

Web 技术大作业

姓 名: _____ 鲍骞月 _____

学 号: _____ 1607104130 _____

专 业: _____ 数字媒体技术 _____

学 院: _____ 大数据学院 _____

2019 年 5 月

目 录

第一章 概述.....	1
第二章 功能需求.....	3
第三章 分层框架.....	7
第四章 设计实现.....	11
第五章 实验结果及分析.....	30
第六章 设计总结.....	34

1. 概述

1.1 技术可行性

Java Web 是使用 Java 技术来解决相关 web 互联网领域的技术总和。web 包括: web 服务器和 web 客户端两部分。Java 在服务器端的应用非常的丰富, 比如 Servlet, JSP 和其他强大的 Web 框架 (Spring 家族) 等等。Java 技术对 Web 领域的发展注入了强大的动力, 正是 Java 在 web 领域以及服务器端开发的整体性, 生态完整性, 易扩展性, 才让 Java 这门语言越来越受到开发者们的青睐。

Java 的 Web 框架虽然种类繁多, 但基本也都是遵循特定的底层设计原则的: 使用 Servlet 或者 Filter 拦截请求, 使用 MVC 的设计架构, 使用约定, XML 或 Java Annotation 实现配置, 运用 Java 面向对象的特点, 面向对象的实现请求和响应的流程, 支持 Jsp, Freemarker, Velocity 等视图。

最基础的 Java Web 框架组件便是 Servlet, 是用 Java 编写的服务器端程序。其主要功能在于交互式地浏览和修改数据, 生成动态 Web 内容。狭义的 Servlet 是指 Java 语言实现的一个接口, 广义的 Servlet 是指任何实现了这个 Servlet 接口的类, Servlet 运行于支持 Java 的应用服务器中, 比如 Tomcat。从实现上讲, Servlet 可以响应任何类型的请求, 但绝大多数情况下 Servlet 只用来扩展基于 HTTP 协议的 Web 服务器。Servlet 的核心工作流程就是 Request 和 Response 和服务器、客户端的信息交互。

但是原生的 Servlet 构建的 web 应用有很多问题, 比如它只能通过硬编码的方式, 虽然也能够将一个 servlet 注册到容器中, 并且也能设置 urlMapping, 但是这种方式不好进行修改。正常的 web 方式是通过在项目中配置 web.xml, 然后给每一个 servlet 配置一个 urlMapping, 当项目业务非常复杂时, web.xml 文件会非常臃肿。

基于上述问题, 本项目的 Web 框架技术选型为 SpringMVC, SpringMVC 的方式是先定义一个全局 Servlet (DispatcherServlet) 接收所有的 Http 请求, 然后通过 Java 的类加载器扫描项目输出包下的所有类, 然后通过注解过滤出所有的业务 Controller 类, 通过 Java 反射机制动态实例化相应的 mappinghandler, 然后进行 url 的响应, 如果响应的结果还包括视图的显示, 则需要封装页面的显示信息, 使用 DispatcherServlet 进行界面的跳转。

为了实现学生信息管理功能，项目首先仿照 SpringMVC 模拟实现了一个轻量级的 Web 框架，称为“mini-SpringMVC”，然后在该框架的基础上进行学生信息管理应用程序的开发，应用程序使用 MVC 模式，数据库交互业务使用 DAO, Service 分层框架进行实现，降低了模块间的耦合性，并且提高了数据业务的可拓展性，前端页面显示使用 JSP 和 Bootstrap 前端框架实现，性能高效，同时可以使用 Bootstrap 核心 CSS 库进行界面的渲染。

1.2 开发环境

服务器：Tomcat8.5.38

开发模式：C/S

开发语言：Java、XML、SQL、JSP、CSS

开发工具：Intellij IDEA 2018.05

MYSQL

Navicat for MYSQL

项目构建工具：Gradle 4.10

前端框架：Bootstrap 3.3.7

开发系统环境：Mac OS 10.14、JDK 1.8

数据库：MySQL 5.7.14

1.3 项目托管 Github 仓库

<https://github.com/shentibeitaokongle/Web-Road>

2. 功能需求

2.1 项目概要

该系统主要分为两大模块，框架模块（framework）和应用模块（App）。

框架模块主要负责 Web 服务器的集成，Web 注解定义，Bean 工厂设计，项目类扫描器的实现，注解处理器 DispatcherServlet 的实现等功能。

应用模块主要实现学生信息管理功能，主要包括管理员登录、全部学生数据展示、修改单个学生信息、增加学生记录和删除学生记录功能。

2.2 项目需求

需求分析概要表（框架模块）如表 2.1 所示：

表 2.1 需求分析概要表（框架模块）

需求编号	需求内容
X1	Web 注解（Controller, RequestMapping, Request Param）定义
X2	Bean 工厂实现，处理反射操作
X3	项目类扫描器 ClassScanner 实现
X4	注解处理器 DispatcherServlet 实现
X5	Servlet 存储器 WebContext 实现
X6	视图 View 及视图数据 ViewData 实现

