# 第二堂課：影像 mmap 辨識 demo

## Project A · 完整逐步技術文件

架構 → 流程 → 照著資料流水線走 10 步（含實機位址/hex/實證）→ 程式碼 → 實測

PC 傳一張圖 → 板子用 mmap 讀進「影像緩衝區」→ 分析 → 回傳結果

淡江大學電機工程學系　AI 機器學習實驗室　·　2026-07

淡江大學
Tamkang University

AI 機器學習實驗室
AI & Machine Learning Lab
淡江大學 電機工程學系

# 這堂課做了什麼？

**從第一堂課的「點 LED」進化到「讀影像緩衝區」**

- **一句話：做一個「PC 傳圖 → 板子讀進來 → 分析這是什麼 → 回傳」的最小 demo**
- **核心一樣是 mmap——只是這次映射的是「影像緩衝區」，不是單一 LED 暫存器**

| 比較 | 第一堂課 | 第二堂課（本專案） |
|---|---|---|
| **mmap 讀什麼** | led_pio 單一暫存器 (4 bytes) | 影像緩衝區 upload.rgb (12 KB) |
| **位址/來源** | 0xFF203000 (FPGA 週邊) | 一個 RGB 檔（模擬影像緩衝區） |
| **做的事** | 寫數字 → 點燈 | 讀像素 → 算平均色 → 判斷 |

**→ 這段「讀取 + 分析」的程式碼，之後把「檔案緩衝區」換成「FPGA/CIS 影像緩衝區」就是真辨識。**

# 這個專案到底要幹嘛？

**先講大目標，才知道 A 在整條路的哪裡**

- **大目標：做一套「影像辨識系統」——用 CIS 感測器掃描圖像，讓機器自動判斷「這是什麼」**
- 分工：前段 FPGA 負責「擷取影像」，後段 ARM 負責「讀影像 → 辨識」（我們做後段）
- 但真 CIS 硬體還沒到 → 先用「一張電腦傳來的圖」模擬影像，把後段的「讀取 + 分析」骨架跑通
- **Project A = 這個骨架的最小可跑版本：**
  - 電腦傳圖 → 板子用 mmap 讀進「影像緩衝區」→ 算平均色/亮度做迷你判斷 → 回傳電腦
- **為什麼重要：這條「讀緩衝區 → 分析」的程式路，之後把來源換成真 CIS 影像、分析換成真模型，就是真辨識**

