Spring Framework Reference 中文版

WORK IN PROGRESS!!!

版本: 1.0

1. 第一章.简介

1.1. 轻量级容器

这章内容没什么价值，需要修正

最近，许多人对我们所说的“轻量级容器”表现出兴趣。我们认为这是一个趋势。轻量级容器对于 web 应用的意义尤为明显；但对于其他类型的应用（包括在 J2EE 环境中运行的应用和独立的应用）来说，轻量级容器也有助于提高它们的复用程度，例如业务对象和数据访问对象（DAO）的复用。

什么是轻量级容器？EJB 可能是最好的反例：

1. 侵略性的 API(你的代码依赖于 EJB)
2. 对容器的依赖(你的代码不能在 EJB 容器之外工作)
3. 只提供固定的一组功能,不具备配置能力
4.
5. 启动时间长
6. 部署过程取决于特定的产品，无法通用

轻量级容器的目标是避免上面所有这些麻烦事情.

1.2. 几句闲谈

在这里多谈谈 Spring!

2. 第二章.高层面概述

2.1. 简介

简单地说，Spring 是一个以控制倒置(Inversion of Control)原则为基础的轻量级框架。控制倒置是一个用于“基于组件的体系结构”的设计模式，它将“判断依赖关系”的职责移交给容器，而不是由组件本身来判断彼此之间的依赖关系。当在 Spring 内实现组件时，容器“轻量级”的方面就展现出来了：针对 Spring 开发的组件不需要任何外部库；而且，容器是轻量级的，它避免了像 EJB 容器那样的重量级方案的主要缺点，例如启动时间长、测试复杂、部署和配置困难，等等。

这一章首先介绍了轻量级容器的总体设计，随后将简单介绍 Spring 除 IoC 实现之外的特性。简单说，这些特性包括：

* 内置 AOP 支持，例如在 EJB 容器之外提供声明式事务管理

* 数据访问框架,支持 JDBC 和 O/R mapping 产品（例如 Hibernate）
* 与 Spring framework 其他功能完全集成的 MVC web 框架，提供一种清晰、无侵略性的 MVC 实现方式,使你无须绑定任何一种特定的视图技术
* 用 JavaMail 或其他邮件系统发送邮件的支持
* 源代码级别的元数据支持，使开发者可以借助 AOP 之类的技术进行企业服务建模
* JNDI 抽象层，便于改变实现细节，例如透明地在远程服务和本地服务之间切换

此外,这一章将指导你“什么时候为特定的项目选择 Spring”。对于 Spring 和 IoC 的一些优势(以及缺陷)，本章也将提供整体性的概述。

3. 第三章.Bean相关包

3.1. 介绍

TODO:我的看法是作一个概括性的描述，帮助读者基本了解 beans 包能做些什么。然而，如你所见，这是一件非常困难的事……我必须作更多的考量……这需要增加更多关于 IOC 的信息，以及一些有关 type2 和 type3 的资料……

Spring 的核心是 org.springframework.beans 包，为使用 JavaBeans 技术、按名检索对象及管理对象间的关系而设计。beans 包及其子包提供的功能为使用 JavaBeans 技术的项目指定了一个基础设施。

关于 beans 包,有三个重要的概念。首先,它提供了设置/读取 Javabeans 属性功能的 BeanWrapper 接口。第二个重要概念是 Bean 工厂。BeanFactory 是一个泛化工厂，具有实例化对象和管理不同对象间关系的能力。BeanFactory 可以管理几种不同类型的 bean,并且支持串行化及其他生命周期方法。BeanFactory 是按照 bean 定义（BeanDefinition）来实例化对象的。BeanDefinition，顾名思义，是你对 bean 的定义。BeanDefinition 不仅定义了 bean 的类结构、实例化的方式，还定义了 bean 在运行时的合作者。这是第三个概念。这三个概念(*BeanFactory*, *BeanWrapper* and *BeanDefinition*)将在下文详细讨论。

3.2. 使用 BeanWrapper 接口操作 Bean

对于 org.springframework.beans 包遵循 Sun 公司发布的 JavaBeans 标准。所谓“JavaBean”其实就是一个 Java 类。不过,它必须拥有默认无参数构造器,并按照既定规则来命名属性——属性 prop 对应一个设置器 setProp(…)和读取器 getProp(…)。更多的关于 JavaBeans 的信息请查阅 Sun 公司网站(java.sun.com/products/javabeans[http://java.sun.com/products/javabeans/]).

一个对于benas包非常重要的概念是BeanWrapper接口及与之对应的实现(BeanWrapperImpl类)。如 JavaDoc 中所载，BeanWrapper 接口提供了设置和读取属性值、获得属性描述以及查询属性是否可读写的功能。BeanWrapper 也提供了嵌套属性的支持，允许设置无限深度的子属性。同时，BeanWrapper 接口还允许你加入标准 JavaBeans 规范中的 PropertyChangeListeners 接口和

VetoableChangeListeners 接口，从而提供订阅属性变化消息和否决属性变化通知的能力，而不需要在目标类中加入额外的支持代码。最后，BeanWrapper 接口提供了按索引随机设置带索引属性（例如类型为 List 或数组的属性）的支持。BeanWrapper 接口通常并不是直接在应用程序代码中使用，而是由 DataBinder 接口和 BeanFactory 管理。

BeanWrapper 接口的这种工作方式多少就如它的名字所暗示的：对 bean 进行包装，然后对包装后的 bean 加以操作，例如设置属性值、检索属性值等等。

3.2.1. 设置和读取基本属性及嵌套属性

设置和读取属性分别通过 setPropertyValue(s)方法和 getPropertyValue(s)方法来完成。为了方便使用，这两个方法都有几个重载版本。有关这两组方法的详尽描述在 Spring 的 JavaDoc 中给出。在这里，你有必要首先了解“标识对象的属性”的几种命名规则。下面是几个例子：

Table 3.1. Examples of properties

表达式	说明
name	表示属性“name”，对应的方法是 getName()或者 isName()或者 setName()
account.name	表示属性“ account”的嵌套属性“name”，对应的方法是 getAccount().setName()或者 getAccount().getName()
account[2]	表示带索引属性“account”的第三个元素。带索引属性可以是 array 类型、list 类型或者其他普通的 collection

下面你会找到几个使用 BeanWrapper 接口读取和设置属性的例子。

注意：假如你不打算直接使用 BeanWrapper 接口的话，这个部分对你来说并不太重要。假如你只打算使用 DataBinder 和 BeanFactory 公开提供的功能，请放心地跳过这些例子，直接进入有关 PropertyEditors 的内容。

考虑如下两个类：

```
public class Company {
    private String name;
    private Employee managingDirector;
    public String getName() {
      return this.name;
    }
    public void setName(String name) {
      this.name = name;
    }
    public Employee getManagingDirector() {
      return this.managingDirector;
    }
    public void setManagingDirector(Employee managingDirector) {
      this.managingDirector = managingDirector;
```

```
    }
}
```

```
public class Employee {
    private float salary;
    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

下面的代码片断展示如何检索和操作 Companies 类和 Employees 类实例属性的几个例子。

```
Company c = new Company();
BeanWrapper bwComp = BeanWrapperImpl(c);
// setting the company name...
bwComp.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue v = new PropertyValue("name", "Some Company Inc.");
```

```
bwComp.setPropertyValue(v);

// ok, let's create the director and tie it to the company:
Employee jim = new Employee();
BeanWrapper bwJim = BeanWrapperImpl(jim);
bwJim.setPropertyValue("name", "Jim Stravinsky");
bwComp.setPropertyValue("managingDirector", jim);
// retrieving the salary of the managingDirector through the company
Float salary = (Float)bwComp.getPropertyValue("managingDirector.salary");
```

3.2.2. 使用 PropertyEditors 包转换属性

有时候，为了使用方便，我们需要以另一种形式展现对象的属性。例如，日期可以以一种更容易阅读的形式表现出来，同时我们也会将人们熟悉的格式转换回原始的日期格式（或者使用一个更好的办法：将所有用户偏好形式转换回统一的 java.util.Date 对象）。为了达到这一目的，我们可以编写自定义编辑器，使其继承 java.beans.PropertyEditor 类型，并将自定义编辑器注册到 BeanWrapper 上。通过注册， BeanWrapper 将知道“如何把属性转换所需类型的信息”。请阅读 Sun 公司提供的 java.beans 包中 PropertyEditors 的 JavaDoc 获得进一步信息。

下面的例子将使用 PropertyEditor，把 java.util.Date 对象转换为人们习惯的形式：

