北華大學

毕业设计外文资料翻译

（译文）

题目名称： 图书馆管理系统的设计与实现

学　　院： 计算机科学技术

专业年级： 计算机科学与技术 15 级

学生姓名： 赵海铭

班级学号： 2 班 13 号

指导教师： 孙　海

二〇一八 年 十二 月 二十八 日

译文题目： Spring 框架介绍

原文题目： Introducing the Spring Framework

原文出处：

http://www.west.cn/info/html/chengxusheji/Javajishu/20080226/49503.html

Introducing the Spring Framework

The Spring Framework: a popular open source application framework that addresses many of the issues outlined in this book. This chapter will introduce the basic ideas of Spring and dis-cuss the central “bean factory” lightweight Inversion-of-Control (IoC) container in detail.

Spring makes it particularly easy to implement lightweight, yet extensible, J2EE archi-tectures. It provides an out-of-the-box implementation of the fundamental architectural building blocks we recommend. Spring provides a consistent way of structuring your applications, and provides numerous middle tier features that can make J2EE development significantly easier and more flexible than in traditional approaches.

The basic motivations for Spring are:

To address areas not well served by other frameworks. There are numerous good solutions to specific areas of J2EE infrastructure: web frameworks, persistence solutions, remoting tools, and so on. However, integrating these tools into a comprehensive architecture can involve significant effort, and can become a burden. Spring aims to provide an end-to-end solution, integrating spe-cialized frameworks into a coherent overall infrastructure. Spring also addresses some areas that other frameworks don’t. For example, few frameworks address generic transaction management, data access object implementation, and gluing all those things together into an application, while still allowing for best-of-breed choice in each area. Hence we term Spring an application framework, rather than a web framework, IoC or AOP framework, or even middle tier framework.

To allow for easy adoption. A framework should be cleanly layered, allowing the use of indi-vidual features without imposing a whole world view on the application. Many Spring features, such as the JDBC abstraction layer or Hibernate integration, can be used in a library style or as part of the Spring end-to-end solution.

To deliver ease of use. As we’ve noted, J2EE out of the box is relatively hard to use to solve many common problems. A good infrastructure framework should make simple tasks simple to achieve, without forcing tradeoffs for future complex requirements (like distributed transactions) on the application developer. It should allow developers to leverage J2EE services such as JTA where appropriate, but to avoid dependence on them in cases when they are unnecessarily complex.

To make it easier to apply best practices. Spring aims to reduce the cost of

adhering to best practices such as programming to interfaces, rather than classes, almost to zero. However, it leaves the choice of architectural style to the developer.

Non-invasiveness. Application objects should have minimal dependence on the framework. If leveraging a specific Spring feature, an object should depend only on that particular feature, whether by implementing a callback interface or using the framework as a class library. IoC and AOP are the key enabling technologies for avoiding framework dependence.

Consistent configuration. A good infrastructure framework should keep application configuration flexible and consistent, avoiding the need for custom singletons and factories. A single style should be applicable to all configuration needs, from the middle tier to web controllers.

Ease of testing. Testing either whole applications or individual application classes in unit tests should be as easy as possible. Replacing resources or application objects with mock objects should be straightforward.

To allow for extensibility. Because Spring is itself based on interfaces, rather than classes, it is easy to extend or customize it. Many Spring components use strategy interfaces, allowing easy customization.

A Layered Application Framework

Chapter 6 introduced the Spring Framework as a lightweight container, competing with IoC containers such as PicoContainer. While the Spring lightweight container for JavaBeans is a core concept, this is just the foundation for a solution for all middleware layers.

Basic Building Blocks

pring is a full-featured application framework that can be leveraged at many levels. It consists of multi-ple sub-frameworks that are fairly independent but still integrate closely into a one-stop shop, if desired. The key areas are:

Bean factory. The Spring lightweight IoC container, capable of configuring and wiring up Java-Beans and most plain Java objects, removing the need for custom singletons and ad hoc configura-tion. Various out-of-the-box implementations include an XML-based bean factory. The lightweight IoC container and its Dependency Injection capabilities will be the main focus of this chapter.

Application context. A Spring application context extends the bean factory concept by adding support for message sources and resource loading, and providing hooks into existing environ-ments. Various out-of-the-box implementations include standalone application contexts and an XML-based web application context.

AOP framework. The Spring AOP framework provides AOP support for method interception on any class managed by a Spring lightweight container. It supports easy proxying of beans in a bean factory, seamlessly weaving in interceptors and other advice at runtime. Chapter 8 dis-cusses the Spring AOP framework in detail. The main use of the Spring AOP framework is to provide declarative enterprise services for POJOs.

Auto-proxying. Spring provides a higher level of abstraction over the AOP framework and low-level services, which offers similar ease-of-use to .NET within a J2EE context. In particular, the provision of declarative enterprise services can be driven by source-level metadata.

