#### 数据库连接的建立

该系统使用JDBC的方式统一进行数据库连接。将mysql-connector-java依赖添加进项目后，在JdbcUtil类中手动写入固定的连接数据库的参数，包括驱动路径、URL、数据库用户名、密码，项目运行后读取参数，反射加载JDBC数据库连接驱动，执行相应步骤，连接数据库。具体代码如下：

public final class JdbcUtil {

private static String connect;

private static String driverClassName;

private static String URL;

private static String username;

private static String password;

private static boolean autoCommit;

/\*\* 声明一个 Connection类型的静态属性，用来缓存一个已经存在的连接对象 \*/

private static Connection conn;

static {

config();

}

/\*\*

\* 开头配置自己的数据库信息

\*/

private static void config() {

/\*

\* 获取驱动

\*/

driverClassName = "com.mysql.jdbc.Driver";

/\*

\* 获取URL

\*/

URL = "jdbc:mysql://localhost:3306/javaweb\_books\_management\_system?useUnicode=true&characterEncoding=utf8&useSSL=false";

/\*

\* 获取用户名

\*/

username = "root";

/\*

\* 获取密码

\*/

password = "123456";

/\*

\* 设置是否自动提交，一般为false不用改

\*/

autoCommit = false;

}

/\*\*

\* 载入数据库驱动类

\*/

private static boolean load() {

try {

Class.forName(driverClassName);

return true;

} catch (ClassNotFoundException e) {

System.out.println("驱动类 " + driverClassName + " 加载失败");

}

return false;

}

/\*\*

\* 专门检查缓存的连接是否不可以被使用 ，不可以被使用的话，就返回 true

\*/

private static boolean invalid() {

if (conn != null) {

try {

if (conn.isClosed() || !conn.isValid(3)) {

return true;

/\*

\* isValid方法是判断Connection是否有效,如果连接尚未关闭并且仍然有效，则返回 true

\*/

}

} catch (SQLException e) {

e.printStackTrace();

}

/\*

\* conn 既不是 null 且也没有关闭 ，且 isValid 返回 true，说明是可以使用的 ( 返回 false )

\*/

return false;

} else {

return true;

}

}

/\*\*

\* 建立数据库连接

\*/

public static Connection connect() {

if (invalid()) { /\* invalid为true时，说明连接是失败的 \*/

/\* 加载驱动 \*/

load();

try {

/\* 建立连接 \*/

conn = DriverManager.getConnection(URL, username, password);

} catch (SQLException e) {

System.out.println("建立 " + connect + " 数据库连接失败 , " + e.getMessage());

}

}

return conn;

}

/\*\*

\* 设置是否自动提交事务

\*\*/

public static void transaction() {

try {

conn.setAutoCommit(autoCommit);

} catch (SQLException e) {

System.out.println("设置事务的提交方式为 : " + (autoCommit ? "自动提交" : "手动提交") + " 时失败: " + e.getMessage());

}

}

/\*\*

\* 创建 Statement 对象

\*/

public static Statement statement() {

Statement st = null;

connect();

/\* 如果连接是无效的就重新连接 \*/

transaction();

/\* 设置事务的提交方式 \*/

try {

st = conn.createStatement();

} catch (SQLException e) {

System.out.println("创建 Statement 对象失败: " + e.getMessage());

}

return st;

}

/\*\*

\* 根据给定的带参数占位符的SQL语句，创建 PreparedStatement 对象

\*

\* @param SQL

\* 带参数占位符的SQL语句

\* @return 返回相应的 PreparedStatement 对象

\*/

private static PreparedStatement prepare(String SQL, boolean autoGeneratedKeys) {

PreparedStatement ps = null;

connect();

/\* 如果连接是无效的就重新连接 \*/

transaction();

/\* 设置事务的提交方式 \*/

try {

if (autoGeneratedKeys) {

ps = conn.prepareStatement(SQL, Statement.RETURN\_GENERATED\_KEYS);

} else {

ps = conn.prepareStatement(SQL);

}

} catch (SQLException e) {

System.out.println("创建 PreparedStatement 对象失败: " + e.getMessage());

}

return ps;

}

public static ResultSet query(String SQL, Object... params) {

if (SQL == null || SQL.trim().isEmpty() || !SQL.trim().toLowerCase().startsWith("select")) {

throw new RuntimeException("你的SQL语句为空或不是查询语句");

