



ESP32-P4

ESP-VISION 编程指南



Release 2026.07.28-32-gecda8004b9
乐鑫信息科技
2026 年 09 月 29 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	6
2.2 芯片能力与板级能力	7
2.3 标准 MicroPython 功能	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 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	29
5	API 参考	31
5.1	MicroPython API	31
5.1.1	文档基线	31
5.1.2	语言与内置 API	31
5.1.3	标准库与微型库	32
5.1.4	JSON 与时间示例	32
5.1.5	硬件与 ESP32 API	32
5.1.6	GPIO 与 PWM 示例	32
5.1.7	网络 API	32
5.1.8	连接 Wi-Fi	33
5.1.9	文件系统与运行时 API	33
5.1.10	列出存储并检查内存	33
5.1.11	组织复杂应用	33
5.1.12	线程与原生任务	35
5.1.13	检查固件中的可用性	36
5.2	sensor-摄像头	36
5.2.1	摄像头初始化与连续采集	36
5.2.2	图像方向与摄像头状态	36
5.2.3	暂时停止采集	37
5.2.4	运行时控制 IPA	37
5.2.5	手动控制 ISP	37
5.2.6	Constants	39
5.2.7	Functions	39
5.2.8	Classes	40
5.3	image-图像处理	41
5.3.1	绘图与颜色色块追踪	41
5.3.2	滤波与二值分割	41
5.3.3	二维码识别	41
5.3.4	直线、线段与圆检测	42
5.3.5	编码并保存图像	42
5.3.6	Constants	42
5.3.7	Functions	45
5.3.8	Classes	46
5.4	display-LCD 显示	58

5.4.1	摄像头预览	58
5.4.2	定位、缩放与裁剪	58
5.4.3	背光与资源释放	58
5.4.4	Classes	59
5.5	espd1-模型推理	59
5.5.1	原始输出 <code>tensor</code>	59
5.5.2	目标检测	60
5.5.3	PicoDet	60
5.5.4	PedestrianDetect	60
5.5.5	图像分类	60
5.5.6	姿态估计	61
5.5.7	运行时调整阈值	61
5.5.8	结果元组	61
5.5.9	Functions	61
5.5.10	Classes	62
5.6	tfLite-模型推理	64
5.6.1	基本用法	64
5.6.2	Callable 输入	65
5.6.3	后处理	65
5.6.4	Classes	65
5.7	image.ImageIO-图像流	66
5.7.1	录制到存储	66
5.7.2	回放已录制的图像流	66
5.7.3	使用内存图像流	66
5.7.4	Constants	67
5.7.5	Classes	67
5.8	h264-H.264 编码	68
5.8.1	编码单帧图像	68
5.8.2	录制 H.264 裸码流	68
5.8.3	资源与吞吐量注意事项	69
5.8.4	Classes	69
5.9	rtsp-RTSP 推流	69
5.9.1	采集、编码与推流	70
5.9.2	客户端与队列行为	70
5.9.3	停止服务	70
5.9.4	Classes	71
6	操作指南	73
6.1	添加新的 Python 模块	73
6.1.1	总览	73
6.1.2	1. 创建绑定源文件	73
6.1.3	2. 按需注册 <code>qstr</code>	74
6.1.4	3. 将源文件接入构建	74
6.1.5	4. 添加类型存根	74
6.1.6	5. 编写模块文档	74
6.1.7	6. 构建并验证	74
6.2	客制化固件功能	75
6.2.1	确定客制化范围	75
6.2.2	客制化 ESP-VISION Python 模块	75
6.2.3	客制化图像算法	75
6.2.4	客制化标准 MicroPython 功能	75
6.2.5	客制化冻结 Python 代码	76
6.2.6	客制化板级服务与可选组件	76
6.2.7	构建与验证	76
6.3	引入新的模型	76
6.3.1	1. 获取或转换模型	76
6.3.2	2. 将模型拷贝到板级存储	77
6.3.3	3. 选择合适的 API	77

6.3.4	4. 运行 ESP-DL 推理	77
6.3.5	5. 在 Python 中解码 ESP-DL 输出	77
6.3.6	6. 运行 TFLite Micro 推理	78
6.3.7	7. 可选：性能分析和验证	78
6.4	训练 ESPDet Pico 模型	78
6.4.1	1. 准备 ESP-Detection 环境	79
6.4.2	2. 准备数据集	79
6.4.3	3. 训练 ESPDet Pico	80
6.4.4	4. 准备校准数据	80
6.4.5	5. 量化并导出 .espdl	81
6.4.6	6. 添加模型元数据	81
6.4.7	7. 拷贝模型到板端存储	82
6.4.8	8. 在 ESP-VISION 上运行	82
6.4.9	9. 验证和调参	83
6.5	添加新的开发板	83
6.5.1	MicroPython 移植侧	83
6.5.2	ESP-VISION 侧	84
6.5.3	构建与烧录	84
6.6	Agent 代码生成与 EV-MUX 闭环测试	84
6.6.1	依赖规则	84
6.6.2	All-in-One 输出	84
6.6.3	USB-OTG CDC 与 USJ 传输约定	85
7	方案架构	91
7.1	分层概览	92
7.2	采集到输出的数据流	93
7.3	底层传输机制	93
7.3.1	物理 sink	93
7.3.2	路由 stream	93
7.3.3	EV-MUX channel	94
7.3.4	EV-MUX 帧格式	94
7.3.5	EV-ATP 控制	95
7.3.6	Host 对接契约	95
7.3.7	接收分流原则	95
7.3.8	传输执行模型	96
7.3.9	IDF console 与接口选择	96
7.3.10	传输可靠性与观测	97
7.4	源码结构	97
7.5	板卡的组成	97
7.6	随芯片变化的源码	97
7.7	MicroPython Overlay	98
8	项目关系	99
8.1	ESP-VISION、MicroPython 与 OpenMV	99
8.2	依赖层次	99
9	许可证	101
9.1	许可证清单	101
9.2	如何理解许可证	101
9.3	OpenMV 与 GPL 代码路径	102
9.4	增加第三方代码	102
Python 模块索引		103
Python 模块索引		103
索引		105
索引		105

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](#)。

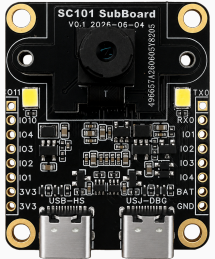
构建与烧录固件请参阅[快速入门](#)，整体架构请参阅[方案架构](#)。

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

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



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。

Chapter 3

快速入门

3.1 前置条件

- 受支持的 ESP-IDF 环境，并已 source 导出脚本，使 `idf.py` 位于 PATH 中；当前分支兼容性请查看项目 README。
- 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。

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> -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 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

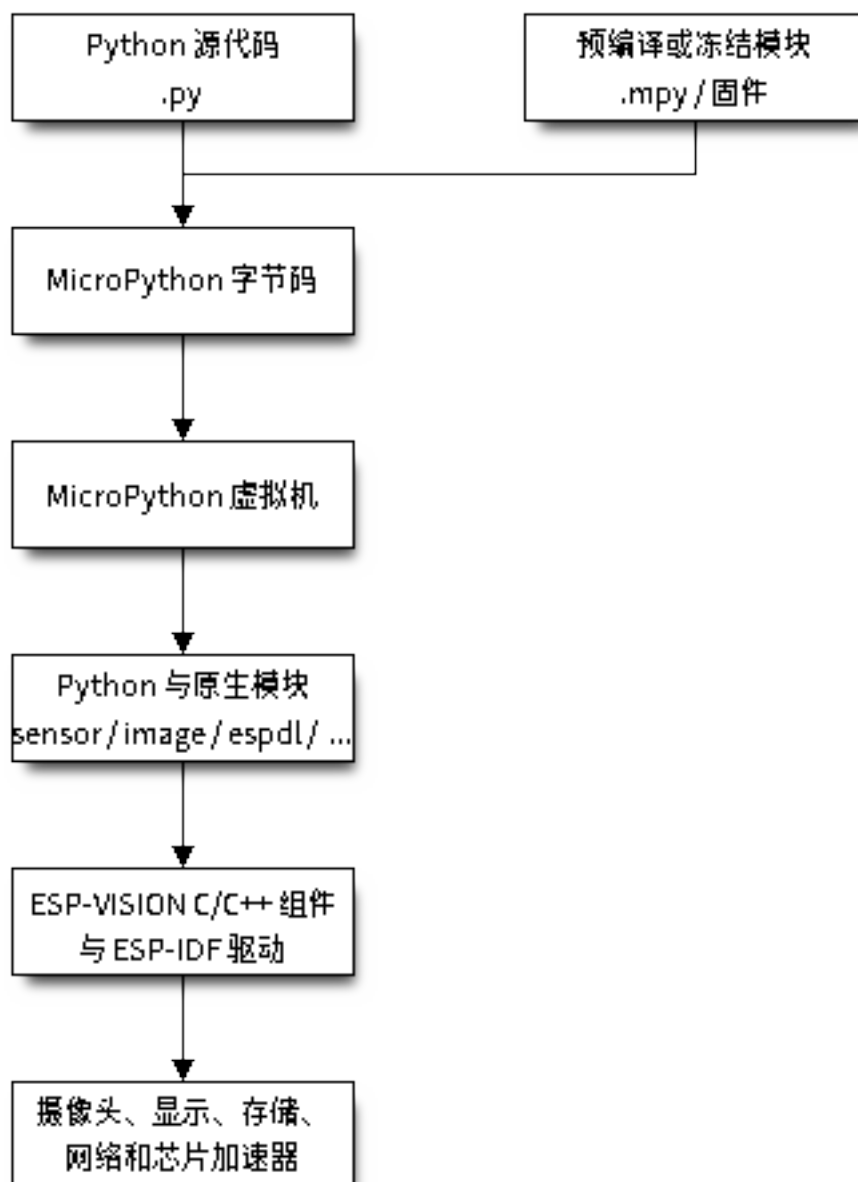
本章解释 ESP-VISION API 背后的原理：图像在内存中的表示方式、各类图像处理算法的工作机制、神经网络推理如何在设备上运行，以及图像帧如何在摄像头与编解码流水线中流转。建议与[API 参考](#) 对照阅读，理解每个函数“为什么”这样工作。

