xiaozhi-esp32 源码导读

第 1 章 项目总览与文件分级总览表

> 本章解决两个问题：① 这份代码是个什么东西、整体怎么跑起来；② 哪些文件是核心、哪些是辅助、哪些一辈子都不用看。后面所有章节都会基于这张总览表展开。

1.1 项目是什么

`xiaozhi-esp32-main` 是开源项目「小智 AI 聊天机器人」的 v2 版本固件源码（仓库 `78/xiaozhi-esp32`）。代码角色：

角色	说明
麦克风	用 I²S 采集 16 kHz PCM
唤醒词识别	板载 ESP-SR（“你好小智”等），整段离线运行
编解码	Opus（窄带语音压缩），上行下行都是 Opus 帧
通信	选 WebSocket 或 MQTT+UDP 之一，向云端发音频 + 收音频/文本
云端语音处理	ASR（语音转文字）→ LLM（Qwen/DeepSeek 等）→ TTS（合成语音），全部在服务器
表情/状态显示	OLED / LCD / AMOLED / e-Paper 均可，UI 走 LVGL
设备控制	通过 **MCP 协议**（一种基于 JSON-RPC 2.0 的工具调用协议）暴露音量、灯光、电机、GPIO 等给云端大模型

一句话：**麦克风 + 网络 + 喇叭 + 一个能被 LLM 调用 GPIO 的协议** = 一个会对话能控制硬件的小盒子。

支持的芯片：ESP32 / ESP32-C3 / ESP32-S3 / ESP32-C5 / ESP32-C6 / ESP32-P4。开发板（boards/）有 110 块。

1.2 顶层目录速览

```
xiaozhi-esp32-main/
├── CMakeLists.txt              # 顶层 CMake，引 ESP-IDF
├── sdkconfig.defaults*         # 各芯片默认 menuconfig 配置
├── README*.md                  # 项目自述（中英日三份）
├── LICENSE                     # 开源协议
├── main/                       # 应用代码（本项目的全部业务逻辑）
├── partitions/                 # 分区表 v1 / v2
├── scripts/                    # Python 辅助脚本：资源打包、发布、音频调试
├── docs/                       # 协议文档（MCP、WebSocket、MQTT+UDP、自定义板子）
└── .github/                    # GitHub Actions CI 配置（每个板子自动构建）
```

只有 `main/` 是业务代码。其它都是构建胶水、文档、CI——读不读完全不影响理解。

`main/` 下进一步拆：

```
main/
├── CMakeLists.txt                # 列出所有源文件 + 110 个板子的条件分支选择
├── Kconfig.projbuild             # 项目级 menuconfig 选项（板子型号、AEC 模式、唤醒词等）
├── idf_component.yml             # ESP-IDF 组件依赖清单（esp-sr、opus、lvgl 等）
├── main.cc                       # app_main()，30 行
├── application.{cc,h}            # ★ 全局单例 Application，主事件循环（1055 行）
├── device_state.h                # 11 个状态枚举
├── device_state_machine.{cc,h}   # ★ 状态机 + 回调观察者
├── ota.{cc,h}                    # ★ 服务器握手 / 激活 / 固件升级
├── settings.{cc,h}               # NVS 键值对封装
├── system_info.{cc,h}            # 芯片型号 / MAC / 堆内存信息
├── mcp_server.{cc,h}             # ★ 设备端 MCP 工具调用服务器
├── assets.{cc,h}                 # ★ assets 分区（字体/表情/唤醒词模型）管理
│
├── audio/                        # ★ 音频子系统
├── protocols/                    # ★ WebSocket / MQTT+UDP 双协议
├── display/                      # 显示子系统（LCD / OLED / e-Paper / 表情）
├── led/                          # 状态指示灯
├── assets/                       # 多语言文字 + 提示音 OGG 文件
└── boards/                       # 110 个开发板的硬件适配
    ├── common/                   # ★ 板级共用基类（Board / WifiBoard / Ml307Board / blufi 等）
    └── <某板子>/                 # 各板子具体的 GPIO 引脚、屏幕驱动、按键映射
```

带 ★ 的就是后面会逐文件逐函数讲的核心模块。

1.3 全局架构与启动流程

1.3.1 一张图看懂启动顺序

```
ESP-IDF 启动
   │
   ▼
app_main()  ── main.cc                                  初始化 NVS 闪存
   │
   ▼
Application::GetInstance()                              ① 单例 → 创建事件组 + 时钟 timer
   │
   ▼
Application::Initialize()                               ② 一系列同步初始化（在 app_main 上下文）
   │    ├─ Board::GetInstance()                             └─ 板级单例（board.cc 里 create_board()）
   │    ├─ display = board.GetDisplay()                     └─ 拿屏（每个板子构造时已创建）
   │    ├─ audio_service_.Initialize(codec)                 └─ 装好编解码器、起 audio 任务
   │    ├─ audio_service_.Start()
   │    ├─ audio_service_.SetCallbacks(...)                 └─ 把"有 Opus 包可发"等回调挂到 event group
   │    ├─ state_machine_.AddStateChangeListener(...)       └─ 状态变 → 触发 event group
   │    ├─ esp_timer_start_periodic(1s)                     └─ 心跳 timer
   │    ├─ McpServer::GetInstance().AddCommonTools()        └─ 注册 self.audio_speaker.set_volume 等通
用工具
   │    ├─ board.SetNetworkEventCallback(...)               └─ 网络连/断/扫描等 UI 反馈
   │    └─ board.StartNetwork()                            └─ 异步启动 WiFi / 4G
   │
   ▼
Application::Run()                                      ③ 永不返回的主事件循环
   │
   │  while (true) {
   │      xEventGroupWaitBits(...);    ← 13 种事件位
   │      处理：状态变 / 网络变 / 唤醒词 / VAD / 心跳 / 主线程任务 / ...
   │  }
   │
   └── 与此并行运行的后台任务：
         ├─ ActivationTask（一次性）：拉服务器配置、CheckNewVersion、激活、初始化协议
         ├─ audio_input_task / audio_output_task / opus_codec_task
         ├─ LVGL 任务（在 display.cc / LvglDisplay 里起）
         ├─ 网络栈本身的 RTOS 任务（WiFi/MQTT/WS/UDP）
         └─ 各板子的辅助任务（按键防抖、电池监控、屏幕背光定时器 等）
```

1.3.2 谁负责什么——一句话总结

模块	责任
`main.cc`	上电跳板，30 行
`Application`	大调度器，所有事件汇总在它的 `event_group_` 上，谁也别绕开
`Board`	板子抽象，每块板子各自实现 `audio_codec / display / network` 等接口的具体硬件
`AudioService`	三个 RTOS 任务管全部音频读写、Opus 编解码、唤醒词馈喂、AFE 处理
`Protocol`（基类）+ `WebsocketProtocol` / `MqttProtocol`	收发两边的 JSON 控制信令 + 二进制 Opus 帧
`McpServer`	把”音量”“灯光”“GPIO”等本地能力注册成 MCP 工具，给云端 LLM 调用
`Ota`	启动后向 `xiaozhi.me` 发 POST 拿配置（含 MQTT/WS 端点），处理激活码与固件升级
`Assets`	管理 SPI Flash 上一个名为 `assets` 的分区（字体、表情、唤醒词模型按文件名 mmap）
`Display`	UI 抽象。`LcdDisplay` / `OledDisplay` / `EmoteDisplay` 各自实现
`Led`	状态指示灯（单 LED 闪烁、WS2812 灯环、RGB）
`Settings`	对 ESP-IDF NVS 的轻量封装，读写小配置

1.3.3 数据流速览

上行：你说话 → 服务器

```
MIC (I²S)
  │
  ▼  AudioCodec::InputData → vector<int16_t>
audio_input_task
  │
  ├─ 喂 → WakeWord::Feed     （若处于 idle、需要监听唤醒词）
  ├─ 喂 → AudioProcessor::Feed（若 listening 状态，AFE 做降噪/VAD/AEC）
  └─ 否则丢弃

AudioProcessor 输出
  │
  ▼  PushTaskToEncodeQueue(kEncodeToSendQueue)
opus_codec_task
  │
  ▼  Opus 编码
audio_send_queue_  ──→ MAIN_EVENT_SEND_AUDIO
                            │
                            ▼  Application::Run() 取包
                            protocol_->SendAudio(...)
                            │
                            ▼
                       WebSocket 二进制帧 / UDP+AES 加密 → 服务端 ASR+LLM+TTS
```

下行：服务器 → 喇叭

```
WebSocket/UDP 收到二进制 Opus
  │
  ▼  protocol_ on_incoming_audio 回调
  audio_service_.PushPacketToDecodeQueue(...)

opus_codec_task
  │
  ▼  Opus 解码
audio_playback_queue_
  │
  ▼
audio_output_task
  │
  ▼  AudioCodec::OutputData → I²S → 喇叭
```

控制信令（JSON）走另一条路：

```
WebSocket/MQTT JSON 文本
  │
  ▼ protocol_ on_incoming_json
   Application::InitializeProtocol 中注册的大 switch（按 type 分发）
       ├─ "tts" / "stt" / "llm" → 改设备状态 + UI
       ├─ "mcp" → McpServer::ParseMessage（工具调用）
       ├─ "alert" → 弹提示
       └─ "system" → reboot 等
```

1.4 全文件分级总览表（这是本章最值钱的一张表）

> 分级说明：
>
> - ★★★ **核心**：业务主干，必须逐文件读懂，关键函数要逐段注释
> - ★★ **重要**：模块的稳定基础，要懂作用 + 主要类/函数 + 关键技术点
> - ★ **辅助**：偶尔读一眼即可，会按用途归类讲，但不展开
> - ○ **板级硬件**：110 块板子里挑代表讲，其余看名字知道是什么
> - — **构建/资源**：CMake / Kconfig / 资源文件 / CI / 文档，不是 C++ 业务代码

1.4.1 main/ 根目录（11 个文件）

文件	行数	分级	一句话作用	主要技术
`main.cc`	30	★★★	入口；NVS 初始化 + Application 单例 + Run()	ESP-IDF `app_main`
`application.h`	186	★★★	Application 类定义 + 13 种事件位宏 + AEC 模式枚举	事件组、单例
`application.cc`	1055	★★★	大调度器：初始化、主循环、所有协议回调、激活、升级、状态切换	FreeRTOS EventGroup、esp_timer、lambda、`Schedule(任务)`、互斥锁、`unique_ptr`
`device_state.h`	17	★★★	11 个状态枚举（idle/listening/speaking 等）	`enum`
`device_state_machine.h`	83	★★★	状态机 + 观察者回调	`std::atomic<>`、互斥锁
`device_state_machine.cc`	161	★★★	状态转换合法性表 + 通知监听者	状态机、`switch case` 转换表、观察者模式
`ota.h`	58	★★	OTA 类接口（CheckVersion / Activate / Upgrade）	—
`ota.cc`	473	★★	跟 `xiaozhi.me` 握手取配置、激活码循环、esp_https_ota 升级	HTTP POST + JSON、HMAC-SHA256、`esp_https_ota`
`assets.h`	52	★★	Assets 单例接口	—
`assets.cc`	532	★★	管理 SPI Flash 上一个 `assets` 分区：下载、校验、mmap、按文件名取数据	`esp_partition_*`、`mmap`、流式 CRC、HTTP 下载
`mcp_server.h`	344	★★★	Property / PropertyList / McpTool / McpServer 完整类型定义 + base64 编码	`std::variant`、`std::optional`、模板、JSON-RPC 2.0
`mcp_server.cc`	563	★★★	MCP 协议解析 + initialize / tools/list / tools/call 三大方法实现 + 内置工具注册	JSON-RPC、cJSON、线程池跑工具
`settings.h`	28	★	Settings 类接口	—
`settings.cc`	108	★	NVS 命名空间封装	`nvs_flash`
`system_info.h`	22	★	静态方法集合	—
`system_info.cc`	151	★	取 MAC / 芯片型号 / 堆内存 / 任务表	`esp_chip_info`、`uxTaskGetSystemState`

1.4.2 main/audio/ —— 音频子系统（19 个文件）

文件	行数	分级	一句话作用	主要技术
`audio_codec.h`	61	★★★	AudioCodec 抽象基类：双工标志、I²S 句柄、Read/Write 纯虚	I²S、抽象基类
`audio_codec.cc`	77	★★	基类默认实现：音量、增益、Start	默认行为
`audio_service.h`	160	★★★	AudioService 接口 + 队列上限宏 + 事件位宏 + AudioServiceCallbacks	任务/队列设计
`audio_service.cc`	686	★★★	三任务（input/output/codec）+ 5 个队列 + 唤醒词/AFE 切换 + 节流 + Opus 重采样	FreeRTOS Task、条件变量、deque 队列、`std::chrono` 节流、Opus 编解码、采样率重采样
`audio_processor.h`	26	★★	AudioProcessor 抽象基类（前端处理）	抽象
`wake_word.h`	26	★★	WakeWord 抽象基类	抽象
`processors/afe_audio_processor.{cc,h}`	187+	★★	ESP-SR AFE 实现：VAD + AEC + 降噪	ESP-SR AFE API
`processors/no_audio_processor.{cc,h}`	59+	★	兜底无处理实现（小芯片用）	直通
`processors/audio_debugger.{cc,h}`	67+	★	录音转发到本机 Python 脚本听效果	UDP 流
`wake_words/afe_wake_word.{cc,h}`	208+	★★	用 ESP-SR AFE 内置唤醒词	wakenet 模型
`wake_words/custom_wake_word.{cc,h}`	254+	★★	自定义 MFCC + 简单分类器	信号处理
`wake_words/esp_wake_word.{cc,h}`	87+	★★	兼容小芯片（C3）的轻量唤醒	wakenet9
`codecs/no_audio_codec.{cc,h}`	359+	★★	软件 I²S codec（无外置芯片，靠 MCU 自驱 PDM/麦克风）	I²S std、PDM
`codecs/es8311_audio_codec.{cc,h}`	196+	○	ES8311 I²S codec 芯片驱动（最常见）	I²C 配置寄存器
`codecs/es8374_audio_codec.{cc,h}`	197+	○	ES8374 codec	同上
`codecs/es8388_audio_codec.{cc,h}`	221+	○	ES8388 codec（lyrat/某些 box）	同上
`codecs/es8389_audio_codec.{cc,h}`	203+	○	ES8389 codec	同上
`codecs/box_audio_codec.{cc,h}`	244+	○	乐鑫 ESP-BOX 自家组合方案	双芯片协作

文件	行数	分级	一句话作用	主要技术
`codecs/dummy_audio_codec.{cc,h}`	20+	○	无音频硬件占位（编译过）	假实现
`README.md`	几行	—	模块说明	—

1.4.3 main/protocols/ —— 通信协议（6 个文件）

文件	行数	分级	一句话作用	主要技术
`protocol.h`	98	★★★	`Protocol` 基类 + `AudioStreamPacket` + 两版二进制帧头 + 几个 enum	抽象基类、PoD struct + `__attribute__((packed))`
`protocol.cc`	90	★★★	公共回调注册 + 发送 abort/listen/mcp 控制 JSON + 超时判断	JSON 字符串拼接
`websocket_protocol.h`	34	★★★	WebsocketProtocol 类声明	—
`websocket_protocol.cc`	253	★★★	单条 WSS 连接同时传 JSON 文本 + Opus 二进制帧	`esp_websocket_client`、HTTP hello 握手
`mqtt_protocol.h`	65	★★★	MqttProtocol 类声明	—
`mqtt_protocol.cc`	382	★★★	MQTT 走控制信令；UDP+AES-128-CTR 走 Opus 媒体流；hello/goodbye 协调	MQTT 客户端、UDP socket、mbedtls AES-CTR、随机 nonce

1.4.4 main/display/ —— 显示子系统（13 个文件）

文件	行数	分级	一句话作用	主要技术
`display.h`	81	★★	Display 基类接口 + `DisplayLockGuard` RAII + `NoDisplay` 占位	抽象、RAII
`display.cc`	56	★	基类默认实现（多数为空、被子类覆盖）	—
`oled_display.{cc,h}`	396+	★★	SSD1306/SSD1315 OLED LVGL 适配	LVGL、I²C 屏
`lcd_display.{cc,h}`	1196+	★★	LCD（ST7789、ILI9341、AMOLED、e-Paper）LVGL 适配，含状态栏/聊天气泡布局	LVGL widget、字体、按需重绘
`emote_display.{cc,h}`	657+	★	表情专用显示风格（动画化、整屏铺满）	LVGL Animation
`lvgl_display/lvgl_display.{cc,h}`	258+	★★	LVGL 启动、tick 任务、主题切换	LVGL Port
`lvgl_display/lvgl_theme.{cc,h}`	30+	★	主题（深/浅色）	—
`lvgl_display/lvgl_font.{cc,h}`	12+	★	字体注册（普惠/font_awesome）	—
`lvgl_display/lvgl_image.{cc,h}`	63+	★	图片资源解码	—
`lvgl_display/emoji_collection.{cc,h}`	123+	★	表情包查表（twemoji 32/64）	—
`lvgl_display/gif/{gifdec.c, lvgl_gif.cc}`	—	★	GIF 解码（emote_display 用）	—
`lvgl_display/jpg/{image_to_jpeg.cpp, jpeg_to_image.c}`	—	★	JPEG 编解码（带相机的板子用）	—

1.4.5 main/led/ —— LED 状态指示（4 个文件）

文件	行数	分级	一句话作用	主要技术
`led.h`	17	★	Led 基类 + NoLed 占位	抽象
`single_led.{cc,h}`	—	★	单 LED（GPIO 闪烁）	esp_timer 定时切换
`gpio_led.{cc,h}`	—	★	多色 LED（多 GPIO 组合）	—
`circular_strip.{cc,h}`	—	★	WS2812 RGB 灯环（state→颜色映射）	RMT 驱动

1.4.6 main/boards/common/ —— 板级共用基类（22 个文件）

文件	行数	分级	一句话作用	主要技术
`board.h`	93	★★★	Board 基类 + `DECLARE_BOARD` 宏 + `NetworkEvent` enum + `PowerSaveLevel`	抽象基类、虚函数、DECLARE_BOARD 单例宏
`board.cc`	178	★★	公共方法默认实现（GenerateUuid、GetSystemInfoJson、GetLed/Display 默认占位）	UUID 生成
`wifi_board.{cc,h}`	359+	★★★	所有 WiFi 板子的基类：连 WiFi、配网（SmartConfig/AP/BluFi）、事件分发	esp_wifi、esp_netif、事件
`ml307_board.{cc,h}`	274+	★★	4G 模组（中移 ML307）板子的基类	UART + AT 指令
`dual_network_board.{cc,h}`	—	★	同板支持 WiFi+4G 切换	双栈共存
`button.{cc,h}`	—	★★	按键防抖、长短按区分	iot_button 组件
`knob.{cc,h}`	—	★	旋钮（编码器）	iot_knob
`backlight.{cc,h}`	—	★	LCD 背光 PWM	ledc
`power_save_timer.{cc,h}`	—	★	空闲计时降功耗	esp_timer
`sleep_timer.{cc,h}`	—	★	深度睡眠倒计时	esp_sleep
`system_reset.{cc,h}`	—	★	长按硬件复位、出厂	nvs erase + reset
`i2c_device.{cc,h}`	—	★	I²C 设备父类（codec/PMIC 复用）	i2c_master
`adc_battery_monitor.{cc,h}`	—	★	ADC 读电压估电量	adc_oneshot
`axp2101.{cc,h}`	—	○	AXP2101 电源管理 IC 驱动	I²C
`sy6970.{cc,h}`	—	○	SY6970 充电 IC	I²C
`camera.h` + `esp32_camera.{cc,h}`	—	○	DVP/MIPI 摄像头驱动（带摄像头的板子用）	esp32-camera 组件
`lamp_controller.h`	—	○	灯具控制接口	—
`afsk_demod.{cc,h}`	—	○	AFSK 声波配网解调（手机用 web 发 wifi 配置）	DSP
`blufi.{cpp,h}`	777	○	蓝牙 BLE 配网（BluFi 协议）	nimble BLE
`press_to_talk_mcp_tool.{cc,h}`	—	★	把"按住说话"模式注册成 MCP 工具	MCP tool 复用模板

1.4.7 main/boards/<各板子>/ —— 110 块开发板

每块板子目录下基本只有 3 类文件：

```
<board-name>/
├── config.h                          硬件 GPIO 引脚号宏定义
├── config.json                       给 CI 用的"这个板子叫什么、追加哪些 sdkconfig"
├── <board>_board.cc                  继承 WifiBoard / Ml307Board，实现 GetAudioCodec / GetDisplay
等
└── README.md / 图片                  可选
```

> 本文档不会把 110 块全部拆开讲——里面 95% 是引脚号差异。后面**第 9 章**会挑 3 个代表（`bread-compact-wifi` 面包板、`esp-box-3` 乐鑫官方开发板、`xmini-c3-4g` 带 4G 的硬件）讲清楚"如何看懂一个板子目录"。其余板子你看到目录名 `xxx-s3-touch-amoled-1.8` 这种就能猜出大概，遇到具体硬件 GPIO 才去对应目录翻 `config.h`。

1.4.8 main/assets/ —— 资源文件（非代码）

子目录	内容	分级
`common/`	5 个通用 OGG 提示音（success / exclamation / vibration / popup / low_battery）	—
`locales/<lang>/`	36 种语言的 `language.json`（文本翻译表）+ 部分语言独有的 OGG 数字播报音（活码激活时）	—
`lang_config.h`	CMake 阶段由 `scripts/gen_lang.py` 自动从对应 locale 的 JSON 生成的 C++ 头	—

讲解时这里只需要知道："文本翻译表 + 提示音 ogg 文件，编译期被嵌进固件二进制"。

1.4.9 顶层辅助资源

路径	分级	用途
`partitions/v1/` 、`partitions/v2/`	—	不同 Flash 容量、不同芯片的分区表 CSV
`scripts/gen_lang.py`	—	从 locales/*/language.json 生成 C++ 头
`scripts/build_default_assets.py`	—	把字体+表情+唤醒词模型打成一个 assets.bin
`scripts/release.py` 、`scripts/versions.py`	—	发版构建脚本
`scripts/audio_debug_server.py`	—	配合 `audio_debugger.cc` 监听设备的 UDP 流听音质
`scripts/ogg_converter/` 、`scripts/mp3_to_ogg.sh`	—	提示音转码工具
`scripts/Image_Converter/`	—	LVGL 图片资源转换
`scripts/spiffs_assets/` 、`scripts/p3_tools/` 、`scripts/acoustic_check/`	—	资源/声学测试小工具
`scripts/sonic_wifi_config.html`	—	AFSK 声波配网用的浏览器页面
`docs/mcp-protocol.md`	—	**MCP 协议官方说明**
`docs/mcp-usage.md`	—	MCP 用法教程
`docs/websocket.md`	—	WebSocket 通信细节
`docs/mqtt-udp.md`	—	MQTT + UDP 媒体流细节
`docs/custom-board.md`	—	怎么添加自己的板子
`docs/blufi.md`	—	BluFi 蓝牙配网说明
`docs/v0/` 、`docs/v1/`	—	旧版资料
`.github/workflows/`	—	每块板子的 CI 构建工作流

1.5 本项目核心技术与方法清单

后面章节会随用随讲，这里先列一份”看这份代码会学到什么”的索引：

1.5.1 FreeRTOS / 嵌入式并发原语

技术	在哪里用到	干什么
EventGroup（事件组）	`application.cc` 的 `event_group_`、`audio_service.cc` 的 `event_group_`	不同 task 之间用 1 bit 1 个事件来”提醒”主循环干活
xTaskCreate	`audio_service.cc`（input/output/codec 三个任务）、`application.cc` 的激活任务、各板子的按键任务	创建独立 RTOS 任务
esp_timer 周期定时	`application.cc` 的 clock_timer（1 秒滴答更新状态栏）、`led/single_led.cc` 的闪烁	软定时器在 timer 任务上下文回调
std::mutex / std::lock_guard	`application.cc` 的 main_tasks_ 队列、`device_state_machine.cc` 的 listeners_、`audio_service.cc` 的 audio_queue_mutex_	跨任务共享数据保护
std::condition_variable	`audio_service.cc` 的 audio_queue_cv_	队列空/满时阻塞/唤醒 codec 任务
std::atomic	`device_state_machine.cc` 的 current_state_	状态字段无锁读写
vTaskDelete	激活任务做完后自杀	一次性任务清理
uxTaskPriorityGet/Set + RAII（`TaskPriorityReset`）	`application.h` 末尾那个 helper 类	临时提权再恢复，常见于关键短任务

1.5.2 状态机 + 观察者

- 11 个状态（`device_state.h`）；
- 一张合法转换表（`device_state_machine.cc::IsValidTransition`）——非法转换直接拒绝并 `ESP_LOGW`；
- 观察者：`AddStateChangeListener` 注册回调，转换成功后挨个调；
- `Application` 把”状态变了”翻译成 `MAIN_EVENT_STATE_CHANGED` 事件位，再在主循环里执行真正的副作用（点灯/切音频任务/UI），**保证副作用都跑在主任务上下文**。

1.5.3 单例模式 + DECLARE_BOARD 宏

- `Application::GetInstance()`、`McpServer::GetInstance()`、`Assets::GetInstance()`、`Board::GetInstance()` 都用 C++11 magic static（线程安全的局部静态变量）；
- 但是 `Board` 比较特别：用宏 `DECLARE_BOARD(BoardClass)` 由具体板子定义 `create_board()`，`Board::GetInstance()` 内部调用 `static_cast<Board*>(create_board())` 完成多态实例化。**整个项目只有一处板级实例化**，靠 CMake 选板子的源文件来决定 `create_board()` 到底返回哪个子类。这就是”110 个板子怎么共用同一份 Application 代码”的关键。

1.5.4 工厂 + 抽象基类

每个子系统都搭一对”抽象基类 + 多个实现 + 工厂选择”：

基类	多个实现	工厂决定点
`Board`	`WifiBoard` / `Ml307Board` / `DualNetworkBoard` / 110 个具体板子	`DECLARE_BOARD` 宏（CMake 选板子）
`AudioCodec`	`NoAudioCodec` / `Es8311AudioCodec` / `Box*` / …	各板子的 `GetAudioCodec()`
`Display`	`OledDisplay` / `LcdDisplay` / `EmoteDisplay` / `NoDisplay`	各板子的构造函数里 new
`WakeWord`	`AfeWakeWord` / `CustomWakeWord` / `EspWakeWord`	`audio_service.cc::Initialize` + Kconfig
`AudioProcessor`	`AfeAudioProcessor` / `NoAudioProcessor`	`audio_service.cc::Initialize` + Kconfig
`Protocol`	`WebsocketProtocol` / `MqttProtocol`	`application.cc::InitializeProtocol` + 服务器下发的配置
`Led`	`SingleLed` / `GpioLed` / `CircularStrip` / `NoLed`	各板子的 `GetLed()`

1.5.5 回调式编程（避免上层关心下层细节）

- `Protocol::OnIncomingJson(...)` 让 `Application` 决定 JSON 来了怎么处理；
- `AudioServiceCallbacks` 让 `Application` 决定唤醒/VAD/可发包了怎么响应；
- `Board::SetNetworkEventCallback(...)` 让 `Application` 在网络变化时改 UI 和状态；
- `McpTool` 的 `std::function<ReturnValue(const PropertyList&)>` 让每个工具的具体实现可以是 lambda、可以是成员函数；
- `Ota::Upgrade(...)` 接受进度回调，把”进度怎么显示”和”OTA 怎么下载”解耦。

1.5.6 JSON-RPC 2.0（MCP 协议）

- `mcp_server.cc` 解析 `method` 字段，分发到 `initialize` / `notifications/initialized` / `tools/list` / `tools/call`；
- 工具描述（input schema）按 JSON Schema 规范写：type=object, properties={}, required=[...];
- 调用结果统一封装成 `{"content":[{"type":"text","text":"..."}],"isError":false}`；
- 大模型那边可以理解”这台设备暴露了 set_volume / set_brightness / press_to_talk 等工具，调用它们”。

1.5.7 二进制流协议（音频帧）

- WebSocket：帧头 `BinaryProtocol2`（16 字节）+ Opus payload；
- MQTT+UDP：UDP 帧头 `BinaryProtocol3`（4 字节）+ AES-128-CTR(nonce, opus_payload)；
- 时间戳字段用于服务端 AEC（回声消除）：服务器知道刚才发的是哪一段 TTS，就能在你这一段输入里减掉它。

1.5.8 资源管理与 Flash 分区

- 分区表（`partitions/v2/*.csv`）划出 `assets` 分区；
- `assets.bin` 的格式：固定头 + 文件索引表 + 各文件原始数据；
- 运行期：`esp_partition_mmap()` 把分区直接映射进 CPU 地址空间，然后按 `Asset { offset, size }` 表去拿数据——**字体直接 mmap 给 LVGL、模型直接 mmap 给 ESP-SR、不占内存**。
- 升级：HTTP 下载新的 `assets.bin` → 写入分区 → 校验 CRC → 重 mmap → 应用。

1.5.9 ESP-IDF 特定 API 速览

- `nvs_flash_init()`：键值存储初始化；
- `esp_event_loop_create_default()`（在 wifi_board 里）：默认事件循环；
- `esp_https_ota_*`：固件 OTA；
- `esp_partition_*`、`esp_partition_mmap()`：分区表读写、mmap；
- `esp_timer_*`：高精度软定时器；
- `esp_chip_info()` / `esp_efuse_*` / `esp_mac_addr_*`：硬件信息；
- `esp_wifi_*` / `esp_netif_*`：网络栈；
- `i2s_std_*` / `i2s_pdm_*`：音频外设；
- `mbedtls_*`：base64 / SHA / AES。

1.5.10 第三方库

- **cJSON**：所有 JSON 解析/序列化；
- **Opus**（含 encoder/decoder/resampler 三个包装类，来自管理组件）：语音编解码；
- **ESP-SR**：唤醒词模型 (wakenet) + AFE 前端处理；
- **LVGL**：UI 框架；
- **esp-websocket-client**：WebSocket；
- **esp-mqtt**：MQTT 客户端；
- **iot_button**、**iot_knob**：按键/旋钮；
- **mbedtls**：加解密；
- 4G 模组用 ML307 SDK（espressif/atomic_*）。

1.6 看完本章你应该掌握的

- 知道 `main/` 是业务代码，其它目录只是构建/资源
- 知道一次开机会经过 `app_main → Application::Initialize → board.StartNetwork → 等网通了 → ActivationTask → 拉服务器配置 → 选 WS/MQTT → 进 idle 等唤醒`
- 能看着分级表估算优先级：核心 7-8 个文件读懂就能讲清整个项目
- 知道这份代码用到的核心技术清单，遇到具体代码段能猜到属于哪个技术

下一章开始进入 `main.cc + application.cc/h` ——主调度器逐段讲解。

> 参考资料：
>
> - 项目自带 `README_zh.md` 已实现功能列表
> - `docs/mcp-protocol.md` MCP 协议
> - `docs/websocket.md` WebSocket 帧格式
> - `docs/mqtt-udp.md` MQTT + UDP 媒体流
> - `docs/custom-board.md` 添加自定义板子

第 2 章 主调度器：`main.cc` + `application.{h,cc}`

> 本章把项目的"心脏"——`Application` 单例——从上电那一刻起逐段讲清楚。整份 `application.cc` 是 1055 行，本章按"成员变量 → 构造 → Initialize → Run 主循环 → 一个个事件处理器 → 协议回调注册 → 工具方法"的顺序展开。每一段先放出原文行号，再讲它在做什么、用到什么技术、为什么这么写。

2.1 `main.cc`（30 行，逐行）

>

完整内容：

```
extern "C" void app_main(void)
{
    // Initialize NVS flash for WiFi configuration
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_LOGW(TAG, "Erasing NVS flash to fix corruption");
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    // Initialize and run the application
    auto& app = Application::GetInstance();
    app.Initialize();
    app.Run();  // This function runs the main event loop and never returns
}
```

行	干什么	知识点
`extern "C" void app_main(void)`	ESP-IDF 框架约定的入口，名字必须是 `app_main`，且不能 C++ name-mangling，所以加 `extern "C"`	C/C++ 链接约定
`nvs_flash_init()`	初始化非易失存储（NVS），后面 `Settings`、Wi-Fi 凭据、OTA 标志都要用	ESP-IDF NVS
`if (ret == ESP_ERR_NVS_NO_FREE_PAGES || …)`	第一次烧录或 NVS 版本变了就擦掉重来	故障自愈
`ESP_ERROR_CHECK(ret)`	宏，错误时打印错误码并 abort	ESP-IDF 错误宏
`Application::GetInstance()`	第一次访问触发构造（C++11 magic static，线程安全）	单例模式
`app.Initialize()`	同步走完所有”立刻能做”的事，**不阻塞等网络**	—
`app.Run()`	永不返回的事件循环	—

`app_main` 自己就是一个 FreeRTOS 任务，所以 `Run()` 里 `xEventGroupWaitBits(... portMAX_DELAY)` 阻塞的就是这个任务。

2.2 `application.h` 解析（187 行）

2.2.1 包含的头文件

```
#include <freertos/FreeRTOS.h>
#include <freertos/event_groups.h>
#include <freertos/task.h>
#include <esp_timer.h>

#include <string>
#include <mutex>
#include <deque>
#include <memory>

#include "protocol.h"
#include "ota.h"
#include "audio_service.h"
#include "device_state.h"
#include "device_state_machine.h"
```

头	用途
`event_groups.h`	FreeRTOS 事件组 API（13 个事件位都在这上面）
`task.h`	`xTaskCreate / vTaskDelete / vTaskDelay`
`esp_timer.h`	1 秒滴答 timer
`<mutex> <deque>`	`Schedule()` 把 lambda 推进 `main_tasks_` 队列
`protocol.h`	`Protocol` 基类
`ota.h`	`Ota`（激活、升级）
`audio_service.h`	`AudioService`
`device_state*.h`	状态枚举 + 状态机

注意没有 include `board.h` —— 这里只前向用到 `Board`，避免循环依赖（板子可能用到 Application）。

2.2.2 13 个事件位（事件组的核心）

```
#define MAIN_EVENT_SCHEDULE             (1 << 0)
#define MAIN_EVENT_SEND_AUDIO           (1 << 1)
#define MAIN_EVENT_WAKE_WORD_DETECTED   (1 << 2)
#define MAIN_EVENT_VAD_CHANGE           (1 << 3)
#define MAIN_EVENT_ERROR                (1 << 4)
#define MAIN_EVENT_ACTIVATION_DONE      (1 << 5)
#define MAIN_EVENT_CLOCK_TICK           (1 << 6)
#define MAIN_EVENT_NETWORK_CONNECTED    (1 << 7)
#define MAIN_EVENT_NETWORK_DISCONNECTED (1 << 8)
#define MAIN_EVENT_TOGGLE_CHAT          (1 << 9)
#define MAIN_EVENT_START_LISTENING      (1 << 10)
#define MAIN_EVENT_STOP_LISTENING       (1 << 11)
#define MAIN_EVENT_STATE_CHANGED        (1 << 12)
```

位	谁会 set 这一位	主循环收到后干什么
`SCHEDULE`	`Schedule(lambda)` 调用方	取出 `main_tasks_` 里全部 lambda 串行执行
`SEND_AUDIO`	`AudioService` 编码完一帧	从 `audio_send_queue_` 拿 Opus 包发给协议
`WAKE_WORD_DETECTED`	`WakeWord` 模型检测到唤醒	进入连接/监听
`VAD_CHANGE`	AFE 检测到”开始说话”或”结束说话”	通知 LED 变颜色
`ERROR`	协议层报网络错误	切回 idle + 弹错提示
`ACTIVATION_DONE`	激活后台任务跑完	进入正常工作流
`CLOCK_TICK`	esp_timer 每 1 秒	刷状态栏；每 10 秒打印堆栈
`NETWORK_CONNECTED` / `NETWORK_DISCONNECTED`	板级网络回调	进入激活 / 关闭音频通道
`TOGGLE_CHAT` / `START_LISTENING` / `STOP_LISTENING`	按键、`WakeWordInvoke`、MCP 工具	切换聊天状态
`STATE_CHANGED`	状态机回调	真正执行”切到新状态”的副作用（点灯、起音频任务、刷 UI）

核心设计哲学：所有”我要在主任务上做点什么”都先 set 一个事件位，主循环 wait 到后处理。这样：

- 跨任务通信不用复杂的队列、信号量；
- 大量副作用在同一上下文里串行执行，**几乎不需要互斥锁**；
- 同一时间一组事件可以一起处理（多 bit 一起触发）。

2.2.3 AecMode 枚举

```
enum AecMode {
    kAecOff,
    kAecOnDeviceSide,
    kAecOnServerSide,
};
```

AEC = Acoustic Echo Cancellation 回声消除。语音对话中麦克风会同时录到喇叭播的声音，必须消掉。

- `kAecOff`：不做 AEC——靠”半双工”，喇叭说话时不录音。
- `kAecOnDeviceSide`：本地 AFE 做（需要 ESP32-S3/P4 + 双麦克风 reference 信号）。
- `kAecOnServerSide`：本地把麦克风 PCM 一并把时间戳发上去，云端做。

构造函数里根据 sdkconfig 选哪一种：

```
#if CONFIG_USE_DEVICE_AEC && CONFIG_USE_SERVER_AEC
#error ...
#elif CONFIG_USE_DEVICE_AEC
    aec_mode_ = kAecOnDeviceSide;
#elif CONFIG_USE_SERVER_AEC
    aec_mode_ = kAecOnServerSide;
#else
    aec_mode_ = kAecOff;
#endif
```

2.2.4 Application 类的成员

公开接口（节选）：

```
static Application& GetInstance();
void Initialize();
void Run();

DeviceState GetDeviceState() const;
bool SetDeviceState(DeviceState state);
bool IsVoiceDetected() const;

void Schedule(std::function<void()>&& callback);
void Alert(...);   void DismissAlert();
void AbortSpeaking(AbortReason reason);
void ToggleChatState();  void StartListening();  void StopListening();
void Reboot();
void WakeWordInvoke(const std::string& wake_word);
bool UpgradeFirmware(const std::string& url, const std::string& version = "");
bool CanEnterSleepMode();
void SendMcpMessage(const std::string& payload);
void SetAecMode(AecMode mode);
AecMode GetAecMode() const;
void PlaySound(const std::string_view& sound);
AudioService& GetAudioService();
void ResetProtocol();
```

私有数据：

```
std::mutex mutex_;                            // 保护 main_tasks_
std::deque<std::function<void()>> main_tasks_; // Schedule() 的目的地
std::unique_ptr<Protocol> protocol_;          // WebSocket 或 MQTT，激活后才 new
EventGroupHandle_t event_group_;              // 主事件组
esp_timer_handle_t clock_timer_handle_;       // 1 秒滴答
DeviceStateMachine state_machine_;            // 状态机（不是指针，直接持有）
ListeningMode listening_mode_;                // AutoStop / ManualStop / Realtime
AecMode aec_mode_;
std::string last_error_message_;
AudioService audio_service_;                  // 直接持有，不是指针
std::unique_ptr<Ota> ota_;                    // 激活时才 new，激活完成就 reset() 释放
bool has_server_time_;
bool aborted_;
bool assets_version_checked_;
bool play_popup_on_listening_;
int clock_ticks_;
TaskHandle_t activation_task_handle_;
```

设计要点：

1. **`audio_service_` 和 `state_machine_` 直接持有**（不是 `unique_ptr`）——构造 Application 时它们一起 new，析构时一起 destroy。它俩生命期等于程序生命期。
2. **`protocol_` 和 `ota_` 用 `unique_ptr`**——前者要等握手拿到配置才能决定 new WS 还是 MQTT；后者激活完就释放（OTA 信息只在激活时有用，丢掉省内存）。
3. **`event_group_` 是 RTOS 句柄**，构造时 `xEventGroupCreate()`，析构时 `vEventGroupDelete()`。

2.2.5 私有事件处理器和 helper

```
void HandleStateChangedEvent();
void HandleToggleChatEvent();
void HandleStartListeningEvent();
void HandleStopListeningEvent();
void HandleNetworkConnectedEvent();
void HandleNetworkDisconnectedEvent();
void HandleActivationDoneEvent();
void HandleWakeWordDetectedEvent();
void ActivationTask();        // 后台任务
void CheckAssetsVersion();
void CheckNewVersion();
void InitializeProtocol();
void ShowActivationCode(const std::string& code, const std::string& message);
void SetListeningMode(ListeningMode mode);
void OnStateChanged(DeviceState old_state, DeviceState new_state);
```

逐个在 2.4–2.7 节展开。

2.2.6 文件末尾的 `TaskPriorityReset`

```
class TaskPriorityReset {
public:
    TaskPriorityReset(BaseType_t priority) {
        original_priority_ = uxTaskPriorityGet(NULL);
        vTaskPrioritySet(NULL, priority);
    }
    ~TaskPriorityReset() {
        vTaskPrioritySet(NULL, original_priority_);
    }

private:
    BaseType_t original_priority_;
};
```

这是一个 **RAII 工具**：构造时把当前任务的优先级临时提升到 `priority`，析构时恢复。用法：

```
{
    TaskPriorityReset boost(20);  // 提到优先级 20
    // ... 这一段不希望被打断
}  // 离开作用域自动恢复
```

源码里目前还没有具体调用点，但保留了这个工具，方便板子作者临时优先级。

技术点：**RAII**（Resource Acquisition Is Initialization）—— 用对象生命期管理资源/状态，C++ 习惯用法。

2.3 `application.cc` 构造与析构

2.3.1 构造函数 (23–47 行)

```
Application::Application() {
    event_group_ = xEventGroupCreate();

#if CONFIG_USE_DEVICE_AEC && CONFIG_USE_SERVER_AEC
#error "CONFIG_USE_DEVICE_AEC and CONFIG_USE_SERVER_AEC cannot be enabled at the same time"
#elif CONFIG_USE_DEVICE_AEC
    aec_mode_ = kAecOnDeviceSide;
#elif CONFIG_USE_SERVER_AEC
    aec_mode_ = kAecOnServerSide;
#else
    aec_mode_ = kAecOff;
#endif

    esp_timer_create_args_t clock_timer_args = {
        .callback = [](void* arg) {
            Application* app = (Application*)arg;
            xEventGroupSetBits(app->event_group_, MAIN_EVENT_CLOCK_TICK);
        },
        .arg = this,
        .dispatch_method = ESP_TIMER_TASK,
        .name = "clock_timer",
        .skip_unhandled_events = true
    };
    esp_timer_create(&clock_timer_args, &clock_timer_handle_);
}
```

行	干什么	技术点
`xEventGroupCreate()`	创建事件组句柄	FreeRTOS
`#if … #error`	编译期防止两种 AEC 同时开	预处理器
`esp_timer_create_args_t` 结构	描述一个软定时器：callback、参数、调度方式	ESP-IDF designated initializers
`.callback = [](void* arg){ … }`	用**无捕获 lambda** 作为 C 回调，可以隐式转换成函数指针	C++ lambda 兼容 C 回调
`xEventGroupSetBits(app->event_group_, MAIN_EVENT_CLOCK_TICK)`	timer 到时只 set 一个 bit	把 timer 上下文切到主循环上下文
`.dispatch_method = ESP_TIMER_TASK`	回调在 esp_timer 的专用任务里执行（而不是 ISR）	避免在中断里干复杂事
`.skip_unhandled_events = true`	卡住一段时间后只补一次 tick，不堆积	防止恢复后 1000 个 tick 突然涌来
`esp_timer_create(...)`	创建 timer，**还没启动**	启动在 Initialize 里

为什么 callback 用 lambda 而不写一个普通函数？因为可以把上下文（`this`）通过 `arg` 字段绑进去，避免全局变量。**无捕获 lambda 可以衰变成函数指针**，这是 C++11 起的语言特性，专门为兼容 C API 设计。

2.3.2 析构函数 (49–55 行)

```
Application::~Application() {
    if (clock_timer_handle_ != nullptr) {
        esp_timer_stop(clock_timer_handle_);
        esp_timer_delete(clock_timer_handle_);
    }
    vEventGroupDelete(event_group_);
}
```

老实的资源回收。**但实际上这个析构永远不会被调用**（单例 + Run() 不返回），写在这里只是好习惯。

2.4 `Initialize()` 逐段解读（61–164 行）

```
void Application::Initialize() {
    auto& board = Board::GetInstance();
    SetDeviceState(kDeviceStateStarting);
```

第一步把状态机推进 `unknown → starting`。`SetDeviceState` 内部调状态机的 `TransitionTo`，会触发已注册的 listener（但是此时还没注册，所以无副作用）。

```
    auto display = board.GetDisplay();
    display->SetChatMessage("system", SystemInfo::GetUserAgent().c_str());
```

拿屏幕，在系统消息位置贴一个”我是谁”——`SystemInfo::GetUserAgent()` 返回类似 `xiaozhi-esp32/v2.0.0 (xiaozhi-esp32-main; esp32s3)` 的字符串。

```
    auto codec = board.GetAudioCodec();
    audio_service_.Initialize(codec);
    audio_service_.Start();
```

拿编解码器，初始化音频服务（创建 input/output/codec 三个任务，准备好 Opus 编解码器实例，但是不立刻开麦克风、不立刻起唤醒词）。`Start()` 之后那三个任务就在 RTOS 里跑了，但都还在”事件位 = 0”状态等触发。

```
    AudioServiceCallbacks callbacks;
    callbacks.on_send_queue_available = [this]() {
        xEventGroupSetBits(event_group_, MAIN_EVENT_SEND_AUDIO);
    };
    callbacks.on_wake_word_detected = [this](const std::string& wake_word) {
        xEventGroupSetBits(event_group_, MAIN_EVENT_WAKE_WORD_DETECTED);
    };
    callbacks.on_vad_change = [this](bool speaking) {
        xEventGroupSetBits(event_group_, MAIN_EVENT_VAD_CHANGE);
    };
    audio_service_.SetCallbacks(callbacks);
```

把音频子系统的三个事件桥到主事件组：

- “send 队列里有 Opus 包了” → 主循环知道，去 Pop 然后发协议；
- “唤醒词触发了” → 主循环知道，决定切状态；
- “说话/不说话了”（VAD） → 主循环知道，去切 LED 颜色。

```
    state_machine_.AddStateChangeListener([this](DeviceState old_state, DeviceState new_state)
        {
        xEventGroupSetBits(event_group_, MAIN_EVENT_STATE_CHANGED);
    });
```

状态机本身的回调也只是 set 一个 bit——具体副作用都到主循环的 `HandleStateChangedEvent` 里干。这一条至关重要，是这份代码并发安全的核心：状态机的 `TransitionTo` 可能在任何任务里被调用（音频任务、协议任务、按键任务、MCP 任务），但所有"切换到 listening 后要起音频处理任务"这种副作用，都在主循环里执行，不会跨任务并发。

```
    esp_timer_start_periodic(clock_timer_handle_, 1000000);
```

启动那个 1 秒滴答 timer（参数单位是微秒，1000000us = 1s）。

```
    auto& mcp_server = McpServer::GetInstance();
    mcp_server.AddCommonTools();
    mcp_server.AddUserOnlyTools();
```

注册 MCP 设备工具。`AddCommonTools` 把所有板子都有的工具（音量、灯光、电池电量、设备状态等）注册一遍；`AddUserOnlyTools` 注册只能给用户看（AI 不可见，audience=["user"]）的工具。详见**第 6 章**。

```
    board.SetNetworkEventCallback([this](NetworkEvent event, const std::string& data) {
        auto display = Board::GetInstance().GetDisplay();

        switch (event) {
            case NetworkEvent::Scanning:
                display->ShowNotification(Lang::Strings::SCANNING_WIFI, 30000);
                xEventGroupSetBits(event_group_, MAIN_EVENT_NETWORK_DISCONNECTED);
                break;
            case NetworkEvent::Connecting: { ... }
            case NetworkEvent::Connected: {
                std::string msg = Lang::Strings::CONNECTED_TO;
                msg += data;
                display->ShowNotification(msg.c_str(), 30000);
                xEventGroupSetBits(event_group_, MAIN_EVENT_NETWORK_CONNECTED);
                break;
            }
            case NetworkEvent::Disconnected:
                xEventGroupSetBits(event_group_, MAIN_EVENT_NETWORK_DISCONNECTED);
                break;
            // ... 4G modem 状态码:
            case NetworkEvent::ModemDetecting: ...
            case NetworkEvent::ModemErrorNoSim: ...
            case NetworkEvent::ModemErrorRegDenied: ...
            case NetworkEvent::ModemErrorInitFailed: ...
            case NetworkEvent::ModemErrorTimeout: ...
        }
    });
```

注册**网络事件回调**。这个 lambda 会被板子里的 WiFi/4G 状态变化驱动。注意区分：

- 立刻就能在回调上下文里干的（如显示 notification、错误 alert）就直接干；
- 涉及到"切设备状态、起激活任务"等业务动作的，仍然只 set 事件位让主循环处理（如 `MAIN_EVENT_NETWORK_CONNECTED`）。

```
    board.StartNetwork();
    display->UpdateStatusBar(true);
}
```

最后调 `board.StartNetwork()` 让板子去连 WiFi 或拨号 4G（异步——`Initialize()` 不会阻塞等连上）。最后再立刻刷一次状态栏。

Initialize 之后此时此刻：

- 三种事件循环都在跑（主循环还没真正进入 Run，但马上会）；
- 音频任务已起，但事件位都没置，所以麦克风没开、Opus 没编；
- 网络在异步尝试连；
- 屏幕显示 starting。

2.5 `Run()` 主事件循环（166–257 行）

```
void Application::Run() {
    const EventBits_t ALL_EVENTS =
        MAIN_EVENT_SCHEDULE | MAIN_EVENT_SEND_AUDIO | MAIN_EVENT_WAKE_WORD_DETECTED |
        MAIN_EVENT_VAD_CHANGE | MAIN_EVENT_CLOCK_TICK | MAIN_EVENT_ERROR |
        MAIN_EVENT_NETWORK_CONNECTED | MAIN_EVENT_NETWORK_DISCONNECTED |
        MAIN_EVENT_TOGGLE_CHAT | MAIN_EVENT_START_LISTENING | MAIN_EVENT_STOP_LISTENING |
        MAIN_EVENT_ACTIVATION_DONE | MAIN_EVENT_STATE_CHANGED;

    while (true) {
        auto bits = xEventGroupWaitBits(event_group_, ALL_EVENTS, pdTRUE, pdFALSE,
        portMAX_DELAY);
```

`xEventGroupWaitBits(group, bitsToWaitFor, clearOnExit, waitForAll, timeout)`：

- `bitsToWaitFor = ALL_EVENTS`：任何一位置位都唤醒；
- `clearOnExit = pdTRUE`：返回时**自动清掉被触发的位**——所以下一次 wait 不会重复消费同一事件；
- `waitForAll = pdFALSE`：OR 语义（任意一位即可，不是要等所有位都置）；
- `timeout = portMAX_DELAY`：永久阻塞，没事件就睡着；
- 返回值 `bits`：被触发的 bit 掩码，可能多位一起。

接下来按位分支处理：

```
        if (bits & MAIN_EVENT_ERROR) {
            SetDeviceState(kDeviceStateIdle);
            Alert(Lang::Strings::ERROR, last_error_message_.c_str(), "circle_xmark",
        Lang::Sounds::OGG_EXCLAMATION);
        }
```

错误事件：切回 idle + 弹一个 Alert（屏幕显示 + 错误提示音）。

```
        if (bits & MAIN_EVENT_NETWORK_CONNECTED)    HandleNetworkConnectedEvent();
        if (bits & MAIN_EVENT_NETWORK_DISCONNECTED) HandleNetworkDisconnectedEvent();
        if (bits & MAIN_EVENT_ACTIVATION_DONE)      HandleActivationDoneEvent();
        if (bits & MAIN_EVENT_STATE_CHANGED)        HandleStateChangedEvent();
        if (bits & MAIN_EVENT_TOGGLE_CHAT)          HandleToggleChatEvent();
        if (bits & MAIN_EVENT_START_LISTENING)      HandleStartListeningEvent();
        if (bits & MAIN_EVENT_STOP_LISTENING)       HandleStopListeningEvent();
```

这一堆都是单独函数，2.6 节展开。

```
        if (bits & MAIN_EVENT_SEND_AUDIO) {
            while (auto packet = audio_service_.PopPacketFromSendQueue()) {
                if (protocol_ && !protocol_->SendAudio(std::move(packet))) {
                    break;
                }
            }
        }
```

有 Opus 包要发：循环 Pop 直到队空。`SendAudio` 返回 false 就停（多半是网络断了），让下次再试。注意用 `std::move` 转移 `unique_ptr` 所有权。

```
        if (bits & MAIN_EVENT_WAKE_WORD_DETECTED) {
            HandleWakeWordDetectedEvent();
        }
```

```
        if (bits & MAIN_EVENT_VAD_CHANGE) {
            if (GetDeviceState() == kDeviceStateListening) {
                auto led = Board::GetInstance().GetLed();
                led->OnStateChanged();
            }
        }
```

