OCRBit：用 Python、浏览器与多模态模型重做截图 OCR

从全局热键、多显示器截图、浏览器 Kiosk 框选，到 SSE 流式 OCR、科学公式提示词与 DPAPI 密钥保护，拆解 OCRBit 的轻量 Windows 架构。

2026年7月27日 · OCR · Python · React · Windows · 多模态模型

OCRBit 是一个面向 Windows 的截图 OCR 工具。它的功能看起来和常见截图翻译软件相似：按下快捷键、框选屏幕区域、得到可复制的文字，再对结果进行翻译、总结或解释。但它没有使用 Electron，也没有维护一个常驻的桌面 GUI，而是把系统能力、业务逻辑和界面拆成了三层：

```
Win32 / Python
  ├─ 注册全局快捷键
  ├─ 捕获鼠标所在显示器
  ├─ 裁切截图并管理临时数据
  └─ 托管本地 HTTP 服务
              |
              ▼
React / 系统浏览器
  ├─ 在 Kiosk 窗口中完成框选
  ├─ 渲染 Markdown、表格与 KaTeX
  └─ 消费 SSE 流式结果
              |
              ▼
SiliconFlow API
  └─ Qwen3.5-397B-A17B 多模态识别与文本处理
```

这套设计的重点不是单纯“减小安装包”，而是复用 Windows 和浏览器已经具备的能力：Python 负责系统调用和密钥，浏览器负责交互与排版，远端多模态模型负责理解图像。本文基于 OCRBit `v1.0.0` 的首个公开提交分析其实现。

从快捷键到截图：Win32 消息循环

程序入口先检查运行平台，然后尝试调用：

```
ctypes.windll.user32.SetProcessDpiAwarenessContext(ctypes.c_void_p(-4))
```

`-4` 对应 `DPI_AWARENESS_CONTEXT_PER_MONITOR_AWARE_V2`。这一步很重要：在使用不同缩放比例的多显示器环境里，如果进程仍使用系统 DPI 虚拟化，鼠标位置、窗口尺寸和截图像素之间可能出现偏差。

全局快捷键并非由第三方热键库监听，而是由 RegisterHotKey 注册固定的 Ctrl+Shift+A。后台守护线程阻塞在 GetMessageW 上；收到 WM_HOTKEY 后，再启动一个线程执行截图回调。这样既不会让 Windows 消息循环被截图过程阻塞，也不需要轮询键盘状态。

启动过程还带有一个简单的握手机制：热键线程通过 threading.Event 报告注册结果，主线程最多等待两秒。如果组合键已被其他程序占用，程序会关闭刚创建的 HTTP 服务并直接退出，而不是进入“看似启动、实际无法截图”的状态。

多显示器捕获与内存截图仓库

截图模块先用 GetCursorPos 取得鼠标的虚拟桌面坐标，再遍历 mss 返回的物理显示器：

```
monitor = next(
    m for m in monitors
    if m["left"] <= point.x < m["left"] + m["width"]
    and m["top"] <= point.y < m["top"] + m["height"]
)
```

因此 OCRBit 捕获的是**鼠标当前所在的整块显示器**，而不是把所有屏幕拼成一张超宽图片。显示器可以位于主屏左侧或上方，left、top 也允许为负数；这些坐标随后会用于把 Kiosk 窗口放回同一块屏幕。

CaptureStore 将截图编码为 PNG 字节，和宽、高、屏幕原点、创建时间一起保存在进程内的字典中。键是 secrets.token_urlsafe(18) 生成的随机 ID，字典访问由互斥锁保护。截图不会先写入临时文件；程序退出后，这些字节自然消失。

这里的清理机制值得准确描述：每次创建新截图时，仓库会删除创建时间早于一小时的条目，而不是运行一个定时清理线程。因此如果一小时后一直没有新截图，旧数据仍可能留在进程内存中，直到下一次截图或进程退出。

为什么用浏览器做框选界面

捕获完成后，后端优先寻找 Edge 或 Chrome，并以 --kiosk --new-window 启动独立窗口。窗口位置和大小来自刚才记录的显示器边界，浏览器数据则放入 %LOCALAPPDATA%\OCRbit\browser-profile，与用户的日常浏览器配置分离。若找不到支持的浏览器，程序才退回系统默认浏览器。

