NanoPi R5S配置为Thingsboard远程监控平台步骤

第一步：准备工作

开始前，准备好以下物品

microSD 卡	容量 > =16GB
nanopi r5s开发板	1个
网线	1条
USB-C 供电线 + PD 电源适配器	1条

电脑上下载好镜像文件和烧录软件Win32 Disk Imager

1.1 下载镜像文件

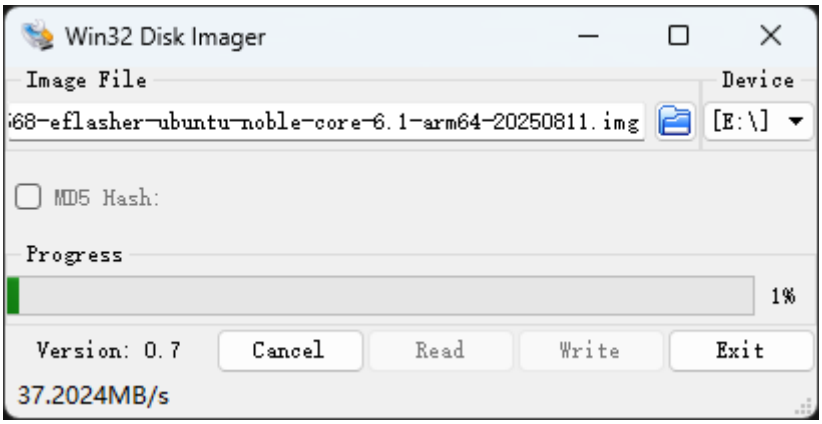
访问下载链接下载官方镜像文件（位于“01_Official images”目录）。

	rk3568-XYZ-ubuntu-noble-core-6.1-arm64-YYYYMMDD.img.zip	noble	64-bit Ubuntu image file based on Ubuntu core 24.04 64bit	6.1.y

下载上图镜像文件。

1.2 下载烧录工具

访问下载链接（在“05_Tools”目录中）下载win32diskimager.rar



第二步：烧录镜像

请按照以下步骤操作：

- 将一张 microSD 卡；
- 访问下载链接下载镜像文件（在“01_Official images/01_SD card images”目录下）；
- 访问下载链接下载win32diskimager工具（在“05_Tools”目录中），或者使用您喜欢的工具；
- 解压.gz格式压缩文件，得到.img格式镜像文件；

- 在 Windows 系统中以管理员身份运行 win32diskimager 实用程序。在实用程序的主窗口中，选择 SD 卡的驱动器、所需的映像文件，然后点击“写入”即可开始刷写 SD 卡。

- 取出SD卡，插入NanoPi-R5S的microSD卡槽；
- 打开 NanoPi-R5S 电源，它将从您的 TF 卡启动，红灯快速闪烁代表正在启动。三个绿灯长亮后即可弹出microSD卡，开发板将自动重启；

第三步：终端操作

3.1 远程登录

使用网线将Nanopi R5S的其中一个lan口连接至电脑，并获取该设备的IP地址

Win+R打开powershell终端命令行界面。使用ssh命令登录该设备

```
ssh pi@10.1.0.154
```

将上面的ip地址替换为你实际的设备，随后会弹出输入密码，默认密码为pi

3.2 终端命令

3.2.1 准备与基础工具

更新软件包索引并安装常用工具（nano、htop 等）

```
sudo apt update
```

```
sudo apt --fix-broken install -y
sudo apt install -y ca-certificates curl gnupg lsb-release
```

```
sudo apt update
sudo apt install -y nano htop
```

3.2.2 设置中文 UTF-8 locales & 安装 locales 包

说明：确保系统使用中文 UTF-8（避免 Python脚本 / 日志中文乱码）

```
sudo apt update
sudo apt install -y locales
sudo locale-gen zh_CN.UTF-8
sudo update-locale LANG=zh_CN.UTF-8
```

使当前 shell 立即生效（不想重启时）：

```
export LANG=zh_CN.UTF-8
export LC_ALL=zh_CN.UTF-8
```

3.2.3 安装 Docker（含常用插件）与配置镜像加速

说明：先安装必要依赖、添加镜像源（此处使用阿里云镜像源），然后安装 Docker 与 compose 插件。

先更新并安装依赖：

```
sudo apt update
sudo apt upgrade -y
sudo apt install -y apt-transport-https ca-certificates curl software-
properties-common gnupg lsb-release
```

导入 Docker GPG（示例：阿里镜像；如果你使用官方源或其它镜像请替换）：

```
curl -fsSL http://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key
add -
```

准备 trusted.gpg 和添加 apt 源（注：请确认 `$(lsb_release -cs)` 对应你的发行版代号）：

```
sudo cp /etc/apt/trusted.gpg /etc/apt/trusted.gpg.d/
sudo add-apt-repository "deb [arch=amd64] http://mirrors.aliyun.com/docker-
ce/linux/ubuntu $(lsb_release -cs) stable"
```

```
Press [ENTER] to continue or Ctrl-c to cancel.
```

