后台项目搭建

13.1　创建后台项目

近年来 Spring Boot 成为了 Java web 开发主框架。Spring Boot 框架本身并不提供 Spring 框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于 Spring 框架的应用，并且在开发过程中大量使用“约定优先配置”（convention over configuration）的思想来摆脱 Spring 框架中各种复杂的手动配置，同时衍生出了 Java Config（取代传统 XML 配置文件的 Java 配置类）这种优秀的配置方式。也就是说，Spring Boot 并不是替代 Spring 框架的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具，同时 Spring Boot 还集成了大量常用的第三方库配置（例如 Jackson、JDBC、Redis、Mail 等）。使用 Spring Boot 开发程序时，几乎是开箱即用（out-of-the-box），大部分的 Spring Boot 应用都只需少量的配置，这一特性更能促使开发者专注于业务逻辑的实现。

另外，随着近几年微服务开发的需求和火爆，怎样快速、简便地构建一个准生产环境的 Spring 应用也是摆在开发者面前的一个难题，而 Spring Boot 框架的出现恰好完美的解决了这些问题，同时内部还简化了许多常用的第三方库配置，使得微服务开发更加便利。

Spring Boot 的优点主要有：可以快速构建项目；对主流开发框架的无配置集成；项目可独立运行；无须外部依赖 Servlet 容器；提供运行时的应用监控；提高了开发、部署效率，特别适合构建微服务系统；与云计算的天然集成；Spring Boot 封装了各种经常使用的套件，比如 mybatis、hibernate、redis、mongodb 等。

总体来说 Spring Boot 其实是对 Spring Framework 做了二次封装，以便简化开发，使程序员将更多的精力和时间放到业务上去。规避了繁琐的配置操作，从而减少了遭遇 bug 的数量。而对于运维人员来说，每种服务的维护均可以采用简单的脚本来进行优雅地维护。Spring Boot 极大地降低了开发和运维的复杂度。

因此，本项目应用 IntelliJ IDEA 创建后台项目，以 Springboot 作为框架基础，数据层使用 Mybatis 或者 Mybatis Plus。

1. 安装带有 Springboot 框架的 IntelliJ IDEA 软件。

2. 打开 IntelliJ IDEA，单击菜单【File】→【New】→【Module...】，弹出窗口如图 13-1 所示，选择【Spring Initializr】→输入如图 13.1 所示各类信息（项目名、项目路径、包名、SDK 等）。

其中，“Server URL”默认为“https://start.spring.io/”，但是由于网络问题，创建项目经常出错或者速度很慢，因此将其更换为“http://start.aliyun.com/”；Java 版本选择下拉菜单中的“8”。

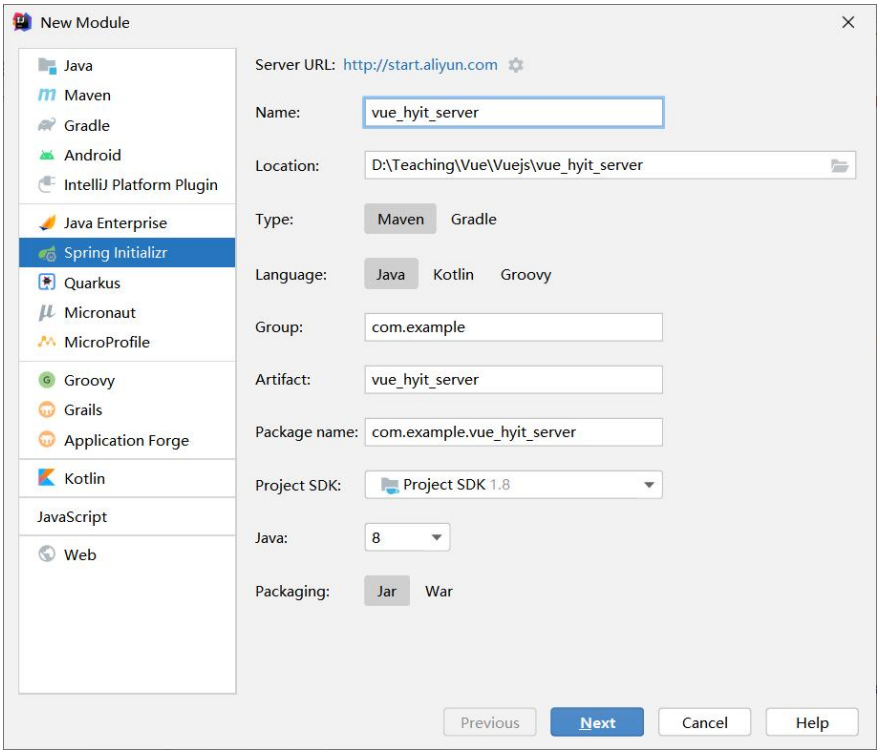

图 13-1 新建 Springboot 框架界面

3. 单击【Next】后进入选择依赖项界面，如图 13-2 所示，分别选中依赖，其他为默认选项即可创建一个新项目。

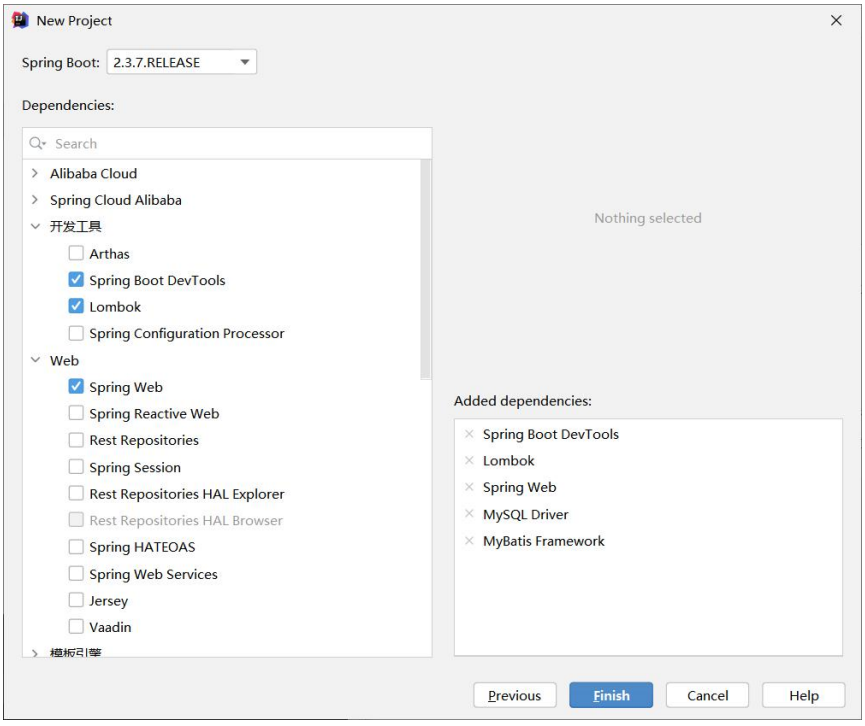

图 13-2 依赖选择界面

4. 项目创建后，若本地没有安装 Maven，项目将提示出错，因此需确认本地已经安装 Maven。Maven 是 Apache 下的一个纯 java 开发的开源项目，它是一个项目管理工具，使用 maven 对 java 项目进行构建、依赖管理。Maven 就是一款帮助程序员构建项目的工具，我们只需要告诉 Maven 需要哪些 Jar 包，它会帮助我们下载所有的 Jar，极大提升开发效率。

