软件架构设计课程Project

2018.01.11

Web应用框架的设计实现与测试

参考自Spring MVC架构

**目 录**

[**第一章 项目综述 1**](#_Toc503701248)

[1.1 项目情况 1](#_Toc503701249)

[**第二章 框架概览 3**](#_Toc503701251)

[2.1 架构图 3](#_Toc503701252)

[2.1.1 前端控制器 3](#_Toc503701253)

[2.1.2 静态资源控制器与JSP控制器 4](#_Toc503701254)

[2.1.3 控制器映射器 5](#_Toc503701255)

[2.1.4 页面控制器 5](#_Toc503701256)

[2.1.5 视图解析器与JSON解析器 5](#_Toc503701257)

[2.2 框架使用 6](#_Toc503701258)

[**第三章 DI设计与实现 7**](#_Toc503701259)

[3.1 依赖注入与控制反转 7](#_Toc503701260)

[3.2 框架中的依赖注入 8](#_Toc503701261)

[**第四章 AOP设计与实现 10**](#_Toc503701262)

[4.1 动态代理 10](#_Toc503701263)

[4.1.1 JDK动态代理 10](#_Toc503701264)

[4.1.2 CGLib动态代理 10](#_Toc503701265)

[4.2 框架中的切面编程 11](#_Toc503701266)

[4.3 框架中的事务控制 12](#_Toc503701267)

[4.3.1 事务的特性 12](#_Toc503701268)

[4.3.2 事务的实现方式 12](#_Toc503701269)

[**第五章 Demo测试与分析 16**](#_Toc503701270)

[5.1 Demo编写 16](#_Toc503701271)

[5.2 测试与分析 17](#_Toc503701272)

第一章 项目综述

# 1.1 项目情况

为了更加高效的开发Java Web系统，Servlet、JSP等技术逐渐被各种框架封装，它们简化了开发难度的同时，优化了系统的结构。目前，Spring系列框架被广泛应用，配合Spring、SpringMVC以及MyBatis，可以有效地构造出组织结构良好的Java Web系统。

借鉴SpringMVC的设计思想，本人参考书籍资料开发了一个轻量级Java Web框架，它实现了Spring框架的两大核心思想：依赖注入（DI，又称控制反转IoC）和切面编程（AOP），与SpringMVC的使用相似，采用注解的方式进行配置，操作简便，易于使用，特别适用于小型Web项目的开发。

