一、Redis

1.1 Redis入门

Redis（Remote Dictionary Server）是**非关系型数据库**，数据以**键值对**的形式存储，**基于内存**（而不是MySQL的磁盘），速度非常快，常用于缓存（存储热点数据，减少数据库压力）、中间件、消息队列等

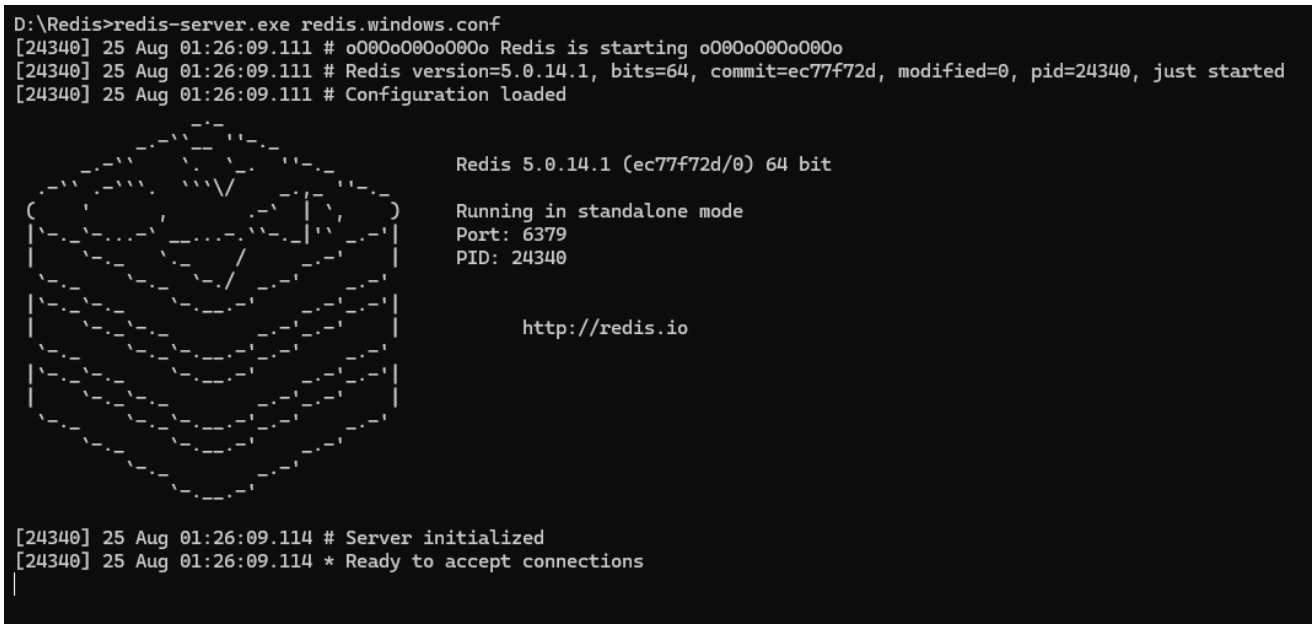
1.安装运行Redis

官网是不提供Windows直接运行的Redis的，需要借助Docker，若要使用Windows直接运行Redis，需前往Releases · tporadowski/redis进行下载msi或zip

在安装目录中，redis.windows.conf是配置文件，可以修改服务器bind、绑定的端口号port（默认6379）等，redis-cli.exe是客户端，redis-server.exe是服务端

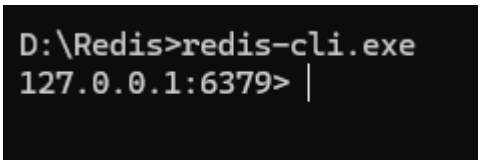
在Redis的安装目录打开终端，运行命令以启动Redis服务端

```
redis-server.exe redis.windows.conf
# 运行时可能报错，这是由于msi在安装时自动将Redis添加为服务并运行
# 输入以下命令删除服务并重启电脑
sc delete Redis
```



再打开一个新的终端，运行命令以启动Redis客户端

```
redis-cli.exe
```



可以将Redis服务器配置成开机自启动，就不用手动启动了

```
# 安装成服务
redis-server.exe --service-install redis.windows.conf --loglevel verbose
# 启动服务
redis-server.exe --service-start
```

2.配置Redis密码

Redis服务器默认是没有密码的，在redis.windows.conf里配置密码，Ctrl+F搜索requirepass，将前面的#删掉，将foobared替换为自己的密码

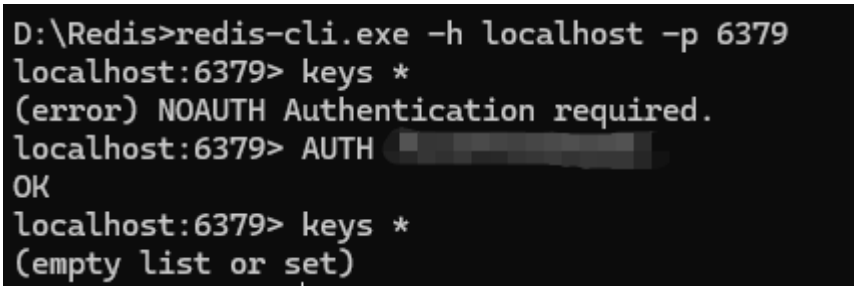
```
# Warning: since Redis is pretty fast an outside user can try up to
# 150k passwords per second against a good box. This means that you should
# use a very strong password otherwise it will be very easy to break.
#
requirepass xxx
```