# 三個角色，各司其職

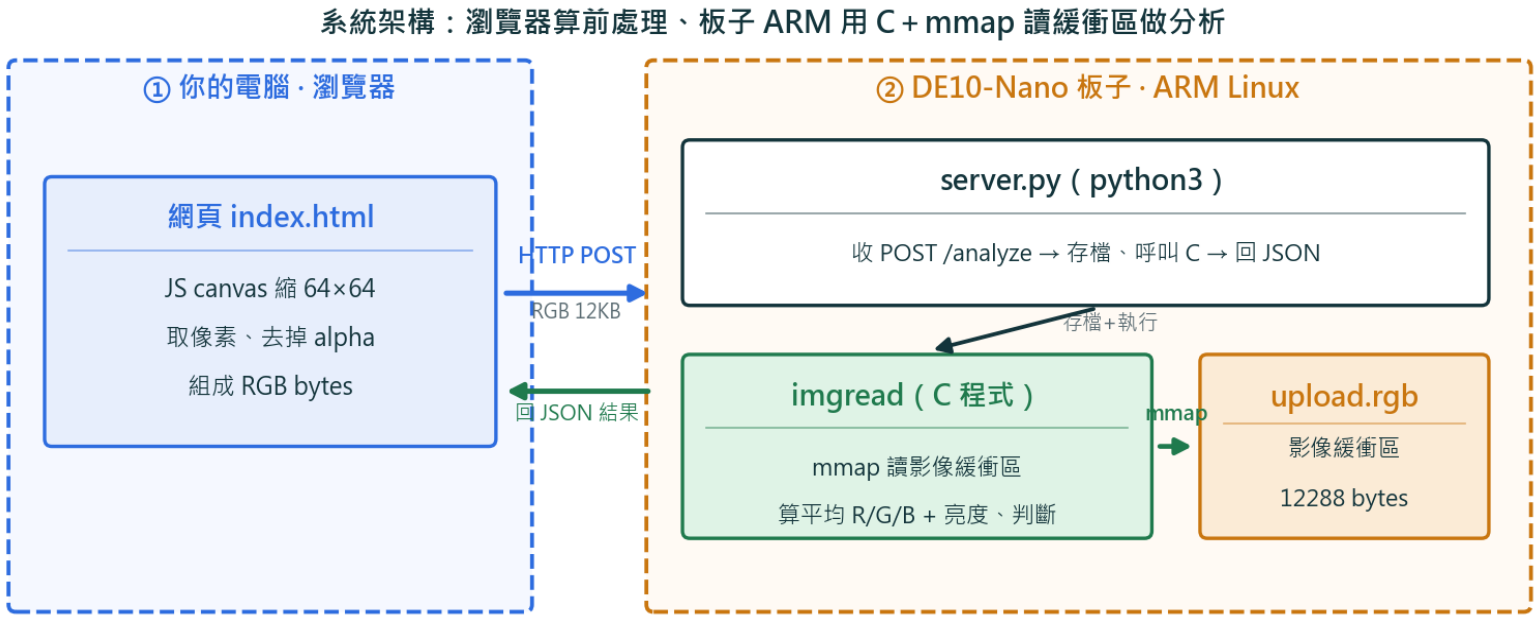
## 整個 demo 就這三個檔

| 檔案 | 跑在哪 | 語言 | 負責什麼 |
| --- | --- | --- | --- |
| `index.html` | 你的電腦（瀏覽器） | HTML/JS | 選圖、縮成 64×64、轉 RGB、上傳、顯示結果 |
| `server.py` | 板子 ARM | Python3 | 收 POST、存成 upload.rgb、呼叫 C、把 JSON 回傳 |
| `imgread.c` | 板子 ARM | C（gcc 編） | ★ mmap 讀影像緩衝區、算平均色/亮度、判斷、印 JSON |

