**Day1**

如何在Mac上运行nginx CSDN文章<http://t.csdnimg.cn/HJMMT>

**nginx反向代理**

server{

listen 80;

server\_name localost;

location/api/{

proxy\_pass http://localhost:8080/admin/;//反向代理

}

}

即将localtion/api 之前的地址转换为<http://localhost:8080/admin/> 如<http://localhost/api/employee/login> 转化为<http://localhost:8080/admin/employee/login>

**nginx 反向代理的好处:**

1、提高访问速度

2、进行负载均衡(即将前端请求平均分配给每一个服务器)

3、保证后端安全

**nginx负载均衡**

底层是通过反向代理来实现的 weight代表权重

upstream webservers{

server 192.168.100.128:8080;

server 192.168.100.129:8080;//后端Tomcat服务器端口号

}

server{

listen 80;

server\_name localhost;

location/api{

proxy\_pass http://webservers/admin/;//负载均衡

}

}

**完善登录功能**

将密码进行加密存储，提高安全性，主要使用了MD5加密算法

// 对前端的密码进行MD5加密算法

password=DigestUtils.md5DigestAsHex(password.getBytes());

**前后端分离开发流程**

确定接口文档---> 前后端并行开发----> 前后端联调

**Swagger**

主要通过knife4j来生成接口文档 其中包括扫描controller包还有静态资源的映射

**区分Yapi与Swagger （同时使用到）**

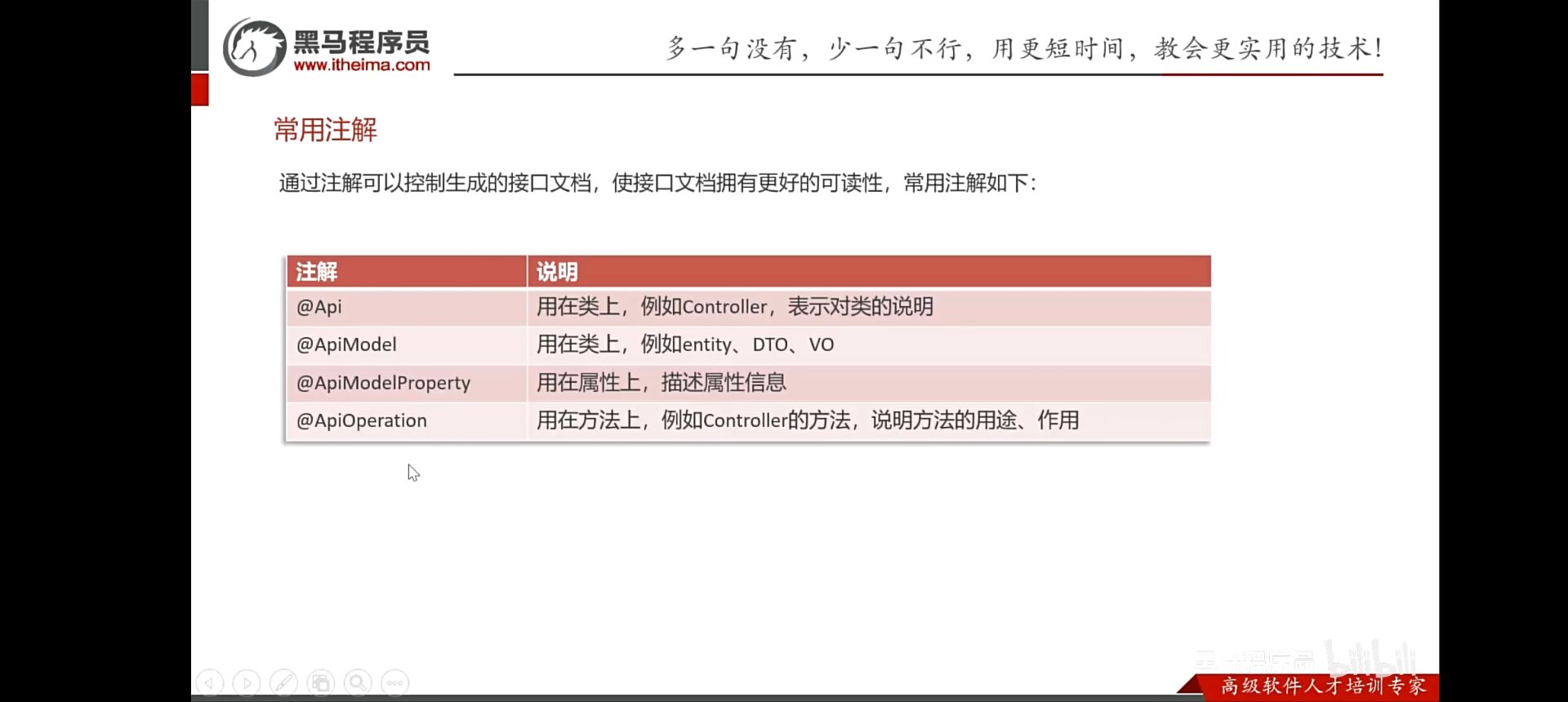
Yapi主要是设计阶段使用的工具，管理和维护接口

Swagger在开发阶段使用的框架，帮助后端开发人员做后端接口测试





Swagger常用注解



**Day2**

**小知识点总结**

报错500是服务端错误 报错401代表未授权

**新增员工**

**对象属性的拷贝**

// 对象属性拷贝

BeanUtils.copyProperties(employeeDTO,employee);

把前面的对象的属性拷贝给后面的对象 顺序不要搞错了

**MD5加密**

// 设置密码 默认密码为123456 加密方式为MD5

employee.setPassword((DigestUtils.md5DigestAsHex("123456".getBytes())));

**swagger文档调试**

添加一个全局的请求头 即JWT令牌 在文档管理中——>全局参数设置

**代码完善**

1、未处理添加员工时username重复的情况

/\*

\* 处理SQL异常

\* \*/

@ExceptionHandler

public Result exceptionHandler(SQLIntegrityConstraintViolationException ex){

String message=ex.getMessage();

if(message.contains("Duplicate entry")){

String[] split=message.split(" ");

String username=split[2];

String msg=username+ MessageConstant.ALREADY\_EXISTS;

return Result.error(msg);

}else {

return Result.error(MessageConstant.UNKNOWN\_ERROR);

}

}

**员工分页查询**

利用pagehelpher插件来实现

Controller层

@GetMapping("/page")

@ApiOperation("员工分页查询")

public Result<PageResult> page(EmployeePageQueryDTO employeePageQueryDTO){

log.info("员工分页查询，参数为:{}",employeePageQueryDTO);

PageResult pageResult=employeeService.pageQuery(employeePageQueryDTO);

return Result.success(pageResult);

}

Services层

@Override

public PageResult pageQuery(EmployeePageQueryDTO employeePageQueryDTO) {

//利用了PageHelper插件

//开始分页查询

PageHelper.startPage(employeePageQueryDTO.getPage(),employeePageQueryDTO.getPageSize());

Page<Employee> page=employeeMapper.pageQuery(employeePageQueryDTO);

long total= page.getTotal();

List<Employee> records=page.getResult();

return new PageResult(total,records);

}

Mapper层 用的mybatis的映射文件

/\*

\* 分页查询

\* \*/

Page<Employee> pageQuery(EmployeePageQueryDTO employeePageQueryDTO);

代码完善 修改日期 日期格式有点问题

第一种加注解

@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")

private LocalDateTime createTime;

第二种添加消息转换器

//扩展Spring MVC框架的消息转换器 用来修改时间格式

@Override

protected void extendMessageConverters(List<HttpMessageConverter<?>> converters) {

log.info("扩展消息转换器.....");

//创建一个消息转换器对象

MappingJackson2HttpMessageConverter converter=new MappingJackson2HttpMessageConverter();

//需要为消息转换器设置一个对象转换器，对象转换器可以将Java对象序列化为json数据 将json转换为Java对象称为反序列化

converter.setObjectMapper(new JacksonObjectMapper());

//将我们自己的消息转换器加入到容器中 即将converter加入到converters中

//消息转换器会有一个排序序列 默认新加入的是最后的 很难使用到 所以加一个参数index 设为第一个使用

converters.add(0,converter);

}

**启用禁用员工账号**

/\*

\* 启用禁用员工账号

\* \*/

@PostMapping("/status/{status}")

@ApiOperation("启用禁用员工账号")

//PathVariable 参数路径

public Result startOrStop(@PathVariable Integer status,Long id){

log.info("启用禁用员工账号:{},{}",status,id);

employeeService.startOrStop(status,id);

return Result.success();

}

两种初始化employee的方式 builder

/\*

\* 启用禁用员工账号

\* \*/

@Override

public void startOrStop(Integer status, Long id) {

/\*Employee employee=new Employee();

employee.setStatus(status);

employee.setId(id);\*/

Employee employee=Employee.builder()

.status(status)

.id(id)

.build();

employeeMapper.update(employee);

}

**编辑员工**

页面回显 然后修改

**根据ID查询员工信息**

Controller层

/\*

\* 根据ID查询员工信息

\* \*/

@GetMapping("/{id}")

@ApiOperation("根据ID查询员工信息")

public Result<Employee> getById(@PathVariable Long id){

Employee employee=employeeService.getById(id);

return Result.success(employee);

}

Service层

/\*

\* 根据ID查询员工

\* \*/

@Override

public Employee getById(Long id) {

Employee employee=employeeMapper.getById(id);

employee.setPassword("\*\*\*\*");

return employee;

}

Mapper层

/\*

\* 根据ID查询信息

\* \*/

@Select("select \* from employee where id=#{id}")

Employee getById(Long id);

**编辑修改员工信息**

Controller层

/\*

\* 编辑员工信息

\* \*/

@PutMapping

@ApiOperation("编辑员工信息")

public Result update(@RequestBody EmployeeDTO employeeDTO){

log.info("编辑员工信息:{}",employeeDTO);

employeeService.update(employeeDTO);

return Result.success();

}

Service层

/\*

\* 修改员工信息

\* \*/

@Override

public void update(EmployeeDTO employeeDTO) {

Employee employee=new Employee();

BeanUtils.copyProperties(employeeDTO,employee);

//拷贝完 还差一些属性

employee.setUpdateTime(LocalDateTime.now());

employee.setUpdateUser(BaseContext.getCurrentId());

//调用之前的update方法

employeeMapper.update(employee);

}

没有Mapper层代码 因为用的已经写好的修改员工的代码

**Day3**

**公共字段自动填充**

比如修改人ID 修改时间 创建人ID 创建时间 均属于公共字段 属于重复性代码

通过

1、自定义注解AutoFill 用于标识需要进行公共字段自动填充的方法

2、自定义切面类AutoFillAspect，统一拦截加入了AutoFill注解的方法，通过反射为公共字段赋值

3、在Mapper方法上加入AutoFill注解

先要自定义一个注解 AutoFill

/\*

\* 自定义注解 用于标识某个方法需要进行功能字段自动填充处理

\* \*/

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface AutoFill {

//数据库操作类型 UPDATE INSERT 只有这两个字段才会使用公共字段填充

//OperationType 中定义了枚举类型

OperationType value();

}

再定义一个切面类

/\*

\* 自定义切面类,实现公共字段自动填充的逻辑

\* \*/

@Aspect

@Component

@Slf4j

public class AutoFillAspect {

/\*

\* 切入点 对类的哪些方法进行拦截

\* \*/

@Pointcut("execution(\* com.sky.mapper.\*.\*(..)) && @annotation(com.sky.annotation.AutoFill)")

public void autoFillPointCut(){

}

/\*

\* 前置通知 为公共字段赋值

\* \*/

@Before("autoFillPointCut()")

//JoinPoint 即连接点

public void autoFill(JoinPoint joinPoint){

log.info("开始进行公共字段的填充.....");

}

}

然后在对应的Mapper方法中加上自定义的注解AutoFill

**切面类具体实现代码**

//获取当前被拦截到的方法操作的数据库操作类型

//获取当前被拦截到的方法操作的数据库操作类型

MethodSignature signature=(MethodSignature) joinPoint.getSignature();//方法签名对象

AutoFill autoFill=signature.getMethod().getAnnotation(AutoFill.class);//获得方法上的注解对象

OperationType operationType=autoFill.value();//获得操作类型 也就是UPDATE或者INSERT

//获取当前被拦截的方法的参数--实体对象

//获取当前被拦截的方法的参数--实体对象

Object[] args=joinPoint.getArgs();//获得所有参数 约定实体对象是第一个参数

if(args==null||args.length==0){

return;

}

Object entity=args[0];

//准备赋值的数据

//准备赋值的数据

LocalDateTime now=LocalDateTime.now();//获取时间

Long currentId= BaseContext.getCurrentId();//获取用户ID

//根据当前不同操作的类型，为对应的属性通过反射来赋值

//根据当前不同操作的类型，为对应的属性通过反射来赋值

if(operationType==OperationType.INSERT){//如果是插入 对四个字段进行赋值

try {

Method setCreatTime=entity.getClass().getDeclaredMethod(AutoFillConstant.SET\_CREATE\_TIME,LocalDateTime.class);

Method setCreateUser=entity.getClass().getDeclaredMethod(AutoFillConstant.SET\_CREATE\_USER,Long.class);

Method setUpdateTime=entity.getClass().getDeclaredMethod(AutoFillConstant.SET\_UPDATE\_TIME, LocalDateTime.class);

Method setUpdateUser=entity.getClass().getDeclaredMethod(AutoFillConstant.SET\_UPDATE\_USER,Long.class);

//通过反射为对象属性赋值

setCreatTime.invoke(entity,now);

setCreateUser.invoke(entity,currentId);

setUpdateTime.invoke(entity,now);

setUpdateUser.invoke(entity,currentId);

} catch (Exception e) {

e.printStackTrace();

}

} else if (operationType==OperationType.UPDATE) {//为两个字段赋值

try {

Method setUpdateTime=entity.getClass().getDeclaredMethod(AutoFillConstant.SET\_UPDATE\_TIME, LocalDateTime.class);

Method setUpdateUser=entity.getClass().getDeclaredMethod(AutoFillConstant.SET\_UPDATE\_USER,Long.class);

//通过反射为对象属性赋值

setUpdateTime.invoke(entity,now);

setUpdateUser.invoke(entity,currentId);

} catch (Exception e) {

e.printStackTrace();

