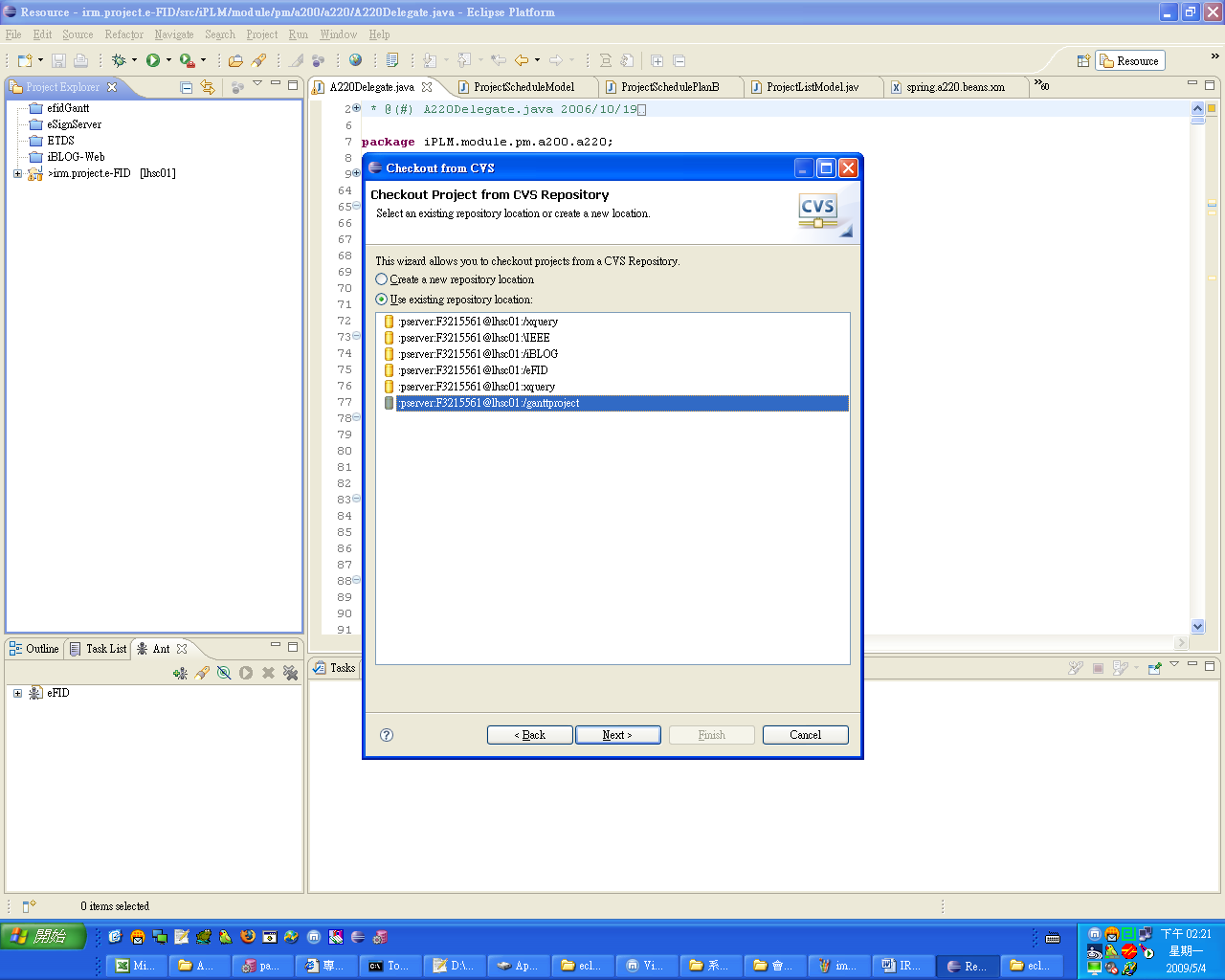
一．開發環境熟悉

1.eFID 工程的CVS位址：



2.eFID連接的測試資料庫在build.xml檔中可以找到：

<property name="springframework.sql.url" value="jdbc:oracle:thin:@10.153.24.96:1521:irmdb"/>

<property name="]" value="oracle.jdbc.driver.OracleDriver"/>

<property name="springframework.sql.uid" value="efid"/>

<property name="springframework.sql.passwd" value="test"/>

3.當第一次建制工程的時候，需要將

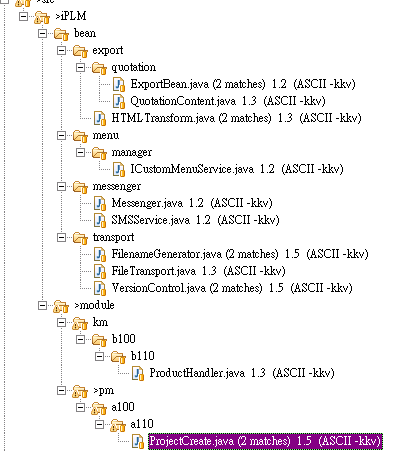
<!-- generate hibernate files-->

<antcall target="hibernate"/>

顯示出來，因為需要建立資料庫對應的hibernate檔

二．系統結構熟悉

1. spring 結構中用到的介面



1. 系統檔的執行流程（以專案管理為例）

點擊主巡覽列的專案管理，通過查找menu.xml文件，繼而查找menu.module.pm.xml文件

<section sid="section.pm.list">

<role>fid.\*</role>

<labelKey>module.pm.project.list</labelKey>

<action actionId="action.pm.listProject">

<mode>RELOAD</mode>

<href>/A220.do?action=showAllProjects</href>

<dwrScript></dwrScript>

<jsTemplate></jsTemplate>

</action>

<section-list selected=""></section-list>

</section>

根據該檔執行showAllProjects方法，該方法在A220Delegate檔裡，

**public** ModelAndView showAllProjects(HttpServletRequest request,

HttpServletResponse response) {

// this will be run twice,weird!

**if** (request.getAttribute("PROJECT\_WIZARD") != **null**

|| request.getUserPrincipal() == **null**) {

request.setAttribute("PROJECT\_WIZARD", **null**);

**return** **null**;

}

Integer status = StringUtility.*isEmpty*(request.getParameter("status")) ? **null**

: Integer.*valueOf*(request.getParameter("status"));

A220Bean bean = **new** A220Bean(m\_SessionFactory);

String userId = request.getUserPrincipal().getName();

**boolean** isManager = request.isUserInRole(m\_ManagerRoleId);

List list = bean.getProjectList(userId, status, isManager);

//List list = **null**;

//bean = **null**;

**try** {

String currentDate = DateUtility.*getStringByDate*(**new** Date(),

"yyyy-MM-dd");

request.setAttribute("currentDate", currentDate);

} **catch** (ParseException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** **new** ModelAndView(ShowAllProjectsPage, "model", list);

}

該方法通過調用A220Bean檔中的相應方法來實現功能，然後返回一個ModelAndView實例，其中三個參數的含義是，返回標識為ShowAllProjectsPage的一個頁面，其中頁面中顯示的值是list實例所包含的值，它是以model返回的，在頁面中調用取得顯示的資料的時候，可以通過取得model的值來獲得。

接下來，通過查找spring.a220.beans.xml檔來查找要跳轉到的頁面，

<bean id="A220Delegate"

class="iPLM.module.pm.a200.a220.A220Delegate" scope="singleton">

<property name="showAllProjectsPage"

value="/pm/A200/A220/projectSchedule.jsp" />

</bean>

這樣，頁面執行完showAllProjects方法後，會自動帶著查詢的結果，跳轉到projectSchedule.jsp頁面。

接下來需要在projectSchedule.jsp頁面顯示查詢的結果

環境的配置

1. 安裝JDK，tomcat，apache已經eclipse。
2. 環境變數的設置：

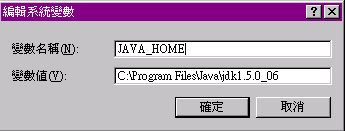
Tomcat安裝路徑的配置：



CLASSPATH：

.;C:\Program Files\Java\jre1.5.0\_06\lib\ext\QTJava.zip

Java安裝路徑的配置：



Path：

%CATALINA\_HOME%\bin;%JAVA\_HOME%\bin;C:\oracle\product\10.2.0\client\_1\bin;%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;C:\Program Files\QuickTime\QTSystem\

1. apache伺服器的配置參考doc下的文檔：



除了按照doc文檔下的配置之外，還需要修改

mod\_jk.properties文檔，用來搭配本機tomcat伺服器的啟動。

（需要特別注意apache伺服器的配置）

開發系統組態

1. spring.properties文件

#SpringFramework variables properties

#Thu May 07 10:54:25 GMT+08:00 2009

kpi.monthly.url=http\://fid.efih-foxconn.com/MonthlyKPI

jbpm.jdbc.password=test

indexer.dir=file\:///c\:/Index

cost.factor=160.0f

cost.chargeCode=R00908

wsdl.sms=http\://10.153.24.188\:8008/ESMS/SMSMessageService?wsdl

ftp.image.root=ftp\://foxconncic\:fox.123@10.153.24.94

contextPath.news.url=http\://fid.efih-foxconn.com/news/

contextPath.url=http\://fid.efih-foxconn.com

jbpm.jdbc.url=jdbc\:oracle\:thin\:@10.153.24.96\:1521\:irmdb

kpi.path=e\:\\\\KPI\\\\

ftp.root=ftp\://efid\:8jamnia3@fid.efih-foxconn.com/eFIDSZ

ftp.realPath=e\:\\eFID

ftp.image.realPath=e\:\\Images

wsdl.fibc=http\://10.153.24.116\:8080/web/services/MessageService?wsdl

jbpm.jdbc.username=efid

kpi.weekly.url=http\://fid.efih-foxconn.com/WeeklyKPI

用來配置spring的環境

2. hibernate.cfg.xml



<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- Generated file - Do not edit! -->