可以使用-h和-p来指定要连接的Redis服务器的ip与端口号

```
redis-cli.exe -h localhost -p 6379
```

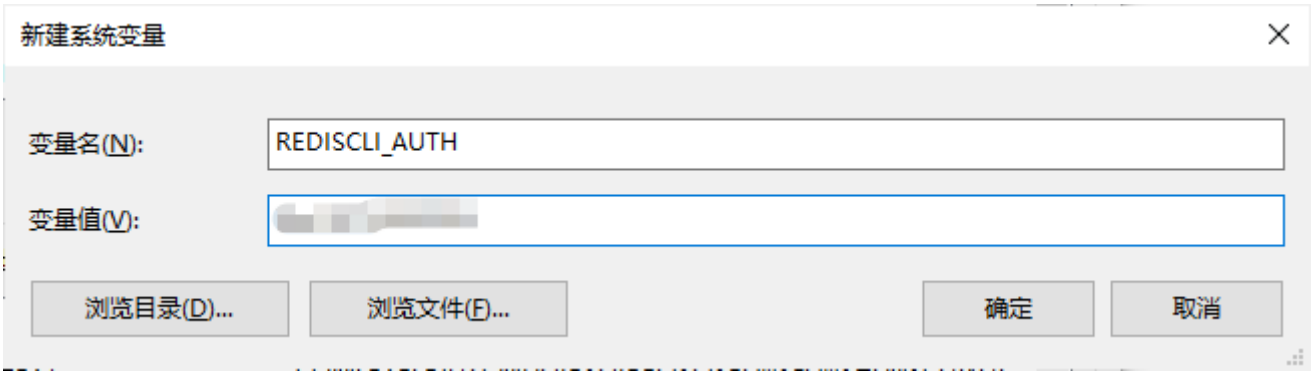
可以使用-a来指定连接密码，但是这样并不安全，在进入客户端后输入

```
AUTH password
```



即可安全登录Redis数据库

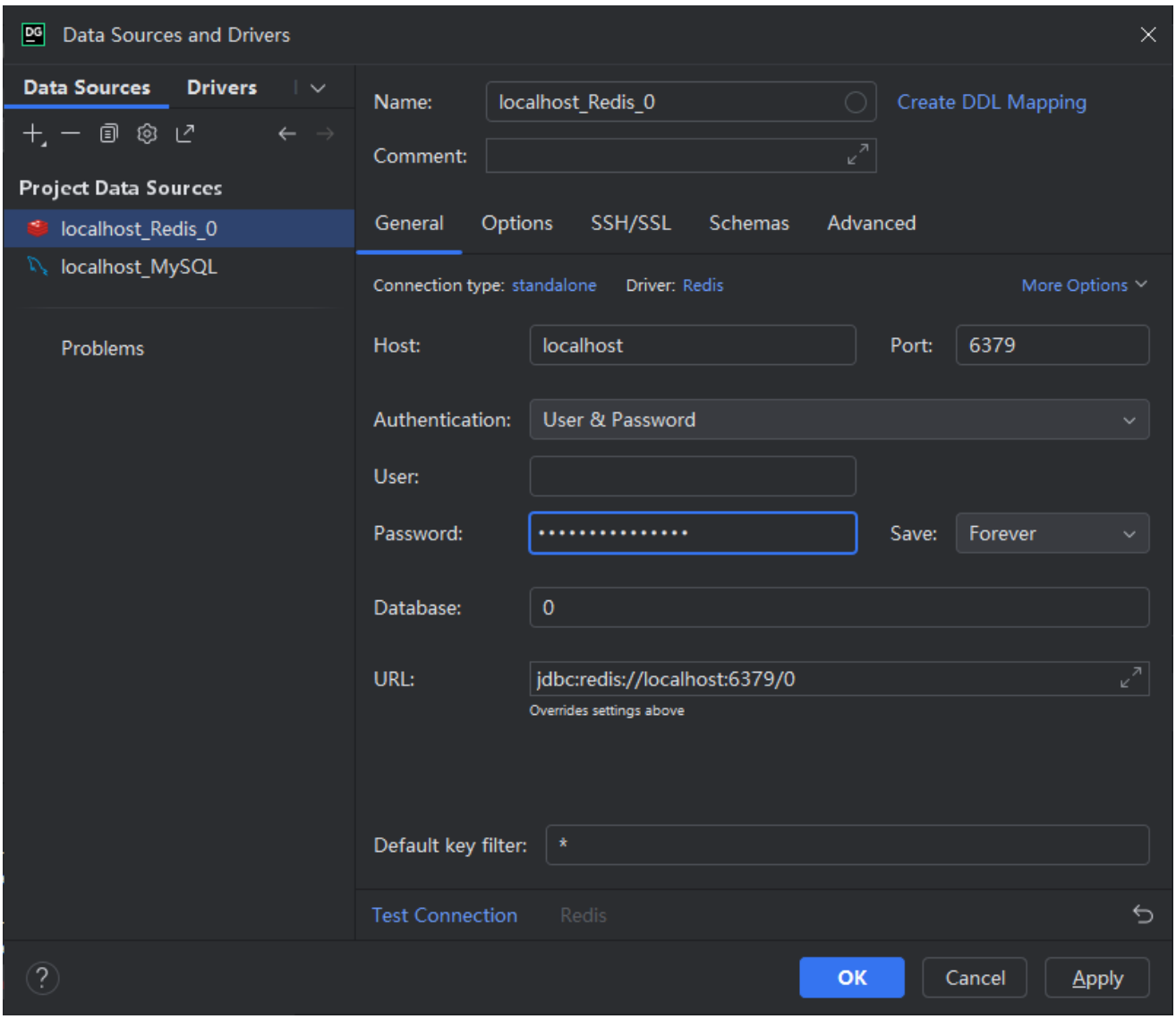
如果不想每次登录都输入密码，可以在环境变量里配置（推荐）



重新打开终端，不用AUTH即可登录

```
D:\Redis>redis-cli.exe -h localhost -p 6379
localhost:6379> keys *
(empty list or set)
localhost:6379>
```

