

TONGJI UNIVERSITY

毕业设计（论文）

课题名称 基于STM32的嵌入式智能家庭终端设计

副 标 题

学 院 　　　　电子与信息工程学院

专　　业 　　　　　　　自动化

学生姓名 　　　　　　　臧宬博

学　　号 　　　　　 1750843

指导教师 　　　　　　 余有灵

日　　期 　　　　　2021年6月3日

基于STM32的嵌入式家庭智能终端设计

摘要

家庭终端作为信息技术领域相互融合而产生的新型智能家居设备，正受到开发者与消费者越来越多的关注。然而，传统家庭终端设计中存在的控制范围小、部署困难、不支持多用户以及交互程序平台兼容性欠缺等问题依然影响着智能家庭终端的应用价值与市场潜力。因此，本设计开发了一款基于STM32的智能家庭终端系统，采用嵌入式设备配合服务器运行、同时以微信小程序作为交互手段的系统架构，提高了智能家庭终端的灵活性、易用性与可移植性，并使其具备了进一步扩展成为大型系统潜力。

本设计首先分析现有家庭智能终端设计方案中存在的问题，基于应用实际进行了系统架构设计，分别提出了STM32、服务器以及微信小程序三部分的架构与系统交互方式。接下来以STM32F4开发板为控制器，相应地选择了OV2640摄像头模块与DHT11温湿度传感器模块并进行硬件连接。之后，设计在嵌入式设备端为STM32载入μC/OS-II嵌入式操作系统与lwIP协议栈，并以此为基础开发了传感器捕获与网络传输功能；在服务器端采用了Flask框架作为Web服务器并设计了Socket服务器模块，实现了用户与设备管理以及数据中转功能；在交互应用程序端基于服务器的支持实现了用户登录、设备绑定与数据交互。设计最后进行了各子系统的单元测试与系统综合测试，发现各子系统运转正常并且能够正确协同，用户可以通过微信小程序进行登录、绑定设备、获取数据并对设备进行控制。

**关键词：**STM32，嵌入式，智能终端，系统设计

Design of Smart Embedded Domestic Terminal Based on STM32

ABSTRACT

Domestic terminals are newly invented home automation devices combining technology from multiple areas of information science. While they are receiving increasing attention of developers and customers, certain problems discovered in traditional designs including limited control range, difficulty in planting, lack of support for multiple-user situations and low compatibility still hinder the application value and potential market of domestic terminals. Therefore, this article comes up with a smart domestic terminal system based on STM32, which consists of embedded device, server and WeChat Miniprogram, improving the flexibility, user-friendliness and portability of the product, while allowing for rich scalability to larger system in future development.

This article firstly analyses the existing problems in present products, designs the system and module structures as well as the communication interface specifications. The project chooses STM32F4 as the micro-controller, correspondingly selecting OV2640 camera and DHT11 temperature and humidity sensor. Then, μC/OS-II operating system and lwIP TCP/IP Stack are transported to STM32, which are used as basic environments for the development of functions including sensor-data capturing and internet transmission. Flask framework is used by the server for Web services and a Socket server module is designed, realizing user-device management and data transfer. WeChat Miniprogram is developed for user login, registration and data interaction. Lastly, the article presents module tests and system integration test results, concluding that the sub-systems all function as anticipated and is able to coordinate correctly. The users are allowed to login and bind their devices so as to request collected data and control their devices.

**Key words:** STM32, embedded system, domestic terminal, system design

目录

摘要 I

ABSTRACT II

目录 III

1 绪 论 1

1.1 课题意义 1

1.2 研究现状 2

1.3 主要工作 6

2 架构设计 8

2.1 系统架构 8

2.1.1 嵌入式设备 8

2.1.2 服务器 9

2.1.3 交互应用程序 9

2.2 项目依赖 10

2.2.1 μC/OS-II操作系统 10

2.2.2 lwIP协议栈 11

2.2.3 Flask框架 12

2.2.4 微信小程序 13

2.3 本章小结 14

3 嵌入式硬件设计 15

3.1 硬件选型 15

3.1.1 微控制器 15

3.1.2 摄像头 17

3.1.3 温湿度传感器 17

3.2 硬件连接 18

3.3 本章小结 19

4 系统软件开发与模块协同 20

4.1 通信接口规范 20

4.1.1 嵌入式设备-服务器 20

4.1.2 服务器-交互应用程序 22

4.2 嵌入式设备开发 23

4.2.1 μC/OS-II与lwIP移植 23

4.2.2 HTTP与Socket网络通信 26

4.2.3 传感器数据捕获 28

4.3 Web服务器开发 31

4.3.1 设计模式 31

4.3.2 用户管理 32

4.3.3 设备校验 34

4.3.4 数据中转 34

4.4 Socket服务器开发 35

4.4.1 监听线程 36

4.4.2 连接可重入 38

4.4.2 数据缓存与并发处理 39

4.5 微信小程序开发 40

4.5.1 用户登录与设备绑定 41

4.5.2 数据交互 42

4.6 本章小结 43

5 项目测试 44

5.1 嵌入式设备测试 44

5.1.1 环境测试 44

5.1.2 功能测试 45

5.1.3 压力测试 46

5.2 服务器测试 47

5.2.1 基于pytest的测试环境配置 48

5.2.2 编写测试用例 49

5.2.3 测试结果 50

5.3 交互应用程序测试 50

5.3.1 编写模拟设备客户端 51

5.3.2 小程序功能测试 51

5.4 系统综合测试 53

5.5 本章小结 56

6 总结与展望 57

参考文献 59

谢 辞 61

附 录 62

1 绪 论

1.1 课题意义

一直以来，居家安全是日常生活中每个人都关注的重要话题。物联网普及之前，人们出门在外几乎无法实时掌握家中的情况，也无法及时对突发事件做出有效的反应，这在一定程度上为人们的日常生活带来了安全隐患。然而，随着信息科技的发展，各种新兴的技术正不断地涌入并填补这一领域的空白。

物联网时代到来后，传感技术、网络技术、人工智能技术和自动控制技术的进步与融合使得设备之间的对话成为可能，从而将物联网的应用带入了家居、物流、农业、交通、医疗等各个领域。同时，嵌入式系统作为物联网设备端的核心部件，因其低功耗、低成本、高效率、软件硬件可裁剪等优点得以迅速发展并成功占有物联网智能化市场。随着技术限制的逐步突破，层出不穷的新产品为广大消费者提高居家生活安全性和便利性提供了越来越多样化的选择。

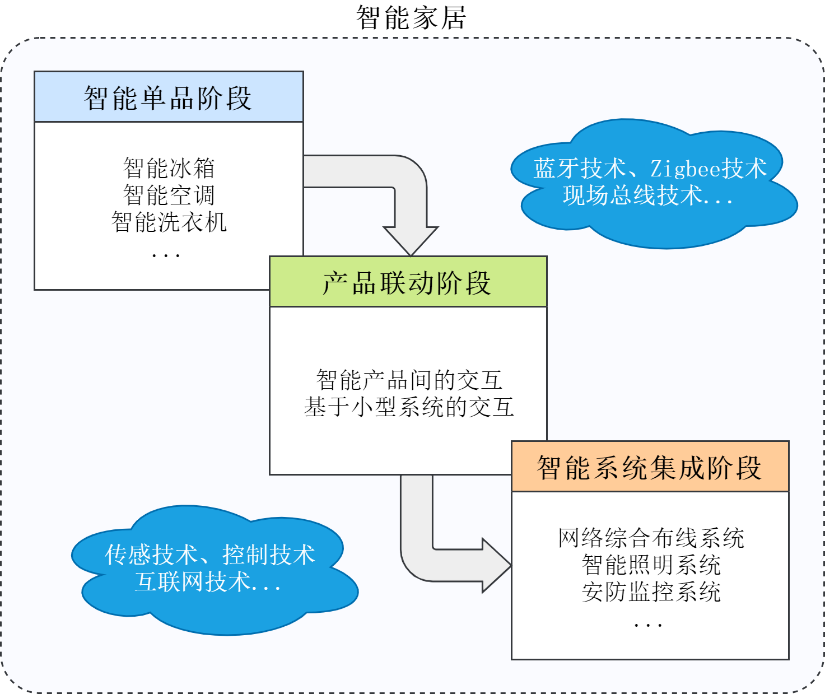
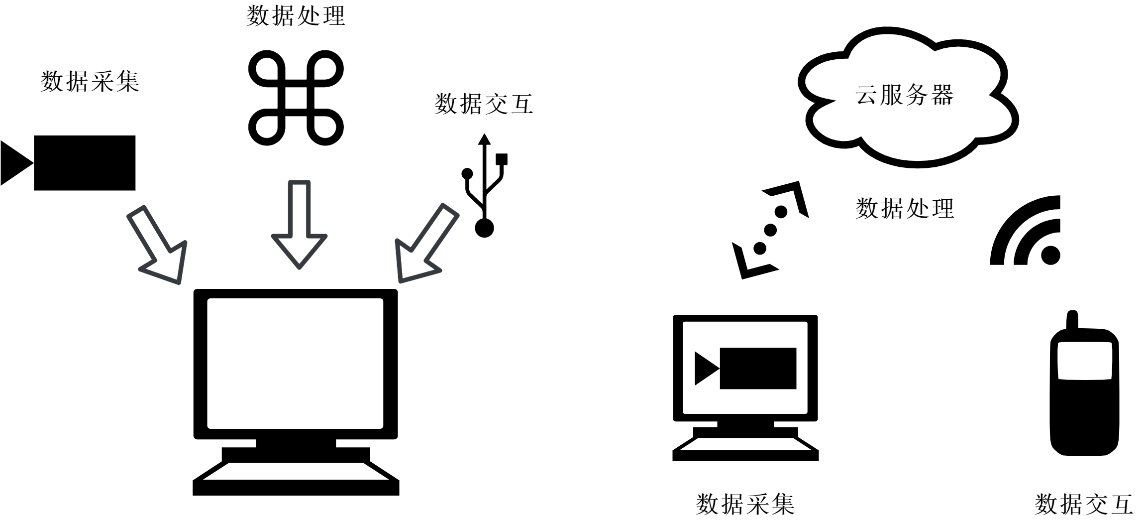


图1.1 智能家居系统发展历程­[1]

在这一背景下，综合考虑领域研究现状、市场需求、技术安全性以及可实现性，从而设计一款能够满足广大消费者对于居家安全与生活便利需求的嵌入式家庭智能终端是十分可行且有必要的。

1.2 研究现状

在现阶段，主流的家庭智能终端可以从系统构成上分为主要工作在本地的单体终端设备与需要协同服务器工作的综合终端系统。其中，单体终端倾向于将信息采集、处理与通信等功能集成于单一设备上，并采用如LCD触摸屏或独立的平板电脑等方式进行人机交互，同时也可以通过蓝牙等手段连接手机以便进行无线控制[2]。这样的设计部署简单，但却只能在设备附近操作，使用灵活性较差。另一方面，综合性终端系统在一般上可以分为嵌入式设备、服务器、交互应用程序三个较为独立的部分。嵌入式设备端采集所需要的数据并上报服务器，由服务器处理后中转至相应的移动端设备，并由交互程序进行显示[3]。



(a) 单体终端系统 (b) 综合终端系统

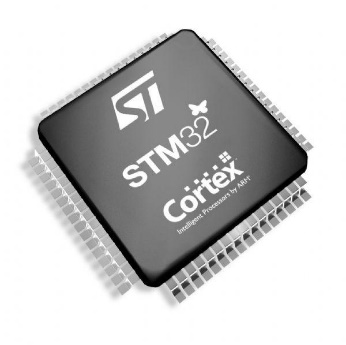
图1.2 常见家庭智能终端系统架构

尽管综合终端系统存在着结构复杂、开发难度较高等缺点，但相较之下这样的设计能够对设备进行真正意义上的远程监控。同时由于数据的采集与处理实现了分离，这种架构也减轻了嵌入式CPU的运算负担，为批量生产条件下进一步减少成本提供了可能。有关于嵌入式设备选型与通信方式、服务器设计方案以及交互应用程序开发路线等方面，现行的综合终端系统设计方案中已经有较多相关研究，具体内容如下。

A. 嵌入式设备选型

嵌入式设备端较为常见的控制器选择包括两种。第一种是树莓派[4]，其作为一个功能强大而全面的嵌入式开发平台，尽管能够完全满足家庭智能终端的设计需求，但由于其造价相对高昂，同时也造成了包括蓝牙、USB、音视频输入输出在内的大量硬件功能与CPU算力的冗余，因此并不完全适用于此种环境。另一种选择是STM32，开发者可以充分利用其高度的可定制性与丰富的硬件接口[5]。

具体而言，F1系列是小型的终端设备中经常被选择的型号，其在价格低廉的同时可以轻松胜任环境参数的检测、少量数据的网络传输与简单的图像处理任务[6]；而在较为大型的终端设备中更为常见的是F4系列，其具有更强的运算能力、更快的处理速度、更丰富的对于网络功能支持以及更完备的硬件设备接口（包括专门的以太网口与数字摄像头接口），同时还集成了部分DSP指令与FPU运算单元，因而能够进行更为高级的图像处理与网络通信任务，也可以轻松的加载诸如μC/OS[7]、FreeRTOS[8]、μCLinux等嵌入式操作系统以支持更为复杂的任务调度，因而是一种非常贴合应用实际的选择。

(a) 树莓派 (b) STM32

图1.3 STM32与树莓派

表1.1 STM32F1、F4与树莓派的性能比较（部分）

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | STM32F103 | STM32F407 | 树莓派-4B |
| CPU | Arm Cortex-M3 | Arm Cortex-M4 | Arm Cortex-A |
| GPU | - | - | 400MHz VedioCore VI |
| 存储 | 64k+512k | 192k+1024k | 4GB |
| 接口 | GPIO | SDIO、DCMI、RMII | HDMI、USB Type-C |
| 裸机价格 | 30-100 | 100-300 | 500-800 |

然而需要注意的是，STM32芯片本身作为一款SoC，其与已经集成了外围电路支持以及各项硬件接口的树莓派开发板存在较大区别。在实际应用中，若想实现STM32芯片与其他传感器模块的连接，则需要另行完成电子线的路设计与焊接或选用市面上已经集成好的开发板，常见厂商包括正点原子、野火等。

B. 设备通信方式

对于综合终端系统，嵌入式设备与系统其他部分之间需要进行大量的信息交互，因此必须有针对性地选用合适的通信方式。

现有设计中，串口与USB等基于有线方式的通信协议经常被应用于小型的嵌入式系统。其中，串口协议实现过程最为简洁、成本最低，但由于其传输距离有限且速率很慢，因此一般仅用于设备调试信息输出而不涉及任何业务数据；USB协议的传输速率较快，支持的数据类型也更为丰富，但其在嵌入式设备端的实现相对较为复杂，且由于没有专用的IO芯片支持，通信时需要占用一定的嵌入式CPU资源[9]。

而在另一方面，基于无线方式的通信协议如GPRS、蓝牙以及Wifi等也在现有设计中被经常选用，其最大的优点在于可以摆脱接线的限制，使得设备的部署方式更为灵活。然而考虑到此类协议的实现较为复杂，因而一般集成于专门的硬件模块，从而导致了设计成本的增加。与此同时，相较于协议本身的通信原理，模块与控制器之间的连接方式将在更大程度上限制通信速率。例如市面上最常见的ESP8266串口Wifi模块采用USART与控制器进行连接，其传输速率远低于Wifi协所允许的传输速率，无法满足较为复杂的网络通信需求。另外，由于此类模块一般倾向于将TCP/IP协议栈集成于模块内部，对外仅暴露高度封装的后接口用于数据交互（例如ESP8266模块仅支持TCP与UDP两种通信方式），因此使用者一般无法自行决定应用程序工作于协议栈的何种层级，其灵活性往往较低。



图1.4 ESP8266串口Wifi模块

一种折中的方案是选择以太网作为设备通信协议。尽管设备需要通过网线才能连接至路由器，但从路由器到服务器的数据传输即可完全交由网络负责，避免了如USB等现场总线协议需要直接连接到服务器硬件接口的情况。此外，由于大部分嵌入式设备上的以太网口仅工作于链路层，用户可以自行配置协议栈，并根据实际情况选用合适层级的接口实现网络通信功能[10]。给出常用设备通信协议的比较情况如下表。

表1.2 嵌入式设备常用通信协议比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 协议 | 连接方式 | 最大速率/bps | 最大距离/m | 备注 |
| USART | 有线 | 20K | 1200 | RS232接口 |
| USB | 有线 | 480M | 3-5 | USB 2.0 |
| BLE | 无线 | 1M | 5-10 | 无障碍条件 |
| Wi-Fi | 无线 | 54M | 100-200 | 2.4G频段 |
| Ethernet | 有线 | 10/100M | 100 | 双绞线 |

在一些智能家居系统的设计中，对于终端设备会采用集群模式，例如在厨房、卧室、客厅等不同的房间中安放多个终端设备，并借助ZigBee技术实现同一组网下不同终端之间的信息交互[11]。值得一提的是，此种大型系统虽然具有更加丰富完善的功能，但同时也有着结构复杂、部署困难、维护成本高等诸多缺点。此外，类似系统所包含的如电动窗帘、灯光调节与家电控制等部分功能的实现需要依赖于完全配套的被控设备[12]，在现实生活中实则难以运用。因此在一般情形下，由单体设备构成的家庭智能终端其实在很大程度上已经能够满足人们日常生活中的需求。

C. 服务器设计方案

在传统的家庭智能终端系统中，服务器更多的只是作为数据中转的途径。对于一些轻量级的应用程序，可以直接使用市面上现成的物联网服务器如阿里云或OneNet等，通过已经集成好的Api进行数据交互，并使用诸如网页组件等技术实现数据的可视化与对设备的控制[13]。然而在通常情况下，运营商为了保证服务器面对简单应用需求时的使用友好性，一般会对相应的接口进行高度封装以保证简洁，随之而来的问题便是极大地削弱了服务器功能的灵活性与可定制性。例如OneNet平台不支持文件操作、不暴露数据库接口、也无法提供除网页组件和电子邮件之外任何方式的人机交互，更无法在服务器端进行个性化的运算或数据处理，因而也很难满足广大用户与开发者多样化的需要。



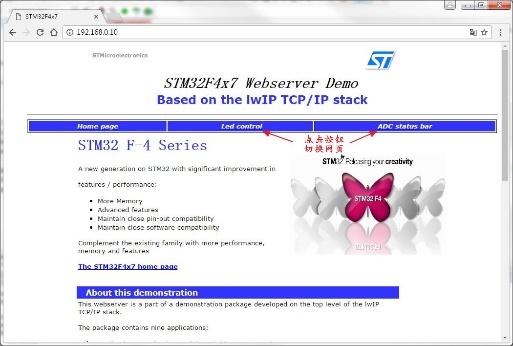
图1.5 OneNet平台

对于该问题的一种解决方案是由开发者自行搭建服务器，例如可以在设备端将采集到的传感器数据通过Socket传送至Apache服务器并暂存，接着由移动端程序使用HTTP请求获取数据并反馈给用户[14]。这样的设计好处在于能够根据项目需求有目标的进行服务器功能的开发，并结合实际情况选用合适的通信协议与接口实现方式。值得注意的是，在许多现有的基于类似架构的设计中，服务器与设备及用户程序之间的通信一般都是一对一的，即一台服务器在同一时刻只能为单一用户提供服务[15]，这显然是对服务器资源的极大浪费。在真正的项目部署与商用化过程中不可避免地需要应对多用户并发的情形，而这一功能的实现需要解决的主要问题在于用户的登录与鉴权，但该功能的设计与实现显然会造成开发成本的大幅度提高并且可能为系统带来更多的安全风险。

在采用多用户机制的系统设计中，用户与设备通过网络进行交互时的数据安全问题是必须加以考虑的，尤其是对于综合性的终端系统，如何保证私密数据在设备到服务器、服务器到交互程序的传输过程中不被窃取，以及如何保证位于服务器端的用户登录与鉴权机制没有明显漏洞是整个系统设计与开发过程中十分重要的一环[16]。由于任何数据在网络传输过程中都无法完全保证不被窃听，因而需要做的首先便是在嵌入式设备端发起传输之前对需要发送的数据进行加密[17]。其次，还应该在系统设计时预估可能存在的安全风险并采取相应规避措施，如恰当的设计系统不同部分之间的信息交互方式、恰当的选择更为安全的网络通信协议如HTTPS、以及对服务器进行SSL安全认证等。然而，考虑到真实应用场景下的网络信息安全是一项极为复杂的系统性工程，因此该部分的相关内容在本设计中仅做参考而不进行过多的研究。

D. 交互应用程序选择

对于交互应用程序，这一部分的开发在现有的设计方案中并没有受到太多的重视。最常见的移动端交互方案是基于安卓系统的手机App，程序采用Java语言编写并通过服务器与嵌入式设备进行数据与指令的沟通[18]。但考虑到该程序无法运行在装有iOS或Windows系统的移动设备上，而是需要在移植时针对不同操作环境进行专门的设计，开发过程较为繁琐。

(a) 基于安卓App (b) 基于Web页面

图1.6 常见人机交互方式

相较之下，一个兼容性更好的选择是使用Web前端技术[19]，通过浏览器内核屏蔽移动终端在硬件与操作系统层面的差异，使得同样的程序能够以更小的代价运行在更多的设备上。除了常见的基于浏览器网页的交互方式，WebView技术的发展使得前端技术可以在原生App环境的依托下被应用于更加广泛的情形，例如微信小程序便是一个十分成功的实践。

1.3 主要工作

综合上述讨论，本课题旨在设计一款部署简单、使用便捷、功能丰富而不冗余、移动设备兼容性强、面向多用户且数据安全性较高的嵌入式家庭智能终端系统，能够在嵌入式设备端实现环境参数检测与视频监控等功能；并且在服务器端进行数据中转以及设备与用户的注册、登录与鉴权，以此保证使用时的数据安全；同时开发一款跨平台的移动端交互程序，使不同终端设备的用户均能够方便的通过该程序获取监测状态。本文具体章节安排如下。

第一章首先介绍智能家居系统的发展现状，分析了现有设计中系统各部分存在的问题并提出了研究方向。

第二章介绍该智能终端系统的架构设计，依照本课题所提出的设计目标制定系统组成、划分模块功能并介绍了相关依赖。

第三章介绍系统硬件，阐述嵌入式端设备硬件组成，确定控制器、摄像头及温湿度传感器等模块的选型及器件特点，介绍传感器与控制器的通信方式并进行实物连接。

第四章介绍软件设计方案，确定了基于Socket与HTTP协议的网络接口规范，给出嵌入式设备数据采集与上报、服务器用户设备管理与数据中转、交互应用程序显示与控制三部分功能的设计实现，完成上述网络通信接口规范下各模块之间的交互协同策略。

第五章介绍项目测试方案，对本设计所包含的功能进行从模块级别到系统级别的测试覆盖，之后开展系统综合测试并给出最终评价指标。

第六章对项目设计与开发过程进行回顾总结，分析项目开发与测试过程中尚存的问题并提出可能的改进方向。

根据前文提出的项目设计目标与相关现有研究成果，结合实际情况给出该设计的各项评价指标如下表所示。

表1.3 设计评价指标

|  |  |  |
| --- | --- | --- |
| 模块 | 预期效果 | 指标 |
| 嵌入式设备 | 使用STM32实现环境参数的检测、采集与上报，接受来自用户的控制指令 | 移植μC/OS-II与lwIP  实现HTTP与Socket通信  实现摄像头与温湿度传感器捕获  实现用户控制指令响应 |
| 服务器 | 使用Flask框架并配置Socket，实现设备数据采集、用户管理与数据中转 | 搭建基于Flask的Web服务器  搭建Socket服务器  实现用户管理与设备校验  实现数据中转 |
| 交互程序 | 使用微信小程序连接服务器，实现用户鉴权与数据可视化 | 实现用户登录与设备绑定  实现数据获取与设备控制 |

2 架构设计

2.1 系统架构

本设计将该家庭智能终端系统分为嵌入式设备、交互应用程序以及服务器三个部分。其中，嵌入式设备负责采集数据并上传至服务器，同时接受来自用户的控制指令；服务器进行用户与设备的身份管理，将设备上传的数据暂存并反馈给移动端；交互应用程序负责从服务器获取相应数据并进行显示，该程序同时携带用户身份信息，以便在服务器端进行登录。