VAD 变化只在 listening 时通知 LED（“我在说话/我没在说话”两种颜色）。其它状态下 VAD 无关紧要。

```
        if (bits & MAIN_EVENT_SCHEDULE) {
            std::unique_lock<std::mutex> lock(mutex_);
            auto tasks = std::move(main_tasks_);
            lock.unlock();
            for (auto& task : tasks) {
                task();
            }
        }
```

`Schedule()` 队列的取出与执行：

1. 拿锁，**整个队列 move 出来**到本地变量（清空 main_tasks_）；
2. 立刻放锁——执行 task 时不持有锁，避免重入死锁（task 里如果再 Schedule，能立即拿到锁追加进新一轮的 main_tasks_）；
3. 顺序执行所有 task。

这种”原子取出+循环执行”是无锁队列消费的经典写法。

```
        if (bits & MAIN_EVENT_CLOCK_TICK) {
            clock_ticks_++;
            auto display = Board::GetInstance().GetDisplay();
            display->UpdateStatusBar();

            if (clock_ticks_ % 10 == 0) {
                SystemInfo::PrintHeapStats();
            }
        }
    }
}
```

1 秒滴答：刷状态栏；每 10 秒（10 个滴答）打印一次堆内存（看有没有内存泄漏）。

2.6 主循环里的事件处理器（一个个讲）

2.6.1 `HandleNetworkConnectedEvent()` (259–282)

```
void Application::HandleNetworkConnectedEvent() {
    ESP_LOGI(TAG, "Network connected");
    auto state = GetDeviceState();

    if (state == kDeviceStateStarting || state == kDeviceStateWifiConfiguring) {
        SetDeviceState(kDeviceStateActivating);
        if (activation_task_handle_ != nullptr) {
            ESP_LOGW(TAG, "Activation task already running");
            return;
        }

        xTaskCreate([](void* arg) {
            Application* app = static_cast<Application*>(arg);
            app->ActivationTask();
            app->activation_task_handle_ = nullptr;
            vTaskDelete(NULL);
        }, "activation", 4096 * 2, this, 2, &activation_task_handle_);
    }

    auto display = Board::GetInstance().GetDisplay();
    display->UpdateStatusBar(true);
}
```

只有”启动中”或”WiFi 配网中”时连上网才触发激活。其它状态（比如已经在用、临时掉线又恢复）就不重启激活流程。

`xTaskCreate(...)` 起一个 8KB 栈的任务跑 `ActivationTask()`，跑完自杀（`vTaskDelete(NULL)`）。再次注意：**用无捕获 lambda 作为 C API 的回调**，把 `this` 通过 `arg` 传进去。

任务优先级 `2`，**比 Application 主任务（默认 5）低**——优先级越大越优先，这样激活时如果有更紧急的事，主循环不会被阻塞。

2.6.2 `HandleNetworkDisconnectedEvent()` (284–295)

```
void Application::HandleNetworkDisconnectedEvent() {
    auto state = GetDeviceState();
    if (state == kDeviceStateConnecting || state == kDeviceStateListening || state ==
        kDeviceStateSpeaking) {
        ESP_LOGI(TAG, "Closing audio channel due to network disconnection");
        protocol_->CloseAudioChannel();
    }
    auto display = Board::GetInstance().GetDisplay();
    display->UpdateStatusBar(true);
}
```

掉线时如果还在通话中，主动关闭音频通道——避免任务在那里干等服务器永远到不来的 ACK。

2.6.3 `HandleActivationDoneEvent()` (297–317)

```
void Application::HandleActivationDoneEvent() {
    SystemInfo::PrintHeapStats();
    SetDeviceState(kDeviceStateIdle);
    has_server_time_ = ota_->HasServerTime();

    auto display = Board::GetInstance().GetDisplay();
    std::string message = std::string(Lang::Strings::VERSION) + ota_->GetCurrentVersion();
    display->ShowNotification(message.c_str());
    display->SetChatMessage("system", "");

    audio_service_.PlaySound(Lang::Sounds::OGG_SUCCESS);

    ota_.reset();   // ★ 释放 OTA 对象，省内存
    auto& board = Board::GetInstance();
    board.SetPowerSaveLevel(PowerSaveLevel::LOW_POWER);
}
```

激活成功后：

- 状态切回 idle；
- 显示版本号通知；
- 播一个”我准备好了”提示音；
- **`ota_.reset()` 释放 Ota 对象**——OTA 检查/激活只在启动时跑一次，之后不需要这个对象占内存；
- 把功耗降回 LOW_POWER（激活过程占性能档以加速握手）。

2.6.4 `ActivationTask()` 后台流程 (319–334)

```
void Application::ActivationTask() {
    ota_ = std::make_unique<Ota>();
    CheckAssetsVersion();
    CheckNewVersion();
    InitializeProtocol();
    xEventGroupSetBits(event_group_, MAIN_EVENT_ACTIVATION_DONE);
}
```

很简洁：

1. new 一个 OTA 对象；
2. 检查资源（字体/表情/唤醒词模型）有没有更新；
3. 检查固件有没有新版本——这里也可能进入激活码循环（详见第 7 章）；

4. 初始化协议层（new 出 WebSocketProtocol 或 MqttProtocol）；
5. 发出 `ACTIVATION_DONE` 事件，主循环接管。

这四步串行执行，里面可能用 `vTaskDelay` 等待，可能阻塞数秒，但因为在独立任务里，不影响主循环。

2.6.5 `CheckAssetsVersion()` (336–392)

精简流程：

1. 查标志位防止重入；
2. 从 NVS 命名空间 `assets` 里读 `download_url`——服务器握手时如果发现需要下新版资源会写进这个 key；
3. 有 URL 就：弹 alert → 切 upgrading 状态 → 提高功耗 → 调 `assets.Download(url, on_progress)`，回调里把"进度% xKB/s"贴到聊天系统消息位；
4. 不论是不是下了新的，最后都 `assets.Apply()` 把分区里的字体/表情/模型应用到 LVGL/ESP-SR；
5. 把表情设成"microchip_ai"（一个"AI 头像"）。

要点：

```
bool success = assets.Download(download_url, [display](int progress, size_t speed) -> void {
    std::thread([display, progress, speed]() {
        char buffer[32];
        snprintf(buffer, sizeof(buffer), "%d%% %uKB/s", progress, speed / 1024);
        display->SetChatMessage("system", buffer);
    }).detach();
});
```

进度回调用 `std::thread().detach()`——**单开一个线程更新 UI**，这样下载任务自己不会被 UI 锁卡住。这里 `std::thread` 在 ESP-IDF 上会创建一个独立 RTOS 任务（pthread 实现）。

2.6.6 `CheckNewVersion()` (394–467) —— 重试退避 + 激活码循环

```
void Application::CheckNewVersion() {
    const int MAX_RETRY = 10;
    int retry_count = 0;
    int retry_delay = 10;

    while (true) {
        esp_err_t err = ota_->CheckVersion();
        if (err != ESP_OK) {
            retry_count++;
            if (retry_count >= MAX_RETRY) return;
            // ... Alert 错误信息
            for (int i = 0; i < retry_delay; i++) {
                vTaskDelay(pdMS_TO_TICKS(1000));
                if (GetDeviceState() == kDeviceStateIdle) break;  // 用户手动退出
            }
            retry_delay *= 2;   // ★ 指数退避
            continue;
        }
        retry_count = 0;
        retry_delay = 10;

        if (ota_->HasNewVersion()) {
            if (UpgradeFirmware(...)) return;
        }

        ota_->MarkCurrentVersionValid();
        if (!ota_->HasActivationCode() && !ota_->HasActivationChallenge()) break;

        if (ota_->HasActivationCode()) {
            ShowActivationCode(ota_->GetActivationCode(), ota_->GetActivationMessage());
        }

        for (int i = 0; i < 10; ++i) {
            esp_err_t err = ota_->Activate();
            if (err == ESP_OK) break;
            else if (err == ESP_ERR_TIMEOUT) vTaskDelay(pdMS_TO_TICKS(3000));
            else vTaskDelay(pdMS_TO_TICKS(10000));
            if (GetDeviceState() == kDeviceStateIdle) break;
        }
    }
}
```

学到的设计：

- **指数退避**：失败一次延迟从 10 秒翻倍，避免对服务器爆雷；
- **可中断**：等待过程中检测状态切回 idle 立刻退出，相当于”用户按了一下”就跳出激活循环；
- **激活码 + 挑战双轨**：服务器可能给你一个 6 位激活码让用户去网页输入，或者一个加密挑战（带 HMAC）让设备直接证明自己是合法设备；两条路并存。

2.6.7 `InitializeProtocol()` (469–606) —— 整个项目的”接线总表”

这一段 130+ 行**是理解整个项目的金钥匙**：把”协议层收到东西”和”业务层该干什么”全部接起来。

```
void Application::InitializeProtocol() {
    auto& board = Board::GetInstance();
    auto display = board.GetDisplay();
    auto codec = board.GetAudioCodec();

    display->SetStatus(Lang::Strings::LOADING_PROTOCOL);

    if (ota_->HasMqttConfig()) {
        protocol_ = std::make_unique<MqttProtocol>();
    } else if (ota_->HasWebsocketConfig()) {
        protocol_ = std::make_unique<WebsocketProtocol>();
    } else {
        ESP_LOGW(TAG, "No protocol specified in the OTA config, using MQTT");
        protocol_ = std::make_unique<MqttProtocol>();
    }
```

工厂选择：握手时服务器告诉你走哪条协议，按需 new 出对应实现。`protocol_` 是 `unique_ptr<Protocol>`，**多态**——后面操作的都是基类接口，不关心是 WS 还是 MQTT。

```
    protocol_->OnConnected([this]() {
        DismissAlert();
    });

    protocol_->OnNetworkError([this](const std::string& message) {
        last_error_message_ = message;
        xEventGroupSetBits(event_group_, MAIN_EVENT_ERROR);
    });
```

连上就关掉之前的提示；出错就 set ERROR 位让主循环弹错。

```
    protocol_->OnIncomingAudio([this](std::unique_ptr<AudioStreamPacket> packet) {
        if (GetDeviceState() == kDeviceStateSpeaking) {
            audio_service_.PushPacketToDecodeQueue(std::move(packet));
        }
    });
```

服务器发音频来：**只有 speaking 状态才接受**。这避免了刚断开但还有 buffer 包延迟到达时被错播。

```
    protocol_->OnAudioChannelOpened([this, codec, &board]() {
        board.SetPowerSaveLevel(PowerSaveLevel::PERFORMANCE);
        if (protocol_->server_sample_rate() != codec->output_sample_rate()) {
            ESP_LOGW(TAG, "Server sample rate %d does not match device output sample rate %d,
        ...");
        }
    });

    protocol_->OnAudioChannelClosed([this, &board]() {
        board.SetPowerSaveLevel(PowerSaveLevel::LOW_POWER);
        Schedule([this]() {
            auto display = Board::GetInstance().GetDisplay();
            display->SetChatMessage("system", "");
            SetDeviceState(kDeviceStateIdle);
        });
    });
```

音频通道开就升性能档，关就降功耗。关闭后清屏并切 idle——但用 `Schedule()` 包一层，把切状态推到主循环上下文。

```
    protocol_->OnIncomingJson([this, display](const cJSON* root) {
        auto type = cJSON_GetObjectItem(root, "type");
        if (strcmp(type->valuestring, "tts") == 0) {
            auto state = cJSON_GetObjectItem(root, "state");
            if (strcmp(state->valuestring, "start") == 0) {
                Schedule([this]() {
                    aborted_ = false;
                    SetDeviceState(kDeviceStateSpeaking);
                });
            } else if (strcmp(state->valuestring, "stop") == 0) {
                Schedule([this]() {
                    if (GetDeviceState() == kDeviceStateSpeaking) {
                        if (listening_mode_ == kListeningModeManualStop) {
                            SetDeviceState(kDeviceStateIdle);
                        } else {
                            SetDeviceState(kDeviceStateListening);
                        }
                    }
                });
            } else if (strcmp(state->valuestring, "sentence_start") == 0) {
                auto text = cJSON_GetObjectItem(root, "text");
                if (cJSON_IsString(text)) {
                    Schedule([this, display, message = std::string(text->valuestring)]() {
                        display->SetChatMessage("assistant", message.c_str());
                    });
                }
            }
        }
        ...
```

JSON 大分发器，按 type 字段：

type	state/data	干什么
`tts`	`start`	进 speaking 状态
	`stop`	speaking 结束 → 根据模式回 idle 或 listening
	`sentence_start` + text	收到一句要说的话文本，贴到 assistant 聊天框
`stt`	text	用户语音被识别成的文字，贴到 user 聊天框
`llm`	emotion	大模型推断的情绪，改表情
`mcp`	payload	委托给 `McpServer::ParseMessage`（详见第 6 章）
`system`	command=reboot	服务器命令设备重启（一般用于强制 OTA 后）
`alert`	status/message/emotion	服务器要弹一条 alert
`custom`（#if CONFIG_RECEIVE_CUSTOM_MESSAGE）	payload	自定义消息，原样贴系统消息

注意所有改 UI/改状态的副作用都包在 `Schedule()` 里，**因为 OnIncomingJson 回调跑在协议线程上**，不能直接动状态机和 UI。

```
    protocol_->Start();
}
```

最后启动协议——开始去连服务器、走鉴权握手等。Start 内部会触发上面注册的 OnConnected。

2.6.8 `ShowActivationCode()` (608–636) —— 把激活码读给用户听

```
void Application::ShowActivationCode(const std::string& code, const std::string& message) {
    struct digit_sound {
        char digit;
        const std::string_view& sound;
    };
    static const std::array<digit_sound, 10> digit_sounds{{
        {'0', Lang::Sounds::OGG_0}, {'1', Lang::Sounds::OGG_1},
        ... {'9', Lang::Sounds::OGG_9}
    }};

    Alert(Lang::Strings::ACTIVATION, message.c_str(), "link", Lang::Sounds::OGG_ACTIVATION);

    for (const auto& digit : code) {
        auto it = std::find_if(digit_sounds.begin(), digit_sounds.end(),
            [digit](const digit_sound& ds) { return ds.digit == digit; });
        if (it != digit_sounds.end()) {
            audio_service_.PlaySound(it->sound);
        }
    }
}
```

弹 alert + 依次播每一位数字的提示音（“1–2–3–4–5–6”）。用 `std::array + std::find_if` 把数字字符映射到 OGG 资源。`PlaySound` 是阻塞的（会把声音排进 playback 队列等播完），所以六位数会顺序播报。

2.6.9 `Alert()` / `DismissAlert()` (638–656)

```
void Application::Alert(const char* status, const char* message, const char* emotion, const
        std::string_view& sound) {
    ESP_LOGW(TAG, "Alert [%s] %s: %s", emotion, status, message);
    auto display = Board::GetInstance().GetDisplay();
    display->SetStatus(status);
    display->SetEmotion(emotion);
    display->SetChatMessage("system", message);
    if (!sound.empty()) {
        audio_service_.PlaySound(sound);
    }
}

void Application::DismissAlert() {
    if (GetDeviceState() == kDeviceStateIdle) {
        auto display = Board::GetInstance().GetDisplay();
        display->SetStatus(Lang::Strings::STANDBY);
        display->SetEmotion("neutral");
        display->SetChatMessage("system", "");
    }
}
```

Alert 是”屏 + 表情 + 声音”四件套同时改的快捷方式；DismissAlert 只在 idle 时清掉提示（其它状态有自己的展示就不动）。

2.6.10 `ToggleChatState/StartListening/StopListening` (658–668) —— 线程安全入口

```
void Application::ToggleChatState() {
    xEventGroupSetBits(event_group_, MAIN_EVENT_TOGGLE_CHAT);
}
void Application::StartListening() {
    xEventGroupSetBits(event_group_, MAIN_EVENT_START_LISTENING);
}
void Application::StopListening() {
    xEventGroupSetBits(event_group_, MAIN_EVENT_STOP_LISTENING);
}
```

就一行——把事件位置上，立即返回。从按键、MCP 工具、网页配网回调任何地方调都安全。真正的逻辑在 `Handle*` 里跑在主循环上下文。

2.6.11 `HandleToggleChatEvent()` (670–705)

```
void Application::HandleToggleChatEvent() {
    auto state = GetDeviceState();

    if (state == kDeviceStateActivating) {
        SetDeviceState(kDeviceStateIdle);
        return;
    } else if (state == kDeviceStateWifiConfiguring) {
        audio_service_.EnableAudioTesting(true);
        SetDeviceState(kDeviceStateAudioTesting);
        return;
    } else if (state == kDeviceStateAudioTesting) {
        audio_service_.EnableAudioTesting(false);
        SetDeviceState(kDeviceStateWifiConfiguring);
        return;
    }

    if (!protocol_) {
        ESP_LOGE(TAG, "Protocol not initialized");
        return;
    }

    if (state == kDeviceStateIdle) {
        if (!protocol_->IsAudioChannelOpened()) {
            SetDeviceState(kDeviceStateConnecting);
            if (!protocol_->OpenAudioChannel()) return;
        }
        SetListeningMode(aec_mode_ == kAecOff ? kListeningModeAutoStop :
        kListeningModeRealtime);
    } else if (state == kDeviceStateSpeaking) {
        AbortSpeaking(kAbortReasonNone);
    } else if (state == kDeviceStateListening) {
        protocol_->CloseAudioChannel();
    }
}
```

按一下按键的语义随当前状态变：

当前状态	按下按键 →
Activating	退出激活码界面回 idle
WifiConfiguring	进音频测试模式（边配网边录音测试）
AudioTesting	退出音频测试
Idle	打开音频通道 + 进 listening
Speaking	打断（abort）
Listening	主动结束这次说话

`SetListeningMode(aec_off ? auto : realtime)`：

- 无 AEC：用 AutoStop——说一句话停下就结束（VAD 控制），不能打断；
- 有 AEC：用 Realtime——可以边说边听服务器返回，类似全双工电话。

2.6.12 `HandleStartListeningEvent` / `HandleStopListeningEvent`（707–752）

跟 ToggleChat 类似但语义不同：

- Start：从 idle 强制进 ManualStop 模式（按住说话），从 speaking 也允许打断进 ManualStop；
- Stop：listening 中 → 发 `SendStopListening` 给服务器并回 idle。

ManualStop 用于”按住说话松开停”，跟物理按键配合。

2.6.13 `HandleWakeWordDetectedEvent()`（754–794）

```
void Application::HandleWakeWordDetectedEvent() {
    if (!protocol_) return;
    auto state = GetDeviceState();

    if (state == kDeviceStateIdle) {
        audio_service_.EncodeWakeWord();   // ★ 把刚才唤醒词那段 PCM 编成 Opus 留着
        if (!protocol_->IsAudioChannelOpened()) {
            SetDeviceState(kDeviceStateConnecting);
            if (!protocol_->OpenAudioChannel()) {
                audio_service_.EnableWakeWordDetection(true);  // 恢复监听
                return;
            }
        }
        auto wake_word = audio_service_.GetLastWakeWord();
#if CONFIG_SEND_WAKE_WORD_DATA
        // 把唤醒词那段音频上传，让服务器知道你说的是哪个唤醒词
        while (auto packet = audio_service_.PopWakeWordPacket()) {
            protocol_->SendAudio(std::move(packet));
        }
        protocol_->SendWakeWordDetected(wake_word);
        SetListeningMode(aec_mode_ == kAecOff ? kListeningModeAutoStop :
        kListeningModeRealtime);
#else
        play_popup_on_listening_ = true;
        SetListeningMode(aec_mode_ == kAecOff ? kListeningModeAutoStop :
        kListeningModeRealtime);
#endif
    } else if (state == kDeviceStateSpeaking) {
        AbortSpeaking(kAbortReasonWakeWordDetected);
    } else if (state == kDeviceStateActivating) {
        SetDeviceState(kDeviceStateIdle);   // 激活中喊唤醒词等于取消激活流程
    }
}
```

要点：

- **`EncodeWakeWord()`**：唤醒词本来用于”判断是否要醒过来”的那段 PCM 不会被丢，而是先编成 Opus 留着。如果 `CONFIG_SEND_WAKE_WORD_DATA=y`，连同唤醒词那一句一起上传，服务器才能做”识别说话人是谁”。
- 说话中喊唤醒词 = 打断（这就是为啥能”打断小智正在说话”）。
- 激活中喊唤醒词 = 取消激活界面回 idle。

`play_popup_on_listening_` 这个 bool 是个**延迟标志**：进 listening 状态后 `HandleStateChangedEvent` 里才播 popup 音效，而不是在这里就播——因为 `EnableVoiceProcessing(true)` 内部会 `ResetDecoder()` 把队列清空，提前播的音会被清掉。这是一个”按事件先后顺序协调副作用”的小巧设计。

2.6.14 `HandleStateChangedEvent()`（796–854）—— 状态切换的”真正动作”

```cpp
void Application::HandleStateChangedEvent() {
    DeviceState new_state = state_machine_.GetState();
    clock_ticks_ = 0;

    auto& board = Board::GetInstance();
    auto display = board.GetDisplay();
    auto led = board.GetLed();
    led->OnStateChanged();   // 灯先变颜色

    switch (new_state) {
        case kDeviceStateUnknown:
        case kDeviceStateIdle:
            display->SetStatus(Lang::Strings::STANDBY);
            display->SetEmotion("neutral");
            audio_service_.EnableVoiceProcessing(false);
            audio_service_.EnableWakeWordDetection(true);   // ★ 重新开唤醒词
            break;
        case kDeviceStateConnecting:
            display->SetStatus(Lang::Strings::CONNECTING);
            display->SetEmotion("neutral");
            display->SetChatMessage("system", "");
            break;
        case kDeviceStateListening:
            display->SetStatus(Lang::Strings::LISTENING);
            display->SetEmotion("neutral");
            if (!audio_service_.IsAudioProcessorRunning()) {
                protocol_->SendStartListening(listening_mode_);
                audio_service_.EnableVoiceProcessing(true);     // ★ 起 AFE
                audio_service_.EnableWakeWordDetection(false);  // 听用户说话期间关闭唤醒检测
            }
            if (play_popup_on_listening_) {
                play_popup_on_listening_ = false;
                audio_service_.PlaySound(Lang::Sounds::OGG_POPUP);  // "叮"
            }
            break;
        case kDeviceStateSpeaking:
            display->SetStatus(Lang::Strings::SPEAKING);
            if (listening_mode_ != kListeningModeRealtime) {
                audio_service_.EnableVoiceProcessing(false);
                // 即使 speaking 也允许 AFE 唤醒词 ("小智停一下"), 但不允许 ESP-SR 兜底唤醒词
                audio_service_.EnableWakeWordDetection(audio_service_.IsAfeWakeWord());
            }
            audio_service_.ResetDecoder();   // 清掉播放队列残留
            break;
        case kDeviceStateWifiConfiguring:
            audio_service_.EnableVoiceProcessing(false);
            audio_service_.EnableWakeWordDetection(false);
            break;
        default:
            break;
    }
}
```

这是把状态变化映射到三个子系统（LED / Display / AudioService）的中央处理器。把整段读懂就基本掌握了项目核心：

- 状态机管 “what state am I in”；
- HandleStateChangedEvent 管 “what should I do when entering this state”；

- 三个子系统 (LED、Display、AudioService) 各管自己的细节。

把"几个状态下分别该做什么"放在一处，状态机以外的代码不用知道这些细节——经典**分层解耦**。

2.6.15 `Schedule()` (856–862) —— 推任务到主循环

```
void Application::Schedule(std::function<void()>&& callback) {
    {
        std::lock_guard<std::mutex> lock(mutex_);
        main_tasks_.push_back(std::move(callback));
    }
    xEventGroupSetBits(event_group_, MAIN_EVENT_SCHEDULE);
}
```

- 拿锁、push、放锁，然后置位；
- 调用方可以是任意 task，绝对线程安全；
- 主循环看到 `MAIN_EVENT_SCHEDULE` 就批量取出执行。

这是整份代码处理"我想在主任务上跑一段代码"的统一入口。配合 lambda 用起来非常顺手。

2.6.16 其余工具方法

`AbortSpeaking(reason)`：发协议 abort 消息。

`SetListeningMode(mode)`：改 mode 后直接 `SetDeviceState(kDeviceStateListening)`——状态机 listener 会触发 `HandleStateChangedEvent` 把音频处理跑起来。

`Reboot()`：

```
void Application::Reboot() {
    if (protocol_ && protocol_->IsAudioChannelOpened()) {
        protocol_->CloseAudioChannel();
    }
    protocol_.reset();
    audio_service_.Stop();
    vTaskDelay(pdMS_TO_TICKS(1000));
    esp_restart();
}
```

清理 → 等 1 秒 → 调 ESP-IDF 的 `esp_restart()` 复位 SoC。

`UpgradeFirmware(url, version)` (890–940):

- 弹"正在升级"alert，播音效；
- 切到 upgrading 状态；
- 性能档 PERFORMANCE；
- 停掉音频服务（节省内存给 OTA 流）；
- 调 `Ota::Upgrade(url, on_progress)` 同步等下载完；
- 成功：重启；失败：恢复音频服务、降功耗、弹错。

`WakeWordInvoke(wake_word)` (942–986)：让外部代码（按键、MCP 工具）模拟一次唤醒。**因为可能从任意上下文调，里面对状态切换都用 `Schedule()` 包一层。**

`CanEnterSleepMode()` (988–1003)：状态 idle 且音频通道关 且音频任务空闲，才能安全进深度睡眠。给板子的 power_save_timer 用。

`SendMcpMessage(payload)`（1005–1012）：

```
void Application::SendMcpMessage(const std::string& payload) {
    Schedule([this, payload = std::move(payload)]() {
        if (protocol_) {
            protocol_->SendMcpMessage(payload);
        }
    });
}
```

被 `McpServer` 调用——发本地工具回应给服务器。用 `Schedule()` 是为了线程安全。lambda 捕获用 `payload = std::move(payload)` 是 C++14 init capture，避免拷贝大字符串。

`SetAecMode(mode)`（1014–1039）：切 AEC 模式时同时更新音频处理器配置 + 改 UI 通知 + 关闭当前音频通道（让下一次开会用新模式）。

`PlaySound(sound)`（1041–1043）：简单透传到 audio_service。

`ResetProtocol()`（1045–1054）：

```
void Application::ResetProtocol() {
    Schedule([this]() {
        if (protocol_ && protocol_->IsAudioChannelOpened()) {
            protocol_->CloseAudioChannel();
        }
        protocol_.reset();
    });
}
```

紧急情况下从任意地方释放协议资源（比如要切换协议、要省内存做 OTA 等）。

2.7 本章用到的核心技术汇总

技术	应用
C++11 magic static	`Application::GetInstance()` 单例
`std::unique_ptr<Base>`	`protocol_`（多态指针）+ `ota_`（生命周期短）
`std::function` + lambda	大量回调挂钩、`Schedule()` 队列、init capture（C++14）
`std::move`	转移 `unique_ptr<AudioStreamPacket>` 所有权
FreeRTOS EventGroup	13 个事件位驱动整个主循环
FreeRTOS xTaskCreate / vTaskDelete	激活后台任务
esp_timer 周期定时	1 秒滴答
`std::mutex` + `lock_guard` + `unique_lock`	`main_tasks_` 队列保护、“原子取出+无锁执行”
C++ 抽象基类 + 多态	`Protocol`、`AudioCodec`、`Display`、`Led` 等
观察者模式	状态机 listener
RAII	`TaskPriorityReset`、`DisplayLockGuard`
JSON 解析（cJSON）	解析所有控制信令
预处理器条件编译	`#if CONFIG_USE_DEVICE_AEC`、`#if CONFIG_SEND_WAKE_WORD_DATA`、`#if CONFIG_RECEIVE_CUSTOM_MESSAGE`
`extern "C"`	`app_main` 兼容 ESP-IDF
指数退避	`CheckNewVersion` 重试
多态指针 + 工厂选择	`InitializeProtocol` 决定 new WS 还是 MQTT

2.8 看完本章你应该掌握的

- `app_main` 30 行做了哪三件事
- Application 单例的 13 个事件位各自代表什么、谁会触发、主循环怎么处理
- Initialize 的 9 步初始化顺序，每一步要点
- Run 主循环的事件分发机制，为什么这种”set 位 + 主循环统一处理”的设计能避免大量锁
- ActivationTask 的串行流程（资源更新 → 版本检查 → 激活码 → 初始化协议）
- InitializeProtocol 的 8 个回调挂钩——这是项目最重要的一段
- HandleStateChangedEvent 把每个状态下三个子系统应该处于什么状态写得很清楚
- `Schedule()` 这个机制的意义——任何后台任务想动主任务的资源都用它
- AEC 模式如何决定 ListeningMode（auto vs realtime），并影响是否允许打断

下一章进入 `device_state_machine.{h,cc}`，把这台状态机本身彻底拆开。

第 3 章 设备状态机：`device_state.h` + `device_state_machine.{h,cc}`

> 三份文件加起来 261 行，是整个项目的”红绿灯指挥官”。本章把状态枚举、合法转换表、观察者机制、线程安全设计、和 `Application` 怎么用它讲清楚。

3.1 三份文件的角色

文件	行数	角色
`device_state.h`	17	只放一个 `enum DeviceState`，11 个值
`device_state_machine.h`	83	状态机类声明
`device_state_machine.cc`	161	状态机实现（转换表 + 观察者通知）

为什么把 enum 单独拆个头？因为很多地方只用 enum 不用 StateMachine（比如 `application.h` 的 `GetDeviceState()` 返回类型只需要 enum，不想 include 整个 StateMachine）——**最小化头文件依赖**。

3.2 `device_state.h` —— 11 个状态枚举

```
enum DeviceState {
    kDeviceStateUnknown,
    kDeviceStateStarting,
    kDeviceStateWifiConfiguring,
    kDeviceStateIdle,
    kDeviceStateConnecting,
    kDeviceStateListening,
    kDeviceStateSpeaking,
    kDeviceStateUpgrading,
    kDeviceStateActivating,
    kDeviceStateAudioTesting,
    kDeviceStateFatalError
};
```

值	含义	设备此时在干什么
`Unknown`	刚 new 出来，从未推进过	状态机初始值，立即应被改成 Starting
`Starting`	启动中	NVS 已初始化、屏幕已开，等网络
`WifiConfiguring`	WiFi 配网中	设备建 AP 或开 BluFi，用户在手机/网页填 SSID + 密码
`Idle`	待机	等唤醒词，可以休眠
`Connecting`	正在连音频通道	主动调 `OpenAudioChannel()` 后到收到 OnConnected 之前
`Listening`	听用户说话	MIC 开、AFE 处理、Opus 编码、上传
`Speaking`	设备说话（播 TTS）	网络下载 Opus、解码、I²S 喂喇叭
`Upgrading`	升级中	OTA 下载新固件 或 下载新 assets.bin
`Activating`	激活中	跑 ActivationTask（拉配置、显示激活码）
`AudioTesting`	音频测试	用于配网期间录音回放，看 MIC/喇叭通了没
`FatalError`	致命错误	单向状态——一旦进入再也出不来

注意命名风格：**Google C++ 风格的 `k` 前缀 + CamelCase**。整个项目 enum 都这么命名。

3.3 `device_state_machine.h` 类声明

完整 83 行已在第 1 章引用过。这里只点关键设计：

3.3.1 公开接口

```
DeviceState GetState() const;                       // 读当前状态
bool TransitionTo(DeviceState new_state);           // 尝试切换
bool CanTransitionTo(DeviceState target) const;     // 试探：能切吗
using StateCallback = std::function<void(DeviceState, DeviceState)>;
int AddStateChangeListener(StateCallback callback); // 加监听者，返 id
void RemoveStateChangeListener(int listener_id);    // 按 id 移除
static const char* GetStateName(DeviceState state); // 调试打印
```

3.3.2 私有成员

```
std::atomic<DeviceState> current_state_{kDeviceStateUnknown};
std::vector<std::pair<int, StateCallback>> listeners_;
int next_listener_id_{0};
std::mutex mutex_;
```

设计要点：

1. **状态本身用 `std::atomic<DeviceState>` 包起来**——读 `GetState()` 不加锁、不阻塞，任意任务都能高频读。
2. **监听者列表用 mutex 保护**，因为可能边遍历边添加/删除。
3. **监听者 ID** 用一个自增整数，加监听者时返回 id；移除时按 id 删。这种 ID 模式比传函数指针/lambda hash 简单。

4. **`StateCallback` 是 `std::function<void(old, new)>`**——签名固定，回调函数可以是 lambda、可以是 bound member function、可以是普通函数指针。

3.3.3 私有方法

```
bool IsValidTransition(DeviceState from, DeviceState to) const;
void NotifyStateChange(DeviceState old_state, DeviceState new_state);
```

转换合法性表 + 通知所有监听者。

3.4 `device_state_machine.cc` 逐段拆解

3.4.1 状态名字符串表（9–22 行）

```
static const char* const STATE_STRINGS[] = {
    "unknown", "starting", "wifi_configuring", "idle", "connecting",
    "listening", "speaking", "upgrading", "activating", "audio_testing",
    "fatal_error", "invalid_state"
};
```

11 个状态对应 11 个字符串，再加一个 `invalid_state` 兜底。

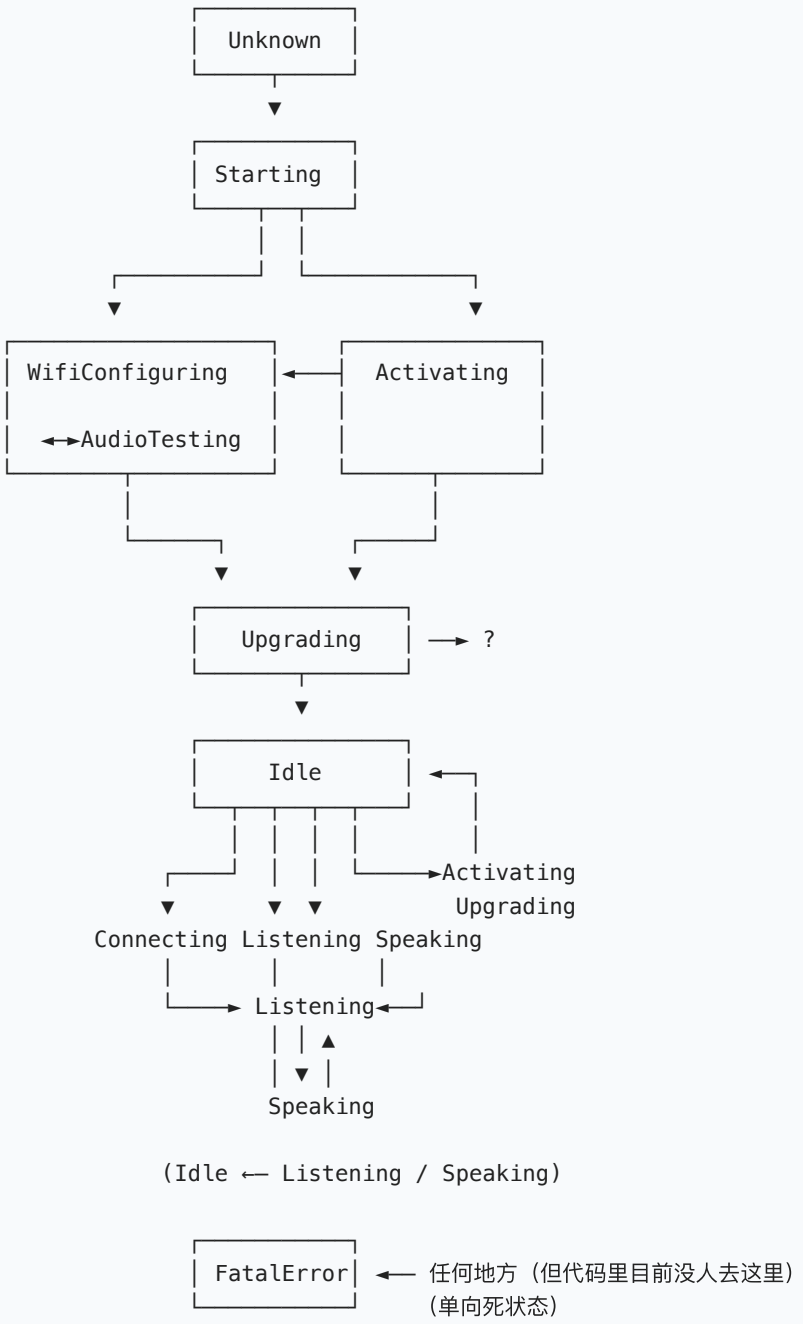
```
const char* DeviceStateMachine::GetStateName(DeviceState state) {
    if (state >= 0 && state <= kDeviceStateFatalError) {
        return STATE_STRINGS[state];
    }
    return STATE_STRINGS[kDeviceStateFatalError + 1];
}
```

边界检查很谨慎：枚举值可能被强转过来不合法，所以做了上下界检查。**注意 enum 默认从 0 开始递增**，所以 `static_cast<int>(kDeviceStateFatalError)` 就是 10，下标取到 `STATE_STRINGS[10] = "fatal_error"`。

3.4.2 合法转换表（34–102 行）—— 整个状态机的核心

```cpp
bool DeviceStateMachine::IsValidTransition(DeviceState from, DeviceState to) const {
    if (from == to) return true;   // ★ 同状态视为合法的 no-op

    switch (from) {
        case kDeviceStateUnknown:
            return to == kDeviceStateStarting;

        case kDeviceStateStarting:
            return to == kDeviceStateWifiConfiguring ||
                   to == kDeviceStateActivating;

        case kDeviceStateWifiConfiguring:
            return to == kDeviceStateActivating ||
                   to == kDeviceStateAudioTesting;

        case kDeviceStateAudioTesting:
            return to == kDeviceStateWifiConfiguring;

        case kDeviceStateActivating:
            return to == kDeviceStateUpgrading ||
                   to == kDeviceStateIdle ||
                   to == kDeviceStateWifiConfiguring;

        case kDeviceStateUpgrading:
            return to == kDeviceStateIdle ||
                   to == kDeviceStateActivating;

        case kDeviceStateIdle:
            return to == kDeviceStateConnecting ||
                   to == kDeviceStateListening ||
                   to == kDeviceStateSpeaking ||
                   to == kDeviceStateActivating ||
                   to == kDeviceStateUpgrading ||
                   to == kDeviceStateWifiConfiguring;

        case kDeviceStateConnecting:
            return to == kDeviceStateIdle ||
                   to == kDeviceStateListening;

        case kDeviceStateListening:
            return to == kDeviceStateSpeaking ||
                   to == kDeviceStateIdle;

        case kDeviceStateSpeaking:
            return to == kDeviceStateListening ||
                   to == kDeviceStateIdle;

        case kDeviceStateFatalError:
            return false;   // ★ 单向死状态

        default:
            return false;
    }
}
```

这就是**状态转换图**的代码表达：



要点：

1. **没人能进入 FatalError 的代码路径**——这个状态在代码里被定义但目前没被设置过，是给以后留的”出大事就锁死设备”的逃生口。
2. **从 idle 出口最多**——idle 是中心枢纽，6 种合法去向。
3. **listening ⇄ speaking 可以双向跳**——支持打断（说话时被新唤醒进 listening）和半双工对话。
4. **WifiConfiguring 和 AudioTesting 互相切**——用于配网时录音测试场景。
5. **Activating → WifiConfiguring**：激活时如果发现配置丢失/无效，回去重新配网。
6. **Upgrading → Activating**：升级失败后回去重试激活。

设计哲学：**用一张表写死所有合法转换**比”在每个调用点判断”清晰得多。后人想改流程只看这一张表即可。

3.4.3 `TransitionTo` 实现（108–131 行）

```
bool DeviceStateMachine::TransitionTo(DeviceState new_state) {
    DeviceState old_state = current_state_.load();

    if (old_state == new_state) return true;   // 同状态 no-op

    if (!IsValidTransition(old_state, new_state)) {
        ESP_LOGW(TAG, "Invalid state transition: %s -> %s",
                 GetStateName(old_state), GetStateName(new_state));
        return false;
    }

    current_state_.store(new_state);
    ESP_LOGI(TAG, "State: %s -> %s",
             GetStateName(old_state), GetStateName(new_state));

    NotifyStateChange(old_state, new_state);
    return true;
}
```

逐行：

1. **`current_state_.load()`** atomic 读，无锁；
2. 同状态直接成功——`Application::SetDeviceState(state)` 经常重复调，避免重复打日志和触发回调；
3. 合法性检查不通过：打 WARN 日志、返回 false（**调用方应该自己看返回值，但代码里大多数地方都没看**——因为合法的状态序列都是经过 application.cc 设计好的，意外非法多半是 bug）；
4. **`current_state_.store(new_state)`** atomic 写；
5. 打 INFO 日志，记录"X -> Y"；
6. 调 `NotifyStateChange` 通知监听者。

潜在并发问题：第 1 步读和第 4 步写之间，可能有别的任务也在 TransitionTo。但是项目里**几乎所有 SetDeviceState 调用都在主循环上下文里**（通过 Schedule()），所以实际不会真的并发。即使并发，atomic 的 load/store 也不会出现"撕裂值"。

3.4.4 监听者增删（133–146 行）

```
int DeviceStateMachine::AddStateChangeListener(StateCallback callback) {
    std::lock_guard<std::mutex> lock(mutex_);
    int id = next_listener_id_++;
    listeners_.emplace_back(id, std::move(callback));
    return id;
}

void DeviceStateMachine::RemoveStateChangeListener(int listener_id) {
    std::lock_guard<std::mutex> lock(mutex_);
    listeners_.erase(
        std::remove_if(listeners_.begin(), listeners_.end(),
            [listener_id](const auto& p) { return p.first == listener_id; }),
        listeners_.end());
}
```

经典的 **erase-remove idiom**——`std::remove_if` 把不满足条件的元素往前挪、要删的元素挪到末尾，返回新逻辑末尾的迭代器，然后 `erase` 真正删除尾巴。

`emplace_back(id, std::move(callback))` 直接在 vector 末尾就地构造一个 `pair<int, StateCallback>`，少一次拷贝。

3.4.5 通知监听者（148–161 行）—— 解锁后调用，避免回调死锁

```
void DeviceStateMachine::NotifyStateChange(DeviceState old_state, DeviceState new_state) {
    std::vector<StateCallback> callbacks_copy;
    {
        std::lock_guard<std::mutex> lock(mutex_);
        callbacks_copy.reserve(listeners_.size());
        for (const auto& [id, cb] : listeners_) {
            callbacks_copy.push_back(cb);
        }
    }

    for (const auto& cb : callbacks_copy) {
        cb(old_state, new_state);
    }
}
```

这是状态机最值得学的设计：

- 第一段：拿锁、**复制**一份所有回调到本地 vector、放锁；
- 第二段：在**没有锁**的情况下挨个调用回调。

为什么不能持锁回调？因为回调里可能再次进入状态机（比如调 `AddStateChangeListener` 或 `TransitionTo`），就会**自己锁自己**（mutex 不可重入）。复制一份后释放锁，回调里随便玩。

代价：每次状态变要复制 N 个 `std::function` 对象。但这里 N 一般只有 1–2 个（实际上 Application 里只有一个监听者），代价可忽略。

如果项目里监听者很多，可以换 `shared_ptr<vector>` 的 copy-on-write 优化。**这里没必要——简单优先。**

`for (const auto& [id, cb] : listeners_)` 用 C++17 **结构化绑定**展开 pair。

3.5 状态机怎么被 `Application` 用？

回顾第 2 章的几个关键点，串起来理解：

3.5.1 注册监听者（application.cc:89–91）

```
state_machine_.AddStateChangeListener([this](DeviceState old_state, DeviceState new_state) {
    xEventGroupSetBits(event_group_, MAIN_EVENT_STATE_CHANGED);
});
```

Application 只有一个监听者——而且它做的事极其简单：set 一个事件位。真正的副作用（点灯、起音频任务、刷 UI）都在 `HandleStateChangedEvent` 里跑在主循环上下文中。

这种”监听者只翻译事件、不做副作用”的设计是状态机能在任意任务里被调用而不出问题的关键。

3.5.2 提交状态变化（application.cc:57–59）

```
bool Application::SetDeviceState(DeviceState state) {
    return state_machine_.TransitionTo(state);
}
```

只是个 trampoline，方便外面调用。返回 false 也很少有人 check（合法转换由开发者保证）。

3.5.3 全项目的 SetDeviceState 调用清单

在哪里	干什么
`application.cc::Initialize` 开头	`kDeviceStateStarting`
`HandleNetworkConnectedEvent`	`kDeviceStateActivating`（开始激活）
`HandleActivationDoneEvent`	`kDeviceStateIdle`（激活完进待机）
`CheckAssetsVersion`	`kDeviceStateUpgrading` / `kDeviceStateActivating`（下资源）
`HandleToggleChat/StartListeningEvent`（在 Idle 时）	`kDeviceStateConnecting`（连音频通道）
`OnIncomingJson tts:start`	`kDeviceStateSpeaking`
`OnIncomingJson tts:stop`	`kDeviceStateIdle` 或 `kDeviceStateListening`
`OnAudioChannelClosed`	`kDeviceStateIdle`（通道断了回待机）
`HandleErrorEvent`	`kDeviceStateIdle`（出错回待机）
`SetListeningMode`	`kDeviceStateListening`（每次设 mode 后必进 listening）
`HandleStopListeningEvent`（在 Listening 时）	`kDeviceStateIdle`
`HandleWakeWordDetectedEvent`（在 Activating 时）	`kDeviceStateIdle`（取消激活）
`UpgradeFirmware`	`kDeviceStateUpgrading`

每一处状态变化都有明确语义。建议读到任何 `SetDeviceState(...)` 时回到第 3.4.2 节那张转换表，对一下”现在在哪、能不能去那”。

3.6 状态机用到的技术清单

技术	应用
`enum` + 命名约定	11 个状态
`std::atomic<T>`	无锁状态读写
`std::function<>` + lambda	监听者回调
`std::vector<std::pair<id, callback>>`	监听者表
`std::mutex` + `std::lock_guard`	监听者表互斥
erase-remove idiom	删除特定 id
C++17 结构化绑定	`for (const auto& [id, cb] : listeners_)`
switch case 合法转换表	状态机核心
解锁后回调	防止 mutex 重入
单向死状态	FatalError
同状态 no-op	重复 set 不重复触发
`ESP_LOGI/W`	标准 ESP-IDF 日志，状态变化全程可见

3.7 一段典型日志（帮你理解整条链路）

打开 ESP-IDF monitor，正常使用一次（说一句"你好小智"，收到回答）会看到大概这样的状态日志：

```
I StateMachine: State: unknown -> starting
I StateMachine: State: starting -> activating
I StateMachine: State: activating -> idle
I Application: Wake word detected: 你好小智
I StateMachine: State: idle -> connecting
I StateMachine: State: connecting -> listening
>> 你好小智
I StateMachine: State: listening -> speaking
<< 你好，有什么可以帮你的?
I StateMachine: State: speaking -> idle    (服务器发完 tts:stop)
```

排查 bug 时**最先看这一行日志**——状态没切到预期值，就是状态机层面的问题。

3.8 看完本章你应该掌握的

- 11 个状态各代表什么
- 合法转换表（能默写主干）
- 状态机为什么用 `atomic + 解锁后回调`（线程安全 + 避免重入）
- 监听者列表用 erase-remove + id 管理
- Application 唯一的监听者只做"翻译成事件位"这件事，副作用都在主循环里
- 看到一行 `State: X -> Y` 日志知道接下来 `HandleStateChangedEvent` 会做什么

下一章进入 `audio/` 子系统——整个项目最复杂、最有学习价值的部分（686 行的 `audio_service.cc` + 多种唤醒词/AFE/codec 实现）。

第 4 章 音频子系统：`main/audio/`

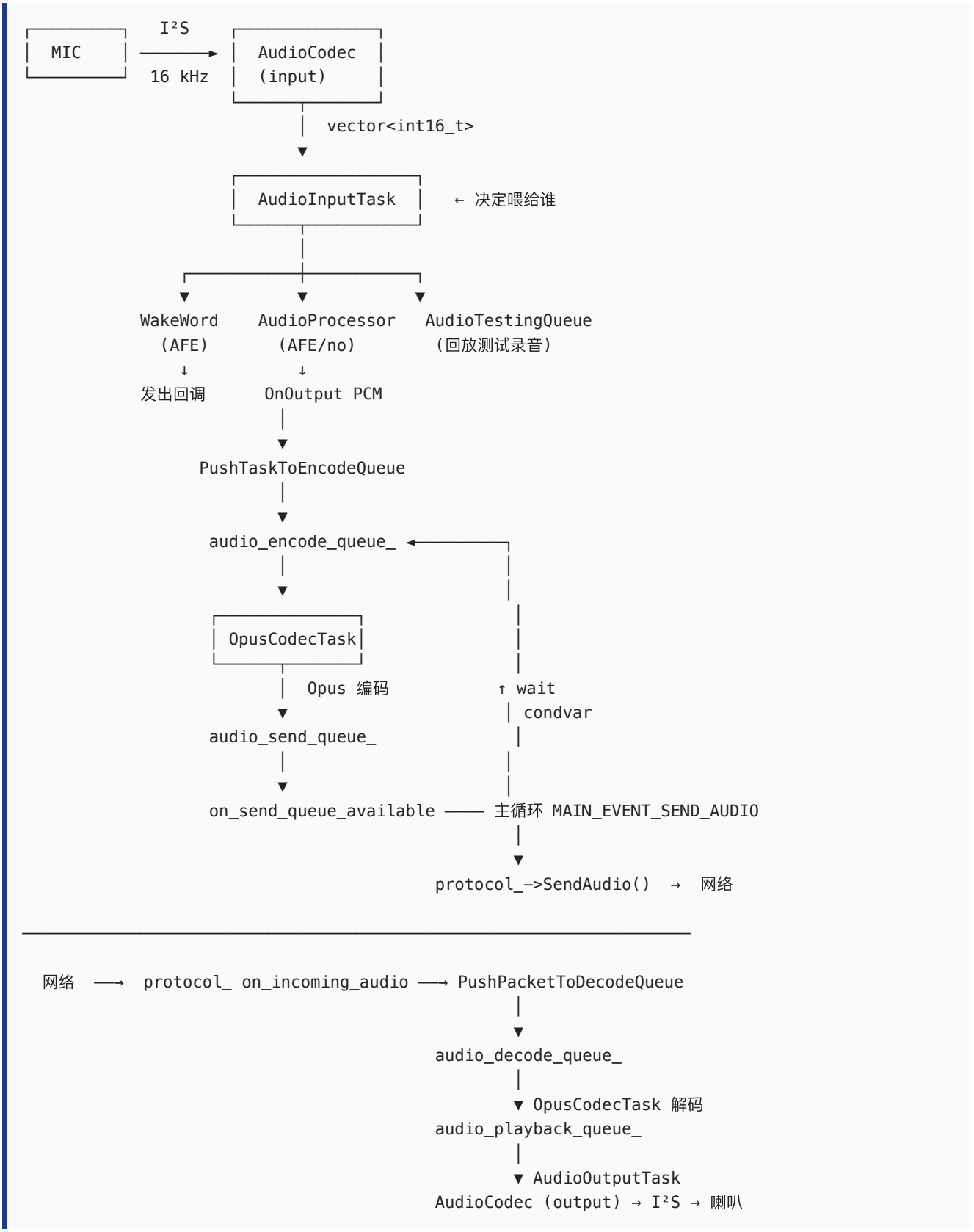
> 这是整个项目最复杂的模块（3338 行）。本章先把整体数据流和任务/队列模型讲清楚，再逐文件展开 `audio_service.cc`（686 行，逐函数），最后讲三个抽象：`AudioCodec`、`AudioProcessor`、`WakeWord` 的设计与具体实现（AFE / ESP-SR / Custom）。

4.1 音频子系统鸟瞰

`main/audio/` 的文件树（已在第 1 章列过，这里只列核心）：

```
audio/
├── audio_codec.{cc,h}            硬件音频外设抽象基类（I²S + 音量 + 增益）
├── audio_service.{cc,h}          ★ 音频任务总调度（本章主战场，686+160 行）
├── audio_processor.h             前端处理器抽象（AFE/无）
├── wake_word.h                   唤醒词检测器抽象
├── codecs/                       I²S codec 芯片驱动（ES8311/ES8388/dummy/no_audio…）
├── processors/                   AFE 实现 + 调试
│   ├── afe_audio_processor.cc    ESP-SR AFE：降噪 + AEC + VAD
│   ├── no_audio_processor.cc     无处理直通
│   └── audio_debugger.cc         把原始 PCM 转发到 UDP 服务给电脑听
├── wake_words/
│   ├── afe_wake_word.cc          基于 ESP-SR AFE 的唤醒（S3/P4）
│   ├── custom_wake_word.cc       基于 ESP-SR multinet（S3/P4 支持任意词）
│   └── esp_wake_word.cc          轻量唤醒（C3 等小芯片）
└── README.md
```

4.1.1 整体数据流



5 个队列（注释见 `audio_service.h`）：

队列	内容	上限	生产者	消费者
`audio_encode_queue_`	`AudioTask`（PCM + 类型 + 时间戳）	2 任务	AudioInputTask / AudioProcessor 回调	OpusCodecTask
`audio_send_queue_`	`AudioStreamPacket`（Opus payload）	40 包（2400/60）	OpusCodecTask	Application 主循环 → protocol
`audio_decode_queue_`	`AudioStreamPacket`（Opus）	40 包	protocol on_incoming_audio	OpusCodecTask
`audio_playback_queue_`	`AudioTask`（PCM）	2 任务	OpusCodecTask	AudioOutputTask
`audio_testing_queue_`	`AudioStreamPacket`	时长上限 10s	AudioInputTask（测试模式）	EnableAudioTesting(false) 时倒灌进 decode 队列回放

`encode/playback` 限到 2 是有意为之——大对象不要堆积；`send/decode` 用 Opus 体积小，可以堆 40 个不怕。

4.1.2 三个常驻 RTOS 任务

`AudioService::Start()` 起 3 个 task：

Task	优先级	栈大小	绑核	职责
`audio_input`	8（高）	6 KB（USE_AUDIO_PROCESSOR）/ 4 KB（无）	core 0（USE_AUDIO_PROCESSOR）/ 不绑	从 codec 读 PCM，喂给唤醒词或处理器或测试队列
`audio_output`	4	4 KB / 2 KB	不绑	从 `audio_playback_queue_` 取 PCM，写 codec
`opus_codec`	2（低）	26 KB	不绑	同时管编码和解码两个方向，按需做

为什么 input 高、codec 低?