Transaction management. Spring provides a generic transaction management infrastructure, with pluggable transaction strategies (such as JTA and JDBC) and various means for demarcat-ing transactions in applications. Chapter 9 discusses its rationale and the power and flexibility that it offers.

DAO abstraction. Spring defines a set of generic data access exceptions that can be used for cre-ating generic DAO interfaces that throw meaningful exceptions independent of the underlying persistence mechanism. Chapter 10 illustrates the Spring support for DAOs in more detail, examining JDBC, JDO, and Hibernate as implementation strategies.

JDBC support. Spring offers two levels of JDBC abstraction that significantly ease the effort of writing JDBC-based DAOs: the org.springframework.jdbc.core package (a template/

callback approach) and the org.springframework.jdbc.object package (modeling RDBMS operations as reusable objects). Using the Spring JDBC packages can deliver much greater pro-ductivity and eliminate the potential for common errors such as leaked connections, compared with direct use of JDBC. The Spring JDBC abstraction integrates with the transaction and DAO abstractions.

Integration with O/R mapping tools. Spring provides support classes for O/R Mapping tools like Hibernate, JDO, and iBATIS Database Layer to simplify resource setup, acquisition, and release, and to integrate with the overall transaction and DAO abstractions. These integration packages allow applications to dispense with custom ThreadLocal sessions and native transac-tion handling, regardless of the underlying O/R mapping approach they work with.

Web MVC framework. Spring provides a clean implementation of web MVC, consistent with the JavaBean configuration approach. The Spring web framework enables web controllers to be configured within an IoC container, eliminating the need to write any custom code to access business layer services. It provides a generic

DispatcherServlet and out-of-the-box controller classes for command and form handling. Request-to-controller mapping, view resolution, locale resolution and other important services are all pluggable, making the framework highly extensi-ble. The web framework is designed to work not only with JSP, but with any view technology, such as Velocity—without the need for additional bridges. Chapter 13 discusses web tier design and the Spring web MVC framework in detail.

Remoting support. Spring provides a thin abstraction layer for accessing remote services without hard-coded lookups, and for exposing Spring-managed application beans as remote services. Out-of-the-box support is included for RMI, Caucho's Hessian and Burlap web service protocols, and WSDL Web Services via JAX-RPC. Chapter 11 discusses lightweight remoting.

While Spring addresses areas as diverse as transaction management and web MVC, it uses a consistent approach everywhere. Once you have learned the basic configuration style, you will be able to apply it in many areas. Resources, middle tier objects, and web components are all set up using the same bean configuration mechanism. You can combine your entire configuration in one single bean definition file or split it by application modules or layers; the choice is up to you as the application developer. There is no need for diverse configuration files in a variety of formats, spread out across the application.

Spring on J2EE

Although many parts of Spring can be used in any kind of Java environment, it is primarily a J2EE application framework. For example, there are convenience classes for linking JNDI resources into a bean factory, such as JDBC DataSources and EJBs, and integration with JTA for distributed transaction management. In most cases, application objects do not need to work with J2EE APIs directly, improving reusability and meaning that there is no need to write verbose, hard-to-test, JNDI lookups.

Thus Spring allows application code to seamlessly integrate into a J2EE environment without being unnecessarily tied to it. You can build upon J2EE services where it makes sense for your application, and choose lighter-weight solutions if there are no complex requirements. For example, you need to use JTA as transaction strategy only if you face distributed transaction requirements. For a single database, there are alternative strategies that do not depend on a J2EE container. Switching between those transac-tion strategies is merely a matter of configuration; Spring's consistent abstraction avoids any need to change application code.

Spring offers support for accessing EJBs. This is an important feature (and relevant even in a book on "J2EE without EJB") because the use of dynamic proxies

as codeless client-side business delegates means that Spring can make using a local stateless session EJB an implementation-level, rather than a fundamen-tal architectural, choice. Thus if you want to use EJB, you can within a consistent architecture; however, you do not need to make EJB the cornerstone of your architecture. This Spring feature can make devel-oping EJB applications significantly faster, because there is no need to write custom code in service loca-tors or business delegates. Testing EJB client code is also much easier, because it only depends on the EJB's Business Methods interface (which is not EJB-specific), not on JNDI or the EJB API.

Spring also provides support for implementing EJBs, in the form of convenience superclasses for EJB implementation classes, which load a Spring lightweight container based on an environment variable specified in the ejb-jar.xml deployment descriptor. This is a powerful and convenient way of imple-menting SLSBs or MDBs that are facades for fine-grained POJOs: a best practice if you do choose to implement an EJB application. Using this Spring feature does not conflict with EJB in any way—it merely simplifies following good practice.