本人使用该框架构造了一个Java Web测试项目，对框架的主要功能进行了测试，结果显示：框架没有出现异常情况，所有功能均达到了预期目标。

第二章 框架概览

# 2.1 架构图

如图2-1所示为框架的架构图，该图同时描述了一个访问请求的基本流程。

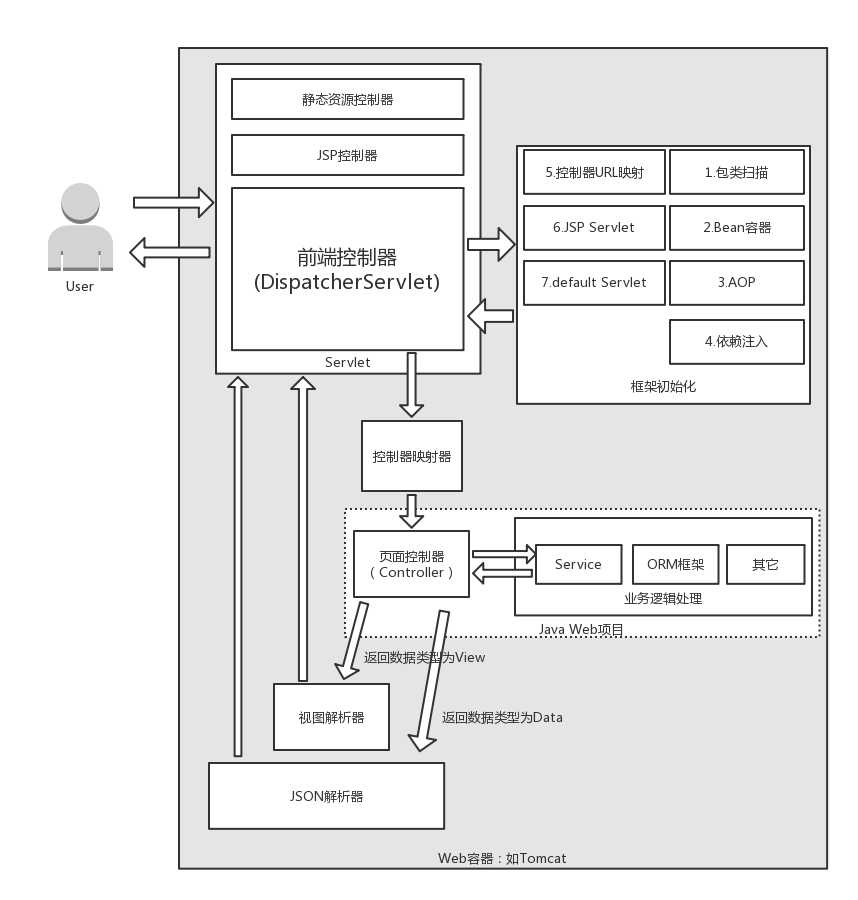


图2-1 框架架构图

## 2.1.1 前端控制器

前端控制器即DispatcherServlet，它是三个提供服务的Servlet之一（其它两个分别是：静态资源控制器、JSP控制器）。当用户请求的URL以/servlet/为开头，则该请求会被定位到DispatcherServlet处理。

DispatcherServlet首次注册时，会执行框架初始化工作：

1. 对Web项目配置文件中配置的基础包（base package）进行扫描，得到所有的项目类并加载，生成Class类实例；（扫描包时，会针对文件夹和jar包采取不同的加载方式）
2. 创建Bean容器，判断每个Class类实例是否带有@Controller注解或者@Service等注解，为符合要求的Class类实例生成指定类的实例对象，放在Map中保存；
3. 扫描Web项目中的切面编程逻辑类（带有@Aspect注解并且继承自框架的AspectProxy类），判断切面逻辑所要修饰的类，并通过动态代理技术生成该类的代理类，最后将该代理类实例放到Bean容器中保存；
4. 扫描Bean容器中的类实例，判断属性中是否带有@Inject注解，将带有该注解的属性，采用Bean容器中匹配的类实例初始化，即依赖注入；
5. 扫描被@Controller标注的Class类实例，提取带有@Action注解的方法，将@Action注解中的参数（即请求方式和URL）与对应方法进行映射；
6. 配置处理JSP页面的Servlet，用于处理JSP请求；
7. 配置处理静态资源的Servlet，用于处理静态资源请求。

框架初始化应当按照以上顺序执行，例如第三步AOP和第四步依赖注入不能顺序颠倒，否则注入的对象实例有误。

今后用户以/servlet/为URL开头请求访问时，DispatcherServlet会通过控制器映射器定位到指定的Controller为用户处理并返回数据。（Controller等业务逻辑部分需要Web项目编写）

## 2.1.2 静态资源控制器与JSP控制器

静态资源和JSP页面不通过DispatcherServlet来处理，会被分别定位到静态资源控制器和JSP 控制器来处理。

当请求的URL以/WEB-INF/view/为开头，则该请求会被定位到JSP控制器来处理。（该请求由于访问了WEB-INF文件夹，所以只可能由Web应用后台来进行请求转发）

当请求的URL以/asset/为开头，则该请求会被定位到静态资源控制器来处理，这类请求是由前端页面自动调用的，静态资源一般包含JS、CSS、图片等文件。

## 2.1.3 控制器映射器

控制器映射器将用户请求URL与Handler进行匹配（Handler包含特定控制器Controller和其内部的特定方法Action），如果请求方式和URL匹配到符合要求的Handler，则提取出该Controller的指定Action方法备用。

映射器接下来将请求URL中的请求参数和请求体中的请求参数提取出来，作为Action方法的传入参数。最后，使用反射技术调用Action方法。

## 2.1.4 页面控制器

页面控制器即Controller，由Web项目编写。类比SpringMVC中带有@Controller注解的类，在这里它们完全相同。

Controller接收Param类型的参数，该参数由控制器映射器构造；返回View类型或者Data类型的参数，返回给控制器映射器。

## 2.1.5 视图解析器与JSON解析器

得到Controller返回的数据后，控制器映射器会根据返回参数的类型，即View或Data，将数据分别定位到视图解析器和JSON解析器。

视图解析器将根据View中的JSP页面地址来进行重定向或者请求转发处理，最终生成页面返回给用户浏览器。

JSON解析器将根据Data中的数据，采用Jackson进行处理，转换成JSON数据并返回给用户。

# 2.2 框架使用

根据图2-1所示，Java Web项目和框架共同运行在Web容器上（如Tomcat），其中Java Web项目所在的位置用虚线框表示。用户不需要关注业务以外的事情，因为这些都交由框架来处理。