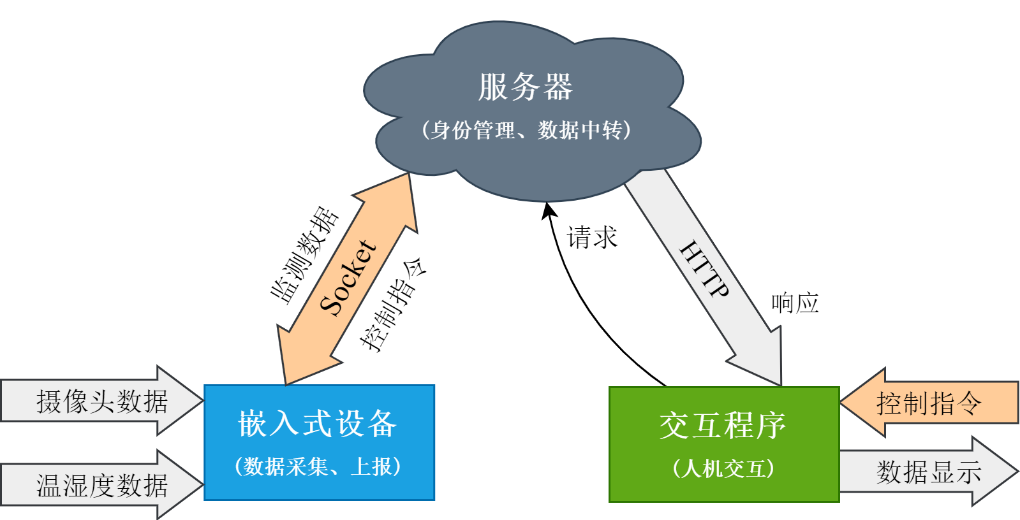


图2.1 系统架构图

2.1.1 嵌入式设备

对于嵌入式设备端，本设计选用STM32F4系列作为控制器，在价格相对低廉的前提下保证了运算速度，同时也可以利用现成的DCMI与1WireDQ等接口方便的连接摄像头与温湿度传感器等硬件设备。所有外设均通过GPIO与控制器连接。

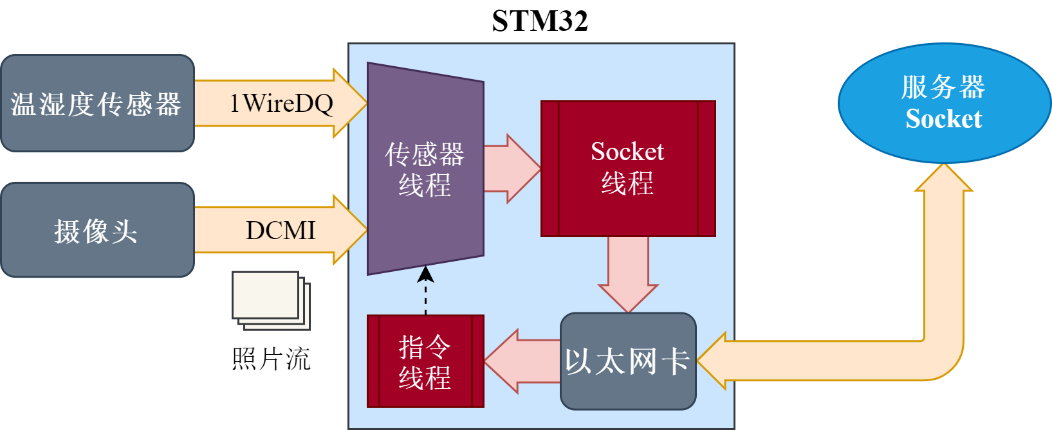


图2.2 嵌入式设备架构图

在通信方式的选择方面，由于设备不仅需要进行检测数据的实时上报，还需要时刻等待并响应通过服务器传来的用户指令。在此种情况下，采用Socket进行通信是一种较为合适的选择，它既实现了服务器与客户端的全双工通信机制，保证了信息交互的灵活性和时效性，也在最大程度上减少了不必要的流量开销。

2.1.2 服务器

为配合嵌入式设备与交互应用程序的正常工作，本设计所采用的服务器需要实现对于Socket通信与HTTP请求的支持。设计选用Flask框架作为Web服务器，程序全部使用Python编写，并在此基础上自行配置Socket服务器的相关功能，同时连接SQLite数据库以支持用户信息在服务器端的访问与存储。

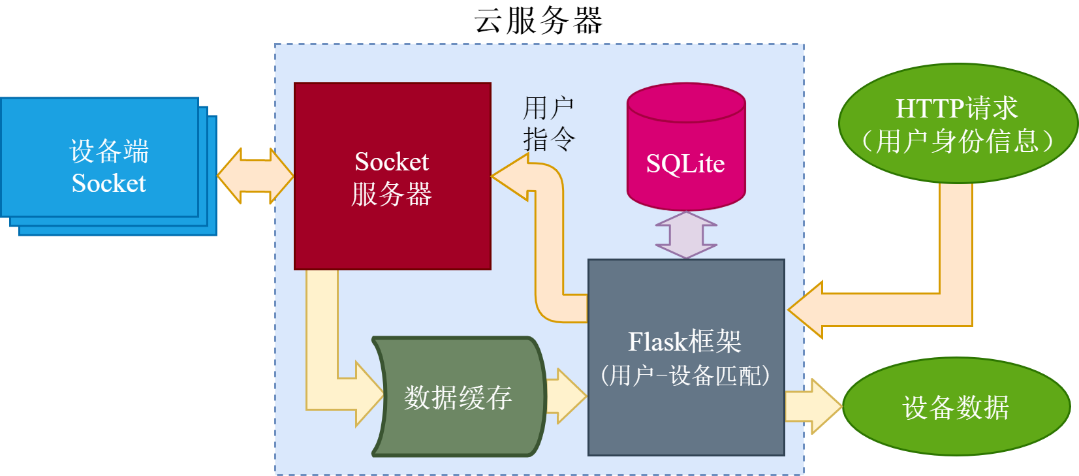


图2.3 云服务器架构图

用户通过向服务器发起HTTP请求进行登录、设备绑定、数据提取与控制指令上传等操作，在此过程中服务器将通过Socket接收到的对应设备的数据反馈给发起请求的用户。这样的设计能够屏蔽交互程序与硬件设备的直接通信，在加强了服务器对系统运行状态掌握的同时，也进一步保证了灵活性与用户数据的安全性。

2.1.3 交互应用程序



图2.4 微信小程序架构图

由于移动端原生App的开发必须依赖于相应的手机操作系统，而为了使交互程序具有更好的兼容性，本设计采用微信小程序作为交互手段，借助WebView技术使程序能够轻易的运行在所有安装有微信的手机平台上。微信小程序基于前端技术路线，通过类似于网页开发的组件绑定事件响应机制与用户进行交互，并利用以HTTP为主的网络传输协议与服务器通信。

同直接开发原生App或设计网页相比，采用微信小程序不仅能够通过现成组件的使用极大地减少软件开发成本，同时也能依托微信平台提供的开放能力辅助实现用户登录与用户身份鉴别等功能，从而能够为设计带来更好的兼容性、便捷性、安全性以及更加丰富的功能。

2.2 项目依赖

为充分利用现有研究成果并减少不必要的工作量，本设计在开发过程中借助了部分开源模块与框架，主要包括嵌入式设备端的μC/OS-II操作系统与lwIP协议栈、Web服务器框架Flask以及微信小程序框架。

2.2.1 μC/OS-II操作系统

由于控制器需要同时执行温湿度检测、摄像拍照、网络传输以及用户指令响应等多项任务，尤其是网络传输部分的开销相对较大、流程相对复杂、传输发生的时间也相对较为灵活，仅依靠单线程式的简单程序逻辑实现存在一定困难。因此，本设计载入了μC/OS-II嵌入式操作系统以提供更为高级的任务调度机制，特别是能够对于较为复杂的网络通信需求实现更好的支持。作为系统硬件层与应用程序软件的接口，μC/OS-II源码主要由三个部分组成。

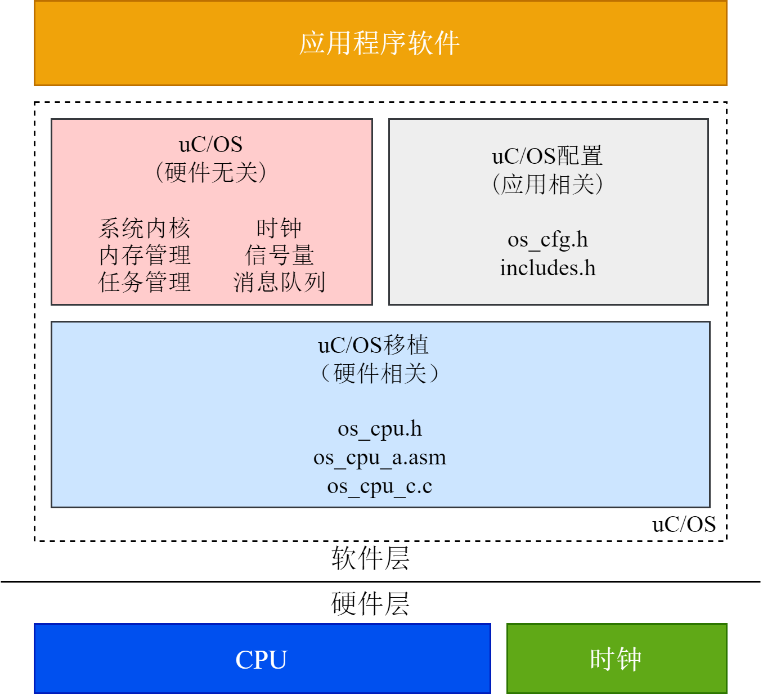


图2.5 μC/OS-II系统内核架构

首先是CPU硬件相关的部分。该部分由一个汇编文件（os\_cpu\_a.asm，负责任务切换的具体实现）、一个源文件（os\_cpu.c，负责用户Hook函数的定义）以及一个头文件（os\_cpu.h，负责数据类型的定义）组成，也是操作系统移植时唯一需要修改的部分。同时，系统依靠硬件提供的Systick定时器提供时钟节拍

其次是硬件无关的内核部分。该部分定义了操作系统的核心功能，包括系统标志位、消息邮箱、内存分配、互斥锁、消息队列、信号量、任务管理、系统时钟与软件定时器等。作为一个可以基于ROM运行的抢占式实时多任务内核，系统内核部分代码全部采用ANSI C编写，具有高度的可移植性。

最后是配置文件部分。该部分由两个头文件组成，其中os\_config.h中通过宏定义结合条件编译机制实现了系统功能裁剪，保证只有在配置文件中被用户选定的功能才会被最终编译并烧录，有助于减小系统内核体积；而includes.h负责向工程中的其他文件提供操作系统相关的变量与接口函数声明。

2.2.2 lwIP协议栈

在网络通信方面，设备提供了专用的以太网口并挂接网卡芯片。但由于该芯片仅工作于链路层，因此应用层网络通信功能的实现需要移植TCP/IP协议栈。本设计选用由瑞典计算机科学学院开发的轻量级TCP/IP协议栈lwIP，并加入了操作系统接口以更好地支持多线程，极为适合小型嵌入式设备对于较复杂网络通信的需求。

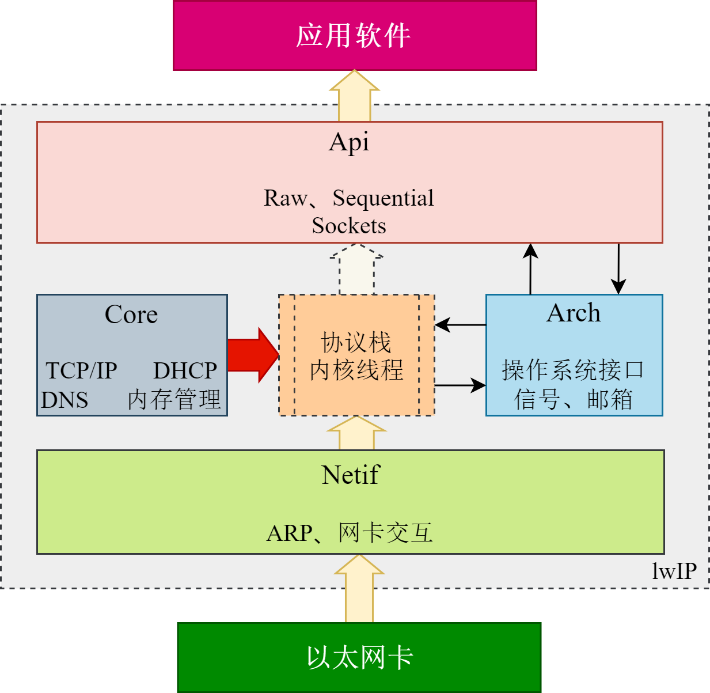


图2.6 lwIP协议栈架构

为基于开发板以太网口实现对于应用层网络通信功能的支持，本设计移植了lwIP-1.4.1，并加入对操作系统的支持以提供更好的多线程性能。该协议栈主要工作于网络层与传输层，其源码部分由网络接口层、内核层、系统架构层与Api层组成，并向上提供了Raw、Sequential与Socket三种Api。

在以太网口接收到数据之后，网络接口层负责进行协议栈与以太网卡间的数据交互并实现ARP协议，之后将数据包上交内核层进行处理。内核层集成了TCP/IP协议族，同时也负责实现DHCP、DNS以及网络数据内存管理。在支持操作系统的lwIP版本中，考虑到应用层的网络通信任务可能是多线程的，因此为了实现更好的并发支持，协议栈不允许应用软件直接与内核层通信，而是在内核层与应用层之间设计了系统架构层，并在其中对操作系统提供的信号量与消息邮箱进行了封装，从而借助内核任务调度与通信机制来支持多线程。

协议栈在初始化时会向操作系统注册一个优先级很高的网络功能内核线程，该线程负责实现包括TCP与IP协议在内的所有核层功能。当应用层软件发起网络通信时，程序将通过协议栈提供的Api调用系统架构层接口，使用信号量与消息邮箱将数据包下发给内核线程。之后，内核线程将依次处理每个通信任务，并同样通过信号量与消息邮箱向应用层线程返回数据。这样的架构设计在保证并发支持的同时并解决了协议栈的线程安全问题，同时也避免了应用软件层出现意外阻塞而影响系统正常的任务调度。

2.2.3 Flask框架

本设计中Web服务器的相关功能基于Flask框架实现。这是一个使用Python编写的轻量级WSGI应用框架，底层由Werkzeug套件与Jinja2引擎支持，在简单易用的同时也具有高度的可定制性与可扩展性。Flask框架处理 HTTP请求的基本方式为将一个视图函数分配给一个URL，在用户访问这个URL时Flask将识别并调用该函数。

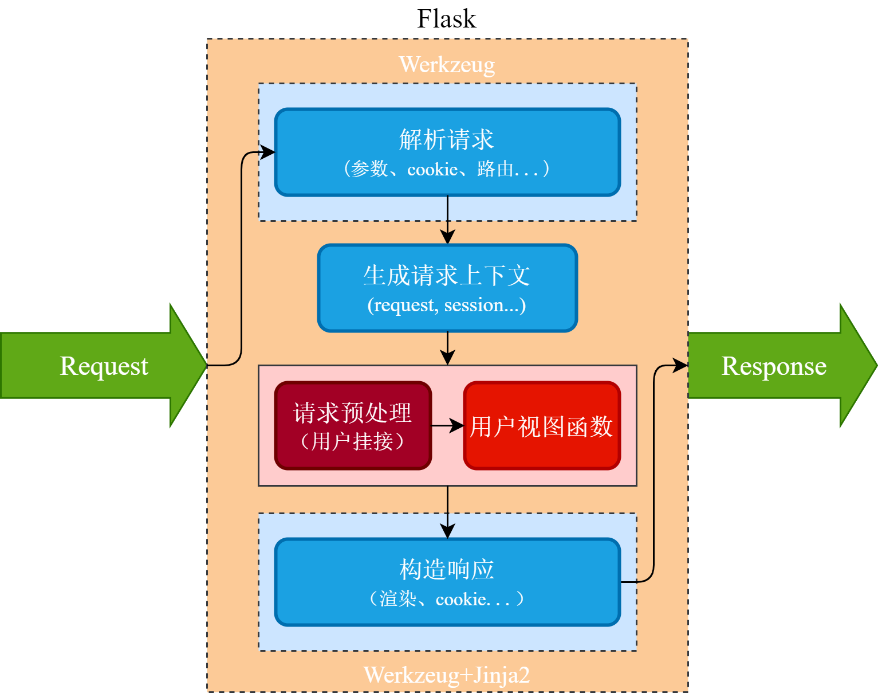


图2.7 Flask框架工作流程

当服务器接收到一个请求时，Flask将首先调用网关接口Werkzeug对请求数据进行预处理，解析得到该请求所包含的参数、cookie与路由等信息。接着，Flask框架将为当前请求生成上下文，包括构造请求对象以及还原会话等，之后便可调用视图函数对请求进行具体的处理。最后，Flask框架再次使用Werkeug与Jinja2将视图函数的返回值渲染为一个合法的HTTP响应，自动为其添加cookie等字段的信息并将其返回给客户端。

为确保灵活性，框架一般不指定数据库与模板引擎对象，相关功能可由用户根据需求自行配置。本设计在数据库方面选用SQLite3，有关数据库接口的具体内容见4.2.2章节。另外，由于客户端模板文件（WXML）的定义与渲染已经包含在微信小程序框架之内，服务器面对请求只需要返回业务数据而不需要生成整个模板文档，这也在一定程度上减轻了服务器端的负担。

2.2.4 微信小程序

微信小程序的代码部分主要由四种文件构成，分别为JSON配置（负责配置页面路径，窗口参数，工程环境等）、WXML模板（负责定义页面结构，类似HTML）、WXSS样式（负责定义页面样式，类似CSS）以及JS逻辑交互（负责响应用户操作）构成。

相比于普通的网页前端应用，微信小程序工作在微信客户端提供的宿主环境内，包括渲染层与逻辑层两部分。当小程序启动时，微信客户端将整个小程序代码包下载到本地，根据全局配置文件中的页面信息注册一个页面实例并根据JSON配置生成界面，之后装载WXML模板与WXSS样式，最后加载JS进行页面渲染。

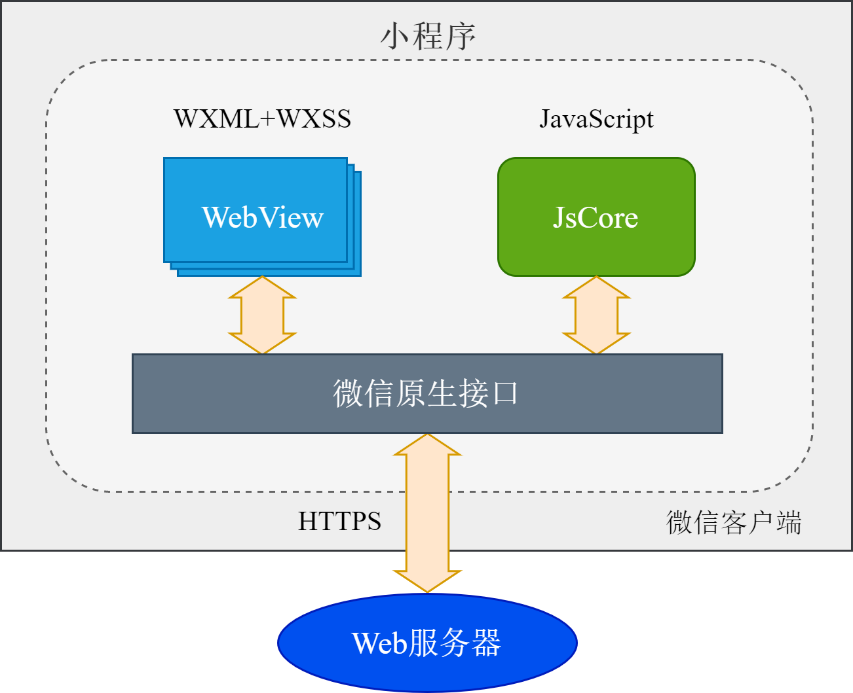


图2.8 小程序通信模型

在小程序中，渲染层和逻辑层分别由两个独立的线程管理。渲染层的界面使用WebView进行控制，而逻辑层则交由JsCore线程负责。由于一个小程序存在多个界面，所以渲染层将同时运行多个视图线程，视图线程与逻辑线程之间的通信以及所有网络请求均会经由微信客户端中转。在微信平台的依托下，开发者可以通过JS方便的调用微信客户端或操作系统提供的Api进行事件监听、页面切换、发起网络请求与展示媒体，还可以进行登录、支付、转发、地图定位以及生物验证等操作。

本设计在小程序开发过程中用到的部分Api如下表所示。

表2.1 小程序部分Api列表

|  |  |
| --- | --- |
| 名称 | 功能 |
| setTimeout/setInterval | 设置一个定时器并在到期后/按周期执行回调函数 |
| clearTimeout/clearInterval | 清除一个定时器 |
| wx.redirectTo | 销毁当前页面并跳转到其他页面 |
| wx.navigateTo | 保留当前页面并跳转到其他页面 |
| wx.request | 发起HTTP（HTTPS）网络请求 |
| wx.downloadFile | 下载文件资源到本地并暂存 |
| wx.getStorageSync | 将数据键值对以同步方式存储在本地缓存中 |
| wx.login | 获取临时登录凭证code |
| wx.getUserProfile | 获取用户信息 |
| wx.authorize | 向用户就某项功能发起授权请求 |

2.3 本章小结

本章主要进行系统架构设计，给出了嵌入式设备、服务器以及交互应用程序的内部架构与整体工作流程，之后针对设计开发过程中依赖的外部开源模块与框架结合本设计实际使用情况进行了简单的介绍。

3 嵌入式硬件设计

3.1 硬件选型

本章主要阐述系统在嵌入式设备端的开发过程中硬件部分的相关内容。嵌入式设备包含的主要硬件模块BOM清单如下。

表3.1 硬件模块BOM清单

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 名称 | 规格 | 数量 | 单价/元 |
| 1 | 微控制器 | STM32F4 | 1 | 448 |
| 2 | 摄像头模块 | OV2640 | 1 | 98 |
| 3 | 温湿度传感器模块 | DHT11 | 1 | 10 |
| 4 | 调试器 | ST-Link | 1 | 78 |

3.1.1 微控制器



图3.1 STM32F4探索者开发板

由于本设计在嵌入式设备端需要载入操作系统并实现较为复杂的传感器检测与较高性能的网络传输任务，因而对处理器的运算能力与硬件功能支持提出了更高的要求。设计选用了正点原子公司STM32F4探索者开发板，该设备的MCU采用STM32F407VGT6，处理器为带FPU与DSP指令的ARM Cortex-M4内核，包含12个16位定时器，2个32位定时器，2个DMA控制器，以及SDIO、FSMC、10/100M以太网MAC控制器和DCMI等。正点原子公司在此基础上进行了外围电路设计与并添加了硬件接口，使得该开发板能够在很大程度上发掘处理器的潜力。本设计中用到的部分硬件资源如下表所示。

表3.2 开发板部分硬件资源

|  |  |
| --- | --- |
| 资源 | 型号说明 |
| 处理器 | STM32F407VGT6、1M Flash、192k SRAM |
| 存储设备 | 外扩1M SRAM，1M Flash |
| 硬件接口 | 以太网口、摄像头接口、DHT11接口 |
| 其他 | LED、按键、JTAG/SWD、USB转TTL串口、12V稳压电源 |

为了实现网络通信功能，MCU集成了MAC 802.3介质访问控制器，通过RMMI与外部PHY芯片通信并提供了独立的DMA控制器与收发缓冲区。开发板在此基础上挂接了LAN8720A以太网卡芯片，能够连接网线并支持链路层的网络通信。