- input 是实时硬件 IO：DMA 缓冲容量有限，**取慢了会丢音频**——必须高优先级抢占；
- codec 任务 CPU 重，做编码解码：但 Opus 帧之间没有严格 deadline，慢一点队列变长而已——给它低优先级，让出 CPU 给 input 与协议。

`xTaskCreatePinnedToCore(..., 0)`：把 input 绑到 core 0。S3 双核，core 0 抢实时，core 1 给 Wi-Fi / LVGL。这是嵌入式调度的常见手段。

4.1.3 三个事件位（`audio_service.h`）

```
#define AS_EVENT_AUDIO_TESTING_RUNNING      (1 << 0)
#define AS_EVENT_WAKE_WORD_RUNNING          (1 << 1)
#define AS_EVENT_AUDIO_PROCESSOR_RUNNING    (1 << 2)
#define AS_EVENT_PLAYBACK_NOT_EMPTY         (1 << 3)
```

`AudioInputTask` `xEventGroupWaitBits` 等这三位中任意一个置位才工作；否则永久睡眠（不耗 CPU）。
`Application` 通过 `EnableWakeWordDetection / EnableVoiceProcessing / EnableAudioTesting` 切换这些位。

4.1.4 与外界交互的回调

```
struct AudioServiceCallbacks {
    std::function<void(void)> on_send_queue_available;     // Opus 包就绪
    std::function<void(const std::string&)> on_wake_word_detected;
    std::function<void(bool)> on_vad_change;               // VAD 改变（说话/沉默）
    std::function<void(void)> on_audio_testing_queue_full; // 测试队列满
};
```

每一个都对应 `Application` 主循环一个事件位（见第 2 章 2.4 节）。

4.2 `audio_service.cc` 逐函数讲解

4.2.1 构造 + 析构（21–29 行）

```
AudioService::AudioService() {
    event_group_ = xEventGroupCreate();
}

AudioService::~AudioService() {
    if (event_group_ != nullptr) {
        vEventGroupDelete(event_group_);
    }
}
```

很简单——只建/删事件组。Opus 编解码器和处理器要等 `Initialize` 才创建（要先知道 codec 的采样率）。

4.2.2 `Initialize(codec)`（32–74 行）—— 装配整个音频流水线

```
void AudioService::Initialize(AudioCodec* codec) {
    codec_ = codec;
    codec_->Start();

    opus_decoder_ = std::make_unique<OpusDecoderWrapper>(codec->output_sample_rate(), 1,
        OPUS_FRAME_DURATION_MS);
    opus_encoder_ = std::make_unique<OpusEncoderWrapper>(16000, 1, OPUS_FRAME_DURATION_MS);
    opus_encoder_->SetComplexity(0);
```

- 启动 codec（开 I^2S 通道）；
- new Opus 解码器，采样率跟 codec 输出对齐（默认 24 kHz 或 16 kHz，按板子）；
- new Opus 编码器，固定 16 kHz（**输入采样率统一 16 kHz**，AFE 也按 16 kHz 输出）；
- `SetComplexity(0)` 把复杂度调到最低——ESP32 算力有限，复杂度高了编不过来。Opus 牺牲一点压缩率换实时。

```
    if (codec->input_sample_rate() != 16000) {
        input_resampler_.Configure(codec->input_sample_rate(), 16000);
        reference_resampler_.Configure(codec->input_sample_rate(), 16000);
    }
```

如果 codec 输入不是 16 kHz（某些芯片只支持 48 kHz 等），就配置重采样器。**Reference 通道单独一份**——双通道板子的回参信号也得重采样。

```
#if CONFIG_USE_AUDIO_PROCESSOR
    audio_processor_ = std::make_unique<AfeAudioProcessor>();
#else
    audio_processor_ = std::make_unique<NoAudioProcessor>();
#endif

    audio_processor_->OnOutput([this](std::vector<int16_t>&& data) {
        PushTaskToEncodeQueue(kAudioTaskTypeEncodeToSendQueue, std::move(data));
    });

    audio_processor_->OnVadStateChange([this](bool speaking) {
        voice_detected_ = speaking;
        if (callbacks_.on_vad_change) {
            callbacks_.on_vad_change(speaking);
        }
    });
```

装配处理器——按编译选项选 AFE 或无处理。**两个回调挂钩**：

- `OnOutput`：处理器吐出干净 PCM → 直接推到编码队列；
- `OnVadStateChange`：处理器检测到说话/沉默 → 同步给上层。

```
    esp_timer_create_args_t audio_power_timer_args = {
        .callback = [](void* arg) {
            AudioService* audio_service = (AudioService*)arg;
            audio_service->CheckAndUpdateAudioPowerState();
        },
        .arg = this,
        .dispatch_method = ESP_TIMER_TASK,
        .name = "audio_power_timer",
        .skip_unhandled_events = true,
    };
    esp_timer_create(&audio_power_timer_args, &audio_power_timer_);
}
```

音频功耗 timer——周期性检查麦克风和喇叭"最近一次活跃时间"，超过 15 秒就自动关闭 I²S 通道省电。`CheckAndUpdateAudioPowerState` 见 4.2.16 节。

4.2.3 `Start()` (76–118 行) —— 起三个任务

```cpp
void AudioService::Start() {
    service_stopped_ = false;
    xEventGroupClearBits(event_group_, AS_EVENT_AUDIO_TESTING_RUNNING |
        AS_EVENT_WAKE_WORD_RUNNING | AS_EVENT_AUDIO_PROCESSOR_RUNNING);

    esp_timer_start_periodic(audio_power_timer_, 1000000);

#if CONFIG_USE_AUDIO_PROCESSOR
    xTaskCreatePinnedToCore([](void* arg) {
        AudioService* audio_service = (AudioService*)arg;
        audio_service->AudioInputTask();
        vTaskDelete(NULL);
    }, "audio_input", 2048 * 3, this, 8, &audio_input_task_handle_, 0);

    xTaskCreate([](void* arg) { ...->AudioOutputTask(); ... },
        "audio_output", 2048 * 2, this, 4, &audio_output_task_handle_);
#else
    // 类似但栈小一点、不绑核
#endif

    xTaskCreate([](void* arg) { ...->OpusCodecTask(); ... },
        "opus_codec", 2048 * 13, this, 2, &opus_codec_task_handle_);
}
```

要点：

- 启动前把三个事件位清零，**初始状态什么都不开**——等上层显式 enable；
- 启动音频功耗 timer 周期 1s；
- 起三个任务（注意栈大小：input 6 KB / codec 26 KB / output 4 KB——codec 大是因为 Opus 编解码内部需要不少临时缓冲）；
- 全部用 lambda 作为 `xTaskCreate` 入口，传 `this` 进 arg，任务跑完自己 `vTaskDelete(NULL)`。

4.2.4 `Stop()` (120–133 行) —— 同时清队列 + 唤醒所有阻塞

```cpp
void AudioService::Stop() {
    esp_timer_stop(audio_power_timer_);
    service_stopped_ = true;
    xEventGroupSetBits(event_group_, AS_EVENT_AUDIO_TESTING_RUNNING |
        AS_EVENT_WAKE_WORD_RUNNING |
        AS_EVENT_AUDIO_PROCESSOR_RUNNING);

    std::lock_guard<std::mutex> lock(audio_queue_mutex_);
    audio_encode_queue_.clear();
    audio_decode_queue_.clear();
    audio_playback_queue_.clear();
    audio_testing_queue_.clear();
    audio_queue_cv_.notify_all();
}
```

精妙之处：

- `service_stopped_ = true` 先设上；
- 然后把三个事件位**全 set 一遍**——这会让正在 `xEventGroupWaitBits` 的 input task 立刻返回，发现 `service_stopped_` 后 break；

- 同时 `notify_all` 把卡在 condvar 上的 output/codec 任务也唤醒，它们 wait 谓词里也会查 `service_stopped_` 然后退出。

“既要保证睡着的任务能被叫醒，又要让它叫醒后立刻知道该退出”是 RTOS 程序优雅停机的标准套路。

4.2.5 `ReadAudioData()` (135–188 行) —— 从 codec 读，按需重采样、拆通道

这是从硬件读音频的统一入口，被 input task 多次调用。

```
bool AudioService::ReadAudioData(std::vector<int16_t>& data, int sample_rate, int samples) {
    if (!codec_->input_enabled()) {
        esp_timer_stop(audio_power_timer_);
        esp_timer_start_periodic(audio_power_timer_, AUDIO_POWER_CHECK_INTERVAL_MS * 1000);
        codec_->EnableInput(true);
    }
```

第一段：如果 codec 输入是关的（之前被功耗策略关了），现在重新开起来；同时把功耗 timer 改成 1 秒频率（活跃时更频繁检查）。

```
    if (codec_->input_sample_rate() != sample_rate) {
        data.resize(samples * codec_->input_sample_rate() / sample_rate * codec_-
        >input_channels());
        if (!codec_->InputData(data)) return false;
```

如果 codec 采样率跟请求的不一样，先按 codec 原生采样率读够样本数，**之后再重采样到 16 kHz**。

```
        if (codec_->input_channels() == 2) {
            // 拆 MIC 和 reference 两路，分别重采样
            auto mic_channel = std::vector<int16_t>(data.size() / 2);
            auto reference_channel = std::vector<int16_t>(data.size() / 2);
            for (size_t i = 0, j = 0; i < mic_channel.size(); ++i, j += 2) {
                mic_channel[i] = data[j];
                reference_channel[i] = data[j + 1];
            }
            auto resampled_mic = std::vector<int16_t>
        (input_resampler_.GetOutputSamples(mic_channel.size()));
            auto resampled_reference = std::vector<int16_t>
        (reference_resampler_.GetOutputSamples(reference_channel.size()));
            input_resampler_.Process(mic_channel.data(), mic_channel.size(),
        resampled_mic.data());
            reference_resampler_.Process(reference_channel.data(), reference_channel.size(),
        resampled_reference.data());
            // 交错回 [mic, ref, mic, ref, ...]
            data.resize(resampled_mic.size() + resampled_reference.size());
            for (size_t i = 0, j = 0; i < resampled_mic.size(); ++i, j += 2) {
                data[j] = resampled_mic[i];
                data[j + 1] = resampled_reference[i];
            }
        } else {
            // 单通道：直接重采样
            auto resampled = std::vector<int16_t>
        (input_resampler_.GetOutputSamples(data.size()));
            input_resampler_.Process(data.data(), data.size(), resampled.data());
            data = std::move(resampled);
        }
    } else {
        data.resize(samples * codec_->input_channels());
        if (!codec_->InputData(data)) return false;
    }

    last_input_time_ = std::chrono::steady_clock::now();
    debug_statistics_.input_count++;

#if CONFIG_USE_AUDIO_DEBUGGER
    if (audio_debugger_ == nullptr) {
        audio_debugger_ = std::make_unique<AudioDebugger>();
    }
    audio_debugger_->Feed(data);
#endif

    return true;
}
```

关键点：

- 双通道 = 主麦克风 + 回参（refrence）。回参通常接到喇叭功放的输出，AEC 算法需要它知道”刚才喇叭放了什么”；
- 单通道板子没有回参（无法做硬件级 AEC）；
- 数据布局始终是 **interleaved**（交错），AFE 自己会去拆；
- `last_input_time_` 给功耗 timer 用；
- `AudioDebugger` 在 `CONFIG_USE_AUDIO_DEBUGGER=y` 时把原始 PCM 转 UDP 发出去，电脑跑 `scripts/audio_debug_server.py` 接收听效果。

4.2.6 `AudioInputTask()`（190–257 行）—— 输入任务主循环

```
void AudioService::AudioInputTask() {
    while (true) {
        EventBits_t bits = xEventGroupWaitBits(event_group_, AS_EVENT_AUDIO_TESTING_RUNNING |
            AS_EVENT_WAKE_WORD_RUNNING | AS_EVENT_AUDIO_PROCESSOR_RUNNING,
            pdFALSE, pdFALSE, portMAX_DELAY);
```

- `clearOnExit=pdFALSE`：**不清位**——任务下次进来还能看到状态（事件位等同于”模式标志”，不是”一次性事件”）；
- `waitForAll=pdFALSE`：任意一位即可；
- 阻塞等任意位置上。

```
        if (service_stopped_) break;
        if (audio_input_need_warmup_) {
            audio_input_need_warmup_ = false;
            vTaskDelay(pdMS_TO_TICKS(120));
            continue;
        }
```

warmup（暖机）逻辑：刚切到 listening 时麦克风刚启动，前 120ms 可能有大爆音（DMA 缓冲带噪），延迟一下让硬件稳定下来再读。

```
        if (bits & AS_EVENT_AUDIO_TESTING_RUNNING) {
            if (audio_testing_queue_.size() >= AUDIO_TESTING_MAX_DURATION_MS /
        OPUS_FRAME_DURATION_MS) {
                ESP_LOGW(TAG, "Audio testing queue is full, stopping audio testing");
                EnableAudioTesting(false);
                continue;
            }
            std::vector<int16_t> data;
            int samples = OPUS_FRAME_DURATION_MS * 16000 / 1000;
            if (ReadAudioData(data, 16000, samples)) {
                if (codec_->input_channels() == 2) {
                    auto mono_data = std::vector<int16_t>(data.size() / 2);
                    for (size_t i = 0, j = 0; i < mono_data.size(); ++i, j += 2) {
                        mono_data[i] = data[j];
                    }
                    data = std::move(mono_data);
                }
                PushTaskToEncodeQueue(kAudioTaskTypeEncodeToTestingQueue, std::move(data));
                continue;
            }
        }
```

音频测试模式：录最多 10 秒 PCM，编成 Opus 推到 testing 队列。配网时按 BOOT 键开测试，再按一次关掉，关时把队列倒灌进 decode 队列回放——这样用户能听到”麦克风→喇叭”通了没。

```
        if (bits & AS_EVENT_WAKE_WORD_RUNNING) {
            std::vector<int16_t> data;
            int samples = wake_word_->GetFeedSize();
            if (samples > 0) {
                if (ReadAudioData(data, 16000, samples)) {
                    wake_word_->Feed(data);
                    continue;
                }
            }
        }

        if (bits & AS_EVENT_AUDIO_PROCESSOR_RUNNING) {
            std::vector<int16_t> data;
            int samples = audio_processor_->GetFeedSize();
            if (samples > 0) {
                if (ReadAudioData(data, 16000, samples)) {
                    audio_processor_->Feed(std::move(data));
                    continue;
                }
            }
        }
```

这里有个隐含的优先级：

- testing 优先于唤醒词（测试模式下不会触发唤醒）；
- 唤醒词和处理器**互斥**——一次只开一个。空闲时只开唤醒词；进 listening 后关唤醒词、开处理器；speaking 时关处理器、（AFE 时）允许唤醒词检测打断。

`GetFeedSize()` 由对应组件返回它需要的样本数（chunk size，例如 AFE 一般要 16ms = 256 样本/16 kHz）。**Read 多少 feed 多少**——不浪费数据也不积压。

```
        ESP_LOGE(TAG, "Should not be here, bits: %lx", bits);
        break;
    }
    ESP_LOGW(TAG, "Audio input task stopped");
}
```

如果三个位都没成功消费，打错误日志并退出——理论上不会到这里。

4.2.7 `AudioOutputTask()` (259–293 行) —— 条件变量等播放

```
void AudioService::AudioOutputTask() {
    while (true) {
        std::unique_lock<std::mutex> lock(audio_queue_mutex_);
        audio_queue_cv_.wait(lock, [this]() { return !audio_playback_queue_.empty() ||
        service_stopped_; });
        if (service_stopped_) break;

        auto task = std::move(audio_playback_queue_.front());
        audio_playback_queue_.pop_front();
        audio_queue_cv_.notify_all();
        lock.unlock();

        if (!codec_->output_enabled()) {
            esp_timer_stop(audio_power_timer_);
            esp_timer_start_periodic(audio_power_timer_, AUDIO_POWER_CHECK_INTERVAL_MS *
        1000);
            codec_->EnableOutput(true);
        }
        codec_->OutputData(task->pcm);

        last_output_time_ = std::chrono::steady_clock::now();
        debug_statistics_.playback_count++;

#if CONFIG_USE_SERVER_AEC
        if (task->timestamp > 0) {
            lock.lock();
            timestamp_queue_.push_back(task->timestamp);
        }
#endif
    }
    ESP_LOGW(TAG, "Audio output task stopped");
}
```

- 用条件变量 `audio_queue_cv_` 而不是事件组——更适合”等队列非空”这种谓词；
- `wait` 的 lambda 谓词是 C++11 condition_variable 的推荐用法（**虚假唤醒安全**）；
- 取到 task 立刻释放锁——`codec_->OutputData()` 是阻塞 I²S 写，可能耗几十毫秒，不能持锁；
- I²S 输出按需开启，进入功耗 timer 监控；
- **`#if CONFIG_USE_SERVER_AEC`**：服务器 AEC 模式下把这一帧的时间戳记录下来，等下次 input 时配对发回——服务器拿这两个时间戳就能在你的输入里减掉自己刚发的 TTS。

4.2.8 `OpusCodecTask()`（295–372 行）—— 双向编解码

```
void AudioService::OpusCodecTask() {
    while (true) {
        std::unique_lock<std::mutex> lock(audio_queue_mutex_);
        audio_queue_cv_.wait(lock, [this]() {
            return service_stopped_ ||
                (!audio_encode_queue_.empty() && audio_send_queue_.size() <
        MAX_SEND_PACKETS_IN_QUEUE) ||
                (!audio_decode_queue_.empty() && audio_playback_queue_.size() <
        MAX_PLAYBACK_TASKS_IN_QUEUE);
        });
        if (service_stopped_) break;
```

条件变量的谓词三选一：

- 服务停了；

- 编码侧：encode 队列非空且 send 队列没满；
- 解码侧：decode 队列非空且 playback 队列没满。

`send/playback 没满` 这一条保证不会无限堆积——背压（back pressure）。

```
        if (!audio_decode_queue_.empty() && audio_playback_queue_.size() <
        MAX_PLAYBACK_TASKS_IN_QUEUE) {
            auto packet = std::move(audio_decode_queue_.front());
            audio_decode_queue_.pop_front();
            audio_queue_cv_.notify_all();
            lock.unlock();

            auto task = std::make_unique<AudioTask>();
            task->type = kAudioTaskTypeDecodeToPlaybackQueue;
            task->timestamp = packet->timestamp;

            SetDecodeSampleRate(packet->sample_rate, packet->frame_duration);
            if (opus_decoder_->Decode(std::move(packet->payload), task->pcm)) {
                if (opus_decoder_->sample_rate() != codec_->output_sample_rate()) {
                    int target_size = output_resampler_.GetOutputSamples(task->pcm.size());
                    std::vector<int16_t> resampled(target_size);
                    output_resampler_.Process(task->pcm.data(), task->pcm.size(),
        resampled.data());
                    task->pcm = std::move(resampled);
                }
                lock.lock();
                audio_playback_queue_.push_back(std::move(task));
                audio_queue_cv_.notify_all();
            } else {
                ESP_LOGE(TAG, "Failed to decode audio");
                lock.lock();
            }
            debug_statistics_.decode_count++;
        }
```

解码侧：

1. 取出 decode 任务、释放锁、解码（可能耗 5–20ms，绝不持锁）；
2. 如果解码器采样率跟 codec 输出不一致就重采样；
3. 重新拿锁，推到 playback 队列。

`SetDecodeSampleRate()`：服务器下行 Opus 可能采样率不同（24 kHz、16 kHz、48 kHz），解码器要按帧重建。

```
        if (!audio_encode_queue_.empty() && audio_send_queue_.size() <
        MAX_SEND_PACKETS_IN_QUEUE) {
            auto task = std::move(audio_encode_queue_.front());
            audio_encode_queue_.pop_front();
            audio_queue_cv_.notify_all();
            lock.unlock();

            auto packet = std::make_unique<AudioStreamPacket>();
            packet->frame_duration = OPUS_FRAME_DURATION_MS;
            packet->sample_rate = 16000;
            packet->timestamp = task->timestamp;
            if (!opus_encoder_->Encode(std::move(task->pcm), packet->payload)) {
                ESP_LOGE(TAG, "Failed to encode audio");
                continue;
            }

            if (task->type == kAudioTaskTypeEncodeToSendQueue) {
                {
                    std::lock_guard<std::mutex> lock(audio_queue_mutex_);
                    audio_send_queue_.push_back(std::move(packet));
                }
                if (callbacks_.on_send_queue_available) {
                    callbacks_.on_send_queue_available();   // ★ 通知主循环
                }
            } else if (task->type == kAudioTaskTypeEncodeToTestingQueue) {
                std::lock_guard<std::mutex> lock(audio_queue_mutex_);
                audio_testing_queue_.push_back(std::move(packet));
            }
            debug_statistics_.encode_count++;
            lock.lock();
        }
    }
}
```

编码侧：

1. 取出 encode 任务、释放锁、Opus 编码；
2. 根据 task 类型：发送队列 / 测试队列；
3. 推送送队列后调 `on_send_queue_available` 触发 `MAIN_EVENT_SEND_AUDIO`（详见第 2 章 2.5）。

注意 send 队列推完后**调回调时不能持锁**——回调可能反过来锁主循环资源造成死锁。

4.2.9 `SetDecodeSampleRate()`（374–387）—— 服务端采样率自适应

```
void AudioService::SetDecodeSampleRate(int sample_rate, int frame_duration) {
    if (opus_decoder_->sample_rate() == sample_rate && opus_decoder_->duration_ms() ==
        frame_duration) {
        return;
    }
    opus_decoder_.reset();
    opus_decoder_ = std::make_unique<OpusDecoderWrapper>(sample_rate, 1, frame_duration);

    auto codec = Board::GetInstance().GetAudioCodec();
    if (opus_decoder_->sample_rate() != codec->output_sample_rate()) {
        ESP_LOGI(TAG, "Resampling audio from %d to %d", opus_decoder_->sample_rate(), codec-
        >output_sample_rate());
        output_resampler_.Configure(opus_decoder_->sample_rate(), codec-
        >output_sample_rate());
    }
}
```

只有变化时才重建解码器（避免每包都新建消耗 CPU）。重建后顺便重配输出重采样器。

4.2.10 `PushTaskToEncodeQueue()` (389–410) —— 配对时间戳

```
void AudioService::PushTaskToEncodeQueue(AudioTaskType type, std::vector<int16_t>&& pcm) {
    auto task = std::make_unique<AudioTask>();
    task->type = type;
    task->pcm = std::move(pcm);

    std::unique_lock<std::mutex> lock(audio_queue_mutex_);

    if (type == kAudioTaskTypeEncodeToSendQueue && !timestamp_queue_.empty()) {
        if (timestamp_queue_.size() <= MAX_TIMESTAMPS_IN_QUEUE) {
            task->timestamp = timestamp_queue_.front();
        } else {
            ESP_LOGW(TAG, "Timestamp queue (%u) is full, dropping timestamp",
        timestamp_queue_.size());
        }
        timestamp_queue_.pop_front();
    }

    audio_queue_cv_.wait(lock, [this]() { return audio_encode_queue_.size() <
        MAX_ENCODE_TASKS_IN_QUEUE; });
    audio_encode_queue_.push_back(std::move(task));
    audio_queue_cv_.notify_all();
}
```

服务器 AEC 时间戳对齐机制：

- TTS 播放（`AudioOutputTask`）时记录”这一帧的时间戳”到 `timestamp_queue_`；
- 后续麦克风采到的 PCM 通过 `PushTaskToEncodeQueue` 推到编码队列时，从 timestamp 队列前端取一个配对——意味着”这一帧麦克风录到的，对应那一帧 TTS 播出的时间”；
- 服务器拿到这个配对的时间戳，就能在你的输入里减掉它刚才发的 TTS。

`wait` 等队列未满：如果 encode 队列满了，这一帧 PCM 就阻塞等空位——**背压传递**到上游（处理器），让它别堆积。

4.2.11 `PushPacketToDecodeQueue()` (412–424) —— 上层入队接口

```
bool AudioService::PushPacketToDecodeQueue(std::unique_ptr<AudioStreamPacket> packet, bool
        wait) {
    std::unique_lock<std::mutex> lock(audio_queue_mutex_);
    if (audio_decode_queue_.size() >= MAX_DECODE_PACKETS_IN_QUEUE) {
        if (wait) {
            audio_queue_cv_.wait(lock, [this]() { return audio_decode_queue_.size() <
        MAX_DECODE_PACKETS_IN_QUEUE; });
        } else {
            return false;
        }
    }
    audio_decode_queue_.push_back(std::move(packet));
    audio_queue_cv_.notify_all();
    return true;
}
```

- 协议层（WebSocket/MQTT）调这个把服务器发来的 Opus 包入队；
- `wait` 参数控制满了是阻塞还是直接丢——网络回调一般传 false（要尽快返回，丢一帧也没办法），`PlaySound` 传 true（必须播完）。

4.2.12 `PopPacketFromSendQueue()`（426–435）—— 主循环出队

```
std::unique_ptr<AudioStreamPacket> AudioService::PopPacketFromSendQueue() {
    std::lock_guard<std::mutex> lock(audio_queue_mutex_);
    if (audio_send_queue_.empty()) return nullptr;
    auto packet = std::move(audio_send_queue_.front());
    audio_send_queue_.pop_front();
    audio_queue_cv_.notify_all();
    return packet;
}
```

主循环在 `MAIN_EVENT_SEND_AUDIO` 事件里不停 Pop，直到 nullptr 为止。

4.2.13 唤醒词三件套（437–475）

```
void AudioService::EncodeWakeWord() {
    if (wake_word_) wake_word_->EncodeWakeWordData();
}

const std::string& AudioService::GetLastWakeWord() const {
    return wake_word_->GetLastDetectedWakeWord();
}

std::unique_ptr<AudioStreamPacket> AudioService::PopWakeWordPacket() {
    auto packet = std::make_unique<AudioStreamPacket>();
    if (wake_word_->GetWakeWordOpus(packet->payload)) {
        return packet;
    }
    return nullptr;
}

void AudioService::EnableWakeWordDetection(bool enable) {
    if (!wake_word_) return;
    if (enable) {
        if (!wake_word_initialized_) {
            if (!wake_word_->Initialize(codec_, models_list_)) {
                ESP_LOGE(TAG, "Failed to initialize wake word");
                return;
            }
            wake_word_initialized_ = true;
        }
        wake_word_->Start();
        xEventGroupSetBits(event_group_, AS_EVENT_WAKE_WORD_RUNNING);
    } else {
        wake_word_->Stop();
        xEventGroupClearBits(event_group_, AS_EVENT_WAKE_WORD_RUNNING);
    }
}
```

唤醒词模型很大（几百 KB 模型权重），**首次 Enable 才 Initialize**——开机不立刻加载，省启动时间。

4.2.14 `EnableVoiceProcessing()` (477–494)

```
void AudioService::EnableVoiceProcessing(bool enable) {
    if (enable) {
        if (!audio_processor_initialized_) {
            audio_processor_->Initialize(codec_, OPUS_FRAME_DURATION_MS, models_list_);
            audio_processor_initialized_ = true;
        }
        ResetDecoder();                  // ★ 清掉播放队列残留
        audio_input_need_warmup_ = true; // ★ 下次输入做 120ms 暖机
        audio_processor_->Start();
        xEventGroupSetBits(event_group_, AS_EVENT_AUDIO_PROCESSOR_RUNNING);
    } else {
        audio_processor_->Stop();
        xEventGroupClearBits(event_group_, AS_EVENT_AUDIO_PROCESSOR_RUNNING);
    }
}
```

进入 listening 状态时：

- AFE 第一次 Initialize（耗时几十 ms，懒加载）；

- **ResetDecoder() 清掉 playback 队列**——避免上一次 TTS 残留还在播；
- **设置暖机标志**——下次 input task 进来先 delay 120ms 再读；
- 启动 AFE 任务并置位 PROCESSOR_RUNNING。

4.2.15 `EnableAudioTesting()` (496–507) —— 测试模式开关

```
void AudioService::EnableAudioTesting(bool enable) {
    if (enable) {
        xEventGroupSetBits(event_group_, AS_EVENT_AUDIO_TESTING_RUNNING);
    } else {
        xEventGroupClearBits(event_group_, AS_EVENT_AUDIO_TESTING_RUNNING);
        std::lock_guard<std::mutex> lock(audio_queue_mutex_);
        audio_decode_queue_ = std::move(audio_testing_queue_);   // ★ 倒灌
        audio_queue_cv_.notify_all();
    }
}
```

测试关闭瞬间，**把 testing 队列里录的所有 Opus 整体 move 进 decode 队列**——codec 任务自然就解码 + 播放出来。一行代码完成回放，**move 而不是 copy**，零开销。

4.2.16 `CheckAndUpdateAudioPowerState()` (637–650) —— 自动省电

```
void AudioService::CheckAndUpdateAudioPowerState() {
    auto now = std::chrono::steady_clock::now();
    auto input_elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now -
        last_input_time_).count();
    auto output_elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(now -
        last_output_time_).count();
    if (input_elapsed > AUDIO_POWER_TIMEOUT_MS && codec_->input_enabled()) {
        codec_->EnableInput(false);
    }
    if (output_elapsed > AUDIO_POWER_TIMEOUT_MS && codec_->output_enabled()) {
        codec_->EnableOutput(false);
    }
    if (!codec_->input_enabled() && !codec_->output_enabled()) {
        esp_timer_stop(audio_power_timer_);  // 两边都关了就连 timer 也停掉
    }
}
```

15 秒（`AUDIO_POWER_TIMEOUT_MS=15000`）没动静就关 I²S 通道。再来活动时 `ReadAudioData` / output task 会重新开。再三停 timer 节省更深一档的功耗。

4.2.17 `PlaySound()` (523–620) —— 在线解析 OGG 容器

这一段独特：项目把多种提示音以 OGG 文件形式打进固件，运行期**在 RAM 里直接解析 OGG 容器、抽出 Opus 包**喂解码器。

```cpp
void AudioService::PlaySound(const std::string_view& ogg) {
    if (!codec_->output_enabled()) {
        codec_->EnableOutput(true);
    }
    const uint8_t* buf = reinterpret_cast<const uint8_t*>(ogg.data());
    size_t size = ogg.size();
    size_t offset = 0;

    auto find_page = [&](size_t start)->size_t {
        for (size_t i = start; i + 4 <= size; ++i) {
            if (buf[i] == 'O' && buf[i+1] == 'g' && buf[i+2] == 'g' && buf[i+3] == 'S') return
        i;
        }
        return static_cast<size_t>(-1);
    };

    bool seen_head = false;
    bool seen_tags = false;
    int sample_rate = 16000;

    while (true) {
        size_t pos = find_page(offset);
        if (pos == static_cast<size_t>(-1)) break;
        offset = pos;
        if (offset + 27 > size) break;
        // OGG page: [27 bytes header] [page_segments] [body]
        const uint8_t* page = buf + offset;
        uint8_t page_segments = page[26];
        size_t seg_table_off = offset + 27;
        if (seg_table_off + page_segments > size) break;

        size_t body_size = 0;
        for (size_t i = 0; i < page_segments; ++i) body_size += page[27 + i];
        size_t body_off = seg_table_off + page_segments;
        if (body_off + body_size > size) break;

        // 按 lacing 表把 body 切成多个包
        size_t cur = body_off;
        size_t seg_idx = 0;
        while (seg_idx < page_segments) {
            size_t pkt_len = 0;
            size_t pkt_start = cur;
            bool continued = false;
            do {
                uint8_t l = page[27 + seg_idx++];
                pkt_len += l;
                cur += l;
                continued = (l == 255);
            } while (continued && seg_idx < page_segments);
            if (pkt_len == 0) continue;
            const uint8_t* pkt_ptr = buf + pkt_start;

            if (!seen_head) {
                // 第一个 packet 是 OpusHead
                if (pkt_len >= 19 && std::memcmp(pkt_ptr, "OpusHead", 8) == 0) {
                    seen_head = true;
                    uint8_t version = pkt_ptr[8];
                    uint8_t channel_count = pkt_ptr[9];
```

```
                    if (pkt_len >= 16) {
                        sample_rate = pkt_ptr[12] | (pkt_ptr[13] << 8) |
                                      (pkt_ptr[14] << 16) | (pkt_ptr[15] << 24);
                    }
                }
                continue;
            }
            if (!seen_tags) {
                // 第二个 packet 是 OpusTags
                if (pkt_len >= 8 && std::memcmp(pkt_ptr, "OpusTags", 8) == 0) {
                    seen_tags = true;
                }
                continue;
            }

            // 后续都是 Opus 音频包
            auto packet = std::make_unique<AudioStreamPacket>();
            packet->sample_rate = sample_rate;
            packet->frame_duration = 60;
            packet->payload.resize(pkt_len);
            std::memcpy(packet->payload.data(), pkt_ptr, pkt_len);
            PushPacketToDecodeQueue(std::move(packet), true);
        }
        offset = body_off + body_size;
    }
}
```

要点：

- **手写 OGG 容器解析器**——OGG 是一种封装格式，里面装 Opus；
- 每个 OGG page 27 字节 header + segment 表 + body；
- 第一个 packet 必须是 OpusHead（带采样率），第二个是 OpusTags（元数据），之后才是音频；
- 把音频包直接推到 decode 队列，复用整个解码 → 播放路径；
- `wait=true`：必须把每一帧都推进去，否则提示音会断。

这种"运行期解析"省去了离线转 RAW 的麻烦，OGG 文件可以直接从 `scripts/mp3_to_ogg.sh` 转出来用。

4.2.18 `ResetDecoder()` (627–635) —— 切状态时的清理

```
void AudioService::ResetDecoder() {
    std::lock_guard<std::mutex> lock(audio_queue_mutex_);
    opus_decoder_->ResetState();
    timestamp_queue_.clear();
    audio_decode_queue_.clear();
    audio_playback_queue_.clear();
    audio_testing_queue_.clear();
    audio_queue_cv_.notify_all();
}
```

进 listening 或 speaking 时调用——把所有下行音频残留清干净。`Opus::ResetState` 是必要的，**Opus 是有状态的**（前向参考），不清理会导致下次解码有"杂音"。

4.2.19 `SetModelsList()` + `IsAfeWakeWord()` (652–686)

```
void AudioService::SetModelsList(srmodel_list_t* models_list) {
    models_list_ = models_list;

#if CONFIG_IDF_TARGET_ESP32S3 || CONFIG_IDF_TARGET_ESP32P4
    if (esp_srmodel_filter(models_list_, ESP_MN_PREFIX, NULL) != nullptr) {
        wake_word_ = std::make_unique<CustomWakeWord>();
    } else if (esp_srmodel_filter(models_list_, ESP_WN_PREFIX, NULL) != nullptr) {
        wake_word_ = std::make_unique<AfeWakeWord>();
    } else {
        wake_word_ = nullptr;
    }
#else
    if (esp_srmodel_filter(models_list_, ESP_WN_PREFIX, NULL) != nullptr) {
        wake_word_ = std::make_unique<EspWakeWord>();
    } else {
        wake_word_ = nullptr;
    }
#endif

    if (wake_word_) {
        wake_word_->OnWakeWordDetected([this](const std::string& wake_word) {
            if (callbacks_.on_wake_word_detected) {
                callbacks_.on_wake_word_detected(wake_word);
            }
        });
    }
}
```

根据芯片型号 + 已加载的模型类型动态选唤醒词实现：

- S3/P4 上若有 multinet（`ESP_MN_PREFIX`）→ `CustomWakeWord`（支持自定义任意词）；
- S3/P4 上若有 wakenet（`ESP_WN_PREFIX`）→ `AfeWakeWord`（标准唤醒）；
- 其它芯片（C3 等）→ `EspWakeWord`（轻量）；
- 都没有 → 不支持唤醒。

这是 `Assets::Apply()` 在第 7 章会调用的——assets 分区里有什么模型就在这里被用上。

```
bool AudioService::IsAfeWakeWord() {
#if CONFIG_IDF_TARGET_ESP32S3 || CONFIG_IDF_TARGET_ESP32P4
    return wake_word_ != nullptr && dynamic_cast<AfeWakeWord*>(wake_word_.get()) != nullptr;
#else
    return false;
#endif
}
```

`dynamic_cast` 检查当前是不是 AFE 唤醒——`HandleStateChangedEvent` 里 speaking 状态判断"能否打断"用到。

4.3 `audio_codec.{h,cc}` —— I²S 抽象基类

4.3.1 接口（`audio_codec.h`）

```
class AudioCodec {
public:
    virtual void SetOutputVolume(int volume);
    virtual void SetInputGain(float gain);
    virtual void EnableInput(bool enable);
    virtual void EnableOutput(bool enable);
    virtual void OutputData(std::vector<int16_t>& data);
    virtual bool InputData(std::vector<int16_t>& data);
    virtual void Start();

    inline bool duplex() const;
    inline bool input_reference() const;
    inline int input_sample_rate() const;   // 重要：常被上层查
    inline int output_sample_rate() const;
    inline int input_channels() const;
    inline int output_channels() const;
    inline int output_volume() const;
    inline float input_gain() const;
    inline bool input_enabled() const;
    inline bool output_enabled() const;

protected:
    i2s_chan_handle_t tx_handle_;
    i2s_chan_handle_t rx_handle_;

    bool duplex_;
    bool input_reference_;
    ...

    virtual int Read(int16_t* dest, int samples) = 0;
    virtual int Write(const int16_t* data, int samples) = 0;
};
```

要点：

- `Read / Write` 纯虚——具体 codec 子类实现 I²S 收发；
- 公开接口 `InputData / OutputData` 包装一层 `std::vector` 友好；
- 几个状态字段（`duplex_` / `input_reference_` / `input_sample_rate_` 等）由子类构造时设置。

4.3.2 基类实现要点（`audio_codec.cc`）

```cpp
void AudioCodec::Start() {
    Settings settings("audio", false);
    output_volume_ = settings.GetInt("output_volume", output_volume_);
    if (output_volume_ <= 0) {
        ESP_LOGW(TAG, "Output volume value (%d) is too small, setting to default (10)",
         output_volume_);
        output_volume_ = 10;
    }
    if (tx_handle_ != nullptr) ESP_ERROR_CHECK(i2s_channel_enable(tx_handle_));
    if (rx_handle_ != nullptr) ESP_ERROR_CHECK(i2s_channel_enable(rx_handle_));
    EnableInput(true);
    EnableOutput(true);
}

void AudioCodec::SetOutputVolume(int volume) {
    output_volume_ = volume;
    Settings settings("audio", true);
    settings.SetInt("output_volume", output_volume_);   // ★ 持久化
}
```

- 启动时从 NVS 读上次保存的音量；
- 改音量自动写回 NVS（断电下次还原）。

子类（如 `Es8311AudioCodec`）会在自己的实现里**覆盖 SetOutputVolume**，先调基类保存到 NVS，再发 I²C 命令改 codec 芯片寄存器实际生效。这种"父类做通用、子类做硬件"是嵌入式常用结构。

4.3.3 具体 codec 子类一览

文件	行数	适用硬件	关键技术
`no_audio_codec.cc`	359	无外置 codec 芯片，用 MCU 直驱 PDM 麦克风 + I²S 喇叭	`i2s_pdm_rx_*` + `i2s_std_tx_*`
`es8311_audio_codec.cc`	196	最常见的 ES8311（单声道），如面包板、bread-compact-wifi 等	I²C 配置寄存器 + I²S
`es8374_audio_codec.cc`	197	ES8374	同上
`es8388_audio_codec.cc`	221	ES8388（带耳机功放）	同上
`es8389_audio_codec.cc`	203	ES8389（双声道）	同上
`box_audio_codec.cc`	244	乐鑫 ESP-BOX 用的组合方案	双芯片协同
`dummy_audio_codec.cc`	20	占位——板子无音频时编译过	全空实现

实现细节几百行重复——基本都是"调 esp-codec-dev 组件配置 I²C 写寄存器初始化 codec，然后 i2s_std_new_channel + i2s_channel_init_std_mode 配置 I²S，再实现 Read/Write 包 `i2s_channel_read/write`"。**学习路径**：读懂 `es8311_audio_codec.cc`（最常见），其它板子的看一眼差异即可。

4.4 `audio_processor.h` + AFE 实现

4.4.1 抽象基类（`audio_processor.h`）

```
class AudioProcessor {
public:
    virtual void Initialize(AudioCodec* codec, int frame_duration_ms, srmodel_list_t*
        models_list) = 0;
    virtual void Feed(std::vector<int16_t>&& data) = 0;
    virtual void Start() = 0;
    virtual void Stop() = 0;
    virtual bool IsRunning() = 0;
    virtual void OnOutput(std::function<void(std::vector<int16_t>&& data)> callback) = 0;
    virtual void OnVadStateChange(std::function<void(bool speaking)> callback) = 0;
    virtual size_t GetFeedSize() = 0;
    virtual void EnableDeviceAec(bool enable) = 0;
};
```

`Feed` 喂原始 PCM 进；处理完后通过 `OnOutput` 吐干净 PCM；VAD 状态变化通过 `OnVadStateChange` 吐。

4.4.2 `afe_audio_processor.cc` —— ESP-SR AFE 适配

`Initialize` 关键段：

```
int ref_num = codec_->input_reference() ? 1 : 0;
std::string input_format;
for (int i = 0; i < codec_->input_channels() - ref_num; i++) input_format.push_back('M');
for (int i = 0; i < ref_num; i++) input_format.push_back('R');

afe_config_t* afe_config = afe_config_init(input_format.c_str(), NULL, AFE_TYPE_VC,
        AFE_MODE_HIGH_PERF);
afe_config->aec_mode = AEC_MODE_VOIP_HIGH_PERF;
afe_config->vad_mode = VAD_MODE_0;
afe_config->vad_min_noise_ms = 100;
if (vad_model_name != nullptr) afe_config->vad_model_name = vad_model_name;
if (ns_model_name != nullptr) {
    afe_config->ns_init = true;
    afe_config->ns_model_name = ns_model_name;
    afe_config->afe_ns_mode = AFE_NS_MODE_NET;
} else {
    afe_config->ns_init = false;
}
afe_config->agc_init = false;
afe_config->memory_alloc_mode = AFE_MEMORY_ALLOC_MORE_PSRAM;

#ifdef CONFIG_USE_DEVICE_AEC
    afe_config->aec_init = true;
    afe_config->vad_init = false;
#else
    afe_config->aec_init = false;
    afe_config->vad_init = true;
#endif

afe_iface_ = esp_afe_handle_from_config(afe_config);
afe_data_ = afe_iface_->create_from_config(afe_config);
```

重要参数解读：

参数	含义
`input_format`	字符串编码通道布局，“M”麦克、“R”参考。单麦无 ref = “M”，双麦+ref = “MMR”
`AFE_TYPE_VC`	语音通讯类型（区别于 `AFE_TYPE_SR` 语音识别——见 4.5）
`AFE_MODE_HIGH_PERF`	高性能模式（更多内存换更低延迟）
`aec_mode = AEC_MODE_VOIP_HIGH_PERF`	VoIP 场景高性能 AEC
`vad_mode = VAD_MODE_0`	VAD 灵敏度（共 0/1/2/3 档）
`vad_min_noise_ms = 100`	100ms 内没声音就判定为静音
`ns_init` + `ns_model_name`	降噪网络（NSNET），从模型目录里读
`agc_init = false`	不开自动增益控制（项目自己做）
`memory_alloc_mode = MORE_PSRAM`	内存优先 PSRAM（SRAM 不够）

AEC 和 VAD 互斥：开了 AEC 就不开 VAD（AEC 会大幅修改信号，VAD 不准）；没开 AEC 才用 VAD 判断说话/沉默。

```
xTaskCreate([](void* arg) {
    auto this_ = (AfeAudioProcessor*)arg;
    this_->AudioProcessorTask();
    vTaskDelete(NULL);
}, "audio_communication", 4096, this, 3, NULL);
```

AFE 自己有一个工作任务，处理”fetch”——AFE 内部用环形缓冲，feed 进去算完后通过 fetch 拿出来。

`AudioProcessorTask`：

```
while (true) {
    xEventGroupWaitBits(event_group_, PROCESSOR_RUNNING, pdFALSE, pdTRUE, portMAX_DELAY);
    auto res = afe_iface_->fetch_with_delay(afe_data_, portMAX_DELAY);
    if ((xEventGroupGetBits(event_group_) & PROCESSOR_RUNNING) == 0) continue;
    if (res == nullptr || res->ret_value == ESP_FAIL) continue;

    // VAD 状态变化通知
    if (vad_state_change_callback_) {
        if (res->vad_state == VAD_SPEECH && !is_speaking_) {
            is_speaking_ = true;
            vad_state_change_callback_(true);
        } else if (res->vad_state == VAD_SILENCE && is_speaking_) {
            is_speaking_ = false;
            vad_state_change_callback_(false);
        }
    }

    if (output_callback_) {
        size_t samples = res->data_size / sizeof(int16_t);
        output_buffer_.insert(output_buffer_.end(), res->data, res->data + samples);
        // 攒够一个 Opus 帧再吐出去
        while (output_buffer_.size() >= frame_samples_) {
            if (output_buffer_.size() == frame_samples_) {
                output_callback_(std::move(output_buffer_));
                output_buffer_.clear();
                output_buffer_.reserve(frame_samples_);
            } else {
                output_callback_(std::vector<int16_t>(output_buffer_.begin(),
        output_buffer_.begin() + frame_samples_));
                output_buffer_.erase(output_buffer_.begin(), output_buffer_.begin() +
        frame_samples_);
            }
        }
    }
}
```

精妙点：

- AFE 内部 chunk 是 16ms（256 样本），但 Opus 一帧是 60ms（960 样本）。在 `output_buffer_` 里**攒满 60ms 才吐**。
- 边界情况优化：刚好等于一帧时直接 `std::move`，零拷贝；超过一帧时拷贝出一段。

4.4.3 `no_audio_processor.cc`（小芯片用）

简化实现：不做 AFE，直接把输入 PCM 透传给 OnOutput，并用简单能量检测做 VAD。代码量 59 行，结构跟 AFE 版相似但内部无 ESP-SR 调用。

4.5 `wake_word.h` + 三种唤醒词实现

4.5.1 抽象基类（`wake_word.h`）

```
class WakeWord {
public:
    virtual bool Initialize(AudioCodec* codec, srmodel_list_t* models_list) = 0;
    virtual void Feed(const std::vector<int16_t>& data) = 0;
    virtual void OnWakeWordDetected(std::function<void(const std::string& wake_word)>
        callback) = 0;
    virtual void Start() = 0;
    virtual void Stop() = 0;
    virtual size_t GetFeedSize() = 0;
    virtual void EncodeWakeWordData() = 0;     // 异步把缓存的 2 秒 PCM 编成 Opus
    virtual bool GetWakeWordOpus(std::vector<uint8_t>& opus) = 0;   // 取一帧 Opus
    virtual const std::string& GetLastDetectedWakeWord() const = 0;
};
```

`EncodeWakeWordData` + `GetWakeWordOpus` 是给"上传唤醒词音频做声纹识别"准备的——AFE 检测到唤醒词后保留了刚才那 2 秒 PCM，上层调 Encode 把它转 Opus 流，再调 GetWakeWordOpus 一帧帧拿去发。

4.5.2 `afe_wake_word.cc`（S3/P4，~208 行）

跟 AFE 处理器**类似，但 AFE 类型选 `AFE_TYPE_SR`**：

```
afe_config_t* afe_config = afe_config_init(input_format.c_str(), models_, AFE_TYPE_SR,
        AFE_MODE_HIGH_PERF);
afe_config->aec_init = codec_->input_reference();
afe_config->aec_mode = AEC_MODE_SR_HIGH_PERF;
afe_config->afe_perferred_core = 1;
afe_config->afe_perferred_priority = 1;
afe_config->memory_alloc_mode = AFE_MEMORY_ALLOC_MORE_PSRAM;

afe_iface_ = esp_afe_handle_from_config(afe_config);
afe_data_ = afe_iface_->create_from_config(afe_config);
```

注意 `afe_perferred_core = 1`——把 AFE 内部任务绑到 core 1。这跟主循环（core 0）错开，**双核并行做语音处理 + UI/网络**。

模型扫描：

```
for (int i = 0; i < models_->num; i++) {
    if (strstr(models_->model_name[i], ESP_WN_PREFIX) != NULL) {
        wakenet_model_ = models_->model_name[i];
        auto words = esp_srmodel_get_wake_words(models_, wakenet_model_);
        std::stringstream ss(words);
        std::string word;
        while (std::getline(ss, word, ';')) {
            wake_words_.push_back(word);
        }
    }
}
```

ESP-SR 模型一个文件里**可以包含多个唤醒词**（"你好小智"; "Hi Lexie"），用分号分隔。这里全部 push 到 `wake_words_` 列表。

检测任务：

```cpp
void AfeWakeWord::AudioDetectionTask() {
    while (true) {
        xEventGroupWaitBits(event_group_, DETECTION_RUNNING_EVENT, pdFALSE, pdTRUE,
        portMAX_DELAY);
        auto res = afe_iface_->fetch_with_delay(afe_data_, portMAX_DELAY);
        if (res == nullptr || res->ret_value == ESP_FAIL) continue;

        StoreWakeWordData(res->data, res->data_size / sizeof(int16_t));

        if (res->wakeup_state == WAKENET_DETECTED) {
            Stop();
            last_detected_wake_word_ = wake_words_[res->wakenet_model_index - 1];
            if (wake_word_detected_callback_) {
                wake_word_detected_callback_(last_detected_wake_word_);
            }
        }
    }
}

void AfeWakeWord::StoreWakeWordData(const int16_t* data, size_t samples) {
    wake_word_pcm_.emplace_back(std::vector<int16_t>(data, data + samples));
    while (wake_word_pcm_.size() > 2000 / 30) {   // 保留 2 秒
        wake_word_pcm_.pop_front();
    }
}
```

滑动窗口缓存 2 秒——AFE 每 30ms 给一个 chunk，最多保留 ~67 个 chunk（约 2 秒）。这是后续要”补发”给服务器的素材。

```
void AfeWakeWord::EncodeWakeWordData() {
    const size_t stack_size = 4096 * 7;
    wake_word_opus_.clear();
    if (wake_word_encode_task_stack_ == nullptr) {
        wake_word_encode_task_stack_ = (StackType_t*)heap_caps_malloc(stack_size,
        MALLOC_CAP_SPIRAM);
    }
    if (wake_word_encode_task_buffer_ == nullptr) {
        wake_word_encode_task_buffer_ = (StaticTask_t*)heap_caps_malloc(sizeof(StaticTask_t),
        MALLOC_CAP_INTERNAL);
    }

    wake_word_encode_task_ = xTaskCreateStatic([](void* arg) {
        auto this_ = (AfeWakeWord*)arg;
        auto encoder = std::make_unique<OpusEncoderWrapper>(16000, 1, OPUS_FRAME_DURATION_MS);
        encoder->SetComplexity(0);
        for (auto& pcm: this_->wake_word_pcm_) {
            encoder->Encode(std::move(pcm), [this_](std::vector<uint8_t>&& opus) {
                std::lock_guard<std::mutex> lock(this_->wake_word_mutex_);
                this_->wake_word_opus_.emplace_back(std::move(opus));
                this_->wake_word_cv_.notify_all();
            });
        }
        this_->wake_word_pcm_.clear();
        // 推一个空包做哨兵，告诉消费端"全部完了"
        std::lock_guard<std::mutex> lock(this_->wake_word_mutex_);
        this_->wake_word_opus_.push_back(std::vector<uint8_t>());
        this_->wake_word_cv_.notify_all();
        vTaskDelete(NULL);
    }, "encode_wake_word", stack_size, this, 2, wake_word_encode_task_stack_,
        wake_word_encode_task_buffer_);
}
```

