ESP32-P4

ESP-VISION 编程指南

Release 2026.11.14-2-gace7ca10bd
乐鑫信息科技
2026 年 11 月 14 日

目录

目录 i

1 简介 3
1.1 什么是 ESP-VISION . . . 3
1.2 快速体验 . . . 3
1.3 主要特性 . . . 3
1.4 支持的开发板 . . . 3

2 芯片与开发板支持 5
2.1 支持的开发板 . . . 5
2.2 芯片能力与板级能力 . . . 7
2.3 标准 MicroPython 功能 . . . 7
2.4 WebREPL 与 EV-MUX 网络控制 . . . 7

3 快速入门 9
3.1 前置条件 . . . 9
3.2 构建、烧录与监视 . . . 9
3.3 常用 idf.py 命令 . . . 10
3.4 仅烧录应用 . . . 10
3.5 运行第一个脚本 . . . 10

4 基础概念 11
4.1 MicroPython 语言基础 . . . 11
4.1.1 执行模型 . . . 11
4.1.2 基本语法 . . . 12
4.1.3 主要数据结构 . . . 13
4.1.4 面向微控制器的设计理念 . . . 14
4.1.5 MicroPython 与 CPython . . . 14
4.2 软件包与部署 . . . 15
4.2.1 模块与软件包结构 . . . 15
4.2.2 当前固件的软件包管理 . . . 15
4.2.3 文件系统中的软件包 . . . 15
4.2.4 嵌入固件的软件包 . . . 16
4.2.5 选择部署方式 . . . 16
4.2.6 导入优先级与版本管理 . . . 17
4.3 图像模型 . . . 17
4.3.1 像素格式 . . . 17
4.3.2 色彩空间 . . . 18
4.3.3 帧缓冲与 `fb_alloc` . . . 18
4.3.4 感兴趣区域（ROI） . . . 18
4.4 图像处理 . . . 18
4.4.1 典型处理流水线 . . . 18
4.4.2 阈值化与分割 . . . 19
4.4.3 邻域（卷积）滤波 . . . 19
4.4.4 秩滤波与保边滤波 . . . 20
4.4.5 形态学 . . . 20
4.4.6 直方图与统计 . . . 20

4.4.7 特征检测 . . . 20
4.5 AI 推理 . . . 20
4.5.1 模型运行时 . . . 21
4.5.2 模型文件与量化 . . . 21
4.5.3 推理流程 . . . 21
4.5.4 预处理 . . . 22
4.5.5 后处理 . . . 22
4.5.6 内存与性能 . . . 23
4.6 摄像头流水线 . . . 23
4.6.1 从传感器到图像 . . . 24
4.6.2 各开发板后端 . . . 24
4.6.3 分辨率 . . . 25
4.7 编解码与推流 . . . 25
4.7.1 静态图像编码：JPEG 与 PNG . . . 25
4.7.2 录制序列：ImageIO . . . 25
4.7.3 视频编码：H.264 . . . 25
4.7.4 网络推流：RTSP . . . 25
4.7.5 主机预览：USB CDC . . . 26
4.8 启动流程 . . . 26
4.8.1 硬复位与软复位 . . . 26
4.8.2 ESP-VISION 启动顺序 . . . 27
4.8.3 首次启动与 Flash 文件系统 . . . 29
4.8.4 使用 boot.py . . . 29
4.8.5 使用 main.py . . . 29
4.8.6 REPL 与故障恢复 . . . 30

5 存储、文件系统与数据保护 31
5.1 存储介质与文件系统所有权 . . . 31
5.2 可选卷与默认自动挂载 . . . 32
5.3 两个卷在线时的 USB 行为 . . . 32
5.4 故障原则：保留数据，保持控制通道可用 . . . 33
5.5 单写入方、文件发布与断电 . . . 33
5.6 备份与恢复规范 . . . 34
5.7 显式恢复未挂载的 Flash 卷 . . . 34
5.8 固件安装与验收 . . . 35

6 API 参考 37
6.1 MicroPython API . . . 37
6.1.1 文档基线 . . . 37
6.1.2 语言与内置 API . . . 37
6.1.3 标准库与微型库 . . . 38
6.1.4 JSON 与时间示例 . . . 38
6.1.5 硬件与 ESP32 API . . . 38
6.1.6 GPIO 与 PWM 示例 . . . 38
6.1.7 网络 API . . . 38
6.1.8 连接 Wi-Fi . . . 39
6.1.9 文件系统与运行时 API . . . 39
6.1.10 列出存储并检查内存 . . . 39
6.1.11 组织复杂应用 . . . 39
6.1.12 线程与原生任务 . . . 41
6.1.13 检查固件中的可用性 . . . 42
6.2 sensor –摄像头 . . . 42
6.2.1 摄像头初始化与连续采集 . . . 42
6.2.2 图像方向与摄像头状态 . . . 42
6.2.3 暂时停止采集 . . . 43
6.2.4 运行时控制 IPA . . . 43
6.2.5 读取 IPA 统计信息 . . . 44
6.2.6 手动控制 ISP . . . 44

6.2.7 Constants . . . 46
6.2.8 Functions . . . 46
6.2.9 Classes . . . 48
6.3 image –图像处理 . . . 48
6.3.1 绘图与颜色色块追踪 . . . 48
6.3.2 滤波与二值分割 . . . 49
6.3.3 二维码识别 . . . 49
6.3.4 直线、线段与圆检测 . . . 49
6.3.5 编码并保存图像 . . . 49
6.3.6 Constants . . . 50
6.3.7 Functions . . . 52
6.3.8 Classes . . . 53
6.4 display –LCD 显示 . . . 65
6.4.1 摄像头预览 . . . 65
6.4.2 复用 Image 绘制内容 . . . 65
6.4.3 定位、缩放与裁剪 . . . 66
6.4.4 背光与资源释放 . . . 66
6.4.5 Classes . . . 66
6.5 lcd_touch –触摸输入 . . . 67
6.5.1 读取触摸坐标 . . . 67
6.5.2 所有权与资源释放 . . . 68
6.5.3 Classes . . . 68
6.6 espdl –模型推理 . . . 68
6.6.1 按键确认单目标追踪 . . . 68
6.6.2 原始输出 tensor . . . 70
6.6.3 目标检测 . . . 71
6.6.4 PicoDet . . . 71
6.6.5 PedestrianDetect . . . 71
6.6.6 图像分类 . . . 71
6.6.7 姿态估计 . . . 72
6.6.8 运行时调整阈值 . . . 72
6.6.9 结果元组 . . . 72
6.6.10 Functions . . . 72
6.6.11 Classes . . . 73
6.7 tflite –模型推理 . . . 76
6.7.1 基本用法 . . . 76
6.7.2 Callable 输入 . . . 76
6.7.3 后处理 . . . 76
6.7.4 Classes . . . 76
6.8 image.ImageIO –图像流 . . . 77
6.8.1 录制到存储 . . . 77
6.8.2 回放已录制的图像流 . . . 77
6.8.3 使用内存图像流 . . . 78
6.8.4 Constants . . . 78
6.8.5 Classes . . . 78
6.9 h264 –H.264 编码 . . . 79
6.9.1 编码单帧图像 . . . 79
6.9.2 录制 H.264 裸码流 . . . 79
6.9.3 资源与吞吐量注意事项 . . . 80
6.9.4 Classes . . . 80
6.10 rtsp –RTSP 推流 . . . 81
6.10.1 采集、编码与推流 . . . 81
6.10.2 客户端与队列行为 . . . 82
6.10.3 停止服务 . . . 82
6.10.4 Classes . . . 82
6.11 uvc —USB 摄像头输出 . . . 82
6.12 storage –存储诊断与确认恢复 . . . 83
6.12.1 Functions . . . 83

7 操作指南 **85**
7.1 用 AI 开发视觉应用 . . . 85
7.1.1 按固件版本查询知识 . . . 85
7.1.2 接入你的 AI 助手 . . . 85
7.1.3 接入开发板并运行应用 . . . 86
7.1.4 日常任务 . . . 87
7.1.5 通过对话改进效果 . . . 89
7.1.6 对话背后的工具 . . . 89
7.2 添加新的 Python 模块 . . . 89
7.2.1 总览 . . . 89
7.2.2 1. 创建绑定源文件 . . . 90
7.2.3 2. 按需注册 qstr . . . 90
7.2.4 3. 将源文件接入构建 . . . 90
7.2.5 4. 添加类型存根 . . . 90
7.2.6 5. 编写模块文档 . . . 91
7.2.7 6. 构建并验证 . . . 91
7.3 贡献与审阅 . . . 91
7.3.1 同步与合并流程 . . . 91
7.4 客制化固件功能 . . . 91
7.4.1 确定客制化范围 . . . 91
7.4.2 客制化 ESP-VISION Python 模块 . . . 92
7.4.3 客制化图像算法 . . . 92
7.4.4 客制化标准 MicroPython 功能 . . . 92
7.4.5 配置内存放置策略 . . . 92
7.4.6 客制化冻结 Python 代码 . . . 93
7.4.7 客制化板级服务与可选组件 . . . 93
7.4.8 构建与验证 . . . 93
7.5 发布合格固件 . . . 93
7.5.1 发布包与 URL 契约 . . . 93
7.5.2 不可变清单与当前入口 . . . 94
7.5.3 构建导出的 Wi-Fi 能力 . . . 94
7.5.4 可选单 BIN 布局 . . . 94
7.5.5 MCP 与发布资格 . . . 94
7.5.6 发布流程 . . . 95
7.5.7 Iris 固件文件 . . . 95
7.6 引入新的模型 . . . 98
7.6.1 1. 获取或转换模型 . . . 98
7.6.2 2. 将模型拷贝到板级存储 . . . 98
7.6.3 3. 选择合适的 API . . . 98
7.6.4 4. 运行 ESP-DL 推理 . . . 99
7.6.5 5. 在 Python 中解码 ESP-DL 输出 . . . 99
7.6.6 6. 运行 TFLite Micro 推理 . . . 99
7.6.7 7. 可选：性能分析和验证 . . . 100
7.7 训练 ESPDet Pico 模型 . . . 100
7.7.1 1. 准备 ESP-Detection 环境 . . . 100
7.7.2 2. 准备数据集 . . . 101
7.7.3 3. 训练 ESPDet Pico . . . 101
7.7.4 4. 准备校准数据 . . . 102
7.7.5 5. 量化并导出.espdl . . . 102
7.7.6 6. 添加模型元数据 . . . 103
7.7.7 7. 拷贝模型到板端存储 . . . 103
7.7.8 8. 在 ESP-VISION 上运行 . . . 104
7.7.9 9. 验证和调参 . . . 104
7.8 添加新的开发板 . . . 105
7.8.1 MicroPython 移植侧 . . . 105
7.8.2 ESP-VISION 侧 . . . 105
7.8.3 构建与烧录 . . . 105
7.9 EV-MUX v4 主机适配入口 . . . 105

7.10 Agent 代码生成指南 . . . 106
7.10.1 依赖规则 . . . 106
7.10.2 All-in-One 输出 . . . 106
7.10.3 设备通信与闭环验收 . . . 106

8 方案架构 107
8.1 分层概览 . . . 107
8.2 采集到输出的数据流 . . . 109
8.3 通信子系统 . . . 109
8.4 源码结构 . . . 110
8.5 板卡的组成 . . . 110
8.6 随芯片变化的源码 . . . 110
8.7 MicroPython Overlay . . . 110

9 通信系统与 EV-MUX 113
9.1 一张图看懂整条链路 . . . 114
9.2 推荐阅读顺序 . . . 114
9.2.1 架构、执行上下文与标准输出 . . . 114
9.2.2 物理接口、连接代次与构建配置 . . . 122
9.2.3 USB 接口、配置与磁盘命名 . . . 126
9.2.4 会话所有权、接管与恢复 . . . 128
9.2.5 帧格式、逻辑通道与 RPC 协议 . . . 133
9.2.6 交互式 Python、脚本与文件操作 . . . 139
9.2.7 预览、调试抓帧、背压与统计 . . . 146
9.2.8 主机接入、Wi-Fi 配置与恢复实践 . . . 151
9.2.9 局域网发现协议与板级默认策略 . . . 157
9.2.10 Device Core 接入指南 . . . 160
9.2.11 通信验收、闭环测试与故障定位 . . . 160
9.3 六个必须分开的概念 . . . 167
9.4 当前实现边界 . . . 167

10 Device Core 主机 SDK 169
10.1 应用 API 完整参考 . . . 169
10.1.1 创建对象与字节传输 . . . 170
10.1.2 device.discover(options) . . . 171
10.1.3 device.connect(options) . . . 171
10.1.4 device.disconnect() . . . 171
10.1.5 device.inspect(request) . . . 171
10.1.6 device.configure(request) . . . 172
10.1.7 device.control(request) . . . 172
10.1.8 device.python.input(data) . . . 173
10.1.9 device.python.execute(request) . . . 173
10.1.10 设备文件操作 . . . 173
10.1.11 device.capture(request) . . . 174
10.1.12 device.on(event, handler) . . . 175
10.1.13 互斥、取消与错误 . . . 175
10.1.14 固件检查与更新 . . . 175
10.1.15 列出并指定固件版本 . . . 177
10.1.16 不可变发布快照与单 BIN 升级（0.4.0） . . . 178
10.2 高级导出 API 完整参考 . . . 178
10.2.1 @esp-vision/device-core . . . 179
10.2.2 @esp-vision/device-core/evmux . . . 179
10.2.3 @esp-vision/device-core/evrepl . . . 179
10.2.4 @esp-vision/device-core/session . . . 181
10.2.5 @esp-vision/device-core/actions . . . 184
10.2.6 @esp-vision/device-core/bindings . . . 196
10.2.7 @esp-vision/device-core/firmware . . . 197
10.2.8 @esp-vision/device-core/node . . . 199
10.2.9 @esp-vision/device-core/discovery . . . 199

10.2.10 @esp-vision/device-core/discovery/node . . . 200
10.2.11 使用边界 . . . 200
10.2.12 FirmwareFlashSession.flash 返回值 . . . 200
10.2.13 @esp-vision/device-core/browser . . . 200
10.2.14 @esp-vision/device-core/firmware 发布读取器（0.5.1） . . . 201
10.2.15 浏览器烧录后的启动 . . . 201
10.3 Device Core npm 包对接 . . . 201
10.3.1 源码、版本与安装 . . . 202
10.3.2 应用、Core 与传输的职责 . . . 202
10.3.3 独立 Node TCP 示例 . . . 202
10.3.4 浏览器 Web Serial 示例 . . . 204
10.3.5 文件与交互式 Python . . . 205
10.3.6 Wi-Fi 配置与状态 . . . 205
10.3.7 发现与身份核验 . . . 206
10.3.8 预览、日志与抓帧 . . . 207
10.3.9 复位、烧写与发布 . . . 207
10.3.10 应用层固件更新入口 . . . 207
10.3.11 浏览器 ROM 烧录 . . . 208
10.4 设备管理命令行 . . . 208
10.4.1 安装与宿主机要求 . . . 208
10.4.2 发现与检查 . . . 209
10.4.3 网络配置、控制与固件 . . . 209
10.4.4 Python 执行 . . . 210
10.4.5 设备文件与图像采集 . . . 210
10.4.6 批处理与 watch . . . 210
10.4.7 输出、超时与取消 . . . 211
10.5 安装与兼容性 . . . 211
10.6 架构与使用入口 . . . 211
10.7 许可证 . . . 213

11 项目关系 **215**
11.1 ESP-VISION、MicroPython 与 OpenMV . . . 215
11.2 依赖层次 . . . 215

12 许可证 **217**
12.1 许可证清单 . . . 217
12.2 如何理解许可证 . . . 219
12.3 OpenMV 与 GPL 代码路径 . . . 219
12.4 增加第三方代码 . . . 219

Python 模块索引 **221**

Python 模块索引 **221**

索引 **223**

索引 **223**

ESP-VISION 是面向乐鑫 SoC 的低代码端侧 AI 与计算机视觉框架，深度整合了摄像头采集、图像处理、视频编解码、网络传输、模型部署和 AI 推理等核心能力，并提供统一标准化的 Python 接口，赋能开发者快速构建集视觉采集、智能识别、画面显示与流媒体传输于一体的边缘应用。

本指南的章节结构如下。

Chapter 1

简介

1.1 什么是 ESP-VISION

ESP-VISION 是面向乐鑫 SoC 的低代码端侧 AI 与计算机视觉框架，深度整合了摄像头采集、图像处理、视频编解码、网络传输、模型部署和 AI 推理等核心能力，并提供统一标准化的 Python 接口，赋能开发者快速构建集视觉采集、智能识别、画面显示与流媒体传输于一体的边缘应用。

1.2 快速体验

访问 ESP-VISION 官网 可快速获取 Web IDE、MCP 配置资源、示例和相关项目资源。

1.3 主要特性

- 为受支持的芯片和开发板统一提供摄像头、图像、显示、视频编码、预览与推流 API。
- 提供绘图、滤波、颜色追踪、特征检测、二维码、条码和 AprilTag 等图像处理能力。
- 基于 ESP-DL 提供目标检测、姿态估计和图像分类能力，同时支持通过 TensorFlow Lite Micro 运行 `.tflite` 模型。
- 底层高效的 C/C++ 基础组件深度协同芯片的多媒体外设与硬件加速模块，切实保障应用的高效与实时运行性能。
- 可通过 VSCode 主机工具或 Web IDE 进行开发，并使用 `idf.py` 管理固件构建。

1.4 支持的开发板

ESP-VISION 支持基于 ESP32-P4、ESP32-S3 与 ESP32-S31 的开发板。完整开发板列表以及各芯片的模块和限制见芯片与开发板支持。

构建与烧录固件请参阅快速入门，整体架构请参阅方案架构。

Chapter 2

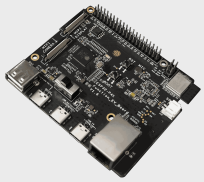
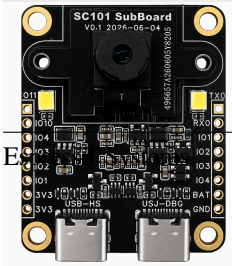
芯片与开发板支持

ESP-VISION 文档按支持的芯片分别构建。请选择开发板固件所使用的芯片，使文档中的模块和 API 与默认固件保持一致。

2.1 支持的开发板

备注： 当前标准 ESP32-P4 预编译固件支持芯片版本 v3.1–v3.99。这里指 esptool 报告的芯片版本，不是开发板或模组丝印版本。烧录前须核对所选镜像的实际版本范围；仅凭板名或笼统的 v3.x 标记不足以确认兼容。

此处不提供 ESP32-P4 v1.3 的预编译固件。有需要的用户须自行编译并验证：选择对应板型，在 menuconfig 中启用 `CONFIG_ESP32P4_SELECTS_REV_LESS_V3`，选择与实际芯片兼容的最低版本，并重新编译 bootloader 和应用。IDF 的 v3 之前版本与 v3.x 目标互不兼容，不能强刷 v3.x 镜像或修改镜像版本头绕过检查。

图片	开发板	芯片	ESP-VISION 支持情况
ESP-VISION	ESP32-S31-CHATBOT	ESP32-S3	支持 | 模块: sensor, image, display, lcd_touch, espdl, tflite, imageio, storage | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | 视觉能力: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-S31-Korvo-1	ESP32-S3	支持 | 模块: sensor, image, display, lcd_touch, espdl, tflite, imageio, uvc, storage | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | 视觉能力: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-S31-MOSAICO	ESP32-S3	支持 | 模块: sensor, image, display, lcd_touch, espdl, tflite, imageio, uvc, storage | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | 视觉能力: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-EYE	ESP32-P4	支持 | 模块: sensor, image, display, lcd_touch, espdl, tflite, imageio, h264, rtsp, uvc, storage, barcode | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | 视觉能力: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-Function-EV-Board	ESP32-P4	支持 | 模块: sensor, image, display, lcd_touch, espdl, tflite, imageio, h264, rtsp, uvc, storage, barcode | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | 视觉能力: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-	ESP32-P4	支持 | 模块: sensor, image, display,

备注： 支持摘要根据 `mpconfigport.h`、各开发板的 `mpconfigboard.h` 与 `imlib_config.h`，以及芯片相关的模块选择自动生成。

2.2 芯片能力与板级能力

所选芯片控制芯片级源文件选择。例如，只有 `IDF_TARGET` 为 `esp32p4` 时，`micropython.cmake` 才会加入 `h264` 和 `rtsp`。板级配置还可以通过 `boards/<BOARD>/board.cmake` 进一步启用或关闭 ZXing-C++ 条形码后端等功能。

API 导航和芯片相关的符号条件会根据这些构建文件生成。因此，`micropython.cmake`、`boards/*/port/mpconfigboard.cmake` 和各开发板的 `board.cmake` 是 ESP-VISION API 可用性的权威来源。

2.3 标准 MicroPython 功能

标准 MicroPython 功能采用另一条配置路径：全局 `overlay/micropython/ports/esp32/mpconfigport.h` 与各开发板的 `mpconfigboard.h`。上表会索引这些文件中显式启用的基础能力，其他条件还可能取决于芯片能力和固件配置。

当前开发板头文件也都关闭了 Bluetooth 和 ESP-NOW；S31 开发板还关闭了 `machine.ADC` 和 `machine.ADCBlock`。开发板仍需具备相应网络硬件和固件配置，`network.WLAN` 才能建立连接。

这些条件并非只由芯片决定，因此芯片选择器会过滤 ESP-VISION API 参考，但不代表标准 MicroPython 模块的完整清单。构建环境兼容性说明统一维护在项目 README 中。

芯片文档描述当前开发板配置共同采用的默认能力。自定义开发板或修改编译选项后的固件可能与已发布芯片页面不同；发生差异时应以固件配置为准。

2.4 WebREPL 与 EV-MUX 网络控制

当前 ESP-VISION EV-MUX 运行态不支持传统 MicroPython WebREPL，ESP-IDF v5.5.6 构建也不例外。部分构建启用了 WebSocket 与内部 `_webrepl` 组件。ESP32_P4X_EYE 等部分板型未包含上游 `webrepl` Python 服务，ESP32_S3_EYE 的 manifest 则包含该服务；服务是否存在以及这些编译选项均不代表与 EV-MUX 的终端或文件传输兼容。

EV-MUX 的 Python 输入、输出与文件操作受唯一 owner 的连接、租约和 VM 代次校验约束。传统 WebREPL 的 `dupterm` 服务未实现此会话约定，因此不支持通过补装 Python 包或转发裸终端字节来绕过它。网络控制请使用*应用 API 完整参考*的 TCP 传输。保留的 `example/07-Network/00-WebREPL/webrepl.py` 入口会在改变 Wi-Fi 或打开监听端口前退出，并说明这一限制。

Chapter 3

快速入门

3.1 前置条件

- ESP32-P4/S3 开发板使用 ESP-IDF **v5.5.6**, ESP32-S31 开发板使用 **release/v6.1**; 请 source 对应 checkout 的 `export.sh`，使 `idf.py` 位于 `PATH` 中。
- 一块列在芯片与开发板支持 中且与所选芯片对应的开发板。

警告： CI 和发布固件仅验证上述开发板与 IDF 组合。v5.5.6 固定到补丁版本；S31 跟随 `release/v6.1` 维护分支。其它 IDF 版本可能涉及 ESP-IDF 或托管组件 API 变更，导致编译失败、运行行为变化，或改变可用的 MicroPython API；可能需要适配源码、overlay 或依赖，并重新进行硬件验证。可选 MicroPython 功能差异见项目 README。

备注： 在 Windows 平台编译时，请先安装 Visual Studio Build Tools，再运行 `idf.py`。在 Visual Studio Build Tools 安装器中，请勾选“Desktop development with C++”（使用 C++ 的桌面开发）工作负载，并确认已勾选“MSVC v143 - VS 2022 C++ x64/x86 build tools”组件。

3.2 构建、烧录与监视

带子模块克隆仓库，然后在仓库根目录使用板级感知的 `idf.py` 扩展：

```
git clone --recursive https://github.com/espressif/esp-vision.git esp-vision
cd esp-vision
```

```
idf.py --board ESP32_P4X_EYE -p /dev/ttyACM0 build flash monitor
```

该命令会先运行 `prepare-micropython`：校验 `lib/micropython` 已检出到固定的 MicroPython v1.28.0 提交，在 `build/micropython/` 下导出干净的 MicroPython 构建副本，再将 `overlay/micropython/` 应用到该副本，然后将每个 `boards/<BOARD>/port/` 投射到该副本的 `ports/esp32/boards/<BOARD>/`。`lib/micropython` 始终保持干净。

3.3 常用 idf.py 命令

命令	说明
`idf.py --board <BOARD> build`	为某块开发板构建固件。
`idf.py --board <BOARD> app`	为某块开发板构建应用。
`idf.py --board <BOARD> -p <PORT> app-flash`	按需构建，并仅向配置的应用分区地址烧录应用。
`idf.py --board <BOARD> -p <PORT> flash`	构建并烧录固件。
`idf.py --board <BOARD> -p <PORT> monitor`	打开串口监视器。
`idf.py --board <BOARD> menuconfig`	打开 menuconfig。
`idf.py --board <BOARD> -p <PORT> erase-flash`	擦除 flash。
`idf.py --board <BOARD> clean`	清理该板的构建输出。
`idf.py --board <BOARD> fullclean`	删除所选开发板的完整构建目录。

3.4 仅烧录应用

`app` 和 `app-flash` 适用于所有受支持的开发板与 ESP-IDF 组合，包括未开启 Iris 的普通固件。`app-flash` 从当前构建的分区表选择应用地址，不写入 bootloader、分区表、OTA 元数据或数据分区。板子必须已经具有匹配的布局和启动选择；该命令不会初始化空白设备或迁移分区。

选择对应板子的 ROM 下载端口，并为该固件配置指定独立构建目录。例如，已经安装匹配普通固件的 P4X EYE 可执行：

```
idf.py --board ESP32_P4X_EYE -B "$ESP_VISION_WORK_DIR/build/ESP32_P4X_EYE" \
  -p "$PORT" app-flash
```

普通构建的 `flash` 仍写入完整烧录清单。仅当 `ESP_VISION_ENABLE_IRISFW=ON` 时，根目录 `flash` 才自动选择 `app-flash`；Mosaico 保留布局的应用地址为 `0x210000`，其他配置不能沿用这一地址。Iris 专用的准备与验证流程见*已有 Iris 布局的应用直刷*。

3.5 运行第一个脚本

烧录完成后，通过 REPL 连接并尝试相机功能。每个*API 参考* 模块页面都会链接到对应 API 的可运行 `example/` 脚本。

Chapter 4

基础概念

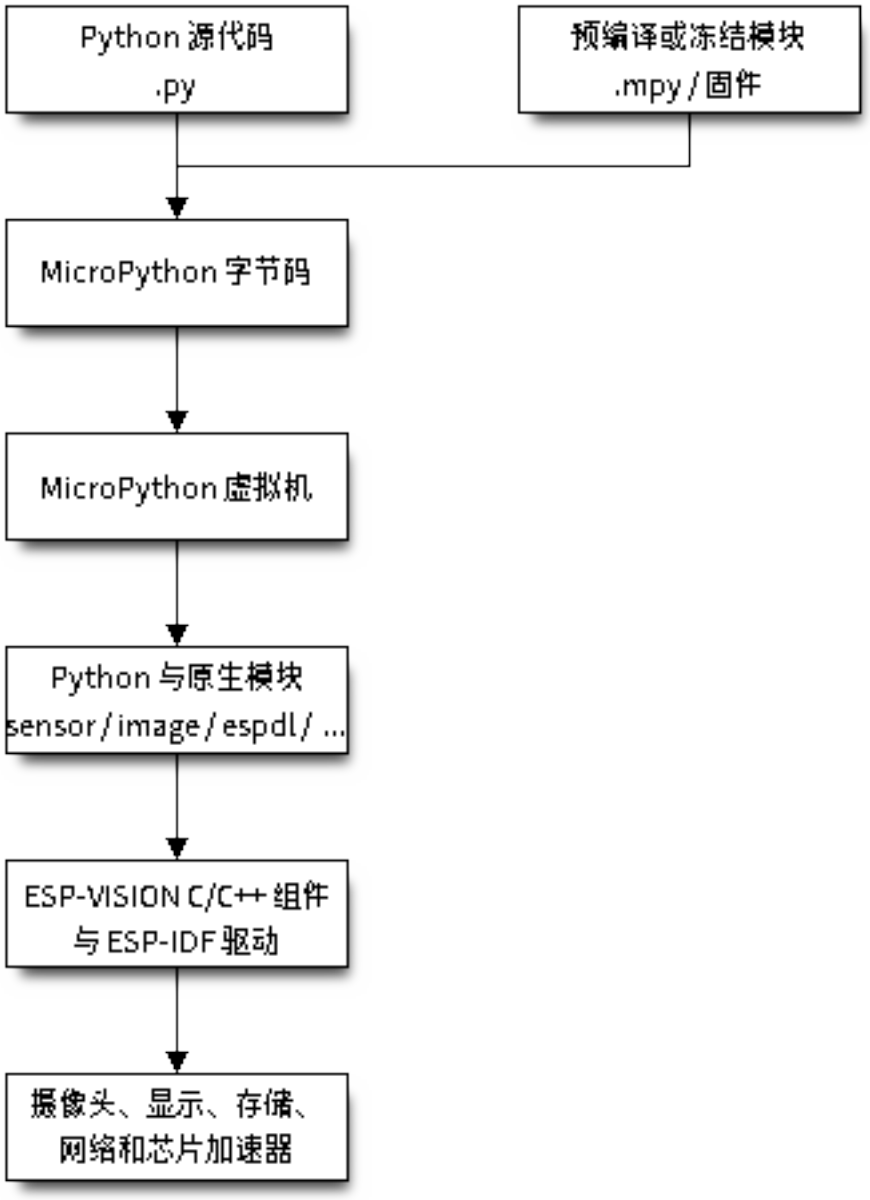
本章解释 ESP-VISION API 背后的原理：图像在内存中的表示方式、各类图像处理算法的工作机制、神经网络推理如何在设备上运行，以及图像帧如何在摄像头与编解码流水线中流转。建议与*API 参考* 对照阅读，理解每个函数“为什么”这样工作。

4.1 MicroPython 语言基础

MicroPython 是面向微控制器直接运行的紧凑型 Python 语言实现。ESP-VISION 使用 MicroPython v1.28.0 作为应用开发语言，同时通过 C/C++ 与 ESP-IDF 组件实现硬件访问和性能关键的视觉处理。本章介绍阅读和编写 ESP-VISION 应用所需的语言模型；完整权威说明仍以 MicroPython v1.28.0 语言与实现参考 为准。

4.1.1 执行模型

上游 MicroPython 以实现 Python 3.4 语法为目标，并选择性支持后续 Python 版本的部分特性。ESP-VISION 以 `EXTRA_FEATURES` 配置级别构建 ESP32 port，并启用设备端编译器、垃圾回收器、MPZ 长整数、单精度浮点、调度器、VFS 和持久字节码加载。源代码仍是动态类型的 Python，但会被编译为紧凑字节码，并由 MicroPython 虚拟机执行。原生模块将 Python 对象连接到 ESP-VISION C/C++ 组件、ESP-IDF 驱动和芯片硬件：



从 `.py` 加载的代码会在设备上编译，并在编译期间占用 RAM。预编译的 `.mpy` 文件和冻结到固件中的模块可以减少运行时编译开销；冻结模块中的不可变常量可以保留在 Flash 中。Python 仍负责应用策略和对象生命周期，而原生组件负责图像帧采集、图像处理、编解码和 AI 推理等操作。

4.1.2 基本语法

ESP-VISION 当前配置使用缩进定义代码块，并支持赋值、表达式、条件、循环、函数、类、异常、上下文管理器、推导式、生成器、导入以及 `async`/`await` 等常用 Python 形式。名称会动态绑定到对象，因此声明时不需要指定 C 风格的存储类型：

```
from micropython import const

_MIN_PIXELS = const(80)
```

(续下页)

(接上页)

```
class Detection:
    def __init__(self, label, score):
        self.label = label
        self.score = score

    def accepted(self, threshold=0.5):
        return self.score >= threshold

def select_results(results):
    selected = []
    for label, score in results:
        item = Detection(label, score)
        if item.accepted():
            selected.append(item)
    return selected

try:
    detections = select_results((("person", 0.91), ("chair", 0.42)))
    for item in detections:
        print(item.label, item.score)
except (ValueError, TypeError) as error:
    print("invalid result:", error)
finally:
    print("processing complete")
```

`const()` 是 MicroPython 扩展，允许编译器直接替换常量值，并可减少字节码和全局字典开销。类型注解有助于开发者和编辑器理解代码，但 MicroPython 不会借此提供 C 风格的静态类型或自动内存布局。

4.1.3 主要数据结构

在微控制器上，可变对象与不可变对象的选择十分重要，因为创建、扩容或拼接对象都会分配堆内存：

类型	可变性	ESP-VISION 使用建议
`None` 与 `bool`	不可变	表示值缺失和状态标志，无需额外定义自有哨兵对象。
`int` 与 `float`	不可变	存储计数、坐标、阈值和分数。重复算术运算可能创建新对象；硬中断上下文中应避免浮点运算。
`str`	不可变	存储文本、路径、标签和 JSON 键。拼接会创建新字符串，因此应避免每帧构造大型字符串。
`bytes`	不可变	存储紧凑的二进制常量、编码数据包以及只读模型或协议数据。
`bytearray`	可变	为外设 I/O 和数据包组装提供可复用二进制缓冲，避免每次操作都分配新对象。
`list`	可变	存储长度变化的有序结果。`append()` 和扩容可能分配内存，因此持续帧循环中的列表应限制大小或重复使用。
`tuple`	不可变	表示矩形和检测结果等固定记录。常量元组通常比反复创建列表更节省内存。
`dict`	可变	表示配置和共享标量状态。新增键可能触发表扩容，应在初始化期间创建预期结构。
`set`	可变	在额外哈希表内存开销合理时，用于成员关系和唯一性检查。
`range`	不可变	描述整数迭代范围，无需构造包含全部数值的列表。
`memoryview` 与 `array`	视图 / 可变	以较少拷贝访问类型化或切片缓冲数据；视图有效期间必须保持底层缓冲存活且不能调整其大小。

对象引用不会自动复制。执行 `b = a` 后，两个名称会引用同一个可变列表、字典、图像或缓冲；仅在确实需要独立数据时才应显式复制。对于由可复用摄像头帧缓冲支持的 *image.Image* 对象，这一点尤其重要。

4.1.4 面向微控制器的设计理念

MicroPython 优先考虑可移植性、交互式开发、紧凑固件集成和直接硬件访问，而不是完整覆盖 CPython 标准库。垃圾回收堆使动态应用代码可行，但 RAM、Flash、栈空间和内存分配时延仍然有限。MicroPython 受限设备指南 建议减少重复对象创建、预分配缓冲、将常量和可复用模块放入冻结字节码，并使用 `gc.mem_free()` 与 `gc.mem_alloc()` 监控堆。

ESP-VISION 应用应在进入帧循环前初始化模块、模型、缓冲和结构稳定的字典。重复使用 `bytearray` 与图像缓冲，限制每帧临时数据量；除非已经复制，否则不要在下一次 `sensor.snapshot()` 后继续持有摄像头帧缓冲支持的图像。Python 适合表达编排与产品行为；确定性实时工作、ISR 处理、大批量像素处理、编解码和推理内核应保留在原生组件中。

4.1.5 MicroPython 与 CPython

本文档在进行运行时对比时，“Python” 指通常运行于 PC 和服务器的参考实现 CPython 3.x。MicroPython 的具体功能集取决于固件配置，因此下表描述的是 ESP-VISION 当前配置的 MicroPython v1.28.0：所有维护中的开发板 manifest 都会冻结 `asyncio`，ESP32 port 启用了带 GIL 的 `_thread`，硬件 API 则仍受芯片和开发板配置约束。

对比项	ESP-VISION MicroPython	CPython
主要运行环境	运行于 MCU 固件内部，可直接访问外设与板级服务。	作为操作系统进程运行于 PC、服务器和较大型嵌入式系统。
语言语法	实现 Python 3.4 语法并选择性支持后续特性；常用控制流、函数、类、异常、生成器和异步语法均可使用。	跟随当前 Python 语言规范，通常最先引入新语法。
类型系统	动态类型；类型注解不会使执行过程变为静态类型。	动态类型；类型注解主要由工具和可选库使用。
标准库	提供构建固件时选择的资源导向子集，并增加 `machine`、`micropython`、`esp` 和 `esp32` 等硬件与运行时模块。	为安装的版本提供完整 CPython 标准库，但不包含标准 MCU 外设 API。
软件包部署	模块可复制到存储、预编译为 `.mpy` 或冻结到固件中；不能假定支持桌面 `pip` 和任意二进制 wheel。	通常使用 `pip`、虚拟环境、源码发行包和平台 wheel。
内存模型	使用与固件资源共享的小型垃圾回收堆，必须显式考虑分配失败和碎片化。	使用容量大得多的进程堆，并采用引用计数和循环垃圾回收。
并发	提供固件内冻结的 `asyncio`、已启用且带 GIL 的 `_thread`，以及经过调度的 GPIO IRQ 与 Timer 回调；外设可用性仍取决于所选芯片和开发板。	提供 `asyncio`、线程、进程和丰富的操作系统同步机制。
性能路径	高负载操作调用原生 C/C++ 模块和硬件加速器；Python 更适合用于编排。	可使用优化扩展模块、带 JIT 的其他运行时或多进程，但通常拥有更多 CPU 与内存资源。
兼容性细节	为减少代码体积和 RAM 使用，部分内置行为、异常文本、反射能力、模块内容和边界情况存在差异。	定义 MicroPython 差异文档所对照的参考行为。
可移植性	使用受支持子集的纯 Python 代码通常可以移植；硬件模块和资源假设与设备相关。	依赖可用时，纯 Python 代码通常可在受支持操作系统间移植。

不能只根据语法判断可移植性，还应检查导入模块、内存使用、文件系统路径、并发假设和硬件访问。上游 MicroPython 与 CPython 差异 记录了已知行为差异，*MicroPython API* 则说明 ESP-VISION 可用的沿用 API 与并发模型。

4.2 软件包与部署

MicroPython 以模块和软件包组织可复用代码，但嵌入式产品还需要确定代码的存储位置和更新方式。本章基于 ESP-VISION 当前固定使用的 MicroPython v1.28.0 介绍软件包管理与部署；完整的通用行为请参考上游 软件包管理 和 manifest 文档。

4.2.1 模块与软件包结构

模块通常是 `.py` 源文件或预编译的 `.mpy` 文件。软件包是包含 `__init__.py` 文件以及一个或多个模块或子包的目录，因此可复用的视觉流水线可以采用以下结构：

```
my_vision/
    __init__.py
    pipeline.py
    transport.py
```

应用程序使用标准 Python 语法导入软件包成员：

```
from my_vision.pipeline import VisionPipeline

pipeline = VisionPipeline()
```

4.2.2 当前固件的软件包管理

MicroPython 使用 `mip` 而不是 CPython 的 `pip`，可从 micropython-lib、软件包索引、URL 或本地 `package.json` 描述中安装软件包。ESP-VISION 默认 manifest 当前没有嵌入设备端 `mip` 模块，因此标准固件不支持 `import mip`；请在开发主机上使用 `mpremote` 将软件包安装到设备文件系统：

重要： ESP-VISION 默认固件不支持设备端 `mip`。在设备上执行 `import mip` 或 `mip.install()` 会引发 `ImportError`。主机端 `mpremote mip` 命令仍然可用，并且是默认的软件包安装方式。

```
mpremote connect <PORT> mip install --target=/lib <PACKAGE>
mpremote connect <PORT> mip install --target=/lib <PACKAGE>@<VERSION>
mpremote connect <PORT> mip install --target=/lib ./package.json
```

`package.json` 用于描述 `mip` 在运行时或设备预配置阶段安装的文件和依赖项。它不是固件 manifest，也不决定哪些模块会被编译进 ESP-VISION。由于 MicroPython 软件包可能依赖固件功能或特定架构的 `.mpy` 文件，因此必须确认其兼容 MicroPython v1.28.0 和所选芯片架构。

在设备端使用 `mip.install()` 安装软件包，需要客制化固件显式加入 `mip` 以及所需的网络和 TLS 支持。默认推荐通过主机端 `mpremote` 安装，因为这种方式不会增大产品固件，也不要求设备通过网络下载代码。

4.2.3 文件系统中的软件包

ESP-VISION 将内部 Flash 文件系统挂载到 `/`，并将 `/lib` 加入 `sys.path`。可复用软件包应安装到 `/lib`；此后应用程序无需修改代码，即可导入当前目录中的模块、冻结模块和 `/lib` 下的软件包。

启用 SD 卡后，SD 卡挂载到 `/sdcard`，但其软件包目录不会自动加入 `sys.path`。导入其中的软件包前需要显式添加路径：

```
import sys

sys.path.append("/sdcard/lib")
from my_vision.pipeline import VisionPipeline
```

.py 源文件在导入时编译，会占用 RAM 保存字节码并增加导入延迟。预编译的 .mpy 文件无需在设备上编译并可减少源代码暴露，但加载后的字节码仍会占用 RAM，而且必须匹配 MicroPython 版本和架构。

4.2.4 嵌入固件的软件包

固件 manifest 用于选择需要编译并冻结到固件镜像中的 Python 模块和软件包。ESP-VISION 将开发板 manifest 保存在 boards/<BOARD>/manifest.py，当前通过这些文件冻结端口模块、ESP-VISION 初始化模块、asyncio 以及特定开发板需要的功能。

例如，将 my_vision/__init__.py 及其模块放在 boards/<BOARD>/packages 下，然后在该开发板的 manifest 中加入软件包：

```
package(
    "my_vision",
    base_path="$(ESP_VISION_ROOT)/boards/<BOARD>/packages",
)
```

底层 freeze() 函数也可以冻结一个目录或指定模块，include() 和 require() 则用于复用已配置 manifest 库中的 manifest 片段和软件包。修改 manifest 后，需要重新配置并构建所选开发板：

```
idf.py --board <BOARD> reconfigure
idf.py --board <BOARD> build
idf.py --board <BOARD> flash
```

4.2.5 选择部署方式

可以根据软件包的预期更新频率及其是否属于产品固定依赖来选择部署方式：

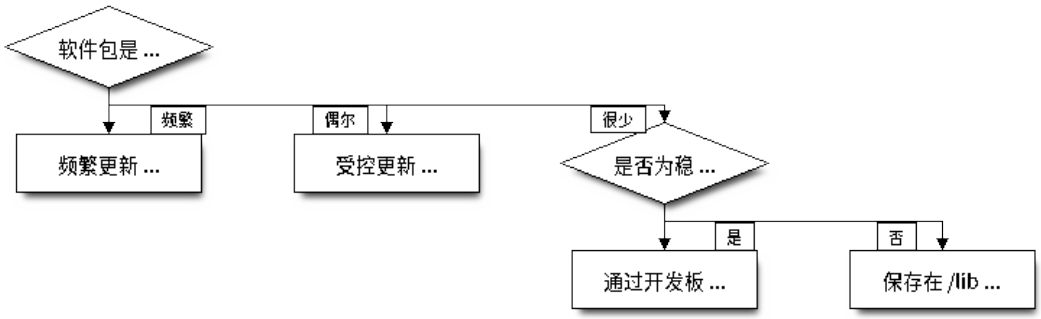


图 1: 软件包部署决策

表 1: 部署方式对比

属性	/lib 下的 .py	/lib 下的 .mpy	冻结到固件
存储位置	可写文件系统	可写文件系统	固件 Flash 镜像
导入行为	导入时编译	无需编译源文件即可加载	直接执行冻结字节码
运行时 RAM	字节码和对象占用 RAM	字节码和对象占用 RAM	冻结字节码和常量可保留在 Flash
更新方式	替换文件或使用 mpremote mip	替换兼容的 .mpy 文件	重新构建并烧录固件
主要用途	开发及频繁更新的应用代码	导入开销较低且可替换的模块	稳定的产品依赖和启动模块

4.2.6 导入优先级与版本管理

当前固件依次搜索当前目录、通过 .frozen 提供的冻结模块，然后搜索 /lib。因此，同名的文件系统软件包在正常导入时不能覆盖冻结软件包；需要从 manifest 中移除冻结软件包并重新构建固件，或使用不同的软件包名称。

应将开发板 manifest 和冻结软件包版本纳入源码管理，以确保固件构建可复现。package.json 用于描述 mip 安装的文件和依赖项，manifest.py 用于定义固件构建时嵌入的模块；两者服务于不同的部署阶段，不能相互替代。

对于 ESP-VISION 产品，建议冻结稳定的框架扩展和启动依赖，将产品脚本、配置及需要现场更新的模块保存在 /lib。大型 AI 模型、图像和视频资源通常应作为数据文件保存在 Flash、SD 卡或外部存储中，而不是作为 Python 软件包数据嵌入固件，以便更新并由适合的运行时组件进行映射。

4.3 图像模型

ESP-VISION 中的每一个视觉操作都在读写一个*image.Image*：一个由像素组成的矩形网格，外加描述其宽、高与像素格式的小头部。理解像素的排布方式、算法使用的色彩空间，以及底层内存的管理方式，是写出高效且正确脚本的关键。

4.3.1 像素格式

像素格式决定每个像素占用多少字节，以及其数值如何被解释。

格式	字节/像素	说明
BINARY	1 bit	黑/白。用于掩膜以及*image.Image.binary()* 的输出。
GRAYSCALE	1	8 位灰度（0-255）。多数滤波与检测算法优先使用的格式。
RGB565	2	16 位彩色：红 5 位、绿 6 位、蓝 5 位。
BAYER	1	来自传感器的原始单通道马赛克，按需去马赛克。
YUV422	2	亮度/色度打包，常见于摄像头与编解码流水线。
JPEG / PNG	不定	压缩字节流。JPEG 由*image.Image.to_jpeg()* 生成；当前固件未启用 PNG 编解码。

RGB565 把一个颜色打包进 16 位，以色彩精度换取相较 RGB888 一半的内存。绿色多分到一位，因为人眼对绿色最敏感。*image.Image.get_pixel()* 既可返回打包值，也可返回展开为每通道 8 位的 (r, g, b) 元组。

4.3.2 色彩空间

尽管像素以灰度或 RGB565 存储，颜色阈值化却在 **LAB** 色彩空间中进行。LAB 把亮度（L）与两条颜色对立轴（A：绿-红，B：蓝-黄）分开，因此用 LAB 表示的阈值对亮度变化远比 RGB 鲁棒。这也是为什么颜色阈值是六元组 `(l_min, l_max, a_min, a_max, b_min, b_max)`，而灰度阈值只是 `(min, max)`。

模块级转换辅助函数（*image.rgb_to_lab()*、*image.lab_to_rgb()*、*image.rgb_to_grayscale()* 以及其余 `*_to_*` 系列）对单个像素暴露这些转换，便于离线计算阈值。

4.3.3 帧缓冲与 `fb_alloc`

嵌入式内存十分有限，因此 ESP-VISION 避免逐帧在堆上分配。*sensor.snapshot()* 返回的图像由一块可复用的 **帧缓冲** 支撑：下一次 `snapshot()` 会覆盖它。如果需要保留某一帧（例如在*image.Image.difference()* 中与后续帧比较），请显式调用 `img.copy()`。

许多算法需要只在调用期间存在的临时内存。它们使用一个栈式分配器 `fb_alloc`，从帧缓冲区域中划出临时缓冲，并在操作返回时一次性释放。这就是重负载方法不会造成堆碎片的原因，也是在热循环中应优先复用同一图像而非反复分配的原因。

4.3.4 感兴趣区域（ROI）

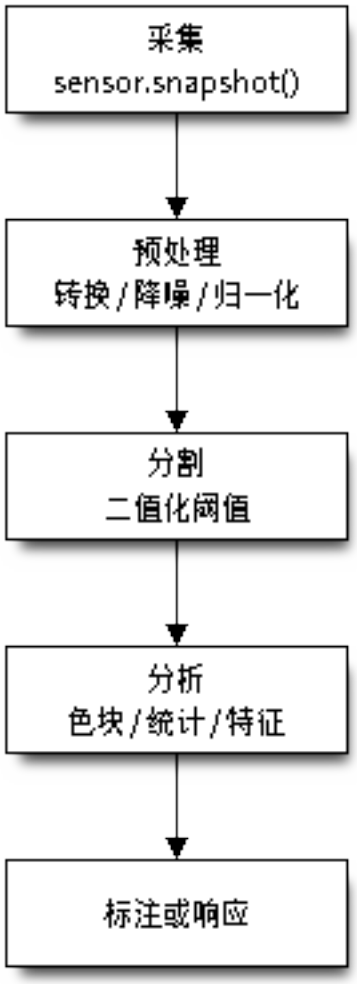
多数分析与转换方法接受 `roi=(x, y, w, h)` 关键字，把操作限制在图像的某个子矩形内。在 ROI 上工作既更快（像素更少）又更有针对性（忽略无关区域）。ROI 始终以源图像的像素坐标表示。

4.4 图像处理

image – 图像处理 模块中的视觉算法来自 OpenMV 的 `imlib` 库，在本仓库中以 `components/imlib` IDF 组件的形式维护。它们大致分为几类：点变换与几何变换、邻域滤波、统计分析，以及特征检测。本页说明每一类的作用与适用场景。

4.4.1 典型处理流水线

多数颜色跟踪或检测脚本都遵循以下处理流程：



预处理可以组合使用*image.Image.to_grayscale()*、*image.Image.gaussian()* 和*image.Image.histeq()*；分割通常使用*image.Image.binary()*；分析阶段再使用*image.Image.find_blobs()*、*image.Image.get_statistics()* 或特征检测器，随后由应用绘制结果或执行相应操作。

4.4.2 阈值化与分割

image.Image.binary() 通过把每个像素与一组颜色阈值比较，把图像转换为掩膜（阈值为何用 LAB 表示见图像模型）。只要像素落入 任一阈值即为 “有效”，因此可同时跟踪多种颜色。配套的*image.Image.find_blobs()* 在相同阈值上运行 8 连通分量标记，为每个连通区域返回一个*image.blob*，包含质心、外接框、像素数、朝向，以及圆度、实心度等形状描述子。

4.4.3 邻域（卷积）滤波

这些方法用像素 `(2*ksize+1)` 方形邻域的某个函数来替换该像素：

- *image.Image.mean()* ——盒式平均；快速模糊。
- *image.Image.gaussian()* ——用帕斯卡三角（二项式）近似高斯核的加权平均；标准降噪模糊。设 `unsharp=True` 则改为锐化。

- *image.Image.laplacian()* ——二阶导数边缘响应；设 sharpen=True 可叠加回原图以增强边缘。
- *image.Image.morph()* ——应用任意整数卷积核。mul 与 add 缩放并偏置结果；mul 默认为 1/sum(kernel)，以保持图像亮度。

它们都带有 threshold/offset/invert 关键字，可在一次遍历中把滤波变成自适应阈值化操作。

4.4.4 秩滤波与保边滤波

秩滤波对邻域排序而非求平均，能在较少模糊边缘的前提下去噪：

- *image.Image.median()* ——取百分位（默认 0.5）值；对椒盐噪声效果极佳。
- *image.Image.mode()* ——取最常见值。
- *image.Image.midpoint()* ——在最小值与最大值之间按 bias 混合。
- *image.Image.bilateral()* ——同时按空间距离（space_sigma）与颜色相似度（color_sigma）加权的高斯模糊，平滑平坦区域的同时保持边缘锐利。

4.4.5 形态学

image.Image.erode() 与*image.Image.dilate()* 用方形结构元收缩或扩张（通常为二值图像的）有效像素；*image.Image.open()* 与*image.Image.close()* 组合二者以去除小噪点或填补小孔。threshold 参数控制需要多少个邻居为有效，从而推广了经典二值算子。

4.4.6 直方图与统计

image.Image.get_histogram() 按通道对像素值分箱；返回的*image.histogram* 可计算 Otsu 阈值（*image.histogram.get_threshold()*）、百分位以及完整统计量。*image.Image.get_statistics()* 一次返回均值、中位数、众数、标准差、四分位数与最小/最大值，便于在运行时自动整定阈值。

4.4.7 特征检测

更高层的检测器用于发现几何结构：

- *image.Image.find_lines()* 与*image.Image.find_circles()* 使用霍夫变换；其 threshold 是最小累加器分数，*_margin 关键字用于合并近似重复的结果。
- *image.Image.find_line_segments()* 使用 Edge Drawing Lines；其 threshold 是最小 Sobel 梯度幅值，merge_distance 和 max_theta_diff 用于控制线段合并。
- *image.Image.find_rects()* 定位四边形（适用于基准标记与屏幕）。
- *image.Image.find_qrcodes()* 与*image.Image.find_apriltags()* 检测并解码相应的标记类型；在提供相机内参时，AprilTag 还能返回 6 自由度位姿。

当前 ESP32-P4 板级配置还通过 ZXing-C++ 后端提供*image.Image.find_barcodes()*。

备注： 本构建禁用了 OpenMV 中受 GPL 许可的代码路径（OMV_NO_GPL=1），因此少量上游算法被有意排除。每块板子的 imlib_config.h 决定编译哪些可选算法；未启用的方法会在调用时抛出异常。

4.5 AI 推理

ESP-VISION 通过*espdl –模型推理* 和*tflite –模型推理* 提供端侧神经网络执行能力：前者使用 ESP-DL，后者通过 TensorFlow Lite Micro 运行 TensorFlow Lite .tflite 模型。可用 API 覆盖从面向任务的辅助接口

到通用模型执行；模型相关的预处理或后处理可以放在模块绑定、辅助代码或 Python 脚本中，取决于具体模型族。

4.5.1 模型运行时

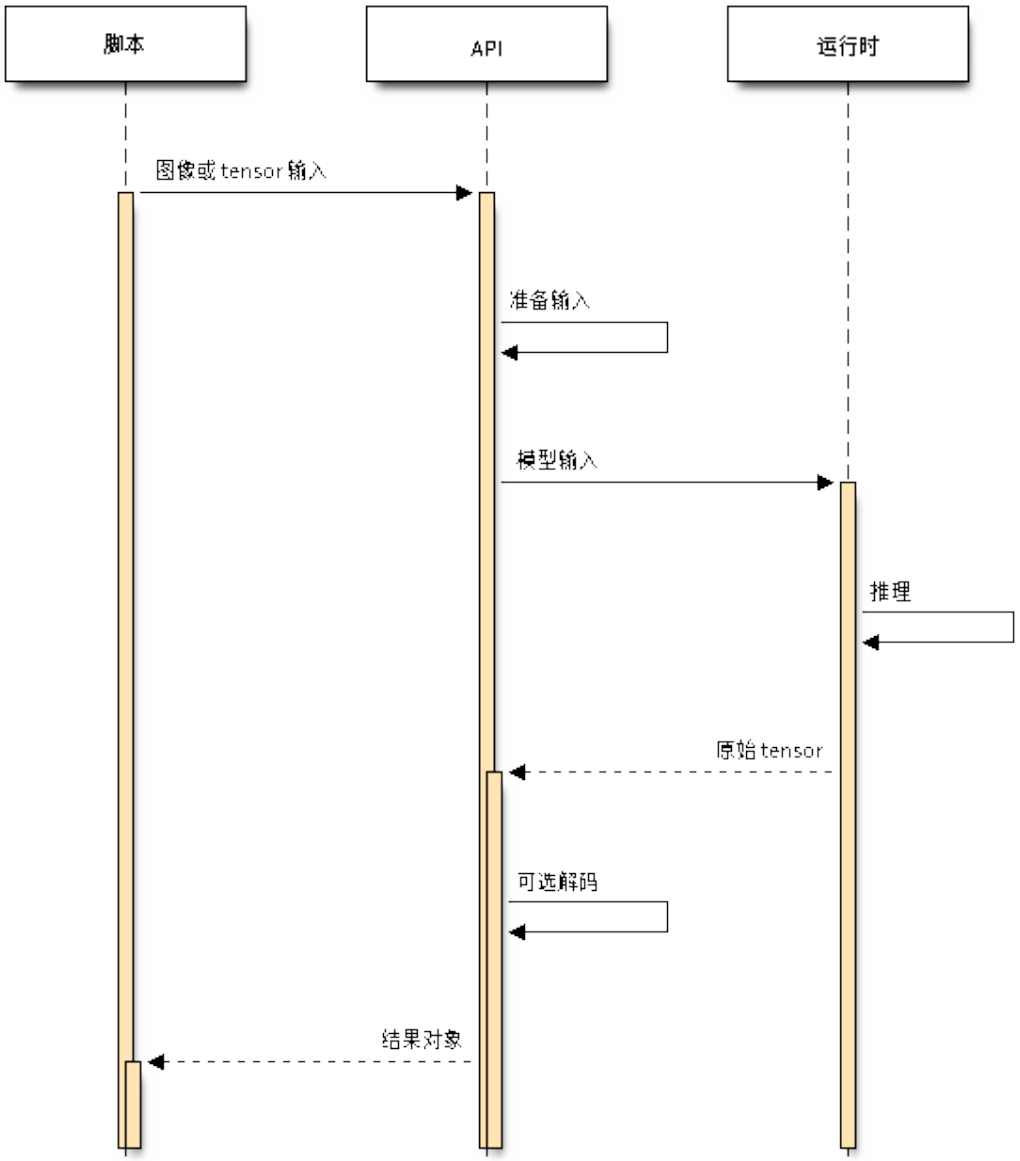
ESP-DL 和 TFLite Micro 是 ESP-VISION 当前暴露的两条模型运行路径。`espdl` 模块使用 `.espdl` 文件，`tflite.Model` 使用 `.tflite` 文件。模型文件可放在 `/sdcard` 或 `/flash` 等板级存储中，并在运行时加载。具体 API 形态、量化元数据、输入输出解释由所选运行时和模型共同定义。

4.5.2 模型文件与量化

模型通常在 PC 端从训练框架导出或转换后，再拷贝到板级存储。对于微控制器，量化模型通常更合适，因为它能降低模型体积、激活内存和计算开销。量化会用 scale 与 zero point 等元数据把浮点 tensor 范围映射到整数值；具体元数据和文件布局取决于运行时及转换工具链。

4.5.3 推理流程

一次推理调用通常组合若干阶段：准备输入数据、运行量化网络，再把原始输出解码成应用可用的结果：



4.5.4 预处理

推理前，输入数据必须与模型训练时一致：

- **形状**必须匹配模型输入 tensor，图像模型还包括分辨率和通道数。
- **颜色**顺序与像素格式必须匹配训练流程。
- **归一化或量化**必须使用模型期望的 scale、zero point、mean、standard deviation 或其他变换。部分 API 会把这些暴露为参数，示例也可能直接在 Python 中展示。

4.5.5 后处理

网络原始输出通常需要按任务解码后，才能被应用直接使用：

- **目标检测**（*espdl.PedestrianDetect*、*espdl.ESPDet*、*espdl.YOLO11*）产生带类别分数的候选框。置信度 `score` 阈值剔除弱框，**非极大值抑制**（`nms`）去除重叠重复，最终得到 `(x, y, w, h, score, category)` 元组。`PedestrianDetect` 和 `YOLO11` 还用 `topk` 限制数量。

- **姿态估计**（*espdl.YOLO11nPose*）为每个检测附加 17 个 COCO 关键点。
- **分类**（*espdl.ImageNetCls*）应用可选的 softmax，返回 topk 个 (label, score) 对。
- **原始输出解码**（*espdl.Model*、*tflite.Model*）暴露任务解码前的模型输出。应用代码需要解包 tensor dtype，使用量化元数据，并执行 anchor 解码、sigmoid 或 softmax、NMS、top-k 选择以及坐标映射等模型相关步骤。

阈值和结果数量通常可以在运行时调整而无需重新加载模型，便于适应光照、距离或场景密度变化。

4.5.6 内存与性能

模型及其激活缓冲较大，分配在 PSRAM 中。请加载一次模型并跨帧复用，而不要逐帧重建。用完后调用 deinit()（或释放最后一个引用）以释放模型。推理开销通常随输入分辨率和模型规模增长，因此较小模型一般比全尺寸网络帧率更高。

4.6 摄像头流水线

sensor.snapshot() 看起来只是一次调用，但其背后是一条与硬件相关的采集流水线，把光子变成*image.Image*。了解这些阶段有助于理解配置调用（set_pixformat、set_framesize、skip_frames）以及各开发板的性能特征。

4.6.1 从传感器到图像



取流前，传感器控制总线会应用开发板默认配置以及请求的图像格式。双缓冲允许处理当前帧的同时填充下一帧。ESP32-P4 使用硬件 PPA 完成缩放和色彩转换，ESP32-S3 则由软件完成转换。配置后调用 `sensor.skip_frames(time=2000)`，可为自动曝光和自动白平衡预留收敛时间。

4.6.2 各开发板后端

摄像头后端在构建时按板选择：

开发板系列	后端	说明
ESP32-P4	`esp_video` / V4L2 + PPA	MIPI-CSI 传感器；硬件加速的缩放与色彩转换。
ESP32-S3	`esp32-camera`	DVP 传感器；软件转换，建议使用较小分辨率。

由于公开的 `sensor` API 在各板上完全一致，同一脚本可在两者上运行；区别仅在于可达到的分辨率与帧率。

4.6.3 分辨率

`set_framesize` 接受来自共享分辨率表的命名尺寸（`QQVGA`、`QVGA`、`VGA` 等）。较小的帧在每个阶段都更省内存、更省处理时间，因此当模型或算法只需小输入时，应直接采集小帧，而不是采集大帧再缩小。

4.7 编解码与推流

拿到图像帧后，常常需要把它送出开发板：在主机上预览、录制，或实时推流。ESP-VISION 提供多种编解码与传输方式，各自适用于不同场景。

4.7.1 静态图像编码：JPEG 与 PNG

`image.Image.to_jpeg()`（及其别名*`image.Image.compress()`*）把一帧编码为 JPEG。JPEG 有损：`quality` 关键字（1-100）在大小与画质之间权衡，`subsampling` 控制色度被压缩的程度（`JPEG_SUBSAMPLING_420` 最小，`444` 画质最好）。带硬件 JPEG 编码器的板会进行硬件卸载，否则使用软件编码器（`esp_new_jpeg`）。

当前固件未启用 PNG 解码器或编码器。通过 `image.Image(path)` 加载 PNG 时立即抛出 `OSError`；*`image.Image.to_png()`* 和将未压缩图像保存为 `.png` 时抛出 `RuntimeError`。请在上传前将 PNG 素材转换为 JPEG。

4.7.2 录制序列：ImageIO

`imageio.ImageIO` 录制一序列帧并保留帧间时间，从而以原始速度回放。**文件流**在存储上写容器文件；**内存流**把帧保存在预分配的 PSRAM 缓冲中，以便快速采集与回放。详见*image.ImageIO –图像流*。

4.7.3 视频编码：H.264

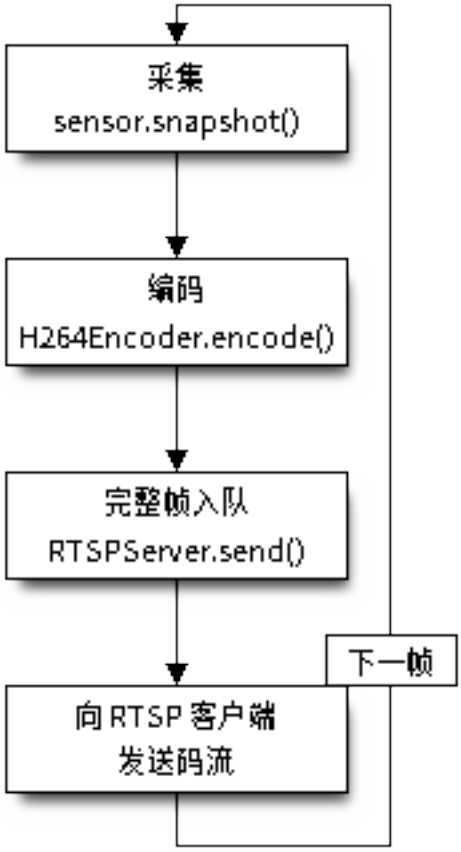
对于连续视频，逐帧 JPEG 很浪费，因为它无法利用相邻帧之间的相似性。*`h264.H264Encoder`* 产生 H.264 流，其中偶发的 **关键帧**（IDR/I）是自包含的，而其间的帧只编码发生变化的部分。编码器参数用于调节这一权衡：

- `gop` ——关键帧间隔。越短则恢复越快、越易跳转，但输出越大；`0` 表示每秒一个关键帧。
- `bitrate` ——目标每秒比特数；大小/画质的主要控制项。
- `qp_min` / `qp_max` ——限定每帧量化参数的范围。

丢失一个 P 帧会导致画面损坏，直到下一个关键帧为止，这对下面的推流传输很重要。

4.7.4 网络推流：RTSP

`rtsp.RTSPServer` 通过 RTSP 发送编码后的 H.264 帧，使 VLC、ffplay 等客户端能在 `rtsp://<board-ip>:8554/` 观看摄像头画面。应将每个*`h264.H264Encoder.encode()`* 的结果交给*`rtsp.RTSPServer.send()`*。



服务器维护一个短小且有序的帧队列；若客户端缓慢或缺席，它会整帧丢弃，而不是阻塞采集循环或发送会破坏码流的半帧。

4.7.5 主机预览：USB CDC

在开发期间，看到摄像头画面最快的方式是*image.Image.flush()*，它通过 EV-MUX `preview.frame` 通道把当前帧作为二进制 JPEG 推送到主机，由配套的主机工具解码并显示。该通路无需 Wi-Fi、无需 SD 卡，是脚本迭代时默认的反馈回路。

4.8 启动流程

ESP-VISION 沿用 MicroPython 的复位与启动模型，并增加 Flash 文件系统、板级存储、摄像头、显示和预览服务的初始化。本章说明当前固件实际实现的启动顺序。关于上游模型和通用行为，请参阅 MicroPython v1.28.0 复位与启动流程。

4.8.1 硬复位与软复位

硬复位会先重启 MCU 和 ESP-IDF 运行时，再创建新的 MicroPython 环境。上电、开发板复位按键、`machine.reset()`、看门狗或欠压复位以及从深度睡眠唤醒都会触发硬复位。应用需要区分复位来源时，可以使用 `machine.reset_cause()`。

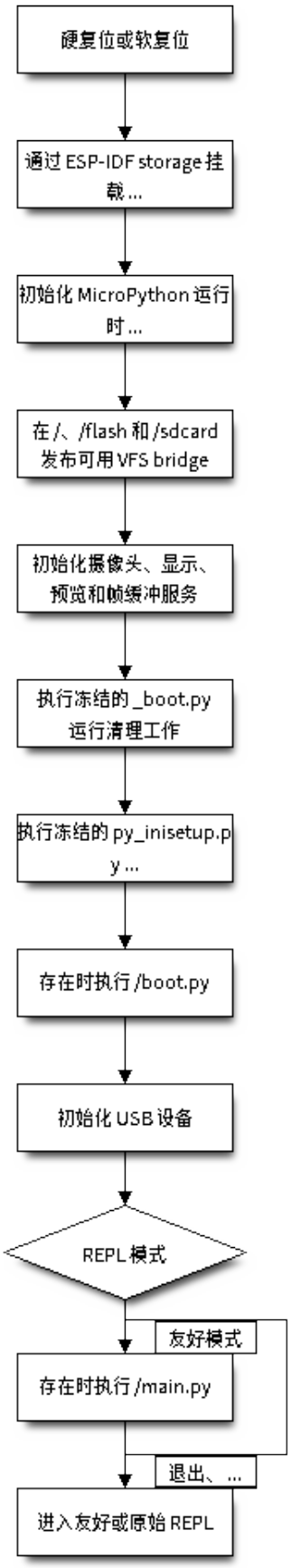
软复位在不重启完整 MCU 运行时的情况下重新启动 MicroPython 环境，可通过友好 REPL 中的 `Ctrl-D` 或 `machine.soft_reset()` 触发。ESP-VISION 会清除 Python 对象和模块，通过 MicroPython 清理过

程关闭文件和 socket，释放摄像头、显示、预览、USB、PWM、定时器、UART、线程及其他托管资源，然后重新执行 Python 启动流程。

部分系统状态可能在软复位后保留，包括 RTC、CPU 时钟配置，以及 IP 层仍处于活动状态的网络接口。应用代码不能假定表示这些资源的 Python 对象仍然存在；每次复位后都应重新创建对象并检查其状态。

4.8.2 ESP-VISION 启动顺序

硬复位或软复位后，ESP-VISION 执行以下启动流程：


硬复位或软复位
通过 ESP-IDF storage 挂载 ...
初始化 MicroPython 运行时 ...
在 /、/flash 和 /sdcard 发布可用 VFS bridge
初始化摄像头、显示、预览和帧缓冲服务
执行冻结的 _boot.py 运行清理工作
执行冻结的 py_inisetup.py ...
存在时执行 /boot.py
初始化 USB 设备
REPL 模式
友好模式
存在时执行 /main.py
退出、...
进入友好或原始 REPL

主机自动化工具使用的原始 REPL 可以在软复位时跳过 `main.py`，从而使开发工具无需自动启动产品应用即可取得设备控制权。

4.8.3 首次启动与 Flash 文件系统

ESP-IDF storage manager 独立于 MicroPython VM 持有开发板文件系统。默认尝试在内部将 `ffat` 分区挂载到 `/flash`，并将可用的 SD 卡挂载到 `/sdcard`。两个卷均为独立的可选卷。每次 VM 启动时，原生 MicroPython VFS bridge 会把 Flash 卷发布到 `/` 及别名 `/flash`，把 SD 卷发布到 `/sdcard`。软复位重新创建 bridge 对象，不会重新挂载已挂载的物理文件系统；开启自动 SD 挂载时，冷启动时不可用的卡会在后续 VM 启动重试。开发板可分别将 `ESP_VISION_STORAGE_AUTO_MOUNT_FLASH` 或 `ESP_VISION_STORAGE_AUTO_MOUNT_SD` 设为 `0` 来推迟对应挂载，两者默认均为 `1`。显式挂载重试及固定 USB MSC 卷槽位见*存储、文件系统与数据保护*。

自动初始化使用两个独立的板级策略：`ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM` 默认 `1`，`ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM` 默认 `0`。精确返回 `FR_NO_FILESYSTEM` 时，内部 `ffat` 默认初始化，SD 只有在板级显式开启后才初始化；这可能擦除出厂、其他格式或损坏的数据。对应策略设为 `0` 后自动启动只尝试挂载。两个介质仍默认自动挂载，`storage.retry()` 始终只挂载。有效 FAT 保持原样，其他挂载错误不触发自动格式化或重启。中断的 Flash 事务保持隔离；SD 没有同等持久日志。策略风险、所有权、备份和显式确认恢复见*存储、文件系统与数据保护*。无法打开文件时仍可查询状态；显式恢复后，由用户选择软复位以发布 Python 挂载。冻结的默认文件初始化仅写入可用文件系统，失败会报告错误，不导致 crash。

正常固件升级或软复位不会覆盖已有启动文件。开发板包可以通过 `boards/<BOARD>/board_inisetup.py` 提供板级默认 `main.py` 和 `README.txt` 内容。

4.8.4 使用 boot.py

`boot.py` 适合执行必须在应用启动前完成、耗时短且结果确定的初始化，例如选择产品模式、准备配置或启动必需的网络接口：

```
import network

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
```

`boot.py` 必须返回，不应包含应用的永久循环。ESP-VISION 仅在 `boot.py` 完成后初始化 MicroPython USB 设备，因此阻塞或长时间运行的 `boot.py` 可能导致 USB REPL 和主机工具无法使用。

4.8.5 使用 main.py

`main.py` 应作为产品应用入口。建议将具体实现放在独立模块中，使启动策略与应用逻辑彼此独立：

```
import sys
import my_app

try:
    my_app.main()
except KeyboardInterrupt:
    raise
except Exception as error:
    print("Fatal application error:")
    sys.print_exception(error)
```

允许 `KeyboardInterrupt` 继续抛出，可以使用 `Ctrl-C` 停止应用并进入友好 REPL。产品应用需要自动恢复时，也可以记录异常后调用 `machine.reset()`，但无条件复位循环会增加开发和故障诊断难度。

ESP-VISION 默认 `main.py` 会输出开发板就绪信息并在循环中休眠，使 VSCode 扩展和其他主机工具能够取得控制权。可按 `Ctrl-C` 中断该脚本并进入 REPL，或使用产品入口替换 `/main.py`。

