# 第一周：

## 看项目代码;

🡪了解一些从未学过的类且项目里面有的配置(Java 11);

🡪了解项目中使用的技术:springboot + mybatis-plus + redis等。

🡪了解项目的结构;

在这个项目的后端代码中，一共有5个工程项目，1个为父工程，另外4个为子工程;

🡪父工程：view\_parent

🡪子工程：view\_common、view\_interface、view\_service与view\_rest

🡪view\_common：在这个工程中主要是一些公共信息。

里面还有一些工具类util：AESUtil、BtoaEncode、CommUtils、DataStructure、DateUtil、ResourceLoader与StringUtils

🡪view\_interface：这个子工程是主要是接口。在这个工程中，具有实体类、dto以及service层的接口

🡪view\_service:在这个子工程中，具有配置类、常数类、Mapper层的接口、service层的实现类、工具类以及mybatis的xml文件

🡪view\_rest：在这一个子工程中，具有Java bean、公共输出类(Response-->具有状态码、成功(失败)信息以及结果集)、配置类(StaticScheduleTask --> 定时器)、controller层、security包(AES加密解密数据等安全性操作)

🡪查看每一层所用到的注解;

🡪entity层：存放的是实体类，属性值与数据库值保持一致，实现 setter 和 getter 方法以及toString方法。

public class IbBuildingFamily extends Model<IbBuildingFamily>{}：继承 Model<类名>

@TableName("tb\_demand")：与数据库表连接

@TableId("id")：修饰主键

@TableField("phone")：修饰非主键

🡪dao层：即 mapper层，对数据库进行持久化操作，他的方法使针对数据库操作的，基本上用的就是增删改查，他就是个接口，只有方法名，具体实现在mapper.xml中实现。

在这一层项目中没有添加持久层注解（原因：https://blog.csdn.net/tianshan2010/article/details/105889133?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522162789850216780274135792%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=162789850216780274135792&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~baidu\_landing\_v2~default-1-105889133.first\_rank\_v2\_pc\_rank\_v29&utm\_term=dao%E5%B1%82%E4%B8%80%E5%AE%9A%E8%A6%81repository%E6%B3%A8%E8%A7%A3%E5%90%97&spm=1018.2226.3001.4187）

public interface DataMapper extends BaseMapper<IbVillageInfo>：继承BaseMapper<类名>，无需再写一些简单的mybatis的xml文件，类似于Jpa

🡪service层：业务层，存放业务逻辑处理，不直接对数据库进行操作，有接口和接口实现类，提供 controller 层调用方法。

@Service

@CacheConfig(cacheNames = "areaCache")

@Transactional：开启事务

@PostConstruct：该注解被用来修饰一个非静态的 void() 方法。被 @PostConstruct 修饰的方法会在服务器加载 Servlet 的时候运行，并且只会被服务器执行一次。@PostConstruct 在构造函数之后执行，init（）方法之前执行。可以使用 @PostConstruct 注释的方法进行初始化，而且该初始化方法中需要使用到 Spring 容器中的对象。因为使用静态代码块、非静态代码块、构造器都是在将 Bean 注入到 IOC 容器之前执行，所以无法借用 IOC 容器中的对象，无法使用这些结构进行初始化。

@PreDestroy:用与在依赖注入完成之前的方法前面执行。(@PostConstruct与@PreDestroy详解：https://blog.csdn.net/ouzhuangzhuang/article/details/82145837?ops\_request\_misc=&request\_id=&biz\_id=102&utm\_term=@PreDestroy()&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~sobaiduweb~default-3-.pc\_search\_result\_before\_js&spm=1018.2226.3001.4187)

🡪controller层：控制层，导入 service层，调用你service方法，controller通过接受前端传来的参数进行业务操作，在返回一个制定的路径或数据表。

@RestController：包含了@Controller与@ResponseBody两个注解

@RequestBody：请求参数体

@Value("${redis.check.key}")：获取配置里面的值

@PostMapping("/areaMatch")

@RequestMapping("/area")

## 学习Mybatis-Plus;

官网：<https://baomidou.com/>

1、添加pom.xml注解：

<?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>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.8.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <groupId>com.chen</groupId>  
 <artifactId>mybatis\_plus</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>mybatis\_plus</name>  
 <description>Demo project for Spring Boot</description>  
 <properties>  
 <java.version>11</java.version>  
 </properties>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <!--mysql-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
 <!--lombok-->  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </dependency>  
 <!--mybatis-plus-->  
 <dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-boot-starter</artifactId>  
 <version>3.0.5</version>  
 </dependency>  
  
 <!-- mybatis plus 自动生成 -->  
 <dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-generator</artifactId>  
 <version>3.4.1</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.velocity</groupId>  
 <artifactId>velocity-engine-core</artifactId>  
 <version>2.2</version>  
 </dependency>  
  
 <!-- mybatis plus 自动生成 -->  
 <dependency>  
 <groupId>cn.hutool</groupId>  
 <artifactId>hutool-all</artifactId>  
 <version>5.6.7</version>  
 </dependency>  
  
 <!--JSONObject转化为实体类-->  
 <dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>fastjson</artifactId>  
 <version>1.2.76</version>  
 </dependency>  
  
 <!--使用StringUtils类-->  
 <dependency>  
 <groupId>org.apache.velocity</groupId>  
 <artifactId>velocity-engine-core</artifactId>  
 <version>2.2</version>  
 </dependency>  
  
 <!--websocket使用依赖-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-websocket</artifactId>  
 </dependency>  
  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

2、编写配置application.yml

# mysql 8 驱动不同 com.mysql.cj.jdbc.Driver . 需要增加时区的配置

serverTimezone=GMT%2B8

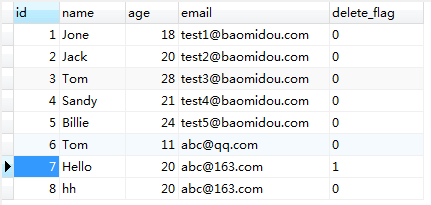
spring.datasource.username=root  
spring.datasource.password=  
spring.datasource.url=jdbc:mysql://localhost:3306/test\_mybatis\_plus?useSSL=false&useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
  
#myBatis配置文件  
mybatis-plus.mapper-locations=classpath:/mappers/\*Mapper.xml  
mybatis-plus.type-aliases-package=com.chen.mybatis\_plus.dao  
  
# 配置日志 (默认控制台输出)  
mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl  
  
# 逻辑删除  
## 全局逻辑删除的实体字段名(since 3.3.0,配置后可以忽略不配置步骤2)  
#mybatis-plus.global-config.db-config.logic-delete-field = flag  
# 逻辑已删除值(默认为 1)  
mybatis-plus.global-config.db-config.logic-delete-value= 1  
#逻辑未删除值(默认为 0)  
mybatis-plus.global-config.db-config.logic-not-delete-value= 0

3、主启动类:

package com.chen.mybatis\_plus;

import com.baomidou.mybatisplus.autoconfigure.MybatisPlusAutoConfiguration;  
import org.mybatis.spring.annotation.MapperScan;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
@MapperScan("com.chen.mybatis\_plus.dao")  
public class MybatisPlusApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(MybatisPlusApplication.class, args);  
 }  
  
}

4、创建数据库



5、创建实体类

package com.chen.mybatis\_plus.model;

import com.baomidou.mybatisplus.annotation.TableField;  
import com.baomidou.mybatisplus.annotation.TableId;  
import com.baomidou.mybatisplus.annotation.TableLogic;  
import com.baomidou.mybatisplus.annotation.TableName;  
import com.baomidou.mybatisplus.extension.activerecord.Model;  
  
@TableName("user")  
public class User extends Model<User> {  
 @TableId("id")  
 private String id;  
 @TableField("name")  
 private String name;  
 @TableField("age")  
 private int age;  
 @TableField("email")  
 private String email;  
 @TableLogic  
 @TableField("delete\_flag")  
 private String delete\_flag;  
  
 public String getId() {  
 return id;  
 }  
  
 public String getDelete\_flag() {  
 return delete\_flag;  
 }  
  
 public void setDelete\_flag(String delete\_flag) {  
 this.delete\_flag = delete\_flag;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "id='" + id + '\'' +  
 ", name='" + name + '\'' +  
 ", age=" + age +  
 ", email='" + email + '\'' +  
 ", delete\_flag='" + delete\_flag + '\'' +  
 '}';  
 }  
}

6、创建dao层：

package com.chen.mybatis\_plus.dao;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;  
import com.chen.mybatis\_plus.model.User;  
import org.apache.ibatis.annotations.Mapper;  
import org.apache.ibatis.annotations.Param;  
  
@Mapper  
public interface UserDao extends BaseMapper<User> {  
 boolean insert1(@Param("user") User user);  
}

7、创建Mapper文件：

<?xml version="1.0" encoding="utf-8" ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >  
<mapper namespace="com.chen.mybatis\_plus.dao.UserDao">  
 <insert id="insert1">  
 insert into user(name,age,email,delete\_flag) values (#{user.name},#{user.age},#{user.email},#{delete\_flag})  
 </insert>  
  
</mapper>

8、创建Service

package com.chen.mybatis\_plus.service;

import com.chen.mybatis\_plus.model.User;  
  
import java.util.List;  
  
public interface UserService {  
 List<User> select();  
 boolean insert1(User user);  
 int update(User user);  
 int delete(User user);  
 List<User> selectWrapper();  
 List<User> selectPage();  
}

9、创建ServiceImpl

package com.chen.mybatis\_plus.service.impl;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;  
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;  
import com.baomidou.mybatisplus.core.toolkit.Wrappers;  
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;  
import com.chen.mybatis\_plus.dao.UserDao;  
import com.chen.mybatis\_plus.model.User;  
import com.chen.mybatis\_plus.service.UserService;  
import org.apache.commons.lang3.StringUtils;  
import org.springframework.stereotype.Service;  
  
import javax.annotation.Resource;  
import java.util.List;  
  
@Service  
public class UserServiceImpl implements UserService {  
 @Resource  
 private UserDao userDao;  
  
 @Override  
 public List<User> select() {  
 return userDao.selectList(null);  
 }  
  
 @Override  
 public boolean insert1(User user) {  
 return userDao.insert1(user);  
 }  
  
 @Override  
 public int update(User user) {  
 return userDao.updateById(user);  
 }  
  
 @Override  
 public int delete(User user) {  
 if (StringUtils.*isNotEmpty*(user.getId())){  
 userDao.deleteById(user);  
 return 1;  
 }  
 return 0;  
 }  
  
 @Override  
 public List<User> selectWrapper() {  
 //查询姓名不为空,邮箱不为空,年龄大于18  
 QueryWrapper<User> lq = new QueryWrapper<>();  
// lq.isNotNull("name").isNotNull("email").gt("age", 18);  
// lq.eq("age", 18);  
// lq.between("age", 20, 24);  
// lq.like("name", "il");  
// lq.likeLeft("name", "Bil");  
// lq.likeRight("name", "Bil");  
// lq.in("age", 18,20,23);  
// lq.inSql("id", "select uid from role,user where user.id = role.uid ");  
// lq.select("count(\*)").groupBy("name");  
// lq.select("name,count(name)").groupBy("name").having("count(\*) = 2");  
// lq.func(i -> {  
// if (true){  
// lq.select("name");  
// }else {  
// lq.eq("name", "Tom");  
// }  
// });  
// lq.gt("age", 22).or().lt("age", 12);  
// lq.exists("select id from user where age = 18");  
 return userDao.selectList(lq);  
 }  
  
 @Override  
 public List<User> selectPage() {  
 //参数一： 当前页数  
 //参数二： 每页多少条数据  
 Page<User> page = new Page<>(1, 3);  
 userDao.selectPage(page, null);  
 List<User> pageRecords = page.getRecords();  
 return pageRecords;  
 }  
  
}

10、创建Controller

package com.chen.mybatis\_plus.controller;

import com.alibaba.fastjson.JSONObject;  
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;  
import com.chen.mybatis\_plus.common.Response;  
import com.chen.mybatis\_plus.dao.UserDao;  
import com.chen.mybatis\_plus.model.User;  
import com.chen.mybatis\_plus.service.UserService;  
import com.chen.mybatis\_plus.service.impl.WebSocketServiceImpl;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.messaging.handler.annotation.MessageMapping;  
import org.springframework.messaging.handler.annotation.SendTo;  
import org.springframework.messaging.simp.SimpMessagingTemplate;  
import org.springframework.messaging.simp.user.SimpUserRegistry;  
import org.springframework.web.bind.annotation.\*;  
  
import javax.annotation.Resource;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
@RestController  
@RequestMapping("/api")  
@Slf4j  
public class UserController {  
 @Resource  
 private UserService userService;  
 @Resource  
 private UserDao userDao;  
  
 @Resource  
 private WebSocketServiceImpl webSocketService;  
  
 /\*=====================mybatis-plus实现增删查改======================\*/  
 @GetMapping("/select")  
 public Response<List<User>> select(){  
 List<User> userInfo = userService.select();  
 /\*=====================List数据======================\*/  
 *log*.info(String.*valueOf*(userInfo));  
 /\*=====================Map数据======================\*/  
 Map check = new HashMap();  
 userInfo.forEach(info -> {  
 check.put("name", info.getName());  
 check.put("age", info.getAge());  
 check.put("email", info.getEmail());  
  
 });  
 *log*.info("/\*=====================Map数据======================\*/");  
 check.forEach((k,v) -> *log*.info("key:value = " + k + ":" + v));  
 *log*.info(String.*valueOf*(check));  
 return new Response<List<User>>(userInfo);  
 }  
  
 @PostMapping("/insert")  
 public Response insert(@RequestBody JSONObject res){  
// String name = res.getString("name");  
// String age = res.getString("age");  
// String email = res.getString("email");  
 User user = JSONObject.*toJavaObject*(res,User.class);  
// User user = new User();  
// user.setName(name);  
// user.setAge(Integer.parseInt(age));  
// user.setEmail(email);  
// if (userService.insert1(user)){  
// return new Response();  
// }  
 if (userDao.insert(user) != 0){  
 /\*=====================websocket推送信息的使用======================\*/  
 try {  
 webSocketService.sendMessage("新增个人信息成功！");  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 return new Response();  
 }  
 return new Response("更改失败!");  
 }  
  
 @PostMapping("/update")  
 public Response update(@RequestBody JSONObject res){  
 User user = JSONObject.*toJavaObject*(res, User.class);  
 int update = userService.update(user);  
 *log*.info(String.*valueOf*(update));  
 return new Response();  
 }  
  
 @DeleteMapping("/delete")  
 public Response delete(@RequestBody JSONObject res){  
 User user = JSONObject.*toJavaObject*(res, User.class);  
 int i = userService.delete(user);  
 if ( i == 1){  
 return new Response();  
 }  
 return new Response("删除失败!");  
 }  
  
 /\*========================条件构造器的使用==========================\*/  
 @GetMapping("/selectWrapper")  
 public Response selectWrapper(){  
 List<User> userList = userService.selectWrapper();  
 return new Response(userList);  
 }  
  
 @GetMapping("/selectPage")  
 public Response selectPage(){  
 List<User> userList = userService.selectPage();  
 return new Response(userList);  
 }  
  
  
}

## 学习fastjson里面的JSONObject类

fastjson.jar是阿里巴巴开发专门处理用于Java开发的包，可以方便的实现JSON对象与JavaBean对象的转换，实现JavaBean对象与Json字符串的转换，实现JSON对象与JSON字符串的转换。

//将Json字符串转化为Json对象

JSONObject obj=JSON.parseObject(jsonStr);//将json字符串转换为json对象

导入依赖包:

<dependency>

<groupId>com.alibaba</groupId>  
 <artifactId>fastjson</artifactId>  
 <version>1.2.76</version>  
</dependency>

使用方式：

@Test

void test6(){  
 //通过原生生成的JSON格式生成  
 JSONObject object = new JSONObject();  
 object.put("name", "张三");  
 object.put("age", 18);  
 object.put("email", "abc@163.com");  
 System.*out*.println("原生生成的JSON格式生成:" + object.toString());  
  
 //通过hashMap数据结构生成  
 HashMap<String, Object> objectHashMap = new HashMap<>();  
 objectHashMap.put("name", "张三");  
 objectHashMap.put("age", 18);  
 objectHashMap.put("email", "abc@qq.com");  
 System.*out*.println("hashMap数据结构生成:" + new JSONObject(objectHashMap).toString());  
  
 //通过实体生成  
 User user = new User();  
 user.setName("张三");  
 user.setAge(18);  
 user.setEmail("abc@qq.com");  
 System.*out*.println("实体生成Json格式:" + JSON.*toJSON*(user)); //生成Json格式  
 String jsonString = JSONObject.*toJSONString*(user); //对象转成String  
 System.*out*.println("对象转成String:" + jsonString);  
  
 //Json字符串转换成Json对象  
 String user1 = "{\"name\":\"张三\",\"age\":18,\"email\":\"abc.@qq.com\"}";  
 JSONObject jsonObject = JSONObject.*parseObject*(user1);  
 System.*out*.println("Json字符串转换成Json对象:" + jsonObject.toString());  
  
 //list对象转listJson  
 ArrayList<User> users = new ArrayList<>();  
 User user2 = new User();  
 user2.setName("张三");  
 user2.setAge(18);  
 user2.setEmail("abc@qq.com");  
  
 users.add(user2);  
  
 User user3 = new User();  
 user3.setName("李四");  
 user3.setAge(20);  
 user3.setEmail("abc@163.com");  
  
 users.add(user3);  
  
 String string = JSON.*toJSON*(users).toString();  
 System.*out*.println("list转Json字符串:" + string);  
  
 JSONArray jsonArray = JSONObject.*parseArray*(string);  
 System.*out*.println("json字符串转listJson格式:" + jsonArray);  
  
 //JsonObject转实体对象  
 JSONObject object1 = new JSONObject();  
 object1.put("name", "张三");  
 object1.put("age", 18);  
 object1.put("email", "abc@163.com");  
 User user4 = JSONObject.*toJavaObject*(object1, User.class);  
 System.*out*.println(user4);  
}

1、JSONObject相当于json对象，所以该类中主要封装了各种get方法，通过"键：值"对中的键来获取其对应的值。且方法的输入参数几乎皆为String类型，这是因为json对象中，"键：值"对的键都是String类型的。

🡪getString(String key)方法，该方法输入参数为String key（键），输出为String ，用于获取json对象中的字符串型数据。例如通过该方法获取 "name"："bob"键值对中name这个键所对应的值bob。

🡪getInteger(String key)方法，该方法获取json对象中的整型数据，例如获取"age：20"键值对中age对应的整型数值20.

🡪JSONObject对应json对象，通过各种形式的get()方法可以获取json对象中的数据，也可利用诸如size()，isEmpty()等方法获取"键：值"对的个数和判断是否为空。其本质是通过实现Map接口并调用接口中的方法完成的。

2、JSONArray代表json对象数组，json数组对象中存储的是一个个json对象，所以类中的方法主要用于直接操作json对象。比如这其中的add(),remove()，containsAll()方法，对应于json对象的添加，删除与判断。

🡪getJSONObject(int index)方法，该方法用于获取json对象数组中指定位置的JSONObject对象，配合size()方法，可用于遍历json对象数组中的各个对象。通过以上两个方法，在配合for循环，即可实现json对象数组的遍历，当然JSONArray中也实现了迭代器方法来遍历，这和List的遍历极为相似。

## 代码项目中的一些理解：

serviceImpl的巧妙用法：

1、通过利用String来创建表名的String tableName = "";的数据，调用自己创建的tableName = getTableName(regionId, countyId, villageId);方法来获取表名，之后可以通过判断来进行CRUD的操作；

2、

（1）在设置登录的时候，可以创建一个service来进行初始化AbilityPlatformService service = new AbilityPlatformService(restTemplate, appid, secretKey, ip + url);

（2）通过从前端传过来的sessionId等数据进行封装，之后调用创建的service实现doPost()方法；

（3）在doPost()方法中通过对这些数据进行加密，如MD5算法（MessageDigest类为应用程序提供信息摘要算法的功能，如 MD5 或 SHA 算法。信息摘要是安全的单向哈希函数，它接收任意大小的数据，并输出固定长度的哈希值。）。通过产生的加密数据，把它称为Token值；

MessageDigest md5 = MessageDigest.*getInstance*("MD5");

md5.update(stringBuilder.append(*APP\_ID*).append(appId).append(*TIMESTAMP*).append(timestamp).append(*TRANS\_ID*).append(transId).append(secretKey).toString().getBytes(StandardCharsets.*UTF\_8*));  
stringBuilder.setLength(0);

byte[] b = md5.digest();

int i;  
for (byte aByte : b) {  
 i = aByte;  
 if (i < 0) {  
 i += 256;  
 }  
 if (i < 16) {  
 stringBuilder.append("0");  
 }  
 //toHexString():以十六进制的无符号整数形式返回一个整数参数的字符串表示形式  
 stringBuilder.append(Integer.*toHexString*(i));  
}

（4）之后把这些数据封装在一个JSONObject类里面。之后创建HttpEntity对象，设置默认的请求头*DEFAULT\_HTTP\_HEADERS*

HttpEntity<String> httpEntity = new HttpEntity<>(jsonObject.toString(), *DEFAULT\_HTTP\_HEADERS*);

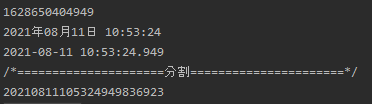
之后使用RestTemplate类里面的postForEntity(this.prefix, httpEntity, JSONObject.class).getBody();返回一个结果result确认是否是登录状态。

result = restTemplate.postForEntity(this.prefix, httpEntity, JSONObject.class).getBody();

## 项目代码中时间转换格式：

long millis = System.*currentTimeMillis*(); //获取系统的时间

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");  
Date date = new Date(millis);  
String s = new Timestamp(millis).toString();  
System.*out*.println(millis);  
System.*out*.println(simpleDateFormat.format(date)); //设置输出格式  
System.*out*.println(s); //输出Timestamp样式的时间  
/\*===========================================\*/  
System.*out*.println("/\*=====================分割======================\*/");  
StringBuffer stringBuilder = new StringBuffer();  
DateFormat dateFormat = new SimpleDateFormat("yyyyMMddHHmmssSSS");  
Random random = new Random();  
stringBuilder.append(dateFormat.format(new Date(millis))).append(String.*format*("%06d",  
 random.nextInt(1000000)));  
System.*out*.println(stringBuilder);

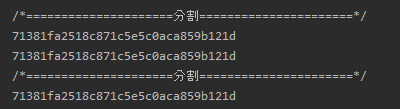
输出结果：

## 测试MD5算法加密：

MD5加密算法是不可逆的。经过该算法的处理之后看不到原文，只能通过比对处理后的数据来确定数据是否被篡改。

@Test

void test4(){  
 //测试MessageDigest类的加密,该加密是没有解码的,通常是通过比对加密的结果进行比较得出结果  
 StringBuilder str = new StringBuilder();  
 str.append("测试MD5算法").append("成功");  
 StringBuilder str1 = new StringBuilder();  
 str1.append("测试MD5算法成功");  
 System.*out*.println("/\*=====================分割======================\*/");  
 *testMD5*(str);  
 System.*out*.println(str); //对象是传地址的  
 System.*out*.println("/\*=====================分割======================\*/");  
 *testMD5*(str1);  
 }  
  
 public static void testMD5(StringBuilder str){  
 //测试MessageDigest类的加密  
 try {  
 MessageDigest md5 = MessageDigest.*getInstance*("MD5");  
 md5.update(str.toString().getBytes(StandardCharsets.*UTF\_8*));  
 str.setLength(0);  
 byte[] b = md5.digest();  
 int i;  
 for (byte aByte : b) {  
 i = aByte;  
 if (i < 0) {  
 i += 256;  
 }  
 if (i < 16) {  
 str.append("0");  
 }  
 //toHexString():以十六进制的无符号整数形式返回一个整数参数的字符串表示形式  
 str.append(Integer.*toHexString*(i));  
 }  
 System.*out*.println(str);  
 } catch (Exception e) {  
 System.*out*.println("执行MD5加密时出现异常:" + e);  
 }  
 }

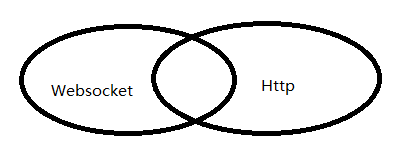
结果：

# 第二周：

## 了解websocket原理：

WebSocket是HTML5出的东西（协议），与HTTP协议可以说是无关系的。HTTP不能做到持久连接（长连接，循环连接不算）；

HTTP有1.1与1.0，也就是所谓的keep-alive，把多个HTTP请求合并成为一个，但是websocket其实是一个新的协议，与HTTP协议无关，只是一种握手的规范，也可以说是HTTP协议的一个补充。



websocket与HTTP关系 1

另外HTML5是指一系列新的API，是一种新规范、新技术。而HTTP协议本身只有1.0和1.1，而且跟HTML5没有直接的关系。通俗来说，使用的HTTP协议传输的是非HTML5的数据。

websocket的优点：websocket是一种持久化的协议，相对于HTTP这种非持久化协议来说。

HTTP的生命周期通过 Request 来界定，也就是一个 Request 一个 Response ，那么在 HTTP1.0 中，这次HTTP请求就结束了。

在HTTP1.1中进行了改进，使得有一个keep-alive，也就是说，在一个HTTP连接中，可以发送多个Request，接收多个Response。但是请记住 Request = Response ， 在HTTP中永远是这样，也就是说一个request只能有一个response。而且这个response也是被动的，不能主动发起。

以下是典型的websocket协议握手：

1. GET /chat HTTP/1.1
2. Host: server.example.com
3. Upgrade: websocket
4. Connection: Upgrade
5. Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
6. Sec-WebSocket-Protocol: chat, superchat
7. Sec-WebSocket-Version: 13
8. Origin: http://example.com

解释：

1. Upgrade: websocket
2. Connection: Upgrade

这个就是Websocket的核心了，告诉 Apache 、 Nginx 等服务器：注意啦，我发起的是Websocket协议，快点帮我找到对应的助理处理~不是那个老土的HTTP。

1. Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
2. Sec-WebSocket-Protocol: chat, superchat
3. Sec-WebSocket-Version: 13

首先， Sec-WebSocket-Key 是一个 Base64 encode 的值，这个是浏览器随机生成的，告诉服务器：泥煤，不要忽悠窝，我要验证尼是不是真的是Websocket助理。

然后， Sec\_WebSocket-Protocol 是一个用户定义的字符串，用来区分同URL下，不同的服务所需要的协议。简单理解：今晚我要服务A，别搞错啦~

最后， Sec-WebSocket-Version 是告诉服务器所使用的 Websocket Draft（协议版本），在最初的时候，Websocket协议还在 Draft 阶段，各种奇奇怪怪的协议都有，而且还有很多期奇奇怪怪不同的东西，什么Firefox和Chrome用的不是一个版本之类的，当初Websocket协议太多可是一个大难题。。不过现在还好，已经定下来啦~大家都使用的一个东西~ 脱水： 服务员，我要的是13岁的噢→\_→

然后服务器会返回下列东西，表示已经接受到请求， 成功建立Websocket啦！

1. HTTP/1.1 101 Switching Protocols
2. Upgrade: websocket
3. Connection: Upgrade
4. Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
5. Sec-WebSocket-Protocol: chat

这里开始就是HTTP最后负责的区域了，告诉客户，我已经成功切换协议啦~

1. Upgrade: websocket
2. Connection: Upgrade

依然是固定的，告诉客户端即将升级的是 Websocket 协议，而不是mozillasocket，lurnarsocket或者shitsocket。

然后， Sec-WebSocket-Accept 这个则是经过服务器确认，并且加密过后的 Sec-WebSocket-Key 。 服务器：好啦好啦，知道啦，给你看我的ID CARD来证明行了吧。。

后面的， Sec-WebSocket-Protocol 则是表示最终使用的协议。

至此，HTTP已经完成它所有工作了，接下来就是完全按照Websocket协议进行了。具体的协议就不在这阐述了。

## websocket协议的作用：

HTTP是一中状态协议，服务器因为每天都要接收很多数据，只要一断开连接，就会把所有数据都丢失，等到第二天连接，又需要重新请求服务器。

而websocket协议的出现解决了HTTP协议的几大难题：

1、被动性。当服务器完成协议更换后（HTTP协议🡪websocket协议），服务端就可以主动地向客户端发送或推送数据了。

客户端：啦啦啦，我要建立Websocket协议，需要的服务：chat，Websocket协议版本：17（HTTP Request）

服务端：ok，确认，已升级为Websocket协议（HTTP Protocols Switched）

客户端：麻烦你有信息的时候推送给我噢。。

服务端：ok，有的时候会告诉你的。

服务端：balabalabalabala

服务端：balabalabalabala

服务端：哈哈哈哈哈啊哈哈哈哈

服务端：笑死我了哈哈哈哈哈哈哈

就变成了这样，只需要经过一次HTTP请求，就可以做到源源不断的信息传送了。（在程序设计中，这种设计叫做回调，即：你有信息了再来通知我，而不是我傻乎乎的每次跑来问你）

2、websocket解决HTTP非常耗费资源：其实我们所用的程序是要经过两层代理的，即HTTP协议在Nginx等服务器的解析下，然后再传送给相应的Handler（PHP等）来处理。简单地说，我们有一个非常快速的 接线员（Nginx） ，他负责把问题转交给相应的 客服（Handler） 。

本身接线员基本上速度是足够的，但是每次都卡在客服（Handler）了，老有客服处理速度太慢。导致客服不够。Websocket就解决了这样一个难题，建立后，可以直接跟接线员建立持久连接，有信息的时候客服想办法通知接线员，然后接线员在统一转交给客户。

这样就可以解决客服处理速度过慢的问题了。

同时，在传统的方式上，要不断的建立，关闭HTTP协议，由于HTTP是非状态性的，每次都要重新传输 identity info （鉴别信息），来告诉服务端你是谁。

虽然接线员很快速，但是每次都要听这么一堆，效率也会有所下降的，同时还得不断把这些信息转交给客服，不但浪费客服的处理时间，而且还会在网路传输中消耗过多的流量/时间。

但是Websocket只需要一次HTTP握手，所以说整个通讯过程是建立在一次连接/状态中，也就避免了HTTP的非状态性，服务端会一直知道你的信息，直到你关闭请求，这样就解决了接线员要反复解析HTTP协议，还要查看identity info的信息。

同时由客户主动询问，转换为服务器（推送）有信息的时候就发送（当然客户端还是等主动发送信息过来的。），没有信息的时候就交给接线员（Nginx），不需要占用本身速度就慢的客服（Handler）了

## WebSocket的实例（推送消息）

### 1、导入依赖包

<!--websocket使用依赖-->

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-websocket</artifactId>  
</dependency>