遇到上述提示，直接按enter回车键继续

安装 Docker 与相关插件：

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
```

将当前用户加入 docker 组（使普通用户可运行 docker）：

```
sudo usermod -aG docker $USER
```

提示：`usermod` 后需要重启或重新登录使组变更生效；你可以重启机器或退出再登录。

```
sudo reboot
```

然后重新通过 SSH 登录（示例）：

```
ssh pi@10.1.0.154
```

3.2.4 配置 Docker 镜像加速与日志策略（daemon.json）

说明：创建或替换 `/etc/docker/daemon.json`，配多个 registry mirror 并设置日志滚动策略。

注意：该命令块包含多行 JSON, 单独作为文件写入的命令块不可与其它命令合并复制。

```
sudo tee /etc/docker/daemon.json <<-'EOF'
{
    "registry-mirrors": [
        "https://docker.m.daocloud.io",
        "https://mirror.baidubce.com",
        "http://hub-mirror.c.163.com",
        "https://docker.mirrors.ustc.edu.cn",
        "https://docker.hpcloud.cloud",
        "https://docker.unsee.tech",
        "https://docker.1panel.live",
        "http://mirrors.ustc.edu.cn",
        "https://docker.chenby.cn",
        "http://mirror.azure.cn",
        "https://dockerpull.org",
        "https://dockerhub.icu",
        "https://hub.rat.dev",
        "https://proxy.1panel.live",
        "https://docker.1panel.top",
        "https://docker.1ms.run",
        "https://docker.ketches.cn",
        "https://registry.docker-cn.com"
    ],
    "log-driver": "json-file",
    "log-opts": {
        "max-size": "10m",
        "max-file": "3"
    },
    "storage-driver": "vfs"
}
EOF
```

重新加载 systemd 并重启 docker：

```
sudo systemctl daemon-reload
sudo systemctl start docker
sudo systemctl enable docker
```

验证 docker 信息（检查 `Registry Mirrors`）：

```
sudo docker info
```

试运行 hello-world 镜像验证 Docker 是否正常：

```
sudo docker run hello-world
```

3.2.5 安装 Python 依赖 & 串口组权限

说明：安装 Python 包与 paho-mqtt、pyserial，确保能通过 pip 使用这些库。

```
# 1) 安装创建 venv 所需的系统包
sudo apt update
sudo apt --fix-broken install
sudo apt install -y python3-venv python3-dev build-essential
```

```
# 2) 确认项目目录存在，并列出内容
ls -la /opt/serial_to_frp
```

```
# 3) 创建新的 venv（以 system python3 创建到 /opt）
sudo python3 -m venv /opt/serial_to_frp/venv
```

```
# 4) 确保 venv 中有 pip（bootstrap）
sudo /opt/serial_to_frp/venv/bin/python -m ensurepip --upgrade || true
```

```
# 5) 升级 pip/setuptools/wheel（在 venv 内）
sudo /opt/serial_to_frp/venv/bin/python -m pip install --upgrade pip setuptools
wheel
```

```
# 6) 在 venv 内安装需要的包（pyserial, paho-mqtt）
sudo /opt/serial_to_frp/venv/bin/python -m pip install pyserial paho-mqtt
```

```
# 7) 验证 pyserial 可导入
sudo /opt/serial_to_frp/venv/bin/python -c "import serial; print('pyserial OK,',
getattr(serial,'__version__','unknown'))"
```

验证 Python 库版本（单行命令）：

```
python3 -c "import importlib.metadata as m; print('paho-mqtt:', m.version('paho-
mqtt')); print('pyserial:', m.version('pyserial'))"
```

给当前用户串口（dialout）权限（需重新登录生效）：

```
sudo usermod -aG dialout $USER
```

```
exit
```

然后重新通过 SSH 登录（示例）：

```
ssh pi@10.1.0.154
```

3.2.6 用 Docker 部署 ThingsBoard CE（PostgreSQL，持久化）

说明：将 ThingsBoard 用 Docker Compose 部署，数据持久化到宿主机目录。

切换到项目目录并创建目录结构：

```
mkdir -p ~/tb-stack
cd ~/tb-stack
```

> 该命令包含 YAML 内容，单独作为文件写入命令块，勿与其它命令混合复制。

```
cat > docker-compose.yml <<'YAML'
services:
  thingsboard:
    image: thingsboard/tb-postgres
    restart: unless-stopped
    ports:
      - "8080:8080"   # ← 确保存在
      - "9090:9090"
      - "1883:1883"
    environment:
      TB_QUEUE_TYPE: in-memory
      HTTP_BIND_ADDRESS: "0.0.0.0"
      HTTP_BIND_PORT: "8080"
      JAVA_OPTS: "-Xms256m -Xmx512m"
    volumes:
      - ./tb-data:/data
      - ./tb-logs:/var/log/thingsboard
YAML
```

创建持久化目录与权限设置：

