第 5 章

CHAPTER 5

SSM 框架整合实战

5.1 SSM 框架整合概述

随着现代软件开发的快速发展，框架技术已成为提升开发效率和代码质量的关键。在众多框架中，SSM 以其轻量级、易扩展和高度解耦的特性，受到了广大开发者的青睐。SSM 框架整合了 Spring 的依赖注入(DI)和面向切面编程(AOP)功能，Spring MVC 的 MVC 设计模式，以及 MyBatis 的 ORM(对象关系映射)能力，形成了一套高效、稳定且灵活的开发解决方案。

5.1.1 框架基础回顾

在软件开发领域，框架是一种预先设计好的、可重用的软件结构，它为开发者提供了一套完整的解决方案，用于解决某类特定问题。框架通常包含一系列预定义的类和接口，以及相关的设计模式、最佳实践等，旨在简化开发过程，提高代码质量，加速软件的开发速度，其中，Spring、Spring MVC 和 MyBatis 是 Java Web 开发领域中的 3 个重要框架，它们各自有着独特的优势和功能，但又能够相互协作，共同构建出高效、稳定、可维护的 Web 应用程序。

Spring 框架是一个全面的、轻量级的开源框架，它提供了丰富的功能并具有强大的扩展性，用于解决企业级应用开发的各种问题。Spring 的核心思想是控制反转(IoC)和面向切面编程(AOP)，通过这两个核心概念，Spring 实现了组件之间的解耦和横切关注点的分离，使应用程序更加灵活、可维护。此外，Spring 还提供了事务管理、数据访问、安全性、Web 集成等一系列功能，为开发者提供了全方位的支持。

Spring MVC 是 Spring 框架中的一个模块，它实现了 MVC(Model-View-Controller)设计模式，用于构建 Web 应用程序的视图层。Spring MVC 通过 DispatcherServlet 作为前端控制器，统一处理用户的请求，并根据请求的不同调用相应的控制器。控制器负责处理业务逻辑，并返回相应的模型数据。最后，Spring MVC 通过视图解析器将模型数据渲染为视图，呈现给用户。这种设计使 Web 应用程序的层次结构更加清晰，提高了代码的可读性和可维护性。

MyBatis是一个优秀的持久层框架，它支持定制化SQL、存储过程及高级映射。MyBatis通过XML或注解的方式配置SQL语句和映射规则，将Java对象与数据库表进行映射，实现了Java对象与数据库之间的自动转换。这使开发者能够专注于业务逻辑的实现，而无须过多地关注底层的数据访问细节。同时，MyBatis还提供了动态SQL功能，可以根据不同的条件生成不同的SQL语句，提高了查询的灵活性和效率。

在将Spring、Spring MVC和MyBatis进行整合时，通常将MyBatis作为数据持久层框架，用于处理与数据库相关的操作；Spring作为业务逻辑层框架，管理业务对象及其之间的依赖关系，而Spring MVC则作为Web层框架，负责处理用户的请求和响应。这种整合方式使应用程序的层次结构更加清晰、合理，各个层次之间的职责更加明确，提高了代码的可读性和可维护性。

此外，整合后的框架还具备以下优势：

(1) 灵活性：Spring、Spring MVC和MyBatis都是高度可配置的框架，可以根据项目的具体需求进行定制和扩展。

(2) 性能优化：通过Spring的AOP特性，可以实现性能监控、日志记录等功能，进一步提升系统的性能。同时，MyBatis的灵活SQL编写和映射功能也能提高数据访问的效率。

(3) 安全性保障：Spring框架提供了强大的安全性支持，包括身份验证、授权、加密等功能，可以确保系统的安全性。

5.1.2 框架整合的必要性

在当前的软件开发实践中，随着Web应用程序规模的扩大和业务需求的日益复杂化，单一框架往往难以全面满足开发者的需求。这种局限性主要体现在功能覆盖不全、性能瓶颈及维护难度增高等方面，因此，为了实现更加高效、稳定和灵活的开发过程，对多个优秀框架进行有机整合成为必然的选择。

Spring、Spring MVC和MyBatis作为Java Web开发领域的三大主流框架，各自在特定的领域具有显著的优势。Spring以其强大的依赖注入和面向切面编程特性，简化了业务逻辑的开发；Spring MVC通过实现MVC设计模式，优化了Web层的开发流程，而MyBatis则以其高效的数据库操作能力和灵活的SQL映射机制，简化了数据持久层的工作，然而，仅仅使用单一的框架往往无法充分发挥这些优势。例如，单独使用Spring虽然可以实现业务逻辑的高效管理，但在Web层和数据持久层的开发上可能显得力不从心；同样，单独使用Spring MVC或MyBatis也无法解决业务逻辑管理和数据访问的复杂性问题。

因此，对Spring、Spring MVC和MyBatis进行整合，以形成一个统一的开发框架，成为解决这些问题的有效途径。通过整合，开发者可以充分地利用各框架的优势，实现业务逻辑、Web层和数据持久层的无缝衔接，提高代码的质量和可维护性。同时，整合后的框架还可以提供更加灵活和可扩展的开发环境，使开发者能够更加轻松地应对业务需求的变化。

随着科技领域的持续进步与技术的日新月异，新的开发框架与工具层出不穷，为软件开发领域带来了源源不断的创新活力，然而，即便在如此繁多的新技术涌现的背景下，Spring、

Spring MVC 及 MyBatis 这三大经典框架依然以其深厚的底蕴和广泛的实践应用，彰显出无可替代的重要地位。这些框架经过长时间的验证与打磨，其稳定性和可靠性得到了业界的广泛认可，成为软件开发领域中的常青树。鉴于它们在业界的重要地位与卓越表现，对它们进行深度融合并持续应用于 Web 应用程序的开发中，不仅有助于提升开发效率，更能确保项目的长期稳定运行。这种策略不仅是对技术发展趋势的精准把握，更是对未来技术发展的长远规划与布局，因此，在追求技术创新的同时，坚持使用并不断完善这些经典框架，将是一种极具智慧与远见的投资策略。

5.1.3　整合后的框架功能

整合 Spring、Spring MVC 和 MyBatis 后，开发者得到的是一个功能强大、结构清晰的综合性框架。这一整合不仅充分地发挥了各框架的专长，还通过协同工作提升了整体性能，使开发出的 Web 应用程序更高效、更稳定且易于维护。

在整合后的框架中，Spring 作为基础框架，提供了全面的支持。它利用依赖注入机制，实现了组件之间的松耦合，使应用程序的各部分能够独立开发、测试和维护。同时，Spring 的事务管理功能确保了数据的完整性和一致性，无论是单个数据库操作还是跨多个操作的复杂事务都可以得到妥善处理。

Spring MVC 作为 Web 层框架，与 Spring 实现了无缝集成。这种集成使 Web 开发能够充分地利用 Spring 提供的丰富功能。通过 Spring MVC 可以实现请求的高效处理，无论是简单的 GET 请求还是复杂的 POST 请求都可以得到快速响应。同时，Spring MVC 还提供了灵活的视图渲染机制，支持多种视图技术，如 JSP、Thymeleaf 等，使开发者能够根据需要选择合适的视图技术，实现用户界面的多样化。

MyBatis 作为持久层框架，在整合后也发挥了重要作用。它简化了数据库操作代码的编写，通过 XML 或注解的方式配置 SQL 语句和映射规则，实现了 Java 对象与数据库表之间的自动转换。这使开发者能够专注于业务逻辑的实现，而无须过多地关注底层的数据访问细节。此外，MyBatis 还提供了动态 SQL 功能，能够根据条件动态地生成 SQL 语句，提高了查询的灵活性和效率。与 Spring 整合后，MyBatis 可以借助 Spring 的事务管理功能，确保数据库操作的原子性和一致性。无论是单个数据库操作还是涉及多个数据源的复杂操作都可以得到可靠的事务保障。

5.1.4　整合的意义与优势

整合 Spring、Spring MVC 与 MyBatis 三大框架，不仅是技术资源优化利用的明智选择，更是应用程序架构升级与革新的重要举措。这一整合在多个层面展现出显著的意义与优势，为 Web 应用程序的开发、维护及扩展提供了坚实的支撑。

整合后，应用程序的层次结构变得更清晰规范。Spring、Spring MVC 与 MyBatis 各自在业务逻辑、Web 层处理和数据持久化方面发挥专长，形成了层次分明、功能互补的架构体

系。这种结构让开发者能够更直观地理解应用程序的运行机制，从而降低了开发难度和出错率。同时，整合有效地降低了各层次间的耦合度，提升了系统的灵活性和可维护性。通过依赖注入、接口定义等机制，组件间的依赖关系被精准地定义与管理，实现了松耦合的设计。这使应用程序在面对需求变更或技术迭代时，能够迅速适应并调整，降低了维护成本和风险。整合框架汇聚了三大框架的丰富功能与特性，展现出强大的扩展性和灵活性。这些功能和特性在整合过程中相互融合，形成了强大的合力。开发者可以根据业务需求轻松地添加新功能或模块，满足不断变化的市场需求。整合还实现了代码的解耦与模块化，提高了代码质量和可维护性。通过合理划分代码模块与组件，并利用框架的依赖管理与配置机制，代码的复杂性和冗余度得到有效降低。这不仅提高了代码的可读性和可维护性，也提升了开发效率和质量。在性能优化与安全性保障方面，整合后的框架同样表现出色。利用缓存机制、连接池管理等特性，应用程序的性能和响应时间得到优化。同时，框架内置的身份验证、授权、加密等安全功能，确保了系统的稳定运行和数据的安全。

5.1.5 SSM 框架整合思路

在 SSM 框架整合过程中，各个组件承担着明确的职责，共同构建了一个高效、稳定的应用程序结构，但是 Spring MVC 和 MyBatis 并没有直接的交集，它们各自扮演着不同的角色，通过 Spring 框架进行连接和协作，因此，开发者只需分别将 Spring 与 MyBatis 和 Spring MVC 进行整合，便可完成 SSM 框架的整合工作。

以一个用户管理案例为例，SSM 框架整合的实现思路如下。

(1) 搭建项目的基础结构：这包括在数据库中创建项目所需的表结构，搭建项目对应的数据库环境，然后创建一个 Maven Web 项目，并引入案例所需的依赖；最后，创建项目的实体类，并设计三层架构对应的模块、类和接口。

(2) 整合 Spring 和 MyBatis：在 Spring 的配置文件中，配置数据源信息，包括数据库连接池、驱动类名、URL 等，然后配置 SqlSessionFactory 对象，用于创建 SqlSession 实例。最后，将 Mapper 接口及其实现类交由 Spring 管理，实现 Mapper 对象的自动注入和调用。

(3) 整合 Spring 和 Spring MVC：由于 Spring MVC 是 Spring 框架的一个模块，因此整合过程相对简单，只需在项目启动时分别加载 Spring 和 Spring MVC 的配置文件。这样，Spring MVC 就可以通过 Spring 容器获取所需的 Service 对象，实现业务逻辑的处理和响应的返回。

完成上述步骤后，客户端可以向服务器端发送查询请求。如果服务器端能够正确地从数据库中获取数据，并将其响应给客户端，则可以认为 SSM 框架整合成功。这标志着整个应用程序已经构建完成，可以投入实际使用。