（1）安装 Maven 并配置 Maven 环境变量，打开 cmd 窗口并输入 mvn -v ，若提示 Maven 的安装版本，则说明 Maven 安装成功。

（2）配置“D:\Maven\apache-maven-3.8.1\conf”文件夹中的 settings.xml 文件，由于 maven 是从中央仓库下载 jar 包，但是下载速度非常慢，因此修改仓库的地址如下所示。

```
<mirror>
  <id>aliyunmaven</id>
  <mirrorOf>*</mirrorOf>
  <name>镜像仓库</name>
  <url>https://maven.aliyun.com/repository/public</url>
</mirror>
```

（3）修改 maven 项目的默认 jdk 版本，找到<profiles>标签，修改为以下代码。

```
<profile>
    <id>jdk-1.8</id>
    <activation>
        <activeByDefault>true</activeByDefault>
        <jdk>1.8</jdk>
    </activation>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
    </properties>
</profile>
```

（4）在 IntelliJ IDEA 中配置 Maven 的本地仓库，打开菜单【File】→【Settings】，选择【Build，Execution，Deployment】→【Build Tools】→【Maven】，设置 Maven 依赖的本地仓库，如图 13-3 所示。其中，Maven home path 设置为 Maven 的安装路径，注意安装路径中不可以有特殊字符，User settings file 和 Local repository 分别设置为 Maven 安装路径下的配置文件和仓库文件路径。

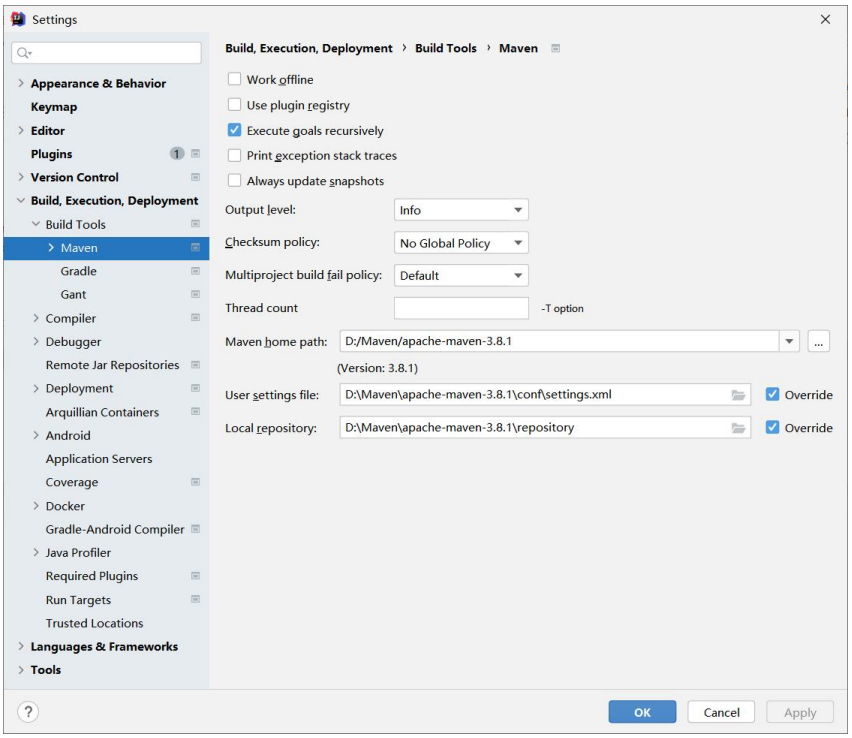

图 13-3 Maven 本地仓库的设置

5. 应用 Navicat Premium 创建 MySQL 数据库文件 elementusers，字符集编码选择 utf-8 编码标准。新建表 t_users 用于存储用户信息，表的字段设计如图 13-4 所示。

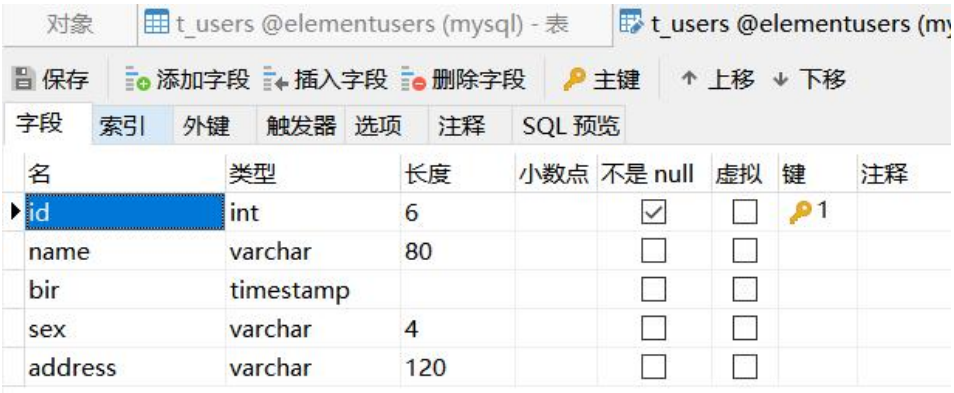

图 13-4 数据表 t_users 设计界面

6. 移除不用文件如文件夹.mvn，文件.gitignore、HELP.md、mvnw、mvnw.cmd。然后打开文件 pom.xml，添加依赖<dependencies></dependencies>部分的内容，代码如下。

```
<dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.19</version>
</dependency>
```

提示：若在 IDEA 中新建项目后发现文件 application.properties 中的以下代码为红色报错，一般是由于数据源的包下载不完整，如图 13-5 所示，在 Maven 面板中重新下载数据源即可。

```
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
```