### 2、创建配置类WebSocket:

@Component

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

### 3、创建实现类WebSocketServiceImpl：

@Component

@ServerEndpoint("/websocket")  
@Slf4j  
public class WebSocketServiceImpl {  
 private Session session;  
 private static CopyOnWriteArrayList<WebSocketServiceImpl> *webSockets* = new CopyOnWriteArrayList<>();  
  
 @OnOpen  
 public void onOpen(Session session){  
 this.session = session;  
 *webSockets*.add(this);  
 *log*.info("websocket有新的连接，总连接数:{}",*webSockets*.size());  
 }  
  
 @OnClose  
 public void onClose(){  
 *webSockets*.remove(this);  
 *log*.info("websocket连接已断开，总连接数为：{}",*webSockets*.size());  
 }  
  
 @OnMessage  
 public void onMessage(String message){  
 *log*.info("从客户端收到的消息:{}", message);  
 }  
  
 public void sendMessage(String message){  
 for (WebSocketServiceImpl webSocket : *webSockets*){  
 *log*.info("websocket广播消息,message={}", message);  
 try {  
 webSocket.session.getBasicRemote().sendText(message);  
 } catch (Exception e){  
 e.printStackTrace();  
 }  
 }  
 }  
  
}

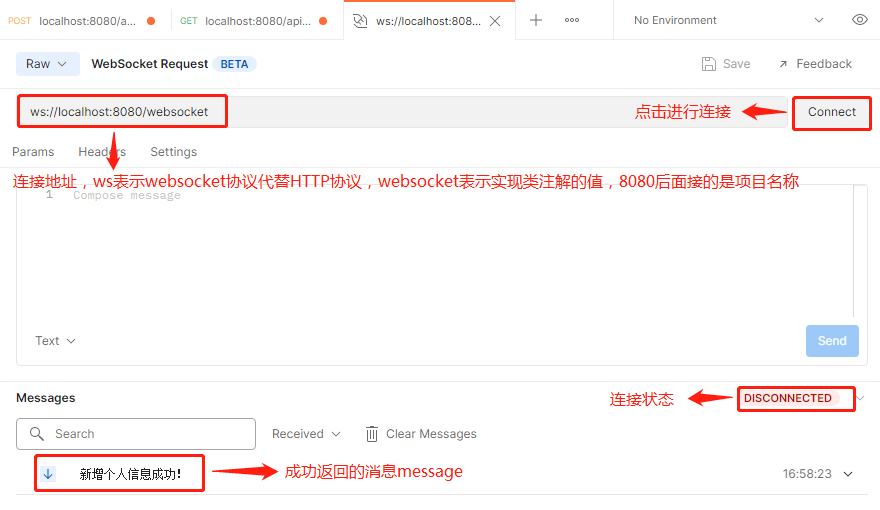
### 4、在Controller里面创建调用：

@Resource

private WebSocketServiceImpl webSocketService;

@PostMapping("/insert")  
 public Response insert(@RequestBody JSONObject res){  
 User user = JSONObject.*toJavaObject*(res,User.class);  
 if (userDao.insert(user) != 0){  
 /\*=====================websocket推送信息的使用======================\*/  
 try {  
 webSocketService.sendMessage("新增个人信息成功！");  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 return new Response();  
 }  
 return new Response("更改失败!");  
 }

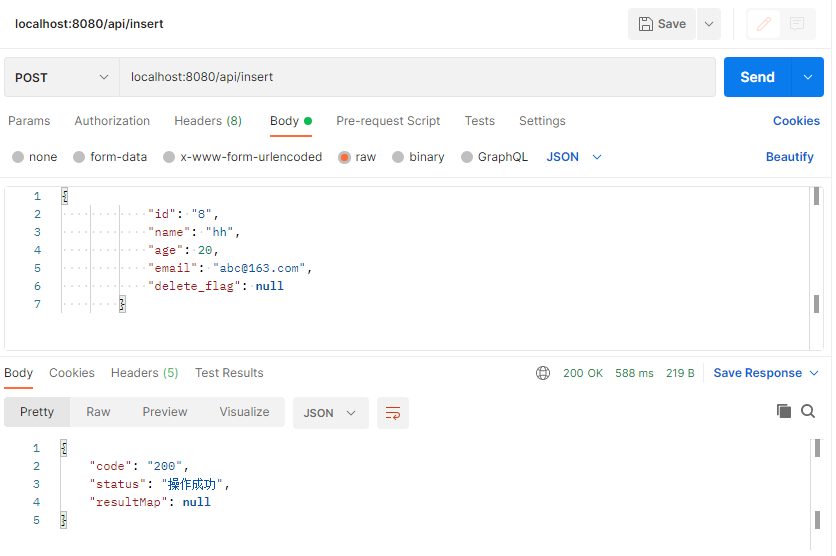
### 5、在postman里面进行websocket连接



连接成功在控制台上打印日志：



### 5、调用localhost:8080/api/insert进行插入数据，成功之后就会返回消息message。



## RedisTemplate操作Redis

Redis的五种数据类型：Redis 可以存储键与5种不同数据结构类型之间的映射，这5种数据结构类型分别为String（字符串）、List（列表）、Set（集合）、Hash（散列）和 Zset（有序集合）。

1. String：可以是字符串、整数或者浮点数
2. List：一个链表，链表上的每个节点都包含了一个字符串
3. Set：包含字符串的无序收集器(unorderedcollection)，并且被包含的每个字符串都是独一无二的、各不相同
4. Hash：包含键值对的无序散列表
5. ZSet：字符串成员(member)与浮点数分值(score)之间的有序映射，元素的排列顺序由分值score的大小决定

常用的操作方法：

1. redisTemplate.opsForValue();*//操作字符串下*

使用：redisTemplate.opsForValue().set(”name”,”Tom”)

结果：redisTemplate.opsForValue().get(”name”)

1. redisTemplate.opsForHash();*//操作hash*
2. redisTemplate.opsForList();*//操作list*
3. redisTemplate.opsForSet();*//操作set*
4. redisTemplate.opsForZSet();*//操作有序set*

*PS：使用之前，先要对RedisTemplate进行序列化，否则在Redis客户端查看不到存储的key数据。*

//序列化RedisTemplate，否则Redis客户端查询不到key

private RedisTemplate redisTemplate;  
@Autowired(required = false)  
public void setRedisTemplate(RedisTemplate redisTemplate) {  
 RedisSerializer stringSerializer = new StringRedisSerializer();  
 redisTemplate.setKeySerializer(stringSerializer);  
 redisTemplate.setValueSerializer(stringSerializer);  
 redisTemplate.setHashKeySerializer(stringSerializer);  
 redisTemplate.setHashValueSerializer(stringSerializer);  
 this.redisTemplate = redisTemplate;  
}

下面是RedisTemplate操作（String、List、Hash、Set和ZSet）五种数据结构的操作：

### 1、String数据结构操作：

/\*=====================Redis的String的操作======================\*/

//储存一个Redis的key:value  
redisTemplate.opsForValue().set("name", "Tom");  
String name = (String) redisTemplate.opsForValue().get("name");  
*log*.info("操作字符串输出:{}", name);  
  
//储存一个Redis的key:value，并且定时删除  
redisTemplate.opsForValue().set("name", "Mark", 10, TimeUnit.*SECONDS*);  
//由于设置的是10秒失效，十秒之内查询有结果，十秒之后返回为null  
String name1 = (String) redisTemplate.opsForValue().get("name");  
//设置定时器来执行输出  
*/\*\*  
 \* Timer：是一个定时器工具，用来执行指定任务  
 \* TimerTask：是一个抽象类，他的子类可以代表一个被Timer计划的任务  
 \*/*TimerTask timerTask = new TimerTask() {  
 @Override  
 public void run() {  
 *log*.info("操作字符串输出，10秒失效:{}", name1);  
 }  
};  
Timer timer = new Timer();  
long delay = 0; //等待时间5秒  
long intevalPeriod = 1000 \* 5; //每次执行时间的间隔  
timer.schedule(timerTask, delay, intevalPeriod); //安排任务在一定时间执行  
  
//用 value 参数覆写(overwrite)给定 key 所储存的字符串值，从偏移量 offset 开始  
redisTemplate.opsForValue().set("key","hello world");  
redisTemplate.opsForValue().set("key", "redis", 6);  
*log*.info("输出结果:{}", redisTemplate.opsForValue().get("key"));  
  
//setIfAbsent()方法是比较key里面的value是否与现在的值相等，相等返回false;否则返回TRUE，并且更新value数据  
*log*.info(String.*valueOf*(redisTemplate.opsForValue().setIfAbsent("multi1","multi1")));  
*log*.info(String.*valueOf*(redisTemplate.opsForValue().setIfAbsent("multi111","multi111")));  
  
//为多个键分别设置它们的值  
Map<String,String> maps = new HashMap<String, String>();  
maps.put("multi1","multi1");  
maps.put("multi2","multi2");  
maps.put("multi3","multi3");  
redisTemplate.opsForValue().multiSet(maps);  
List<String> keys = new ArrayList<String>();  
keys.add("multi1");  
keys.add("multi2");  
keys.add("multi3");  
System.*out*.println(redisTemplate.opsForValue().multiGet(keys));  
  
//为多个键分别设置它们的值，如果存在则返回false，不存在返回true,并且更新value数据  
Map<String, String> map1 = new HashMap<>();  
map1.put("multi11","multi11");  
map1.put("multi22","multi22");  
map1.put("multi33","multi33");  
Map<String,String> maps2 = new HashMap<>();  
maps2.put("multi1","multi1");  
maps2.put("multi2","multi2");  
maps2.put("multi3","multi3");  
System.*out*.println(redisTemplate.opsForValue().multiSetIfAbsent(map1));  
System.*out*.println(redisTemplate.opsForValue().multiSetIfAbsent(maps2));  
  
//设置键的字符串值并返回其旧值  
redisTemplate.opsForValue().set("getSetTest","test");  
System.*out*.println(redisTemplate.opsForValue().getAndSet("getSetTest","test2"));  
System.*out*.println(redisTemplate.opsForValue().get("getSetTest"));  
  
//increment()支持整数、浮点型数据存储  
redisTemplate.opsForValue().increment("increment1", 1);  
redisTemplate.opsForValue().increment("increment2", 1.32);  
  
//如果key已经存在并且是一个字符串，则该命令将该值追加到字符串的末尾。如果键不存在，则它被创建并设置为空字符串，之后再拼接因此APPEND在这种特殊情况下将类似于SET。  
redisTemplate.opsForValue().append("appendTest", "Hello");  
System.*out*.println(redisTemplate.opsForValue().get("appendTest"));  
redisTemplate.opsForValue().append("appendTest","world");  
System.*out*.println(redisTemplate.opsForValue().get("appendTest"));  
  
//遍历截取key所对应的value字符串,-1代表倒数第个元素，0代表第一个元素  
System.*out*.println(redisTemplate.opsForValue().get("appendTest",0,5));//输出结果:Hellow  
System.*out*.println(redisTemplate.opsForValue().get("appendTest",0,-1));//输出结果:Helloworld  
System.*out*.println(redisTemplate.opsForValue().get("appendTest",-3,-1));//输出结果:rld  
System.*out*.println(redisTemplate.opsForValue().get("appendTest", -1, -5));//没有输出结果  
  
//返回key所对应的value值得长度  
System.*out*.println(redisTemplate.opsForValue().size("appendTest"));  
  
//setBit()方法是对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)，key键对应的值value对应的ascii码,在offset的位置(从左向右数)变为value  
redisTemplate.opsForValue().set("bitTest","a");  
// 'a' 的ASCII码是 97。转换为二进制是：01100001  
// 'b' 的ASCII码是 98 转换为二进制是：01100010  
// 'c' 的ASCII码是 99 转换为二进制是：01100011  
//因为二进制只有0和1，在setbit中true为1，false为0，因此我要变为'b'的话第六位设置为1，第七位设置为0  
redisTemplate.opsForValue().setBit("bitTest",6, true);//第一位参数是键；第二位参数是二进制的第几位，从0开始；第三位参数是设置修改值，true为1，false为0  
redisTemplate.opsForValue().setBit("bitTest",7, false);  
System.*out*.println(redisTemplate.opsForValue().get("bitTest")); //输出: b  
  
//getBit()方法是获取键对应值的ascii码的在offset处位值，TRUE代表1，false代表0  
System.*out*.println(redisTemplate.opsForValue().getBit("bitTest",7));

***List：***

/\*=====================Redis的List结构======================\*/

//leftPush()方法从左添加List数据，range()方法遍历List数据  
redisTemplate.opsForList().leftPush("lPushList","a");  
redisTemplate.opsForList().leftPush("lPushList","b");  
redisTemplate.opsForList().leftPush("lPushList","c");  
redisTemplate.opsForList().leftPush("lPushList","d");  
//输出结果： 遍历lPushList的值:[d, c, b, a]；原因：从左边添加数据，已添加的需向右移  
*log*.info("遍历lPushList的值:{}", redisTemplate.opsForList().range("lPushList", 0, -1));  
  
//rightPush()方法从右添加List数据，range()方法遍历List数据  
redisTemplate.opsForList().rightPush("rPushList","a");  
redisTemplate.opsForList().rightPush("rPushList","b");  
redisTemplate.opsForList().rightPush("rPushList","c");  
redisTemplate.opsForList().rightPush("rPushList","d");  
//输出结果：遍历rPushList的值:[a, b, c, d]；原因：从右边添加数据，已添加的需向左移  
*log*.info("遍历rPushList的值:{}", redisTemplate.opsForList().range("rPushList", 0, -1));  
  
//trim()方法截取集合元素长度，保留长度内的数据。起始和停止都是基于0的索引  
*log*.info(String.*valueOf*(redisTemplate.opsForList().range("rPushList", 0, -1)));//输出结果：[a, b, c, d]  
redisTemplate.opsForList().trim("rPushList", 0, 2);  
*log*.info(String.*valueOf*(redisTemplate.opsForList().range("rPushList", 0, -1)));//输出结果：[a, b, c]  
  
//size()方法返回存储在键中的列表的长度。如果键不存在，则将其解释为空列表，并返回0。当key存储的值不是列表时返回错误。  
System.*out*.println(redisTemplate.opsForList().size("rPushList"));//rPushList里面的值为：[a, b, c]；输出结果：3  
System.*out*.println(redisTemplate.opsForList().size("appendTest"));//appendTest不是列表；编译出现错误，报红：Error in execution; nested exception is io.lettuce.core.RedisCommandExecutionException  
  
//leftPushAll()方法是批量把一个数组插入到列表中  
String[] stringArrays = new String[]{"1","2","3"};  
redisTemplate.opsForList().leftPushAll("listArray",stringArrays);  
System.*out*.println(redisTemplate.opsForList().range("listArray",0,-1));//输出结果:[3, 2, 1]  
  
//leftPushAll()方法还可以批量把一个集合插入到列表中  
List<Object> strings = new ArrayList<>();  
strings.add("1");  
strings.add("2");  
strings.add("3");  
redisTemplate.opsForList().leftPushAll("listCollection", strings);  
System.*out*.println(redisTemplate.opsForList().range("listCollection",0,-1));//输出结果：[3, 2, 1]  
  
