1 分布式架构实现

后台笔者采用spring cloud全家桶作为我们的分布式基础架构。

1.1 注册中心

Eureka是spring cloud的重要组成部分之一，也是广为使用的spring cloud注册中心。使用eureka，注册中心是作为一个单独的项目存在的，因此首先，笔者新建了一个spring boot工程，命名为eureka-server，让它作为整个工程的服务注册与发现中心。

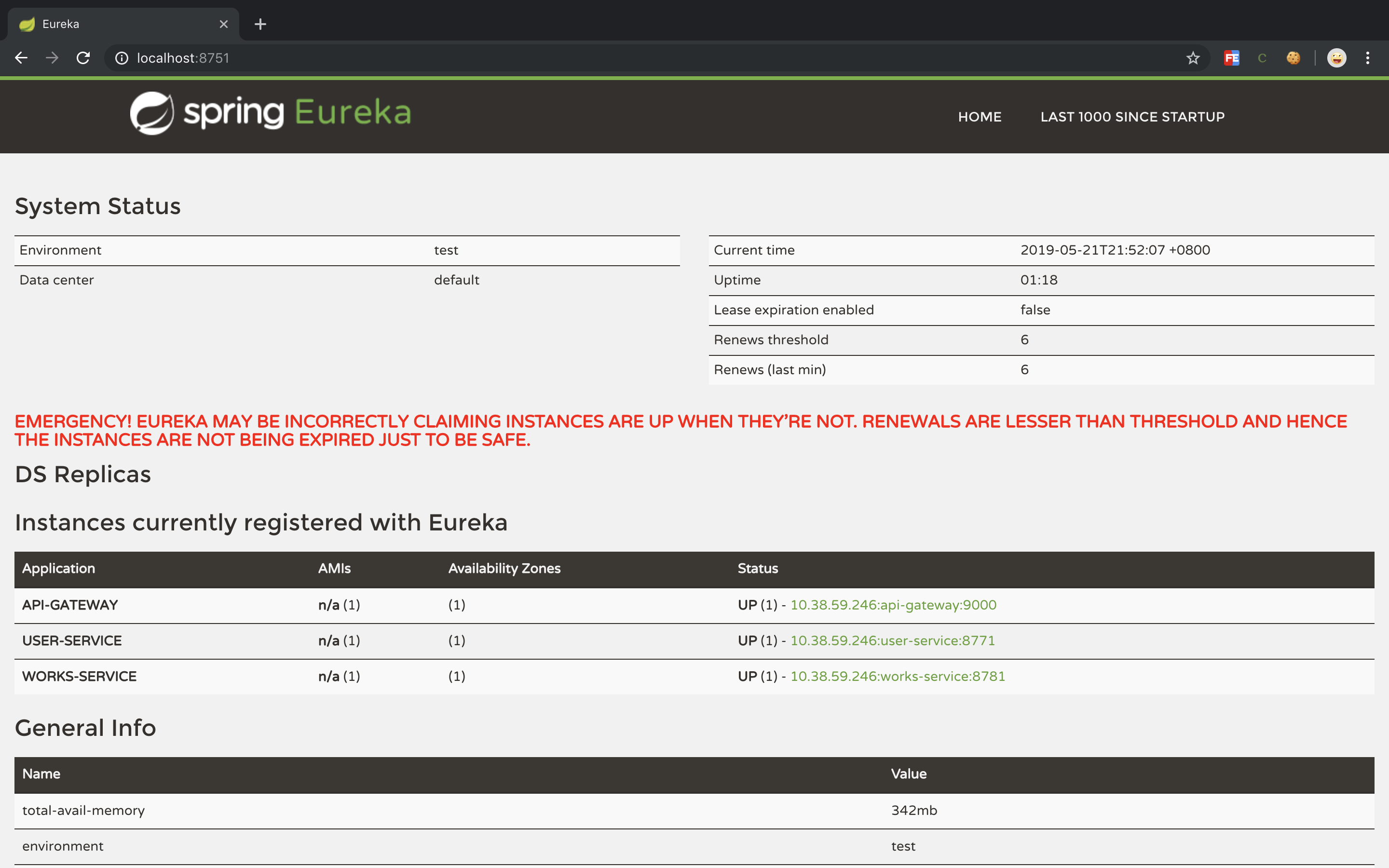
首先，需要加入eureka相关的maven依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-eureka-server</**artifactId**> </**dependency**> |

