
南京大学

# 研 究 生 毕 业 论 文

# （申请工程硕士学位）

论 文 题 目 面向智能新风设备的场景构建系统设计与实现

作 者 姓 名 梁越勇

学科、专业名称 工程硕士（软件工程领域）

研 究 方 向 软件工程

指 导 教 师 刘嘉 副教授

**2021** 年 **05** 月 **20** 日

学　　号：**MF1932110**

论文答辩日期：**2021** 年 **05** 月 **20** 日

指 导 教 师：　　（签字）


# Design and implementation of a scene construction system for intelligent fresh air devices

By

**YueYong Liang**

Supervised by

Associate Professor **Jia Liu**

A Thesis
Submitted to the Software Institute
and the Graduate School
of Nanjing University
in Partial Fulfillment of the Requirements
for the Degree of
**Master of Engineering**

Software Institute

May 2021

# 南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：面向智能新风设备的场景构建系统设计与实现

工程硕士（软件工程领域） 专业 2019 级硕士生姓名：梁越勇

指导教师（姓名、职称）：刘嘉 副教授

## 摘 要

随着智能新风设备的普及，设备的控制体验将影响用户的接受程度。本文设计并实现了一个面向智能新风设备的场景构建系统，将用户已有设备作为最基本的控制单元，并以场景的概念将多个设备实现逻辑上的组合，主要包括场景组合的相关数据展示、场景内多个设备的联动控制等功能。该系统以移动应用的形式供用户使用，在一定程度上解决了传统智能家居方案中设备数量增多时控制体验较差的情形，可以为智能新风设备使用者以及系统管理员提供更高效的服务。

相较于传统控制多设备的手段，本文提出的方案更多地考虑设备间的相关性，并以此整合多设备数据，最终以移动应用的形式展现给用户。文中提出的场景构建方案从设备和场景两个方面入手。针对单一的智能新风设备如新风机、风扇等，提供基础的数据查看、设备控制等功能；针对多设备的联动，提供场景作为控制维度，实现对场景内多设备的数据总量查询与管理。本系统基于对设备的远程控制，在场景构建上实现逻辑的组合，用户可在本系统中查看已创建场景的空气信息，通过消息推送模块构建发送控制指令，并实现应用消息推送。与此同时，本系统会收集新风设备在运行时获取的空气质量数据和用户操作数据，用户可通过指定接口查询操作记录，并获取设备在运行时记录的数据。这些数据存储在指定数据库中，并作为后续数据分析模块的输入源，以此分析用户所在环境情况和使用习惯等。为了保证用户在多个平台有相似的场景控制体验，本系统采用 Flutter 跨平台 UI 框架，可以快速生成统一的多平台客户端。

用户可以通过下载 Apk 文件使用该系统进行设备的添加、控制，场景的构建、查看和运行，通过接口运行设备运行分析结果，并根据相应的指标生成对应的运行报告，对于智能家居产品的使用者和公司运营人员，具有较高的实用性。

**关键词**：场景构建，智能家居，微服务，异步消息推送

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Design and implementation of a scene construction system for intelligent fresh air devices

SPECIALIZATION: Software Engineering

POSTGRADUATE: YueYong Liang

MENTOR: Associate Professor Jia Liu

## Abstract

With the popularity of intelligent fresh air devices, the control experience of the devices will affect the acceptance of users. This thesis designs and implements a scene construction system for smart fresh air devices, taking users' existing devices as the most basic control unit and logically combining multiple devices with the concept of scenes, mainly including functions such as data display related to the combination of scenes and linkage control of multiple devices within the scenes. The system is available to users in the form of mobile application, which to a certain extent solves the situation of poor control experience when the number of devices increases in traditional smart home solutions, and can provide more efficient services to users of smart new air devices and system administrators.

Compared to traditional means of controlling multiple devices, the solution proposed in this thesis takes more into account the correlation between devices and uses this to integrate data from multiple devices, ultimately presenting it to the user in the form of a mobile application. The scene construction scheme proposed in this thesis starts with two aspects: devices and scenes. For single intelligent fresh air devices such as fresh air machines and fans, basic data viewing and device control functions are provided; for the linkage of multiple devices, scenes are provided as control dimensions to realize the total data query and management of multiple devices within the scenes. This system is based on the remote control of the equipment, and the combination of logic is realized in the scene construction. Users can view the air information of the created scene in this system, send control commands through the message push module construction, and realize the application message push. At the same time, this system will collect the air quality data and user operation data acquired by the fresh air equipment

during operation, and the user can query the operation records through the specified interface and acquire the data recorded by the equipment during operation. These data are stored in the specified database and used as the input source for the subsequent data analysis module to analyze the user's environmental conditions and usage habits, etc. To ensure that users have similar scenario control experience in multiple platforms, this system adopts Flutter cross-platform UI framework, which can quickly generate a unified multi-platform client.

Users can use the technology by downloading Apk file, adding and controlling devices, building, viewing and running scenes, running device operation analysis results through the interface and generating corresponding operation reports according to the corresponding indicators, which is highly practical for users of smart home products and company operators.

**Keywords:** Scene Construction, Smart Home, Micro Services, Asynchronous Message Push

# 目录

# 表 目 录

# 图 目 录

# 第一章　引言

## 1.1　选题的背景和意义

随着 5G 等技术的快速普及，物联网再一次成为技术发展的一大热门，并在多个领域有着广泛的应用，物联网中和人们日常生活最息息相关且最重要的应用之一就是智能家居 (Smart Home)。智能家居的历史最早可以追述到上世纪 80 年代，当时的美国联合科技公司通过已有科技手段改造了一栋老旧大楼并实现了该建筑的信息化 [1]。

目前，智能家居在一些海外国家和地区已经得到了普及。以美国为例，随着 Google、Apple 等巨头科技公司在该领域扎根，美国在智能家居领域领跑全球，更是成为全球智能设备最大的市场 [2]。与此同时，在中国，不论是传统家电公司，如格力、美的、海尔，还是新兴科技公司，如小米、华为等，都开始布局智能家居领域。正是由于这些公司的推动，智能家居在中国发展的这十年里，才逐渐被人们接受。近年来，随着硬件技术的智能化、自动化，智能家居领域的发展也愈加迅速。各地政府对智慧城市的建设越发重视，国内相关市场也得到了极大扩张，甚至达到千万亿级别 [3]。

由此可见，在智能家居领域潜藏着一个具有光明前景且消费潜力庞大的市场。正因为如此，才吸引越来越多的优秀科技公司加大对该领域的市场研究及调研，特别是对用户需求变化以及企业平台发展环境等有关方面的调研。如今智能家居在中国已经历了近 12 年的发展 [4]，一大批国内优秀的智能家居品牌快速崛起，成为智能家居产业中的中流砥柱。

与此同时，在家庭设备数量不断增加的情况下，室内污染也已经成为一个让人们忧心的话题，建筑物在修建的过程中会造成包括甲醛在内的多种室内空气污染 [5]。而这些室内空气污染往往消散周期长，对人体有害。人们长时间处在这种环境之下，不免会损害身体健康。针对这种情况，智能新风系统应运而生。

一份来自中怡康的调查数据显示[1]，该调查选取了 100 名住户并对其家庭空气展开调查，数据显示，98% 的住户家庭空气夏日白天自然常温在 28°C 以上，只有 38% 的住户空气湿度在 40-60% 的舒适区间，半数以上的住户家庭空气 PM2.5 浓度超过 75 微克/立方米（以我国 PM2.5 标准），近半成住户家庭空气甲醛浓度

[1] http://news.sina.com.cn/c/2014-09-04/005930790742.shtml

超过 0.08 毫克/立方米，77% 的住户家庭空气风速测试为零，既无流动。

空气舒适度包括湿度、温度、洁净度以及风速 [6]。从上列数据可以看出，室内空气整体状况并不理想。如果想让室内空气达到令人满意的舒适度，只靠空调或新风机等单一产品很难实现。本技术通过联动智能新风机、智能风扇等智能产品，提供了一种基于多设备的场景构建方案，以此构建智能室内空气场景，实现对室内空气温湿度、洁净度、微颗粒物等指标的一体化、全方面的快速调节，以此提高室内空气舒适度。

## 1.2　国内外研究现状及分析

和过去的计算机与互联网类似，物联网逐渐被视为整个通信和信息技术领域发展必然趋势，越来越多的人们相信物联网技术的发展会给人类的日常生活带来翻天覆地的变化。也正因为如此，世界各国无论是从理论上还是实际应用上都加强了对物联网的研究。

为了实现智能家居设备的场景构建，第一步需要实现与智能设备的连接配对，配对主要分为有线与无线两种，有线即通过数据线访问相应串口发送控制指令实现对硬件设备的控制，该种方式局限性较大，因此现阶段智能家居大多采用无线配网方式。

常见的无线配网方式主要有三种: 插 SIM 卡入网、连接 WI-FI 入网和使用 Zigbee 等连接网关代理入网 [7]。使用 SIM 卡直接入网摆脱了智能家居设备对无线路由器、网关等第三方硬件的依赖，更为灵活，但其自身携带的 SIM 卡在网络通信时会产生一定的资费，成本较高。使用 Zigbee 等连接网关代理入网需要提前与相应网关进行绑定 [7]，且不同厂商对于自家网关的协议定制程度不同，兼容性较差，难以通用。使用 Wi-Fi 入网是三者之间最为简单的一种配网方式，只需让智能设备获取所在环境内对应 Wi-Fi 的 SSID 与 Password 即可，缺点是需要用户安装相应的应用程序将网络信息发送至硬件设备 [8]。为了让用户可以最大化利用应用程序，该系统除设备配网外，还提供了设备控制等其余功能。

上世纪 90 年代，IBM 公司开发出一种用于将石油管道通信传感器连接到卫星的通信协议，该协议基于 TCP 协议，并利用发布-订阅模式成为了一种轻量的消息传输协议。IBM 公司将其命名为 MQTT 协议，该消息协议一经面世，便被应用到网络质量较差的环境中或硬件性能较差的设备端的通信场景。目前随着智能硬件的普及，MQTT 协议已经成为最通用的物联网通信协议之一了 [9]。

对于普通用户，如果想使用到智能家居产品，首要任务便是选择合适的使用入口。智能家居发展的不同方向将导致进入智能家居场景的入口也不尽相同。

作为最基本的切入点，路由器被许多人视为接入智能家居的最佳入口。由于家庭网络需要经过路由器得以分发，因此在一个普通用户的家中，几乎所有的联网设备都需要经过路由器才可以正常访问网络 [10]，在此联网过程中，大量交互产生的数据会被存储在路由器上，这些数据可以很好的反应整体智能家居互联的基础信息。继智能路由器之后，智能手机作为智能家居的入口迅速受到关注 [11]。由于其便携性，移动性和广泛的普及性，智能手机被广泛认为是未来智能家居的“通用遥控器”。后来，由于人工智能和人机交互技术的发展，智能语音助手也开始进入作为智能家居入口的阶段，无论是苹果的 Siri 还是小爱同学 [12]，就智能家居中的场景控制而言，智能语音助手将会在未来大放异彩。

## 1.3　本文的工作

根据功能对智能家居进行划分，可以将全屋智能分为八个模块：娱乐系统、安防系统、控制系统、照明系统、厨卫家电系统、网络及通信系统、健康医疗系统、室内环境系统 [13]。这八个模块共同联动，相辅相成，共同实现全屋智能。而本文工作面向的便是室内环境系统中的智能新风系统。

在生活中，我们可以直接接触到的最简易的智能新风系统是一台壁挂式新风机，通常情况下，壁挂式新风机其自身就携带 CO2/PM2.5 感应器 [14]，自己检测室内环境，按照实际情况和用户设置的目标数值自动运行。但也有更复杂的系统，包括管理基站、云端服务器、手机 App、电脑版用户界面、外置检测仪，还有各种各样的附带装置。它能够根据外界空气中的污染物（如 PM2.5）、温度、湿度，室内 CO2 含量，按照预先设定好的指标对传感器传导的信息进行分析判断，及时自动开启/关闭、开启和调节加热等功能。

在智能管理的同时，用户随时都可以自己重新调节需要的新风量，手动开机或关闭设备。但智能新风系统的终极目的是由系统自动根据环境中的空气数值如温湿度等空气指标让设备自动地执行对应地指令，使用户无需参与对智能新风设备的控制与管理。当你在整个住宅想要实现智能新风系统，则需要在每个房间检测空气质量 [15]。如果新风系统只能在一个房间检测到 CO2 或 PM2.5 浓度，新风系统无法真正地“智能”运行，因为它“不知道”哪个房间需要通风换气，哪些房间并没有人。

传统智能家居通常是以房间作为控制单元 [16]，虽然可以将设备组合起来，但仍需要一步步操作单个设备，如今随着智能设备的普及，房间内的设备数量也在不断增加，如果仍然需要通过单独操作某个设备才能实现房间内整体的空气质量的变化，显然是低效的。

由此可见，随着智能家居的发展，未来将会有越来越多的家庭配备智能新风系统，其基础设备也会越来越多，单一控制设备只会增加使用者的负担。因此，本文提出一种面向智能新风系统的场景构建技术，将多个设备抽象成一个场景，然后通过对单个场景的构建、执行，实现场景内所有设备的共同联动。本技术利用智能设备所携带的传感器获取所载环境的空气数据，并通过连接上的额网络实现和服务端的远程数据交互，在收集环境数据和用户行为数据的同时，还可以为用户定制使用建议，并根据使用习惯远程控制家具实现自动化调节室内空气环境。

## 1.4　本文的组织结构

智能新风系统是智能家居的一个重要组成部分，然而，目前的智能新风系统仅仅停留在对设备的远程控制，本质上还是人工操控手段。因此一旦设备数量增加，人工操作的成本将会提升。本文将会设计一套场景构建的方案，实现用户设备的逻辑统一，并基于该方案实现了一套面向智能新风设备的场景构建系统。本文的组织结构如下：

开篇引言部分阐述了智能家居中的智能新风设备实现场景构建的背景及意义，并介绍了国内外对于智能家居研究现状、相关技术发展的主要研究现状，并说明了本文的主要研究目标以及为了实现一个完整的面向智能新风设备的场景构建系统所需要做的主要工作。

第二章的技术综述部分，介绍了本系统在实现过程中使用到的开发技术与开源库和框架，其中包括跨平台应用设计框架、微服务框架、以消息队列为主的异步消息处理技术、相关的数据存储技术以及 Docker 容器技术。

为了推进整个系统的实现，在第三章对整个系统的技术方案进行了需求分析与概要设计，在本章提出了整个系统在实现过程中所需要满足的基本需求，并对面向智能新风设备的场景构建系统构思了总体进度的设计，从功能的角度对系统进行了各模块的划分。

第四章详细介绍各功能模块的设计思路及实现方法，在原有需求分析的基础上，结合相应的技术阐述了各功能模块的实现过程，并以代码和图表的形式将方法与结果展示出来。

第五章基于第三、四章的需求分析和详细设计针对系统的功能性和非功能性设计了相应的测试用例并验证，以此确保整个系统的高可用以及稳定运行。

最后对整个系统进行总结，阐述了在项目实现过程中的些许不足，并对其进行总结描述，并在修后指出了本系统未来可以改进的方向。

# 第二章　技术综述

## 2.1　Spring 开发框架

Spring 是一款 Java Web 应用开发框架，简化了 Java 企业级应用的开发流程 [17]，为 Java 企业级应用开发注入了新的活力。Spring 的核心思想有两个，分别是控制反转和面向切面编程）[18]。和另一个 Java 应用框架 EJB 相比，Spring 依赖资源更少，更加轻量。依赖控制反转这一思想，Spring 这一框架使得项目更加方便解耦 [18]，简化开发者的开发流程。其面向切面编程的思想，可以快捷方便地实现对程序进行权限拦截、运行监控等功能。借助这两个思想，Spring 快速成为一款极其优秀地 Web 应用开发框架 [19]。

### 2.1.1　Spring Boot

Spring Boot 是一个基于 Java 平台的一个开源框架。其设计目的是用来简化 Spring 应用的初始搭建以及开发过程 [20]。该框架从 2002 年开始立项，经历了 18 年的更新，是目前最为稳定、生态环境最为完备的框架之一。Spring Boot 可以较为简单的创建生产级别的应用程序。通过 maven 或者 Gradle 等包管理工具，可以十分简便的调用 Spring 官方实现的类库和第三方用户或机构发布的类库。Spring Boot 由 Java 语言实现，在提供对 Java 语言的基础上，在 Spring 5.0 版本引入了对 Kotlin 语言的支持。通过使用 Kotlin 的语言特性可以有效简化代码的繁琐程度，

### 2.1.2　选择 Spring Cloud 作为微服务框架的优势

微服务化的核心就是将传统的单体应用（一体多功能）根据业务拆分成一个一个的服务（一服务一功能），彻底的消除耦合 [21]。从技术角度看微服务就是一种小而独立的处理过程。Spring Cloud 利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，Spring Cloud 为开发人员提供了快速构建分布式系统的一些工具。

在本项目中，由于功能模块较多，且需要使用到的开发中间件较多，且这些中间件均开放 Java 的接口，在使用的便捷性以及成熟度上相较于其他微服务框架更具优势，因此本文中的技术采用 Spring Cloud 作为微服务框架。

## 2.2　Flutter 技术

Flutter 是一个制作跨平台应用的框架，其目的是允许一套代码在多操作系统平台的复用，同时提供 MethodChannel 的方式实现应用程序与底层平台服务的直接对接。通过使用 Flutter，可以让开发者快速实现在不同平台的高性能应用，在更多的平台实现代码共享 [22]。

在开发过程中，Flutter 应用运行在一个虚拟机中，该虚拟机会保存应用程序的状态，即提供有状态的变化热重载，而不需要完全重新编译。发布时，Flutter 应用直接编译成机器代码，无论是 Intel x64，还是 ARM 指令，如果针对 Web，则编译成 JavaScript[23]。该框架是开源的，采用允许的 BSD 许可证，并拥有一个繁荣的第三方包生态系统，补充核心库功能。

Flutter 支持主流操作系统作为开发环境，如 Windows、macOS、Linux 和 ChromeOS，并且得到了目前使用广泛的集成开发环境的支持，如 Android Studio 和 Visual Studio Code。Flutter 采用 Dart 作为编程语言，该语言支持 AOT 和 JIT 两种运行方式，并通过“热重载”[24] 技术实现应用程序的快速更改，无需重新编译运行。

除 Flutter 之外，许多科技公司和开发者都提出过对应的跨平台应用方案，例如 Facebook 提出的 ReactNative、基于 Vue 的 Weex 和基于 WebView 的 Cordova、AppCan 等 [25]。

以 WebView 为基础的框架具有非常明显的优点，这类框架可以很容易的使用现代 Web 开发中已经成熟的插件、工具库等。但由于各个平台对于 WebView 的实现方式不同，各安卓厂商系统版本以及定制程度不同，其渲染效率和 JavaScript 的执行效率层次不齐 [26]，很难保证基于 WebView 框架制作出的应用程序提供近乎一致的优秀体验。

对于上述这种情况，Flutter 采取了一种全新的解决方案，即不使用已有 Web 的控件，重新编写一套完整的跨平台 UI，包括 UI 控件、渲染逻辑甚至开发语言。将应用程序的渲染交给跨平台的 Skia 图形渲染库实现，只使用各自平台提供的图形渲染接口。通过这种方式可以最大程度的保证开发的应用程序可以最小的依赖平台，以更高的执行效率使得其在各个平台的体验近乎一致 [27]。

## 2.3　异步消息处理技术

相对于同步技术，异步消息的传递不需要等待来自接受端的结果响应。服务端触发消息时，即使客户端被关闭，也依然可以实现消息传递。从沟通方式看，异步消息发送是一种单向的通信方式。

当使用异步消息传送时，消息的生产者在将消息发出后无需等待消费者响应，即可处理剩余任务。和同步处理不同的是，异步通信的两者即生产者和消费者无需同时运行，也无需同时进行任务处理 [28]。为了实现异步消息处理，通常会采用多线程或者消息队列等方案。

无论是多线程或消息队列都使用了队列这一数据结构 [29]，如阻塞队列、优先级队列。因此，两者实现原理上具有一定的相似性。多线程在使用时会依赖宿主机，即宿主机即充当消费者也充当生产者，所有的运算压力均会由宿主机承担，因此该方案适合非分布式系统。但在分布式架构 [30] 下消息队列明显是更优项；而且消息队列的耦合性更低，可扩展性更好，适用于弱一致性的场景。同样的，由于本技术使用到了分布式技术，在不同系统间的进行服务调用（调用协议也可能不一致），通过多线程很难实现或开销很大，而消息队列可以屏蔽不同机器或不同协议的问题 [31]。因此本文主要使用消息队列进行异步消息处理。

## 2.4　数据存储技术

### 2.4.1　MySQL 数据库

MySQL 是目前市面上使用最广泛的关系型数据库之一，因为其具有完整的事务安全机制、ACID 特性以及健全的回滚机制 [32] 而广受好评。

MySQL 使用 SQL 语言进行数据库访问操作，可以实现复杂的数据库查询操作。通常以数据表的形式存储数据，这些数据通常也被成为关系型数据，与其他数据库相比，MySQL 具有体积小、开源、成本低的特点，同时允许存储大量数据，支持集群部署且不会对访问数据库的用户数量进行限制，本文需要对数据进行整合预处理，并持久化相关的数据，因此采用 MySQL 完成对该部分数据的持久化。

### 2.4.2　MongoDB

MongoDB 是一种面向文档的非关系型数据库，即 NoSQL(Not Only SQL) 数据库。和传统数据库不同的是，在 MongoDB 中，并不存在传统关系型数据库中的数据表，在 MongoDB 中，数据以集合的形式存在，集合中的数据以文档 (Document) 的形式存储 [33]。文档是一系列字段 (Field) 的集合，其中字段可以包含如列表等复杂的结构，并且以键值对的形式存储。MongoDB 中集合可以包含任意种类的文档，即文档不需要设置相同的字段，且相同的字段不需要相同的数据库类型。MongoDB 的数据插入、以及更新简单的查询具有较好的性能，但

是不按照键值进行更新和查找时效率低，由于本文所介绍的系统需要大量非结构化数据，同时只需要对其进行简单的查找操作，例如场景内的设备操作指令，这一部分数据适合使用 MongoDB 数据库进行存储。

### 2.4.3 Redis

Redis 是一个开源的，使用内存存储数据的数据存储系统，由于其将数据直接存储在宿主机内存中，其运行速度相较于将数据存储在磁盘的数据库会更块，并可在短时间内实现多次大批量数据的读写操作。因此通常被用作缓存以及消息代理 [34]。

Redis 的存储基于键值对，且 Redis 中的值支持多种类型的数据，包括字符串、哈希、列表、集合、有序集合等 [35]。Redis 作为内存数据库，读写速度较快，且支持高可用和分布式，因此本文所述系统使用 Redis 作为缓存数据库，临时存储需要频繁进行存取的少量数据，提升系统的运行性能。

## 2.5 容器虚拟化技术

在软件开发中，一大难题就是软件运行环境的配置，由于用户计算机的环境大不相同，程序运行时往往会出现意想不到的问题。因此为了保证程序的正常运行，开发者通常需要保证操作系统与运行时组件的正确安装。虚拟机是一种可以在操作系统里面运行另一种操作系统的技术，从而让内部操作系统对自身的环境毫无感知，只是这种技术占用资源较多、而且由于荣誉步骤较多使其启动速度也非常缓慢。由于上述原因，Linux 容器技术便应运而生 [36]，Linux 容器不是模拟一个完整的操作系统，而是对进程进行隔离 [37]。正是由于其是进程级别的，相较于虚拟机，他占用的资源更少、启动也更快，可以将 Linux 容器看成一种轻量级的虚拟机 [38]，可以以更少的资源实现与虚拟机类似的功能。

### 2.5.1 Docker

Docker 并不是一种 Linux 容器，而是对 Linux 容器的一种封装方案，通过提供简单易用的容器使用接口 [39]，使开发者可以更方便的使用 Linux 容器。在使用 Docker 时，引擎会将应用程序及其所需要的依赖环境、组件打包成一个文件，通过运行该文件，可以生成一个虚拟容器。应用程序在该容器重的运行效果与在物理机中的运行效果几乎一致 [40]。使用 Docker 可以让开发者更方便的创建和使用容器，并进行版本管理、复制、分享、修改等操作。利用 Docker 可以提供一次性的环境，提供弹性的云服务并通过多个容器组建微服务架构。

### 2.5.2　Docker-Compose 容器编排

Docker-Compose 是一个命令行工具，该工具由 Docker 提供，用来定义和运行由多个容器组成的应用 [41]。使用 Docker-Compose，我们可以通过 YAML 文件声明式的定义应用程序的各个服务，并由单个命令完成应用的创建和启动。在软件开发中，一个项目可以由多个服务（容器）关联而成 [42]，Compose 面向项目进行管理，通过子命令对项目中的一组容器进行便捷地生命周期管理。

