**基于微信小程序的微服务化健康助手系统开发**

**摘 要**

当下，互联网产品层出不穷，人们也变得异常忙碌，关乎个人健康的意识也似乎正在被忙碌的人群所唤醒。此刻需要一款同社会同类产品不同的软件产品。去解决已有系统的遗留问题，去解决用户的使用诉求。因此本项目致力打造一套关乎个人健康的微信小程序，并通过运维界面提升运维人员简单运维的目的方便运维人员进行服务弹性扩容。项目采用微服务架构思想，力争在完成项目的同时为用户提供更加完美的体验和健壮的服务。

在技术层面：项目采用微服务架构思想，将原有的庞大单体应用拆分成多个服务模块，方便服务模块动态扩容以提升节点负载能力；采用网关对流量合法性进行验证以及节点压力监控；项目采用中心化配置，项目中较为多变的配置发布在Github中，采用Config实时监控配置变化并实时推送给对应的服务模块。避免重新打包上线。

**关键词**  微服务；运维；微信小程序；健康

**Development of micro service health assistant system based on wechat applet**

**Abstract**

At present, Internet products emerge in an endless stream, people are also very busy, and the awareness of personal health seems to be awakened by busy people. Now we need a software product different from the same kind of social product. To solve the legacy problems of the existing system, to solve the user's use demands. Therefore, the project is committed to creating a set of wechat small programs related to personal health, and improving the simple operation and maintenance of operation and maintenance personnel through the operation and maintenance interface to facilitate the flexible expansion of service for operation and maintenance personnel. The project adopts the idea of microservice architecture, striving to provide users with more perfect experience and robust services while completing the project.

At the technical level: the project adopts the micro service architecture idea to split the original huge single application into multiple service modules, which is convenient for the dynamic expansion of service modules to improve the node load capacity; the gateway is used to verify the legitimacy of traffic and monitor the node pressure; the project adopts the centralized configuration, and the more changeable configuration in the project is published in GitHub, and the config real-time monitoring is used Configuration changes are pushed to the corresponding service module in real time. Avoid repacking to go live.

**Keywords** microservice; operation and maintenance; WeChat applet; health

目 录

[第1章 绪论 5](#_Toc41564538)

[1.1 项目背景 5](#_Toc41564539)

[1.2 现状分析 5](#_Toc41564540)

[1.3 主要工作 6](#_Toc41564541)

[1.4 目的和意义 6](#_Toc41564542)

[1.5 研究内容 7](#_Toc41564543)

[1.6 章节介绍 7](#_Toc41564544)

[1.7 小结 8](#_Toc41564545)

[第2章 系统架构 9](#_Toc41564546)

[2.1 语言选型及原因 9](#_Toc41564547)

[2.1.1项目采用Java语言原因如下 9](#_Toc41564548)

[2.1.2项目采用VueJs的原因如下 9](#_Toc41564549)

[2.2 框架选型 10](#_Toc41564550)

[2.3 微服务 11](#_Toc41564551)

[2.4 总体架构 12](#_Toc41564552)

[2.5 具体技术 13](#_Toc41564553)

[2.5.1 GateWay流量网关 13](#_Toc41564554)

[2.5.2 Eureka注册发现中心 14](#_Toc41564555)

[2.5.3 Kafka消息队列 15](#_Toc41564556)

[2.5.4 Druid数据库连接池 15](#_Toc41564557)

[2.5.5 Hystrix熔断与Dashborad仪表盘 16](#_Toc41564558)

[2.5.6配置文件动态配置 16](#_Toc41564559)

[2.5.7 服务RPC与Http 17](#_Toc41564560)

[2.5.8 定时任务框架 17](#_Toc41564561)

[2.6 项目难点 18](#_Toc41564562)

[2.7 技术官网文档汇总 18](#_Toc41564563)

[2.8 硬、软件开发环境 18](#_Toc41564564)

[2.9 小结 19](#_Toc41564565)

[3.1 需求分析 20](#_Toc41564566)

[3.2 用例分析 21](#_Toc41564567)

[3.2.1 小程序端用例分析 21](#_Toc41564568)

[3.2.2运营PC端 22](#_Toc41564569)

[3.3 数据库设计 23](#_Toc41564570)

[3.4 界面设计 25](#_Toc41564571)

[3.4.1 小程序界面设计 25](#_Toc41564572)

[3.4.2 网页PC端 25](#_Toc41564573)

[3.5 总体功能模块 25](#_Toc41564574)

[3.6 小结 27](#_Toc41564575)

[第4章 详细设计 28](#_Toc41564576)

[4.1 简述 28](#_Toc41564577)

[4.2 小程序免密登录 28](#_Toc41564578)

[4.3 小程序请求封装 31](#_Toc41564579)

[4.4 Gateway流量鉴权 32](#_Toc41564580)

[4.5 Kafka记录流量 33](#_Toc41564581)

[4.6 邮件告警 34](#_Toc41564582)

[4.7 SpringBoot统一配置 36](#_Toc41564583)

[4.8 节点健康状态 38](#_Toc41564584)

[4.9 小结 40](#_Toc41564585)

[5.1 简述 41](#_Toc41564586)

[5.2 小程序免密登录测试 41](#_Toc41564587)

[5.3 阅读后端发布文章 42](#_Toc41564588)

[5.4 悄悄话模块 43](#_Toc41564589)

[5.5 基本信息 44](#_Toc41564590)

[5.6 每日目标 45](#_Toc41564591)

[5.7 健康周报 46](#_Toc41564592)

[5.8 划任务 46](#_Toc41564593)

[5.9 系统帮助 47](#_Toc41564594)

[5.10 节点健康检查 48](#_Toc41564595)

[5.11 仪表盘 49](#_Toc41564596)

[5.12 Druid统计 49](#_Toc41564597)

[5.13 Eureka中心 49](#_Toc41564598)

[5.14 食物重量定制化 50](#_Toc41564599)

[5.15 小结 50](#_Toc41564600)

[第6章 项目部署 51](#_Toc41564601)

[6.1 简述 51](#_Toc41564602)

[6.2 配置JDK 51](#_Toc41564603)

[6.3 配置Zookeeper 51](#_Toc41564604)

[6.4 配置Kafka 51](#_Toc41564605)

[6.5 部署业务模块 52](#_Toc41564606)

[6.6 小结 53](#_Toc41564607)

[第7章 总结 54](#_Toc41564608)

[致 谢 55](#_Toc41564609)

[参 考 文 献 56](#_Toc41564610)

**第1章 绪论**

## 1.1 项目背景

逢毕设项目编写之际，想利用用个人所学结合在校期间学习的Java编程、Web编程、数据库设计，设计编写出高可用、性能良好、分区容错性高的软件项目。

毛主席曾经说过：“一个人要想做成一件事，必须拥有多方面的素质，但所有这些都必须依赖于一个前提条件：健康的体魄”。现如今，新一代年亲人越来越多的走向办公室，日常的办公都在相对狭隘的空间下完成无限的脑力活动。越来越少的年轻人靠体力劳动去实现自身价值。

社会正在飞速的发展，城市的生活节奏也变得非常的快，自然要求年轻一代花费更多的时间与精力投入到工作当中，熬夜工作，不吃早餐为了赶时间等情况比比皆是。

与之相比最可怕的是心理疾病。现如今，抑郁症等相关心理疾病也在逐渐控制着越来越多的人。

再此背景之下，真正做出一款UI良好、用户体验感良好、内容科学的软件去解决上述所提出的问题迫在眉睫。

## 1.2 现状分析

出于上述的背景之下。在开发之前还需要进行现有类似软件的调研，去研究现有软件的优点及缺点对于后期的项目设计与开发非常有帮助。

打开手机商城，搜索“健康”类别的软件比比皆是。

分别从项目UI、项目易用性、体验感、内容科学性等多个方面试用市面上已存在的项目。UI和体验感方面都非常的好，但基本都未为用户生成类似于用户报表的相关功能模块，这就导致用户在使用该软件的时候无法及时形象的的收到个人相关反馈。比如某用户跟着软件进行健康生活的作息但一段时间之后并未收到个人训练的成果报表，这很可能让使用者慢慢的丧失对健康兴趣。

从功能全面性来看也有不少相关软件功能性缺失。比如只推荐健康饮食但是没有根据用户的健康状况去推荐。再比如，在健康标准中只选择了某一种算法，而实际上在国际上不同的人种的健康标准其实都不一样，健康的算法自然也就不一样。

因此，总结上述的问题，列出几个问题如下：

1. 健康算法种类不全，计算不严谨；
2. 没有及时反馈用户成果，使得用户丧失信息；
3. 文章内容没有及时的更新，部分内容不具有科学性；
4. 平台运维迭代工作较低。

## 1.3 主要工作

出于上述背景，决定做出一款健康类别的软件，包括饮食健康类别的软件，这其中包括身体健康和心理健康。尽可能的在该主题中满足大多数用户的需求，与此同时尽可能解决市面上已有类似项目的弊端。

微信的用户量巨大涉及到各个行业，不论是工作还是休闲生活，微信的用户群里只会越来越大。因此项目依托微信小程序进行开发，使得项目成立之初就具有大批量用户的潜能。

项目采用了微服务架构，所用技术多为目前主流、新颖的技术。在开发过程中，在项目中添加了大量关于运维的功能点。比如SQL记录、SQL黑名单拦截；服务节点健康状况查询；Hystrix Dashborad服务压力展示等相关技术。

在项目完全开发完毕之后进行一系列科学测试，在覆盖项目所有功能模块之外尽可能的覆盖项目所有分支，以保证项目的正确性。在测试优化项目之前编写每个功能对应的测试用例。项目使用百盒测试、黑盒测试、冒烟测试等相关测试技术手段进行全方位测试。

最终交付一篇论文用于对项目设计、开发、技术选型以及相关技术的特点、技术架构进行详细说明。

本次毕设将交付出一款“健康”类别的小程序及相关的运维平台，并命名为“惜己”小程序。

## 1.4 目的和意义

出于上述的背景分析，以及目前现状的分析，做出一款真正意义上的“健康”小程序与PC界面运维端。从用户需求入手，从方便运维方向进行入手，力争做出一款方便使用者进行健康事宜活动。使得使用者方便的使用项目，方便的查看个人的身体健康状态以及个人每日完成任务的折线图。

在运维方向，可以方便的看到各个服务节点的状况以及压力情况，方便运维人员选择性的对相关服务模块进行扩容，通过Eureka进行注册，从而达成负载均衡效果，以减轻对应服务的负载压力。

## 1.5 研究内容

本毕业设计以微服务架构思想结合Spring“全家桶”、“VueJs”、“微信小程序”一起做出一套带有用户端、运维端的健壮性良好的平台。

在开发中学习并将SpringCloud相关组件的学习及时的运用在项目中。

项目着重于微服务架构的设计思路，分别从流量合法性验证、流量网关、流量路由、注册发现中心、服务健壮性仪表盘、SQL执行分析与统计、流量存储与统计（借助Kafka中间件）等来进行对项目健壮性的加固以及流量的实时分析以追求良好的用户体验以及良好的运维体验。

分别对上述提到的技术进行简短说明，在本论文的详细设计会有对该项技术的详细介绍：

1. Zuul Gateway网关：用来做流量统一入口合法性鉴权并路由用户请求；
2. Eureka注册中心：用来协调各个微服务节点；
3. Druid数据源：开源于阿里巴巴，是最快且功能最全的数据源，内置的Web平台可以方便运维；
4. Kafka消息队列：存储流式数据，某些场景下并不需要将数据持久化进入数据库，这样的场景往往需要保存在消息队列即可；
5. Hystrix Dashborad：用于服务压力展示以及服务熔断，方便运维人员及时发现可能宕机的服务并及时进行扩容以达到高压力节点负载均衡的效果。

## 1.6 章节介绍

本论文一共分为七个章节，下面分别对着七个章节内容进行

项目的编写与测试，项目最终的打包公网的部署流程以及使用项目核心功能的展示与使用说明详见第六章。最后关于项目的编写感受，所领悟的知识，项目展望详见第七章。

1. 第一章：针对需求分析、主要工作、现状分析、研究内容、章节介绍来进行项目背景介绍；
2. 第二章：从系统架构详细描述系统的技术选型，该章节是技术相关核心内容；
3. 第三章：概要设计方面描述数据库设计、UI设计、用例图、系统功能模块等进行说明；
4. 第四章：详细的描述系统相关功能是如何开发，描述技术代码是如何实现；
5. 第五章：项目测试，测试各个功能模块功能能否正确无误的运行；
6. 第六章：项目部署，因为项目为多结点项目，因此有必要进行节点部署、服务环境搭建的说明；
7. 第七章：总结，包括项目的编写新的包括对以后的展望和认识现在的不足。

## 1.7 小结

在本章节中，从需求方面以及目前现状两个方面分析了该毕设项目的意义与重要性。从技术方面规划了项目研究内容，之后的开发围绕着这些技术点进行开发。在本章节的最后对本论文章节进行简短说明。

# 第2章 系统架构

## 2.1 语言选型及原因

项目采用Java后端开发，VueJs以及微信小程序开发作为前端开发语言。

### 2.1.1项目采用Java语言原因如下

1.Java语言的普遍性，社区群体庞大，网上解决方案很多；

2.Java的顺风车可以直接对接Spring全家桶，可以开箱即用Spring提供的很多解决方案，这里当然包括SpringCloud微服务的一站式解决方案；

3.Java语言的灵活性：Java不仅可以编译加载已经编写好的Java代码，更可以在运行期间动态的创建对象并调用其中的方法，这项技术就是Java提供的强大反射机制，是一种从Class文件反射成Java文件的技术；

4.Spring提供的AOP技术可以方便快捷的补充Java的面向对象思想，AOP是一种面向切面编程。其AOP的流行也得益于Java的动态代理技术；

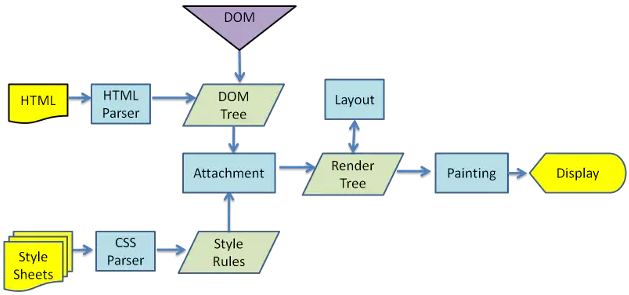
5.一次编写到处运行，写好的程序打包之后不仅可以在Windows OS运行，也可以在Linux OS运行，这得益于Java的JVM技术，因为JVM的存在使得Java代码可以翻译成不同操作系统可识别的指令集合。