4.8.6 REPL 与故障恢复

`main.py` 正常退出或抛出未捕获异常后，控制权会转交给友好 REPL。`Ctrl-C` 会向正在运行的 Python 脚本注入 `KeyboardInterrupt`，在友好 REPL 中按 `Ctrl-D` 则会启动软复位。

如果应用导致设备无法正常启动，可以连接 REPL，使用 `Ctrl-C` 中断应用，然后重命名或删除启动文件：

```
import os

print(os.listdir("/"))
os.rename("/main.py", "/main.disabled.py")
# 使用 Ctrl-D 重新启动，并避免运行之前的 main.py。
```

如果无法恢复 REPL，应在仓库根目录擦除并重新烧录开发板。`erase-flash` 会清除设备的全部 Flash，包括内部文件系统和用户数据：

```
idf.py --board <BOARD> -p <PORT> erase-flash
idf.py --board <BOARD> -p <PORT> flash monitor
```

不执行 `erase-flash` 的重新烧录通常会保留数据文件系统，因此也可能保留存在问题的 `boot.py` 或 `main.py`。

Chapter 5

存储、文件系统与数据保护

本章统一规定固件、Python 程序、Device Core、IDE/Agent 客户端和恢复工具的存储契约，目标是保护用户文件，并让破坏性恢复由用户决定。运行时诊断和恢复 API 见*storage – 存储诊断与确认恢复*。

5.1 存储介质与文件系统所有权

表 1: 存储职责

介质	当前用途	规范
内部 Flash `ffat`	FAT；Python 路径 `/` 及别名 `/flash`，IDF 内部路径 `/flash`；支持 USB device 的开发板通过 MSC LUN 0 暴露	ESP-IDF 唯一持有文件系统。Python VFS 与 MSC 访问相同数据，不是两份副本。这个命名的 data/FAT 分区可按已配置的自动初始化策略格式化，或由用户明确确认后执行恢复。
旧 Flash `vfs`	挂载已有 FAT；也可能包含旧固件的其他文件系统	自动启动和在线恢复都只允许挂载。未知格式或 LittleFS 卷不能被当成可丢弃的 FAT。
SD 卡	FAT，路径 `/sdcard`；支持 SD 的 USB device 开发板通过 MSC LUN 1 暴露	ESP-IDF 唯一持有文件系统。缺卡、坏卡不得阻止 Flash、VM 和通信工作。自动策略可以初始化无法识别的卡；显式 Python SD 格式化仍不支持。其他恢复应备份后在主机上明确选择目标卡处理。
NVS	网络配置、写入所有权和格式化事务记录等小型状态	与用户文件分离。NVS 错误不授权擦除 NVS 或格式化数据卷。Flash 文件备份不包含 NVS。
配有外部 NAND 的开发板	当前尚未暴露为 ESP-VISION 文件系统，包括 ESP-Mosaico 的板载 NAND	后端完成验收前应报告不支持，不能当作 SD，也不能直接套用原始 NOR `ffat` 驱动；后续支持必须处理 NAND 坏块、ECC 和恢复问题。
冻结模块 / 只读 RAM VFS	固件模块或临时模型资源	不是可写用户卷，也不是持久备份。RAM 内容重启即失，冻结内容属于固件镜像。

开发板分区表是偏移和大小的唯一依据。不能根据模组版本、旧示例或其他开发板上同名分区推断破坏性操作范围。Board Manager 驱动和相机、模型等使用方应复用公共存储管理器，不能在同一介质上再挂一套文件系统。

`/.esp_vision_disk` 只是创建默认文件的标记。它和已保存的 NVS 状态都不是备份，也不构成新的格式化授权。格式化授权来自下面的板级策略或用户明确作出的恢复决定。

当前 Flash 后端使用项目自有的原始块驱动：512 字节逻辑扇区，4096 字节擦除块读改写。它既不是日志文件系统，也不是 ESP-IDF wear levelling 卷。高频日志、图片和录像应使用合适的可移除介质并限制保留量，同时考虑剩余空间、擦写寿命和元数据开销。

5.2 可选卷与默认自动挂载

Flash 和 SD 是两个独立的可选卷。默认启动时尝试挂载 Flash，并在支持 SD 的开发板上挂载可用的卡。Flash 缺失 / 挂载失败不阻止 SD 挂载；SD 缺卡 / 挂载失败不阻止 Flash 挂载。初始化遵循下面的独立策略，普通挂载错误不触发初始化。两个卷均不可用时，VM 和控制通道仍可使用虚拟根目录与诊断功能，启动脚本不会自动改从 SD 执行。

各开发板可在 `boards/<BOARD>/boardconfig.h` 中分别覆盖默认值：

```
#define ESP_VISION_STORAGE_AUTO_MOUNT_FLASH (1)
#define ESP_VISION_STORAGE_AUTO_MOUNT_SD    (1)
#define ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM (1)
#define ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM    (0)
```

将任一 `AUTO_MOUNT` 值设为 `0` 可关闭该介质的自动挂载。这是相互独立的编译期策略，不是 Python 持久配置，也不改变硬件能力。SD 仍需要 `MICROPY_HW_ENABLE_SDCARD` 和支持的 SDMMC 后端。介质可用但关闭自动挂载时，报告 `unmounted` / `STORAGE_NOT_MOUNTED`；即使开启自动初始化，显式调用 `storage.retry("flash")` 或 `storage.retry("sd")` 也只挂载、绝不格式化。检查结果后，由用户选择软复位来发布新的 Python 挂载。启动时不可用的 SD 卡仅在开启自动 SD 挂载时于后续 VM 启动重试；已挂载的物理文件系统在软复位期间保留。

`ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM` 默认是 `1`；`ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM` 默认是 `0`。自动挂载时，只有对应介质的策略开启，FatFs 精确返回 `FR_NO_FILESYSTEM` 才授权格式化并重新挂载。因此内部 `ffat` 默认允许初始化；无法识别的 SD 卡默认保留，只有显式开启 SD 策略后才允许初始化。非空出厂镜像、其他文件系统或损坏的 FAT 都可能返回此结果，它不能证明区域内没有有价值的数据。开启任一策略都表示接受无需运行时确认便擦除这些内容；需要保留未知数据时，应将对应策略保持为 `0`。旧 `vfs` 无论这两个选项如何都保持只挂载。

已有可挂载 FAT 会保留。`FR_INT_ERR`、I/O / 未就绪 / 写保护错误、缺少介质、未关闭句柄以及被占用 / 隔离的写入方均不授权自动格式化。关闭某介质的自动挂载后，不会自动初始化它。`storage.retry()` 和写入方交接时的文件系统刷新始终只挂载。破坏性写入前，日志会明确标识由策略触发的初始化；初始化失败返回错误，不重启、不导致 crash。

内部 Flash 在 NVS 中持久保存 PENDING / READY 格式化事务；即使开启 Flash 策略，中断的事务也阻止继续自动格式化，需要用户重新明确选择恢复。SD 当前没有持久格式化日志或同等的中断保护。若显式开启 SD 自动初始化，断电后下次挂载返回 `FR_NO_FILESYSTEM` 时可能再次初始化。无论策略开关如何，部分创建的 FAT 都可能看起来可读，这不能证明断电恢复成功。应保留经校验的设备外备份；默认 SD 策略会保留无法识别的内容，而非格式化它们。

Flash 上的 `/main.py` 和 `/flash/main.py` 指向同一文件。别名只是连接到已有 IDF 文件系统的第二个 Python VFS bridge，不会再挂一套 FAT、复制文件或增加 USB 磁盘。相对路径跟随当前选中 bridge 的工作目录。`os.rename()` 的两个参数应使用一致的前缀：MicroPython 拒绝不同 VFS 挂载点之间的重命名，包括 `/` 与 `/flash`。`/flash` 保留为挂载点；已有的物理 `flash` 目录会保留，可通过 `/flash/flash` 访问。`/flash/sdcard` 指向 Flash 上的目录，不是位于 `/sdcard` 的独立 SD 卷。

备份和恢复工具必须按卷识别数据，不能把 `/` 和 `/flash` 当作两份副本或递归进入虚拟挂载点。建议用 `/flash` 作为明确的 Flash 备份根目录，另行备份 `/sdcard`。

5.3 两个卷在线时的 USB 行为

默认卷标为 `ESP_FLASH` 和 `ESP_SDCARD`。自动挂载仅为未命名的卷设置默认名，保留自定义卷标；卷标错误不授权格式化。USB 配置、命名限制、接口字符串和主机盘符见*USB 接口、配置与磁盘命名*。

开启原生 USB device MSC 的开发板 / 接口使用一个 MSC 接口暴露固定逻辑单元：LUN 0 是内部 Flash，支持 SD 的开发板再提供 LUN 1。两个卷都已挂载、健康且写入方准入允许主机访问时，主机可以枚举两个独立介质。盘符、主机挂载路径和空槽显示方式由操作系统决定。不支持 SD 的开发板只提供 Flash 槽位。USB Serial/JTAG 调试接口不提供这些 MSC 磁盘。

某介质缺失、未挂载或不可用时，只让对应槽位报告未就绪；Flash 不可用时，SD 不会改成 LUN 0。关闭自动挂载只推迟该槽位就绪，不影响另一卷或整个 USB 接口。`/flash` 不增加 LUN。弹出和写入方交

接按卷进行：Python / IDE 写入前，应安全弹出所选操作系统卷并核对其所有权；看到两个磁盘不表示主机与 VM 可以同时写入任一共享文件系统。

5.4 故障原则：保留数据，保持控制通道可用

挂载、分配内存、NVS、写入、关闭、同步和卡就绪状态等失败，都应作为可恢复的存储条件处理。存储错误路径必须返回状态或错误，必要时关闭 I/O 准入，并保持 REPL、会话管理和网络 / 控制通信可用。不得通过致命断言、abort、crash 或重启掩盖失败。Python 参数异常和操作返回 `ok: False` 不等于设备 crash。

挂载失败本身不构成格式化授权，自动处理的例外是上述仅适用于精确 `FR_NO_FILESYSTEM` 的独立板级策略，默认 Flash 开启、SD 关闭。关闭某介质的策略后，其空白分区、出厂 / 未知数据和损坏卷都保持原样。NVS 记录缺失不会独立授权格式化，NVS 错误也不能绕过 Flash 事务保护。`storage.retry()` 始终不创建文件系统、不擦除区域、不复位设备。

表 2: 诊断与用户选择

状态	含义	用户决定的下一步
`ready`	文件系统已挂载；写入权限仍需单独满足所有权规则	按规则访问文件，并维护经过验证的设备外备份。
`blank`	完整读取后确认分区全部已擦除	确认这就是预期的空卷后，选择初始化。
`unmounted`	可用介质的自动挂载被关闭	如需挂载，可使用 `storage.retry()`；这不是格式化请求。
`unrecognized`	无法挂载 FAT；可能是出厂资源、其他文件系统，也可能是损坏的用户数据	先保留 / 导出所需数据、核对布局，再选择重试、离线恢复或确认格式化。
`absent` / `unsupported`	介质缺失或能力不可用	检查卡和开发板能力；不能向用户推荐格式化不存在的硬件。
`io_error` / `unavailable`	驱动、挂载或资源错误	查看错误码、供电、连接和可用资源，处理原因后再重试。
`state_error`	NVS 格式化事务元数据不可读或校验失败	保留 NVS 和数据卷用于诊断，不能通过清除历史绕过保护。
`format_interrupte`	格式化未能完成	Flash 会持久保存此保护并要求重新决定；SD 没有同等持久日志。重启前应检查介质和板级策略。
写入方占用 / 隔离	其他写入方持有介质，或交接结果不确定	完成经过验证的交接，或调查隔离原因；格式化请求不覆盖所有权限制。

`storage.status()` 返回状态、稳定错误码、可读提示、挂载 / 存在 / 能力标志、原生错误、已知的 FatFs 结果和 Flash 几何信息。Device Core 的 `inspect({ fields: ['storage'] })` 保留原有写入方字段，并增加 `fs_state`、`fs_error`、`fatfs_result` 和 `format_supported`。这些字段是状态快照，不是销毁数据的授权。控制通信启动后会输出可执行的存储诊断；向未挂载卷上传文件时，原生上传返回 `STORAGE_UNAVAILABLE` 和恢复建议，而不是含糊的文件打开失败。

前端必须显示受影响卷、原因和数据影响，在显式破坏性操作前提供检查、重试和导出选项，并等待用户主动确认手动格式化。Agent 和无人值守任务不能把超时、重连、泛化的“修好错误”请求或出厂数据解释成新的运行时确认。固件已配置的自动初始化属于独立启动策略。本次改动提供状态、日志、错误和显式 REPL API，不代表所有既有 IDE / Web UI 已有备份向导或格式化按钮。

5.5 单写入方、文件发布与断电

EV-MUX 会话所有权不等于文件系统写入所有权。Python / IDE 写入、备份、恢复或卸载前，应按*交互式 Python、脚本与文件操作* 中的存储写入方交接契约与操作系统的 MSC 访问协调：安全弹出主机卷，停止应用和后台写入，排空句柄和写入，再核对实际所有权。结果不确定的交接应跨重启保持隔离。旧版兼容模式不能作为已经验收的并发写入保证；仅连接 SDK 不会让操作系统和 Python 同时写入变安全。

未挂载 Flash 对 MSC 不就绪，格式化事务完成前不会发布给主机。如果后续 FAT 变得无法识别且自动初始化策略仍开启，事务保护不能保证保留后来写入的文件。SD 卡独立判断就绪状态，其格式化没有持久事务保护。不能忽略关闭失败、仍持有句柄的上传或写入隔离来强行完成格式化。

原生二进制上传使用临时文件、预期长度和 SHA-256、同步 / 关闭和最终发布；普通替换失败会尝试从旧文件回滚。这能防护传输错误和普通发布失败，不能保证 FAT 多次目录更新之间每一种断电情况都安全。同一介质上的临时文件和回滚文件不是独立备份。`fsync` 和成功的传输响应，不会把 FAT 或原始 Flash 擦除块更新变成有日志、断电原子的事务。最终响应丢失后，应先检查目标文件，再决定是否重试。

重要应用数据应写到新文件，校验并关闭 / 同步后，再发布且保留已验证的上一代，直到确认成功。限制日志大小，为新旧两代文件预留空间。写入或关闭失败后应停止后续写入并上报原始错误；反复重试或删除其他文件不等于恢复。

5.6 备份与恢复规范

备份必须位于不同的故障域，通常是主机电脑或独立保留的其他介质。同一 Flash / 卡上的备份目录和旧文件重命名，不能抵御整个介质损坏、误格式化或介质故障。

1. 确认物理设备、开发板、固件和选定卷。停止脚本及所有后台写入，安全弹出 MSC 并核对所有权后，再复制一致的文件系统快照。若无法停止写入，只能把结果标为未验证副本，不能称为一致备份。
2. 导出用户脚本、隐藏配置、所需包、模型元数据和其他必要文件。在主机清单中记录路径、大小、SHA-256、设备身份和文件系统 / 布局信息。保留多个备份版本，不能覆盖最后一份已知可用备份。
3. 根据清单验证下载的字节，并检查预期文件是否齐全。下载结束不等于备份可用。应定期在合适的备用介质 / 设备上验证恢复。
4. 恢复前检查目标身份、布局兼容性和剩余空间。通过正常的校验上传及所有权流程导入，核对长度、哈希和回读；验证成功前保留备份。用户检查之前，不能自动执行恢复的启动脚本。

当前文件 API 支持通过 Device Core 导出 / 导入选定文件，也支持安全交接后的主机 MSC 复制。本次固件 API 未实现完整的清单式备份 / 恢复向导。主机实现此功能时，应遵守上述规范，区分成功、部分完成、失败和未验证副本，不能把部分复制标成已验证备份。

文件系统无法挂载时，可以导出指定原始分区镜像保留离线恢复证据。进入 ROM 模式会中断运行；用户明确请求后，可按已安装分区表中的实际偏移和长度读取：

```
esptool --chip <CHIP> --port <PORT> read-flash <OFFSET> <LENGTH> <HOST_BACKUP.bin>
```

原始镜像不是已经验证的文件级备份，恢复时可能需要匹配的布局和加密上下文。NVS / 系统状态备份应单独明确范围，且可能包含凭据，应妥善保护文件。日常开发烧录不自动备份或恢复固件，也不自动读取整片 Flash；只有用户明确要求时才执行这些操作。

5.7 显式恢复未挂载的 Flash 卷

先进行不写盘的检查：

```
import storage
print(storage.status("flash"))
print(storage.status("sd"))
print(storage.retry("flash"))  # 只重试挂载，绝不格式化
```

如果 `retry` 返回 `ok: True` 和 `soft_reset_required: True`，用户可选择软复位以发布新的 Python 挂载。失败时保留运行环境和诊断状态。

只有在用户确认目标 Flash 卷、保留所需文件，并同意擦除后，才执行以下独立的破坏性操作：

```
result = storage.format("flash", confirm=True)
print(result)
```

必须传入布尔值 `True`；省略、假值或真值字符串都会被拒绝。恢复只格式化未挂载的 data/FAT `ffat`，并先检查打开句柄和写入准入。已挂载 / 可读取卷、旧 `vfs`、在线 SD 格式化以及不可用 / 隔离写入方都会被拒绝。硬件 I/O 错误不会触发格式化。范围来自设备分区表，不擦除 NVS、固件或 SD。

格式化前，会先提交与分区绑定的 NVS `PENDING` 记录。只有格式化、重新挂载和 `READY` 持久提交全部成功，才发布该卷。提交、格式化或挂载失败时保持不可用；中断的 `PENDING` 不会自动恢复，即使残留 FAT 头看起来可以读取，也必须由用户再次明确决定。清除 NVS 不是绕过该状态的合规恢复方式。

检查 `ok` 和 `operation_error`。成功且返回 `soft_reset_required` 时，用户可显式软复位，以发布 Python 挂载并创建缺失的默认文件：

```
import machine
machine.soft_reset()
```

复位是用户选择的后续步骤，不是存储故障的自动响应。恢复应用之前，应重新检查存储，并验证创建、上传、回读以及重启后的文件持久性。失败或响应丢失意味着完成状态可能不确定；再次执行破坏性操作前必须先检查状态。

5.8 固件安装与验收

普通固件升级保留用户文件和 NVS；完整安装、布局改变和整片擦除具有不同的数据影响，主机必须在用户选择前说明。不能因布局不一致而静默把升级改为安装，或格式化数据卷。升级与安装契约见*应用 API 完整参考*。

验收应覆盖空白和残留数据介质、有效旧 FAT、损坏 FAT、缺卡 / 拔卡、NVS 错误、空间耗尽、写入 / 关闭 / 同步失败、格式化中断及所有权交接不确定等情况。必须验证已开启且精确匹配 `FR_NO_FILESYSTEM` 的初始化策略以外，挂载失败不会自动擦除 / 格式化 / 写入 NVS；对策略授权的初始化，验证各介质默认值及 Flash 的 PENDING / READY 事务保护。同时验证 REPL 和会话仍可用，提示可定位，显式确认保护有效，其他卷不受影响。主机故障注入不等于真实断电或特定模组实机验收，应报告准确提交、介质、开发板、接口及实际硬件证据。

Chapter 6

API 参考

ESP-VISION 提供摄像头采集、图像处理、画面显示、视频编码与传输以及 AI 推理等 Python API。开发者还可以直接使用固件中启用的 MicroPython 标准 API，完成硬件控制、网络通信、文件访问、系统管理及通用应用开发。ESP-VISION 在适用范围内兼容常用的 OpenMV 模块名称，包括 `sensor` 和 `image`，并进一步提供 `display`、`espdl` 和 `tflite` 等模块。

ESP-VISION 模块页面与 `stubs/` 下的类型存根保持同步，存根同样可用于 IDE 补全；其可用性在文档构建时根据 `micropython.cmake` 自动推导。*MicroPython API* 页面索引 ESP-VISION 固件从固定 MicroPython 版本沿用的语言、标准库、硬件、网络、文件系统和运行时 API。

6.1 MicroPython API

ESP-VISION 基于 MicroPython ESP32 port 构建，并直接沿用其语言、标准库、硬件控制、网络、文件系统和运行时 API。这些 API 可以在同一脚本中与 `sensor`、`image`、`display`、`espdl` 及其他 ESP-VISION 模块组合使用；ESP-VISION 不会对其进行二次封装或重命名。

6.1.1 文档基线

ESP-VISION 固定使用 MicroPython v1.28.0。应以 MicroPython v1.28.0 文档 为权威 API 参考，重点参阅 MicroPython 库 和 ESP32 快速参考。MicroPython 面向资源受限设备实现 CPython 功能子集，因此不能将 CPython 文档视为模块内容或行为的精确说明。

6.1.2 语言与内置 API

ESP-VISION 沿用 MicroPython 语言，以及内置类型、函数、异常、迭代协议、上下文管理器、类和异步语法。详细说明请参阅 MicroPython 语言与实现 和 内置功能。

使用异常和上下文管理器可以提高长时间运行的视觉应用的可恢复性，并确保文件能够确定性关闭：

```
try:
    with open("/sdcard/result.txt", "w") as output:
        output.write("capture complete\n")
except OSError as error:
    print("storage error:", error)
```

6.1.3 标准库与微型库

常用的沿用模块包括 array、asyncio、binascii、cmath、collections、errno、gc、gzip、hashlib、heapq、io、json、marshal、math、os、platform、random、re、select、socket、struct、sys、time、weakref、zlib 和 _thread。ESP-VISION 的开发板 manifest 还会将 MicroPython 的 asyncio 软件包冻结到固件中。

各模块的类和函数请参阅 MicroPython 库索引。上游文档中存在的模块仍可能因所选固件配置而被裁减或省略。

6.1.4 JSON 与时间示例

```
import json, time

result = {
    "timestamp_ms": time.ticks_ms(),
    "objects": 3,
    "confidence": 0.91,
}
payload = json.dumps(result)
print(payload)
```

time.ticks_ms() 适合测量时间间隔和生成本地运行时间戳；计算两个 tick 值之差时应使用 time.ticks_diff()，以正确处理计数回绕。json 可用于在存储或网络传输前序列化推理结果。

6.1.5 硬件与 ESP32 API

沿用的 machine 模块提供 MCU 控制和外设类，包括 Pin、Signal、ADC、ADCBlock、PWM、I2C、SoftI2C、SPI、SoftSPI、UART、Timer、RTC、WDT、SDCard，以及芯片支持时可用的 DAC、I2S 和 I2CTarget。引脚分配和外设冲突取决于开发板，因此将这些 API 与摄像头、显示、存储或编解码硬件组合使用前，应查阅开发板原理图。

沿用的 esp 和 esp32 模块提供 ESP32 port 专用的系统和外设功能。esp32.PCNT、esp32.RMT 及其他 SoC 专用模块中各个类的可用性取决于所选芯片和 ESP-IDF 版本。

6.1.6 GPIO 与 PWM 示例

```
from machine import Pin, PWM

led = Pin(LED_GPIO, Pin.OUT)
led.value(1)

pwm = PWM(Pin(PWM_GPIO), freq=1000, duty_u16=32768)
pwm.duty_u16(49152)
pwm.deinit()
```

应将 LED_GPIO 和 PWM_GPIO 替换为所选开发板上的空闲引脚。摄像头、显示、存储、USB 和编解码外设可能已经占用部分 GPIO，因此使用前应检查原理图和板级配置。

6.1.7 网络 API

沿用的 network、socket 和 select 模块提供网络接口管理及 TCP/UDP 通信能力。当前产品开发板配置均启用了 network.WLAN，但实际使用仍需要兼容的网络硬件和固件配置。

当前开发板配置关闭了 bluetooth 和 espnow。SSL/TLS、cryptolib、WebSocket 和 socket event 支持取决于构建固件使用的 ESP-IDF 版本。ESP-VISION EV-MUX 运行态不支持传统 MicroPython WebREPL，即

使构建包含 _webrepl，其终端与文件操作也不经过 EV-MUX 的 owner 授权。网络控制请使用*应用 API 完整参考* 的 TCP 连接；维护中的支持摘要见*芯片与开发板支持*。

6.1.8 连接 Wi-Fi

```
import network, time

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect("ssid", "password")

for _ in range(100):
    if wlan.isconnected():
        break
    time.sleep_ms(100)

if not wlan.isconnected():
    raise OSError("Wi-Fi connection failed")
print("address:", wlan.ifconfig()[0])
```

Wi-Fi 需要兼容的硬件和开发板配置。产品代码应设置超时、避免打印凭据，并处理重新连接，而不是无限期等待。

6.1.9 文件系统与运行时 API

ESP-VISION 沿用 MicroPython 的虚拟文件系统和流行为，包括 os、io、vfs 和内置 open() 函数。当相应分区与硬件可用时，开发板启动代码会挂载 /flash、/sdcard 等产品存储。

运行时服务继续通过 micropython、gc、sys、time 和 _thread 提供。ESP-VISION 启用了带 GIL 的 MicroPython 线程；除非 API 明确说明支持并发使用，视觉对象和硬件资源仍应作为共享资源处理。

6.1.10 列出存储并检查内存

```
import gc, os

print("flash:", os.listdir("/flash"))
if "sdcard" in os.listdir("/"):
    print("sdcard:", os.listdir("/sdcard"))

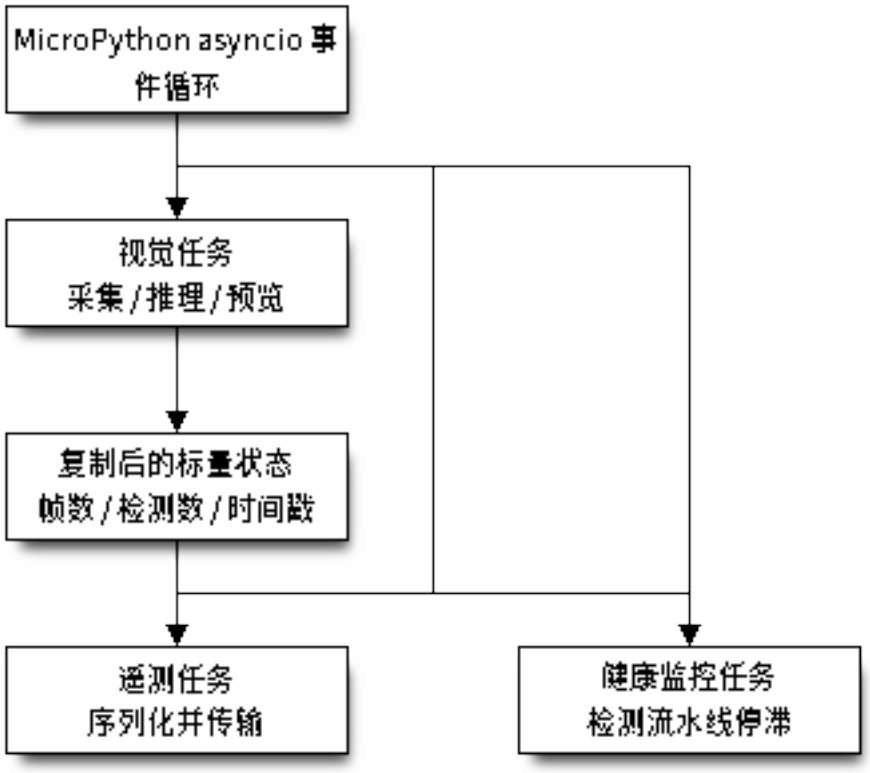
gc.collect()
print("free heap:", gc.mem_free())
```

使用可选挂载点前应先检查其是否存在。加载大型模型或分配图像历史缓冲区前可以调用 gc.collect()，但在每帧循环中反复强制回收可能降低吞吐量。

6.1.11 组织复杂应用

对于熟悉 ESP-IDF 的开发者，asyncio task 是协作调度的 Python 协程，而不是 FreeRTOS task。多个协程共享同一个 Python 执行上下文，并且仅在执行 await 时切换。控制逻辑、网络 I/O、状态上报和定时工作应优先采用这种结构，因为它可以避免并发访问摄像头、模型、显示和帧缓冲对象。

以下结构由一个协程独占视觉流水线，遥测与健康监控协程只读取复制后的标量结果：




```
import asyncio
import espdl
import json
import sensor
import time

MODEL = "/sdcard/espdet_pico_224_224_face.espdl"

state = {
    "frames": 0,
    "detections": 0,
    "last_frame_ms": time.ticks_ms(),
}
state_lock = asyncio.Lock()

async def vision_task(detector):
    while True:
        img = sensor.snapshot()
        results = detector.detect(img)
        for x, y, w, h, score, category in results:
            img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
        img.flush()

        async with state_lock:
            state["frames"] += 1
            state["detections"] = len(results)
            state["last_frame_ms"] = time.ticks_ms()

        # 每帧结束后允许已就绪的控制与网络任务运行。
```

(续下页)

(接上页)

```
        await asyncio.sleep_ms(0)

async def telemetry_task():
    while True:
        await asyncio.sleep_ms(1000)
        async with state_lock:
            payload = json.dumps({
                "frames": state["frames"],
                "detections": state["detections"],
            })
        print("telemetry:", payload)
        # 实际应用中应替换为非阻塞 socket 传输。

async def health_task():
    while True:
        await asyncio.sleep_ms(500)
        async with state_lock:
            age = time.ticks_diff(
                time.ticks_ms(), state["last_frame_ms"]
            )
        if age > 3000:
            print("warning: vision pipeline stalled")

async def main():
    sensor.reset()
    sensor.set_pixformat(sensor.RGB565)
    sensor.set_framesize(sensor.QVGA)
    sensor.skip_frames(time=1000)
    detector = espdl.ESPDet(MODEL, score=0.5, nms=0.7)

    vision = asyncio.create_task(vision_task(detector))
    telemetry = asyncio.create_task(telemetry_task())
    health = asyncio.create_task(health_task())
    try:
        await asyncio.gather(vision, telemetry, health)
    finally:
        vision.cancel()
        telemetry.cancel()
        health.cancel()
        detector.deinit()

asyncio.run(main())
```

`sensor.snapshot()`、推理、图像处理和 `img.flush()` 等调用都是同步操作。执行其中任一调用时，其他协程不能运行；显式调用 `await asyncio.sleep_ms(0)` 可在帧与帧之间提供调度点。如果某个网络操作可能阻塞，应使用 `asyncio` stream 或非阻塞 socket，而不是长时间执行阻塞调用。

6.1.12 线程与原生任务

沿用的 `_thread.start_new_thread()` API 会创建另一个 FreeRTOS task，但它作为应用接口并不等同于 `xTaskCreatePinnedToCore()`。Python 在此不提供任务优先级或核亲和性参数；MicroPython 线程固定在 `MP_TASK_COREID` 上，并通过 GIL 共享解释器。可以使用 `_thread.stack_size()` 配置后续创建线程所分配的栈。

仅当某个阻塞操作无法集成到 `asyncio` 时才应使用 `_thread`，使用 `_thread.allocate_lock()` 保护共享 Python 数据；除非 API 明确说明线程安全，否则摄像头、显示、编解码器、模型和帧缓冲应始终由同一个线程持有。IRQ 或其他线程可以通过 `asyncio.ThreadSafeFlag` 通知事件循环，而无需直接操作视觉对象。

如果应用需要确定的时延、明确的 FreeRTOS 优先级或核亲和性、ISR 到任务的通知，或者需要在 Python

持有 GIL 时仍持续执行的工作，应将工作任务实现为 ESP-IDF C/C++ 组件，并向 Python 暴露边界清晰的 API。用 ESP-IDF 的思维理解：应用状态机使用 `asyncio` task，特殊的阻塞集成使用 `_thread`，实时服务使用原生 task。

6.1.13 检查固件中的可用性

标准 MicroPython API 的可用性由 `overlay/micropython/ports/esp32/mpconfigport.h`、所选开发板的 `mpconfigboard.h` 与 `mpconfigboard.cmake`、ESP-IDF 版本条件和 SoC 能力宏共同决定。由于可用性并非只取决于芯片，最终应以实际固件检查结果为准：

```
help("modules")
import machine
help(machine)
```

应用需要跨多块 ESP-VISION 开发板运行时，应使用 `hasattr()` 或受保护的 import。产品固件需要移除或增加沿用 API 时，请按照客制化固件功能 操作。

参见：

使用标准 MicroPython API 的可运行示例包括 `example/00-HelloWorld/helloworld.py`（首个脚本）和 `example/06-Peripherals/00-Storage/sdcard.py`（需要 SD 卡）。`example/07-Network/00-WebREPL/webrepl.py` 用于说明传统 WebREPL 不受支持的边界，会在改变网络前退出，不是 ESP-VISION 上可运行的网络终端。

6.2 sensor –摄像头

`sensor` 模块用于控制摄像头并采集图像帧。它的接口与 OpenMV 的 `sensor` 保持一致，因此大多数 OpenMV 脚本无需改动即可移植。

6.2.1 摄像头初始化与连续采集

典型流程是复位传感器、选择像素格式与分辨率、等待自动曝光和自动白平衡稳定，然后连续采集图像：

```
import sensor

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=2000)

while True:
    img = sensor.snapshot()
```

`sensor.reset()` 会应用开发板的默认摄像头配置。`RGB565` 适用于显示、绘图和大多数 AI 工作流，`GRAYSCALE` 则可降低许多检测算法的内存占用和处理开销。`sensor.snapshot()` 返回由可复用帧缓冲承载的*image.Image*，因此需要在下一次采集后继续保持图像内容不变时，应先复制该图像。也可以传入一个足以容纳当前帧的可写缓冲区：`sensor.snapshot(buffer)`；返回的图像会保留该缓冲区的引用。

6.2.2 图像方向与摄像头状态

摄像头传感器的实际安装方向可能导致画面翻转，可通过水平镜像和垂直翻转进行校正。`status()` 返回当前分辨率、像素格式、传感器 ID、图像方向和裁剪信息，可用于诊断：

```
import sensor

sensor.reset()
sensor.set_hmirror(True)
sensor.set_vflip(False)

info = sensor.status()
print("sensor id:", info["id"])
print("output:", info["width"], "x", info["height"])
print("ready:", info["ready"])
```

镜像和翻转设置会影响后续采集的图像。产品代码可在初始化后调用 `status()`，确认摄像头已经就绪，并验证协商得到的输出尺寸与处理链路一致。

6.2.3 暂时停止采集

不需要采集时可以停止摄像头流，并在下一次采集前重新启动：

```
sensor.shutdown(True)
# 执行不需要摄像头的任务。
sensor.shutdown(False)
sensor.skip_frames(n=3)
img = sensor.snapshot()
```

重新启动摄像头流后，如果曝光或输入帧队列需要重新稳定，应丢弃少量图像帧。

备注： `ESP32_P4X_EYE` 的默认 SPI 配置支持 QVGA 摄像头采集与 LCD 同时使用。SPI 图像传输需要 DMA 可用内存，因此提高摄像头分辨率或增大面板传输块/队列深度后，可能超出最大连续 DMA 空闲块，即使 `gc.mem_free()` 和 PSRAM 总可用量仍然很大。如果 `display.Display()` 或 `lcd.write()` 抛出 `ESP_ERR_NO_MEM`，先释放对 `sensor.snapshot()` 图像的引用并调用 `sensor.shutdown()`，再初始化和使用 LCD；之后调用 `lcd.deinit()`，再调用 `sensor.reset()` 恢复摄像头采集。

6.2.4 运行时控制 IPA

IPA 和 ISP 控制接口是可选固件功能。使用前可通过 `hasattr(sensor, "get_ipa")` 或 `hasattr(sensor, "get_isp")` 检查；固件未包含该功能时，这些方法不存在。

使用 `sensor.set_ipa(False)` 停止图像处理算法（IPA）控制器，使用 `sensor.set_ipa(True)` 重新启动。调用前需要初始化摄像头。`sensor.get_ipa()` 返回控制器的当前状态；不支持运行时控制时返回 `None`。对不支持的摄像头设置 IPA 会抛出 `OSError`。

```
sensor.reset()
sensor.skip_frames(time=2000)
if hasattr(sensor, "get_ipa") and sensor.get_ipa() is not None:
    sensor.set_ipa(False)
    print("IPA:", sensor.get_ipa())
    img = sensor.snapshot()
    img.flush()
    sensor.set_ipa(True)
```

切换 IPA 状态会短暂停止并重新启动采集，保留输出像素格式、分辨率、镜像和翻转设置。重复设置当前状态不会执行切换。`sensor.reset()` 以及 `sensor.shutdown(True)` 后重新启动摄像头会恢复默认 IPA 状态。

切换失败时摄像头不会退回原状态，而是被完全关闭：抛出 `OSError` 之前，采集缓冲、缩放器和摄像头设备都已释放。此时需调用 `sensor.reset()` 重新启动摄像头。

关闭 IPA 会停止其自动参数更新，但不会绕过 ISP 或清除已有调校参数，因此画面最初可能没有明显变化。对比时可以在预览中改变照度或光源颜色，再开启 IPA 并等待算法重新稳定。此开关只控制当前 IPA JSON 配置中声明且后端支持的算法。重新开启 IPA 会把 CCM、Gamma、锐化等配置中启用的模块交还给自动算法；手动值不会作为覆盖项保留。此开关不关闭图像传感器内部实现的自动曝光或自动白平衡。ESP32_P4X_EYE 的 OV2710 曝光和增益由传感器片上自动控制，不受 IPA 控制；停止 IPA 不会冻结这些参数，也不会使手动曝光或增益接口变为可用。

6.2.5 读取 IPA 统计信息

`sensor.get_ipa_stats(timeout=1000)` 在 IPA 运行期间读取一份新的 ISP 统计数据。使用前通过 `hasattr(sensor, "get_ipa_stats")` 检查固件是否提供接口，并初始化摄像头、开启 IPA。`timeout` 为 1 至 1000 毫秒的整数，超出范围抛出 `ValueError`。超时或摄像头不支持运行时 IPA 控制时返回 `None`；摄像头未初始化、IPA 已关闭或设备操作失败时抛出 `OSError`。

每次调用创建一个单条目的统计队列，返回前释放，超时也会清理。读取不会停止采集或修改调校参数，但会临时增加统计复制开销，并可能在超时期限内阻塞 Python，因此建议定期采样。两次读取之间继续调用 `sensor.snapshot()` 消费图像缓冲。统计序号与 Python 的 `snapshot()` 图像没有一一对应保证。

返回字典始终包含以下键。本次数据包中无有效标记的统计块为 `None`，不会拼接不同数据包的统计值。

表 1: IPA 统计字段

键	含义
`seq`	64 位统计序号；摄像头或 IPA 重新初始化后可能重新计数。
`flags`	IPA 有效位：AWB=1、AE=2、HIST=4、SHARPEN=8、AF=16、AWB_SUBWIN=32。AE/AWB 位序与原始 ESP Video 标记不同。
`ae`	5×5 亮度网格，按 `[x][y]` 索引。
`awb`	包含 `counted`（白点计数）、`sum_r`、`sum_g`、`sum_b` 的字典。有效结果的白点计数也可能为零。
`awb_subwin`	按 `[x][y]` 索引的 5×5 网格，各单元字段与 `awb` 相同。需要后端、芯片及统计配置支持。
`hist`	按硬件直方图区间顺序排列的 16 个像素计数；区间边界由当前 ISP 配置决定。
`sharpen`	高频像素最大值。
`af`	三个窗口的字典列表，各包含 `definition`（清晰度）和 `luminance`（亮度）。这些是对焦统计值，不是镜头位置。

```
sensor.reset()
sensor.set_ipa(True)
sensor.skip_frames(time=2000)
sensor.snapshot()
stats = sensor.get_ipa_stats(timeout=200)
if stats is not None:
    print(stats["seq"], stats["ae"], stats["awb"])
```

对于有效且 `counted > 0` 的 `awb`，各通道总和除以 `counted` 可得到对应统计源的通道均值。硬件白点统计只包含筛选后的样本，并非整帧 RGB 均值；OV2710 后端也可能提供传感器增益前的通道平均值，此时 `counted=1`。两种来源都不是最终 `image.Image` 的通道均值。

`example/01-Camera/04-IPA/ipa_stats.py` 演示预览期间定期读取统计。该接口返回 IPA 消费的 ISP 输入统计，不包含算法推算的色温、收敛曝光等输出状态；驱动支持的参数可通过 `get_isp()` 回读。

6.2.6 手动控制 ISP

初始化后，`sensor.get_isp(block)` 返回驱动当前参数的字典；不支持该处理模块时返回 `None`。摄像头驱动没有实现读取接口，或固件针对的芯片版本不具备相应硬件时，都属于不支持：`lsc` 需要

ESP32-P4 v1.0 及以上，blc 需要 v3.0 及以上；而 SC2336 等传感器的曝光是只写的，因此 sensor.get_isp("exposure") 通常返回 None，但仍可通过 sensor.set_isp("exposure", value=...) 设置曝光。ESP32_P4X_EYE 使用的 OV2710 驱动没有实现手动 exposure 或 pixel_gain：读取返回 None，写入抛出包含 ESP_ERR_NOT_SUPPORTED 的 OSError。其他传感器的只写接口则不能根据读取不支持来判断写入能力。手动写入前先调用 sensor.set_ipa(False)。sensor.set_isp(block, **params) 修改一个处理模块，未指定的字段沿用当前配置。IPA 运行期间写入会抛出 OSError，避免算法覆盖手动参数。

```
sensor.reset()
sensor.skip_frames(time=2000)
sensor.set_ipa(False)

print(sensor.get_isp("wb"))
sensor.set_isp("wb", enable=False)
sensor.set_isp("ccm", enable=False)
sensor.set_isp("gamma", enable=False)
sensor.snapshot().flush()

sensor.set_isp("wb", enable=True, rg=1.8, bg=1.4)
# 仅用于支持手动曝光和增益的传感器（不含 OV2710）：
# sensor.set_isp("exposure", value=10000)  # 单位：微秒，按 100 微秒取整。
# sensor.set_isp("pixel_gain", value=2.0)  # 增益倍数。

sensor.set_ipa(True)
```

get_isp() 读取驱动配置，不是硬件提交确认，也不是返回图像的实测参数。get_isp("wb") 只报告 ISP 白平衡处理模块，不报告 OV2710 内部 AWB 增益，因此 enable=False, rg=1.0, bg=1.0 可以与传感器内部 AWB 同时存在。对于没有独立白平衡处理模块的 P4 版本，驱动将手动白平衡增益与 CCM 合成。

采集缓冲中可能已有写入前生成的图像，单纯休眠不会消费这些旧帧。评估修改时应持续采集，或先调用 sensor.skip_frames(n=3) 消费排队的图像。丢弃旧帧并不构成硬件提交屏障，也不保证传感器自动控制已经稳定；重复写入或固定休眠不能可靠确定图像对应哪次配置。

处理模块名与字段沿用调参参数的命名：

表 2: ISP 参数

模块	字段
exposure	value：曝光时间，单位微秒，须为正数。设备控制项以 100 微秒为单位计数，因此写入会就近取整到 100 微秒，取整后还须符合传感器上报的范围与步长，可用上下限即由此决定。例如 SC2336 在 1280x720 30fps 下大致接受 200 微秒到 33.2 毫秒，步长为 100 微秒。
pixel_gain	value：正数增益倍数，在传感器范围内选择增益菜单中最接近的一项。
demosaic	enable、gradient_ratio：梯度比例，非负数，范围由驱动的定点数格式决定。
wb	enable、rg、bg：红、蓝通道增益倍数，范围为 0 到 1023/256。
lsc	enable、lsc_gain_size、gain_r、gain_gr、gain_gb、gain_b：四个增益列表，长度均须与 get_isp("lsc") 返回的长度一致，取值范围为 0 到 1023/256。长度由采集尺寸决定。
blc	enable、stretch_enable、top_left_offset、top_right_offset、bottom_left_offset、bottom_right_offset：黑电平偏移量，为 0 到 255 的整数。
bf	enable、level（2–20）、matrix：按行排列的九个整数，范围为 0 到 15。
sharpen	enable、h_thresh、l_thresh（0–255，低阈值不大于高阈值）、h_coeff、m_coeff（0 到 255/32）、matrix：按行排列的九个整数，范围为 0 到 31。
ccm	enable、matrix：按行排列的九个有限数值，严格位于 -4 到 4 之间。
gamma	enable、x、y：两个长度为 16 的整数列表，取值为 0 到 255。X 从隐含的零点开始递增，每段间隔必须为 2 的幂；最后一点须为 255，计算最后一段间隔时按 256 处理。

向已关闭的模块应用新参数时，应同时设置 `enable=True`。仅传入 `enable=False` 会关闭该处理模块，而不替换已保存的表数据。白平衡增益在关闭后仍然保留，再次打开时恢复。Gamma 读回红通道曲线；设置 `x` 或 `y` 会将合成后的曲线应用于三个通道，仅修改开关则保留各通道曲线。LSC 表复制到摄像头持有的存储中，在垃圾回收及 IPA 切换后仍然有效；在驱动接受新表之前，摄像头会保留已生效的旧表，但底层驱动可能在报告硬件错误之前就更新其缓存指针，因此设备写入失败时配置或硬件状态可能不确定；此类错误发生后应先重置摄像头再继续。定点数参数和传感器设置可能被驱动量化。

未知模块、无效数值及数组长度错误会在应用修改前被拒绝。不支持的写入会抛出 `OSError`。设备错误可能留下部分已应用的配置，此时可调用 `sensor.reset()` 重新初始化。

`example/01-Camera/04-IPA/isp_preview.py` 示例先配置输出格式，再关闭 IPA，然后持续预览，每 2 秒随机调整曝光、增益、白平衡增益和 CCM，并打印请求的曝光值及驱动读回的增益、白平衡增益和 CCM 参数，直到按 Ctrl-C 停止。可以修改 `UPDATE_INTERVAL_MS` 及各参数范围常量调整试验；当前传感器不接受的取值会被打印并跳过，不会中断预览。停止后，IPA 保持关闭并保留最后一次设置。

参见:

摄像头流水线 说明了一帧图像如何从图像传感器到达*image.Image*；图像模型 介绍像素格式与色彩空间。

可运行示例：`example/01-Camera/00-Snapshot`（采集并保存）与 `example/01-Camera/03-MJPEG`（Wi-Fi MJPEG 串流）。

6.2.7 Constants

`sensor.`**`GRAYSCALE`**

Pixel format constants accepted by set_pixformat().

`sensor.`**`RGB565`**

`sensor.`**`QQVGA`**

Frame size constants accepted by set_framesize().

`sensor.`**`QVGA`**

`sensor.`**`VGA`**

6.2.8 Functions

`sensor.`**`reset`**`()`

Reset and initialize the camera with the board default sensor configuration.

`sensor.`**`shutdown`**`(`*enable=...*`)`

Stop or restart the camera stream.

参数

`enable` –True shuts the camera down; False starts it again.

`sensor.`**`set_pixformat`**`(`*pixformat*`)`

Select the output pixel format.

参数

`pixformat` –sensor.GRAYSCALE or sensor.RGB565.

`sensor.`**`get_pixformat`**`()`

Return the current pixel format constant.

`sensor.`**`set_framesize`**`(`*framesize*`)`

Select the output frame size.

参数

`framesize` –sensor.QQVGA, sensor.QVGA, or sensor.VGA when supported by the board.

sensor.**get_framesize**()

Return the current frame size constant.

sensor.**width**()

Return the current output image width in pixels.

sensor.**height**()

Return the current output image height in pixels.

sensor.**get_id**()

Return the camera sensor ID.

sensor.**set_hmirror**(*enable*)

Mirror the camera image horizontally.

参数

enable –True enables horizontal mirror.

sensor.**get_hmirror**()

Return whether horizontal mirror is enabled.

sensor.**set_vflip**(*enable*)

Flip the camera image vertically.

参数

enable –True enables vertical flip.

sensor.**get_vflip**()

Return whether vertical flip is enabled.

sensor.**set_ipa**(*enable*)

Optional API: present only in firmware with IPA/ISP control enabled. Start or stop the IPA controller after camera initialization; briefly pauses capture. Existing ISP tuning is retained when disabled; enabling returns configured blocks to IPA. IPA controls only algorithms declared in its active JSON configuration and supported by the backend. Sensor-side automatic exposure/white balance is independent. Unsupported cameras raise OSError. OV2710 exposure/gain remain sensor-controlled when IPA is stopped; manual controls stay unsupported.

sensor.**get_ipa**()

Optional API: present only in firmware with IPA/ISP control enabled. Return the IPA controller state, or None when runtime control is unsupported. A supported camera must be initialized first.

sensor.**get_ipa_stats**(*timeout=...*)

Optional API: collect one new statistics packet while IPA is running. Uninitialized/stopped cameras or disabled IPA raise OSError. AE and AWB subwindow grids use [x][y]; unavailable blocks are None.

参数

timeout –1..1000 milliseconds; None means timeout or unsupported backend.

sensor.**get_isp**(*block*)

Optional API: present only in firmware with IPA/ISP control enabled. Read current ISP block parameters, or None if unsupported. Initialize the camera first. Blocks the camera driver or chip revision does not implement return None, including "exposure" on sensors whose driver makes it write-only. Readback is driver configuration, not a frame commit barrier or sensor-side AWB gains. OV2710 does not support manual exposure or pixel_gain reads or writes.

sensor.**set_isp**(*block*, ***params*)

Optional API: present only in firmware with IPA/ISP control enabled. Update one ISP block using keyword parameters; unspecified fields retain their values. Disable IPA before writing. Invalid parameters raise ValueError; device errors raise OSError. BF level is 2..20; use enable=False to disable filtering. Consume queued frames before evaluating a change; skip_frames is not a hardware commit barrier.

sensor.**skip_frames**(*time*=..., *n*=...)

Drop frames while camera exposure and processing settle.

参数

time –milliseconds to wait. n: number of frames to skip.

sensor.**snapshot**(*buffer*=..., /)

Capture one image from the camera.

参数

buffer –optional writable buffer of at least the configured frame size. The returned Image retains it.

sensor.**status**()

Return camera readiness, size, format, mirror, flip, and crop status.

6.2.9 Classes

class sensor.**SensorStatus**

Dictionary returned by status().

class sensor.**IPAAWBStats**

class sensor.**IPAAFStats**

class sensor.**IPAStats**

6.3 image –图像处理

image 模块提供 Image 对象以及基于 OpenMV imlib 的视觉算法：绘图、格式转换、滤波、颜色/色块分析，以及特征检测（直线、线段、圆、矩形、二维码、AprilTag）。编解码流类型*imageio.ImageIO*见*image.ImageIO –图像流*。

当前 ESP32-P4 板级配置还会启用 ZXing-C++ 条形码后端和*image.Image.find_barcodes()*。

6.3.1 绘图与颜色色块追踪

```
import sensor

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

img = sensor.snapshot()
img.draw_rectangle(10, 10, 50, 50, color=(255, 0, 0))
thresholds = [(30, 100, 15, 127, 15, 127)]
for blob in img.find_blobs(thresholds, pixels_threshold=80, area_threshold=80):
    img.draw_rectangle(blob.rect(), color=(255, 0, 0), thickness=2)
    img.draw_cross(blob.cx(), blob.cy(), color=(0, 255, 0))
```

RGB565 阈值包含六个 LAB 范围值：(L_min, L_max, A_min, A_max, B_min, B_max)。pixels_threshold 会过滤匹配像素过少的区域，area_threshold 会过滤外接矩形面积过小的区域。色块坐标以源图像像素为单位，可直接传给绘图方法。

6.3.2 滤波与二值分割

```
sensor.set_pixformat(sensor.GRAYSCALE)
img = sensor.snapshot()
img.gaussian(1)
img.binary([(80, 255)])
img.erode(1)
img.dilate(1)
img.flush()
```

`gaussian()` 在阈值处理前抑制高频噪声。`binary()` 将匹配的灰度值转换为有效二值，随后先腐蚀再膨胀可去除孤立像素并恢复主体区域。这些方法会原地修改图像。

6.3.3 二维码识别

```
sensor.set_pixformat(sensor.GRAYSCALE)
img = sensor.snapshot()
for code in img.find_qrcodes():
    img.draw_rectangle(code.rect(), color=255, thickness=2)
    print(code.payload())
```

`find_qrcodes()` 返回二维码的几何信息和解码内容。灰度输入通常可以降低处理开销；如果二维码只会出现在画面的固定区域，还可以指定较小的 ROI。

6.3.4 直线、线段与圆检测

```
img = sensor.snapshot()
for line in img.find_lines(threshold=1400, theta_margin=25, rho_margin=25):
    img.draw_line(line.line(), color=255, thickness=2)

for segment in img.find_line_segments(threshold=50, merge_distance=10):
    img.draw_line(segment.line(), color=255, thickness=2)

for circle in img.find_circles(threshold=2500, r_min=8, r_max=80, r_step=4):
    img.draw_circle(circle.x(), circle.y(), circle.r(), color=255, thickness=2)
```

`find_lines()` 和 `find_circles()` 使用霍夫累加器阈值。`find_line_segments()` 使用 Edge Drawing Lines 检测器，其中 `threshold` 是用于产生并追踪边缘的最小 Sobel 梯度幅值，`merge_distance` 用于合并邻近线段。提高阈值可减少较弱的检测结果；限制半径范围、ROI 或图像分辨率可以降低计算量，并通常能够提升稳定性。

6.3.5 编码并保存图像

```
import image

img = sensor.snapshot()
jpg = img.to_jpeg(quality=85, subsampling=image.JPEG_SUBSAMPLING_420)
jpg.save("/sdcard/snapshot.jpg")
print("encoded bytes:", jpg.size())
```

JPEG quality 用于权衡编码体积与图像细节，4:2:0 色度抽样通常可以得到更小的彩色图像。保存图像前，目标文件系统必须已经挂载且可写。

大多数原地操作的方法会返回图像本身，因此可以链式调用：`img.to_grayscale().gaussian(1).binary([(128, 255)])`。

参见:

关于这些方法背后的原理，参见图像模型（像素格式、色彩空间、帧缓冲）与图像处理（滤波、阈值化、特征检测）。

可运行示例：`example/02-Image-Processing`（绘图、滤波、颜色追踪、帧差分）、`example/04-Barcodes`（二维码）与 `example/05-Feature-Detection`（AprilTag、直线、线段、圆）。

6.3.6 Constants

`image.`**`BINARY`**
: 1-bpp black-and-white pixel format (one bit per pixel).

`image.`**`GRAYSCALE`**
: 8-bit grayscale pixel format.

`image.`**`RGB565`**
: 16-bit RGB565 color pixel format.

`image.`**`BAYER`**
: Raw Bayer mosaic pixel format from the sensor.

`image.`**`YUV422`**
: Packed YUV422 pixel format.

`image.`**`JPEG`**
: JPEG-compressed image format.

`image.`**`PNG`**
: PNG-compressed image format.

`image.`**`PALETTE_RAINBOW`**
: False-color rainbow palette (cold to hot).

`image.`**`PALETTE_IRONBOW`**
: “Ironbow” thermal palette.

`image.`**`PALETTE_DEPTH`**
: Depth-map palette.

`image.`**`PALETTE_EVT_DARK`**
: Event-camera dark palette.

`image.`**`PALETTE_EVT_LIGHT`**
: Event-camera light palette.

`image.`**`AREA`**
: Scaling/geometry hint flags for copy()/crop()/scale()/draw_image(): area-average downscaling.

`image.`**`BILINEAR`**
: Bilinear interpolation hint.

`image.`**`BICUBIC`**
: Bicubic interpolation hint.

`image.`**`HMIRROR`**
: Mirror horizontally.

`image.`**`VFLIP`**
: Flip vertically.

image.**TRANSPOSE**
Swap X and Y (transpose).

image.**CENTER**
Center the source in the destination.

image.**EXTRACT_RGB_CHANNEL_FIRST**
Extract the RGB channel before applying the color palette.

image.**APPLY_COLOR_PALETTE_FIRST**
Apply the color palette before other steps.

image.**SCALE_ASPECT_KEEP**
Keep aspect ratio, fitting inside the destination.

image.**SCALE_ASPECT_EXPAND**
Keep aspect ratio, expanding to fill the destination.

image.**SCALE_ASPECT_IGNORE**
Ignore aspect ratio, stretching to the destination.

image.**BLACK_BACKGROUND**
Treat the background as black (needed for correct alpha blending).

image.**ROTATE_90**
Rotate 90 degrees.

image.**ROTATE_180**
Rotate 180 degrees.

image.**ROTATE_270**
Rotate 270 degrees.

image.**JPEG_SUBSAMPLING_AUTO**
Let the JPEG encoder pick chroma subsampling automatically.

image.**JPEG_SUBSAMPLING_444**
4:4:4 JPEG chroma subsampling (best quality).

image.**JPEG_SUBSAMPLING_422**
4:2:2 JPEG chroma subsampling.

image.**JPEG_SUBSAMPLING_420**
4:2:0 JPEG chroma subsampling (smallest size).

image.**SEARCH_EX**
Exhaustive template-search strategy.

image.**SEARCH_DS**
Diamond-search (faster, approximate) template-search strategy.

image.**EDGE_CANNY**
Canny edge-detection method for find_edges().

image.**EDGE_SIMPLE**
Simple gradient-threshold edge-detection method for find_edges().

image.**CORNER_FAST**
FAST corner detector for find_keypoints().

image.**CORNER_AGAST**
AGAST corner detector for find_keypoints().

image.**EAN2**
Barcode symbology constants returned by barcode.type() / accepted by find_barcodes().

image.**EAN5**

image.**EAN8**

image.**UPCE**

image.**ISBN10**

image.**UPCA**

image.**EAN13**

image.**ISBN13**

image.**I25**

image.**DATABAR**

image.**DATABAR_EXP**

image.**CODABAR**

image.**CODE39**

image.**PDF417**

image.**CODE93**

image.**CODE128**

image.**TAG16H5**
AprilTag family constants accepted by find_apriltags().

image.**TAG25H7**

image.**TAG25H9**

image.**TAG36H10**

image.**TAG36H11**

6.3.7 Functions

image.**binary_to_grayscale**(*pixel*)
Convert a 1-bpp binary pixel to a grayscale value.

image.**binary_to_rgb**(*pixel*)
Convert a 1-bpp binary pixel to an (r, g, b) tuple.

image.**binary_to_lab**(*pixel*)
Convert a 1-bpp binary pixel to a LAB tuple.

image.**binary_to_yuv**(*pixel*)
Convert a 1-bpp binary pixel to a YUV tuple.

image.**grayscale_to_binary**(*pixel*)
Convert a grayscale value to a 1-bpp binary pixel.

image.**grayscale_to_rgb**(*pixel*)
Convert a grayscale value to an (r, g, b) tuple.

image.**grayscale_to_lab**(*pixel*)
Convert a grayscale value to a LAB tuple.

image.**grayscale_to_yuv**(*pixel*)
Convert a grayscale value to a YUV tuple.

image.**rgb_to_binary**(*pixel*)
Convert an (r, g, b) color to a 1-bpp binary pixel.

image.**rgb_to_grayscale**(*pixel*)
Convert an (r, g, b) color to a grayscale value.

image.**rgb_to_lab**(*pixel*)
Convert an (r, g, b) color to a LAB tuple.

image.**rgb_to_yuv**(*pixel*)
Convert an (r, g, b) color to a YUV tuple.

image.**lab_to_binary**(*pixel*)
Convert a LAB tuple to a 1-bpp binary pixel.

image.**lab_to_grayscale**(*pixel*)
Convert a LAB tuple to a grayscale value.

image.**lab_to_rgb**(*pixel*)
Convert a LAB tuple to an (r, g, b) tuple.

image.**lab_to_yuv**(*pixel*)
Convert a LAB tuple to a YUV tuple.

image.**yuv_to_binary**(*pixel*)
Convert a YUV tuple to a 1-bpp binary pixel.

image.**yuv_to_grayscale**(*pixel*)
Convert a YUV tuple to a grayscale value.

image.**yuv_to_rgb**(*pixel*)
Convert a YUV tuple to an (r, g, b) tuple.

image.**yuv_to_lab**(*pixel*)
Convert a YUV tuple to a LAB tuple.

6.3.8 Classes

class image.**line**
A detected line segment with Hough-space attributes.

line()
Return the line as (x1, y1, x2, y2).

x1()
Start-point x coordinate.

y1()
Start-point y coordinate.

x2()
End-point x coordinate.

y2()
End-point y coordinate.

length()
Segment length in pixels.

magnitude()
Hough accumulator magnitude (line strength).

theta()
Line angle theta in degrees (Hough space).

rho()
Line distance rho in pixels (Hough space).

class image.**circle**
A detected circle.

circle()
Return the circle as (x, y, r).

x()
Center x coordinate.

y()
Center y coordinate.

r()
Radius in pixels.

magnitude()
Hough accumulator magnitude (circle strength).

class image.**rect**
A detected rectangle (quadrilateral).

corners()
Return the four corner points in clockwise order.

rect()
Return the bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

magnitude()
Detection strength.

class image.**blob**
A connected color region found by find_blobs().

corners()
The four bounding-box corners.

min_corners()
The four corners of the minimum-area enclosing rectangle.

rect()
Bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

pixels()
Number of pixels in the blob.

cx()
Centroid x (integer).

cxf()
Centroid x (float).

cy()
Centroid y (integer).

cyf()
Centroid y (float).

rotation()
Orientation of the blob’ s major axis in radians.

rotation_deg()
Orientation in degrees.

rotation_rad()
Orientation in radians.

code()
Bitmask of the thresholds this blob matched.

count()
Number of blobs merged into this one.

perimeter()
Bounding-box perimeter.

roundness()
Roundness metric in [0, 1] (1 == circle).

elongation()
Elongation metric in [0, 1].

area()
Bounding-box area (w * h).

density()
Pixel density: pixels / area.

extent()
Alias of density().

compactness()
Compactness metric in [0, 1].

solidity()
Solidity (pixels / convex-hull area) in [0, 1].

convexity()
Convexity metric in [0, 1].

x_hist_bins()
Horizontal projection histogram bins.

y_hist_bins()
Vertical projection histogram bins.

major_axis_line()
Major-axis line as (x1, y1, x2, y2).

minor_axis_line()
Minor-axis line as (x1, y1, x2, y2).

enclosing_circle()
Minimum enclosing circle as (x, y, r).

enclosed_ellipse()
Best-fit enclosed ellipse as (x, y, rx, ry, rotation).

class image.**qrcode**
A decoded QR code found by find_qrcodes().

corners()
The four corner points.

rect()
Bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

payload()
Decoded payload string.

version()
QR version (size) number.

ecc_level()
Error-correction level.

mask()
Data mask pattern index.

data_type()
Encoding mode of the payload.

eci()

Extended Channel Interpretation value.

is_numeric()

True if the payload is numeric.

is_alphanumeric()

True if the payload is alphanumeric.

is_binary()

True if the payload is binary.

is_kanji()

True if the payload is Kanji.

class image.**barcode**

A decoded 1D/2D barcode found by find_barcodes().

corners()

The four corner points.

rect()

Bounding box as (x, y, w, h).

x()

Bounding-box top-left x.

y()

Bounding-box top-left y.

w()

Bounding-box width.

h()

Bounding-box height.

payload()

Decoded payload string.

type()

Symbology type constant (e.g. EAN13, CODE128).

rotation()

Rotation of the barcode in degrees.

quality()

Decode quality (number of agreeing scan lines).

class image.**apriltag**

A detected AprilTag with optional 6-DoF pose. Attributes are fields, not methods.

class image.**histogram**

Channel histogram returned by get_histogram(). For RGB565 the L/A/B channels correspond to the LAB color space.

bins()

Combined bins (grayscale/binary) or L bins (RGB565). Values are normalized floating-point pixel frequencies, not integer counts. For a non-empty selection their sum is approximately 1.0 (floating-point rounding applies).

l_bins()

L-channel (lightness) bins.

a_bins()

A-channel bins (RGB565 only).

b_bins()

B-channel bins (RGB565 only).

get_percentile(*p*)

Return the value at the given cumulative percentile (0..1).

参数

p –percentile in the range 0 to 1.

get_threshold()

Compute an Otsu threshold from the histogram.

get_stats()

Compute statistics (mean/median/mode/stdev/…) from the histogram.

get_statistics()

Alias of get_stats().

statistics()

Alias of get_stats().

class image.**percentile**

A percentile result from histogram.get_percentile().

value()

Combined/L value at the percentile.

l_value()

L-channel value.

a_value()

A-channel value.

b_value()

B-channel value.

class image.**threshold**

An Otsu threshold result from histogram.get_threshold().

value()

Combined/L threshold value.

l_value()

L-channel threshold.

a_value()

A-channel threshold.

b_value()

B-channel threshold.

class image.**statistics**

Region statistics returned by get_statistics(). Combined accessors apply to grayscale/binary; `l_*`/`a_*`/`b_*` apply to RGB565 LAB channels.

mean()

median()

mode()

stdev()

min()

max()

lq()
Lower quartile (25th percentile).

uq()
Upper quartile (75th percentile).

l_mean()

l_median()

l_mode()

l_stdev()

l_min()

l_max()

l_lq()

l_uq()

a_mean()

a_median()

a_mode()

a_stdev()

a_min()

a_max()

a_lq()

a_uq()

b_mean()

b_median()

b_mode()

b_stdev()

b_min()

b_max()

b_lq()

b_uq()

class image.**Image**(*path*)

class image.**Image**(*width*, *height*, *pixformat*, *, *buffer*=...)

class image.**Image**(*array*, *, *buffer=...*)

A 2D image backed by a pixel buffer. Create one from a file path, from a width/height/pixformat, or from an array; most often you get one from sensor.snapshot(). Most in-place methods return `self` so calls can be chained.

Load an image from a file (BMP/PPM/PGM/JPEG/PNG depending on build).

参数

path –source file path.

width()

Image width in pixels.

height()

Image height in pixels.

format()

Pixel format constant (BINARY/GRAYSCALE/RGB565/…).

size()

Size of the pixel buffer in bytes.

bytearray()

Return the pixel buffer as a bytearray (no copy).

get_pixel(*x*, *y*, *, *rgbtuple=...*)

Read one pixel. x, y: pixel coordinates.

参数

rgbtuple –for RGB565, True returns an (r, g, b) tuple, False the packed value.

set_pixel(*x*, *y*, *pixel*)

Write one pixel. x, y: pixel coordinates.

参数

pixel –grayscale value or (r, g, b) color.

flush()

Push this image to the host as a binary EV-MUX `preview.frame` JPEG.

save(*path*, *, *roi=...*, *quality=...*)

Save the image to a file; the format is inferred from the extension. PNG encoding is not enabled in current firmware.

参数

- **path** –destination path.
- **roi** –optional source rectangle (x, y, w, h).
- **quality** –JPEG quality from 1 to 100 when saving as JPEG.

copy(*, *roi=...*, *x_scale=...*, *y_scale=...*, *rgb_channel=...*, *alpha=...*, *color_palette=...*, *alpha_palette=...*, *hint=...*, *copy=...*)

Return a scaled/cropped/converted copy of the image. x_scale, y_scale: horizontal/vertical scale factors.

参数

- **roi** –source rectangle (x, y, w, h) to read from.
- **hint** –bitwise OR of AREA/BILINEAR/BICUBIC/HMIRROR/VFLIP/TRANSPOSE/ROTATE_*/SCALE_ flags.
- **rgb_channel** –extract a single RGB channel (0, 1, or 2; -1 for all).
- **alpha** –blend alpha from 0 to 255.
- **copy** –True returns a new image; False converts in place.

crop(*, *roi=...*, *x_scale=...*, *y_scale=...*, *hint=...*, *copy=...*)

Crop/scale the image to a region of interest. Same options as copy().

scale(*, *x_scale=...*, *y_scale=...*, *roi=...*, *hint=...*, *copy=...*)

Scale the image by x_scale/y_scale (alias of crop in this build).

compress(*, *quality=...*, *subsampling=...*, *roi=...*)

Compress the image to JPEG in place (alias of to_jpeg).

参数

- **quality** –JPEG quality from 1 to 100.
- **subsampling** –one of the JPEG_SUBSAMPLING_* constants.

to_bitmap(*, *copy=...*, *roi=...*)

Convert to a 1-bpp bitmap.

to_grayscale(*, *copy=...*, *roi=...*)

Convert to grayscale.

to_rgb565(*, *copy=...*, *roi=...*)

Convert to RGB565.

to_rainbow(*, *copy=...*, *roi=...*)

Apply the rainbow palette and convert to RGB565.

to_ironbow(*, *copy=...*, *roi=...*)

Apply the ironbow thermal palette and convert to RGB565.

to_jpeg(*, *quality=...*, *subsampling=...*, *copy=...*)

Encode to JPEG.

参数

- **quality** –JPEG quality from 1 to 100.
- **subsampling** –one of the JPEG_SUBSAMPLING_* constants.

to_png(*, *copy=...*)

Encode to PNG (not enabled in current firmware).

clear(*, *mask=...*)

Fill the image with zeros (optionally only where mask is set).

参数

mask –optional 1-bpp image limiting which pixels are cleared.

draw_line(*x0*, *y0*, *x1*, *y1*, *color=...*, *thickness=...*)

draw_line(*line*, *color=...*, *thickness=...*)

draw_rectangle(*x*, *y*, *w*, *h*, *color=...*, *thickness=...*, *fill=...*)

draw_rectangle(*rect*, *color=...*, *thickness=...*, *fill=...*)

draw_circle(*x*, *y*, *r*, *color=...*, *thickness=...*, *fill=...*)

draw_circle(*circle*, *color=...*, *thickness=...*, *fill=...*)

draw_ellipse(*x*, *y*, *rx*, *ry*, *rotation*, *color=...*, *thickness=...*, *fill=...*)

draw_ellipse(*ellipse*, *color=...*, *thickness=...*, *fill=...*)

draw_string(*x*, *y*, *text*, *color=...*, *scale=...*, *x_spacing=...*, *y_spacing=...*, *mono_space=...*, *char_rotation=...*, *char_hmirror=...*, *char_vflip=...*, *string_rotation=...*, *string_hmirror=...*, *string_vflip=...*)

Draw text using the built-in bitmap font. x, y: top-left position of the text.

参数

- **text** –string to draw.
- **color** –text color.
- **scale** –integer/float scale factor of the font.

draw_cross(*x*, *y*, *color=...*, *size=...*, *thickness=...*)

Draw a cross marker centered at (x, y).

draw_arrow(*x0*, *y0*, *x1*, *y1*, *color=...*, *size=...*, *thickness=...*)

draw_arrow(*line*, *color=...*, *size=...*, *thickness=...*)

draw_edges(*corners*, *color=...*, *size=...*, *thickness=...*, *fill=...*)

Draw connecting edges between a sequence of corner points.

draw_keypoints(*keypoints*, *color=...*, *size=...*, *thickness=...*, *fill=...*)

Draw a set of keypoints.

draw_image(*image*, *x=...*, *y=...*, ***, *x_scale=...*, *y_scale=...*, *roi=...*, *rgb_channel=...*, *alpha=...*, *color_palette=...*, *alpha_palette=...*, *hint=...*, *mask=...*)

Alpha-blend/scale another image (or a solid color) onto this one. x, y: destination top-left position. x_scale, y_scale: scale factors; roi: source rectangle; alpha: 0..255;

参数

- **image** –source Image, an (r, g, b) tuple, or a packed scalar color.
- **hint** –SCALE_ASPECT_*/interpolation flags; mask: optional 1-bpp mask.

binary(*thresholds*, ***, *invert=...*, *zero=...*, *mask=...*, *to_bitmap=...*, *copy=...*)

Threshold the image into a binary mask against a list of color thresholds.

参数

- **thresholds** –list of grayscale (min, max) or LAB 6-tuple thresholds.
- **invert** –invert the match.
- **zero** –zero out matching pixels instead of setting them.
- **mask** –optional 1-bpp image limiting the operation.
- **to_bitmap** –output a 1-bpp bitmap.
- **copy** –return a new image instead of modifying in place.

invert()

Invert pixel values.

erode(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological erosion with a (2*ksize+1) square structuring element.

参数

- **ksize** –structuring-element radius.
- **threshold** –minimum number of set neighbors to keep a pixel.
- **mask** –optional 1-bpp image limiting the operation.

dilate(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological dilation. See erode() for parameters.

open(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological opening (erode then dilate).

close(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological closing (dilate then erode).

difference(*image*, *x=...*, *y=...*, ***, *roi=...*, *mask=...*)

Absolute per-pixel difference against another image/color (frame differencing). x, y: placement of the other image.

参数

image –other Image, (r, g, b) tuple, or scalar color.

histeq(***, *adaptive=...*, *clip_limit=...*, *mask=...*)

Histogram equalization (global). adaptive is not supported in this build.

参数

clip_limit –reserved; mask: optional 1-bpp image.

mean (*ksize*, *, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Box (mean) blur over a (2*ksize+1) window. threshold/offset/invert: adaptive-threshold the result.