## 2.6　本章小结

本章主要介绍了面向场景构建技术的相关技术框架、异步消息处理技术和容器技术。首先介绍了场景构建系统的微服务框架 Spring Cloud 以及每个模块内的开发框架 Spring Boot。随后介绍了前端采用的跨平台应用开发框架 Flutter，在介绍了本技术中用于实现场景与设备控制的异步消息处理技术后，随即介绍了本技术背后涉及到的相关数据库，包括 MongoDB、MySQL 和 Redis，并根据其自身特性在不同的场景下使用，最后介绍了本技术整合这些框架、中间件的基础应用容器技术即 Docker 和 Docker-Compose 容器编排技术。

# 第三章　智能新风设备场景构建系统的需求分析与设计

## 3.1　系统总体规划

本技术主要针对的是已有新风系统的用户，如图3.1所示的场景构建系统流程示意图。智能家居终端设备（如新风机、风扇等）在运行过程中其自身传感器会不断产生设备数据，这些数据主要包括 PM2.5、甲醛、二氧化碳等空气数据。并通过自身硬件模块将数据上传至云端服务器，系统会根据数据的特性进行整合，清理一些不重要的数据，最终将数据存储在相应的数据库中。用户在使用场景前需要先构建场景，并在其中选择添加自己已拥有设备和该设备可进行的操作选项。创建成功后便会将场景数据（如场景内设备和操作指令等）存储在云端，之后触发场景即可让智能家居终端设备响应。

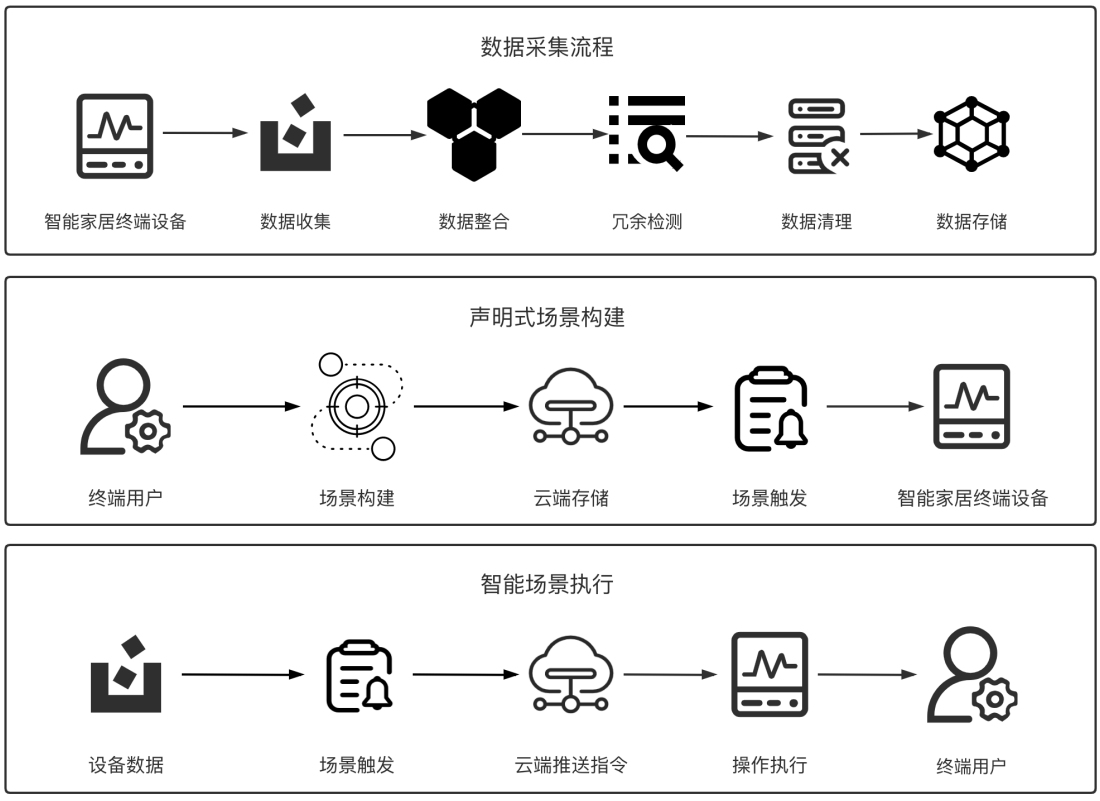


图 3.1: 场景构建系统流程示意图

本技术计划由权限管理模块、设备控制模块、场景构建模块、消息推送模

块、日志处理和数据分析六部分组成：权限管理模块负责对用户进行身份认证、鉴权授权以及对系统的网关进行配置，以此来确保整个系统的安全和稳定；设备控制模块以 MQTT 协议为核心，负责实现服务器对设备的远程控制，并通过 HTTP 协议实现用户和服务器间的通信；场景构建模块使用户可以在客户端实现场景的创建、修改和删除，设备和操作的添加等，为设备的控制提供了一种抽象；消息推送模块以 APP 推送为核心，通过对设备数值的监控，一旦发现异常数据便会由服务器推送消息到用户的手机。日志处理模块负责记录用户的行为操作、设备的指令执行情况，并对外提供页面供用户查看设备的运行情况，并给数据分析模块提供数据源；数据分析模块以其他各个模块作为数据的输入源，通过对这些数据的整理与分析，构建出区域及用户对于智能新风机的使用情况分布。多模块相互协同，最终实现一个完整的场景构建功能。

## 3.2　场景构建技术需求分析

### 3.2.1　功能性需求

从系统角度看，本技术支持用户通过终端设备远程控制智能新风设备，即在 App 内给出设备控制页面。并且在每次操作前都会验证用户身份，判断该用户是否有权限执行此类操作。除去单独对设备进行远程控制外，用户还可以在自身操作权限允许的情况下对场景进行操作。在执行操作后会通过消息推送模块将所需指令发送给设备。如果监测到异常数值，也会以消息的方式通知到用户设备上。上述这些操作，在执行时均会以日志的形式保存下来，之后再通过对这些数据进行分析和整理从而更加了解用户的使用习惯，不断完善设备和终端产品的性能和用户体验。根据上述功能，本技术主要分成六个模块：权限管理模块、设备控制模块、消息推送模块、日志处理模块、场景构建模块和数据分析模块。各个模块相互协调配合，以此实现整个系统的完美运行。

权限管理模块主要实现对系统用户的认证及请求鉴权的工作。所谓鉴权，指的是用户是否拥有访问系统的权利，用户在操作终端 App 时，会以 HTTP 请求的形式和云端服务器沟通，服务器通过 OAuth2.0 方式进行认证，判断该用户是否具备执行某操作的权限。而对于不合法的请求，服务器也会拒绝相应的操作，并作出提示，同时记录设备的异常登录行为，并结合消息推送模块通知用户。权限管理模块的功能需求如表3.1所示。

表 3.1: 权限管理功能性需求列表

| 需求 ID | 需求名称 | 需求描述 |
| --- | --- | --- |
| R1 | 使用者身份认证 | 系统用户需要通过手机号和验证码登录该系统，确认声明者的身份后，服务端发送相应的 token 到用户手中 |
| R2 | 鉴权 | 用户的每次请求需携带服务端发送的鉴权信息，如 token，服务端对其所声明的真实性进行鉴别，确认无误后才允许用户进行相应的操作。 |
| R3 | 保存操作记录 | 如有异常登录行为，后端自动记录操作日志，并监控敏感操作，完成后发送通知信息。 |

为了实现智能设备与云端服务器的双向通信，设备控制模块采用 MQTT 协议实现。在场景构建系统运行时，后端服务将会监听指定的 MQTT 信息主题。设备在运行过程中产生的数据经由网络上传至服务器后，设备控制模块将会接收并保存这些数据到指定数据库中。不仅如此，设备控制模块还负责向物联网设备发送控制指令，并在发出指令的同时将操作日志记录到日志模块，以此实现对硬件设备的固件升级、操作行为等的监控，然后结合数据分析模块完成分析报表导出。设备控制模块的主要功能需求如表3.2所示。

表 3.2: 设备控制模块功能性需求列表

| 需求 ID | 需求名称 | 需求描述 |
| --- | --- | --- |
| R1 | 发送设备控制指令 | 根据用户选择需要控制的设备及指令，设置对应的控制消息，然后发送对设备的控制请求，收到指令后，设备端执行对应的动作。 |
| R2 | 数据存储 | 定时获取设备端采集的数据，并存储到指定数据库中，同时视情况而定设置对应的缓存，以此用作数据分析。 |
| R3 | 保存操作记录 | 用户执行操作时，视情况将操作行为记录到对应数据库中。 |

场景构建模块负责实现用户对场景的操作，包括场景的创建、修改、场景内设备操作的添加、删除等。通过该模块，用户可以以场景的方式对多个设备进行操作，而无需对场景内的设备一一进行控制。同时还可以直接查看场景内的环境数据，如温度、湿度、PM2.5 等数值。其目的便是方便用户直观的查看和操作

场景。场景构建模块的主要功能需求如表3.3所示。

表 3.3: 场景构建模块功能性需求列表

| 需求 ID | 需求名称 | 需求描述 |
| --- | --- | --- |
| R1 | 显示设备列表 | 显示用户所拥有设备列表及相关状态，例如开关机状态、温湿度等反应空气质量的数值。 |
| R2 | 显示可操作选项 | 不同设备可操作的选项不同，即控制指令不同，在创建场景前需显示出可操作设备及操作选项，例如设备的开关机等。 |
| R3 | 创建、删除、执行场景 | 对场景的基本操作，用户可在系统内实现对场景的创建、修改、删除、设备添加等操作。 |
| R4 | 显示场景列表 | 分页显示用户已创建的场景列表，并给出执行选项。 |
| R5 | 查看场景状态 | 根据用户请求分析出目前场景内的相关数据。 |

消息推送模块负责消息的收发。在云端服务器收到执行场景的请求后，会查询出场景内所需要执行的操作，由于场景内设备数量和操作数量较多，若是以同步的方式对设备进行操作，用户需要等待的时间过长，影响用户体验，因此采用异步的方式实现消息的推送。消息推送模块的主要功能需求如表3.4所示。

表 3.4: 消息推送模块功能性需求列表

| 需求 ID | 需求名称 | 需求描述 |
| --- | --- | --- |
| R1 | 消息的发送与接收 | 允许消息的服务端可以发送出消息，并且客户端可以收到消息，及时响应 |
| R2 | 设备通知 | 如果有异常的设备数据，可以通过该模块将异常信息推送到用户的手机等设备上 |

用户在创建场景、执行场景、操作设备等情况时会以日志的形式将操作信息记录下来，并给日志处理模块提供数据源。在对数据进行了一个简单的整理后，给前端暴露相关的 Restful API 供用户查询，查询结果按照时间进行降序排序。由于设备日志数据过多，在记录日志时通过多线程，在主线程外实现对日志的记录。在查询日志时，为避免一次产生大量数据，提供分页查询的方式，同时也可以优化用户体验。日志处理模块的主要功能需求如表3.5所示。

表 3.5: 日志处理模块功能性需求列表

| 需求 ID | 需求名称 | 需求描述 |
| --- | --- | --- |
| R1 | 日志的创建与存储 | 提供接口供其他模块按照指定的格式创建日志，并将消息存放到指定位置的数据库中 |
| R2 | 日志的整理与获取 | 将存储的日志进行简单的整理，并对外暴露相关的接口供用户查看设备、场景等操作记录信息。 |

在设备运行、场景构建、消息推送等环节均会产生大量的数据，通过日志处理模块可以对数据数据进行一个简单的整理，但仅通过简单的整理不足以让这些数据发挥应有的优势。数据分析模块便是实现通过数据为业务赋能的核心部分。通过对设备产生的数据进行分析，如学习用户操作设备的时间段、操作习惯等指标，可以针对性的给出使用建议，给用户带来更好的使用体验。数据分析模块的主要功能需求如表3.6所示。

表 3.6: 数据分析模块功能性需求列表

| 需求 ID | 需求名称 | 需求描述 |
| --- | --- | --- |
| R1 | 数据的读取 | 实现对日志数据、行为数据等数据的读取 |
| R2 | 数据分析 | 根据相关指标实现对数据的整理与分析 |

### 3.2.2　非功能性需求

如表3.7所示为面向智能新风设备的场景构建系统的非功能性需求列表。非功能性需求主要从系统可靠性、系统并发性、用户行为安全性和系统可拓展性几个方面进行描述，主要是为了提高系统的开发效率，保障软件系统运行质量。

表 3.7: 场景构建系统的非功能需求

| 类型 | 非功能需求描述 |
| --- | --- |
| 时间特性 | 系统内页面跳转，延迟时间不超过 2s；系统发出设备控制命令后，设备执行延迟时间不超过 2s，页面响应时间不超过 2s |
| 并发性 | 支持至少 10 位用户并发访问同一系统接口 |
| 可靠性 | 发生单一故障时不影响其他模块使用且可回溯 |
| 安全性 | 本系统仅允许系统内用户使用，未鉴权或已过期用户无法使用功能 |
| 可拓展性 | 在尽量不影响当前系统的情况下可以快速新增设备及功能 |

### 3.2.3　场景构建系统用例设计

根据上文所述功能需求，得到如图3.2所示的场景构建技术系统用例图。根据的背景和目标分析，面向智能新风系统的场景构建技术主要针对设备用户。用例图中主要涉及 7 个用例，分别是权限校验、设备添加、设备控制、场景创建、场景查看、场景控制和消息推送。

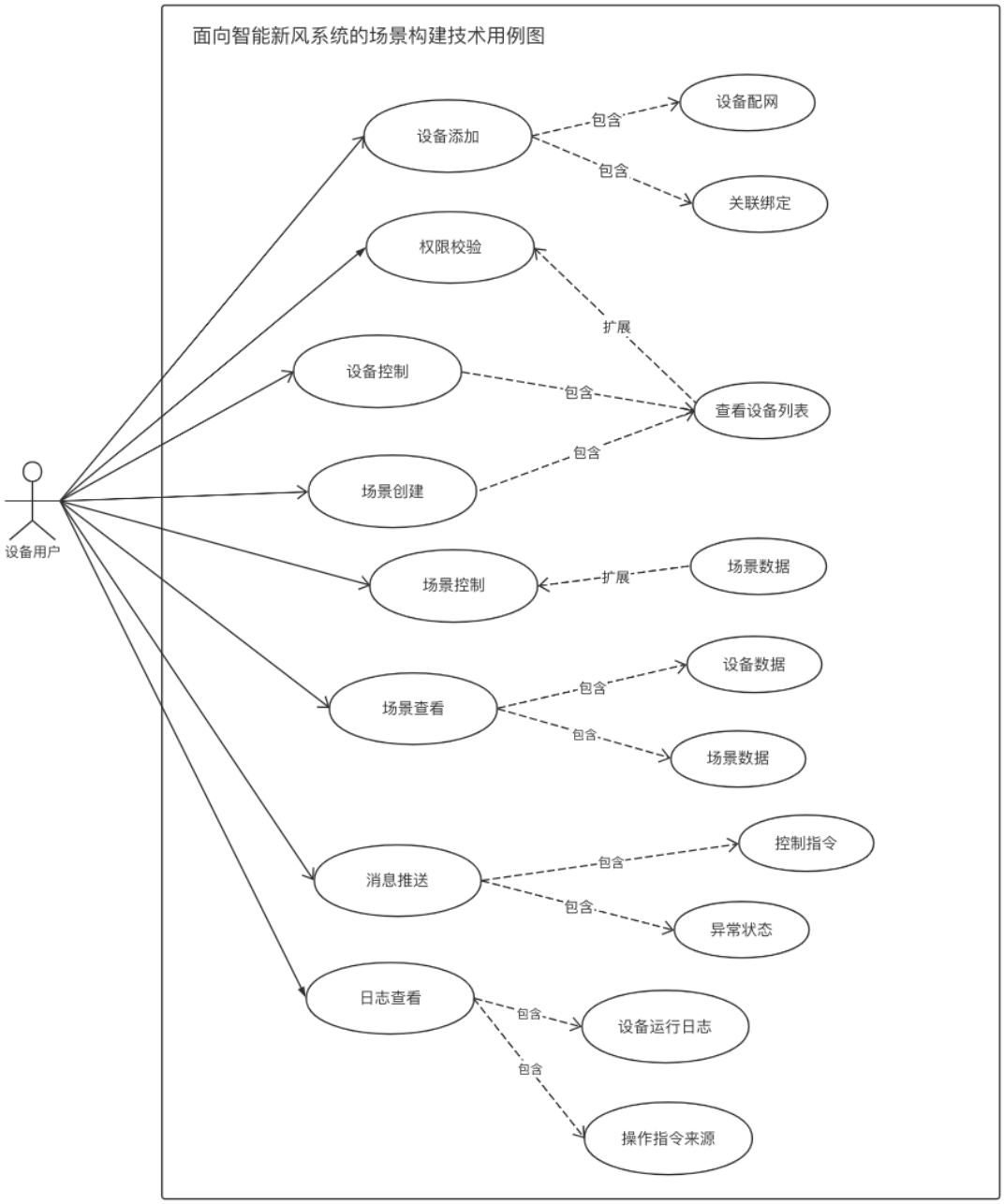


图 3.2: 场景构建技术系统用例图

用户登录系统后，可通过系统进行设备的添加，如果设备未入网，则会弹出提示要求设备优先入网，待设备入网后随机与设备用户账号绑定。同理，也可以对设备和用户进行解绑操作，以及强制使设备处于离线状态重新配网。根据用户权限的设置，用户可查看本人所添加的设备和场景列表，有利于保护私人数据的隐私。此外，用户还可以根据设备特性定时启动关闭新风设备，以通过对设备的操作达到对所处环境的控制。

除去设备操作外，用户还可以依据设备属性将多个设备组成场景。设备用户可以创建场景，从已添加关联绑定的设备中选择需要组合的设备，新建场景信息，并添加设备操作。系统收到请求后便会在云端记录该场景的相关信息。所有的场景信息均可在场景列表中查看，用户可以查看对应场景的空气情况。所有场景均支持远程启动，用户仅需要点击手机 APP 上的场景执行按钮即可。场景创建并执行成功后，用户可以收到提示并检测到相应的设备运行状态。

权限校验用例如表3.8所示，用户登录系统需验证身份，通过手机号 + 验证码或微信登录等方式登录到系统。只有在登录系统后才可以进行设备添加、设备列表查看、场景创建控制等行为。用户成功登录系统后会收到一个来自系统的凭证，之后的所有操作请求需携带此凭证才可以完成用户身份的验证。如果凭证已过期，还需要重新获取新的凭证以保证用户的正常使用。

表 3.8: 权限校验用例描述

| ID | UC01 | 名称 | 权限验证 |
|---|---|---|---|
| 描述 | 用户进行权限验证 | | |
| 参与者 | 系统用户 | | |
| 触发条件 | 用户进入到登录页面 | | |
| 前置条件 | 无 | | |
| 后置条件 | 系统反馈用户登录状态 | | |
| 优先级 | 高 | | |
| 正常流程 | 1. 用户进入到登录界面<br>2. 用户选择登录方式<br>3. 用户根据所选登录方式填写相关信息，如手机号和验证码<br>4. 用户点击确认按钮，提交登录信息<br>5. 系统校验成功后返回登录凭证<br>6. 用户保存系统下发的登录凭证 | | |
| 扩展流程 | 3a. 获取验证码失败，系统提示错误信息<br>4a. 验证码校验失败，系统提示信息 | | |

| | |
|---|---|
| 扩展流程 | 4b. 用户点击取消按钮，返回系统首页<br>5a. 系统校验失败，系统反馈失败信息 |
| 特殊需求 | 允许用户在多台设备上登录 |

表3.9描述了设备添加用例，用户登录系统后执行设备添加操作，需要扫描设备二维码，判断设备是否已经入网，如果设备未入网，则会提示用户进行设备配网，然后再执行设备与账户的关联绑定。设备配网时需要知道所处环境的 Wi-Fi 名称和密码，设备入网并进行账号关联后，后台会进行设备记录等操作，设备状态可以在系统的设备列表中查看。

表 3.9: 设备添加用例描述

| ID | UC02 | 名称 | 设备添加 |
|---|---|---|---|
| 描述 | 用户添加设备入网 | | |
| 参与者 | 系统使用者 | | |
| 触发条件 | 用户选择设备添加选项 | | |
| 前置条件 | 用户需要进行设备配网、将设备与账号关联 | | |
| 后置条件 | 系统反馈设备状态 | | |
| 优先级 | 高 | | |
| 正常流程 | 1. 用户进入到添加设备界面<br>2. 用户扫描设备二维码<br>3. 用户对设备进行配网操作<br>4. 用户配置设备别名、位置等信息<br>5. 用户点击确认按钮，提交设备关联信息 | | |
| 扩展流程 | 2a. 用户手机摄像头异常，从相册中选取二维码信息<br>3a. 连接 Wi-Fi 种类非 2.4GHZ Wi-Fi，系统提示错误信息<br>5a. 用户取消操作，返回系统首页 | | |
| 特殊需求 | 若设备已被其他账号关联绑定，系统应提示该设备已被关联。避免设备重复添加 | | |

设备控制用例如表3.10所示，用户在进行权限校验后可以查看已关联的设备列表，并且依据设备特性对设备进行例如温度、风速、冷热风切换、定时、蜂鸣器开关、按键锁等远程操作。用户执行操作后，会在前端查看到反馈信息，例如设备执行成功等相关提示。与此同时，后台会对这些操作以日志的形式进行记录，方便对操作行为进行分析。用户可以直接通过查看设备状态判断远程控制是否执行成功。

表 3.10: 设备控制用例描述

| ID | UC03 | 名称 | 设备控制 |
|---|---|---|---|
| 描述 | 用户进行设备控制操作 | | |
| 参与者 | 系统用户 | | |
| 触发条件 | 用户进入到单个设备操作界面 | | |
| 前置条件 | 用户需要根据界面中的控制选项点击相关按键 | | |
| 后置条件 | 设备执行相应的操作，后端记录操作行为 | | |
| 优先级 | 高 | | |
| 正常流程 | 1. 用户进入到设备列表页面<br>2. 选择设备列表中的某一个设备，进入到单个设备操作界面<br>3. 用户根据界面中不同的控制选项选择操作方式<br>4. 用户执行相应的操作，如开关、温度设置、风速设置等<br>5. 指令发送成功，设备收到响应后并执行操作<br>6. 用户点击返回，推出设备操作页面 | | |
| 扩展流程 | 2a. 设备离线，无法进入操作页面，系统给出重新配网选项<br>4a. 用户发送指令后云端服务器异常，系统反馈失败信息<br>5a. 云端服务器将指令发送到设备时异常，系统反馈失败信息 | | |
| 特殊需求 | 若发送控制请求时间过长，应根据情况判断设备是否已接收到指令避免指令重复发送 | | |

用户可以对添加的设备进行管理，以场景为维度对设备进行状态查看、设备联动操作等动作，并以此获取场景的相关信息，如名称、设备控制指令等。同时允许用户进行查看、修改和执行已经创建的场景。创建场景时，用户需要设置场景的名称，获取场景内设备列表后在选择对应设备可进行的操作，对设备的多个操作进行排序后方可将场景信息提交到服务器端。场景创建的用例描述如表3.11所示。

表 3.11: 场景创建用例描述

| ID | UC04 | 名称 | 场景创建 |
|---|---|---|---|
| 描述 | 用户进行场景创建操作 | | |
| 参与者 | 系统用户 | | |
| 触发条件 | 用户进入到场景创建界面 | | |
| 前置条件 | 用户对已拥有设备进行查看、控制的能力 | | |
| 后置条件 | 系统反馈用户执行后的操作 | | |

| 优先级 | 高 |
| --- | --- |
| 正常流程 | 1. 用户进入场景创建页面<br>2. 用户选择创建/更新场景操作 |
| 正常流程 | 3. 用户查看设备列表<br>4. 用户选择对应的设备<br>5. 用户选择设备可操作的选项<br>6. 用户配置场景基础信息，如名称等<br>7. 用户点击确认按钮，提交场景创建信息 |
| 扩展流程 | 3a. 设备列表获取失败，系统反馈失败信息<br>5a. 所选设备无可操作选项，系统反馈失败信息<br>7a. 提交状态未完成，系统反馈无法创建 |
| 特殊需求 | 检查待创建的场景的名称，避免场景名称重复 |

用户在场景列表页面中可执行相应的场景，点击对应的执行按钮，即可运行指定场景。通过执行一次用户创建的场景，即可实现场景内所有设备的联动，联动顺序由用户创建场景时添加的控制选项顺序决定。场景执行成功后，用户可以通过查看对应设备状态判断场景执行结果。不仅如此，还可以可以通过系统提示，取消某场景的执行，即停止场景内设备的运行状态，或删除某场景。场景控制用例表述如表3.12所示。

表 3.12: 场景控制用例描述

| ID | UC05 | 名称 | 场景执行 |
| --- | --- | --- | --- |
| 描述 | 用户控制已创建的场景 | | |
| 参与者 | 系统用户 | | |
| 触发条件 | 用户进入到场景列表 | | |
| 前置条件 | 用户选择场景执行按钮 | | |
| 后置条件 | 系统反馈用户执行后的操作 | | |
| 优先级 | 高 | | |
| 正常流程 | 1. 用户进入到场景列表<br>2. 用户选择场景<br>3. 用户选择场景执行按钮 | | |
| 扩展流程 | 2a. 场景列表获取失败，系统反馈失败信息<br>3a. 场景执行失败，系统反馈测试报告尚未生成 | | |