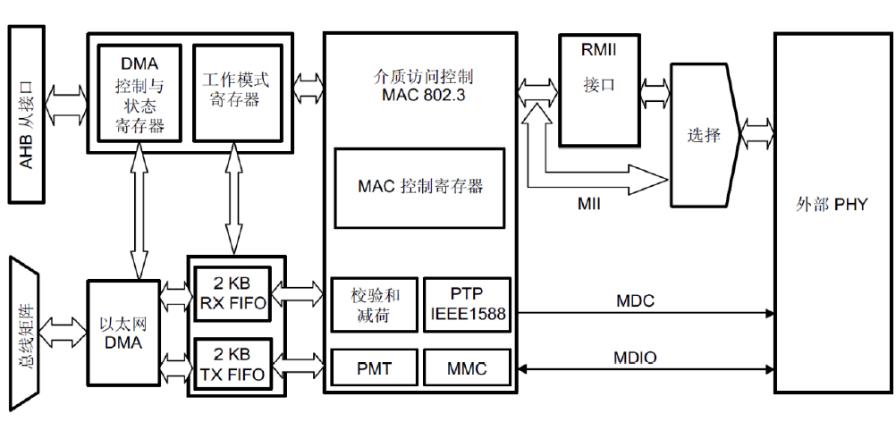


图3.2 STM32F4以太网口

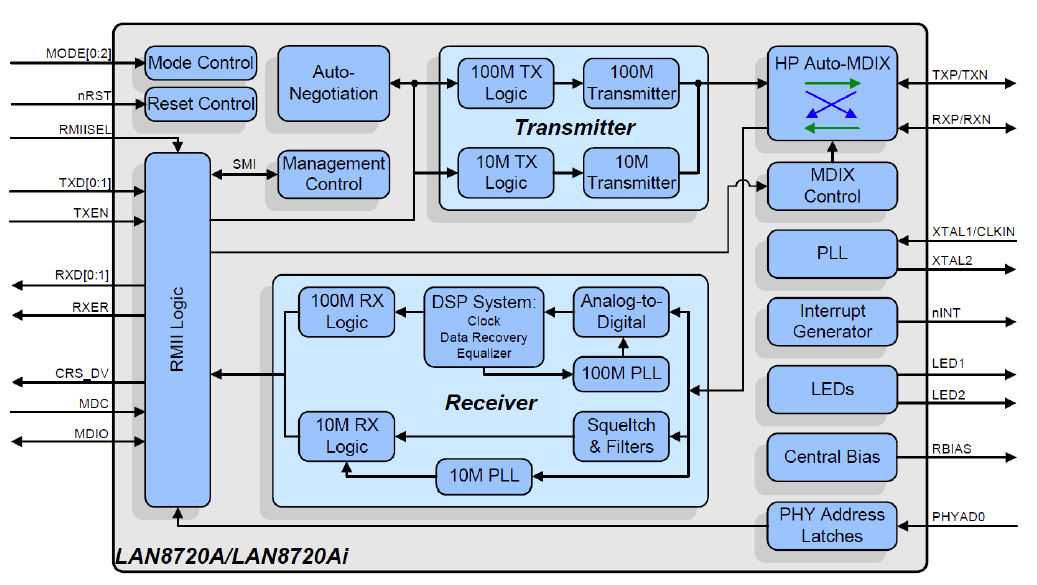
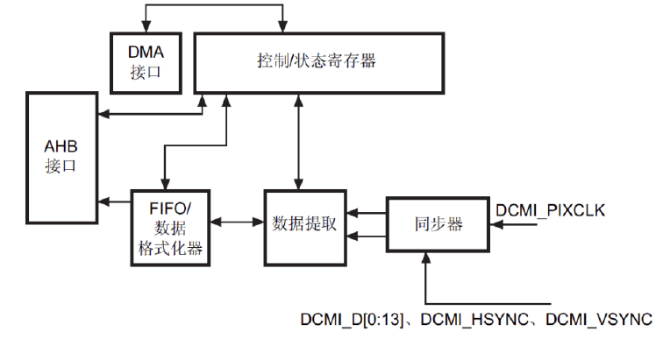


图3.3 LAN8720A芯片

3.1.2 摄像头

(a) OV2640摄像头模块 (b) STM32F4 DCMI

图3.4 OV2640模块与DCMI

OV2640摄像头模块是OV公司生产的CMOS UXGA图像传感器，通过串行摄像头控制总线（SCCB）连接，最高捕获帧率可达15帧，支持自动曝光控制、自动增益与自动白平衡等功能，并可进行JPEG格式输出。

器件连接方式选用STM32F4芯片自带的数字摄像头接口（DCMI），支持8-14位的CMOS数字摄像头控制与数据接收，并由DMA负责FIFO管理与数据搬运。正点原子公司在此基础上进行了开发板硬件设计，提供了一个符合SCCB规范的8位并行OV2640插口并提供了硬件驱动程序。开发板摄像头接口各引脚功能如下表所示。

表3.3 开发板摄像头接口引脚

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号 | 作用 |  | 信号 | 作用 |
| VCC | 电源 |  | OV\_PCLK | 像素时钟输出 |
| GND | 接地 |  | OV\_PWDN | 掉电使能信号 |
| OV\_SCL | SCCB时钟信号 |  | OV\_VSYNC | 帧同步信号 |
| OV\_SDA | SCCB数据信号 |  | OV\_HREF | 行同步信号 |
| OV\_D[0:7] | 8位数据输出 |  | OV\_RESET | 复位信号 |

3.1.3 温湿度传感器

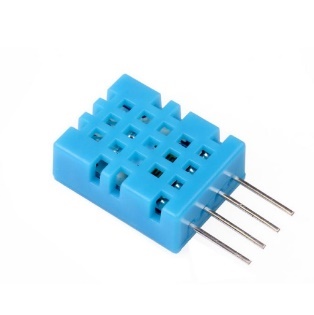


图3.5 DHT11温湿度传感器模块

DHT11是一款温湿度一体化数字传感器，包括一个电阻式测湿元件和一个 NTC测温元件，并与一个高性能 8 位单片机相连接，与开发板之间采用简单的单总线协议进行通信。

该协议下，传感器的单个完整数据包由5个字节组成，格式为一字节湿度整数数据、一字节湿度小数数据、一字节温度整数数据、一字节温度小数数据与一字节校验和。该模块的通信协议与开发过程相对较为简单。

3.2 硬件连接

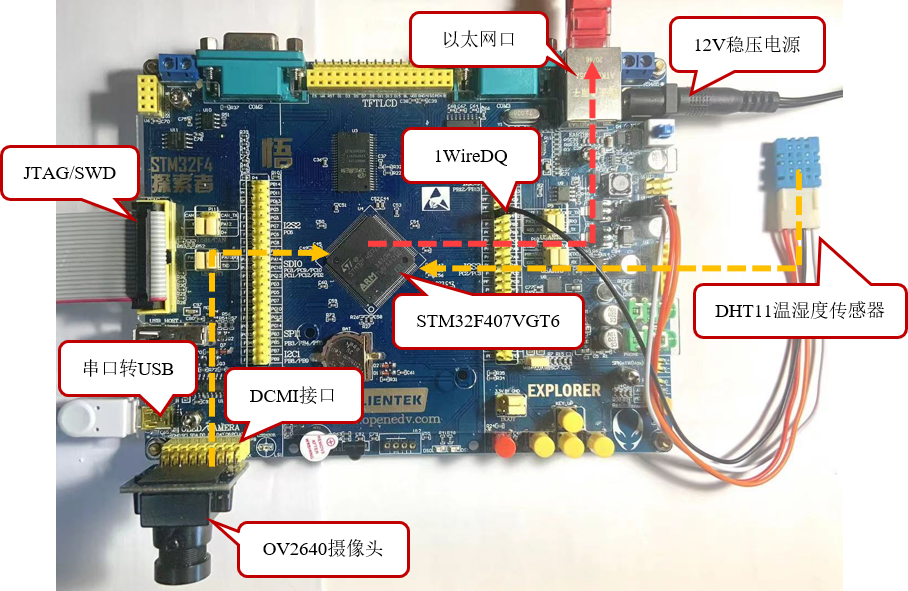


图3.6 嵌入式设备硬件连接图

本设计中，OV2640摄像头传感器数据通过DCMI接口传输，DHT11温湿度传感器通过1WireDQ（PF11）连接至STM32，之后通过以太网口上报服务器。给出本设计中使用的MCU各IO引脚及其对应功能如下表所示。

表3.4 STM32F407 IO引脚使用情况

|  |  |  |
| --- | --- | --- |
| 引脚 | 功能 | 说明 |
| PA0 | KEY\_WKUP | 开发板WKUP按键 |
| PA1 | RMII\_REF\_CLK | RMII参考时钟信号 |
| PA2 | ETH\_MDIO | 以太网卡MDIO信号 |
| PA4 | DCMI\_HREF | DCMI水平参考信号 |

续表3.4

|  |  |  |
| --- | --- | --- |
| 引脚 | 功能 | 说明 |
| PA6 | DCMI\_PCLK | DCMI像素时钟输出 |
| PA7 | RMII\_CRS\_DV | RMII载波监听与接受有效信号 |
| PA8 | DCMI\_XCLK | DCMI外部时钟信号 |
| PA9 | USART1\_TX | 串口1发送信号 |
| PA10 | USART1\_Rx | 串口1接收信号 |
| PA13 | SWDIO | SWD接口IO信号 |
| PA14 | SWDCLK | SWD接口时钟信号 |
| PB6 | DCMI\_D5 | DCMI数据信号5 |
| PB7 | DCMI\_VSYNC | DCMI垂直同步信号 |
| PC1 | ETH\_MDC | 以太网卡MDIO时钟信号 |
| PC4-5 | RMII\_RXD0-1 | RMII数据接收信号0-1 |
| PC6-9 | DCMI\_D0-3 | DCMI数据信号0-3 |
| PC12 | DCMI\_D4 | DCMI数据信号4 |
| PE2-4 | KEY2-0 | 开发板按键0-2 |
| PE5-6 | DCMI6-7 | DCMI数据信号6-7 |
| PF9-10 | LED0-1 | 开发板LED0-1 |
| PF11 | 1WIRE\_DQ | DHT11单总线数据信号（方案二） |
| PG9 | DCMI\_PWDN | DCMI掉电使能信号 |
| 1WIRE\_DQ | DHT11单总线数据信号（方案一） |
| PG13-14 | RMII\_TXD0-1 | RMII数据发送信号 |
| PG15 | DCMI\_RESET | DCMI重置信号 |

3.3 本章小结

本章主要分析并确定了嵌入式系统所涉控制器与各传感器模块型号，给出了传感器与控制器之间的通信方式，同时完成了硬件电路连接。

4 系统软件开发与模块协同

4.1 通信接口规范

由于本设计所包含的模块数量较多，且不同模块之间的数据交互行为较为繁复，因此有必要在进行具体功能的开发之前根据需求先行设计并制定接口规范。在本设计中，嵌入式设备到服务器以及服务器到交互应用程序之间全部采用网络进行数据传输，但所涉及的具体场景与通信需求又有所区别。因此，本节分别对两者从软硬件环境、数据类型、可实现性与时效性等角度出发进行具体的分析。

4.1.1 嵌入式设备-服务器

本设计中，嵌入式设备与服务器的通信接口主要负责向服务器传输摄像头与温湿度传感器数据以及向设备下发控制指令，因而该接口需要采用一种实时性强、允许数据量大的全双工通信机制。在嵌入式设备端，设计选用的lwIP协议栈对于TCP/IP下的网络通信进行了BSD Socket Api封装，从功能角度已经能够较好地满足需求，但从信息安全角度却为服务器带来了一定的安全隐患。

由于简单的Socket通信对于连接请求的发起与接受没有任何限制，则一旦服务器的IP地址与端口号信息暴露，外界设备将有可能不受限制的与服务器建立连接，这将有可能造成服务器负担过重甚至崩溃。因此，本设计在Socket创建过程中加入了设备校验机制。

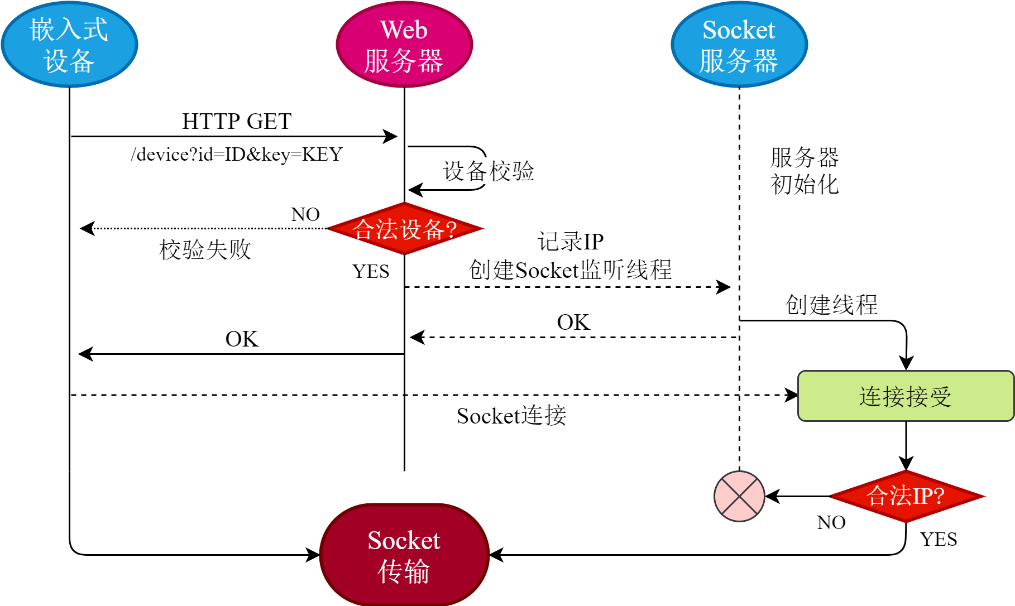


图4.1 设备校验与Socket创建接口

在设备投入使用之前，由设计者为每台设备分配公开的ID与私有的密钥，并在服务器内部维护一个合法设备IP映射表。当一台设备试图与服务器建立Socket连接时，应首先向服务器提交身份信息进行校验。校验通过后，服务器即判定该请求来自可信设备，之后将设备IP地址存入合法设备IP表，最后为该设备注册独立的Socket监听线程并等待设备发起连接。在监听线程接收到一个新的连接请求后，会首先查询该连接发起者的IP地址是否在服务器创建的合法设备IP表内，若查询成功则允许设备向服务器上传数据，反之则直接关闭连接。

当嵌入式设备与服务器的Socket连接被成功创建后，包括摄像头等传感器信息与用户指令在内的所有数据交互即可全部交由Socket完成。其中，从设备到服务器的摄像头数据采用JPEG格式，温湿度传感器数据则直接发送字符串，由服务器接收到后负责甄别。为实现用户对设备的控制，本设计定义了服务器与下发至设备的控制指令格式，其中各字段允许的取值及含义如下表所示。

表4.1 控制指令格式与字段含义

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 位 | [7:6] | [5:4] | [3:2] | [1] | [0] |
| 功能 | RESERVED | FRAME[1:0] | MODE[1:0] | CAM[0] | DHT[0] |
| 含义 | 保留 | 00：低帧率  01：中帧率  10：高帧率 | 00：QVGA  01：VGA  10：SVGA | 0：关闭  1：开启 | 0：不刷新  1：刷新 |

当用户试图通过交互程序改变设备状态时，需要通过POST请求向服务器提交控制参数。服务器在接收到该请求后按照预先定义的格式，将各控制参数编码为一字节长度的控制指令，并通过Socket下发至该用户对应的嵌入式设备。设备端在接收到相关请求后按照对应格式解析得到各项参数并做出响应。

例如，当用户期望设备刷新温湿度传感器数值、开启摄像头、设置图片格式为SGVA并采用高帧率时，交互应用程序需要向服务器提交的请求格式如下：

POST /device/command HTTP/1.1

Host: *SERVER\_IP*:8080

Content-Length: 38

Content-type: application/x-www-form-urlencoded

Cookie: *COOKIE*

DHT=true & CAM=true & MODE=SVGA & FRAME=HIGH

服务器在接收到该请求后，会根据上表将其编码为一字节指令如下：

command[7:0] = 00101011

之后调用Socket服务器接口将该指令发送至嵌入式设备，在设备端进行指令的解析与响应，相关代码见附件。

4.1.2 服务器-交互应用程序

在面向多用户的应用场景下，任何功能的使用都必首先须进行用户注册与登录。服务器通过数据库管理用户身份信息，同时负责生成cookies和会话。由于设计采用微信小程序作为移动端交互方案，设计中并不需要自定义用户身份，而是可以利用微信平台提供的用户全局唯一标识（openid）作为确定用户身份的最终依据。因而，用户登陆过程的核心内容便在于通过服务器向微信后台请求该用户的openid。

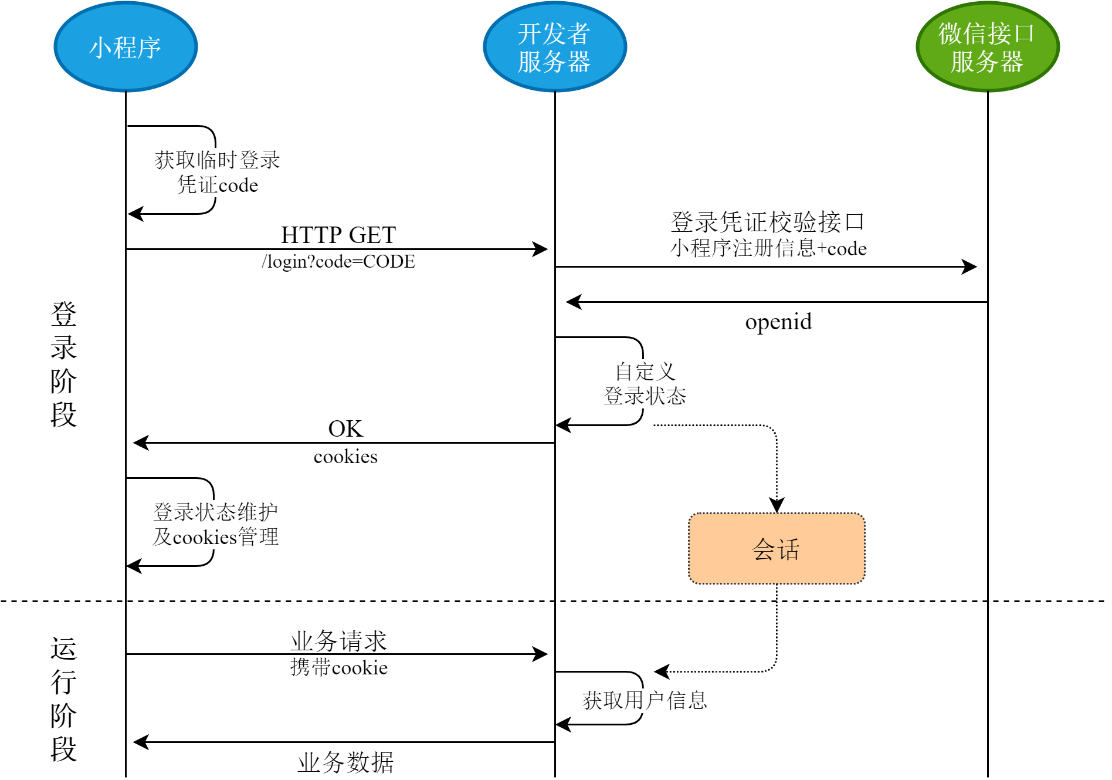


图4.2 用户登录与业务请求接口

进行登录时，首先需要通过小程序调用微信平台接口并获取临时登录凭证code，接着通过HTTP请求将code发送至开发者服务器。之后，服务器使用code与小程序注册信息向微信接口服务器换取该用户的openid，并以此为依据来维护用户在服务器的登录状态。之后，服务器向数据库查询该用户注册的设备信息，创建会话并为用户分配cookie。出于安全性考虑，openid等敏感信息一般只允许在服务器内部保存而不允许下发至用户程序。有关登录功能的具体实现见4.3.2章节。在之后的所有业务请求中，用户将通过在请求中携带cookie的方式向服务器表明登录身份、构建请求上下文并访问会话中保存的信息，服务器将以此为依据进行用户身份的识别以及用户与设备的匹配。

在登录功能的支持下，交互应用程序与服务器的所有数据交互全部由HTTP请求完成，有关服务器数据中转功能的具体实现见4.3.4章节。本设计中，服务器对交互应用程序提供的所有Api如下表所示。

表2.2 服务器部分Api列表

|  |  |  |  |
| --- | --- | --- | --- |
| 请求方法 | URL | 参数 | 功能 |
| GET | /login | 微信临时登录凭证 | 用户登录 |
| GET | /login/logout | - | 用户登出 |
| POST | /login/register | 设备ID、设备密钥 | 设备绑定 |
| POST | /device/command | 控制参数 | 设备控制 |
| GET | /stream | - | 请求图像数据 |
| GET | /stream/sensors | - | 请求传感器数据 |

4.2 嵌入式设备开发

在本设计中，嵌入式设备负责采集传感器数据并通过Socket上传至服务器。开发过程中的主要任务包括加载μC/OS-II操作系统、移植lwIP协议栈、网络通信功能的实现、传感器驱动与数据捕获以及用户指令响应等。嵌入式设备软件部分的功能使用Keil μVision5进行开发。



图4.3 Keil μVision5

4.2.1 μC/OS-II与lwIP移植

A. μC/OS-II环境配置

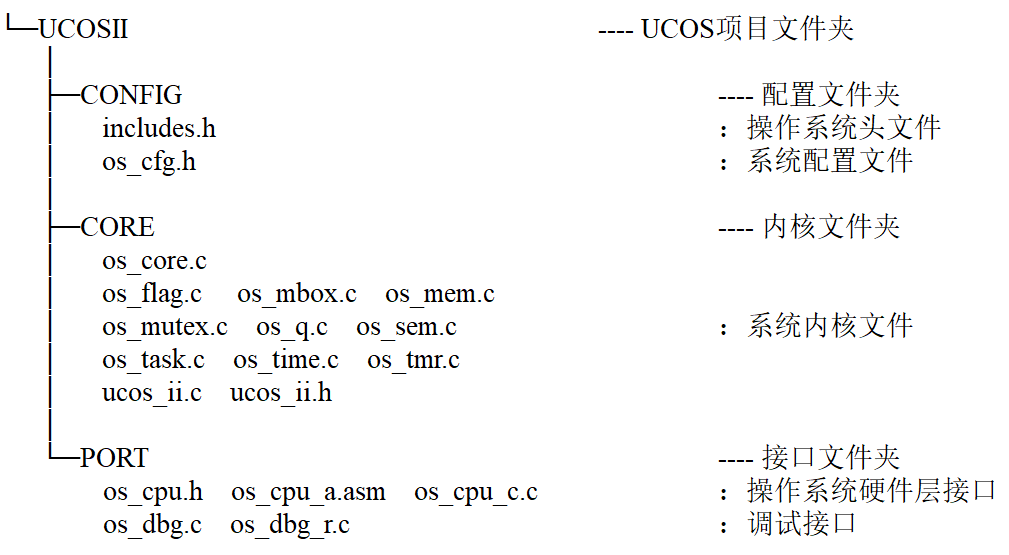


图4.4 μC/OS-II文件夹目录结构

为实现设备的多线程运行以及对于网络通信功能的支持，设计首先配置μC/OS-II操作系统环境。在工程内，依据功能不同将所有μC/OS系统源码划分为硬件接口（PORT）、系统内核（CORE）以及系统配置（CONFIG）三个文件夹。

首先在os\_cpu\_a.asm文件中，根据开发板CPU型号（Arm Cortex-M4）配置各类中断向量地址，同时定义有关系统临界保护、任务调度与上下文切换的关键函数如下表所示。

表4.3 os\_cpu\_a.asm关键函数列表

|  |  |
| --- | --- |
| 函数 | 功能 |
| CPU\_SR\_SAVE | 保存并置位PRIMASK，关闭系统中断响应 |
| CPU\_SR\_RESTORE | 重置PRIMASK，恢复系统中断响应 |
| OSStartHighRdy | 切换到已就绪任务中优先级最高者，并将其投入运行 |
| OSStartHang | 任务调度异常句柄，抛出HardFault |
| OSCtxSw | 任务级上下文切换 |
| OSIntCtxSw | 中断级上下文切换 |
| PendSV\_Handler | PendSV中断服务函数，调用Nosave并保存现场 |
| PendSV\_Handler\_Nosave | PendSV中断处理 |

接下来，直接将系统内核部分代码拷贝入工程，无需进行任何修改。之后在includes.h中包含C标准库与操作系统相关的各项头文件以供其他文件调用，最后在os\_cfg.h中结合当前应用实际对系统内核参数进行配置，给出其中部分关键参数如下表所示。