开发Web项目时，新建Maven Java Web项目，将框架jar包导入，配置框架需要读取的配置文件（smart.properties），该配置文件的示例内容如代码2-1所示。

代码2-1 示例配置文件的内容

|  |
| --- |
| #数据库驱动  smart.framework.jdbc.driver=com.mysql.jdbc.Driver  #数据库地址和数据库名  smart.framework.jdbc.url=jdbc:mysql://localhost:3306/demo  #数据库账号  smart.framework.jdbc.username=root  #数据库密码  smart.framework.jdbc.password=zhangshuhao  #创建的Web项目的基础包名  smart.framework.app.base\_package=org.smart4j.smart.demo  #Web项目的JSP文件的基础路径  smart.framework.app.jsp\_path=/WEB-INF/view/  #Web项目静态资源文件的基础路径，比如JS CSS 图片等  smart.framework.app.asset\_path=/asset/ |

配置完成后，便可以使用@Controller、@Service、@Inject以及@Action等注解，像使用SpringMVC一样开发项目，不需要在web.xml中进行额外配置。

同时，项目可以结合使用MyBatis框架操作数据库，使数据库操作更加方便。

第三章 DI设计与实现

本章对依赖注入（DI，又称控制反转IoC）的原理以及在框架中的代码实现进行说明。依赖注入及其实现原理——反射，贯穿在整个框架的运行过程中，是框架的核心之一。

# 3.1 依赖注入与控制反转

随着面向对象分析与设计的发展，一个良好的设计，核心原则之一就是将变化隔离，即：高内聚，低耦合。使得变化部分发生变化时，不变部分不受影响。为了做到这一点，要利用面向对象中的多态性。使用多态性后，客户类不再直接依赖服务类，而是依赖于一个抽象的接口，这样，客户类就不能在内部直接实例化具体的服务类。但是，客户类在运作中又客观需要具体的服务类提供服务，因为接口是不能实例化去提供服务的。就产生了“客户类不准实例化具体服务类”和“客户类需要具体服务类”这样一对矛盾。为了解决这个矛盾，开发人员提出了一种模式：客户类定义一个注入点，用于服务类（实现具体类）的注入，而客户类的客户类（Program，即测试代码）负责根据情况，实例化服务类，注入到客户类中，从而解决了这个矛盾。

依赖注入（Dependency Injection）就是这样一个过程：由于某客户类只依赖于服务类的一个接口，而不依赖于具体服务类，所以客户类只定义一个注入点。在程序运行过程中，客户类不直接实例化具体服务类实例，而是客户类的运行上下文环境或专门组件负责实例化服务类，然后将其注入到客户类中，保证客户类的正常运行。

当一个类的实例需要另一个类的实例协助时，在传统的程序设计过程中，通常有调用者来创建被调用者的实例。如果采用依赖注入原则，创建被调用者的实例的工作不再由调用者完成，而是由IOC容器来完成，这就是“控制反转”的意思，然后，将其注入调用者，因此也称为“依赖注入”。

控制反转和依赖注入之间的关系其实就是同一个概念的不同角度描述依赖注入是从应用程序的角度在描述，可以把依赖注入描述完整点：应用程序依赖容器创建并注入它所需要的外部资源；而控制反转是从容器的角度在描述，描述完整点就是说容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。

# 3.2 框架中的依赖注入

项目依赖注入的核心就是实现Controller，通过@Controller注解可以来定义Controller类。在该类中，可通过@Inject注解定义一系列Service成员变量（被@Service注解的类），这就是“依赖注入”。此外，有一系列被@Action注解所定义的方法，在这些Action方法中，调用了Service成员变量的方法来完成具体的业务逻辑。若返回View对象，则表示JSP页面；若返回Data对象，则表示一个JSON数据。