用户在场景页可以选择查看创建成功的场景列表，长按查看对应的场景数据，包括场景内的温度、湿度、风速、PM2.5 等数值，这些数值是通过对场景内所有设备数据整合分析后得出。同时用户也可以通过该用例实现对场景内单个设备的状态进行查看，判断设备运行状态、获取对应的空气数值等。场景查看用例描述如表3.13所示。

表 3.13: 场景查看用例描述

| ID | UC06 | 名称 | 查看场景信息 |
|---|---|---|---|
| 描述 | 用户查看场景信息 | | |
| 参与者 | 系统用户 | | |
| 触发条件 | 用户进入场景界面 | | |
| 前置条件 | 用户需要查看指定场景的场景信息 | | |
| 后置条件 | 无 | | |
| 优先级 | 高 | | |
| 正常流程 | 1. 用户进入场景列表界面<br>2. 用户选择需要查看的场景<br>3. 用户长按对应场景的查看按钮<br>4. 系统跳转到对应场景信息的详情页 | | |
| 扩展流程 | 1a. 场景列表获取失败，系统反馈获取失败<br>4a. 点击场景内设备跳转至设备详情页 | | |

消息推送是由云端发送消息，接收对象包括设备、用户等对象。对用户来说，用户在设备配网并关联设备后可以自行选择是否开启消息提醒功能，可自行取消。如果后端发现设备有异常信息，例如过滤网需清洁、滤芯需更换、室内污染物过多时，也可选择是否推送通知到用户终端。对场景来说，用户创建完场景并添加好对应的设备及操作后，便可以通过对单个场景的执行实现对场景内多个设备的联动控制，这一过程同样是以消息的方式实现的，后端收到用户执行场景的请求后，便会通过异步消息的方式发送对应的指令到智能设备上，消息推送用例描述如表3.14所示。

表 3.14: 消息推送用例描述

| ID | UC07 | 名称 | 消息推送 |
|---|---|---|---|
| 描述 | 消息推送 | | |
| 参与者 | 系统用户 | | |
| 触发条件 | 系统用户打开消息推送开关 | | |

| | |
|---|---|
| 前置条件 | 无 |
| 后置条件 | 无 |
| 优先级 | 中 |
| 正常流程 | 1. 用户进入消息推送页面 |
| 正常流程 | 2. 用户打开/关闭消息推送提醒开关<br>3. 用户退出消息推送界面 |
| 扩展流程 | 2a. 系统请求失败，系统反馈操作失败 |

日志查看包含用户行为日志和设备操作指令日志等。对用户来说，用户在进入客户端后，访问日志模块，即可查看自己所拥有的设备的操作和运行日志，包括设备何时启动、何时更该模式等指标。在查看日志信息时，用户可以访问指定时间内的日志数据，并通过上拉加载更多数据的方式获取更多的设备日志。这一部分由日志处理功能模块提供能力，在用户进行一系列操作时会异步的创建日志信息体并将日志信息发送到日志处理模块，该模块接收到数据后按照格式整理好存入对应的数据库中，日志查看用例描述如表3.15所示。

表 3.15: 日志查看用例描述

| ID | UC07 | 名称 | 消息推送 |
|---|---|---|---|
| 描述 | 消息推送 | | |
| 参与者 | 系统用户 | | |
| 触发条件 | 系统用户进入日志查看模块 | | |
| 前置条件 | 无 | | |
| 后置条件 | 无 | | |
| 优先级 | 中 | | |
| 正常流程 | 1. 用户进入日志查看页面<br>2. 用户上滑即可查看设备日志，当滑至页面底部时，上拉即可<br>3. 用户退出日志查看界面 | | |
| 扩展流程 | 2a. 系统请求失败，系统反馈操作失败 | | |

## 3.3　场景构建技术系统设计

### 3.3.1　系统架构设计

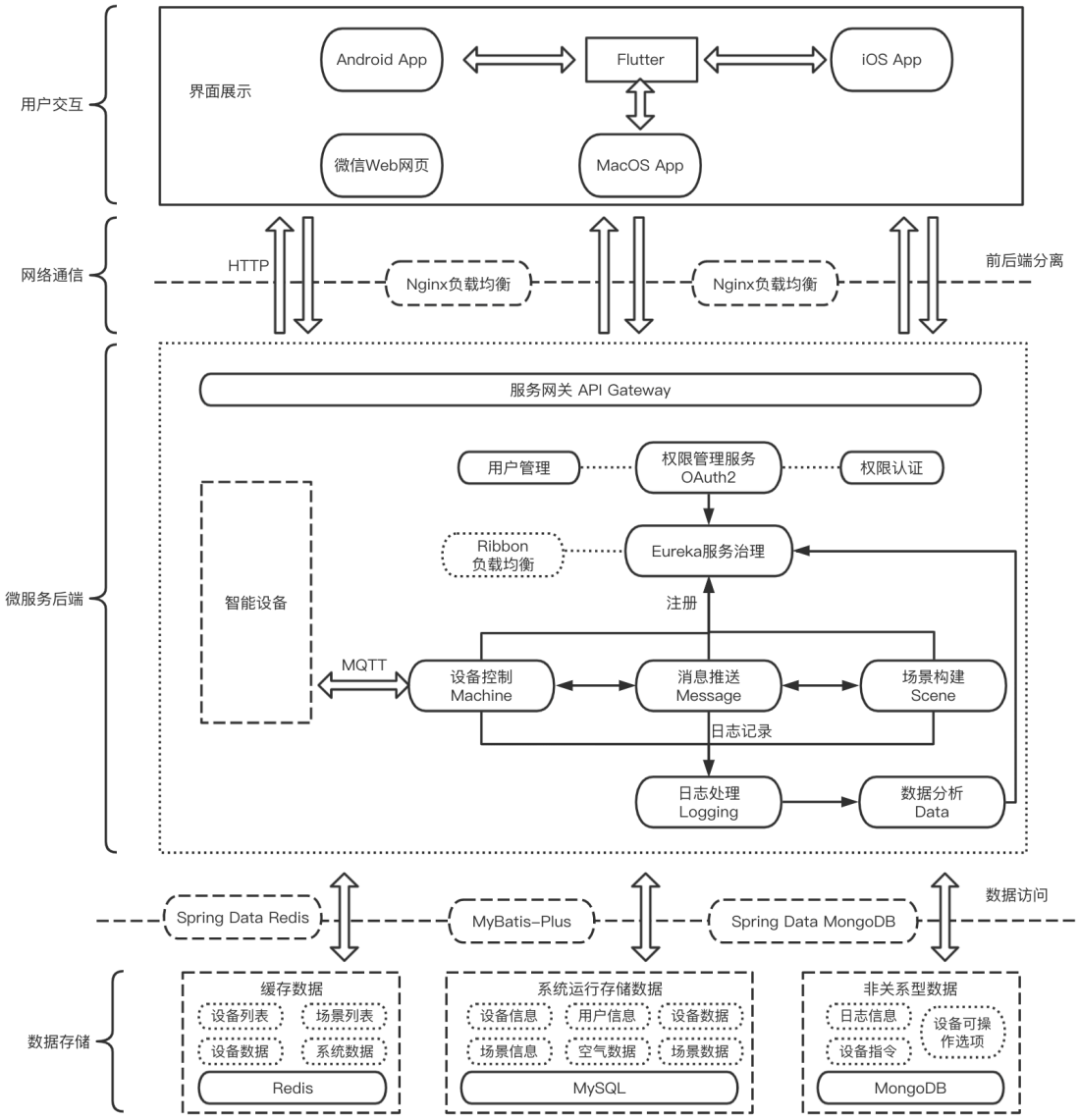


图 3.3: 场景构建技术框架图

本系统为 C/S 架构，并结合前后端分离的开发思想，其中前端采用 Flutter 框架实现，依靠该框架，实现了 Android、iOS、MacOS 等平台的应用开发，实现了近乎统一的用户交互体验。后端基于网关对外提供 RESTful API，并通过微服务架构将系统分为多个独立的微服务模块，每个模块单独使用 Spring Boot 快速开发。如图3.3所示，面向智能新风设备的场景构建系统自底向上，可以分为

数据存储、微服务后端、网络通信和用户交互。

后端架构分为权限管理、设备控制、消息推送、场景构建、日志处理和数据分析六个微服务。在该系统中，通过权限管理微服务来进行用户管理和权限认证，该模块采用 Spring Security 框架实现 OAuth2.0 方式鉴权，确保服务的安全性，与系统功能的权限管理模块相对应；设备控制微服务负责实现对设备进行远程操作，该部分以 MQTT 为通信协议，实现对设备诸如开关、风速调整等操作，设备控制指令与消息推送模块交互，使其转为通过 HTTP 协议进行控制，与系统功能的设备控制模块相对应。消息推送服务，以 RocketMQ 为核心，实现了异步消息推送机制，通过确定的 topic 和 tag 实现消息的发送与接收，场景的执行与取消同样依赖该部分。与系统功能的消息推送模块相对应。场景构建服务，主要实现场景的增删改查，并通过消息推送模块实现与多个设备间的异步通信。微服务后端部分通过不同数据访问框架实现与不同数据库之间的对接。

数据层主要负责整个系统的数据存储与缓存构建。在数据层中，主要利用 MySQL、MongoDB 和 Redis 实现数据的存储与缓存。由于用户创建的场景数据结构化程度相对固定且使用频率较高，以此采用 MySQL 这一关系型数据库存储，并为了加快场景数据的获取，利用 Redis 实现数据缓存。设备在运行过程中会产生许多数据，这种数据的写入频率高、量大且结构化较弱，因此使用 MongoDB 存储。

### 3.3.2　4+1 视图模型

在软件工程领域，为了对系统进行一个抽象的描述，通常会使用一种名为“4+1”视图 [43] 模型的方法。这种方法主张将软件系统架构分为五个组成部分，分别是逻辑视图、开发视图、进程视图、物理视图以及场景视图。如本文 3.3.3 小节中的逻辑视图，就将本系统的功能实现了逻辑上的抽象分解；如 3.3.4 小节中的前后端结构图以及 3.3.7 小节的 E-R 图就很好展示出本系统的内部开发需求；如 3.3.5 小节中时序图，展示了整个系统的运行特性和部分功能的执行流程；物理视图通常用于描述系统部署安装、系统组件通信等问题，如 3.3.6 小节的系统部署图展示了将软件的不同部分映射到不同的节点。场景视图，如 3.2.3 小节的系统用例图将前四个视图有机联系起来，是重要系统活动的抽象。

### 3.3.3　系统逻辑设计

依据上文的系统架构与用例设计，系统的逻辑视图如图3.4所示：

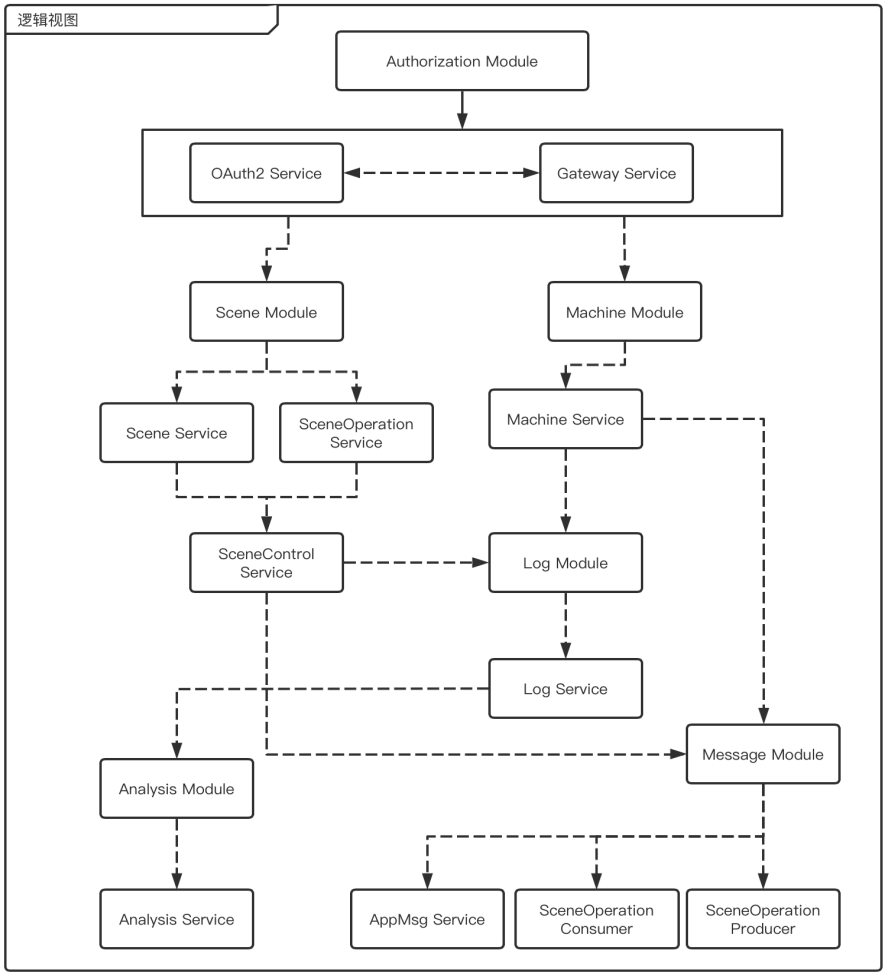


图 3.4: 场景构建技术系统逻辑视图

系统从逻辑层面主要分为鉴权模块（Authorization Module）、设备模块（Machine Module）、消息模块（Message Module）、场景模块（Scene Module）、日志模块（Log Module）和分析模块（Analysis Module）。鉴权模块负责整个系统的用户鉴权授权任务，本系统仅支持登录过系统的用户使用，具体通过网关服务和 OAuth2 服务处理请求。设备模块服务实现用户对设备的控制管理；场景模块负责场景的增删改，场景创建由 Scene Service 实现，还可以通过该 Service 获取场景的基本信息，如场景名称、场景内空气数据等信息。场景内设备的添加、设备的操作行为由 SceneOperation Service 设置。执行场景的动作由 SceneControl Service 完成，通过消息队列将消息发送给 Message Module 实现任务的调度；Message Module 负责整个技术中的消息通信，模块之间的通信通过 SceneOperation Producer 发送数据，并由 SceneOperation Consumer 接收后执行相关的操作，例如远程给设备发送控制指令等。额外的，通过 AppMsg Service 实现对 App 消息的远程推送，

例如每天的天气情况、使用建议等内容；Log Module 用于记录 Machine Module 和 Scene Module 运行过程中产生的日志消息；Analysis Module 模块通过对 Log Module 中保存的日志分析，生成相关的结果报表。

### 3.3.4　系统开发设计

本系统的前端采用 Flutter 跨平台框架，该部分主要由 main、ui、model、router、service、controller、util、constant 等模块组成，如图3.5所示。

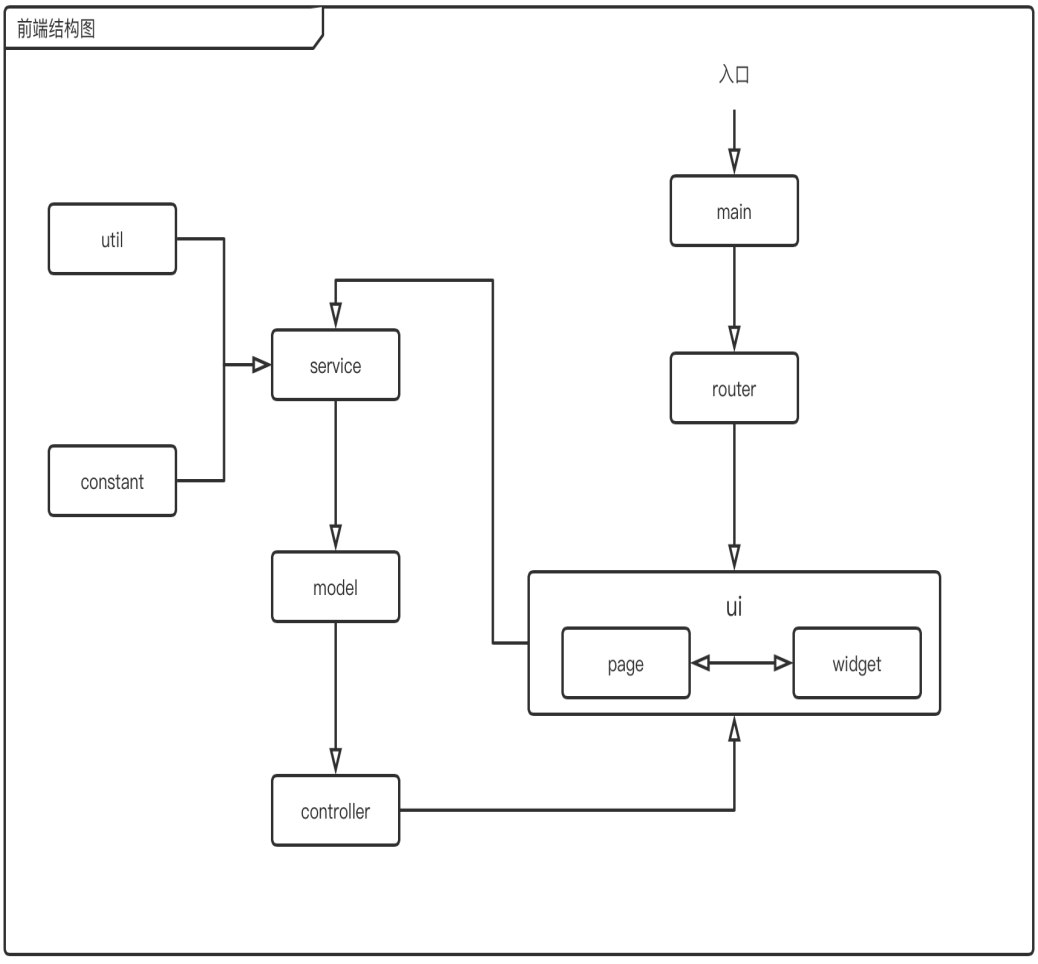


图 3.5: 场景构建技术前端结构图

在 Flutter 框架中，默认 main.dart 为整个项目的入口，程序启动时会首先渲染 main.dart 文件。在 Flutter 中，一切都是 Widget，ui 文件夹存储前端页面文件，包括 page 和 widget，即页面文件和组件文件。widget 包含应用程序中使用到的第三方组件，如空气图标、操作导航栏、菜单栏等组件。前端页面通过 dart 语言

组织，并利用声明式 UI 的方式描述用户界面。router 模块统一管理前端项目的路由，通过管理路径，实现不同页面之间的切换、跳转以及状态的变化。

util 文件夹用于存放封装的工具函数，供系统内相关模式使用，节约代码行数，提高复用性，如数据操作、网络请求等。constant 模块用于存储在页面渲染时所需要的常量资源，例如请求接口列表，相关事件名称，存储键值等信息。model 模块用于序列化和反序列化页面渲染所需要的数据，通过 service 模块获取到 json 数据后需要按照 model 模块中的文件的要求序列化成能被直接使用的对象。controller 模块为前端提供了一套整体的状态信息表示的功能，通过该模块实现对应用程序的数据状态控制、监听等功能。

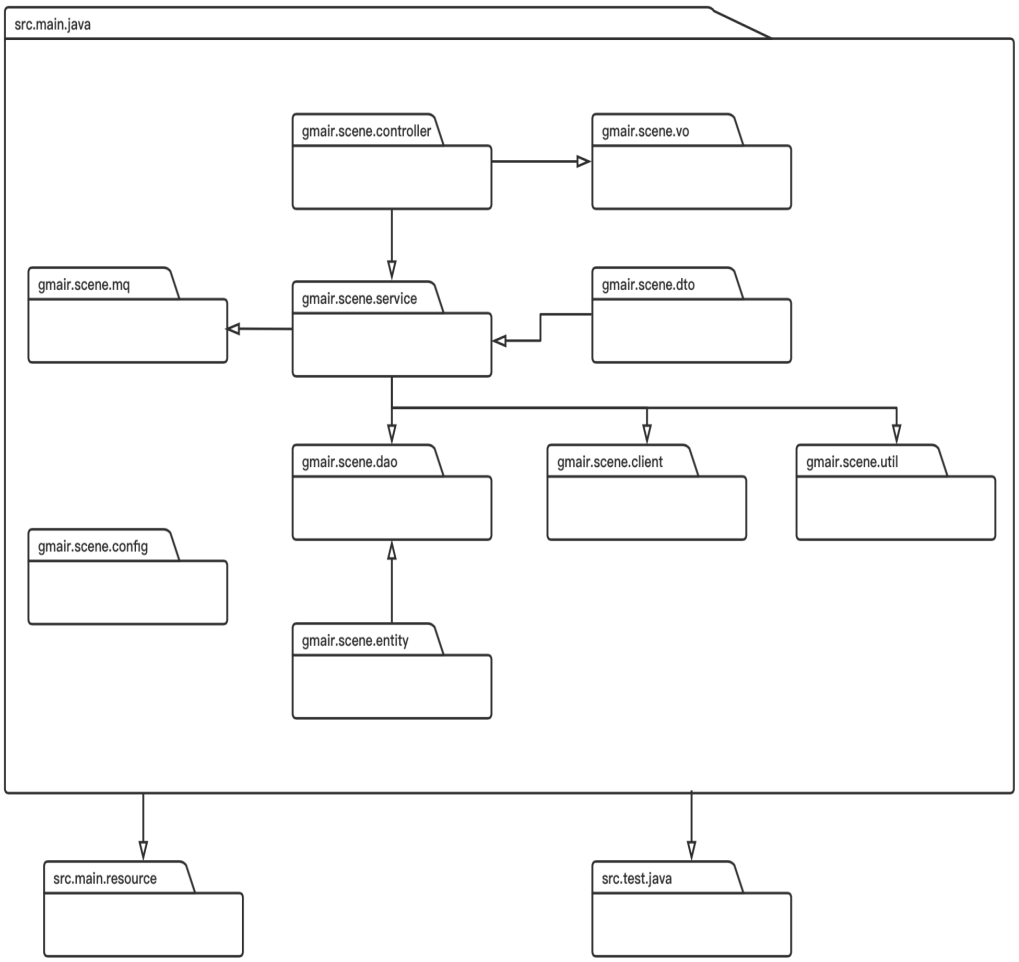


图 3.6: 场景构建技术系统后端结构图

本项目后端结构如图3.6所示，service 包内文件负责业务逻辑的接口定义和代码的具体实现，controller 包用于指定对外暴露的 RESTful 接口以及请求的流程控制。dao 包用于定义系统服务和数据库之间连接，系统使用 Mybatis-Plus 进行与 MySQL 数据库中数据的增删改查操作。使用 Spring Data MongoDB 实现与

MongoDB 数据库内数据的交互。client 包用于申明各个微服务指阿金的接口，该包内的接口通过 @FeignClient 注解实现微服务的远程调用。

mq 包中对应的是通过消息队列进行异步通信所需要执行的代码，通过由 service 层引入 producer 并创建消息、发送后由 consumer 进行消费，进而进行后端的逻辑处理。dto 包用于指定各类在系统中流通的数据结构体，在一定程度上可以提高数据传递的安全性。和前端架构类似，后端模块中的 utils 也用于存放系统通用的封装好的工具函数，例如序列化与反序列化、文件加密、日期转换等。resources 中包含项目启动和运行时所需要的静态配置信息文件，例如服务启动的端口、数据库连接的地址与密码等。

### 3.3.5　系统进程设计

根据上述需求及设计，本节从设备控制和场景构建来分析介绍场景构建系统的进程设计。

场景构建技术的设备管理进程视图3.7所示，系统用户主要进行设备添加和设备控制的操作。用户进行设备添加操作时，首先需要先访问登陆界面，随后经由鉴权系统获取验证码，用户登录后便会从鉴权系统中获得相应的 token 作为后续接口访问的唯一凭证。登录成功后便会由用户服务获取到用户信息，经过上述步骤后，用户便可进行设备添加操作。

用户添加设备的第一步便会扫描设备自身携带的二维码，该二维码是设备出库时已经记录完成的，用户扫描二维码结束后便会进入配网页面，当配网完成后，用户便会进入设备初始化界面，在初始化阶段，用户需要设置机器的名称，当提交相关的信息后，便会在设备表中新增相应的记录，当完成上述操作后，设备初始化过程便会结束。当用户控制设备时，首先需要选择需要控制的设备，然后根据应用程序界面上的提示执行相应的操作，系统会根据用户的操作对设备发送相对应的操作指令，随后返回设备的执行结果。

场景构建技术的场景管理进程视图3.8所示，系统用户主要进行场景创建和场景控制的操作。用户创建场景前的鉴权操作和上述一致，这里不再赘述。当用户创建场景时需要先获取设备列表，在完成设备的选择后便会获取该设备可以执行的操作列表，随后添加相应的设备和操作，完成后将会把场景信息写入场景数据库中。当执行场景时，用户只需点击相应的按钮，剩下的指令操作便会由后端服务器远程发送到相应的设备上，随后返回场景的执行结果。

