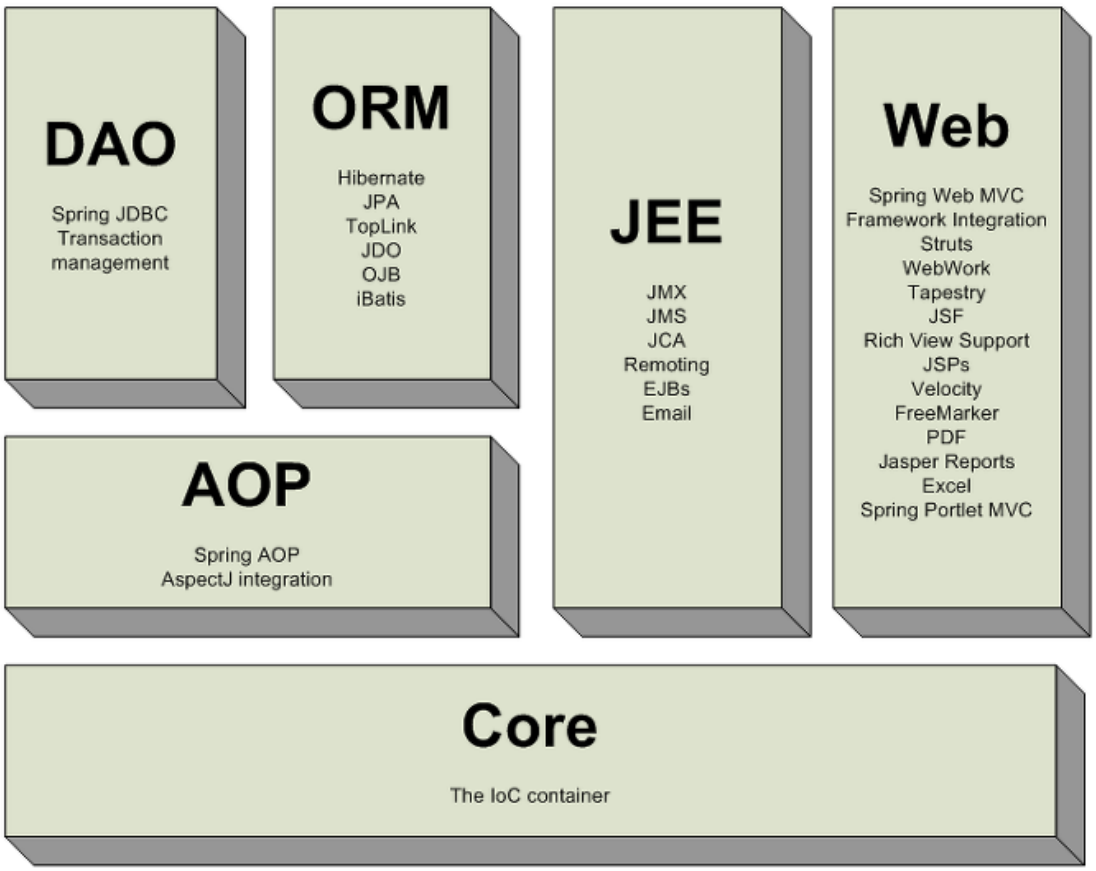
Web 全栈开发之 Spring 框架

13302010053　张亚中

- 什么是 Spring 框架

  Spring 是一个开源框架，Spring 是于 2003 年兴起的一个轻量级的 Java 开发框架，由 Rod Johnson 在其著作《Expert One-On-One J2EE Development and Design》中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。Spring 使用基本的 JavaBean 来完成以前只可能由 EJB 完成的事情。然而，Spring 的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何 Java 应用都可以从 Spring 中受益。Spring 的核心是控制反转（IoC）和面向切面（AOP）。简单来说，Spring 是一个分层的 JavaSE/EE full-stack(一站式) 轻量级开源框架。

- Spring 架构概况



Overview of the Spring Framework

Spring 最主要的特点有两个：IoC 和 AOP。它们解决了 J2EE 开发企业软件时经常碰到的问题：

1．对象太多如何管理
2．共同逻辑和业务逻辑纠缠在一起，错综复杂

在这次项目实践中，主要关注了 Spring 框架的三个方面：IoC、AOP 和数据库访问。借由 Spring 框架，我实现了一个有登录、注册以及用户信息修改的 Web 项目。为了更方便的控制页面跳转也业务逻辑，我还添加了 Spring MVC 框架。