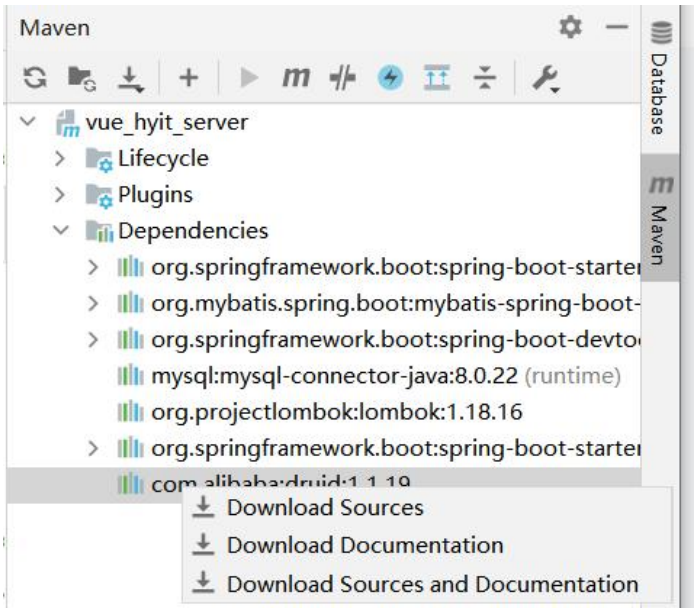

图 13-5 重新下载数据源界面

7. 设置项目的 src→main→resources→application.properties 文件，补充应用服务 WEB 访问端口等信息，代码如下。

```
# 应用名称
spring.application.name=vue_hyit_server
#下面这些内容是为了让 MyBatis 映射
#指定 Mybatis 的 Mapper 文件
mybatis.mapper-locations=classpath:com/example/mapper/*.xml
#指定 Mybatis 的实体目录
mybatis.type-aliases-package=com.example.vue_hyit_server.entity
# 数据库驱动：
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
# 数据源名称
spring.datasource.name=elementusers
# 数据库连接地址
spring.datasource.url=jdbc:mysql://localhost:3306/elementusers?serverTimezone=UTC&use
Unicode=true&characterEncoding=UTF-8
   #serverTimezone=UTC
# 数据库用户名&密码：
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
# 应用服务 WEB 访问端口
server.port=8989
server.servlet.context-path=/
```

8. 在 Navicat Premium 中打开数据库，并右键选中“t-users”表，选择“转储 SQL 文件→仅结构”，导出数据表结构文件“t_users.sql”；并将 Navicat 导出的数据表文件“t_users.sql”复制在项目的 src→main→resources 路径下创建新数据库文件夹“com/example/sql”路径下。

9. 在路径 src→main→java→com→example→vue_hyit_server 下新建类“entity.User”，并在其中添加类的设计，代码如下：

```
package com.example.vue_hyit_server.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;
import lombok.experimental.Accessors;
import java.util.Date;
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
@Accessors(chain = true)

public class User {
    private String id;
    private String name;
    @JsonFormat(pattern = "yyyy-MM-dd")      //用于设计日期格式
    private Date bir;
    private String sex;
    private String address;
}
```

10. 设计后端接口类，在路径 src→main→java→com→example→vue_hyit_server 下新建类“dao.UserDAO”，注意其中类型选择“interface”，并在其中接口设计，代码如下：

```
package com.example.vue_hyit_server.dao;
import com.example.vue_hyit_server.Entity.User;
import java.util.List;
@Mapper
public interface UserDAO {
    //查询所有用户信息
    List<User> findAll();
}
```

11. 若新建文件类型中没有 mapper 类型文件，则在菜单 File →Settings →File and Code Templates 中点击“+”，如图 13-6 所示，分别设置 Name 为“mapper”，Extension 为“xml”，并添加如下模版内容即可。

```
<?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="">

</mapper>
```

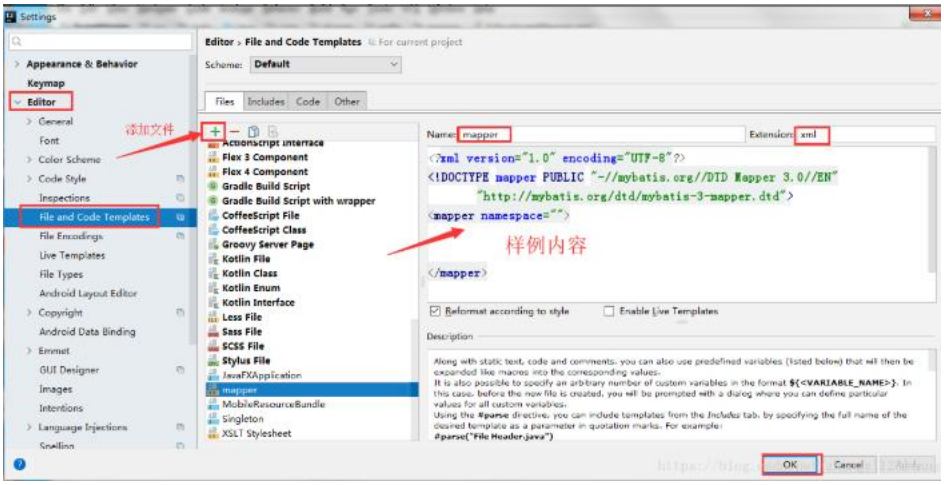

图 13-6 mapper 文件类型设置

12. 在路径 src → main → resources → com → example 下新建 mapper 文件“mapper.UserDAOMapper.xml”，代码如下：

```
<?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.example.vue_hyit_server.dao.UserDAO">
    <!--查询所有的方法-->
    <select id="findAll" resultType="User">
        select id,name,bir,sex,address
        from t_users
    </select>
</mapper>
```

13. 在路径 src → main → java → com → example → vue_hyit_server 下新建接口类“service.UserService”，注意其中类型选择“interface”，并在其中接口设计，代码如下：

```
package com.example.vue_hyit_server.service;
import com.example.vue_hyit_server.entity.User;
import java.util.List;

public interface UserService {
    //查询所有方法
    List<User> findAll();
}
```

14. 在路径 src → main → java → com → example → vue_hyit_server → service 下新建类“UserServiceImpl”，设计代码如下：

```
package com.example.vue_hyit_server.service;
import com.example.vue_hyit_server.dao.UserDAO;
import com.example.vue_hyit_server.entity.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;
```

```
@Service
@Transactional
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDAO userDAO;
    @Override
    public List<User> findAll() {
        return userDAO.findAll();
    }
}
```

15. 在路径 src → test → java → com → example → vue_hyit_server 下新建测试类“TestUserService”，设计代码如下：

```
package com.example.vue_hyit_server;
import com.example.vue_hyit_server.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class TestUserService {
    @Autowired
    private UserService userService;
    @Test
    public void testFindAll(){
        userService.findAll().forEach(user -> System.out.println("user = " + user));
    }
}
```