Kiosk 页面访问如下地址：

```
http://127.0.0.1:8765/capture/<capture_id>
```

在框选阶段，React 页面固定为 100vw × 100vh，截图也被拉到整个视口。用户拖动时记录的是 CSS 像素坐标；真正发起识别前，前端用图片的原始尺寸换算回截图像素：

```
const scaleX = image.naturalWidth / image.clientWidth
const scaleY = image.naturalHeight / image.clientHeight
```

这使框选逻辑不必假定浏览器坐标与截图像素永远是 $1:1$。后端收到矩形后还会把起点、宽度和高度限制在截图边界内，并保证裁切结果至少为 1×1 像素，避免畸形参数越界。

识别结束后，页面可以向 `/api/captures/<id>/close` 发请求。后端等待 150 ms，让 HTTP 响应先到达浏览器，再用 `taskkill /T /F` 关闭自己创建的浏览器进程树。它关闭的是本次捕获对应的独立进程，而不是用户正在使用的普通浏览器窗口。

一个没有 Web 框架的本地后端

OCRBit 的后端没有引入 Flask 或 FastAPI，而是直接使用 Python 标准库的 `ThreadingHTTPServer`。它同时承担两项工作：

- 托管 Vite 构建后的 `web-dist` 静态文件；
- 提供截图、设置和 AI 请求接口。

主要接口可以概括为：

路径	作用
`GET /api/health`	返回服务状态和当前模型
`GET /api/settings`	返回隐藏真实密钥后的设置
`GET /api/captures/<id>/image`	读取内存中的整屏截图
`POST /api/captures/<id>/close`	关闭本次 Kiosk 窗口
`POST /api/settings`	保存 API 地址、密钥和主题
`POST /api/ai`	裁切选区并流式调用模型

静态文件处理会先对路径执行 `resolve()`，并确认目标仍位于 `web-dist` 内；找不到文件时返回 `index.html`，让 React 根据 `/capture/<id>` 路径决定界面状态。JSON 请求体被限制在 1 MB 以内，响应统一带有 `X-Content-Type-Options: nosniff`。

整个服务只绑定 `127.0.0.1:8765`，局域网内的其他设备无法直接访问。但“仅监听本机”并不等同于完整的 Web 身份验证：接口本身没有会话或 CSRF 令牌，其主要安全边界仍是回环地址、随机截图 ID，以及用户本机的可信环境。

OCR 请求：先裁切，再编码，再上传

用户点击识别后，后端才从整屏 PNG 中裁出选区，将裁切结果编码为 Base64 Data URL，并构造兼容 OpenAI Chat Completions 格式的多模态消息：

```
{
  "model": "Qwen/Qwen3.5-397B-A17B",
  "stream": true,
  "enable_thinking": false,
  "temperature": 0.1,
  "messages": [{
    "role": "user",
    "content": [
      {"type": "text", "text": "<OCR prompt>"},
      {"type": "image_url", "image_url": {
        "url": "...",
        "detail": "high"
      }}
    ]
  }]
}
```

这里有两个取舍：

1. `detail: high` 让小字号、上下标和复杂公式得到更高分辨率的视觉输入；
2. `enable_thinking: false` 配合较低温度，减少延迟和自由发挥，让输出更接近忠实转写。

因此，“截图只保存在本地内存”并不意味着图像从不离开电脑。**整屏截图不直接上传，但用户选中的裁切区域会发送到配置的 SiliconFlow API**。对于含有密码、个人信息或未公开代码的画面，仍应先判断是否适合交给远端模型处理。

为什么它不仅是通用 OCR

OCRBit 的提示词明显针对数学、化学和技术材料做过优化。它要求模型保留标题、段落、列表、表格与代码块，并把数学表达式输出为 LaTeX。化学式要使用直立字体和下标，例如：

```
$\mathrm{MnC_2O_4}$
```

更细的一层是反应箭头语义。提示词明确区分带条件的单向反应和双向平衡，要求模型忠实保留原图方向，不能因为“化学上可能可逆”就擅自改写。这类约束不是传统 OCR 的字符识别，而是在限制视觉语言模型的推理倾向：模型需要理解版面，但不能补充图片中没有的信息。