Introducing the Spring Framework

The main aim of Spring is to make J2EE easier to use and promote good programming practice. It does not reinvent the wheel; thus you'll find no logging packages in Spring, no connection pools, no distributed transaction coordinator. All these features are provided by other open source projects—such as Jakarta Commons Logging (which Spring uses for all its log output), Jakarta Commons DBCP (which can be used as local DataSource), and ObjectWeb JOTM (which can be used as transaction manager)—or by your J2EE application server. For the same reason, Spring doesn't provide an O/R mapping layer: There are good solutions for this problem area, such as Hibernate and JDO.

Spring does aim to make existing technologies easier to use. For example, although Spring is not in the business of low-level transaction coordination, it does provide an abstraction layer over JTA or any other transaction strategy. Spring is also popular as middle tier infrastructure for Hibernate, because it provides solutions to many common issues like SessionFactory setup, ThreadLocal sessions, and exception handling. With the Spring HibernateTemplate class, implementation methods of Hibernate DAOs can be reduced to one-liners while properly participating in transactions.

The Spring Framework does not aim to replace J2EE middle tier services as a whole. It is an application framework that makes accessing low-level J2EE container ser-vices easier. Furthermore, it offers lightweight alternatives for certain J2EE services in some scenarios, such as a JDBC-based transaction strategy instead of JTA

when just working with a single database. Essentially, Spring enables you to write appli-cations that scale down as well as up.

Spring for Web Applications

A typical usage of Spring in a J2EE environment is to serve as backbone for the logical middle tier of a J2EE web application. Spring provides a web application context concept, a powerful lightweight IoC container that seamlessly adapts to a web environment: It can be accessed from any kind of web tier, whether Struts, WebWork, Tapestry, JSF, Spring web MVC, or a custom solution.

The following code shows a typical example of such a web application context. In a typical Spring web app, an applicationContext.xml file will reside in the WEB-INF directory, containing bean defini-tions according to the “spring-beans” DTD. In such a bean definition XML file, business objects and resources are defined, for example, a “myDataSource” bean, a “myInventoryManager” bean, and a “myProductManager” bean. Spring takes care of their configuration, their wiring up, and their lifecycle.

```
<beans>
<bean id=”myDataSource” class=”org.springframework.jdbc.
datasource.DriverManagerDataSource”>
<property name=”driverClassName”> <value>com.mysql.jdbc.
Driver</value>
</property> <property name=”url”>
<value>jdbc:mysql:myds</value>
</property>
</bean><bean id=”myInventoryManager” class=”ebusiness.
DefaultInventoryManager”> <property name=”dataSource”>
<ref bean=”myDataSource”/> </property></bean>
<bean id=”myProductManager” class=”ebusiness.
DefaultProductManager”>
  <property name=”inventoryManager”>
<ref bean=”myInventoryManager”/> </property>
<property name=”retrieveCurrentStock”> <value>true</value>
</property>
</bean>
</beans>
```

By default, all such beans have “singleton” scope: one instance per context. The “myInventoryManager” bean will automatically be wired up with the defined DataSource, while “myProductManager” will in turn receive a reference to the “myInventoryManager” bean. Those objects (traditionally called “beans” in Spring

terminology) need to expose only the corresponding bean properties or constructor arguments (as you'll see later in this chapter); they do not have to perform any custom lookups.

A root web application context will be loaded by a ContextLoaderListener that is defined in web.xml as follows:

<web-app>
<listener> <listener-class>
org.springframework.web.context.ContextLoaderListener </listener-class>
</listener>
...
</web-app>

After initialization of the web app, the root web application context will be available as a ServletContext attribute to the whole web application, in the usual manner. It can be retrieved from there easily via fetching the corresponding attribute, or via a convenience method in org.springframework.web. context.support.WebApplicationContextUtils. This means that the application context will be available in any web resource with access to the ServletContext, like a Servlet, Filter, JSP, or Struts Action, as follows:

WebApplicationContext wac = WebApplicationContextUtils.
getWebApplicationContext(servletContext);

The Spring web MVC framework allows web controllers to be defined as JavaBeans in child application contexts, one per dispatcher servlet. Such controllers can express dependencies on beans in the root application context via simple bean references. Therefore, typical Spring web MVC applications never need to perform a manual lookup of an application context or bean factory, or do any other form of lookup.

Neither do other client objects that are managed by an application context themselves: They can receive collaborating objects as bean references.

The Core Bean Factory

In the previous section, we have seen a typical usage of the Spring IoC container in a web environment: The provided convenience classes allow for seamless integration without having to worry about low-level container details. Nevertheless, it does help to look at the inner workings to understand how Spring manages the container. Therefore, we will now look at the Spring bean container in more detail, starting at the lowest building block: the bean factory. Later, we'll continue with resource setup and details on the application context concept.

One of the main incentives for a lightweight container is to dispense with the multitude of custom facto-ries and singletons often found in J2EE applications. The

Spring bean factory provides one consistent way to set up any number of application objects, whether coarse-grained components or fine-grained busi-ness objects. Applying reflection and Dependency Injection, the bean factory can host components that do not need to be aware of Spring at all. Hence we call Spring a non-invasive application framework.