5.1.6 搭建 SSM 框架整合的项目基础结构

在构建 SSM 框架整合的项目时，首先需要搭建一个稳固且高效的基础结构。这一基础

结构不仅承载着项目的核心逻辑和数据处理能力，还为后续的扩展与维护提供了坚实的支撑。

1. 创建项目并引入项目依赖

在 IntelliJ IDEA 集成开发环境中，创建一个名为 user-system 的 Maven Web 项目，并在项目的 pom.xml 文件中引入以下依赖，代码如下：

```
//第 5 章/pom.xml
<dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.2.8.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>5.2.8.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.2.8.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>5.2.8.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.2.8.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.2</version>
    </dependency>
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis-spring</artifactId>
      <version>2.0.1</version>
    </dependency>
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid</artifactId>
      <version>1.1.20</version>
    </dependency>
```

```
        <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
          <scope>test</scope>
        </dependency>
        <dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>javax.servlet-api</artifactId>
          <version>3.1.0</version>
          <scope>provided</scope>
        </dependency>
        <dependency>
          <groupId>javax.servlet.jsp</groupId>
          <artifactId>jsp-api</artifactId>
          <version>2.2</version>
          <scope>provided</scope>
        </dependency>
        <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <version>8.0.16</version>
        </dependency>
        <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
          <scope>compile</scope>
    </dependency>
    <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.11</version>
          <scope>test</scope>
        </dependency>
      </dependencies>
```

以下是对上述依赖的介绍。

1）Spring 相关依赖

（1）org.springframework:spring-context：Spring 框架的核心容器模块，提供了依赖注入、事件发布等功能，是构建 Spring 应用程序的基础。

（2）org.springframework:spring-tx：提供了对 Spring 框架的事务管理的支持，包括声明式事务管理和编程式事务管理，帮助开发者轻松地控制和管理数据库事务。

（3）org.springframework:spring-jdbc：Spring JDBC 模块简化了 JDBC 操作，提供了 JdbcTemplate 等工具类，帮助开发者快速地构建数据库访问层。

（4）org.springframework:spring-test：支持 Spring 应用程序单元测试的模块，提供了

对 Spring 组件的测试支持，包括测试上下文加载、依赖注入等。

(5) org.springframework:spring-webmvc：Spring MVC 是 Spring 框架的 Web 模块，用于构建基于 Java 的 Web 应用程序。它提供了模型—视图—控制器(MVC)架构的实现，支持 RESTful Web 服务开发。

2) MyBatis 相关依赖

org.mybatis:mybatis：持久层框架 MyBatis，它支持自定义 SQL、存储过程及高级映射。MyBatis 免除了绝大多数的 JDBC 代码及设置参数和获取结果集的手工操作。

3) MyBatis 与 Spring 整合包

org.mybatis:mybatis-spring：提供了 MyBatis 与 Spring 框架的无缝集成，使开发者能够同时使用 MyBatis 的数据库访问优势及 Spring 的依赖注入和事务管理功能。

4) 数据源相关依赖

com.alibaba:druid：数据库连接池，提供了强大的监控和扩展功能，性能出色，能有效地防止 SQL 注入攻击，是 Java 应用中常用的数据库连接池解决方案。

5) 单元测试相关依赖

junit:junit：Java 编程语言中流行的单元测试框架，它允许开发者编写和运行可重复的自动化测试，以确保代码的正确性和质量。

6) Servlet API 相关依赖

(1) javax.servlet:javax.servlet-api：Java Web 应用程序开发的基础，提供了处理 HTTP 请求和响应的接口和方法。

(2) javax.servlet.jsp:jsp-api：Java Server Pages(JSP)的技术规范，它提供了在 HTML 页面中嵌入 Java 代码以动态生成 Web 页面的功能。

7) 数据库相关依赖

mysql:mysql-connector-java：MySQL 数据库的 JDBC 驱动，它允许 Java 应用程序与 MySQL 数据库进行连接和通信，是进行数据库操作的基础。

这些依赖项旨在确保项目能够稳定构建与高效运行，同时满足项目所需的功能和库支持。这些依赖项涵盖了诸如 Spring、Spring MVC、MyBatis 等核心框架库，以及一系列必要的辅助库和插件，它们共同构成了项目的基础架构。在添加依赖项时，务必确保每个依赖项都包含了准确无误的 groupId、artifactId 和 version 信息，以便 Maven 能够精确无误地下载并引入这些依赖项。

依赖项添加完成后，需要单击 install 导入这些依赖，Maven 将自动执行下载操作并将这些依赖项引入项目中，为后续编写和构建基于 SSM 框架的 Web 应用程序提供支撑，如图 5-1 所示。

2. 搭建项目数据库的环境

使用 MySQL 客户端工具在 MySQL 数据库中创建一个名为 user-system 的数据库，并在该数据库中创建一个名为 user 的数据表，然后向这个数据表中插入数据。

通过客户端工具创建数据库，如图 5-2 所示。

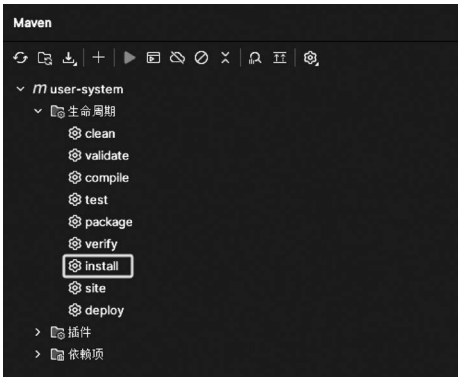

图 5-1 Maven 中的 install

图 5-2 创建数据库

通过客户端工具创建数据表,如图 5-3 所示。

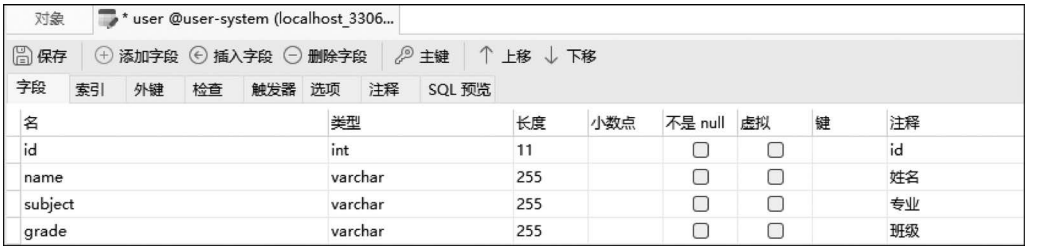

图 5-3 创建数据表

通过客户端工具插入数据,如图 5-4 所示。

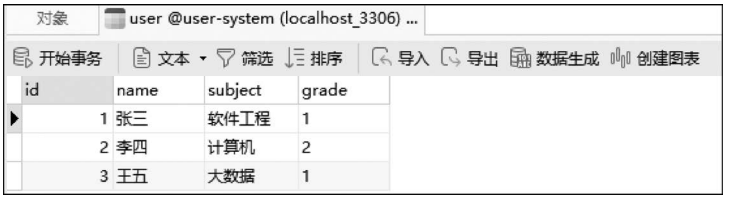

图 5-4 插入数据

以上创建数据库、数据表及向数据表中插入数据的操作也可以通过 SQL 语句实现,代码如下:

```
//第 5 章/user - system.sql
CREATE DATABASE user - system;
USE user - system;

CREATE TABLE `user` (
```

```
  `id` int(11) NULL DEFAULT NULL COMMENT 'id',
  `name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin NULL DEFAULT NULL COMMENT '姓名',
  `subject` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin NULL DEFAULT NULL COMMENT
'专业',
  `grade` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin NULL DEFAULT NULL COMMENT '班
级'
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_bin ROW_FORMAT = Dynamic;

INSERT INTO `user` VALUES (1, '张三', '软件工程', '1');
INSERT INTO `user` VALUES (2, '李四', '计算机', '2');
INSERT INTO `user` VALUES (3, '王五', '大数据', '1');
```

3. 根据数据库内容创建实体类

在项目的 src/main/java 目录下，需要创建一个实体类。首先，创建一个名为 com.demo.domain 的包，用于组织和管理与业务逻辑相关的类。接下来，在 com.demo.domain 包下创建一个名为 User 的实体类。实体类用于映射数据库中的表，它包含了与该表相关的属性和方法，代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/domain/User.java
package com.demo.domain;
public class User {
    //id
    private Integer id;
    //姓名
    private String name;
    //专业
    private String subject;
    //班级
private String grade;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSubject() {
        return subject;
    }
```

```
    public void setSubject(String subject) {
        this.subject = subject;
    }
    public String getGrade() {
        return grade;
    }
    public void setGrade(String grade) {
        this.grade = grade;
    }
}
```

4. 创建三层架构对应模块的类和接口

(1) 在项目的 src/main/java/com/demo 目录下,构建一个名为 dao 的包,这个包将负责实现数据访问对象(DAO)的相关功能。之后在 dao 包内,创建一个名为 UserMapper 的接口,该接口将作为持久层接口,用于与数据库进行交互。在 UserMapper 接口中,定义一个名为 getUserById()的方法,该方法的目的是通过学生 ID 来获取对应的学生信息,代码如下:

```
//第 5 章/user - system/src/main/java/com/demo/dao/UserMapper.java
package com.demo.dao;
import com.demo.domain.User;
public interface UserMapper {
    public User getUserById(Integer id);
}
```

之后,在项目的 src/main/resources 目录下,创建一个 com.demo.dao 的文件夹结构,并在该文件夹下创建 UserMapper 接口对应的映射文件 UserMapper.xml。这个映射文件是 MyBatis 框架中用于定义 SQL 语句与 Java 方法之间映射关系的关键部分,它确保了数据访问层能够正确地执行数据库操作。UserMapper.xml 映射文件的具体实现代码如下:

```
//第 5 章/user - system/src/main/resources/com/demo/dao/UserMapper.xml
<?xml version = "1.0" encoding = "utf - 8" ?>
<!DOCTYPE mapper
        PUBLIC " - //mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis - 3 - mapper.dtd">
<mapper namespace = "com.demo.dao.UserMapper">
    <!-- 通过学生 ID 来获取对应的学生信息 -->
    <select id = "getUserById" parameterType = "int"
            resultType = "com.demo.domain.User">
        select *
        from user
        where id = #{id}
    </select>
</mapper>
```

(2) 在项目的 src/main/java/com/demo 目录下,构建一个名为 service 的包,这个包将负责实现业务逻辑层的相关功能。之后,在 service 包内,创建一个名为 UserService 的接

口，作为业务逻辑层的核心接口，用于处理与用户相关的业务操作，并且在 UserService 接口中，定义一个名为 getUserById()的方法，该方法的目的是通过学生 ID 来获取对应的学生信息，代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/service/UserService.java
package com.demo.service;
import com.demo.domain.User;
public interface UserService {
    public User getUserById(Integer id);
}
```