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)

```

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 int 与 float	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。 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。 <code>append()</code> 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 <code>machine</code> 、 <code>micropython</code> 、 <code>esp</code> 和 <code>esp32</code> 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 <code>.mpy</code> 或冻结到固件中；不能假定支持桌面 <code>pip</code> 和任意二进制 <code>wheel</code> 。	通常使用 <code>pip</code> 、虚拟环境、源码发行包和平台 <code>wheel</code> 。
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 <code>asyncio</code> 、已启用且带 GIL 的 <code>_thread</code> ，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 <code>asyncio</code> 、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 of 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：

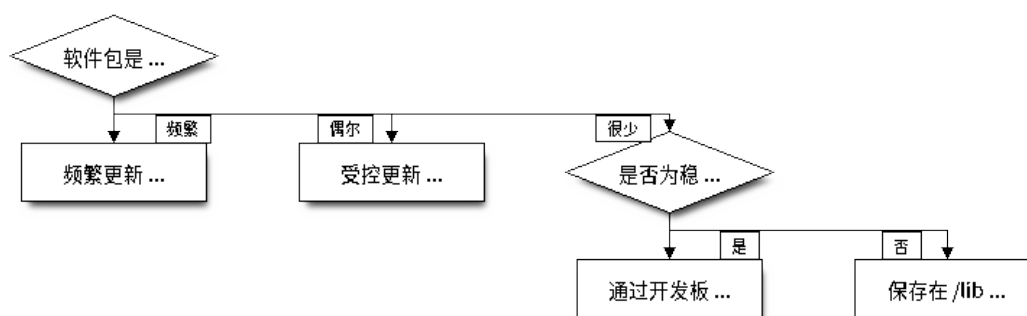


图 1: 软件包部署决策

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 <code>image.Image.binary()</code> 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	由 <code>image.Image.to_jpeg()</code> / <code>to_png</code> 生成的压缩字节流。

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)

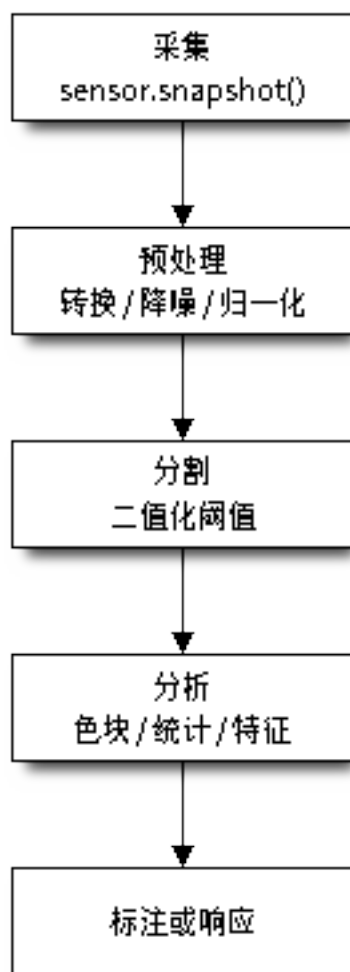
多数分析与转换方法接受 `roi=(x, y, w, h)` 关键字，把操作限制在图像的某个子矩形内。在 ROI 上工作既更快（像素更少）又更有针对性（忽略无关区域）。ROI 始终以源图像的像素坐标表示。

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 $(2 \times \text{ksize} + 1)$ 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

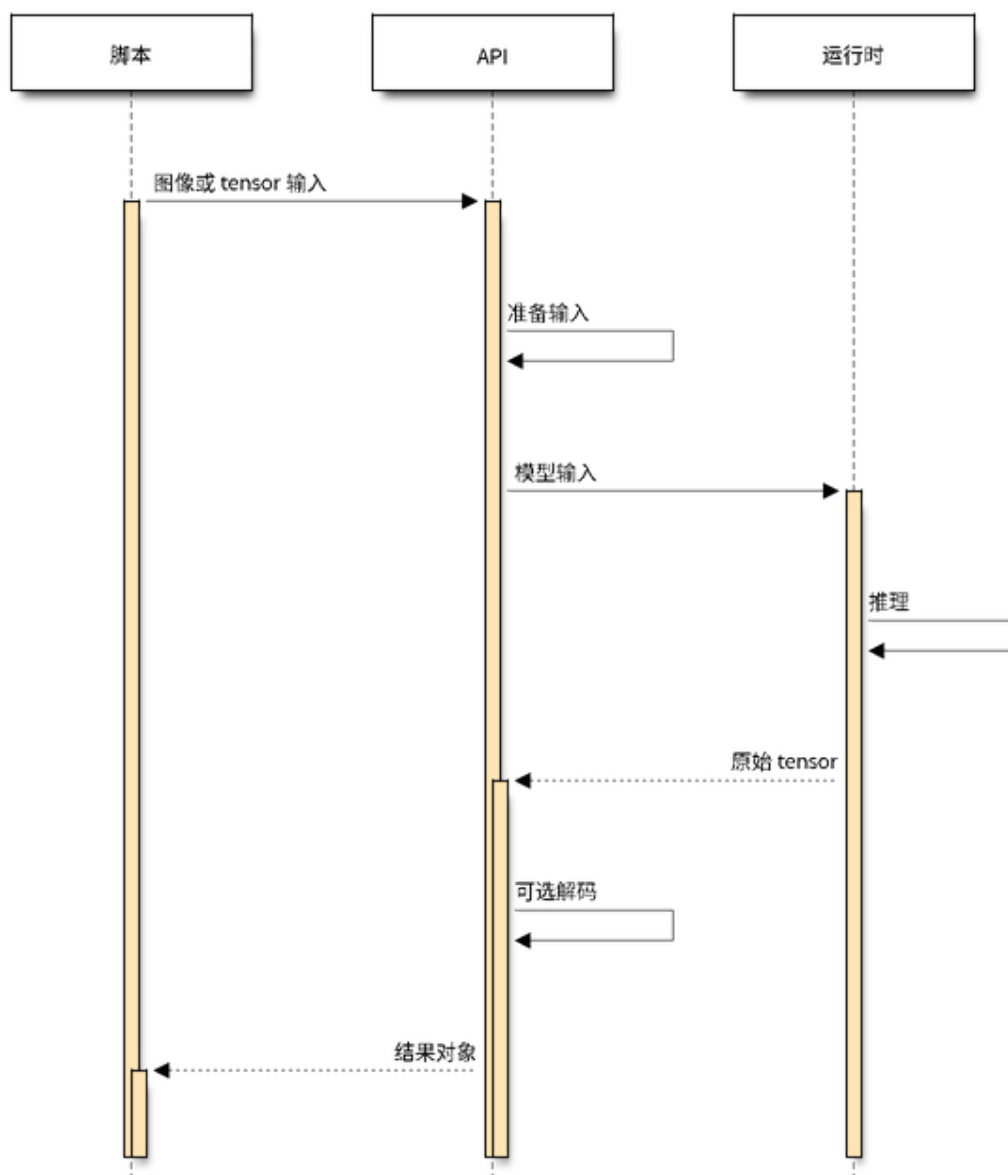
ESP-DL 和 TFLite Micro 是 ESP-VISION 当前暴露的两条模型运行路径。espd1 模块使用 .espd1 文件，tflite.Model 使用 .tflite 文件。模型文件可放在 /sdcard 或 /flash 等板级存储中，并在运行时加载。具体 API 形态、量化元数据、输入输出解释由所选运行时和模型共同定义。

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 (`espd1.PedestrianDetect`、`espd1.ESPDet`、`espd1.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`）。`image.Image.to_png()` 产生无损 PNG，体积更大但精确，适合保存参考帧。

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

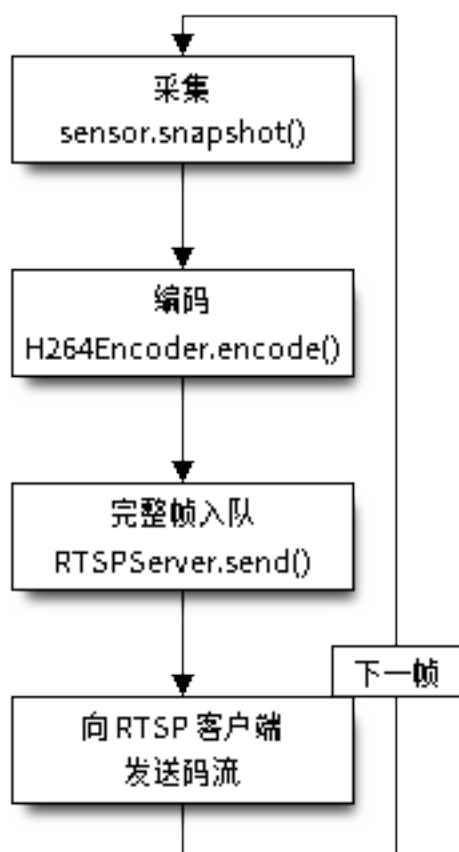
对于连续视频，逐帧 JPEG 很浪费，因为它无法利用相邻帧之间的相似性。`h264.H264Encoder` 产生 H.264 流，其中偶发的关键帧（IDR/I）是自包含的，而其间的帧只编码发生变化的部分。编码器参数用于调节这一权衡：

- `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()`。

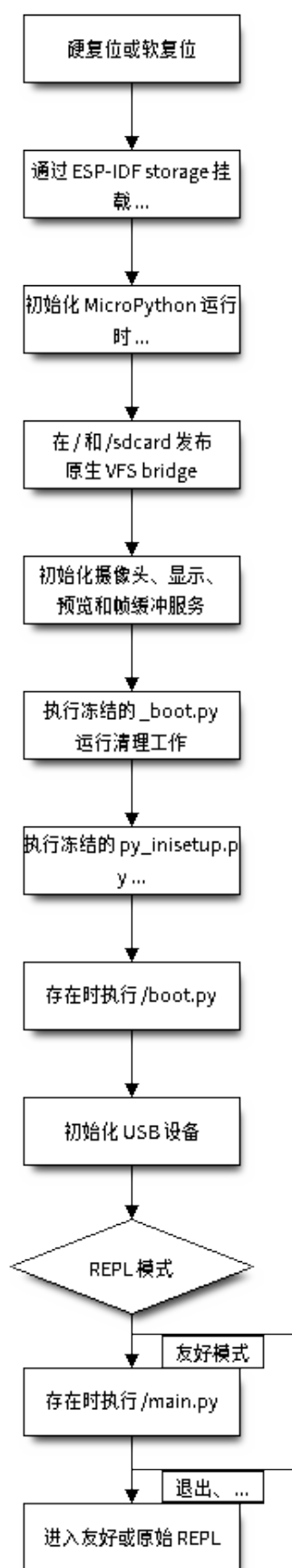
软复位在不重启完整 MCU 运行时的情况下重新启动 MicroPython 环境，可通过友好 REPL 中的 Ctrl-D 或 `machine.soft_reset()` 触发。ESP-VISION 会清除 Python 对象和模块，通过 MicroPython 清理过

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：



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。它在内部将 `ffat` 分区挂载到 `/flash`，并将可用的 SD 卡挂载到 `/sdcard`。每次 VM 启动时，原生 MicroPython VFS bridge 会把 Flash 卷发布到 `/`，把 SD 卷发布到 `/sdcard`。软复位只会重新创建 `bridge` 对象，不会重新挂载已挂载的物理文件系统；如果 SD 卡在冷启动时未挂载，后续每次 VM 启动都会重试，因此开机后插入的 SD 卡可在软复位后变为可用。

storage manager 会在初始化时将未格式化的 `ffat` 分区格式化为 FAT。兼容旧固件的 `vfs` 分区只有在已经包含 FAT 时才会挂载，绝不会自动格式化，因此不会静默擦除可能存在的 `LittleFS2` 卷。其他挂载或写入失败只会上报，不会执行破坏性修复。随后，冻结的 `py_inisetup.py` 会在对应文件不存在时创建 `/boot.py`、`/main.py`、`/README.txt` 和 `/.esp_vision_disk` 标记文件。

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

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、espd1 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。

5.1 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、espd1 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。

5.1.1 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。

5.1.2 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)
```

