Spring Core Tutorial

Revision History

Date	By	Version	Description
03/09/2005		0.1	Initial draft.
04/01/2005		0.2	Settle the content
04/20/2005		0.3	release
11/11/2005		0.4	Add one more section on integration

Table of Contents

1 Spring 简介

自从 J2EE 诞生以来，它已经成为大型企业应用软件的最佳平台。在 2002 年，Rod Johnson 的书 Expert One-on-One J2EE Design and Development 更进一步地改变我们的 J2EE 编程。很多 JAVA 书是后学院式的讲解，但是这本书却是经验和实践的结晶，从而成为一本经典著作。Spring 框架是在这本书中范例的基础上拓展出来的。自从 Spring 框架 1.0 发布以来，它很快地被 J2EE 社区所接受。有人评论说，Spring 框架又把 OO 编程带回到 J2EE 中来了。今天，Spring 框架已经相当成熟，远远超出了 Rod 在他书中所陈述的，而且它还在不断地完善和拓展。

我们将首先陈述大型企业应用编程中的一些常见问题和需求，然后解释 Spring 是如何解决这些问题和满足这些需求的。这样做会比近乎宗教式的说教更具有针对性和实用性。Spring 框架是实践和经验的积累，不是某个天才坐在办公室凭空想象出来的（当然，我们不排除这种可能性，某一天被苹果砸一下后会有超出寻常的想法）。但是我们所见到的 Spring 框架已经是这些积累的抽象了，所以，了结它所解决的问题及上下文会有助于理解和使用 Spring 框架。我们因为经验有限，不可能概括所有的方方面面，所以只讨论我们认为最常见的和最重要的案例。如果读者想进一步探讨，请参阅其它资料或 google，以下资料应该是一个良好的开端：

- Spring 网站：http://www.springframework.org/，这里有 Spring 框架的下载，官方文件，讨论社区等。
- Expert One-on-One J2EE Design and Development，这是 Rod Johnson 在 2002 年下半年出版的书。这本书所讨论的实践经验是非常具有实用价值的。
- Expert One-on-One J2EE Development Without EJB，这是 Rod Johnson 继上一本书的更新。但不是上一本书的的替代。
- Spring in Action，作者是 Craig Walls 和 Ryan Breidenbach
- Spring Live，作者是 Matt Raible，他的 Blog 有很多有用的技巧。
- Pro Spring

另外，Spring 的官方文件已经有中文版了，读者可以参阅。

我们假定读者具有 JDK 和 J2EE 的一般知识。为了让读者可以运行例子中的程序，我们列出了完整的程序并且尽量使它们短小和独立，希望读者动手尝试。

1.1 J2EE 应用软件

企业应用软件不同于桌面上的应用程序，它们的要求差别很大。企业应用软件通常考虑如下的重要方面：

- 应用程序的安全性（security）
- 应用操作的事务（transaction）和记录（audit）
- 服务的不间断性（service continuity）
- 软件的运行环境（runtime environment）和发布过程（deployment）
- 软件的开发成本（development cost）及开发周期（development cycle）
- 软件的维护成本（maintenance）和维修速度（response time）
- 软件的伸缩性（scalability）

J2EE 在很大程度上提供了一个良好的基础设施来可以满足这些需求。例如 JDBC，它很好的概括了数据库的操作，使得在很大程度上简化了数据库操作的编程。在 B2C 的 Web 应用中，因为客户端用 Web 浏览器，所以几乎不需要安装客户程序，也就没有客户程序的更新和维护了。这使得软件更新和维护得到很大的简化。而且，Web 浏览器也差不多形成不是标准的标准了，能出现很大变化的余地也很小了。在 B2B 的应用中，Java 内在的网络功能已经使它很方便，JMS 和 WebServices 更是锦上添花。

1.2 Spring 框架

J2EE 尽管在企业应用中取得了很大成功，但是在许多情况下还是不尽人意，例如它只是提供了最基本的服务，而没有更高级的功能；它的 EJB 容器经常被误用。Spring 框架正是在这种需求中开发出来的，它填补了

很多介于 J2EE 的基本功能和企业应用软件的需求之间的空缺，从而使得 J2EE 编程非常简便。Spring 框架的几个重要特点是：

- Spring 框架的核心是一个具有依赖注入（Dependency Injection）功能的轻型对象容器。它通过依赖注入（Dependency Injection）来设定对象之间的依赖关系。它是无侵入性的，即不需要我们的类实现任何接口，只需我们遵循 JavaBean 的一些规范而已。因为依赖关系的设定是由 Spring 容器来完成的，所以编程得到大量的简化，同时耦合性也大大地降低了。从而也使得测试得到简化，因为在测试中我们可以注入 mock 对象。它也支持 AOP，使得我们可以面向切面编程。
- Spring 框架集成了很多 J2EE 组件，它利用它的轻容器扩展了这些组件，提供了更高级的功能。它不仅集成了很多 J2EE 的最佳实践，而且有的功能更是先驱性的实践，例如，无侵入性的事务，WEB 中的 Action 和 View 的抽象和分离（这使得我们可以在不更改 Action 的情况下替换 View）。它更进一步地集成了很多成熟的工具，例如 Hibernate，iBATIS，JDO 等数据库编程工具，JSP/JSTL，Struts，Velocity，WebWork，Tapestry，Portal，Tiles 等表示层工具。
- 使得我们更便捷地遵循 OOP，面向对象的原则（Object-oriented principles）。相比之下，如果我们的编程环境和实践使得遵循 OOP 很困难或很麻烦，例如 J2EE 核心模式中的一些例子，那么人们很可能就会在实践中放弃 OOP。所以，Spring（还有其它一些类似的工具）不仅仅是一个工具的改进，同时也是对 OOP 的编程环境和实践的极大改进和更新。它更掀起一股轻容器(Lightweight container)的热潮（轻容器是相对于 EJB 等重容器而言的，轻重实际上是指是否易用，是否容易部署，可以快速启动）。
- 同时，Spring 框架本身很好地遵循了 OOP 原则，所以它具有很好的扩展性。

当然，没有什么东西是尽善尽美的，万能的银弹（silver bullet）和圣杯（Holy Grail）是不存在的。在实际应用中，Spring 也存在着一些问题：

- 因为对象之间的依赖是在容器中设定的，当对象较多时，依赖关系不是像写在 Java 程序里那样显而易见和易于跟踪。这多多少少只是一个习惯而已。
- 因为对象之间的依赖关系是用 XML 来描述的，所以设置的正确与否只能用运行来检验，而不能用编译来检验。
- 当我们用 setter()来设置依赖关系时，可能会破坏数据的封装，可能会用错误的对象属性的组合或顺序，可能会使公共界面更复杂。这一点可以通过接口来克服。
- 它的 XML 配置有些烦亢，不过我们可以分成数个文件来进行分类。

但是，这些很少会构成很大的问题。事实上，Spring 框架是在我们用过的所有框架中问题最少的。

Spring 框架的学习难度是和编程经验，以及对 OOP 的认识和理解成反比。所以下面我们简单地介绍一下 OOP，有兴趣的读者请参阅其它书籍和资料，OOP 这 20 年的积累是非常丰富的。我们仅从实用出发，讲解一些要点，为以后的讨论做些准备。

1.3 面向对象的准则

人们编写软件的目的除了使用外，另一个目的是容易修改，否则我们就改制硬件了。所以说，变更是软件的天性。我们不应该限制软件的更新，而应该适应更新，创造环境使得容易更新，甚至预计可能的变化（所以我们才有更多的工作机会）。人们经过长期的经验积累和研究认识到，具有可重用性和可维护性的软件能够很好地适应变化。这里，可重用性是指一个软件模块能否被用在很多不同场合；可维护性是指一个软件模块是否容易修改，更新和扩展。在面向对象的实践中，人们为了写出具有这两个性质的模块，总结了一些原则。下面，我们简略地介绍一下这些原则。值得指出的是，除了第一个开闭原则外，其它原则主要是针对类而言的。现在，人们又总结了一些针对包的原则，但是因为它和我们的内容没有太多的直接关系，所以我们略去不谈。有兴趣的读者可以参阅一下相关的资料。

1.3.1 开闭原则（OCP，The Open-Closed Principle）

开闭原则是说，所有软件模块都应该可以扩展，但不可以修改。遵循这个原则的关键在于抽象化。我们在写一个模块时，不论是一个类，还是一个构件，都应该认真思考它的真正功能，它对其它模块的依赖性，输入和输出，等等。分离出它的可变部分（例如，用接口或外部配置等），对不变部分进行封装。这些不变部分就是这个模块的本质。这里需要说明的是，在对不变部分进行封装时，我们如何定义不变的部分。在数学中，当我们

谈到不变量时，总是要指明它是在什么变化下的不变量。否则是没有意义的，因为在一种变化下的不变量很有可能在另一种变化下就不是不变量了。所以，当我们定义不变的部分时，首先要明确它的变化范围。但是，在软件开发中，很难事先准确的知道这些变化，很多时候是凭经验或行业知识来判断的。所以，这个原则多多少少带有主观性，更像一个总纲而不像一个硬性的法律条文。Martin Fowler 的书 Analysis Patterns 讲解了一些实际经验，有兴趣的读者可以参考。下面这些原则是讲如何安排依赖性使得模块具有良好的封闭性，可重用性和可维护性。

1.3.2 依赖反向原则（DIP，Dependency Inversion Principle）

依赖反向原则是说，要依赖于抽象，而不要依赖于具体。这也就是我们所说的：要针对接口编程，而不要针对实现编程。之所以是倒置，是因为通常在开始依照需求编程时，我们几乎总是依赖于具体的实现。但是，这些具体的实现都不易适应变化，所以要抽象出一些不变的，本质的功能，把可变的留到具体的实现中去。这种抽象的过程是前面过程的反向，例如，当我们需要写出结果时，开始时可能会写到文件里，后来可能会写到网络流里，等等。抽象的结果是写这个功能。针对接口编程是一个不可能过分强调的原则。接口就像高楼大厦中层与层之间，户与户之间的防火墙；大船巨舰中的隔离舱。软件的更新有时就像水火一样难以预料和不可避免（所以我们叫它软件而不是硬件），而接口会适当地屏蔽软件更新所带来的改动扩散（连锁传播）。通常，类的依赖性由这个原则和组合/继承原则主导，而不是由继承主导。

1.3.3 接口分离原则（ISP，Interface Segregation Principle）

接口分离原则是说，不相关的功能应在不同的接口里。不然，在需要一个功能时也不得不同时也依赖于另一个没必要的功能。例如，在早期的 EJB 中，数据库调用和远程调用混在一起，在不需要远程调用时恰好是最糟糕的组合。这个原则说的是，接口的依赖性宽度越窄越好。下面的原则说的是接口的依赖性深度越浅越好。

1.3.4 迪米特法则（Law of Demeter）

Demeter 法则是说尽量不要有传递性，链型的依赖关系。不要和陌生人交谈，只和亲属和朋友交谈。在一个类中，可以调用类里的其它方法，可以调用类属性的方法，可以调用方法中传递变量的方法，可以调用在方法中声明的变量的方法。但不应该调用属性的属性的方法。

1.3.5 组合/继承原则（CRP，Composite Reuse Principle）

组合/继承原则是说，优先使用组合而不是继承。当我们用低级对象来构造高级对象时，应该用组合。一个系统的框架也应该是由组合和接口构成的。下面的原则说明，继承仅仅是帮助实现，而不是帮助构造新的对象的。

1.3.6 李斯可夫代换原则（Liskov Substitution Principle）

Liskov 代换原则是说，任何父类可以出现的地方，子类也可以出现。子类在改变父类属性状态时要遵守父类的行为（即不能有更强的先决条件，只能有更弱的先决条件）。这就意味着无法用继承来构造行为差异很大的对象。

1.3.7 参考资料

读者可以参考以下资料:
http://www.objectmentor.com
http://c2.com/cgi/wiki?LawOfDemeter

2 Spring 核心包(Core)和 Context 包

在这一章里，我们将讨论 Spring 框架中的轻型容器。它由两个最基础的包组成，Spring 核心包和 Context 包。在这两个包里有三个最重要的接口，BeanWrapper，BeanFactory，和 ApplicationContext。我们将通过探讨这三个接口来讲解这两个包的目的，用法，功能以及它们是如何实现这些功能的。只有明白了它们所解决的问题和它们所用的方法，才能有效地使用它们。实际上，这三个接口和它们的实现及附属类构成了三个层次，它们各自解决不同而相关的问题。

核心包是整个 Spring 框架的基础，它同时又是一个独立的组件，可以单独地使用。Spring 的发布包里的 dist（是英文 distribution，发布的缩写）目录下有一个文件 spring-core.jar，它大约有 240K。这个 jar 文件可以单独使用，它不依赖其它的 Spring 包。Apache commons-logging.jar 是它唯一依赖的第三方的包。

核心包最早是用来解决程序配置中的一些问题的，例如，程序配置方式的不统一，配置中单例模式的无限制的调用，组件配置时接口和实现的对应，和其它相关问题。

在 Java 中，我们可以用 Properties 文件，XML 文件，JVM 初始变量，JNDI 查询，甚至是 ANT/Maven 的属性，或其它格式及来配置应用程序中的某些属性。这些配置方式五花八门，眼花缭乱，但又各为所需。如果这些各式各样的配置方式越多，越复杂，那么程序维护的代价就越高。如果能有一种统一的方式来配置这些属性，同时又不限制使用哪种实现，那么开发和维护程序就会容易很多，代价也会降低。Spring 提供了这种机制。如果用前面 OOP 来说，我们仅仅是在做配置属性这一件事，尽管它有许多不同的实现方法。

核心包同时也是一个对象容器，用依赖关系的注入来设置对象的属性。在 OO 编程中， 一个很重要的准则就是调用接口来编程，而不是调用实现来编程。如果我们不遵循这个原则，那么程序就会有很大的粘合性，也就是说改动一个地方很有可能会波及到很多地方。例如，一个被调用的程序和调用者之中任何一个的改动都会很容易地波及到另一个。单例模式违背了这一准则，因为它的实现和接口是在一起。而且它把这种粘合推到了极端，因为单例可以在任何地方引用，它没有范围的限制，从而可以使得任何类都依赖于它。Spring 利用对象容器来管理单例对象的生成和多例对象的复制及生成。这个容器分离了接口和实现，使得我们可以依不同环境而使用不同的实现机制。因为接口还是同一个接口，所以调用这个接口的程序几乎无需改动。