参数

- **ksize** –kernel radius.
- **mask** –optional 1-bpp image.

median (*ksize*, *, *percentile=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Median filter (edge-preserving denoise).

参数

- **ksize** –kernel radius.
- **percentile** –rank in [0, 1] to select (0.5 == median).

mode (*ksize*, *, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Mode (most-common value) filter.

参数

ksize –kernel radius.

midpoint (*ksize*, *, *bias=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Midpoint filter ((min + max)/2 blended by bias).

参数

ksize –kernel radius; bias: 0 == min, 1 == max, 0.5 == midpoint.

morph (*ksize*, *kernel*, *, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Apply an arbitrary NxN convolution kernel.

参数

- **ksize** –kernel radius; the kernel must be (2*ksize+1) square.
- **kernel** –flat or nested sequence of integer weights.
- **mul** –output multiplier (defaults to 1/sum(kernel)); add: output bias.

blur (*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur using a separable Pascal-triangle kernel (alias: gaussian, gaussian_blur).

参数

- **ksize** –kernel radius.
- **unsharp** –True turns the filter into an unsharp-mask sharpener.

gaussian (*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur. See blur() for parameters.

gaussian_blur (*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur. See blur() for parameters.

laplacian (*ksize*, *, *sharpen=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Laplacian edge filter (or sharpener when sharpen is True).

参数

ksize –kernel radius.

bilateral (*ksize*, *, *color_sigma=...*, *space_sigma=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Bilateral filter (edge-preserving smoothing).

参数

- **ksize** –kernel radius.
- **color_sigma** –range sigma (color closeness); larger blurs across more contrast.
- **space_sigma** –spatial sigma (distance closeness).

get_histogram (*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Compute a channel histogram over the image or a region. bins / l_bins / a_bins / b_bins: bin counts; -1 selects the channel default.

参数

- **thresholds** –optional list of thresholds to restrict counted pixels.
- **invert** –invert the threshold match.
- **roi** –region of interest (x, y, w, h).
- **difference** –optional image to histogram the difference against.

get_hist(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_histogram().

histogram(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_histogram().

get_statistics(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Compute region statistics (mean/median/mode/stdev/quartiles). thresholds/invert/roi/bins: as in get_histogram().

get_stats(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_statistics().

statistics(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_statistics().

get_regression(*thresholds*, ***, *invert=...*, *roi=...*, *x_stride=...*, *y_stride=...*, *area_threshold=...*, *pixels_threshold=...*, *robust=...*)

Fit a line to the thresholded pixels via least-squares (or robust) regression. x_stride, y_stride: pixel sampling steps. area_threshold, pixels_threshold: minimum region area / pixel count to fit.

参数

- **thresholds** –list of color thresholds selecting the pixels to fit.
- **invert** –invert the match; roi: region of interest.
- **robust** –use a robust (Theil-Sen) estimator.

find_blobs(*thresholds*, ***, *invert=...*, *roi=...*, *x_stride=...*, *y_stride=...*, *area_threshold=...*, *pixels_threshold=...*, *merge=...*, *margin=...*, *threshold_cb=...*, *merge_cb=...*, *x_hist_bins_max=...*, *y_hist_bins_max=...*)

Find connected color regions (blobs) matching the thresholds. x_stride, y_stride: pixel sampling steps. area_threshold, pixels_threshold: minimum bounding-box area / pixel count. threshold_cb / merge_cb: optional Python filter/merge callbacks.

参数

- **thresholds** –list of color thresholds.
- **invert** –invert the match; roi: region of interest.
- **merge** –merge overlapping blobs; margin: extra merge margin.

find_lines(***, *roi=...*, *x_stride=...*, *y_stride=...*, *threshold=...*, *theta_margin=...*, *rho_margin=...*)

Find straight lines with the Hough transform. theta_margin, rho_margin: merge tolerance for similar lines.

参数

- **roi** –region of interest; x_stride, y_stride: sampling steps.
- **threshold** –minimum Hough accumulator magnitude to report a line.

find_line_segments(***, *roi=...*, *merge_distance=...*, *max_theta_diff=...*, *threshold=...*)

Find line segments with the Edge Drawing Lines detector. merge_distance and max_theta_diff merge nearby, similarly angled segments.

参数

threshold –minimum Sobel gradient magnitude used to seed and trace edges.

find_circles(***, *roi=...*, *x_stride=...*, *y_stride=...*, *threshold=...*, *x_margin=...*, *y_margin=...*, *r_margin=...*, *r_min=...*, *r_max=...*, *r_step=...*)

Find circles with the Hough transform. x_margin, y_margin, r_margin: merge tolerances. r_min, r_max, r_step: radius search range and step.

参数

threshold –minimum accumulator magnitude.

find_rects(*, *roi*=..., *threshold*=...)

Find rectangles/quadrilaterals (e.g. for fiducials).

参数

threshold –minimum edge-magnitude score.

find_qrcodes(*, *roi*=...)

Find and decode QR codes.

参数

roi –region of interest.

find_barcodes(*, *roi*=...)

Find and decode 1D/2D barcodes (requires the barcode backend).

参数

roi –region of interest.

find_apriltags(*, *roi*=..., *families*=..., *fx*=..., *fy*=..., *cx*=..., *cy*=..., *pose*=...)

Find and decode AprilTags, optionally estimating 6-DoF pose. fx, fy, cx, cy: camera intrinsics for pose estimation.

参数

- **roi** –region of interest; families: OR of TAG* family constants.
- **pose** –True computes translation/rotation (needs intrinsics).

6.4 display –LCD 显示

`display` 模块驱动开发板上的 LCD。创建一个 `ESP32Display`（别名为 `display.Display`），并用 `ESP32Display.write()` 把图像帧送到屏幕。

6.4.1 摄像头预览

```
import sensor, display

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)

lcd = display.ESP32Display()
while True:
    lcd.write(sensor.snapshot(), fit=True)
```

使用 `fit=True` 时，源图像会缩放以适配屏幕，并沿用显示驱动的默认放置方式。显示对象应只创建一次并持续复用；重复创建对象会重新初始化屏幕资源。

6.4.2 复用 Image 绘制内容

如果应用需要自行绘制图形，可以创建一个普通的 RGB565 图像缓冲区并重复更新，再传给 `lcd.write()`：

```
import display, image

lcd = display.Display()
canvas = image.Image(lcd.width(), lcd.height(), image.RGB565)
canvas.draw_rectangle(0, 0, canvas.width(), canvas.height(), color=(6, 8, 18),␣
↪fill=True)
canvas.draw_string(20, 20, "ESP-VISION", color=(255, 255, 255), scale=2)
lcd.write(canvas, fit=False)
```

后续帧可以继续修改并复用同一个 `Image`。显示驱动会将图像像素写入屏幕；图像无需分配在显示驱动的帧缓冲区中。

6.4.3 定位、缩放与裁剪

使用 `x` 和 `y` 设置显示位置，使用 `x_scale` 和 `y_scale` 显式缩放，并通过 `roi` 仅显示源图像中的指定区域：

```
img = sensor.snapshot()
lcd.write(
    img,
    x=20,
    y=10,
    x_scale=0.5,
    y_scale=0.5,
    roi=(40, 30, 160, 120),
    fit=False,
)
```

当 `fit=False` 时，输出由显式指定的位置和缩放比例控制。ROI 使用源图像坐标，格式为 `(x, y, width, height)`；缩小 ROI 还可以减少送入显示链路的图像数据量。

6.4.4 背光与资源释放

```
lcd = display.Display(backlight=80)
print("panel:", lcd.width(), "x", lcd.height())
lcd.backlight(30)
lcd.clear()
lcd.deinit()
```

背光值为 0 到 100 的百分比。应用永久释放显示设备时应调用 `deinit()`；驱动反初始化后不得继续写入图像帧。

参见：

可运行示例：`example/06-Peripherals/01-Display/lcd_preview.py`。

6.4.5 Classes

class display.**ESP32Display**(*width=...*, *height=...*, *refresh=...*, *, *backlight=...*)

ESP32 display object for the board LCD.

Create and initialize the display.

参数

- **width** –requested display width; 0 uses board default.
- **height** –requested display height; 0 uses board default.
- **refresh** –target refresh rate in Hz.
- **backlight** –initial backlight percentage, from 0 to 100.

deinit()

Release the display driver resources.

width()

Return the physical display width in pixels.

height()

Return the physical display height in pixels.

clear(*display_off=...*)

Clear the display.

参数

display_off –True may turn the panel output off when supported by the board.

backlight(*value=...*)

Get or set backlight brightness.

参数

value –None returns current brightness; otherwise set 0 to 100 percent.

write(*image*, *, *x=...*, *y=...*, *x_scale=...*, *y_scale=...*, *roi=...*, *fit=...*)

Draw an image on the display. x, y: destination top-left position in display pixels. x_scale, y_scale: optional manual scale factors.

参数

- **image** –source image, normally from sensor.snapshot().
- **roi** –optional source rectangle (x, y, w, h).
- **fit** –True scales the image to fit the display area.

6.5 lcd_touch –触摸输入

`lcd_touch` 模块读取开发板的触摸屏，无需先初始化显示设备。`Touch()` 要求开发板已配置触摸硬件，不支持的开发板会抛出 `OSError`。

6.5.1 读取触摸坐标

```
import lcd_touch
import time

touch = lcd_touch.Touch()
try:
    while True:
        points = touch.read()
        if points:
            print(points)
        time.sleep_ms(20)
finally:
    touch.deinit()
```

`read()` 返回由 `(x, y)` 像素坐标组成的元组，没有触点时返回 `()`。坐标遵循开发板的触摸配置；应用需要自行处理图像缩放、裁剪和显示位置对应的坐标转换。

6.5.2 所有权与资源释放

触摸设备由实例共享。创建新的 `Touch()` 会将所有权转移给新实例；旧实例无法继续读取，其 `deinit()` 调用也不会释放新实例持有的设备。使用完毕后应调用 `deinit()`，重复调用是安全的，MicroPython 软复位也会释放设备。

参见:

可运行示例：`example/06-Peripherals/03-Touch/touch_read.py` 和 `example/06-Peripherals/03-Touch/touch_lcd.py`。

6.5.3 Classes

class lcd_touch.**Touch**

Touchscreen object for the board LCD. Requires a board with configured touch hardware; otherwise raises OSError.

Initialize touch input without requiring display initialization. A new instance takes ownership of the shared touch device.

read()

Return (x, y) pixel coordinates for each touch, or an empty tuple. Raise OSError if this instance is inactive or reading fails.

deinit()

Release the touch device if this instance still owns it.

6.6 espdl –模型推理

`espdl` 模块在采集到的图像上运行 ESP-DL 的 `.espdl` 模型，并提供常见任务封装：目标检测（`PedestrianDetect`、`ESPDet`、`YOLO11`）、姿态估计（`YOLO11nPose`）、图像分类（`ImageNetCls`）和单目标追踪（`NanoTrack`），同时可通过 `Model` 将 ESP-DL 原始输出 tensor 暴露给 Python。

6.6.1 按键确认单目标追踪

`espdl.NanoTrack` 运行 NanoTrack V2 全 INT8 模型。将 `models/nanotrack_v2/` 中的三个 `.espdl` 文件复制到设备的 `/sdcard/nanotrack_v2/` 即可，本地量化实验保存在被忽略的 `build/experiments/nanotrack-v2`，不包含在全新检出的仓库中。P4/S31 使用 ESP32-P4 导出物；本封装不接受混合 INT16 实验模型。S3 需要面向 S3 重新量化并单独验证。

`example/03-Machine-Learning/00-ESP-DL/nanotrack.py` 默认显示中心 64x64 选框。将目标放入框内，按 BOOT 键（ESP32-P4X-EYE 和 P4 Function EV Board 均为 GPIO35）开始追踪；再次按键返回选框状态。丢失目标后保留原模板并继续局部搜索。相机图像按原尺寸显示，小屏幕只裁切中心区域，不缩放。可通过 `BUTTON_PIN` 和 `BOX_SIZE` 适配其他硬件或目标尺寸。推理在绘制标注前完成，避免图形进入模型输入。

示例使用可配置的 `MATCH_THRESHOLD = 0.95`，画面上的 `match` 是模型匹配分数，不是目标身份正确的概率。

```
from machine import Pin
import display
import espdl
import sensor
import time
```

(续下页)

(接上页)

```
MODEL_DIR = "/sdcard/nanotrack_v2"
# Active-low BOOT button on ESP32-P4X-EYE and P4 Function EV Board.
# Change this pin for other boards.
BUTTON_PIN = 35
BOX_SIZE = 64
DEBOUNCE_MS = 30
# A match score is not a calibrated target-identity probability.
MATCH_THRESHOLD = 0.95


lcd = None
tracker = None
try:
    button = Pin(BUTTON_PIN, Pin.IN, Pin.PULL_UP)
    sensor.reset()
    sensor.set_pixformat(sensor.RGB565)
    sensor.set_framesize(sensor.QVGA)
    sensor.skip_frames(time=1000)
    lcd = display.Display()
    tracker = espdl.NanoTrack(
        MODEL_DIR + "/template_deploy.espdl",
        MODEL_DIR + "/search_deploy.espdl",
        MODEL_DIR + "/head_deploy.espdl",
    )
    frame = sensor.snapshot()
    width, height = frame.width(), frame.height()
    canvas = frame.copy()
    # Center at native size; smaller displays show the center crop.
    visible_w = min(width, lcd.width())
    visible_h = min(height, lcd.height())
    view = ((width - visible_w) // 2, (height - visible_h) // 2,
            visible_w, visible_h)
    offset_x = (lcd.width() - visible_w) // 2
    offset_y = (lcd.height() - visible_h) // 2
    size = min(BOX_SIZE, visible_w, visible_h)
    roi = ((width - size) // 2, (height - size) // 2, size, size)
    tracking = False
    last_button = stable_button = button.value()
    changed_at = time.ticks_ms()
    # A button held at startup must be released before it can confirm.
    armed = stable_button == 1

    while True:
        now = time.ticks_ms()
        level = button.value()
        pressed = False
        if level != last_button:
            last_button = level
            changed_at = now
        if level != stable_button and time.ticks_diff(now, changed_at) >= DEBOUNCE_
↪MS:
            stable_button = level
            if level:
                armed = True
            elif armed:
                armed = False
                pressed = True

        if pressed:
            if tracking:
                tracking = False
```

(续下页)

(接上页)

```
            else:
                canvas.draw_image(frame, 0, 0)
                canvas.draw_string(view[0] + 2, view[1] + 2, "Initializing...",
                                   color=(255, 255, 0))
                lcd.write(canvas, x=offset_x, y=offset_y, roi=view, fit=False)
                # Use the last displayed raw frame, without any annotations.
                tracker.init(frame, roi)
                tracking = True

        frame = sensor.snapshot()
        begin = time.ticks_ms()
        result = tracker.track(frame, threshold=MATCH_THRESHOLD) if tracking else␣
↪None
        canvas.draw_image(frame, 0, 0)
        if result is not None:
            x, y, w, h, score = result
            message = "Tracking - BOOT: select"
            canvas.draw_rectangle(x, y, w, h, color=(0, 255, 0), thickness=2)
            canvas.draw_string(view[0] + 2, view[1] + visible_h - 14,
                               "match %.2f  %d ms" % (score, time.ticks_diff(time.
↪ticks_ms(), begin)),
                               color=(0, 255, 0))
        elif tracking:
            # Preserve the template and retry near the last accepted position.
            message = "Lost - searching..."
        else:
            message = "Aim box, press BOOT"
            canvas.draw_rectangle(roi, color=(255, 255, 0), thickness=2)
        canvas.draw_string(view[0] + 2, view[1] + 2, message, color=(255, 255,␣
↪255))
        lcd.write(canvas, x=offset_x, y=offset_y, roi=view, fit=False)
        time.sleep_ms(0 if tracking else 10)
finally:
    try:
        if tracker is not None:
            tracker.deinit()
    finally:
        if lcd is not None:
            lcd.deinit()
```

`init(image, roi)` 缓存目标特征，然后释放模板模型；搜索模型和 head 常驻，重新框选时临时重载模板模型。同一次追踪使用尺寸不变、未绘制标注的 RGB565 或灰度图像。结果为 `(x, y, w, h, score)`；低于阈值时返回 `None`，不更新上一搜索状态。分数不能保证目标仍然可见。这是局部单目标追踪，不包含全局自动找回或网络权重训练。

6.6.2 原始输出 tensor

```
import sensor, espdl

model = espdl.Model("/sdcard/custom.espdl", mean=(0, 0, 0), std=(255, 255, 255),␣
↪letterbox=True)
try:
    print("inputs:", model.inputs())
    print("outputs:", model.outputs())
    img = sensor.snapshot()
    outputs = model.predict(img)
    for name, tensor in outputs.items():
        _, shape, dtype, exponent, raw = tensor
        print(name, shape, dtype, exponent, len(raw))
```

(续下页)

(接上页)

```
finally:
    model.deinit()
```

当内置任务封装不匹配模型输出布局时，可使用 `Model`。ESP-DL 仍负责图像预处理和推理；`predict()` 返回原始输出 tensor，Python 代码再执行模型相关解码。`inputs()` 和 `outputs()` 返回 `TensorInfo` 元组 `(name, shape, dtype, exponent)`。`predict()` 返回按输出 tensor 名称索引的 `RawTensor` 元组 `(name, shape, dtype, exponent, bytes)`。应根据 `dtype` 解包原始字节，应用 ESP-DL 的二次幂 `exponent` 量化比例，再执行 sigmoid、框解码、NMS、softmax 或 top-k 选择等逻辑。若启用 `letterbox=True`，将坐标映射回源图像前需要先去除模型输入中的 padding。

6.6.3 目标检测

```
import sensor, espdl

det = espdl.ESPDet("/sdcard/espdet_pico_224_224_face.espdl", score=0.5, nms=0.7)
try:
    img = sensor.snapshot()
    for x, y, w, h, score, category in det.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0), thickness=2)
        img.draw_string(x, max(0, y - 12), "%.2f:%d" % (score, category))
finally:
    det.deinit()
```

`score` 用于过滤低置信度候选框，`nms` 用于控制重叠检测框的抑制。检测坐标会映射回输入图像，可直接用于绘图。模型应加载一次并在多帧之间复用；`deinit()` 会释放模型权重和中间缓冲区。

6.6.4 PicoDet

PicoDet 是仓库自带行人模型采用的检测器家族。它的 anchor-point 输出 tensor 由 ESP-DL 的 Pico 后处理器解码，输出布局与 ESPDet、YOLO11 不同。

6.6.5 PedestrianDetect

`PedestrianDetect` 是仓库自带 224x224 PicoDet 行人模型的 Python API。它使用该模型的预处理默认值，并返回标准检测元组：

```
det = espdl.PedestrianDetect(
    "/sdcard/pedestrian_detect/pedestrian_detect_pico.espdl",
    score=0.7,
    nms=0.5,
)
```

6.6.6 图像分类

```
import espdl, image

img = image.Image("/sdcard/cat.jpg").to_rgb565(copy=True)
classifier = espdl.ImageNetCls(
    "/sdcard/imagenet_cls_mobilenetv2_s8_v1.espdl",
    topk=5,
    score=0.0,
)
try:
    for label, score in classifier.classify(img):
```

(续下页)

(接上页)

```
        print(label, "%.4f" % score)
finally:
    classifier.deinit()
```

分类结果按照模型输出顺序返回最多 `topk` 个 `(label, score)` 元组。必要时应将文件图像转换为 RGB565，确保输入格式可被封装层接受。

6.6.7 姿态估计

```
pose = espdl.YOLO11nPose("/sdcard/espdet_yolo11n_pose_160_160_coco.espdl", score=0.
↪35, topk=5)
try:
    img = sensor.snapshot()
    for x, y, w, h, score, category, keypoints in pose.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
        for px, py in keypoints:
            if px > 0 and py > 0:
                img.draw_circle(px, py, 2, color=(0, 255, 0), fill=True)
finally:
    pose.deinit()
```

每个姿态结果包含 17 个 COCO 关键点。缺失或低置信度关键点会返回为 `(0, 0)`，绘图或计算关节几何关系前应将其跳过。

6.6.8 运行时调整阈值

```
det.set_thresholds(score=0.65, nms=0.6)
```

无需重新加载模型即可修改阈值，适合应用根据光照、距离或场景目标密度的变化进行动态调整。

6.6.9 结果元组

- TensorInfo：`(name, shape, dtype, exponent)`
- RawTensor：`(name, shape, dtype, exponent, bytes)`
- 检测：`(x, y, w, h, score, category)`
- 姿态：`(x, y, w, h, score, category, keypoints)`，含 17 个 COCO 关键点
- 分类：`(label, score)`
- 追踪：`(x, y, w, h, score)`，低于匹配阈值时为 `None`

参见：

AI 推理 介绍 ESP-DL 推理流程、`.espdl` 格式、量化以及前后处理。部署新模型请参见*引入新的模型*。

可运行示例：`example/03-Machine-Learning/00-ESP-DL`（行人检测、ESP-DL 原始输出解码、ESPDet、YOLO11、姿态、ImageNet 分类、BOOT 按键确认的 NanoTrack 追踪）。

6.6.10 Functions

`espdl.`**`load_model`**(*path*, *, *profile*=...)

Preload an ESP-DL model file.

参数

- **`path`** –model path, for example "/sdcard/model.espdl" or "/flash/model.espdl".
- **`profile`** –True enables ESP-DL profiling output when supported.

6.6.11 Classes

class espdl.**NanoTrack**(*template*, *search*, *head*)

NanoTrack V2 single-object tracker using full-INT8 NHWC ESP-DL models.

Load search and head models. The template model is loaded only during init().

参数

- **template** –127x127 template backbone .espdl path.
- **search** –255x255 search backbone .espdl path.
- **head** –two-input NanoTrack V2 head .espdl path.

init(*image*, *roi*)

Cache target features from an unannotated RGB565/grayscale image, then release the template model.

参数

roi –(x, y, w, h), inside the image, at least 10x10 pixels.

track(*image*, *, *threshold=...*)

Track in a same-size image; return (x, y, w, h, score), or None below threshold.

参数

threshold –raw foreground score in [0, 1]; rejected frames do not update the search state.

deinit()

Release models and cached features. Repeated calls are safe.

class espdl.**Model**(*path*, *, *mean=...*, *std=...*, *rgb_swap=...*, *letterbox=...*, *pad=...*)

Generic ESP-DL model runner for examples that implement model-specific decoding in Python.

Create a raw model runner from an .espdl model.

参数

- **path** –model path.
- **mean** –optional RGB mean values for image preprocessing.
- **std** –optional RGB standard deviation values for image preprocessing.
- **rgb_swap** –True converts RGB input to BGR before model inference.
- **letterbox** –True keeps aspect ratio and pads the model input.
- **pad** –optional RGB padding values used when letterbox is enabled.

deinit()

Release model resources.

inputs()

Return model input metadata keyed by tensor name.

outputs()

Return model output metadata keyed by tensor name.

predict(*image*)

Run model inference on an image and return raw output tensors keyed by tensor name.

参数

image –RGB565 or grayscale image.

class espdl.**ESPDet**(*path*, *, *score=...*, *nms=...*, *mean=...*, *std=...*)

ESP-DL object detection wrapper for ESPDet models.

Create a detector from an .espdl model.

参数

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.

- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run object detection on an image.

参数

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

参数

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**PedestrianDetect**(*path*, *, *score*=..., *nms*=..., *topk*=...)

ESP-DL PicoDet pedestrian detection wrapper.

Create a pedestrian detector from the bundled ESP-DL model.

参数

- **path** –model path.
- **score** –optional confidence threshold; defaults to the upstream model's 0.7.
- **nms** –optional non-maximum suppression threshold; defaults to the upstream model's 0.5.
- **topk** –maximum number of detections returned per frame.

deinit()

Release model resources.

detect(*image*)

Run pedestrian detection on an RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

参数

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**YOLO11**(*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11 object detection wrapper.

Create a YOLO11 detector from an .espdl model.

参数

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.
- **topk** –maximum number of detections returned per frame.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run object detection on an image.

参数

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

参数

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**YOLO11nPose**(*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11n COCO pose wrapper.

Create a YOLO11n pose detector from an .espdl model.

参数

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.
- **topk** –maximum number of pose results returned per frame.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run COCO pose detection on an image. Returns 17 COCO keypoints for each person.

参数

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

参数

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**ImageNetCls**(*path*, *, *topk*=..., *score*=..., *mean*=..., *std*=..., *softmax*=...)

ESP-DL ImageNet classification wrapper.

Create a classifier from an .espdl model.

参数

- **path** –model path.
- **topk** –maximum number of classes returned.
- **score** –optional minimum score threshold.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.
- **softmax** –True applies softmax to output scores.

deinit()

Release model resources.

classify(*image*)

Run image classification on an image.

参数

image –RGB565 or grayscale image.

set_thresholds(*, *topk*=..., *score*=...)

Update classifier thresholds.

参数

- **topk** –new maximum result count, or None to keep current value.
- **score** –new minimum score, or None to keep current value.

6.7 tflite –模型推理

`tflite` 模块加载 TensorFlow Lite `.tflite` flatbuffer，并通过 TensorFlow Lite Micro 运行。它为 TFLite 模型提供通用执行接口；当某类模型需要共享预处理或后处理时，也可以在其上继续封装更高层的任务接口。

6.7.1 基本用法

```
import tflite

model = tflite.Model("/sdcard/sine.tflite")
try:
    print("input:", model.input_shape, model.input_dtype, model.input_scale, model.
↪input_zero_point)
    print("output:", model.output_shape, model.output_dtype, model.output_scale,␣
↪model.output_zero_point)
    outputs = model.predict([input_array])
    raw = outputs[0]
finally:
    model.deinit()
```

`predict()` 会以 ulab ndarray 返回原始输出 tensor。量化输出不会自动解码；模型需要时，应使用 `output_scale` 和 `output_zero_point` 自行换算。

6.7.2 Callable 输入

对于图像模型，直接填充输入 tensor 往往比先构造中间 ndarray 更省：

```
def fill_input(buffer, shape, dtype_code):
    # 按模型要求填充量化后的原始字节。
    ...

outputs = model.predict([fill_input])
```

callable 会收到输入 tensor 的可写 bytearray 视图、tensor shape，以及 `ord("b")` 这类整数 dtype code。这个路径适合在 Python 示例中处理图像缩放、灰度转换和量化。

6.7.3 后处理

`Model` 可以接收默认 `postprocess` 回调，`predict()` 也可以接收单次 `callback`。回调参数为 `(model, inputs, raw_outputs)`，并且可以返回任意 Python 对象。

参见:

可运行示例：`example/03-Machine-Learning/01-TFLite`（person/no-person 与 sine smoke test）。

6.7.4 Classes

class tflite.**Model**(*path*, *, *postprocess*=...)

TensorFlow Lite Micro model wrapper.

Load a .tflite flatbuffer from the filesystem.

参数

- **path** –model path, for example “/sdcard/model.tflite” or “/flash/model.tflite”.

- **postprocess** –optional callback that receives (model, inputs, raw_outputs) and returns the final result.

deinit()

Release model, interpreter, and tensor arena resources.

predict(*inputs*, *, *callback*=...)

Run inference.

参数

- **inputs** –one item per input tensor. Each item is an ulab ndarray or a callable that fills the raw tensor buffer.
- **callback** –optional one-shot post-processing callback; overrides the default postprocess callback.

6.8 image.ImageIO –图像流

`image.ImageIO` 类型用于录制和回放图像序列，并保留帧间时间间隔。文件流在存储上读写容器文件；内存流则把帧保存在预分配的缓冲区中。它以 `image.ImageIO` 的形式暴露在 `image` 模块上；类型存根位于 `stubs/imageio.pyi`。

6.8.1 录制到存储

```
import sensor, image

stream = image.ImageIO("/sdcard/stream.bin", "w")
for _ in range(30):
    stream.write(sensor.snapshot())
stream.sync()
print("frames:", stream.count(), "bytes:", stream.size())
stream.close()
```

每次 `write()` 都会保存图像以及与上一帧之间的时间间隔。`sync()` 可在不关闭流的情况下将文件缓冲数据写入存储。文件流使用完毕后必须关闭，以完成容器元数据和存储缓冲区的写入。

6.8.2 回放已录制的图像流

```
import image

stream = image.ImageIO("/sdcard/stream.bin", "r")
try:
    while True:
        img = stream.read(loop=False, pause=True)
        if img is None:
            break
        img.flush()
finally:
    stream.close()
```

`pause=True` 会按照录制时的帧间隔进行回放，`loop=False` 则会在流结束时返回 `None`，而不是回到第一帧。需要以处理能力允许的最快速度读取时，可设置 `pause=False`。

6.8.3 使用内存图像流

```
stream = image.ImageIO((320, 240, image.RGB565), 10)
for _ in range(10):
    stream.write(sensor.snapshot())

stream.seek(0)
first = stream.read(pause=False)
print("buffer bytes:", stream.buffer_size())
stream.close()
```

内存流不产生存储 I/O，适合保存短时间图像历史或实现帧差处理，但创建流时会在 RAM 或 PSRAM 中一次性分配全部容量。

参见:

编解码与推流 一并介绍了 ImageIO、JPEG、H.264 以及 USB CDC 预览通路。

可运行示例：`example/02-Image-Processing/03-Frame-Differencing/in_memory_frame_differencing.py` 使用内存 ImageIO 流。

6.8.4 Constants

`imageio.`**`FILE_STREAM`**

Stream backed by a file on storage (e.g. `/sdcard/stream.bin`).

`imageio.`**`MEMORY_STREAM`**

Stream backed by a fixed-size buffer in PSRAM/RAM.

6.8.5 Classes

class `imageio.`**`ImageIO`**(*stream*, *mode*)

class `imageio.`**`ImageIO`**(*stream*, *count*)

Record and replay sequences of images, preserving inter-frame timing. Exposed as `image.ImageIO`. A file stream reads/writes the OpenMV "OMV IMG STR" container on storage; a memory stream keeps frames in a pre-allocated buffer for fast capture/playback.

Open a file stream for reading ("r") or writing ("w").

参数

- **`stream`** –path to the stream file.
- **`mode`** –"r" to read, "w" to create/overwrite.

`type`()

Return the stream type: FILE_STREAM or MEMORY_STREAM.

`is_closed`()

Return True if the stream has been closed.

`count`()

Return the number of frames recorded in the stream.

`offset`()

Return the current frame index (read/write cursor).

`version`()

Return the container version for file streams, or None for memory streams.

`buffer_size`()

Return the per-frame buffer size for memory streams, or None for file streams.

size()

Return the total stream size in bytes.

write(*image*)

Append one image to the stream (records the elapsed time since the last write).

参数

image –frame to store.

read(*copy_to_fb=...*, ***, *loop=...*, *pause=...*)

Read the next frame from the stream.

参数

- **copy_to_fb** –True loads the frame into the frame buffer (and updates the preview).
- **loop** –True rewinds to the first frame at end-of-stream instead of returning None.
- **pause** –True sleeps to honor the frame's recorded timestamp (real-time playback).

seek(*offset*)

Move the read/write cursor to the given frame index.

参数

offset –target frame index.

sync()

Flush buffered file data to storage (no-op for memory streams).

close()

Close the stream and release its resources.

6.9 h264 –H.264 编码

`h264` 模块把图像编码成 Annex-B H.264 NAL 单元，可用于将视频录制到存储，或送入*rtsp –RTSP 推流* 服务器。该模块可用于 ESP32-P4 构建。

6.9.1 编码单帧图像

```
import sensor, h264

enc = h264.H264Encoder(320, 240, fps=15)
nal = enc.encode(sensor.snapshot())
print("bytes:", len(nal), "keyframe:", enc.keyframe())
enc.close()
```

编码器尺寸必须与每一帧输入图像一致。`encode()` 以 `bytes` 返回一帧编码数据，`keyframe()` 用于判断最近编码的图像是否为 IDR/I 帧。

6.9.2 录制 H.264 裸码流

```
import sensor, h264

FRAME_COUNT = 300
OUT_PATH = "/sdcard/out.h264"

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
```

(续下页)

(接上页)

```
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

first = sensor.snapshot()
enc = h264.H264Encoder(
    first.width(),
    first.height(),
    fps=15,
    bitrate=1_500_000,
    gop=15,
)

try:
    with open(OUT_PATH, "wb") as output:
        output.write(enc.encode(first))
        for _ in range(FRAME_COUNT - 1):
            output.write(enc.encode(sensor.snapshot()))
finally:
    enc.close()
```

该示例写入不包含 MP4 容器的 Annex-B 基本码流，可使用 `ffplay out.h264` 播放，或在主机上重新封装。`bitrate` 表示目标每秒比特数，`gop` 表示以帧为单位的关键帧间隔，QP 上下限则控制允许的质量与体积范围。

6.9.3 资源与吞吐量注意事项

应为固定分辨率创建一个编码器，并在整个视频流中持续复用，使用完毕后再将其关闭。如果采集或存储无法维持配置的帧率和码率，应降低分辨率、帧率或码率，而不是反复创建编码器。

参见:

编解码与推流 说明了“编码—推流”流程，以及 H.264 关键帧与 RTSP 传输之间的关系。

可运行示例：`example/01-Camera/01-H264/record_h264.py`。

6.9.4 Classes

class h264.**H264Encoder**(*width*, *height*, *, *fps=...*, *gop=...*, *bitrate=...*, *qp_min=...*, *qp_max=...*)

Hardware-accelerated H.264 video encoder. Feed it images frame by frame with encode(); it returns Annex-B NAL units ready to mux into a file or stream over the network (see the rtsp module).

Open an encoder for a fixed frame size.

参数

- **width** –frame width in pixels.
- **height** –frame height in pixels.
- **fps** –target frame rate; also the default GOP length.
- **gop** –keyframe (IDR) interval in frames; 0 selects one keyframe per second (== fps).
- **bitrate** –target bitrate in bits per second; 0 auto-selects width*height*fps.
- **qp_min** –lower quantization-parameter bound (better quality, larger frames).
- **qp_max** –upper quantization-parameter bound (lower quality, smaller frames).

encode(*image*)

Encode one image and return its Annex-B NAL units as bytes.

参数

image –source frame; its size must match the encoder width/height.

keyframe()

Return True if the most recently encoded frame was a keyframe (IDR/I).

close()

Release the encoder. Using the object afterwards raises OSError.

6.10 rtsp –RTSP 推流

`rtsp` 模块通过 RTSP 协议发送 H.264 NAL 单元，使 VLC、ffplay 等客户端能通过网络观看摄像头画面。需与*h264 –H.264 编码* 配合使用。该模块可用于 ESP32-P4 构建。

6.10.1 采集、编码与推流

```
import network, sensor, h264, rtsp, time
from machine import Pin

WIDTH, HEIGHT, FPS = 320, 240, 30

lan = network.LAN(
    mdc=Pin(31),
    mdio=Pin(52),
    reset=Pin(51),
    phy_addr=1,
    phy_type=network.PHY_IP101,
)
lan.active(True)

for _ in range(100):
    if lan.isconnected():
        break
    time.sleep_ms(100)
if not lan.isconnected():
    raise OSError("Ethernet connection failed")

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

enc = h264.H264Encoder(WIDTH, HEIGHT, fps=FPS, bitrate=3_000_000)
server = rtsp.RTSPServer(WIDTH, HEIGHT, fps=FPS, listen_port=8554)

print("rtsp://%s:8554/" % lan.ifconfig()[0])
try:
    while True:
        server.send(enc.encode(sensor.snapshot()))
finally:
    server.stop()
    enc.close()
```

上述以太网引脚映射适用于带 IP101 RMII PHY 的 ESP32-P4X-Function-EV-Board，其他产品开发板应使用对应的网络配置。`RTSPServer` 公布的宽度、高度和帧率必须与编码器配置一致。

6.10.2 客户端与队列行为

可使用 `ffplay rtsp://<board-ip>:8554/` 或其他支持 RTSP 的播放器打开输出地址。`send()` 会将一帧编码数据加入队列，并且不会等待客户端；没有客户端连接或客户端处理过慢时，帧可能被丢弃，从而避免阻塞采集循环。如果高分辨率或高码率编码帧超过默认限制，可增大 `max_frame_len`。

6.10.3 停止服务

释放网络或编码器资源前必须调用 `server.stop()`。使用 `try/finally` 可以确保脚本被中断或抛出异常时仍会完成资源清理。

参见:

编解码与推流 完整介绍了采集、编码与推流的整个流程。

可运行示例：`example/01-Camera/02-RTSP/stream_rtsp.py`。

6.10.4 Classes

class rtsp.**RTSPServer**(*width*, *height*, *, *fps=...*, *listen_port=...*, *max_frame_len=...*)

Minimal RTSP server that streams H.264 video over the network. Create it once Wi-Fi is up, then push encoded NAL units from an h264.H264Encoder with send(). Clients connect to `rtsp://<board-ip>:<listen_port>/`. Only video (no audio) is served.

Start the RTSP server and advertise the given stream format.

参数

- **width** –advertised video width in pixels.
- **height** –advertised video height in pixels.
- **fps** –advertised frame rate, used for SDP and pacing.
- **listen_port** –TCP port the RTSP service binds to.
- **max_frame_len** –max accepted encoded frame size in bytes; 0 uses width*height as a safe ceiling.

send(*nal*)

Queue one encoded H.264 frame (Annex-B NAL units) for delivery. Frames are sent in order; if the client is slow or absent the frame is dropped rather than blocking the caller. Frames larger than max_frame_len are ignored.

参数

nal –encoded frame bytes from h264.H264Encoder.encode().

stop()

Stop the server, join its task, and free queued frames.

6.11 uvc —USB 摄像头输出

启用 USB-OTG Device 的板卡启动时同时提供 CDC、MSC 和 UVC。PC 摄像头软件打开 ESP-VISION Camera 即可接收图像。默认输出固件内置 ESP-VISION logo，不初始化或修改 `sensor`。禁用 USB Device 的板卡不提供此模块。

首版支持 MJPEG、320×240、10 FPS，使用等时传输，JPEG 帧上限为 48 KiB。实际帧率受主机调度和其他 USB 流量影响。`get_mode()` 返回支持的模式；`is_streaming()` 仅表示主机已启用流接口，不能证明某个软件已显示图像。

uvc.**write**(*img*)

发布处理后的 QVGA `image.Image`。主机未开启输出时返回 `False`；帧已复制到固件持有的缓冲区时返回 `True`，不代表主机已收到。RGB565、灰度和 JPEG 输入通过共享 JPEG 编码层处理。尺寸错误或编码结果超限抛出 `ValueError`，编码或分配失败抛出 `OSError`。返回后可以立即复用相机缓冲区。待发送帧可被新帧替换，正在发送的帧不会被覆盖。

uvc.**set_frame**(*img*)

复制固定 QVGA 图像，无需主机已打开。解除 preview 绑定，持续重复该帧，直到替换或 reset。成功返回 `True`，失败时保留旧图源。

uvc.**bind**(*source*)

`"preview"` 绑定到 `img.flush()` 发布的处理结果。UVC 和 EV-MUX 共用一次 JPEG 编码，并独立判断传输就绪状态；无需 IDE 会话即可使用 UVC。其他尺寸的帧不送往 UVC，仍可通过 EV-MUX 预览。帧超限或 UVC 分配失败时丢帧，不影响 EV-MUX。`None` 解除绑定并恢复 logo。新绑定在收到匹配帧前也显示 logo。

uvc.**reset**()

解除图源并恢复 logo，不重置 USB 控制器，不断开 CDC/MSC。VM 软复位也恢复 logo。实时生产者暂停或停止发布时，默认重复最后一帧。

uvc.**is_streaming**()

返回主机是否已启用流接口且 USB 总线未挂起。

uvc.**get_mode**()

返回 `{"format": "MJPEG", "width": 320, "height": 240, "interval_us": 100000}`，仅公布并接受这一模式。

相机处理结果可以同时送往两种输出：

```
import sensor
import uvc

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
uvc.bind("preview")
try:
    while True:
        img = sensor.snapshot()
        img.draw_rectangle((20, 20, 100, 80), color=(255, 0, 0))
        img.flush()
finally:
    uvc.reset()
```

无需 EV-MUX 预览时，也可直接使用 `uvc.write(img)`。同一帧不要同时走两条发布路径。USB 回调内不执行 Python 回调或相机抓帧。

6.12 storage –存储诊断与确认恢复

可在不修改数据的情况下检查卷、重试挂载，或明确确认恢复未挂载的 Flash `ffat`。存储失败通过诊断结果返回；无效参数会抛出 Python 异常。数据保护、备份和恢复的完整规范见*存储、文件系统与数据保护*。恢复成功后，可能需要用户选择软复位来发布 Python 挂载。

当前不支持在线格式化 SD，也不支持格式化已挂载卷。用户同意擦除选定卷后，调用方才可传入布尔值 `True`；不能在遇到错误后自动设置确认参数。

6.12.1 Functions

storage.**status**(*volume=...*)

Return non-destructive status and recovery guidance for flash or sd. Storage faults are represented in the result, not by rebooting the board.

参数

volume – “flash” (default) or “sd”.

storage.**retry**(*volume=...*)

Retry mounting without formatting or erasing any data, including when board policy disables automatic mounting. Check ok and operation_error; after success, soft_reset_required requests an explicit soft reset to publish the Python mount.

参数

volume – “flash” (default) or “sd”.

storage.**format**(*volume=...*, *, *confirm=...*)

Explicitly initialize an unmounted flash data/FAT partition named ffat. This destroys files and requires the literal confirm=True after user approval and any required backup. Mounted volumes, legacy vfs, open handles, quarantined writers and online SD formatting are not supported. Check ok, operation_error and soft_reset_required; failure never triggers automatic reboot or another format.

参数

- **volume** – “flash” (default); “sd” returns an unsupported operation result.
- **confirm** –Must be the boolean True; omission, False and truthy strings are rejected before any write.

Chapter 7

操作指南

面向具体任务的 ESP-VISION 扩展指南。

7.1 用 AI 开发视觉应用

把想法告诉 AI 助手，一起将它变成运行在 ESP-VISION 开发板上的 Python 脚本。助手可以查找示例、准备固件和模型、部署应用、查看摄像头画面和日志，并根据实际效果持续改进。

7.1.1 按固件版本查询知识

插件在查询 API、硬件能力和示例前，会读取所选开发板的实际固件版本。在线 Knowledge 服务保留历史快照，每次内容查询都必须指定准确的固件版本。更新插件不要求同步升级固件；旧固件可以继续使用对应的知识快照。

烧录或切换开发板后，助手会重新检查设备。若固件版本缺失或没有匹配的快照，会说明原因并提醒更新官方固件。只有你明确选择不更新，助手才会使用当前可用的最新官方固件 Knowledge 快照，告知所选版本，并说明其 API 可能与设备存在差异。最新版由官方快照目录指定，助手保留原始 TAG，不按字符串猜测先后。助手会等待你的选择，不会静默切换版本；更新失败或无法读取知识目录也不会触发回退。切换设备、重新连接或烧录后，这次选择会失效。开发版本后缀会原样保留。离线提问时请明确目标版本，例如“说明固件 2026.10.21 的摄像头拍照方法”。助手可以先列出已支持的版本。

7.1.2 接入你的 AI 助手

ESP-VISION 插件将官方知识和本地设备工具接入 AI 助手。可以在 官网上手入口 选择对应客户端的接入方式。

WorkBuddy 与 QoderWork

推荐在应用市场搜索 `esp-vision`，找到对应条目并安装。以下名称与官网展示的应用市场条目一致。

表 1: 应用市场条目

客户端	条目名称
WorkBuddy	嵌入式视觉开发专家	ESP-Vision AI 视觉
QoderWork	ESP-Vision 嵌入式 AI 视觉开发

Codex 与 Claude Code

使用 Codex 时，可以这样开始：

备注： 帮我在 Codex 中用上 ESP-VISION。官方安装说明：https://www.npmjs.com/package/@esp-vision/plugin

使用 Claude Code 时：

备注： 帮我在 Claude Code 中用上 ESP-VISION。官方安装说明：https://www.npmjs.com/package/@esp-vision/plugin

Cursor 与其他助手

Cursor 和其他客户端的接入方式见 插件安装指南。

备注： 我想在当前 AI 助手中使用 ESP-VISION。官方接入说明：https://github.com/espressif/esp-vision/blob/master/host/esp-vision-plugin/README.md

接入后，在项目中开启新会话。通过 USB 开发时，AI 助手需要能访问连接开发板的计算机。

7.1.3 接入开发板并运行应用

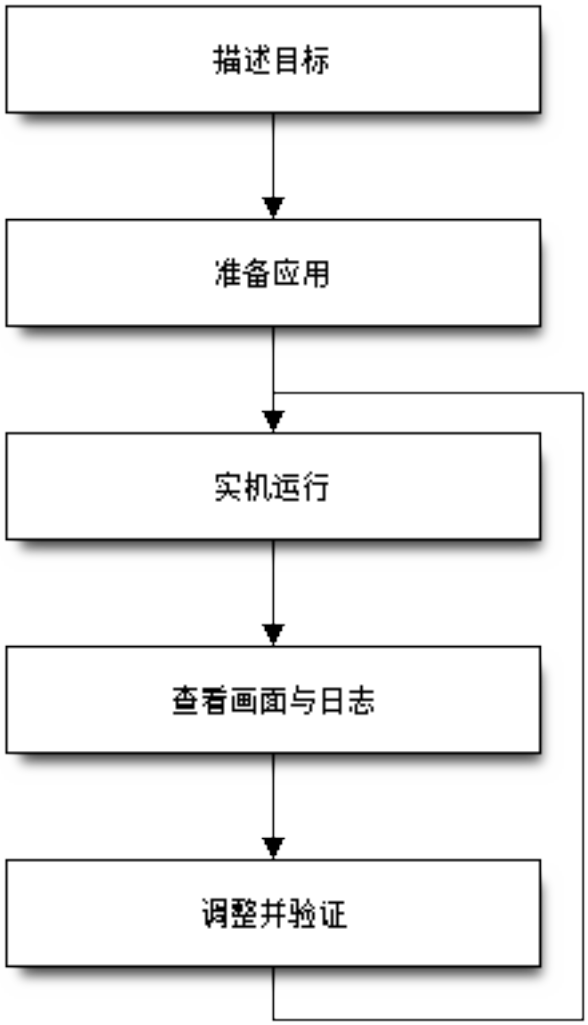
准备一块*ESP-VISION 已支持的开发板*，用 USB 数据线连接电脑。首次使用时，助手可以完成官方固件的下载与烧录，将开发板准备到可运行视觉应用的状态：

备注： 这是一块 ESP-VISION 支持的新开发板。帮我装好 ESP-VISION 固件，让摄像头能正常拍照。板内没有需要保留的数据。

完成首次固件安装后，就可以开始开发应用。已有兼容固件的开发板可以直接从应用需求开始；配置过网络的设备也可以通过局域网连接。

备注： 在开发板屏幕上显示实时摄像头画面，加一个帧计数，保持运行让我看看。

应用运行后，屏幕上的画面和帧计数会持续更新。你也会得到对应的 Python 脚本，方便继续修改和复用。



7.1.4 日常任务

从想实现的功能、使用场景或遇到的问题开始交流。插件中的 workflow 负责具体实现，你可以围绕结果继续提出需求。

颜色跟踪

备注： 让摄像头跟踪画面中的红色物体，实时框出它的位置，物体移动时框也跟着移动。

在镜头前移动红色物体，看看跟踪框是否能跟上。需要关注的颜色、物体或运动速度，都可以作为后续需求补充。

升级固件

备注： 把开发板升级到最新兼容的官方 ESP-VISION 固件，保留我的脚本和设置。

升级完成后，你会得到实际安装版本和设备运行结果。

了解连接的设备

备注： 电脑上连着的这块开发板是什么型号、装了什么固件，能做哪些视觉功能?

设备信息可以帮助你选择适合的应用方向，例如图像处理、目标检测或画面显示。

人脸检测

备注： 让开发板识别人脸，在屏幕上实时框出来，同时显示检测帧率。

你可以观察人脸进入、移动和离开画面时的检测效果，并根据显示的帧率判断流畅度。

二维码识别

备注： 我想用摄像头读取二维码，在屏幕上框出二维码并显示内容，移动二维码时持续更新。

可继续补充使用场景，例如希望识别较小的二维码，或让内容显示得更清楚。

通过 Wi-Fi 开发

备注： 帮我把开发板连上 Wi-Fi，之后通过无线连接修改程序、查看摄像头画面。

首次配网时用 USB 连接支持 Wi-Fi 的开发板，按助手提示提供网络信息。配网成功后可通过局域网继续开发，设备重新上电时会自动连接已保存的网络。也可以在 官网 的“浏览器烧录入口”中，将“烧录方式”设为“仅配置 WiFi”，连接串口后填写网络信息并保存。完整配网流程和连接排查见*主机接入、Wi-Fi 配置与恢复实践*。

USB 摄像头

备注： 把开发板变成 USB 摄像头，让电脑能看到实时画面，并把检测到的人脸框出来。

使用支持 USB-OTG Device 的开发板，并连接其 USB-OTG 接口。在电脑摄像头软件中选择 `ESP-VISION Camera` 查看效果。支持的图像格式、分辨率和 API 见*uvc —USB 摄像头输出*。

排查并修复问题

备注： 摄像头程序启动后，开发板屏幕就黑了，帮我修好。

把实际看到的现象告诉助手，有报错或图片时一起提供。也可以说明问题出现前应用原本的表现。

7.1.5 通过对话改进效果

在同一个会话里继续交流，助手可以结合项目、设备和前面的结果进行调整。例如，跟踪应用运行起来后，可以继续说：

备注： 光线暗一点时，红色物体的跟踪框会闪，我希望它在暗光下也能稳定跟踪。

对于检测应用，可以继续说：

备注： 状态文字太小了，调大一些，再加上检测帧率。

用实际体验描述下一步目标，例如跟踪更稳定、响应更快、画面更清楚或文字更易读。

7.1.6 对话背后的工具

主机侧工具将自然语言开发连接到官方知识和真实设备：

表 2: ESP-VISION 主机工具

工具	作用
Plugin（`@esp-vision/plugin`）	为 AI 客户端提供 ESP-VISION Skill、随包 Device CLI 和官方知识连接。
Device CLI（`@esp-vision/device-cli`）	供助手在本地主机上发现设备、管理固件和文件、运行 Python 脚本以及采集图像。
Knowledge MCP	提供官方 API 文档、开发板能力和示例，供助手查询参考。

进一步了解工具和开发约定，可阅读*Device CLI 参考*、*Python 脚本开发说明* 和*主机 SDK 指南*。

7.2 添加新的 Python 模块

ESP-VISION 通过 MicroPython 的 `USER_C_MODULES` 机制暴露 Python 模块。绑定层位于 `modules/`，仅做对象转换与轻量 API 适配；重逻辑应放在纯 C 或 `platform/` 中。本指南以添加名为 `foo` 的模块为例。

7.2.1 总览

```
modules/py_foo.c              # 绑定 + MP_REGISTER_MODULE(MP_QSTR_foo, ...)
modules/qstrdefs_esp_vision.h   # 任何受功能开关控制的 qstr
micropython.cmake             # 将 py_foo.c 加入源文件列表
stubs/foo.pyi                 # 供 IDE 补全的类型存根
docs/.../api-reference/foo.rst  # 参考页面
```

7.2.2 1. 创建绑定源文件

新增 `modules/py_foo.c`：定义模块函数，构建 globals 字典，并自注册模块。现有模块是最佳模板——纯函数模块可参考 `modules/py_sensor.c`，类型/类模块可参考 `modules/py_display.c`。

```
/*
 * SPDX-FileCopyrightText: 2026 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#include "py/runtime.h"

static mp_obj_t foo_hello(void) {
    return mp_obj_new_int(42);
}
static MP_DEFINE_CONST_FUN_OBJ_0(foo_hello_obj, foo_hello);

static const mp_rom_map_elem_t foo_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_foo) },
    { MP_ROM_QSTR(MP_QSTR_hello), MP_ROM_PTR(&foo_hello_obj) },
};
static MP_DEFINE_CONST_DICT(foo_module_globals, foo_module_globals_table);

const mp_obj_module_t mp_module_foo = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&foo_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_foo, mp_module_foo);
```

C++ 模块使用 `.cpp``（参见 ``modules/py_espdl.cpp`）；构建已为 C++ 源文件设置 `-std=gnu++2b`。

7.2.3 2. 按需注册 qstr

大多数 qstr（`MP_QSTR_foo`、`MP_QSTR_hello`）会自动从源码收集。仅当名称受板级功能开关控制、或无法被 qstr 扫描器识别时，才需以 `Q(name)` 形式添加到 `modules/qstrdefs_esp_vision.h`。

7.2.4 3. 将源文件接入构建

在 `micropython.cmake` 的 `ESP_VISION_MODULE_SOURCES` 中加入该源文件：

```
set(ESP_VISION_MODULE_SOURCES
    ${ESP_VISION_ROOT}/modules/py_display.c
    ${ESP_VISION_ROOT}/modules/py_image.c
    ${ESP_VISION_ROOT}/modules/py_foo.c    # 新增
    ...
)
```

若模块仅在部分芯片上有效，请放入对应的 `IDF_TARGET` 判断块中（H.264 与 RTSP 模块即以此方式限定为 `esp32p4`）。若需额外组件，可像 `idf::zxing` 那样用 `target_link_libraries` 链接。

7.2.5 4. 添加类型存根

创建 `stubs/foo.pyi` 描述公开接口，便于 IDE 补全。保留 Apache SPDX 头并与现有存根风格保持一致。

7.2.6 5. 编写模块文档

新增 `docs/en/api-reference/foo.rst` 与 `docs/zh_CN/api-reference/foo.rst`，使用 Python 域指令（`.. py:module::`、`.. py:function::`、`.. py:class::`），并在两种语言的 `api-reference/index.rst` 的 `toctree` 中加入 `foo`。

7.2.7 6. 构建并验证

```
idf.py --board ESP32_P4X_EYE build
```

然后在 REPL 中检查：

```
import foo
print(foo.hello())
```

7.3 贡献与审阅

请在 GitHub 上的 ESP-VISION 向对应分支提交 Pull Request（PR），说明改动、测试和使用的硬件。遵循仓库的版权与许可证规则，并在 `CHANGELOG.md` 的 `Unreleased` 部分添加条目。维护者在 GitHub 上审阅 PR，并在需要时提出修改意见。

7.3.1 同步与合并流程

1. 审阅完成后，维护者以 `sha=<commit>` 评论当前 PR 的最新提交 SHA，并添加 `PR-Sync-Merge` 或 `PR-Sync-Rebase` 标签。评论和标签必须由同一位维护者提交。当前 Action 会拒绝修改路径中包含 `.github` 或 `.gitlab` 的 PR；此类修改请与维护者协调。
2. Action 创建 GitLab MR。贡献者更新 PR 后，维护者评论新的最新提交 SHA，并在保留原合并或变基标签的同时添加或重新添加 `PR-Sync-Update`。
3. 维护者在 GitLab 审阅 MR，运行适用的 CI 和硬件测试，并在 GitHub PR 上反馈需要修改的内容。通过后，将 MR 合并到 GitLab 的目标分支。
4. GitLab CI 将通过测试的分支同步到 GitHub。维护者核对 PR 状态并关联最终提交；如果改写提交历史导致 GitHub 未自动识别，则说明情况后手动关闭 PR。

工作流需要配置 GitHub 密钥并能够访问内部 GitLab。同步失败或尚未完成不代表贡献已经合并。此流程不需要 Jira 项目或 Jira issue。

7.4 客制化固件功能

ESP-VISION 将面向应用的模块、图像算法、标准 MicroPython 功能、冻结 Python 代码和板级服务拆分为相互独立的配置层。产品固件既可以移除不需要的能力，以减少 Flash、RAM、依赖项和攻击面，也可以增加产品专用模块，而无需重写摄像头与媒体处理链路。

7.4.1 确定客制化范围

修改配置前，应先确定变更是面向所有 ESP-VISION 固件，还是仅面向某块开发板。修改根目录 `micropython.cmake` 会影响所有满足对应芯片条件的开发板；修改 `boards/<BOARD>/``（包括其中的 MicroPython 移植文件 ``boards/<BOARD>/port/`）仅影响对应开发板。对于产品变体，应按照添加新的开发板 创建独立板级包，而不是直接修改共享的开发板配置。

7.4.2 客制化 ESP-VISION Python 模块

`micropython.cmake` 中的 `ESP_VISION_MODULE_SOURCES` 列表决定暴露给 Python 的 C/C++ 绑定。移除源码时，应同时处理其依赖的辅助源码；增加源码时可参考*添加新的 Python 模块*。仅适用于部分芯片的模块应放在对应的 `IDF_TARGET` 条件中，例如 H.264 与 RTSP 当前仅为 `esp32p4` 启用。

```
set(ESP_VISION_MODULE_SOURCES
    ${ESP_VISION_ROOT}/modules/py_display.c
    ${ESP_VISION_ROOT}/modules/py_image.c
    ${ESP_VISION_ROOT}/modules/py_imageio.c
    ${ESP_VISION_ROOT}/modules/py_helper.c
    ${ESP_VISION_ROOT}/modules/py_sensor.c
)
```

移除绑定源码不会自动移除其背后的所有平台服务或托管组件。修改源码列表后，还需检查 `target_sources`、`target_link_libraries`、组件清单和最终 size 报告，确认不需要的依赖已不再链接。

7.4.3 客制化图像算法

每块开发板的 `boards/<BOARD>/imlib_config.h` 用于选择可选的 `imlib` 算法。删除某个 `IMLIB_ENABLE_*` 定义可排除不使用的算法，增加受支持的定义可启用其他算法。常见功能组包括滤波、几何检测、二维码和 AprilTag。

```
#define IMLIB_ENABLE_MEAN
#define IMLIB_ENABLE_GAUSSIAN
#define IMLIB_ENABLE_QRCODES
#define IMLIB_ENABLE_APRILTAGS
```

部分 Python 方法在后端关闭后仍会保留在绑定层中，并在调用时抛出异常。应确保 API 文档、示例和自动生成的开发板支持表与所选算法保持一致。未经许可证审查，不应移除 `OMV_NO_GPL`。

7.4.4 客制化标准 MicroPython 功能

使用 `boards/<BOARD>/port/mpconfigboard.h` 覆盖单块开发板的标准 MicroPython 功能宏，例如网络、Bluetooth、ESP-NOW、ADC、SD 卡、USB，以及其他 `MICROPY_PY_*` 和 `MICROPY_HW_*` 选项。CMake 级选项和板级 sdkconfig 链则在 `mpconfigboard.cmake` 中配置。

```
#define MICROPY_PY_BLUETOOTH (0)
#define MICROPY_PY_ESPNOW (0)
#define MICROPY_PY_NETWORK_WLAN (1)
```

这些宏可能依赖 ESP-IDF 版本与 SoC 能力宏。关闭 Python 功能后，可能还需移除对应的 ESP-IDF 配置或托管依赖，才能获得可观测的固件体积缩减。

7.4.5 配置内存放置策略

带 PSRAM 的开发板通过 `boards/<BOARD>/port/` 下的 sdkconfig 配置链设置分配边界。`CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL` 是 ESP-IDF 默认分配器和 ESP-VISION 自动分配策略使用的字节阈值：不大于该阈值的分配优先使用内部 RAM，更大的分配优先使用 PSRAM。`CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL` 则单独为驱动和其他必须使用内部 RAM 的分配预留内存。需要调整内存分配比例时，应修改开发板的 sdkconfig 默认值。

大型图像、模型、帧和临时缓冲区在所需能力允许时优先使用 PSRAM。调用方要求直接 DMA 访问或可执行访问时，分配回退仍保留该能力。P4X_EYE 的 SPI 显示也可以把普通 8-bit framebuffer 作为绘制源，因为 LCD 传输层会将其分块搬运到有界 DMA 传输缓冲区；这是明确的复制路径，不会放宽直接 DMA 的要求。没有 PSRAM 的开发板会在能力允许时使用内部内存。

7.4.6 客制化冻结 Python 代码

开发板的 boards/<BOARD>/manifest.py 控制冻结到固件中的 Python 模块。可根据 MicroPython manifest 机制使用 freeze()、module()、package() 或 include() 增加产品启动代码、库或软件包；不需要某个冻结模块时则删除对应条目。

```
freeze("$(PORT_DIR)/modules")
freeze("$(ESP_VISION_ROOT)/modules", "py_inisetup.py")
freeze("$(ESP_VISION_ROOT)/boards/<BOARD>", "board_inisetup.py")
include("$(MPY_DIR)/extmod/asyncio")
```

冻结代码可提升部署一致性并确保启动阶段可用，但会占用固件 Flash。开发期间或产品部署后仍需替换的文件，应保留在根文件系统中（例如 /lib 下的软件包）或 /sdcard 中。

7.4.7 客制化板级服务与可选组件

板级摄像头、显示和 SD 卡实现位于 boards/<BOARD>/。存在 camera.c、display.c 和 sdcard.c 时，micropython.cmake 会自动选用。板级 CMake 开关应放在 boards/<BOARD>/board.cmake 中，例如当前 P4 开发板通过 ESP_VISION_ENABLE_BARCODE 启用可选的 ZXing 条码后端。

```
set(ESP_VISION_ENABLE_BARCODE OFF)
```

增加组件时，应注册其源码或 IDF 组件，并将其链接到 usermod_esp_vision_platform；移除组件时，应一次性移除源码、头文件路径、编译定义、链接依赖和组件清单条目。

7.4.8 构建与验证

修改 CMake、sdkconfig、manifest 或功能宏后，应重新配置并构建所选开发板：

```
idf.py --board <BOARD> reconfigure
idf.py --board <BOARD> build
idf.py --board <BOARD> size
```

在 REPL 中验证模块导入和受影响的 API，运行相关示例，并对比修改前后的 size 报告。还应重新构建文档，因为模块导航和开发板能力摘要会根据固件配置自动生成。

7.5 发布合格固件

固件与 Device Core 独立发布。发布 npm 包（包括 Device Core 0.3）不会运行固件上传任务，只有符合发布 CI 规则的固件标签才会运行 pack_launchpad、qualify_firmware_release 和 upload-dl-trigger。2026.08.27 等已有版本早于元数据生成功能，因此二进制下载可能正常，但 firmware/<board>/<version>.json 返回 HTTP 404。

7.5.1 发布包与 URL 契约

打包器复用发布矩阵的 IDF 产物，不重新编译固件。binaries/ 产物包含：

```
config.toml
releases/<tag>/manifest.json
esp-vision-<board>-<tag>.bin
esp-vision-<board>-<tag>-app.bin
firmware/<board>/<tag>.json
```

上传器必须将 config.toml、releases/ 和 firmware/ 保留在 https://dl.espressif.com/AE/esp-vision/ 根目录，两种二进制文件则放入 bin/。Core 根据根配置 URL 查找元数据；即使生成成功，误放到 bin/firmware/ 也会导致 HTTP 404。上传器仅接受清单声明的发布对象和可选 MCP 文件，拒绝无关产物。

按八个发布板型计算，默认布局有 16 个 BIN、8 个逐板 JSON、1 个不可变清单和 1 个根配置，共 26 个对象；加上可选的版本化 MCP 包及 bin/mcp/latest.json 兼容 alias 为 28 个。单 BIN 模式去掉 8 个应用 BIN 和 8 个逐板 JSON，剩下 8 个 BIN 加清单/config，共 10 个对象（含 MCP 为 12 个）。源码配置仍只生成一份 TOML；服务端以根 config 内一个权威可变固件入口引用每版不可变清单。MCP latest alias 是独立的兼容便利入口。

生成后，tools/ci/launchpad/validate_release.py <tag> 检查每个发布板型，包括未在官网展示的板型。缺少应用升级或完整安装元数据、板型/版本/构建身份不匹配、URL 错误、二进制损坏、引导程序/应用/分区表散列错误、芯片修订范围交集错误或分区越界都会被拒绝。仅要求本次发布具有元数据，发布列表中的历史版本无需补齐。检查失败时发布任务停止，不会将不完整的包交给上传器。

7.5.2 不可变清单与当前入口

每版只有一个不可变 releases/<tag>/manifest.json，包含 schemaVersion = 1、版本、完整 40 位 sourceCommit、正整数 CI pipelineId、layout、全部板型发布描述、该版完整 website 快照及明确的 mcp 描述或 JSON null。模型下载链接固定到源提交，不使用漂移分支。根 config.toml 保留 launchpad 字段和兼容网站快照，并增加 [publication]：同一身份、manifestUrl、清单原始字节的 SHA-256、previousConfigSha256 和 releases 记录。每条记录只引用该版清单及哈希，并列出实际发布的非空、唯一 boards 集合。

打包器读取上一份公开 config，验证其全部历史清单的哈希、身份与板型集合后原样继承记录。下载或校验失败直接阻断，不丢弃历史继续发布。Git 标签不证明发布成功，不会自动成为记录。旧网站版本标签仅保留下载入口，不能凭空获得清单或板型与版本的组合。已经发布的版本不能同名重新打包。

manifest.website 是该版板型和模型的权威快照。根 config.website 为兼容旧消费者镜像最新快照。历史版本必须跟随自己的 publication 记录，不能把今天的板型列表与所有版本标签做笛卡尔积。

7.5.3 构建导出的 Wi-Fi 能力

新发布项的 capabilities.wifi 来自 release-build-identity.json。CI 在成功构建后复用网络服务源文件的真实编译命令，仅运行预处理，沿用实际 include 路径和宏定义。配网能力要求 MICROPY_PY_NETWORK_WLAN && ESP_VISION_ENABLE_BACKGROUND_NETWORK && (CONFIG_ESP_WIFI_ENABLED || CONFIG_ESP_WIFI_REMOTE_ENABLED)。身份文件还绑定板型、固件版本、完整源提交及应用 SHA-256。缺失或错配身份不能通过资格门禁；不能用展示数据或板型白名单宣称支持 Wi-Fi。

7.5.4 可选单 BIN 布局

默认仍为 ESP_VISION_PUBLICATION_LAYOUT=legacy。只有支持 manifest 和切片的 Core 版本及官网消费者已部署后，才显式启用 single-bin。它只输出八个完整 BIN 和中央清单，不生成应用 BIN 或逐板 JSON。应用的 image.source 包含完整容器的 url、bytes、sha256 及应用在容器内的字节 offset。image.url 等于容器 URL，而 image.bytes 和 image.sha256 仍描述应用。source 必须与 install 的完整容器一致，切片偏移等于应用烧写地址。Core 先验证完整容器和应用切片，再仅烧写应用；完整安装继续执行独立的布局与安全校验。

7.5.5 MCP 与发布资格

先生成并按 schema 验证 MCP，再组装清单。版本化描述绑定本版包的字节、哈希、版本和完整源提交；失败默认阻断。只有显式 ESP_VISION_ALLOW_MISSING_MCP=1 才允许 mcp: null，清单不能引用旧 latest.json。兼容 alias 由上传器在当前入口发布成功后维护，不作为发布资格输入。

打包复用构建产物，可以与硬件检查并行。`qualify_firmware_release` 明确等待全部必需固件矩阵、Function 板的三组 IDF HIL、SDK 和 host/config/MCP 检查，并以实际提交及流水线 ID 全量校验包。上传 trigger 同时依赖打包和资格门禁成功。上传器再次核对这两个 job，先上传并验证不可变资产与清单，最后基于 `previousConfigSha256` 比较并交换可变 config 入口。过期或并发候选不能覆盖更新的历史。

7.5.6 发布流程

1. 先合入新的上传器与 Core 契约，并保留根目录布局修复依赖。旧消费者不能启用单 BIN 模式。
2. 打标签前分别更新 Plugin 和 Knowledge MCP 的 npm 版本，并在两个包的清单中声明相同的 Core 和 CLI minor 范围及本次解析版本。指定的 CLI 必须已经发布且实际打包指定 Core；Plugin 打包该已发布 CLI。patch 版本只能兼容地修复问题，只有 minor 版本可以增加兼容功能，只有 major 版本可以引入不兼容变更；patch 和 major 版本不得额外添加新功能。Core 和 CLI 的版本由受保护默认分支流水线独立发布。
3. 先确认该完整源提交已成功镜像到 GitHub，固定到提交的模型链接已经可用，再通过正常受保护流程创建新固件标签；上传器在推进 config 前会拒绝尚不可用的源提交。等待构建、HIL、SDK 及包资格通过；npm 发布不等于固件发布。
4. 用根 config 的 publication 哈希验证版本清单、精确板型集合、website/MCP 快照和镜像哈希。配置 dry-run 或分支 CI 成功不代表生产上传成功。

历史 BIN 和不可变清单继续保留。不要为旧镜像编造元数据、覆盖同名版本或从 git 标签猜兼容性。离线验证和合入这些修改不会发布标签、npm 包或 CDN 数据。

7.5.7 Iris 固件文件

`ESP_VISION_ENABLE_IRISFW` 是默认关闭的 CMake 选项和 C 编译宏，仅支持 `ESP32_S31_MOSAICO`；其他板型开启时配置失败。普通 Launchpad BIN 构建继续采用原 factory 布局。独立的 `firmware_irisfw_mosaico` 作业使用 IDF `release-v6.1` 开启该宏，产物存放在 `build/ESP32_S31_MOSAICO/idf6.1-irisfw`。固件标签的 `pack_irisfw` 只消费这一作业，其他板型不生成 Iris 包。

切换选项时使用新的构建目录。构建会核对实际分区选择、QIO Flash 模式和回滚配置，复用旧配置也不能跳过检查：

```
idf.py --board ESP32_S31_MOSAICO -B "$ESP_VISION_WORK_DIR/build/irisfw" \
  -D ESP_VISION_ENABLE_IRISFW=ON build
```

保留区布局

16 MiB 布局遵循 Mosaico Recovery ABI 2 的固定系统前缀，产品契约为 `esp-mosaico/v2`，布局 ID 为 `mosaico-retained-test-2m-v2`，分区表仍位于 `0x8000`。应用 NVS 地址保持不变；7 MiB 应用分区之后的全部剩余空间划给 FAT，直至 16 MiB Flash 末尾，不再单独预留 `ui_apps`：

下表也列出命名分区之前的区域。bootloader 一行表示预留地址窗口，并非镜像实际大小；它和分区表扇区都不是应用分区。Flash 的结束边界为 `0x1000000`，不含该地址。

表 3: 完整 Iris Flash 布局

区域	类型/子类型	起始地址	大小
保留前缀	命名分区之外	0x0	0x2000 (8 KiB)
保留 bootloader 窗口	命名分区之外	0x2000	0x6000 (24 KiB)
分区表	命名分区之外	0x8000	0x1000 (4 KiB)
otadata	data/ota	0x9000	0x2000 (8 KiB)
phy_init	data/phy	0xB000	0x1000 (4 KiB)
sysmeta	data/nvs	0xC000	0x14000 (80 KiB)
vibe_mode	app/test	0x20000	0x1C0000 (1.75 MiB)
coredump	data/coredump	0x1E0000	0x20000 (128 KiB)
nvs	data/nvs	0x200000	0x10000 (64 KiB)
main_app	app/ota_0	0x210000	0x700000 (7 MiB)
ffat	data/fat	0x910000	0x6F0000 (6.9375 MiB)

固定的 `0x210000` 应用起点使应用与 FAT 合计只有 13.9375 MiB；两者各占 7 MiB 会超出 16 MiB Flash 末尾 64 KiB，因此默认分配应用 7 MiB、FAT 6.9375 MiB。

打包器拒绝超过 7 MiB 应用分区的镜像。此布局将原 UI 资源区并入 FAT；修改已有 FAT 分区的起始地址或容量不会自动迁移或扩展文件系统，切换布局前应保存需要保留的文件。

Iris 构建直接从 NVS 分区 `sysmeta`、命名空间 `shared` 的字符串键 `wifi_ssid` 和 `wifi_password` 读取配网信息，有效配置默认启用。只有两个共享键都不存在时才只读回退到旧版 `sysmeta/wifi` 的 `ssid/password`；启动时不复制配网信息、不写 NVS。此构建不使用普通 ESP-VISION 的 `nvs/ev_network` 记录或出厂配网信息。没有配网记录时保持未配置；记录不完整、超长或读取失败时报告配置错误，保留原数据。

ESP-VISION 配网操作写回相同的共享键，另用 `ev_enabled`` (u8) 和 ``ev_revision`` (u32) 保存服务的启用状态及修订号；Vibe Mode 继续使用共享 SSID/密码。忘记网络写入一对空字符串，避免任一固件重新启用旧版凭据。操作不擦除 ``sysmeta，也不改动其他共享设置。