运行“testFindAll”，测试程序是否正常运行，是否可以正常取出数据表中所有用户。

16. 设计控制器，在路径 src→main→java→com→example→vue_hyit_server 下新建类“controller.UserController”，由于项目前台访问端口是 8080，项目后台端口设置的是 8989，因此需要解决跨域访问问题，设计代码如下：

```
package com.example.vue_hyit_server.controller;
import com.example.vue_hyit_server.entity.User;
import com.example.vue_hyit_server.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;
@RestController
@CrossOrigin   //解决跨域问题
@RequestMapping("user")
public class UserController {
    @Autowired
```

```
        private UserService userService;
        //查询所有用户信息
        @GetMapping("findAll")
        public List<User> findAll(){
            return userService.findAll();
        }
    }
```

17. 运行后台项目入口类文件 src→main→java→VueHyitServerApplication，项目运行成功后，在浏览器中输入访问地址“http://localhost:8989/user/findAll”，如图 13-7 所示，取出了数据表 t-users 中的两条数据。

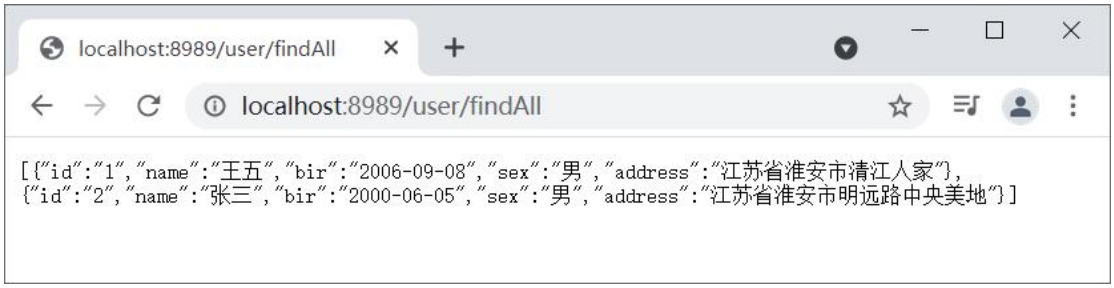

图 13-7 浏览器显示数据表中数据

13.2 项目前端访问后端

axios 是一个基于 promise 的 HTTP 库，简单的说是可以发送 get、post 请求，可以用在浏览器和 node.js 中，在进行项目前后端对接时，使用 axios 工具可以提高项目的开发效率。在 Vue 中安装 axios（http://www.axios-js.com/）。

axios 的优势主要有以下几点：从浏览器中创建 XMLHttpRequests；从 node.js 创建 http 请求；支持 Promise API；拦截请求和响应；转换请求数据和响应数据；取消请求；自动转换 JSON 数据；客户端支持防御 XSRF。

1. axios 的安装命令如下：

```
cnpm install axios --save
```

2. 在 Vue 中安装其他插件的时候，可以直接在 main.js 中引入并 Vue.use()，但是 axios 并不可以，只能在每个需要发送请求的组件中即时引入。为了解决这个问题，有两种开发思路，一种方法是在引入 axios 之后，修改原型链；另外一种方法是结合 Vuex，封装一个 aciton。这里介绍修改原型链的方式。首先在 main.js 中全局引入 axios，代码如下：

```
import axios from 'axios'
```

这时候如果在其它的组件中，是无法使用 axios 命令的，所以将 axios 改写为 Vue 的原型属性，代码如下：

```
Vue.prototype.$http= axios
```

在 main.js 中添加了上述两行代码之后，就可以直接在组件的 methods 中使用$http 命令发起网络请求，例如：

```
 methods: {
     show() {
        this.$http({
            method: 'get',
            url: '/user',
            data: {
```

```
                name: 'virus'
            }
        })
    }
```

3. 在 Vue 的前端用户管理组件中添加项目后端网络请求，请求地址即为如图 5.7 所示的测试地址“http://localhost:8989/user/findAll”，代码如下，Vue 前端用户管理运行效果如图 13-8 所示，同时可以实现搜索功能，这样可以实现项目前端访问后端数据。

```
<script>
export default {
    name: "List",
    data() {
        return {
            tableData: [],
            search:'',
        }
    },
    methods: {
        handleEdit(index, row) {
            console.log(index, row);
        },
        handleDelete(index, row) {
            console.log(index, row);
        }
    },
    created() {
        this.$http.get("http://localhost:8989/user/findAll").then(res=>{
            this.tableData=res.data;
        })
    }
}
</script>
```

图 13-8 浏览器显示数据表中数据

13.3 项目前后端交互应用

本节在项目的用户管理中新增用户的前后端功能。

13.3.1 新增用户信息的前端设计

1. 设计如图 5.7 所示中单击“添加”按钮后的弹出式界面的效果，这里使用 Element UI 组件中的“内置过渡动画”效果，代码如下所示：

```
<el-button @click="show=!show" type="success" size="mini" style="margin: 10px 0px;">添加</el-button>
<transition name="el-zoom-in-center">
        <div v-show="show" class="transition-box">
        </div>
</transition>
<style scoped>
.transition-box {
   margin-bottom: 10px;
   width: 100%;
   height: 600px;
   border-radius: 4px;
   padding: 40px 20px;
   box-sizing: border-box;
   margin-right: 20px;
}
</style>
```

2. 在上述弹出式过渡动画效果之间新增用户表单，这里使用 Element UI 组件中的 Form 典型表单效果，代码如下所示。

```
<div v-show="show" class="transition-box">
          <el-form ref="form" :model="form" label-suffix="：" label-width="85px">
            <el-form-item label="姓名">
              <el-input v-model="form.name"></el-input>
            </el-form-item>
            <el-form-item label="生日">
              <el-date-picker value-format="yyyy-MM-dd" type="date" placeholder="选择日期" v-model="form.bir" style="width: 100%;"></el-date-picker>
            </el-form-item>
            <el-form-item label="性别">
              <el-radio-group v-model="form.sex">
                <el-radio label="男"></el-radio>
                <el-radio label="女"></el-radio>
              </el-radio-group>
            </el-form-item>
            <el-form-item label="详细地址">
              <el-input type="textarea" v-model="form.address"></el-input>
            </el-form-item>
            <el-form-item>
              <el-button type="primary" @click="onSubmit">添加用户</el-button>
              <el-button>取消</el-button>
```

```
                </el-form-item>
            </el-form>
        </div>

data() {
    return {
        tableData: [],
        search:'',
        show:true,
        form: {
            name: '',
            bir: '',
            sex: '男',
            address: ''
        },
    }
},
```