可以在Datagrip里直接操作Redis数据库



（Database里的0指的是，Redis有0~15共16个逻辑数据库，默认0）

3.常用数据类型

String：最基本的数据类型，用于存储数字和字符串

Redis存储的是key-value结构的数据，其中key是String类型，value除了String类型以外，还有下面四种常用的数据类型

Hash：用于存储对象，里面有对象的各个field与value

List：有序可重复的**双向链表**

Set：无序不重复的集合

ZSet（Sorted Set）：有序不重复的集合，通过集合的权重**score**来表示集合的顺序，根据rank升序排序，常用于存储排行榜

1.2 Redis常用命令

1.Key命令

```
KEYS *  # 查看所有key
KEYS pattern  # 查找符合pattern（正则）的key
EXISTS key  # 查看key是否存在
DEL key  # 删除key
EXPIRE key seconds  # 设置key的过期时间为seconds
# 查看key剩余时间
# 返回-1说明key永久存在
# 返回-2说明key不存在（已过期或根本没创建过）
TTL key
# 获取key存值类型
TYPE key
```

2.String命令

```
# 设置与获取
SET key value
GET key
# 批量操作
MSET key1 value1 key2 value2 ...
MGET key1 key2 ...

# 设置key的过期时间（对于其他数据类型也适用）
SET key value EX seconds
# 只有key不存在时设置key值
SET key value NX
# 获取长度
STRLEN key

# 数值操作
INCR key  # 自增1
INCRBY key n  # 自增n
DECR key  # 自减1
DECRBY key n  # 自减n
```

3.Hash命令

```
# 设置与获取
HSET key field1 value1 field2 value2 ...
HGET key field
# 批量获取
HMGET key field1 field2 ...
HGETALL key
# 删除字段
HDEL key field
# 获取所有字段或所有值
HKEYS key
HVALS key

# 数值操作（对于其他数据类型也适用，改前缀）
HINCRBY key field n
...
```

4.List命令

```
# 插入元素
LPUSH key value1 value2 ...  # 从左边插入
RPUSH key value1 value2 ...  # 从右边插入
# 注意：插入的执行是依次的，也就是说
# LPUSH list "a" "b" "c"
# 得到的list是 "c" "b" "a"（先插a，在a的左边插b，在b的左边插c）

# 获取元素
LINDEX key index  # 按下标取值（0开始）
LRANGE key start stop  # 按下标范围取值
LLEN key  # 获取列表长度

# 弹出元素（第一个元素）
LPOP key  # 左边第一个
RPOP key  # 右边第一个
```

5.Set命令

```
# 添加与获取
SADD key value1 value2 ...
SMEMBER key  # 获取集合所有成员
SCARD key  # 获取集合成员数
# 删除
SREM key value1 value2 ...
# 判断
SISMEMBER key value

# 集合运算
SINTER key1 key2 ...  # 交
SUNION key1 key2 ...  # 并
SDIFF key1 key2 ...  # 差
```

6.ZSet命令

```
# 添加元素
ZADD key score1 value1 score2 value2 ...
# 获取元素，需指定获取的区间
ZRANGE key start stop WITHRANGE  # 按score升序排序
ZREVRANGE key start stop WITHSCORE  # 按score降序排序
# 获取排名与分数
ZRANK key value
ZSCORE key value
# 删除
ZREM key value1 value2 ...
# 统计操作
ZCOUNT key s1 s2  # 统计分数在s1~s2的成员的数量
ZRANGEBYSCORE key s1 s2  # 统计分数在s1~s2的成员
```

1.3 Spring Data Redis

在Java里操作Redis的方式有很多，常用的是Jedis和Lettuce，而Spring Data Redis作为Spring Data家族的一部分，抽象了Jedis和Lettuce的操作，让开发者能够以统一、面向对象的方式来访问Redis

1.配置

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>${springboot}</version>
    <scope>test</scope>
</dependency>
<!-- 此时springboot版本已更新到3.5.5，需要将父类的springboot和其他springboot依赖的版本修改为3.5.5 -->
```

在application.yml中配置数据源

```yaml
# application.yml
spring:
  data:
    redis:
      host: ${sky.data.redis.host}
      port: ${sky.data.redis.port}
      database: ${sky.data.redis.database}
      password: ${sky.data.redis.password}
      lettuce:
        pool:
          max-active: 8
          max-idle: 8
          min-idle: 0

# application-dev.yml
sky:
  data:
    redis:
      host: localhost
      port: 6379
      database: 0
      password: xxx
```

编写Redis配置类RedisConfiguration

```java
package com.sky.config;

import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
@Slf4j
public class RedisConfiguration {

    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {
        log.info("开始创建Redis模板类");
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(factory);

        // key使用String序列化
        template.setKeySerializer(new StringRedisSerializer());
        template.setHashKeySerializer(new StringRedisSerializer());

        return template;
    }
}

```

2.操作Redis

我们需要通过RedisTemplate对象来操作Redis，在需要操作Redis的类里面写一个Autowired

```java
@SpringBootTest
public class SkyApplicationTests {

    private final RedisTemplate<String, Object> redisTemplate;
    @Autowired
    public SkyApplicationTests(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }
}
```

RedisTemplate提供了五个方法，用于获取操作五种数据类型的Java对象，RedisTemplate本身可用于操作Key

```java
redisTemplate.opsForValue()  // String
redisTemplate.opsForHash()  // Hash
redisTemplate.opsForList()  // List
redisTemplate.opsForSet()  // Set
redisTemplate.opsForZSet()  // ZSet
```

这些对象操作数据库的方法名与Redis语法类似，可以通过看IDE的提示来进行代码的编写