此包要求设备已安装兼容的 Vibe Mode，版本不低于 0.2.0，通过其原生 System Update 流程安装 `.irisfw`。这个构建的原始 IDF 烧录清单不是 Vibe Mode 初始化包，其中的 bootloader 和空白 OTA 元数据不能替换保留系统。从普通 factory 布局切换不会迁移原有文件。新的 `ffat` 遵循板级文件系统初始化策略，更新包不携带空白文件系统镜像。

已有 Iris 布局的应用直刷

根目录 `app` 和 `app-flash` 命令也支持所有受支持板型的普通固件，见仅烧录应用。本节介绍已经采用 Iris 布局并启动 `main_app` 的设备所需的专用流程。先确认板型、运行应用和分区布局；分区定义以 `boards/ESP32_S31_MOSAICO/port/partitions-16MiB-irisfw.csv` 为准。仅烧应用不能把普通 factory 布局转换为 Iris 布局。

1. 选择受支持的 ESP-IDF 环境，将 `ESP_VISION_WORK_DIR` 设为 checkout 之外的可写工作区。使用独立的 Iris 构建目录，并显式开启 `ESP_VISION_ENABLE_IRISFW`。应用必须来自此 Iris 配置；普通 factory 构建或完整的 `.irisfw` 压缩包都不能作为直接写入的应用镜像。
2. 释放其他客户端和串口监视器。对于正在运行的 ESP-VISION，已安装 CLI 的 `control --action bootloader` 可进入 ROM 下载模式，参见设备管理命令行。等待 USB 重新枚举，核对同一块板子，将 `PORT` 设为其下载端口；不要假定应用端口名称保持不变。该操作不会进入 Vibe Mode。
3. 在仓库根目录执行下方命令。IDF 会按需构建应用，并仅烧录该应用。
4. 复位后通过应用端口重新连接，核对板型、版本和运行中的 `main_app` 地址；涉及相机的改动还应抓取一张新帧。写入完成本身不能证明预期应用已启动。

```
idf.py --board ESP32_S31_MOSAICO -B "$ESP_VISION_WORK_DIR/build/irisfw" \
  -D ESP_VISION_ENABLE_IRISFW=ON -p "$PORT" app-flash
```

IDF 目标从当前构建的分区表取得应用地址，本布局的 `main_app` 为 `0x210000`。它只写入 `esp-vision.bin`，保留设备上的 bootloader、分区表、OTA 元数据、Vibe Mode、NVS 和 FAT。启用 Iris 选项时，`flash`` (包括 ``build flash monitor`) 也会选择同一个仅应用烧录目标，并输

出提示。后续调用会识别该构建目录 CMake 缓存中的选项；显式 `-D` 值优先。普通固件与 Iris 固件应使用独立构建目录。普通构建的 `flash` 仍执行完整烧录。

直接烧录使用芯片的 ROM 下载接口，与 Vibe Mode 不同。ESP-VISION 的 USB bootloader 控制可以进入 ROM 下载模式；烧录前应选择重新枚举的下载端口。仅烧应用不会为 factory 布局的板子安装此分区布局，也不会改变启动应用的选择。它适用于运行时开发测试，但不经过原生 Iris 更新事务、HEALTHY 验收或安装回执流程；这些项目仍需手动进入 Vibe Mode 并安装 `.irisfw` 单独验证。

原生 Iris 安装确认

MicroPython 和 EV-MUX 就绪后，启用 Iris 宏的应用确认待验证 OTA 镜像，并在现有 ABI 2 恢复元数据中记录应用地址。同一宏在现有 USB CDC 接口启用最小 Iris 0.2 安装验收子集：硬件与固件身份、认证绑定、BOOT/LINK_READY/HEALTHY 事件、状态、ping/时间及只读系统清单。原生更新器无需修改，即可核对新启动、准确 ELF 身份、分区表哈希和保留的安装操作回执。HEALTHY 要求启动已确认、VM 就绪且复位原因不是崩溃；外设仍须分别验收。

兼容代码复用 EV-MUX 的先到先得会话租约和发送队列。仅在 EV-MUX 帧之外解析 Iris 记录，发送时不插入已经开始的 EV-MUX 帧。Iris owner 不会启用 EV-MUX 的 REPL、日志或预览输出；断开或租约到期后释放所有权。Iris 构建为 USB 产品字符串增加 `ESP-Iris` 标记以供原生发现，沿用原 CDC 接口。普通构建不编译该适配，不引入 Iris 运行库、第二套 USB、应用更新器或远程进入 Vibe Mode 的命令。

后续安装新包前，请按住 AI 键并按 Reset，手动进入 Vibe Mode。应用拒绝 Iris 复位和更新写入命令。原生更新器释放设备后，ESP-VISION 客户端继续使用 EV-MUX；其 `firmware.status.v1` 能力提供应用 BIN 准确哈希等额外只读证据，详见帧格式、逻辑通道与 *RPC* 协议。

容器与发布

无签名 ZIP 使用 `esp-iris-system-update/0.2`，恰好包含 `manifest.json`、补齐擦除字节至 4096 字节的 `partition-table.bin` 和 `app.bin`，清单声明 `minimum_recovery_version=0.2.0`。包不包含 bootloader、OTA 元数据、Vibe Mode、NVS、coredump、FFAT 或 UI 资源镜像。打包器核对保留分区策略；除 IDF 生成的标准空白 OTA 元数据外，额外烧录输入均被拒绝。Secure Boot 签名和加密镜像需要独立策略，目前拒绝。

Iris 构建保留标准 ESP-IDF 应用描述符的工程名和版本。标签必须适配 31 字节版本字段，并与实际镜像完全一致。构建来源记录绑定编译配置类型、板型、源提交、流水线/任务 ID、实际 IDF 提交、配置/布局哈希和所有输入哈希，不包含个人路径或配置内容。打包仅消费当前流水线的准确产物，不重新编译或选择其他 IDF 目录。

格式基线为 esp-mosaico-utils 的 be151ae047014e79a9c7ae0092dd203844499003 提交。打包器和独立校验器使用标准库及哈希锁定的 YAML 解析依赖。`dist/irisfw/` 包含此板型的固件包、`.provenance.json` 和 `.validation.json` 伴随文件，以及 `SHA256SUMS` 和 `release.json`。

`publish_irisfw` 等待 `qualify_firmware_release` 成功，通过 `CI_JOB_TOKEN` 将冻结产物发布到本项目 Generic Package Registry，包名 `esp-vision-irisfw`、版本 `<tag>`。下载端点为 `${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/esp-vision-irisfw/<tag>/<filename>`，沿用 registry 现有访问权限。原 Launchpad BIN 地址及外部 DL 上传器不变。

发布前检查全部远端文件，相同字节复用，不同字节拒绝覆盖。逐文件下载比对成功后，最后写入 `release.json`。重试使用相同冻结产物，换流水线或重新生成来源记录须使用新版本；CI resource group 将发布串行化。校验和证明完整性，不代表安装授权。平台上传、审批及公开上架与 registry 发布和实机测试分别验收。

从同一流水线下载产物后打包：

```
python -m pip install --require-hashes -r tools/ci/irisfw/requirements.lock
python tools/ci/irisfw/package_release.py --tag "$CI_COMMIT_TAG" \
  --source-sha "$CI_COMMIT_SHA" --pipeline-id "$CI_PIPELINE_ID" \
  --build-root build --output dist/irisfw
python tools/ci/irisfw/validate_release.py --tag "$CI_COMMIT_TAG" \
  --source-sha "$CI_COMMIT_SHA" --pipeline-id "$CI_PIPELINE_ID" --root dist/irisfw
```

每个候选使用新的输出目录。API/web CI 同时设置 `ESP_VISION_IRISFW_ONLY=1`、`ESP_VISION_TEST_BOARD=ESP32_S31_MOSAICO` 和 `ESP_VISION_TEST_IDF_IMAGE=release-v6.1` 时，仅编译 Iris 固件，不操作硬件。离线回归命令为 `python -m unittest tools.tests.test_irisfw_release`。分支和 MR 不发布 registry 文件。

7.6 引入新的模型

ESP-VISION 支持两类模型运行路径：ESP-DL `.espdl` 模型使用*espdl –模型推理* 模块，TensorFlow Lite `.tflite` 模型使用*tflite –模型推理* 模块。模型不会编入固件，而是存放在板级存储中、运行时加载。本指南介绍如何添加并运行一个新模型。

7.6.1 1. 获取或转换模型

先选择模型运行时：

- ESP-DL：可从 ESP-DL 模型库 获取现成的 `.espdl`，或使用 ESP-DL 的量化/导出工具链将自有模型转换为 `.espdl` 格式，并与目标芯片（ESP32-P4、ESP32-S3 或 ESP32-S31）匹配。
- TFLite Micro：使用 TensorFlow Lite `.tflite` flatbuffer，并确保模型适配 TensorFlow Lite Micro 以及固件启用的算子集合。在当前板级资源下，量化 int8 模型通常是更实际的目标。

向仓库添加共享资源时，请保持 `models/` 下的目录结构，参照 `models/espdet/` 和 `models/tflite/`。

7.6.2 2. 将模型拷贝到板级存储

将 `.espdl` 或 `.tflite` 文件放到固件可读取的存储中，例如 `/sdcard` 或根目录 `/`：

- SD 卡：将文件拷贝到卡上，挂载点为 `/sdcard`。
- 片上 FAT（`ffat`）：数据分区通过 USB MSC 暴露，可将文件拖入大容量存储盘，挂载路径为根目录 `/`，例如模型路径可写作 `/my_model.espdl`。

7.6.3 3. 选择合适的 API

根据运行时和模型任务选择对应 API：

任务	API	结果
ESP-DL 目标检测（ESPDet）	*`espdl.ESPDet`*	`(x, y, w, h, score, category)`
ESP-DL 目标检测（YOLO11）	*`espdl.YOLO11`*	`(x, y, w, h, score, category)`
ESP-DL 姿态检测	*`espdl.YOLO11nPose`*	检测结果加 17 个 COCO 关键点
ESP-DL 图像分类	*`espdl.ImageNetCls`*	`(label, score)`
ESP-DL 输出由 Python 解码	*`espdl.Model`*	带 tensor 元数据的原始输出字节
通用 TFLite Micro 执行	*`tflite.Model`*	原始输出 tensor，或回调返回值

对于 ESP-DL 封装，若模型需要不同的预处理或过滤参数，可向构造函数传入 `mean`、`std`、`rgb_swap`、`score`、`nms`、`topk` 或 `softmax`。当需要保留 ESP-DL 推理、但在 Python 中解码输出时，应检查 `espdl.Model.inputs()` 和 `espdl.Model.outputs()`，再解码 `predict()` 返回的 `RawTensor` 字节。对于 TFLite Micro 模型，应检查 `input_shape`、`input_dtype`、`input_scale`、`input_zero_point`、`output_shape`、`output_dtype`、`output_scale` 和 `output_zero_point`，并在 Python 或辅助代码中实现模型相关的预处理或后处理。

7.6.4 4. 运行 ESP-DL 推理

```
import sensor, image, espdl

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)

det = espdl.ESPDet("/sdcard/my_model.espdl", score=0.5, nms=0.45)

while True:
    img = sensor.snapshot()
    for x, y, w, h, score, category in det.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
    img.flush()
```

7.6.5 5. 在 Python 中解码 ESP-DL 输出

当模型可使用 ESP-DL 图像预处理和推理，但输出 tensor 需要由 Python 侧解码时，可使用*espdl.Model*。

```
import sensor, espdl

model = espdl.Model("/sdcard/my_model.espdl", mean=(0, 0, 0), std=(255, 255, 255),␣
↪letterbox=True)
try:
    print("inputs:", model.inputs())
    print("outputs:", model.outputs())
    img = sensor.snapshot()
    outputs = model.predict(img)
    for name, tensor in outputs.items():
        _, shape, dtype, exponent, raw = tensor
        print(name, shape, dtype, exponent, len(raw))
finally:
    model.deinit()
```

每个输出 tensor 都会以原始字节形式返回，并附带 `shape`、`dtype` 和 ESP-DL `exponent` 元数据。应按 tensor 类型解码字节、应用 exponent 比例，再执行 sigmoid、框解码、NMS、分类 top-k 或坐标去 letterbox 等模型相关后处理。

7.6.6 6. 运行 TFLite Micro 推理

```
import tflite

def fill_input(buffer, shape, dtype_code):
    # 按模型要求填充量化后的输入字节。
    ...

model = tflite.Model("/sdcard/my_model.tflite")
try:
    print("input:", model.input_shape, model.input_dtype, model.input_scale, model.
↪input_zero_point)
    print("output:", model.output_shape, model.output_dtype, model.output_scale,␣
↪model.output_zero_point)
    outputs = model.predict([fill_input])
    raw = outputs[0]
finally:
    model.deinit()
```

ESP-DL 可运行脚本见 `example/03-Machine-Learning/00-ESP-DL/`` (``espdet_pico.py`、`espdet_pico_python.py`、`yolo11.py`、`yolo11n_pose.py`、`imagenet_cls.py`)，TFLite Micro 可运行脚本见 `example/03-Machine-Learning/01-TFLite/`` (``person_detection.py` 和 `sine.py`)。

7.6.7 7. 可选：性能分析和验证

在验证新的 `.espdl` 模型性能时，可使用*espdl.load_model()* 并设置 `profile=True`，以输出 ESP-DL 的性能分析信息。对于 TFLite Micro 模型，可打印 `model.len`、`model.ram`、输入元数据和输出元数据，先确认 flash 大小、arena 大小、tensor 布局和量化信息，再调试后处理。

7.7 训练 ESPDet Pico 模型

ESP-VISION 通过*espdl.ESPDet* 运行 ESPDet Pico 目标检测模型。模型训练、导出和量化在 ESP-Detection 项目中完成，ESP-VISION 侧只负责加载导出的 `.espdl` 文件，并通过 `sensor`、`image` 和 `espdl` 模块完成实时推理。

本文以单类别目标检测模型为例，说明从环境安装、数据集准备、训练、量化导出到 ESP-VISION 上运行的完整流程。

7.7.1 1. 准备 ESP-Detection 环境

训练仓库地址如下：

```
git clone -b feat/esp-vision-train https://github.com/YanKE01/esp-detection.git
cd esp-detection
```

ESP-Detection 使用 Python 训练环境，不需要先安装 ESP-IDF。只有编译 ESP-VISION 固件，或者运行 ESP-DL 生成的 C++ 示例工程时才需要 ESP-IDF。

ESP-Detection 使用 `uv` 管理依赖。根据操作系统选择对应安装方式。

Linux/macOS：

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows PowerShell：

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

安装完成后，重新打开终端，或根据安装脚本提示刷新当前终端的 `PATH`。确认 `uv` 可用后，同步项目环境：

```
uv sync
```

`uv sync` 会安装 Ultralytics、PyTorch、ONNX、ONNX Runtime、OpenCV 和 `esp-ppq`。如果机器有 NVIDIA GPU，训练和量化会明显更快；没有 GPU 时也可以走 CPU，但训练时间会长很多。

环境安装完成后，可以先确认命令入口可用：

```
uv run python espdet_train.py --help
uv run python -m deploy.quantize_aligned --help
```

7.7.2 2. 准备数据集

使用 Roboflow 准备数据集时，推荐导出 `YOLOv11` 格式。导出后可将数据集整理到 ESP-Detection 仓库，例如：

```
datasets/my_object/
  images/
    train/
    val/
  labels/
    train/
    val/
```

ESP-Detection 读取的是 YOLO detection 标签格式。每张图片在 `labels` 目录下有一个同名 `.txt` 标签文件。例如 `images/train/0001.jpg` 对应 `labels/train/0001.txt`。标签文件每一行表示一个目标框：

```
class_id x_center y_center width height
```

其中 `x_center`、`y_center`、`width` 和 `height` 都是相对图片宽高归一化后的浮点数，范围通常为 `0.0` 到 `1.0`。

例如单类别安全帽检测的数据可以写成：

```
0 0.5123 0.4381 0.1840 0.1365
```

数据集标签应和后续 ESP-VISION 端显示的类别名保持一致。

接着在 `cfg/datasets/` 下创建数据集配置，例如 `cfg/datasets/my_object.yaml`：

```yaml
path: datasets/my_object
train: images/train
val: images/val
test:

names:
  0: my_object
```

`path` 是数据集根目录，`train` 和 `val` 是相对 `path` 的图片目录。`names` 的顺序非常重要，它决定了板端 `espdl.ESPDet.detect()` 返回的 `category` 含义。训练完成后不要随意改变这个顺序。

如果是多类别模型，继续添加类别即可：

```yaml
names:
  0: helmet
  1: person
  2: vest
```

7.7.3 3. 训练 ESPDet Pico

ESP-Detection 提供了 `espdet_train.py` 作为只训练入口。它会使用 `cfg/models/espdet_pico.yaml` 构建 ESPDet Pico 网络，并输出最佳权重。

单类别模型训练示例：

```
uv run python espdet_train.py \
  --class_name my_object \
  --pretrained_path None \
  --dataset cfg/datasets/my_object.yaml \
  --size 320 320
```

关键参数说明：

参数	说明
`--class_name`	训练任务名称，会影响 Ultralytics 输出目录名称。
`--pretrained_path`	预训练权重路径。没有预训练权重时使用 `None`，脚本会从 `cfg/models/espdet_pico.yaml` 构建新模型。
`--dataset`	数据集 YAML 路径。
`--size`	输入尺寸，格式为 `height width`。例如 `320 320` 表示 320x320 输入。

训练脚本内部默认使用较长训练轮数、较大 batch、RAM cache 和数据增强策略。对于小数据集，训练可能需要观察验证集曲线，避免过拟合。训练完成后，终端会打印类似下面的信息：

```
Training complete: runs/detect/my_object
Best weights: runs/detect/my_object/weights/best.pt
```

后续量化时使用 `weights/best.pt`。

7.7.4 4. 准备校准数据

导出 `.espdl` 前需要做 int8 量化，量化需要校准图片。校准图片应尽量覆盖真实使用场景，例如不同光照、距离、角度、背景和目标大小。

最简单的做法是直接使用验证集图片：

```
datasets/my_object/images/val
```

也可以单独准备一个校准目录：

```
deploy/my_object_calib/
  0001.jpg
  0002.jpg
  0003.jpg
```

校准图片不需要标签，但分布要接近摄像头实际输入。如果校准集过于单一，可能出现模拟器指标正常、板端分数偏低或漏检的问题。

7.7.5 5. 量化并导出.espdl

使用 `deploy.quantize_aligned` 将训练好的 `best.pt` 导出为 ESP-DL 可加载的 `.espdl` 模型：

```
uv run python -m deploy.quantize_aligned \
  --model runs/detect/my_object/weights/best.pt \
  --size 320 320 \
  --target esp32p4 \
  --calib_data datasets/my_object/images/val \
  --espdl deploy/espdet_pico_320_320_my_object.espdl \
  --data cfg/datasets/my_object.yaml
```

参数说明：

参数	说明
`--model`	训练得到的 `best.pt`，也可以是已导出的 ONNX。
`--size`	模型输入尺寸，必须和训练尺寸一致。
`--target`	目标芯片，例如 `esp32p4` 或 `esp32s3`。应和最终运行模型的芯片匹配。
`--calib_data`	校准图片目录。
`--espdl`	导出的 ESP-DL 模型路径。
`--data`	数据集 YAML。传入后脚本会额外输出模拟器验证指标。

该量化脚本会在内部完成 pt 到 ONNX 的准备，然后使用 ESP-PPQ 导出 int8 ESP-DL 模型。当前配置使用 percentile 校准、bias correction 和面向 scale 一致性的 fusion 设置。脚本会打印导出模型的算子统计；如果传入 --data，还会打印模拟器 mAP50 和 mAP50-95。

量化完成后，确认 .espdl 文件存在：

```
ls -lh deploy/espdet_pico_320_320_my_object.espdl
```

7.7.6 6. 添加模型元数据

如果模型需要作为 ESP-VISION 仓库中的共享模型发布，建议放在 models/espdet/pico/<name>/ 下：

```
models/espdet/pico/my_object/
  README.md
  espdet_pico_320_320_my_object.espdl
  espdet_pico_320_320_my_object.json
```

其中 .espdl 是实际模型文件，.json 是模型列表和工具链读取的 sidecar 元数据。最小 JSON 示例：

```
{
    "name": "ESPDet Pico My Object",
    "architecture": "ESPDet Pico",
    "api": "espdl.ESPDet",
    "task": "detection",
    "input": "320x320",
    "inputFormat": "RGB565",
    "dataset": "My Object",
    "labels": ["my_object"],
    "sizeBytes": 574864,
    "description": "Detects my_object in camera images."
}
```

labels 必须和数据集 YAML 中 names 的顺序一致。sizeBytes 应填写实际模型文件大小，在 Linux 上可以这样获取：

```
stat -c '%s' models/espdet/pico/my_object/espdet_pico_320_320_my_object.espdl
```

README.md 建议说明模型用途、数据集来源、输入尺寸、量化指标和基本使用方法。对于公开数据集，数据集链接放在 README 中即可，不建议写进 JSON。

7.7.7 7. 拷贝模型到板端存储

ESP-VISION 运行时从文件系统加载模型。常用路径有两个：

路径	用途
/sdcard	SD 卡。适合模型较多、文件较大或频繁替换的场景。
/flash	片上 FAT 数据分区。通常通过 USB MSC 暴露为 U 盘，适合放少量常用模型。

例如将模型复制到 SD 卡根目录后，板端路径可以写成：

```
MODEL = "/sdcard/espdet_pico_320_320_my_object.espdl"
```

如果复制到 flash 根目录，则路径通常为：

```
MODEL = "/flash/espdet_pico_320_320_my_object.espdl"
```

7.7.8 8. 在 ESP-VISION 上运行

下面是一个完整的 MicroPython 推理脚本：

```
import espdl
import sensor
import time

MODEL = "/sdcard/espdet_pico_320_320_my_object.espdl"
LABELS = ("my_object",)

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

det = espdl.ESPDet(MODEL, score=0.5, nms=0.7)

try:
    while True:
        img = sensor.snapshot()
        for x, y, w, h, score, category in det.detect(img):
            label = LABELS[category] if category < len(LABELS) else str(category)
            img.draw_rectangle(x, y, w, h, color=(255, 0, 0), thickness=2)
            img.draw_string(x, max(0, y - 12), "%s %.2f" % (label, score),␣
↪color=(255, 0, 0))
        img.flush()
        time.sleep_ms(20)
finally:
    det.deinit()
```

`sensor` 输出应使用 `RGB565`。`score` 是置信度阈值，`nms` 是 NMS 阈值。第一次上板时建议先用较低 `score` 观察模型是否能稳定出框，再根据误检和漏检情况调整。

7.7.9 9. 验证和调参

建议按以下顺序验证：

- 先看训练日志和验证集指标，确认浮点模型没有明显欠拟合或过拟合。
- 再看量化脚本输出的模拟器指标，确认 int8 模型没有大幅掉点。
- 然后在 ESP-VISION 上运行真实摄像头脚本，观察不同距离、角度和光照下的检测框。
- 最后调节 `score` 和 `nms`，并固定最终模型、标签和示例脚本。

如果模拟器指标正常但板端效果差，优先检查这些问题：

- `--size` 是否和训练尺寸完全一致。
- `labels` 是否和数据集 YAML 中 `names` 顺序一致。
- 校准图片是否覆盖真实摄像头输入分布。
- 板端摄像头画面是否曝光异常、颜色异常或目标过小。
- `score` 是否设得过高，导致低分检测结果被过滤。

模型文件、JSON 元数据、README 和测试脚本都确认后，就可以将模型加入 ESP-VISION 的 `models/` 和 `example/` 目录，作为可复用模型发布。

7.8 添加新的开发板

一块开发板的板级包集中在单棵目录树 `boards/<BOARD>/` 中：顶层为 ESP-VISION 文件，`boards/<BOARD>/port/` 子目录为 MicroPython ESP32 移植文件。构建时会将 `boards/<BOARD>/port/` 投射到生成的 MicroPython 副本的 `ports/esp32/boards/<BOARD>/`。建议从 `TEMPLATE` 板开始。

7.8.1 MicroPython 移植侧

位于 `boards/<BOARD>/port/`：

文件	用途
`mpconfigboard.cmake`	`IDF_TARGET` 配置值与 `SDKCONFIG_DEFAULTS` 链。
`mpconfigboard.h`	MicroPython 功能开关与 USB 字符串。
`sdkconfig.board`	板级 ESP-IDF Kconfig 覆盖项。
`partitions-*.csv`	分区表。
`board.json`、`board.md`	上游板卡清单元数据。

7.8.2 ESP-VISION 侧

位于 `boards/<BOARD>/`：

文件	用途
`boardconfig.h`	引脚分配与板级运行时常量。
`imlib_config.h`	OpenMV `imlib` 功能开关。
`manifest.py`	冻结的 Python 模块。
`camera.c`	板级相机后端。
`display.c`	LCD 面板与背光实现。
`sdcard.c`	SD 卡供电与插卡检测实现。

当板目录下存在 `camera.c`、`display.c`、`sdcard.c` 时，`micropython.cmake` 会自动选用它们，并包含板卡可选的 `board.cmake`。

7.8.3 构建与烧录

```
idf.py --board <NEW_BOARD> -p /dev/ttyACM0 build flash monitor
```

备注： 本页为起步提纲。详细的适配步骤（传感器选择、PPA 配置、显示时序）将在后续补全。

7.9 EV-MUX v4 主机适配入口

EV-MUX 的完整说明已迁移至*通信系统与 EV-MUX*。本页保留旧链接入口；通信架构、接口、会话、协议与验收均以新章节为准。

7.10 Agent 代码生成指南

本指南是 AI Agent 生成 ESP-VISION 应用代码时的强制性约束。必须将目标环境视为在 ESP32 微控制器上运行的 ESP-VISION MicroPython v1.28.0，而不是运行 CPython 的 PC。只能使用本知识包中存在、已为所选开发板启用，或已由仓库示例验证的 API；不得假定设备上存在桌面端标准库模块、`pip` 包、原生 wheel、操作系统服务、子进程或 PC 文件系统。

7.10.1 依赖规则

默认 ESP-VISION 固件不能从 `requirements.txt` 动态安装依赖，也没有包含设备端 `mip`。生成的目标端代码不得调用 `pip`、解析 `requirements.txt`、调用 `mip.install()`，也不得静默依赖未验证的第三方包。如果知识包中没有必需模块，应改用已支持的模块重写，或明确说明运行前必须通过主机工具复制到 `/lib`、预编译为兼容的 `.mpy`、冻结到定制固件，或实现为原生模块。

主机端工具与目标端代码属于两个不同环境。PC 工具可以使用 `pyserial` 等 CPython 包，但必须将这类代码标记为“仅主机端”，且不能在 ESP-VISION 脚本中导入这些包。

7.10.2 All-in-One 输出

除非用户明确要求软件包或多文件架构，否则应返回一个可独立运行的设备端 `main.py`。将已支持的 import、常量、初始化、应用逻辑、主循环或有界测试、异常处理和资源释放都放在该文件中，避免要求用户再手工创建辅助模块。模型和其他二进制资产可以保持为独立文件，但必须列出它们的精确设备路径和部署前置条件。

在进入无限产品循环前，优先生成有界且可观测的验证代码。只有在摄像头、模型、存储、网络或用户要求的其他操作真正成功后，才打印唯一成功标记。下面的目标端冒烟测试是 all-in-one MicroPython 脚本，只有在成功拍摄一帧后才打印 `EVTEST:OK`：

```
import gc
import sensor


def main():
    print("EVTEST:START")
    sensor.reset()
    try:
        sensor.set_pixformat(sensor.RGB565)
        sensor.set_framesize(sensor.QQVGA)
        sensor.skip_frames(time=500)
        image = sensor.snapshot()
        width = image.width()
        height = image.height()
    finally:
        sensor.shutdown()
    print("EVTEST:OK {}x{} heap={}".format(
        width, height, gc.mem_free()))


main()
```

7.10.3 设备通信与闭环验收

Agent 与 IDE、CLI 同属主机应用，设备交互使用 Device Core，不再维护另一套 EV-MUX 客户端；统一的主机/Core/协议/固件关系见*方案架构*。生成代码后的设备接管、上传、执行与结果核验统一遵循*通信系统与 EV-MUX*。完整主机端摄像头闭环示例已移至*通信验收、闭环测试与故障定位*；不得用握手、上传 ACK、命令回显或 REPL 提示符替代应用成功证据。

Chapter 8

方案架构

ESP-VISION 将主机工具、在线知识与设备固件分开：IDE 和自定义 SDK 应用直接使用 Device Core；AI 客户端与独立 Agent 使用插件的 Skill，通过内置 Device CLI 在本地操作设备，并通过 HTTPS 查询官方在线 Knowledge MCP。Device CLI 调用 Device Core，由 SDK 通过开发板支持的 USB CDC、USB Serial/JTAG、UART 或 TCP 实现 EV-MUX v4 主机端。设备固件包含 MicroPython、视觉/推理模块、平台服务和板级后端；固件代码进一步按是否触及 MicroPython（`mp_obj_t` / `py/*.h`）分层。

8.1 分层概览

- **主机应用：** IDE 提供编辑/调试界面，自定义应用使用 SDK，AI 客户端和独立 Agent 根据任务选择工具；Agent 自己拥有 AI loop。
- **Plugin**（`host/esp-vision-plugin`，`@esp-vision/plugin`）：为 Codex、Claude Code、WorkBuddy 打包一个能力导向的 Skill、固定版本 Device CLI 和官方 Knowledge HTTP 端点配置，并提供独立的 WorkBuddy Expert、QoderWork 与 Cursor 导出适配。宿主加载和市场上架须分别完成平台验收。
- **Device CLI**（`host/device-cli`，`@esp-vision/device-cli`）：通过 shell 命令、JSON 结果、流式事件及二进制输出文件提供设备能力；插件内置副本和独立命令都使用 Device Core 公共 API。
- **Knowledge MCP**（`host/knowledge-mcp`，`@esp-vision/knowledge-mcp`）：基于版本化快照提供 API、文档和示例查询。官方插件使用在线 HTTP 服务；npm 包支持 stdio、HTTP 自行部署。主仓构建快照、npm 包和不可变镜像，website 负责部署该镜像；HTTP 记录匿名每日汇总统计。
- **Device Core**（`host/device-core`）：独立版本的 `@esp-vision/device-core` npm 包负责协议解析、请求关联、会话、保活与操作生命周期。图中只选有代表性的 API 名称，完整契约见*应用 API 完整参考*。
- **EV-MUX v4：** Core 与固件之间的运行时线协议，在 USB CDC、USJ、UART、TCP 上统一帧格式、逻辑通道、所有权与 VM epoch。主机适配器和网络桥只传字节，不另建会话 owner。
- **设备应用：** MicroPython 脚本运行于开发板并调用 Python 绑定；主机断开后，设备应用仍可独立运行。
- **绑定层**（`modules/`）：即 `USER_C_MODULES` 层。主要模块 `image`、`sensor`、`display`、`espdl` 和 `tflite` 通过 `MP_REGISTER_MODULE` 自注册。`py_imageio.c` 提供 `image.ImageIO` 类型，`py_helper.c` 为共享辅助代码。绑定层只做对象转换与轻量 API 适配，重逻辑放在纯 C 或 `platform/` 中。
- **平台服务**（`platform/`）：自研的 ESP32 胶水层。`ev_channel.c` / `ev_mux.c` / `ev_control_transport.c` / `ev_stdio.c`（EV-MUX / EV-ATP 底层传输）、`preview.c`（EV-MUX JPEG 预览）、`display.c`（通用显示层）、`sdcard.c`（挂载到 `/sdcard`）、`usb_msc.c`（通过 TinyUSB MSC 暴露 `ffat` 分区）、`usb_auto_download.c`（S3/P4 OTG CDC 复位/下载协议）、`jpeg.c`（硬件或软件 JPEG）、`debug.c`，以及 `main.c`（启动初始化与软复位循环）。
- **imlib 组件**（`components/imlib/`）：纯 C 视觉算法，作为以 MIT 维护的 IDF 组件，源自 OpenMV `lib/imlib`。
- **板级后端**（`boards/<BOARD>/`）：各板配置及真实的相机/显示/SD 卡实现。P4X 与 S31 使用 `esp_video`/V4L2，P4X 还使用 PPA；S3 使用 `esp32-camera`。

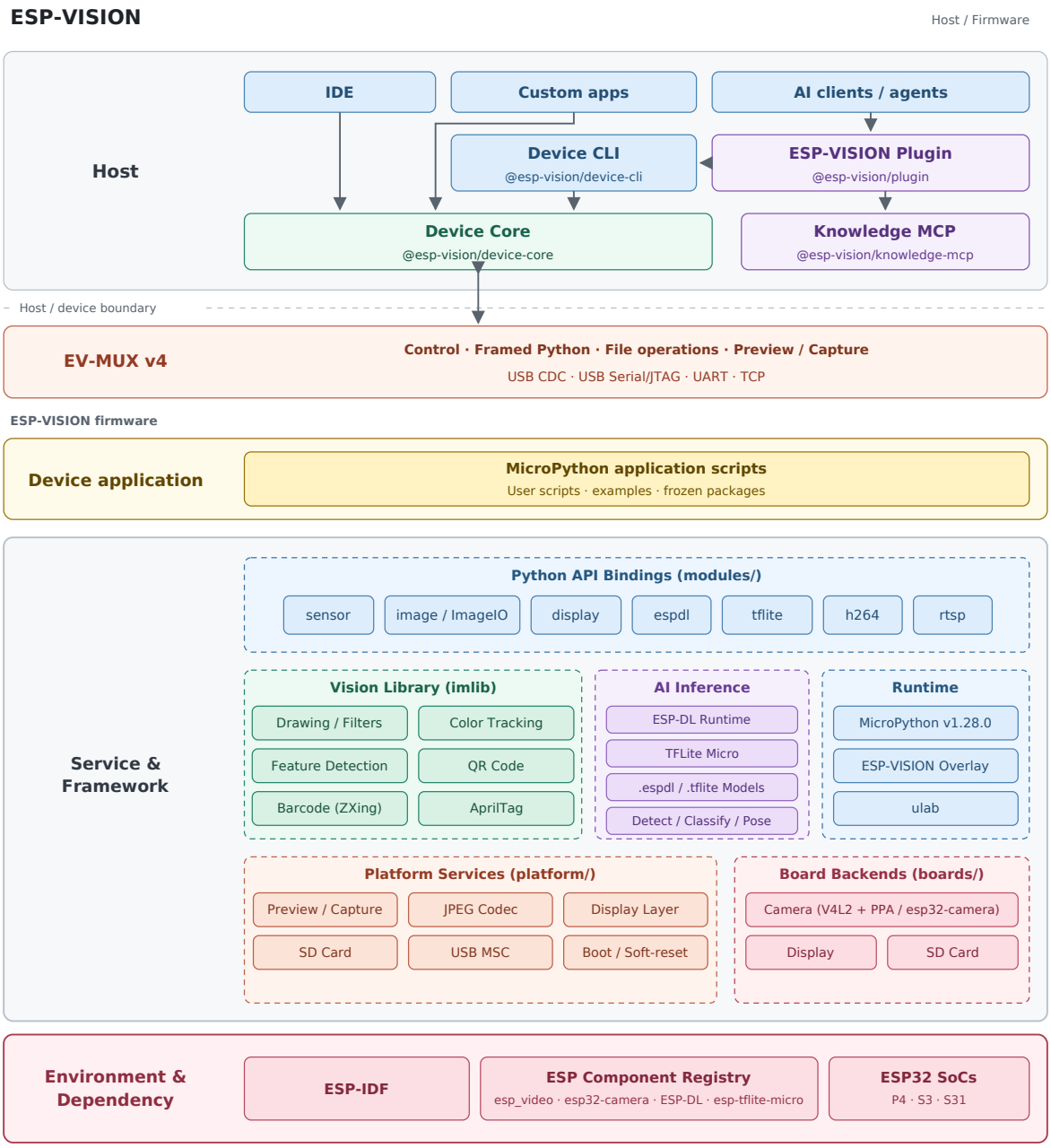
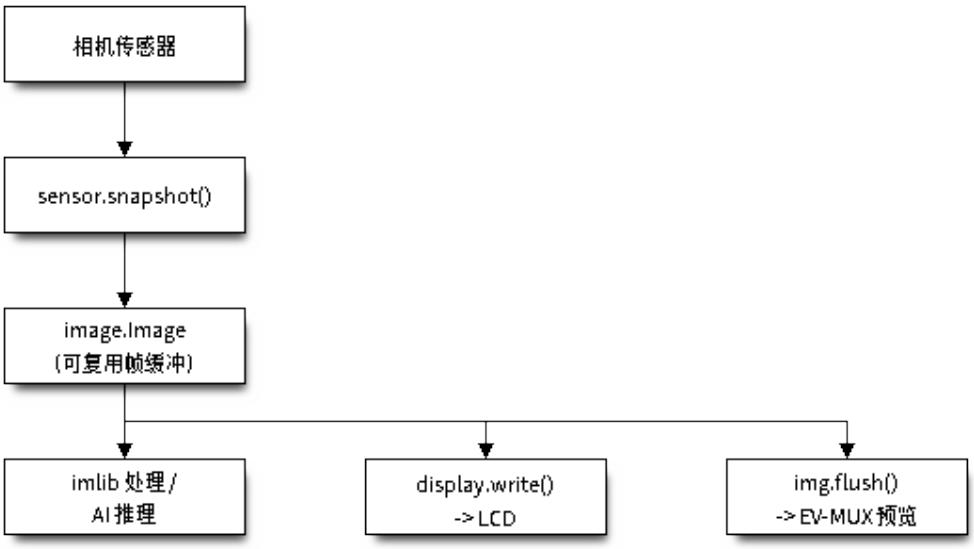


图 1: Device CLI / Device Core 承接本地设备控制，在线 Knowledge MCP 独立提供知识服务

- **MicroPython + overlay**：以 MicroPython v1.28.0 为固定基线；项目改动位于 `overlay/micropython/`，并应用到 `build/micropython/` 下的生成副本。

宿主提供的字节传输适配器承接 SDK 协议之下的物理 I/O。局域网 DNS-SD 发现属于连接前的枚举路径；ROM 固件烧写使用独立 loader 生命周期，不通过 EV-MUX 传输应用镜像。固件与 SDK 版本保持独立；主机接入及 API 文档见*Device Core 主机 SDK*。

8.2 采集到输出的数据流



`sensor.snapshot()` 将一帧采集到可复用的帧缓冲中，并封装为 `image.Image`。脚本随后对图像进行 `imlib` 处理、ESP-DL 推理或 TFLite Micro 推理，再通过 `display.write()` 送往 LCD，或通过 `img.flush()` 送往主机预览。

8.3 通信子系统

主机通信采用 EV-MUX v4，会话同时持有 user/debug 两个 stream，逻辑通道复用在当前 owner 的 USJ、USB、UART 或 TCP link 上。MicroPython HAL、C stdio、IDF 日志和 JPEG preview 分别接入对应通道。

完整的输出层级、物理接口、会话接管、帧格式、文件/Python/图像协议和验收说明已独立为*通信系统与 EV-MUX*，避免在工程架构、Agent 指南和主机适配页重复维护。当前 MicroPython 异常与 sys.stderr 的具体输出路径见架构、执行上下文与标准输出。

8.4 源码结构

路径	职责
host/device-core/	公共主机 SDK 和传输适配器（@esp-vision/device-core）。
host/device-cli/	独立设备命令工具包（@esp-vision/device-cli）。
host/knowledge-mcp/	Knowledge MCP 运行时、知识快照和服务镜像（@esp-vision/knowledge-mcp）。
host/esp-vision-plugin/	Skill、内置 CLI、Knowledge 端点配置及安装器（@esp-vision/plugin）。
idf_ext.py	仓库根目录下板级感知的 idf.py 扩展。
micropython.cmake	集成枢纽：注册用户模块、平台与板级源文件、include 路径、条件性 zxing 与 ulab。
lib/	固定版本的第三方子模块（MicroPython、ulab、ZXing-C++）。
overlay/micropython/	采用 MicroPython 路径布局的 ESP-VISION MicroPython 增量。
boards/	各板配置、冻结清单与板级外设后端。
platform/	共享运行时服务（相机、预览、存储、显示、USB、JPEG）。
modules/	MicroPython C/C++ 绑定（sensor、image、display、imageio、espdl、tflite，以及随芯片启用的 h264 和 rtsp）。
components/	ESP-IDF 组件，包括 OpenMV imlib 与 ZXing 后端。
models/	运行时从板级存储加载的可选模型资源。
example/	MicroPython 示例脚本。
stubs/	描述 C 模块的 .pyi 类型存根。

8.5 板卡的组成

一块开发板的定义集中在单棵目录树 boards/<BOARD>/ 中：

- ESP-VISION 侧（顶层）：boardconfig.h、imlib_config.h、manifest.py，以及可选的 camera.c / display.c / sdcard.c。
- MicroPython 移植侧（boards/<BOARD>/port/）：IDF 目标、sdkconfig、分区表、USB 字符串。构建时会将该子目录投射到生成的 MicroPython 副本的 ports/esp32/boards/<BOARD>/。

完整步骤请参阅添加新的开发板。

8.6 随芯片变化的源码

micropython.cmake 根据 IDF_TARGET 和板级配置选择模块。ESP32-P4 构建包含 h264 与 rtsp；当前 P4 板级配置还会启用 ZXing-C++ 条形码后端。最终公开 API 矩阵见芯片与开发板支持。

8.7 MicroPython Overlay

ESP-VISION 以 MicroPython v1.28.0 作为固定上游基线。ESP32 port 的项目增量维护在 overlay/micropython/ 下。prepare-micropython 构建步骤会将其应用到 build/micropython/idf<ESP_IDF_VERSION>/micropython/ 下的生成副本；lib/micropython 子模块保持为干净的上游参考。

ESP-VISION 与上游项目的关系见项目关系；各组件的许可证见许可证。

Device Core 的同一 firmware API 提供两种显式 ROM 操作：默认匹配现有分区表、只更新应用；首次安装则使用包含 bootloader、分区表与应用的完整镜像清单。两者均不通过 EV-MUX 传输固件；IDE、Agent、CLI 复用同一校验与回读逻辑，无需各自实现烧写流程。

Chapter 9

通信系统与 EV-MUX

本章介绍 ESP-VISION 的整个主机通信系统：从 MicroPython 的输入输出、ESP-IDF 日志和摄像头数据，到物理接口、会话所有权、EV-MUX 帧，再经共享 Device Core SDK 到达 IDE、Agent、CLI 或自定义主机应用。协议说明以当前仓库的 EV-MUX v4 实现为准；固件发布版本、EV-MUX 会话版本和线上帧格式版本是三个不同概念。

重要： `evMuxVersion:4` 是会话与授权契约；`EVMUX/1` 是字节帧格式。v4 固件继续使用 EVMUX/1，不应把帧头改成 EVMUX/4。IDE 1.3.0 的 Device Core 只接受明确支持 v4 的固件。

9.1 一张图看懂整条链路

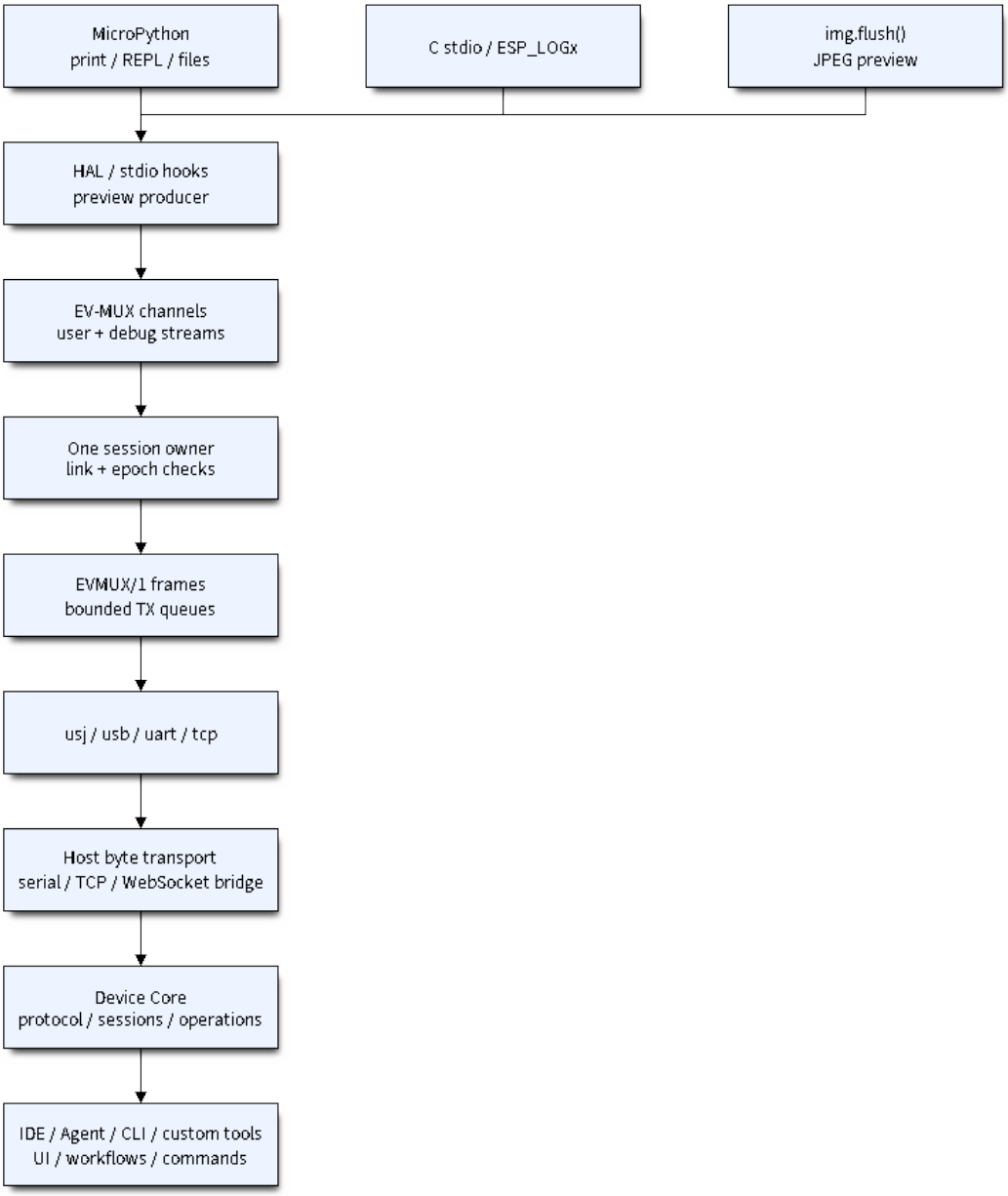


图 1：设备输出经过语义分流、会话约束和帧封装，再进入同一条物理连接。主机命令沿相反方向到达 parser、授权检查和 REPL/控制处理器。主机必须持续收包，因为请求响应、文本输出和图像可以交错出现。

9.2 推荐阅读顺序

9.2.1 架构、执行上下文与标准输出

系统架构与职责边界

IDE、Agent、CLI 和自定义工具是平级主机应用，共用同一个 Device Core 包。Core 实现主机 API 与协议/会话状态；EV-MUX 是 USB CDC、USJ、UART、TCP 上的运行时线协议；固件实现设备操作和 MicroPython 运行时。下图中的代表性 API 仅作总览，不是完整清单。

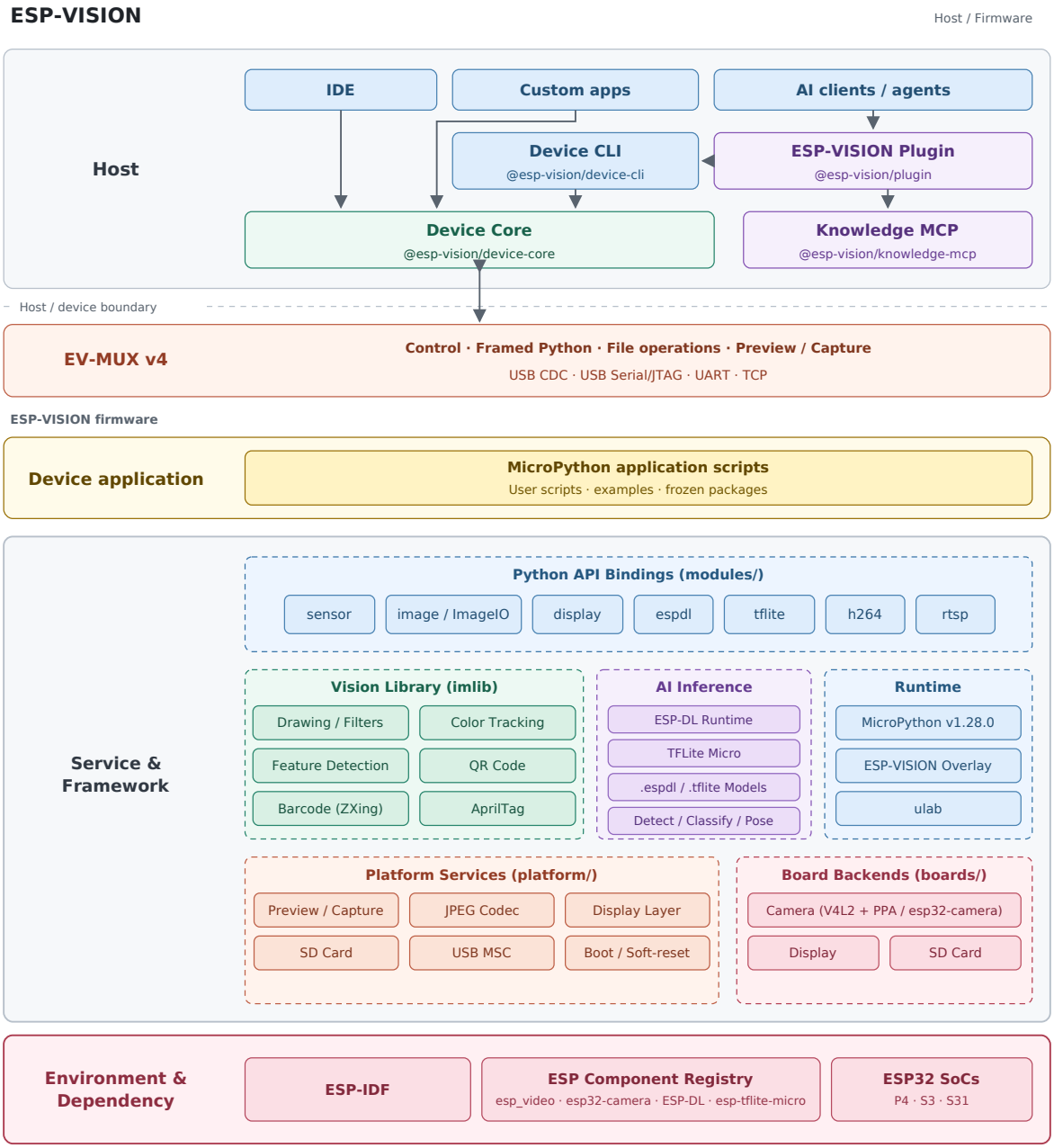


图 1: 统一系统架构图；下文展开固件执行路径。

本 SVG 与 README、系统架构及 SDK 指南共用。发现和 ROM 烧写独立于 EV-MUX 运行时路径，具体边界见方案架构；设备脚本在开发板执行，不要求主机持续连接。

EV-MUX 在哪一层

EV-MUX 是运行期字节封装和多路复用层。它不执行 Python，也不把原有 MicroPython VM 替换成“EV-MUX VM”。MicroPython 保留词法分析、编译、解释执行、友好 REPL、stdin/stdout 和异常打印；

ESP-VISION 在 HAL 和平台服务边界接入帧协议与会话路由。

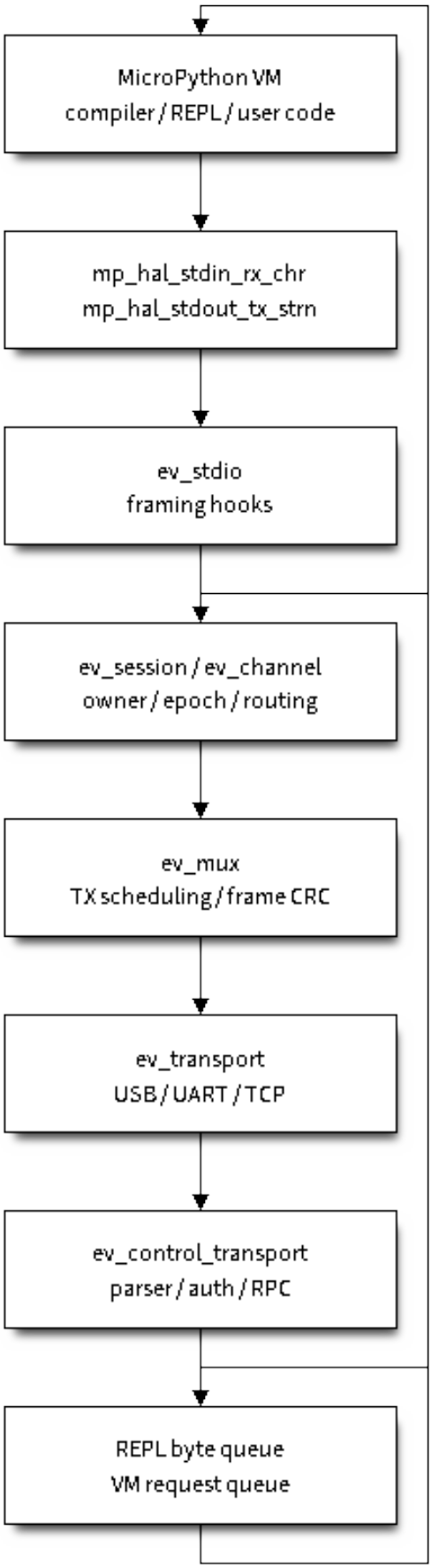

MicroPython VM
compiler / REPL / user code
mp_hal_stdin_rx_chr
mp_hal_stdout_tx_strn
ev_stdio
framing hooks
ev_session / ev_channel
owner / epoch / routing
ev_mux
TX scheduling / frame CRC
ev_transport
USB / UART / TCP
ev_control_transport
parser / auth / RPC
REPL byte queue
VM request queue

图 2：解释器层、HAL 适配层、控制面、会话路由与物理后端的关系。上行 stdout 和下行 stdin 没有绕开 VM；纯传输控制 RPC 则可以不等待用户 Python 返回。

输出来源决定通道，而不是文本内容

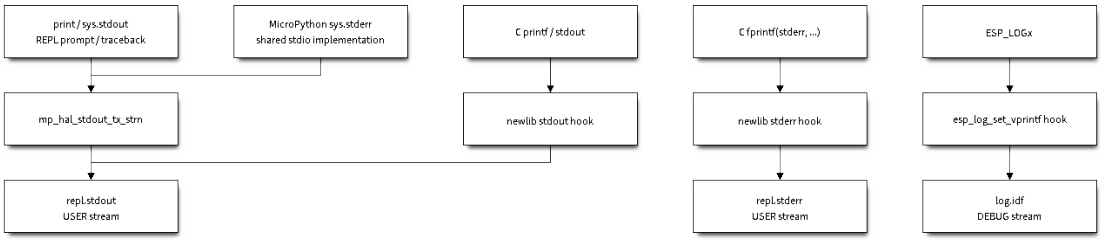


图 3：当前构建中各类打印的真实去向。 Python 的 `print()` 与 C 的 `printf()` 可能进入相同 channel，但调用链不同；日志文字中包含“error”并不会把它自动变成 `repl.stderr`。

表 1: 输出路径核对表

来源	固件入口	线上结果
Python `print()`、`sys.stdout.write()`、友好 REPL 提示符	MicroPython print/stdio → `mp_hal_stdout_tx_strn()` → `ev_stdio_write_repl_stdout()`	`repl.stdout`, `encoding:text`。
REPL 中的未处理异常、`KeyboardInterrupt` traceback	`pyexec.c` 使用 `mp_obj_print_exception(&mp_plat ...)`，仍进入 HAL stdout。	当前可能出现在 `repl.stdout`，不能仅观察 stderr 判定失败。
当前 MicroPython `sys.stderr.write()`	`shared/runtime/sys_stdio_mphal.c` 的 stdout/stderr 对象共享 `stdio_write()`；两者调用 cooked HAL stdout。	当前仍进入 `repl.stdout`，并非桌面 CPython 的独立 stderr 管道。
C `printf()` / `fprintf(stdout, ...)`	`CONFIG_LIBC_NEWLIB` 构建安装 newlib `stdout->_write` hook。	`repl.stdout`，metadata 的 `method:c-stdio` 表示 C stdio 来源。
C `fprintf(stderr, ...)`	newlib `stderr->_write` hook。	`repl.stderr`, `method:c-stdio`。
ESP-IDF `ESP_LOGI/W/E` 等正常日志	`esp_log_set_vprintf()` → `ev_stdio_log_vprintf()`。	`log.idf`, `method:esp-idf`；走 debug stream。
`esp_vision_debug_` 等平台日志	平台日志路径调用 EV-MUX IDF log 写入函数。	`log.idf`；具体 `method` 取决于调用入口。
`img.flush()`	preview producer → JPEG 编码 → lossy EV-MUX TX。	`preview.frame`，二进制 JPEG；不经过 printf。

`encoding:text` 不保证每个帧都在一个 UTF-8 字符或一行结束处切分。主机需要保留增量 UTF-8 解码器和未完成行缓冲；不能逐帧独立解码并把替换字符当作原始文本。

备注：上表描述当前源码，不能由通道名字反推所有语言层的 stderr 行为。如果将来为 Python stderr 增加专门 HAL 路径，应同时更新本表、客户端判断和协议测试。

IDF system console 与 MicroPython console

IDF system console 的 Kconfig 选择底层系统输出目的地。MicroPython 的 port 配置决定其 UART REPL、USJ、TinyUSB CDC 是否编译和初始化。EV-MUX 的会话路由再决定运行期业务帧发往哪一个已启用 link。这是三个配置层，不是一组互为别名的开关。

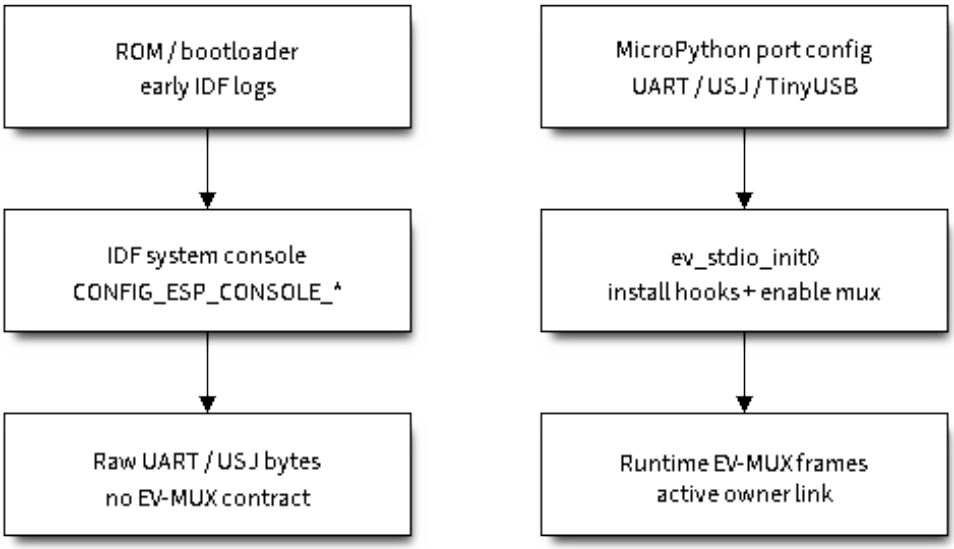


图 4：初始化前的 system console 与运行态 EV-MUX 输出不是同一契约。设备上电、进入 ROM 或发生绕过 hook 的 panic 时，主机可能读到没有帧头的文字；活动业务仍必须靠完整帧确认。

`MICROPY_HW_ESP_USB_SERIAL_JTAG` 决定 ESP-VISION/MicroPython 的 USJ 路径。即使它为 0，某些 IDF target/version 的默认 system console 仍可能使用 USJ。反过来，启用 MicroPython USJ 时，端口初始化会显式打开外设时钟和 internal PHY，不依赖 console 顺带完成初始化。

`sensor.evmux(False)` 是运行期调试开关，可以恢复非 EV-MUX 的文本路径。当前源码的 `ESP_VISION_EV_MUX_DEFAULT_ENABLED` 默认值在 `ev_stdio.c` 内定义，但该文件没有接入 `boardconfig.h` 的这项设置；只在板级文件定义同名宏不会关闭上电默认启用。主机不应把关闭 framing 当作 v4 的降级协商机制。

哪个任务负责哪类工作

表 2: 执行上下文

上下文	主要工作	与用户 Python 的关系
transport task	读取各 link、拼帧、检查授权，处理发现、保活、释放、网络事务和纯 C transport stats。	用户脚本执行时仍能处理这些控制，不取得另一套 MicroPython VM。
VM task	友好 REPL、Python 用户代码、触及 VFS/对象/相机的 VM RPC。	长脚本可能占用该任务；队列有界，需先停止脚本再做依赖 VM 空闲的操作。
TX task	发送完整帧，处理短写、拥塞、队列优先级与 link 失效。	生产者提交的数据被复制或转移到受控生命周期，不能直接悬挂 VM 临时指针。
network worker / TCP 管理	Wi-Fi 初始化、事务性配网、连接/监听维护、TCP 槽位回收。	与 Python WLAN 共享底层服务；不会因普通 VM 软复位自动重复初始化无线驱动。
USB CDC 回调 / restart timer	回调记录 DTR/RTS 状态，延迟处理下载/复位。	不在 USB control callback 内立即拆除驱动；与 EV-MUX 接管是不同控制面。

`file.write` 在 transport task 中直接写原生 VFS，检查会话/VM 代次并持有存储交接计数，不进入解释器队列。旧版 `script.write`、`script.run`、`debug.capture_frame`、`debug.capture_screen` 和涉及 Python 状态的 `debug.info` 进入 VM 请求队列。`debug.info` 的 `scope:transport.stats` 是例外：它在 VM 忙时也可查询。队列数组有两个槽位，以环形队列空槽规则实际只容纳一个待执行请求；过期预算为 5000 ms，忙或过期会报告 `VM_BUSY` / `VM_TIMEOUT`。

LCD 抓屏在同一 VM 任务中读取显示层最后提交的 RGB565 帧并编码 JPEG，不调用 `sensor` 或摄像头，不改变显示内容。它不是 preview 订阅，也不是物理面板回读；与 display/软复位清理串行执行，避免读取已释放的帧缓冲。

`repl.signal` 中的 Ctrl-C 在解释器已启用中断时调度 `KeyboardInterrupt`，否则作为编辑控制字符进入 REPL 输入队列。它不是远程抢占 FreeRTOS 任务，也不能保证及时中断不返回 VM 的原生函数。

软复位与后台通信的生命周期

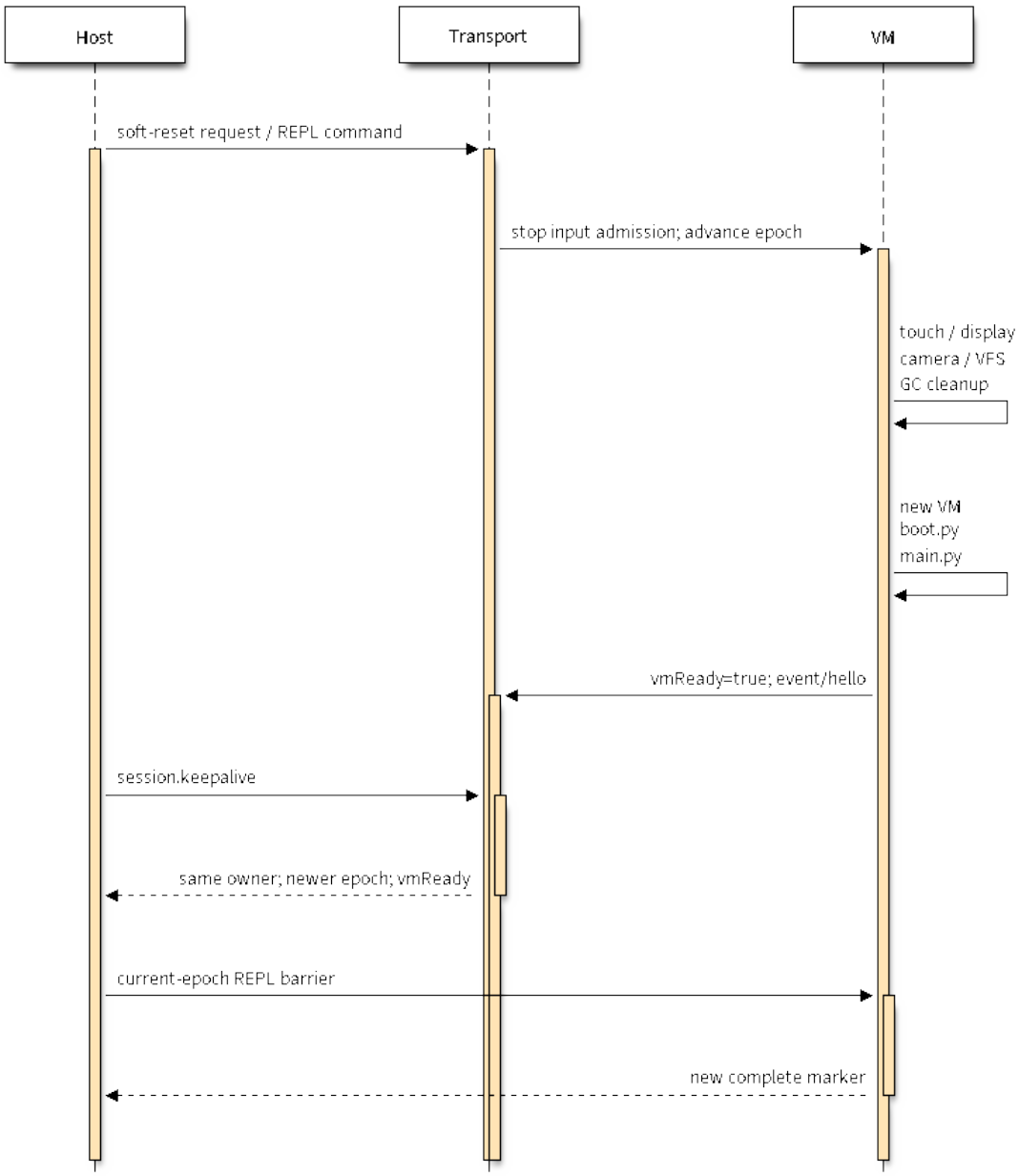


图 5：软复位推进 VM 代次，但可以保留会话、网络和可保留的 link。先停止旧 VM 输入，再清理 LCD touch、display、preview、camera 等对象。`vmEpoch` 变化发生在复位开始，不是“VM 已可执行”的证据；须继续等待 `vmReady` 与真实 REPL 标记。

队列、日志失败与不会发生的回退

正常队列每端点 16 KiB / 8 帧，两个 TCP slot 共享一个队列预算；另为控制流量预留 4 KiB / 2 帧。短 REPL 写入有 512 字节 staging 与约 5 ms 合并窗口，避免每个字符生成一个完整帧。preview 只保留最新待发图像，详见预览、调试抓帧、背压与统计。

EV-MUX 已启用时，帧提交/发送失败不会把同一条 Python 或 C 输出再次广播到原始串口。stdio hook 有递归保护；在拥塞错误路径中再次打印日志可能造成递归，固件改用计数器观测丢弃。当前 IDF vprintf 临时缓冲为 512 字节，格式化结果放不下时记录 drop 并返回错误，不把超长日志伪装为完整成功输出。

初始化前与关闭 mux 后的既有 console 路径、低层 panic/ROM 输出是另外的生命周期边界，不应与“发送失败后回退”混为一谈。

阅读源码的入口

`platform/ev_stdio.c` 与 `overlay/micropython/ports/esp32/mphalport.c` 决定输入输出接入；`platform/ev_control_transport.c` 决定入站控制分流；`platform/ev_session.c` / `ev_channel.c` 决定所有权与路由；`platform/ev_mux.c` 决定输出封装、快照、队列和发送；`platform/ev_vm_output.c` 管理 VM 输出范围；`platform/main.c` 管理启动和复位。MicroPython stderr 的关键上游文件是 `lib/micropython/shared/runtime/sys_stdio_mphal.c`。

9.2.2 物理接口、连接代次与构建配置

警告： TCP 控制接口用于隔离、可信本地网络中的无线开发和调试，不提供对端身份认证或传输加密。能够访问该接口的任意客户端均可取得空闲会话并执行文件上传、脚本运行等开发操作；会话所有权及 CRC 校验不提供操作者身份认证。启用 TCP 传输的开发板在已启用的后台网络服务上线后会自动开放监听，绑定 STA IPv4 地址并不限制客户端必须位于同一子网。请勿将该接口暴露到互联网或不可信网络。不需要该接口时，请在所选开发板的 `boardconfig.h` 中将 `ESP_VISION_ENABLE_TCP_TRANSPORT` 设为 `0` 并重新构建。使用定位、赏金范围及私下报告渠道详见仓库的 安全政策。

sink 类型与实际 link

`capabilities.sinks` 表示固件编译支持的后端及当前状态。sink 名称是 `usj`、`usb`、`uart`、`tcp`，不是操作系统的 `/dev/ttyACM0`，也不是 channel。`usb_otg_cdc` 表示硬件 kind；对外端点 ID 仍是 `usb`，不能用 `cdc` 或 `usb_otg_cdc` 替代。

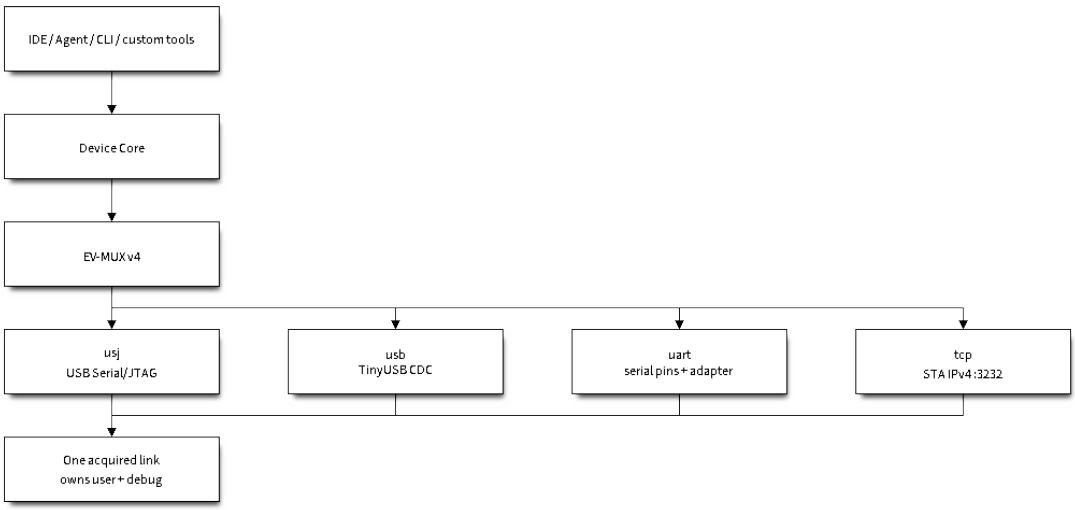


图 6：多个物理入口都可以参与发现和接管，设备只有一个控制 owner。这些入口并不是各有独立 VM 的设备副本。

主机/Core/协议关系沿用方案架构。本图只展开 EV-MUX 之下的物理运行时链路；发现与 ROM loader 各有独立生命周期。串口/socket 适配器不再次接管会话。

表 3: 物理接口比较

sink	物理形态	连接/就绪判断	常见用途与限制
usj	芯片 USB-Serial/JTAG 外设，通常在主机呈现为串口。	无可靠拔线检测时 present 为未知；静默断线依赖租约恢复。	P4-EYE 默认/回退口；还可能出现 ROM、IDF 原始启动输出。它与 USB-OTG CDC 不是同一个驱动。
usb	USB-OTG 使用现有 TinyUSB CDC；可与 MSC 在同一复合设备中出现。	USB 配置完成表示 present/ready；真实 USB 断开使 link generation 失效。	适合交互、文件和图像；业务收发不要求 DTR。CDC 和 MSC 的功能与写入权限不同。
uart	真实 UART TX/RX/GND 与 USB-UART 转接器。	由板级 UART REPL 启用情况和实际初始化状态决定；通常不能可靠获知对端消失。	可承载相同 channel，但受实际波特率、引脚/串口占用与有界发送期限限制。
tcp	已配网设备的 STA IPv4 TCP server。	accept 分配槽位，FIN/RST、socket 错误和超时回收 link。	默认 3232；三个 accepted slot，最多两个只读旁观者和一个控制者。浏览器需要 WebSocket-to-TCP 字节桥接。