在项目的 src/main/java/com/demo/service 目录下，构建一个名为 impl 的子包，该包将用于存放业务逻辑层接口的具体实现类。在 impl 包内，创建一个名为 UserServiceImpl 的类，该类将作为 UserService 接口的业务层实现。在 UserServiceImpl 类中，实现了 UserService 接口所定义的 getUserById()方法，并在类中注入了一个 UserMapper 对象。这个 UserMapper 对象是数据访问层的关键组件，它负责执行与数据库相关的操作。在 getUserById()方法的实现中，通过注入的 UserMapper 对象调用其 getUserById()方法，并传入学生 ID 作为参数，从而获取对应的学生信息。通过这种方式来实现业务逻辑层与数据访问层的交互，确保业务逻辑的正确执行。UserServiceImpl 类的具体实现代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/service/impl/UserServiceImpl.java
package com.demo.service.impl;

import com.demo.dao.UserMapper;
import com.demo.domain.User;
import com.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserMapper userMapper;

    public User getUserById(Integer id) {
        return userMapper.getUserById(id);
    }
}
```

(3) 在项目的 src/main/java/com/demo 目录下，创建一个名为 controller 的包，以组织和管理与 Web 请求处理相关的控制器类，并在 controller 包下创建一个名为 UserController 的类，该类将作为 Web 请求的前端控制器，负责处理与用户相关的 HTTP 请求。在 UserController 类中，注入了一个 UserService 对象，这个对象提供业务逻辑层的功能，能够处理复杂的业务规则和数据交互。之后在 UserController 类中定义一个名为 getUserById 的方法。这种方法的目的是响应前端发送的获取学生信息的请求。当方法被

调用时，它获取请求中传递过来的学生 ID 作为参数，并将该参数传递给注入的 UserService 对象调用的 getUserById 方法。通过调用业务逻辑层的方法确保能够按照正确的业务规则获取学生信息，具体实现代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/controller/UserController.java
package com.demo.controller;

import com.demo.domain.User;
import com.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/user")
    public ModelAndView findBookById(Integer id) {
        User user = userService.getUserById(id);
        ModelAndView view = new ModelAndView();
        view.setViewName("user.jsp");
        view.addObject("user", user);
        return view;
    }
}
```

5.2 Spring 与 MyBatis 的整合配置

在 SSM 框架整合中，Spring 与 MyBatis 的整合是至关重要的一环。通过将 Spring 的依赖注入和事务管理与 MyBatis 的持久层操作相结合，可以实现高效、可维护的数据访问层。Spring 与 MyBatis 的整合过程可以分为两个主要步骤：首先需要搭建 Spring 框架环境，随后需要将 MyBatis 无缝地集成到已搭建的 Spring 环境中。在这个整合过程中，框架环境的构建至关重要，它涉及框架所需的各种依赖和配置文件的准备工作。具体来讲，这包括 Spring 的核心依赖、MyBatis 的数据库操作依赖，以及确保两者顺畅协作的整合依赖。在项目的初始结构搭建阶段，这些依赖通常已经被适当地引入项目中。接下来，开发者需要聚焦于配置文件的编写工作，这包括 Spring 的核心配置文件，以及用于定义 Spring 与 MyBatis 之间交互细节的整合配置文件。通过这些配置文件的精确编写，可以确保 Spring 和 MyBatis 能够协同工作，从而提供了高效、稳定的数据访问层支持。下面将详细讲解如何对 Spring 和 MyBatis 进行整合，包括配置文件的编写、Bean 的配置及事务管理的设置等。

在 Spring 与 MyBatis 的整合中，主要涉及两个配置文件：applicationContext. xml（Spring 的配置文件）和 mybatis-config. xml（MyBatis 的配置文件）。

5.2.1　Spring 的配置文件

在 Spring 框架中，配置文件扮演着至关重要的角色，它们不仅定义了应用程序的上下文环境，还指导了 Spring 容器如何加载和管理 Bean。为了配置 Spring 服务层的 Bean，需要创建一个名为 application-service. xml 的配置文件。application-service. xml 的主要职责是指导 Spring 容器在启动时扫描特定的包路径。通过配置这些扫描路径，Spring 能够自动检测和注册 Service 层中定义的 Bean，从而极大地简化在开发过程中的配置工作。

application-service. xml 文件中的示例代码如下：

```
//第 5 章/user - system/src/main/resources/application - service.xml
< beans xmlns = "http://www.springframework.org/schema/beans"
           xmlns:xsi = "http://www.w3.org/2001/XMLSchema - instance"
           xmlns:context = "http://www.springframework.org/schema/context"
           ...>
    <!-- 开启注解扫描，扫描包 -->
    < context:component - scan base - package = "com.test.service"/>
</beans >
```

在上述配置中，< context：component-scan >元素用于开启注解扫描功能，并指定了需要扫描的包路径 com. test. service。这意味着 Spring 容器会扫描 com. test. service 包及其子包下的所有类，并自动注册带有@Service 注解的类作为 Bean。通过这种方式，开发者无须手动编写每个 Service 类的 Bean 定义，从而提高开发效率和代码的可维护性。

除了配置注解扫描外，application-service. xml 文件中还可以添加其他配置，如数据源配置、事务管理等。这些配置将根据具体的应用需求进行定制和扩展。

5.2.2　jdbc. properties 的属性文件

在 Spring 与 MyBatis 的集成方案中，SqlSessionFactoryBean 是一个核心组件，其重要性不言而喻。这个 Bean 不仅负责创建 SqlSessionFactory 的实例，而且作为 Spring 与 MyBatis 之间的桥梁，负责将 MyBatis 集成到 Spring 框架中。为了充分发挥 SqlSessionFactoryBean 的功能，必须将其与数据源（DataSource）进行关联，确保数据库连接的正确性和稳定性。通过数据源的配置，可以实现对数据库连接的有效管理，包括连接池的设置、连接的安全性和可靠性等方面的控制。

为了管理和维护数据源信息，通常会创建一个名为 jdbc. properties 的属性文件。这个文件集中存放了数据库连接所需的关键信息，如数据库 URL、用户名、密码等。通过配置 jdbc. properties，可以实现数据库连接信息的集中管理和安全控制，避免在代码中硬编码敏感信息。同时，jdbc. properties 文件还有助于提高代码的可维护性和灵活性，方便对数据库连接进行集中管理和配置。

jdbc.properties 文件中的示例代码如下：

```
//第 5 章/user - system/src/main/resources/jdbc.properties
#数据库连接信息
jdbc.driverClassName = com.mysql.cj.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/ssm?useUnicode = true&characterEncoding =
utf8&serverTimezone = Asia/Shanghai
jdbc.username = myuser
jdbc.password = mypassword

#其他可能的配置,如连接池属性等
jdbc.maxPoolSize = 10
jdbc.minPoolSize = 5
jdbc.initialPoolSize = 5
```

在这个配置中，jdbc.driverClassName 指定了数据库驱动类，jdbc.url 是数据库的连接 URL，jdbc.username 和 jdbc.password 分别是数据库访问的用户名和密码。此外，还可以根据需要配置连接池的相关属性，如最大连接数（jdbc.maxPoolSize）、最小连接数（jdbc.minPoolSize）和初始连接数（jdbc.initialPoolSize）等。通过合理配置 jdbc.properties 文件，并结合 SqlSessionFactoryBean 的设置，可以确保 Spring 与 MyBatis 之间顺畅整合，实现高效、稳定的数据库访问操作。同时，这种配置方式还提高了代码的可维护性和灵活性，便于对数据库连接进行集中管理和配置。

5.2.3 SSM 框架项目中 Spring 与 MyBatis 的整合配置

在构建 SSM 框架项目时，5.1.6 节中已经在项目基础结构搭建阶段将整合所需的依赖项引入项目中，并且创建了对应的数据库和三层架构对应模块的类和接口。接下来，主要需要完成 Spring 的配置文件编写工作，并配置 Spring 与 MyBatis 的整合文件，以确保两者能够无缝衔接，实现业务逻辑与数据访问的协同工作。通过这样的配置能够确保项目的稳定性和高效性，为后续的业务开发奠定坚实的基础。

1. 创建 Spring 的配置文件

在项目的 src/main/resources 目录下，创建一个名为 application-service.xml 的配置文件，该文件的主要作用是配置 Spring 框架对 Service 层的扫描信息。通过定义扫描路径，Spring 能够自动检测和加载在该路径下的 Service 层组件，实现依赖注入和自动装配等功能。application-service.xml 配置文件的具体代码如下：

```
//第 5 章/user - system/src/main/resources/application - service.xml
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns = "http://www.springframework.org/schema/beans"
        xmlns:xsi = "http://www.w3.org/2001/XMLSchema - instance"
        xmlns:context = "http://www.springframework.org/schema/context"
        xsi:schemaLocation = "
                http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans.xsd
                    http://www.springframework.org/schema/context

        http://www.springframework.org/schema/context/spring-context.xsd
                ">
        <!-- 注解扫描 -->
        <context:component-scan base-package="com.demo.service"/>
    </beans>
```

2. 创建 Spring 和 MyBatis 整合配置文件

在项目的 src/main/resources 目录下创建数据源属性文件 jdbc.properties，并在其中配置相应的数据源信息，如数据库 URL、用户名、密码等。通过这样的配置方式，能够实现 Spring 与 MyBatis 的紧密整合，确保数据访问层与业务逻辑层的无缝衔接，具体的代码如下：

```
//第5章/user-system/src/main/resources/jdbc.properties
jdbc.driverClassName=com.mysql.cj.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/user-system?useUnicode=true&characterEncoding=
utf-8&serverTimezone=Asia/Shanghai

jdbc.username=root
jdbc.password=123456
```

接下来，把 MyBatis 整合进 Spring 的框架环境中，以确保两者的无缝协同工作。在项目的 src/main/resources 目录下，创建一个名为 application-dao.xml 的配置文件。该文件的主要作用是配置 Spring 与 MyBatis 的整合信息，包括数据源、SqlSessionFactory 及 Mapper 接口的扫描等关键配置，具体的代码如下：

```
//第5章/user-system/src/main/resources/application-dao.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd
    ">
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
            <property name="driverClassName" value="${jdbc.driverClassName}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
        </bean>
        <bean id="sqlSessionFactory"
            class="org.mybatis.spring.SqlSessionFactoryBean">
```

```
        <property name = "dataSource" ref = "dataSource"/>
    </bean>
    <bean class = "org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name = "basePackage" value = "com.demo.dao"/>
    </bean>
</beans>
```

(1) 引入之前配置的 jdbc.properties 属性文件,代码如下:

```
//第 5 章/user - system/src/main/resources/application - dao.xml
<context:property - placeholder location = "classpath:jdbc.properties"/>
```

(2) 定义一个数据源(DataSource)的 Bean。数据源是应用程序与数据库之间的连接池,负责管理和复用数据库连接,以提高应用程序的性能和响应速度,代码如下:

```
//第 5 章/user - system/src/main/resources/application - dao.xml
    <bean id = "dataSource" class = "com.alibaba.druid.pool.DruidDataSource">
        <property name = "driverClassName" value = " ${jdbc.driverClassName}"/>
        <property name = "url" value = " ${jdbc.url}"/>
        <property name = "username" value = " ${jdbc.username}"/>
        <property name = "password" value = " ${jdbc.password}"/>
</bean>
```