```
/** Details in this class are excluded for brevity */
```

```
public class Person {
	public void setBirthDay(Date d);
	public Date getBirthDay();
}
/** and some other method */
public void doIt() {
	SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy");
	// CustomDateEditor located in org.springframework.beans.propertyeditors
	// true indicated that null values are NOT allowed
	CustomDateEditor editor = new CustomDateEditor(df, false);
	Person p = new Person();
	BeanWrapper bw = new BeanWrapper(p);
	bw.registerCustomEditor(editor);
	// this will convert the String to the desired object type: java.util.Date!
	bw.setPropertyValue("birthDay", "22-12-1966");
}
```

PropertyEditors的观念无论是对于MVC框架还是对于Spring框架中的其他部分都非常重要，所以在这个文档的剩余部分还会多次出现这个概念。更多的有关自定义编辑器的信息请参考JavaBeans规范(http://java.sun.com/products/javabeans)。

3.2.3. 事件传播

TODO: some info on changelisteners and things like that

3.2.4. 其他有用特性

除了以上你所看到的特性，别的一些东西或许也会引起你的兴趣，虽然这些特性不太值得用完整的一节来介绍。

- 决定属性是否可读写：你可以使用 isReadable()和 isWritable()方法决定一个属性是否可以被读写。
- 检索属性描述符：你可以使用 getPropertyDescriptor(String) and getPropertyDescriptors()方法检索 java.beans.PropertyDescriptor 类型对象的属性描述符，有时这是一个很方便的功能。

3.3. The BeanFactory

org.springframework.beans.factory包及其子包提供了SpringIoC容器的基础。Spring的BeanFactory接口支持IoC类型 2 和类型 3。更多的相关信息可以在后文找到。BeanFactory接口通过一个中心配置仓库提供了一种按名获得bean的方法，免除了需要在不同地方使用单独的Java对象从实例配置文件中读取配置信息的不便。BeanFactory的职责是在需要的时候创建可用的实例。
对于 BeanFactory 接口，有两个问题很重要，第一个是，BeanFactory 接口的不同实现以及如何使用 BeanFactory 接口检索 bean？其次，BeanFactory 接口的实现如何知道实例化对象的方式？它们在返回对象给我们使用前做了些什么？后者涉及到 bean definitions 的概念。

3.3.1. Bean Definitions

Bean definitions 是你的 bean 的详细描述。Bean 就是一些提供某些特定功能的普通类，BeanFactory 接口如何管理你的 bean 以及它们是怎样配置的,都是在一个 bean definition 中规定的。以下就是 bean definition 实际模型，这个模型使得 bean definition 能够实例化 bean。

- beanclass——bean definition 所描述的 bean 的真正实现者
- bean 行为配置元素——bean 在容器中应表现出的状态（如：prototype 模式还是 singleton 模式，自动装配模式，依赖检测模式，初始化及析构方法）
- properties——bean 的配置数据。你可以把它想象成一个类似连接池 bean 缓存的连接数这样的参数（通过属性或者构造函数的参数指定）
- 其他的 bean——你的 bean 工作所需要的其他 bean(同样也是通过属性或者构造函数的参数指定)

上述几个概念直接地对应 bean definition 的元素集合。这些元素在下面的表格中列出，有关它们的更深入的文档将在参考中一一给出。

表 3.2. Bean definition 说明

特性	详细信息
class	参考 3.3.2 ， “The bean class”
singleton or prototype	参考 3.3.3 ， 单例还是原型
bean properties	参考 3.3.4 ， 设置bean的属性及其合作者
constructor arguments	参考 3.3.4 ， 设置bean的属性及其合作者
autowiring mode	参考 3.3.5 ， 自动装配的合作者
dependency checking mode	参考 3.3.6 ， 依赖检查
initialization method	参考 3.3.7 ， bean的自然特性和生命周期特性
destruction method	参考 3.3.7 ， bean的自然特性和生命周期特性

借助现成的 BeanFactory 实现（如：XmlBeanFactory），以上所述各种特性使你的 bean 能够在应用程序外部配置。关于以上特性更详细的信息见后文。

3.3.2. The bean class

当然，你需要为你的 bean 指定一个真正的类，这是显而易见的。这里对你的 bean 类完全没有任何特别的要求，甚至不必实现特定的接口与 Spring 整合。只要指定 bean 类就足够了。但是，取决于你将对某些特殊的 bean 实施何种类型的 IoC，你的 bean 要有一个与之相应的默认构造器。

BeanFactory 接口不仅仅能管理 bean，实际上它能管理你所期望的任意类。很多人在使用 Spring 时喜欢用 BeanFactory 管理“真正”的 bean（仅有一个默认构造器、私有属性及其读取器和设置器），事实上它也可以持有非 bean 形式的类。比如一个从以前系统遗留下来的连接池类－－它完全不符合 bean 规范，不用担心，Spring 同样可以很好的管理它。

3.3.3. 单例还是原型

Bean 以两种形态存在：singletons 形式和 prototypes 形式。当 bean 以 singletons 形态存在时，BeanFactory 只管理一个共享的实例。所有对这个特定 bean 的实例请求，都导致返回这个唯一 bean 实例的引用。

当 bean 以 prototype 形态存在时，每次对这个 bean 的实例请求都会导致一个新的实例的创建。当用户需要不受其他用户对象影响的对象或有类似的需求时，这是一个较理想的解决办法。

Bean 默认是以 singleton 形态存在的，除非你另外显式加以指定。留神，当把 bean 的设置改为 prototype 模式时，每次对这个 bean 的实例请求都会导致一个新的 bean 实例被创建，而这可能并不是你所期望的。所以，只应该在确实需要的情况下把 bean 设置为 prototype 模式。

3.3.4. 设置 bean 的属性及其合作者

IoC（反向控制）的基本原理通常又被称作“好莱坞原则”（不要给我们打电话，我们会给你打电话）。这种理念意味着 bean 不知道它的合作者是谁，也不知道它自己需要哪些附加属性。取而代之的是（在 Spring 的 Ioc 实现中）：BeanFactory 将负责识别 bean 的合作者，并将它们指派给需要使用它们的 bean。这些工作在之前讨论的 bean definitions 中完成。在 bean definitions 结构中，合作者和属性是通过 PropertyValue 对象指定的。在实例化一个 bean 时，PropertyValue 对象将被用于判断 bean 之间的引用关系。

理解 IoC 的概念非常重要，因为这是 Spring 的基础之一，而且 IoC 能够令你的应用程序变得更优雅、具有更强的可配置性和可伸缩性。使用 BeanDefinitions 配置 bean 的每一个合作者和属性（更多相关的实现信息请参阅下一节）！

依赖关系的判断有些过于复杂，不适合在这里深入讨论。下面是它的基本流程：

1. 检查属性类型（可以是原始类型，如 int 或 String;集合类型，如 Map 或者 List；合作者）。合作者是其他的能被 BeanFactory 识别的 bean。

2. 假如是第一种情况（原始类型或者集合），Spring 构建一个 collection 并按照 bean definition 的定义以 PropertyValue 填充它
3. 假如是一个合作者，Spring 将构建一个 RuntimeBeanReference（运行时 Bean 引用）对象。当使用合作者的 bean 真正被实例化时，将会用到 RuntimeBeanReference 对象。随后，Spring 才真正解析合作者的实例，并将它（引用）指派给原 bean。

Bean 合作者和属性的设置可以通过两种不同的 IoC 类型完成——类型 2 和类型 3。类型 2 的实现是通过 bean 的读取器和设置器完成的，类型 3 的实现则是通过给构造方法传递初始化参数完成的。我们将使用 XmlBeanFactory 类作一个示范——正如它的名字所暗示的，使用 XML 文件来存储 bean definition。这是一个说明如何正确使用一个 BeanFactory 具体实现的最好方式。

第一个例子以 IoC 类型 2（使用设置器）方式使用 BeanFactory。下面是指定 bean definition 的部分 XML 文件片断。你同样可以通过适当的设置器声明找到真正的 bean。

