```
/* 本文档将按照以下顺序记录项目源代码:
main.cpp->oledfont.h->OLED.h->OLED.cpp->Alert.h->Alert.cpp->WebServer.h->WeatherNow.h->WeatherNow.cpp->Tool.h->Tool.cp
p-> Hash.h->Hash.cpp->Universal.h->rivest_cipher_4.hpp->make_ptr.hpp->fourier_transform.hpp->tree.hpp
*/

// main.cpp-------------------------------------------------------------------------------------------------------
// [RMSHE Infinty] 嵌入式智能可燃气体报警系统主控软件 GasSensorGen3_Program V2023.04.15 Powered by 马山河(RMSHE)
// MCU: ESP8266; MODULE: ESP12F;
#include <Hash.h>
#include <PubSubClient.h>
#include <WeatherNow.h>
#include <stack>
#include "Alert.h"
#include "OLED.h"
#include "Tool.h"
#include "Universal.h"
#include "WebServer.h"
#include "fourier_transform.hpp"
#include "tree.hpp"

ALERT alert;
OLED oled;
HTTPClient http;
WiFiClient client;
TOOL tool;
Ticker TimeRefresh_ticker;
Ticker System_time;
Ticker Desktop_ticker;
Ticker CMDControlPanel_ticker;
Ticker UpdateWeather;
Ticker WIFI_Test;       // 用于检测 WIFI 是否连接;
WeatherNow weatherNow;  // 建立 WeatherNow 对象用于获取天气信息

PubSubClient mqtt_client(client);     // 建立 MQTT 客户端对象;
ESP8266WebServer server(ServerPort);  // 建立网络服务器对象，该对象用于响应 HTTP 请求。监听端口（80）

//[System Mode&Status(系统模式和状态)]
bool Charging_State = false;    // 充电状态(0:没在充电; 1:正在充电);
bool WIFI_State = false;        // WIFI 状态(0:断网; 1:联网);
bool Developer_Mode = false;    // 开发者模式(0:常规运行; 1:进入开发者模式);
bool allowResponse = true;      // true:允许服务器对客户端进行响应;
bool allowDownloadMode = true;  // true:允许进入下载模式;
bool freezeMode = false;  //[浅度休眠模式-freeze]，冻结 I/O 设备，关闭外设，ESP-12F 进入 Modem-sleep 模式，程序上只运行
CMDControlPanel 网络服务，其他服务冻结;
bool diskMode = false;  //[深度休眠模式-disk] 运行状态(GPIO_Status，系统模式和状态，文本框信息)数据存到 Flash(醒来时恢复状
态)，然后 ESP12F 进入深度睡眠;

bool CMDCP_State = false;  // CMDCP 是否被打开(trrue:表示被打开);

class FlashFileSystem {
   private:
    FSInfo Flash_info;
    Dir FileDirectory;

    stack<String> WorkingDirectoryStack;  // 工作路径的栈(用来储存历史工作路径，一遍返回上一个路径);
    String WorkingDirectory = "/";        // 当前工作路径;

    // 递归删除文件夹;
    void deleteFolder(String path) {
        // 打开待删除目录;
        Dir FileDirectory = LittleFS.openDir(path);

        // 遍历目录;
        while (FileDirectory.next()) {
```

```
            // 获取当前项文件的名称;
            String entryPath = path + "/" + FileDirectory.fileName();

            // 检查当前项是否为目录;
            if (LittleFS.exists(entryPath)) {
                deleteFolder(entryPath);  // 如果是目录则循环调用自身函数递归删除该目录;
            } else {
                LittleFS.remove(entryPath);  // 如果不是目录则删除该文件;
            }
        }

        LittleFS.rmdir(path);  // 在删除该目录下的所有文件和目录后删除这个父目录;
    }

   public:
    // 获取 Flash 信息;
    String getFlash_info() {
        LittleFS.begin();           // 启动 LittleFS
        LittleFS.info(Flash_info);  // 获取闪存文件系统信息

        /*
        // 可用空间总和（单位：字节）
        Serial.print("totalBytes: ");
        Serial.print(Flash_info.totalBytes);
        Serial.println(" Bytes");

        // 已用空间（单位：字节）
        Serial.print("usedBytes: ");
        Serial.print(Flash_info.usedBytes);
        Serial.println(" Bytes");

        // 使用占比;
        Serial.print("Proportion: ");
        Serial.print(Proportion);
        Serial.println(" %");

        // 最大文件名字符限制（含路径和'\0'）
        Serial.print("maxPathLength: ");
        Serial.println(Flash_info.maxPathLength);

        // 最多允许打开文件数量
        Serial.print("maxOpenFiles: ");
        Serial.println(Flash_info.maxOpenFiles);

        // 存储块大小
        Serial.print("blockSize: ");
        Serial.println(Flash_info.blockSize);

        // 存储页大小
        Serial.print("pageSize: ");
        Serial.println(Flash_info.pageSize);
        */

        // 计算空间使用占比;
        unsigned char Proportion =
            static_cast<unsigned char>(round((static_cast<float>(Flash_info.usedBytes) /
static_cast<float>(Flash_info.totalBytes)) * 100));

        // 计算使用占比的文本进度条;
        String ProportionBar = "[          ] ";
        for (unsigned char i = 1; i < static_cast<unsigned char>(0.1 * Proportion); ++i) ProportionBar[i] = '=';

        /*显示到 OLED 屏幕上*/
```

```
        return "Flash Info (Bytes)\nTotal:" + String(Flash_info.totalBytes) + "\nUsed: " + String(Flash_info.usedBytes)
+ "\n" + ProportionBar +
               String(Proportion) + "%\nMaxPathLength:" + String(Flash_info.maxPathLength) + "\nMaxOpenFiles:" +
String(Flash_info.maxOpenFiles) +
               "\nBlockSize:" + String(Flash_info.blockSize) + "\nPageSize:" + String(Flash_info.pageSize);
    }

    // 获取当前的工作目录;
    String getWorkDirectory() { return WorkingDirectory; }

    // 返回上一个工作目录;
    void backDirectory() {
        // 如果栈非空则弹出一个上一个工作目录将其设置为当前工作目录并返回;
        if (!WorkingDirectoryStack.empty()) {
            WorkingDirectory = WorkingDirectoryStack.top();
            WorkingDirectoryStack.pop();
        }
    }

    // 切换当前工作目录;
    void changeDirectory(String path) {
        // 工作路径压入堆栈;
        WorkingDirectoryStack.push(path);

        WorkingDirectory = path;
    }

    // 显示工作目录下之内容;
    String listDirectoryContents() {
        // 文件路径 = 工作路径 + 文件名;
        String path = WorkingDirectory;

        /*获取工作目录下的所有文件名*/

        LittleFS.begin();  // 启动闪存文件系统

        // 显示目录中文件内容以及文件大小
        FileDirectory = LittleFS.openDir(path.c_str());  // 建立“目录”对象

        String FlashlistDirectory = "";
        while (FileDirectory.next()) {  // dir.next()用于检查目录中是否还有“下一个文件”
            FlashlistDirectory += path + FileDirectory.fileName() + "\n";
        }

        return FlashlistDirectory;
    }

    // 读文件(工作目录下的文件名，或直接指定文件路径[指定文件路径后工作目录下的文件名就无效了]);
    String readFile(String fileName, String filePath = "") {
        String path;
        // 如果直接指定文件路径就优先使用文件路径(filePath 是用来方便给系统内部调用的);
        if (filePath == "")
            path = WorkingDirectory + fileName;  // 文件路径 = 工作路径 + 文件名(CMD 调用);
        else
            path = filePath;  // 优先使用(系统 API 调用);

        LittleFS.begin();  // 启动 LittleFS;

        File dataFile;
        String File_Info = "";
        // 确认闪存中是否有文件
        if (LittleFS.exists(path.c_str())) {
            File dataFile = LittleFS.open(path.c_str(), "r");  // 建立 File 对象用于从 LittleFS 中读取文件;
```

```
            // 读取文件内容并且通过串口监视器和 OLED 输出文件信息
            while (dataFile.available()) {
                File_Info += (char)dataFile.read();
            }

            dataFile.close();  // 完成文件读取后关闭文件
        } else {
            File_Info = "[FLASH FILE NOT FOUND]: " + path;
        }
        return File_Info;
    }

    // 文件追加内容，如果文件不存在则创建后追加(内容，工作目录下的文件名，或直接指定文件路径[指定文件路径后工作目录下的文件名就无效了]);
    void fileAppend(String text, String fileName, String filePath = "") {
        String path;
        // 如果直接指定文件路径就优先使用文件路径(filePath 是用来方便给系统内部调用的);
        if (filePath == "")
            path = WorkingDirectory + fileName;  // 文件路径 = 工作路径 + 文件名(CMD 调用);
        else
            path = filePath;  // 优先使用(系统 API 调用);

        LittleFS.begin();  // 启动 LittleFS;

        File dataFile;
        // 确认闪存中是否有文件
        if (LittleFS.exists(path)) {
            dataFile = LittleFS.open(path.c_str(), "a");  // 建立 File 对象用于向 LittleFS 中的 file 对象追加信息(添加);
        } else {
            dataFile = LittleFS.open(path.c_str(), "w");  // 建立 File 对象用于向 LittleFS 中的 file 对象写入信息(新建&覆盖);
        }

        dataFile.print(text);  // 向 dataFile 写入字符串信息
        dataFile.close();      // 完成文件写入后关闭文件}
    }

    // 文件覆盖内容(内容，工作目录下的文件名，或直接指定文件路径[指定文件路径后工作目录下的文件名就无效了]);
    void fileCover(String text, String fileName, String filePath = "") {
        String path;
        // 如果直接指定文件路径就优先使用文件路径(filePath 是用来方便给系统内部调用的);
        if (filePath == "")
            path = WorkingDirectory + fileName;  // 文件路径 = 工作路径 + 文件名(CMD 调用);
        else
            path = filePath;  // 优先使用(系统 API 调用);

        LittleFS.begin();  // 启动 LittleFS;

        File dataFile;
        // 确认闪存中是否有文件
        dataFile = LittleFS.open(path.c_str(), "w");  // 建立 File 对象用于向 LittleFS 中的 file 对象写入信息(新建&覆盖);

        dataFile.print(text);  // 向 dataFile 写入字符串信息
        dataFile.close();      // 完成文件写入后关闭文件
    }

    // 创建或覆盖文件(工作目录下的文件名);
    void createFile(String fileName) {
        // 文件路径 = 工作路径 + 文件名;
        String path = WorkingDirectory + fileName;

        LittleFS.begin();  // 启动 LittleFS;

        File dataFile;
        dataFile = LittleFS.open(path.c_str(), "w");  // 建立 File 对象用于向 LittleFS 中的 file 对象写入信息(新建&覆盖);
```

```
        dataFile.println("");  // 向 dataFile 写入空信息;
        dataFile.close();      // 完成文件写入后关闭文件}
    }

    // 创建目录(工作目录下的文件夹名);
    void makeDirector(String dirName) {
        // 文件路径 = 工作路径 + 文件名;
        String path = WorkingDirectory + dirName;

        LittleFS.begin();  // 启动 LittleFS;

        LittleFS.mkdir(path.c_str());  // 创建目录;
    }

    // 删除文件(工作目录下的文件名，或直接指定文件路径[指定文件路径后工作目录下的文件名就无效了]);
    bool removeFile(String fileName, String filePath = "") {
        String path;
        // 如果直接指定文件路径就优先使用文件路径(filePath 是用来方便给系统内部调用的);
        if (filePath == "")
            path = WorkingDirectory + fileName;  // 文件路径 = 工作路径 + 文件名(CMD 调用);
        else
            path = filePath;  // 优先使用(系统 API 调用);

        LittleFS.begin();  // 启动闪存文件系统

        // 从闪存中删除文件
        if (LittleFS.remove(path.c_str())) {
            return true;
        } else {
            return false;
        }
    }

    // 删除文件夹(工作目录下的文件夹名，或直接指定文件路径[指定文件路径后工作目录下的文件名就无效了]);
    void removeDirector(String dirName, String dirPath = "") {
        String path;
        // 如果直接指定文件路径就优先使用文件路径(filePath 是用来方便给系统内部调用的);
        if (dirPath == "")
            path = WorkingDirectory + dirName;  // 文件路径 = 工作路径 + 文件名(CMD 调用);
        else
            path = dirPath;  // 优先使用(系统 API 调用);

        LittleFS.begin();  // 启动闪存文件系统

        deleteFolder(path);  // 递归删除文件夹;
    }

    // 复制文件(源文件路径，目标文件路径，是否移动文件[true:复制完成后删除源文件]);
    void copyFile(String sourceFilePath, String targetFilePath, bool moveMode = false) {
        LittleFS.begin();  // 启动闪存文件系统

        File source = LittleFS.open(sourceFilePath, "r");  // 打开源文件(读);
        File target = LittleFS.open(targetFilePath, "w");  // 打开(创建)目标文件(写);

        /*
        这段代码将从源文件 source 读取的数据写入目标文件 target。
        它通过使用 source.available() 方法来检查源文件是否还有未读取的数据，并使用 target.write(source.read()) 方法将读取的数
据写入目标文件。
        这段代码的前提是：源文件 source 和目标文件 target 已经打开。
        */
        if (source && target) {
            while (source.available()) {
                target.write(source.read());
```

```
            }
        }

        source.close();
        target.close();

        if (moveMode) LittleFS.remove(sourceFilePath.c_str());  // 如果设为移动模式则删除源文件;
    }

    // 复制目录(源目录路径，目标目录路径，是否移动目录[true:复制完成后删除源目录]);
    void copyDir(String sourceDirPath, String targetDirPath, bool moveMode = false) {
        LittleFS.begin();  // 启动闪存文件系统

        if (!LittleFS.exists(sourceDirPath)) return;                          // 判断源路径是否是目录;
        if (!LittleFS.exists(targetDirPath)) LittleFS.mkdir(targetDirPath);  // 判断目标路径是否是目录(否则创建一个目录);

        Dir sourceDir = LittleFS.openDir(sourceDirPath);  // 打开源目录;

        // 读取源目录下的所有文件和子目录;
        while (sourceDir.next()) {
            // 获得源文件路径和目标文件路径;
            String sourceFilePath = sourceDirPath + "/" + sourceDir.fileName();
            String targetFilePath = targetDirPath + "/" + sourceDir.fileName();

            // 判断父目录下的内容是目录还是文件;
            if (sourceDir.isDirectory()) {
                copyDir(sourceFilePath, targetFilePath, moveMode);  // 如果是目录则循环调用自身函数递归复制所有子目录下的文
件;
            } else {
                copyFile(sourceFilePath, targetFilePath, moveMode);  // 如果是文件则直接复制文件;
            }
        }

        if (moveMode) deleteFolder(sourceDirPath);  // 如果设为移动模式则递归删除源目录;
    }

    /*
    这段代码是一个查找文件的函数。该函数接收两个参数：dirPath（目录路径）和 fileName（待查找文件名称）。该函数执行以下步骤：

    启动闪存文件系统
    打开目录 dirPath
    循环读取目录中的所有文件/目录：
    a．如果读取的是目录，递归调用该函数，并将其结果加入 foundFile 字符串。
    b．如果读取的是文件：
    i．以"."分割该文件的文件名，以得到其扩展名。
    ii．比较待查找文件名和该文件的文件名：
    如果待查找文件名为"."，说明查找所有文件，不进行筛选。
    如果待查找文件名为"*.txt"，说明查找所有扩展名为"txt"的文件，按扩展名筛选。
    如果待查找文件名为"a.txt"，说明查找文件名为"a.txt"的文件，按文件名筛选。
    返回找到的文件路径列表(存储在 foundFile 字符串中)
    注：

    LittleFS.begin()是闪存文件系统的初始化函数。
    LittleFS.openDir(dirPath)是打开目录的函数。
    dir.next()是读取下一个文件/目录的函数，如果还有下一个文件/目录，则返回 true，否则返回
    */
    // 查找指定目录下的文件(dirPath(目录路径)，fileName(待查找文件名称));
    String findFiles(String dirPath, String fileName) {
        String foundFile = "";

        // 按"."分割 fileName 待查找文件名字符串;
        vector<String> targetName = oled.strsplit(fileName, ".");

        LittleFS.begin();  // 启动闪存文件系统
```

```
        Dir dir = LittleFS.openDir(dirPath);  // 打开目录;

        while (dir.next()) {
            if (dir.isDirectory()) {
                foundFile += findFiles(dirPath + dir.fileName() + "/", fileName);
            } else {
                String foundName = dir.fileName();                              // 获取目录中文件的文件名;
                vector<String> foundName_Split = oled.strsplit(foundName, ".");  // 按"."分割查找到的文件名字符串;

                if (targetName[0] == "*" && targetName[1] == "*") {  //[fileName] = *.* : 查找所有文件;
                    // 不筛选;
                    foundFile += dirPath + foundName + "\n";

                } else if (targetName[0] == "*" && targetName[1] != "*") {  //[fileName] = *.txt : 查找所有扩展名为 txt 的
文件;
                    // 按扩展名筛选文件;
                    if (foundName_Split[1] == targetName[1]) foundFile += dirPath + foundName + "\n";

                } else if (targetName[0] != "*" && targetName[1] != "*") {  //[fileName] = a.txt : 查找 a.txt 文件;
                    // 按文件名筛选文件;
                    if (foundName == fileName) foundFile += dirPath + foundName + "\n";
                }
            }
        }

        return foundFile;
    }

    // 获取文件类型
    String getContentType(String filename) {
        if (filename.endsWith(".htm"))
            return "text/html";
        else if (filename.endsWith(".html"))
            return "text/html";
        else if (filename.endsWith(".css"))
            return "text/css";
        else if (filename.endsWith(".js"))
            return "application/javascript";
        else if (filename.endsWith(".png"))
            return "image/png";
        else if (filename.endsWith(".gif"))
            return "image/gif";
        else if (filename.endsWith(".jpg"))
            return "image/jpeg";
        else if (filename.endsWith(".ico"))
            return "image/x-icon";
        else if (filename.endsWith(".xml"))
            return "text/xml";
        else if (filename.endsWith(".pdf"))
            return "application/x-pdf";
        else if (filename.endsWith(".zip"))
            return "application/x-zip";
        else if (filename.endsWith(".gz"))
            return "application/x-gzip";
        return "text/plain";
    }

} FFileS;

// 时间控制类;
class TimeRefresh {
   public:
    // 系统的本地时间;
```

```
    typedef struct SystemTime {
        // Year Month Day Hour Minute Second
        unsigned short year = 0;
        unsigned char month = 0;
        unsigned char day = 0;
        unsigned char hour = 0;
        unsigned char minute = 0;
        unsigned char second = 0;

        // 当前代码实现了对系统日期的更新;
        void updateTime() {
            second++;
            if (second >= 60) {
                second = 0;
                minute++;
            }
            if (minute >= 60) {
                minute = 0;
                hour++;
            }
            if (hour >= 24) {
                hour = 0;
                day++;
            }

            unsigned char daysInMonth = 31;
            if (month == 2) {
                // 当 month 等于 2 时，使用三目运算符，以计算当前年份是否为闰年。如果是闰年，则 daysInMonth 的值将设置为 29；否则为
28。
                daysInMonth = (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0)) ? 29 : 28;
            } else if (month == 4 || month == 6 || month == 9 || month == 11) {
                daysInMonth = 30;  // 对于 4,6,9,11 月只有 30 天,将 daysInMonth 的值设置为 30;
            }

            // 如果天数大于当前月份的天数，代码将月份增加 1，并在必要时将年份增加 1;
            if (day > daysInMonth) {
                day = 1;
                month++;
                if (month > 12) {
                    month = 1;
                    year++;
                }
            }
        }

        // 解析网络时间的字符串，并设置到系统时间;
        void setSystemTime(String netWorkTimeStr) {
            year = static_cast<unsigned short>(netWorkTimeStr.substring(0, 4).toInt());