(3) 创建和配置一个 SqlSessionFactory 对象。SqlSessionFactory 是 MyBatis 框架中的一个核心接口,负责创建 SqlSession 实例,而 SqlSession 是执行 SQL 命令、获取映射器(Mapper)及管理事务的核心接口,代码如下:

```
//第 5 章/user - system/src/main/resources/application - dao.xml
<bean id = "sqlSessionFactory"
          class = "org.mybatis.spring.SqlSessionFactoryBean">
        <property name = "dataSource" ref = "dataSource"/>
    </bean>
```

(4) 定义一个 MapperScannerConfigurer 类型的 Bean,这个 Bean 的主要作用是扫描指定包(这里是 com.demo.dao)下的接口,并为这些接口创建动态代理对象,这些代理对象会被自动存储到 Spring 的 IoC(控制反转)容器中,代码如下:

```
//第 5 章/user - system/src/main/resources/application - dao.xml
    <bean class = "org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name = "basePackage" value = "com.demo.dao"/>
    </bean>
```

3. 测试整合结果

通过实施单元测试的方式,对 Spring 和 MyBatis 整合情况进行全面检测与验证。为了达到这一目的,将在项目的 src/test/java 目录下创建一个名为 UserServiceTest 的测试类。这个测试类将专注于检验 Spring 与 MyBatis 的整合效果,确保其协同工作无误,代码如下:

```
//第 5 章/user - system/src/main/java/com/demo/testUserServiceTest.java
package com.demo.test;
```

```
import com.demo.domain.User;
import com.demo.service.UserService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:application-service.xml", "classpath:
application-dao.xml"})
public class UserServiceTest {
    @Autowired
    private UserService userService;
    @Test
    public void getUserById() {
        User user = userService.getUserById(1);
        System.out.println("ID:" + user.getId() + "姓名:" + user.getName() + "专业:" +
user.getSubject() + "班级:" + user.getGrade());
    }
}
```

Spring 与 MyBatis 的整合完成，如图 5-5 所示。

图 5-5　运行结果

5.2.4　注解方式整合 Spring 与 MyBatis

SSM 框架整合传统上依赖于 XML 配置文件与注解的结合使用，然而 Spring 框架的强大之处在于它允许开发者通过注解的方式完全替代 XML 配置，实现纯注解的 SSM 框架整合。这种方式不仅提高了代码的可读性和可维护性，还使配置更加灵活和易于管理。

在纯注解的整合思路中，开发者可以利用配置类代替 XML 配置文件的作用。这些配置类使用 Spring 提供的注解来定义 Bean、扫描组件、配置属性等，从而实现了与 XML 配置文件相同的效果。

使用注解方式整合 Spring 与 MyBatis 需要一个替代 application-dao.xml 的配置类。这个类将负责读取 jdbc.properties 文件中的数据库连接信息，创建 Druid 数据连接池对象，并注入 SqlSessionFactoryBean 中，同时还需要创建一个替代 MapperScannerConfigurer 的配置，用于指定 Mapper 接口的扫描路径。

首先，在项目的 src/main/java/com/demo 目录下创建一个名为 config 的包，专门用于存放项目的配置类。在这个 config 包中，创建了一个名为 JdbcConfig 的类，其主要职责是获取数据库连接信息并定义创建数据源的方法，具体的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/configJdbcConfig.java
package com.demo.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.PropertySource;
import javax.sql.DataSource;

@PropertySource("classpath:jdbc.properties")
public class JdbcConfig {

    @Value("${jdbc.driverClassName}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String userName;
    @Value("${jdbc.password}")
private String password;

    @Bean("dataSource")
    public DataSource getDataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(userName);
        dataSource.setPassword(password);
        return dataSource;
    }
}
```

在 JdbcConfig 类中，利用@PropertySource 注解来读取 jdbc.properties 文件中的数据库连接信息。这个注解的作用等同于 XML 配置中的< context:property-placeholder >元素，它指定了属性文件的加载路径。接着定义了几个私有属性，包括数据库驱动类名、连接 URL、用户名和密码，并通过@Value 注解将这些属性与 jdbc.properties 文件中的对应值进行绑定，这种绑定方式在功能上等同于 XML 配置中的< property >元素。最后，还需要编写一个名为 getDataSource 的方法，并使用`@Bean`注解将其标记为一个 Spring 管理的 Bean。这种方法负责创建并配置一个 DruidDataSource 对象，用于提供数据库连接。在方法内部设置了数据源的各项属性，包括驱动类名、连接 URL、用户名和密码，并返回配置好的数据源对象。

完成 JdbcConfig 类的配置后，还需要在 config 包中定义一个名为 MyBatisConfig 的

类，专门用于配置 MyBatis 的相关组件。该类中包含了两个重要的方法：getSqlSessionFactoryBean()和 getMapperScannerConfigurer()。这两种方法分别负责创建 SqlSessionFactoryBean 对象和 MapperScannerConfigurer 对象，并返给 Spring 容器进行管理，具体的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/MyBatisConfig.java
package com.demo.config;

import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.mapper.MapperScannerConfigurer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import javax.sql.DataSource;

public class MyBatisConfig {
    @Bean
    public SqlSessionFactoryBean getSqlSessionFactoryBean(
            @Autowired DataSource dataSource) {
        SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);
        return sqlSessionFactoryBean;
    }
    @Bean
    public MapperScannerConfigurer getMapperScannerConfigurer() {
        MapperScannerConfigurer mapperScannerConfigurer = new MapperScannerConfigurer();
        mapperScannerConfigurer.setBasePackage("com.demo.dao");
        return mapperScannerConfigurer;
    }
}
```

在上面的代码中，getSqlSessionFactoryBean()方法通过@Bean 注解标识为一个 Spring 管理的 Bean，这意味着 Spring 将负责创建并管理该方法的返回值，并且在该方法中将创建一个 SqlSessionFactoryBean 对象，并通过@Autowired 注解自动装配数据源 DataSource。这个数据源来自 JdbcConfig 类中的配置，它是数据库连接的关键组件，然后通过调用 setDataSource()方法将数据源设置到 SqlSessionFactoryBean 对象中，以完成 MyBatis 的核心连接工厂的配置，这个过程等同于 XML 配置中的< bean class＝"org. mybatis. spring. SqlSessionFactoryBean">及相关的属性设置。

之后，getMapperScannerConfigurer()方法同样通过@Bean 注解标识为一个 Spring 管理的 Bean。在这种方法中创建了一个 MapperScannerConfigurer 对象，它负责扫描指定包下的 Mapper 接口，并自动将它们注册为 Spring 的 Bean。这样就可以在应用程序中直接使用这些 Mapper 接口，而无须手动配置。在方法中通过调用 setBasePackage()方法指定了要扫描的包路径，在示例代码中是 com. demo. dao。这个设置等同于 XML 配置中的< bean class＝"org. mybatis. spring. mapper. MapperScannerConfigurer">及< property name＝

"basePackage" value="com.demo.dao"/>。

通过 MyBatisConfig 类的这两种方法，可以成功地将 MyBatis 的核心连接工厂和 Mapper 扫描配置转换为纯注解形式。这不仅提高了配置的灵活性和可维护性，也使整个 SSM 框架的整合过程更加简洁和清晰。这种配置方式也充分利用了 Spring 框架的特性，使开发者能够更好地管理和控制应用程序的组件。

5.3 Spring 和 Spring MVC 的整合配置

在 SSM 项目中，Spring 作为业务逻辑层，负责提供事务管理、对象管理等功能，而 Spring MVC 作为表示层，负责处理用户的请求和响应。两者的整合是构建 SSM 项目的关键步骤之一，下面将详细介绍如何进行整合。

5.3.1 Spring 与 Spring MVC 的配置文件

1. 配置 web.xml 以加载 Spring 容器

首先，通过< context-param >元素指定 Spring 容器的配置文件的位置。在下面的代码中，配置文件被命名为 applicationContext.xml，并且位于类路径（classpath）下。param-name 标签中的 contextConfigLocation 是 Spring 框架约定的参数名，用于指定配置文件的位置。param-value 标签则包含了配置文件的实际路径，其次，< listener >元素用于配置 ContextLoaderListener。这个监听器负责在 Web 应用启动时加载 Spring 容器。当 Web 应用服务器启动时，它会扫描 web.xml 文件中的监听器配置，并自动创建和初始化这些监听器。当 ContextLoaderListener 被初始化时，它会读取前面通过< context-param >指定的配置文件，并据此创建和配置 Spring 容器。通过这些配置，业务逻辑组件，如服务层组件等，就可以通过 Spring 容器进行管理。Spring 容器会负责这些组件的实例化、依赖注入及生命周期管理，从而简化了应用的开发和维护。

此外，在配置文件中还可以使用< servlet-mapping >元素进行配置，它将前端控制器（dispatcherServlet）映射到根 URL 路径上，使所有的请求都将首先被 dispatcherServlet 处理，再根据请求的路径和配置，将请求分发到相应的控制器进行处理。

除了上述核心配置外，web.xml 文件中还可以包含其他 Web 应用的配置，如过滤器、安全设置等，这些配置共同确保了 Web 应用的正常运行和安全性，其示例代码如下：

```
//第 5 章/user-system/src/main/webapp/WEB-INFweb.xml
<web-app ...>
    <!-- 配置 Spring 的上下文配置文件位置 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
```

```
    <!-- 配置 Spring 的 ContextLoaderListener 来加载 Spring 容器 -->
    <listener>

    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>
    <servlet-mapping>
        <servlet-name>dispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <!-- 其他 web.xml 配置 -->
</web-app>
```

2. 编写 Spring MVC 配置文件

在 Spring MVC 框架中，Spring MVC 的配置文件扮演着至关重要的角色，它负责定义和配置 Spring MVC 的核心组件和行为。通过编写此配置文件，能够精确地控制 Spring MVC 如何处理 HTTP 请求、如何解析视图及如何扫描和注册控制器。

首先需要在配置文件中启用注解驱动，这可以通过添加<mvc:annotation-driven />元素来实现。这一步骤至关重要，因为它开启了 Spring MVC 对诸如@Controller、@RequestMapping 等注解的支持，使开发者可以使用注解来定义控制器和映射请求。其次，需要配置视图解析器，以便 Spring MVC 能够将逻辑视图名称解析为实际的视图资源，可以使用 InternalResourceViewResolver 类来实现 JSP 视图的解析。通过设置 prefix 和 suffix 属性指定视图文件所在的基础路径和文件扩展名。这样，当 Spring MVC 需要渲染一个视图时，它会根据这些配置找到对应的 JSP 文件。此外，还需要配置控制器的扫描路径，通过<context:component-scan>元素来指定 Spring MVC 应该扫描哪些包以查找带有@Controller 注解的类。例如，将 base-package 属性设置为 com.demo.controller，这意味着 Spring MVC 将扫描此包及其子包下的所有类，并将带有@Controller 注解的类注册为控制器。

除了上述基本配置外，spring-mvc-servlet.xml 文件还可以包含其他 Spring MVC 相关的配置，例如拦截器的定义、消息转换器的配置等。这些配置可以根据项目的具体需求进行定制和扩展，其示例代码如下：

```
//第 5 章/spring-mvc-servlet.xml
<beans ...>
    <!-- 配置注解驱动，开启对@Controller 等注解的支持 -->
    <mvc:annotation-driven />

    <!-- 配置视图解析器 -->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <!-- 配置控制器扫描路径 -->
```

```
    <context:component-scan base-package="com.demo.controller" />

    <!-- 其他 Spring MVC 配置,如拦截器、消息转换器等 -->
</beans>
```