//leftPushIfPresent()方法是只有存在key对应的列表才能将这个value值插入到key所对应的列表中  
System.*out*.println(redisTemplate.opsForList().leftPushIfPresent("leftPushIfPresent","aa"));//输出结果：0  
System.*out*.println(redisTemplate.opsForList().leftPushIfPresent("leftPushIfPresent","bb"));//输出结果：0  
System.*out*.println("不存在leftPushIfPresent的key，输出结果：" + redisTemplate.opsForList().range("leftPushIfPresent", 0, -1));//输出结果：不存在leftPushIfPresent的key，输出结果：[]  
/\*=====================分割线======================\*/  
System.*out*.println(redisTemplate.opsForList().leftPush("leftPushIfPresent","aa"));//输出结果：1  
System.*out*.println(redisTemplate.opsForList().leftPushIfPresent("leftPushIfPresent","bb"));//输出结果：2  
System.*out*.println("存在leftPushIfPresent的key，输出结果：" + redisTemplate.opsForList().range("leftPushIfPresent", 0, -1));//输出结果：存在leftPushIfPresent的key，输出结果：[bb, aa]  
  
//Long leftPush(K key, V pivot, V value)作用是把value值放到key对应列表中pivot值的左面，如果pivot值存在的话  
ArrayList arrayList = new ArrayList();  
arrayList.add("C++");  
arrayList.add("C#");  
arrayList.add("Java");  
arrayList.add("Python");  
redisTemplate.opsForList().leftPushAll("list",arrayList);  
System.*out*.println(redisTemplate.opsForList().range("list", 0, -1));//输出结果：[Python, Java, C#, C++]  
redisTemplate.opsForList().leftPush("list","Java","oc");  
System.*out*.print(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, oc, Java, C#, C++]  
  
//Long rightPush(K key, V pivot, V value)作用是把value值放到key对应列表中pivot值的右面，如果pivot值存在的话  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, oc, Java, C#, C++]  
redisTemplate.opsForList().rightPush("list","C#","oc");  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, oc, Java, C#, oc, C++]  
  
//void set(K key, long index, V value)的作用是在列表中index的位置设置value值，有点类似与更新  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, oc, Java, C#, oc, C++]  
redisTemplate.opsForList().set("list",1,"setValue");  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, setValue, Java, C#, oc, C++]  
  
//Long remove(K key, long count, Object value)的作用是从存储在键中的列表中删除等于值的元素的第一个计数事件。  
//计数参数以下列方式影响操作：  
//count> 0：删除等于从头到尾移动的值的元素。  
//count <0：删除等于从尾到头移动的值的元素。  
//count = 0：删除等于value的所有元素。  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, setValue, Java, C#, oc, C++]  
redisTemplate.opsForList().remove("list",1,"setValue");//将删除列表中存储的list列表中第一次次出现的“setValue”。  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, Java, C#, oc, C++]  
  
//V index(K key, long index)作用是根据下表获取列表中的值，下标是从0开始的  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, Java, C#, oc, C++]  
System.*out*.println(redisTemplate.opsForList().index("list",2));//输出结果：C#  
  
//V leftPop(K key)的作用是弹出最左边的元素，弹出之后该值在列表中将不复存在  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Python, Java, C#, oc, C++]  
System.*out*.println(redisTemplate.opsForList().leftPop("list"));//输出结果：Python  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Java, C#, oc, C++]  
  
//V rightPop(K key)的作用是弹出最右边的元素，弹出之后该值在列表中将不复存在  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Java, C#, oc, C++]  
System.*out*.println(redisTemplate.opsForList().rightPop("list"));//输出结果：C++  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果哟：[Java, C#, oc]  
  
//V rightPopAndLeftPush(K sourceKey, K destinationKey)的作用是用于移除列表的最后一个元素，并将该元素添加到另一个列表并返回。  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Java, C#, oc]  
redisTemplate.opsForList().rightPopAndLeftPush("list","rightPopAndLeftPush");  
System.*out*.println(redisTemplate.opsForList().range("list",0,-1));//输出结果：[Java, C#]  
System.*out*.println(redisTemplate.opsForList().range("rightPopAndLeftPush",0,-1));//输出结果：[oc]

### 2、Hash数据结构操作：

/\*=====================Redis的Hash数据结构======================\*/

//put()设置插入Hash，putAll()设置批量插入Hash；Map<HK, HV> entries(H key)根据密钥获取整个哈希存储的数据，返回的是Map数据  
redisTemplate.opsForHash().put("redisHash","name","tom");  
redisTemplate.opsForHash().put("redisHash","age","26");  
redisTemplate.opsForHash().put("redisHash", "class", "6");  
redisTemplate.opsForHash().put("redisHash","email","abc@qq.com");  
Map<String,Object> testMap = new HashMap();  
testMap.put("name","jack");  
testMap.put("age","27");  
testMap.put("class", "1");  
testMap.put("email","abc@163.com");  
redisTemplate.opsForHash().putAll("redisHash1",testMap);  
System.*out*.println("redisHash的值" + redisTemplate.opsForHash().entries("redisHash"));//输出结果：redisHash的值{name=tom, age=26, class=6, email=abc@qq.com}  
System.*out*.println("redisHash1的值" + redisTemplate.opsForHash().entries("redisHash1"));//输出结果：redisHash1的值{name=jack, class=1, age=27, email=abc@163.com}  
  
//Long delete(H key, Object... hashKeys)的作用是删除给定的哈希hashKeys  
System.*out*.println(redisTemplate.opsForHash().delete("redisHash","class"));//输出结果：1  
System.*out*.println(redisTemplate.opsForHash().entries("redisHash"));//输出结果：{name=tom, age=26, email=abc@qq.com}  
  
//Boolean hasKey(H key, Object hashKey)的作用是确定哈希hashKey是否存在，存在输出true，不存在输出false  
System.*out*.println(redisTemplate.opsForHash().hasKey("redisHash", "age"));//输出结果：true  
System.*out*.println(redisTemplate.opsForHash().hasKey("redisHash", "abc"));//输出结果：false  
  
//HV get(H key, Object hashKey)的作用是从键中的哈希获取给定hashKey的值  
System.*out*.println(redisTemplate.opsForHash().get("redisHash","email"));//输出结果：abc@qq.com  
  
//List<HV> multiGet(H key, Collection<HK> hashKeys)的作用是从哈希中获取给定hashKey的值，返回的是List  
ArrayList keys1 = new ArrayList();  
keys.add("name");  
keys.add("age");  
keys.add("email");  
System.*out*.println(redisTemplate.opsForHash().multiGet("redisHash", keys1));//输出结果：[tom, 26, abc@qq.com]  
  
//Long(或者Double) increment(H key, HK hashKey, long delta)的作用是通过给定的delta增加散列hashKey的值（整型或浮点型），相当于加法与减法  
System.*out*.println(redisTemplate.opsForHash().get("redisHash","age"));//输出结果：26  
System.*out*.println(redisTemplate.opsForHash().increment("redisHash","age",10));//输出结果：36  
System.*out*.println(redisTemplate.opsForHash().increment("redisHash","age",-10));//输出结果：26  
System.*out*.println(redisTemplate.opsForHash().increment("redisHash","age",-2.5));//输出结果：23.5  
  
//Set<HK> keys(H key)的作用是获取Hash中所对应的散列表的key  
System.*out*.println(redisTemplate.opsForHash().entries("redisHash1"));//输出结果：{name=jack, class=1, age=27, email=abc@163.com}  
System.*out*.println(redisTemplate.opsForHash().keys("redisHash1"));//输出结果：[name, class, age, email]  
  
//Long size(H key)的作用是获取key所对应的散列表的大小个数  
System.*out*.println(redisTemplate.opsForHash().entries("redisHash1"));//输出结果：{name=jack, class=1, age=27, email=abc@163.com}  
System.*out*.println(redisTemplate.opsForHash().size("redisHash1"));//输出结果：4  
  
//Boolean putIfAbsent(H key, HK hashKey, HV value)的作用是如果hashKey不存在的时候，返回TRUE并且设置散列hashKey的值，否则不做更改，并且返回false  
System.*out*.println(redisTemplate.opsForHash().putIfAbsent("redisHash","age","20"));//输出结果：false  
System.*out*.println(redisTemplate.opsForHash().entries("redisHash"));//输出结果：{name=tom, age=33.5, email=abc@qq.com}  
System.*out*.println(redisTemplate.opsForHash().putIfAbsent("redisHash","kkk","kkk"));//输出结果：true  
System.*out*.println(redisTemplate.opsForHash().entries("redisHash"));//输出结果：{name=tom, age=33.5, email=abc@qq.com, kkk=kkk}  
  
//List<HV> values(H key)的作用是根据密钥获取整个哈希存储的值  
System.*out*.println(redisTemplate.opsForHash().values("redisHash"));//输出结果：[tom, 33.5, abc@qq.com, kkk]  
  
//Cursor<Map.Entry<HK, HV>> scan(H key, ScanOptions options)的作用是使用Cursor在key的hash中迭代，相当于迭代器。  
Cursor<Map.Entry<Object, Object>> cursor = redisTemplate.opsForHash().scan("redisHash", ScanOptions.*NONE*);  
while(cursor.hasNext()){  
 Map.Entry<Object, Object> entry = cursor.next();  
 System.*out*.print(entry.getKey() + ":" + entry.getValue() + "\t");//输出结果：name:tom age:33.5 email:abc@qq.com kkk:kkk  
}

### 3、Set数据结构操作：

/\*=====================Redis的Set数据结构======================\*/

//Long add(K key, V... values)的作用是无序集合中添加元素，返回添加个数;也可以直接在add里面添加多个值 如：template.opsForSet().add("setTest","aaa","bbb")  
String[] strings0 = {"yes", "no", "ok", "bad"};  
String[] strings1 = {"哇", "哦", "嗯", "哈"};  
System.*out*.println(redisTemplate.opsForSet().add("setTest", strings0));//输出结果：4  
redisTemplate.opsForSet().add("setTest1", strings1);  
  
//Long remove(K key, Object... values)的作用是移除集合中一个或多个成员  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[no, yes]  
System.*out*.println(redisTemplate.opsForSet().remove("setTest","no"));//输出结果：1  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[yes]  
  
//V pop(K key)的作用是移除并返回集合中的一个随机元素  
System.*out*.println(redisTemplate.opsForSet().pop("setTest"));//原来的数据：[yes]；输出结果：yes  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[]  
  
//Boolean move(K key, V value, K destKey)的作用是将 member 元素从 source 集合移动到 destination 集合  
redisTemplate.opsForSet().move("setTest","yes","setTest1");  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().members("setTest1"));//输出结果：[嗯, 哦, 哈, 哇, yes]  
  
//Boolean isMember(K key, Object o)的作用是判断 member 元素是否是集合 key 的成员  
System.*out*.println(redisTemplate.opsForSet().isMember("setTest", "bad"));//输出结果：true  
  
//Set<V> intersect(K key, K otherKey)的作用是key对应的无序集合与otherKey对应的无序集合求交集  
//Set<V> intersect(K key, Collection<K> otherKeys)的作用是key对应的无序集合与多个otherKey对应的无序集合求交集，这里就不展示了(先申请一个ArrayList，之后add)  
redisTemplate.opsForSet().add("setTest", "交集");  
redisTemplate.opsForSet().add("setTest1", "交集");  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[交集, no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().members("setTest1"));//输出结果：[哈, 哇, 交集, 哦, yes, 嗯]  
System.*out*.println(redisTemplate.opsForSet().intersect("setTest", "setTest1"));//输出结果：[交集]  
  
//Long intersectAndStore(K key, K otherKey, K destKey)的作用是key无序集合与otherkey无序集合的交集存储到destKey无序集合中  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[交集, no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().members("setTest1"));//输出结果：[交集, 哦, 哈, yes, 哇, 嗯]  
redisTemplate.opsForSet().intersectAndStore("setTest", "setTest1", "and");  
System.*out*.println(redisTemplate.opsForSet().members("and"));//输出结果：[交集]  
  
//Set<V> union(K key, K otherKey)的作用是key无序集合与otherKey无序集合的并集  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[交集, no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().members("setTest1"));//输出结果：[交集, 哦, 哈, yes, 哇, 嗯]  
System.*out*.println(redisTemplate.opsForSet().union("setTest", "setTest1"));//输出结果：  
  
//Long unionAndStore(K key, K otherKey, K destKey)的作用是key无序集合与otherkey无序集合的并集存储到destKey无序集合中，返回值是key与otherkey的并集中个数  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[交集, no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().members("setTest1"));//输出结果：[交集, 哦, 哈, yes, 哇, 嗯]  
System.*out*.println(redisTemplate.opsForSet().unionAndStore("setTest", "setTest1", "or"));//输出结果：9  
System.*out*.println(redisTemplate.opsForSet().members("or"));//输出结果：[哦, 交集, no, 哈, yes, bad, 哇, ok, 嗯]  
  
//Set<V> difference(K key, K otherKey)的作用是key无序集合与otherKey无序集合的差集  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[交集, no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().members("setTest1"));//输出结果：[交集, 哦, 哈, yes, 哇, 嗯]  
System.*out*.println(redisTemplate.opsForSet().difference("setTest", "setTest1"));//输出结果：[no, bad, ok]  
  
//Long differenceAndStore(K key, K otherKey, K destKey)的作用是key无序集合与otherkey无序集合的差集存储到destKey无序集合中，返回值是key与otherkey的差集中个数  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[交集, no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().members("setTest1"));//输出结果：[交集, 哦, 哈, yes, 哇, 嗯]  
System.*out*.println(redisTemplate.opsForSet().differenceAndStore("setTest", "setTest1", "different"));//输出结果：3  
System.*out*.println(redisTemplate.opsForSet().members("different"));//输出结果：[no, bad, ok]  
  
//V randomMember(K key)的作用是随机获取key无序集合中的一个元素；  
//List<V> randomMembers(K key, long count)获取多个key无序集合中的元素，count表示个数  
//Set<V> distinctRandomMembers(K key, long count)的作用是获取多个key无序集合中的元素（去重），count表示个数  
System.*out*.println(redisTemplate.opsForSet().members("setTest"));//输出结果：[交集, no, bad, ok]  
System.*out*.println(redisTemplate.opsForSet().randomMember("setTest"));//输出结果：交集  
System.*out*.println(redisTemplate.opsForSet().randomMembers("setTest", 3));//输出结果：[bad, 交集, bad]  
System.*out*.println(redisTemplate.opsForSet().distinctRandomMembers("setTest", 3));//输出结果：[bad, 交集, ok]  
  
//Cursor<V> scan(K key, ScanOptions options)的作用是遍历set  
Cursor<Object> cursor1 = redisTemplate.opsForSet().scan("setTest", ScanOptions.*NONE*);  
while (cursor1.hasNext()){  
 System.*out*.print(cursor1.next() + "\t");//输出结果：交集 no bad ok  
}

### 4、ZSet数据结构操作：

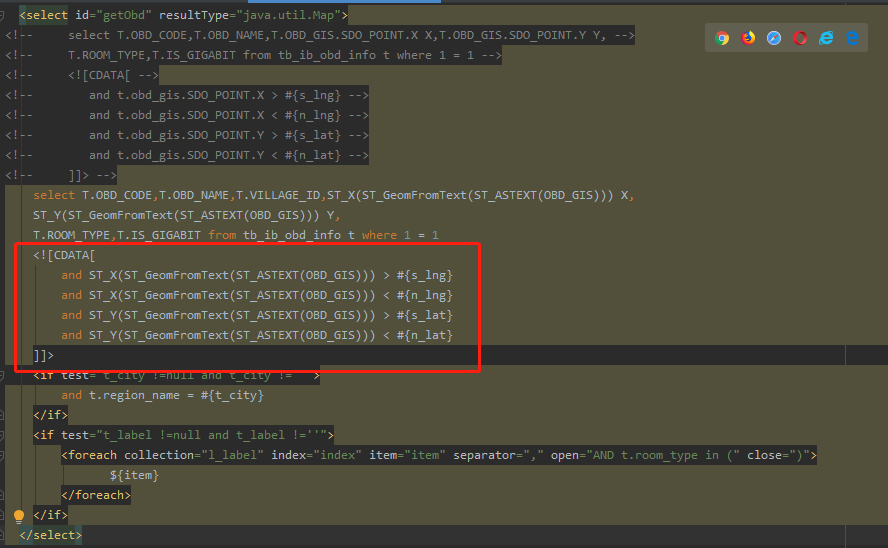
/\*=====================Redis的ZSet数据结构======================\*/

