服务器端搭建

SSM框架：Spring+SpringMVC+MyBatis SpringBoot工具

创建所有项目都是基于maven创建。 maven项目管理(清理-编译-测试-打包-部署)和依赖(jar)管理的工具。

> 解压maven

```
D:\tools\apache-maven-3.8.8
```

> 配置maven

打开D:\tools\apache-maven-3.8.8\conf\settings.xml

```
-- 本地仓库    首先创建一个本地仓库文件夹 D:\tools\repository2
<!-- 配置maven的本地仓库的地址 -->
<localRepository>D:\tools\repository2</localRepository>

-- 下载jar服务器镜像改为国内
<mirror>
    <id>aliyunmaven</id>
    <mirrorOf>*</mirrorOf>
    <name>阿里云公共仓库</name>
    <url>https://maven.aliyun.com/repository/public</url>
</mirror>
```

> maven集成Idea

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- 当前项目坐标 -->
    <groupId>com.haiyang</groupId>
    <artifactId>mall-server-021</artifactId>
    <version>1.0.0</version>

    <packaging>jar</packaging>

    <!-- 运行参数 -->
    <properties>
        <java.version>1.8</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <!-- springboot版本 -->
        <spring-boot.version>2.3.7.RELEASE</spring-boot.version>
    </properties>

    <!-- 依赖 -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- MySQL数据库驱动-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <!-- lombok插件-->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>2.3.7.RELEASE</version>
            </plugin>
        </plugins>
    </build>
</project>
```

application.yaml

```
server:
  port: 10001
  servlet:
    context-path: /
```

```
注意：
    yaml文件配置属性节点层次必须对其。
    属性值前面必须有一个空格

项目启动，浏览器访问：
http://localhost:10001/index

如果设置 context-path: /api
http://localhost:10001/api/index
```

> 启动类

包：包名全部小写。包是分层级。

```
例如：com.haiyang
对应物理磁盘上就是文件夹。

包名--域名倒置   cctv.com   baidu.com
```

分层开发：

```
Vue界面
    |
控制层：  处理前端发来的请求. Vue请求( 登录 )。    com.haiyang.controller
    |
业务逻辑层   项目具体功能和流程   登录方法    com.haiyang.service
    |
数据访问层：  对数据库CRUD操作        com.haiyang.mapper
    |
MySQL
```

```
package com.haiyang;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MallServer021Application {
    public static void main(String[] args) {
        SpringApplication.run(MallServer021Application.class, args);
    }
}
```

> 创建IndexController控制器

```
package com.haiyang.controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController   //当前类是一个控制器
@RequestMapping("/index")
public class IndexController {
```

```
    //处理请求方法
    //方法返回数据  就是响应给前端的数据
    @RequestMapping("/test")
    public String test(){
        return "Hello SpringBoot";
    }
}
```

实体类

封装数据:

```
MySQL sys_account表 -----查询account_id是19988999988----->实体类对象
      sys_account表-------------所有用户--------------->List<实体类>    泛型

Account       sys_account
实体类<--映射-->表
变量  --------- 字段
```

```
---实体类代码结构---
私有变量 -- 映射表中字段
无参构造方法
变量set和get方法
```

```
package com.haiyang.entity;
import lombok.Data;

@Data
public class Account {
    private String accountId;
    private String password;
    private String accountName;
}
```

代码逆向生成

```
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>5.1.49</version>
</dependency>
```

编写控制器的父类

```
package com.haiyang.common;
//控制器的父类
public class BaseController {

}
```

编写实体类父类

```
package com.haiyang.common;
import com.baomidou.mybatisplus.annotation.TableField;
import java.time.LocalDateTime;
@Data
public class BaseEntity {
    /**
     * 创建时间
     */
    @TableField("created")
    private LocalDateTime created;

    /**
     * 修改时间
     */
    @TableField("updated")
    private LocalDateTime updated;

    @TableField("statu")
    private Integer statu;
}
```

> 配置mybatis-plus框架

导入mybatisPlus逆向代码生成的相关jar包，修改pom.xml

```
<!--  mybatis-plus -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.2</version>
</dependency>
<!-- 代码生成器 -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>3.4.1</version>
</dependency>
<!-- 代码自动生成器模板依赖-->
<dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity-engine-core</artifactId>
    <version>2.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>
```

配置springboot项目，整合myBatisPlus，修改application.yaml

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/system?useUnicode=true&characterEncoding=utf-
8&useSSL=false&serverTimezone=GMT%2B8
    username: root
    password: root
```

修改CategroyMapper接口

```
加上@Mapper
package com.haiyang.mapper;
import com.haiyang.entity.Category;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface CategoryMapper extends BaseMapper<Category> {

}
```

编写CategoryController中测试方法

```
package com.haiyang.controller;
import com.haiyang.entity.Category;
import com.haiyang.service.CategoryService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;
import com.haiyang.common.BaseController;

import java.util.List;

@RestController
@RequestMapping("/category")
public class CategoryController extends BaseController {
    //需要创建CategoryService对象
    //@Autowired Spring框架自动创建好该对象
    @Autowired
    private CategoryService cService;

    //处理 前端请求所有商家分类的请求
    @RequestMapping("/list")
    public List<Category> list(){
        //获得所有的商家分类数据  sys_category表中数据

        //返回 List<Category>
        return cService.list();
    }
}
```

```
访问地址:http://localhost:10001/category/list
```