```
mkdir -p ~/tb-stack/tb-data ~/tb-stack/tb-logs
```

将目录所有权改为容器内 TB 运行的 UID（ThingsBoard 使用 UID=799）：

```
sudo chown -R 799:799 ~/tb-stack/tb-data ~/tb-stack/tb-logs
```

可选：给读写权限便于排查：

```
sudo chmod -R u+rwX,g+rX,o+rX ~/tb-stack/tb-data ~/tb-stack/tb-logs
```

备份当前 daemon.json（以防）：

```
sudo mkdir -p /etc/docker
sudo cp /etc/docker/daemon.json /etc/docker/daemon.json.bak 2>/dev/null || true
```

在目录下启动 compose：

```
cd ~/tb-stack
sudo docker compose up -d
```

#查看 thingsboard 日志（首启会初始化 DB，等待 `Installation finished successfully!`）：

```
#docker compose logs -f thingsboard
```

MQTT 客户端自检（本机）：

```
sudo apt update
sudo apt install -y mosquitto-clients
```

3.2.7 安装 frpc（SAKURA FRP客户端）

登录SAKURA FRP官网并点击管理面板可进行隧道管理，账户名 `xiaomohaa` 密码 `Ap119119`

说明：下载对应架构（ARM64）的 frpc 二进制到 `/usr/local/bin`，并设置 systemd 模板启用多个实例。

切换到 root shell（可选）：

```
sudo -s
```

进入放置目录并使用wget下载（请以面板给出的下载地址为准）：

```
cd /usr/local/bin
```

```
sudo wget -O frpc https://nya.globalslb.net/natfrp/client/frpc/0.51.0-sakura-12.3/frpc_linux_arm64
```

或用 curl（任选其一）：

```
sudo curl -Lo frpc https://nya.globalslb.net/natfrp/client/frpc/0.51.0-sakura-12.3/frpc_linux_arm64
```

授权并校验：

```
chmod 755 frpc
ls -ls frpc
md5sum frpc
```

验证版本（若命令失败请确认路径）：

```
frpc -v
```

创建配置目录并检查：

```
sudo mkdir -p /etc/frp
sudo chown root:root /etc/frp
sudo chmod 755 /etc/frp
ls -la /etc/frp
```

确认 frpc 可执行：

```
ls -l /usr/local/bin/frpc || which frpc || echo "frpc 未找到"
```

3.2.8 创建两个独立的 frpc 配置文件（test1.ini / test2.ini）

说明：每个 ini 对应一个 `frpc@<name>.service` systemd 实例。以下为写入文件的命令块（单独写入）。

test1.ini：

```
sudo tee /etc/frp/test1.ini > /dev/null <<'EOF'
[common]
user = 78yrqxeswkb5kgshk4ygrnzolzecsagc

sakura_mode = true
login_fail_exit = false

server_addr = frp-rug.com
server_port = 8088

[test1]
type = tcp
local_ip = 127.0.0.1
local_port = 1883
remote_port = 10276
EOF
```

test2.ini：

```
sudo tee /etc/frp/test2.ini > /dev/null <<'EOF'
[common]
user = 78yrqxeswkb5kgshk4ygrnzolzecsagc

sakura_mode = true
login_fail_exit = false

server_addr = frp-rug.com
server_port = 8088

[test2]
type = tcp
local_ip = 127.0.0.1
local_port = 8443
remote_port = 63719
EOF
```

设置权限（推荐 root:root，0644）：

```
sudo chown root:root /etc/frp/test1.ini /etc/frp/test2.ini
sudo chmod 644 /etc/frp/test1.ini /etc/frp/test2.ini
ls -l /etc/frp/*.ini
```

创建 systemd 模板单元（`frpc@.service`）：

```
sudo tee /etc/systemd/system/frpc@.service > /dev/null <<'EOF'
[Unit]
```

```
Description=SakuraFrp frpc (%i)
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
User=root
ExecStart=/usr/local/bin/frpc -c /etc/frp/%i.ini
Restart=always
RestartSec=3

[Install]
WantedBy=multi-user.target
EOF
```

re-load systemd 并启用/启动两个实例：

```
sudo systemctl daemon-reload
sudo systemctl enable --now frpc@test1.service frpc@test2.service
```

检查状态与日志：

```
sudo systemctl status frpc@test1 --no-pager -l
sudo systemctl status frpc@test2 --no-pager -l
```

3.2.9 安装并配置 Caddy（本地 TLS + 反向代理到 ThingsBoard）

说明：Caddy 用作本地 TLS 的反向代理（把外部通过 frp 转发来的 8443 请求反代到本机 8080）。

安装前准备（添加仓库密钥并安装）：

