# 手术麻醉临床信息系统 BS架构

1. 流程状态名称：手术申请、手术排班、手术准备、入手术室、出手术室、入介入室、出介入室、入复苏室、出复苏室、入诱导室、出诱导室、手术开始、手术结束、麻醉开始、麻醉结束、镇痛开始、镇痛结束、手术准备、病案归档、手术取消。
2. 病人详细信息中的“病情状态”，改为用“ASA分级”表示。
3. 前端：住院登记工作站、门诊登记工作站、护士长工作站、主治工作站、麻醉医生工作站、麻醉护士工作站、复苏室工作站、信息科工作站、质控展示中心、调度监控中心、一体自助终端、家属公示系统、远程会诊中心
4. 后端：系统配置管理、用户和角色管理系统、手术预约系统、手术排班系统、消息通知系统、预警决策系统、事务查询系统、报表生成系统、药品管理系统、手术跟踪系统、设备管理系统、外部设备和系统交互系统、财务资源系统、科研统计系统、专家咨询系统、视频示教系统
5. 设计思路：后端专注业务数据模型，前端呈现灵活定制，系统基于maven构建，采用分布式架构，微服务框架采用SpringCloud技术，支持主流数据库
6. 涉及技术栈：SpringBoot、SpringCloud、SpingBoot Data Jpa、Redis、WebSocket、ELK、queryDSL、Netty、database
7. 手术申请排班系统：将工作时间划分时间段并确定可预约手术数量，获取日程状态，判断是否重复申请无审批，提交至待排班手术列表，有审批，分发至审核人员，通过后放入待排班，不通过结束申请。将最近一周期待排班列表排班提交至排班系统生成已排班列表并将排班消息发到的消息通知系统。排班成功后可修改日程状态。无空闲人员安排调班，调班后重新进入排班系统排班。可撤销申请，撤销排班，申请，排班状态可查看，可能还有意外情况。
8. 手术排班系统：手术申请后进行排班，提示成功与否
9. 基础信息管理：医生数据，患者数据，护士数据，科室数据，字典数据，病床数据，手术室数据，复苏室数据，介入室数据，诱导室数据，病区数据
10. 手术申请 手术排班 手术准备 是否需要诱导室，是入诱导室，否判断是否需要介入手术，是入介入室，否入手术室 在相应室中麻醉开始 判断是否需要诱导室，是出诱导室判断是否介入手术，是入介入室，否入手术室，在相应的室内手术开始 手术结束 判断是否介入手术 出相应的室 判断是否需要复苏室 是入复苏室 麻醉结束 镇痛开始 镇痛结束 出复苏室 判断是否再手术 是回到手术准备 否病案归档
11. 开始 病人登记 是否需要手术 是手术申请 判断能否做手术 否递交申请 申请审批再回判断是否能做手术 能做手术 确认手术 手术同意书 结束
12. 登录 判断时候含有未结束的患者，是选择未结束患者，展开编辑输入，否患者列表中选择患者判断是否代办患者，否展开编辑输入，判断是否修改基本信息，是信息仅查询，查看基本信息
13. 住院登记工作站、门诊登记工作站：登录验证、登出，个人信息显示设置，修改个性化配置，患者管理，患者手术和检查信息查看，在线状态设置，工作站切换
14. 主治工作站：待审核，待诊断，排班干预，诊断急诊，科研计算，质控统计

# 手麻项目开发

## Resource

创建了数据库表的地方貌似都没有这个文件夹，在src—main下

### Schema.sql 初始化数据库

CREATE DATABASE IF NOT EXISTS aims DEFAULT CHARACTER SET utf8mb4 DEFAULT COLLATE utf8mb4\_general\_ci;//代表数据库校对规则ci大小写不敏感，cs敏感

show tables;

//aims是连接可视化navicat后的数据库如aed，aims

1. 创建数据库aims 将该库的默认编码格式设置为utf8mb4格式
2. Utf8mb4：most bytes 4 完全兼容utf8用四个字节存储更多字节，手机端常用表情字符emoji和不常用汉字“墅”等需要四字节才能编码出来
3. 校对规则有utf8mb4-unicode-ci 还有utf8mb4-general-ci

可以直接create table 插入表

### Data.sql 初始化超级管理员科室权限等

使用INSERT INTO 向表中插入数据，数据格式按照表字段 最后关键字commit

### Logback-spring.xml 自定义日志配置

### Bootstrap.yml 配置文件

1. Bootstrap.yml是系统级的优先级更高，application.yml是用户及的资源配置项，所以同一目录下先执行前者
2. 使用spring cloud config server时应在文件中指定spring.application.name和spring.cloud.config.server.git.uri
3. 一些加密/解密信息，会用到
4. 一旦被加载，则内容不会被覆盖

# 经验总结：

## Entity

1. 根据数据库表定义创建实体都大写，使用Java类，五个注解@Data @JsonInclude(JsonInclude.include.NON\_NULL) @DynamicUpdate(value = true) @Entity@Table(name = "数据库表名字如dat\_bed")
2. 类名 implements serializable
3. 主键用@id标识
4. 基本用法：@Apimodelproperty(value=”数据库中字段名，可适当增加注释")

@column（length=111）限定长度，可不添加

Private 类型 列名；（long id；string name； integer status ；date createtime ）类型、列名根据数据库文档来。

## Repository

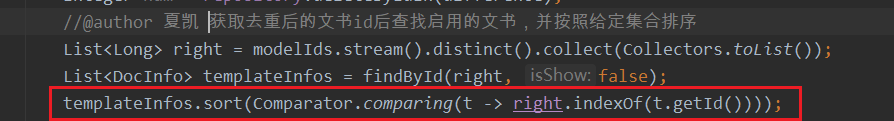
1. 创建仓储层使用interface 命名i+实体，extends JpaRepository <实体名字,Long>, JpaSpecificationExecutor<实体名字> {} 大括号内可以不含东西
2. 大括号内的方法：list<实体> findallbyidinandstatusin(set<long> ids,list<integer>status); Findall后面很多都能够自动生成
3. 用set也可以用list（批量操作）
4. 命名接口可被扩展<extends>,使用@norepositorybean注解定义改接口不会被spring容器注册成bean。
5. JpaSpecificationExecutor，按条件分页排序查询
6. 当用dsl查询不行的时候，可以先将一部分内容查出来，再把剩余的部分用其他方法查出来后赋值，最后拼上就是一个整体查询。还是不行的话就可以在仓储层写原生SQL语句使用 @query ( nativequery =true )注解，只要是对数据库操作都可以写sql语句。

## 测试

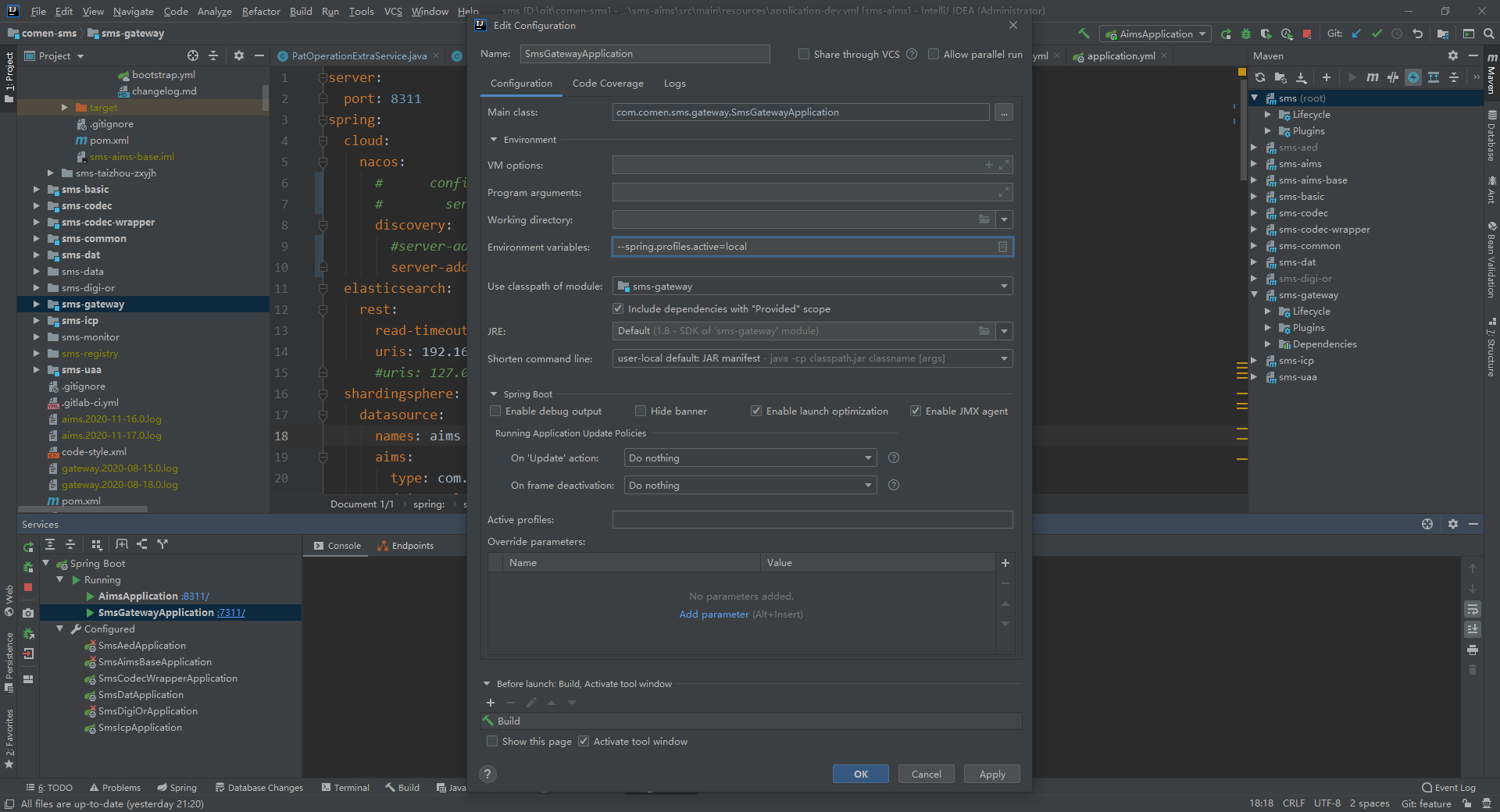
1. 首先网络正确，需要连接到内网
2. 查看源代码中资源的数据库配置，连哪一个数据库是否正确
3. 数据库连接正确，可视化工具连接正确
4. 点击debug不是运行（不走断点）
5. 只经过api不经过service，等会重启，自己变好了

## 技术总结

1. 优先使用string.valueof()方法代替tostring（）
2. 判断一个String的实例s 是否等于“a”时，不要写成s.equals(“a”),这样容易抛异常，写成”a”.equals(s)
3. 两个集合，其中一个集合按照另一个集合排序



1. 网关配置：



# Maven

Pom project object model项目对象模型，maven工程基本工作单元，xml文件，包含项目的基本信息，如何构建，声明依赖等

所有 POM 文件都需要 project 元素和三个必需字段：groupId，artifactId，version

<!-- 模型版本 -->

<modelVersion>4.0.0</modelVersion>

<!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成， 如com.companyname.project-group，maven会将该项目打成的jar包放本地路径：/com/companyname/project-group -->

<groupId>com.companyname.project-group</groupId>

<!-- 项目的唯一ID，一个groupId下面可能多个项目，就是靠artifactId来区分的 -->

<artifactId>project</artifactId>

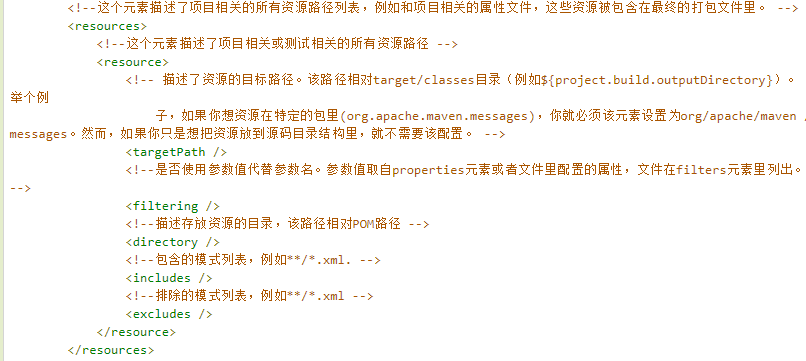
<!-- 版本号 -->

<version>1.0</version>

1. <parent> 被继承的父项目
2. <license> 该元素描述了项目所有License列表。 应该只列出该项目的license列表，不要列出依赖项目的 license列表。如果列出多个license，用户可以选择它们中的一个而不是接受所有license，license标签下：l<distribution>repo</distribution> !--项目分发的主要方式： repo，可以从Maven库下载 manual， 用户必须手动下载和安装依赖
3. <properties> 用户自定义一个或多个maven属性，然后再POM的其他地方使用${属性名} 的方式，消除重复统一管理



1. Source也是在build下：



1. 
2. <dependencies> 该元素描述了项目相关的所有依赖。 这些依赖组成了项目构建过程中的一个个环节。它们自动从项目定义的仓库中下载
3. <scope> 依赖范围。在项目发布过程中，帮助决定哪些构件被包括进来。- compile ：默认范围，用于编译 - provided：类似于编译，但支持你期待jdk或者容器提供，类似于classpath 。- runtime: 在执行时需要使用 - test: 用于test任务时使用 - system: 需要外在提供相应的元素。通过systemPath来取得。 - systemPath: 仅用于范围为system。提供相应的路径 - optional: 当项目自身被依赖时，标注依赖是否传递。用于连续依赖时使用
4. <exclusions> 当计算传递依赖时， 从依赖构件列表里，列出被排除的依赖构件集。即告诉maven你只依赖指定的项目，不依赖项目的依赖。此元素主要用于解决版本冲突问题

查看文件：<https://www.runoob.com/maven/maven-pom.html>

## Pom.xml文件配置

<?xml?>

<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>com.comen</groupId>

<artifactId>sms</artifactId>

<version>1.0.0</version>

//先定义父项目，内容有三项，此例是sms即整个大项目，此例是uaa子模块，sms的父项目是：

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

//<artifactId>spring-boot-starter-parent</artifactId>

//<version>2.2.6.RELEASE</version>

//<relativePath/> <!-- lookup parent from repository -->

<groupId>com.comen</groupId>

<artifactId>sms-uaa</artifactId>

<version>1.0.0</version>

<name>sms-uaa</name>

<description>User authentication authorization</description>

//再定义自己的内容三项，其中groupid 不论父子项目全部相同，一个项目中name和artifactid相同

//在父项目中 version标签下多一句话<packaging>pom</packaging>

<properties>

<java.version>1.8</java.version>

<spring-cloud.version>Hoxton.SR3</spring-cloud.version>

<com.alibaba.cloud.nacos.version>2.2.1.RELEASE

//配置项 此例是父项目，一般就一个Java version，有的项目有分表分库<sharding-sphere.version>

<modules>

<module>sms-common</module>

//父项目专有 内容是所有子项目的名称

<dependencyManagement>

<dependencies> <dependency><groupId> <artifactId> <version> <type> <scope>

<distributionManagement>

<repository><id> <url> < uniqueVersion>

<snapshotRepository> <id> <url>

<repositories>

<repository> <id> <url> <releases>true <snapshots>false

<pluginRepositories>

<pluginRepository> <id> <url> <releases> true <snapshots> false

//父项目到此结束

<dependencies>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.9.2</version>

//添加依赖 子项目可以相互依赖，具体需要的时候网上搜再加

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<configuration>

<skipTests>true</skipTests>

//添加插件，第一个插件子项目都有

<plugin>

<groupId>com.mysema.maven</groupId>

<artifactId>apt-maven-plugin</artifactId>

<version>1.1.3</version>

<executions>

<execution>

<goals>

<goal>process</goal>

<configuration> <outputDirectory>target/generated-sources/java

<processor>com.querydsl.apt.jpa.JPAAnnotationProcessor

<options>

<querydsl.packageSuffix>.dsl</querydsl.packageSuffix>

</options>

//生成Q对象

//子项目到此结束

## 手动安装JAR到本地maven仓库

mvn install:install-file -Dfile=<path-to-file>

因为[Maven-install-plugin](http://maven.apache.org/plugins/maven-install-plugin/) 2.5版本以后这些信息可以从指定的pom文件中提取。

当然也可以指定这些信息：

mvn install:install-file -Dfile=<path-to-file> -DgroupId=<group-id> -DartifactId=<artifact-id> -Dversion=<version>