表4.4 操作系统部分关键参数

|  |  |  |
| --- | --- | --- |
| 参数 | 值 | 说明 |
| OS\_LOWEST\_PRIO | 63 | 系统最低任务优先级 |
| OS\_MAX\_EVENTS | 15 | 系统最大事件数 |
| OS\_MAX\_FLAGS | 5 | 系统最大信号量（集）数 |
| OS\_MAX\_MEM\_PART | 5 | 系统最大内存块数 |
| OS\_MAX\_QS | 5 | 系统最大队列控制块数 |
| OS\_MAX\_TASKS | 15 | 系统最大任务数 |
| OS\_TICKS\_PER\_SEC | 200 | 系统每秒时钟节拍数 |

由于μC/OS-II最多支持64个任务优先级且不允许优先级重复，同时系统内优先级最高的4个任务与优先级最低的4个任务均由内核占用（如软件定时器、统计任务与空闲任务等），因而系统最多可以支持56个用户自定义任务，对于小型的嵌入式系统是完全足够的。本设计共包含3个系统任务与10个用户自定义任务，如下表所示。

表4.5 嵌入式设备线程表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 优先级 | 名称 | 所属模块 | 堆栈大小/B | 说明 |
| 3 | TimerTask | μC/OS | 128 | 软件定时器任务 |
| 5 | tcpip\_thread | lwIP | 2000 | lwIP内核线程 |
| 7 | dhcp\_thread | lwIP | 128 | DHCP线程 |
| 11 | recv\_task | 用户（网络） | 128 | Socket接收线程 |
| 12 | http\_get\_task | 用户（网络） | 128 | HTTP请求GET处理线程 |
| 13 | app\_main\_task | 用户（网络） | 128 | 网络功能主线程 |
| 14 | camera\_task | 用户（传感器） | 256 | 摄像头线程 |
| 15 | dht\_task | 用户（传感器） | 128 | 温湿度传感器线程 |
| 18 | key\_task | 用户（传感器） | 128 | 按键响应线程 |
| 19 | led\_task | 用户（传感器） | 64 | LED线程 |
| 20 | start\_task | 用户 | 64 | 系统起始任务 |
| 62 | StatTask | μC/OS | 128 | 统计任务 |
| 63 | IdleTask | μC/OS | 128 | 空闲任务 |

B. lwIP移植

为实现网络通信的相关功能，使开发板具备向服务器进行HTTP请求与Socket通信的能力，设计在操作系统的支持下移植了lwIP协议栈。

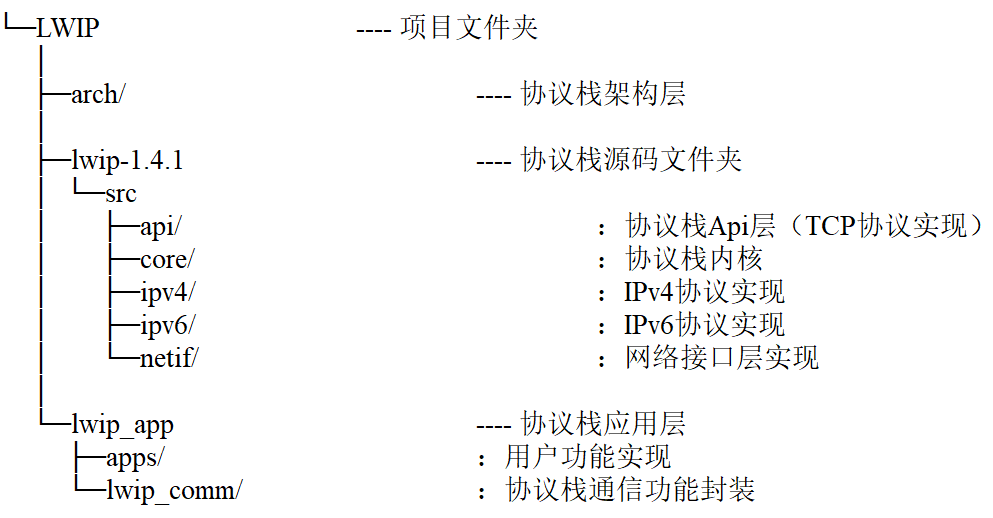


图4.5 lwIP文件夹目录结构（部分）

首先添加协议栈源码文件，根据功能将其分为网卡接口（NETIF）、Api（API）、内核（CORE）三个文件夹，并配置架构层（ARCH）功能。其中，架构层作为应用层与内核层的通信接口，其内部在操作系统提供的平台基础上定义了协议栈进行线程间数据交互时使用的信号量与消息邮箱，同时定义了关键的协议栈内核线程函数。

接下来配置协议栈与以太网卡的数据交互。在ST以太网库的支持下，针对开发板网卡类型（LAN2870）配置状态寄存器与通信延时。之后配置以太网DMA控制器，定义网络数据包缓冲区的内存分配与释放机制，并在网卡驱动程序中定义有关网卡硬件配置与数据交互功能的关键函数如下表所示。

表4.6 LAN2870.c关键函数列表

|  |  |
| --- | --- |
| 函数 | 说明 |
| LAN2870A\_Init | 网卡硬件初始化 |
| ETHERNET\_NVICConfiguration | 以太网中断配置 |
| LAN2870A\_Get\_Speed | 获取网卡连接速度和双工状态 |
| ETH\_MACDMA\_Config | 以太网MAC与DMA配置 |
| ETH\_IRQHandler | 以太网DMA中断服务函数 |
| ETH\_Rx\_Packet | 数据包接收 |
| ETH\_Tx\_Packet | 数据包发送 |
| ETH\_Get\_Current\_Tx\_Buffer | 获取当前发送缓冲区地址 |
| ETH\_Mem\_Malloc | 网卡数据缓冲区内存分配 |
| ETH\_Mem\_Free | 网卡数据缓冲区内存释放 |

最后添加协议栈通信功能封装接口，负责协议栈初始化、内核线程注册启动以及实现动态地址分配（DHCP）的相关功能实现，并根据应用实际在lwipopts.h中对协议栈内核参数进行配置，其中部分关键参数如下表所示。

表4.7 协议栈部分关键参数

|  |  |  |
| --- | --- | --- |
| 参数 | 值 | 说明 |
| TCPIP\_THREAD\_PRIO | 5 | 协议栈内核线程优先级 |
| MEM\_SIZE | 20000 | 协议栈堆区空间大小 |
| MEMP\_NUM\_PBUF | 50 | 数据缓冲区数 |
| MEMP\_NUM\_TCP\_PCB | 10 | 最大TCP连接数 |
| PBUF\_POOL\_SIZE | 60 | 内存池数 |
| PBUF\_POOL\_BUFSIZE | 512 | 内存池大小 |
| LWIP\_NETCONN | 1 | 使能Netconn |
| LWIP\_SOCKET | 1 | 使能Socket |

4.2.2 HTTP与Socket网络通信

由于协议栈没有实现对于HTTP协议的直接支持，本设计基于Sequential Api通过TCP连接实现了HTTP请求的发起与响应数据的解析，并注了册独立的HTTP GET请求处理任务，利用信号量机制向其他线程提供接口。

网络应用在初始化时会向操作系统注册HTTP线程并与服务器建立TCP连接。在收到请求信号后，线程通过TCP连接将报文发送至服务器并等待服务器响应，若响应超时则进行重传。收到响应数据后，HTTP线程将数据存入全局接收缓冲区并触发响应信号。有关HTTP请求发送与响应接收的完整实现见附录1所示。

对于Socket通信，应用初始化时会基于协议栈提供的BSD Socket Api向操作系统注册独立的Socket接收线程以负责相关网络数据的处理，该任务负责监听并响应来自服务器的用户控制指令。摄像头缓冲区内的数据以及温湿度传感器数据分别由各自的传感器线程调用Socket接口进行网络传输。传感器数据传输与指令响应相关内容见后续章节。

在各个功能线程的支持下，设计编写了网络任务主线程以负责设备不同运行阶段对于网络通信功能的控制，包括应用环境初始化、缓冲区内存管理、向服务器发起设备校验以及创建Socket连接等。

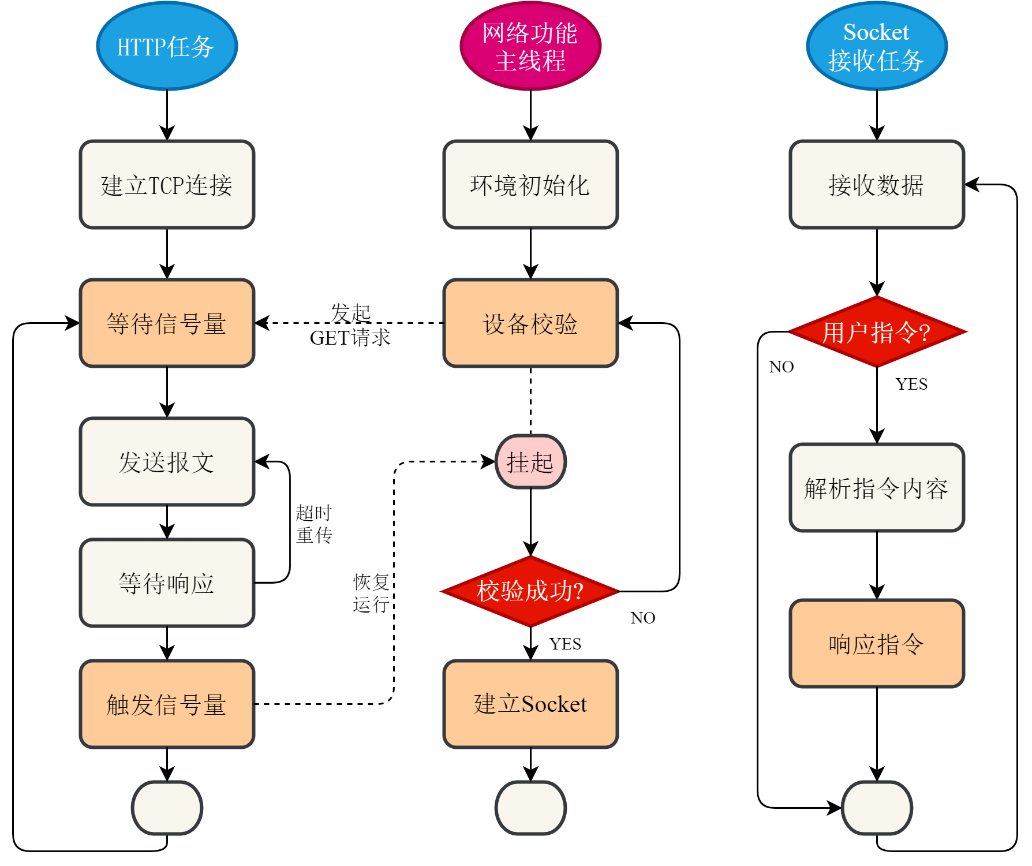


图4.6 嵌入式设备网络通信流程

主线程通过信号量和挂起机制与其他功能线程进行通信。具体而言，主线程在投入运行后，会首先进行HTTP发送缓冲区、接收缓冲区以及Socket接收缓冲区的内存申请。在设备校验阶段，网络功能主线程负责在全局发送缓冲区内构建HTTP报文并触发请求信号，之后挂起直到监听到来自HTTP线程的响应信号。此时主线程恢复运行，解析响应数据并判断校验是否成功。若成功则创建与服务器的Socket连接并开始数据传输，之后主线程挂起，后续所有数据交互任务全部由各功能线程完成。

4.2.3 传感器数据捕获

本设计中，控制器与各传感器模块的交互主要包括OV2640摄像头传感器与DHT11温湿度传感器两部分。

A. 摄像头模块

摄像头模块通过DCMI与处理器连接，并由DMA控制器负责进行数据搬运与缓冲区管理。当DCMI接收到一帧完整的图像后，DMA控制器会触发DCMI中断并将数据块搬运至内存。系统在进入中断后会首先等待数据传输完成，之后关闭DMA并将数据块标记为待处理状态，由其他线程负责进行数据处理并发送。直到本次数据被处理完成之前，内存中的图像缓冲区将不再会进行刷新。

在摄像头任务中，当程序检测到数据处于待处理状态时，会将数据通过Socket上传至服务器并将标志位归零。在下一次DCMI中断时，系统发现到上一次的数据已经处理完成后会清除缓存并重新使能DMA以进行下一次的数据传输。

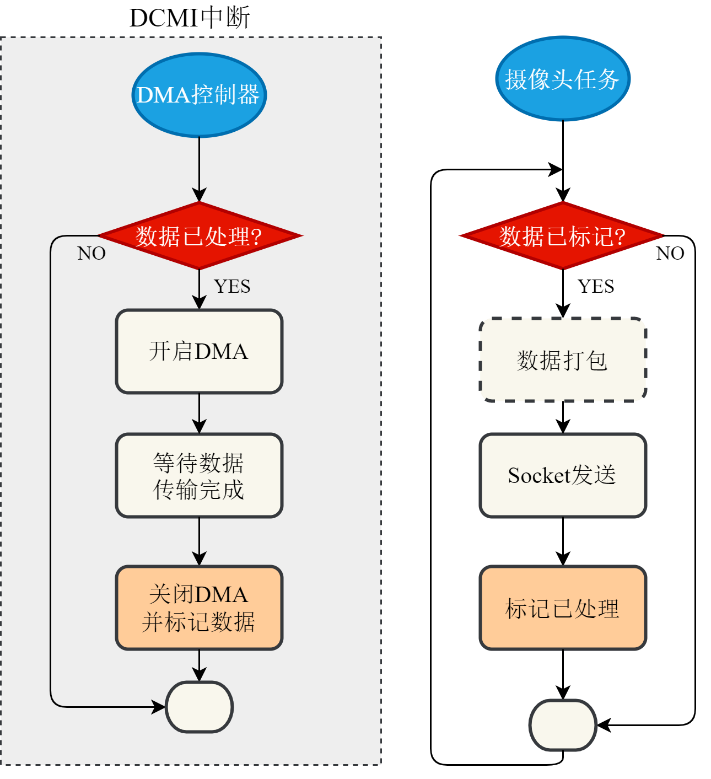


图4.7 DCMI与摄像头线程工作流程

在DCMI与摄像头线程的运行过程中，程序通过摄像头开关标志位、摄像头数据状态标志位、帧率标志位以及传输模式标志位等变量控制设备运行状态。其中，传输模式标志位是一个宏定义，表示摄像头数据的Socket传输是否采用批量方式。当该标志位被定义时，程序在解析得到一帧图像数据后并不会马上发起传输，而是将其拷贝进入一个公共的发送数据缓冲区中，同时增加缓存计数并开始下一帧图像的接收。当计数值达到所定义的阈值时，线程会调用Socket将缓冲区内所有图像数据一次性发送给服务器。

特别地，给出摄像头线程函数部分代码如下：

void camera\_task(void \*pdata) {

    /\* \* \* \* \* \* \* \*

    \* 缓冲区内存申请

    \* 摄像头、DCMI与DMA初始化

    \* \* \* \* \* \* \* \*/

    while(1)

    {

        if(cam\_on && jpeg\_data\_rdy == 1) {  //摄像头开启且数据已标记

#ifdef JPEG\_BATCH                           //采用批量模式

            send\_data\_len += jpeg\_data\_len; //记录数据总长度

            static u8 send\_buf\_cnt = 0;

            send\_buf\_cnt++;                 //缓存计数+1

            if(send\_buf\_cnt == SEND\_NUM) //SEND\_NUM为计数阈值，常取3-5

                send\_buf\_cnt = 0;           //已达到阈值，计数清零并准备发送

            else {

                jpeg\_data\_rdy = 2;          //未达到阈值，继续接收下一帧数据

                continue;

            }

            u8\* p = (u8\*)send\_buf;          //指向待发送批量数据

#else                                       //不采用批量模式

            send\_data\_len = jpeg\_data\_len;  //记录单帧数据长度

            u8\* p = (u8\*)jpeg\_buf;          //指向待发送单帧数据

#endif /\* JPEG\_BATCH \*/

                                            //Socket发送

            int ret = write(sock, p, send\_data\_len \* 4);

            send\_data\_len = 0;              //数据长度清零

            jpeg\_data\_rdy = 2;              //标记数据已处理，继续接收下一帧

        }

        tval = (u16)1000/frame;             //计算刷新间隔

        delay\_ms(tval);                     //线程延时

    }

}

B. 温湿度传感器模块

对于温湿度传感器数据，由于本设计使用的开发板所提供的单总线接口数据线引脚与摄像头接口的掉电使能引脚重合（均使用了PG9），设备运行过程中的对于温湿度传感器的读取操作将导致摄像头模块掉电关闭，因而只能对两者进行分时复用。

开发过程中发现，由于摄像头在数据传输未完成时掉电会使DCMI与摄像头失去同步，并导致DMA控制器进入未知状态，因而对与温湿度传感器的读取必须等待本次摄像头数据全部处理完成且下一次传输还未开始时进行，且在读取完成后需要对摄像头模块与DCMI全部进行重新初始化。

尽管对于温湿度传感器的读取将打断摄像头数据的正常传输，但由于实际应用情形中用户对于温湿度数据的请求频率远远低于对摄像头数据的请求频率，因此这种暂时的打断在一般情况下是可以接受的。

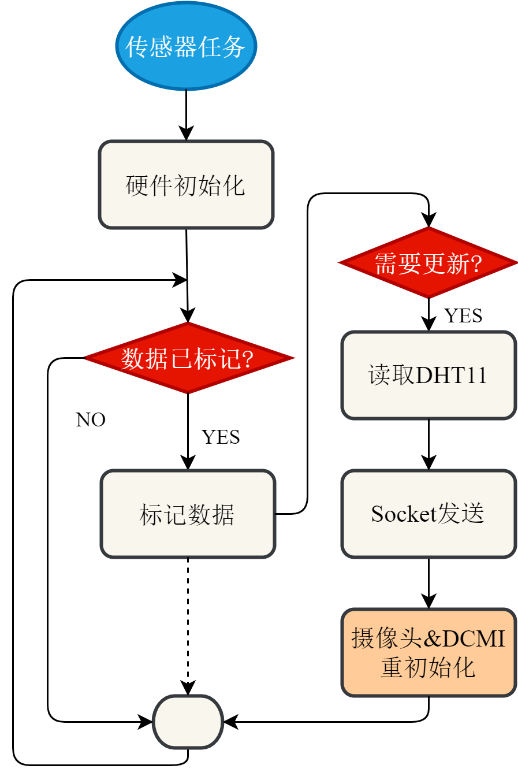


图4.8 温湿度传感器线程工作流程（方案一）

另一种更为简单直接的解决方案是将DHT11传感器的数据线更换为其他的引脚，舍弃板载接口并通过杜邦线进行连接。尽管接线更为繁琐且增加了硬件成本，但该方案可将温湿度传感器任务与摄像头任务完全分离，使得两者之间独立运行互不干扰。

给出传感器捕获过程中涉及到的重要标志位如下表所示。

表4.8 传感器捕获任务关键标志位

|  |  |  |  |
| --- | --- | --- | --- |
| 标志位 | 数据类型 | 作用域 | 说明 |
| jpeg\_data\_len | volatile u32 | 全局 | 摄像头单帧数据长度 |
| jpeg\_data\_rdy | volatile u8 | 全局 | 摄像头单帧数据标记 |
| send\_data\_len | u32 | 全局 | 摄像头待发送数据总长 |
| cam\_on | u8 | 全局 | 摄像头开关 |
| dht\_on | u8 | 全局 | 温湿度传感器开关 |
| frame | u8 | 全局 | 摄像头帧率 |
| send\_buf\_cnt | static u8 | 局部 | 摄像头缓存计数 |

4.3 Web服务器开发

本设计的服务器部分由一个负责响应HTTP请求的Web服务器和一个Socket服务器两部分组成。其中，本节介绍的Web服务器主要面向微信小程序提供登录注册以及获取设备数据的接口，同时也面向嵌入式设备提供身份校验服务。服务器端的功能全部通过Python编写并使用VSCode作为开发环境。



图4.9 VS Code

4.3.1 设计模式

为方便项目测试、加载配置以及发布运行时的托管，本设计中Flask采用工厂模式，并通过Python包对应用进行管理。



图4.10 服务器项目文件夹目录结构

应用工厂函数编写在项目文件夹下应用目录内的\_\_init\_\_.py文件中。该函数负责创建一个Flask应用实例并对其进行配置。函数首先导入包括应用密钥、项目实例文件夹位置、数据库路径以及客户端微信小程序ID与密钥等全局配置信息，之后通过添加蓝图（blueprint）的方式注册项目中所有用户编写的视图函数，之后生成Socket服务器实例（具体见4.4章节），最后返回一个Flask应用实例以供其它托管程序调用。在部署服务器并运行时，Flask提供的开发服务器或其他WSGI托管服务器将导入该Python包（flaskr），并通过工厂函数获取一个应用实例加以运行。项目根目录下的tests文件夹内编写了有关服务器单元测试的相关内容，详见5.2章节。

4.3.2 用户管理

本设计中用户管理的相关功能主要包含用户登录与设备绑定两部分。为配合相关功能的实现，设计采用SQLite3数据库进行用户与设备信息的存储。这是一个遵循ACID规范的轻型关系数据库，基于文件系统运行而不需要专门的服务器支持。需要访问数据库的视图函数通过调用db.py文件提供的接口来进行数据库连接并获取一个可操作的游标对象，之后通过该对象进行具体的数据库查询。

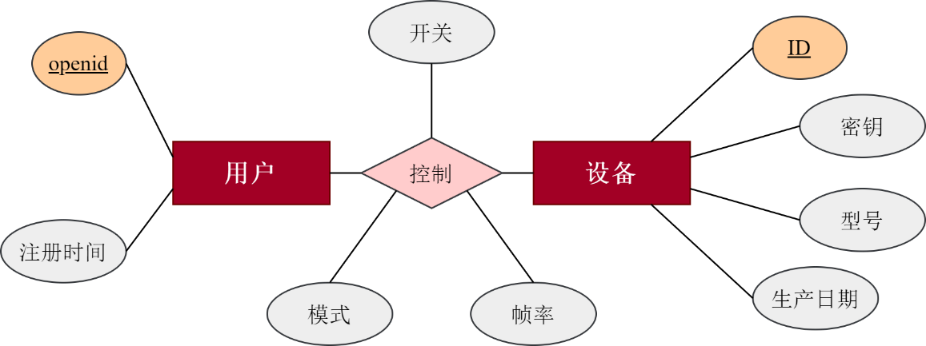


图4.11 SQLite3 E-R图

数据库中所存储的信息主要包括两个表。其一是运行时创建的用户表（Users），包括已注册用户的openid、注册日期与绑定设备信息；其二是出厂前创建的设备表（Devices），包括由开发者登记并信任的设备ID、密钥、型号与生产日期信息。

由于服务器将openid作为用户身份的唯一识别依据，当客户端微信小程序投入运行时会上报临时登录凭证code并由服务器通过该凭证向微信接口服务器换取openid，之后向数据库Users表查询用户信息。

对于首次登陆的用户来说，此时其openid并不存在于数据库之中，则登录视图函数将返回信息提示客户端进行注册与设备绑定。具体方式为客户端通过POST请求向服务器提交待绑定设备的ID与密钥，之后由负责管理注册功能的视图函数查询Devices表并进行校验，若正确无误则将当前用户的openid与其绑定的设备ID作为一条新数据插入Users表，之后向客户端返回“OK”表示注册成功。

对于已经完成设备绑定的用户，其注册信息已经存在于Users表中。此时登录视图函数将为该用户创建会话，最后向用户返回 “OK”以表示登陆成功。当开发者在视图函数中操作了会话对象之后，Flask框架将自动为用户分配cookie并将其添加到响应当中。

