下在第 2 章中，我们选择在商用需求中定义依赖关系。这些都是为了描述的目的，它们都不是最好的实践。

Setter依赖注入还允许你在运行中将依赖关系替换为一个不同的实现，而不需要重新建立一个父组件的实例。Spring目前还不支持这项功能，但是一旦Spring支持JMX，这个功能就会自我实现了。也许setter依赖注入的最大优点就是它对注入机制的侵入最小。

如果你为一个恰巧只有默认构造器的类定义了构造器依赖注入，那么你将会在非IoC环境下使用依赖于它的所有类的时候遇到麻烦。以IoC为目的在类中定义额外的setters并不会影响到其它类与之交互的能力。

总的来说，基于setter的依赖注入是最佳选择，因为它会使你在非IoC的设置下将其对你代码的影响降到最低。构造器注入，当你需要在依赖关系传递给组件时要确认的情况下，是一个不错的选择，但是要记住，很多容器为这种情况提供了它们自己的基于setter依赖注入的机制。示例程序中的大部分代码都使用了setter依赖注入，虽然那里也有一些例子是基于构造器依赖注入的。

Spring 中的控制反转

我们先前提到过，控制反转是Spring提供的非常重要的一个功能，并且Spring的实现中的核心部分就是基于依赖注入的，同时还提供了依赖查找的功能。Spring提供自动使独立的对象合作的功能，当然这使用了依赖注入实现。在基于Spring的应用程序中，总是偏向于通过依赖注入将合作关系传

递给独立的对象，而不是让独立的对象通过查找来获取合作关系。尽管依赖注入是连接独立对象使之合作的首选方案，你还是需要依赖查找来访问独立的对象。在很多环境中，Spring不能自动的将你的应用程序组件通过依赖注入连接起来，这是你必须通过依赖查找来访问刚刚初始化的一组对象。当你使用Spring MVC支持来构建一个Web应用程序时，Spring可以避免这些问题，将你的整个程序自动的粘合起来。在Spring中只要可以使用依赖注入，那你就应该尽量使用它；否则你只能求助于依赖查找的能力了。你会在本章的教程中看到两种机制的例子，我们在它们出现的地方会指出的。

Spring的IoC容器的一个有趣的功能是，它能够作为其自己的依赖注入容器和外部的依赖查找容器之间的一个适配器。

我们会在第5章中进一步分析。

Spring支持构造器依赖注入和setter依赖注入，支持标准的IoC功能和很多的有用的附加功能，这些能够使你的生活更加便利。

本章的其它部分会介绍Spring的DI容器的基本功能，配以大量的具体示例。

使用 Spring 依赖注入

Spring的依赖注入支持非常全面，已经远远超越了一般意义上的标准IoC功能，你在第5章将会看到更加详细的讨论。本章中的余下部分将会介绍Spring依赖注入容器的基本知识，分析setter和构造器依赖注入，同时会深入的分析Spring中的依赖注入是如何配置的。

Beans 和 Bean 工厂（BeanFactores）

Spring的依赖注入容器的核心是Bean工厂。Bean工厂负责管理组件和它们之间的依赖关系。Spring中，这种bean用来查阅所有受容器管理的组件。典型的情况你的beans会在某种程度上依附于JavaBeans规范，但是这不是必须的，尤其如果你计划使用构造器依赖注入来连接你的beans的时候。

你的应用程序需要通过BeanFactory接口来使用Spring的DI容器。也就是说，你的程序必须创建实现了 BeanFactory 接口的类来配置它的 Bean 和依赖的消息。完成以后，你的程序能够通过 BeanFactory 来访问这些beans，继续处理其它工作。在某些情况下，所有的这些内容的设置都会被自动处理，但是在另一些情况下，你需要自己来完成编码。本章中的所有例子都需要手动设置 BeanFactory 的实现。

虽然BeanFactory都可以通过编程来进行配置，但是更常见的是通过外部的一些配置文件来完成配置。Bean配置在内部是通过实现了BeanDefinition接口的类的实例来表现的。Bean的配置不仅存储着关于bean自己的信息，同时还有其依赖的beans的信息。对于任何同时实现了beanDefinitionRegistry接口的BeanFactory类，你可以从配置文件中读取 Bean 定义（BeanDefinition）数据，既可以通过 PropertiesBeanDefinitionReader（基于.properties文件）也可以通过XMLBeanDefinitionReader（基于XML文件）。这两种主要的BeanFactory实现都与Spring实现的BeanDefinitionRegistry 共存。

如此，你能够在BeanFactory中标记你的beans，每个bean都被命名。每一个bean至少要有一个命名，但是可以拥有多个。第一个命名后的任何命名都被认为是同一个bean的别名。你可以使用bean的命名来从BeanFactory获取它，同时也可以建立依赖关系——例如bean X依赖于bean Y。

BeanFactory 的实现

对于BeanFactory的描述也许是他看起来过渡复杂了，但是应用中，它并不复杂。实际上，我们已经在前面的部分和第2张的简单例子中讨论了的所有的概念。Listing 4-9展示了第2章的代码。

Listing 4-9. 使用 BeanFactory

```
package com.apress.prospring.ch2;
import java.io.FileInputStream;
import java.util.Properties;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;

public class HelloWorldSpringWithDI {
    public static void main(String[] args) throws Exception {
        // get the bean factory
       BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
      mr.render();
   }
    private static BeanFactory getBeanFactory() throws Exception {
       // get the bean factory
       DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        // create a definition reader
       PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
              factory);
        // load the configuration options
       Properties props = new Properties();
       props.load(new FileInputStream("./ch2/src/conf/beans.properties"));
        rdr.registerBeanDefinitions(props);
        return factory;
   }
}
```

在这个例子中，你会发现我们在使用DefaultListableBeanFactory——Spring提供的两个主要的BeanFactory实现中的一个，我们会使用PropertiesBeanDefinitionReader从一个属性(properties)文件中读取Bean定义信息。只要BeanFactory的实现被创建和配置好，我们就可以通过MessageRenderer的命名来获取它，Renderer是通过属性文件来进行配置的。

除了PropertiesBeanDefinitionReader以外，Spring还提供了XmlBeanDefinitionReader，它允许你使用XML文件来代替属性文件配置和管理你的bean。虽然属性文件对于小而简单的程序很理想，但是当你处理大量的beans的时候你会发现这是一个麻烦。因为这个原因，除了对于特别细碎的程序我们应该全部选择XML配置格式。所以这里要详细的讨论一下两个主要BeanFactory中的第二个：XmlBeanFactory。

XmlBeanFactory 派生于DefaultListableBanFactory 并且简单的扩展了它，使它能够通过

XmlBeanDefinitionReader自动获取配置信息。这种方式优于创建如下代码：

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.core.io.FileSystemResource;
public class XmlConfig {
    public static void main(String[] args) {
       DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
      XmlBeanDefinitionReader rdr = new XmlBeanDefinitionReader(factory);
      rdr.loadBeanDefinitions(new FileSystemResource("ch4/src/conf/beans.xml"));
      Oracle oracle = (Oracle)factory.getBean("oracle");
  }
}
```

你可以如此替换这段代码：

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class XmlConfigWithBeanFactory {
    public static void main(String[] args) {
      XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "ch4/src/conf/beans.xml"));
      Oracle oracle = (Oracle)factory.getBean("oracle");
  }
}
```

书中余下部分，包括示例程序，我们将只使用格式XML配置。你可以自己去研究一下properties格式——你会发现在Spring的基础代码中充斥着这样的例子。

当然，你也可以定义自己的BeanFactory实现，然而这样做是很麻烦的；你需要实现很多的接口来达到与BeanFactory相同级别的功能，而不象所提供的BeanFactory那样简单。如果你想定义一个新的配置机制，那么你需要创建一个定义读取器（definition reader），然后把它封装到一个派生自DefaultLstableBeanFactory的简单的BeanFactory实现里。这就是实现XmlBeanFactory所使用的方法；可以查看Spring的代码来了解具体的细节。