<hibernate-configuration>

<!-- a SessionFactory instance listed as /jndi/name -->

<session-factory>

<!-- properties -->

<property name="dialect">org.hibernate.dialect.OracleDialect</property>

<property name="show\_sql">false</property>

<property name="use\_outer\_join">false</property>

<property name="connection.username">efid</property>

<property name="connection.password">test</property>

<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>

<property name="connection.url">jdbc:oracle:thin:@10.153.24.96:1521:irmdb</property>

<property name="hibernate.cache.provider\_class">org.hibernate.cache.EhCacheProvider</property>

<property name="connection.pool\_size">10</property>

<property name="show\_sql">true</property>

<property name="hibernate.autoReconnect">true</property>

<property name="hibernate.cache.use\_second\_level\_cache">true</property>

<!-- mapping files -->

<mapping resource="iPLM/vo/Phone.hbm.xml"/>

<mapping resource="iPLM/vo/SignOpinion.hbm.xml"/>

<mapping resource="iPLM/vo/RootTypeInfo.hbm.xml"/>

<mapping resource="iPLM/vo/CustomerInfo.hbm.xml"/>

<mapping resource="iPLM/vo/Competitor.hbm.xml"/>

<mapping resource="iPLM/vo/PermissionInfo.hbm.xml"/>

<mapping resource="iPLM/vo/Task.hbm.xml"/>

<mapping resource="iPLM/vo/CostInfo.hbm.xml"/>

<mapping resource="iPLM/vo/Contact.hbm.xml"/>

<mapping resource="iPLM/vo/Discuss.hbm.xml"/>

<mapping resource="iPLM/vo/Logger.hbm.xml"/>

<mapping resource="iPLM/vo/Meeting.hbm.xml"/>

<mapping resource="iPLM/vo/KpiInfo.hbm.xml"/>

<mapping resource="iPLM/vo/ProjectIssueType.hbm.xml"/>

<mapping resource="iPLM/vo/Calendar.hbm.xml"/>

<mapping resource="iPLM/vo/ProjectInfo.hbm.xml"/>

<mapping resource="iPLM/vo/ShareCalendar.hbm.xml"/>

<mapping resource="iPLM/vo/ProjectCost.hbm.xml"/>

<mapping resource="iPLM/vo/Project.hbm.xml"/>

<mapping resource="iPLM/vo/MeetingIssue.hbm.xml"/>

<mapping resource="iPLM/vo/AgentTime.hbm.xml"/>

<mapping resource="iPLM/vo/Role.hbm.xml"/>

<mapping resource="iPLM/vo/Member.hbm.xml"/>

<mapping resource="iPLM/vo/NewsKeyword.hbm.xml"/>

<mapping resource="iPLM/vo/Event.hbm.xml"/>

<mapping resource="iPLM/vo/ProjectAgent.hbm.xml"/>

<mapping resource="iPLM/vo/ExchangeRate.hbm.xml"/>

<mapping resource="iPLM/vo/Issue.hbm.xml"/>

<mapping resource="iPLM/vo/TypeInfo.hbm.xml"/>

<mapping resource="iPLM/vo/StatusInfo.hbm.xml"/>

<mapping resource="iPLM/vo/Category.hbm.xml"/>

<mapping resource="iPLM/vo/News.hbm.xml"/>

<mapping resource="iPLM/vo/Notice.hbm.xml"/>

<mapping resource="iPLM/vo/ShareEvent.hbm.xml"/>

<mapping resource="iPLM/vo/User.hbm.xml"/>

<mapping resource="iPLM/vo/ContactWindow.hbm.xml"/>

<mapping resource="iPLM/vo/MeetingConclusion.hbm.xml"/>

<mapping resource="iPLM/vo/Permission.hbm.xml"/>

<mapping resource="iPLM/vo/PhoneBrand.hbm.xml"/>

<mapping resource="iPLM/vo/Attachment.hbm.xml"/>

</session-factory>

</hibernate-configuration>

Spring框架的基本機制

1. 先編寫介面程式
2. 編寫介面程式的實現類
3. 講實現類配置到xml檔中，通過讀取xml檔資訊來動態的生成類的物件。

**通過Bean Factory來封裝類**

Bean Factory負責根據設定檔創建Bean實例，可以配置的專案有：

1． Bean屬性值及依賴關係（對其他Bean的引用）

2． Bean創建模式（是否Singleton模式，即是否只針對指定類維持全域唯一的實例）

3． Bean初始化和銷毀方法

4． Bean的依賴關係

下面是一個較為完整的Bean配置示例：

<beans>

<description>Spring Bean Configuration Sample</description>

<bean

id="TheAction" **⑴**

class="net.xiaxin.spring.qs.UpperAction" **⑵**

singleton="true" **⑶**

init-method="init" **⑷**

destroy-method="cleanup" **⑸**

depends-on="ActionManager" **⑹**

>

<property name="message">

<value>HeLLo</value> **⑺**

</property>

<property name="desc">

<null/>

</property>

<property name="dataSource">

<ref local="dataSource"/> **⑻**

</property>

</bean>

<bean id="dataSource"

class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName">

<value>java:comp/env/jdbc/sample</value>

</property>

</bean>

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

</beans>

**⑴** id

Java Bean在BeanFactory中的唯一標識，代碼中通過BeanFactory獲取

JavaBean實例時需以此作為索引名稱。

**⑵** class

Java Bean 類名

**⑶** singleton

指定此Java Bean是否採用單例（Singleton）模式，如果設為“true”，則在

BeanFactory作用範圍內，只維護此Java Bean的一個實例，代碼通過BeanFactory

獲得此Java Bean實例的引用。反之，如果設為“false”，則通過BeanFactory獲取

此Java Bean實例時，BeanFactory每次都將創建一個新的實例返回。

**⑷** init-method

初始化方法，此方法將在BeanFactory創建JavaBean實例之後，在向應用層返回引

用之前執行。一般用於一些資源的初始化工作。

**⑸** destroy-method

銷毀方法。此方法將在BeanFactory銷毀的時候執行，一般用於資源釋放。

**⑹** depends-on

Bean依賴關係。一般情況下無需設定。Spring會根據情況組織各個依賴關係的構建工作（這裡

示例中的depends-on屬性非必須）。

只有某些特殊情況下，如JavaBean中的某些靜態變數需要進行初始化（這是一種Bad

Smell，應該在設計上應該避免）。通過depends-on指定其依賴關係可保證在此Bean加

載之前，首先對depends-on所指定的資源進行載入。

**⑺** <value>

通過<value/>節點可指定屬性值。BeanFactory將自動根據Java Bean對應的屬性

類型加以匹配。

下面的”desc”屬性提供了一個null值的設定示例。注意<value></value>代表一

個空字串，如果需要將屬性值設定為null，必須使用<null/>節點。

**⑻** <ref>

指定了屬性對BeanFactory中其他Bean的引用關係。示例中，TheAction的dataSource屬

性引用了id為dataSource的Bean。BeanFactory將在運行期創建dataSource bean實例，並將其

引用傳入TheAction Bean的dataSource屬性。

下面的代碼演示了如何通過BeanFactory獲取Bean實例：

InputStream is = **new** FileInputStream("bean.xml");

XmlBeanFactory factory = **new** XmlBeanFactory(is);

Action action = (Action) factory.getBean("TheAction");

**通過ApplicationContext來封裝類：**

**功能：**

1． 國際化支持

我們可以在Beans.xml檔中，對程式中的語言資訊（如提示資訊）進行定義，將程式中的提示

資訊抽取到設定檔中加以定義，為我們進行應用的各語言版本轉換提供了極大的靈活性。

2． 資源訪問

支援對檔和URL的訪問。

3． 事件傳播

事件傳播特性為系統中狀態改變時的檢測提供了良好支援。

4． 多實例載入

可以在同一個應用中載入多個Context實例。

1. 國際化支持：

<beans>

<description>Spring Quick Start</description>

<bean id="messageSource"

class="org.springframework.context.support.ResourceB

undleMessageSource">

<property name="basenames">

<list>

<value>messages</value>

</list>

</property>

</bean>

</beans>

這裡聲明了一個名為messageSource的Bean（注意對於Message定義，Bean ID必須為

messageSource，這是目前Spring的編碼規約），對應類為ResourceBundleMessageSource，

目前Spring中提供了兩個MessageSource介面的實現，即

ResourceBundleMessageSource和ReloadableResourceBundleMessageSource，後

者提供了無需重啟即可重新載入配置資訊的特性。

在配置節點中，我們指定了一個配置名“messages”。Spring會自動在CLASSPATH根路徑中按照如下順序搜尋設定檔並進行載入（以Locale為zh\_CN為例）:

messages\_zh\_CN.properties

messages\_zh.properties

messages.properties

messages\_zh\_CN.class

messages\_zh.class

messages.class

1. 資源訪問：

ApplicationContext.getResource方法提供了對資源檔訪問支援，如：

Resource rs = ctx.getResource("classpath:config.properties");