Spring 对象容器还有其它一些优点， 例如我们可以用接口来声明物体的类属性，而用 Spring 容器来声明类属性的实现。所以，Spring 容器使得物体之间的调用自然地符合调用接口来编程的准则。它的另一个很大的好处是无侵入性，换句话说，我们不需要实现任何 Spring 接口或继承任何 Spring 父类，只需要遵循 Java bean 的规则就可以了。相比之下，EJB 则要求我们继承 EJB 父类。无侵入性意味着良好的可移植性，例如，我们依然可以用程序来做同样的事，自己来管理对象的生命周期。

Context 包是核心包的补充和扩展，它增加了一些应用程序的运行环境的配置功能，例如资源的配置功能和对象定义的覆盖。同样，它也是一个相对独立的组件，它的文件是 spring-context.jar，仅依赖于前面提到的 spring-core.jar（和 commons-logging.jar）。但要注意的是，AOP 和 CGLIB 的包可能会间接的引用。

2.1 BeanWrapper 类及其应用

BeanWrapper 接口和它的实现 BeanWrapperImpl 是整个 Spring 框架最基本的类。它们利用反射机制（Reflection）和 JavaBeans 的标准提供一些对 Java 类属性的操作。其中最重要的操作是数据类型的转换和嵌套属性的引用（这使人联想起 BeanUtils, OGNL 和 JSTL EL）。它们对 JavaBeans 标准的要求对调用者来说是很容易满足的，而且它不需要调用者实现任何接口或者继承任何父类。所以，它几乎可以用在任何需要的场合。另一方面，尽管它的要求很低，但是它的功能却非常强，容易扩展。Spring 更是尽量减少 JavaBeans 的限制，例如，它加入了 constructor 注入的功能，只要求 getter/setter 对应 property 属性即可，而不要求它们对应类的属性。

2.1.1 简单例子

我们先建一个简单的 JavaBean，Person。它有四个类属性，每个属性的类型都不同，而且都有 getter 和 setter 方法。为了测试打印方便，我们还重写（覆盖）了 toString()方法。我们将在这一章中反复使用这个类作为例子。

```
package springsamples.springcore;

public class Person
{
    protected int age = 0;
    protected String name = null;
    protected Person spouse = null;
    protected String[] belongings = null;

    // getter and setter
    public int getAge() { return age; }
    public void setAge(int a) { age = a; }

    public String getName() { return name; }
    public void setName(String n) { name = n; }

    public Person getSpouse() { return spouse; }
    public void setSpouse(Person s) { spouse = s; }

    public void setBelongings(String p, int i) { belongings[i] = p; }
    public void setBelongings(String[] p) { belongings = p; }
    public String getBelongings(int i) { return belongings[i]; }
    public String[] getBelongings() { return belongings; }

    // just want to see the values
    public String toString()
    {
        StringBuffer ret = new StringBuffer();
        ret.append("name = ").append(name).append(" | age = ").append(age);
        if (spouse != null)
        {
            ret.append(" | spouse = ").append(spouse.getName()).
                                    append(" & ").append(spouse.getAge());
            ret.append(" & its sponse=" + spouse.getSpouse().getName());
        }
        if (belongings != null)
        {
            ret.append(" | belongings = ");
            for (int i = 0, j = belongings.length; i < j; i++)
                ret.append(belongings[i]).append(" & ");
        }
        return ret.toString();
    }
}
```

下面我们看看怎样使用 BeanWrapper 和 BeanWrapperImpl。首先我们要引入 Spring 包，有两种办法引入 Spring 的包，一种方法是直接使用 Spring.jar，这是一种简洁的办法，不用考虑各个组件包；另一种是用同一个目录下的组件包。当然，无论那种方法，我们都必须加入所依赖的第三方的包。对于 BeanWrapper，我们只需要 spring-core.jar 和 commons-logging.jar（在以后的发布中，这些包可能会有变化，我们现在用版本 1.1.5）。commons-logging.jar 是一个很小的日志包代理，它或者使用 log4j，或者使用 JDK 的日志包。

```
package springsamples.springcore.beanwrapper.simple;

import org.springframework.beans.BeanWrapper;
import org.springframework.beans.BeanWrapperImpl;
import springsamples.springcore.Person;

public class TestBeanWrapperWithPerson
{
    public static void main(String[] args) throws java.beans.PropertyVetoException
```

```
    {
        Person person = new Person();
        BeanWrapper bw = new BeanWrapperImpl(person);

        // 1. set simple properties, like int or String
        // set the integer field with a string value, it should be converted automatically.
        bw.setPropertyValue("age", "25");
        bw.setPropertyValue("name", "John Smith");
        System.out.println("person: " + person);

        // 2. set a bean object
        bw.setPropertyValue("spouse", new Person());
        bw.setPropertyValue("spouse.name", "Mary Smith");
        // another way to set integer
        bw.setPropertyValue("spouse.age", new Integer(23));
        bw.setPropertyValue("spouse.spouse", person);
        System.out.println("spouse: " + person.getSpouse());
        System.out.println("self: " + person);

        // 3. set arrays
        bw.setPropertyValue("belongings", "house, car, boat");
        System.out.println("bean properties: " + person);
    }
}
```

在第一步中，类属性 name 的设定很简单，把字符串“John Smith”赋给字符串属性 name。类属性 age 的设定是把字符串“25”赋给了 int 型的属性 age，这个类型的转换是由 BeanWrapperImpl 通过 JavaBeans 的 PropertyEditor 来完成的。BeanWrapperImpl 类自动加载一些 PropertyEditor。在第二步中，BeanWrapper 甚至可以设定我们的 Person 类以及它的属性。这里，我们用到了嵌套属性 spouse.name 和 spouse.age 等。而且还可以把配偶的配偶设置成自己。在第三步中，我们用一个逗号分开的字符串“house, car, boat”设置了一个字符串型的数组 belongings。读者可以从打印结果来验证这些功能。如果要想看 Spring 的信息，读者可以把 JDK 的 Logging 或 Log4J 设置设为 DEBUG。

2.1.2 新的类型转换

从这个简单的例子不难看出 BeanWrapper 的数据类型的转换和嵌套属性的引用是很方便的。现在我们举例来说明如何加入新的类型转换。在上面的例子中，当我们设置配偶时，我们设置了四个值。现在，我们将通过一次赋值来做同样的事，为此先加入一个新的类型转换

```
package springsamples.springcore.beanwrapper.propertyeditors;

import java.beans.PropertyEditorSupport;
import java.util.*;
import springsamples.springcore.Person;

public class SpousePropertyEditor extends PropertyEditorSupport
{
    public void setAsText(String text)
    {
        Person p = new Person();
        StringTokenizer st = new StringTokenizer(text, "|");
        p.setName(st.nextToken());
        p.setAge(Integer.parseInt(st.nextToken()));
        setValue(p);
    }
}
```

这是一个标准的 PropertyEditor，仅依赖于 Java 的 API 和 Person 类。它的输入是一个有“|”字符串，输出是一个 Person 对象，输入字符串“|”前的部分作为 name，“|”后的部分作为 age。下面是如何使用这个扩展的例子。

```
package springsamples.springcore.beanwrapper.propertyeditors;

import java.beans.PropertyEditorManager;
```

```
import org.springframework.beans.BeanWrapper;
import org.springframework.beans.BeanWrapperImpl;
import springsamples.springcore.Person;

class TestSpousePropertyEditorWithBeanWrapper
{
    public static void main(String[] args) throws java.beans.PropertyVetoException
    {
        Person person = new Person();
        BeanWrapper bw = new BeanWrapperImpl(person);
        // There are two ways to register a new PropertyEditor, both works:
        // a. This is the standard java bean way.
        PropertyEditorManager.registerEditor(Person.class, SpousePropertyEditor.class);
        // b. This is the special spring bean wrapper way.
        //bw.registerCustomEditor(Person.class, new SpousePropertyEditor());

        // after register it, we can do these
        bw.setPropertyValue("name", "John Smith");
        bw.setPropertyValue("age", "25");

        // 2. set a bean object with the new PropertyEditor
        bw.setPropertyValue("spouse", "Mary Smith|23");
        bw.setPropertyValue("spouse.spouse", person);
        System.out.println("self: " + person);
        System.out.println("spouse: " + person.getSpouse());
    }
}
```

有两种方式来设定这个新的 PropertyEditor，一种是用 JavaBeans 标准的 PropertyEditorManager，一种是用 BeanWrapper 的 API（读者可以都试一试）。设定这个新的 PropertyEditor 以后，我们就可以用“Mary Smith|23”来设 spouse 的属性了。读者有兴趣的话，可以试一试如何把一个字符串转换成 BigDecimal，把一个字符串组转换成 int 型的数组，或者把上面的 Person 类扩充成 Parent 类，加入一个 Person 型的数组 children 作为属性，再试着如何加一个 PropertyEditor 来设置这个 children 属性。

BeanWrapper 和它的实现是一个极为优秀的思路，因为它不仅解决了数据类型的转换和（嵌套）属性的引用问题，而且彻底地消除了这二者的依赖性（更准确的说，是把依赖性留在了 JDK 中），使得它不需改动 BeanWrapper 实现就可以扩展。在 BeanWrapper 出现之前，也有过类似的实现，但它们都很难扩展，界面不标准且难用。最重要的扩展就是新的类型转换。BeanWrapper 由于是基于 JavaBeans 标准的，所以它克服了这些弱点，并且扩展接口也很简单，标准。这个想法非常符合我们前面讨论过的 OOP。

当然，这个优秀的思路也不是一帆风顺的，这一点可以从它的源程序看出来。BeanWrapperImpl 类居然有 42K 之大，犹如一个庞然大物。人人都知道这是不对的，但即使是 Spring 团队的人也不敢轻易改动它。这仅仅是一个 4，5 年旧的程序，要是一个 20 年旧的，那就得要读者发挥一下想象力了。Unix 上的 SCC 就是这样的一个巨兽，任何人拿它没办法。所以，我们要吸取教训，在编程时一定要考虑以后的维护，不是在说 SCC 不好用，恰恰相反，它很好用而且现在仍在用，但是几乎没有人可以维护它。

2.2 BeanFactory 接口和它的实现类

BeanWrapper 虽然功能强大，但是它仅仅是解决了整个问题的一半，即如何设置属性，而没有考虑如何得到这些属性所需的值。这些值可能是外部常量（显然，我们不愿意把常量写死在 Java 程序中，不然的话，改起来是很麻烦的），可能是从别的对象的属性变化而来的，例如，在设置数据源时，有时我们要设置 JDBC 的参数，有时我们要设置 DataSource，有时我们要设置 JNDI 查询。再比如，网络 Proxy 设置，输出格式设置等。以往的解决方法有着几乎和设置属性时谈到的一样的问题：

- 数据类型转换不足或者不容易增加新的数据类型转换
- 大量使用单例模式，从而使得程序太紧凑，不易扩展
- 对于类似的问题，没有一个统一的处理界面

BeanFacoty 正是在 BeanWrapper 的基础上扩展，来解决整个问题的另一半。

2.2.1 BeanFactory 的实现类

既然现在我们明白了它的目的，让我们来看看 Spring 是如何解决这个问题的。BeanFactory 有几个实现类，我们将从基本的开始。

2.2.1.1 由 Properties 定义的 BeanFactory

我们依旧使用前面的例子，假定有如下的设置

```
simplebean.class=springsamples.springcore.Person
simplebean.name=John Smith
simplebean.age=25
```

我们想把“John Smith”和“25”分别设置到一个 Person 类的 name 属性和 age 属性中去。这里的 simplebean 仅仅是个标签，是为了区别其它的类似设置。假定我们把以上内容存在文件 simplebean.properties 中，下面的程序显示如何得到这些设置

```
package springsamples.springcore.beanfactory.listablebeanfactory.simple;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;
import springsamples.springcore.Person;

public class TestListableBeanFactory
{
    public static void main(String[] args)
    {
        DefaultListableBeanFactory dlbf = new DefaultListableBeanFactory();
        FileSystemResource fsr = new FileSystemResource("config/simplebean.properties");
        PropertiesBeanDefinitionReader pbdr = new PropertiesBeanDefinitionReader(dlbf);
        pbdr.loadBeanDefinitions(fsr);
        BeanFactory bf = (BeanFactory) dlbf;

        Person person = (Person)bf.getBean("simplebean");
        System.out.println("person=" + person);
    }
}
```

我们用 DefaultListableBeanFactory 来创建一个 PropertiesBeanDefinitionReader 对象，用这个对象来读取 simplebean.properties 中的内容，然后调用 DefaultListableBeanFactory 来得到我们所想要的 Person 对象。我们是从 BeanFactory 这个接口得到 Person 对象的，这是调用者所依赖的。在这个例子中，我们用的是 DefaultListableBeanFactory 这个 BeanFactory 的实现，这是因为我们的设置是.properties 的格式。在以后的例子中，我们会使用另一个 BeanFactory 的实现，XmlFactory。PropertiesBeanDefinitionReader 实现的接口 BeanDefinitionReader 是为了隔离不同定义格式的差别，例如 properties 或 XML。它也可以同时载入几个文件。

2.2.1.2 在 BeanFactory 中对象间的引用

现在，我们增加一些复杂性，把 simplebean.properties 的内容改为如下并保存在一个新的文件 beanref.properties 中

```
simplebean1.class=springsamples.springcore.Person
simplebean1.name=John Smith
simplebean1.age=25
simplebean1.spouse(ref)=simplebean2

simplebean2.class=springsamples.springcore.Person
simplebean2.name=Mary Smith
```

```
simplebean2.age=23
simplebean2.spouse(ref)=simplebean1
```