异步编码任务：

- 把 28 KB 栈通过 `heap_caps_malloc(MALLOC_CAP_SPIRAM)` **分配在 PSRAM** 而不是内部 SRAM（节省内部内存）；
- 用 `xTaskCreateStatic` 而不是 `xTaskCreate`——栈和 TCB 自己管，避免动态分配；
- 编码完所有 PCM 后**推一个空包作为哨兵**，消费端拿到空包就知道"流结束了"。

这是**生产者–消费者模式**的标准做法：用哨兵（sentinel）通知流结束，避免另设标志位。

```
bool AfeWakeWord::GetWakeWordOpus(std::vector<uint8_t>& opus) {
    std::unique_lock<std::mutex> lock(wake_word_mutex_);
    wake_word_cv_.wait(lock, [this]() { return !wake_word_opus_.empty(); });
    opus.swap(wake_word_opus_.front());
    wake_word_opus_.pop_front();
    return !opus.empty();   // 返回 false 表示哨兵到达
}
```

消费端在 `Application::HandleWakeWordDetectedEvent` 里循环调，看到 `false` 就停。

4.5.3 `custom_wake_word.cc`（254 行，自定义任意词）

基于 ESP-SR multinet (MN)。和 AFE wakenet 区别：multinet 支持运行期注册"任意短语"，不局限于预训练词。Initialize 时调 `esp_mn_iface_t` 的 `add_speech_commands` 注册一组词。**适合"你给设备起个名"的场景**。详细 API 跟 wakenet 类似，篇幅原因不展开。

4.5.4 `esp_wake_word.cc`（87 行，C3 等小芯片）

不走 AFE（小芯片内存不够开 AFE），直接喂 ESP-SR 的 wakenet：

```
wakenet_iface_ = esp_wn_iface_init(wakenet_model_);
wakenet_data_ = wakenet_iface_->create(wakenet_model_, DET_MODE_90);

void EspWakeWord::Feed(const std::vector<int16_t>& data) {
    auto state = wakenet_iface_->detect(wakenet_data_, const_cast<int16_t*>(data.data()));
    if (state == WAKENET_DETECTED) {
        if (wake_word_detected_callback_) {
            wake_word_detected_callback_(wake_words_[0]);
        }
    }
}
```

很直接：每喂一块 PCM 就 detect 一次，返回检测到就回调。没有 2 秒缓存、不支持上传唤醒词音频。

4.6 `audio_debugger.cc` —— 远程听音质

```
void AudioDebugger::Feed(std::vector<int16_t>& data) {
    if (sock_ < 0) {
        // 第一次：建 UDP socket，从 NVS 读目标地址
    }
    sendto(sock_, data.data(), data.size() * sizeof(int16_t), 0, (struct sockaddr*)&addr_,
        sizeof(addr_));
}
```

把 16 kHz mono PCM 直接 UDP 喷到指定地址。电脑跑 `scripts/audio_debug_server.py`：

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(("0.0.0.0", port))
while True:
    data, _ = sock.recvfrom(65535)
    pcm = np.frombuffer(data, dtype=np.int16)
    stream.write(pcm.tobytes())
```

调试录音质量、AEC 残留时非常顺手。

4.7 第三方库依赖（ESP-SR & Opus）

4.7.1 ESP-SR（`espressif/esp-sr`）

由 `idf_component.yml` 声明依赖，编译期下载。提供：

- `wakenet`：低算力唤醒词检测；
- `multinet`（MN）：命令词识别（运行期可注册短语）；
- `AFE`：声学前端处理框架（VAD、AEC、降噪、波束形成）。

模型文件在 ESP-SR 组件的 `model/` 目录或我们项目 `assets` 分区里。`esp_srmodel_init("model")` 扫描模型目录加载所有 `.bin`。

4.7.2 Opus

通过 `78__esp-opus` 组件（修改过的 Opus 端口）提供：

- `OpusEncoderWrapper(sr, channels, frame_ms)` / `Encode(pcm, payload)`；
- `OpusDecoderWrapper(...)` / `Decode(payload, pcm)`；
- `OpusResampler` / `Configure(in, out)` / `Process(in, in_size, out)` / `GetOutputSamples(in_size)`。

Opus 的窄带语音模式 16 kHz mono 60ms/帧，码率默认约 24–32 kbps——非常省带宽。

4.8 本章用到的核心技术汇总

技术	应用
多 RTOS 任务 + 条件变量 + 事件组	input/output/codec 三任务协作
绑核（`xTaskCreatePinnedToCore`）	input 跑 core 0，AFE 跑 core 1
生产者–消费者 + 背压	5 个有上限队列，满时 wait
哨兵元素通知流结束	wake_word_opus_ 空包
mutex 解锁后做耗时操作	取出 task 后放锁再编解码
抽象基类 + 子类工厂选择	AudioCodec / AudioProcessor / WakeWord 三套
`std::chrono::steady_clock`	功耗 timer 计时差
`std::move` 转移所有权	`unique_ptr<AudioStreamPacket>` 全流程零拷贝
PSRAM 分配栈 `heap_caps_malloc(MALLOC_CAP_SPIRAM)`	唤醒词编码任务的 28 KB 栈
`xTaskCreateStatic`	静态分配 TCB 和栈避免动态分配
手写 OGG 容器解析	PlaySound 实时拆 OGG 取 Opus
采样率重采样	input/reference/output 三套 OpusResampler
ESP-SR AFE 框架	VAD + AEC + 降噪
`dynamic_cast` 运行期类型识别	IsAfeWakeWord
预处理器条件编译	区分 S3/P4 vs C3、是否启用 SERVER_AEC / DEVICE_AEC / USE_AUDIO_PROCESSOR
`std::string_view`	PlaySound 接受嵌入式资源不拷贝
C++17 init capture in lambda	wake_word 编码任务

4.9 看完本章你应该掌握的

- 5 个队列 + 3 个任务 + 4 个事件位的数据流模型
- 上行（MIC → Opus → 网络）和下行（网络 → Opus → 喇叭）两条路径每一步在做什么
- AFE 处理器为什么要等 60ms 才输出（攒一个 Opus 帧）
- Server AEC 的时间戳如何在 input/output 队列间配对
- 唤醒词 2 秒滑动缓存 + 异步编码 + 哨兵流结束的设计

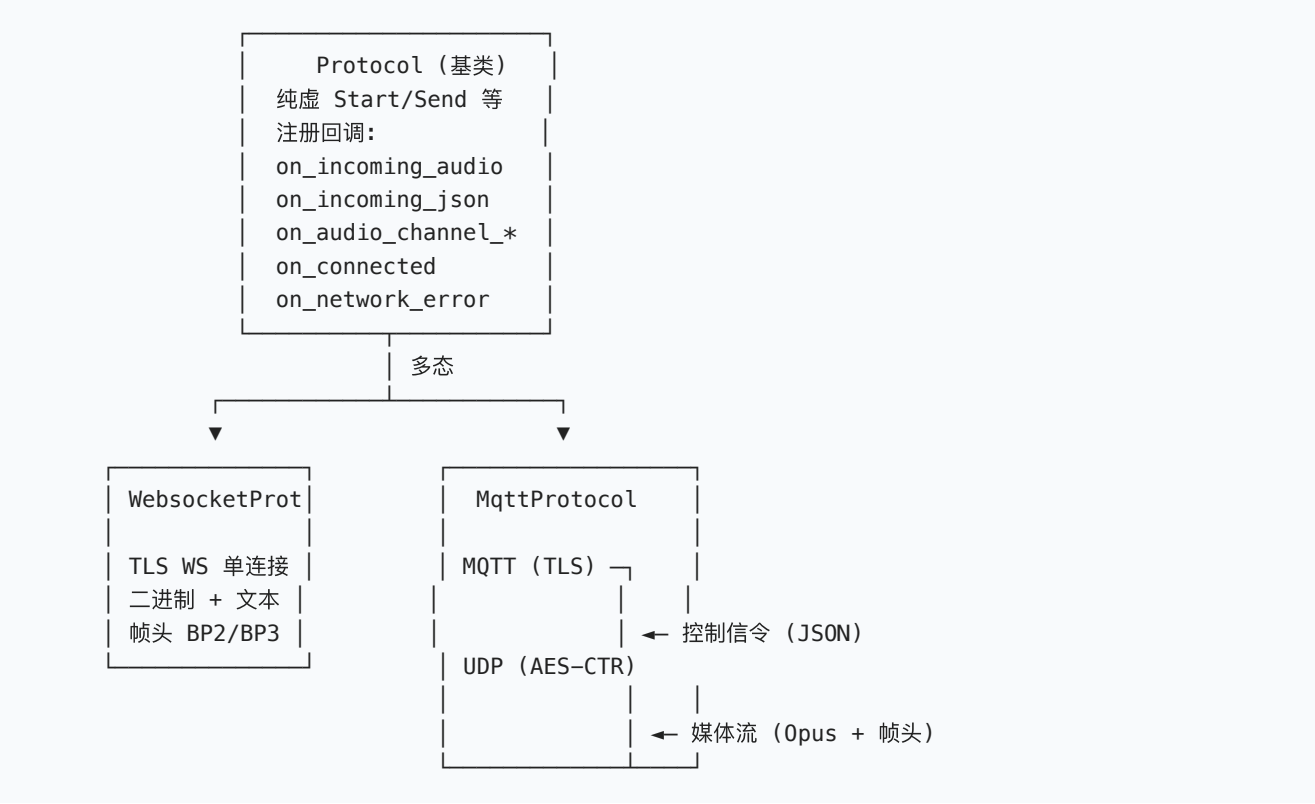
- codec / processor / wake_word 三套抽象基类的责任划分
- I²S 双通道（MIC + reference）和单通道的区别
- 自动功耗管理（15 秒无活动关 I²S）
- PlaySound 是如何在线解析 OGG 容器抽 Opus 帧的

下一章进入 `protocols/` ——WebSocket 和 MQTT+UDP 两套通信协议。

第 5 章 通信协议：`main/protocols/`

> 922 行代码、3 个 .cc 文件、2 套传输方式（WebSocket 单连接 / MQTT 控制信令 + UDP 媒体流），都实现同一个 `Protocol` 抽象基类。本章把基类、两种实现、帧格式、加密细节、握手流程全部拆开。

5.1 通信协议鸟瞰



两种协议对比表：

维度	WebSocket	MQTT + UDP
连接数	1 条 WSS（TLS）	1 条 MQTTS（TLS）+ 1 条 UDP（应用层加密）
控制信令	WSS 文本帧	MQTT publish/subscribe
音频数据	WSS 二进制帧 + 帧头	UDP + AES-128-CTR + 自定义帧头
加密	TLS（端到端）	MQTTS（控制信令）+ AES-CTR（媒体流）
延迟	较高（TCP/TLS 重传 + 头部）	UDP 媒体流低延迟
抗丢包	TCP 重传，可能阻塞	UDP 容忍丢包（每帧带序号）
双向能力	全双工，自然	控制走 MQTT，媒体走 UDP，能并发
服务器实现复杂度	简单（一个 ws server）	较复杂（mqtt broker + udp gateway）
适用场景	普通互联网，对延迟要求一般	移动网络/有 NAT，对延迟敏感的实时语音

服务器端在握手时下发 `mqtt` 配置或 `websocket` 配置，设备就走对应路径。

5.2 `protocol.h` —— 抽象基类与帧头

5.2.1 `AudioStreamPacket` 结构

```
struct AudioStreamPacket {
    int sample_rate = 0;
    int frame_duration = 0;
    uint32_t timestamp = 0;
    std::vector<uint8_t> payload;
};
```

贯穿全项目的”一帧音频”基本单元：

- `sample_rate`：当前帧的 Opus 采样率（决定解码器配置）；
- `frame_duration`：帧长（毫秒，常用 60）；
- `timestamp`：用于服务器 AEC 和 UDP 顺序追踪；
- `payload`：原始 Opus bytes（不含帧头）。

5.2.2 二进制帧头两版

```
struct BinaryProtocol2 {
    uint16_t version;       // = 2，网络字节序
    uint16_t type;          // 0=OPUS, 1=JSON
    uint32_t reserved;
    uint32_t timestamp;     // 毫秒级时间戳，用于服务端 AEC
    uint32_t payload_size;
    uint8_t payload[];
} __attribute__((packed));

struct BinaryProtocol3 {
    uint8_t type;
    uint8_t reserved;
    uint16_t payload_size;
    uint8_t payload[];
} __attribute__((packed));
```

两版帧头：

- **v2**：16 字节头，含版本号、消息类型、时间戳、长度。适合 WebSocket 复用单连接（同一条 WS 流上跑多种消息）；
- **v3**：4 字节头，只有类型 + 长度。MQTT+UDP 模式用，时间戳和顺序号放在 UDP 的 nonce 里更紧凑。

`__attribute__((packed))`：**禁止编译器按对齐补齐字段**——结构体直接覆盖网络上的字节流。再加上 `uint8_t payload[]` 这种"灵活数组成员"（C99 引入），实现"零拷贝指向连续 payload"。

5.2.3 `AbortReason` 和 `ListeningMode`

```
enum AbortReason {
    kAbortReasonNone,
    kAbortReasonWakeWordDetected
};

enum ListeningMode {
    kListeningModeAutoStop,    // VAD 静音超时自动结束
    kListeningModeManualStop,  // 显式 stop 才结束（按住说话）
    kListeningModeRealtime     // 全双工，需要 AEC
};
```

由协议在 SendStartListening / SendAbortSpeaking 时携带。

5.2.4 基类公开接口

```
class Protocol {
public:
    void OnIncomingAudio(std::function<void(std::unique_ptr<AudioStreamPacket>)>);
    void OnIncomingJson(std::function<void(const cJSON* root)>);
    void OnAudioChannelOpened(std::function<void()>);
    void OnAudioChannelClosed(std::function<void()>);
    void OnNetworkError(std::function<void(const std::string&)>);
    void OnConnected(std::function<void()>);
    void OnDisconnected(std::function<void()>);

    virtual bool Start() = 0;
    virtual bool OpenAudioChannel() = 0;
    virtual void CloseAudioChannel() = 0;
    virtual bool IsAudioChannelOpened() const = 0;
    virtual bool SendAudio(std::unique_ptr<AudioStreamPacket>) = 0;
    virtual void SendWakeWordDetected(const std::string& wake_word);
    virtual void SendStartListening(ListeningMode mode);
    virtual void SendStopListening();
    virtual void SendAbortSpeaking(AbortReason reason);
    virtual void SendMcpMessage(const std::string& message);
    ...

protected:
    std::function<...> on_incoming_audio_;     // 等等 7 个回调
    int server_sample_rate_ = 24000;
    int server_frame_duration_ = 60;
    bool error_occurred_ = false;
    std::string session_id_;
    std::chrono::time_point<std::chrono::steady_clock> last_incoming_time_;

    virtual bool SendText(const std::string& text) = 0;
    virtual void SetError(const std::string& message);
    virtual bool IsTimeout() const;
};
```

设计要点：

- **6 个发送类方法是虚函数有默认实现**（`SendWakeWordDetected/SendStartListening/SendStopListening/SendAbortSpeaking/SendMcpMessage` + 一个 `Send*`），共用底层 `SendText(json_str)`；
- `SendText` 才是纯虚——具体走 WS 文本还是 MQTT publish 由子类决定；
- 7 个回调全部 `std::function`，子类只负责"在合适的时机调用"，业务逻辑由 `Application` 通过 `OnXxx(...)` 注入；
- `server_sample_rate_ / server_frame_duration_` 由握手时服务器 hello 消息确定，下行音频解码用；
- `session_id_` 也是握手时服务器分配，所有消息都带；
- `last_incoming_time_` 给 `IsTimeout()` 判断 120 秒无任何数据则视为掉线。

5.2.5 `protocol.cc` 基类实现 —— 通用控制消息生成

```
void Protocol::SendAbortSpeaking(AbortReason reason) {
    std::string message = "{\"session_id\":\"" + session_id_ + "\",\"type\":\"abort\"";
    if (reason == kAbortReasonWakeWordDetected) {
        message += ",\"reason\":\"wake_word_detected\"";
    }
    message += "}";
    SendText(message);
}

void Protocol::SendWakeWordDetected(const std::string& wake_word) {
    std::string json = "{\"session_id\":\"" + session_id_ +
                       "\",\"type\":\"listen\",\"state\":\"detect\",\"text\":\"" + wake_word +
        "\"}";
    SendText(json);
}

void Protocol::SendStartListening(ListeningMode mode) {
    std::string message = "{\"session_id\":\"" + session_id_ + "\"";
    message += ",\"type\":\"listen\",\"state\":\"start\"";
    if (mode == kListeningModeRealtime) {
        message += ",\"mode\":\"realtime\"";
    } else if (mode == kListeningModeAutoStop) {
        message += ",\"mode\":\"auto\"";
    } else {
        message += ",\"mode\":\"manual\"";
    }
    message += "}";
    SendText(message);
}

void Protocol::SendStopListening() {
    std::string message = "{\"session_id\":\"" + session_id_ +
        "\",\"type\":\"listen\",\"state\":\"stop\"}";
    SendText(message);
}

void Protocol::SendMcpMessage(const std::string& payload) {
    std::string message = "{\"session_id\":\"" + session_id_ +
        "\",\"type\":\"mcp\",\"payload\":" + payload + "}";
    SendText(message);
}

bool Protocol::IsTimeout() const {
    const int kTimeoutSeconds = 120;
    auto now = std::chrono::steady_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::seconds>(now -
        last_incoming_time_);
    bool timeout = duration.count() > kTimeoutSeconds;
    if (timeout) {
        ESP_LOGE(TAG, "Channel timeout %ld seconds", (long)duration.count());
    }
    return timeout;
}
```

字符串拼接 JSON 而不是用 cJSON：

- 这些消息字段固定，手写更快；
- 避免 cJSON 频繁 alloc/free 内存碎片；

- 对应文档：`docs/websocket.md` 和 `docs/mqtt-udp.md` 里有完整字段说明。

要点：

消息	type	state	附加字段
Hello（子类各自写）	hello	—	version, transport, features, audio_params
Goodbye（mqtt 关闭通道时）	goodbye	—	—
Abort	abort	—	reason: wake_word_detected（可选）
WakeWordDetected	listen	detect	text: 唤醒词
StartListening	listen	start	mode: realtime/auto/manual
StopListening	listen	stop	—
McpMessage	mcp	—	payload: 完整 MCP 报文（已 JSON 字符串）

服务器下行 JSON 见第 2 章 2.6.7（tts/stt/llm/mcp/system/alert/custom）。

5.3 `WebsocketProtocol` 实现详解

5.3.1 构造与生命期

```
WebsocketProtocol::WebsocketProtocol() {
    event_group_handle_ = xEventGroupCreate();
}

WebsocketProtocol::~WebsocketProtocol() {
    vEventGroupDelete(event_group_handle_);
}
```

事件组里只用了一位 `WEBSOCKET_PROTOCOL_SERVER_HELLO_EVENT`——等服务器 hello 回包的同步。

5.3.2 `Start()` —— 懒连接

```
bool WebsocketProtocol::Start() {
    // Only connect to server when audio channel is needed
    return true;
}
```

WebSocket 不在 `Start()` 里建连——**懒连接**：等到 `OpenAudioChannel()` 才真正握手，避免空 idle 也持有 TCP 连接耗资源。

5.3.3 `OpenAudioChannel()` —— 完整握手流程

```cpp
bool WebsocketProtocol::OpenAudioChannel() {
    Settings settings("websocket", false);
    std::string url = settings.GetString("url");
    std::string token = settings.GetString("token");
    int version = settings.GetInt("version");
    if (version != 0) version_ = version;

    error_occurred_ = false;

    auto network = Board::GetInstance().GetNetwork();
    websocket_ = network->CreateWebSocket(1);
    if (websocket_ == nullptr) return false;

    if (!token.empty()) {
        if (token.find(" ") == std::string::npos) {
            token = "Bearer " + token;
        }
        websocket_->SetHeader("Authorization", token.c_str());
    }
    websocket_->SetHeader("Protocol-Version", std::to_string(version_).c_str());
    websocket_->SetHeader("Device-Id", SystemInfo::GetMacAddress().c_str());
    websocket_->SetHeader("Client-Id", Board::GetInstance().GetUuid().c_str());
```

第一步：

- 从 NVS 命名空间 `websocket` 取 url / token / version（OTA 阶段写进去的）；
- 通过板子的 `network->CreateWebSocket()` 拿一个 WebSocket 句柄（`network` 抽象了 WiFi 和 4G 两种栈）；
- 设三个标准头：Authorization（带 Bearer 前缀）、Protocol-Version、Device-Id（MAC）、Client-Id（UUID，软件生成）。

```
    websocket_->OnData([this](const char* data, size_t len, bool binary) {
        if (binary) {
            // ... 见下 5.3.4 二进制收包
        } else {
            auto root = cJSON_Parse(data);
            auto type = cJSON_GetObjectItem(root, "type");
            if (cJSON_IsString(type)) {
                if (strcmp(type->valuestring, "hello") == 0) {
                    ParseServerHello(root);
                } else {
                    if (on_incoming_json_ != nullptr) {
                        on_incoming_json_(root);
                    }
                }
            } else {
                ESP_LOGE(TAG, "Missing message type, data: %s", data);
            }
            cJSON_Delete(root);
        }
        last_incoming_time_ = std::chrono::steady_clock::now();
    });

    websocket_->OnDisconnected([this]() {
        if (on_audio_channel_closed_ != nullptr) {
            on_audio_channel_closed_();
        }
    });
```

挂回调：

- 文本帧 → 解 JSON，hello 内部处理，其它交给 `on_incoming_json_`（业务层装的大分发器）；
- 断开 → 通知业务层。

```
    if (!websocket_->Connect(url.c_str())) {
        SetError(Lang::Strings::SERVER_NOT_CONNECTED);
        return false;
    }

    auto message = GetHelloMessage();
    if (!SendText(message)) return false;

    EventBits_t bits = xEventGroupWaitBits(event_group_handle_,
        WEBSOCKET_PROTOCOL_SERVER_HELLO_EVENT,
                                           pdTRUE, pdFALSE, pdMS_TO_TICKS(10000));
    if (!(bits & WEBSOCKET_PROTOCOL_SERVER_HELLO_EVENT)) {
        SetError(Lang::Strings::SERVER_TIMEOUT);
        return false;
    }

    if (on_audio_channel_opened_ != nullptr) on_audio_channel_opened_();
    return true;
}
```

握手三步：

1. TCP+TLS 连接（`websocket_->Connect`）；
2. 发 client hello；

3. **同步等 10 秒收 server hello**——超时报错。

为什么用 `xEventGroupWaitBits` 等而不是直接在 OnData 回调里走？因为 `OpenAudioChannel` 是被业务层在主循环上调用的，需要返回 true/false。把"server hello 收到"翻译成事件位，让调用者同步阻塞等。

5.3.4 `OnData` 二进制帧解析

```
if (version_ == 2) {
    BinaryProtocol2* bp2 = (BinaryProtocol2*)data;
    bp2->version = ntohs(bp2->version);
    bp2->type = ntohs(bp2->type);
    bp2->timestamp = ntohl(bp2->timestamp);
    bp2->payload_size = ntohl(bp2->payload_size);
    auto payload = (uint8_t*)bp2->payload;
    on_incoming_audio_(std::make_unique<AudioStreamPacket>(AudioStreamPacket{
        .sample_rate = server_sample_rate_,
        .frame_duration = server_frame_duration_,
        .timestamp = bp2->timestamp,
        .payload = std::vector<uint8_t>(payload, payload + bp2->payload_size)
    }));
} else if (version_ == 3) {
    BinaryProtocol3* bp3 = (BinaryProtocol3*)data;
    bp3->payload_size = ntohs(bp3->payload_size);
    auto payload = (uint8_t*)bp3->payload;
    on_incoming_audio_(std::make_unique<AudioStreamPacket>(AudioStreamPacket{
        .sample_rate = server_sample_rate_,
        .frame_duration = server_frame_duration_,
        .timestamp = 0,
        .payload = std::vector<uint8_t>(payload, payload + bp3->payload_size)
    }));
} else {
    // version 1：无帧头，整个 binary 就是 Opus payload
    on_incoming_audio_(std::make_unique<AudioStreamPacket>(AudioStreamPacket{
        .sample_rate = server_sample_rate_,
        .frame_duration = server_frame_duration_,
        .timestamp = 0,
        .payload = std::vector<uint8_t>((uint8_t*)data, (uint8_t*)data + len)
    }));
}
```

要点：

- **`ntohs / ntohl`** 把网络字节序（大端）转主机字节序（小端 = ESP32 默认）；
- 用 C 风格 cast `(BinaryProtocol2*)data` 直接覆盖结构体到字节流——配合 `packed` 属性，零拷贝读字段；
- **payload 还是要拷贝一份到 `std::vector`**——原 `data` 缓冲属于 websocket 客户端栈，回调返回后失效；
- v1 没有帧头，整段就是 Opus。

5.3.5 `SendAudio()` 发送（与 OnData 镜像）

```cpp
bool WebsocketProtocol::SendAudio(std::unique_ptr<AudioStreamPacket> packet) {
    if (websocket_ == nullptr || !websocket_->IsConnected()) return false;

    if (version_ == 2) {
        std::string serialized;
        serialized.resize(sizeof(BinaryProtocol2) + packet->payload.size());
        auto bp2 = (BinaryProtocol2*)serialized.data();
        bp2->version = htons(version_);
        bp2->type = 0;
        bp2->reserved = 0;
        bp2->timestamp = htonl(packet->timestamp);
        bp2->payload_size = htonl(packet->payload.size());
        memcpy(bp2->payload, packet->payload.data(), packet->payload.size());
        return websocket_->Send(serialized.data(), serialized.size(), true);
    } else if (version_ == 3) {
        // ... 类似但用 BP3
    } else {
        return websocket_->Send(packet->payload.data(), packet->payload.size(), true);
    }
}
```

`Send(data, size, true)` 第三参 `true` = binary 帧。

5.3.6 `GetHelloMessage()` —— 客户端 hello

```cpp
std::string WebsocketProtocol::GetHelloMessage() {
    cJSON* root = cJSON_CreateObject();
    cJSON_AddStringToObject(root, "type", "hello");
    cJSON_AddNumberToObject(root, "version", version_);
    cJSON* features = cJSON_CreateObject();
#if CONFIG_USE_SERVER_AEC
    cJSON_AddBoolToObject(features, "aec", true);
#endif
    cJSON_AddBoolToObject(features, "mcp", true);
    cJSON_AddItemToObject(root, "features", features);
    cJSON_AddStringToObject(root, "transport", "websocket");
    cJSON* audio_params = cJSON_CreateObject();
    cJSON_AddStringToObject(audio_params, "format", "opus");
    cJSON_AddNumberToObject(audio_params, "sample_rate", 16000);
    cJSON_AddNumberToObject(audio_params, "channels", 1);
    cJSON_AddNumberToObject(audio_params, "frame_duration", OPUS_FRAME_DURATION_MS);
    cJSON_AddItemToObject(root, "audio_params", audio_params);
    auto json_str = cJSON_PrintUnformatted(root);
    std::string message(json_str);
    cJSON_free(json_str);
    cJSON_Delete(root);
    return message;
}
```

发出去类似：

```
{
  "type": "hello",
  "version": 3,
  "features": {"aec": false, "mcp": true},
  "transport": "websocket",
  "audio_params": {
    "format": "opus",
    "sample_rate": 16000,
    "channels": 1,
    "frame_duration": 60
  }
}
```

服务器看了知道客户端能力 → 决定下行 Opus 帧用什么采样率，回 hello。

5.3.7 `ParseServerHello()` —— 协议参数同步

```
void WebsocketProtocol::ParseServerHello(const cJSON* root) {
    auto transport = cJSON_GetObjectItem(root, "transport");
    if (transport == nullptr || strcmp(transport->valuestring, "websocket") != 0) return;

    auto session_id = cJSON_GetObjectItem(root, "session_id");
    if (cJSON_IsString(session_id)) {
        session_id_ = session_id->valuestring;
    }

    auto audio_params = cJSON_GetObjectItem(root, "audio_params");
    if (cJSON_IsObject(audio_params)) {
        auto sample_rate = cJSON_GetObjectItem(audio_params, "sample_rate");
        if (cJSON_IsNumber(sample_rate)) server_sample_rate_ = sample_rate->valueint;
        auto frame_duration = cJSON_GetObjectItem(audio_params, "frame_duration");
        if (cJSON_IsNumber(frame_duration)) server_frame_duration_ = frame_duration->valueint;
    }

    xEventGroupSetBits(event_group_handle_, WEBSOCKET_PROTOCOL_SERVER_HELLO_EVENT);
}
```

记下 session_id 和服务器音频参数，触发事件位让 `OpenAudioChannel` 解除阻塞。

5.3.8 `CloseAudioChannel()` 和 `IsAudioChannelOpened()`

```
void WebsocketProtocol::CloseAudioChannel() {
    websocket_.reset();
}

bool WebsocketProtocol::IsAudioChannelOpened() const {
    return websocket_ != nullptr && websocket_->IsConnected() && !error_occurred_ &&
        !IsTimeout();
}
```

WebSocket 直接 reset 句柄触发析构，连接随之关闭。Open 判断要四条都满足：句柄存在 + TCP 连着 + 没出错 + 没超时。

5.4 `MqttProtocol` 实现详解

5.4.1 构造 —— 设置重连 timer

```
MqttProtocol::MqttProtocol() {
    event_group_handle_ = xEventGroupCreate();

    esp_timer_create_args_t reconnect_timer_args = {
        .callback = [](void* arg) {
            MqttProtocol* protocol = (MqttProtocol*)arg;
            auto& app = Application::GetInstance();
            if (app.GetDeviceState() == kDeviceStateIdle) {
                auto alive = protocol->alive_;
                app.Schedule([protocol, alive]() {
                    if (*alive) {
                        protocol->StartMqttClient(false);
                    }
                });
            }
        },
        .arg = this,
    };
    esp_timer_create(&reconnect_timer_args, &reconnect_timer_);
}
```

MQTT 经常掉线（移动网络弱信号），自带重连机制：

- 一个一次性 timer；
- 触发时**先检查是不是 idle**（其它状态比如正在通话不重连，避免打断）；
- **`alive_` 是 `shared_ptr<atomic<bool>>` 哨兵**——析构时 set false，避免 timer 调度的 lambda 在对象已销毁后还跑（dangling pointer 的经典防御）。
- 真正的重连推到主线程（`Schedule`）跑——单线程化协议状态。

5.4.2 析构 —— 优雅停机

```
MqttProtocol::~MqttProtocol() {
    *alive_ = false;   // ★ 先标记自己死了
    if (reconnect_timer_ != nullptr) {
        esp_timer_stop(reconnect_timer_);
        esp_timer_delete(reconnect_timer_);
    }
    udp_.reset();
    mqtt_.reset();
    if (event_group_handle_ != nullptr) vEventGroupDelete(event_group_handle_);
}
```

`*alive_ = false` 是关键——后续任何已经在 Schedule 队列里 pending 的 lambda 检查 `if (*alive)` 都会跳过执行。

5.4.3 `Start()` + `StartMqttClient()`

```cpp
bool MqttProtocol::Start() {
    return StartMqttClient(false);
}

bool MqttProtocol::StartMqttClient(bool report_error) {
    if (mqtt_ != nullptr) mqtt_.reset();

    Settings settings("mqtt", false);
    auto endpoint = settings.GetString("endpoint");
    auto client_id = settings.GetString("client_id");
    auto username = settings.GetString("username");
    auto password = settings.GetString("password");
    int keepalive_interval = settings.GetInt("keepalive", 240);
    publish_topic_ = settings.GetString("publish_topic");

    if (endpoint.empty()) {
        if (report_error) SetError(Lang::Strings::SERVER_NOT_FOUND);
        return false;
    }

    auto network = Board::GetInstance().GetNetwork();
    mqtt_ = network->CreateMqtt(0);
    mqtt_->SetKeepAlive(keepalive_interval);
```

跟 WS 不一样的是——**MQTT 在 `Start()` 就建连**，因为 MQTT 需要持续在线接受 publish。控制信令随时可能从服务器发来（如 OTA 强制重启）。

```cpp
    mqtt_->OnDisconnected([this]() {
        if (on_disconnected_ != nullptr) on_disconnected_();
        esp_timer_start_once(reconnect_timer_, MQTT_RECONNECT_INTERVAL_MS * 1000);
    });

    mqtt_->OnConnected([this]() {
        if (on_connected_ != nullptr) on_connected_();
        esp_timer_stop(reconnect_timer_);
    });
```

- 断开 → 启动 60 秒倒计时重连 timer；
- 连上 → 取消重连 timer（如果有的话）。

```
    mqtt_->OnMessage([this](const std::string& topic, const std::string& payload) {
        cJSON* root = cJSON_Parse(payload.c_str());
        if (root == nullptr) return;
        cJSON* type = cJSON_GetObjectItem(root, "type");
        if (!cJSON_IsString(type)) { cJSON_Delete(root); return; }

        if (strcmp(type->valuestring, "hello") == 0) {
            ParseServerHello(root);
        } else if (strcmp(type->valuestring, "goodbye") == 0) {
            auto session_id = cJSON_GetObjectItem(root, "session_id");
            if (session_id == nullptr || session_id_ == session_id->valuestring) {
                auto alive = alive_;
                Application::GetInstance().Schedule([this, alive]() {
                    if (*alive) CloseAudioChannel();
                });
            }
        } else if (on_incoming_json_ != nullptr) {
            on_incoming_json_(root);
        }
        cJSON_Delete(root);
        last_incoming_time_ = std::chrono::steady_clock::now();
    });
```

MQTT 消息处理：

- `hello` → 本地处理（拿 UDP 端点）；
- `goodbye` → 服务端主动断开（比如会话结束或被踢），关闭 UDP；
- 其它 → 透传给业务层。

注意 `goodbye` 用 `Schedule()` 延后到主线程，**MQTT 回调线程不直接做关闭操作**。

```
    std::string broker_address;
    int broker_port = 8883;
    size_t pos = endpoint.find(':');
    if (pos != std::string::npos) {
        broker_address = endpoint.substr(0, pos);
        broker_port = std::stoi(endpoint.substr(pos + 1));
    } else {
        broker_address = endpoint;
    }
    if (!mqtt_->Connect(broker_address, broker_port, client_id, username, password)) {
        SetError(Lang::Strings::SERVER_NOT_CONNECTED);
        return false;
    }
    return true;
}
```

解析 `host:port`，默认 8883（MQTTS），连接 broker。

5.4.4 `OpenAudioChannel()` —— 申请 UDP 通道

```cpp
bool MqttProtocol::OpenAudioChannel() {
    if (mqtt_ == nullptr || !mqtt_->IsConnected()) {
        if (!StartMqttClient(true)) return false;
    }

    error_occurred_ = false;
    session_id_ = "";
    xEventGroupClearBits(event_group_handle_, MQTT_PROTOCOL_SERVER_HELLO_EVENT);

    auto message = GetHelloMessage();
    if (!SendText(message)) return false;

    EventBits_t bits = xEventGroupWaitBits(event_group_handle_,
        MQTT_PROTOCOL_SERVER_HELLO_EVENT, pdTRUE, pdFALSE, pdMS_TO_TICKS(10000));
    if (!(bits & MQTT_PROTOCOL_SERVER_HELLO_EVENT)) {
        SetError(Lang::Strings::SERVER_TIMEOUT);
        return false;
    }

    std::lock_guard<std::mutex> lock(channel_mutex_);
    auto network = Board::GetInstance().GetNetwork();
    udp_ = network->CreateUdp(2);
    udp_->OnMessage([this](const std::string& data) { /* 见 5.4.6 */ });
    udp_->Connect(udp_server_, udp_port_);

    if (on_audio_channel_opened_ != nullptr) on_audio_channel_opened_();
    return true;
}
```

握手流程：

1. MQTT 控制信令通道已连（或重连）；
2. 通过 MQTT 发 client hello（type=hello, transport=udp）；
3. 等服务端 hello 回 UDP 服务器地址 + 加密 key + nonce；
4. 用拿到的信息开 UDP 客户端、绑回调。

`channel_mutex_` 保护 udp_ 指针——SendAudio 在另一个 task（主循环）调用。

5.4.5 `ParseServerHello()` —— 拿 UDP 凭证 + 初始化 AES

```
void MqttProtocol::ParseServerHello(const cJSON* root) {
    auto transport = cJSON_GetObjectItem(root, "transport");
    if (transport == nullptr || strcmp(transport->valuestring, "udp") != 0) return;

    auto session_id = cJSON_GetObjectItem(root, "session_id");
    if (cJSON_IsString(session_id)) session_id_ = session_id->valuestring;

    auto audio_params = cJSON_GetObjectItem(root, "audio_params");
    if (cJSON_IsObject(audio_params)) {
        auto sample_rate = cJSON_GetObjectItem(audio_params, "sample_rate");
        if (cJSON_IsNumber(sample_rate)) server_sample_rate_ = sample_rate->valueint;
        auto frame_duration = cJSON_GetObjectItem(audio_params, "frame_duration");
        if (cJSON_IsNumber(frame_duration)) server_frame_duration_ = frame_duration->valueint;
    }

    auto udp = cJSON_GetObjectItem(root, "udp");
    if (!cJSON_IsObject(udp)) return;
    udp_server_ = cJSON_GetObjectItem(udp, "server")->valuestring;
    udp_port_ = cJSON_GetObjectItem(udp, "port")->valueint;
    auto key = cJSON_GetObjectItem(udp, "key")->valuestring;
    auto nonce = cJSON_GetObjectItem(udp, "nonce")->valuestring;

    aes_nonce_ = DecodeHexString(nonce);
    mbedtls_aes_init(&aes_ctx_);
    mbedtls_aes_setkey_enc(&aes_ctx_, (const unsigned char*)DecodeHexString(key).c_str(),
        128);
    local_sequence_ = 0;
    remote_sequence_ = 0;
    xEventGroupSetBits(event_group_handle_, MQTT_PROTOCOL_SERVER_HELLO_EVENT);
}
```

服务器返回的 hello 类似：

```
{
  "type": "hello",
  "transport": "udp",
  "session_id": "abc123",
  "audio_params": { "sample_rate": 24000, "frame_duration": 60 },
  "udp": {
    "server": "1.2.3.4",
    "port": 9000,
    "key": "0011...AABB",      // 32 hex chars = 16 bytes = AES-128 key
    "nonce": "01...0000"       // 32 hex chars = 16 bytes = AES nonce 初始模板
  }
}
```

设备处理：

1. 16 字节 AES-128 key 解 hex 后初始化 `mbedtls_aes_setkey_enc`（只 setkey_enc，因为 CTR 模式加密解密同一函数）；
2. nonce 解 hex 后保存进 `aes_nonce_`；
3. 序号清零；
4. 触发事件位让 `OpenAudioChannel` 解除阻塞。

5.4.6 `SendAudio()` —— AES-128-CTR 加密

```cpp
bool MqttProtocol::SendAudio(std::unique_ptr<AudioStreamPacket> packet) {
    std::lock_guard<std::mutex> lock(channel_mutex_);
    if (udp_ == nullptr) return false;

    std::string nonce(aes_nonce_);
    *(uint16_t*)&nonce[2] = htons(packet->payload.size());
    *(uint32_t*)&nonce[8] = htonl(packet->timestamp);
    *(uint32_t*)&nonce[12] = htonl(++local_sequence_);

    std::string encrypted;
    encrypted.resize(aes_nonce_.size() + packet->payload.size());
    memcpy(encrypted.data(), nonce.data(), nonce.size());

    size_t nc_off = 0;
    uint8_t stream_block[16] = {0};
    if (mbedtls_aes_crypt_ctr(&aes_ctx_, packet->payload.size(), &nc_off,
        (uint8_t*)nonce.c_str(), stream_block,
        (uint8_t*)packet->payload.data(), (uint8_t*)&encrypted[nonce.size()]) != 0) {
        return false;
    }

    return udp_->Send(encrypted) > 0;
}
```

UDP 加密音频帧结构（16 字节 nonce + payload）：

```
nonce (16 字节):
┌──────┬──────┬──────────┬──────────┬─────────────┬─────────────┐
│type  │flag  │ size     │  ssrc    │timestamp    │ sequence    │
│ 1u   │ 1u   │   2u     │   4u     │    4u       │    4u       │
└──────┴──────┴──────────┴──────────┴─────────────┴─────────────┘
encrypted payload (变长):
  AES-128-CTR(key, nonce, opus_bytes)
```

关键点：

- nonce 的高 8 字节由服务器固定（type/flag/ssrc 等），后 8 字节客户端每帧改（payload size、timestamp、sequence）；
- 这样每帧 nonce 唯一，**CTR 模式安全前提是 nonce 不重用**；
- AES-CTR 是流密码，输出长度等于输入，不需要对齐 padding；
- nonce 也明文发出去——接收方需要它来解密；
- `local_sequence_` 严格递增——服务器可以检测乱序/重放。

`mbedtls_aes_crypt_ctr` 参数：

- ctx：AES 上下文（设了 128 位 key）；
- length：加密多少字节；
- nc_off：内部偏移，连续多次调用时用；
- nonce_counter：CTR 计数器（也是 IV）；
- stream_block：内部流块；
- input/output：源/目的。

5.4.7 `udp_->OnMessage` —— UDP 收包解密

```
udp_->OnMessage([this](const std::string& data) {
    if (data.size() < sizeof(aes_nonce_)) return;
    if (data[0] != 0x01) return;   // 类型校验

    uint32_t timestamp = ntohl(*(uint32_t*)&data[8]);
    uint32_t sequence = ntohl(*(uint32_t*)&data[12]);
    if (sequence < remote_sequence_) {
        ESP_LOGW(TAG, "Received audio packet with old sequence: %lu, expected: %lu", sequence,
         remote_sequence_);
        return;
    }
    if (sequence != remote_sequence_ + 1) {
        ESP_LOGW(TAG, "Received audio packet with wrong sequence: %lu, expected: %lu",
         sequence, remote_sequence_ + 1);
    }

    size_t decrypted_size = data.size() - aes_nonce_.size();
    size_t nc_off = 0;
    uint8_t stream_block[16] = {0};
    auto nonce = (uint8_t*)data.data();
    auto encrypted = (uint8_t*)data.data() + aes_nonce_.size();
    auto packet = std::make_unique<AudioStreamPacket>();
    packet->sample_rate = server_sample_rate_;
    packet->frame_duration = server_frame_duration_;
    packet->timestamp = timestamp;
    packet->payload.resize(decrypted_size);
    if (mbedtls_aes_crypt_ctr(&aes_ctx_, decrypted_size, &nc_off, nonce, stream_block,
                              encrypted, (uint8_t*)packet->payload.data()) != 0) return;

    if (on_incoming_audio_ != nullptr) on_incoming_audio_(std::move(packet));
    remote_sequence_ = sequence;
    last_incoming_time_ = std::chrono::steady_clock::now();
});
```

要点：

- **乱序丢弃**：sequence 小于已收的最大值直接丢；
- **乱序但更大**：打 warn 但仍然接收（可能丢了几帧但要继续，UDP 容忍丢包）；
- 解密用同一个 AES 上下文（CTR 模式加解密同函数）；
- 解密后 payload 包成 `AudioStreamPacket` 推到上层 audio_service 的 decode 队列。

5.4.8 `CloseAudioChannel()` —— 发 goodbye + 关 UDP

```
void MqttProtocol::CloseAudioChannel() {
    {
        std::lock_guard<std::mutex> lock(channel_mutex_);
        udp_.reset();
    }
    std::string message = "{";
    message += "\"session_id\":\"" + session_id_ + "\",";
    message += "\"type\":\"goodbye\"";
    message += "}";
    SendText(message);

    if (on_audio_channel_closed_ != nullptr) on_audio_channel_closed_();
}
```

先关 UDP（释放本地资源），再通过 MQTT 发 goodbye 通知服务器（不强制等待 ack）。

5.4.9 `IsAudioChannelOpened()`

```
bool MqttProtocol::IsAudioChannelOpened() const {
    return udp_ != nullptr && !error_occurred_ && !IsTimeout();
}
```

注意只看 udp_——MQTT 即使断了但 UDP 还在跑也可以发音频（控制信令暂时收不到罢了）。

5.4.10 `DecodeHexString()` —— hex 字符串转字节

```
static inline uint8_t CharToHex(char c) {
    if (c >= '0' && c <= '9') return c - '0';
    if (c >= 'A' && c <= 'F') return c - 'A' + 10;
    if (c >= 'a' && c <= 'f') return c - 'a' + 10;
    return 0;
}

std::string MqttProtocol::DecodeHexString(const std::string& hex_string) {
    std::string decoded;
    decoded.reserve(hex_string.size() / 2);
    for (size_t i = 0; i < hex_string.size(); i += 2) {
        char byte = (CharToHex(hex_string[i]) << 4) | CharToHex(hex_string[i + 1]);
        decoded.push_back(byte);
    }
    return decoded;
}
```

把 `"AB12CD"` → `\xAB\x12\xCD`。简单的两位组合。

5.5 协议层 vs 业务层的”接线总表”

回看第 2 章 2.6.7 `InitializeProtocol()`，把它的所有挂钩列在这里方便整体回顾：

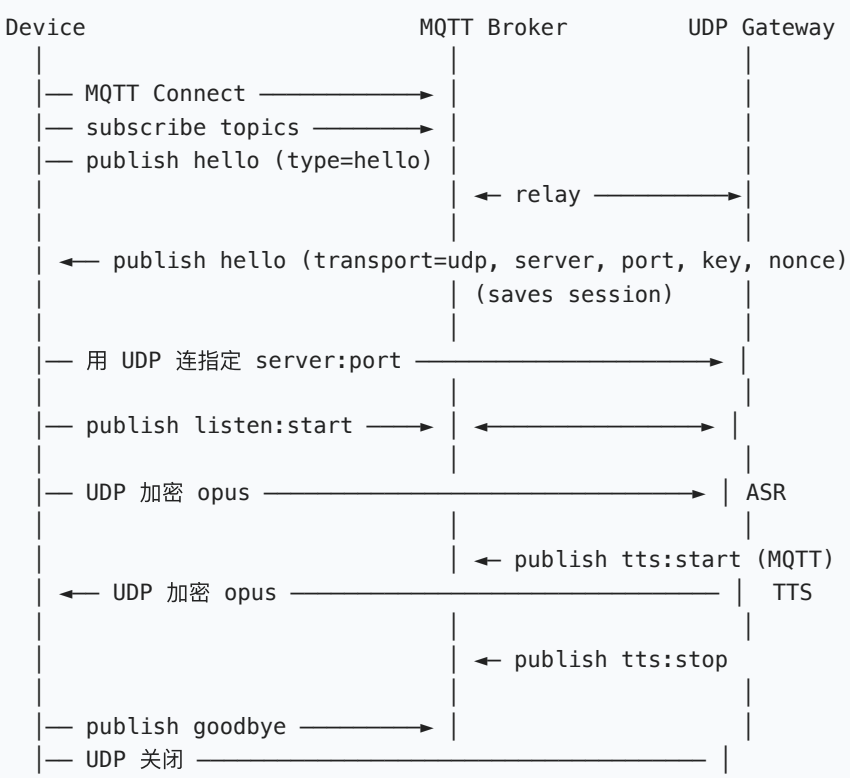
```
Application 注册的回调                    协议层会在什么时候调
──────────────────────────────────  ──────────────────────────────────────────
OnConnected (DismissAlert)           MQTT 重连成功 / 任意协议连上
OnNetworkError (set ERROR 位)        连接失败 / Send 失败 / Timeout
OnIncomingAudio (push decode 队列) WS binary 帧 / UDP 解密后
OnAudioChannelOpened (升性能档) OpenAudioChannel 握手成功
OnAudioChannelClosed (降功耗+清 UI) WS 断开 / udp_ 重置后
OnIncomingJson (大分发器)            WS 文本帧 / MQTT publish (除 hello/goodbye)

Application 主动调协议                   做什么
──────────────────────────────────  ──────────────────────────────────────────
protocol_->Start()                   MQTT: 立即连; WS: 不做事
protocol_->OpenAudioChannel()        建立音频通道 (同步等 hello)
protocol_->CloseAudioChannel()       关音频通道
protocol_->IsAudioChannelOpened() 状态查询
protocol_->SendAudio(packet)         发 Opus 包
protocol_->SendStartListening(mode)  控制信令
protocol_->SendStopListening()
protocol_->SendAbortSpeaking(reason)
protocol_->SendWakeWordDetected(word)
protocol_->SendMcpMessage(payload)
```

5.6 一次完整对话的协议时序图

WebSocket 模式：

```
Device                          Server
  |                               |
  |── TCP+TLS 握手 ──────────────► |
  |                               |
  |── WS Upgrade ────────────────► | (HTTP 101)
  |                               |
  |── {type:hello, ...} ─────────► |
  |                               |
  | ◄── {type:hello, session_id} |  保存 session
  |                               |
  |── {type:listen,state:start} ► |
  |                               |
  | ──BP3 + opus ────────────────► |
  | ──BP3 + opus ────────────────► |
  | ──BP3 + opus ────────────────► |
  |                               | (服务器做 ASR)
  | ◄── {type:stt, text:"...."} |
  | ◄── {type:llm, emotion:".."} |
  | ◄── {type:tts, state:start}  |
  | ◄── {type:tts, sentence_start, text:"..."}
  | ◄──BP3 + opus ──────────────  |
  | ◄──BP3 + opus ──────────────  |
  | ◄── {type:tts, state:stop}   |
  |                               |
  |── {type:listen,state:start} ► | (auto 模式回 listening)
  |   ...                         |
```

MQTT+UDP 模式：



5.7 本章用到的核心技术汇总

技术	应用
C++ 抽象基类 + 多态	`Protocol` 基类 + WS/MQTT 两实现
`std::function` + lambda 回调	7 个回调挂钩业务层
`__attribute__((packed))` + 灵活数组	帧头结构直接覆盖字节流
`ntohs / htons / ntohl / htonl`	网络字节序转换
C 风格 cast 直接读结构体	零拷贝读字段
FreeRTOS EventGroup	同步等服务器 hello（最多 10 秒）
`std::shared_ptr<std::atomic<bool>>` 哨兵	防止延迟 timer 回调访问已销毁对象
`esp_timer_start_once`	60 秒一次性重连
`std::mutex` 保护 udp_	SendAudio 和 OnMessage 并发访问
`mbedtls_aes_crypt_ctr`	AES-128-CTR 加密音频
Hex 字符串转字节	服务器下发 key/nonce 编码方式
UDP 自定义序号	检测重放/乱序
cJSON 解析/生成	控制信令
手写 JSON 字符串拼接	性能优化（高频小消息）
`std::chrono::steady_clock`	120 秒无活动超时
`auto alive = alive_` 在 lambda 里捕获 shared_ptr	Schedule 队列里也安全
NVS 配置存储	url/token/version/endpoint/keepalive 等
板级 NetworkInterface 抽象	WS/MQTT/UDP 各自由板子提供具体实现，WiFi 板和 4G 板共用上层代码

5.8 看完本章你应该掌握的

- `Protocol` 基类的 7 个回调和 5 个 send 方法
- 两版二进制帧头 BP2/BP3 字段含义
- WebSocket 模式：单连接、懒连接、HTTP header 鉴权、Hello 协商音频参数
- MQTT+UDP 模式：MQTT 长在线 + UDP 一次性加密媒体流；hello 时拿 UDP key/nonce
- AES-128-CTR 在 nonce 里嵌入 size/timestamp/sequence 的设计
- UDP 序号检测乱序 + 重放
- Application 注册的 7 个回调 vs 主动调的 9 个方法 —— 完整的”协议接线表”
- MqttProtocol 用 shared_ptr<atomic> 防止延迟回调悬空指针的技巧
- 一次对话从 hello 到 goodbye 的完整时序

下一章进入 `mcp_server.cc/h` ——设备端 MCP 协议实现（让大模型直接调用设备功能的桥梁）。

第 6 章 设备端 MCP 协议：`mcp_server.h / .cc`

> 909 行（345 头 + 564 实现），这是本项目”让大模型能直接控制硬件”的桥梁。所有 LLM 通过云端协议下发的工具调用都从这里进入。

6.1 MCP 是什么、为什么要 MCP

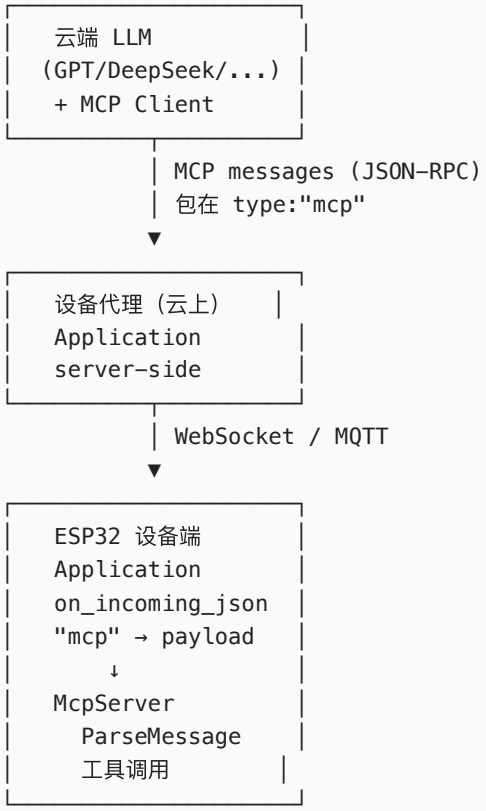
MCP（Model Context Protocol） 是 2024 年底由社区推出的开放协议，规范了”工具调用”在 LLM 客户端和服务器之间的统一接口。设备端实现 MCP server 后，云端的 LLM（甚至来自不同厂商）只要支持 MCP 就能调用本设备的功能，比如：

- 调节扬声器音量
- 拍照并解释（Vision）
- 重启设备
- 切换屏幕主题
- 设备厂商扩展的”打开灯”“开空调”等私有工具

不用 MCP 之前，云端要单独对接每个厂商私有的 JSON RPC；用了 MCP 之后，工具列表由设备端动态注册并通过 `tools/list` 返回，LLM 自己决定调用哪个 + 怎么传参。

为什么放在设备端而不是云端？因为有些动作（控制本地硬件、读取实时传感器）必须在设备端执行，绕一圈云就太慢。

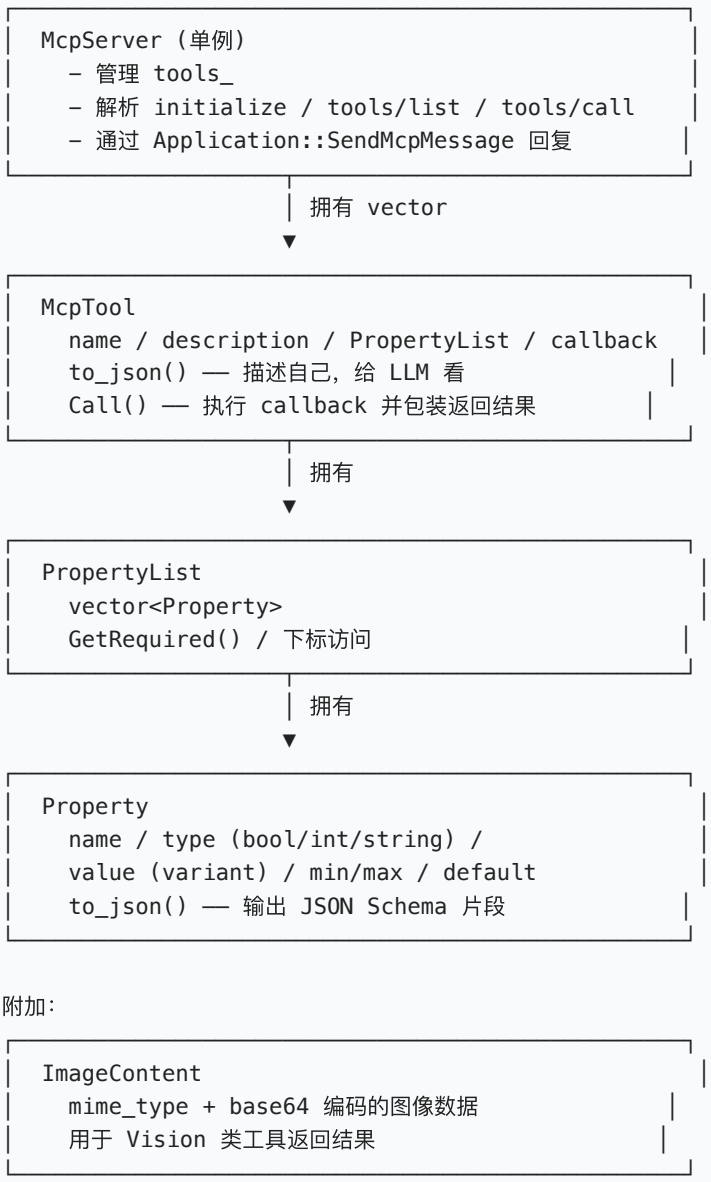
底层传输怎么走？MCP 是个 JSON-RPC 2.0 协议，本项目把它**封进 WebSocket / MQTT 现有通道**——所有 MCP 消息都包在 `{type: "mcp", payload: {...}}` 的外层信封里。



设备端 `McpServer` 的工作 = **接收 JSON-RPC 报文 + 路由到工具 + 返回结果**。

6.2 文件分层结构

`mcp_server.h` 把 4 个层级的类全堆在一个头文件里（350 行）：



附加：

6.3 `Property` 类逐行讲解

6.3.1 字段