需求分析概要表（应用模块）如表 2.2 所示：

表 2.2 需求分析概要表（应用模块）

需求编号	需求内容
X1	管理员登录

X2	全部学生数据展示
X3	修改单个学生信息
X4	增加学生记录
X5	删除学生记录
X6	根据学生姓名查找

2.3 系统功能结构

如图 2.1 所示为框架模块（framework）功能结构图。

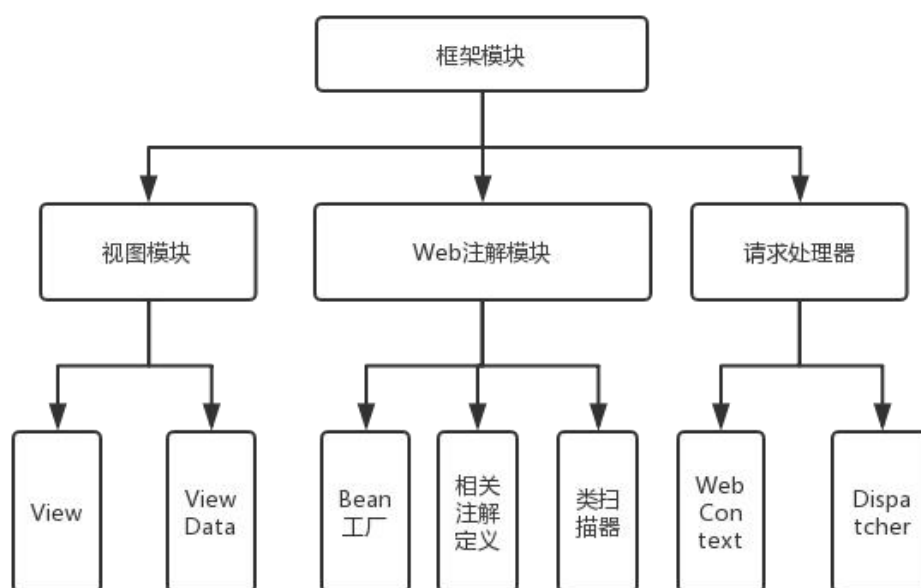


图 2.1 功能结构图(框架模块)

如图 2.2 所示为应用模块（App）功能结构图。

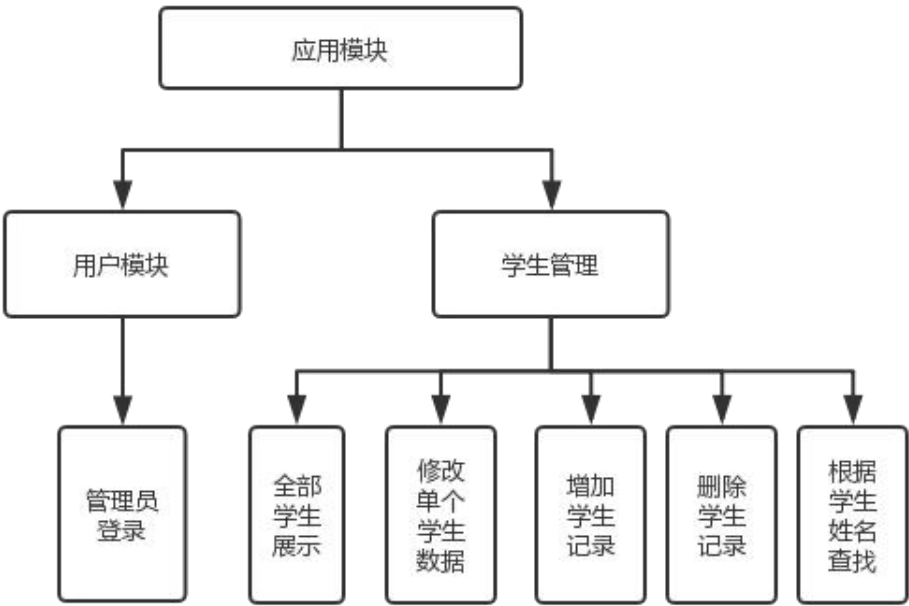


图 2.2 功能结构图(应用模块)

2.4 功能摘要

如表 2.3 所示为框架模块（framework）功能摘要说明。

表 2.3 功能摘要说明（框架模块）

功能模块	主要功能点	功能描述	优先级
视图模块	展示参数传递	当用户响应 Http 请求后，可能需要携带数据到界面展示，这时需要 View 和 ViewData 对象协助处理	中
Web 注解模块	类标记和依赖注入	将对 url 的响应都捕获到独立业务的 Controller 中实现，将 Url 映射使用 RequestMapping 进行标记	中

Web 注解模块	Bean 对象管理	在整个应用生命周期内部需要有某些持久化对象的存在, 尤其是需要进行依赖注入时, 开发者可以直接到 Bean 工厂中进行获取	高
请求处理器	当前 Request、Response 状态信息保存	WebContext 主要是用来存储当前线程中的 HttpServletRequest 和 HttpServletResponse, 当别的地方需要使用 HttpServletRequest 和 HttpServletResponse, 就可以通过 requestHolder 和 responseHolder 获取	高
请求处理器	Http 请求捕获	用来拦截所有请求, 获取客户端请求路径, 利用 java 反射技术, 实例化所有带有 Controller 注解的类, 去执行请求路径对应的 Controller 的方法	高

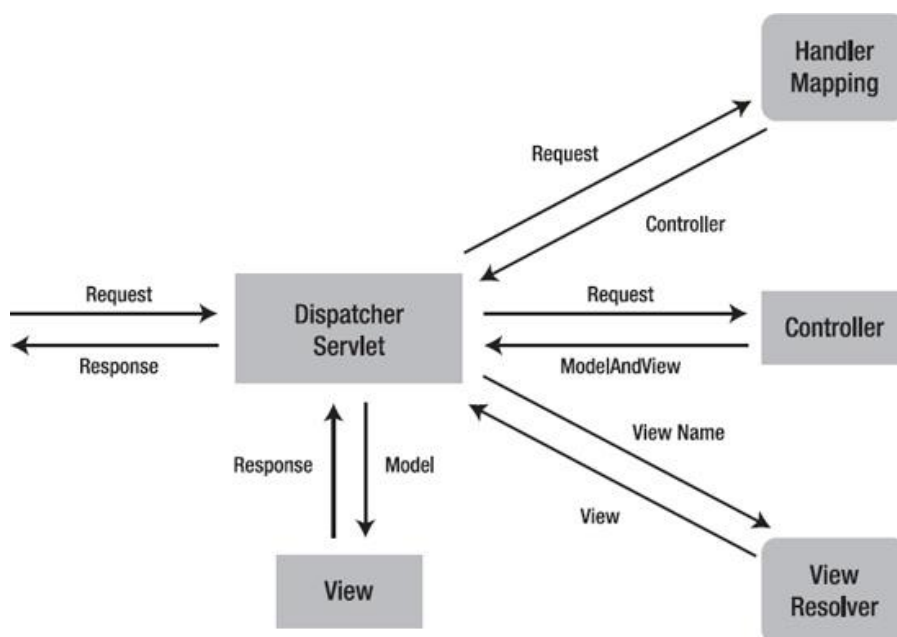


图 3.2 SpringMVC 请求响应

3.1 应用模块 App 分层结构

在学生信息管理模块中，主要分为 controller、db、dao、service 和 model 等 package，分别负责请求 url 的响应，数据库连接事务，学生数据访问接口定义与实现和数据模型定义等功能。