然后需要在项目的启动类上面加入@EnableEurekaServer注解，这样，注册中心的雏形就出来了，但是这还不够。我们需要让服务的消费者和生产者知道注册中心的地址，这样才能进行注册，因此笔者在这里配置了它的连接URL，如以下代码中是一个本地化的典型配置：

|  |
| --- |
| **server:  port:** 8751 **eureka:  client:** *#以下两条声明自己是个服务端* **registerWithEureka: false  fetchRegistry: false  service-url:  defaultZone:** http://${**eureka.instance.hostname**}:${**server.port**}/eureka/  **instance:  hostname:** localhost |

经过上述操作之后，启动项目，我们即可通过hostname(域名)和port(端口号)去访问到eureka默认的管理界面了，如下图所示：



1.2 服务提供者(provider)和消费者

Spring cloud的设计思想是把业务的不同模块进行划分，然后单独管理、各自部署，例如在我们的系统中就包括用户模块、作品模块以及用户行为模块等等，这些模块划分之后，每个模块都会低耦合地用到其他模块，比如用户发布作品同时需要获取到用户信息，用户作品被点赞同时也需要获取到作品信息，所以每个模块既是服务提供者，同时也是服务消费者。

要成为一个服务消费者或生产者，我们需要首先声明它是eureka的一个客户端，这样它才能去向注册中心进行注册，对此我们需要加入以下依赖，顾名思义，它就是声明本工程是一个eureka客户端，可以去eureka注册中心进行注册：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**> </**dependency**> |

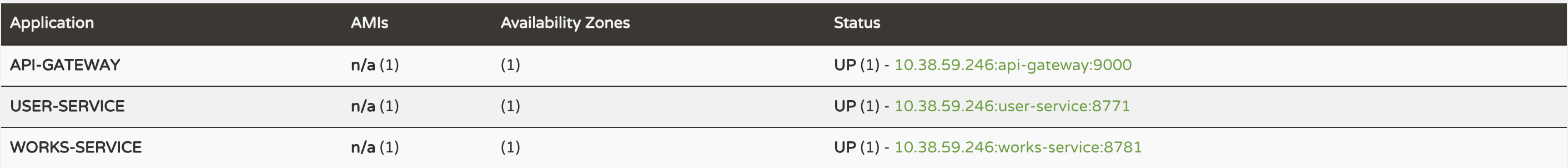
注册的时候，我们首先需要知道eureka的部署地址，然后才能去指定注册中心进行注册：

|  |
| --- |
| **server:  port:** 8781  *#指定注册中心地址* **eureka:  client:  serviceUrl:  defaultZone:** http://localhost:8751/eureka/ |

同时为了让eureka知道当前是谁进行注册，并且为了以后多节点部署的时候可以明确某个客户端的调用关系，我们还需要声明当前项目注册到eureka的时候所用的名称，这样之后，请求客户端在后期进行调用的时候可以根据名称来进行请求，然后eureka根据同一命名的多个节点按照设置的负载均衡策略随机选取一个节点进行调用：

|  |
| --- |
| **spring:** *#本服务的名称* **application:  name:** works-service |

经过以上之后，一个eureka的客户端就建好了，现在去访问eureka的管理页面，就可以看到新注册的eureka客户端了，如下图所示笔者共建立了三个eureka客户端：



1.3 模块之间相互调用

在本章第二节中我们讲到，spring cloud各模块之间会存在相互调用，相互调用的方式有很多种，例如dubbo基于rpc进行远程调用，相反，spring cloud是使用http的方式进行调用的。它的原理很简单：就是模拟我们的客户端发送一个http请求，然后解析请求的到的数据。