如：

mvn install:install-file –Dfile=C:\dev\app.jar -DgroupId=com.roufid.tutorials -DartifactId=example-app -Dversion=1.0

在pom.[xml](https://so.csdn.net/so/search?q=xml&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/w605283073/article/details/_blank)中添加如下依赖，就可以在maven项目中使用了

<dependency>

<groupId>com.roufid.tutorials</groupId>

<artifactId>example-app</artifactId>

<version>1.0</version>

</dependency>

## lifecycle基本用法

### 1.clean

用于清除之前构建生成的所有文件

其中具体为清楚了Target目录中的所有文件，包括该目录

i.e：删除了install生成的所有文件

### 2.validate

用于验证项目是否真确，并且其说有必要信息是否都可用

### 3.compile

编译项目的源代码，主要是java文件

一般是编译scr/main/java或是scr/test/java里面的文件

### 4.test

用合适的测试框架来进行测试，测试compile中编译出来的代码

测试的东西一般不加包和部署

### 5.packaging

获取compile中编译好的代码并将其打包成可分类的格式，i.e:JAR

### 6.vertify

这步是用来验证test

检查test的结果是否满足标准

### 7.install

将软件包安装到本地存储库中

确保本地其他项目可能需要使用他（eg:装了core才能用oms）

### 8.site

站点，生成项目的站点文档

### 9.deploy

复制最终的包至远程仓库，共享给其它开发人员和项目

1.mvn compile，编译类文件

2.mvn install，包含mvn compile，mvn package，然后上传到本地仓库

3.mvn deploy,包含mvn install,然后，上传到私服

PS:在install的时候可能会出现乱码，此时对着install点右键，选择create xxx install

在command line里写 install -Dmaven.test.skip=true -f pom.xml 然后用新写的命令代替旧的install即可

### Plugins：插件

上面的生命周期规定了整体的框架，而具体的工作则是由插件提供。

### Dependencies：依赖

依赖，maven项目的pom.xml中配置的依赖都会在这里

maven的一般操作：先clean在package或者install

## 常用命令：

显示版本信息 :  
mvn -version/-v

生成target目录，编译、测试代码，生成测试报告，生成jar/war文件 :  
mvn package

显示详细错误 信息:  
mvn -e

运行任何检查，验证包是否有效且达到质量标准  
mvn verify

想要查看完整的依赖踪迹，包含那些因为冲突或者其它原因而被拒绝引入的构件，打开 Maven 的调试标记运行  
mvn install -X

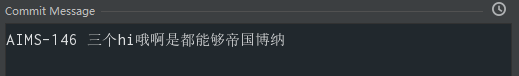
打印整个依赖树mvn dependency:tree

# **Git**

## **提交代码**：

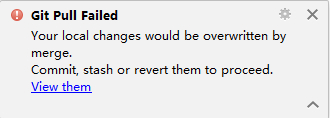
注意提交代码是在git dev分支

1. 新增代码都是红色，点击VSC—update project/或者蓝色向左下的箭头—merge
2. 成功合并以后，点击文件（如datroomservice）—git—add，把需要的文件全部add
3. 点击整个项目名称—git—commit directory—选择需要上传的文件—在commit message中加入项目问题的关键字（关键字创建流程在下面）如下，和描述



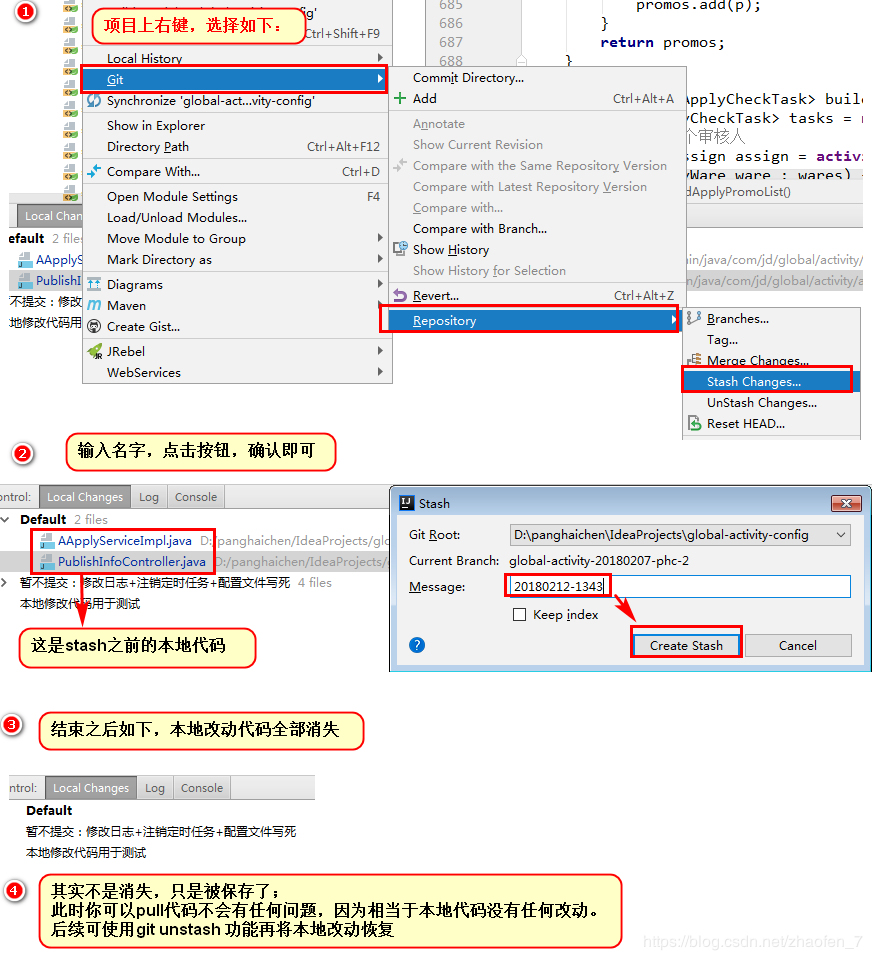
1. 再点击commit旁边的commit and push
2. 点击项目名称—git—repository 检查是否提交到git而不是本地仓库/或者在gitlab—activity中查看是否有更新活动。

## 更新代码失败

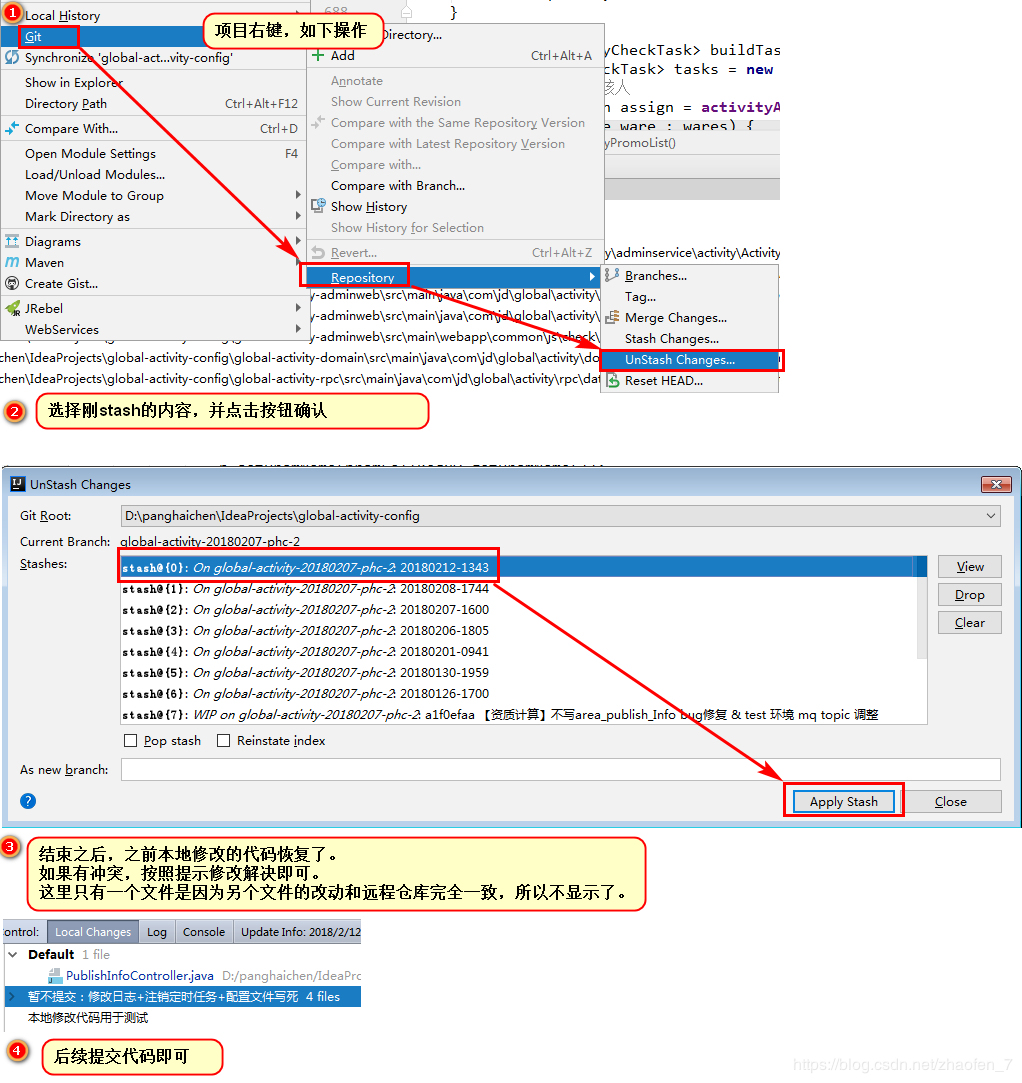


提示有三个选择，提交（更新都失败了，提交不了），revert（操作清零）

Stash：在项目或者当前页面右键git—repository—stash changes



将正在修改的代码保存，message中填入信息有利于找到刚刚stash，--create以后会看到local changes没有东西，所有蓝色标识变成白色。当你正在改代码，突然要修复一个bug，总不能撤销所有操作，这时用stash，保留现在的工作状态，stash以后，本地你的所有刚刚写到一半的代码就被保存了，还可以恢复的那种。当修改bug结束后，提交远程，再通过unstash就又回到修改bug之前的工作状态。



# BUG

## Java.lang.NullPointeerException

原因：

1. 字符串变量未初始化
2. 接口类型的对象没有用具体的类初始化

//没有初始化Map map(会报错) Map map = new Map()（不会报错）

1. 当一个对象的值为空，你没有判断为空的情况
2. 空值与字符串、Enum的元素比较

//String str =null; if(str.equals(“tset”); （代码不会被触发抛上述异常）

1. 优先使用String.valueof()方法代替tostring()

//当程序代码需要对象的字符串表示形式时，请避免使用该对象的TOstring方法，如果你的对象的引用等于 null，NullPointerExcrption则会抛出，使用静态方法string.valueof（）方法，该方法不会抛出任何异常并打印“null”

1. Class被声明了类型，默认class = null，这样调用class中方法的时候系只能给你个空指针异常，实例化就好了 class= new Class()；
2. 返回null，方法的返回值不要定义成为一般的类型，而是用数组，这样如果想要返回null的时候就能避免许多不必要的nullpointerexception

解决：

1. 首先debug报空指针错误后，把项目运行run，再去postman里面请求这个接口，此时再运行项目的控制台就会打印输出日志，日志里面蓝色的就是报错的位置，点击可跳转

## Java parse error

1. Cannot deserialize instance of `java.lang.Integer` out of START\_OBJECT token 使用postman传参格式错误，看postman的json格式是否正确，方法中传参是否正确，看是object，list，json，从而判断body中写{}还是写[]。

## 服务无法启动，控制台打印

1. Error creating bean with name 'IDatRoomRepository': Invocation of init method failed;repository.IDatRoomRepository.findAllByCode(java.util.Set)! Operator SIMPLE\_PROPERTY on code requires a scalar argument

原因是在错误里面的仓储层定义了重复功效的方法或者两个入参相同名字一点点不一样的方法

## Page分页查询

1. Last最后一页true/false 错误在new pageimpl里面的参数应该是（数据，pageable，数据.gettotle）不是getsize或者其他

## 服务启动很久

端口被占用，在任务管理器把Java platform se binary都结束任务再重启

## Entity

新增字段时 定义类型和名称，启动报错

no viable alternative at input 'integer'/ mismatched input 'integer' expecting

这个问题是integer版本不兼容 不要用这个integer类型

第二个是 定义的名称不要是数据库的关键字

# Jara software

关键字创建：再科曼项目库—看板—新建（填写概要，优先级，描述等信息）—新建—把经办人改为自己。

# 生成Q实体类



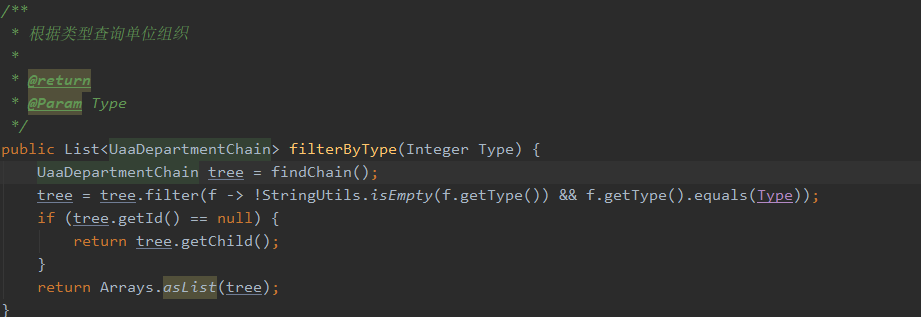
q对象自动生成不需要手动添加 在实体对象页面 打开右侧maven找到对应页面，点击compile就能够自动生成 他是通过实体类映射到q对象的

Q对象的生成是因为在pom.xml文件写入了方法，在pom.xml里面查找这个方法是否存在，在手麻中这个东西被注释掉了。再在右侧maven里面install或者clean，再compile后生成的文件在pom.xml所在文件夹的target文件夹中，复制粘贴。Compile以后还报错就install或者clean

# 写接口api

1. 注意@xxmapping 删除是delete 还有get post 注意别弄错
2. 接口中对应的服务是服务层.服务层的方法实现，不是通过接口命名。

**Uaadepartmentservice:根据类型查询单位组织**



# Java IO流

## SequenceInputStream 序列流

可以将两个或多个其他input stream 合并为一个，按照顺序读取。用构造方法sequenceInputSream(inputstream s1,inputStream s2)

组合两种以上的inputStream ，将实例放入vector

Vector<inputstream> stream = new vector<>();

Stream.add(input1); input1是input stream流 类似三个就能添加进去

SequenceInputstream sequenceinputstream = new sequenceinputstream（stream.elements()）; 或者用构造方法两两组合

## ByteArrayInputStream 内存流

缓冲区buffered：高效

## InputstreamReader OutputStreamWriter 转换流

## DataInputStream 数据流

## 操作基本数据类型 ObjectInputStream 对象流

操作对象，对象序列化

## 创建一个再内存操作临时可用的outputstream

Outputstream outputstream = new bytearrayoutputstream();

## 内存操作流处理临时信息

/\*\*

\* @author shusheng

\* @description 内存操作流（处理临时信息）

\* @Email shusheng@yiji.com

\* @date 2018/12/23 2:35

\*/public class ByteArrayStreamDemo {

/\*\*

\* 内存操作流：用于处理临时存储信息的，程序结束，数据就从内存中消失。操作字节数组

\* ByteArrayInputStream ByteArrayOutputStream

\* 操作字符数组

\* CharArrayReader CharArrayWrite

\* 操作字符串

\* StringReader StringWriter

\*/

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

// 写数据 ByteArrayOutputStream()

ByteArrayOutputStream baos = new ByteArrayOutputStream();

// 写数据

for (int x = 0; x < 10; x++) {

baos.write(("hello" + x + "\n").getBytes());

}

// public byte[] toByteArray()

byte[] bys = baos.toByteArray();

// 读数据

// ByteArrayInputStream(byte[] buf)

ByteArrayInputStream bais = new ByteArrayInputStream(bys);

int by = 0;

while ((by = bais.read()) != -1) {

System.out.print((char) by);

}

}

## 复制inputstream

public List<InputStream> copy(InputStream inputStream, int num)

throws Exception{

byte[] bytes = IOUtils.toByteArray(inputStream); //org.apache.commons.io.IOUtils

List<InputStream> list = new ArrayList<>();

while (num-- > 0){

list.(new ByteArrayInputStream(bytes));