}

}

**新增菜品**

包括一共三个接口 查询菜品分类（已开发） 文件上传 添加菜品

**文件上传**

要使用阿里云的仓库 配置yml文件

alioss:

endpoint: ${sky.alioss.endpoint}

access-key-id: ${sky.alioss.access-key-id}

access-key-secret: ${sky.alioss.access-key-secret}

bucket-name: ${sky.alioss.bucket-name}

alioss:

endpoint: https://[oss-cn-hangzhou.aliyuncs.com](https://oss-cn-hangzhou.aliyuncs.com)

accessKeyId: LTAI5[tPpZCKeJuZgYmRd81JL](https://oss-cn-hangzhou.aliyuncs.com)

accessKeySecret: e[J1yjZQvGCLSURVydkVXLMqk50lsPe](https://oss-cn-hangzhou.aliyuncs.com)

bucketName: web-tl[ias-renhaoming](https://oss-cn-hangzhou.aliyuncs.com)

通用接口 实现文件上传

@RestController

@RequestMapping("/admin/common")

@Api(tags="通用接口")

@Slf4j

public class CommonController {

/\*

\* 文件上传

\* \*/

@PostMapping("/upload")

@ApiOperation("文件上传")

public Result<String> upload(MultipartFile file){

log.info("文件上传:{}",file);

return null;

}

}

具体实现 创建配置类对象

/\*

\* 配置类用于创建AliOssUtil对象

\* \*/

@Configuration

@Slf4j

public class OssConfiguration {

@Bean

@ConditionalOnMissingBean

public AliOssUtil aliOssUtil(AliOssProperties aliOssProperties) {

log.info("开始创建阿里云文件上传工具类对象:{}",aliOssProperties);

return new AliOssUtil(aliOssProperties.getEndpoint(), aliOssProperties.getAccessKeyId(),

aliOssProperties.getAccessKeySecret(), aliOssProperties.getBucketName());

}

}

完成上传功能 上传到阿里云中

@RestController

@RequestMapping("/admin/common")

@Api(tags="通用接口")

@Slf4j

public class CommonController {

@Autowired

private AliOssUtil aliOssUtil;

/\*

\* 文件上传

\* \*/

@PostMapping("/upload")

@ApiOperation("文件上传")

public Result<String> upload(MultipartFile file){

log.info("文件上传:{}",file);

try {

//通过UUID加原始文件名 防止重复而导致的文件覆盖

//原始文件名

String originalFilename=file.getOriginalFilename();

//截取原始文件名的后缀

String extension= originalFilename.substring(originalFilename.lastIndexOf("."));

//构造的新文件名

String objectName = UUID.randomUUID().toString()+extension;

//返回的网址 即文件的请求路径

String filePath = aliOssUtil.upload(file.getBytes(), objectName);

return Result.success(filePath);

} catch (IOException e) {

log.error("文件上传失败:{}",e);

throw new RuntimeException(e);

}

}

}

**新增菜品**

Controller层代码

/\*

\* 菜品管理

\* \*/

@RestController

@RequestMapping("/admin/dish")

@Api(tags="菜品相关接口")

@Slf4j

public class DishController {

@Autowired

private DishService dishService;

/\*

\* 新增菜品

\* \*/

@PostMapping

@ApiOperation("新增菜品")

public Result save(@RequestBody DishDTO dishDTO){

log.info("新增菜品:{}",dishDTO);

dishService.saveWithFlavor(dishDTO);

return Result.success();

}

}

service层

@Service

@Slf4j

public class DishServiceImpl implements DishService {

@Autowired

private DishMapper dishMapper;

@Autowired

private DishFlavorMapper dishFlavorMapper;

/\*

\* 新增菜品和对应的口味

\* \*/

@Override

//在启动类中也要开启才行 事务管理

@Transactional

public void saveWithFlavor(DishDTO dishDTO) {

Dish dish=new Dish();

//通过对象拷贝的方式给dish赋值

BeanUtils.copyProperties(dishDTO,dish);

//向菜品表插入一条数据

dishMapper.insert(dish);

//获取insert属性的主键值 然后给flavors赋值 因为前端没有给id值

Long dishId=dish.getId();

//向口味表插入N条数据 其中口味可能为0

List<DishFlavor> flavors=dishDTO.getFlavors();

if(flavors!=null&&flavors.size()>0){

flavors.forEach(dishFlavor -> {

dishFlavor.setDishId(dishId);

});

dishFlavorMapper.insertBatch(flavors);

}

}

}

两个Mapper 一个是在菜品表中插入菜品 一个在口味表中插入数据

<!-- 获取插入数据生成的主键值-->

<insert id="insert" useGeneratedKeys="true" keyProperty="id">

insert into dish(name, category\_id, price, image, description, create\_time, update\_time, create\_user, update\_user,status)

values

(#{name},#{categoryId},#{price},#{image},#{description},#{createTime},#{updateTime},#{createUser},#{status})

</insert>

</mapper>

<!--动态sql便利-->

<insert id="insertBatch">

insert into dish\_flavor(dish\_id, name, value) VALUES

<foreach collection="flavors" item="df" separator=",">

(#{df.dishId},#{df.name},#{df.value})

</foreach>

</insert>

</mapper>

**菜品分页查询**

业务规则：

根据页码展示菜品信息 每页展示10条数据 分页查询时可以根据需要输入菜品名称、菜品分类、菜品状态进行查询

Controller层

@GetMapping("/page")

@ApiOperation("菜品分页查询")

public Result<PageResult> page(DishPageQueryDTO dishPageQueryDTO){

log.info("菜品分页查询:{}",dishPageQueryDTO);

PageResult pageResult=dishService.pageQuery(dishPageQueryDTO);

return Result.success(pageResult);

}

Service层

/\*

\* 分页查询

\* \*/

@Override

public PageResult pageQuery(DishPageQueryDTO dishPageQueryDTO) {

PageHelper.startPage(dishPageQueryDTO.getPage(),dishPageQueryDTO.getPageSize());

Page<DishVO> page=dishMapper.pageQuery(dishPageQueryDTO);

return new PageResult(page.getTotal(), page.getResult());

}

Mapper层

<select id="pageQuery" resultType="com.sky.vo.DishVO">

select d.\*,c.name as categoryName from dish d left outer join category c on c.id = d.category\_id

<where>

<if test="name !=null">

and d.name like concat('%',#{name},'%')

</if>

<if test="categoryId !=null">

and d.category\_id=#{categoryId}

</if>

<if test="status !=null">

and d.status=#{status}

</if>

</where>

order by d.create\_time desc

</select>

**删除菜品**

业务规则：

可以一次删除一个菜品 也可以批量删除菜品 起售中的菜品不能删除 被套餐关联的菜品不能删除 删除菜品后，关联的口味数据也要被删除

Controller层

/\*

\* 菜品批量删除

\* \*/

@DeleteMapping

@ApiOperation("菜品批量删除")

public Result delete(@RequestParam List<Long> ids){

log.info("菜品批量删除：{}",ids);

dishService.deleteBatch(ids);

return Result.success();

}

Service层实现删除

/\*

\* 菜品批量删除

\* \*/

@Override

@Transactional //事务注解 因为涉及到多个表的查询 为了保证数据的一致性

public void deleteBatch(List<Long> ids) {

//判断当前菜品是否能够删除 ---是否存在起售中的菜品

for (Long id : ids) {

Dish dish=dishMapper.getById(id);

if(dish.getStatus()== StatusConstant.ENABLE){

//当前菜品处于起售中 不能删除

throw new DeletionNotAllowedException(MessageConstant.DISH\_ON\_SALE);

}

}

//判断当前菜品是否能够删除---是否被套餐关联

List<Long> setmealIds = setmealDishMapper.getSetmealIdsByDishIds(ids);

if(setmealIds!=null&&setmealIds.size()>0){

throw new DeletionNotAllowedException(MessageConstant.DISH\_BE\_RELATED\_BY\_SETMEAL);

}

//删除菜品表中的菜品数据

for (Long id : ids) {

dishMapper.deleteById(id);

//删除菜品表中关联的口味数据 直接删除 不需要查询

dishFlavorMapper.deleteByDishId(id);

}

}

有三个Mapper 分别是dishMapper删除菜品数据 查询菜品是否起售 dishFlavorMapper删除与之关联的口味数据 setmealDishMapper查询与之关联的套餐ID

/\*

\*根据主键查询菜品

\* \*/

@Select("select \*from dish where id=#{id}")

Dish getById(Long id);

/\*

\*根据主键删除菜品数据

\* \*/

@Delete("delete dish from dish where id=#{id}")

void deleteById(Long id);

/\*

\*根据菜品ID来删除对应的口味数据

\* \*/

@Delete("delete from dish\_flavor where dish\_id=#{dishId}")

void deleteByDishId(Long dishID);

/\*

\* 根据菜品ID来查询多个套餐ID

\* \*/

List<Long> getSetmealIdsByDishIds(List<Long> dishIds);

**功能优化**

使用SQL语句数量过多 采用一次性批量删除

//删除菜品表中的菜品数据 原本代码效率较低 多次执行SQL 可以一次性批量删除 提高效率

// for (Long id : ids) {

// dishMapper.deleteById(id);

// //删除菜品表中关联的口味数据 直接删除 不需要查询

// dishFlavorMapper.deleteByDishId(id);

// }

//根据菜品ID集合批量删除菜品数据

dishMapper.deleteByIds(ids);

//根据菜品ID集合批量删除口味数据

dishFlavorMapper.deleteByDishIds(ids);

xml映射文件中配置差不多

<delete id="deleteByIds">

delete from dish where id in

<foreach collection="ids" open="(" close=")" separator="," item="id">

#{id}

</foreach>

</delete>

**修改菜品**

先查询回显 再存储到数据库

**菜品回显**

页面查询数据 用于回显 Controller层

/\*

\* 根据ID查询菜品数据 用于页面回显

\* \*/

@GetMapping("/{id}")

@ApiOperation("根据ID查询菜品")

public Result<DishVO> getById(@PathVariable Long id){

log.info("根据ID查询菜品：{}",id);

DishVO dishVo=dishService.getIdWithFlavor(id);

return Result.success(dishVo);

}

Service层

/\*

\* 根据ID查询对应的菜品和口味数据

\* \*/

@Override

public DishVO getIdWithFlavor(Long id) {

//根据ID查询菜品数据

Dish dish = dishMapper.getById(id);

//根据ID查询口味数据

List<DishFlavor> dishFlavors=dishFlavorMapper.getByDishId(id);

//封装查询到的数据到VO

DishVO dishVO=new DishVO();

BeanUtils.copyProperties(dish,dishVO);

dishVO.setFlavors(dishFlavors);

return dishVO;

}

增加了一个查询口味的Mapper 其余均已有

/\*

\* 根据菜品ID查询口味数据

\* \*/

@Select("select \*from dish\_flavor where dish\_id=#{dishId}")

List<DishFlavor> getByDishId(Long dishId);

**菜品更新**

Controller层

/\*

\* 修改菜品

\* \*/

@PutMapping

@ApiOperation("修改菜品")

public Result update(@RequestBody DishDTO dishDTO){

log.info("修改菜品：{}",dishDTO);

dishService.updateWithFlavor(dishDTO);

return Result.success();

}

Service层

/\*

\* 根据ID修改菜品信息

\* \*/

@Override

public void updateWithFlavor(DishDTO dishDTO) {

Dish dish = new Dish();

BeanUtils.copyProperties(dishDTO,dish);

//删除菜品表的基本信息

dishMapper.update(dish);

//删除关联的口味表信息

dishFlavorMapper.deleteByDishId(dish.getId());

//插入新的口味表信息

List<DishFlavor> flavors=dishDTO.getFlavors();

if (flavors!=null&&flavors.size()>0){

flavors.forEach(dishFlavor -> {

dishFlavor.setDishId(dishDTO.getId());

});

dishFlavorMapper.insertBatch(flavors);

}

}

Mapper xml映射

<update id="update">

update dish

<set>

<if test="name!=null">

name=#{name},

</if>

<if test="categoryId!=null">

categoryId=#{categoryId},

</if>

<if test="price!=null">

price=#{price},

</if>

<if test="image!=null">

image=#{image},

</if>

<if test="description!=null">

description=#{description},

</if>

<if test="updateUser!=null">

update\_user=#{updateUser},

</if>

<if test="updateTime!=null">

update\_time=#{updateTime},

</if>

</set>

where id=#{id}

</update>

**Day5**

**Redis入门**

Redis简介：Redis是一个基于**内存**的key- value结构数据库 MySql是通过二维表来存储的 Redis是通过键值对存储

为什么要学习Redis？

因为Redis基于内存存储，读写性能高/ 内存有限，适合存储热点数据（热点商品，资讯）/

**Redis使用指令**

Redis 默认没有密码 要想设置密码 要去redis.conf 中修改requirepass 123456 再次连接客户端需要使用 redis-cli -h localhost -p 6379 -a 123456