配置 Bean 工厂（BeanFactory）

开始建立一个基于Spring的应用程序的关键在于为你的程序创建BeanFactory配置。一个没有任何bean定义的的基本配置文件看起来是这个样子的：

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
</beans>
```

任何bean的定义都是通过在根节点的<beans>标记下通过使用<bean>来声明的。<bean>标记下面有两个属性是必需的：id和class。Id属性用来给这个bean一个默认的命名，class属性用来指定这个bean的类型。Listing 4-10显示了第2章中Hello World示例中的两个bean（renderer和provider）是如何通过配置文件定义的。

Listing 4-10.使用 XML 配置的 Hello World 示例

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
```

```
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="renderer"
        class="com.apress.prospring.ch2.StandardOutMessageRenderer"/>
   <bean id="provider"
        class="com.apress.prospring.ch2.HelloWorldMessageProvider"/>
</beans>
```

我们可以修改第2章的代码来通过 XmlBeanFactory**读取这些配置。**

Listing 4-11.**从** XML **文件读取** Hello World **配置信息**

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import com.apress.prospring.ch2.MessageProvider;
import com.apress.prospring.ch2.MessageRenderer;
public class HelloWorldXml {
    public static void main(String[] args) throws Exception {
        // get the bean factory
       BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
       MessageProvider mp = (MessageProvider) factory.getBean("provider");
        mr.setMessageProvider(mp);
       mr.render();
   }
    private static BeanFactory getBeanFactory() throws Exception {
       // get the bean factory
       XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "ch4/src/conf/beans.xml"));
        return factory;
   }
}
```

注意Listing 4-11**的**main()**方法与第**2**章**Listing 2-10**的**main()**方法之间没有区别，但是这里的**getBeanFactory()**方法的代码量显著减少。此处非常有意思的地方就在于**main()**方法的代码没有任何改变。这是因为它通过**BeanFactory**接口来协同，而不是其它子接口或者类。虽然你也许需要在配置的时候指定它与特定的**BeanFactory**合作，但是你不需要在你的应用程序中的其它地方通过**getBean()**方法来查找**beans**。这是一种应该遵循的很好的模式，你应该避免使你的应用程序与一个特定的BeanFactory **实现间过渡耦合。**

注意在Listing 4-11**的代码中有一个问题，这个问题我们在第**2**张遇到并已经解决了——应用程序依然必须将**provider bean**传递给参考**bean**来满足它们的依赖。在第**2**章，我们修改了它的配置；**Spring**通过**setter**依赖注入来解决这个问题。我们当然也可以通过**XML**配置支持来完成。**

使用 Setter 依赖注入

通过XML**支持来配置**setter**依赖注入，你需要在**<bean>**标记下指定**<property>**标记，将**<property>**标记放置到你需要注入依赖关系的地方。例如，给**provider bean**的**messageProvider**属性指派**renderer bean**，我们可以简单的修改**renderer**的**<bean>**标记下的内容，就像下面这样：**

```
   <bean id="renderer"
       class="com.apress.prospring.ch2.StandardOutMessageRenderer">
```

```
      <property name="messageProvider">
         <ref local="provider"/>
      </property>
   </bean>
```

从这段代码里，我们将provider bean指派给messageProvider属性。我们使用<ref>标记将一个bean的引用分配给一个属性（稍后详细讨论）。现在我们可以移除Hellow World示例中的无用的属性分配，就像Listing 4-12中的样子：

Listing 4-12. 使用 XML 配置依赖注入

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import com.apress.prospring.ch2.MessageRenderer;
public class HelloWorldXmlWithDI {
    public static void main(String[] args) throws Exception {
       // get the bean factory
       BeanFactory factory = getBeanFactory();
       MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
       mr.render();
   }
    private static BeanFactory getBeanFactory() throws Exception {
       // get the bean factory
       XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "ch4/src/conf/beans.xml"));
       return factory;
   }
}
```

这个例子充分利用了Spring的依赖注入能力，完全使用XML格式进行配置。

使用构造器依赖注入

前面的例子中，MessageProvider的实现即HelloWorldMessageProvider为每个getMessage()方法返回相同的手工编码的信息。在Spring配置文件中，你可以轻松的创建一个允许在外部定义消息的可配置的messageProvider，就像Listing 4-13种的样子。

Listing 4-13. 可配置的 MessageProvider 类

```
package com.apress.prospring.ch4;
import com.apress.prospring.ch2.MessageProvider;
public class ConfigurableMessageProvider implements MessageProvider {

    private String message;
    public ConfigurableMessageProvider(String message) {
       this.message = message;
   }
    public String getMessage() {
       return message;
   }
}
```

如你所见，不可能在不提供某个消息的值（除非支持null）的情况下创建一个ConfigurableMessageProvider的实例。这正是我们所需要的，这个类非常适合使用构造器依赖注入。

Listing 4-14展示了如何能够为创建一个ConfigurableMessageProvider的实例重新定义provider bean。

Listing 4-14. 使用构造器依赖注入

```
    <bean id="provider" class="com.apress.prospring.ch4.ConfigurableMessageProvider">
      <constructor-arg>
          <value>This is a configurable message</value>
      </constructor-arg>
  </bean>
```

这段代码中，代替<property>标记，我们使用了<constructor-arg>标签。因为我们这次没有传递另一个bean进去，只是一个String字面值，我们使用了<value>标记代替<ref>指定了构造器依赖注入的值。

当你拥有超过一个的构造器参数或者你的类拥有多个构造器，你需要给每个<constructor-arg>标记一个索引属性在构造器署名指定参数的索引，它起始于0。在你处理一个具有多个参数的构造器的时候你最好使用索引属性来避免在参数之间发生混淆并且确认Spring选用了正确的沟造器。

避免构造器混淆

早某些情况，Spring会难以确定你需要使用哪个构造器进行依赖注入。这一般发生在你拥有两个具有相同数量的参数且参数的类型也完全相同的构造器的时候。考虑一下Listing 4-15中的代码。

Listing 4-15. 构造器混淆

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class ConstructorConfusion {
    private String someValue;
    public ConstructorConfusion(String someValue) {
       System.out.println("ConstructorConfusion(String) called");
       this.someValue = someValue;
   }
    public ConstructorConfusion(int someValue) {
       System.out.println("ConstructorConfusion(int) called");
       this.someValue = "Number: " + Integer.toString(someValue);
   }
    public static void main(String[] args) {
       BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "./ch4/src/conf/beans.xml"));
       ConstructorConfusion cc = (ConstructorConfusion)
                                    factory.getBean("constructorConfusion");
       System.out.println(cc);
   }
    public String toString() {
       return someValue;
   }
}
```

这里，你可以清楚地看到这些代码的功能——它简单的从BeanFactory获取一个ConstructorConfusion类型的bean，然后将值写入到标准输出。现在，看看Listing 4-16中的配置代码。

Listing 4-16. 混淆的沟造器

```
    <bean id="constructorConfusion"
        class="com.apress.prospring.ch4.ConstructorConfusion">
      <constructor-arg>
         <value>90</value>
      </constructor-arg>
   </bean>
```

这里会调哪一个构造器？运行一下次出的代码得到如下的输出：

```
ConstructorConfusion(String) called
90
```

这表明具有String参数的构造器被调用了。这不是所想要的效果，因为我们希望任何具有整数前缀的值都被传递到通过数字注入的沟造器中，即例子中的int构造器。改变这个问题，我们需要对配置文件进行一个小的修改，就像Listing 4-17那样。

Listing 4-17. 克服沟造器混淆