在整个框架的实现中，最核心的类就是请求转发器（DispatcherServlet）。我们通过DispatcherServlet来处理所有请求，根据请求信息从ControllerHelper中获取对应的Action方法，然后使用反射技术调用Action方法，同时需要具体的传入方法参数，最后拿到返回值并判断返回值的类型，进行相应的处理。而在处理请求之前，需要对整个框架进行初始化。

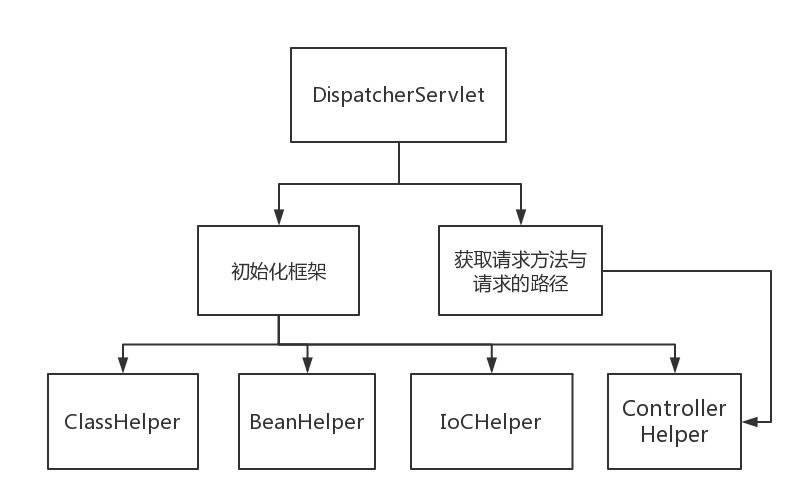


图3-1 DispatcherServlet的功能结构

ClassHelper的作用是作为类加载器来加载项目基础包名下的所有类，比如使用了某注解的类，或者实现了某接口的类，再或者继承了某父类的所有子类。使用ClassHelper类可以获取所加载的类，但无法通过类来实例化对象，所以我们需要BeanHelper。

BeanHelper可以通过ClassHelper获取应用包名下的所有bean类，包括Service、Controller等。然后使用循环调用Reflection中的newInstance方法，根据类来实例化对象，最后将每次创建的对象存放在一个静态的Map<Class<?>, Object>中。需要时通过该Map的key（类名）去获取所对应的value（Bean对象）。

针对@Inject注解的依赖注入，先通过BeanHelper获取所有Bean Map，然后遍历这个映射关系，分别取出Bean类与Bean实现。进而通过反射获取类中的所有成员变量。继续遍历这些成员变量，在循环中判断当前成员变量是否具有Inject注解，若带有该注解，则从Bean Map中根据Bean类取出Bean实例，然后修改当前成员变量的值。

最后，我们实现一个ControllerHelper类，在ControllerHelper中封装了一个ActionMap，通过它来存放Request（包含请求方式和URL）与Handler（对应的处理函数）之间的关系，然后通过ClassHelper来获取所有带有@Controller的注解，接着遍历这些Controller类，从Action注解中提取URL，最后初始化Request与Handler之间的映射关系。

我们搭建的框架中，定义了一系列注解：通过@Controller注解来定义Controller类；通过@Inject注解来实现依赖注入；通过@Action注解来定义Action方法。通过一列的Helper类来初始化MVC框架；通过DispatcherServlet来处理所有的请求；根据请求方法与请求路径来调用具体的Action方法，判断Action方法的返回值，若为View类型，则跳转到JSP页面，若为Data类型，则返回JSON数据。

在程序中我们没有自己去new对象，而是交给了我们自己写出来的方法去new。我们的代码会根据我们自己写的配置文件得到配置文件的信息，然后去new配置文件中写好的对象，将依赖注入，如果我们要使用的实现类出现了修改，那么只需要修改配置文件中的信息就好了，大大减少了修改代码的成本。

具体代码详见附件中的smart.zip或lightframework.zip。

第四章 AOP设计与实现

本章对切面编程（AOP）的原理以及在框架中的代码实现进行说明。切面编程及其实现原理——动态代理，在数据库事务管理、性能监控以及增强指定方法的功能等方面被广泛使用，是框架的核心之一。