这里我们增加了一个新的 Bean 的设置，并且加入了二者之间的引用，即 simplebean1 的 spouse 值将是 simplebean2，同样地，simplebean2 的 spouse 的值将是 simplebean1。相应的程序改动如下（加粗部分）

```
package springsamples.springcore.beanfactory.listablebeanfactory.simple;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;
import springsamples.springcore.Person;

public class TestListableBeanFactoryWithBeanRef
{
    public static void main(String[] args)
    {
        DefaultListableBeanFactory dlbf = new DefaultListableBeanFactory();
        FileSystemResource fsr = new FileSystemResource("config/beanref.properties");
        PropertiesBeanDefinitionReader pbdr = new PropertiesBeanDefinitionReader(dlbf);
        pbdr.loadBeanDefinitions(fsr);
        BeanFactory bf = (BeanFactory) dlbf;

        Person person = (Person)bf.getBean("simplebean1");
        System.out.println("person=" + person);
    }
}
```

读者可以从打印结果来证实这一点。在这里，要注意的是，是我们的 BeanFactory，而不是 main()直接地设置这两个 Bean 的依赖关系，在后面我们还要再从另一个角度来讨论这一点。

2.2.1.3 新的类型转换

现在我们仍继续讨论如何取得所需的值。如果我们想加入一个 PropertyEditor，就像在 BeanWrapper 中一样可以设置 spouse 属性，Spring 提供了两种办法

```
package springsamples.springcore.beanfactory.listablebeanfactory.propertyeditor;

import java.beans.PropertyEditorManager;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;
import springsamples.springcore.Person;
import springsamples.springcore.beanwrapper.propertyeditors.SpousePropertyEditor;

public class TestListableBeanFactoryWithPropertyEditor
{
    public static void main(String[] args)
    {
        DefaultListableBeanFactory dlbf = new DefaultListableBeanFactory();
        FileSystemResource fsr = new FileSystemResource("config/beanwithpropertyeditor.properties");
        PropertiesBeanDefinitionReader pbdr = new PropertiesBeanDefinitionReader(dlbf);
        pbdr.loadBeanDefinitions(fsr);
        // both lines below work!
        //PropertyEditorManager.registerEditor(Person.class, SpousePropertyEditor.class);
        dlbf.registerCustomEditor(Person.class, new SpousePropertyEditor());
        BeanFactory bf = (BeanFactory) dlbf;

        Person person = (Person)bf.getBean("simplebean");
        System.out.println("person=" + person);
    }
}
```

这里我们可以用标准 JavaBeans 中的方法或者 DefaultListableBeanFactory（或更确切的讲，是 ConfigurableBeanFactory）中的方法。这样，我们的设置可以改写为

```
simplebean.class=springsamples.springcore.Person
simplebean.name=John Smith
simplebean.age=25
simplebean.spouse=Mary Smith|23
```

2.2.1.4 用 XML 来定义对象

BeanFactory 可以看成是管理设置的 BeanWrapper。下面我们来看一下如何使用 BeanFactory 的另外一种比 properties 格式更常用的格式，XML 格式。Spring 提供的 XML 格式比 properties 格式功能更多，这个格式的定义在 Spring 发布包 dist 目录下 spring-beans.dtd 中。这是前面例子的 XML 格式的设置

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

<beans>
    <bean id="simplebeanfromxml" class="springsamples.springcore.Person">
        <property name="name"><value>John Smith</value></property>
        <property name="age"><value>25</value></property>
        <property name="spouse"><ref local="spousebean"/></property>
        <property name="belongings"><value>cloth,car,cat</value></property>
    </bean>

    <bean id="spousebean" class="springsamples.springcore.Person">
        <property name="name"><value>Mary Smith</value></property>
        <property name="age"><value>23</value></property>
        <property name="spouse"><ref local="simplebeanfromxml"/></property>
    </bean>
</beans>
```

这里，每个对象都包在<bean>标签中，每个对象的属性都由<property>标签来定义，或者是常量，或者是其它对象的引用。每个对象是由 id 属性来识别，由 class 属性来定义它的类。调用的 Java 程序稍稍有一些变化

```
package springsamples.springcore.beanfactory.xmlbeanfactory.simple;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import springsamples.springcore.Person;

public class TestXmlBeanFactory
{
    public static void main(String[] args)
    {
        // This is one way to do this
        //BeanFactory bf = new XmlBeanFactory(new FileSystemResource("config/simplebean.xml"));
        // Here is another way
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions(new FileSystemResource("config/simplebean.xml"));
        BeanFactory bf = (BeanFactory) factory;

        Person person = (Person)bf.getBean("simplebeanfromxml");
        System.out.println("person=" + person);
    }
}
```

我们提供了两种用法，在第二种用法中，它和前面 properties 格式的用法只差别于读入不同格式的设置。

2.2.1.5 用 CustomEditorConfigurer 来定义数据类型转换

下面我们来看看怎样加入新的数据类型转换。为了避免厌烦，我们试一个新的例子。先引入一个新类 Parent，并且加入一个新的类属性 children

```
package springsamples.springcore;

public class Parent extends Person
{
    // an array of objects of non-trivial type
    protected Person[] children;

    public void setChildren(Person p, int i) { children[i] = p; }
    public void setChildren(Person[] p) { children = p; }
    public Person getChildren(int i) { return children[i]; }
    public Person[] geCchildren() { return children; }

    public String toString()
    {
        StringBuffer ret = new StringBuffer(super.toString());
        if (children != null)
        {
            ret.append(" | children = ");
            for (int i = 0, j = children.length; i < j; i++)
            {
                ret.append(children[i].getName()).append(" | ");
                ret.append(children[i].getAge()).append(" & ");
            }
        }
        return ret.toString();
    }
}
```

类似于 Person，我们加入 getters 和 setters，以及重写 toString()来输出这个新的类属性。有了这个新的类，我们需要一个新的数据类型转换来输入 children 的值。

```
package springsamples.springcore.beanwrapper.arraypropertyeditor;

import java.beans.PropertyEditorSupport;
import java.util.*;
import springsamples.springcore.Person;

public class ParentPropertyEditor extends PropertyEditorSupport
{
    // input format: george|5,Kate|3
    // output array of person set to the field.
    public void setAsText(String text)
    {
        List list = new ArrayList();
        StringTokenizer st = new StringTokenizer(text, ",");
        while (st.hasMoreTokens())
        {
            String a = st.nextToken();
            StringTokenizer st1 = new StringTokenizer(a, "|");
            while (st1.hasMoreTokens())
            {
                Person p = new Person();
                p.setName(st1.nextToken().trim());
                if (st1.hasMoreTokens()) p.setAge(Integer.parseInt(st1.nextToken().trim()));
                list.add(p);
            }
        }
        Person[] people = new Person[list.size()];
        System.arraycopy(list.toArray(), 0, people, 0, list.size());
        setValue(people);
    }
```

```
}
```

这里，我们把一个字符串数组转换成一个 Person[]型的数组，而每个字符串和以前 Person 的字符串格式是一样的。有了这个转换，我们就可以这样来描述

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

<beans>
    <bean id="simpleParent" class="springsamples.springcore.Parent">
        <property name="name"><value>John Smith</value></property>
        <property name="age"><value>40</value></property>
        <property name="spouse"><ref local="spousebean"/></property>
        <property name="belongings"><value>cloth,car,cat</value></property>
        <property name="children"><value>Joe Smith|5,Kathy Smith|2</value></property>
    </bean>

    <bean id="spousebean" class="springsamples.springcore.Parent">
        <property name="name"><value>Mary Smith</value></property>
        <property name="age"><value>38</value></property>
        <property name="spouse"><ref local="simpleParent"/></property>
    </bean>
</beans>
```

调用的 Java 程序所需的改变和以前几乎一样

```
package springsamples.springcore.beanfactory.xmlbeanfactory.editors;

import java.beans.PropertyEditorManager;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import springsamples.springcore.*;
import springsamples.springcore.beanwrapper.arraypropertyeditor.ParentPropertyEditor;

public class TestXmlBeanFactoryPropertyEditor
{
    public static void main(String[] args)
    {
        ConfigurableBeanFactory bf = new
                    XmlBeanFactory(new FileSystemResource("config/parentbean.xml"));

        // 1. register globally
        // PropertyEditorManager.registerEditor(Person[].class, ParentPropertyEditor.class);
        // 2. regiser through factory
        bf.registerCustomEditor(Person[].class, new ParentPropertyEditor());

        Parent parent = (Parent)bf.getBean("simpleParent");
        System.out.println("person name=" + parent);
    }
}
```

考虑到我们几乎肯定会加入新的数据类型转换，Spring 提供了一个很方便的手段。在上面的 XML 中加入一个如下新的 bean 并且把整个内容存到一个新的文件 parentbeanwitheditor.xml 中

```
<bean id="customEditorConfigurer"
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
<property name="customEditors">
        <map>
        <entry key="[Lspringsamples.springcore.Person;">
            <bean
             class="springsamples.springcore.beanwrapper.arraypropertyeditor.ParentPropertyEditor"/>
        </entry>
        </map>
    </property>
```

```
</bean>
```

相应的调用程序改为如下

```
package springsamples.springcore.beanfactory.xmlbeanfactory.editors;

import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.beans.factory.config.*;
import springsamples.springcore.*;

public class TestXmlBeanFactoryPropertyEditorInXml
{
    public static void main(String[] args)
    {
        ConfigurableListableBeanFactory  bf = new XmlBeanFactory(new
                                FileSystemResource("config/parentbeanwitheditor.xml"));

        // ApplicationContext would register that automatically. But BeanFactory
        // doesn't, so we have to manually register this.
        BeanFactoryPostProcessor cec =
                                (BeanFactoryPostProcessor)bf.getBean("customEditorConfigurer");
        cec.postProcessBeanFactory(bf);

        Parent parent = (Parent)bf.getBean("simpleParent");
        System.out.println("person name=" + parent);
    }
}
```

这样做的好处是当我们想再增加 PropertyEditor 时，我们不须改动 Java 调用程序，而只需在 customEditorConfigurer 这个 bean 中改动即可。它的另一个好处我们将会在以后提到，简单地讲，就是 ApplicationContext 会自动地设置这个 customEditorConfigurer，因为这些本身就是对象容器的功能。

2.2.1.6 外部配置文件

很多时候，我们需要在设置中保存诸如用户名和密码（这些随环境而变动，或随时间而变动，或敏感的数据都不应该嵌在程序中），或其它数据，但是为了安全又想让尽可能少的人知道。BeanFactory 提供了这样的机制，使得我们的设置可以引用外部文件。我们依然沿用上面例子，假如说年龄是隐私，我们想把它存在别的文件里。

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

<beans>
    <bean id="simplebeanfromxml" class="springsamples.springcore.Person">
        <property name="name"><value>John Smith</value></property>
        <property name="age"><value>${john.age}</value></property>
        <property name="spouse"><ref local="spousebean"/></property>
        <property name="belongings"><value>cloth,car,cat</value></property>
    </bean>

    <bean id="spousebean" class="springsamples.springcore.Person">
        <property name="name"><value>Mary Smith</value></property>
        <property name="age"><value>${mary.age}</value></property>
        <property name="spouse"><ref local="simplebeanfromxml"/></property>
    </bean>

    <bean id="propertyConfigurer"
              class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >
        <property name="location">
        <!-- There are options to set this value:
             none/classpath: it will check classpath
```

```
            file: can be used with relative or absolute url

            In a web aware app, it also search through the web root.

            The file: and classpath: notions are only for Spring's Resource type.
            Since the file: accepts absolute path, so we can keep config files
            outside war/ear files.
        -->
        <value>file:config/externalprops.properties</value>
        </property>
    </bean>
</beans>
```

这里的 john.age 和 mary.age 都是引用性的占位符号，它们的实际值是由 propertyConfigurer 里的指定文件的值来定义的。这里我们给出的是相对路径，但也可以用绝对路径或类路径（classpath）。文件内容如下

```
john.age=25
mary.age=23
```

当然，在 Java 调用程序中，我们需要像注册 customEditorConfigurer 一样也注册 propertyConfigurer。

```
package springsamples.springcore.beanfactory.xmlbeanfactory.properties;

import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.beans.factory.config.*;
import springsamples.springcore.*;

public class TestExternalFileConfig
{
    public static void main(String[] args)
    {
        ConfigurableListableBeanFactory  bf = new XmlBeanFactory(new
                                              FileSystemResource("config/externalprops.xml"));

        // ApplicationContext would register that automatically. But BeanFactory
        // doesn't, so we have to manually register this.
        BeanFactoryPostProcessor cec = (BeanFactoryPostProcessor)bf.getBean("propertyConfigurer");
        cec.postProcessBeanFactory(bf);

        Person parent = (Person)bf.getBean("simplebean");
        System.out.println("person name=" + parent);
    }
}
```

值得指出的是，在 WEB 编程中，因为上面的路径可以是绝对路径，所以这个配置文件可以在任何地方，甚至在 WAR 或 EAR 文件之外。但是这是不符合 J2EE 标准的，所以要有足够的理由才能这样做。

2.2.1.7 JVM 系统参数

这里值得一提的是，如果 Spring 在外部文件中找不到占位符号的值，它会在 JVM 的系统中找（如果这也找不到，它只有放弃）。现在把上面的文件改为如下

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

<beans>
    <bean id="simplebeanfromxml" class="springsamples.springcore.Person">
        <property name="name"><value>John Smith</value></property>
        <property name="age"><value>${john.age}</value></property>
        <property name="spouse"><ref local="spousebean"/></property>
        <property name="belongings"><value>cloth,car,cat</value></property>
    </bean>
```

```
    <bean id="spousebean" class="springsamples.springcore.Person">
        <property name="name"><value>Mary Smith</value></property>
        <property name="age"><value>${mary.age}</value></property>
        <property name="spouse"><ref local="simplebeanfromxml"/></property>
    </bean>

    <bean id="propertyConfigurer"
          class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />
</beans>
```

我们删去了外部文件的路径。再试着运行上面的 Java 程序，它找不到所需的值。现在，在 Java 程序 main() 的开始加入如下部分

```
System.setProperty("john.age", "25");
System.setProperty("mary.age", "23");
```

这时应该找到所需的值。这一点有时很有用，例如，我们可以在不同的环境中设置一个 JVM 的系统变量来命名不同的环境，如 DEV，QA，PRODUCTION 等。这样就可以根据不同的环境来选不同的配置。另一方面，也有人用 JNDI 来这样做，这通常不是一个好主意，因为这会依赖于 JNDI，使得测试麻烦。当然，我们可以用诸如 simple-jndi 包来帮忙。