File file = rs.getFile();

上例從CLASSPATH根路徑中查找config.properties文件並獲取其檔案控制代碼。

getResource方法的參數為一個資源訪問位址，如：

file:C:/config.properties

/config.properties

classpath:config.properties

注意getResource返回的Resource並不一定實際存在，可以通過Resource.exists()方法對

其進行判斷。

**Web Context**

上面的示例中，ApplicationContext均通過編碼載入。對於Web應用，Spring提供了可配置的

ApplicationContext載入機制。

載入器目前有兩種選擇：ContextLoaderListener和ContextLoaderServlet。這兩者在功能上完全

等同，只是一個是基於Servlet2.3版本中新引入的Listener介面實現，而另一個基於Servlet介面實現。

開發中可根據目標Web容器的實際情況進行選擇。

配置非常簡單，在web.xml中增加：

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

或：

<servlet>

<servlet-name>context</servlet-name>

<servlet-class>

org.springframework.web.context.ContextLoaderServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

通過以上配置，Web容器會自動載入/WEB-INF/applicationContext.xml初始化

ApplicationContext實例，如果需要指定設定檔位置，可通過context-param加以指定：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/myApplicationContext.xml</param-value>

</context-param>

配置完成之後，即可通過

WebApplicationContextUtils.getWebApplicationContext

方法在Web應用中獲取ApplicationContext引用。

在eFID專案中，同樣存在這以上的配置：

1. eFID是通過ContextLoaderListener來載入ApplicationContext的

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

1. 然後web.xml配置了

<!-- SpringFramework configurations -->

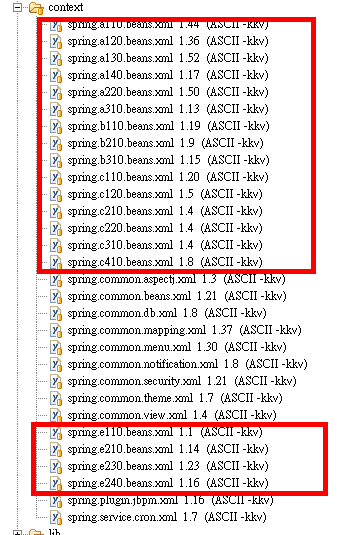
<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/context/spring.\*.beans.xml</param-value>

</context-param>

這樣，在Web容器會自動載入/WEB-INF/context/spring.\*.beans.xml這些XML檔，在工程中我們發現，web容器會自動載入以下檔



這寫xml檔中，配置的就是我們常見的一些bean，它實現了一些頁面的跳轉以及一些邏輯功能。

Spring MVC

1. 頁面

<html>

<body>

<form method="POST" action="/login.do">

<p align="center">登錄</p>

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not Open your Documents?

<br>

用戶名:

<input type="text" name="username" >

<br>

密 碼 :

<input type="password" name="password" >

<br>

<p>

<input type="submit" value="提交" name="B1">

<input type="reset" value="重置" name="B2">

</p>

</form>

</body>

</html>

點擊按鈕，講請求提交給login.do.

Spring 框架如何處理這個請求？？？

首先在工程的web.xml檔中定義：

<?xml version="1.0" encoding="ISO-8859-1"?>

**<web-app xmlns=**"http://java.sun.com/xml/ns/j2ee"

**xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"

**xsi:schemaLocation=**"http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd"

**version=**"2.4"**>**

**<servlet> ⑴**

**<servlet-name>Dispatcher</servlet-name>**

**<servlet-class>**

**org.springframework.web.servlet.DispatcherServlet**

**</servlet-class>**

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

**<init-param>**

**<param-name>contextConfigLocation</param-name>**

**<param-value>/WEB-INF/Config.xml</param-value>**

**</init-param>**

**</servlet>**

**<servlet-mapping> ⑵**

**<servlet-name>Dispatcher</servlet-name>**

**<url-pattern>\*.do</url-pattern>**

**</servlet-mapping>**

**</web-app>**

**⑴** Servlet定義

這裡我們定義了請求分發Servlet，即：

**org.springframework.web.servlet.DispatcherServlet**

**DispatcherServlet** 是Spring MVC 中負責請求調度的核心引擎，所有的請求將

由此Servlet 根據配置分發至各個邏輯處理單元。其內部同時也維護了一個

ApplicationContext實例。

我們在**<init-param>**節點中配置了名為“**contextConfigLocation**”的

Servlet參數，此參數指定了Spring設定檔的位置“**/WEB-INF/Config.xml**”。

如果忽略此設定，則默認為“**/WEB-INF/<servlet name>-servlet.xml**”，其

中**<servlet name>**以Servlet 名替換（在當前環境下，預設值也就是

“**/WEB-INF/Dispatcher-servlet.xml**）。

**⑵ 請求映射**

我們將所有以.do結尾的請求交給Spring MVC進行處理。當然，也可以設為其他值，

如.action、.action等。

通過以上設定，Web 伺服器將把登錄介面提交的請求轉交給Dispatcher 處理，

Dispatcher將提取請求（HttpServletRequest）中的輸入資料，分發給對應的處理單元，

各單元處理完畢後，將輸出頁面返回給Web伺服器，再由Web伺服器返回給用戶流覽器。

Dispatcher 根據什麼分發這些請求？顯然，我們還需要一個設定檔加以設定。這也就

是上面提及的Config.xml，此檔包含了所有的“請求/處理單元”關係映射設定，以及返回

時表現層的一些屬性設置。 Config.xml

<?xml version="1.0" encoding="UTF-8"?>

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

"http://www.springframework.org/dtd/spring-beans.dtd">

**<beans>**

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

<!--Definition of View Resolver -->

**<bean id=**"viewResolver" **⑴**

**class=**"org.springframework.web.servlet.view.InternalResou

rceViewResolver"**>**

**<property name=**"viewClass"**> ⑵**

**<value>**

**org.springframework.web.servlet.view.JstlView**

**</value>**

**</property>**

**<property name=**"prefix"**> ⑶**

**<value>**

**/WEB-INF/view/**

**</value>**

**</property>**

**<property name=**"suffix"**> ⑷**

**<value>.jsp</value>**

**</property>**

**</bean>**

<!--Request Mapping -->

**<bean id=**"urlMapping" **⑸**

**class=**"org.springframework.web.servlet.handler.SimpleUr

lHandlerMapping"**>**

**<property name=**"mappings"**>**

**<props>**

**<prop key=**"/login.do"**>LoginAction</prop>**

**</props>**

**</property>**

**</bean>**

<!---Action Definition-->

**<bean id=**"LoginAction" **⑹**

**class=**"net.xiaxin.action.LoginAction"**>**

**<property name=**"commandClass"**> ⑺**

**<value>net.xiaxin.action.LoginInfo</value>**

**</property>**

**<property name=**"fail\_view"**> ⑻**

**<value>loginfail</value>**

**</property>**

**<property name=**"success\_view"**>**

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

**<value>main</value>**

**</property>**

**</bean>**

**</beans>**

**⑴** Resolver設定

Resolver將把輸出結果與輸出介面相融合，為表現層提供呈現資源。

**⑵** View Resolver的viewClass參數

這裡我們使用JSP頁面作為輸出，因此，設定為：

**org.springframework.web.servlet.view.JstlView**

其餘可選的viewClass還有:

**org.springframework.web.servlet.view.freemarker.FreeMarker**

**View**（用於基於FreeMarker範本的表現層實現）

**org.springframework.web.servlet.view.velocity.VelocityView**

（用於基於velocity範本的表現層實現）

等。

**⑶⑷**View Resolver的prefix和suffix參數

指定了表現層資源的首碼和尾碼，運行時，Spring 將為指定的表現層資源自動追加

首碼和尾碼，以形成一個完整的資源路徑。另參見**⑻**

**⑸ “**請求/處理單元”關係映射

可以看到，這裡我們將“/login.do”請求映射到處理單元LoginAction。

**<props>**節點下可以有多個映射關係存在，目前我們只定義了一個。

**⑹** LoginAction定義

這裡定義了邏輯處理單元LoginAction 的具體實現，這裡，LoginAction 的實現

類為net.xiaxin.action.LoginAction。

**⑺** LoginAction的請求資料物件

commandClass 參數源於LoginAction 的基類BaseCommandController，

BaseCommandControlle 包含了請求資料封裝和驗證方法

（ BaseCommandController.bindAndValidate ） ， 它將根據傳入的

HttpServletRequest構造請求資料物件。

這裡我們指定commandClass 為**net.xiaxin.action.LoginInfo**，這是一個非

常簡單的Java Bean，它封裝了登錄請求所需的資料內容

public class LoginInfo {

private String username;

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

private String password;

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

}

Spring會根據LoginAction的commandClass定義自動載入對應的LoginInfo

實例。

之後，對Http 請求中的參數進行遍歷，並查找LoginInfo 物件中是否存在與之同

名的屬性，如果找到，則將此參數值複製到LoginInfo物件的同名屬性中.