```java
// String
redisTemplate.opsForValue().set("username", "voidept");
String name = (String) redisTemplate.opsForValue().get("username");
// Hash
redisTemplate.opsForHash().put("user1", "name", "voidept");
redisTemplate.opsForHash().put("user1", "age", "20");
```

为了方便，可以先创建ops对象，再操作，减少重复代码的编写

```java
@SpringBootTest
public class SkyApplicationTests {

    private final RedisTemplate<String, Object> redisTemplate;
    private final ValueOperations<String, Object> valueOperations;
    private final HashOperations<String, String, Object> hashOperations;
    private final ListOperations<String, Object> listOperations;
    private final SetOperations<String, Object> setOperations;
    private final ZSetOperations<String, Object> zSetOperations;
    @Autowired
    public SkyApplicationTests(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
        this.valueOperations = redisTemplate.opsForValue();
        this.hashOperations = redisTemplate.opsForHash();
        this.listOperations = redisTemplate.opsForList();
        this.setOperations = redisTemplate.opsForSet();
        this.zSetOperations = redisTemplate.opsForZSet();
    }

    @Test
    public void testRedis() {
        // String
        valueOperations.set("name", "voidept", 60, TimeUnit.SECONDS);
        System.out.println((String) valueOperations.get("name"));
        valueOperations.setIfAbsent("name", "aaa");
        // Hash
        hashOperations.put("user:1", "name", "voidept");
        hashOperations.put("user:1", "age", "20");
        // List
        listOperations.leftPush("userList", "user:1");  // 把key名存起来，就相当于把Hash存起来了
        // Set
        setOperations.add("userSet", "user:1");
        // ZSet
        zSetOperations.add("userZSet", "user:1", 100);
        // Key
        Set<String> keys = redisTemplate.keys("*");
    }
}
```

3.修改查询营业状态

为什么我们在修改和查询营业状态时，要用Redis来存储营业状态?

①速度快：因为用户端要随时能够查到这个营业状态，以根据营业状态来判断是否能下单外卖，而管理员也可能随时修改外卖状态，对查询数据速度的要求非常高，所以放在Redis里，Redis依托于内存，响应速度非常快，也可以减小MySQL的压力

②高并发：用户量变大时，查询营业状态的请求会非常多，Redis可以轻松抗住高并发

③分布式一致性：如果系统部署在多台机器，Redis可以作为这些机器的共享缓存，使得所有节点都能拿到同一个营业状态，可以避免不同节点的数据不一致

代码实现层面就很简单了，此处略

4.将菜品添加至Redis

当菜品数据变多时，每次读取都从MySQL中读取，会对数据库造成很大压力，读取时间也会变慢，所以我们选择将菜品数据缓存到Redis中

执行逻辑与计组中Cache的逻辑相同，即小程序查询菜品时首先判断Redis中有没有数据，有则直接查，没有则查MySQL，然后添加到Redis

我们在controller层完成此部分，Redis存储数据的格式为dish_分类id

```
@GetMapping("/list")
public Result<List<DishVO>> gerByCategory(Long categoryId) {
    log.info("根据分类id{}查询菜品（C端）", categoryId);

    // 查询Redis中是否存在菜品数据
    String key = "dish_" + categoryId;
    List<DishVO> dishVOList = (List<DishVO>) valueOperations.get(key);

    // 如果存在，直接返回查询数据
    if (dishVOList != null && !dishVOList.isEmpty()) {
        return Result.success(dishVOList);
    }
    // 如果不存在，则查询数据库，并将数据存到Redis中
    dishVOList = dishService.getByCategoryWithFlavors(categoryId);
    // 用value即可
    valueOperations.set(key, dishVOList);
    return Result.success(dishVOList);
}
```

此外，当我们修改MySQL的数据时，还要及时清理缓存，以避免数据冲突

在admin.DishController中，我们编写统一的清理缓存方法

```java
private void cleanCache(String pattern) {
    Set<String> keys = redisTemplate.keys(pattern);
    if (keys != null) {
        redisTemplate.delete(keys);
    }
}
```

然后在涉及到增删改的方法里清理缓存即可

二、HttpClient与小程序开发

2.1 HttpClient入门

1.简介

HttpClient是Apache提供的，**在Java客户端发送http请求**的编程工具包，支持发送GET/POST/PUT/DELETE等各种请求方式

```xml
<dependency>
    <groupId>org.apache.httpcomponents.client5</groupId>
    <artifactId>httpclient5</artifactId>
    <version>5.5</version>
</dependency>
```

使用HttpClient发送请求的步骤：

①创建HttpClient对象

②创建Http请求对象

③使用HttpClient的execute方法将请求发送出去

2.发送GET请求

GET请求没有请求体

```java
@Test
public void testHttpClientGET() throws Exception {
    // 创建HttpClient对象
    CloseableHttpClient httpClient = HttpClients.createDefault();
    // 创建http get请求的对象，参数为请求路径
    HttpGet request = new HttpGet("http://localhost:8080/admin/shop/status");
    // 发送请求，获取响应结果
    CloseableHttpResponse response = httpClient.execute(request);

    // 解析数据，注意是core5包下的
    int statusCode = response.getCode();  // 响应状态码
    HttpEntity httpEntity = response.getEntity();  // 响应体
    String responseBody = EntityUtils.toString(httpEntity);  // String响应体

    response.close();
    httpClient.close();
}
```

3.发送POST请求

这里POST请求的请求体为json数据

```java
@Test
public void testHttpClientPOST() throws Exception {
    CloseableHttpClient httpClient = HttpClients.createDefault();
    HttpPost request = new HttpPost("http://localhost:8080/admin/employee/login");