2.2.1.8 两个外部配置文件

一个自然延伸的情况就是如何设置两个外部文件。这时，我们需要区分哪个属性是在哪个文件里定义的，这在 Spring 的类 PropertyPlaceholderConfigurer 中由 placeholderPrefix 决定，我们对不同的外部文件设置不同的前缀值。

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

<beans>
    <bean id="simplebeanfromxml" class="springsamples.springcore.Person">
        <property name="name"><value>John Smith</value></property>
        <property name="age"><value>${johnProps:john.age}</value></property>
        <property name="spouse"><ref local="spousebean"/></property>
        <property name="belongings"><value>cloth,car,cat</value></property>
    </bean>

    <bean id="spousebean" class="springsamples.springcore.Person">
        <property name="name"><value>Mary Smith</value></property>
        <property name="age"><value>${maryProps:mary.age}</value></property>
        <property name="spouse"><ref local="simplebeanfromxml"/></property>
    </bean>

    <bean id="johnConfigurer"
              class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >
        <property name="location">
            <value>file:config/john.properties</value>
        </property>
        <property name="placeholderPrefix">
            <value>${johnProps:</value>
        </property>
    </bean>

    <bean id="maryConfigurer"
              class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >
        <property name="location">
            <value>file:config/mary.properties</value>
        </property>
        <property name="placeholderPrefix">
            <value>${maryProps:</value>
        </property>
```

```
    </bean>
</beans>
```

我们用文件的前缀加上配置占位符来决定配置真实的值。当然，我们可以使用任意可以区分的值。
文件 john.properties 的内容如下：

```
john.age=25
```

文件 mary.properties 的内容如下：

```
mary.age=23
```

配置完了以后，我们在 Java 中需要把这两个 PropertyPlaceholderConfigurer 都注册

```
package springsamples.springcore.beanfactory.xmlbeanfactory.properties;

import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.beans.factory.config.*;
import springsamples.springcore.*;

public class TestTwoExternalFileConfig
{
    public static void main(String[] args)
    {
        ConfigurableListableBeanFactory  bf = new XmlBeanFactory(new
                        FileSystemResource("config/2externalprops.xml"));

        // ApplicationContext would register that automatically. But BeanFactory
        // doesn't, so we have to manually register this.
        BeanFactoryPostProcessor bfpp = (BeanFactoryPostProcessor)bf.getBean("johnConfigurer");
        bfpp.postProcessBeanFactory(bf);
        bfpp = (BeanFactoryPostProcessor)bf.getBean("maryConfigurer");
        bfpp.postProcessBeanFactory(bf);

        Person parent = (Person)bf.getBean("simplebeanfromxml");
        System.out.println("person name=" + parent);
    }
}
```

读者可以参阅 JavaDoc 来了解其它的属性和功能。另一个类似的类是 PropertyOverrideConfigurer。

这里值得指出的一点是，在上面的配置中，我们可以用通配符来定义 properties 文件名。例如，把上面的 micky.properties 换成${micky}。这时，上面的测试会出错。如果我们在 main()的开始加入这一行

```
System.getProperties().setProperty("micky", "micky");
```

那么测试应该通过。这里我们实际上是给${micky}赋了一个系统值，正是这个系统值使得我们可以轻松地指定不同的配置文件。例如，我们可以这样做，对不同的运行环境有这样的配置文件 dev.mysetting.properties，qa.mysetting.properties，prod.mysetting.properties。我们用 ${env}.mysetting.properties 在 XML 文件中指定 location 属性，在不同的环境中设置这个 env 的值。

如果需要配置的内容不是上面的格式，例如是二进制的，我们可以用后面提到的 Resource。

2.2.1.9 其它的配置方式

到目前为止，我们讨论了几种常用的获取属性的办法，Properties 文件，JVM 的系统参数。另外，Spring 自带 JNDI 的查询 JndiBeanFactoryLocator 和 Servlet 的 Context 参数查询 ServletContextPropertyPlaceholderConfigurer。但前者不能像后者一样可以和通配符混用。我们可以仿照后者写出一个 JndiPropertyPlaceholderConfigurer 来做同样的事，但是依赖于 JNDI 的设置是非常有争议的。

2.2.2 BeanFactory 是一个对象容器

从前面的例子中可以看出，BeanFactory 实际上已经成为一个对象容器，它管理对象的生成，协调对象间的依赖关系（以后我们还会谈及对象的存活周期）。所以现在我们从对象容器的角度来探讨一下 BeanFactory。

对象容器有效地松懈了依赖关系，好比是把焊接变成螺丝扣。这样一来，我们可以很轻松地替换组件。同时，由于容器的介入，对象之间的依赖演变成对象依赖于容器，再依赖于另一个对象。所以，它也减少了很多依赖关系。例如，我们不必依赖于 JNDI 和 DataSource，而只依赖于 DataSource，因为 JNDI 已经被移到容器里。所以，对象容器帮助我们对依赖关系分而治之（divide and conquer），把一些标准的，常用的依赖关系都放在专门管理依赖关系的对象容器上。但是，这并不等于我们可以做写程序都做不到的事，例如某些循环依赖。同时，它也要求我们对某些依赖需要特殊处理，例如单例对多例的依赖（Spring 文件 3.3.4 节），因为对象的存活周期不同，生成方式也不同。

BeanFactory 这个对象容器更放松了对 JavaBean 的要求，加入了 constructor 式的注入。尽管我们仍然沿用 bean 这个名字。这实际上把 Java 中的 new 操作交给 BeanFactory 来完成了。这也为陈旧落后的程序或第三方不可改动的程序的集成铺平了道路。

2.2.2.1 单例和多例

BeanFactory 的对象可以通过 bean 标签中的 singleton 属性来生成单例或多例。在单例的情况下，BeanFactory 还管理对象的存活周期(通过 init-method 和 destroy-method)。在多例的情况下，它仅创建所要的对象而不管理该对象的存活周期。在大多数情况下，我们都应该使用无状态的单例，而不是有状态的多例。当然，我们需要考虑单例的线程问题，特别是我们有很多对象之间的引用时，或单例引用多例。另外，Spring 有几个自带的 FactoryBean，这些对象必须是单例，但这些工厂对象的产品却可以是多例。所以，要小心使用这个标签。如果实在没有把握，问计算机(ask the computer)。用 Thread 类测试一下。

2.2.2.2 父 Bean 和抽象 Bean

BeanFactory 有一中类似于样板的机制，用来配置相似的 bean 对象。我们可以先定义一个样板 bean，然后通过修改这个样板 bean 来定义其它的 bean 对象。值得一提的是，尽管这里 bean 的属性叫做 parent，但是这并不要求样板 bean 的类必须是父类。如果我们不想创建样板 bean 的话，我们可以将 bean 的 abstract 属性设置成 true。这种属性继承节省了很多设置，特别是在用 FactoryBean 时。

2.2.2.3 内部 Bean

当我们不想显示某些仅在配置中出现的的对象时，可以使用内部对象。这样，调用者就不能获取这些对象（不论是有意的还是无意的）。事实上，应该有一个专门的 bean 标签来做这件事。

2.2.2.4 别名（alias）

Spring 用 name 标签来定义别名，但这只是为了引用而已。在 1.2 新版本中，增加了新功能，因为它很有用，所以我们在这里介绍一下。新增加的功能就是常说的有名设置，它通过增加一层重新指向（redirection）来

提供引用的选择，即用别名来选择。换句话说，它不是一种覆盖，而是重新定向。例如，我们在定义某个 DAO 的具体实现时，可以指定 JDBC 实现，也可以指定 Hibernate 实现，或 iBATIS 实现。在没有新功能之前，我们只能把一个实现指到 DAO，同时把其它的实现删除或注释掉，不然会出错，因为他们共享一个 id。有了这个新功能，我们就可以用别名连到所要的实现的 id，而所有的实现都有不同的 id，可以共存。所以，它实际上是分离了实现和引用。如果我们把所有的别名放在一起，就很容易更改，而不至于东一个，西一个，忘了改某一个。

2.2.2.5 输入其它 XML 配置（Import）

Import 标签可以用来输入其它的配置文件。这些配置文件也同样是完整的 Spring 文件，而且路径是相对于 Import 所在的地方。路径必须是常量，不能有通配符，如${…}。这些文件的顺序是有意义的，即如果不同文件中的同名对象，后面的定义会覆盖前面的定义。这与后面提到的 ApplicationContext 具有的功能是一样的。

2.2.2.6 Spring 自带的一些工厂方法

对象还可以通过静态或非静态的工厂方法来创建。如果想得到工厂，只需在 bean 的 id 前面加&。在 org.springframework.beans.factory.config 里有一些很有用的 Factory，如 MethodInvokingFacotryBean 等。其它的一些则散布在 AOP，Transaction 等包中。这些类也通过增加一层重新指向（redirection）来提供新的功能或间接引用，例如，我们所需的对象是某个方法的结果，而不是通过 new 来得到的。

2.2.2.7 BeanFactoryWare 接口和 Spring 扩展

BeanFactory 是一个功能强大的对象容器，但有时我们还希望能扩展它来加入一些新的功能。Spring 提供了一些接口来满足这些要求，其中重要的几个接口是 BeanNameAware（知道自己是谁），BeanFactoryAware（知道自己在哪里），BeanPostProcessor 和 BeanFactoryPostProcessor。在前面的讨论中，我们已经见过了一些例子，如 PropertyHolderConfigurer。注意到这些都是 Spring 的 API，所以通常它们是用来扩展 Spring 的而不是用在应用中的。

现在，我们来举一个例子，它是在实践中遇到的问题，即在用 HTTP 和网站通讯时，有时我们要设置 Proxy，有时我们要设置数字证书，还有时我们要管理 Cookies。读者现在应该知道 Spring 是一个轻容器，它可以做依赖注入（属性），也可以做方法注入（lookup-method 和 replace-method）。下面我们将扩展 Spring，加入一种行为注入，目的是分离行为和对象。假定我们有这样一个类

```
package springsamples.springcore.methodinjection;

public class MyTargetExecutor
{
    private String field1 = null;
    private String field2 = "";

    public void setField1(String f1)
    {
        field1 = f1;
    }

    public void setField2(String f2)
    {
        field2 = f2;
    }

    public String getme()
    {
        return field1 + field2;
    }
```

```
}
```

它有两个属性和一个行为，getme()。我们想在调用这个方法之前先设置那两个属性。假定这两个属性的设置很不易，需要另外两个类。

```
package springsamples.springcore.methodinjection;

public class MySetter1
{
    public void setField1(MyTargetExecutor target)
    {
        //假设我们要做很多事才能得到"Hello".
        target.setField1("Hello");
    }
}
```

和

```
package springsamples.springcore.methodinjection;

public class MySetter2
{
    public void setField2(MyTargetExecutor target)
    {
        //假设我们要做很多事才能得到", World".
        target.setField2(", World");
    }
}
```

一种最直接的方法是在 getme()里用 MySetter1 和 MySetter2，这样的后果是 MyTargetExecutor 依赖于这两个类。当我们需要不同的设置时，MyTargetExecutor 就需要更改。我们不想把这种可变性硬写在 MyTargetExecutor 里。另一种方法是先用 MySetter1 和 MySetter2 设置，再调用 MyTargetExecutor 的 getme()。如果用 Spring 来做，程序如下：

```
package springsamples.springcore.methodinjection;

import org.springframework.beans.factory.config.MethodInvokingFactoryBean;

public class TestBehaviorInjectionInMain
{
    public static void main(String[] args)
    {
        MyTargetExecutor executor = new MyTargetExecutor();
        MySetter1 setter1 = new MySetter1();
        MySetter2 setter2 = new MySetter2();

        // now we try to inject behavior.
        MethodInvokingFactoryBean mifb = new MethodInvokingFactoryBean();

        try
        {
            mifb.setTargetObject(setter1);
            mifb.setTargetMethod("setField1");
            mifb.setArguments(new Object[]{executor});
            mifb.afterPropertiesSet();
            mifb.invoke();

            mifb.setTargetObject(setter2);
            mifb.setTargetMethod("setField2");
            mifb.setArguments(new Object[]{executor});
            mifb.afterPropertiesSet();
            mifb.invoke();

            System.out.println("result=" + executor.getme());
        }
```

```
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

如果我们把这段程序用 Spring 的 XML 来描述，就得到下面的内容：

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

<beans>
    <bean id="executor" class="springsamples.springcore.methodinjection.MyTargetExecutor"/>

    <bean id="injection1"
              class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject">
            <bean id="setter1" class="springsamples.springcore.methodinjection.MySetter1"/>
        </property>
        <property name="targetMethod"><value>setField1</value></property>
        <property name="arguments">
            <list>
                <ref local="executor"/>
            </list>
        </property>
    </bean>

    <bean id="injection2"
              class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject">
            <bean id="setter2" class="springsamples.springcore.methodinjection.MySetter2"/>
        </property>
        <property name="targetMethod"><value>setField2</value></property>
        <property name="arguments">
            <list>
                <ref local="executor"/>
            </list>
        </property>
    </bean>
</beans>
```

但是这样一来，就产生出了一个问题，对于上面的 executor 这个 Bean，我们是用单例还是多例？如果它是多例，那么在 injection1 和 injection2 里的对 executor 的引用都会有问题，这些引用都会产生一个新的对象，而不是原来的。如果是单例，那么 injection1 和 injection2 都必须在 executor 之后生成，这一点可以由容器通过依赖性来解决或者由 bean 的 lazy-init 属性来设置。同时，我们在获取 executor 之前，必须以某种方式触发 injection1 和 injection2。先看看简单的单例情况，假设 executor 是单例。这时，我们的程序变动如下：

```
package springsamples.springcore.methodinjection;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class TestBehaviorInjectionInXml
{
    public static void main(String[] args)
    {
        BeanFactory bf = new XmlBeanFactory(new FileSystemResource("config/methodinjection.xml"));
        MyTargetExecutor exe = (MyTargetExecutor)bf.getBean("executor");
        bf.getBean("injection1");
        bf.getBean("injection2");
        System.out.println("result=" + exe.getme());
    }
```

```
}
```

另一种方法是用一个 BeanPostProcessor 来做上面的 injection1 和 injection2 调用。这里，我们可以写一个更一般的类来做这件事。在获取 executor 的时候，我们先调用这个类