```
<bean id="exampleBean" class="examples.ExampleBean">
    <property name="beanOne"><ref bean="anotherExampleBean"/></property>
    <property name="beanTwo"><ref bean="yetAnotherBean"/></property>
    <property name="integerProperty">1</property>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;
    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }
    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }
    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

正如你所看到的，声明的设置器在 XML 文件中被指定为一个相应的属性。（XML 文件中的属性直接与 RootBeanDefinition 中定义的 PropertyValues 相联系）。

然后是一个以 IoC 类型 3（使用构造器）方式使用 BeanFactory 的例子。你可以在下面的 XML 配置文件片断中发现 Spring 是如何通过指定构造器参数序列及实际的 bean 来指定构造器的。

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
    <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>
    <constructor-arg>1</constructor-arg>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

[2]For more information about IoC types, see Section 8.1, “Inversion of Control”

[3]See Section 3.4, “BeanFactory implementations” for more information about BeanFactory implementations

```
public class ExampleBean {
    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;
    public ExampleBean(AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
      this.beanOne = anotherBean;
      this.beanTwo = yetAnotherBean;
      this.i = i;
    }
}
```

正如你所见，bean definition 指定的构造器参数序列将会传递到 ExampleBean 的构造器中。

3.3.5. 自动装配合作者

Spring 具有自动装配的能力，这意味着可以让 Spring 自动检视 BeanFactory，判断 bean 之间的合作关系。自动装配功能有 4 个工作模式。“是否开启自动装配”是针对每个 bean 来指定的，因此你可以对一部分 bean 开启自动装配，对其余的 bean 则不开启。使用自动装配时，可以不必明确指定需要装配的属性或者构造器参数。

模式	解释
No	不使用自动装配功能。这是默认设置。不鼓励在大型的应用程序中改变这个设定。因为手动控制合作者能让你知道你正在干什么，同时在某种程度上，这也是为你的系统结构建立规范文档的一种好办法。
byName	使用这个选项 Spring 将会检视 BeanFactory，同时严格按名对 property 进行查找。比如，你在类 Cat 的 Bean Definition 中定义了一个名叫 dog 的合作者（当然，Cat 类必须有一个 setDog(Dog)方法），Spring 将会查找名叫 dog 的 Bean Definition 并将它作为合作者的 Bean Definition。
byType	这个选择也可以在一些 IoC 容器中看到，它赋予你按类型而不是按名对合作者进行解引用的能力。假设你拥有一个 DataSource 类型的合作者，Spring 将会查找整个 BeanFactory，试图获得一个 DataSource 类型的 bean definition 并将它作为合作者的 Bean Definition。假如没有找到或者找到多于一个同类型的 Bean Definition，BeanFactory 将会报告“自动装配失败”，同时你将不能为这个特定的 bean 使用自动装配功能。

注意：如前文所提及，不鼓励在大型的应用程序中使用自动装配，因为这将影响你的合作者类结构的透明度。

3.3.6. 依赖检查

Spring 也为你的 bean 提供了必需的依赖检查能力。当某些属性必须全部满足而你又无法提供默认值的时候（这种情况很常见），这个特性是很方便的。依赖检查可以有三种不同的方式。如同自动装配功能，每个 bean 可以自由选择是否使用依赖检查。默认值是不使用。

表格 3.4. 依赖检查模式

模式	解释
simple	只对基本类型属性和集合类型属性进行依赖检查（所有的合作者都不检查）
object	只对合作者属性进行依赖检查
all	对基本类型属性、集合类型属性及合作者属性均进行依赖检查

3.3.7. Bean 的自然特性及生命周期特性

Spring 提供了一组标记接口（marker interface）及一些其他特性，允许用户自定义 BeanFactory 管理的 bean 的自然特性和生命周期。每一个标记接口的特性和功能如下所述。

1. FactoryBean

如果一个对象希望成为它本身的工厂（提供工厂方法创建本身的实例），那么它应该实现 org.springframework.beans.factory.FactoryBean 接口。
BeanFactory 接口提供了 3 个方法

- Object getObject():返回由这个对象的工厂创建的一个实例。这个实例可以共享（取决于实例是以 singleton 模式还是 prototypes 模式创建）
- boolean isSingleton():如果这个 FactoryBean 返回的是 singleton 形态的实例则返回 true，否则返回 false
- Class getObjectType():返回 getObject()方法获得的实例的类型，如果类型未知则返回 null。

2. InitializingBean

org.springframework.beans.factory.InitializingBean 接口赋予你在 BeanFactory 完成对所有必需的属性的设置后进行初始化工作的能力。InitializingBean 接口只指定了一个方法：

- void afterPropertiesSet():在 BeanFactory 设置完所有的属性后被调用。这个方法允许你检查所需的属性是否被正确设置或者进行初始化工作。你可以抛出一个异常以指出配置失败，初始化失败，等等

注意：一般来说，完全可以避免使用 InitializingBean（有人根本就不鼓励使用 InitializingBean 接口）。beans 包在多种 beanconfiguration 存储方式之上（可以是 XML 文件，properties 文件或者是数据库）为 beandefinition 提供一种通用的初始化方法。更多关于此特性的信息请看下一节。

3. init-method

除了 InitializingBean，Spring 还提供了一种侵入性更小的办法为你的 bean 定义初始化方法。BeanFactory 的不同实现可用不同的方式指定初始化特性，但获得的结果是相同的：在设置好所有的属性及 InitializingBean 的 afterPropertiesSet()方法被调用后唤起一个无参的方法以完成初始化动作。
TODO：参考中是否应包括所有不同的初始化方法？

4. DisposableBean

org.springframework.beans.factory.DisposableBean 接口为你提供了在一个 beanfactory 实例进入析构期时使用回调方法的能力。DisposableBean 接口指定了一个方法：

- void destroy():在 beanfactory 实例析构前被调用。这个动作允许你释放在 bean 中持有的任何资源（如数据库连接）。你可以在这里抛出一个异常，但这不会停止这个 beanfactory 的析构过程。不过，至少这个异常会被日志记录下来。

注意：一般来说，完全可以避免使用 DisposableBean（有人根本不鼓励使用 DisposableBean 接口）。beans 包在多种 beanconfiguration 存储方式之上（可以是 XML 文件，properties 文件或者是数据库）为 beandefinition 提供一种通用的析构方法。更多关于此特性的信息请看下一节。

5. destroy-method

除了 Disposable，Spring 还提供了一种侵入性更小的办法为你的 bean 定义析构方法。BeanFactory 的不同实现可用不同的方式指定析构特性，但获得的结果是相同的：在设置好所有的属性及 *DisposableBean* 的 *destroy*()方法被调用后唤起一个无参的方法以完成析构动作。

TODO：参考中是否应包括所有不同的析构方法？

6. BeanFactoryAware

org.springframework.beans.factory.BeanFactoryAware 接口赋予你获得 BeanFactory——它负责管理这个“实现了 BeanFactoryAware 接口”的 bean——引用的能力。这个特性允许实现者在 BeanFactory 中查找合作者。此接口指定了一个方法：

- void setBeanFactory(BeanFactory):此方法将在初始化方法完成后（设置完属性及初始化方法完成后）被调用。

3.3.8. 客户端与 factory 的交互

客户端介面惊人的简单－－BeanFactory 接口只有 4 个方法与客户端交互：

- Object getBean(String):按名返回一个已注册 bean 的实例。取决于该 bean 在 BeanFactory 中的配置，将返回一个共享的唯一的 bean 实例或者是一个重新创建的 bean 实例。当无法找到 bean 的定义（此时将引发一个 NoSuchBeanDefinitionException 异常）或者在进行实例化和初始化 bean 时引发了一个异常，都将引起一个 BeansException 异常被抛出。
- Object getBean(String,Class)：按名返回一个已注册 bean 的实例。这个 bean 实例的 Class 类型必须和给出的 Class 类型参数相匹配，否则相应的异常将会被抛出（BeanNotOfRequiredTypeException 异常）将会被抛出。此外，这个方法的其他的行为和 getBean(String)方法相同。（请参考 getBean(String)方法）
- boolean isSingleton(String):按名检测已注册的 beandefinition 是被设定为 singleton 模式还是 prototype 模式。如果给定的 beandefinition 没有找到，则抛出一个异常（NoSuchBeanDefinitionException 异常）
- String[] getAliases(String):返回某个已配置 bean 的别名（TODO:这意味着什么？）