            month = static_cast<unsigned char>(netWorkTimeStr.substring(4, 6).toInt());
            day = static_cast<unsigned char>(netWorkTimeStr.substring(6, 8).toInt());
            hour = static_cast<unsigned char>(netWorkTimeStr.substring(8, 10).toInt());
            minute = static_cast<unsigned char>(netWorkTimeStr.substring(10, 12).toInt());
            second = static_cast<unsigned char>(netWorkTimeStr.substring(12, 14).toInt());
        }

    } SystemTime;

    SystemTime sysTime;
    String netWorkTimeStr = "";  // 储存从网络获取的时间(20230202135512);

    bool allow = false;  // 允许时间刷新;

    // 用于更新时间;
    void begin() {
```

```
        // 不触发警报的条件下每隔 10min 同步一次网络时间(多线程);
        TimeRefresh_ticker.attach(600, [this](void) -> void {
            // 如果系统进入 freezeMode(浅度睡眠)停止定时调用函数;
            if (freezeMode == true) TimeRefresh_ticker.detach();

            if (digitalRead(SENOUT) == HIGH) allow = true;  // 授予获取网络时间许可证;
        });

        // 每隔 1s 更新一次系统本地日期和时间;
        System_time.attach(1, [this](void) -> void { sysTime.updateTime(); });
    }

    // 从授时网站获得时间
    void getNetWorkTime() {
        uint8 httpCode = http.GET();
        if (httpCode > 0) {
            if (httpCode == HTTP_CODE_OK) {
                netWorkTimeStr = http.getString();  // 获取 JSON 字符串

                // 解析 JSON 数据
                StaticJsonDocument<200> doc;           // 创建一个 StaticJsonDocument 对象
                deserializeJson(doc, netWorkTimeStr);  // 使用 deserializeJson()函数来解析 Json 数据

                netWorkTimeStr = doc["sysTime1"].as<String>();  // 读取 JSON 数据;

                sysTime.setSystemTime(netWorkTimeStr);  // 解析网络时间的字符串，并设置到系统时间;

                // Serial.println(netWorkTimeStr);
            } else {
                Serial.printf("[HTTP GET Failed] ErrorCode: %s\n", http.errorToString(httpCode).c_str());
            }
        } else {
            Serial.printf("[HTTP GET Failed] ErrorCode: %s\n", http.errorToString(httpCode).c_str());
        }
        http.end();
    }

    // 格式化时间(在个位数前添加 0; 例如: 1 -> 01);
    String format(unsigned char timeInt) {
        if (timeInt < 10) {
            return "0" + String(timeInt);
        } else {
            return String(timeInt);
        }
    }

    // 读取时间(mode: true 读取 netWorkTimeStr_Format 的时间[时:分]; false 读取 netWorkTimeStr 的时间[年月日时分秒]);
    String timeRead(bool mode = true) {
        if (mode == true) {
            return format(sysTime.hour) + ":" + format(sysTime.minute);
        } else {
            return String(sysTime.year) + format(sysTime.month) + format(sysTime.day) + format(sysTime.hour) +
format(sysTime.minute) +
                   format(sysTime.second);
        }
    }

} timeRef;

// 获取实时天气类;
class Weather {
   public:
    void beginWeather() {
        // 读取 Weather_Config.ini 文件(保存了私钥和位置), 以<PRIVATEKEY/LOCATION>分割字符串;
```

```
        vector<String> WeatherConfig = oled.strsplit(FFileS.readFile("", "/Weather_Config.ini"),
"<PRIVATEKEY/LOCATION>");

        // 配置心知天气请求信息
        weatherNow.config(WeatherConfig[0], WeatherConfig[1], "c");

        // 不触发警报的条件下每隔 15min 更新一次天气信息(多线程);
        UpdateWeather.attach(900, [this](void) -> void {
            // 如果系统进入 freezeMode(浅度睡眠)停止定时调用函数;
            if (freezeMode == true) UpdateWeather.detach();

            if (digitalRead(SENOUT) == HIGH) updateWeather();
        });
    }

    // 更新天气信息;
    String updateWeather() {
        if (!weatherNow.update()) {
            return "Weather Update Fail: " + weatherNow.getServerCode();  // 更新失败;
        } else {
            return "";
        }
    }

    // 设置城市 ID;
    String setCityID(String cityID) {
        // 读取 Weather_Config.ini 文件(保存了私钥和位置), 以<PRIVATEKEY/LOCATION>分割字符串;
        vector<String> WeatherConfig = oled.strsplit(FFileS.readFile("", "/Weather_Config.ini"),
"<PRIVATEKEY/LOCATION>");

        // 保持私钥不变, 覆盖原有的配置文件;
        FFileS.fileCover(WeatherConfig[0] + "<PRIVATEKEY/LOCATION>" + cityID, "", "/Weather_Config.ini");

        // 配置心知天气请求信息
        weatherNow.config(WeatherConfig[0], cityID, "c");

        // 更新天气信息;
        return updateWeather();
    }

} weather;

typedef struct ANIM_INDEX {
    unsigned short name;      // 动画名称
    unsigned short Duration;  // 动画播放时长;
    unsigned short Begin;     // 动画播放起始帧;
    unsigned short End;       // 动画播放结束帧;
    unsigned char IMG_Width;  // 动画帧宽度;
    unsigned char IMG_Hight;  // 动画帧高度;
} ANIM_INDEX;

// 动画控制类(动画数组的最后一张用来清空动画显示区);
class Animation {
   private:
    ANIM_INDEX Index;  // 当前选中的动画;

    POINT Pos = {0, 0};                    // 动画播放的坐标(左上角为原点);
    unsigned short Duration = UINT16_MIN;  // 动画播放时长;
    unsigned short Begin = UINT16_MAX;     // 动画播放起始帧;
    unsigned short End = UINT16_MIN;       // 动画播放结束帧;

    // 动画控制器(索引);
    void AnimController() {
        // 如果没有设置(自定义)动画播放参数则使用对应动画的默认参数;
```

```
        if (Duration == UINT16_MIN) Duration = Index.Duration;
        if (Begin == UINT16_MAX) Begin = Index.Begin;
        if (End == UINT16_MIN) End = Index.End;

        // 根据播放时长计算每一帧的显示在屏幕上的时间(帧长度);
        unsigned long Sleep_ms = static_cast<unsigned long>(static_cast<float>(Duration) / static_cast<float>(End -
Begin));

        // 根据索引内的动画名称播放指定动画;
        if (Index.name == loading_X16_30F.name) {
            for (unsigned short i = Begin; i < End; ++i) {
                oled.OLED_DrawBMP(Pos.x, Pos.y, Index.IMG_Width, Index.IMG_Hight, Loading_X16_30F[i]);
                delay(Sleep_ms);
            }
        }
        if (Index.name == loading_X16_60F.name) {
            for (unsigned short i = Begin; i < End; ++i) {
                oled.OLED_DrawBMP(Pos.x, Pos.y, Index.IMG_Width, Index.IMG_Hight, Loading_X16_60F[i]);
                delay(Sleep_ms);
            }
        }
        if (Index.name == loadingBackForthBar_60x8_60F.name) {
            for (unsigned short i = Begin; i < End; ++i) {
                oled.OLED_DrawBMP(Pos.x, Pos.y, Index.IMG_Width, Index.IMG_Hight, LoadingBackForthBar_60x8_60F[i]);
                delay(Sleep_ms);
            }
        }
        if (Index.name == loadingBar_60x8_30F.name) {
            for (unsigned short i = Begin; i < End; ++i) {
                oled.OLED_DrawBMP(Pos.x, Pos.y, Index.IMG_Width, Index.IMG_Hight, LoadingBar_60x8_30F[i]);
                delay(Sleep_ms);
            }
        }
        if (Index.name == loadingBar_60x8_60F.name) {
            for (unsigned short i = Begin; i < End; ++i) {
                oled.OLED_DrawBMP(Pos.x, Pos.y, Index.IMG_Width, Index.IMG_Hight, LoadingBar_60x8_60F[i]);
                delay(Sleep_ms);
            }
        }
        //......
    }

  public:
    // 在这里声明动画索引;
    // ANIM_INDEX{Name, Duration, Begin, End, IMG_Width, IMG_Hight};
    ANIM_INDEX loading_X16_30F = {0, 250, 0, 30, 16, 16};
    ANIM_INDEX loading_X16_60F = {1, 500, 0, 60, 16, 16};
    ANIM_INDEX loadingBackForthBar_60x8_60F = {2, 500, 0, 60, 60, 8};
    ANIM_INDEX loadingBar_60x8_30F = {3, 300, 0, 30, 60, 8};
    ANIM_INDEX loadingBar_60x8_60F = {4, 500, 0, 60, 60, 8};
    //......

    // 设置动画播放参数(坐标{x, y}, 播放时长[ms], 起始帧, 结束帧);
    void setAnimation(u8 x, u8 y, u16 duration = UINT16_MIN, u16 begin = UINT16_MAX, u16 end = UINT16_MIN) {
        Pos = {x, y};
        Duration = duration;
        Begin = begin;
        End = end;
    }

    // 开始播放指定动画(索引);
    void runAnimation(ANIM_INDEX index) {
        Index = index;
        AnimController();
```

```
    }
} anim;

// OLED 显示消防预警;
void ShowFireWarning() {
    oled.OLED_DrawBMP(0, 0, 32, 64, FireWarning_32x64[0]);
    oled.OLED_DrawBMP(32, 0, 32, 64, FireWarning_32x64[1]);
    oled.OLED_DrawBMP(64, 0, 32, 64, FireWarning_32x64[2]);
    oled.OLED_DrawBMP(96, 0, 32, 64, FireWarning_32x64[3]);
}

class Desktop {
   private:
    // 任务栏图标排序表(储存了任务栏图标的动态显示位置);
    typedef struct StatusBars_Ranked {
        // 图标编号(对应 oledfont.h 中的编号);
        unsigned char Clear_Icon = 46;

        unsigned char Charging = 0;
        unsigned char WIFI = 1;
        unsigned char ProgramDownload = 2;
        unsigned char Disconnected = 3;
        unsigned char Battery = 4;
        unsigned char CMDCP = 5;

        unsigned char Sunny_Day_0 = 6;
        unsigned char Clear_Night_1 = 7;
        unsigned char Sunny_Day_2 = 8;
        unsigned char Clear_Night_3 = 9;
        unsigned char Cloudy_4 = 10;
        unsigned char Partly_Cloudy_Day_5 = 11;
        unsigned char Partly_Cloudy_Night_6 = 12;
        unsigned char Mostly_Cloudy_Day_7 = 13;
        unsigned char Mostly_Cloudy_Night_8 = 14;
        unsigned char Overcast_9 = 15;
        unsigned char Shower_10 = 16;
        unsigned char Thundershower_11 = 17;
        unsigned char Thundershower_with_Hail_12 = 18;
        unsigned char Light_Rain_13 = 19;
        unsigned char Moderate_Rain_14 = 20;
        unsigned char Heavy_Rain_15 = 21;
        unsigned char Storm_16 = 22;
        unsigned char Heavy_Storm_17 = 23;
        unsigned char Severe_Storm_18 = 24;
        unsigned char Ice_Rain_19 = 25;
        unsigned char Sleet_20 = 26;
        unsigned char Snow_Flurry_21 = 27;
        unsigned char Light_Snow_22 = 28;
        unsigned char Moderate_Snow_23 = 29;
        unsigned char Heavy_Snow_24 = 30;
        unsigned char Snowstorm_25 = 31;
        unsigned char Dust_26 = 32;
        unsigned char Sand_27 = 33;
        unsigned char Duststorm_28 = 34;
        unsigned char Sandstorm_29 = 35;
        unsigned char Foggy_30 = 36;
        unsigned char Haze_31 = 37;
        unsigned char Windy_32 = 38;
        unsigned char Blustery_33 = 39;
        unsigned char Hurricane_34 = 40;
        unsigned char Tropical_Storm_35 = 41;
        unsigned char Tornado_36 = 42;
        unsigned char Cold_37 = 43;
        unsigned char Hot_38 = 44;
```

```
        unsigned char Unknown_99 = 45;
        //......

        unsigned char unit = 16;  // 图标显示单位(16x16 图标);

        vector<unsigned char> StatusBars_Pos;  // 动态储存图标位置和注册状态，图标在状态栏中的位置就是在 vector 中的位置，
vector 中的内容表示注册状态;

        /*
        // 图标注册标记状态(true:注册; false:注销);
        bool Register_State[64];

        // 动态储存图标位置(初始位置设为-16);
        int StatusBars_Pos[8] = {-16, -16, -16, -16, -16, -16, -16, -16};
        */

    } StatusBars_Ranked;
    StatusBars_Ranked SBR;

    // 注册状态栏图标的位置(状态名称,模式[false:注销图标; true:注册图标]);
    void Icon_Register(unsigned char name, bool mode) {
        /*
        if (mode == false && SBR.Register_State[name] == true) {  // 在状态栏注销图标;
            SBR.Register_State[name] = false;                     // 标记注销图标;

            // 遍历所有图标的位置,将注销图标右方的所有图标左移一个图标显示单位;
            for (auto& i : SBR.StatusBars_Pos)
                if (i > SBR.StatusBars_Pos[name]) i -= SBR.unit;

            // 清空注销图标(包括注销图标)右方的区域;
            for (unsigned char i = SBR.StatusBars_Pos[name]; i <= 112; i += SBR.unit) oled.OLED_DrawBMP(i, 6, 16, 16,
StatusBars[SBR.Clear_Icon]);

            SBR.StatusBars_Pos[name] = 0;  // 将注销图标的位置归零;

        } else if (mode == true && SBR.Register_State[name] == false) {  // 在状态栏注册图标
            SBR.Register_State[name] = true;                              // 标记注册图标;

            // 分配注册图标的位置: 注册图标的位置 = 所有图标位置的最大值 + 一个图标显示单位;
            SBR.StatusBars_Pos[name] = tool.findArrMax(SBR.StatusBars_Pos, 8) + SBR.unit;
        }
        */

        short Register_Pos = -16;  // 将注册位置初始化为-16;

        // 查找 vector 中"name"的下标，下标位置就是注册位置(注册位置*16 就是图标在显示屏中的位置);
        auto it = std::find(SBR.StatusBars_Pos.begin(), SBR.StatusBars_Pos.end(), name);
        if (it != SBR.StatusBars_Pos.end()) {
            Register_Pos = static_cast<short>(std::distance(SBR.StatusBars_Pos.begin(), it));
        }

        if (mode == false && Register_Pos != -16) {  // 在状态栏注销图标;
            // 清空注销图标(包括注销图标)右方的区域;
            for (unsigned char i = Register_Pos * SBR.unit; i <= 112; i += SBR.unit) oled.OLED_DrawBMP(i, 6, 16, 16,
StatusBars[SBR.Clear_Icon]);

            // remove 函数在 vector 中删除所有与 name 相等的元素，erase 函数删除 vector 中剩余的空元素(注销图标)。
            SBR.StatusBars_Pos.erase(remove(SBR.StatusBars_Pos.begin(), SBR.StatusBars_Pos.end(), name),
SBR.StatusBars_Pos.end());

        } else if (mode == true && Register_Pos == -16) {  // 在状态栏注册图标
            SBR.StatusBars_Pos.push_back(name);            // 在状态栏的最后添加新注册的图标;
            Register_Pos = SBR.StatusBars_Pos.size() - 1;  // 获取最后一个图标位置(即新注册图标的位置);
        }
```

```
    if (mode == true) {
        oled.OLED_DrawBMP(Register_Pos * SBR.unit, 6, 16, 16, StatusBars[name]);  // 显示图标;
    }
}

// 渲染状态栏(注意多线程使用循环可能会出问题);
void StatusBars_Render() {
    // 正在充电状态;
    if (Charging_State == true) {
        Icon_Register(SBR.Charging, true);  // 注册图标位置;
    } else {
        Icon_Register(SBR.Charging, false);  // 注销图标位置;
    }

    // WIFI 连接状态;
    if (WIFI_State == true) {
        Icon_Register(SBR.WIFI, true);  // 注册图标位置;
    } else {
        Icon_Register(SBR.WIFI, false);  // 注销图标位置;
    }

    // 程序下载模式状态;
    if (digitalRead(0) == LOW) {
        Icon_Register(SBR.ProgramDownload, true);  // 注册图标位置;
    } else {
        Icon_Register(SBR.ProgramDownload, false);  // 注销图标位置;
    }

    // WIFI 断开状态;
    if (WIFI_State == false) {
        Icon_Register(SBR.Disconnected, true);  // 注册图标位置;
    } else {
        Icon_Register(SBR.Disconnected, false);  // 注销图标位置;
    }

    // 停止充电状态;
    if (Charging_State == false) {
        Icon_Register(SBR.Battery, true);  // 注册图标位置;
    } else {
        Icon_Register(SBR.Battery, false);  // 注销图标位置;
    }

    // CMDCP 被打开;
    if (CMDCP_State == true) {
        Icon_Register(SBR.CMDCP, true);  // 注册图标位置;
    } else {
        Icon_Register(SBR.CMDCP, false);  // 注销图标位置;
    }

    if (weatherNow.getWeatherCode() == 99) {
        Icon_Register(SBR.Unknown_99, true);  // 注册图标位置;
    } else {
        Icon_Register(SBR.Unknown_99, false);  // 注销图标位置;
    }

    for (unsigned char i = SBR.Sunny_Day_0; i <= SBR.Hot_38; ++i) {
        if (weatherNow.getWeatherCode() == (i - SBR.Sunny_Day_0)) {
            Icon_Register(i, true);  // 注册图标位置;
        } else {
            Icon_Register(i, false);  // 注销图标位置;
        }
    }
```

```
        /*
        // 晴（国内城市白天晴）;
        if (weather.weatherCode == 0) {
            Icon_Register(SBR.Sunny_Day_0, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Sunny_Day_0, false);  // 注销图标位置;
        }
        // 晴（国内城市夜晚晴）;
        if (weather.weatherCode == 1) {
            Icon_Register(SBR.Clear_Night_1, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Clear_Night_1, false);  // 注销图标位置;
        }
        // 晴（国外城市白天晴）;
        if (weather.weatherCode == 2) {
            Icon_Register(SBR.Sunny_Day_2, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Sunny_Day_2, false);  // 注销图标位置;
        }
        // 晴（国外城市夜晚晴）;
        if (weather.weatherCode == 3) {
            Icon_Register(SBR.Clear_Night_3, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Clear_Night_3, false);  // 注销图标位置;
        }
        // 多云;
        if (weather.weatherCode == 4) {
            Icon_Register(SBR.Cloudy_4, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Cloudy_4, false);  // 注销图标位置;
        }
        // 晴间多云(日);
        if (weather.weatherCode == 5) {
            Icon_Register(SBR.Partly_Cloudy_Day_5, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Partly_Cloudy_Day_5, false);  // 注销图标位置;
        }
        // 晴间多云(夜);
        if (weather.weatherCode == 6) {
            Icon_Register(SBR.Partly_Cloudy_Night_6, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Partly_Cloudy_Night_6, false);  // 注销图标位置;
        }
        // 大部多云(日);
        if (weather.weatherCode == 7) {
            Icon_Register(SBR.Mostly_Cloudy_Day_7, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Mostly_Cloudy_Day_7, false);  // 注销图标位置;
        }
        // 大部多云(夜);
        if (weather.weatherCode == 8) {
            Icon_Register(SBR.Mostly_Cloudy_Night_8, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Mostly_Cloudy_Night_8, false);  // 注销图标位置;
        }
        // 阴;
        if (weather.weatherCode == 9) {
            Icon_Register(SBR.Overcast_9, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Overcast_9, false);  // 注销图标位置;
        }
        // 阵雨;
        if (weather.weatherCode == 10) {
            Icon_Register(SBR.Shower_10, true);  // 注册图标位置;
```

```
        } else {