安装Redis 使用 brew install redis

启动Redis 使用 brew services start redis

关闭Redis 使用 brew services stop redis

连接Redis客户端 使用 redis-cli

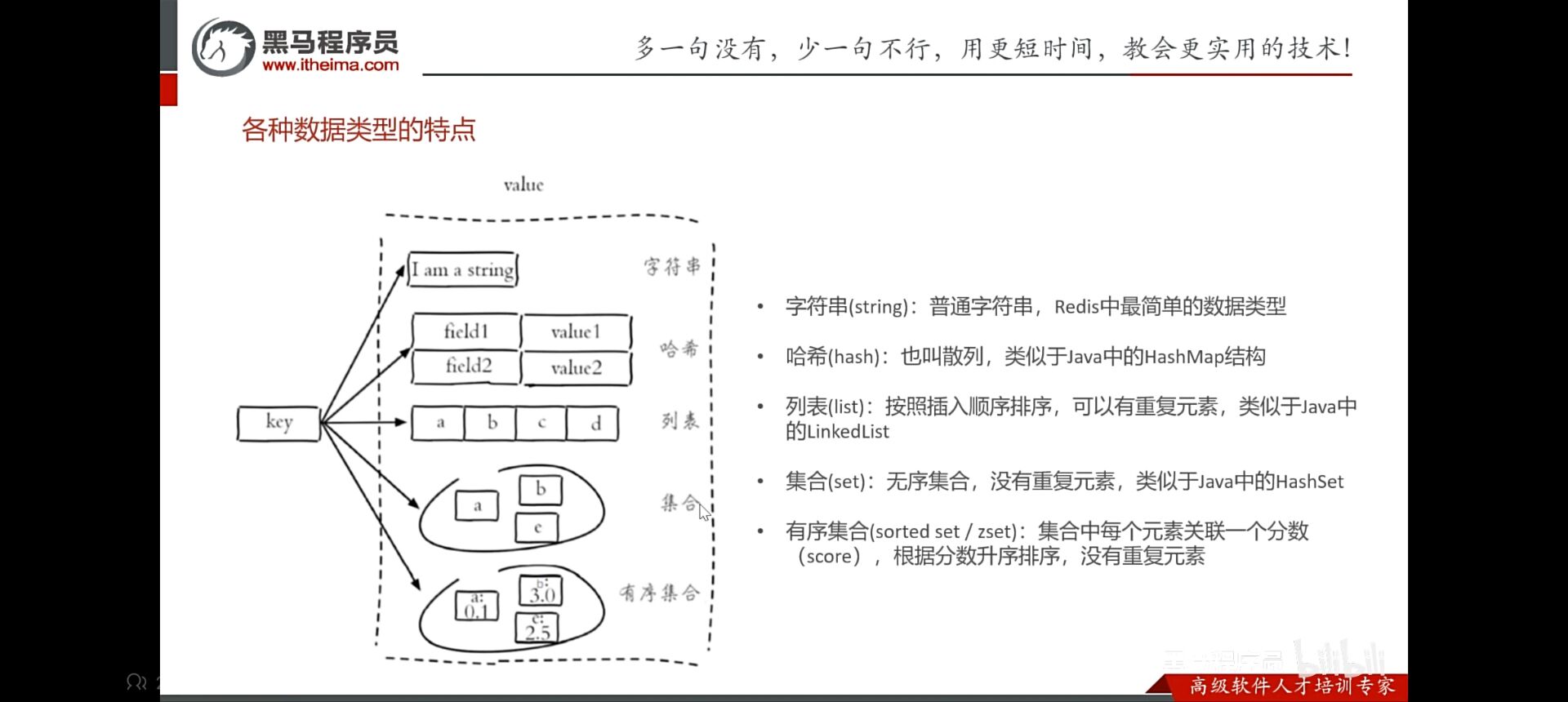
查询数据库 使用 key \*

连接特定客户端 使用 redis-cli -h localhost -p 6379

退出 使用 exit

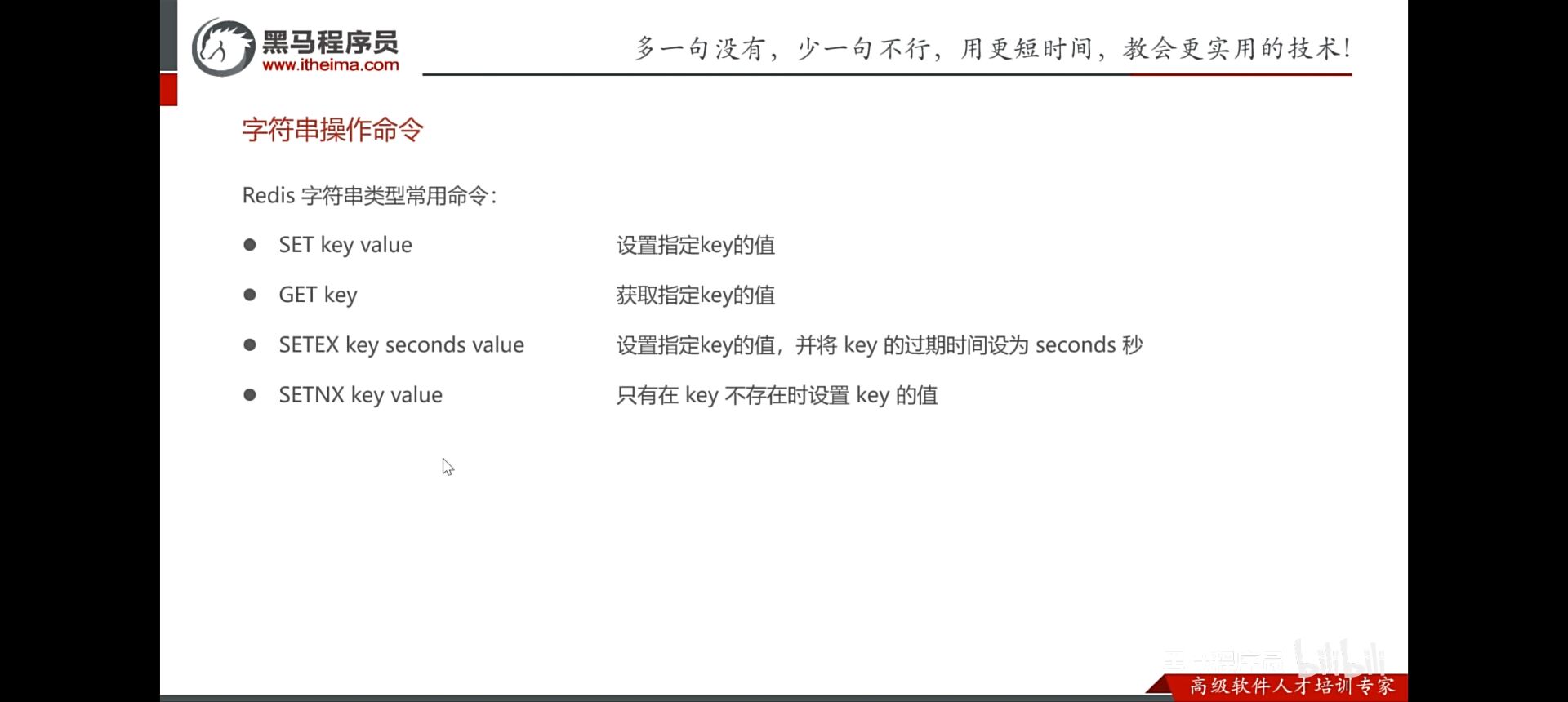
安装Redis客户端 brew install --cask another-redis-desktop-manager