}

baos.close();

return list;

}

## 图片转pdf

@SneakyThrows  
 public void imageToPdfUpload(String operationId, Map<String, List<String>> docinfoMap) throws IOException, DocumentException {  
 long s = System.currentTimeMillis();  
 log.debug("bucket{},开始转换图片成pdf{}", operationId, JSON.toJSONString(docinfoMap));  
 docinfoMap.forEach((k, v) -> {  
 try {  
 ByteArrayOutputStream outputStream = new ByteArrayOutputStream();  
 Document pdfDocument = new Document(PageSize.A4, 20, 20, 20, 30);  
 PdfWriter.getInstance(pdfDocument, outputStream);  
 pdfDocument.open();  
 for (int i = 0; i < v.size(); i++) {  
 InputStream inputStream = objectStorage.getObject(operationId, v.get(i));  
 ImageInputStream imageInputStream = ImageIO.createImageInputStream(inputStream);  
 byte[] buffer = new byte[1024];  
 ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream(inputStream.available());  
 int len = imageInputStream.read(buffer);  
 while (len > 0) {  
 byteArrayOutputStream.write(buffer, 0, len);  
 len = imageInputStream.read(buffer);  
 }  
 Image image = Image.getInstance(byteArrayOutputStream.toByteArray());  
 float percent = getPercent(image.getHeight(), image.getWidth());  
 image.setAlignment(Image.MIDDLE);  
 image.scalePercent(percent - 3f);  
 pdfDocument.newPage();  
 pdfDocument.add(image);  
  
 byteArrayOutputStream.close();  
 imageInputStream.close();  
 inputStream.close();  
 }  
 pdfDocument.close();  
 InputStream uploadPdfStream = new ByteArrayInputStream(outputStream.toByteArray());  
 String filename = k + ".pdf";  
 objectStorage.putObject(operationId, filename, uploadPdfStream);  
// IOUtils.copy(uploadPdfStream, response.getOutputStream());  
// byteArrayOutputStream.close();  
 uploadPdfStream.close();  
 } catch (Exception e) {  
 log.error(">>>>>图片转换PDF错误", e);  
 }  
 });  
 log.debug("转换结束，耗时{}ms", System.currentTimeMillis() - s);  
 }  
 /\*\*  
 \* 计算比率  
 \*  
 \* @param height 高  
 \* @param weight 宽  
 \* @return  
 \*/  
 private static float getPercent(float height, float weight) {  
 float percent = 0.0F;  
 if (height > weight) {  
 percent = PageSize.A4.getHeight() / height \* 100;  
 } else {  
 percent = PageSize.A4.getWidth() / weight \* 100;  
 }  
 return percent;  
 }

一些参考性的建议

StreamUtils.copy()  
// InputStream inputStream = minioClient.getObject(GetObjectArgs.builder().bucket(operationId.toString()).object(t).build());  
// PdfStamper pdfStamper =  
// IOUtils.copy(inputStream, response.getOutputStream());  
// int length = inputStream.read(buffer);  
// while (inputStream.read(buffer) != -1) {  
// byteArrayOutputStream.write(buffer, 0, length);  
// length = inputStream.read(buffer);  
// }  
// IOUtils.copy(inputStream, byteArrayOutputStream);

# set、list、map的区别和联系

1. Set集合继承collection的接口，是一个不包含重复元素的集合，set和list都是以接口的形式进行声明。Set主要包含三中国存放数据类型的变量：hashset基于ha哈希表（无序，唯一）依赖两个方法保证元素唯一性hashcode，equals，linkedhashset基于链表和哈希表（FIFO插入有序，唯一）由链表保证有序，哈希表保证唯一，treeset基于红黑树
2. List和set是存储单列数据的集合，map是存储键值对这样的双列数据的集合。
3. List中存储的数据是有顺序的，并且允许重复；map中存储的数据是无序的，他的键不允许重复，但是值允许重复；set中存储的数据是是无序，并且不允许重复，但元素在集合中的位置由元素的hashcode决定，即位置是固定的
4. List接口由三个实现类**linkedlist**：基于链表实现，链表内存散列，增删快，查找慢。**Arraylist**基于数据实现，非线程安全，效率高 增删慢 查找快。**Vector**：基于数组实现，线程安全，效率低，增删查找慢
5. Map ：hashtable、linkedhashmap、hashmap、treemap 。Set的实现类能对集合中的对象按照特定的方式排序
6. Map的实现类有HashMap,LinkedHashMap,TreeMap。 2. HashMap是有无序的,LinkedHashMap和TreeMap都是有序的（LinkedHashMap记录了添加数据的顺序；TreeMap默认是自然升序）。 LinkedHashMap底层存储结构是哈希表+链表，链表记录了添加数据的顺序TreeMap底层存储结构是二叉树，二叉树的中序遍历保证了数据的有序性LinkedHashMap有序性能比较高，因为底层数据存储结构采用的哈希表。

Map接口的实现类 Map的区别

java为数据结构中的映射定义了一个接口java.util.Map;它有四个实现类,分别是HashMap Hashtable LinkedHashMap 和TreeMap.

HashMap最多只允许一条记录的键为Null;

允许多条记录的值为 Null;

HashMap不支持线程的同步，即任一时刻可以有多个线程同时写HashMap;可能会导致数据的不一致。

Hashtable继承自Dictionary类，

不同的是:它不允许记录的键或者值为空;它支持线程的同步，即任一时刻只有一个线程能写Hashtable,因此也导致了 Hashtable在写入时会比较慢。

LinkedHashMap 是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的.也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比HashMap慢，不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比 LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。

TreeMap实现SortMap接口，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。

## [Collections.singletonList方法](https://www.cnblogs.com/crazy-lc/p/13724101.html)

这个方法主要用于只有一个元素的优化，减少内存分配，无需分配额外的内存，可以从SingletonList内部类看得出来,由于只有一个element,因此可以做到内存分配最小化，相比之下ArrayList的DEFAULT\_CAPACITY=10个

## contains和containsAll

contains

判断集合中是否包含指定元素

containsAll

判断该集合是否包含另一集合的所有元素

注意：

当要表示集合c1包含c2的所有元素时，不能直接用c1.contains(c2),此时表示的是c1的元素中是否包含c2这个集合元素,应该用c1.containsAll(c2)

## MAP接口

1. 给定一个键和一个值，你可以将该值存储在一个map对象，可以通过键来访问对应的值
2. 当访问的值不存在的时候，方法就会抛出一个NoSuchElementException异常
3. 当对象的类型和 Map 里元素类型不兼容的时候，就会抛出一个 ClassCastException 异常。
4. 当在不允许使用 Null 对象的 Map 中使用 Null 对象，会抛出一个 NullPointerException 异常。
5. 当尝试修改一个只读的 Map 时，会抛出一个 UnsupportedOperationException 异常。
6. put(Object key,Object value); //该方法会将原先Key所对应的键值对覆盖掉，即可以用该方法修改Map中的值。

### 功能

a:添加功能

V put(K key,V value):添加元素。这个其实还有另一个功能?替换

如果键是第一次存储，就直接存储元素，返回null

如果键不是第一次存在，就用值把以前的值替换掉，返回以前的值

map.putAll()//形参为map集合

把一个map集合合并到另一个map集合里，map1.putAll(map2)如果key由重复的，则map2的键值会覆盖与map1重复的键值

b:删除功能

void clear():移除所有的键值对元素

V remove(Object key)：根据键删除键值对元素，并把值返回

c:判断功能

boolean containsKey(Object key)：判断集合是否包含指定的键

boolean containsValue(Object value):判断集合是否包含指定的值

boolean isEmpty()：判断集合是否为空

d:获取功能

Set<Map.Entry<K,V>> entrySet(): 返回一个键值对的Set集合

V get(Object key):根据键获取值

Set keySet():获取集合中所有键的集合

Collection values():获取集合中所有值的集合

e:长度功能

int size()：返回集合中的键值对的对数

### map遍历

1. map.foreach((k,v)->{//代码})；

//第一种：普遍使用，二次取值（for循环取值，再用值获取键）

      System.out.println("通过Map.keySet遍历key和value：");

      for (String key : map.keySet()) {  //通过foreach方法来遍历

      System.out.println("key= "+ key + " and value= " + map.get(key));  }

 //第四种  
      System.out.println("通过Map.values()遍历所有的value，但不能遍历key");

      for (String v : map.values()) {

       System.out.println("value= " + v);  }}  
 //第二种（while循环，使用迭代器）  
      System.out.println("通过Map.entrySet使用iterator遍历key和value：");

      Iterator<Map.Entry<String, String>> it = map.entrySet().iterator();  //通过构造操作符来遍历

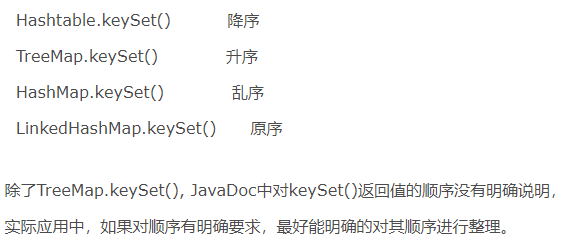
      while (it.hasNext()) {

       Map.Entry<String, String> entry = it.next();

       System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue()); }  
      //第三种：推荐，尤其是容量大时  
      System.out.println("通过Map.entrySet遍历key和value");  //同样是foreach

      for (Map.Entry<String, String> entry : map.entrySet()) {

       System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());}



我也不知道这是写了干嘛的 好像是key去重的

Map<Long, LocalDateTime> node = v.stream().collect(HashMap::new, (x, y) -> x.put(y.getProcessid(), y.getStarttime()), HashMap::putAll);

## Set

Set转string 用英文逗号隔开

String.join(“,”,set); package java.lang;

## List

### 按照给定大小分割list

List<List<Map>> splitList(List<Map> list , int groupSize){undefined  
    return  Lists.partition(list, groupSize); // 使用guava进行分组  
} import com.google.common.collect.Lists

List<List<Integer>> subSets = ListUtils.partition(intList, 3);

使用[apache](https://so.csdn.net/so/search?q=apache&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/PandaWang1989/article/details/_blank) commons collection4

Colletion.stream.collect(collectors.partitioningby()); 这个方法list、set都可以用

### list转数组

String[] array = (String[]) list.toArray(new String[list.size()]);

//ArrayList<String> list3=new ArrayList<String>();  
//String strings[]=(String [])list.toArray();

### 数组转list

1. List<String> list2 = new ArrayList<String>(Arrays.asList(array));
2. List<String> list4=new ArrayList<String>(str.length); Collections.addAll(list4, str);
3. List<String> list5=Stream.of(str).collect(Collectors.toList());

# ftp上传和下载

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.OutputStream;

import org.apache.commons.net.ftp.FTPClient;

import org.apache.commons.net.ftp.FTPFile;

import org.apache.commons.net.ftp.FTPReply;

import org.apache.log4j.Logger;

public class FtpUtil {

private static Logger logger=Logger.getLogger(FtpUtil.class);

private static FTPClient ftp;

/\*\*

\* 获取ftp连接

\* @param f

\* @return

\* @throws Exception

\*/

public static boolean connectFtp(Ftp f) throws Exception{

ftp=new FTPClient();

boolean flag=false;

int reply;

if (f.getPort()==null) {

ftp.connect(f.getIpAddr(),21);

}else{

ftp.connect(f.getIpAddr(),f.getPort());

}

ftp.login(f.getUserName(), f.getPwd());

ftp.setFileType(FTPClient.BINARY\_FILE\_TYPE);

reply = ftp.getReplyCode();

if (!FTPReply.isPositiveCompletion(reply)) {

ftp.disconnect();

return flag;

}

ftp.changeWorkingDirectory(f.getPath());

flag = true;

return flag;

}

/\*\*

\* 关闭ftp连接

\*/

public static void closeFtp(){

if (ftp!=null && ftp.isConnected()) {

try {

ftp.logout();

ftp.disconnect();

} catch (IOException e) {

e.printStackTrace();

}

}

}

/\*\*

\* ftp上传文件

\* @param f

\* @throws Exception

\*/

public static void upload(File f) throws Exception{

if (f.isDirectory()) {

ftp.makeDirectory(f.getName());

ftp.changeWorkingDirectory(f.getName());

String[] files=f.list();

for(String fstr : files){

File file1=new File(f.getPath()+"/"+fstr);

if (file1.isDirectory()) {

upload(file1);

ftp.changeToParentDirectory();

}else{

File file2=new File(f.getPath()+"/"+fstr);

FileInputStream input=new FileInputStream(file2);

ftp.storeFile(file2.getName(),input);

input.close();

}

}

}else{

File file2=new File(f.getPath());

FileInputStream input=new FileInputStream(file2);

ftp.storeFile(file2.getName(),input);

input.close();

}

}

/\*\*

\* 下载链接配置

\* @param f

\* @param localBaseDir 本地目录

\* @param remoteBaseDir 远程目录

\* @throws Exception

\*/

public static void startDown(Ftp f,String localBaseDir,String remoteBaseDir ) throws Exception{

if (FtpUtil.connectFtp(f)) {

try {

FTPFile[] files = null;

boolean changedir = ftp.changeWorkingDirectory(remoteBaseDir);

if (changedir) {

ftp.setControlEncoding("GBK");

files = ftp.listFiles();

for (int i = 0; i < files.length; i++) {

try{

downloadFile(files[i], localBaseDir, remoteBaseDir);

}catch(Exception e){

logger.error(e);

logger.error("<"+files[i].getName()+">下载失败");

}

}

}

} catch (Exception e) {

logger.error(e);

logger.error("下载过程中出现异常");

}

}else{

logger.error("链接失败！");

}

}

/\*\*

\*

\* 下载FTP文件

\* 当你需要下载FTP文件的时候，调用此方法

\* 根据<b>获取的文件名，本地地址，远程地址</b>进行下载

\*

\* @param ftpFile

\* @param relativeLocalPath

\* @param relativeRemotePath

\*/

private static void downloadFile(FTPFile ftpFile, String relativeLocalPath,String relativeRemotePath) {

if (ftpFile.isFile()) {

if (ftpFile.getName().indexOf("?") == -1) {

OutputStream outputStream = null;

try {

File locaFile= new File(relativeLocalPath+ ftpFile.getName());

//判断文件是否存在，存在则返回

if(locaFile.exists()){

return;

}else{

outputStream = new FileOutputStream(relativeLocalPath+ ftpFile.getName());

ftp.retrieveFile(ftpFile.getName(), outputStream);

outputStream.flush();

outputStream.close();

}

} catch (Exception e) {

logger.error(e);

} finally {

try {

if (outputStream != null){

outputStream.close();

}

} catch (IOException e) {

logger.error("输出文件流异常");

}

}

}

} else {

String newlocalRelatePath = relativeLocalPath + ftpFile.getName();

String newRemote = new String(relativeRemotePath+ ftpFile.getName().toString());

File fl = new File(newlocalRelatePath);

if (!fl.exists()) {

fl.mkdirs();

}

try {

newlocalRelatePath = newlocalRelatePath + '/';

newRemote = newRemote + "/";

String currentWorkDir = ftpFile.getName().toString();

boolean changedir = ftp.changeWorkingDirectory(currentWorkDir);

if (changedir) {

FTPFile[] files = null;

files = ftp.listFiles();

for (int i = 0; i < files.length; i++) {

downloadFile(files[i], newlocalRelatePath, newRemote);

}

}

if (changedir){

ftp.changeToParentDirectory();

}

} catch (Exception e) {

logger.error(e);

}

}

}

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

Ftp f=new Ftp();

f.setIpAddr("1111");

f.setUserName("root");

f.setPwd("111111");

FtpUtil.connectFtp(f);

File file = new File("F:/test/com/test/Testng.java");

FtpUtil.upload(file);//把文件上传在ftp上

FtpUtil.startDown(f, "e:/", "/xxtest");//下载ftp文件测试

System.out.println("ok");

}

}

# Criteriabuilder类和predicate类

C是一个工厂类，用来创建安全查询的criteria query对象

CQ查询主语句—必须作用与实体类型（指明主表），root根对象对应于from后面的表，如select \* from a，b，c abc相当于root，root，root（root创建是要指定他的类型）