```
    <bean id="constructorConfusion"
        class="com.apress.prospring.ch4.ConstructorConfusion">
      <constructor-arg type="int">
         <value>90</value>
      </constructor-arg>
   </bean>
```

注意，现在<constructor-arg>标记拥有一个附加属性type，它指定了Spring所查找的参数的类型。重新修改配置文件，运行例子中的代码可以得到正确的输出：

```
ConstructorConfusion(int) called
Number: 90
```

注入参数

在前面的两个例子里，你看到了如何使用setter依赖注入和沟造器依赖注入将其他组件和值注射到bean里面。Spring支持非常多的依赖注入参数选项，不仅允许你注入组件和简单的值，还支持包括Java集合类、外部定义的属性文件，甚至其它工厂中的beans。你可以在setter依赖注入和构造器依赖注入中使用所有这些参数类型，通过在<property>和<constructor-args>标记下分别设置的相应标记。

简单值注入

将简单的值注入到beans里面是很简单的。如果需要，只需要简单的在配置标记中指定值，将其封装在一个<value>标记中。默认情况，<value>标记只能读取String值，但是也可以将这些值转换到任何原生数据类型或者原生封装类。Listing 4-18展示了一个暴露了多种属性的简单的bean。

Listing 4-18. 注入简单值

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class InjectSimple {
    private String name;
    private int age;
    private float height;
    private boolean isProgrammer;
```

```
    private Long ageInSeconds;
    public static void main(String[] args) {
       XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "./ch4/src/conf/beans.xml"));
       InjectSimple simple = (InjectSimple)factory.getBean("injectSimple");
       System.out.println(simple);
   }
    public void setAgeInSeconds(Long ageInSeconds) {
       this.ageInSeconds = ageInSeconds;
   }
    public void setIsProgrammer(boolean isProgrammer) {
       this.isProgrammer = isProgrammer;
   }
    public void setAge(int age) {
       this.age = age;
   }
    public void setHeight(float height) {
       this.height = height;
   }
    public void setName(String name) {
       this.name = name;
   }
    public String toString() {
      return    "Name :" + name + "\n"
             + "Age:" + age + "\n"
              + "Age in Seconds: " + ageInSeconds + "\n"
             + "Height: " + height + "\n"
             + "Is Programmer?: " + isProgrammer;
   }
}
```

除了这些属性外，InjectSimple**类还可以定义**main()**方法来创建一个**XmlBeanFactory**然后从**Spring**获取一个**InjectSimple**的**bean**。这个**bean**的属性值被写入到标准输出。这个**bean**的配置在**Listing 4-19**中。**

Listing 4-19. **配置简单值注入**

```
<bean id="injectSimple" class="com.apress.prospring.ch4.InjectSimple">
      <property name="name">
         <value>John Smith</value>
      </property>
      <property name="age">
         <value>35</value>
      </property>
      <property name="height">
         <value>1.78</value>
      </property>
      <property name="isProgrammer">
         <value>true</value>
      </property>
      <property name="ageInSeconds">
            <value>1103760000</value>
      </property>
  </bean>
```

从Listing 4-18**和**4-19**中你可以看到，可以在**bean**里面定义接受**String**值的属性，可以是原生数据类型值或者原生封装类值，使用**<value>**标记把这些值注入到这些属性。下面是运行这个例子所生成的输出，如我们所料：**

```
Name: John Smith
Age: 35
Age in Seconds: 1103760000
Height: 1.78
Is Programmer?: true
```

在第5章，你会看到如何扩展使用<value>能够注入的数据类型。

在同一个工厂注入 Beans

就像你所看到的，允许使用<ref>标记将一个bean注入到另一个bean中。Listing 4-20展示了一个暴露了setter的类，它允许注入一个bean。

Listing 4-20. 注入 Beans

```
package com.apress.prospring.ch4;
public class InjectRef {
   private Oracle oracle;

   public void setOracle(Oracle oracle) {
      this.oracle = oracle;
   }
}
```

要配置Spring将一个bean注入到另一个中，你首先需要配置两个beans：一个是要注入的，另一个是注入的目标。这样做了以后，你可以使用<ref>标记在目标bean里面配置注入。记住，<ref>必须在一个<property>或者<constructor-arg>标记下，这决定于你在使用setter依赖注入还是构造器依赖注入。Listing 4-21种展示了这种配置的例子。

Listing 4-21. 配置 Bean 注入

```
    <bean id="injectRef" class="com.apress.prospring.ch4.InjectRef">
       <property name="oracle">
          <ref local="oracle"/>
       </property>
    </bean>
    <bean id="oracle" class="com.apress.prospring.ch4.BookwormOracle"/>
```

要说明的非常重要的一点是，要注入的类型不一定就是目标中定义的类型；这些类型只需要相容就可以了。相容意味着如果目标类中定义的类型是个接口，那么注入的类型必须实现了这个接口。如果定义了的类型诗歌类，那么注入的类型必须是相同的类或者一个子类。在例子中，InjectRef类型定义了一个setOracle()方法接受一个Oracle的实例，那是一个接口，注入的类型是一个实现了Oracle的类BookwormOracle。这点可能会造成一些开发者的混淆，但是这真的非常简单。注入受控于任何Java代码所需要遵循的类型控制关系，所以当你理解了Java的类型是如何工作的，那么理解注入中的类型控制就非常简单了。

在前面的例子里，要注入的bean的id使用了本地的<ref>标记属性来指定。你后面将会看到，在“理解Bean命名”那一节中，你可以给一个bean多个命名，这样你可以通过很多的别名来引用它。当你使用本地属性，那么意味着<ref>标记仅仅通过bean的id属性查找，而不通过任何的别名查找。通过任何命名来注入bean，使用bean的<ref>标记的属性代替本地属性。Listing 4-22展示了前面例子

的另一种配置方式，通过别名来注入bean。

Listing 4-22. 使用 Bean 别名注入

```
<bean id="injectRef" class="com.apress.prospring.ch4.InjectRef">
   <property name="oracle">
      <ref bean="wiseworm"/>
   </property>
</bean>
<bean id="oracle"
     name="wiseworm"
     class="com.apress.prospring.ch4.BookwormOracle"/>
```

在这个例子中，oracle bean通过name属性的一个别名给出，然后它被通过别名和bean属性的<ref>标记之间的连接注入到injectRef bean。这里不用过渡担心命名的语义学——我们会在本章中详细讨论这个问题。

注入和 BeanFactory 嵌套

到目前为止我们注入的bean与需要被注入的bean都放在同一个bean工厂里。可是，Spring支持Bean工厂的分层结构，如此一个工厂可以被当作另一个工厂的父节点。通过允许Bean工厂的嵌套，Spring允许你将你的配置文件分割为多个不同的文件——对于拥有大量beans的工程是个福音。

当嵌套Bean工厂时，Spring允许其中的beans将子工厂作为父工厂的参考beans。唯一的缺点是这只能在配置中使用。调用子Bean工厂的getBean()来访问父工厂中的一个bean是不可能的。

使用XmlBeanFactory的Bean工厂嵌套非常容易被约束。将一个XmlBeanFactory嵌套到另一个里，只需要简单的将父XmlBeanFactory 作为子XmlBeanFactory 的构造器参数传递过去就可以了。在Listing 4-23展示了这种方法。

Listing 4-23. 嵌套 XmlBeanFactories

```
BeanFactory parent = new XmlBeanFactory(new FileSystemResource(
           "./ch4/src/conf/parent.xml"));
      BeanFactory child = new XmlBeanFactory(new FileSystemResource(
            "./ch4/src/conf/beans.xml"), parent);
```

