華東師範大學
East China Normal University

OpenHarmony 嵌入式系统原理与应用

智慧农业物联网系统

课　　程：OpenHarmony 嵌入式系统原理与应用

姓　　名：戴斯哲

学　　号：51275904073

学　　院：通信与电子工程学院

专　　业：通信工程

目录

1. Docker 容器安装

1.1 依赖包安装

首先安装 apt 依赖包，用于通过 HTTPS 来获取仓库。命令如下：

```
sudo apt-get install \
apt-transport-https \
ca-certificates \
curl \
gnupg-agent \
software-properties-common
```

- **apt-transport-https:** 这个包允许 apt 使用 HTTPS 协议来访问软件源和仓库。默认情况下，apt 只能通过 HTTP 访问仓库，而安装这个包后，apt 就能够通过 HTTPS 安全地连接到软件源。
- **ca-certificates:** 这个包包含了各种根证书，它们用于验证通过 HTTPS 连接的服务器的身份。当通过 HTTPS 连接到一个仓库时，系统会用这些证书来验证服务器的公钥，以确保你连接的是一个可信的源。
- **curl:** 这是一个用于从命令行进行数据传输的工具，支持多种协议，包括 HTTP 和 HTTPS。它通常用于下载仓库的公钥、软件包等文件。
- **gnupg-agent:** 这个包提供了用于管理和存储 GPG 密钥的代理。GPG 密钥通常用于签署软件包，以确保下载的包是由可信的发布者提供的。此工具用于支持与密钥相关的操作，如验证仓库签名。
- **software-properties-common:** 包含了一些常用的工具来管理 APT 软件源，例如添加新的仓库、管理 PPA（个人包存档）等。它使得添加和管理外部软件源变得更加方便。

1.2 添加 Docker 官方 GPG 密钥

bash 输入如下命令：

curl -fsSL https://mirrors.ustc.edu.cn/docker-ce/linux/ubuntu/gpg | sudo apt-key add –

使用上小节下载的 curl 工具包，从网络上获取数据。-fsSL 是下载选项，**-f** 表示如果下载失败（例如 404 错误），curl 会返回一个错误，而不是继续执行；**-s** 表示表示“silent”，即不输出任何进度信息、错误信息等，只输出请求的内容；**-S** 表示和 -s 一起使用时，表示当发生错误时会显示错误信息，便于调试；**-L** 表示表示在遇到重定向时，curl 会自动跟随重定向。

https://mirrors.ustc.edu.cn/docker-ce/linux/ubuntu/gpg 是 Docker 官方仓库的 GPG 公钥文件的地址。这个公钥用于验证通过 Docker 仓库下载的软件包的完整性和真实性。使用这个公钥后，apt 可以确认下载的软件包是否来自合法的 Docker 仓库，而不是篡改过的版本。

1.3 设置稳定仓库版本

```
sudo add-apt-repository \
"deb [arch=amd64] https://mirrors.ustc.edu.cn/docker-ce/linux/ubuntu/ \
$(lsb_release -cs) \
stable"
```

这个命令的整体作用是从 mirrors.ustc.edu.cn 下载 Docker 仓库的 GPG 公钥，并将其添加到系统的 APT 密钥管理中。这样，系统就能够验证 Docker 软件包的签名，以确保它们是安全和未被篡改的。

1.4 安装最新版本 Docker Engine-Community 与 Containerd

- **docker-ce** 是 Docker Community Edition（社区版）的核心软件包，它包含了 Docker 引擎（Docker Daemon）。Docker 引擎是一个后台服务，它负责容器的创建、运行、停止和管理。
- **docker-ce-cli** 是 Docker 的命令行工具（CLI），用于与 Docker 引擎进行交互。通过这个命令行工具，你可以使用 docker 命令来管理容器（如 docker run、docker ps、docker build 等）。
- **containerd** 是一个高性能的容器运行时，它负责管理容器的生命周期，包括镜像的拉取、容器的创建和运行等。containerd.io 是 Docker 使用的容器运行时之一，它通常作为一个独立的包安装。

使用如下命令安装：

sudo apt-get install docker-ce docker-ce-cli containerd.io

安装完成之后使用如下命令为普通用户添加 docker 操作权限：

sudo usermod -aG docker $USER

1.5 修改管理模式并添加国内镜像源代

使用如下命令打开文件：

sudo vim /etc/docker/daemon.json

添加以下配置

native.cgroupdriver=systemd：指定 Docker 使用 systemd 来管理容器的

cgroup（控制组）。systemd 是现代 Linux 系统的初始化系统，它负责管理系统的进程、资源和服务。使用 systemd 作为 cgroup 驱动程序可以提供更好的兼容性，特别是在使用 systemd 管理系统资源时（例如在使用 Docker 的同时也使用 systemd 管理其他服务）。

后续代码就是配置中的各个镜像源解释

退出 vim 界面，重启 Ubuntu 操作系统，让修改生效。

1.6 安装 docker-compose 工具

输入如下命令：

sudo curl -L https://gitee.com/meatball-org-cn/compose/releases/download/v2.24.6/docker-compose-linux-x86_64 /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

https://gitee.com/meatball-org-cn/compose/releases/download/v2.24.6/docker-compose-linux-x86_64 是 Docker Compose 工具的下载链接，指向 Gitee 的镜像。