这样，新增用户的前端设计如图 13-9 所示。

图 13-9 新增用户前端设计

13.3.2 新增用户信息的后端设计

1. 在项目后台中添加新增用户保存的方法：设计数据访问层，在接口文件 UserDAO.java 中添加保存用户信息的方法，代码如下：

```
public interface UserDAO {
    //查询所有用户信息
    List<User> findAll();
```

```
    //保存用户信息
    void save(User user);
}
```

2. 在 UserDAOMapper.xml 文件中新增添加用户信息的实现方法，其中，parameterType 表示保存的对象是 User，useGeneratedKeys 设置为 true 表示插入数据时 mybatis 获取从数据库生成的主键，主键为 id，代码如下：

```
<!--保存用户信息-->
    <insert id="save" parameterType="User" useGeneratedKeys="true" keyProperty="id">
        insert into t_users values (#{id},#{name},#{bir},#{sex},#{address})
    </insert>
```

3. 设计业务层，在 UserService.java 中添加保存用户信息的方法，代码如下：

```
public interface UserService {
    //查询所有方法
    List<User> findAll();
    //保存用户信息
    void save(User user);
}
```

4. 设计新增用户方法的业务实现，在 UserServiceImpl.java 文件中添加以下代码，其中，Spring 的事务传播机制中 Propagation.SUPPORTS 级别的意义是，如果当前环境有事务，就加入到当前事务；如果没有事务，就以非事务的方式执行。

```
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDAO userDAO;
    @Override
    @Transactional(propagation = Propagation.SUPPORTS)
    public List<User> findAll() {
        return userDAO.findAll();
    }
    @Override
    public void save(User user){
        userDAO.save(user);
    }
}
```

5. 测试前述新增用户功能是否正确实现，在 TestUserService.java 中添加测试代码，代码如下，运行 testSave 方法测试，若程序运行无误，则在数据表 t_users 中可以新增一条测试数据，即业务方法测试通过。

```
@SpringBootTest
public class TestUserService {
    @Autowired
    private UserService userService;
    @Test
    public void testSave(){
        User user=new User();
        user.setName("测试姓名 1");
```

```
            user.setBir(new Date());
            user.setSex("男");
            user.setAddress("测试地址 1");
            userService.save(user);
        }
        @Test
        public void testFindAll(){
            userService.findAll().forEach(user -> System.out.println("user = " + user));
        }
    }
```

6. 设计表示层 Controller，在 UserController.java 文件中新增 save 方法，代码如下：

```
    public class UserController {
        @Autowired
        private UserService userService;
        //查询所有用户信息
        @GetMapping("findAll")
        public List<User> findAll(){
            return userService.findAll();
        }
        //保存所有用户信息
        @PostMapping("save")
        public void save(@RequestBody User user){
            userService.save(user);
        }
    }
```

7. 在保存数据的同时，需要设计发送数据是否正确添加的状态信息给项目前端，在路径 src→main→java→com.example.vue_hyit_server 下新建类文件“vo.Result.java”，代码如下：

```
    package com.example.vue_hyit_server.vo;
    import lombok.Data;
    @Data
    public class Result {
        private Boolean status=true;
        private String msg;
    }
```

8. 为了将状态信息发送给项目前端，继续完善 UserController.java 文件中的 save 方法，代码如下：

```
    //保存用户信息
        @PostMapping("save")
        public Result save(@RequestBody User user){
            Result result=new Result();
            try {
                userService.save(user);
                result.setMsg("用户信息保存成功！");
            }catch (Exception e){
```

```
                result.setStatus(false);
                result.setMsg("系统错误：保存用户信息失败！");
            }
            return result;
        }
```

9. 至此，已完成项目后端关于添加用户接口的设计，运行后端项目VueHyitServerApplication 后，再设计项目前端 Vue 对于添加用户接口的调用。

10. 在 Vue 前端如图 5.9 所示，为单击“添加用户”按钮设计 submit 事件，即在 submit 事件中向项目后端地址“http://localhost:8989/user/save”发送一个异步 ajax 请求，代码如下，其中，使用 Element UI 中的“Message 消息提示”组件设计信息添加成功后的提示信息。运行项目前端并测试添加一个用户信息数据。

```
<script>
export default {
  name: "List",
  data() {
    return {
      tableData: [],
      search:'',
      show:true,
      form: {
        name: '',
        bir: '',
        sex: '男',
        address: ''
      },
    }
  },
  methods: {
    handleEdit(index, row) {
      console.log(index, row);
    },
    handleDelete(index, row) {
      console.log(index, row);
    },
    onSubmit() {
      //发送一个 ajax 请求
      this.$http.post("http://localhost:8989/user/save",this.form).then(res=>{
        if(res.data.status){
          this.$message({
            message: '恭喜你，'+res.data.msg,
            type: 'success'
          });
        }
        else{
```

```
                this.$message.error(res.data.msg);
            }
          })
        }
      },
      created() {
        this.$http.get("http://localhost:8989/user/findAll").then(res=>{
          this.tableData=res.data;
        })
      }
    }
    </script>
```

11. 上述已实现用户信息的成功添加，但信息添加成功后，我们希望信息成功添加的同时重新加载最新的用户表中的所有信息，并清空表单信息，继续完善上述代码设计。

应用 this.form={};清空表单信息；应用 this.show=false;隐藏表单。

添加信息后需要重新渲染加载用户信息表信息，因此将发送异步请求的部分代码单独封装为一个方法 findAllTableData()，改进的代码如下：

```
 methods: {
      handleEdit(index, row) {
        console.log(index, row);
      },
      handleDelete(index, row) {
        console.log(index, row);
      },
      onSubmit() {
        //发送一个 ajax 请求
        this.$http.post("http://localhost:8989/user/save",this.form).then(res=>{
          if(res.data.status){
            this.$message({
              message: '恭喜你，'+res.data.msg,
              type: 'success'
            });
            //清空表单信息，同时设置性别默认值
            this.form={sex: '男'};
            //隐藏表单
            this.show=false;
            //信息添加成功后，调用刷新数据的方法
            this.findAllTableData();
          }
          else{
            this.$message.error(res.data.msg);
          }
        })
      },
```

```
        findAllTableData(){
          this.$http.get("http://localhost:8989/user/findAll").then(res=>{
            this.tableData=res.data;
          })
        }
      },
      created() {
        this.findAllTableData();
      }
```

13.3.3 删除用户信息的设计

1. 在项目后台中设计删除用户的方法：设计数据访问层，在接口文件 UserDAO.java 中添加删除用户信息的方法，代码如下：

```
public interface UserDAO {
    //查询所有用户信息
    List<User> findAll();
    //保存用户信息
    void save(User user);
    //根据 id 删除一个用户
    void delete(String id);
}
```