### 2.1.2项目采用VueJs的原因如下

在前端的发展历程中，经历了几次改革，原生JS操作的复杂，譬如原生JS中操作DOM元素需要大量的代码，前端页面代码会显得特别臃肿与代码膨胀，在这之后有各大厂商基于JS封装自己的库，这其中JQuery就是封装的比较完美的JS库，JQuery虽然开发起来简洁很多，但使用JQuery的开发效率也并不算高，再加上JQuery封装了大量的API，需要开发者有较强的记忆性。VUEJS的出现几乎解决了上述的问题，VUEJS基于观察者模式的MVVN模型使得页面几乎不需要操作任何的DOM元素而是简单的使用模板语言就可以解决。

上述是从开发者角度而言，下面从不同脚本语言浏览器渲染性能进行对比：

在了解浏览器渲染之前首先需要知道浏览器渲染页面的流程，如下图，是浏览器渲染HTML文件的过程如图2-1所示：



**图2-1 webkit渲染流程**

对上图简述为五个步骤：①根据HTML文件分析标签构造一颗DOM树；②CSS分析器分析CSS文件和HTML标记中的Inline样式（css内联样式）并根据优先级原则进行取舍，最终生成页面样式表；③将DOM树和样式表进行匹配关联生成一颗Render树；④浏览器根据生成的Render树进行唯一确定元素在浏览器页面的准确坐标轴；⑤根据元素坐标轴进行Paint绘制。经过上述步骤最终渲染出整个页面。

在明白上述的渲染原理之后还需要理解两个概念：重排（ReFlow）、重绘（Repaint）。其中ReFlow更加的影响页面的性能。在如下情况会导致页面发生重排：①加、减DOM元素；②元素的位置发生改变影响了原局部DOM树；③DOM尺寸发生变化；④页面初始化；⑤窗口大小的改变。

明白了浏览器渲染原理与ReFlow、Repaint原理之后回顾JS与JQuery代码中经常有大量DOM树的操作。这样一来页面的ReFlow次数将会非常多，页面性能开销巨大显得网页卡顿现象严重。但VUEJS内部已经对其进行了优化，在收到DOM元素更新时，其会先维护在一个队列中并对重排的CSS和新的页面唯一进行合并，最终进行一次重排，这样大大节省了页面的性能开销。

## 2.2 框架选型

项目后端使用基于Spring框架。具体而言后端使用了SpringBoot框架开发，SpringBoot相较于其他的Java后端框架有配置少、内置Tomcat等优势，使得开发后端变得容易。除此之外项目使用SpringCloud作为微服务一站式解决方案。

项目前端使用IViewUi、ColorUi、Vant组件库或CSS库，方便前端快速开发，可以在短时间内完成界面风格统一等优势。

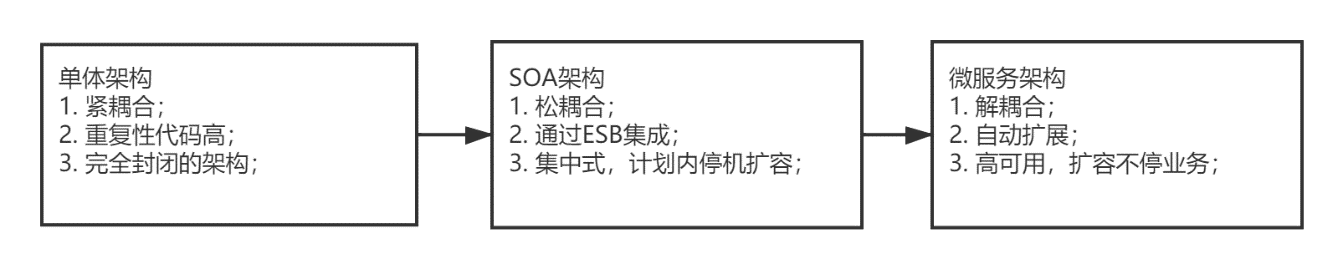
## 2.3 微服务

现代应用中，为了更好的应用健壮性，微服务的思想也逐渐成为了流行的架构思路，因此有必要花合适的篇幅对微服务思想进行描述。

微服务思想由Martin Fowler提出，概述其提出的微服务为一句话：将原来庞大的单体应用拆分成一组小型服务，在服务和服务之间采用HTTP协议的RestFul API进行通信，最终为用户创造价值。

微服务是一种开发思想并不是落地的某项技术，微服务的诞生完全颠覆了以往的单体应用的开发模式。

下图展示了软件架构的演变过程，如图2-2：

****

**图2-2 服务架构演变**

因为微服务的特性，其优势也不言而喻：①避免出现庞大、复杂的单体“怪兽”；②专门的团队做专业的服务（框架设计、数据库的选型不会被局限为一种）；③拆分出的每个微服务独立部署互不影响，相互之前通过远程任务调度框架协调工作共同完成用户的请求。这里列出几个常见的RPC、Http框架：Alibaba厂商的Dubbo、SpringCloud封装的FeignClient、大众点评的XXL-RPC等等；④使得上线项目的弹性扩容成为可能。

在此可以做一个简单的假设，可以更加直观的了解微服务的优势：

假设有一个后端服务，其对外提供着成百上千个接口，假设突然有一天服务器宕机，这可能导致所有的接口无法对外提供快速、优质的服务，且造成大量请求数据丢失，那么后果将不堪设想。

当然，微服务架构并不是没有缺点：①原有的庞大单体应用拆分成多个服务，因此拆分的粒度不易掌握，业界也没有明确的参考论证；②微服务应用是分布式系统，其有固有的开发复杂性，服务和服务之间需要通过RPC调度，而不像Java代码中的进程调度；③数据库的一致性设计（尤其的数据一致性要求很高的系统中，如证券、交易平台）对于开发者而言将是更大的挑战；④如果服务之间没有处理得当，很容易发生可怕的【服务雪崩】效应。

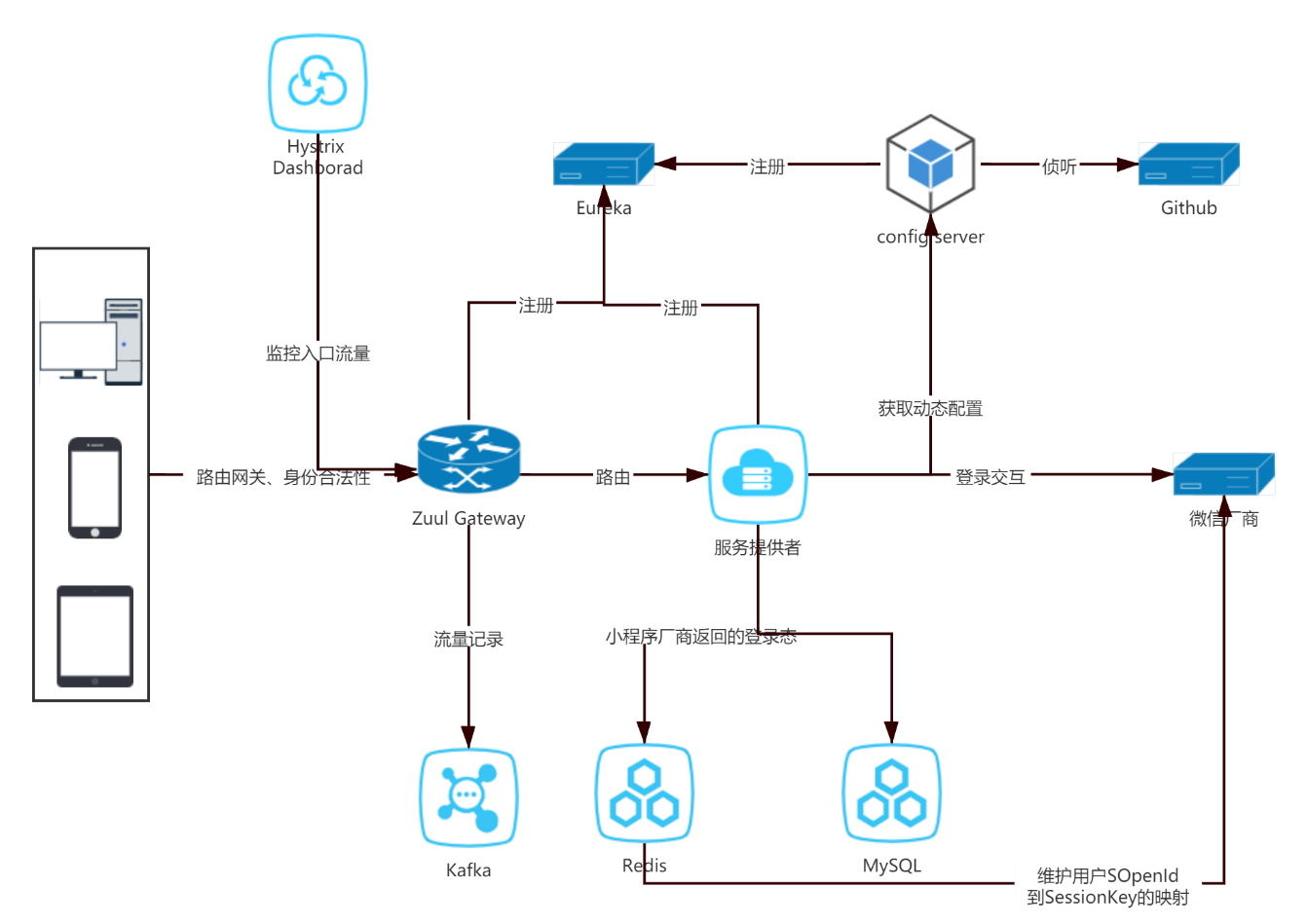
此处，对服务雪崩效应做出简单说明（前提开发者未对服务做特殊处理）：服务相互之间调用，当某个服务突然变得不可用时，那么依赖他的服务都会出现请求等待状态指导服务超时被销毁，这就是请求积压，但需要知道的是，服务在请求等待状态时依然会消耗CPU资源。假设有一万个请求都依赖于某个已经坏掉的服务，那么这一万个请求会进入请求等待状态并消耗着服务器资源，久而久之导致Java出现可怕的OutOfMemeory Exception.最终导致该服务坏掉，同样的道理这样的恶性现象一旦出现会逐渐让整个微服务都变得不可用，这是一个渐变的过程。这就是可怕的【服务雪崩效应】。

## 2.4 总体架构

1. 项目采用目前主流的前后端分离模式开发，该模式有很多好处，譬如：前后端完全解耦、前端工程化开发完全不依赖于后端、升级版本可减少对方停机损失... ...可以说前后端分离是传统软件开发漫漫长路上具有划时代的意义的大事件。

2. 项目采用微服务的架构。

下图为项目架构图：



**图2-3 项目架构图**

使用文字对上述框架进行简述：

所有的流量统一从Gateway进入，对用户的身份进行合法性认证，并使用Kafka消息中间件对流量进行记录以及Hystrix Dashborad对入口流量监控。若身份合法就会进入服务提供者，服务提供者根据获取到的用户信息去Redis中查询该用户是否已经免密登录微信服务，若是，则直接提供服务，若不是则返回登录页面。因为服务和服务之间需要相互调度，这里采用Eureka服务注册发现中心维护各个服务的关系表。

## 2.5 具体技术

### 2.5.1 GateWay流量网关

Zuul图标：



**图2-4 Zuul图标**

Zuul在业界中一直被称为“门户神兽”，其提供着各类型的Filter为系统的安全性和系统容错性提供了强大的支撑，并且Zuul内聚整合了Hystrix提供了强大的总入口的服务降维处理。在SpringCloud整合的Zuul中还包括了Ribbon的负载均衡，Ribbon提供的重试机制以及配置的超时时间与Hystrix配置的熔断时间相结合，可以使得项目在满足负载均衡的同时阻止服务雪崩的现象。

GateWay会根据流量请求的地址去Eureka（微服务下的注册发现中心）中查找服务的真实地址并引导流量进行路由到真实的服务提供者，当然这些流程对用户而言都是无感知的，是透明的。

### 2.5.2 Eureka注册发现中心

Eureka图标：



**图2-5 Eureka图标**

Eureka是常见的服务注册发现中心，它的作用在于分布式中，当多个服务提供者不在同一个服务中，那么相互之间的通信如果全部依赖于IP地址加端口加服务路径那么将变得非常麻烦，如果将来服务迁移到了其他服务器，那么依赖它的服务都需要需要手动的修改IP地址以及端口号，这样就导致需要经常弹性扩容和服务迁移变得非常的麻烦。因此希望我们能够给服务取一个别名，然后让这个别名和服务的Ip地址进行动态的绑定并且需要动态的感知到IP地址的变化，这样的需求就需要Eureka的支持。

可以讲Eureka简单的理解为网络中的DNS。

当然，满足该需求的除了Eureka技术，还有Zookeeper技术。

EurekaVSZookeeper

在著名的分布式架构思想中著名的CAP原则，其中C表示强一致性，A表示高可用性，P表示分区容错性。在这三个字母中，不论是什么样的技术也只能实现其中的两个，要么是CP，亦或是AP。对于Zookeeper而言其选择的是CP，它需要保证高强度的一致性，这就导致，当Zookeeper正在选择Leader或者超过半数以上的Zookeeper宕机时，为了保证整个项目的高强度一致性，Zookeeper不会对外提供服务，那么整个项目都会变得不可用。与Zookeeper不同的是，Eureka选择了AP，在Eureka的世界里，项目的服务可用性远比不可用要强的多。项目考虑到服务的高可用性选择放弃了“强一致性”但不代表放弃了“一致性”原则。因此选择了Eureka作为服务注册中心。

### 2.5.3 Kafka消息队列

图标：



**图2-6 Kafka图标**

在项目中使用了Kafka消息中间件，消息中间件有很多比如此处的Kafka还有ActiveMQ... ... 在这其中被使用最多的就是Kafka了，从源码开源的GitHub上来看，Kafka的Start数为15.7k远远比其他MQ消息队列要多得多。

Kafka主要应对与数据流式处理的场景。比如日志、流量统计等场景。对于这些数据而言，并不适合将数据落盘到关系型数据库或者是非关系型数据库，因为这些数据并不重要到浪费磁盘空间或者浪费内存。这些数据只适合于保存一段时间（Kafka对某条数据默认保存168小时），消费者可以对最近一段时间的数据进行消费统计分析。

### 2.5.4 Druid数据库连接池

图标：暂无图标

在项目数据源连接池技术选型中，需要先明白连接池的作用。

数据源连接池的概念可以先类比到线程池的概念，这些概念都可以统称为“池”类技术，“池”类技术的诞生都源于性能消耗的考虑，比如创建一个线程、创建一个数据源对象都是非常消耗性能的一件事，但是整个项目不可能就只有一个线程一个数据源连接，因为这样是无法应对高并发的场景。因此，项目中可以创建一个“池”，可以在初始化阶段为这些池创建很多对应的实例，这样下面需要使用到这些实例的时候直接从对应的线程池中去获取即可。而这些实例的个数往往大于1个，这样又可以满足高并发的基础需求，也不至于在今后的每次使用之前都需要手动的创建实例。既省开销又满足一定的并发需求。

在所有的数据源连接池中（比如：C3P0、DBCP、Druid、），Druid是兼顾性能与功能的完美结合体，Druid内置的Filter，比如WallFilter用于SQL防注入，StatFilter用于监控统计，LogFilter用于日志记录或者慢查询记录。甚至可以自定义Filter用于操作。

其中StatFilter非常强大，Druid基于它可以统计各种SQL查询报表并内置运维界面，使得运维人员可以非常清晰的了解到在一段时间中执行了哪些SQL，以及分布等情况。

也正是出于上述的考虑，项目使用了Druid数据源。

### 2.5.5 Hystrix熔断与Dashborad仪表盘

图标：



**图2-7 Hystrix图标**

在本论文第一章中，详细的阐述了服务雪崩的效应，Hystrix就是能阻止服务雪崩的利器。他能感知一个请求的请求时间，不论如何配置了Ribbon负载均衡，为了阻止服务雪崩，Hystrix可以让超时的请求迅速退出并进行降维处理，这样可以非常友好的给出服务超时的回应与用户体验。

对于HystrixDashborad来说，Hystrix可以量化的展示出服务的压力。为了开发的便利性，把HystrixDashborad对Gateway网关进行了监控，因为Gateway是流量压力的总入口，总入口的压力大就表明该项目的压力大，因为当前项目中为了节省开发的成本，服务提供者只有一个。

### 2.5.6配置文件动态配置

图标：暂无

在SpringBoot开发中，某些配置文件我们希望可以动态的去配置并且理解生效，比如项目中的Log日志级别... ...