}

ResultSet rs = null;

if (params.length > 0) {

/\* 说明 有参数 传入，就需要处理参数 \*/

PreparedStatement ps = prepare(SQL, false);

try {

for (int i = 0; i < params.length; i++) {

ps.setObject(i + 1, params[i]);

}

rs = ps.executeQuery();

} catch (SQLException e) {

System.out.println("执行SQL失败: " + e.getMessage());

}

} else {

/\* 说明没有传入任何参数 \*/

Statement st = statement();

try {

rs = st.executeQuery(SQL); // 直接执行不带参数的 SQL 语句

} catch (SQLException e) {

System.out.println("执行SQL失败: " + e.getMessage());

}

}

return rs;

}

private static Object typeof(Object o) {

Object r = o;

if (o instanceof java.sql.Timestamp) {

return r;

}

// 将 java.util.Date 转成 java.sql.Date

if (o instanceof java.util.Date) {

java.util.Date d = (java.util.Date) o;

r = new java.sql.Date(d.getTime());

return r;

}

// 将 Character 或 char 变成 String

if (o instanceof Character || o.getClass() == char.class) {

r = String.valueOf(o);

return r;

}

return r;

}

public static boolean execute(String SQL, Object... params) {

if (SQL == null || SQL.trim().isEmpty() || SQL.trim().toLowerCase().startsWith("select")) {

throw new RuntimeException("你的SQL语句为空或有错");

}

boolean r = false;

/\* 表示 执行 DDL 或 DML 操作是否成功的一个标识变量 \*/

/\* 获得 被执行的 SQL 语句的 前缀 \*/

SQL = SQL.trim();

SQL = SQL.toLowerCase();

String prefix = SQL.substring(0, SQL.indexOf(" "));

String operation = ""; // 用来保存操作类型的 变量

// 根据前缀 确定操作

switch (prefix) {

case "create":

operation = "create table";

break;

case "alter":

operation = "update table";

break;

case "drop":

operation = "drop table";

break;

case "truncate":

operation = "truncate table";

break;

case "insert":

operation = "insert :";

break;

case "update":

operation = "update :";

break;

case "delete":

operation = "delete :";

break;

}

if (params.length > 0) { // 说明有参数

PreparedStatement ps = prepare(SQL, false);

Connection c = null;

try {

c = ps.getConnection();

} catch (SQLException e) {

e.printStackTrace();

}

try {

for (int i = 0; i < params.length; i++) {

Object p = params[i];

p = typeof(p);

ps.setObject(i + 1, p);

}

ps.executeUpdate();

commit(c);

r = true;

} catch (SQLException e) {

System.out.println(operation + " 失败: " + e.getMessage());

rollback(c);

}

} else { // 说明没有参数

Statement st = statement();

Connection c = null;

try {

c = st.getConnection();

} catch (SQLException e) {

e.printStackTrace();

}

// 执行 DDL 或 DML 语句，并返回执行结果

try {

st.executeUpdate(SQL);

commit(c); // 提交事务

r = true;

} catch (SQLException e) {

System.out.println(operation + " 失败: " + e.getMessage());

rollback(c); // 回滚事务

}

}

return r;

}

/\*

\*

\* @param SQL

\* 需要执行的 INSERT 语句

\* @param autoGeneratedKeys

\* 指示是否需要返回由数据库产生的键

\* @param params

\* 将要执行的SQL语句中包含的参数占位符的 参数值

\* @return 如果指定 autoGeneratedKeys 为 true 则返回由数据库产生的键； 如果指定 autoGeneratedKeys

\* 为 false 则返回受当前SQL影响的记录数目

\*/

public static int insert(String SQL, boolean autoGeneratedKeys, Object... params) {

int var = -1;

if (SQL == null || SQL.trim().isEmpty()) {

throw new RuntimeException("你没有指定SQL语句，请检查是否指定了需要执行的SQL语句");

}

// 如果不是 insert 开头开头的语句

if (!SQL.trim().toLowerCase().startsWith("insert")) {

System.out.println(SQL.toLowerCase());

throw new RuntimeException("你指定的SQL语句不是插入语句，请检查你的SQL语句");

}

// 获得 被执行的 SQL 语句的 前缀 ( 第一个单词 )

SQL = SQL.trim();

SQL = SQL.toLowerCase();

if (params.length > 0) { // 说明有参数

PreparedStatement ps = prepare(SQL, autoGeneratedKeys);