# 4.1 动态代理

代理，或称为Proxy，意思就是别人替代你去处理一些事物。它在程序开发中起到了非常重要的作用，比如AOP就是针对代理的一种应用。此外在设计模式中，还有一个“代理模式”。

## 4.1.1 JDK动态代理

JDK提供了动态代理方案DynamicProxy类，它需要一个Object类型的target变量，这就是被代理的目标对象，通过构造函数来初始化。DynamicProxy实现了InvocationHandler接口，所以DynamicProxy必须实现该接口中的invoke方法。invoke方法在生成的目标代理对象中，每次执行方法时被调用，因此invoke中的代码就是为原方法扩展的一些功能代码。

## 4.1.2 CGLib动态代理

JDK实现动态代理需要目标类通过接口定义业务方法，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

CGLib使用方法：实现MethodInterceptor接口中的方法intercept，该方法是具体的增强代码与原方法代码调用，原方法调用使用MethodProxy的invokeSuper方法实现。获得生成的代理实例采用Enhancer类的create方法，代理实例调用方法时会使用intercept方法。

CGLib创建的动态代理对象性能比JDK创建的动态代理对象的性能高不少，但是CGLib在创建代理对象时所花费的时间却比JDK多得多，所以对于单例的对象，因为无需频繁创建对象，用CGLib合适，反之，使用JDK方式要更为合适一些。同时，由于CGLib由于是采用动态创建子类的方法，对于final方法，无法进行代理。

# 4.2 框架中的切面编程

面向切面编程（AOP是Aspect Oriented Program的首字母缩写） ，我们知道，面向对象的特点是继承、多态和封装。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。

AOP技术中将这些通用性功能称为方面（Aspect）或切面（Pointcut）。在JDK提供的动态代理AOP和CGLib等字节码的AOP中，方面通过拦截器(Interceptor)来实现。具体解释就是，在对Java方法调用时，AOP机制能自动进行方法拦截，允许你在方法调用前，调用后，以及执行异常时添加特定的代码，来完成需要的功能。

框架中采用了CGLlib实现切面编程，切面编程相关的类图如图4-1所示。由ProxyManager生成目标代理类实例，可以对目标类赋予多个切面逻辑，这些切面逻辑组成ProxyChain代理链，按顺序作用于目标类的方法中。具体切面逻辑由使用框架的Java Web项目实现。

每个切面逻辑继承自AspectProxy抽象类，具体实现的切面逻辑通过实现或重写AspectProxy类中的方法，使得切面逻辑有针对性的对部分方法进行修改，而其它方法不被修改；同时，也可以使得切面逻辑对指定方法的调用前、调用后进行扩展操作，使用灵活、简便。