Spring cloud提供了两种调用http API的方法，分别的ribbon和fegin，其中ribbon是比较底层的方法，它直接通过RestTemplate去发起http请求，类似于HttpClient，如以下代码所示：

|  |
| --- |
| Map<String,Object> productMap = **restTemplate**.getForObject(**"http://product-service/api/v1/product/find?id="** + productId, Map.**class**); |

由于ribbon需要自己去构造http请求，步骤比较繁琐，因此这里被笔者弃用，反而选用了fegin。Fegin采用接口的方式去进行请求，在响应的接口上加上注解表明请求的url即可完成模拟发送http请求的任务。

使用fegin，首先我们要加入如下依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-openfeign</**artifactId**> </**dependency**> |

然后需要在启动类上面加入@EnableFeignClients注解，表明它是一个fegin请求的客户端。这样配置之后我们就可以进行相应的调用了。首先接口提供者(被调用者要实现一个接口，进行具体的操作)：

|  |
| --- |
| @RequestMapping(**"/find"**) **public** Object findById(**long** id) **throws** InterruptedException {  User user = **userService**.findById(id);  **return** user;  } |

在调用方，我们只需要实现一个接口，让它的参数与上面的api参数一致，并用注解声明请求的url即可：

|  |
| --- |
| @FeignClient(name = **"product-service"**) **public interface** UserClient {   */\*\*  \* 通过id查找,跟user-service的参数要一致  \** ***@param id*** *\** ***@return*** *\*/* @GetMapping(**"/api/v1/user/find"**)  String findById(@RequestParam(value = **"id"**) **long** id); } |

使用的时候，只需要把相应的调用接口注入到我们的服务中即可，使用方法跟本地的接口调用方法几乎完全一致：

|  |
| --- |
| @Autowired **private** UserClient **UserClient**; |

1.4 服务降级

为了提高系统的可靠性，防止系统因某个模块不可用而崩溃，笔者加入了合适的降级策略，spring cloud也提供了一个完善的降级组件----hystrix。

使用hystrix首先同样是加入hystrix依赖，但是由于fegin中已经内置了hystrix，因此我们可以不必加入，但如果需要使用hystrix的相关注解的话，则必须引入它：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-hystrix</**artifactId**> </**dependency**> *<!--hystrix Dashboard 仪表盘监控依赖-->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-hystrix-dashboard</**artifactId**> </**dependency**> |

hystrix是为了降低模块之间调用失败而导致的系统crash的可能性，因此，它一般用于http调用其他模块的接口中，首先我们要在启动类中开启hystrix注解：

|  |
| --- |
| @EnableCircuitBreaker @EnableHystrixDashboard |

然后需要定义一个请求失败时的回调类，它直接实现相应的调用接口，在这里面，我们可以添加入报警机制，例如短信、邮件通知等：

|  |
| --- |
| @Component **public class** UserClientFallBack **implements** UserClient {  @Override  **public** String findById(**long** id) {  System.***out***.println(**"feign 调用 user-service findById 异常"**);   *//****TODO 添加报警机制* return null**;  } } |

接着在对应的调用接口中，要声明回调类：

|  |
| --- |
| @FeignClient(name = **"user-service"**,fallback = UserClientFallBack.**class**) **public interface User**Client {  @GetMapping(**"/api/v1/user/find"**)  String findById(@RequestParam(value = **"id"**) **long** id); } |

这样，就可以实现一个简单的降级机制了。

以上的降级方法是接口层，同时hystrix也支持api层的降级，在笔者的工程中，这种降级方法同样用了很多。api层的降级主要是方法级别的，为每个api设置各自的降级策略(以下代码并不是全部代码)：

|  |
| --- |
| @HystrixCommand(fallbackMethod = **"saveCommenFail"**) **public** Object saveByFeign(**long** userId, **long** commentId, HttpServletRequest request) {Map<String, Object> result = **new** HashMap<>(2);  result.put(**"code"**, 0);  result.put(**"data"**, **commentService**.saveByFeign(userId, comment));  **return** result; } |