3.3.9. bean 的基本生命周期

这一节描述了 BeanFactory 中的 bean 的生命周期，以及在 BeanFactory 和不同类型的 bean 身上所发生事件的基本特征。

3.3.9.1 bean 的基本生命周期

Bean 的生命周期从它的 bean 定义开始，例如定义在一个 XML 文件或者一个 properties 文件中。第一步是调用 bean 的默认构造器：

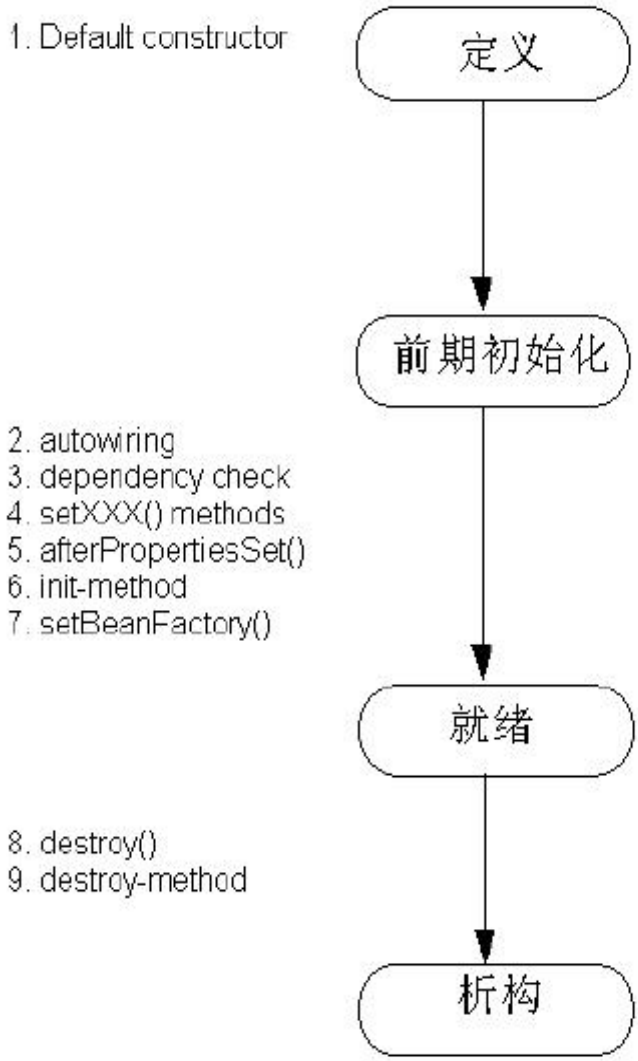
1、默认构造器
第二步是 bean 的初始化过程。完成这一步后，你的 bean 就已经准备好可以使用了。
2、自动装配过程，在这里，一切能够被自动识别的合作者都将被设置给待装配的 bean。（TODO：参考）
3、依赖检查，这意味必须保证所有的相关属性都必须得到满足（即不为 null），否则一个 UnsatisfiedDependencyException 异常将会被抛出。（TODO:参考）
4、属性设置，这意味着该 bean 在 bean 定义中（例如一个 XML 文件）定义的所有属性都将在此时得到设置。
5、afterPropertiesSet()方法被调用。这个方法是由 InitializingBean 指定的，因此只有你的 bean 实现了这个接口，此事件才会发生。（TODO:参考）
6、额外的初始化方法被调用——如果你在 bean 定义中指定了这些方法的话。（TODO:参考）
7、假如你的 bean 实现了 BeanFactoryAware 接口，setBeanFactory()方法将会在此时被调用，允许你的 bean 访问 BeanFactory（TODO:参考）

现在你的 bean 已经准备好，可以使用了。按 bean 的名字调用 BeanFactory.getBean()将会返回该 bean 的一个实例。取决于该 bean 是否处于 singleton 形态，你将获得一个共享的实例或者一个新建的实例。更多关于 singleton 的结论可在下文找到。

当 BeanFactory 进入析构期时（例如应用程序服务器停止时），BeanFactory 的析构过程将会尝试将它所持有的 bean 全部销毁。这个过程仅仅包括 *singletons 形态的 bean.*

8、假如你的 bean 实现了 DisposableBean 接口，BeanFactory 将会调用 destroy()方法
9、假如你的 bean 定义包含了析构方法的声明，这个方法也将被调用。

Bean 生命周期的一个图示：



3.4. BeanFactory 的实现

Spring 已经提供了一组现成的 BeanFactory 实现。XmlBeanFactory 类支持在 XML 文件中指定 bean 定义，ListableBeanFactory 类则支持从 properties 文件中获得 bean 定义。大部分人使用 XmlBeanFactory 类，但要自己实现“从数据库中获得 bean 定义”的 BeanFactory 也不会有多大困难。让我们先来讨论 XmlBeanFactory 类和 ListableBeanFactory 类及它们的特性。

BeanFactory 的两个主要的实现 Spring 都提供了以上所述的所有特性，如生命周期方法指定、自动装配指定等等。唯一的不同之处是配置数据（bean 定义）的存储方式。

3.4.1. 在 XML 中指定 bean 定义（XmlBeanFactory）

BeanFactory 接口的一个实现是 XmlBeanFactory 类（位于 org.springframework.beans.factory.xml 包中）。正如它的名字所告诉你的，它为你提供了在 XML 文件中指定 bean 定义的能力。Spring 提供了相应的 DTD，使校验 bean 定义 XML 文件合法性的工作变得简单些。这份 DTD 的文档描述非常详细，你能在其中找到所有你需要的东西。下面我们将简单讨论这个 XML 格式，并给出一些例子。

Spring XML bean 定义文档的根元素是<beans>。<beans>元素可以包含一个或多个 bean 定义。我们通常指定每个 bean 定义的类和属性。同时，我们也必须指定 id——这是 bean 的名字，我们将在代码中以这个名字使用这个 bean（请参阅前文有关客户端与 BeanFactory 交互的部分获得更多信息）。前文提到的初始化方法及析构方法均可指定为<bean>的 attributes。自动装配功能和依赖检查也可以作为 attributes 在同一元素中指定。此外，属性及合作者可以被指定为嵌套的<property>元素。在接下来的例子中，我们将使用 Jakarta Commons DBCP 项目中的 BasicDataSource 类。 这个类（和其他很多早已存在的类一样）可以在 Spring 轻松地使用，因为它提供了 JavaBean 风格的配置方式。此类的实例在销毁前需要调用 close 方法。这个方法通过 Spring 的“destroy-method” attribute 在 BeanFactory 中进行注册，而无需实现任何 Spring 接口（否则你就需要实现早前在“生命周期特性”一节中提到的 DisposableBean 接口）。

```
<beans>
   <bean id="myDataSource"
         class="org.apache.commons.dbcp.BasicDataSource"
         destroy-method="close">
   <!-- results in a setDriverClassName(String) call -->
        <property name="driverClassName">
           <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
           <value>jdbc:mysql://localhost:3306/mydb</value>
        </property>
        <property name="username">
           <value>root</value>
        </property>
   </bean>
</beans>
```

就像使用 destroy-method attribute 指定一个析构方法，我们可以使用 init-method attribute 指定一个初始化方法。

在 XML 文件中指定属性及合作者你必须使用嵌套的<property>元素。在前面的示例中，你已经看到内建类型（JDK 提供的类型）的属性是如何设置的；合作者的设置是通过使用嵌套的<ref>元素完成的。

```
<beans>
   ...
   <bean id="exampleDataAccessObject"
         class="example.ExampleDataAccessObject">
      <!-- results in a setDataSource(BasicDataSource) call -->
      <property name="dataSource">
         <ref bean="myDataSource"/>
      </property>
   </bean>
</beans>
```

正如你在下文中所看到的，我们在这里以一个合作者的形式使用前面例子中的 Commons DBCP datasource，用<ref bean> 元素指定它的引用。可以用三种不同的方式指定引用关系，根据指定的方式不同，Spring 可能只在当前 XML 文件中寻找合作者，也可能到别的 XML 文件中去寻找（下文将继续讨论“多个 XML 配置文件”的话题）：