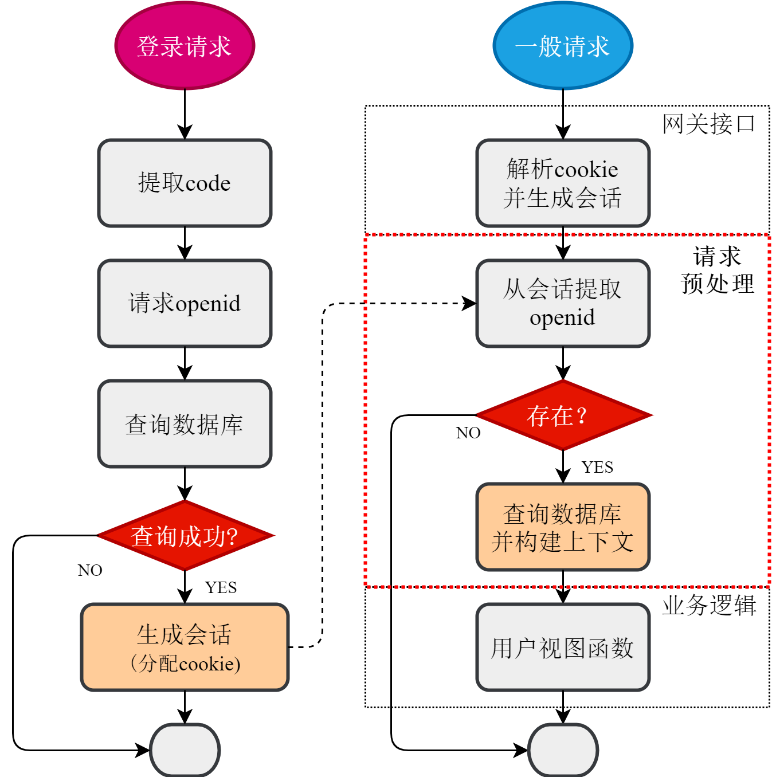


图4.12 服务器用户管理流程

在获取所有具体的服务之前，客户端需要在每次请求中上报cookie。Flask会自动对其进行解析并在上下文中还原客户端登录时保存在session里的用户信息。在此基础上，用户管理系统向Flask注册并挂接了请求预处理函数，该函数将在所有视图处理具体的客户端请求前被Flask调用。

特别地，给出实现用户身份管理功能的请求预处理函数与视图装饰器如下。

@bp.before\_app\_request                  # 挂接在请求处理之前

def load\_logged\_in\_user():

    ''' 载入用户信息 '''

    openid = session.get('openid')      # 从会话中查询openid

    if openid is None:                  # 未登录用户

        g.user = None

    else:

        g.user = get\_db().execute(      # 已登录，查询数据库Users表

            'SELECT \* FROM Users WHERE openid=?', (openid,)

        ).fetchone()                    # 将用户信息存入全局上下文对象g

def login\_required(view):

    ''' 需要用户登陆的视图函数装饰器 '''

    @functools.wraps(view)              # 保留被装饰函数属性

    def wrapped\_view(\*\*kwargs):

        if g.user is None:              # 未登录用户

            return 'LOGIN'              # 返回登录提示信息

        return view(\*\*kwargs)           # 已登录，调用视图函数处理请求

    return wrapped\_view

对于已经在服务器成功登录的用户，函数利用会话中保存的用户openid向数据库Users表查询得到该用户绑定的设备信息，之后将其存入Flask提供的全局上下文对象以供视图函数访问。之后只需要将所有需要用户登陆的视图函数使用“@login\_required”进行装饰，若用户并未进行过登录则该函数将跳过视图函数而直接向用户返回登录提示信息，表示该功能的访问需要首先向服务器表明用户身份。

4.3.3 设备校验

由于设备身份信息在服务器端存储在SQLite数据库中，而本设计用于响应HTTP请求的Flask服务器已经提供了一套完整的面向数据库的连接规范，因此设备校验的相关工作直接交由Flask完成，具体方式为设备向服务器发起GET请求并携带相关身份信息，之后由处理该请求的视图函数进行数据库连接、查询与设备合法性判定，并在验证通过时调用Socket服务器提供的相关接口函数创建监听线程。

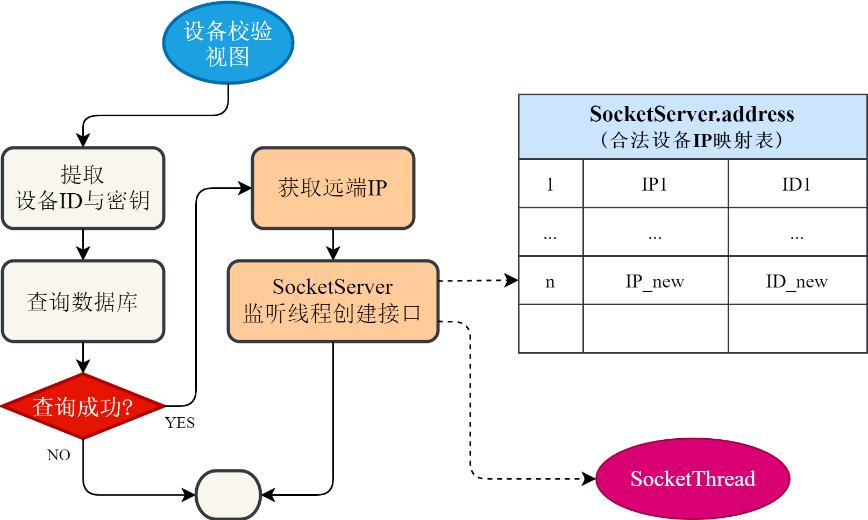


图4.13 服务器设备校验流程

该视图函数从来自设备的HTTP请求中取得设备ID与密钥，并向数据库Devices表发起查询并进行比对校验。若信息正确，服务器将记录当前请求的IP作为该设备的合法IP地址，并将设备IP到ID的映射存入Socket服务器合法设备表（address），之后调用Socket服务器提供的接口函数创建监听线程，并通知Socket服务器允许接受来自该IP地址的Socket连接请求。有关Socket服务器接口的具体实现见4.4章节。

4.3.4 数据中转

当客户端向服务器请求获得一份设备数据时，Flask会首先检查全局上下文对象中是否包含与该用户对应的有效设备信息。在视图函数中，服务器会根据用户绑定的设备信息从Socket服务器线程表（详见4.4.2章节）中查询得到与该设备对应的监听线程对象，之后从线程对象的数据缓存（详见4.4.3章节）中取得该设备数据并将其返回客户端，具体流程如图所示。

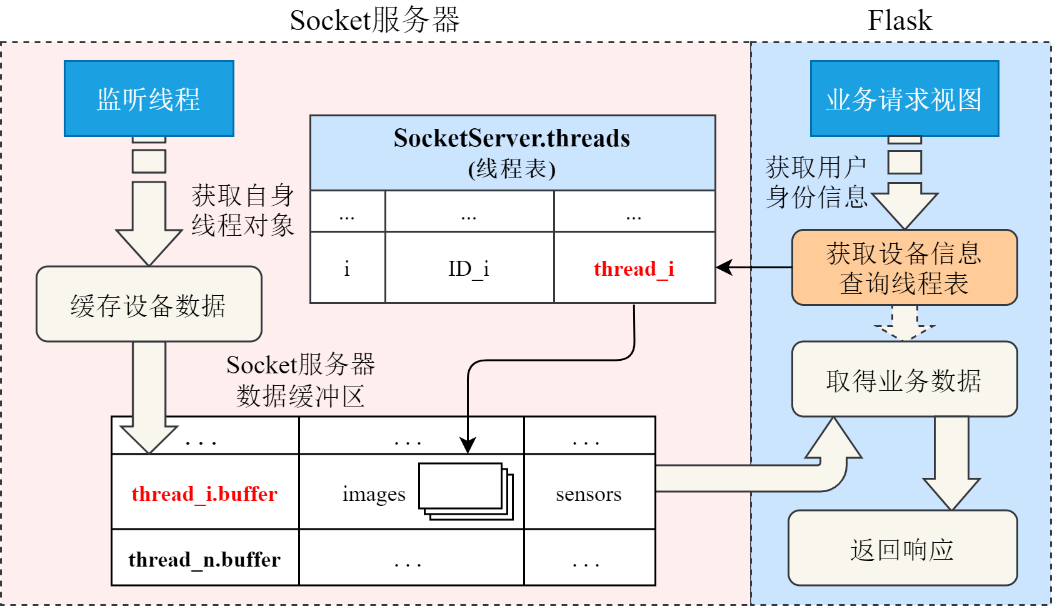


图4.14 服务器数据中转流程

特别地，给出负责响应图像数据请求的视图函数代码如下（上图蓝色部分）：

@bp.route('/')                          # 声明请求路由

@login\_required                         # 声明需要用户登录

@device\_required                        # 声明需要设备连接

def stream():

    ''' 从指定数据缓存中获取并返回一帧图像 '''

    device\_id = g.user['devices']       # 获取用户绑定的设备ID

    img = pop\_img(device\_id)            # 调用接口获取一帧图像

    arr = BytesIO()                     # 构造字节流IO对象

    img.save(arr, format('JPEG'))       # 向IO对象写入图片

    res = make\_response(arr.getvalue()) # 构造HTTP响应

    res.headers['Content-Type'] = 'image/jpeg'

    return res                          # 返回响应数据

4.4 Socket服务器开发

Socket服务器主要负责接收嵌入式设备通过Socket上传的监测数据，在服务器内部缓存并将其提供给Flask。本设计通过一个SocketServer类管理服务器端Socket通信的所有相关工作，并向Flask中的视图函数提供接口，具体如下表所示。

表4.9 Socket服务器模块接口列表

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 参数 | 返回值类型 | 说明 |
| init\_server | 服务器配置参数 | SocketServer实例 | 服务器初始化 |
| create\_socket | 设备ID、IP | str（提示信息） | 创建监听线程 |

续表4.9

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 参数 | 返回值 | 说明 |
| get\_thread | 设备ID | 监听线程对象 | 查询监听线程 |
| pop\_img | 设备ID | 标准图像对象 | 获取一帧图像数据 |
| pop\_sensors | 设备ID | 传感器数据对象 | 获取一份传感器数据 |
| send\_command | 设备ID、指令内容 | str（提示信息） | 下发控制指令 |
| close\_socket | 设备ID | str（提示信息） | 关闭监听线程 |
| socket\_alive | 设备ID | bool | 查询线程是否存在 |

于简洁性考虑， Socket服务器本身并不会注册独立的线程，其工作方式为在服务器初始化阶段生成一个全局唯一的SocketServer实例，当Flask需要进行Socket创建或访问某一设备的数据时，视图函数将通过接口调用该实例提供的相关方法来获取对应的服务。

4.4.1 监听线程

在本设计中，服务器与设备创建Socket连接的本质即在服务器端注册一个新的Socket监听线程并使其投入运行。为了实现设备数据在服务器端的缓存以及外部函数对线程运行状态的控制，本设计自行编写了监听线程对象SocketThread类。该类继承自Python标准库的Thread对象，除了具有普通线程应具备的基本属性与方法之外，SocketThread类增添了设备数据缓冲区与运行状态标志位，同时提供了访问缓冲区数据的接口方法。

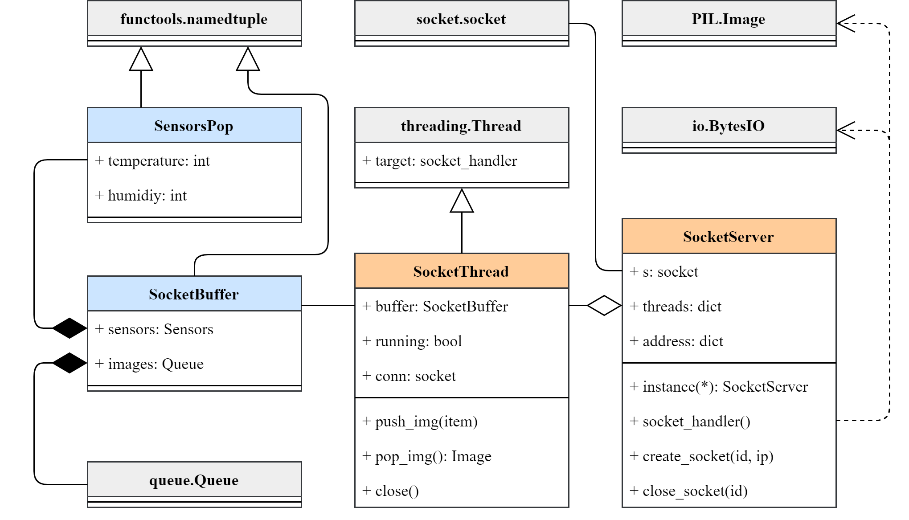


图4.15 Socket服务器模块UML图

与普通的线程对象类似，SocketThread类在实例化时接受一个函数对象作为线程函数句柄。该函数在运行时会首先接受一个Socket连接并校验该IP地址是否合法。若校验成功，则从服务器IP映射表中则查询得到对应的设备ID，以此为依据从服务器线程表中查询得到自身所属的线程对象以便访问缓冲区；若校验失败则直接关闭连接。

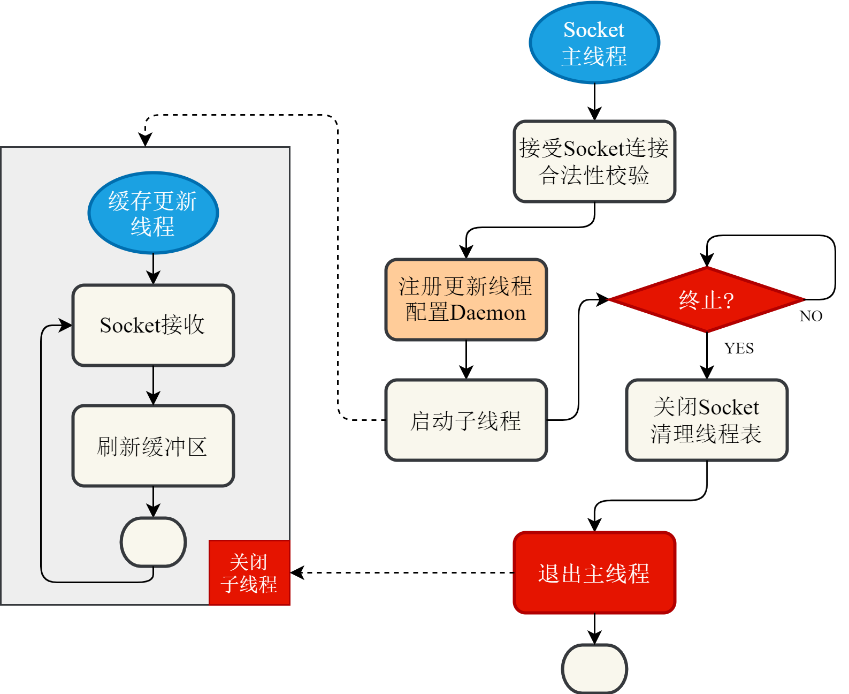


图4.16 Socket监听线程工作流程

特别地，给出Socket主线程函数代码如下：

def socket\_handler(self):               # SocketServer成员函数

    ''' 监听线程函数 '''

    conn, addr = self.s.accept()        # 接受一个Socket连接

    ip = addr[0]                        # 获取远端IP

    if not self.verify\_ip(ip):          # 校验IP合法性

        conn.close()                    # 非法IP，关闭连接并退出

        return

    t = self.threads[ip]                # 查询监听线程对象

    subt = threading.Thread(            # 注册缓存刷新子线程

        target=self.refresh\_handler,    # 子线程函数句柄

        args=(t,)                       # 线程对象做参数传入

    )

    subt.setDaemon(True)                # 设置子线程Daemon属性

    subt.start()                        # 启动线程

    while t.running:                    # 判断服务器是否通知线程终止

        subt.join(

            timeout=self.handler\_timeout

        )                               # 循环等待子线程执行

    conn.close()                        # 接到线程终止通知，关闭连接

    self.threads.pop(ip)                # 清理服务器线程表

# 此时当前线程退出

# 同时子线程subt被操作系统强制终止

由于正常情况下的Socket接收函数是一个阻塞操作，当服务器通知线程需要进行终止时，若监听线程恰好处于阻塞接收状态则不能及时关闭。这将阻碍Socket服务器的正常运行，并可能导致“幽灵线程”的产生。

为保证服务器通知线程关闭后与当前设备相关的所有工作线程及其子线程均能够正常终止，此处借助了Python的守护进程（Daemon）机制。当监听线程创建接口被成功调用后，Socket服务器将注册并运行一个主监听线程，在该线程中接受Socket连接、获取线程对象、接着注册一个专门负责缓冲区更新的子线程并设置该线程的Daemon属性，这将保证当主线程退出时子线程也被操作系统强制关闭。

4.4.2 连接可重入

开发过程中发现，由于服务器无法实时获知设备端Socket的连接状态，且服务器上已经启动的监听线程在非异常状态下不会自动退出，则当设备意外重启或出于其他原因需要重新与服务器建立Socket连接时，服务器内部可能仍然维持着上一次的Socket连接状态而拒绝设备新发起的连接请求。因此，为了提高系统鲁棒性，设计在Socket服务器内部加入了系统线程表，用以记录已经创建并投入运行的Socket监听线程，同时以该表作为Flask视图函数查询对应设备数据缓冲区的索引。

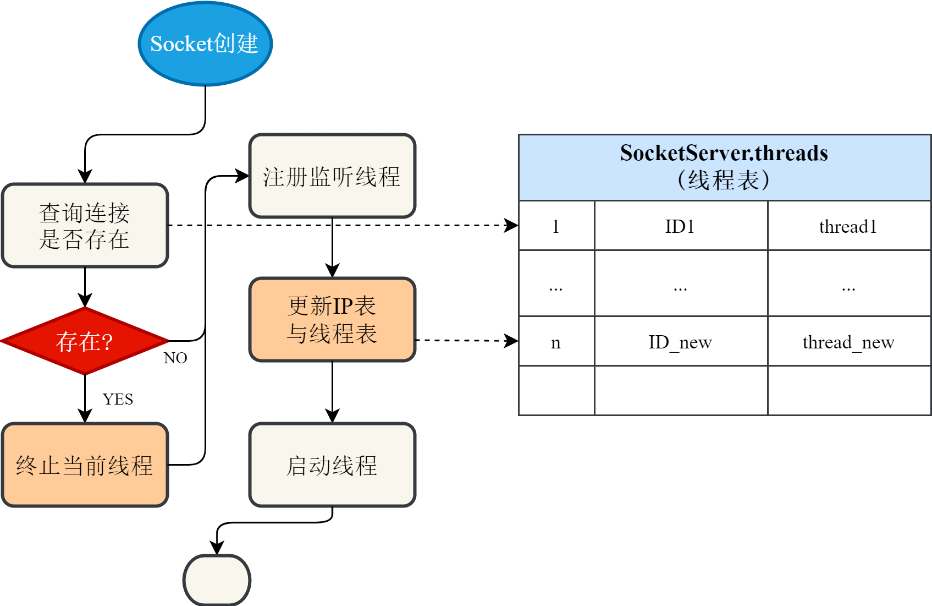


图4.17 服务器Socket创建接口工作流程

当设备首次与服务器建立连接时，服务器将生成该设备的监听线程并更新线程表。在设备意外重启并重新向服务器发起设备校验与Socket连接请求时，服务器发现与当前设备匹配的Socket监听线程已经处于运行状态，此时服务器将通知并等待该线程关闭，由Deamon机制保证所有相关子线程全部终止。之后，服务器重新注册一个监听线程，按照相同的流程将其投入运行并等待新的连接即可。

4.4.2 数据缓存与并发处理

为方便Flask视图函数访问Socket服务器接收到的设备数据，本设计为SocketThread对象添加了数据缓存属性，该属性使用具名元组（namedtuple）存储一个包含温湿度数据的自定义传感器数据对象以及一个图像数据缓冲队列。其中图像数据缓冲队列使用Python标准库queue中的Queue模块实现，这是一个基于Python标准库双端队列deque的线程安全的FIFO对象，内置互斥锁并提供了多种阻塞与非阻塞式的访问方法。



图4.18 数据缓存模块UML图

由于本设计中用户对服务器的请求较为频繁，而当前框架下服务器并发规模有限，因此在处理请求时应尽量避免使用阻塞方法从而导致的服务器负担过重。

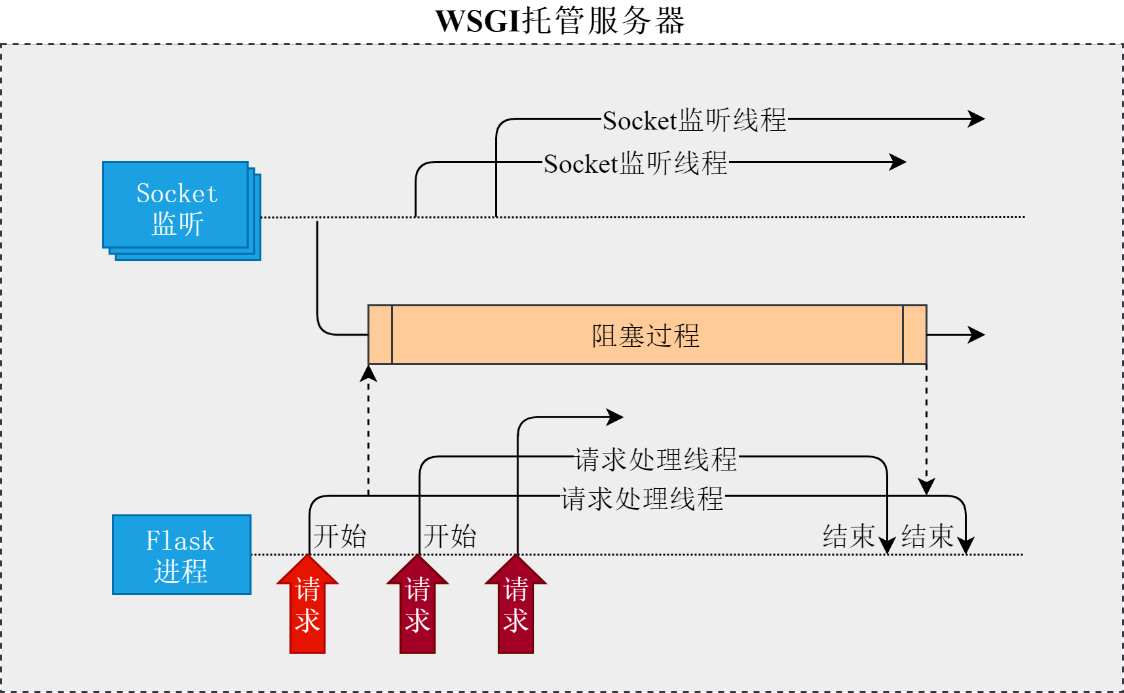


图4.19 服务器并发处理流程

值得一提的是，由于Flask多作为一种基于WSGI的应用框架，其处理请求的方式为单线程还是多线程并不取决于Flask本身，而是取决于托管Flask应用的WSGI服务器具体工作于何种模式之下。因此，在请求处理过程中可能含有较多耗时操作的前提下，应首先保证Flask应用服务于多线程模式，以避免调用该方法的视图函数阻塞其他请求的处理。

以图像数据缓冲区的访问为例，当Socket服务器监听线程试图将一帧JPEG数据存入缓冲区时，监听线程函数调用非阻塞式入队接口函数将图像入队。若此时缓冲队列已满，该方法会抛出队满异常。服务器在运行时将会捕获该异常并直接将该帧图像丢弃，之后等待下一次插入操作。类似地，当Flask试图访问并获得缓冲区中的数据时，视图函数调用非阻塞式出队接口函数获得一帧图像。该方法同样将在队列为空时抛出异常并返回空值。完整代码实现见附录2所示。

4.5 微信小程序开发



图4.20 微信开发者工具

本设计使用微信小程序作为客户端的人机交互方案，主要负责提供基于微信平台身份系统的用户登录操作并对来自设备的监测数据进行显示。相关功能使用微信开发者工具进行开发。



图4.21 小程序项目文件夹目录结构

4.5.1 用户登录与设备绑定

由于小程序所有功能的实现均依赖于用户登录信息，因此用户登录的相关功能注册在app.js文件里的全局应用实例中。相关功能在小程序运行环境初始化完成时触发且全局只触发一次。小程序会首先调用登录方法获取一个临时登录凭证code，之后通过请求接口将该凭证通过HTTP GET请求发送至服务器。服务器登录请求响应数据段的可能取值与含义如下表所示。

表4.10 登录请求返回字段说明

|  |  |
| --- | --- |
| 数据段内容 | 含义 |
| OK | 登录成功 |
| Code missing | 请求中未包含code |
| Identification failed | 请求openid失败 |
| REG | 用户未注册 |

对于一个已经完成设备绑定的用户，服务器在成功向微信后台换取该用户的openid并完成登录操作后，会向小程序返回带有cookie的“OK”以表示登录成功；而对于一个新用户，服务器会返回“REG”以提示小程序进行设备绑定。