Connection c = null;

try {

c = ps.getConnection(); // 从 PreparedStatement 对象中获得 它对应的连接对象

} catch (SQLException e) {

e.printStackTrace();

}

try {

for (int i = 0; i < params.length; i++) {

Object p = params[i];

p = typeof(p);

ps.setObject(i + 1, p);

}

int count = ps.executeUpdate();

if (autoGeneratedKeys) { // 如果希望获得数据库产生的键

ResultSet rs = ps.getGeneratedKeys(); // 获得数据库产生的键集

if (rs.next()) { // 因为是保存的是单条记录，因此至多返回一个键

var = rs.getInt(1); // 获得值并赋值给 var 变量

}

} else {

var = count; // 如果不需要获得，则将受SQL影像的记录数赋值给 var 变量

}

commit(c);

} catch (SQLException e) {

System.out.println("数据保存失败: " + e.getMessage());

rollback(c);

}

} else { // 说明没有参数

Statement st = statement();

Connection c = null;

try {

c = st.getConnection(); // 从 Statement 对象中获得 它对应的连接对象

} catch (SQLException e) {

e.printStackTrace();

}

// 执行 DDL 或 DML 语句，并返回执行结果

try {

int count = st.executeUpdate(SQL);

if (autoGeneratedKeys) { // 如果企望获得数据库产生的键

ResultSet rs = st.getGeneratedKeys(); // 获得数据库产生的键集

if (rs.next()) { // 因为是保存的是单条记录，因此至多返回一个键

var = rs.getInt(1); // 获得值并赋值给 var 变量

}

} else {

var = count; // 如果不需要获得，则将受SQL影像的记录数赋值给 var 变量

}

commit(c); // 提交事务

} catch (SQLException e) {

System.out.println("数据保存失败: " + e.getMessage());

rollback(c); // 回滚事务

}

}

return var;

}

/\*\* 提交事务 \*/

private static void commit(Connection c) {

if (c != null && !autoCommit) {

try {

c.commit();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

/\*\* 回滚事务 \*/

private static void rollback(Connection c) {

if (c != null && !autoCommit) {

try {

c.rollback();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

/\*\*

\* 释放资源

\* \*\*/

public static void release(Object cloaseable) {

if (cloaseable != null) {

if (cloaseable instanceof ResultSet) {

ResultSet rs = (ResultSet) cloaseable;

try {

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (cloaseable instanceof Statement) {

Statement st = (Statement) cloaseable;

try {

st.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (cloaseable instanceof Connection) {

Connection c = (Connection) cloaseable;

try {

c.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

}

#### 登录注册功能的实现

本系统的功能页面由服务器内部跳转访问，登录页面验证用户身份通过后，跳转到相应页面。用户输入的登录文本框中的账号密码首先到达LoginServlet的doGet方法中，根据前端传来的用户身份信息判断用户是普通用户还是管理员，再通过Service、Dao层，调用Jdbc工具类方法，根据用户名查询数据库中的用户信息，成功则再判断用户密码，再次成功后在服务器内部工具类和session中记录用户ID和用户名信息，Filter过滤器判断服务器内部存在用户信息放行，内部跳转至功能页面JSP；失败后则返回失败提示信息给前端，前端使用JS提示框显示。点击个人信息页面退出登录按钮后，服务器内部清除用户信息，内部跳转至登录页面。

在注册页面文本框中输入必要信息，点击按钮提交注册请求后，服务器内部LoginServlet内doGet根据method参数判断用户要执行注册操作，将前端传递的信息作为参数调用Service层注册方法，判断是否合法后再通过Dao层将用户信息写入数据库。将执行后的状态信息作为参数返回到前端，前端判断注册成功后跳转至登录页面，失败则显示提示信息。具体代码如下：

public class UserService {

public UserDao userDao = new UserDao();

public User selectByUsername(String username) {

return userDao.selectByUsername(username);

}

public Result<String> verifyLogin(User user) {

String username = user.getUsername(),

password = user.getPassword();

User databaseUser = selectByUsername(username);

if (databaseUser == null) {

return new Result<>(401, "没有该用户");

}

if (password.equals(databaseUser.getPassword())) {

return new Result<>(200, "登录成功");

}

return new Result<>(401, "登录失败");

}

public Result<String> verifyLogin(String username, String password) {

User user = new User();

user.setUsername(username);

user.setPassword(password);

return verifyLogin(user);