定义接口通用返回类型

所有的Controller控制器中的方法都是返回Result类

```
返回类代码结构：
Result.java
1、code 前端请求服务器端状态码    20000成功   30001          30002
2、message    请求操作提示信息      执行成功     用户名不存在     密码错误
3、Object     返回数据，Object存储就是返回数据对象，实体类对象，List集合对象。
    动态原理：  父类是可以引用子类对象。
```

Result.java

```
package com.haiyang.common;
import lombok.Data;

@Data
public class Result {
    private Integer code;  //请求操作是否成功状态码 20000成功
    private String message;   //请求操作提示信息。
    private Object resultdata;  //如果请求进行是查询操作，存储就查询返回数据对象。List集合,实体
类。

    //成功：   重载
    public static Result success(int code,String message,Object data){
        Result r = new Result();
        r.setCode(code);
        r.setMessage(message);
        r.setResultdata(data);
        return r;
    }

    public static Result success(Object data){
        return success(20000,"操作成功",data);
    }

    //失败：
    public static Result fail(int code,String message,Object data){
        Result r = new Result();
        r.setCode(code);
        r.setMessage(message);
        r.setResultdata(data);
        return r;
    }
    public static Result fail(String message){
        return fail(400,message,null);
    }
}
```

SpringBoot项目热部署

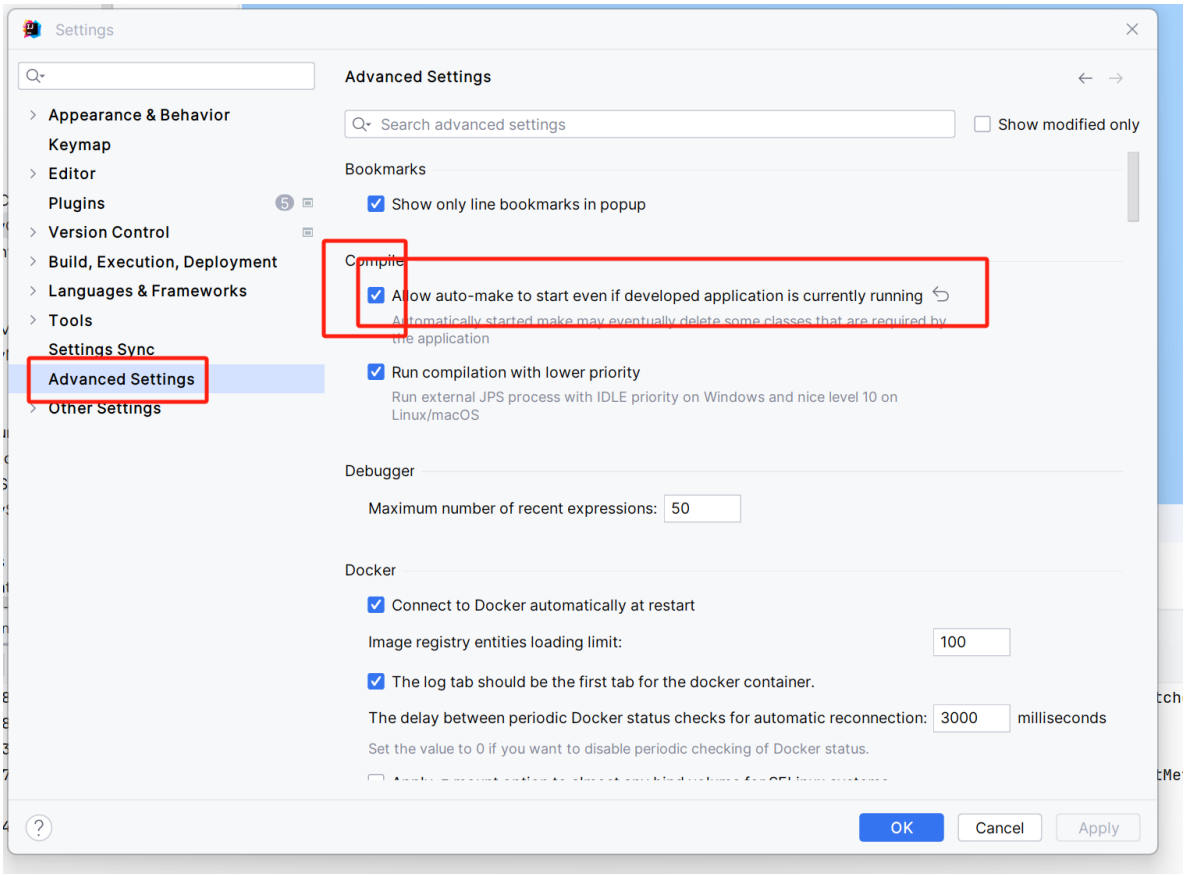
修改代码不需要手动重启服务器（自动）。

首先需要修改pom.xml文件，添加依赖

```
 <!-- 热部署 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

配置Idea自动编译

路径：File-> Settings -> Build,Execution,Deployment -> Compiler -> Build project automatically



注意： 配置完毕，将服务器先手动重启一下，以后修改代码变为热部署（自动重启）。

MyBatis配置类