```
sudo apt update
sudo apt install -y debian-keyring debian-archive-keyring apt-transport-https
```

导入 GPG 并添加 Caddy 源：

```
# 1. 如果目录不存在就创建
sudo mkdir -p /usr/share/keyrings

# 2. 下载并 dearmor（推荐：生成二进制 keyring）
curl -fsSL https://dl.cloudsmith.io/public/caddy/stable/gpg.key \
  | gpg --dearmor \
  | sudo tee /usr/share/keyrings/caddy-stable-archive-keyring.gpg >/dev/null
```

安装 Caddy：

```
sudo apt update
sudo apt install -y caddy
```

创建 Caddyfile（单独写入文件）：

```
sudo tee /etc/caddy/Caddyfile > /dev/null <<'EOF'

# Caddy 反代 ThingsBoard（把下面域名换成你实际的 frp 域名）
```

```
frp-rib.com:8443 {
    reverse_proxy 127.0.0.1:8080
    tls internal
}

frp-rug.com:8443 {
    reverse_proxy 127.0.0.1:8080
    tls internal
}

# 本地调试备用
localhost:8443 {
    reverse_proxy 127.0.0.1:8080
    tls internal
}

127.0.0.1:8443 {
    reverse_proxy 127.0.0.1:8080
    tls internal
}

EOF
```

重载并重启 Caddy 服务：

```
sudo systemctl daemon-reload

sudo systemctl restart caddy
#服务初次启动时请使用 sudo systemctl start caddy
```

查看 Caddy 状态与日志（快速查看最近日志）：

```
sudo systemctl status caddy --no-pager -l
sudo journalctl -u caddy -n 80 --no-pager
```

导出 Caddy 本地 CA 根证书到系统证书存储并更新证书存储（Debian/Ubuntu）：

```
sudo cp /var/lib/caddy/.local/share/caddy/pki/authorities/local/root.crt
/usr/local/share/ca-certificates/Caddy_Local_Authority.crt
sudo chmod 644 /usr/local/share/ca-certificates/Caddy_Local_Authority.crt
sudo update-ca-certificates
```

本机 TLS 验证（使用 openssl）：

```
openssl s_client -connect 127.0.0.1:8443 -servername localhost -showcerts
</dev/null 2>&1 | sed -n '1,240p'
```

用 curl 测试通过 SNI 访问（把域名解析到 127.0.0.1）：

```
curl -vk --resolve frp-rug.com:8443:127.0.0.1 https://frp-rug.com:8443/ -o
/dev/null -w "%{http_code}
"
```

3.2.10 串口采集脚本（serial_to_frp）和虚拟环境

说明：把脚本放到 `/opt/serial_to_frp`，建立 venv 并安装依赖，脚本会把串口数据通过 MQTT 发到对应的 ThingsBoard device（token 规则在脚本中）。

创建目录与设置权限：

```
sudo mkdir -p /opt/serial_to_frp
sudo chown -R $USER:$USER /opt/serial_to_frp
```

创建 venv 并安装 Python 包：

```
python3 -m venv /opt/serial_to_frp/venv
/opt/serial_to_frp/venv/bin/pip install --upgrade pip
/opt/serial_to_frp/venv/bin/pip install paho-mqtt pyserial
```

将完整 Python 脚本写入文件：