5.1.3 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。

5.1.4 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。

5.1.5 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。

5.1.6 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。

5.1.7 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`、`WebREPL` 和 `socket event` 支持取决于构建固件使用的 ESP-IDF 版本；维护中的支持摘要见 [芯片与开发板支持](#)。

5.1.8 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，而不是无限期待。

5.1.9 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。

5.1.10 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"))

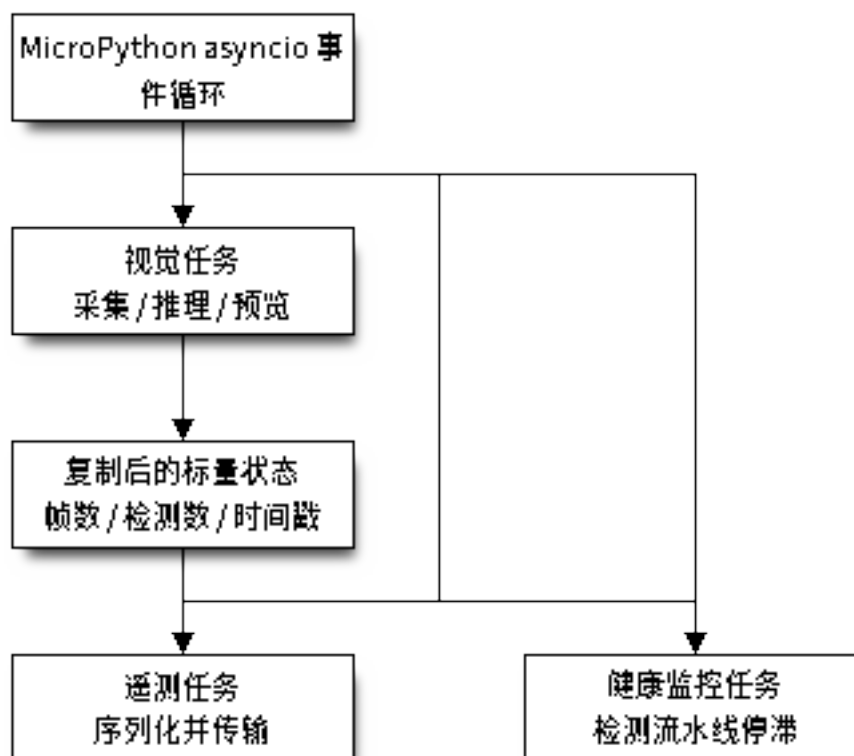
gc.collect()
print("free heap:", gc.mem_free())
```

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。

5.1.11 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 espd1
import json
import sensor
import time

MODEL = "/sdcard/espdet_pico_224_224_face.espd1"

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)

```

        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 = espd1.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。

5.1.12 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`。

5.1.13 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：

仅使用标准 MicroPython API 的可运行示例：`example/00-HelloWorld/helloworld.py`（首个脚本）、`example/06-Peripherals/00-Storage/sdcard.py`（文件系统）与 `example/06-Peripherals/02-WiFi/webrepl.py`（Wi-Fi 与 WebREPL）。

5.2 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。

5.2.1 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。

5.2.2 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()
```

(续下页)

(接上页)

```

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。

5.2.3 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。

5.2.4 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。此开关控制哪些算法取决于当前 IPA 配置。

5.2.5 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。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)
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)
```

处理模块名与字段沿用调参参数的命名：

表 1: 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 <code>get_isp("lsc")</code> 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，因此写入被拒绝时画面不受影响。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)。

5.2.6 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`

5.2.7 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. Unsupported cameras raise `OSError`.

`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_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.

`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`.

`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 –reserved for compatibility; pass `None` in current ESP-VISION builds.

`sensor.status()`

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

5.2.8 Classes

class `sensor.SensorStatus`

Dictionary returned by `status()`.

5.3 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()`。

5.3.1 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。

5.3.2 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。

5.3.3 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。

5.3.4 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。

5.3.5 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）。

5.3.6 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`

5.3.7 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.

5.3.8 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).

cx_f ()

Centroid x (float).

cy ()

Centroid y (integer).

cy_f ()

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.histogramChannel 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).

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 `image.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*, *, *copy_to_fb=...*)

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

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

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.
- **copy_to_fb** –True loads the decoded image into the frame buffer.

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.

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=..., transform=..., 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/TRANSPPOSE/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.

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=..., transform=..., 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).

5.4 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。

5.4.1 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。

5.4.2 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。

5.4.3 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。

5.4.4 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.

5.5 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) 和图像分类 (ImageNetCls)，同时可通过 Model 将 ESP-DL 原始输出 tensor 暴露给 Python。

5.5.1 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())
```

(续下页)

(接上页)

```

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()

```

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`。

5.5.2 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。

5.5.3 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。

5.5.4 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,
)

```

5.5.5 图像分类

```

import espdl, image

img = image.Image("/sdcard/cat.jpg").to_rgb565(copy=True)
classifier = espdl.ImageNetCls(

```

(续下页)

(接上页)

```

    "/sdcard/imagenet_cls_mobilenetv2_s8_v1.espd1",
    topk=5,
    score=0.0,
)
try:
    for label, score in classifier.classify(img):
        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。

5.5.6 姿态估计

```

pose = espd1.YOLO11nPose("/sdcard/espdet_yolo11n_pose_160_160_coco.espd1", 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。

5.5.7 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。

5.5.8 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)

参见:

[AI 推理](#) 介绍 ESP-DL 推理流程、.espd1 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)。

5.5.9 Functions

`espd1.load_model(path, *, profile=...)`

Preload an ESP-DL model file.

参数

- **path** –model path, for example “/sdcard/model.espd1” or “/flash/model.espd1” .
- **profile** –True enables ESP-DL profiling output when supported.

5.5.10 Classes

class `espd1.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 .espd1 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 `espd1.ESPDet(path, *, score=..., nms=..., mean=..., std=...)`

ESP-DL object detection wrapper for ESPDet models.

Create a detector from an .espd1 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 `espd1.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 `espd1.YOLO11` (*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11 object detection wrapper.

Create a YOLO11 detector from an .espd1 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 `espd1.YOLO11nPose` (*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11n COCO pose wrapper.

Create a YOLO11n pose detector from an .espd1 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.

5.6 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。

5.6.1 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。

5.6.2 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。

5.6.3 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`）。

5.6.4 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.

5.7 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；类型根位于 `stubs/imageio.pyi`。

5.7.1 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。

5.7.2 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`。

5.7.3 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。

5.7.4 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.

5.7.5 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.

5.8 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。

5.8.1 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。

5.8.2 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)
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:
```

(续下页)

```

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。

5.8.3 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`。

5.8.4 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`.

5.9 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。

5.9.1 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。

5.9.2 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`。

5.9.3 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`。

5.9.4 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.

Chapter 6

操作指南

面向具体任务的 ESP-VISION 扩展指南。

6.1 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。

6.1.1 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
```

6.1.2 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) },
};
```

(续下页)

(接上页)

```
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`。

6.1.3 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`。

6.1.4 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。

6.1.5 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。

6.1.6 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`。

6.1.7 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())
```

6.2 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。

6.2.1 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。

6.2.2 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。

6.2.3 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。未经许可审查，不应移除 `OMV_NO_GPL`。

6.2.4 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。

6.2.5 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。

6.2.6 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。

6.2.7 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。

6.3 引入新的模型

ESP-VISION 支持两类模型运行路径：ESP-DL `.espd1` 模型使用 [espd1-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。

6.3.1 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 `.espd1`，或使用 ESP-DL 的量化/导出工具链将自有模型转换为 `.espd1` 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/`。

6.3.2 2. 将模型拷贝到板级存储

将 `.espd1` 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.espd1`。

6.3.3 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)	<code>espd1.ESPDet</code>	(x, y, w, h, score, category)
ESP-DL 目标检测 (YOLO11)	<code>espd1.YOLO11</code>	(x, y, w, h, score, category)
ESP-DL 姿态检测	<code>espd1.YOLO11nPose</code>	检测结果加 17 个 COCO 关键点
ESP-DL 图像分类	<code>espd1.ImageNetCls</code>	(label, score)
ESP-DL 输出由 Python 解码	<code>espd1.Model</code>	带 tensor 元数据的原始输出字节
通用 TFLite Micro 执行	<code>tflite.Model</code>	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 `espd1.Model.inputs()` 和 `espd1.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。

6.3.4 4. 运行 ESP-DL 推理

```
import sensor, image, espd1

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

det = espd1.ESPDet("/sdcard/my_model.espd1", 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()
```

6.3.5 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 `espd1.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。

6.3.6 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)`。

6.3.7 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。

6.4 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。

6.4.1 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
```

6.4.2 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`：

```
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 `espd1.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：

```
names:
  0: helmet
  1: person
  2: vest
```

6.4.3 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
<code>--class_name</code>	训练任务名称，会影响 Ultralytics 输出目录名称。
<code>--pretrained_path</code>	预训练权重路径。没有预训练权重时使用 <code>None</code> ，脚本会从 <code>cfg/models/espdet_pico.yaml</code> 构建新模型。
<code>--dataset</code>	数据集 YAML 路径。
<code>--size</code>	输入尺寸，格式为 <code>height width</code> 。例如 <code>320 320</code> 表示 <code>320x320</code> 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`。

6.4.4 4. 准备校准数据

导出 `.espd1` 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。

6.4.5 5. 量化并导出.espd1

使用 `deploy.quantize_aligned` 将训练好的 `best.pt` 导出为 ESP-DL 可加载的 `.espd1` 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 \
  --espd1 deploy/espdet_pico_320_320_my_object.espd1 \
  --data cfg/datasets/my_object.yaml
```

参数说明：

参数	说明
<code>--model</code>	训练得到的 <code>best.pt</code> ，也可以是已导出的 ONNX。
<code>--size</code>	模型输入尺寸，必须和训练尺寸一致。
<code>--target</code>	目标芯片，例如 <code>esp32p4</code> 或 <code>esp32s3</code> 。应和最终运行模型的芯片匹配。
<code>--calib_data</code>	校准图片目录。
<code>--espd1</code>	导出的 ESP-DL 模型路径。
<code>--data</code>	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 `.espd1` 文件存在：

```
ls -lh deploy/espdet_pico_320_320_my_object.espd1
```

6.4.6 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.espd1
  espdet_pico_320_320_my_object.json
```

其中 `.espd1` 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": "espd1.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.espd1
```

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。

6.4.7 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.espd1"
```

如果复制到 flash 根目录，则路径通常为：

```
MODEL = "/flash/espdet_pico_320_320_my_object.espd1"
```

6.4.8 8. 在 ESP-VISION 上运行

下面是一个完整的 MicroPython 推理脚本：

```
import espd1
import sensor
import time

MODEL = "/sdcard/espdet_pico_320_320_my_object.espd1"
LABELS = ("my_object",)

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

det = espd1.ESPDet(MODEL, score=0.5, nms=0.7)
```

(续下页)

(接上页)

```

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。

6.4.9 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。

6.5 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。构建时会把 boards/<BOARD>/port/ 投射到生成的 MicroPython 副本的 ports/esp32/boards/<BOARD>/。建议从 TEMPLATE 板开始。

6.5.1 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。

6.5.2 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。

6.5.3 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。

6.6 Agent 代码生成与 EV-MUX 闭环测试

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。

6.6.1 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。

6.6.2 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()