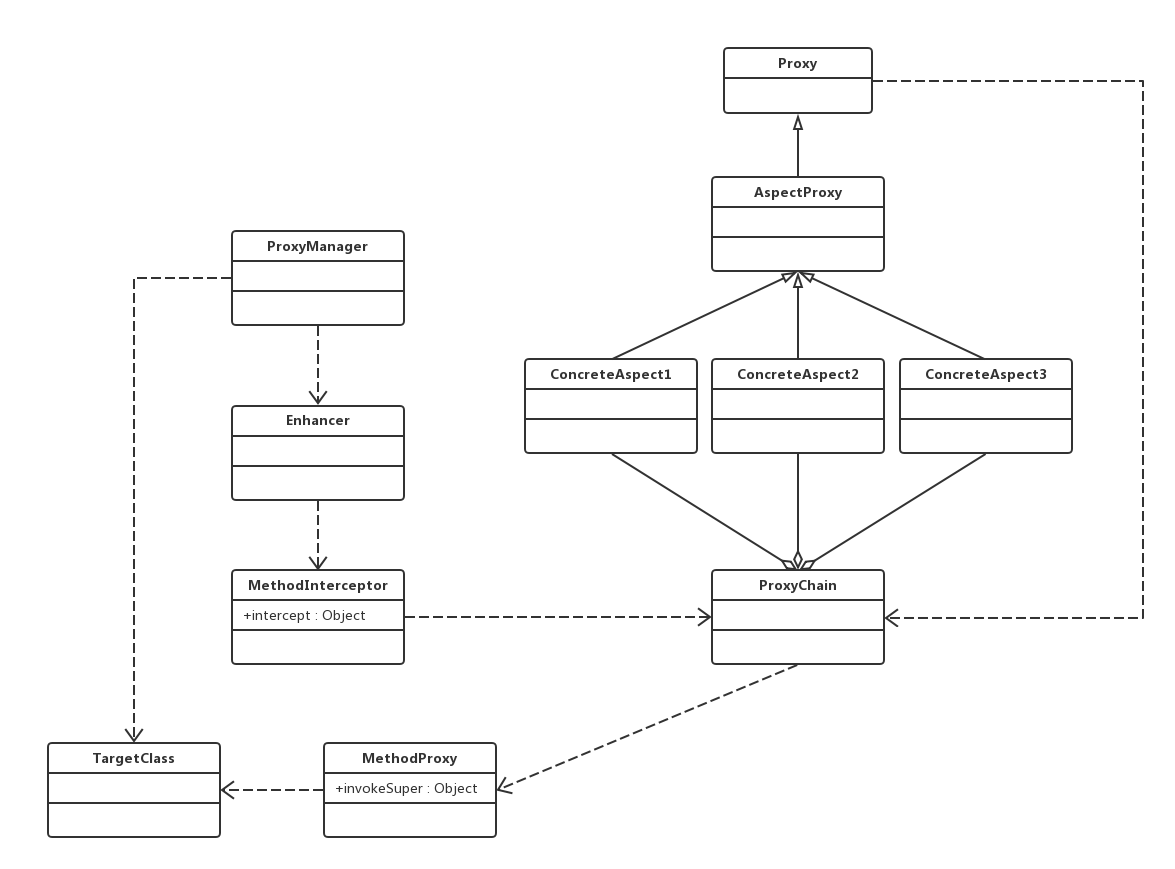


图4-1 切面编程相关的类图

# 4.3 框架中的事务控制

## 4.3.1 事务的特性

原子性：一个事务中所有对数据库的操作是一个不可分割的操作序列，要么全做，要么全不做。

一致性：数据不会因为事务的执行而遭到破坏。

隔离性：一个事务的执行，不受其他事务（进程）的干扰。即并发执行的各事务之间互不干扰。

持久性：一个事务一旦提交，它对数据库的改变将是永久的。

## 4.3.2 事务的实现方式

实现方式共有两种：编码方式；声明式事务管理方式。

基于AOP技术实现的声明式事务管理，实质就是：在方法执行前后进行拦截，然后在目标方法开始之前创建并加入事务，执行完目标方法后根据执行情况提交或回滚事务。

声明式事务管理又有两种方式：基于XML配置文件的方式；另一个是在业务方法上进行@Transaction注解，将事务规则应用到业务逻辑中。

项目中框架采用声明式事务管理方式，采用@Transaction注解，部分源码如代码4-1所示。

代码4-1 部分数据库相关代码

|  |
| --- |
| private static final Logger LOGGER = LoggerFactory.getLogger(DatabaseHelper.class);  private static final String url = ConfigHelper.getJdbcUrl();  private static final String username = ConfigHelper.getJdbcUsername();  private static final String password = ConfigHelper.getJdbcPassword();  private static ThreadLocal<Connection> connContainer = new ThreadLocal<Connection>();  static {  try{  Class.forName(ConfigHelper.getJdbcDriver());  }catch(ClassNotFoundException e){  LOGGER.error("class not found", e);  throw new RuntimeException(e);  }  }  //获得数据库连接  private static Connection getConnection() throws Exception{  Connection conn = connContainer.get();  if(conn==null){  conn = DriverManager.getConnection(url,username,password);  connContainer.set(conn);  }  return conn;  }  //开启事务  public static void beginTransaction() throws Exception{  Connection conn = getConnection();  if(conn!=null){  try{  conn.setAutoCommit(false);  }catch (SQLException e){  LOGGER.error("begin transaction failure", e);  throw new RuntimeException(e);  }  }  }  //提交事务  public static void commitTransaction() throws Exception{  Connection conn = getConnection();  if(conn!=null){  try{  conn.commit();  conn.close();  }catch(SQLException e){  LOGGER.error("commit transaction failure", e);  throw new RuntimeException(e);  }  }  }  //回滚事务  public static void rollbackTransaction() throws Exception{  Connection conn = getConnection();  if(conn!=null){  try {  conn.rollback();  conn.close();  }catch (SQLException e){  LOGGER.error("rollback transaction failure", e);  throw new RuntimeException(e);  }  }  } |