```
class Property {
private:
    std::string name_;
    PropertyType type_;
    std::variant<bool, int, std::string> value_;
    bool has_default_value_;
    std::optional<int> min_value_;
    std::optional<int> max_value_;
};
```

`std::variant<bool, int, std::string>` —— C++17 引入，是类型安全的 union。能容纳 3 种类型之一，编译器追踪当前是哪种。配合 `std::get<T>(value_)` 和 `std::holds_alternative<T>(value_)` 做类型分发。

`std::optional<int>` —— C++17 引入，表达"可能没有值"的整型（用 `has_value()` 检查）。比 sentinel value（-1 表示无）干净得多。

`PropertyType` 是枚举三选一：boolean / integer / string。**故意不支持 float**——LLM 给浮点参数太容易出 corner case，整型 + 缩放因子更稳。

6.3.2 4 个构造函数

```
// 1) 必填字段（无默认值，无范围）
Property(const std::string& name, PropertyType type)
    : name_(name), type_(type), has_default_value_(false) {}

// 2) 模板化的"有默认值"构造（不带范围限制）
template<typename T>
Property(const std::string& name, PropertyType type, const T& default_value)
    : name_(name), type_(type), has_default_value_(true) {
    value_ = default_value;
}

// 3) 整数必填 + 范围限制
Property(const std::string& name, PropertyType type, int min_value, int max_value)
    : name_(name), type_(type), has_default_value_(false),
      min_value_(min_value), max_value_(max_value) {
    if (type != kPropertyTypeInteger) {
        throw std::invalid_argument("Range limits only apply to integer properties");
    }
}

// 4) 整数 + 默认值 + 范围
Property(const std::string& name, PropertyType type, int default_value, int min_value, int
         max_value)
    : name_(name), type_(type), has_default_value_(true),
      min_value_(min_value), max_value_(max_value) {
    if (type != kPropertyTypeInteger) {
        throw std::invalid_argument("Range limits only apply to integer properties");
    }
    if (default_value < min_value || default_value > max_value) {
        throw std::invalid_argument("Default value must be within the specified range");
    }
    value_ = default_value;
}
```

四种语义：

#	用途	例子
1	必填、无范围	`Property("question", kPropertyTypeString)`
2	选填（带默认）、无范围	`Property("url", kPropertyTypeString, "http://default")`
3	必填、有范围	`Property("volume", kPropertyTypeInteger, 0, 100)`
4	选填、有默认 + 范围	`Property("quality", kPropertyTypeInteger, 80, 1, 100)`

构造函数里抛 `std::invalid_argument` 是编译期 + 运行时双重保护。

6.3.3 模板化 set_value + 范围检查

```
template<typename T>
inline void set_value(const T& value) {
    if constexpr (std::is_same_v<T, int>) {
        if (min_value_.has_value() && value < min_value_.value()) {
            throw std::invalid_argument("Value is below minimum allowed: " +
        std::to_string(min_value_.value()));
        }
        if (max_value_.has_value() && value > max_value_.value()) {
            throw std::invalid_argument("Value exceeds maximum allowed: " +
        std::to_string(max_value_.value()));
        }
    }
    value_ = value;
}
```

`if constexpr (std::is_same_v<T, int>)` 是 C++17 的编译期 if——T 不是 int 时整个分支被编译器丢掉，零运行时开销。这种"模板里只对部分类型做事"的写法干净（旧 C++ 要写特化 / SFINAE）。

`std::invalid_argument` 让 `tools/call` 解析阶段就能捕获 + 上报错误。

6.3.4 `to_json()` —— 生成 JSON Schema 片段

```
std::string to_json() const {
    cJSON *json = cJSON_CreateObject();
    if (type_ == kPropertyTypeBoolean) {
        cJSON_AddStringToObject(json, "type", "boolean");
        if (has_default_value_) cJSON_AddBoolToObject(json, "default", value<bool>());
    } else if (type_ == kPropertyTypeInteger) {
        cJSON_AddStringToObject(json, "type", "integer");
        if (has_default_value_) cJSON_AddNumberToObject(json, "default", value<int>());
        if (min_value_.has_value()) cJSON_AddNumberToObject(json, "minimum",
        min_value_.value());
        if (max_value_.has_value()) cJSON_AddNumberToObject(json, "maximum",
        max_value_.value());
    } else if (type_ == kPropertyTypeString) {
        cJSON_AddStringToObject(json, "type", "string");
        if (has_default_value_) cJSON_AddStringToObject(json, "default", value<std::string>
        ().c_str());
    }
    // ...
}
```

输出形如：

```
{ "type": "integer", "default": 80, "minimum": 1, "maximum": 100 }
```

这是 JSON Schema 的最小子集——LLM 拿到后就知道该参数的取值约束，能自动校验和生成合法的调用。

6.4 `PropertyList` 容器

```cpp
class PropertyList {
private:
    std::vector<Property> properties_;
public:
    PropertyList() = default;
    PropertyList(const std::vector<Property>& properties) : properties_(properties) {}
    void AddProperty(const Property& property) { properties_.push_back(property); }

    const Property& operator[](const std::string& name) const {
        for (const auto& property : properties_) {
            if (property.name() == name) return property;
        }
        throw std::runtime_error("Property not found: " + name);
    }

    auto begin() { return properties_.begin(); }
    auto end() { return properties_.end(); }

    std::vector<std::string> GetRequired() const {
        std::vector<std::string> required;
        for (auto& property : properties_) {
            if (!property.has_default_value()) required.push_back(property.name());
        }
        return required;
    }
    std::string to_json() const { /* 调用每个 property.to_json() */ }
};
```

关键设计：

- **下标按 name 而不是 index 访问**——可读性 > 性能（属性数通常 1–3 个，线性查找够用）；
- `begin/end` 让 `PropertyList` 可以 `for (auto& argument : arguments)`（C++ range-for 仅需要 begin/end）；
- `GetRequired()` 把没有默认值的属性名挑出来，放进 `inputSchema.required` 字段（这是 MCP 强制要求的格式）；
- 构造支持初始化列表风格 `PropertyList({ Property(...), Property(...) })`。

6.5 `McpTool` —— 工具元数据 + 执行器

6.5.1 字段与构造

```
class McpTool {
private:
    std::string name_;
    std::string description_;
    PropertyList properties_;
    std::function<ReturnValue(const PropertyList&)> callback_;
    bool user_only_ = false;
public:
    McpTool(const std::string& name,
            const std::string& description,
            const PropertyList& properties,
            std::function<ReturnValue(const PropertyList&)> callback)
        : name_(name), description_(description), properties_(properties), callback_(callback)
        {}
};
```

注意 `ReturnValue` 这个别名：

```
using ReturnValue = std::variant<bool, int, std::string, cJSON*, ImageContent*>;
```

5 种可能的返回类型：

- **`bool`**：成功 / 失败的工具（set_volume 之类）；
- **`int`**：返回数值（如电量百分比）；
- **`std::string`**：返回文本（如 OCR 结果、状态描述）；
- **`cJSON*`**：返回结构化数据（如 GetDeviceStatusJson）；
- **`ImageContent*`**：返回图片（如拍照）。

注意是裸指针 `cJSON*` 和 `ImageContent*`——所有权转给 McpTool::Call，由 Call 内部释放（这是有点”反 RAII”的设计，但因为返回类型不固定+性能考虑，用 `std::unique_ptr` 跟 variant 混合会有些麻烦）。

6.5.2 `user_only_` 含义

```
void set_user_only(bool user_only) { user_only_ = user_only; }
inline bool user_only() const { return user_only_; }
```

`user_only_ = true` 的工具**对 LLM 不可见**——只有客户端的”用户控制台”能调用（reboot / upgrade_firmware / screen.snapshot 等）。安全考虑：不能让 LLM 自己决定重启 / 升级固件。在 `to_json()` 里通过 `annotations.audience = ["user"]` 标注：

```
if (user_only_) {
    cJSON *annotations = cJSON_CreateObject();
    cJSON *audience = cJSON_CreateArray();
    cJSON_AddItemToArray(audience, cJSON_CreateString("user"));
    cJSON_AddItemToObject(annotations, "audience", audience);
    cJSON_AddItemToObject(json, "annotations", annotations);
}
```

而且在 `GetToolsList` 里默认不返回这些工具（除非显式带 `withUserTools: true`）。

6.5.3 `to_json()` —— 完整工具描述

```
std::string to_json() const {
    std::vector<std::string> required = properties_.GetRequired();
    cJSON *json = cJSON_CreateObject();
    cJSON_AddStringToObject(json, "name", name_.c_str());
    cJSON_AddStringToObject(json, "description", description_.c_str());

    cJSON *input_schema = cJSON_CreateObject();
    cJSON_AddStringToObject(input_schema, "type", "object");
    cJSON *properties = cJSON_Parse(properties_.to_json().c_str());
    cJSON_AddItemToObject(input_schema, "properties", properties);
    if (!required.empty()) {
        cJSON *required_array = cJSON_CreateArray();
        for (const auto& property : required) {
            cJSON_AddItemToArray(required_array, cJSON_CreateString(property.c_str()));
        }
        cJSON_AddItemToObject(input_schema, "required", required_array);
    }
    cJSON_AddItemToObject(json, "inputSchema", input_schema);
    // ... (user_only annotations)
}
```

输出形如：

```
{
  "name": "self.audio_speaker.set_volume",
  "description": "Set the volume of the audio speaker. ...",
  "inputSchema": {
    "type": "object",
    "properties": {
      "volume": { "type": "integer", "minimum": 0, "maximum": 100 }
    },
    "required": ["volume"]
  }
}
```

LLM 读到这段就知道：要传 volume 整数、范围 0–100、必填。

6.5.4 `Call()` —— 执行工具并包装结果

```
std::string Call(const PropertyList& properties) {
    ReturnValue return_value = callback_(properties);
    cJSON* result = cJSON_CreateObject();
    cJSON* content = cJSON_CreateArray();

    if (std::holds_alternative<ImageContent*>(return_value)) {
        auto image_content = std::get<ImageContent*>(return_value);
        cJSON* image = cJSON_CreateObject();
        cJSON_AddStringToObject(image, "type", "image");
        cJSON_AddStringToObject(image, "image", image_content->to_json().c_str());
        cJSON_AddItemToArray(content, image);
        delete image_content;
    } else {
        cJSON* text = cJSON_CreateObject();
        cJSON_AddStringToObject(text, "type", "text");
        if (std::holds_alternative<std::string>(return_value)) {
            cJSON_AddStringToObject(text, "text", std::get<std::string>
        (return_value).c_str());
        } else if (std::holds_alternative<bool>(return_value)) {
            cJSON_AddStringToObject(text, "text", std::get<bool>(return_value) ? "true" :
        "false");
        } else if (std::holds_alternative<int>(return_value)) {
            cJSON_AddStringToObject(text, "text", std::to_string(std::get<int>
        (return_value)).c_str());
        } else if (std::holds_alternative<cJSON*>(return_value)) {
            cJSON* json = std::get<cJSON*>(return_value);
            char* json_str = cJSON_PrintUnformatted(json);
            cJSON_AddStringToObject(text, "text", json_str);
            cJSON_free(json_str);
            cJSON_Delete(json);
        }
        cJSON_AddItemToArray(content, text);
    }
    cJSON_AddItemToObject(result, "content", content);
    cJSON_AddBoolToObject(result, "isError", false);
    // ...
}
```

要点：

1. **调用 callback** 拿到 ReturnValue；
2. 根据 variant 当前持有的类型分发包装：
   - ImageContent → `{type:"image", image:"..."}` 并 `delete image_content`（**消费所有权**）；
   - 其它 → `{type:"text", text:"..."}`；
3. 包装成 MCP 标准的 `content` 数组（注意是数组，可以多个内容混排）；
4. 加 `isError: false` 标记成功；
5. 返回完整的 JSON 字符串。

异常路径在 `DoToolCall` 里 catch（见 6.7.6）。

6.6 `ImageContent` —— 图像返回结果封装

```
class ImageContent {
private:
    std::string encoded_data_;
    std::string mime_type_;

    static std::string Base64Encode(const std::string& data) {
        size_t dlen = 0, olen = 0;
        // 第一次调用：传 nullptr 让 mbedtls 计算需要多大 buffer，写到 dlen
        mbedtls_base64_encode((unsigned char*)nullptr, 0, &dlen,
                              (const unsigned char*)data.data(), data.size());
        std::string result(dlen, 0);
        // 第二次调用：真正写入 buffer
        mbedtls_base64_encode((unsigned char*)result.data(), result.size(), &olen,
                              (const unsigned char*)data.data(), data.size());
        return result;
    }

public:
    ImageContent(const std::string& mime_type, const std::string& data) {
        mime_type_ = mime_type;
        encoded_data_ = Base64Encode(data);
    }

    std::string to_json() const { /* {"type":"image","mimeType":...,"data":<base64>} */ }
};
```

两次调用 `mbedtls_base64_encode` 的惯用法：

- 第一次：传空 buffer → mbedtls 把所需字节数写进 `dlen`，但不写数据（这是 mbedtls 标准模式：alloc-size-first）；
- 第二次：分配好 buffer 再调用真正写入。

避免预估错 buffer 大小（base64 输出长度大致是输入的 4/3，再加换行符）。

6.7 `McpServer` —— 核心调度器

6.7.1 单例 + 构造析构

```cpp
class McpServer {
public:
    static McpServer& GetInstance() {
        static McpServer instance;
        return instance;
    }
private:
    McpServer();
    ~McpServer();
    std::vector<McpTool*> tools_;
};

McpServer::McpServer() {}
McpServer::~McpServer() {
    for (auto tool : tools_) delete tool;
    tools_.clear();
}
```

Meyers 单例（line 316–319）：`static` 局部变量在 C++11 起线程安全。

工具用裸指针 + 手动 `delete`——是历史代码 / 性能权衡的产物。如果重写，会用 `std::vector<std::unique_ptr<McpTool>>`。

6.7.2 `AddCommonTools()` —— 注册"对 LLM 可见"的工具

被 `Application::Initialize()` 在启动末尾调用：

```cpp
void McpServer::AddCommonTools() {
    auto original_tools = std::move(tools_);   // ★ 先把已注册的板级私有工具备份
    auto& board = Board::GetInstance();

    // 工具 1: 设备状态查询
    AddTool("self.get_device_status",
        "Provides the real-time information of the device, ...",
        PropertyList(),
        [&board](const PropertyList& properties) -> ReturnValue {
            return board.GetDeviceStatusJson();   // 返回 cJSON*
        });

    // 工具 2: 音量控制
    AddTool("self.audio_speaker.set_volume",
        "Set the volume of the audio speaker. ...",
        PropertyList({ Property("volume", kPropertyTypeInteger, 0, 100) }),
        [&board](const PropertyList& properties) -> ReturnValue {
            auto codec = board.GetAudioCodec();
            codec->SetOutputVolume(properties["volume"].value<int>());
            return true;
        });

    auto backlight = board.GetBacklight();
    if (backlight) {
        AddTool("self.screen.set_brightness", ...);
    }

#ifdef HAVE_LVGL
    auto display = board.GetDisplay();
    if (display && display->GetTheme() != nullptr) {
        AddTool("self.screen.set_theme", ...);
    }
    auto camera = board.GetCamera();
    if (camera) {
        AddTool("self.camera.take_photo", ...);
    }
#endif

    // ★ 把板级私有工具 append 到末尾
    tools_.insert(tools_.end(), original_tools.begin(), original_tools.end());
}
```

两段式注册的精妙（line 39 / line 125）：

1. 板级 `InitializeTools()` 已经把板子私有工具加进 `tools_`（在 board.cc 里）；
2. `AddCommonTools` 先用 `std::move(tools_)` 把它们暂存到 `original_tools`；
3. 注册通用工具（按顺序 push 到现在空的 `tools_`）；
4. 最后把暂存的 append 回去。

为什么把通用工具放前面？注释明确说了——**利用 prompt cache**。大模型给同样的 system prompt 时会缓存计算，工具列表越前面越固定，缓存命中率越高，响应越快。私有工具因板而异，放后面就只破坏后面的缓存。

`take_photo` 工具特别有意思：

```
[camera](const PropertyList& properties) -> ReturnValue {
    TaskPriorityReset priority_reset(1);  // ★ 临时降低优先级
    if (!camera->Capture()) {
        throw std::runtime_error("Failed to capture photo");
    }
    auto question = properties["question"].value<std::string>();
    return camera->Explain(question);   // 返回 string（云端的视觉结果）
}
```

`TaskPriorityReset` 是 2 章 2.5.3 见过的 RAII helper——拍照耗时的 JPEG 编码不阻塞主任务和音频任务。`camera->Explain` 内部走 HTTP 上传图到 vision URL（`ParseCapabilities` 见 6.7.4 设置的），返回 LLM 看图描述。

6.7.3 `AddUserOnlyTools()` —— 注册”对用户客户端可见”的工具

被设备厂商的”用户控制台 APP”调用（不是 LLM）：

```
void McpServer::AddUserOnlyTools() {
    // 系统信息
    AddUserOnlyTool("self.get_system_info", "Get the system information",
        PropertyList(),
        [this](const PropertyList& properties) -> ReturnValue {
            return Board::GetInstance().GetSystemInfoJson();
        });

    // 重启
    AddUserOnlyTool("self.reboot", "Reboot the system",
        PropertyList(),
        [this](const PropertyList& properties) -> ReturnValue {
            auto& app = Application::GetInstance();
            app.Schedule([&app]() {
                vTaskDelay(pdMS_TO_TICKS(1000));   // 给响应回去的时间
                app.Reboot();
            });
            return true;
        });

    // 固件升级
    AddUserOnlyTool("self.upgrade_firmware", "Upgrade firmware from a specific URL. ...",
        PropertyList({ Property("url", kPropertyTypeString, "...") }),
        [this](const PropertyList& properties) -> ReturnValue {
            auto url = properties["url"].value<std::string>();
            auto& app = Application::GetInstance();
            app.Schedule([url, &app]() {
                bool success = app.UpgradeFirmware(url);
                if (!success) ESP_LOGE(TAG, "Firmware upgrade failed");
            });
            return true;
        });

    // 屏幕信息 / 截图 / 预览（仅 LVGL）
#ifdef HAVE_LVGL
    auto display = dynamic_cast<LvglDisplay*>(Board::GetInstance().GetDisplay());
    if (display) {
        AddUserOnlyTool("self.screen.get_info", ...);
#if CONFIG_LV_USE_SNAPSHOT
        AddUserOnlyTool("self.screen.snapshot", ...);
        AddUserOnlyTool("self.screen.preview_image", ...);
#endif
    }
#endif

    // 资源下载 URL
    auto& assets = Assets::GetInstance();
    if (assets.partition_valid()) {
        AddUserOnlyTool("self.assets.set_download_url", "Set the download url for the assets",
            PropertyList({ Property("url", kPropertyTypeString) }),
            [](const PropertyList& properties) -> ReturnValue {
                auto url = properties["url"].value<std::string>();
                Settings settings("assets", true);
                settings.SetString("download_url", url);
                return true;
            });
    }
}
```

要点：

- **重启 / 升级用 `app.Schedule`** 推到主循环——MCP 调用线程不能直接重启自己；
- `dynamic_cast<LvglDisplay*>` 检测是不是 LVGL 显示（不是的话 SnapshotToJpeg 不存在）；
- `snapshot` 工具实现了**完整的 `multipart/form-data`** HTTP POST：手写 boundary、文件字段头、JPEG 数据、尾部分隔符；
- `preview_image` 先 HTTP GET 拿 JPEG/PNG 数据到 PSRAM（`heap_caps_malloc(content_length, MALLOC_CAP_8BIT)`），再交给 LVGL 显示。

6.7.4 `ParseCapabilities()` —— 服务器告诉设备它能怎么用 Vision

```
void McpServer::ParseCapabilities(const cJSON* capabilities) {
    auto vision = cJSON_GetObjectItem(capabilities, "vision");
    if (cJSON_IsObject(vision)) {
        auto url = cJSON_GetObjectItem(vision, "url");
        auto token = cJSON_GetObjectItem(vision, "token");
        if (cJSON_IsString(url)) {
            auto camera = Board::GetInstance().GetCamera();
            if (camera) {
                std::string url_str = std::string(url->valuestring);
                std::string token_str;
                if (cJSON_IsString(token)) token_str = std::string(token->valuestring);
                camera->SetExplainUrl(url_str, token_str);
            }
        }
    }
}
```

在 `initialize` 消息里，云端会传 `capabilities.vision.url` 和 `token` 给设备——这样 `take_photo` 工具内部就知道 JPEG 该上传到哪里。

6.7.5 `ParseMessage()` —— MCP 主入口

被 `Application` 在 `on_incoming_json_` 收到 `type=mcp` 时调用：

```
void McpServer::ParseMessage(const cJSON* json) {
    // 1. 校验 JSONRPC 版本
    auto version = cJSON_GetObjectItem(json, "jsonrpc");
    if (version == nullptr || !cJSON_IsString(version) || strcmp(version->valuestring, "2.0")
        != 0) {
        ESP_LOGE(TAG, "Invalid JSONRPC version");
        return;
    }

    // 2. method 必填
    auto method = cJSON_GetObjectItem(json, "method");
    if (method == nullptr || !cJSON_IsString(method)) return;
    auto method_str = std::string(method->valuestring);

    // 3. notifications 类（无 id）直接忽略
    if (method_str.find("notifications") == 0) return;

    // 4. params 可选但若存在必须是 object
    auto params = cJSON_GetObjectItem(json, "params");
    if (params != nullptr && !cJSON_IsObject(params)) return;

    // 5. id 必须是 number
    auto id = cJSON_GetObjectItem(json, "id");
    if (id == nullptr || !cJSON_IsNumber(id)) return;
    auto id_int = id->valueint;

    // 6. 路由
    if (method_str == "initialize") {
        if (cJSON_IsObject(params)) {
            auto capabilities = cJSON_GetObjectItem(params, "capabilities");
            if (cJSON_IsObject(capabilities)) ParseCapabilities(capabilities);
        }
        auto app_desc = esp_app_get_description();
        std::string message = "{\"protocolVersion\":\"2024-11-05\",\"capabilities\":
        {\"tools\":{}},\"serverInfo\":{\"name\":\"" BOARD_NAME "\",\"version\":\"";
        message += app_desc->version;
        message += "\"}}";
        ReplyResult(id_int, message);
    } else if (method_str == "tools/list") {
        std::string cursor_str = "";
        bool list_user_only_tools = false;
        if (params != nullptr) {
            auto cursor = cJSON_GetObjectItem(params, "cursor");
            if (cJSON_IsString(cursor)) cursor_str = cursor->valuestring;
            auto with_user_tools = cJSON_GetObjectItem(params, "withUserTools");
            if (cJSON_IsBool(with_user_tools)) list_user_only_tools = with_user_tools-
        >valueint == 1;
        }
        GetToolsList(id_int, cursor_str, list_user_only_tools);
    } else if (method_str == "tools/call") {
        if (!cJSON_IsObject(params)) { ReplyError(id_int, "Missing params"); return; }
        auto tool_name = cJSON_GetObjectItem(params, "name");
        if (!cJSON_IsString(tool_name)) { ReplyError(id_int, "Missing name"); return; }
        auto tool_arguments = cJSON_GetObjectItem(params, "arguments");
        if (tool_arguments != nullptr && !cJSON_IsObject(tool_arguments)) {
            ReplyError(id_int, "Invalid arguments");
            return;
        }
        DoToolCall(id_int, std::string(tool_name->valuestring), tool_arguments);
```

```
    } else {
        ReplyError(id_int, "Method not implemented: " + method_str);
    }
}
```

支持的 3 个 method：

method	用途
`initialize`	客户端首次握手，交换协议版本和能力
`tools/list`	列出所有可用工具（支持分页）
`tools/call`	调用某个工具

忽略 `notifications/*`——MCP 规范里 notifications 是单向无响应消息，设备端不处理任何（因为目前没有需要主动接收的通知场景）。

`initialize` 返回的内容：

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2024-11-05",
    "capabilities": { "tools": {} },
    "serverInfo": {
      "name": "<BOARD_NAME>",
      "version": "<app_desc->version>"
    }
  }
}
```

宏 `BOARD_NAME` 由 Kconfig 注入（如 “xiaozhi-s3-cardputer”），`app_desc->version` 是 esp_app_format 自动生成的固件版本（git tag）。

6.7.6 `GetToolsList()` —— 分页输出工具清单

```
void McpServer::GetToolsList(int id, const std::string& cursor, bool list_user_only_tools) {
    const int max_payload_size = 8000;
    std::string json = "{\"tools\":[";

    bool found_cursor = cursor.empty();
    auto it = tools_.begin();
    std::string next_cursor = "";

    while (it != tools_.end()) {
        if (!found_cursor) {
            if ((*it)->name() == cursor) {
                found_cursor = true;
            } else {
                ++it;
                continue;
            }
        }

        if (!list_user_only_tools && (*it)->user_only()) {
            ++it;
            continue;
        }

        std::string tool_json = (*it)->to_json() + ",";
        if (json.length() + tool_json.length() + 30 > max_payload_size) {
            next_cursor = (*it)->name();
            break;
        }

        json += tool_json;
        ++it;
    }

    if (json.back() == ',') json.pop_back();

    if (json.back() == '[' && !tools_.empty()) {
        ESP_LOGE(TAG, "tools/list: Failed to add tool %s because of payload size limit",
        next_cursor.c_str());
        ReplyError(id, "Failed to add tool " + next_cursor + " because of payload size
        limit");
        return;
    }

    if (next_cursor.empty()) {
        json += "]}";
    } else {
        json += "],\"nextCursor\":\"" + next_cursor + "\"}";
    }

    ReplyResult(id, json);
}
```

为什么要分页?

- WebSocket / MQTT 单次消息体积有上限（这里硬限 8000 字节）；
- 一些板子注册了 20+ 工具，全发会爆栈或被网关丢弃；
- MCP 协议规范里 `nextCursor` 是标准分页字段。

算法：

1. `cursor` 空 → 从头开始；
2. `cursor` 非空 → 跳到 `name == cursor` 的位置开始（**继续上次未完成的列表**）；
3. 边累加 JSON 边算长度，**预留 30 字节给结尾结构**；
4. 加之前发现要超 → 设 `nextCursor = 当前工具名` 并停；
5. 如果一个工具都没加上去（单个工具大到超 8000）→ 报错；
6. 加 `nextCursor` 字段或不加。

LLM 收到带 `nextCursor` 的响应会再发一次 `tools/list` 带这个 cursor 拉下一页。

6.7.7 `DoToolCall()` —— 工具调用 + 异步执行

```
void McpServer::DoToolCall(int id, const std::string& tool_name, const cJSON* tool_arguments)
        {
    auto tool_iter = std::find_if(tools_.begin(), tools_.end(),
                                  [&tool_name](const McpTool* tool) {
                                      return tool->name() == tool_name;
                                  });
    if (tool_iter == tools_.end()) {
        ReplyError(id, "Unknown tool: " + tool_name);
        return;
    }

    PropertyList arguments = (*tool_iter)->properties();
    try {
        for (auto& argument : arguments) {
            bool found = false;
            if (cJSON_IsObject(tool_arguments)) {
                auto value = cJSON_GetObjectItem(tool_arguments, argument.name().c_str());
                if (argument.type() == kPropertyTypeBoolean && cJSON_IsBool(value)) {
                    argument.set_value<bool>(value->valueint == 1);
                    found = true;
                } else if (argument.type() == kPropertyTypeInteger && cJSON_IsNumber(value)) {
                    argument.set_value<int>(value->valueint);
                    found = true;
                } else if (argument.type() == kPropertyTypeString && cJSON_IsString(value)) {
                    argument.set_value<std::string>(value->valuestring);
                    found = true;
                }
            }
            if (!argument.has_default_value() && !found) {
                ReplyError(id, "Missing valid argument: " + argument.name());
                return;
            }
        }
    } catch (const std::exception& e) {
        ReplyError(id, e.what());
        return;
    }

    // Use main thread to call the tool
    auto& app = Application::GetInstance();
    app.Schedule([this, id, tool_iter, arguments = std::move(arguments)]() {
        try {
            ReplyResult(id, (*tool_iter)->Call(arguments));
        } catch (const std::exception& e) {
            ReplyError(id, e.what());
        }
    });
}
```

完整流程：

1. **找工具**：`std::find_if` 按 name 查；找不到立即回错。
2. **拷贝默认参数模板**：`PropertyList arguments = (*tool_iter)->properties();` ——拷贝一份**模板**，再覆盖具体值。**为什么拷贝**？避免多个同时调用互相覆盖，并发安全。
3. **填充参数 + 类型校验**：遍历每个参数，从 `tool_arguments` 里按名字找，类型必须严格匹配（boolean ↔ bool / integer ↔ number / string ↔ string）；未找到且无默认值 → 报错。**`set_value` 内部还会做范围检查**，越界抛 `std::invalid_argument`。

4. **catch + ReplyError**：任何参数异常立即回错，不进入 callback。
5. **异步执行**：用 `Schedule` 推到主循环跑 callback。**`arguments = std::move(arguments)` 是 C++14 init capture**——把 lambda 内的 arguments 用 move 构造，避免再拷贝一份。
6. **callback 内异常也 catch**：例如 `take_photo` 拍照失败抛 runtime_error。

为什么所有 callback 在主循环跑?

- 工具可能动 LVGL / 显示 / Audio 这些非线程安全的子系统；
- 多个工具调用要串行化，不能并发（比如同时调两个 `set_volume`）；
- 错误日志和上报统一在一处。

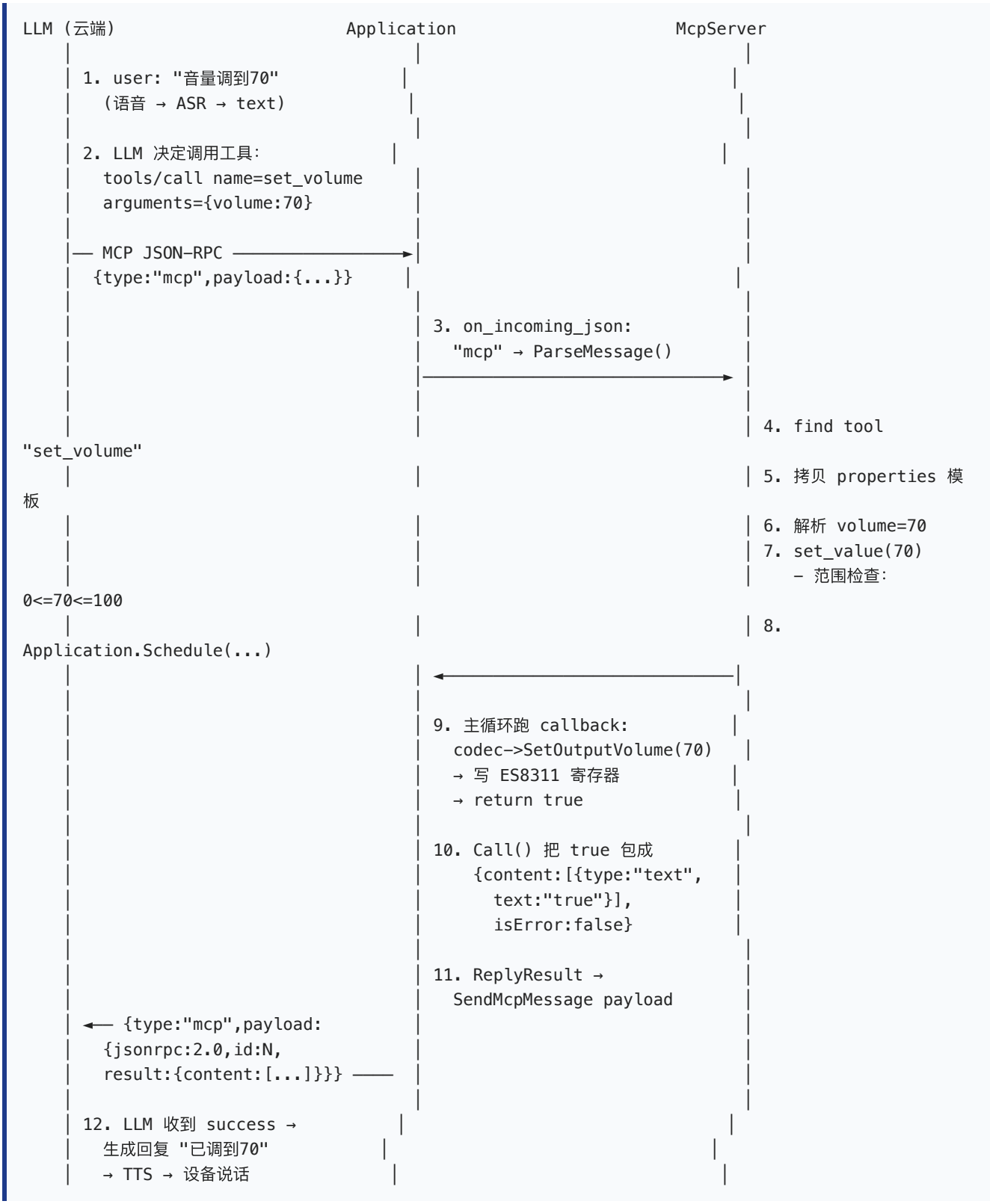
6.7.8 `ReplyResult()` / `ReplyError()`

```
void McpServer::ReplyResult(int id, const std::string& result) {
    std::string payload = "{\"jsonrpc\":\"2.0\",\"id\":";
    payload += std::to_string(id) + ",\"result\":";
    payload += result;
    payload += "}";
    Application::GetInstance().SendMcpMessage(payload);
}

void McpServer::ReplyError(int id, const std::string& message) {
    std::string payload = "{\"jsonrpc\":\"2.0\",\"id\":";
    payload += std::to_string(id);
    payload += ",\"error\":{\"message\":\"";
    payload += message;
    payload += "\"}}";
    Application::GetInstance().SendMcpMessage(payload);
}
```

手写字符串拼接而不是 cJSON：因为 `result` 已经是合法 JSON 字符串，直接嵌入比 cJSON 解析再序列化快得多。`Application::SendMcpMessage` 内部再包一层 `{type:mcp, payload: ...}` 信封发出去。

6.8 一次完整的工具调用时序

以”调音量到 70”为例：



6.9 本章用到的关键 C++ 技术

技术	应用
`std::variant<...>` + `std::holds_alternative` + `std::get<T>`	Property::value_, ReturnValue 多类型容器
`std::optional<int>`	min/max 可空范围限制
`std::function<ReturnValue(const PropertyList&)>`	工具回调统一签名
`if constexpr (std::is_same_v<T, int>)`	编译期分支
模板构造 + SFINAE 替代	Property 的多 ctor 重载
range-for + begin/end	for (auto& arg : arguments)
`std::find_if` + lambda	工具查找
`std::move(tools_)` + insert	AddCommonTools 两段式注册
init capture `arguments = std::move(arguments)`	C++14 lambda 移动捕获
try/catch 双层	参数解析期 + callback 执行期分别捕获
mbedtls Base64 encode (两步法)	图像 base64
dynamic_cast 检测子类型	区分 LvglDisplay / OledDisplay
`heap_caps_malloc(MALLOC_CAP_8BIT)`	PSRAM 上分配大块图像缓冲
手写 multipart/form-data	snapshot 工具的 HTTP 上传
Meyers 单例（静态局部）	McpServer::GetInstance

6.10 设计哲学小结

1. **JSON Schema 描述自己**：每个 Property → 一片 schema；每个 Tool → 完整 inputSchema。让 LLM 自助理解工具能力，不需要 prompt 里硬编码。
2. **prompt cache 友好**：通用工具放前面，板级工具放后面，最大化命中云端 LLM 的 prompt cache。
3. **user-only 权限分级**：危险操作（reboot / upgrade）对 LLM 不可见，只暴露给厂商客户端。
4. **异步执行 + 主线程串行化**：所有工具 callback 都 Schedule 到主循环跑，避免并发问题。
5. **强类型 + 范围限制 + 默认值**：边界全在 Property 类内固化，调用方不可能传出错的值进 callback。
6. **分页 + 8000 字节硬限**：兼顾大型工具集 + 受限传输通道。
7. **错误两阶段捕获**：参数解析期 / callback 执行期两次 try/catch，错误码精准。

6.11 看完本章你应该掌握的

- MCP 是什么、为什么放在设备端
- 4 层类结构（Property → PropertyList → McpTool → McpServer）
- Property 4 种构造场景对应的语义
- `std::variant` / `std::optional` 的实战用法
- ReturnValue 5 种类型的包装方式
- `user_only_` 工具的权限隔离
- `AddCommonTools` 两段式注册的精妙（prompt cache）
- initialize / tools/list / tools/call 三种 method 完整处理

- 分页机制 + 8000 字节硬限
- DoToolCall 的两阶段异常捕获
- ImageContent 用两步 base64 编码
- snapshot 工具的 multipart/form-data 手写实现

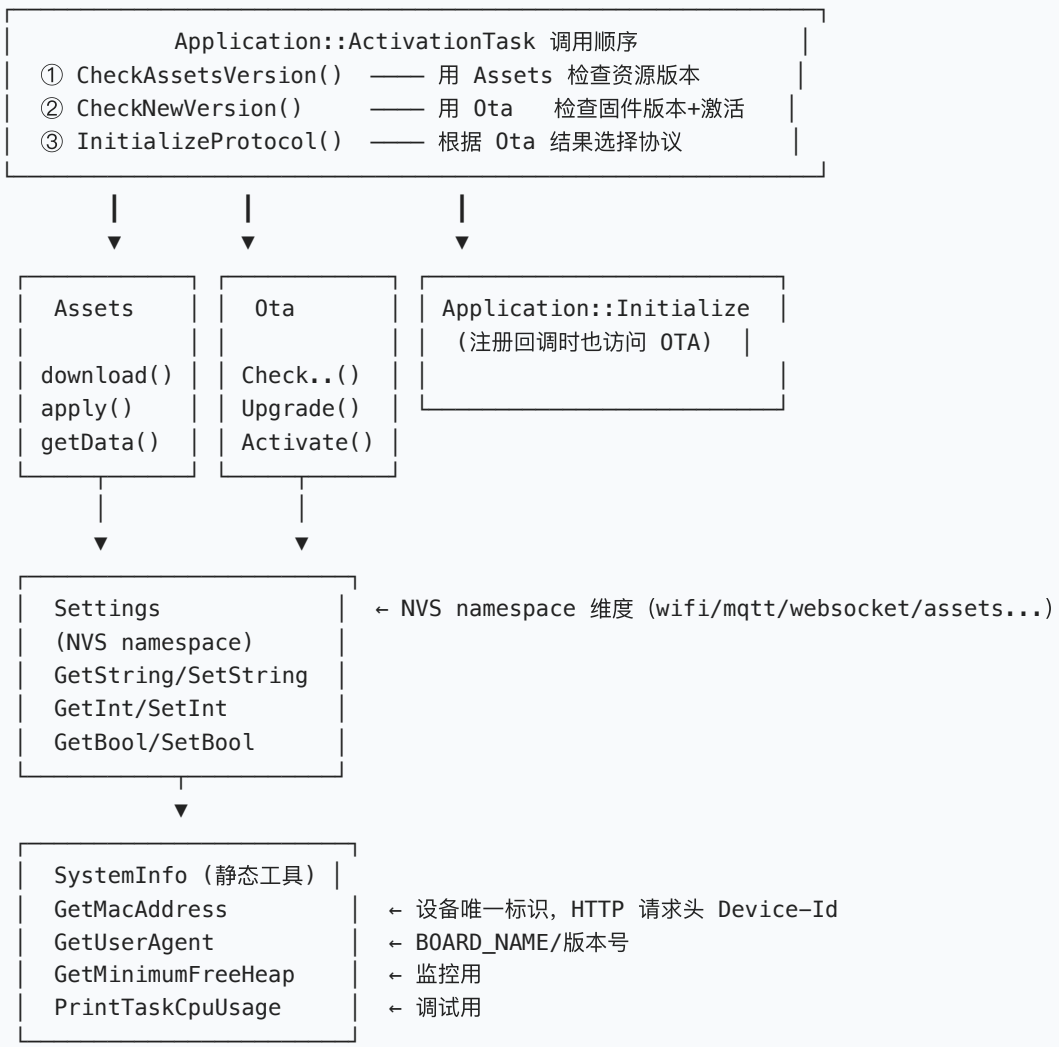
下一章覆盖 `ota / assets / settings / system_info` 系统服务层。

第 7 章 系统服务层：OTA / Assets / Settings / SystemInfo

这一章覆盖 4 个独立但相互配合的”系统级”服务文件：

- `settings.{h,cc}` —— NVS（Non-Volatile Storage）配置存取（最底层，所有其它模块都用）
- `system_info.{h,cc}` —— 设备信息查询（MAC、芯片型号、内存、任务列表）
- `ota.{h,cc}` —— 检查/下载/激活/安装固件（最复杂，含 HMAC 设备激活）
- `assets.{h,cc}` —— 资源分区管理（语音模型、字体、表情、皮肤），含下载和 mmap

7.1 整体关系图



7.2 `settings.{h,cc}` —— NVS 配置封装

109 行代码，最简洁但被全项目使用最多。

7.2.1 NVS 是什么

NVS（Non-Volatile Storage）是 ESP-IDF 提供的键值对持久化机制，落盘在专门的 NVS 分区（`partitions/v1/16mb.csv` 里能看到）。特点：

- **键值结构**（key 最长 15 字节）；
- **支持 `namespace` 隔离**（一个项目可用多个 namespace）；
- **掉电安全**（写入后调用 `commit` 持久化）；
- **磨损均衡**（NVS 自己处理 flash 寿命问题）；
- **不同类型有不同 API**（`nvs_get_str / nvs_get_i32 / nvs_get_u8` 等）。

7.2.2 RAII 风格的 `Settings` 类

```
Settings::Settings(const std::string& ns, bool read_write)
    : ns_(ns), read_write_(read_write) {
    nvs_open(ns.c_str(), read_write_ ? NVS_READWRITE : NVS_READONLY, &nvs_handle_);
}

Settings::~Settings() {
    if (nvs_handle_ != 0) {
        if (read_write_ && dirty_) {
            ESP_ERROR_CHECK(nvs_commit(nvs_handle_));
        }
        nvs_close(nvs_handle_);
    }
}
```

经典 RAII：

- 构造 = 打开句柄；
- 析构 =（如有改动）提交 + 关闭句柄；
- 用法：在栈上创建 → 离开作用域自动提交关闭。

`dirty_` 标志的作用：只有真改过才提交，避免每次”只读检查”也触发 flash 写入（NVS commit 有 flash 操作开销）。

典型使用：

```
{
    Settings settings("mqtt", true);   // 打开 mqtt namespace, 读写
    settings.SetString("endpoint", "mqtts://server.com:8883");
    settings.SetInt("keepalive", 240);
    // 析构 → 自动 commit
}
```

7.2.3 三种类型的存取

```
std::string Settings::GetString(const std::string& key, const std::string& default_value) {
    if (nvs_handle_ == 0) return default_value;
    size_t length = 0;
    if (nvs_get_str(nvs_handle_, key.c_str(), nullptr, &length) != ESP_OK) {
        return default_value;
    }
    std::string value;
    value.resize(length);
    ESP_ERROR_CHECK(nvs_get_str(nvs_handle_, key.c_str(), value.data(), &length));
    while (!value.empty() && value.back() == '\0') {
        value.pop_back();
    }
    return value;
}
```

两步获取字符串：

1. 传 nullptr 让 NVS 告诉你需要多大 buffer；
2. resize string 后再读到 buffer。

最后 `while (back == '\0') pop_back()` 去掉 NVS 存储末尾的 null 终止符——`nvs_get_str` 会把 C 字符串的 `\0` 也读进来。

```
int32_t Settings::GetInt(const std::string& key, int32_t default_value) {
    if (nvs_handle_ == 0) return default_value;
    int32_t value;
    if (nvs_get_i32(nvs_handle_, key.c_str(), &value) != ESP_OK) return default_value;
    return value;
}

bool Settings::GetBool(const std::string& key, bool default_value) {
    if (nvs_handle_ == 0) return default_value;
    uint8_t value;
    if (nvs_get_u8(nvs_handle_, key.c_str(), &value) != ESP_OK) return default_value;
    return value != 0;
}
```

`int32` 直接读，`bool` 用 `uint8_t` 当容器（NVS 没有原生 bool 类型）。

7.2.4 `Set*` + `dirty_` 标志

```
void Settings::SetString(const std::string& key, const std::string& value) {
    if (read_write_) {
        ESP_ERROR_CHECK(nvs_set_str(nvs_handle_, key.c_str(), value.c_str()));
        dirty_ = true;
    } else {
        ESP_LOGW(TAG, "Namespace %s is not open for writing", ns_.c_str());
    }
}
```

写操作都先检查 `read_write_`，再调对应 `nvs_set_*`，最后置 `dirty_`。

注意 `ESP_ERROR_CHECK` 是**会 panic 的宏**——NVS 出错（如分区满、key 太长）会直接 abort 整个程序。对配置存储而言，写不进去本来就是致命问题，宁可崩了重启。

7.2.5 `EraseKey / EraseAll`

```
void Settings::EraseKey(const std::string& key) {
    if (read_write_) {
        auto ret = nvs_erase_key(nvs_handle_, key.c_str());
        if (ret != ESP_ERR_NVS_NOT_FOUND) {
            ESP_ERROR_CHECK(ret);
        }
    }
}
```

`EraseKey` 对 `ESP_ERR_NVS_NOT_FOUND` 容错——擦不存在的 key 不算错误。

7.2.6 项目中已知的 NVS namespace

通过 `rg 'Settings\("'` 可以列出：

namespace	内容	谁写
`wifi`	`ota_url`（OTA 服务器 URL）、wifi SSID/密码	OTA 阶段 / wifi_board
`mqtt`	endpoint / client_id / username / password / keepalive / publish_topic	OTA Check 写
`websocket`	url / token / version	OTA Check 写
`assets`	download_url（自定义资源 URL）	MCP 工具 `self.assets.set_download_url`
`audio`	volume（用户音量持久化）	AudioCodec::SetOutputVolume
`display`	brightness / theme	LvglDisplay
`board`	uuid（板子软件 UUID，首次启动随机生成）	Board::GetUuid

7.3 `system_info.{h,cc}` —— 设备信息查询

152 行代码，全是 `static` 静态方法。无状态、随处可用。

7.3.1 `GetMacAddress()` —— 设备唯一标识

```
std::string SystemInfo::GetMacAddress() {
    uint8_t mac[6];
#if CONFIG_IDF_TARGET_ESP32P4
    esp_wifi_get_mac(WIFI_IF_STA, mac);
#else
    esp_read_mac(mac, ESP_MAC_WIFI_STA);
#endif
    char mac_str[18];
    snprintf(mac_str, sizeof(mac_str), "%02x:%02x:%02x:%02x:%02x:%02x", mac[0], mac[1],
        mac[2], mac[3], mac[4], mac[5]);
    return std::string(mac_str);
}
```

- **ESP32-P4 特殊化**：P4 自身没有 Wi-Fi，要走 `esp_wifi_remote`（通过 SPI 接外部 ESP32-C6 Wi-Fi 模块）拿 MAC；
- 其它芯片用 `esp_read_mac(ESP_MAC_WIFI_STA)` —— Wi-Fi station 模式的烧入 MAC（全球唯一）；
- 格式化为 `xx:xx:xx:xx:xx:xx` 字符串（HTTP header 用）。

7.3.2 `GetUserAgent()`

```
std::string SystemInfo::GetUserAgent() {
    auto app_desc = esp_app_get_description();
    auto user_agent = std::string(BOARD_NAME "/") + app_desc->version;
    return user_agent;
}
```

`BOARD_NAME` 是 Kconfig 注入的板子名，`app_desc->version` 是 ESP-IDF 自动从 git tag 注入的版本号。形如 `xiaozhi-s3-cardputer/1.5.0`。

HTTP 请求都带这个 User-Agent，服务器侧可以做版本分流。

7.3.3 `GetFlashSize / GetFreeHeapSize / GetMinimumFreeHeapSize`

```
size_t SystemInfo::GetFreeHeapSize() {
    return esp_get_free_heap_size();
}

size_t SystemInfo::GetMinimumFreeHeapSize() {
    return esp_get_minimum_free_heap_size();
}
```