```
Spring框架配置：
1、XML配置，所有的配置都是以.xml文件为主。
2、基于注解配置，所有的配置使用注解完成。
    @Autowired 注解，自动装配
3、Java配置，所有配置以一个类为主（配置类 -- 方法[配置对象信息]）。
    类 @Configuration
            方法 @Bean  -->创建以类对象

Spring核心概念：
    依赖注入，通过Ioc容器。解耦
    面向切面，通过AOP实现。
```

```
目前项目：
    @RestController
    CategoryController
        |                          @Service
    CategoryService接口 ----实现---- CategoryServiceImpl实现类（生成）
        |   @Mapper
    CatgoryMapper接口（生成）

springIoc可以将三层中 控制层、业务逻辑层、数据访问层对象全部自动创建好，自动创建好的对象存到Ioc容器（内存）。
```

添加MyBatisPlus配置类，创建一个分页拦截器对象到 Ioc容器

```
package com.haiyang.config;
import com.baomidou.mybatisplus.annotation.DbType;
import com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor;
import com.baomidou.mybatisplus.extension.plugins.inner.BlockAttackInnerInterceptor;
import com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInterceptor;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
@MapperScan("com.haiyang.mapper")  //所有Mapper接口就不需要手动添加@Mapper
public class MyBatisPlusConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        //分页  方言
        interceptor.addInnerInterceptor( new PaginationInnerInterceptor( DbType.MYSQL
));
        //防止全表更新和删除
        interceptor.addInnerInterceptor( new BlockAttackInnerInterceptor());
        return interceptor;
    }
}
```

加入密码加密工具类

首先配置pom.xml，导入依赖

```
<!-- MD5加密 -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.15</version>
</dependency>
```

创建一个MD5加密的工具类

```
package com.haiyang.utils;
import org.apache.commons.codec.digest.DigestUtils;
public class MD5Utils {
    //调用MD5加密
    public static String md5(String str){
        return DigestUtils.md5Hex(str);
    }
    //加密字符 基数
    private static final String privateKey = "1xc2d34f3p";

    //123123 ，密码加工
    public static String inputPassToNewPass(String pass){
        String newPass =
privateKey.charAt(0)+privateKey.charAt(1)+pass+privateKey.charAt(5)+"";
        return newPass; //1x1231233
    }

    public static void main(String[] args) {
        System.out.println(MD5Utils.md5("123123"));
    }
}
```

登录功能的实现

编写服务器端 AccountController中的login方法

```
package com.haiyang.controller;
import com.haiyang.common.Result;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;
import com.haiyang.common.BaseController;

@RestController
@RequestMapping("/account")
@Slf4j
public class AccountController extends BaseController {

    @PostMapping("/login")
    public Result login(String accountId,String password){
        log.info("手机号为 {} 用户正在登录APP端--",accountId);

        return Result.success(null);
    }
}
```

密码加密之后如何对比

```
注册:
    原始123123 ---->录入数据库---->加密---->4297f44b13955235245b2497399d7a93
登录:
    页面输入原文123123---->先加密(4297f44b13955235245b2497399d7a93)---->用密文对比密文<----
--4297f44b13955235245b2497399d7a93
```

```
@PostMapping("/login")
    public Result login(String accountId,String password){
        log.info("手机号为 {} 用户正在登录APP端--",accountId);

        //手机号在sys_account表中是主键，是不重复的，查询返回就是单一对象
        Account account = aService.getOne(new QueryWrapper<Account>
().eq("account_id",accountId));

        if(account==null){
           return Result.fail("账户手机号码不存在");
        }else{
            //account不等于null，说明查询到该手机号码，继续比较密码是否一致
            String newPwd = MD5Utils.md5(password);  //将登录原文密码 先加密，变为 密文
            if(newPwd.equals(account.getPassword())){
                if(account.getStatu() == 0){
                    return Result.fail("该账户被禁用或被注销，暂不可用");
                }else{
                    //登录成功，直接反馈登录账户对象信息
                    return Result.success(account);
                }
            }else{
                return Result.fail("登录密码不正确");
```

```
            }
        }
    }
```

登录前端代码实现

前端请求服务器，必须 ajax异步请求技术。

Ajax技术实现就是使用javascript实现。 直接使用Javascript原生代码效率低。

axios框架完成异步请求。对应请求方法：get()、post()，使用npm安装插件库。