***Redis中五种数据类型***

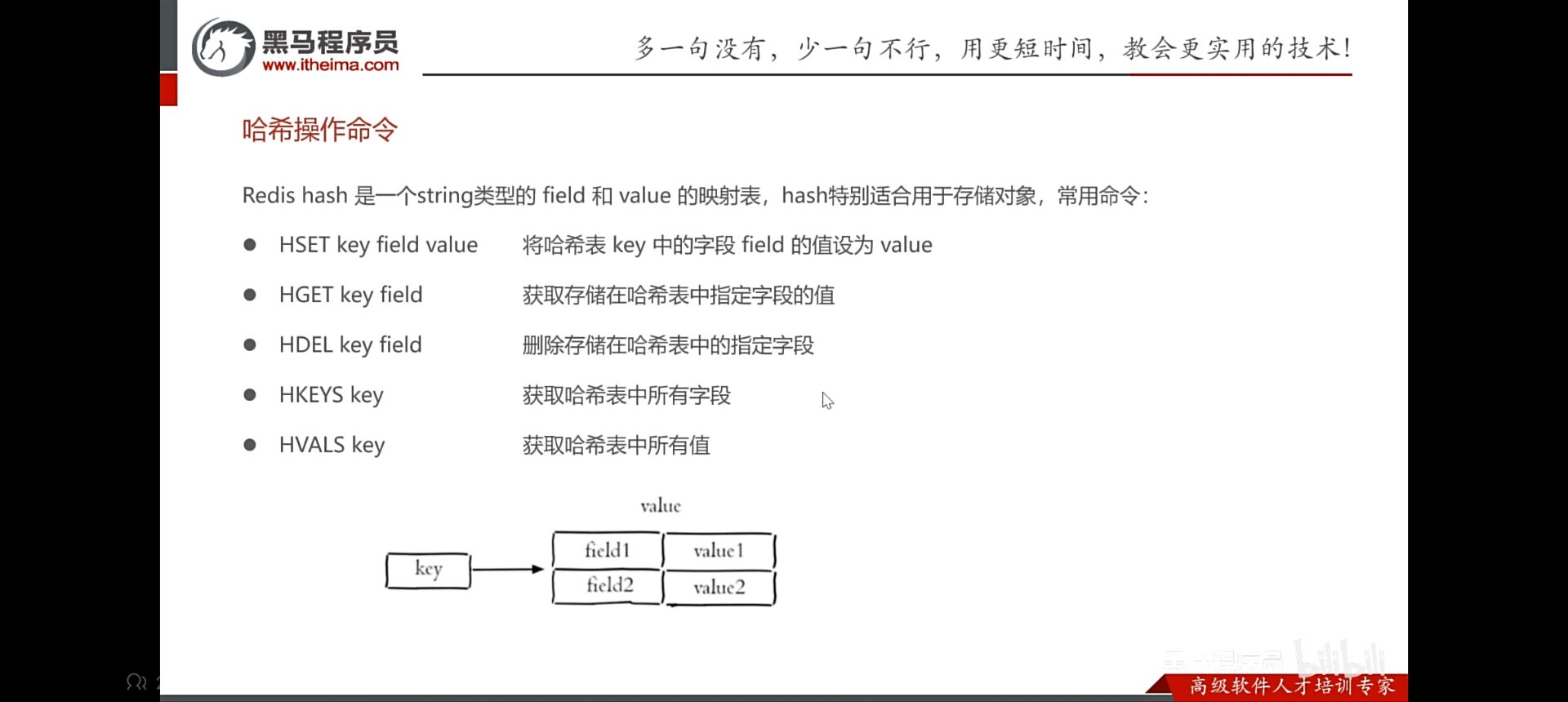


**Redis常用命令**

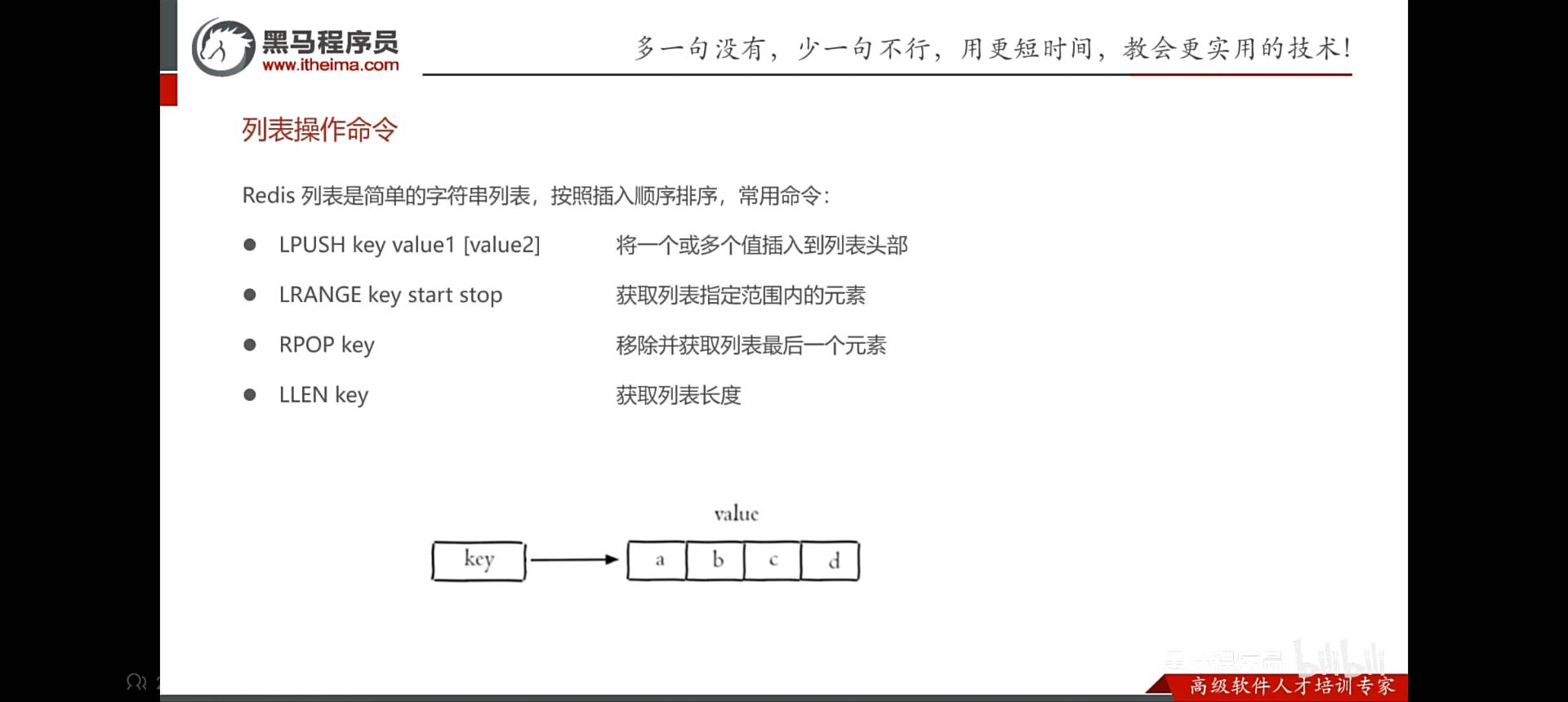
**字符串操作命令**



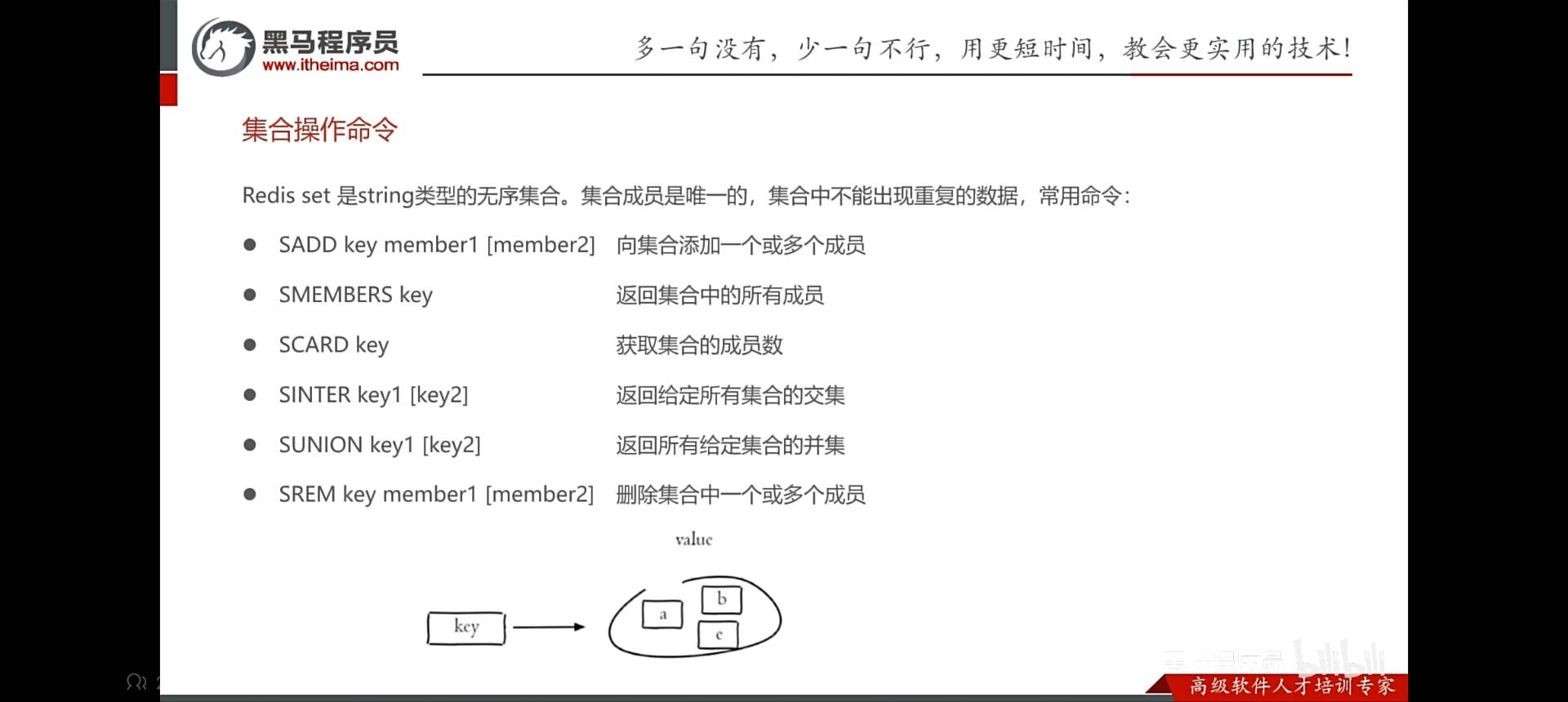
**哈希操作命令**



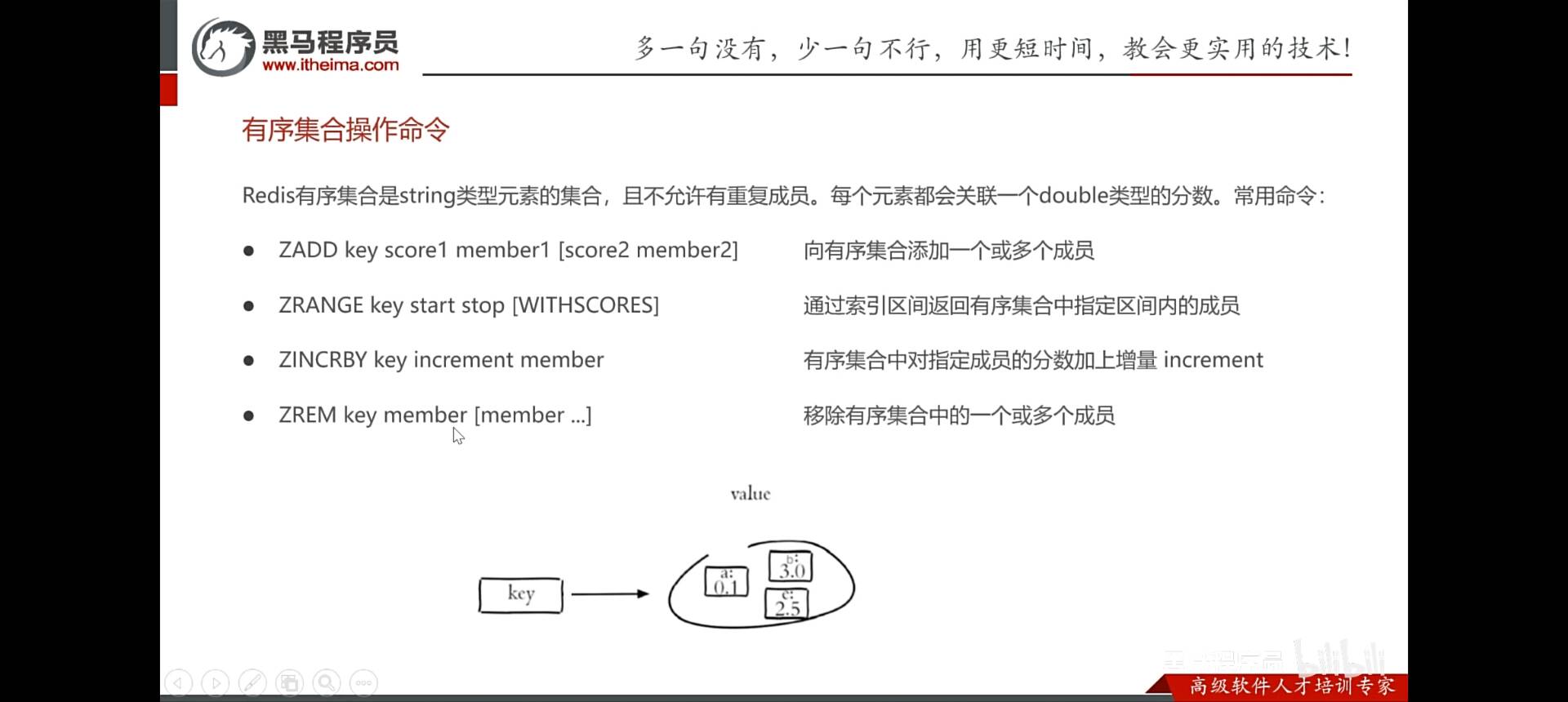
**列表操作命令**



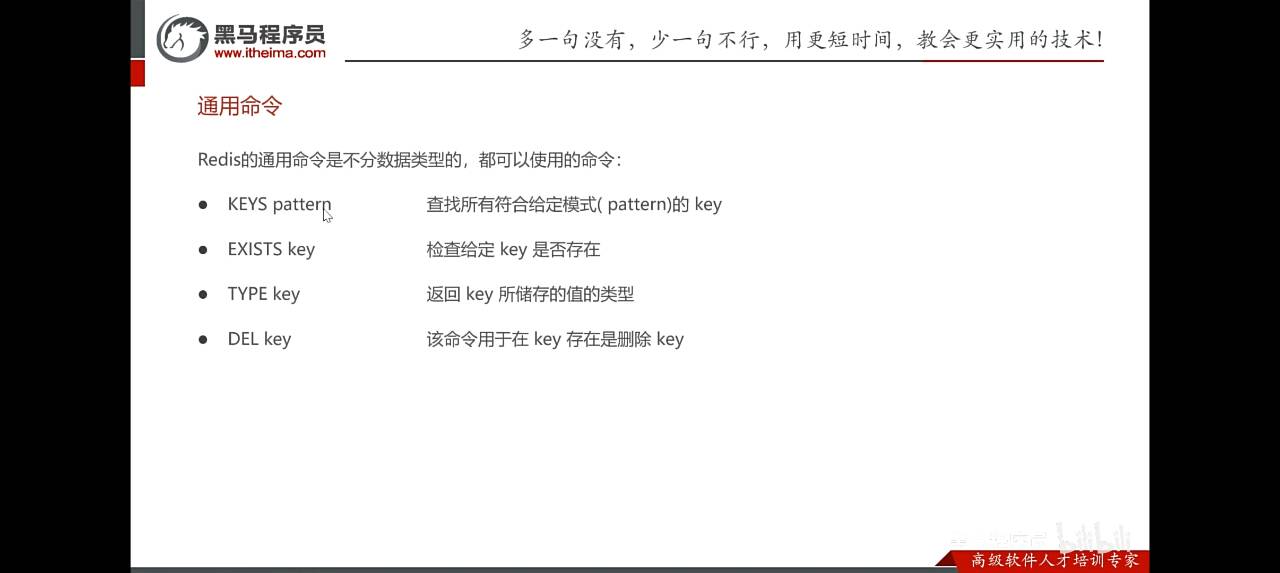
**集合操作命令**



**有序集合操作命令**



**通用命令**



**在JAVA中操作Redis**

Redis的JAVA客户端：jedis Lettuce Spring Data Redis（主要使用）

**Spring Data Redis使用方式:**

1、导入Spring Data Redis的Maven坐标

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

2、配置Redis数据源(注意yml格式，要有空格）

#redis配置数据源

redis:

host: ${sky.redis.host}

port: ${sky.redis.port}

password: ${sky.redis.password}

database: ${sky.redis.database}

3、编写配置类，创建RedisTemplate对象

@Configuration

@Slf4j

public class RedisConfiguration {

@Bean

public RedisTemplate redisTemplate(RedisConnectionFactory redisConnectionFactory){

log.info("开始设置reids模板对象......");

RedisTemplate redisTemplate = new RedisTemplate();

//设置Redis的连接工厂对象

redisTemplate.setConnectionFactory(redisConnectionFactory);

//设置Redis序列化器

redisTemplate.setKeySerializer(new StringRedisSerializer());

return redisTemplate;

}

}

4、通过RedisTemplate对象操作Redis

@SpringBootTest

public class SpringDataRedisTest {

@Autowired

private RedisTemplate redisTemplate;

@Test

public void testRedisTemplate(){

System.out.println(redisTemplate);

ValueOperations valueOperations = redisTemplate.opsForValue();//字符串对象

HashOperations hashOperations = redisTemplate.opsForHash();//Hash列表对象

ListOperations listOperations = redisTemplate.opsForList();//List集合对象

SetOperations setOperations = redisTemplate.opsForSet();//Set集合对象

ZSetOperations zSetOperations = redisTemplate.opsForZSet();//ZSet集合对象

}

/\*

\* 操作字符串类型的数据

\* \*/

@Test

public void testString(){

//set get setex setnx

redisTemplate.opsForValue().set("name","小明");

String city = (String) redisTemplate.opsForValue().get("name");

System.out.println(city);

//最后一个参数代表时间单位 3分钟

redisTemplate.opsForValue().set("code","1234",3, TimeUnit.MINUTES);

redisTemplate.opsForValue().setIfAbsent("lock","1");

redisTemplate.opsForValue().setIfAbsent("lock","2");

}

**店铺营业状态设置**

营业状态值存储方式：基于Redis的字符串存储 1：营业 0：打烊

只在Controller中 直接通过RedisTemplate调用Redis(一个为用户端，一个为管理端)

@RestController("adminShopController")

@RequestMapping("/admin/shop")

@Api(tags = "店铺相关接口")

@Slf4j

public class ShopController {

private static final String KEY="SHOP\_STATUS";

@Autowired

private RedisTemplate redisTemplate;

/\*

\* 设置店铺营业状态

\* \*/

@PutMapping("/{status}")

@ApiOperation("设置店铺营业状态")

public Result setStatus(@PathVariable Integer status){

log.info("设置的店铺状态为：{}",status==1?"营业中":"打烊中");

redisTemplate.opsForValue().set(KEY,status);

return Result.success();

}

/\*

\* 获取店铺营业状态

\* \*/

@GetMapping("/status")

@ApiOperation("获取店铺营业状态")

public Result<Integer> getStatus(){

Integer status = (Integer) redisTemplate.opsForValue().get(KEY);

log.info("获取的店铺状态为：{}",status==1?"营业中":"打烊中");

return Result.success(status);

}

}

用户端的店铺相关状态接口

@RestController("userShopController")

@RequestMapping("/user/shop")

@Api(tags = "店铺相关接口")

@Slf4j

public class ShopController {

private static final String KEY="SHOP\_STATUS";

@Autowired

private RedisTemplate redisTemplate;

/\*

\* 获取店铺营业状态

\* \*/

@GetMapping("/status")

@ApiOperation("获取店铺营业状态")

public Result<Integer> getStatus(){

Integer status = (Integer) redisTemplate.opsForValue().get(KEY);

log.info("获取的店铺状态为：{}",status==1?"营业中":"打烊中");

return Result.success(status);

}

}

**Day6**

**HttpClient**

HttpClient是一个客户端编程工具包 发送Http请求 实现微信登录

发送HTTP请求步骤:

创建HttpClient对象----->创建HTTP请求对象（get方式或者post对象）----->调用HttpClient的execute方法



通过单元测试，实现HttpClient入门案例，发送Get方式的请求

/\*

\* 测试通过HttpClient发送Get方式的请求

\* \*/

@Test

public void testGET() throws Exception{

//创建HttpClient对象

CloseableHttpClient httpClient = HttpClients.createDefault();

//创建请求对象

HttpGet httpGet = new HttpGet("http://localhost:8080/user/shop/status");

//发送请求,接收响应结果

CloseableHttpResponse response = httpClient.execute(httpGet);

//获取服务端返回的状态码

int statusCode = response.getStatusLine().getStatusCode();

System.out.println("服务端返回的状态码："+statusCode);

HttpEntity entity=response.getEntity();

String body = EntityUtils.toString(entity);

System.out.println("服务端返回的数据为："+body);

//关闭资源

response.close();

httpClient.close();

}

发送Post方式请求

/\*

\* 测试通过HttpClient发送Post方式的请求

\* \*/

@Test

public void testPOST() throws Exception{

//创建HttpClient对象

CloseableHttpClient httpClient = HttpClients.createDefault();

//创建Post请求对象

HttpPost httpPost = new HttpPost("http://localhost:8080/admin/employee/login");

//通过JSONObject快速生成JSON对象

JSONObject jsonObject=new JSONObject();

jsonObject.put("username","admin");

jsonObject.put("password","123456");

StringEntity entity=new StringEntity(jsonObject.toString());

//指定请求编码方式

entity.setContentEncoding("utf-8");

//指定数据的传输格式

entity.setContentType("application/json");

httpPost.setEntity(entity);

//发送请求

CloseableHttpResponse response = httpClient.execute(httpPost);

//解析返回结果

int statusCode = response.getStatusLine().getStatusCode();

System.out.println("响应码为："+statusCode);

HttpEntity entity1 = response.getEntity();//获得的是entity对象

String body = EntityUtils.toString(entity1);//将entity对象转成String

System.out.println("响应数据为："+body);

//关闭资源

response.close();

httpClient.close();

}

提前在Common中封装了utils方法 可以直接调用其中的HttpClient方法

**微信小程序开发**

**微信登录**

Controller层

@RestController

@RequestMapping("/user/user")

@Api(tags = "c端用户接口")

@Slf4j

public class UserController {

@Autowired

private UserService userService;

@Autowired

private JwtProperties jwtProperties;

/\*

\* 微信登录

\* \*/

@PostMapping ("/login")

@ApiOperation("微信用户登录")

public Result<UserLoginVO> login(@RequestBody UserLoginDTO userLoginDTO){

log.info("微信用户登录：{}",userLoginDTO.getCode());

//微信登录

User user = userService.wxLogin(userLoginDTO);

//为微信用户生成JWT令牌

Map<String, Object> claims=new HashMap<>();

claims.put(JwtClaimsConstant.USER\_ID,user.getId());

String token = JwtUtil.createJWT(jwtProperties.getUserSecretKey(), jwtProperties.getUserTtl(), claims);

UserLoginVO userLoginVO = UserLoginVO.builder()

.id(user.getId())

.openid(user.getOpenid())

.token(token)

.build();

return Result.success(userLoginVO);