//Boolean add(K key, V value, double score)的作用是新增一个有序集合，存在的话为false，不存在的话为true  
System.*out*.println(redisTemplate.opsForZSet().add("zSet1", "zSet->1", 1.0));//输出结果：true  
  
//Long add(K key, Set<TypedTuple<V>> tuples)新增一个有序集合  
//Set<V> range(K key, long start, long end)通过索引区间返回有序集合成指定区间内的成员，其中有序集成员按分数值递增(从小到大)顺序排列  
ZSetOperations.TypedTuple<Object> objectTypedTuple1 = new DefaultTypedTuple<>("zSet->2", 2.0);  
ZSetOperations.TypedTuple<Object> objectTypedTuple2 = new DefaultTypedTuple<>("zSet->3", 3.0);  
Set<ZSetOperations.TypedTuple<Object>> tuples = new HashSet<>();  
tuples.add(objectTypedTuple1);  
tuples.add(objectTypedTuple2);  
System.*out*.println(redisTemplate.opsForZSet().add("zSet1", tuples));//输出结果：2  
System.*out*.println(redisTemplate.opsForZSet().range("zSet1", 0, -1));//输出结果：[zSet->1, zSet->2, zSet->3]  
  
//Long remove(K key, Object... values)从有序集合中移除一个或者多个元素  
System.*out*.println(redisTemplate.opsForZSet().range("zSet1",0,-1));//输出结果：[zSet->1, zSet->2, zSet->3]  
System.*out*.println(redisTemplate.opsForZSet().remove("zSet1","zSet->3"));//输出结果：1  
System.*out*.println(redisTemplate.opsForZSet().range("zSet1",0,-1));//输出结果：[zSet->1, zSet->2]  
  
//Double incrementScore(K key, V value, double delta)的作用是增加元素的score值，并返回增加后的值  
System.*out*.println(redisTemplate.opsForZSet().incrementScore("zSet1", "zSet->2", 2.3));//原来的score的值为2.0；输出结果：4.3  
  
//Long rank(K key, Object o)的作用是返回有序集中指定成员的排名，其中有序集成员按分数值递增(从小到大)顺序排列  
//Long reverseRank(K key, Object o)的作用是返回有序集中指定成员的排名，其中有序集成员按分数值递减(从大到小)顺序排列  
System.*out*.println(redisTemplate.opsForZSet().range("zSet1",0,-1));//输出结果：[zSet->1, zSet->3, zSet->2]  
System.*out*.println(redisTemplate.opsForZSet().rank("zSet1","zSet->1"));//输出结果：0  
System.*out*.println(redisTemplate.opsForZSet().reverseRank("zSet1","zSet->1"));//输出结果：2  
  
//Set<TypedTuple<V>> rangeWithScores(K key, long start, long end)的作用是通过索引区间返回有序集合成指定区间内的成员对象，其中有序集成员按分数值递增(从小到大)顺序排列  
Set<ZSetOperations.TypedTuple<Object>> tupleSet = redisTemplate.opsForZSet().rangeWithScores("zSet1", 0, -1);  
Iterator<ZSetOperations.TypedTuple<Object>> iterator = tupleSet.iterator();  
while (iterator.hasNext()){  
 ZSetOperations.TypedTuple<Object> operations = iterator.next();  
 System.*out*.println("value:" + operations.getValue() + "\t" + "score:" + operations.getScore());  
 */\*\*输出结果：  
 \* value:zSet->1 score:1.0  
 \* value:zSet->3 score:3.0  
 \* value:zSet->2 score:4.3  
 \*/*}  
  
//Set<V> rangeByScore(K key, double min, double max)通过score分数返回有序集合指定区间内的成员，其中有序集成员按分数值递增(从小到大)顺序排列  
System.*out*.println(redisTemplate.opsForZSet().rangeByScore("zSet1",0,2));//输出结果：[zSet->1]  
  
//Set<TypedTuple<V>> rangeByScoreWithScores(K key, double min, double max)通过分数返回有序集合指定区间内的成员对象，其中有序集成员按分数值递增(从小到大)顺序排列  
Set<ZSetOperations.TypedTuple<Object>> tuples1 = redisTemplate.opsForZSet().rangeByScoreWithScores("zSet1",0,5);  
Iterator<ZSetOperations.TypedTuple<Object>> iterator1 = tuples1.iterator();  
while (iterator1.hasNext()) {  
 ZSetOperations.TypedTuple<Object> typedTuple = iterator1.next();  
 System.*out*.println("value:" + typedTuple.getValue() + "\t" + "score:" + typedTuple.getScore());  
 */\*\*  
 \* 输出结果：  
 \* value:zSet->1 score:1.0  
 \* value:zSet->3 score:3.0  
 \* value:zSet->2 score:4.3  
 \*/*}  
  
//Long count(K key, double min, double max)通过分数返回有序集合指定区间内的成员个数  
System.*out*.println(redisTemplate.opsForZSet().rangeByScore("zSet1",0,5));//输出结果：[zSet->1, zSet->3, zSet->2]  
System.*out*.println(redisTemplate.opsForZSet().count("zSet1",0,5));//输出结果：3  
  
//Long size(K key)获取有序集合的成员数  
//Long zCard(K key)获取有序集合的成员数  
System.*out*.println(redisTemplate.opsForZSet().size("zSet1"));//输出结果：3  
System.*out*.println(redisTemplate.opsForZSet().zCard("zSet1"));//输出结果：3  
  
//Double score(K key, Object o)获取指定成员的score值  
System.*out*.println(redisTemplate.opsForZSet().score("zSet1","zSet->1"));//输出结果：1.0  
  
//Long removeRange(K key, long start, long end)移除指定索引位置的成员，其中有序集成员按分数值递增(从小到大)顺序排列  
System.*out*.println(redisTemplate.opsForZSet().removeRange("zSet1",1,2));  
  
//Long removeRangeByScore(K key, double min, double max)根据指定的score值得范围来移除成员  
  
//Long unionAndStore(K key, K otherKey, K destKey)计算给定的一个有序集的并集，并存储在新的 destKey中，key中的value相同的话会把score值相加  
redisTemplate.opsForZSet().add("zzset1","zset->1",1.0);  
redisTemplate.opsForZSet().add("zzset1","zset->2",2.0);  
redisTemplate.opsForZSet().add("zzset1","zset->3",3.0);  
redisTemplate.opsForZSet().add("zzset1","zset->4",4.0);  
  
redisTemplate.opsForZSet().add("zzset2","zset->1",1.0);  
redisTemplate.opsForZSet().add("zzset2","zset->2",2.0);  
redisTemplate.opsForZSet().add("zzset2","zset->3",3.0);  
redisTemplate.opsForZSet().add("zzset2","zset->4",4.0);  
redisTemplate.opsForZSet().add("zzset2","zset->5",5.0);  
redisTemplate.opsForZSet().unionAndStore("zzset1","zzset2","destZSet11");//输出结果：5  
  
Set<ZSetOperations.TypedTuple<Object>> tuples2 = redisTemplate.opsForZSet().rangeWithScores("destZSet11",0,-1);  
Iterator<ZSetOperations.TypedTuple<Object>> iterator2 = tuples2.iterator();  
while (iterator2.hasNext())  
{  
 ZSetOperations.TypedTuple<Object> typedTuple = iterator2.next();  
 System.*out*.println("value:" + typedTuple.getValue() + "\t" + "score:" + typedTuple.getScore());  
 */\*\*  
 \* 输出结果：  
 \* value:zset->1 score:2.0  
 \* value:zset->2 score:4.0  
 \* value:zset->5 score:5.0  
 \* value:zset->3 score:6.0  
 \* value:zset->4 score:8.0  
 \*/*}  
  
//Long intersectAndStore(K key, K otherKey, K destKey)的作用是计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中  
System.*out*.println(redisTemplate.opsForZSet().intersectAndStore("zzset1", "zzset2", "destZSet2"));//输出结果：4  
Set<ZSetOperations.TypedTuple<Object>> objectTypedTuple = redisTemplate.opsForZSet().rangeWithScores("destZSet2", 0, -1);  
Iterator<ZSetOperations.TypedTuple<Object>> iterator3 = objectTypedTuple.iterator();  
while (iterator3.hasNext()){  
 ZSetOperations.TypedTuple<Object> tuples3 = iterator3.next();  
 System.*out*.println("value:" + tuples3.getValue() + "\t" + "score:" + tuples3.getScore());  
 */\*\*  
 \* 输出结果：  
 \* value:zset->1 score:2.0  
 \* value:zset->2 score:4.0  
 \* value:zset->3 score:6.0  
 \* value:zset->4 score:8.0  
 \*/*}  
  
//Cursor<TypedTuple<V>> scan(K key, ScanOptions options)遍历zset  
Cursor<ZSetOperations.TypedTuple<Object>> cursor2 = redisTemplate.opsForZSet().scan("zzset1", ScanOptions.*NONE*);  
while (cursor2.hasNext()){  
 ZSetOperations.TypedTuple<Object> tuple2 = cursor2.next();  
 System.*out*.println(tuple2.getValue() + ":\t" + tuple2.getScore());  
 */\*\*  
 \* 输出结果：  
 \* zset->1: 1.0  
 \* zset->2: 2.0  
 \* zset->3: 3.0  
 \* zset->4: 4.0  
 \*/*}

# 第三周：

## 看项目代码中的\*\*Mapper.xml：

***1、<![CDATA[]]> 的使用：***在mybatis的xml文件中，“>”、“<”等字符都需要进行转义，否则就会出现SQL语句出错。使用<![CDATA[]]>就可以在里面直接编写>、<等字符。



被<![CDATA[]]>这个标记所包含的内容将表示为**纯文本**，比如<![CDATA[<]]>表示文本内容“<”。   
　　此标记用于xml文档中，我们先来看看使用转义符的情况。我们知道，在xml中，”<”、”>”、”&”等字符是不能直接存入的，否则xml语法检查时会**报错**，如果想在xml中使用这些符号，必须将其转义为实体，如”&lt;”、”&gt;”、”&amp;”，这样才能保存进xml文档。

在XML中，需要转义的字符有：   
　　(1)&　　　&amp;   
　　(2)<　　　&lt;   
　　(3)>　　　&gt;   
　　(4)＂　　　&quot;   
　　(5)＇　　　&apos;

不管怎么样，转义前的字符也好，转义后的字符也好，都会被xml解析器解析，**为了方便起见**，使用<![CDATA[]]>来包含不被xml解析器解析的内容。但要注意的是：   
　　(1) 此部分不能再包含”]]>”；   
　　(2) 不允许嵌套使用；   
　　(3)”]]>”这部分不能包含空格或者换行。   
　　最后，说说<![CDATA[]]>和xml转移字符的关系，它们两个看起来是不是感觉功能重复了？   
　　是的，它们的功能就是一样的，只是应用场景和需求有些不同：   
　　(1)<![CDATA[]]>不能适用所有情况，转义字符可以；   
　　(2) 对于短字符串<![CDATA[]]>写起来啰嗦，对于长字符串转义字符写起来可读性差；   
　　(3) <![CDATA[]]>表示xml解析器忽略解析，所以更快。

***mybatis中<![CDATA[   ]]> 的作用：***

       在使用mybatis 时我们sql是写在xml 映射文件中，如果写的sql中有一些特殊的字符的话，在解析xml文件的时候会被转义，但我们不希望他被转义，所以我们要使用<![CDATA[ ]]>来解决。

<![CDATA[   ]]> 是什么，这是XML语法。在CDATA内部的所有内容都会被解析器忽略。

如果文本包含了很多的"<"字符 <=和"&"字符——就象程序代码一样，那么最好把他们都放到CDATA部件中。

但是有个问题那就是 <if test="">   </if>   <where>   </where>  <choose>  </choose>  <trim>  </trim> 等这些标签都不会被解析，所以我们只把有特殊字符的语句放在 <![CDATA[   ]]>  尽量缩小 <![CDATA[  ]]> 的范围。

***2、空间数据操作（GIS）（学习网址：http://postgis.net/docs/reference.html）***

GIS开发对数据的操作，主要是对空间数据的操作。PostgreSql提供了一些处理空间数据的函数，操作起来是比较方便的。

空间数据是指用来表示空间实体的位置、形状、大小及其分布特征诸多方面信息的数据，它可以用来描述来自现实世界的目标，它具有定位、定性、时间和空间关系等特性。空间数据是一种用点、线、面以及实体等基本空间数据结构来表示人们赖以生存的自然世界的数据。简单来说，操作空间数据主要是对点、线、面等基本结构的操作。

这些空间数据可以在数据库中存储，存储时可以是geometry格式；也可以以文件的形式存储，存储成shape文件。在数据库中存储时，geometry是以十六进制串组成的，表示的是几何形状。shape文件存储时，对应点线面分别由三种类型，point、line、polygon，这是单一的类型，对应联合的类型又有multipoint、multiline、multipolygon，所以说，shape文件存储的是wkt类型的数据。暂且把空间数据分为这两种类型：geometry和wkt。

***操作函数：***

在系统中我们用的地图插件是Openlayer，它对地图的操作是操作wkt数据，所以就涉及到geometry和wkt的转换。

st\_geomfromtext(wkt,坐标系参数)：该方法是把wkt格式数据转换成geometry类型，参数有两个，第一个是wkt类型的数据，第二个是坐标系参数。

st\_astext(geometry)：该方法是把geometry类型数据转换成wkt格式数据，参数只有一个，是geometry类型的数据。

图层是由各个图斑构成的，如果要在地图上定位某个图斑，就需要获取这个图斑的中心点坐标。

st\_centroid(geometry)：这个方法是获取这个geometry的中心点，参数是要获取中心点的geometry。

st\_x(point)：该方法是获取点的x坐标，它操作的对象是一个点。

st\_y(point)：该方法是获取点的y坐标，它操作的对象也是一个点。

有时我们需要实现两个图斑相交，也可以用函数执行。

st\_intersects(geometry,geometry)：该方法是实现两个图斑相交，传入的参数是两个图斑的geometry。

跟这个方法类似的还有好多函数，比如判断两个图斑是否相交，返回一个布尔值；或者获取两个图斑相交的geometry和不相交的geometry。

如果要使图斑联合，则需要union函数：

st\_union(geometry)：如果一个图层表中存储好多图斑，其中有一列名为geom，存储的是各个图斑的geometry，那么st\_union(geom)就是将该图层的所有图斑联合成一个，得到的结果就是multi类型的数据。

如果要查看某个图斑方圆多少米之内的情况，就要对该图斑进行缓冲。

st\_buffer(geometry,长度)：该方法是对某个图斑进行缓冲，传入要缓冲的geometry，在传入缓冲的距离即可。

当ST\_Contains - 返回true时，当且仅在B中的所有点都在 A中，并且至少有一个B中的点位于A中。

st\_distance查询两个经纬度点的距离

***3、distinct作用于表字段，主要是为了去重。***

***PS：distinct必须放在开头，比如select id, distinct name from A;会提示错误***

聚合函数count()使用distinct：

select count(distinct name) from A; --表中name去重后的数目， SQL Server支持，而Access不支持

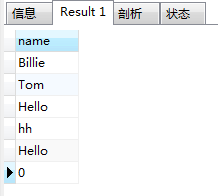
count是不能统计多个字段的，下面的SQL在SQL Server和Access中都无法运行。

select count(distinct name, id) from A;