```
npm install axios --save

还需安装一个插件库 qs作用就是将传递的json参数转化为 参数形式.
{
  accountId:'13388998899',  //登录手机号
  password:'123123',   //密码
}
前端传递给服务器端，传递是这个account对象,后端Java是无法处理json对象，需要将json对象转换为 参数形式。
accountId=13388998899&password=123123

npm install qs --save
```

再创建工具类封装axios

/api/index.js文件

```
创建axios对象，做一些基础设置

常用get方式请求，封装成一个get().
常用post方式请求，封装成一个post().
```

Login.vue

```
import Footer from '@/components/Footer.vue';
import {ref,reactive} from "vue"
import {useRouter} from "vue-router"
import {get,post} from "@/api/index.js"
import { ElMessage } from 'element-plus'
import {setSessionStorage} from '@/common.js'

const login = () =>{
  //通过loginFrom表单对象，调用该对象 validate()验证方法，如果定义验证规则通过，v是true，否则是false
  loginForm.value.validate((v,f)=>{
    if(v){
      post('/account/login',account,false).then(res=>{
        //响应处理代码  res.data
        if(res.data.code == 20000){
          setSessionStorage('account',res.data.resultdata);
          //跳转至 首页
          router.push('/')

          ElMessage({
            message: '登录成功',
```

```
            type: 'success',
          })
        }else{
          ElMessage({
            message: res.data.message,
            type: 'error',
          })

        }
      });
    }else{
      return false;
    }
  });
}
```

跨域

浏览器：同源策略

```
vue程序  ------------------------------------------>   Java服务器
http://localhost:8080/                              http://localhost:10001/

http:请求协议
localhost: 请求IP地址
8080或10001 端口
```

```
在Contoller控制器类上，加上注解@CrossOrigin
```

注册相关功能实现

手机号码验证实现

传递参数: RESTFul风格，get请求浏览器地址传参。

```
http://localhost:10001/account请求地址
? 分隔，问号后面就是传递参数    传统方式: 参数名=值&参数名=值&参数名=值
http://localhost:10001/account/check?accountId=1990099009


RESTFul风格
参数是请求地址一部分。
http://localhost:10001/account/check/1990099009
```

```
 @GetMapping("/check/{accountId}")
public Result check(@PathVariable String accountId){
    Account account = aService.getById(accountId);
    if(account==null){
        return Result.success("手机号码可以注册");
    }else{
        return Result.fail(20005,"手机号码已经注册",null);
    }
}
```

> 前端请求 手机号码验证

```
//验证手机号码
const checkAccountId=()=>{
  if(account.accountId !=''){
    let url = `/account/check/${account.accountId}`;
    get(url).then(res =>{
      if(res.data.code == 20005){
          ElMessage({
            message: res.data.message,
            type: 'error',
          });

          account.accountId='';  //填入的手机号清空
      }
    });
  }
}
```

注册功能实现

解决难题，在Java服务器端如何接收前端请求json对象参数

```
public Result register(@RequestBody Account account)

@RequestBody 接收前端提交json对象，并且转化为Java中的Account对象。
```

> mybatisplus生成实体类的注解

```
@TableId(value = "account_id", type = IdType.AUTO)  映射数据库表中主键
    type 设置主键映射策略。
            IdType.AUTO 自动增长
            IdType.INPUT 主键手动录入
            IdType.UUID  随机一段字符作为主键
@TableField  映射数据库表中字段
@TableName("sys_account")   映射数据库表名
```

因为accountId是用户注册的手机号码，也是sys_account表中主键，所以该表的主键策略需要调整为INPUT，表示主键的值是手动录入，需要修改Account实体类。

```
@TableId(value = "account_id", type = IdType.INPUT)
private String accountId;
```

> 服务器端注册接口实现

```
@PostMapping("/register")
public Result register(@RequestBody Account account){
    //对页面提交密码在此加密
    account.setPassword( MD5Utils.md5(account.getPassword()) );
    account.setCreated(LocalDateTime.now());
    account.setUpdated(LocalDateTime.now());
    account.setStatu(1);  //0禁用  1正常
    if(account.getAccountSex()==1){
        account.setAccountImg(Const.DEFAULT_IMG_1 );
    }else{
        account.setAccountImg(Const.DEFAULT_IMG_0);
    }
    //插入数据
    aService.save(account);
    return Result.success("用户信息注册成功");
}
```

> 前端注册请求实现

```
const register =  () =>{
  //通过loginFrom表单对象，调用该对象 validate()验证方法，如果定义验证规则通过，v是true，否则是
false
   registerForm.value.validate((v,f)=>{
    if(v){
      let url = "/account/register";
      post(url,account,true).then(res=>{
        if(res.data.code == 20000){
          ElMessage({
            message: '注册成功，请登录',
            type: 'success',
          })
          router.push('/login');
        }
      });
    }else{
      return false;
    }
  });
}
```

前端首页相关实现

> 显示登录成功的用户头像

登录成功，会将用户的信息对象存储到sessionStorage，直接取。

```
v-if 指令
v-if="方法|表达式"
方法或者表达式为true，该标签显示，如果为false，就不显示
```

> 前端添加判断是否登录的方法

```
import { getSessionStorage } from "@/common.js"

//用户登录信息
const account = getSessionStorage("account");
```

```
//判断是否登录
const isLogin = ()=>{
    if( !account || getSessionStorage("account")==null){
        return false;
    }else{
        return true;
    }
}
```

页面判断实现用户名和用户头像

```
<!-- 头部分 -->
<div class="header">
    <div class="location-text">
        <i class="location_icon" ></i>
        {{ account?account.accountName:'' }}
    </div>
    <!-- 用户登录之后显示头像 -->