            Icon_Register(SBR.Shower_10, false);  // 注销图标位置;
        }
        // 雷阵雨;
        if (weather.weatherCode == 11) {
            Icon_Register(SBR.Thundershower_11, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Thundershower_11, false);  // 注销图标位置;
        }
        // 雷阵雨伴有冰雹;
        if (weather.weatherCode == 12) {
            Icon_Register(SBR.Thundershower_with_Hail_12, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Thundershower_with_Hail_12, false);  // 注销图标位置;
        }
        // 小雨;
        if (weather.weatherCode == 13) {
            Icon_Register(SBR.Light_Rain_13, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Light_Rain_13, false);  // 注销图标位置;
        }
        // 中雨;
        if (weather.weatherCode == 14) {
            Icon_Register(SBR.Moderate_Rain_14, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Moderate_Rain_14, false);  // 注销图标位置;
        }
        // 大雨;
        if (weather.weatherCode == 15) {
            Icon_Register(SBR.Heavy_Rain_15, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Heavy_Rain_15, false);  // 注销图标位置;
        }
        // 暴雨;
        if (weather.weatherCode == 16) {
            Icon_Register(SBR.Storm_16, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Storm_16, false);  // 注销图标位置;
        }
        // 大暴雨;
        if (weather.weatherCode == 17) {
            Icon_Register(SBR.Heavy_Storm_17, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Heavy_Storm_17, false);  // 注销图标位置;
        }
        // 特大暴雨;
        if (weather.weatherCode == 18) {
            Icon_Register(SBR.Severe_Storm_18, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Severe_Storm_18, false);  // 注销图标位置;
        }
        // 冻雨;
        if (weather.weatherCode == 19) {
            Icon_Register(SBR.Ice_Rain_19, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Ice_Rain_19, false);  // 注销图标位置;
        }
        // 雨夹雪;
        if (weather.weatherCode == 20) {
            Icon_Register(SBR.Sleet_20, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Sleet_20, false);  // 注销图标位置;
        }
        // 阵雪;
```

```
        if (weather.weatherCode == 21) {
            Icon_Register(SBR.Snow_Flurry_21, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Snow_Flurry_21, false);  // 注销图标位置;
        }
        // 小雪;
        if (weather.weatherCode == 22) {
            Icon_Register(SBR.Light_Snow_22, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Light_Snow_22, false);  // 注销图标位置;
        }
        // 中雪;
        if (weather.weatherCode == 23) {
            Icon_Register(SBR.Moderate_Snow_23, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Moderate_Snow_23, false);  // 注销图标位置;
        }
        // 大雪;
        if (weather.weatherCode == 24) {
            Icon_Register(SBR.Heavy_Snow_24, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Heavy_Snow_24, false);  // 注销图标位置;
        }
        // 暴雪;
        if (weather.weatherCode == 25) {
            Icon_Register(SBR.Snowstorm_25, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Snowstorm_25, false);  // 注销图标位置;
        }
        // 浮尘;
        if (weather.weatherCode == 26) {
            Icon_Register(SBR.Dust_26, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Dust_26, false);  // 注销图标位置;
        }
        // 扬沙;
        if (weather.weatherCode == 27) {
            Icon_Register(SBR.Sand_27, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Sand_27, false);  // 注销图标位置;
        }
        // 沙尘暴;
        if (weather.weatherCode == 28) {
            Icon_Register(SBR.Duststorm_28, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Duststorm_28, false);  // 注销图标位置;
        }
        // 强沙尘暴;
        if (weather.weatherCode == 29) {
            Icon_Register(SBR.Sandstorm_29, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Sandstorm_29, false);  // 注销图标位置;
        }
        // 雾;
        if (weather.weatherCode == 30) {
            Icon_Register(SBR.Foggy_30, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Foggy_30, false);  // 注销图标位置;
        }
        // 霾;
        if (weather.weatherCode == 31) {
            Icon_Register(SBR.Haze_31, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Haze_31, false);  // 注销图标位置;
```

```
        }
        // 风;
        if (weather.weatherCode == 32) {
            Icon_Register(SBR.Windy_32, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Windy_32, false);  // 注销图标位置;
        }
        // 大风;
        if (weather.weatherCode == 33) {
            Icon_Register(SBR.Blustery_33, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Blustery_33, false);  // 注销图标位置;
        }
        // 飓风;
        if (weather.weatherCode == 34) {
            Icon_Register(SBR.Hurricane_34, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Hurricane_34, false);  // 注销图标位置;
        }
        // 热带风暴;
        if (weather.weatherCode == 35) {
            Icon_Register(SBR.Tropical_Storm_35, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Tropical_Storm_35, false);  // 注销图标位置;
        }
        // 龙卷风;
        if (weather.weatherCode == 36) {
            Icon_Register(SBR.Tornado_36, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Tornado_36, false);  // 注销图标位置;
        }
        // 冷;
        if (weather.weatherCode == 37) {
            Icon_Register(SBR.Cold_37, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Cold_37, false);  // 注销图标位置;
        }
        // 热;
        if (weather.weatherCode == 38) {
            Icon_Register(SBR.Hot_38, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Hot_38, false);  // 注销图标位置;
        }
        // 未知;
        if (weather.weatherCode == 99) {
            Icon_Register(SBR.Unknown_99, true);  // 注册图标位置;
        } else {
            Icon_Register(SBR.Unknown_99, false);  // 注销图标位置;
        }
        */

        //......
    }

  public:
    void begin() {
        // 不触发警报的条件下每隔 500ms 刷新一次状态栏(多线程);
        Desktop_ticker.attach_ms(500, [this](void) -> void {
            // 如果系统进入 freezeMode(浅度睡眠)停止定时调用函数;
            if (freezeMode == true) Desktop_ticker.detach();

            // 如果处于开发者模式则停止刷新状态栏;
            if (digitalRead(SENOUT) == HIGH && Developer_Mode == false) {
                // 如果每分钟中秒数走到 0(即开头)则刷新一次桌面时钟;
```

```
                if (timeRef.sysTime.second == 0) oled.OLED_ShowString(4, -1, timeRef.timeRead().c_str(), 49);  // 刷新时
间;

                StatusBars_Render();  // 刷新状态栏;
            }
        });
    }

    // 渲染主桌面;
    void Main_Desktop() {
        oled.OLED_ShowString(4, -1, timeRef.timeRead().c_str(), 49);  // 刷新时间;
        StatusBars_Render();                                          // 渲染状态栏;
    }
} Desktop;

// 判断电池是否进入充电并显示电池开始充电的信息;
void Show_Charging_info() {
    if (digitalRead(CHRG) == LOW && Charging_State == false) {
        Charging_State = true;  // 充电状态设为 true;

        // 播放动画等待电压稳定(电压不稳就操作大功耗器件有可能造成 MCU 复位);
        anim.setAnimation(112, 6, 400);                                    // 设置加载动画显示位置,设置播放时长为 400ms;
        for (u8 i = 0; i < 2; ++i) anim.runAnimation(anim.loading_X16_60F);  // 播放加载动画(两次);
        oled.OLED_DrawBMP(112, 6, 16, 16, Loading_X16_60F[60]);             // 清空动画播放区;

        /*
        delay(1000);                                    // 等待电压稳定(电压不稳就操作大功耗器件有可能造成 MCU 复位);
        oled.OLED_DrawBMP(0, 0, 128, 64, Charging_IMG);  // 显示正在充电提示;
        delay(1000);
        oled.OLED_Clear();  // 清除界面
        Main_Desktop();     // 渲染主桌面;
        */
    } else if (digitalRead(CHRG) == HIGH && Charging_State == true) {
        Charging_State = false;  // 充电状态设为 false;
    }
}

// 程序下载模式(下载程序必须下拉 GPIO0 并且复位);
void ProgramDownloadMode() {
    // 条件: GPIO0 下拉, 允许进入下载模式, 不处于浅度休眠模式;
    if (digitalRead(Decoder_C) == LOW && allowDownloadMode == true && freezeMode == false) {
        bool allowReset = true;

        // 显示主桌面提示即将进入下载模式;
        oled.OLED_DrawBMP(0, 0, 128, 48, DownloadMode_IMG);

        // 设置加载动画显示位置,设置播放时长为 400ms;
        anim.setAnimation(112, 6, 400);

        // 等待 2400ms, 反悔时间约 3s 内断开 GPIO0 的下拉, 系统会进入开发者模式;
        for (u8 i = 0; i < 6; ++i) {
            anim.runAnimation(anim.loading_X16_60F);  // 播放等待动画;
            if (digitalRead(0) == HIGH) {
                oled.OLED_DrawBMP(112, 6, 16, 16, Loading_X16_60F[60]);  // 清空动画播放区;
                allowReset = false;                                      // 不允许复位进入下载模式;
                break;
            }
        }

        if (allowReset == true) {
            oled.OLED_Display_Off();  // 关闭 OLED 显示屏;
            // LittleFS.format();        // 格式化闪存文件系统;
            digitalWrite(RST, LOW);  // 复位进入下载模式;
        }
```

```
    }
}

class SystemSleep {
   private:
   public:
    // 读取主要 GPIO 管脚的状态;
    String GPIO_Read() {
        // 首先读取译码器控制引脚的状态;
        int decoderC = digitalRead(Decoder_C);
        int decoderB = digitalRead(Decoder_B);
        int decoderA = digitalRead(Decoder_A);

        // 解释译码器引脚的状态;
        String decodedWith = "\nDecoded_with=";
        switch (decoderC << 2 | decoderB << 1 | decoderA) {
            case 1:
                decodedWith += "Buzzer_Enable";
                break;
            case 2:
                decodedWith += "RedLED_Enable";
                break;
            case 3:
                decodedWith += "GreenLED_Enable";
                break;
            case 4:
                decodedWith += "BlueLED_Enable";
                break;
            case 6:
                decodedWith += "Sensor_and_OLED_disabled";
                break;
            default:
                decodedWith += "NULL";
                break;
        }

        // 其余引脚的状态直接读出;
        String GPIO_State = "RST=" + String(digitalRead(RST)) + "\nTXD=" + String(digitalRead(TXD)) + "\nRXD=" +
String(digitalRead(RXD)) +
                            "\nSCL=" + String(digitalRead(SCL)) + "\nSDA=" + String(digitalRead(SDA)) + "\nCHRG=" +
String(digitalRead(CHRG)) +
                            "\nLOWPOWER=" + String(digitalRead(LOWPOWER)) + "\nSENOUT=" + String(digitalRead(SENOUT)) +
"\nDecoder_C=" + String(decoderC) +
                            "\nDecoder_B=" + String(decoderB) + "\nDecoder_A=" + String(decoderA) + decodedWith;
        return GPIO_State;
    }

    // 获取系统模式和状态;
    String getSysModeAndStatus() {
        return "Charging_State=" + String(Charging_State) + "\nWIFI_State=" + String(WIFI_State) + "\nDeveloper_Mode="
+ String(Developer_Mode) +
               "\nallowResponse=" + String(allowResponse) + "\nallowDownloadMode=" + String(allowDownloadMode) +
"\nfreezeMode=" + String(freezeMode) +
               "\ndiskMode=" + String(diskMode);
    }

    // 删除 diskMode 深度睡眠缓存的数据文件;
    void removeSleepFile() {
        // 如果存放休眠文件的目录存在则删除这个目录;
        if (LittleFS.exists("/SleepFile")) FFileS.removeDirector("", "/SleepFile");
    }

    // 从深度睡眠 diskMode 恢复系统;
    void resumeFromDeepSleep() {
```

```
        LittleFS.begin();       // 启动闪存文件系统
        String SleepFile = "";  // 休眠文件字符串结构:
String("Name1="+"Status1"+"\nName2="+"Status2"+"\nName3="+"Status3".....)

        // 如果[GPIO_Status]休眠文件存在则从该文件恢复系统;
        if (LittleFS.exists("/SleepFile/GPIO_Status.txt")) {
            // [GPIO_Status]-恢复深度睡眠前输出 GPIO 的状态;
            SleepFile = FFileS.readFile("", "/SleepFile/GPIO_Status.txt");

            // 以换行符分割字符串;
            for (auto& i : oled.strsplit(SleepFile, "\n")) {
                vector<String> GPIO_Status = oled.strsplit(i, "=");  // 以等于符分割字符串;

                // 恢复译码器控制引脚的状态(恢复输出 GPIO 的状态);
                if (GPIO_Status[0] == "Decoder_C") digitalWrite(Decoder_C, GPIO_Status[1].toInt());
                if (GPIO_Status[0] == "Decoder_B") digitalWrite(Decoder_B, GPIO_Status[1].toInt());
                if (GPIO_Status[0] == "Decoder_A") digitalWrite(Decoder_A, GPIO_Status[1].toInt());
            }
            FFileS.removeFile("", "/SleepFile/GPIO_Status.txt");  // 恢复完成后删除[GPIO_Status]深度休眠文件;
        }

        // 如果[系统模式和状态]休眠文件存在则从该文件恢复系统;
        if (LittleFS.exists("/SleepFile/SysModeAndStatus.txt")) {
            // [System Mode&Status(系统模式和状态)]-恢复深度睡眠前系统模式和状态;
            SleepFile = FFileS.readFile("", "/SleepFile/SysModeAndStatus.txt");

            // 以换行符分割字符串;
            for (auto& i : oled.strsplit(SleepFile, "\n")) {
                vector<String> SysModeAndStatus = oled.strsplit(i, "=");  // 以等于符分割字符串;

                // if (SysModeAndStatus[0] == "Charging_State") Charging_State = (SysModeAndStatus[1].toInt() != 0);//
实时检测不需要恢复;
                // if (SysModeAndStatus[0] == "WIFI_State") WIFI_State = (SysModeAndStatus[1].toInt() != 0);//实时检测不
需要恢复;
                if (SysModeAndStatus[0] == "Developer_Mode") Developer_Mode = (SysModeAndStatus[1].toInt() != 0);  // 恢
复"开发者模式"的设置;
                if (SysModeAndStatus[0] == "allowResponse") allowResponse = (SysModeAndStatus[1].toInt() != 0);  // 恢复
"允许服务器对客户端进行响应"的设置;
                if (SysModeAndStatus[0] == "allowDownloadMode") allowDownloadMode = (SysModeAndStatus[1].toInt() !=
0);  // 恢复"允许进入下载模式"的设置;
                if (SysModeAndStatus[0] == "freezeMode") freezeMode = (SysModeAndStatus[1].toInt() !=
0);               // 恢复"浅度睡眠"设置;
                if (SysModeAndStatus[0] == "diskMode") diskMode = (SysModeAndStatus[1].toInt() !=
0);                    // 恢复"深度睡眠"设置;
            }
            FFileS.removeFile("", "/SleepFile/SysModeAndStatus.txt");  // 恢复完成后删除[系统模式和状态]深度休眠文件;
        }

        diskMode = false;  // 禁用深度睡眠;
    }

    //[休眠模式-freeze], 冻结 I/O 设备, 关闭外设, ESP-12F 进入 Modem-sleep 模式, 程序上只运行 CMDControlPanel 网络服务, 其他服务
冻结;
    void Sys_freezeMode(bool Enable = true) {
        if (Enable == true) {
            freezeMode = true;  // 启用浅度睡眠;

            oled.OLED_Display_Off();  // OLED 显示屏停止显示;

            // 关闭所有传感器(红外&气体),关闭 OLED 屏幕,关闭所有声光警报,切断除 MCU 外的一切供电;
            digitalWrite(Decoder_C, HIGH);
            digitalWrite(Decoder_B, HIGH);
            digitalWrite(Decoder_A, LOW);
```

```
            WiFi.setSleepMode(WIFI_MODEM_SLEEP);  // ESP-12F 进入 Modem-sleep 模式;
        } else if (Enable == false) {
            WiFi.setSleepMode(WIFI_NONE_SLEEP);  // ESP-12F 离开睡眠模式;

            freezeMode = false;  // 禁用浅度睡眠;

            // 重新给所有传感器(红外&气体)，OLED 屏幕，声光警报器上电;
            digitalWrite(Decoder_C, HIGH);
            digitalWrite(Decoder_B, HIGH);
            digitalWrite(Decoder_A, HIGH);

            // 大功率器件上电可能会造成局部电压波动，等待一段时间至电压稳定(最少等待 1s，最多等待 10s);
            for (uint8 i = 0; i < 10; ++i) {
                delay(1000);
                if (digitalRead(SENOUT) == HIGH) break;
            }

            oled.OLED_Init();          // 初始化 OLED
            oled.OLED_ColorTurn(0);    // 0 正常显示 1 反色显示
            oled.OLED_DisplayTurn(0);  // 0 正常显示 1 翻转 180 度显示

            timeRef.getNetWorkTime();  // 获取网络时间;
            oled.OLED_Clear();         // 清除界面
            Desktop.Main_Desktop();    // 渲染主桌面;

            timeRef.begin();  // 恢复时间刷新服务;
            Desktop.begin();  // 恢复桌面刷新服务;
        }
    }

    // disk [sleep time_us];
    //[深度休眠模式-disk] 运行状态(GPIO_Status，系统模式和状态，文本框信息)数据存到 Flash(醒来时恢复状态)，然后 ESP12F 进入深度
睡眠;
    void Sys_diskMode(uint64_t time_us = 0) {
        // 登出和锁定 CMDCP;
        CMDCP_State = false;     // 用户关闭 CMDCP;
        Developer_Mode = false;  // 退出开发者模式(显示状态栏和桌面时钟);

        diskMode = true;   // 启用深度睡眠;
        LittleFS.begin();  // 启动闪存文件系统

        // [GPIO_Status]-保存进入深度睡眠前的 GPIO 状态到 Flash，以便从深度睡眠醒来时恢复 GPIO 状态;
        File GPIO_StatusFile = LittleFS.open("/SleepFile/GPIO_Status.txt", "w");  // 创建&覆盖并打开 GPIO_Status.txt 文件;
        GPIO_StatusFile.print(GPIO_Read());                                        // 向 GPIO_StatusFile 写入 GPIO 状态信息;
        GPIO_StatusFile.close();                                                   // 完成文件写入后关闭文件;

        //[System Mode&Status(系统模式和状态)]-保存系统现在处于的系统模式和状态到 Flash，以便从深度睡眠醒来时恢复;
        File SysModeAndStatusFile = LittleFS.open("/SleepFile/SysModeAndStatus.txt", "w");  // 创建&覆盖并打开
SysModeAndStatus.txt 文件;
        SysModeAndStatusFile.print(getSysModeAndStatus());  // 向 SysModeAndStatusFile 写入当前系统模式和状态状态信息;
        SysModeAndStatusFile.close();                       // 完成文件写入后关闭文件;

        // [PrintBox(文本框)]-保存进入深度睡眠前的 OLED 上输出的文本信息到 Flash;
        // 文本框休眠文件的数据结构: String("OLED 屏幕第 1 行"+"\n"+"OLED 屏幕第 2 行"+"\n"+"OLED 屏幕第 3 行"+.....)