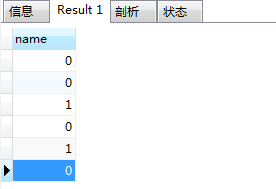
若想使用，请使用嵌套查询，如下：

select count(\*) from (select distinct xing, name from B) AS M;

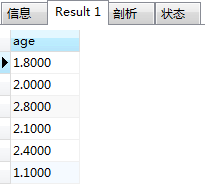
4、select if（a is null, 0, name）as name代表如果在字段name中有值为null的，会把空值变为0查询出来。



select if（a is null, 0, name）= ‘Hello’ as name的结果如下：表示如果在name字段中出现‘Hello’这个值，就将其表示为1查询出来，否则如果值不是‘Hello’，那么就会变成0查询出来。

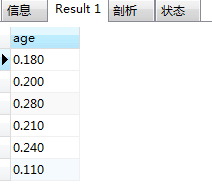


select if(age is null,"-",age/10) as age from user第三个参数代表执行的操作



select if(age is null,"-",FORMAT(`age`/100,3)) as age from user

在Format()中，第一个是字段名称，第二个是保留多少位小数。



## 学习swagger：

### 1、导入依赖包：

<!--swagger的依赖包-->

<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger2</artifactId>  
 <version>2.7.0</version>  
</dependency>  
<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger-ui</artifactId>  
 <version>2.7.0</version>  
</dependency>

### 2、配置配置类

@Configuration

@EnableSwagger2  
public class SwaggerConfig {  
 @Bean  
 public Docket createRestApi(){  
 return new Docket(DocumentationType.*SWAGGER\_2*) //主要api配置机制初始化为swagger规范2.0  
 .apiInfo(apiInfo())  
 .select().apis(RequestHandlerSelectors.*basePackage*("com.chen.mybatis\_plus.controller")) //扫描哪些controller  
 .paths(PathSelectors.*any*())  
 .build();  
 }  
  
 private ApiInfo apiInfo() {  
 return new ApiInfoBuilder()  
 .title("Mybatis\_Plus使用Swagger2构建RESTful API") // 标题  
 .description("rest api 文档构建利器") // 描述信息  
 .termsOfServiceUrl("http://www.baidu.com") //服务网址  
 .contact(new Contact("PURE", "https://blog.csdn.net/weixin\_47234994","1638595707@qq.com")) // 联系方式  
 .version("2.0") //版本号  
 .build();  
 }  
}

配置完之后可以访问：<http://localhost:8080/swagger-ui.html>



### 3、使用

在controller中使用：

@ApiOperation(value = "删除用户数据", notes = "")

@DeleteMapping("/delete")  
public Response delete(@ApiParam(value = "需要删除的数据") @RequestBody JSONObject res){  
 User user = JSONObject.*toJavaObject*(res, User.class);  
 int i = userService.delete(user);  
 if ( i == 1){  
 return new Response();  
 }  
 return new Response("删除失败!");  
}

在model中使用：

@ApiModelProperty(hidden = true) //隐藏

@TableId("id")  
private String id;  
  
@ApiModelProperty(value = "姓名")  
@TableField("name")  
private String name;

***swagger注解的使用说明：***

@Api：用在类上，说明该类的作用

@ApiOperation：用在方法上，说明方法的作用  
@ApiIgnore：使用该注解忽略这个API  
@ApiImplicitParams：用在方法上包含一组参数说明  
@ApiImplicitParam：用在@ApiImplicitParams注解中，指定一个请求参数的各个方面  
 paramType：参数放在哪个地方  
 header-->请求参数的获取：@RequestHeader  
 query-->请求参数的获取：@RequestParam  
 path（用于restful接口）-->请求参数的获取：@PathVariable  
 body（不常用）  
 form（不常用）  
 name：参数名  
 dataType：参数类型  
 required：参数是否必须传  
 value：参数的意思  
 defaultValue：参数的默认值  
@ApiResponses：用于表示一组响应  
@ApiResponse：用在@ApiResponses中，一般用于表达一个错误的响应信息  
 code：数字，例如400  
 message：信息，例如"请求参数没填好"  
 response：抛出异常的类  
@ApiModel：描述一个Model的信息（这种一般用在post创建的时候，使用@RequestBody这样的场景，请求参数无法使用@ApiImplicitParam注解进行描述的时候）  
@ApiModelProperty：描述一个model的属性

## 学习消息队列RabbitMQ

RabbitMQ是一个消息代理。它的核心思想非常简单：接收并转发消息。你可以把它想象成一个邮局：当你把邮件丢进邮箱时，你非常确定邮递员先生会把它送到收件人手中。在这个比喻中，RabbitMQ就是邮箱、邮局和邮递员。

RabbitMQ和邮局的主要区别是它处理的不是纸张。它接收、存储并转发二进制数据块，也就是message，消息。

*RabbitMQ官网:* [*https://www.rabbitmq.com/*](https://www.rabbitmq.com/)

*RabbitMQ官方教程:* [*https://www.rabbitmq.com/#getstarted*](https://www.rabbitmq.com/#getstarted)

*学习教程：https://www.rabbitmq.com/tutorials/tutorial-one-python.html*

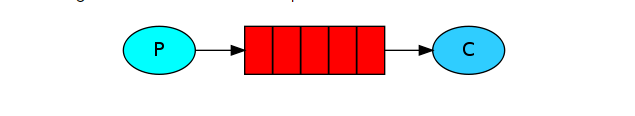
RabbitMQ常用的交换器类型有direct、topic、fanout、headers四种。我分别叫他们：“直接连接交换机”，“主题路由匹配交换机”，“无路由交换机”，“头信息交换”。、

导入RabbitMQ的依赖包：

<!--消息队列RabbitMQ的依赖包-->

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-amqp</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.amqp</groupId>  
 <artifactId>spring-rabbit-test</artifactId>  
 <scope>test</scope>  
</dependency>

### 1、“Hello World”



P（producer/ publisher）：生产者，发送消息的服务

C（consumer）：消费者，接收消息的服务

红色区域就是MQ中的Queue，可以把它理解成一个邮箱

* 首先信件来了不强求必须马上马去拿
* 其次,它是有最大容量的(受主机和磁盘的限制,是一个缓存区)
* 允许多个消费者监听同一个队列，争抢消息

添加发布者代码：

@Slf4j

public class RabbitMQProducer {  
 private final static String *QUEEN\_NAME* = "Hello World";  
 private final static String *MESSAGE* = "第一个RabbitMQ";  
  
 public static void main(String[] args) {  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setHost("localhost");//设置主机名  
 try {  
 Connection connection = factory.newConnection();  
 Channel channel = connection.createChannel();  
 channel.queueDeclare(*QUEEN\_NAME*, false, false, false, null);  
 channel.basicPublish("", *QUEEN\_NAME*, null, *MESSAGE*.getBytes());  
 *log*.info("发送消息：'{}'", *MESSAGE*);  
 channel.close();  
 connection.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 } catch (TimeoutException e) {  
 e.printStackTrace();  
 }  
 }  
  
}

添加接收者代码：

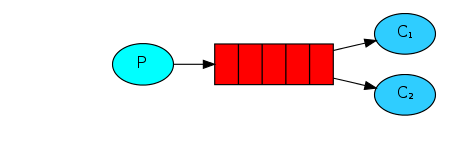
@Slf4j

public class RabbitMQConsumer {  
 private final static String *QUEEN\_NAME* = "Hello World";  
  
 public static void main(String[] args) {  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setHost("localhost");  
 try {  
 Connection connection = factory.newConnection();  
 Channel channel = connection.createChannel();  
 channel.queueDeclare(*QUEEN\_NAME*, false, false, false, null);  
 */\*\*  
 \** ***@Author:*** *chenchao  
 \** ***@Date:*** *2021/8/17 17:28  
 \** ***@Description:*** *第一种方式利用匿名类创建  
 \*/*// Consumer consumer = new DefaultConsumer(channel){  
// @Override  
// public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {  
// String message = new String(body, "UTF-8");  
// log.info("消费者收到消息：'{}'", message);  
// }  
// };  
// channel.basicConsume(QUEEN\_NAME, true, consumer);  
 */\*\*  
 \** ***@Author:*** *chenchao  
 \** ***@Date:*** *2021/8/17 17:28  
 \** ***@Description:*** *第二种方式利用Lambda创建  
 \*/* DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(), "UTF-8");  
 *log*.info("消费者收到消息：'{}'", message);  
 };  
 channel.basicConsume(*QUEEN\_NAME*, true, deliverCallback, consumerTag -> {});  
 } catch (IOException e) {  
 e.printStackTrace();  
 } catch (TimeoutException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

***发送者会通过RabbitMQ发送一条信息，而接收者会把它打印出来。接收者将会一直运行，等待接收消息。***

布置成功之后就可以在浏览器输入：[http://服务器的ip:15672/](http://xn--ip-fr5c86lx7z0gv:15672/) 可以找到登陆入口。

### 2、工作队列（work queue）



Worker模型中也只有一个工作队列。但它是一种竞争消费模式。可以看到同一个队列我们绑定上了多个消费者，消费者争抢着消费消息，**这可以有效的避免消息堆积**。

比如对于短信微服务集群来说就可以使用这种消息模型，来了请求大家抢着消费掉。

如何实现这种架构：对于上面的HelloWorld这其实就是相同的服务我们启动了多次罢了，自然就是这种架构。

发布者代码：

*/\*\**

*\** ***@Author:*** *chenchao  
 \** ***@Date:*** *2021/8/18 10:53  
 \** ***@Description:*** *学习地址：https://www.rabbitmq.com/tutorials/tutorial-two-java.html  
 \*/*public class NewTask {  
 //定义一个消息队列的名称  
 private static final String *QUEEN\_NAME* = "work";  
  
 public static void main(String[] args) {  
 //开启连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名称或IP  
 factory.setHost("localhost");  
 try {  
 //开启连接，抛出异常  
 Connection connection = factory.newConnection();  
 //开启连接通道  
 Channel channel = connection.createChannel();  
 //设置队列参数  
 boolean autoAck = true; //消息确认与持久性，消费者发回确认消息，告诉 RabbitMQ 特定消息已被接收、处理，并且 RabbitMQ 可以自由删除它。  
 channel.queueDeclare(*QUEEN\_NAME*, autoAck, false, false, null);  
 //发布消息  
 String []msg = {"one", "two", "three", "four", "five", "six", "seven", "eight", "night", "ten"};  
 String message = String.*join*("-", msg);//从控制台编译  
 System.*out*.println(message);  
 //我们需要将我们的消息标记为持久性 - 通过将MessageProperties（实现BasicProperties）设置为值PERSISTENT\_TEXT\_PLAIN  
 channel.basicPublish("", *QUEEN\_NAME*, MessageProperties.*PERSISTENT\_TEXT\_PLAIN*, message.getBytes("UTF-8"));  
 //关闭通道  
 channel.close();  
 //关闭连接  
 connection.close();  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 }  
}

消费者1代码：

@Slf4j

public class WorkConsumer {  
 //定义消息队列名称  
 private static final String *QUEEN\_NAME* = "work";  
  
 public static void main(String[] args) {  
 //开启连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名称或IP  
 factory.setHost("localhost");  
 try {  
 //开启连接，抛出异常  
 Connection connection = factory.newConnection();  
 //开启连接通道  
 Channel channel = connection.createChannel();  
 //设置队列参数  
 boolean autoAck = true; //消息确认与持久性，消费者发回确认消息，告诉 RabbitMQ 特定消息已被接收、处理，并且 RabbitMQ 可以自由删除它。  
 channel.queueDeclare(*QUEEN\_NAME*, autoAck, false, false, null);  
 System.*out*.println(" [\*] Waiting for messages. To exit press CTRL+C");  
  
 channel.basicQos(1); //公平调度，一次只接受一条未确认的消息  
  
 DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(), "UTF-8");  
 System.*out*.println("接收到的信息：" + message);  
 try {  
 //do somethings  
 *doWork*(message);  
 } finally {  
 System.*out*.println(" [x] Done");  
 //当您的客户端退出时，消息将被重新传送（这可能看起来像随机重新传送），但 RabbitMQ 将消耗越来越多的内存，因为它无法释放任何未确认的消息。所以利用basicAck()必须在接收交付的同一通道上发送确认  
 channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);//确认消息，设置为false  
 }  
 };  
 channel.basicConsume(*QUEEN\_NAME*, false, deliverCallback, consumerTag -> {});  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 }  
  
 //模拟执行时间的假任务  
 private static void doWork(String message) {  
 int count = 0;  
 for (char ch  
 : message.toCharArray()){  
 //遇到'-'符号耗时1s  
 if (ch == '-'){  
 try {  
 *log*.info("等待时间：{}秒" , ++count);  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e){  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

消费者2代码：

@Slf4j

public class WorkConsumer2 {  
 //定义消息队列名称  
 private static final String *QUEEN\_NAME* = "work";  
  
 public static void main(String[] args) {  
 //开启连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名称或IP  
 factory.setHost("localhost");  
 try {  
 //开启连接，抛出异常  
 Connection connection = factory.newConnection();  
 //开启连接通道  
 Channel channel = connection.createChannel();  
 //设置队列参数  
 boolean autoAck = true; //消息确认与持久性，消费者发回确认消息，告诉 RabbitMQ 特定消息已被接收、处理，并且 RabbitMQ 可以自由删除它。  
 channel.queueDeclare(*QUEEN\_NAME*, autoAck, false, false, null);  
 System.*out*.println(" [\*] Waiting for messages. To exit press CTRL+C");  
  
 channel.basicQos(1); //公平调度，一次只接受一条未确认的消息  
  
 DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(), "UTF-8");  
 System.*out*.println("接收到的信息：" + message);  
 try {  
 //do somethings  
 *doWork*(message);  
 } finally {  
 System.*out*.println(" [x] Done");  
 //当您的客户端退出时，消息将被重新传送（这可能看起来像随机重新传送），但 RabbitMQ 将消耗越来越多的内存，因为它无法释放任何未确认的消息。所以利用basicAck()必须在接收交付的同一通道上发送确认  
 channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);//确认消息，设置为false  
 }  
 };  
 channel.basicConsume(*QUEEN\_NAME*, false, deliverCallback, consumerTag -> {});  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 }  
  
 //模拟执行时间的假任务  
 private static void doWork(String message) {  
 int count = 0;  
 for (char ch  
 : message.toCharArray()){  
 //遇到'-'符号耗时1s  
 if (ch == '-'){  
 try {  
 *log*.info("工人做这项工作时间：{}秒" , ++count);  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e){  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

总结：在工作队列当中，假如将消费者比作工人，队列表示工人需要被分派的任务，那么当工人没有完成一项工作的时候，是不会重新获取其他任务，同时如果因为某个原因导致任务中断，那么RabbitMQ就不会将这项任务在消息队列中去除，而是会被分派给空闲的工人，直到这项任务被完成，也可称为消息应答或消息确认。