    // 构造请求体并set到请求上（使用alibaba的fastjson2）
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("username", "admin");
    jsonObject.put("password", "123456");
    StringEntity stringEntity = new StringEntity(jsonObject.toString(), ContentType.APPLICATION_JSON, "utf-8", true);
    request.setEntity(stringEntity);

    CloseableHttpResponse response = httpClient.execute(request);
    // 解析数据与GET相同
    System.out.println(EntityUtils.toString(response.getEntity()));
    response.close();
    httpClient.close();
}
```

2.2 微信小程序开发

1.准备工作

①注册

前往微信小程序官网公众号，按照要求进行注册

②完善小程序信息

在首页将小程序信息、小程序类目补充完整，不需要进行备案和认证

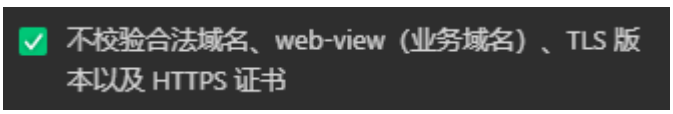
进入开发与服务→开发管理，获取AppID和AppSecret

③下载开发者工具

前往下载下载开发者工具，扫码登录进入

点击加号创建小程序，按要求填写信息，不使用模板，完成创建小程序

在右上角详情→本地设置里，勾选不校验合法域名...



2.目录结构

在根目录下，有三个重要文件：

app.js：小程序的逻辑
app.json：小程序的公共配置
app.wxss：小程序的公共样式表（类似css）
pages文件夹存放小程序的页面，一个小程序页面由四个文件组成：
.js：页面逻辑
.wxml：页面结构（类似html）
.json：页面配置
.wxss：页面样式表
也就是说小程序开发与前端开发是非常类似的，语法也类似，我们直接使用资料提供的代码即可

3.微信登录的流程

开放能力 / 用户信息 / 小程序登录
①首先小程序前端调用wx.login()方法获取code，然后调用wx.request()将code发送给后端
②后端收到这个code后，将appid、appsecret、code作为请求发送给微信接口服务小程序登录 / 小程序登录，接口响应给后端openid、session_key等，我们主要用openid
③后端通过openid生成一个token，发送给前端，前端就会带着这个token来调用业务方法，后端在拦截器里拦截这个token
我们主要做的是②③步
在sky-common的utils包里有HttpClient的工具类，可以直接发送出GET请求和POST请求，并接收响应的字符串数据

4.项目开发

首先在application.yml配置微信登录所需的配置项（配合jwt和wechat配置类）

```
sky:
  jwt:
    admin-secret-key: ${sky.jwt.admin-secret-key}
    admin-ttl: ${sky.jwt.admin-ttl}
    admin-token-name: ${sky.jwt.admin-token-name}  # token
    user-secret-key: ${sky.jwt.user-secret-key}
    user-ttl: ${sky.jwt.user-ttl}
    user-token-name: ${sky.jwt.user-token-name}  # authentication
  wechat:
    appid: ${sky.wechat.appid}
    secret: ${sky.wechat.secret}
```

然后编写三层架构以及拦截器，具体逻辑详见代码

2.3 微信支付

1.简介

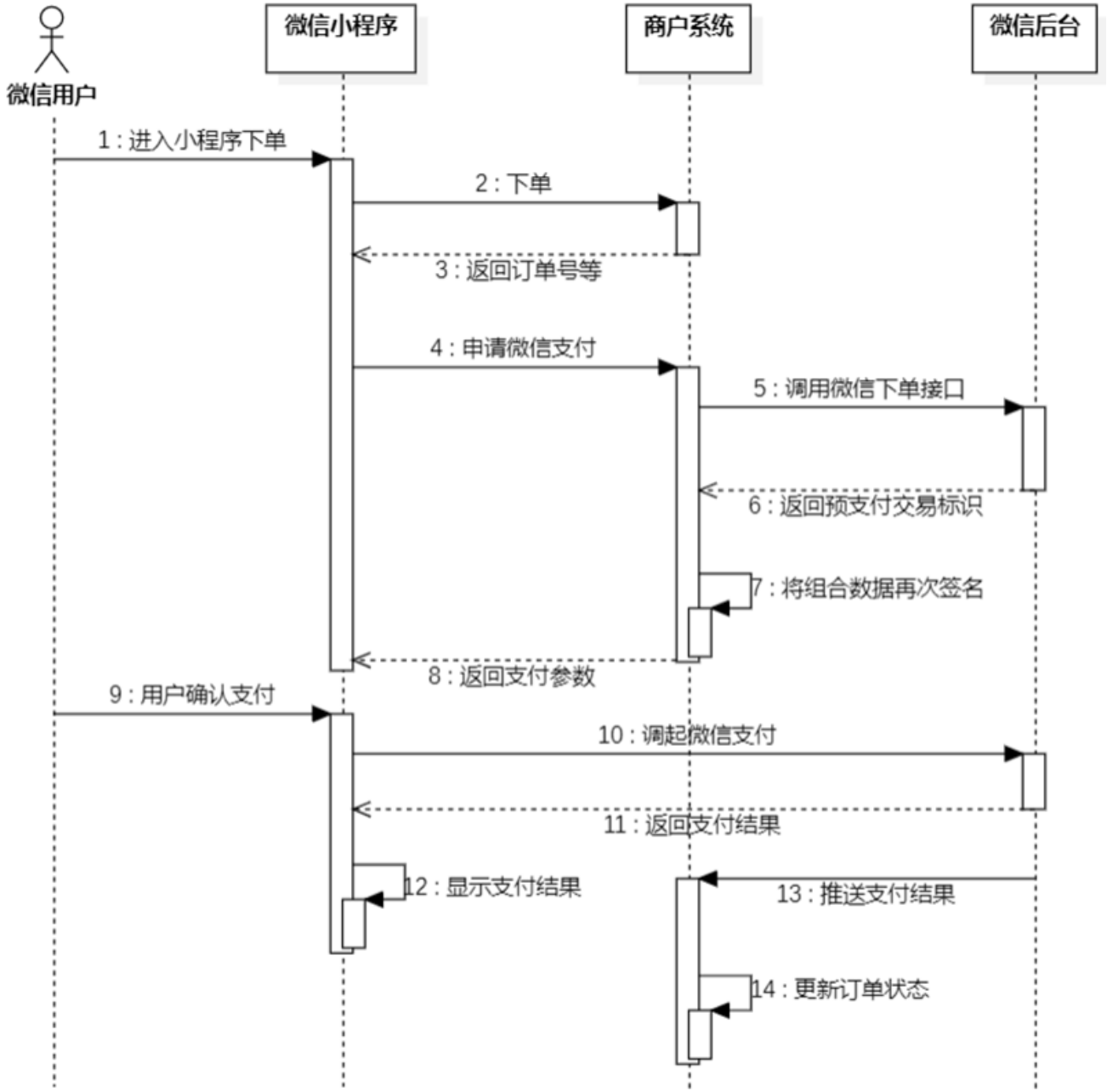
产品中心 - 微信支付商户平台

微信支付提供了多个支付API，包括付款码支付、小程序支付、APP支付、刷脸支付等，我们要使用的是小程序支付

小程序 - 微信支付接入指引 - 微信支付商户平台

右上角点击接入微信支付，可以通过营业执照等相关资料来使用微信支付功能

2.微信支付的流程



其中后端主要处理的是5 13 14，代码相对比较固定，要用时直接导入即可

PayNotifyController.java 3.92KB

OrderController.java 1.52KB

OrderMapper.java 621B

OrderServiceImpl.java 6.17KB

OrderService.java 664B

OrderMapper.xml 1.99KB

开发文档：开发指引_小程序支付|微信支付商户文档中心

三、Spring Cache

3.1 简介

Spring Cache是Spring提供的缓存抽象框架，统一了缓存的操作模式，这使得开发者不需要写具体的缓存逻辑，只需要在方法上加注解即可完成缓存

Spring Cache在底层可以自由切换缓存实现，比如Redis、EhCache、Caffeine等，只需要导入不同的缓存的Maven坐标即可

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
    <version>${springboot}</version>
</dependency>
```