        File PrintBoxFile = LittleFS.open("/SleepFile/PrintBox.txt", "w");  // 创建&覆盖并打开 PrintBox.txt 文件;
        for (auto& i : oled.getPrintBox()) {
            PrintBoxFile.print(i + "\n");  // 向 PrintBoxFile 写入 OLED 屏幕打印的文本信息，在 OLED 屏幕上的每行字符串的末尾追
加"\n"后把所有行合并;
        }
        PrintBoxFile.close();  // 完成文件写入后关闭文件;

        ESP.deepSleep(time_us);  // ESP-12F 进入深度睡眠;
    }
} SysSleep;
```

```
class WebServer {
   private:
    File UploadFile;  // 建立文件对象用于文件上传至服务器闪存;
    bool routeUploadEnabled = false;

   public:
    String UploadRespond = "";  // 用于回复终端文件是否上次成功;

    // 检查 WIFI 是否连接,若没有连接则连接;
    void WiFi_Connect() {
        // 读取时间数据(从 RAM)如果数据为"00:00"则表示系统正在启动，否则表示系统正常运行时需要确认 WIFI 连接正常(两种情况播放的动画不同);
        if (timeRef.timeRead() == "00:00")
            anim.setAnimation(67, 6);  // 设置动画播放位置(其他参数默认);
        else
            anim.setAnimation(112, 6);  // 设置动画播放位置(其他参数默认);

        if (WiFi.status() != WL_CONNECTED) {
            WIFI_State = false;

            // 读取 WIFI_Config.ini 文件(保存了 WIFI 名称和密码)，以<SSID/PASSWD>分割字符串;
            vector<String> SSID_PASSWD = oled.strsplit(FFileS.readFile("", "/WIFI_Config.ini"), "<SSID/PASSWD>");

            // WiFi.begin(SSID_PASSWD[0], SSID_PASSWD[1]);
            WiFi.begin(SSID, PASSWORD);
            // 等待 WIFI 连接(超时时间为 10s);
            for (unsigned char i = 0; i < 100; ++i) {
                if (WiFi.status() == WL_CONNECTED) {
                    Serial.print("IP Address: ");
                    Serial.print(WiFi.localIP());
                    Serial.println(":" + String(ServerPort));

                    // WIFI 连接完成后清空动画播放区域;
                    //  读取时间数据(从 RAM)如果数据为"00:00"则表示系统正在启动，否则表示系统正常运行时需要确认 WIFI 连接正常(两种情况播放的动画不同);
                    if (timeRef.timeRead() == "00:00")
                        // 清空进度条加载动画区域;
                        oled.OLED_DrawBMP(67, 6, anim.loadingBar_60x8_30F.IMG_Width, anim.loadingBar_60x8_30F.IMG_Hight,
LoadingBar_60x8_30F[30]);

                    else
                        // 清空加载动画区域;
                        oled.OLED_DrawBMP(112, 6, anim.loading_X16_60F.IMG_Width, anim.loading_X16_60F.IMG_Hight,
Loading_X16_60F[30]);

                    WIFI_State = true;
                    break;
                } else {
                    // 读取时间数据(从 RAM)如果数据为"00:00"则表示系统正在启动，否则表示系统正常运行时需要确认 WIFI 连接正常(两种情况播放的动画不同);
                    if (timeRef.timeRead() == "00:00")
                        anim.runAnimation(anim.loadingBar_60x8_30F);  // 播放进度条加载动画;
                    else
                        anim.runAnimation(anim.loading_X16_60F);  // 播放加载动画;
                }
            }
        } else {
            WIFI_State = true;
        }
    }

    // 配置 MQTT;
    void MQTT_Begin() {
```

```
        // 设置 MQTT 服务器
        mqtt_client.setServer(MQTT_SERVER, 1883);
        // 一定要设置 keepAlive time 为较大值，默认值 15 会无法建立连接，推荐 60
        mqtt_client.setKeepAlive(60);

        MQTT_Client();  // 运行 MQTT 客户端;
    }

    // MQTT 客户端;
    void MQTT_Client() {
        if (mqtt_client.connected()) {  // 如果开发板成功连接服务器
            mqtt_client.loop();         // 处理信息以及心跳
        } else {                        // 如果开发板未能成功连接服务器
            // 则尝试连接服务器
            // 连接 MQTT 服务器
            if (mqtt_client.connect(MQTT_CLIENT_ID, MQTT_USRNAME, MQTT_PASSWD)) {
                Serial.println("MQTT Server Connected.");
                Serial.println("Server Address: ");
                Serial.println(MQTT_SERVER);
                Serial.println("ClientId:");
                Serial.println(MQTT_CLIENT_ID);
            } else {
                Serial.print("MQTT Server Connect Failed. Client State:");
                Serial.println(mqtt_client.state());
                delay(2000);
            }
        }
    }

    bool getRouteUploadStatus() { return routeUploadEnabled; }

    // 处理上传文件函数(用于将终端文件上传到服务器 Flash);
    void handleFileUpload() {
        HTTPUpload& upload = server.upload();

        if (upload.status == UPLOAD_FILE_START) {                          // 如果上传状态为 UPLOAD_FILE_START
            String filepath = FFileS.getWorkDirectory() + upload.filename;  // 建立字符串变量用于存放上传文件路径

            UploadFile = LittleFS.open(filepath, "w");  // 在 LittleFS 中建立文件用于写入用户上传的文件数据

        } else if (upload.status == UPLOAD_FILE_WRITE) {                       // 如果上传状态为 UPLOAD_FILE_WRITE
            if (UploadFile) UploadFile.write(upload.buf, upload.currentSize);  // 向 LittleFS 文件写入浏览器发来的文件数据

        } else if (upload.status == UPLOAD_FILE_END) {                                                  // 如果上传状
态为 UPLOAD_FILE_END
            if (UploadFile) {                                                                          // 如果文件成功
建立
                UploadFile.close();                                                                    // 将文件关闭
                UploadRespond = "Size: " + String(upload.totalSize) + " Byte" + "\nFile upload succeeded.";  // 返回完成
信息;
            } else {                                                                                  // 如果文件未能
成功建立
                UploadRespond = "File upload failed.";                                                 // 返回错误信
息;
            }
        }
    }

    String uploadFile() {
        UploadRespond = "";         // 清空响应字符串;
        routeUploadEnabled = true;  // 授予"Upload"路由开启许可，打开上传通道，允许文件上传;

        // 发送"上传许可"通知，告诉客户端服务器已就绪可以上传文件;
        server.send(200, "text/plain", "EnableUpload");
```

```
        // 接收文件流;
        while (true) {
            server.handleClient();

            // 如果"UploadRespond"有字符串，则表示文件已经上传完成(或者失败);
            if (UploadRespond != "") {
                // 关闭"/upload"路由(文件上传通道关闭);
                routeUploadEnabled = false;
                return UploadRespond;
            }
        }
    }
} WebServer;

class CMDControlPanel {
   private:
    String CMD = "";           // 用来缓存 CMD 指令;
    String PassWord = "";      // 使用 CMDControlPanel 前需要输入密码(系统启动时将 flash 中的密码缓存到此);
    bool LockerState = false;  // PassLocker 为 true 时表示 CMD 已经解锁，解锁状态将一直保存在 RAM 中直到 MCU 断电或用户通过命令
吊销;
    String CMDCP_Online_Response = "";  // 临时储存 CMD 的响应数据;
    vector<String> clientLogedIP;       // 这里储存了已登录并且正在使用还未登出 CMD 用户的 IP 地址(登出时删除 IP);

    // 使用 CMDControlPanel 前需要输入密码(无密码会要求用户设置密码，密码正确返回 true，密码错误返回 false);
    bool PassLocker() {
        allowResponse = false;  // 禁止服务器对客户端进行响应;
        CMD = "";               // 清空 CMD 缓存;
        String PassWord_Temp1 = "", PassWord_Temp2 = "";
        LittleFS.begin();  // 启动闪存文件系统;

        // 判断 Flash 中储存密匙的文件是否存在，如果不存在则要求用户设置密匙，如果存在则要用户输入密匙;
        if (LittleFS.exists("/PassWord.txt") == false) {
            // for 循环决定了超时时间，如果什么都不做则 90s 后超时;
            for (unsigned char j = 0; j < 3; ++j) {
                CMDCP_Response("Set password:");
                server.send(200, "text/html", CMDCP_Online_Response);  // 向客户端发送请设置密码的字符串(主动发送不是响应);

                // 等待 30s，期间可以输入第一次密码;
                for (unsigned char i = 0; i < 60; ++i) {
                    // SerialReceived();       // 获取串口或 WIFI 数据;
                    server.handleClient();  // 接收网络请求(获取从网络输入的密匙);

                    // 如果串口没有发送数据并且 CMD 不为空则将 CMD 的内容作为第一次输入的密匙;
                    if (CMD != "") {
                        PassWord_Temp1 = CMD;
                        CMD = "";  // 清空 CMD 缓存;
                        break;
                    }
                    delay(500);
                }
                // 若等待超时后仍然没有输入密匙则跳过本次的循环进入下一次循环;
                if (PassWord_Temp1 == "") continue;

                CMDCP_Response("Re-enter password:");
                server.send(200, "text/html", CMDCP_Online_Response);  // 向客户端发送请再次输入密码的字符串(主动发送不是响
应);

                // 等待 30s，期间可以输入第二次密码;
                for (unsigned char i = 0; i < 60; ++i) {
                    // SerialReceived();       // 获取串口或 WIFI 数据;
                    server.handleClient();  // 接收网络请求(获取从网络输入的密匙);

                    // 如果串口没有发送数据并且 CMD 不为空则将 CMD 的内容作为第二次输入的密匙;
```

```
                if (CMD != "") {
                    PassWord_Temp2 = CMD;
                    CMD = "";  // 清空 CMD 缓存;
                    break;
                }
                delay(500);
            }

            // 比较两次输入的密码(通过的条件是两次密码相同并且不为空);
            if (PassWord_Temp1 == PassWord_Temp2 && PassWord_Temp1 != "") {
                // 将设置的密码进行哈希加密后存入 Flash 中;
                File dataFile = LittleFS.open("/PassWord.txt", "w");  // 创建&覆盖并打开 PassWord.txt 文件;
                dataFile.print(String(sha1(PassWord_Temp1)));         // 向 dataFile 写入哈希加密后的密匙信息;
                dataFile.close();                                     // 完成文件写入后关闭文件

                CMDCP_Response("Accepted");
                server.send(200, "text/html", CMDCP_Online_Response);  // 向客户端发送密码设置成功的字符串(主动发送不是
响应);
                allowResponse = true;                                 // 允许服务器对客户端进行响应;
                CMDCP_State = false;                                  // 用户离开 CMDCP;
                return false;  // 密码设置成功结束 PassLocker 程序，重新启动 PassLocker 程序输入新密码即可;
            }
        }

    } else {
        // 将 Flash 中的密匙文件缓存到 PassWord 变量中;
        PassWord = FFileS.readFile("", "/PassWord.txt");
        // for 循环决定了超时时间，如果什么都不做则 90s 后超时，并且用户有三次机会输入正确的密码;
        for (unsigned char j = 0; j < 3; ++j) {
            CMDCP_Response("Enter password:");
            server.send(200, "text/html", CMDCP_Online_Response);  // 向客户端发送请输入密码的字符串(主动发送不是响应);

            // 等待 30s，期间可以输入密码;
            for (unsigned char i = 0; i < 60; ++i) {
                // SerialReceived();       // 获取串口或 WIFI 数据;
                server.handleClient();  // 接收网络请求(获取从网络输入的密匙);

                // 如果串口没有发送数据并且 CMD 不为空则将 CMD 的内容作为第二次输入的密匙;
                if (CMD != "") {
                    PassWord_Temp1 = String(sha1(CMD));  // 将输入的密码进行哈希加密后缓存到变量 PassWord_Temp1;
                    CMD = "";                            // 清空 CMD 缓存;
                    break;
                }
                delay(500);
            }

            // 比较输入密码与 PassWord 缓存中的密码是否一致，如果一致则输入的密码正确，返回 true;
            if (PassWord == PassWord_Temp1) {
                CMDCP_Response("Passed");
                server.send(200, "text/html", CMDCP_Online_Response);  // 向客户端发送密码正确的字符串(主动发送不是响
应);
                allowResponse = true;                                 // 允许服务器对客户端进行响应;
                return true;
            }
        }
    }

    CMDCP_State = false;   // 用户离开 CMDCP;
    allowResponse = true;  // 允许服务器对客户端进行响应;
    return false;          // 密码错误返回 false;
}

void CMDCP_Response(String Response) {
    CMDCP_Online_Response = "";  // 清空网络响应缓存;
```

```
        if (Response != "") {
            CMDCP_Online_Response = Response;  // 设置网络响应缓存;
            oled.print(Response);
            Serial.println(Response);
        }
    }

    // 保存执行的命令;
    void saveCmdHistory(String CMD, String clientIP) {
        LittleFS.begin();  // 启动闪存文件系统

        File cmdHistory = LittleFS.open("/CMD_History.txt", "a");                    // 打开 CMD_History.txt 追加日志;
        cmdHistory.print(clientIP + "-" + timeRef.timeRead(false) + "-" + CMD + "\n");  // IP 地址+时间+命令+换行;
        cmdHistory.close();
    }

   public:
    bool allow = false;

    void begin() {
        oled.setTextBox(0, 0, 128, 48);  // 设置文本框;
        // CMDControlPanel_ticker.attach_ms(1000, [this](void) -> void { allow = true; });
    }

    String CMDControlPanelOnlinePortal(String CMDCP_Online_Message) {
        // 如果客户端请求了空指令则直接结束函数响应空字符串;
        if (CMDCP_Online_Message == "") return "";

        CMD = CMDCP_Online_Message;  // 更新在线控制台发送的网络命令到 CMD 缓存;

        String clientLogingIP = server.client().remoteIP().toString();  // 获取用户的 IP 地址;

        // 检查用户的 IP 地址;
        bool LockerState = false;        // 将 CMD 设为锁定状态;
        for (auto& i : clientLogedIP) {  // 将请求用户的 IP 与已登录并且正在使用还未登出 CMD 用户的 IP 地址进行比对;
            if (i == clientLogingIP) {
                LockerState = true;  // 如果发现该 IP 还在登录状态(未登出)则为此用户开放 CMD;
                break;
            }
        }

        // 当处于用户未登录状态(锁定状态)并且接收到进入 CMDCP 的命令;
        if (LockerState == false && (CMD == "CMD" || CMD == "cmd" || CMD == "login")) {
            CMDCP_State = true;  // 用户打开 CMDCP;

            LockerState = PassLocker();  // 新用户登录或已退出登录的用户重新登录，需要输入密码更新 PassLocker 状态;

            // 如果用户输入了正确的密码;
            if (LockerState == true) {
                // 只有 clientLogedIP 为空时(即首个用户登入)才进入开发模式，并设置文本框全屏;
                if (clientLogedIP.empty() == true) {
                    Developer_Mode = true;           // 进入开发者模式(此模式下不会显示状态栏，不会显示桌面时钟);
                    oled.setTextBox(0, 0, 128, 64);  // 使控制台文本框全屏显示;
                }

                clientLogedIP.push_back(clientLogingIP);  // 将该用户的 IP 地址标记为已登录并且正在使用还未登出 CMD 用户的 IP;

                /*记录用户的登录时间和 IP 地址*/
                LittleFS.begin();                                                    // 启动闪存文件系统
                File cmdLoggedInfo = LittleFS.open("/CMD_Logged_Info.txt", "a");     // 打开
CMD_Logged_Info.txt 追加日志;
                cmdLoggedInfo.print(clientLogingIP + "-" + timeRef.timeRead(false) + "-login\n");  // IP 地址+时间+登入记
录;
                cmdLoggedInfo.close();
```

```
        }
    }

    if (LockerState == true) {
        saveCmdHistory(CMD, clientLogingIP);  // 保存执行的命令;
        commandIndexer();                     // 已登录用户可使用 CMDCP;
    }

    allow = false;
    return CMDCP_Online_Response;
}

void CMDControlPanelSerialPortal() {
    SerialReceived();  // 获取和更新通过串口发送的命令到 CMD 缓存;

    // 当处于锁定状态并且没有在接收串口数据并且接收到进入 CMDCP 的命令;
    if (LockerState == false && Serial.available() == false && (CMD == "CMD" || CMD == "cmd" || CMD == "login")) {