### 3、订阅模式：

订阅模型借助一个新的概念：Exchange（交换机）实现，不同的订阅模型本质上是根据交换机(Exchange)的类型划分的。

订阅模型有三种

Fanout（广播模型）: 将消息发送给绑定给交换机的所有队列(因为他们使用的是同一个RoutingKey)。

Direct（定向）: 把消息发送给拥有指定Routing Key (路由键)的队列。

Topic（通配符）: 把消息传递给拥有 符合Routing Patten(路由模式)的队列。

#### 订阅之Fanout模型

这个模型的特点就是它在发送消息的时候,并没有指明Rounting Key , 或者说他指定了Routing Key,但是所有的消费者都知道,大家都能接收到消息,就像听广播。

**临时队列：**

在 Java 客户端中，当我们不向queueDeclare()提供参数时， 我们会创建一个具有生成名称的非持久、独占、自动删除队列：

String queueName = channel.queueDeclare().getQueue();

此时queueName包含一个随机队列名称。例如，它可能看起来像amq.gen-JzTY20BRgKO-HjmUJj0wLg。

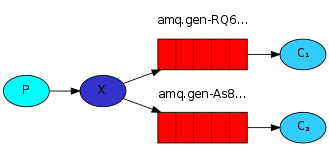
**绑定：**



我们已经创建了一个fanout交换和一个队列。现在我们需要告诉交换器向我们的队列发送消息。交换和队列之间的这种关系称为*绑定*。

channel.queueBind(queueName, "logs" , "" );

从现在开始，logs交换会将消息附加到我们的队列中。



发出日志消息的生产者程序与之前的教程看起来没有太大区别。最重要的变化是我们现在想要将消息发布到我们的logs交换而不是无名的交换。我们需要在发送时提供一个routingKey，但它的值在fanout交换时被忽略。

生产者代码：

*/\*\**

*\** ***@Author:*** *chenchao  
 \** ***@Date:*** *2021/8/18 15:02  
 \** ***@Description:*** *RabbitMQ订阅模型：Fanout  
 \* 学习地址：https://www.rabbitmq.com/tutorials/tutorial-three-java.html  
 \*/*public class EmitLog {  
 //创建交换机名称  
 private static final String *EXCHANGE\_NAME* = "logs";  
  
 public static void main(String[] args) {  
 //创建连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名或者IP  
 factory.setHost("localhost");  
 try {  
 //创建连接  
 Connection connection = factory.newConnection();  
 //开启连接通道  
 Channel channel = connection.createChannel();  
 //定义交换机类型为Fanout  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "fanout");  
 //设置发布的消息  
 String []msg = {"one", "two", "three", "four", "five", "six", "seven", "eight", "night", "ten"};  
 for (int i = 0; i < 10; i++) {  
 channel.basicPublish(*EXCHANGE\_NAME*, "", null, msg[i].getBytes("UTF-8"));  
 }  
 System.*out*.println(msg);  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

消费者1代码：

public class LogsConsumer {

//创建交换机名称  
 private static final String *EXCHANGE\_NAME* = "logs";  
  
 public static void main(String[] args) {  
 //创建连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名或IP  
 factory.setHost("localhost");  
 try {  
 //创建连接  
 Connection connection = factory.newConnection();  
 //开启通道  
 Channel channel = connection.createChannel();  
 //定义交换机名称  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "fanout");  
 //创建随机队列  
 String queueName = channel.queueDeclare().getQueue();  
 //绑定随机队列与交换机  
 channel.queueBind(queueName, *EXCHANGE\_NAME*, "");  
  
 System.*out*.println(" [\*] Waiting for messages. To exit press CTRL+C");  
  
 DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(),"UTF-8");  
 System.*out*.println(" [x] Received '" + message + "'");  
 };  
 channel.basicConsume(queueName, true, deliverCallback, consumerTag -> {});  
  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 }  
}

消费者2代码：

public class LogsConsumer2 {

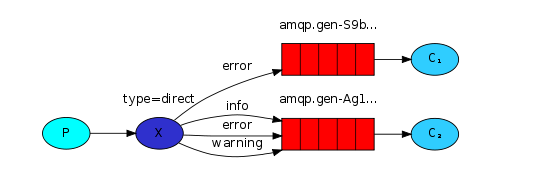
//创建交换机名称  
 private static final String *EXCHANGE\_NAME* = "logs";  
  
 public static void main(String[] args) {  
 //创建连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名或IP  
 factory.setHost("localhost");  
 try {  
 //创建连接  
 Connection connection = factory.newConnection();  
 //开启通道  
 Channel channel = connection.createChannel();  
 //定义交换机名称  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "fanout");  
 //创建随机队列  
 String queueName = channel.queueDeclare().getQueue();  
 //绑定随机队列与交换机  
 channel.queueBind(queueName, *EXCHANGE\_NAME*, "");  
  
 System.*out*.println(" [\*] Waiting for messages. To exit press CTRL+C");  
  
 DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(),"UTF-8");  
 System.*out*.println(" [x] Received '" + message + "'");  
 };  
 channel.basicConsume(queueName, true, deliverCallback, consumerTag -> {});  
  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 }  
}

总结：这种发布/订阅模式是一种广播机制，并没有设置路由key模式。它在发布者代码中创建一个交换机，并没有没有在发布者代码中创建队列，而是在消费者中创建临时队列，通过临时队列与交换机相连，从而达到广播效果。

#### 订阅之Direct模型

我们使用的是fanout交换，它没有给我们很大的灵活性——它只能进行无意识的广播。

我们将改用direct交换。direct交换背后的路由算法很简单 - 消息进入其绑定密钥与消息的路由密钥完全匹配的队列 。



P：生产者，向Exchange发送消息，发送消息时，会指定一个routing key。

X：Exchange（交换机），接收生产者的消息，然后把消息递交给 与routing key完全匹配的队列

C1：消费者，其所在队列指定了需要routing key 为 error 的消息

C2：消费者，其所在队列指定了需要routing key 为 info、error、warning 的消息

拥有不同的RoutingKey的消费者，会收到来自交换机的不同信息，而不是大家都使用同一个Routing Key 和广播模型区分开来。

发布者代码：

*/\*\**

*\** ***@Author:*** *chenchao  
 \** ***@Date:*** *2021/8/18 16:14  
 \** ***@Description:*** *订阅模式：Direct  
 \*/*public class EmitLogDirect {  
 //创建交换机名称  
 private static final String *EXCHANGE\_NAME* = "direct\_logs";  
  
 public static void main(String[] args) {  
 //创建连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名或IP  
 factory.setHost("localhost");  
 try {  
 //创建连接  
 Connection connection = factory.newConnection();  
 //开启连接通道  
 Channel channel = connection.createChannel();  
 //定义交换机类型与名称  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "direct");  
 //设置绑定key与信息  
 String []bindKey = {"info","warn", "error", "test"};  
 String []message = {"Hello","Warning","Error","ok"};  
 //发布消息  
 for (int i = 0; i < bindKey.length; i++) {  
 channel.basicPublish(*EXCHANGE\_NAME*, bindKey[i], null, message[i].getBytes("UTF-8"));  
 System.*out*.println(" [x] Sent '" + bindKey[i] + "':'" + message[i] + "'");  
 }  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

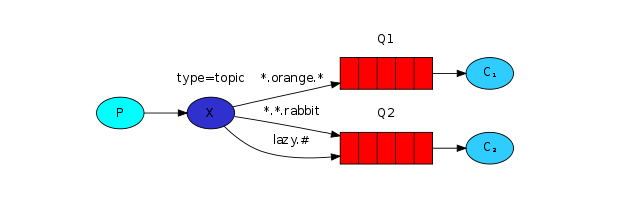
消费者代码：

public class LogsDirectConsumer {

//创建交换机名称  
 private static final String *EXCHANGE\_NAME* = "direct\_logs";  
  
 //创建线程订阅'info','warn','error','test'  
 public static Runnable *runnable* = () -> {  
 //创建连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名或IP  
 factory.setHost("localhost");  
 try {  
 //创建连接  
 Connection connection = factory.newConnection();  
 //开启连接通道  
 Channel channel = connection.createChannel();  
 //定义交换机类型与名称  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "direct");  
 //创建临时队列  
 String queueName = channel.queueDeclare().getQueue();  
 //绑定队列、交换机和路由密钥  
 String []bindKey = {"info","warn", "error", "test"};  
 for (String severity :  
 bindKey) {  
 channel.queueBind(queueName, *EXCHANGE\_NAME*, severity);  
 }  
 System.*out*.println(" [\*] Waiting for messages. To exit press CTRL+C");  
  
 DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(), "UTF-8");  
 System.*out*.println(" [x] Received '" +  
 delivery.getEnvelope().getRoutingKey() + "':'" + message + "'");  
 };  
 channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 };  
  
 //创建线程只订阅'info','warn','error'  
 public static Runnable *runnable1* = () -> {  
 //创建连接工厂  
 ConnectionFactory factory = new ConnectionFactory();  
 //设置主机名或IP  
 factory.setHost("localhost");  
 try {  
 //创建连接  
 Connection connection = factory.newConnection();  
 //开启连接通道  
 Channel channel = connection.createChannel();  
 //定义交换机类型与名称  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "direct");  
 //创建临时队列  
 String queueName = channel.queueDeclare().getQueue();  
 //绑定队列、交换机和路由密钥  
 String []bindKey = {"info","warn", "error"};  
 for (String severity :  
 bindKey) {  
 channel.queueBind(queueName, *EXCHANGE\_NAME*, severity);  
 }  
 System.*out*.println(" [\*] Waiting for messages. To exit press CTRL+C");  
  
 DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(), "UTF-8");  
 System.*out*.println(" [x] Received queue2 '" +  
 delivery.getEnvelope().getRoutingKey() + "':'" + message + "'");  
 };  
 channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 };  
  
 public static void main(String[] args) {  
 new Thread(*runnable*, "queue1").start();//订阅'info','warn','error','test'  
 new Thread(*runnable1*, "queue2").start();//只订阅'info','warn','error'  
 new Thread(*runnable*, "queue3").start();//订阅'info','warn','error','test'  
 }  
}

总结：在direct订阅模型中，生产者订阅交换机发布消息，产生routingKey（用于消费者绑定）；而在消费者中，根据消费者的需要，可以随意订阅自己所需的信息（通过绑定routingKey实现），即选择性接收。

#### 订阅之Topic模型



类似于Direct模型。区别是Topic的Routing Key支持通配符。

topic交换功能强大，可以像其他交换一样运行。

当队列与“ # ”（散列）绑定键绑定时——它将接收所有消息，而不管路由键——就像在fanout交换中一样。

当绑定中不使用特殊字符“ \* ”（星号）和“ # ”（散列）时，主题交换的行为就像direct交换一样。

***通配符如下：***

* ***\*（星号）可以正好代替一个词。***
* ***# (hash) 可以代替零个或多个单词。***

根据上图所示，以下的代码是根据上图进行编写的。

我们发送描述动物的信息。这是由三个字符以及两个点组合的路由密钥进行发送。消费者那边用\*.orange.\*、\*.\*.rabbit以及lazy.#进行适配；生产者那边发布动物的消息以及设置路由key：quick.orange.rabbit、lazy.orange.elephant、quick.orange.fox、lazy.brown.fox、lazy.pink.rabbit、quick.brown.fox、quick.orange.male.rabbit、lazy.orange.male.rabbit

发布者代码：

*/\*\**

*\** ***@Author:*** *chenchao  
 \** ***@Date:*** *2021/8/19 9:42  
 \** ***@Description:*** *RabbitMQ订阅模型： Topics  
 \* 学习网址： https://www.rabbitmq.com/tutorials/tutorial-five-java.html  
 \*/*public class EmitLogTopic {  
 private static final String *EXCHANGE\_NAME* = "topic\_logs";  
  
 public static void main(String[] args) throws IOException, TimeoutException {  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setHost("localhost");  
 Connection connection = factory.newConnection();  
 Channel channel = connection.createChannel();  
  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "topic");  
  
 String[] routingKey = {"quick.orange.rabbit", "lazy.orange.elephant", "quick.orange.fox", "lazy.brown.fox",  
 "lazy.pink.rabbit", "quick.brown.fox", "quick.orange.male.rabbit", "lazy.orange.male.rabbit"};  
  
 String[] message = {"快.橙子.兔子", "慢.橙色.大象", "快.橙色.狐狸", "慢.棕色.狐狸",  
 "慢.粉色.兔子", "快.棕色.狐狸", "快.橙色.雄性.兔子", "慢.橙色.雄性.兔子"};  
  
 for (int i = 0; i < routingKey.length; i++) {  
 channel.basicPublish(*EXCHANGE\_NAME*, routingKey[i], null, message[i].getBytes("UTF-8"));  
 System.*out*.println(" [x] Sent '" + routingKey[i] + "':'" + message[i] + "'");  
 }  
 channel.close();  
 connection.close();  
 }  
  
}

消费者代码：

public class LogsTopicsConsumer {

private static final String *EXCHANGE\_NAME* = "topic\_logs";  
  
 public static void consumer(String []routingKey){  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setHost("localhost");  
 Connection connection = null;  
 try {  
 connection = factory.newConnection();  
 Channel channel = connection.createChannel();  
  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "topic");  
  
 String queueName = channel.queueDeclare().getQueue();  
  
 for (String key :  
 routingKey) {  
 channel.queueBind(queueName, *EXCHANGE\_NAME*, key);  
 }  
  
 System.*out*.println(" [\*] Waiting for messages. To exit press CTRL+C");  
  
 DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
 String message = new String(delivery.getBody(), "UTF-8");  
 System.*out*.println(" [x] Received '" +  
 delivery.getEnvelope().getRoutingKey() + "':'" + message + "'");  
 };  
 channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });  
 } catch (IOException | TimeoutException e) {  
 e.printStackTrace();  
 }  
 }  
  
  
 public static void main(String[] args) {  
 String[] routingKey1 = {"\*.orange.\*"};  
 String[] routingKey2 = {"\*.\*.rabbit"};  
 String[] routingKey3 = {"lazy.#"};  
 *consumer*(routingKey1);  
 *consumer*(routingKey2);  
 *consumer*(routingKey3);  
 }  
  
}

结果分析：

路由键设置为“ quick.orange.rabbit ”的消息将发送到两个队列。

消息“ lazy.orange.elephant ”也会发给他们两个。

“ quick.orange.fox ”只会进入第一个队列，而“ lazy.brown.fox ”只会进入第二个队列。“ lazy.pink.rabbit ”只会被传送到第二个队列一次，即使它匹配了两个绑定。“ quick.brown.fox ”不匹配任何绑定，因此将被丢弃。

如果我们违反合同并发送一到四个字的消息，例如 “ quick.orange.male.rabbit ”，这些消息不会匹配任何绑定并且会丢失。

另一方面，“ lazy.orange.male.rabbit ”，即使它有四个单词，也会匹配最后一个绑定，并将被传递到第二个队列。

## SpringBoot整合RabbitMQ:

### 1、导入RabbitMQ依赖包：

<!--消息队列RabbitMQ的依赖包-->

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-amqp</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.amqp</groupId>  
 <artifactId>spring-rabbit-test</artifactId>  
 <scope>test</scope>  
</dependency>

### 2、配置application.yml

#RabbitMQ配置

#配置服务器地址  
spring.rabbitmq.host=127.0.0.1  
#配置RabbitMQ用户名  
spring.rabbitmq.username=guest  
#配置RabbitMQ密码  
spring.rabbitmq.password=guest  
#如果是默认可以不配置端口号  
#spring.rabbitmq.port=5672

### 3、生产者：

@Component

public class Producer {  
 @Resource  
 private RabbitTemplate rabbitTemplate;  
  