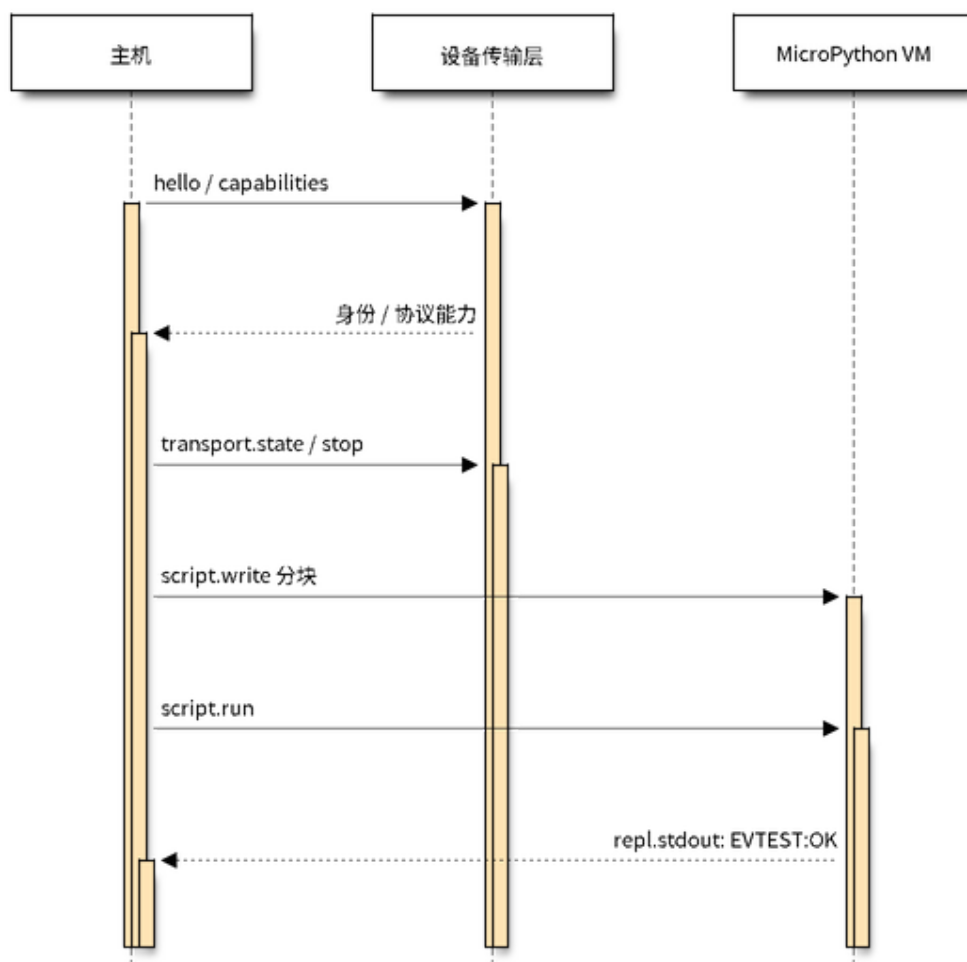
```

6.6.3 USB-OTG CDC 与 USJ 传输约定

ESP-VISION 在 USB-OTG CDC (sink cdc) 和 USB-Serial-JTAG (sink usj) 上承载相同的 EV-MUX 协议。每次测试只打开一个设备端口。以 DTR 有效状态打开 CDC 端口后，CDC 会成为自动路由；否则支持的开发板会回退到 USJ。测试时不要同时打开两个端口，也不要保留其他串口监视器连接。USB 波特率不属于协议，但主机串口 API 仍需一个占位波特率。

每个帧由 0x1e、EVMUX/1 h=<metadata_len> p=<payload_len> c=<crc32>\r\n、精确 h 字节的 UTF-8 JSON metadata、精确 p 字节的 payload 以及最后的 0x1f 组成。metadata 和 payload 边界由长度而不是分隔符决定。设备帧包含 payload CRC32；主机命令可发送零 CRC，但建议仍计算 CRC。接收端必须在 0x1e 上重新同步，验证长度与 CRC，然后按 metadata.channel 分发每个帧，不能按 USB 端口或到达顺序推断帧类型。

最小闭环是：打开所选端口；交换 hello 与 capabilities；验证 firmware.id == "esp-vision" 及协商的 evMuxVersion；查询 transport.state；停止当前脚本以进入 REPL；通过 script.write 上传一个完整的设备脚本；通过 script.run 启动；并必须在 repl.stdout 上看到期望标记。script.run 响应只表示脚本已入队，不能单独作为应用测试成功的依据。



下面的仅主机端 CPython 脚本对 USB-OTG CDC 或 USJ 实现完整闭环。在 PC 上执行 `python -m pip install pyserial` 安装 pyserial，将下面内容保存为一个文件，关闭其他串口工具，然后使用所选设备端口运行，例如 `python evmux_smoke.py /dev/ttyACM0`。要上传的 MicroPython 程序已嵌入主机脚本，无需第二个源文件。

```
#!/usr/bin/env python3
import argparse
import base64
import json
import re
import sys
import time
import zlib

import serial

DEVICE_SCRIPT = """\
import gc
import sensor

def main():
    print("EVTEST:START")
    sensor.reset()
    try:
```

(续下页)

(接上页)

```

        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()
"""

SOF = b"\x1e"
EOF = b"\x1f"
HEADER_RE = re.compile(
    rb"EVMUX/1 h=([0-9]+) p=([0-9]+) c=([0-9A-Fa-f]{8})\r\n")
MAX_METADATA = 64 * 1024
MAX_PAYLOAD = 16 * 1024 * 1024
WRITE_CHUNK = 768

class EvMux:
    def __init__(self, port):
        self.port = port
        self.next_seq = 1
        self.text_tail = bytearray()

    def _read_exact(self, size, deadline):
        data = bytearray()
        while len(data) < size:
            if time.monotonic() >= deadline:
                raise TimeoutError("timed out while reading EV-MUX frame")
            part = self.port.read(size - len(data))
            if part:
                data.extend(part)
        return bytes(data)

    def read_frame(self, deadline):
        while time.monotonic() < deadline:
            if self.port.read(1) != SOF:
                continue

            header = bytearray()
            while len(header) < 80:
                char = self._read_exact(1, deadline)
                header.extend(char)
                if char == b"\n":
                    break
            match = HEADER_RE.fullmatch(bytes(header))
            if match is None:
                continue

            metadata_len = int(match.group(1))
            payload_len = int(match.group(2))
            expected_crc = int(match.group(3), 16)
            if not 0 < metadata_len <= MAX_METADATA:
                raise RuntimeError("invalid metadata length: {}".format(metadata_
↵len))

            if not 0 <= payload_len <= MAX_PAYLOAD:

```

(续下页)

(接上页)

```

        raise RuntimeError("invalid payload length: {}".format(payload_
↪len))

    metadata_bytes = self._read_exact(metadata_len, deadline)
    payload = self._read_exact(payload_len, deadline)
    if self._read_exact(1, deadline) != EOF:
        raise RuntimeError("invalid EV-MUX EOF")
    actual_crc = zlib.crc32(payload) & 0xFFFFFFFF
    if expected_crc and actual_crc != expected_crc:
        raise RuntimeError("EV-MUX payload CRC mismatch")
    metadata = json.loads(metadata_bytes.decode("utf-8"))
    return metadata, payload
    raise TimeoutError("timed out waiting for EV-MUX frame")

def send_request(self, channel, method, body):
    seq = self.next_seq
    self.next_seq += 1
    payload = json.dumps(body, separators=(",", ":")).encode("utf-8")
    metadata = {
        "sid": "debug" if channel == "debug.rpc" else "user",
        "seq": seq,
        "channel": channel,
        "type": "req",
        "method": method,
        "encoding": "json",
        "ts_ms": int(time.monotonic() * 1000),
    }
    metadata_bytes = json.dumps(
        metadata, separators=(",", ":")).encode("utf-8")
    crc = zlib.crc32(payload) & 0xFFFFFFFF
    header = "EVMUX/1 h={} p={} c={:08X}\r\n".format(
        len(metadata_bytes), len(payload), crc).encode("ascii")
    self.port.write(SOF + header + metadata_bytes + payload + EOF)
    self.port.flush()
    return seq

def _observe(self, metadata, payload):
    channel = metadata.get("channel")
    if channel in ("repl.stdout", "repl.stderr", "log.idf"):
        self.text_tail.extend(payload)
        del self.text_tail[:-8192]
        text = payload.decode("utf-8", errors="replace")
        sys.stdout.write("[{}] {}".format(channel, text))
        sys.stdout.flush()

def rpc(self, channel, method, body, timeout=10):
    seq = self.send_request(channel, method, body)
    deadline = time.monotonic() + timeout
    while True:
        metadata, payload = self.read_frame(deadline)
        self._observe(metadata, payload)
        if (metadata.get("channel") != channel
            or metadata.get("type") != "rsp"
            or metadata.get("seq") != seq):
            continue
        if metadata.get("encoding") != "json":
            raise RuntimeError("unexpected binary RPC response")
        result = json.loads(payload.decode("utf-8"))
        if not result.get("ok"):
            raise RuntimeError("{} failed: {}".format(method, result))
        return result

```

(续下页)

(接上页)

```

def wait_for_text(self, marker, timeout=15):
    deadline = time.monotonic() + timeout
    while marker not in self.text_tail:
        metadata, payload = self.read_frame(deadline)
        self._observe(metadata, payload)
    return bytes(self.text_tail)

def upload_script(mux, path, source):
    content = source.encode("utf-8")
    for offset in range(0, len(content), WRITE_CHUNK):
        chunk = content[offset:offset + WRITE_CHUNK]
        result = mux.rpc("user.rpc", "script.write", {
            "path": path,
            "mode": "overwrite",
            "encoding": "utf-8",
            "contentBase64": base64.b64encode(chunk).decode("ascii"),
            "offset": offset,
            "totalBytes": len(content),
        })
        expected_complete = offset + len(chunk) == len(content)
        if result.get("complete") != expected_complete:
            raise RuntimeError("unexpected script.write completion state")

def main():
    parser = argparse.ArgumentParser(
        description="ESP-VISION EV-MUX USB/USJ closed-loop smoke test")
    parser.add_argument("port", help="USB-OTG CDC or USB-Serial-JTAG port")
    args = parser.parse_args()

    with serial.Serial(
        args.port, baudrate=115200, timeout=0.2, write_timeout=5) as port:
        try:
            port.dtr = True
        except (OSError, serial.SerialException):
            pass
        port.reset_input_buffer()
        time.sleep(0.1)
        mux = EvMux(port)

        hello = mux.rpc("user.rpc", "hello", {})
        if hello.get("firmware", {}).get("id") != "esp-vision":
            raise RuntimeError("connected device is not ESP-VISION")

        capabilities = mux.rpc("user.rpc", "capabilities", {})
        version = capabilities.get("protocol", {}).get("evMuxVersion")
        if version != 3:
            raise RuntimeError("unsupported EV-MUX version: {}".format(version))
        required = {"user.rpc", "debug.rpc", "repl.stdout"}
        if not required.issubset(set(capabilities.get("channels", []))):
            raise RuntimeError("required EV-MUX channels are unavailable")

        state = mux.rpc("debug.rpc", "transport.state", {})
        print("routes: user={user[route]} debug={debug[route]}".format(**state))

        mux.rpc("user.rpc", "device.control", {"action": "stop"})
        mux.wait_for_text(b">>>", timeout=10)

    path = "/evmux_smoke.py"

```

(续下页)

(接上页)