5.3.2 SSM 框架项目中 Spring 和 Spring MVC 的整合配置

Spring 与 Spring MVC 的整合过程相对简洁高效,完成相关依赖的导入后,核心任务在于加载各自所需的配置文件。在之前整合 Spring 和 MyBatis 时,已经配置了 Spring 的各类组件和属性,确保业务逻辑层和数据访问层的顺畅运作。接下来,将 Spring MVC 融入这一体系,仅需确保在项目启动时正确加载 Spring 容器及其配置文件。

1. Spring MVC 的配置文件

在项目的 src/main/resources 目录下创建 Spring MVC 的配置文件并命名为 spring-mvc.xml,在这个文件添加配置及其他与 Spring MVC 相关的设置,通过正确配置这个文件,可以确保 Spring MVC 框架能够正常工作,并与 Controller 层进行有效交互。

在 Spring MVC 的配置文件中需要配置包扫描(package scanning),它指定了 Spring MVC 需要扫描哪些包来查找 Controller 层的类,通过精确指定包路径,可以确保只有包含 Controller 的类被扫描并注册到 Spring MVC 的上下文中,从而实现了对 Controller 层的有效管理。其次,还需要配置注解驱动(annotation-driven),使项目在启动时能够启用注解驱动功能。通过启用注解驱动,Spring MVC 能够自动注册 HandlerMapping 和 HandlerAdapter,从而实现了请求映射和处理器适配的自动化,简化了配置过程,提高了项目的可维护性和可扩展性。

具体配置代码如下:

```
//第 5 章/user-system/src/main/resources/spring-mvc.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans

    http://www.springframework.org/schema/beans/spring-beans.xsd
                http://www.springframework.org/schema/mvc
                http://www.springframework.org/schema/mvc/spring-mvc.xsd
                http://www.springframework.org/schema/context

    http://www.springframework.org/schema/context/spring-context.xsd">
    <!-- 扫描包 -->
    <context:component-scan base-package="com.demo.controller"/>
    <!-- 注解驱动 -->
    <mvc:annotation-driven/>
</beans>
```

2. web.xml 文件

接下来需要在项目的 web.xml 文件中配置 Spring 的监听器。这一监听器负责在 Web 应用启动时初始化 Spring 容器，并加载 Spring 的配置文件。通过这一配置，可以确保 Spring 框架的核心组件得到正确初始化，为后续的 Web 请求处理提供坚实的支撑。首先，在 web.xml 文件中声明一个 context-param 元素，用于指定 Spring 配置文件的位置，然后配置一个 listener 元素，将其 listener-class 属性设置为 org.springframework.web.context.ContextLoaderListener，这样 Spring 容器就会在应用启动时自动加载在配置文件中定义的 bean。

由于在之前已经完成了 spring-mvc.xml 文件的配置，因此也需要在`web.xml`中配置 Spring MVC 的前端控制器(通常称为 DispatcherServlet)。它是 Spring MVC 框架的核心组件，负责拦截请求，将请求分发到相应的控制器，并返回响应。在初始化这个前端控制器时，需要加载 Spring MVC 的配置文件，以便 Spring MVC 能够正确运行。

具体配置代码如下：

```
//第 5 章/user-system/src/main/webapp/WEB-INF/web.xml
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:application-*.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

在进行了上述的配置之后，项目在启动时会根据< context-param >元素的< param-value >子元素所指定的参数值，自动加载位于类路径(classpath)下所有以“application-”为前缀且以“.xml”为后缀的配置文件。这一过程确保了项目能够正确地识别并加载所需的Spring配置文件，从而建立起完整的Spring容器上下文环境，为后续的业务逻辑处理和数据访问操作提供必要的支持和保障。

3. 创建前端页面

在项目的src/main/webapp目录下创建了一个名为user.jsp的JSP(Java Server Pages)文件。这个文件将充当前端页面的角色，其主要职责是展示后端处理器经过处理并返回的学生信息。为了实现这一功能，可以利用JSP的标签库和表达式语言(EL)动态地从后端获取实时数据，并将其集成到页面的相应位置。最后通过页面查询学生信息来测试SSM框架的整合情况。如果页面成功查询到了学生信息，则将表明Controller层有效地将从Service层获取的学生信息传递给了前端页面，也证明了SSM框架整合的成功。

具体的代码如下：

```
//第5章/user-system/src/main/webapp/WEB-INF/user.jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" isELIgnored="false"
%>
<html>
<head><title>学生信息</title></head>
<body>
<table border="1">
    <tr>
        <th>ID</th>
        <th>姓名</th>
        <th>专业</th>
        <th>班级</th>
    </tr>
    <tr>
        <td>${user.id}</td>
        <td>${user.name}</td>
        <td>${user.subject}</td>
        <td>${user.grade}</td>
    </tr>
</table>
</body>
</html>
```

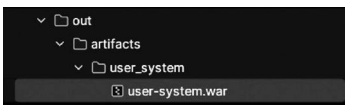

图 5-6 打包结果

4. 测试整合结果

在进行测试之前，需要将项目构建成可执行的WAR包，将项目的所有依赖项、配置文件及源代码打包成一个单独的文件，如图5-6所示。

打包后需要将这个WAR包部署到Tomcat服务器中。在部署过程中，需要将JAR包

放置在 Tomcat 的特定目录下，如图 5-7 所示。

图 5-7　部署到 Tomcat 服务器

完成部署后，通过 bin 目录下的 startup.bat 脚本启动 Tomcat 服务器（Windows 环境下），使项目得以在服务器上运行。在浏览器中访问地址 http://localhost:8080/user?id=1 来查询学生信息，如图 5-8 所示。

ID	姓名	专业	班级
1	张三	软件工程	1

图 5-8　运行结果

5.3.3　注解方式整合 Spring 和 Spring MVC

1. 替代 application-service.xml 配置类

在纯注解的整合思路中，需要利用配置类代替 XML 配置文件。在使用注解方式整合 Spring 和 Spring MVC 的过程中，首先需要一个替代 application-service.xml 的配置类，这个类将负责配置 Service 层的包扫描，指定 Spring 需要扫描的包路径，以便自动发现和注册 Service 层的 Bean。

在 config 包中创建一个名为 SpringConfig 的类，作为项目定义 Bean 的源头，并负责扫描 service 层对应的包。SpringConfig 类不仅是一个简单的 Java 类，它更是 Spring 框架中配置 Bean 定义的关键入口点，通过 Spring 的注解能够在 Java 代码中直接定义和管理 Bean，避免了烦琐的 XML 配置，具体的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/config/SpringConfig.java
package com.demo.config;

import org.springframework.context.annotation.*;

@Configuration
```

```
@Import({MyBatisConfig.class, JdbcConfig.class})
@ComponentScan(value = "com.demo.service")
public class SpringConfig {
}
```

在上述代码中 SpringConfig 类被标记为@Configuration，这意味着它定义了一个或多个@Bean 方法，并且可以被 Spring 容器处理以生成 Bean 定义和服务请求。此外，通过@Import 注解将 MyBatisConfig 类和 JdbcConfig 类导入当前的配置类中，这样它们的 Bean 定义也会被 Spring 容器所管理。这种导入机制允许将复杂的配置分解为多个小的、可管理的配置类，提高了配置的可读性和可维护性。@ComponentScan 注解用于告诉 Spring 容器要扫描哪个包以查找带有@Component、@Service、@Repository 和@Controller 等注解的类，并将它们注册为 Spring 容器中的 Bean。在本例中指定了 com.demo.service 作为扫描的包路径，这意味着 Spring 将自动检测该包及其子包下带有上述注解的类，并创建相应的 Bean 实例。

通过这种方式，SpringConfig 类简化了 Spring 的配置过程，并使配置更加灵活和易于管理。它允许开发者直接在 Java 代码中定义 Bean，避免了 XML 配置的复杂性，同时也提高了代码的可读性和可维护性。通过将配置分解为多个小的配置类，并使用@Import 注解进行导入，实现了配置的模块化，使每个配置类都专注于特定的功能或模块，提高了代码的复用性和可测试性。

2. 替代 spring-mvc.xml 配置类

在完成 SpringConfig 类的编写后，需要再编写一个替代 spring-mvc.xml 的配置类，这个类将配置 Spring MVC 的组件扫描路径和注解驱动，确保 Controller 层能够正确地被 Spring MVC 管理。

在 config 包中，创建了一个名为 SpringMvcConfig 的类，这个类专门用于配置 Spring MVC 框架。开发者可以通过 SpringMvcConfig 类去控制 Spring MVC 的各方面，包括 Controller 层的组件扫描路径。相比 XML 配置，这种配置方式提供了更灵活和可维护的 Spring MVC 设置，具体的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/configSpringMvcConfig.java
package com.demo.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@ComponentScan("com.demo.controller")
@EnableWebMvc
public class SpringMvcConfig {
}
```

在上述代码中，SpringMvcConfig 类被标记为@Configuration，表示它是一个配置类，

用于定义 Bean。在配置类中，还使用了@ComponentScan 注解来指定 Controller 层的扫描路径。通过@ComponentScan("com. demo. controller")注解告诉 Spring MVC 在 com. demo. controller 包及其子包中查找带有@Controller 注解的类，并将它们作为 Controller 组件进行注册。这种方式与 XML 配置中的< context:component-scan base-package = "com. demo. controller"/>具有相同的效果，但更加简洁和直观。此外，SpringMvcConfig 类还使用了@EnableWebMvc 注解，该注解用于启用 Spring MVC 的配置支持，它告诉 Spring 容器要使用 Spring MVC 的功能，并触发相关的自动配置和组件注册。虽然这与 XML 配置中的< mvc:annotation-driven/>有些相似，但@EnableWebMvc 注解提供了更强大和更灵活的配置能力，它不仅局限于注解驱动的配置，还可以结合其他配置选项来满足更复杂的 Web MVC 需求。

3. 替代 web. xml 的配置类

为了确保项目在初始化 Servlet 容器时能够加载特定的初始化信息，并以此来替代传统的 web. xml 在配置文件中的设置，需要利用 Spring 框架提供的一种高级特性。Spring 框架中有一个非常有用的抽象类，名为 AbstractAnnotationConfigDispatcherServletInitializer。这个抽象类为开发者提供了一种在项目启动时自动配置 DispatcherServlet、初始化 Spring MVC 容器及 Spring 容器的机制。通过继承这个抽象类可以避免烦琐的 XML 配置，直接以 Java 配置的方式来设定 Spring MVC 的映射路径，并加载相关的配置类信息。

在项目中创建了一个名为 ServletContainersInitConfig 的类，该类继承了 AbstractAnnotationConfigDispatcherServletInitializer 抽象类。通过重写抽象类中的方法来实现对项目的特定配置。

其中需要重写的方法包括以下几种。

(1) getRootConfigClasses()方法：该方法用于将 Spring 配置类的信息加载到 Spring 容器中，通过返回包含 Spring 配置类的数组来确保 Spring 容器在初始化时能够加载到正确的配置信息。

(2) getServletConfigClasses()方法：该方法用于将 Spring MVC 配置类的信息加载到 Spring MVC 容器中，通过返回包含 Spring MVC 配置类的数组来确保 Spring MVC 容器能够加载到正确的配置信息。

(3) getServletMappings()方法：该方法用于指定 DispatcherServlet 的映射路径，通过返回映射路径的字符串数组来定义哪些 URL 请求会被 DispatcherServlet 处理。

具体的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/config/ServletContainersInitConfig.java
package com.demo.config;

import org.springframework.web.servlet.support.
        AbstractAnnotationConfigDispatcherServletInitializer;

public class ServletContainersInitConfig extends
```

```
        AbstractAnnotationConfigDispatcherServletInitializer {

    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfig.class};
    }

    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{SpringMvcConfig.class};
}

    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}
```