在响应的Controller类中，我们需要实现对应的回调方法，方法名就是以上代码中的fallbackMethod的值，以下模拟了一个发送短信的方法：

|  |
| --- |
| */\*\*  \* 方法签名（参数）一定要和api方法一致,不管用不用  \*  \** ***@param userId*** *\** ***@param commentId*** *\** ***@return*** *\*/* **private** Object saveCommentFail(**long** userId, **long** commentId, HttpServletRequest request) {  *//****TODO 添加报警机制，通知相关人员（电话、短信、邮件等），用缓存（例如redis等）去存储是否已经发送过（具有时效性）*** String saveCommentKey = **"save-comment"**;  String sendValue = **redisTemplate**.opsForValue().get(saveCommentKey );   *//开启一个线程异步调用* **new** Thread(() -> {  **if** (StringUtils.*isEmpty*(sendValue)) {  System.***out***.println(**"发送短信，保存失败，请立刻查找原因"**);  *//****TODO 发送一个http请求，调用发送短信服务接口* redisTemplate**.opsForValue().set(saveCommentKey, **"save-comment-fail"**, 20, TimeUnit.***SECONDS***);  } **else** {  System.***out***.println(**"已发送短信,20s内不重复发送"**);  }  }).start();   Map<String, Object> msg = **new** HashMap<>(2);  msg.put(**"code"**, -1);  msg.put(**"msg"**, **"接口异常"**);   **return** msg; } |

我们也可以通过设置超时时间来打开服务降级阈值，当超过一定时间没有获取到返回值的时候，则自动转入提供的降级策略中，配置方法如下：

|  |
| --- |
| *#hystrix超时时间调整，默认为1s，与上面feign的超时时间相互独立* **hystrix:  command:  default:  execution:  isolation:  thread:  timeoutInMilliseconds:** 1000 |

1.5 网关

网关负则对外提供统一的接口，这样用户就无法得知系统内部的调用细节，极大提高了系统的安全性。另外统一接口对于前端开发人员来说也是十分便利的。

Spring cloud同样提供了完善的网关组件----zuul，它也是作为一个独立的项目进行部署的。同样的，使用zuul我们首先要加入依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-zuul</**artifactId**> </**dependency**> |

从以上依赖中我们可以发现，zuul也需要eureka-client依赖，这表明它也是一个eureka客户端，需要向eureka注册中心进行注册。

接下来，在启动类中加入注解开启网关：

|  |
| --- |
| @EnableZuulProxy |

然后配置它注册到eureka的名称：

|  |
| --- |
| **server:  port:** 9000  *#服务名称，网关服务* **spring:  application:  name:** api-gateway  *#指定注册中心地址* **eureka:  client:  serviceUrl:  defaultZone:** http://localhost:8761/eureka/ |

启动项目之后，现在我们应该就可以在eureka的管理页面中看到它了。

接下来我们需要配置地址映射，把实际请求的url映射到网关中：

|  |
| --- |
| **zuul:  routes:  user-service:** /apigateway-user/\*\* *#路径要唯一，不同的服务不要冲突，因为是以map的方式存储的* **works-service:** /apigateway-works/\*\*  **behavior-service:** /apigateway-behavior/\*\* |

为了提高系统的安全性，我们同时需要关闭原来接口的访问权限：

|  |
| --- |
| **zuul:  routes:**  **ignored-patterns:** /\*-service/api/\*\* *#正则匹配* |

这样配置之后，就只能通过9000/apigateway-\*/进行访问了。

1.6 负载均衡

fegin默认采用了平均的负载均衡策略，即每个节点被访问的可能性相同，如果要自定义负载均衡，可以通过ribbon的LoadBalancerClient来实现，由于笔者的项目中并未涉及到这些，因此这里就不再进行赘述。