OCR 结果还能作为下一轮纯文本请求的输入，用于翻译、总结、解释或代码分析。二次处理最多接收 200,000 个字符，并继续要求保留 Markdown、代码、表格和公式。换句话说，系统把工作流拆成了两个阶段：

```
图像 ──多模态识别──> 结构化 Markdown
                          |
                          └──文本模型处理──> 翻译 / 总结 / 解释 / 代码分析
```

SSE 如何把模型输出送回页面

`ai.py` 使用 `httpx.Client.stream()` 读取上游的流式响应，只取每个 `choices[0].delta.content`。本地服务再把文本块包装为 Server-Sent Events：

```
data: {"text":"第一个文本块"}

data: {"text":"第二个文本块"}

event: done
data: {}
```

浏览器通过 `fetch()` 获取响应体，用 `ReadableStream` 和 `TextDecoder` 增量解析以空行分隔的事件。每收到一块文本，React 就追加到当前结果中，因此用户不必等待整份 OCR 完成。前端还把 `AbortController` 传给请求，允许用户停止仍在生成的任务。

最终内容由 `react-markdown` 渲染，`remark-gfm` 负责表格等 GFM 语法，`remark-math` 与 `rehype-katex` 负责公式。原始 HTML 没有被启用，模型即使输出 `<script>` 也不会直接作为页面脚本执行；外部链接则统一在新标签页打开并附加 `rel="noreferrer"`。

API Key 的存储边界

设置可以来自 EXE 同目录的 `.env`，也可以在 Web 界面中填写。通过界面保存时，配置模块调用 Windows DPAPI 的 `CryptProtectData` 加密密钥，再写入：

```
%LOCALAPPDATA%\OCRbit\settings.json
```

程序没有使用 `CRYPTPROTECT_LOCAL_MACHINE`，所以密文默认绑定当前 Windows 用户。浏览器读取设置时只能获得 `••••••••••••` 掩码；真正的 API Key 只在 Python 进程中解密，并由后端添加到上游请求的 `Authorization` 头。

这能防止前端代码、浏览器开发者工具和静态资源直接读到密钥，但不能抵御已经取得当前用户权限、能够操纵本机进程的恶意程序。DPAPI 解决的是静态配置明文落盘问题，而不是把一台已被攻陷的电脑变成可信执行环境。

单文件 EXE 是怎样生成的

前端开发阶段由 Vite 提供热更新，并把 `/api` 代理到本地 Python 服务。发布时先运行 TypeScript 检查和 Vite 构建，生成 `web-dist`；随后 PyInstaller 使用：

```
--onefile --console --add-data "web-dist;web-dist"
```

把 Python 后端、依赖和前端静态资源封装进 `OCRbit.exe`。它保留命令行窗口，用于展示启动状态、请求日志和错误，也把窗口关闭作为退出整个服务的直接方式。

“单文件”并不代表程序完全不依赖系统环境：OCRBit 仍需要 Windows 10/11、网络、SiliconFlow API Key，以及一个可用的系统浏览器。它省去的是 Electron/Chromium 随应用重复打包的体积，而不是浏览器这一运行时本身。

这套架构的价值与边界

OCRBit 的实现规模不大，却展示了一种很实用的桌面工具架构：用少量 Win32 调用获得系统级入口，用 Python 处理本地数据和远端 API，再把复杂交互交给成熟的 Web 技术。相比完整桌面框架，它的模块边界清楚，开发时也能分别调试后端和前端。

它目前同样存在清晰的边界：

- 只支持 Windows，快捷键和模型均为固定值；
- OCR 依赖远端服务，离线时无法识别；
- 本地接口依赖回环地址隔离，没有额外鉴权；
- 截图缓存按“下一次创建时”惰性清理；
- 浏览器 Kiosk、强制结束进程和独立 profile 的行为需要在不同浏览器版本上持续验证。

这些限制并不削弱它的设计意义。恰恰相反，OCRBit 的代码把每一个取舍都摆在了明面上：它不是把网页套进桌面壳，而是让操作系统、浏览器和多模态模型各自承担最合适的一段工作。

参考

- OCRBit 源代码
- v1.0.0 Changelog
- Python 后端入口
- React 前端入口