- bean:同时尝试在当前 XML 文件或其他 XML 文件中查找指定的合作者 bean
- local:仅在当前 XML 文件中查找合作者。这个 attribute 是一个 XML IDREF，所以它必须存在，否则校验会失败。
- external:显式指定在其他 XML 文件中查找合作者而不搜索当前 XML 文件。在这样一组情况下，这是可能的：指定一些更复杂的属性，如 lists, properties object 和 maps。

指定多个灵活属性，像 list, properties 对象，或者 Map 会出现两种可能。下面的例子展示了这种行为：

```
<beans>
   ...
   <bean id="moreComplexObject"
         class="example.ComplexObject">
   <!-- results in a setPeople(java.util.Properties) call -->
      <property name="people">
         <props>
            <prop key="HaaryPotter">The magic property</prop>
            <prop key="JerrySeinfeld">The funny property</prop>
         </props>
      </property>
   <!-- results in a setSomeList(java.util.List) call -->
      <property name="someList">
         <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource"/>
         </list>
```

```
        </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
        <property name="someMap">
            <map>
                <entry key="yup an entry">
                    <value>just some string</value>
                </entry>
                <entry key="yup a ref">
                    <ref bean="myDataSource</ref>
                </entry>
            </map>
        </property>
    </bean>
</beans>
```

注意，Map 的每个元素（entry）可以是一个 list 或另一个 map

4. 第四章.The Context 包

4.1. 介绍

在beans包之上的是context包，它提供了管理和操作bean的一般能力。 context包在你的应用程序中扮演了一个对象注册表的角色。其功能类似J2EE应用程序服务器提供的enterprise naming context特性，然而有几个重要的不同之处。
例如，对象的绑定在Spring的application context中不需要额外的远程接口或对象。其次，application contexts可以在你的应用程序中的各个层面使用，从基于Swing的客户端到Web应用程序，除此以外，也能使用在EJB环境中。

Context包的基础是org.springframework.context.ApplicationContext接口，提供了国际化、事件处理及beans在context中的自查能力。它也可创建具有层次结构context环境，将bean的作用域和可访问区域限制在应用程序的一个特定部分中。

4.2. 由 ApplicationContext 管理的 bean

ApplicationContext 包括了 bean factory 所有的功能。这意味着你可以使用 XmlBeanFactory 定制 bean 并且通过指定它们的名字或 ID 查找 bean。对于 BeanFactory 提供给用户所有的特性，包括生命周期接口、初始化方法、析构方法、依赖关系检测及自动装配等，ApplicationContext 也同样提供。BeanFactory 将不会在这章讨论，取而代之的是 ApplicationContext 独有的特性。在前面的章节中，有更多关于 BeanFactory 的信息。

4.3. ApplicationContext 基础

ApplicationContext 提供了——参见上文

5. 第五章.校验和数据绑定

5.1. 介绍

有一个重要的(Big)问题：是否应该在验证时考虑业务逻辑。两种选择都有自己各自的优点和缺点。而 spring 为验证（和数据绑定）提供了一种不必排斥这两者中任何一个的设计。验证不应该被特定限制于网络层，它应该很容易定位并且应该可以有效的插入任何验证。考虑到上边的因素，Spring 提供了一个基本的 Validator 接口，你可以在应用程序的任何一层使用它。

数据绑定的作用在于：可以将用户输入动态地绑定到应用程序的领域模型（或者任何用于处理用户输入的对象）。Spring 提供了一个所谓的”DataBinder”正是为了满足这种需要。Validator 和 DataBinder 组成了 validation 包，这个包主要应用在 MVC 框架里。

5.2. 使用 DataBinder 来绑定数据

DataBinder 建立在 BeanWrapper 的基础之上。

6. 第六章.web framework

6.1. web framework 简介

Spring 的 web 框架是围绕着 DispatcherServlet 来设计的，通过可配置的处理器映射（handler mappings），视图解析（view resolution），本地化和主题解析（theme resolution）以及对文件上载的支持，DispatcherServlet 将请求分发给处理器。缺省的处理器是一个非常简单的 Controller 接口，只是提供一个 ModelAndView handleRequest(request，response)方法。这已经能够用于应用控制器（application controller）上，但是你将会更喜欢包含了实现的层次体系，例如包括 AbstractController, AbstractCommandController 和 SimpleFormController。

通常，应用控制器将会是这些类的子类。注意：你可以选择一个合适的基类：如果你没有表单（form），你便不需要 FormContrller。这是与 Struts 的一个主要不同之处。你可以将任何对象作为 command 或 form 对象：不需要实现一个接口或从一个基类进行继承。

Spring 的数据绑定（data binding）是非常灵活的，例如，它将类型失配（type mismatches）作为能够为应用程序所评估的验证错误（validation errors），而不非系统错误（system errors）。从而，你不需要在 form 对象中将你业务对象的属性作为 String 进行复制，只需能够处理无效的提交

或正确转换字符串即可。相反，直接绑定到你的业务对象往往更为可取。这是另外一个与 Struts 的主要不同点，Struts 是围绕着所必需的基类进行构建的，如 Action 或 Action Form－对于每种 action 而言。

与 Webwork 相比，Spring 具有更多不同的对象角色：它支持一个 Controller，一个可选的 command 或 form 对象，以及一个能够被传递给视图的模型（model）的概念。Model 通常会包含 command 或 form 对象，而且还可以是任意的引用数据（reference data）。相反，Webwork 的 Action 则将所有这些角色结合到一个单独的对象中。WebWork 不允许你使用已有的业务对象来作为你 form 的一部分，而只能将它们作为各自 Action 类的 bean 属性。最终，相同的处理请求的 Action 实例可被用于评估和视图中的表单分布（form population）。因此，同样需要将引用数据建模成 Action 的 bean 属性。这种在一个对象中具有过多角色的作法尚存在着争议。

关于视图：Spring 的视图解析极为灵活。Controller 的实现甚至可以直接将视图写到响应中去，返回 ModelAndView 为 null。在正常情况下，ModelAndView 实例由视图名称和模型 Map 组成，包括 bean 名称以及相应的对象（如 command 或 form，引用数据等）。视图的名称解析具有非常高的可配置性，既可以通过 bean 名，或者通过属性文件，也可以通过你自己的 ViewResolver 实现。抽象的模型 Map 涉及到了视图技术的全部抽象，无需任何的争论：JSP，Velocity，或任何其他的东西－每一种渲染器（render）都能够被直接进行集成。模型 Map 可被简单地转换成一种适当的格式，如 JSP 的 request 属性或 Velocity 的 template model。

6.1.1. MVC 实现插入能力

许多团队试图促进他们在技术和工具方面的投入，以用于既有项目和新项目。实际上，不但有大量关于 Struts 的书籍和工具，而且有大量富有经验的 Struts 开发人员。因此，如果你能够忍受 Struts 在架构上的缺陷的话，它依然是 web 层的一个可行的选择。这同样适用于 WebWork 和其他的 web 框架。

如果你不想使用 Spring 的 web MVC，而是倾向于采用 Spring 提供的其他解决方案，你可以轻松地将你所选择的 web 框架与 Spring 进行集成。仅仅需要通过 ContextLoaderListener 开始一个 Spring 的 root application context，并且通过其 ServletContext 属性（或 Spring 各自的 helper 方法）从 Struts 或 WebWork 的 action 内部来访问它。

注意：没有任何相关的“插件（plugins）”，便不需要进行专用的集成：从 web 层的角度来看，你只是将 Spring 用作是一个库（library），并且将 root applicaiton context 作为入口点（entry point）。所有你注册的 bean 以及所有 Spring 的服务（service）便可信手拈，甚至不需要 Spring 的 web MVC。Spring 并不是在这种用途上与 Struts 和 WebWork 进行竞争，它只是涉及到了纯 we 框架所不具有的多个方面，从 bean 配置到数据访问和事务处理。从而，你能够通过 Spring 的中间层和（或）数据访问层来丰富你的应用，即使你只是想使用与 JDBC 或 Hibernate 相关的事务抽象（transaction abstraction）。

6.1.2. Spring MVC 特性

如果只关注 web 支持，Spring 所独有的一些特性为：

- 明确的角色分工：控制器－验证器（validator）－Command 对象－Form 对象－模型对象，DispatcherServlet－处理器映射－视图解析器（view resolver），等等。