板级后端启用条件

USB 功能组合、配置、卷标和固定 MSC LUN 见*USB 接口、配置与磁盘命名*。

后端由 boards/<BOARD>/boardconfig.h、boards/<BOARD>/port/mpconfigboard.h 和 SoC 能力共同决定。关闭 MicroPython 接口宏会关闭对应接口，而非仅禁止 EV-MUX 使用它；当前没有 MUX 专用串口白名单。下表描述当前源码默认配置，运行时以 capabilities.sinks 为准。

表 4: 编译条件与当前板级示例

sink	编译条件	当前板级示例及运行限制
usj	SoC 提供 USJ 且 MICROPY_HW_ESP_USB_SERIAL_ 启用。	P4 开发板及 ESP32_S31_CHATBOT；部分 IDF system console 原始输出仍有独立配置。
usb	SoC/port 提供 USB device 且 MICROPY_HW_USB_CDC 有效。	ESP32_S3_EYE、ESP32_S31_KORVO、ESP32_S31_MOSAICO、AtomS3R-M12；P4 板卡也可依据 SoC 支持同时暴露 CDC 与 USJ。
uart	MICROPY_HW_ENABLE_UART_REI 启用，使用板级引脚与波特率。	P4、ESP32_S3_EYE、AtomS3R-M12、ESP32_S31_KORVO 启用；当前 ESP32_S31_MOSAICO、ESP32_S31_CHATBOT 关闭。
tcp	ESP_VISION_ENABLE_TCP_TRAI 底层 WLAN 驱动可用。	STA 获得 IPv4 后才能监听。当前 ESP32_P4X_VISION 的 port 配置关闭 WLAN，不能由 TCP 编译开关推断其网络可运行。

板级例子见 boards/ESP32_P4X_EYE/boardconfig.h、boards/ESP32_S3_EYE/boardconfig.h 和 boards/ESP32_S31_MOSAICO/port/mpconfigboard.h；后端约束由 platform/ev_transport_config.h 检查。

USB-Serial/JTAG 与 USB-OTG 怎么区分

P4-EYE 可以同时提供 USJ 和 OTG CDC，两路都可能呈现 ttyACM，操作系统编号不稳定。主机应依据 USB 描述符、VID/PID、序列号和板卡连接拓扑识别接口，再用 hello/capabilities 确认设备身份。

USJ 与 TinyUSB CDC 使用不同外设/驱动路径。为其中一路发送 hello，响应只回到该路实际 link；不会因为另一接口是默认口就把响应复制过去。一个接口已持有会话时，另一个接口可发现设备，但其接管会返回忙。

P4-EYE 的运行态 CDC PID 当前为 `0x4003`。部分 S3 构建使用 PID `0x1001` 以选择 esptool 的 USB reset 序列；PID 看起来像 Serial/JTAG，并不让 TinyUSB OTG CDC 获得 JTAG 硬件能力。ROM 下载模式可能重新枚举为另一个节点，不能继续假定原句柄有效。

link = 后端 + 槽位 + generation

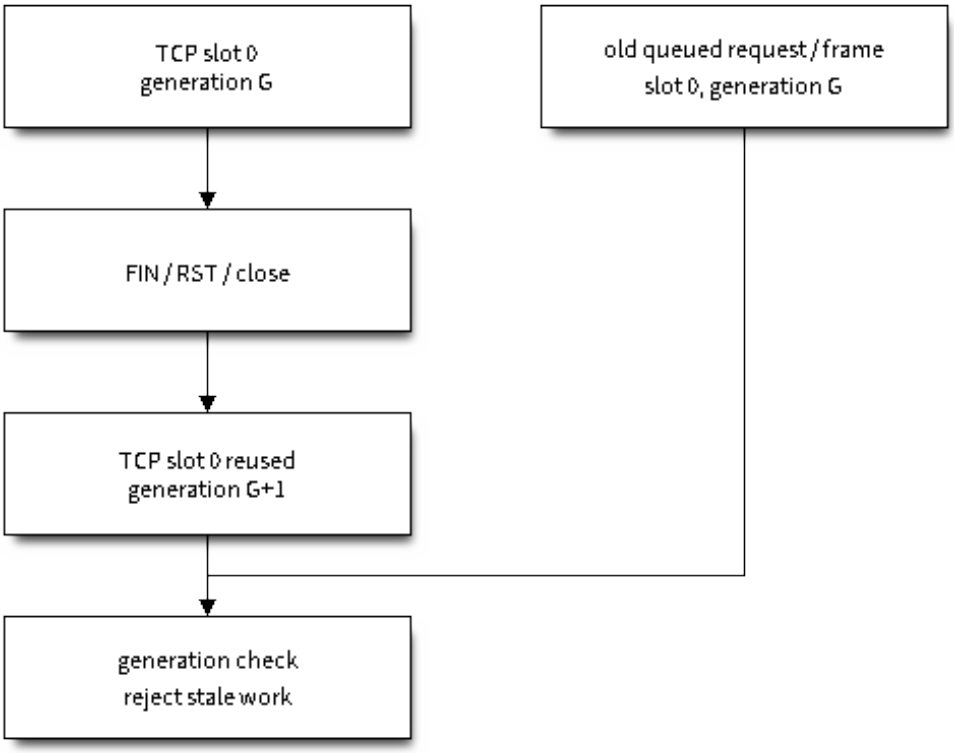


图 7：TCP 槽位复用不意味着旧连接继续存在。会话绑定准确的 link 引用；TX 目标、待执行 VM 请求和异步完成回调都需要避免向新连接投递旧业务。USB 真实断开与显式释放也会触发相应的失效边界。

板级配置不等于 IDF console 配置

表 5: 配置层

配置	作用
CONFIG_ESP_CONSOLE_*	ESP-IDF system console、启动与异常路径的底层输出路由；不直接决定 EV-MUX owner。
MICROPY_HW_ESP_USB_SERIAI	MicroPython/ESP-VISION USJ 编译与初始化。
MICROPY_HW_USB_CDC / MSC 相关 port 宏	现有 TinyUSB CDC/MSC 组成；不会安装第二套 USB driver 或 descriptor owner。
MICROPY_HW_ENABLE_UART_RE	UART REPL 是否成为可注册的串口后端。
ESP_VISION_TRANSPORT_DEFA /ESP_VISION_TRANSPORT_FAL	无 owner 的默认/回退串口。必须是实际启用的 usj、usb 或 uart；不能默认到尚未配网的 TCP。
ESP_VISION_ENABLE_BACKGRC	后台 Wi-Fi/TCP 服务策略；WLAN 硬件、原生/Hosted 驱动与 sdkconfig 仍需单独满足。
ESP_VISION_WIFI_DEFAULT_S /ESP_VISION_WIFI_DEFAULT_	可选出厂 Wi-Fi 凭据。默认空；只在没有保存 NVS 记录时采用。
ESP_VISION_ENABLE_TCP_TRA / ESP_VISION_TCP_PORT	板级 TCP 后端开关和监听端口，通常为 3232。

后端注册表在编译期确定，不提供运行期插件注册接口；USB 插拔、网络启停和 TCP 槽位状态仍是动态的。supported、present、ready 和 owner 分别回答“有没有实现”“物理连接是否已知存在”“能否收发”“是否持有会话”，不能相互替代。

console / null``（别名 ``none）只是 ev_channel.c 的本地输出选项，不是公开的注册 sink，不出现在 capabilities.sinks。不要用它们构造 v4 接管请求。

USB CDC 没有会话持有者时，即使 USB 仍处于挂载状态，关闭主机读取端也可能使默认/回退输出填满 FIFO。已发送一部分的帧会保留在有界发送队列中，直到主机恢复读取或物理链路代次改变；后续帧不能越过它。尚未开始发送的帧仍会过期，其他端点继续运行，会话绑定帧仍遵守原有发送时限，避免摄像头启动日志给下一次连接留下不完整帧。

UART 为什么不能照搬 USB 的预览大小

UART 应用通信以 115200 bps 启动。固件支持 capabilities.uartBaud 时，Device Core 按 2000000、1500000、921600、460800、115200 顺序测试，采用首个通过验证的速率。设备与主机串口必须一起切换；USB-OTG、USB-Serial-JTAG 和 TCP 不执行 UART 协商。prepare/probe/commit 与回退流程见帧格式、逻辑通道与 *RPC* 协议。ROM/stub 下载速率在烧录流程中独立协商。

UART 的发送预算使用当前运行态波特率，按整帧估算：

```
timeout_ms = max(1500, ceil(frameBytes * 10 * 1000 / baud) * 2 + 250)
timeout_ms <= 10000
```

10 表示常见 8N1 每字节占用的位数。超过 10 秒预算的帧在提交前被拒绝；支持 preview.frame 并不保证任意 JPEG 都能在低波特率上传完。应提高波特率、降低图像分辨率/质量或改用 USB/TCP，仍须处理发送失败和短写。

TCP 的三个时间预算

1. **首次握手**：从未接管的连接约 3000 ms 后关闭，因此 TCP 客户端不应先做 300 ms 串口启动排空或长时间 UI 操作。
2. **会话租约**：默认 15000 ms，无合法授权活动则释放 owner；这一步是所有权回收。
3. **曾持有会话后的空闲回收**：失去 owner 后，socket 有约 30000 ms 窗口可重新接管；窗口到期后回收该准确 generation 的槽位。这一步是连接容量回收。

三个槽位允许两个只读旁观者与一个控制者并存，详见预览、调试抓帧、背压与统计。竞争接管失败的 TCP 连接在发送错误后关闭；正常释放后的同一 link 可以在规定窗口内重新接管。

DTR/RTS、RESET 与 ROM 下载

DTR 变化不会取得或释放 EV-MUX 会话。已关闭主机串口句柄但 USB 仍连接时，固件未必立即知道；正确做法是关闭前发送 `session.release`，否则等租约到期。

CDC DTR/RTS 自动下载是另外一条 USB control 路径。它与 Python `machine.bootloader()` 共用 `esp_vision_restart(download_mode)` 底层函数；Python 请求可立即执行，USB control callback 只记录状态，延迟 timer 在回调外执行复位。P4 和 S31 设置/清除下载位时保留系统寄存器其他位；S3 还遵循其 ROM USB persistence/PHY 交接契约。S31 Korvo 和 Mosaico 启用这条 CDC 入口；Chatbot 保留实际连接的 USB Serial/JTAG，不启用未连接的 OTG 口。正式 2026.10.23 的 S31 固件尚无此支持，仍需要硬件方式进入 ROM。

此固件修改提供 S31 从应用进入 ROM 的能力。烧录后正常返回应用还要求主机清除并验证 S31 的下载标志，例如 Device Core 0.7.0；旧版 Device Core 0.6.1 不包含此步骤。发布此固件前，应先发布匹配的 Core，并由 IDE 和官网采用。

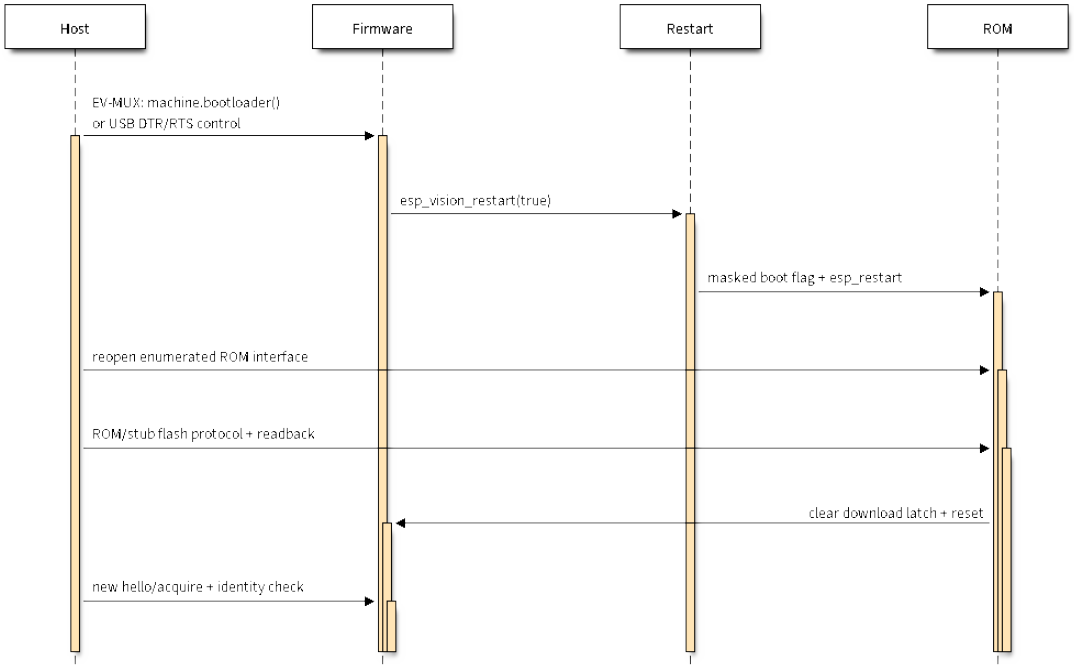


图 8：进入下载模式、写镜像与恢复运行态是三个独立确认点。 EV-MUX 自身不传输 ROM 烧写命令；EV-MUX 端口消失也不能单独证明 ROM 已就绪或烧写已成功。

P4-EYE OTG 从运行态进入 ROM 时，esptool v5 使用 `--before usb-reset`；已经进入 ROM、或等待重新枚举后重试时使用 `--before no-reset`。v4 工具对应拼写为 `usb_reset` / `no_reset`。下载工具还要确认写入、完整读回校验、下载位清除回读与运行态设备身份，不能把一个 `EIO` 直接当作成功。

9.2.3 USB 接口、配置与磁盘命名

本章说明 USB 配置和主机侧可见名称。EV-MUX 连接见*物理接口、连接代次与构建配置*，恢复、备份和写入所有权见*存储、文件系统与数据保护*。

物理接口与功能

表 6: USB 功能

接口	主机功能	配置与限制
Serial/JTAG（usj）	串口 / 调试及 ROM 下载入口。	由 SoC 及 MICROPY_HW_ESP_USB_SERIAL_JTAG 控制，不提供 MSC 磁盘。TinyUSB 字符串不能重命名这个硬件接口。
OTG CDC（usb）	用于 EV-MUX、Python 和原生文件传输的串口。	需要原生 USB device 连接、MICROPY_HW_ENABLE_USBDEV 和 MICROPY_HW_USB_CDC。USJ 可连接不代表 OTG 已连接。
OTG MSC	一个大容量存储接口，提供一个或两个 LUN。	需要 MICROPY_HW_USB_MSC 和可用介质；与 CDC 及启用的 UVC 共用现有 TinyUSB device，不另挂载一套文件系统。
OTG UVC	受支持构建中的摄像头 / 视频功能。	由 MICROPY_HW_USB_UVC 及现有描述符 / 流实现控制。枚举成功不代表相机已初始化或正在产帧。

根据板级原理图选择插座，枚举描述符、明确选择端口，再确认板型和固件身份。P4 板可能同时提供 USJ 与 OTG；复位或进入 ROM 后 COM / ttyACM 编号可能改变。下载模式交接见物理接口、连接代次与构建配置。

配置位置

表 7: 配置归属

位置	配置内容
boards/<BOARD>/port/mpconfigboard.h	Device / CDC / MSC / UVC 功能宏、USJ 开关、厂商 / 产品 / 接口字符串，以及板级 VID / PID 覆盖。
boards/<BOARD>/boardconfig.h	EV-MUX 默认 / 回退 sink、存储自动挂载 / 初始化策略。选择 sink 不会启用 USB 类或物理插座。
板级 sdkconfig.* 与共享 port 默认配置	IDF console、PHY / 驱动和 FatFs 设置。CONFIG_ESP_CONSOLE_* 不选择 EV-MUX owner，也不会让 USJ 提供 MSC。CONFIG_FATFS_USE_LABEL 启用共享卷标 API。
overlay/micropython/shared/tinyusb/	共用复合设备描述符和 class / endpoint 配置。保持唯一 TinyUSB 驱动 / 描述符所有者，不为磁盘另装 USB 栈。
platform/usb_msc.c 与 platform/storage.h	固定 LUN 映射和介质名称。MSC inquiry product 为 ESP_FLASH / ESP_SDCARD；MICROPY_HW_USB_MSC_INTERFACE_STRING 保留各板原有名称，不强制统一前缀。

修改板级 / overlay 源文件，通过仓库根目录的 idf.py --board <BOARD> ... 重新构建，不修改生成的 MicroPython 副本。VID / PID 必须符合实际 USB 功能和主机驱动约定；复制 Serial/JTAG PID 不会带来 JTAG 硬件能力。

卷标、LUN 与盘符

表 8: 磁盘身份

介质	Python 挂载位置	默认 FAT 卷标 / inquiry product	USB LUN
内部 Flash	/ 及别名 /flash	ESP_FLASH	0
SD 卡	默认 /sdcard	ESP_SDCARD	编译支持 SD 时为 1

FAT 卷标保存在介质上，通常由主机文件管理器显示。固件在自动挂载时，只为卷标为空的已挂载卷设置默认名；保留已有非空卷标，包括用户自定义 SD 卡名称。改名不会格式化磁盘。写入方准入拒绝或卷标 I/O 失败会记录警告，不导致 crash，也不把成功挂载转变为格式化请求。显式 `storage.retry()` 和写入方交接刷新仍只挂载，不修改卷标。

FAT 卷标最多 11 个 OEM 字节、不允许点号，小写转换为大写；长文件名不能放宽卷标限制。因此 `esp-vision.flash` / `esp-vision.sdcard` 不能作为 FAT 卷标，实际使用 `ESP_FLASH` / `ESP_SDCARD`。参见 FatFs 卷标 API。

USB 厂商 / 产品 / 接口字符串标识设备或功能；SCSI inquiry product 标识 LUN，字段为 16 字节。它们均不会设置 FAT 卷标。Windows 自行分配 `D:` / `E:` 盘符，其他系统自行选择挂载路径，固件不能预留这些盘符。识别磁盘应综合物理设备、LUN 和卷元数据，不能只记盘符或仅凭卷名判断目标。

Windows 资源管理器中的磁盘名称通常来自文件系统卷标。原先 Function EV 的 Flash 卷标为空，因此显示 “U 盘”；`ESP32-P4X-Function-EV-Board Flash` 是 USB MSC 接口字符串，描述同一个包含两个 LUN 的接口，不会成为每个卷的名称。SCSI inquiry product 可用于磁盘设备身份识别，同样不设置卷标。现在为无标签的已挂载卷设置 `ESP_FLASH` / `ESP_SDCARD`，即可区分两种介质；已有自定义卷标仍保留。无卡的 SD 槽没有可读取的 FAT 卷标，Windows 仍可能显示通用 “U 盘”，不能仅靠接口字符串为该空槽指定资源管理器名称。参见 Microsoft 的 卷命名 与 USB 接口描述符。

不可用介质在其固定槽位报告未就绪；Flash 失败不会把 SD 改为 LUN 0，`/flash` 不增加磁盘。不支持 SD 的板型只提供 Flash。空槽是否显示及盘符分配由主机决定。

挂载、所有权与验证

Flash 和 SD 自动挂载互不依赖，默认均开启。无法识别文件系统时是否格式化属于另一套策略：默认 `ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM=1`、`ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM=0`。磁盘命名不覆盖这些策略，也不授权格式化，参见*存储、文件系统与数据保护*。

EV-MUX owner 不自动拥有 MSC 写权限。Python / IDE 写入或备份前，应安全弹出对应操作系统卷，并按*交互式 Python、脚本与文件操作* 完成逐卷交接。双盘在线不允许主机 / VM 同时写入任一共享 FAT 文件系统；重要文件须保留经校验的设备外备份。

1. 检查选定设备及 `capabilities.sinks`。MSC 验收需要连接原生 USB device 端口；格式化卡不能解决 OTG 未连接。
2. 检查 `storage.status("flash")` / `storage.status("sd")`，卷已挂载且主机写入方准入成功后，对应 LUN 才可就绪。
3. 分别检查 USB 功能与逻辑磁盘，记录物理序列号、固件、LUN、实际卷标及系统盘符。仅看到 COM 口或产品字符串不能证明磁盘已挂载。
4. 用独立测试文件 / 哈希验证读写和复位后持久化，保留既有数据及自定义卷标。不可用介质应返回诊断，另一卷 / 控制连接仍可使用。
5. 分别报告 CDC、实际双 LUN 枚举和健康 SD 读写结果；仅在 USJ 上通过不能覆盖原生 USB MSC 或双盘。

9.2.4 会话所有权、接管与恢复

“抢占”在 v4 中的准确含义

所有支持的物理入口都能争用接管资格，但只有设备 ** 先受理 ** 的合法接管请求成为 owner。当前 owner 有效时，后来请求不会强制抢走控制权；默认口、DTR、USB 插入顺序、debug stream 都不是抢占优先级。

`route.bind` 和 `route.auto` 已停止作为公开操作，返回 `UNSUPPORTED_OPERATION`。user/debug 两个 stream 随唯一 owner 一起交接。要让主机 B 接手，应由 A 释放，或等待 A 的 link 失效/租约到期，然后 B 重新接管。

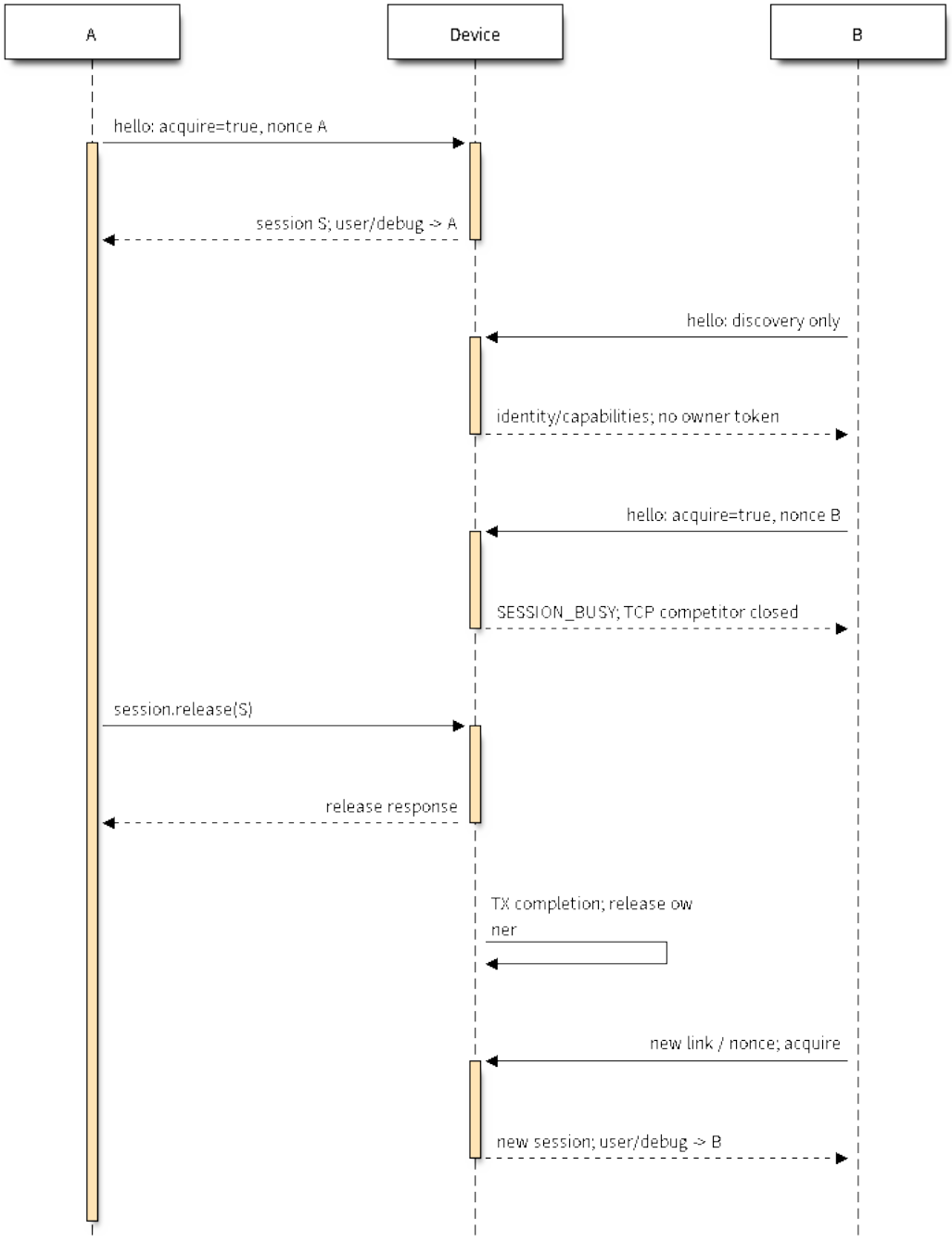


图 9：发现、竞争失败与显式交接。 B 能发现设备，不表示 B 可发送 Python 或文件命令；旧 owner 的 token 也不能交给 B 使用。

会话状态机

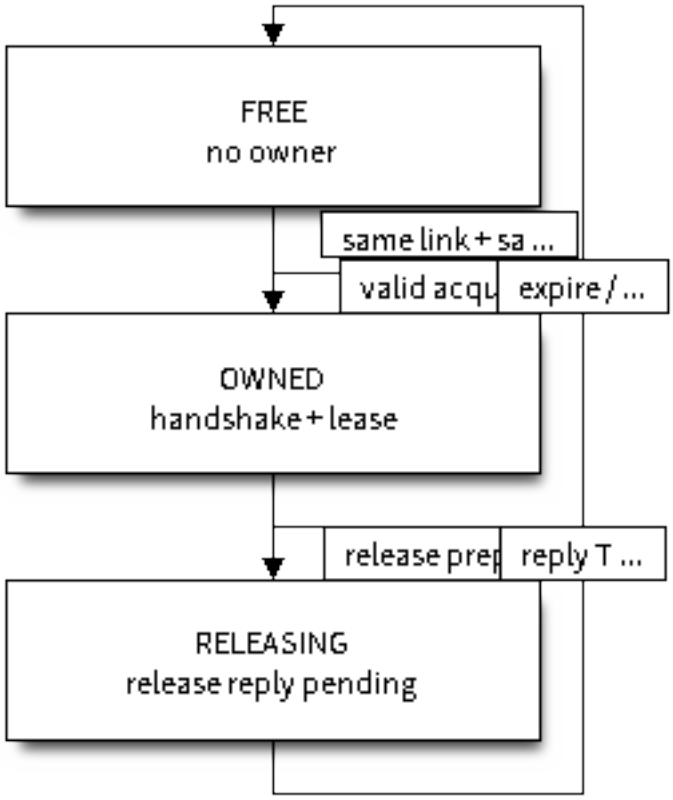


图 10：释放分成准备与完成，避免在回复发出前销毁回复所需的 link。已收到释放回复后立刻重接管，仍可能短暂遇到 `SESSION_RELEASING`。同 link 的主机可在总期限内用相同 nonce 重试该次 hello；不能把它变成无限自动重试。

首次发现与接管请求

串口打开后先处理启动残包；TCP 直接接管。以下代码块中的 metadata 与 payload 是同一个帧的两段内容，不是两个独立 JSON 帧：

```
{"seq":1,"channel":"user.rpc","type":"req","method":"hello","encoding":"json"}

{"evMuxVersion":4,"acquire":true,"clientNonce":"0123456789abcdef0123456789abcdef"}
```

以下为省略附加能力字段的成功 payload 示例：

```
{
  "ok": true,
  "evMuxVersion": 4,
  "session": {
    "id": "89abcdef0123456789abcdef01234567",
    "owner": "tcp",
    "leaseMs": 15000,
    "keepaliveMs": 5000,
    "vmEpoch": 1
  },
  "vmReady": true,
```

(续下页)

(接上页)

```
  "routes": {"user": "tcp", "debug": "tcp"}
}
```

必须先正确匹配 `rsp/hello` 的 seq、channel、method，再接受新 token。空 hello / capabilities 只是发现；发现响应回到其准确入站 link，不把有效 owner 的 ID 暴露给竞争连接。

`clientNonce` 是一次逻辑接管的 32 字符小写十六进制标识。只在同一 link 重试同一接管时复用它；新 socket、重新打开串口或新一轮接管应生成新值。同 link、同 nonce 的幂等 hello 可以返回同一有效会话，不能用于透明重放其他业务。

设备在接管回复 TX 完成前还有握手完成门控，不能因为某个生产者已生成 sessionId 就在主机收到有效回复前发送依赖会话的操作。

保活不是应用完成确认

```
{"seq":2,"channel":"user.rpc","type":"req","method":"session.keepalive","encoding":
↪"json","sessionId":"89abcdef0123456789abcdef01234567"}

{}
```

默认每 5000 ms 保活，租期 15000 ms；合法、已授权的业务活动也续租。主机应按设备报告值调度，并在长脚本、长文件传输和持续预览时继续读包及保活，不能“等当前操作完成再续租”。

保活成功只表示当前 link/session 有效，并可报告 VM 代次与就绪；它不证明脚本执行完成、文件已完整发布或某次网络配置已生效。

四个身份/关联维度

表 9: 不可互换的字段

维度	作用	不可推导的事实
link generation	拒绝旧物理连接残留的工作。	同一端口名或 TCP slot 不保证仍是同一次连接。
`sessionId`	授权某个准确 link 持有当前会话。	不能从发现响应、其他主机或旧 socket 转移过来。
`vmEpoch`	对 VM 操作与 VM 输出标识代次。	代次前进不等于 VM 或 REPL 已空闲可执行。
`seq` + channel + method	在一个请求上下文中关联响应。	相同 seq 的 event、preview 或旧响应不能完成当前 RPC。

所有非发现业务携带 sessionId。绑定 VM 代次的 `file.write` / `script.write` / `script.run`、相机/对象/VFS 调试、`repl.stdin` / `repl.signal` 还携带当前 vmEpoch。传输统计、保活、释放和后台网络控制不必依赖 VM 空闲，但仍要求相应的 owner 授权。

启动 VM-ready 与软复位

当前初始 `vmEpoch` 是随机非零 uint32；较早的 v4 构建可能在启动阶段报告 0。传输控制应保留实际返回值，VM 操作则等待正数 epoch 与 `vmReady:true`。非零序号从 4294967295 回绕到 1，主机按序号比较规则判断更新，不能简单要求新数值大于旧数值。

软复位可以保持原 sessionId 和网络；新的 epoch 在复位开始推进。`event/hello` 是额外通知，不能只靠它判成功：应使用关联的 keepalive 回复确认当前会话的新 epoch/ready，再执行当前代次的 REPL barrier，要求真实的新标记。

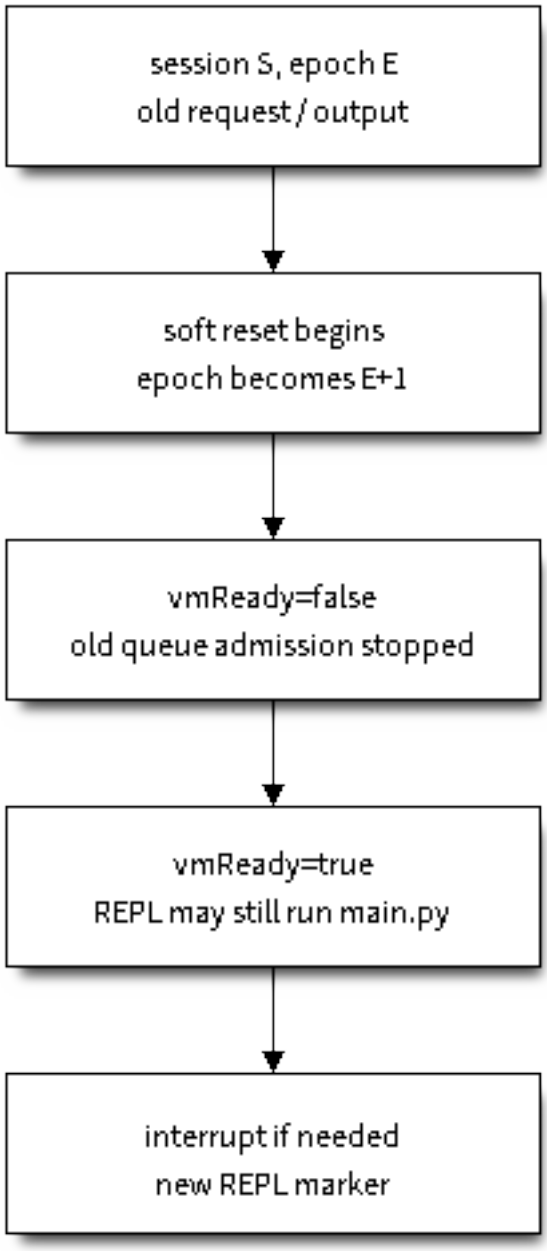


图 11：传输存活、VM 就绪和 REPL 可执行是不同条件。 `boot.py` 或 `main.py` 重新运行后仍可能占用 REPL；不能仅看到 ready 就立即开始 VFS/相机 VM RPC。

硬复位/进入 ROM 则必须废弃旧 token、epoch、待响应请求和解码器状态，恢复运行态后重新发现/接管。软复位时只重置旧 VM 的输出解析状态，不能把正在拼接的整条物理 EV-MUX 字节缓冲随意清空。

断线、FIN/RST、静默消失

- TCP FIN/RST 或已确认 link 失效：回收对应 owner，旧 generation 的输出失效；创建全新连接与 nonce，再接管。
- USB 真实断开：link 失效。仅关闭主机串口、改变 DTR 不保证设备得知句柄已关闭，应显式释放。
- USJ/UART 静默消失：等租约回收 owner；端口类型和 default/fallback 不授予提前接管权。

- 曾持有会话的 TCP link 失去 owner：仍可能保留 30 秒 idle 窗口，然后回收 slot；此预算与 15 秒租约不同。

租约释放不会自动撤销已经完成的外部副作用，也不能视为“前一个用户程序保证已经停止”。B 接管后应确认 VM 状态，按需要显式中断，并用新标记确认 REPL 可执行。

错误与恢复决定

表 10: 常见错误

错误	主机处理
SESSION_BUSY	当前另一个 owner 有效；提示释放/等待，而不是抢占或重用其 token。
SESSION_RELEASING	同 link 释放完成边界尚未结束；仅在有界接管流程内重试同 nonce。
SESSION_REQUIRED / NOT_OWNER	停止发送依赖旧会话的业务，废弃本地会话并重新建立。
VM_EPOCH_MISMATCH	原 VM 请求不再有效，重新确认当前代次；不要自动重放脚本或写文件。
VM_BUSY / VM_TIMEOUT	VM 请求无法入队或未及时处理；先处理当前脚本、取消/复位与真实 REPL liveness。
链路发送失败 / 主机等待超时	可能已产生副作用，标记完成情况未知；先查询或核验文件/事务状态。

带相同 operationId 的网络事务有专门的关联与恢复语义；seq 本身不是所有 RPC 的幂等键。脚本执行、REPL 输入和文件 append 没有通用透明重放契约。

9.2.5 帧格式、逻辑通道与 RPC 协议

三个版本号

firmware.version 表示 Git/发布构建身份；evMuxVersion 表示客户端必须遵守的会话协议；帧头 EVMUX/1 表示 wire framing。应先用 evMuxVersion 与 capabilities 判断兼容性，再以 firmware.version 展示、记录或校验指定固件，不按日期字符串猜测协议。

只读固件安装证据

firmware.status 是 user.rpc 方法，仅启用 Iris 打包的 ESP32_S31_MOSAICO 构建声明 firmware.status.v1 能力。调用前先获取 EV-MUX v4 owner 会话。请求为 {"imageSize": N}，N 必须是已校验包中应用 BIN 的准确字节数（1–7340032）。请求在传输任务执行，不进入 REPL、不执行 Python、不写入、不复位。缺失或错误长度返回 INVALID_ARGUMENT，VM 未就绪返回 NOT_READY，OTA 未确认或安装证据不可读返回 EVIDENCE_UNAVAILABLE。

成功响应包含 ok=true、schemaVersion=1、deviceId``（出厂 MAC 的 12 位小写十六进制）``、``board、firmware.id/version、32 位十六进制 bootId、vmReady=true、bootConfirmed=true、IDF 数值 resetReason 和 crashReset。启动标识在 VM 软复位后保持不变，硬件重新启动后变化，安装确认适配将其前 8 字节按小端解释为 Iris Boot ID。

application 包含 address、分区 capacity、请求的 imageSize、完整 BIN 的 sha256、elfSha256 和 otaState="valid"。完整 BIN 哈希包括指定长度内的全部字节和末尾镜像摘要，不等同于 ESP-IDF 的内嵌镜像哈希。partitionTableSha256 覆盖配置分区表地址起的 4096 字节。installation.operationId 和有符号 installation.result 只读返回保留安装器持久化的提交回执，不发送或实现其协议。回执缺失或格式错误时查询失败，不生成成功回执；非零结果原样返回，主机必须将其判为验收失败。

安装验收应对照本次目标，核对实时设备身份、板型、包内准确长度、哈希、布局和操作 ID，要求提交结果为零、`crashReset=false`，并在验收前后观察相同的启动标识。其他镜像或历史操作的健康响应不能证明本次安装成功。启动确认不验证用户 Python 代码或外设。本接口不进入 Vibe Mode，后续安装由用户通过 AI + Reset 手动进入。

一帧的字节布局

```
0x1e + ASCII "EVMUX/1 h=<H> p=<P> c=<CRC32>\r\n"
       + H bytes of UTF-8 metadata JSON
       + P bytes of payload
       + 0x1f
```

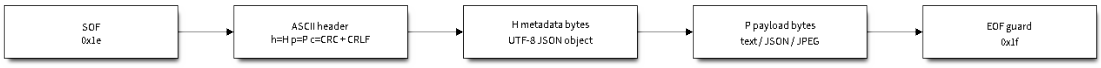


图 12：只有 H/P 给出正文边界，EOF 字节不是 payload 分隔符。 JPEG、二进制文件数据或文本中可以包含 `0x1e`、`0x1f`、CRLF；不能按这些字节截断正文。

H/P 使用 UTF-8 或实际二进制的 ** 字节数 **。例如字符 “你” 通常占三个 UTF-8 字节，不能把 JavaScript 字符串 length 直接当成 payload_bytes。

CRC32 只覆盖 payload，不覆盖 metadata/header；使用标准 CRC32（主机 Python 可用 `zlib.crc32(payload) & 0xffffffff`），以八位十六进制表示。设备生成的帧带 payload CRC；入站命令 `c=00000000` 被解释为跳过校验。CRC 提供损坏检测，不提供身份认证，也不证明操作最终完成。

JPEG 响应和预览还需独立验证图像有效性。固件在入队前校验编码输出，Core 在交付前再次检查 JPEG 结构与尺寸；不能只凭 EV-MUX 帧完整或 CRC 正确就交付图像。坏抓图响应在本地返回 `INVALID_IMAGE`，坏预览整帧丢弃；校验范围及限制见预览、调试抓帧、背压与统计。

主机必须处理任意拆包与粘包

```
read #1: [SOF][header first half]
read #2: [header tail][metadata first half]
read #3: [metadata tail][payload first half]
read #4: [payload tail][EOF][whole next frame][next SOF]
```

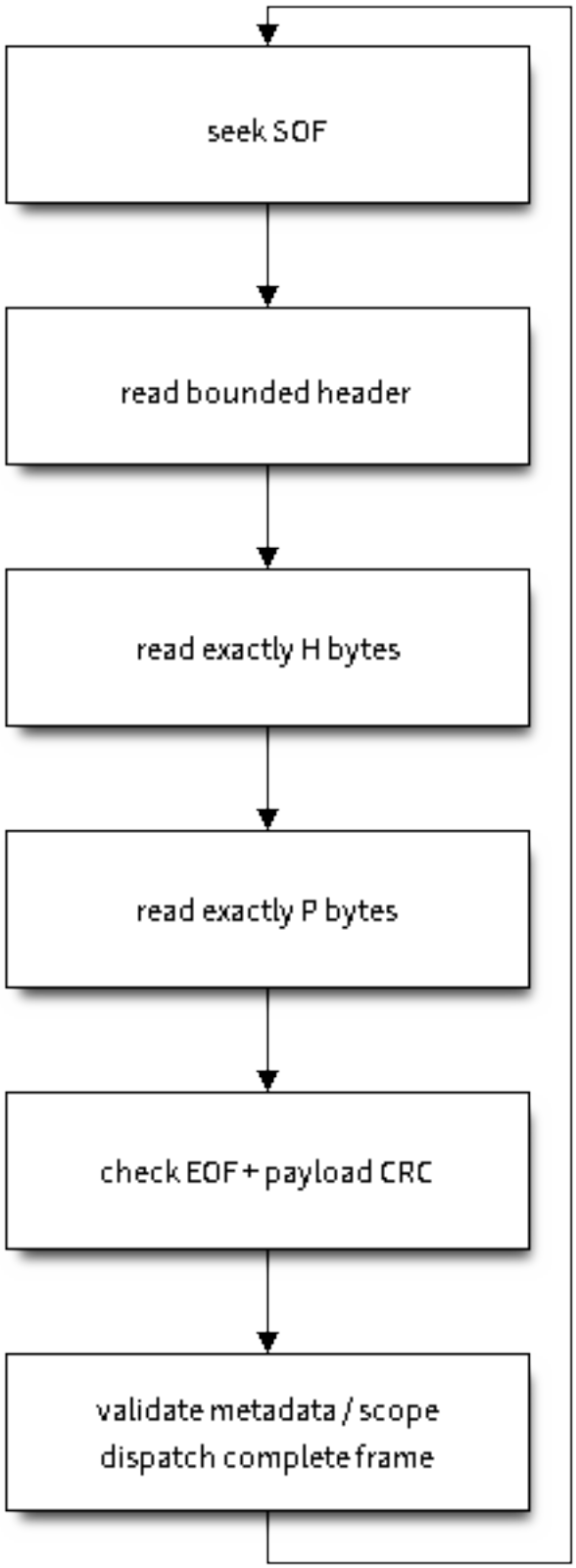


图 13：流式 parser 保存跨 read 的状态，完整帧验证后才交付。若格式、长度、CRC 或期限不满足，应记诊断并执行受控重同步；不得把半帧送进 REPL 或文件写入。

当前设备 RX metadata 上限 384 字节、payload 上限 2048 字节。RX parser 有约 3000 ms 空闲和 10000 ms

整帧期限。这些是 ** 设备接收主机命令 ** 的限制，不是 preview 最大图像尺寸，也不应机械作为主机收大图的超时。

metadata 字段

表 11: 公共字段

字段	类型	用途
`channel`	字符串	指定 channel，例如 `user.rpc`、`repl.stdout`。
`type`	字符串	RPC 请求 `req`、响应 `rsp`、异步通知 `event`、数据 `data`。
`seq`	uint32	请求响应相关序号；不同 channel/类型上的相同值不能互相完成请求。
`method`	字符串	RPC 方法或 stdio 来源描述；不能把 `c-stdio` 当成 RPC 调用。
`encoding`	可选字符串	`json`、`text`、`binary` 等语义提示。
`sessionId`	32 字符小写十六进制字符串	当前 owner 授权/输出范围；发现请求不带 token。
`vmEpoch`	uint32	VM 相关操作及输出范围。
`sid`	可选字符串	逻辑来源/域，例如 `stdio`；不是 sessionId。
`ts_ms`	可选整数	设备时间戳，不与主机时钟天然同步。
`contentType`、`width`、`height`	按 channel 定义	描述图像 payload；JPEG 字节仍在 payload，而非 JSON 中。

metadata 与 RPC 请求 payload 必须是完整顶层 JSON object；已知字段类型正确、不能重复。未知扩展字段可以忽略。字符串中的转义不能用简单的逗号分割器解析，JSON 后多余垃圾也不能被视为完整合法对象。

channel 与 stream 对照

表 12: 线上通道

channel	方向	stream	payload / 用途
`user.rpc`	双向	user	JSON 控制：发现、会话、脚本、设备/网络事务。
`debug.rpc`	双向	debug	调试 RPC；capture 的成功响应直接是二进制 JPEG。
`repl.stdin`	主机 → 设备	user	原始 REPL 输入字节；有 session/epoch 授权与独立长度上限。
`repl.signal`	主机 → 设备	user	当前仅合法单字节 `0x03`，表示 Ctrl-C。
`repl.stdout`	设备 → 主机	user	Python/REPL/HAL 与 C stdout 的文本块。
`repl.stderr`	设备 → 主机	user	C stderr 等显式 stderr 路径；Python 异常也可能在 stdout。
`log.idf`	设备 → 主机	debug	IDF/平台日志文本。
`preview.frame`	设备 → 主机	user	`img.flush()` 产生的 JPEG 数据帧。

所有 channel 所属 stream 固定；host 不为 preview、stdout 和 RPC 各绑定一个不同串口。owner 持有两组 stream，RPC 响应回到请求的准确 link。

RPC 方法清单与执行层

表 13: EV-ATP 控制面

channel	method	执行层	说明
user.rpc	`hello` / `capabilities`	transport	发现；hello 可显式接管。
user.rpc	`session.keepalive` / `session.release`	transport	续租、查询 ready/epoch、释放 owner。
user.rpc	`transport.uart.prepare` / `transport.uart.probe` / `transport.uart.commit`	transport	UART owner 协商波特率；提交或回退前暂停普通 UART 输出。
user.rpc	`file.write`	transport / native VFS	可靠二进制上传；绑定会话/VM 代次，不执行解释器。
user.rpc	`script.write` / `script.run`	VM	旧版 Base64 写入 / 排入脚本执行。
user.rpc	`device.control`	主要为 transport	stop/reset/status、network.*、storage.writer.*、launcher 状态等按 action 分发。
debug.rpc	`transport.state` / `route.get`	transport	状态与路由观测，不更改公开 stream 绑定。
debug.rpc	`debug.info`	VM，stats 例外	device/memory/sensor/fs.list/fs.read；transport.stats 不依赖 VM。
debug.rpc	`debug.capture_frame`	VM	在此 RPC rsp 中编码应用最近发布的独立采帧副本；不初始化传感器或物理采帧。要求 `debug.capture_frame.observed.v1` 能力。
debug.rpc	`debug.capture_screen`	VM	LCD 最后 RGB565 帧编码 JPEG，在此 RPC rsp 中返回；不调用摄像头。
debug.rpc	`route.bind` / `route.auto`	拒绝	v4 返回 `UNSUPPORTED_OPERATION`。

高层 Device Core 的 `files.read/remove` 与 `python.execute` 是客户端 API，不代表线上存在同名 RPC。文件与 Python 的具体组合见*交互式 Python、脚本与文件操作*。

`debug.capture_screen` 请求通过 `debug.rpc` 携带当前会话与 VM epoch，可选 `quality` 为 1–100。成功响应保持原请求 seq/method，`encoding:binary`、`contentType:image/jpeg`，携带 `width` / `height` 与 JPEG payload。无 LCD 返回 JSON error `NOT_SUPPORTED`，尚无有效显示帧返回 `DISPLAY_NOT_READY`；不能用摄像头图像代替。Core 的 `capture({ source: 'screen', format: 'jpeg' })` 使用这一 RPC，详见*预览、调试抓帧、背压与统计*。

请求、响应与异步事件可以交错

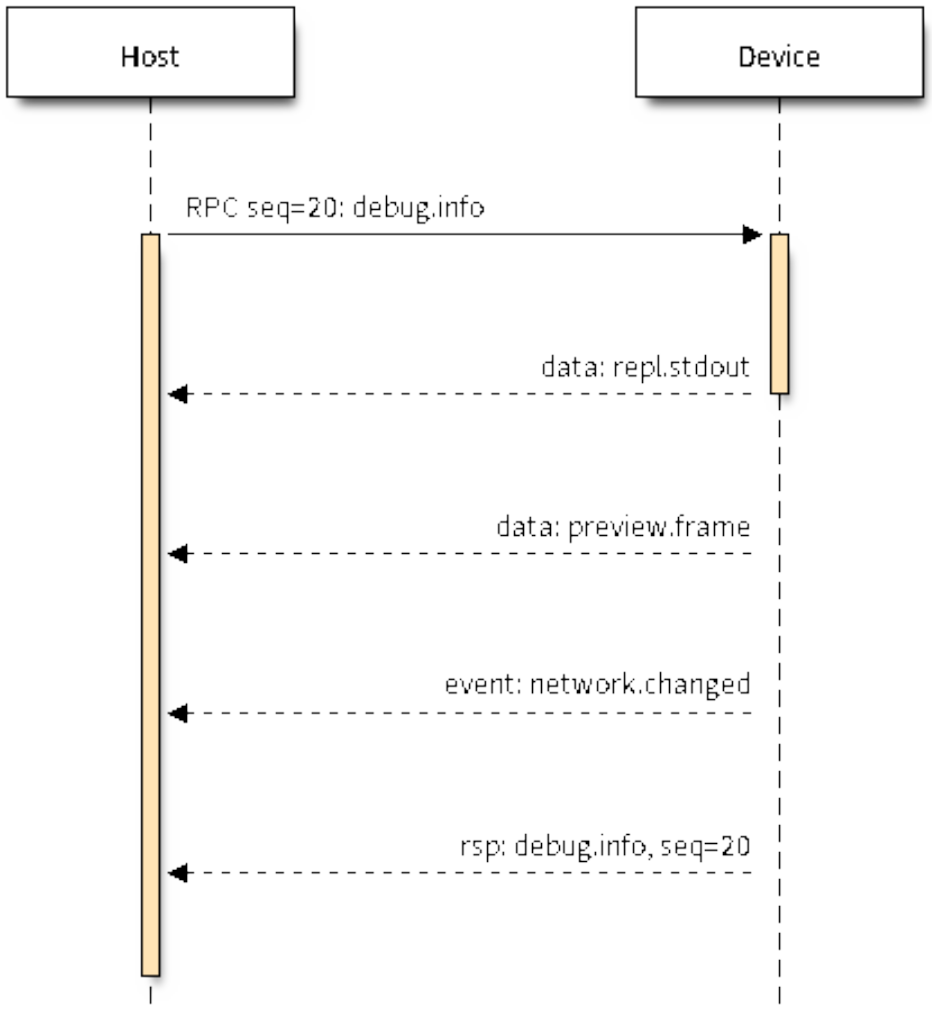


图 14: 等待 seq=20 的响应时，reader 仍应处理其他完整帧。响应匹配至少比较 type、seq、channel、method；随后按适用规则检查 session/epoch。`event/hello` 即使恰好使用同 seq，也不能作为 `rsp/hello`。

接管响应可分配新的 session；发现响应没有 owner 授权范围。VM 成功响应必须属于请求的 epoch；控制响应可能报告更晚的当前 epoch。部分授权失败响应不带 session 字段，须按错误码处理，不采纳里面的值为新的授权。

收到 owner 的 `repl.stdout`、`repl.stderr`、`preview.frame` 时，拒绝其他 session 或旧 VM 的输出；图像与控制回复不能根据“最近的请求”猜归属。

一个可验证的主机编码器

以下为 ** 主机 CPython** 示例，用于解释 framing；真正通信推荐使用*主机接入、Wi-Fi 配置与恢复实践*中的共享客户端。

```
import json
import zlib

def frame(metadata, payload):
    meta = json.dumps(metadata, ensure_ascii=False,
```

(续下页)

(接上页)

```
                      separators=(",", ":")).encode("utf-8")
    data = payload if isinstance(payload, bytes) else payload.encode("utf-8")
    crc = zlib.crc32(data) & 0xffffffff
    header = ("\x1eEVMUX/1 h=%d p=%d c=%08x\r\n" %
              (len(meta), len(data), crc)).encode("ascii")
    return header + meta + data + b"\x1f"

request = frame(
    {"channel": "user.rpc", "type": "req", "method": "hello",
     "seq": 1, "encoding": "json"},
    json.dumps({"evMuxVersion": 4, "acquire": True,
                "clientNonce": "0123456789abcdef0123456789abcdef"}),
)
```

该函数只编码帧，不实现会话、并发 reader、期限、RPC 关联或恢复。它不能替代完整客户端，也不能在设备 MicroPython 中直接假定可用主机 `zlib`/线程/socket 组合。

file.write 是经过会话授权后的有界例外：二进制 payload 可携带 16384 字节文件内容和 20 字节 ID/offset 前缀，并校验 CRC32；其他控制 payload 仍限制 2048 字节。begin/data/commit/abort 见 *交互式 Python、脚本与文件操作*。

运行态 UART 波特率协商

启用 UART 的固件通过 `capabilities.uartBaud` 发布 `baseline:115200`、`current`、`candidates:[2000000,1500000,921600,460800,115200]`、`probeBytes:512`、`timeoutMs:5000` 和 `switchDelayMs:100`。未启用 UART 时该字段为 `null`。这些参数用于应用运行期间的通信；ROM/stub 烧录使用独立的协议与连接生命周期。

已授权的 UART owner 在 `user.rpc` 上携带当前 session ID 调用以下传输层方法，VM 软复位期间仍可调用。USB-OTG、USB-Serial-JTAG 和 TCP owner 无权改变 UART 波特率。

1. `transport.uart.prepare` 接受 `{baudRate,token}`，速率须为公布的候选值，token 为 1–32 个 ASCII 字母、数字、下划线或连字符。固件以 **旧速率**返回 `{ok:true,baudRate,token,timeoutMs,switchDelayMs}`。协商期间暂停普通 UART 输出，让已开始的帧发送完毕，并确认硬件 FIFO **及移位寄存器**排空，再在公布的延迟后切换。主机传输适配器重新打开或调整串口，并等待公布的延迟加保护时间后开始探测。
2. `transport.uart.probe` 接受 `{token,data}`，data 为 1–512 字节可打印 ASCII。响应原样返回 `token`、`data` 以及 `baudRate`。主机使用不同长度和内容进行多组双向校验，按候选列表顺序选择首个通过的速率并停止。探测帧使用普通 EV-MUX CRC32。
3. 至少一次探测后调用 `transport.uart.commit`，参数为 `{token}`，响应为 `{ok:true,token,baudRate,...}`。提交响应进入硬件后恢复普通输出。相同已提交 token 的 probe/commit 支持幂等重试，以恢复丢失的最终确认。

未完成的协商在 5000 ms 后超时，释放 owner、恢复 115200，并清空 UART RX 与解析状态。协商成功后，正常释放会话或租约过期同样恢复 115200；丢失最终提交响应的情况可能需要通过 15000 ms 会话租约恢复。硬复位从 115200 启动，VM 软复位保留物理速率与会话生命周期。旧固件未发布该能力时保持初始速率，主机不得试发变速命令。

9.2.6 交互式 Python、脚本与文件操作

一个解释器，多种主机操作

友好 REPL、脚本执行和文件查询共用一个 MicroPython VM；二进制上传在相同会话、VM 代次和存储交接门控下直接写原生 VFS，不执行 Python。Core 仍串行化文件与解释器操作。一个长时间占用 VM 的程序，会影响 REPL 和需要 VM 的 RPC，但不会让 transport task 自动失去保活/网络控制能力。

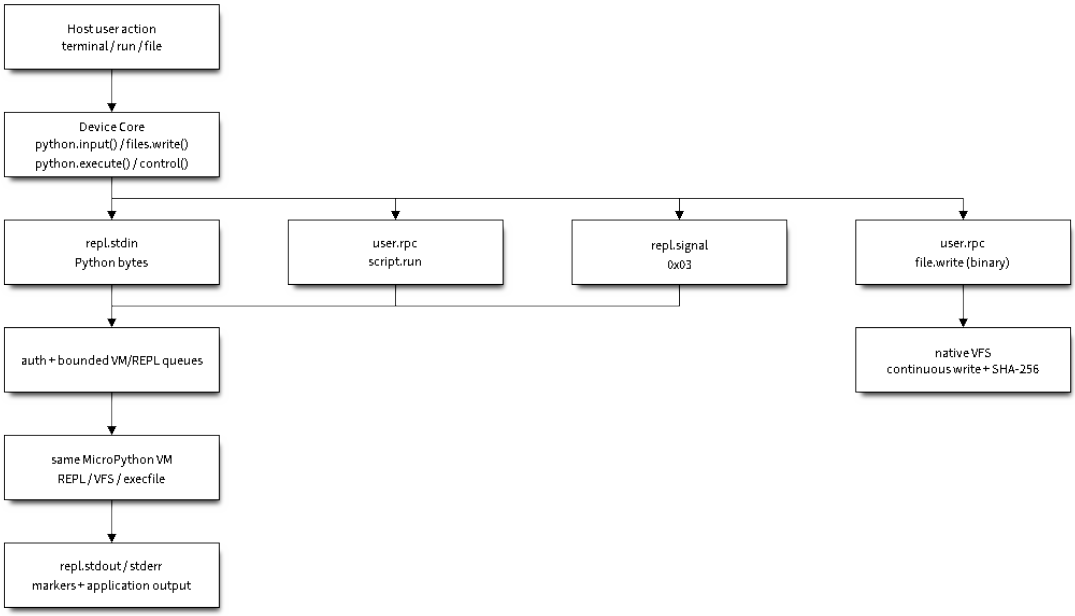


图 15：交互输入、文件命令和运行命令是不同入口，但不提供独立解释器隔离。一个 session 的 owner 应协调这些操作，不能让两个调用方同时拼接 REPL 命令。

交互式 Python：framed stdin

以下 metadata 与 payload 组成一帧，示例 token/epoch 仅为占位，实际必须来自本次接管：

```
{"seq":10,"channel":"repl.stdin","type":"data","encoding":"text","sessionId":
↪"89abcdef0123456789abcdef01234567","vmEpoch":1}
```

```
print("hello from current VM")
```

Python 文本之后还应发送适合 REPL 的行结束，例如 `\r\n`。`repl.stdin` payload 是 Python 字节，不是 `{"code":"..."}` JSON。该 channel 本身不是 RPC，不为每次输入返回同 seq 的 `rsp`；主机观察 REPL 输出并自行关联执行标记。

当前 capabilities 的 `replInputMax` 通常为 511 字节，来自 512 字节环形队列的有效容量。必须按实际编码后长度限制每个输入帧；多行输入、中文、emoji、Python 转义、CRLF 都占用字节。不要把 2048 字节控制 payload 限额误用于 stdin。

Device Core 使用 ASCII 转义的 Python literal 构造执行行，并将多行源码放进 `exec(...)`，避免中文源码的 UTF-8 字节预算与解释器输入容量不匹配。**具体编码策略是客户端实现，不是 firmware 自动转义任意超长源码的能力。**

Ctrl-C、Ctrl-D 与停止程序

Ctrl-C 必须经 `repl.signal`，payload 只允许单字节 `0x03`：

```
{"seq":11,"channel":"repl.signal","type":"signal","sessionId":
↪"89abcdef0123456789abcdef01234567","vmEpoch":1}
```

```
payload hex: 03
```

EV-MUX 启用时，在物理串口上裸发 `0x03` 不会被当作 Ctrl-C，它是无效/不完整 framing 输入。将字符“03”或 JSON `3` 填入 payload 也不是同一个字节。

友好 REPL 的 Ctrl-D（`0x04`）可以作为 stdin 控制字节触发其软复位行为；Device Core 的软复位流程也可通过 framed Python 命令请求。`device.control` 的 `action:stop` 提供控制面的停止入口，但停止请求被接受不等于被中断代码已退出。

停止后，应等待当前 session/epoch 中新的完整标记，证明 REPL 可执行。脚本若正在无法及时返回的原生调用中，Ctrl-C 可能延后；不能由一次 RPC ack 推断它已释放相机/文件资源。

完成标记与命令回显

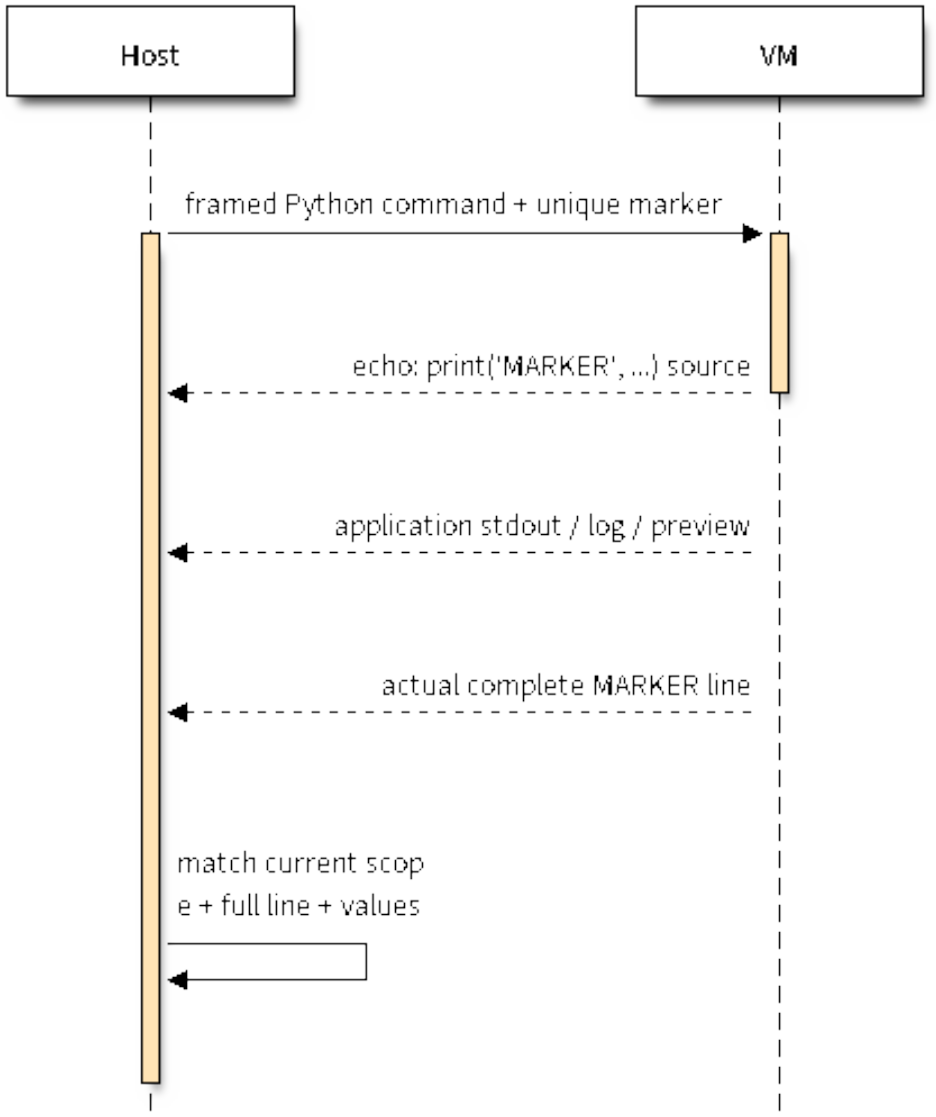


图 16：回显包含标记字符串，实际执行后的打印行才是完成依据。主机需要跨帧拼接完整行；仅用字符串 contains 查找 marker，会把回显误认为成功。

标记应每次运行唯一。结合 sessionId/vmEpoch、完整行结构和预期结果值判断；对摄像头任务，还要验证实际图像尺寸或其他应用证据。异常可能出现在 stdout，stderr 为空也不是成功依据。

可靠二进制文件上传

SDK 0.2.0 的 Python、模型、图片与其他字节文件共用一个上传引擎；大脚本执行先经该引擎上传临时源码，再 script.run。公开 files.write 参数保持不变。旧 v4 固件缺少 file.write.binary 时仍可查询，但须更新固件后才能上传；Core 不静默回退旧写入器。固件的 script.write 只为此前已发布客户端保留。

控制和数据均使用 user.rpc 的 file.write 方法，携带当前 sessionId/vmEpoch 与对应 seq。小型 JSON 控制仍限制 2048 字节。主机在 begin 前生成新的 16 字节随机 upload ID，begin 响应丢失或取消时仍可针对该 ID 发 abort。

```
{"action":"begin","path":"/models/example.tflite","mode":"overwrite","totalBytes
↪":2097152,"sha256":"<64 lowercase hex>","uploadId":"<32 lowercase hex>"}
{"ok":true,"uploadId":"<same ID>","chunkBytes":16384}
```

数据请求 metadata 设置 encoding:binary，payload 为 16 字节原始 upload ID、4 字节小端 offset，随后是 1..16384 字节文件原始内容；CRC32 覆盖整个 payload（含前缀）。非最后一块按 4096 字节对齐，最后一块可较短。NUL、CRLF、SOF、EOF 都是普通数据，不做 Base64 或 Python 输入解析。固件返回 JSON {ok:true,writtenBytes:N}，Core 校验后才发送下一块。重复、跳跃和乱序 offset 被拒绝，不提供随机写、自动重放或断点续传。

```
{"action":"commit","uploadId":"<same ID>"}
{"ok":true,"bytes":2097152,"complete":true}
{"action":"abort","uploadId":"<same ID>"}
```

固件在整个上传期间保持一个原生临时文件打开，并持有一次完整 VFS 操作的准入计数，阻止 MSC 在活跃描述符尚未关闭时交接。commit 校验总长度与流式 SHA-256，随后 fsync、close、发布；空文件只需 begin/commit。接收器只有在确认当前所有者的 file.write 二进制请求后才分配大块帧存储，其他控制 RPC 限额不变；二进制 payload 上限为 16384+20 字节，与 JSON 的 2048 字节限额分开。

上传绑定物理接口代次、会话、VM epoch 和会话转换代次；旧所有者不能追加或取消新的上传。abort、旧代次、失去会话或 30 秒无文件进展会关闭并删除临时文件；关闭失败保留存储交接屏障，不伪造释放成功。虽然原生 I/O 不执行解释器，VM 复位仍使上传失效。启动时不预留文件传输大缓冲。

传输与完整性检查期间保留原目标。FAT 无法单次重命名覆盖已有目标，因此替换时先保留旧文件备份，已校验临时文件发布成功后才删除备份，普通发布失败会尝试回滚。FAT 目录更新之间掉电可能留下临时/备份文件，不承诺日志式掉电原子替换。commit 成功但响应丢失时，主机无法确认完成状态，重试前应核验目标。SHA-256 校验内容一致性，不用于认证恶意主机。

上传设计与实测性能

文件本质是字节序列，扩展名不改变传输语义；上传与执行是不同操作，因此 Python、模型和图片共用一个二进制引擎。整次传输只打开、哈希和关闭一次，替代每小块重新 open/append/close。原始 16 KiB 块减少封装与往返，非末块按 Flash 的 4 KiB 擦除粒度对齐。方案复用现有 EV-MUX 帧和所有权，不增加物理连接、公开上传 API 或传输窗口。

2026-10-16，P4X-EYE revision 3.2 使用 ESP-IDF 6.1 固件 `af41759` 与 SDK 0.2.0 开发源码，在现有 Flash 文件系统得到以下上传耗时。计时包含 begin、数据与校验提交；每次上传后另外在设备端完整读取文件计算 SHA-256，全部一致。这些是当前设备实测样本，不是所有开发板的吞吐保证。

表 14: P4X-EYE Flash 上传样本

接口	256 KiB 旧写入器	256 KiB 二进制上传	2 MiB 二进制上传
USB-Serial-JTAG	40.94 s	7.24 s	55.75 s
USB-OTG	40.75 s	7.01 s	55.55 s
TCP / Wi-Fi	92.33 s	9.26 s	62.17 s

256 KiB 的数据 RPC 从 292 次降为 16 次，另加 begin/commit；应用层收发总字节从文件大小约 2.14 倍降至 1.04 倍。本机 Flash/FAT 顺序写入基线约为 55 KiB/s，这是当前存储实现的基线，不是 USB 物理带宽上限。小文件仍需承担 begin/commit 与 FAT 元数据的固定成本。进一步加速应继续测量存储与接收调度，而不是继续放大 JSON 分块。

公开文件 API 与线上实现

表 15: 文件接口分层

主机操作	线上实现	使用场景
`device.files.write({ path, data, overwrite, ... })`	Core 在 user.rpc 使用 file.write 控制与原始二进制块，固件持续写原生 VFS，完整校验后提交；契约见上文。	应用上传字符串或二进制字节的统一入口，不自动执行 Python。
Device Core `EvRepl.writeFile()`	framed REPL 执行 Python `open` / `binascii.unhexlify` / `write` / `rename`。	高级底层辅助方法，不是当前应用 `files.write()` 的实现路径，也不是名为 writeFile 的固件 RPC。
`device.files.read()` / `list()` / `remove()` / `mkdir()`	framed REPL 执行 `seek` / `read` / `os` 查询 / 删除，并解析受控输出。	支持较大文件、结果校验与取消，仍要求 VM/REPL 可执行。
`debug.info` 的 `fs.list` / `fs.read`	`debug.rpc`，受限 JSON/base64 调试响应。	快速诊断；当前 fs.read 只读起始最多 384 字节，不是带分页 offset 的完整下载 API。

旧版 script.write 请求与分块

以下契约仅为 SDK 0.1.x 客户端保留，SDK 0.2.0 不再调用。

```
{"seq":20,"channel":"user.rpc","type":"req","method":"script.write","encoding":
↪"json","sessionId":"89abcdef0123456789abcdef01234567","vmEpoch":1}

{"path":"/hello.py","encoding":"utf-8","mode":"overwrite","offset":0,"totalBytes
↪":11,"contentBase64":"cHJpbnQoMTIzKQo="}
```

示例内容是 11 字节的 `print(123)\n`。真实客户端必须先生成源码 bytes，再把 `len(source_bytes)` 作为 totalBytes；下方示例以代码计算，避免手写长度误差。

```
import base64
source = b"print(123)\n"
payload = {
    "path": "/hello.py", "encoding": "utf-8", "mode": "overwrite",
    "offset": 0, "totalBytes": len(source),
    "contentBase64": base64.b64encode(source).decode("ascii"),
}
```

`mode` 可为 `overwrite` / `create`，`encoding` 可为 `utf-8` / `binary`。offset/totalBytes 要么一起提供，要么一起省略。Base64 会把数据扩大到约 4/3，还要加上 path 和 JSON 结构；当前设备控制 payload 最大 2048 字节，不是可直接发送 2048 字节原始文件的容量。

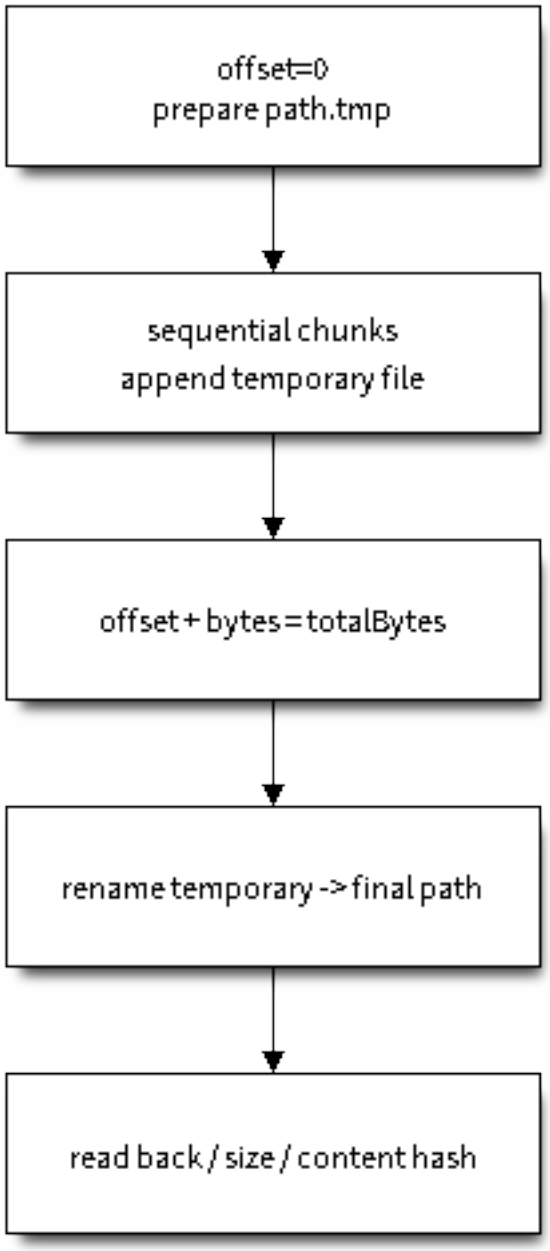


图 17：script.write 在最后一块前使用目标路径的.tmp 文件。首块会重新准备临时文件，后续块 append，最后一块提交 rename。未完成上传不应被主机报告为最终文件可用。