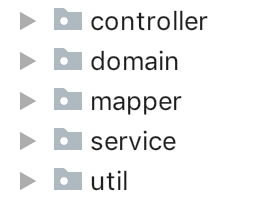
1. 持久层框架集成

持久层框架笔者选用了mybatis的升级版mybatis-plus，它对mybatis进行了很好的扩展和封装，通过sql语句构造器大大减少了sql的书写。

首先要引入mybatis-plus的依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>com.baomidou</**groupId**>  <**artifactId**>mybatis-plus-boot-starter</**artifactId**>  <**version**>3.0.7.1</**version**> </**dependency**> |

然后进行项目分层：包括：DAO层负则与数据库进行交互，做一些基础的增删改查等工作；Serivce层负责业务逻辑；Controller层负责接口调用；Domain层负责存放持久层数据：



笔者是通过xml来与dao层进行映射的。在配置文件中，配置dao层与sql语句的映射：

|  |
| --- |
| **mybatis-plus:  type-aliases-package:** com.miujike.worksservice.domain  **mapper-locations:** classpath:/sqlMappers/\*.xml |

这样配置之后，系统会自动把xml文件中定义的sql语句通过id映射到对应的dao层的接口中。例如：

|  |
| --- |
| */\*\*  \* 拉取用户music列表（分页）  \** ***@param map*** *userId, start, end  \** ***@return*** *list  \*/* List<Map<String,Object>> getUserMusicList(Map<String,Object> map); |

对应的查询语句为：

|  |
| --- |
| <**select id="getUserMusicList" parameterType="map" resultType="map"**>  select music\_id as musicId,music\_url as musicUrl,singer ,  original\_singer as originalSinger,play\_times as playTimes,cover,  title,duration  from miu\_music  where user\_id = #{userId}  <**if test='lastId != 0'**>  <![CDATA[  and music\_id < #{lastId}  ]]>  </**if**>  order by music\_id desc  limit #{fetchNum} </**select**> |

在xml中，我们需要声明它对应的命名空间，也就是对应的Dao层接口：

|  |
| --- |
| <**mapper namespace="com.miujike.worksservice.mapper.MusicMapper"**> |

为了让系统知道要做映射的Dao层的位置，需要在启动类中加入如下注解开启扫描，这样，系统会自动地进行增删改查的映射：

|  |
| --- |
| @MapperScan(**"com.miujike.worksservice.mapper"**) |

这样，持久层框架就配置成功了。要注意的是，在model层中，需要将java object与数据库的表进行映射，并对每个字段进行相应的映射，其中表字段和object属性相同的，可以省略映射的注解：

|  |
| --- |
| @TableName(**"miu\_music"**) **public class** Music **extends** Model<Music> {  **private static final long *serialVersionUID*** = 1L;  @TableId(value = **"music\_id"**, type = IdType.***AUTO***)  **private long musicId**;  @TableField(**"user\_id"**)  **private long userId**;  } |

1. 小程序端：

3.1 组件和模版的使用

由于小程序中存在很多相同的布局方式，笔者选用了小程序的自定义组件或模板来简化代码量，并提高复用率。其中，模版只是视图层的复用，没有js文件；而组件却有自己的Comment类，在里面可以定义属性，声明方法，因此，如果存在相同逻辑的，笔者都选用了组件，比如粉丝列表和关注列表，组件里面都有分享、关注/取关等逻辑，因此在这部分笔者选用了组件；而对于搜索框等，由于要搜索的内容不同，相应的处理逻辑也就不同，因此笔者选用了模版，然后在各自的Page类中定义相应的处理逻辑。