請求資料轉換完成之後，我們得到了一個封裝了所有請求參數的Java 物件，並將此

對象作為輸入參數傳遞給LoginAction。

**⑻** 返回視圖定義

對於這裡的LoginAction 而言，有兩種返回結果，即登錄失敗時返回錯誤介面，登

錄成功時進入系統主介面。

對應我們配置了fail\_view、success\_view兩個自訂參數。

參數值將由Resolver進行處理，為其加上首碼尾碼，如對於fail\_view而言，實

際的視圖路徑為**/WEB-INF/view/loginfail.jsp。**

之後，Resolver 會將LoginAction的返回資料與視圖相融合，返回最終的顯示界

面。

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

private String success\_view;

protected ModelAndView onSubmit( **⑴**

Object cmd,

BindException ex

)throws Exception {

LoginInfo loginInfo = (LoginInfo) cmd; **⑵**

if (login(loginInfo) == 0) {

HashMap result\_map = new HashMap();

result\_map.put("logininfo", loginInfo);

List msgList = new LinkedList();

msgList.add("msg1");

msgList.add("msg2");

msgList.add("msg3");

result\_map.put("messages", msgList);

return new

ModelAndView(this.getSuccess\_view(), result\_map); **⑶**

} else {

return new ModelAndView(this.getFail\_view());

}

}

private int login(LoginInfo loginInfo) {

if ("Erica".equalsIgnoreCase(loginInfo.getUsername())

&& "mypass".equals(loginInfo.getPassword())) {

return 0;

}

return 1;

}

public String getFail\_view() {

return fail\_view;

}

public String getSuccess\_view() {

return success\_view;

}

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

public void setFail\_view(String string) {

fail\_view = string;

}

public void setSuccess\_view(String string) {

success\_view = string;

}

}

繼承的來自父類的onSubmit方法：

有倆種：

protected ModelAndView onSubmit(

Object cmd,

BindException ex

)

protected ModelAndView onSubmit(

HttpServletRequest request,

HttpServletResponse response,

Object cmd,

BindException ex

)

返回處理結果

ModelAndView類包含了邏輯單元返回的結果資料集和表現層資訊。ModelAndView

本身起到關係保存的作用。它將被傳遞給Dispatcher，由Dispatcher 根據其中

保存的結果資料集和表現層設定合成最後的介面。

這裡我們用到了兩種簽名版本的ModelAndView構造方法：

public ModelAndView(String viewname)

返回介面無需通過結果資料集進行填充。

public ModelAndView(String viewname, Map model)

返回介面由指定的結果資料集加以填充。可以看到，結果資料集採用了Map介面

實現的資料類型。其中包含了返回結果中的各個資料單元。關於結果資料集在界

面中的填充操作，可參見下面關於返回介面的描述。

上面這兩個版本的構造子中，通過viewname指定了展示層資源。

另外，我們也可以通過傳遞View物件指定展示層資源。

public ModelAndView(View view)

public ModelAndView(View view, Map model)

我們可以結合RedirectView完成轉向功能，如：

return new ModelAndView(

new RedirectView(“/redirected.jsp”

));

當然，我們也可以在帶有HttpServletRequest參數的onSubmit方法實現中，通

過HttpServletRequest/HttpServletResponse完成forward/redirect功

能，這兩種途徑可以達到同樣的效果。

返回失敗的介面之時純文字，不在浪費篇幅，返回成功的介面如下：

**<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core\_rt" %>**

**<html>**

**<body>**

**<p>**Login Success!!!**</p>**

**<p>**Current User:

**<c:out value="${logininfo.username}"/><br>**

**</p>**

**<p>**Your current messages:**</p>**

**<c:forEach items="${messages}"**

**var="item"**

**begin="0"**

**end="9"**

**step="1"**

**varStatus="var">**

**<c:if test="${var.index % 2 == 0}">**

\*

**</c:if>**

${item}**<br>**

**</c:forEach>**

**</body>**

**</html>**

頁

getAttribute(“logininfo”);

String username = logininfo.getUsername();

可以看到，EL大大節省了編碼量。

這裡引出的另外一個問題就是，EL 將從哪裡找到logininfo 物件，對於

**${logininfo.username}**這樣的運算式而言，首先會從當前頁面中尋找之前是

否定義了變數logininfo，如果沒有找到則依次到Request、Session、

Application 範圍內尋找，直到找到為止。如果直到最後依然沒有找到匹配的

變數，則返回null.

如果我們需要指定變數的尋找範圍，可以在EL運算式中指定搜尋範圍：

**${pageScope.logininfo.username}**

**${requestScope.logininfo.username}**

**${sessionScope.logininfo.username}**

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

**${applicationScope.logininfo.username}**

在 Spring 中，所有邏輯處理單元返回的結果資料，都將作為Attribute 被放

置到HttpServletRequest 物件中返回（具體實現可參見Spring 源碼中

org.springframework.web.servlet.view.InternalResourceView.

exposeModelAsRequestAttributes方法的實現代碼），也就是說Spring

MVC 中，結果資料物件預設都是**requestScope**。因此，在Spring MVC 中，

以下定址方法應慎用：

**${sessionScope.logininfo.username}**

**${applicationScope.logininfo.username}  
<c:forEach items="${messages}"**

**var="item"**

**begin="0"**

**end="9"**

**step="1"**

**varStatus="var">**

……

**</c:forEach>**

**<c:if test="${var.index % 2 == 0}">**

\*

**</c:if>**

**<c:choose>**

**<c:when test="${var.index % 2 == 0}">**

\*

**</c:when>**

**<c:otherwise>**

!

**</c:otherwise>**

**</c:choose>**

在 eFID專案中，首先在web.xml配置spring

<!-- Spring configuration -->

<servlet>

<servlet-name>iPLM SpringFramework Servlet</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/context/spring.plugins.\*.xml</param-value>

</init-param>

</servlet>

<!-- Spring mappings -->

<servlet-mapping>

<servlet-name>iPLM SpringFramework Servlet</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

對spring進行了配置，這樣，所有的以.do結尾的請求，都會通過spring MVC

進行處理，在這裡，spring配置的資源檔指向/WEB-INF/context/spring.plugins.\*.xml

也就是spring.plugin.jbpm.xml這個檔，

**spring.plugin.jbpm.xml：**

因為已經在build.xml檔中定義了spring的資料庫連接，

<property name="springframework.sql.url" value="jdbc:oracle:thin:@10.153.24.96:1521:irmdb"/>

<property name="springframework.sql.driver" value="oracle.jdbc.driver.OracleDriver"/>

<property name="springframework.sql.uid" value="efid"/>

<property name="springframework.sql.passwd" value="test"/>

<property name="springframework.sms.wsdl" value="http://10.153.24.188:8008/ESMS/SMSMessageService?wsdl"/>

<property name="springframework.fibc.wsdl" value="http://10.153.24.116:8080/web/services/MessageService?wsdl"/>

<property name="springframework.ftp.realPath" value="e:\eFID"/>

<property name="springframework.ftp.image.realPath" value="e:\Images"/>

<property name="springframework.kpi.path" value="e:\\KPI\\"/>

<property name="springframework.kpi.weekly.url" value="http://fid.efih-foxconn.com/WeeklyKPI"/>

<property name="springframework.kpi.monthly.url" value="http://fid.efih-foxconn.com/MonthlyKPI"/>

<property name="springframework.ftp.root" value="ftp://efid:8jamnia3@fid.efih-foxconn.com/eFIDSZ"/>

<property name="springframework.ftp.image.root" value="ftp://foxconncic:fox.123@10.153.24.94"/>

<property name="springframework.contextPath" value="http://fid.efih-foxconn.com"/>

<property name="springframework.news.contextPath" value="http://fid.efih-foxconn.com/news/"/>

<property name="springframework.hibernate.dialect" value="org.hibernate.dialect.OracleDialect"/>

<property name="springframework.lucene.indexDir" value="file:///c:/Index"/>

<property name="springframework.cost.chargeCode" value="R00908"/>

<property name="springframework.cost.factor" value="160.0f"/>

在建立結構的時候

<!-- ============================================================ -->

<!-- 建立目錄結構 -->

<!-- ============================================================ -->

<target name="init">

<mkdir dir="${build.path}"/>

<echo message="編譯後的程式放置於: ${build.path}"/>

<mkdir dir="${javadoc.path}"/>

<echo message="API 檔放置於: ${javadoc.path}"/>

<mkdir dir="${bin.path}"/>

<echo message="包裝後的應用程式放置於: ${bin.path}"/>

<echo message="更新 Springframework 變數 ..."/>

<propertyfile

file="${build.path}/spring.properties"

comment="SpringFramework variables properties">

<entry key="jbpm.jdbc.url" value="${springframework.sql.url}"/>

<entry key="jbpm.jdbc.username" value="${springframework.sql.uid}"/>

<entry key="jbpm.jdbc.password" value="${springframework.sql.passwd}"/>

<entry key="wsdl.sms" value="${springframework.sms.wsdl}"/>

<entry key="wsdl.fibc" value="${springframework.fibc.wsdl}"/>

<entry key="ftp.realPath" value="${springframework.ftp.realPath}"/>

<entry key="ftp.image.realPath" value="${springframework.ftp.image.realPath}"/>

<entry key="ftp.root" value="${springframework.ftp.root}"/>

<entry key="ftp.image.root" value="${springframework.ftp.image.root}"/>

<entry key="contextPath.url" value="${springframework.contextPath}"/>

<entry key="cost.chargeCode" value="${springframework.cost.chargeCode}"/>

<entry key="cost.factor" value="${springframework.cost.factor}"/>

<entry key="kpi.path" value="${springframework.kpi.path}"/>

<entry key="kpi.weekly.url" value="${springframework.kpi.weekly.url}"/>

<entry key="kpi.monthly.url" value="${springframework.kpi.monthly.url}"/>

<entry key="indexer.dir" value="${springframework.lucene.indexDir}"/>

<entry key="contextPath.news.url" value="${springframework.news.contextPath}"/>

</propertyfile>

<mkdir dir="${doc.path}/Report"/>

</target>

也給每一個屬性定義了名字，所以在spring.plugins.jdpm.xml檔裡，首先定義了資料庫的連結：

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource" scope="singleton">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