}

}

Service层

@Service

@Slf4j

public class UserServiceImpl implements UserService {

//微信服务接口地址

private static final String WX\_LOGIN="https://api.weixin.qq.com/sns/jscode2session";

@Autowired

private WeChatProperties weChatProperties;

@Autowired

private UserMapper userMapper;

/\*

\* 微信登录

\* \*/

@Override

public User wxLogin(UserLoginDTO userLoginDTO) {

String openid = getOpenid(userLoginDTO.getCode());

//判断openID是否为空，如果为空则登录失败，抛出业务异常

if(openid==null){

throw new LoginFailedException(MessageConstant.LOGIN\_FAILED);

}

//判断当前用户是否为新用户

User user=userMapper.getByOpenid(openid);

//如果是新用户自动完成注册

if(user==null){

user=User.builder()

.openid(openid)

.createTime(LocalDateTime.now())

.build();

userMapper.insert(user);

}

//返回这个用户对象

return user;

}

/\*

\* 调用微信接口服务，获取微信用户的openid

\* \*/

private String getOpenid(String code){

//调用微信接口，获取当前微信用户的openID

Map<String, String> map=new HashMap<>();

map.put("appid",weChatProperties.getAppid());

map.put("secret",weChatProperties.getSecret());

map.put("js\_code", code);

map.put("grant\_type","authorization\_code");

String json = HttpClientUtil.doGet(WX\_LOGIN, map);

JSONObject jsonObject = JSON.parseObject(json);

String openid = jsonObject.getString("openid");

return openid;

}

}

Mapper层

@Mapper

public interface UserMapper {

/\*

\* 根据openID查询用户

\* \*/

@Select("select \* from sky\_take\_out.user where openid=#{openid}")

User getByOpenid(String openid);

/\*

\* 插入数据

\* \*/

void insert(User user);

}

微信小程序需要token校验 与管理端拦截器差不多

/\*\*

\* jwt令牌校验的拦截器

\*/

@Component

@Slf4j

public class JwtTokenUserInterceptor implements HandlerInterceptor {

@Autowired

private JwtProperties jwtProperties;

/\*\*

\* 校验jwt

\*

\* @param request

\* @param response

\* @param handler

\* @return

\* @throws Exception

\*/

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

//判断当前拦截到的是Controller的方法还是其他资源

if (!(handler instanceof HandlerMethod)) {

//当前拦截到的不是动态方法，直接放行

return true;

}

//1、从请求头中获取令牌

String token = request.getHeader(jwtProperties.getUserTokenName());

//2、校验令牌

try {

log.info("jwt校验:{}", token);

Claims claims = JwtUtil.parseJWT(jwtProperties.getUserSecretKey(), token);

Long userId = Long.valueOf(claims.get(JwtClaimsConstant.USER\_ID).toString());

log.info("当前用户id：", userId);

BaseContext.setCurrentId(userId);

//3、通过，放行

return true;

} catch (Exception ex) {

//4、不通过，响应401状态码

response.setStatus(401);

return false;

}

}

}

记得在config配置文件中配值自定义拦截器

@Autowired

private JwtTokenUserInterceptor jwtTokenUserInterceptor;

/\*\*

\* 注册自定义拦截器

\*

\* @param registry

\*/

protected void addInterceptors(InterceptorRegistry registry) {

log.info("开始注册自定义拦截器...");

registry.addInterceptor(jwtTokenAdminInterceptor)

.addPathPatterns("/admin/\*\*")

.excludePathPatterns("/admin/employee/login");

registry.addInterceptor(jwtTokenUserInterceptor)

.addPathPatterns("/user/\*\*")

.excludePathPatterns("/user/user/login")

.excludePathPatterns("/user/shop/status");

}

**Day7**

**缓存菜品**

问题所在：小程序端的查询访问压力较大 使用缓存机制即Redis来缓存菜品数据

对根据ID查询菜品进行修改 即使用redis缓存

@GetMapping("/list")

@ApiOperation("根据分类id查询菜品")

public Result<List<DishVO>> list(Long categoryId) {

//构造redis中的key，规则：dish\_分类id

String key="dish\_"+categoryId;

//查询redis是否存在菜品数据

List<DishVO> list = (List<DishVO>) redisTemplate.opsForValue().get(key);

if(list!=null&&list.size()>0){

//如果存在直接返回，无需查询数据库

return Result.success(list);

}

Dish dish = new Dish();

dish.setCategoryId(categoryId);

dish.setStatus(StatusConstant.ENABLE);//查询起售中的菜品

//如果不存在，查询数据库，将查询到的数据放入redis

list = dishService.listWithFlavor(dish);

//查询到之后放入到redis中

redisTemplate.opsForValue().set(key,list);

return Result.success(list);

}

引入问题：在管理端修改了数据之后，会导致小程序端依然用的是redis缓存中的数据，无法实现数据的一致性

不止是修改数据 还有删除数据 新增数据 起售停售菜品均会导致小程序端显示的数据与数据库不一致。

解决方案：定义了一个公用的方法 根据传入的pattern来精确删除 或者全部直接删除

/\*

\* 统一清理缓存数据

\* \*/

private void cleanCache(String pattern){

Set keys = redisTemplate.keys(pattern);

redisTemplate.delete(keys);

}

全部删除

/\*

\* 修改菜品

\* \*/

@PutMapping

@ApiOperation("修改菜品")

public Result update(@RequestBody DishDTO dishDTO){

log.info("修改菜品：{}",dishDTO);

dishService.updateWithFlavor(dishDTO);

//将所有的菜品的缓存数据都删除，所有以dish\_ID开头的

cleanCache("dish\_\*");

return Result.success();

}

精确删除

/\*

\* 新增菜品

\* \*/

@PostMapping

@ApiOperation("新增菜品")

public Result save(@RequestBody DishDTO dishDTO){

log.info("新增菜品:{}",dishDTO);

dishService.saveWithFlavor(dishDTO);

//精确清理缓存数据

String key="dish\_"+dishDTO.getCategoryId();

cleanCache(key);

return Result.success();

}

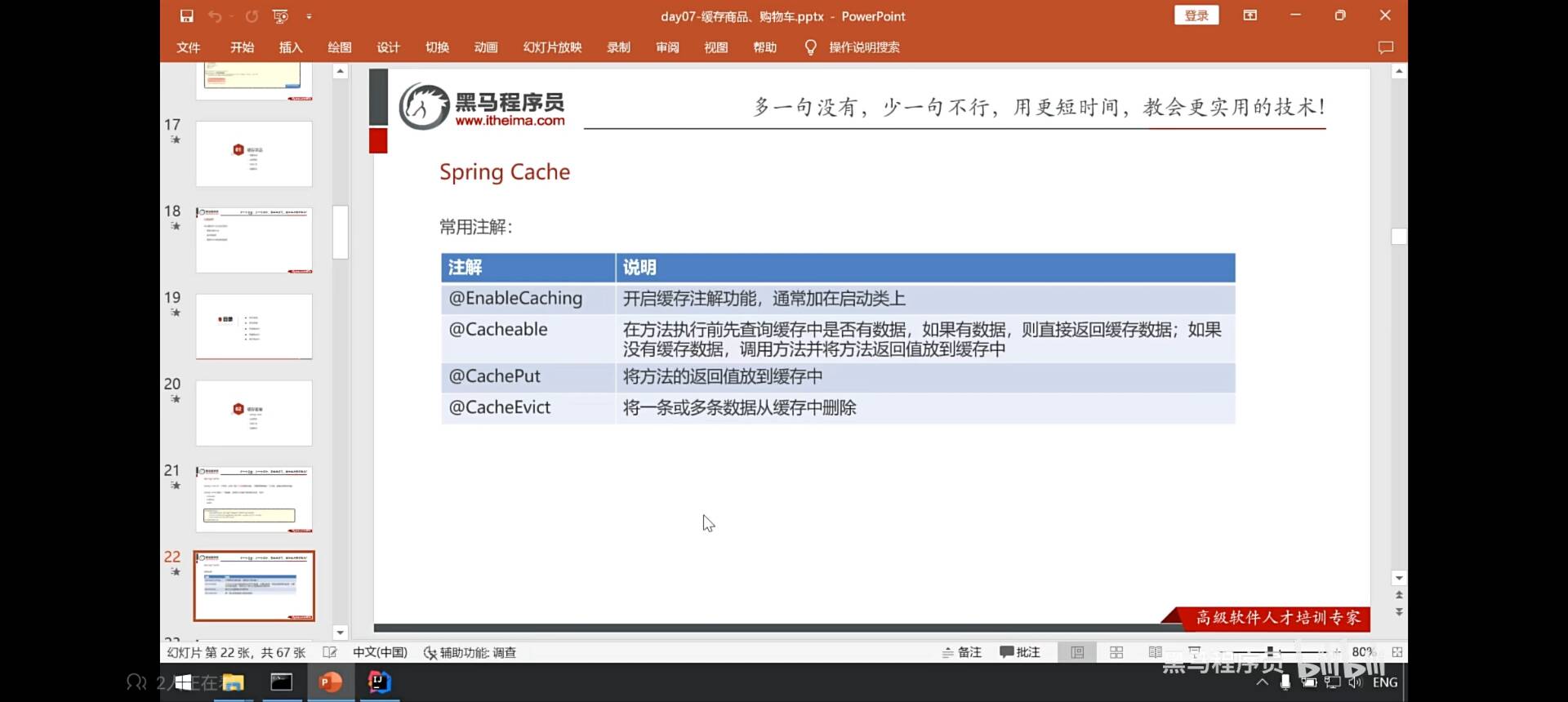
**缓存套餐**

**Spring Cache**

Spring Cache 是一个框架，实现了基于注解的缓存功能，只需要加一个简单的注解就能够实现缓存功能

Spring Cache 只是一层抽象，底层可以切换不同的缓存来实现，例如：EHCache Caffeine Redis

使用Spring Cache 要导入maven文件



Spring-Cache-Demo中有入门案例

注解EnableCaching是要加在启动类上面的 注解CachePut有五种写法 如下 一般第一种

@CachePut(cacheNames = "userCache",key = "#user.id") //如果使用Spring Cache缓存数据。key的生成：userCache::+user.id 实现动态命名

//@CachePut(cacheNames = "userCache",key = "#result.id") //通过result获得返回值user

//@CachePut(cacheNames = "userCache",key = "#p0.id") //#p0代表第一个参数 即user #p1代表第二个参数 以此类推

//@CachePut(cacheNames = "userCache",key = "#a0.id") //用a代表p也是可以的

//@CachePut(cacheNames = "userCache",key = "#root.args[0].id") //root.args代表参数数组

public User save(@RequestBody User user){

userMapper.insert(user);

return user;

}

注解Cacheable是先查询redis缓存是否有数据 如果有就返回结果 不用去查询数据库 没查到就要去查数据库 如下

@Cacheable(cacheNames = "userCache",key = "#id") //key的生成：userCache::id

public void deleteById(Long id){

userMapper.deleteById(id);

}

注解CacheEvict在删除数据库数据的同时删除redis中的缓存数据

@CacheEvict(cacheNames = "userCache",key="#id") //key生成：userCache::id

public void deleteById(Long id){

userMapper.deleteById(id);

}

删除所有数据加上一个allEntries=ture

@CacheEvict(cacheNames = "userCache",allEntries = true)//allEntries代表所有的键值对

public void deleteAll(){

userMapper.deleteAll();

}

**添加购物车**

Controller层

@RestController

@RequestMapping("user/shoppingCart")

@Slf4j

@Api(tags = "C端购物车接口")

public class ShoppingCartController {

@Autowired

private ShoppingCartService shoppingCartService;

/\*

\* 添加购物车

\* \*/

@PostMapping("/add")

@ApiOperation("添加购物车")

public Result add(@RequestBody ShoppingCartDTO shoppingCartDTO){

log.info("添加购物车：商品信息为：{}",shoppingCartDTO);

shoppingCartService.addShoppingCart(shoppingCartDTO);

return Result.success();

}

}

Service层

@Service

@Slf4j

public class ShoppingCartServiceImpl implements ShoppingCartService {

@Autowired

private ShoppingCartMapper shoppingCartMapper;

@Autowired

private DishMapper dishMapper;

@Autowired

private SetmealMapper setmealMapper;

/\*

\* 添加购物车

\* \*/

@Override

public void addShoppingCart(ShoppingCartDTO shoppingCartDTO) {

//判断当前加入购物车的商品是否已经存在了

ShoppingCart shoppingCart=new ShoppingCart();

BeanUtils.copyProperties(shoppingCartDTO,shoppingCart);

Long userId = BaseContext.getCurrentId();

shoppingCart.setUserId(userId);

List<ShoppingCart> list = shoppingCartMapper.list(shoppingCart);

//如果已经存在了，商品数量加一

if(list!=null&&list.size()>0){

ShoppingCart cart = list.get(0);

cart.setNumber(cart.getNumber()+1);

shoppingCartMapper.updateNumberById(cart);

}else {

//如果不存在，再插入一条购物车数据

//判断本次添加到购物车的是菜品还是套餐

Long dishId = shoppingCartDTO.getDishId();

if(dishId!=null){

//本次添加到购物车的是菜品

Dish dish = dishMapper.getById(dishId);

shoppingCart.setName(dish.getName());

shoppingCart.setImage(dish.getImage());

shoppingCart.setAmount(dish.getPrice());

}else {

//本次添加到购物车的套餐

Long setmealId = shoppingCartDTO.getSetmealId();

Setmeal setmeal = setmealMapper.getById(setmealId);

shoppingCart.setName(setmeal.getName());

shoppingCart.setImage(setmeal.getImage());

shoppingCart.setAmount(setmeal.getPrice());

}

shoppingCart.setNumber(1);

shoppingCart.setCreateTime(LocalDateTime.now());

shoppingCartMapper.insert(shoppingCart);

}

}

}

Mapper层

@Mapper

public interface ShoppingCartMapper {

/\*

\* 动态条件查询

\* \*/

List<ShoppingCart>list(ShoppingCart shoppingCart);

/\*

\* 根据ID修改商品数量

\* \*/

@Update("update shopping\_cart set number=#{number} where id=#{id}")

void updateNumberById(ShoppingCart shoppingCart);

/\*

\* 插入购物车数据

\* \*/

@Insert("insert into shopping\_cart(name, user\_id, dish\_id, setmeal\_id, dish\_flavor,number, amount,image, create\_time) " +

"values(#{name},#{userId},#{dishId},#{setmealId},#{dishFlavor},#{number},#{amount},#{image},#{createTime})")

void insert(ShoppingCart shoppingCart);

}

**查看购物车**

Controller层

/\*

\* 查看购物车

\* \*/

@GetMapping("/list")

@ApiOperation("查看购物车")

public Result<List<ShoppingCart>> list(){

List<ShoppingCart> list=shoppingCartService.showShoppingCart();

return Result.success(list);