```
upload_script(mux, path, DEVICE_SCRIPT)
mux.rpc("user.rpc", "script.run", {"path": path})
mux.wait_for_text(b"EVTEST:OK", timeout=20)
print("\nPASS: EV-MUX upload, execution, camera capture, and stdout_
↪verified")

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

该测试会将超时、帧格式错误、路由拒绝、RPC 失败响应、未收到摄像头标记或协议版本不支持都视为失败。在产品集成中应保留相同的帧处理和序列关联规则，等待 RPC 时继续读取无关事件，显式处理 VM_BUSY 和 VM_TIMEOUT，并在观测到预期设备端输出或结果之前绝不报告成功。

Chapter 7

方案架构

ESP-VISION 围绕 MicroPython 固件构建、板级硬件后端、共享平台服务以及面向 Python 的视觉模块进行组织。代码按是否触及 MicroPython (`mp_obj_t / py/* .h`) 进行分层。

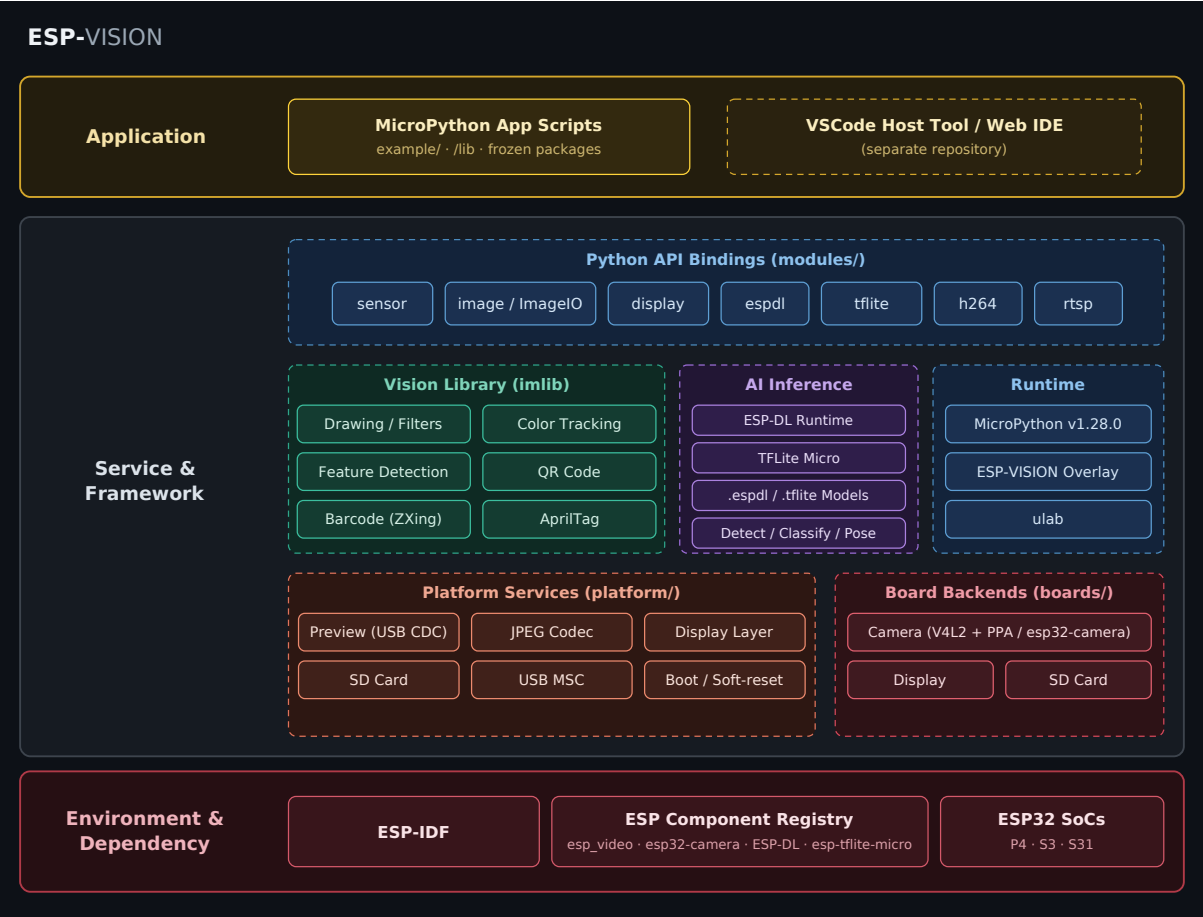
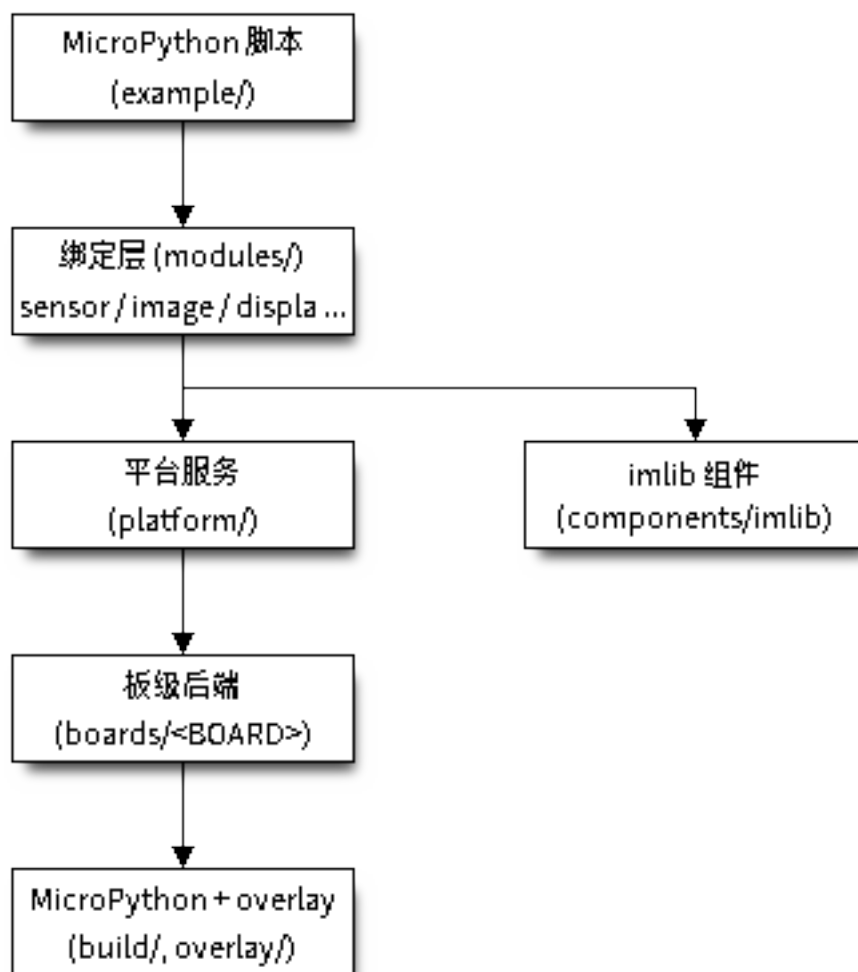


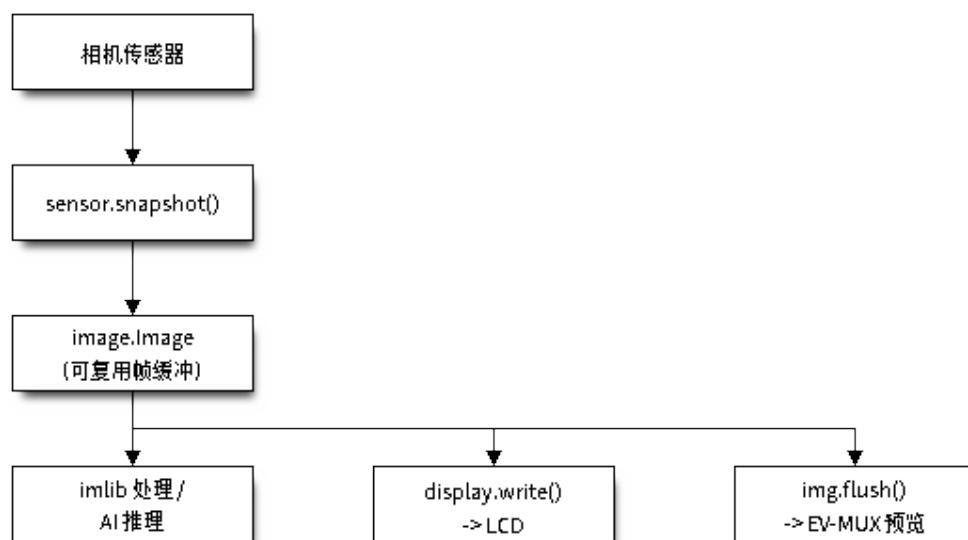
图 1: ESP-VISION 分层架构总览

7.1 分层概览



- **绑定层 (modules/)**: 即 USER_C_MODULES 层。主要模块 image、sensor、display、espd1 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 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。
- **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。

7.2 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。

7.3 底层传输机制

ESP-VISION 的主机传输分为三层：物理 sink、路由 stream、EV-MUX 逻辑 channel。

7.3.1 物理 sink

`platform/ev_channel.c` 定义可写入的物理目标：

sink	含义
usj	USB-Serial-JTAG。在提供该外设的板卡上可用；没有 host 占用 USB-OTG CDC 时优先作为回退 sink。
cdc	USB-OTG CDC。在可用时，host 打开端口（DTR）即成为所有 stream 的自动路由。同时提供 MSC 文件预览。
uart	用于运行时重定向（ <code>route.bind / route.auto</code> ）的有线辅助 sink；永远不会被自动选中。stream 绑定到 UART 后，也从 UART 接收其 framed 输入。
console	MicroPython stdout fallback。
null	丢弃输出；none 是它的别名。

cdc 上报两种状态：`present` 表示 USB-OTG 接口已枚举；`ready` 表示有 host 占用端口（DTR 有效）。只有 `ready` 决定路由——设计中不存在激活 RPC、租约或心跳。

7.3.2 路由 stream

`ev_stream_t` 是物理路由面的抽象，数量少于 EV-MUX channel。路由操作的单位是完整 stream，而不是单个逻辑 channel：一个 stream 绑定到某个 sink 后，归属于该 stream 的所有 channel 都使用该 sink。