主体 Package 关系如图 3.3 所示：

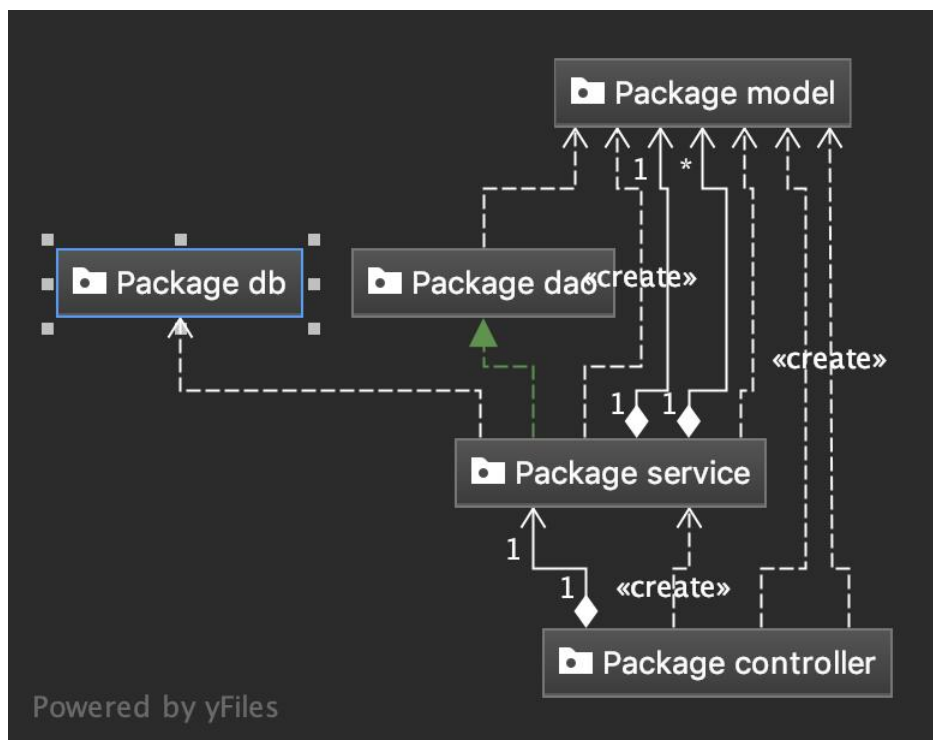


图 3.3 package 关系图(应用模块)

数据库访问事务 db 层设计

该层中需要实现对数据库连接的配置、使用单例模式获取数据库对象数据库连接，并且将项目中所用到的 SQL 语句进行整合管理，同时实现用户登录功能，如图 3.4 为数据库辅助类 JDBCUtils 的 UML 图。

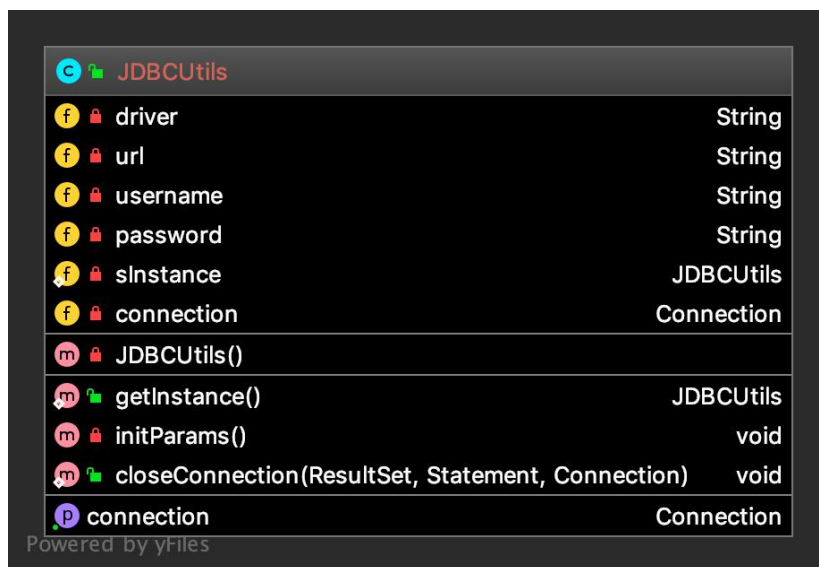


图 3.4 数据库辅助类 UML

数据访问接口 DAO 层设计

该层中需要实现基本数据访问接口的定义，增删改查，以及特殊性的查询接口定义，根据基本访问接口，根据学生信息管理的业务进行数据访问接口的进一步细化。图 3.5，3.6 分别为基本访问接口和学生访问接口的 UML。