<property name="url" value="${jbpm.jdbc.url}"/>

<property name="username" value="${jbpm.jdbc.username}"/>

<property name="password" value="${jbpm.jdbc.password}"/>

</bean>

交易處理

**事務管理**

對於J2EE 應用程式而言，事務的處理一般有兩種模式：

1． 依賴特定事務資源的交易處理

這是應用開發中最常見的模式，即通過特定資源提供的事務機制進行事務管理。

如通過JDBC、JTA 的rollback、commit方法；Hibernate Transaction 的

rollback、commit方法等。這種方法大家已經相當熟悉。

2． 依賴容器的參數化事務管理

通過容器提供的集約式參數化事務機制，實現事務的外部管理，如EJB 中的事

務管理模式。

Spring的交易處理方式如下：

<beans>

<bean id="dataSource"

class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName">

<value>org.gjt.mm.mysql.Driver</value>

</property>

<property name="url">

<value>jdbc:mysql://localhost/sample</value>

</property>

<property name="username">

<value>user</value>

</property>

<property name="password">

<value>mypass</value>

</property>

</bean>

//首先定義了連接資料庫的javaBean，用來連接資料庫

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTr

ansactionManager">

<property name="dataSource">

<ref local="dataSource" />

</property>

</bean>

//定義了交易處理的bean，用來管理事務，其中有一個屬性，datasource，引用到剛才定義的datasource bean。

<bean id="userDAO" class="net.xiaxin.dao.UserDAO">

<property name="dataSource">

<ref local="dataSource" />

</property>

</bean>

//定義了一個普通的bean，來為下面的UserDaoProxy來服務

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

<bean id="userDAOProxy"

class="org.springframework.transaction.interceptor.Tran

sactionProxyFactoryBean">

<property name="transactionManager">

<ref bean="transactionManager" />

</property>

<property name="target">

<ref local="userDAO" />

</property>

<property name="transactionAttributes">

<props>

<prop key="insert\*">PROPAGATION\_REQUIRED</prop>

<prop key="get\*">

PROPAGATION\_REQUIRED,readOnly

</prop>

</props>

</property>

</bean>

</beans>

可以看到，在userDAOProxy 的"transactionAttributes"屬性中，我們定義了

針對userDAO 的事務策略，即將所有名稱以insert 開始的方法（如

UserDAO.insertUser方法）納入事務管理範圍。如果此方法中拋出異常，則Spring

將當前交易復原，如果方法正常結束，則提交事務。

而對所有名稱以get 開始的方法（如UserDAO.getUser 方法）則以唯讀的事務

處理機制進行處理。（設為唯讀型事務，可以使持久層嘗試對資料操作進行優化，如對

於唯讀事務Hibernate將不執行flush操作，而某些資料庫連接池和JDBC 驅動也對

唯讀型操作進行了特別優化。）

Spring交易處理的特點：

1． Spring可以將任意Java Class 納入事務管理

這裡的UserDAO只是我們編寫的一個普通Java Class，其中包含了一些

基本的資料應用邏輯。通過Spring，我們即可簡單的實現事務的可配置

化。也就是說，我們可以隨意為某個類的某個方法指定事務管理機制。

2． Spring事務管理並不依賴特定的事務資源。

持久層的封裝（JDBC）

Spring對JDBC進行了良好的封裝，通過提供相應的範本和輔助類，在相當程度上降低

了JDBC操作的複雜性。並且得益於Spring良好的隔離設計，JDBC封裝類庫可以脫離

Spring Context獨立使用，也就是說，即使系統並沒有採用Spring作為結構性框架，我們

也可以單獨使用Spring的JDBC部分（spring-dao.jar）來改善我們的代碼。

借助Template模式，我們大致也能實現這樣一個範本，不過，Spring的設計

者已經提前完成了這一步驟。org.springframework.jdbc.core.JdbcTemplate中包含了

這個範本實現的代碼，經過Spring設計小組精心設計，這個實現可以算的上是範本應用的

典範。特別是回檔（CallBack）的使用，使得整個範本結構清晰高效。

JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);

jdbcTemplate

.update(

"UPDATE user SET age = ? WHERE id = ?",

**new** PreparedStatementSetter() {

**public void** setValues(PreparedStatementSetter ps)

**throws** SQLException {

ps.setInt(1, 18);

ps.setString(2, "erica");

}

}

);

**JdbcTemplate與事務**

上例中的JdbcTemplate操作採用的是JDBC預設的AutoCommit模式，也就是說我們還

無法保證資料操作的原子性（要麼全部生效，要麼全部無效），如：

JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);

jdbcTemplate.update("UPDATE user SET age = 10 WHERE id = 'erica'");

jdbcTemplate.update("UPDATE user SET age = age+1 WHERE id = 'erica'");

由於採用了AutoCommit模式，第一個update操作完成之後被自動提交，資料庫

中”erica”對應的記錄已經被更新，如果第二個操作失敗，我們無法使得整個交易復原到最

初狀態。對於這個例子也許無關緊要，但是對於一個金融帳務系統而言，這樣的問題將導致

致命錯誤。

為了實現資料操作的原子性，我們需要在程式中引入事務邏輯，在JdbcTemplate中引入

事務機制，在Spring中有兩種方式：

1． 代碼控制的事務管理

2． 參數化配置的事務管理

代碼控制的事務管理：

首先，進行以下配置，假設設定檔為（Application-Context.xml）：

<beans>

<bean id="dataSource"

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName">

<value>net.sourceforge.jtds.jdbc.Driver</value>

</property>

<property name="url">

<value>jdbc:jtds:sqlserver://127.0.0.1:1433/Sample</value>

</property>

<property name="username">

<value>test</value>

</property>

<property name="password">

<value>changeit</value>

</property>

</bean>

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransac

tionManager">

<property name="dataSource">

<ref local="dataSource" />

</property>

</bean>

<bean id="userDAO" class="net.xiaxin.dao.UserDAO">

<property name="dataSource">

<ref local="dataSource" />

</property>

<property name="transactionManager">

<ref local="transactionManager" />

</property>

</bean>

</beans>

配置中包含了三個節點:

dataSource

這裡我們採用了apache dhcp元件提供的DataSource實現，並為其配置了

JDBC驅動、資料庫URL、用戶名和密碼等參數。

transactionManager

針對JDBC DataSource類型的資料來源，我們選用了

DataSourceTransactionManager

作為事務管理元件。

如果需要使用基於容器的資料來源（JNDI），我們可以採用如下配置：

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

<bean id="dataSource"

class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName">

<value>jdbc/sample</value>

</property>

</bean>

<bean id="transactionManager"

class="org.springframework.transaction.jta.JtaTrans

actionManager"

/>

userDAO

申明了一個UserDAO Bean，並為其指定了dataSource和

transactionManger資源。

UserDAO對應的代碼如下：

**public class** UserDAO {

**private** DataSource dataSource;

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

**private** PlatformTransactionManager transactionManager;

**public** PlatformTransactionManager getTransactionManager() {

**return** transactionManager;

}

**public void** setTransactionManager(PlatformTransactionManager

transactionManager) {

**this**.transactionManager = transactionManager;

}

**public** DataSource executeTestSource() {

**return** dataSource;

}

**public void** setDataSource(DataSource dataSource) {

**this**.dataSource = dataSource;

}

**public void** insertUser() {

TransactionTemplate tt =

**new** TransactionTemplate(getTransactionManager());

tt.execute(**new** TransactionCallback() {

**public** Object doInTransaction(TransactionStatus status) {

JdbcTemplate jt = **new** JdbcTemplate(executeTestSource());

jt.update(

"insert into users (username) values ('xiaxin');");

jt.update(

"insert into users (id,username) values(2,

'erica');");

**return null**;

}

});

}

}

**參數化配置的事務管理**

在上面的Application-Context.xml增加一個事務代理（UserDAOProxy）配置，同時，

由於事務由容器管理，UserDAO不再需要TransactionManager設定，將其移除：

<bean id="UserDAOProxy"

class="org.springframework.transaction.interceptor.Transac

tionProxyFactoryBean">

<property name="transactionManager">

<ref bean="transactionManager" />

</property>

<property name="target">

<ref local="userDAO" />

</property>

<property name="transactionAttributes">

<props>

<prop key="insert\*">PROPAGATION\_REQUIRED</prop>

<prop key="\*">PROPAGATION\_REQUIRED,readOnly</prop>

</props>

</property>

</bean>

<bean id="userDAO" class="net.xiaxin.dao.UserDAO">

<property name="dataSource">

<ref local="dataSource" />

</property>

</bean>

與之對應，UserDAO.insertUser的代碼修改如下：

**public void** insertUser(RegisterInfo regInfo) {

JdbcTemplate jt = **new** JdbcTemplate(executeTestSource());

jt.update("insert into users (username) values ('xiaxin');");

jt.update("insert into users (id,username) values (2,'erica');");