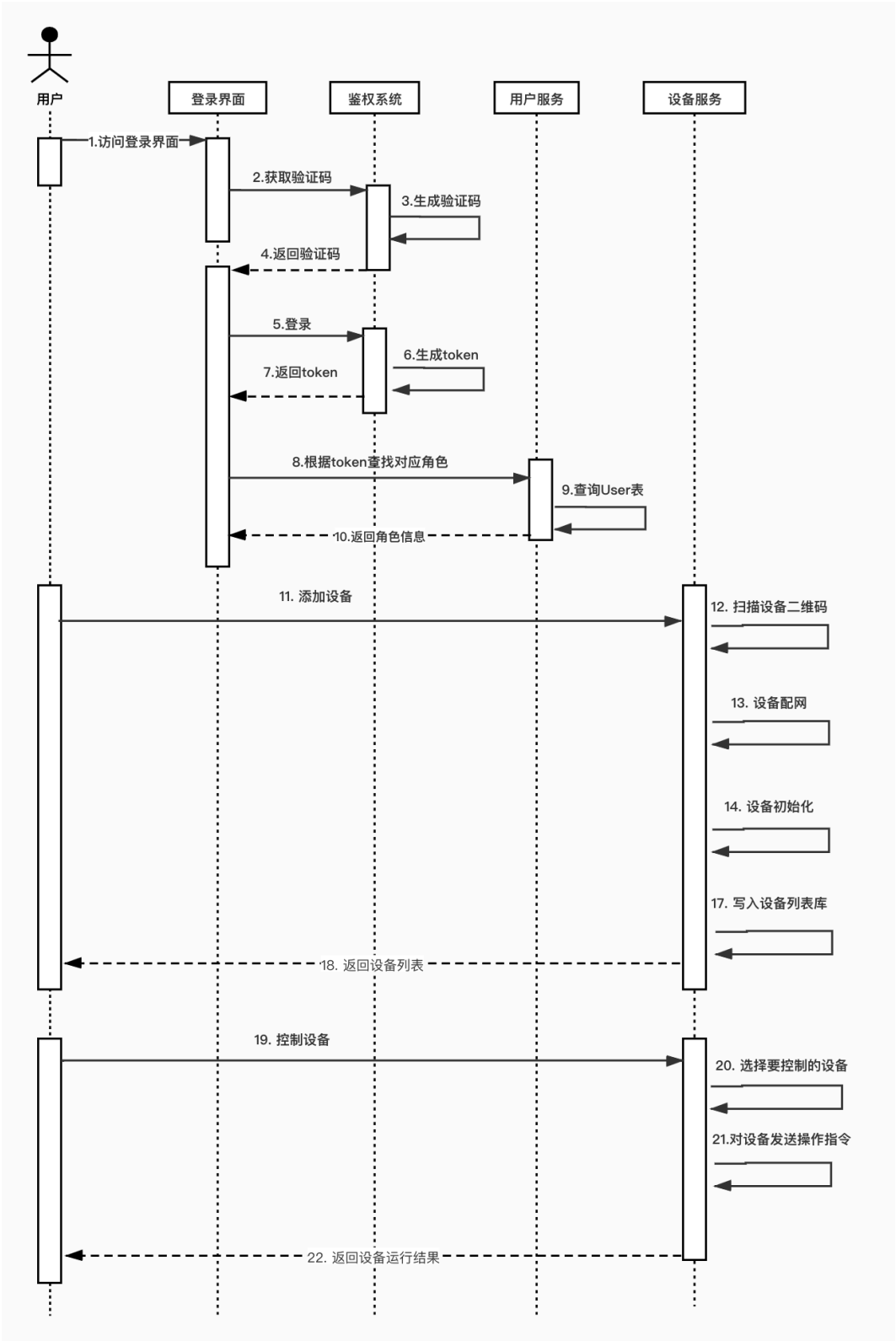


图 3.7: 设备管理进程视图

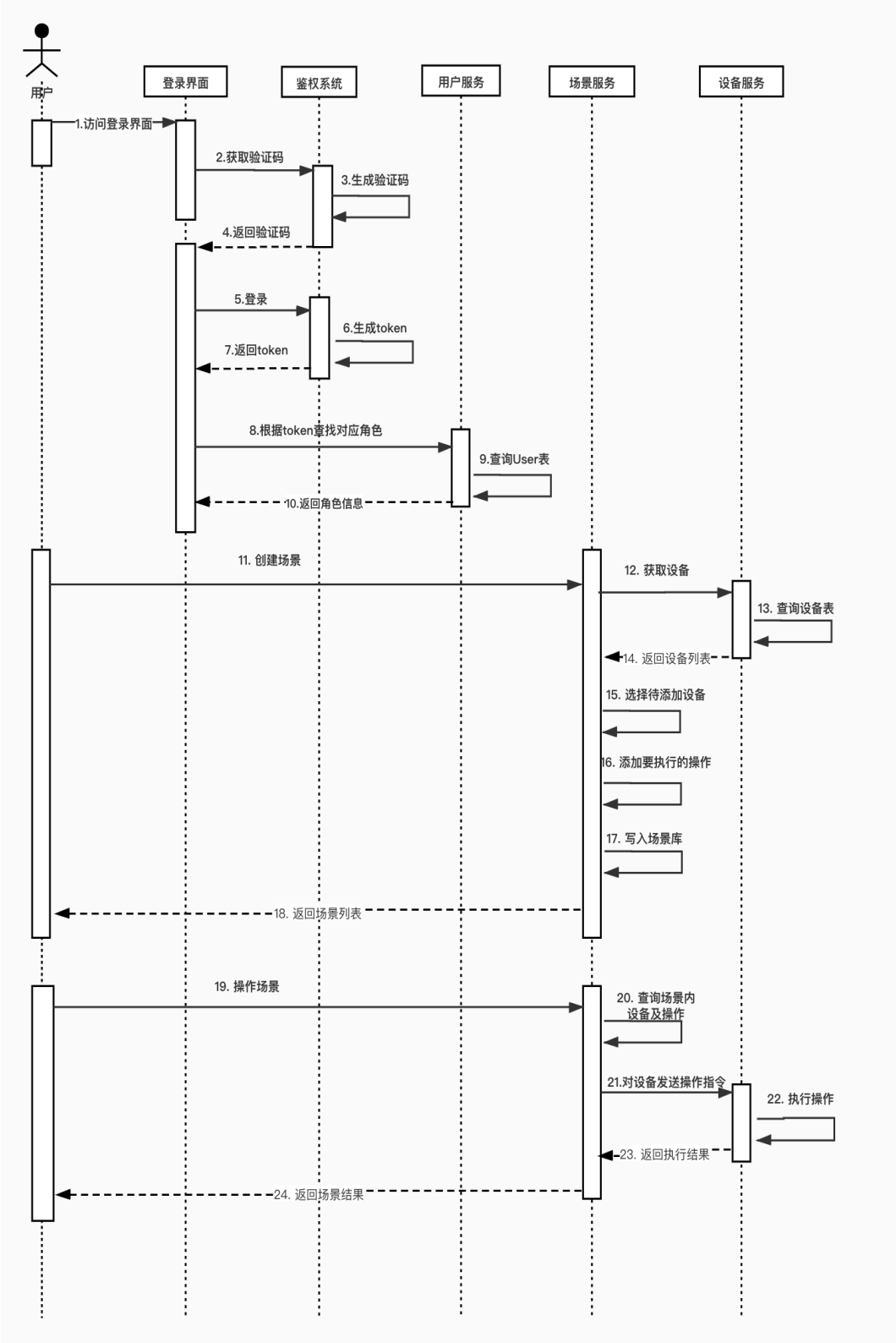


图 3.8: 场景构建管理进程视图

### 3.3.6　系统部署设计

如图3.9所示，该系统将前后端项目部署在不同的地方，后端通过微服务架构将系统划分为多个模块，分别部署在不同的节点，并最终利用 Spring Cloud 将多个功能模块整合通信。每个微服务各自部署，并根据项目的实际情况动态调整环境资源，以此提高整个系统的可用性。同时抽象出数据节点，无需考虑每个微服务模块如何访问数据库，实现了数据的安全访问，简化了操作流程，优化了开发效率。前端项目由 Flutter 框架实现，并针对不同的平台生成不同的安装文件，最后运行至用户的设备上。

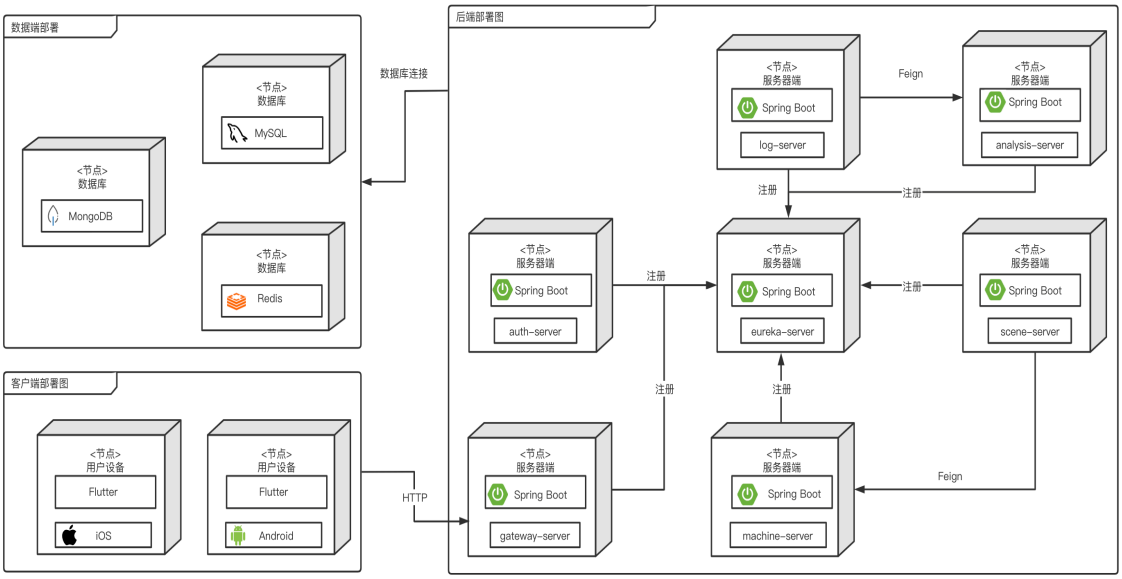


图 3.9: 场景构建技术系统部署图

### 3.3.7　系统数据实体设计

本系统的核心实体关系图如图3.10所示，将场景设计为一个实体，并与场景使用用户、设备、场景控制选项相关联，用户在前端创建场景后，后端会对传输的数据解析，校验成功后，数据库中将会新增对应的场景记录。在创建场景记录成功后，后端会通过场景内的设备获取分析出场景内的数值详情。同样的，设备也被设计成一个数据实体，并和智能终端设备一一对应。

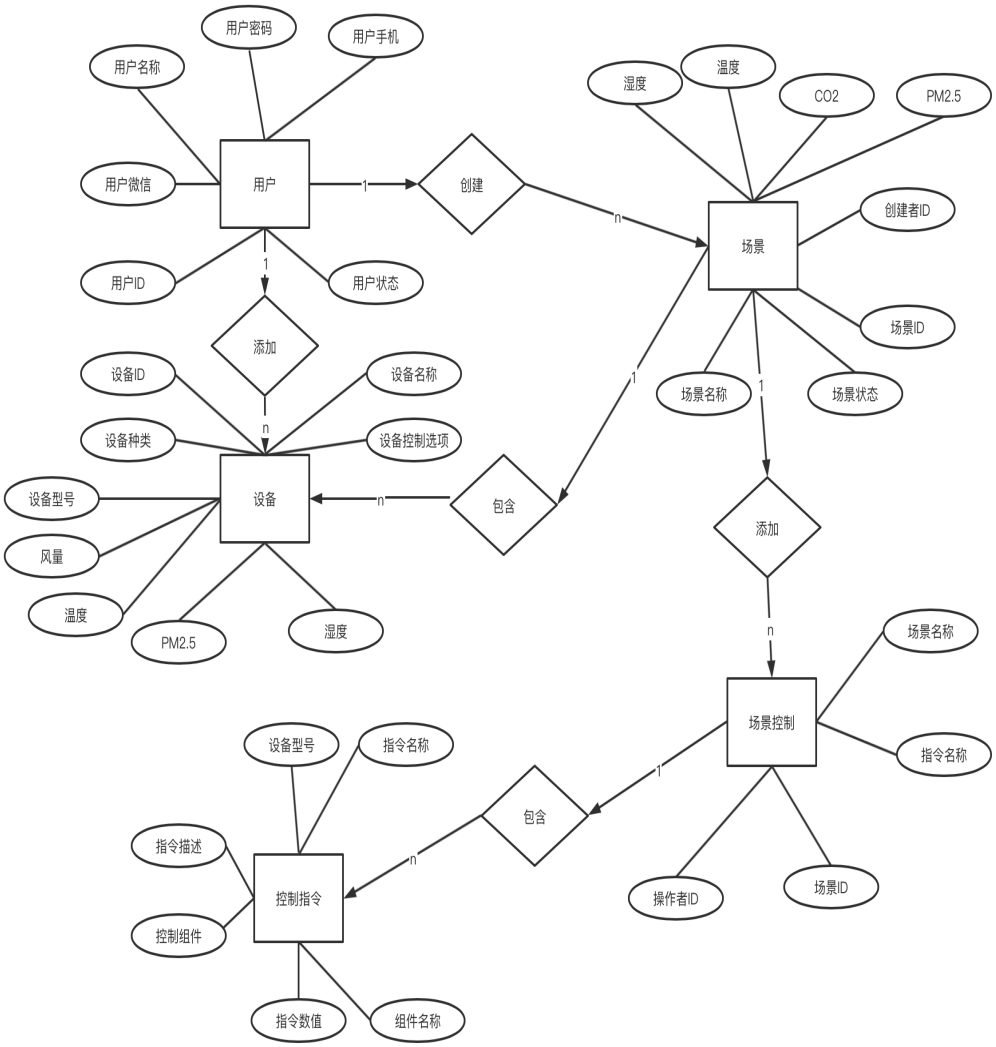


图 3.10: 场景构建技术核心实体关系图

一般而言，一个用户可以创建多个场景，也可以添加多个设备，也可在一个场景中添加多个设备及控制选项。控制选项实体主要包括，用户可进行操作的设备实体及设备对应的额控制指令。本节主要对六个模块所涉及到的数据实体进行详细的表结构设计，设备控制模块涉及到设备记录实体 MachineLoc，设备控制选项实体 ControlOptionAction，分别对应数据库中的 consumer_code_machine_view 视图和 control_option_action 表，数据库表字段设计分别如表3.16和表3.17所示。

在表3.16中，系统利用 bind_id 作为设备与用户相关联的唯一标识，用户在绑定设备时还需要设置相对应的名称，对应的数据库字段为 bind_name，在绑定设备时还需要获取设备的 ID 和设备对应二维码值，对应的字段为 machine_id 和 code_value；在使用时亦会记录设备所处的位置，即省份、城市和其经纬度。

表 3.16: consumer_code_machine_view 视图字段详细设计

| 字段 | 含义 | 类型 | 描述 |
| --- | --- | --- | --- |
| bind_id | 设备关联 id | varchar | 设备关联主键 |
| consumer_id | 系统用户 id | varchar | 系统用户主键 |
| bind_name | 设备关联名称 | varchar | 系统用户绑定设备的名称 |
| code_value | 设备二维码 | varchar | 设备二维码 |
| machine_id | 设备 ID | varchar | 设备 ID |
| province_id | 设备所在省份 ID | varchar | 设备所在省份 ID |
| city_id | 设备所在城市 ID | varchar | 设备所在城市 ID |
| city_name | 设备所载城市名称 | varchar | 设备所在城市名称 |
| longitude | 设备位置经度 | double | 设备所在位置经度 |
| latitude | 设备位置纬度 | double | 设备所在位置纬度 |
| block_flag | 逻辑删除标志 | tinyint | 逻辑删除标志，1 为删除，0 为为删除 |
| create_time | 设备添加时间 | datetime | 设备添加时间 |

表 3.17: control_option_action 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
| --- | --- | --- | --- |
| value_id | 主键 | varchar | 控制选项主键 |
| control_id | 控制行为 ID | varchar | 控制行为对应的 ID |
| model_id | 模式 ID | varchar | 可控制设备模式的 ID |
| action_name | 可控制动作名称 | varchar | 设备可控制动作的名称 |
| action_operator | 可控制组件 | varchar | 控制组件名称 |
| command_value | 控制值 | int | 设备控制的值 |
| block_flag | 控制选项是否删除 | tinyint | 逻辑删除标识 |
| create_time | 创建时间 | datetime | 记录创建时间 |

消息推送模块中，服务与服务之间的通信不需要通过数据表的方式连接，但是在实现 APP 推送功能时，我们需要获取用户自身的推送状态，即用户是否开启消息推送功能，该部分对应的记录实体为 UserMsgPush，与之对应的数据表为 user_msg_push，该表的设计字段如表3.18所示。

表 3.18: user_msg_push 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
|---|---|---|---|
| id | 主键 | varchar | 记录主键 |
| consumer_id | 用户 ID | varchar | 待推送用户 ID |
| status | 用户推送状态 | varchar | 用户是否打开推送开关 |
| create_time | 创建时间 | datetime | 记录创建时间 |
| update_time | 更新时间 | datetime | 记录修改时间 |
| block_flag | 记录是否删除 | tinyint | 逻辑删除标识 |

status 字段表示用户是否打开了消息推送的开关，用户在前端更改状态后便会给后端发送相应的请求用于更改该字段。在执行推送操作时便会读取该用户的该条记录，如果该字段结果为 true，则推送相关信息；反之，则不推送。

场景构建模块中,我们需要存储用户创建场景的信息(对应实体为 Scene)、场景内所包含的设备及可操作的指令（对应实体为 SceneOperation）以及可操作选项(对应实体为 SceneCommands), 分别对应数据库中的 scene 表、scene_operation 表和 scene_commands 表, 数据库表字段设计分别如表3.19、表3.20和表3.22所示。

表 3.19: scene 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
|---|---|---|---|
| scene_id | 场景 ID | bigint | 场景 ID 作为记录主键 |
| consumer_id | 用户 ID | varchar | 场景创建者用户 ID |
| name | 场景名称 | varchar | 场景名称 |
| pm25 | 场景内 PM2.5 值 | double | 场景内 PM2.5 值 |
| temperature | 场景的温度值 | double | 场景的温度值 |
| humidity | 场景的湿度值 | double | 场景的湿度值 |
| co2 | 场景内的 CO2 浓度 | double | 场景内的 CO2 浓度 |
| status | 场景状态 | varchar | 场景状态 |
| create_time | 创建时间 | datetime | 记录创建时间 |
| update_time | 更新时间 | datetime | 记录修改时间 |
| deleted | 记录是否删除 | tinyint | 逻辑删除标识 |

用户在前端创建场景后，会执行一步获取场景信息的操作，用户可获取的信息主要包括名称和相应的空气数值，如 CO2、PM2.5 和温湿度等。这些数值由场景内的设备数据整合而成，因此我们在创建场景时还需添加对应的设备及操作。

考虑到一个场景内可能包含多个操作，为方便存储，将该数据存储在 MongoDB 中，如表3.20所示。

表 3.20: scene_operation 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
|---|---|---|---|
| scene_id | 场景 ID | bigint | 和 scene 表关联的主键 |
| scene_name | 场景名称 | varchar | 场景名称 |
| consumer_id | 用户 ID | varchar | 场景创建者用户 ID |
| commands | 控制指令 | Array | 场景执行的操作列表 |
| create_time | 创建时间 | datetime | 记录创建时间 |
| update_time | 更新时间 | datetime | 记录修改时间 |
| deleted | 记录是否删除 | tinyint | 逻辑删除标识 |

在用户创建完场景记录后，会生成一个 scene_id 作为场景的唯一标识，在添加场景内的操作时便会使用该标识，同时记录场景的名称和操作者 ID，在该结构中，场景内的操作数据为 commands 字段，该字段是一个数组，用于记录场景内的操作选项，如设备开关和设备某一模块的具体数值。Commands 中每个实体数据对应的数据结构如表3.21所示。

表 3.21: commands 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
|---|---|---|---|
| cid | 指令 ID | bigint | 指令 ID，和 scene_commands 表关联的主键 |
| qr_code | 设备二维码 | varchar | 设备二维码，即指令操作的具体对象 |
| device_name | 设备别名 | varchar | 用户添加设备时设置的别名 |
| seq | 指令顺序 | varchar | 执行场景内的操作时需要遵循的顺序 |
| c_name | 指令名称 | varchar | 指令名称 |
| c_component | 指令控制组件 | varchar | 指令控制组件 |
| c_operation | 指令控制数值 | varchar | 远程控制命令的具体数值 |

用户在前端创建场景时需要添加相应的设备操作，需要指定的数据为设备二维码（用于表明需要控制的是哪一个设备），指令控制的设备的组件和具体数值，以及指令执行的次序。因为在一个场景中可能包含多个控制指令，因此控制

顺序就变得尤其重要，这里我们以 seq 字段作为指令次序的标识。其中可添加的控制选项的实体为 SceneCommands，对应的数据表为 scene_commands 表。

表 3.22: scene_commands 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
| --- | --- | --- | --- |
| id | 记录 ID | bigint | 主键 |
| goods_id | 设备型号 | varchar | 设备型号 |
| command_name | 指令名称 | varchar | 指令名称 |
| command_component | 组件名称 | varchar | 组件名称（英文），用于发送指令 |
| command_operation | 执行动作 | varchar | 执行动作的具体数值，如 on、off 等 |
| command_description | 指令描述 | varchar | 指令的功能描述 |
| component_name | 组件名称（中文） | varchar | 组件名称（中文），用于前端展示 |
| enable | 指令否删是否可用 | tinyint | 逻辑删除标识 |

日志模块记录了用户在使用该系统时的情况，如用户账户行为日志、用户对设备控制的行为日志等。用户账户行为主要包括用户绑定、解绑微信，修改用户名、修改用户地址等。用户对设备的控制行为主要包括设备的绑定、解绑，设备开关，设备温湿度、风量等的控制。由于该模块的数据结构性较弱，为了存储方便和数据分析，这里选择使用 mongodb 作为存储数据库，分别对应数据库中的 user_account_operation_log 数据表和 user_machine_operation_log 数据表。数据库表字段设计分别如表3.23和表3.24所示。

表 3.23: user_account_operation_log 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
| --- | --- | --- | --- |
| logId | 记录 ID | bigint | 主键 |
| userId | 执行人 | varchar | 操作执行人 |
| ip | 请求 ip | varchar | 用户发送请求的 IP 地址 |
| createAt | 创建时间 | varchar | 记录创建的时间 |
| component | 操作模块 | varchar | 用户操作的模块 |
| detail | 操作详情 | varchar | 用户执行的操作的详细信息 |

为了记录用户的行为数据，及用户对设备的控制，我们需要保存以下信息，分别是设备二维码、执行人、执行组件和具体的操作值，并以日志的形式保存在数据库中，方便数据分析模块整理汇总并分析。

表 3.24: user_machine_operation_log 表字段详细设计

| 字段 | 含义 | 类型 | 描述 |
| --- | --- | --- | --- |
| logId | 记录 ID | bigint | 主键 |
| userId | 执行人 | varchar | 操作执行人 |
| qrcode | 设备二维码 | varchar | 设备二维码 |
| time | 执行时间 | varchar | 指令执行时间 |
| component | 执行组件 | varchar | 指令执行组件 |
| value | 操作值 | varchar | 指令执行的具体数值 |

## 3.4　本章小结

本章首先对面向智能新风设备的场景构建系统进行总体规划，并根据实际应用场景分析得出该系统的功能需求与非功能性需求。在系统设计过程中，以“4+1”视图为指导，对本系统进行了多方面的设计，并按照功能将该系统分为权限管理模块、消息推送模块、设备控制模块、场景构建模块、日志处理模块和数据分析模块 6 个部分。

# 第四章　智能新风设备场景构建系统的详细设计与实现

本章将会详细的介绍面向智能新风设备的场景构建系统的实现过程，结合第三章所做的需求分析与设计，本系统从逻辑上分为权限管理、设备控制、消息推送、场景构建、日志处理和数据分析六个模块，本章将会对这 6 个模块的设计与实现进行阐述。

## 4.1　权限管理模块设计与实现

### 4.1.1　权限管理模块的详细设计

权限管理模块的详细设计类图如图4.1所示，该模块负责实现用户登录已经请求鉴权，在该模块中，结合业务 ConsumerService 进行统一的用户状态管理，适配接入的用户认证操作，通过 OAuth2ServerConfig、RedisConfig、SecurityConfig 将认证配置应用于服务中。同时利用网关服务对一些恶意请求实现拦截处理，以确保整个系统的安全与稳定。