Predicate 过滤条件，应用到SQL语句的from子句中，判断参数是否符合要求

Specification<Users> spec=new Specification<Users>() {

/\*\*

\* 匿名内部类

\* Predicate:封装了单个的查询条件

\* Root<Users> root:查询对象的属性的封装

\* CriteriaQuery<?> query:封装了我们要执行的查询中的各个部分的信息

\* CriteriaBuilder cb:查询条件的构造器。定义不同的查询条件

\*/

public Predicate toPredicate(Root<Users> root, CriteriaQuery<?> query, CriteriaBuilder cb) {

//这里用一个list，主要是为了将不同的条件分别封装起来，最后在组合成一个，有利于管理，如果只有一个条件，不用这个也可以

List<Predicate> predicates = new ArrayList<>();

if (userName!=null) {

//这里使用CD更像是拼接的意思，and就是在SQL后面增加一个and ，其他的雷同

//查询条件例子：where name ='李健'

//参数一：查询条件的属性 参数二：条件值

predicates.add(cb.and(cb.equal(root.get("userName"),userName )));

}

CriteriaBuilder 提供了 .and(且) .or(或) .like(模糊) .equals(=) isNotNull(非空) .ge(大于) 等方法来构造查询条件，

提供方法名称查询方式

方法名称遵循findBy+属性名（首字母大写）+查询条件（首字母大写Is Equals）

findByNameLike(string name)

先判断是否查询实体的一个属性，如果是进行查询，没有该属性，从右往左截取第一个大写字母 检查剩下的部分是否为查询实体的一个属性，重复第二步。

# 序列化 Serializable:

将内存中的数据变成可保存或可传输的数据的过程叫做序列化

一个对象序列化接口，序列化是将对象状态转换为可保持相对或传输的格式的过程。与序列化相对的时反序列化，他将流转换为对象，两者相加可以存储和传输数据，把对象转换成字节序列的过程，成为对象的序列化。

1. 当我们需要把对象的状态信息通过网络进行传输，或者需要将对象的状态信息持久化，以便将来使用时需要把对象序列化。SerialversionUID用辅助对象序列化和反序列化，原则上序列化后的数据当中serialversionUID 与当前类点各种的serialversionUID一致才能被反序列化成功，对象所属的类必须要实现serializable接口。
2. 对象的反序列化创建对象的时候并不会调用构造方法
3. 使用objectionputstream反序列化的时候，objectinputobject会先读取文件中的servialversionUID 然后与本地class文件的UID
4. 如果一个对象某个数据不想被序列化到硬盘上，可以用关键字transient
5. 如果一个类维护了另一个类的引用，则另一个类也要实现serialzable接口
6. 如果某个类能够被序列化，其子类也可以被序列化。声明为static和transient类型的成员数据不能被序列化。因为static代表类的状态，transient代表对象的临时数据， static对象变量在反序列化时取得的值为当前jvm中对应类中对应static变量的值，而transient（瞬态）关键字则一般用于标识那些在序列化时不需要传递的状态变量。

# Java 8 stream

## Filter过滤多重条件：

filter(t ->(t.getLevel() == null && StringUtils.*isEmpty*(t.getGroups())) ||  
(t.getLevel() != null && !StringUtils.*isEmpty*(t.getGroups())))  
 .collect(Collectors.*toList*());

过滤字符：

filter(t -> !StringUtils.isEmpty(t)).collect(Collectors.toList())

# main函数

public static void main （String[] args）{}

/\*public :被jvm调用，所以权限要足够大

//static：被jvm调用，不要创建对象，直接类名.调用即可

//void： 被jvm调用，不需要有任何返回值

//String[] args: 以前用来接收键盘录入

# copyonwritearraylist

写入时复制，读取不加锁，只有写入与写入之间进行同步。在对一块内存修改时，不直接在原有内存块中进行写操作，而是将内存拷贝一份，在新的内存中进行写操作，写完之后，再将原来指向的内存指针指到新的内存，这样的内存就可以被回收。Copyonwritearraylist写入操作add（）方法在添加集合的时候加了锁，保证同步，避免多线程写的时候会copy出多个副本。

# **JavaBean**:

1. 实体类，用来封装对象，这个类里面全部都是属性值和getset方法，一个可复用的类，在mvc设计模式中是model，又称为模型层，数据层。写成JavaBean类必须是具体的和公共的，并且具有无参构造器、属性私有化、私有化的属性必须通过public类型的方法暴露给其他程序
2. 用户可以使用JavaBean将功能、处理、值、数据库访问和其他任何可以用Java代码创造的对象进行打包，用户可以认为JavaBean提供了一种随时随地的复制和粘贴功能，而不用关心任何改变
3. JavaBean分两种，一种是有用户界面（ui、user interface）的JavaBean，还有一种是没有用户界面，主要处理事务（数据运算、操纵数据库）的JavaBean，JSP通常访问的是后一种
4. Singleton，默认作用域，使用该属性定义bean时，ioc容器可以创建多个bean实例，每次返回都是一个新的实例。
5. 每个属性通常都需要具有相应的setter、getter方法（属性修改器，属性访问器）修改器（访问器）必须以小写set（get）前缀开始，后跟属性名，且属性名的第一个字母要大写。可以只有set或者get称为只写、只读属性。

# 左连接，右连接

Jeft join ：返回左表中的所有记录以及右表中连接字段相等的记录

Right join：返回右表中的所有记录以及左表中连接字段相等的记录

Inner join：内连接 等值连接 只返回两个表中连接字段相等的行

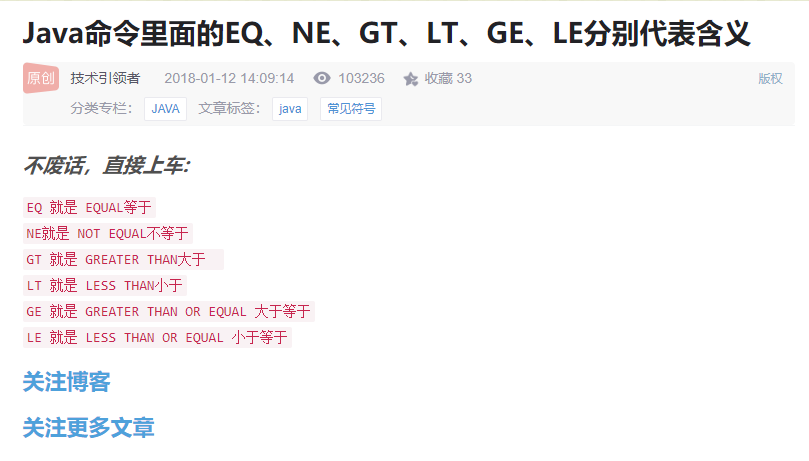
关键字on 数据库再通过连接多张表返回记录时，生成临时表返回给用户

1. On和where的区别

On是在生成临时表时使用的条件，不管on中的条件是否真，都会返回左表记录（关联条件）

Where是在临时表生成后，再对临时表进行过滤的条件，这时已经没有left join的含义了，条件不为真就全部过滤掉。（过滤条件）

1. 关于左连接，不论是在左表的条件还是右表的，写在on和where中不同
2. （左连接）不管on的条件是否和左表有关，都不会影响到左表的内容
3. 不论左表右表，条件写在on或者where里结果是不一样的



# 数据库（mysql）

MariaDB 知识库：<https://mariadb.com/kb/en/built-in-functions/> 可以在上面找到很多mysql用的函数和功能。

## 数据库分组

public List<TTimeAndNumber> qcNeopathyGroupByTime(List<String> name, LocalDateTime start, LocalDateTime end) {  
 //左连接手术表  
 QPatOperation qPatOperation = QPatOperation.patOperation;  
 StringTemplate dataExpr = Expressions.stringTemplate("DATE\_FORMAT({0},'%Y-%m-%d')", qPatOperation.outroomtime);  
 JPAQuery jpaQuery = jpaQueryFactory.select(Projections.bean(TTimeAndNumber.class,  
 dataExpr.as("time"), QPatNeopathy.patNeopathy.bid.countDistinct().as("number"))).from(QPatNeopathy.patNeopathy)  
 .leftJoin(QPatOperation.patOperation).on(QPatOperation.patOperation.id.eq(QPatNeopathy.patNeopathy.bid));  
 jpaQuery.where(QPatNeopathy.patNeopathy.name.in(name), QPatOperation.patOperation.outroomtime.isNotNull(),  
 QPatOperation.patOperation.outroomtime.between(start, end));  
 jpaQuery.groupBy(dataExpr);  
 return jpaQuery.fetch();  
}

## Group\_concat函数

group\_concat([DISTINCT] 要连接的字段 [Order BY 排序字段 ASC/DESC ] [Separator '分隔符'])

SEPARATOR 是一个字符串值，它被用于插入到结果值中。缺省为一个逗号 (",")，可以通过指定 SEPARATOR "" 完全地移除这个分隔符。

"sql": "select concat('[', group\_concat(concat('{\"id\":',id,',\"name\":\"',name,'\"}') order by (FIND\_IN\_SET(id,(select TRIM(BOTH '[' FROM (TRIM(BOTH ']' FROM scrubnurses))) from pat\_operation where id=#OPERATIONID#)))), ']') from uaa\_user where FIND\_IN\_SET(id,(select TRIM(BOTH '[' FROM (TRIM(BOTH ']' FROM scrubnurses))) from pat\_operation where id=#OPERATIONID#))"

## find\_in\_set

in查询相当于多个or条件的叠加，例如：

select \* from user where user\_id in (1,2,3);  
等效于  
select \* from user where user\_id = 1 or user\_id = 2 or user\_id = 3;  
not in与in相反，如下  
select \* from user where user\_id not in (1,2,3);  
等效于  
select \* from user where user\_id != 1 and user\_id != 2 and user\_id != 3;

1.find\_in\_set基本语法

FIND\_IN\_SET(str,strlist)

str 要查询的字符串，strlist 字段名 参数以”,”分隔 如 (1,2,6,8)  
如果str不在strlist 或strlist 为空字符串，则返回值为 0 。如任意一个参数为NULL，则返回值为 NULL。这个函数在第一个参数包含一个逗号(‘,’)时将无法正常运行。

1、如果待查询的条件是常量那就使用IN，是变量则使用FIND\_IN\_SET，可以使用索引的，貌似，哈哈。

2、如果使用IN和FIND\_IN\_SET都能满足条件，则最好使用IN，理由同上，特别是查询字段为主键时或有索引时。

3、如果使用IN不能满足功能需求，那只能使用FIND\_IN\_SET了，哈哈，有时候说不定IN中条件加个%号也可以解决问题，加个%号IN就不只是比较是否相等了！

## 实现中文排序

select \* from table\_name ORDER BY CONVERT(name USING gbk);

select \* from table\_name ORDER BY CONVERT(name USING gb2312) desc

# QueryDSL

<http://querydsl.com/static/querydsl/4.1.3/reference/html_single/>

[QueryDsl官网文档](https://link.zhihu.com/?target=http://www.querydsl.com/static/querydsl/4.1.3/reference/html_single/" \t "https://zhuanlan.zhihu.com/p/_blank)

## querydsl查询使用函数DATE\_FORMAT

日期查询使用函数

String date = "2017-08-01";

*//建立格式化模板*

StringTemplate dateExpr = Expressions.

stringTemplate("DATE\_FORMAT({0},'%Y-%m-%d')", qCommissionCashOrder.createTime);

expression = expression.and(dateExpr.eq(date));

Dsl的groupby 方法：

错误示例：

BooleanExpression alwaysTrueExpression = match(null, null, datePeriod, period);

if (minAmount != null) {

alwaysTrueExpression = alwaysTrueExpression.and(aggregatedOrder.totalUnitAmountWithoutTax.sum().goe(minAmount));

}

if (maxAmount != null) {

alwaysTrueExpression = alwaysTrueExpression.and(aggregatedOrder.totalUnitAmountWithoutTax.sum().loe(maxAmount));

}

JPAQuery<AggregatedOrder> query = selectFrom(aggregatedOrder);

query = query.where(alwaysTrueExpression);

log.debug("sql={}", query);

return query.transform(GroupBy.groupBy(aggregatedOrder.tradingPromoterBranch.promoter).as(GroupBy.sum(aggregatedOrder.totalUnitAmountWithoutTax)));

正确示例：

BooleanExpression alwaysTrueExpression = Expressions.asBoolean(true).isTrue();

NumberExpression<BigDecimal> totalUnitAmountWithoutTax = aggregatedOrder.totalUnitAmountWithoutTax.sum();

NumberExpression<Integer> totalQuantity = aggregatedOrder.quantity.sum();

NumberPath<Long> qPromoter = aggregatedOrder.tradingPromoterBranch.promoter.id;

BooleanExpression amountExpression = Expressions.asBoolean(true).isTrue();

if (minAmount != null) {

amountExpression = amountExpression.and(totalUnitAmountWithoutTax.goe(minAmount));

}

if (maxAmount != null) {

amountExpression = amountExpression.and(totalUnitAmountWithoutTax.loe(maxAmount));

}

JPAQuery<Tuple> query = jpaQueryFactory.select(qPromoter, totalUnitAmountWithoutTax, totalQuantity).from(aggregatedOrder).groupBy(qPromoter);

query = query.where(alwaysTrueExpression).having(amountExpression);

return query.fetch();

# 学习：

list转page：

public <T> Page<T> listConvertToPage(Pageable pageable, List<T> list) {

int start = (int)pageable.getOffset();

int end = (start + pageable.getPageSize()) > list.size() ? list.size() : (start + pageable.getPageSize());

return new PageImpl<T>(list.subList(start, end), pageable, list.size());

}

多表级联分页查询，JPA执行原生SQL：

@Query(value = "SELECT \* FROM xxx WHERE xxx.xxx = ?1",nativeQuery = true,countProjection = "1")

Page<T> findPageByLastName(Object args,Pageable page);

QueryDSL也是可以支持分页的，返回对象QueryResult<T>

1. 所有涵数返回值为Optional时，不能return null，为空返回一个空对象，不然就不要定义成Optional，他是一个对象容器，可以判断对象存在，返回对象，不能用于对象之间的比较。



1. @Query注解中有6个参数，value参数是我们需要填入的JPQL/SQL查询语句；nativeQuery参数是标识该查询是否为原生SQL查询，默认为false；countQuery参数为当你需要使用到分页查询时，可以自己定义（count查询）计数查询的语句，如果该项为空但是如果要用到分页，那么就使用默认的主sql条件来进行计数查询；name参数为命名查询需要使用到的参数，一般配配合@NamedQuery一起使用；countName参数作用与countQuery相似，但是使用的是命名查询的（count查询）计数查询语句；countProjection为涉及到投影部分字段查询时的计数查询（count查询）
2. 原生SQL：数据库能直接执行的SQL语句，关系数据库语言的国际标准。
3. 数据定义语言DDL（data definition language）关键字：数据库database 表table，列colunm。关键字：create，修改alter，删除drop（结构）
4. 数据操作语言DML(data manipulation language)，用来查询数据库中表的记录进行更新。关键字：插入insert，删除delete，更新update（数据）
5. 数据查询语言DQL(data query language) 用来查询数据库中表的记录，关键字：select，from where
6. 数据控制语言DCL(data control language) 用来定义数据库的访问权限和安全级别及创建用户 关键字grant

非关系型数据库noSQL（not only SQL）如mongDB（文档数据库）、redis（键值对存储）Hbase(列存储) infogrid（图形数据库）

1. 对于整形integer类型是元素，用array.aslist转换为集合，判断某个元素是否存在与此集合中用contains：list.contains(e)
2. Keyset: 将Map中所有的键存入到set集合中。因为set具备迭代器。所有可以迭代方式取出所有的键，再根据get方法。获取每一个键对应的值。 keySet():迭代后只能通过get()取key,用list.map(Map::keyset),就能获取

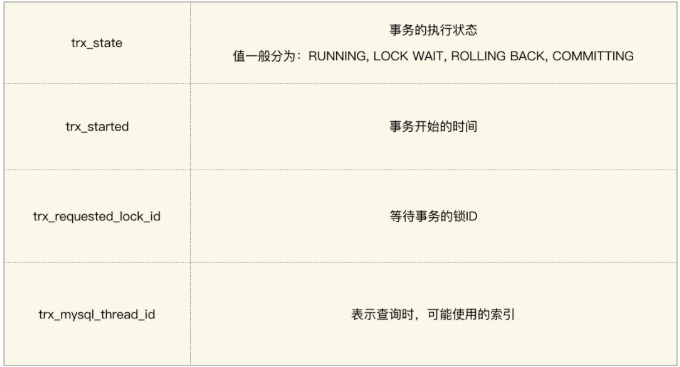
## Java

Java中有三种变量，分别是类变量，成员变量，局部变量，他们分别存放在jvm的方法区，堆内存，栈内存

## SQL

SELECT \* FROM information\_schema.INNODB\_TRX;