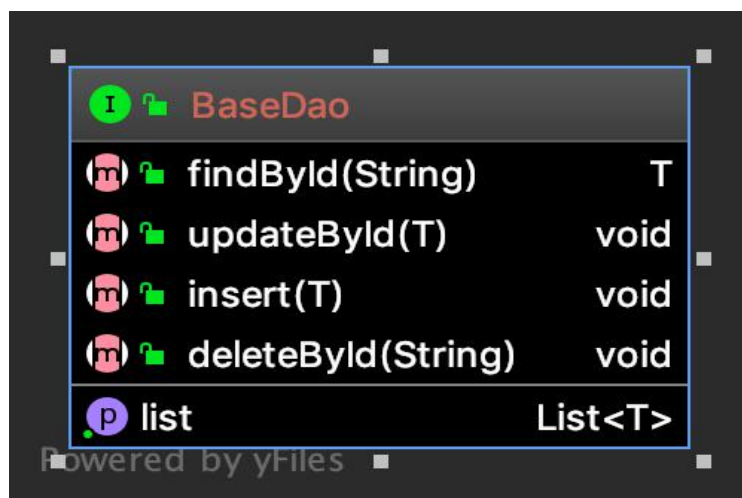


图 3.5 基本访问接口 UML

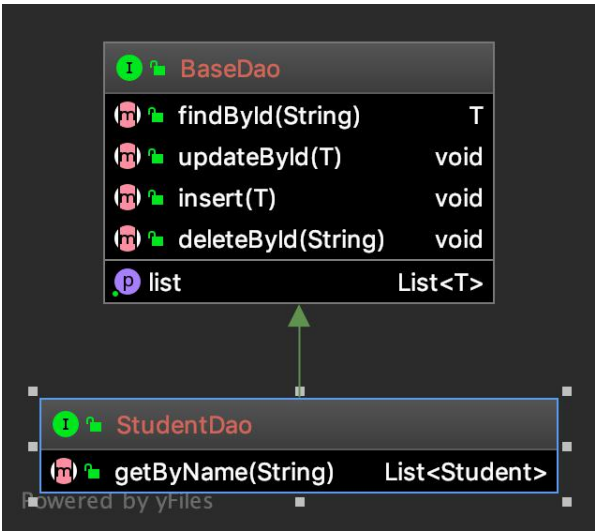


图 3.6 学生访问接口 UML

学生数据接口实现层 Service 设计

该层主要是对学生数据库访问接口的实现，需要获取数据库连接，并且将查询结果返回，图 3.7 为学生 service 的 UML。

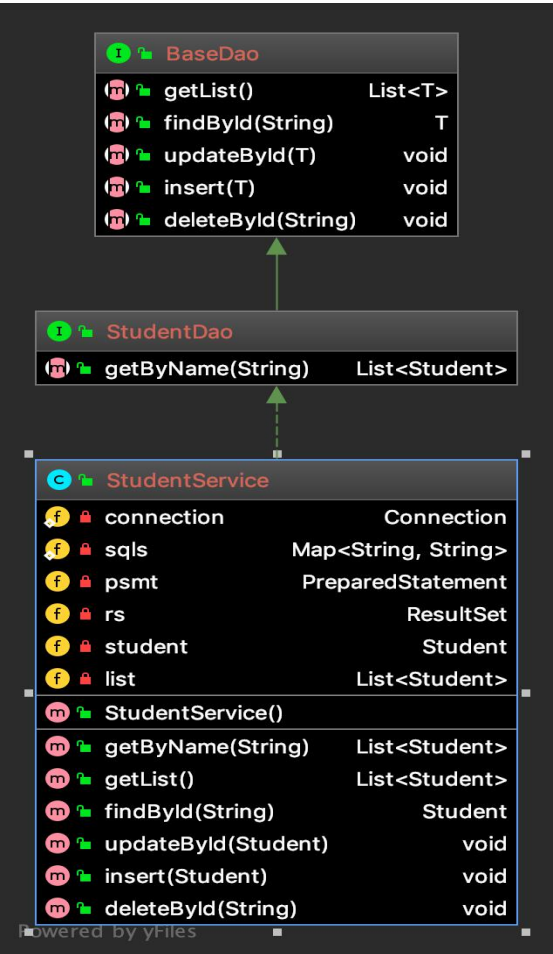


图 3.6 学生 service UML

4. 设计实现

4.1 Mini-SpringMVC 框架模块实现

在 Spring MVC 中，将一个普通的 java 类标注上 Controller 注解之后，再将类中的方法使用 RequestMapping 注解标注，那么这个普通的 java 类就够处理 Web 请求，然后我们只需要在 web.xml 文件中配置一个 Dispatcher Servlet (SpringMvc 控制器) 就可以了。

1) 编写注解

Controller 注解，该注解只有一个 value 属性，默认值为空字符串，代码如下：

```
import java.lang.annotation.*;

/**
 * Created by barackbao on 2019-06-07
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Controller {
    public String value() default "";
}
```

RequestMapping 注解，用户定义请求路径，该注解只有一个 value 属性，默认值为空字符串，代码如下：

```
import java.lang.annotation.*;

/**
 * Created by barackbao on 2019-06-07
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface RequestMapping {
```

```
        // 保存 url 信息  
        String value();  
    }  
}
```

2) 核心注解处理器

这里使用一个 `HttpServlet` 作为注解处理器，命名为 `DispatcherServlet`，实现思路为：

1. 在该 `Servlet` 进行初始化时 (`init`) 扫描指定的包下面使用了 `Controller` 注解的类。

2. 遍历类中的方法，找到类中使用了 `RequestMapping` 注解标注的那些方法，获取 `RequestMapping` 注解的 `value` 属性值，`value` 属性值指明了该方法的访问路径，以 `RequestMapping` 注解的 `value` 属性值作为 `key`，`Class` 类作为 `value` 将存储到一个静态 `Map` 集合中。

3. 当用户请求时(无论是 `get` 还是 `post` 请求)，会调用封装好的 `execute` 方法，`execute` 会先获取请求的 `url`，然后解析该 `URL`，根据解析好的 `URL` 从 `Map` 集合中取出要调用的目标类，再遍历目标类中定义的所有方法，找到类中使用了 `RequestMapping` 注解的那些方法，判断方法上面的 `RequestMapping` 注解的 `value` 属性值是否和解析出来的 `URL` 路径一致，如果一致，说明了这个就是要调用的目标方法，此时就可以利用 `java` 反射机制先实例化目标类对象，然后再通过实例化对象调用要执行的方法处理用户请求。

4. 另外，方法处理完成之后需要给客户端发送响应信息，比如告诉客户端要跳转到哪一个页面，采用的是服务器端跳转还是客户端方式跳转，或者发送一些数据到客户端显示，那么该如何发送响应信息给客户端呢，可以设计一个 `View`(视图)类，对这些操作属性进行封装，其中包括跳转的路径、展现到页面的数据、跳转方式。