        LockerState = PassLocker();  // 更新 PassLocker 状态;

        // 如果用户进入了 CMDCP;
        if (LockerState == true) {
            Developer_Mode = true;           // 进入开发者模式(此模式下不会显示状态栏，不会显示桌面时钟);
            oled.setTextBox(0, 0, 128, 64);  // 使控制台文本框全屏显示;
        }
    }

    if (LockerState == true) commandIndexer();  // 处于解锁状态时可使用 CMDCP;
    allow = false;
}

void SerialReceived() {
    CMD = "";                                                          // 清空缓存 CMD 指令;
    while (Serial.available()) CMD += static_cast<char>(Serial.read());  // 获取指令;
}

void commandIndexer() {
    oled.print("> " + CMD);
    Serial.println("> " + CMD);

    // 以空格分割字符串;
    vector<String> CMD_Index = oled.strsplit(CMD, " ");

    // {CMDCP 指令帮助}help
    if (CMD_Index[0] == "help") {
        CMDCP_Response(CMDCP_HELP);
    }

    //{显示当前工作目录(print work directory)}pwd
    if (CMD_Index[0] == "pwd") {
        CMDCP_Response(FFileS.getWorkDirectory());
    }

    // {显示工作目录下的文件列表(List files)}ls
    if (CMD_Index[0] == "ls") {
        String listDirectory = FFileS.listDirectoryContents();
        CMDCP_Response(listDirectory);
    }

    /*
    {切换当前工作目录(Change directory)}cd [dirName]
    [cd ~][cd /] : 切换到 Flash 根目录;
    [cd -] : 返回上一个打开的目录;
    */
    if (CMD_Index[0] == "cd") {
```

```
    if (CMD_Index[1] == "~") {
        FFileS.changeDirectory("/");
    } else if (CMD_Index[1] == "-") {
        FFileS.backDirectory();
    } else {
        // 检查字符串 CMD_Index[1] 的最后一个字符是否是斜杠"/"，如果不是，就在字符串末尾添加一个斜杠。
        if (CMD_Index[1].charAt(CMD_Index[1].length() - 1) != '/') CMD_Index[1] += '/';
        FFileS.changeDirectory(CMD_Index[1]);
    }
    CMDCP_Response("");  // 空响应(该指令无响应内容);
}

// {打开工作目录下的文件(concatenate)}cat [fileName]
if (CMD_Index[0] == "cat") {
    String File_Info = FFileS.readFile(CMD_Index[1]);
    CMDCP_Response(File_Info);
}

// {在工作目录下创建空文件}touch [fileName]
if (CMD_Index[0] == "touch") {
    FFileS.createFile(CMD_Index[1]);
    CMDCP_Response("");  // 空响应(该指令无响应内容);
}

// {在工作目录下新建文件夹(Make Directory)}mkdir [dirName]
if (CMD_Index[0] == "mkdir") {
    FFileS.makeDirector(CMD_Index[1]);
    CMDCP_Response("");  // 空响应(该指令无响应内容);
}

/*
echo [string] : 内容打印到控制台;
echo [string] > [fileName] : 将内容直接覆盖到工作目录的文件中;
echo [string] >> [fileName] : 将内容追加到工作目录的文件中;
*/
if (CMD_Index[0] == "echo") {
    if (CMD_Index[2] == ">") {
        FFileS.fileCover(CMD_Index[1], CMD_Index[3]);
        CMDCP_Response("");  // 空响应(该指令无响应内容);
    } else if (CMD_Index[2] == ">>") {
        FFileS.fileAppend(CMD_Index[1], CMD_Index[3]);
        CMDCP_Response("");  // 空响应(该指令无响应内容);
    } else {
        CMDCP_Response(CMD_Index[1]);
    }
}

/*
{删除一个文件或者目录(Remove)}
rm [fileName] : 删除工作目录下的文件;
rm -r [dirName] : 删除工作目录下的文件夹;
*/
if (CMD_Index[0] == "rm") {
    if (CMD_Index[1] == "-r") {
        FFileS.removeDirector(CMD_Index[2]);
    } else {
        FFileS.removeFile(CMD_Index[1]);
    }
    CMDCP_Response("");  // 空响应(该指令无响应内容);
}

/*
{复制一个文件或目录(Copy file)}cp [-options] [sourcePath] [targetPath];
cp [源文件路径] [目标文件路径] : 复制一个文件到另一个文件;
```

```
    cp -r [源目录路径] [目标目录路径] : 复制一个目录到另一个目录;
    */
    if (CMD_Index[0] == "cp") {
        if (CMD_Index[1] == "-r") {
            FFileS.copyDir(CMD_Index[2], CMD_Index[3]);
        } else {
            FFileS.copyFile(CMD_Index[1], CMD_Index[2]);
        }
        CMDCP_Response("");  // 空响应(该指令无响应内容);
    }

    /*
    {文件或目录改名或将文件或目录移入其它位置(Move)}mv [-options] [sourcePath] [targetPath];
    mv [源文件路径] [目标文件路径] : 移动一个文件到另一个文件;
    mv -r [源目录路径] [目标目录路径] : 移动一个目录到另一个目录;
    */
    if (CMD_Index[0] == "mv") {
        if (CMD_Index[1] == "-r") {
            FFileS.copyDir(CMD_Index[2], CMD_Index[3], true);
        } else {
            FFileS.copyFile(CMD_Index[1], CMD_Index[2], true);
        }
        CMDCP_Response("");  // 空响应(该指令无响应内容);
    }

    /*
    {查找指定目录下的文件(包含子目录的文件)}find [dirPath] [fileName] : 在 dirPath 目录下按文件名查找文件;
    [fileName] = *.* : 查找所有文件;
    [fileName] = *.txt : 查找所有扩展名为 txt 的文件;
    [fileName] = a.txt : 查找 a.txt 文件;
    */
    if (CMD_Index[0] == "find") {
        // 检查字符串 CMD_Index[1] 的最后一个字符是否是斜杠"/", 如果不是, 就在字符串末尾添加一个斜杠。
        if (CMD_Index[1].charAt(CMD_Index[1].length() - 1) != '/') CMD_Index[1] += '/';

        String foundFile = FFileS.findFiles(CMD_Index[1], CMD_Index[2]);
        CMDCP_Response(foundFile);
    }

    //{显示操作系统版本信息}osinfo
    if (CMD_Index[0] == "osinfo") {
        CMDCP_Response(GSG3_Os_Info);

        oled.OLED_DrawBMP(0, 0, 128, 48, GasSensorGen3OS_Info);  // 显示 GasSensorGen3OS_Info;
    }

    //{立刻重新启动 MCU}reboot
    if (CMD_Index[0] == "reboot") {
        digitalWrite(RST, LOW);  // MCU 复位;
        CMDCP_Response("");      // 空响应(该指令无响应内容);
    }

    //{打印主要 GPIO 引脚的状态(Print GPIO status)}pios
    if (CMD_Index[0] == "pios") {
        CMDCP_Response(SysSleep.GPIO_Read());
    }

    //{打印主要系统状态(Print System status)}pss
    if (CMD_Index[0] == "pss") {
        CMDCP_Response(SysSleep.getSysModeAndStatus());
    }

    /*
    {点亮板载的 RGBLED}led [color] [state]
```

```
                for (unsigned int k = length; k < Width_MaxNumChar; ++k) SpaceChar += " ";
                return SpaceChar;
            }(i.length(), Width_MaxNumChar));
        }
    }

    // 计算滚动条高度;
    sliderHeight = static_cast<unsigned char>(8 * Hight_MaxNumChar * (static_cast<float>(Hight_MaxNumChar) /
static_cast<float>(PrintBox.size())));

    // 如果启用自动滚动并且文本行数超过屏幕能够显示的行数则滚动到文本最底部;
    if (autoScroll == true && PrintBox.size() > Hight_MaxNumChar) First_Line = PrintBox.size() - Hight_MaxNumChar;

    drawPrintBox();  // 渲染文本箱;
}

// 清空文本框并释放内存;
void OLED::clearTextBox() {
    PrintBox.clear();
    PrintBox.shrink_to_fit();
    First_Line = 0;  // 首行位置设为 0;
};

// 获取文本框的全部文本;
vector<String> OLED::getPrintBox() { return PrintBox; }

// 用一个新的 PrintBox 结构的数据替换掉现有的 PrintBox(要求保证数据结构正确);
void OLED::replacePrintBox(vector<String> newPrintBox) { PrintBox = newPrintBox; }

// 移动滚动条(true:向下滚动, falas:向上滚动);
void OLED::moveScrollBar(bool direction) {
    if (direction == true && (First_Line + Hight_MaxNumChar) < PrintBox.size()) {
        ++First_Line;
    } else if (direction == false && First_Line > 0) {
        --First_Line;
    }
    drawPrintBox();
    delay(1);
}

// Alert.h-------------------------------------------------------------------------------------------------------
#pragma once
#include "Universal.h"

class ALERT {
   private:
    String alertFlashFile = "/alert_log.txt";  // 被读取的文件位置和名称
   public:
    // 使能或失能蜂鸣器(Time = 使能/失能 时间[ms], SetState = 状态[true 为使能 false 为失能]);
    void BUZZER_Enable(unsigned short Time, bool SetState = true);

    // RGBLED 使能或失能;
    void LED_R_Enable(unsigned short Time, bool SetState = true);
    void LED_G_Enable(unsigned short Time, bool SetState = true);
    void LED_B_Enable(unsigned short Time, bool SetState = true);

    // 关闭所有声光警报;
    void ALERT_Disable();

    // 初始化声光报警器;
    void AlertInit();

    // Flash 写警报日志;
    void flashWriteAlertLog(String alertLog);
```

```
    // Flash 读警报日志;
    void flashReadAlertLog();
};

// Alert.cpp-------------------------------------------------------------------------------------------------------
#include "Alert.h"

void ALERT::BUZZER_Enable(unsigned short Time, bool SetState) {
    if (SetState == true) {
        digitalWrite(0, LOW);
        digitalWrite(2, LOW);
        digitalWrite(15, HIGH);
    } else {
        digitalWrite(0, HIGH);
        digitalWrite(2, HIGH);
        digitalWrite(15, HIGH);
    }
    delay(Time);
}

void ALERT::LED_R_Enable(unsigned short Time, bool SetState) {
    if (SetState == true) {
        digitalWrite(0, LOW);
        digitalWrite(2, HIGH);
        digitalWrite(15, LOW);
    } else {
        digitalWrite(0, HIGH);
        digitalWrite(2, HIGH);
        digitalWrite(15, HIGH);
    }
    delay(Time);
}
void ALERT::LED_G_Enable(unsigned short Time, bool SetState) {
    if (SetState == true) {
        digitalWrite(0, LOW);
        digitalWrite(2, HIGH);
        digitalWrite(15, HIGH);
    } else {
        digitalWrite(0, HIGH);
        digitalWrite(2, HIGH);
        digitalWrite(15, HIGH);
    }
    delay(Time);
}
void ALERT::LED_B_Enable(unsigned short Time, bool SetState) {
    if (SetState == true) {
        digitalWrite(0, HIGH);
        digitalWrite(2, LOW);
        digitalWrite(15, LOW);
    } else {
        digitalWrite(0, HIGH);
        digitalWrite(2, HIGH);
        digitalWrite(15, HIGH);
    }
    delay(Time);
}

void ALERT::ALERT_Disable() {
    digitalWrite(0, HIGH);
    digitalWrite(2, HIGH);
    digitalWrite(15, HIGH);
}
```

```
void ALERT::AlertInit() {
    // 138 译码器输出引脚(控制声光报警);
    pinMode(0, OUTPUT);
    pinMode(2, OUTPUT);
    pinMode(15, OUTPUT);

    // 关闭所有声光警报;
    ALERT_Disable();
}

void ALERT::flashWriteAlertLog(String alertLog) {
    LittleFS.begin();  // 启动 LittleFS;

    File dataFile;
    // 确认闪存中是否有 alertFlashFile 文件
    if (LittleFS.exists(alertFlashFile)) {
        dataFile = LittleFS.open(alertFlashFile, "a");  // 建立 File 对象用于向 LittleFS 中的 file 对象追加信息(添加);
    } else {
        dataFile = LittleFS.open(alertFlashFile, "w");  // 建立 File 对象用于向 LittleFS 中的 file 对象写入信息(新建&覆盖);
    }

    dataFile.print(alertLog);  // 向 dataFile 写入字符串信息
    dataFile.close();          // 完成文件写入后关闭文件
}

void ALERT::flashReadAlertLog() {
    LittleFS.begin();  // 启动 LittleFS;

    File dataFile;
    // 确认闪存中是否有 alertFlashFile 文件
    if (LittleFS.exists(alertFlashFile)) {
        Serial.println("[FLASH FILE FOUND]" + alertFlashFile);

        File dataFile = LittleFS.open(alertFlashFile, "r");  // 建立 File 对象用于从 LittleFS 中读取文件;

        // 读取文件内容并且通过串口监视器输出文件信息
        for (unsigned int i = 0; i < dataFile.size(); ++i) Serial.print((char)dataFile.read());

        dataFile.close();  // 完成文件读取后关闭文件
    } else {
        Serial.println("[FLASH FILE NOT FOUND]" + alertFlashFile);
    }
}

// WebServer.h-----------------------------------------------------------------------------------------------------
#pragma once
#ifndef __WEBSERVER_H
#define __WEBSERVER_H
#include <avr/pgmspace.h>

// 用于打印的操作系统信息;
const char GSG3_Os_Info[] PROGMEM = {
    "GasSensorGen3 OS\nBuild: GS.20230130.Mark0\nUpdate: github.com/RMSHE-MSH\nHardware:
GS.Gen3.20230110.Mark1\nPowered by "
    "RMSHE\nE-mail: asdfghjkl851@outlook.com"};

// CMDControlPanel 帮助信息;
const char CMDCP_HELP[] PROGMEM = {
    "GasSenserOS RMSHE >> CMDControlPanel help"
    "\npwd : Print work directory."
    "\nls : List work directory files."
    "\ncd [dirName] : Change work directory."
    "\ncat [fileName] : Open the file in the work directory."
    "\ntouch [fileName] : Create an empty file in the work directory."
```

```
    "\nmkdir [dirName] : Create a directory under the work directory."
    "\necho [string] : Printed to the CMD."
    "\necho [string] > [fileName] : Overwrite the file in the work directory."
    "\necho [string] >> [fileName] : Append to the file in the work directory."
    "\nrm [fileName] : Remove files in the work directory."
    "\nrm -r [dirName] : Remove the directory under the work directory."
    "\ncp [sourceFilePath] [targetFilePath] : Copy file."
    "\ncp -r [sourceDirPath] [targetDirPath] : Copy directory."
    "\nmv [sourceFilePath] [targetFilePath] : Move file."
    "\nmv -r [sourceDirPath] [targetDirPath] : Move directory."
    "\nfind [dirPath] [fileName] : Find files in the directory, fileName example = (*.*/*.txt/a.txt)."
    "\nosinfo : Display operating system version information."
    "\nreboot : MCU reset."
    "\npios : Print GPIO status."
    "\npss : Print System status."
    "\nled [color] [state] : Turn on RGB LED, color = (r/g/b), state = ((1/true/enable), (0/false/disable))."
    "\nbuzz [state] : Turn on BUZZER, state = ((1/true/enable), (0/false/disable))."
    "\nalertdis : Alert disable."
    "\nfreeze [enable] : Light sleep, enable = ((1/true/enable), (0/false/disable))."
    "\ndisk [time_us] : Deep sleep, time_us = (1 to 4294967295 Microsecond)."
    "\nhistory : Show command history."
    "\nhistory -s : Display commands executed before deep sleep."
    "\nhistory -c : Remove command history."
    "\nwho : View the IP address of the user terminal logged into the current host."
    "\nlast : View system login logs."
    "\nlast -c : Remove system login logs."
    "\ndate : Display system time."
    "\ndate -n : Synchronize network time."
    "\ndate -s [timeStr] : Set system time, timeStr = 20230203121601 (Year Month Day Hour Minute Second)."
    "\nweather : Show current real-time weather."
    "\nweather -n : Synchronize live weather on the web."
    "\nweather -s [cityID] : To set the city, fill in the \"cityID\" with the city ID of Know Your Weather."
    "\npgup : Text box scrolls up one line."
    "\npgup -s [line] : The text box is scrolled up, \"line\" is the number of lines to scroll."
    "\npgdn : Text box scrolls down."
    "\npgdn -s [line] : The text box is scrolled down, \"line\" is the number of lines to scroll."
    "\nclear : Clear console and free memory."
    "\nupload : Uploading files from the terminal to the server."
    "\nupload -s : View the results of the last file upload."
    "\ndf : Display Flash information."
    "\nfree : Display remaining RAM."
    "\nwifi [SSID] [PASSWORD] : Configure WIFI connection, set WIFI SSID and PASSWORD."
    "\npoweroff : Indefinite deep sleep."
    "\nlogout : Log out and lock CMDCP."
    "\nlogout -k [clientIP] : Logout of the terminals with the specified IP address."
    "\nlogout -k other : Logout of other terminals except yourself."