 //设置交换机名称  
 private static final String *EXCHANGE\_FANOUT\_NAME* = "user.fanout.exchange";  
 private static final String *EXCHANGE\_DIRECT\_NAME* = "user.direct.exchange";  
 private static final String *EXCHANGE\_Topic\_NAME* = "user.topic.exchange";  
  
 public void sendFanoutMessage() {  
 String id = UUID.*randomUUID*().toString();  
 String message = "收到信息：" + id + "\t" + "success" + new Date();  
 */\*\*  
 \** ***@Author:*** *chenchao  
 \** ***@Date:*** *2021/8/20 9:02  
 \** ***@Description:*** *\* convertSendAndReceive(*…*)：可以同步消费者。使用此方法，当确认了所有的消费者都接收成功之后，才触发另一个convertSendAndReceive(…)，也就是才会接收下一条消息。RPC调用方式。  
 \* convertAndSend(…)：使用此方法，交换机会马上把所有的信息都交给所有的消费者，消费者再自行处理，不会因为消费者处理慢而阻塞线程。  
 \*/* rabbitTemplate.convertAndSend(*EXCHANGE\_FANOUT\_NAME*, "", message);  
 }  
  
 public void directSendMessage() {  
 */\*\*  
 \* convertAndSend - 转换并发送消息的template方法。  
 \* 是将传入的普通java对象，转换为rabbitmq中需要的message类型对象，并发送消息到rabbitmq中。  
 \* 参数一：交换器名称。 类型是String  
 \* 参数二：路由键。 类型是String  
 \* 参数三：消息，是要发送的消息内容对象。类型是Object  
 \* 订阅模式：direct --> 消费者订阅哪一种routingKey，就输出哪一种信息，在实际开发中，可以在生产者这里进行判断再传输message  
 \*/* //设置绑定key与信息  
 String []routingKeys = {"info","warn", "error", "test"};  
 String []message = {"Hello","Warning","Error","ok"};  
 for (int i = 0; i < routingKeys.length; i++) {  
 System.*out*.println("路由键： " + routingKeys[i] + "\t 发布消息：" + message[i]);  
 rabbitTemplate.convertAndSend(*EXCHANGE\_DIRECT\_NAME*, routingKeys[i], message[i]);  
 }  
  
 }  
  
 public void topicSendMessage() {  
 String[] routingKeys = {"quick.orange.rabbit", "lazy.orange.elephant", "quick.orange.fox", "lazy.brown.fox",  
 "lazy.pink.rabbit", "quick.brown.fox", "quick.orange.male.rabbit", "lazy.orange.male.rabbit"};  
  
 String[] message = {"快.橙子.兔子", "慢.橙色.大象", "快.橙色.狐狸", "慢.棕色.狐狸",  
 "慢.粉色.兔子", "快.棕色.狐狸", "快.橙色.雄性.兔子", "慢.橙色.雄性.兔子"};  
  
 for (int i = 0; i < routingKeys.length; i++) {  
 rabbitTemplate.convertAndSend(*EXCHANGE\_Topic\_NAME*, routingKeys[i], message[i]);  
 System.*out*.println(" [x] Sent '" + routingKeys[i] + "':'" + message[i] + "'");  
 }  
 }  
  
}

### 4、消费者：

*/\*\**

*\* 消息接收者 - consumer  
 \*  
 \** ***@RabbitListener*** *- 可以注解类和方法。  
 \* 注解类，当表当前类的对象是一个rabbit listener。  
 \* 监听逻辑明确，可以由更好的方法定义规范。  
 \* 必须配合@RabbitHandler才能实现rabbit消息消费能力，一个类可以有多个方法，但是仅有一个方法注解@RabbitHandler。  
 \* 注解方法，代表当前方法是一个rabbit listener处理逻辑。  
 \* 方便开发，一个类中可以定义若干个listener逻辑。  
 \* 方法定义规范可能不合理。如：一个方法的处理逻辑太多，造成方法的bad smell。  
 \*  
 \** ***@RabbitListener*** *- 代表当前类型是一个rabbitmq的监听器。  
 \* bindings:绑定队列  
 \** ***@QueueBinding*** *- @RabbitListener.bindings属性的类型。绑定一个队列。  
 \* value:绑定队列， Queue类型。  
 \* exchange:配置交换器， Exchange类型。  
 \* key:路由键，字符串类型。  
 \*  
 \** ***@Queue*** *- 队列。*

*\* value:队列名称  
 \* autoDelete:是否是一个临时队列。  
 \* true ：当所有的consumer关闭后，自动删除queue。  
 \* false：当任意一个consumer启动并创建queue后，如果queue中有消息未消费，无论是否有consumer继续执行，都保存queue。  
 \*  
 \** ***@Exchange*** *- 交换器  
 \* value:为交换器起个名称  
 \* type:指定具体的交换器类型  
 \*/*@Component  
public class Consumer {  
 @RabbitHandler  
 @RabbitListener(bindings = @QueueBinding(value = @Queue(value = "fanout", autoDelete = "true"), exchange = @Exchange(value = "user.fanout.exchange", type = ExchangeTypes.*FANOUT*)))  
 public void processMessage(String msg) {  
 System.*out*.format("Receiving Fanout Message: -----[%s]----- \n", msg);  
 }  
  
 @RabbitHandler  
 @RabbitListener(bindings = @QueueBinding(value = @Queue(value = "direct", autoDelete = "true"), exchange = @Exchange(value = "user.direct.exchange", type = ExchangeTypes.*DIRECT*), key = {"info","error"}))  
 public void directMessage(String msg) {  
 System.*out*.format("Receiving Direct Message: -----[%s]----- \n", msg);//只订阅路由键为："info"，"error"  
 }  
  
 @RabbitHandler  
 @RabbitListener(bindings = @QueueBinding(value = @Queue(value = "direct1", autoDelete = "true"), exchange = @Exchange(value = "user.direct.exchange", type = ExchangeTypes.*DIRECT*), key = {"info","test","warn"}))  
 public void directMessage1(String msg) {  
 System.*out*.format("Receiving Direct1 Message: -----[%s]----- \n", msg);//只订阅路由键为："test"，"warn"  
 }  
  
 @RabbitHandler  
 @RabbitListener(bindings = @QueueBinding(value = @Queue(value = "topic1", autoDelete = "true"), exchange = @Exchange(value = "user.topic.exchange", type = ExchangeTypes.*TOPIC*), key = {"\*.orange.\*"}))  
 public void topicMessage1(String msg) {  
 System.*out*.format("匹配Key： \*.orange.\* \t Receiving Topic1 Message: -----[%s]----- \n", msg);//只订阅路由键为："test"，"warn"  
 }  
  
 @RabbitHandler  
 @RabbitListener(bindings = @QueueBinding(value = @Queue(value = "topic2", autoDelete = "true"), exchange = @Exchange(value = "user.topic.exchange", type = ExchangeTypes.*TOPIC*), key = {"\*.\*.rabbit"}))  
 public void topicMessage2(String msg) {  
 System.*out*.format("匹配Key： \*.\*.rabbit \t Receiving Topic2 Message: -----[%s]----- \n", msg);//只订阅路由键为："test"，"warn"  
 }  
  
 @RabbitHandler  
 @RabbitListener(bindings = @QueueBinding(value = @Queue(value = "topic3", autoDelete = "true"), exchange = @Exchange(value = "user.topic.exchange", type = ExchangeTypes.*TOPIC*), key = {"lazy.#"}))  
 public void topicMessage3(String msg) {  
 System.*out*.format("匹配Key： lazy.# \t Receiving Topic3 Message: -----[%s]----- \n", msg);//只订阅路由键为："test"，"warn"  
 }  
  
}

### 5、service层调用生产者代码：

@Service

public class UserServiceImpl implements UserService {  
 @Resource  
 private UserDao userDao;  
 @Resource  
 Producer producer;//RabbitMQ设置生产者，在接口调用  
  
 @Override  
 public List<User> select() {  
 producer.sendFanoutMessage();//订阅模式：fanout  
 producer.directSendMessage();//订阅模式：direct  
 producer.topicSendMessage();  
 return userDao.selectList(null);  
 }

}

# 第四周：

## 给项目代码添加注释

# 第五周

## LeetCode官网上学习算法，给自己充电

# 第六周

## 看牛客网的资料：https://www.nowcoder.com/tutorial/94/1f2614a674cf4854b01166df67fd7366

**悲观锁：**顾名思义悲观的认为在数据处理过程中极有可能存在修改数据的并发事务（包括本系统的其他事务或来自外部系统的事务），于是将处理的数据设置为锁定状态。悲观锁必须依赖数据库本身的锁机制才能真正保证数据访问的排他性

**乐观锁：**顾名思义，对并发事务持乐观态度（认为对数据的并发操作不会经常性的发生），通过更加宽松的锁机制来解决由于悲观锁排他性的数据访问对系统性能造成的严重影响。最常见的乐观锁是通过数据版本标识来实现的，读取数据时获得数据的版本号，更新数据时将此版本号加1，然后和数据库表对应记录的当前版本号进行比较，如果提交的数据版本号大于数据库中此记录的当前版本号则更新数据，否则认为是过期数据无法更新。Hibernate中通过Session的get()和load()方法从数据库中加载对象时可以通过参数指定使用悲观锁；而乐观锁可以通过给实体类加整型的版本字段再通过XML或@Version注解进行配置。

使用乐观锁会增加了一个版本字段，很明显这需要额外的空间来存储这个版本字段，浪费了空间，但是乐观锁会让系统具有更好的并发性，这是对时间的节省。因此乐观锁也是典型的空间换时间的策略。

## 请简述一下Hibernate常见优化策略。

### 参考回答：

①制定合理的缓存策略（二级缓存、查询缓存）。  
② 采用合理的Session管理机制。  
③ 尽量使用延迟加载特性。  
④ 设定合理的批处理参数。  
⑤ 如果可以，选用UUID作为主键生成器。  
⑥ 如果可以，选用基于版本号的乐观锁替代悲观锁。  
⑦ 在开发过程中, 开启hibernate.show\_sql选项查看生成的SQL，从而了解底层的状况；开发完成后关闭此选项。  
⑧ 考虑数据库本身的优化，合理的索引、恰当的数据分区策略等都会对持久层的性能带来可观的提升，但这些需要专业的DBA（数据库管理员）提供支持。

## [mybatis的缓存机制：一级缓存和二级缓存的区别](https://www.cnblogs.com/yunianzeng/p/11826449.html)

（镶嵌点：）Sqlsession（接口）的底层是hashmap存储，线程不安全，sqlsessionTemplate是其实现类线程安全的

区别：一级缓存的作用域是一个sqlsession内；二级缓存作用域是针对mapper进行缓存.

一级缓存：

1、第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。

2、如果中间sqlSession去执行commit操作（执行插入、更新、删除），则会清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。

3、第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。

小结：一级缓存时执行commit，close，增删改等操作，就会清空当前的一级缓存；当对SqlSession执行更新操作（update、delete、insert）后并执行commit时，不仅清空其自身的一级缓存（执行更新操作的效果），也清空二级缓存（执行commit()的效果）。

二级缓存：

不管是不是相同的session,只要mapper的namespace相同,可能共享缓存，要求：如果开启了二级缓存，那么在关闭sqlsession后(close)，才会把该sqlsession一级缓存中的数据添加到namespace的二级缓存中。

开启了二级缓存后，还需要将要缓存的pojo实现Serializable接口，为了将缓存数据取出执行反序列化操作，因为二级缓存数据存储介质多种多样，不一定只存在内存中，有可能存在硬盘中

那么在同一个session下,执行同一个select语句时,Cache Hit Ratio [Mapper]: 0.0,二级缓存的命中率为0那?

答：这里要讲解一下二级缓存的缓存什么时候存入了:只有当当前的session.close()时,该session的数据才会存入二级缓存.在同一session下时,肯定没有执行.close()关闭session,自然也就没有存入二级缓存.第二次执行却没有重新发送sql语句,是因为第二次调用的是一次缓存中的数据.

如果想让二级缓存命中率不为0,需要先开启一个session,执行一个sql语句,然后关闭该session,然后在创建一个新的session,执行相同的sql语句,这时,二级缓存才会命中

# 正则表达式整理：

|  |  |
| --- | --- |
| 元字符 | 描述 |
| \ | 将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，“\\n”匹配\n。“\n”匹配换行符。序列“\\”匹配“\”而“\(”则匹配“(”。即相当于多种编程语言中都有的“转义字符”的概念。 |
| ^ | 匹配输入字符串的开始位置。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。 |
| \* | 匹配前面的子表达式任意次。例如，zo\*能匹配“z”，也能匹配“zo”以及“zoo”。\*等价于o{0,} |
| + | 匹配前面的子表达式一次或多次(大于等于1次）。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“do”或“does”中的“do”。?等价于{0,1}。 |
| {n} | n是一个非负整数。匹配确定的n次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。 |
| {n,} | n是一个非负整数。至少匹配n次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o\*”。 |
| {n,m} | m和n均为非负整数，其中n<=m。最少匹配n次且最多匹配m次。例如，“o{1,3}”将匹配“fooooood”中的前三个o为一组，后三个o为一组。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。 |
| ? | 当该字符紧跟在任何一个其他限制符（\*,+,?，{n}，{n,}，{n,m}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+”将尽可能多的匹配“o”，得到结果[“oooo”]，而“o+?”将尽可能少的匹配“o”，得到结果 ['o', 'o', 'o', 'o'] |
| .点 | 匹配除“\r\n”之外的任何单个字符。要匹配包括“\r\n”在内的任何字符，请使用像“[\s\S]”的模式。 |
| (pattern) | 匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JScript中则使用$0…$9属性。要匹配圆括号字符，请使用“\(”或“\)”。 |
| (?:pattern) | 非获取匹配，匹配pattern但不获取匹配结果，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分时很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。 |
| (?=pattern) | 非获取匹配，正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。 |
| (?!pattern) | 非获取匹配，正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。 |
| (?<=pattern) | 非获取匹配，反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。 |
| (?<!pattern) | 非获取匹配，反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。这个地方不正确，有问题  此处用或任意一项都不能超过2位，如“(?<!95|98|NT|20)Windows正确，“(?<!95|980|NT|20)Windows 报错，若是单独使用则无限制，如(?<!2000)Windows 正确匹配 |
| x|y | 匹配x或y。例如，“z|food”能匹配“z”或“food”(此处请谨慎)。“[zf]ood”则匹配“zood”或“food”。 |
| [xyz] | 字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。 |
| [^xyz] | 负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。  注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身. |
| [^a-z] | 负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置（即正则表达式的“匹配”有两种概念，一种是匹配字符，一种是匹配位置，这里的\b就是匹配位置的）。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。 |
| \B | 匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。 |
| \cx | 匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的“c”字符。 |
| \d | 匹配一个数字字符。等价于[0-9]。grep 要加上-P，perl正则支持 |
| \D | 匹配一个非数字字符。等价于[^0-9]。grep要加上-P，perl正则支持 |
| \f | 匹配一个换页符。等价于\x0c和\cL。 |
| \n | 匹配一个换行符。等价于\x0a和\cJ。 |
| \r | 匹配一个回车符。等价于\x0d和\cM。 |
| \s | 匹配任何不可见字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。 |
| \S | 匹配任何可见字符。等价于[^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于\x09和\cI。 |
| \v | 匹配一个垂直制表符。等价于\x0b和\cK。 |
| \w | 匹配包括下划线的任何单词字符。类似但不等价于“[A-Za-z0-9\_]”，这里的"单词"字符使用Unicode字符集。 |
| \W | 匹配任何非单词字符。等价于“[^A-Za-z0-9\_]”。 |