下面为 `DispatcherServlet` 代码：

```
/**  
 * Created by barackbao on 2019-06-07  
 */  
  
public class DispatcherServlet extends HttpServlet {
```

```
/**
 * 从 HttpServletRequest 中解析出 请求路径,即 RequestMapping() 的 value 值.
 *
 * @param request
 * @return
 */
private String pareRequestURI(HttpServletRequest request) {
    String path = request.getContextPath() + "/";
    String requestUri = request.getRequestURI();
    String midUrl = requestUri.replace(path, "");
    String lastUrl = midUrl.substring(0, midUrl.lastIndexOf("."));
    return lastUrl;
}

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    System.out.println("DispatcherServlet-->doGet....");
    this.excute(request, response);
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    System.out.println("DispatcherServlet-->doPost....");
    this.excute(request, response);
}

private void excute(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

    //1.将当前 HttpServletRequest 放到 ThreadLocal 中,方便在 Controller 中使用
    WebContext.requestHolder.set(request);

    //将 HttpServletResponse 放到 ThreadLocal 中,方便在 Controller 中使用
    WebContext.responseHolder.set(response);
}
```

```
//2.解析请求的 url
String requestUrl = pareRequestURI(request);
//3.根据 请求的 url 获取要使用的类
Class<?> clazz = RequestMappingMap.getClassName(requestUrl);
//4.创建类的实例
Object classInstance = BeanUtils.instanceClass(clazz);
//5.获取类中定义的方法
Method[] methods = BeanUtils.findDeclaredMethods(clazz);
//遍历所有方法,找出 url 与 RequestMapping 注解的 value 值相匹配的方法
Method method = null;
for (Method m : methods) {
    if (m.isAnnotationPresent(RequestMapping.class)) {
        String value = m.getAnnotation(RequestMapping.class).value();
        if (value != null && !"".equals(value.trim()) &&
requestUrl.equals(value.trim())) {
            //找到要执行的目标方法
            method = m;
            break;
        }
    }
}
//6.执行 url 对应的方法,处理用户请
if (method != null) {
    Object retObject = null;
    try {
        //利用反射执行这个方法
        retObject = method.invoke(classInstance);
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
```



```
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        //如果有返回值,就代表用户需要返回视图
        if (retObject != null) {
            View view = (View) retObject;
            //判断要使用的跳转方式
            if
(view.getDispatchType().equals(DispatchActionConstants.FORWARD)) {
                //使用服务器端跳转方式
                System.out.println(view.getUrl());
                System.out.println(request.getAttribute("list"));
                request.getRequestDispatcher(view.getUrl()).forward(request,
response);
            } else if
(view.getDispatchType().equals(DispatchActionConstants.REDIRECT)) {
                //使用客户端跳转方式
                response.sendRedirect(request.getContextPath() +
view.getUrl());
            } else {
                request.getRequestDispatcher(view.getUrl()).forward(request,
response);
            }
        }
    }
}

@Override
public void init(ServletConfig config) throws ServletException {
    /**
```

* 重写了 Servlet 的 init 方法后一定要记得调用父类的 init 方法,
 * 否则在 service/doGet/doPost 方法中使用 getServletContext()方法获取
 ServletContext 对象时

* 就会出现 java.lang.NullPointerException 异常

*/

```
super.init(config);
```

```
System.out.println("---初始化开始---");
```

```
//获取 web.xml 中配置的要扫描的包
```

```
String basePackage = config.getInitParameter("basePackage");
```

```
//如果配置了多个包， 例如:
```

```
<param-value>me.gacl.web.com.barackbao.app.controller,me.gacl.web.UI</param-value>
```

```
if (basePackage.indexOf(",") > 0) {
```

```
    //按逗号进行分隔
```

```
    String[] packageNameArr = basePackage.split(",");
```

```
    for (String packageName : packageNameArr) {
```

```
        initRequestMappingMap(packageName);
```

```
    }
```

```
} else {
```

```
    initRequestMappingMap(basePackage);
```

```
}
```

```
System.out.println("----初始化结束---");
```

```
}
```

```
/**
```

```
 * @Method: initRequestMappingMap
```

```
 * @Description:添加使用了 Controller 注解的 Class 到 RequestMappingMap
```

中

```
*/
```

```
private void initRequestMappingMap(String packageName) {
```

```
//得到扫描包下的 class
Set<Class<?>> setClasses = ClassScanner.getClasses(packageName);
for (Class<?> clazz : setClasses) {

    if (clazz.isAnnotationPresent(Controller.class)) {
        Method[] methods = BeanUtils.findDeclaredMethods(clazz);
        for (Method m : methods) { //循环方法，找匹配的方法进行执行
            if (m.isAnnotationPresent(RequestMapping.class)) {
                String anoPath =
m.getAnnotation(RequestMapping.class).value();
                if (anoPath != null && !"".equals(anoPath.trim())) {
                    if
(RequestMappingMap.getRequesetMap().containsKey(anoPath)) {
                        throw new
RuntimeException("RequestMapping 映射的地址不允许重复! ");
                    }
                    //把所有的映射地址存储起来 映射路径--类
                    RequestMappingMap.put(anoPath, clazz);
                }
            }
        }
    }
}
}
```

5. 在 Web.xml 文件中注册 DispatcherHandler

```
<servlet>

    <servlet-name>DispatcherServlet</servlet-name>

    <servlet-class>com.barackbao.framework.web.servlet.DispatcherServlet</servlet
-class>
```

```
<init-param>
    <description>配置要扫描包及其子包, 如果有多个包,以逗号分隔</description>
    <param-name>basePackage</param-name>
    <param-value>com.barackbao.app.controller</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <!-- 拦截所有以.do 后缀结尾的请求 -->
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

3) 类扫描器

当应用开始运行后, 首先要对 JVM 中已经加载的类进行扫描并保存成文件的格式, 便于下一步对带有 Web 注解的类进行过滤, 以及反射调用。

类扫描器 ClassScanner 代码略。

4) 视图及视图数据定义

视图类对一些客户端响应操作进行封装, 其中包括跳转的路径、展现到页面的数据、跳转方式。ViewData 是 request 范围的数据存储类, 当需要发送数据到客户端显示时, 就可以将要显示的数据存储到 ViewData 类中。使用 ViewData.put(String name, Object value) 方法往 request 对象中存数据。

ViewData 核心代码如下:

```
/**
 * Created by barackbao on 2019-06-11
 * <p>
 * 存储 View 中的数据
 */
public class ViewData {
    private HttpServletRequest request;
```

```
public ViewData() {
    initRequest();
}

private void initRequest() {
    // 从 requestHolder 中获取当前 request 对象
    this.request = WebContext.requestHolder.get();
}