    "\nlogout -k all : logout of all terminals"};

const char CMDCP_Online[] PROGMEM = R"rawliteral(

<!DOCTYPE html>
<html>

<head>
    <title>GasSenserOS Web Server</title>
    <style>
        body {
            background-color: #343541;
            color: #ececf1;
            font-family: Arial, sans-serif;
            font-size: 16px;
            line-height: 1.5;
            margin: 0;
            padding: 0;
```

```
    transition: all 0.24s ease-in-out;
}

#console {
    position: fixed;
    top: 30px;
    left: 0;
    right: 0;

    float: left;
    overflow-y: scroll;
    overflow-x: auto;
    width: calc(90vw - 40px);
    height: calc(100vh - 194px);
    border: 0px solid rgba(255, 255, 255, 0);
    padding: 10px;
    margin: 10px;
    background-color: rgba(0, 0, 0, 0.7);
    color: #ececf1;
    border-radius: 6px;
    font-size: 20px;
    scrollbar-width: none;
    resize: none;
    position: relative;
    box-shadow: 0px -12px 0px #202123;
    left: 50%;
    transform: translateX(-50%);
    transition: all 0.24s ease-in-out;
}

.text-RMSHE {
    position: absolute;
    font-size: 80px;
    color: #40414f00;
    filter: blur(0px);
    align-items: center;

    left: 50%;
    bottom: 40px;
    transform: translateX(-50%);
    transition: all 0.5s ease-in-out;
}


#console:focus {
    background-color: #000000;
    outline: none;
}

#console::selection {
    background-color: #33333300;
    color: #79b8ff;
}

#console::-webkit-scrollbar {
    width: 8px;
    background-color: #1f1f1f00;
    position: absolute;
    right: 0;
    top: 200;
    border-radius: 20px;
}

#console::-webkit-scrollbar-thumb {
```

```
        border-radius: 10px;
        background-color: #565869;
        border: 0px solid #1f1f1f;
    }

    #container {
        position: absolute;
        bottom: 30px;
        float: left;
        width: calc(90vw - 40px);
        height: 60px;
        margin: 10px;
        padding: 10px;
        background-color: #40414f00;
        color: #ececf1;
        border-radius: 10px;
        resize: auto;
        left: 50%;
        transform: translateX(-50%);
        display: flex;
        align-items: center;
        transition: all 0.24s ease-in-out;

    }

    #message {
        width: calc(100% - 139px);
        height: 60px;
        background-color: #40414f;
        color: #ececf1;
        border: 0px solid #FFF;
        border-radius: 10px;
        resize: auto;
        font-size: 24px;
        text-align: auto;
        padding-left: 18px;
        box-shadow: 0px 0px 6px #303139;
        transition: all 0.2s ease-in-out;
    }

    #sendButton,
    #uploadButton {
        margin-left: 16px;
        width: 90px;
        height: 60px;
        background-color: #40414f;
        color: #acacbe;
        border: none;
        font-size: 24px;
        border-radius: 10px;
        box-shadow: 0px 0px 6px #303139;
        transition: all 0.2s ease-in-out;
    }

    #uploadButton {
        width: 90px;
        height: 60px;
        padding: 10px;
        box-shadow: 0px 0px 6px #303139;
        transition: all 0.2s ease-in-out;
    }

    #message:focus {
        background-color: #444654;
```

```
            outline: none;
            box-shadow: 0px 0px 20px #303139;
        }

        #sendButton:active,
        #uploadButton:active {
            background-color: #40414f;
            box-shadow: 0px 0px 10px #202123;
            border: 2px solid #565869;
            filter: blur(2px);
            font-size: 22px;
        }

        #sendButton:hover,
        #uploadButton:hover {
            background-color: #202123;
        }

        .text-info {
            font-family: Arial, sans-serif;
            font-weight: bold;
            position: absolute;
            bottom: 4px;
            font-size: 12px;
            color: rgba(153, 153, 161, 100);
            text-align: center;
            left: 50%;
            transform: translateX(-50%);
            filter: blur(0px);
            transition: all 0.5s ease-in-out;
            user-select: none;
            white-space: nowrap;
        }
    </style>
    <script>
        function updateMessage() {
            //服务器响应;
            var xhttp = new XMLHttpRequest();
            xhttp.onreadystatechange = function () {
                if (this.readyState == 4 && this.status == 200 && this.responseText != "") {

                    let input = this.responseText;
                    let text = input.split("\n");//对响应字符串进行按行分割;
                    let index = 0;

                    //创建定时器(每隔 8ms)使字符串逐行出现;
                    const interval = setInterval(function () {
                        document.getElementById("console").value += "\n" + text[index];//逐行输出;
                        document.getElementById("console").scrollTop =
document.getElementById("console").scrollHeight;//滚动到最底部;
                        index++;

                        //如果全部输出完成则停止定时器;
                        if (index === text.length) {
                            clearInterval(interval);
                        }
                    }, 8);


                    //document.getElementById("console").value += "\n" + this.responseText;
                    //document.getElementById("console").scrollTop = document.getElementById("console").scrollHeight;

                    if (this.responseText == "EnableUpload") {
                        var fileInput = document.getElementById("fileInput");
```

```
                var file = fileInput.files[0];
                var formData = new FormData();
                formData.append("file", file);

                var xhr = new XMLHttpRequest();
                xhr.open("POST", "http://192.168.43.164:80/upload", true);
                xhr.send(formData);

                //等待 1000ms;
                setTimeout(function () {
                    document.getElementById("fileInput").value = "";//清空文件选择器;

                    //向服务器发送指令查看文件是否上传成功;
                    xhttp.open("GET", "http://192.168.43.164:80/CMD?message=upload -s", true);
                    xhttp.send();
                }, 1000);
            }
        }
    };

    //向服务器请求;
    var message = document.getElementById("message").value;

    document.getElementById("console").value += "\n> " + message;
    document.getElementById("console").scrollTop = document.getElementById("console").scrollHeight;
    xhttp.open("GET", "http://192.168.43.164:80/CMD?message=" + message, true);
    xhttp.send();

    if (message == "clear") document.getElementById("console").value = "GasSenserOS RMSHE >> CMDControlPanel";

    document.getElementById("message").value = "";
}

window.onload = function () {
    //将 uploadButton 按钮与隐藏的"选择文件"控件绑定;
    document.getElementById("uploadButton").addEventListener("click", function (event) {
        event.preventDefault();
        document.getElementById("fileInput").click();
    });

    //监测文件选择器是否有文件，如果有文件则将按钮文本修改为"Load";
    var fileInput = document.getElementById("fileInput");
    document.querySelector("Input[type=file]").addEventListener("change", function (e) {
        if (e.target.files.length) {
            uploadButton.innerHTML = "Load";
            document.getElementById("message").value = "upload";
        } else {
            uploadButton.innerHTML = "File";
        }
    });

    /*
    //当输入框或编辑框获得焦点时将背景 RMSHE 虚化;
    document.getElementById("message").addEventListener("focus", function (event) {
        document.querySelector(".text-RMSHE").style.filter = "blur(16px)";
    });
    document.getElementById("console").addEventListener("focus", function (event) {
        document.querySelector(".text-RMSHE").style.filter = "blur(16px)";
    });
    //当输入框或编辑框获得焦点时将背景 RMSHE 实体化;
    document.getElementById("message").addEventListener("blur", function (event) {
        document.querySelector(".text-RMSHE").style.filter = "blur(0)";
    });
    document.getElementById("console").addEventListener("blur", function (event) {
```

```
            document.querySelector(".text-RMSHE").style.filter = "blur(0)";
        });
        */
    }

    //浏览器窗口大小改变时将文本虚化，清除任何现有的计时器并重新设置一个新的计时器。如果 500 毫秒内没有更改窗口大小，则将触发计时器回调函数并将文本实体化。
    var resizeTimer;
    window.addEventListener("resize", function (event) {
        //document.querySelector(".text-RMSHE").style.filter = "blur(30px)";

        document.querySelector(".text-info").style.filter = "blur(30px)";
        document.querySelector(".text-info").style.color = "rgba(153, 153, 161, 0)";
        document.querySelector(".text-info").style.fontSize = "0px";
        clearTimeout(resizeTimer);
        resizeTimer = setTimeout(function () {
            //document.querySelector(".text-RMSHE").style.filter = "blur(0)";

            document.querySelector(".text-info").style.filter = "blur(0)";
            document.querySelector(".text-info").style.color = "rgba(153, 153, 161, 100)";
            document.querySelector(".text-info").style.fontSize = "12px";
        }, 500);
    });



    // 调整字体大小的函数
    window.onresize = function () {
        var textarea = document.getElementById("console");
        var InputBox = document.getElementById("message");
        var sendButton = document.getElementById("sendButton");
        var uploadButton = document.getElementById("uploadButton");
        var container = document.getElementById("container");
        var fileInput = document.getElementById("fileInput");

        var width_size = window.innerWidth / 50;
        var height_size = window.innerHeight / 10;
        var Button_width_size = window.innerWidth / 15;

        //textarea 编辑框字体动态调整;
        if (width_size < 12) {
            textarea.style.fontSize = "12px";
        } else if (width_size > 20) {
            textarea.style.fontSize = "20px";
        } else {
            textarea.style.fontSize = width_size + "px";
        }

        //message 输入框字体动态调整;
        if (width_size < 16) {
            InputBox.style.fontSize = "16px";
        } else if (width_size > 24) {
            InputBox.style.fontSize = "24px";
        } else {
            InputBox.style.fontSize = width_size + "px";
        }

        //sendButton 发送按钮字体动态调整;
        if (width_size < 20) {
            sendButton.style.fontSize = "20px";
        } else if (width_size > 24) {
            sendButton.style.fontSize = "24px";
        } else {
            sendButton.style.fontSize = width_size + "px";
```

```
            }

            //sendButton 发送按钮宽度动态调整;
            if (Button_width_size < 40) {
                sendButton.style.width = "40px";
            } else if (Button_width_size > 90) {
                sendButton.style.width = "90px";
            } else {
                sendButton.style.width = Button_width_size + "px";
            }

            if (Button_width_size < 65) {
                sendButton.innerHTML = "S";//修改发送按钮文本内容为缩写;
            } else {
                sendButton.innerHTML = "Sead";//修改发送按钮文本内容为全拼;
            }

            //uploadButton 文件选择按钮字体动态调整;
            if (width_size < 20) {
                uploadButton.style.fontSize = "20px";
            } else if (width_size > 24) {
                uploadButton.style.fontSize = "24px";
            } else {
                uploadButton.style.fontSize = width_size + "px";
            }

            //uploadButton 文件选择按钮宽度动态调整;
            if (Button_width_size < 40) {
                uploadButton.style.width = "40px";
            } else if (Button_width_size > 90) {
                uploadButton.style.width = "90px";
            } else {
                uploadButton.style.width = Button_width_size + "px";
            }

            if (Button_width_size < 65) {
                //修改选择文件按钮文本内容为缩写;
                if (fileInput.value != "") { uploadButton.innerHTML = "L"; } else { uploadButton.innerHTML = "F"; }
            } else {
                //修改选择文件按钮文本内容为全拼;
                if (fileInput.value != "") { uploadButton.innerHTML = "Load"; } else { uploadButton.innerHTML =
"File"; }
            }

            //container 输入栏高度动态调整;
            if (height_size < 40) {
                container.style.height = "40px";
                InputBox.style.height = "40px";
                sendButton.style.height = "40px";
                uploadButton.style.height = "40px";
            } else if (height_size > 60) {
                container.style.height = "60px";
                InputBox.style.height = "60px";
                sendButton.style.height = "60px";
                uploadButton.style.height = "60px";
            } else {
                container.style.height = height_size + "px";
                InputBox.style.height = height_size + "px";
                sendButton.style.height = height_size + "px";
                uploadButton.style.height = height_size + "px";
            }
        };

    </script>
```

```
</head>

<body>
    <div class="console-RMSHE">
        <p class="text-RMSHE">RMSHE</p>

        <textarea id="console">GasSenserOS RMSHE >> CMDControlPanel</textarea><br>
    </div>

    <div id="container">
        <input type="text" id="message" onkeydown="if (event.key === 'Enter') { updateMessage(); }">
        <button id="sendButton" onclick="updateMessage()">Send</button><br>

        <form method="POST" enctype="multipart/form-data">
            <input type="file" id="fileInput" style="display:none">
            <button id="uploadButton">File</button>
        </form>
    </div>

    <p class="text-info">GasSenserOS CMDControlPanel Online. Powered by RMSHE and ChatGPT.</p>

</body>

</html>

)rawliteral";

#endif

/*

<!DOCTYPE html>
<html>

<head>
    <title>GasSenserOS Web Server</title>
    <style>
        body {
            background-color: #343541;
            color: #ececf1;
            font-family: Arial, sans-serif;
            font-size: 16px;
            line-height: 1.5;
            margin: 0;
            padding: 0;
        }

        #console {
            position: fixed;
            top: 20px;

            float: left;
            overflow-y: scroll;
            overflow-x: auto;
            width: 91.3%;
            height: 500px;
            border: 0px solid rgba(255, 255, 255, 0);
            padding: 10px;
            margin: 10px;
            background-color: #000000c7;
            color: #ececf1;
            border-radius: 6px;
            scrollbar-width: none;
            resize: none;
```

```
    position: relative;
    font-size: 16px;
    box-shadow: 0px -10px 0px #202123;
}

#console:focus {
    background-color: #000000;
    outline: none;
}

#console::selection {
    background-color: #33333300;
    color: #79b8ff;
}

#console::-webkit-scrollbar {
    width: 8px;
    background-color: #1f1f1f00;
    position: absolute;
    right: 0;
    top: 200;
    border-radius: 20px;
}

#console::-webkit-scrollbar-thumb {
    border-radius: 10px;
    background-color: #565869;
    border: 0px solid #1f1f1f;
}

#message {
    position: relative;
    top: 20px;

    float: left;
    width: 75%;
    height: 40px;
    margin: 10px;
    border: 0px solid #FFF;
    padding: 10px;
    background-color: #40414f;
    color: #ececf1;
    font-size: 24px;
    border-radius: 10px;
    resize: auto;
    box-shadow: 0px 0px 6px #303139;
}

#message:focus {
    background-color: #444654;
    outline: none;
}

button {
    position: relative;
    top: 20px;

    float: left;
    width: 59px;
    height: 59px;
    background-color: #acacbe;
    color: #343541;
    border: none;
    margin: 10px 0 10px 10px;
```

```
            font-size: 24px;
            border-radius: 10px;
            box-shadow: 0px 0px 6px #303139;
        }
    </style>
    <script>
        function updateMessage() {
            var xhttp = new XMLHttpRequest();
            xhttp.onreadystatechange = function () {
                if (this.readyState == 4 && this.status == 200 && this.responseText != "") {
                    document.getElementById("console").value += "\n" + this.responseText;
                    document.getElementById("console").scrollTop = document.getElementById("console").scrollHeight;
                }
            };
            var message = document.getElementById("message").value;
            document.getElementById("console").value += "\n> " + message;
            document.getElementById("console").scrollTop = document.getElementById("console").scrollHeight;
            xhttp.open("GET", "http://192.168.31.175:80/CMD?message=" + message, true);
            xhttp.send();
            document.getElementById("message").value = "";
        }
    </script>
</head>

<body>
    <textarea id="console">GasSenserOS RMSHE >> CMDControlPanel</textarea><br>
    <input type="text" id="message" onkeydown="if (event.key === 'Enter') { updateMessage(); }">
    <button onclick="updateMessage()">>>></button><br>
</body>

</html>

*/

// WeatherNow.h-----------------------------------------------------------------------------------------------------
#pragma once
#ifndef _WEATHERNOW_H_
#define _WEATHERNOW_H_

#include <Arduino.h>
#include <ArduinoJson.h>
#include <ESP8266WiFi.h>

// #define DEBUG  // 调试用宏定义

// 获取当前天气信息类
class WeatherNow {
  public:
    void config(String userKey, String location, String unit);
    bool update();

    String getCityID();  // 返回当前城市（字符串格式）

    String getCityName();  // 返回当前城市名称（字符串格式）

    String getCountry();  // 返回当前城市国家（字符串格式）

    String getPath();  // 返回当前城市路径（字符串格式）

    String getTimezone();  // 返回当前城市时区名称（字符串格式）

    String getTimezoneOffset();  // 返回当前城市时区（字符串格式）

    String getWeatherText();  // 返回当前天气信息（字符串格式）
```

```
    int getWeatherCode();  // 返回当前天气信息（整数格式）

    int getTemperature();  // 返回当前气温;

    String getLastUpdate();  // 返回心知天气信息更新时间;

    String getServerCode();  // 返回服务器响应状态码;

   private:
    const char* _host = "api.seniverse.com";  // 服务器地址

    String _reqUserKey;   // 私钥
    String _reqLocation;  // 城市
    String _reqUnit;      // 摄氏/华氏

    void _parseNowInfo(WiFiClient client);  // 解析实时天气信息信息

    String _status_response = "no_init";  // 服务器响应状态行
    String _response_code = "no_init";    // 服务器响应状态码

    String _now_id_str = "no_init";
    String _now_name_str = "no_init";
    String _now_country_str = "no_init";
    String _now_path_str = "no_init";
    String _now_timezone_str = "no_init";
    String _now_timezone_offset_str = "no_init";

    String _now_text_str = "no_init";
    int _now_code_int = -1;
    int _now_temperature_int = -127;
    String _last_update_str = "no_init";
};
#endif

// WeatherNow.cpp-------------------------------------------------------------------------------------------------
#include "WeatherNow.h"

/* 配置心知天气请求信息
 * @param userKey  用户心知天气私钥
 * @param location 获取信息的城市参数
 * @param location 获取信息的温度单位(摄氏/华氏)
 */
void WeatherNow::config(String userKey, String location, String unit) {
    _reqUserKey = userKey;