```
sudo nano /opt/serial_to_frp/serial_to_frp.py
```

进入编辑界面，粘贴下方完整代码，Ctrl+O保存修改，Ctrl+X退出

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
serial_to_frp.py

功能：
  - 从串口读取传感器/节点数据
  - 解析 CASE / CHASSIS / NODE 等格式
  - 按 CASE 建立 MQTT 客户端并通过 FRP 隧道发送遥测与属性

注意：
  - device_token 规则：
      device_token = BASE_TOKEN_PREFIX + PROJECT_NO + "CASE" + 3位case_id
  - 本脚本只替换了 parse_data，使其更宽容、健壮，能识别像
      CASE:1,Temp=29.31C, Smoke=42.73PPM
    这种带空格、单位的输入。
"""

import serial, time, json, re, threading
from collections import defaultdict
import queue, sys, glob, logging
import paho.mqtt.client as mqtt

# ---------------- logging ----------------
logging.basicConfig(
    level=logging.INFO,  # 如需调试解析细节，把这里改为 logging.DEBUG
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[logging.FileHandler("serial_to_frp.log"), logging.StreamHandler()]
)
logger = logging.getLogger(__name__)

# ===== 自动串口检测 =====
def find_serial_port():
```

```
    for pattern in ["/dev/ttyUSB*", "/dev/ttyACM*", "/dev/ttyS*"]:
        ports = glob.glob(pattern)
        if ports:
            ports.sort()
            return ports[0]
    return None

# ===== 配置区（按需修改）=====
SERIAL_PORT = find_serial_port()
BAUD_RATE = 115200

BASE_TOKEN_PREFIX = "UX4XLX1ZGZL416"
PROJECT_NO = "NO002"
# ===这里PROJECT_NO需要根据不同的项目来进行修改===

# 指向 test1 隧道（MQTT）
FRP_SERVER = "frp-rug.com"
FRP_PORT   = 10276

if SERIAL_PORT is None:
    logger.error("未找到可用的串口设备; 程序将重试启动由 systemd 拉起")
    sys.exit(1)

logger.info(f"使用串口设备: {SERIAL_PORT}")

# ===== 全局 =====
case_node_last_seen = defaultdict(lambda: defaultdict(float))
all_known_nodes = defaultdict(set)
mqtt_clients = {}

def get_mqtt_client(case_id):
    """
    为指定 case_id 返回（或创建）对应的 MQTT 客户端并启动 loop。
    使用 device_token 作为用户名进行认证（无密码）。
    """
    if case_id not in mqtt_clients:
        device_token = f"{BASE_TOKEN_PREFIX}
{PROJECT_NO}CASE{str(case_id).zfill(3)}"
        logger.info(f"创建MQTT客户端: CASE {case_id}, 令牌: {device_token}")
        client = mqtt.Client()
        client.username_pw_set(device_token)
        def on_connect(client, userdata, flags, rc):
            if rc == 0:
                logger.info(f"☑ CASE {case_id} MQTT连接成功")
            else:
                logger.error(f"❌ CASE {case_id} MQTT连接失败，错误码: {rc}")
        client.on_connect = on_connect
        # 非阻塞连接（loop_start() 启动线程）
        client.connect(FRP_SERVER, FRP_PORT, 60)
        client.loop_start()
        mqtt_clients[case_id] = client
    return mqtt_clients[case_id]

# ===== 数据解析函数 =====
def chassis_to_case(chassis_id: int) -> int:
    """
    将 CHASSIS 映射为 CASE:
      1/2 -> CASE 1, 3/4 -> CASE 2, 5/6 -> CASE 3, 以此类推。
```

```
    """
    cid = int(chassis_id)
    return (cid + 1) // 2

def parse_data(raw_data: str):
    """
    更稳健的串口数据解析。

    支持格式（宽松匹配）：
      - CASE:<n>,NODE:<m>,VOLT:<v>V,TEMP:<t>
      - CHASSIS:<n>,NODE:<m>,VOLT:<v>V,TEMP:<t>  （映射为对应 CASE）
      - CASE:<n>,Temp=<tt>C,Smoke=<ss>PPM
    特性：
      - 忽略以 'SensorData:' 开头的行（大小写不敏感）
      - 兼容 : 或 = 分隔，允许字段间有空格或逗号后的空格
      - 自动去除单位后缀（如 C / V / PPM / % 等）
      - 若无法用宽松方式解析，则回退到严格正则以兼容旧格式
    返回：
      list of dict，dict 的 type 为 'node' 或 'case'，字段与原脚本一致
    """
    results = []
    if not raw_data:
        return results

    # 逐行解析，避免跨行拼接导致匹配失败
    for line in raw_data.splitlines():
        s = line.strip()
        if not s:
            continue
        # 忽略 SensorData: 开头的行
        if s.lower().startswith('sensordata:'):
            logger.debug(f"忽略行 (SensorData): {s}")
            continue

        # 宽松的键值对匹配：Key :|= Number [units optional]
        # 捕获形式例如： "Temp=29.31C" 或 "VOLT:12.3V" -> ('Temp','29.31') ; 忽略大小
写
        kv_pattern = r'([A-Za-z]+)\s*[:=]\s*([+-]?\d+(?:\.\d+)?)(?:[A-Za-z%]*)'
        matches = re.findall(kv_pattern, s, flags=re.IGNORECASE)
        kv_pairs = {k.upper(): v for k, v in matches}  # 统一为大写 key ->
value(string)

        # 获取 case_id：优先 CASE，其次 CHASSIS -> 转换为 CASE
        case_id = None
        if 'CASE' in kv_pairs:
            try:
                case_id = int(kv_pairs['CASE'])
            except Exception:
                case_id = None
        elif 'CHASSIS' in kv_pairs:
            try:
                case_id = chassis_to_case(int(kv_pairs['CHASSIS']))
            except Exception:
                case_id = None

        # 1) 节点数据：需要 NODE, VOLT, TEMP 三个字段
        if 'NODE' in kv_pairs and 'VOLT' in kv_pairs and 'TEMP' in kv_pairs:
            try:
```

```
                node_id = int(kv_pairs['NODE'])
                volt = float(kv_pairs['VOLT'])
                temp = float(kv_pairs['TEMP'])
                # 如果未在 kv_pairs 得到 CASE，但行中可能使用 CHASSIS 或明确写了 CASE
（上面已尝试）
                # 若仍未找到 case_id，尝试使用更宽松的正则去搜 CASE:xx 或 CHASSIS:xx
                if case_id is None:
                    m_case = re.search(r'CASE\s*[:=]\s*(\d+)', s, re.IGNORECASE)
                    if m_case:
                        case_id = int(m_case.group(1))
                    else:
                        m_ch = re.search(r'CHASSIS\s*[:=]\s*(\d+)', s,
re.IGNORECASE)
                        if m_ch:
                            case_id = chassis_to_case(int(m_ch.group(1)))
                if case_id is None:
                    # 如果仍然无法确认 CASE，记录 debug 并跳过该行
                    logger.debug(f"节点行无法确定 CASE，跳过: {s}")
                else:
                    results.append({
                        'type': 'node',
                        'case_id': int(case_id),
                        'node_id': node_id,
                        'volt': volt,
                        'temp': temp
                    })
                    continue  # 处理下一行
            except Exception as e:
                logger.debug(f"解析节点行出错: {s} ; err: {e}")

        # 2) 机箱数据：需要至少 TEMP 或 SMOKE，并且有 CASE（或 CHASSIS）
        if case_id is not None and ('TEMP' in kv_pairs or 'SMOKE' in kv_pairs):
            try:
                temp = float(kv_pairs.get('TEMP')) if 'TEMP' in kv_pairs else
None
                smoke = float(kv_pairs.get('SMOKE')) if 'SMOKE' in kv_pairs else
None
                # 若没有某个值，按原脚本惯例传 0.0（也可选择跳过）
                results.append({
                    'type': 'case',
                    'case_id': int(case_id),
                    'temp': temp if temp is not None else 0.0,
                    'smoke': smoke if smoke is not None else 0.0
                })
                continue
            except Exception as e:
                logger.debug(f"解析机箱行出错: {s} ; err: {e}")

        # 3) 回退：尝试原有更严格的正则（兼容旧输入）
        # 节点（CASE:1,NODE:2,VOLT:12.3V,TEMP:36.1）
        node_pat_case = r'CASE\s*[:=]\s*(\d+)\s*,\s*NODE\s*[:=]\s*
(\d+)\s*,\s*VOLT\s*[:=]\s*([\d.]+)V\s*,\s*TEMP\s*[:=]\s*([+-]?\d+(?:\.\d+)?)'
        m_node_case = re.search(node_pat_case, s, re.IGNORECASE)
        if m_node_case:
            try:
                case_id_r = int(m_node_case.group(1))
                node_id = int(m_node_case.group(2))
                volt = float(m_node_case.group(3))
```

```
                temp = float(m_node_case.group(4))
                results.append({
                    'type': 'node',
                    'case_id': case_id_r,
                    'node_id': node_id,
                    'volt': volt,
                    'temp': temp
                })
                continue
            except Exception:
                pass

        # CHASSIS 节点（回退）
        node_pat_chassis = r'CHASSIS\s*[:=]\s*(\d+)\s*,\s*NODE\s*[:=]\s*
(\d+)\s*,\s*VOLT\s*[:=]\s*([\d.]+)V\s*,\s*TEMP\s*[:=]\s*([+-]?\d+(?:\.\d+)?)'
        m_node_chassis = re.search(node_pat_chassis, s, re.IGNORECASE)
        if m_node_chassis:
            try:
                chassis_id = int(m_node_chassis.group(1))
                case_id_r = chassis_to_case(chassis_id)
                node_id = int(m_node_chassis.group(2))
                volt = float(m_node_chassis.group(3))
                temp = float(m_node_chassis.group(4))
                results.append({
                    'type': 'node',
                    'case_id': case_id_r,
                    'node_id': node_id,
                    'volt': volt,
                    'temp': temp
                })
                continue
            except Exception:
                pass

        # 原有 CASE 机箱严格模式：CASE:1,Temp=29.31C,Smoke=42.73PPM
        case_pat = r'CASE\s*[:=]\s*(\d+)\s*,\s*Temp\s*[=:\s]\s*
([\d.]+)C\s*,\s*Smoke\s*[=:\s]\s*([\d.]+)PPM'
        m_case = re.search(case_pat, s, re.IGNORECASE)
        if m_case:
            try:
                results.append({
                    'type': 'case',
                    'case_id': int(m_case.group(1)),
                    'temp': float(m_case.group(2)),
                    'smoke': float(m_case.group(3))
                })
                continue
            except Exception:
                pass

        # 未识别行（记录 debug，便于定位）
        logger.debug(f"未识别的串口行: {s}")

    return results

# ===== 数据处理函数 =====
def process_node(case_id, node_id, volt, temp):
    """
```

```
    处理单个节点数据：更新最后见到时间、发现新节点时记录、并通过 MQTT 发布遥测
    """
    case_node_last_seen[case_id][node_id] = time.time()
    if node_id not in all_known_nodes[case_id]:
        logger.info(f"发现新节点: CASE {case_id}, NODE {node_id}")
        all_known_nodes[case_id].add(node_id)
    client = get_mqtt_client(case_id)
    telemetry = { f"节点{node_id}电压": volt, f"节点{node_id}温度": temp }
    try:
        # publish 返回 MQTTMessageInfo 对象，检查 rc
        r = client.publish("v1/devices/me/telemetry", json.dumps(telemetry),
qos=1)
        # 如果需要更严格地等待发送完成可以使用 r.wait_for_publish()
        if getattr(r, "rc", None) == mqtt.MQTT_ERR_SUCCESS:
            logger.info(f"☑ CASE {case_id} 节点{node_id}遥测发送成功")
        else:
            logger.error(f"❌ CASE {case_id} 节点{node_id}发送失败: {getattr(r,
'rc', 'unknown')}")
    except Exception as e:
        logger.error(f"❌ CASE {case_id} 节点{node_id}异常: {e}")

def process_case(case_id, temp, smoke):
    """
    处理机箱数据：通过 MQTT 发布机箱温度与烟雾浓度
    """
    client = get_mqtt_client(case_id)
    telemetry = {"机箱温度": temp, "烟雾浓度": smoke}
    try:
        r = client.publish("v1/devices/me/telemetry", json.dumps(telemetry),