- 框架和应用程序类可以作为 JavaBean 进行强大和直接的配置，包括通过 application context 进行方便的中间引用（in-between referencing），比如从 web 控制器到业务对象和验证器。

- 适应性，抗干扰性：对于给定的情节，你可以使用任何需要的 Controller 子类（普通对象，Command，Form，Wizard，Multi Action，或定制对象），而非所有事情都要从 Action/ActionForm 继承。

- 可复用的业务代码，无需复制：你可将已有的业务对象用作 Command 或 Form 对象，而不是将它们映射为特定的 ActionFrom 子类。

- 可定制的绑定和验证：可以将类型失配作为应用级别的验证错误，其能够保持非法数值（offending value），本地化日期和数字的绑定，等等，而不是需要手工解析的纯字符形式的 form 对象以及向业务对象的转换。

- 可定制的处理器映射和视图解析：灵活的模型传递（通过 name/value Map），处理器映射和视图解析策略由简到繁，而非单一方式。

- 可定制的本地化和主题解析，支持使用和不使用 Spring 标签库的 JSP，支持 JSTL，支持 Velocity 而无需额外的桥接部分（bridge），等等。

- 简单而强大的标签库能够避免产生任何的 HTML，使得在标记代码方面具有最大限度的灵活性。

6.2. The DispatcherServlet

像许多其他的 web 框架一样，Spring 的 Web 框架是围绕着 Servlet 来进行设计的，该 servlet 负责分发请求给控制器和提供其他功能以促进 web 应用的开发的。然而，Spring 的 DispatcherServlet 做的比这些要多。它完全与 Spring 的 ApplicationContext 相集成并充分地进行使用。

在你 web 应用的 web.xml 中声明了 Servlet，即 DispatcherServlet。你必须对 DispatcherServlet 所要处理的请求进行映射，在同一 web.xml 文件中使用 url-mapping 来配置。

```
<web-app>
    ...
    <servlet>
       <servlet-name>example</servlet-name>
       <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
       <load-on-startup>1</load-on-startup>
    </servlet>
```

```
    <servlet-mapping>
      <servlet-name>example</servlet-name>
      <url-pattern>*.form</url-pattern>
    </servlet-mapping>
</web-app>
```

在上面的例子中，DsipatcherServlet 将会处理所有以.form 结尾的请求。然后，需要配置 DispatcherServlet。像第四章－Context 包－所演示的那样，Spring 中的 ApplicationContext 是有作用域的。在 Web 框架中，每个 DispatcherServlet 有其自己的 WebApplicationContext，其包含了 DispatcherServlet 的配置 bean。DispatcherServlet 所使用的缺省 BeanFactory 为 XMLBeanFactory，并且 DispatcherServlet 在初始化时将会在你 web 应用的 WEB－INF 目录下查找名为[servlet－name]－servlet.xml 的文件。DispatcherServlet 所用到的缺省值可以使用 Servlet 的初始化参数进行修改(详细信息可参见下文)。

WebApplicationContext 只是一个普通的 ApplicationContext，其具有 web 应用所必需的一些额外特性。它不同于一般的 ApplicationContext，其具有处理主题（theme）的能力(可参见???)以及知道与哪个 servlet 相关联（具有指向 ServletContext 的 link）。WebApplicationContext 被绑定在 ServletContext 中，你需要时总是可以使用 RequestContextUtils 来查找到 WebApplicationContext。

为了能处理请求和呈现（render）适当的视图，Spring 的 DispatcherServlet 有一组自己专用的 bean。这些 Bean 包含在 Spring 的框架之中，并且（可选地）必须在 WebApplicationContext 中进行配置，就像必须要配置任何其他的 bean 一样。随后给出这些每个 Bean 的更详细描述。现在，我们只是提及它们，只是想让你知道它们的存在，从而使我们能够继续来谈论 DispatcherServlet。对于大多数的 Bean，都提供了缺省实现，因此你不必担心这些。

Table 6.1. Special beans in the WebApplicationContext 表

Expression	Explanation
handler mapping(s)	处理器映射 6.3 节 处理器映射 预/后处理器和控制器的列表，如果它们符合特定的标准便将会被执行（例如一个与控制器相匹配的特定的 URL）
view resolver	视图解析器 6.4 节 视图与视图解析 能够解析视图名称，并且 DispatcherServlet 需要它来一起解析这些视图
locale resolver	本地化解析器 6.5 节 使用本地化 为了能提供一个国际化的视图而解决 client 本地化的能力
theme resolver	主题解析器 6.6 节 使用主题 能够解析你 web 应用所使用的主题，例如提供个性化的布局设计。
multipart resolver	multipart 解析器 6.7 节 Spring 的 multipart(文件上载支持) 提供处理从 HTM 表单上载文件的功能

当一个 DispatcherServlet 被配置好以供使用，并且针对该特定 DispacherServlet 的一个请求进来时，其便开始处理它。下面列表描述了请求在被 DispatcherServlet 进行处理时所经历的完整的过程：

1、 索 WebApplicationContext 并将其作为属性绑定到请求中，以便于控制器和处理链中的其他元素使用。缺省情况下，它被绑定在 DispatcherServlet 的 WEB_APPLICATION_CONTEXT_ATTRIBUTE 键值上。
2、 本地化解析器被绑定到请求中，使得在请求处理（呈现视图，准备数据等等）时位于链中解决本地化的元素来使用。如果你没有使用解析器，它不会影响任何事物，因此，如果你不需要本地化的解析，便无需为此操心。
3、 主题解析器被绑定到请求中，可由视图来决定使用哪些主题（如果你不需要主题，则不必管它，如果你不用它，解析器也只是被进行绑定而不会影响到任何东西）。
4、 如果指定了 multipart 解析器，则需要对请求进行 multipart 检查，如果这样的话，它会被封装到 MultipartHttpServletRequest 中，使得链中的其他元素作进一步的处理（更多关于分段式处理的内容请参见下文）。
5、 搜索适当的处理器。如果一个处理器被绑定，与处理器（预处理器，后处理器，控制器）相关的执行链将会次序执行以准备出一个 model。
6、 如果返回一个 model，使用由 WebApplicationContext 所配置的视图解析器来呈现视图。如果没有 model 返回（可能是因为实例安全的原因预处理器或后处理器截获了请求），也便不会呈现视图，因为请求可能已经完成。

Spring 的 DispatcherServlet 也支持最后修改日期的返回，正如 Servlet API 所说明的那样。确定一个特定请求的最后修改日期的处理非常简单。DispatcherServlet 将首先查找适当的处理器映射并且测试处理器是否匹配 LastModied 接口的实现，如果是这样，客户端会得到 long getLastModified(request)的返回值。

需要完成：Servlet 的初始化参数

6.3. 处理器映射

6.4. 视图和解析视图

6.5. 使用本地化

6.6. 使用主题

6.7. Spring 的文件上传支持

6.7.1. 简介

Spring 内建了对 multipart 支持来处理 web 应用中的文件上载。设计支持 multipart 是以一种可插拔方式（使用所谓的 MultipartResovlers）来完成的。有新意的是，Spring 提供了结合使用 Commons 的 FileUpload 和 COS 的 FileUpload 组件的 MultipartResolver。如何支持文件的上传将会

在本章其余部分予以描述。

已经说过，multipart 的支持是通过 MultipartResolver 接口来提供的，其位于 org.springframework.web.multipart 包内。缺省情况下，Spring 并不支持 multipart 的处理。你必须自己通过添加一个 multipart 解析器到 web 应用的 context 中来激活它。你这样做完之后，每个请求将会检查可能包含的 Mutltipart 信息。如果没有发现这样的 Multipart 信息，请求将按照预期的方式继续处理。但如果在请求中发现 multipart，你在 context 中声明的 MultipartResolver 将会进行解析。然后，你请求中的 multipart 属性将会被作为其他属性一样看待。

6.7.2. 使用 MultipartResolver

为了能够解析来自请求中的 Multipart，你将必须声明一个 MultipartResolver。Spring 提供了两种 Multipart 解析器。第一种是与 Commons FileUpload（http://jakarta.apache.org/commons/fileupload）一起工作的解析器，第二种是使用 O'Reilly COS 包（http://www.servlets.com/cos）的解析器。如果你未在 web 应用的 context 中声明 MultipartResolver，那么将不会对 Multipart 进行检测，因为有些开发者可能发觉他们需要自己来解析出 Multipart 信息。