而不希望由于修改了某些配置而需要重新启动项目。这个时候可以考虑SpringCloud提供的SpringCloud-Config技术。

该技术的使用简述如下：开启一个针对Github的服务，动态的感知保存在Github中的配置文件，然后推送给需要它的服务模块并动态加载配置。

### 2.5.7 服务RPC与Http

图标：暂无

在分布式的架构中，不可避免的出现服务和服务不在同一个项目中，甚至不在同一个服务器中，这就会导致服务和服务之间的交互将会变得困难。

SpringCloud封装了FeignClient，其定位就是为了替代Java原始的HttpUrlConnection和HttpClient，因为Java原生的方式代码量非常的多且不易维护，在IO流的处理上还需要开发者有一定的经验。而FeignClient使得不同服务之间的调用就像调用接口一样的方便。

与此对应的是Apache的Dubbo（以前由Alibaba进行维护，最后成为Apache的顶级项目）。Dubbo也是基于接口，但他需要有Zookeeper的服务注册中心。他的特点是调用方像使用本项目中的类一样的去使用远程服务。

Dubbo和FeignClient最大的区别，前者是基于RPC调度，FeignClient是基于Http方式调度。

为了节省开发复杂和控制成本，项目不会再去假设Zookeeper，因此直接采用FeignClient开发。

### 2.5.8 定时任务框架

图标：



**图2-8 Quartz图标**

项目中存在定时邮件告警，目前暂定于每天的整点时间进行将数据库记录的ERROR级别的日志进行发送到指定的邮箱。服务中除了需要有邮件发送的功能外，还需要有定时任务调度框架，并且该框架可以支持在线CORN表达式的修改，以此来动态的修改定时任务。

考虑到复杂性，项目今后的升级会考虑将Quartz更换成大众点评的XXL-JOB，因为其能够分布式的任务执行并不会有数据库的侵入性。

## 2.6 项目难点

项目因为是微服务的架构，因为微服务的世界里，每个服务都可以有其独立的数据库。那么就需要考虑数据库数据的一致性，这点是非常复杂的。

因为项目业务的功能性并不多且并不复杂，因此使用线上的一个数据库就可以完美的规避数据一致性的问题。

微服务的项目必然是多模块化，因为所有的服务模块都是基于SpringBoot开发，那么在整合的时候可能出现SpringBoot配置文件冲突或者未被加载，这一点随着功能的增加会变得尤为的复杂。需要开发者对每个配置文件的熟悉，可以在每个配置文件中添加响应的注释以往忘记。

## 2.7 技术官网文档汇总

1.SpringBoot：https://spring.io/projects/spring-boot

2.SpringCloud：https://spring.io/projects/spring-cloud

3.Gateway：https://spring.io/projects/spring-cloud-gateway

4.Kafka：http://kafka.apache.org/

5.Hystrix：https://github.com/Netflix/Hystrix

6.MySQL：https://www.mysql.com/

7.Quartz：http://www.quartz-scheduler.org/

## 2.8 硬、软件开发环境

1.JDK版本：1.8.0\_131

2.WebStorm版本：2018.2.4

3.IntelliJ IDEA版本：2018.2.4

4.OS版本：Windows 10 64Bit

5.MySQL版本：5.5

6.Redis版本：2.8

7.NodeJs版本：v10.16.0

8.VueJs版本：v2.9.6

## 2.9 小结

本章节中，从项目语言选型开始，阐述了项目后端采用Java语言，项目前端使用微信小程序以及VueJs的背景及原因。后来分析了软件行业中架构演变的过程，分析了以往单一架构的缺点，以及现在微服务架构的优点及缺点。之后分析了项目中所使用的的新技术，逐一对新技术的介绍以及使用场景。

**第3章 概要设计**

## 3.1 需求分析

从宏观来看，项目分为一个后端项目，两个前端项目，两个前端分别为：小程序端和PC运营端。

小程序端：

①需要对用户的健康状况进行收集，在收集的过程中需要使用BMI等身体指数计算公式进行对用户身体状况进行计算；

②根据小程序使用者填写的身体参数并根据中国饮食膳食标准提供定制化的饮食推荐；

③用户可以实时的填写今天已经完成了哪些定制化推荐的任务，并填写实时的百分比；

④系统柜根据用户填写的定制化需求的百分比进行实时的计算当天各个任务完成的任务量占据总任务量的百分比，为了更加直观的展示给用户，这些数据都将使用百度Echarts产品进行数据展示，方便用户直观的对比每天完成任务量的趋势图以及每一项的完成占比；

⑤平台提供“悄悄话”功能，打破平台用户与用户之间交际壁垒并，在此功能上在不违法的情况下可以进行畅所欲言，释放压力或者分享收获的喜悦亦或者诉说内心的委屈，用户也可对其进行评论或私信，私信将以站内消息的形式发送给文章编写者。

运维PC端：

①运维端需要对项目中饮食种类以及定制化进行数据添加、删除、修改、查询；

②运维界面需要对接微信厂商获取用户访问趋势图以及页面访问量等，方便运维者进行及时调整项目功能；

③运维平台需要展示页面SQL执行情况、执行时间分布、由哪些URL触发的SQL、SQL执行黑名单、白名单展示，以及SQL执行的时间，这些数据可以非常方便告知开发者，哪些SQL需要进行优化；

④运维平台需要展示服务压力，并且以图表形式直观展示，这样可以方便告知运维者需要将哪些节点弹性扩容以释放并发压力；

⑤平台运营者需要编写关键健康类的文章以提供小程序首页的支持

后端：

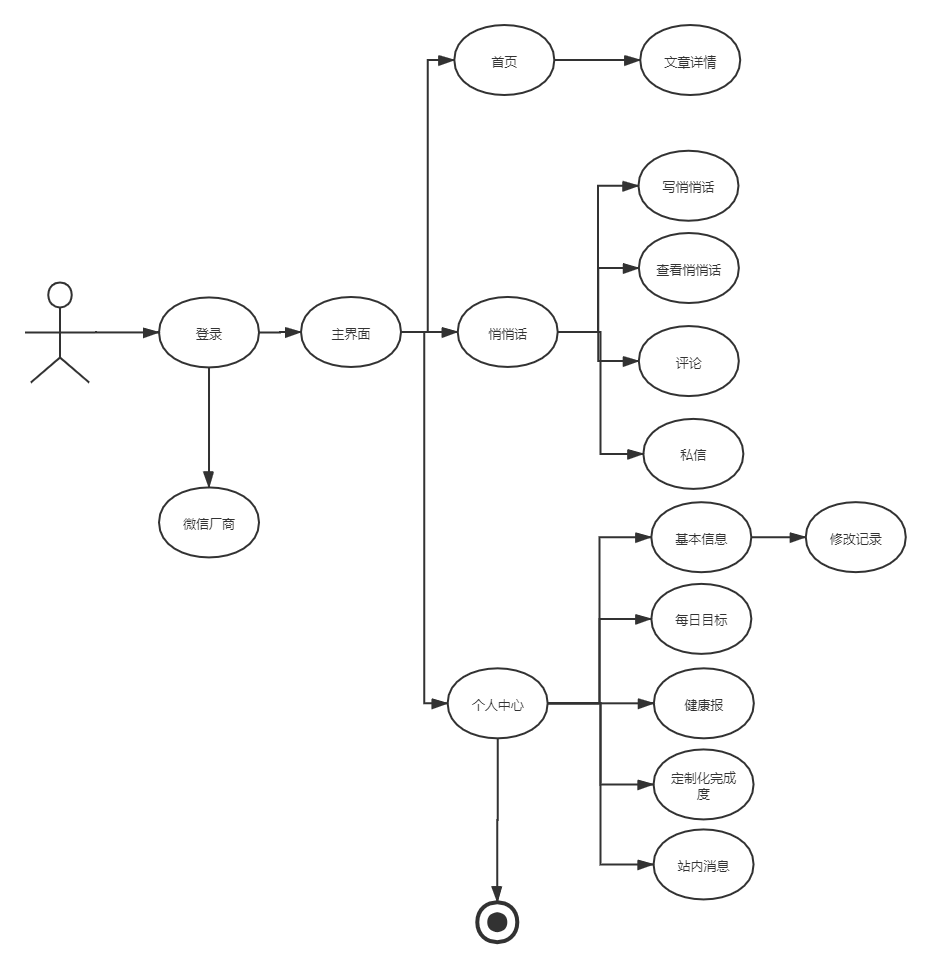
①为小程序端和PC端提供接口支持；

②支持邮件发送功能并整合Quartz进行项目错误日志发送邮箱进行错误邮箱告警，并且日志错误级别是可配置的。

## 3.2 用例分析

### 3.2.1 小程序端用例分析

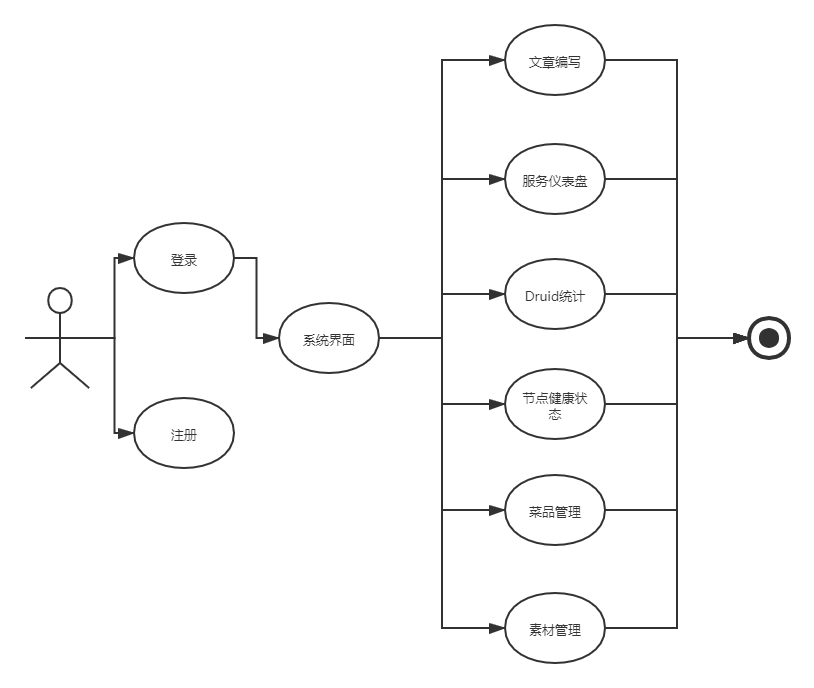
从小程序使用者出发，用户使用微信小程序进行免密登录，登录进去之后小程序大体分为三个模块，分别是：“首页”、“悄悄话”、“个人中心”三大模块。首页对接运营者写的健康类别的文章；悄悄话模块，用户可看到别人说的“悄悄话”，可以对其追加评论或者私信作者引发共鸣；“个人中心”中可以查看个人编写记录个人的身体参数系统会生成每日目标，用户填写每日定制化任务的完成度，系统以此来进行生成健康日报。如下图，为系统用例图：



**图3-1 小程序端用例图**

### 3.2.2运营PC端

运营这登录运营系统之后，进入主界面，主界面左侧有很多一级菜单而每个一级菜单都表示一个功能点，分别有“文章编写”、“服务仪表盘”、“节点健康状态”、“Druid统计”、“菜品管理”。如下图，为运维PC端用例图：

****

**图3-2 运营PC端用例图**

## 3.3 数据库设计

概要：在数据库设计中，根据企业实习的经验来看，数据库设计使用Innodb数据库引擎，通过锁的机制来保证数据库事务，进而保证数据的可靠性和一致性。

主外键关联：在数据库的数据一致性中而言，数据的主外键的存在也在一定的程度上保证了数据的可靠性和正确性。但这在一定的程度上添加了开发的维护难度，在开发初期，“子”数据的删除必须要保证“父”数据已经删除，很显然在开发过程中带来了开发成本，在每一次的数据删除之前都需要对数据进行是否“可删除”的逻辑判断。

索引方面：数据库中的索引，可以大大的提高数据查询的能力。以MySQL关系型数据库而言，索引分为：普通索引、唯一索引、主键索引和聚集索引。项目对于每一张表而言，都使用了主键索引并且根据整形递增的方式进行保存索引。

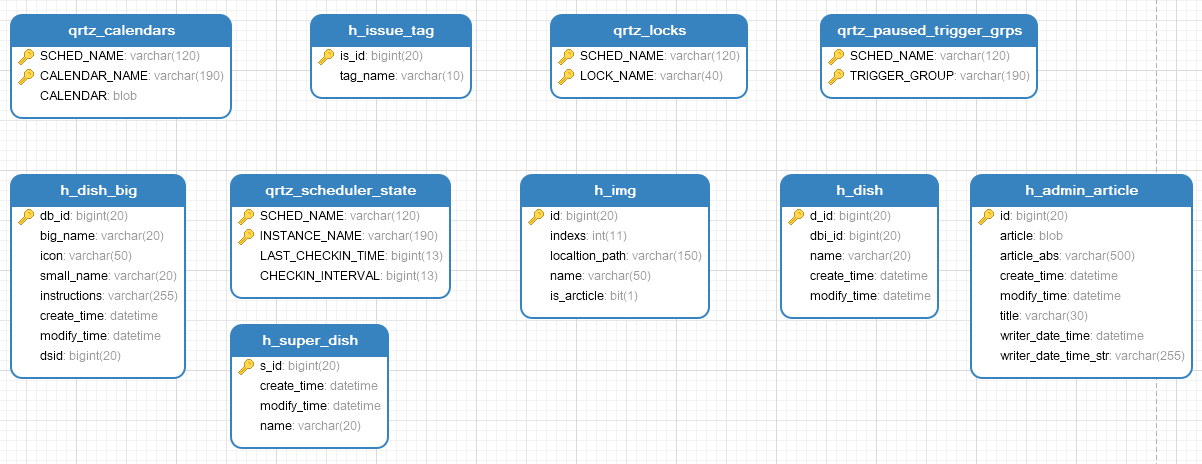
表数目：数据库中存在表数为26张表，其中11张表为定时任务相关表，剩下的15张表为业务相关表。