最小可用堆是个**历史水位标记**——任何时候堆使用峰值最高时的剩余值。监控这个能发现”运行了 1 小时之后内存就吃紧了”这种泄漏问题。

7.3.4 `PrintTaskCpuUsage()` —— 任务 CPU 占用统计

```
esp_err_t SystemInfo::PrintTaskCpuUsage(TickType_t xTicksToWait) {
    TaskStatus_t *start_array = NULL, *end_array = NULL;
    UBaseType_t start_array_size, end_array_size;
    configRUN_TIME_COUNTER_TYPE start_run_time, end_run_time;

    // 1. 第一次采样
    start_array_size = uxTaskGetNumberOfTasks() + ARRAY_SIZE_OFFSET;
    start_array = (TaskStatus_t*)malloc(sizeof(TaskStatus_t) * start_array_size);
    start_array_size = uxTaskGetSystemState(start_array, start_array_size, &start_run_time);

    vTaskDelay(xTicksToWait);   // 2. 等一段时间

    // 3. 第二次采样
    end_array_size = uxTaskGetNumberOfTasks() + ARRAY_SIZE_OFFSET;
    end_array = (TaskStatus_t*)malloc(sizeof(TaskStatus_t) * end_array_size);
    end_array_size = uxTaskGetSystemState(end_array, end_array_size, &end_run_time);

    // 4. 计算差值并打印
    total_elapsed_time = (end_run_time - start_run_time);
    for (int i = 0; i < start_array_size; i++) {
        // 在 end_array 里找同一个 handle 匹配
        for (int j = 0; j < end_array_size; j++) {
            if (start_array[i].xHandle == end_array[j].xHandle) { k = j; break; }
        }
        if (k >= 0) {
            uint32_t task_elapsed_time = end_array[k].ulRunTimeCounter -
        start_array[i].ulRunTimeCounter;
            uint32_t percentage_time = (task_elapsed_time * 100UL) / (total_elapsed_time *
        CONFIG_FREERTOS_NUMBER_OF_CORES);
            printf("| %-16s | %8lu | %4lu%%\n", start_array[i].pcTaskName, task_elapsed_time,
        percentage_time);
        }
    }
}
```

思路：

- 两次采样所有任务的 `ulRunTimeCounter`（FreeRTOS 内部维护的累计运行 tick）；
- 求差 = 这段时间该任务占的 CPU 时间；
- 除以总耗时 = CPU 占比；
- 多核（双核）要把分母乘 `CONFIG_FREERTOS_NUMBER_OF_CORES`。

输出例子：

```
| Task             | Run Time | Percentage
| main             | 12345    |  5%
| audio_input      | 65432    | 32%
| audio_output     | 38291    | 19%
| afe_processor    | 28194    | 14%
| ...
```

调试 CPU 瓶颈用——本项目主要在 `wake_word_test.cc` 和某些板子的 debug 路径下用。

7.3.5 `PrintHeapStats()` —— 堆统计

```
void SystemInfo::PrintHeapStats() {
    int free_sram = heap_caps_get_free_size(MALLOC_CAP_INTERNAL);
    int min_free_sram = heap_caps_get_minimum_free_size(MALLOC_CAP_INTERNAL);
    ESP_LOGI(TAG, "free sram: %u minimal sram: %u", free_sram, min_free_sram);
}
```

`MALLOC_CAP_INTERNAL` 专门统计内置 SRAM（不含 PSRAM）。SRAM 满 = 系统快崩，PSRAM 满 = 性能严重下降但不会立即崩。

7.4 `ota.{h,cc}` —— 固件升级与设备激活

473 行代码，是本章最复杂的文件。功能分四块：

1. **检查版本**：调云端接口、对比版本号、缓存 mqtt/websocket 配置；
2. **设备激活**：用 efuse 里烧的序列号 + HMAC challenge/response 完成首次激活；
3. **下载安装固件**：流式写入 OTA 分区；
4. **回滚保护**：启动后标记当前 firmware 为 valid。

7.4.1 构造 —— 读取序列号

```
Ota::Ota() {
#ifdef ESP_EFUSE_BLOCK_USR_DATA
    uint8_t serial_number[33] = {0};
    if (esp_efuse_read_field_blob(ESP_EFUSE_USER_DATA, serial_number, 32 * 8) == ESP_OK) {
        if (serial_number[0] == 0) {
            has_serial_number_ = false;
        } else {
            serial_number_ = std::string(reinterpret_cast<char*>(serial_number), 32);
            has_serial_number_ = true;
        }
    }
#endif
}
```

eFuse 是 ESP32 内部的一次性可编程位，烧入后**不可修改**。`ESP_EFUSE_USER_DATA` 是 256 位（32 字节）的用户区——出厂时厂商烧 32 字节序列号进去。

读法：

- `esp_efuse_read_field_blob` 第三参是 bit 数，`32 * 8 = 256` bit；
- 第一个字节为 0 → 未烧序列号（属于"未量产"的开发板）；
- 有序列号 → 启用激活模式 v2（HMAC challenge-response）。

7.4.2 `SetupHttp()` —— 共用的 HTTP 头部

```
std::unique_ptr<Http> Ota::SetupHttp() {
    auto& board = Board::GetInstance();
    auto network = board.GetNetwork();
    auto http = network->CreateHttp(0);
    auto user_agent = SystemInfo::GetUserAgent();
    http->SetHeader("Activation-Version", has_serial_number_ ? "2" : "1");
    http->SetHeader("Device-Id", SystemInfo::GetMacAddress().c_str());
    http->SetHeader("Client-Id", board.GetUuid());
    if (has_serial_number_) {
        http->SetHeader("Serial-Number", serial_number_.c_str());
    }
    http->SetHeader("User-Agent", user_agent);
    http->SetHeader("Accept-Language", Lang::CODE);
    http->SetHeader("Content-Type", "application/json");
    return http;
}
```

OTA 请求的标准头：

- `Activation-Version`：1 或 2，告诉服务端用哪套激活协议；
- `Device-Id`：MAC（硬件唯一，伪造代价高）；
- `Client-Id`：软件 UUID（首次启动 NVS 随机生成）；
- `Serial-Number`：efuse 烧的序列号（仅 Activation-Version 2）；
- `User-Agent`：板名/版本号；
- `Accept-Language`：语言代码（zh-CN / en-US / 日语等）。

7.4.3 `CheckVersion()` —— 一次拿全部配置

```
esp_err_t Ota::CheckVersion() {
    auto& board = Board::GetInstance();
    auto app_desc = esp_app_get_description();
    current_version_ = app_desc->version;

    std::string url = GetCheckVersionUrl();
    if (url.length() < 10) return ESP_ERR_INVALID_ARG;

    auto http = SetupHttp();
    std::string data = board.GetSystemInfoJson();
    std::string method = data.length() > 0 ? "POST" : "GET";
    http->SetContent(std::move(data));

    if (!http->Open(method, url)) return http->GetLastError();
    auto status_code = http->GetStatusCode();
    if (status_code != 200) return status_code;
    data = http->ReadAll();
    http->Close();
```

向 `ota_url`（默认从 Kconfig 的 `CONFIG_OTA_URL` 来）发 POST 请求，请求体是设备的 `GetSystemInfoJson()`（含芯片、版本、剩余内存、电量等），服务端响应一个聚合 JSON。

为什么用 POST 而不是 GET？因为要把设备信息送上去——服务端可能根据芯片型号下发不同 mqtt 配置，根据版本号决定是否给新固件 URL。

服务端响应示例：

```
{
  "activation": {
    "message": "请打开 https://xz.com/activate 输入激活码：",
    "code": "ABC123",
    "challenge": "0x1234abcd...",
    "timeout_ms": 60000
  },
  "mqtt": {
    "endpoint": "mqtt.example.com:8883",
    "client_id": "device_xxx",
    "username": "...",
    "password": "...",
    "publish_topic": "device/in"
  },
  "websocket": {
    "url": "wss://ws.example.com/",
    "token": "Bearer xxx",
    "version": 3
  },
  "server_time": {
    "timestamp": 1715500000000,
    "timezone_offset": 480
  },
  "firmware": {
    "version": "1.6.0",
    "url": "https://firmware.example.com/v1.6.0.bin",
    "force": 0
  }
}
```

下面逐段解析。

7.4.3.1 解析 activation 段

```
has_activation_code_ = false;
has_activation_challenge_ = false;
cJSON *activation = cJSON_GetObjectItem(root, "activation");
if (cJSON_IsObject(activation)) {
    cJSON* message = cJSON_GetObjectItem(activation, "message");
    if (cJSON_IsString(message)) activation_message_ = message->valuestring;
    cJSON* code = cJSON_GetObjectItem(activation, "code");
    if (cJSON_IsString(code)) {
        activation_code_ = code->valuestring;
        has_activation_code_ = true;
    }
    cJSON* challenge = cJSON_GetObjectItem(activation, "challenge");
    if (cJSON_IsString(challenge)) {
        activation_challenge_ = challenge->valuestring;
        has_activation_challenge_ = true;
    }
    cJSON* timeout_ms = cJSON_GetObjectItem(activation, "timeout_ms");
    if (cJSON_IsNumber(timeout_ms)) {
        activation_timeout_ms_ = timeout_ms->valueint;
    }
}
```

两种激活路径：

- **`code` 模式**（Activation-Version 1，无 efuse 序列号）：服务端给出一串数字 code，设备语音播报 + 屏幕显示，让用户去网页输入这段码绑定设备账号。Application 在 `ShowActivationCode` 里处理。
- **`challenge` 模式**（Activation-Version 2，有 efuse 序列号）：服务端给一个随机 challenge 字符串，设备用 efuse 里的 HMAC key 算 HMAC-SHA256，回传给服务端验证。这是真·硬件级身份认证，不可伪造。详见 7.4.6。

7.4.3.2 解析 mqtt / websocket 段并写 NVS

```
has_mqtt_config_ = false;
cJSON *mqtt = cJSON_GetObjectItem(root, "mqtt");
if (cJSON_IsObject(mqtt)) {
    Settings settings("mqtt", true);
    cJSON *item = NULL;
    cJSON_ArrayForEach(item, mqtt) {
        if (cJSON_IsString(item)) {
            if (settings.GetString(item->string) != item->valuestring) {
                settings.SetString(item->string, item->valuestring);
            }
        } else if (cJSON_IsNumber(item)) {
            if (settings.GetInt(item->string) != item->valueint) {
                settings.SetInt(item->string, item->valueint);
            }
        }
    }
    has_mqtt_config_ = true;
}
```

要点：

- 用 `cJSON_ArrayForEach` 遍历 mqtt object 的所有 key（cJSON 实现里 object 内部也用链表，能用 ArrayForEach）；
- **遍历前先比对再写**：避免对未变化的字段无谓的 flash 写入（NVS 写入会增加 flash 寿命磨损）；
- 由 Settings 析构时统一 `commit`。

`websocket` 段处理完全一样的逻辑。

7.4.3.3 解析 server_time 段 —— NTP 替代

```
has_server_time_ = false;
cJSON *server_time = cJSON_GetObjectItem(root, "server_time");
if (cJSON_IsObject(server_time)) {
    cJSON *timestamp = cJSON_GetObjectItem(server_time, "timestamp");
    cJSON *timezone_offset = cJSON_GetObjectItem(server_time, "timezone_offset");
    if (cJSON_IsNumber(timestamp)) {
        struct timeval tv;
        double ts = timestamp->valuedouble;
        if (cJSON_IsNumber(timezone_offset)) {
            ts += (timezone_offset->valueint * 60 * 1000);  // 分钟转毫秒
        }
        tv.tv_sec = (time_t)(ts / 1000);
        tv.tv_usec = (suseconds_t)((long long)ts % 1000) * 1000;
        settimeofday(&tv, NULL);
        has_server_time_ = true;
    }
}
```

走自家服务器同步时间替代 NTP 协议：

- NTP 在某些受限网络（公司防火墙、运营商封 UDP）不可用；
- HTTP 已经能通 → 顺便把时间塞进响应；
- `timezone_offset` 是分钟为单位的时区偏移（如东八区 = 480）；
- `settimeofday` 直接设置系统时间——后续 `time(NULL)` / RTC / 日志时间戳全都对。

注意 `valuedouble` 而非 `valueint`——`int` 最多 ~21 亿 = 2.1×10^9 毫秒约 24 天，存不下 1715500000000 这种 13 位毫秒时间戳，必须用 double。

7.4.3.4 解析 firmware 段

```
has_new_version_ = false;
cJSON *firmware = cJSON_GetObjectItem(root, "firmware");
if (cJSON_IsObject(firmware)) {
    cJSON *version = cJSON_GetObjectItem(firmware, "version");
    if (cJSON_IsString(version)) firmware_version_ = version->valuestring;
    cJSON *url = cJSON_GetObjectItem(firmware, "url");
    if (cJSON_IsString(url)) firmware_url_ = url->valuestring;

    if (cJSON_IsString(version) && cJSON_IsString(url)) {
        has_new_version_ = IsNewVersionAvailable(current_version_, firmware_version_);
        cJSON *force = cJSON_GetObjectItem(firmware, "force");
        if (cJSON_IsNumber(force) && force->valueint == 1) {
            has_new_version_ = true;
        }
    }
}
```

`force: 1` 强制覆盖比较结果——即使版本号相同或更老也升级。用于回滚下发或紧急修复版。

7.4.4 `ParseVersion / IsNewVersionAvailable` —— 语义化版本比较

```
std::vector<int> Ota::ParseVersion(const std::string& version) {
    std::vector<int> versionNumbers;
    std::stringstream ss(version);
    std::string segment;
    while (std::getline(ss, segment, '.')) {
        versionNumbers.push_back(std::stoi(segment));
    }
    return versionNumbers;
}

bool Ota::IsNewVersionAvailable(const std::string& currentVersion, const std::string&
        newVersion) {
    std::vector<int> current = ParseVersion(currentVersion);
    std::vector<int> newer = ParseVersion(newVersion);
    for (size_t i = 0; i < std::min(current.size(), newer.size()); ++i) {
        if (newer[i] > current[i]) return true;
        else if (newer[i] < current[i]) return false;
    }
    return newer.size() > current.size();
}
```

逐段比较 `1.2.3` → `[1, 2, 3]`：

- 高位优先（1.2.0 < 2.0.0）；
- 相等继续比下一位；
- 长度不等以更长为新（1.2 < 1.2.1）。

注意没有处理 `1.2.3-rc1` 这种预发布标签——本项目版本号约定纯数字。

7.4.5 `Upgrade()` —— 流式下载 + 写入 OTA 分区

最复杂的函数（104 行），完整复述：

```
bool Ota::Upgrade(const std::string& firmware_url, std::function<void(int, size_t)> callback)
        {
    esp_ota_handle_t update_handle = 0;
    auto update_partition = esp_ota_get_next_update_partition(NULL);
    if (update_partition == NULL) return false;
```

关键概念：ESP32 用 **A/B 双分区 OTA**——`ota_0` 和 `ota_1` 交替使用。
`esp_ota_get_next_update_partition(NULL)` 返回**当前没在运行的那一个**——升级时写它，不影响正在运行的固件。

```
    bool image_header_checked = false;
    std::string image_header;
    auto http = network->CreateHttp(0);
    if (!http->Open("GET", firmware_url)) return false;
    if (http->GetStatusCode() != 200) return false;
    size_t content_length = http->GetBodyLength();
    if (content_length == 0) return false;
```

GET 固件 URL，获取 Content-Length 用于进度条。

```
    char buffer[512];
    size_t total_read = 0, recent_read = 0;
    auto last_calc_time = esp_timer_get_time();
    while (true) {
        int ret = http->Read(buffer, sizeof(buffer));
        if (ret < 0) return false;

        recent_read += ret;
        total_read += ret;
        if (esp_timer_get_time() - last_calc_time >= 1000000 || ret == 0) {
            size_t progress = total_read * 100 / content_length;
            if (callback) callback(progress, recent_read);
            last_calc_time = esp_timer_get_time();
            recent_read = 0;
        }
        if (ret == 0) break;
```

512 字节一块流式读，每秒计算一次进度（百分比 + 实时速度 B/s）调 callback。`Application::CheckNewVersion` 注册的 callback 会更新屏幕上“下载中 35% (240KB/s)”。

```
        if (!image_header_checked) {
            image_header.append(buffer, ret);
            if (image_header.size() >= sizeof(esp_image_header_t) +
        sizeof(esp_image_segment_header_t) + sizeof(esp_app_desc_t)) {
                esp_app_desc_t new_app_info;
                memcpy(&new_app_info, image_header.data() + sizeof(esp_image_header_t) +
        sizeof(esp_image_segment_header_t), sizeof(esp_app_desc_t));

                auto current_version = esp_app_get_description()->version;

                if (esp_ota_begin(update_partition, OTA_WITH_SEQUENTIAL_WRITES,
        &update_handle)) {
                    esp_ota_abort(update_handle);
                    return false;
                }
                image_header_checked = true;
                std::string().swap(image_header);   // ★ 立即释放临时 buffer
            }
        }
        auto err = esp_ota_write(update_handle, buffer, ret);
        if (err != ESP_OK) {
            esp_ota_abort(update_handle);
            return false;
        }
    }
```

首个块的头部校验：

- ESP32 固件的前部依次是 `esp_image_header_t` + `esp_image_segment_header_t` + `esp_app_desc_t`；
- 解析出 `esp_app_desc_t` 里的版本号打日志（不强制检查，因为可能是同版本 force 升级）；
- 通过 `esp_ota_begin` 申请写入句柄；
- `OTA_WITH_SEQUENTIAL_WRITES` flag：声明会顺序写入，让底层做优化（无需擦掉整个分区，分段擦分段写）；
- `std::string().swap(image_header)` 是**强制释放 string 内存**的 idiom——`clear()` 只是 size=0 容量还在。

后续每块都通过 `esp_ota_write` 写到分区。

```
    http->Close();

    esp_err_t err = esp_ota_end(update_handle);
    if (err != ESP_OK) {
        if (err == ESP_ERR_OTA_VALIDATE_FAILED) {
            ESP_LOGE(TAG, "Image validation failed, image is corrupted");
        }
        return false;
    }

    err = esp_ota_set_boot_partition(update_partition);
    if (err != ESP_OK) return false;
    return true;
}
```

收尾：

- `esp_ota_end` 内部做**整体 SHA-256 校验**（ESP32 固件末尾有 hash），不通过返回 `ESP_ERR_OTA_VALIDATE_FAILED`；
- `esp_ota_set_boot_partition` 把启动指针指向新分区，但**不立即重启**——交给上层（`Application::UpgradeFirmware` 之后会调 `esp_restart()`）。

7.4.6 `Activate()` + `GetActivationPayload()` —— 硬件 HMAC 激活

```
std::string Ota::GetActivationPayload() {
    if (!has_serial_number_) return "{}";

    std::string hmac_hex;
#ifdef SOC_HMAC_SUPPORTED
    uint8_t hmac_result[32];
    esp_err_t ret = esp_hmac_calculate(HMAC_KEY0,
                                       (uint8_t*)activation_challenge_.data(),
                                       activation_challenge_.size(),
                                       hmac_result);
    if (ret != ESP_OK) return "{}";

    for (size_t i = 0; i < sizeof(hmac_result); i++) {
        char buffer[3];
        sprintf(buffer, "%02x", hmac_result[i]);
        hmac_hex += buffer;
    }
#endif

    cJSON *payload = cJSON_CreateObject();
    cJSON_AddStringToObject(payload, "algorithm", "hmac-sha256");
    cJSON_AddStringToObject(payload, "serial_number", serial_number_.c_str());
    cJSON_AddStringToObject(payload, "challenge", activation_challenge_.c_str());
    cJSON_AddStringToObject(payload, "hmac", hmac_hex.c_str());
    // ...
}
```

ESP32 内置 HMAC 加速外设（`SOC_HMAC_SUPPORTED`，S3/C6 有，C3 没有）的工作机制：

- 出厂时厂商在 eFuse 烧入一个 **HMAC key**（key0/key1/key2 三个槽位之一）；
- key 一旦烧入**软件无法读出**——只能让硬件用它做 HMAC 运算；
- `esp_hmac_calculate(HMAC_KEY0, msg, len, out)` 让硬件用 key0 算 HMAC-SHA256；

- 即使攻击者拷贝了完整固件 + 序列号，没有原厂 eFuse 也算不出正确 HMAC。

```
esp_err_t Ota::Activate() {
    if (!has_activation_challenge_) return ESP_FAIL;

    std::string url = GetCheckVersionUrl();
    if (url.back() != '/') url += "/activate";
    else url += "activate";

    auto http = SetupHttp();
    std::string data = GetActivationPayload();
    http->SetContent(std::move(data));

    if (!http->Open("POST", url)) return ESP_FAIL;
    auto status_code = http->GetStatusCode();
    if (status_code == 202) return ESP_ERR_TIMEOUT;
    if (status_code != 200) return ESP_FAIL;
    return ESP_OK;
}
```

- 把 `{algorithm, serial_number, challenge, hmac}` POST 到 `/activate`；
- 202 Accepted = 服务器还没准备好（如用户还没在手机端输激活码绑定），让设备 5 秒后再试（Application::CheckNewVersion 里有重试循环）；
- 200 OK = 激活成功，下一次 CheckVersion 不会再下 activation 段。

7.4.7 `MarkCurrentVersionValid()` —— 防回滚保护

```
void Ota::MarkCurrentVersionValid() {
    auto partition = esp_ota_get_running_partition();
    if (strcmp(partition->label, "factory") == 0) return;

    esp_ota_img_states_t state;
    if (esp_ota_get_state_partition(partition, &state) != ESP_OK) return;

    if (state == ESP_OTA_IMG_PENDING_VERIFY) {
        ESP_LOGI(TAG, "Marking firmware as valid");
        esp_ota_mark_app_valid_cancel_rollback();
    }
}
```

Anti-Bricking 机制：

- ESP32 OTA 支持”先启动新固件，跑一段时间没崩才标记为 valid”；
- `esp_ota_set_boot_partition` 后第一次启动新固件，状态是 `ESP_OTA_IMG_PENDING_VERIFY`；
- 如果不调 `mark_app_valid_cancel_rollback`，**下次重启会自动回滚到旧固件**——保护因新固件有 bug 卡死的情况；
- 本函数在 `Application::Initialize` 早期调用——意思是”只要能跑到这一步说明新固件基本正常”。

注意 `factory` 分区跳过——出厂固件不参与 A/B 切换。

7.5 `assets.{h,cc}` —— 资源分区管理

533 行代码，管理一个独立的 `assets` flash 分区，里面打包了：

- 语音模型（ESP-SR 唤醒词模型 `srmodels.bin`）
- 字体（`fonts.bin`）
- 表情图片（emoji_collection / icon_collection）
- 主题皮肤（light / dark 的颜色 + 背景图）
- 布局配置（`layout_json`）

为什么单独一个分区？因为这些资源**比固件大、按需更新**（换一个语种 → 只换模型不刷固件）。

7.5.1 分区数据格式（手写的二进制协议）

```
offset 0:  stored_files     (uint32_t)  ← 文件数量
offset 4:  stored_chksum    (uint32_t)  ← 后续数据的校验和
offset 8:  stored_len       (uint32_t)  ← 元数据 + 数据总长

offset 12: 文件索引表 stored_files 个 mmap_assets_table
  - asset_name[32]
  - asset_size (uint32_t)
  - asset_offset (uint32_t)  ← 相对于"文件数据区"起始的偏移
  - asset_width (uint16_t)
  - asset_height (uint16_t)

offset 12 + sizeof(table)*stored_files: 各文件数据连续排列
  每个文件以 'Z' 'Z' magic 起始
```

7.5.2 `InitializePartition()` —— 内存映射 + 校验

```
bool Assets::InitializePartition() {
    partition_valid_ = false;
    checksum_valid_ = false;
    assets_.clear();

    partition_ = esp_partition_find_first(ESP_PARTITION_TYPE_ANY, ESP_PARTITION_SUBTYPE_ANY,
        "assets");
    if (partition_ == nullptr) return false;
```

按 label 查找 assets 分区（partitions/v1/16mb.csv 里定义了 label=“assets”）。

```
    int free_pages = spi_flash_mmap_get_free_pages(SPI_FLASH_MMAP_DATA);
    uint32_t storage_size = free_pages * 64 * 1024;
    if (storage_size < partition_->size) return false;

    esp_err_t err = esp_partition_mmap(partition_, 0, partition_->size,
        ESP_PARTITION_MMAP_DATA, (const void**)&mmap_root_, &mmap_handle_);
    if (err != ESP_OK) return false;

    partition_valid_ = true;
```

`esp_partition_mmap` 是 ESP32 的杀手锏 —— **把 flash 分区映射到 CPU 可寻址的虚拟地址**，之后用 `mmap_root_[offset]` 就能直接像内存一样访问，硬件 cache 加速。不用每次 `esp_partition_read` 把数据拷贝到 RAM。

mmap 占用 64KB-aligned 的虚拟地址槽（“pages”），先检查可用 pages 够不够，不够则报错。

```
    uint32_t stored_files = *(uint32_t*)(mmap_root_ + 0);
    uint32_t stored_chksum = *(uint32_t*)(mmap_root_ + 4);
    uint32_t stored_len = *(uint32_t*)(mmap_root_ + 8);

    if (stored_len > partition_->size - 12) return false;

    uint32_t calculated_checksum = CalculateChecksum(mmap_root_ + 12, stored_len);
    if (calculated_checksum != stored_chksum) return false;
    checksum_valid_ = true;

    for (uint32_t i = 0; i < stored_files; i++) {
        auto item = (const mmap_assets_table*)(mmap_root_ + 12 + i *
        sizeof(mmap_assets_table));
        auto asset = Asset{
            .size = static_cast<size_t>(item->asset_size),
            .offset = static_cast<size_t>(12 + sizeof(mmap_assets_table) * stored_files +
        item->asset_offset)
        };
        assets_[item->asset_name] = asset;
    }
    return checksum_valid_;
}
```

读 3 个 header 字段：文件数、校验和、长度。

`CalculateChecksum` 是最朴素的**累加和** mod 0x10000：

```
uint32_t Assets::CalculateChecksum(const char* data, uint32_t length) {
    uint32_t checksum = 0;
    for (uint32_t i = 0; i < length; i++) checksum += data[i];
    return checksum & 0xFFFF;
}
```

这种校验只能防"误写"——抗不住有针对性的篡改，但对 firmware 配套用够了（真要防篡改在固件签名层做）。

最后遍历索引表，把每个文件的 size + 绝对偏移塞进 `std::map<std::string, Asset>` 加速后续按名字查找。

7.5.3 `GetAssetData()` —— 按名字找

```
bool Assets::GetAssetData(const std::string& name, void*& ptr, size_t& size) {
    auto asset = assets_.find(name);
    if (asset == assets_.end()) return false;
    auto data = (const char*)(mmap_root_ + asset->second.offset);
    if (data[0] != 'Z' || data[1] != 'Z') {
        ESP_LOGE(TAG, "The asset %s is not valid with magic %02x%02x", name.c_str(), data[0],
        data[1]);
        return false;
    }
    ptr = static_cast<void*>(const_cast<char*>(data + 2));
    size = asset->second.size;
    return true;
}
```

返回的 `ptr` 直接指向 mmap 区域——零拷贝。后续 LVGL 字体/图像组件直接拿 ptr 用，省下数 MB 内存。

每个文件以 `ZZ` magic 起始作为防误读保险。

7.5.4 `Apply()` —— 解析 index.json 并加载所有资源

```
bool Assets::Apply() {
    void* ptr = nullptr;
    size_t size = 0;
    if (!GetAssetData("index.json", ptr, size)) return false;

    cJSON* root = cJSON_ParseWithLength(static_cast<char*>(ptr), size);
    if (root == nullptr) return false;

    cJSON* version = cJSON_GetObjectItem(root, "version");
    if (cJSON_IsNumber(version)) {
        if (version->valuedouble > 1) {
            ESP_LOGE(TAG, "The assets version %d is not supported, please upgrade the
        firmware", version->valueint);
            return false;
        }
    }
```

资源包入口是 `index.json`：

```
{
  "version": 1,
  "srmodels": "srmodels.bin",
  "text_font": "fonts/fonts.bin",
  "emoji_collection": [
    { "name": "happy", "file": "emojis/happy.bin" },
    { "name": "sad", "file": "emojis/sad.bin" }
  ],
  "icon_collection": [...],
  "skin": {
    "light": { "text_color": "#000000", "background_color": "#FFFFFF", "background_image":
        "bg_light.bin" },
    "dark": { ... }
  },
  "hide_subtitle": false,
  "layout": [...]
}
```

`version > 1` → 资源包格式比当前固件支持的版本更新，提示用户升级固件。**前向兼容性保护**。

7.5.4.1 加载 srmodels

```
cJSON* srmodels = cJSON_GetObjectItem(root, "srmodels");
if (cJSON_IsString(srmodels)) {
    std::string srmodels_file = srmodels->valuestring;
    if (GetAssetData(srmodels_file, ptr, size)) {
        if (models_list_ != nullptr) {
            esp_srmodel_deinit(models_list_);
            models_list_ = nullptr;
        }
        models_list_ = srmodel_load(static_cast<uint8_t*>(ptr));
        if (models_list_ != nullptr) {
            auto& app = Application::GetInstance();
            app.GetAudioService().SetModelsList(models_list_);
        }
    }
}
```

`srmodel_load(ptr)` 是 ESP-SR 库的 API——直接拿 mmap 指针解析模型 metadata。模型实际权重数据**保留在 flash 上不拷贝**，运行时按需读。

`SetModelsList` 见第 4 章 4.6 —— audio_service 根据模型列表选 AfeWakeWord / EspWakeWord / CustomWakeWord。

7.5.4.2 加载字体（LVGL 路径）

```
#ifdef HAVE_LVGL
cJSON* font = cJSON_GetObjectItem(root, "text_font");
if (cJSON_IsString(font)) {
    std::string fonts_text_file = font->valuestring;
    if (GetAssetData(fonts_text_file, ptr, size)) {
        auto text_font = std::make_shared<LvglCBinFont>(ptr);
        if (text_font->font() == nullptr) return false;
        if (light_theme != nullptr) light_theme->set_text_font(text_font);
        if (dark_theme != nullptr) dark_theme->set_text_font(text_font);
    }
}
```

`LvglCBinFont(ptr)` 用 mmap 指针构造 LVGL 字体对象（CBIN 是 LVGL 自定义的紧凑二进制字体格式）。`shared_ptr` 让 light + dark 主题共享同一份字体。

7.5.4.3 加载 emoji_collection

```
cJSON* emoji_collection = cJSON_GetObjectItem(root, "emoji_collection");
if (cJSON_IsArray(emoji_collection)) {
    auto custom_emoji_collection = std::make_shared<EmojiCollection>();
    int emoji_count = cJSON_GetArraySize(emoji_collection);
    for (int i = 0; i < emoji_count; i++) {
        cJSON* emoji = cJSON_GetArrayItem(emoji_collection, i);
        if (cJSON_IsObject(emoji)) {
            cJSON* name = cJSON_GetObjectItem(emoji, "name");
            cJSON* file = cJSON_GetObjectItem(emoji, "file");
            cJSON* eaf = cJSON_GetObjectItem(emoji, "eaf");
            if (cJSON_IsString(name) && cJSON_IsString(file) && (NULL == eaf)) {
                if (!GetAssetData(file->valuestring, ptr, size)) continue;
                custom_emoji_collection->AddEmoji(name->valuestring, new LvglRawImage(ptr,
        size));
            }
        }
    }
    if (light_theme != nullptr) light_theme->set_emoji_collection(custom_emoji_collection);
    if (dark_theme != nullptr) dark_theme->set_emoji_collection(custom_emoji_collection);
}
```

把每个表情图直接构造成 `LvglRawImage(ptr, size)` 并以 name 注册到 EmojiCollection。

注意 `(NULL == eaf)` —— **EAF 格式（emote-animation-frame）走 elif 分支的另一条路径**（emote_display），不是 LVGL 路径。

7.5.4.4 emote 路径（OLED / 像素屏特殊设备）

某些板子用 emote_display 而非 LVGL（例如带极小 OLED 又要播放表情动画的板子）：

```
#elif defined(CONFIG_USE_EMOTE_MESSAGE_STYLE)
auto emote_display = dynamic_cast<emote::EmoteDisplay*>(display);

cJSON* emoji_collection = cJSON_GetObjectItem(root, "emoji_collection");
if (cJSON_IsArray(emoji_collection)) {
    // ... 加载带 fps/loop/lack 参数的动画
    emote_display->AddEmojiData(name->valuestring, ptr, size,
                                static_cast<uint8_t>(fps_value),
                                loop_value, lack_value);
}

cJSON* layout_json = cJSON_GetObjectItem(root, "layout");
// ... emote_display->AddLayoutData(name, align, x, y, width, height);
```

`fps` / `loop` / `lack` 字段控制 GIF-like 动画的播放参数；`layout` 数组定义 UI 元素的绝对定位。

7.5.5 `Download()` —— 下载并写入 assets 分区

```
bool Assets::Download(std::string url, std::function<void(int, size_t)> progress_callback) {
    if (mmap_handle_ != 0) {
        esp_partition_munmap(mmap_handle_);
        mmap_handle_ = 0;
        mmap_root_ = nullptr;
    }
    checksum_valid_ = false;
    assets_.clear();
```

先 unmap——要往这个分区写入了，不能再让 mmap 句柄持有它。

```
    auto http = network->CreateHttp(0);
    if (!http->Open("GET", url)) return false;
    if (http->GetStatusCode() != 200) return false;
    size_t content_length = http->GetBodyLength();
    if (content_length > partition_->size) return false;

    const size_t SECTOR_SIZE = esp_partition_get_main_flash_sector_size();
    size_t sectors_to_erase = (content_length + SECTOR_SIZE - 1) / SECTOR_SIZE;
```

ESP32 flash 最小擦除单位是扇区（4KB），先算需要擦多少扇区。

```
    char buffer[512];
    size_t total_written = 0;
    size_t current_sector = 0;
    while (true) {
        int ret = http->Read(buffer, sizeof(buffer));
        if (ret <= 0) break;

        // 边下载边擦除：刚好够覆盖当前要写的位置就擦
        size_t write_end_offset = total_written + ret;
        size_t needed_sectors = (write_end_offset + SECTOR_SIZE - 1) / SECTOR_SIZE;
        while (current_sector < needed_sectors) {
            size_t sector_start = current_sector * SECTOR_SIZE;
            esp_partition_erase_range(partition_, sector_start, SECTOR_SIZE);
            current_sector++;
        }

        esp_partition_write(partition_, total_written, buffer, ret);
        total_written += ret;
        // ... 进度回调
    }

    if (!InitializePartition()) return false;
    return true;
}
```

渐进式擦写的好处：

- 不需要一次擦完整个分区（避免几秒钟阻塞）；
- 擦除和下载交错进行 → 总耗时 = max(擦除时间, 下载时间) 而不是两者之和；
- 同时不会触发分区"擦完一半但写入失败 → 资源完全丢失"的中间状态（虽然中断仍会损坏，但概率小）。

下载完后立即重新 `InitializePartition` 校验 checksum，失败说明文件损坏（或下载到一半网络断了）。

7.5.6 析构 —— 释放 mmap

```cpp
Assets::~Assets() {
    if (mmap_handle_ != 0) {
        esp_partition_munmap(mmap_handle_);
    }
}
```

单例模式下其实析构很少触发（程序结束才会），但写出来是良好习惯。

7.6 跨模块时序：从启动到正常工作的完整生命周期

```
启动
  │
  ├─ app_main()                                [main.cc]
  │      └─ Application::Run()
  │
  ├─ Application::Initialize()                 [application.cc]
  │      ├─ Ota::MarkCurrentVersionValid()     ← OTA 防回滚标记
  │      ├─ Assets::InitializePartition()      ← mmap + checksum
  │      ├─ AudioService::Initialize()
  │      ├─ McpServer::AddCommonTools()        ← 注册 LLM 可见工具
  │      └─ board.StartNetwork()               ← Wi-Fi 异步连接
  │
  ├─ [NETWORK_CONNECTED 事件触发]
  │      │
  │      └─ ActivationTask
  │            │
  │            ├─ CheckAssetsVersion()         [application.cc]
  │            │    └─ HTTP 检查资源版本
  │            │    └─ Assets::Download()      ← 如有新版下载
  │            │    └─ Assets::Apply()         ← 加载字体/模型/皮肤
  │            │
  │            ├─ CheckNewVersion()            [application.cc]
  │            │    └─ Ota::CheckVersion()     ← 拿固件URL/mqtt配置/时间
  │            │    └─ 若 has_new_version_:    Ota::Upgrade() + esp_restart
  │            │    └─ 若 has_activation_*_:   Ota::Activate() / ShowActivationCode
  │            │    └─ 否则继续
  │            │
  │            └─ InitializeProtocol()         [application.cc]
  │                 └─ 根据 has_mqtt_config_ / has_websocket_config_
  │                    实例化 MqttProtocol 或 WebsocketProtocol
  │
  ├─ Application::Run() 主循环   ← 至此进入正常工作状态
  │      ├─ 处理 MAIN_EVENT_TOGGLE_CHAT
  │      ├─ 处理 MAIN_EVENT_SEND_AUDIO
  │      ├─ 处理 MAIN_EVENT_STATE_CHANGED
  │      └─ ...
```

7.7 安全设计要点小结

1. **eFuse 序列号 + HMAC**：硬件级身份证明，软件层无法伪造。
2. **OTA A/B 双分区**：升级失败可回滚，永不变砖。
3. **PENDING_VERIFY 状态**：新固件没跑起来的话自动回滚。

4. **固件签名**：`esp_ota_end` 内部做整体 SHA-256 校验。
5. **NVS 隔离 namespace**：不同模块的配置互不污染。
6. **assets checksum**：防误读，下载完立即校验。
7. **assets magic ZZ**：每个资源文件加 magic 防错位。
8. **NVS 写入比对**：避免无谓的 flash 写入磨损。

7.8 本章用到的关键技术

技术	应用
NVS（Non-Volatile Storage）	settings 全部
RAII（构造打开/析构关闭）	Settings 类
`dirty_` 脏标志	只在改过时 commit
eFuse / esp_efuse_read_field_blob	读出厂烧的序列号
SOC HMAC 加速器 / esp_hmac_calculate	设备激活的身份验证
`esp_partition_mmap`	把 flash 分区映射到 CPU 地址空间，零拷贝读
`esp_ota_*` API 全家桶	begin / write / end / set_boot_partition / mark_app_valid
A/B 分区 OTA	旧固件不变，新固件写另一槽
流式 HTTP 下载	512 字节块 + 进度回调，不一次性占大 RAM
`esp_partition_erase_range` + `esp_partition_write`	擦写交错节省时间
`std::string().swap()` idiom	强制释放 string 容量
`std::stringstream + getline`	语义化版本字符串解析
`cJSON_ArrayForEach` + 对象字段	遍历 mqtt/websocket object
`settimeofday`	HTTP 同步时间替代 NTP
Meyers 单例（Assets / Ota）	全局唯一实例
二进制文件 magic ‘ZZ’	资源文件结构错位检测
简单累加校验和	资源分区数据完整性
`shared_ptr` + LVGL 资源	多主题共享字体/图片

7.9 看完本章你应该掌握的

- NVS 是什么、namespace 怎么用
- Settings 的 RAII + dirty 设计
- SystemInfo 全部静态方法 + ESP32-P4 的 Wi-Fi 特殊化
- OTA 的完整流程：CheckVersion → 拿固件 URL/MQTT 配置/时间 → Activate → Upgrade
- Activation 两套协议（code 模式 vs HMAC challenge 模式）
- ESP32 HMAC 硬件加速器的工作原理（eFuse key 不可读 + 硬件签名）
- A/B 双分区 OTA + PENDING_VERIFY 防变砖机制
- Assets 二进制分区格式（header + 索引表 + 数据区）

- `esp_partition_mmap` 零拷贝访问 flash
- Assets 渐进式擦写下载
- index.json 解析 + 模型/字体/表情/皮肤的多路径加载
- LVGL 路径 vs emote 路径的差异（一个用统一主题、一个用 AddXxxData）

下一章覆盖 `display/` + `led/` 显示与指示子系统。

第 8 章 显示与指示：`main/display/` 与 `main/led/`

> 总计 3,426 行（display 2,613 行 + led 813 行）。这两个目录都用同一套思路：抽象基类定义 `OnStateChanged()` / `SetXxx()` 接口，再用多种子类对接不同的硬件。本章把所有派生类摊开讲。

8.1 整体结构

```
main/display/
├── display.{h,cc}            ★ Display 抽象基类（无操作的 NoDisplay 也在里面）
│                              定义 SetStatus/SetEmotion/SetChatMessage/SetTheme...
│
├── lcd_display.{h,cc}        ★ LCD/IPS 屏（SPI/RGB/MIPI 三种总线，LVGL）
│                              SpiLcdDisplay / RgbLcdDisplay / MipiLcdDisplay
│
├── oled_display.{h,cc}       ★ 黑白单色屏（128×32 / 128×64，LVGL monochrome）
│
├── emote_display.{h,cc}      ★ 表情动画专用屏（不走 LVGL，自己渲染管线）
│
└── lvgl_display/             [子目录]
    LvglDisplay 通用 LVGL 基类、字体、主题、GIF 控制器等

main/led/
├── led.h                     ★ Led 抽象（OnStateChanged 一个纯虚函数 + NoLed）
├── single_led.{h,cc}         ★ 单粒 WS2812B（最常见，支持 RGB + 闪烁/常亮）
├── circular_strip.{h,cc}     ★ WS2812B 环形灯带（多粒，支持 Scroll/Breathe/FadeOut）
└── gpio_led.{h,cc}           ★ 普通 GPIO + PWM 单色 LED（LEDC 外设 + 硬件淡入淡出）
```

8.2 Display 抽象基类

8.2.1 接口

```
class Display {
public:
    virtual void SetStatus(const char* status);
    virtual void ShowNotification(const char* notification, int duration_ms = 3000);
    virtual void ShowNotification(const std::string &notification, int duration_ms = 3000);
    virtual void SetEmotion(const char* emotion);
    virtual void SetChatMessage(const char* role, const char* content);
    virtual void SetTheme(Theme* theme);
    virtual Theme* GetTheme() { return current_theme_; }
    virtual void UpdateStatusBar(bool update_all = false);
    virtual void SetPowerSaveMode(bool on);

    inline int width() const { return width_; }
    inline int height() const { return height_; }

protected:
    int width_ = 0;
    int height_ = 0;
    Theme* current_theme_ = nullptr;

    friend class DisplayLockGuard;
    virtual bool Lock(int timeout_ms = 0) = 0;    // 纯虚
    virtual void Unlock() = 0;                     // 纯虚
};
```

要点：

- 全部业务接口（SetStatus / SetEmotion / SetChatMessage / ShowNotification / SetTheme / UpdateStatusBar / SetPowerSaveMode）都有**默认空实现**——基类做日志就行，子类按需重写。
- **`Lock` / `Unlock` 是纯虚**——LVGL/EmoteEngine 都各自有自己的锁机制，子类必须给。
- `friend class DisplayLockGuard` 让 `DisplayLockGuard` 能调 private Lock/Unlock。

8.2.2 `DisplayLockGuard` —— RAII 锁

```
class DisplayLockGuard {
public:
    DisplayLockGuard(Display *display) : display_(display) {
        if (!display_->Lock(30000)) {
            ESP_LOGE("Display", "Failed to lock display");
        }
    }
    ~DisplayLockGuard() {
        display_->Unlock();
    }
private:
    Display *display_;
};
```

LVGL 不是线程安全的——任何非 LVGL 任务调 lv_xx API 前必须先 lock。RAII 模式让锁在离开作用域时自动释放，避免漏 unlock 死锁。

典型用法（在 OledDisplay::SetChatMessage 里）：

```
void OledDisplay::SetChatMessage(const char* role, const char* content) {
    DisplayLockGuard lock(this);   // ← 此处加锁
    if (chat_message_label_ == nullptr) return;
    // ... 调 lv_label_set_text 等
    // 函数返回时锁自动释放
}
```

8.2.3 `NoDisplay` —— 哑实现

```
class NoDisplay : public Display {
private:
    virtual bool Lock(int timeout_ms = 0) override { return true; }
    virtual void Unlock() override {}
};
```

板子没屏时用这个，所有接口都是基类的空日志实现。这是**Null Object 模式**——避免在调用方到处写 `if (display)` 判空。

8.2.4 `SetTheme` 持久化

```
void Display::SetTheme(Theme* theme) {
    current_theme_ = theme;
    Settings settings("display", true);
    settings.SetString("theme", theme->name());
}
```

主题切换写到 NVS `display` namespace，下次启动自动恢复。

8.3 `OledDisplay` —— 单色 OLED 屏

396 行。最常见的低成本 0.96” 128×64 OLED 屏走这条路。

8.3.1 构造 —— LVGL Port 初始化

```
OledDisplay::OledDisplay(esp_lcd_panel_io_handle_t panel_io, esp_lcd_panel_handle_t panel,
    int width, int height, bool mirror_x, bool mirror_y)
    : panel_io_(panel_io), panel_(panel) {
    width_ = width;
    height_ = height;

    auto text_font = std::make_shared<LvglBuiltInFont>(&BUILTIN_TEXT_FONT);
    auto icon_font = std::make_shared<LvglBuiltInFont>(&BUILTIN_ICON_FONT);
    auto large_icon_font = std::make_shared<LvglBuiltInFont>(&font_awesome_30_1);

    auto dark_theme = new LvglTheme("dark");
    dark_theme->set_text_font(text_font);
    dark_theme->set_icon_font(icon_font);
    dark_theme->set_large_icon_font(large_icon_font);

    auto& theme_manager = LvglThemeManager::GetInstance();
    theme_manager.RegisterTheme("dark", dark_theme);
    current_theme_ = dark_theme;
```

要点：

- **三种字体**：text（正文小字）/ icon（状态栏小图标）/ large_icon（中间大表情）；
- `font_awesome_30_1` 是 Font Awesome 字体的 30px 子集（项目自带源码 `font_awesome.h`）；
- OLED 只支持 dark 主题（单色屏没法做 light）；
- 注册到全局 `LvglThemeManager`（让 assets / mcp 的 `self.screen.set_theme` 能查到）。

```
    lvgl_port_cfg_t port_cfg = ESP_LVGL_PORT_INIT_CONFIG();
    port_cfg.task_priority = 1;
    port_cfg.task_stack = 6144;
#if CONFIG_SOC_CPU_CORES_NUM > 1
    port_cfg.task_affinity = 1;   // 双核时绑核 1
#endif
    lvgl_port_init(&port_cfg);
```

`lvgl_port` 是 espressif/esp_lvgl_port 组件——把 LVGL 跑在一个独立 FreeRTOS 任务里：

- `task_priority = 1` 低优先级，让网络/音频先跑；
- `task_stack = 6144` 6KB 栈，LVGL 调用比较深；
- **绑核 1**：双核芯片时把 UI 推到 core 1 跟 audio_input（也在 core 1）共享，避免和网络任务（core 0）竞争。注意 audio_input 优先级远高于 UI 所以 UI 不阻塞 audio。

```
    const lvgl_port_display_cfg_t display_cfg = {
        .io_handle = panel_io_,
        .panel_handle = panel_,
        .buffer_size = static_cast<uint32_t>(width_ * height_),
        .double_buffer = false,
        .hres = static_cast<uint32_t>(width_),
        .vres = static_cast<uint32_t>(height_),
        .monochrome = true,
        .rotation = { .mirror_x = mirror_x, .mirror_y = mirror_y },
        .flags = { .buff_dma = 1, .buff_spiram = 0 }
    };
    display_ = lvgl_port_add_disp(&display_cfg);
```

- `buffer_size = width × height`：单色屏每像素 1 bit，但 LVGL buffer 还是 8bpp（128×64=8192 = 8KB）—— OLED 这点小不心疼；
- `double_buffer = false`：单色屏刷新慢但全屏小，不开双 buffer；
- `monochrome = true`：告诉 LVGL 不要走 RGB565 路径；
- `buff_dma = 1`：DMA 传输（CPU 不阻塞）；
- `buff_spiram = 0`：buffer 放 SRAM（PSRAM 对 DMA 不友好）。

```
    if (height_ == 64) {
        SetupUI_128x64();
    } else {
        SetupUI_128x32();
    }
```

64 行高 vs 32 行高两套布局，差别在于 status_bar 位置。

8.3.2 `SetupUI_128x64()` 布局解构

```
container_ (128×64, 纵向 flex)
├── top_bar_ (128×16, 横向 flex, space_between)
│   ├── network_label_  ← Wi-Fi/4G 信号图标
│   └── right_icons (横向 flex)
│       ├── mute_label_   ← 静音图标
│       └── battery_label_ ← 电池图标
│
├── status_bar_ (覆盖在 top_bar 上方, 显示状态文字)
│   ├── notification_label_ (默认 hidden)
│   └── status_label_ (LV_LABEL_LONG_SCROLL_CIRCULAR)
│
└── content_ (128×48, 横向 flex, center)
    ├── content_left_ (32×content)
    │   └── emotion_label_ (font_awesome_30_1 字体, 显示大图标)
    │
    └── content_right_ (content×content, 默认 hidden)
        └── chat_message_label_ (滚动字幕)
```

每个 LVGL 对象都要逐一设置 padding/border/flex/scrollbar：

```
lv_obj_set_style_pad_all(container_, 0, 0);
lv_obj_set_style_border_width(container_, 0, 0);
lv_obj_set_style_pad_row(container_, 0, 0);
```

LVGL 风格 API 第二参数是 selector（`LV_PART_MAIN | LV_STATE_DEFAULT` 等组合，0 = main+default）。

滚动字幕的字体动画：

```
static lv_anim_t a;
lv_anim_init(&a);
lv_anim_set_delay(&a, 1000);
lv_anim_set_repeat_count(&a, LV_ANIM_REPEAT_INFINITE);
lv_obj_set_style_anim(chat_message_label_, &a, LV_PART_MAIN);
lv_obj_set_style_anim_duration(chat_message_label_,
    lv_anim_speed_clamped(60, 300, 60000), LV_PART_MAIN);
```

`lv_anim_speed_clamped(60, 300, 60000)` = “每秒滚 60 像素，最少 300ms 一轮，最多 60 秒一轮”。短文字停顿明显，长文字滚得过去。

8.3.3 `SetEmotion()` —— 字符转 emoji 字体

```
void OledDisplay::SetEmotion(const char* emotion) {
    const char* utf8 = font_awesome_get_utf8(emotion);
    DisplayLockGuard lock(this);
    if (emotion_label_ == nullptr) return;
    if (utf8 != nullptr) {
        lv_label_set_text(emotion_label_, utf8);
    } else {
        lv_label_set_text(emotion_label_, FONT_AWESOME_NEUTRAL);
    }
}
```

`font_awesome_get_utf8` 把 “happy”/“sad”/“thinking” 这类英文 emotion key 翻译为 Font Awesome 字体里对应的 UTF-8 字符串（如 `\xf118` 微笑脸）。LVGL label 把这个字符当成普通字符渲染——只是字体里这个码位实际画的是表情图案。

未知 emotion 时 fallback 到 NEUTRAL（中性脸），不会黑屏。

8.3.4 `SetChatMessage()` —— 滚动字幕

```
void OledDisplay::SetChatMessage(const char* role, const char* content) {
    DisplayLockGuard lock(this);
    if (chat_message_label_ == nullptr) return;

    std::string content_str = content;
    std::replace(content_str.begin(), content_str.end(), '\n', ' ');

    if (content_right_ == nullptr) {
        lv_label_set_text(chat_message_label_, content_str.c_str());
    } else {
        if (content == nullptr || content[0] == '\0') {
            lv_obj_add_flag(content_right_, LV_OBJ_FLAG_HIDDEN);
        } else {
            lv_label_set_text(chat_message_label_, content_str.c_str());
            lv_obj_remove_flag(content_right_, LV_OBJ_FLAG_HIDDEN);
        }
    }
}
```

- 把所有 `\n` 替换为空格——LVGL 滚动 label 在单行模式下不处理换行；
- 空字符串 → hide content_right_（emoji 区域占满屏幕）；
- 有内容 → show content_right_（emoji 缩到 32px 左侧，字幕显示右侧）。

8.4 `LcdDisplay` —— 彩色 LCD 屏

1196 行（实现），最复杂。

8.4.1 三种子类

```
class LcdDisplay : public LvglDisplay { ... };

class SpiLcdDisplay : public LcdDisplay {
    SpiLcdDisplay(io, panel, w, h, ox, oy, mx, my, sx);  // SPI 总线
};
class RgbLcdDisplay : public LcdDisplay {
    RgbLcdDisplay(io, panel, w, h, ...);                  // RGB 并行总线
};
class MipiLcdDisplay : public LcdDisplay {
    MipiLcdDisplay(io, panel, w, h, ...);                 // MIPI-DSI (P4)
};
```

接口完全一致，**子类只在构造函数里做不同的 LVGL flush/draw_buf 配置**——SPI 屏小，PSRAM 双缓冲；RGB 屏大（800×480），需要部分刷新；MIPI 屏走 DSI 总线。