```
package springsamples.springcore.methodinjection;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanFactory;

public class BehaviorInjectionPostProcessor implements BeanPostProcessor, BeanFactoryAware
{
    private String targetBean;
    private String[] behaviorTargets;

    private BeanFactory factory;

    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException
    {
        if (targetBean == null) return bean;

        if (!targetBean.equals(beanName)) return bean;


        if (behaviorTargets != null)
        {
            for (int i = 0, j = behaviorTargets.length; i < j; i++)
            {
                factory.getBean(behaviorTargets[i]);
            }
        }

        return bean;
    }

    public Object postProcessBeforeInitialization(Object bean, String beanName)
                                                                        throws BeansException
    {
        return bean;
    }

    public void setBeanFactory(BeanFactory beanFactory) throws BeansException
    {
        this.factory = beanFactory;
    }

    public void setTargetBean(String tb) { this.targetBean = tb; }
    public void setBehaviorTarget(String target) { setBehaviorTargets(new String[] {target}); }
    public void setBehaviorTargets(String[] targets) { this.behaviorTargets = targets; }
}
```

我们检查对象是否是我们感兴趣的，如果是就引发行为。这样，我们只需加载这个类即可。

```
package springsamples.springcore.methodinjection;

import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.beans.factory.config.*;

public class TestBehaviorInjectionInXml2
{
    public static void main(String[] args)
    {
        ConfigurableListableBeanFactory bf = new XmlBeanFactory(new
                                        FileSystemResource("config/methodinjection2.xml"));
        BeanPostProcessor load = (BeanPostProcessor)bf.getBean("loadhook");
```

```
        bf.addBeanPostProcessor(load);

        MyTargetExecutor exe = (MyTargetExecutor)bf.getBean("executor");
        System.out.println("result=" + exe.getme());
    }
}
```

配置文件如下

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "c:/cui/springcore/config/spring-beans.dtd">

<beans>
    <bean id="pre" class="java.lang.String"/>

    <bean id="executor" class="springsamples.springcore.methodinjection.MyTargetExecutor"/>


    <!-- This is static, since it takes longer to initiate the object -->
    <bean id="injection1"
                    class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject">
            <bean id="setter1" class="springsamples.springcore.methodinjection.MySetter1"/>
        </property>
        <property name="targetMethod"><value>setField1</value></property>
        <property name="arguments">
            <list>
                <ref local="executor"/>
            </list>
        </property>
    </bean>

    <bean id="injection2"
                    class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject">
            <bean id="setter2" class="springsamples.springcore.methodinjection.MySetter2"/>
        </property>
        <property name="targetMethod"><value>setField2</value></property>
        <property name="arguments">
            <list>
                <ref local="executor"/>
            </list>
        </property>
    </bean>

    <bean id="loadhook"
                  class="springsamples.springcore.methodinjection.BehaviorInjectionPostProcessor">
            <property name="targetBean"><idref local="executor"/></property>
               <property name="behaviorTargets">
            <list>
                <idref bean="injection1"/>
                <idref bean="injection2"/>
            </list>
        </property>
    </bean>

</beans>
```

在下面我们讨论 ApplicationContext 时，这个 loadhook 对象会被自动加载。

现在，我们考虑 executor 是多例的情况。这时，上面对 executor 的引用都会返回一个新的对象。所以我们对 injection*的注入也是新的对象，而不是所要的。因此，我们必须只有在索取 executor 时才能把 executor 注入到 injection*中去，而不能提前这样做。这就意味着我们必须先把依赖关系保存起来。为此，我们先定义一个类

```
package springsamples.springcore.methodinjection;
/**
```

```
 * This class serves as a temporary holder for the behaviors. The reason is when
 * we want to inject singleton behaviors to non-singleton beans, we can not reference
 * non-singleton beans in the singleton behavior settings(otherwise, a new bean
 * is created, instead of the one that we want to inject into). So we need a temp
 * holder to hold on, then when we use the FactoryBean to create non-singleton target
 * beans, we resolve the reference back to the singleton behaviors.
 *
 * Of course, this class has no behavior, merely a holder. So it's not a first-class
 * object, can we avoid this class?
 */

public class BehaviorHolder
{
    // Behavior object, method, and arguments
    private Object behaviorObject;
    private String behavior;
    private String[] behaviorArgs;

    // specifying this field will trigger static method(behavior field) call on the behaviorObject.
    private Class behaviorClass;
    // or you may specify both class and method in one shot.
    private String staticBehavior;

    public Object getBehaviorObject() { return behaviorObject; }
    public void setBehaviorObject(Object behaviorObject) { this.behaviorObject = behaviorObject; }

    public Class getBehaviorClass() { return behaviorClass; }
    public void setBehaviorClass(Class behaviorClass) { this.behaviorClass = behaviorClass; }

    public String getBehavior() { return behavior; }
    public void setBehavior(String behavior) { this.behavior = behavior; }

    public String getStaticBehavior() { return staticBehavior; }
    public void setStaticBehavior(String staticBehavior) { this.staticBehavior = staticBehavior; }

    public String[] getBehaviorArgs() { return behaviorArgs; }
    public void setBehaviorArgs(String[] behaviorArgs) { this.behaviorArgs = behaviorArgs; }
}
```

这里我们没有考虑 static 的行为。然后再用一个 FactoryBean 来实时地把 executor 或它的属性注入到 injection*中作为参量，最后再把 injection*这些行为注入回到 executor 中。

```
package springsamples.springcore.methodinjection;
/**
 * A factory to create behaviors. They are not AOP, just create a method to call.
 */
import org.springframework.beans.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.config.*;

public class BehaviorInjectionFactoryBean implements FactoryBean, InitializingBean
{
    private Object targetObject;
    private BehaviorHolder[] behaviors;

    private boolean singleton = true;

    private final BeanWrapper bw = new BeanWrapperImpl();

    // we need to sync this method!
    public synchronized Object getObject() throws Exception
    {
        if (this.singleton)
        {
            return this.targetObject;
        }
        else
        {
            // The getter is modified through lookup-method.
            this.setTargetObject(this.getTargetObject());
```

```
            behaviorConfig();
            return this.targetObject;
        }
    }

    public Class getObjectType()
    {
        return targetObject.getClass();
    }

    public void afterPropertiesSet() throws Exception
    {
        if (targetObject == null)
        {
            throw new IllegalArgumentException("No target Object found");
        }
        if (behaviors == null || behaviors.length == 0)
        {
            throw new IllegalArgumentException("No behavior found");
        }

        if (this.singleton)
        {
            behaviorConfig();
        }
    }

    private void behaviorConfig() throws Exception
    {
        bw.setWrappedInstance(targetObject);
        if (behaviors != null)
        {
            for (int i=0, j=behaviors.length; i<j; i++)
            {
                singleBehave(behaviors[i]);
            }
        }
    }

    private void singleBehave(BehaviorHolder behavior) throws Exception
    {
        if (behavior != null)
        {
            MethodInvokingFactoryBean mifb = new MethodInvokingFactoryBean();
            mifb.setSingleton(this.singleton);
            mifb.setTargetObject(behavior.getBehaviorObject());
            mifb.setTargetClass(behavior.getBehaviorClass());
            mifb.setTargetMethod(behavior.getBehavior());
            if (behavior.getStaticBehavior() != null)
            {
                mifb.setStaticMethod(behavior.getStaticBehavior());
            }

            if (behavior.getBehaviorArgs() != null && behavior.getBehaviorArgs().length > 0)
            {
                mifb.setArguments(convertArguments(behavior.getBehaviorArgs()));
            }

            mifb.afterPropertiesSet();
            // the above will fire off invoke in case of singleton.
            if (!this.singleton) mifb.invoke();
        }
    }
    /**
     * convert the arguments of String array to Object array, the base object is
     * the targetObject, which is specified by "this". "null" is null. For anything
     * else, it will treat as targetObject's property. Nested property accepted.
     * @param stringArgs String[]
     * @return Object[]
     */
    private Object[] convertArguments(String[] stringArgs)
    {
```

```
        if (stringArgs == null) return null;
        if (stringArgs.length <= 0) return new Object[0];
        Object[] tmp = new Object[stringArgs.length];
        for (int i=0, len=stringArgs.length; i<len; i++)
        {
            String arg = stringArgs[i];
            if (arg == null) throw new IllegalArgumentException("Argument can not be null");

            if (arg.trim().equalsIgnoreCase("this"))
            {
                tmp[i] = targetObject;
            }
            else if (arg.trim().equalsIgnoreCase("null"))
            {
                break;
            }
            else
            {
                tmp[i] = bw.getPropertyValue(arg);
            }
        }

        return tmp;
    }

    public boolean isSingleton() { return singleton; }
    public void setSingleton(boolean singleton) { this.singleton = singleton; }

    public void setTargetObject(Object targetObject) { this.targetObject = targetObject; }
    public Object getTargetObject() { return this.targetObject; }

    public void setBehavior(BehaviorHolder behavior)
    { setBehaviors(new BehaviorHolder[]{behavior}); }
    public void setBehaviors(BehaviorHolder[] behaviors) { this.behaviors = behaviors; }
}
```

这个类和 MethodInvokingFactoryBean 很相似，用 BeanWrapper 来获取 targetObject 的属性并传给行为(behavior 或 behaviors)。它也可以返回一个单例或多例。有了这个类，我们就可以在 XML 中做如下的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "c:/cui/springcore/config/spring-beans.dtd">

<beans>
    <!-- non-singleton class -->
    <bean id="InjectedExecutor"
                  class="springsamples.springcore.methodinjection.BehaviorInjectionFactoryBean">
        <property name="targetObject">
            <bean id="executor" class="springsamples.springcore.methodinjection.MyTargetExecutor"
                        singleton="false"/>
        </property>
        <property name="behaviors">
            <list>
                <ref bean="injection1"/>
                <ref bean="injection2"/>
            </list>
        </property>
        <property name="singleton">false</property>
    </bean>

    <!-- singleton class, injector-->
    <bean id="injection1" class="springsamples.springcore.methodinjection.BehaviorHolder"
                    lazy-init="true">
        <property name="behaviorObject">
            <bean id="setter1" class="springsamples.springcore.methodinjection.MySetter1"/>
        </property>
        <property name="behavior"><value>setField1</value></property>
        <property name="behaviorArgs">
            <list>
                <value>this</value>
            </list>
```

```
        </property>
    </bean>

    <bean id="injection2" class="springsamples.springcore.methodinjection.BehaviorHolder"
          lazy-init="true">
        <property name="behaviorObject">
            <bean id="setter2" class="springsamples.springcore.methodinjection.MySetter2"/>
        </property>
        <property name="behavior"><value>setField2</value></property>
        <property name="behaviorArgs">
            <list>
                <value>this</value>
            </list>
        </property>
    </bean>

</beans>}
```

这里，行为（behaviors）的配置对象是可以重用的，例如，对象 1 有行为 A 和 B，对象 2 有行为 B，C，D，而行为 B 被重用。另外，这种用 FactoryBean 增加一层抽象或间接的办法是一个很常用的。它的调用如下

```
package springsamples.springcore.methodinjection;

import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.beans.factory.config.*;

public class TestBehaviorInjectionInXml3
{
    public static void main(String[] args)
    {
        // This is one way to do this
        ConfigurableListableBeanFactory bf = new XmlBeanFactory(new
                                                FileSystemResource("config/methodinjection3.xml"));

        MyTargetExecutor exe = (MyTargetExecutor)bf.getBean("InjectedExecutor");
        System.out.println("result=" + exe.getme());
    }
}
```

这样做的好处是我们彻底地分离了 executor 和 injection*，使得 executor 不需要知道 injections。另外，我们也分离了 Spring 的扩展和我们的具体应用，从而使得这个扩展可以被重用。这是 Spring 的一个优点。

但是，我们有一点还没有考虑，就是线程的同步问题。在多个行为同时修改目标对象（这时可能会修改同一个属性）时，或者当目标对象是多例时，我们必须同步，所以要加上 synchronized 关键字。

熟悉 AOP 的读者可以看出，这是 AOP 的一个简单特例，所以也可以用 AOP 来完成，但是会产生对 AOP 的依赖性。

2.3 ApplicationContext 接口和它的实现类

ApplicationContext 是指应用程序的运行环境。这个接口是从 BeanFactory 扩展出来的，所以它拥有 BeanFactory 的一切功能。另外，它增加了一些新的功能：

- 事先创建所有单例对象，不论它们是否被引用（但是可以用 lazy-init 属性来滞后）。
- 自动载入所有 PostProcessor 类，不需要我们手动来做。在前面我们已经看到，如果不这样，我们的程序就会依赖于 Spring 的 API。
- 可以同时载入几个配置文件，如果在这几个配置文件中有相同的 Bean 定义（id 属性相同），后载入的覆盖先后载入的（The lastest wins）。这是一个很有用的功能，它在组件开发和测试中非常有帮助。
- 资源载入
- 国际化支持。

- 事件机制。但是它是依赖 Spring 的 API。如果可能的话，AOP 是更好的选择。

这个接口有很多实现，我们在这里将重点介绍两个实现，ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext。XmlWebApplicationContext 将在后面介绍。另外，StaticApplicationContext 可以用来注册对象。ClassPathXmlApplicationContext 可以从 classpath 中读取配置，FileSystemXmlApplicationContext 可以从文件系统（File System）中读取配置。它们可以接受像 file:或 classpath:或 classpath:bean*.xml 或 classpath*:bean*.xml 等 wildcards，也可以从 jar 文件中批量或单个读取配置。所以，它们非常方便，特别是在测试中。

ApplicationContext 还有一个从 BeanFactory 中继承的功能，即它本身可以组成一个层次结构。每个 ApplicationContext 可以设置一个父类 ApplicationContext。这在 Web 应用中可以在 WAR 和 WAR 之间，或者在 WEB 和 EJB 之间共享 ApplicationContext。值得注意的是，父类 ApplicationContext 和子类 ApplicationContext 没有依赖性同步的覆盖，而只是简单的覆盖。例如，当父类和子类都有某个 bean 时，子类返回，但是父类中其它 bean 对这个 bean 的引用仍然是指向父类。这有别于下面讨论中的覆盖。

2.3.1 同时载入几个配置文件

这是一个非常实用的功能。当我们开发组件(Components)时，如果在组件内部有 Spring 的 XML 配置，我们可以利用这个功能从外面来覆盖组件内的设置。这个功能最重要的一点是它在更新设置时仍维持原有的依赖关系。请看下面的例子。我们这里有 3 个对象，MyParcel 引用 senderAddress 和 shippingAddress。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
                                          "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
       <bean id="senderAddress" class="springsamples.context.Address">
        <property name="city"><value>Mahattan</value></property>
        <property name="state"><value>New York</value></property>
        <property name="country"><value>US</value></property>
     </bean>

    <bean id="shippingAddress" class="springsamples.context.Address">
        <property name="city"><value>St. Louis</value></property>
        <property name="state"><value>Missouri</value></property>
        <property name="country"><value>US</value></property>
     </bean>

    <bean id="MyParcel" class="springsamples.context.Parcel">
        <property name="shipAddress"><ref local="senderAddress"/></property>
        <property name="senderAddress"><ref local="shippingAddress"/></property>
     </bean>
</beans>
```