当前实现 ** 不提供通用 offset 随机写或重复块自动去重 **：后续非零 offset 实际执行 append，不应把任意 offset 解释为 seek。主机必须顺序串行发送、核对每个 ack 的 bytes/offset/totalBytes/complete；响应丢失后不要自动重复同一个 append 块，否则可能追加重复内容。

`complete:true` 表示本次上传发布路径已走完，不包含整文件的密码学 hash 证明。重要文件要通过读回或受支持的设备端 hash 再校验内容。`runAfterWrite` 只在 complete 后尝试把执行命令放入 REPL，不能证明应用启动或执行成功。

路径按 UTF-8 字节限制。当前 `MICROPY_ALLOC_PATH_MAX` 与 `.tmp` 后缀约束使脚本公开路径预算为 123 字节；还必须通过固件的允许字符检查。主机预检与最终固件检查都不能省略。

高级 framed REPL 辅助方法与公开读取接口

高级 `EvRepl.writeFile()` 辅助方法（不是 `device.files.write()`）默认每块写入 224 个原始字节，以 hex 编码构造 Python 写入行，编码后的命令仍须满足 stdin 预算。默认写入 `.viper.tmp`，关闭后移除旧目标再 rename；direct 模式直接写目标。这是客户端策略，与 RPC 的 `<path>.tmp` 不是同一个暂存约定，也不能被误写为通用跨掉电原子事务。

同一 REPL 上不应并行操作多个共享临时文件上传。取消或断线后，既要确认文件状态，也要清理仍开的目标端句柄/残留暂存；不能只清主机内存中的 promise。

读文件每次请求约 1 KiB 原始数据，目标端使用 hex 文本返还，并包含实际文件总大小与本块字节数。客户端验证总大小稳定、块长度与 hex 一致、是否过早 EOF，以及总期限；最后根据实际读回数据构造结果。一个短输出 burst 不能代表 EOF。

公开 `device.files.read()` 的默认总期限为 30 秒；IDE 可以为已实测的大文件/TCP 调度延迟使用有界的 60 秒预算。每个子请求仍需独立期限与取消检查；扩大总期限不等于允许无界阻塞或业务重放。

执行脚本不是上传 ACK 的延伸

```
{"seq":30,"channel":"user.rpc","type":"req","method":"script.run","encoding":"json
↪","sessionId":"89abcdef0123456789abcdef01234567","vmEpoch":1}

{"path":"/hello.py"}
```

```
{"ok":true,"path":"/hello.py","queued":true}
```

此响应只表示 `execfile(...)` 已排入 REPL 输入，不是 execution completed。应用还可能导入失败、缺模型、相机初始化失败、抛异常或进入无限循环。必须持续收包，等待应用级标记、真实预览或明确的共享执行生命周期结果。

应用使用 `device.python.execute({ path, completion: 'start' })` 进行交互运行，返回含 `done` Promise 的句柄。Core 内部委托给会话实现，应用不直接调用 `startScript()`；普通无限运行不应套用有界试运行的自动停止期限。Run/Stop UI 根据共同的执行状态更新，不能覆盖 REPL 内部完成回调以猜测结束。

有界测试使用超时、取消和停止确认；交互应用持续运行，保活与 reader 继续工作。文件上传期间再次 Run 应被协调器拒绝/排队，而不是混入另一个脚本命令。

会话 lease 与存储 writer lease 是两件事

USB MSC 的主机块设备写入与 MicroPython VFS 共享同一卷，不能因为取得 EV-MUX session 就推定它可安全并行写入。`device.control` 的 `storage.writer.status` / `storage.writer.acquire` / `storage.writer.release` 是独立的卷写入所有权接口。

它们涉及 volume（flash/sd）、operationId、writer leaseId 与当前 session，以及 available/held/handoff_ready/quarantined 等状态。交接时先关闭新 I/O admission，再等待进行中的操作/文件句柄 drain；异常条件可以隔离卷。主机应查询状态、执行其能力允许的安全交接；不能把 CDC 与 MSC 共存当作两个独立文件系统副本。

会话 token、writer leaseId 和网络 operationId 的字段格式虽可能相似，但作用不同，不能互换。writer 事务详细结构以 capabilities 与实际控制响应为准，不应把所有 file RPC 自动包装成已获得写权限。

文件系统就绪状态独立于写入所有权。状态响应包含 `fs_state`、`fs_error`、`fatfs_result` 和 `format_supported`，未挂载卷返回 `writable: false`。自动初始化要求对应板级策略开启且精确返回 `FR_NO_FILESYSTEM`，默认内部 Flash 开启、SD 关闭。需要保留未知数据时应关闭对应策略。其他失败应诊断并由用户决定恢复操作，`storage.retry()` 始终不格式化。统一的介质、备份和恢复规范见*存储、文件系统与数据保护*，显式 REPL API 见*storage – 存储诊断与确认恢复*。

9.2.7 预览、调试抓帧、背压与统计

preview 图像由应用主动发布

应用调用 `img.flush()` 后，preview producer 在需要发送时将当前图像编码为 JPEG，以 `preview.frame` 数据帧发给当前 user stream owner。只读 TCP 旁观者可独立订阅，详见下文。已绑定的 USB 摄像头输出也是独立消费者，详见*uvc —USB 摄像头输出*；它不接管 EV-MUX 控制会话，也不初始化相机。

UVC 使用 USB 等时传输。任何失败或短包都会终止当前帧，固件不会把后续 JPEG 片段继续拼到不完整的前缀上。它会尝试发送一次仅含 UVC 头的 `ERR|EOF` 结束包，将失败与完整帧传输分开报告，并释放源缓冲槽。下一帧切换 FID 并清除错误标志。总线复位或 alternate setting 取消也会释放旧源。这不保证 USB 无损交付；验收必须完整解码所有保留的 UVC 帧，并覆盖 CDC/TCP 并发场景，不能只检查输出文件非空。

```
import sensor
import time

sensor.reset()
try:
    sensor.set_pixformat(sensor.RGB565)
    sensor.set_framesize(sensor.QQVGA)
    while True:
        img = sensor.snapshot()
        img.flush()
        time.sleep_ms(200)
finally:
    sensor.shutdown()
```

无限循环由主机显式中断；`finally` 提供应用资源释放路径。这个设备端示例不自动保证帧一定到达或保持某个帧率，主机仍需验证完整 JPEG 与 session/epoch。

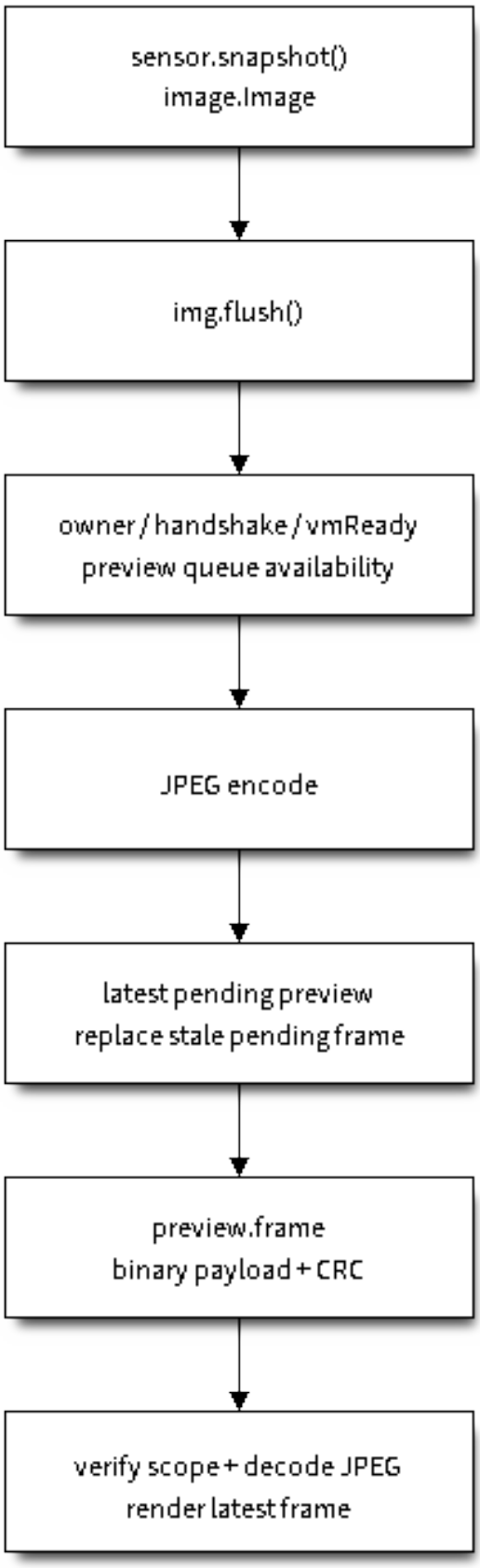

sensor.snapshot()
image.Image
img.flush()
owner / handshake / vmReady
preview queue availability
JPEG encode
latest pending preview
replace stale pending frame
preview.frame
binary payload + CRC
verify scope + decode JPEG
render latest frame

图 18：先判断发送资格和队列条件，再编码、提交图像。没有有效 owner、只读旁观者或已绑定 UVC 消费者时，不编码 preview；相机是否继续采集是用户程序自己的行为。

图像帧长什么样

以下为概念性 metadata；实际 seq、route、时间戳、sessionId、vmEpoch 均由当前发送快照产生：

```
{
  "sid": "preview", "seq": 40, "channel": "preview.frame",
  "type": "data", "encoding": "binary", "route": "usb",
  "contentType": "image/jpeg", "width": 160, "height": 120,
  "ts_ms": 123456,
  "sessionId": "89abcdef0123456789abcdef01234567", "vmEpoch": 1
}
```

payload 直接从 JPEG 的二进制字节开始，p 是编码后 JPEG 字节数，不是 width × height，也不是 source RGB565 帧缓冲大小。主机不应把 JPEG payload 交给 JSON/UTF-8 parser。

`seq` 可帮助观察帧序列，但丢帧不应按“缺哪一帧就补传哪一帧”理解。主机可保留最新完整图像用于渲染，旧 session/epoch 的图像必须丢弃，不能在软复位后闪回上一程序的画面。

Preview、摄像头与 LCD 三条图像路径

表 16: 明确区分图像来源

属性	`preview.frame`	`debug.capture_frame`	`debug.capture_screen`
触发	应用 `img.flush()`。	主机摄像头抓帧请求。	主机 LCD 抓屏请求。
来源	应用提交的 image。	应用最近一次采帧后发布的摄像头图像。	显示层最后提交的 RGB565 帧。
线上类型	独立 `type:data` JPEG。	原 RPC `type:rsp` 携带二进制 JPEG。	原 RPC `type:rsp` 携带二进制 JPEG。
执行	应用持有图像并提交 preview。	VM 队列读取独立且只读的采帧副本。	VM 队列读取显示缓冲并编码，不调用摄像头。
交付	允许丢帧，最新待发帧优先。	按请求 seq/channel/method、会话与 epoch 关联响应。	相同 RPC 关联规则，无图像时返回错误。

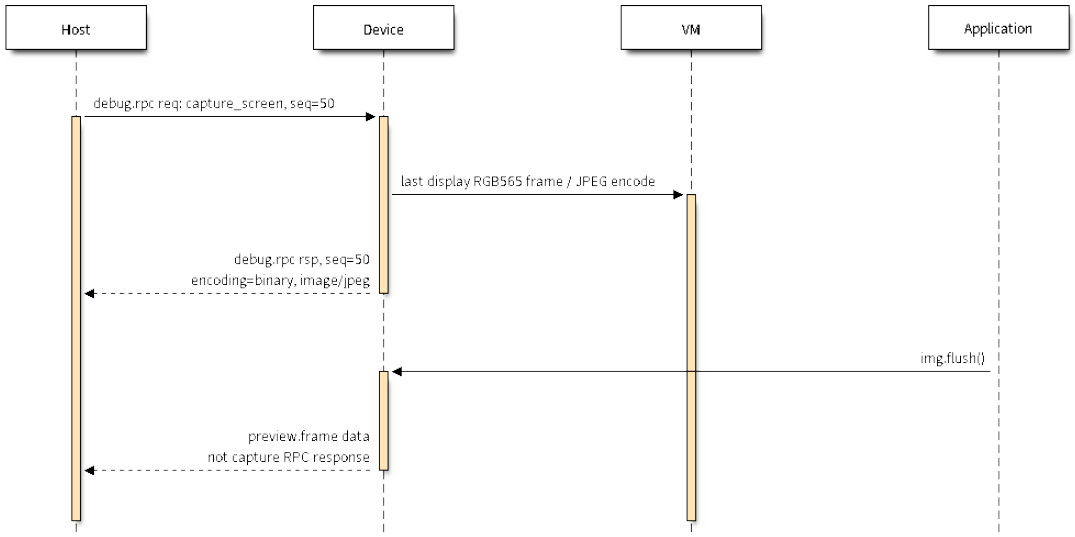


图 19：单次抓图由原 RPC 响应交付，不等待另一个图像 channel。 Core 对外统一为 `capture({ source: 'camera' | 'screen', format: 'jpeg' })`；两种来源不会相互回退。成功结果是 `{ data, mimeType, metadata }`，metadata 包含尺寸；失败仍可为 JSON error，客户端须先检查 encoding/contentType。

发送前，共享 JPEG 编码器先确认返回长度为正且不超过已分配缓冲区，并验证图像结构及其与源图像的尺寸一致性。摄像头抓图、LCD 抓屏和预览均经过这一入口。异常输出不会作为图像进入队列；抓图返回 `JPEG_FAILED`。Core 在 EV-MUX 长度、EOF、CRC 和 session/epoch 检查之后验证 JPEG 结构及元数据尺寸。坏响应仅使对应抓图立即返回 `INVALID_IMAGE`；坏预览记录诊断后整帧丢弃，不进入图像回调。

图像检查覆盖自包含的 8 位 Huffman JPEG 头、有界标记段与编码表引用、扫描边界及无尾随数据的最终 EOI，不对熵编码像素执行完整解码。完整解码验证仍属于真机验收。链路中途断开仍可能留下部分字节；接收端必须等待完整、校验通过的 EV-MUX 帧，丢弃不完整输入，不能直接显示。

摄像头抓图是被动观察，要求固件声明 `debug.capture_frame.observed.v1` 能力。它编码应用最近一次成功采帧后发布的独立副本，包括 `sensor.snapshot(buffer)` 使用调用方缓存的路径，以及 `sensor.skip_frames()` 采到的帧。它不会初始化传感器、额外物理采帧、写入应用持有的 `Image`，也不会在采帧返回后继续借用调用方缓存。应用后续的图像绘制应通过 `img.flush()` 预览或 LCD 缓存观察，这个来源保持采帧时的原始像素。Device Core 0.6 会拒绝向缺少此能力的旧固件发送不安全的摄像头 RPC；只升级 Core 并不会让旧固件具备安全抓图行为。

没有有效发布帧时返回 `CAMERA_FRAME_NOT_READY`：包括应用首次采帧前、采帧失败后、摄像头重配/关闭/重置或 VM 软重置后。请求不会启动空闲摄像头。输出尺寸、像素格式、镜像或翻转实际变化时，在改变硬件之前使旧帧失效；重配失败也可能已经改变硬件，不能继续返回旧帧。同值设置保留当前帧。IPA 模式实际变化也会在控制器停流/重启之前使旧帧失效，重启失败也不保留旧帧；同值模式保留当前帧。手动 ISP 写入在访问硬件前保守地使旧帧失效，包括部分失败，即使请求值相同也如此。这些变化同样会释放观察副本的分配失败状态。应用下一次成功采帧会发布带有新尺寸/格式的图像。摄像头观察与 LCD 抓图不会相互替代。

摄像头发布、状态修改与 JPEG 编码均在主 VM 任务执行。Python 工作任务修改摄像头时返回 `OSError`，不会与采帧、编码或释放并发。transport task 只排队请求，不持有摄像头像素。排队请求可因忙或超时返回 `VM_BUSY` / `VM_TIMEOUT`；取消、所有者丢失或 epoch 变化可丢弃响应，但不会采帧或改写应用缓存。VM 软重置先停止接收请求，再释放两份摄像头缓存；硬件重置后两份缓存均为空。

LCD 抓屏不初始化屏幕、不重绘、不启动摄像头，只读取最后有效 RGB565 显示帧。没有 LCD 返回 `NOT_SUPPORTED`，未初始化或没有有效显示帧返回 `DISPLAY_NOT_READY`。读取及编码在 VM 任务中与显示操作、软复位清理串行执行，VM 忙时也可能返回 `VM_BUSY` / `VM_TIMEOUT`。可选 JPEG `quality` 为 1–100。它反映软件显示帧而非物理面板回读，不能证明背光或屏幕硬件正常。

使用统一 Core 真机 CLI 的 `--screen-output` 显示测试图案并保存 JPEG；具体命令与验收要求见通信验收、闭环测试与故障定位。

队列优先级与丢帧

控制、会话、REPL、日志、preview 对应不同 TX 类。普通小帧队列预算有界，控制流量有预留预算；不能用不读取主机数据的方式验证“所有输出永不丢失”。

preview 只保留最新待发帧，可替换尚未开始发送的旧图。已经发送到一半的字节不能撤回，也不能把两张 JPEG 的尾巴拼成一张。preview 与大图 debug response 共用有限特殊缓冲：最多三个按需分配的 256 KiB 缓冲，不代表永远可容纳三个任意大图或允许 256 KiB 的主机命令。



图 20：降低积压时延比交付每一张预览更重要。若需要保存全部图像，应使用设备端存储或专门的视频/记录接口，并独立设计其容量、回放与验收；preview 不承担这个保证。

UART 的帧期限随 baud 与总字节数计算；USB/TCP 也有有界等待。底层 write 可只接受一部分字节或返回 WOULD_BLOCK，TX task 必须保留未发送后缀并在期限内继续。帧发送失败或 socket 错误可能使 TCP link 关闭，而 RX 半帧超时只重置仍在线连接的 parser，不应一概认为设备重启。

日志通道与系统日志 UI

IDE 可以将 `log.idf`、RPC 诊断和连接事件合并显示为“系统日志”，但这个 UI 面板不是 firmware 的第九个 channel。`repl.stdout` 则主要显示应用/REPL 文本，channel 的真实来源与限制见架构、执行上下文与标准输出。

系统日志的格式、颜色、过滤器可以由主机选择，不能反过来改变固件所有权和物理路由。无 owner 的默认/回退日志、启动 console 原始文本与当前 session 业务输出，应在诊断记录中分开标识。

如何看待统计计数

`debug.info` 的 `scope:transport.stats` 与 `transport.state` 可在用户 Python 忙时查询，用于观察队列、RX 拒绝、发送超时、preview 丢弃和后端状态。必须区分 endpoint/link 统计、全局统计、业务完成标记：

- 未读的 USJ 默认输出积压可增加全局 drop；这不等于活动 TCP 上的特定 RPC 已失败。
- 某个 RPC 已拿到完整关联响应，不代表整段应用业务完成。
- preview 丢弃是允许的实时行为，不自动否定文件传输，但错误 owner/epoch 的图像被接受属于协议错误。
- reset/generation 变化可能改变统计范围；应把采样时刻、端点和会话身份与结果一起记录。

诊断时保持有界，只记录必要的计数与状态；不要在错误路径中再制造大量打印以填满同一 TX 队列。

帧缓冲的生命周期

LCD 缓存在显示初始化时分配，反初始化时释放。SPI 彩色传输完成后绘图函数才返回，后续绘图不会覆盖仍被 DMA 借用的缓存。应用默认 snapshot 缓存在首次使用时分配，后续应用采帧可复用同一缓存；显式传入的缓存仍归调用方所有。应用若要跨自己的下一次采帧或摄像头重配保留图像，仍需自行复制。

安全的摄像头观察持有独立原始帧副本，在应用首次成功采帧后按需分配，尺寸稳定时复用。例如 QVGA RGB565 会在应用 snapshot 缓存之外增加 153,600 字节；开机时不预占缓存，每次主机请求也不会新分配一帧。副本优先使用 PSRAM，失败时回退到支持 8 位访问的内部堆；配置启用不能证明 PSRAM 初始化成功。各板后端先完成 DMA/PPA 与缓存同步，再复制 CPU 可读的结果；观察副本不会直接交给摄像头 DMA。

观察副本分配失败时，应用采帧仍成功、像素保持不变，之后摄像头观察持续返回 `NO_MEM`，直到释放此分配失败状态；即使后续某次采帧失败，也不会每帧反复尝试分配。摄像头重配/关闭/重置或 VM 软重置会释放这个状态，使应用后续采帧可重新尝试分配。实际重配在修改硬件前释放观察副本，VM 清理

也释放默认应用 snapshot 缓存。固件借用的观察视图只在本次 VM 调用内有效；Core 在主机端返回独立接收的 JPEG。驱动 DMA、PPA、JPEG 和传输缓存有各自生命周期，均需计入图像功能的总内存预算。

TCP 只读预览

预览客户端连接同一个 TCP 端口（默认 3232），在 3000 ms 握手期限内发送 `user.rpc/preview.subscribe`，payload 为 `{}`，无需先接管控制会话。成功响应为 `{"ok":true,"subscribed":true,"channel":"preview.frame","maxObservers":2}`。发送 `preview.unsubscribe` 和 `{}` 可取消订阅。两者均幂等，不携带 `sessionId` 或 `vmEpoch`。capabilities 发布 `preview.observers.v1`、`preview.subscribe` 和 `preview.unsubscribe`。

最多两个旁观者可与独占的 USB、USJ、UART 或 TCP 控制者并存。第三个旁观者收到 `OBSERVER_LIMIT`，三个 TCP 槽中保留一个给控制连接。旁观者连接的 `hello` 返回 `READ_ONLY`；Python、文件、REPL 和设备控制仍须通过 owner 授权。需要控制时使用另一条连接。旁观者只收到主动发布的图像和自身请求的响应，预览 metadata 不含控制者的会话 ID 或执行身份；其中 `route` 保留当前 user stream 的路由值，旁观者 socket 独立传输。这种角色隔离不增加网络身份认证。

应用通过 `img.flush()` 发布处理后的图像，连接本身不会启动 sensor。复用控制预览和已绑定 UVC 的 JPEG 准备过程，不增加 HTTP 服务或重复 JPEG 编码。每次发布只为所有旁观者复制一份不可变 JPEG、计算一次 CRC，每个连接独立维护发送位置。TCP 通过 scatter/gather `sendmsg` 聚合帧头、metadata、JPEG 和帧尾，避免每个片段单独 send。TCP 仍须向每个旁观者发送数据，因此网络带宽随人数增加；既有 owner 预览池保持独立。

RX 解析器按实际端点槽位静态分配，避免扩大 TCP 容量时重复为串口分配闲置解析器。TCP 发送时临时内存不足按背压处理，仍受同一有界期限约束。socket 关闭操作在共享 I/O 锁外进行，并使用短发送超时限制 lwIP 分配 FIN 时的重试等待。`debug.info` 的 `scope:transport.stats` 提供 `lastTxTimeout`，记录最近一次超时的端点、槽位、发送位置、总字节数、经过时间和无进展时间。

旁观者共用最多三个延迟分配、持续复用的 256 KiB PSRAM 槽（上限 768 KiB），每个旁观者有两个小型帧描述符。尚未开始发送的帧可替换为最新帧；只读帧总期限为 8000 ms，无进展期限为 4000 ms，分别通过 `tcp.observerFrameTimeoutMs` 和 `tcp.observerIdleTimeoutMs` 上报。每次发送进展刷新空闲计时，替换首字节尚未发出的帧不会刷新任一期限。已开始的帧必须在这些预算内完成，否则关闭该 socket。慢旁观者不会占住控制者或另一旁观者的队列。link generation 校验防止旧帧及旧订阅串入重连后的 socket。订阅期间无需控制租约或 keepalive，空闲时可保持连接；取消订阅后恢复普通空闲回收。

主机脚本仅使用 Python 标准库和内置有界 EV-MUX 解析器；`--view` 还需要 PATH 上存在 `ffplay`，其显示队列仅保留最新待显示图像。传输解析始终保留帧边界并校验 JPEG CRC。

```
python tools/preview_tcp.py 192.168.1.100 --view
python tools/preview_tcp.py 192.168.1.100 --duration 10 --save-last preview.jpg
```

可显式运行 `tools/test_preview_tcp.py --host <设备IP> --port <串口> --device-cli <已安装插件>/cli/device-cli.mjs --output-dir <项目内目录>` 验收；加入 `--video <V4L2 节点>` 可检查 UVC 并发。测试会替换当前 Python 应用，运行有限时长测试程序，使用 Pillow 解码收到的 JPEG，结束后恢复 UVC logo。

9.2.8 主机接入、Wi-Fi 配置与恢复实践

首次 Wi-Fi 配网与无线连接

准备一块支持 Wi-Fi、已安装兼容 ESP-VISION 固件的开发板，用 USB 数据线连接电脑。电脑与开发板需要接入可互通的局域网。支持情况见*局域网发现协议与板级默认策略* 中的板卡说明；首次安装固件见*用 AI 开发视觉应用*。

使用官网配网 也可以直接在 ESP-VISION 官网 配网，无需安装本地主机工具。

1. 用 USB 连接开发板，保持 ESP-VISION 固件正常运行。在支持 Web Serial 的桌面浏览器（如 Chrome 或 Edge）中打开官网，找到“浏览器烧录入口”。
2. 在“烧录方式”中选择“仅配置 WiFi”，点击“连接串口”，在浏览器弹窗中选择开发板的运行串口。
3. 在“设备 Wi-Fi 配网”中填写 WiFi 名称（SSID）和密码，点击“保存”。
4. 等待页面显示“WiFi 已连接”和 TCP 地址；配置会保存在开发板上，重新上电后自动连接。需要再次确认时，点击“读取联网状态”。
5. 点击“断开”释放官网的设备连接，再让 AI 助手或其他主机工具通过局域网连接开发板。

“仅配置 WiFi”只更新网络配置，无需重新烧录固件。已有兼容固件时可以直接使用此入口；首次安装固件后，也可以在官网的后续配网页面完成配置。

使用 AI 助手配网 安装 ESP-VISION 插件后，可以直接提出需求：

备注： 帮我把开发板连上 Wi-Fi，之后通过无线连接修改程序、查看摄像头画面。

---

1. 保持 USB 连接，告诉助手要连接的 Wi-Fi 名称，并按提示完成密码输入。
2. 等待助手确认配网成功，并给出设备的局域网地址。首次连接需要等待获取 IP 和保存配置完成。
3. 通过无线连接继续提出应用需求，例如“让摄像头跟踪红色物体，我想在电脑上看到效果”。
4. 无线连接验证成功后，可按开发板要求改用独立供电。设备再次上电时会使用已保存的配置自动联网。

使用 CLI 配网 手动操作时，先按设备管理命令行 安装 Device CLI 及串口依赖。以下命令中的 `COM3`、`MyWiFi` 和 `192.168.1.100` 是示例值，分别替换为枚举到的串口、实际网络名称和设备返回的 IP 地址。

1. 找到通过 USB 连接的开发板：

```
esp-vision-device inspect --interface serial
```

2. 配置 Wi-Fi 并启用网络服务；在终端提示中输入密码，输入内容不会显示：

```
esp-vision-device configure --port COM3 --ssid "MyWiFi" --network-enabled true␣
↪--timeout 60000
```

命令会等待网络配置事务完成，这里将等待预算设为 60 秒。成功后返回 `ok: true`；取得 IP 并保存配置后才算配网成功。非交互输入方式见 CLI 参考中的 `--password-stdin`。

3. 通过 USB 确认设备的网络状态与地址：

```
esp-vision-device inspect --port COM3 --fields identity,network
```

检查 `network` 中的 `enabled`、`configured` 均为 `true`，`state` 为 `online`，并记录 `ipv4`。如果命令超时或失败，先查询此状态及 `operation` / `lastOperation` 中的错误，再决定是否重新配网。

4. 发现局域网设备，并用返回的 IP 验证无线连接：

```
esp-vision-device inspect --interface tcp
esp-vision-device inspect --tcp 192.168.1.100:3232 --fields identity,network
```

对照 USB 查询结果确认是同一设备。连接成功后，CLI 的文件管理、Python 脚本执行和图像采集均可使用 `--tcp`；连续图像预览见预览、调试抓帧、背压与统计。无线开发期间仍需为开发板供电，固件烧录仍使用 USB 串口。

配网失败或发现不到设备

- 密码错误或找不到网络：核对 Wi-Fi 名称、密码、信号及开发板支持的频段；通过 USB 重新执行配网命令。
- 已连接但获取 IP 失败：检查路由器的 DHCP 服务，再通过 USB 查询网络状态。

- 设备有 IP 但列表中找不到：先尝试通过该 IP 直接连接；检查电脑和开发板是否处于可互通的局域网，以及访客网络隔离、防火墙或 mDNS 过滤。
- 无线地址变化或连接中断：重新发现设备；仍无法接入时，保留 USB 供电和连接，用上面的网络状态命令排查。

建议的主机栈

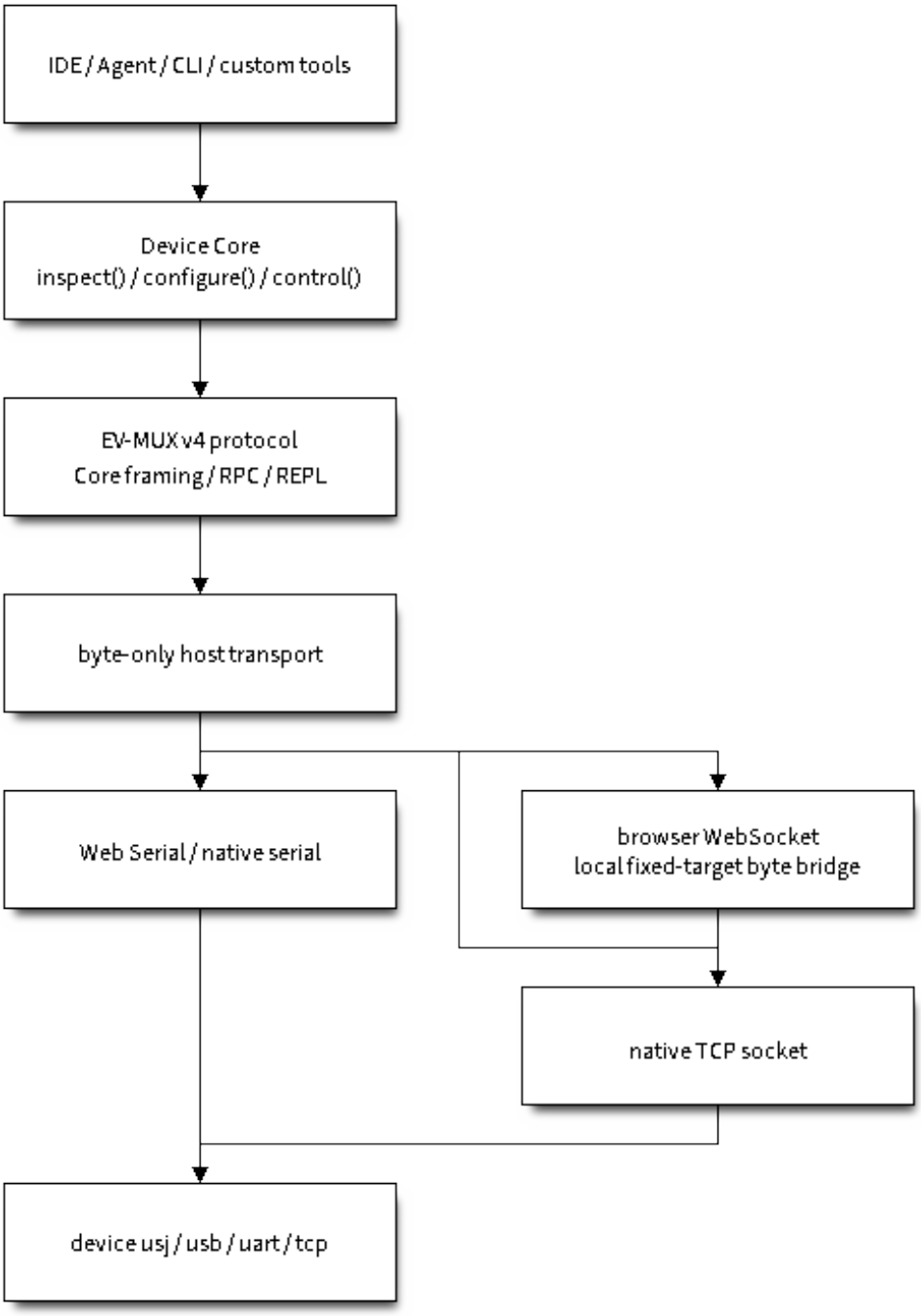


图 21：共享 Core 持有协议状态，transport 只传字节。网络桥接不再次 acquire、重编码帧或执行 Python；否则会出现两个会话 owner 和两份请求关联逻辑。

`@esp-vision/device-core` 在 `host/device-core` 维护，提供唯一共享协议实现。IDE、独立 Agent 和固件真机检查均使用 Core；Agent 工作流、模型提供方和 MCP 插件属于 Agent 主机应用，不属于 Core 或固件通信依赖。应用通过 `createDevice()` 获取设备接口，parser、framed REPL 与会话保活是内部实现，不需要主机再次组装。

完整 Node TCP、浏览器 Web Serial 和高层 API 示例见*Device Core 接入指南*。真机测试统一运行包的 `npm run test:hardware`，串口适配器仅转发字节；它不是另一套 Python EV-MUX 客户端。摄像头与 LCD 检查选择见通信验收、闭环测试与故障定位。

此适配器细节图展开统一的方案架构：主机应用 → Device Core → EV-MUX → 设备固件。Core 持有协议/会话状态，原生串口、Web Serial、TCP 与 WebSocket 桥只传字节。DNS-SD 发现位于连接前，ROM 烧写另走 loader，不是 EV-MUX 镜像传输。

接入顺序

1. 选中物理端点，核实设备/USB 拓扑，打开 transport。
2. 串口仅在首次请求前进行一次有界启动排空；TCP 立即 acquire。
3. 启动持续 reader，以新 nonce 的 hello 接管，验证 evMuxVersion 和响应关联，再保存 session。
4. 查询 capabilities 与 firmware identity，协商命令/REPL 大小和 timeout。
5. 按设备报告间隔启动保活；reader 在所有业务期间持续运行。
6. 等当前 epoch 的 vmReady，并在要执行 VM 工作时确认 REPL 可执行。
7. 串行协调文件、运行、停止、复位；接收日志和 preview。
8. 关闭前尽量释放 session，再关闭 transport、reader 和 pending requests。

串口启动排空为什么只做一次

USB 串口打开可能读到上一次连接的完整帧、残帧、旧 traceback 和 IDF 启动文字；“打开串口”不会触发新的 discovery hello。`serial.flush()` 通常等待的是主机 TX，不是读取/清除设备响应。

Core 在连接开始时，通过同一个 parser 接收并处理目标 300 ms 的启动数据；这一步不建立业务会话。可能未完成的 header/body 跨越该窗口时，Core 会在 parser 剩余的进度和整帧预算内继续接收（两项默认均为 12 秒），持续拒绝旧帧，直到到达完整帧边界才发送 hello。额外等待开始时固定总截止时间，后续新帧不能延长它。取消、断线或接收预算到期时，连接失败并清理；parser 因超时丢弃半帧不算到达完整边界，Core 也不会通过清空或截断缓冲区制造成功。

排空只减少已读到的旧数据，不保证全部设备/USB/主机缓冲已空，也不诊断 USB completion 错误。后续请求绝不能每次清输入，否则会丢弃保活、preview 或业务响应。

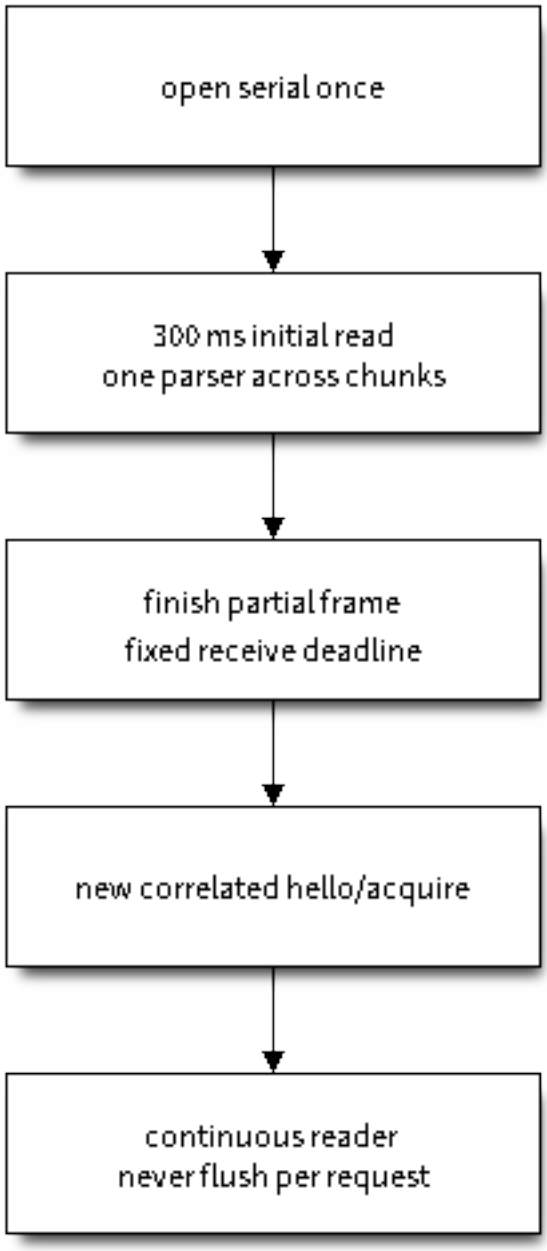


图 22：排空是接入准备，不是完成关联的替代品。后续仍需拒绝旧 seq/session/epoch。

统一真机入口

在仓库根目录使用 Node.js 22 运行；串口模式使用 npm 原生绑定，串口和 TCP 均无需主机 Python。下例先验证已联网设备的身份和摄像头，再对 USB 设备运行完整检查：

```
npm --prefix host/device-core run test:hardware -- --tcp <device-ipv4>:3232 --
↪checks identity,camera
npm --prefix host/device-core run test:hardware -- --serial <selected-port> --role␣
↪usj --checks all --timeout 600000
```

测试会停止现有 Python 程序，并按所选检查初始化摄像头或显示测试图案。为设备安排独占测试时段，先关闭占用它的 IDE/终端；不会通过切换接口绕过有效 owner。详细检查集、LCD JPEG 输出和验收证据见通信验收、闭环测试与故障定位。

Wi-Fi 与 NVS 的真实状态机

第一次配网需要任一已取得会话的连接，通常使用 USB/USJ/UART；没有已配置网络时不能依赖这个 TCP server 做完全无线首次配网。Native/Hosted Wi-Fi 与 Python WLAN 共享底层实例，后台网络服务不会创建第二套 WLAN 驱动。

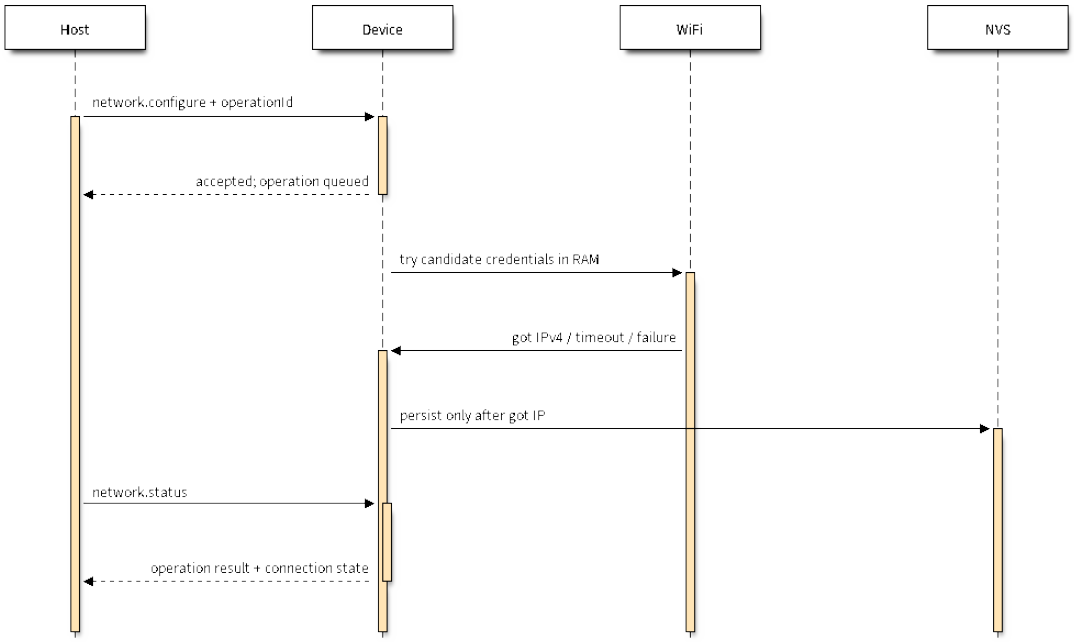


图 23：accepted 与连接成功、保存成功分别确认。网络修改在 accepted 回复 TX 完成后执行，最多等待约 1000 ms；configure 候选在 RAM 中尝试，约 30 秒内取得 IP 后才提交记录，失败恢复上一份保存的配置。

```
{
  "action": "network.configure",
  "operationId": "fedcba9876543210fedcba9876543210",
  "ssid": "<wifi-ssid>",
  "password": "<wifi-password>"
}
```

通过 `user.rpc/device.control` 发送，metadata 携带当前 sessionId。网络控制不依赖 VM 空闲。`network.status` 查询最终状态，`network.enable` 控制保存的启用状态，`network.forget` 保存忘记配置状态；不能将它们视为 Python `WLAN` 方法名直接塞进 RPC。

修改事务使用新的 32 字符 operationId；只在恢复同一事务时复用相同 ID 和完全相同 payload。事件 `network.changed` 可提示变化，但 status 是有界关联查询；重启/断线后不凭旧事件推断最新事务结果。

已有 NVS 记录优先，包括 disabled、forgotten 空记录和损坏记录；出厂宏不会覆盖它们。出厂默认仅在完全无记录时先保存再尝试连接；RPC 用户配置则先取得 IP 后保存，这是两种不同来源的流程。

后台事务期间，Python WLAN 修改会返回 EBUSY；普通 Python WLAN 修改可临时影响运行状态，但不隐式改写后台持久记录。关闭后台开关仍可显式使用 Python WLAN，不过 TCP 后台服务和其配置策略也随服务边界变化。

公开示例使用占位凭据，主机日志与诊断应脱敏。TCP 本版没有认证/加密，配置操作按其受信任开发网络的实现边界使用。

浏览器网络接入

浏览器不能直接打开任意 TCP socket。IDE 的固定目标本机 WebSocket-to-TCP bridge 将二进制字节送到设备 TCP；它只允许显式 origin、固定设备地址、单个活动浏览器，并在浏览器断开时关闭 TCP。它本身不解释 EV-MUX 或 acquire。

主机桥接程序位于独立 IDE 仓库，不能在固件根目录直接假定有相同 npm 脚本：

```
npm run network:bridge -- --device <device-ipv4>
```

本地 IDE 通常使用 `http://127.0.0.1:8766/` 与 `ws://127.0.0.1:8766/device`。远端 HTTPS 页面需显式允许精确 origin，并满足浏览器本地网络访问权限/安全策略；需要 TLS 时使用受信任的 WSS 代理。USB Web Serial 与网络桥共用 `createDevice` 和字节传输契约。

复位与取消的确认

软复位：等待同 owner 的新 vmEpoch、vmReady 和 REPL barrier；异步 event/hello 有独立观察预算，缺事件本身不是复位失败。有事件本身也不证明 REPL 已可执行。

硬复位/ROM：废弃旧会话与请求；等物理枚举/固件恢复，重新接管并核对 firmware identity。主机烧写器的验证/复位状态应分开记录，已完整读回验证但未观察到运行态恢复，不应自动重复写镜像。

取消：停止继续上传/执行，尽可能中断目标工作并确认目标状态；若写入或执行是否完成不可知，保留“未知”结论。网络事务可按其 operationId 关联；普通业务 RPC、REPL 输入和文件 append 没有通用自动重放。

主机超时预算不能共用一个数字

- discovery/capabilities：参考客户端的关联 RPC 默认预算约 6 秒。
- keepalive：单次约 3 秒，同时满足固件 lease 总预算。
- 串口 startup drain：目标 300 ms，首次连接一次；跨窗口的半帧可在剩余接收预算内补齐，但不能延长固定的总截止时间。
- 主机 RX 拼帧：参考实现约 12 秒预算，允许 TCP 收包出现数秒间隔。
- 设备 RX：空闲/整帧约 3/10 秒，与主机收大图预算不是同一个方向。
- VM 排队：设备约 5 秒过期，增加主机等待不扩大设备队列。
- 文件 readFile：Core 默认总期限 30 秒；IDE 实测场景使用有界 60 秒。
- Wi-Fi 配置：事务尝试约 30 秒；accepted 回复只是开始，须另查询最终状态。

这些是当前默认/参考实现值，运行前以实际 capabilities 和所用客户端选项为准。只对具有明确幂等恢复语义的操作设计有界重试；不要因网络慢就无限扩大等待或自动执行第二次应用命令。

9.2.9 局域网发现协议与板级默认策略

备注： 实现通过 DNS-SD 公布 ESP-VISION 服务，并为 Device Core 提供 Node 和浏览器网络桥发现适配器。固件构建/真机资格验证仍是独立发布关卡。现有 hello discovery 只在 transport 已连接后工作。

轻量协议选型

首版采用 mDNS/DNS-SD，无需中心服务器，复用 IDF 5.5、6.0、6.1 manifest 已有的 `espressif/mdns: ~1.1.0` 依赖。DNS-SD 用 PTR 枚举服务，SRV 提供主机与端口，TXT 提供属性，A 记录提供当前 IPv4 地址。

表 17: 实现取舍

协议	成本与收益	结论
mDNS/DNS-SD	已有组件及主机工具；缓存和命名冲突处理有标准约定。	优先采用；注册一个服务，并限制 TXT 属性规模。
自定义 UDP 查询/应答	专用应答器可能更小，但需新增端口、解析器，以及重试、缓存、多网卡规则。	仅在实测资源不足时考虑；首版不同时维护两套发现协议。
SSDP/UPnP	引入另一套发现格式，完整 UPnP 接入还涉及设备描述。	对已有 EV-MUX 服务没有明确优势。

固定的 mDNS 1.1.0 Kconfig 默认任务栈为 4096 字节，这不是全部堆内存占用。复用已有单实例，不新增发现任务/socket。需对比服务注册前后的镜像大小、内部堆剩余量/最低值和任务栈水位；目前尚无实测 RAM 增量。避免应用层持续发送心跳广播，复用组件的查询、公布和 TTL 行为。

服务约定

- 在本地链路通过 mDNS UDP 5353 查询 `_esp-vision._tcp.local`。
- 主机名：`esp-vision-<完整开发板名>-<MAC 后 6 位十六进制>-<固件版本>.local`，板名/版本转换为小写 DNS 标签并用连字符分隔。版本来自固件构建，绝不使用 IDE 版本。标签不超过 63 字节；长版本保留前缀并增加确定性的八位构建摘要，TXT 保留原始版本。所有板卡均使用主芯片 MAC。USB、USJ、UART、TCP 的 hello 事件/应答、capabilities 和设备 debug 信息返回相同的 hostname 标签（不含 `.local`）及完整设备 ID。
- 实例显示名称为板型加标识后缀。SRV 公布实际 `ESP_VISION_TCP_PORT`（默认 3232），不在 TXT 重复端口。
- TXT 键：`txtvers=1`、`id=<device-id>`、`board=<board>`、`fw=<firmware-version>`、`evmux=4`、`hostname=<统一主机名>`。编码后的完整 TXT 限制为 256 字节，注册前检查长度。
- 复用 Launcher 从主芯片 eFuse MAC 派生的设备标识，Hosted Wi-Fi 板卡也遵循此规则。不能替换为无线协处理器 MAC、IP、boot ID 或 session ID。稳定标识不可用时报告发现不可用，不生成随机永久身份。
- 不公布密码、会话凭据或周期刷新的 owner 状态。发现属性不可信，不构成身份认证。

固件生命周期与 Python 共存

平台网络服务统一负责 mDNS 初始化，并协调当前会初始化 mDNS、设置 MicroPython hostname 的 `overlay/micropython/ports/esp32/network_wlan.c`。不保留两个独立初始化标记。Python mDNS 查询复用共享实例。`network.hostname()` 返回同一个统一名称；设置其他名称会抛出 ValueError，保证固件只有一个定义明确的身份。DHCP 和 Python port 也使用此名称，软重启不会重新生成。ESP-IDF 网络接口的主机名上限对齐 DNS 的 63 字节；DHCP 选项空间设置为 255 字节，以同时容纳完整标签和其他选项。

仅在 STA 获得 IPv4 且 EV-MUX TCP listener 成功绑定后注册服务。listener 关闭、Wi-Fi 禁用或 STA 断开时撤销；重连后公布新地址。意外断电依赖主机 TTL 过期。软重启不能重复注册服务或停止平台发现应答器。mDNS 失败需可观测，但不能中断正常 TCP/USB 访问。共享网络 sdkconfig 将 mDNS 限制在 STA 接口，与 TCP 绑定地址一致；AP/Ethernet 接口不公布此服务。

主机与 Device Core 对接

Device Core 统一发现结果约定及取消/超时语义，通过注入环境适配器执行网络枚举。Node 主机可直接使用 DNS-SD 适配器。普通网页无法枚举 multicast DNS，浏览器 IDE 通过本地 Node 网络桥获取结果。这不引入 Core 对 Agent 的依赖。

结果包括稳定设备 ID、显示名称、板型、固件、EV-MUX 版本、hostname、IPv4 地址、TCP 端口和网卡标识。按稳定 ID 聚合，同时保留不同网卡/地址候选；删除过期地址，保留有效备选。hostname 不能单独作为设备身份。查询有界，支持 AbortSignal，取消时释放枚举资源。

枚举阶段不逐台连接 TCP，也不申请所有权。用户选择后，Core 连接并发送不申请所有权的 hello，校验 v4、稳定设备身份及真实占用状态，再明确申请会话。申请及不申请所有权的 hello 应答均包含稳定的 `deviceId`、`hostname`。传入 `expectedDevice` 后，`DeviceSession.negotiate` 会先发送不申请所有权的 hello 核对这些字段，再申请会话。会话占用时提示用户，接管继续遵循现有显式策略。

网络桥只负责枚举和字节转发，EV-MUX 仍由 Core 处理。发现 HTTP/控制接口与字节桥采用相同的精确 Origin 和 loopback 策略。通过网络桥生成的不透明句柄选择已解析候选，并重新校验缓存地址/端口；保留显式固定目标的手动模式，不暴露浏览器任意指定 TCP 目标的转发能力。

IDE 在现有网络连接区域增加刷新和发现设备选择。保留手动 IP/hostname 输入，以应对 multicast 过滤、客户端隔离或不同 VLAN。本地网络访问权限本身不提供 DNS-SD 枚举能力。

Wi-Fi 板卡默认启用策略

带 Wi-Fi 的板卡默认启用 WLAN、`ESP_VISION_ENABLE_BACKGROUND_NETWORK=1` 和 `ESP_VISION_ENABLE_TCP_TRANSPORT=1`。当前七种 Wi-Fi 配置已满足：`ESP32_P4X_EYE` 和 `ESP32_P4X_FUNCTION_EV_BOARD` 使用 Hosted Wi-Fi；`ESP32_S3_EYE`、`AtomS3R-M12`、`ESP32_S31_CHATBOT`、`ESP32_S31_KORVO` 和 `ESP32_S31_MOSAICO` 使用原生 Wi-Fi。两种 Hosted 板卡均包含 `sdkconfig.p4_wifi_common`。

`ESP32_P4X_VISION` 当前关闭 WLAN，且没有 Hosted Wi-Fi 配置。P4 本身没有原生 Wi-Fi，应先确认外接无线模组和板级连线，再启用该 port。`TEMPLATE` 也依赖具体硬件。不能直接复制其他板卡的 Hosted SDIO/reset 引脚。

默认启用表示固件服务在启动时可用，无需 Python 脚本。出厂 SSID 为空时保持未配网，有效保存配置自动重连，保存的禁用/忘记意图仍被保留。TCP 仅在网络条件满足时监听。发现功能跟随 TCP listener 生命周期，采用同样的默认策略。

实现与验收

1. 固件：统一 mDNS 生命周期，注册/撤销服务，在不申请所有权的 hello 中提供稳定身份。
2. Core：定义适配器/结果、取消语义及选择时身份核对；Node 和浏览器发现都与 EV-MUX 解析器保持职责边界。
3. IDE：扩展本地网络桥和连接面板，保留手动地址。
4. 验证多设备、不抢占已占用会话、主机多网卡、DHCP 地址变化、禁用/再启用、意外断电/TTL、软重启、取消发现及 multicast 受限时的手动回退。在 CI 运行完整固件矩阵，并在原生及 Hosted 真机测量资源增量。

Device Core 发现示例

Node 安装可选 multicast-dns 适配器依赖；浏览器使用已有本地网络桥。通过 `npm run network:bridge` 启动（无需固定 IP），再在 IDE 网络面板刷新/选择设备。候选句柄最多 30 秒过期，选择过期后重连须先刷新。

```
npm install @esp-vision/device-core multicast-dns
```

```
import { createNodeDiscovery } from '@esp-vision/device-core/discovery/node';
const devices = await createNodeDiscovery().list({ timeout: 3000 });
// Optional: createNodeDiscovery({ interfaceAddress: '<local-interface-ipv4>' })
```

```
import { createBridgeDiscovery } from '@esp-vision/device-core/discovery';
const devices = await createBridgeDiscovery('ws://127.0.0.1:8766/device').list();
// Connect the chosen connectionUrl, then:
// await session.negotiate({ role: 'tcp', expectedDevice: devices[0] });
```

协议参考

- Multicast DNS，RFC 6762
- DNS-Based Service Discovery，RFC 6763
- Espressif mDNS 1.1.0 Kconfig

9.2.10 Device Core 接入指南

Device Core 文档统一发布在本文档中心。SDK 概述见*Device Core 主机 SDK*，完整应用 API 见*应用 API 完整参考*，高级导出见*高级导出 API 完整参考*，可运行的 TCP/Web Serial 适配示例见*Device Core npm 包对接*。

9.2.11 通信验收、闭环测试与故障定位

验收证据的四个层级

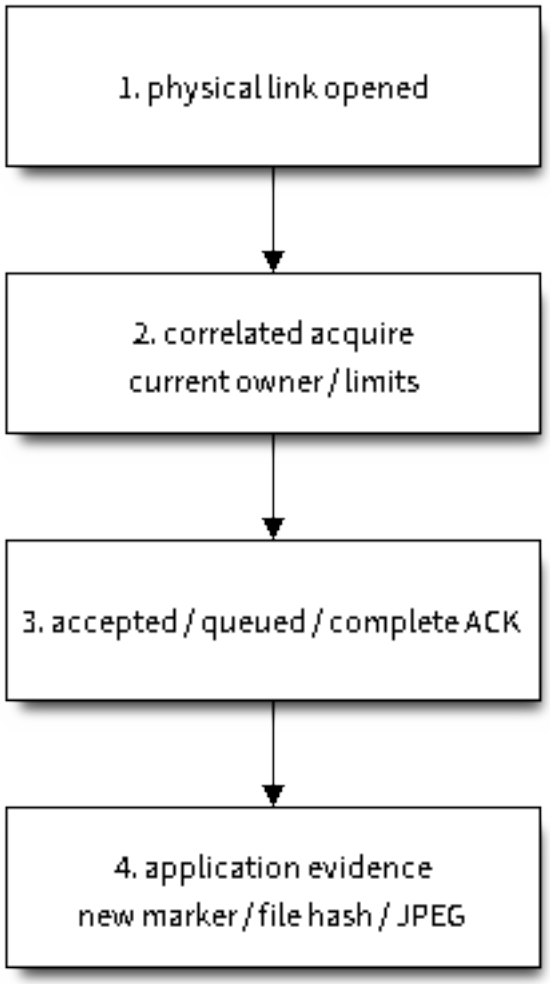


图 24：下一级成功不能替代上一级的应用证据。打开串口、接管成功、上传 ACK 与摄像头真的拍到一张图，分别是四种结论。

只证明控制面能响应，不足以报告文件、脚本、preview 或完整硬件矩阵通过。真实验收记录应包括板卡/物理身份、安装固件提交、所用端点、当前 session/epoch、请求阶段、时间预算和应用级证据。

建议的验收矩阵

表 18: 按功能与边界验证

类别	操作	成功依据
framing	分片 metadata/payload，连续粘包，CRC 错误，payload 含 SOF/EOF，UTF-8 跨帧。	仅完整合法帧交付，错误有界重同步，正文没有被分隔字节截断。
会话	两连接争用、同 nonce acquire 重试、release 重接管、租约到期、TCP 槽位复用。	唯一 owner，正确错误码，旧 link/session 输入输出不进入新连接。
VM	当前脚本占用、Ctrl-C、软复位、新 epoch ready、旧输出延迟到达。	真实 REPL 新标记，旧 epoch 不能完成新请求。
文件	空文件、中文/二进制、分块、64 KiB、取消、发布结果未知。	bytes/文件大小/内容 hash 校验，原文件及保护文件不被破坏。
preview	持续 JPEG、慢读者、串口切换/重连、与 RPC 交错。	完整图像、正确尺寸和当前范围；允许 preview 丢帧但不接受旧 owner 数据。
网络	配置 accepted 与最终 IP、关闭/恢复、FIN/RST、静默客户端。	最终事务状态、owner/slot 分别回收，新连接可恢复；未知业务不重放。
下载/复位	Python 请求、USB control 请求、ROM 枚举、写入/读回、运行态重接管。	分别证明模式切换、镜像校验与新运行态身份，而非仅观察端口消失。

使用真实安装固件的版本记录，不把“从源码推断”“CI 编译通过”“在另一块板运行过”写成当前板卡已经通过真机验收。浏览器字节桥测试也应说明是否覆盖原生串口 chooser/DTR。

统一 Core 真机检查

设备协议与业务真机测试统一运行 `host/device-core` 中的 `npm run test:hardware`，直接调用 `createDevice()` 的公开 API。摄像头、文件和 LCD 均走同一个 Core，不再维护独立的主机协议客户端或 Agent smoke 入口。串口适配器只传字节，TCP 使用 Node socket；无需安装 Agent。

使用 Node.js 22。串口模式使用 npm 原生绑定，无需主机 Python。所有输出放在项目目录中；以下从仓库根目录执行：

```
npm --prefix host/device-core test
npm --prefix host/device-core run check:schema
npm --prefix host/device-core run check:package
npm --prefix host/device-core run test:hardware -- --serial <USJ-port> --role usj -
↪-checks all --timeout 600000
npm --prefix host/device-core run test:hardware -- --serial <USB-OTG-port> --role␣
↪usb --checks identity,camera,display
npm --prefix host/device-core run test:hardware -- --tcp <device-ipv4>:3232 --
↪checks identity,memory,camera
```

`--checks` 接受 `all` 或逗号分隔的检查名。摄像头/显示检查包括 `identity`、`memory`、`camera`、`framesizes`、`pixformats`、`orientation`、`ipa_stats`、`snapshot_rate`、`preview`、`image_ops`、`display`、`display_stream`、`camera_display_lifecycle`、`image_copy_to_fb`、`image_transform_argument`、`sensor_snapshot_buffer`、`memory_policy`、`espdl_inference`、`isp_ccm_frames` 和 `isp_lsc_display`。`all` 选择完整检查集，不表示所有板卡都具备对应硬件；报告应明确区分通过、失败与缺少能力的跳过，不能将跳过算作已通过。初次实验只测试当前硬件，最终提交才由 CI 运行完整固件矩阵。

ISP 图像与内存检查 调查 ISP 时使用静止且有足够照明的目标，并指定准确板型和实际安装的固件版本：

```
npm --prefix host/device-core run test:hardware -- --serial <USB-OTG-port> --role␣
↪usb --checks isp_ccm_frames,isp_lsc_display --expect-board ESP32_P4X_EYE --
↪expect-firmware-version <installed-version> --frames 400 --require-all
```

`isp_ccm_frames` 对受控的红色/灰度 CCM 矩阵各单写一次，记录连续八张实际图像的帧序号、耗时、RGB 总和及平均饱和度。其中一次灰度采集故意在写入后休眠两秒：第一张仍可能来自旧缓冲，但持续消费后应在有限序列内看到新效果。检查还验证 IPA 占用时拒绝手动写入，以及重写单位 CCM 后显式 ISP 白平衡增益的读回值和实际 RGB 比例保持一致。比例对照要求目标静止，不测量或冻结传感器内部自动白平衡。亮度足够的采样点少于十六个时，检查会以效果不可观测失败，不将暗图当作 CCM 生效证据。

`isp_lsc_display` 使用 QVGA RGB565、驱动报告的 LSC 表长度和原报告从 1.0 到 0.5 的二次增益曲线，将摄像头图像居中绘制到全屏 canvas 后调用 `lcd.write()`。预热两个 LSC 表存储区和显示暂存区后至少执行八十帧（`--frames` 可增加帧数），每帧双写 CCM，每二十帧切换 LSC。报告包含 DMA/内部/PSRAM 的空闲量与最大连续块、非法表被拒绝后恢复显示、资源释放及新一轮摄像头/LCD 初始化。内存历史最多保留十个采样点，避免观测记录持续增长。小幅内存容差允许无关分配；长时间测试应结合完整序列检查趋势。这属于软件图像与资源证据；判断物理面板是否仍花屏，需要在 LSC 切换时观察真实 LCD，检查通过或软件截图均不能单独证明。当前固件通过也不等于已经查明旧固件报告的根因。

CI target test 契约 CI 硬件门禁使用 ESP-IDF v5.5.6，在 P4 Function EV Board 上执行显式选定的 18 项运行时检查，包括 `espdl_inference`。两项 ISP 检查保留为照亮的静止目标上的单独人工验收；新增注册检查不会静默扩大无人值守 CI 套件。CI 同时保存摄像头 JPEG 与 LCD 测试图案 JPEG，并将截图尺寸与活动显示对象核对。所有选中项必须通过（`--require-all`），不支持能力的 SKIP 也会使该板卡门禁失败。摄像头模式循环仅容忍明确的不支持错误；内存分配失败、超时及非预期的模式切换错误都会使检查失败。这只验证所选开发板与 IDF 组合，不代表编译矩阵中的其它板卡。驱动会将 `--expect-board` 和下载构建的 `project_description.json` 中的精确 `project_version` 同时传给启动探测与完整套件，在执行 Python、摄像头和显示检查前拒绝错误板型或旧固件。

通过 CI 验证单块开发板时，将 `ESP_VISION_TEST_BOARD` 和 `ESP_VISION_TEST_IDF_IMAGE` 设为固件矩阵中的准确组合，并显式设置 `ESP_VISION_TEST_HARDWARE=1`。P4 Function EV Board 使用 `ESP32_P4X_FUNCTION_EV_BOARD` 和 `v5.5.6`。此时只选择匹配的固件和硬件任务；未启用硬件开关的定向任务仅构建固件。`ESP_VISION_DEVICE_CORE_ONLY=1` 仍禁用所有固件和硬件任务。

运行时辅助脚本通过有界的 `python.execute()` 调用装载到独立 RAM 命名空间。每条编码后命令都满足固件声明的 `replInputMax`（最多 511 字节）；更长的请求会触发 Device Core 的临时文件执行路径。成功、失败或取消后都会释放辅助命名空间和 LCD 资源，因此内存、摄像头和显示检查不依赖可写文件系统。

辅助脚本源码按 UTF-8 字节偏移传输，回执包含会话 nonce、偏移和实际提交的内容。相同分块可安全重复提交；内容冲突或无效回执立即失败。加载器在安装前验证拼接后的字节数和 SHA-256，安装后要求明确确认安装完成、入口函数可调用且源码缓冲已释放。只有 `TIMEOUT` 允许恢复，每个初始化操作最多通过公共 Python API 尝试三次，单次调用和整体超时保持不变。安装完成后的响应丢失可以恢复，且不重新执行辅助脚本；执行中断造成的部分安装会失败，不会重放。取消和语义错误均不重试。

初始化重试和失败会记录阶段（`init`、`chunk`、`assemble`、`verify` 或 `install`）、从零开始的命令序号、尝试次数、编码输入字节数、毫秒耗时及最后一个已确认操作；分块诊断还包含从零开始的分块序号、UTF-8 偏移和字节数。能力查询和主机源码读取失败单独标记阶段。公共执行接口超时可能发生在 SDK 同步或命令执行期间；这些诊断能定位初始化操作，但不能据此认定传输或固件层根因。报告不包含原始 Python 输出或源码。

辅助脚本不可用时，报告只记录一次 `fixture-bootstrap` 失败。所有依赖项（包括首个受影响的检查）均报告 `BLOCKED`，携带 `blockedBy: fixture-bootstrap` 和 `FIXTURE_UNAVAILABLE`；独立身份检查仍继续执行。汇总分别统计 `passed`、`failed`、`skipped` 和 `blocked`。受阻检查既不算已执行失败，也不算能力不支持的跳过，即使未指定 `--require-all`，硬件任务仍失败。清理使用独立取消信号；清理失败单独报告，不替换原始初始化失败或取消原因。

`espdl_inference` 通过有界的公共 `python.execute()` 调用，将仓库自带的 495,840 字节 ESPDet Pico Face 模型装入只读 RAM VFS，在主机和设备两端验证 SHA-256，再通过真实 `espdl.ESPDet` 对

RGB565 摄像头图像推理。检查同一复制帧的结果可重复性、检测元组结构、检测框与图像重叠、置信度/类别范围及正的推理耗时；报告包含模型 SHA-256、传输分块数/耗时、加载耗时、帧数/检测数及平均/最慢推理延迟。镜头内没有人脸时允许空检测结果；这项合理性检查不代表模型准确率或性能指标已达标。无论成功、失败或取消，都会释放模型、RAM 挂载及摄像头，不读写 Flash/SD 文件，也不取得 MSC writer 租约。

模型上传通过命名空间 nonce 和准确的已提交字节偏移确认完成。只有超时分块允许重试，每块最多调用三次公共执行接口；重放只接受最近一个字节完全相同的分块。数据冲突或乱序、确认缺失或错误以及取消均立即失败。报告包含 `modelRetries` 和 `modelRetryOffsets`。挂载、模型清理及最终命名空间不存在均要求显式确认或读回，不能将打印 Python 异常后执行包装器返回成功当作操作成功。各命令与整体的现有超时保持不变。

默认模型 SHA-256 为 `04eec2220adbf642a843c953817fb219a06c094aedb76c181ba2a958c65201dc`。`--model MODEL.espdl` 可以选择兼容的 ESPDet 模型（至多 2 MiB），并通过 `--model-sha256 HEX` 固定预期字节。已安装的 npm 包不携带模型资源，需要显式提供 `--model`。模型缺失只会跳过可选验收，在 `--require-all` 下仍失败。严格推理验收要求摄像头和 ESP-DL 能力，与持久化存储是否可用无关。

存储单独取证。身份检查会报告各卷的存在、挂载及写入状态；运行时通过不代表文件上传可用。在已准备好文件系统的板卡上使用 `--write`，显式要求 Flash 已挂载且可写，执行唯一临时文件的二进制写入与读回，并仅清理测试自己的文件。未挂载或不可用的存储会使该项以 `STORAGE_UNAVAILABLE` 失败。CI 和硬件 CLI 都不会格式化存储、通过擦除数据掩盖挂载失败，或用 SD 卡代替 Flash。

摄像头闭环要求当前会话中新的完整 stdout marker 或经校验的 JPEG。Core 管理上传、执行和关联，测试继续检查像素格式、尺寸、方向、IPA 状态、快照速率、preview、图像处理及显示。握手、上传 ACK、执行排队或 REPL 提示符本身都不能替代业务完成证据。长期运行时持续接收数据并处理 `VM_BUSY` / `VM_TIMEOUT`。

LCD 抓屏与图案核验

`--screen-output` 显示已知测试图案，通过 `device.capture({ source: 'screen', format: 'jpeg' })` 读取 LCD 最后一帧 RGB565 的 JPEG，再保存到指定路径。先将 `ESP_VISION_WORK_DIR` 设置为检出目录外可写的工作目录，再创建输出目录：

```
mkdir -p "$ESP_VISION_WORK_DIR/hardware-validation"
npm --prefix host/device-core run test:hardware -- --serial <USJ-port> --role usj -
↪-checks display --screen-output "$ESP_VISION_WORK_DIR/hardware-validation/lcd.jpg
↪"
```

抓屏调用固件 `debug.capture_screen`，在 VM 任务中读取显示缓冲区，不调用摄像头。应核验 JPEG 可解码、尺寸与图案符合预期，并对照设备真实屏幕观察。LCD 未初始化返回 `DISPLAY_NOT_READY`；无 LCD 或该固件不支持时返回 `NOT_SUPPORTED`，不得改用摄像头掩盖失败。抓屏反映软件提交的最后显示帧，不证明物理面板、电缆或背光正常。

这些检查会停止当前 Python 程序，并可能改变显示内容。安排独占设备时段，不与 IDE/其他测试同时持有设备。设备有有效 owner 时应释放或等待租约，不通过换接口抢占。测试使用独立临时文件并清理自己的文件；保存报告、JPEG 和日志时记录实际固件、板卡与接口。复位、掉线和取消后保留未知业务状态，不自动重放写入。

测试入口与职责

表 19: 一套协议实现，分层验证

入口	覆盖范围
`host/device-core: npm test`	Core 公开 API、协议、会话、发现及烧写契约的确定性主机测试。
`host/device-core: npm run test:hardware`	串口/TCP 实际连接、摄像头及显示闭环，同一个公开 Core。
`tools/ci/flash_and_test.py`	CI 板卡/IDF 产物选择与烧写，随后调度 Core 真机检查；不另实现 EV-MUX。
`tools/tests/`	固件主机原生状态机、JSON、队列、存储及复位契约。

故障定位顺序

表 20: 现象与下一步

现象	先检查
串口有字但 hello 超时	是否为原始启动/panic 文本；选错 USB 接口；startup 半帧；EV-MUX 是否被运行期关闭；帧 CRC/字节长度与 reader 是否持续运行。
USB 已配置却不能接管	capabilities 与有效 owner；不要拿 DTR=false 当成 v4 不可用条件。
`SESSION_BUSY`	哪个 IDE/Agent/终端仍保活；先显式释放或等待租约，不修改 route 来抢占。
vmReady=true 但文件/抓帧超时	boot/main.py 是否仍运行；VM 队列是否忙；中断后是否打印了真实的新 REPL marker。
收到 seq 相同的数据却 RPC 未完成	type/channel/method/session/epoch 是否符合请求；是否拿 event 代替 rsp。
preview 偶尔跳帧	JPEG 是否完整、是否属于当前范围；慢读者和 latest-frame 策略是否符合预期。
TCP 断线后两槽耗尽	handshake/lease/30 秒 idle 各自状态，旧 generation 是否被回收，而不是仅看 owner 已释放。
下载请求后端口消失	ROM 是否重新枚举，USJ 是否有 ROM 日志，下载位是否按掩码设置/清除，原句柄是否已失效。
全局 drop 增长但 TCP 标记通过	未读默认/回退口、队列类别和采样范围；不要把不同端点的计数等同于这个业务失败。

只保存必要的状态与标量计数。SSID/密码、完整文件内容和 Flash/NVS 备份可能含敏感信息，不应混入公开诊断报告；占位示例不代表设备实际配置。

文档与实现的同步边界

若新增 channel、后端、控制 action 或修改 stderr/所有权/限额，应同时更新本章相关页、capabilities、参考客户端与契约测试。仅修改图中的名称不构成协议变更；新方法还须有实际固件分发实现。