2. 在 UserDAOMapper.xml 文件中设计删除用户信息的实现方法，主键为 id，代码如下：

```
<!--根据 id 删除用户信息-->
    <delete id="delete" parameterType="String">
        delete from t_users where id=#{id}
    </delete>
```

3. 设计业务层，在 UserService.java 中添加删除用户信息的方法，代码如下：

```
public interface UserService {
    //查询所有方法
    List<User> findAll();
    //保存用户信息
    void save(User user);
    //根据 id 删除用户信息
    void delete(String id);
}
```

4. 设计删除用户方法的业务实现，在 UserServiceImpl.java 文件中添加以下代码：

```
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDAO userDAO;
    @Override
    @Transactional(propagation = Propagation.SUPPORTS)
    public List<User> findAll() {
        return userDAO.findAll();
    }
```

```
    @Override
    public void save(User user){
        userDAO.save(user);
    }
    @Override
    public void delete(String id){
        userDAO.delete(id);
    }
}
```

5. 测试前述删除用户功能是否正确实现，在 TestUserService.java 中添加测试代码，代码如下，运行 testDelete 方法测试，若程序运行无误，则在数据表 t_users 中可以删除指定 id 的一条数据，即业务方法测试通过。

```
@SpringBootTest
public class TestUserService {
    @Autowired
    private UserService userService;
    @Test
    public void testFindAll(){
        userService.findAll().forEach(user -> System.out.println("user = " + user));
    }
    @Test
    public void testSave(){
        User user=new User();
        user.setName("测试姓名 1");
        user.setBir(new Date());
        user.setSex("男");
        user.setAddress("测试地址 1");
        userService.save(user);
    }
    @Test
    public void testDelete(){
        userService.delete("7");
    }
}
```

6. 设计表示层 Controller，在 UserController.java 文件中新增 delete 方法，代码如下：

```
public class UserController {
    @Autowired
    private UserService userService;
    //删除用户
    @GetMapping("delete")
    public Result delete(String id){
        Result result=new Result();
        try {
            userService.delete(id);
```

```
                result.setMsg("删除用户信息成功！");
            }catch (Exception e){
                e.printStackTrace();
                result.setStatus(false);
                result.setMsg("删除用户信息失败！");
            }
            return result;
        }
    }
```

7. 至此，已完成项目后端关于删除用户接口的设计，运行后端项目 VueHyitServerApplication 后，再设计项目前端 Vue 对于删除用户接口的调用。

在 Vue 前端如图 5.9 所示，为表单的“删除”按钮设计删除方法，即在向项目后端地址“http://localhost:8989/user/delete”发送一个异步 ajax 请求，代码如下。

```
methods: {
    handleEdit(index, row) {
      console.log(index, row);
    },
    handleDelete(index, row) {
      console.log(index, row);
      //发送 axios 请求处理删除数据
      this.$htt.get("http://localhost:8989/user/delete?id="+row.id).then(res=>{
        if(res.data.status){
          this.$message({
            message: res.data.msg,
            type: 'success'
          });
          this.findAllTableData();//刷新数据
        }else{
          this.$message.error(res.data.msg);
        }
      });
    },
```

其中，使用 Element UI 中的“Popconfirm”组件设计信息是否确定删除的确认弹出框，以防止用户的误删除操作，修改 Vue 前端表单的删除按钮的设计，代码如下。运行项目前端并测试删除一个用户信息数据。

```
<template slot-scope="scope">
            <el-button
                size="mini"
                @click="handleEdit(scope.$index, scope.row)">编辑</el-button>
            <el-popconfirm
                confirm-button-text='好的'
                cancel-button-text='不用了'
                icon="el-icon-info"
                icon-color="red"
```

```
                    title="确定删除吗？"
                    @confirm="handleDelete(scope.$index, scope.row)"
                >
                  <el-button size="mini"
                             type="danger"
                             slot="reference">删除</el-button>
                </el-popconfirm>
              </template>
```

13.3.4 编辑用户信息的设计

1. 在 Vue 前端如图 5.8 所示，为表单的“编辑”按钮设计编辑方法。单击“编辑”按钮后使之前设计的表单 Form 信息显示出来，同时将编辑的表数据显示在表单中，实现双向绑定数据，代码如下。

```
methods: {
    handleEdit(index, row) {
      console.log(index, row);
      this.show=true;//展示编辑表单
      this.form=row;//回显编辑信息
    },
```

添加用户信息与编辑用户信息同使用一个表单，若没有回传表格行 id 则是新增用户信息，若有回传表格行 id 信息则是编辑用户信息，因此设计一个新方法“saveUserInfo”，代码如下。

```
<el-button @click="saveUserInfo" type="success" size="mini" style="margin: 10px 0px;">添加</el-button>
```

```
methods: {
    saveUserInfo(){//点击添加时清空信息
      this.show=true;
      this.form={sex:'男'};
    },
```

2. 为表单的“编辑”按钮设计编辑方法，将图 5.9 中“添加用户”按钮改为“确认”按钮。即若点击“编辑”按钮则点击“确认”按钮后执行编辑用户操作，若用户点击“添加”按钮则点击“确认”按钮后执行新增用户操作，二者区别为是否回传表格行 id。前端的异步请求都是地址“http://localhost:8989/user/save”，为了区分不同的操作，这里进行完善和修改，即 save 请求中既要完成保存操作又要完成编辑更新操作。

3. 修改项目后端 UserController.java 文件中 save 方法，改为 saveOrUpdate 方法，代码如下：

```
public class UserController {
    @Autowired
    private UserService userService;
    //查询所有用户信息
    @GetMapping("findAll")
    public List<User> findAll(){
        return userService.findAll();
```

```
    }
    //保存用户
    @PostMapping("saveOrUpdate")
    public Result saveOrUpdate(@RequestBody User user){
        Result result=new Result();
        try {
            if(StringUtils.isEmpty(user.getId())){
                userService.save(user);
                result.setMsg("用户信息保存成功！");
            }else {
                userService.update(user);
                result.setMsg("用户信息编辑成功！");
            }

        }catch (Exception e){
            result.setStatus(false);
            result.setMsg("系统错误：保存用户信息失败！");
        }
        return result;
    }
}
```

4. 设计业务层，在 UserService.java 中添加更新用户信息的方法，代码如下：

```
public interface UserService {
    //查询所有方法
    List<User> findAll();
    //保存用户信息
    void save(User user);
    //根据 id 删除用户信息
    void delete(String id);
    //更新用户信息
    void update(User user);
}
```