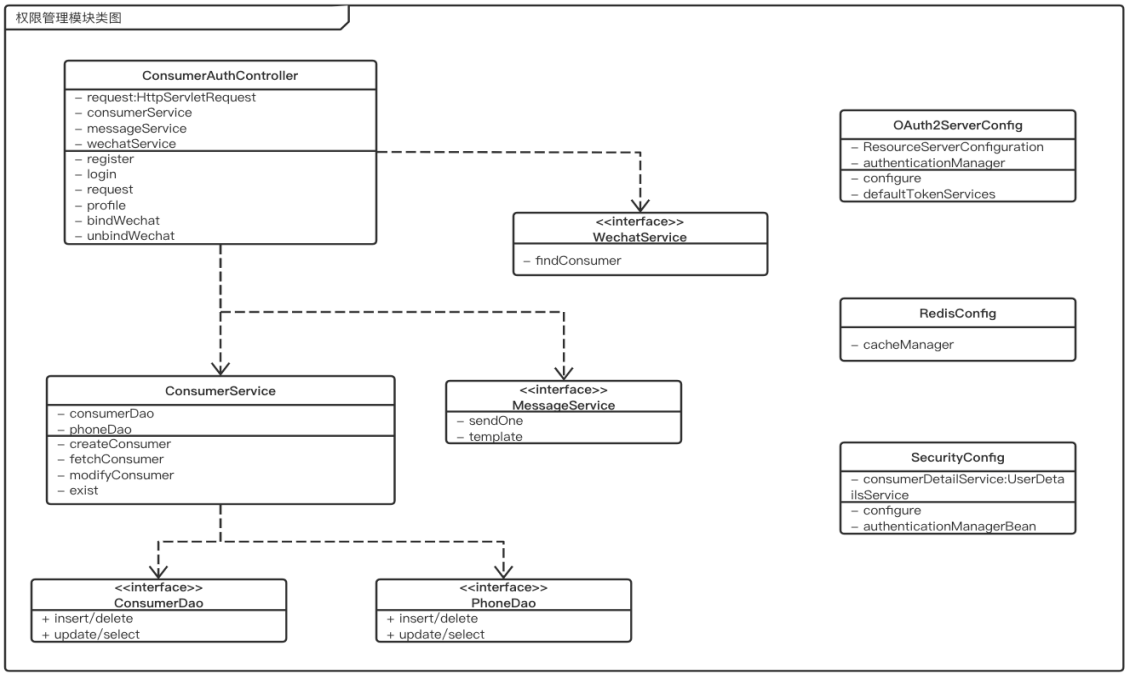


图 4.1: 权限管理模块类图

### 4.1.2　身份认证的实现

为了正确使用面向智能新风设备的场景构建系统，用户需通过登录接口获取对应的用户信息，无论是采取验证码登录还是微信登录的手段，都需要将用户登录凭证持久化到对应的数据库中。本系统以 access_token 和 refresh_token 的形式维护用户的登录状态，已通过将 access_token 和 refresh_token 保存在 redis 并设置失效时间的方式实现授权的实效性，以此减轻数据库请求压力，提高认证效率，身份认证关键代码如图4.2所示。

图 4.2: 身份认证关键代码

```
public DefaultTokenServices defaultTokenServices() {
    DefaultTokenServices services = new DefaultTokenServices();
        services.setAccessTokenValiditySeconds(60 * 60 * 48);
    services.setRefreshTokenValiditySeconds(60 * 60 * 168);
    services.setTokenStore(tokenStore());
    return services;
}
public UserDetails loadUserByUsername(String s) {
    List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
    ResultData response = consumerService.fetchConsumer(condition);
    if (response.getResponseCode() == ResponseCode.RESPONSE_OK) {
        ConsumerVo consumerVo = ((List<ConsumerVo>)
            response.getData()).get(0);
        return new User(consumerVo.getConsumerId(),
            serialService.fetch(s).getSerial(), grantedAuthorities);
    } else {
        condition.put("wechat", s);
        response = consumerService.fetchConsumer(condition);
        if (response.getResponseCode() == ResponseCode.RESPONSE_OK) {
            ConsumerVo consumerVo = ((List<ConsumerVo>)
                response.getData()).get(0);
            return new User(consumerVo.getConsumerId(), "",
                grantedAuthorities);
        }
    }
}
```

### 4.1.3　用户微信号绑定的实现

为方便用户登录使用以及多端账号统一，在权限管理模块接入了微信登录。微信登录是基于 OAuth2.0 协议标准构建的微信 OAuth2.0 授权登录系统。这种授权登录方式让微信用户可以使用微信身份安全登录场景构建系统中，在微信用户授权登录已接入微信 OAuth2.0 的场景构建系统后，便可以获取到用户的接口调用凭证（access_token），通过 access_token 可以进行微信开放平台授权关系接口调用，从而可实现获取微信用户基本开放信息等功能。获取微信 access_token 的关键代码如图4.3所示。

图 4.3: 微信账号绑定关键代码

```
public static String queryToken(String appid, String secret) {
    String result = "";
    String url = "xxxxxxx";
    try {
        URL address = new URL(url);
        HttpURLConnection connection =
            address.openConnection();
        connection.setRequestMethod("GET");
        connection.setDoOutput(true);
        connection.setDoInput(true);
        connection.connect();
        InputStream is = connection.getInputStream();
        int size = is.available();
        byte[] bytes = new byte[size];
        String message = new String(bytes, "UTF-8");
        JSONObject object = JSON.parseObject(message);
        result = object.getString("access_token");
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }
    return result;
}
```

## 4.2　设备控制模块设计与实现

### 4.2.1　设备控制模块的详细设计

在设备控制模块中，我们主要围绕两个方面展开，分别是设备的远程控制和设备状态的监控。用户设备如手机、平板等终端设备如果想要和新风设备通信，需要通过云端服务器作为桥梁，而服务器与设备的通信是通过 MQTT 协议进行的，因此设备控制功能需要构建相应的 Restful API 用于用户设备与云端服务器通信。为了对设备的运行状态进行监控，我们需要将设备运行的数据进行存储，通过定时任务获取这些数据已实现对设备运行状态的监控。

如图4.4所示为设备控制模块的相关类图。MachineController 类为整个设备模块的主要入口，负责对外提供接口服务，其余模块若想实现设备控制，需接入该接口。该接口依赖 MachineService 提供服务，并以此实现对设备的开关、属性控制等操作。MachineService 的实现基于 MessageController，通过 Feign 实现对该微服务接口的远程调用，其中 MqttService 类通过接入 MqttClient 实现对 mqtt 消息的订阅与发布。

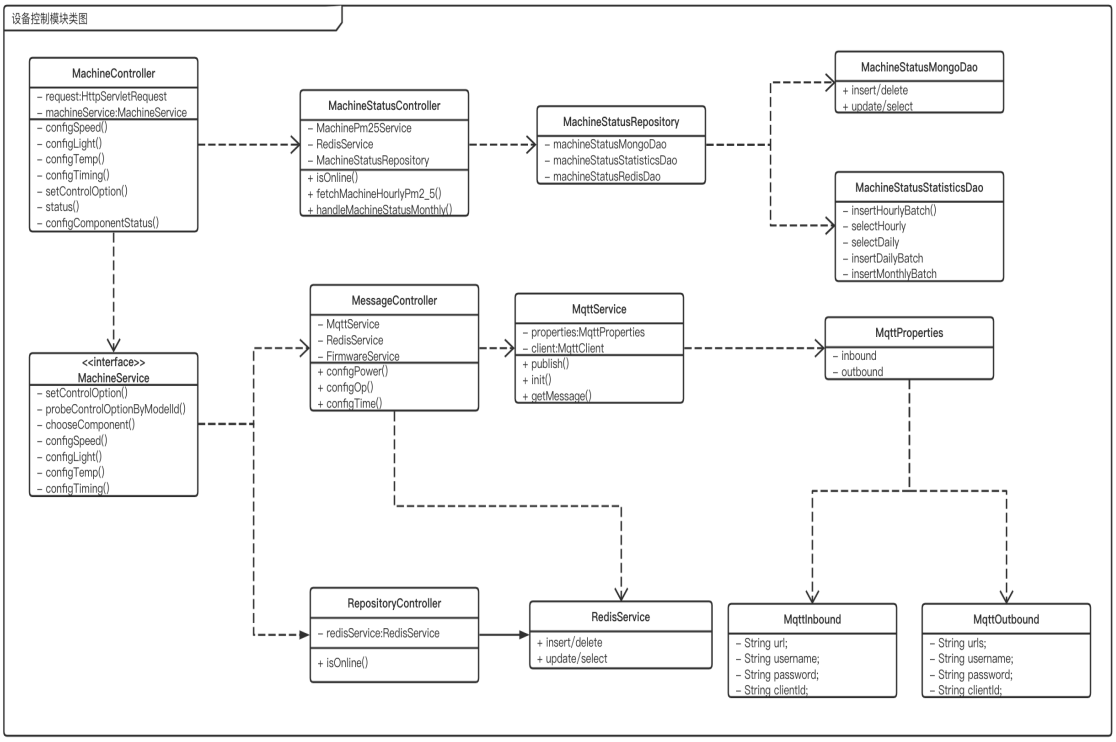


图 4.4: 设备控制模块类图

### 4.2.2　设备远程控制的实现

不同于 web 通信使用 HTTP 协议，设备的远程控制通常会使用 MQTT 协议，在该系统的技术实现中，利用 MqttInbound 实现 MQTT 的消息接收，并配置 MQTT 消息接收的服务器地址等配置。当需要设备上传操作信息时，该模块会利用 MessageService 调用日志处理模块的接口实现日志记录。设备控制的关键代码如图4.5所示，客户端实现设备远程控制界面如图4.6所示。

图 4.5: 设备远程控制关键代码

```
// 发送MQTT指令
try {
    MqttConnectOptions options = new MqttConnectOptions();
    options.setCleanSession(false);
    options.setKeepAliveInterval(30);
    options.setConnectionTimeout(10);
    client.setCallback(new PushCallback());
    client.connect(options);
    MqttMessage message = getMessage(object);
    client.publish(topic, message);
} catch (Exception e) {
    logger.error(e.getMessage());
}
// 设备控制接口调用（以控制风量为例）
@PostMapping("/config/speed")
public ResultData configSpeed(String qrcode, int speed,
    HttpServletRequest request) {
    String consumerId = (String) SecurityContextHolder.getContext()
                    .getAuthentication().getPrincipal();
    ReceptionPool.getLogExecutor().execute(new Thread(() ->
        logService.createUserMachineOperationLog(consumerId, qrcode,
        "speed", new StringBuffer("User ")
    .append(consumerId).append(" operate ").append("speed").append("
        set to ").append(speed).toString(), IPUtil.getIP(request),
        String.valueOf(speed))));
    return machineService.configSpeed(qrcode, speed);
}
```

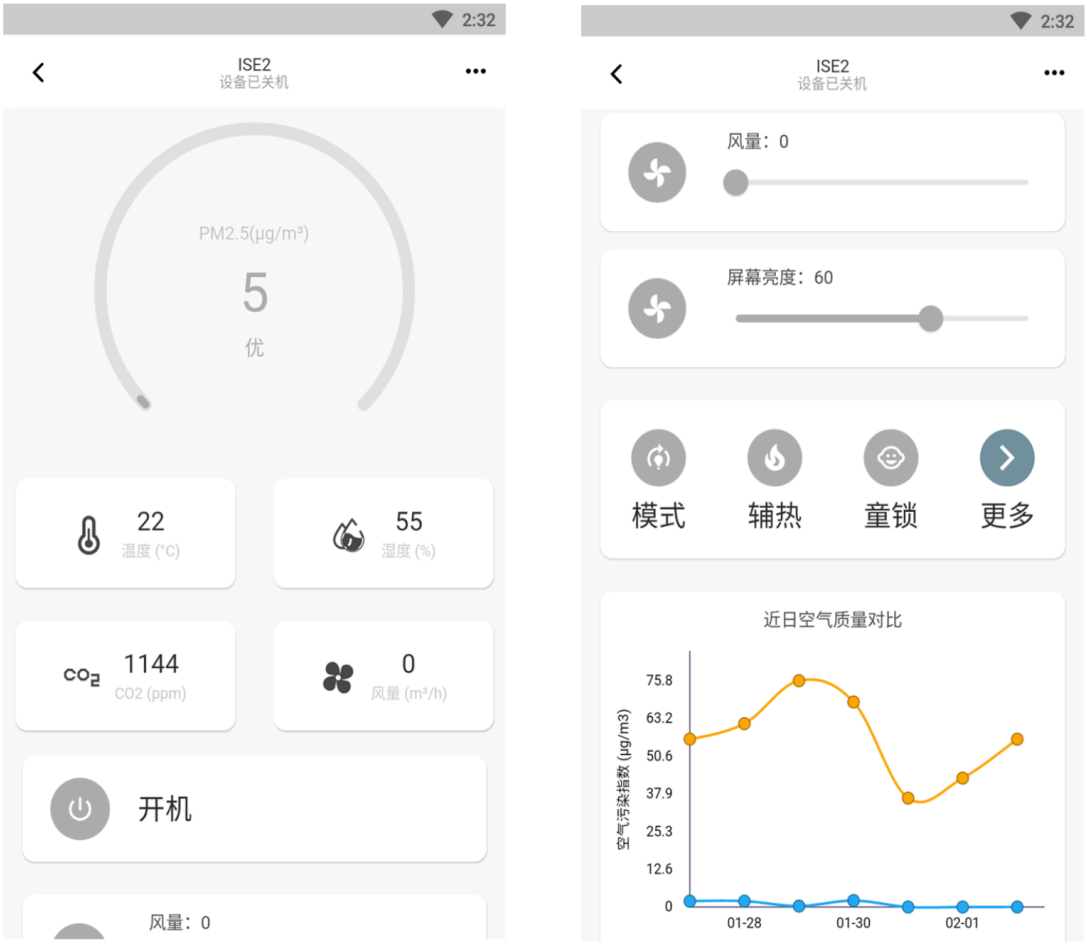


图 4.6: 设备远程控制界面

### 4.2.3　设备运行数据存储与获取的实现

日志数据的来源主要有两个方面，除去用户操作设备时会产生用户操作的日志数据外，设备在运行时也会产生一系列的运行时数据。获取、整理并利用这些数据对于衡量设备所处环境的恶劣情况以及使用建议的个性化推荐有重要的作用。在该技术中，MachineStatusRepository 提供了将日志数据写入到 MongoDB 的功能，并利用 Redis 为常用的数据设计了缓存，客户端可以通过发送 HTTP 请求到 MachineStatusController 获取设备的数据，在获取设备数据时，我们需要指定相应的查询参数，如查询的时间段、查询的具体设备 ID 等信息。设备运行数据存储与获取的关键代码如图4.7所示。

图 4.7: 设备运行数据存储与获取关键代码

```
// 构造查询条件
LocalDate lastMonth = LocalDateTime.now().minusMonths(1).toLocalDate();
LocalDate currentMonth = LocalDateTime.now().toLocalDate();
int month = lastMonth.getMonth().getValue();
Map<String, Object> condition = new HashMap<>();
condition.put("createTimeGTE", lastMonth);
condition.put("createTimeLTE", currentMonth);
condition.put("blockFlag", false);
ResultData response =
    machineStatusStatisticsDao.selectDaily(condition);
// 用子线程记录设备状态
CorePool.getComExecutor().execute(new Thread(() ->
       communicationService.create(status)
));
CorePool.getComExecutor().execute(new Thread(() -> {
    LimitQueue<MachineStatus> queue;
    if (redisService.exists(uid) == false) {
        queue = new LimitQueue<>(120);
        queue.offer(status);
    } else {
        queue = (LimitQueue<MachineStatus>) redisService.get(uid);
        queue.offer(status);
    }
    redisService.set(uid, queue, (long) 120);
}));
```

## 4.3　消息推送模块设计与实现

### 4.3.1　消息推送模块的详细设计

消息推送模块主要实现场景构建时的异步消息发送接收和应用程序消息推送功能的实现，分为异步消息发送和应用程序推送两部分。异步消息发送用于场景执行时分批次向设备发送控制指令，在设备控制模块中，我们提到了是通过一条指令实现对单个设备的单个操作，因此如果想一次同时对多个设备发送多个指令我们需要对指令进行打包，由于这是一个耗时操作，因此这里我们构

造消息体后以异步的方式将消息发送出去。异步消息的发送我们借助 RocketMQ 消息队列来实现。

消息推送模块的第二个功能是实现应用程序的消息推送。应用程序的消息推送其实就是从服务器端向移动终端发送连接，传输一定的信息。目前最主流的终端设备是 iOS 和 Android，这两个终端都有对应的推送机制，为了实现对两个平台的消息实现定向推送，需要针对每个平台的特性分别实现一套推送机制，并且保证推送时的连接稳定。因此，开发者大多选用第三方推送服务商实现推送功能，一个因为是服务器的开发和维护成本，再一个就是第三方的推送能够做到精准投放。在这部分，该技术利用极光推送服务实现服务端向应用程序端消息的推送。

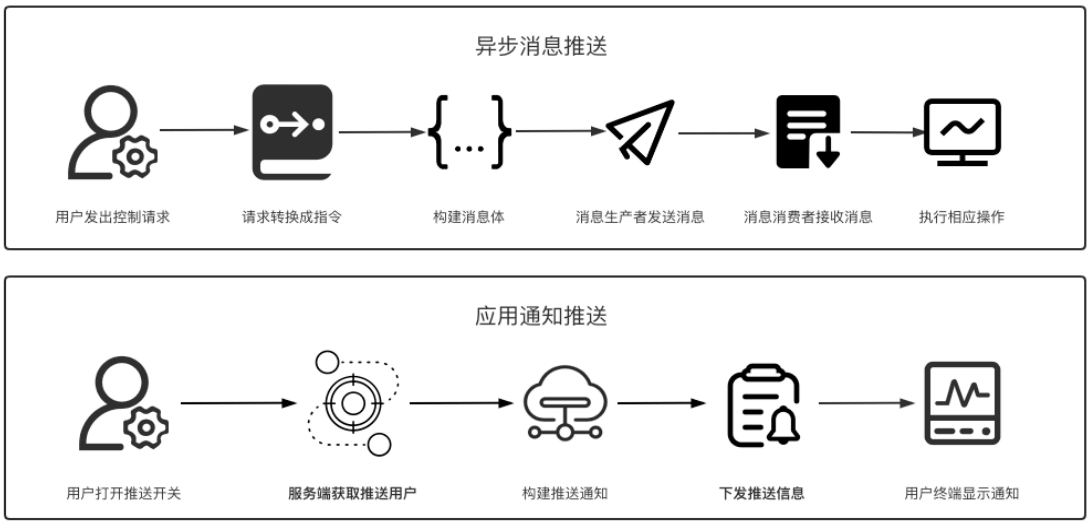


图 4.8: 消息推送模块流程示意图

如图4.8所示为该模块的流程示意图。在异步消息推送时，首先由用户先发出控制请求，服务端在接收到用户请求后便会转换成相应的指令，随后构建用于消息推送的消息体。利用消息生产者发送消息后，会有消息消费者接收到消息并随即执行对应的操作。在应用通知功能中，由用户在功能设置选项中打开消息推送开关，服务端会根据用户是否打开推送开关定向推送消息，服务端推送消息后，用户终端会依赖系统功能显示下发的通知信息，至此完成整个消息推送流程。

该模块核心类图如图4.9所示，MsgPushController 用于接收发送消息体的请求，并依赖 AppMsgPushService 进行业务逻辑处理。在推送消息时，会通过 UserMsgPushMapper 获取打开消息推送的用户列表，以此实现消息的定向推送。

ScheduledTask 定义了一些定时任务，例如向用户定时推送天气信息、使用建议等，此部分同样依赖 AppMsgPushService。

在执行场景时，我们需要在 SceneOperationService 中构建需要异步发送的消息体，然后利用 SceneOperationProducer 实现消息的异步发送。当消息发送成功后，我们将会在 SceneOperationConsumer 中接收消息，并验证消息的 topic 和 tag，验证成功后便会在 process 方法中处理该消息，该部分，我们会调用 machineClient 接口按次序的实现对设备的远程控制。

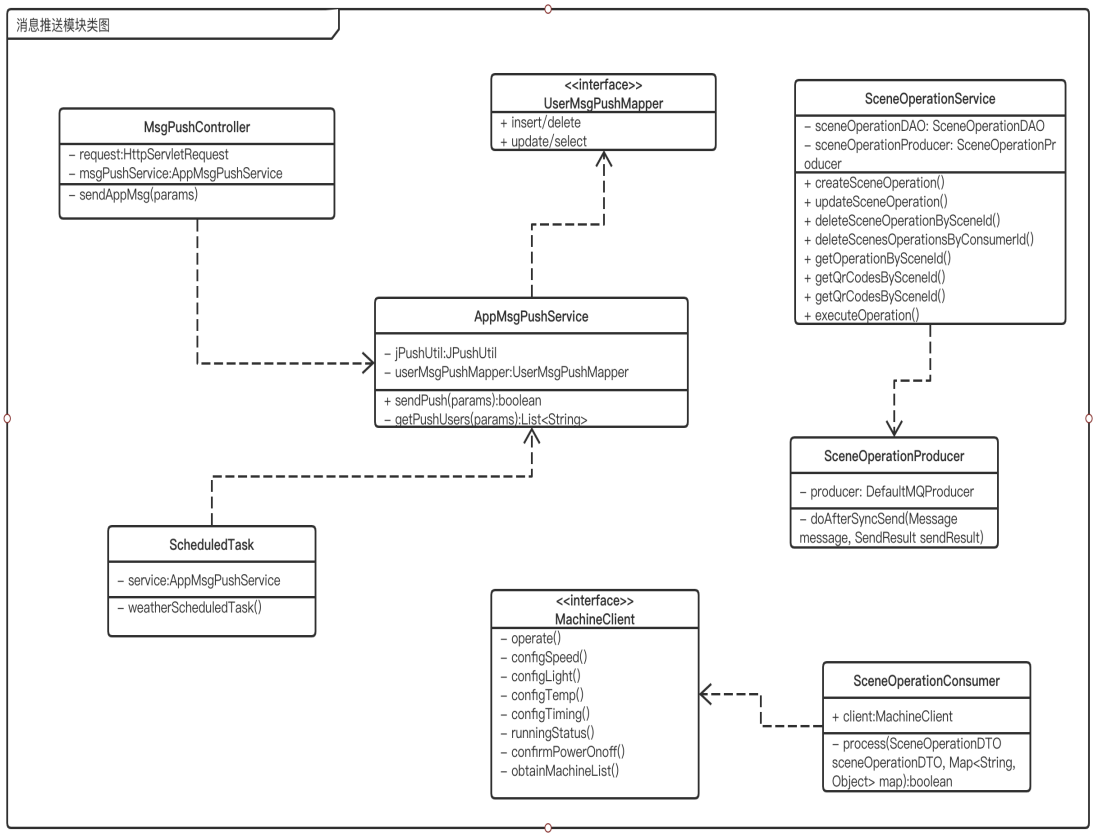


图 4.9: 消息推送模块类图

### 4.3.2　服务端异步消息推送的实现

服务端异步消息推送主要指的是服务端内部模块的通信，例如在执行场景时发送的场景执行指令。在设备控制模块中，一条指令只能对应单个设备的单个行为，因此我们在创建场景时有必要将需要执行的指令按照一定的次序进行封装。在消息推送的过程中有两个主要角色，分别是消息的生产者 producer 和消息的消费者 consumer，在接收到用户执行场景的指令后，服务端会生成相应的消息，消息中包含要传递的消息体、消息的主题 topic 和标签 tag 等信息，随

后通过 producer 将消息发出，此时消息的消费者 consumer 在收到消息后将会验证此条消息的主题 topic 和标签 tag，验证成功后才会执行相应的业务逻辑。

如图4.10中代码所示，在推送消息前，我们需要利用 MessageBuilder 构建一个指定主题 topic 的消息，然后再利用我们创建的消息生产者执行 syncSend() 函数即可实现消息的发送。消息发送成功后，首先会将消息持久化到消息队列的 broker 服务器中，待到消息的消费者消费该条消息后，改变消费状态。如图4.11所示为异步消息推送的控制台界面示意图，通过控制台我们可以查看每个 broker 的消费情况、每个 topic 的消费情况以及消费趋势等信息。

图 4.10: 异步消息推送关键代码

```
// 创建消息发送者
@MQProducer
public class SceneOperationProducer extends AbstractMQProducer {
    @Override
    public void doAfterSyncSend(Message message, SendResult sendResult){
        super.doAfterSyncSend(message, sendResult);
    }
}

// 构建消息体并发送消息
Message msg = MessageBuilder.of(sceneOperationDTO)
    .topic("scene-operation").build();
sceneOperationProducer.syncSend(msg);
public void syncSend(Message message) throws MQException {
    try {
        SendResult sendResult = producer.send(message);
        log.debug("send rocketmq message ,messageId : {}",
            sendResult.getMsgId());
        this.doAfterSyncSend(message, sendResult);
    } catch (Exception e) {
        throw new MQException("消息发送失败, topic :" +
            message.getTopic() + ",e:" + e.getMessage());
    }
}
```