Fundamental Interfaces

The fundamental lightweight container interface is org.springframework.beans.factory.Bean Factory. This is a simple interface, which is easy to implement directly in the unlikely case that none of the implementations provided with Spring suffices. The BeanFactory interface offers two getBean() methods for looking up bean instances by String name, with the option to check for a required type (and throw an exception if there is a type mismatch).

```
public interface BeanFactory {
Object getBean(String name) throws BeansException;
Object getBean(String name, Class requiredType) throws BeansException;
boolean containsBean(String name);
boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
String[] getAliases(String name) throws NoSuchBeanDefinitionException;

}
```

The isSingleton() method allows calling code to check whether the specified name represents a sin-gleton or prototype bean definition. In the case of a singleton bean, all calls to the getBean() method will return the same object instance. In the case of a prototype bean, each call to getBean() returns an inde-pendent object instance, configured identically.

The getAliases() method will return alias names defined for the given bean name, if any. This mecha-nism is used to provide more descriptive alternative names for beans than are permitted in certain bean factory storage representations, such as XML id attributes.

The methods in most BeanFactory implementations are aware of a hierarchy that the implementation may be part of. If a bean is not found in the current factory, the parent factory will be asked, up until the root factory. From the point of view of a caller, all factories in such a hierarchy will appear to be merged into one. Bean definitions in ancestor contexts are visible to descendant contexts, but not the reverse.

All exceptions thrown by the BeanFactory interface and sub-interfaces extend org.springframework. beans.BeansException, and are unchecked. This reflects the fact that low-level configuration prob-lems are not usually recoverable: Hence, application developers can choose to write code to recover from such failures if they wish to, but

should not be forced to write code in the majority of cases where config-uration failure is fatal.

Most implementations of the BeanFactory interface do not merely provide a registry of objects by name; they provide rich support for configuring those objects using IoC. For example, they manage dependen-cies between managed objects, as well as simple properties. In the next section, we'll look at how such configuration can be expressed in a simple and intuitive XML structure.

The sub-interface org.springframework.beans.factory.ListableBeanFactory supports listing beans in a factory. It provides methods to retrieve the number of beans defined, the names of all beans, and the names of beans that are instances of a given type:

```
public interface ListableBeanFactory extends BeanFactory {
int getBeanDefinitionCount();
String[] getBeanDefinitionNames();
String[] getBeanDefinitionNames(Class type);
boolean containsBeanDefinition(String name);
Map getBeansOfType(Class type, boolean includePrototypes,
boolean includeFactoryBeans) throws BeansException
}
```

The ability to obtain such information about the objects managed by a ListableBeanFactory can be used to implement objects that work with a set of other objects known only at runtime.

In contrast to the BeanFactory interface, the methods in ListableBeanFactory apply to the current factory instance and do not take account of a hierarchy that the factory may be part of. The org.spring framework.beans.factory.BeanFactoryUtils class provides analogous methods that traverse an entire factory hierarchy.

There are various ways to leverage a Spring bean factory, ranging from simple bean configuration to J2EE resource integration and AOP proxy generation. The bean factory is the central, consistent way of setting up any kind of application objects in Spring, whether DAOs, business objects, or web controllers. Note that application objects seldom need to work with the BeanFactory interface directly, but are usu-ally configured and wired by a factory without the need for any Spring-specific code.

For standalone usage, the Spring distribution provides a tiny spring-core.jar file that can be embed-ded in any kind of application. Its only third-party dependency beyond J2SE 1.3 (plus JAXP for XML parsing) is the Jakarta Commons Logging API.

The bean factory is the core of Spring and the foundation for many other services that the framework offers. Nevertheless, the bean factory can easily be used stan-dalone if no other Spring services are required.

外文翻译

Spring 框架介绍

Spring 框架：这是一个流行的开源应用框架，它可以解决很多问题。这里主要介绍 Spring 的基本概念，并详细介绍其中以“bean 工厂”为核心的轻量级控制反转（IoC）容器。

Spring 让开发者更轻松的实现轻量级、可扩展的 J2EE 架构。对于我们推荐的架构模块，让 Spring 提供了即时可用的实现。此外，Spring 还提供了同意的应用架构方式，以及大量的中间层功能模块，能大大简化了 J2EE 的开发，并且比传统的开发方式更加灵活。