然后更改下载文件权限，这一步是必要的，因为下载的 Docker Compose 二进制文件默认没有执行权限，需要通过 chmod 命令赋予执行权限。

2. 数据库

2.1 建立数据库空间

Bash 输入命令：

mkdir /home/mongodb-persistence

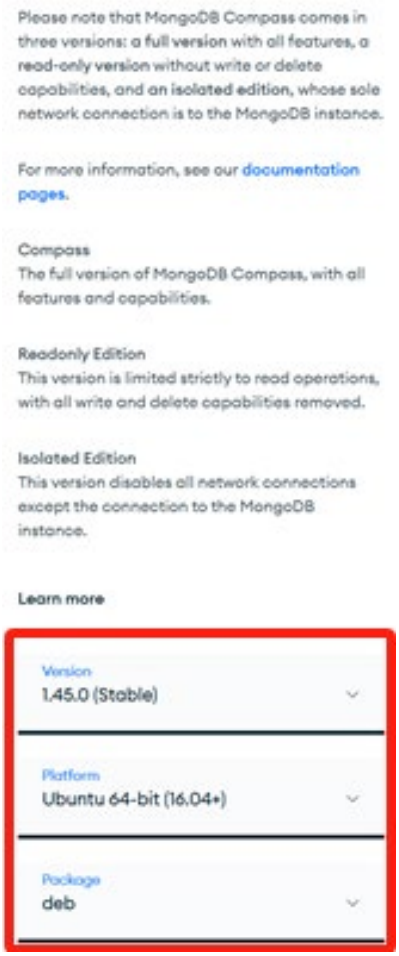
sudo chmod 777 -R /home/mongodb-persistence

这里的目录是 /home/mongodb-persistence，它将用于存储 MongoDB 的数据。通常，MongoDB 会将数据库的数据文件存储在这个目录中，确保数据能够在 MongoDB 容器重启或主机重启后保留。