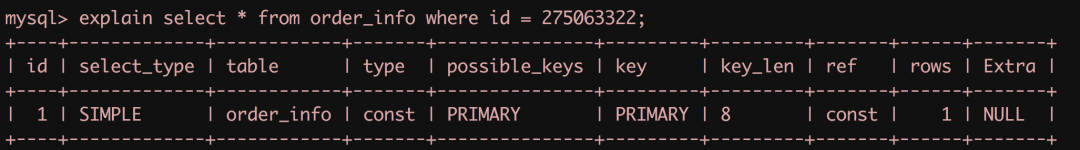
通过在命令行执行语句，可以查看当前运行的事务情况

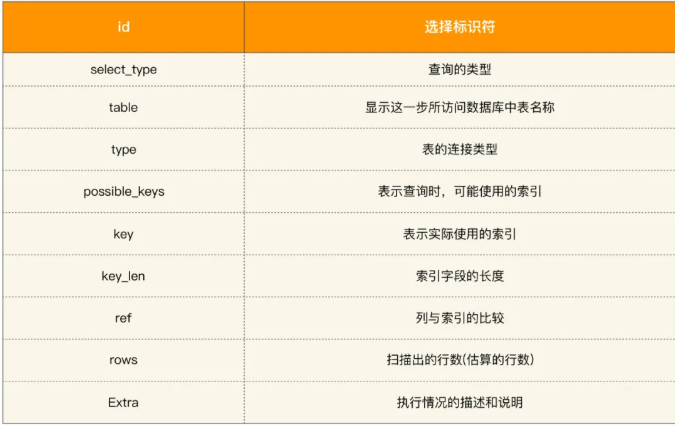


当前事务如果等待时间过长或出现死锁 可以通过kill 线程id释放当前锁

这里的线程id指的是表中trx\_mysql\_thread\_id

还有一些命令：explain，show processlist





1、type

表示 MySQL 在表中找到所需行的方式。其中常用的类型有：ALL、index、range、 ref、eq\_ref、const、system、NULL 这些类型从左到右，性能逐渐变好。

ALL：Mysql 遍历全表来找到匹配的行；

index：与 ALL 区别为 index 类型只遍历索引树；

range：只检索给定范围的行，使用一个索引来选择行；

ref：表示上述表的连接匹配条件，哪些列或常量被用于查找索引列上的值；

eq\_ref：类似ref，区别在于使用的是否为唯一索引。对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用 primary key 或者 unique key作为关联条件；

const、system：当 Mysql 对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于 where 列表中，Mysql 就能将该查询转换为一个常量，system 是 const类型的特例，当查询的表只有一行的情况下，使用system；

NULL：Mysql 在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

2、possible\_keys

查询时可能使用到的索引（但不一定会被使用，没有任何索引时显示为 NULL）。

3、key

实际使用到的索引。

4、rows

估算查找到对应的记录所需要的行数。

5、Extra

比较常见的是下面几种：

Useing index：表明使用了覆盖索引，无需进行回表；

Using where：不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤；

Using temporary：表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询，常见 group by，order by；

Using filesort：当Query中包含 order by 操作，而且无法利用索引完成的排序操作称为“文件排序”。

对于要一张表的多个字段对应同一张表的主键，在不采用连接多次同一张表的时候，可以采用写原生sql 将另一张表的某个字段全部取出来，再get/set进去，具体代码在数字化手术室digior ipatoperationrepository中

SELECT o.id, (SELECT GROUP\_CONCAT(name) FROM digior.uaa\_user where JSON\_CONTAINS(o.anesthetists, JSON\_ARRAY(id))) anesthetistnames,\n" + "(SELECT GROUP\_CONCAT(name) FROM digior.uaa\_user where JSON\_CONTAINS(o.doctors, JSON\_ARRAY(id))) doctornames,\n" + "(SELECT GROUP\_CONCAT(name) FROM digior.uaa\_user where JSON\_CONTAINS(o.nurses, JSON\_ARRAY(id))) nursenames,\n" + "d.name as deptname\n" + "FROM digior.pat\_operation o INNER JOIN digior.uaa\_department d ON o.deptid = d.id WHERE o.id IN ?1

## JSON

解析jsonarray 获取jsonarray里（对象）属性和属性值

JSONArray jsonArray1 = jsonObject.getJSONArray("result");

for (int i = 0; i < jsonArray1.length(); i++) {

JSONObject temp = (JSONObject) jsonArray1.get(i);

String x = temp.getString("x");

String y = temp.getString("y");

if (x == null || y == null) {

flag = false;

handler.sendEmptyMessage(NO\_LOCATION);

}

}

## [LocalDateTime、LocalDate、Long、Date、String 相互转换](https://www.cnblogs.com/puke/p/11314431.html)

DateTimeFormatter dateTimeFormatter1 = DateTimeFormatter.*ofPattern("yyyy-MM-dd HH:mm:ss");  
DateTimeFormatter dateTimeFormatter2 = DateTimeFormatter.ofPattern("yyyy-MM-dd");  
  
LocalDateTime localDateTime = LocalDateTime.parse("2019-07-31 00:00:00",dateTimeFormatter1);  
LocalDate localDate = LocalDate.parse("2019-07-31",dateTimeFormatter2);  
Date date = Date.from(LocalDateTime.parse("2019-07-31 00:00:00",dateTimeFormatter1).atZone(ZoneId.systemDefault()).toInstant());  
  
  
String strDateTime = "2019-07-31 00:00:00";  
String strDate = "2019-07-31";  
Long timestamp=1564502400000l;  
  
/\*\* LocalDateTime 转 LocalDate \*/  
System.out.println("LocalDateTime 转 LocalDate: "+localDateTime.toLocalDate());  
/\*\* LocalDateTime 转 Long \*/  
System.out.println("LocalDateTime 转 Long: "+localDateTime.atZone(ZoneId.systemDefault()).toInstant().toEpochMilli());  
/\*\* LocalDateTime 转 Date \*/  
System.out.println("LocalDateTime 转 Date: "+Date.from(localDateTime.atZone(ZoneId.systemDefault()).toInstant()));  
/\*\* LocalDateTime 转 String \*/  
System.out.println("LocalDateTime 转 String: "+localDateTime.format(dateTimeFormatter1));  
  
System.out.println("-------------------------------");  
  
/\*\* LocalDate 转 LocalDateTime \*/  
System.out.println("LocalDate 转 LocalDateTime: "+LocalDateTime.of(localDate,LocalTime.parse("00:00:00")));  
/\*\* LocalDate 转 Long \*/  
System.out.println("LocalDate 转 Long: "+localDate.atStartOfDay(ZoneId.systemDefault()).toInstant().toEpochMilli());  
/\*\* LocalDate 转 Date \*/  
System.out.println("LocalDate 转 Date: "+Date.from(localDate.atStartOfDay().atZone(ZoneId.systemDefault()).toInstant()));  
/\*\* LocalDate 转 String \*/  
System.out.println("LocalDateTime 转 String: "+localDateTime.format(dateTimeFormatter2));  
  
System.out.println("-------------------------------");  
  
/\*\* Date 转 LocalDateTime \*/  
System.out.println("Date 转 LocalDateTime: "+LocalDateTime.ofInstant(date.toInstant(), ZoneId.systemDefault()));  
/\*\* Date 转 Long \*/  
System.out.println("Date 转 Long: "+date.getTime());  
/\*\* Date 转 LocalDate \*/  
System.out.println("Date 转 LocalDateTime: "+LocalDateTime.ofInstant(date.toInstant(), ZoneId.systemDefault()).toLocalDate());  
/\*\* Date 转 String \*/  
SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS" );  
System.out.println("Date 转 String: "+sdf.format(date));  
  
System.out.println("-------------------------------");  
  
/\*\* String 转 LocalDateTime \*/  
System.out.println("String 转 LocalDateTime: "+LocalDateTime.parse(strDateTime,dateTimeFormatter1));  
/\*\* String 转 LocalDate \*/  
System.out.println("String 转 LocalDate: "+LocalDateTime.parse(strDateTime,dateTimeFormatter1).toLocalDate());  
System.out.println("String 转 LocalDate: "+LocalDate.parse(strDate,dateTimeFormatter2));  
/\*\* String 转 Date \*/  
System.out.println("String 转 Date: "+Date.from(LocalDateTime.parse(strDateTime,dateTimeFormatter1).atZone(ZoneId.systemDefault()).toInstant()));  
  
System.out.println("-------------------------------");  
  
/\*\* Long 转 LocalDateTime \*/  
System.out.println("Long 转 LocalDateTime:"+LocalDateTime.ofInstant(Instant.ofEpochMilli(timestamp), ZoneId.systemDefault()));  
/\*\* Long 转 LocalDate \*/  
System.out.println("Long 转 LocalDate:"+LocalDateTime.ofInstant(Instant.ofEpochMilli(timestamp), ZoneId.systemDefault()).toLocalDate());*

1.将LocalDateTime转为自定义的时间格式的字符串

public static String getDateTimeAsString(LocalDateTime localDateTime, String format) {

　　DateTimeFormatter formatter = DateTimeFormatter.ofPattern(format);

　　return localDateTime.format(formatter);

}

2.将long类型的timestamp转为LocalDateTime

public static LocalDateTime getDateTimeOfTimestamp(long timestamp) {

　　 Instant instant = Instant.ofEpochMilli(timestamp);

　　ZoneId zone = ZoneId.systemDefault();

　　return LocalDateTime.ofInstant(instant, zone);

}

3.将LocalDateTime转为long类型的timestamp

　　public static long getTimestampOfDateTime(LocalDateTime localDateTime) {

　　ZoneId zone = ZoneId.systemDefault();

　　Instant instant = localDateTime.atZone(zone).toInstant();

　　return instant.toEpochMilli();

}

4.将某时间字符串转为自定义时间格式的LocalDateTime

public static LocalDateTime parseStringToDateTime(String time, String format) {

　　DateTimeFormatter df = DateTimeFormatter.ofPattern(format);

　　return LocalDateTime.parse(time, df);

}

## [BigDecimal](https://so.csdn.net/so/search?q=BigDecimal&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/xx123698/article/details/_blank)

这边特别提一下，如果进行除法运算的时候，结果不能整除，有余数，这个时候会报java.lang.ArithmeticException:

，这边我们要避免这个错误产生，在进行除法运算的时候，针对可能出现的小数产生的计算，必须要多传两个参数

divide(BigDecimal，保留小数点后几位小数，舍入模式)

舍入模式

ROUND\_CEILING //向正无穷方向舍入

ROUND\_DOWN //向零方向舍入

ROUND\_FLOOR //向负无穷方向舍入

ROUND\_HALF\_DOWN //向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，向下舍入, 例如1.55 保留一位小数结果为1.5

ROUND\_HALF\_EVEN //向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，如果保留位数是奇数，使用ROUND\_HALF\_UP，如果是偶数，使用ROUND\_HALF\_DOWN

ROUND\_HALF\_UP //向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，向上舍入, 1.55保留一位小数结果为1.6,也就是我们常说的“四舍五入”

ROUND\_UNNECESSARY //计算结果是精确的，不需要舍入模式

ROUND\_UP //向远离0的方向舍入



public static String formatProportion(Integer count,Integer totalCount){

BigDecimal countBigDecimal = new BigDecimal(count);

BigDecimal totalCountBigDecimal = new BigDecimal(totalCount);

BigDecimal divide = countBigDecimal.divide(totalCountBigDecimal, 20, BigDecimal.ROUND\_UP);

for (int i=19;i>=2;i--){

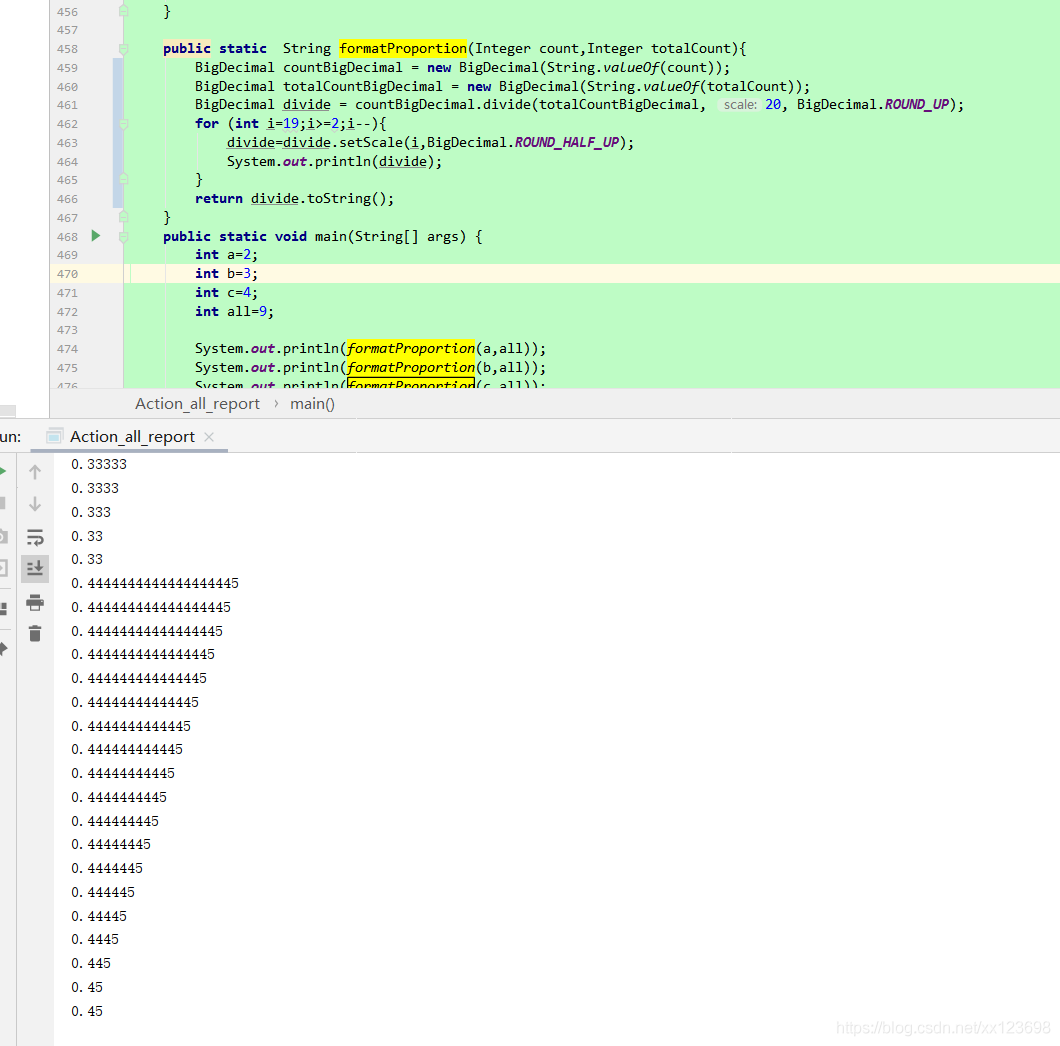
divide=divide.setScale(i,BigDecimal.ROUND\_HALF\_UP);

System.out.println(divide);

}

return divide.toString();

}



# String字符串

## 字符串拼接:

不涉及循环的，就是那种很简单的那种拼接，就用 + ，简单方便 ；

非循环体中的字符串拼接，若只是两个字符串拼接，推荐使用concat。

涉及到循环的，比如说 for 的，可以考虑使用 StringBuilder , 要求线程安全的就选择 StringBuffer ;

有 List 这种的，StringJoiner 不免一个好的选择。

### [+号拼接](https://blog.csdn.net/qq_44705904/article/details/114581004" \l "_2" \t "https://blog.csdn.net/qq_44705904/article/details/_self)

编译的时候把“+”号操作符替换成了 StringBuilder 的 append 方法，效率最差

在for循环的时候不建议使用+号拼接；在编译阶段就能够确定的字符串常量直接使用字符串常量的"+"连接操作效率最高（如：String str = “a” + “b” + “c”;每循环一次都会创建一个StringBuilder对象，且都会调用toString方法转换成字符串，所以开销很大（时间、空间）

### [concat方法](https://blog.csdn.net/qq_44705904/article/details/114581004" \l "concat_22" \t "https://blog.csdn.net/qq_44705904/article/details/_self)

如果拼接的字符串是 null，concat 时候就会抛出 NullPointerException，“+”号操作符会当做是“null”字符串来处理 会创建新的string对象