多表查询：在数据库中多表关联查询的方式有很多种，项目中多采用左外连接的方式（Left Join）。

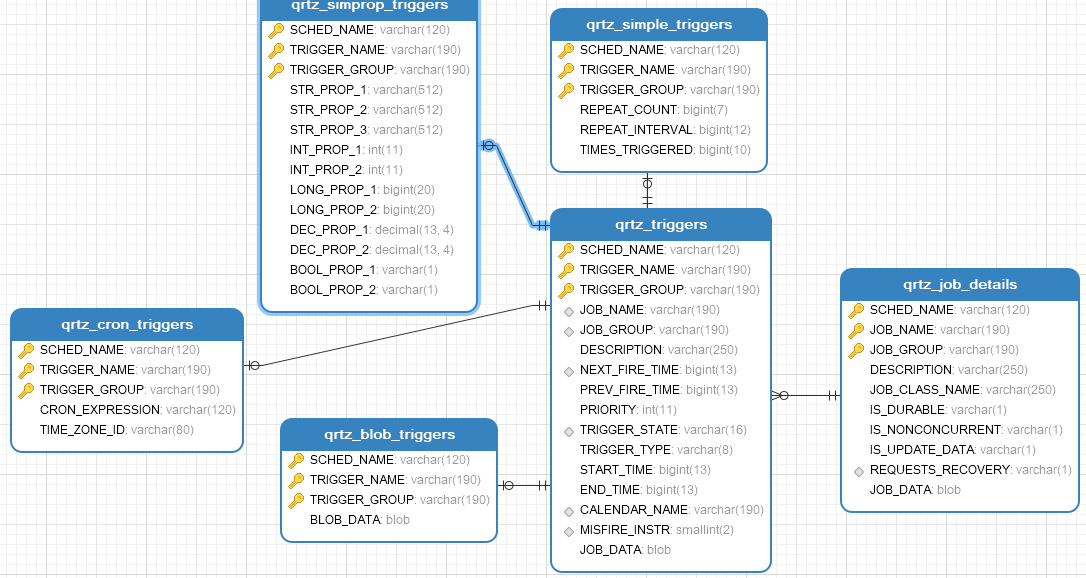
以下为项目数据库ER图，如下图：



**图3-3 ER图**



**图3-4 ER图**



**图3-5 ER图**

## 3.4 界面设计

### 3.4.1 小程序界面设计

主题色十六进制：#2d355b，蓝色系；

图标：图标采用Iconfont的多彩扁平化图标；

辅助性文字颜色为#657180；

### 3.4.2 网页PC端

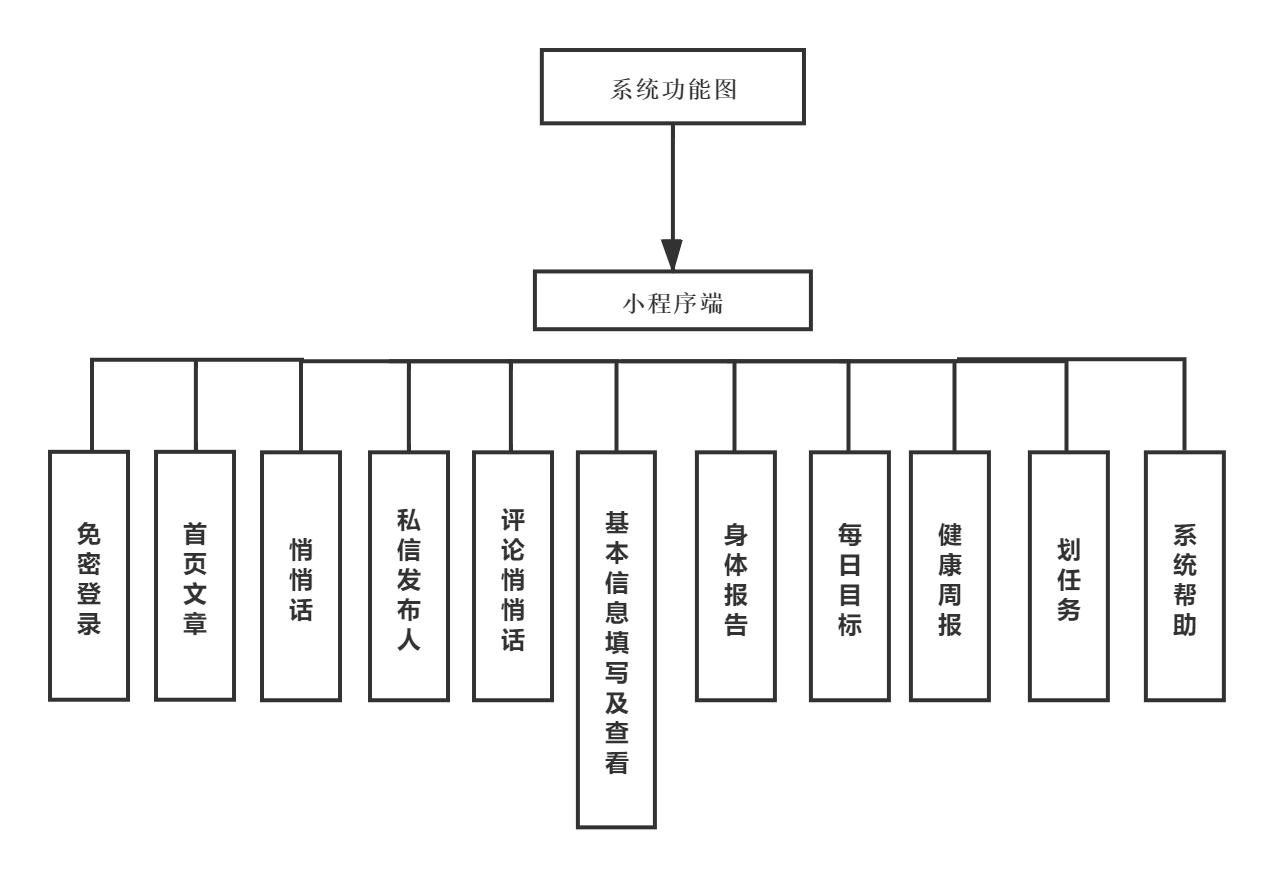
主题色十六进制：#515a6e；

网页控件背景色：#2b85e4（中性主题色）、#ff9900（警告主题色）、#ed4014（错误主题色）、#808695（辅助性文字）；

## 3.5 总体功能模块

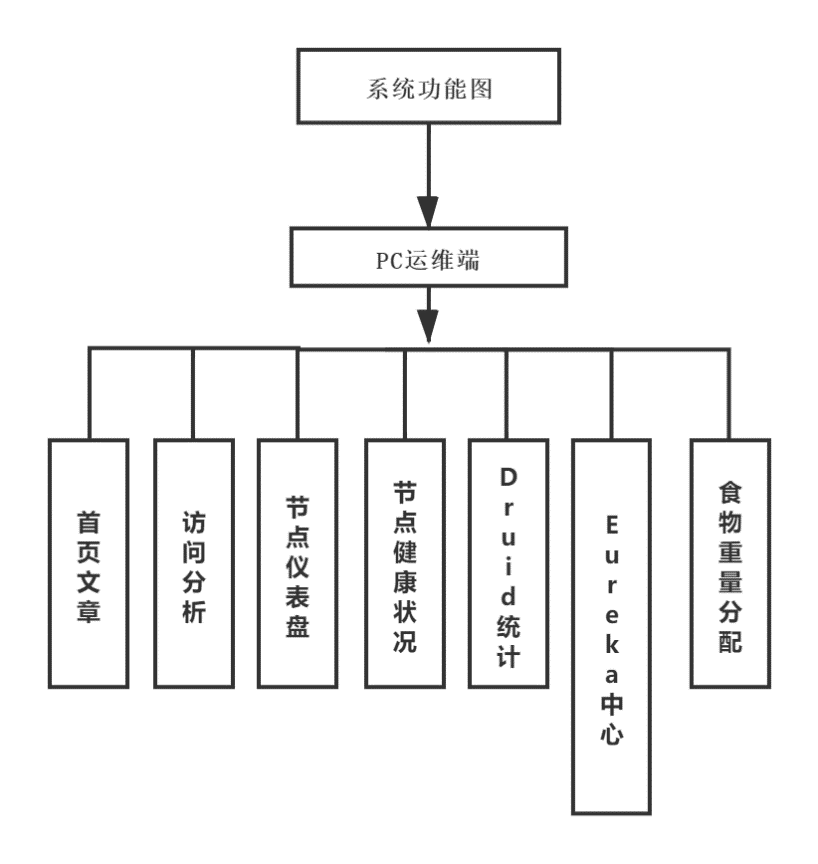
简述：在本项目中分为三个开发项目，分别为微信小程序端、运维PC端，Java后端，其中前两者统称为项目前端，最后者称为项目后端。由于图片排版问题，将这三者的功能图分开说明：

小程序端总体功能模块，小程序端主要给普通用户所使用，如下图：

****

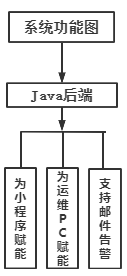
**图3-6 小程序端总体功能模块图**

运维PC端总体功能模块，主要给运维者使用，如下图：

****

**图3-7 运维PC端总体功能模块图**

Java后端功能，主要为整个项目提供技术与接口支持，如下图：



**图3-8 Java 后台总体功能模块图**

## 3.6 小结

本章节开头部分分析了需求，从需求入手进行拆分出一个个功能模块。之后根据需求和功能画出小程序端和PC端的用例分析图。功能性需求分析完毕之后进行对数据库表的设计，以及前端的UI设计。本章节最后给出小程序、运营PC端、Java后端的总体功能图。

**第4章 详细设计**

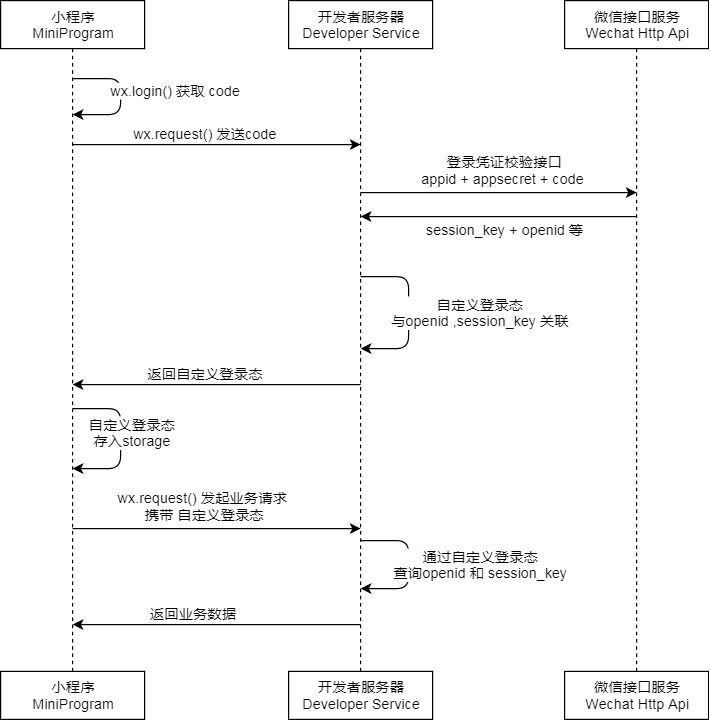
## 4.1 简述

本章节为项目的详细设计，由于项目角色多，功能较多，由于篇幅的问题，本章节将会选取重点功能和核心功能进行从需求分析，思路分析、生命周期图方法、核心代码说明等方面进行说明。

## 4.2 小程序免密登录

简述：项目使用微信小程序作为用户的使用平台是出于微信庞大群体生态圈的考虑，既然使用了微信小程序，那么用户免密登录而使用微信自然的登录接口是必然的选择。因为微信小程序的登录逻辑较为复杂，因此有必要在“系统详细设计”中进行详细说明：

微信官网给出了对接微信厂商进行用户免密登录的生命周期图。如下图：



**图4-1 Java 后台总体功能模块图**

文字描述如下：

1. 小程序端触发wx.login()获取一次性登录凭证Code，并传递到后端服务中；

②开发者服务携带收到的Code请求微信厂商的auth,code2Session进行换取该用户的OpenId和SessionKey，需要注意的是SessionKey的存活时间并不是一成不变；

1. 开发者服务器根据获取的OpendId、SessionKey自行组织系统凭证；

注：在微信登录流程中有必要对SessionKey和OpenId进行了解，

SessionKey：微信官方开发文档给出了SessionKey以及其生命周期的详细解释，大概意思为：用户SessionKey的过期时间对于开发者而言完全透明，某个用户在某次使用中，使用的越频繁SessionKey的生命周期就会越长。

OpenId：OpenId是一个用户对一个微信小程序的唯一认证，简单而言类似于该用户在该小程序用户表中的主键ID。

了解了登录流程以及必要字段的说明，下面对上述步骤二进行详细说明，因为步骤二直接关系系统合法性用户的判断方案。

①平台不会直接使用OpenId作为用户的主键，引用微信官方文档是：OpenId、SessionKey都是较为隐私的数据；

②当用户发起登录流程时，首先经过项目的路由网关（网关的白名单决定不对登录接口进行认证检查）并放行到服务提供者，服务提供者会拿着用户传递来的Code请求去请求微信厂商并获得OpenId和SessionKey；

③维护OpenId->SessionKey的映射关系并保存在Redis缓存中；

④使用JWT技术，将OpenId和用户的ID放入JWT令牌中并返回小程序端，往后的请求只需在请求头Header中携带JWT令牌即可。后端收到请求之后会对用户令牌进行认证，认证通过即可进入服务，验证不通过返回登录页面；

⑤微信厂商的某些接口需要使用用户的SessionKey，若用到SessionKey，即根据OpenId去缓存中换取SessionKey即可。

小程序端核心代码如下：

wx.login({      success: res => {        request.sendHttp('POST', '/user/login', {          code: res.code        }).then(res => {          let result = res.data.result;          if (!result.users) {             this.setData({              btn\_disable: false            })          } else {            app.globalData.userInfo = result.users;            util.go.switchTab('/pages/index/index', null);          }          app.globalData.token = result.jwt;        }).catch(err => {          console.log(err);        });      },      fail: function(res) {},      complete: function(res) {        wx.hideLoading();      }    })