8.4.2 关键成员

```
class LcdDisplay : public LvglDisplay {
protected:
    esp_lcd_panel_io_handle_t panel_io_;
    esp_lcd_panel_handle_t panel_;

    lv_draw_buf_t draw_buf_;
    lv_obj_t* top_bar_;        // 顶部状态栏
    lv_obj_t* status_bar_;
    lv_obj_t* content_;
    lv_obj_t* container_;
    lv_obj_t* side_bar_;
    lv_obj_t* bottom_bar_;
    lv_obj_t* preview_image_;   // self.screen.preview_image MCP 工具的预览
    lv_obj_t* emoji_label_;     // 当用字体表情时
    lv_obj_t* emoji_image_;     // 当用 GIF/位图表情时
    std::unique_ptr<LvglGif> gif_controller_;   // GIF 解码器
    lv_obj_t* emoji_box_;       // emoji 容器
    lv_obj_t* chat_message_label_;
    esp_timer_handle_t preview_timer_;
    std::unique_ptr<LvglImage> preview_image_cached_;
    bool hide_subtitle_ = false;
};
```

比 OLED 多了：

- `emoji_image_` + `gif_controller_`：彩屏可以播放动画 GIF 表情；
- `preview_image_*`：MCP 工具 `self.screen.preview_image` 用，显示远程图 5 秒后自动隐藏；
- `hide_subtitle_`：从 assets index.json 读取，某些场景关掉字幕只看表情。

8.4.3 `SetEmotion` 双路径

伪代码（实际 lcd_display.cc 第 600 行附近）：

```
void LcdDisplay::SetEmotion(const char* emotion) {
    DisplayLockGuard lock(this);
    auto* lvgl_theme = static_cast<LvglTheme*>(current_theme_);
    auto emoji_collection = lvgl_theme->emoji_collection();

    // 路径 1: 主题里注册了自定义表情图 (assets/emoji_collection) → 显示位图
    if (emoji_collection != nullptr) {
        auto image = emoji_collection->GetEmoji(emotion);
        if (image != nullptr) {
            lv_obj_remove_flag(emoji_image_, LV_OBJ_FLAG_HIDDEN);
            lv_obj_add_flag(emoji_label_, LV_OBJ_FLAG_HIDDEN);
            if (image->is_gif()) {
                gif_controller_->SetSource(image);   // 播放 GIF
            } else {
                lv_image_set_src(emoji_image_, image->src());
            }
            return;
        }
    }

    // 路径 2: 回退到 Font Awesome 字符
    lv_obj_add_flag(emoji_image_, LV_OBJ_FLAG_HIDDEN);
    lv_obj_remove_flag(emoji_label_, LV_OBJ_FLAG_HIDDEN);
    lv_label_set_text(emoji_label_, font_awesome_get_utf8(emotion));
}
```

设计思路：

- **优先用资源包里的高分辨率位图**（彩色 / 动画）；
- **没有则降级到字体**（一种字符 = 一种表情，字体单色但通用）；
- 路径切换通过 hidden flag 控制 emoji_image_ 和 emoji_label_ 互斥显示。

8.4.4 GIF 表情控制 —— `LvglGif`

```
std::unique_ptr<LvglGif> gif_controller_;
```

LcdDisplay 持有 GIF 解码器。GIF 是分帧的，需要按 fps 定期切下一帧——`LvglGif::SetSource(image)` 内部启 esp_timer 周期刷帧。

注意是 `unique_ptr`——切换到字体表情时直接 `gif_controller_.reset()` 释放解码器和内存。

8.4.5 `SetPreviewImage()` —— 临时图像预览

```
void LcdDisplay::SetPreviewImage(std::unique_ptr<LvglImage> image) {
    DisplayLockGuard lock(this);
    if (preview_timer_) esp_timer_stop(preview_timer_);
    preview_image_cached_ = std::move(image);
    lv_image_set_src(preview_image_, preview_image_cached_->src());
    lv_obj_remove_flag(preview_image_, LV_OBJ_FLAG_HIDDEN);
    esp_timer_start_once(preview_timer_, PREVIEW_IMAGE_DURATION_MS * 1000);
}
```

5 秒后 timer 触发隐藏 preview_image_ 并 reset 缓存——避免内存常驻。这是 `self.screen.preview_image` MCP 工具的实现核心（第 6 章 6.7.3）。

8.4.6 三种总线的特殊化

SPI（最常见，小屏 ≤320×240）：

```
SpiLcdDisplay::SpiLcdDisplay(io, panel, w, h, ox, oy, mx, my, sx) {
    // 用 PSRAM 分配 draw_buf (单 buffer, 全屏字节 = w*h*2 字节 RGB565)
    // 全屏 buffer：刷新慢但帧间无撕裂
    // partial mode: 只重画 dirty 区域
}
```

RGB（大屏 ≥480×320，并行总线带宽够）：

```
RgbLcdDisplay::RgbLcdDisplay(...) {
    // 用专用 RGB framebuffer
    // direct_mode：LVGL 直接写 framebuffer，无需 flush
    // 双 buffer：撕裂稍可见但帧率高
}
```

MIPI（仅 ESP32-P4，HD 屏 800×1280）：

```
MipiLcdDisplay::MipiLcdDisplay(...) {
    // 类似 RGB 但走 MIPI-DSI 总线（差分串行，速度更快）
    // 通常配 PPA 硬件加速
}
```

具体的总线差异在每种屏的 init 函数里——读者用到时再深入。

8.5 `EmoteDisplay` —— 表情动画专用屏

657 行。这是个**绕开 LVGL** 的特殊路径，用于追求"主屏只显示一个大表情 + 极少字幕"场景，对刷新率和动画流畅度极致优化。

8.5.1 为什么不用 LVGL

- LVGL 渲染管线在 ESP32-C3 / S3 这种 SRAM 紧张的芯片上跑 480×480 全屏动画很吃力（每帧要做 alpha 合成 / 抗锯齿）；
- 表情动画需求简单：固定 fps + 全屏直贴；
- 走自定义 `EmoteEngine`（项目里以 .a 静态库形式提供）能用硬件 PPA 直接 DMA 推到屏。

8.5.2 `AssetData` —— 位字段压缩

```
struct AssetData {
    const void* data;
    size_t size;
    union {
        uint8_t flags;
        struct {
            uint8_t fps : 6;    // 0-63
            uint8_t loop : 1;
            uint8_t lack : 1;
        };
    };
};
```

union + 位字段：

- 一个字节 = 8 位，分成 6 + 1 + 1 三个字段；
- fps 6 位最大 63，对 60fps 也够；
- `loop`：是否循环播放；
- `lack`：是否有”缺帧补帧”逻辑；
- 通过 union 既能整字节读 `flags`（拷贝/重置），也能位字段语义访问。

构造时主动 clamp：

```
AssetData(const void* d, size_t s, uint8_t f, bool l, bool k) : data(d), size(s) {
    fps = f > 63 ? 63 : f;  // 防溢出
    loop = l;
    lack = k;
}
```

8.5.3 `LayoutData` —— 元素定位

```
struct LayoutData {
    char align;   // 单字符 = 一种对齐方式
    int x, y, width, height;
    bool has_size;
};

char StringToGfxAlign(const std::string &align_str);
```

`align_str` 是 assets 里写的“top_left” / “center” / “bottom_right”等字符串，运行时转成 EmoteEngine 内部用的单字符代码（节省 50% 内存）。

8.5.4 接口同 Display 但内部走 EmoteEngine

```
class EmoteDisplay : public Display {
public:
    virtual void SetEmotion(const char* emotion) override;
    virtual void SetStatus(const char* status) override;
    virtual void SetChatMessage(const char* role, const char* content) override;
    virtual void SetTheme(Theme* theme) override;
    virtual void ShowNotification(const char* notification, int duration_ms = 3000) override;
    virtual void UpdateStatusBar(bool update_all = false) override;
    virtual void SetPowerSaveMode(bool on) override;
    virtual void SetPreviewImage(const void* image);

    void AddEmojiData(const std::string &name, const void* data, size_t size, uint8_t fps = 0,
        bool loop = false, bool lack = false);
    void AddIconData(const std::string &name, const void* data, size_t size);
    void AddLayoutData(...);
    void AddTextFont(std::shared_ptr<LvglFont> text_font);

private:
    std::unique_ptr<EmoteEngine> engine_;
    std::shared_ptr<LvglFont> text_font_;
    std::map<std::string, AssetData> emoji_data_map_;
    std::map<std::string, AssetData> icon_data_map_;
};
```

`assets.cc` 在 emote 路径下调用 `AddEmojiData` / `AddIconData` / `AddLayoutData` 把资源送进 EmoteDisplay 的 map。SetEmotion 时根据 emotion key 从 map 查表，再交给 `engine_` 推到屏。

`EmoteEngine` 本身是闭源的（`libs/` 下的 .a），内部包含：

- EAF 格式（Emote Animation Frame）解码器；
- 软硬件混合的帧合成器；
- DMA 推帧到屏。

虽然源码不可见，但接口在 emote_display.h 里完整，从外部就能正确驱动。

8.6 LED 抽象与三种实现

8.6.1 `Led` 接口

```
class Led {
public:
    virtual ~Led() = default;
    virtual void OnStateChanged() = 0;
};

class NoLed : public Led {
public:
    virtual void OnStateChanged() override {}
};
```

接口极简——**只有一个 `OnStateChanged()` 纯虚函数**。

由 Application::HandleStateChangedEvent 在所有设备状态变化时调用：

```
void Application::HandleStateChangedEvent() {
    auto led = board_->GetLed();
    led->OnStateChanged();
    // ...
}
```

LED 子类自己内部查 Application 的当前状态做对应的动作。

8.6.2 `SingleLed` —— 单粒 WS2812B 全彩

163 行。最常见的板子上唯一一颗状态灯（WS2812B 是带控制器的智能 LED，能输出 24bit 任意颜色）。

初始化：RMT 外设驱动 WS2812

```
SingleLed::SingleLed(gpio_num_t gpio) {
    led_strip_config_t strip_config = {};
    strip_config.strip_gpio_num = gpio;
    strip_config.max_leds = 1;
    strip_config.color_component_format = LED_STRIP_COLOR_COMPONENT_FMT_GRB;
    strip_config.led_model = LED_MODEL_WS2812;

    led_strip_rmt_config_t rmt_config = {};
    rmt_config.resolution_hz = 10 * 1000 * 1000;  // 10MHz

    ESP_ERROR_CHECK(led_strip_new_rmt_device(&strip_config, &rmt_config, &led_strip_));
    led_strip_clear(led_strip_);
```

RMT 是 ESP32 的红外遥控外设，但常被借用来生成 WS2812B 需要的特定时序（高电平 0.4μs/0.8μs 等编码 0/1）。`led_strip` 组件封装了这套逻辑。

```
    esp_timer_create_args_t blink_timer_args = {
        .callback = [](void *arg) {
            auto led = static_cast<SingleLed*>(arg);
            led->OnBlinkTimer();
        },
        .arg = this,
        .dispatch_method = ESP_TIMER_TASK,
        .name = "blink_timer",
    };
    esp_timer_create(&blink_timer_args, &blink_timer_);
}
```

闪烁靠 esp_timer 周期触发 `OnBlinkTimer`，`ESP_TIMER_TASK` 在专用 timer 任务里调度（不在 ISR 里跑），可以调任意 API。

`OnBlinkTimer()` 实现闪烁

```
void SingleLed::OnBlinkTimer() {
    std::lock_guard<std::mutex> lock(mutex_);
    blink_counter_--;
    if (blink_counter_ & 1) {                // 奇数 → 亮
        led_strip_set_pixel(led_strip_, 0, r_, g_, b_);
        led_strip_refresh(led_strip_);
    } else {                                 // 偶数 → 灭
        led_strip_clear(led_strip_);
        if (blink_counter_ == 0) {
            esp_timer_stop(blink_timer_);
        }
    }
}
```

巧用计数器奇偶：

- 初始 `blink_counter_ = times * 2`（每次闪烁 = 亮一次 + 灭一次）；
- 每次 timer tick 减 1；
- 奇数亮、偶数灭；
- 减到 0 自动停。

无限闪烁时 `times = BLINK_INFINITE = -1`，乘 2 = 0xFFFE，几乎用不完。

`OnStateChanged()` —— 状态到颜色的映射表

```
void SingleLed::OnStateChanged() {
    auto& app = Application::GetInstance();
    auto device_state = app.GetDeviceState();
    switch (device_state) {
        case kDeviceStateStarting:
            SetColor(0, 0, DEFAULT_BRIGHTNESS);      // 蓝
            StartContinuousBlink(100);                // 快闪
            break;
        case kDeviceStateWifiConfiguring:
            SetColor(0, 0, DEFAULT_BRIGHTNESS);      // 蓝
            StartContinuousBlink(500);                // 慢闪
            break;
        case kDeviceStateIdle:
            TurnOff();                                // 关
            break;
        case kDeviceStateConnecting:
            SetColor(0, 0, DEFAULT_BRIGHTNESS);      // 蓝
            TurnOn();                                 // 常亮
            break;
        case kDeviceStateListening:
        case kDeviceStateAudioTesting:
            if (app.IsVoiceDetected()) {
                SetColor(HIGH_BRIGHTNESS, 0, 0);     // 强红（说话中）
            } else {
                SetColor(LOW_BRIGHTNESS, 0, 0);      // 弱红（静音中）
            }
            TurnOn();
            break;
        case kDeviceStateSpeaking:
            SetColor(0, DEFAULT_BRIGHTNESS, 0);      // 绿
            TurnOn();
            break;
        case kDeviceStateUpgrading:
            SetColor(0, DEFAULT_BRIGHTNESS, 0);      // 绿
            StartContinuousBlink(100);                // 快闪
            break;
        case kDeviceStateActivating:
            SetColor(0, DEFAULT_BRIGHTNESS, 0);      // 绿
            StartContinuousBlink(500);                // 慢闪
            break;
    }
}
```

状态→颜色映射的设计准则：

- **冷色（蓝）= 系统/网络忙**（启动/配网/连接）；
- **暖色（红）= 在听用户**（高亮度=正在说话，低亮度=静音）；
- **绿 = 在说话给用户 / 系统就绪状态**（speaking/upgrade/activate）；
- **快闪 (100ms) = 关键事件**（启动、升级）；
- **慢闪 (500ms) = 等待状态**（配网、激活）；
- **常亮 = 进行中**（已连接、说话中）；
- **熄灭 = 待机**（idle）。

注意 Listening 状态里**还会根据 VAD（语音检测）实时切换亮度**——能直观看到设备"听到我说话了"。

8.6.3 `CircularStrip` —— 环形灯带（多粒 LED）

233 行。某些表盘式板子（如 box-3）周边一圈 12 颗 WS2812B 做装饰。

数据结构

```
struct StripColor { uint8_t red, green, blue; };

class CircularStrip {
    std::vector<StripColor> colors_;     // 每颗 LED 的当前颜色
    uint8_t max_leds_;
    uint8_t default_brightness_ = ...;
    uint8_t low_brightness_ = ...;

    std::function<void()> strip_callback_;  // 每帧执行的闭包
    esp_timer_handle_t strip_timer_;
};
```

`strip_callback_` 是 std::function——可以装任意 lambda，每个动画效果就是一个不同的 lambda。

4 种动画效果

1. `Blink` —— 整体闪烁

```
void CircularStrip::Blink(StripColor color, int interval_ms) {
    for (int i = 0; i < max_leds_; i++) colors_[i] = color;
    StartStripTask(interval_ms, [this]() {
        static bool on = true;
        if (on) {
            for (int i = 0; i < max_leds_; i++)
                led_strip_set_pixel(led_strip_, i, colors_[i].red, ...);
            led_strip_refresh(led_strip_);
        } else {
            led_strip_clear(led_strip_);
        }
        on = !on;
    });
}
```

注意 `static bool on` —— **每个 lambda 拷贝各自一份 static**，因为是匿名 lambda 类。能跨 timer 调用持续。

2. `FadeOut` —— 逐渐变暗

```
void CircularStrip::FadeOut(int interval_ms) {
    StartStripTask(interval_ms, [this]() {
        bool all_off = true;
        for (int i = 0; i < max_leds_; i++) {
            colors_[i].red /= 2;       // 每次减半
            colors_[i].green /= 2;
            colors_[i].blue /= 2;
            if (colors_[i].red != 0 || colors_[i].green != 0 || colors_[i].blue != 0)
                all_off = false;
            led_strip_set_pixel(led_strip_, i, colors_[i].red, ...);
        }
        if (all_off) {
            led_strip_clear(led_strip_);
            esp_timer_stop(strip_timer_);
        } else {
            led_strip_refresh(led_strip_);
        }
    });
}
```

几何衰减（每次减半）—— 比线性减更有"渐隐"感（人眼对亮度感知接近对数）。

3. `Breathe` —— 呼吸

```
void CircularStrip::Breathe(StripColor low, StripColor high, int interval_ms) {
    StartStripTask(interval_ms, [this, low, high]() {
        static bool increase = true;
        static StripColor color = low;
        if (increase) {
            // 每个分量 +1
            color.red < high.red ? color.red++ : 0;
            // ...
            if (color reaches high) increase = false;
        } else {
            // 每个分量 -1
            if (color reaches low) increase = true;
        }
        // refresh all pixels
    });
}
```

像人呼吸一样亮度起伏。`[this, low, high]` **值捕获** low/high——避免 lambda 持有的引用悬空。

4. `Scroll` —— 旋转

```
void CircularStrip::Scroll(StripColor low, StripColor high, int length, int interval_ms) {
    // 全部底色为 low
    for (int i = 0; i < max_leds_; i++) colors_[i] = low;
    StartStripTask(interval_ms, [this, low, high, length]() {
        static int offset = 0;
        // 重置为低色
        for (int i = 0; i < max_leds_; i++) colors_[i] = low;
        // 在 offset 位置画 length 颗高色
        for (int j = 0; j < length; j++) {
            int i = (offset + j) % max_leds_;
            colors_[i] = high;
        }
        // refresh
        offset = (offset + 1) % max_leds_;   // 下一帧偏移 +1
    });
}
```

模拟”一段亮带”沿环形顺时针转。`offset % max_leds_` 让它无限循环。

`OnStateChanged()` 映射到不同动画

```
case kDeviceStateStarting:
    Scroll({0,0,0}, {low_b, low_b, def_b}, 3, 100);  // 蓝色 3 颗滚动
    break;
case kDeviceStateWifiConfiguring:
    Blink({low_b, low_b, def_b}, 500);                // 蓝闪
    break;
case kDeviceStateIdle:
    FadeOut(50);                                       // 渐隐
    break;
case kDeviceStateConnecting:
    SetAllColor({low_b, low_b, def_b});               // 全蓝
    break;
case kDeviceStateListening:
case kDeviceStateAudioTesting:
    SetAllColor({def_b, low_b, low_b});               // 全红
    break;
case kDeviceStateSpeaking:
    SetAllColor({low_b, def_b, low_b});               // 全绿
    break;
case kDeviceStateUpgrading:
    Blink({low_b, def_b, low_b}, 100);                // 绿快闪
    break;
case kDeviceStateActivating:
    Blink({low_b, def_b, low_b}, 500);                // 绿慢闪
    break;
```

比 SingleLed 多了：

- Starting → Scroll（旋转感）；
- Idle → FadeOut（不是立即灭，有”睡眠”渐变）。

8.6.4 `GpioLed` —— 普通 GPIO + PWM 单色

262 行。最朴素的”一颗带电阻的 LED 接 GPIO”，无颜色控制，但能调亮度。

LEDC 外设原理

ESP32 的 LEDC（LED Controller）是专门的 PWM 外设：

- 硬件定时器周期翻转 GPIO；
- duty cycle = on-time / period 决定亮度（0–100%）；
- 硬件自带 **fade** 功能（从 duty A 平滑变到 duty B），CPU 不用参与。

```
GpioLed::GpioLed(gpio_num_t gpio, int output_invert, ledc_timer_t timer, ledc_channel_t
        channel) {
    ledc_timer_config_t ledc_timer = {};
    ledc_timer.duty_resolution = LEDC_TIMER_13_BIT;  // 8192 档亮度
    ledc_timer.freq_hz = 4000;                       // 4 kHz PWM
    ledc_timer.speed_mode = LEDC_LS_MODE;
    ledc_timer.timer_num = timer_num;
    ledc_timer_config(&ledc_timer);

    ledc_channel_.channel = channel;
    ledc_channel_.gpio_num = gpio;
    ledc_channel_.timer_sel = timer_num;
    ledc_channel_.flags.output_invert = output_invert & 0x01;
    ledc_channel_config(&ledc_channel_);

    ledc_fade_func_install(0);
    ledc_cbs_t ledc_callbacks = { .fade_cb = FadeCallback };
    ledc_cb_register(speed_mode, channel, &ledc_callbacks, this);

    xTaskCreate(EventTask, "LedEvent", 2048, this, tskIDLE_PRIORITY + 2, &event_task_handle_);
}
```

- 13 位 duty 分辨率 = 8192 档亮度（眼睛根本分不清这么多档，做 fade 时极平滑）；
- 4 kHz PWM 频率，远高于人眼闪烁感知阈值；
- **`output_invert`**：有些板子 LED 接 VCC + GPIO（低电平亮），加这个翻转；
- 注册 `FadeCallback` 让 fade 完成后通知；
- 创建专用 event_task 处理 fade end 事件。

`StartFadeTask()` —— 硬件呼吸

```cpp
void GpioLed::StartFadeTask() {
    std::lock_guard<std::mutex> lock(mutex_);
    fade_up_ = true;
    ledc_set_fade_with_time(speed_mode, channel, LEDC_DUTY, LEDC_FADE_TIME);
    ledc_fade_start(speed_mode, channel, LEDC_FADE_NO_WAIT);
}

void GpioLed::OnFadeEnd() {
    std::lock_guard<std::mutex> lock(mutex_);
    fade_up_ = !fade_up_;
    ledc_set_fade_with_time(speed_mode, channel, fade_up_ ? LEDC_DUTY : 0, LEDC_FADE_TIME);
    ledc_fade_start(speed_mode, channel, LEDC_FADE_NO_WAIT);
}

bool IRAM_ATTR GpioLed::FadeCallback(const ledc_cb_param_t *param, void *user_arg) {
    if (param->event == LEDC_FADE_END_EVT) {
        auto led = static_cast<GpioLed*>(user_arg);
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;
        xTaskNotifyFromISR(led->event_task_handle_, 0x01, eSetValueWithOverwrite,
        &xHigherPriorityTaskWoken);
    }
    return true;
}

void GpioLed::EventTask(void* arg) {
    GpioLed* led = static_cast<GpioLed*>(arg);
    while (1) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        led->OnFadeEnd();
    }
}
```

完整的硬件呼吸流程：

1. CPU 调 `ledc_set_fade_with_time(8191, 1000ms)` + `ledc_fade_start` → CPU 撒手，硬件接管；
2. LEDC 硬件 1 秒内逐级把 duty 从 0 提到 8191；
3. fade 结束，硬件触发 ISR `FadeCallback`；
4. `FadeCallback` 在 ISR 里只做一件事：`xTaskNotifyFromISR` 通知 event_task；
5. event_task 醒来调 `OnFadeEnd`；
6. `OnFadeEnd` 翻转 `fade_up_` 设新目标值并 `fade_start`；
7. 循环往复 = 呼吸效果。

为什么不能在 ISR 里直接调 `ledc_set_fade_with_time`？因为这个函数内部用了互斥锁，ISR 不能 take 锁。**用 TaskNotify 把 ISR 工作搬到 task** 是 ESP-IDF 标准模式。

`IRAM_ATTR` 属性：把 FadeCallback 放到内置 SRAM 而非 Flash，避免 ISR 访问 flash 阻塞。

`OnStateChanged` 状态映射

跟 SingleLed 结构一样，但用 SetBrightness 代替 SetColor，listening 时调 StartFadeTask 而不是简单 TurnOn——单色 LED 没法用颜色区分状态，只能靠"明暗节奏"。

8.7 Display + LED + State 状态联动总图

```
Application::HandleStateChangedEvent (主循环上)
   ├─ board.GetLed()->OnStateChanged()            [LED 类按自己逻辑变色/闪烁]
   ├─ board.GetDisplay()->SetStatus(...)         [屏顶状态栏更新文字]
   ├─ board.GetDisplay()->SetChatMessage(...)    [字幕滚动]
   ├─ audio_service.EnableVoiceProcessing(true) [开/关 AFE]
   └─ ...

Listening 状态特殊处理 (VAD 中变化):
   audio_processor 检测到 VadStateChange
   → on_vad_state_change 回调
   → MAIN_EVENT_VAD_STATE_CHANGED
   → Application 重新调 led->OnStateChanged()  ← LED 实时反映"有声音 / 静音"
```

8.8 本章用到的关键技术

技术	应用
多态 + Null Object 模式	NoDisplay / NoLed
RAII 锁守卫	DisplayLockGuard
LVGL flex 布局	OLED/LCD 容器嵌套
LVGL label long mode + anim	滚动字幕
`font_awesome_get_utf8`	字符串 → 字体码位
位图 + GIF + 字体三层 emoji fallback	LcdDisplay::SetEmotion
位字段 union	EmoteDisplay::AssetData
lvgl_port + task affinity	UI 任务绑核 1
RMT 驱动 WS2812B	SingleLed/CircularStrip
奇偶计数器 blink	简洁的闪烁实现
std::function + 闭包	CircularStrip 4 种动画
lambda 值捕获 vs 引用捕获	延迟执行的安全
lambda 内 static 变量	跨 timer 调用持久状态
LEDC 外设 + 13bit duty + 4kHz	GpioLed 平滑亮度
硬件 fade + ISR + TaskNotify	不占 CPU 的呼吸效果
IRAM_ATTR	ISR 放内置 SRAM
几何衰减（除 2 减半）	FadeOut 渐隐符合人眼感知
状态 → 颜色/节奏 映射表	用户体验一致性

8.9 看完本章你应该掌握的

- Display / Led 两个抽象基类的设计
- DisplayLockGuard RAII 守卫
- OledDisplay 128×64 / 128×32 两套 LVGL 布局

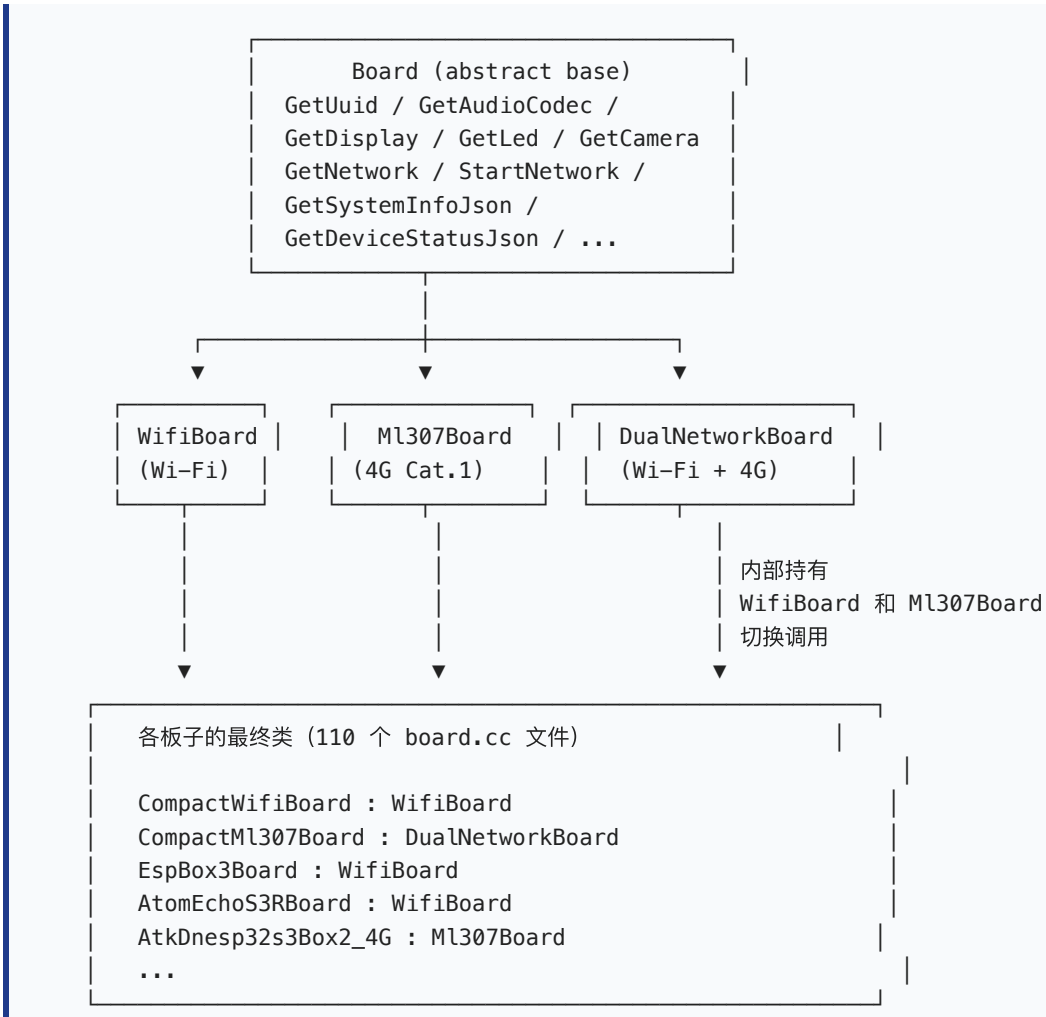
- Font Awesome 字符级 emoji
- LcdDisplay 三种总线（SPI/RGB/MIPI）的差别
- LcdDisplay 位图+GIF+字体三层 fallback
- EmoteDisplay 走自家 EmoteEngine 的原因（流畅度优化）
- AssetData 用位字段压缩存储
- SingleLed 状态→颜色映射的设计哲学
- CircularStrip 4 种动画（Blink/FadeOut/Breathe/Scroll）实现
- GpioLed 用 LEDC 硬件 fade 实现呼吸（CPU 零占用）
- ISR → TaskNotify → Task 的标准 ESP-IDF 模式

下一章覆盖 `boards/` 板级抽象 + 几个代表性板子细节。

第 9 章 板级抽象：`main/boards/`

> 110 个板子目录、4250+ 行共用代码。这一层把芯片型号、网络栈（Wi-Fi vs 4G）、屏幕型号、按键布局、电源管理、外设差异全部封装在 `Board` 类层级里。本章先讲共用基础设施（`common/`），再挑 3 个代表性板子拆开看。

9.1 板级机制鸟瞰



共用零件（在 main/boards/common/）：
- Button 按键封装
- Backlight 背光控制
- PowerSaveTimer / SleepTimer
- SystemReset 长按复位
- PressToTalkMcpTool 按住说话 MCP 工具
- Esp32Camera 摄像头封装
- Axp2101 / Sy6970 电源管理芯片 I²C 驱动
- I2cDevice 通用 I²C 设备基类
- Knob 旋钮编码器
- LampController 灯控（MCP 演示用）
- AfskDemod 声波配网解调
- Blufi BLE 配网
- AdcBatteryMonitor ADC 测电池电压

110 个 `boards/<xxx>/<xxx>_board.cc` 都遵守一个套路：

1. **继承** `WifiBoard` / `Ml307Board` / `DualNetworkBoard`；
2. **私有成员**：按键对象、I²C bus、面板句柄、display 指针；
3. **Initialize* 系列**：分别初始化 I²C / 显示屏 / 按键 / 工具；
4. **重写**：`GetLed()` / `GetAudioCodec()` / `GetDisplay()`（这三个必须）；
5. **构造函数**：按 init 函数顺序依次调用；
6. **`DECLARE_BOARD(BoardClassName)` 宏**注册类工厂。

9.2 `Board` 抽象基类详解

9.2.1 单例工厂模式

```
class Board {
public:
    static Board& GetInstance() {
        static Board* instance = static_cast<Board*>(create_board());
        return *instance;
    }
    // ...
};

#define DECLARE_BOARD(BOARD_CLASS_NAME) \
void* create_board() { \
    return new BOARD_CLASS_NAME(); \
}
```

`create_board()` 是个全局 C 函数，由具体板子在文件末尾用 `DECLARE_BOARD(MyBoard)` 宏定义。链接时只能有一个实现——所以**编译时通过 CMake 选择板子**：

```
# main/CMakeLists.txt 大致
if(CONFIG_BOARD_TYPE_BREAD_COMPACT_WIFI)
    set(BOARD_TYPE "bread-compact-wifi")
endif()
target_sources("${COMPONENT_LIB}" PRIVATE
        "${BUILD_PATH}/boards/${BOARD_TYPE}/${BOARD_TYPE}_board.cc")
```

`Board::GetInstance()` 内 `create_board()` 返回新 `new` 的具体板子对象。**裸 new + 永不 delete**——程序运行期单例。

9.2.2 关键接口

```
class Board {
public:
    virtual std::string GetBoardType() = 0;                 // "wifi" or "ml307"
    virtual std::string GetUuid() { return uuid_; }         // 软件 UUID
    virtual Backlight* GetBacklight() { return nullptr; }
    virtual Led* GetLed();                                   // 默认 NoLed
    virtual AudioCodec* GetAudioCodec() = 0;                // 必须
    virtual bool GetTemperature(float& esp32temp);
    virtual Display* GetDisplay();                          // 默认 NoDisplay
    virtual Camera* GetCamera();                            // 默认 nullptr
    virtual NetworkInterface* GetNetwork() = 0;
    virtual void StartNetwork() = 0;
    virtual void SetNetworkEventCallback(NetworkEventCallback);
    virtual const char* GetNetworkStateIcon() = 0;
    virtual bool GetBatteryLevel(int &level, bool& charging, bool& discharging);
    virtual std::string GetSystemInfoJson();
    virtual void SetPowerSaveLevel(PowerSaveLevel level) = 0;
    virtual std::string GetBoardJson() = 0;
    virtual std::string GetDeviceStatusJson() = 0;
};
```

接口	默认实现	设计意图
GetUuid	返回 uuid_	NVS 生成或读取
GetLed	NoLed	不会崩，板子可省略
GetDisplay	NoDisplay	同上
GetCamera	nullptr	take_photo 工具自动跳过
GetBacklight	nullptr	screen.set_brightness 自动跳过
GetBatteryLevel	false	状态栏不显示电池
GetAudioCodec	**纯虚**	必须实现，没声音的板子 → NoAudioCodec
GetNetwork / StartNetwork / GetBoardJson 等	**纯虚**	网络栈相关，必须实现

9.2.3 `Board::Board()` 构造 —— UUID 生成

```
Board::Board() {
    Settings settings("board", true);
    uuid_ = settings.GetString("uuid");
    if (uuid_.empty()) {
        uuid_ = GenerateUuid();
        settings.SetString("uuid", uuid_);
    }
}
```

UUID 设计：

- 首次启动 NVS 没有 → 调 `GenerateUuid()` 生成；
- 后续启动直接读 NVS；
- 即使设备重置 NVS（恢复出厂），下次开机会重新随机一次——所以 UUID 不是硬件唯一标识，**MAC 才是**。

9.2.4 `GenerateUuid()` —— 标准 UUID v4

```
std::string Board::GenerateUuid() {
    uint8_t uuid[16];
    esp_fill_random(uuid, sizeof(uuid));   // ESP32 硬件 TRNG
    uuid[6] = (uuid[6] & 0x0F) | 0x40;     // 版本 4
    uuid[8] = (uuid[8] & 0x3F) | 0x80;     // 变体 1
    char uuid_str[37];
    snprintf(uuid_str, sizeof(uuid_str),
        "%02x%02x%02x%02x-%02x%02x-%02x%02x-%02x%02x-%02x%02x%02x%02x%02x%02x",
        uuid[0], ..., uuid[15]);
    return std::string(uuid_str);
}
```

按 RFC 4122 UUIDv4 规范：

- 第 7 字节（uuid[6]）高 4 位强制 0100（版本号 4）；
- 第 9 字节（uuid[8]）高 2 位强制 10（变体 RFC 4122）；
- 其它 122 位由硬件 TRNG 填充（`esp_fill_random` 用芯片随机数发生器，质量好）。

输出形如 `f47ac10b-58cc-4372-a567-0e02b2c3d479`。

9.2.5 `GetSystemInfoJson()` —— 设备自报家门

这个函数是 OTA 阶段 POST 给服务器的请求体，含：

- 协议版本：`"version": 2`
- 语言代码：从 `Lang::CODE` 拿（zh-CN / en-US / 日语等）
- Flash 大小、剩余堆
- MAC、UUID
- 芯片型号（cores / revision / features 位）
- 应用名 / 版本号 / 编译时间 / IDF 版本 / ELF SHA256
- 分区表（label / type / address / size 每个分区）
- 当前 OTA 运行分区 label
- 显示信息（monochrome / width / height）
- 板子自己附加的 JSON（`GetBoardJson()` 返回）

手写字符串拼接 JSON——不用 cJSON 因为是一次性序列化，自己拼性能更好：

```
json += R"("mac_address":")" + SystemInfo::GetMacAddress() + R"(",)";
```

`R"(...)"` 是 C++11 **原始字符串字面量**——`\"` 这种转义可以直接写 `"`，写 JSON 模板省心。

服务器拿到这堆信息可以做：

- 按芯片型号下发不同 mqtt/websocket 配置；
- 按版本号决定是否给新固件 URL；
- 按 elf_sha256 做篡改检测（已知设备就该跑某版固件）；
- 按显示器类型给不同 emoji 资源包。

9.3 `WifiBoard` —— Wi-Fi 板基类

359 行实现，处理 Wi-Fi 板的核心逻辑：

9.3.1 状态机：连接 → 超时 → 配网

```cpp
void WifiBoard::StartNetwork() {
    auto& wifi_manager = WifiManager::GetInstance();
    WifiManagerConfig config;
    config.ssid_prefix = "Xiaozhi";
    config.language = Lang::CODE;
    wifi_manager.Initialize(config);

    wifi_manager.SetEventCallback([this, &wifi_manager](WifiEvent event) {
        // 把 WifiEvent 翻译成统一的 NetworkEvent
        std::string ssid = wifi_manager.GetSsid();
        switch (event) {
            case WifiEvent::Connected: OnNetworkEvent(NetworkEvent::Connected, ssid); break;
            // ...
        }
    });
    TryWifiConnect();
}

void WifiBoard::TryWifiConnect() {
    auto& ssid_manager = SsidManager::GetInstance();
    bool have_ssid = !ssid_manager.GetSsidList().empty();
    if (have_ssid) {
        esp_timer_start_once(connect_timer_, CONNECT_TIMEOUT_SEC * 1000000ULL);  // 60 秒超时
        WifiManager::GetInstance().StartStation();
    } else {
        vTaskDelay(pdMS_TO_TICKS(1500));   // 等屏幕显示版本信息
        StartWifiConfigMode();
    }
}

void WifiBoard::OnWifiConnectTimeout(void* arg) {
    auto* board = static_cast<WifiBoard*>(arg);
    WifiManager::GetInstance().StopStation();
    board->StartWifiConfigMode();
}
```

3 种状态：

1. **有 SSID + 能连**：60 秒内 `Connected` 触发 → `esp_timer_stop` 取消超时；
2. **有 SSID 但连不上**：60 秒到 → 进入配网模式；
3. **没 SSID**：等 1.5 秒（让屏幕显示完版本号）→ 进入配网模式。

9.3.2 三种配网方式 (Kconfig 任选)

```
void WifiBoard::StartWifiConfigMode() {
    in_config_mode_ = true;
    Application::GetInstance().SetDeviceState(kDeviceStateWifiConfiguring);

#ifdef CONFIG_USE_HOTSPOT_WIFI_PROVISIONING
    // 方式 1: 设备开热点 + 网页配置
    auto& wifi_manager = WifiManager::GetInstance();
    wifi_manager.StartConfigAp();
    Application::GetInstance().Schedule([&wifi_manager]() {
        std::string hint = Lang::Strings::CONNECT_TO_HOTSPOT;
        hint += wifi_manager.GetApSsid();
        hint += Lang::Strings::ACCESS_VIA_BROWSER;
        hint += wifi_manager.GetApWebUrl();
        Application::GetInstance().Alert(Lang::Strings::WIFI_CONFIG_MODE, hint.c_str(),
        "gear", Lang::Sounds::OGG_WIFICONFIG);
    });
#endif

#if CONFIG_USE_ESP_BLUFI_WIFI_PROVISIONING
    // 方式 2: BLE 配网（手机 App）
    Blufi::GetInstance().init();
#endif

#if CONFIG_USE_ACOUSTIC_WIFI_PROVISIONING
    // 方式 3: 声波配网 (AFSK)
    xTaskCreate([](void* arg) {
        auto ch = reinterpret_cast<intptr_t>(arg);
        audio_wifi_config::ReceiveWifiCredentialsFromAudio(&app, &wifi, disp, ch);
        vTaskDelete(NULL);
    }, "acoustic_wifi", 4096, reinterpret_cast<void*>(channel), 2, NULL);
#endif
}
```

配网方式	实现	用户操作
Hotspot 网页	wifi_manager.StartConfigAp + HTTP server	手机连“Xiaozhi-xxx”热点 → 浏览器输 SSID/密码
BLE BluFi	Espressif BluFi 协议	手机 App“EspBluFi”蓝牙发 SSID/密码
声波 AFSK	麦克风听 AFSK 调制信号解调	手机 App 播放声音 → 设备麦克风收

三种**可同时启用**——用户选最方便的方式。声波最神奇：完全不需要 BLE/AP，弱网环境下也能搞定。

9.3.3 `OnNetworkEvent` —— 事件分发

```
void WifiBoard::OnNetworkEvent(NetworkEvent event, const std::string& data) {
    switch (event) {
        case NetworkEvent::Connected:
            esp_timer_stop(connect_timer_);     // 取消超时
            Blufi::GetInstance().deinit();      // 释放 BluFi 资源
            in_config_mode_ = false;
            break;
        case NetworkEvent::WifiConfigModeExit:
            in_config_mode_ = false;
            TryWifiConnect();                   // 重新连
            break;
        // ...
    }
    if (network_event_callback_) {
        network_event_callback_(event, data);   // 转发给上层
    }
}
```

业务层（Application）通过 `SetNetworkEventCallback` 注册自己的回调，所有网络事件都会过来。

9.3.4 `GetDeviceStatusJson()` —— MCP 工具用的状态查询

```
std::string WifiBoard::GetDeviceStatusJson() {
    // 拼成类似:
    // { "audio_speaker": {"volume": 70}, "screen": {"brightness": 50},
    //   "battery": {"level": 80, "charging": true}, "network": {"type": "wifi", "ssid":
        "xxx", "rssi": -55} }
    // 返回给 self.get_device_status 工具
}
```

MCP 工具 `self.get_device_status`（第 6 章 6.7.2）的实现来源。

9.3.5 `GetNetworkStateIcon()`

```
const char* WifiBoard::GetNetworkStateIcon() {
    if (in_config_mode_) return FONT_AWESOME_GEAR;
    auto rssi = WifiManager::GetInstance().GetRssi();
    if (rssi == 0) return FONT_AWESOME_WIFI_SLASH;
    if (rssi > -50) return FONT_AWESOME_WIFI_HIGH;
    if (rssi > -65) return FONT_AWESOME_WIFI_MEDIUM;
    return FONT_AWESOME_WIFI_LOW;
}
```

屏幕顶部 network_label_ 显示什么 Font Awesome 图标。RSSI 阈值是经验值（-50 强、-65 中、其它弱）。

9.4 `Ml307Board` —— 4G 蜂窝板基类

274 行。基于上海移芯通信的 **ML307R Cat.1 模块**（UART AT 命令控制）。

特点（vs WiFi）：

- **无配网阶段**——SIM 卡插上就能用，无需用户参与；
- **PPP 拨号上网**：ESP32 通过 UART 跟 ML307 通信，ML307 跟基站建立 PPP 链路；

- **网络速度有限**（Cat.1 上行约 5Mbps 下行 10Mbps，但延迟稳定）；
- **状态比 Wi-Fi 多**：SIM 卡未插 / 网络未注册 / 信号弱 / 漫游等。

9.4.1 Modem 检测和错误事件

```
enum class NetworkEvent {
    // ... Wi-Fi 共用的事件
    ModemDetecting,       // 正在自动识别 baud rate + 模块型号
    ModemErrorNoSim,      // 没 SIM 卡
    ModemErrorRegDenied,  // 网络注册被运营商拒绝
    ModemErrorInitFailed, // 初始化失败
    ModemErrorTimeout     // 超时
};
```

`Ml307Board::StartNetwork()` 内部依次：

1. 初始化 UART（最常见 921600 baud）；
2. 发 AT 命令检测模块在线（响应 OK）；
3. 检测模块型号（`AT+CGMM`）；
4. 检测 SIM 卡状态（`AT+CPIN?`）；
5. 等待网络注册（`AT+CREG?` 返回 1 或 5 表示已注册本地或漫游）；
6. 启动 PPP；
7. 触发 `Connected` 事件。

每一步失败都触发对应 ModemError* 事件，让屏幕显示具体原因。

9.4.2 `GetNetwork()` 返回的不同栈

```
NetworkInterface* WifiBoard::GetNetwork() {
    return EspNetworkAdapter::GetInstance();      // ESP-IDF 自带 LWIP 栈
}

NetworkInterface* Ml307Board::GetNetwork() {
    return Ml307AtModem::GetInstance();           // ML307 内置 TCP/IP 栈，通过 AT 命令调用
}
```

关键设计：`NetworkInterface` 是抽象接口，定义了 `CreateHttp / CreateWebSocket / CreateMqtt / CreateUdp` 4 个工厂方法。Wi-Fi 板返回基于 LWIP 的实现，4G 板返回基于 ML307 AT 命令的实现——**协议层完全感知不到差异**。

这就是为什么第 5 章 protocols 全部代码可以同时在 Wi-Fi 和 4G 板上跑。

9.5 `DualNetworkBoard` —— 双网切换

98 行，最简洁但用法巧妙。

9.5.1 思路

某些板子既有 Wi-Fi 又有 4G 模块（如 `atk-dnesp32s3-box2-4g`），用户可以选用其中之一：

- 室内 Wi-Fi 信号好 → 用 Wi-Fi 省流量；
- 户外 / Wi-Fi 不稳定 → 切到 4G 用流量包。

实现方式不是合并两套代码，而是**内部包两个 sub-board**：

```
class DualNetworkBoard : public Board {
private:
    std::unique_ptr<WifiBoard> wifi_board_;
    std::unique_ptr<Ml307Board> ml307_board_;
    NetworkType current_type_ = NetworkType::WIFI;

public:
    DualNetworkBoard(int ml307_tx, int ml307_rx, gpio_num_t reset) {
        // 创建两个内部子板
        Settings settings("board", false);
        std::string saved = settings.GetString("network_type", "wifi");
        if (saved == "ml307") current_type_ = NetworkType::ML307;
    }

    Board& GetCurrentBoard() {
        if (current_type_ == NetworkType::WIFI) return *wifi_board_;
        else return *ml307_board_;
    }

    // 所有接口都转发给 GetCurrentBoard()
    NetworkInterface* GetNetwork() override { return GetCurrentBoard().GetNetwork(); }
    void StartNetwork() override { GetCurrentBoard().StartNetwork(); }
    // ...

    void SwitchNetworkType() {
        current_type_ = (current_type_ == NetworkType::WIFI) ? NetworkType::ML307 :
        NetworkType::WIFI;
        Settings settings("board", true);
        settings.SetString("network_type", current_type_ == NetworkType::WIFI ? "wifi" :
        "ml307");
        esp_restart();   // 简单粗暴：重启重新初始化
    }
};
```

设计要点：

- **当前网络类型存 NVS**——下次启动直接用上次的；
- 接口透传给当前 sub-board；
- **`SwitchNetworkType()` 直接重启**——避免在运行时切换网络栈带来的复杂状态机问题（哪个网络断开、新的开起来、所有连接重建等）。重启代价就是几秒钟。

用户触发切换：长按某个按键、按某个组合键或菜单选项——具体哪个按键由板子决定。

9.6 共用零件：`Button` / `Backlight` / `PowerSaveTimer`

9.6.1 `Button` —— 6 种事件

```
class Button {
public:
    Button(gpio_num_t gpio_num, bool active_high = false,
           uint16_t long_press_time = 0, uint16_t short_press_time = 0,
           bool enable_power_save = false);
    void OnPressDown(std::function<void()>);
    void OnPressUp(std::function<void()>);
    void OnLongPress(std::function<void()>);
    void OnClick(std::function<void()>);
    void OnDoubleClick(std::function<void()>);
    void OnMultipleClick(std::function<void()>, uint8_t click_count = 3);
};
```

底层基于 `iot_button` 组件（Espressif 提供的按键库）：

- **去抖**：内部 debounce 算法过滤毛刺；
- **6 类事件**：按下/抬起/长按/单击/双击/多击；
- **`enable_power_save`**：light-sleep 模式下保持响应（特殊配置）；
- **`active_high`**：按下为高电平还是低电平。

典型用法（compact_wifi_board.cc 第 103 行）：

```
boot_button_.OnClick([this]() {
    auto& app = Application::GetInstance();
    if (app.GetDeviceState() == kDeviceStateStarting) {
        EnterWifiConfigMode();
        return;
    }
    app.ToggleChatState();   // 启动后单击 boot 键开始对话
});

touch_button_.OnPressDown([this]() {
    Application::GetInstance().StartListening();
});
touch_button_.OnPressUp([this]() {
    Application::GetInstance().StopListening();
});
```

按住 touch 按键说话（PressDown → PressUp 配对），单击 boot 按键切换状态。

9.6.2 `Backlight` —— 屏幕背光控制

`backlight.h`：

```
class Backlight {
public:
    Backlight(...);
    void SetBrightness(uint8_t brightness, bool save = false);
    uint8_t GetBrightness() const;
    void RestoreBrightness();   // 从 NVS 读回
};
```

背光是独立的 PWM GPIO（不是 LCD 的内部控制），用 LEDC 外设调节亮度。`save = true` 时写 NVS `display.brightness`，下次开机恢复。

9.6.3 `PowerSaveTimer` + `SleepTimer`

两个 timer 配合实现省电：

- **PowerSaveTimer**：进入 idle 状态 N 秒后**降频**（CPU 80MHz → 40MHz）；
- **SleepTimer**：再 idle M 秒后**进入 light_sleep**（CPU 完全停，等中断唤醒）。

```
// 简化实现
class PowerSaveTimer {
    void Start() { esp_timer_start_once(timer_, timeout * 1000000); }
    void Stop() { esp_timer_stop(timer_); }
    void OnTimer() {
        auto state = Application::GetInstance().GetDeviceState();
        if (state == kDeviceStateIdle) {
            Board::GetInstance().SetPowerSaveLevel(PowerSaveLevel::LOW_POWER);
        }
    }
};
```

非 Idle 状态（说话/听话/升级中）会**重置 timer**——保证不会在工作时降频。

电池供电板（atom-echo 等）一定要开省电否则跑不了几小时，USB 供电板可以选择性关闭。

9.7 代表板子 1：`bread-compact-wifi`（最小 Wi-Fi 麦克风）

定位：**入门面包板套件**。最便宜的配置——ESP32-S3 + OLED + 单麦克风 + 单喇叭 + 4 按键 + 1 WS2812B。

9.7.1 配置（`config.h`）

```
#define AUDIO_INPUT_SAMPLE_RATE  16000
#define AUDIO_OUTPUT_SAMPLE_RATE 24000
#define AUDIO_I2S_METHOD_SIMPLEX

// I²S 麦克风（MEMS，标准 SPH0645）
#define AUDIO_I2S_MIC_GPIO_WS   GPIO_NUM_4
#define AUDIO_I2S_MIC_GPIO_SCK  GPIO_NUM_5
#define AUDIO_I2S_MIC_GPIO_DIN  GPIO_NUM_6

// I²S 喇叭（PCM5102 / MAX98357 等）
#define AUDIO_I2S_SPK_GPIO_DOUT GPIO_NUM_7
#define AUDIO_I2S_SPK_GPIO_BCLK GPIO_NUM_15
#define AUDIO_I2S_SPK_GPIO_LRCK GPIO_NUM_16

#define BUILTIN_LED_GPIO        GPIO_NUM_48   // WS2812B
#define BOOT_BUTTON_GPIO        GPIO_NUM_0
#define TOUCH_BUTTON_GPIO       GPIO_NUM_47
#define VOLUME_UP_BUTTON_GPIO   GPIO_NUM_40
#define VOLUME_DOWN_BUTTON_GPIO GPIO_NUM_39

#define DISPLAY_SDA_PIN GPIO_NUM_41
#define DISPLAY_SCL_PIN GPIO_NUM_42
#define DISPLAY_WIDTH   128

// Kconfig 选择哪种 OLED
#if CONFIG_OLED_SSD1306_128X32
#define DISPLAY_HEIGHT  32
#elif CONFIG_OLED_SSD1306_128X64
#define DISPLAY_HEIGHT  64
#elif CONFIG_OLED_SH1106_128X64
#define DISPLAY_HEIGHT  64
#define SH1106
#endif

#define LAMP_GPIO GPIO_NUM_18   // MCP 演示：控制一颗灯
```

Simplex I²S：用两组 I²S 引脚（一组麦克风 in，一组喇叭 out），不共享 BCLK/WS。逻辑简单，开发板布线灵活。

为什么 input 16k / output 24k：

- 输入 16k 够覆盖人声频段（8k 奈奎斯特），AFE/wake word 都按 16k 设计；
- 输出 24k 是 LLM TTS 的常用配置（音色更好），但稍微多耗带宽。