}

public Result<String> register(String username, String password, String phone) {

if (selectByUsername(username) != null) {

return new Result<>(403, "注册失败");

}

User user = new User();

user.setUsername(username);

user.setPassword(password);

user.setPhone(phone);

userDao.insertOne(user);

return new Result<>(200, "注册成功");

}

public List<User> selectAll() {

return userDao.selectAll();

}

public void deleteOne(int id) {

userDao.deleteOne(id);

}

public List<User> selectByUsernameLike(String username) {

return userDao.selectByUsernameLike(username);

}

public User selectOne(int id) {

return userDao.selectOne(id);

}

public void updateOne(User user) {

if (selectByUsername(user.getUsername()) != null) {

throw new RuntimeException("重复用户名，修改失败");

}

userDao.updateOne(user);

}

}

#### 图书查询、借阅归还功能的实现

通过Dao、Service层查询到全部图书信息后，作为图书查询页面默认展示的数据。在前端接收到作者、书名条件查询参数后，作为参数通过Service层传递至Dao层，通过多分支if判断每个参数的空白与否，划分出会出现的四种情况，分别拼接使用不同的SQL查询语句，然后调用Jdbc工具类相关方法从数据库获取相应图书信息，并返回到前端页面进行展示。

数据库book图书信息表包含tinyint类型的is\_borrowed列，用来标记该图书是否被借阅，1代表已被借阅，0代表未被借阅。用户通过点击前端页面的借阅归还按钮向服务器发送需要更改图书的ID信息和需要执行的操作，服务器调用Service、Dao方法，更改数据库中图书is\_borrowed状态。且当用户点击借阅按钮后判断该图书状态是否已经为1，如是则返回失败结果。以实现图书借阅归还功能。具体代码如下：

case "condition":

String author = request.getParameter("author");

String name = request.getParameter("name");

Book book = new Book();

book.setAuthor(author);

book.setName(name);

request.setAttribute("books", bookService.selectCondition(book));

request.getRequestDispatcher("/router?page=book").forward(request, response);

break;

case "borrow":

int id = Integer.parseInt(request.getParameter("id"));

// 借阅

if (bookService.selectOne(id).getIsBorrowed()) {

throw new RuntimeException("借阅失败");

}

bookService.bookBorrow(id, true);

// 插入借阅记录

Borrowing borrowing = new Borrowing();

borrowing.setUserId(CommonUtil.getUserId());

borrowing.setBookId(id);

borrowing.setType("borrowing");

borrowing.setDatetime(new Timestamp(new Date().getTime()));

borrowingService.insertOne(borrowing);

// 跳转

request.getRequestDispatcher("/router?page=book").forward(request, response);

break;

case "return":

int returnId = Integer.parseInt(request.getParameter("id"));

// 借阅

if (!bookService.selectOne(returnId).getIsBorrowed()) {

throw new RuntimeException("归还失败");

}

bookService.bookBorrow(returnId, false);

// 插入借阅记录

borrowing = new Borrowing();

borrowing.setUserId(CommonUtil.getUserId());

borrowing.setBookId(returnId);

borrowing.setType("returning");

borrowing.setDatetime(new Timestamp(new Date().getTime()));

borrowingService.insertOne(borrowing);

// 跳转

request.getRequestDispatcher("/router?page=book").forward(request, response);

break;

case "add":

book = new Book();

book.setName(request.getParameter("name"));

book.setAuthor(request.getParameter("author"));

book.setPublisher(request.getParameter("publisher"));

book.setIsbn(request.getParameter("isbn"));

book.setInfo(request.getParameter("info"));

book.setPricing(Double.parseDouble(request.getParameter("pricing")));

book.setIsBorrowed(request.getParameter("is\_borrowed").equals("是"));

bookService.insertOne(book);

request.getRequestDispatcher("/router?page=admin\_book\_management").forward(request, response);

break;

#### 图书管理功能的实现

图书管理功能包括图书的查询、删除、修改。默认通过Dao、Service将查询到的全部图书信息展示在前端页面，条件查询根据传递参数在Dao层分情况拼接不同SQL语句查询特定参数信息图书。前端点击图书删除按钮后，获取到该行图书ID信息，并将影响参数传递到BookServlet，调用Service方法，通过Dao层，将该图书信息从数据库删除。前端点击修改按钮后，将图书ID传递至图书修改页面，修改页面根据图书ID首先向数据库发起获取该图书ID信息的请求，并将服务器传递数据展示在相应文本框中，以方便图书信息修改，用户在想要修改的文本框中输入自己想要的信息后提交表单，服务器根据前端传递图书信息参数调用Service、Dao层根据图书ID实现图书信息修改操作。具体代码如下：

case "condition":