固件单元测试与 Core 验收的边界

`tools/tests/test_ev_*.py` 在主机上借助 ESP-IDF/FreeRTOS 替身编译运行生产 C 逻辑，不连接设备，也不实现另一套主机协议客户端。保留部分写入、队列背压、分配失败、旧连接 generation、会话过期和存储交接等确定性覆盖。配套 C 测试和 stub 头文件属于测试基础设施，不是另一套设备操作工具。

用户可见功能和真机验收统一走 Device Core；新增连接、Python、文件、preview 和抓图场景使用公开 API。固件故障注入仍与固件同处维护，因为普通 SDK 请求无法稳定强制任务调度、内存分配或驱动失败。源码字符串断言仅作为补充，不能替代行为执行和真实硬件验收。

11.14 发布 Target 验收

Function EV、P4 Vision 和 S31 Korvo 发布 Target 任务在原有板级相机、图像、显示和推理检查之外，强制执行以下检查。`--require-all` 将不支持的已选项目判为失败；缺板或缺外设不能记为通过。

- `--verify-flash` 使用 esptool 将 `flasher_args.json` 中每个镜像与 Flash 独立比较，并应用与写入相同的镜像头设置。比较失败后不执行运行时测试。存在多个串口候选时，写入前必须明确指定 `ESPPORT`/`--port`；校验与运行时连接继续使用选中的端口。
- `discovery` 在设备保存带随机标记的 Python 变量时，通过 SDK 被动枚举选中的串口，再检查变量和运行时身份仍保持不变。显式选择 TCP 端点时，要求 mDNS 广播的端点和设备 ID 同时匹配。枚举不会探测其他设备。默认串口 CI 任务不代表已验证局域网发现。
- `communication` 检查五轮不同的 Unicode 输出，每轮 UTF-8 数据超过 4 KiB，并在执行之间插入身份和运行状态 RPC。截断、旧输出或数据改变均失败。
- `reconnect` 对选中的传输连接执行三次释放与重新获取，每轮检查设备 ID、板型、固件版本以及新的 Python 标记。这验证正常断开重连，不代表已验证拔线、断电或租约争用。
- `modules` 在设备上执行 SHA-256 已知答案、十六进制/base64 二进制往返、struct 字节序、Unicode JSON、BytesIO 定位读取及垃圾回收。`ulab` 实算并核对均值、逐元素乘法与求和、平方根及排序。两项均使用 RAM，不依赖 Flash/SD 写入。

Python 驱动的 `--junit PATH` 为各项 SDK 检查、烧录与校验阶段及运行时就绪生成 GitLab 测试报告。夹具加载进度事件仅保留为诊断日志；被阻塞的检查显示为失败，并保留其前置依赖。运行时验收前的失败也会生成失败的驱动测试项。日志及相机、屏幕制品继续保留；仅烧录成功不构成发布验收通过。

2026.11.14 候选版应共同保留最终提交、流水线、板型、接口、固件版本及镜像制品。功能完成后再运行完整 CI 固件矩阵。现有发布门禁仍要求三个 Target 任务全部通过。旧固件上的测试仅验证测试程序，不能作为候选版验收。创建标签与发布是独立的发布操作。

协调发布 npm 包时，受保护 master 上的 CLI 打包必须等待 Core 发布成功或确认该版本已存在；取得已发布制品地址与 SHA-512 后，再执行 `npm ci`、构建及测试，并使用解析后的锁文件发布同一份已验证 CLI 制品。等待发布期间，MR 检查可以使用版本匹配的 Core 源码包。Plugin 的发布打包要求已发布 CLI 实际内置的 Core 版本与 Plugin 清单记录一致，不会回退到旧版 CLI 或 Core。因此，Core 0.8.1 必须先于 CLI 0.2.1 可用，两者均可用后才能打包 Plugin 0.2.0。固件资格检查、npm 发布与在线 Knowledge 可用性是独立的验收节点；仅创建固件标签不能证明对应 Knowledge 已上线。

Core 0.8.1、CLI 0.2.1、Plugin 0.2.0 和 Knowledge MCP 0.2.0 均已发布到 npm。Knowledge MCP 0.2.0 不绑定 Core/CLI。初始保留目录覆盖 2026.10.21、2026.10.23 和 2026.11.06，尚不覆盖 2026.11.14 固件候选版。应在标签通过资格检查且固件发布后归档新快照，再通过独立评审的 Knowledge 发布导入。Plugin 0.2.0 的发布门禁会等待同流水线的 Knowledge 部署成功，并在写入 npm 前校验线上版本化接口。

表 21: 按问题定位

想解决的问题	阅读入口
Python print、printf、IDF log 为什么不是同一条输出?	架构、执行上下文与标准输出
两个 USB 口、UART、Wi-Fi/TCP 分别做什么?	物理接口、连接代次与构建配置
两台主机争用、断线、软复位如何恢复?	会话所有权、接管与恢复
怎样实现自己的帧 parser 和 RPC 客户端?	帧格式、逻辑通道与 *RPC* 协议
怎样输入 Python、读写文件和运行长脚本?	交互式 *Python*、脚本与文件操作
preview 和 debug.capture_frame 有什么不同?	预览、调试抓帧、背压与统计
怎样配网、使用共享客户端并处理启动残包?	主机接入、*Wi-Fi* 配置与恢复实践
如何从 npm 安装 Device Core，并接入串口/TCP、文件与预览?	*Device Core* 接入指南
怎样证明命令真的完成，怎样定位问题?	通信验收、闭环测试与故障定位

9.3 六个必须分开的概念

- **物理接口 / sink**：后端类型，例如 `usj` 或 `tcp`。
- **link**：一次实际连接，以后端、槽位和 generation 标识；同一个 TCP 槽位再次连接仍是新 link。
- **stream**：内部路由分组，只有 `user`、`debug` 两组，不是八个独立串口。
- **channel**：帧的业务语义，例如 `repl.stdout`、`preview.frame`。
- **sessionId**：某个 link 持有设备所有权的凭证；不能迁移到另一个 socket。
- **vmEpoch**：MicroPython VM 的代次。会话仍存在，不表示旧 VM 的工作仍有效。

9.4 当前实现边界

整个设备同时只有一个会话 owner，并一起拥有 user/debug 两个 stream。其他连接可以发现设备，但不能并行执行另一个控制会话。所谓“抢占”在本实现中是争用接管资格与显式交接，不存在强制踢走有效 owner 的接口。

TCP 是基于 STA IPv4 的原始字节流服务，默认端口 3232，最多两个已接受的 TCP link；它不是 WebSocket、HTTP、MicroPython WebREPL，也不提供本版之外的身份认证或传输加密。浏览器通过主机侧字节桥接访问 TCP，详见*主机接入、Wi-Fi 配置与恢复实践*。

ROM 下载模式与 EV-MUX 运行态属于不同协议阶段；MicroPython 的 REPL 仍是解释器控制台，而 IDF system console 仍承担初始化前、ROM/bootloader 及部分致命故障输出。不能把这些边界上的原始文字当成合法业务帧。

Chapter 10

Device Core 主机 SDK

`@esp-vision/device-core` 是 ESP-VISION EV-MUX v4 固件对应的无 UI JavaScript 主机 SDK。IDE、Device CLI 和自定义主机应用共用它的字节协议、会话所有权、保活、Python、文件与图像能力。Agent 工作流通过 CLI 控制设备；官方 Knowledge MCP 通过 HTTP 提供文档。SDK 源码在固件仓库 `host/device-core` 维护。

10.1 应用 API 完整参考

本页覆盖正在准备、此改动尚不发布的 0.4.0 中 `createDevice` 返回对象的全部应用 API。除特别说明外，操作异步执行，失败时 reject。请求中的未知字段会被拒绝；不能给所有方法统一附加未声明的 `timeout` 等选项，应以各方法列出的字段为准。超时单位为毫秒，须为有限正数。

方法	用途
`createDevice(options)`	创建一个独立管理连接的设备对象。
`device.discover(options)`	枚举局域网公告，不获取设备所有权。
`device.connect(options)`	打开连接、协商 v4 会话并验证 REPL 可用。
`device.disconnect()`	释放所有权、取消工作并关闭连接。
`device.inspect(request)`	查询身份、能力、运行态、网络或存储。
`device.configure(request)`	局部修改网络配置。
`device.control(request)`	停止、复位或进入 ROM 下载模式。
`device.firmware.check(request)`	检查开发板推荐固件版本。
`device.firmware.update(request)`	下载、校验、烧写并验证 application 镜像。
`device.capture(request)`	获取摄像头或 LCD 的 JPEG。
`device.on(event, handler)`	订阅事件并获得退订函数。
`device.python.input(data)`	发送交互式 REPL/stdin 输入。
`device.python.execute(request)`	执行源码或设备中已有脚本。
`device.files.list(request)`	列出目录。
`device.files.read(request)`	读取字节或 UTF-8 文本。
`device.files.write(request)`	创建或覆盖文本/二进制文件。
`device.files.remove(request)`	删除一个文件或空目录。
`device.files.mkdir(request)`	创建目录，可递归创建父目录。

10.1.1 创建对象与字节传输

```
createDevice({ transport?, discovery? } = {}) → device
```

`transport` 提供原始字节连接；`discovery` 提供 `list({ timeout, signal }) → Promise<Array>`。创建时二者均可省略，连接时必须具备 transport。仅发现用途的对象可在 `connect` 时补入 transport。Core 不会自动选择串口或根据发现记录建立 socket。

传输成员	契约
`writeBytes(bytes)`	必需；接收 Uint8Array，全部字节被宿主连接接受后 Promise 才完成。短写或失败必须报错。
`subscribe({ data, error, close })`	必需；同步返回退订函数，data 接收原始 Uint8Array 字节块。
`connect({ signal })`	可选；Core 完成订阅后打开连接。
`disconnect()`	可选；关闭物理连接并释放 reader/writer 资源。

10.1.2 device.discover(options)

```
{ timeout = 3000, signal? } → Promise<DiscoveredDevice[]>
```

`timeout` 为 50–10000 的整数。必须配置 discovery provider，否则抛出 `DISCOVERY_UNAVAILABLE`。结果会校验并按主机名排序，最多接受 256 条。发现不连接设备，也不代表设备当前没有 owner。

```
const discovered = {
  deviceId: 'e8f60ae75642',
  hostname: 'esp-vision-esp32-p4x-eye-e75642-2026-08-27',
  board: 'ESP32_P4X_EYE', firmwareVersion: '2026.08.27',
  evMuxVersion: 4, compatible: true, port: 3232,
  addresses: ['192.168.1.107'],
  // Optional: connectionUrl (bridge), expiresAt (expiry timestamp).
}
```

上述为数据结构示例，不是应当连接的固定设备。`deviceId` 是去掉分隔符的主芯片 MAC；`firmwareVersion` 是固件版本，不是 IDE 版本。根据所选地址/端口创建字节连接，再将记录传为 `expectedDevice`。Node 与浏览器 provider 见*高级导出 API 完整参考*。

10.1.3 device.connect(options)

```
{ transport?, role = "unknown", expectedDevice = null, timeout = 60000,
probeRepl = true, autoBaud = true, signal? } → Promise<Inspection>
```

`role` 接受 `unknown`、`usj`、`usb`、`uart`、`tcp`。Core 先订阅再打开连接，协商 v4，按需检查 `expectedDevice`，获取会话并探测 REPL。返回默认查询结果 identity/capabilities/runtime。正在连接、已连接或正在断开的对象会以 `DEVICE_BUSY` 拒绝重复连接。身份匹配不等于认证。默认 `probeRepl: true` 会发送中断并探测 REPL，因此可能停止既有 Python 程序。设置 `probeRepl: false` 可获取 EV-MUX 会话而不发送 REPL 输入，供程序持续运行时观测或调用原生 RPC；仍遵循会话所有权，不会强制抢占。后续 Python 执行和依赖 REPL 的文件操作仍可能中断程序。该参数只接受布尔值。

`autoBaud` 默认为 `true`。当固件声明支持 UART 协商且实际会话 owner 为 `uart` 时，Core 通过传输层可选的 `setBaudRate(baudRate, { signal })` 依次测试 2000000、1500000、921600、460800、115200 bps，首个通过即停止。`autoBaud: false` 让短时身份探测保留 115200，跳过速率测试。原生 USB 与旧固件会跳过协商。该选项必须为布尔值。运行态速率与 ROM/stub 烧录速率独立。

10.1.4 device.disconnect()

返回 Promise。释放会话、终止待处理工作、关闭会话解释器、退订输入并调用可选的 transport 关闭方法。重复调用共用清理过程，已断开的对象直接完成。释放或传输清理失败仍可能 reject，因此连接失败也应在 `finally` 中清理。重连需要可重新打开的 transport，或通过 `connect` 提供新 transport。

10.1.5 device.inspect(request)

```
{ fields = ["identity", "capabilities", "runtime"], timeout = 5000, signal?
} → Promise<Inspection>
```

字段	结果
`identity`	`{ deviceId, hostname, board, firmwareVersion } | null`
`capabilities`	协商得到的固件能力、features、channels、limits。
`runtime`	会话/parser/transport 快照，附 vmReady、vmEpoch 和当前 run 或 null。`scriptRunning` 和 `run` 只跟踪本 SDK 会话启动的执行；新连接不能根据 false/null 判断设备上的后台程序空闲。`vmReady` 表示 VM 已启动，不表示空闲。
`network`	network.status 结果，包括 enabled/configured/state/IPv4 与事务状态，不含已保存密码。
`storage`	flash 与 sd 的 storage.writer.status 结果，要求 storage.writer.lease 能力。新固件包含 fs_state/fs_error/fatfs_result/format_supported，见存储、文件系统与数据保护。写入所有权本身不代表文件系统就绪，也不授权格式化。

`fields` 必须是上述五个名字组成的非空数组，仅返回请求的键，重复字段只查询一次。身份和能力是协商数据的副本，不是新采集的传感器数据。网络与存储会发起 RPC；该超时应用于各底层查询，不是整个 inspect 的统一总时限。

10.1.6 device.configure(request)

```
{ network: { enabled?, credentials? }, timeout = 45000, signal? } → Promise<{ ok, completed, network }>
```

`credentials` 为 `{ ssid, password }` 或 `null`。SSID 为 1–32 个 UTF-8 字节，不得包含 NUL。密码可为空（开放网络）、8–63 个可打印 ASCII 字符或 64 位十六进制 PSK。`enabled` 为布尔值。至少提供一个更新字段，省略字段保留原设置。清除凭据会关闭网络，不能同时设置 `enabled: true`。

```
await device.configure({ network: {
  credentials: { ssid: 'example', password: 'example-password' },
  enabled: true,
} })
await device.configure({ network: { enabled: false } })
await device.configure({ network: { credentials: null } })
```

配置总时限包含轮询事务最终结果，不只等待 accepted ACK。`completed` 包含已确认的 `{ action, operationId }` 步骤。失败时附加 `error.completed`，必要时附 `error.pending`；已完成步骤不自动回滚。局部配置由顺序操作组成，不是原子事务。修改 Wi-Fi 可能断开 TCP，建议通过 USB 配网，不得记录凭据。

10.1.7 device.control(request)

```
{ action, mode?, timeout = 15000, signal? } → Promise<control result>
```

请求	行为
`{ action: "stop" }`	中断/停止 Python，可在 python.execute 运行期间调用。
`{ action: "reset", mode: "soft" }`	软重置 VM，观察新 epoch 和就绪状态。
`{ action: "reset", mode: "hard" }`	重启硬件，连接可能消失。
`{ action: "bootloader" }`	请求 ROM 下载模式，随后断开并使用 ROM 烧写器。

`mode` 仅在 reset 时必需，其他动作提供此字段会报错。stop ACK 不代表脚本清理已完成，应继续观察 `done`。复位/下载 ACK 不代表正常启动或 ROM 枚举已完成，设备恢复后须重连并查询。

10.1.8 device.python.input(data)

接收字符串或 Uint8Array，返回底层写入 Promise。交互行以 `\r` 提交；在 `completion: "start"` 的运行期间可向 stdin 写入（例如响应 Python `input()`）。此方法没有请求对象形式的 timeout/signal 参数。输出由 `python.output` 事件交付，输入发送完成不代表求值成功。其他自动操作占用解释器时，冲突的终端输入以 `DEVICE_BUSY` 拒绝。

10.1.9 device.python.execute(request)

```
{ source? | path?, completion = "wait", timeout = null, maxOutputChars?,
signal? }
```

`source`` (字符串) 与 ``path`` (设备绝对路径) 必须且只能提供一个。``completion: "wait"` 返回 `{ ok, path, output, outputTruncated }`，直接执行 source 时 path 为 null。`completion: "start"` 返回 `{ ok: true, path, done }`，done 完成后返回同样的结果结构。start 返回前会等待生成的 runner 发出私有 VM 进入标记，不代表应用初始化或执行完成。`timeout` 仅允许用于 wait 模式；省略或 null 表示无整体执行时限，准备阶段的协议交互仍有边界。`maxOutputChars` 为非负整数，限制保留的输出字符数，不限制串口带宽或脚本运行时长。

```
const off = device.on('python.output', text => terminal.write(text))
try {
  const run = await device.python.execute({
    source: 'name = input("Name? "); print(name)', completion: 'start',
  })
  await device.python.input('example\r')
  console.log(await run.done)
} finally { off() }
```

示例 terminal 由 UI 提供。start 模式的 source 会上传到 Core 拥有的唯一临时文件，运行完成后清理；传入的用户 path 不会因此被删除。start 句柄返回后，`device.disconnect()` 会释放会话但不会请求停止程序；会话关闭时 `done` 在本地拒绝，设备上的程序可能继续运行。请求中的 `signal` 会一直绑定到 `done` 结束；会话仍可用时，取消会请求停止程序。断线可能导致临时文件未清理。Python stdout 始终绑定原 EV-MUX 会话，无法由新连接重新接收；需要实时输出时应保持会话，用截图或应用进度文件确认后台程序状态。须检查输出中的 Python traceback，`ok` 是执行/传输结果，不能通用地断言任意 Python 程序获得了正确业务结果。

10.1.10 设备文件操作

设备路径必须是绝对路径，最多 123 个 UTF-8 字节。禁止控制字符、反斜杠、双引号、任意位置的 `..`、重复斜杠及尾随斜杠。只有 list 可使用根路径 `/`。操作要求文件系统已挂载、存储所有权允许且空间足够。以下方法与 Python 共用解释器互斥。

device.files.list(request = {})

```
{ path = "/", recursive = false, signal? }
```

返回平铺数组 { name, path, type: “file” | “directory”, size }，目录 size 为 0；递归结果仍是平铺结构。

device.files.read(request)

```
{ path, encoding = "binary", timeout = null, chunkTimeout = 30000, maxBytes
= null, signal? }
```

binary 返回 Uint8Array，utf-8 返回字符串。省略/null `timeout` 表示无整体读取时限；`chunkTimeout` 限制每次 REPL 交互，默认 30000 毫秒。显式指定整体时限时，每次交互也受剩余时间限制。maxBytes 为 null 或 0..16777216 的整数，限制整文件读取；超限会拒绝，不会悄悄返回文件前缀。

device.files.write(request)

```
{ path, data, overwrite = true, onProgress = null, signal? }
```

data 为字符串或 Uint8Array；overwrite:false 请求只创建。进度回调接收 { writtenBytes, totalBytes }。完整文件通过固件 SHA-256 校验和发布后，返回包含总 bytes、complete:true 与 path 的提交确认；不会执行文件。

device.files.remove(request)

```
{ path, type = "file", signal? }
```

type 为 file 或 directory；删除目录要求目录为空。返回 Promise<void>，不提供递归删除标志。

device.files.mkdir(request)

```
{ path, recursive = false, signal? }
```

返回 {ok:true}；递归模式创建父目录，允许目标已是目录，但不允许被同名文件占用。

SDK 0.2.0 及以上的所有文件(包括 Python 源码和模型)共用可靠的原始二进制 `file.write` 上传。固件须支持 `file.write.binary` 并提供有效 `limits.fileWriteChunkMax``(4096..16384,按 4096 对齐)；不支持时返回 UNSUPPORTED_OPERATION 并提示更新固件，不回退到 Base64/JSON 或 REPL 上传。list/read/remove/mkdir 仍使用 framed REPL,无需另加 files.writeBinary 公开方法。进度是已确认接收的字节，最终 Promise 成功才代表完整长度/哈希校验和发布完成。取消/失败会尽力 abort，固件也会清理超时、旧会话或旧 VM 上传。传输、哈希和普通发布前失败保留原目标；FAT 覆盖使用旧文件备份与普通重命名失败回滚，不承诺掉电原子事务。线上契约见:doc:`../communication/python-files`。应用更新使用 ``device.firmware.update()，复用 /firmware 子路径的 ROM 烧写实现。

10.1.11 device.capture(request)

```
{ source, format = "jpeg", quality?, timeout = 15000, signal? } → Promise<{ data, mimeType, metadata }>
```

`source` 为 `camera` 或 `screen`，格式只支持 `jpeg`。可选 quality 为 1–100 的整数，省略时采用固件默认值。data 为 Uint8Array，mimeType 为 image/jpeg，尺寸位于 metadata.width/height，而非返回对象顶层。

```
const frame = await device.capture({ source: 'screen', quality: 80 })
console.log(frame.metadata.width, frame.metadata.height, frame.data.length)
const blob = new Blob([frame.data], { type: frame.mimeType })
```

相机观测要求同时具有 `debug.capture_frame` 和 `debug.capture_frame.observed.v1`，在 VM 中编码应用最近一次成功 snapshot 发布的独立帧；不初始化摄像头、不触发新物理采帧，也不覆盖应用图像。观测内容是原始捕获像素，不包含应用随后的加工。对于旧的不安全固件，Core 在本地以 `UNSUPPORTED_OPERATION` 拒绝并提示升级，包括直接会话 RPC/action 抓图路径。没有帧或摄像头重配、关闭、VM 复位后观测失效时返回 `CAMERA_FRAME_NOT_READY`；观察副本分配失败返回 `NO_MEM`，应用 snapshot 仍成功。超时、取消都不会触发另行采帧。观察运行中的程序应使用 `connect({ probeRepl: false })`。

LCD 抓图要求 LCD 已初始化，只读取最后提交的软件 RGB565 帧，不启动摄像头或重新绘制屏幕。不支持硬件返回 NOT_SUPPORTED，LCD 未初始化返回 DISPLAY_NOT_READY。软件抓图不是物理面板回读。按需缓存、DMA 完成同步与 preview 交付见*preview-debug*。

10.1.12 device.on(event, handler)

同步返回 `unsubscribe()`。需要启动输出时应在 connect 前订阅。监听器抛错不会中断解析或其他监听器，会产生 listener.error 诊断事件。UI 销毁时须退订并释放图像 object URL。

事件	载荷	含义
`python.output`	`string`	Python stdout/stderr、提示符与交互输出。
`raw`	`string`	EV-MUX 帧之外的文字，例如早期启动输出。
`log`	`string`	固件/IDF 诊断文字。
`preview`	`{ data, mimeType, metadata }`	应用 img.flush() 产生的 JPEG，允许丢帧。
`state`	`{ state, identity?, reason? }`	connecting/ready/disconnected，以及 hello/network.changed 提示；ready 时附 identity。
`diagnostic`	`{ event, ... details }`	rpc.request/response/timeout、listener.error 与清理诊断；网络敏感凭据会脱敏。
`rpc`	`{ channel, metadata, payload }`	高级入站 RPC 观察，业务操作优先使用设备方法。

10.1.13 互斥、取消与错误

查询、配置、控制、Python、文件和抓图方法要求连接处于 ready。发现、事件订阅、固件检查不要求连接；固件更新指定开发板及 ROM loader 后也可在未连接时执行。设备对象遇到 Python、文件、配置、抓图等互斥操作冲突时返回 DEVICE_BUSY，不会自动把任意业务请求排队。stop/reset 可以与正在执行的 python.execute 协调；start 运行持续占用至 done，运行接受 stdin 时允许交互输入。

错误	处理
`TypeError / RangeError / INVALID_PATH`	修正参数；未知字段、非法枚举和路径会被拒绝。
`DEVICE_NOT_READY / DEVICE_BUSY`	先连接，或等待/停止当前操作。
`DISCOVERY_UNAVAILABLE / UNSUPPORTED_OPERATION`	补充 provider，或使用具备能力的固件。
`SESSION_BUSY / SESSION_LOST / VERSION_MISMATCH`	释放其他有效 owner、会话丢失后重连或升级固件，不强制抢占或盲目重放写入。
`TIMEOUT / AbortError / CONNECTION_CLOSED`	AbortSignal 可能保留调用方 Error 原因；完成状态可能未知，重试修改前须核对身份、状态与文件。
`VM_BUSY / VM_TIMEOUT / DISPLAY_NOT_READY / NOT_SUPPORTED`	明确处理固件错误，停止/初始化所需资源或提示硬件不支持。

所有异步调用均须处理 rejection，错误可能包含 code、errno、completed、pending，但不是所有错误类型都有 code。断开/取消不回滚设备修改、不保证脚本已终止，也不意味着可以重放非幂等操作。

10.1.14 固件检查与更新

```
device.firmware.check({ catalog?, identity?, version?, mode = 'update',
fetch = globalThis.fetch, signal? })
```

返回 { available, currentVersion, release, releases }。已连接时使用真实设备身份；未连接时传入此前查询得到的 identity，包含 board 与 firmwareVersion。检查不会接管连接、复位或写入设备。catalog 接受内存对象或绝对 HTTPS JSON 地址，下载失败会 reject；无匹配开发板时返回 release: null, available: false。省略 catalog 时，Core 自动读取 dl.espressif.com 上的 ESP-VISION 官方目录及各版本更新元数据；自定义 catalog 只是高级覆盖。历史条目缺少更新元数据时仍可列出，但不可选为安装目标。

目录格式为 { schemaVersion: 1, releases: [{ board, version, image }] }，发布方按推荐的新版本在前排序；默认选择首个匹配开发板的条目并校验 v4 镜像元数据；指定版本时按该版本选择。available 表示目标与当前固件版本不同，不表示任意日期/git 版本字符串数值更大。发布方负责推荐版本顺序，UI 在更新前展示目标版本，包括有意降级的情况。

image 包含 evMuxVersion: 4、非负整数 chipFamily、offset，以及精确 chipRevision 或整数范围 chipRevisionMin/chipRevisionMax，正整数 bytes 与 partitionSize``（镜像须放得下），64 位小写十六进制 ``sha256，以及 partitionTable: { offset, bytes, sha256 }。需要下载镜像时增加绝对 HTTPS url，直接指向 application bin；zip/artifact 下载不是原始 bin，Core 不负责解压。开发板名必须与固件身份完全一致，版本号是固件版本，不是 SDK/IDE 版本。SHA-256 用于校验与所选目录一致，不能证明恶意发布源可信。

```
device.firmware.update({ release, board?, mode = 'update', data?, fetch =
globalThis.fetch, openLoader, reset?, onProgress?, onPhase?, signal? })
```

release 为检查结果中选择的条目。已连接设备必须匹配该开发板；未连接时必须通过 board 显式指定实际选择的开发板。data 为 Uint8Array application bin；省略时下载 release.image.url。Core 先校验尺寸和哈希，再释放 EV-MUX 连接，随后调用宿主的 openLoader({ signal }) 获取 ROM 下载端口。宿主适配器负责串口/WebUSB 选择及 ROM 进入操作，Core 不实现网络 OTA，也不另造第二套进入下载模式的线上指令。即使通过 TCP 发现/查询设备，更新仍需要本地下载连接。

loader 契约沿用高级导出 *API 完整参考* 中的 FirmwareFlashSession。可选 reset(loader) 替代其 hard-reset 策略，onProgress(percent) 报告包含写后校验在内的 0–100 进度，onPhase(text) 报告阶段。默认 update 模式仅写 application 分区：写入前校验芯片/修订版本、安全状态及现有分区表，写入后由内置烧录器在设备端计算整段 MD5 并核对；不返回校验凭据的自定义烧录器保留完整回读和 SHA-256 校验。分区不匹配会拒绝；默认更新不做整片迁移，不擦除 NVS 或用户文件；显式完整安装见下文。

固件写后校验成功后返回 { verified: true, needsReset, version, board, reconnected: false }。needsReset: true 要求宿主提示用户物理复位。不自动重连 EV-MUX，也不宣称新版本已运行，须另行重连并查询固件版本。取消信号在下载、获取 loader 以及写入前有效；开始烧写后延迟取消，确保完成写后校验。失败会 reject 并关闭已获取的 loader，不承诺回滚。更新期间连接、其它设备操作及并发更新以 DEVICE_BUSY 拒绝。未知字段会拒绝。交接后设备保持断开，包括交接之后发生失败的情况。

```
const result = await device.firmware.check({
  catalog: trustedCatalog, // 宿主发布方提供的有序固件目录。
  // 离线/ROM 恢复时：identity: previouslyInspectedIdentity
})
if (result.available && userApprovedVersion === result.release.version) {
  const updated = await device.firmware.update({
    release: result.release,
    openLoader: ({ signal }) => host.openRomLoader({ signal }),
    onProgress: percent => console.log(percent),
  })
  console.log(updated) // 另行重连/查询；needsReset 时提示复位。
}
```

空白设备首次安装

复用 firmware.check/update，显式传入 mode: 'install'。不需要 EV-MUX 连接或运行中的固件；用户必须选择真实开发板，使芯片进入 ROM 下载模式，并选择 ROM 支持的 USB-Serial-JTAG（USJ）或 UART 端口。空白设备不能依赖应用固件提供的 USB-OTG CDC 接口。浏览器宿主仍提供 ROM 适配器，CLI 内置 Node/esptool 适配器。此流程不是 TCP OTA。

默认 `mode: 'update'` 保留原有只更新应用的行为；分区不匹配时拒绝，不会自动切换到 install。从 Core 0.7.0 起，显式 install 会先删除内部 Flash 文件系统中的文件（包括脚本、模型），再将经过校验的完整合并镜像从地址 0 写入，包括 bootloader、分区表和应用。镜像还可能替换 NVS 配置，包括已保存的 Wi-Fi 设置；不擦除 SD 卡。宿主必须在用户选择完整安装前明确提示删除行为。不执行整片擦除、固件备份、自动恢复或回滚。

沿用 schemaVersion 1，发布条目新增可选 `install: { offset: 0, bytes, flashSize, sha256, url, chipRevisionMin, chipRevisionMax, bootloader: { offset, bytes, sha256 } }`。flashSize 是布局要求的 Flash 容量（字节）；发布器取 bootloader 与应用芯片修订范围的交集。原有 image 仍是应用描述；安装会核验完整镜像中 bootloader、分区表和应用各自范围的哈希，范围必须有序、不重叠且不越界。发布流水线从现有合并镜像与实际 IDF 产物生成这些元数据；历史版本缺少合格安装元数据时不会推测兼容。

`firmware.check({ mode: 'install', ... })` 选择第一个具备安装元数据的版本，或精确选择指定版本。仅提供应用更新的版本仍显示在 releases 中，显式选择它安装时报 FIRMWARE_INSTALL_UNAVAILABLE；未知 mode 被拒绝。`firmware.update({ mode: 'install', ... })` 在释放会话和获取 ROM 端口前校验镜像，并解析通过哈希校验的实际 ESP 分区表；检查边界、重叠、标签及存在的 MD5，要求匹配的应用分区，以及位于固件写入范围外、按扇区对齐、可写且未加密的 `ffat` data/FAT 分区。缺失或不支持的布局会拒绝，不会推测旧 `vfs` 就是 FAT。随后检查真实芯片、修订、安全状态及 JEDEC 实测容量，只擦除这个 `ffat` 范围；擦除命令成功后直接写固件，不回读、抽查或计算擦后摘要。擦除失败或在写入前取消均不报告成功、不复位、不重放不确定操作。固件写入开始后沿用完成完整镜像校验和复位的策略；之后仍须重新连接并 inspect，确认文件系统已挂载且新固件实际运行。

```
const device = createDevice() // 不需要运行态 transport 或 connect()。
const selected = await device.firmware.check({
  identity: { board: 'ESP32_P4X_EYE', firmwareVersion: 'unknown' },
  mode: 'install', version: selectedFirmwareVersion,
})
await device.firmware.update({
  release: selected.release, board: 'ESP32_P4X_EYE', mode: 'install',
  openLoader: ({ signal }) => host.openRomLoader({ signal }),
})
```

10.1.15 列出并指定固件版本

`releases` 返回目录中当前开发板的全部版本，保留发布方排序；省略 `version` 时，`release` 是第一个推荐版本。传入 `version` 则按固件版本精确匹配，不存在时以 FIRMWARE_VERSION_UNAVAILABLE 拒绝，不会悄悄换成其它版本。同一开发板存在重复版本条目时拒绝该歧义目录。能查询的历史范围由发布目录决定，Core 不抓取所有 Git tag、CI 产物或未发布版本。显式选择旧版本也沿用相同的更新校验。

```
const history = await device.firmware.check({ catalog: trustedCatalog })
console.log(history.releases.map(item => item.version))
const selected = await device.firmware.check({
  catalog: trustedCatalog, version: selectedFirmwareVersion,
})
await device.firmware.update({
  release: selected.release,
  openLoader: ({ signal }) => host.openRomLoader({ signal }),
})
```

Node 宿主可直接复用随包提供的 ROM 适配器：

```
import { openNodeRom } from '@esp-vision/device-core/node'
await device.firmware.update({
  release: selected.release, board: 'ESP32_P4X_EYE', mode: 'install',
  openLoader: ({ signal }) => openNodeRom({ port: selectedRomPort, signal }),
})
```

SDK 0.5.1 已包含浏览器 ROM 适配器 `@esp-vision/device-core/browser` 的

`openBrowserRom({ port, logger?, signal? })`，复用上述安装/升级契约；用户手势串口授权和打包器要求见*Device Core npm 包对接*。

10.1.16 不可变发布快照与单 BIN 升级（0.4.0）

SDK 0.5.1 已包含这些发布读取器。`loadOfficialFirmware` 与 `loadReleaseManifest` 从 `@esp-vision/device-core/firmware` 导入；根入口仍只导出 `createDevice`。

`loadOfficialFirmware({ board?, version?, fetch = globalThis.fetch, signal? })` 优先读取 `config.toml` 的 `[publication]`：`schemaVersion: 1`、`version`、40 位 `sourceCommit`、`manifestUrl`、原始字节 `manifestSha256` 及 `releases: [{ version, manifestUrl, manifestSha256, boards }]`。历史版本不重复，各板集非空且唯一；发布器的其他事务字段保留但不改变读取逻辑。指定板型/版本必须存在于该引用。仅下载所选版本的一份 manifest，不遍历历史板型 × 版本矩阵；`firmware.check({ version })` 会传递版本给 reader。显式 publication、引用或 manifest 无效时明确失败，包括 404 或哈希不匹配，不作回退。只有完全不含 `publication` 的配置保留旧 feed；指定版本也会约束旧元数据请求范围。

`loadReleaseManifest({ url, sha256, version?, sourceCommit?, boards?, fetch = globalThis.fetch, signal? })` 不读取可变 config，只读取引用的快照。SHA-256 覆盖 UTF-8 JSON 原始字节。结果包含 `schemaVersion: 1`、`version`、`sourceCommit`、正整数 `pipelineId`、`layout: 'legacy' | 'single-bin'`、现有合格 `releases` 数组、完整 `website` 快照及可选 `mcp``（null 表示明确缺失）。可选身份/板集参数须完全匹配；返回前校验所有 release 与来源。官方 URL 限定在 ``https://dl.espressif.com/AE/esp-vision/`；模型下载固定到 `raw.githubusercontent.com/espressif/esp-vision/<sourceCommit>/models/...`。MCP 描述含 `url/bytes/sha256/version/sourceCommit`，使用官方 `bin/mcp/` 来源；website 版本/板型/模型也须有效，隐藏固件板型不要求出现在可视化板列表。读取限制为最多 64 个发布板型、256 个版本/模型记录、4 MiB manifest 字节。固定哈希检测相对可信发布引用的漂移，不等于为不可信发布者提供签名认证。

`loadOfficialFirmware` 在 manifest 快照外返回 `publication`；可选 `board` 只在整份 manifest 校验后筛选返回的固件条目。`publication.releases` 提供合格历史版本/板型引用；`website.releases` 可以保留缺少合格元数据的旧版本下载列表，不能因为可以拼出 BIN URL 就开放烧录。网站从固定引用构建后，应直接将该引用交给 `loadReleaseManifest` 选择版本，使界面与固件始终来自同一快照。

`layout: 'single-bin'` 中每个 release 含 `image.source: { url, bytes, sha256, offset }`；URL 同时等于 `image.url` 与完整镜像的 `install.url`，长度/哈希等于 `install.bytes/sha256`。`source.offset` 等于 `image.offset`，应用范围须在完整容器内；`image.bytes/sha256` 仍仅描述应用。Core 一次下载整个 source，在获取 ROM 前拒绝长度/哈希/内部 bootloader、分区表或应用哈希不符，切出应用后复用设备安全、分区检查、写入与写后校验流程。分区布局变化仍拒绝默认升级，不自动改为安装。显式 `mode: 'install'` 写入并校验完整镜像；传入 `data` 也遵循描述，存在 `image.source` 时必须是完整容器，否则沿用原始应用 BIN。继续支持 `legacy` manifest 与旧应用独立描述。下载受合格精确长度及 256 MiB 上限约束，使用 HTTPS、拒绝重定向，暂不使用 HTTP Range。浏览器调用方无需自行下载、切片或实现校验。

固定哈希的 manifest 和完整 BIN source 使用浏览器 HTTP 缓存（`cache: default`）；可变 config 与旧独立应用 BIN 保持 `no-store`。固定资产在 fetch 阶段遇到 TypeError 时，仅一次 `no-store` 重试以恢复缺少 CORS 头的旧缓存。HTTP、读取响应、SHA-256 或元数据错误均不重试、不降级。

10.2 高级导出 API 完整参考

本参考按 package.json exports 覆盖全部公开导入路径和命名导出，列出所有导出类不以下划线开头的公开原型方法。下划线方法及未通过包导出的 src/device_* 实现模块属于内部接口。应用从*应用 API 完整参考* 开始，不应在同一连接并行创建多个所有者/解析器。以下签名使用 JavaScript 参数/默认值语法，异步方法返回 Promise。

10.2.1 @esp-vision/device-core

根入口仅导出 `createDevice(options)`，全部参数与返回设备对象的方法见*应用 API 完整参考*。

10.2.2 @esp-vision/device-core/evmux

EvMuxParser

```
new EvMuxParser(handlers = {})
```

增量字节解析器，处理不完整帧边界、CRC、过滤和 channel 分发。handlers 包括 text、rawText、log、command、other、preview(base64, metadata, bytes)、debugImage(base64, metadata)、rpc(metadata, payload)、filterFrame(metadata, payload)、frameRejected(metadata)。filterFrame 返回 false 时阻止该帧交付。

`EvMuxParser.setLimits({ metadataMax, payloadMax, frameProgressTimeoutMs, frameTotalTimeoutMs } = {})`
: 更新正值 metadata/payload 大小和不完整帧时间限制。

`EvMuxParser.resetReplTextDecoders()`
: 重置 stdout/stderr 的 UTF-8 解码状态。

`EvMuxParser.reset()`
: 清除输入缓冲、不完整帧定时器与统计。

`EvMuxParser.getDiagnostics()`
: 返回 active/rxBytes/rxChunks/bufferedBytes/frames/droppedFrames/invalidImageFrames/lastChannel、incompleteFrameTimeouts，以及独立的 partialFrame 时间快照（null 或以毫秒表示的 startedAt/lastProgressAt/progressDeadline/totalDeadline）。诊断不包含帧载荷，reset 会清除计数和半帧状态。

`EvMuxParser.hasPartialFrame()`
: 判断缓冲中是否存在可能的不完整帧。

`EvMuxParser.feedText(text)`
: 将兼容文本编码后输入解析器，优先使用字节入口。

`EvMuxParser.feedBytes(bytes)`
: 输入任意 Uint8Array 块，允许拆帧、粘帧。

`EvMuxParser.dispatch(metadata, payload)`
: 将已解析 metadata/payload 分发至 channel handler，此入口绕过线上帧校验。

makeEvMuxFrame

```
makeEvMuxFrame({ channel, type = 'data', method = null, payload = '', seq = 0, sid = 'web-ide', sessionId = undefined, vmEpoch = undefined, contentType = undefined, width = undefined, height = undefined, route = undefined, encoding = undefined })
```

返回含 metadata、文本/二进制 payload 和 CRC 的线上 Uint8Array。seq 用于关联，sessionId/vmEpoch 标识所有权代次；编码不发送数据，也不获取会话。

10.2.3 @esp-vision/device-core/evrepl

EV_REPL_FILE_CHUNK_BYTES

兼容 REPL 文件写入器的默认块大小，与 script.write RPC 分块不同。

```
224
```

EV_REPL_INPUT_MAX_BYTES

完整 REPL 输入命令的默认 UTF-8 预算，包括 exec 转义开销。

```
511
```

EvRepl

```
new EvRepl(io)
```

framed MicroPython REPL 协调器。io.writeEvmux(channel, type, method, payload) 必须返回 Promise，并持续输入 stdout。下列公开方法也包含解释器记账辅助操作，应用通常使用 device.python/files。

`EvRepl.execInputBytes(source)`
计算转义后的 REPL exec 命令 UTF-8 字节数，包含命令包装开销。

`EvRepl.createScriptEndSentinel()`
生成并返回唯一脚本完成标记打印命令。

`EvRepl.createScriptRunCommand(path)`
生成含内联完成标记的 execfile 运行命令。

`EvRepl.cancelScriptEndSentinel()`
清除待匹配的完成标记。

`EvRepl.drainScriptEnd()`
消费缓冲的脚本完成标记，通知 onScriptEnd。

`EvRepl.partialMarkerTail()`
查找末尾不完整完成标记以保留缓冲。

`EvRepl.feedStdout(text)`
输入文字，推进提示符/输出等待者。

`EvRepl.drain()`
处理已缓冲的提示符与完成数据。

`EvRepl.failAll(error)`
以错误拒绝排队/当前等待者。

`EvRepl.resetEpoch(error)`
重置解释器 epoch 状态，使旧等待者失效。

`EvRepl.enqueue(operation, { signal = null } = {})`
串行执行 operation，返回其结果 Promise，支持取消。

`EvRepl.writeStdin(data)`
向 repl.stdin 写字符串/Uint8Array。

`EvRepl.execLine(line, { timeout = 8000, signal = null } = {})`
执行准备好的行，等待提示符/输出。

`EvRepl.execPy(source, { timeout = 8000, signal = null } = {})`
将源码转义为 exec 命令，返回收集的输出。

`EvRepl.getDeviceInfo({ signal = null } = {})`
查询平台、实现、板卡及解释器信息。

`EvRepl.getFsStats(path = '/', { signal = null } = {})`
查询路径所在文件系统容量/空闲信息。

`EvRepl.listFs(path, { recursive = false, signal = null } = {})`
返回平铺 {name,path,type,size} 条目，可递归。

`EvRepl.walkFs({ signal = null } = {})`
返回兼容接口使用的嵌套文件树。

`EvRepl.readFile(fn, { timeout = null, chunkTimeout = 30000, signal = null, maxBytes = null } = {})`
返回完整文件 Uint8Array；文件变化、超限或不完整时拒绝。

`EvRepl.touchFile(fn)`
创建/截断一个文件，Promise<void>。

`EvRepl.removeFile(path, { signal = null } = {})`
删除一个文件，Promise<void>。

`EvRepl.removeDir(path, { signal = null } = {})`
删除空目录，Promise<void>。

`EvRepl.makePath(path)`
通过兼容 REPL 辅助方法创建路径。

`EvRepl.writeFile(fn, data, chunk_size = EV_REPL_FILE_CHUNK_BYTES, direct = false, { signal = null } = {})`

兼容 REPL 字节/分块写入器；应用 files.write 使用会话 RPC 写入器。

EvRepl.readSysInfoMD()

返回 Markdown 格式的系统信息。

evReplExecInputBytes

```
evReplExecInputBytes(source)
```

计算转义后的 REPL exec 命令 UTF-8 字节数，包含命令包装开销。

10.2.4 @esp-vision/device-core/session

DeviceSession

```
new DeviceSession({ transport, EvMuxParser, makeEvMuxFrame, EvRepl,
handlers = {}, sid = 'ide-client', sleep = defaultSleep, })
```

底层 v4 解释器，注入 transport、EvMuxParser、EvRepl、makeEvMuxFrame；handlers 接收 text/rawText/log/preview/rpc/diagnostic 及会话生命周期回调。调用方须订阅传输输入并喂入字节，构造类本身不打开连接；应用入口 createDevice 已封装这些生命周期操作。

DeviceSession.feedBytes(bytes)

向会话解析器输入接收字节。

DeviceSession.feedText(text)

输入兼容文本。

DeviceSession.hasFeature(feature)

判断协商能力是否声明指定 feature。

DeviceSession.writeFrame(channel, type, method, payload, { authorize = null, vmRequest = null, seq: requestedSeq = null, signal = null, writeTimeout = null, } = {})

按当前会话/epoch 编码发送帧，返回传输 Promise。

DeviceSession.rpc(method, payload = {}, { timeout = 4000, channel = null, signal = null, binaryResponse = false } = {})

发送关联 RPC，等待 JSON 或 binaryResponse；超时、取消、固件错误会拒绝。

DeviceSession.captureImage(method, payload = {}, options = {})

调用抓图方法，返回 JPEG {data,mimeType,metadata}。相机方法（包括直接 `rpc` 和请求 `writeFrame`）要求 `debug.capture_frame.observed.v1`，旧的不安全固件会在发送字节前被拒绝。`CAMERA_FRAME_NOT_READY` 和 `NO_MEM` 保留为明确固件错误，不另行物理采帧。

DeviceSession.negotiate(options = {})

探测 v4、校验身份/限制、获取所有权，可验证 REPL。

DeviceSession.writeFile(path, content, { encoding = 'utf-8', mode = 'overwrite', runAfterWrite = false, signal = null, onProgress = null, } = {})

通过统一的 file.write 二进制引擎写入 UTF-8/字节，完成 SHA-256 提交后返回；要求 file.write.binary，不回退旧上传路径。

DeviceSession.writeScript(path, content, { runAfterWrite = false, signal = null } = {})

writeFile 的 UTF-8 兼容封装。

DeviceSession.createOperationId()

生成新的网络修改 operation ID。

DeviceSession.networkStatus(options = {})

返回 network.status 状态/结果。

DeviceSession.configureNetwork({ ssid, password, operationId = undefined }, options = {})

发送凭据和 operationId，返回 accepted ACK，不代表最终联网成功。

DeviceSession.setNetworkEnabled(enabled, options = {})

发送持久化网络开关修改。

DeviceSession.forgetNetwork(options = {})
请求清除持久化凭据。
DeviceSession.release({ timeout = 1200 } = {})
释放 EV-MUX 会话租约，Promise 返回 RPC 结果。
DeviceSession.executePython(source, { timeout = null, signal = null } = {})
通过 framed REPL 执行源码并返回输出。
DeviceSession.interrupt()
通过 framed signal 通道发送 Ctrl-C。
DeviceSession.control(operation, { timeout = 15000, signal = null } = {})
执行 stop/soft-reset/hard-reset/bootloader 控制。
DeviceSession.stopForRepl({ timeout = 5000, signal = null } = {})
中断并等待可执行 REPL，返回就绪布尔值。
DeviceSession.runScript(path, content, { timeout = 10000, signal = null, onStarted = null, maxOutputChars = Number.POSITIVE_INFINITY, trial = null, inlineCompletion = false, } = {})
执行设备脚本，可先上传 content，等待有界完成。
DeviceSession.startScript(path, content, { onStarted = null, ...options } = {})
启动脚本，返回含 done Promise 的句柄。
DeviceSession.scriptRunSnapshot()
返回当前运行快照或 null。
DeviceSession.snapshot()
返回会话状态、role、能力、parser/transport 统计和待处理 RPC 数量。
DeviceSession.close(error = new Error('connection closed'))
拒绝待处理工作，重置会话状态/定时器；物理连接由宿主关闭。

可读取的协商状态包括 state、role、identity、capabilities、sessionId、vmEpoch、vmReady、limits、parser、repl；应通过会话方法更新，不直接赋值。`negotiate` 选项为 role=”unknown”、drainMs=300、acquire=true、probeRepl=true、expectedDevice=null、signal=null，TCP 跳过启动 drain。串口可在 parser 剩余接收预算内补齐跨窗口的半帧，期间继续拒绝旧帧，后续新帧不能延长最初半帧的总截止时间。注入的旧 parser 若不支持接收预算，则保留半帧连接失败的保守行为，不进行无界等待。`control` 接受 stop/soft-reset/hard-reset/bootloader 字符串，不是应用对象的请求结构。底层网络修改返回 accepted ACK，应用 configure 才会等到对应事务完成。使用 typed run trial 时要求 operation_id 为 UUIDv4、source_path 不超过 512 个 UTF-8 字节、source_hash 为 64 位小写 SHA-256，并要求固件声明 launcher trial/typed-runner 能力；trial 不能与 inlineCompletion 组合。

EV_MUX_DEFAULT_LIMITS

协议回退限制，实际操作使用协商的固件 limits；txLargePayloadMax=0 是初始回退值，不表示所有设备只能发送零字节图像。

```
{
  "metadataMax": 384,
  "rxPayloadMax": 2048,
  "replInputMax": 511,
  "vmPendingMax": 1,
  "txLargePayloadMax": 0,
  "txFrameTimeoutMs": 1500,
  "txQueueTimeoutMs": 2000
}
```

EV_MUX_PATH_MAX_BYTES

设备路径 UTF-8 字节限制，已考虑临时文件名余量。

```
123
```

EV_MUX_PAYLOAD_LIMIT

兼容 payload 预算常量，不替代协商的 RX 限制。

```
1800
```

EV_MUX_REQUIRED_CHANNELS

v4 能力校验要求的 channel 集合。

```
[
  "user.rpc",
  "debug.rpc",
  "repl.stdin",
  "repl.stdout",
  "repl.stderr",
  "repl.signal",
  "log.idf",
  "preview.frame"
]
```

EV_MUX_REQUIRED_FEATURES

v4 能力校验要求的 feature 集合。

```
[
  "capabilities",
  "route.changed",
  "framed.repl",
  "session.firstCome",
  "session.keepalive"
]
```

EV_MUX_SCRIPT_CHUNK_BYTES

旧 script.write 的参考常量；SDK 0.3.0 上传不再使用，二进制块长取自 fileWriteChunkMax。

```
900
```

EV_MUX_VERSION

支持的会话协议版本。

```
4
```

evmuxRpcChannelForMethod

`evmuxRpcChannelForMethod(method)`

debug.* 方法返回 debug.rpc，其他返回 user.rpc。

requiresEvmuxFirmwareUpgrade

`requiresEvmuxFirmwareUpgrade(err)`

判断是否为需要升级固件的错误，返回布尔值，包括版本不匹配或固件升级标记。

validateEvmuxCapabilities

`validateEvmuxCapabilities(payload)`

校验 v4 版本、必需 features/channels，并校验会话租约，返回 void；不匹配或缺失能力时抛错。

10.2.5 @esp-vision/device-core/actions

DEVICE_ACTION_DESCRIPTORS

下列 17 个描述定义宿主动作契约，不代表 Core 已实现平台端口枚举、连接 UI 或 workspace 部署；这些 handler 须显式绑定。输入以各描述的 inputSchema 为准，outputSchema 允许 handler 自定义附加字段，risk 含 readOnly/destructive/openWorld 标志。

`device.listPorts`

List platform-native serial candidates for the user to select; this action never chooses a device.


```
{
  "inputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": false,
  "concurrency": "shared-read"
}
```


`device.connect`

Connect to an ESP Vision device using standard elicitation for a serial port or an explicit device TCP address, then acquire an EV-MUX v4 session.


```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "transport": {
        "enum": [
          "serial",
          "tcp"
        ],
        "default": "serial",
        "description": "Use serial for a platform port or tcp for an IP address on␣
↪the device network."
```


(续下页)

(接上页)

```
      },
      "port": {
        "type": "string",
        "description": "Platform-native serial identifier, such as /dev/ttyACM0 or␣
↪/dev/cu.usbmodem*. Omit it only when the MCP host advertises elicitation or ESP_
↪VISION_PORT is configured. Non-interactive callers must pass it explicitly. A␣
↪direct network address may also be supplied as tcp://IP:PORT or IP:PORT;␣
↪alternatively set transport to tcp and provide host and tcpPort."
      },
      "host": {
        "type": "string",
        "description": "IPv4 or IPv6 address of the device when transport is tcp."
      },
      "tcpPort": {
        "type": "integer",
        "minimum": 1,
        "maximum": 65535,
        "default": 3232,
        "description": "Device EV-MUX TCP port."
      },
      "baudRate": {
        "type": "integer",
        "minimum": 1200,
        "maximum": 4000000,
        "default": 115200
      },
      "drainMs": {
        "type": "integer",
        "minimum": 0,
        "maximum": 3000,
        "default": 300
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": false,
    "openWorld": true
  },
  "requiresReadyDevice": false,
  "concurrency": "exclusive"
}
```

`device.disconnect`

Disconnect the active device session and release its lease.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
```

(续下页)

(接上页)

```
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": false
  },
  "requiresReadyDevice": false,
  "concurrency": "exclusive"
}
```

`device.negotiate`

Discover EV-MUX v4, explicitly acquire a session, and prove the framed REPL is live.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "role": {
        "type": "string",
        "default": "unknown"
      },
      "drainMs": {
        "type": "integer",
        "minimum": 0,
        "maximum": 3000,
        "default": 300
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": false,
  "concurrency": "exclusive"
}
```

`device.inspect`

Inspect protocol state, device identity, filesystem, and recent logs.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "includeFiles": {
        "type": "boolean",
        "default": true
      },
      "logLimit": {
        "type": "integer",
        "minimum": 0,
```

(续下页)

(接上页)

```
        "maximum": 500,
        "default": 100
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "shared-read"
}
```

`python.execute`

Execute MicroPython through the shared interpreter using bounded REPL or binary file staging.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "code": {
        "type": "string"
      },
      "timeoutMs": {
        "type": "integer",
        "minimum": 100,
        "maximum": 120000,
        "default": 8000
      }
    },
    "required": [
      "code"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`python.runEphemeral`

Run one Python draft from the reserved temporary device path without modifying a user-owned file.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "code": {
        "type": "string"
      },
      "maxDurationMs": {
        "type": "integer",
        "minimum": 1000,
        "maximum": 180000,
        "default": 180000
      }
    },
    "required": [
      "code"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`python.observeEphemeral`

Observe the bounded output and status of the current temporary Python run.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "runId": {
        "type": "string"
      },
      "waitMs": {
        "type": "integer",
        "minimum": 0,
        "maximum": 10000,
        "default": 1000
      },
      "outputOffset": {
        "type": "integer",
        "minimum": 0,
        "maximum": 1048576,
        "default": 0
      }
    },
    "required": [
      "runId"
    ],
    "additionalProperties": false
  },
```

(续下页)

(接上页)

```
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "shared-read"
}
```

`python.stopEphemeral`

Stop the current temporary Python run and remove its reserved device file.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "runId": {
        "type": "string"
      }
    },
    "required": [
      "runId"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceFile.read`

Read a file from the device filesystem.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "path": {
        "type": "string"
      },
      "encoding": {
        "enum": [
          "utf-8",
          "base64"
        ],
        "default": "utf-8"
```

(续下页)

(接上页)

```
      }
    },
    "required": [
      "path"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceFile.write`

Write a file to the device using the negotiated transfer path.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "path": {
        "type": "string"
      },
      "content": {
        "type": "string"
      },
      "encoding": {
        "enum": [
          "utf-8",
          "base64"
        ],
        "default": "utf-8"
      },
      "runAfterWrite": {
        "type": "boolean",
        "default": false
      }
    },
    "required": [
      "path",
      "content"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
```

(续下页)

(接上页)

```
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`workspaceFile.deploy`

Stream one workspace-relative file to the device without placing binary content in the model context.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "sourcePath": {
        "type": "string",
        "description": "Workspace-relative source file path."
      },
      "devicePath": {
        "type": "string",
        "description": "Absolute destination path on the device."
      },
      "expectedBytes": {
        "type": "integer",
        "minimum": 0,
        "maximum": 16777216
      },
      "expectedSha256": {
        "type": "string",
        "description": "Optional lowercase SHA-256 expected before transfer."
      },
      "overwrite": {
        "type": "boolean",
        "default": false
      }
    },
    "required": [
      "sourcePath",
      "devicePath"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceFile.remove`

Remove one file or one empty directory from the device filesystem.

```
{
  "inputSchema": {
```

(续下页)

(接上页)

```
    "type": "object",
    "properties": {
      "path": {
        "type": "string"
      },
      "kind": {
        "enum": [
          "file",
          "directory"
        ],
        "default": "file"
      }
    },
    "required": [
      "path"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceTest.run`

Write, run, and capture a target-side MicroPython test.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "path": {
        "type": "string",
        "default": "/.agent_test.py"
      },
      "source": {
        "type": "string"
      },
      "timeoutMs": {
        "type": "integer",
        "minimum": 500,
        "maximum": 120000,
        "default": 10000
      }
    },
    "required": [
      "source"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
```

(续下页)

(接上页)

```
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`device.control`

Interrupt, reset, or enter the bootloader.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "operation": {
        "enum": [
          "stop",
          "soft-reset",
          "hard-reset",
          "bootloader"
        ]
      },
      "confirmBootloader": {
        "type": "boolean",
        "default": false
      }
    },
    "required": [
      "operation"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`device.rpc`

Call a negotiated user or debug RPC method. For network.configure, network.enable, and network.forget, include a 32-character lowercase hexadecimal operationId and reuse the exact same ID and payload when retrying after a timeout.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
```

(续下页)

(接上页)

```
      "method": {
        "type": "string"
      },
      "payload": {
        "type": "object",
        "default": {}
      },
      "channel": {
        "enum": [
          "user.rpc",
          "debug.rpc"
        ]
      },
      "timeoutMs": {
        "type": "integer",
        "minimum": 100,
        "maximum": 120000,
        "default": 4000
      }
    },
    "required": [
      "method"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`frame.capture`

Capture one JPEG frame for an explicitly requested observation.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "timeoutMs": {
        "type": "integer",
        "minimum": 500,
        "maximum": 30000,
        "default": 8000
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
```

(续下页)

(接上页)

```
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

DEVICE_ACTION_SCHEMA_VERSION

宿主动作描述的 schema 版本。

```
1
```

DEVICE_ACTORS

宿主动作层允许的 actor 名称，普通设备对象不要求 actor。

```
[
  "agent"
]
```

DeviceActionError

```
new DeviceActionError(code, message, { recovery = null, details = null,
cause = null } = {})
```

结构化宿主动作错误，含 code/message 和可选 recovery/details/cause；toJSON 不序列化 cause。

DeviceActionError.toJSON()

返回 {code,message,recovery?,details?}。

DeviceActionGateway

```
new DeviceActionGateway({ registry = new DeviceActionRegistry(),
coordinator = new OperationCoordinator(), lease = new DeviceLease(),
getSession = () => null, authorize = async () => true, } = {})
```

校验动作输入/context，检查宿主租约与 ready 会话，调用 authorize(descriptor, input, context)，再协调已绑定 handler。context.actor 必须受支持，context.requestId 必填，context.signal 支持取消。宿主须提供实际授权策略，默认 authorize 返回 true。

DeviceActionGateway.invoke(actionName, input = {}, context = {})

Promise：返回校验/授权后 handler 的结果；错误附动作上下文。

DeviceActionRegistry

```
new DeviceActionRegistry(descriptors = DEVICE_ACTION_DESCRIPTORS, {
schemaVersion = DEVICE_ACTION_SCHEMA_VERSION } = {})
```

动作描述与显式绑定 handler 的注册表。描述要求 name、inputSchema、outputSchema、risk。未绑定描述仅是元数据，Gateway.invoke 会以 ACTION_UNAVAILABLE 拒绝调用；register/bind 返回 this，可链式调用。

DeviceActionRegistry.register(descriptor, handler = null)

注册描述与可选 handler。

DeviceActionRegistry.bind(name, handler)
为已有动作绑定 handler。
DeviceActionRegistry.descriptor(name)
返回描述或 null。
DeviceActionRegistry.handler(name)
返回 handler 或 null。
DeviceActionRegistry.list()
返回描述数组。
DeviceActionRegistry.validate(name, input)
校验输入、应用默认值，返回校验后的副本。

DeviceLease

```
new DeviceLease()
```

宿主侧 actor 租约，与固件 EV-MUX 会话租约不同。actor 必须属于 DEVICE_ACTORS，同 actor 的多个句柄按数量维护；它不获取或强制抢占设备会话。

DeviceLease.acquire(actor)
返回 {id,actor,release}，不同 actor 冲突时抛错。
DeviceLease.release(id)
释放一个宿主租约 ID。
DeviceLease.assertAccess(actor)
actor 未持有宿主租约时抛错。
DeviceLease.snapshot()
返回 {state,primary,actors,holders}。

DeviceLeaseConflictError

```
new DeviceLeaseConflictError(owner, requester)
```

宿主 actor 与当前租约持有者冲突时的 DeviceActionError。

OperationCoordinator

```
new OperationCoordinator()
```

支持取消与关闭处理的宿主串行队列。此适配层会排队，设备对象自身则拒绝冲突；operation 接收对应 AbortSignal。

OperationCoordinator.run({ actor, requestId, actionName, signal }, operation)
将 operation 排队，返回结果 Promise。
OperationCoordinator.close(reason = actionAbortError('Device session closed'))
拒绝排队工作并取消当前工作，停止接受新操作。
OperationCoordinator.snapshot()
返回当前及排队的 actor/request/action 身份。

10.2.6 @esp-vision/device-core/bindings

DeviceEvidenceBuffer

```
new DeviceEvidenceBuffer(limit = MAX_EVIDENCE_ENTRIES)
```

有界宿主证据历史，构造默认上限 500，snapshot 默认 100；值会限长，应用仍须处理敏感信息。

DeviceEvidenceBuffer.append(kind, value, details = null)
追加带时间戳、限长的值及可选 details。

DeviceEvidenceBuffer.snapshot(limit = 100)
返回最近若干条记录。
DeviceEvidenceBuffer.clear()
清空历史。

EPHEMERAL_PYTHON_PATH

高级临时运行控制器的保留路径；应用 source/start 使用自己的唯一临时路径。

```
"/.espvision_agent_run.py"
```

EphemeralPythonController

```
new EphemeralPythonController(session, { evidence = null, generation =
'session' } = {})
```

每个控制器管理一个有时限的临时 Python 运行，使用 EPHEMERAL_PYTHON_PATH；session 为 DeviceSession。快照含 runId/path/status/output/nextOffset/finished/cleaned，可附 error/outputTruncated/cleanupError/residualPath；启动成功不代表清理完成。

EphemeralPythonController.start(code, { maxDurationMs = 180000, signal = null } = {})
启动有时限的临时源码运行，返回快照，完成后清理。
EphemeralPythonController.observe(runId, { waitMs = 1000, outputOffset = 0, signal = null } = {})
最多等待 waitMs，从 outputOffset 返回增量输出。
EphemeralPythonController.stop(runId)
停止指定运行，返回清理/状态快照。
EphemeralPythonController.close()
控制器关闭时停止/清理当前运行。

bindSessionDeviceActions

```
bindSessionDeviceActions(registry, { evidence = new DeviceEvidenceBuffer(),
lease = null, getOperations = () => null, getGeneration = () => 'session',
} = {})
```

向注册表绑定通用会话/文件/Python/抓图 handler，返回 { evidence, controllerFor }。此函数不提供宿主专用的 listPorts/connect/disconnect/workspace 部署实现，须由宿主绑定。

10.2.7 @esp-vision/device-core/firmware

按地址操作 ROM Flash

公开的 `openNodeRom` 和 `openBrowserRom` 适配器也支持自行管理 Flash 布局的调用方。先 `initialize()`，核对选中芯片/修订版及 `getSecurityInfo()`，再 `runStub()`；只在 stub 就绪后操作，并在清理中 `disconnect()`。以下操作按地址执行，不限定为 FAT 分区：

- `stub.eraseRegion(address, size)` 擦除调用方给定范围，起始地址和正长度须按 4096 字节对齐，且在 32 位地址空间内。
- `stub.flashData(arrayBuffer, onProgress, offset)` 通过官方烧录器把给定字节写到指定地址。
- `stub.readFlash(offset, byteCount, onProgress?, options?)` 读取指定范围。`options` 可指定 `chunkSize`、`blockSize`` (1024 或 4096)、``maxInFlight`；大范围校验应以不超过 65536 字节分段、1024 字节包和单个未确认包循环读取。
- `stub.flashId()` 返回整数 JEDEC ID，位 16–23 为容量指数，不是容量字节数；调用方须核对范围不超过实际 Flash，并负责分区选择、范围重叠判断、数据哈希及读回校验。

跨 Node/浏览器使用时，每个适配器实例应只调用一次 `flashData()`。浏览器适配器会以 `ROM_WRITE_REUSE` 拒绝第二次调用并关闭会话。多段数据应合并为一个连续 payload，并明确间隙内容；或每次写入前重新建立并校验 ROM 会话。结果不确定时，不要重放写操作。

自定义分区工具可以直接组合这些通用操作。`FirmwareFlashSession` 在其上提供受认可镜像的完整安装/应用更新、完整校验与芯片复位处理。Core 0.7.0 增加的是此前缺失的跨平台范围擦除能力，以及完整安装清理内部 FAT 的行为变化；没有新增写死的 FAT 地址。安装流程解析受认可的分区表，因此仅移动分区地址不需要再次发布 Core。只更新应用仍保留用户文件。

`stub.flashData()` 仅在官方 esptool 已成功核对设备端整段摘要后，可返回 `{ algorithm: 'md5', offset, bytes, digest }`；digest 为 32 位小写十六进制串。Core 将地址、长度和摘要与通过 SHA-256 校验的源数据比较；凭据无效或不匹配则拒绝，不复位、不重试。返回 null/undefined 时保留旧的完整回读。下载源和安装镜像内部各段仍必须通过 SHA-256 校验；MD5 用于检测烧写错误，不替代 SHA-256 的密码学保证。

FirmwareFlashSession

```
new FirmwareFlashSession(loader, image, { reset = () => loader.hardReset(),
mode = 'update', signal? } = {})
```

ROM 烧写器适配，与 EV-MUX 运行态会话分开，每个实例只使用一次。state 为 idle → connecting → connected → flashing → verified → closed，失败可经 error → closed；flash 最终会断开，单凭 closed 不能判定校验成功。

`FirmwareFlashSession.connect(bytes)`
: 写入前校验镜像、芯片/版本/安全模式和现有分区表哈希。

`FirmwareFlashSession.flash(bytes, { onProgress = () => {}, onPhase = () => {} } = {})`
: 写应用分区，完整读回并核验 SHA-256，然后复位；最终断开。onProgress 为 0..100，onPhase 接收状态文字。

`FirmwareFlashSession.disconnect()`
: 关闭 loader 一次，重复调用不重复关闭。

镜像描述包含 evMuxVersion=4、bytes、offset、partitionSize、sha256、chipFamily、chipRevision、partitionTable:{offset,bytes,sha256}。须提供可信构建元数据与精确 Uint8Array；默认模式不执行整片擦除或分区迁移。注入的 loader 提供 initialize()、chipFamily/chipRevision、getSecurityInfo()、runStub(true)、hardReset()、disconnect()；stub 提供 flashData()、readFlash()、readRegister()、writeRegister()。内置 P4/S31 正常启动锁存处理，使用各芯片自己的寄存器地址。写后校验通过后的复位失败由 onPhase 提示，不会撤销校验或自动重写。

`mode: 'install'` 使用已校验的完整镜像，从实际分区表解析可写、未加密的 `ffat` 范围；仍检查芯片/修订、安全模式、JEDEC 容量、范围和分区完整性。stub 提供 `flashId()`、`eraseRegion(address, size)`、`flashData()` 和 `readFlash()`。先擦除按扇区对齐的 FFAT 范围，擦除命令成功即进入固件写入，不回读、抽查或计算擦后摘要。擦除失败或在写入前取消会拒绝，不复位、不重放。布局无效返回 FIRMWARE_PARTITION_INVALID 或 FIRMWARE_FATFS_UNAVAILABLE；缺能力返回 FIRMWARE_ERASE_UNSUPPORTED。不整片擦除、不备份、不自动恢复。删除语义见*应用 API 完整参考*。

`firmwareMode(mode = 'update')` 校验并返回 update/install，其他值抛出 TypeError。`validateFirmwareImage(image, bytes, { mode = 'update' } = {})` 在 install 模式下还会校验嵌入的 bootloader、分区表和应用范围及哈希。

firmwareSha256

```
firmwareSha256(bytes)
```

Promise<string>，返回小写 SHA-256 十六进制摘要，要求 Web Crypto。

validateFirmwareImage

`validateFirmwareImage(image, bytes, { mode = 'update' } = {})`

Promise<Uint8Array>，校验 v4、精确字节数、应用分区边界和 SHA-256，返回输入字节；不推断或下载可信构建清单。

10.2.8 @esp-vision/device-core/node

`listSerial({ signal, timeout = 10000 } = {}, dependencies = {})`
通过 npm 原生绑定被动枚举系统串口，无需 Python，也不打开端口。

`filterSerialPorts(ports, { platform = process.platform, allPorts = false } = {})`
按平台路径及 USB 描述符提示筛选系统枚举记录，不打开端口，也不认定固件身份。

`serialEntry(port)`
将串口描述符转换为未探测的接口/地址候选并附带 USB 元数据；描述符分类不等于设备身份。

`transportFor(entry, options = {})`
为枚举项创建被动 TCP 或串口字节传输。

`inventory(options = {}, dependencies = {})`
从串口枚举和局域网发现返回 `{ devices, warnings, sourcesAvailable }`。默认只扫描，不连接；`probe: true` 会连接每个候选。可用 `timeout`、`interfaces`、`allPorts` 和 `signal` 控制扫描。

被动枚举原生 USB ROM 接口时返回 `status: rom`、`identity: null`，不打开端口、不复位。主动 `probe: true` 检测先核对 ROM 出厂 MAC 和安全模式，清除对应芯片的软件强制下载标记，再尝试一次正常 SPI 启动；不加载 stub、不擦写或读取 Flash。只对 MAC 和物理 USB 位置均匹配的唯一端口重新进行运行态握手。`romProbe` 记录 `attempted`、`outcome``（``runtime`、`rom` 或 `no-runtime`）、目标身份及复位/清理诊断；设备已经回到运行态时不再复位。没有运行端口返回时标为 `unidentified` 并报告 ROM_RUNTIME_NOT_FOUND，重新返回 ROM 时仍标为 `rom`，两者均不直接断言固件有无。应用不可启动时，安装前可能需要通过 BOOT/RESET 重新进入下载模式。只有成功的 ESP-VISION 握手才能确认固件身份；串口或会话占用返回 `busy`。

Core 0.8 移除 `defaultPython`、`nodeAdapterAssets` 与 Python 辅助文件。打包器须保留 npm 原生加载器和预编译文件，参考 CLI 构建脚本。

`tcpTransport({ host, port })`
返回被动构造的 Node TCP 字节适配器。由 Core 在 `device.connect({ role: 'tcp' })` 时打开。

`serialTransport({ serial })`
返回 npm 原生串口字节适配器。显式指定端口及 `usj`、`usb` 或 `uart` 角色，由 Core 打开、关闭并管理设备会话。

`openNodeRom({ port, signal?, logger?, baudRate?, onStatus? })`
返回内置 ROM 适配器，创建时不打开端口。用于 `device.firmware.update({ openLoader: ({ signal }) => openNodeRom({ port, signal }), ... })`，安装与应用更新共用。适配器与浏览器共用 esptool-js 引擎，不再需要主机 Python。Core 负责初始化、校验、写入及写后校验、复位与释放。SDK 0.3.0 新增，仅用于 Node；区别于运行态 serialTransport，也不是 TCP OTA。

10.2.9 @esp-vision/device-core/discovery

ESP_VISION_DISCOVERY_SERVICE

Node provider 查询的 DNS-SD 服务名。