3.2 常用注解

`@EnableCaching` ： 开启缓存注解功能，通常加在启动类上

`@Cacheable` ： 在方法执行前先查询缓存中有没有数据，如果有数据，则直接返回缓存数据，如果没有，则调用方法并将返回值放到缓存中

`@CachePut` ： 将返回值放到缓存中

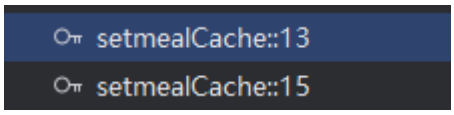
`@CacheEvict` ： 将数据从缓存中删除

有了这些注解，我们要想将套餐也缓存到Redis，只需要在根据id查询套餐上加@Cacheable注解，在增删改方法上加@CacheEvict注解即可

```java
@GetMapping("/list")
@Cacheable(value = "setmealCache", key = "#categoryId")
public Result<List<Setmeal>> getByCategoryId(Long categoryId) {
    log.info("根据分类id{}查询套餐", categoryId);
    return Result.success(setmealService.getByCategoryId(categoryId));
}
```

参数说明：

value（也可以是cacheNames）是缓存的分区，key是缓存的编号，二者共同组成Redis的key名




```java
// 新增套餐
@PostMapping
@CacheEvict(value = "setmealCache", key = "#setmealDTO.categoryId")
public Result<String> save(@RequestBody SetmealDTO setmealDTO) {...}
// 批量删除套餐
@DeleteMapping
@CacheEvict(value = "setmealCache", allEntries = true)
public Result<String> delete(@RequestParam List<Long> ids) {...}
```

删除的参数也是一样的，allEntries表示是否删除该缓存分区下的全部数据

四、Spring Task

4.1 简介

Spring Task是Spring框架自带的定时任务调度器（SpringBoot会自带，不需要引入依赖），用于**自动执行**定时任务或周期性的任务

比如在我们的项目中，当用户下单未支付时间超过15分钟之后，程序需要自动调用取消订单接口，这就是一个定时任务

或者是在每天闭店时都检查一下是否有未完成的订单，如果有的话就自动调用完成订单接口，这就是一个周期性的任务

Spring Task的底层是对JDK提供的线程池定时器ScheduledExecutorService的封装

4.2 常用注解

`@EnableScheduling`：开启定时任务功能，通常加载启动类或配置类上

`@Scheduled`：定义一个定时任务，里面放的参数即规定了什么时候执行这段代码

参数常见的有四种类型：

`initialDelay`：程序启动后延迟多久（毫秒）才第一次执行

`fixedRate`：以上一次任务**开始执行**的时间为基准，往后推固定毫秒执行

`fixedDelay`：以上一次任务**结束执行**的时间为基准，往后推固定毫秒执行

举例：https://chatgpt.com/s/t_68b3a38676a8819194eedfc9bdc9801f

```
@Scheduled(initialDelay = 2000, fixedRate = 5000)
// 2s → 7s → 12s → ...
```

`cron`：一个表达式，用于设定更为精确的调度规则

4.3 cron表达式

cron表达式为6个字段，中间为空格（Quartz表达式为7个字段，最后一个为年）

```
秒 分 时 日 月 星期
```

其中有一些特殊符号：

`*`：任意值

`,`：多个枚举值

`-`：范围

`?`：不指定值，只能用在日和星期

`/`：间隔执行

比如秒的0/5表示从0秒开始，每隔5秒执行一次

`L`：最后（周的最后一天为周六）

`W`：离指定日期最近的工作日，只能用在日

`#`：第几周的星期几

<星期几>#<第几周>，比如1#1为第一周的星期天，2#3为第三周的星期一

在线Cron表达式生成器

4.4 注意事项

1.定时任务的方法必须是**void**且**无参数**的（这就意味着无法将@Scheduled直接用于三层架构上，我们需要自己创建定时任务来调用三层架构的方法）