后端核心代码

public Map<String, Object> login(Map<String, String> map) {

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

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

String response = auth.getOpenIdAndSessionKey(appId, secret, map.get("code"), "authorization\_code");

try {

map = objectMapper.readValue(response, Map.class);

Users usersFromData = userDao.findByOpenId(String.valueOf(map.get("openid")));

if (usersFromData != null) {

jwtMap.put("uId", usersFromData.getUId());

result.put("users", usersFromData.createUserVO());

redisUtils.putSessionKey(map.get("openid"), map.get("session\_key"));

}

jwtMap.put("openId", map.get("openid"));

result.put("jwt", genJWTUTil.creatToken(jwtMap));

} catch (JsonProcessingException e) {

e.printStackTrace();

} catch (JOSEException e) {

e.printStackTrace();

}

return result;

}

## 4.3 小程序请求封装

简述：在小程序开发中，小程序提供请求API（wx.request）进行前后端分离异步请求，而本项目就是采用原生小程序的请求方式。

在开发的过程中常常需要进行请求代码的编写用于调用后端提供的API接口，但这需要编写大量的重复性代码导致开发效率低下，因此说，在前端开发过程中，首先进行请求代码的封装是关键的步骤，这直接影响着前端开发效率。

本项目中使用使用wx.request和Promise异步方案进行封装，对Content-type和请求数据也进行封装。

前端核心代码如下：

function sendHttp(method, url, data, header = {

"Content-Type": "application/json",

"Token": app.globalData.token

}) {

return new Promise((resolve, reject) => {

wx.request({

url: server\_api + url,

method: method,

data: data,

header: header,

success: res => {

if (res.data.code === 1) {

resolve(res);

} else {

util.toast.showToast(res.data.result, 'none');

resolve(res);

}

},

fail: err => {

console.log(err);

wx.showToast({

title: 'NETERROR',

icon: 'loading',

duration: 2000

})

reject(err);

}

}

对上述核心代码进行说明：

使用了ES6函数参数默认值的写法：

在ES6中支持了函数参数默认值填充，它存在的最大意义在于，对于一些不太容易变化的参数，在函数参数状态就进行初始化赋值，若调用者没有对其数据进行传参填充那么就使用默认值，若调用者在调用的同时对参数进行赋值，那么调用者的参数值将会覆盖方法的默认值，从而方便开发，一般而已，该技术点常常用于某个参数几乎固定不变的情况下。

使用了ES6的Promise的语法糖：

Promise的“诞生”直接影响了整个前端开发人员，在早期，前后端分离的异步请求需要采用JS构建出XMLHttpRequest对象进行数据的填充，但这开发起来代码量非常大，后来Ajax的出现解决了开发复杂性和代码膨胀的问题。但Ajax有他致命的弱点：“回调炼狱”（回调简单理解为提供方法给对方调用）。回调中代码出现大量的垂直结构，如下为简单的伪代码体现“回调炼狱”：res->{res->{res操作代码……}}。而Promise的出现会将回调变成扁平状态，于是上面的“回调炼狱”伪代码近乎斜撑：res->{}.res->{}.res->{}……

为了解决“回调炼狱”的问题，项目还采用了Async、Await的方式。

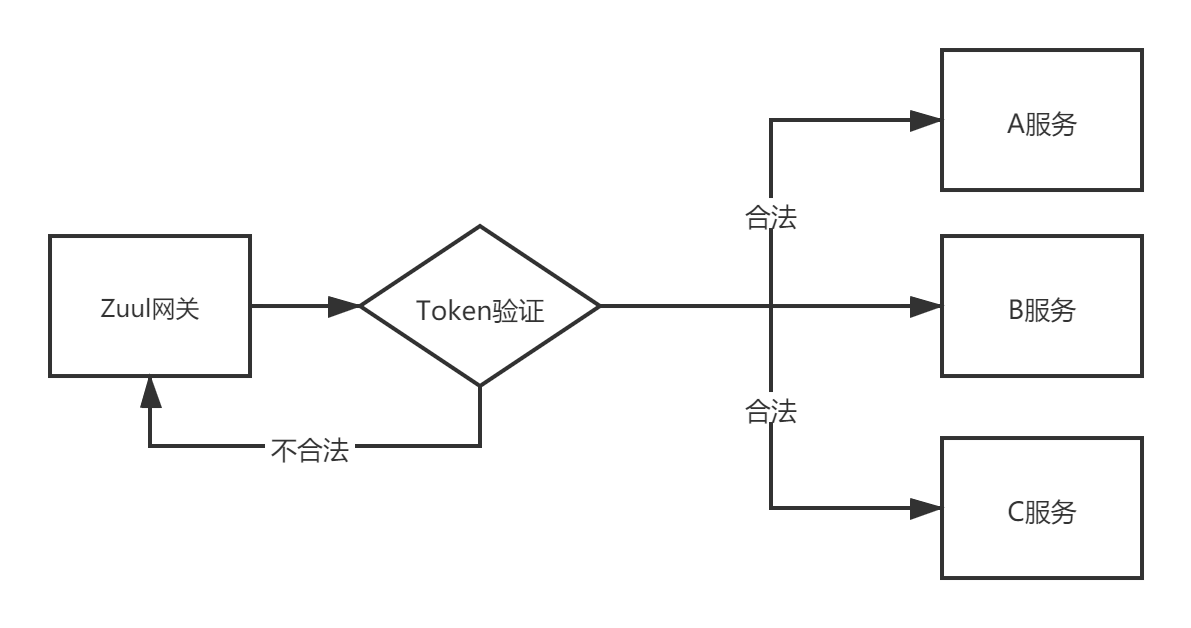
## 4.4 Gateway流量鉴权

简述：项目中对于流量鉴权非常的重要，哪些流量是正常的，哪些流量是异常的，系统需要进行严格的判断，进而减少服务器的DDS攻击，从而浪费服务器资源。

由于项目为微服务架构思想，因此后期的多个服务模块若每个模块都进行鉴权操作，那么代码会非常的冗余，且在架构中给路由也代码一定的困难，因此项目使用Zuul网关进行统一流量入口，并对用户请求进行统一鉴权。

总体思路：构建网关服务，所有的请求统一请求到Zuul网关，网关的前置过滤器（Pre）拦截请求获取RequestUrl、Header、Token，网关配置白名单（登录、注册、访问图片资源等相关路径）。若RequestUrl根据匹配原则匹配到了白名单，那么对该请求直接放行。若RequestUrl不属于白名单范围，就会验证请求头中的Token信息，Token信息有JWT进行构造，若JWT合法则进行路由，若不合法则退出。

总体设计如下：



**图4-2 Zuul验证**

生成Token核心代码如下：

public String creatToken(Map<String, Object> payloadMap) throws JOSEException {

JWSHeader jwsHeader = new JWSHeader(JWSAlgorithm.HS256);

Payload payload = new Payload(new JSONObject(payloadMap));

JWSObject jwsObject = new JWSObject(jwsHeader, payload);

JWSSigner jwsSigner = new MACSigner(jwtSecret.getBytes());

jwsObject.sign(jwsSigner);

return jwsObject.serialize();

}

核心代码说明：

在Jwt中由三个部分组成，分别是头部，载荷，签名。其中头部采用了HS256的加密算法，载荷为放在JWT中的自定义信息，尾部为头部喝载荷的压缩签名，尾部用于验证JWT是否有限是否合法是否被窜改。

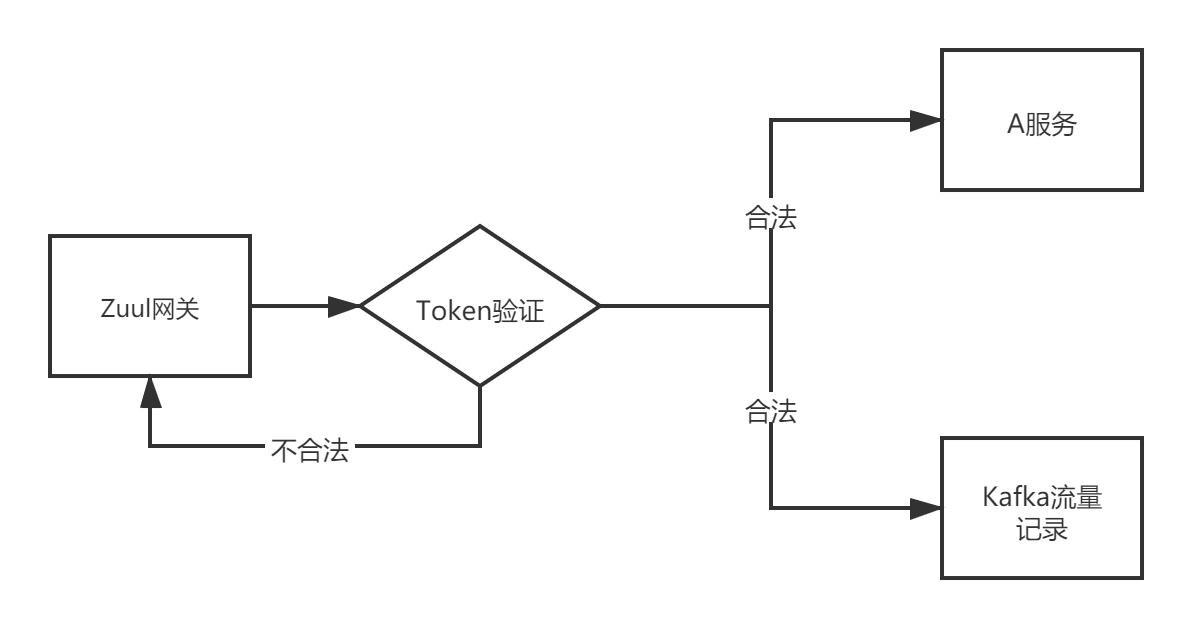
## 4.5 Kafka记录流量

简述：Kafka为流式数据的短暂存储（Kafka的数据保留一般设置为7天）。Kafka一般用于存储日志信息或者流量记录统计，这些数据并不适合持久化在数据库中，而是暂时保存的一个地方，若后续需要可以对其进行数据消费。而这就是消息队列技术。

在本项目中消息队列用于存储系统流量。即在用户访问之后将用户的ID信息，访问URL等敏感信息进行记录。以防知道项目每日PV量以及访问者信息等。

为了统一进行记录，Kakfa用于Zuul Gateway处，因为该模块是访问流量总入口。

总体设计如下：



**图4-3 流量记录**

核心代码如下：

public void sendToTopic(String topic, String jsonMsg) {

kafkaTemplate.send(Contanst.\_FLOWTOPIC, jsonMsg);

ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, jsonMsg);

future.addCallback(success -> {

String sendTopicName = success.getProducerRecord().topic();

Integer partion = success.getProducerRecord().partition();

Long offset = success.getRecordMetadata().offset();

log.info("\n Success:\nTopName：{}、Partition：{}、Offset：{}", sendTopicName, partion, offset);

},

fail -> log.error("\n ERROR \n {}", fail.getMessage()));

}

核心代码说明：

为了提供向消息队列中发送消息，项目中发送方法采用默认的异步方式进行消息发送。

但平台中想将发送信息是否成功以及发送完毕之后消息在消息队列中的位移（Offset）的显示，因此编写了发送消息的回调。在发送消息的成功回调中进行消息的位移记录等。

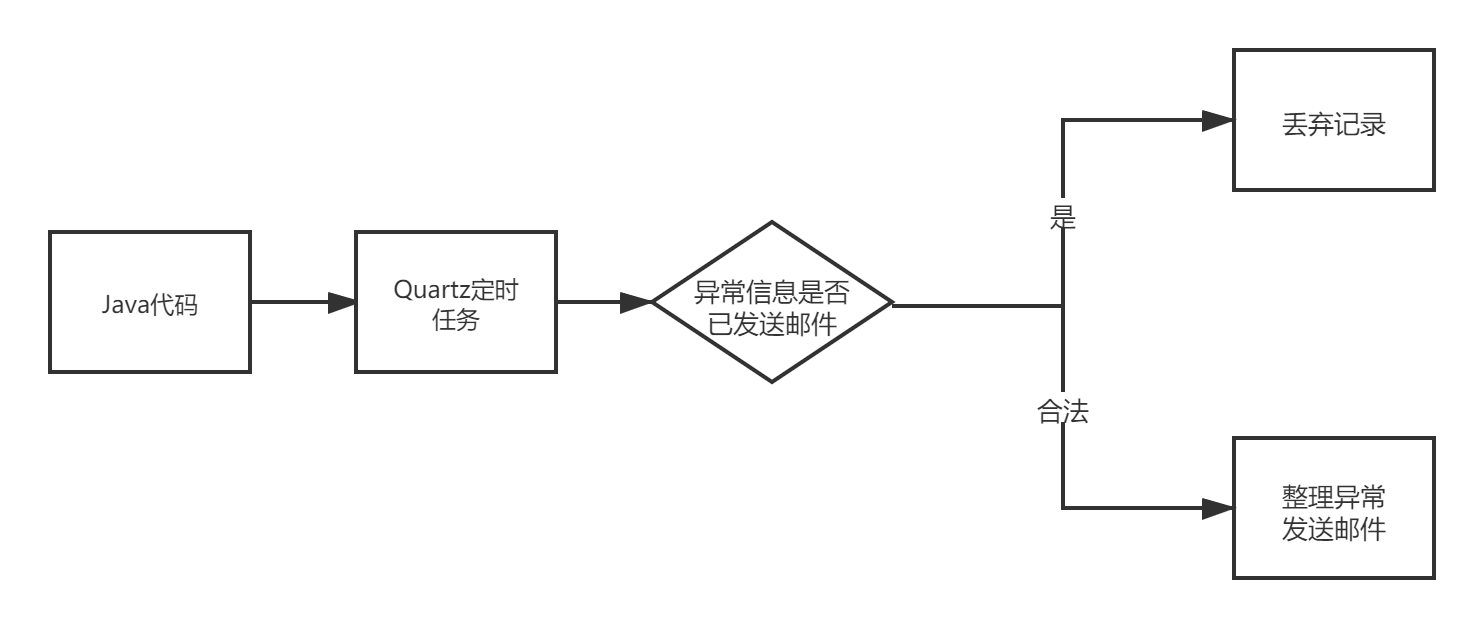
## 4.6 邮件告警

简述：在项目中邮件告警尤为的重要。尤其是互联网相关的软件，除非出现Error级别的大问题导致项目运行崩溃，否则开发人员一般无法感知到线上产品近期运行是否健康的问题，除非开发人员去翻看项目日志，查看是否有Error级别的错误日志来判断，这种通过“人肉”查看日志的情形即费时费力还非常的低效。

因此项目再小程序端或者是后端都采用了异常统一处理的机制，当小程序端发现错误，将会统一收集并将错误信息发送给后端，由后端记录数据库，而对于Java后端，采用了AOP的原理进行异常捕获并记录在数据库中。