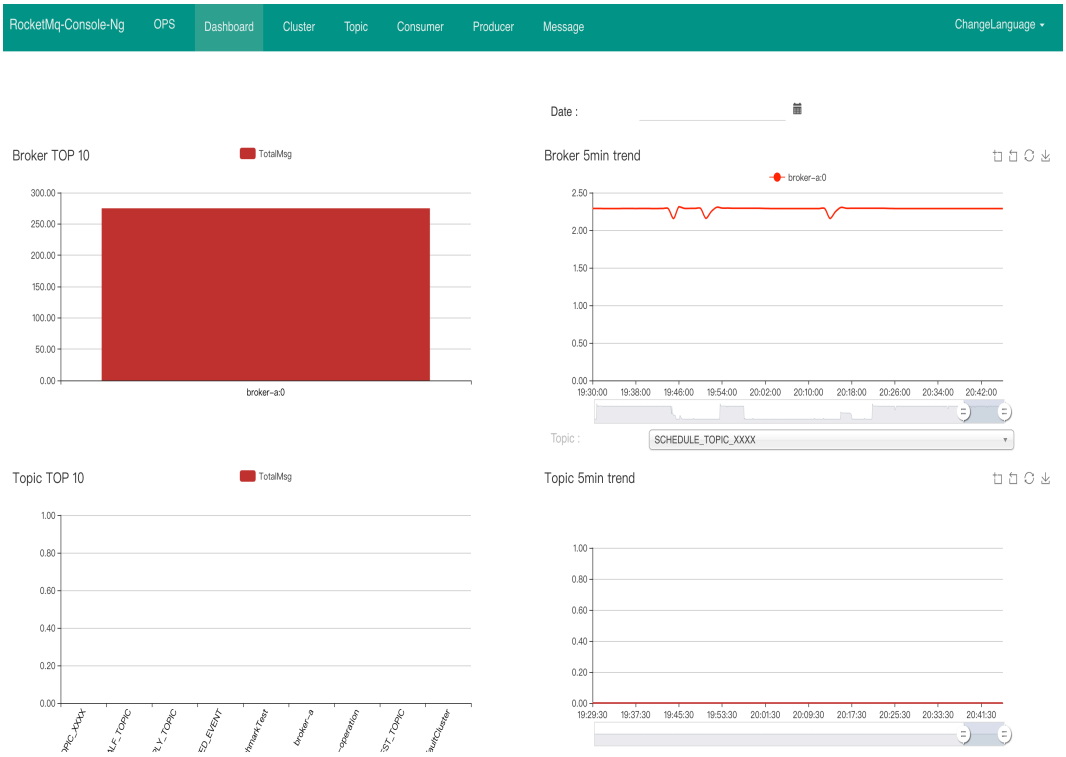


图 4.11: 异步消息控制台界面示意图

消息发送成功后，我们创建的指定主题 topic 和消费者群组的消费者便会接收消息并消费。我们利用 @MQConsumer 实现 topic 和 consumerGroup 的注入，利用 process() 实现消息的消费，消息接受的关键代码如图4.12 所示，执行消息体的代码如图4.13所示。

图 4.12: 异步消息接收关键代码

```
// 执行接收消息的主题topic和消费组名称consumerGroup
@MQConsumer(topic = "scene-operation", consumerGroup =
    "scene-consumer-group-1")
public boolean process(SceneOperationDTO sceneOperationDTO,
    Map<String, Object> map){
    String[] configArr = { "speed", "light", "timing", "temp" };
    // 先根据sequence对指令顺序进行排序
    List<SceneOperationCommand> commands =
        sceneOperationDTO.getCommands().stream()
        .sorted(Comparator.comparing(SceneOperationCommand::getSequence))
        .collect(Collectors.toList());
}
```

图 4.13: 执行接受消息体代码

```
// 执行接收消息的主题topic和消费组名称consumerGroup
public boolean process(SceneOperationDTO sceneOperationDTO,
    Map<String, Object> map){
    for (SceneOperationCommand command : commands) {
        String device = command.getDeviceName();
        String component = command.getCommandComponent();
        String operation = command.getCommandOperation();
        log.info("device is:{},component is: {}, operation is: {}",
            device, component, operation);
        int value = NumberUtils.toInt(operation);
        ResultData resultData =
            machineClient.config(command.getQrCode(), value);
    }
    return true;
}
```

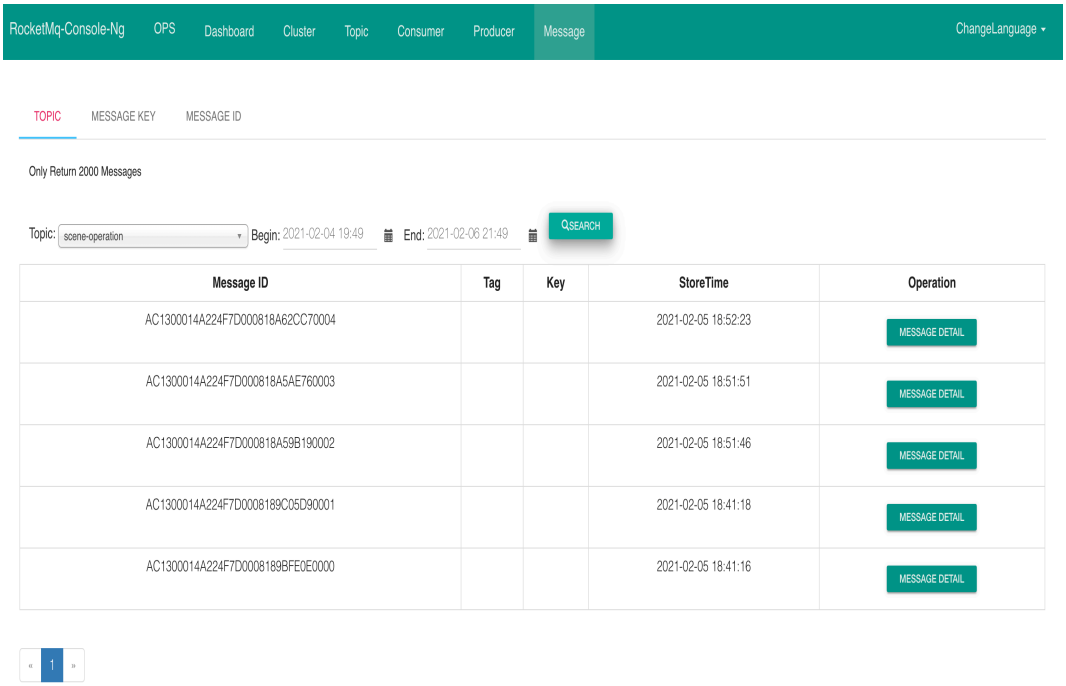


图 4.14: 异步消息管理界面示意图

异步消息的管理如图4.14所示，支持时间范围、消息主题等信息对异步发送的消息进行列表查询。异步消息的消费情况如图4.15所示，包括消费队列信息、消费时间等信息。

| broker | queue | consumerClient | brokerOffset | consumerOffset | diffTotal | lastTimestamp |
|---|---|---|---|---|---|---|
| broker-a | 0 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 94 | 94 | 0 | 2021-01-20 16:41:10 |

| Topic | scene-operation | Delay | 0 | LastConsumeTime | 2021-02-05 18:52:23 |
|---|---|---|---|---|---|

| broker | queue | consumerClient | brokerOffset | consumerOffset | diffTotal | lastTimestamp |
|---|---|---|---|---|---|---|
| broker-a | 0 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 2 | 2 | 0 | 2021-01-22 12:24:51 |
| broker-a | 1 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 1 | 1 | 0 | 2021-02-05 18:41:16 |
| broker-a | 2 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 3 | 3 | 0 | 2021-02-05 18:51:51 |
| broker-a | 3 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 2 | 2 | 0 | 2021-01-20 17:11:16 |
| broker-a | 4 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 1 | 1 | 0 | 2021-01-20 15:05:19 |
| broker-a | 5 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 1 | 1 | 0 | 2021-02-05 18:41:18 |
| broker-a | 6 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 1 | 1 | 0 | 2021-01-20 16:25:28 |
| broker-a | 7 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 0 | 0 | 0 | 1970-01-01 08:00:00 |
| broker-a | 8 | 172.19.0.1@c2de9a96-3103-4e93-b76b-dc17af9ed093 | 3 | 3 | 0 | 2021-01-20 16:13:05 |

图 4.15: 异步消息消费情况

### 4.3.3　应用程序消息推送的实现

应用程序的消息推送指的是由服务端向用户终端设备推送通知消息 [44]。由于该系统需要同时运行在安卓和 iOS 系统上，因此需要针对不同的使用环境采取不同的推送手段。安卓系统本身具有推送模块，但由于国内主流手机厂商对于安卓系统的高度定制，原生推送模块几乎被屏蔽，并且由于原生安卓的推送模块由于其服务器在海外，连接性能较差，并不稳定。相比之下，iOS 端的推送则更有章程，系统内部继承苹果公司独有的推送服务，开发者只需满足苹果公司的请求即可接入应用推送。

正是基于上述原因，以极光为首的第三方推送服务在国内迅速发展，这些产品大多采用长连接方式实现信息推送。由于本文重点不在于着手实现一套自用的推送服务，因此这里直接采用第三方推送服务实现应用程序的消息推送。

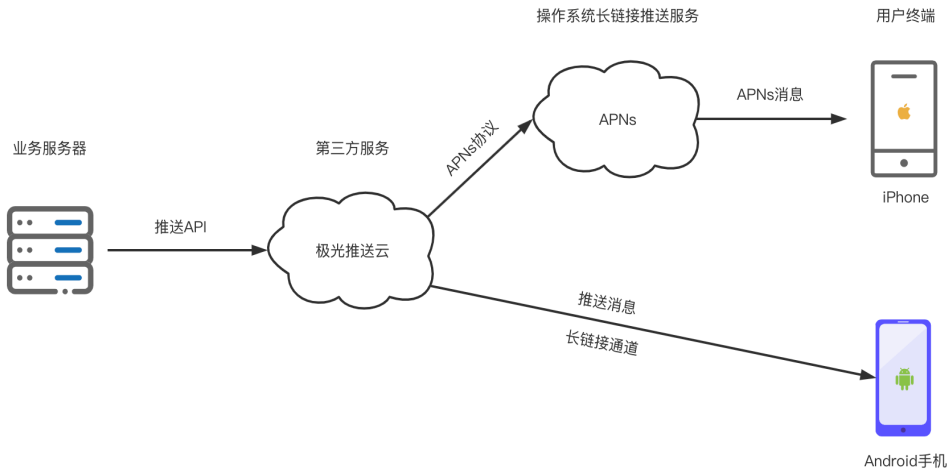


图 4.16: 场景构建应用消息推送架构图

如图4.16所示为本应用程序推送架构，在业务服务器中，接入第三方推送服务后，我们需要在业务服务器创建消息推送的客户端，用于将消息推送至极光推送云，随后构建待推送的消息体，主要包括待推送的标题、内容和推送对象等信息。我们可以通过两种方式实现应用通知的下发，一是通过控制台直接下发，如图4.17所示。二是使用代码在服务端下发，详细代码如图4.18所示。


图 4.17: 控制台下发应用通知

发送消息后，接入了极光推送的客户端，首先会验证一些额外的参数，验证完成后便可以接收来自服务端的推送消息并作出相应的操作了。如图4.19所示，在服务端消息发送成功后，我们可以在消息推送后台查看到消息推送的详细情况，如消息发送时间、消息内容、目标用户、覆盖平台等。

图 4.18: 应用程序消息推送关键代码

```
public PushResult sendPush(String title, String content, Map<String,
    String> extrasMap, String... alias) {
    ClientConfig clientConfig = ClientConfig.getInstance();
    clientConfig.setTimeToLive(Long.valueOf(jpushConfig.getLiveTime()));
    // 使用NativeHttpClient网络客户端，连接网络的方式，不提供回调函数
    JPushClient jpushClient = new
        JPushClient(jpushConfig.getMasterSecret(),
        jpushConfig.getAppkey(), null,
           clientConfig);
    // 设置推送方式
    PushPayload payload = buildPushPayload(title, content, extrasMap,
        alias);
    PushResult result = null;
    try {
        result = jpushClient.sendPush(payload);
        log.info("推送结果 - " + result);
    } catch (APIConnectionException e) {
        log.error("推送连接错误，请稍后重试 ", e);
        log.error("Sendno: " + payload.getSendno());
    } catch (APIRequestException e) {
        log.error("服务器响应出错，请修复！ ", e);
    }
    return result;
}
// 构建消息体、标题及推送内容
AppMsgPushDTO appMsgPush = new AppMsgPushDTO();
appMsgPush.setTitle(title);
appMsgPush.setContent(content);
appMsgPush.setAlias(service.getPushUsers());
// 发送消息
service.sendPush(appMsgPush);
```

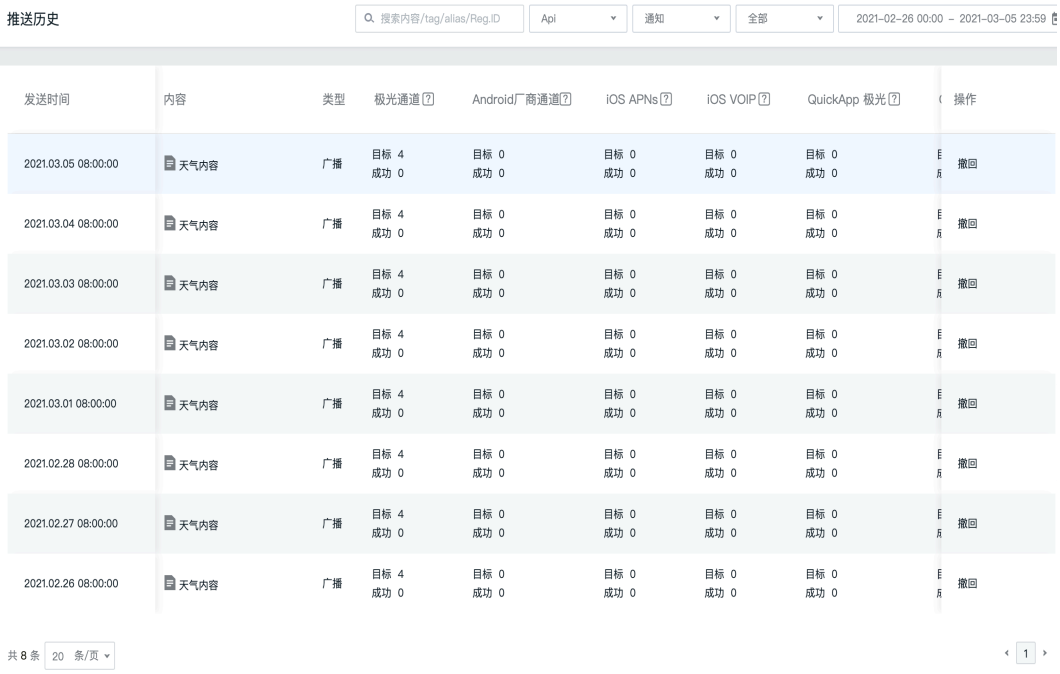


图 4.19: 场景构建应用消息推送详情

客户端只有在应用启动时初始化推送服务后才可以接收到来自服务端推送的消息，在初始化时需要注入 appKey、channel、用户身份等初始化信息，待收到服务端发送消息后，客户端将会进行校验随即做出响应。

图 4.20: 客户端接收消息关键代码

```
// 创建接收消息客户端
JPush jpush = new JPush();
///配置应用 Key
jpush.setup(appKey: "",channel: "");
// 设置别名
jpush.setAlias(user.consumerId);
jpush.applyPushAuthority(new NotificationSettingsIOS(sound: true,
    alert: true, badge: true));
```

如图4.21所示，在客户端完成应用服务推送的配置后，一旦服务端完成消息推送，应用程序将会以通知的形式在用户通知栏显示推送消息。用户点击通知栏的消息后便会启动应用程序进入相应的页面。

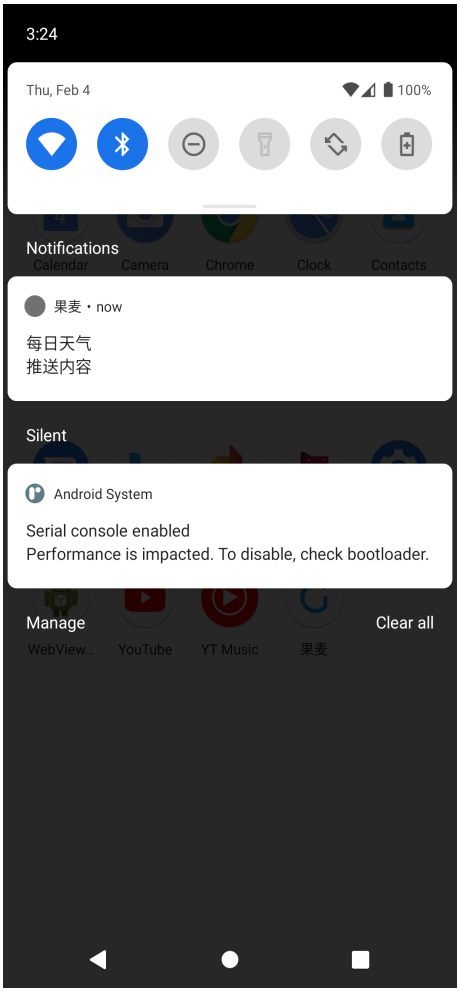


图 4.21: 应用程序推送通知

## 4.4　场景构建模块设计与实现

### 4.4.1　场景构建模块的详细设计

场景构建模块主要包括用户将多设备联动创建场景与执行场景两个功能。用户创建场景时需添加场景内包含的设备以及设备需要执行的操作。场景内设备的添加来自于用户已绑定的设备，这部分数据由设备控制模块提供。场景创建后，用户可以查看场景内的环境数据详情，如温湿度、PM2.5、CO2 浓度等信息。在场景创建时，我们需要在服务端构建用户可操作设备的控制选项，只有符合条件的用户和设备可以在场景创建时被添加，场景创建后便会将场景数据写入数据库中。

第二个功能便是场景的执行。所谓场景执行指的是按照场景创建时添加的设备操作依次执行场景内设备的控制，该功能依赖消息推送模块和设备控制模

块。在场景执行时，由于场景内设备操作较多，以同步的方式执行场景将会严重影响用户体验，通过消息推送模块的异步消息推送功能将消息体发送至待执行的服务，可以让用户在客户端无需等待较长时间，优化用户体验。消息消费者收到消息后将会按照次序依次同设备控制模块交互，以实现远程对设备进行顺序的指令控制。

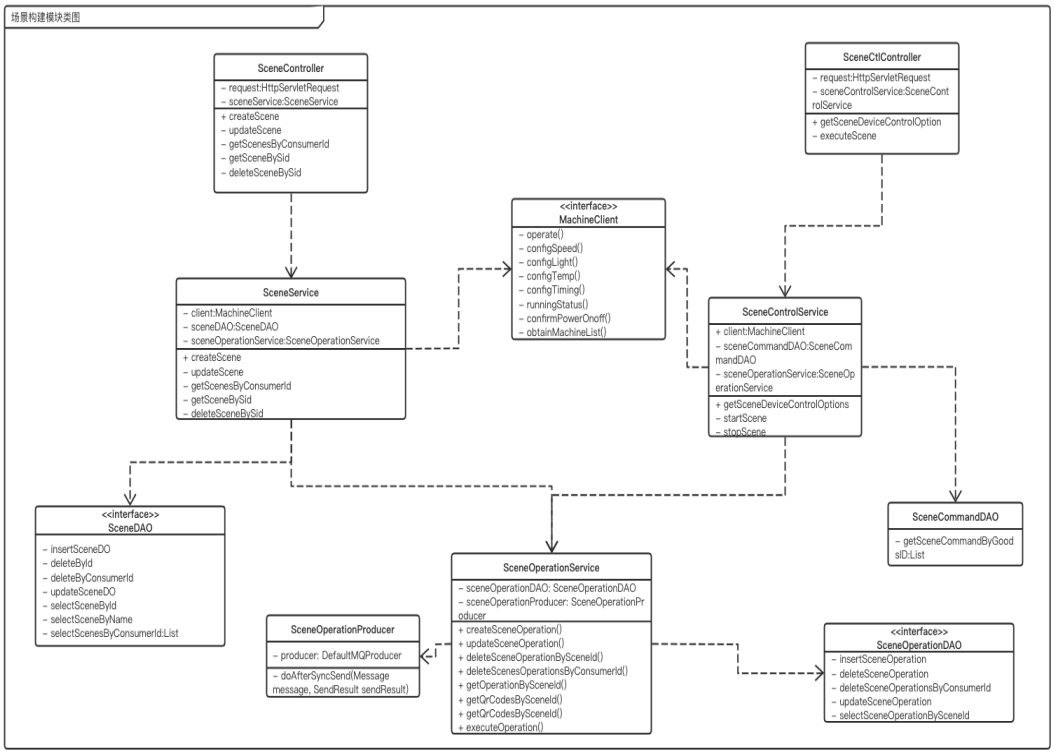


图 4.22: 场景构建模块类图

该模块的核心类图如图4.22所示。SceneController 用于接收和场景相关的请求，如场景的创建、名称修改、场景详情的获取等，SceneCtlController 用于接收对场景的控制请求，如执行场景内的操作等行为。这两个控制器分别依赖 SceneService 和 SceneControlService 这两个服务。

SceneService 中通过设备控制模块中的 MachineClient 实现对设备数据的获取，SceneControlService 通过 MachineClient 实现对设备的远程控制，这两个服务同时又依赖 SceneOperationService，该类负责场景内的有关操作，包括控制选项的增删改查、场景内指令的远程发送，指令发送依赖消息推送模块中的异步消息推送功能，即 SceneOperationProducer。SceneDAO 用于场景信息和数据库的访问，SceneOperationDAO 用于场景内控制选项与数据库访问，SceneCommandDAO 用于控制指令与数据库的访问。

### 4.4.2　场景创建的实现

用户在客户端创建场景时，需要填写场景名称、场景内的操作选项。用户需选择已绑定的设备和相关的操作，随后便会向服务端发送创建场景的请求，在客户端以对象的方式创建场景，在请求时转换成 JSON 文本，后端接收到请求后便会将数据存储在数据库中，由于场景信息和场景操作信息的数据结构不同，我们针对数据特点将场景信息存放在 MySQL 中，将场景内操作信息存放在 MongoDB 中。如图4.25所示是客户端创建场景的界面示意图，客户端创建场景以及服务端写入场景的关键代码如图4.23和4.24所示。

图 4.23: 客户端创建场景操作关键代码

```
SceneOperation operation = _sceneOperation;
User user = SpUtil.getObj(SpStr.user, (v) => User.fromJson(v));
operation.consumerId = user.consumerId;
operation.sceneName = _sceneNameController.text;
int len = _sceneOperation.commands.length;
List<SceneOperationCommand> commands = [];
for (int i = 0; i < len; i++) {
    SceneOperationCommand command = new SceneOperationCommand();
    command.commandName = _sceneOperation.commands[i].commandName;
    command.commandComponent =
        _sceneOperation.commands[i].commandComponent;
    command.commandOperation =
        _sceneOperation.commands[i].commandOperation;
    command.sequence = i + 1;
    commands.add(command);
}
operation.commands = commands;
Scene scene = new Scene();
scene.sceneName = operation.sceneName;
scene.consumerId = user.consumerId;
scene.sceneOperation = operation;
scene.status = 'off';
// 发送请求，执行创建场景的方法
BaseResp resp = await SceneService.createScene(scene);
```

图 4.24: 服务端写入场景操作关键代码

```
SceneDO tmp = sceneDAO.selectSceneByName(sceneDTO.getName());
SceneDO sceneDO = new SceneDO();
BeanUtils.copyProperties(sceneDTO, sceneDO);
// 写入场景基本信息
long sceneId = sceneDAO.insertSceneDO(sceneDO);
sceneDTO.setId(sceneId);
// 获取场景内的操作
SceneOperationDTO sceneOperationDTO = sceneDTO.getSceneOperation();
sceneOperationDTO.setSceneId(sceneId);
boolean flag =
    sceneOperationService.createSceneOperation(sceneOperationDTO);
sceneDTO.setSceneOperation(sceneOperationDTO);
```


图 4.25: 场景创建界面示意图

### 4.4.3　场景信息获取的实现

场景是由多个设备组成的逻辑实体，因此场景信息的获取即场景内设备数据整理后的结果，如场景内的 CO2 浓度等。创建场景时，我们需要将场景内相关数值置为 0，在获取场景信息时，我们再获取相关的数据并更新数据库信息。

在获取场景信息时，值得注意的是，由于设备的种类不同，每个设备可以获取的空气数据的类型也不一样，如果以平均值作为衡量场景信息的指标会影

响到环境数据的准确性，无法起到警示的作用，因此在展示场景数据时我们采用最大值作为场景数据的指标。从数据库中获取的数据为 SceneDO，初始化场景时只包含场景名称和创建人等基本信息，为了让返回至前端的结果更加丰富，我们需要获取场景内所包含的设备、场景内空气数据以及用户给场景设置的一些列指令操作。

图 4.26: 场景信息获取关键代码

```
// 场景获取
public List<SceneDTO> getScenesByConsumerId(String consumerId) {
    List<SceneDO> sceneDOS =
        sceneDAO.selectScenesByConsumerId(consumerId);
    List<SceneDTO> sceneDTOS = sceneDOS.stream().map(this::sceneDO2DTO)
    .collect(Collectors.toList());
    return sceneDTOS;
}
public SceneDTO sceneDO2DTO(SceneDO sceneDO) {
    long sceneId = sceneDO.getId();
    SceneDTO sceneDTO = new SceneDTO();
    BeanUtils.copyProperties(sceneDO, sceneDTO);
    // 获取场景内操作
    SceneOperationDTO sceneOperationDTO =
        sceneOperationService.getOperationBySceneId(sceneId);
    sceneDTO.setSceneOperation(sceneOperationDTO);
    // 获取场景内包含的设备（从redis中获取）
    List<String> qrCodes = getSceneQrCodesBySceneId(sceneId);
    sceneDTO.setQrCodes(qrCodes);
    setData2Scene(sceneDTO, qrCodes);
    return sceneDTO;
}
```