qos=1)
        if getattr(r, "rc", None) == mqtt.MQTT_ERR_SUCCESS:
            logger.info(f"☑ CASE {case_id} 机箱遥测发送成功")
        else:
            logger.error(f"❌ CASE {case_id} 机箱遥测发送失败: {getattr(r, 'rc',
'unknown')}")
    except Exception as e:
        logger.error(f"❌ CASE {case_id} 机箱异常: {e}")

def update_node_status(case_id):
    """
    定期更新节点在线/离线属性并通过 attributes 发布
    """
    now = time.time()
    status = {}
    for node_id in all_known_nodes[case_id]:
        is_online = (now - case_node_last_seen[case_id].get(node_id,0)) <= 120
        status[f"节点{node_id}"] = "在线" if is_online else "离线"
    client = get_mqtt_client(case_id)
    try:
        r = client.publish("v1/devices/me/attributes", json.dumps(status),
qos=1)
        if getattr(r, "rc", None) == mqtt.MQTT_ERR_SUCCESS:
            logger.info(f"☑ CASE {case_id} 节点状态属性发送成功")
        else:
            logger.error(f"❌ CASE {case_id} 节点状态属性发送失败: {getattr(r, 'rc',
'unknown')}")
    except Exception as e:
        logger.error(f"❌ CASE {case_id} 属性发送异常: {e}")
```

```python
def check_node_status():
    """
    后台线程：每分钟检查所有 case 的节点状态并更新 attributes
    """
    while True:
        try:
            for case_id in list(case_node_last_seen.keys()):
                update_node_status(case_id)
            time.sleep(60)
        except Exception as e:
            logger.error(f"检查节点状态时出错: {e}")
            time.sleep(10)

def monitor_mqtt():
    """
    后台线程：定期检查 MQTT 客户端连接状态，尝试重连
    """
    while True:
        try:
            for case_id, client in list(mqtt_clients.items()):
                if not client.is_connected():
                    logger.warning(f"⚠ CASE {case_id} MQTT断开，重连")
                    try:
                        client.reconnect()
                    except Exception as e:
                        logger.error(f"❌ CASE {case_id} 重连失败: {e}")
            time.sleep(30)
        except Exception as e:
            logger.error(f"连接监控出错: {e}")
            time.sleep(10)

def main():
    """
    主循环：打开串口、读取数据、累积缓冲、按超时或完整解析触发上报
    """
    try:
        ser = serial.Serial(SERIAL_PORT, BAUD_RATE, bytesize=serial.EIGHTBITS,
                            parity=serial.PARITY_NONE,
stopbits=serial.STOPBITS_ONE, timeout=1)
        ser.flushInput()
        logger.info(f"开始监听串口 {SERIAL_PORT}，波特率 {BAUD_RATE}...")
        # 启动后台线程
        threading.Thread(target=check_node_status, daemon=True).start()
        threading.Thread(target=monitor_mqtt, daemon=True).start()

        buf, last = "", time.time()
        while True:
            # 有数据时读取并追加到缓冲
            if ser.in_waiting > 0:
                raw = ser.read(ser.in_waiting).decode('utf-8', errors='ignore')
                buf += raw
                last = time.time()
                if raw.strip():
                    logger.info(f"收到原始数据: {repr(raw)}")
                parsed = parse_data(buf)
                if parsed:
                    # 按 case 聚合并处理
```

```
                    grouped = defaultdict(list)
                    for d in parsed:
                        grouped[d['case_id']].append(d)
                    for case_id, items in grouped.items():
                        logger.info(f"处理 CASE {case_id} 数据，共 {len(items)}
条")
                        for d in items:
                            if d['type'] == 'node':
                                process_node(d['case_id'], d['node_id'],
d['volt'], d['temp'])
                                update_node_status(d['case_id'])
                            else:
                                process_case(d['case_id'], d['temp'],
d['smoke'])
                    # 处理后清空缓冲（若你的串口可能有多条独立消息但不含换行，需要按需修改）
                    buf = ""

            # 超时处理（0.5s 无新数据则尝试解析现有缓冲）
            if time.time() - last > 0.5 and buf:
                parsed = parse_data(buf)
                grouped = defaultdict(list)
                for d in parsed:
                    grouped[d['case_id']].append(d)
                for case_id, items in grouped.items():
                    logger.info(f"超时处理 CASE {case_id} 共 {len(items)} 条")
                    for d in items:
                        if d['type'] == 'node':
                            process_node(d['case_id'], d['node_id'], d['volt'],
d['temp'])
                            update_node_status(d['case_id'])
                        else:
                            process_case(d['case_id'], d['temp'], d['smoke'])
                buf = ""
                last = time.time()

            time.sleep(0.1)

    except KeyboardInterrupt:
        logger.info("用户中断")
    except Exception as e:
        logger.error(f"程序运行出错: {e}")
    finally:
        try:
            ser.close()
            logger.info("☑ 串口已关闭")
        except Exception:
            pass
        for case_id, client in mqtt_clients.items():
            try:
                client.disconnect()
                logger.info(f"☑ CASE {case_id} MQTT关闭")
            except Exception:
                pass
        logger.info("🎯 退出")

if __name__ == "__main__":
    main()
```