**分工原則：瀏覽器做「影像前處理」(縮圖/轉格式)，板子只處理最單純的像素陣列——因為板子沒 PIL/numpy。**

# 系統架構：資料走到「ARM」不是 FPGA

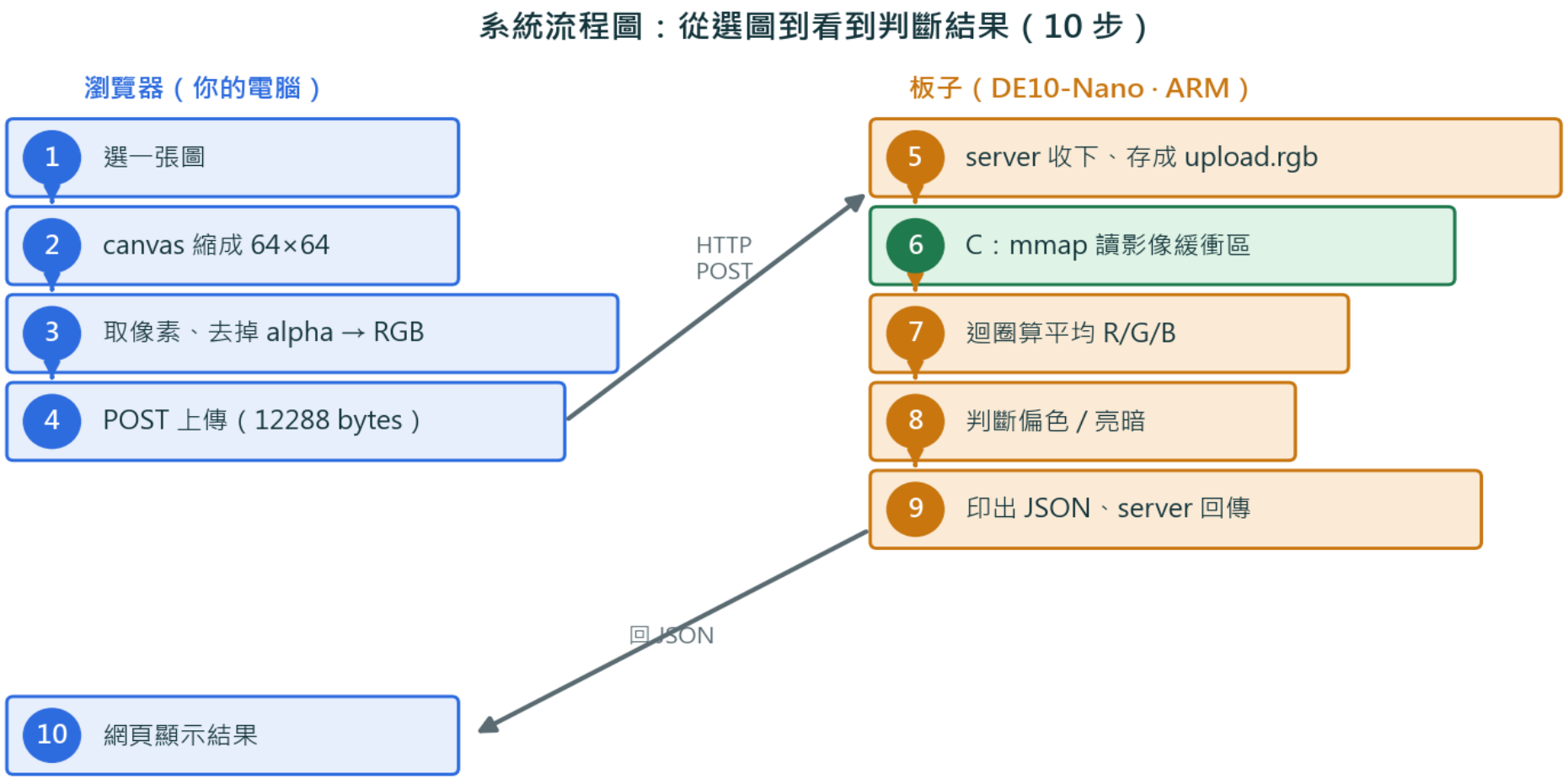
## 【全景 1/3】誰在哪裡、負責什麼 + 破除誤會




- ⚠ A 目前只用到 ARM ( 跑 Linux 那顆 )
- 資料路：網路 → ARM → SD 檔 → mmap → ARM 運算 → 回網路
- 全程在 ARM 這半邊，沒碰 FPGA fabric
- **FPGA 是「B 方案」( 真 CIS 影像 ) 才進場**
- 第一堂課戳 LED 才用到 FPGA 的 led_pio；A 不需要