/**
 * 给 request 对象设置携带属性
 * @param name
 * @param value
 */

public void put(String name, Object value) {
    this.request.setAttribute(name, value);
}
}
```

5) WebContext 状态保存器

WebContext 主要是用来存储当前线程中的 `HttpServletRequest` 和 `HttpServletResponse`，当别的地方需要使用 `HttpServletRequest` 和 `HttpServletResponse`，就可以通过 `requestHolder` 和 `responseHolder` 获取，通过 `WebContext.java` 这个类，我们可以在作为 Controller 的普通 java 类中获取当前请求的 `request`、`response` 或者 `session` 相关请求类的实例变量，并且线程间互不干扰的，因为用到了 `ThreadLocal` 这个类。

WebContext 核心代码实现：

```
public class WebContext {

    // 使用 ThreadLocal 在多线程中隔离变量

    public static ThreadLocal<HttpServletRequest> requestHolder = new
ThreadLocal<>();

    public static ThreadLocal<HttpServletResponse> responseHolder = new
ThreadLocal<>();
}
```

```
public HttpServletRequest getRequest() {
    return requestHolder.get();
}

public HttpSession getSession() {
    return requestHolder.get().getSession();
}

public ServletContext getServletContext() {
    return requestHolder.get().getSession().getServletContext();
}

public HttpServletResponse getResponse() {
    return responseHolder.get();
}
}
```

4.2 App 应用模块实现

1) 数据访问层实现

使用单例模式获取数据库对象和数据库连接，并且提供公有静态数据库关闭方法，保证资源的及时释放。

核心代码如下：

```
public class JDBCUtils {
    private String driver;
    private String url;
    private String username;
    private String password;
    private static volatile JDBCUtils sInstance;
    private Connection connection;
    // 单例模式
    private JDBCUtils() {
    }
    public static JDBCUtils getInstance() {
        if (null == sInstance) {
```

```
synchronized (JDBCUtils.class) {
    if (null == sInstance) {
        sInstance = new JDBCUtils();
    }
}

sInstance.initParams();
return sInstance;
}

private void initParams() {
    driver = "com.mysql.jdbc.Driver";
    url =
"jdbc:mysql://localhost:3306/stmamvc?characterEncoding=UTF-8";
    username = "root";
    password = "admin";
    // 加载 JDBC 驱动
    try {
        Class.forName(driver);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    connection = getConnection();
}

public Connection getConnection() {
    try {
        connection = DriverManager.getConnection(url, username,
password);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
        return connection;
    }

    public static void closeConnection(ResultSet res, Statement stat, Connection
conn) {
        try {
            if (res != null) res.close();
            if (stat != null) stat.close();
            if (conn != null) conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2) 项目基本数据访问接口和学生信息访问接口

核心代码如下:

```
public interface BaseDao<T> {
    // 查询整个表
    public List<T> getList();
    // 根据 id 查找单个元素
    public T findById(String id);
    // 根据 id 修改某个元素
    public void updateById(T t);
    // 插入数据
    public void insert(T t);
    // 根据 id 删除某个元素
    public void deleteById(String id);
}

public interface StudentDao extends BaseDao<Student> {
    // 增加通过学生姓名查找元素
    public List<Student> getByName(String name);
}
```



```
}
```

3) 数据访问接口的具体实现

代码略, SQL 语句集合如下:

```
public class Sqli {  
    public static final String GET_LIST = "SELECT * FROM Student";  
    public static final String INSERT_ONE = "INSERT INTO Student  
(sno,name,sex,age) VALUES (?,?,,?)";  
    public static final String UPDATE_ONE = "UPDATE Student SET  
sno=?,name=?,sex=?,age=? WHERE id=?";  
    public static final String GETONEBY_ID = " SELECT id,sno,name,sex,age  
FROM Student WHERE id=?";  
    public static final String DELETEONEBY_ID = "DELETE from Student  
WHERE id=?";  
}
```

4) 前端界面实现

前端界面共包括登录界面 Login.jsp、学生信息展示界面 studentList.jsp、学生信息修改界面 updateStudent.jsp 和增加学生界面 insertStudent.jsp。

为了方便控件查找,项目中使用 JSTL 标准标签库,并且使用了 Bootstrap 前端框架进行页面渲染。

学生信息展示界面 studentList.jsp 代码如下:

```
<%@ page contentType="text/html;charset=UTF-8" %>  
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<%@ page isELIgnored="false" %>  
<html>  
<!-- 引入 bootstrap -->  
<link href="https://cdn.bootcss.com/twitter-bootstrap/4.3.1/css/bootstrap.min.css"  
rel="stylesheet">  
<script src="https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"></script>  
<script  
src="https://cdn.bootcss.com/twitter-bootstrap/4.3.1/js/bootstrap.min.js"></script>
```

```
<head>
  <title>学生列表</title>
</head>
<body>
<div class="container" id="content">
  <div class="row">
    <div class="col-md-10">
      <div class="panel panel-default">
        <div class="panel-heading">
          <div class="row">
            <h1 class="col-md-5">学生信息显示</h1>
            <form class="bs-example bs-example-form col-md-5"
role="form" style="margin: 20px 0 10px 0;"
              method="get" action="/getStudentByName">
              <div class="input-group">
                <input type="text" class="form-control"
placeholder="请输入学生姓名" name="inputName"
                  id="inputName">
                <span class="input-group-addon
btn"><button type="submit" id="select"
disabled="disabled">查询</button></span>
                <a href="/insertStudentget.do"><span
class="btn btn-danger">新增记录</span></a>
              </div>
            </form>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

```
<table class="table table-bordered">
  <thead>
    <tr>
      <th nowrap="nowrap">序号</th>
      <th nowrap="nowrap">学号</th>
      <th nowrap="nowrap">姓名</th>
      <th nowrap="nowrap">性别</th>
      <th nowrap="nowrap">年龄</th>
      <th nowrap="nowrap">操作</th>
    </tr>
  </thead>
  <tbody>
    <c:forEach var="student" items="${list}">

      <tr>
        <td>${student.id}</td>
        <td>${student.sno}</td>
        <td>${student.name}</td>
        <td>${student.sex}</td>
        <td>${student.age}</td>