}

Hibernate in spring

Hibernate-Context.xml:

<beans>

<bean id="dataSource"

class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName">

<value>net.sourceforge.jtds.jdbc.Driver</value>

</property>

<property name="url">

<value>jdbc:jtds:sqlserver://127.0.0.1:1433/Sample</value>

</property>

<property name="username">

<value>test</value>

</property>

<property name="password">

<value>changeit</value>

</property>

</bean>

<bean id="sessionFactory"

class="org.springframework.orm.hibernate.LocalSessionFactoryBean"

>

<property name="dataSource">

<ref local="dataSource" />

</property>

<property name="mappingResources">

<list>

<value>net/xiaxin/dao/entity/User.hbm.xml</value>

</list>

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">

net.sf.hibernate.dialect.SQLServerDialect

</prop>

<prop key="hibernate.show\_sql">

true

</prop>

</props>

</property>

</bean>

<bean id="transactionManager"

class="org.springframework.orm.hibernate.HibernateTransactionMana

ger">

<property name="sessionFactory">

<ref local="sessionFactory" />

</property>

</bean>

<bean id="userDAO" class="net.xiaxin.dao.UserDAO">

<property name="sessionFactory">

<ref local="sessionFactory" />

</property>

</bean>

<bean id="userDAOProxy"

class="org.springframework.transaction.interceptor.TransactionPro

xyFactoryBean">

<property name="transactionManager">

<ref bean="transactionManager" />

</property>

<property name="target">

<ref local="userDAO" />

</property>

<property name="transactionAttributes">

<props>

<prop key="insert\*">PROPAGATION\_REQUIRED</prop>

<prop key="get\*">PROPAGATION\_REQUIRED,readOnly</prop>

</props>

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

</property>

</bean>

</beans>

與上面JDBC中的配置相對比，區別主要在於：

1． SessionFactory的引入

Hibernate中通過SessionFactory創建和維護Session。Spring對

SessionFactory的配置也進行了整合，無需再通過Hibernate.cfg.xml對

SessionFactory進行設定。

SessionFactory節點的mappingResources屬性包含了映射檔的路徑，list

節點下可配置多個映射檔。

hibernateProperties節點則容納了所有的屬性配置。

可以對應傳統的Hibernate.cfg.xml檔結構對這裡的SessionFactory配置

進行解讀。

2． 採用面向Hibernate的TransactionManager實現：

org.springframework.orm.hibernate.HibernateTransactionManag

er

對應剛才的Users表，建立如下映射類：

User.java:

/\*\*

\* @hibernate.class table="users"

\*/

**public class** User {

**public** Integer id;

**public** String username;

**public** String password;

/\*\*

\* @hibernate.id

\* column="id"

\* type="java.lang.Integer"

\* generator-class="native"

\*/

**public** Integer getId() {

**return** id;

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

}

**public void** setId(Integer id) {

**this**.id = id;

}

/\*\*

\* @hibernate.property column="password" length="50"

\*/

**public** String getPassword() {

**return** password;

}

**public void** setPassword(String password) {

**this**.password = password;

}

/\*\*

\* @hibernate.property column="username" length="50"

\*/

**public** String getUsername() {

**return** username;

}

**public void** setUsername(String username) {

**this**.username = username;

}

}

上面的代碼中，通過xdoclet指定了類/表；屬性/欄位的映射關係，通過xdoclet ant

task 我們可以根據代碼生成對應的user.hbm.xml文件。具體細節請參見《hibernate開

髮指南》一文。

下麵是生成的user.hbm.xml：

<hibernate-mapping>

<class

name="net.xiaxin.dao.entity.User"

table="users"

dynamic-update="false"

dynamic-insert="false"

>

<id

name="id"

column="id"

SpringFrameWork Developer’s Guide Version 0.6

October 8, 2004 So many open source projects.Why not **Open** your **Doc**uments?

type="java.lang.Integer"

>

<generator class="native">

</generator>

</id>

<property

name="password"

type="java.lang.String"

update="true"

insert="true"

access="property"

column="password"

length="50"

/>

<property

name="username"

type="java.lang.String"

update="true"

insert="true"

access="property"

column="username"

length="50"

/>

</class>

在eFID系統中,spring.plugs.jbpm.xml檔配置了系統的交易處理機制：

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource" scope="singleton">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

<property name="url" value="${jbpm.jdbc.url}"/>

<property name="username" value="${jbpm.jdbc.username}"/>

<property name="password" value="${jbpm.jdbc.password}"/>

</bean>

配置連接的資料參數。

<bean id="jbpmSessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean" destroy-method="destroy" scope="singleton">

<property name="dataSource" ref="dataSource"/>

<property name="mappingLocations">

<list>

<value>classpath\*:/org/jbpm/\*\*/\*.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">org.hibernate.dialect.OracleDialect</prop>

<prop key="hibernate.show\_sql">false</prop>

<prop key="hibernate.generate\_statistics">false</prop>

<prop key="hibernate.cache.use\_second\_level\_cache">false</prop>

<prop key="hibernate.cache.use\_query\_cache">false</prop>

</props>

</property>

</bean>

**DWR技術**

DWR（Direct Web Remoting）是一個[開源](http://baike.baidu.com/view/9664.htm)的類庫,可以幫助開發人員開發包含[AJAX](http://baike.baidu.com/view/1641.htm)技術的網站.它可以允許在流覽器裡的代碼使用運行在WEB伺服器上的JAVA函數,就像它就在流覽器裡一樣.

　　它包含兩個主要的部分:允許JavaScript從WEB伺服器上一個遵循了AJAX原則的Servlet中獲取資料.另外一方面一個JavaScript庫可以幫助網站開發人員輕鬆地利用獲取的資料來動態改變網頁的內容.

DWR採取了一個類似AJAX的新方法來動態生成基於JAVA類的JavaScript代碼.這樣WEB開發人員就可以在JavaScript裡使用Java代碼就像它們是流覽器的本地代碼(用戶端代碼)一樣;但是Java代碼運行在WEB伺服器端而且可以自由訪問WEB 伺服器的資源.出於安全的理由,WEB開發者必須適當地配置哪些Java類可以安全的被外部使用.

　　這個從JAVA到JavaScript的遠端功能方法給DWR的用戶帶來非常像傳統的RPC機制,就像RMI或者SOAP一樣,而且擁有運行在WEB上但是不需要流覽器外掛程式的好處.

　　DWR不認為流覽器/WEB伺服器協議是重要的,而更樂於保證程式設計介面的簡單自然.對此最大的挑戰就是把AJAX的非同步特性和正常JAVA方法調用的同步特性相結合.在非同步模式下,結果資料在開始調用之後的一段時間之後才可以被非同步訪問獲取到.DWR允許WEB開發人員傳遞一個回呼函數,來非同步處理Java函式呼叫過程.

**1.1、dwr.xml的配置**

　　<dwr>

　　<allow>

　　<create creator="new" javascript="testClass" >

　　<include method="testMethod1"/>

　　</create>

　　</allow>

　　</dwr>

　　<allow>標籤中包括可以暴露給javascript訪問的東西。

　　<create>標籤中指定javascript中可以訪問的java類，並定義DWR應當如何獲得要進行遠端的類的實例。creator="new"屬性指定java類實例的生成方式，new意味著DWR應當調用類的預設構造函數來獲得實例，其他的還有spring方式，通過與IOC容器Spring進行集成來獲得實例等等。javascript=" testClass "屬性指定javascript代碼訪問物件時使用的名稱。

　　標籤指定要公開給javascript的java類名。

　　<include>標籤指定要公開給javascript的方法。不指定的話就公開所有方法。

　　<exclude>標籤指定要防止被訪問的方法。

**1.2、javascript中調用**

　　首先，引入javascript腳本

　　<script src='dwr/interface/ testClass.js'></script>

　　<script src='dwr/engine.js'></script>

　　<script src='dwr/util.js'></script>

　　其中TestClass.js是dwr根據設定檔自動生成的，engine.js和util.js是dwr自帶的指令檔。

　　其次，編寫調用java方法的javascript函數

　　Function callTestMethod1(){

　　testClass.testMethod1();

　　}

**2、調用有簡單返回值的java方法**

**2.1、dwr.xml的配置**

　　配置同1.1

　　<dwr>

　　<allow>

　　<create creator="new" javascript="testClass" >

　　<include method="testMethod2"/>

　　</create>

　　</allow>

　　</dwr>

**2.2、javascript中調用**

　　首先，引入javascript腳本

　　其次，編寫調用java方法的javascript函數和接收返回值的回呼函數

　　Function callTestMethod2(){

　　testClass.testMethod2(callBackFortestMethod2);

　　}

　　Function callBackFortestMethod2(data){

　　//其中data接收方法的返回值

　　//可以在這裡對返回值進行處理和顯示等等

　　alert("the return value is " + data);

　　}

　　其中callBackFortestMethod2是接收返回值的回呼函數

**3、調用有簡單參數的java方法**

　　3.1、dwr.xml的配置

　　配置同1.1

　　<dwr>

　　<allow>

　　<create creator="new" javascript="testClass" >

　　<include method="testMethod3"/>

　　</create>

　　</allow>

　　</dwr>

　　3.2、javascript中調用

　　首先，引入javascript腳本

　　其次，編寫調用java方法的javascript函數

　　Function callTestMethod3(){

　　//定義要傳到java方法中的參數

　　var data;