具体代码详见附件中的smart.zip或lightframework.zip。

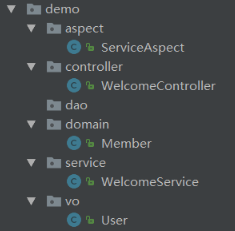
第五章 Demo测试与分析

在框架上创建了一个Java Web测试项目，名称为Demo，用来测试框架的运行结果是否符合预期。

# 5.1 Demo编写

测试项目在标注@Controller注解的类中通过@Inject注解将特定类实例动态注入进来（被注入的类通过@Service注解，称为Service类）；在Service类中，会使用框架对数据库进行操作；返回给用户的信息包括JSP页面或者JSON数据；同时，项目创建了切面逻辑用于修饰指定的类，增强它们的方法，获得每个方法的执行时长。至此，一个基本的Java Web便搭建出来，包含了主要业务流程且对框架的各项功能进行了测试。

项目结构如图5-1所示，具体代码详见附件。



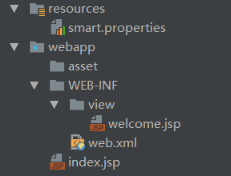


图5-1 Web项目结构

# 5.2 测试与分析

1. 输入URL：http://localhost:8080/smart/index.jsp或者http://localhost:8080/ smart/

返回index.jsp页面，本次请求没有经过DispatcherServlet处理，而是被JSP Servlet处理。

可以说明：三个Servlet：JSP Servlet、Asset Servlet以及DispatcherSelvert分工正确，没有出现拦截不该处理的URL，并导致404找不到页面的问题。

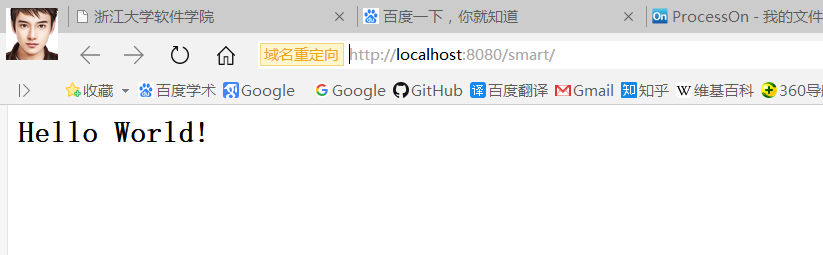


图5-2 首页访问成功

2. 输入URL：http://localhost:8080/smart/servlet/welcomeTest?viewId=1

该请求由DispatcherServlet处理，定位到Web项目的WelcomeController，通过控制器映射器，找到@Action注解参数为”get:/welcomeTest”的方法（称为方法1，源码见代码5-1），将得到的参数：viewId=1封装到Param类中，作为方法1的传入参数，并通过反射调用。

WelcomeController类的WelcomeService类型的属性由依赖注入功能注入实例，并在方法1中使用。

|  |
| --- |
| @Inject  private WelcomeService welcomeService;  @Action("get:/welcomeTest")  public View welcomeTest(Param param){  long viewId = param.getLong("viewId");  if(viewId==1){  System.out.println("\*\*\*\*\*welcome.jsp\*\*\*\*\*");  //当viewId为1时，返回welcome.jsp页面  String name = welcomeService.getUserName();  return new View("welcome.jsp").addModel("name",name);  }else{  //其它情况返回index.jsp页面（通过重定向方式）  System.out.println("\*\*\*\*\*index.jsp\*\*\*\*\*");  return new View("/index.jsp");  }  } |