第一章：Spring 项目的出发点

1.1 解决被其它框架忽略的部分

在 J2EE 的各个具体领域都有很多出色的解决方案----web 框架、持久化方案、远程调用工具，凡此种种。然而，将这些工具整合成一个全面的架构却困难重重，甚至成为一种负担。Spring 的目标就是提供一种贯穿始终的解决方案，将各种专用框架整合成一个连贯的整体架构。Spring 还涉及到其它框架没有触及到的地方。例如，很少有框架提供普遍适用的事务管理或是数据库存储对象实现；即便各个领域都有很好的选择，也没有一个框架能够帮忙把所有这些东西整合到一个应用中去。因此，我们说 Spring 是一个应用框架，而不是 web 框架、IoC 框架、AOP 框架或者中间层框架什么的。

1.2 易于选择

一个框架应该有清楚的层次划分，允许用户使用其中的单项功能，而不是把自己的整个世界观强加给用户。Spring 中的许多特性---例如 JDBC 抽象层或者 Hibernation 集成----都可以作为一个库单独使用，当然也可以作为 Spring 整体方案的一个部分。

1.3 易于使用

正如前面说过的，仅仅 J2EE 本身并不易用---很多常见的问题光靠 J2EE 都很难解决。一个好的基础框架首先应该让容易的任务容易完成，而不需要让应用开发人员现在就为将来的复杂需求（例如分布式事务）买单。框架应该帮助开发人员合理使用 J2EE 的服务（例如 JTA），同时又避免造成对服务的依赖，从而减少不必要的复杂性。

1.4 鼓励最佳实现

Spring 力求把遵循最佳实践--例如“针对接口编程”--的成本降低到最小。另一方面，Spring 又力求不强加任何架构风格，而是把选择的权利留给开发者。

1.5 非入侵性

一个好的基础设施框架应保持应用组态灵活和一致，避免自定义单身和工厂的需要。一个单一的风格应该是适用于所有配置的需要，从中间层的网络控制器。

1.6 统一配置

测试整个应用程序或单个应用程序类的单元测试应该尽可能的容易。模仿对象替换资源或应用程序的对象应该是简单的。

1.7 可扩展

由于 Spring 本身是基于接口的，而不是类，它是很容易扩展或定制。许多 Spring 组件使某种形式的 Strategy 模式，因此可以方便定制。

第二章：一个分层的应用框架

前面我们从轻量级容器（lightwight container）的角度介绍了 Spring 框架，并将其与别的 IoC 容器（例如 PicoContainer）作了比较。虽然为 JavaBean 设计的轻量级容器是 Spring 的核心概念，但这也只是所有中间层解决方案的基础而已。

2.1 基础构建模块

Spring 是一个完整的应用框架，它可以在很懂应用层发挥作用。Spring 由多个子框架组成，而且这些框架都是相对独立的。不过，只要你愿意，你就可以将它们无缝集成起来，成为一个全面的应用框架，下面讲解 Spring 框架的关键概念。

2.1.1 bean 工厂

Spring 轻量级 IoC 容器能够配置、装备 JavaBean 和大多数 Java 对象，使得开发者不必定制 Singleton 和自己的配置机制。Spring 提供了多个“即拿即用”的 bean 工厂实现。其中包括一个基于 XML 的 bean 工厂。轻量级 IoC 容器和依赖注入（Dependency Injcction）将是本章节关注的内容。

2.1.2 应用上下文

Spring 应用程序上下文添加消息来源和资源加载支持扩展 bean 工厂的概念，并提供钩到现有的环境中。不同的盒子的实现包括独立的应用程序上下文和基于 Web 的应用程序上下文的 XML。

2.1.3 AOP 框架

Spring 的 AOP 框架提供的任何类的 Spring 轻量级容器管理方法拦截 AOP 支持。可以对轻量级容器管理的任何对象进行方法拦截。在 bean 工厂中，可以轻松的为 JavaBean 提供代理，从而在运行天衣无缝地将拦截器和其他的 adcice 整合进来。我们将在后面详细讨论 Spring 的 AOP 框架。Spring AOP 的主要用途就是为了 POJO 提供声明性的企业级服务。

2.1.4 自动代理

Spring 提供了一个更高的抽象层次的 AOP 框架和低水平的服务，同时也提供了很多基础性的服务，从而在 J2EE 环境中提供了类似于.NET 的易用性--特别是，声明的企业提供的服务可以通过源代码级的元数据驱动的。

2.1.5 事务管理

Spring 提供了一个通用的交易管理基础设施，可插拔的交易策略（例如 JTA 和 JDBC）和不同的事务划分方式。后面会详细介绍 Spring 事务管理的基本原理，及其强大的威力和灵活性。

2.1.6 DAO 的抽象

Spring 定义一组通用的数据访问异常，在创建通用的 DAO 接口时可以用这些异常类型抛出有意义的异常信息,，独立于底层的存储机制。后面阐述了更多的细节讨论 Spring 对 DAO 的支持，以及针对 JDBC JDO，Hibernate 的不同实施策略。