　　//構造參數

　　data = “test String”;

　　testClass.testMethod3(data);

　　}

**4、調用返回JavaBean的java方法**

　　4.1、dwr.xml的配置

　　<dwr>

　　<allow>

　　<create creator="new" javascript="testClass" >

　　<include method="testMethod4"/>

　　</create>

　　<convert converter="bean" match=""com.dwr.TestBean">

　　</convert>

　　</allow>

　　</dwr>

　　<creator>標籤負責公開用於Web遠程的類和類的方法，<convertor>標籤則負責這些方法的參數和返回類型。convert元素的作用是告訴DWR在伺服器端Java 物件表示和序列化的JavaScript之間如何轉換資料類型。DWR自動地在Java和JavaScript表示之間調整單一資料型別。這些類型包括Java原生類型和它們各自的封裝類表示，還有String、Date、陣列和集合類型。DWR也能把JavaBean轉換成JavaScript 表示，但是出於安全性的原因，要求顯式的配置，<convertor>標籤就是完成此功能的。converter="bean"屬性指定轉換的方式採用JavaBean命名規範，match=""com.dwr.TestBean"屬性指定要轉換的javabean名稱，標籤指定要轉換的JavaBean屬性。

　　4.2、javascript中調用

　　首先，引入javascript腳本

　　其次，編寫調用java方法的javascript函數和接收返回值的回呼函數

　　Function callTestMethod4(){

　　testClass.testMethod4(callBackFortestMethod4);

　　}

　　Function callBackFortestMethod4(data){

　　//其中data接收方法的返回值

　　//對於JavaBean返回值，有兩種方式處理

　　//不知道屬性名稱時，使用如下方法

　　for(var property in data){

　　alert("property:"+property);

　　alert(property+":"+data[property]);

　　}

　　//知道屬性名稱時，使用如下方法

　　alert(data.username);

　　alert(data.password);

　　}

　　其中callBackFortestMethod4是接收返回值的回呼函數

**5、調用有JavaBean參數的java方法**

　　5.1、dwr.xml的配置

　　配置同4.1

　　<dwr>

　　<allow>

　　<create creator="new" javascript="testClass" >

　　<include method="testMethod5"/>

　　</create>

　　<convert converter="bean" match="com.dwr.TestBean">

　　</convert>

　　</allow>

　　</dwr>

　　5.2、javascript中調用

　　首先，引入javascript腳本

　　其次，編寫調用java方法的javascript函數

　　Function callTestMethod5(){

　　//定義要傳到java方法中的參數

　　var data;

　　//構造參數，data實際上是一個object

　　data = { username:"user", password:"password" }

　　testClass.testMethod5(data);

　　}

**6、調用返回List、Set或者Map的java方法**

　　6.1、dwr.xml的配置

　　配置同4.1

　　<dwr>

　　<allow>

　　<create creator="new" javascript="testClass" >

　　<include method="testMethod6"/>

　　</create>

　　<convert converter="bean" match="com.dwr.TestBean">

　　</convert>

　　</allow>

　　</dwr>

　　注意：如果List、Set或者Map中的元素均為簡單類型（包括其封裝類）或String、Date、陣列和集合類型，則不需要<convert>標籤。

　　6.2、javascript中調用(以返回List為例，List的元素為TestBean)

　　首先，引入javascript腳本

　　其次，編寫調用java方法的javascript函數和接收返回值的回呼函數

　　Function callTestMethod6(){

　　testClass.testMethod6(callBackFortestMethod6);

　　}

　　Function callBackFortestMethod6(data){

　　//其中data接收方法的返回值

　　//對於JavaBean返回值，有兩種方式處理

　　//不知道屬性名稱時，使用如下方法

　　for(var i=0;i<data.length;i++){

　　for(var property in data){

　　alert("property:"+property);

　　alert(property+":"+data[property]);

　　}

　　}

　　//知道屬性名稱時，使用如下方法

　　for(var i=0;i<data.length;i++){

　　alert(data.username);

　　alert(data.password);

　　}

　　}

**7、調用有List、Set或者Map參數的java方法**

　　7.1、dwr.xml的配置

　　<dwr>

　　<allow>

　　<create creator="new" javascript="testClass" >

　　<include method="testMethod7"/>

　　</create>

　　<convert converter="bean" match="com.dwr.TestBean">

　　</convert>

　　</allow>

　　<signatures>

　　<![CDATA[

　　import java.util.List;

　　import com.dwr.TestClass;

　　import com.dwr.TestBean;

　　TestClass.testMethod7(List<TestBean>);

　　]]>

　　</signatures>

　　</dwr>

　　<signatures>標籤是用來聲明java方法中List、Set或者Map參數所包含的確切類，以便java代碼作出判斷。

　　7.2、javascript中調用(以返回List為例，List的元素為TestBean)

　　首先，引入javascript腳本

　　其次，編寫調用java方法的javascript函數

　　Function callTestMethod7(){

　　//定義要傳到java方法中的參數

　　var data;

　　//構造參數，data實際上是一個object陣列，即陣列的每個元素均為object

　　data = [

　　{

　　username:"user1",

　　password:"password2"

　　},

　　{

　　username:"user2",

　　password:" password2"

　　}

　　];

　　testClass.testMethod7(data);

　　}

　　注意：

　　1、對於第6種情況，如果java方法的返回值為Map，則在接收該返回值的javascript回呼函數中如下處理：

　　function callBackFortestMethod(data){

　　//其中data接收方法的返回值

　　for(var property in data){

　　var bean = data[property];

　　alert(bean.username);

　　alert(bean.password);

　　}

　　}

　　2、對於第7種情況，如果java的方法的參數為Map（假設其key為String，value為TestBean），則在調用該方法的javascript函數中用如下方法構造要傳遞的參數：

　　function callTestMethod (){

　　//定義要傳到java方法中的參數

　　var data;

　　//構造參數，date實際上是一個object，其屬性名為Map的key，屬性值為Map的value

　　data = {

　　"key1":{

　　username:"user1",

　　password:"password2"

　　},

　　"key2":{

　　username:"user2",

　　password:" password2"

　　}

　　};

　　testClass.testMethod(data);

　　}

　　並且在dwr.xml中增加如下的配置段

　　<signatures>

　　<![CDATA[

　　import java.util.List;

　　import com.dwr.TestClass;

　　import com.dwr.TestBean;

　　TestClass.testMethod7(Map<String,TestBean>);

　　]]>

　　</signatures>

　　3、由以上可以發現，對於java方法的返回值為List(Set)的情況，DWR將其轉化為Object陣列，傳遞個javascript；對於java方法的返回值為Map的情況，DWR將其轉化為一個Object，其中Object的屬性為原Map的key值，屬性值為原Map相應的value值。

　　4、如果java方法的參數為List(Set)和Map的情況，javascript中也要根據3種所說，構造相應的javascript資料來傳遞到java中

**[**[**編輯本段**](http://baike.baidu.com/view/73492.htm)**]**

**安裝配置**

　　1. 安裝 DWR JAR 包

　　下載 dwr.jar 檔，然後將它放在 web 程式的 WEB-INF/lib 目錄下面，很可能在這個目錄下已經有一些 jar 檔了。

　　2. 編輯 config 檔

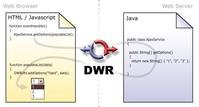
　　將下面的代碼添加到 WEB-INF/web.xml 檔中，<servlet>需要放在另外的<servlet>之後，<servlet-mapping>也是如此。

　　<servlet>

　　<servlet-name>dwr-invoker</servlet-name>

　　<display-name>DWR Servlet</display-name>

　　<servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>

[](http://imgsrc.baidu.com/baike/pic/item/91ae68c67ac954079d163de0.jpg)

workjjh

　　<init-param>

　　debug</param-name>

　　true</param-value>

　　</init-param>

　　</servlet>

　　<servlet-mapping>

　　<servlet-name>dwr-invoker</servlet-name>

　　<url-pattern>/dwr/\*</url-pattern>

　　</servlet-mapping>

　　接下來創建 dwr.xml 文件並將此放在 web.xml 所在的 WEB-INF 目錄下。檔中類似下面的內容：

　　<!DOCTYPE dwr PUBLIC

　　”-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN”

　　”http://www.getahead.ltd.uk/dwr/dwr10.dtd”>

　　<dwr>

　　<allow>

　　<create creator=”new” javascript=”JDate”>

　　</create>

　　<create creator=”new” javascript=”Demo”>

　　</create>

　　</allow>

　　</dwr>

　　DWR 設定檔定義了由 DWR 創建和被 Javascript 遠端使用的類。在上面的例子中我們在遠端創建了2個類，並且給出了在 Javascript 中的類名。

　　上面使用的 new creator 使用了所有的 JavaBeans 必須含有的公有（public）無參（no-args）的構造函數。值得一提的是， DWR 還有一些限制：

　　避免使用 JavaScript 保留字；以保留字命名的方法將自動被排除。大多數 JavaScript 的保留字同時也是 Java 的保留字，因此無論如何你也不能使用一個名為 “try()” 的方法。但是最常用的一個詞 “delete()”，在 JavaScript 中有特殊的含義而在 Java 中沒有。