如果此处不更改权限，在后续 mongodb-compass 中连接本机 27017 的端口号则会出现问题。

2.2 安装 mongodb-compass

下载并安装 mongodb-compass：https://www.mongodb.com/try/download/compass

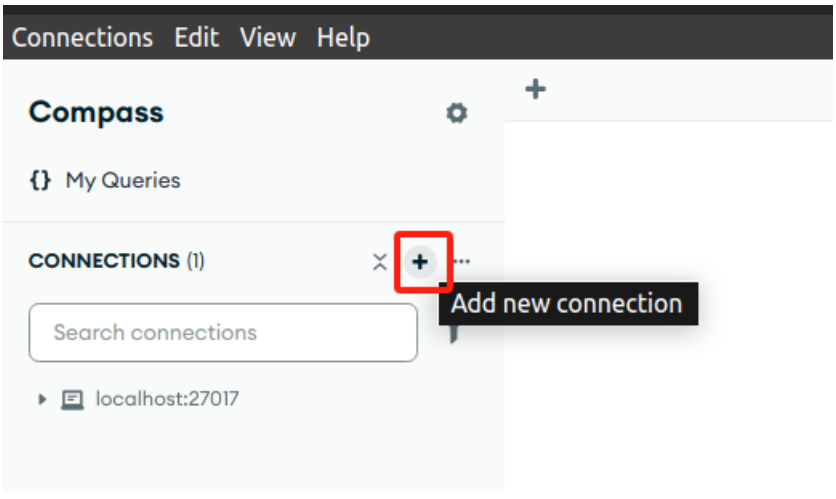


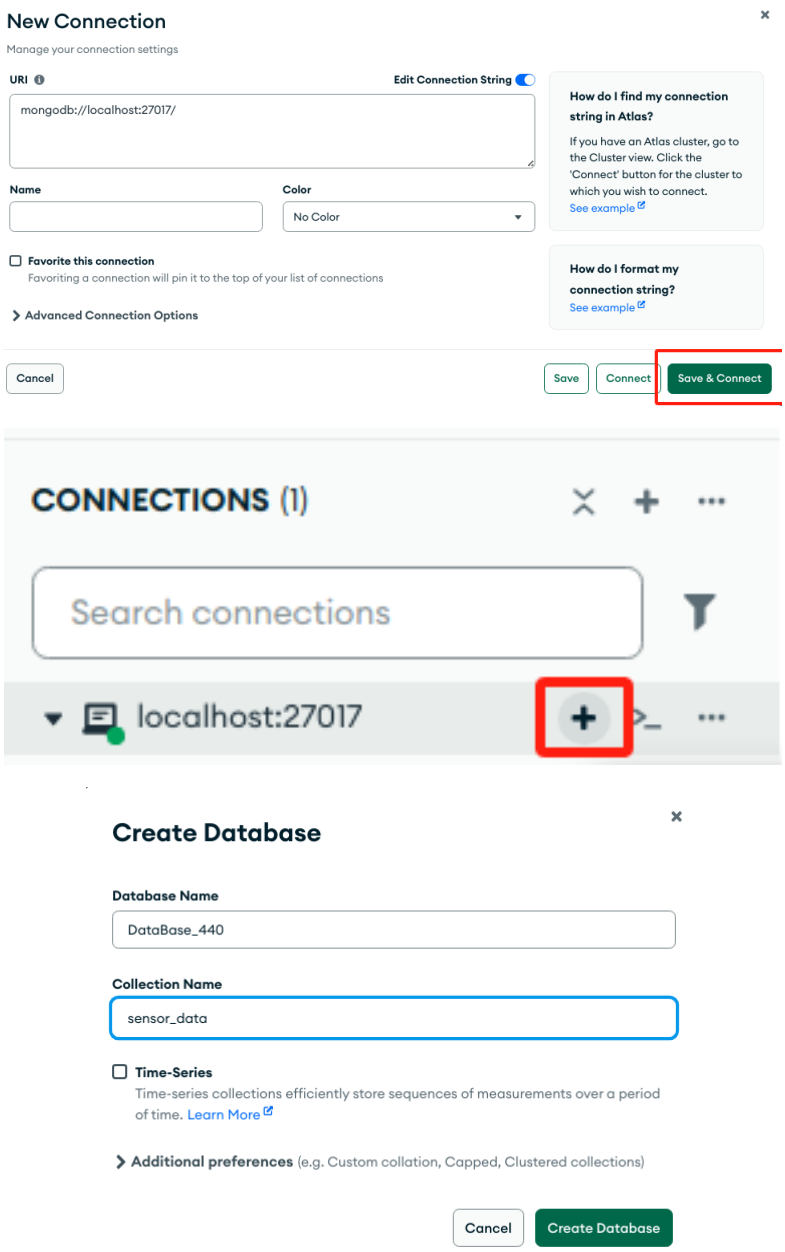
2.3 创建数据库表格

将 demo 压缩包解压并切换目录至 demo 下，输入下述命令运行容器：

docker-compose up -d

容器运行成功之后输入 mongodb-compass，打开数据库预览软件，点击创建连接，并创建数据库以及表格。





3. 数据上传搭建

MQ 队列传输协议（MQTT, Message Queuing Telemetry Transport）是一种轻量级的消息传输协议，广泛应用于物联网（IoT）设备之间的通信。MQTT 连接模型涉及到多个组件和交互方式，采用客户端-服务器（发布/订阅）模型，主要由客户端、代理（Broker）和主题（Topic）三个核心元素构成：

客户端是使用 MQTT 协议的设备或应用程序，可以是物联网设备、传感器、手机应用、Web 客户端等，可以充当 发布者（Publisher）或 订阅者（Subscriber），有时也可以是两者的组合。

代理是一个中间服务器，负责接收来自客户端的消息并根据订阅的主题将消息转发到相应的客户端，是 MQTT 系统中的核心组件，处理客户端连接、认证、

消息路由、主题订阅和发布、持久化等工作。

主题是 MQTT 消息的标识符，用于指定消息的类型或分类。客户端通过订阅特定的主题来接收与该主题相关的消息。

3.1 MQTT 消息传递过程

- 连接（Connect）：

客户端通过 MQTT CONNECT 请求连接到代理。连接时，客户端可以提供客户端标识符、用户名、密码、清理会话（是否清理先前的会话）、遗嘱消息（Last Will and Testament）等信息。代理收到连接请求后，会验证客户端身份并返回 CONNACK 响应。如果验证通过，连接成功，客户端可以开始发送或接收消息。

- 订阅（Subscribe）：

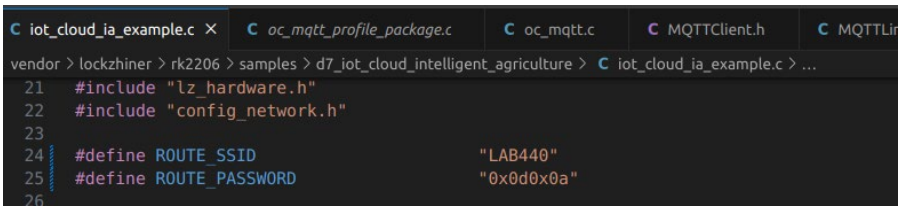
客户端可以通过 SUBSCRIBE 请求订阅一个或多个主题。订阅之后，客户端会接收到与该主题匹配的所有消息。订阅通常伴随 QoS（服务质量等级）设置，决定了消息传输的可靠性：

- 发布（Publish）：

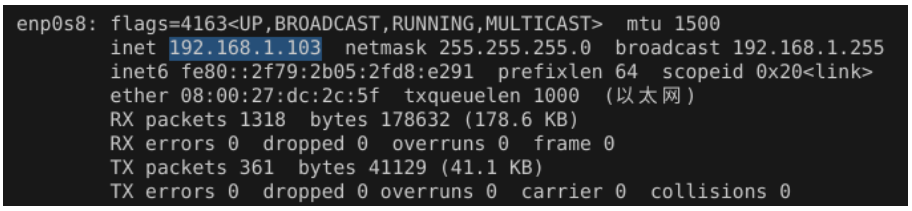
客户端通过 **PUBLISH** 请求将消息发布到特定的主题上。代理收到发布的消息后，根据订阅关系将消息转发给订阅该主题的客户端。消息也可以带有 **QoS** 设置，决定了消息如何被传递给订阅者。