如图4.26中代码所示，在获取场景信息时，我们需要先从数据库中获取对应的数据，并通过 sceneDO2DTO 方法对数据库中的数据做一些处理，这些处理包括属性的复写，这里使用的是 BeanUtils 中的 copyProperties 方法。

在设备控制模块中我们利用设备二维码作为设备的唯一标识，由于场景是由多个设备组成的，因此为了获取场景的空气数据，我们同样需要利用设备的二维码实现对场景内多设备的场景数据的获取与处理。在获取时，结合设备控制模块获取单个设备的相关数据，并将结果数据以 JSON 格式保存，对数据进行

解析后我们以最大值作为指标衡量场景内的空气情况，场景数据填充的关键代码如图4.27所示。

图 4.27: 场景数据填充关键代码

```
public void setData2Scene(SceneDTO sceneDTO, List<String> qrCodes) {
    double co2 = 0;
    double humidity = 0;
    double pm25 = 0;
    double temperature = 0;
    for (String qrCode : qrCodes) {
        // 依次获取设备状态
        ResultData data = machineClient.runningStatus(qrCode);
        if (data.getResponseCode() == ResponseCode.RESPONSE_OK) {
            JSONObject object =
                JSON.parseObject(JSON.toJSONString(data.getData()));
            log.info("json data is:{}", JSON.toJSONString(object));
            co2 = Math.max(co2, object.getDoubleValue("co2"));
            humidity = Math.max(humidity,
                object.getDoubleValue("humidity"));
            temperature = Math.max(temperature,
                object.getDoubleValue("temperature"));
            pm25 = Math.max(pm25, object.getDoubleValue("pm2_5"));
        }
    }
    sceneDTO.setTemperature(temperature);
    sceneDTO.setCo2(co2);
    sceneDTO.setHumidity(humidity);
    sceneDTO.setPm25(pm25);
}
```

如图4.28所示，用户点击所创建的场景列表后便会展示该场景内的空气数值。通过上述代码，前端在访问 SceneController 时便会获得一个包含了场景数据的场景实体对象，除刚才展示的空气数值外，还包括场景内的一些控制选项，这些信息控制着场景执行的实现过程。

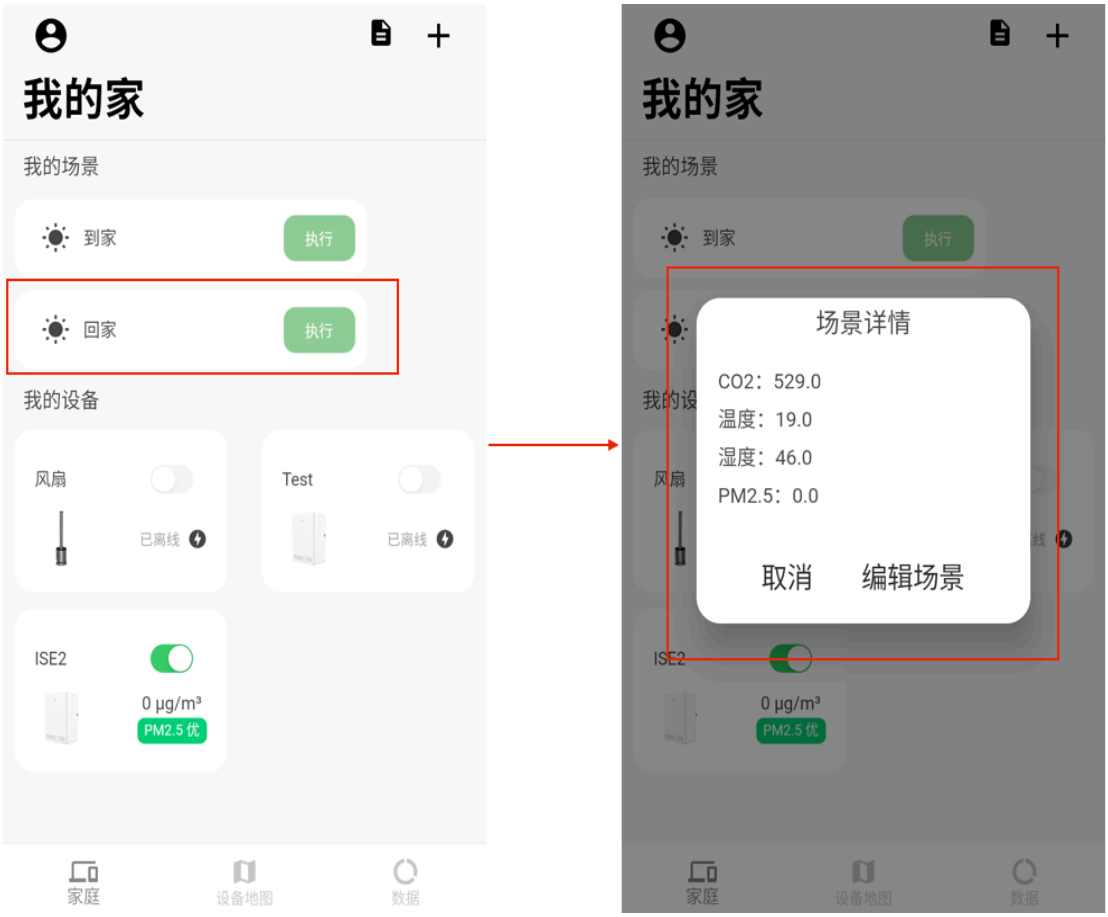


图 4.28: 场景信息界面示意图

### 4.4.4　场景执行的实现

执行场景即执行场景中的所有操作，由于一个场景可能包含多个操作，因此如果以同步的方式执行指令将会影响用户体验，因此这里使用到消息推送模块中的异步消息通知功能，即以消息队列的方式将执行压力进行转移。

因此在执行场景时，我们首先需要根据场景 ID 获取该场景内的操作详情，其中包括场景的指令、场景内设备列表等信息。然后对场景内的指令按照场景创建时的顺序即 sequence 进行升序排序。

之所以采取这种方案，其目的是为了保证在场景执行时可以按照用户创建时的意愿执行，而不会造成设备运行顺序的紊乱，后端服务只需利用设备控制模块分别执行场景指令即可。场景执行的关键实现代码如图4.29所示。如图4.30为场景执行效果图。

图 4.29: 场景执行的关键代码

```
SceneOperationDTO sceneOperationDTO =
    sceneOperationService.getOperationBySceneId(sceneId);
// 用消息队列发送指令
Message msg = MessageBuilder.of(sceneOperationDTO)
    .topic("scene-operation").build();
sceneOperationProducer.syncSend(msg);
// 消费消息,先根据sequence对指令顺序进行排序
List<SceneOperationCommand> commands =
    sceneOperationDTO.getCommands().stream()
    .sorted(Comparator.comparing(SceneOperationCommand::getSequence))
    .collect(Collectors.toList());
for (SceneOperationCommand command : commands) {
    String device = command.getDeviceName();
    String component = command.getCommandComponent();
    String operation = command.getCommandOperation();
    resultData = machineClient.operate(command.getQrCode(), component,
        operation);
}
```

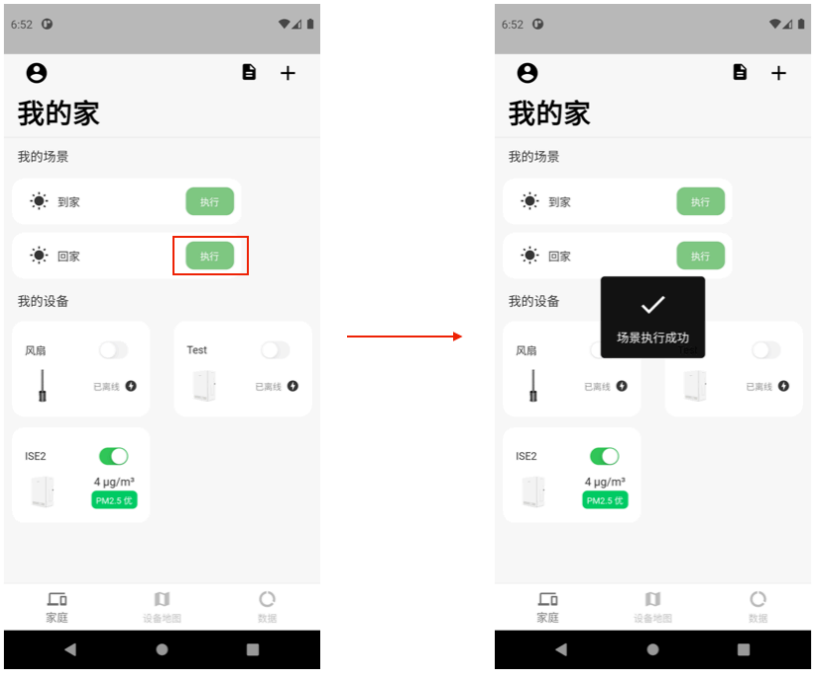


图 4.30: 场景执行界面示意图

## 4.5　日志处理模块设计与实现

### 4.5.1　日志处理模块的详细设计

日志模块主要功能是提供创建日志的接口，供其他模块在使用时记录使用日志。主要包括用户行为日志和设备控制日志等。行为日志主要指的是用户执行和自身账户有关的操作如绑定微信、修改用户名等；设备控制日志主要指的是用户对设备的操作记录，如控制设备的开关、温湿度控制等。该模块的核心类图如图4.31所示。

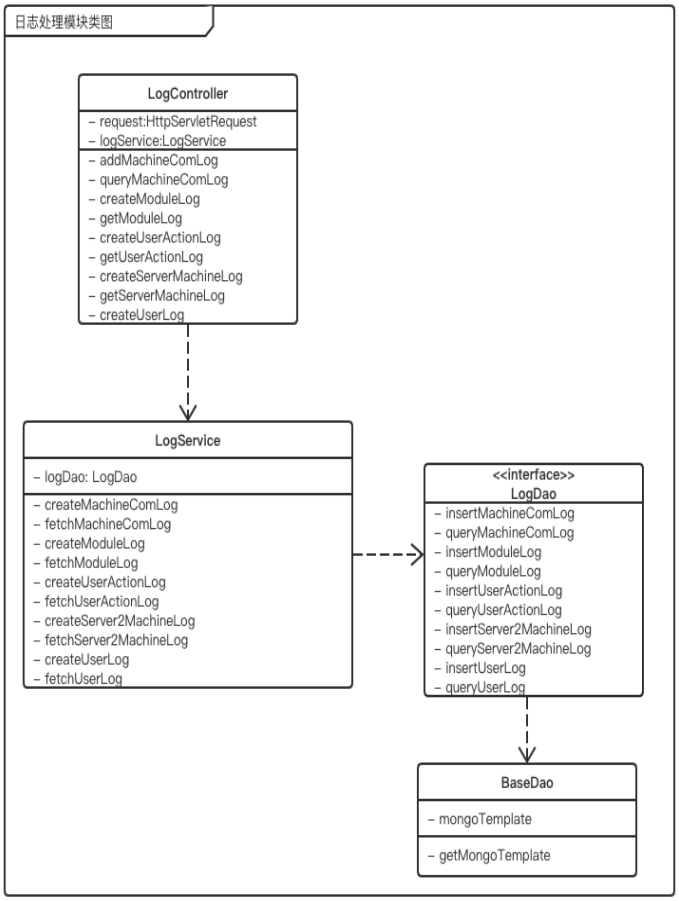


图 4.31: 日志处理模块类图

LogController 为请求入口类，其他模块例如设备控制模块和场景构建等模块记录日志时便会调用该入口类的方法，如用户行为日志的创建与查询、用户设备操作日志的创建与查询。该类主要依赖 LogService，该类提供了有关行为日志、设备操作日志的增删改查的方法，这些方法以 LogDao 作为和数据库的访问层，考虑到日志数据的结构性较弱，以 mongodb 作为数据库存储。

### 4.5.2　日志处理模块的实现

系统发送创建日志的请求后，便会执行 LogService 中的相关方法，用于记录日志数据。以微服务的方式调用该模块时，需要记录日志创建的时间、执行人、执行模块等信息，并将结果返回。如图4.33是日志查看的界面示意图。

图 4.32: 日志写入的关键代码

```
// 写入日志的代码
public ResultData insertUserActionLog(UserMachineOperationLog
    userActionLog) {
    ResultData result = new ResultData();
    userActionLog.setLogId(IDGenerator.generate("USL"));
    mongoTemplate.insert(userActionLog, Collection_UserActionLog);
    result.setData(userActionLog);
    return result;
}
```

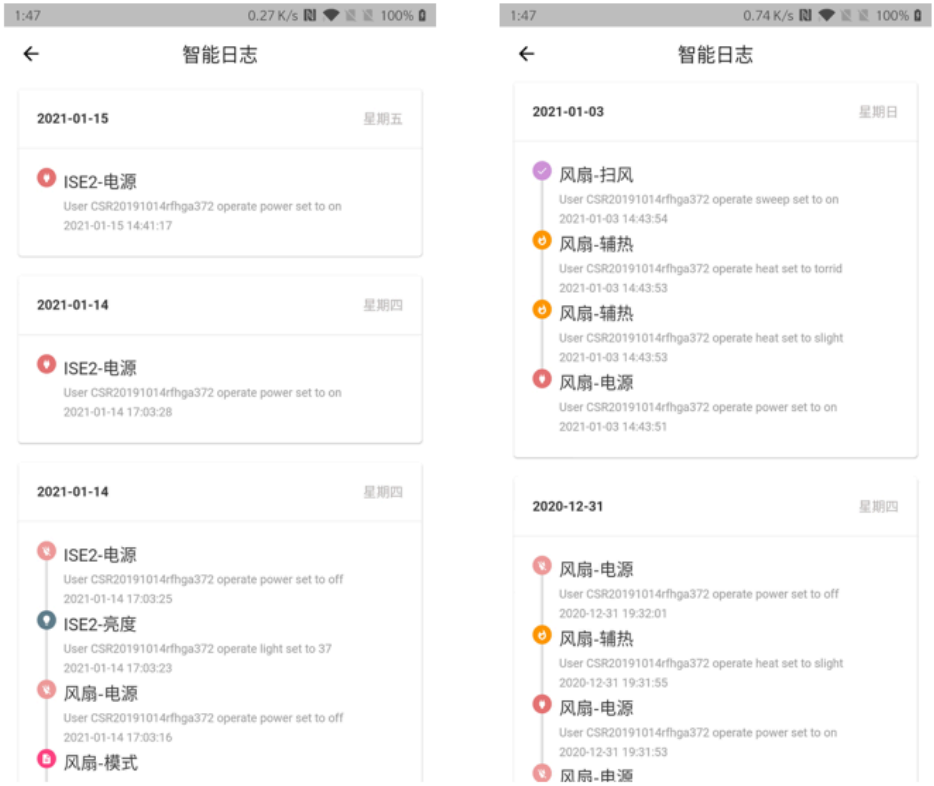


图 4.33: 日志查询界面示意图

## 4.6　数据分析模块设计与实现

### 4.6.1　数据分析模块的详细设计

本技术在使用时会产生两类数据，一类是设备运行时产生的数据，另一类则是设备被操作时的日志数据。MachineStatusController 用于接收有关设备运行数据分析的请求，该接口依赖众多和数据有关的服务，如 CO2、温湿度等，通过这些服务和数据库进行交互，获取设备运行时产生的数据并分析。UserActionController 用于分析有关设备操作时产生的操作数据，该部分主要和日志处理模块进行交互，通过对操作日志进行分析获取分析结果。该模块类图如图4.34所示。

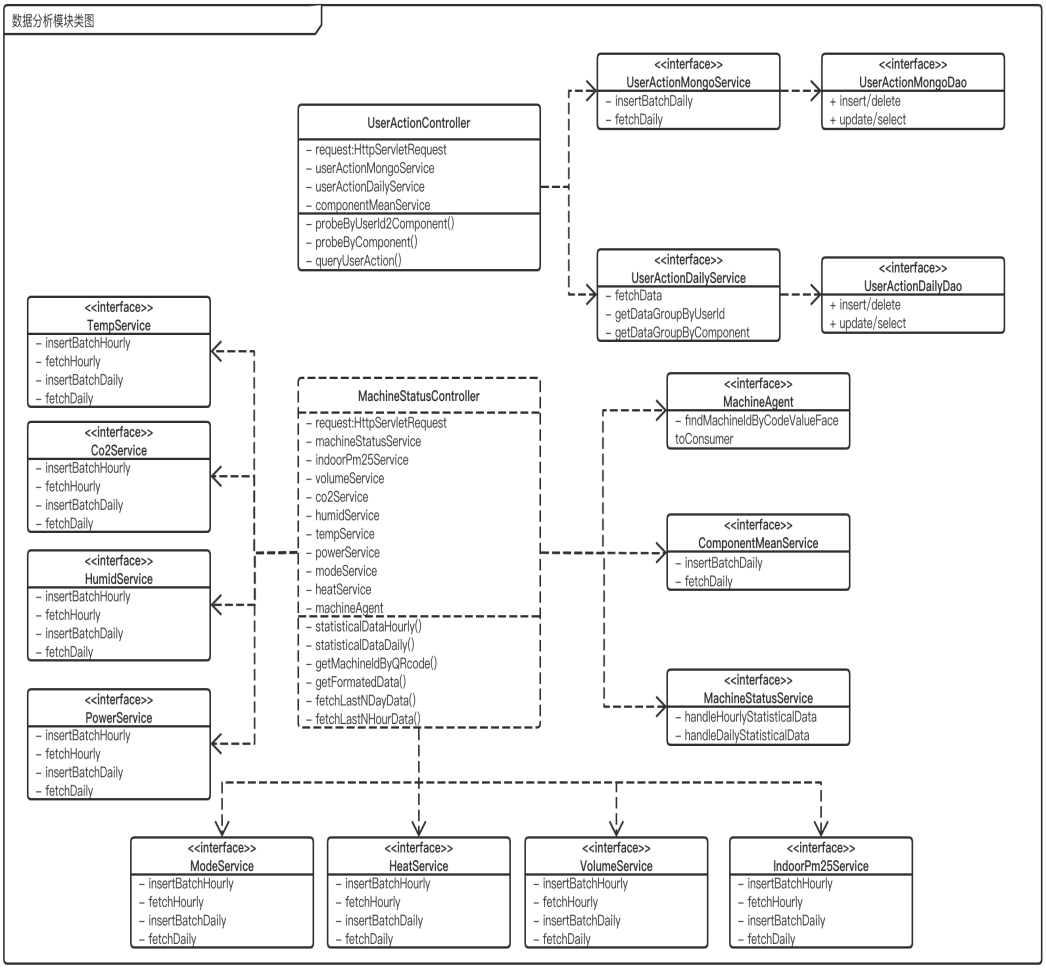


图 4.34: 数据分析模块类图

### 4.6.2　设备运行数据分析的实现

设备运行数据数据由设备运行时产生，设备运行数据分析包含两个方面，一方面为定时数据分析任务，系统会定时对 Redis 中存储的数据进行统计，经过运算分析后将结果持久化到 MySQL 中; 另一方面为数据获取与可视化，用户获取指定设备在规定时间段内的有效数据并予以分析及可视化。

系统通过定时任务每小时从 Redis 中获取设备运行时产生的数据，并通过计算引擎进行数据统计与分析，定时模块每天还将执行一次定时任务，从 MySQL 中查询当前的设备数据，将数据按天进行统计分析。设备运行数据分析的关键代码如图4.35所示。

图 4.35: 设备运行数据分析的关键代码

```
for (String machineId : map.keySet()) {
    Object queue = map.get(machineId);
    //若这个queue存了v1的status
    if (((LimitQueue<Object>) queue).getLast() instanceof
        MachineStatus) {
        LinkedList<MachineStatus> list = new LinkedList<>();
        for (int i = 0; i < ((LimitQueue) queue).size(); i++) {
            list.add(((LimitQueue<MachineStatus>) queue).get(i));
        }
        V1MachineStatusHourly msh = countV1Status(list);
        if (msh != null)
            statisticalDataList.add(msh);
    }
}
List<JSONObject> dataList = statisticalDataList.stream().map(e ->
    JSONObject.parseObject(e.toString())).collect(Collectors.toList());
List<IndoorPm25Hourly> pm25HourlyList = dataList.stream().map(e -> new
    IndoorPm25Hourly(e.getString("machineId"),
    e.getDouble("averagePm25"), e.getIntValue("maxPm25"),
    e.getIntValue("minPm25"))).collect(Collectors.toList());
indoorPm25HourlyDao.insertBatch(pm25HourlyList);
```

### 4.6.3　设备操作数据分析的实现

设备操作数据由设备控制模块产生，当用户对设备发出控制指令时便会调用日志处理模块将操作日志记录下来，然后由本模块读取日志处理模块整理的操作数据。对于用户操作设备的数据，我们给出如下的衡量指标，分别是：用户每天操作最多的设备、用户每天操作最多的设备模块、所有用户使用设备最频繁的时间段，所有用户使用最频繁的设备模块等。设备操作数据分析的关键代码如图4.36所示。

图 4.36: 设备操作数据分析的关键代码

```
//指定用户每天使用的最多的机器
public static String userDailyMachine(List<UserMachineOperationLog>
    list){
    Map<String,Integer> map = new HashMap<>();
    for(UserMachineOperationLog userMachineOperationLog : list){
        String key = userMachineOperationLog.getValue();
        map.put(key, map.getOrDefault(key, 0) + 1);
    }
    return getDiskMax(map);
}
//指定用户每天使用最多的时间段
public static String userDailyTime(List<UserMachineOperationLog> list){
    Map<String,Integer> map = new HashMap<>();
    for(UserMachineOperationLog userMachineOperationLog : list){
        String s =
            stampToDate(userMachineOperationLog.getTime()).substring(11,13);
        map.put(s, map.getOrDefault(s, 0) + 1);
    }
    return getDiskMax(map);
}
```

通过该模块对于日志数据的分析,我们获得的数据结果如下所示。如图4.37所示为不同地区用户对智能新风系统操作数量柱状图。图4.38为用户对于各个模块使用情况的频率饼状图。从上述两张图中我们得知，北京地区对于设备的操作频率最高，且操作最多的模块为风量模块。

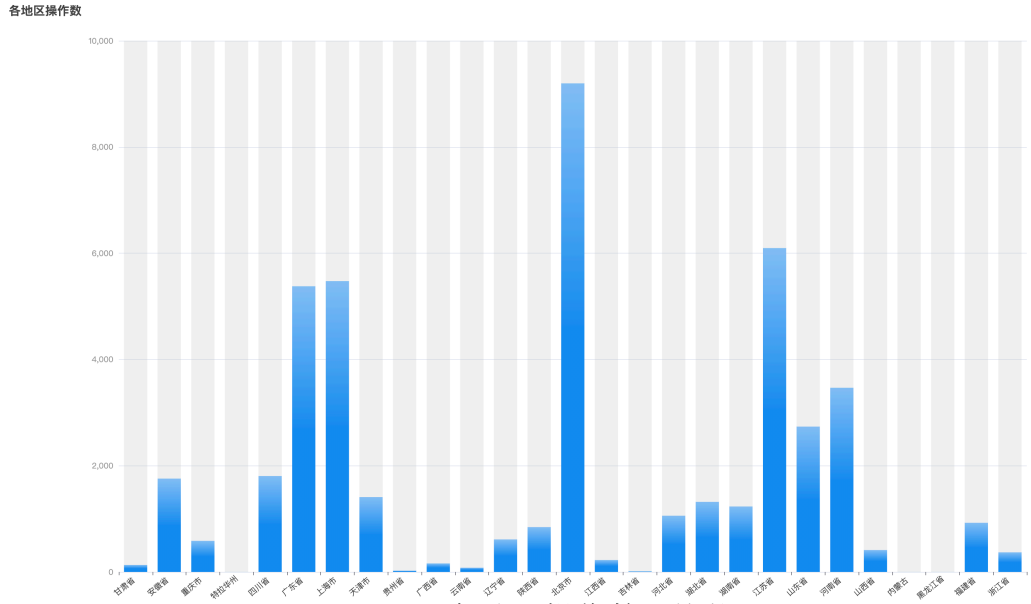


图 4.37: 各地区操作数量柱状图

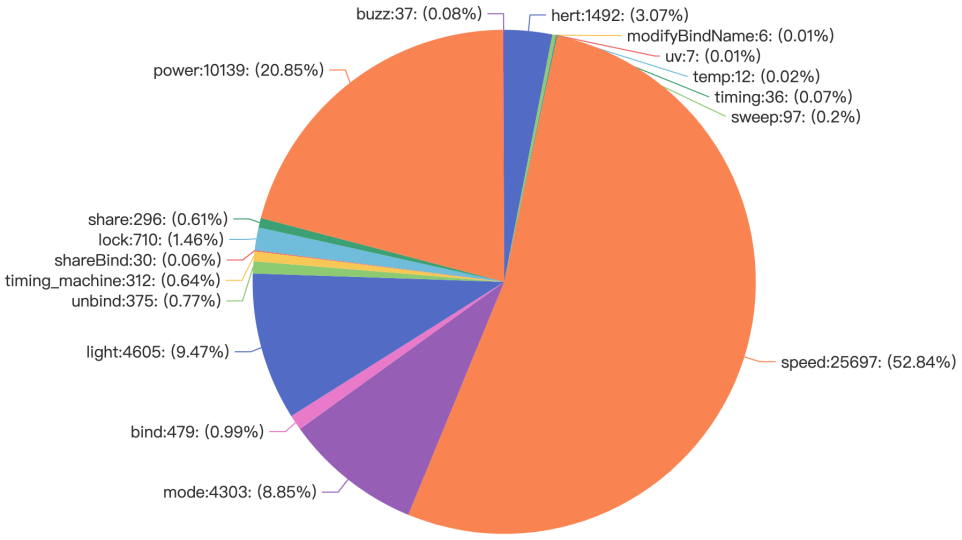


图 4.38: 各模块使用频率饼状图

## 4.7　本章小结

本章主要围绕着第三章中描述的额需求分析和概要设计介绍了场景构建技术的详细设计与实现，本章权限管理、设备控制、消息推送、场景构建、日志处理和数据分析将六个模块逐一拆解出多个功能，并详细介绍了各个功能的实现方式、关键代码以及功能效果展示。