9.7.2 实现（`compact_wifi_board.cc`）

```
class CompactWifiBoard : public WifiBoard {
private:
    i2c_master_bus_handle_t display_i2c_bus_;
    esp_lcd_panel_io_handle_t panel_io_;
    esp_lcd_panel_handle_t panel_;
    Display* display_ = nullptr;
    Button boot_button_;
    Button touch_button_;
    Button volume_up_button_;
    Button volume_down_button_;

    void InitializeDisplayI2c() {
        i2c_master_bus_config_t bus_config = {
            .i2c_port = 0,
            .sda_io_num = DISPLAY_SDA_PIN,
            .scl_io_num = DISPLAY_SCL_PIN,
            .clk_source = I2C_CLK_SRC_DEFAULT,
            .glitch_ignore_cnt = 7,
            .flags = { .enable_internal_pullup = 1 },
        };
        i2c_new_master_bus(&bus_config, &display_i2c_bus_);
    }

    void InitializeSsd1306Display() {
        esp_lcd_panel_io_i2c_config_t io_config = {
            .dev_addr = 0x3C,        // SSD1306 I²C 地址固定 0x3C
            .control_phase_bytes = 1,
            .dc_bit_offset = 6,
            .lcd_cmd_bits = 8,
            .lcd_param_bits = 8,
            .scl_speed_hz = 400 * 1000,   // 400 kHz Fast Mode
        };
        esp_lcd_new_panel_io_i2c_v2(display_i2c_bus_, &io_config, &panel_io_);

        esp_lcd_panel_dev_config_t panel_config = {};
        panel_config.reset_gpio_num = -1;
        panel_config.bits_per_pixel = 1;

        esp_lcd_panel_ssd1306_config_t ssd1306_config = { .height = static_cast<uint8_t>
        (DISPLAY_HEIGHT) };
        panel_config.vendor_config = &ssd1306_config;
#ifdef SH1106
        esp_lcd_new_panel_sh1106(panel_io_, &panel_config, &panel_);
#else
        esp_lcd_new_panel_ssd1306(panel_io_, &panel_config, &panel_);
#endif
        esp_lcd_panel_reset(panel_);
        if (esp_lcd_panel_init(panel_) != ESP_OK) {
            display_ = new NoDisplay();    // 屏挂了 fallback
            return;
        }
        esp_lcd_panel_disp_on_off(panel_, true);
        display_ = new OledDisplay(panel_io_, panel_, DISPLAY_WIDTH, DISPLAY_HEIGHT,
        DISPLAY_MIRROR_X, DISPLAY_MIRROR_Y);
    }

    void InitializeButtons() {
        boot_button_.OnClick([this]() {
            auto& app = Application::GetInstance();
```

```
            if (app.GetDeviceState() == kDeviceStateStarting) {
                EnterWifiConfigMode();   // 启动阶段单击进配网
                return;
            }
            app.ToggleChatState();        // 已启动单击开始/结束对话
        });
        touch_button_.OnPressDown([this]() {
            Application::GetInstance().StartListening();
        });
        touch_button_.OnPressUp([this]() {
            Application::GetInstance().StopListening();
        });
        volume_up_button_.OnClick([this]() {
            auto codec = GetAudioCodec();
            auto volume = codec->output_volume() + 10;
            if (volume > 100) volume = 100;
            codec->SetOutputVolume(volume);
            GetDisplay()->ShowNotification(Lang::Strings::VOLUME + std::to_string(volume));
        });
        volume_up_button_.OnLongPress([this]() {
            GetAudioCodec()->SetOutputVolume(100);
            GetDisplay()->ShowNotification(Lang::Strings::MAX_VOLUME);
        });
        // ... 类似 volume_down_button_
    }

    void InitializeTools() {
        static LampController lamp(LAMP_GPIO);   // 注册一个 MCP 工具控制 GPIO18 上的灯
    }

public:
    CompactWifiBoard() :
        boot_button_(BOOT_BUTTON_GPIO),
        touch_button_(TOUCH_BUTTON_GPIO),
        volume_up_button_(VOLUME_UP_BUTTON_GPIO),
        volume_down_button_(VOLUME_DOWN_BUTTON_GPIO) {
        InitializeDisplayI2c();
        InitializeSsd1306Display();
        InitializeButtons();
        InitializeTools();
    }

    Led* GetLed() override {
        static SingleLed led(BUILTIN_LED_GPIO);   // 单颗 WS2812B
        return &led;
    }

    AudioCodec* GetAudioCodec() override {
        static NoAudioCodecSimplex audio_codec(
            AUDIO_INPUT_SAMPLE_RATE, AUDIO_OUTPUT_SAMPLE_RATE,
            AUDIO_I2S_SPK_GPIO_BCLK, AUDIO_I2S_SPK_GPIO_LRCK, AUDIO_I2S_SPK_GPIO_DOUT,
            AUDIO_I2S_MIC_GPIO_SCK, AUDIO_I2S_MIC_GPIO_WS, AUDIO_I2S_MIC_GPIO_DIN);
        return &audio_codec;
    }

    Display* GetDisplay() override { return display_; }
};
```

```
DECLARE_BOARD(CompactWifiBoard);
```

典型流程：

1. 构造函数初始化 4 个按键句柄；
2. 函数体里依次跑 4 个 Init 函数；
3. 5 个虚函数重写（implicit Wi-Fi 网络）。

注意 `LampController lamp(LAMP_GPIO)` 是 static 局部变量——首次调用 `InitializeTools()` 时构造，构造里向 `McpServer` 注册一个“灯开关”工具，让 LLM 可以”打开桌上的灯”。这是项目自带的最小 MCP 演示。

9.8 代表板子 2：`bread-compact-ml307`（4G 双模版本）

跟 9.7 几乎一样，但继承 `DualNetworkBoard`：

```
class CompactMl307Board : public DualNetworkBoard {
public:
    CompactMl307Board() : DualNetworkBoard(ML307_TX_PIN, ML307_RX_PIN, GPIO_NUM_NC), ... { }
};
```

特有逻辑：

```
boot_button_.OnClick([this]() {
    if (GetNetworkType() == NetworkType::WIFI) {
        if (app.GetDeviceState() == kDeviceStateStarting) {
            // 转换 wifi_board 调 EnterWifiConfigMode
            auto& wifi_board = static_cast<WifiBoard&>(GetCurrentBoard());
            wifi_board.EnterWifiConfigMode();
            return;
        }
    }
    app.ToggleChatState();
});
boot_button_.OnDoubleClick([this]() {
    if (app.GetDeviceState() == kDeviceStateStarting || app.GetDeviceState() ==
        kDeviceStateWifiConfiguring) {
        SwitchNetworkType();   // 双击切换 Wi-Fi / 4G
    }
});
```

双击切换网络类型，单击在 Wi-Fi 模式下进配网。`static_cast<WifiBoard&>` 是因为已经知道 `GetNetworkType() == WIFI`。

9.9 代表板子 3：`esp-box-3`（旗舰彩屏板）

文件清单不一样：

```
esp-box-3/
├── config.h           GPIO 定义
├── config.json        板子元数据
├── esp_box3_board.cc 板子实现（继承 WifiBoard）
├── emote.json         EmoteDisplay 用的表情/图标/布局
├── layout.json        UI 元素位置
└── README.md
```

特点：

- **彩色 320×240 IPS LCD**（ST7789 SPI 总线，走 SpiLcdDisplay 路径，但本板用 EmoteDisplay）；
- **电容触摸屏**（GT911 I²C）；
- **ES8311 音频编解码器**（专业 codec 替代裸 I²S+MEMS 麦克风）；
- **背光 PWM 控制**；
- **加速度计 BMI270**（可选体感操作）；
- **更大 SPI Flash（16 MB） + PSRAM**（必须，跑 LVGL 彩屏耗内存）。

`esp_box3_board.cc` 比 compact 板代码量大 3–4 倍，但套路完全一样：

- I²C bus 初始化（共用一条 bus 接 GT911 + ES8311 + 加速度计）；
- LCD 初始化（SPI panel + ST7789 driver）；
- 触摸初始化；
- 音频 codec 初始化（`Es8311AudioCodec`）；
- 按键（板上 3 个物理键）；
- Backlight；
- 各种 MCP 工具注册。

由于代码量较大不全文复述。读者按需求点开 `esp_box3_board.cc` 看。**关键认知**：所有彩屏板都按”I²C bus + LCD panel + 触摸 + codec + 按键 + 工具”6 部分组织。

9.10 其它 107 个板子的简表

按命名前缀整理：

前缀	数量	类型	代表
`atk-dnesp32s3*`	7	正点原子 S3 系列（含 4G）	atk-dnesp32s3-box, atk-dnesp32s3m-4g
`atom-*` / `atoms3*`	5	M5Stack Atom Echo 系列	atom-echos3r, atoms3-echo-base
`bread-compact-*`	5	面包板套件（最入门）	bread-compact-wifi/-ml307/-lcd
`esp-box*`	3	Espressif 官方 Box 板	esp-box, esp-box-3, esp-box-lite
`df-*`	2	DFRobot 系列	df-k10, df-s3-ai-cam
`esp32-*`	4	普通 ESP32（非 S3）	esp32-cgc, esp32-cgc-144
`esp32-s3-touch-*`	多个	触摸彩屏板	s3-touch-amoled-1.8, s3-touch-lcd-1.46
`doit-s3-aibox` / `du-chatx` / `echoear`	3	设备厂私有板	—
`esp-p4-function-ev-board`	1	P4 评估板（最高端）	—
`esp-s3-lcd-ev-board*`	2	Espressif S3 RGB LCD 评估板	—
`esp-spot`	1	最便宜的 S3 套件	—
`esp-hi`	1	高端音质板	—
`electron-bot`	1	桌面机器人形态	—
`esp-sensairshuttle`	1	带传感器扩展板	—
`esp-sparkbot`	1	小机器人	—
其它 60+	—	各种第三方板子	看名字推测

每个板子大致这些信息：

- `config.h`：GPIO 定义 + 屏幕尺寸 + 采样率；
- `config.json`：板子元数据（名字、描述、链接、屏幕信息）；
- `<name>_board.cc`：板子类实现（按 9.7 的套路）；
- 可选 `emote.json` / `layout.json`：EmoteDisplay 资源；
- 可选 `README.md`：板子专属说明。

按 chip / 显示器 / 麦克风 / 喇叭 / 网络栈 / 电源 6 个维度去推断板子组合即可。

9.11 加 `boards/<new_board>/` 添新板的最小步骤

1. **复制最接近的板子目录**作为模板，重命名；
2. **改 `config.h`**：所有 GPIO + 屏幕尺寸 + 采样率改成新板；
3. **改 `<name>_board.cc`**：类名 + 显示器 driver + codec 类型 + Init 函数；
4. **加 `DECLARE_BOARD(NewBoardName)`** 在文件末尾；
5. **在 `Kconfig.projbuild`** 加 `config BOARD_TYPE_<NEW_NAME>`，并把它加入 `choice BOARD_TYPE`；

6. 在 `CMakeLists.txt` 加对应的 `if(CONFIG_BOARD_TYPE_<NEW_NAME>) set(BOARD_TYPE "<new_name>") endif()`；
7. **更新 `partitions/`**：根据 Flash 大小选 4MB / 8MB / 16MB 分区表；
8. **`menuconfig` 选板子** → 编译 → 烧录。

整个过程 100% 不动其他 109 个板子和上层代码。

9.12 本章用到的关键技术

技术	应用
抽象基类 + 多层继承	Board → WifiBoard → 具体板
DECLARE_BOARD 宏 + 链接选板	编译时选板，零运行时开销
NetworkInterface 抽象	Wi-Fi/4G 网络栈对协议层透明
`std::unique_ptr` 持有 sub-board	DualNetworkBoard 双栈
RFC 4122 UUIDv4 + esp_fill_random	设备软件唯一标识
`R"(...)"` 原始字符串字面量	手写 JSON 模板
NVS 持久化 network_type / uuid / brightness	跨重启状态
`iot_button` + 6 类事件	按键标准化
LEDC 外设 PWM 控背光	屏幕亮度
I²C bus 共享多设备	节省 GPIO（display + codec + touch 共一组 SDA/SCL）
`esp_lcd_panel_*` API	屏幕驱动统一接口（SSD1306 / SH1106 / ST7789 / GC9A01 等）
dynamic_cast 检查子板类型	DualNetworkBoard 切到 WifiBoard 调专属方法
Static 局部变量做 lazy init	GetLed/GetAudioCodec 等的常见模式
3 种配网（Hotspot / BLE / 声波）+ Kconfig 同时启用	灵活适配
Modem AT 命令 + PPP	4G 板的网络初始化
`PowerSaveTimer` + `SleepTimer`	自动降频 + light_sleep

9.13 看完本章你应该掌握的

- `Board` 单例工厂 + `DECLARE_BOARD` 宏的工作原理
- 4 个继承层级（Board / WifiBoard / Ml307Board / DualNetworkBoard）
- 网络栈抽象 NetworkInterface 让协议层不感知 Wi-Fi vs 4G
- Wi-Fi 板的连接→超时→配网状态机
- 3 种配网方式（Hotspot / BluFi / Acoustic AFSK）
- 4G 板的 Modem 检测和错误事件
- DualNetworkBoard 双栈切换的”重启大法”
- Board 基类的 UUID 生成逻辑
- GetSystemInfoJson 手写 JSON 的内容结构
- Button 6 类事件 + 典型用法

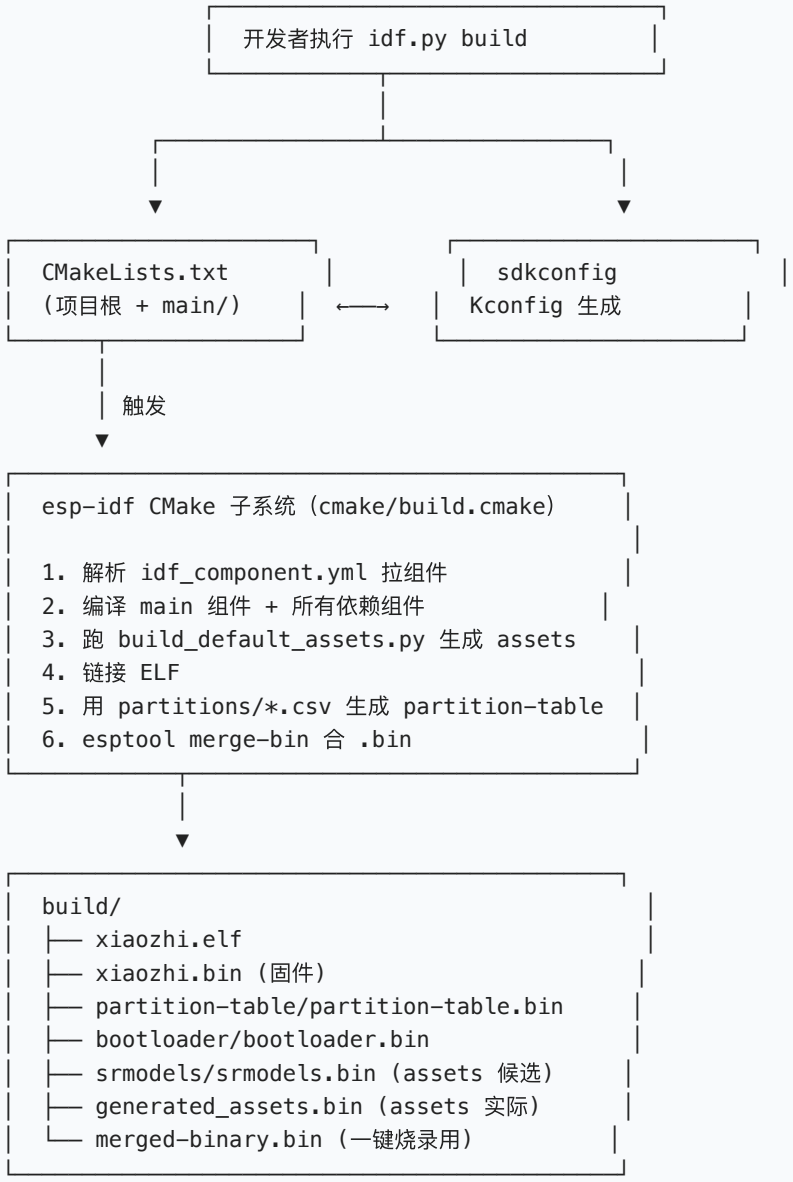
- 一个具体板子（compact-wifi）的 Init* / GetXxx 6 个标准函数
- 添加新板子的最小修改清单

下一章覆盖剩下的辅助资源：`scripts/` / `docs/` / `partitions/` / Kconfig / CMake。

第 10 章 构建系统与辅助资源

> 前 9 章覆盖的是 `main/` 下跑在 ESP32 上的运行时代码。本章是收尾——把分散在仓库根目录的 5 类辅助资源讲清楚：CMake 与 Kconfig（怎么从源码变成 `.bin`）、`partitions/`（Flash 怎么布局）、`scripts/`（Python 工具链）、`docs/`（设计文档与接线图）、`idf_component.yml`（依赖管理）。

10.1 构建系统总览



涉及的入口文件：

文件	角色
`CMakeLists.txt`（根）	声明项目名 + PROJECT_VER + 包含 `$ENV{IDF_PATH}/tools/cmake/project.cmake`
`main/CMakeLists.txt`	真正的"项目入口"：列源文件 + 根据 BOARD_TYPE 添加板子专属代码 + 注册 partition 烧录命令
`main/Kconfig.projbuild`	板子选项 / 语言选项 / 字体选项 / OTA URL 等可视化配置
`main/idf_component.yml`	列依赖的第三方组件（espressif/esp_lvgl_port、espressif__esp-sr、xiaozhi-fonts 等）和版本
`partitions/v{1,2}/*.csv`	Flash 分区表
`sdkconfig.defaults.*`	不同芯片型号的默认 Kconfig 值

10.2 `main/CMakeLists.txt` —— 板子选择的大门

837 行（去掉 Kconfig 部分），核心结构：

10.2.1 第 1 段：通用源文件列表（第 1–46 行）

```
set(SOURCES "audio/audio_codec.cc"
            "audio/audio_service.cc"
            "audio/codecs/no_audio_codec.cc"
            "audio/codecs/box_audio_codec.cc"
            ...
            "audio/processors/audio_debugger.cc"
            "led/single_led.cc"
            "led/circular_strip.cc"
            "led/gpio_led.cc"
            "display/display.cc"
            "display/lcd_display.cc"
            ...
            "protocols/protocol.cc"
            "protocols/mqtt_protocol.cc"
            "protocols/websocket_protocol.cc"
            "mcp_server.cc"
            "system_info.cc"
            "application.cc"
            "ota.cc"
            "settings.cc"
            "device_state_machine.cc"
            "assets.cc"
            "main.cc"
            )

set(INCLUDE_DIRS "." "display" "display/lvgl_display" "audio" "protocols")

# 板级共享文件（button.cc、backlight.cc、power_save_timer.cc 等）
file(GLOB BOARD_COMMON_SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/boards/common/*.cc)
list(APPEND SOURCES ${BOARD_COMMON_SOURCES})
list(APPEND INCLUDE_DIRS ${CMAKE_CURRENT_SOURCE_DIR}/boards/common)
```

所有 110 个板子都共享上面这套。 `file(GLOB ...)` 自动把 `boards/common/` 下所有 `.cc` 全打进去——加新的 common 工具时只要丢进去，不用改 CMakeLists。

10.2.2 第 2 段：BOARD_TYPE 派发（第 67–660 行）

110 个 `if-elseif-endif` 分支，每个分支：

```
elseif(CONFIG_BOARD_TYPE_ESP_BOX_3)
    set(BOARD_TYPE "esp-box-3")
    set(BUILTIN_TEXT_FONT font_puhui_basic_20_4)
    set(BUILTIN_ICON_FONT font_awesome_20_4)
    set(DEFAULT_EMOJI_COLLECTION twemoji_64)
```

设置 4 个变量：

变量	用途
`BOARD_TYPE`	拼接路径 `main/boards/<BOARD_TYPE>/<BOARD_TYPE>_board.cc`
`BUILTIN_TEXT_FONT`	编译进固件的中文字体（不依赖 assets 分区也能显示）
`BUILTIN_ICON_FONT`	编译进固件的 Font Awesome 图标字体
`DEFAULT_EMOJI_COLLECTION`	默认 emoji 集合（彩屏可用 twemoji_64，OLED 用 `none`）

字体之所以可选是因为：

- 大屏板用 20pt 字（可读性优先）+ Twemoji 64×64 彩色 emoji；
- 小屏板用 14pt 字（128×64 OLED 屏太小）+ 文字 emoji 代替；
- 这俩字体直接 `.a` 链进固件，**`assets.bin` 即使损坏也能显示界面**——容错设计。

随后：

```
list(APPEND SOURCES "boards/${BOARD_TYPE}/${BOARD_TYPE}_board.cc")
list(APPEND INCLUDE_DIRS "boards/${BOARD_TYPE}")
```

把对应板子的 `<name>_board.cc` 加入编译，把板子目录加入 include path（`config.h` 能被找到）。

10.2.3 第 3 段：动态查找组件路径（第 760–769 行）

```
find_component_by_pattern("espressif__esp-sr" ESP_SR_COMPONENT ESP_SR_COMPONENT_PATH)
if(ESP_SR_COMPONENT_PATH)
    set(ESP_SR_MODEL_PATH "${ESP_SR_COMPONENT_PATH}/model")
endif()

find_component_by_pattern("xiaozhi-fonts" XIAOZHI_FONTS_COMPONENT
        XIAOZHI_FONTS_COMPONENT_PATH)
if(XIAOZHI_FONTS_COMPONENT_PATH)
    set(XIAOZHI_FONTS_PATH "${XIAOZHI_FONTS_COMPONENT_PATH}")
endif()
```

ESP-IDF 组件被下载到 `managed_components/` 目录，但名字带 hash 和版本号（如 `espressif__esp-sr_2.1.5`），具体路径事先不知道。`find_component_by_pattern` 在第 50–59 行定义，遍历 `BUILD_COMPONENTS` 找匹配。

得到组件路径后，提取 `model/` 子目录（语音模型）和字体目录给后面 `build_default_assets.py` 用。

10.2.4 第 4 段：构建 assets.bin（第 810–857 行）

```
function(build_default_assets_bin)
    set(GENERATED_ASSETS_BIN "${CMAKE_BINARY_DIR}/generated_assets.bin")

    set(BUILD_ARGS
        "--sdkconfig" "${SDKCONFIG}"
        "--output" "${GENERATED_ASSETS_BIN}"
    )
    if(BUILTIN_TEXT_FONT) list(APPEND BUILD_ARGS "--builtin_text_font" "${BUILTIN_TEXT_FONT}")
        endif()
    if(DEFAULT_EMOJI_COLLECTION) list(APPEND BUILD_ARGS "--emoji_collection"
        "${DEFAULT_EMOJI_COLLECTION}") endif()
    if(DEFAULT_ASSETS_EXTRA_FILES) list(APPEND BUILD_ARGS "--extra_files"
        "${DEFAULT_ASSETS_EXTRA_FILES}") endif()
    list(APPEND BUILD_ARGS "--esp_sr_model_path" "${ESP_SR_MODEL_PATH}")
    list(APPEND BUILD_ARGS "--xiaozhi_fonts_path" "${XIAOZHI_FONTS_PATH}")

    add_custom_command(
        OUTPUT ${GENERATED_ASSETS_BIN}
        COMMAND python ${PROJECT_DIR}/scripts/build_default_assets.py ${BUILD_ARGS}
        DEPENDS ${SDKCONFIG} ${PROJECT_DIR}/scripts/build_default_assets.py
        COMMENT "Building default assets.bin based on configuration"
        VERBATIM
    )
    add_custom_target(generated_default_assets ALL DEPENDS ${GENERATED_ASSETS_BIN})
endfunction()
```

CMake 调用 Python 脚本（`build_default_assets.py`）把语音模型 + 字体 + emoji 打包成 `generated_assets.bin`，烧到 `assets` 分区。这是把”运行时静态数据”和”代码”分开的关键——固件更新换 `xiaozhi.bin`，资源更新换 `assets.bin`。

`add_custom_command` 的 `DEPENDS sdkconfig` 表示：sdkconfig 改了（比如换字体）会重新跑脚本，否则缓存。

10.2.5 第 5 段：分区表条件烧录（第 917–936 行）

```
partition_table_get_partition_info(size "--partition-name assets" "size")
partition_table_get_partition_info(offset "--partition-name assets" "offset")
if ("${size}" AND "${offset}")
    # v2 分区表有 assets 分区
    if(CONFIG_FLASH_DEFAULT_ASSETS)
        build_default_assets_bin()
        esptool_py_flash_to_partition(flash "assets" "${GENERATED_ASSETS_LOCAL_FILE}")
    elseif(CONFIG_FLASH_CUSTOM_ASSETS)
        # 用户提供自己的 assets.bin（本地路径或 URL）
        get_assets_local_file("${CONFIG_CUSTOM_ASSETS_FILE}" ASSETS_LOCAL_FILE)
        esptool_py_flash_to_partition(flash "assets" "${ASSETS_LOCAL_FILE}")
    elseif(CONFIG_FLASH_NONE_ASSETS)
        message(STATUS "Assets flashing disabled")
    endif()
else()
    # v1 分区表没 assets 分区
    message(STATUS "Assets partition not found, using v1 partition table")
endif()
```

3 个 Kconfig 分支：

1. `FLASH_DEFAULT_ASSETS`：跑 Python 脚本自动生成；
2. `FLASH_CUSTOM_ASSETS`：用户填 `CUSTOM_ASSETS_FILE` 路径或 URL，直接烧入；
3. `FLASH_NONE_ASSETS`：不烧（用户自己 `idf.py flash` 别的）。

`v1 分区表`（老板子）没有 assets 分区，自动跳过——向后兼容。

10.2.6 特殊板子：在线下载 emoji（第 771–806 行）

```
if(CONFIG_BOARD_TYPE_ESP_HI)
    set(URL "https://github.com/espressif2022/image_player/raw/main/test_apps/test_8bit")
    set(FILES_TO_DOWNLOAD "Anger_enter.aaf" "Anger_loop.aaf" ...)
    foreach(FILENAME IN LISTS FILES_TO_DOWNLOAD)
        if(EXISTS ${LOCAL_FILE})
            message(STATUS "File ${FILENAME} already exists, skipping")
        else()
            file(DOWNLOAD ${REMOTE_FILE} ${LOCAL_FILE} STATUS DOWNLOAD_STATUS)
            ...
        endif()
    endforeach()
endif()
```

`esp-hi` 这块板子的 EmoteEngine 资源（`.aaf` 动画文件，27 个）太大不放仓库里，构建时按需从 GitHub 拉。CMake `file(DOWNLOAD)` 是构建期下载，跟 OTA 完全无关。

10.3 `Kconfig.projbuild` —— 编译时配置入口

```
menu "Xiaozhi Assistant"
├── OTA_URL（字符串，默认 https://api.tenclass.net/xiaozhi/ota/）
├── FLASH_DEFAULT_ASSETS / FLASH_CUSTOM_ASSETS / FLASH_NONE_ASSETS
├── CUSTOM_ASSETS_FILE（字符串，自定义 assets 路径）
├── DEFAULT_LANGUAGE（40+ 语言可选）
├── BOARD_TYPE（110+ 板子可选）
├── 板子相关子选项（OLED 类型、4G 模块型号、麦克风类型等）
├── USE_HOTSPOT_WIFI_PROVISIONING（bool，默认 y）
├── USE_ESP_BLUFI_WIFI_PROVISIONING（bool，默认 n）
├── USE_ACOUSTIC_WIFI_PROVISIONING（bool，默认 n）
├── ENABLE_AUDIO_DEBUGGER（bool，默认 n）
├── AUDIO_DEBUG_SERVER_IP（字符串）
├── WAKE_WORD_TYPE（none / afe / esp_sr / custom）
└── ...
```

`menuconfig` 把这些做成 ncurses 菜单，开发者勾选 → 生成 `sdkconfig` 文件 → CMake 读 sdkconfig → 决定编译哪些代码。

10.3.1 Kconfig 设计模式

```
choice
    prompt "Board Type"
    default BOARD_TYPE_BREAD_COMPACT_WIFI

    config BOARD_TYPE_BREAD_COMPACT_WIFI
        bool "面包板 (WiFi)"
        depends on IDF_TARGET_ESP32S3
    config BOARD_TYPE_BREAD_COMPACT_ML307
        bool "面包板 (ML307 4G)"
        depends on IDF_TARGET_ESP32S3
    ...
endchoice
```

- `choice/endchoice` 块表示**互斥单选**；
- `depends on IDF_TARGET_ESP32S3` 表示**只有 ESP32-S3 目标芯片才能选**；
- 选中的会被 sdkconfig 设成 `CONFIG_BOARD_TYPE_BREAD_COMPACT_WIFI=y`，其它的 `=n`；
- CMakeLists 用 `if(CONFIG_BOARD_TYPE_BREAD_COMPACT_WIFI)` 取这个 flag。

10.3.2 条件子菜单

```
config OLED_SSD1306_128X32
    bool "SSD1306 128x32"
    depends on BOARD_TYPE_BREAD_COMPACT_WIFI || BOARD_TYPE_BREAD_COMPACT_ML307
```

只有选了面包板才会出现 OLED 子选项——避免菜单噪音。

10.3.3 唤醒词类型

```
choice
    prompt "Wake Word Type"
    default WAKE_WORD_AFE
    config WAKE_WORD_NONE
        bool "No Wake Word"
    config WAKE_WORD_AFE
        bool "AFE Wake Word (recommended)"
    config WAKE_WORD_ESP
        bool "ESP Wake Word (single-channel)"
    config WAKE_WORD_CUSTOM
        bool "Custom Wake Word (Sherpa-ONNX)"
endchoice
```

第 4 章讲过，4 种唤醒词类型在编译期挑一个——避免不必要的代码进入固件。

10.4 `partitions/` —— Flash 分区表

ESP32 的 Flash 是**按分区表布局**的，分区表本身也是个固定地址（0x8000）的小数据块。

10.4.1 v1 vs v2

v1（老版本，无 assets 分区）：

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,    0x4000,
otadata,  data, ota,     0xd000,    0x2000,
phy_init, data, phy,     0xf000,    0x1000,
ota_0,    app,  ota_0,   0x20000,   0x3f0000,
ota_1,    app,  ota_1,   ,          0x3f0000,
```

5 个分区：nvs（16KB） / otadata（8KB） / phy_init（4KB） / ota_0（约 4MB） / ota_1（约 4MB）。**资源数据要么编译进固件，要么用 SPIFFS 文件系统挂在 ota 分区某处**——但小智项目从 v1 升 v2 后改成专门的 assets 分区。

v2（新版本，含 assets 分区）——`partitions/v2/16m.csv`：

```
nvs,      data, nvs,     0x9000,    0x4000,
otadata,  data, ota,     0xd000,    0x2000,
phy_init, data, phy,     0xf000,    0x1000,
ota_0,    app,  ota_0,   0x20000,   0x3f0000,
ota_1,    app,  ota_1,   ,          0x3f0000,
assets,   data, spiffs,  0x800000,  8M
```

新增 8MB 的 `assets` 分区（SubType 标 `spiffs` 是为了让分区表工具识别成数据分区，实际上代码用 raw flash 访问，不挂 SPIFFS）。**第 7 章讲过 `Assets::Apply()` 用 `esp_partition_mmap` 直接把分区映射到 CPU 地址空间，零拷贝访问。**

10.4.2 不同容量的分区表

`partitions/v2/` 下：

文件	适用	OTA 分区大小	Assets 大小
`4m.csv`	4MB Flash（如 esp-hi）	factory only, ~2.5MB	1.5MB
`8m.csv`	8MB Flash	~3MB × 2	1MB
`16m.csv`	16MB Flash（主流）	~4MB × 2	8MB
`16m_c3.csv`	ESP32-C3 16MB	优化布局	8MB
`32m.csv`	32MB Flash（高端）	~12MB × 2	8MB

4MB Flash 的 `4m.csv` 没有 OTA 双分区——空间太紧，只用 `factory` 分区，**不能 OTA 升级**，必须 USB 重新烧录。

注意 `,  ,` 中间留空：CMake 工具会自动填上一个分区的 offset+size，所以 ota_1 的 offset 自动是 0x410000（0x20000+0x3f0000）。

10.4.3 切换分区表的方法

`sdkconfig` 里：

```
CONFIG_PARTITION_TABLE_CUSTOM=y
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions/v2/16m.csv"
```

由板子的 `sdkconfig.defaults` 默认设置或 `menuconfig` 手动改。

10.5 `idf_component.yml` —— 第三方组件依赖

```yaml
dependencies:
  espressif/cjson: "*"
  espressif/esp-sr: "==2.1.5"
  espressif/esp_lvgl_port: "==2.6.1"
  espressif/esp_codec_dev: "^1.0.0"
  espressif/button: "==4.1.4"
  espressif/esp_lcd_panel_io_additions: "==1.0.0"
  espressif/esp_audio_codec: "==2.3.0"
  78__xiaozhi-fonts: "==0.4.0"
  ...
```

`idf.py reconfigure` 时 ESP-IDF Component Manager 会：

1. 解析 yml；
2. 从 https://components.espressif.com 下载对应版本；
3. 解压到 `managed_components/<owner>__<name>_<version>/`；
4. 把它们加入构建（每个组件自己有 CMakeLists.txt）。

组件	角色
`cjson`	第 6 章 MCP JSON 解析、第 5 章协议 JSON
`esp-sr`	第 4 章 AFE / WakeNet / 语音模型
`esp_lvgl_port`	第 8 章 LCD/OLED 的 LVGL 集成
`esp_codec_dev`	音频 codec 抽象（ES8311 等）
`button`	第 9 章 `Button` 类的底层（`iot_button`）
`esp_audio_codec`	Opus 编解码器（第 4 章用到的 `opus_encoder` 等）
`xiaozhi-fonts`	自家维护的字体集合（普惠 / Font Awesome）

版本固定（`==`）：保证不同人编译出来一致；`^` 表示允许补丁版本变化。

10.6 `scripts/` —— 配套 Python 工具链

13 个 Python 脚本/目录，按用途分 5 类。

10.6.1 构建辅助

`build_default_assets.py`（883 行） —— 上面 10.2.4 提到的"打包 assets.bin"脚本。流程：

1. 解析 `sdkconfig` 找选了哪些 emoji / 字体 / 语言；
2. 跑 `pack_models()` 把 esp-sr 的 `wn9_nihaoxiaozhi.bin` 等模型按特定格式打包成 `srmodels.bin`；
3. 拷贝字体文件（`font_puhui_basic_14_1.bin` 等）到临时目录；
4. 跑 `gen_lang.py` 生成 `lang_config.h`（语言资源 C 头文件）；
5. 把 `srmodels.bin` + 字体 + emoji + `index.json` 用 `spiffs_assets_gen.py` 打成单一二进制；
6. 输出 `generated_assets.bin`。

关键技巧：`pack_models()` 内手写二进制格式（不用 SPIFFS），是因为 SPIFFS 有元数据开销，对只读资源直接平铺 `name + size + offset + data` 更省。第 7 章 `Assets::InitializePartition()` 解析的就是这个格式。

`gen_lang.py`（187 行） —— 把 `assets/locales/<lang>/language.json` 转成 `lang_config.h`：

```
namespace Lang {
    constexpr const char* CODE = "zh-CN";
    namespace Strings {
        constexpr std::string_view VOLUME = "音量: ";
        constexpr std::string_view MAX_VOLUME = "音量已最大";
        ...
    }
    namespace Sounds {
        constexpr std::string_view OGG_WIFICONFIG = "/wificonfig.ogg";
        ...
    }
}
```

`constexpr std::string_view`：

- 编译期常量，零运行时开销；
- 比 `const char*` 多个长度信息，避免 strlen；
- 字符串直接放 Flash（`.rodata`），不占 RAM。

fallback 机制：当前语言缺某个 key 时用 `en-US` 的——保证不会显示空字符串。

`versions.py` —— 处理 PROJECT_VER 的 bump 逻辑（patch / minor / major）。

10.6.2 发布与打包

`release.py`（300+ 行） —— 批量编译所有板子变体 + 打 zip 发布包。流程：

1. 读 `main/boards/<board>/config.json` 列出 `builds`（一个板子可能有多个变体，如不同语言、不同麦克风方向）；
2. 对每个变体：
    - 写 sdkconfig append 块（`CONFIG_BOARD_TYPE_xxx=y` + 变体专属 sdkconfig）；
    - `idf.py set-target esp32s3` → `idf.py -DBOARD_NAME=... build`；
    - `idf.py merge-bin` 合成 `merged-binary.bin`；
    - 压缩成 `releases/v1.5.0_<variant>.zip`；
3. 跳过已存在的 zip（增量构建）。

关键代码——10.2 节提到的 `_AUTO_SELECT_RULES`：

```
_AUTO_SELECT_RULES = {
    "CONFIG_USE_ESP_BLUFI_WIFI_PROVISIONING": [
        "CONFIG_BT_ENABLED=y",
        "CONFIG_BT_BLUEDROID_ENABLED=y",
        "CONFIG_BT_BLE_42_FEATURES_SUPPORTED=y",
        "CONFIG_BT_BLE_50_FEATURES_SUPPORTED=n",
        "CONFIG_BT_BLE_BLUFI_ENABLE=y",
        "CONFIG_MBEDTLS_DHM_C=y",
    ],
}
```

坑：Kconfig 的 `select` 关键字只在 `menuconfig` 交互式选择时自动应用依赖。`release.py` 是非交互式的——直接 append `CONFIG_USE_ESP_BLUFI=y` 不会自动开启它依赖的蓝牙栈。所以脚本里手动写一份依赖映射，保证 BluFi 配网编译时蓝牙栈也开。

`download_github_runs.py` —— 从 GitHub Actions 拉构建产物，方便 CI 协作。

10.6.3 资源转换

`p3_tools/`（音频 P3 格式工具）：

- `convert_audio_to_p3.py`：MP3/WAV → 自定义 P3 格式（Opus + 头）；
- `convert_p3_to_audio.py`：反向解码；
- `play_p3.py`：本地播放 P3 文件验证；
- `batch_convert_gui.py`：拖拽式 GUI 批量转换。

P3 格式（10.6.3 用到的）：

> 每帧：`[1B type=0, 1B reserved=0, 2B opus_len BE] [Opus payload...]`

`type=0` 是普通音频帧，预留 type 字段可以扩展（控制命令、静音帧、关键帧等）。第 5 章 `BinaryProtocol2` / `BinaryProtocol3` 的雏形就是这个。

编码细节 —— `convert_audio_to_p3.py`：

1. `librosa.load` 加载任意格式音频；
2. 立体声 → 单声道（`librosa.to_mono`）；
3. 可选 LUFS 响度归一化（`pyloudnorm`）—— 让所有提示音音量一致；
4. 重采样到 16kHz（`librosa.resample`）；
5. `opuslib.Encoder` 编码 60ms 帧；
6. 按 P3 格式写文件。

LUFS（Loudness Units Full Scale）：广电级响度标准，比简单的 RMS 准。−16 LUFS 是流媒体常见值（YouTube −14 / Spotify −14 / Apple Music −16）。

`ogg_converter/xiaozhi_ogg_converter.py` —— MP3 → OGG（用于第 4 章 `PlaySound()` 播放的提示音）。OGG 容器内 Opus 编码，跟设备解码栈一致。

`mp3_to_ogg.sh` —— 一行 `ffmpeg` shell 包装。

`Image_Converter/` —— PNG/JPG → C 字节数组（编进固件）。

10.6.4 测试与调试

`audio_debug_server.py` —— UDP 调试服务端（第 4 章 `AudioDebugger` 用到）：

```python
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(('0.0.0.0', 8000))
wav_file = wave.open(f"{samplerate}_{channels}.wav", "wb")
wav_file.setsampwidth(2)
wav_file.setframerate(samplerate)

while True:
    message, address = server_socket.recvfrom(8000)
    wav_file.writeframes(message)   # PCM 直接写 WAV
```

设备开了 `ENABLE_AUDIO_DEBUGGER` 后，每个 audio frame 通过 UDP 发到 PC 8000 端口，PC 直接存成 WAV——查 AFE 处理效果、回声消除好不好、噪声大不大。

`acoustic_check/` —— 声学验证工具。

`sonic_wifi_config.html` —— **声波配网网页端**。打开 HTML 输 SSID + 密码 → 网页用 Web Audio API 合成 AFSK 调制波形通过手机喇叭播放 → ESP32 麦克风听到 → 解调拿到凭证。零依赖配网神器。

10.6.5 资源打包

`spiffs_assets/`：

- `build.py`：单板的 assets 打包；
- `build_all.py`：批量；
- `pack_model.py`：esp-sr 语音模型打包逻辑；
- `spiffs_assets_gen.py`：主入口，被 `build_default_assets.py` import。

10.7 `docs/` —— 设计文档与接线图

```
docs/
├── blufi.md            BluFi 蓝牙配网协议说明
├── custom-board.md     怎么添加新板子（步骤教程）
├── mcp-based-graph.jpg MCP 整体架构图（架构图）
├── mcp-protocol.md     MCP 协议详解（设计文档）
├── mcp-usage.md        MCP 工具使用指南
├── mqtt-udp.md         MQTT+UDP 协议详解（第 5 章基础）
├── websocket.md        WebSocket 协议详解（第 5 章基础）
├── v0/                 老版本板子的接线图（jpg）
└── v1/                 新版本板子的接线图 + 渲染图
```

`v0/` 9 张图，`v1/` 18 张——大部分板子的实物接线图，方便用户自己焊。

`mcp-protocol.md` 和 `mcp-usage.md` 是第 6 章实现的"协议说明书"——理解协议先读 doc 再读代码会容易很多。

`websocket.md` / `mqtt-udp.md` 是第 5 章协议的对端文档（服务器侧也要照着实现）。

`custom-board.md` 是第 9 章 9.11 的官方教程，比本章详细。

10.8 `sdkconfig.defaults` 系列

项目根有多个 `sdkconfig.defaults` 文件：

```
sdkconfig.defaults                # 通用默认
sdkconfig.defaults.esp32          # 普通 ESP32 默认（4MB 分区表等）
sdkconfig.defaults.esp32c3        # C3 默认
sdkconfig.defaults.esp32c5        # C5 默认
sdkconfig.defaults.esp32c6        # C6 默认
sdkconfig.defaults.esp32p4        # P4 默认（PSRAM 大、双核高频）
sdkconfig.defaults.esp32s3        # S3 默认（最常用，PSRAM Octal）
```

ESP-IDF 启动构建时按当前 IDF_TARGET 自动合并对应文件到初始 sdkconfig：

```
idf.py set-target esp32s3
# 此时项目根的 sdkconfig 会从 sdkconfig.defaults + sdkconfig.defaults.esp32s3 合并生成
```

典型内容（s3）：

```
CONFIG_ESPTOOLPY_FLASHSIZE_16MB=y
CONFIG_SPIRAM=y
CONFIG_SPIRAM_MODE_OCT=y
CONFIG_SPIRAM_SPEED_80M=y
CONFIG_FREERTOS_UNICORE=n
CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240=y
CONFIG_FREERTOS_HZ=1000
CONFIG_PARTITION_TABLE_CUSTOM=y
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions/v2/16m.csv"
CONFIG_LWIP_MAX_SOCKETS=10
...
```

这些都是性能/功能基线——`menuconfig` 选板子时可以再覆盖（如某些小内存板子改 8m 分区表）。

10.9 完整构建流程串起来

```
# Step 1: 选目标芯片（写 sdkconfig 默认值）
idf.py set-target esp32s3

# Step 2: 配置（菜单选板子、语言、唤醒词、配网方式等）
idf.py menuconfig
# 选了 BOARD_TYPE_ESP_BOX_3 + LANGUAGE_ZH_CN + WAKE_WORD_AFE

# Step 3: 构建
idf.py build
# 内部依次：
#   1. 检查 idf_component.yml，下载缺失组件到 managed_components/
#   2. CMake configure：读 sdkconfig，根据 BOARD_TYPE 派发，准备源文件列表
#   3. CMake generate：生成 ninja/make 文件
#   4. ninja 增量编译所有源文件
#   5. 链接 ELF（xiaozhi.elf）
#   6. esptool elf2image 生成 xiaozhi.bin
#   7. 生成 partition-table.bin（按 16m.csv）
#   8. 跑 build_default_assets.py 生成 generated_assets.bin
#   9. 用 esptool merge-bin 把上面 4 个合成 merged-binary.bin

# Step 4: 烧录（首次完整刷）
idf.py flash
# 烧入：bootloader / partition-table / xiaozhi.bin / generated_assets.bin

# Step 5: 监视串口
idf.py monitor
# 按 Ctrl+] 退出

# 一条命令搞定后三步：
idf.py build flash monitor
```

后续小修改（改代码）→ `idf.py build flash monitor`，CMake 增量编译，几十秒就好。

10.10 本章用到的关键技术

技术	应用
CMake `if-elseif` 派发	110 板子选 1
`file(GLOB ...)`	boards/common 自动包含
`add_custom_command + DEPENDS`	sdkconfig 变 → 重跑 assets 打包
`file(DOWNLOAD)` 构建期下载	esp-hi 板拉 .aaf 资源
`find_component_by_pattern`	managed_components 路径不固定
`esptool_py_flash_to_partition`	自定义分区烧录命令
`partition_table_get_partition_info`	检测分区是否存在
Kconfig `choice/endchoice`	互斥单选
Kconfig `depends on`	条件可见性
`sdkconfig.defaults.<target>`	按芯片自动合并默认配置
`idf_component.yml` + Component Manager	第三方组件版本固定
`constexpr std::string_view`	Lang::Strings 零开销字符串
手写紧凑二进制资源格式	assets.bin 比 SPIFFS 省空间
LUFS 响度归一化	提示音音量一致
AFSK 声波编解调（Web Audio + 麦克风）	零依赖配网
`_AUTO_SELECT_RULES` 手动依赖映射	绕过 Kconfig `select` 在非交互场景失效的问题

10.11 看完本章你应该掌握的

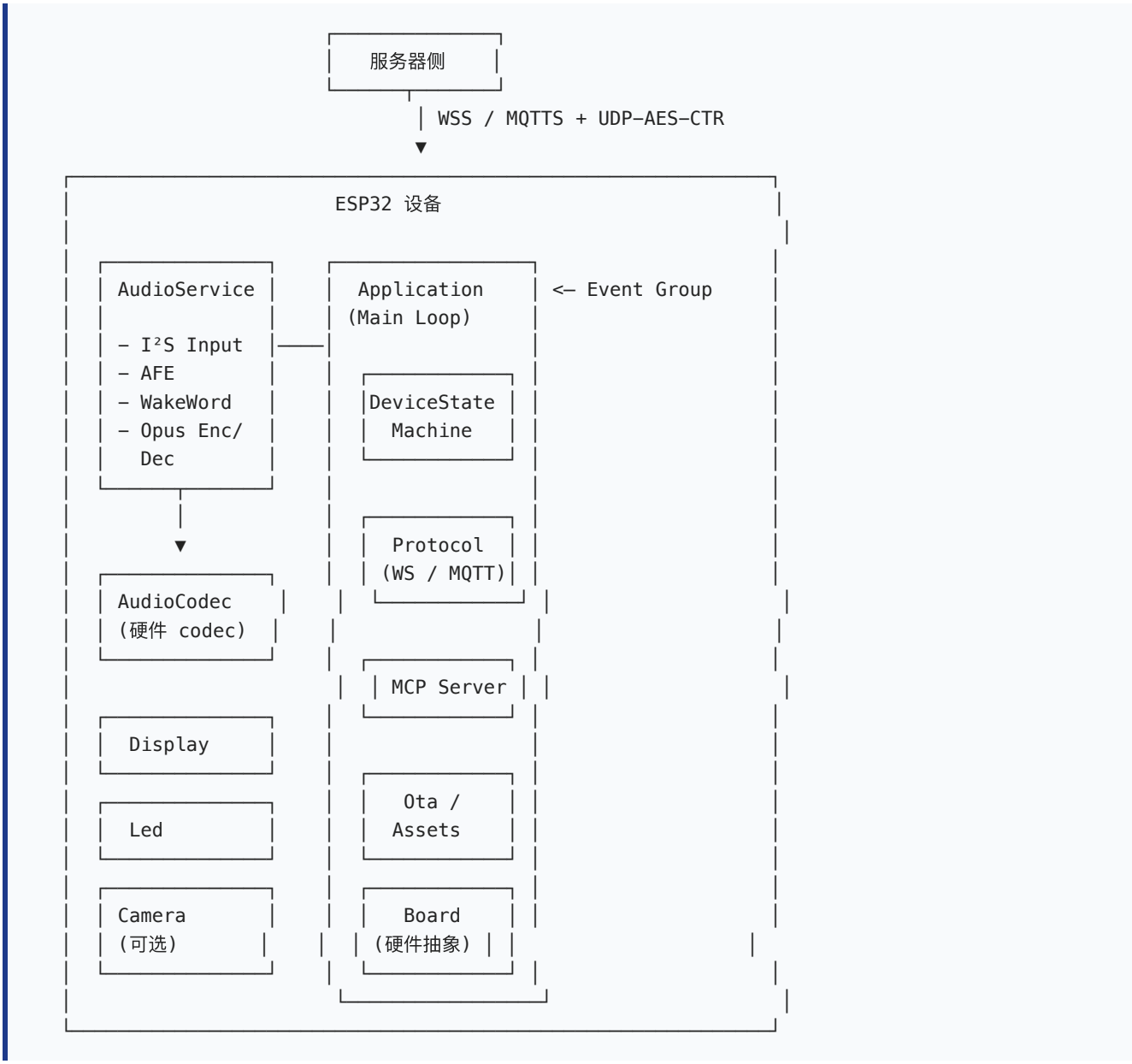
- 从 `idf.py build` 到 `merged-binary.bin` 的完整流程
- `main/CMakeLists.txt` 里 110 个板子怎么用 `if-elseif` 派发
- `Kconfig.projbuild` 怎么生成 `menuconfig` 菜单
- v1 vs v2 分区表的区别（assets 分区）
- 不同 Flash 容量（4M/8M/16M/32M）的分区布局差异
- `build_default_assets.py` 怎么把语音模型+字体+emoji+语言资源打成一个 .bin
- `release.py` 怎么批量编译所有板子变体并自动处理依赖
- `gen_lang.py` 怎么把 JSON 翻译表转成 `constexpr` C 头
- `audio_debug_server.py` 怎么收 ESP32 UDP 调试数据并存 WAV
- `p3_tools/` 转换音频到 Opus P3 格式的关键步骤
- `idf_component.yml` 怎么管理第三方组件版本
- `sdkconfig.defaults.<target>` 的合并机制

终章 全文回顾

至此 10 章覆盖完整个项目：

章节	主题	关键文件
1	项目总览 + 启动流程 + 全文件分级表	—
2	主调度器：Application	main.cc / application.cc/h
3	设备状态机	device_state_machine.cc/h / device_state.h
4	音频子系统：AFE / 唤醒词 / Opus	audio/*
5	网络协议：WebSocket / MQTT+UDP	protocols/*
6	MCP 工具调用协议	mcp_server.cc/h
7	系统服务：OTA / Assets / Settings	ota.cc/h, assets.cc/h, settings.cc/h, system_info.cc/h
8	显示与指示	display/, *led/*
9	板级抽象	boards/*
10	构建系统 + 工具链	CMakeLists.txt, Kconfig, partitions/, scripts/

核心架构再回顾



项目的几个核心设计思想

1. **状态机驱动**：所有用户交互都从 `DeviceState` 转换中得到响应。
2. **事件组解耦**：FreeRTOS EventGroup 让任务间通信无锁、可超时。
3. **抽象层透明**：NetworkInterface / Protocol / AudioCodec / Display / Led 5 大抽象让上层完全不感知硬件/网络差异。
4. **资源外部化**：assets 分区让语音模型 / 字体 / emoji / 语言资源跟代码分离，可独立 OTA。
5. **MCP 工具开放**：通过 JSON-RPC 让 LLM 直接调设备能力，不用写专有 API。
6. **离线唤醒 + 流式 ASR**：唤醒词本地跑省功耗，识别上云保准确率。
7. **双协议并存**：WebSocket 用户网络好用；MQTT+UDP 移动网络/IoT 场景更稳。
8. **配网三选一**：Hotspot / BLE / 声波，照顾各种用户的网络环境。
9. **A/B OTA + Anti-bricking**：ota_0 / ota_1 双分区，刷坏自动回滚。
10. **编译期板子选择**：110 板共享代码，CMake `if-elseif` 派发零运行时开销。

读到这里，您应该可以：

- 独立拿一块 ESP32-S3 + 屏幕 + I²S 麦克风跑起这个项目；
- 添加自己的 MCP 工具（控制空调、查询天气、播报新闻……）；

- 用同一服务器对接的话，把这套设备端代码移植到非 ESP32 平台也只是替换硬件抽象层；
- 自己添加新板子（按 9.11 步骤）；
- 看懂任意 commit 改动跟哪一章相关。

全文完。