### [StringBuffer （线程安全）](https://blog.csdn.net/qq_44705904/article/details/114581004" \l "StringBuffer__41" \t "https://blog.csdn.net/qq_44705904/article/details/_self)

### [StringBuilder（线程不安全）](https://blog.csdn.net/qq_44705904/article/details/114581004" \l "StringBuilder_49" \t "https://blog.csdn.net/qq_44705904/article/details/_self)

效率最快

### String类的join方法

Join和buffer相差不大

### 使用java.util.StringJoiner类的add方法

# 深度复制和浅复制

1. 值传递：直接复制一份数据，在调用函数时将实际参数复制一份到函数中，这样在函数中如果对参数进行修改不会影响到实际参数，当数据量大时，运行效率变低，如基本数据类型int，double，Boolean等
2. 引用传递：直接把内存地址传过去，操作的是源数据，如object，二维数组，list，map等

属性一个个赋值就是深度复制，解决方法：

对象映射

二进制流序列化反序列化

Automapper（helper）

Json序列化和反序列化

# JSON

正常的序列化是将编程语言中的对象直接转成可保存或可传输的，这样会保存对象的类型信息，而json序列化不会保留对象类型，只保留对象属性

Json不能存储date对象，需要存储date，需要将其转换成字符串

通过Session管理用户登录状态成本越来越高，慢慢发展成为Token的方式做登录身份校验，然后通过Token去取Redis中的缓存的用户信息，随着之后json web token的出现，无需通过Redis缓存，而是直接根据Token取出保存的用户信息，以及对Token可用性校验

Jwt做app的登录系统，大概流程：

用户输入用户名密码

App请求登录中心验证用户名密码

如果验证通过生成一个token，包含：用户的uid，token过期时间等返回给app

App获得token，保存在cookie中下次请求其他服务带上

其他服务获取到token之后调用登录中心接口验证验证通过则响应

1. Json的6种数据类型

String：字符串，在用双引号中

Number：数值，与JavaScript的number一致，整数15位，小数最大位数17

Object：对象，{key（必须是字符串）：value，key：value}表示方式，可嵌套

Array：数组，[value，value] 表示方式，可嵌套

True/false：布尔类型，JavaScript的Boolean类型

Null：空值

1. Json使用场景

Ajxa异步访问数据

Rpc远程调用

前后端分离后端返回的数据

开放api

企业合作接口

# ORM

对象关系映射，通过使用描述对象和数据库之间映射的元数据，将面向对象语言程序中的对象自动持久化到关系数据库中。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。

ORM框架就是用于实现ORM技术的程序，常见的ORM框架：hibernate，toplink

Java中ORM的原理： 要实现JavaBean的属性到数据库表的字段的映射，任何ORM框架不外乎是读某个配置文件把JavaBean的属 性和数据库表的字段自动关联起来，当从数据库Query时，自动把字段的值塞进JavaBean的对应属性里，当做INSERT或UPDATE时，自动把 JavaBean的属性值绑定到SQL语句中。

# postman接口测试：

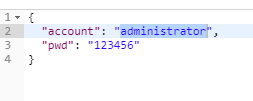
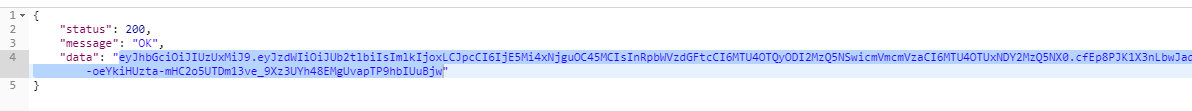
用postman，首先创建token，send方式用post，地址是

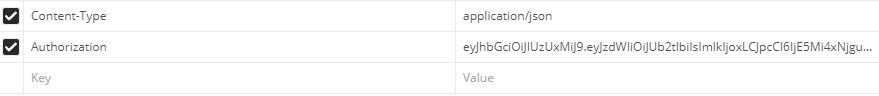


在swagger ui中找到认证授权相关，里面有一个地址以token结尾的

“/r/uaa/token根据用户登录帐号、密码获取Token，Token具有有效期（24小时），过期后如果可以自动刷新，则会返回新的Token，前端需拿到新token完成上一次的请求”

创建后，head不用加，body中包含账号密码，如下图：

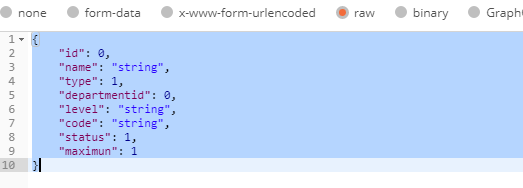
send以后，会有200，ok等显示并有一个data：“eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJUb2tlbiIsImlkIjoxLCJpcCI6IjE5Mi4xNjguOC45MCIsInRpbWVzdGFtcCI6MTU4OTQyODI2MzQ5NSwicmVmcmVzaCI6MTU4OTUxNDY2MzQ5NX0.cfEp8PJK1X3nLbwJadBKV6yTEtI1uNvv-oeYkiHUzta-mHC2o5UTDm13ve\_9Xz3UYh48EMgUvapTP9hbIUuBjw”

复制data中数据，创建一个新请求，send方式根据自己写代码用的delete/put/post/getmapping，地址为代码中接口写的地址



head中加入authorization，value值为复制的data数据

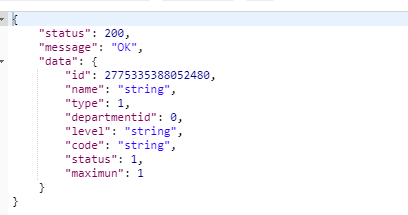
body中加入访问接口实体中对应写的表字段和数据



注意：上下两个栏body都是用json不用其他。

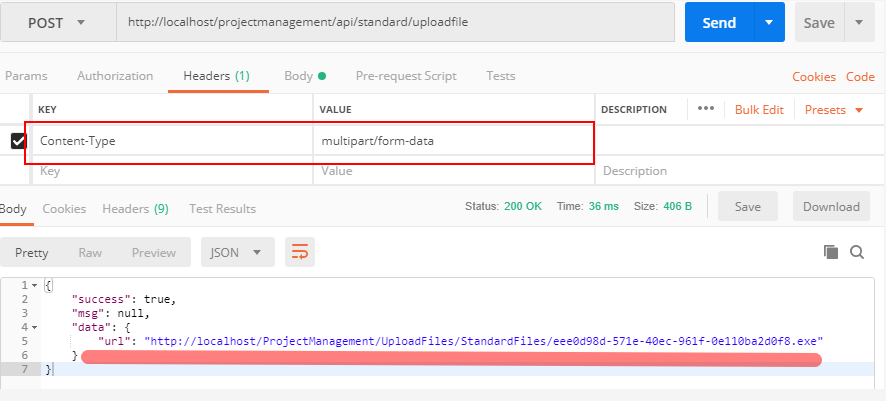
在代码中一开始用的<http://192.168.8.92:8761/eureka/>，后改为<http://localhost:8761/eureka/>，可以发送成功，结果如下

Create，findinfo

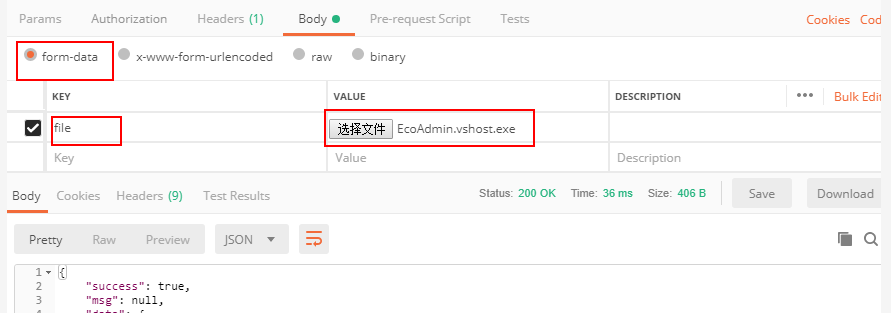


Postman请求上传文件

1. 选择post请求方式，输入请求地址，填写headers multipart/form-data



1. 填写body选择form-data，key选择file类型后，file后有个text/file隐藏选择，点击按钮选择文件，最后点击send



# JPA

JPA全称 java persistence api ,可以通过注解或者xml描述【对象-关系表】之间的映射关系，并将实体对象持久化到数据库中。作用就是 实现对象操作数据库，不用写SQL

ORM 使用对象操作数据库

1. ORM映射元数据：元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中。如@entity,@table,@column,@transient等注解
2. JPA的API：用来操作实体对象，执行CRUD操作，框架在后台替我们完成所有的事情，开发者从繁琐JDBC和SQL代码中解脱出来。如entitymanager.Merge（T t）
3. JPQL查询语言：通过面向对象而非面向数据库的查询语言查询数据，避免程序的SQL语句紧密耦合 如from student s where s.name = ?
4. 在@Query注解中编写JPQL实现DELETE和UODATE操作的时候必须加@Modifying注解，以通知Springdata这是一个DELETE或UPDATE操作。
5. UPDATE或DELETE操作需要使用事务，此时需要定义Service层，在Service层的方法上添加事务操作。
6. 注意JPQL不支持INSERT操作

JPA 仅仅是一种规范，也就是说JPA仅仅定义了一些接口，而接口是需要实现才能工作的，所以底层需要某种实现，而hibernate就是实现了JPA接口的ORM框架 JPA是一套ORM规范，hibernate实现了JPA规范

**什么是 spring data jpa**

Spring data jpa 是spring提供的一套简化JPA开发的框架，按照约定好的【方法命名规则】写dao层接口，就可以在不写接口实现的情况下，实现对数据库的访问和操作。同时还提供了很多处理CRUD之外的功能，如分页、排序、复杂查询。可以理解为JPA规范再次封装抽象，底层还是是哟了hibernate的JPA技术实现

接口约定命名规则：关键字and,equals,after,orderby,in，方法命名 findbyname,findbyidin，SQL语句

实例：//模糊查询 Standard findbyname(string name,integer length)

Springboot集成spring data jpa 只需要两步：

1. 导入maven坐标：

<dependency>

<Groupid>org.springframework.boot<>

<artifactid>spring-boot-starter-data-jpa<><>

1. Yml配置文件中的jpa信息
2. spring:  
    datasource:  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    url: jdbc:mysql://192.168.8.90:3306/aims?useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8  
    username: root  
    password: Comen\_2020  
    *#初始化数据* sql-script-encoding: UTF-8  
    schema: classpath\*:schema.sql  
    data: classpath\*:data.sql  
    continue-on-error: true  
    initialization-mode: *always #never,always* jpa:  
    hibernate:  
    ddl-auto: update  
    show-sql: true  
    redis:  
    host: 192.168.8.90  
    password: comen  
    port: 6379

实现模糊查询：findByNameLike（”%”+name+“%“）

## page分页模型

1. 常用方法：

总记录数：page.gettotalelements()

当前第几页：page.getnumber()

总页数：page.gettltalpages()

当前页面的list：page.getcontent()

当前页面的记录数：page.getnumber()

## Jpa注解

@Entity:

@Table(name=”“):

表明这是一个实体类。一般用于jpa ，这两个注解一般一块使用，但是如果表名和实体类名相同的话，@Table可以省略；

@MappedSuperClass:

用在确定是父类的entity上。父类的属性子类可以继承；

@NoRepositoryBean:

一般用作父类的repository，有这个注解，spring不会去实例化该repository；

@Column:

如果字段名与列名相同，则可以省略；

@Id:

表示该属性为主键；

@GeneratedValue(strategy=GenerationType.SEQUENCE,generator = “repair\_seq”):

表示主键生成策略是sequence（可以为Auto、IDENTITY、native等，Auto表示可在多个数据库间切换），指定sequence的名字是repair\_seq；

@SequenceGeneretor(name = “repair\_seq”, sequenceName = “seq\_repair”, allocationSize = 1):

name为sequence的名称，以便使用，sequenceName为数据库的sequence名称，两个名称可以一致；

@Transient:

表示该属性并非一个到数据库表的字段的映射,ORM框架将忽略该属性.

如果一个属性并非数据库表的字段映射,就务必将其标示为@Transient,否则,ORM框架默认其注解为@Basic；

@Basic(fetch=FetchType.LAZY):

标记可以指定实体属性的加载方式；

@JsonIgnore:

作用是json序列化时将java bean中的一些属性忽略掉,序列化和反序列化都受影响；

@JoinColumn(name=”loginId”):

一对一：本表中指向另一个表的外键。

一对多：另一个表指向本表的外键。

@OneToOne

@OneToMany

@ManyToOne:

对应Hibernate配置文件中的一对一，一对多，多对一。

@JsonFormat主要是后台到前台的时间格式的转换

@DataFormAT主要是前后到后台的时间格式的转换

@Slf4j注解实现日志输出

<https://github.com/mrdear/JavaWEB/tree/master/JPA-Demo/doc>

## Jpa详情

<https://www.cnblogs.com/z-sm/p/9176636.html>

## 环境配置

### 1.maven配置

首先需要spring相关架包,其实spring-data-jpa里面已经依赖了,如果你想用自己的版本则需要额外引入spring相关包.JPA实现还都是hibernate去实现的,所以还需要hibernate相关包.mysql就更不用说了.

<!--JPA start-->

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

<version>1.10.4.RELEASE</version>

</dependency>

<!--JPA end-->

<!--hibernate start-->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>${hibernate.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>${hibernate.version}</version>

</dependency>

<!--hibernate end-->

<!--mysql start-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>${mysql.version}</version>

</dependency>

<!--mysql end-->

### 2.整合Spring

整合Spring主要有以下几点要注意: 1.数据源配置 2.JPA提供者,JPA属性配置 3.事务配置 4.jpa:repositories 配置,具体如下代码:

<!--配置数据库连接池Druid-->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">

<!-- 数据库基本信息配置 -->

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

<property name="driverClassName" value="${jdbc.driver}" />

<property name="filters" value="${filters}" />

<!-- 最大并发连接数 -->

<property name="maxActive" value="${maxActive}" />

<!-- 初始化连接数量 -->

<property name="initialSize" value="${initialSize}" />

<!-- 配置获取连接等待超时的时间 -->

<property name="maxWait" value="${maxWait}" />

<!-- 最小空闲连接数 -->

<property name="minIdle" value="${minIdle}" />

<!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->

<property name="timeBetweenEvictionRunsMillis" value="${timeBetweenEvictionRunsMillis}" />

<!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->

<property name="minEvictableIdleTimeMillis" value="${minEvictableIdleTimeMillis}" />

<property name="validationQuery" value="SELECT 1" />

<property name="testWhileIdle" value="${testWhileIdle}" />

<property name="testOnBorrow" value="${testOnBorrow}" />

<property name="testOnReturn" value="${testOnReturn}" />

<property name="maxOpenPreparedStatements" value="${maxOpenPreparedStatements}" />

<!-- 打开removeAbandoned功能 -->

<property name="removeAbandoned" value="${removeAbandoned}" />

<!-- 1800秒，也就是30分钟 -->

<property name="removeAbandonedTimeout" value="${removeAbandonedTimeout}" />

<!-- 关闭abanded连接时输出错误日志 -->

<property name="logAbandoned" value="${logAbandoned}" />

</bean>

<!--第二步-->

<!--定义实体的工厂bean-->

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="dataSource" ref="dataSource"/>

<!--实体类位置-->

<property name="packagesToScan" value="cn.mrdear.entity"/>

<!--持久化单元名-->

<property name="persistenceUnitName" value="TestJPA" />

<!--JPA提供者-->

<property name="persistenceProviderClass" value="org.hibernate.ejb.HibernatePersistence"/>

<!--JPA属性-->

<property name="jpaProperties">

<props>

<!--配置方言-->

<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>

<!--激活查询日志功能-->

<prop key="hibernate.show\_sql">false</prop>

<!--优雅地输出Sql-->

<prop key="hibernate.format\_sql">false</prop>

<!--添加一条解释型标注-->

<prop key="hibernate.use\_sql\_comments">false</prop>

<!--配置如何根据java模型生成数据库表结构,常用update,validate-->

<prop key="hibernate.hbm2ddl.auto">none</prop>

</props>

</property>

</bean>

<!--第三步-->

<!--定义事务管理器-->

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="dataSource" ref="dataSource"/>

<property name="entityManagerFactory" ref="entityManagerFactory"/>

</bean>

<!--第四步-->

<!--定义repository接口的存放目录-->

<!--定义接口实现的后缀，通常用Impl-->

<!--定义实体工厂的引用-->

<!--定义事务管理器的引用-->

<jpa:repositories base-package="cn.mrdear.repository"

repository-impl-postfix="Impl"

entity-manager-factory-ref="entityManagerFactory"

transaction-manager-ref="transactionManager"/>

<!--第五步-->

<!--声明采用注解的方式申明事务-->

<tx:annotation-driven transaction-manager="transactionManager"/>

### 3.创建实体类

实体类中常用注解:

@Entity ：声明这个类是一个实体类

@Table：指定映射到数据库的表格

@Id ：映射到数据库表的主键属性,一个实体只能有一个属性被映射为主键

@GeneratedValue：主键的生成策略

@Column配置单列属性

@Entity//标识该为一个实体

@Table(name = "user")//关联数据库中的user表

public class User {

@Id//标识该属性为主键

private Integer id;

private String name;

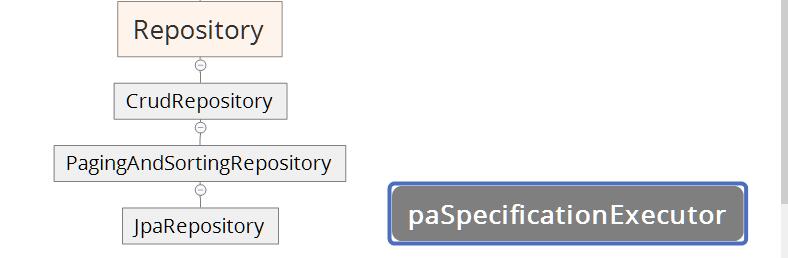
private String address;

private String phone;

//省略get和set

}

3.Repository接口

* Repository： 最顶层的接口，是一个空接口，目的是为了统一所有的Repository的类型，且能让组件扫描时自动识别
* CrudRepository: Repository的子接口，提供CRUD 的功能。
* PagingAndSortingRepository:CrudRepository的子接口, 添加分页排序。
* JpaRepository: PagingAndSortingRepository的子接口,增加批量操作等。
* JpaSpecificationExecutor: 用来做复杂查询的接口。

由图来看,一般使用JpaRepository这个接口做查询即可.这个接口拥有如下方法:

delete删除或批量删除

findOne查找单个

findAll查找所有

save保存单个或批量保存

saveAndFlush保存并刷新到数据库

创建repository

//User表示该Repository与实体User关联,主键类型为Integer

public interface UserRepository extends JpaRepository<User,Integer> {

}

这样就完成了一个基本Repository的创建,可以直接使用其中的方法,而不需要去写实现类.