在子Bean工厂的配置文件中，在父Bean工厂中引用一个bean与子Bean工厂中引用一个bean的工作方式是相同的，除非你在子Bean工厂中有一个与之同名的bean。如果那样，你只要用父工厂替代<ref>标记的bean属性，就可以了。Listing 4-24展示了父Bean工厂的配置文件的样例。

Listing 4-24. 父 Bean 工厂的配置

```
<bean id="injectBean" class="java.lang.String">
     <constructor-arg>
        <value>Bean In Parent</value>
     </constructor-arg>
</bean>
<bean id="injectBeanParent" class="java.lang.String">
     <constructor-arg>
        <value>Bean In Parent</value>
     </constructor-arg>
</bean>
```

如你所见，配置简单定义了两个beans：injectBean和injectBeanParent。两个都是父工厂中数值Bean的String对象。Listing 4-25展示了子Bean工厂的配置样例。

Listing 4-25. 子 Bean 工厂配置

```
<bean id="target1" class="com.apress.prospring.ch4.SimpleTarget">
      <property name="val">
         <ref bean="injectBeanParent"/>
      </property>
  </bean>

  <bean id="target2" class="com.apress.prospring.ch4.SimpleTarget">
      <property name="val">
         <ref bean="injectBean"/>
      </property>
  </bean>

  <bean id="target3" class="com.apress.prospring.ch4.SimpleTarget">
      <property name="val">
         <ref parent="injectBean"/>
      </property>
  </bean>
      <bean id="injectBean" class="java.lang.String">
        <constructor-arg>
           <value>Bean In Child</value>
        </constructor-arg>
  </bean>
```

注意我们在此处定义了四个beans。这个listing中的injectBean与父工厂的injectBean类似，除了那个String的值不同，这表明他是在从子BeanFactory获取的。

Target1这个bean使用了<ref>标记的bean属性来引用另外一个命名为injectBeanParent的bean。因为这个bean只在父Bean工厂中存在，target1获得那个bean的引用。这里有两点很有意义。第一点，你既可以在子Bean工厂也可以在父Bean工厂使用bean属性来引用bean。这使透明的引用bean易于实现，当你的程序膨胀时允许你在配置文件中移动beans。第二点，你不能使用本地属性来引用父工厂中的beans。XML解析器会在同一个文件中检查本地的属性是否存在一个可用的元素（element）对应，阻止引用父工厂中的beans。

Target2这个bean使用了<ref>标记的bean属性来引用injectBean。因为这个bean在两个Bean工厂中都有定义，target2 bean所得到的injectBean的引用来自自己那个BeanFactory。

Target3这个bean使用了<ref>标记的parent属性来直接引用父Bean工厂中的injectBean。因为target3使用了<ref>标记的parent属性，在子Bean工厂中所声名的injectBean被完全忽略了。

Listing 4-26. HierarchicalBeanFactoryUsage（使用分层 Bean 工厂）类

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class HierarchicalBeanFactoryUsage {
    public static void main(String[] args) {
       BeanFactory parent = new XmlBeanFactory(new FileSystemResource(
             "./ch4/src/conf/parent.xml"));
```

```
      BeanFactory child = new XmlBeanFactory(new FileSystemResource(
            "./ch4/src/conf/beans.xml"), parent);
       SimpleTarget target1 = (SimpleTarget) child.getBean("target1");
      SimpleTarget target2 = (SimpleTarget) child.getBean("target2");
      SimpleTarget target3 = (SimpleTarget) child.getBean("target3");
    System.out.println(target1.getVal());
      System.out.println(target2.getVal());
      System.out.println(target3.getVal());
  }
}
```

下面是这个例子运行后的输出：

```
Bean In Parent
Bean In Child
Bean In Parent
```

和我们料想的一样，target1**和**target3**的**bean**都得到了父**Bean**工厂中的**bean**的引用，而**target2 bean**从子**BeanFacory**中得到了**bean**的引用。**

使用集合进行注入

一般，你的bean**需要访问对象的集合，而不是访问一个单一的**bean**或者值。因此，理所当然的，**Spring**允许你在你的一个**bean**中注入一个对象的集合。使用集合很简单：你可以选择**<list>、<map>、<set>**或者**<props>**来描述**List、Map、Set**或者**Properties**的实例，然后就像你在其它注入中所用到的方式一传递这些对象。这个**<props>**标记只允许传递**String**值，因为**Properties**类只允许**String**作为属性的值。当使用**<list>、<map>**或者**<set>**时，你可以使用注入到属性时可以使用的任何标记，甚至可以是其它的集合标记。这样，允许你传递存储**Map**的**List**，存储**Set**的**Map**，甚至是**List**存储**Map、Map**中存储**Set、Set**中存储**List**这样的嵌套！**Listing 4-27**展示了可以注入四种集合类型的类。**

Listing 4-27. **集合注入**

```
package com.apress.prospring.ch4;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class CollectionInjection {
    private Map map;
    private Properties props;
    private Set set;
    private List list;
    public static void main(String[] args) {
       BeanFactory factory = new XmlBeanFactory(
                       new FileSystemResource("./ch4/src/conf/beans.xml"));
        CollectionInjection instance = (CollectionInjection)
        factory.getBean("injectCollection");
       instance.displayInfo();
   }
    public void setList(List list) {
       this.list = list;
```

```
  }
   public void setSet(Set set) {
      this.set = set;
  }
   public void setMap(Map map) {
      this.map = map;
  }
   public void setProps(Properties props) {
      this.props = props;
  }
   public void displayInfo() {
       // display the Map
      Iterator i = map.keySet().iterator();
       System.out.println("Map contents:\n");
      while (i.hasNext()) {
         Object key = i.next();
          System.out.println("Key: " + key + " - Value: " + map.get(key));
      }
       // display the properties
      i = props.keySet().iterator();
      System.out.println("\nProperties contents:\n");
      while (i.hasNext()) {
          String key = i.next().toString();
          System.out.println("Key: " + key + " - Value: "
                + props.getProperty(key));
      }
       // display the set
      i = set.iterator();
      System.out.println("\nSet contents:\n");
      while (i.hasNext()) {
          System.out.println("Value: " + i.next());
      }
       // display the list
      i = list.iterator();
      System.out.println("\nList contents:\n");
      while (i.hasNext()) {
         System.out.println("Value: " + i.next());
      }
  }
}
```

这是很长的一段代码，但是它所作的工作去很少。Main()**方法从**Spring**获得了一个**CollectionInjection**的**bean**，然后调用**displayInfo()**方法。这个方法只是输出从**Spring**注入的**List**、**Map**、**Properties**和**Set**的实例的内容。在**Listing 4-28**中，你可以看到对于**CollectionInjection**类中注入的每一个属性所对应的注入值的配置。**

Listing 4-28. **集合注入的配置**

```
<bean id="injectCollection" class="com.apress.prospring.ch4.CollectionInjection">
      <property name="map">
         <map>
            <entry key="someValue">
               <value>Hello World!</value>
            </entry>
            <entry key="someBean">
               <ref local="oracle"/>
            </entry>
         </map>
      </property>
```