        <td><a
href="/updateStudentget.do?id=${student.id}">
          <button class="btn btn-info" onclick="return
confirmUpdate()">
            <span class="glyphicon glyphicon-pencil"
aria-hidden="true">

          </span>
          修改
        </button>
```

```
</a>
    <a
href="/deleteStudent.do?id=${student.id}">
        <button class="btn btn-danger"
onclick="return confirmDelete()">
            <span class="glyphicon glyphicon-trash"
aria-hidden="true">
                </span>
                删除
            </button>
        </a></td>
    </tr>
</c:forEach>
</tbody>
</table>
</div>
</div>
</div>
</body>

<script type="text/javascript">
    function confirmDelete() {
        return confirm("确定删除该记录吗");
    }

    function confirmUpdate() {
        return confirm("确定修改该记录吗");
    }
}
```

```
$(function () {  
    $("#inputName").change(function () {  
        var a = $("#inputName").val().trim();  
        if (null == a || a.length == 0) {  
            $("#select").attr('disabled', 'disabled');  
        } else {  
            $("#select").removeAttr("disabled");  
        }  
    });  
});  
</script>  
</html>
```

5) 请求响应 Controller

负责对 DispatcherServlet 拦截到的请求进行处理，调用学生信息访问接口去数据库中进行查询，并且将数据设置到 View 对象中，进行 jsp 界面的跳转。

核心代码如下：

@Controller

```
public class StudentController {  
    private StudentService service = new StudentService();  
    @RequestMapping("getList")  
    public View getStudentList() {  
        HttpServletRequest request = WebContext.requestHolder.get();  
        List<Student> list = service.getList();  
        ViewData viewData = new ViewData();  
        viewData.put("list", list);  
        return new View("/studentList.jsp");  
    }  
    // get 方法  
    @RequestMapping("insertStudentget")
```

```
public View insertStudentGet() {
    return new View("/insertStudent.jsp");
}

@RequestMapping("insertStudentpost")
public View insertStudentPost() throws UnsupportedEncodingException {
    HttpServletRequest request = WebContext.requestHolder.get();
    request.setCharacterEncoding("UTF-8");
    Student student = new Student();
    student.setName(request.getParameter("name"));
    student.setSno(request.getParameter("sno"));
    student.setSex(request.getParameter("sex"));
    student.setAge(Integer.parseInt(request.getParameter("age")));

    System.out.println(student);
    service.insert(student);
    List<Student> list = service.getList();
    //插入成功后返回到首页学生列表
    ViewData viewData = new ViewData();
    viewData.put("list", list);
    return new View("/studentList.jsp");
}

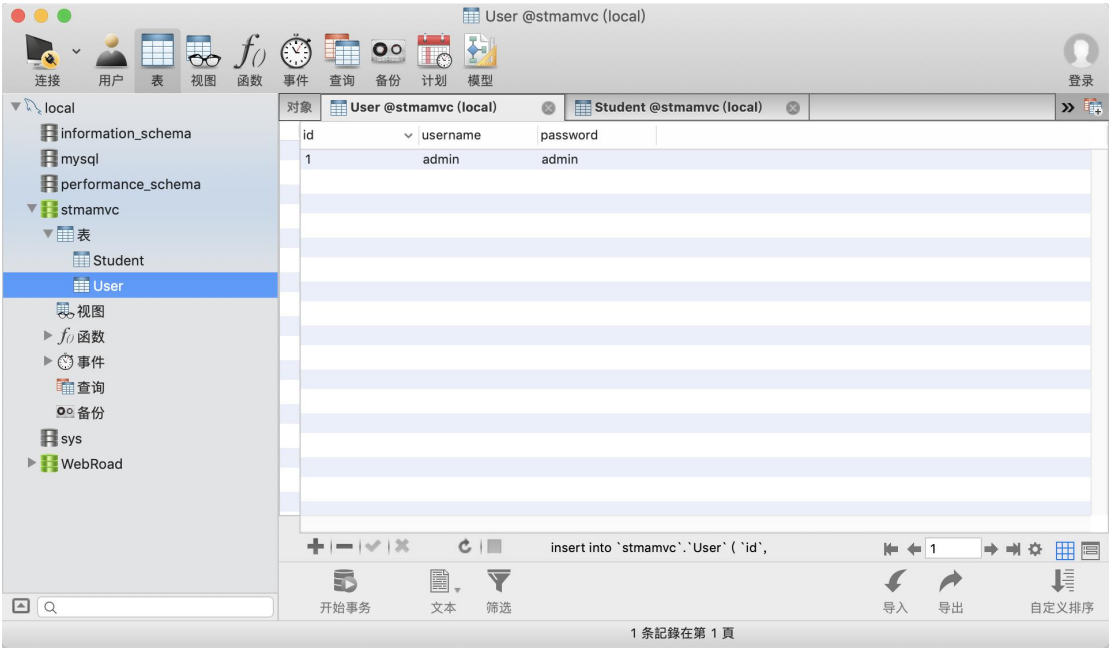
@RequestMapping("updateStudentget")
public View updateStudentGet() {
    HttpServletRequest request = WebContext.requestHolder.get();
    String id = request.getParameter("id");
    Student student = service.findById(id);
    ViewData viewData = new ViewData();
    viewData.put("student", student);
    return new View("updateStudent.jsp");
}
```

```
@RequestMapping("updateStudentpost")
public View updateStudentPost() throws UnsupportedEncodingException {
    HttpServletRequest request = WebContext.requestHolder.get();
    request.setCharacterEncoding("UTF-8");
    Student student = new Student();
    student.setId(Integer.parseInt(request.getParameter("id")));
    student.setName(request.getParameter("name"));
    student.setSno(request.getParameter("sno"));
    student.setSex(request.getParameter("sex"));
    student.setAge(Integer.parseInt(request.getParameter("age")));
    service.updateById(student);
    List<Student> list = service.getList();
    ViewData viewData = new ViewData();
    viewData.put("list", list);
    return new View("/studentList.jsp");
}