- IoC

IoC 的全称是 Inversion of Control，中文称为控制反转或依赖注入。IoC 的实质是如何管理对象，传统方式是使用“new”来创建对象，但在企业应用开发的过程中，大量的对象创建都在程序中维护很容易造成资源浪费，并且不利于程序的扩展。

实现 IoC 通常有三种方式：

1. 利用接口或者继承，这种实现方式类似于平时提到的 lazy load。
2. 设置注入：IoC 容器使用属性的 setter 方法来注入被依赖的实例。
3. 构造注入：IoC 容器使用构造器来注入被依赖的实例。

在这次项目中，UserDao、UserService 均采用了 Spring 的依赖注入技术。UserDao 封装了程序与数据交互接口，用于数据访问，而 UserService 是面向业务的，封装了相关联的业务。这里先以 UserService 为例：

首先，web.xml 中添加 Spring 配置文件

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
```

然后，在 applicationContext.xml 中配置 UserService。从这里可以看出 Bean 于 Bean 之间的依赖关系放在配置文件里，而不是写在代码里。通过配置文件的指定，Spring 能够精确地为每个 Bean 注入属性。因此，配置文件里的<bean…/>元素的 class 属性值不能是接口，而必须是真正的实现类。

```
<bean name="userService" class="adweb.service.UserService">
    <property name="userDao" ref="userDao"></property>
</bean>
```

下面是业务封装类 UserService 的代码：

```java
public class UserService {

    private UserDao userDao;

    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
    public boolean addUser(User user) {
        if (userDao.query(user.getAccount()) != null) {
            return false;
        }
        userDao.save(user);
        return true;
    }
    public void updateUser(User user) {
        userDao.update(user);
    }
    public User findUserByAccount(String account) {
        return userDao.query(account);
    }
}
```

可以看到，UserService 类里有个 UserDao 的对象 userDao，这个变量被配置在 Spring 中，而不用在 UserService 里被实例化。当需要调用 userDao 时，这个实例将通过 Spring 容器负责注入。Spring 会自动接管配置文件中每个<bean…/>定义里的<property …/>元素定义，Spring 会在调用无参数的构造器、创建默认的 Bean 实例后，调用相应的 setter 方法为程序注入属性值。<property…/>定义的属性值将不再有该 Bean 来主动设置、管理，而是接受 Spring 的注入。

- 数据库操作

Spring JDBC 封装了数据库连接、增删查改、处理异常等功能，这让我们在项目开发中摒弃掉 JDBC API 单调乏味的底层细节处理工作。为了提高性能和稳定性，我还配置了 c3p0 连接池。有关数据库的用户操作，我都写在了 UserDao 里，同样利用的 Spring 的依赖注入技术。

applicationContext.xml 的相关配置如下：

```xml
<bean name="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="jdbc:mysql://10.131.226.239:3306/adweb"/>
    <property name="user" value="root" />
    <property name="password" value="******" />
    <property name="minPoolSize" value="10"/>
```