3.2 开发板连接服务器

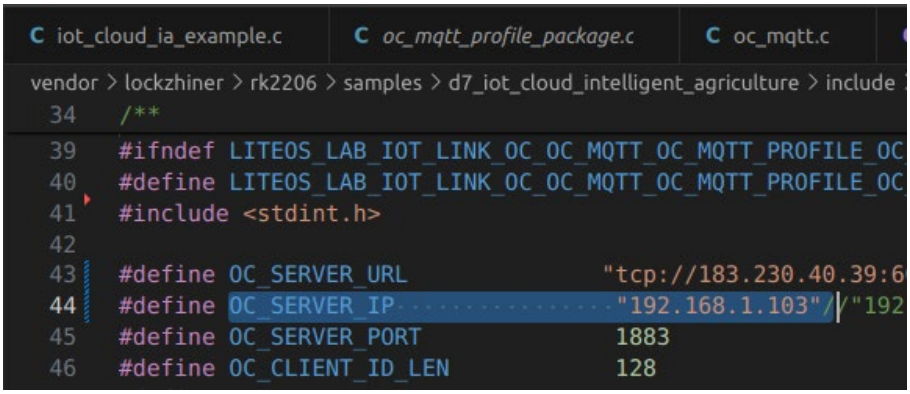
打开工程目录，切换到 d7 智慧农业目录下，首先更改连接的路由以及路由密码：



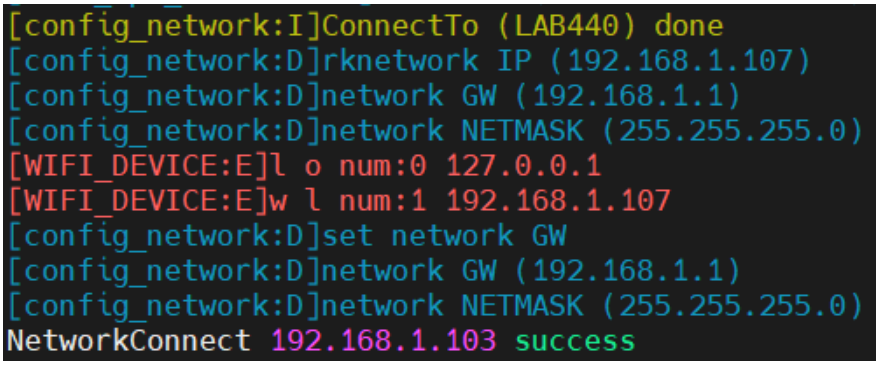
查看虚拟机 ip，即 broker 的 ip：



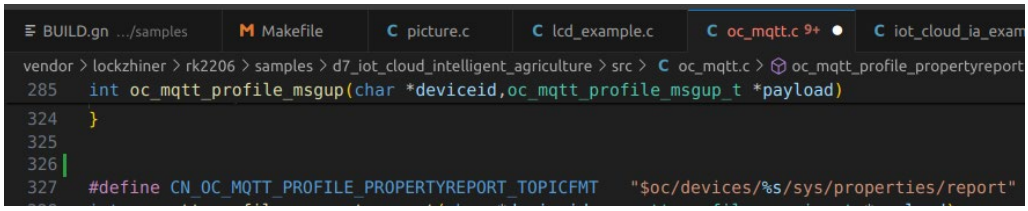
更改开发板需要连接的 MQTT 服务器 ip 地址为上述 broker 的 ip：



上电，确保开发板能够连接到虚拟机，通过串口查看信息如下，则表示连接成功：



查看开发板发布的主题如下：



其中%s 在后续会被替换成设备的 id 号，在确定了发布消息的主题之后，在 demo 目录下新建 sensor_collection.py 文件，输入如下代码，其中使用 Python 的 Paho MQTT 库创建一个 MQTT 客户端并连接到代理。

首先，通过 mqtt.Client() 创建一个客户端实例。接着，为客户端设置回调函数：on_connect 用于处理连接成功后的操作，如订阅主题，on_message 用于处理接收到的消息。然后，通过 client.connect("127.0.0.1", 1883, 60) 连接到本地代理，指定代理的 IP 地址、端口和心跳时间。最后，使用 client.loop_forever() 启动一个无限循环，保持与代理的连接并持续处理消息。当连接成功时，客户端订阅 test/topic 主题，并在接收到该主题的消息时，调用 on_message 回调函数进行处理。

```
index.html    sensor_collection.py
home > dai > work > openharmony_web_demo > sensor_collection.py > ..
 23    def on_message(client, userdata, msg):
110            sensor_db3.update_one(myquery, newvalue)
111
112
113
114    client = mqtt.Client()
115    client.on_connect = on_connect
116    client.on_message = on_message
117    client.connect("127.0.0.1", 1883, 60)
118    client.loop_forever()
119
```

on_connect 与 on_message 函数如下，其中 on_connect 函数在连接成功后会订阅上述查询到的开发板发布消息的主题，随后只要 broker 接受到消息，便将消息打印出来，以便查看格式方便后续数据解析。

```
index.html    sensor_collection.py
home > dai > work > openharmony_web_demo > sensor_collection.py > ...
15
16
17  def on_connect(client, userdata, flags, rc):
18      print("Connected with result code "+str(rc))
19      client.subscribe("$oc/devices/6758ffd5bab900244b0e32ad_1111/sys/properties/report")
20
21
22  def on_message(client, userdata, msg):
23
24      if msg.topic == "$oc/devices/6758ffd5bab900244b0e32ad_1111/sys/properties/report":
25          sensor_data = json.loads(msg.payload.decode('utf-8'))
26          print("设备1",sensor_data)
```