下边的例子展示了如何来使用 CommonsMultipartResolver：

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
<!-- one of the properties available; the maximum file size in bytes -->
    <property name="maximumFileSize">
        <value>100000</value>
    </property>
</bean>
```

然而你也可以使用 CosMultipartResolver：

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.cos.CosMultipartResolver">
<!-- one of the properties available; the maximum file size in bytes -->
    <property name="maximumFileSize">
        <value>100000</value>
    </property>
</bean>
```

当然，如果你要使用一个 multipartresolver，那么你需要在你的 classpath 中添加适当的 jar 文件。在使用 CommonsMultipartResolver 时，你需要使用 commons-fileupload.jar，在使用 CosMultipartResolver 时，你需要使用 cos.jar。

现在，设置 Spring 来处理 Multipart 请求，让我们谈一下如何实际地来使用它。当 Spring 的 DispatcherServlet 检测到一个 Multipart 请求，它会激活你 context 中所声明的解析器并传递该请求。其所做的主要的工作是将当前的 HttpServletRequest 包装成支持 Multipart 的 MultipartHttpServletRequest。使用 MultipartHttpServletRequest，你便可得到包含在该请求中的 Multipart 信息，并在你的控制器中得到实际的 Multipart 信息本身。

6.7.3. 处理表单中的文件上载

MultipartResolver 完成其工作后，请求将会像其他请求一样被继续进行处理。因此实际上，你可以创建一个带有表单上载字段（form upload field）的表单，并让 Spring 把文件绑定到你的表单上。像任何其他属性一样，无法自动地将其转换为 String 或基本类型，为了能将二进制的数据放到你的 bean 中，你就必须通过 ServletRequestDatabinder 来注册一个定制的编辑器。有多个编辑器可用来处理文件以及设置结果到 bean 上。StringMultipartEditor 能够将文件转换成字符串（使用一种用户自定义的字符集）以及 ByteArrayMultipartEditor 能够将文件转换成字节数组。它们的功能，例如 CustomDateEditor，就是能够使用 web 站点内的表单来上载文件，声明一个解析器，一个处理 bean 的控制器的 url 映射以及控制器本身。

```
<beans>
...

  <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
  <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/upload.form">fileUploadController</prop>
      </props>
    </property>
  </bean>
  <bean id="fileUploadController" class="examples.FileUploadController">
    <property name="commandClass"><value>examples.FileUploadBean</value></property>
    <property name="formView"><value>fileuploadform</value></property>
    <property name="successView"><value>confirmation</value></property>
  </bean>
</beans>
```

此后，创建控制器和实际的包含文件属性的 bean。

```
// snippet from FileUploadController
public class FileUploadController extends SimpleFormController {


   protected ModelAndView onSubmit(
```

```
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors)
        throws ServletException, IOException {

        // cast the bean
        FileUploadBean bean = (FileUploadBean)command;

        // let's see if there's content there
        byte[] file = bean.getFile();

        if (file == null) {
             // hmm, that's strange, the user did not upload anything
        }

        // well, let's do nothing with the bean for now and return:
        return super.onSubmit(request, response, command, errors);
    }
    protected void initBinder(
        HttpServletRequest request,
        ServletRequestDataBinder binder)
        throws ServletException {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor (in this case the
        // ByteArrayMultipartEditor
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }
}

// snippet from FileUploadBean
public class FileUploadBean {
        private byte[] file;

        public void setFile(byte[] file) {
               this.file = file;
        }

        public byte[] getFile() {
               return file;
        }
}
```

正如你所见，FileUploadBean 具有一个保持文件的 byte[]类型的属性。控制器注册一个定制的编辑器以使 Spring 知道如何将解析器发现的 multipart 对象实际转换成该 bean 所指定的属性。现在，还没有对 byte[]以及该 bean 本身进行任何的处理，但是你可以对它做任何你想要做的事情（将其保存到数据库中，将其 mail 给其他人，等等）。但我们还没有结束。事实上，为了让用户能够实际进行某些上载工作，我们还要创建表单：

```
<html>
   <head>
      <title>Upload a file please</title>
   </head>
   <body>
      <h1>Please upload a file</h1>
      <form method="post" action="upload.form" enctype="multipart/form-data">
         <input type="file" name="file"/>
         <input type="submit"/>
      </form>
   </body>
</html>
```

正如你所见，我们以及创建了一个根据「保持 byte[]的 bean」命名的字段，按 bean 中 byte[]属性的名字来命名。此外，我们添加了编码属性，该属性是让浏览器知道如何对 Multipart 字段进行编码所必需的（不要忘记！）。现在一切皆可工作。

7. 第七章.与第三方软件集成

7.1. 介绍

Spring 是一个面向所有层的应用程序框架：它提供了 bean 式配置基础, AOP 支持，JDBC 的概要框架，抽象事务支持等。他实现了一个真正无干扰(non-intrusive))的效果:非必要情况下，你的程序里的类不需要依赖于任何的 Spring 类，并且如果你喜欢，你可以单独重用它的每一个部分。由于其特别的设计，这个框架鼓励将各层清楚的分离，其中最重要就是将网络层和业务逻辑分离：例如验证框架不需要依赖于网络控制器(web controller)。这样做的最主要的目标就是为了提高重用性和可测试性。不需要依赖容器或者框架可以避免极大的麻烦。

Spring 可以算是一个一站式商店（one-stop shop，见译文参考 1）。它关注典型应用程序的大部分基础设施。然而，它并不是要代替其它的框架，如果你希望在你的程序中的某一层使用其他的技术（比如在网络层或持久层）Spring 允许你将它为特定层所提供的默认解决方案替换为任意的其他解决方案。Spring 提供了几个已经被预先集成了的技术。我们将在这一章介绍这一部分的知识。

7.2. 与 Velocity 集成

Velocity 是 Jakarta 项目下的一种 view 技术。关于 Velocity 的更多信息可以在 http://jakarta.apache.org/velocity 找到。这一章将讲述怎样将 Velocity 集成到 Spring 里去

7.2.1. 依赖

在使用 Velocity 之前，你的网络应用程序需要先满足它的一个依赖条件：必须保证 velocity-1.x.x.jar 包有效。在典型的情况下，这个包含在 WEB-INF/lib 文件夹里，这样就可以保证能被 J2EE 服务器找到并且为你的程序添加到类路径里。当然，我们还假定你的 WEB/lib 文件夹里已经有了 spring-full.jar 的包。（或者是其等价物）。Velocity 最新的稳定版本通常已经被作为 Spring 框架的一部分提供了，你可以从那里拷贝。

7.2.2. Dispatcher Servlet Context

你的 Spring dispatcher servlet 所对应的配置文件应该已经包含针对 view resolver 的 bean 式定义。我们还要在这里添加一个对 Velocity 环境进行配置的 bean。我把我的 dispatcher 命名为’frontcontroller’,我下边配置文件将按照这个名字进行配置。

下边的代码例子展示了各种带有相应注释的配置文件。

```
<!-- ============================================================-->
<!-- View resolver. Required by web framework. -->
<!-- ============================================================-->
<!--
     View resolvers can be configured with ResourceBundles or XML files. If you need
     different view resolving based on Locale, you have to use the resource bundle resolver,
     otherwise it makes no difference. I simply prefer to keep the Spring configs and
     contexts in XML. See Spring documentation for more info.
-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
<property name="cache"><value>true</value></property>
<property name="location"><value>/WEB-INF/frontcontroller-views.xml</value></property>
</bean>
<!-- ============================================================-->
<!-- Velocity configurer. -->
<!-- ============================================================-->
<!--
   The next bean sets up the Velocity environment for us based on a properties file, the
   location of which is supplied here and set on the bean in the normal way. My example shows
   that the bean will expect to find our velocity.properties file in the root of the
   WEB-INF folder. In fact, since this is the default location, it's not necessary for me
   to supply it here. Another possibility would be to specify all of the velocity
```

```
    properties here in a property set called "velocityProperties" and dispense with the
    actual velocity.properties file altogether.
-->
<bean
    id="velocityConfig"
    class="org.springframework.web.servlet.view.velocity.VelocityConfigurer"
    singleton="true">
       <property name="configLocation"><value>/WEB-INF/velocity.properties</value></property>
</bean>
```