数据库中存在一个是否已经发送邮件的字段，项目中集成了Quartz定时调度框架，该框架通过设定的CRON表达式进行任务触发，而在一个任务中会收集数据库中的异常且未被发送过邮件的错误信息并整合起来，通过邮件的形式发送给内置设定好的开发人员。在邮件中设置了错误信息回调的功能，当开发人员收到邮件之后可点击异常详情则可进行错误信息详细查看。

总体设计如下：



**图4-4 邮件告警**

核心代码如下：

private void appendSegment(StringBuffer stringBuffer, boolean isAHref, Object data, Long dataId) {

stringBuffer.append("<td style=\"border:solid #add9c0; border-width:0px 1px 1px 0px;\">");

if (isAHref) {

stringBuffer.append("<span><a target='\_blank' href='" + callBackServer + "/log/by/id/" + dataId + "'>" + data + "</span></a>");

} else {

stringBuffer.append("<span>" + (data == null ? "暂无" : data) + "</span>");

}

stringBuffer.append("</td>");

}

上述代码说明：在上述代码中主要用来构建HTML Segment，因为在本项目中所有的邮件告警采用HTML的方式，这样可以更加美观和直观的查看Error信息。

核心代码：

private boolean sendMail(StringBuffer htmlMessage) {

if (StringUtils.isEmpty(htmlMessage)) {

return false;

}

MimeMessage mimeMailMessage = javaMailSender.createMimeMessage();

try {

MimeMessageHelper mimeMessageHelper = new MimeMessageHelper(mimeMailMessage, true);

mimeMessageHelper.setFrom(sendMail);

mimeMessageHelper.setTo(sendMail);

mimeMessageHelper.setSubject("Healthy平台日志运维，日志级别 > " + emailLogLeavel);

mimeMessageHelper.setText(htmlMessage.toString(), true);

javaMailSender.send(mimeMailMessage);

return true;

} catch (MessagingException e) {

e.printStackTrace();

return false;

}

}

核心代码说明：

上述代码是将组织好的HTML Segment进行邮件发送功能。上述代码核心类为：MimeMessage，该类有所使用的SpringBoot邮件功能提供。

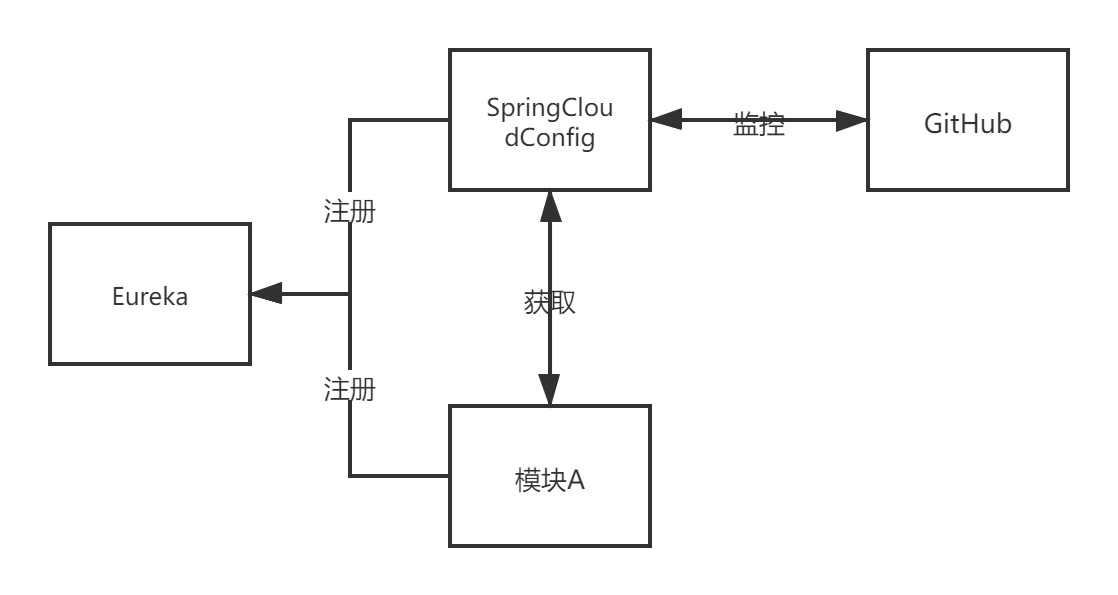
## 4.7 SpringBoot统一配置

简述：在多服务模块中，所有的SpringBoot单个节点服务的启动都需要依赖自身定制化的项目配置文件。在开发过程中，经常需要该服务模块的功能而去定制化的配置相关文件。但如果很多的模块以及打包部署发布在了公网上，再需要修改配置，然后停止项目运行重新启动，那会带来太多的麻烦且带来了很不好的用户体验。

为了能够在线修改配置文件，且让项目自动的感知配置文件的变化，最终要的是不需要重新运行项目是一个必须的需求。

因此项目采用SpringCloudConfig组件进行统一配置。

总体设计如下：



**图4-5 统一配置**

核心配置：

spring:

application:

name: config-single-server

cloud:

config:

server:

git:

uri: https://github.com/xxx/config

username: xxx

password: xxx

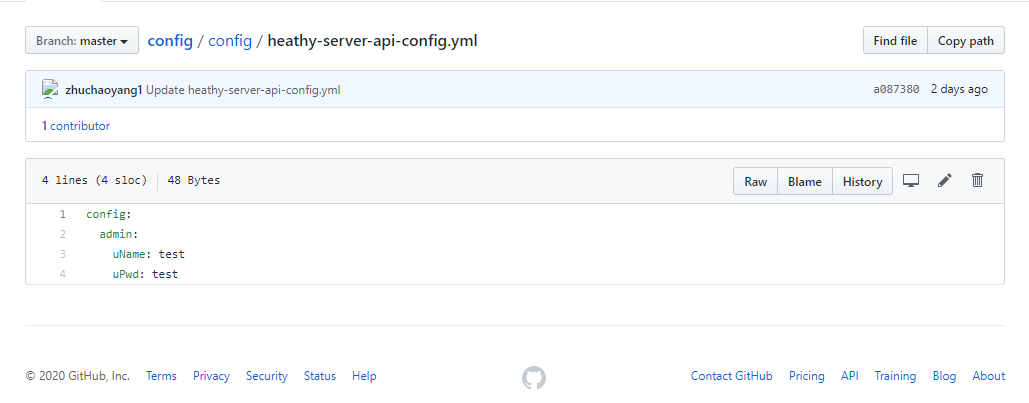
default-label: master

search-paths: config

核心配置说明：

在SpringCloudConfig模块中，需要配置Github地址，并填写远程仓库地址以及仓库分支。

仓库配置文件截图如下：



**图4-6 GitHub截图**

需要使用动态配置项目的配置：

spring:

cloud:

config:

name: heathy-server-api # 如果不配置name则默认为当前服务API，则Github config：API-xxx

label: master

profile: config

discovery:

enabled: true # 开启Eureka路由

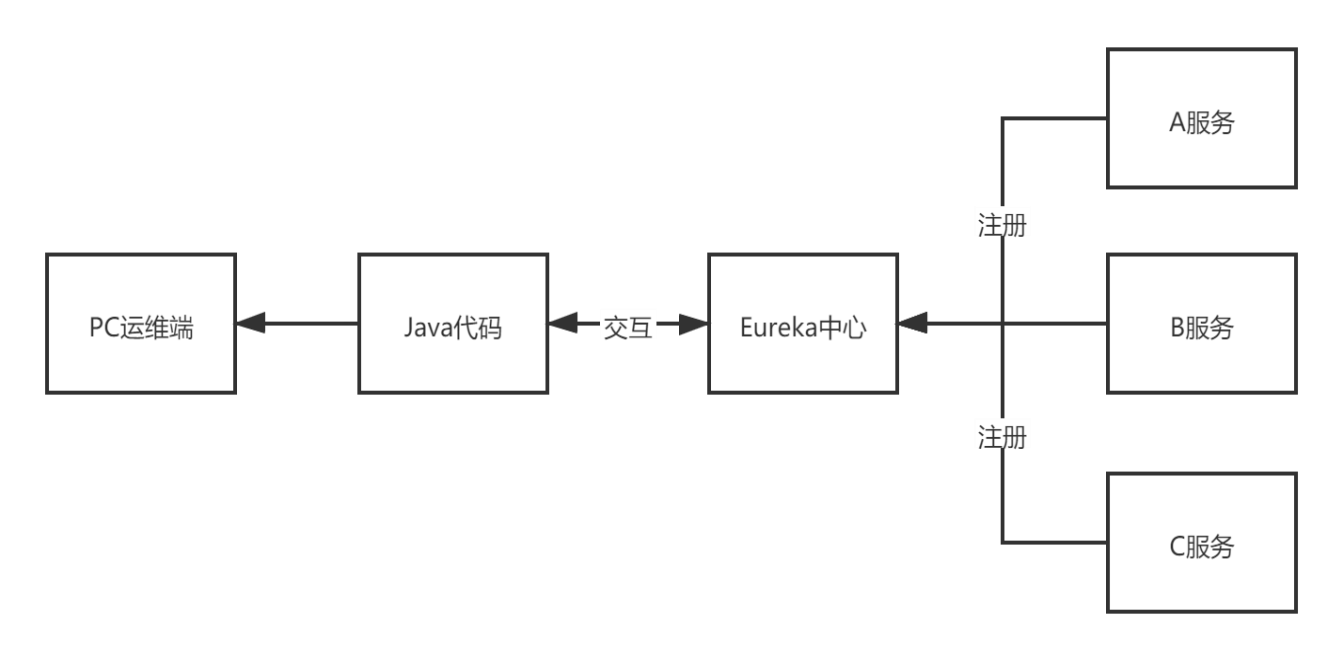
service-id: config-single-server # config-server Application ID

上述代码需要配置SpringCloudConfig的服务模块，上述配置中尤其需要注意name的属性值，name的属性值加上下述的Profile为Github的配置文件名称。

## 4.8 节点健康状态

简述：在项目运维过程中，开发人员实时的查看项目各个节点的健康状况是非常重要的，本项目中采用Acutor进行节点健康检查，并采用注册中心客户端获取所有的节点信息以及对应的健康信息，最终展现在运维平台中。

总体设计如下：



**图4-7 健康状态**

核心代码：

@GetMapping("/get/application")

public ResultHttp getServicesList() {

List<ServiceInstance> servicesList = new ArrayList<>();

List<String> serviceNames = discoveryClient.getServices();

for (String serviceName : serviceNames) {// 可能一个appname对应多个服务做负载均衡

List<ServiceInstance> serviceInstances = discoveryClient.getInstances(serviceName);

servicesList.addAll(serviceInstances);

}

return ResultHttp.builder().code(1).result(servicesList).build();

}

核心代码说明：

通过Eureka客户端DiscoveryClient获取所有的Service，并迭代获得的Services，并按照名称进行获取服务实例。至于为什么每一个Service都需要再使用For循环迭代，这是因为存在一个ServiceName对应多个实例的情况，这种情况就是负载均衡。

这样项目可以非常方便的通过注册中心获取服务实例以及对应状态，从而避免开发服务器脚本的编写。

但是上述代码仍需要改进，因为Eureka有“自我保护”机制，即Eureka一段时间没有收到服务的心跳（可能已经宕机），那么这时候拿到服务节点的状态信息可能不是最新的，该问题可以关闭Eureka“自我保护”功能。

## 4.9 小结

本章节详细介绍了项目中的核心功能点。分别从功能点所使用的技术点、核心代码说明、功能流程图等方面进行详细阐述。其中主要对如下功能进行说明：小程序免密登录、小程序请求封装、Gateway流量鉴权、Kafka记录流量、邮件告警、SpringBoot统一配置、节点健康状态。

**第5章 项目测试**

## 5.1 简述

本论文中前四章分别从设计和实际开发维度进行说明，本章对项目的若干功能选取典型功能进行测试。主要测试在一些极端的使用情况下，项目是否会出现错误。项目所有的功能能否正确走通并不报错。

首先需要声明的是在本章中所有的小程序截图右下角都有一个绿色的Console按钮，该按钮为小程序真机调试的按钮并非项目中因业务需要而设置。

## 5.2 小程序免密登录测试

用户打开小程序即进行免密登录，如果用户为第一次使用，首页按钮会提醒用户授权登录，若该用户为老用户，那么首页的按钮就为灰化无法点击的状态而直接保持为登录状态。登录之后即可看见小程序的主界面，如下图5-1，登录之后进入主界面为图5-2。

**** ****

**图5-1 免密登录**   **图5-2 首页**

## 5.3 阅读后端发布文章

在本项目中，平台运维者可以通过运维界面定期发布健康或心理的好文章，发布之后，小程序端可以立即看到系统推送的“好文”，并进行文章阅览。图5-3为运维界面添加文章，图5-4为小程序用户查阅系统推送的文章。



**图5-3 运维界面添加文章**



**图5-4 文章阅读**

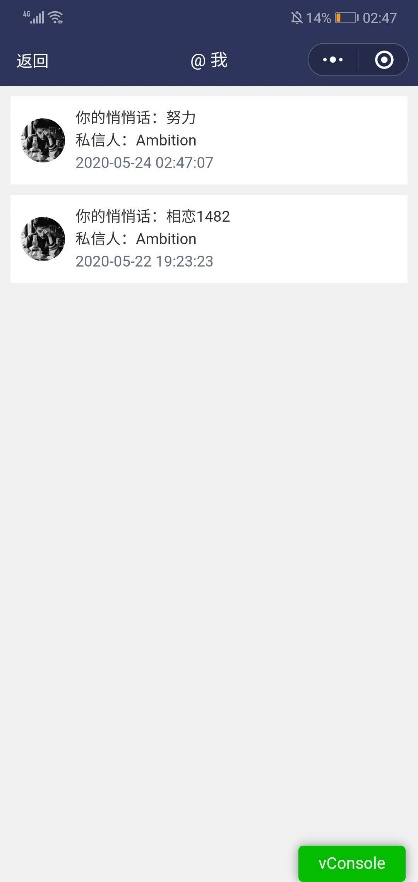
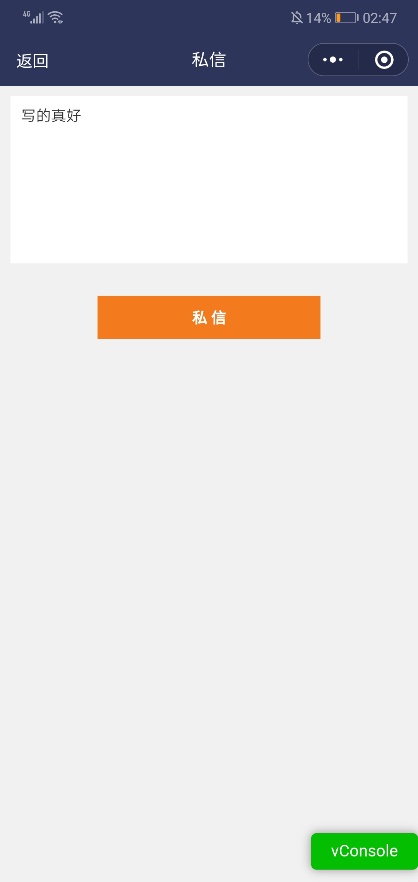
## 5.4 悄悄话模块