由于小程序所有网络请求的发起均需要通过微信后台中转，因而小程序本身并不是一个严格意义上的HTTP客户端，也没有内建的cookie机制。则若想在不同请求间保持服务器端的会话则需要开发者手动设置各个请求头部的cookie字段。本设计中，服务器为小程序分配的cookie在登录成功后会被从响应头解析出来并保存在小程序实例的全局数据对象中，以方便所有页面在发起请求时访问并添加cookie。

当登录请求返回的响应数据段为“REG”时，表示服务器未在数据库Users表中查询到该用户的设备信息，需要用户进行设备绑定。此时，小程序在完成加载后将自动跳转至设备绑定页面，用户可以在当前页面中输入欲绑定设备的ID和密钥，之后由小程序通过POST请求将表单提交至服务器进行验证。服务器将根据表单查询并比对数据库设备表（Devices）中保存的相关信息进行校验。校验通过并完成绑定的用户将收到来自服务器的“OK”响应，之后重新进行登录，完成后将页面重定向至应用首页。服务器注册请求响应数据段的可能取值与含义如下表所示。

表4.11 注册请求返回字段说明

|  |  |
| --- | --- |
| 数据段内容 | 含义 |
| OK | 注册成功 |
| Openid missing | 会话中openid丢失 |
| No such device as (ID) | 无法识别的设备 |
| Wrong device-key | 设备密钥错误 |
| Already registered | 已注册用户 |

4.5.2 数据交互

在用户登录和设备绑定功能的基础上，微信小程序可以向服务器请求由已绑定设备上传并暂存在服务器的监测数据。

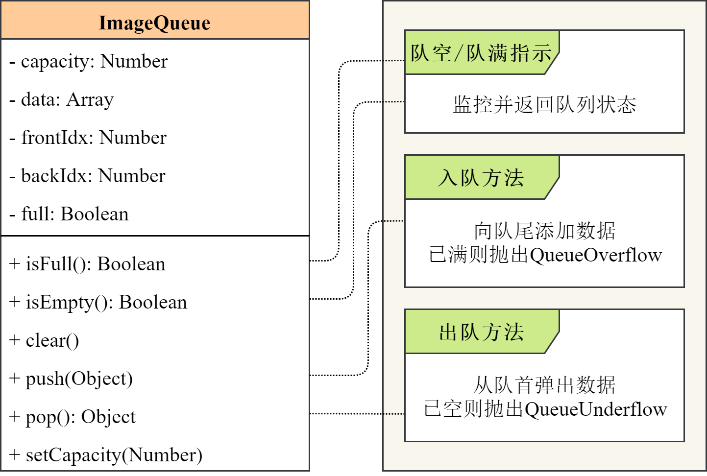


图4.22 ImgQueue模块UML图

对于来自摄像头的JPEG流数据来说，由于图像以每秒数帧的频率刷新，而一个获取图像的HTTP请求可能需要更长的时间来完成响应并渲染。因此，若在程序中每次只请求并刷新一张图片则会导致较为严重的卡顿。相应地，本设计在小程序中也同样编写了图像缓冲队列ImgQueue，该模块采用循环队列实现并利用JS的单线程机制保证访问安全，对外提供入队和出队的接口方法。由于本设计使用的文件下载接口为异步接口，程序并不会在请求一张图片并等待响应的过程中阻塞，因此小程序可以在允许范围内向服务器批量发起并等待多个请求，同时不受干扰地将之前已经入队的图片资源取出后渲染至视图层。

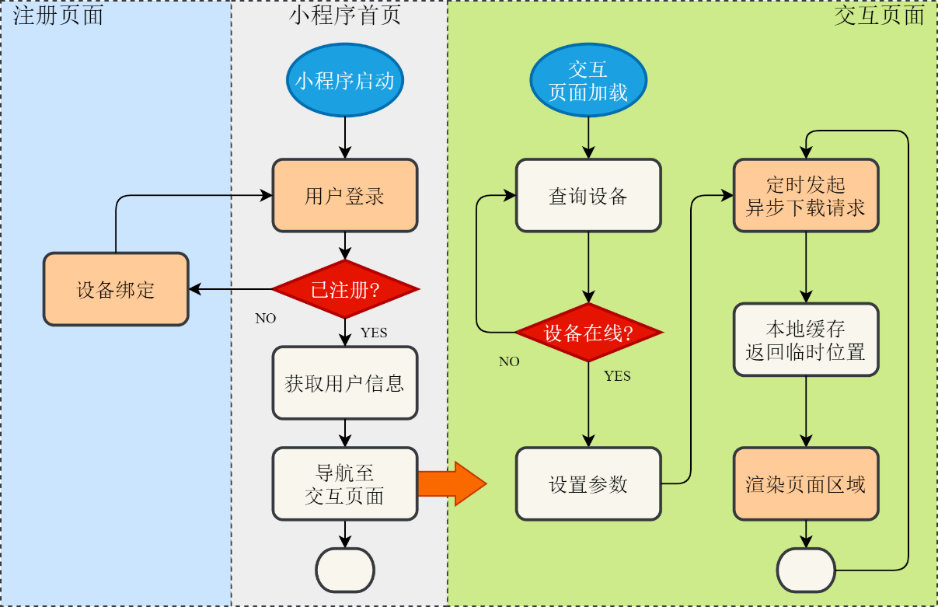


图4.23 微信小程序工作流程

特别地，给出小程序数据交互页面加载函数部分代码如下：

loadPage() {

    imgBuf = new ImgQueue(imgBufSize)   //申请图像缓冲队列

var page = this                     //page指向当前页面对象

//定义一个页面定时器

    imgInter = setInterval(() => {      //定时器回调函数参数

        page.imgDownload()              //下载一帧图片并添加其临时URL至队尾

        page.setData({                  //调用接口，更新数据域指定字段

            datetime: formatTime(

                new Date()              //更新页面时间字段

            ),

            imgSrc: imgBuf.pop()        //弹出一个URL并更新图片组件源字段

        })                              //小程序底层重新渲染页面

    }, refreshPeriod)                   //定时器时间间隔参数

}

在小程序数据显示页面进行加载时，该函数会定义一个定时器对象，驱动程序按照固定的时间间隔从服务器下载一帧图片，并将其临时URL存入缓冲队列末尾。随后小程序从缓冲队列首端取得一个临时资源，调用接口方法更新模板文档中图片组件的URL，随后由小程序底层的JsCore线程通知WebView线程重新渲染页面相关区域以完成显示。

本设计中温湿度数据的请求与显示原理与此类似，考虑到其数据量很小且请求频率远低于图像数据，所以并不需要加入缓冲机制，因而其实现也相对简单，在此不再赘述。完整代码实现见附录3所示。

4.6 本章小结

本章主要进行了系统软件部分的设计，根据前述章节的模块架构与目标功能分别完成了嵌入式设备、Web服务器、Socket服务器以及微信小程序的功能开发，同时制定并实现了系统各模块之间的交互协同策略。

5 项目测试

5.1 嵌入式设备测试

本节主要进行嵌入式设备端的单元测试。

作为整个系统的主要数据来源，嵌入式设备所承担的传感器捕获与网络传输任务是实现本设计各项功能的基础，同时也是应首先进行测试的部分。相关测试环境如下表所示。

表5.1 嵌入式设备测试环境

|  |  |  |
| --- | --- | --- |
| 环境 | 型号/版本 | 说明 |
| 控制器 | STM32F4 | 正点原子探索者开发板 |
| 电源 | GQ12-120100-CC | 12V直流稳压电源，用于设备供电 |
| 调试器 | ST-Link/V2 | 用于代码烧写与程序调试 |
| 路由器 | DIR-823G | 通过以太网线连接开发板，提供网络通信 |
| 串口调试助手 | XCOM/V2.3 | 监听串口调试信息 |
| 网络调试助手 | XNET/V1.1 | 作为TCP服务器接收网络数据 |

5.1.1 环境测试

首先对系统运行环境进行测试，内容主要包括μC/OS-II操作系统任务调度能力与lwIP协议栈网络通信能力。

A. 操作系统移植测试

首先测试操作系统是否移植成功并能够正常进行任务调度。定义两个任务分别控制LED0与LED1。将程序编译无误后烧录至开发板，启动并观察运行效果。

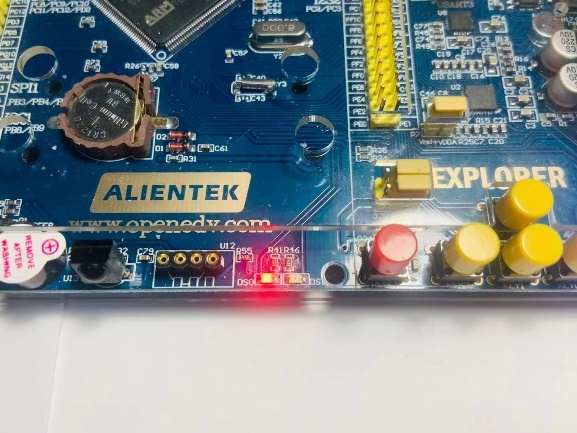
  

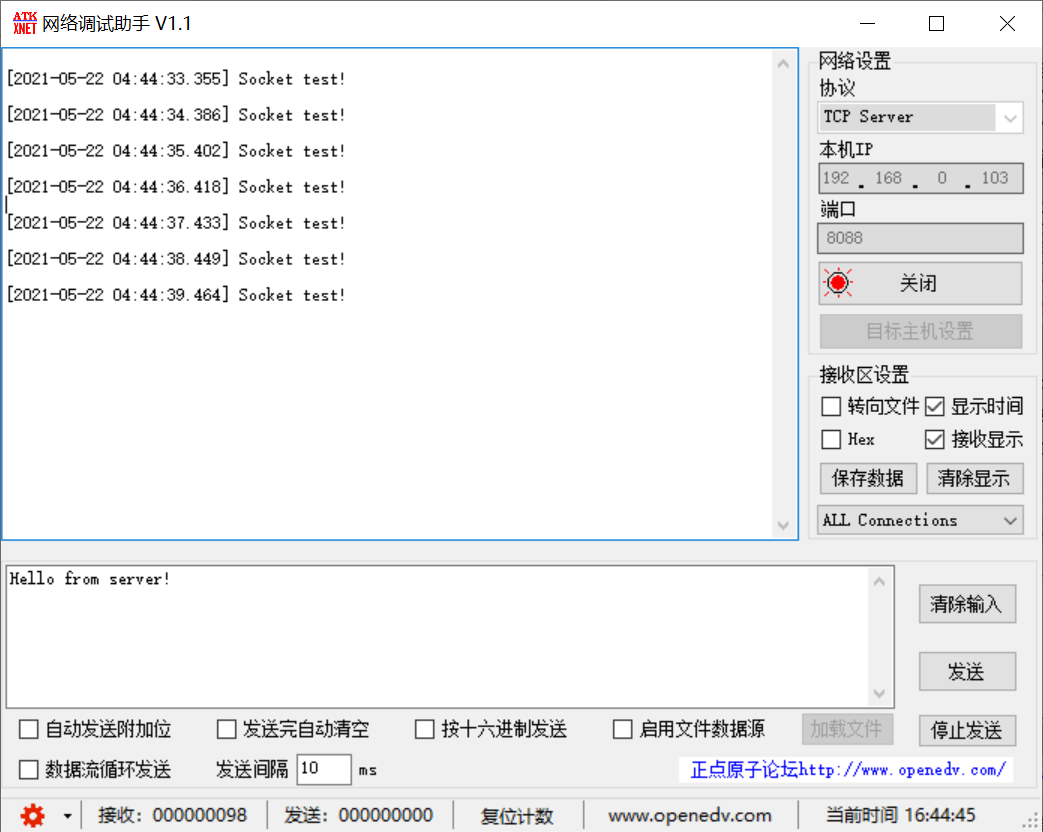
图5.1 操作系统移植测试结果

通过串口调试信息，可以看到操作系统成功初始化并投入运行。此时观察到两个LED分别以不同频率闪烁。测试表明操作系统移植成功且运行正常。

B. 网络通信测试

接下来测试lwIP协议栈的移植情况以及网络通信功能。配置Web服务器IP地址为PC端WLAN网卡地址并设置端口号为8080；配置Socket服务器为相同IP，端口号为8088。

定义HTTP请求发送与Socket通信的相关功能，将开发板通过以太网线连接至路由器，同时在PC运行网络调试助手，启动并观察运行效果。

(a) TCP连接测试 (b) Socket连接测试

图5.2 网络通信测试结果

通过串口信息以及网络调试助手窗口，可以看到协议栈成功初始化，能够通过DHCP自动获取设备IP地址与端口号。按下KEY0键，开发板通过TCP连接至网络调试助手，成功发送测试信；按下KEY1键，开发板通过Socket连接至网络调试助手并能够进行数据接收；按下KEY2键，开发板能够向网络调试助手进行数据发送。测试表明系统HTTP与Socket网络通信功能运行正常。

5.1.2 功能测试

之后测试摄像头与温湿度传感器是否能正常捕获数据并实现传输。将摄像头模块与温湿度传感器模块分别插入开发板的相应接口，编写测试程序使用按键控制传感器捕获与数据传输，启动并观察运行效果。

按下KEY0键，开发板将一帧JPEG数据通过Socket发送至网络调试助手，在16进制下查看发现有效数据部分以“0xFF 0xD8”起始并终止于“0xFF 0xD9”，为合法JPEG数据格式：

b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00`\x00`\x00

\x00\xff\xdb\x00C\x00\x03\x02\x02\x03\x02\x02\x03\x03\x03\x03

\x04\x03\x03\x04\x05\x08\x05\x05\x04\x04\x05\n\x07\x07\x06

……

\x14Q\\\x86\xc3c\x18\xac\xcdZ\x8a+\xae$\x1c\xd5\xd7\xde\xa8\xa8

\xa2\xba\x08\x912\xfd\xca\x86\x8a(2&\xa2\x8a\*\x80(o\xb9E\x14\x01^

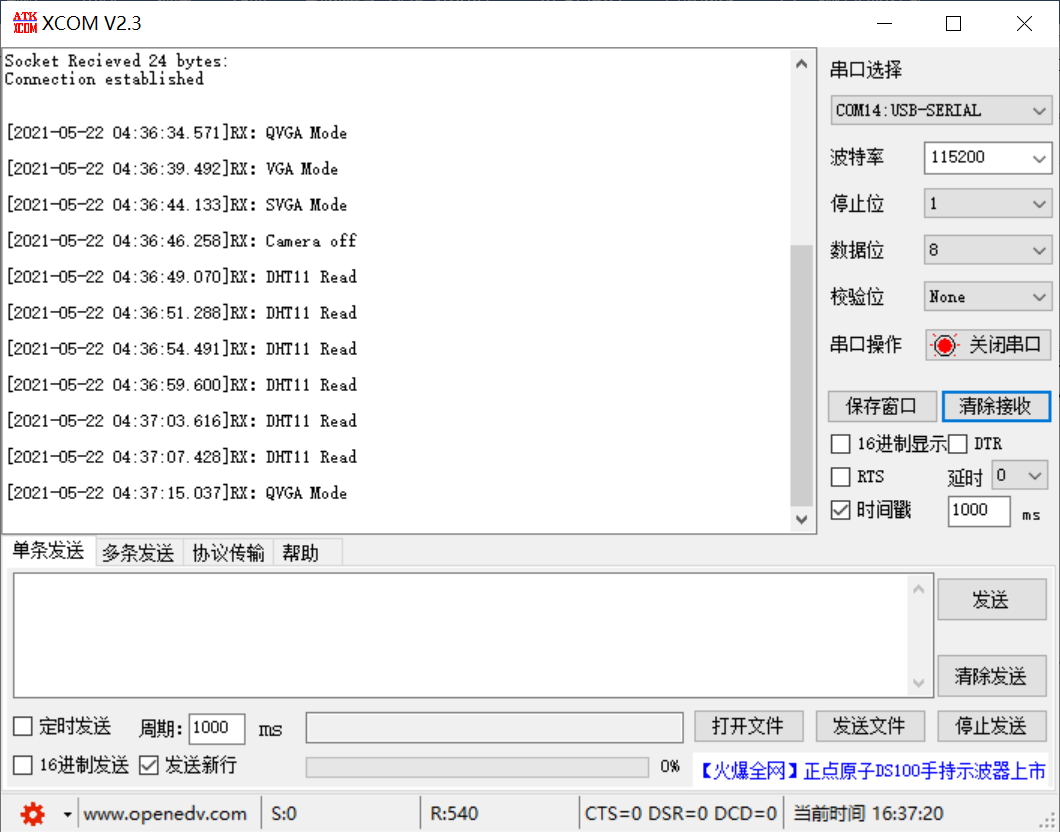
\x99E\x15%\x9f\xff\xd9'

在PC端读取并解析JPEG数据，可以得到一帧来自摄像头的图片。



图5.3 摄像头拍摄效果

按下KEY1键，开发板读取温湿度传感器数据并通过Socket发送至网络调试助手。测试表明传感器捕获与数据传输功能运行正常。

(a) 串口信息 (b) 服务器Socket信息

图5.4 传感器功能测试结果

5.1.3 压力测试

本节主要针对嵌入式设备网络通信能力进行压力测试。在不同摄像头工作模式下，通过按钮调整图像捕获频率，在PC端编写脚本程序，循环读取并记录传输帧率。测试得到不同模式下摄像头线程捕获帧率与实际网络传输帧率之间的关系如下图所示。

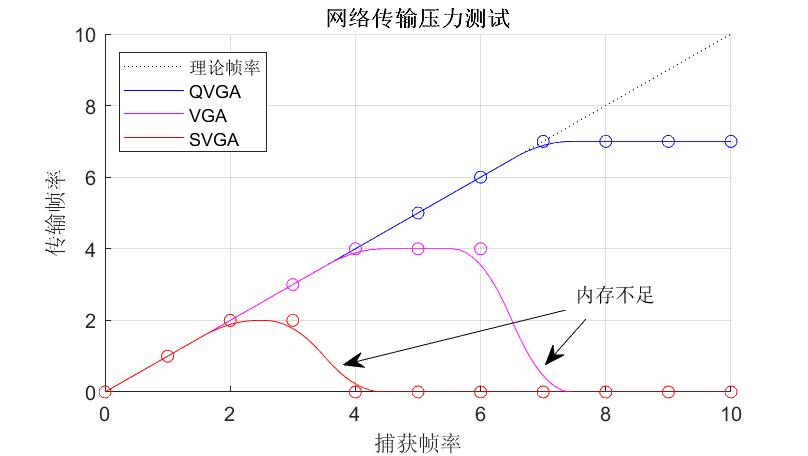


图5.5 网络传输压力测试结果

分析得到当前设计机制下不同图像模式可以达到的最高帧率与实际应用中的建议平均帧率如下表。

表5.2 不同图像模式最高传输帧率比较

|  |  |  |
| --- | --- | --- |
| 模式 | 最高帧率/fps | 建议平均帧率/fps |
| QVGA | 7 | 5 |
| VGA | 4 | 3 |
| SVGA | 2 | 1 |

有关设备传输帧率受限的具体原因分析及可能的解决方案见第六章。

5.2 服务器测试

本节主要进行服务器端的测试覆盖。

为了方便开发与调试，Flask框架提供了测试客户端，可以在不借助托管应用真正启动服务器的条件下模拟向应用发送请求并返回响应数据。本设计基于pytest编写单元测试用例，该模块将自动识别并运行测试目录下全部以test\_开头的文件中所有以test\_开头的函数，并通过断言机制判定测试结果是否符合设计预期。相关测试环境如下表所示。

表5.3 服务器测试环境

|  |  |  |
| --- | --- | --- |
| 环境 | 型号/版本 | 说明 |
| python | 3.7.1 | 开发环境 |
| flask | 1.1.2 | Web服务器框架 |
| click | 7.1.2 | 命令行工具 |

续表5.3

|  |  |  |
| --- | --- | --- |
| 环境 | 型号/版本 | 说明 |
| pillow | 7.2.0 | 图像处理库 |
| pytest | 6.2.2 | 测试框架 |
| pip | 21.0.1 | 包管理工具 |

5.2.1 基于pytest的测试环境配置

由于单元测试中每个测试用例之间都是相互独立的，pytest会通过固件（fixture）为测试用例配置运行环境并提供调用相关方法的接口，使用时只需要将测试所需要的固件作为测试函数参数传入即可。测试配置文件conftest.py定义了测试中可能用到的固件如下图所示。

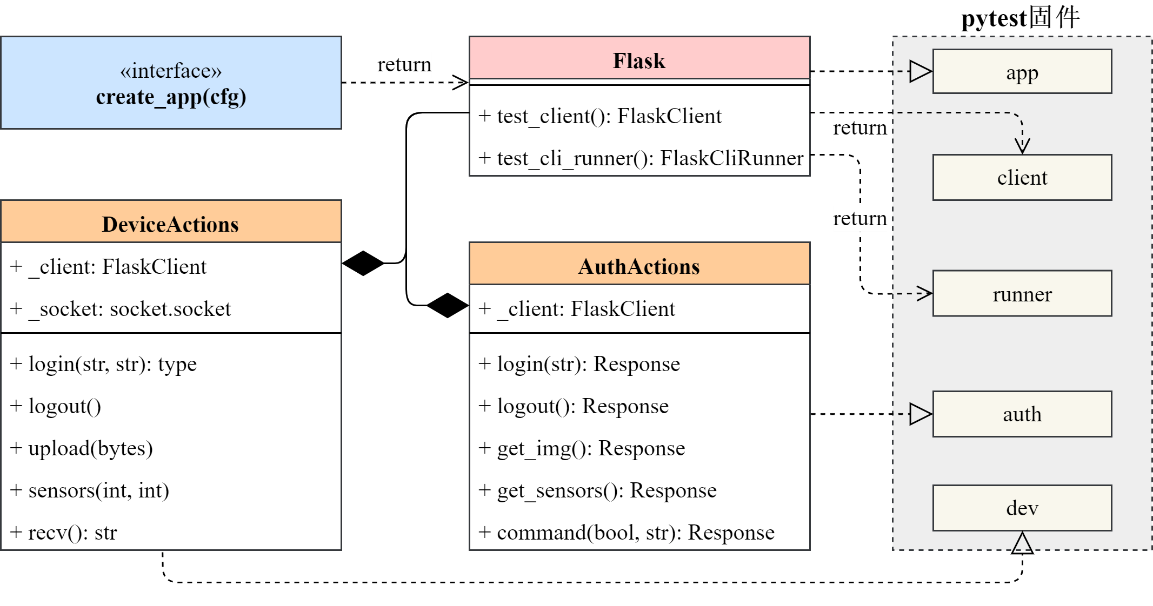


图5.6 pytest测试固件UML图

测试配置阶段首先注册应用固件。注册时需要创建用于测试的临时文件夹，调用厂生成一个应用实例并初始化数据库，最后将该实例作为一个固件返回给pytest。随后，文件又使用Flask应用实例提供的测试客户端以及测试命令行工具分别向pytest注册了客户端固件以及命令行固件以供测试用例调用。

由于本设计中所有功能的实现均依赖于用户管理系统，不妨定义一个用户对象（AuthActions）来管理和模拟用户行为。将客户端固件添加为该类的一个属性，并在成员函数中编写与用户登录和注册相关的操作，之后生成一个该类的实例并将其注册为一个pytest固件（auth）。

类似地，还可以定义一个设备对象（DeviceActions）用以管理和模拟设备行为。该类同样包含一个客户端固件属性负责进行设备校验，另外还包含一个socket属性负责连接Socket服务器并进行数据交互。之后文件同样生一个该类的实例并将其注册为一个pytest固件（dev）。

5.2.2 编写测试用例

基于配置文件中注册的各个固件，可以编写单元测试用例对应用工厂、数据库、用户管理系统、设备校验系统以及数据中转系统进行测试。此处给出全部用例如下表所示。

表5.4 服务器单元测试用例

|  |  |  |
| --- | --- | --- |
| 模块 | 用例 | 内容 |
| 应用工厂测试 | test\_factory | 测试应用工厂 |
| 数据库测试 | test\_get\_close\_db  test\_init\_db\_command | 测试数据库连接与关闭  测试数据库命令行指令 |
| 用户测试 | test\_login\_logout  test\_register | 测试用户登录与登出  测试设备绑定 |
| 设备测试 | test\_no\_device  test\_create\_close\_socket  test\_reconnect | 测试设备连接状态检测  测试Socket连接创建与关闭  测试连接可重入性 |
| 数据中转测试 | test\_upload  test\_commamd  test\_match  test\_sensors  test\_match\_multiple | 测试数据上传  测试控制指令下发  测试用户-设备匹配  测试传感器数据中转  测试多用户匹配 |