(1) 组件

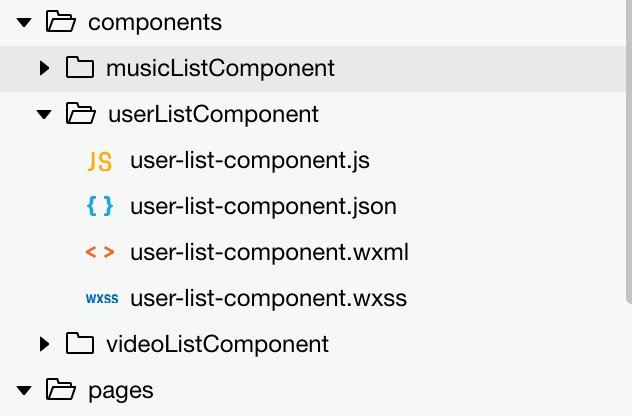
使用组件，首先我们要定义一个组件，它与普通页面的目录结构一模一样，都包含四个文件。然后在对应的json配置文件中要声明它是一个组件：

|  |
| --- |
| {  "component": true  } |

同样，在调用的页面里面，也要声明被调用的组件：

|  |
| --- |
| {  "usingComponents": {  "user-list-component": "/components/userListComponent/user-list-component"  },  } |

这里的组件的目录层次是从小程序的最外层开始的，例如以上路径对应的组件路径如下图所示，其中components与pages都在最外层，这里也是笔者花了挺长时间找到的一个规则：



在使用的时候，我们只需要在页面中加入组件对应的文件名称作为标签即可，例如以上组件对应的标签为<user-list-component></user-list-component>，为了与小程序组件命名的方式一致，笔者采用了单词间加“-”的命名方式。

关于向组件传递参数，要注意的是，这里存在命名规则的转换，例如在组件中接收的参数为，它符合驼峰命名法：

|  |
| --- |
| properties: {  fromWhere: String  }, |

对应传递的参数是：

|  |
| --- |
| <user-list-component from-where="{{fromWhere}}"></user-list-component> |

笔者也尝试过不用该方法的映射，结果总是报错的。

在组件中，还有一个小细节是在笔者开发过程中发现的：组件中的css样式不能直接使用app.xcss中定义的样式，如果要使用，只能通过相对路径的方式来引入，这可能也是因为笔者组件存放的位置不对，笔者也未曾深究过：

|  |
| --- |
| @import "../../app.wxss"; |

(2) 模版

模版相对来说比较容易些，只需要在wxml和xcss中引入模板文件，并在需要的地方加入<template is=”...” data=”{{...}}”/>即可，这里有一个小细节，在传递多个data的时候，只需要在data后面的双括号中用逗号分隔开就行。另外与组件不同，模版可以直接使用app.xcss中的文件。

3.2 首页

首页效果图如下图所示：



其中，最上面是一个轮播图，用小程序的swipper可以很方便的实现，但是从图中可与发现，笔者加入了缩放的功能，单使用swipper组件是无法实现这个功能的，因此笔者在该组件的基础上做了进一步拓展，用了css的动画来实现图片在移动的过程中平滑的进行缩放：

|  |
| --- |
| .new-video-swiper swiper-item .one-new-item {  width: 420rpx;  height: 244rpx;  transition: transform 1s;  position: relative;  }  .new-video-swiper swiper-item .cur-show {  transform: scale(1.2, 1.2);  -webkit-transform: scale(1.2, 1.2);  } |

其中，transition属性表明元素的过度动画，在其中笔者动画使用的是transform，它的作用是如果当前容器是显示的容器，则让它的宽高各变为原来的1.2倍，再配合transition就可以实现在1秒内平滑的放大到原来的1.2倍，这里的1秒，正是我们在swipper中定义的切换时间：

|  |
| --- |
| <swiper class="new-video-swiper"  indicator-dots="true"  autoplay="true"  interval="3000"  duration="1000">  </swiper-item> |

3.3 播放页面

播放页面包括视频播放页面和音乐播放页面，效果图如下图所示：

其中，音乐播放器是笔者自制的一个播放器，利用了小程序的音乐播放接口wx.createInnerAudioContext()，它会创建一个InnerAudioContext上下文对象，通过它我们可以操控音乐的开始与暂停，至于音乐的切换，笔者选择用setData({})直接修改当前正在播放的音频文件:

|  |
| --- |
| onNext(e){  let that = this;  let index = e.currentTarget.dataset.index;  this.setData({  curMusic: that.data.musicList[index];  })  } |