String author = request.getParameter("author");

String name = request.getParameter("name");

Book book = new Book();

book.setAuthor(author);

book.setName(name);

request.setAttribute("books", bookService.selectCondition(book));

request.getRequestDispatcher("/router?page=book").forward(request, response);

break;

case "delete":

id = Integer.parseInt(request.getParameter("id"));

bookService.deleteOne(id);

request.getRequestDispatcher("/router?page=admin\_book\_management").forward(request, response);

break;

case "update":

book = new Book();

book.setId(Integer.parseInt(request.getParameter("id")));

book.setName(request.getParameter("name"));

book.setAuthor(request.getParameter("author"));

book.setPublisher(request.getParameter("publisher"));

book.setIsbn(request.getParameter("isbn"));

book.setInfo(request.getParameter("info"));

book.setPricing(Double.parseDouble(request.getParameter("pricing")));

book.setIsBorrowed(request.getParameter("is\_borrowed").equals("是"));

bookService.update(book);

request.getRequestDispatcher("/router?page=admin\_book\_management").forward(request, response);

break;

#### 图书添加功能的实现

用户在图书添加页面相应图书信息文本框内输入需要添加的图书信息，提交表单至BookServlet类doGet方法，并根据add参数，调用BookService、BookDao执行图书添加操作。具体代码如下：

BookServlet

case "add":

book = new Book();

book.setName(request.getParameter("name"));

book.setAuthor(request.getParameter("author"));

book.setPublisher(request.getParameter("publisher"));

book.setIsbn(request.getParameter("isbn"));

book.setInfo(request.getParameter("info"));

book.setPricing(Double.parseDouble(request.getParameter("pricing")));

book.setIsBorrowed(request.getParameter("is\_borrowed").equals("是"));

bookService.insertOne(book);

request.getRequestDispatcher("/router?page=admin\_book\_management").forward(request, response);

break;

public class BookService {

public BookDao bookDao = new BookDao();

public void insertOne(Book book) {

bookDao.insertOne(book);

}

}

public class BookDao {

public void insertOne(Book book) {

String sql = "INSERT INTO `book` (`name`, `author`, `publisher`, `isbn`, `info`, `pricing`, `is\_borrowed`) VALUES (?, ?, ?, ?, ?, ?, ?)";

JdbcUtil.insert(sql, true, book.getName(), book.getAuthor(), book.getPublisher(), book.getIsbn(), book.getInfo(), book.getPricing(), book.getIsBorrowed());

}

}

#### 图书类别管理、添加功能的实现

图书类别管理功能包括图书类别的查询、删除、修改。Dao、Service默认查询所有图书类别信息返回前端，并展示在前端页面上。条件查询根据前端传递至服务器的类别名参数，SQL语句使用Like进行模糊匹配，返回相应类别信息至前端。点击相应图书类别的删除按钮后，将图书类别ID传递至后端，通过调用Service层方法执行图书类别删除操作，执行删除操作之前判断该图书类别之下是否存在图书类别，如存在则返回失败信息。图书类别修改在后端获取到前端传递的参数后，返回该图书类别信息至前端，并展示在相应图书类别信息文本框中，以方便修改图书类别信息。输入需要修改的信息并提交表单后，后端调用相应方法执行图书类别信息修改操作。在图书类别添加页面相应文本框中输入需要添加图书类别信息后提交表单，后端根据前端参数向数据库中添加图书类别信息。具体代码如下：

@WebServlet("/bookType")

public class BookTypeServlet extends HttpServlet {

BookTypeService bookTypeService = new BookTypeService();

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

doPost(request, response);

}

@Override

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String method = request.getParameter("method");

switch (method) {

case "condition":

String name = request.getParameter("name");

request.setAttribute("types", bookTypeService.selectByNameLike(name));

request.getRequestDispatcher("/router?page=admin\_book\_type\_management").forward(request, response);

break;

case "update":