### 4.测试

关于测试这里,我把测试案例写到test文件夹的话,总是报实体类未被JPA管理,所以改写到java文件夹,具体原因未知.

public static void main(String[] args) {

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring/applicationContext.xml");

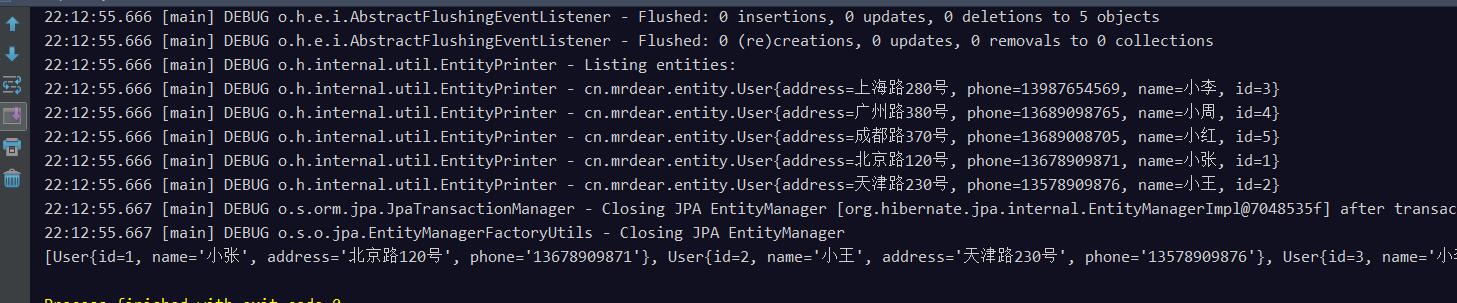
UserRepository userRepository = (UserRepository) applicationContext.getBean("userRepository");

System.out.println(userRepository.findAll());

System.out.println(userRepository.findOne(1));

System.out.println(userRepository.findAll(new Sort(new Sort.Order(Sort.Direction.ASC,"id"))));

}



## 增删改查

### 4.查询

单表查询,大部分都可以使用下面三种方法解决,多表联合查询的话,下面方法就不是很实用,下一节分析多表查询.

1.使用JpaRepository方法

//查找全部

userRepository.findAll();

//分页查询全部,返回封装了分页信息

Page<User> pageInfo = userRepository.findAll(new PageRequest(1, 3, Sort.Direction.ASC,"id"));

//查找全部,并排序

userRepository.findAll(new Sort(new Sort.Order(Sort.Direction.ASC,"id")));

User user = new User();

user.setName("小红");

//条件查询,可以联合分页,排序

userRepository.findAll(Example.of(user));

//查询单个

userRepository.findOne(1);

解析方法名创建查询

规则: find+全局修饰+By+实体的属性名称+限定词+连接词+ ...(其它实体属性)+OrderBy+排序属性+排序方向

/分页查询出符合姓名的记录,同理Sort也可以直接加上

public List<User> findByName(String name, Pageable pageable);

全局修饰： Distinct， Top， First 关键词： IsNull， IsNotNull， Like， NotLike， Containing， In， NotIn， IgnoreCase， Between， Equals， LessThan， GreaterThan， After， Before... 排序方向： Asc， Desc 连接词： And， Or

And --- 等价于 SQL 中的 and 关键字，比如 findByUsernameAndPassword(String user, Striang pwd)； Or --- 等价于 SQL 中的 or 关键字，比如 findByUsernameOrAddress(String user, String addr)； Between --- 等价于 SQL 中的 between 关键字，比如 findBySalaryBetween(int max, int min)； LessThan --- 等价于 SQL 中的 "<"，比如 findBySalaryLessThan(int max)； GreaterThan --- 等价于 SQL 中的">"，比如 findBySalaryGreaterThan(int min)； IsNull --- 等价于 SQL 中的 "is null"，比如 findByUsernameIsNull()； IsNotNull --- 等价于 SQL 中的 "is not null"，比如 findByUsernameIsNotNull()； NotNull --- 与 IsNotNull 等价； Like --- 等价于 SQL 中的 "like"，比如 findByUsernameLike(String user)； NotLike --- 等价于 SQL 中的 "not like"，比如 findByUsernameNotLike(String user)； OrderBy --- 等价于 SQL 中的 "order by"，比如 findByUsernameOrderBySalaryAsc(String user)； Not --- 等价于 SQL 中的 "！ ="，比如 findByUsernameNot(String user)； In --- 等价于 SQL 中的 "in"，比如 findByUsernameIn(Collection userList) ，方法的参数可以是 Collection 类型，也可以是数组或者不定长参数； NotIn --- 等价于 SQL 中的 "not in"，比如 findByUsernameNotIn(Collection userList) ，方法的参数可以是 Collection 类型，也可以是数组或者不定长参数；

嵌套实体:

主实体中子实体的名称+ \_ +子实体的属性名称 List findByAddress\_ZipCode(ZipCode zipCode) 表示查询所有 Address（地址）的zipCode（邮编）为指定值的所有Person(人员)

3.JPQL查询

一个类似HQL的语法,在接口上使用@Query标识

@Query("select a from user a where a.id = ?1")

public User findById(Long id);

使用@Modifying标识修改

@Modifying

@Query("update User a set a.name = ?1 where a.id < ?2")

public int updateName(String name, Long id);

携带分页信息:

@Query("select u from User u where u.name=?1")

public List<User> findByName(String name, Pageable pageable);

除此之外也可以使用原生sql,只需要@Query(nativeQuery=true)标识即可.

创建查询顺序:

创建查询的顺序 Spring Data JPA 在为接口创建代理对象时，如果发现同时存在多种上述情况可用，它该优先采用哪种策略呢？为此，jpa:repositories 提供了 query-lookup-strategy 属性，用以指定查找的顺序。它有如下三个取值： create --- 通过解析方法名字来创建查询。即使有符合的命名查询，或者方法通过 @Query 指定的查询语句，都将会被忽略。 create-if-not-found --- 如果方法通过 @Query 指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则通过解析方法名字来创建查询。这是 query-lookup-strategy 属性的默认值。 use-declared-query --- 如果方法通过 @Query 指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则抛出异常。

### 5.计数

计数就直接使用JpaRepository的count方法

//查找总数量

userRepository.count();

User user = new User();

user.setName("小红");

//条件计数

userRepository.count(Example.of(user));

### 6.判断是否存在

计数就直接使用JpaRepository的exists方法

//根据主键判断是否存在

userRepository.exists(1);

User user = new User();

user.setName("小红");

//根据条件判断是否存在

userRepository.exists(Example.of(user));

### 7.自定义查询

首先自定义一个接口,用于定义自定义方法,如UserRepositoryCustom

然后让UserRepository实现该接口,这样的话就可以使用其中的方法.

然后写UserRepositoryImpl实现UserRepositoryCustom接口

最后设置jpa:repositories的repository-impl-postfix="Impl",这样的话JPA会查找自定义实现类命名规则,这样的话JPA在相应UserRepository包下面查找实现类,找到则会使用其中的实现方法,而不去自己实现.

具体可以看项目demo,或者下一节的复杂查询

### 8.备注

###1.尽量避免getSingleResult

下面是该方法的部分源码,可以看出,如果查询出来的个数为0或者大于1都抛出异常,一般来说我们希望查找不到就返回null,而抛异常则是不想看到的,如果要使用,则需要try一下该代码块,在catch里面返回null.

try {

List he = this.query.list();

if(he.size() == 0) {

NoResultException uniqueResult1 = new NoResultException("No entity found for query");

this.getEntityManager().handlePersistenceException(uniqueResult1);

throw uniqueResult1;

} else if(he.size() > 1) {

HashSet uniqueResult = new HashSet(he);

if(uniqueResult.size() > 1) {

NonUniqueResultException nure = new NonUniqueResultException("result returns more than one elements");

this.getEntityManager().handlePersistenceException(nure);

throw nure;

} else {

return uniqueResult.iterator().next();

}

} else {

return he.get(0);

}

} catch (QueryExecutionRequestException var4) {

throw new IllegalStateException(var4);

} catch (TypeMismatchException var5) {

throw new IllegalArgumentException(var5);

} catch (HibernateException var6) {

throw this.getEntityManager().convert(var6);

}

# SpringBoot

<https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/html/index.html>

<https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/html/common-application-properties.html>

## spring45个注解

@SpringBootApplication:

包含@Configuration、@EnableAutoConfiguration、@ComponentScan通常用在主类上；

@Repository:

用于标注数据访问组件，即DAO组件；

@Service:

用于标注业务层组件；

@RestController:

用于标注控制层组件(如struts中的action)，包含@Controller和@ResponseBody；

@Controller:

用于标注是控制层组件，需要返回页面时请用@Controller而不是@RestController；

@Component:

泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注；

@ResponseBody:

表示该方法的返回结果直接写入HTTP response body中，一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，

加上@responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中；比如异步获取json数据，加上@responsebody后，会直接返回json数据；

@RequestBody:

参数前加上这个注解之后，认为该参数必填。表示接受json字符串转为对象 List等；

@ComponentScan:

组件扫描。个人理解相当于，如果扫描到有@Component @Controller @Service等这些注解的类，则把这些类注册为bean\*；

@Configuration:

指出该类是 Bean 配置的信息源，相当于XML中的，一般加在主类上；

@Bean:

相当于XML中的,放在方法的上面，而不是类，意思是产生一个bean,并交给spring管理；

@EnableAutoConfiguration:

让 Spring Boot 根据应用所声明的依赖来对 Spring 框架进行自动配置，一般加在主类上；

@AutoWired:

byType方式。把配置好的Bean拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作；

当加上（required=false）时，就算找不到bean也不报错；

@Qualifier:

当有多个同一类型的Bean时，可以用@Qualifier(“name”)来指定。与@Autowired配合使用；

@Resource(name=”name”,type=”type”)：

没有括号内内容的话，默认byName。与@Autowired干类似的事；

@RequestMapping:

RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径；

该注解有六个属性:

params:指定request中必须包含某些参数值是，才让该方法处理。

headers:指定request中必须包含某些指定的header值，才能让该方法处理请求。

value:指定请求的实际地址，指定的地址可以是URI Template 模式

method:指定请求的method类型， GET、POST、PUT、DELETE等

consumes:指定处理请求的提交内容类型（Content-Type），如application/json,text/html;

produces:指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回。

@GetMapping、@PostMapping等:

相当于@RequestMapping（value=”/”，method=RequestMethod.GetPostPutDelete等） 。是个组合注解；

@RequestParam:

用在方法的参数前面。相当于 request.getParameter；

@PathVariable:

路径变量。如 RequestMapping(“user/get/mac/{macAddress}”) ；

public String getByMacAddress(

@PathVariable(“macAddress”) String macAddress){

//do something;

}

参数与大括号里的名字相同的话，注解后括号里的内容可以不填。

## 全局异常处理

@ControllerAdvice:

包含@Component。可以被扫描到。统一处理异常；

@ExceptionHandler(Exception.class):

用在方法上面表示遇到这个异常就执行以下方法。

## springcloud

@EnableEurekaServer:

用在springboot启动类上，表示这是一个eureka服务注册中心；

@EnableDiscoveryClient:

用在springboot启动类上，表示这是一个服务，可以被注册中心找到；

@LoadBalanced:

开启负载均衡能力；

@EnableCircuitBreaker:

用在启动类上，开启断路器功能；

@HystrixCommand(fallbackMethod=”backMethod”):

用在方法上，fallbackMethod指定断路回调方法；

@EnableConfigServer:

用在启动类上，表示这是一个配置中心，开启Config Server；

@EnableZuulProxy:

开启zuul路由，用在启动类上；

@SpringCloudApplication:

包含

• @SpringBootApplication

• @EnableDiscovertyClient

• @EnableCircuitBreaker

分别是SpringBoot注解、注册服务中心Eureka注解、断路器注解。对于SpringCloud来说，这是每一微服务必须应有的三个注解，所以才推出了@SpringCloudApplication这一注解集合。

# Spring mail

基础知识：邮箱协议：

什么是SMTP？SMTP全称为Simple Mail Transfer Protocol（简单邮件传输协议），它是一组用于从源地址到目的地址传输邮件的规范，通过它来控制邮件的中转方式。SMTP认证要求必须提供账号和密码才能登陆服务器，其设计目的在于避免用户受到垃圾邮件的侵扰。

什么是IMAP？IMAP全称为Internet Message Access Protocol（互联网邮件访问协议），IMAP允许从邮件服务器上获取邮件的信息、下载邮件等。IMAP与POP类似，都是一种邮件获取协议。

什么是POP3？POP3全称为Post Office Protocol 3（邮局协议），POP3支持客户端远程管理服务器端的邮件。POP3常用于“离线”邮件处理，即允许客户端下载服务器邮件，然后服务器上的邮件将会被删除。目前很多POP3的邮件服务器只提供下载邮件功能，服务器本身并不删除邮件，这种属于改进版的POP3协议。

## 配置邮件服务

### 依赖：

<!--邮件模块依赖-->

<dependency>

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

<artifactId>spring-boot-starter-mail</artifactId>

</dependency>

### 配置：

spring.mail.host=smtp.163.com

spring.mail.username=your\_email@163.com

spring.mail.properties.from=bunianxiakai@163.com

spring.mail.password=your\_password # 注意这里不是邮箱密码，而是SMTP授权密码

spring.mail.port=465

spring.mail.default-encoding=UTF-8 ##我们使用的字符编码

# 如果设置false，可以不设置用户名和密码，当然也得看你的对接的平台是否支持无密码进行访问的。设置是否需要认证，如果为true,那么用户名和密码就必须的，

spring.mail.properties.mail.smtp.auth=true

# STARTTLS[1] 是对纯文本通信协议的扩展。它提供一种方式将纯文本连接升级为加密连接（TLS或SSL），而不是另外使用一个端口作加密通信。

spring.mail.properties.mail.smtp.starttls.enable=true

spring.mail.properties.mail.smtp.starttls.required=true

#设置为SSL协议

spring.mail.properties.mail.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory

spring.mail.properties.mail.imap.ssl.socketFactory.fallback=false

spring.mail.properties.mail.smtp.ssl.socketFactory.class=com.fintech.modules.base.util.mail.MailSSLSocketFactory

