附录 1　外文原文

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.