　　重載方法（Overloaded methods）將會在調用的時候陷入未知的狀況，因此應該避免重載方法。

　　3. 訪問下麵的位址

　　http://localhost:8080/[YOUR-WEBAPP]/dwr/

　　你應該會看到一個頁面，上面顯示了剛才你在第二步所創建的類。進入一個連結之後你會看到所有等待調用方法的列表。這些動態產生的例子你也能通過 DWR 來實現。

　　親自嘗試和體會下吧。

　　怎樣應用到你的 Web 程式中？

　　在側邊欄有很多例子演示了怎樣改變網頁中的文本、更新清單、動作表單和動態修改表格。每一個例子都有詳細的說明。

　　另外一個開始方法就是從頁面中查看原始程式碼，這些頁面你剛剛流覽過：

　　進入 http://localhost:8080/[YOUR-WEBAPP]/dwr/ 然後點擊你創建的類；

　　查看原始程式碼然後定位到你所感興趣的方法的代碼行；

　　將這些文本粘貼到你 Web 程式的一個 HTML 或 JSP頁面中；

　　包含下麵的 javascrip 檔：

　　<script src=’/[YOUR-WEBAPP]/dwr/interface/[YOUR-SCRIPT].js’></script>

　　<script src=’/[YOUR-WEBAPP]/dwr/engine.js’></script>

　　你可以根據實際情況修改 /[YOUR-WEBAPP]/ 部分。

AOP

一個spring框架的AOP例子

2009-04-30 11:09

|  |  |
| --- | --- |
| |  | | --- | | （轉）一個spring框架的AOP例子  接 觸spring有一段時間了，不過都是看的多，寫的少，工作忙，公司也不准備用。自己寫過一些小東西用到，也只用到了BeanFactory組裝對 象，JdbcTemplate代替jdbc，事務管理。。。東抓一把，西抓一把，沒形成系統。最近也在看spring自帶的reference，一時手 癢，寫了個AOP的創建advice的例子，比之單純地使用TransationProxyFactoryBean對AOP的理解又深入了點，打算看看它 的原始程式碼，基於CGLIB的實現對類的代理不瞭解，倒是好奇它如何實現對介面的代理——也就是利用J2SE的動態代理技術。  例子如下：講述一間書店開始打折促銷，規則是每一名顧客只能買一本書，而且<spring in action>已經脫銷了。。。。。你可以去掉TestAdvice裡的注釋看看各種運行結果，具體就不解釋咯，在代碼注釋裡。  首先，你必須對增強(advice)有所瞭解，增強就是在特定連接點執行的動作。advice contains the logic of your aspect。增強，分為4類：   1. **前增強**(before) 在連接點之前調用 2. **後增強**(after) 在連接點執行之後調用、 3. **環繞增強**(around) 完整控制整個方法流程，必須調用MethodInvocation的proceed促使真實操作發生 4. **異常增強** 針對某個異常拋出時調用   **書店，一個buyBook業務：**  package com.denny\_blue.springdemo.aop;  public interface BuyBook {        public void buyBook(String customer,String book)throws NoThisBookException; }  **實現此介面的一個業務物件，如果顧客要買<spring in action>就拋出NoThisBookException異常。**  package com.denny\_blue.springdemo.aop;  public class MyBuyBook implements BuyBook { public void buyBook(String customer,String book)throws NoThisBookException{    if(book.equals("<spring in action>"))     throw new NoThisBookException("對不起，沒有"+book+"存貨了！");    System.out.println(customer+",你好，你已經購買了一本"+book+"!"); } }  **自訂兩個異常類，其中NoThisBookException被拋出時將觸發MyThrowsAdvice調用：**  //NoThisBookException.java  package com.denny\_blue.springdemo.aop;  public class NoThisBookException extends RuntimeException { public NoThisBookException(String msg){    super(msg); }  }  //BuyBookException  package com.denny\_blue.springdemo.aop;  public class BuyBookException extends RuntimeException { public BuyBookException(String msg){    super(msg);    System.out.println(msg); }  }  **OK，接下來就是各類增強了，前增強，在業務方法buyBook之前調用，顯示歡迎資訊：**  package com.denny\_blue.springdemo.aop;  import java.lang.reflect.Method;  import org.springframework.aop.MethodBeforeAdvice;  public class MyBeforeAdvice implements MethodBeforeAdvice {  public void before(Method arg0, Object[] arg1, Object target)     throws Throwable {    String customer=(String)arg1[0];      //第2個參數組就是被增強的方法傳入的參數，本例中即customer,book    System.out.println("歡迎光臨!"+customer+"!");   //顯示歡迎資訊！，在buyBook方法前調用  }  }  **然後是後增強，當顧客已經買了書之後，顯示歡送資訊：**  package com.denny\_blue.springdemo.aop;  import java.lang.reflect.Method;  import org.springframework.aop.AfterReturningAdvice;  public class MyAfterAdvice implements AfterReturningAdvice {  public void afterReturning(Object arg0, Method arg1, Object[] arg2,     Object arg3) throws Throwable {     String customer=(String)arg2[0];      //同前增強一樣，參數組3為傳入參數，具體見spring doc    System.out.println("歡迎下次再來！"+customer+"!");   //顯示歡送資訊! }  }  **OK，有了上面兩個advice我們就能提供給顧客很好的服務態度了，等等？我們還有規則沒實現，不是說一名顧客只能買一本書嗎？OK，我們用環繞增強來解決，在環繞增強中保存一個HashSet，判斷顧客是否來過，來過就拋出一個異常，沒來過再放入此Set中：**  package com.denny\_blue.springdemo.aop;  import java.util.HashSet; import java.util.Set;  import org.aopalliance.intercept.MethodInterceptor; import org.aopalliance.intercept.MethodInvocation;  public class MyAroundAdvice implements MethodInterceptor {  private Set customers=new HashSet();    //保存購過書的顧客資訊 public Object invoke(MethodInvocation invocation) throws Throwable {         String customer=(String)invocation.getArguments()[0];         if(customers.contains(customer)){         throw new BuyBookException("對不起，一名顧客只能買一本打折書！");         }         Object result=invocation.proceed();   //調用MyBuyBook中的buyBook方法，即**真實操作**         customers.add(customer);         return result; }  }  **好了，最後一個，異常增強，當顧客要買<spring in action>時，我們的書店沒書了，請倉庫部門趕快訂貨！！**  package com.denny\_blue.springdemo.aop;  import org.springframework.aop.ThrowsAdvice;  public class MyThrowsAdvice implements ThrowsAdvice {     public void afterThrowing(NoThisBookException e){    //可以定義多個方法，只要傳入的參數是不同異常       System.out.print("通知倉庫，趕緊加訂書！");     }  }  好了沒？還沒，我們需要一個XML檔來組裝這些物件，來代理業務介面，完整的beans.xml如下，各元素的含義請自己查看spring reference  ?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "<http://www.springframework.org/dtd/spring-beans.dtd>">  <beans>      <bean id="myBuyBookTarget" class="com.denny\_blue.springdemo.aop.MyBuyBook"/>      <bean id="myBeforeAdvice" class="com.denny\_blue.springdemo.aop.MyBeforeAdvice"/>      <bean id="myAroundAdvice" class="com.denny\_blue.springdemo.aop.MyAroundAdvice"/>      <bean id="myAfterAdvice" class="com.denny\_blue.springdemo.aop.MyAfterAdvice"/>      <bean id="myThrowsAdvice" class="com.denny\_blue.springdemo.aop.MyThrowsAdvice"/>      <bean id="buyBook" class="org.springframework.aop.framework.ProxyFactoryBean">         <property name="proxyInterfaces">             <value>com.denny\_blue.springdemo.aop.BuyBook</value>         </property>         <property name="interceptorNames">             <list>                   <value>myBeforeAdvice</value>                   <value>myAfterAdvice</value>                   <value>myAroundAdvice</value>                   <value>myThrowsAdvice</value>             </list>         </property>         <property name="target">             <ref bean="myBuyBookTarget"/>         </property>      </bean> </beans>  我們先聲明所有的bean，通過  <bean id="myBuyBookTarget" class="com.denny\_blue.springdemo.aop.MyBuyBook"/> 確定將要被增強的物件——目標物件(target)，我們可以很容易地替換這個目標物件，只要它實現業務介面。代理的介面通過:  <property name="proxyInterfaces">             <value>com.denny\_blue.springdemo.aop.BuyBook</value> </property> 設定，然後是要用到一系列增強，注意，順序是很有影響的！你可以嘗試著改變順序看看結果：）  <property name="interceptorNames">             <list>                   <value>myBeforeAdvice</value>                   <value>myAfterAdvice</value>                   <value>myAroundAdvice</value>                   <value>myThrowsAdvice</value>             </list> </property>  **一切準備好了，我們來測試吧，GO GO GO**  package test;  import org.springframework.context.ApplicationContext; import org.springframework.context.support.FileSystemXmlApplicationContext;  import com.denny\_blue.springdemo.aop.BuyBook;  public class TestAdvice { public static void main(String args[]){    ApplicationContext ctx=new FileSystemXmlApplicationContext("/src/beans.xml");  //我的beans.xml放在專案下的src目錄，eclipse環境下，請自己調整    BuyBook buybook=(BuyBook)ctx.getBean("buyBook");    buybook.buyBook("jordan","<深入淺出hibernate>"); // buybook.buyBook("dennis","<spring in action>");    //去掉注釋即可觀察異常增強 // buybook.buyBook("jordan","<深入淺出hibernate>");   //去掉注釋即可觀察環繞增強  } }  一個簡單的例子，脫胎於spring reference，希望對您幫助：） | |