由于本设计旨在面向多用户环境，而实际开发过程中实体设备数量受限，因此有必要在单元测试中验证多用户并发情形。特别地，给出多用户匹配测试用例代码如下：

def test\_match\_multiple(auth, auth2, dev, dev2):

    dev.login('test1', '111111')        # 设备1登录

    dev.sensors('11', '11')             # 设备1数据上报

    auth.login('test1')                 # 用户1登录

    rv1 = auth.get\_sensors()            # 用户1数据请求

    dev.logout()                        # 设备1连接关闭

    auth.logout()                       # 用户1登出

    ''' 设备2与用户2登录（略） '''

    dev2.sensors('22', '22')            # 设备2数据上报

    rv2 = auth2.get\_sensors()           # 用户2数据请求

    ''' 设备2与用户2登出（略） '''

    assert b'11' in rv1.data            # 断言用户1数据为‘11’

    assert b'22' in rv2.data            # 断言用户2数据为‘22’

使用pytest运行该测试用例，得到命令行输出结果如下：

> pytest -v tests/test\_user.py::test\_register

========================= test session starts =========================

platform win32 -- Python 3.7.1, pytest-6.2.2, py-1.10.0, pluggy-0.13.1 -- python.exe

cachedir: .pytest\_cache

rootdir: {rootfolder}\server, configfile: setup.cfg

plugins: socket-0.3.5

collected 1 item

tests/test\_user.py::test\_register PASSED [100%]

========================= 1 passed in 0.28s =========================

5.2.3 测试结果

将测试项目路径等配置参数到添加至项目根目录下的setup.cfg文件中，使用pytest运行测试并使用coverage生成测试覆盖率报告，得到结果如下表所示。

表5.5 服务器单元测试覆盖率

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 文件名 | 已执行行数 | 未执行行数 | 分支数 | 未测试分支数 | 覆盖率 |
| \_\_init\_\_.py | 30 | 0 | 2 | 0 | 100% |
| db.py | 24 | 0 | 4 | 0 | 100% |
| device.py | 32 | 3 | 10 | 3 | 86% |
| login.py | 69 | 17 | 18 | 5 | 75% |
| socket\_server.py | 157 | 19 | 22 | 6 | 86% |
| stream.py | 52 | 5 | 8 | 4 | 85% |
| 总计 | 364 | 44 | 64 | 18 | 86% |

测试结果表明服务器各项主要功能运行正常，有关测试覆盖率等问题的讨论与后续改进方向详见章节六。

5.3 交互应用程序测试

本节主要进行微信小程序的测试。

基于已经编写好并测试通过的服务器，可以对微信小程序中的相关功能展开测试。由于微信开发者工具已经集成了面向多种移动端设备的模拟器，因此在最初的调试阶段程序可以直接使用PC编译运行而不需要借助手机，可以等待开发完成后再使用手机设备查看具体效果。相关测试环境如下表所示。

表5.6 小程序测试环境

|  |  |  |
| --- | --- | --- |
| 环境 | 型号/版本 | 说明 |
| 微信开发者工具 | 1.05.2103200 | 小程序集成开发环境 |
| iOS版微信客户端 | 8.0.4 | 真机调试环境 |

5.3.1 编写模拟设备客户端

为配合微信小程序对于数据请求部分功能的测试需要，设计使用Python编写了用于模拟嵌入式设备工作状态的HTTP与Socket客户端测试脚本。

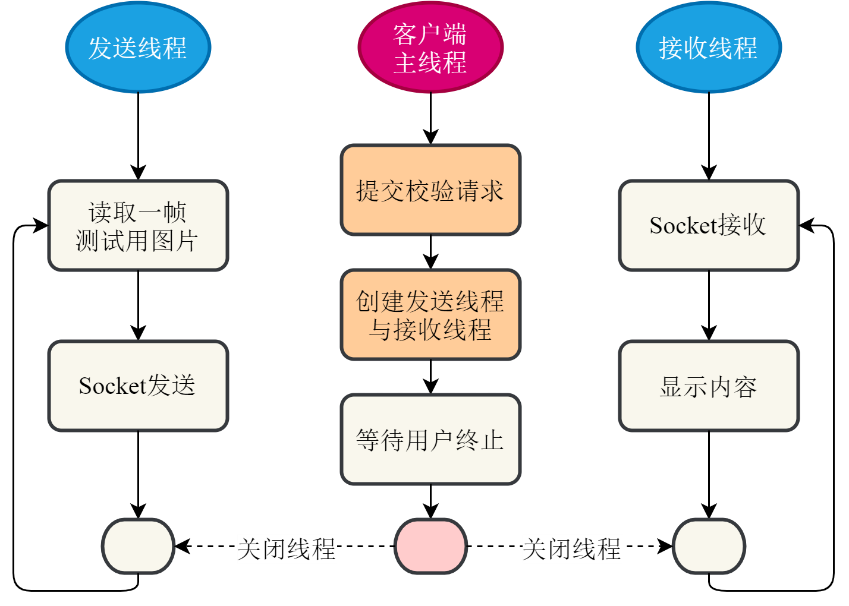


图5.7 模拟设备脚本工作流程

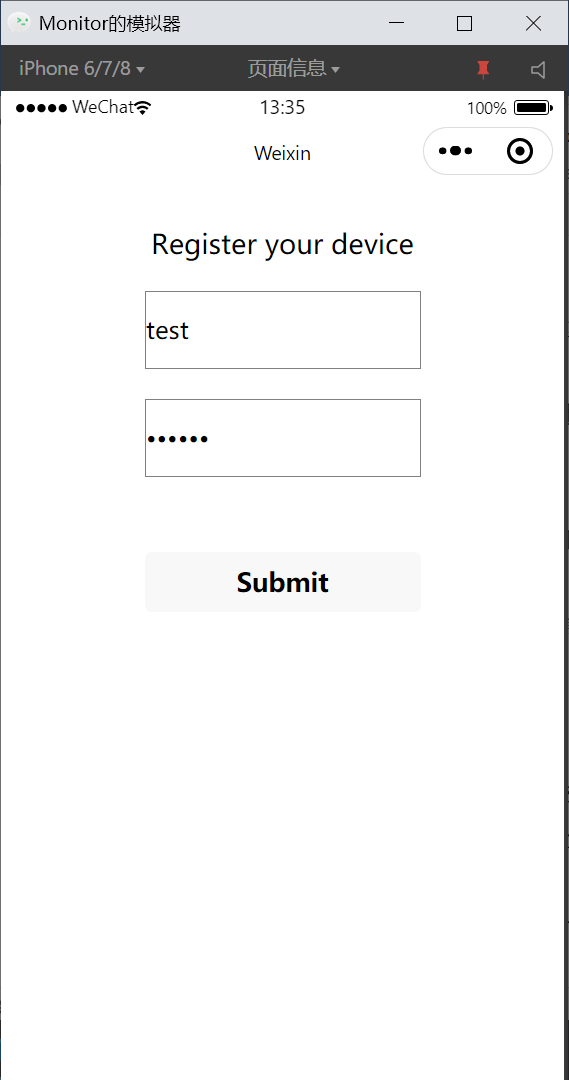
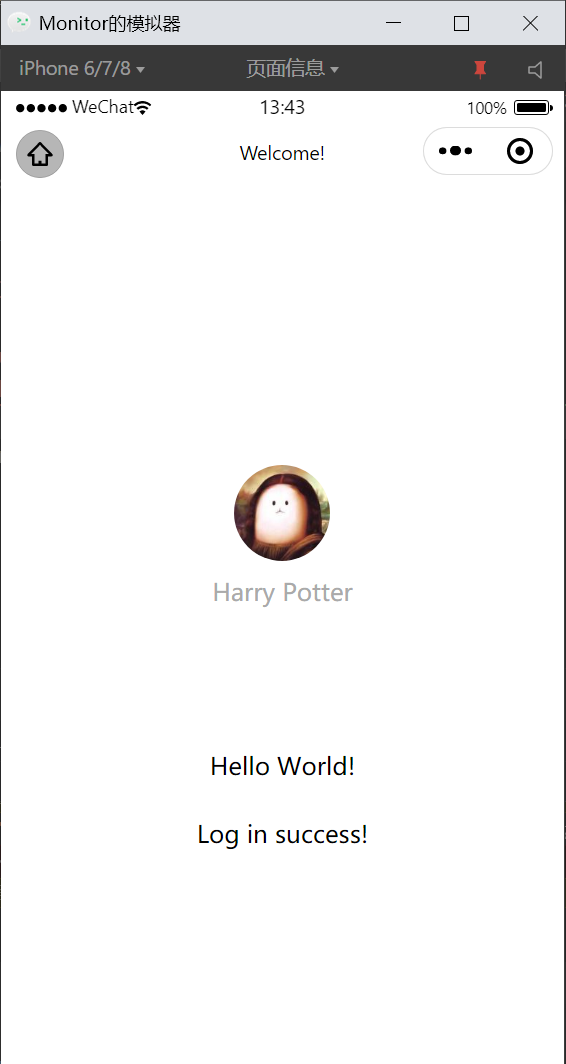
该脚本首先仿照设备启动时的工作方式，使用HTTP向服务器发起设备校验请求，之后使用Socket连接至服务器并上传用于测试的JPEG数据等信息以供小程序在开发测试时进行获取。

5.3.2 小程序功能测试

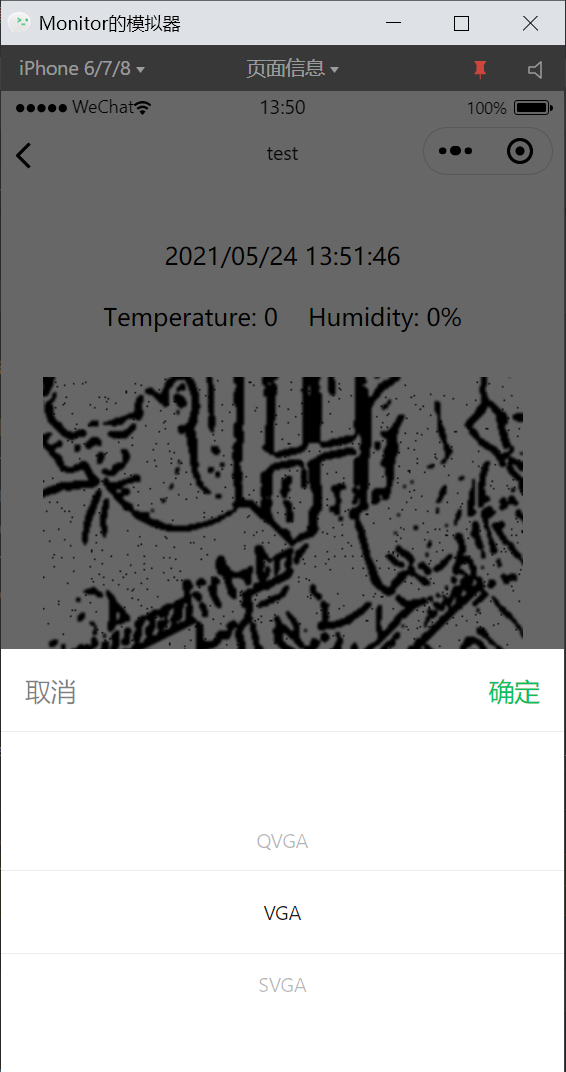
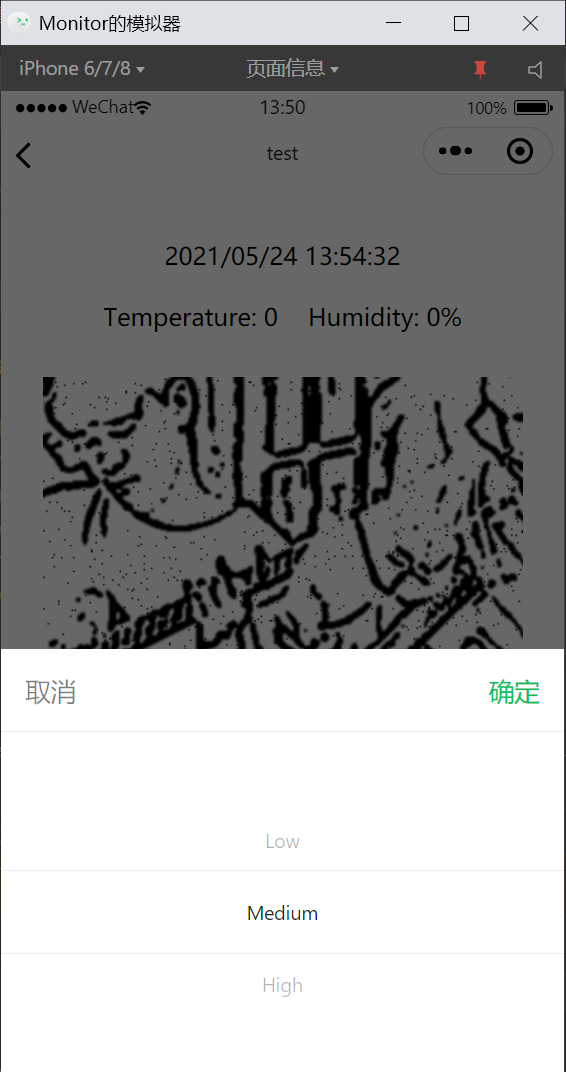
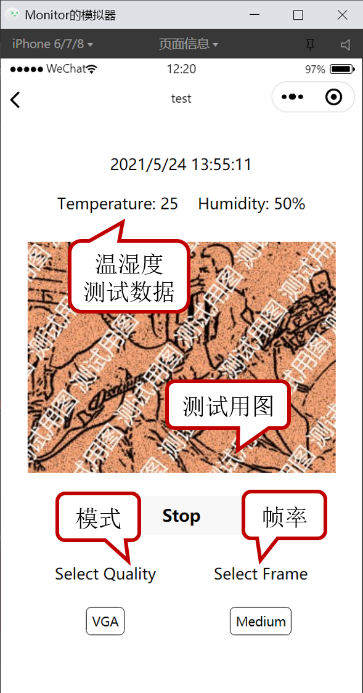
初始化数据库并重启服务器，编译启动小程序，授权小程序获取用户信息并进行登录，之后进入应用首页。

等待调试器中的输出信息显示用户需要进行设备绑定，程序自动跳转至设备绑定页面。输入设备测试ID（“test”）以及测试密钥（“123456”），点击Register按钮，等待设备绑定成功，程序自动跳转至数据显示页面。

在PC端运行模拟设备客户端脚本，通过该页面可以看到脚本上传的测试图片与测试温湿度信息。

(a) 注册页面 (b) 应用首页

(c) 模式选择 (d) 帧率选择 (e) 测试效果

图5.8 小程序测试结果（iOS环境）

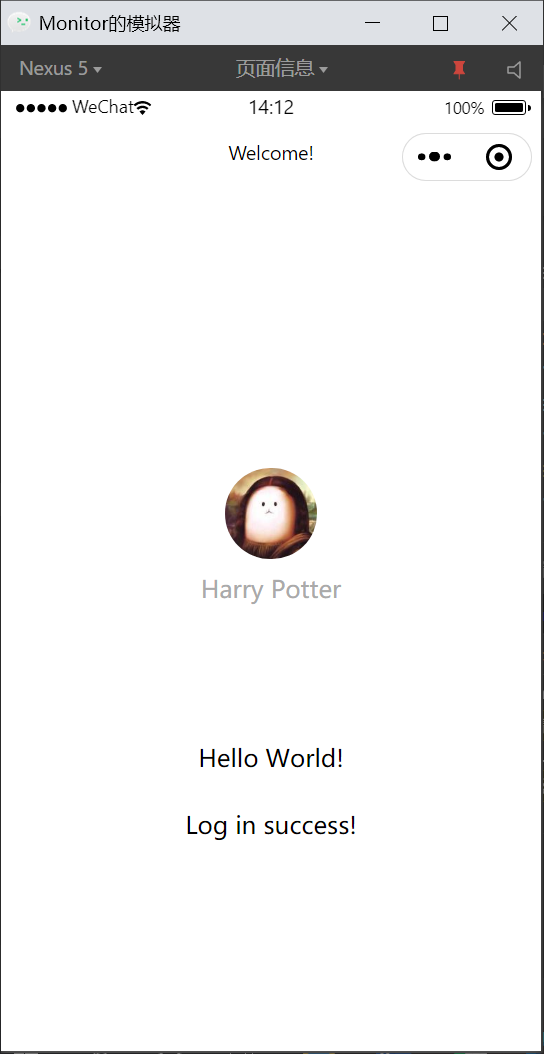
特别地，查看调试器输出，解析得到带有cookie的登录请求响应部分内容如下：

{data: "OK", …, statusCode: 200, cookies: Array(1), …}

cookies: Array(1)

0: "session=eyJvcGVuaWQiOiJvN01TTzFnOVVPeWIxNTlwUDBWSTlJUmFSWHZzIn0.YKtBeA.Jk4\_NE62CFuey1fbnfxbNBdqFzo; HttpOnly; Path=/"

改用安卓客户端模拟器，重新进行上述步骤以测试程序移植能力，得到结果如下。

(a) 应用首页 (b) 功能选择 (c) 测试效果

图5.9 小程序测试结果（安卓环境）

测试表明微信小程序各项功能运行正常，并且可以进行跨平台运行。

5.4 系统综合测试

本节对整个系统进行综合测试。

为了使服务器能够真正运行在一个WSGI托管服务器上并接受来自嵌入式设备与手机的网络信息，需要首先使应用项目可安装化。在项目根目录下添加setup.py文件，在其中指定包名称、版本、目录以及依赖项等信息，并在MANIFEST.in文件中指定其他需要包含的文件夹，之后使用pip工具将项目打包为一个可安装的Python包。

表5.7 项目安装情况

|  |  |  |
| --- | --- | --- |
| 包 | 版本 | 路径（默认为lib\site-packages） |
| click | 7.1.2 | - |
| Flask | 1.1.2 | - |
| flaskr | 1.0.0 | server |
| itsdangerous | 1.1.0 | - |
| Jinja2 | 2.11.2 | - |
| Pillow | 8.2.0 | - |
| pip | 21.1.1 | - |

续表5.7

|  |  |  |
| --- | --- | --- |
| 包 | 版本 | 路径（默认路径：lib\site-packages） |
| setuptools | 49.2.0.post20200714 | - |
| waitress | 2.0.0 | - |
| Werkzeug | 1.0.1 | - |
| wheel | 0.34.2 | - |

打包完成后进行项目部署工作。使用wheel工具为应用构建一个.whl格式的发行版，之后复制该文件至部署位置，创建虚拟环境并使用pip命令安装即可。

投入正式运行前，在项目实例文件夹创建一个config.py文件用以装载配置信息如随机密钥、小程序AppID与SecretKey等。应用工厂在被调用时能够自动识别该文件并导入相关的配置。特别地，给出本设计遵循的部分配置如下：

SECRET\_KEY = b'&random\_string'          # 项目密钥（真实值已隐藏）

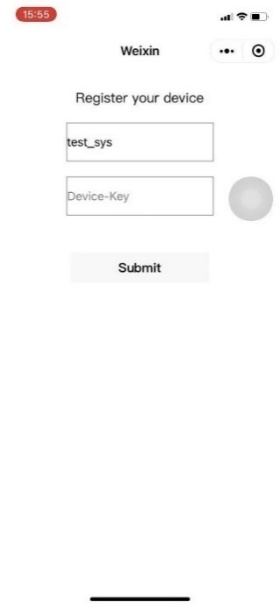
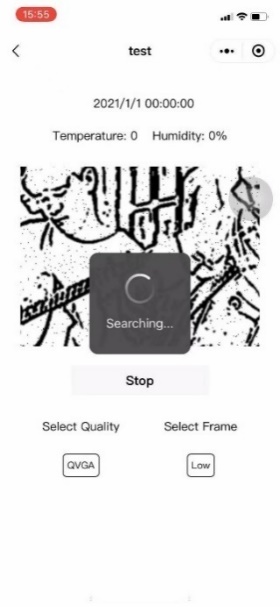
SERVER\_IP = '192.168.0.103'             # PC端WLAN网卡IP地址

SOCKET\_PORT = 8088                      # Socket服务器端口号

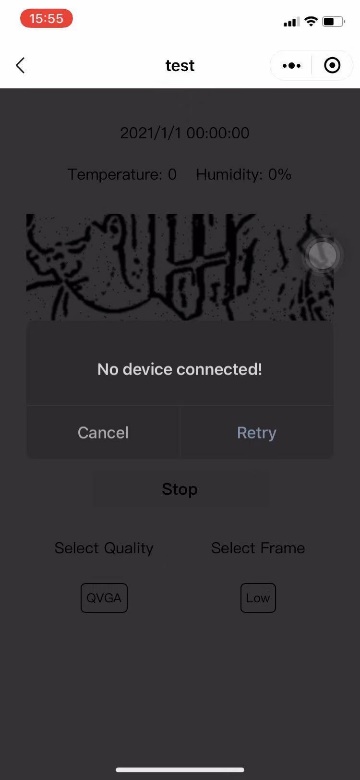
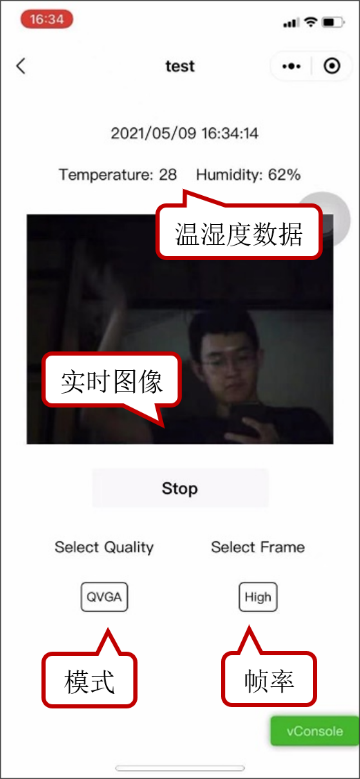
APP\_ID = '&appid'                       # 小程序ID（真实值已隐藏）

APP\_SECRET = '&appsecret'               # 小程序密钥（真实值已隐藏）

在将应用投入运行时，使用产品级别的WSGI服务器替代flask的内建开发服务器，本设计选用waitress作为应用托管对象。在服务器开始运行后，启动嵌入式设备，等待其初始化完成并连接至服务器。之后启动微信小程序，按提示注册绑定设备信息并完成登录。待应用跳转至数据显示页面后，即可通过屏幕看到来自嵌入式设备的摄像头数据与温湿度信息。

(a) 用户注册 (b) 应用首页 (c) 查询设备

(d) 设备下线 (e) 设置参数 (f) 数据交互

图5.10 系统综合测试结果

测试表明整个系统各部分功能运行正常且相互之间能够正确协同。给出项目最终评价指标如下表所示。

表5.8 项目评价指标

|  |  |  |
| --- | --- | --- |
| 指标 | 是否完成 | 累计开发时间/天 |
| 移植μC/OS-II | √ | 3 |
| 移植lwIP | √ | 3 |
| 实现HTTP通信 | √ | 2 |
| 实现Socket通信 | √ | 2 |
| 实现摄像头捕获 | √ | 7 |
| 实现温湿度传感器捕获 | √ | 3 |
| 实现用户控制指令响应 | √ | 2 |
| 搭建基于Flask的Web服务器框架 | √ | 5 |
| 搭建Socket服务器 | √ | 5 |
| 实现服务器用户管理 | √ | 2 |
| 实现服务器设备校验 | √ | 3 |
| 实现服务器数据中转 | √ | 5 |
| 实现小程序用户登录 | √ | 3 |

续表5.8

|  |  |  |
| --- | --- | --- |
| 指标 | 是否完成 | 累计开发时间/天 |
| 实现小程序设备绑定 | √ | 2 |
| 实现小程序数据交互 | √ | 2 |
| 实现小程序设备控制 | √ | 2 |
| 实现项目测试覆盖 | √ | 10 |

5.5 本章小结

本章主要针对系统各模块进行了单元测试，包括嵌入式设备软环境、功能实现与负载能力，同时测试了服务器各项接口功能以及微信小程序运行效果。之后对整个系统进行了综合测试并给出了最终评价指标。