2.1.7 JDBC 的支持

Spring 提供了两个层次的抽象，使得编写基于 JDBC 的 DAO 特别简单：org.springframework.jdbc.core 包提供了基于模板、回调的 JDBC 用法，org.springframework.jdbc.object 包则把关系数据库操作建模为可服用对象。比起直接使用 JDBC，用 Spring 的 JDBC 包可以提供更大的生产力和消除常见的错误，如泄漏等。Spring JDBC 抽象层集成了事务抽象和 DAO 的抽象。

2.1.8 集成 O/R 映工具

Spring 提供了多种 O/R 映射工具的支持，如 Hibernate，JDO 和 ibatis 数据库简化资源设置，采集，和释放，并且将 O/R 映射与整个事务和 DAO 抽象集成起来。这些集成软件包允许应用程序分配自定义 ThreadLocal 会话和本地事务进行处理，不必考虑底层究竟采用哪种 O/R 映射工具。

2.1.9 web MVC 框架

Spring 提供了一个相当干净的 Web MVC 实现--同样使用了统一 JavaBean 配置方式。使用 Spring web 框架时，web 控制器也可以在 IoC 容器中配置，从而不必为“访问业务层服务”额外编写代码。Spring 还提供了通用的 DispatcherServlet 和“即拿急用”的控制器类。请求与控制器之间的映射方式、师徒的判断、本地化、以及其他重要服务都是可以插的，使得整个框架酷友更好的扩展性。Spring web 框架设计不仅仅局限于 JSP，还可以与其他的视图技术--例如 Velocity--无缝的结合。

2.1.10 远程调用支持

Spring 提供一种薄的抽象层用于访问远程服务，避免了在应用对象中硬编码对服务的查找。线成支持的远程调用方式包括 RMI，Caucho 的 Hessian 和 Burlap Web 服务的协议，和基于 JAX-RPCWSDL 的 Web 服务。

虽然 Spring 涉及了下至事务管理、上至 web MVC 的不同领域，它但它解决问题的方式却是一以贯之的。一旦你学会了基本的配置方式，你将能够应用在许多领域。资源，中间层对象，和 Web 组件都使用同一个 bean 配置机制的建立。你可以将你的整个配置在一个单一的 bean 定义文件或分裂，它的应用模块或层；选择是为应用程序开发人员到你。有没有需要在各种不同的格式，不同的配置文件，遍布的应用。

第三章：J2EE 之上的 Spring

虽然 Spring 的许多部分可用于任何 Java 环境，它主要是一个 J2EE 应用框架。例如，有链接到一个 bean 工厂的 JNDI 资源方便的类，如 JDBC 数据源和 EJB 和 JTA，分布式事务管理一体化。在大多数情况下，应用程序对象不需要与 j2ee

api 直接，提高可重用性和意义，不需要写冗长，难以测试，JNDI 查找。

因此，Spring 允许应用程序代码的无缝集成到 J2EE 环境不被不必要地捆绑在一起。你可以建立在 J2EE 服务，让您的应用程序，如果没有复杂的要求选择重量轻的解决方案。例如，你需要使用 JTA 事务的策略，如果你面对分布式事务的要求。为一个单一的数据库，有不依赖于一个 J2EE 容器替代策略。这些交易和策略之间的切换只是配置；春天的一致的抽象避免任何需要更改应用程序代码。

Spring 访问 EJB 提供支持。这是一个重要的特征(甚至在“没有 EJB 的 J2EE 一本有关)因为动态使用代理作为客户端业务代表无代码意味着春天可以使用本地的无状态会话 EJB 的实现程度，而不是一个根本的建筑，选择。因此，如果你想使用 EJB，您可以在一个一致的架构；然而，你不需要让你的架构的基石，EJB。今年春天功能可以使开发 EJB 应用程序更快，因为不需要编写自定义代码在服务中心职责范围或业务代表。测试 EJB 客户端代码也容易得多，因为它不仅取决于 EJB 的业务方法接口（这是不特定的，不是在 EJB）或 EJB JNDI API。

Spring 也提供了实现 EJB 的支持，在方便的超类 EJB 实现类的形式，其中负载基于 ejb-jar.xml 部署描述符指定的环境变量的 Spring 轻量级容器。这是一个强大的和方便的方式实现 SLSBs 或 MDBS 是细粒度的 POJO 立面：最佳实践，如果你选择实现 EJB 应用。Spring 的这些特性和 EJB 毫无冲突，因为它只是遵循了公认的最佳实践而已。

Spring 的主要目的是使 J2EE 更容易使用和促进良好的编程实践。Spring 不重新发明轮子，这样你会在 spring 找不到登录包，连接池，分布式事务协调器。这些特征是由其他的开源项目，如 Jakarta Commons 测井提供（Spring 使用其所有的日志输出），Jakarta Commons DBCP(可作为局部数据源)和 ObjectWeb JOTM(可作为事务管理器)或通过你的 J2EE 应用服务器。出于同样的原因，Spring 不提供 O/R 映射层：有良好的解决问题的方法，如 Hibernate，JDO。