2.任务方法所在的类必须是Spring管理的Bean（@Component）

3.默认情况下，Spring Task的线程池只有一个线程，多个任务是串行执行，如果多个任务耗时长，就需要配置线程池，让任务并行执行，防止堵塞（**推荐**）

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.SchedulingConfigurer;
import org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler;
import org.springframework.scheduling.config.ScheduledTaskRegistrar;

@Configuration
@EnableScheduling
public class TaskConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar registrar) {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(5); // 线程池大小
        scheduler.setThreadNamePrefix("spring-task-");
        scheduler.initialize();
        registrar.setTaskScheduler(scheduler);
    }
}
```

4.5 项目开发

如4.4的1所说，我们需要自己写定时任务去调用三层架构的方法

新建一个包com.sky.task用于存放定时任务类

对于4.1的第一个例子，即自动检查未支付订单，我们的逻辑是每1分钟检查一次，看有没有未支付且超时的订单，有就将其取消掉

```java
@Scheduled(cron = "0 * 7-23 * * ?")
public void processTimeoutOrder() {
    log.info("支付超时订单处理{}", LocalDateTime.now());

    Integer status = Orders.PENDING_PAYMENT;
    LocalDateTime orderTime = LocalDateTime.now().minusMinutes(15);

    List<Orders> timeoutOrdersList = orderMapper.getByStatusAndTime(status, orderTime);
    for (Orders orders : timeoutOrdersList) {
        // 取消
        orders.setStatus(Orders.CANCELLED);
        orders.setCancelTime(LocalDateTime.now());
        orderMapper.update(orders);
    }
}
```

对于4.1的第二个例子，即自动检查当天未完成订单，我们的逻辑是每天晚上11点检查一次，看有没有status处于派送中的订单，有就使其已完成

```java
@Scheduled(cron = "0 0 23 * * ?")
public void processDeliveryOrder() {
    log.info("闭店未完成订单处理");

    Integer status = Orders.DELIVERY_IN_PROGRESS;
    LocalDateTime orderTime = LocalDateTime.now();

    List<Orders> deliveryOrdersList = orderMapper.getByStatusAndTime(status, orderTime);
    if (deliveryOrdersList != null && !deliveryOrdersList.isEmpty()) {
        for (Orders orders : deliveryOrdersList) {
            orders.setStatus(Orders.COMPLETED);
            orderMapper.update(orders);
        }
    }
}
```

五、WebSocket

5.1 简介

WebSocket是基于TCP的一种新的网络协议，与HTTP相比，它实现了“**全双工通信**”，意思是，客户端和服务端只需要进行一次握手，就可以同时发送和接收数据，不需要每次重新建立连接，连接是持久的，适用于实时性强的场景，比如实时聊天、游戏、视频弹幕等

而HTTP的连接是单向的短连接，客户端跟服务器建立连接，然后客户端发送请求，服务器给出响应，之后连接就断开了，之后需要重新建立连接，服务器也无法主动给客户端发送数据

WebSocket链接的格式为ws://localhost:8080或者wss://localhost:8080类型的（类似http与https）

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
    <version>${springboot}</version>
</dependency>
```

有两种方法可以操作WebSocket，一种是用Java EE的标准API（5.3），另一种是用Spring的WebSocket搭配STOMP协议（5.2），本项目使用第一种方法

5.2 使用Spring WebSocket + STOMP协议

首先编写配置类：

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;
import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;

@Configuration
@EnableWebSocketMessageBroker  // 开启STOMP协议支持
public class WebSocketConfiguration implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        // 定义客户端连接的端点（ws://localhost:8080/ws）
        registry.addEndpoint("/ws")
                .setAllowedOrigins("*")  // 允许跨域（任何域，可更改为指定的前端地址）
                .withSockJS();   // 兼容不支持WS的浏览器
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        // 客户端订阅消息的前缀（广播、群发）
        registry.enableSimpleBroker("/topic", "queue");
        // 客户端发送消息的前缀
        // 约定了客户端发消息用/app/xxx，服务端用/xxx接收消息
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

然后定义消息对象：

```java
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class ChatMessage {

    private String from;
    private String content;

}
```

最后定义Controller：

```java
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;

@Controller
public class ChatController {

    // 客户端发消息到 /app/chat
    @MessageMapping("/chat")
    @SendTo("/topic/messages") // 广播到 /topic/messages
    public ChatMessage send(ChatMessage message) {
        System.out.println("收到消息: " + message.getContent());
        return message; // 广播给所有订阅的客户端
    }
}
```

同时前端也需要引入sock.js和stomp.js，此处略

5.3 使用标准API

首先编写配置类：

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.server.standard.ServerEndpointExporter;

@Configuration

public class WebSocketConfiguration {

    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }

}

```

然后编写操作WebSocket的类：

```java
import jakarta.websocket.OnClose;
import jakarta.websocket.OnMessage;
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
import jakarta.websocket.server.PathParam;
import jakarta.websocket.server.ServerEndpoint;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

@Component
// 声明一个WebSocket端点，每当客户端通过ws://localhost:8080/ws/xxx连接时，都由这个类来处理
@ServerEndpoint("/ws/{sid}")
@Slf4j
public class WebSocketServer {

    // 存放会话对象
    private static Map<String, Session> sessionMap = new HashMap<>();