对于滑动改变进度，笔者采用了小程序的slider来实现，并在音乐播放的过程中每隔一段很少的时间就改变一下它的进度。

这样，一个带切换和播放暂停的音乐播放器就实现了。

对于视频播放器，笔者采用了小程序原生的视频播放器，原生的播放器本身就具有很强的控制能力，因此没必要再花费精力去自己实现。

3.4 作品上传

作品上传包括视频上传和音频上传两大模块，效果图如下图所示：

其中视频上传笔者直接从相册里面选取文件，这方面可以调用微信的接口很容易实现。对于音频上传，小程序没有开放音频选择的接口，实际上，在主流的IOS和android系统中，也不会存在音频直接上传，因此笔者在这里选择先进行录音，然后上传的方式，录音与暂停，微信小程序是提供接口的：

对于文件的上传，微信同样提供了丰富的上传接口：

|  |
| --- |
| wx.chooseImage({  success: function(res) {  that.setData({  coverImg:res.tempFilePaths  })  wx.uploadFile({  url: '',  filePath: res.tempFilePaths,  name: 'file',  })  },  }) |

其中res.tempFilePaths代表文件的临时路径，name字段表明这是一个文件，那么在服务端，通过@RequestParam(**"file"**)我们可以接收到该文件，并完成上传操作，对应代码如下所示：

|  |
| --- |
| @ResponseBody @PostMapping(**"/uploadInMini"**) **public** Map<String, Object> uploadFile(@RequestParam(**"file"**) MultipartFile uploadFile) {  Map<String, Object> map = **new** HashMap<>();  *// 上传是否成功的判断标识，0-成功，1-失败* String url = saveUploadFile(uploadFile, uploadFile.getOriginalFilename());  **log**.info(**"url=======>"** + **QINIU\_BASE\_PATH** + url);  map.put(**"url"**, **QINIU\_BASE\_PATH** + url);  map.put(**"error"**, 0);  **return** map; } |

3.5 加载和刷新

小程序提供了上拉加载和下拉刷新接口，需要在配置文件中开启：

|  |
| --- |
| {  "backgroundTextStyle": "dark",  "enablePullDownRefresh": true  } |

但是这种情况只适用于页面中不是scroll-view布局的情况，对于scroll-view，上述的配置将会失效，对应的接口也同样会失效，因此对于scroll-view的上拉加载和下拉刷新，笔者选用了scroll-view的两个属性bindscrolltoupper负责实现下拉加载，bindscrolltolower实现上拉刷新。

在scroll的使用过程中，需要开启滑动的方向(scroll-y或者scroll-x)，否则被视为无效，将不会产生滑动。

3.5 分页加载

笔者加入了简单分页加载的能力，之所以说它是简单的分页加载，因为它并不是完全意义上的根据页数和每页的条数来实现的，而是根据自增长的属性来实现。比如，我们按照id排序并进行加载，这样在每次加载过程中，我们只需要记录已加载的数据中的最后一条，然后拿它去与数据库中的数据做比较，找出比它大/小的n条数据即可，至于如何确定加载完成，很简单，当某次加载的数量不足一次性最大加载条数时，就默认为加载完成即可。这样的话，对于有删除方法的列表的加载来说将会异常简单，我们无需去判断删除了多少条，然后下次加载的时候计算偏移量，我们只需要做的就只有找到剩下数据中的最后一条即可：

|  |
| --- |
| Wx.request({  url:"apigateway-user/api/v1/user/fan/list/"+app.globalData.userInfo.userId,  data: {  lastId: lastId,  },  success(res){  wx.hideLoading();  wx.stopPullDownRefresh();  that.setData({  userList: that.data.userList.concat(res.data.data)  }  if (res.data.data.length < app.globalData.fetchNum) {  that.setData({  tipShow: true **// 代表加载完成**  })  }  }  }) |