Device -> Host 的帧写入该 sink；Host -> Device 的对应 channel 帧只接受从该 sink 进入。

所有 stream 的自动路由由同一条规则决定：**host 占用 USB-OTG 端口（DTR）期间活跃 sink 为“cdc”，否则使用板级首选回退（有 USJ 时为 USJ，再回退到 console）**。专用 transport 任务监视 DTR 边沿并把它应用到全部两个 stream；每次切换都按 stream 发出 route.changed (reason 为 cdc_connected / cdc_disconnected)。host 断开或崩溃会释放 DTR，因此回退是自动的——无需任何保活。

stream	承载内容	路由策略
EV_STREAM_USER	正常工作面：user.rpc、repl.stdin、repl.signal、repl.stdout、repl.stderr、preview.frame	跟随活跃 sink（上述 DTR 规则）。
EV_STREAM_DEBUG	调试与系统流量：debug.rpc/log.idf	跟随活跃 sink，因此已连接的 USB 端口承载所有逻辑 channel。

route.bind 把单个 stream 固定到某个 sink（运行时重定向，例如把 debug 重定向到 uart 做有线日志采集）；手动绑定的 stream 脱离 DTR 规则，直到 route.auto 恢复自动路由。

7.3.3 EV-MUX channel

EV-MUX channel 是端到端协议语义面，不等同于物理端口。每个 channel 固定归属于一个 stream，host 不得为单个 channel 单独选择 sink。direction 描述帧方向；Device -> Host 按所属 stream 的当前 sink 输出，Host -> Device 则按所属 stream 的当前 sink 做入站授权。

channel	direction	所属 stream	payload 类型	用途
user.rpc	双向	user (rsp / event)	JSON	正常工作控制，例如 hello、capabilities、script.write、script.run、device.control。
debug.rpc	双向	debug (rsp / event)	JSON；debug.capture_frame 的应答帧自身直接携带二进制 JPEG payload	系统/调试 EV-ATP 请求、响应、事件、错误；req 为 Host -> Device，rsp / event 为 Device -> Host。
repl.stdin	Host -> Device	user	text	Host 输入到 REPL，不是输出 stream。
repl.signal	Host -> Device	user	small binary/text	Host 发送 Ctrl-C 等信号。
repl.stdout	Device -> Host	user	text	Python print()、REPL prompt、C stdout。
repl.stderr	Device -> Host	user	text	Python exception、C stderr。
log.idf	Device -> Host	debug	text	ESP-IDF ESP_LOGx 输出。
preview.frame	Device -> Host	user	binary JPEG	img.flush() 持续产生的预览帧；无需订阅、不经 RPC。

7.3.4 EV-MUX 帧格式

所有 EV-MUX 帧都使用长度前缀，payload 可以是二进制：

```
\x1eEVMUX/1 h=<metadata_len> p=<payload_len> c=<crc32>\r\n
<metadata JSON bytes>
<payload bytes>
\x1f
```

`h` 与 `p` 是权威长度；`0x1f` 只作为帧尾保护与重同步辅助。固件生成的帧带有 payload CRC32。Host -> Device 的命令帧可以使用 `c=00000000`，固件接收端把 0 视为“跳过 CRC 校验”。

7.3.5 EV-ATP 控制

EV-ATP RPC 按语义拆分：

- `user.rpc` 承载 IDE 正常工作控制：`hello`、`capabilities`、`script.write`、`script.run`、`device.control`。
- `debug.rpc` 承载系统/调试命令：`transport.state`、`route.get`、`route.bind`、`route.auto`、`debug.info`、`debug.capture_frame`。应答回到请求到达的 sink。`debug.capture_frame` 的图片由 `debug.rpc` 应答帧自身携带（`metadata` 带 `contentType=image/jpeg` 与 `width/height`），不存在独立的数据 channel。

`route.changed` 是 `user.rpc/user` 上的路由事件，不是 `debug.rpc` 事件。它报告 stream 级别的 sink 变化（`reason` 为 `cdc_connected/cdc_disconnected`），host 收到后必须更新该 stream 下所有 channel 的连接状态。

路由本身不需要任何 RPC：打开 USB-OTG 端口即成为活跃 sink，关闭则选择板级回退。设计中刻意不提供激活、租约或心跳方法——线路状态（DTR）就是完整的路由协议。

7.3.6 Host 对接契约

上位机同一时刻只使用一条物理 USB 连接；在双 USB 板卡上，USB-OTG 与 USJ 是等价单选项，任一端口都承载全部 channel。连接顺序如下：

1. 打开所选端口。EV-MUX 上电默认启用，无需任何 REPL 访问：USB 链路出现时设备会发出 `bootstrap hello` 事件（每次软复位后也会重发）。
2. 发送 framed `hello` 请求（或消费启动事件），再请求 `capabilities`；确认 `firmware.id` 为 `esp-vision`，检查 `evMuxVersion`，并协商固件声明的 `feature/channel`。
3. 正常使用设备：REPL channel、`script.write/script.run`、预览、`debug.rpc`、`log.idf` 全部在已打开的端口上可用。不存在激活步骤，也不存在心跳；端口即路由。
4. 断开时直接关闭端口即可。USB-OTG 端口关闭（或线缆脱落）后，所有 stream 自动选择板级回退，并发出 `route.changed` 事件上报。

`sensor.evmutex()` 保留为调试开关（例如用 `sensor.evmutex(False)` 恢复纯文本 REPL）；板级如需关闭上电默认启用，可在 `boardconfig.h` 把 `ESP_VISION_EV_MUX_DEFAULT_ENABLED` 定义为 0。

`hello`、`capabilities` 和 `debug.info` 的 device payload 都包含相同的 `firmware` 对象。其中稳定的 `id` 为 `esp-vision`；`version` 使用 ESP-IDF 根据 Git 自动生成的 `PROJECT_VER`：正式发布构建返回 `YYYY.MM.DD tag`，未打 tag 的开发构建带 `-N-g<commit>` 后缀（工作区构建还可能带 `-dirty`）。Host 可以用这个构建身份做版本展示和升级提示，但协议兼容性必须根据 `evMuxVersion` 与声明的 `capabilities` 判断，不能只比较发布版本字符串。

Host 的 `demux` 必须逐帧读取 `metadata.channel`，再按 channel 分发，不能依据端口或先前帧推断当前帧类型。推荐维护静态 `channel -> stream` 表和动态 `stream -> sink` 表；路由变化只更新后者。

7.3.7 接收分流原则

固件接收侧只保留字节流拼帧所需的状态。每个完整 EV-MUX 帧到达后，先根据实际入站 sink 校验其所属 stream，再立即按 `metadata.channel/type/method` 分流：

- `repl.stdin` 写入 REPL 输入环形缓冲。
- `repl.signal` 调度键盘中断等信号。
- `user.rpc/debug.rpc` 先选择 RPC domain，再按 method 执行 EV-ATP 控制逻辑。

入站授权是单一白名单：发现方法 `hello/capabilities` 在任一 USB sink 上都受理；其余 channel 只从所属 stream 的当前路由受理。显式 `route.bind` 到 UART 后，也可从 UART 接收 framed 控制。业务层不得依赖“当前模式”推断帧类型。

7.3.8 传输执行模型

EV-MUX 上电默认启用 (ESP_VISION_EV_MUX_DEFAULT_ENABLED)，控制面无需 REPL 访问即可探测；每次软复位后 mux 状态与路由都回到同一组确定初值。

帧接收不依赖 REPL。专用 transport 任务 (ev_transport, 见 platform/ev_control_transport.c) 拥有接收路径：泵送现有 TinyUSB CDC 设备栈、把 USJ/CDC/UART 三个入站环形缓冲喂给各 ingress 的帧 parser、分发完整帧，并应用 DTR 路由规则（活跃 sink 切换与 route.changed 发出）。mp_hal_stdin_rx_chr() 不再解析帧；EV-MUX 开启时它只消费 framed repl.stdin 字节。

分发按执行上下文拆分：

- transport 任务 RPC（即使用户代码运行中也立即应答）：hello、capabilities、transport.state、route.*、device.control。repl.signal 由 transport 任务调度键盘中断，host Ctrl-C 可以到达正在运行的脚本。
- VM 任务 RPC（排队到解释器，在 mp_hal_stdin_rx_chr() 上下文执行）：script.write、script.run、debug.info、debug.capture_frame——所有涉及 MicroPython 对象、VFS 或 camera 的方法。队列满时回复 VM_BUSY 错误。
- repl.stdin payload 写入 framed REPL 输入环形缓冲，由 REPL 循环消费。

EV-MUX 开启后，其 USJ/CDC/UART 底层接收路径只解析帧，不再把 0x03 解释为 Ctrl-C（中断语义只属于 framed repl.signal channel），不成帧的字节由帧 parser 的 SOF 重同步丢弃。IDF system console 在协议初始化前或致命故障路径上输出的原始字节属于下述例外。

7.3.9 IDF console 与接口选择

ESP-IDF system console 与 ESP-VISION 接口选择是两层独立配置。板级 sdkconfig 不覆盖 CONFIG_ESP_CONSOLE_* 或 CONFIG_USJ_ENABLE_USB_SERIAL_JTAG，console 路由、bootloader 输出与系统调试能力均采用当前 ESP-IDF target/version 的默认值。某些 target 的默认 secondary console 可能使用 USB-Serial-JTAG；这属于 IDF system console 行为，不作为 ESP-VISION 是否启用 USJ 的开关。

MicroPython USJ 接口只由板级 mpconfigboard.h 的 MICROPY_HW_ESP_USB_SERIAL_JTAG 控制：P4 板卡显式设为 1，S3/S31 板卡设为 0。启用时，usb_serial_jtag_init 会按照 ESP-IDF USJ driver 的初始化顺序显式开启 USJ bus clock 并选择/启用 internal PHY，因此不依赖 console Kconfig 是否顺带保持外设开启；禁用时不编译也不初始化 ESP-VISION 的 USJ 路径，但 IDF 仍可按其默认 console 配置使用该外设。

ESP32-S3 板卡另由 mpconfigboard.h 显式开启 MICROPY_HW_USB_CDC_DTR_RTS_BOOTLOADER，并把 OTG CDC 设备 PID 设为 0x1001。现有 MicroPython/EV-MUX TinyUSB CDC 栈会在 VM 初始化后立即启动，早于 camera 初始化、文件系统恢复和 boot.py；用户代码运行时仍由 EV-MUX transport task 独立服务同一个栈，并且该任务每轮至少延迟一个 FreeRTOS tick。不安装第二套 USB driver 或 descriptor owner，CDC/MSC 仍由 mpconfigboard.h 选择，IDF system console 配置不变。PID 0x1001 使 esptool 选择 USB-Serial/JTAG 复位序列；固件仅在软件中实现相同的 DTR/RTS 状态表，OTG 设备本身不提供 JTAG 接口。TinyUSB 回调只记录 DTR/RTS 请求并启动延迟 timer，由 timer task 在 control-request callback 之外执行复位。在 S3 上，该任务独立遵循 MicroPython 的 ROM 契约 (usb_usj_mode、usb_dc_prepare_persist 和 USBDC_BOOT_DFU)，随后由 shutdown handler 设置 RTC_CNTL_FORCE_DOWNLOAD_BOOT 并调用 esp_restart；不会等待 host bus reset，也不会再在回调中拆除 TinyUSB controller。因此 host 会正常看到重新枚举并需要重新打开 ROM CDC 节点；Linux 上 esptool 5.3 可能因首个句柄已经失效而报告 EIO，即使 ROM loader 已经就绪，此时可在新节点上使用 --before no-reset 重试。其他板卡默认关闭该能力。