@RequestMapping("deleteStudent")
public View deleteStudent() {
    HttpServletRequest request = WebContext.requestHolder.get();
    String id = request.getParameter("id");
    service.deleteById(id);
    List<Student> list = service.getList();
    ViewData viewData = new ViewData();
    viewData.put("list", list);
    return new View("studentList.jsp");
}
}
```

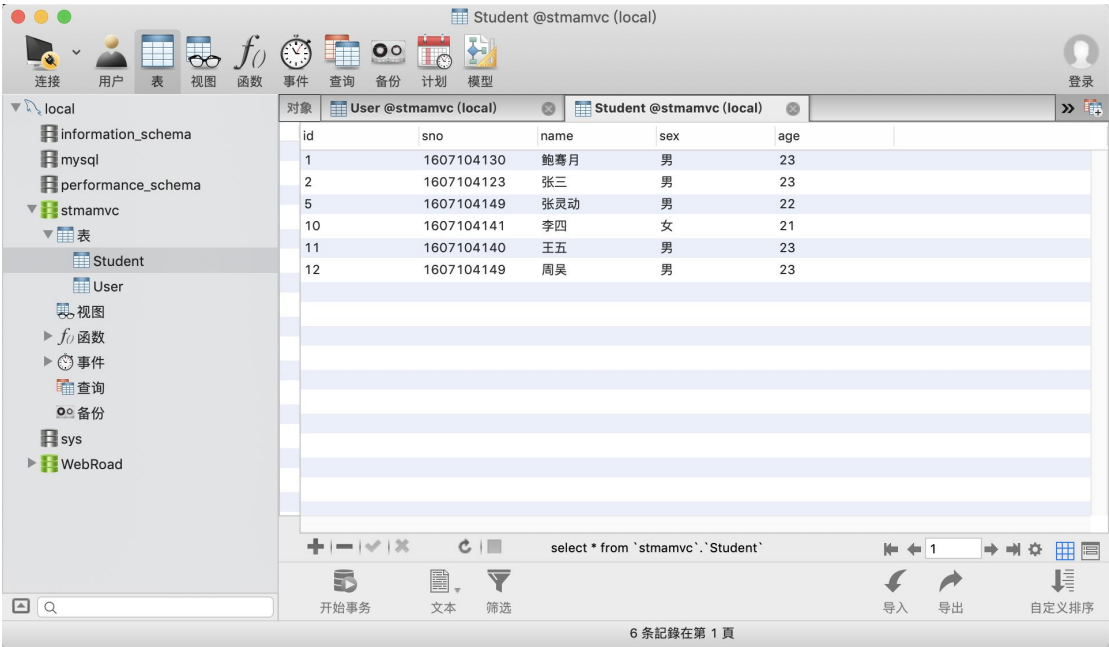
5. 实验结果及分析

1) 数据库表数据如图 5.1， 5.2 所示。



id	username	password
1	admin	admin

图 5.1 User 表



id	sno	name	sex	age
1	1607104130	鲍蓉月	男	23
2	1607104123	张三	男	23
5	1607104149	张灵动	男	22
10	1607104141	李四	女	21
11	1607104140	王五	男	23
12	1607104149	周吴	男	23

图 5.2 Student 表

2) 登录界面

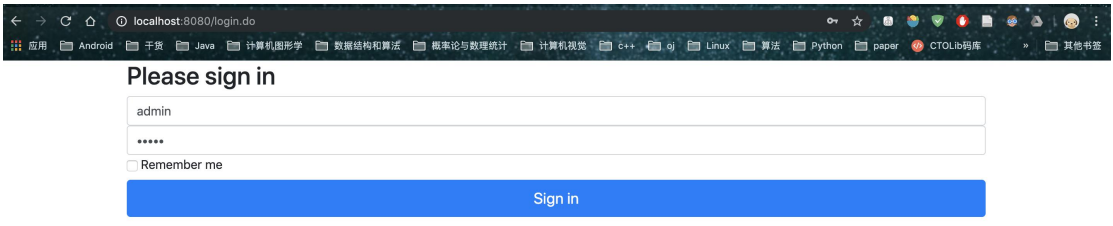


图 5.3 登录界面

3) 学生信息展示



图 5.4 学生信息展示界面

4) 插入学生信息

新增学生信息

学号:

姓名:

性别:

年龄:

图 5.5 插入学生信息界面

5) 修改单个学生信息

更新学生信息

学号:

姓名:

性别:

年龄:

图 5.6 修改学生信息界面

6) 删除单个学生信息



图 5.7 删除学生信息界面

经测试，上述功能均能正常访问数据库，对学生信息表 Student 进行增删改查操作。

6. 设计总结

本项目首先通过自主实现了一个轻量级的 Spring MVC 框架，解决了普通 servlet 和 jsp 框架 URL 映射复杂高耦合的问题，使用 Controller 来对 Http 请求进行响应和处理，并使用控制反转和依赖注入机制完成对 Bean 对象的完整生命周期管理，通过自己模拟实现 Spring MVC 框架，不仅使我对 Servlet 的运行过程更加了解，同时能够清楚的了解到这种传统模式的缺点，也让我更加熟悉了 SpringMVC 框架，同时也是对 Java 高级特性，注解和反射基础的一次巩固，这对之后我学习其他先进的 Spring 框架有巨大的好处。

在项目应用模块 App 中，首先使用了 Bootstrap 前端框架对用户操作 JSP 界面进行了美化，并且为相应的 Button 配置了响应 URL，然后使用了 DB, DAO, service 三层数据操作模式，将具体的业务分别在不同的层级进行实现，这种模式可以较为优雅的实现用户的数据操作功能，同时能够降低模块之间的耦合性，各模块各司其职，在项目 DEBUG 阶段，更容易展示出问题所在，在具体实现了对学生数据表 Student 的各项访问接口后，为每个 URL 在 StudentController 中配置了响应处理方法，对有界面跳转的 URL，首先通过 WebContext 从隔离线程 ThreadLocal 中获取到保存的 request 对象，然后将跳转参数封装在 ViewData 对象中，然后使用 request.getRequestDispatcher().forward()方法进行页面跳转。通过框架模块和应用模块的相互配合，本系统能够完整的实现学生信息管理系统的需求。