```
    <property name="maxPoolSize" value="20"/>
    <property name="maxIdleTime" value="3600" />
    <property name="initialPoolSize" value="3" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<bean id="userDao" class="adweb.dao.UserDao">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>
```

dataSource 与 com.mchange.v2.c3p0.ComboPooledDataSource 类相关联，其中包含诸如 url，user，password 等属性，用于依赖注入。

其中，minPoolSize 表示连接池的最小连接数，maxPoolSize 表示最大连接数。

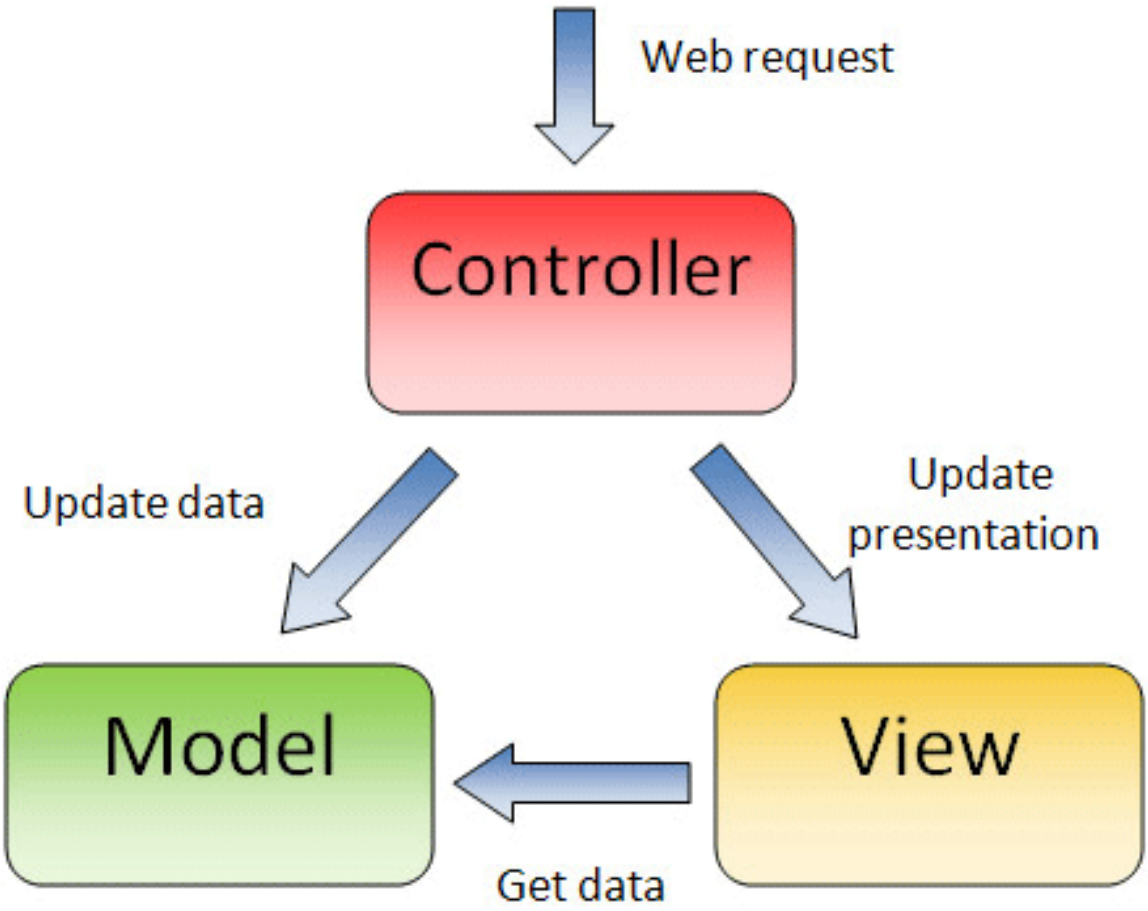
JdbcTemplate 在初始化时，会把 dataSource 作为参数注入。而 UserDao 中的 jdbcTemplate 属性，会引用 JdbcTemplate 的默认值。每个 Bean 的 id 属性是该 Bean 的唯一标识，程序通过 id 属性访问 Bean，Bean 与 Bean 的依赖关系也是通过 id 属性关联。

UserDao 的代码如下：

```
public class UserDao {

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) { this.jdbcTemplate = jdbcTemplate; }

    public void save(User user) {
        jdbcTemplate.update("INSERT INTO user(account, name, password, age)VALUES(?, ?, ?, ?)",
                new Object[]{user.getAccount(), user.getName(), user.getPassword(), user.getAge()},
                new int[]{java.sql.Types.VARCHAR, java.sql.Types.VARCHAR, java.sql.Types.VARCHAR,
                        java.sql.Types.INTEGER});
    }
```

- Spring MVC

  Spring MVC 是一种基于 Java 的实现了 Web MVC 设计模式的请求驱动类型的轻量级 Web 框架，它主要通过分离模型、视图及控制器在应用程序中的角色将业务逻辑从界面中解耦。通常，模型负责封装应用程序数据在视图层展示。视图仅仅只是展示这些数据，不包含任何业务逻辑。控制器负责接收来自用户的请求，并调用后台服务（manager 或者 dao）来处理业务逻辑。处理后，后台业务层可能会返回了一些数据在视图层展示。控制器收集这些数据及准备模型在视图层展示。MVC 模式的核心思想是将业务逻辑从界面中分离出来，允许它们单独改变而不会相互影响。



在 Spring MVC 应用程序中，模型通常由 POJO 对象组成，它在业务层中被处理，在持久层中被持久化。视图通常是用 JSP 标准标签库（JSTL）编写的 JSP 模板。控制器部分是由 dispatcher servlet 负责，在本教程中我们将会了解更多它的相关细节。

轻度使用 Spring MVC 是非常简单的，如下配置。

首先，在 web.xml 里添加一个 servlet，指向 Spring 框架提供的 DispatherServlet。并对任何符合 “/*” 形式的 url，交给该 servlet 处理。

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

然后，在 spring-mvc.xml 中，添加扫描注解的包、视图解析器、拦截器等：