通过命令运行 py 文件，得到下述数据打印结果，接下来就对相应的 json 数据进行解析。

python3 sensor_collection.py

```
dai@dai-VirtualBox:~/work/openharmony_web_demo$ python3 sensor_collection.py
sensor_collection.py:114: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
Connected with result code 0
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 30, '亮度': 961, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 30, '亮度': 964, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 30, '亮度': 970, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 29, '亮度': 960, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 28, '亮度': 974, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 29, '亮度': 971, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 28, '亮度': 974, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 29, '亮度': 965, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 27, '亮度': 952, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 27, '亮度': 962, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 27, '亮度': 965, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 27, '亮度': 959, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 27, '亮度': 965, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 26, '亮度': 970, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 26, '亮度': 61, '紫光灯状态': '关', '电机状态': '关'}}]}
设备1 {'services': [{'service_id': '智慧农业', 'properties': {'温度': 26, '湿度': 28, '亮度': 1965, '紫光灯状态': '关', '电机状态': '关'}}]}
```

下述代码从 sensor_data 中提取传感器的相关数据，并将其更新到数据库中。首先从 sensor_data 字典中提取温度、光照、湿度、紫光灯状态和电机状态等传感器数据。然后，代码依次使用 update_one 方法更新数据库中的相应记录。

- 从 sensor_data 中通过 get()方法获取温度、亮度、湿度、紫光灯状态和电机状态的值。
- 对每个传感器的值，构建查询条件 myquery，该查询条件用于在数据库中查找具有特定 sensor_id 的记录。
- 构建新的更新值 newvalue，通过 $set 操作将新的传感器值赋给 sensor_value 字段。
- 使用 update_one()方法更新数据库中符合查询条件的记录，将新的传感器值存入数据库。

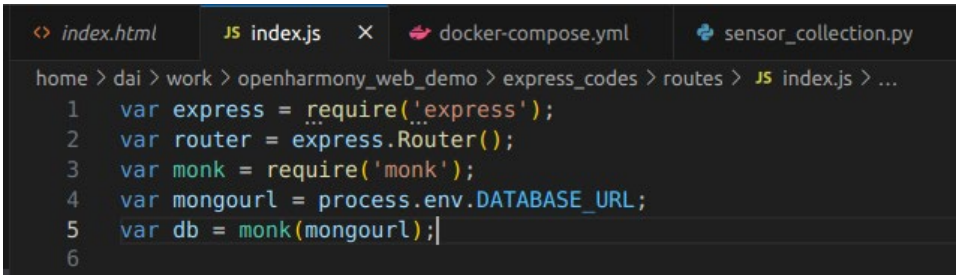
```
temperature = sensor_data['services'][0]['properties'].get('温度')
light = sensor_data['services'][0]['properties'].get('亮度')
hum = sensor_data['services'][0]['properties'].get('湿度')
UV = sensor_data['services'][0]['properties'].get('紫光灯状态')
motor = sensor_data['services'][0]['properties'].get('电机状态')
# 更新温度
myquery = {"sensor_id":"temperature"}
newvalue = {"$set":{"sensor_value": temperature}}
sensors_db.update_one(myquery, newvalue)
#更新光照
myquery = {"sensor_id":"light"}
newvalue = {"$set":{"sensor_value": light}}
sensors_db.update_one(myquery, newvalue)
# 更新湿度
myquery = {"sensor_id":"hum"}
newvalue = {"$set":{"sensor_value": hum}}
sensors_db.update_one(myquery, newvalue)
# 更新紫光灯状态
myquery = {"sensor_id":"紫光灯状态"}
newvalue = {"$set":{"sensor_value": UV}}
sensors_db.update_one(myquery, newvalue)
# 更新电机状态
myquery = {"sensor_id":"电机状态"}
newvalue = {"$set":{"sensor_value": motor}}
sensors_db.update_one(myquery, newvalue)
```

在 index.html 文件中，通过 jQuery 的 **$.get()** 方法发送一个 HTTP GET 请求，访问 /get_sensors 路径，向服务器请求开发板的传感器数据。