    <div class="location-account" v-if="isLogin" >
        <img :src="account.accountImg" />
    </div>
</div>
```

> 商品分类数据和商家数据显示

```
package com.haiyang.controller;
import com.haiyang.common.Result;
import com.haiyang.entity.Category;
import com.haiyang.service.CategoryService;
import com.haiyang.service.impl.CategoryServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;
import com.haiyang.common.BaseController;

import java.util.List;

@RestController
@RequestMapping("/category")
@CrossOrigin
public class CategoryController extends BaseController {
    //需要创建CategoryService对象
    //@Autowired Spring框架自动创建好该对象
    @Autowired
    private CategoryService cService;

    //处理 前端请求所有商家分类的请求
    @GetMapping("/list")
    public Result list(){
        //获得所有的商家分类数据  sys_category表中数据
        //返回 List<Category>

        List<Category> list = cService.list();
```

```
        return Result.success(list);
    }
}
```

```
package com.haiyang.controller;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.haiyang.common.Result;
import com.haiyang.entity.Business;
import com.haiyang.service.BusinessService;
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 com.haiyang.common.BaseController;

import java.util.List;

@RestController
@CrossOrigin
@RequestMapping("/business")
public class BusinessController extends BaseController {

    @Autowired
    private BusinessService bService;

    @GetMapping("/list")
    public Result list(){
        List<Business> list = bService.list(new QueryWrapper<Business>
().ne("statu",0));
        if(list == null){
            return Result.fail(30001,"暂无商家数据显示",null);
        }
        return Result.success(list);
    }
}
```

> 前端请求商家分类和商家数据

```
//加载页面的商家数据
const loadBusiness = ()=>{
    get('/business/list').then(res=>{
        businessList.value = res.data.resultdata;
    });
}

//加载页面的商家分类
const loadCategory=()=>{
    get('/category/list').then(res=>{
        categoryList.value = res.data.resultdata;
    });
}

//页面初始化的方法
const init = ()=>{
    loadCategory();
    loadBusiness();
```

```
}
init();  //调用
```

首页动态效果

```
Vue组件生命周期
    钩子函数();  不需要我们手动调用，自动调用（根据生命周期到达哪个阶段）。
```

```
/* ---头部固定搜索--- */
const fixedBox = ref(null);
onMounted(() => {
  document.onscroll = () => {
    //获取滚动条位置
    let s1 = document.documentElement.scrollTop;
    let s2 = document.body.scrollTop;
    let scroll = s1 == 0 ? s2 : s1;
    //获取视口宽度
    let width = document.documentElement.clientWidth;

    //获取顶部固定块
    let search = fixedBox.value;

    //判断滚动条超过视口宽度的12%时，搜索块变固定定位
    if (scroll > width * 0.12) {
      search.style.position = "fixed";
      search.style.left = "0";
      search.style.top = "0";
    } else {
      search.style.position = "static";
    }
  };
});

// 销毁
onUnmounted(() => {
  //当切换到其他组件时，就不需要document滚动条事件，所以将此事件去掉
  document.onscroll = null;
});
```

首页添加商品分类更多

```
<ul class="category-ul">
    <span>...</span>  <!-- 添加商品分类更多 按钮选项 -->
    <li v-for="(item,index) in categoryList" :key="item.categoryId">
        <img :src="item.categoryCover" />
        <p>{{ item.categoryName }}</p>
    </li>
</ul>
```

修改和添加样式

```
/*覆盖样式*/
.wrapper .category-ul{
```

```
    width:100%; height: 44vw; display: flex; flex-wrap: wrap;
    justify-content: space-around;
    align-content:center; padding:4.4vw; box-sizing: border-box;
    position: relative;
}
/*添加样式*/
.wrapper .category-ul span{
    position: absolute;
    right: 1.5vw;
    top:-2vw;
    background-color: #ffde09;
    color:#fb8b06;
    border:0.3vw solid #444;
    border-radius: 1vw;
    padding: 0.5vw 2vw;
    font-weight: 800;

    box-shadow: 0.2vw 0.2vw 0.2vw rgba(0, 0, 0, 0.5);
    z-index: 9999;
    height:5vw;
}
```

根据商家分类 查询该分类下的商家数据

```
@Autowired
private BusinessService bService;
@Autowired
private BusinessCategoryService bcService;

@GetMapping("/listByCategoryId/{categoryId}")
public Result listByCategoryId(@PathVariable Integer categoryId){
    //步骤1：先通过分类编号 查询sys_business_category表中 某个分类下所有的商家编号。
    List<BusinessCategory> bcList = bcService.list(new QueryWrapper<BusinessCategory>
().eq("category_id", categoryId));
    List<Business> businessList = new ArrayList<>();

    //        for(int i=0;i<bcList.size();i++){
    //            Long businessId = bcList.get(i).getBusinessId();
    //            Business business = bService.getById(businessId);
    //            businessList.add(business);
    //        }

    bcList.stream().forEach(bc -> {
        Business business = bService.getById(bc.getBusinessId());
        businessList.add(business);
    });

    return Result.success(businessList);
}
```

> 前端修改Home.vue首页的代码

实现点击分类的图标，获得分类的编号categoryId，然后跳转至商家列表页面，同时传递categoryId分类编号到商家列表页面。