```
      <property name="props">
         <props>
            <prop key="firstName">
               Rob
            </prop>
            <prop key="secondName">
               Harrop
            </prop>
         </props>
      </property>
       <property name="set">
         <set>
            <value>Hello World!</value>
            <ref local="oracle"/>
         </set>
      </property>
      <property name="list">
         <list>
            <value>Hello World!</value>
            <ref local="oracle"/>
         </list>
      </property>
   </bean>
```

在这段代码中，你可以看到，我们给ConstructorInjection类暴露的四个setter注入了值。对于map属性，我们使用<map>标记注入了一个Map的实例。注意，每一个条目都通过<entry>标签来指定，并且每一个都有一个String的键值和一个条目值。条目的值可以是你给单独的属性能够注入的任何类型；这个例子展示了<value>和<ref>标记的使用方法，添加了一个String值和一个bean的引用到Map中。对于props属性，我们使用了<props>标记来建立一个java.util.Properties的实例，通过<prop>标记来扩展它。注意，虽然<prop>标记使用了类似<entry>标记的方式来键入，但是你只能给Properties实例的每个属性指定一个String的值。

<list>和<set>标记工作的方式完全相同：你给每个元素指定值，可以使用在给一个属性注入单独的值的时候所能使用的任何标记，如<value>和<ref>。在Listing 4-28，你可以但到我们给List和Set各添加了一个String值和一个bean引用。

下面是Listing 4-28所产生的输出。和我们意料的一样，它列出了配置文件中生命的那些集合的元素。

```
Map contents:
Key: someValue - Value: Hello World!
Key: someBean - Value: com.apress.prospring.ch4.BookwormOracle@1ccce3c
Properties contents:
Key: secondName - Value: Harrop
Key: firstName - Value: Rob
Set contents:
Value: com.apress.prospring.ch4.BookwormOracle@1ccce3c
Value: Hello World!
List contents:
Value: Hello World!
Value: com.apress.prospring.ch4.BookwormOracle@1ccce3c
```

要记住，对于<list>、<map>和<set>元素，你可以使用任何用来设置非集合属性的标记来指定集合的条目的值。这个理念非常的强大，因此你在注入集合的时候后就不会被限制于使用原生值了，

你也可以将一个集合的bean注入到另一个集合中。

使用这项功能，很容易将你的程序模块化，提供不同的、用户可选择的程序关键逻辑块的实现。考虑一个这样的系统，在线的提供定制的商业逻辑，能够允许集体事务的创建、验证、下订单。在这个系统中，当每个订单已经准备要生产时，它的完成稿需要被发送到合适的打印机去。唯一比较复杂的事，这些打印机收取完成稿的方式不一样，有的通过e-mail，有的通过FTP，还有的通过安全拷贝协议SCP（SecureCopyProtocol）。使用Spring的集合注入，你可以为这个功能创建一个标准接口，就像Listing 4-29展示的样子。

Listing 4-29. ArtworkSender 接口

```
package com.apress.prospring.ch4;
public interface ArtworkSender {
    public void sendArtwork(String artworkPath, Recipient recipient);
    public String getFriendlyName();
    public String getShortName();
}
```

通过这个接口，你可以创建多种实现，它们都具有自我描述的能力，就像Listing 4-30展示的。

Listing 4-30. FtpArtworkSender 类

```
package com.apress.prospring.ch4;
public class FtpArtworkSender implements ArtworkSender {
    public void sendArtwork(String artworkPath, Recipient recipient) {
       // ftp logic here...
   }
    public String getFriendlyName() {
       return "File Transfer Protocol";
   }
    public String getShortName() {
       return "ftp";
   }
}
```

通过上面这样的实现，你只需简单的将一个List传入到ArtworkManager类就可以完成工作了。通过getFriendlyName()方法，你可以显示一个传送方式的列表，在你配置每个公文稿的模版时提供给管理员来选择。还有，如果你面向ArtworkSender接口编写代码，你的程序可以与每个独立的实现保持完全的解耦。

理解 Bean 命名

Spring提供了非常复杂的bean命名结构，它允许你灵活的处理多种情况。每个bean在包含它的Bean工厂中至少要有一个唯一的命名。Spring遵循一个简单的决策过程来决定为每个bean使用哪个命名。如果没有指定id属性，Spring会寻找name属性，如果具有定义，使用在name属性中定义的第一个命名。（我们说第一个命名，是应为在name属性中可以定义多个命名；我们在后面会详细介绍。）如果name和id属性都没有被指定，Spring使用bean的类名作为命名，当然是假如没有其它的bean使用了相同的命名的情况下。Listing 4-31展示了一个配置的例子，里面使用了全部的三种命名方案。

Listing 4-31. Bean 命名

```
<bean id="string1" class="java.lang.String"/>
<bean name="string2" class="java.lang.String"/>
<bean class="java.lang.String"/>
```

这三种方法从技术角度来说是等价的，但是哪种方案才是你的应用程序的最佳选择呢？首先，要避免使用基于类名的自动命名机制。这样就不允许你自由的定义相同类型的多个bean，所以最好自己给他们命名。这样，如果Spring以后修改了默认的命名方式，你的程序依旧可以运行。当选择使用id还是name的时候，一定使用id来指定bean的默认命名。XML的Id属性在Spring配置文件的DTD中被声明为XML唯一标记（XML identity）。这意味着不仅XML解析器可以验证你的文件，任何好的XML编辑器也可以同样进行验证，因此可以减少错误输入bean命名所造成的问题的数量。很重要的，这允许你的XML编辑器检验你在本地属性的<ref>标记中引用的bean是否存在。

这样做的唯一缺点是，使用id属性会限制你只能使用XML元素的ID标记中允许的字符。如果你发现你不能够在你的命名中使用某个字符，你可以通过name属性来指定命名，那样就可以不用受限制于XML命名规则了。如此说，你还是应该考虑给你的bean通过id命名，然后你可以给你的bean通过别名来定义一个描述性的命名，在下一节我们将讨论。

Bean 别名

Spring允许一个bean拥有多个命名。你可以通过在bean的<bean>标记的name属性中，指定逗号分隔或者分号分隔的名称列表来实现这个功能。你可以通过这样代替使用id属性或者与id属性共同使用。Listing 4-32展示了一个简单的<bean>配置，对单一的bean定义了多个命名。

Listing 4-32. 配置多个 bean 命名

```
<bean  id="name1" name="name2,name3,name4" class="java.lang.String"/>
```

如你所见，我们定义了四个命名：一个使用了id属性，其余的三个使用了逗号分隔的名字属性的列表。Listing 4-33展示了一个简单的Java程序，它从Bean工厂中四次获取取了相同一个bean，每次使用了不同的命名，并且验证了获取的是同一个bean。

Listing 4-33. 使用别名访问 Bean

```
package com.apress.prospring.ch4;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class BeanNameAliasing {
   public static void main(String[] args) {
      BeanFactory factory = new XmlBeanFactory(
                        new FileSystemResource("./ch4/src/conf/beans.xml"));

      String s1 = (String)factory.getBean("name1");
      String s2 = (String)factory.getBean("name2");
      String s3 = (String)factory.getBean("name3");
      String s4 = (String)factory.getBean("name4");