6 总结与展望

本设计完成了一个基于STM32的家庭智能终端，开发了嵌入式设备、服务器与交互应用程序三个子系统，并实现了各个模块间的协同。在综合测试中，嵌入式设备能够完成摄像头与温湿度数据的采集上报与用户指令响应，服务器能够正确实现用户与设备的匹配及数据的暂存和中转，微信小程序能够进行用户登录注册、数据显示与设备控制。

回顾项目开发过程并结合测试结果，总结发现本设计在硬件选型、程序性能、测试覆盖以及数据安全性方面仍在存在问题，具体如下。

A. 开发板硬件设备及接口冗余

由于控制器选用了正点原子公司的探索者开发板，产品出于功能的丰富性与通用性考虑添加了大量硬件接口，而设计中真正使用到的仅为其中的一部分。选择使用开发板尽管节省了硬件电路设计与焊接的步骤，但导致了包括USB接口、TFTLCD接口、RS232、扩音器以及SD读卡器等设备的冗余，并且带来了引脚重合问题从而限制了部分功能之间的协同。

在后续改进中，可根据需求自行设计硬件电路与接口，在保证功能的前提下最大限度减小不必要的开销。

B. 图像传输帧率偏低

嵌入式设备压力测试中发现，尽管摄像头的捕获帧率理论上可以达到每秒15帧，但实际通过Socket传输的速率在QVGA格式下最高仅能稳定在每秒6帧左右；而VGA与SVGA格式下所能达到的帧率则更低。分析发现，这种现象主要由以下两方面的原因导致。

a. 协议栈机制缺陷

在操作系统的支持下，本设计用于处理网络通信的lwIP协议栈在内核层与应用层之间所有的数据交互全部通过单一的内核线程完成。因此，网络通信相关任务的处理速度不仅受限于处理器的运算能力，同时也受限于操作系统的任务调度效率。

考虑到lwIP协议栈专门针对小型嵌入式系统相对简单的网络通信需求而设计，并且其BSD Socket Api封装也仅仅适用于开发测试而不适用于投产，则本设计若想达到更高的数据传输速率，需要借助更加高端而完善的网络协议栈的支持。

b. 处理器内存不足

由于操作系统、协议栈内核与用户任务占用了大量的处理器栈区空间，本设计中lwIP下的所有网络数据缓冲区以及包括摄像头数据缓存在内的各项用户自定义缓冲区均位于堆区，从而导致系统对于处理器内存空间的占用率偏高。

在系统运行过程中发现，处理器自带的1M Flash以及用于动态内存分配的192k SRAM在各项功能完全启动后已经几乎被完全占满，这也导致了系统无法为摄像头及网络数据包分配更大的缓存空间，进而限制了摄像头捕获速度以及协议栈的数据吞吐能力。如果想在当前的网络通信机制下更好的挖掘系统软件潜能，可以考虑在项目经费允许的条件下更换内存空间更大、处理速度更快的芯片作为控制器。

特别地，给出嵌入式设备工程项目生成结果如下表所示。

表6.1 嵌入式设备工程项目生成结果

|  |  |
| --- | --- |
| 项目 | 大小/B |
| 程序代码 | 74954 |
| 只读数据 | 2206 |
| 已初始化的读写数据 | 452 |
| 未初始化的读写数据 | 1229140 |

C. 缺乏基于实体设备的并发测试

本设计在服务器端采用了面向多用户的设计机制，实现了对于设备与用户的登录身份管理。然而由于资金问题，本设计只有一个开发板能够作为嵌入式设备客户端。设计尽管在服务器单元测试用例中加入了多用户并发条件下的数据中转测试，但这一测试缺乏真正的实体设备，从而也无法完全模拟真实情况且并发规模有限。

在服务器部分的单元测试覆盖率方面，由于程序中存在大量的边界条件判断，代码分支结构较为繁杂，在测试用例编写时并未对全部模块实现100%的覆盖率。这一问题在后续的开发中仍需要进行完善。此外，设计在Web服务器部分采用的Flask框架是一款成熟的、经过检验的、能够用于实际部署的服务器框架；而Socket服务器部分自行编写的代码并没有经过系统而全面的压力测试，因此尽管其在本设计中表现良好哦，但当系统规模扩大时仍然可能存在一定的安全风险。这一问题在后续的开发设计中需要继续改进。

D. 数据安全性方面有待提高

尽管系统在架构设计阶段就数据安全问题进行了参考，并且尝试从模块协同方式以及通信接口规范等角度对系统进行安全保护，但这些措施仍然是初级且未经严格检验的。

在后续开发过程中，若想要进一步提高系统数据安全性，应首先对服务器进行SSL安全认证，采用HTTPS全面代替HTTP进行通信；同时也应相应的在嵌入式设备端配置与SSL相关的协议沟通与加解密等内容，从而在更大程度上保证用户的数据安全。

总体来说，项目较好的遵循了设计阶段提出的研究方案与研究路线，实现了全部预期功能并通过了多项测试，但在一些方面仍有较大的改进空间。由于本设计所采用的系统架构具有较好的可维护性与可扩展性，后续开发过程中可以在当前设计的基础上针对现存问题进行进一步的改进与提升，以期达到更加完善的功能。

参考文献

[1] 王工儒. 智能家居系统工程实用技术[M]. 中国铁道出版社有限公司, 2019.

[2] 邓亮才,肖卫初,黄楚喻,等. 基于STM32的书童机器人控制系统设计与实现[J]. 湖南城市学院学报(自然科学版), 2021, 30(1): 70-75.

[3] 李淑敬,孙楠,柴文君,等. 基于WebSocket服务型机器人远程交互的开发[J]. 西安文理学院学报(自然科学版), 2021, 24(1): 52-58.

[4] Lu li,Changkun wang,Fuqiang zhao. A Smart Home Control System Integrated the Improved Streaming Media Technology and Raspberry Pi[C]//Proceedings of 2015 Ssr International Conference on Social Sciences and Information(ssr-ssi 2015 V11), Singapore Management and Sports Science Institute, 2015: 656-659.

[5] Zhang ya'nan,Xiao guijin,Xu jiansheng. The Wireless Image Transmission System of Capsule Endoscope Based on Stm32f103[C]//Proceedings of 2016 2nd International Conference on Mechanical,electronic and Information Technology Engineering(icmite 2016), Destech Publications, 2016: 309-315.

[6] Deshun fan,Jichun zhao. The Design of Image Transmission System Based on Stm32f429zi[C]//Proceedings of 2017 4th International Conference on Machinery,materials and Computer(macmc 2017), Atlantis Press, 2017: 256-260.

[7] Zheng liu,Gang du,Wenhui zhuang, et al. Design of Camera Type Handling Trolley Based on Stm32[C]//Proceedings of 2019 3rd International Conference on Computer Engineering,information Science and Internet Technology(cii 2019), Clausius Scientific Press,canada, 2019: 145-150.

[8] 文馨,朱思曼,廖香芝. 基于微信小程序的学习平台研究[J]. 家庭科技, 2021, (1): 14-15.

[9] 朱宝明. 基于ZigBee和STM32的智能家居系统的研究与设计[D]. 华北理工大学, 2018.

[10] 刘后文,唐成章. 基于STM32的物联网智能家居控制系统分析[J]. 集成电路应用, 2021, 38(1): 16-17.

[11] Ghoraani behnaz,Galvin james e.,Jimenez-shahed joohi. Point of View: Wearable Systems for At-home Monitoring of Motor Complications in Parkinson's Disease Should Deliver Clinically Actionable Information[J]. Parkinsonism and Related Disorders, 2021, 84: .

[12] Wang ju,Spicher nicolai,Warnecke joana m., et al. Unobtrusive Health Monitoring in Private Spaces: the Smart Home[J]. Sensors, 2021, 21(3): .

[13] 高天,朱晨旭,刘博通,等. 基于OneNet云平台的多功能防盗监测系统设计[J]. 电子制作, 2020, 393(7): 25-28, 51.

[14] 黄松茂. 基于STM32的家庭环境监测系统的设计与实现[D]. 西北师范大学, 2018.

[15] 杨高扬. 远程遥控与家庭安防聊天机终端设计[D]. 西南交通大学, 2018.

[16] 鲍敏. 家用智能摄像头的网络安全问题及应对策略分析[J]. 现代信息科技, 2019, 3(13): 172-174.

[17] 柯亚文,蔡挺,夏晓峰,等. 面向嵌入式系统的加密算法性能检测方法[J]. 重庆大学学报, 2020, 43(11): 1-10.

[18] 夏祥祥. 基于UCOSⅢ的嵌入式无线照片采集系统设计[J]. 科学技术创新, 2021, (2): 171-172.

[19] 褚治广,陈昊,张兴. 基于微信小程序的智能家居系统[J]. 辽宁工业大学学报(自然科学版), 2020, 40(6): 362-366.

[20] Evaluation of the Effectiveness, Implementation and Cost-effectiveness of the Stay One Step Ahead Home Safety Promotion Intervention for Pre-school Children: a Study Protocol.[J]. Injury Prevention : Journal of the International Society for Child and Adolescent Injury Prevention, 2020, 26(6): .

[21] Evan r. polzer ma,Kathryn nearing phd M,Christopher e. knoepke msw P, et al. “safety in Dementia”: Development of an Online Caregiver Tool for Firearm, Driving, and Home Safety[J]. Journal of the American Geriatrics Society, 2020, 68(9): .

[22] 王凯彬,李梦思,陈佳雪,等. 基于STM32的居所智能防盗系统设计[J]. 山西电子技术, 2020, 213(6): 26-28.

[23] 王丹,刘国栋,张海涛,等. 基于ARM的嵌入式系统开发[J]. 微处理机, 2021, 42(1): 62-64.

[24] 辛慧娟,肖军. 基于NB-IoT技术的多传感器数据采集系统设计[J]. 微型电脑应用, 2020, 36(12): 35-37.

[25] Ghorayeb abir,Comber rob,Gooberman hill rachael. Older Adults' Perspectives of Smart Home Technology: Are We Developing the Technology That Older People Want?[J]. International Journal of Human-computer Studies, 2021, 147: .

[26] 张佳君. Design of Wireless Target Detection System Based on Stm32[D]. 华中师范大学, 2020.

[27] 盛琦鑫. 基于WebSocket和MSE的视频监控Web页播放的实现方法[J]. 广播电视网络, 2020, 27(12): 62-66.

[28] 刘宇健. 流媒体网络视频监控系统的研究与开发[J]. 电子技术与软件工程, 2020, 188(18): 15-17.

[29] 谈学钊,肖磊. 基于物联网云平台的热水实时监控设计[J]. 计算机与网络, 2020, 46(20): 67-70.

[30] 姜艳茹,孟令军,尚桠朝,等. 基于STM32的图像采集与网络传输系统设计[J]. 自动化与仪表, 2020, 35(12): 45-49, 53.

谢 辞

提笔写下这段文字的时候，毕业设计的相关工作已经断续绵延近五个月，而今终于隐约可以看到完成的曙光。拔掉三四根数据线、关闭七八个程序窗口、归档不知道多少个文件，蓦然抬头，才发现毕业这件一直以来缺乏实感的事情此时倏地出现在眼前。恍惚四年间，除去大一结束搬家、大二半截选专业以及大三敲定申请出国之外，很少有任何其它事情的存在提示着时间的飞逝。鲜有知觉地，人生的列车再一次来到了换乘站。

四年前我拿着一张高考成绩单稀里糊涂的来到了同济。感谢阿立、吴广伯伯和陈增堂老师作为过来人的帮助。感谢大活，让我从大一开始就得到机会为一些有意义的事毫无保留地投入自己。感谢原来的以及现在的室友们，漫长的四年里离不开各位的包容与搀扶。艰难的申请季，感谢李莉老师、余有灵老师和赵霞老师欣然接受我厚着脸皮的推荐信请求，感谢邬同学的鼓励和赵同学毫无保留的信息共享，尤其感谢父母对我个人决定一如既往的尊重和支持。当然，感谢余有灵老师在毕业设计阶段给予我的悉心指导与宝贵建议，这也是本文最终能够出现在这里的重要原因。

如果说四年大学教会了我什么生活哲理，那么最重要的大概就是人要活在当下。四年里我试着望向前路却总是不能够清晰的分别，直到发现最好的选择便是低下头来努力吃好每一顿饭、睡好每一个觉、写好每一行代码，努力去爱身边的人，同时尝试赋予每一个今天更多存在的意义，如此便是在不自觉地向前走着。要感谢gc和甘老师，谢谢我们有出现在彼此大学生活中相当一部分时刻，谢谢两位的不嫌弃与始终如一的支持；也要感谢人生中有幸遇到的每一位我实在无法一一列举的朋友，谢谢你们给予的所有陪伴并且成为我心中无比坚实的后盾。最后的最后要感谢吴老师，谢谢你让我意识到自己不想仅仅生活在当下，我开始抬起头来，期盼着与你迎接每一个明天的太阳。

如今本科阶段即将结束，研究生更加厚重的日子也开始在即。在此，希望自己于今后漫长的时光里能够始终拥有张开双臂拥抱生活的能力，毕竟欣喜也好失落也罢，到头来最重要的可能就是我们一直都还在路上这件事。

人生没有终点的到达，人生只是不断出发。

附 录

附录1：嵌入式设备网络通信部分功能函数（Terminal/LWIP/lwip-app/apps.c）

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* apps.c

\* - 网络功能线程定义

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

//recieve task

void socket\_recv\_task(void \*args)

{

    LWIP\_UNUSED\_ARG(args);

    OSTaskSuspend(OS\_PRIO\_SELF);

    while(1)

    {

        int ret = read(sock, NET\_RX\_BUF, RX\_BUFSIZE);

        if(ret > 0 && ret < RX\_BUFSIZE - 1)

        {

            NET\_RX\_BUF[ret] = '\0';

            if(strlen(NET\_RX\_BUF) > 5)

            {

                printf("Socket Recieved %d bytes:\r\n%s\r\n", ret, NET\_RX\_BUF);

                sprintf(NET\_RX\_BUF, "");

                continue;

            }

            else

                printf("Command:\r\n0x%X\r\n", (u8)NET\_RX\_BUF[0]);

            u8 cmd = NET\_RX\_BUF[0];

            sprintf(NET\_RX\_BUF, "");

            dht\_on = cmd & 0x01;

            if(dht\_on)

                printf("DHT: on\r\n");

            cam\_on = (cmd & 0x02) >> 1;

            if(cam\_on)

            {

                u8 mode = ((cmd & 0x0C) >> 2) + 1;

                if(mode > 0 && mode < 4)

                {

                    camera\_mode\_set(mode);

                    printf("CAM: on\r\nMODE: %d\r\n", mode);

                }

                else

                    printf("Invalid Command\r\n");

            }

            u8 frame\_n = (cmd & 0x30) >> 4;

            printf("FRAME: %d\r\n\r\n", frame\_n);

            switch(frame\_n)

            {

                case 0:

                {

                    frame = 1;

                    break;

                }

                case 1:

                {

                    frame = 3;

                    break;

                }

                case 2:

                {

                    frame = 5;

                    break;

                }

                default:

                    frame = 1;

            }

        }

        delay\_ms(1000);

    }

}

//http get task

void http\_get\_task(void\* args)

{

    LWIP\_UNUSED\_ARG(args);

    ip\_addr\_t ip\_addr = resolve\_ip(SERVER\_IP);

    while(1)

    {

        tcp\_conn = netconn\_new(NETCONN\_TCP);

        netconn\_set\_recvtimeout(tcp\_conn, TCP\_RECV\_TIMEOUT);

        u16\_t port = SERVER\_PORT\_TCP;

        err\_t err = netconn\_connect(tcp\_conn, &ip\_addr, port);

        if(err != ERR\_OK)

        {

            netconn\_delete(tcp\_conn);

            printf("Connection Failed with Code %d\r\n", err);

        }

        else

        {

            OSSemPost(sem\_http\_rdy);

            tcp\_conn->recv\_timeout = 100;

            while(1)

            {

                INT8U err\_sem;

                OSSemPend(sem\_req, 0, &err\_sem);

                u8 len = strlen(HTTP\_TX\_BUF);

                err = netconn\_write(tcp\_conn, HTTP\_TX\_BUF, len, NETCONN\_COPY);

                if(err != ERR\_OK)

                    printf("Netconn Write Failed!\r\n");

                else

                {

                    printf("Netconn Sent: %d bits\r\n", len);

                    sprintf(HTTP\_TX\_BUF, "");

                    u8 timeout = TCP\_RECV\_TIMEOUT;

                    while(timeout > 0)

                    {

                        err = fetch\_rx\_data(tcp\_conn, HTTP\_RX\_BUF);

                        if(err == ERR\_OK)

                        {

                            netconn\_close(tcp\_conn);

                            netconn\_delete(tcp\_conn);

                            OSSemPost(sem\_resp);

                            break;

                        }

                        else if(err == ERR\_TIMEOUT)

                            --timeout;

                        else

                        {

                            printf("Recieve Failed with Code: %d\r\n", err);

                            break;

                        }

                        delay\_ms(100);

                    }

                    if(timeout <= 0)

                        printf("HTTP Response Timeout!\r\n");

                }

            }

        }

        delay\_ms(1000);

    }

}

//apps main task

void app\_main\_task(void\* args)

{

    LWIP\_UNUSED\_ARG(args);

    NET\_RX\_BUF = mymalloc(SRAMIN, RX\_BUFSIZE);

    HTTP\_RX\_BUF = mymalloc(SRAMIN, RX\_BUFSIZE);

    HTTP\_TX\_BUF = mymalloc(SRAMIN, TX\_BUFSIZE);

    sprintf(HTTP\_TX\_BUF, "GET /device/?device\_id=%s&device\_key=%s HTTP/1.1\r\n"

                         "Host: http://%s:%d\r\n"

                         "\r\n\r\n",

DEVICE\_ID, DEVICE\_KEY, SERVER\_IP, SERVER\_PORT\_TCP);

    INT8U err;

    OSSemPend(sem\_http\_rdy, 0, &err);

    while(verify\_device())

    {

        printf("Verify Device Failed! Retrying...\r\n");

        delay\_ms(1000);

    }

    socket\_connect();

    OSTaskSuspend(OS\_PRIO\_SELF);

}

附录2：Socket服务器部分模块（server/flaskr/socket\_server.py）

''' socket\_server.py - Socket服务器模块定义

'''

# socket thread class

class SocketThread(threading.Thread):

    ''' websocket thread '''

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        self.buffer = SocketBuffer(

            Sensors(),

            Queue(5)

        )

        self.running = True

        self.conn = None

        return super().\_\_init\_\_(\*args, \*\*kwargs)

    def push\_img(self, item):

        try:

            self.buffer.images.put\_nowait(item)

        except:

            pass

    def pop\_img(self):

        return self.buffer.images.get\_nowait()

    def push\_sensors(self, temp=0, humid=0):

        self.buffer.sensors.set(temp, humid)

    def pop\_sensors(self):

        return self.buffer.sensors.get()

    def close(self):

        self.running = False

# socket server class

class SocketServer:

    ''' websocket server

    :singleton

    '''

    def \_\_init\_\_(self, ip='127.0.0.1', port=6000, bufsize\_k=500):

        ''' create a SocketServer instance

        :threads - a dict that stores all SocketThreads

        :DO NOT get new instance after initializiation

        '''

        self.s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

        self.s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

        self.s.bind((ip, port))

        self.s.listen(5)

        self.threads = {}

        self.address = {}

        self.bufsize = bufsize\_k \* 1024

        self.handler\_timeout = 0.1

        self.connect\_timeout = 5

    # singleton

    @classmethod

    def instance(cls, \*args, \*\*kwargs):

        ''' one way to realize singleton '''

        if not hasattr(SocketServer, "\_instance"):

            SocketServer.\_instance = SocketServer(\*args, \*\*kwargs)

        return SocketServer.\_instance

    # refresh

    def refresh\_handler(self, t):

        ''' handles data recieving

        :a sub-thread of main

        '''

        while t.running:

            try:

                data = t.conn.recv(self.bufsize)

            except WindowsError as win\_err:

                print(win\_err)

                break

            except Exception as e:

                continue

            if not data:

                break

            elif len(data) > 10:

                try:

                    img = Image.open(BytesIO(data))

                    t.push\_img(img)

                except Exception as e:

                    print(e)

                    continue

            else:

                try:

                    temp, humid = map(int, data.decode().strip('\r\n').split())

                    t.push\_sensors(temp, humid)

                except Exception as e:

                    print(e)

                    continue

    # handler

    def socket\_handler(self):

        ''' handler for socket thread '''

        self.s.settimeout(5)

        try:

            conn, addr = self.s.accept()

        except Exception as e:

            print(e)

            return

        self.s.settimeout(None)

        ip = addr[0]

        if not self.verify\_ip(ip):

            print('IP {} unmatched'.format(ip))

            conn.close()

            return

        conn.send(b'Connection established\r\n')

        print('Accepted socket from {}'.format(ip))

        t = self.threads[ip]

        t.conn = conn

        subt = threading.Thread(

            target=self.refresh\_handler,

            args=(t,)

        )

        subt.setDaemon(True)

        subt.start()

        while t.running:

            subt.join(timeout=self.handler\_timeout)

        conn.close()

        print('Socket {} closed'.format(ip))

        self.threads.pop(ip)

    # create new socket

    def create\_socket(self, dev\_id, dev\_ip):

        ''' a method that creates a new websocket '''

        if dev\_ip in self.threads.keys():

            self.close\_socket(dev\_id)

        t = SocketThread(target=self.socket\_handler)

        self.threads[dev\_ip] = t

        self.address[dev\_id] = dev\_ip

        t.start()

        print('Socket created for {}: {}'.format(dev\_id, dev\_ip))

        return 'OK'

    # close a socket

    def close\_socket(self, dev\_id):

        ''' close a websocket '''

        try:

            ip = self.address[dev\_id]

            t = self.threads[ip]

            t.close()

            t.join()

            return 'OK'

        except KeyError:

            return None

附录3：小程序数据交互页面对象部分成员（miniprogram/pages/stream/stream.js）

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* stream.js

\* - 交互页面实例注册与运行逻辑

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

Page({

//periodically refresh current page

refresh (e) {

    var alv = this.data.alive

    var btn = this.data.button

    if (btn == 'Start') {

      btn = 'Stop'

    } else {

      btn = 'Start'

    }

    this.setData({

      alive: !alv,

      button: btn

    })

    wx.request({

      url: app.globalData.server + '/stream/command',

      method: 'POST',

      header: {

        'content-type': 'application/x-www-form-urlencoded',

        cookie: app.globalData.cookie

      },

      data: {

        DHT: true,

        CAM: this.data.alive,

        MODE: this.data.modeItems[this.data.modeIdx],

        FRAME: this.data.frameItems[this.data.frameIdx],

      }

    })

    switch (this.data.frameIdx) {

      case '0': {

        this.data.lmt = 10

        break

      }

      case '1': {

        this.data.lmt = 3

        break

      }

      case '2': {

        this.data.lmt = 2

        break

      }

      default: {

        this.data.lmt = 10

      }

    }

  },

//download a frame from server

  imgDownload () {

    if (!this.data.imgBuf.isFull()) {

      var page = this

      wx.downloadFile({

        url: app.globalData.server + '/stream?' + Date.now(),

        header: {

          cookie: app.globalData.cookie

        },

        success: (res) => {

          if (res.statusCode === 200) {

            page.data.imgBuf.push(res.tempFilePath)

          }

        },

        fail: res => {

          console.error(res.errMsg)

        }

        timeout: page.data.refreshPeriod \* page.data.imgBufSize

      })

    }

  },

  getSensors () {

    wx.request({

      url: app.globalData.server + '/stream/sensors',

      header: {

        cookie: app.globalData.cookie

      },

      success: (res) => {

        console.log(res)

        this.setData({

          temp: res.data.temperature,

          humid: res.data.humidity,

        })

      },

      fail: (res) => {

        console.error(res)

      }

    })

  },

//load current page on start up

  loadPage () {

    this.getSensors()

    this.data.imgBuf = new ImgQueue(this.data.imgBufSize \* 2)

    for (var i = 0; i < this.data.imgBufSize; ++i) {

      this.imgDownload()

    }

    var page = this

    this.data.imgInter = setInterval( () => {

      if (page.data.cnt++ >= page.data.lmt) {

        page.data.cnt = 0

        page.setData({

          datetime: formatTime(new Date())

        })

        if (this.data.alive) {

          page.imgDownload()

          page.setData({

            imgSrc: page.data.imgBuf.pop()

          })

        }

      }

    }, this.data.refreshPeriod)

  }

})