7.2.3. Velocity.properties

这个文件包含了为了让 velocity 能够在运行时对自己进行配置而需要传递给 Velocity 的值。虽然实际上仅需要很少的一部分属性，但是也还有许多其他有效的可选属性。-要获取更多的信息请查看 Velocity 的文档。这里，我仅仅演示一下一个能让 Velocity 跑起来，并且在你的 Spring MVC 程序里运行的最小例子。

最重要的属性是那些用于 Velocity 模版定位的部分。Velocity 模版既可以从类路径里加载也可以从文件系统中加载。但两者都有自己各自的优缺点。从类路径加载具有完整的可移植性并且可以工作在所有的服务器下，但你可能会发现那些模版把你的 java 包们（java packages）弄得一团糟（除非你为它们新创一棵源码树）。使用类路径存储的一个更糟糕的缺点是：在开发阶段，在源码树里的任何改变常常会导致刷新在 WEB-INF/classes 树里的资源，而这又会反过来影响你服务器，导致它重启程序（代码热部署）这将会让人觉得很不爽。但是，一旦完成了大部分的开发工作了以后，你就可以将模版都存入一个 jar 文件。如果你将这个 jar 文件放在 WEB-INF/lib 目录下,他们就会对程序有效。

这个例子将 velocity 模版存储在文件系统中 WEB-INF 下的某个位置，这样他们就不会直接对客户端浏览器有效，但不会在开发过程中因为修改而导致程序重启。这种做法的缺点就是目标服务器可能不能正确的解决这些文件的路径问题。特别是当你的服务器不能 explode(打开)文件系统的 WAR 组件时。不过这种文件加载方法在在 Tomcat4.1.x, WebSphere 4.x and WebSphere 5.x.都可以很好的工作。你可以自由选择喜欢的加载方式（Your mileage may vary.）

```
#
# velocity.properties - example configuration
#
# uncomment the next two lines to load templates from the
# classpath (WEB-INF/classes)
#resource.loader=class
#class.resource.loader.class=org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
# comment the next two lines to stop loading templates from the
# file system
```

```
resource.loader=file
file.resource.loader.class=org.apache.velocity.runtime.resource.loader.FileResourceLoader
# additional config for file system loader only.. tell Velocity where the root
# directory is for template loading. You can define multiple root directories
# if you wish, I just use the one here. See the text below for a note about
# the ${webapp.root}
file.resource.loader.path=${webapp.root}/WEB-INF/velocity
# caching should be 'true' in production systems, 'false' is a development
# setting only. Change to 'class.resource.loader.cache=false' for classpath
# loading
file.resource.loader.cache=false
# override default logging to direct velocity messages
# to our application log for example. Assumes you have
# defined a log4j.properties file
runtime.log.logsystem.log4j.category=com.mycompany.myapplication
```

上边的资源文件装载配置使用了一个标记来表示网络应用程序的根目录。这个标记将会在属性值被提供给 Velocity 之前被转化成特定操作系统的实际的路径。这将令你的文件资源装载器不可移植。如果你认为为 VelocityConfigurer 定义一个不同的“appRootMarker”很重要，你可以改变标记的实际的名字。要了解实现这种做法的细节，你可以查看 Spring 的文档。

7.2.4. View 配置

配置的最后一步是定义那些将在 velocity 模版中出现的 views。views 总是以一致的方式在 Spring 的环境文件中定义。就如前边提到的，这个例子使用 XML 文件来定义 view beans,但也可以使用资源文件(ResourceBundle)。view 的名字被定义在我们前面的 ViewResolver bean 里 -作为 WEB-INF/frontcontroller-servlet.xml 文件的一部分

```
<!--
   Views can be hierarchical, here's an example of a parent view that
   simply defines the class to use and sets a default template which
   will normally be overridden in child definitions.
-->
<bean id="parentVelocityView"
class="org.springframework.web.servlet.view.velocity.VelocityView">
   <property name="templateName"><value>mainTemplate.vm</value></property>
</bean>
<!--
  -  The main view for the home page. Since we don't set a template name, the value
     from the parent is used.
-->
<bean id="welcomeView" parent="parentVelocityView">
```

```
    <property name="attributes">
      <props>
        <prop key="title">My Velocity Home Page</prop>
      </props>
    </property>
</bean>
<!--
  - Another view - this one defines a different velocity template.
-->
<bean id="secondaryView" parent="parentVelocityView">
    <property name="templateName"><value>secondaryTemplate.vm</value></property>
    <property name="attributes">
      <props>
        <prop key="title">My Velocity Secondary Page</prop>
      </props>
    </property>
</bean>
```

7.2.5. 创建模版并且测试

最后，你只需要简单地创建实际的模版。我们已经定义了涉及到两个模版的视图，mainTemplate.vm 和 secondaryTemplate.vm 这两个文件要和上边 velocity.properties 文件中提到的一样的都必须在/WEB-INF/velocity/ 目录下。如果你在 velocity.properties 中选择类路径的加载方式，这些文件就要在缺省的的包里（WEB-INF/classes），或者在 WEB-INF/lib 目录下的一个 jar 文件中。我们的 secondaryView 看起来可能象：

```
## $title is set in the view definition file for this view.
<html>
    <head><title>$title</title></head>
    <body>
        <h1>This is $title!!</h1>
    </body>
</html>
```

现在，当你的控制器返回一个带有”secondaryView”的 ModelAndView 作为视图来显示。Velocity 将会跳回到上面那页。最后，简单地总结一下：下边这个是在上边所讨论的例子程序的文件的树型结构。仅显示了一部分，某些必须的目录在这里被加亮了。Velocity views 不能工作的最常见原因是文件放置位置错误，其次就是错误的属性定义。

```
ProjectRoot
       |
```

```
+- WebContent
|
+- WEB-INF
     |
     +- lib
     |    |
     |    +- velocity-1.3.1.jar
     |    +- spring-full-1.0M1.jar
     |
     +- velocity
     |      |
     |      +- mainTemplate.vm
     |      +- secondaryTemplate.vm
     |
     +- frontcontroller-servlet.xml
     +- frontcontroller-views.xml
     +- velocity.properties
```

8. 第八章.Background articles

8.1. Inversion of Control

This is supposed to be a background article about Inversion of Control in order to give users a better feeling of
why they actually should use IoC in general and Spring specifically.

Model-View-Controller pattern (MVC)

This is supposed to be a background article about Model View Controller.

9. 结束语

Spring reference 中文版由 Spring 中文论坛组织，经过近 20 天翻译小组的不懈努力终于完成了。感谢翻译团队的辛苦工作和每一个关注我们支持我们的 Spring 爱好者！

9.1. 项目手记

A. 根据Spring开发小组的要求，发起Spring-reference中文版计划(28 Nov 2003)
   确定项目目标,人员,方式(29 Nov 2003)
   详细内容：http://xglw.51.net/5team/springframework/viewtopic.php?t=102
B. 分析任务、制定计划书和Task sheet(30 Nov 2003)
   详细内容：http://xglw.51.net/5team/springframework/viewtopic.php?t=106
C. 任务执行 Phase I (Dec 1,2003 ~Dec 3,2003)
D. 任务执行 Phase II (Dec 3,2003 ~Dec 8,2003)
E. 任务执行 Phase III (Dec 10,2003 ~Dec 20,2003)
F. 复审和反馈(Dec 18,2003~Dec 22,2003)
   详细内容：http://xglw.51.net/5team/springframework/viewtopic.php?t=166
G. 调整(Dec 22,2003~Dec 23,2003)
H. 交付(Dec 24,2003)

BTW:在项目过程中沟通的不顺利导致项目延期13 天，这将是我们宝贵的经验和教训。

December 25, the day on which this feast is celebrated. Launch Date is Christmas Eve. As Christmas-box ,Merry Christmas to every Spring fans. – yanger

9.2. 翻译小组成员：

发起：yanger
译者:thinking(第一、二章)，无明(第三、四章)，yoshiyan(第五、七章)，victor_jan(第六章)
校审:gigix,outmyth

更多Springframework资料请访问Spring 中文论坛 http://xglw.51.net/5team/springframework。