}

Service层

/\*

\* 查看购物车

\* \*/

@Override

public List<ShoppingCart> showShoppingCart() {

//获取当前微信用户的ID

Long userId = BaseContext.getCurrentId();

ShoppingCart shoppingCart = ShoppingCart.builder()

.userId(userId)

.build();

List<ShoppingCart> list = shoppingCartMapper.list(shoppingCart);

return list;

}

**清空购物车**

Controller层

/\*

\* 清空购物车

\* \*/

@DeleteMapping("/clean")

@ApiOperation("清空购物车")

public Result clean(){

shoppingCartService.cleanShoppingCart();

return Result.success();

}

Service层

/\*

\* 清空购物车

\* \*/

@Override

public void cleanShoppingCart() {

//获取当前微信用户的ID

Long userId = BaseContext.getCurrentId();

shoppingCartMapper.deleteByUserId(userId);

}

Mapper层

/\*

\* 根据userID删除购物车数据

\* \*/

@Delete("delete from shopping\_cart where user\_id=#{userId}")

void deleteByUserId(Long userId);

**Day8**

地址簿功能直接导入

**用户下单**

Controller层

@RestController("userOrderController")

@RequestMapping("/user/order")

@Api(tags = "用户端相关接口")

@Slf4j

public class OrderController {

@Autowired

private OrderService orderService;

/\*

\* 用户下单

\* \*/

@PostMapping("/submit")

@ApiOperation("用户下单")

public Result<OrderSubmitVO> submit(@RequestBody OrdersSubmitDTO ordersSubmitDTO){

log.info("用户下单参数为：{}",ordersSubmitDTO);

OrderSubmitVO orderSubmitVO= orderService.submitOrder(ordersSubmitDTO);

return Result.success(orderSubmitVO);

}

}

Service层

@Autowired

private OrderMapper orderMapper;

@Autowired

private OrderDetailMapper orderDetailMapper;

@Autowired

private AddressBookMapper addressBookMapper;

@Autowired

private ShoppingCartMapper shoppingCartMapper;

/\*

\* 用户下单

\* \*/

@Override

@Transactional

public OrderSubmitVO submitOrder(OrdersSubmitDTO ordersSubmitDTO) {

//1.处理各种业务异常情况（购物车为空或者地址簿为空）

AddressBook addressBook = addressBookMapper.getById(ordersSubmitDTO.getAddressBookId());

if(addressBook==null){

//抛出业务异常

throw new AddressBookBusinessException(MessageConstant.ADDRESS\_BOOK\_IS\_NULL);

}

//查询当前购物车数据

Long userId = BaseContext.getCurrentId();

ShoppingCart shoppingCart=new ShoppingCart();

shoppingCart.setUserId(userId);

List<ShoppingCart> shoppingCartList = shoppingCartMapper.list(shoppingCart);

if(shoppingCartList==null||shoppingCartList.size()==0){

//抛出业务异常

throw new ShoppingCartBusinessException(MessageConstant.SHOPPING\_CART\_IS\_NULL);

}

//2.向订单表插入1条数据

Orders orders=new Orders();

BeanUtils.copyProperties(ordersSubmitDTO,orders);

orders.setOrderTime(LocalDateTime.now());

orders.setPayStatus(Orders.UN\_PAID);

orders.setStatus(Orders.PENDING\_PAYMENT);

orders.setNumber(String.valueOf(System.currentTimeMillis()));

orders.setPhone(addressBook.getPhone());

orders.setConsignee(addressBook.getConsignee());

orders.setUserId(userId);

orderMapper.insert(orders);

List<OrderDetail> orderDetailList=new ArrayList<>();

//向订单明细表插入n条数据

for (ShoppingCart cart : shoppingCartList) {

OrderDetail orderDetail=new OrderDetail();//订单明细表

BeanUtils.copyProperties(cart,orderDetail);

orderDetail.setOrderId(orders.getId());//关联的订单ID

orderDetailList.add(orderDetail);

}

orderDetailMapper.insertBatch(orderDetailList);

//清空当前用户的购物车数据

shoppingCartMapper.deleteByUserId(userId);

//封装VO返回结束

OrderSubmitVO orderSubmitVO = OrderSubmitVO.builder()

.id(orders.getId())

.orderTime(orders.getOrderTime())

.orderNumber(orders.getNumber())

.orderAmount(orders.getAmount())

.build();

return orderSubmitVO;

}

}

Mapper层 一个是订单表 一个是订单明细表

<insert id="insert" useGeneratedKeys="true" keyProperty="id">

insert into orders

(number, status, user\_id, address\_book\_id, order\_time, checkout\_time, pay\_method,

pay\_status, amount, remark, phone, address, user\_name, consignee, cancel\_reason,

rejection\_reason, cancel\_time, estimated\_delivery\_time, delivery\_status, delivery\_time, pack\_amount,

tableware\_number, tableware\_status)

values

(#{number},#{status},#{userId},#{addressBookId},#{orderTime},#{checkoutTime},#{payMethod},#{payStatus}

,#{amount},#{remark},#{phone},#{address},#{userName},#{consignee},#{cancelReason},#{rejectionReason},

#{cancelTime},#{estimatedDeliveryTime},#{deliveryStatus},#{deliveryTime}

,#{packAmount},#{tablewareNumber},#{tablewareStatus})

</insert>

<insert id="insertBatch">

insert into order\_detail(name, image, order\_id, dish\_id, setmeal\_id, dish\_flavor, number,amount)

VALUES

<foreach collection="orderDetailList" item="od" separator=",">

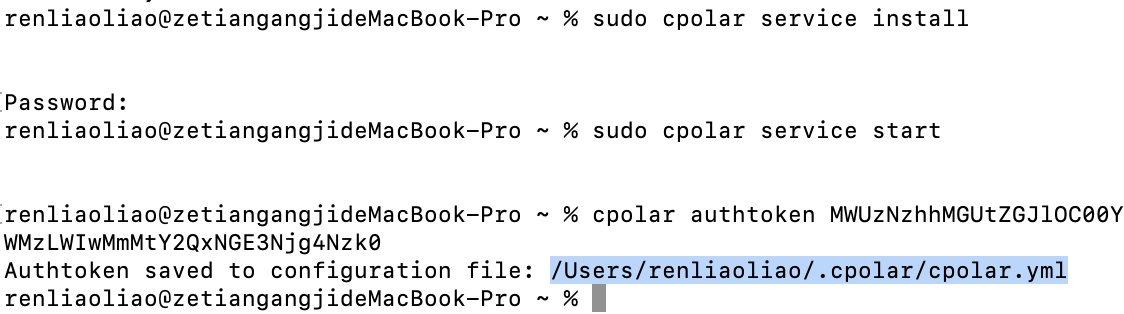
(#{od.name},#{od.image},#{od.orderId},#{od.dishId},#{od.setmealId},#{od.dishFlavor},#{od.number},#{od.amount})

</foreach>

</insert>

**微信支付功能**

**内网穿透**



在局域网中无法实现共用IP地址 故而使用Cpolar实现内网穿透

首先下载Cpolar 我们通过HomeBrew来自动安装Cpolar 确保有HomeBrew

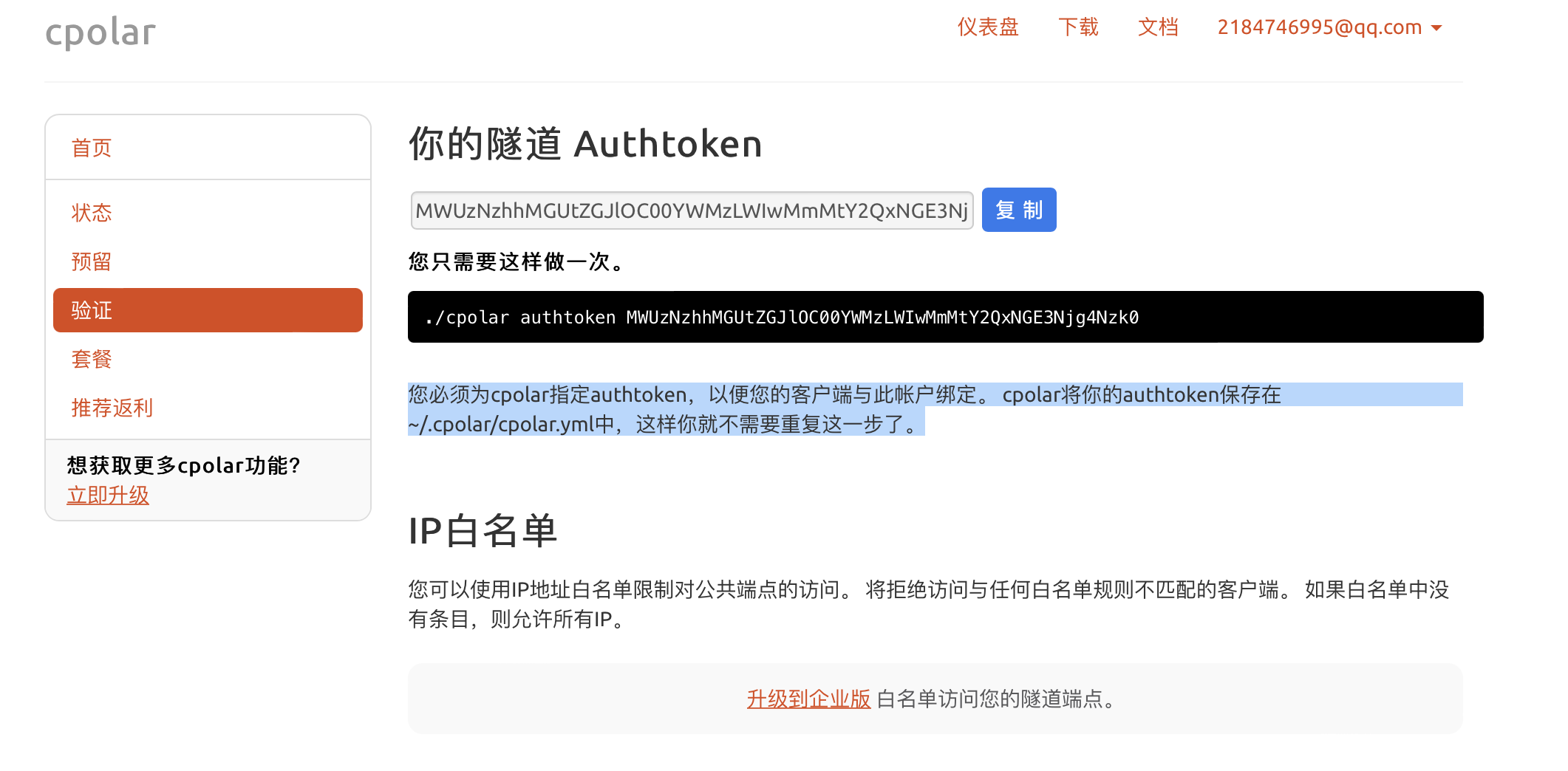
没有的话用以下命令来安装homebrew

/bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)>"

安装好后继续执行命令

brew tap probezy/core && brew install cpolar

使用cpolar命令执行Authtoken



cpolar authtoken xxxxxxx

您必须为cpolar指定authtoken，以便您的客户端与此帐户绑定。 cpolar将你的authtoken保存在~/.cpolar/cpolar.yml中，这样你就不需要重复这一步了。