在项目启动时，这个文件会被自动加载，加载完成后，它会触发 Spring MVC 容器和 Spring 容器的初始化过程，并加载对应的配置类信息。同时，它还会配置好 DispatcherServlet 的映射路径，确保请求能够正确地被分发到相应的处理器。

通过上述的配置过程，可以成功地将 SSM 框架原有的 XML 配置模式转换为纯注解配置，这个操作不仅简化了配置流程，而且大幅地增强了代码的可读性和可维护性，使项目结构更加清晰，易于理解与管理。同时，这也充分展示了 Spring 框架在配置灵活性方面的卓越优势，它允许开发者根据项目的实际需求，灵活选择最适合的配置方式，无论是注解还是 XML 都可以得到良好的支持。然而，在实际开发中也需要注意，纯注解配置虽然带来了诸多便利，但并非适用于所有情况。在某些复杂的配置需求中，XML 配置可能仍然具有其独特的优势，因此，在决定采用何种配置方式时，需要充分考虑项目的实际需求，综合权衡利弊，选择最适合的配置策略。这样才能在确保项目顺利进行的同时，充分发挥出 Spring 框架的强大功能。

5.4 实战案例：SSM 框架整合实现

在讲解完 SSM 框架的整合原理与步骤之后，为了进一步加深读者对该框架组合在实际项目中的理解，本章将通过一个具体的实战案例——用户管理模块的实现来展示 SSM 框架的应用（该章主要讲解该模块的整合和接口编写等内容）。本模块后台采用 SSM 框架进行编写，旨在提供一个高效、稳定且易于维护的用户管理系统。用户管理模块作为整个系统的基础模块之一，主要实现了用户信息的增、删、改、查等功能。

5.4.1 数据库设计

用户登录模块作为系统的基础，自然离不开用户数据的存储与查询，因此，在构建用户登录模块时，首先需要设计并创建一个用户表，用于存储用户的基本信息，如用户名、密码、

邮箱、角色等。通过合理设计用户表的结构和字段，可以确保用户数据的完整性和安全性，为后续的用户登录、身份验证及权限管理等功能提供有力的数据支持。

使用客户端工具创建数据表，如图 5-9 所示。

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	int	32		☑	☐	🔑1	用户id
name	varchar	32		☐	☐		用户名
password	varchar	32		☐	☐		用户密码
email	varchar	32		☐	☐		用户邮箱
hiredate	varchar	32		☐	☐		入职时间
role	varchar	32		☐	☐		用户角色
departuredate	varchar	32		☐	☐		离职时间
status	varchar	1		☐	☐		用户状态（0:正常,1:禁用）

图 5-9　创建数据表

通过客户端工具插入数据，如图 5-10 所示。

id	name	password	email	hiredate	role	departuredate	status
1	admin	123456	admin@qq.com	2024-03-01	ADMIN	(Null)	0
2	张三	123456	zhangsan@qq.com	2024-03-01	USER	(Null)	0
3	李四	123456	lisi@qq.com	2024-03-01	USER	(Null)	0

图 5-10　插入数据

以上创建数据库、数据表及向数据表中插入数据的操作也可以通过 SQL 语句实现，SQL 语句代码如下：

```
//第 5 章/user - system.sql
CREATE DATABASE user - system;
USE user - system;
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(32) NOT NULL AUTO_INCREMENT COMMENT '用户 id',
  `name` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '用户
名',
  `password` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT
'用户密码',
  `email` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '用
户邮箱',
  `hiredate` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT
'入职时间',
  `role` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '用户
角色',
  `departuredate` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL
COMMENT '离职时间',
  `status` varchar(1) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '用
户状态(0:正常,1:禁用)',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 8 CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_
FORMAT = DYNAMIC;
```

```
INSERT INTO `user` VALUES (1, 'admin', '123456', 'admin@qq.com', '2024-03-01', 'ADMIN', NULL,
'0');
INSERT INTO `user` VALUES (2, '张三', '123456', 'zhangsan@qq.com', '2024-03-01', 'USER',
NULL, '0');
INSERT INTO `user` VALUES (3, '李四', '123456', 'lisi@qq.com', '2024-03-01', 'USER', NULL,
'0');

SET FOREIGN_KEY_CHECKS = 1;
```

5.4.2 引入相关依赖

在 IntelliJ IDEA 集成开发环境中，创建一个名为 user-system 的 Maven Web 项目，并在项目的 pom.xml 文件中引入以下依赖，具体代码如下：

```
//第 5 章/user-system/pom.xml
<dependencies>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.2</version>
</dependency>
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
```

```
        <version>5.1.10</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.1</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.16</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.20</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.9.2</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.9.2</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.9.0</version>
    </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.10.0</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.10.0</version>
        </dependency>
    </dependencies>
```

(1) spring-context：Spring 框架的核心容器，提供了 Spring 框架的核心功能，如依赖注入(DI)和面向切面编程(AOP)，并管理应用程序中 bean 的生命周期，即从创建、配置、装配到销毁。此外，它还提供了事件处理、资源加载、国际化等实用功能。作为 Spring 框架的基础，spring-context 使开发者能够轻松地将应用程序组件组装在一起，形成一个功能完整的应用程序，并通过 DI 和 AOP 降低了代码间的耦合，增强了代码的可重用性和可测试性。

(2) spring-tx：Spring 框架中负责事务管理的关键组件，它提供了对事务管理的全面支持，包括声明式事务和编程式事务。同时，它能够整合多种数据源和事务管理器，如 JDBC、JPA、Hibernate 等，以适应不同的应用场景。spring-tx 还提供了灵活的事务属性配置，如事务传播行为、隔离级别和只读属性等，以满足复杂的业务需求，其作用是确保数据的完整性和一致性，特别是在涉及多个数据库操作的场景中，极大地简化了事务管理的代码，使开发者能够专注于业务逻辑的实现，而无须在每个需要事务的地方编写烦琐的事务代码。

(3) spring-jdbc：Spring 框架中专门用于简化 JDBC 操作数据库的依赖项。它提供了对 JDBC 的封装，包括 JdbcTemplate 等类，使数据库操作更加简单高效。此外，它还包含了 Spring 自带的数据源实现，进一步简化了数据源的配置工作。通过 spring-jdbc，开发者能够降低直接使用 JDBC 的复杂性，提高数据库操作的效率，也可以轻松地执行 SQL 查询、更新和批处理等操作，而无须过多地关注底层的 JDBC 细节，从而更专注于业务逻辑的实现。

(4) spring-webmvc：Spring MVC 的核心，提供了构建 Web 应用程序的完整框架，包括前端控制器、视图解析器、处理器映射等组件，并支持注解驱动的控制器开发，简化了控制器代码的编写。同时，它还提供了数据绑定、格式化、校验等实用功能，使开发者能够快速地构建出结构清晰、易于维护的 Web 应用程序。通过注解和配置，开发者可以灵活地定义 URL 映射、请求处理方法等，从而实现 Web 请求的快速响应和处理。

(5) mybatis：一个优秀的持久层框架，它支持定制化 SQL、存储过程及高级映射，不仅避免了绝大多数的 JDBC 代码和手动设置参数及获取结果集的烦琐，而且可以通过简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs 映射成数据库中的记录，从而使开发者能够更专注于 SQL 本身，而不是 JDBC 的烦琐细节，并提供了映射标签以简化数据库操作，允许开发者通过 XML 配置文件或注解灵活地编写 SQL 语句，实现复杂的数据库操作。

(6) pagehelper：MyBatis 的分页插件依赖，它作为一个插件为 MyBatis 提供了分页功能，能够在不修改原有 MyBatis 映射文件和 SQL 语句的情况下实现物理分页，并提供了简单的 API 来控制分页参数，如当前页和每页显示数量。该依赖简化了分页查询的编码过程，避免了手动编写分页 SQL 语句或处理分页逻辑的烦琐，同时提高了查询性能，只返回所需的分页数据，而不是一次性返回所有数据再进行内存分页，因此适用于各种复杂的分页场

景，并支持多种数据库。

(7) mybatis-spring：为 MyBatis 与 Spring 框架提供了整合功能，简化了 MyBatis 的配置和集成过程，允许开发者在 Spring 容器中配置 SqlSessionFactory 和 Mapper 接口，并且提供了事务管理的支持，可以将 MyBatis 整合到 Spring 容器中，使 SqlSessionFactory 和 Mapper 成为 Spring 管理的 Bean，通过 Spring 的依赖注入功能，开发者能够方便地将 Mapper 注入其他 Spring Bean 中，同时确保了数据库操作的原子性，实现了统一的事务管理。

(8) mysql-connector-java：MySQL 数据库的 Java 连接驱动，它不仅允许 Java 应用程序与 MySQL 数据库进行通信和交互，而且提供了必要的 API 和类库，使开发者能够轻松地执行 SQL 查询、更新、删除等操作，实现数据的持久化。

(9) druid：开源的数据库连接池实现，它具备高效的数据库连接管理和监控功能，负责维护和管理应用程序与数据库之间的连接，通过减少应用程序频繁地创建和关闭数据库连接的开销，提高了数据库访问的性能和稳定性，同时提供了丰富的监控和统计功能，帮助开发者更好地了解数据库的使用情况。

(10) javax.servlet：包含 Java Servlet API 的类库，专为 Web 应用程序开发而设计。它提供了开发 Web 应用所需的多种接口和类，如 ServletRequest 和 ServletResponse 等，使开发者能够轻松地创建处理 HTTP 请求的 Servlet，进而实现 Web 页面的动态生成与交互功能，为 Web 应用的构建提供了坚实的基础。

(11) jackson：Jackson 是一个广泛使用的 Java 库，专注于处理 JSON 格式的数据，其中，jackson-core 库提供了处理 JSON 的核心功能，包括解析和生成 JSON 数据；jackson-databind 库则负责实现 Java 对象与 JSON 之间的转换，支持处理复杂的 Java 对象结构，而 jackson-annotations 库则提供了一系列注解，帮助开发者在 Java 类上定义 JSON 序列化和反序列化的规则。这些依赖的引入使开发者能够轻松地在 Java 应用程序中处理 JSON 数据，无论是将 Java 对象转换为 JSON 字符串，还是从 JSON 字符串中解析出 Java 对象都变得十分便捷。这一功能在 Web 开发、API 交互及数据交换等场景中发挥着重要作用。将 javax.servlet-api 的< scope >标签设置为 provided，表明这个依赖在编译时是必要的，但在运行时将由运行环境(如 Servlet 容器)提供，因此在打包应用程序时不会包含这个依赖，这有助于减小应用程序的大小，并避免与运行环境中的库版本冲突。