# 超时设置，防止阻塞，有些是无限时间

spring.mail.properties.mail.smtp.connectiontimeout=5000 spring.mail.properties.mail.smtp.timeout=3000 spring.mail.properties.mail.smtp.writetimeout=5000



需要先构建 SimpleMailMessage或 MimeMessage邮件信息类来填写邮件标题、邮件内容等信息，最后提交给JavaMailSenderImpl发送邮件

### 例子：

@Value("${spring.mail.username}")

private String from;

@Autowired

private JavaMailSender mailSender;

MimeMessage message;

/\*\*

\* 简单文本邮件

\* @param to 接收者邮件

\* @param subject 邮件主题

\* @param contnet 邮件内容

\*/

public void sendSimpleMail(String to, String subject, String contnet){

SimpleMailMessage message = new SimpleMailMessage();

message.setTo(to);

message.setSubject(subject);

message.setText(contnet);

message.setFrom(from);

mailSender.send(message);

}

/\*\*

\* HTML 文本邮件

\* @param to 接收者邮件

\* @param subject 邮件主题

\* @param contnet HTML内容

\* @throws MessagingException

\*/

public void sendHtmlMail(String to, String subject, String contnet) throws MessagingException {

MimeMessage message = mailSender.createMimeMessage();

MimeMessageHelper helper = new MimeMessageHelper(message, true);

helper.setTo(to);

helper.setSubject(subject);

helper.setText(contnet, true);

helper.setFrom(from);

mailSender.send(message);

}

/\*\*

\* 附件邮件

\* @param to 接收者邮件

\* @param subject 邮件主题

\* @param contnet HTML内容

\* @param filePath 附件路径

\* @throws MessagingException

\*/

public void sendAttachments(List<String> tos, String subject, String content, String attachmentName, InputStream in) {  
 message = mailSender.createMimeMessage();  
 try {  
 MimeMessageHelper helper = new MimeMessageHelper(message, true, "UTF-8");  
 helper.setFrom(from);  
 for (String to : tos) {  
 helper.setTo(to);  
 helper.setSubject(subject);  
 helper.setText(content, true);  
 if (!StringUtils.isEmpty(attachmentName)) {  
 helper.addAttachment(attachmentName, new ByteArrayResource(IOUtils.toByteArray(in)));  
 }  
 mailSender.send(message);  
 log.info("邮件已经发送。");  
 }  
 } catch (MessagingException | IOException e) {  
 log.error("发送邮件时发生异常！", e);  
 }  
}

/\*\*

\* 图片邮件

\* @param to 接收者邮件

\* @param subject 邮件主题

\* @param contnet HTML内容

\* @param rscPath 图片路径

\* @param rscId 图片ID

\* @throws MessagingException

\*/

public void sendInlinkResourceMail(String to, String subject, String contnet,

String rscPath, String rscId) {

logger.info("发送静态邮件开始: {},{},{},{},{}", to, subject, contnet, rscPath, rscId);

MimeMessage message = mailSender.createMimeMessage();

MimeMessageHelper helper = null;

try {

helper = new MimeMessageHelper(message, true);

helper.setTo(to);

helper.setSubject(subject);

helper.setText(contnet, true);

helper.setFrom(from);

FileSystemResource res = new FileSystemResource(new File(rscPath));

helper.addInline(rscId, res);

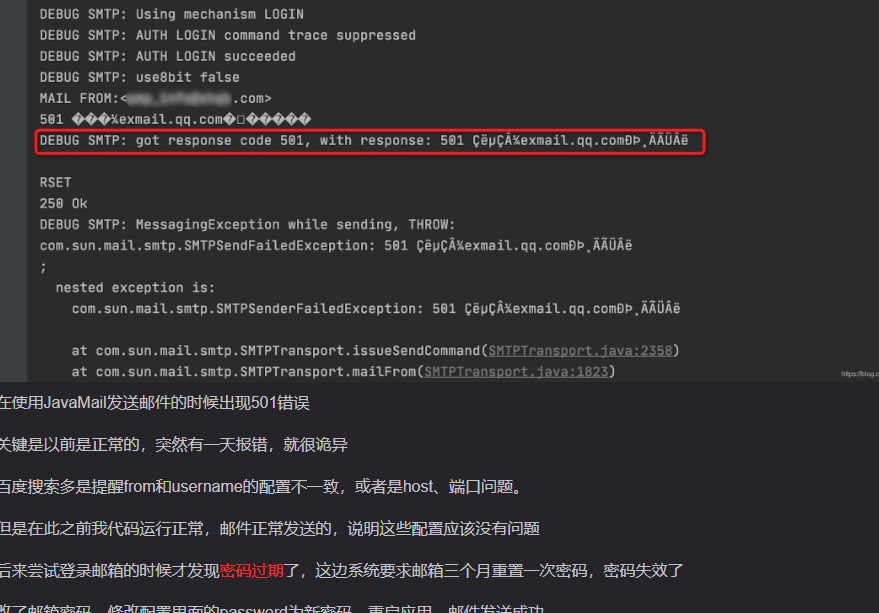
mailSender.send(message);

logger.info("发送静态邮件成功!");

} catch (MessagingException e) {

logger.info("发送静态邮件失败: ", e);

## 异常错误：



密码过期，像腾讯的邮箱需要二次验证，可能需要修改登录的密码



# 多线程

## 创建线程的四种方式

### 继承Thread类创建线程

1. 定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的任务，run()方法也称为线程执行体
2. 创建Thread子类的实例，也就是创建了线程对象
3. 启动线程，调用线程的start()方法

public class MyThread extends Thread{//继承Thread类

　　public void run(){

　　//重写run方法}

public class Main {

　　public static void main(String[] args){

　　　　new MyThread().start();//创建并启动线程

### 实现runable接口创建

1. 定义Runable接口的实现类，一样要重写run()方法，这个run()方法和Thread中的run()方法一样是线程的执行体
2. 创建Runable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象。
3. 调用线程对象的start()方法启动对象

public class MyThread2 implements Runnable {//实现Runnable接口

　　public void run(){

　　//重写run方法}

public class Main {

　　public static void main(String[] args){

　　　　//创建并启动线程

　　　　MyThread2 myThread=new MyThread2();

　　　　Thread thread=new Thread(myThread);

　　　　thread().start();

　　　//或者new Thread(new MyThread2()).start();

### 使用callable和futrue 创建线程

和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()方法功能要强大

》call()方法可以有返回值

》call()方法可以声明抛出异常

1. 创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例（从java8开始可以直接使用lambda表达式来创建Callable对象）
2. 使用FutrueTask类来包装Callable对象，该FutrueTask对象封装了Callable对象的call()方法返回值
3. 使用FutrueTask对象作为Thread对象的target创建并启动线程（因为FutrueTask实现了Runable接口）
4. 调用FutrueTask对象的get()方法来获得子线程执行结束后的返回值

public class Main {

　　public static void main(String[] args){

　　　MyThread3 th=new MyThread3();

　　　//使用Lambda表达式创建Callable对象

　　 //使用FutureTask类来包装Callable对象

　　　FutureTask<Integer> future=new FutureTask<Integer>(

　　　　(Callable<Integer>)()->{

　　　　　　return 5; });

　　　new Thread(task,"有返回值的线程").start();//实质上还是以Callable对象来创建并启动线程

　　 try{

　　　　System.out.println("子线程的返回值："+future.get());//get()方法会阻塞，直到子线程执行结束才返回

　　 }catch(Exception e){

　　　　ex.printStackTrace();

### 使用线程池如executor框架

在Java 5之后，通过Executor来启动线程比使用Thread的start方法更好，优点在于任务提交和执行解耦。Executor框架包括：线程池，Executor，Executors，ExecutorService，CompletionService，Future，Callable等。因此我们一般用executorService接口来实现和管理多线程。只需把Task描述清楚，然后提交即可。这个Task是怎么被执行的，被谁执行的，什么时候执行的，提交的人就不用关心了。具体点讲，就是提交一个Callable对象给ExecutorService（如最常用的线程池ThreadPoolExecutor），将得到一个Future对象，调用Future对象的get方法等待执行结果就好了。

1. public static ExecutorService newFixedThreadPool(int nThreads)

创建固定数目线程的线程池。（任意时间点最多只能有固定数目的活动线程存在，如果有新的线程建立，只能等待队列，直到当前的线程中某个线程终止被移出池子。没有时间idle，多用于服务器）

1. public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用execute将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。(适用于执行一些生存期很短的异步型任务，不用担心结束，超过TIMEOUT不活动，会自动被终止，没有考虑主机资源承受能力)

1. public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的Executor。（连续运行的任何事务，长期粗活的任务 ，监听进入的套接字连接

1. public staticScheduledExecutorService newScheduledThreadPool (int corePoolSize) 创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

newWorkStealingPool适合多任务队列，需并行操作的任务

//示例代码

public static void main(String[] args) {

//创建线程池

ExecutorService executorService = Executors.newFixedThreadPool(10);

ThreadPool threadPool = new ThreadPool();

for(int i =0;i<5;i++){

executorService.submit(threadPool); //为线程池分配任务}

executorService.shutdown();//关闭线程池

class ThreadPool implements Runnable {

@Override

public void run() {

for(int i = 0 ;i<10;i++){

System.out.println(Thread.currentThread().getName() + ":" + i);

实现Runnable和实现Callable接口的方式基本相同，不过是后者执行call()方法有返回值，后者线程执行体run()方法无返回值，因此可以把这两种方式归为一种这种方式与继承Thread类的方法之间的差别如下：

1、线程只是实现Runnable或实现Callable接口，还可以继承其他类。

2、这种方式下，多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。

3、但是编程稍微复杂，如果需要访问当前线程，必须调用Thread.currentThread()方法。

4、继承Thread类的线程类不能再继承其他父类（Java单继承决定）。

5、前三种的线程如果创建关闭频繁会消耗系统资源影响性能，而使用线程池可以不用线程的时候放回线程池，用的时候再从线程池取，项目开发中主要使用线程池

注：在前三种中一般推荐采用实现接口的方式来创建多线程

体系结构

Java.util.concurrent.Executor: 负责线程的使用与调度的根接口

ExcutorService 子接口：线程池的主要接口

ThreadPoolExecutor 线程池的实现类

ScheduledThreadPoolExecutor子接口：负责线程调度

ScheduledThreadPoolExcutor:继承ThreadPoolExecutor实现ScheduledPoolExcutor

工具类：Excecutor 就是上面共享池的四类

确保线程安全：

Synchronized/lock

做为关键字加在方法上

Public synchronized void dx（int j）

# 医疗知识

## IVD

体外诊断，指在人体之外通过对人体样本进行检测，而获取临床诊断信息，进而判断疾病或机体功能的产品和服务，从体内拿东西到体外检测

分类：生化诊断，免疫诊断，分子诊断，微生物诊断，血液诊断，POCT

光电比色法，比浊法

## 信息化

HIS 医院信息管理系统

AIMS手麻信息管理系统

EMR 电子病历管理系统

CIS 临床信息系统

PACS 影像归档和通信系统

LIS 实验室信息管理系统

RTS 放射科信息管理系统

HL7 卫生信息交换标准（Health level 7）HIS/RIS

SCP-ECG 欧洲计算机辅助心电图标准通信协议

DICOM 医学数字成像和通信协议标准 PACS/RIS

FDA-XML ECG的XML数据存储格式

# 设计模式

1. 使用设计模式是为了可重用代码、让代码更容易被理解、保证代码的可靠性，降低依赖，降低耦合
2. 创建型：工厂、抽象工厂、单例、建造者、原型 （提供了一种在创建对象的同时隐藏创建逻辑的方式，不适用new直接实例化对象）
3. 结构型：适配器、装饰器、过滤器，代理、外观、桥接、组合、享元（关注类和对象的组合，继承的概念被用来组合接口和定义组合对象获得新功能的方式）
4. 行为型：策略、模板方法、观察者、迭代器、责任链、命令、备忘录、状态、访问者、中介者、解释器、空对象（关注对象间的通信）

## **设计模式六大原则：**

1. 开闭原则：对扩展开放，对修改关闭，在程序需要进行拓展的时候，不能去修改原有的代码，需要使用接口和抽象类
2. 里氏代换原则（LSP）：任何基类可以出现的地方，子类一定可以出现，LSP时继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类基础上增加新行为。抽象化的具体实现，对实现抽象化的具体步骤的规范
3. 依赖倒转原则：开闭原则的基础。针对接口编程，依赖于抽象而不依赖于具体
4. 接口隔离原则：使用多个隔离的接口，比使用单个接口要好，降低类之间的耦合度
5. 迪米特法则（最少知道）：一个实体应当尽量少的与其他实体之间发生相互作用
6. 合成复用原则：尽量使用合成聚合的方式，而不是继承

## 工厂

凡是出现大量产品创建，并且具有共同的接口时可以通过工厂模式。复杂对象适合工厂方法。第一种传入字符串有无不能正确创建对象，第三种不要实例化工厂类，大多数选择第三种。类的创建依赖工厂类，拓展程序必须对工厂类修改违背闭包原则。

1. 主要解决接口选择的问题，当明确地计划不同条件下创建不同实例时使用。

普通工厂：建立工厂类，对实现了同一接口的一些类进行实例的创建，如果传递的字符串出错，则不能正确创建对象。创建两个实现类，和二者的共同接口。最后创建工厂类，在类中返回相应的实体类

多个工厂：提供多个工厂方法，分别创建对象。和上面一样，但是创建工厂类不做判断，而是本来在工厂类里面的一个方法，拆分成两个方法，用到的时候需要哪个就返回哪个。

静态工厂方法：将上面的多个工厂里的方法设置为静态（在方法前面加static），不需要创建实例（不要工厂 名字=new 工厂方法），直接调用即可

### 步骤

1. 创建一个接口（实体类共有的 1接口）
2. 创建实现接口的实体类（可在类中重写接口 1类）
3. 创建一个工厂，生成基于给定信息的实体类的对象（1工厂）
4. 使用该工厂，通过传递类型信息来获取实体类的对象（1具体实现类）

## 抽象工厂

创建多个工厂类。这样一旦需要增加新功能，直接增加新工厂，不需要修改之前的代码。无需指定他们具体的类

1. 主要解决接口选择问题，系统的产品有多于一个的产品族，而系统值消费其中某一族的产品，聚合多个同类产品

### 步骤

1. 创建一个接口，创建实现接口的实体类（1接口，1类）
2. 创建另一个接口，创建实现这一个接口的实体类（1接口，1类）
3. 为这两个对象创建抽象类来获取工厂（1抽象工厂类）
4. 创建扩展了抽象类的工厂类，基于给定的信息生成实体类的对象（2工厂类）
5. 创建一个工厂创造器类，通过传递信息来获取工厂（1工厂创造器类）
6. 使用工厂创造器来获取抽象工厂，通过传递类型信息来获取实体类的对象（1具体实现方法类）

## 单例模式（singleton）

单例对象能保证在一个jvm中该对象只有一个实例存在，自己创建唯一实例。单例类必须给所有其他的对象提供这一实例。需要定义大量的静态常量和静态方法。大型的对象创建类频繁，系统开销大，省去new操作符降低内存使用频率，类如交易所的核心交易引擎，控制着交易流程，创建多个不能保证核心交易服务器独立控制整个流程

1. 主要解决一个全局使用的类频繁地创建与销毁，想要控制实例数目，节省系统资源时使用。构造函数私有
2. 没有接口不能继承，与单一职责原则冲突。

### 步骤

1. 创建一个singleton类（private static 一个对象 = new （），构造函数私有，获取唯一可用对象）从singleton类获取唯一对象（1单例类 1实现类）

### 单例模式的几种实现方式

//todo 懒汉式 饿汉式。。

## 建造者模式

1. 使用多个简单对象一步一步构建成一个负责对象，面临创建一个复杂对象，用于在一些基本部件不会变，而其组合经常变化的时候。产品必须有共同点，内部变化复杂会有很多建造类。
2. 与工厂模式的区别是：建造者更加关注方法调用的顺序，工厂注重创建产品。建造者可用创建复杂产品，工厂都一样。
3. 建造者不止要创建产品，还需要知道有哪些部件组成，工厂模式不需要知道里面的细节

## 原型模式

用于创建重复的对象，当创建对象代价较大时。原型模式通过拷贝一个现有对象生成新对象，浅拷贝实现cloneable重写，深拷贝通过实现serializable读取二进制流。