      System.out.println((s1 == s2));
```

```
        System.out.println((s2 == s3));
        System.out.println((s3 == s4));
    }
}
```

使用Listing 4-32的配置，这段代码在标准输出中输出了3次true，验证了使用不同命名访问的bean实际上是同一个bean。 你可以通过将bean的任意一个命名传送给Bean工厂的getAliases(String)方法获得bean的别名列表。返回的命名的列表中命名的数量重视少于bean所有命名数量1个，因为Spring认为其中的一个命名是默认的。哪个命名是默认的决定与你如何配置这个bean。如果你通过id属性指定了一个命名，那么它就是默认的。如果你没有使用id属性，那么传送命名属性的列表中的第一个命名被作为默认的。Bean别名是一个奇怪的玩意，因为在你建立一个新的应用程序时一般不会使用。因为如果你将其它的很多bean注入到另一个bean中，这时他们也许凑巧使用了相同的命名来访问。然而，如果你的应用程序已经应用且开始维护，对其进行修改等等，那么bean别名将会变得有用。

考虑如下场景：你拥有一个应用程序，其中有50个不同的bean通过Spring进行配置，它们都需要一个Foo接口的具体实现。其中的二十五个bean使用了StandardFoo实现（bean命名为standardFoo），其余的25个使用了SuperFoo的实现（bean命名为superFoo）。六个月后，你的程序已经付诸应用，你决定将前25个bean转换到使用SuperFoo的实现。这样，你有三种选择。

第一种，将standardFoo的bean的实现类改变为SuperFoo。这样做的缺点是，你拥有两个SuperFoo类的实例存在，而事实上你只需要一个。进一步说，当配置需要改变的时候你需要修改两个bean的配置。

第二种选择是更新需要修改的25个bean的注入配置，将bean的名字从standardFoo修改为superFoo。这样做并不优雅——你要进行查找和替换，但是当需要回滚变动的时候，管理程序并不舒服，需要从你们的版本控制系统重新获取一个配置文件的早前版本。

第三种，也是最理想的方式，删除（或者注释掉）standardFoo bean的定义，然后给superFoo添加一个standardFoo的别名。这样的变动工作量很小，将系统恢复到以前的配置也非常简单。

Bean 实例化模式

默认情况下，Spring中的所有bean都是单例（singletons）。这意味着Spring维护bean的唯一实例，所有的以来对象引用同一个实例，对Bean工厂的getBean()方法的每一次调用都返回同一个实例。我们在前面的例子Listing 4-33中演示了这个情况，那里我们使用了标识对比（==）而不是equals()方法来检查那些bean是否是同一个。

单例这个术语在Java中可替代的表示两种不同的概念：一个对象在应用程序中只有单一的实例，或者但例的设计模式。我们在这里称第一个概念为单例（singleton），而对于模式的概念我们成为单例模式（Singleton）。单例模式在基本的设计模式中算比较普及的：参见Erich Game所写的《设计模式：可复用面向对象软件的基础》（Addison-Wesley，1995）。当人们混淆需要使用单一实例和需要应用单例模式这两个概念的时候，问题显现出来。

Listing 4-34. 单例设计模式

```
package com.apress.prospring.ch4;
public class Singleton {
   private static Singleton instance;

   static {
       instance = new Singleton();
   }

   public static Singleton getInstance() {
      return instance;
   }
}
```

这种模式的目的是允许你在你的应用程序中能够维护和访问一个单一的实例，但是使用它会带来耦合度的上升的危害。你的应用程序代码必须知道单例类的确切信息才可以得到它的实例——完全的将这种能力转移到接口的代码中。实际上，单例模式是两种模式的合体。首先，必须的，模式使其可以维护一个对象的单一实例。其次，必要性少一些，模式将对象查找能力完全移出到接口中。使用单例模式还会使任意替换具体实现的工作变得难以进行，因为大部分对象需要单例的实例来直接访问单例对象。当你试图对你的程序进行单元测试的时候，这会使你头疼不堪，因为你不能将这些单例通过测试中的假冒(mock)对象代替。

幸运的是，通过Spring你可以享受单例实例化样式，而不用遵从单例设计模式的规范。Spring中的所有bean默认都被创建为单例实例，Spring使用同一个实例来完成所有对该bean的请求。当然，Spring并不限制于使用单例实例；它也可以为满足每个依赖和每次调用getBean()而创建该bean的新的实例。实现这些对你的程序的代码没有任何的侵入，因为这个原因，我们喜欢称Spring为创建模式无关的。这是一个非常强有力的概念。如果你一开始将一个对象设计为单例的，但是后来发现对于多线程的访问这样并不合适，那么你可以将它修改为一个非单例的对象却对你的代码没有任何的影响。

> 注意：改变你的 bean 的实例化模式对你的应用程序代码没有影响，但是如果你依赖于 Spring 的生命周期接口，它会造成一些问题。我们在第 5 章中会进一步说明。

改变实例化模式，将单例转换为非单例是很简单的（请看Listing 4-35）。

Listing 4-35. 非单例 Bean 配置

```
<bean id="nonSingleton" class="java.lang.String" singleton="false">
      <constructor-arg>
         <value>Rob Harrop</value>
      </constructor-arg>
  </bean>
```

如你所见，这个bean的声明与前面你看到的其它声明的唯一区别是将单例属性设置为false。Listing 4-36展示了这个设置对你的应用程序的影响。

Listing 4-36. 非单例 Bean 试验

```
package com.apress.prospring.ch4;
```

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class NonSingleton {
    public static void main(String[] args) {
       BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "./ch4/src/conf/beans.xml"));

       String s1 = (String)factory.getBean("nonSingleton");
       String s2 = (String)factory.getBean("nonSingleton");

       System.out.println("Identity Equal?: " + (s1 ==s2));
       System.out.println("Value Equal:? " + s1.equals(s2));
       System.out.println(s1);
       System.out.println(s2);
   }
}
```

运行这个例子会得到如下输出：

```
Identity Equal?: false
Value Equal:? true
Rob Harrop
Rob Harrop
```

你可以从中看到这两个String**对象的值明显是相同的，但是标识却不同，尽管是实际上这两个实例都是使用相同的**bean**命名获取的。**

选择一个实例化模式

在大部分场景，很容易看出哪种实例化模式更适合。有代表性的是我们发现单例是我们的bean**默认的实例化模式。一般而言，单例方式应该在如下的场景下使用：**

- **不分状态的共享对象：当你拥有一个没有状态之分并且有多个依赖对象的对象。因为如果没有状态之分，你就不需要同步，当每次某个依赖对象在工作中需要使用这个**bean**的时候你就不需要创建一个新的实例。**
- **只读状态的共享对象：这与前一点类似，但是你有一些只读状态。这种情况，你依然不需要同步，所以为了满足每个请求而创建新的实例对于这个**bean**来说只不过是增加无用的开销。**
- **使用共享状态的共享对象：如果你的**bean**的状态必须共享，那么单例是最理想的选择。则何种情况，要保证你的状态回写的同步操作的粒度越小越好。**
- **具有可记录状态的高吞吐量对象：如果你有个**bean**在你的程序中被大量使用，那么你也许会发现维护一个单例和其所有**bean**状态回写的同步比持续创建上百个该**bean**的实例能够提供更好的性能。当使用这种方法，一定要努力保持同步的细粒度而不要牺牲一致性。你会发现这种方法在你的程序运行了很长的时间而创建了大量的实例后会显得特别有用，此时你的共享对象只有少量的可回写状态，而这时实例化新的实例将会耗费巨大。**
- **在以下场景你需要考虑使用非单例创建模式：**
- **具有可回写状态的对象：如果你的**bean**拥有一些可回写状态，这时你会发现对它们进行同**

步操作要比为每个请求创建一个新的实例耗费的资源大。

- 具有私有状态的对象：在一些情况，你的某个独立对象需要拥有它私有的状态，这样它们能够与其所依赖的bean单独处理自己的操作。这种情况下，单例显然不合适，你需要使用非单例模式。

从Spring的实例化管理中你可以得到的主要益处是，你的应用程序可以马上从与单例相关的低内存占用率上得到好处，而你并不需要付出太多的工作。这样，如果你发现单例不能够满足你的程序的要求，也可以轻松的修改你的配置，让它们使用非单例模式。

依赖解析

在一般的操作中，Spring能够通过简单查找配置文件解析依赖关系。这样说，Spring能够确保每个bean配置的顺序正确，这样每个bean的依赖关系都可以被正确配置。如果Spring不这样做，只是无序的创建和配置它们，一个bean可能在它依赖的对象初始化前被创建。这显然不是你所要的，它可能在你的应用程序中引发各种各样的问题。