用户登录之后选择底部“悄悄话TabBar”，然后就可以进行悄悄话编写。悄悄话的种类很多，用户可以查看对应种类的悄悄话，且可以进行评论，亦可以进行编写，或者私信等相关功能。

下图5-5为编写悄悄话，图5-6为查看悄悄话列表，图5-7为查看悄悄话，图5-8为私信悄悄话，图5-9为查收私信消息，图5-10查看私信。

****

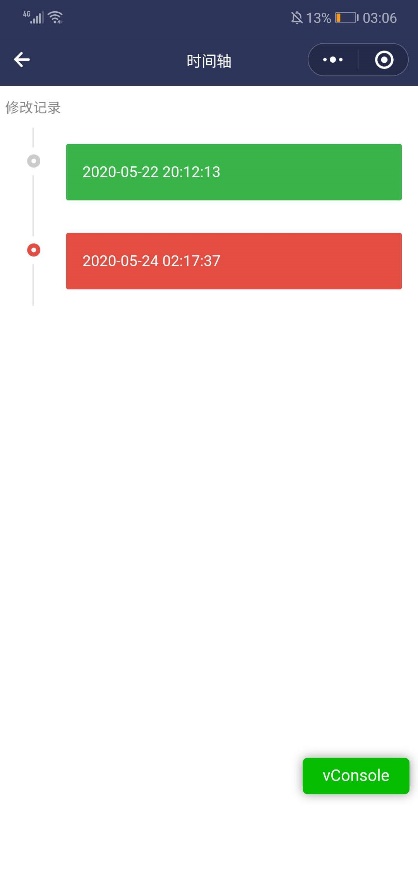
**图5-5 为编写悄悄话**  **图5-6 悄悄话列表**   **图5-7 查看悄悄话**

****

**图5-8 私信** **图5-9 查收私信消息**  **图5-10 查看私信**

## 5.5 基本信息

用户登录之后，在“我的”页面中找到“个人信息”，并可以按照不同的标准进行选择。在填写完身体报告之后就可以查看身体报告，身体报告分别从“标准体重”和“BMI”指数进行说明。除此之外还可以查看修改记录。下图5-11为信息参数填写，图5-12为身体报告查看，图5-13为修改记录查询。



**图5-11 信息参数填写**  **图5-12 身体报告查看**   **图5-13 修改记录查询**

## 5.6 每日目标

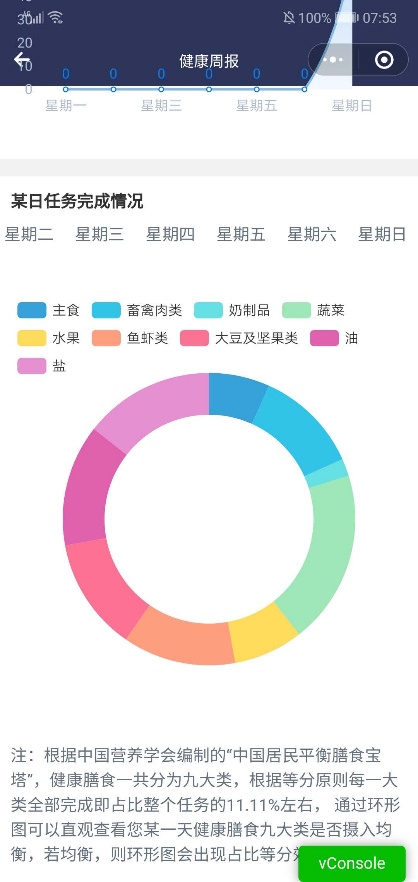
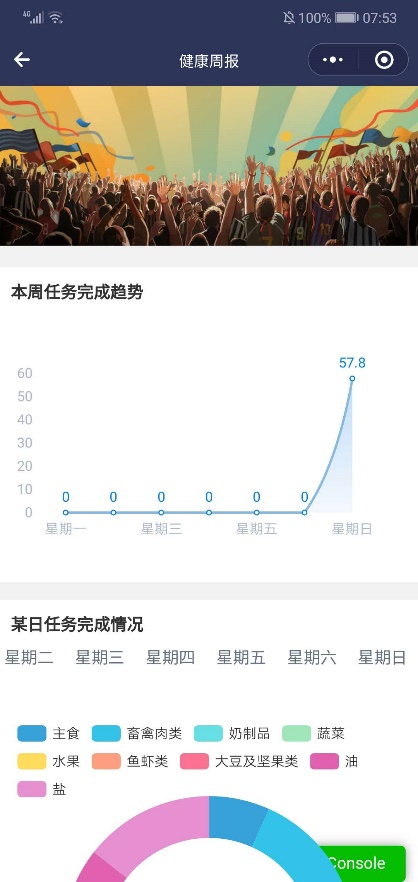
用户登录之后，打开“我的”，点击每日目标，系统会根据用户填写的最新身体参数定制化的推荐健康饮食，如下图5-14：

****

**图5-14 每日任务**

## 5.7 健康周报

用户每日填写任务进度之后，系统会根据用户所填写的进度进行百分比计算，分别以折线图和饼图的形式展示。折线图用于展示用户近几日完成任务百分比的对比。饼状图是某一天各项任务完成在总任务的占比，如下图5-15折线图，5-16饼图。



**图5-15 折线图**  **图5-16 饼状图**

## 5.8 划任务

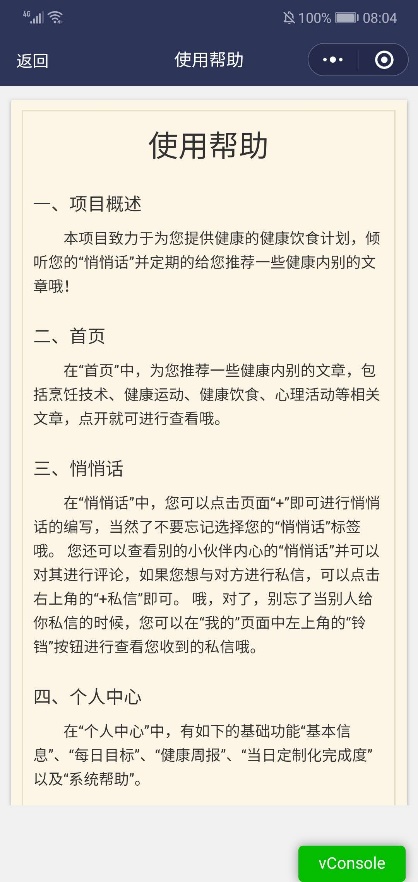
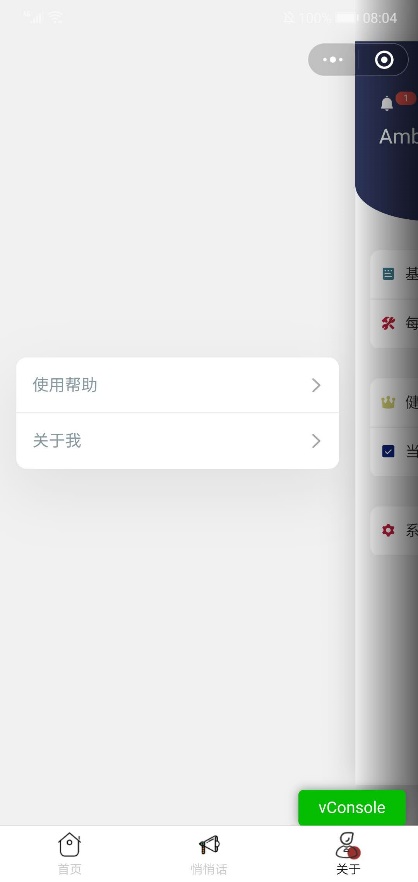
用户登录之后，在用户已经填写完身体参数信息之后，系统会每天为用户推荐近一阶段的健康饮食，用户每日可以对任务完成度进行反馈，如图5-17：



**图5-17 划任务**

## 5.9 系统帮助

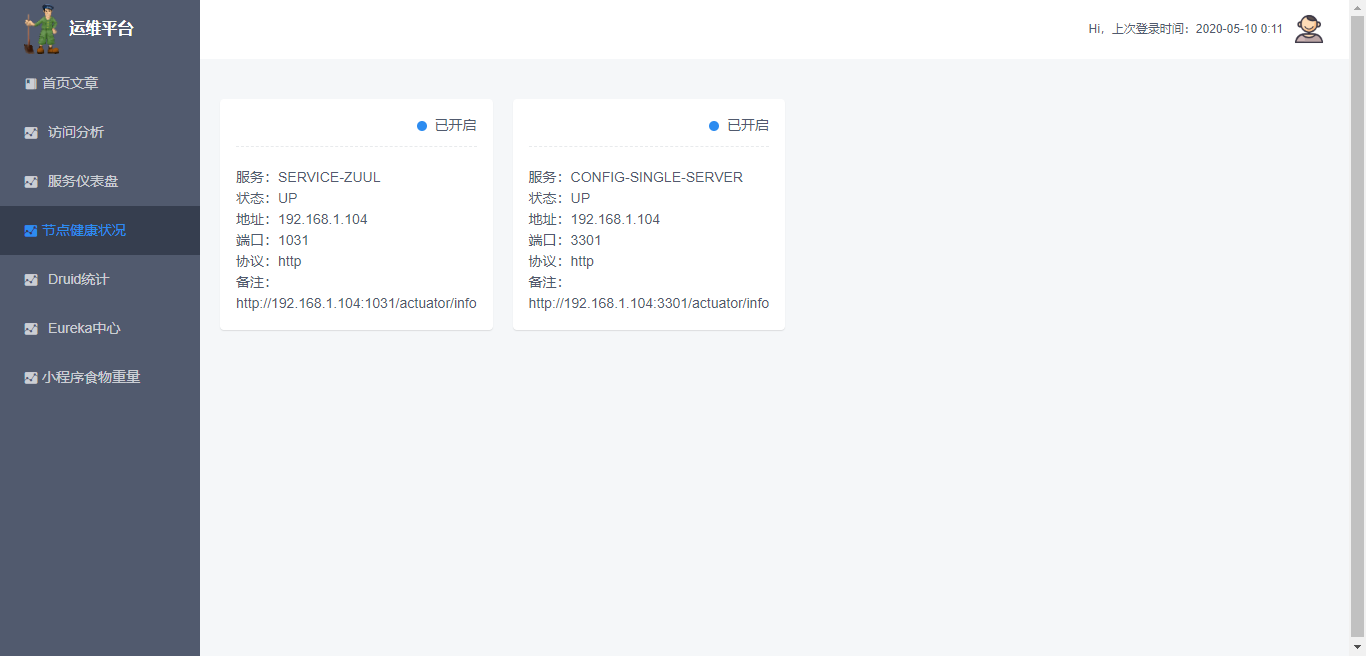
在小程序端，提供“系统帮助”功能，为了帮助用户更好的了解如何使用系统，用户可以进行查看，如图：

****

**图5-18 系统帮助**

## 5.10 节点健康检查

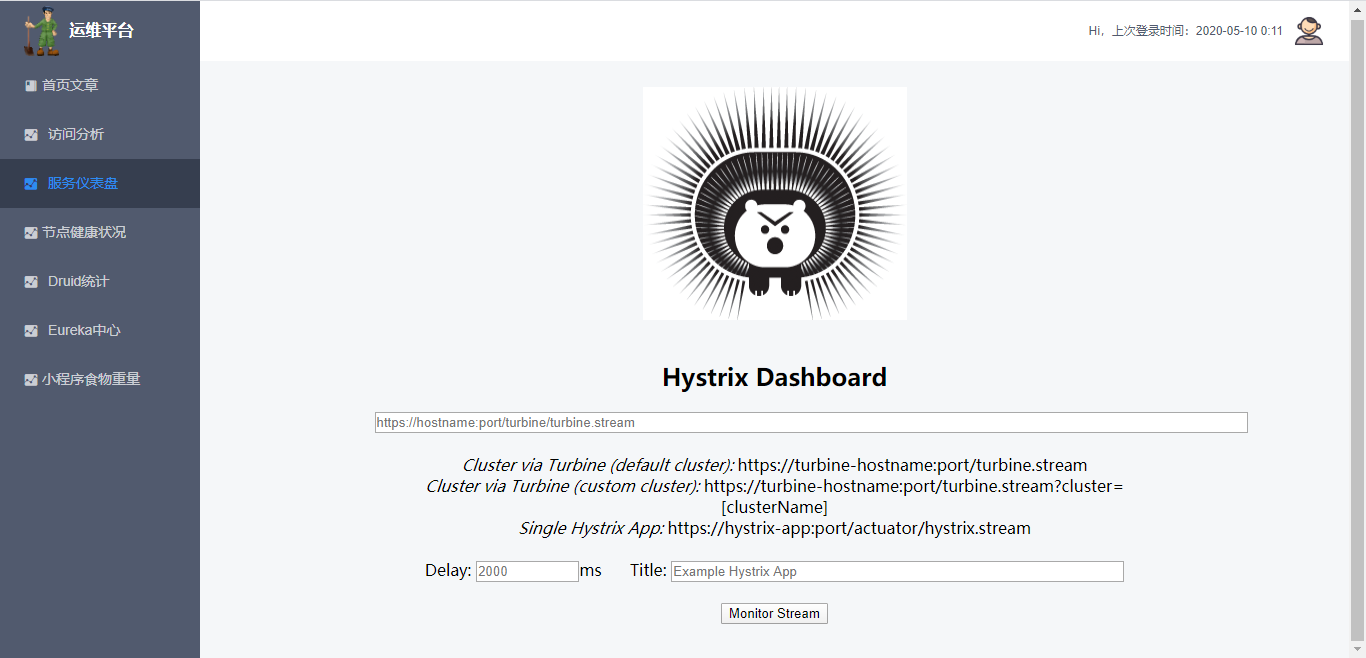
项目部署之后，运维人员可以时刻查看各个节点的“健康”状况以及当前节点所在的服务器地址、端口等信息，如下图：