```
<li v-for="(item,index) in categoryList" :key="item.categoryId"
@click="toBusinessList(item.categoryId)">
    ........
</li>
```

```
//根据商家分类编号 跳转至商家列表页面
const toBusinessList = (id) =>{
    //需要使用路由跳转时 传递参数
    router.push({path:'/businessList',query:{categoryId:id}})
}
```

> 编写商家列表页面代码

获得首页跳转过来传递的分类编号categoryId参数值：

```
route.query.categoryId;
```

再通过获得categoryId参数值，请求服务器接口查询该分类下面的所有的商家数据

```
import Footer from '../components/Footer.vue'
import {ref} from "vue"
import { get } from '@/api';
import {useRouter,useRoute} from "vue-router"
const router = useRouter();  //创建路由对象
const route = useRoute();

const businessList = ref([]);

//点击商家，获得商家编号，传递至商家详情页面
const toBusinessInfo=(id)=>{
    router.push({path:'/businessInfo',query:{businessId:id}});
}


//请求服务器端，查询该分类下的所有商家数据
const loadBusiness=()=>{
    //获得首页跳转过来传递的分类编号categoryId参数值：
    let categoryId = route.query.categoryId;
    let url = `/business/listByCategoryId/${categoryId}`;
    get(url).then(res=>{
        businessList.value = res.data.resultdata;
    })
}

const init = ()=>{

    loadBusiness();
}
init();
```

商品详情页面的实现

商家的详情数据显示出来

首先通过商家编号businessId查询商家详细数据

```
@GetMapping("/info/{businessId}")
public Result info(@PathVariable Long businessId){
    Business business = bService.getById(businessId);
    if(business == null){
        return  Result.fail("商家的详情数据加载失败");
    }else{
        return Result.success(business);
    }
}
```

修改前端代码，显示商家的数据

```
//获得商家编号businessId，查询商家
const businessId = route.query.businessId;
//页面显示商家详情对象
const business = ref({});

const loadBusinessInfo=()=>{
  let url = `/business/info/${businessId}`;
  get(url).then(res=>{
    if(res.data.code == 20000){
      business.value = res.data.resultdata;
    }else{
        ElMessage({
          message: res.data.message,
          type: 'error',
        })
    }
  });
}
```

> 显示页面上商品数据

实现服务器端的接口

```
@GetMapping("/listByBusinessId/{businessId}")
public Result listByBusinessId(@PathVariable Long businessId){
    List<Goods> goodsList = gService.list(new QueryWrapper<Goods>().eq("business_id",
businessId));
    if(goodsList==null){
        return Result.fail("商品数据加载失败");
    }else{
        return Result.success(goodsList);
    }
}
```

前端页面的代码

```
const loadGoodsByBusinessId=()=>{
  let url = `/goods/listByBusinessId/${businessId}`;
  get(url).then(res=>{
    if(res.data.code == 20000){
      let tempArray = res.data.resultdata;

      //循环tempArray,给每个商品添加quantity属性
      for(let i=0;i<tempArray.length;i++){
        tempArray[i].qauntity = 0;  //每个商品的数量默认为 0
```

```
      }

      goods.value = tempArray;
    }else{
        ElMessage({
          message: '商品数据加载失败',
          type: 'error',
        })
    }
  })
}
```

> ElementPlus 图标使用

```
首先需要安装图片库
npm install @element-plus/icons-vue
```

在 main.js中设置图标库

```
import * as ElementPlusIconsVue from '@element-plus/icons-vue'

// 全局注册所有图标
for (const [key, component] of Object.entries(ElementPlusIconsVue)) {
    app.component(key, component)
}
```

在页面添加图标

```
<el-icon><RemoveFilled /></el-icon>

<el-icon><CirclePlusFilled /></el-icon>
```

> 添加和删减购物车

逻辑:

```
数量:  - 0  +
添加+:
    第一次（数量为0）：向sys_cart表录入一条记录。 insert
    不是第一次添加（数量=>1）: sys_cart表中已经存在该商品，是需要更新数量  update 数量 +1
删减-:
    数量==1：数量是1，再删减执行delete操作。
    数量>1: 执行update，数量-1
```

> 编写服务器端购物车相关接口

```
//插入购物车数据
@PostMapping("/add")
public Result add(@RequestBody Cart cart){
    cart.setCreated(LocalDateTime.now());
    cart.setUpdated(LocalDateTime.now());
    cart.setStatu(1);
    cartService.save(cart);
    //cart.getCartId() 得到录入购物车表中 自动生成主键值。
    return Result.success(cart.getCartId());
}
```

```
//更新购物车数据 （数量）
@PostMapping("/update")
public Result update(@RequestBody Cart cart){
    cart.setUpdated(LocalDateTime.now());
    QueryWrapper<Cart> qw = new QueryWrapper<>();
    qw.eq("goods_id",cart.getGoodsId());
    qw.eq("account_id",cart.getAccountId());
    cartService.update(cart,qw);
    return Result.success("购物车数更新成功");
}