    /**
     * 建立连接成功后调用该方法
     * @param session
     * @param sid
     */
    @OnOpen
    public void onOpen(Session session, @PathParam("sid") String sid) {
        log.info("客户端{}建立连接", sid);
        sessionMap.put(sid, session);
    }

    /**
     * 关闭连接后调用该方法
     * @param sid
     */
    @OnClose
    public void onClose(@PathParam("sid") String sid) {
        log.info("连接{}断开", sid);
```

```
        sessionMap.remove(sid);
    }

    /**
     * 收到客户端消息后发送的方法
     * @param message
     * @param sid
     */
    @OnMessage
    public void onMessage(String message, @PathParam("sid") String sid) {
        log.info("收到客户端消息");
    }


    /**
     * 群发消息到客户端
     * @param message
     */
    public void sendToClient(String message) {
        Collection<Session> sessions = sessionMap.values();
        for (Session session : sessions) {
            try {
                session.getBasicRemote().sendText(message);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

}
```

5.4 来单提醒与客户催单

来单提醒与客户催单，就是当客户支付完成后或者点击催单后，服务器要向admin端发送一条提醒的消息，已约定好提醒消息的格式：

```
{
    "type": " "  // 1：来单提醒 2：客户催单
    "orderId": " "  // 订单id
    "content": " "  // 消息内容
}
```

对于来单提醒，我们只需要修改OrderService的paySuccess方法，添加调用WebSocketServer的广播方法即可

```
// 来单提醒
Map<String, Object> webServerMessage = new HashMap<>();
webServerMessage.put("type", 1);
webServerMessage.put("orderId", ordersDB.getId());
webServerMessage.put("content", "订单号" + outTradeNo);
webSocketServer.sendToClient(JSON.toJSONString(webServerMessage));
```

对于客户催单，只需要在service层写一个方法处理这个请求，然后同上

```
/**
 * 催单
 * @param id
 */
@Override
public void reminder(Long id) {

    Orders orders = orderMapper.getById(id);

    Map<String, Object> webSocketMessage = new HashMap<>();
    webSocketMessage.put("type", 2);
    webSocketMessage.put("orderId", id);
    webSocketMessage.put("content", "订单号" + orders.getNumber());
    webSocketServer.sendToClient(JSON.toJSONString(webSocketMessage));
}
```

六、Apache POI

Apache POI是Apache软件基金会的开源项目，提供了Java库用于操作MS Office文件，包括docx、xlsx、pptx等等，最常用的是操作xlsx文件

```
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>5.4.1</version>
</dependency>
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>5.4.1</version>
</dependency>
```

6.1 写入xlsx

```
// 创建一个Excel文件对象
XSSFWorkbook excel = new XSSFWorkbook();
// 创建一个工作表
XSSFSheet sheet1 = excel.createSheet("sheet1");
// 在sheet1里创建第1行（0表示第1行）
XSSFRow sheet1_row1 = sheet1.createRow(0);
// 在第一行创建单元格并设置值（单元格编号也是从0开始）
sheet1_row1.createCell(0).setCellValue("姓名");
sheet1_row1.createCell(1).setCellValue("年龄");
// 创建第二行并设置单元格值
XSSFRow sheet1_row2 = sheet1.createRow(1);
sheet1_row2.createCell(0).setCellValue("voidept");
sheet1_row2.createCell(1).setCellValue(20);
// ...
// 通过输出流将文件写入磁盘
FileOutputStream out = new FileOutputStream("D:\\study\\sky-take-out\\test.xlsx");
excel.write(out);
// 关闭资源
out.flush();
out.close();
excel.close();
```

```
// 创建日期格式
XSSFCellStyle dateStyle = excel.createCellStyle();
XSSFDataFormat dataFormat = excel.createDataFormat();
dateStyle.setDataFormat(dataFormat.getFormat("yyyy-MM-dd"));
// 设置日期格式
XSSFCell dateCell = row.createCell(0);
dateCell.setCellValue(dateList.get(i - 1));
dateCell.setCellStyle(dateStyle);
```

6.2 读取xlsx

```
// 处理不同类型单元格的工具方法
private String getCellValue(XSSFCell cell) {
    if (cell == null) return "";

    String value = "";
    if (cell.getCellType().equals(CellType.STRING)) {
        // 字符串类型单元格
        value = cell.getStringCellValue();
    } else if (cell.getCellType().equals(CellType.NUMERIC)) {
        if (DateUtil.isCellDateFormatted(cell)) {
            // 日期类型单元格
            value = cell.getDateCellValue().toString();
        } else {
            // 普通数字单元格
            value = String.valueOf(cell.getNumericCellValue());
        }
    } else if (cell.getCellType().equals(CellType.BOOLEAN)) {
        // 布尔类型单元格
        value = String.valueOf(cell.getBooleanCellValue());
    } else if (cell.getCellType().equals(CellType.FORMULA)) {
        // 公式类型单元格，获取计算结果
        value = String.valueOf(cell.getCellFormula());
    }
    return value;
}
```

```java
// 通过输入流将文件写入内存
FileInputStream in = new FileInputStream("D:\\study\\sky-take-out\\test.xlsx");
XSSFWorkbook excel = new XSSFWorkbook(in);
// 获取第一个工作表
XSSFSheet sheet1 = excel.getSheetAt(0);
// 获取sheet1的最后一个行号
int lastRowNum = sheet1.getLastRowNum();

// 打印工作表
for (int i = 0; i <= lastRowNum; i++) {
    XSSFRow row = sheet1.getRow(i);
    if (row == null) continue;  // 处理空行

    String value1 = getCellValue(row.getCell(0));
    String value2 = getCellValue(row.getCell(1));
    System.out.println(value1 + " " + value2);
}
```

2025.8.22-2025.9.3 完成