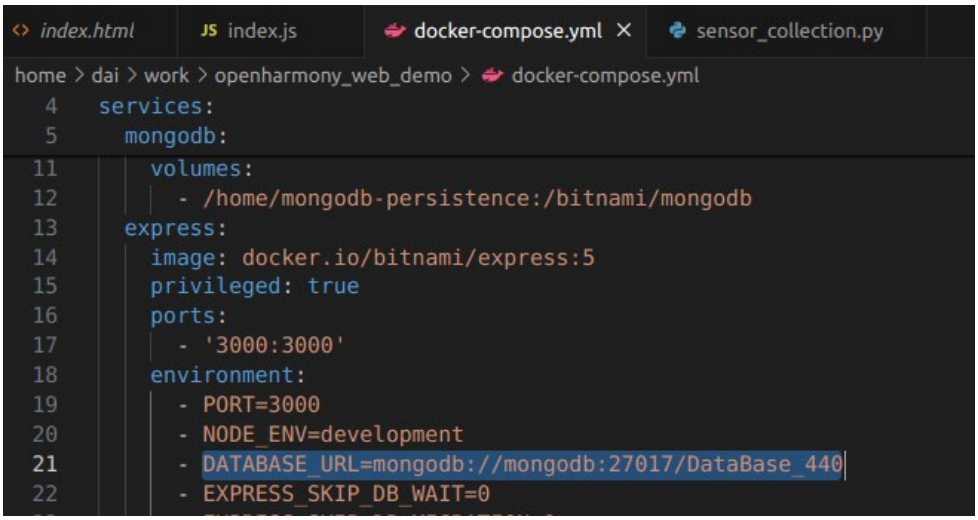
```
setInterval(function() {
greenhouses.forEach(g => {
    // 为每个大棚单独发请求
    $.get(`/get_sensors${g.id}`, function(data, status) {
        // 更新该大棚的传感器数据
        g.temp = data[0].sensor_value;
        g.light = data[1].sensor_value;
        g.humidity = data[2].sensor_value;
        g.uvStatus = data[3].sensor_value;
        g.motorStatus = data[4].sensor_value;
        fanStates_string[g.id] = g.motorStatus
        shadeStates_string[g.id] = g.uvStatus;


        renderGreenhouseList();
    });
});
```

在 js 文件中，使用 **Express.js** 和 **Monk** 库建立了一个与 MongoDB 的连接。首先通过 require('express') 引入 Express 框架，并创建一个路由实例 router。

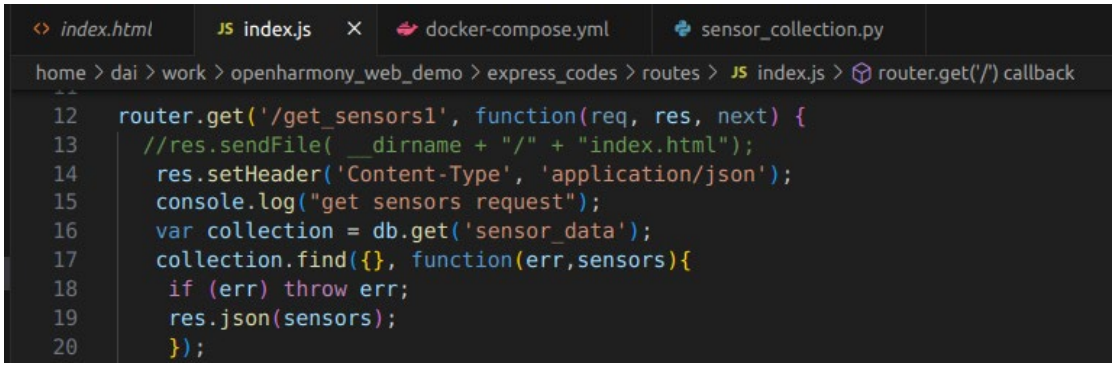
然后，使用 monk 库连接到 MongoDB，连接字符串通过环境变量 DATABASE_URL 获取，创建的数据库连接对象赋值给 db，可以通过该对象执行数据库操作。



环境变量 DATABASE_URL 定义在 demo 根目录下的 docker-compose.yml 中，在此处指向了之前创建的数据库 DataBase_440。



get 路由 /get_sensors1 在下述代码定义，当客户端访问该路径时，服务器会查询数据库中名为 sensor_data 的集合，获取所有传感器数据，并将其作为 JSON 格式的响应返回给客户端。在查询过程中，如果发生错误，会抛出异常。响应头被设置为 Content-Type: application/json，确保客户端知道接收到的数据是 JSON 格式。