现在，我们用另一个配置来更新 senderAddress。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
                                          "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
       <bean id="senderAddress" class="springsamples.context.Address">
           <property name="city"><value>Brooklyn</value></property>
           <property name="state"><value>New York</value></property>
           <property name="country"><value>US</value></property>
       </bean>
</beans>
```

由于 ApplicationContext 可以同时加载数个配置，而且后入为主，所以我们可以后加载新的配置

```
package springsamples.context.overriding.test;

import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.*;
import springsamples.context.*;

public class TestConfigFileOverriding
{
    public static void main(String[] args)
    {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            new String[]{"springsamples/context/overriding/test/config1.xml",
             "springsamples/context/overriding/test/config2.xml"}
        );

        Address addr1 = (Address)ac.getBean("senderAddress");
        System.out.println(addr1);
        Address addr2 = (Address)ac.getBean("shippingAddress");
        System.out.println(addr2);
        Parcel p = (Parcel)ac.getBean("MyParcel");
        System.out.println(p);
    }
}
```

这里，我们省略了 Address 类和 MyParcel 类的程序，因为它们只有一些属性和 setter/getter 而已，读者可以自行补上。从打印结果可以看出 Address 后一个配置覆盖了前一个配置，MyParcel 也引用新的 Address。

它的另一个用途是不同环境的设置。例如，如果编程环境和生产环境有 10 对象有不同配置，那么我们可以把这 10 个对象存在一个单独的文件里，在最后载入这个文件即可。这样做的好处是批量修改，不然要一个一个地修改，这时很容易忘掉一个，特别是在因某种原因而压力很大时。这和前面提到的别名异曲同工。

2.3.2 资源载入

Spring 的资源是对诸如文件，URL 等输入流的抽象和扩展。它的输入配置非常多式多样，例如，我们可以用 classpath:，file:，和 classpath*:等。但是它的输出配置却较少，在 ApplicationContext 里面有一个 getResource()方法，另外还有两个 FactoryBean, ResourceFactoryBean 和 PropertyResourceConfigurer。其它的支持则与国际化有关。如果我们想读入二进制的资源，又不想在应用程序中有 Spring 的依赖，Spring 就不够了。例如，我们需要读入数字证书的文件，再把它转换成 Java 的 Certificate 类。这就需要我们对 Spring 进行一下扩展。下面我们就示范如何进行扩展。当然，我们需要一个类来设置从资源读出的值

```
package springsamples.context.resource.Test;

import springsamples.context.*;
import java.io.*;
import java.util.*;

public class TestBean
{
    private String config;

    // This is the method we want to set fields from Resources.
    public void initMe(InputStream is) throws Exception
    {
        Properties props = new Properties();
        props.load(is);
        config = props.getProperty("aaa");
        if (is != null) is.close();
    }

    public String getConfig() { return config; }
    public void setConfig(String config) { this.config = config; }

    public String toString() { return config: " + config; }
}
```

这里，本应该用一个非 String 的属性 config，但为了简单起见，我们仍用 String 类，读者可以替换成其它类型。这里的关键在于 initMe()方法，它的输入是 InputStream，而不是 Spring 的 Resource。在这个方法里，我们可以做任何设置。这样，我们就需要一个新的类来把 Resource 转换成 InputStream。用前面同样的思路，我们构建一个 FactoryBean.

```
package springsamples.context.resource;

import org.springframework.core.io.Resource;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.config.*;
import java.io.*;

public class ResourceSettingFactoryBean implements FactoryBean, InitializingBean
{
    private Object targetObject;
    private String initMethod;
    private Resource[] resources;

    private boolean singleton = true;

    // I think we need to sync this method
    public synchronized Object getObject() throws Exception
    {
        if (this.singleton)
        {
            return this.targetObject;
        }
        else
        {
            // The getter is modified through lookup-method.
            this.setTargetObject(this.getTargetObject());
            resourceConfig();
            return this.targetObject;
        }
    }

    public Class getObjectType()
    {
        return targetObject.getClass();
    }

    public void afterPropertiesSet() throws Exception
    {
        if (resources == null || resources.length == 0)
        {
            throw new IllegalArgumentException("No resource found");
        }

        // singleton setting.
        if (this.singleton)
        {
            resourceConfig();
        }
        // if it's prototype, we have to open and close everytime, really ugly.
    }

    private void resourceConfig() throws Exception
    {
        MethodInvokingFactoryBean mifb = new MethodInvokingFactoryBean();
        mifb.setSingleton(this.singleton);
        mifb.setTargetObject(this.targetObject);
        mifb.setTargetMethod(initMethod);

        InputStream[] mr = convertArguments(resources);
        mifb.setArguments(mr);

        mifb.afterPropertiesSet();
        // mifb will fire off invoke() for singletons only
        if (!this.singleton)
        {
```

```
            // need to manually invoke this.
            mifb.invoke();
        }

        closeInputStreams(mr);
    }

    /**
     * convert the arguments of String array to InputStream array, which will be
     * passed to the init method. So this class hides the reference to the
     * Resource class, just use InputStream.
     * @param resourcesArgs Resource[]
     * @return InputStream[]
     * @throws IOException
     */
    private InputStream[] convertArguments(Resource[] resourcesArgs) throws IOException
    {
        InputStream[] inputStreams = new InputStream[resourcesArgs.length];

        for (int i = 0, j = resourcesArgs.length; i < j; i++)
        {
            inputStreams[i] = resourcesArgs[i].getInputStream();
        }


        return inputStreams;
    }

    /**
     * close the input stream in the array.
     * @param inputStreams InputStream[]
     * @throws IOException
     */
    private void closeInputStreams(InputStream[] inputStreams) throws IOException
    {
        if (inputStreams != null)
        {
            for (int i = 0, j = inputStreams.length; i < j; i++)
            {
                if (inputStreams[i] != null) inputStreams[i].close();
            }
        }

    }

    public boolean isSingleton() { return singleton; }
    public void setSingleton(boolean singleton) { this.singleton = singleton; }

    public void setTargetObject(Object targetObject) { this.targetObject = targetObject; }
    public Object getTargetObject() { return this.targetObject; }
    public void setInitMethod(String initMethod) { this.initMethod = initMethod; }

    public void setResource(Resource resource) { setResources(new Resource[]{resource}); }
    public void setResources(Resource[] resources) { this.resources = resources; }
}
```

这里值得一提的是，通常我们应该只读取资源一次，所以这个对象应该是单例，但我们仍然考虑了多例的情况。下面是 XML 设置，这里我们没有给 config 属性赋值，而由 initMe()来设定。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
                                        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="testBean" class="springsamples.context.resource.ResourceSettingFactoryBean">
        <property name="targetObject">
            <bean id="address" class="springsamples.context.resource.Test.TestBean"/>
        </property>
        <property name="initMethod"><value>initMe</value></property>
        <property name="resource">
            <value>classpath:/springsamples/context/resource/Test/resource1.properties</value>
```

```
        </property>
    </bean>
</beans>
```

资源文件如下

```
aaa=nnnn
```

调用如下

```
package springsamples.context.resource.Test;
/**
 * to load from classpath in web servlet context:
 * http://www.jroller.com/comments/raible/home/package_your_spring_config_files
 *
 */
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.*;
import springsamples.context.*;

public class ResourceTest
{
    public static void main(String[] args)
    {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
                        "springsamples/context/resource/Test/config3.xml");
        Address addr1 = (Address)ac.getBean("testBean");
        System.out.println(addr1);
    }
}
```

这样，我们的类就没有 Resource 的依赖性了。

2.3.3 国际化支持

ApplicationContext 支持 i18n 国际化。它在整个 Context 中自动寻找一个叫 messageSource 的 Bean 并且加载它。在这个 Bean 里，我们可以定义一个或数个某种语言的信息。通常对任何语言只定义一个信息文件。在定义好了后，ApplicationContext 有几个 getMessage()的方法可以用来提取这些信息。但是，Spring 把载入时的 Locale 和获取时的 Locale 混在一起。这就意味着当我们从 ApplicationContext 获取某种语言的信息时，必须同时也用同样的 Locale 从文件系统中读取信息。这就会造成二者的连带性，而且通常二者不一样。一个解决方案是用 native2ascii 工具把文件中的信息转成 unicode。

所有的后缀在 Locale 类的 JavaDoc 里可以查找。另一个有用的类是 ReloadableResourceBundleMessageSource。

2.4 启动 ApplicationContext 的几种方式

前面已经讨论了一些 ApplicationContext 的功能，现在我们扼要地说明一下如何启动它。在 WEB 环境中，我们可以通过 Servlet 容器载入 ApplicationContext。在 EJB 容器中加载它，可以参阅 http://forum.springframework.org/viewtopic.php?t=1863

如果在容器之外，例如单独运行的 Swing 应用，可以考虑 Spring 自带的两个类 SingletonBeanFactoryLocator 和 ContextSingletonBeanFactoryLocator
另外，Spring 另有一个项目 Rich Client Project 可以参考。这里需要注意的是，我们可以构造单例，但必须限制它的引用，使得引用它的对象越少越好。一个办法是用一个工厂对象构造 ApplicationContext，然后

把它存在 System.getProperties()中。这样其它对象(在同一个 JVM 里)可以从 System 里拿到。这似乎是一个可行的办法。一个很流行的错误做法是把 ApplicationContext 存在一个单例里，再在很多类里引用这个单例。

下面是一些关于 ApplicationContext 刷新的讨论：
http://forum.springframework.org/viewtopic.php?t=3937
http://forum.springframework.org/viewtopic.php?t=2786

2.5 控制倒置（IoC），依赖注入（Dependency Injection）和轻型容器

现在，IoC(Inversion of Control)已经成为一个时髦的术语了。控制倒置和轻型容器也成为时髦的技术了。我们在这里无意于时尚，只是来探讨控制倒置和轻型容器的利弊。

控制倒置的原意是指程序流程控制的逆转，例如，Martin Fowler 在
http://www.martinfowler.com/articles/injection.html
指出的例子，本来是我们的过程控制着与用户相互运作的流程，后来变为由用户控制流程，而我们的程序变为根据用户的指令来运行。

我们再用 JNDI 的例子来说明一下（之所以用 JNDI，是因为它很典型，它的查找不是很平常的，实际运行也有很大的依赖性，所以它是典型的麻烦制造者），在以往的情况下，通常是我们先用 JNDI 来查询 DataSource，然后再用 DataSource 来得到 Connection。在控制倒置的情况下，JNDI 查询被移出去了，而我们只关心如何使用 DataSource，不必考虑我们是如何得到 DataSource 的，因为这是由容器来完成的（它在 DataSource 被引用之前已经解决了依赖关系并且设置好了 DataSource），而不是由我们的程序来控制。我们通过在容器里的适当配置，可以用不同的方法得到不同的接口实现。例如，我们可以用 JDBC 驱动器来定义 DataSource，也可以用 JNDI 来查询在应用服务器中已经定义好了的 DataSource。所以，这种控制倒置实际上是分离了依赖对象的使用和获得。换一个角度讲，它分离了依赖对象的界面接口和实现，这使得调用者仅依赖于接口，而不依赖于实现。于是我们可以在不改动调用者的情况下使用不同的实现，所以我们的程序结构是有弹性的。这是 IoC 带来的最大的好处。Martin Fowler 在上面的文章中还指出，这种控制倒置和依赖注入仅仅是分离使用和获得的一种方法，另一种方法是 ServiceLocator。两者的区别在于，依赖注入是一种推入（push），它把依赖对象推入到需要的对象中，而 ServiceLocator 是一种拉出（pull），即去把依赖对象从某个地方拉出来，然后再使用它。这通常会产生出对 ServiceLocator 的依赖性。

依赖注入主要有 3 种形式：Type 1, 2, 3。（不知道是什么意思？我们也不知道！哈哈，是个玩笑）这是旧时工程师典型的命名方式，嗯…，现在仍在用。歼 8 战斗机，明级潜艇，都是类似的命名方式。言归正传，因为这样的三种形式实在是不方便，Martin Fowler 就把它们根据意义从新起了名字：接口注入，setter 注入，和 constructor 注入。接口注入需要我们的类实现特定的接口或继承特定的类。但是这样一来，我们的类就必须依赖于这些特定的接口或特定的类，这也就意味着侵入性。Apache 开源的 Avalon 和 EJB 容器属于这一类。因为它的侵入性，它基本上已经被遗弃了。setter 注入，和 constructor 注入都是无侵入性的。setter 注入需要对每个需要注入的类属性定义 public setter。这种注入具有很大的灵活性，但容易破坏类的状态和行为。例如，一个类有两个属性，a 和 b,还有一个方法是加法，a+b。如果我们只设置了 a 的值而没有设置 b 的值就直接调用加法，那么就肯定出错。所以，在设置属性时，我们必须知道哪些属性组合是有效的，哪些是无效的。而且这些属性的设置顺序也是需要注意的。Spring 容器既有这种注入又有 constructor 注入。单纯的 constructor 注入就没有上面这些问题，因为它是通过 constructor 来定义的，所有的有效属性组合都在 constructor 的参量里。人们形容这样的对象是一等公民，意思是它不会出现类似上面说的这些 “坏”行为。但是它又有别的问题。首先，constructor 里出现的错误有时是很难查的。其次，在继承中，constructor 是不会自动继承下去的，必须要手动来完成。最后，它很难处理一些特殊情况，例如，不是必须的而是可选择的属性（部分原因是 Java 不能在 constructor 中设置默认值），循环的依赖关系。所以，每种形式各有利弊，选择哪种形式，哪