安装Cpolar服务程序

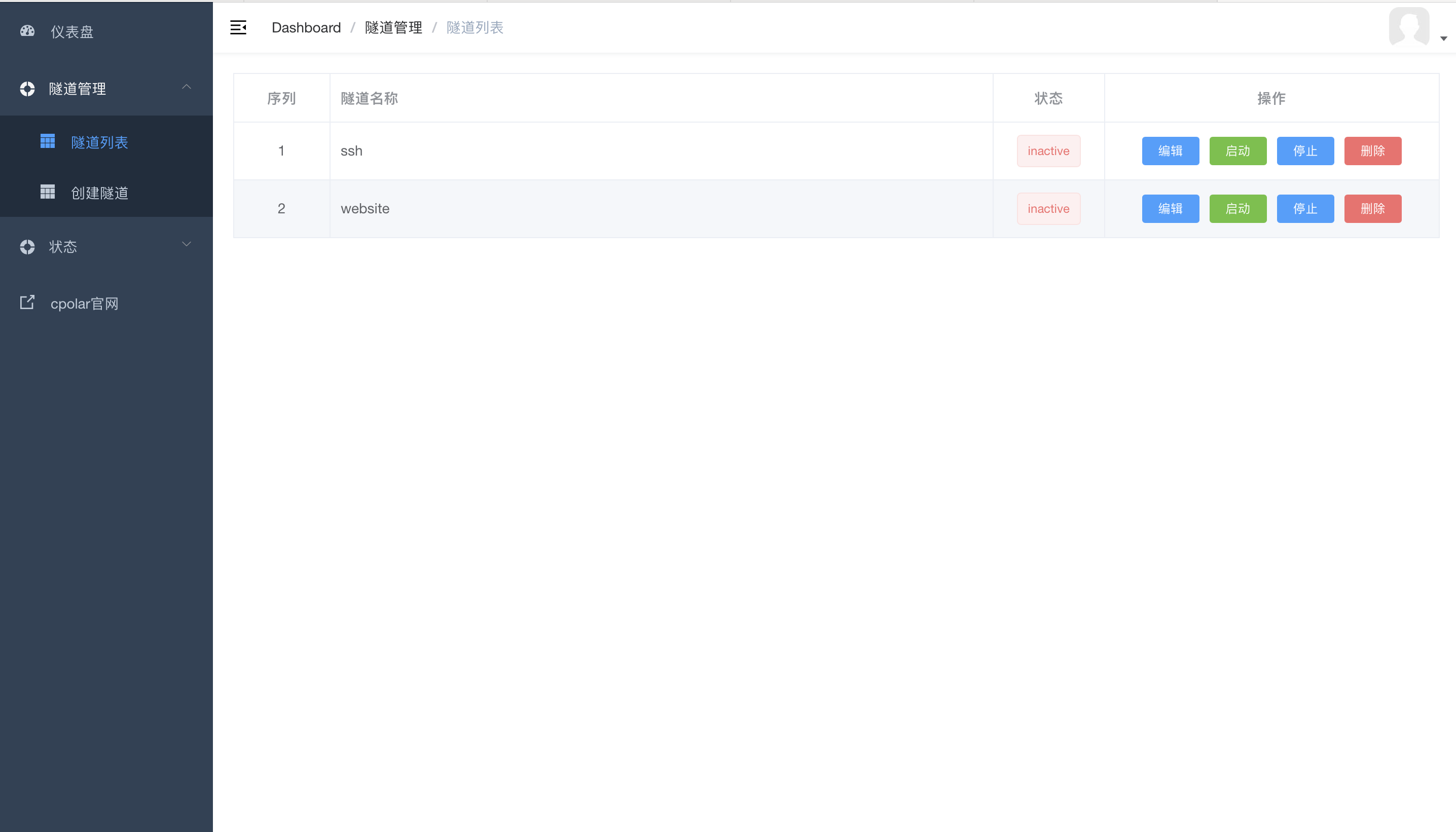
sudo cpolar service install

启动服务

sudo cpolar service start

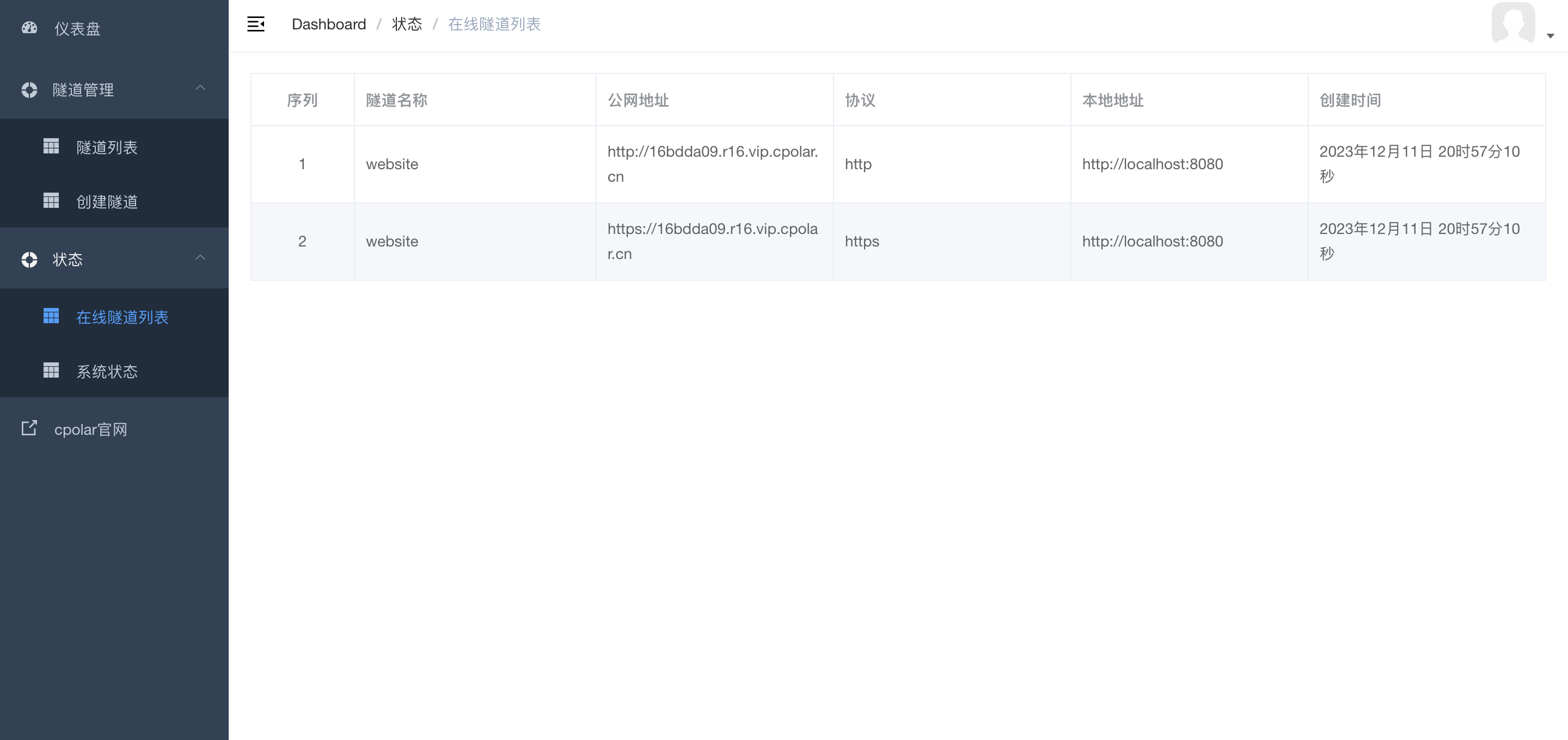
登录cpolar web UI管理界面

在浏览器上访问本地9200端口【<http://127.0.0.1:9200/>】，使用cpolar账号登录。



我们使用的是website 编辑一下（不编辑也行，因为端口默认为8080，正好与后端程序的端口对应）如果不对应的话就需要修改一下。（修改完记得点击更新才行）然后点击启动。

然后点击状态里面的在线隧道列表就可以查看临时的公网IP地址了



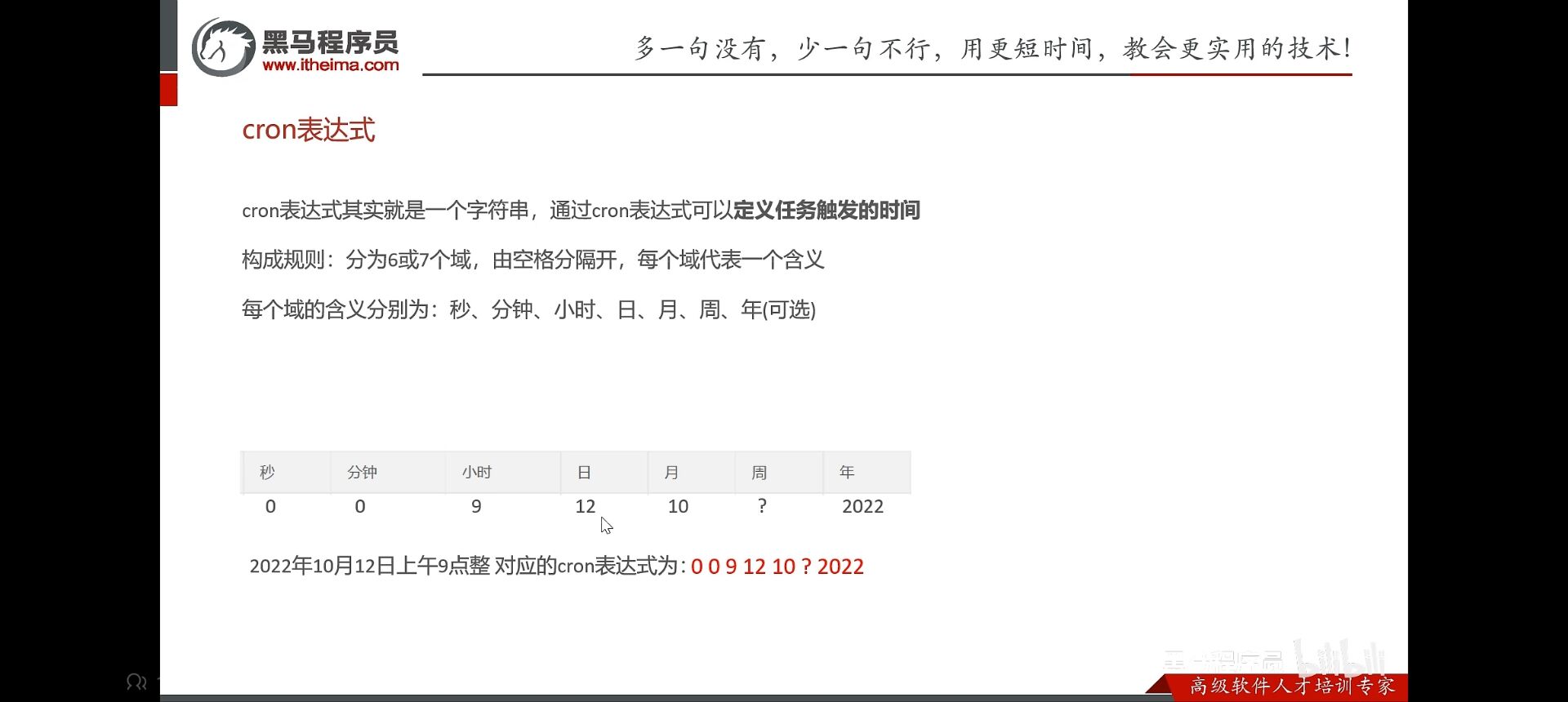
就可以直接使用该地址访问了 复制地址后面再加上/doc.html就可以访问接口文档了

**Day10**

**Sping Task**

一个任务调度工具 通过springboot来自动执行 定时自动执行相关java代码

cron表达式：



**订单状态定时处理**

订单一直处在“待支付”的状态 用户收货之后未点击完成按钮订单一直处在“配送中”

待支付 设置10分钟一次 配送中 每天一次

/\*

\* 定时任务类 定时处理订单状态

\* \*/

@Component

@Slf4j

public class OrderTask {

@Autowired

private OrderMapper orderMapper;

/\*

\* 处理超时订单的方法

\* \*/

@Scheduled(cron = "0 \* \* \* \* ? ")//每分钟触发一次

public void processTimeoutOrder(){

log.info("定时处理超时订单：{}", LocalDateTime.now());

LocalDateTime time = LocalDateTime.now().plusMinutes(-15);

List<Orders> ordersList = orderMapper.getByStatusAndOrderTimeLT(Orders.PENDING\_PAYMENT, time);

if(ordersList!=null&&ordersList.size()>0){

for (Orders orders : ordersList) {

orders.setStatus(Orders.CANCELLED);

orders.setCancelReason("订单超时，自动取消");

orders.setCancelTime(LocalDateTime.now());

orderMapper.update(orders);

}

}

}

/\*

\* 处理配送中订单

\* \*/

@Scheduled(cron = "0 0 1 \* \* ?")//每天凌晨一点触发一次

public void processDeliveryOrder(){

log.info("定时处理一直处于派送中的订单:{}",LocalDateTime.now());

LocalDateTime time = LocalDateTime.now().plusMinutes(-60);

List<Orders> ordersList = orderMapper.getByStatusAndOrderTimeLT(Orders.DELIVERY\_IN\_PROGRESS, time);

if(ordersList!=null&&ordersList.size()>0){

for (Orders orders : ordersList) {

orders.setStatus(Orders.COMPLETED);

orderMapper.update(orders);