Spring 的目标是使现有的技术更容易使用。例如，虽然 Spring 不是低级事务协调业务，它提供了在 jta 或者任何交易策略的一个抽象层。Spring 为 Hibernate 提供的中间层的基础设施也很受欢迎，因为它提供了像 SessionFactory 的设置，许多常见问题可以解决，ThreadLocal 会议，与异常处理。与 Spring HibernateTemplate 类，Hibernate DAO 实现方法可以减少到一个衬垫，同时适当参与交易。

Spring 框架的目的并不是要取代 J2EE 中间层服务。这是一个应用程序框架，使访问低级 J2EE 容器服务更容易。此外，它提供了在某些情况下，某些 J2EE 服务的轻量级的替代品，如代替 JTA JDBC 基础交易策略时，只使用一个数据库。基本上，Spring 将帮助你编写极具可伸缩性的应用程序。

第四章：web 应用中的 Spring

在 J2EE 环境中，Spring 有一种典型的用法：为 J2EE web 应用提供逻辑中间层的基本骨架。Spring 提供了“web 应用上下文”的概念，将一个强大的轻量

级 IoC 容器无缝结合到 web 环境中，各种 web 层程序--不管是 Struts、webWork、JSF、Spring web MVC 还是其他自制的方案--都可以使用 IoC 容器。

下面的代码显示了这样一个 Web 应用程序上下文的一个典型的例子。在一个典型的 Spring web 应用中，applicationContext.xml 文件驻留在 WEB-INF 目录，包含 bean 的声明（遵循“spring-beans”DTD）。这个文件中定义了业务对象和资源对象，如“myDataSource”、“myInventoryManager”和“myProductManager”等。Spring 将负责管理所有这些 bean 的配置、关联和生命周期。

```
<beans>
<bean    id=”myDataSource”    class=”org.springframework.jdbc.
datasource.DriverManagerDataSource”>
<property name=”driverClassName”> <value>com.mysql.jdbc.
Driver</value>
</property> <property name=”url”>
<value>jdbc:mysql:myds</value>
</property>
</bean><bean id=”myInventoryManager” class=”ebusiness.
DefaultInventoryManager”> <property name=”dataSource”>
<ref bean=”myDataSource”/> </property></bean>
<bean id=”myProductManager” class=”ebusiness.
DefaultProductManager”>
 <property name=”inventoryManager”>
<ref bean=”myInventoryManager”/> </property>
<property name=”retrieveCurrentStock”> <value>true</value>
</property>
</bean>
</beans>
```

在默认情况下，所有的 bean 都是通过 Singleton 模式创建的：每个应用上下文中只有一个实例。当 Spring 开始构造上述应用上下文时，“myInventoryManager”bean 将自动获得生命的 DataSource 对象（“myDataSource”bean）：随后，“myProductManager”也会获得“myInventoryMnager”bean 的引用。这些对象（也就是 Spring 属于常说的“bean”）只需要暴露相应的 bean 属性或者后遭子参数即可，而不需要进行任何特定的查找操作。

我们在 web.xml 中定义一个监听器（ContextLoaderListener），用于加载根应用上下文（root applicatiob context）。这个监听器定义如下：

```
<web-app>
<listener> <listener-class>
```

org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
...
</web-app>

Web 应用程序的初始化后，根 Web 应用程序上下文将作为 ServletContext 的属性对整个 Web 应用程序，以通常的方式。它可以很容易地通过取相应的属性，从那里检索，或通过 org.springframework.web 便利的方法。context.support.webapplicationcontextutils。这意味着应用程序的上下文将在任何 Web 资源访问 ServletContext 可用，如 Servlet，JSP，过滤器，或 Struts 动作，如下：

WebApplicationContext WAC = webapplicationcontextutils。
getwebapplicationcontext（ServletContext）;

Spring Web MVC 框架允许 web 控制器被定义为在应用程序上下文中的孩子，每一个调度 servlet。这种控制器可以表达依赖于豆类中的根的应用程序上下文中通过简单的 bean 的引用。因此，典型的 Spring Web MVC 应用程序不需要执行一个应用程序上下文或 bean 工厂手工查找，或做任何其他形式的查找。

其他客户端对象是由应用程序上下文自己管理不：他们可以接收的协作对象作为 bean 的引用。

第五章：核心 bean 工厂

在前面的章节中，我们已经看到在一个网络环境的 Spring IoC 容器典型用途：提供便利类允许无缝集成，而不必担心底层容器的细节。然而，它确实帮助的内部运作看明白春天经营集装箱。因此，现在我们将看看更详细的 Spring bean 容器，从最低的建筑块：bean 工厂。后来，我们将继续在应用程序上下文的概念资源设置和细节。