方法1返回参数为View类型，因此会由视图解析器处理，视图解析器将JSP页面和传入的数据进行处理，得到最终返回给用户的页面，如图5-3所示。

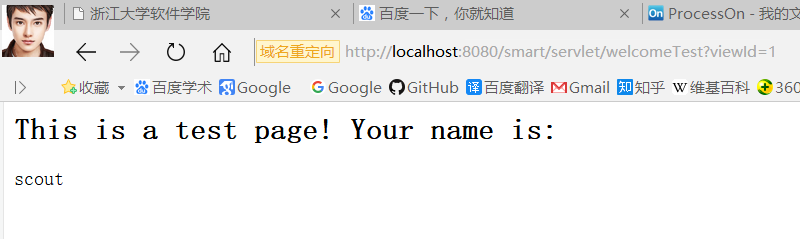


图5-3 成功返回的JSP页面

3. 输入URL：http://localhost:8080/smart/servlet/welcomeTest?viewId=2

与测试2中调用方法一样，但由于传入参数viewId=2，所以会执行不同的逻辑返回不同的页面。注意到在代码5-1中，当viewId!=1时，返回的是new View(“/index.jsp”)，页面名称以/开头，所以在视图解析器中会进行特殊处理，进行重定向操作，而不是请求转发。

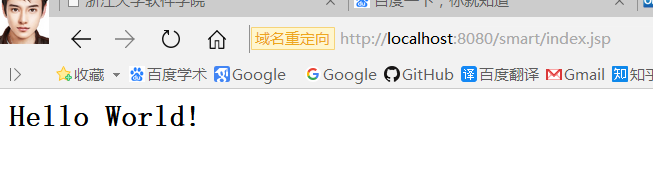


图5-4 重定向到index.jsp跳转成功

测试2和测试3可以说明：DispatcherServlet运行正确，需要的属性通过依赖注入正确的注入到DispatcherServlet类实例中，且@Action注解成功定位到指定处理方法。其它注解也运行正常，各项处理器运行正确，返回数据时，可以通过视图解析器正确构造呈献给用户的最终页面。

4. 输入URL：http://localhost:8080/smart/servlet/testJson?name=Dean&id=12

&age=23

本次测试访问指定处理方法（带有注解@Action("get:/testJson")），解析传入参数：name=Dean id=12 age=23。方法处理逻辑是只接受name参数和age参数，并将参数以JSON格式返回，如图5-5所示，Web项目通过框架的帮助，成功返回JSON数据。

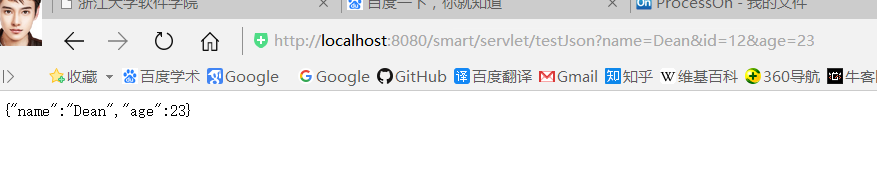


图5-5 成功返回JSON数据

5. 输入URL：http://localhost:8080/smart/servlet/testDb

本次测试访问指定处理方法（带有注解@Action("get:/testDb")），没有传入参数。方法会调用WelcomeService类实例，并通过框架访问本地数据库，得到信息后，以JSON数据的形式返回给用户，如图5-6所示。

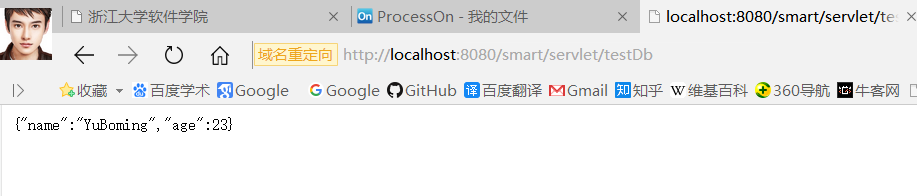


图5-6 数据库数据成功以JSON形式返回

6. 本Web项目中，切面逻辑代码对标有@Service注解的类的所有方法进行增强，使这些方法的执行时长可以显示到控制台，如图5-7所示是测试5中方法的执行时长展示。

其中“\*\*\*\*\*切面逻辑执行，当前时间：1515765659864 \*\*\*\*\*\*”与“\*\*\*\*\*该方法执行共花时间：1229 \*\*\*\*\*”分别是在目标方法执行前后执行的增强代码。

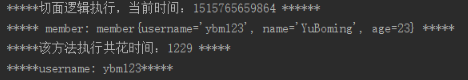


图5-7 getMemberInfo方法执行所花时间

综上所述，Web项目在框架的辅助下，正确的完成了所有测试用例，证明了框架的有效性，且开发过程简便，达到预期目标。