(12) slf4j-log4j12：SLF4J(Simple Logging Facade for Java)与 Log4j 1.x 版本之间的桥接库，它允许开发者在代码中仅与 SLF4J 的 API 交互，而实际使用的日志框架可在运行时确定，slf4j-log4j12 这个依赖能够使 SLF4J 将日志请求转发给 Log4j 1.x 进行处理，从而在不修改代码的情况下通过配置轻松地切换至不同的日志框架，实现日志系统的灵活性和可替换性。

(13) log4j-api：提供了 Log4j 2.x 的日志 API，定义了日志记录、级别设置、插件配置等核心功能，使开发者能够利用这些 API 编写日志记录代码，并与其他 Log4j 组件(如 log4j-core)无缝交互，从而构建出灵活且高效的日志系统。

(14) log4j-core：Log4j 2.x 的核心实现库，它负责实际处理日志记录请求，并将日志输出到不同的目的地，如控制台、文件或数据库等，同时根据配置决定日志的格式化、过滤和输出方式。开发者通常会在项目中同时包含 log4j-api 和 log4j-core，以充分利用 Log4j 2.x 的完整功能。

在添加上述依赖项时，需要确保 groupId、artifactId 和 version 信息的准确无误，以便 Maven 能够精准地下载并引入这些依赖，保证项目的顺利运行。

5.4.3 编写配置文件和配置类

1. 创建 jdbc.properties 配置文件

首先在项目的 src/main/resources 目录下创建数据源属性文件 jdbc.properties，并在其中配置相应的数据源信息，如数据库 URL、用户名、密码等，代码如下：

```
//第 5 章/user - system/src/main/resources/jdbc.properties
jdbc.driverClassName = com.mysql.cj.jdbc.Driver

jdbc.url = jdbc:mysql://localhost:3306/user - system?useUnicode = true&characterEncoding =
utf - 8&serverTimezone = Asia/Shanghai

jdbc.username = root
jdbc.password = 123456
```

2. 创建 JdbcConfig 配置类

在项目的 src/main/java/com/demo 目录下创建一个 config 包来存放该项目的配置类，并在该包中创建一个名为 JdbcConfig 的配置类，该配置类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/config/JdbcConfig.java
package com.demo.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.PropertySource;

import javax.sql.DataSource;

@PropertySource("classpath:jdbc.properties")
public class JdbcConfig {

    @Value("${jdbc.driverClassName}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String userName;
    @Value("${jdbc.password}")
```

```
    private String password;

    @Bean("dataSource")
    public DataSource getDataSource(){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(userName);
        dataSource.setPassword(password);
        return dataSource;
    }
}
```

这个配置类用于创建并配置 Druid 数据库连接池。它使用@PropertySource 注解从 jdbc.properties 文件中读取数据库连接信息(如驱动类名、URL、用户名和密码),然后在 getDataSource 方法中创建一个 DruidDataSource 对象并设置这些信息。最后,该数据源以 dataSource 的名称注册到 Spring 容器中,供其他组件使用。

3. 创建 MyBatisConfig 配置类

在项目的 src/main/java/com/demo/config 目录下创建一个名为 MyBatisConfig 的配置类,该配置类的代码如下:

```
//第 5 章/user - system/src/main/java/com/demo/config/MyBatisConfig.java
package com.demo.config;

import com.github.pagehelper.PageInterceptor;
import org.apache.ibatis.plugin.Interceptor;
import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.mapper.MapperScannerConfigurer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;

import javax.sql.DataSource;
import java.util.Properties;

public class MyBatisConfig {

    @Bean
    public PageInterceptor getPageInterceptor() {
        PageInterceptor pageInterceptor = new PageInterceptor();
        Properties properties = new Properties();
        properties.setProperty("value", "true");
        pageInterceptor.setProperties(properties);
        return pageInterceptor;
    }

    @Bean
       public SqlSessionFactoryBean getSqlSessionFactoryBean(@Autowired DataSource
dataSource,@Autowired PageInterceptor pageInterceptor){
```

```
        SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);
        Interceptor[] plugins = {pageInterceptor};
        sqlSessionFactoryBean.setPlugins(plugins);
        return sqlSessionFactoryBean;
    }

    @Bean
    public MapperScannerConfigurer getMapperScannerConfigurer(){
        MapperScannerConfigurer mapperScannerConfigurer = new MapperScannerConfigurer();
        mapperScannerConfigurer.setBasePackage("com.demo.mapper");
        return mapperScannerConfigurer;
    }
}
```

(1) PageInterceptor 配置：创建了`PageInterceptor`对象，该对象专门用于实现分页功能，通过它可高效地满足数据库查询结果的分页需求。

(2) SqlSessionFactoryBean 配置：创建了 SqlSessionFactoryBean 对象，用于构建和执行 SQL 语句。在配置过程中，利用 @ Autowired 注解实现了 DataSource 和 PageInterceptor 的自动装配。同时，在配置中设定了 dataSource。最后将配置好的 PageInterceptor 插件添加至 SqlSessionFactoryBean，以支持分页等高级数据库操作特性。

(3) MapperScannerConfigurer 配置：创建了 MapperScannerConfigurer 对象，用于自动化扫描并注册 MyBatis 映射文件。它负责扫描指定包(这里是 com.demo.mapper)下的 MyBatis 映射接口，并将它们自动注册为 Spring 容器中的 Bean。通过此配置，可直接在 Spring 应用中注入并使用这些 Mapper 接口，简化了 MyBatis 的集成过程。

4. 创建 SpringConfig 配置类

在项目的 src/main/java/com/demo/config 目录下创建一个名为 SpringConfig 的配置类，该配置类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/config/SpringConfig.java
package com.demo.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

@Configuration
@Import({MyBatisConfig.class,JdbcConfig.class})
```

```
@ComponentScan("com.demo.service")
@EnableTransactionManagement
public class SpringConfig {

    @Bean("transactionManager")
     public DataSourceTransactionManager getDataSourceTxManager(@Autowired DataSource
dataSource){
        DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();
        transactionManager.setDataSource(dataSource);
        return transactionManager;
    }
}
```

SpringConfig 类是一个核心的 Spring 配置类，负责整合和配置多个关键组件。它导入了 MyBatisConfig 和 JdbcConfig 类的配置，以设置数据库连接和 MyBatis 框架。同时，它还扫描 com.demo.service 包，自动地将服务层组件注册为 Spring Bean，并且通过开启事务管理功能，SpringConfig 允许使用 @Transactional 注解来管理数据库事务。最后，它定义了一个名为 transactionManager 的事务管理器 Bean，并自动地装配了数据源作为该管理器的数据源。通过这些配置，SpringConfig 类确保了 Spring 框架的顺利运行，并提供了事务管理和数据源配置的功能。

5. 创建 SpringMvcConfig 配置类

在项目的 src/main/java/com/demo/config 目录下创建一个名为 SpringMvcConfig 的配置类，该配置类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/config/SpringMvcConfig.java
package com.demo.config;

import org.springframework.context.annotation.*;
import org.springframework.web.servlet.config.annotation.*;

@Configuration
@ComponentScan({"com.demo.controller"})
@EnableWebMvc
public class SpringMvcConfig implements WebMvcConfigurer {
    @Override
        public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.jsp("/user/",".jsp");
    }
}
```

SpringMvcConfig 配置类的主要作用是启用 Spring MVC，并定义了一些基础配置。它扫描了 com. demo. controller 包以便自动地将控制器注册为 Spring Bean，启用了默认 Servlet 处理以支持静态资源服务，并配置了 JSP 视图解析器，使控制器返回的视图名称能够正确地被映射到 JSP 文件。通过这些配置，Spring MVC 应用就能按照这些配置处理请求和返回视图了。

6. 创建 ServletContainersInitConfig 配置类

在项目的 src/main/java/com/demo/config 目录下创建一个 ServletContainersInitConfig 配置类，该配置类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/config
    ServletContainersInitConfig. java
package com. demo. config;
import org. springframework. web. servlet. support.
AbstractAnnotationConfigDispatcherServlet Initializer;

public class ServletContainersInitConfig extends
AbstractAnnotationConfigDispatcherServlet Initializer {

    protected Class <?>[ ] getRootConfigClasses() {
        return new Class[ ]{SpringConfig. class};
    }

    protected Class <?>[ ] getServletConfigClasses() {
        return new Class[ ]{SpringMvcConfig. class};
    }

    protected String[ ] getServletMappings() {
        return new String[ ]{"/"};
    }
}
```

这个配置类的作用是初始化 Spring MVC 的 Web 环境，它指定了如何创建和配置 Spring 容器(通过 SpringConfig)和 Spring MVC 容器(通过 SpringMvcConfig)。该配置类还定义了 DispatcherServlet(Spring MVC 的核心组件)的 URL 映射，即处理应用中的所有请求。当应用启动时，这些配置会自动加载并初始化，使 Spring MVC 能够正常工作。

7. 创建 EncodingFilter 配置类

在项目的 src/main/java/com/demo/config 目录下创建一个名为 EncodingFilter 的配置类，该配置类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/config/EncodingFilter. java
package com. demo. config;

import javax. servlet. * ;
import javax. servlet. annotation. WebFilter;
import java. io. IOException;
```

```
@WebFilter(filterName = "encodingFilter",urlPatterns = "/*")
public class EncodingFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) {}
    @Override
     public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
        servletRequest.setCharacterEncoding("UTF-8");
        servletResponse.setCharacterEncoding("UTF-8");
        filterChain.doFilter(servletRequest,servletResponse);
    }
    @Override
    public void destroy() {}
}
```

EncodingFilter 是一个实现了 Filter 接口的 Java 过滤器，通过@WebFilter 注解被定义并配置为作用于所有 URL 路径。这个过滤器的主要作用是在 Web 请求处理过程中将请求和响应的字符编码设置为 UTF-8，从而确保在 Web 应用中正确地处理文本数据，避免出现乱码问题。当请求到达时，doFilter 方法会被调用，首先将请求的字符编码设置为 UTF-8，然后将响应的字符编码也设置为 UTF-8，最后通过 filterChain.doFilter 方法将请求传递给后续的过滤器或目标资源。这样，整个应用中的请求和响应都能够以统一的 UTF-8 编码进行数据的编码和解码，确保了数据的准确性和一致性。这个过滤器在 Web 应用的初始化阶段配置好后会在整个应用的生命周期内有效，为 Web 应用提供了一致的字符编码处理机制。

5.4.4　用户管理模块实现

1. 创建返回类

在项目的 src/main/java/com/demo 目录下创建一个 entity 包，并在该包中创建一个名为 Result 的类，该类的代码如下：

```
//第 5 章/user-system/src/main/java/com/demo/Result.java
package entity;
import java.io.Serializable;

public class Result<T> implements Serializable{
    private boolean success;
    private String message;
    private T data;

    public Result(boolean success, String message) {
        super();
        this.success = success;
        this.message = message;
    }
```