正常运行时，ev_stdio_init0 通过 esp_log_set_vprintf 与 newlib stdout/stderr hook 把运行期输出封装为 EV-MUX 帧。IDF 在 hook 安装前发送的 boot 日志，以及绕过 hook 的致命故障输出，仍可能按 system console 默认路由以原始字节出现在 UART 或 USJ；这些字节不属于活动期 EV-MUX 契约，host 会丢弃它们并在后续 SOF 重同步。

若 ROM loader 是手动进入的，或刷写后仍停留在 ROM，请在 esptool 结束阶段使用 --after watchdog-reset：USB-Serial-JTAG 的默认 core reset 不会重新采样 GPIO0，而 watchdog 路径会执行完整系统复位并回到 SPI 启动。

7.3.10 传输可靠性与观测

- 每个帧段都使用带进度检测和 1500ms 总预算的 write-all loop (platform/ev_mux.c)。底层 sink 返回实际写入字节数；无法继续前进或超时会中止本次发送、增加 txTimeout 并把 sink 标记为拥塞。失败前已经写出的字节无法撤回，因此 host parser 仍须丢弃截断数据并在后续有效 SOF 重同步；这里保证的是失败可见且有界，而不是让一次 partial write 变成原子写入。
- preview.frame 是唯一允许主动丢弃的类别。sink 拥塞后的 500ms 冷却期内，后续 preview 会在编码或写入前整帧丢弃并增加 txDrop；冷却结束后的第一帧用于探测恢复。USJ 路由还限制为每 100ms 至多一帧。RPC 与 REPL 不会被拥塞预丢弃，但真实 sink 写失败仍会使当前发送返回错误。
- USJ、CDC、UART 三个 ingress 各有一个 2048 字节物理输入环形缓冲；它们是 parser 持续排空的暂存区，不代表协议帧必须小于该缓冲。debug.info 的 transport.stats scope 是纯 C 路径，VM 忙时仍可返回 RX 字节数、ring 满事件、完整/畸形/拒绝帧数，以及 replRx、vmBusy、txTimeout、txDrop。其中 replRx 是独立的 512 字节 REPL 队列；溢出目前只计数，没有额外的异步 host 事件。

7.4 源码结构

路径	职责
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。

7.5 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](#)。

7.6 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](#)。

7.7 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](#)。

Chapter 8

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](#)。

8.1 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。

8.2 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 9

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。

9.1 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
MicroPython v1.28.0	lib/micropython	Python 运行时与 ESP32 port 基线	MIT
micropython-ulab 6.12	lib/ulab	数组与数值计算	MIT
OpenMV v5.0.0 imlib 子集	components/imlib	图像处理与绘制算法	MIT，部分文件单独授权
OpenMV Python 绑定代码	modules/py_image*、modules/py_helper*	OpenMV 风格的 image API 绑定	MIT
OpenMV AprilTag 实现	components/imlib/upstream/apriltag.c	AprilTag 与矩形检测	BSD-2-Clause
ZXing-C++ v3.0.2	lib/zxing-cpp	一维条码识别后端	Apache-2.0
ESP-DL	Component Registry	模型推理运行时	MIT
TensorFlow Lite Micro / esp-tflite-micro	Component Registry	TensorFlow Lite Micro 推理运行时	Apache-2.0
esp_new_jpeg	Component Registry	软件 JPEG 编解码	Espressif MIT（限乐鑫产品）
esp32-camera	Component Registry	相机驱动	Apache-2.0
ESP-IDF	外部 SDK	构建系统、驱动和媒体组件	Apache-2.0

9.2 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。

9.3 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。

9.4 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。

Python 模块索引

d

display, [58](#)

e

espd1, [61](#)

h

h264, [69](#)

i

image, [42](#)

imageio, [67](#)

r

rtsp, [70](#)

s

sensor, [39](#)

t

tflite, [65](#)

索引

A

`a_bins()` (`image.histogram` 方法), 50
`a_lq()` (`image.statistics` 方法), 52
`a_max()` (`image.statistics` 方法), 52
`a_mean()` (`image.statistics` 方法), 51
`a_median()` (`image.statistics` 方法), 51
`a_min()` (`image.statistics` 方法), 52
`a_mode()` (`image.statistics` 方法), 51
`a_stdev()` (`image.statistics` 方法), 51
`a_uq()` (`image.statistics` 方法), 52
`a_value()` (`image.percentile` 方法), 50
`a_value()` (`image.threshold` 方法), 51
`APPLY_COLOR_PALETTE_FIRST()` (在 `image` 模块中), 43
`apriltag` (`image` 中的类), 50
`area()` (`image.blob` 方法), 48
`AREA()` (在 `image` 模块中), 43

B

`b_bins()` (`image.histogram` 方法), 50
`b_lq()` (`image.statistics` 方法), 52
`b_max()` (`image.statistics` 方法), 52
`b_mean()` (`image.statistics` 方法), 52
`b_median()` (`image.statistics` 方法), 52
`b_min()` (`image.statistics` 方法), 52
`b_mode()` (`image.statistics` 方法), 52
`b_stdev()` (`image.statistics` 方法), 52
`b_uq()` (`image.statistics` 方法), 52
`b_value()` (`image.percentile` 方法), 51
`b_value()` (`image.threshold` 方法), 51
`backlight()` (`display.ESP32Display` 方法), 59
`barcode` (`image` 中的类), 49
`BAYER()` (在 `image` 模块中), 42
`BICUBIC()` (在 `image` 模块中), 43
`bilateral()` (`image.Image` 方法), 56
`BILINEAR()` (在 `image` 模块中), 43
`binary()` (`image.Image` 方法), 54
`BINARY()` (在 `image` 模块中), 42
`binary_to_grayscale()` (在 `image` 模块中), 45
`binary_to_lab()` (在 `image` 模块中), 45
`binary_to_rgb()` (在 `image` 模块中), 45
`binary_to_yuv()` (在 `image` 模块中), 45
`bins()` (`image.histogram` 方法), 50
`BLACK_BACKGROUND()` (在 `image` 模块中), 43
`blob` (`image` 中的类), 47

`blur()` (`image.Image` 方法), 56
`buffer_size()` (`imageio.ImageIO` 方法), 67
`bytearray()` (`image.Image` 方法), 52

C

`CENTER()` (在 `image` 模块中), 43
`circle()` (`image.circle` 方法), 46
`circle` (`image` 中的类), 46
`classify()` (`espd1.ImageNetCls` 方法), 64
`clear()` (`display.ESP32Display` 方法), 59
`clear()` (`image.Image` 方法), 53
`close()` (`h264.H264Encoder` 方法), 69
`close()` (`image.Image` 方法), 55
`close()` (`imageio.ImageIO` 方法), 68
`CODABAR()` (在 `image` 模块中), 44
`code()` (`image.blob` 方法), 48
`CODE128()` (在 `image` 模块中), 44
`CODE39()` (在 `image` 模块中), 44
`CODE93()` (在 `image` 模块中), 44
`compactness()` (`image.blob` 方法), 48
`compress()` (`image.Image` 方法), 53
`convexity()` (`image.blob` 方法), 48
`copy()` (`image.Image` 方法), 53
`CORNER_AGAST()` (在 `image` 模块中), 44
`CORNER_FAST()` (在 `image` 模块中), 44
`corners()` (`image.barcode` 方法), 49
`corners()` (`image.blob` 方法), 47
`corners()` (`image.qrcode` 方法), 48
`corners()` (`image.rect` 方法), 47
`count()` (`image.blob` 方法), 48
`count()` (`imageio.ImageIO` 方法), 67
`crop()` (`image.Image` 方法), 53
`cx()` (`image.blob` 方法), 47
`cxr()` (`image.blob` 方法), 47
`cy()` (`image.blob` 方法), 47
`cyf()` (`image.blob` 方法), 47

D

`data_type()` (`image.qrcode` 方法), 49
`DATABAR()` (在 `image` 模块中), 44
`DATABAR_EXP()` (在 `image` 模块中), 44
`deinit()` (`display.ESP32Display` 方法), 59
`deinit()` (`espd1.ESPDet` 方法), 62
`deinit()` (`espd1.ImageNetCls` 方法), 64
`deinit()` (`espd1.Model` 方法), 62

deinit() (espd1.PedestrianDetect 方法), 63
 deinit() (espd1.YOLO11 方法), 63
 deinit() (espd1.YOLO11nPose 方法), 64
 deinit() (tflite.Model 方法), 65
 density() (image.blob 方法), 48
 detect() (espd1.ESPDet 方法), 62
 detect() (espd1.PedestrianDetect 方法), 63
 detect() (espd1.YOLO11 方法), 63
 detect() (espd1.YOLO11nPose 方法), 64
 difference() (image.Image 方法), 55
 dilate() (image.Image 方法), 55
 display
 module, 58
 draw_arrow() (image.Image 方法), 54
 draw_circle() (image.Image 方法), 54
 draw_cross() (image.Image 方法), 54
 draw_edges() (image.Image 方法), 54
 draw_ellipse() (image.Image 方法), 54
 draw_image() (image.Image 方法), 54
 draw_keypoints() (image.Image 方法), 54
 draw_line() (image.Image 方法), 54
 draw_rectangle() (image.Image 方法), 54
 draw_string() (image.Image 方法), 54

E

EAN13() (在 image 模块中), 44
 EAN2() (在 image 模块中), 44
 EAN5() (在 image 模块中), 44
 EAN8() (在 image 模块中), 44
 ecc_level() (image.qrcode 方法), 49
 eci() (image.qrcode 方法), 49
 EDGE_CANNY() (在 image 模块中), 44
 EDGE_SIMPLE() (在 image 模块中), 44
 elongation() (image.blob 方法), 48
 enclosed_ellipse() (image.blob 方法), 48
 enclosing_circle() (image.blob 方法), 48
 encode() (h264.H264Encoder 方法), 69
 erode() (image.Image 方法), 55
 ESP32Display (display 中的类), 59
 ESPDet (espd1 中的类), 62
 espd1
 module, 61
 extent() (image.blob 方法), 48
 EXTRACT_RGB_CHANNEL_FIRST() (在 image 模块中), 43

F

FILE_STREAM() (在 imageio 模块中), 67
 find_apriltags() (image.Image 方法), 57
 find_barcode() (image.Image 方法), 57
 find_blobs() (image.Image 方法), 57
 find_circles() (image.Image 方法), 57
 find_line_segments() (image.Image 方法), 57

find_lines() (image.Image 方法), 57
 find_qrcodes() (image.Image 方法), 57
 find_rects() (image.Image 方法), 57
 flush() (image.Image 方法), 52
 format() (image.Image 方法), 52

G

gaussian() (image.Image 方法), 56
 gaussian_blur() (image.Image 方法), 56
 get_framesize() (在 sensor 模块中), 39
 get_hist() (image.Image 方法), 56
 get_histogram() (image.Image 方法), 56
 get_hmirror() (在 sensor 模块中), 40
 get_id() (在 sensor 模块中), 39
 get_ipa() (在 sensor 模块中), 40
 get_isp() (在 sensor 模块中), 40
 get_percentile() (image.histogram 方法), 50
 get_pixel() (image.Image 方法), 52
 get_pixformat() (在 sensor 模块中), 39
 get_regression() (image.Image 方法), 56
 get_statistics() (image.histogram 方法), 50
 get_statistics() (image.Image 方法), 56
 get_stats() (image.histogram 方法), 50
 get_stats() (image.Image 方法), 56
 get_threshold() (image.histogram 方法), 50
 get_vflip() (在 sensor 模块中), 40
 GRAYSCALE() (在 image 模块中), 42
 GRAYSCALE() (在 sensor 模块中), 39
 grayscale_to_binary() (在 image 模块中), 45
 grayscale_to_lab() (在 image 模块中), 45
 grayscale_to_rgb() (在 image 模块中), 45
 grayscale_to_yuv() (在 image 模块中), 45