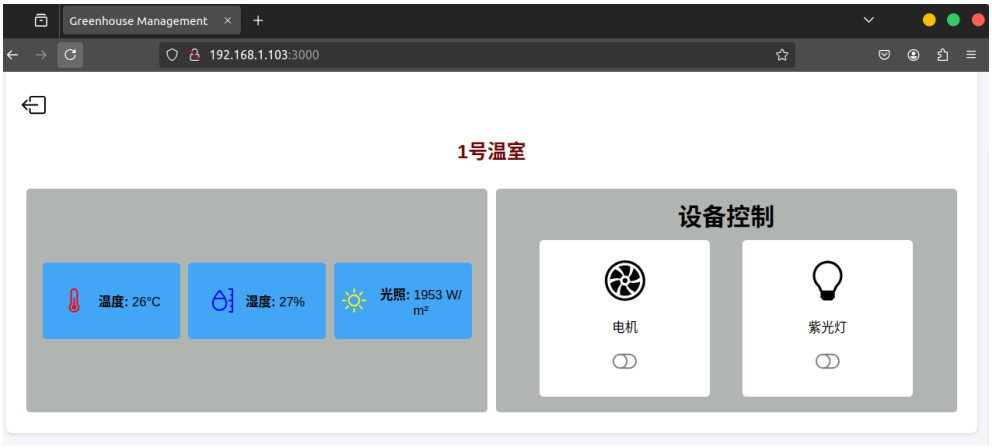


下述代码通过遍历 greenhouses 数组，为每个开发板创建一个 div 元素，显示大棚的 ID、温度、湿度、光照值，并为每个开发板添加一个按钮，点击后调用 showDetails 函数进行设备管理。最终将生成的内容插入到页面上的 greenhouselist 元素中。

```
function renderGreenhouseList() {
    const list = document.getElementById('greenhouselist');
    list.innerHTML = ''; // 清空列表内容
    greenhouses.forEach(g => {
        const item = document.createElement('div');
        item.className = 'greenhouse-item';
        item.innerHTML = `
            <div>${g.id}号大棚</div>
            <div>温度: ${g.temp}°C</div>
            <div>湿度: ${g.humidity}%</div>
            <div>光照: ${g.light} W/m²</div>
            <button onclick="showDetails(${g.id})">设备管理</button>
        `;
        list.appendChild(item);
```

拿到开发板传感器数据后，网页端则会实时显示数据，显示效果如下：

也可以单独点击详情页面查看：



4. 命令下发搭建

在详情页面中，编写了电机与紫光灯的命令控制按钮。当用户点击按钮时，更新相应的状态（开/关），并通过 updateToggleIcon 函数更新按钮图标。同时，调用 toggleMotor 和 toggleUV 函数，将风扇和紫光灯的状态发送到服务器进行更新。toggleMotor 和 toggleUV 函数通过 $.post 方法向服务器发送设备状态，服务器响应后打印相应的消息。

```
97              fanToggle.onclick = () => {
98                  fanStates[id] = !fanStates[id];
99                  fanStates_string[id] = fanStates[id]=== true ? "开" : "关";
100                 updateToggleIcon(fanToggle, fanStates[id]);
101                 toggleMotor(id,fanStates[id]);
102             };
103
104
105             shadeToggle.onclick = () => {
106                 shadeStates[id] = !shadeStates[id];
107                 shadeStates_string[id] = shadeStates[id] === true ? "开" : "关";
108                 updateToggleIcon(shadeToggle, shadeStates[id]);
109                 toggleUV(id,shadeStates[id]);
110
111             };
```

```
131     // 切换电机状态
132     function toggleMotor(id,motorstatus) {
133         // 发送状态到服务器
134         temp_status = motorstatus
135         $.post(`/update_status${id}`, { device: 'motor', status: temp_status }, function(response) {
136             console.log(response.message);
137         });
138     }
139
140     // 切换紫光灯状态
141     function toggleUV(id,UVstatus) {
142         // 发送状态到服务器
143         temp_status = UVstatus
144         $.post(`/update_status${id}`, { device: 'uv_light', status: temp_status }, function(response) {
145             console.log(response.message);
146         });
147     }
148
```

js 中使用了几个关键函数来完成数据库操作和响应处理。首先，router.post('/update_status1', ...) 用于处理前端发送的 POST 请求，并从 req.body 中提取设备名称和状态。然后，db.get('sensor_data') 获取数据库的 sensor_data 集合。接着，update()用于根据设备名称更新数据库中的状态字段。myquery 变量确定要更新的文档条件，newvalue 包含新的状态值。更新操作使用 update 函数执行，并传入 {single: true} 参数表示更新单个文档。若更新成功，findOne() 用于查询更新后的文档并返回结果，最终通过 res.json() 返回成功消息。若在数据库操作中出现错误，则使用 res.status(500).json() 返回错误消息。

```
// 更新设备1状态
router.post('/update_status1', function(req, res) {
  const { device, status } = req.body; // 接收前端发送的数据
  // 数据库集合
  var collection = db.get('sensor_data');
  // 根据设备名称更新状态
  let myquery = { sensor_id: device === 'motor' ? "电机状态" : "紫光灯状态" };
  let newvalue = { $set: { sensor_value: status == "true"? "开" : "关" } };
  update_result = collection.update(myquery, newvalue, {single: true},function(err, result) {
    if (err) {
        res.status(500).json({ message: '数据库更新失败' });
    } else {
        // 查询更新后的文档
            collection.findOne(myquery, function(findErr, updatedDoc) {
            if (findErr) {
                res.status(500).json({ message: '查询更新的文档失败' });
            } else {
                res.json({
                    message: `${device}设置${status}状态更新成功`,
                    result:result,
                });
            }
        });
    }
});
});
```

在 demo 文件夹根目录创建 metor_uv_publish.py 文件，用于发布特定主题的控制信息。通过华为云进行调试，追踪下发命令的格式：

下发消息

状态	缓存
消息 ID	2dc34d37-f217-474d-b610-fabbe94d3aeb
消息名称	
消息内容	{"service_id":"智慧农业","command_name":"紫光灯控制","paras":{"Light":"ON"}}
内容编码格式	非base64编码