**图5-19 节点健康状况**

## 5.11 仪表盘

项目仪表盘为Hystrix DashBorad技术。运维人员可以根据该技术进行查看服务器压力，如下图：

****

**图5-20 服务器压力仪表盘**

## 5.12 Druid统计

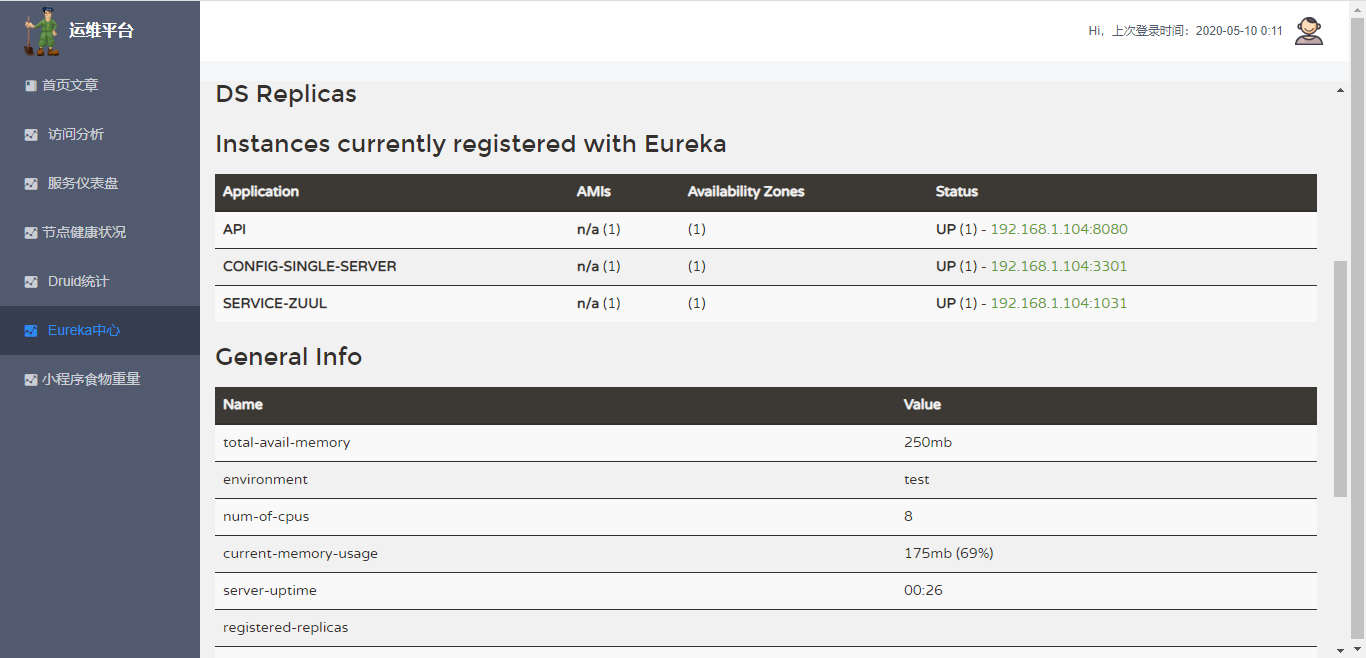
项目中使用Druid数据源，该数据源提供Web控制台，里面包括SQL执行的时间，以及是哪个URL触发了SQL记录，如下图：

****

**图5-21 Druid控制台**

## 5.13 Eureka中心

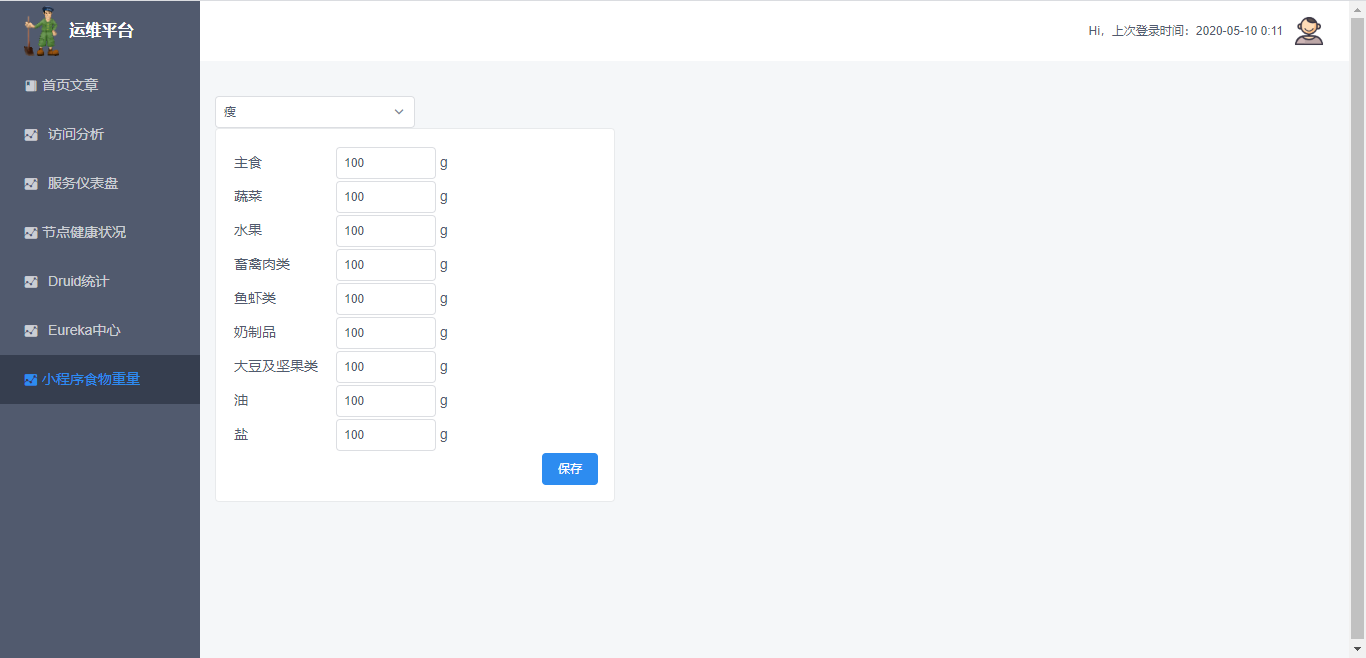
项目中，使用Eureka作为注册中心。Eureka注册中心自带控制台，用于显示所有注册到Eureka的服务节点，如下图：



**图5-22 Eureka控制台**

## 5.14 食物重量定制化

项目中，需要定制化的保存不同的BMI体型然后进行保存，小程序端会根据用户的BMI指数进行动态的推荐饮食，如下图：

****

**图5-23 饮食推荐**

## 5.15 小结

本章节主要讲述了项目测试，尽可能的覆盖项目中所有的测试功能。本章节主要对如下功能进行测试：小程序免密登录、发布文章、悄悄话、基本信息维护、每日目标、健康周报、划任务、系统帮助展示、系统节点健康检查展示、项目服务节点仪表盘、Druid SQL统计、Eureka微服务注册中心、食物重量定制化等功能。

**第6章 项目部署**

## 6.1 简述

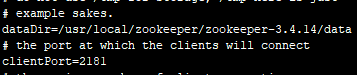
项目需要发布在服务器中，目前项目有三个角色，分别为小程序、运维PC端、Java后台，其中Java后端的服务分别部署在两台服务器：106.14.160.110以及47.101.216.18服务器。

## 6.2 配置JDK

每台服务器需要搭建JDK环境，并配置环境变量。

## 6.3 配置Zookeeper

在服务器106.14.160.110下载安装Zookeeper，并修改Zookeeper目录下的Zoo.cfg信息，修改如下图：

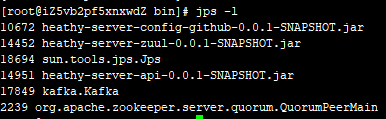


**图6-1 Zookeeper配置文件修改**

配置完毕之后，使用如下命令进行启动Zookeeper（以守护进程方式）：

nohup ./zkServer.sh > zookeeper.out &

使用Jps –l命令验证Zookeeper是否启动成功，如图：



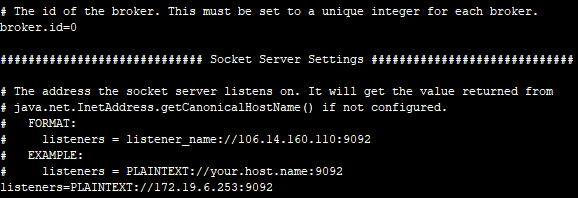
**图6-2 验证Zookeeper**

进程2239即为Zookeeper进程，说明启动成功。

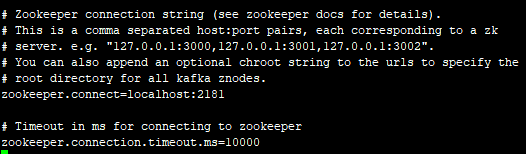
## 6.4 配置Kafka

前提：在106服务器确保Zookeeper安装且正确的运行才能保证Kafka的运行。

下载Kafka，修改Kafka如下配置，如下图：



**图6-3 配置Kafka**

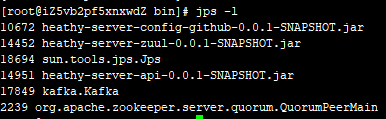


**图6-4 配置Kafka**

配置完毕之后，使用如下命令进行启动Kafka（以守护进程方式）：

nohup ../bin/kafka-server-start.sh ./server.properties &

使用Jps –l命令验证Kafka是否启动成功，如图：



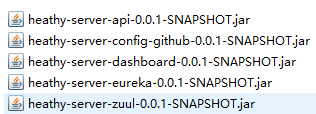
**图6-5 验证Kafka**

进程17849即为Kafka进程，说明启动成功。

## 6.5 部署业务模块

项目使用IDEA Maven进行各个SpringBoot服务模块打成Jar包，并部署在两台服务器中运行。

如下为项目中服务模块的Jar包：



**图6-6 服务Jar包**

其中heathy-server-api-0.0.1-SNAPSHOT.jar、heathy-server-config-github-0.0.1-SNAPSHOT.jar、heathy-server-zuul-0.0.1-SNAPSHOT.jar部署在106.14.160.110服务器中，启动模式格式均为如下方式：

nohup java –jar xxxx.jar > xxx.out &

剩下的服务：heathy-server-dashboard-0.0.1-SNAPSHOT.jar、heathy-server-eureka-0.0.1-SNAPSHOT.jar均部署在47.101.216.18服务器中，且启动方式与上相同。

## 6.6 小结

在前述所有章节中，分别对项目需求分析、技术选型、详细开发以及最后的软件测试进行全方位的说明。在系统开发完毕之后就需要对项目进行部署上线。本章节首先对配置Java基础环境：JDK进行说明，由于项目中使用了Kafka而其依赖于Zookeeper，因此先进行Zookeeper安装与启动随后对Kafka进行安装启动。最后在所有的基础环境得到保证之后开始部署项目业务服务。在最后进行验证。

**第7章 总结**

本项目采用微服务架构方式，采用SpringCloud一站式解决方案，所有的服务模块均采用SpringBoot作为服务单元进行开发。以最大程度的靠近微服务的架构思想。微服务架构的灵活性避免了传统软件的开发模式。

在最大限度上进行服务模块之间的相互解耦。也方便后期的服务扩容和负载。

项目中使用MySQL关系型数据库。并采用Innodb引擎。

在项目业务层，项目采用前后端分离的开发方式，两者之间的交互通过Axios或者Wx.Request进行交互，交互数据采用JSON格式数据，因为JSON有着轻便解析轻便等良好特点。

项目后端封装了统一应答（Response）模块，方便后端开发。后端多采用Java容器技术，如Map，ArrayList,LinkedList等相关容器。在容器之间，也做过比较，若需要对链表进行删除、查询操作那么项目中会多使用双向链表的容器模式：LinkedList，因为该中容器采用链表的方式进行扩容，而ArrayList本质为数据实现，当需要动态的添加或删除时，ArrayList需要将原数据重新转移到新的内存空间。该中做法相当的耗时和浪费内存。

在整个项目开发完成之后，对微服务有了更深的认识，发现有些业务场景不能为了微服务而去微服务，因为微服务会直接导致开发成本提高，比如服务节点的内存需要更大一些，宽带需要更好一些，这样才能承载一个项目包分成多个包多个端口同时运行，相互之间还需要通过注册中心进行相互调度。且在微服务中，模块之间的错误率也会提升，若一个服务宕机可能会导致整个项目的宕机。

比如在本项目中，若网关宕机，那么会导致整个项目无法使用。

因此开发过程中必须进行取舍，对于业务场景不复杂的情况下可以简单的使用业务处理。为了微服务而微服务不是明智之举。

**致 谢**

首先，感谢此处的毕设项目的参与机会。能够让我在做毕设的过程之中能够学习更多，感触更多。

感谢本次毕设的指导老师：傅启明老师。在本项目开发过程中，傅老师一直孜孜不倦，不吝教导我去从多方面考虑技术问题，参与了整个毕设项目的全方位指导。在傅老师的带领下，学习到了更多的开发技巧，更重要的是开发思路。我想只有开发思路明确，目标明确，做法明确，才能非常的高效开发出毕业设计出来。

感谢大学期间所有任课的教师，在他们的引路之下，我才能迅速的在软件开发的世界中迅速的找到自我，并使我对软件开发产生了非常浓烈的兴趣爱好，我也在这爱好之中在技术中不断徜徉。再次，谢谢各位老师。

感谢电子学院为我提供了良好的学习环境和非常浓烈的学习氛围。班级同学的浓烈氛围使得我能够在安静平和的环境之下不断的提升自我，电子学院计算机实验室的计算机设备为我提供了物质需求，我可以使用这些计算机设备提高自己的Linux运维等相关能力，谢谢电子学院。

再次也要感谢我的大学舍友，我们就像是家人一般，在困难面前我们互帮互助，上课的问题我们课后在宿舍及时的讨论，谈天论地，聊着各种社会热门的话题，热门的技术。再此，我感谢你们。谢谢你们大学时光的陪伴。

**参 考 文 献**

[1] 陈宇收,饶宏博,王英明,谷国栋,胡进贤.基于JWT的前后端分离程序设计研究[J].电脑编程技巧与维护,2019(09):11-12.

[2] 韩菊茹,杨秩,纪兆轩,马存庆.基于微信小程序的文件加密系统设计与实现[J].信息网络安全,2019(09):81-85.

[3] 曹郁.基于Docker容器的微服务研究与实现[J].科学技术创新,2019(28):97-98.

[4] 李苗,李志豪,徐芸,付宝君.微信平台下的个性化学习研究[J].中国教育信息化,2019(17):38-41.

[5] 欧阳宏基,杨铎.基于微服务架构的学位论文写作辅助平台[J].计算机与现代化,2019(10):34-39.

[6] 万书鹏,易强,张凯,彭晖,王毅,杨明.新一代调度控制系统基于微服务架构的服务编排技术[J].电力系统自动化,2019(09):10-30.

[7] 田子兰.浅谈linux系统的安全[J].电脑知识与技术,2019(30):1-2.

[8] 夏松竹. 高校本科毕业设计一体化教学管理系统的设计与实现[J]. 计算机教育, 2008(11):61-63.

[9] 陈亚莉.基于计算机Web 3D和Java程序开发的农机信息系统[J].农机化研究,2020,42(12):200-204.

[10] 曾裕涵,杨立敏,刘小西.基于分布式技术的数字档案信息管理系统设计[J].计算机时代,2020(04):58-60.

[11] 云岩. 基于微服务的校园应用门户网站的设计与实现[D].黑龙江大学,2018.