```
    public Result(boolean success, String message, T data) {
        this.success = success;
        this.message = message;
        this.data = data;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public boolean isSuccess() {
        return success;
    }

    public void setSuccess(boolean success) {
        this.success = success;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
```

该类是一个泛型类，用于封装操作的结果。它包含 3 个字段：success(表示操作是否成功)、message(提供操作结果的详细信息)和 data(携带与操作结果相关的数据，类型为泛型 T)。类提供了两个构造函数来初始化这些字段，并提供了相应的 getter 和 setter 方法来获取和设置这些字段的值。这个类用于后端服务和 API 的响应中，以标准化的方式返回操作的结果和相关信息。

2. 创建分页结果的实体类

在项目的 src/main/java/com/demo/entity 目录下创建一个名为 PageResult 的类，该类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/entity/PageResult.java
package entity;

import java.io.Serializable;
import java.util.List;
```

```
public class PageResult implements Serializable{
    private long total;
    private List rows;

    public PageResult(long total, List rows) {
          super();
          this.total = total;
          this.rows = rows;
    }

    public long getTotal() {
          return total;
    }

    public void setTotal(long total) {
          this.total = total;
    }

    public List getRows() {
          return rows;
    }

    public void setRows(List rows) {
          this.rows = rows;
    }
}
```

PageResult 实体类用于表示分页查询的结果。它包含两个字段：total 表示查询结果的总数，rows 是一个列表，包含查询到的数据集合。类中有一个构造函数和两个字段的 getter 和 setter 方法，用于创建和操作 PageResult 对象。这个类用于后端服务，将分页查询结果封装后返给前端或其他调用者。

3. 创建持久化类

在项目的 src/main/java/com/demo 目录下，创建一个名为 domain 的包。在该包中，需创建一个持久化类 User，用于定义与用户相关的属性，并为这些属性提供相应的 getter 和 setter 方法，代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/domain/User.java
package com.demo.domain;

import java.io.Serializable;

public class User implements Serializable {
    //用户 id
    private Integer id;
    //用户名称
    private String name;
```

```
    //用户密码
    private String password;
    //用户邮箱
    private String email;
    //用户角色
    private String role;
    //入职时间
    private String hiredate;
    //离职时间
    private String departuredate;
    //用户状态(0:正常,1:禁用)
    private String status;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
```

```
        this.role = role;
    }

    public String getHiredate() {
        return hiredate;
    }

    public void setHiredate(String hiredate) {
        this.hiredate = hiredate;
    }

    public String getDeparturedate() {
        return departuredate;
    }

    public void setDeparturedate(String departuredate) {
        this.departuredate = departuredate;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

4. 创建 MyBatis 的映射文件

在项目的 src/main/resources/com/demo/mapper 目录下创建一个名为 UserMapper.xml 的映射文件，该映射文件的代码如下：

```
//第 5 章/user - system/src/main/resources/UserMapper.xml
<?xml version = "1.0" encoding = "UTF - 8"?>
<! DOCTYPE mapper PUBLIC " - //mybatis.org//DTD Mapper 3.0//EN" " http://mybatis.org/dtd/
mybatis - 3 - mapper.dtd">
< mapper namespace = "com.demo.mapper.UserMapper">

    < insert id = "insertUser">
    insert into user(id,name,password,email,role,hiredate,departuredate,status)
          values ( # {id}, # {name}, # {password}, # {email}, # {role}, # {hiredate},
# {departuredate}, # {status})
    </insert >

    < update id = "updateUser" parameterType = "com.demo.domain.User">
        update user
        < trim prefix = "set" suffixOverrides = ",">
            < if test = "name != null" >
```

```
                name = #{name},
            </if>
            <if test="password != null">
                password = #{password},
            </if>
            <if test="email != null">
                email = #{email},
            </if>
            <if test="role != null">
                role = #{role},
            </if>
            <if test="hiredate != null">
                hiredate = #{hiredate},
            </if>
            <if test="departuredate != null">
                departuredate = #{departuredate}
            </if>
            <if test="status != null">
                ustatus = #{status},
            </if>
        </trim>
        where id = #{id}
    </update>
</mapper>
```

该映射文件旨在定义与数据库交互的 SQL 语句。文件内包含两个核心操作：一是插入操作，向 user 表插入数据，涉及多个字段，如 id、name、password 等，并运用占位符“#{}”以接收实际参数；二是更新操作，根据特定条件更新 user 表中的数据，充分利用了 MyBatis 的动态 SQL 特性，根据传入的 User 对象属性值动态地构建 SQL 语句的 SET 子句。此映射文件的设计使 Java 代码能够便捷地调用这些预定义的 SQL 语句，并传递相应参数以执行数据库操作。此外，待完成的方法将基于注解的方式实现。

5. 创建 DAO 层

在项目的 src/main/java/com/demo 目录下创建一个 mapper 包来存放该项目的 DAO 层的类，并在该包中创建一个名为 UserMapper 的类，该类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/dao/UserMapper.java
package com.demo.mapper;

import com.github.pagehelper.Page;
import com.demo.domain.User;
import org.apache.ibatis.annotations.*;

public interface UserMapper{

    //新增
void insertUser(User user);
```

```
    //编辑
    void updateUser(User user);

    //搜索
    @Select({"<script>" +
            "SELECT * FROM user " +
            "where 1 = 1 " +
            "<if test = \"id != null\"> AND id like CONCAT('%',#{id},'%')</if>" +
            "<if test = \"name != null\"> AND name like CONCAT('%', #{name},'%') </if>" +
            "order by status" +
            "</script>"
    })
    @ResultMap("userMap")
    Page<User> searchUsers(User user );

    //根据 id 查询用户
    @Select(" select * from user where id = #{id}")
    @ResultMap("userMap")
    User findById(Integer id);
}
```

该代码通过 MyBatis 的 Mapper 接口定义了针对 user 表的操作，其中包括插入新用户、更新用户信息、根据 id 和 name 模糊搜索用户并支持分页与按 status 排序，以及通过 id 查询单个用户的方法，并且在代码中使用 MyBatis 的动态 SQL 和注解，使 SQL 语句和结果映射的定义更灵活和更简洁。

6. 创建 Service 层

在项目的 src/main/java/com/demo 目录下创建一个 service 包来存放该项目的 Service 层的类，并在该包中创建一个名为 UserService 的类，该类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/service/UserService.java
package com.demo.service;

import com.demo.domain.User;
import entity.PageResult;

public interface UserService{
    //新增
    void insertUser(User user);
    //编辑
    void updateUser(User user);
    //搜索
    PageResult searchUsers(User user, Integer pageNum, Integer pageSize);
    //根据 id 查询用户
    User findById(Integer id);
}
```

完成后在项目的 src/main/java/com/demo/service 目录下创建一个 impl 包来存放

Service 层的实现类，并在该包中创建一个名为 UserServiceImpl 的类，该类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/service/impl /UserServiceImpl.java
package com.demo.service.impl;

import com.github.pagehelper.Page;
import com.github.pagehelper.PageHelper;
import com.demo.domain.User;
import com.demo.mapper.UserMapper;
import com.demo.service.UserService;
import entity.PageResult;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

@Service
public class UserServiceImpl implements UserService {
    @Autowired
private UserMapper userMapper;

    //新增(将状态设置为 0)
    public void insertUser(User user) {
        user.setStatus("0");
        userMapper.insertUser(user);
    }

    //编辑
    public void updateUser(User user) {
        userMapper.updateUser(user);
    }

    //搜索,使用分页插件显示结果
    public PageResult searchUsers(User user, Integer pageNum, Integer pageSize) {
        PageHelper.startPage(pageNum, pageSize);
        Page<User> page = userMapper.searchUsers(user);
        return new PageResult(page.getTotal(),page.getResult());
    }

    //根据 id 查询用户
    public User findById(Integer id) {
        return userMapper.findById(id);
    }
}
```

该代码中 searchUsers 方法使用 PageHelper 插件进行分页，通过 userMapper 执行用户搜索，并返回一个封装了分页结果信息的 PageResult 对象。首先 PageHelper.startPage 初始化分页参数，然后执行搜索并将结果封装在 Page<User>中，最后提取总记录数和当前页的用户列表，创建 PageResult 对象并返回。

7. 创建 Controller 层

在项目的 src/main/java/com/demo 目录下创建一个 controller 包来存放该项目的 Controller 层的类，并在该包中创建一个名为 UserController 的类，该类的代码如下：

```
//第 5 章/user - system/src/main/java/com/demo/controller /UserController.java
package com.demo.controller;

import com.demo.domain.User;
import com.demo.service.UserService;
import entity.PageResult;
import entity.Result;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.servlet.ModelAndView;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

@Controller
@RequestMapping("/user")
public class UserController {
    @Autowired
private UserService userService;

        //新增
    @ResponseBody
    @RequestMapping("/insertUser")
    public Result insertUser(User user) {
        try {
            userService.insertUser(user);
            return new Result(true, "新增成功!");
        } catch (Exception e) {
            e.printStackTrace();
            return new Result(false, "新增失败!");
        }
}

    //编辑
    @ResponseBody
    @RequestMapping("/updateUser")
    public Result updateUser(User user) {
        try {
            userService.updateUser(user);
            return new Result(true, "修改成功!");
        } catch (Exception e) {
            e.printStackTrace();
            return new Result(false, "修改失败!");
```

```
        }
    }

    //搜索
    @RequestMapping("/search")
    public ModelAndView search(User user, Integer pageNum, Integer pageSize)
    {   if (pageNum == null) {
            pageNum = 1;
        }
        if (pageSize == null) {
            pageSize = 10;
        }
        PageResult pageResult = userService.searchUsers(user, pageNum, pageSize);
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("user");
        modelAndView.addObject("pageResult", pageResult);
        modelAndView.addObject("search", user);
        modelAndView.addObject("pageNum", pageNum);
        modelAndView.addObject("gourl", "/user/search");
        return modelAndView;
    }

    //根据 id 查询用户
    @ResponseBody
    @RequestMapping("/findById")
    public User findById(Integer id) {
        return userService.findById(id);
    }
}
```

其中，search()方法用于处理用户搜索请求，它接收搜索条件、页码和每页大小作为参数（如果页码或每页大小未指定，则使用默认值），通过用户服务获取分页搜索结果，并将结果及其他信息封装在 ModelAndView 对象中返回，以便渲染到名为 user 的视图中。

8. 练习

经过上述的深入剖析与详细讲解，读者应当对 SSM 框架的整合有了更加全面且深刻的理解。接下来，为了进一步巩固和提升在 SSM 框架应用方面的能力，读者可独立实现以下两个接口功能，其中一个功能需通过注解方式实现，另一个则需通过配置文件来配置。

（1）实现停用用户功能：此功能需要编写相应的业务逻辑代码，在用户被停用时，确保数据库内对应用户的 status 字段更新为 1（代表禁用状态），以此确保用户无法继续访问系统。

（2）实现用户离职功能：对于此功能，需处理用户离职后的系列操作，包括将用户状态更改为禁用（将 status 字段置为 1），并同时将离职时间记录至数据库中的相应字段，以确保离职用户的信息得到妥善管理。

通过实现这两个功能，能够更深入地理解 SSM 框架在实际项目中的应用场景和实现细节，同时也将提升读者在 SSM 框架应用方面的技术能力和实践操作经验。