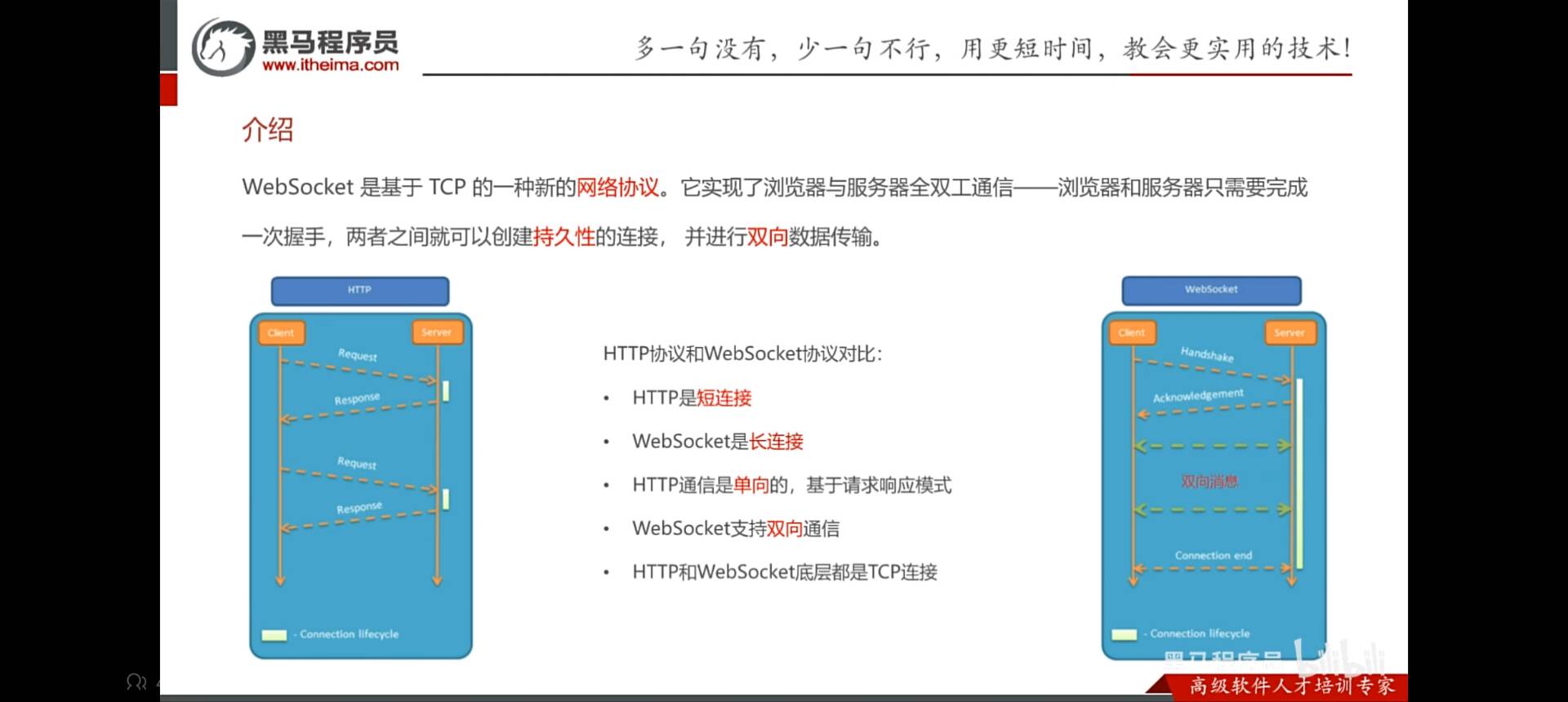
}

}

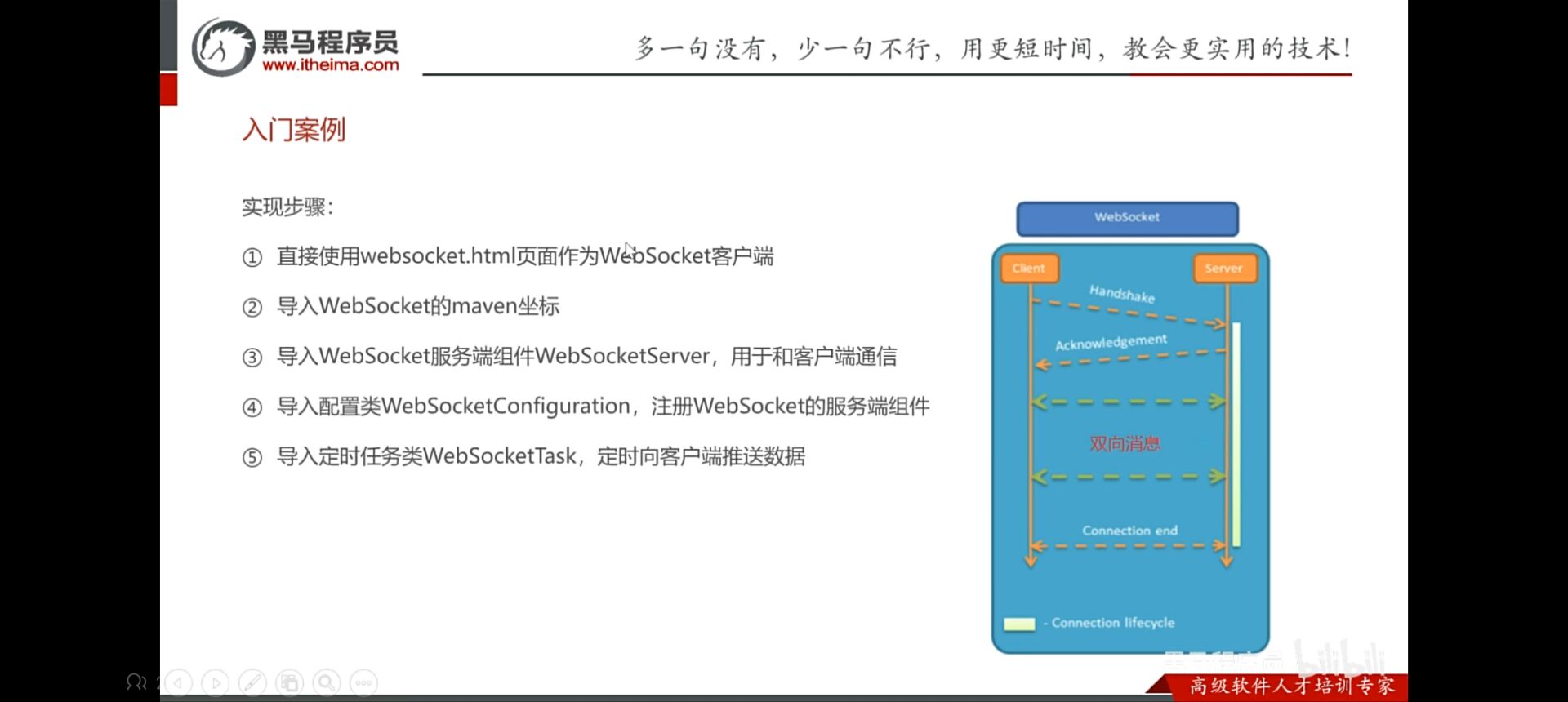
}

}

**WebSocket**



应用场景：视频弹幕 网页聊天 等等



**来单提醒**

用户下单成功后，需要第一时间通知外卖商家，通知有两种：语音弹幕 弹出提示框

消息分为两种：来单提醒 和 客户催单

在paysuccess后面加

//通过websocket向客户端推送数据 type orderId content

Map map=new HashMap<>();

map.put("type",1);//1 表示来电提醒 2表示客户催单

map.put("orderId",ordersDB.getId());

map.put("content","订单号： "+outTradeNo);

String json= JSON.toJSONString(map);

webSocketServer.sendToAllClient(json);

**客户催单**

Controller层

/\*

\* 客户催单

\* \*/

@GetMapping("/reminder/{id}")

@ApiOperation("客户催单")

public Result reminder(@PathVariable("id") Long id){

orderService.reminder(id);

return Result.success();

}

service层

/\*

\* 客户催单

\* \*/

@Override

public void reminder(Long id) {

// 根据id查询订单

Orders ordersDB = orderMapper.getById(id);

// 校验订单是否存在，并且状态为4

if (ordersDB == null) {

throw new OrderBusinessException(MessageConstant.ORDER\_STATUS\_ERROR);

}

Map map=new HashMap<>();

map.put("type",2);//1表示来单提醒 2表示客户催单

map.put("orderId",id);

map.put("content","订单号: "+ordersDB.getId());

//调用webSocket向客户端浏览器发送消息

webSocketServer.sendToAllClient(JSON.toJSONString(map));

}

**Day11**

**Apache ECharts**

是前端的 一款基于javascript的数据可视化图标库

**营业额统计**

Controller层

@Autowired

private ReportService reportService;

/\*

\* 营业额统计

\* \*/

@ApiOperation("营业额统计")

@GetMapping("turnoverStatistics")

public Result<TurnoverReportVO> turnoverStatistics(@DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate begin,@DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate end){

log.info("营业额数据统计：{},{}",begin,end);

return Result.success(reportService.getTurnoverStatistics(begin,end));

}

Service层

@Service

@Slf4j

public class ReportServiceImpl implements ReportService {

/\*

\* 统计指定时间区间内的营业额数据

\* \*/

@Autowired

private OrderMapper orderMapper;

public TurnoverReportVO getTurnoverStatistics(LocalDate begin, LocalDate end) {

//当前集合用于存放从begin到end范围内的每天的日期

List<LocalDate> dataList=new ArrayList<>();

dataList.add(begin);

while(!begin.equals(end)){

//计算指定日期的后一天

begin=begin.plusDays(1);

dataList.add(begin);

}

//存放每天营业额

List<Double> turnoverList=new ArrayList<>();

for (LocalDate date : dataList) {

//查询date日期对应的营业额数据，营业额是指：状态为："已完成"的订单金额合计

LocalDateTime beginTime = LocalDateTime.of(date, LocalTime.MIN);

LocalDateTime endTime = LocalDateTime.of(date, LocalTime.MAX);

Map map=new HashMap<>();

map.put("begin",beginTime);

map.put("end",endTime);

map.put("status", Orders.COMPLETED);

Double turnover = orderMapper.sumByMap(map);

turnover = turnover == null ? 0.0 : turnover;

turnoverList.add(turnover);

}

//封装返回结果

return TurnoverReportVO

.builder()

.dateList(StringUtils.join(dataList,","))

.turnoverList(StringUtils.join(turnoverList,","))

.build();

}

}

Mapper层

<select id="sumByMap" resultType="java.lang.Double">

select sum(amount) from orders

<where>

<if test="begin!=null">

and order\_time &gt;#{begin}

</if>

<if test="end!=null">

and order\_time &lt;#{end}

</if>

<if test="status!=null">

and status=#{status}

</if>

</where>

</select>

**用户统计**

分为总的数量和新增的数量 根据时间选择区间，来展示相关数据

Controller层

/\*

\* 用户统计

\* \*/

@GetMapping("/userStatistics")

@ApiOperation("用户统计")

public Result<UserReportVO> userStatistics(@DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate begin,@DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate end){

log.info("用户数据统计：{},{}",begin,end);

return Result.success(reportService.getUserStatistics(begin,end));

}

Service层

/\*

\* 统计指定时间区间内的用户数据

\* \*/

@Override

public UserReportVO getUserStatistics(LocalDate begin, LocalDate end) {

//存放从begin到end之间的每天对应的日期

List<LocalDate> dateList=new ArrayList<>();

dateList.add(begin);

while(!begin.equals(end)){

begin=begin.plusDays(1);

dateList.add(begin);

}

//存放每天新增用户数量

List<Integer> newUserList=new ArrayList<>();

//存放每天总的用户数量

List<Integer> totalUserList=new ArrayList<>();

for (LocalDate date : dateList) {

LocalDateTime beginTime = LocalDateTime.of(date, LocalTime.MIN);

LocalDateTime endTime = LocalDateTime.of(date, LocalTime.MAX);

Map map=new HashMap<>();

map.put("end",endTime);

//总的用户数量

Integer totalUser = userMapper.countByMap(map);

//新增用户数量

map.put("begin",beginTime);

Integer newUser = userMapper.countByMap(map);

totalUserList.add(totalUser);

newUserList.add(newUser);

}

//封装结果数据

return UserReportVO

.builder()

.dateList(StringUtils.join(dateList,","))

.totalUserList(StringUtils.join(totalUserList,","))

.newUserList(StringUtils.join(newUserList,","))

.build();

}

Mapper层

<select id="countByMap" resultType="java.lang.Integer">

select count(id) from sky\_take\_out.user

<where>

<if test="begin!=null">

and create\_time &gt;#{begin}

</if>

<if test="end!=null">

and create\_time &lt;#{end}

</if>

</where>

</select>

**Top十统计**

Controller层

/\*

\* 销量排名top10

\* \*/

@GetMapping("/top10")

@ApiOperation("销量排名top10")

public Result<SalesTop10ReportVO> top10(@DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate begin, @DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate end){

log.info("销量排名top10：{},{}",begin,end);

return Result.success(reportService.getSalesTop10(begin,end));

}

Service层

/\*

\* 统计指定区间内的销量排名前十

\* \*/

@Override

public SalesTop10ReportVO getSalesTop10(LocalDate begin, LocalDate end) {

LocalDateTime beginTime = LocalDateTime.of(begin, LocalTime.MIN);

LocalDateTime endTime = LocalDateTime.of(end, LocalTime.MAX);

List<GoodsSalesDTO> salesTop10 = orderMapper.getSalesTop10(beginTime, endTime);

List<String> names = salesTop10.stream().map(GoodsSalesDTO::getName).collect(Collectors.toList());

String nameList=StringUtils.join(names,",");

List<Integer> numbers = salesTop10.stream().map(GoodsSalesDTO::getNumber).collect(Collectors.toList());

String numberList=StringUtils.join(numbers,",");

//封装返回结果数据

return SalesTop10ReportVO

.builder()

.nameList(nameList)

.numberList(numberList)

.build();

}

Mapper层

<select id="getSalesTop10" resultType="com.sky.dto.GoodsSalesDTO">

select od.name,sum(od.number) number

from order\_detail od,orders o

where od.order\_id =o.id and o.status=5

<if test="begin!=null">

and o.order\_time &gt;#{begin}

</if>

<if test="end !=null">

and o.order\_time &lt;#{end}

</if>

group by od.name

order by number desc

limit 0,10

</select>

**Day12**

**Apache POI**

是一个处理微软各种文件格式的开源项目，一般情况下，POI都是用来操作EXCEL文件的 读和写操作

test测试

/\*

\* 使用POI操作Excel文件

\* \*/

public class POITest {

/\*

\* 通过POI创建Excel文件并写入文件内容

\* \*/

public static void write() throws Exception{

//在内存中创建一个excel文件

XSSFWorkbook excel=new XSSFWorkbook();

//在Excel文件中创建一个sheet页

XSSFSheet sheet = excel.createSheet("info");

//在sheet中创建行对象，rownum编号从0开始

XSSFRow row = sheet.createRow(1);

//创建单元格并写入文件内容

row.createCell(1).setCellValue("姓名");

row.createCell(2).setCellValue("城市");

//创建一个新行

row = sheet.createRow(2);

row.createCell(1).setCellValue("张三");

row.createCell(2).setCellValue("北京");

row = sheet.createRow(3);

row.createCell(1).setCellValue("李四");

row.createCell(2).setCellValue("南京");

//通过输出流将内存中的Excel文件写入磁盘中

FileOutputStream out=new FileOutputStream(new File("/Users/renliaoliao/Desktop/info.xlsx"));

excel.write(out);

//关闭资源

out.close();

excel.close();

}

/\*

\* 通过POI读取Excel文件中的内容

\*

\* \*/

public static void read() throws Exception{

//读取磁盘上以及存在的excel文件

FileInputStream in = new FileInputStream(new File("/Users/renliaoliao/Desktop/info.xlsx"));

XSSFWorkbook excel = new XSSFWorkbook(in);

//将文本文件读取出来 读取第一个sheet页

XSSFSheet sheet = excel.getSheetAt(0);

//获取sheet页中最后一行的行号

int lastRowNum = sheet.getLastRowNum();

for (int i=1;i<=lastRowNum;i++){

//获取某一行

XSSFRow row = sheet.getRow(i);

//获取单元格对象

String cellValue1 = row.getCell(1).getStringCellValue();

String cellValue2 = row.getCell(2).getStringCellValue();

System.out.println(cellValue1+" "+cellValue2);

}

//关闭资源

in.close();

excel.close();

}

public static void main(String[] args) throws Exception {

//write();

read();

}

}

**导出运营数据表数据**

/\*

\* 导出运营数据报表

\* \*/

@Override

public void exportBusinessData(HttpServletResponse response) {

//1.查询数据库 获取运营数据

LocalDate dateBegin = LocalDate.now().minusDays(30);

LocalDate dateEnd = LocalDate.now().minusDays(1);

//查询概览数据

BusinessDataVO businessDataVO = workspaceService.getBusinessData(LocalDateTime.of(dateBegin, LocalTime.MIN), LocalDateTime.of(dateEnd, LocalTime.MAX));

//2.通过POI将数据写入到Excel文件中

InputStream in = this.getClass().getClassLoader().getResourceAsStream("template/运营数据报表模板.xlsx");

try {

//基于模板文件创建一个新的excel文件

XSSFWorkbook excel = new XSSFWorkbook(in);

//获取表格标签的sheet页

XSSFSheet sheet = excel.getSheet("Sheet1");

//填充数据--时间

sheet.getRow(1).getCell(1).setCellValue("时间："+dateBegin+"至"+dateEnd);

//获取第四行

XSSFRow row = sheet.getRow(3);

row.getCell(2).setCellValue(businessDataVO.getTurnover());

row.getCell(4).setCellValue(businessDataVO.getOrderCompletionRate());

row.getCell(6).setCellValue(businessDataVO.getNewUsers());

//获得第五行

row = sheet.getRow(4);

row.getCell(2).setCellValue(businessDataVO.getValidOrderCount());

row.getCell(4).setCellValue(businessDataVO.getUnitPrice());

//填充明细数据

for (int i = 0; i < 30; i++) {

LocalDate date = dateBegin.plusDays(i);

//查询某一天的营业数据

BusinessDataVO businessData = workspaceService.getBusinessData(LocalDateTime.of(date, LocalTime.MIN), LocalDateTime.of(date, LocalTime.MAX));

//获得某一行

row=sheet.getRow(7+i);

row.getCell(1).setCellValue(date.toString());

row.getCell(2).setCellValue(businessData.getTurnover());

row.getCell(3).setCellValue(businessData.getValidOrderCount());

row.getCell(4).setCellValue(businessData.getOrderCompletionRate());

row.getCell(5).setCellValue(businessData.getUnitPrice());

row.getCell(6).setCellValue(businessData.getNewUsers());

}

//3.通过输出流将excel文件下载到客户端浏览器

ServletOutputStream out = response.getOutputStream();

excel.write(out);

//关闭资源

out.close();

excel.close();

} catch (IOException e) {

throw new RuntimeException(e);

}

}

}