个实现，Spring, Pico，HiveMind, 或 Avalon 容器除了具体的技术要求外，更多的是个人的喜好。至于那些关于容器的争论，还是留给律师们和哲学家吧，他们更喜欢这些基于假设的争论。

轻型容器是和类似于 EJB 等重型容器相比而言的。EJB 等容器的启动一般需要很长时间，而且通常需要在应用服务器中运行。所以，在这些容器中的对象是需要很长时间才能测试的。其它类似的重型容器是 JNDI，IBM 的 WSAD 中的 Servlet 容器（因为这个容器的启动太慢了）。值得一提的是，这里所说的长时间不是一，二个小时，而是 10 秒钟甚至更长的启动时间（一般来讲，用户界面的反应时间应是 5 秒钟以内）。在编程和测试时我们通常需要启动这些容器很多次，如果每次启动需要 5 分钟的话，你可以想象这样的编程会是什么样的效率。我们在实际中见过比这更糟糕的情况。我们还用前面的 JNDI 查询 DataSource 来做例子。在通常情况下，我们是在应用服务器中来做 JNDI 查询的，所以从实质上讲，我们是在应用服务器中用 JNDI 来查询一个和 JNDI 无关的 DataSource。如果我们能在应用服务器外得到 DataSource，那就不必考虑应用服务器的启动时间了。而且，如果我们能不通过 JNDI 得到 DataSource，那么我们就消除了对 JNDI 的依赖性。控制倒置的轻型容器不仅满足了这种需要，而且使得我们很容易地在应用服务器之中或之外进行选择。

同时，轻型容器比普通的注册表的功能多，因为它解决对象间的依赖关系。所以它比注册表又重。

Spring 容器不仅是一个轻型容器，它还提供了很多 J2SE/J2EE 的高级功能的合成，以后再谈吧。

2.6 应用程序的配置

在这一章最后一节里，我们来讨论一下一般的应用程序配置。一般来讲，一个应用软件由多个子系统组成，而每个子系统又有若干个模块组成。由于这些子系统和模块可以有不同的名字，例如服务（Services），组件（components），等等，我们有必要澄清这些含义。我们将使用服务和组件，服务是指可以一个单独发布和运行的系统，而组件是无法单独发布的，通常是通过 API 来调用的。这两者之间的区别主要在于以下几点：

- 维护，例如，我们使用 log4j，现在需要更新它的版本，从 1.28 升级到 1.29。假设我们有两个服务，每个服务都有 100 个组件。由于服务是可以单独发布和运行的，所以这两个服务的 log4j 的更新可以分开进行，彼此互不干扰。另一方面，在每个服务内部，当我们更新 log4j 时，如果这 100 个组件都调用 log4j，那么我们必须同时测试它们，确定它们能共存于新的运行环境。有时还会出现这个组件在低版本上不运行，那个组件在高版本上不运行等问题，需要我们做必要的特殊组装（IBM 和 Microsoft 是这些问题最大的厂商）。
- 另一个差别是通常服务仅发布在一个地方（一个服务器或一个集群），而组件则可能会重用在几十个服务里。这样的话，当我们需要更新这个组件所依赖的某个 lib 时，必须更新几十个地方。而更新服务所依赖的某个 lib 时，则只需更新一个地方。
- 虽然几个服务可以共享一台服务器，但是服务一般是远程的，分布式的。

由于 Spring 在设计时就是在组件层次上的，所以我们将在这一层次上讨论，而不考虑服务层次。我们的中心话题是如何用 Spring 组装组件来产生服务。尽管依赖注入使我们不再顾及依赖是从哪里得到的，但是我们仍需要把这些相互依赖的组件组装起来，以便生产出一个产品。

一个简单的 Spring 配置是数据源：

```
<bean id="simpleDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>org.hsqldb.jdbcDriver</value></property>
    <property name="url"><value>jdbc:hsqldb:c:/springdao/db/springdb</value></property>
    <property name="username"><value>sa</value></property>
    <property name="password"><value></value></property>
</bean>
```

这个配置通常是在编程和测试时用的，在实际运行时我们用数据池，例如用 JNDI 查询：

```
<bean id="jndiDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName"><value>jdbc/MyDataSource</value></property>
</bean>
```

这种具有选择性的配置很常见，例如，

- 我们对不同的运行环境（PROD，QA，DEV）有不同的配置，例如数据库的用户名和口令。
- 不同厂家的产品，例如 WebLogic 的 JMS 或是 TIBCO 的 JMS。
- 或是以上的组合，例如，在 DEV 环境我们用 oscache，在 QA 和 PROD 环境我们用 Tangosol 的分布式缓存。

所以 Spring 提供了几种简便的配置方法。首先，Spring 提供了一个标签 alias，它使得我们可以指定哪一个是我们想要的。例如，假定我们在一个 DAO 中需要一个数据源：

```
<bean id="myDao" class="....">
    ........
    <property name="dataSourceField"><ref bean="dataSource"/></property>
</bean>
```

我们可以把这个数据源用 alias 指到上面定义的两个数据源：

```
<alias name="simpleDataSource" alias="dataSource">
```

或者

```
<alias name="jndiDataSource" alias="dataSource">
```

在编写组件时，为了重用性，数据源通常是组件之外的配置。这样可以使组件适用于不同的数据源，所以上面在组件中（myDao）对 dataSource 的引用是固定的，由组件决定。一旦我们有了组件（或者是我们自己写的，或者是别人写的），我们就可以通过 alias 来配置数据源，让这个组件做它应该做的工作。另一种应用是反过来使用，即我们有两个不同的组件，因为某种原因需要同一个数据源，而它们各自定义了不同的引用，那么我们就可以用 alias 把这两个不同的引用都指到我们定义的数据源上。所以，在开发数个组件时，如果它们有共同的依赖，那么我们就可以利用 alias 来配置这些共同的依赖。

使用 alias 的另一个好处是我们可以把一组 alias 放在一处，在选择时可以一组一组地选择，不必丢三落四。

另一种简便的配置方法是相同 id 覆盖，即对于相同 id 的 bean，后定义的 bean 覆盖先定义的 bean。例如，在 app.xml 中我们有如下定义：

```
<bean id="myDao" class="....">
    ........
    <property name="dataSourceField"><ref bean="dataSource"/></property>
</bean>
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName"><value>jdbc/MyDataSource</value></property>
</bean>
```

一般来讲，当我们在服务器上发布一个应用时，我们会上面这样配置服务器的数据源池。但在测试时，我们希望能在服务器外运行，为此，我们另外在 app-test.xml 中定义一个和上面 dataSource 一样的 bean：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>org.hsqldb.jdbcDriver</value></property>
    <property name="url"><value>jdbc:hsqldb:c:/springdao/db/springdb</value></property>
    <property name="username"><value>sa</value></property>
    <property name="password"><value></value></property>
</bean>
```

如果我们在测试时同时加载这两个 XML 文件

```
ApplicationContext ac = new ClassPathXmlApplicationContext(new String[]
                        {"app.xml", "app-test.xml"});
```

那么在 app-test.xml 中定义的 dataSource 就会覆盖掉在 app.xml 中定义的 dataSource，而且也会覆盖掉所有对它的引用（相比之下，用上一层（parent）ApplicationContext 和下一层（child）ApplicationContext 来加载，例如

```
ApplicationContext ac = new ClassPathXmlApplicationContext(new String[]{"app-test.xml"},
                         new ClassPathXmlApplicationContext("app.xml"});
```

那么在 app-test.xml 中定义的 dataSource 会覆盖掉在 app.xml 中定义的 dataSource，但是不会覆盖掉所有对它的引用。这种覆盖是针对 ClassLoader 的层次的，比如在 EAR 中有 EJB 和 WEB 的 ClassLoader）。这对于测试非常合适，因为不需要改动原有的配置。

最后我们谈一下 Spring 中最灵活的配置手段，PropertyPlaceholderConfigurer。它的简单应用如下：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>${jdbc.driver}</value></property>
    <property name="url"><value>${jdbc.url}</value></property>
    <property name="username"><value>${jdbc.username}</value></property>
    <property name="password"><value>${jdbc.password}</value></property>
</bean>

<bean id="placeholderConfig"
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="jdbc.properties"/>
</bean>;
```

这里我们用了四个占位符${…}，它们真正的值是在 jdbc.properties 文件中定义的（这个文件的默认位置是 classpath）：

```
jdbc.driver=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:c:/springdao/db/springdb
jdbc.username=sa
jdbc.password=pswd
```

Spring 在加载 XML 文件时会自动把这些占位符替换成相应的值。这样做的好处是我们可以把 jdbc.properties 文件放在一个有特定权限的地方来保护 username 和 password（用 file:前缀）。有些公司有着非常严格的安全规定，特别是在访问权限上。所以对于这些公司，这是一个很有用的功能。

另外，PropertyPlaceholderConfigurer 可以嵌套使用，例如我们可以定义若干个占位符

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

<beans>
    <!-- usage -->
    <bean id="usage" class="com.snowlotus.spring.core.SinglePropertyUsage">
        <property name="testValue">
            <value>${app-${env:environment}.${env1:nestedvalue}}</value>
        </property>
    </bean>

    <bean id="myConfigurer2"
             class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="order" value="3"/>
        <property name="location">
            <value>classpath:com/snowlotus/spring/core/prop4.properties</value>
```

```
        </property>
    </bean>

    <!-- single property selected by the env selector -->
    <bean id="myConfigurer"
              class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="order" value="2"/>
        <property name="location">
            <value>classpath:com/snowlotus/spring/core/prop3.properties</value>
        </property>
        <property name="placeholderPrefix"><value>${env1:</value></property>
    </bean>

    <!-- env selector -->
    <bean id="env" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="order" value="1"/>
        <property name="location">
            <value>classpath:com/snowlotus/spring/core/version.properties</value>
        </property>
        <property name="placeholderPrefix"><value>${env:</value></property>
    </bean>
</beans>
```

当然，我们需要指定它们应该按什么样的顺序来替换 Properties 里的值，即先替换${env:environment}和${env1:nestedvalue}的值（假设分别是 dev 和 status），然后再替换${app-dev.status}的值。

有时，我们只需替换一个值，例如，当我们需要和另一个系统交谈，通常这个系统会像我们自己的系统一样有不同的运行环境，DEV，QA，和 PROD（如果没有，你应该考虑这个系统是否可靠）。这时，通常我们只需改变 HTTP 地址即可，我们可以很方便地用 PropertyPlaceholderConfigurer 来定义单个变量，例如

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

<beans>
    <!-- usage -->
    <bean id="usage" class="springsamples.springcore.SinglePropertyUsage">
        <property name="testValue">
            <value>${app-${env:environment}.target}</value>
        </property>
    </bean>

    <!-- The order is significant!!! env resolves first, then app-...
         using depend-on attribute doesn't work-->
    <!-- env selector -->

    <bean id="env" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >
        <property name="location">
            <value>classpath:springsamples/springcore/version.properties</value>
        </property>
        <property name="placeholderPrefix"><value>${env:</value></property>
    </bean>


    <!-- single property selected by the env selector -->
    <bean id="myConfigurer"
                class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="properties">
            <props>
                <prop key="app-dev.target">http://m1/test1</prop>
                <prop key="app-qa.target">http://m2/test2</prop>
                <prop key="app-prod.target">http://m2/test2</prop>
            </props>
        </property>
    </bean>
</beans>
```

这里，在第一个 bean 中我们需要一个变量 testValue，它要根据另一个变量 env（在第二个 bean 中定义）的值（分别为 dev, qa, 和 prod）来决定取那个值（在第三个 bean 中定义）。这样，我们就可以避免为了一个值而出现很多小文件。而且，当我们由于变化而需要多于一个变量时，只需更改这个 XML 文件即可，无需改动 java 程序。这里必须注意的是 env 和 myConfigurer 的定义是有顺序的，要按照引用的顺序，内层为先，外层为后。

PropertyPlaceholderConfigurer 的另一个很有用的功能是它的 location/locations 属性可以是一个 JVM 的参数占位符。由于前面我们已讲过，这里就不重复了，仅举一例：

```
<bean id="myDao" class="....">
            ........
            <property name="dataSourceField"><ref bean="${dataSource}"/></property>
</bean>

<bean id="placeholderConfig"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="dataSource.${env}.properties"/>
</bean>
```

这里，env 可以定义为 test 或 prod。根据 env 的值，我们决定从哪个文件加载 dataSource 的值。dataSource.test.properties 文件的定义如下：

```
dataSource=simpleDataSourced
```

而 dataSource.prod.properties 的定义如下：

```
dataSource=jndiDataSource
```

这个功能使得我们可以在运行启动时，通过 JVM（-D）的参数选择不同的配置。这是最常用的一种配置方法之一。另一种方法是通过文件里的参数来选择，例如，我们可以定义一个文件 env.properties，它的内容如下：

```
# values are local, dev, qa, prod
env=dev
```

我们希望能像前面 myDao 一样通过这个值来选择我们的配置，但是 Spring 目前还没有这个功能，这是因为我们的占位符${env}在 PropertyPlaceholderConfigurer 的配置里面，而 Spring 是先把 location 的设置用 PropertyEditor 从 String 转成 Resource，然后再替换占位符。所以错误是在从 String 转成 Resource 时发生的，因为这时我们还没有替换占位符。一种解决的办法是我们延迟从 String 到 Resource 的转换。这种延迟有几种思路可以实施，但是我们只试出了下面的方法。这个方法的思路是我们新建一个 BeanFactoryPostProcessor（使得 ApplicationContext 能自动加载），用 String 暂时存储 location，让 PropertyPlaceholderConfigurer 在这个类内部替换占位符后，再把 String 转成 Resource。这样做的一个不方便是我们必须把 PropertyPlaceholderConfigurer 的所有属性再在这个类中重复一遍（为了在我们的这个类里使用这些属性）。（我们试图扩展 PropertyPlaceholderConfigurer 从而弥补这个缺陷，但没有成功）。下面就是我们的实现，因为这仅仅是一个示范，我们就省略了一些 PropertyPlaceholderConfigurer 的属性，请读者自行补上。