不幸的是，Spring并不知道你的代码中bean间存在的所有依赖关系。例如，有一个bean，叫bean A，它里面得到另一个bean的实例，叫bean B，通过构造器中的getBean()调用。在这种情况下，Spring不知道bean A依赖于bean B，这可能会引起bean A在bean B之前被实例化。你可以通过附加信息通知Spring你的bean依赖其它bean，使用<bean>标记的depends-on属性。Listing 4-37展示了上面bean A和bean B的那种情况要如何配置。

Listing 4-37. 手工定义依赖

```
<bean id="A" class="com.apress.prospring.ch4.BeanA" depends-on="b"/>
<bean id="B" class="com.apress.prospring.ch4.BeanB"/>
```

在这个配置里，我们声明bean A依赖于bean B。Spring在实例化bean的时候会考虑这些内容，保证bean B在bean A前被实例化。

自动装配你的 Bean

上面所有这些例子中，我们都必须通过配置文件清晰的定义每个bean是如何进行装配的。如果你不喜欢自己将你的程序装配起来，你可以尝试让Spring自动装配。默认情况，自动装配是被关闭的。开启它，需要指定你想要使用哪种自动装配方法，给你想要自动装配的bean配置指定装配属性。

Spring支持四种自动装配模式：通过命名、通过类型、构造器或自动监测。当使用通过命名的自动装配，Spring尝试将每个属性连接到一个同名的bean上。如此，如果目标bean拥有一个叫做foo的属性而Bean工厂中定义了一个命名为foo的bean，那么foo这个bean会被分配给目标的foo属性。

当使用通过类型的自动装配，Spring试图将目标bean的每个属性与Bean工厂中对应的同类型的bean连接起来。如果在目标bean中有一个String类型的属性，而在Bean工厂中有String类型的bean，

那么Spring会将String的bean和目标类型的String属性连接起来。如果你在同一个Bean工厂中有多个相同类型的bean，比如String，那么Spring不能确定使用哪个进行自动装配，而后会抛出异常。

构造器自动装配模式运作方式与通过类型的自动装配类似，只是它通过构造器代替setter进行注入。Spring试图最大数量的匹配构造器中包含的参数。比如，如果你的bean拥有两个构造器，一个接受一个String另一个接受一个String和一个整数，而你的Bean工厂中同时有String和整数bean，Spring会使用那个接受两个参数的构造器。

最后以中模式，自动检测，通知Spring在构造器自动装配和通过类型的自动装配间进行选择。如果你的bean拥有一个默认构造器（没有参数），那么Spring会使用通过类型的方式；否则，会使用构造器方式。

Listing 4-38展示了自动装配的简单配置，四个相同类型的bean各自使用不同的模式。

Listing 4-38. 配置自动装配

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="foo" class="com.apress.prospring.ch4.autowiring.Foo"/>
   <bean id="bar" class="com.apress.prospring.ch4.autowiring.Bar"/>

   <bean id="targetByName" autowire="byName"
            class="com.apress.prospring.ch4.autowiring.Target"/>
   <bean id="targetByType" autowire="byType"
            class="com.apress.prospring.ch4.autowiring.Target"/>
   <bean id="targetConstructor" autowire="constructor"
            class="com.apress.prospring.ch4.autowiring.Target"/>
   <bean id="targetAutodetect" autowire="autodetect"
            class="com.apress.prospring.ch4.autowiring.Target"/>
</beans>
```

这种配置你可能看其来感觉非常眼熟。注意每个目标bean都拥有一个不同值的自动装配属性。Listing 4-39展示了一个简单的Java应用程序，从Bean工厂获取每个目标bean。

Listing 4-39. 自动装配合作者

```
package com.apress.prospring.ch4.autowiring;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class Target {
    private Foo foo;
   private Foo foo2;

   private Bar bar;
    public Target() {

   }

   public Target(Foo foo) {
      System.out.println("Target(Foo) called");
```

```
  }
  public Target(Foo foo, Bar bar) {
      System.out.println("Target(Foo, Bar) called");
  }

  public void setFoo(Foo foo) {
      this.foo = foo;
      System.out.println("Property foo set");
  }

  public void setFoo2(Foo foo) {
      this.foo2 = foo;
      System.out.println("Property foo2 set");
  }
   public void setMyBarProperty(Bar bar) {
      this.bar = bar;
      System.out.println("Property myBarProperty set");
  }
   public static void main(String[] args) {
       BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "./ch4/src/conf/autowiring.xml"));
       Target t = null;

       System.out.println("Using byName:\n");
       t = (Target) factory.getBean("targetByName");

       System.out.println("\nUsing byType:\n");
       t = (Target) factory.getBean("targetByType");

       System.out.println("\nUsing constructor:\n");
       t = (Target) factory.getBean("targetConstructor");

       System.out.println("\nUsing autodetect:\n");
       t = (Target) factory.getBean("targetAutodetect");

  }
}
```

在这段代码中，你可以看到Target这个类有三个构造器：一个没有参数的构造器，一个接受Foo实例的构造器，一个接受Foo和Bar实例的构造器。对应这三个构造器，Target Bean有三个属性：两个是Foo类型一个是Bar类型。每个属性和构造器当被调用时都回相标准输出写一段信息。Main方法会从Bean工厂的声明中获取每个目标bean，触发自动装配过程。下面是运行这个例子的输出：

```
Using byName:
   Property foo set
Using byType:
Property foo set
Property foo2 set
Property myBarProperty set
Using constructor:
Target(Foo, Bar) called
Using autodetect:
```

```
Property foo set
Property foo2 set
Property myBarProperty set
```

从输出中，你可以看到Spring什么时候使用通过命名的自动装配，只有一个foo属性使用这种方式，因为只有这个属性与配置文件中的Bean相对应。当使用通过类型的自动装配，Spring对三个属性都进行了装配。Foo和foo2属性连接到foo bean，而myBarProperty连接到bar bean。当使用构造器自动装配，Spring使用了两个参数的构造器，因为Spring能够满足两个参数的bean，这样就不用回退去使用另一个构造器了。这种情况下，自动检测功能与通过类型的方式相同，因为我们定义了一个默认的构造器。如果我们没这么做，自动检测所做的会与构造器自动装配一样。

什么时候使用自动装配

在大部分情况，关于是否应该使用自动装配这个问题的答案当然应该是“不！”，虽然自动装配可以在小规模的程序中节省你的时间，但是它会养成你的坏习惯，在大规模的程序中弹性也不够。使用通过命名的自动装配看起来是个好主意，但是它要求你给属性进行人工命名，才能享受自动装配功能带来的好处。Spring背后的思想是你可以自由的创建你的类，然后由Spring为你工作，而不用你为它们做其它工作。你也许试图使用通过类型的自动装配，直到你意识到这样你只能在Bean工厂给每种类型创建一个bean——当你需要维护同一个类型但具有不同配置的多个bean的时候这个限制是个大问题。当使用构造器自动装配的时候也会发生同样的问题。这些模式遵从相同的语义学，如通过类型的和自动检测的自动装配，后者只不过是通过类型的和通过构造器的自动装配绑定在一起而已。

在一些情况下，自动装配可以节省你的时间，但是对它们进行精确的定义实际上并不会消耗你太多的工作，那样你能够从精确的语义、完全自由的属性命名还有规定要管理同一个类型的多少个实例这些地方受益。除了非常小的应用程序，无论如何也要绕开使用自动装配。

依赖检查

在创建bean实例和装配依赖关系时，Spring默认情况不会检查bean的每个属性是否都有对应的值。很多情况下，你不需要Spring进行检查，但是如果你有一个每个属性都必须有值对应的bean，那么你可以让Spring帮你检查。

如本文指出的，这并不总是有效，因为你可能给某些属性提供了默认值，也许只是声明某些特殊的属性一定要有对应的值；Spring的依赖检查能力并不考虑这种情况。这就是说，在特定的情况下让Spring进行这种检查对你是很有用的。大多数情况下，它允许你将检查从代码中移除，只让Spring在启动时进行一次检查。

除了默认的不检查以外，Spring有三种依赖检查的模式：

简单模式、对象模式、全模式。简单模式检查是否所有的集合类和内建类属性都有值对应。这种模式下，Spring不检查其它类型的属性是否有值对应。这种模式对检查bean的所有配置参数是否有

值对应非常有意义，因为那些参数一般都是内建类值或者内建类集合值。

对象模式检查那些简单模式所不检查的类型的属性，但是它也不检查简单模式检查的那些属性。所以如果你有一个具有两个属性的bean，一个是int类型另一个是Foo类型，那么对象模式检查会检查Foo属性是否被指定了值，但是不会对int属性进行检查。

全模式检查所有的属性，对简单模式和对象模式所检查的属性都进行检查。Listing 4-40展示了一个具有两个属性的简单的类：有一个int属性，还有一个自己同类型的属性。

Listing 4-40. SimpleBean 类