在 metor_uv_publish.py 中，定义了一个 query_and_publish 函数，用于查询数据库中的电机和紫光灯状态，并根据状态变化向物联网平台发布命令。首先，函数从数据库 sensors_db 中获取电机和紫光灯的当前状态，并分别存储在 motor_status 和 uv_status 变量中。接着，它检查当前状态与上次状态（存储在 previous_motor_status 和 previous_uv_status 中）是否发生变化。如果发生变化，就生成相应的命令消息并通过 MQTT 客户端 client.publish() 发布该消息到物联网平台。对于电机状态，如果电机开启则发布 "电机控制" 命令并设置 Motor: "ON"，否则发布 "电机控制" 命令并设置 Motor: "OFF"。同理，紫光灯状态变化时也会发布相应的控制命令。每当状态变化时，函数会打印电机或紫光灯的开关状态，并确保在每次查询时更新上次的状态变量。同时结合定时器对象，将 query_and_publish 传入定时器中，以一秒一次的频率运行该函数。

```python
def query_and_publish():
    global previous_motor_status, previous_uv_status,previous_uv_status2,previous_motor_status2,previous_motor_status3,previous_uv_status3

    motor_status = sensors_db.find_one({"sensor_id": "电机状态"})["sensor_value"]
    uv_status = sensors_db.find_one({"sensor_id": "紫光灯状态"})["sensor_value"]

    # 检查电机状态变化
    if motor_status != previous_motor_status:
        previous_motor_status = motor_status
        if motor_status == "开":
            message = {
                "service_id": "智慧农业",
                "command_name": "电机控制",
                "paras": {"Motor": "ON"}
            }
            print("电机开启")
            client.publish("$oc/devices/6758ffd5bab900244b0e32ad_1111/sys/commands", json.dumps(message))
        else:
            message = {
                "service_id": "智慧农业",
                "command_name": "电机控制",
                "paras": {"Motor": "OFF"}
            }
            print("电机关闭")
            client.publish("$oc/devices/6758ffd5bab900244b0e32ad_1111/sys/commands", json.dumps(message))

    # 检查紫光灯状态变化
    if uv_status != previous_uv_status:
        previous_uv_status = uv_status
        if uv_status == "开":
            message = {
                "service_id": "智慧农业",
                "command_name": "紫光灯控制",
                "paras": {"Light": "ON"}
            }
            print("紫光灯开启")
            client.publish("$oc/devices/6758ffd5bab900244b0e32ad_1111/sys/commands", json.dumps(message))
        else:
            message = {
                "service_id": "智慧农业",
                "command_name": "紫光灯控制",
                "paras": {"Light": "OFF"}
```

```python
165     threading.Timer(1, query_and_publish).start()
```

在开发板端，则需要在 MQTT 协议初始化的时候订阅相关主题：

```c
// 订阅新的 topic
const char *receive_topic = topic_make(OC_MQTT_COMMAND_RECEIVE, oc_info.username, NULL);
int qos = 0;  // 设置 QoS
rc = MQTTSubscribe(&mq_client, receive_topic, qos, msgHandler);
if (0 == rc) {
    printf("Successfully subscribed to topic: %s\n", receive_topic);
} else {
    printf("Failed to subscribe to topic: %s, rc: %d\n", receive_topic, rc);
}

return rc;
```

完成相关工作后，将代码烧录到开发板当中，上电运行，网页端控制开发板的紫光灯灯亮：

在开发板的串口信息中可以看到灯亮信息，同时紫光灯点亮。

```
NetworkConnect 192.168.1.103 success
xxxxxxxxxxxxxxxxxxxxxxxx enter msghandler xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
test:0 0x100828d1 0x380012c0
MQTTConnect success! rt:0
Successfully subscribed to topic: $oc/devices/6758ffd5bab900244b0e32ad_1111/sys/
commands
oc package(V) initialize success.enter netcmd callback!!
Light On
enter netcmd callback!!
```

5. 多节点与本地显示搭建

按照上述搭建方法，对本组的四块开发板做板卡 id 的区分，并在数据上传与命令下发的 py 文件里做出相应的修改，前端文件也做出相应的修改，在原有的基础上多复制几份相同的显示代码即可。

最终系统成品如下，本地显示基于 lcd 的 demo 修改，显示了相应的传感器数据与外设状态，云端可以实时显示传感器数据与外设状态，同时也能下发命令控制每个开发板的外设，其中云端显示争对不同设备屏幕做了适配。

图 1.本地显示

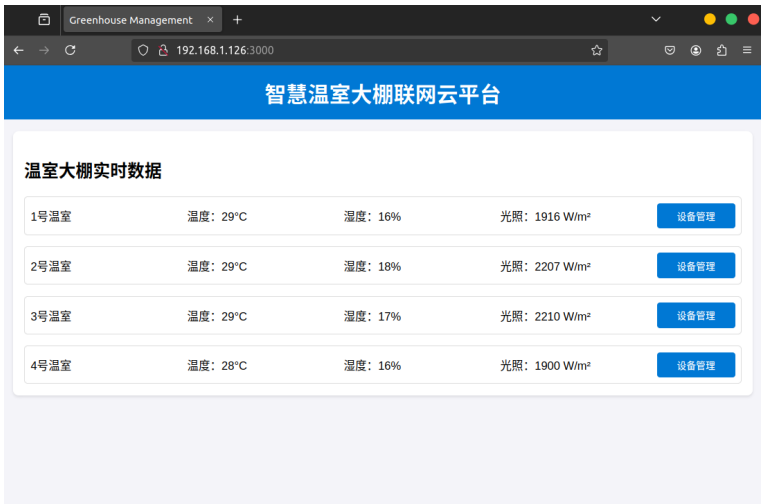


图 2-3 网页端显示

图 4-5 手机端显示