5. 设计更新用户方法的业务实现，在 UserServiceImpl.java 文件中添加以下代码：

```
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDAO userDAO;
    @Override
    @Transactional(propagation = Propagation.SUPPORTS)
    public List<User> findAll() {
        return userDAO.findAll();
    }
    @Override
    public void save(User user){
        userDAO.save(user);
```

```
    }
    @Override
    public void update(User user){
        userDAO.update(user);
    }
    @Override
    public void delete(String id){
        userDAO.delete(id);
    }
}
```

6. 设计数据访问层，在接口文件 UserDAO.java 中添加更新用户信息的方法，代码如下：

```
public interface UserDAO {
    //查询所有用户信息
    List<User> findAll();
    //保存用户信息
    void save(User user);
    //根据 id 删除一个用户
    void delete(String id);
    //修改用户信息
    void update(User user);
}
```

7. 在 UserDAOMapper.xml 文件中设计更新用户信息的实现方法，主键为 id，代码如下：

```
<!--根据 id 修改用户信息-->
    <update id="update" parameterType="User">
        update t_users set name=#{name},bir=#{bir},sex=#{sex},address=#{address}
        where id=#{id}
    </update>
```

8. 在 Vue 前端“确认”按钮触发的 Submit 方法中的异步请求地址也进行修改为更新后的方法，最后测试添加或更新用户信息，代码如下：

```
onSubmit() {
      //发送一个 ajax 请求
      this.$http.post("http://localhost:8989/user/saveOrUpdate",this.form).then(res=>{
        if(res.data.status){
          this.$message({
            message: '恭喜你，'+res.data.msg,
            type: 'success'
          });
```

13.3.5 表单输入规则的设计

本项目使用 Element UI 中的表单验证设计，Vue 前端用户管理部分有修改的部分设计代码如下，用户管理部分表单验证在浏览器中的运行效果如图 13-10 所示。

```
<transition name="el-zoom-in-center">
      <div v-show="show" class="transition-box">
        <el-form          :hide-required-asterisk="false"          :rules="rules"
```

```
ref="userForm" :model="form" label-suffix="：" label-width="85px">
                    <el-form-item label="姓名" prop="name">
                      <el-input v-model="form.name"></el-input>
                    </el-form-item>
                    <el-form-item label="生日" prop="bir">
                      <el-date-picker type="date" placeholder="选择日期" v-model="form.bir"
style="width: 100%;"></el-date-picker>
                    </el-form-item>
                    <el-form-item label="性别">
                      <el-radio-group v-model="form.sex">
                        <el-radio label="男"></el-radio>
                        <el-radio label="女"></el-radio>
                      </el-radio-group>
                    </el-form-item>
                    <el-form-item label="详细地址" prop="address">
                      <el-input type="textarea" v-model="form.address"></el-input>
                    </el-form-item>
                    <el-form-item>
                      <el-button type="primary" @click="onSubmit('userForm')">确认</el-button>
                      <el-button @click="saveUserInfo">重置</el-button>
                    </el-form-item>
                  </el-form>
                </div>
              </transition>
```

```
    rules: {
                  name: [
                    { required: true, message: '请输入用户姓名', trigger: 'blur' },
                  ],
                  bir: [
                    { required: true, message: '请输入用户生日', trigger: 'blur' },
                  ],
                  address: [
                    { required: true, message: '请输入用户地址', trigger: 'blur' },
                  ],
        }
```

```
    onSubmit(userForm) {
              this.$refs[userForm].validate((valid) => {
                if (valid) {
                  //发送一个 ajax 请求
                  this.$http.post("http://localhost:8989/user/saveOrUpdate",this.form).then(res=>{
                    if(res.data.status){
                      this.$message({
```

```
                    message: '恭喜你，'+res.data.msg,
                    type: 'success'
                });
                //清空表单信息
                this.form={sex: '男'};
                //隐藏表单
                this.show=false;
                //信息添加成功后，调用刷新数据的方法
                this.findAllTableData();
            }
            else{
                this.$message.error(res.data.msg);
            }
        });
    } else {
        this.$message.error("输入的数据不合法！");
        return false;
    }
  });
},
```

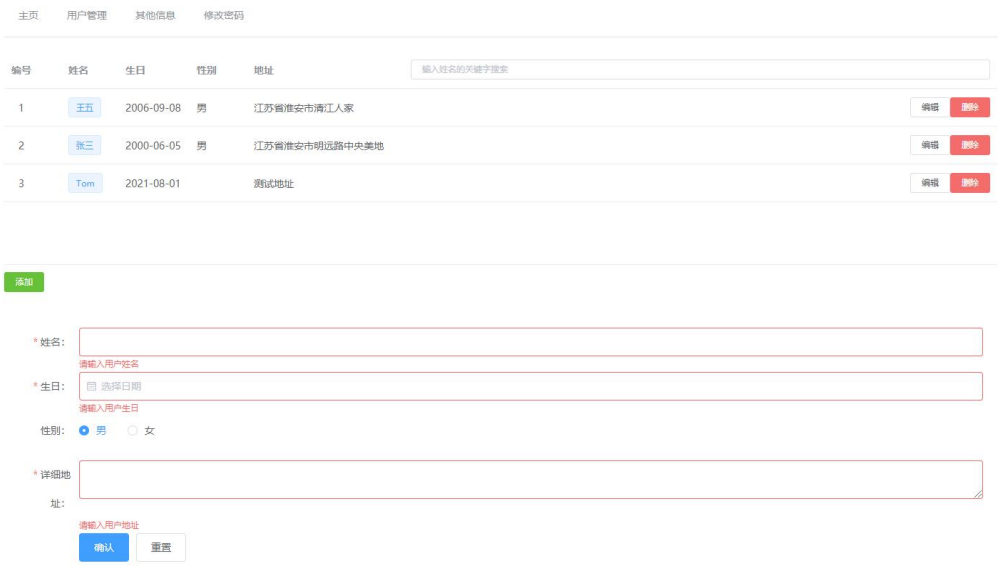

图 13-10 用户管理表单验证运行效果

13.3.6 表单分页组件的设计

1. 本项目使用 Element UI 中的 Pagination 分页组件设计表单的分页，同时使用 Layout 布局设计分页组件的位置，Vue 前端用户信息的表格展示部分代码设计如下：

```
...
</el-table>
    <el-row>
      <el-col :span="12" :offset="12">
```

```
            <el-pagination style="margin: 10px 0px;"
                           background
                           prev-text="上一页"
                           next-text="下一页"
                           layout="prev, pager, next, jumper, total, sizes"
                           @current-change="findPage"
                           @size-change="findSize"
                           :total="50">
            </el-pagination>
          </el-col>
        </el-row>
```

```
    methods: {
        findSize(size){
          console.log(size);
        },
        findPage(){//用于处理分页方法
          console.log(page);
        },
```