//删除购物车数据
@PostMapping("/remove")
public Result remove(@RequestBody Cart cart){
    cart.setUpdated(LocalDateTime.now());
    QueryWrapper<Cart> qw = new QueryWrapper<>();
    qw.eq("goods_id",cart.getGoodsId());
    qw.eq("account_id",cart.getAccountId());
    cartService.remove(qw);
    return Result.success("购物车数删除成功");
}
```

前端代码

```
//取出登录用户数据
const account = getSessionStorage('account');

const minus =(index)=>{
  if(account==null){
    router.push('/login');
    return;
  }
  if(goods.value[index].quantity==0){
    return;
  }
  if(goods.value[index].quantity > 1){
    //数量-1 执行数据update
    updateCart(index,-1);
  }else{
    //数据库记录 从(1)到(0) ，执行delete
    let url = "/cart/remove";
    let cart = {
      goodsId: goods.value[index].goodsId,
      businessId: businessId,
      accountId: account.accountId,
    };
    post(url,cart,true).then(res=>{
      if(res.data.code==20000){
        goods.value[index].quantity = 0;
      }
    })
  }
}

const add = (index)=>{
  if(account==null){
    router.push('/login');
    return;
  }
```

```
  if(goods.value[index].quantity == 0){
    //购物车记录 (0)到(1)，往数据库新增(insert)
    let url = "/cart/add";
    let cart = {
      goodsId: goods.value[index].goodsId,
      businessId: businessId,
      accountId: account.accountId,
      quantity:1   //减 -1  加 1
    };
    post(url,cart,true).then(res=>{
      if(res.data.code==20000){
        goods.value[index].quantity = 1;
      }
    });

  }else{
    //购物车记录已经存在该商品，执行 数量+1  （update）
    updateCart(index,1);
  }
}

const updateCart=(index,num)=>{
  let url = "/cart/update";
  let cart = {
    goodsId: goods.value[index].goodsId,
    businessId: businessId,
    accountId: account.accountId,
    quantity:goods.value[index].quantity + num   //减 -1  加 1
  };
  post(url,cart,true).then(res=>{
    if(res.data.code==20000){
      goods.value[index].quantity += num;
    }
  });
}
```

> v-show指令

```
两个指令的作用完全一样，就是控制标签是否显示。
v-if="true"    如果为false，标签不显示是 根本就不会加载到页面。
v-show="true"  如果为fasle，标签会加载页面，只是设置不显示 display=none
```

计算属性

```
变量（改变）<------------>计算属性 （新）
```

```
const 计算名字 = computed( ()=>{
    //计算规则
} );
```

> 计算购物车总价

```
// 计算属性
const totalPrice = computed(()=>{
  let s =0;
  for(let item of goods.value){
    s += item.quantity * item.goodsPrice;
  }
  return s;
});
```

```
//计算属性 计算商品数量
const totalQuantity = computed(()=>{
  let count =0;
  for(let g of goods.value){
    count += g.quantity;
  }
  return count;
});
```

加载购物车数据

```
//商家详情页面 加载购物车方法
@GetMapping("/listCart/{accountId}/{businessId}")
public Result listCart(@PathVariable String accountId,@PathVariable Long businessId){
    QueryWrapper<Cart> qw = new QueryWrapper<>();
    qw.eq("account_id",accountId);
    qw.eq("business_id",businessId);

    List<Cart> list = cartService.list(qw);
    return Result.success(list);
}
```

编写前端

```
//加载购物车数据
const loadCart = ()=>{
  let url = `/cart/listCart/${account.accountId}/${businessId}`;
  get(url).then(res=>{
    if(res.data.code == 20000){
      let cartArray = res.data.resultdata;
      for(let g of goods.value){
        g.quantity = 0;

        for(let cart of cartArray){
          if(g.goodsId == cart.goodsId){
            g.quantity = cart.quantity;
          }
        }
      }
    }
  });
}
```

上面这个方法，调用时间必须是在页面调用商品操作完毕:

```
//先查询商品数据，再取购物车数据
if(account!=null){
    loadCart();
}
```

修改businessList商家列表页面，加载购物车角标

服务器端添加了一个参数accountId查询购物车数据的方法

```
 @GetMapping("/listCartByAccountId/{accountId}")
public Result listCart(@PathVariable String accountId) {
    QueryWrapper<Cart> qw = new QueryWrapper<>();
    qw.eq("account_id",accountId);
    List<Cart> list = cartService.list(qw);
    return Result.success(list);
}
```

修改BusinessList页面，添加加载购物车角标方法

```
//加载购物车数据
const loadCart =()=>{
    let url = `/cart/listCartByAccountId/${account.accountId}`;
    get(url).then(res=>{
        if(res.data.code == 20000){
            let cartArray = res.data.resultdata;
            for(let i=0;i<businessList.value.length;i++){
                businessList.value[i].quantity =0;
                for(let cart of cartArray){
                    if(businessList.value[i].businessId == cart.businessId){
                        businessList.value[i].quantity += cart.quantity;
                    }
                }
            }
        }
    });
}
```

```
调用loadCart()方法
get(url).then(res=>{
    businessList.value = res.data.resultdata;