H

h() (image.barcode 方法), 50
 h() (image.blob 方法), 47
 h() (image.qrcode 方法), 49
 h() (image.rect 方法), 47
 h264
 module, 69
 H264Encoder (h264 中的类), 69
 height() (display.ESP32Display 方法), 59
 height() (image.Image 方法), 52
 height() (在 sensor 模块中), 39
 histeq() (image.Image 方法), 55
 histogram() (image.Image 方法), 56
 histogram (image 中的类), 50
 HMIRROR() (在 image 模块中), 43

I

I25() (在 image 模块中), 44
 image
 module, 42

- imageio
 - module, 67
- ImageIO (imageio 中的类), 67
- ImageNetCls (espd1 中的类), 64
- Image (image 中的类), 52
- inputs() (espd1.Model 方法), 62
- invert() (image.Image 方法), 54
- is_alphanumeric() (image.qrcode 方法), 49
- is_binary() (image.qrcode 方法), 49
- is_closed() (imageio.ImageIO 方法), 67
- is_kanji() (image.qrcode 方法), 49
- is_numeric() (image.qrcode 方法), 49
- ISBN10() (在 image 模块中), 44
- ISBN13() (在 image 模块中), 44
- J**
- JPEG() (在 image 模块中), 42
- JPEG_SUBSAMPLING_420() (在 image 模块中), 44
- JPEG_SUBSAMPLING_422() (在 image 模块中), 44
- JPEG_SUBSAMPLING_444() (在 image 模块中), 44
- JPEG_SUBSAMPLING_AUTO() (在 image 模块中), 44
- K**
- keyframe() (h264.H264Encoder 方法), 69
- L**
- l_bins() (image.histogram 方法), 50
- l_lq() (image.statistics 方法), 51
- l_max() (image.statistics 方法), 51
- l_mean() (image.statistics 方法), 51
- l_median() (image.statistics 方法), 51
- l_min() (image.statistics 方法), 51
- l_mode() (image.statistics 方法), 51
- l_stdev() (image.statistics 方法), 51
- l_uq() (image.statistics 方法), 51
- l_value() (image.percentile 方法), 50
- l_value() (image.threshold 方法), 51
- lab_to_binary() (在 image 模块中), 45
- lab_to_grayscale() (在 image 模块中), 45
- lab_to_rgb() (在 image 模块中), 45
- lab_to_yuv() (在 image 模块中), 45
- laplacian() (image.Image 方法), 56
- length() (image.line 方法), 46
- line() (image.line 方法), 46
- line (image 中的类), 46
- load_model() (在 espd1 模块中), 61
- lq() (image.statistics 方法), 51
- M**
- magnitude() (image.circle 方法), 46
- magnitude() (image.line 方法), 46
- magnitude() (image.rect 方法), 47
- major_axis_line() (image.blob 方法), 48
- mask() (image.qrcode 方法), 49
- max() (image.statistics 方法), 51
- mean() (image.Image 方法), 55
- mean() (image.statistics 方法), 51
- median() (image.Image 方法), 55
- median() (image.statistics 方法), 51
- MEMORY_STREAM() (在 imageio 模块中), 67
- midpoint() (image.Image 方法), 55
- min() (image.statistics 方法), 51
- min_corners() (image.blob 方法), 47
- minor_axis_line() (image.blob 方法), 48
- mode() (image.Image 方法), 55
- mode() (image.statistics 方法), 51
- Model (espd1 中的类), 62
- Model (tflite 中的类), 65
- module
 - display, 58
 - espd1, 61
 - h264, 69
 - image, 42
 - imageio, 67
 - rtsp, 70
 - sensor, 39
 - tflite, 65
- morph() (image.Image 方法), 55
- O**
- offset() (imageio.ImageIO 方法), 67
- open() (image.Image 方法), 55
- outputs() (espd1.Model 方法), 62
- P**
- PALETTE_DEPTH() (在 image 模块中), 43
- PALETTE_EVT_DARK() (在 image 模块中), 43
- PALETTE_EVT_LIGHT() (在 image 模块中), 43
- PALETTE_IRONBOW() (在 image 模块中), 43
- PALETTE_RAINBOW() (在 image 模块中), 43
- payload() (image.barcode 方法), 50
- payload() (image.qrcode 方法), 49
- PDF417() (在 image 模块中), 44
- PedestrianDetect (espd1 中的类), 63
- percentile (image 中的类), 50
- perimeter() (image.blob 方法), 48
- pixels() (image.blob 方法), 47
- PNG() (在 image 模块中), 42
- predict() (espd1.Model 方法), 62
- predict() (tflite.Model 方法), 65
- Q**
- QQVGA() (在 sensor 模块中), 39
- qrcode (image 中的类), 48
- quality() (image.barcode 方法), 50
- QVGA() (在 sensor 模块中), 39
- R**
- r() (image.circle 方法), 46
- read() (imageio.ImageIO 方法), 67

rect() (image.barcode 方法), 49
 rect() (image.blob 方法), 47
 rect() (image.qrcode 方法), 49
 rect() (image.rect 方法), 47
 rect (image 中的类), 46
 reset() (在 sensor 模块中), 39
 RGB565() (在 image 模块中), 42
 RGB565() (在 sensor 模块中), 39
 rgb_to_binary() (在 image 模块中), 45
 rgb_to_grayscale() (在 image 模块中), 45
 rgb_to_lab() (在 image 模块中), 45
 rgb_to_yuv() (在 image 模块中), 45
 rho() (image.line 方法), 46
 ROTATE_180() (在 image 模块中), 43
 ROTATE_270() (在 image 模块中), 43
 ROTATE_90() (在 image 模块中), 43
 rotation() (image.barcode 方法), 50
 rotation() (image.blob 方法), 47
 rotation_deg() (image.blob 方法), 48
 rotation_rad() (image.blob 方法), 48
 roundness() (image.blob 方法), 48
 rtsp
 module, 70
 RTSPServer (rtsp 中的类), 71

S

save() (image.Image 方法), 53
 scale() (image.Image 方法), 53
 SCALE_ASPECT_EXPAND() (在 image 模块中), 43
 SCALE_ASPECT_IGNORE() (在 image 模块中), 43
 SCALE_ASPECT_KEEP() (在 image 模块中), 43
 SEARCH_DS() (在 image 模块中), 44
 SEARCH_EX() (在 image 模块中), 44
 seek() (imageio.ImageIO 方法), 68
 send() (rtsp.RTSPServer 方法), 71
 sensor
 module, 39
 SensorStatus (sensor 中的类), 40
 set_framesize() (在 sensor 模块中), 39
 set_hmirror() (在 sensor 模块中), 39
 set_ipa() (在 sensor 模块中), 40
 set_isp() (在 sensor 模块中), 40
 set_pixel() (image.Image 方法), 52
 set_pixformat() (在 sensor 模块中), 39
 set_thresholds() (espd1.ESPDet 方法), 62
 set_thresholds() (espd1.ImageNetCls 方法), 64
 set_thresholds() (espd1.PedestrianDetect 方法), 63
 set_thresholds() (espd1.YOLO11 方法), 63
 set_thresholds() (espd1.YOLO11nPose 方法), 64
 set_vflip() (在 sensor 模块中), 40

shutdown() (在 sensor 模块中), 39
 size() (image.Image 方法), 52
 size() (imageio.ImageIO 方法), 67
 skip_frames() (在 sensor 模块中), 40
 snapshot() (在 sensor 模块中), 40
 solidity() (image.blob 方法), 48
 statistics() (image.histogram 方法), 50
 statistics() (image.Image 方法), 56
 statistics (image 中的类), 51
 status() (在 sensor 模块中), 40
 stdev() (image.statistics 方法), 51
 stop() (rtsp.RTSPServer 方法), 71
 sync() (imageio.ImageIO 方法), 68

T

TAG16H5() (在 image 模块中), 45
 TAG25H7() (在 image 模块中), 45
 TAG25H9() (在 image 模块中), 45
 TAG36H10() (在 image 模块中), 45
 TAG36H11() (在 image 模块中), 45
 tflite
 module, 65
 theta() (image.line 方法), 46
 threshold (image 中的类), 51
 to_bitmap() (image.Image 方法), 53
 to_grayscale() (image.Image 方法), 53
 to_ironbow() (image.Image 方法), 53
 to_jpeg() (image.Image 方法), 53
 to_png() (image.Image 方法), 53
 to_rainbow() (image.Image 方法), 53
 to_rgb565() (image.Image 方法), 53
 TRANSPOSE() (在 image 模块中), 43
 type() (image.barcode 方法), 50
 type() (imageio.ImageIO 方法), 67

U

UPCA() (在 image 模块中), 44
 UPCE() (在 image 模块中), 44
 uq() (image.statistics 方法), 51

V

value() (image.percentile 方法), 50
 value() (image.threshold 方法), 51
 version() (image.qrcode 方法), 49
 version() (imageio.ImageIO 方法), 67
 VFLIP() (在 image 模块中), 43
 VGA() (在 sensor 模块中), 39

W

w() (image.barcode 方法), 49
 w() (image.blob 方法), 47
 w() (image.qrcode 方法), 49
 w() (image.rect 方法), 47
 width() (display.ESP32Display 方法), 59
 width() (image.Image 方法), 52
 width() (在 sensor 模块中), 39
 write() (display.ESP32Display 方法), 59
 write() (imageio.ImageIO 方法), 67

X

`x()` (`image.barcode` 方法), [49](#)
`x()` (`image.blob` 方法), [47](#)
`x()` (`image.circle` 方法), [46](#)
`x()` (`image.qrcode` 方法), [49](#)
`x()` (`image.rect` 方法), [47](#)
`x1()` (`image.line` 方法), [46](#)
`x2()` (`image.line` 方法), [46](#)
`x_hist_bins()` (`image.blob` 方法), [48](#)

Y

`y()` (`image.barcode` 方法), [49](#)
`y()` (`image.blob` 方法), [47](#)
`y()` (`image.circle` 方法), [46](#)
`y()` (`image.qrcode` 方法), [49](#)
`y()` (`image.rect` 方法), [47](#)
`y1()` (`image.line` 方法), [46](#)
`y2()` (`image.line` 方法), [46](#)
`y_hist_bins()` (`image.blob` 方法), [48](#)
`YOLO11nPose` (`espd1` 中的类), [63](#)
`YOLO11` (`espd1` 中的类), [63](#)
`YUV422()` (在 `image` 模块中), [42](#)
`yuv_to_binary()` (在 `image` 模块中), [45](#)
`yuv_to_grayscale()` (在 `image` 模块中), [46](#)
`yuv_to_lab()` (在 `image` 模块中), [46](#)
`yuv_to_rgb()` (在 `image` 模块中), [46](#)