```
<context:component-scan base-package="adweb.controller"/>
<context:component-scan base-package="adweb.service"/>
<context:component-scan base-package="adweb.dao"/>

<bean id="jspViewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".html"/>
</bean>

<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/user.do"/>
        <bean class="adweb.interceptor.UserInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

controller 包即为控制层，用于响应前端请求并处理。在这个项目中，只有对 User 的处理，即只写了一个 UserController，结构如下：

```
@Controller
@RequestMapping(value = "/user.do")
public class UserController {

    @Resource
    private UserService userService;

    @RequestMapping(params = "method=login")
    public String login(String account, String password, HttpSession httpSession) throws JSONException {...}

    @RequestMapping(params = "method=register")
    public String register(String account, String password, String name, int age, HttpSession httpSession) throws JSONException {...}

    @RequestMapping(params = "method=edit")
    public String edit(String account, String password, String name, int age, int id, HttpSession httpSession) throws JSONException {...}

    @ResponseBody
    @RequestMapping(params = "method=getinfo", produces = "text/plain; charset=utf-8")
    public String edit(HttpSession httpSession) throws JSONException {...}
}
```

类前面的注解@Controller 表示这是一个 Controller，@RequestMapping 定义了类或方法会响应哪个 url 的请求。params = “method=login” 规定了请求参数中必须包含 method 标签，并且其值要等于 login。produces 规定了某个方法返回值的形式和编码。标注了 @ResponseBody 的方法会直接返回给前端数据，而不会经过视图解析器转为某个页面，一般用于响应 ajax 请求。

- 其它

拦截器（Interceptor）：

```java
public class UserInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o) throws Exception {
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, ModelAndView modelAndView) throws Exception {
    }

    @Override
    public void afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) throws Exception {
    }
}
```

- preHandle 方法是进行处理器拦截用的，顾名思义，该方法将在 Controller 处理之前进行调用，SpringMVC 中的 Interceptor 拦截器是链式的，可以同时存在多个 Interceptor，然后 SpringMVC 会根据声明的前后顺序一个接一个的执行，而且所有的 Interceptor 中的 preHandle 方法都会在 Controller 方法调用之前调用。SpringMVC 的这种 Interceptor 链式结构也是可以进行中断的，这种中断方式是令 preHandle 的返回值为 false，当 preHandle 的返回值为 false 的时候整个请求就结束了。
- postHandle 方法只会在当前这个 Interceptor 的 preHandle 方法返回值为 true 的时候才会执行。postHandle 是进行处理器拦截用的，它的执行时间是在处理器进行处理之后，也就是在 Controller 的方法调用之后执行，但是它会在 DispatcherServlet 进行视图的渲染之前执行，也就是说在这个方法中你可以对 ModelAndView 进行操作。这个方法的链式结构跟正常访问的方向是相反的，也就是说先声明的 Interceptor 拦截器该方法反而会后调用。
- afterCompletion 方法也是需要当前对应的 Interceptor 的 preHandle 方法的返回值为 true 时才会执行。该方法将在整个请求完成之后，也就是 DispatcherServlet 渲染了视图执行，这个方法的主要作用是用于清理资源的，当然这个方法也只能在当前这个 Interceptor 的 preHandle 方法的返回值为 true 时才会执行。

监听器（Listener）：

```java
public class SessionListener implements HttpSessionListener {
    @Override
    public void sessionCreated(HttpSessionEvent httpSessionEvent) {
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent httpSessionEvent) {
    }
}
```

任何一个 Session 被创建或者销毁时，都会通知 SessionListener 这个类。通常用这个功能来监测实时在线人数、用户流量等。当然，前提是必须在 web.xml 文件中做相关的配置工作。如下面的配置代码：

```
<listener>
    <listener-class>adweb.listener.SessionListener</listener-class>
</listener>
```