    //先加载页面上商家的数据，然后再去加载购物车角标
    if(account!=null){
        loadCart();
    }
})
```

订单确认页面的数据显示

定义服务器端，修改Cart实体类，添加一个自定义属性Goods goods，为了查询购物车数据的同时也一起获得商品的对象数据。

```
//自定义：商品对象 ，表示该变量不是映射数据库sys_cart表，该字段就是为了查询需要添加的
@TableField(exist = false)
private Goods goods;
```

修改CartController中加载购物车的方法，查询购物车同时获得商品详细对象

```
//商家详情页面 加载购物车方法
@GetMapping("/listCart/{accountId}/{businessId}")
public Result listCart(@PathVariable String accountId,@PathVariable Long businessId){
    QueryWrapper<Cart> qw = new QueryWrapper<>();
    qw.eq("account_id",accountId);
    qw.eq("business_id",businessId);

    List<Cart> list = cartService.list(qw);

    //使用购物车中 goods_id，再次查询商品数据
    list.stream().forEach( cart ->{
        Goods goods = goodsService.getById(cart.getGoodsId());
        cart.setGoods(goods);
    });
    return Result.success(list);
}
```

前端确认订单的编码

```
//加载商家对象信息
const loadBusiness=()=>{
    let url = `/business/info/${businessId}`;
    get(url).then(res=>{
        if(res.data.code == 20000){
            business.value = res.data.resultdata;

            //查询商家下面 当前用户购买的商品信息
            loadCart();
        }else{
            ElMessage({
                message: '订单商家数据加载失败',
                type: 'error',
            })
        }
    });
}
```

```
//加载购物车的数据
const loadCart=()=>{
    let url =`/cart/listCart/${account.accountId}/${businessId}`;
    get(url).then(res=>{
        if(res.data.code == 20000){
            cartList.value = res.data.resultdata;
        }else{
            ElMessage({
                message: '订单的商品数据加载失败',
                type: 'error',
            })
        }
    });
}
```

编写计算属性 计算订单总价

```
const totalPrice = computed(()=>{
    let total =0;
    cartList.value.forEach(cart=>{
        total += cart.goods.goodsPrice * cart.quantity;
    });
    //商品总价+配送费
    total+= business.value.deliveryPrice;
    return total;
});
```

设置默认的配送信息

服务器端定义配送信息的请求接口

```
@RestController
@RequestMapping("/deliveryaddress")
@CrossOrigin
public class DeliveryaddressController extends BaseController {
    @Autowired
    private DeliveryaddressService dyService;

    @GetMapping("/list/{accountId}")
    public Result list(@PathVariable String accountId){
        QueryWrapper<Deliveryaddress> qw = new QueryWrapper<>();
        qw.eq("account_id",accountId);
        List<Deliveryaddress> list = dyService.list(qw);
        if(list == null){
            return Result.fail("配送信息加载失败");
        }else{
            return Result.success(list);
        }
    }
}
```

编辑 address.vue

```
const init =()=>{
    let url=`/deliveryaddress/list/${account.accountId}`;
    get(url).then(res=>{
        if(res.data.code == 20000){
            addresslist.value = res.data.resultdata
        }
    })
}
init();
```

点击配送信息，设置默认的配送地址

```
//设置默认配送地址的方法
const setDefaultAddress = (da) =>{
    setLocalStorage(account.accountId,da);
    //默认配送地址设置成功，返回上一页(确认订单)
    ElMessage({
        message: '配送信息设置成功',
        type: 'success',
    })
    router.push({path:'/orderConfirm',query:{businessId:businessId}});
}
```

修改订单确认页面，读取默认配送信息

```
  const deliveryAddress = getLocalStorage(account.accountId);
```

定义全局过滤器

首先在前端项目src下面创建filter文件夹，创建index.js

```
//全局过滤器
export default{
    //value就是过滤的数据，value就是查询 1或0
    fmtSex:(value)=>{
        if(value==0){
            return '女士';
        }else if(value == 1){
            return '先生';
        }else{
            return '性别出错'
        }
    },
    fmtPrice:(value)=>{
        return parseFloat(value).toFixed(2);
    }
}
```

需要再main.js入口文件中，对全局过滤器进行注册，注册之后在Vue组件界面就可以直接使用

```
import filter from './filter/index.js'
//全局注册过滤器
Object.keys(filter).forEach(key => {
    app.config.globalProperties[`$${key}`] = filter[key];
});
```

MyBatisPlus

MyBatisPlus常规查询是不需要输入SQL，定义查询条件(QueryWrapper)，根据条件自动生成执行SQL。

MyBatisPlus在 service层次生成CRUD的方法：

```
getOne()    查询单一对象，返回是一个Java对象。   根据主键作为查询条件，进行查询返回的都是单一对象。
getById()   根据主键查询对象。
list();     查询所有的数据。

save(对象);   插入数据
update(对象,Qw);   qw是更新条件对象
remove(qw);    qw是删除条件
```

数据库

```
1 --- 多

sys_category 多 ----- sys_business  多   KFC 美食、早餐
1 早餐                       101 KFC
2 美食                       102 老盛昌
3 汉堡
                关系表
             sys_business_category
             1                101
             2                101
             3                101
             1                102
             2                102
```