BookType bookType = new BookType();

bookType.setId(Integer.parseInt(request.getParameter("id")));

bookType.setName(request.getParameter("name"));

bookTypeService.updateOne(bookType);

request.getRequestDispatcher("/router?page=admin\_book\_type\_management").forward(request, response);

break;

case "delete":

int id = Integer.parseInt(request.getParameter("id"));

bookTypeService.deleteOne(id);

request.getRequestDispatcher("/router?page=admin\_book\_type\_management").forward(request, response);

break;

case "add":

name = request.getParameter("name");

bookType = new BookType();

bookType.setName(name);

bookTypeService.insertOne(bookType);

request.getRequestDispatcher("/router?page=admin\_book\_type\_management").forward(request, response);

break;

}

}

}

#### 借阅信息记录功能的实现

当普通用户在系统中借阅、归还图书后，相应方法内会调用相应方法向数据库borrowing表中插入用户借阅信息，记录借阅（归还）图书的用户ID、借阅（归还）图书ID、操作、操作时间。作为普通用户登录系统时，借阅记录页面展示该用户以往所有借阅记录，作为管理员登录系统时，借阅记录页面展示所有用户所有借阅记录（根据时间顺序）。后端在BookServlet类doGet方法内相应参数操作内，调用Service、Dao层代码执行记录插入操作。具体代码如下：

@WebServlet("/book")

public class BookServlet extends HttpServlet {

BookService bookService = new BookService();

BorrowingService borrowingService = new BorrowingService();

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

doPost(request, response);

}

@Override

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String method = request.getParameter("method");

switch (method) {

case "condition":

String author = request.getParameter("author");

String name = request.getParameter("name");

Book book = new Book();

book.setAuthor(author);

book.setName(name);

request.setAttribute("books", bookService.selectCondition(book));

request.getRequestDispatcher("/router?page=book").forward(request, response);

break;

case "borrow":

int id = Integer.parseInt(request.getParameter("id"));

// 借阅

if (bookService.selectOne(id).getIsBorrowed()) {

throw new RuntimeException("借阅失败");

}

bookService.bookBorrow(id, true);

// 插入借阅记录

Borrowing borrowing = new Borrowing();

borrowing.setUserId(CommonUtil.getUserId());

borrowing.setBookId(id);

borrowing.setType("borrowing");

borrowing.setDatetime(new Timestamp(new Date().getTime()));

borrowingService.insertOne(borrowing);

// 跳转

request.getRequestDispatcher("/router?page=book").forward(request, response);

break;

case "return":

int returnId = Integer.parseInt(request.getParameter("id"));

// 借阅

if (!bookService.selectOne(returnId).getIsBorrowed()) {

throw new RuntimeException("归还失败");

}

bookService.bookBorrow(returnId, false);

// 插入借阅记录

borrowing = new Borrowing();

borrowing.setUserId(CommonUtil.getUserId());

borrowing.setBookId(returnId);

borrowing.setType("returning");

borrowing.setDatetime(new Timestamp(new Date().getTime()));

borrowingService.insertOne(borrowing);

// 跳转

request.getRequestDispatcher("/router?page=book").forward(request, response);

break;

case "add":

book = new Book();

book.setName(request.getParameter("name"));

book.setAuthor(request.getParameter("author"));

book.setPublisher(request.getParameter("publisher"));

book.setIsbn(request.getParameter("isbn"));

book.setInfo(request.getParameter("info"));

book.setPricing(Double.parseDouble(request.getParameter("pricing")));

book.setIsBorrowed(request.getParameter("is\_borrowed").equals("是"));

bookService.insertOne(book);

request.getRequestDispatcher("/router?page=admin\_book\_management").forward(request, response);

break;

case "delete":

id = Integer.parseInt(request.getParameter("id"));

bookService.deleteOne(id);

request.getRequestDispatcher("/router?page=admin\_book\_management").forward(request, response);

break;

case "update":

book = new Book();

book.setId(Integer.parseInt(request.getParameter("id")));

book.setName(request.getParameter("name"));

book.setAuthor(request.getParameter("author"));

book.setPublisher(request.getParameter("publisher"));

book.setIsbn(request.getParameter("isbn"));

book.setInfo(request.getParameter("info"));

book.setPricing(Double.parseDouble(request.getParameter("pricing")));

book.setIsBorrowed(request.getParameter("is\_borrowed").equals("是"));

bookService.update(book);

request.getRequestDispatcher("/router?page=admin\_book\_management").forward(request, response);

break;

}

}

}