```
"_esp-vision._tcp.local"
```

NetworkDiscovery

`new NetworkDiscovery(provider)`

包装含 list({timeout, signal}) 的 provider，校验结果并按主机名排序；list 与 device.discover 使用相同限制及结果结构，构造时不扫描。

`NetworkDiscovery.list({ timeout = 3000, signal } = {})`
在发现时间窗口内收集规范记录，支持 AbortSignal。

createBridgeDiscovery

`createBridgeDiscovery(connectionUrl, { fetch: fetcher = globalThis.fetch } = {})`

返回基于桥接 GET /devices?timeout=…的 NetworkDiscovery；接收 ws/wss connectionUrl，可注入 fetch。结果中的 connectionUrl 须同源且为 /device/<32-hex> 路径。

normalizeDiscoveredDevice

`normalizeDiscoveredDevice(value)`

返回规范记录或抛错：12 位小写 MAC 十六进制、DNS 主机名不超过 63 字符、board 不超过 96、firmwareVersion 不超过 255、整数 evMuxVersion、端口 1..65535、至少一个合法 IPv4；地址去重，compatible 表示 v4。

10.2.10 @esp-vision/device-core/discovery/node

createNodeDiscovery

`createNodeDiscovery({ createMdns, interfaceAddress } = {})`

返回使用 _esp-vision._tcp.local 的 PTR/SRV/TXT/A 记录的 NetworkDiscovery。createMdns 可注入兼容 multicast-dns 的工厂，interfaceAddress 选择网卡 IPv4；否则动态导入可选依赖 multicast-dns。仅接受私网/链路本地 IPv4 候选，遵守 TTL，每轮扫描后关闭 socket。

10.2.11 使用边界

子路径导出是实际实现或宿主契约，不是所有应用请求的别名。同一物理设备保持一个会话协调入口；设备所有权、宿主 actor 租约、ROM 烧写生命周期和发现扫描含义不同。普通设备交互使用应用对象，仅在实现对应宿主层时选用子路径。

10.2.12 FirmwareFlashSession.flash 返回值

写后校验后，flash 返回 `{ verified: true, needsReset: boolean }`。复位失败时 needsReset 为 true；仅凭 closed 状态不能证明校验成功。应用通常使用 `device.firmware.update()` 完成发布目录校验与端口交接。

10.2.13 @esp-vision/device-core/browser

`openBrowserRom({ port, logger = {}, signal?, expectedMac? })`
SDK 0.6.0 包含此入口。为用户选择的 Web Serial `SerialPort` 返回被动 ROM 适配器；应在用户手势内申请端口，先释放已有运行时连接。`initialize()` 延迟加载官方 `esptool-js@0.7.1`，打开端口、识别芯片并提供 `chipName/chipFamily/chipRevision`，不启动 stub。适配器

实现 `initialize/getSecurityInfo/runStub/hardReset/disconnect` 供 `firmware.update` 使用；Core 校验镜像与 ROM 安全模式后才启动 stub，统一负责写入、写后校验和复位。`logger` 可提供 `log/error/debug`；`signal` 取消初始化和范围擦除；Core 还在擦除后及开始写固件前检查取消。固件写入一旦开始，仍按原策略完成校验与复位。`disconnect()` 幂等释放端口，初始化失败也清理资源；关闭后不可复用。须使用浏览器打包器，不支持直接加载上游原生 ESM。完整示例见*Device Core npm 包对接*。此入口不加载 Node 模块；Node 通过自身串口适配器使用同一个 ROM 引擎。

Core 0.7.0 另提供 `chipMac`：资格确认前为 `null`，之后为与 Node 适配器一致的 12 位小写十六进制 factory MAC，由固定版本官方芯片的 `readMac(loader)` 原语读取。可选 `expectedMac` 只接受恰好 12 位十六进制字符（大写归一为小写）；分隔符、空白、`null` 及非字符串值会在访问端口前同步拒绝。宿主须获取选中运行态设备经过核验的 factory MAC，例如已验证与该板 factory MAC 一致的 `machine.unique_id()`；任意历史 `deviceId` 或 USB VID/PID 都不能证明这种关联。无法核验时应使用用户明确选择的手动 ROM 流程，不能推测自动关联。首次不匹配，或恢复同步发现芯片家族、修订、MAC 改变时，返回 ROM_IDENTITY_CHANGED、关闭适配器，不再启动另一个 stub、擦除或写入；MAC 返回值不完整时返回 ROM_IDENTITY_INVALID。USB-OTG 保留这些错误码，不替换成手动下载提示。不传此选项仍支持手动选择首次 ROM，但后续同步继续绑定首次设备。

10.2.14 @esp-vision/device-core/firmware 发布读取器（0.5.1）

`loadOfficialFirmware({ board?, version?, fetch = globalThis.fetch, signal? })`

通过 publication 引用加载官方当前或精确指定版本的快照，返回 manifest 及 publication 历史；可选板型筛选前校验整份快照。无 publication 的配置保留旧元数据兼容。

`loadReleaseManifest({ url, sha256, version?, sourceCommit?, boards?, fetch = globalThis.fetch, signal? })`

只加载一份不可变官方 manifest，校验原始字节 SHA-256、身份、完整板集、所有固件描述及 website/model/MCP 来源边界，返回 manifest 对象，不下载固件也不打开 ROM。字段、限制与单 BIN 升级行为见*应用 API 完整参考*。

10.2.15 浏览器烧录后的启动

浏览器 ROM 适配器在校验后通过当前运行的 flash stub 复位。写入校验成功不代表固件已经启动：宿主必须等待新的运行态握手后才允许配网。若 `needsReset` 为真，`resetError` 会保留复位失败的代码和消息；手动复位只作为异常恢复方式。

Node ROM 回读安全边界

Node 与浏览器 Flash 读取均只允许一个未确认的 1024 字节包，并校验每个有界命令的尾部 MD5。Node 保留大范围读取能力，内部拆分为至多 64 KiB 的命令；旧读包与窗口参数作为预算上限。写入前的分区表校验使用同一路径。读取不完整、超时或摘要错误均失败，不自动重放。

若先前已观察到的原生 ROM 设备正在重新枚举，只读初始化失败后 Core 会被动等待至多五秒。只有工厂 MAC 与物理 USB 位置均相同、且唯一出现的运行态设备才可收到一次经过身份验证的 bootloader 请求。身份歧义或冲突不会触发引导切换；该恢复不会重复已经提交的 bootloader 请求或 Flash 读写。

10.3 Device Core npm 包对接

完整的 SDK 介绍与 API 索引见*device-core/index*，应用 API 参数、返回值和事件见*device-core/api*，所有高级导出见*device-core/advanced*。本页保留可直接运行的接入示例。

本页面向把设备接入 IDE、Device CLI、Agent 工作流或自定义主机应用的开发者。`@esp-vision/device-core` 是无 UI 的共享通信包；主机只提供物理字节连接，不需要复制参考 Python 客户端

或实现第二套协议。Agent 设备操作使用 CLI；Knowledge MCP 通过 HTTP 提供文档。完整帧格式见*communication/wire-protocol*，物理接口见*communication/interfaces*。

10.3.1 源码、版本与安装

Device Core 源码在 ESP-VISION 固件仓库的 `host/device-core/` 维护。`host/` 表示运行在主机的代码，不是固件模块；IDE 和自定义应用直接使用 SDK，Agent 设备控制工作流使用单独打包的 Device CLI。协议变更应在同一个 MR 对齐固件、Core、文档与测试。

包使用独立版本，不与固件版本或 IDE 版本绑定。本页示例面向 `@esp-vision/device-core@0.6.0`，只支持 EV-MUX v4；v3 返回 `VERSION_MISMATCH`，应用应提示更新固件。依赖锁定到 minor，并在 lockfile 记录解析后的 patch；安装前请在公开 npm registry 确认所选版本；dry run 或创建 MR 不等于发布。

```
npm view @esp-vision/device-core@0.6.0 version dist.integrity
npm install '@esp-vision/device-core@~0.6.0'
```

Node 主机基线是 Node.js 22。浏览器需要打包器或 import map 解析包名，以及 Web Serial 或二进制 WebSocket 桥。Device Core 不附带串口驱动、网络桥、模型或 IDE UI。Node 串口枚举使用可选 npm 原生绑定；Node mDNS 发现使用 `multicast-dns`。单独的 Device CLI 以 `@esp-vision/device-cli` 安装，是 Agent 设备操作的受支持命令。固定 npm 版本并更新 lockfile。

10.3.2 应用、Core 与传输的职责

本图与项目 README、系统架构章节复用同一 SVG。主机应用负责设备选择及界面/流程策略，Device Core 负责共享 API、协议和会话状态，EV-MUX 定义与固件的运行时通信；字节适配器提供串口/socket I/O，不维护另一套协议栈。图中代表性 API 不替代完整参考；发现与 ROM 烧写各有独立生命周期，ROM loader 不通过 EV-MUX 写镜像。

每个设备连接创建一个 `createDevice({ transport })` 对象；根入口只导出这个工厂函数。Core 负责解析、关联请求、会话、保活、REPL 协调和操作生命周期。应用负责设备选择、展示与业务策略；传输适配器负责打开连接及可靠收发原始字节，不获取 EV-MUX 所有权、不重编码协议。

传输契约只有两个必需方法：`writeBytes(Uint8Array): Promise` 在全部字节被宿主连接接受后完成，失败时 reject；`subscribe({ data, error, close })` 转发原始字节、错误与断线，且必须同步返回取消订阅函数。可选 `connect({ signal })` / `disconnect()` 打开、关闭物理连接。Core 先订阅再连接、握手；已打开的连接可以省略 `connect`。短写或 WOULD_BLOCK 不能当成完整发送，旧连接退订后不能继续输入数据。

10.3.3 独立 Node TCP 示例

将以下内容存为 `core-tcp.mjs`，在安装包的项目运行 `node core-tcp.mjs <device-ip>`。设备需已联网并支持 v4，TCP 默认端口为 3232。该示例使用命令行指定的设备地址，发现流程见后文。

```
import net from 'node:net'
import { once } from 'node:events'
import { createDevice } from '@esp-vision/device-core'

const host = process.argv[2]
if (!host) throw new Error('Usage: node core-tcp.mjs <device-ip>')
const socket = new net.Socket()
const transport = {
  subscribe({ data, error, close }) {
    socket.on('data', data)
    socket.on('error', error)
    socket.on('close', close)
    return () => {
      socket.off('data', data)
      socket.off('error', error)
```

(续下页)

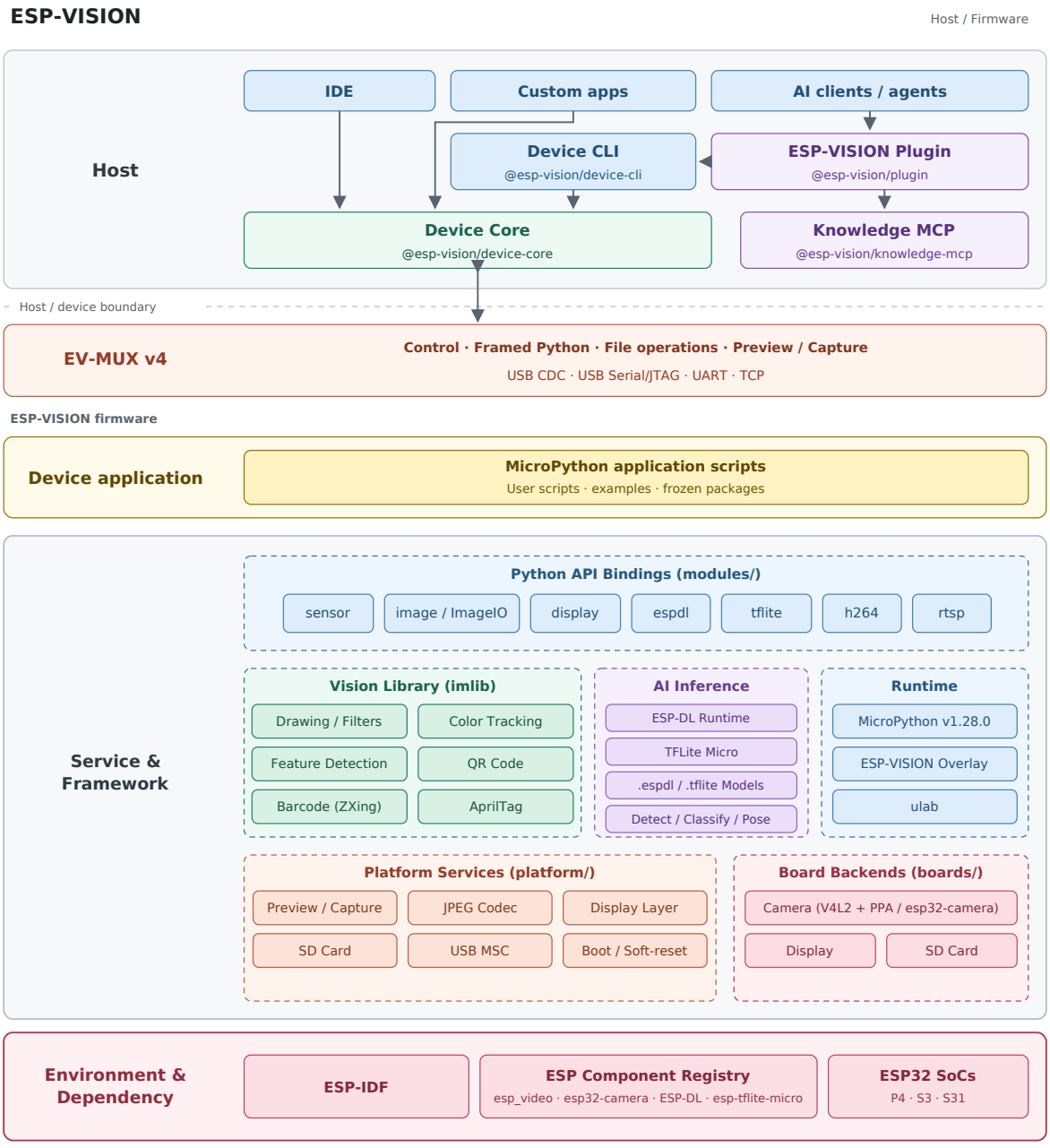


图 1: IDE / Device CLI / 自定义主机应用 → Device Core → EV-MUX → 固件；Knowledge MCP → HTTP 文档

(接上页)

```
      socket.off('close', close)
    }
  },
  async connect({ signal }) {
    socket.connect({ host, port: 3232 })
    await once(socket, 'connect', { signal })
  },
  writeBytes(bytes) {
    return new Promise((resolve, reject) => {
      socket.write(bytes, error => error ? reject(error) : resolve())
    })
  },
  async disconnect() { socket.destroy() },
}
const device = createDevice({ transport })
const unsubscribe = device.on('python.output', text => process.stdout.write(text))
try {
  await device.connect({ role: 'tcp' })
  console.log(await device.inspect({ fields: ['identity', 'capabilities'] }))
  console.log(await device.python.execute({ source: 'print(1 + 2)', completion:
↪'wait' }))
} finally {
  try { await device.disconnect() }
  finally { unsubscribe() }
}
```

`connect()` 协商会话并探测 REPL，可能中断原有 Python 程序；它不是无副作用的在线探测。`inspect()` 默认查询身份、能力和运行状态，也可以通过 `fields` 明确查询 `network` / `storage`。`disconnect()` 负责释放会话、终止待处理操作、退订并调用传输关闭方法；即使连接失败，也应在 `finally` 清理。

10.3.4 浏览器 Web Serial 示例

从用户点击事件调用 `connectSerial()`；`requestPort` 需要安全上下文及用户选择。以下示例将 USB CDC 作为 `role: 'usb'`，USJ 对应 `'usj'`。先释放 reader/writer 锁，再关闭串口。

```
import { createDevice } from '@esp-vision/device-core'

async function connectSerial() {
  const port = await navigator.serial.requestPort()
  await port.open({ baudRate: 115200 })
  const reader = port.readable.getReader()
  const writer = port.writable.getWriter()
  let listeners = null
  let reading = null
  const transport = {
    subscribe(next) {
      listeners = next
      reading = (async () => {
        try {
          for (;;) {
            const { value, done } = await reader.read()
            if (done) break
            listeners?.data(value)
          }
          listeners?.close()
        } catch (error) { listeners?.error(error) }
        finally { reader.releaseLock() }
      })()
      return () => { listeners = null }
```

(续下页)

(接上页)

```
    },
    writeBytes: bytes => writer.write(bytes),
    async disconnect() {
      await reader.cancel().catch(() => {})
      await reading
      writer.releaseLock()
      await port.close()
    },
  }
  const device = createDevice({ transport })
  const unsubscribe = device.on('python.output', text => console.log(text))
  try {
    await device.connect({ role: 'usb' })
    await device.python.execute({ source: 'print(123)', completion: 'wait' })
  } finally {
    try { await device.disconnect() }
    finally { unsubscribe() }
  }
}
```

浏览器网络模式通过二进制 WebSocket 桥接，仍使用相同的 transport 契约与 `role: 'tcp'`。适配器须检查 WebSocket OPEN、限制 `bufferedAmount` 并在断线时报告失败；调用 `send` 不能证明设备已收到数据。UI 应订阅 `state` 并处理意外断线，桥接细节见*communication/host-integration*。

10.3.5 文件与交互式 Python

以下操作假设 `device` 已连接，输出通过 `device.on('python.output', handler)` 接收。上传与执行分别进行，没有隐式上传后运行选项。

```
const source = 'print("device-core example")\n'
await device.files.write({ path: '/core-example.py', data: source })
const readback = await device.files.read({ path: '/core-example.py', encoding:
↪'utf-8' })
if (readback !== source) throw new Error('Readback mismatch')
const run = await device.python.execute({ path: '/core-example.py', completion:
↪'start' })
const result = await run.done
console.log(result.ok, result.path)
await device.files.remove({ path: '/core-example.py' })

await device.python.input('print(123)\r')
await device.control({ action: 'stop' })
```

`python.execute` 的 `source` 与 `path` 二选一；`completion: 'wait'` 等待结果，`'start'` 返回包含 `done` Promise 的启动句柄。启动返回或上传成功不能替代执行完成证据。无限运行需 `control({ action: 'stop' })`，保持接收并等待执行结果；停止 ACK 不能替代资源清理完成。

`files.list/read/write/remove/mkdir` 是唯一应用文件入口。`files.write` 接收字符串或 `Uint8Array`，SDK 0.3.0 均通过 `file.write` 原始二进制块、连续原生写入和 SHA-256 提交完成；固件须声明 `file.write.binary`，缺少能力时须升级。读取和目录操作当前使用 framed REPL。文件系统必须已挂载、允许写入且有临时文件空间。终端输入、自动脚本和文件操作共用一个解释器；Core 对冲突的自动操作返回 `DEVICE_BUSY`，应用不能同时向自动操作使用的 REPL 注入终端输入。详细边界见*communication/python-files*。

10.3.6 Wi-Fi 配置与状态

```
await device.configure({ network: {
  credentials: { ssid: '<wifi-ssid>', password: '<wifi-password>' },
  enabled: true,
}, timeout: 45000 })
const { network } = await device.inspect({ fields: ['network'] })
console.log(network.state, network.ipv4)
await device.configure({ network: { enabled: false } })
// Clear saved credentials and disable networking:
// await device.configure({ network: { credentials: null } })
```

`configure` 是局部更新：省略字段不变，`credentials: null` 清除凭据并关闭网络，不能同时指定 `enabled: true`。提供凭据时必须同时包含 `ssid` 与 `password`，开放网络使用空密码。Core 关联事务并等待结果，不只返回固件 accepted；配置由顺序操作组成而非原子事务。失败时 `error.completed` / `error.pending` 描述已确认步骤与未完成事务，不自动回滚。`timeout` 与 `signal` 在同一个请求对象中传入。

优先通过 USB 配网，避免网络修改使 TCP 连接断开。配置与诊断不得记录密码；查询不返回已保存密码。配网后先释放 USB，再建立 TCP 会话，不能转移旧 token。

10.3.7 发现与身份核验

固件通过 `_esp-vision._tcp.local` 发布主芯片设备 ID、完整板名、固件版本、规范主机名和协议版本。主机名格式为 `esp-vision-<full-board-name>-<last-6-mac-hex>-<firmware-version>`，串口、TCP、Python 与发现查询一致。发现不会连接设备或抢占会话；在线不表示空闲。

Node 安装可选 `multicast-dns` peer 后使用：

```
import { createDevice } from '@esp-vision/device-core'
import { createNodeDiscovery } from '@esp-vision/device-core/discovery/node'
import { tcpTransport } from '@esp-vision/device-core/node'

const device = createDevice({ discovery: createNodeDiscovery() })
const devices = await device.discover({ timeout: 3000 })
const selected = devices.find(item => item.compatible && item.addresses.length)
if (!selected) throw new Error('No compatible device found')
const tcpAdapter = tcpTransport({ host: selected.addresses[0], port: selected.port␣
↪})
try {
    await device.connect({ transport: tcpAdapter, role: 'tcp', expectedDevice:␣
↪selected })
    console.log(await device.inspect({ fields: ['identity'] }))
} finally { await device.disconnect() }
```

应用调用 `device.discover()`。`createNodeDiscovery()` 和 `createBridgeDiscovery()` 返回 provider，契约为 `list({ timeout, signal })`，不提供 `discover()`；通过 `createDevice({ discovery: provider })` 注入。SDK 0.8 从 `@esp-vision/device-core/node` 导出 `inventory`、`listSerial`、`filterSerialPorts`、`serialEntry`、`transportFor`。默认 inventory 只枚举，不连接；传输构造也不会打开连接。串口枚举使用 npm 原生预编译模块，无需主机 Python。见高级导出 *API* 完整参考。

```
import { inventory } from '@esp-vision/device-core/node'

const { devices, warnings } = await inventory({ timeout: 3000 })
console.log(devices.map(({ interface, address, status }) => ({ interface, address,␣
↪status })))
console.warn(warnings)
```

Node inventory 汇总系统串口枚举和局域网发现，并返回 `devices`、`warnings` 与 `sourcesAvailable`。默认不打开候选端口；`probe: true` 会逐一连接，可能中断正在运行的 Python。选项默认值、候选筛选和打包器资源处理见高级导出 *API* 完整参考。

浏览器使用 `@esp-vision/device-core/discovery` 中的 `createBridgeDiscovery('ws://127.0.0.1:8766/device')` 注入本地桥。浏览器没有原生 DNS-SD 枚举能力。结果包含 `deviceId`、`hostname`、`board`、`firmwareVersion`、`addresses`、`port`、`compatible`，桥接结果还可能有 `connectionUrl`。宿主据此构造连接适配器，使用 `expectedDevice` 在获取会话前核验 ID 与主机名，防止 DHCP 地址陈旧；这不是认证。VLAN、组播过滤和 AP 隔离可能需要手动地址，过期桥接句柄需要重新发现。

10.3.8 预览、日志与抓帧

```
const unsubscribePreview = device.on('preview', image => {
  const url = URL.createObjectURL(new Blob([image.data], { type: image.mimeType }))
  const previous = imageView.src
  imageView.src = url
  if (previous.startsWith('blob:')) URL.revokeObjectURL(previous)
})
const unsubscribeLog = device.on('log', text => systemLog.write(text))
const image = await device.capture({ source: 'camera', format: 'jpeg', timeout:␣
↪15000 })
// image: { data: Uint8Array, mimeType: 'image/jpeg', metadata }
// Call unsubscribePreview()/unsubscribeLog() and revoke the final URL on UI␣
↪teardown.
```

`imageView` 与 `systemLog` 是宿主 UI 对象。`on` 返回取消订阅函数，图像事件和抓帧结果都包含原始 JPEG 字节，不需要 UI 再解析 EV-MUX。Core 分发前核验会话及 epoch；preview 由设备程序 `img.flush()` 触发，拥塞时允许丢帧。相机观测要求 VM 可执行，并同时具有 `debug.capture_frame` 和 `debug.capture_frame.observed.v1`。它只编码应用最近一次成功 snapshot 发布的独立帧，不初始化或物理采集摄像头，不覆盖应用持有的图像。缺少安全能力时会在发送 RPC 前以 `UNSUPPORTED_OPERATION` 拒绝并提示升级。`CAMERA_FRAME_NOT_READY` 表示没有有效发布帧；`NO_MEM` 表示观察副本分配失败，但应用 snapshot 仍成功。观察既有程序时应通过 `probeRepl: false` 连接，避免默认的 REPL 中断。

`capture({ source: 'screen', format: 'jpeg' })` 调用专用 `debug.capture_screen`，返回 LCD 最后提交的 RGB565 帧编码的 JPEG；它在 VM 任务中运行，不调用摄像头或改变屏幕。无 LCD 返回 `NOT_SUPPORTED`，未初始化/无有效帧返回 `DISPLAY_NOT_READY`；不会回退到摄像头。旧固件缺少该能力时由 Core 报告不支持。高速原始二进制上传仍未实现。真机测试通过 `npm run test:hardware` 的 `--screen-output` 显示测试图案并保存抓屏，见*communication/validation*。

10.3.9 复位、烧写与发布

```
await device.control({ action: 'reset', mode: 'soft', timeout: 15000 })
// await device.control({ action: 'reset', mode: 'hard' })
// await device.control({ action: 'bootloader' })
```

软复位由 Core 协调新的 ready epoch 和 REPL 屏障；超时或取消后先检查状态，不自动重放原业务。硬复位和进入下载模式会使原有链路失效，成功应答不等于设备已重新上线。

ROM 烧写仍使用独立子路径 `@esp-vision/device-core/firmware`；宿主注入 ROM loader、已批准镜像、芯片/分区约束和复位策略。Core 不附带镜像，不在 EV-MUX 上执行 ROM 下载协议。`control({ action: 'bootloader' })` 只请求进入下载模式，不进行烧写。物理交接见*communication/interfaces*。

包的源码、协议测试与发布流程在 `host/device-core/` 同仓维护；IDE 和自定义应用固定 npm 版本，Agent 设备工作流使用 Device CLI。仅修改 SDK 时运行 JS 测试、schema/package 检查与发布 dry run，不应触发固件编译矩阵。发布与固件版本独立，正式 registry 可用性以发布 job 和 `npm view` 为准。协议能力仍需由版本/能力协商与真机测试确认，验收见*communication/validation*。

10.3.10 应用层固件更新入口

应用调用 `device.firmware.check()` 自动获取官方发布目录及各版本元数据；`catalog` 仅用于覆盖为自定义源。通过 `device.firmware.update({ release, openLoader, ... })` 安装选定

的兼容版本。包内 CLI 已提供 Node ROM 适配器，命令行用户无需配置 loader 脚本；JavaScript 宿主通过 `openLoader` 适配其串口/Web Serial 环境。完整目录、取消与复位契约见*应用 API 完整参考*，底层 loader 适配仍可直接使用 FirmwareFlashSession。

Node 宿主可从 `@esp-vision/device-core/node` 导入 `openNodeRom`，传入 `openLoader: ({ signal }) => openNodeRom({ port, signal })`，无需自己实现烧写器。浏览器宿主提供 Web Serial ROM 适配器。空白设备在 check/update 中显式设置 `mode: 'install'`，默认模式仅更新应用。

10.3.11 浏览器 ROM 烧录

SDK 0.6.0 通过 `/browser` 入口提供共享的运行态串口与 ROM 适配器。须使用 webpack、Vite 或 esbuild 打包，在支持 Web Serial 的浏览器中通过 HTTPS 或 localhost 运行。无需 Python、主机端 tasmota 适配器或包管理器补丁。

```
import { createDevice } from '@esp-vision/device-core'
import { openBrowserRom } from '@esp-vision/device-core/browser'

// 在点击处理函数内申请端口，再等待其他异步任务。
const port = await navigator.serial.requestPort()
// 打开 ROM 适配器前先释放已有运行时串口连接。
const result = await createDevice().firmware.update({
  release: selectedRelease, board: selectedRelease.board,
  mode: 'update', // 显式 install 需要合格的完整镜像元数据。
  openLoader: ({ signal }) => openBrowserRom({ port, signal, logger: console }),
  onProgress: percent => updateProgress(percent),
})
if (result.needsReset) showManualReset()
```

`selectedRelease` 来自针对物理板型的 `firmware.check`。Core 在获取 ROM 端口前下载与校验镜像，在启动 stub 前校验芯片、修订和安全模式，检查既有分区（update）或实测容量（install），完整回读每个写入字节。不可用猜测偏移替代缺失元数据，也不自动切换安装模式。Promise 完成前保持页面与端口可用；写入前可取消，开始写入后不允许中断写入与回读。单独的芯片识别界面可保留一个适配器、先 `initialize()`，再把同一对象作为 `openLoader` 返回；应在 `finally` 关闭，即使镜像校验失败早于 Core 获取端口。原生 USB 烧录后可能需要物理复位。此 ROM 适配器与 EV-MUX 运行时串口、TCP OTA 分开。

SDK 0.6.0 锁定官方 `esptool-js@0.7.1`，在压缩写入前协商 UART 下载速率，处理 USB-OTG ROM 直接同步，并校验有界读回包、尾随 MD5 与完整写入范围的 SHA-256。运行态 UART 协商仍是独立的 EV-MUX 生命周期。不支持的 stub 或损坏的 SLIP 帧仍明确失败。包的根与 Node 入口不加载浏览器后端。

10.4 设备管理命令行

独立 npm 包 `@esp-vision/device-cli` 安装 `esp-vision-device` 命令。CLI 使用公开的 `@esp-vision/device-core` API，支持设备发现与管理、Python 执行、设备文件、图像采集、批处理计划和限时事件监视。需要 Node.js 22 或更高版本。

插件安装以及 AI 查找、烧录、开发和验证的 prompt 示例，参见*用 AI 开发视觉应用*。

10.4.1 安装与宿主机要求

```
npm install -g @esp-vision/device-cli
esp-vision-device --help
esp-vision-device inspect
```

串口访问使用随包 npm 原生绑定，烧录使用与网站共用的 esptool-js 后端，无需安装主机 Python、pyserial 或 Python esptool。仍需 Node.js 22+、系统驱动和串口权限。局域网发现使用内置 multicast-DNS。已废弃的 `--python` 参数兼容接受但不再生效。后端缺失、权限不足和端口占用分别给出提示；普通串口错误不能直接判定为设备故障。

10.4.2 发现与检查

不指定目标时，`inspect` 枚举系统串口和局域网公告，不打开串口或接管设备会话。USB 描述符仅用于候选分类，不代表已确认固件身份。默认显示 USB 串口候选；`--all-ports` 会显示其它系统已枚举的串口。`--interface` 接受逗号分隔的 `all`、`serial`、`usb`、`usj`、`uart` 和 `tcp`。显式 `--port` 不受发现筛选限制，也支持 Linux `/dev/serial/by-id/...` 别名。

```
esp-vision-device inspect --json
esp-vision-device inspect --interface serial --all-ports
esp-vision-device inspect --interface usj,usb --probe
esp-vision-device inspect --select 1 --fields identity,capabilities,runtime
esp-vision-device inspect --port /dev/ttyACM0 --role usj
esp-vision-device inspect --tcp esp-vision-board.local:3232 --fields identity,
↪network,storage
```

`--probe` 查询发现到的候选设备，`--select` 连接本次枚举结果中选定的设备。两者都使用被动 EVMUX 获取，不会向正在运行的应用发送 Ctrl-C。`--probe-timeout` 每个候选设备默认 5000 毫秒，最大 60000 毫秒。发现收集默认 3000 毫秒，局域网收集上限为 10000 毫秒。指定目标时默认查询 identity、capabilities 和 runtime；`--fields` 还接受 `network` 和 `storage`。指定目标的 inspect、configure、固件检查、capture、watch 和原生 `files write` 使用被动连接。Python 执行、依赖 REPL 的 `files list/read/remove/mkdir`、control、固件更新和所有批处理计划使用常规 REPL 连接探测，可能发送 Ctrl-C。显式串口目标可用 `--role` 指定 `usj`、`usb`、`uart` 或 `unknown`，并且必须配合 `--port`；默认角色为 `unknown`。

Runtime 中的 `scriptRunning` 和 `run` 表示当前 SDK 会话跟踪的工作。新建的 CLI 进程无法通过这些字段判断先前启动的脱离连接程序是否仍在设备上运行。`vmReady` 表示虚拟机已就绪，不表示虚拟机空闲。

10.4.3 网络配置、控制与固件

首次配网、确认联网和切换无线连接的完整步骤见*主机接入、Wi-Fi 配置与恢复实践*。

```
esp-vision-device configure --port COM3 --ssid MyWiFi
esp-vision-device configure --port COM3 --ssid MyWiFi --password-stdin
esp-vision-device configure --port COM3 --network-enabled false
esp-vision-device configure --port COM3 --forget-wifi
esp-vision-device control --tcp 192.168.1.100 --action stop
esp-vision-device control --port COM3 --action reset --mode soft
esp-vision-device control --port COM3 --action bootloader
esp-vision-device firmware check --port COM3 --version 2026.10.15
esp-vision-device firmware check --board ESP32_P4X_EYE --json
esp-vision-device firmware update --port COM3 --version 2026.10.15
```

配置 Wi-Fi 时会隐藏密码提示输入；非交互环境可用 `--password-stdin` 从标准输入读取。没有命令行密码参数，配置结果也不会包含凭据。reset 必须指定 `--mode soft` 或 `--mode hard`；收到确认不代表设备已重新连接。

固件检查默认使用官方版本源；`--catalog` 可指定本地目录文件或 HTTPS URL。离线检查需要 `--board`，也可用 `--current-version` 指定当前版本。更新必须指定本地 `--port` 和明确的 `--version`；内置串口烧录器使用 `esptool-js@0.7.1`。TCP 不是 OTA。默认 `--mode update` 只写应用并保留现有分区布局和用户数据。`--mode install` 需要 `--board`。CLI 捆绑 Core 0.7.0 或更新版本时，会清除内部 `ffat` 文件并写完整镜像，镜像也可能替换 NVS 配置；不擦除 SD 卡。旧版 CLI 包仍使用其原有安装行为；CLI 独立发布，不会自动获得新版 Core，依赖清空能力前应确认捆绑版本。空

白设备应使用可进入 ROM 下载模式的 USJ/UART 串口。`--image` 提供本地固件字节，但不会绕过版本完整性校验。

10.4.4 Python 执行

Python 输入必须且只能选择一种：内联 `--source CODE`、本地 `--file LOCAL.py`，或设备上的 `--path DEVICE.py`。默认 `--completion wait` 会等待执行结束。正整数 `--timeout MS` 超时时会取消等待中的执行。`--max-output-chars` 限制捕获的输出（默认 16,384，最大 1 MiB）。

```
esp-vision-device python --source 'print(42)' --port COM3
esp-vision-device python --file ./main.py --port COM3 --timeout 30000
esp-vision-device python --path /main.py --completion start --port COM3
```

`--completion start` 仅接受 `--path`，并在设备 VM 进入 runner 后确认启动。该确认不表示应用已成功初始化或执行完成。CLI 随后释放连接并退出，不会继续监视或恢复脚本。EV-MUX 将 Python 标准输出固定到启动程序的会话。需要实时 Python 输出时使用前台 `--completion wait`；后续 watch 无法重新连接到脱离连接任务的标准输出。可通过图像采集或应用写入的进度文件检查脱离连接的程序。

前台 Python 执行和批处理中的 Python 步骤会将 `python.output`、`python.log` 和 `python.state` NDJSON 事件实时写入 stderr。输出受 `--max-output-chars` 限制，达到上限时发送一次截断标记。stdout 结果包含元数据，不会重复包含已发送的 Python 文本。

10.4.5 设备文件与图像采集

`/models` 等路径属于设备，Windows 上也应保持原样。无效路径会在连接前被拒绝。Git Bash/MSYS 可能将斜杠开头的参数改写为 Windows 宿主路径；请使用 PowerShell，或仅为该条命令添加 `MSYS2_ARG_CONV_EXCL="*"` 前缀，并为本地路径提供经确认的 Windows 格式。仅给 `/models` 加引号不能阻止转换。不要从转换后的路径中直接删除盘符，也不要盲目重试设备操作。

```
esp-vision-device files list --path / --recursive --port COM3
esp-vision-device files read /main.py --output ./main.py --port COM3
esp-vision-device files write /backup.bin --file ./backup.bin --port COM3
esp-vision-device files remove /old.bin --type file --port COM3
esp-vision-device files mkdir /data --recursive --port COM3
esp-vision-device capture camera --output ./frame.jpg --quality 80 --port COM3
esp-vision-device capture screen --output ./screen.jpg --port COM3
```

`capture camera` 观察应用最近的相机帧，不是 LCD 画面。应用绘制是否可见取决于固件发布帧的方式；声明 `debug.capture_frame.observed.v1` 的固件在采集时发布独立副本，因此不包含之后的绘制，也不会触发新的采集。`capture screen` 获取当前 LCD 帧缓冲，包含已经绘制到该缓冲中的标注和界面，适合检查 LCD 显示效果，并要求显示器已初始化。相机与显示器的图像尺寸可能不同；两种命令都不会启动应用或初始化传感器、显示器。输出的 `source` 只标明请求来源，不保证是否存在标注。

文件读取和 JPEG 采集会将二进制数据写入指定的本地 `--output`，并返回路径、字节数和元数据。读取和采集默认上限为 16 MiB；`--max-bytes` 可调低上限。`files write` 通过二进制文件 RPC 发送本地字节。JSON 中不会嵌入图像或文件内容。

10.4.6 批处理与 watch

`batch PLAN.json` 会在连接前完整校验 `schemaVersion: 1` 计划，然后在一个连接中按顺序执行，并在首个失败操作处停止。报告包含已完成操作和失败信息。支持的操作为 `python`、`files.list`、`files.read`、`files.write`、`files.remove`、`files.mkdir` 和 `capture`。Python 步骤会等待执行完成。本地文件路径以计划文件所在目录为基准；设备路径必须是绝对路径。批处理使用常规 REPL 连接探测，因为计划可能执行 Python 或依赖 REPL 的文件操作。

```
{
  "schemaVersion": 1,
  "operations": [
    { "op": "python", "source": "print('hello')" },
    { "op": "files.read", "path": "/boot.py", "output": "./boot.py" },
    { "op": "capture", "source": "screen", "output": "./screen.jpg" }
  ]
}
```

`watch --duration MS` 需要 `--port` 或 `--tcp`，时长范围为 1 毫秒至 1 小时。它被动观察当前获取会话可见的输出、日志、状态变化和诊断事件，然后释放连接。watch 不会启动脚本、重新连接或重新接收先前脱离连接任务的 Python 标准输出。SIGINT/SIGTERM 会取消命令并释放所有权。

10.4.7 输出、超时与取消

`--json` 会在 stdout 输出一个 `schemaVersion: 1` JSON 信封。状态、Python 输出、日志、固件阶段、取消、错误和耗时以 NDJSON 事件写入 stderr。二进制内容始终写入本地文件。`--timeout` 接受 1..600000 毫秒，适用于 inspect/configure/control 和设备发现收集；`--probe-timeout` 设置每个已发现候选设备的探测超时。显式指定 `--timeout` 时，也适用于前台 Python 执行、文件读取、图像采集及批处理中的这些操作。它不是连接超时：设备连接固定使用 30000 毫秒。Python `--completion start` 不接受 `--timeout`；watch 使用 `--duration`；固件操作不使用此选项，但支持信号取消。固件更新开始写入后，Core 会完成回读验证和复位处理。files list/write/remove/mkdir 支持信号取消，但没有单操作超时。SIGINT 返回码为 130，SIGTERM 为 143，参数/计划错误为 2，其它失败为 1，成功为 0。占用或结果不确定的操作不会自动重试。运行 `esp-vision-device --help --json` 可查看机器可读的命令和能力参考。

10.5 安装与兼容性

```
npm install '@esp-vision/device-core@~0.6.0'
# Optional Node LAN discovery adapter:
npm install multicast-dns
```

本页示例面向 SDK `0.6.0`；依赖锁定到 minor、用 lockfile 记录解析后的 patch，安装前确认 registry 中有所选版本。Node 主机基线为 Node.js 22；浏览器需要打包器/import map，并由宿主提供 Web Serial 或二进制 WebSocket 字节连接。仅支持 EV-MUX v4，v3 设备必须升级固件。包版本、固件版本与 IDE 版本相互独立。源码 0.6.0 是继 0.5.1 之后的 minor 发布；源码变更与 dry run 不代表 npm 已发布，也不会更新既有 CLI/Plugin 内嵌的 Core。相机观测要求固件能力 `debug.capture_frame.observed.v1`，旧的不安全实现会在本地被拒绝。

10.6 架构与使用入口

本图与项目 README、系统架构章节复用同一 SVG。主机应用负责设备选择及界面/流程策略，Device Core 负责共享 API、协议和会话状态，EV-MUX 定义与固件的运行时通信；字节适配器提供串口/socket I/O，不维护另一套协议栈。图中代表性 API 不替代完整参考；发现与 ROM 烧写各有独立生命周期，ROM loader 不通过 EV-MUX 写镜像。

根入口只导出 `createDevice`，返回设备对象，不提供公开的 `Device` 构造函数。所有应用方法、选项、返回值和事件见*应用 API 完整参考*；其余包导出及公开类方法完整列于*高级导出 API 完整参考*，供集成维护者查阅。Agent 工作流通过 Device CLI 操作设备；Knowledge MCP 是独立的文档接口。

```
import { createDevice } from '@esp-vision/device-core'

// transport is supplied by the host; its complete contract is in api.
const device = createDevice({ transport })
```

(续下页)

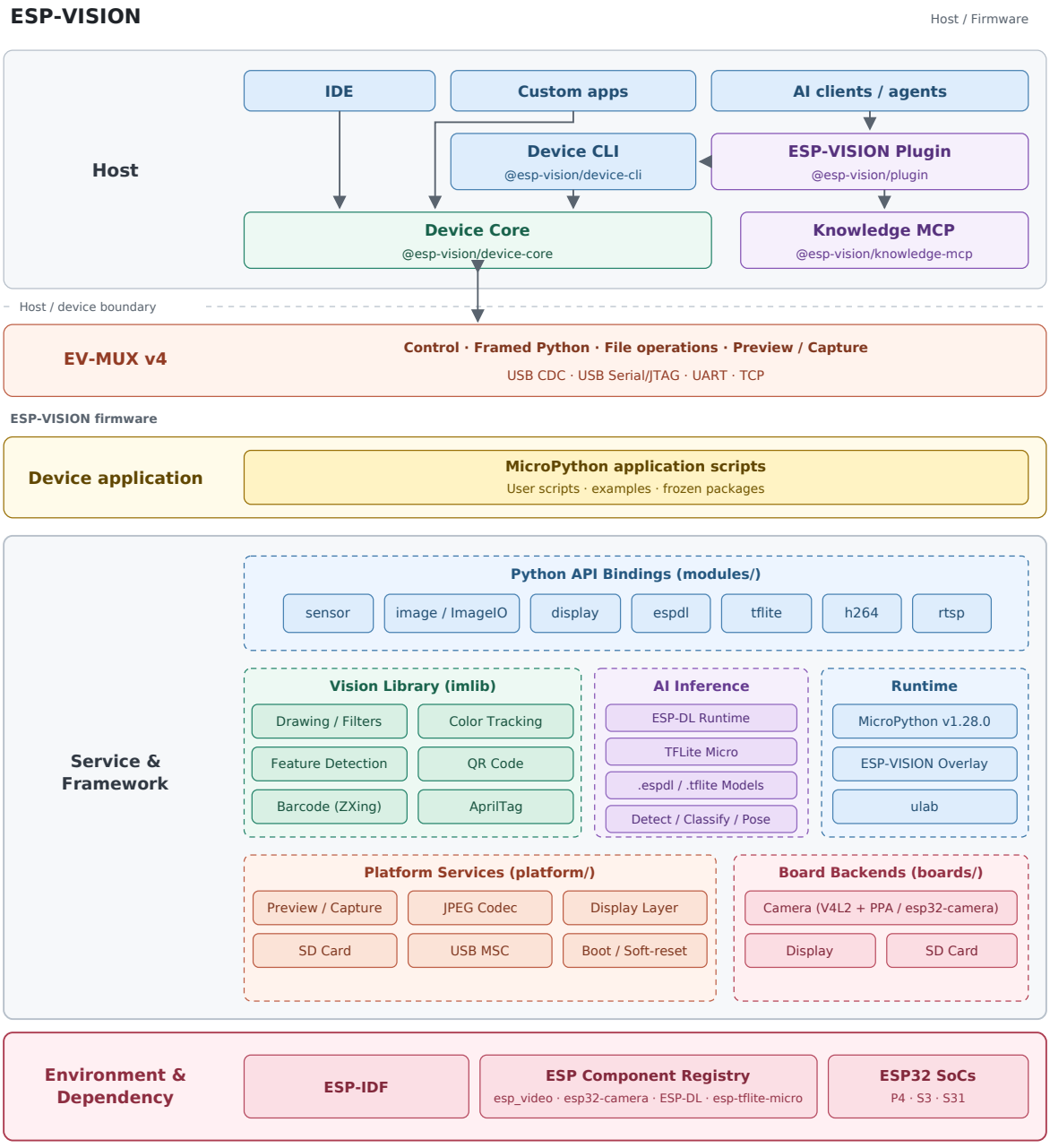


图 2: IDE / Device CLI / 自定义主机应用 → Device Core → EV-MUX → 固件；Knowledge MCP → HTTP 文档

(接上页)

```
const off = device.on('python.output', text => console.log(text))
try {
  await device.connect({ role: 'usb' })
  console.log(await device.python.execute({ source: 'print(1 + 2)' }))
} finally {
  try { await device.disconnect() } finally { off() }
}
```

可直接运行的 Node TCP 与浏览器 Web Serial 适配示例见*Device Core npm 包对接*。帧格式、路由、所有权见*index*，统一真机测试入口见*validation*。

10.7 许可证

npm 包及 ESP-VISION 自有贡献采用 Apache-2.0，与固件一致。包内包含 LICENSE 与 NOTICE；NOTICE 保留历史 MIT 版权与许可原文，第三方依赖仍遵守各自许可证。项目清单见许可证。

SDK 0.5.1 包含 Node 传输、串口/网络设备枚举、ROM 烧写、不可变发布快照和单 BIN 应用升级。`@esp-vision/device-core/node` 导出这些宿主适配器及打包器所需的资产 URL。安装后可执行 `npm --prefix node_modules/@esp-vision/device-core test` 和 `npm --prefix node_modules/@esp-vision/device-core run test:hardware -- --help`；缺少测试或空测试集将失败。应用 `discover()` 与 provider `list()` 的关系见*Device Core npm 包对接*。

Chapter 11

项目关系

本章说明 ESP-VISION 与 MicroPython、OpenMV 及主要第三方组件之间的关系。许可证与分发义务请参阅许可证。

11.1 ESP-VISION、MicroPython 与 OpenMV

三者的关系可以概括为：

- **MicroPython 是语言运行时和固件基础**。 ESP-VISION 以固定版本的 MicroPython ESP32 port 为基线，通过 `overlay/micropython/`、MicroPython 用户 C 模块以及 ESP-IDF 组件加入视觉和板级能力。它不是一个独立的 Python 解释器。
- **OpenMV 是部分视觉实现和 API 设计的上游来源**。 ESP-VISION 复用了 OpenMV `imlib` 的一部分图像算法，以及部分 `image` 绑定与辅助代码，并保留这些文件原有的许可证和版权声明。
- **ESP-VISION 是面向乐鑫芯片的独立集成项目**。相机、显示、存储、USB、编解码、ESP-DL 推理、TFLite Micro 推理和板级适配由 ESP-VISION 结合 ESP-IDF 组件实现。ESP-VISION 不是 OpenMV 固件的分支，也不包含完整的 OpenMV 硬件抽象层、IDE 或全部功能。

因此，`sensor`、`image` 等模块会尽量保持 OpenMV 风格，方便迁移已有脚本，但 **API 名称相同不表示行为和功能完全相同**。支持范围还会受到芯片、开发板、内存、硬件外设和编译选项影响。应以本指南的*API 参考* 为准。

11.2 依赖层次

ESP-VISION 固件中的主要代码层次如下：

```
用户脚本
   |
   +-- MicroPython 运行时与 ESP32 port
   |
   +-- ESP-VISION Python 模块
   |       +-- OpenMV 风格的 sensor / image API
   |       +-- display / imageio / espdl / tflite / h264 / rtsp
   |
   +-- 算法与中间件
   |       +-- OpenMV imlib 子集
   |       +-- ulab / ZXing-C++ / ESP-DL / TFLite Micro
   |
   +-- ESP-IDF、托管组件与板级后端
```

依赖来自三种位置：

- `lib/` 中的 Git submodule，例如 MicroPython、`ulab` 和 ZXing-C++；
- `components/` 与部分 `modules/` 中随仓库维护的第三方派生代码；
- `idf_component.yml` 解析并下载的 ESP Component Registry 组件。具体组件及版本可能随 ESP-IDF 版本、目标芯片和开发板而变化。

各依赖的版本、本地路径和许可证见许可证 清单。

Chapter 12

许可证

本章是 ESP-VISION 许可证相关说明的唯一参考。这里的内容用于帮助开发者理解代码来源，不构成法律意见；具体义务始终以对应版本的许可证原文和源文件头为准。

ESP-VISION 自有代码（`platform/`、大部分 `modules/`、`boards/` 以及构建文件）以 Apache License 2.0 发布，记录在仓库顶层的 `LICENSE` 中。引入的第三方代码保持其源文件、SPDX 标识或上游许可证声明的原始许可证；仓库顶层 `LICENSE` 不会覆盖或替换这些许可证。

12.1 许可证清单

下表列出与分发最相关的直接依赖。它不是所有传递依赖的完整清单。

项目或代码	本地路径	在 ESP-VISION 中的作用	许可证
esptool-js 0.7.1	host/device-core/package.json 依赖，由 src/rom_loader.js 延迟加载	浏览器 Web Serial 与 Node 共用的官方 ROM 后端；上游 src/targets/stub_flasher/README.md 标明 stub 来源为 esp-flasher-stub v1.2.2（set 2）	Apache-2.0（安装包 LICENSE）；stub 二进制双许可 Apache-2.0/MIT。作为依赖安装，实例级适配不修改上游源文件。间接依赖 pako 保留其 MIT/Zlib 声明，atob-lite 保留 MIT。
@serialport/bindings-cpp 13.0.0; Koffi 2.16.3	host/device-core/package.json optional dependencies; CLI/plugin native bundles	原生串口与 Linux USJ 控制线；原版 npm 预编译二进制和加载器	MIT; original package licenses and Koffi Node API/cnoke notices retained in portable bundles
Device Core 主机 SDK	host/device-core	EV-MUX v4 主机通信、Python、资源文件、发现及图像接入	Apache-2.0（host/device-core/LICENSE）；历史 MIT 声明保留于 NOTICE
设备端相机/LCD 测试片段	host/device-core/tests/fixtures/device-checks.py	从固件测试迁入的真机验收函数，不包含在 npm 包中	Apache-2.0，保留原版权声明
sync-pr-to-gitlab (6ea5db6)	.github/actions/sync-pr-to-gitlab	GitHub PR 到 GitLab MR 的同步，移除 Jira 专用元数据；保留上游 Python 版权和许可证声明	Apache-2.0
MicroPython v1.28.0	lib/micropython	Python 运行时与 ESP32 port 基线	MIT
MicroPython 执行与线程适配	overlay/micropython/shared/runtime/pyexec.c、overlay/micropython/ports/esp32/mpthreadport.c	基于固定上游文件捕获和继承会话来源，保留原版权和许可证头	MIT
MicroPython 复合 USB 描述符适配	overlay/micropython/shared/tinyusb/mp_usbd_descriptor.c（v1.28.0 基线）	CDC/MSC/UVC 描述符及速率适配，保留上游许可证头	MIT
OV2710 定时器生命周期适配	tools/prepare_ov2710.py、platform/ov2710_lifecycle.inc 与原样测试源 tools/tests/fixtures/ov2710/upstream.c；esp_cam_sensor 2.4.0，提交 8fc931636ffa937c89c	构建时校验并复制 sensors/ov2710/ov2710.c``（SHA-256 ``4a9af395151527b156b1d 确认回调停止后才释放 SCCB，保留原始文件头，不修改托管组件	Apache-2.0
TinyUSB UVC 传输适配	overlay/micropython/shared/tinyusb/esp_vision_video_trinc、tools/tests/tinyusb_video_stubs/upstream_transfer.c；espressif/tinyusb	构建时生成经检查的 src/class/video/video_device.c 副本；终止失败或短包 ISO 帧，单独报告失败完成，并保留原版权和许可证头	MIT

Submit Document Feedback

12.2 如何理解许可证

仓库顶层的 `LICENSE` 适用于 ESP-VISION 自有代码，但不会覆盖第三方许可证：

- 第三方文件继续由其原始许可证授权，并保留原始版权与许可证头；
- 同一个目录可能包含不同许可证，例如 `imlib` 主要为 MIT，而 `apriltag.c` 为 BSD-2-Clause；
- Apache-2.0、MIT 和 BSD 等宽松许可证通常允许组合分发，但仍需履行保留版权、许可证文本和声明等条件；
- 某些乐鑫组件使用带产品范围条件的许可证。例如当前 `esp_new_jpeg` 许可证文本限定用于乐鑫产品，不能只根据名称将其视为无附加条件的通用 MIT；
- 商标权、专利权、模型文件和数据集可能有独立条款，不能从源码许可证自动推导。

发布源码或固件前，至少应检查：

1. 仓库顶层 `LICENSE`；
2. 每个 Git submodule 的 `LICENSE`；
3. 所选构建生成的 `managed_components/*/LICENSE`；
4. 被修改或新增源文件的 SPDX 标识、许可证头和版权声明；
5. 模型、字体、测试图片等非代码资产的来源和使用条款。

12.3 OpenMV 与 GPL 代码路径

OpenMV 上游并非所有可选算法都采用相同许可证。ESP-VISION 在 `micropython.cmake` 和 `components/imlib/CMakeLists.txt` 中设置 `OMV_NO_GPL=1`，不编译 OpenMV 中受 GPL 条件控制的代码路径。这个开关只说明当前默认构建排除了这些代码，不应被理解为对任意新增文件许可证的自动判定。

修改 `imlib` 功能开关、同步 OpenMV 上游代码或加入新算法时，必须逐文件检查许可证；不要仅根据文件所在目录推断许可证。

12.4 增加第三方代码

向 ESP-VISION 增加第三方包或源文件时：

1. 固定可复现的版本或提交；
2. 逐文件确认许可证及其与项目分发方式的兼容性；
3. 保留上游版权、SPDX 标识、许可证文本和必要的 NOTICE；
4. 在本清单中记录来源、版本、本地路径、用途和许可证；
5. 确认构建开关确实排除了不准备分发的代码，而不是仅在文档中声明；
6. 许可证不清楚或存在商业发布约束时，在合入前完成法律审查。

二进制上传原生测试通过 PSA 测试替身使用系统 OpenSSL 3 libcrypto（Apache-2.0）作为 SHA-256 对照；仅用于主机测试，不链接到固件，也不随 npm 分发。固件使用 ESP-IDF PSA Crypto。

Python 模块索引

d

display, 66

e

espdl, 72

h

h264, 80

i

image, 50
imageio, 78

l

lcd_touch, 68

r

rtsp, 82

s

sensor, 46
storage, 83

t

tflite, 76

u

uvc, 82

索引

A

a_bins() (image.histogram 方法), 57
a_lq() (image.statistics 方法), 59
a_max() (image.statistics 方法), 59
a_mean() (image.statistics 方法), 59
a_median() (image.statistics 方法), 59
a_min() (image.statistics 方法), 59
a_mode() (image.statistics 方法), 59
a_stdev() (image.statistics 方法), 59
a_uq() (image.statistics 方法), 59
a_value() (image.percentile 方法), 58
a_value() (image.threshold 方法), 58
APPLY_COLOR_PALETTE_FIRST() (在 image 模块中), 51
apriltag (image 中的类), 57
area() (image.blob 方法), 55
AREA() (在 image 模块中), 50

B

b_bins() (image.histogram 方法), 58
b_lq() (image.statistics 方法), 59
b_max() (image.statistics 方法), 59
b_mean() (image.statistics 方法), 59
b_median() (image.statistics 方法), 59
b_min() (image.statistics 方法), 59
b_mode() (image.statistics 方法), 59
b_stdev() (image.statistics 方法), 59
b_uq() (image.statistics 方法), 59
b_value() (image.percentile 方法), 58
b_value() (image.threshold 方法), 58
backlight() (display.ESP32Display 方法), 67
barcode (image 中的类), 57
BAYER() (在 image 模块中), 50
BICUBIC() (在 image 模块中), 50
bilateral() (image.Image 方法), 63
BILINEAR() (在 image 模块中), 50
binary() (image.Image 方法), 62
BINARY() (在 image 模块中), 50
binary_to_grayscale() (在 image 模块中), 52
binary_to_lab() (在 image 模块中), 52
binary_to_rgb() (在 image 模块中), 52
binary_to_yuv() (在 image 模块中), 52
bind() (在 uvc 模块中), 83
bins() (image.histogram 方法), 57
BLACK_BACKGROUND() (在 image 模块中), 51
blob (image 中的类), 54
blur() (image.Image 方法), 63
buffer_size() (imageio.ImageIO 方法), 78
bytearray() (image.Image 方法), 60

C

CENTER() (在 image 模块中), 51
circle() (image.circle 方法), 54
circle (image 中的类), 54
classify() (espdl.ImageNetCls 方法), 75
clear() (display.ESP32Display 方法), 67
clear() (image.Image 方法), 61
close() (h264.H264Encoder 方法), 81
close() (image.Image 方法), 62
close() (imageio.ImageIO 方法), 79
CODABAR() (在 image 模块中), 52
code() (image.blob 方法), 55
CODE128() (在 image 模块中), 52
CODE39() (在 image 模块中), 52
CODE93() (在 image 模块中), 52
compactness() (image.blob 方法), 55
compress() (image.Image 方法), 61
convexity() (image.blob 方法), 56
copy() (image.Image 方法), 60
CORNER_AGAST() (在 image 模块中), 51
CORNER_FAST() (在 image 模块中), 51
corners() (image.barcode 方法), 57
corners() (image.blob 方法), 54
corners() (image.qrcode 方法), 56
corners() (image.rect 方法), 54
count() (image.blob 方法), 55
count() (imageio.ImageIO 方法), 78
crop() (image.Image 方法), 60
cx() (image.blob 方法), 55
cxf() (image.blob 方法), 55
cy() (image.blob 方法), 55
cyf() (image.blob 方法), 55

D

data_type() (image.qrcode 方法), 56
DATABAR() (在 image 模块中), 52
DATABAR_EXP() (在 image 模块中), 52
deinit() (display.ESP32Display 方法), 66
deinit() (espdl.ESPDet 方法), 74
deinit() (espdl.ImageNetCls 方法), 75

deinit() (espdl.Model 方法), 73
deinit() (espdl.NanoTrack 方法), 73
deinit() (espdl.PedestrianDetect 方法), 74
deinit() (espdl.YOLO11 方法), 74
deinit() (espdl.YOLO11nPose 方法), 75
deinit() (lcd_touch.Touch 方法), 68
deinit() (tflite.Model 方法), 77
density() (image.blob 方法), 55
detect() (espdl.ESPDet 方法), 74
detect() (espdl.PedestrianDetect 方法), 74
detect() (espdl.YOLO11 方法), 74
detect() (espdl.YOLO11nPose 方法), 75
difference() (image.Image 方法), 62
dilate() (image.Image 方法), 62
display
    module, 66
draw_arrow() (image.Image 方法), 62
draw_circle() (image.Image 方法), 61
draw_cross() (image.Image 方法), 61
draw_edges() (image.Image 方法), 62
draw_ellipse() (image.Image 方法), 61
draw_image() (image.Image 方法), 62
draw_keypoints() (image.Image 方法), 62
draw_line() (image.Image 方法), 61
draw_rectangle() (image.Image 方法), 61
draw_string() (image.Image 方法), 61

E

EAN13() (在 image 模块中), 52
EAN2() (在 image 模块中), 51
EAN5() (在 image 模块中), 52
EAN8() (在 image 模块中), 52
ecc_level() (image.qrcode 方法), 56
eci() (image.qrcode 方法), 56
EDGE_CANNY() (在 image 模块中), 51
EDGE_SIMPLE() (在 image 模块中), 51
elongation() (image.blob 方法), 55
enclosed_ellipse() (image.blob 方法), 56
enclosing_circle() (image.blob 方法), 56
encode() (h264.H264Encoder 方法), 80
erode() (image.Image 方法), 62
ESP32Display (display 中的类), 66
ESPDet (espdl 中的类), 73
espdl
    module, 72
extent() (image.blob 方法), 55
EXTRACT_RGB_CHANNEL_FIRST() (在 image 模块中), 51

F

FILE_STREAM() (在 imageio 模块中), 78
find_apriltags() (image.Image 方法), 65
find_barcodes() (image.Image 方法), 65
find_blobs() (image.Image 方法), 64
find_circles() (image.Image 方法), 64
find_line_segments() (image.Image 方法), 64
find_lines() (image.Image 方法), 64
find_qrcodes() (image.Image 方法), 65
find_rects() (image.Image 方法), 65
flush() (image.Image 方法), 60
format() (image.Image 方法), 60
format() (在 storage 模块中), 84

G

gaussian() (image.Image 方法), 63
gaussian_blur() (image.Image 方法), 63
get_framesize() (在 sensor 模块中), 46
get_hist() (image.Image 方法), 64
get_histogram() (image.Image 方法), 63
get_hmirror() (在 sensor 模块中), 47
get_id() (在 sensor 模块中), 47
get_ipa() (在 sensor 模块中), 47
get_ipa_stats() (在 sensor 模块中), 47
get_isp() (在 sensor 模块中), 47
get_mode() (在 uvc 模块中), 83
get_percentile() (image.histogram 方法), 58
get_pixel() (image.Image 方法), 60
get_pixformat() (在 sensor 模块中), 46
get_regression() (image.Image 方法), 64
get_statistics() (image.histogram 方法), 58
get_statistics() (image.Image 方法), 64
get_stats() (image.histogram 方法), 58
get_stats() (image.Image 方法), 64
get_threshold() (image.histogram 方法), 58
get_vflip() (在 sensor 模块中), 47
GRAYSCALE() (在 image 模块中), 50
GRAYSCALE() (在 sensor 模块中), 46
grayscale_to_binary() (在 image 模块中), 52
grayscale_to_lab() (在 image 模块中), 52
grayscale_to_rgb() (在 image 模块中), 52
grayscale_to_yuv() (在 image 模块中), 53

H

h() (image.barcode 方法), 57
h() (image.blob 方法), 55
h() (image.qrcode 方法), 56
h() (image.rect 方法), 54
h264
    module, 80
H264Encoder (h264 中的类), 80
height() (display.ESP32Display 方法), 67
height() (image.Image 方法), 60
height() (在 sensor 模块中), 47
histeq() (image.Image 方法), 62
histogram() (image.Image 方法), 64
histogram (image 中的类), 57

HMIRROR() (在 image 模块中), 50

I

I25() (在 image 模块中), 52
image
    module, 50
imageio
    module, 78
ImageIO (imageio 中的类), 78
ImageNetCls (espdl 中的类), 75
Image (image 中的类), 59
init() (espdl.NanoTrack 方法), 73
inputs() (espdl.Model 方法), 73
invert() (image.Image 方法), 62
IPAAFStats (sensor 中的类), 48
IPAAWBStats (sensor 中的类), 48
IPAStats (sensor 中的类), 48
is_alphanumeric() (image.qrcode 方法), 57
is_binary() (image.qrcode 方法), 57
is_closed() (imageio.ImageIO 方法), 78
is_kanji() (image.qrcode 方法), 57
is_numeric() (image.qrcode 方法), 57
is_streaming() (在 uvc 模块中), 83
ISBN10() (在 image 模块中), 52
ISBN13() (在 image 模块中), 52

J

JPEG() (在 image 模块中), 50
JPEG_SUBSAMPLING_420() (在 image 模块中), 51
JPEG_SUBSAMPLING_422() (在 image 模块中), 51
JPEG_SUBSAMPLING_444() (在 image 模块中), 51
JPEG_SUBSAMPLING_AUTO() (在 image 模块中), 51

K

keyframe() (h264.H264Encoder 方法), 80

L

l_bins() (image.histogram 方法), 57
l_lq() (image.statistics 方法), 59
l_max() (image.statistics 方法), 59
l_mean() (image.statistics 方法), 59
l_median() (image.statistics 方法), 59
l_min() (image.statistics 方法), 59
l_mode() (image.statistics 方法), 59
l_stdev() (image.statistics 方法), 59
l_uq() (image.statistics 方法), 59
l_value() (image.percentile 方法), 58
l_value() (image.threshold 方法), 58
lab_to_binary() (在 image 模块中), 53
lab_to_grayscale() (在 image 模块中), 53
lab_to_rgb() (在 image 模块中), 53
lab_to_yuv() (在 image 模块中), 53
laplacian() (image.Image 方法), 63
lcd_touch
    module, 68
length() (image.line 方法), 53
line() (image.line 方法), 53
line (image 中的类), 53
load_model() (在 espdl 模块中), 72
lq() (image.statistics 方法), 59

M

magnitude() (image.circle 方法), 54
magnitude() (image.line 方法), 54
magnitude() (image.rect 方法), 54
major_axis_line() (image.blob 方法), 56
mask() (image.qrcode 方法), 56
max() (image.statistics 方法), 59
mean() (image.Image 方法), 63
mean() (image.statistics 方法), 58
median() (image.Image 方法), 63
median() (image.statistics 方法), 58
MEMORY_STREAM() (在 imageio 模块中), 78
midpoint() (image.Image 方法), 63
min() (image.statistics 方法), 59
min_corners() (image.blob 方法), 54
minor_axis_line() (image.blob 方法), 56
mode() (image.Image 方法), 63
mode() (image.statistics 方法), 58
Model (espdl 中的类), 73
Model (tflite 中的类), 76
module
    display, 66
    espdl, 72
    h264, 80
    image, 50
    imageio, 78
    lcd_touch, 68
    rtsp, 82
    sensor, 46
    storage, 83
    tflite, 76
    uvc, 82
morph() (image.Image 方法), 63

N

NanoTrack (espdl 中的类), 73

O

offset() (imageio.ImageIO 方法), 78
open() (image.Image 方法), 62
outputs() (espdl.Model 方法), 73

P

PALETTE_DEPTH() (在 image 模块中), 50
PALETTE_EVT_DARK() (在 image 模块中), 50
PALETTE_EVT_LIGHT() (在 image 模块中), 50
PALETTE_IRONBOW() (在 image 模块中), 50
PALETTE_RAINBOW() (在 image 模块中), 50
payload() (image.barcode 方法), 57
payload() (image.qrcode 方法), 56

PDF417() (在 image 模块中), 52
PedestrianDetect (espdl 中的类), 74
percentile (image 中的类), 58
perimeter() (image.blob 方法), 55
pixels() (image.blob 方法), 55
PNG() (在 image 模块中), 50
predict() (espdl.Model 方法), 73
predict() (tflite.Model 方法), 77

Q

QQVGA() (在 sensor 模块中), 46
qrcode (image 中的类), 56
quality() (image.barcode 方法), 57
QVGA() (在 sensor 模块中), 46

R

r() (image.circle 方法), 54
read() (imageio.ImageIO 方法), 79
read() (lcd_touch.Touch 方法), 68
rect() (image.barcode 方法), 57
rect() (image.blob 方法), 54
rect() (image.qrcode 方法), 56
rect() (image.rect 方法), 54
rect (image 中的类), 54
reset() (在 sensor 模块中), 46
reset() (在 uvc 模块中), 83
retry() (在 storage 模块中), 84
RGB565() (在 image 模块中), 50
RGB565() (在 sensor 模块中), 46
rgb_to_binary() (在 image 模块中), 53
rgb_to_grayscale() (在 image 模块中), 53
rgb_to_lab() (在 image 模块中), 53
rgb_to_yuv() (在 image 模块中), 53
rho() (image.line 方法), 54
ROTATE_180() (在 image 模块中), 51
ROTATE_270() (在 image 模块中), 51
ROTATE_90() (在 image 模块中), 51
rotation() (image.barcode 方法), 57
rotation() (image.blob 方法), 55
rotation_deg() (image.blob 方法), 55
rotation_rad() (image.blob 方法), 55
roundness() (image.blob 方法), 55
rtsp
    module, 82
RTSPServer (rtsp 中的类), 82

S

save() (image.Image 方法), 60
scale() (image.Image 方法), 60
SCALE_ASPECT_EXPAND() (在 image 模块中), 51
SCALE_ASPECT_IGNORE() (在 image 模块中), 51
SCALE_ASPECT_KEEP()(在 image 模块中), 51
SEARCH_DS() (在 image 模块中), 51
SEARCH_EX() (在 image 模块中), 51
seek() (imageio.ImageIO 方法), 79
send() (rtsp.RTSPServer 方法), 82
sensor
    module, 46
SensorStatus (sensor 中的类), 48
set_frame() (在 uvc 模块中), 83
set_framesize() (在 sensor 模块中), 46
set_hmirror() (在 sensor 模块中), 47
set_ipa() (在 sensor 模块中), 47
set_isp() (在 sensor 模块中), 47
set_pixel() (image.Image 方法), 60
set_pixformat() (在 sensor 模块中), 46
set_thresholds() (espdl.ESPDet 方法), 74
set_thresholds() (espdl.ImageNetCls 方法), 75
set_thresholds() (espdl.PedestrianDetect 方法), 74
set_thresholds() (espdl.YOLO11 方法), 74
set_thresholds() (espdl.YOLO11nPose 方法), 75
set_vflip() (在 sensor 模块中), 47
shutdown() (在 sensor 模块中), 46
size() (image.Image 方法), 60
size() (imageio.ImageIO 方法), 78
skip_frames() (在 sensor 模块中), 47
snapshot() (在 sensor 模块中), 48
solidity() (image.blob 方法), 56
statistics() (image.histogram 方法), 58
statistics() (image.Image 方法), 64
statistics (image 中的类), 58
status() (在 sensor 模块中), 48
status() (在 storage 模块中), 83
stdev() (image.statistics 方法), 58
stop() (rtsp.RTSPServer 方法), 82
storage
    module, 83
sync() (imageio.ImageIO 方法), 79

T

TAG16H5() (在 image 模块中), 52
TAG25H7() (在 image 模块中), 52
TAG25H9() (在 image 模块中), 52
TAG36H10() (在 image 模块中), 52
TAG36H11() (在 image 模块中), 52
tflite
    module, 76
theta() (image.line 方法), 54
threshold (image 中的类), 58
to_bitmap() (image.Image 方法), 61
to_grayscale() (image.Image 方法), 61
to_ironbow() (image.Image 方法), 61
to_jpeg() (image.Image 方法), 61
to_png() (image.Image 方法), 61
to_rainbow() (image.Image 方法), 61
to_rgb565() (image.Image 方法), 61
Touch (lcd_touch 中的类), 68
track() (espdl.NanoTrack 方法), 73

TRANSPOSE() (在 image 模块中), 50
type() (image.barcode 方法), 57
type() (imageio.ImageIO 方法), 78

U

UPCA() (在 image 模块中), 52
UPCE() (在 image 模块中), 52
uq() (image.statistics 方法), 59
uvc
    module, 82

V

value() (image.percentile 方法), 58
value() (image.threshold 方法), 58
version() (image.qrcode 方法), 56
version() (imageio.ImageIO 方法), 78
VFLIP() (在 image 模块中), 50
VGA() (在 sensor 模块中), 46

W

w() (image.barcode 方法), 57
w() (image.blob 方法), 55
w() (image.qrcode 方法), 56
w() (image.rect 方法), 54
width() (display.ESP32Display 方法), 67
width() (image.Image 方法), 60
width() (在 sensor 模块中), 47
write() (display.ESP32Display 方法), 67
write() (imageio.ImageIO 方法), 79
write() (在 uvc 模块中), 82

X

x() (image.barcode 方法), 57
x() (image.blob 方法), 55
x() (image.circle 方法), 54
x() (image.qrcode 方法), 56
x() (image.rect 方法), 54
x1() (image.line 方法), 53
x2() (image.line 方法), 53
x_hist_bins() (image.blob 方法), 56

Y

y() (image.barcode 方法), 57
y() (image.blob 方法), 55
y() (image.circle 方法), 54
y() (image.qrcode 方法), 56
y() (image.rect 方法), 54
y1() (image.line 方法), 53
y2() (image.line 方法), 53
y_hist_bins() (image.blob 方法), 56
YOLO11nPose (espdl 中的类), 75
YOLO11 (espdl 中的类), 74
YUV422() (在 image 模块中), 50
yuv_to_binary() (在 image 模块中), 53
yuv_to_grayscale() (在 image 模块中), 53
yuv_to_lab() (在 image 模块中), 53
yuv_to_rgb() (在 image 模块中), 53