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.*;
import org.springframework.core.io.*;

public class NestedPropertyPlaceholderConfigurer implements BeanFactoryPostProcessor
{
    private String beanName;

    private String[] stringLocations;

    private String placeholderPrefix = PropertyPlaceholderConfigurer.DEFAULT_PLACEHOLDER_PREFIX;

    private String placeholderSuffix = PropertyPlaceholderConfigurer.DEFAULT_PLACEHOLDER_SUFFIX;

    private int systemPropertiesMode =
```

```
                    PropertyPlaceholderConfigurer.SYSTEM_PROPERTIES_MODE_FALLBACK;

    private boolean ignoreUnresolvablePlaceholders = false;

    private String fileEncoding;

    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
              throws BeansException
    {
        if (stringLocations == null || stringLocations.length == 0) return;

        Resource[] resources = new Resource[stringLocations.length];
        ResourceResolver resolver = new ResourceResolver();
        for (int i=0; i<resources.length; i++)
        {
            resources[i] = resolver.resolveResource(stringLocations[i]);
        }

        PropertyPlaceholderConfigurer configurer = new PropertyPlaceholderConfigurer();

        //我们仅设置了一些属性
        configurer.setBeanFactory(beanFactory);
        configurer.setBeanName(this.beanName);
        configurer.setFileEncoding(this.fileEncoding);
        //configurer.setIgnoreResourceNotFound(this.ignoreUnresolvablePlaceholders);
        configurer.setIgnoreUnresolvablePlaceholders(this.ignoreUnresolvablePlaceholders);
        //configurer.setLocalOverride();
        configurer.setLocations(resources);
        //configurer.setOrder();
        configurer.setPlaceholderPrefix(this.placeholderPrefix);
        configurer.setPlaceholderSuffix(this.placeholderSuffix);
        //configurer.setProperties();
        //configurer.setPropertiesPersister();
        configurer.setSystemPropertiesMode(this.systemPropertiesMode);
        //configurer.setSystemPropertiesModeName();

        configurer.postProcessBeanFactory(beanFactory);
    }

    // all the settings
    。。。。。
}
```

这里我们用到了一个 ResourceResolver，它的功能相当于 Spring 的 ResourceEditor

```
mport org.springframework.core.io.*;

public class ResourceResolver extends AbstractPathResolvingPropertyEditor
{
    private final ResourceLoader resourceLoader;

    public ResourceResolver()
    {
        this.resourceLoader = new DefaultResourceLoader();
    }

    public ResourceResolver(ResourceLoader resourceLoader)
    {
        this.resourceLoader = resourceLoader;
    }

    public Resource resolveResource(String text)
    {
        String locationToUse = resolvePath(text).trim();
        return this.resourceLoader.getResource(locationToUse);
    }
}
```

这个新的类和 PropertyPlaceholderConfigurer 的用法是一样的，只是它可以用 properties 来设置占位符。例如，

```
<bean id="myDao" class="....">
    ........
    <property name="dataSourceField"><ref bean="${dataSource}"/></property>
</bean>

<bean id="dataSourceConfig" class="......NestedPropertyPlaceholderConfigurer" depends-on="env">
    <property name="location" value="dataSource.${env}.properties"/>
</bean>

<bean id="env" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="environment.properties"/>
</bean>v
```

在 dataSourceConfig 中，${env}先被 bean env 里 environment.properties 中定义的值替换掉，然后 myDao 中的 ${dataSource}会被相应的 dataSource.XXX.properties 中的值替换掉，最后 Spring 再配置依赖关系。

值得指出的一点是，我们可以结合以上这两种方式，即用 JVM 参数和 properties 文件中的设置。在 Spring 的类 PropertyPlaceholderConfigurer 中有一个属性，systemPropertiesMode，我们可以通过设置这个参数来指定是先检查 JVM 参数是否设置还是先检查 properties 文件中是否设置。

最后，我们举一个简单例子来说明如何综合使用 Spring 的配置功能。为了简便，我们假定我们有三种运行环境，DEV（development，即开发环境），QA（Quality Assurance，即质量控制，测试环境），和 PROD（生产运行环境）。通常在编译程序时有两种方法，一种是针对每种运行环境编译，这样我们就有三种编译后的软件，分别运行在各自的环境中。这也就意味着在运行时没有运行环境的选择，选择是在编译是做出的。另一种是编译时不做选择，而在运行时选择，可以通过运行时的参数或某个文件中的配置。这两种方法各有好处和不足，例如前者需要三次编译，这就有如何保证每次都正确无误的问题；后者虽然只有一次编译，但是因为要在不同环境下运行而有不必要的累赘。所以，这两种方法的选择更多地是由具体产品和每个公司的政策决定的。

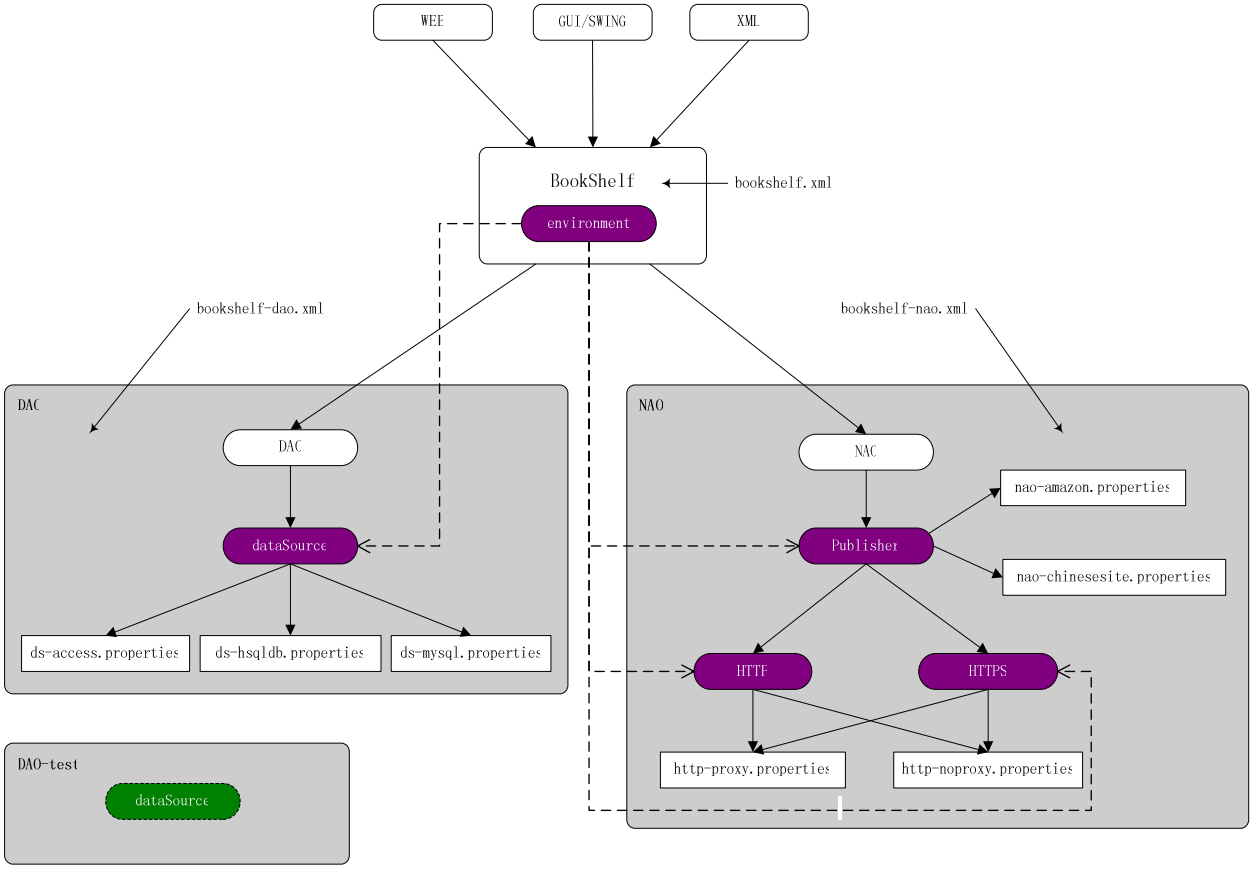
由于前一种方法是由编译决定的，我们略去不谈，有兴趣的读者可以参考 ANT 或 Makefile 相关的资料。我们下面主要讨论后一种方法，即如何在运行时选择配置。

这个例子是一个管理电子图书的软件，我们给它起了个名字叫 BookShelf（书架），它有如下功能：

- 它有一个数据库，可以存储电子图书的信息，例如书名，页数，出版社等。
- 它能上网到网上书店查询，例如 Amazon 等。
- 它可以定义新的分类，可以记读书笔记。

假定我们需要两种运行环境，Office 和 Home。在家上网不用通过防火墙，在办公室上网则须通过防火墙。另外，查英文书可以去 Amazon，但是查中文书则须去中文网。有的网上书店用 HTTP，有的用 HTTPS（这里，我们是一个虚构的场景，我们假定要求是或者去 Amazon，或者去中文网，而且 Amazon 有 HTTPS）。另外，每个人的数据库喜好也不一样，有人喜欢用 Access，有人喜欢用 hsqldb，等等。它可能用 WEB 网页或 SWING 作为前端，也可以输出 XML。读者应该注意到，我们没有把这些和上面三条列在一起，因为这些不是行业行为。

我们将不深入到具体细节，不考虑每个类是如何实现的，而是在组件层次上讨论如何配置组装。下面是这个软件的设计图，这里有中间的行业组件 bookshelf，它的界面就是上面列出的三个功能。另外还有 DAO，NAO，WEB，SWING，和 XML 组件。为了简化，我们只考虑 DAO 和 NAO 两个组件（这两个和 bookshelf 组件已经足够让我们说明问题了）。



DAO 组件需要可以配置几个不同的 DataSource，这些 DataSource 被 DAO 的实现类调用。NAO 组件有两层可变的配置，第一层是网站的选择；在选定网站后，第二层的选择是通过 HTTP 还是通过 HTTPS。而最后的选择是由 bookshelf 这个行业组件的配置决定（为了清晰，所有的紫颜色单元都是配套，即 bean 的定义再加上 PropertyPlaceholderConfigurer）。在 bookshelf 中，有一个 environment 参数，它可以是定义在另一个 properties 文件中，也可以是直接定义在它所在的 PropertyPlaceholderConfigurer 里，更可以是 JVM 的-D 参数。它的功能就是选择其它的配置。

下面这是 DAO 中 DataSource 的定义：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>${daoEnv:jdbc.driver}</value></property>
    <property name="url"><value>${daoEnv:jdbc.url}</value></property>
    <property name="username"><value>${daoEnv:jdbc.username}</value></property>
    <property name="password"><value>${daoEnv:jdbc.password}</value></property>
</bean>

<bean id="daoPlaceholderConfig" class="。。。。。。/NestedPropertyPlaceholderConfigurer">
    <property name="location"
         value="classpath:springsamples/springcore/dao/ds-${environment}.properties "/>
    <property name="placeholderPrefix"><value>${daoEnv:</value></property>
</bean>
```

这里，dataSource 中的变量由 daoPlaceholderConfig 中的 properties 文件定义，而这个文件是由${environment}来选择。在测试时，我们另外定义一个 dataSource，用前面讲过的方法覆盖掉这个 dataSource。

NAO 中的定义也是类似的，我们不再重复。

${environment}是在 bookshelf 中定义的

```
<bean id="bookshelfPlaceholderConfig"
          class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer ">
    <property name="location" value="classpath:springsamples/springcore/env.properties "/>
    <property name="placeholderPrefix"><value>${env:</value></property>
</bean>
```

这里我们可以用 properties 文件，或者直接在 props 属性中定义。而在运行时，还可以先检查-Denvironment 的定义。

现在来看看如何把${environment}的值映射到 DAO 和 NAO 的选择上。一种直接的办法是照葫芦画瓢，例如，如果${environment}的值有 office，home-access，home-hsqldb，那么我们就相应的在 DAO 里有 ds-office.properties，ds-home-access.properties，和 ds-home-hsqldb.properties；在 NAO 里也有相应的定义。虽然在大多数场合下，这种方法已经足够了，但有时这种映射会搅乱不同组件的配置，例如在这里，如果我们定义${environment}的值有 office，home-access，home-hsqldb，那么在 NAO 中就要相应的配置：http-proxy-office.properties，http-proxy-home-access.properties，和 http-proxy-home-hsqldb.properties。这就意味着在 NAO 的配置中有 dataSource 的因素。当然，我们可以起一个更好的名字，即有行业意义。另一种映射的方法是在 DAO 和 NAO 里再定义一层间接映射，把外面的值转换成内部的值，例如在 NAO 中我们可以把 office 映射到 office，把 home-access 和 home-hsqldb 同时映射到 home（因为 access 和 hsqldb 在 NAO 里是无意义的选择）。这样做的好处一个是我们可以分开定义，而且还可以定义默认值；另一个好处是我们可以保持组件的封闭性，从而遵循开闭原则。封闭性的保持使得添加或更改${environment}的值更容易。

Spring 给我们提供了一种手段，使得我们可以在行业组件中选择 DAO 和 NAO 组件中的配置。注意到在行业组件中，我们根本就不应该知道 DAO 和 NAO 组件的配置，所以 Spring 帮助我们解决了一个本身是矛盾的问题：如何选择我们本来就不应该知道的配置。在 Spring 出现之前，每个需要编写配置的程序员都面临这个问题，它一直就没有得到很好的解决。Spring 通过引入另一维运行（不是正常的程序执行运行，诸如 request-response）而有效地解决了这个本身就自相矛盾的问题。通常情况下，类是通过发送消息（即方法调用）来相互作用的，这就意味着在每次执行时，执行过程必须配置必要的对象，因为我们只有执行过程这一维，换句话讲，我们在一条腿走路。这就迫使我们不得不把所需的都统统地挂在执行过程上。这是很多编程和设计缺陷的根源。Spring 通过引入“对象容器”使得我们有了另一维，即配置的运行过程。有了这两个分开的运行，我们就有了很大的回旋余地，可以分离很多无关的行为和焦点，所以我们这时是两条腿走路。

Spring 的这个优点是无出其右的，它同时也解决了单例的滥用问题，对象的产生界面和调用界面的分离问题等一些残留了很久的难点。这使得我们可以更好地遵循前面提到的 OO 原则。