一个轻量级容器的主要动机之一是用定制的工厂，经常发现在 J2EE 应用程序的许多单身。Spring bean 工厂提供了设置任何数量的应用程序对象的一个一致的方式，无论是粗粒度组件或细粒度业务对象。应用反射和依赖注入，bean 工厂可以不需要知道 Spring 所有的组件的宿主。因此，我们称之为 Spring 的一种非侵入性的应用程序框架。

第六章：基础接口

在 Spring 中，轻量级容器最基本的接口时 org.springframework.beans.facory.BeanFactory。这是一个非常简单的接口，所以如果没有任何现成的 bean 工厂实现能够满足你的要求（这种情况非常罕见）。你也可以很容易地创建自己的实现。BeanFactory 接口提供了两个 getBean（）方法，它们都可以根据 String 类型的名称查找获取 bean 实例，两者的不同之处在于：其中一个 getBean()方法允许使用者检查获得的 bean 是否具有所需的类型（如果类型不符，会抛出一个 BeanNotOfRequireTypeExpection 异常）。

```
public interface BeanFactory {
Object getBean(String name) throws BeansException;
Object getBean(String name, Class requiredType) throws BeansException;
boolean containsBean(String name);
boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
String[] getAliases(String name) throws NoSuchBeanDefinitionException;
}
```

使用者可以通过 issingleton()方法检查牧歌特定名称声明的 bean 被定为 Singleton 还是 Prototype。如果 bean 被定义为 Singleton，所有对 getBean()方法的调用将返回同一个对象实例的引用；如果 bean 被定义为 Prototype，每次对 getBean()的调用都将新建一个独立的对象实例。

如果某个 bean 配置了别名，getAliases()方法将返回它所有的别名。借助这种机制，我们可以为 bean 提供更具描述力的名称，而不必受特定的 bean 工厂存储机制的约束--例如 XML 文档对 id 属性的取值就有诸多限制。

在大多数的 BeanFactory 实现方法是意识到一个层次，可实施的一部分。如果一个 bean 是不是在目前厂发现，母厂会问，直到根厂。从一个人的角度，在这样一个层次结构的所有工厂会合并成一个。在祖先的上下文的 bean 定义中是可见的后代，而不是反。

BeanFactory 接口的大多数实现不仅提供给了一个可以按名称查询的对象注册表，对于“用 IoC 配置这些对象”也提供了丰富的支持。例如，他们可以管理对象之间的依赖关系，也可以将简单属性值赋给对象。在下一节，我们将看到如何在一个简单、直观的 XML 结构中描述这些配置信息。

大多数的 BeanFactory 接口实现不只是提供一个注册的对象的名称;他们提供配置这些对象使用 IOC 丰富的支持。例如，他们管理的托管对象之间的关联，以及简单的性质。在下一节，我们将看看这样的配置可以在一个简单而直观的 XML 结构表达。

Org.springframework.beans.factory.listablebeanfactory 子接口可以列出工厂中所有的 bean。这个子接口提供了一系列方法，可以用于获得工厂中定义 bean 的数量、所有 bean 的名称、具有特定类型的所有的名称等：

```
public interface ListableBeanFactory extends BeanFactory {
int getBeanDefinitionCount();
String[] getBeanDefinitionNames();
String[] getBeanDefinitionNames(Class type);
boolean containsBeanDefinition(String name);
Map getBeansOfType(Class type, boolean includePrototypes,
```

boolean includeFactoryBeans) throws BeansException
}

ListableBeanFactory 可以获得管理对象的相关信息，可以利用这一能力来实现这样的对象：这种对象需要与一组别的对象协作，并且只有到运行时才知道与其协作的究竟是哪些对象。

相反，BeanFactory 接口，在 ListableBeanFactory 方法适用于目前的工厂实例和不带一个层次的帐户，工厂可能是部分。该 org.springframework.beans.factory.beanfactoryutils 类提供了类似的方法，遍历整个工厂层次。

Spring 的 bean 工厂有很多种方法,从简单的 bean 配置 J2EE 资源整合和 AOP 代理生成。bean 工厂是中央的，一致的方式设置任何一种在春天，应用对象是否刀，业务对象，或网络控制器。注意，应用对象很好需要直接用到 BeanFactory 接口--它们通常都在工厂中配置、组装，不需要针对 Spring 编写任何特殊的代码。

如果只需要使用 bean 工厂，Spring 发布了一个很小的 spring-core.jar 文件，可以植入任何种类的应用中去。除了 J2SE 1.3（以及用于 XML 解析的 JAXP）之外，它唯一依赖的第三方只有 Jakarta Commons Logging API。

小结：

在本文中，我们介绍了 Spring 应用框架。并介绍了一些 Spring 框架的基本知识，例如：如何搭建 Spring 框架，IOC，AOP，以及 Spring 事务控制等。接下来我们将深度讨论 ssm 框架的整合。