# 第五章　智能新风设备场景构建系统的测试分析与实现

本章主要对面向智能新风设备的场景构建系统进行系统测试，主要内容有测试前准备、功能性以及非功能性测试。

## 5.1　测试准备

### 5.1.1　测试目标

在该系统正式部署运行前，需要对该系统的有效性进行测试。测试手段为为该系统设计测试用例并执行，在执行过程中检测系统的核心功能模块是否可以满足系统正常运行的需求，主要包括设备控制、场景添加、数据查看、设备入网、场景运行、消息推送、日志获取等功能点，通过执行测试任务并比较测试结果来衡量该系统的软件质量。

### 5.1.2　测试环境

如表5.1所示为系统功能测试环境配置信息，面向智能新风系统的场景构建技术在 macOS 环境下开发，并通过 Docker 容器部署在阿里云服务器集群中。后端微服务均通过 maven 构建打包部署，并通过 Nginx 配置反向代理为前端项目提供 HTTP 接口。前端以 APP 的形式安装在 Android 系统中。

表 5.1: 系统测试环境

| 字段 | 测试环境 |
| --- | --- |
| 服务器 | 阿里云服务器集群 |
| 部署容器 | Docker |
| 部署系统 | Linux、Android |
| 开发环境 | MacOS |
| 数据库 | Mysql 5.7.29、Redis 5.0.3、MongoDB |
| 测试软件 | Postman |
| 其他软件 | Nginx 1.15.7、Maven 3.3.9 |

## 5.2　功能性测试

本部分主要从设备添加、设备控制、场景添加、场景运行、数据查看和消息推送等方面展开测试，其中设备添加包括设备入网与设备绑定两个方面，消息推送包含应用程序消息推送和消息队列消息推送，能充分覆盖用例描述对应的功能点。

表 5.2: 设备添加测试用例

| 测试项 | 操作/输入 | 预期结果 | 测试结果 |
| --- | --- | --- | --- |
| 设备入网 | 1. 系统用户登录<br>2. 点击侧边栏或用户下拉框中添加按钮<br>3. 点击设备添加按钮，选择相机扫描或选择设备二维码<br>4. 重复选择二维码识别<br>5. 完善设备表单信息，点击确认按钮 | 1. 登录成功<br>2. 进入相机扫描页面或选择二维码页面<br>3. 识别成功后，界面提示识别成功<br>4. 系统提示已绑定该设备<br>5. 系统提示设备入网成功，并跳转至首页 | 通过 |
| 设备绑定 | 1. 点击侧边栏或用户下拉框中设备添加按钮<br>2. 点击设备添加按钮，选择相机扫描或选择设备二维码<br>3. 选择任意一张非设备二维码<br>4. 选择一张未配网的设备二维码<br>5. 完善设备表单信息，点击确认按钮 | 1. 进入相机扫描页面或选择二维码页面<br>2. 系统验证设备二维码后进入设备绑定页面<br>3. 系统提示二维码不合法<br>4. 系统提示设备未入网，并跳转至设备入网页面<br>5. 系统提示设备绑定成功，并返回首页 | 通过 |

表5.2描述了设备添加的测试用例，分别进行设备入网功能测试和设备绑定功能测试，其中覆盖对同一二维码和非法二维码扫描的功能测试，最终的测试结果显示符合预期测试结果。

表 5.3: 智能新风机设备控制功能测试用例

| 测试项 | 操作/输入 | 预期结果 | 测试结果 |
| --- | --- | --- | --- |
| 新风机<br>开关控制 | 1. 点击新风机信息卡片<br>2. 点击开关按钮 | 1. 进入新风机控制界面<br>2. 页面展示操作成功，设备开关状态发生改变 | 通过 |
| 新风机<br>风量控制 | 1. 点击新风机信息卡片<br>2. 拖动风量控制进度条 | 1. 进入新风机控制界面<br>2. 页面展示操作成功，设备风速发生改变 | 通过 |
| 新风机<br>屏幕亮度控制 | 1. 点击新风机信息卡片<br>2. 拖动屏幕亮度控制进度条 | 1. 进入新风机控制界面<br>2. 页面展示操作成功，设备屏幕亮度发生改变 | 通过 |
| 新风机<br>定时设置 | 1. 点击新风机信息卡片<br>2. 选择设备定时开关<br>3. 选择设备定时时长 | 1. 进入新风机控制界面<br>2. 设备定时开关打开，默认 30 分钟<br>3. 页面展示定时成功，定时时间 | 通过 |
| 新风机<br>模式切换 | 1. 点击新风机信息卡片<br>2. 点击模式按钮<br>3. 点击选择的模式 | 1. 显示新风机控制页面<br>2. 页面显示模式切换选择选项<br>3. 页面提示操作成功，设备模式状态发生变化 | 通过 |
| 新风机<br>空气质量<br>图片推送 | 1. 点击新风机信息卡片<br>2. 点击右上角空气图片按钮 | 1. 显示新风机控制页面<br>2. 页面显示加载信息，一定时间后用户微信会收到图片推送 | 通过 |
| 设备状态展示 | 点击首页新风机信息卡 | 展示设备收集到的 PM2.5、风速、温湿度等信息 | 通过 |

表5.3是新风机设备控制的测试用例，主要验证智能新风机设备的开关控制、模式切换、风量、屏幕亮度控制设备状态展示等功能的正确性，最终的测试结果显示符合预期测试结果。

表 5.4: 智能风扇设备控制功能测试用例

| 测试项 | 操作/输入 | 预期结果 | 测试结果 |
| --- | --- | --- | --- |
| 风扇<br>开关控制 | 1. 点击风扇信息卡片<br>2. 点击开关按钮 | 1. 进入风扇控制界面<br>2. 页面展示操作成功，设备开关状态发生改变 | 通过 |
| 风扇<br>模式切换 | 1. 点击风扇信息卡片<br>2. 点击模式按钮<br>3. 点击选择的模式 | 1. 显示风扇控制页面<br>2. 页面显示模式选择控制选项<br>3. 页面提示操作成功，设备模式状态发生变化 | 通过 |
| 风扇<br>定时设置 | 1. 点击风扇信息卡片<br>2. 选择设备定时开关<br>3. 选择设备定时时长 | 1. 进入新风机控制界面<br>2. 设备定时开关打开，默认 30 分钟<br>3. 页面展示定时成功，定时时间 | 通过 |
| 风扇<br>风量控制 | 1. 点击风扇信息卡片<br>2. 拖动风量控制进度条 | 1. 进入风扇控制界面<br>2. 页面展示操作成功，设备风速发生改变 | 通过 |
| 设备状态展示 | 点击风扇信息卡片 | 展示设备收集到的风速、温湿度等信息 | 通过 |

表5.4是智能风扇设备控制的测试用例，和智能新风机控制类似，主要验证智能风扇的开关控制、模式切换、设备状态展示等功能的正确性，最终的测试结果显示，对应测试用例通过，符合预期测试结果。

表5.5是场景创建功能测试的测试用例，主要验证创建场景的任务流程是否正常。测试任务包括场景创建、可操作设备及控制选项选择和场景信息更新等任务。创建场景时用户需填入场景名称等信息，用户需从已绑定的设备中指定待添加的设备。更新场景信息时，需要配置待更新的场景名称、操作等关键信息，最终的测试结果显示符合预期测试结果。

表 5.5: 场景创建测试用例

| 测试项 | 操作/输入 | 预期结果 | 测试结果 |
| --- | --- | --- | --- |
| 新建场景<br>创建任务 | 1. 选择首页右上角场景添加按钮<br>2. 选择自定义场景创建<br>3. 填入场景名称<br>4. 选择添加设备按钮<br>5. 从可添加设备选择待添加设备<br>6. 选择待添加的操作信息，点击确认按钮 | 1. 跳转到任务页面<br>2. 弹出自定义创建场景页面供用户填写<br>3. 系统显示用户填写的场景名称<br>4. 弹出用户已添加的设备列表供用户选择<br>5. 系统提示选择成功，场景创建页面显示用户选择的控制选项<br>6. 提示新建成功，进入首页显示场景列表 | 通过 |
| 更新场景信息 | 1. 选择首页已创建场景列表中的场景并点击<br>2. 输入待更新场景相关信息，如场景名称、场景内操作建议，点击确认按钮 | 1. 跳转到更新场景信息页面<br>2. 提示更新成功，进入首页场景列表界面 | 通过 |

表 5.6: 场景运行和数据查看测试用例

| 测试项 | 操作/输入 | 预期结果 | 测试结果 |
| --- | --- | --- | --- |
| 场景数据查看 | 1. 选择首页已创建场景并点击<br>2. 等待页面加载 | 1. 系统显示信息加载页面<br>2. 信息加载成功 | 通过 |
| 场景运行 | 选择首页已创建场景列表中的场景执行按钮并点击 | 系统显示执行成功，场景内设备行为状态发生变化 | 通过 |

表5.6是场景运行和数据查看功能测试的测试用例，主要验证场景运行和场景信息查看等任务的正确性，场景查看功能测试主要用于测试场景数据是否正确展示，场景运行测试用于检测场景内设备是否如预期指令正确执行，最终的测试结果显示符合预期测试结果。

表 5.7: 消息推送功能测试用例

| 测试项 | 操作/输入 | 预期结果 | 测试结果 |
| --- | --- | --- | --- |
| 消息队列<br>消息发送 | 1. 点击场景执行按钮<br>2. 进入消息队列后台控制页面<br>3. 设置相应的筛选条件 | 1. 页面展示操作成功，设备开关状态发生改变<br>2. 页面展示筛选结果，显示消息队列发送的数据 | 通过 |
| 应用通知推送<br>控制台发送 | 1. 进入应用消息推送后台管理界面<br>2. 设置要推送的消息<br>3. 点击推送按钮 | 1. 显示应用推送后台<br>2. 页面提示应用程序消息设置成功<br>3. 页面提示消息发送成功，终端设备出现通知 | 通过 |
| 应用通知推送<br>定时任务发送 | 代码中设置定时推送的时间和消息内容 | 等到规定时间时，终端设备显示通知 | 通过 |
| 空气质量图片<br>微信推送 | 1. 点击新风机信息卡片<br>2. 点击右上角按钮<br>3. 点击空气图片按钮 | 1. 进入新风机控制界面<br>2. 页面展示更多操作对话框<br>3. 等待数秒后，用户绑定微信收到空气图片推送 | 通过 |

表5.7是场景构建技术中消息推送功能测试的测试用例，主要验证场景构建技术中的消息推送流程是否正常。消息推送任务包括消息队列的异步消息推送、应用通知推送和微信消息推送等任务。其中应用通知推送包括控制台推送和定时任务推送两种推送模式。最终的测试结果显示符合预期测试结果。

## 5.3　非功能性测试

第三章中对该系统提出了一些非功能性需求，如系统可扩展性、并发性等，在本小节中设计了对应的测试用例以此来衡量该系统的可靠性、安全性等指标，

如表5.8所示。

表 5.8: 响应时间非功能测试用例

| 界面 | 平均响应时间 | 最大响应时间 | 最小响应时间 | 测试结果 |
|---|---|---|---|---|
| 获取验证码 | 4206ms | 7723ms | 868ms | 通过 |
| 用户登录 | 1459ms | 4964ms | 16ms | 通过 |
| 设备绑定 | 1142ms | 1821ms | 782ms | 通过 |
| 新风机<br>开关控制 | 114ms | 260ms | 82ms | 通过 |
| 新风机<br>其他设置控制 | 2453ms | 5040ms | 587ms | 通过 |
| 场景创建 | 4948ms | 8725ms | 340ms | 通过 |
| 场景数据查看 | 2712ms | 4750ms | 336ms | 通过 |
| 场景运行 | 1649ms | 3345ms | 30ms | 通过 |
| 日志查看 | 19257ms | 21042ms | 433ms | 通过 |

## 5.4　本章小结

本章主要对面向智能新风设备的场景构建系统进行了较为完成的测试，以保证系统上线时的稳定运行，主要从功能性测试和非功能性测试两个方面入手。本章主要介绍了该系统的测试目标、系统所处测试环境、系统测试用例的设计预计测试执行情况。通过对整个系统进行完整的功能测试以及非功能性测试，验证了整个场景构建系统的稳定与可靠。

# 第六章　总结与展望

## 6.1　总结

随着日常生活中物联网设备数量的快速增加以及智能家居在生活中的普及，对于物联网平台中的设备控制也提出了更高的要求，用户单一控制设备不光操作繁琐，更会消耗用户较多的时间。为了提升用户对于设备控制的体验，本文提出了一种面向智能新风系统的场景构建技术，并介绍了该平台的设计与实现过程。

本文首先介绍了相关的项目背景以及国内外研究现状，接着介绍了本文使用的跨平台应用开发技术 Flutter、异步消息处理技术、数据存储方案及容器部署等关键技术。在第三章与第四章对该系统进行了详细的需求分析，并阐述了对整个系统的设计思路，并以图文的方式将系统展示。为了让整个系统得意稳定运行，本文在第五章对整个系统进行了较为完整的功能性与非功能性测试。

本文的主要工作如下：

1. 结合智能家居等相关技术的发展情况，对整个场景构建系统进行了详细的需求设计，并依据需求对可能使用到的技术进行深入调研，主要包括跨平台应用开发框架、数据库管理系统以及 Docker 和 Docker-Compose 容器编排技术。

2. 在完成了对整个系统的总体规划后，本文详细分析了场景构建系统的相关需求，并结合系统测试用例图对需求进行可视化描述，并依据“4+1”视图理论对整个系统的逻辑、进程、开发、部署等环节进行描述。

3. 根据需求分析结果，确定系统功能点，并将系统模块划分为权限管理模块、消息推送模块、设备控制模块、场景构建模块、日志处理模块和数据分析模块六个模块，并针对各个模块实现各自完成了详细的功能设计。

本系统针对某智能新风设备在使用过程中产生的实际需求，为该系统内的智能新风设备提供了场景构建能力，支持单一操作实现多设备的联动控制，并为用户提供了直观的场景查看页面，减轻了用户对场景的管理难度。同时还提供了对设备数据的可视化分析。本人主要负责本技术的后端和前端部分的开发。

## 6.2 展望

本文初步实现了面向智能新风系统的场景构建技术，实现了场景构建中最重要的场景详情查看和场景运行功能，该系统虽然已经通过功能性测试，但在后续的维护中仍可从以下方面进行完善：

1. 目前的联动只是单一用户的多个设备间的联动，以后可以考虑根据一定设备内的多个用户的设备数据进行分析联动。
2. 加强系统的可扩展性，在出现新设备时，仍需要针对设备新增相应的 UI 及逻辑代码，并实现重新部署。此处可通过云端配置中心实现新设备的动态扩展。
3. 场景构建功能目前只支持用户自定义操作，在用户行为数据达到一定程度后可以实现针对不同用户的针对性的场景推荐。
4. 本系统端采用 Flutter 框架，虽然在一定程度上可以简化开发，但仍需要对各自的平台实现针对性适配，后续应完善对全平台的适配情况。

未来，本系统将针对上述点持续迭代更新，不断在可用性、可扩展性方面进行升级优化。

# 参考文献

[1] 张玉清, 周威, 彭安妮, 物联网安全综述, 计算机研究与发展 54 (10) (2017) 2130.

[2] X. Fan, B. Qiu, Y. Liu, H. Zhu, B. Han, Energy visualization for smart home, Energy Procedia 105 (2017) 2545–2548.

[3] 臧根林, 王亚强, 吴庆蓉, 占春丽, 李熠, 智慧城市知识图谱模型与本体构建方法, 大数据 6 (2) (2020) 0.

[4] 杨士元, 中国智能家居的技术发展趋势——清华大学自动化系杨土元教授在" 全国数字家居与平安社区高峰论坛" 上的演讲, 智能建筑 (08) (2006) 23–24.

[5] 夏吉品, 邹滨, 杨忠霖, 李沈鑫, 室外空气污染暴露规避健康路径规划服务, 计算机工程与应用 (3).

[6] 杨裕祯, 于莹莹, 王育珩, 王芳, 广藏列车空气质量及乘客主观舒适度的分析, 华南预防医学 (4).

[7] 黄海昆, 邓佳佳, 物联网网关技术与应用, 电信科学 26 (4) (2010) 20–24.

[8] H. Jiang, C. Cai, X. Ma, Y. Yang, J. Liu, Smart home based on wifi sensing: A survey, IEEE Access 6 (2018) 13317–13325.

[9] U. Hunkeler, H. L. Truong, A. Stanford-Clark, Mqtt-s—a publish/subscribe protocol for wireless sensor networks, in: 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08), IEEE, 2008, pp. 791–798.

[10] S. S. Gill, P. Garraghan, R. Buyya, Router: Fog enabled cloud based intelligent resource management approach for smart home iot devices, Journal of Systems and Software 154 (2019) 125–138.

[11] R. Piyare, S. R. Lee, Smart home-control and monitoring system using smart phone, ICCA, ASTL 24 (2013) 83–86.

[12] P. Mtshali, F. Khubisa, A smart home appliance control system for physically disabled people, in: 2019 Conference on Information Communications Technology and Society (ICTAS), IEEE, 2019, pp. 1–5.

[13] 田锋, 语音助手提升智能家居互动性, 中国公共安全 7.

[14] A. Gupta, R. Goyal, P. Kulshreshtha, A. Jain, Environmental and of delhi, co2 india in indoor monitoring office spaces of pm2. 5, Indoor Environmental Quality: Select Proceedings of the 1st ACIEQ 60 (2020) 67.

[15] J. D. Spengler, K. Sexton, Indoor air pollution: a public health perspective, Science 221 (4605) (1983) 9–17.

[16] R. Harper, Inside the smart home, Springer Science & Business Media, 2006.

[17] R. Johnson, J. Höller, A. Arendsen, T. Risberg, C. Sampaleanu, Professional Java Development with the Spring Framework, John Wiley & Sons, 2007.

[18] H. Suryotrisongko, D. P. Jayanto, A. Tjahyanto, Design and development of backend application for public complaint systems using microservice spring boot, Procedia Computer Science 124 (2017) 736–743.

[19] S. T. Pandeeswari, S. Padmavathi, N. Hemamalini, Engineering full stack iot systems with distributed processing architecture—software engineering challenges, architectures and tools, in: A Journey Towards Bio-inspired Techniques in Software Engineering, Springer, 2020, pp. 71–87.

[20] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, et al., The spring framework–reference documentation, interface 21 (2004) 27.

[21] C. M. Aderaldo, N. C. Mendonça, C. Pahl, P. Jamshidi, Benchmark requirements for microservices architecture research, in: 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), IEEE, 2017, pp. 8–13.

[22] L. Dagne, Flutter for cross-platform app and sdk development.

[23] M. L. Napoli, Beginning Flutter: A Hands on Guide to App Development, John Wiley & Sons, 2019.

[24] M. Mikolaj, Using flutter framework in multi-platform application implementation.

[25] W. Wu, React native vs flutter, cross-platforms mobile application frameworks.

[26] A. E. Fentaw, Cross platform mobile application development: a comparison study of react native vs flutter.

[27] J. Fayzullaev, Native-like cross-platform mobile development: Multi-os engine & kotlin native vs flutter.

[28] 王绪亮, 聂铁铮, 唐欣然, 黄菊, 李迪, 闫铭森, 刘畅, 流式数据处理的动态自适应缓存策略研究, 计算机科学 47 (11) 122–127.

[29] 余永城, 翁秋华, 段卿, 袁伟, Rabbitmq 在气象通信系统中的应用研究, 计算机技术与发展 030 (004) (2020) 216–220.

[30] 周悦芝, 张迪, 近端云计算: 后云计算时代的机遇与挑战, 计算机学报 42 (04) (2019) 3–26.

[31] X. Wu, H. Wang, Y. Xiong, Q. Qiang, G. Li, X. Jin, T. Qin, L. Wang, Sunve: Distributed message middleware towards heterogeneous database synchronization, in: 2018 10th International Conference on Advanced Infocomm Technology (ICAIT), IEEE, 2018, pp. 160–166.

[32] M. Widenius, D. Axmark, K. Arno, MySQL reference manual: documentation from the source, ” O’Reilly Media, Inc.”, 2002.

[33] 胡程, 叶枫, 一种高效的 flink 与 mongodb 连接中间件的研究与实现, 计算机工程与应用 055 (023) (2019) 64–69.

[34] 姚经纬, 杨福军, Redis 分布式缓存技术在 hadoop 平台上的应用, 计算机技术与发展 027 (006) (2017) 146–150.

[35] J. Han, E. Haihong, G. Le, J. Du, Survey on nosql database, in: 2011 6th international conference on pervasive computing and applications, IEEE, 2011, pp. 363–366.

[36] 陆志刚, 徐继伟, 黄涛, 基于分片复用的多版本容器镜像加载方法, 软件学报 v.31 (06) (2020) 295–308.

[37] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux journal 2014 (239) (2014) 2.

[38] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, B.-J. Kim, Performance comparison analysis of linux container and virtual machine for building cloud, Advanced Science and Technology Letters 66 (105-111) (2014) 2.

[39] 刘思尧, 李强, 李斌, 基于 docker 技术的容器隔离性研究, 软件 (4) (2015) 110–113.

[40] T. Combe, A. Martin, R. Di Pietro, To docker or not to docker: A security perspective, IEEE Cloud Computing 3 (5) (2016) 54–62.

[41] M. List, Using docker compose for the simple deployment of an integrated drug target screening platform, Journal of integrative bioinformatics 14 (2).

[42] 刘钱超, 董超群, 基于容器技术的软件测试优化研究, 计算机技术与发展 029 (004) (2019) 13–18.

[43] P. B. Kruchten, The 4+ 1 view model of architecture, IEEE software 12 (6) (1995) 42–50.

[44] 曹鹏飞, 李杰, 杨君, 云环境下实时消息推送服务构建研究, 计算机技术与发展 v.30;No.275 (03) (2020) 210–214.

# 简历与科研成果

**基本情况** 梁越勇，男，汉族，1999 年 11 月出生，江苏省南京市人。

**教育背景**

**2019.9 ~ 2021.6**　南京大学软件学院　硕士

**2015.9 ~ 2019.6**　江苏科技大学计算机学院　本科

**读研期间的成果（包括发表的论文及参与的专利）**

202110277811.6　一种面向智能新风系统的场景构建技术　南京普森斯信息科技有限公司　梁越勇，章许帆，刘嘉，吴清已受理

# 致　　谢

两年的工程硕士生活一晃而过，我的学生时代也终将落下帷幕，时光如白驹过隙，未来也不知能否还有重返校园的机会。在此论文完成之际，我想向硕士就读生涯中所有帮助和关心过我的老师、同学、朋友、亲人致以衷心的感谢。

我的论文指导老师刘老师，也是我硕士就读时的课程老师，研究生的这两年不仅在其课程中受益良多，在此次论文编写过程中也给予我很多建议和帮助，同时也感谢实验室的学长以及其他老师的耐心指导。尤其是章学长，毕设期间在内容编排、大纲拟定等方面更是给予了我细致入微的建议与指导，令我获益良多!

其次，我要感谢来自舍友以及实验室同窗们的关怀，感谢在项目进展的过程中遇到问题时有你们的探讨，在你们身上我感受到对技术的热爱，以及对完美的追求。正因为有你们的陪伴，我才可以在两年的硕士生涯中不断地解决接踵而至的问题，是你们，让这两年的生活变得更加充实有趣。

最后，我要感谢父母对我学习和生活无私的奉献和支持，在我情绪低沉时，对我悉心呵护开导，让我走出阴霾重拾信心，在我骄傲浮躁时，告诫我自律慎独，让我戒骄戒躁稳打稳扎，养育之恩无以为报! 同时，向论文评阅及论文答辩委员会的老师致以最诚挚的谢意，感谢各位老师的辛勤工作。

# 版权与原创性说明



本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

作者签名：

日期：2021 年 05 月 20 日