    _reqLocation = location;
    _reqUnit = unit;
}

/* 尝试从心知天气更新信息
 * @return: bool 成功更新返回真，否则返回假
 */
bool WeatherNow::update() {
    WiFiClient _wifiClient;

    String reqRes = "/v3/weather/now.json?key=" + _reqUserKey + +"&location=" + _reqLocation + "&language=en&unit=" +
_reqUnit;

    String httpRequest = String("GET ") + reqRes + " HTTP/1.1\r\n" + "Host: " + _host + "\r\n" + "Connection:
close\r\n\r\n";

#ifdef DEBUG
    Serial.print("Connecting to ");
    Serial.print(_host);
```

```
#endif DEBUG

    if (_wifiClient.connect(_host, 80)) {
#ifdef DEBUG
        Serial.println(" Success!");
#endif DEBUG

        // 向服务器发送 http 请求信息
        _wifiClient.print(httpRequest);

#ifdef DEBUG
        Serial.println("Sending request: ");
        Serial.println(httpRequest);
#endif DEBUG

        // 获取并显示服务器响应状态行
        String _status_response = _wifiClient.readStringUntil('\n');
#ifdef DEBUG
        Serial.print("_status_response: ");
        Serial.println(_status_response);
#endif DEBUG

        // 查验服务器是否响应 200 OK
        _response_code = _status_response.substring(9, 12);
        if (_response_code == "200") {
#ifdef DEBUG
            Serial.println("Response Code: 200");
#endif DEBUG
        } else {
#ifdef DEBUG
            Serial.println(F("Response Code: NOT 200"));
#endif DEBUG
            _wifiClient.stop();
            return false;
        }

        // 使用 find 跳过 HTTP 响应头
        if (_wifiClient.find("\r\n\r\n")) {
#ifdef DEBUG
            Serial.println("Found Header End. Start Parsing.");
#endif DEBUG
        }

        _parseNowInfo(_wifiClient);
        _wifiClient.stop();
        return true;
    } else {
#ifdef DEBUG
        Serial.println(" connection failed!");
#endif DEBUG
        _wifiClient.stop();
        return false;
    }
}

// 配置心知天气请求信息
void WeatherNow::_parseNowInfo(WiFiClient httpClient) {
    const size_t capacity = JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(1) + 2 * JSON_OBJECT_SIZE(3) + JSON_OBJECT_SIZE(6) +
230;
    DynamicJsonDocument doc(capacity);

    deserializeJson(doc, httpClient);

    JsonObject results_0 = doc["results"][0];
```

```
    JsonObject results_0_location = results_0["location"];
    JsonObject results_0_now = results_0["now"];

    // 通过串口监视器显示以上信息
    _now_id_str = results_0_location["id"].as<String>();
    _now_name_str = results_0_location["name"].as<String>();
    _now_country_str = results_0_location["country"].as<String>();
    _now_path_str = results_0_location["path"].as<String>();
    _now_timezone_str = results_0_location["timezone"].as<String>();
    _now_timezone_offset_str = results_0_location["timezone_offset"].as<String>();

    _now_text_str = results_0_now["text"].as<String>();
    _now_code_int = results_0_now["code"].as<int>();
    _now_temperature_int = results_0_now["temperature"].as<int>();

    _last_update_str = results_0["last_update"].as<String>();
}

// 返回当前城市（字符串格式）
String WeatherNow::getCityID() { return _now_id_str; }

// 返回当前城市名称（字符串格式）
String WeatherNow::getCityName() { return _now_name_str; }

// 返回当前城市国家（字符串格式）
String WeatherNow::getCountry() { return _now_country_str; }

// 返回当前城市路径（字符串格式）
String WeatherNow::getPath() { return _now_path_str; }

// 返回当前城市时区名称（字符串格式）
String WeatherNow::getTimezone() { return _now_timezone_str; }

// 返回当前城市时区（字符串格式）
String WeatherNow::getTimezoneOffset() { return _now_timezone_offset_str; }

// 返回当前天气信息（字符串格式）
String WeatherNow::getWeatherText() { return _now_text_str; }

// 返回当前天气信息（整数格式）
int WeatherNow::getWeatherCode() { return _now_code_int; }

// 返回当前气温
int WeatherNow::getTemperature() { return _now_temperature_int; }

// 返回心知天气信息更新时间
String WeatherNow::getLastUpdate() { return _last_update_str; }

// 返回服务器响应状态码
String WeatherNow::getServerCode() { return _response_code; }

// Tool.h-----------------------------------------------------------------------------------------------------------
#pragma once
#include "Universal.h"

class TOOL {
   public:
    int findArrMax(int arr[], int n);
    int findArrMin(int arr[], int n);

    String XOR_encrypt(String plaintext, String key);
};
```

```
// Tool.cpp-------------------------------------------------------------------------------------------------------
#pragma once
#include "Tool.h"

// 查找数组中的最大值(目标数组, 数组长度);
int TOOL::findArrMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

// 查找数组中的最小值(目标数组, 数组长度);
int TOOL::findArrMin(int arr[], int n) {
    int min = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < min) {
            min = arr[i];
        }
    }
    return min;
}

String TOOL::XOR_encrypt(String plaintext, String key) {
    String ciphertext = "";
    unsigned char j = 0;
    for (char c : plaintext) {
        ciphertext += char(c ^ key[j]);
        j = (j + 1) % key.length();
    }
    return ciphertext;
}

// Hash.h---------------------------------------------------------------------------------------------------------
#pragma once
#include <Arduino.h>
#include <bearssl/bearssl_hash.h>

#ifndef HASH_H_
#define HASH_H_

// #define DEBUG_SHA1

void sha1(const uint8_t* data, uint32_t size, uint8_t hash[20]);
void sha1(const char* data, uint32_t size, uint8_t hash[20]);
void sha1(const String& data, uint8_t hash[20]);

String sha1(const uint8_t* data, uint32_t size);
String sha1(const char* data, uint32_t size);
String sha1(const String& data);

#endif /* HASH_H_ */

// Hash.cpp-------------------------------------------------------------------------------------------------------
#include "Hash.h"

/**
 * create a sha1 hash from data
 * @param data uint8_t *
 * @param size uint32_t
 * @param hash uint8_t[20]
```

```
 */
void sha1(const uint8_t* data, uint32_t size, uint8_t hash[20]) {
    br_sha1_context ctx;

#ifdef DEBUG_SHA1
    os_printf("DATA:");
    for (uint16_t i = 0; i < size; i++) {
        os_printf("%02X", data[i]);
    }
    os_printf("\n");
    os_printf("DATA:");
    for (uint16_t i = 0; i < size; i++) {
        os_printf("%c", data[i]);
    }
    os_printf("\n");
#endif

    br_sha1_init(&ctx);
    br_sha1_update(&ctx, data, size);
    br_sha1_out(&ctx, hash);

#ifdef DEBUG_SHA1
    os_printf("SHA1:");
    for (uint16_t i = 0; i < 20; i++) {
        os_printf("%02X", hash[i]);
    }
    os_printf("\n\n");
#endif
}

void sha1(const char* data, uint32_t size, uint8_t hash[20]) { sha1((const uint8_t*)data, size, hash); }

void sha1(const String& data, uint8_t hash[20]) { sha1(data.c_str(), data.length(), hash); }

String sha1(const uint8_t* data, uint32_t size) {
    uint8_t hash[20];
    String hashStr((const char*)nullptr);
    hashStr.reserve(20 * 2 + 1);

    sha1(&data[0], size, &hash[0]);

    for (uint16_t i = 0; i < 20; i++) {
        char hex[3];
        snprintf(hex, sizeof(hex), "%02x", hash[i]);
        hashStr += hex;
    }

    return hashStr;
}

String sha1(const char* data, uint32_t size) { return sha1((const uint8_t*)data, size); }

String sha1(const String& data) { return sha1(data.c_str(), data.length()); }

// Universal.h-----------------------------------------------------------------------------------------------
#pragma once
#include <Arduino.h>
#include <ArduinoJson.h>
#include <ESP8266HTTPClient.h>
#include <ESP8266WebServer.h>
#include <ESP8266WiFi.h>
#include <Ticker.h>
#include <Wire.h>
```

```
#include <vector>

#include "LittleFS.h"
#include "oledfont.h"

#define UINT8_MIN 0
#define UINT16_MIN 0
#define UINT32_MIN 0
#define UINT64_MIN 0

#define UINT8_MAX 255
#define UINT16_MAX 65535
#define UINT32_MAX 4294967295
#define UINT64_MAX 18446744073709551615

#define DeepSleep_MAX 4294967295

// 复位引脚定义
#define RST 16

// 模拟引脚定义;
#define SENANALOG A0

// 74HC138 译码器输出引脚定义;
#define Decoder_C 0
#define Decoder_B 2
#define Decoder_A 15

// I2C 管脚的定义;
#define SDA 4
#define SCL 5

// 串口管脚定义;
#define TXD 1
#define RXD 3

// 电池状态引脚定义;
#define CHRG 14

// 电池低电量引脚定义;
#define LOWPOWER 12

// 传感器输出引脚定义;
#define SENOUT 13

// WIFI 信息;
#define ServerPort 80
#define SSID "RMSHE"
#define PASSWORD "GAATTC-A23187"

/* 设备的三元组信息*/
#define PRODUCT_KEY "i6abR7NBjfB"
#define DEVICE_NAME "GasSensor_OS_ESP8266"
#define DEVICE_SECRET "6269beb7dd1c4f92a29560441970f9de"
#define REGION_ID "cn-shanghai"

/* 线上环境域名和端口号，不需要改 */
#define MQTT_SERVER PRODUCT_KEY ".iot-as-mqtt." REGION_ID ".aliyuncs.com"
#define MQTT_PORT 1883

#define MQTT_CLIENT_ID "i6abR7NBjfB.GasSensor_OS_ESP8266|securemode=2,signmethod=hmacsha256,timestamp=1675522358253|"

// 算法工具: http://iot-face.oss-cn-shanghai.aliyuncs.com/tools.htm 进行加密生成 password
// password 教程 https://www.yuque.com/cloud-dev/iot-tech/mebm5g
```

```
#define MQTT_USRNAME DEVICE_NAME "&" PRODUCT_KEY
#define MQTT_PASSWD "01a7128c6d546845b23f4c0521355c96e285b360a469be8f906371ac3a5e9d01"

// 连接超时时间;
#define TimeOut 5000  // ms;

// 授时网站;
#define GetSysTimeUrl http://quan.suning.com/getSysTime.do

// rivest_cipher_4.hpp-------------------------------------------------------------------------------------------------
/**
 * @file rivest_cipher_4.hpp
 * @date 05.04.2023
 */
#pragma once
#include <array>
#include <cstring>
#include <iomanip>
#include <vector>

class RivestCipher4 {
   public:
    // 构造函数，接收一个字符串 key 作为参数
    explicit RivestCipher4(const std::string &key) {
        // 初始化 S 盒
        for (uint16_t i = 0; i < 256; ++i) S_box_[i] = static_cast<uint8_t>(i);

        // 对 S 盒进行置换
        uint32_t j = 0;
        for (uint16_t i = 0; i < 256; ++i) {
            // 用循环的方式扩展 key
            j = (j + S_box_[i] + key[i % key.size()]) & 0xFF;
            uint8_t tmp = S_box_[i];
            S_box_[i] = S_box_[j];
            S_box_[j] = tmp;
        }
    }

    // 加密函数，接收一个字符串 plaintext 作为参数
    std::string encrypt(const std::string &plaintext) {
        // 初始化一个输出字符串流
        std::ostringstream ciphertext_stream;
        // 使用十六进制输出，并用 0 填充未满两位的字节
        ciphertext_stream << std::hex << std::setfill('0');

        // 初始化两个状态变量 i 和 j
        uint8_t i = 0, j = 0;
        // 遍历明文字符串中的每个字节
        for (size_t index = 0; index < plaintext.size(); ++index) {
            // 更新状态变量 i 和 j
            i = (i + 1) & 0xFF;
            j = (j + S_box_[i]) & 0xFF;
            // 交换 S 盒中的两个元素
            std::swap(S_box_[i], S_box_[j]);
            // 计算 k 和 temp
            uint8_t t = (S_box_[i] + S_box_[j]) & 0xFF;
            uint8_t k = S_box_[t];
            uint8_t temp = static_cast<uint8_t>(plaintext[index]) ^ k;

            // 将加密后的字节写入输出流
            ciphertext_stream << std::setw(2) << static_cast<uint16_t>(temp);
        }

        // 返回加密后的字符串
```

```
        return ciphertext_stream.str();
    }

    // 解密函数，调用加密函数实现
    std::string decrypt(const std::string &ciphertext) { return encrypt(ciphertext); }

   private:
    // S 盒，使用 std::array 存储
    std::array<uint8_t, 256> S_box_;
};

// make_ptr.hpp-------------------------------------------------------------------------------------------------
#pragma once
#include <memory>

/*
用于创建一个智能指针对象：
这段代码定义了一个模板函数 make_unique，用于创建并返回一个智能指针 unique_ptr，其中模板参数 T 代表指向的类型，参数 Args 代表构造函数参数类型的列表。

函数的实现部分首先创建一个 unique_ptr，指向一个通过 new 运算符创建的类型为 T 的对象，并传递构造函数所需的参数列表。 std::forward 用于将 args
转发给构造函数，确保构造函数接收到正确类型的参数。

由于返回的是 unique_ptr，因此可以确保指针所有权的唯一性，避免内存泄漏的问题.
*/
template <typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

template <typename T, typename... Args>
std::shared_ptr<T> make_shared(Args&&... args) {
    return std::shared_ptr<T>(new T(std::forward<Args>(args)...));
}

// fourier_transform.hpp----------------------------------------------------------------------------------------
/**
 * @file fourier_transform.hpp
 * @date 05.04.2023
 * @author RMSHE
 */

#pragma once
#include <algorithm>
#include <cmath>
#include <complex>
#include <vector>

class FastFourierTransform {
   public:
    /**
     * @brief 快速傅里叶变换
     * @param input_sequence std::vector<float>类型实数域数据
     * @param inverse false 为正变换，true 为反变换(默认为正变换)
     */
    std::vector<std::complex<float>> FFT(std::vector<float> input_sequence, bool inverse = false) {
        /*
         * 首先，n & (n - 1) 的结果为 0，当且仅当 n 是 2 的幂次方。如果 n 是 2 的幂次方，则 m 直接取 n，不需要进行补零或截断操作。
         * 否则，m 需要取离 n 最近的较小的 2 的幂次方或较大的 2 的幂次方。
         * 使用 builtin_clz 函数来计算 n 的二进制表示中前导零的个数，然后通过移位运算来得到最接近的 2 的幂次方。
         * 最后，使用三目运算符来判断是否需要进行补零或截断操作，直接 resize 到 m 即可。
         */
        uint32_t n = input_sequence.size();
```

```
        uint32_t m = n & (n - 1) ? 1 << (32 - CountLeadingZeros(n)) : n;
        input_sequence.resize(std::min(m, n));

        // 这里将实数域转为复数域;
        std::vector<std::complex<float>> data(input_sequence.begin(), input_sequence.end());

        // 若反转为 false 则进行 FFT 正变换，若反转为 true 则进行 FFT 逆变换;
        inverse == false ? fast_fourier_transform(data) : inverse_fast_fourier_transform(data);

        return data;
    }

  private:
    // 计算一个 32 位无符号整数的二进制表示中前导零的个数;
    static uint32_t CountLeadingZeros(uint32_t x) {
        uint32_t n = 0;
        if (x == 0) return 32;
        if (x <= 0x0000FFFF) n += 16, x <<= 16;
        if (x <= 0x00FFFFFF) n += 8, x <<= 8;
        if (x <= 0x0FFFFFFF) n += 4, x <<= 4;
        if (x <= 0x3FFFFFFF) n += 2, x <<= 2;
        if (x <= 0x7FFFFFFF) n += 1;
        return n;
    }

    // 快速傅里叶变换
    static void fast_fourier_transform(std::vector<std::complex<float>> &input_sequence) {
        const uint32_t sequence_size = input_sequence.size();

        // 如果序列长度小于等于 1，则不需要继续计算
        if (sequence_size <= 1) return;

        // 缓存计算结果，避免重复计算
        std::vector<std::complex<float>> even_sequence(sequence_size / 2), odd_sequence(sequence_size / 2);
        for (uint32_t i = 0; i < sequence_size / 2; ++i) {
            even_sequence[i] = input_sequence[2 * i];     // 偶数序列
            odd_sequence[i] = input_sequence[2 * i + 1];  // 奇数序列
        }

        // 递归计算偶数和奇数序列的 FFT
        fast_fourier_transform(even_sequence);
        fast_fourier_transform(odd_sequence);

        // 合并偶数和奇数序列的 FFT 结果
        const auto twiddle_factor = std::polar(1.0, -2 * PI / sequence_size);
        std::complex<float> factor(1, 0);
        for (uint32_t i = 0; i < sequence_size / 2; ++i) {
            input_sequence[i] = even_sequence[i] + factor * odd_sequence[i];
            input_sequence[i + sequence_size / 2] = even_sequence[i] - factor * odd_sequence[i];
            factor *= twiddle_factor;  // 更新旋转因子
        }
    }

    // 快速傅里叶逆变换
    static void inverse_fast_fourier_transform(std::vector<std::complex<float>> &input_sequence) {
        // 先计算 FFT，然后对结果取共轭并除以 n 即可得到逆变换的结果
        fast_fourier_transform(input_sequence);
        const uint32_t sequence_size = input_sequence.size();
        for (auto &element : input_sequence) {
            element = std::conj(element) * static_cast<float>(1.0 / sequence_size);  // 取共轭并除以 n
        }
    }
};
```

```
class DiscreteFourierTransform {
   public:
    std::vector<std::complex<float>> DFT(std::vector<float> input_sequence, bool inverse = false) {
        // 这里将实数域转为复数域;
        std::vector<std::complex<float>> data(input_sequence.begin(), input_sequence.end());

        // 若反转为 false 则进行 DFT 正变换，若反转为 true 则进行 DFT 逆变换;
        if (inverse == false)
            return discrete_fourier_transform(data);
        else
            return inverse_discrete_fourier_transform(data);
    }

   private:
    // 计算离散傅里叶变换
    std::vector<std::complex<float>>  discrete_fourier_transform(const  std::vector<std::complex<float>>  &input_vector)
const {
        uint32_t N = input_vector.size();
        std::vector<std::complex<float>> output_vector(N);
        for (uint32_t k = 0; k < N; k++) {
            // 对于每个频率 k，计算对应的和
            for (uint32_t n = 0; n < N; n++) {
                // 对于每个样本 n，计算频率为 k 的分量
                output_vector[k] += input_vector[n] * std::exp(std::complex<float>(0, -2 * M_PI * k * n / N));
            }
        }
        return output_vector;
    }

    // 计算离散傅里叶逆变换
    std::vector<std::complex<float>>     inverse_discrete_fourier_transform(const     std::vector<std::complex<float>>
&input_vector) const {
        uint32_t N = input_vector.size();
        std::vector<std::complex<float>> output_vector(N);
        for (uint32_t n = 0; n < N; n++) {
            // 对于每个样本 n，计算对应的和
            for (uint32_t k = 0; k < N; k++) {
                // 对于每个频率 k，计算样本为 n 的分量
                output_vector[n] += input_vector[k] * std::exp(std::complex<float>(0, 2 * M_PI * k * n / N));
            }
            // 对和进行缩放，得到正确的输出
            output_vector[n] /= N;
        }
        return output_vector;
    }
};

// tree.hpp-------------------------------------------------------------------------------------------------------
/**
 * @file tree.hpp
 * @date 26.02.2023
 * @author RMSHE
 */

/*
这段代码实现了一个通用的树数据结构，包括节点的添加、删除、查找、遍历等基本操作。
该树数据结构由两个类组成：TreeNode 和 Tree。TreeNode 表示一个树节点，包含节点数据、子节点列表、父节点等属性；Tree 表示整个树结构，
包含根节点以及树的遍历、节点查找、节点删除等操作。
具体来说，TreeNode 类包含了添加子节点、查找子节点、删除子节点等方法。其中，添加子节点使用了 C++11 中的智能指针 std::unique_ptr，
确保了子节点的内存管理安全；查找子节点使用了递归的方式，深度优先遍历整个子树；
删除子节点则使用了迭代的方式，遍历整个子树进行删除操作。
Tree 类包含了树的遍历、节点查找、节点删除等方法。其中，遍历操作分为深度优先遍历和广度优先遍历两种方式；
节点查找操作同样使用了递归的方式，在根节点开始向下搜索整个子树；节点删除操作则使用了递归的方式，在整个子树中进行删除操作。
总的来说，该树数据结构提供了基本的树操作，能够满足一些基本的需求。但是需要注意的是，该树数据结构没有进行任何的平衡操作，因此对于较大
```

```
的树可能会存在效率问题。
*/

#pragma once

#include <algorithm>
#include <make_ptr.hpp>
#include <memory>
#include <queue>
#include <unordered_map>
#include <vector>

// @note 一个 TreeNode 对象代表了一棵树中的一个节点，其中包含了当前节点的数据和指向它的父节点的指针以及指向其子节点的所有指针。
template <typename T>
class TreeNode {
   public:
    T node_data;                                        // 储存这个节点的值
    std::vector<std::unique_ptr<TreeNode<T>>> children;  // 储存指向这个节点的子节点的指针
    TreeNode<T>* parent;                                 // 储存指向这个节点的父节点的指针

    /**
     * @brief "TreeNode"树节点构造函数：创建一个新的节点对象，构造节点.
     * @param data const T&类型的参数，表示根节点的数据(data 的数据类型可任意).
     * @param parent_node_ptr TreeNode<T>* 类型的参数，表示指向父节点的指针，默认为 nullptr.
     * @note 用法: TreeNode< std::string > node("data", parent_node_ptr);
     */
    TreeNode(const T& data, TreeNode<T>* parent_node_ptr = nullptr) : node_data(data), parent(parent_node_ptr) {}

    /**
     * @brief 向当前节点添加一个子节点
     * @param data const T&类型的参数，表示节点的值.
     * @return TreeNode<T>* 返回一个指向新加子节点的指针
     * @note 当调用 addChild() 函数时，它将创建一个新的 TreeNode 对象，该对象保存传递给函数的数据，并将指向新创建节点的指针添加到
当前节点的 children
     * 向量中。也就是说，addChild() 添加的是一个新的子节点。使用示例： parent_node_ptr->addChild(data) /
parent_node_ptr->addChild(data0)->addChild(data1);
     */
    TreeNode<T>* addChild(const T& data) {
        // 为类分配内存并创建对象时会自动调用类的构造函数 TreeNode(const T& data, TreeNode<T>* parent_node_ptr = nullptr);
        // parent_node_ptr->addChild(data); 在这个语句中 this 即是 parent_node_ptr;
        children.emplace_back(make_unique<TreeNode>(data, this));  // 向父节点添加一个指向子节点的指针;

        return this->findChild(data);  // 返回一个指向刚刚添加的子节点的指针
    }

    /**
     * @brief 在当前节点的子节点中查找指定数据的节点
     * @param target_child_data const T&类型的参数，表示要查找的节点数据(值).