2. 在项目后端增加分页的设计，设计数据访问层，在接口文件 UserDAO.java 中添加分页查询和查询总条数的方法，代码如下：

```
public interface UserDAO {
        //分页查询
        List<User> findByPage(@Param("start") Integer start,@Param("rows") Integer rows);
        //查询总条数
        Long findTotals();
}
```

3. 在 UserDAOMapper.xml 文件中新增分页的实现方法，代码如下：

```
<!--分页查询-->
        <select id="findByPage" resultType="User">
            select id,name,bir,sex,address
            from t_users limit #{start},#{rows}
        </select>
        <!--查询总条数-->
        <select id="findTotals" resultType="Long">
            select count(id) from t_users
        </select>
```

4. 设计业务层，在 UserService.java 中添加分页的方法，代码如下：

```
public interface UserService {
    //分页查询
        List<User> findByPage(Integer pageNow, Integer rows);
        //查询总条数
        Long findTotals();
```

```
}
```

5. 设计分页方法的业务实现，在 UserServiceImpl.java 文件中添加以下代码。

```
@Override
public List<User> findByPage(Integer pageNow, Integer rows) {
    int start=(pageNow-1)*rows;
    return userDAO.findByPage(start,rows);
}
@Override
public Long findTotals() {
    return userDAO.findTotals();
}
```

6. 为了将状态信息发送给项目前端，继续完善 UserController.java 文件中的分页方法，代码如下：

```
//分页查询方法
@GetMapping("findByPage")
public Map<String,Object> findByPage(Integer pageNow, Integer pageSize){
    Map<String,Object> result=new HashMap<>();
    pageNow=pageNow==null?1:pageNow;
    pageSize=pageSize==null?4:pageSize;
    List<User> users=userService.findByPage(pageNow,pageSize);
    Long totals=userService.findTotals();
    result.put("users",users);
    result.put("total",totals);
    return result;
}
```

7. 完善 Vue 前端用户信息的表格分页展示部分，代码设计如下，用户信息分页显示效果如图 13-11 所示。

```
<el-row>
    <el-col :span="12" :offset="12">
      <el-pagination style="margin: 10px 0px;"
                     background
                     prev-text="上一页"
                     next-text="下一页"
                     layout="prev, pager, next, jumper, total, sizes"
                     :page-size="size"
                     :current-page="pageNow"
                     :page-sizes="[2,4,6,8,10]"
                     @current-change="findPage"
                     @size-change="findSize"
                     :total="total">
      </el-pagination>
    </el-col>
  </el-row>
```

```
data() {
        return {
            tableData: [],
            search:'',
            show:false,
            form: {
                name: '',
                bir: '',
                sex: '男',
                address: ''
            },
            total:0,
            size:4,
            pageNow:1,
```

```
methods: {
        findSize(size){//用于处理每页显示的记录发生变化的方法
            console.log(size);
            this.size=size;
            this.findAllTableData(this.page,size);
        },
        findPage(page){//用于处理分页方法
            this.page=page;
            this.findAllTableData(page,this.size);
        },
```

```
findAllTableData(page,size){
            page=page?page:this.pageNow,
                size=size?size:this.size;
this.$http.get("http://localhost:8989/user/findByPage?pageNow="+page+"&pageSize"+size).then(
res=>{
                this.tableData=res.data.users;
                this.total=res.data.total;
            })
        }
```

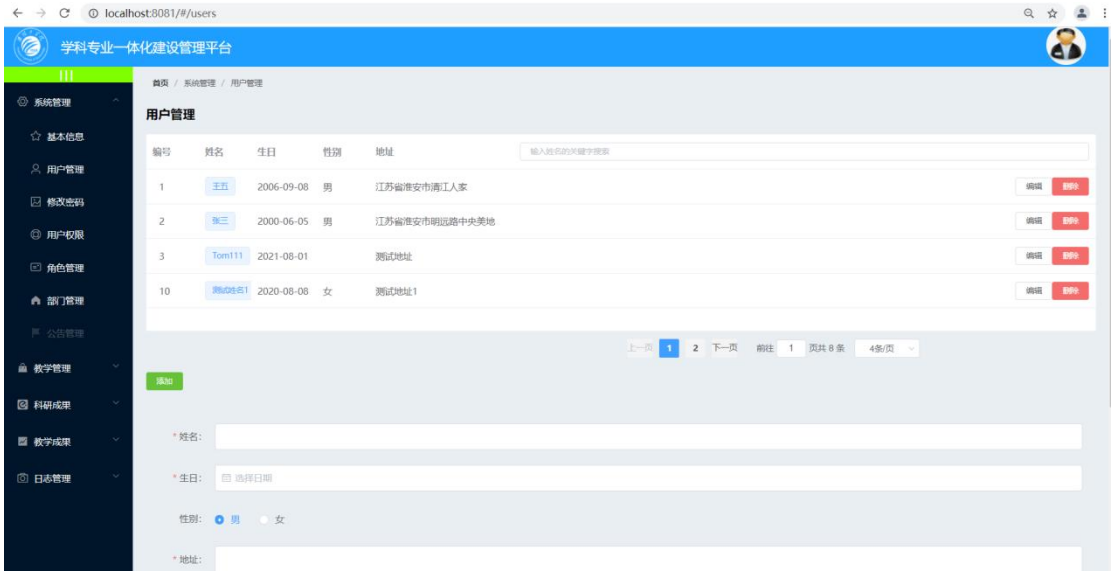

图 13-11 用户信息分页显示效果

13.3.7 项目部署

项目前端部署可以使用 npm run build 语句进行前端项目的部署，部署成功后在项目路径中生成静态部署文件夹 dist，直接将 dist 文件夹复制到项目后端的 src→main→resources→static 路径下即可。重新运行后端项目，在浏览器中输入访问地址 http://localhost:8989/即可正常运行前后端分离的项目。

1. 启动项目后端工程步骤：

（1）打开项目后端文件夹根路径，在地址栏中输入“cmd”打开命令窗口，或者应用 cd 命令进入项目后端文件根路径；

（2）执行命令 npm install 安装项目运行所需要的包；

（3）执行命令 node app.js 运行后端项目。项目运行过程中不要关闭命令窗口。

2. 启动项目前端工程步骤：

（1）打开项目前端文件夹根路径，在地址栏中输入“cmd”打开命令窗口，或者应用 cd 命令进入项目后端文件根路径；

（2）执行命令 npm install 安装项目运行所需要的包；

（3）执行命令 npm run serve 运行前端项目。项目运行过程中不要关闭命令窗口。