最后一步赋予可执行权限：

```
sudo chmod +x /opt/serial_to_frp/serial_to_frp.py
```

3.2.11 systemd 单元：串口脚本、ThingsBoard Stack 服务

serial_to_frp systemd（随开机自启）

```
sudo tee /etc/systemd/system/serial_to_frp.service > /dev/null <<'UNIT'
[Unit]
Description=Serial to ThingsBoard via FRP
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
User=root
ExecStart=/opt/serial_to_frp/venv/bin/python /opt/serial_to_frp/serial_to_frp.py
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
UNIT
```

启用并启动：

```
sudo systemctl daemon-reload
sudo systemctl enable --now serial_to_frp
sudo systemctl status serial_to_frp --no-pager -n 20
```

tb-stack systemd（用 systemd 管理 docker compose）

说明：把 `docker compose up -d` 放入 oneshot 服务，随开机启动 ThingsBoard Stack。

```
sudo tee /etc/systemd/system/tb-stack.service > /dev/null <<'UNIT'
[Unit]
Description=ThingsBoard Stack (docker compose)
After=docker.service network-online.target
Wants=docker.service network-online.target

[Service]
Type=oneshot
WorkingDirectory=/home/pi/tb-stack
ExecStart=/usr/bin/docker compose up -d
ExecStop=/usr/bin/docker compose down
RemainAfterExit=yes
TimeoutStartSec=0

[Install]
WantedBy=multi-user.target
UNIT
```

启用并启动：

```
sudo systemctl daemon-reload
sudo systemctl enable --now tb-stack
sudo systemctl status tb-stack --no-pager
```

3.2.12 一键使服务随开机启动 & 重启验证

说明：把常用服务一次性 enable 并重启验证系统启动项。

```
sudo systemctl enable --now frpc@test1 frpc@test2 serial_to_frp caddy docker
```

建议手动重启并检查各服务状态：

```
sudo reboot
```

重启后检查服务：

```
sudo systemctl status frpc@test1 --no-pager -l
sudo systemctl status frpc@test2 --no-pager -l
sudo systemctl status serial_to_frp --no-pager -l
sudo systemctl status caddy --no-pager -l
sudo systemctl status docker --no-pager -l
```

至此完成所有的配置步骤 ⬅ END

常见验证与故障排查命令

- 查看 docker 日志 / 容器状态：

```
docker ps -a
docker logs <container> --tail 200
```

- 查看 journal 日志（Caddy/Frp/serial）：

```
sudo journalctl -u caddy -n 200 --no-pager
sudo journalctl -u frpc@test1 -n 200 --no-pager
sudo journalctl -u serial_to_frp -n 200 --no-pager
```

- 检查端口监听（确认 8080/1883/8443 是否就绪）：

```
sudo ss -tulpn | grep -E '8080|1883|8443'
```

- 本机 curl 健康检查（ThingsBoard）：

```
curl -sS http://127.0.0.1:9090/api/health ; echo
```