     * @return TreeNode<T>* 指向查找到的节点的指针，如果未找到返回 nullptr.
     * @note 使用示例: parent_node_ptr->findChild(target_child_data);
     */
    TreeNode<T>* findChild(const T& target_child_data) {
        // 遍历当前节点的每一个子节点
        for (auto& child : children) {
            if (child->node_data == target_child_data) {
                // 如果当前子节点的数据等于要查找的数据，则返回该子节点的指针。
                return child.get();
            }
        }
        return nullptr;  // 如果遍历完所有子节点都没有找到，则返回 nullptr 表示没有找到该节点。
    }

    /**
     * @brief 在当前节点的后裔中查找一个指定数据值的节点.
```

```
     * @param target_node_data const T&类型的参数，表示要查找的节点数据(值).
     * @return TreeNode<T>* 指向查找到的节点的指针，如果未找到返回 nullptr.
     * @note 使用示例: parent_node_ptr->findDescendant(target_node_data);
     */
    TreeNode<T>* findDescendant(const T& target_node_data) {
        // 遍历当前节点的每一个子节点
        for (auto& child : children) {
            if (child->node_data == target_node_data) {
                // 如果当前子节点的数据等于要查找的数据，则返回该子节点的指针。
                return child.get();
            } else {
                // 否则，递归地调用子节点的 findDescendant
                // 方法来查找是否存在指定数据的节点，如果找到，则返回该子节点的指针。这是一个深度优先的递归遍历方式。
                TreeNode<T>* found = child->findDescendant(target_node_data);
                if (found != nullptr) return found;
            }
        }
        return nullptr;  // 如果遍历完所有子节点都没有找到，则返回 nullptr 表示没有找到该节点。
    }

    /**
     * @brief 判断当前节点的一个子节点是否有孩子.
     * @param node_ptr TreeNode<T>* 待判断节点的指针
     * @return 如果存在子节点返回 true，否则返回 false
     * @note 使用方法: node_ptr->hasChildren();
     */
    bool hasChildren() const {
        return !this->children.empty();  // 如果 node_ptr->children 不为空，则表示这个节点有子节点。
    }

    /**
     * @brief 删除父节点的一个无孩子子节点(删除树叶节点)
     * @param target_child_data const T& 这里需要提供待删除的目标子节点的值
     * @return 删除成功返回 true，否则返回 false
     * @note 注意这个函数只支持删除没有子节点的节点，即树枝的末端(树叶)。
     * 若要删除整颗树或部分树枝请使用"deleteNode();"函数
     * 使用示例: parent_node_ptr->deleteChild(target_child_data);
     */
    bool deleteChild(const T& target_child_data) {
        TreeNode<T>* child_node_ptr = findChild(target_child_data);  // 从当前父节点查找要删除的子节点的指针

        // 判断这个子节点是否也存在子节点，这里只支持删除没有子节点的节点(树叶)，如果存在子节点或找不到要删除的子节点，返回 false
        if (child_node_ptr == nullptr || hasChildren(child_node_ptr) == true) return false;

        // 遍历当前父节点的所有子节点，在父节点中删除要删除的节点
        for (auto it = children.begin(); it != children.end(); ++it) {
            // 从迭代器获取子节点的指针，如果该指针是要删除的目标子节点则删除它，
            // 由于使用了 std::unique_ptr 来管理子节点，所以父节点可以在内存管理方面自动处理子节点的内存释放，不需要手动释放。
            if (it->get() == child_node_ptr) {
                children.erase(it);  // 移除的目标子节点
                break;
            }
        }

        return true;
    }
};

template <typename T>
class Tree {
  public:
    std::unique_ptr<TreeNode<T>> root;          // 储存树的根节点
    TreeNode<T>* current_node_ptr = root.get();  // 储存最后一次添加节点后的指针位置(初始化时设为根节点指针)
```

```
    /**
     * @brief "Tree"树构造函数：创建一个新的 Tree 对象，构造根节点.
     * @param data const T&类型的参数，表示根节点的数据.
     * @return void
     * @note 用法: Tree< std::string > tree0("root");
     */
    Tree(const T& data) : root(make_unique<TreeNode<T>>(data)) {}

    /**
     * @brief 向当前节点添加一个子节点
     * @param node_ptr TreeNode<T>*类型的参数(指向节点的指针)，表示在该节点下添加子节点.
     * @param data const T&类型的参数，表示要添加的节点的数据.
     * @return TreeNode<T>* 返回一个指向刚刚添加的子节点的指针
     * @note 当调用 `addNode()` 函数时，它将创建一个新的 `TreeNode`
     * 对象，该对象保存传递给函数的数据，并将指向新创建节点的指针添加到当前节点的`children`向量中。也就是说，`addNode()` 添加的是一个新的子节点。`addNode`
     * 与 `addChild` 不同，`addNode` 是 `Tree class` 的成员，而 `addChild` 是`TreeNode class` 的成员，`addNode` 将父节点指针作为参数传递.
     * 该函数还会将指向新增节点的指针保存到类成员变量 `current_node_ptr`中,以便用户更清楚当前树的编辑位置.
     */
    TreeNode<T>* addNode(TreeNode<T>* node_ptr, const T& data) { return current_node_ptr = node_ptr->addChild(data); }

    /**
     * @brief 以深度优先的方式遍历树
     * @param node_ptr TreeNode<T>* 提供一个节点指针，函数会以该节点为根节点递归遍历它所有的子嗣节点(若不传参则默认遍历整颗树).
     * @return 返回一个向量，其中包含从指定节点开始子树的所有节点数据值和对应的指针 std::vector<std::pair<T, TreeNode<T>*>>
     * @note 深度优先遍历算法是递归的，它首先访问根节点，然后再递归地遍历每个子树。在每个节点访问完成后，递归函数回溯到其父节点继续遍历其他子树
     */
    std::vector<std::pair<T, TreeNode<T>*>> traversalDFS(TreeNode<T>* node_ptr = nullptr) {
        // 默认节点指针设置为根节点
        if (node_ptr == nullptr) node_ptr = root.get();

        // 如果节点为空，直接返回一个空向量
        if (node_ptr == nullptr) return {};

        // 创建一个向量，用于存储当前节点和其子节点的数据
        std::vector<std::pair<T, TreeNode<T>*>> tree_data;

        // 将当前节点的数据插入到 tree_data 向量的末尾
        tree_data.emplace_back(std::pair<T, TreeNode<T>*>(node_ptr->node_data, node_ptr));

        // 遍历当前节点的每个子节点
        for (auto& child : node_ptr->children) {
            // 递归遍历当前子节点的子树，并将其存储在 deep_tree_data 向量中
            auto deep_tree_data = traversalDFS(child.get());

            // 将当前子节点的子树数据插入到 tree_data 向量的末尾
            tree_data.insert(tree_data.end(), deep_tree_data.begin(), deep_tree_data.end());
        }

        // 返回包含当前节点及其所有子节点的数据的向量
        return tree_data;
    }

    /**
     * @brief 以广度优先的方式遍历树。
     * @param node_ptr TreeNode<T>* 提供一个节点指针，函数会以该节点为根节点递归遍历所有的子节点(若不传参则默认遍历整颗树).
     * @return 返回一个向量，其中包含从指定节点开始子树的所有节点数据值和对应的指针 std::vector<std::pair<T, TreeNode<T>*>>
     * @note 广度优先遍历算法是按层遍历，从根节点开始，先遍历根节点，然后按照从左到右的顺序遍历其子节点，再依次遍历下一层的所有节点。
     */
    std::vector<std::pair<T, TreeNode<T>*>> traversalBFS(TreeNode<T>* node_ptr = nullptr) {
        // 默认节点指针设置为根节点
```

```
        if (node_ptr == nullptr) node_ptr = root.get();

        // 当节点为空时返回空向量
        if (node_ptr == nullptr) return {};

        // 创建一个向量，用于存储当前节点和其子节点的数据
        std::vector<std::pair<T, TreeNode<T>*>> tree_data;

        // 定义队列，并将根节点推入队列
        std::queue<TreeNode<T>*> node_queue;
        node_queue.push(node_ptr);

        // 遍历队列，直到队列为空
        while (!node_queue.empty()) {
            // 取出队列头部的节点后将其弹出
            auto current_node = node_queue.front();
            node_queue.pop();

            // 将当前节点的数据值和指针插入到 tree_data 向量的末尾
            tree_data.emplace_back(std::pair<T, TreeNode<T>*>(current_node->node_data, current_node));

            // 将当前节点的所有子节点推入队列
            for (auto& child : current_node->children) node_queue.push(child.get());
        }

        // 返回包含当前节点及其所有子节点的数据的向量
        return tree_data;
    }

    /**
     * @brief 在树中查找指定数据的节点
     * @param target_node_data const T&类型的参数，表示要查找的节点数据(值).
     * @return TreeNode<T>* 指向查找到的节点的指针，如果未找到返回 nullptr.
     * @note 使用示例: tree.findNode(target_node_data);
     */
    TreeNode<T>* findNode(const T& target_node_data) {
        // 在根节点下查找目标节点(查找范围不包含根节点).
        TreeNode<T>* node_ptr = root->findDescendant(target_node_data);

        /*
        如果上一步的查找结果 node_ptr 为空指针，则:
            如果根节点的数据等于要查找目标数据，则我们可断定用户查找的是根节点直接(返回根节点的指针)，否则即在包含根节点的整颗树的范围内找不到目标节点(返回空指针)
        否则:
            在根节点下查找到目标节点，直接返回查找到的节点的指针.
        */
        return node_ptr == nullptr ? (root->node_data == target_node_data ? root.get() : nullptr) : node_ptr;
    }

    /**
     * @brief 判断指定节点是否存在子节点
     * @param node_ptr TreeNode<T>* 待判断节点的指针
     * @return 如果存在子节点返回 true，否则返回 false
     */
    bool hasChildren(TreeNode<T>* node_ptr) const {
        return !node_ptr->children.empty();  // 如果 node_ptr->children 不为空，则表示这个节点有子节点。
    }

    /**
     * @brief 递归地计算树的深度(高度)(默认统计整颗树的深度)
     * @param node_ptr TreeNode<T>*类型的参数(指向节点的指针)，表示从该节点开始统计树枝的深度，若设为 root 则为统计整颗树的深度(这也是无传参时的默认设置)
     * @return uint32_t 返回树的深度.
     * @note 使用示例: 1.统计整颗树的深度: tree.getHeight();    2.统计从 node1 节点开始的树枝深度: tree.getHeight(node1_ptr);
```

```
     */
    uint32_t getDepth(TreeNode<T>* node_ptr = nullptr) {
        // 默认节点指针设置为根节点
        if (node_ptr == nullptr) node_ptr = root.get();

        // 如果节点没有子节点，说明当前节点是叶子节点，返回 1 作为高度
        if (hasChildren(node_ptr) == false) return 1;

        uint32_t max_depth = 0;  // 定义最大深度（一个树中最少有一个树枝最长）;

        // 如果节点有子节点，则递归计算子节点的高度，并找到其中最大的高度
        for (const auto& child : node_ptr->children) max_depth = std::max(max_depth, getDepth(child.get()));

        // 返回最大高度加上 1，即为整个树的高度
        return max_depth + 1;
    }

    /**
     * @brief 获取节点的度(对于一个给定的节点，其子节点的数量称为度．一个叶子的度数一定是零)
     * @param node_ptr TreeNode<T>*类型的参数(指向节点的指针)，表示获取该节点的子节点的个数，无传参时默认获取根节点的度．
     * @return uint32_t 返回目标节点的子节点的个数．
     */
    uint32_t getDegree(TreeNode<T>* node_ptr = nullptr) {
        // 默认节点指针设置为根节点
        if (node_ptr == nullptr) node_ptr = root.get();

        // 直接返回目标节点的子节点个数;
        return node_ptr->children.size();
    }

    /**
     * @brief 获取树的度(树的度是指树中一个节点的最大度，即树中某个拥有最多子节点的父节点的子节点数)
     * @return 返回树的度
     */
    uint32_t get_degree_of_tree() {
        uint32_t max_degree = 0;  // 初始化最大度为 0

        // 对树进行深度优先遍历获取所有节点的子节点个数，这里将当前最大值 max_degree 与遍历到的父节点的子节点个数进行比较后取较大值更新回 max_degree 中．
        for (auto& node : traversalDFS()) max_degree = std::max(max_degree, node.second->children.size());

        // 遍历完成后返回树中所有结点的度的最大值
        return max_degree;
    }

    /**
     * @brief 获取树或树枝的叶子数量(叶子即没有子节点的节点，也称做终端节点)
     * @param node_ptr TreeNode<T>*类型的参数(指向节点的指针)，表示获取以该节点为起点的树枝的叶子个数，无传参时默认获取整个树的叶子数量．
     * @return 返回树或指定树枝的叶子数量
     */
    uint32_t getBreadth(TreeNode<T>* node_ptr = nullptr) {
        // 默认节点指针设置为根节点
        if (node_ptr == nullptr) node_ptr = root.get();

        uint32_t num_leaves = 0;  // 初始化叶子数为 0

        // 对树进行深度优先遍历;
        for (auto& node : traversalDFS(node_ptr))
            // 如果一个节点没有子节点，则增加叶子数．
            if (hasChildren(node.second) == false) ++num_leaves;

        // 返回树的叶子数量
        return num_leaves;
```

```
    }

    /**
     * @brief 获取树的宽度或指定节点所在层的宽度(宽度指一个层的节点数)
     * @param node_ptr TreeNode<T>*类型的参数(指向节点的指针)，表示获取该节点所在层的宽度，无传参时默认获取整个树的宽度(拥有最大宽度的层级).
     * @return 无参数时返回树的宽度，有参数时返回参数节点所在层的宽度.
     */
    uint32_t getWidth(TreeNode<T>* node_ptr = nullptr) {
        // 如果没有传递指针，则默认使用根节点指针。
        if (node_ptr == nullptr) node_ptr = root.get();

        uint32_t max_level_width = 0;                       // 用于储存树的宽度(拥有最大宽度的层级)
        std::unordered_map<uint32_t, uint32_t> level_widths;  // 用于存储每个层级的节点数的哈希表

        // 使用广度优先搜索遍历每个节点
        for (auto& node : traversalBFS(node_ptr)) {
            uint32_t level = static_cast<uint32_t>(getLevel(node.second));  // 获取当前节点的层级
            auto iter = level_widths.find(level);                          // 查找当前层级是否已经在哈希表中存在

            if (iter != level_widths.end()) {
                ++iter->second;                                           // 如果存在，就增加该层级的节点数
                max_level_width = std::max(max_level_width, iter->second);  // 更新树的宽度
            } else
                level_widths.insert({level, 1});  // 如果不存在，就将该层级的节点数设置为 1
        }

        // 如果传入的指针是根节点指针或空指针，则返回树的宽度；否则返回该节点所在层级的节点数
        return node_ptr == root.get() ? max_level_width : level_widths.find(getLevel(node_ptr))->second;
    }

    /**
     * @brief 获取指定节点的层级(一个节点的层级是它与根节点之间唯一路径上的边的数量，根节点层级为零)
     * @param node_ptr TreeNode<T>*类型的参数(指向节点的指针)，表示获取该节点的层级.
     * @return 返回指定节点的所在层级数，如果提供的节点指针为空指针则返回-1
     */
    int32_t getLevel(TreeNode<T>* node_ptr) {
        // 当节点为空时返回错误信息
        if (node_ptr == nullptr) return -1;

        int32_t level = 0;  // 初始化节点层级为 0

        // 从提供的节点 node_ptr 开始向根节点查找，在每一次循环中增加 level 的数量，直到遍历到根节点(根节点没有父节点)
        while (node_ptr->parent != nullptr) {
            level++;
            node_ptr = node_ptr->parent;
        }

        return level;
    }

    /**
     * @brief 获取树或树枝的大小(节点数)
     * @param node_ptr TreeNode<T>*类型的参数(指向节点的指针)，表示获取以该节点为起点的树枝的节点总数，无传参时默认获取整个树的节点总数.
     * @return 返回树或指定树枝的节点总数
     */
    uint32_t getSize(TreeNode<T>* node_ptr = nullptr) {
        // 默认节点指针设置为根节点
        if (node_ptr == nullptr) node_ptr = root.get();

        // 对树进行深度优先遍历然后返回树中的节点数;
        return traversalDFS(node_ptr).size();
    }
```

```
    /**
     * @brief 判断树是否为空
     * @return 如果树为空，则返回 true；如果向量不为空，则返回 false。
     */
    bool empty() { return root == nullptr ? true : false; }

    /**
     * @brief 递归删除一个节点及其后裔
     * @param node_ptr TreeNode<T>* 这里需要提供指向待删除节点的指针
     * @return 删除成功返回 true，否则返回 false
     * @note 使用示例: tree.deleteNode(node_1);
     */
    bool deleteNode(TreeNode<T>* node_ptr) {
        // 如果节点为空，直接返回
        if (node_ptr == nullptr) return false;

        // 获取该节点的所有子节点，并遍历删除它们
        auto& children = node_ptr->children;
        for (auto it = children.begin(); it != children.end();) {
            auto& child = *it;  // it 是迭代器，*it 是迭代器所指的内容.

            // 这里是在判断一个节点是否有孩子.
            if (hasChildren(child.get()) == true)
                deleteNode(child.get());  // 如果子节点不是树叶节点(有孩子)，则递归调用 deleteNode
            else
                it = children.erase(it);  // 移除树叶节点(如果删除成功，该函数会返回指向被删除元素之后的元素的迭代器)
        }

        // 检查当前节点 node_ptr 是否为根节点，如果是，则返回 false，这里无法删除根节点.
        if (node_ptr->parent == nullptr) return false;

        // 在删除完这个节点的孩子后，删除它自身
        auto parent = node_ptr->parent;  // 获取当前节点的父节点指针

        // 在父节点的子节点列表中查找并移除当前节点
        parent->children.erase(
            // 在容器中查找符合某个条件的元素，并将其移动到容器的末尾
            std::remove_if(parent->children.begin(),  // 查找的起始位置
                           parent->children.end(),    // 查找的终止位置

                           // 查找谓词，用于确定哪些元素符合要求。该函数或函数对象接受一个元素作为参数.
                           // remove_if 算法会将 child 传入这个匿名函数，如果 child 指针与 node_ptr 相同则返回 true.
                           [node_ptr](std::unique_ptr<TreeNode<T>>& child) { return child.get() == node_ptr; }),
            parent->children.end()  // 移除的终止位置
        );

        return true;
    }

    /**
     * @brief 移除一颗树.
     * @note 使用示例: tree.deleteTree();
     */
    void deleteTree() {
        deleteNode(root.get());  // 删除根节点的所有子嗣节点;
        root.reset();            // 移除根节点(将根节点重置为 nullptr);
    }

    // Tree 的析构函数
    ~Tree() { deleteTree(); }
};
```