```
package com.apress.prospring.ch4.depcheck;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class SimpleBean {
    private int someInt;
    private SimpleBean nestedSimpleBean;
    public void setSomeInt(int someInt) {
       this.someInt = someInt;
   }
    public void setNestedSimpleBean(SimpleBean nestedSimpleBean) {
       this.nestedSimpleBean = nestedSimpleBean;
   }
    public static void main(String[] args) {
       BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "./ch4/src/conf/depcheck.xml"));

      SimpleBean simpleBean1 = (SimpleBean)factory.getBean("simpleBean1");
       SimpleBean simpleBean2 = (SimpleBean)factory.getBean("simpleBean2");
       SimpleBean simpleBean3 = (SimpleBean)factory.getBean("simpleBean3");
   }
}
```

代码中的main()方法从Bean工厂获取三个bean，都是SimpleBean类型。Listing 4-41展示了Bean工厂的配置。

Listing 4-41. 配置依赖检测

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="simpleBean1" class="com.apress.prospring.ch4.depcheck.SimpleBean"
        dependency-check="simple">
      <property name="someInt">
         <value>16</value>
      </property>
   </bean>

   <bean id="simpleBean2" class="com.apress.prospring.ch4.depcheck.SimpleBean"
        dependency-check="objects">
      <property name="nestedSimpleBean">
         <ref local="nestedSimpleBean"/>
      </property>
   </bean>

   <bean id="simpleBean3" class="com.apress.prospring.ch4.depcheck.SimpleBean"
        dependency-check="all">
```

```
      <property name="someInt">
         <value>16</value>
      </property>
      <property name="nestedSimpleBean">
         <ref local="nestedSimpleBean"/>
      </property>
   </bean>

   <bean id="nestedSimpleBean"
        class="com.apress.prospring.ch4.depcheck.SimpleBean"/>
</beans>
```

如你在配置中看到的，Listing 4-40这个Java程序里，从Bean工厂中获取的每个bean都具有一个不同的dependency-check属性。这里的配置确保通过dependency-check属性配置的需要创建的那些属性都有值对应，这样的结果是这个Java程序可以无误的运行。你可以尝试注释掉一些<property>标记，看看会发生什么——Spring会抛出一个org.springframework.beans.factory.UnsatisfiedDependencyException的异常，其中会指明是哪个属性应该有值对应但却没有。

Bean 继承

某些情况下，你也许需要定义多个相同类型或是实现了共用接口的bean。这时，如果你希望这些bean共享一些配置但又有一些不同的设置，这会是个问题。保持共享配置同步的过程经常出错，在大项目中这样做还非常的耗时。为了解决这个问题，Spring允许你定义一个<bean>从Bean工厂的其它bean处继承属性设置。如果需要，你可以重写需要的子bean中的任何一个属性的值，这使你可以进行完全的控制，父bean可以给你的每个bean提供一个基础的配置。Listing 4-42展示了两个bean的简单配置，其中一个继承自另一个。

Listing 4-42. 配置 Bean 继承

```
<bean id="inheritParent" class="com.apress.prospring.ch4.inheritance.SimpleBean">
      <property name="name">
         <value>Rob Harrop</value>
      </property>
      <property name="age">
         <value>22</value>
      </property>
   </bean>

   <bean id="inheritChild" class="com.apress.prospring.ch4.inheritance.SimpleBean"
        parent="inheritParent">
      <property name="age">
         <value>35</value>
      </property>
   </bean>
```

在这段代码中，你可以看到inheritChild bean的<bean>标记中有个附加的属性parent，它表明Spring应该将inheritParent bean视为这个bean的父bean。因为inheritChild bean的age属性有自己的值定义，Spring将这个值传送给bean。然而，inheritChild没有name属性的值，所以Spring使用了来自

inheritParent bean的值传递给它。Listing 4-43展示了前面配置中出现的SimpleBean类的代码。

Listing 4-43. SimpleBean 类

```
package com.apress.prospring.ch4.inheritance;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class SimpleBean {
    public String name;
    public int age;
    public static void main(String[] args) {
       BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
             "./ch4/src/conf/beans.xml"));

       SimpleBean parent = (SimpleBean)factory.getBean("inheritParent");
       SimpleBean child = (SimpleBean)factory.getBean("inheritChild");

       System.out.println("Parent:\n" + parent);
       System.out.println("Child:\n" + child);
   }
    public void setName(String name) {
       this.name = name;
   }
    public void setAge(int age) {
       this.age = age;
   }

   public String toString() {
       return    "Name: " + name + "\n"
             + "Age:" + age;
   }
}
```

如你所见，SimpleBean的main()方法从Bean工厂获取了inheritChild和inheritParent的bean，然后将它们属性的内容写入到标准输出。下面是例子运行的输出：

```
Parent:
Name: Rob Harrop
Age: 22
Child:
Name: Rob Harrop
Age: 35
```

如我们期望的，inheritChild bean从inheritParent bean的name属性获得了值，但是在age属性还可以提供自己的值。

使用 Bean 继承的思考

子bean可以从父bean继承构造器参数和属性值，所以你可以通过继承使用两种风格的注入。这个层面的灵活性使bean继承成为一个构建应用程序时强有力的工具，可以代替大批的bean定义。如果你要声明一批具有共享的相同属性值的bean时，可以避免通过重复的拷贝和粘贴操作来共享值；取而代之，可以在你的配置中添加继承层次关系。

当你使用继承时，记住，bean继承不需要符合Java的继承层次关系。在五个相同类型的bean间建立继承关系是完全可行的。应该把bean继承看作模版功能，而不是一个实际的继承功能。但是要注意，如果你修改了子bean的类型，那么新的类型必须与父bean兼容。

综述

这一章，总体上讲述了Spring内核和IoC。我们展示了不同类型IoC的很多例子，对在你的应用程序中使用每种机制的优缺点进行了讨论。我们考察了Spring提供哪些IoC机制，还有每种机制何时该在你的程序中使用，何时不该。在探索IoC时，我们介绍了Spring的Bean工厂，它是使Spring具有IoC功能的核心组件，更明确的说，我们专注于介绍XmlBeanFactory，它允许通过XML对Spring进行外部配置。

本章还介绍了Spring的IoC的基础功能，包括setter依赖注入、构造器依赖注入、自动装配和bean继承。在有关配置的讨论中，我们展示了如何通过XmlBeanFactory使用多种不同值配置你的bean属性（包括其它的bean）。

本章只是浅显得介绍了Spring和Spring的IoC容器。在下一章，我们会详细分析Spring中的一些与IoC相关的功能，而后我们会尽一步细致的分析Spring内核提供的其它功能。