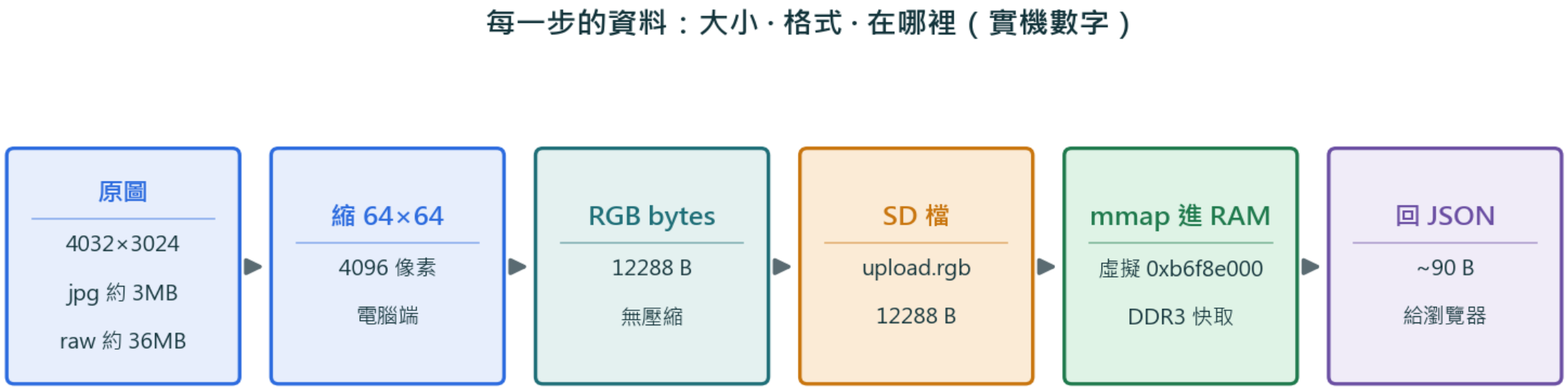
# 系統流程圖（資料流 10 步總覽）

【全景 2/3】等下每一步會逐一拆解



# 全景：每一步的資料有多大

【全景 3/3】一張大圖 → 一串小數字 → 一個判斷（實機數字）




重點：原圖幾十 MB，縮成 64×64 後只剩 12288 bytes（12KB）——小了約 3000 倍，板子才好即時處理。

# 你給的圖有多大？壓縮/解壓是什麼

STEP
1 / 10

**資料流水線　第 1 步 / 共 10 步　·　電腦端 · raw 像素 vs 壓縮檔**

- **原圖：任意大小，例如手機照片 4032×3024（約 1200 萬像素）**
- 「原始像素(raw)」大小 = 寬 × 高 × 3(RGB) = 4032×3024×3 ≈ 36 MB
- 但你檔案通常只有 ~3MB——因為存成 JPG「壓縮」過了

| 格式 | 壓縮 | 特性 |
| --- | --- | --- |
| JPG | 有損壓縮(DCT) | 檔案小很多，但丟掉人眼不敏感的細節 |
| PNG | 無損壓縮 | 還原 100% 原樣，檔案較大 |
| RAW RGB | 不壓縮 | 每像素 3 bytes 直接排，最單純但最大 |

**關鍵：瀏覽器把圖載入時會「解碼(解壓縮)」還原成原始像素；我們處理的是「還原後的原始像素」，不是壓縮檔。**

# 縮成 64×64、變成一串 RGB



**資料流水線　第 2 步 / 共 10 步　·　電腦端瀏覽器 · 用 JS 做影像前處理**

### 資料怎麼一步步變（一張圖 → 一個判斷）


- 縮圖：ctx.drawImage(img, 0,0, 64,64) → 64×64 = 4096 像素（解析度 = 像素數）
- 取像素：getImageData() 拿到 RGBA（每像素 4 個 byte）→ 丟掉 A(透明度) → RGB
- **結果：4096 像素 × 3 = 12288 個 byte，就是「影像緩衝區」的內容**
- 用到的 JS 功能：Image / canvas / drawImage / getImageData / Uint8Array / fetch

# 程式碼：index.html（前端·瀏覽器）



**資料流水線　第 2 步 / 共 10 步　·　把圖變成板子能吃的 RGB，再上傳**

跑在：你的電腦瀏覽器

語言：JavaScript

```
// 圖載入後：畫到 64x64 的 canvas（等於縮圖）
ctx.drawImage(img, 0,0, 64,64);
const d = ctx.getImageData(0,0,64,64).data;    // 拿到 RGBA

// 每個像素取 R,G,B、丟掉透明度 A → 組成 12288 bytes
const rgb = new Uint8Array(64*64*3);
for (let i=0,j=0; i<d.length; i+=4) {
  rgb[j++]=d[i]; rgb[j++]=d[i+1]; rgb[j++]=d[i+2];
}

// 上傳到板子，等 JSON 結果
fetch('/analyze', { method:'POST', body: rgb })
  .then(r => r.json()).then(showResult);
```

- **為什麼在瀏覽器縮圖：**
  - 板子沒 PIL/numpy，不會解 jpg/png
  - 瀏覽器 canvas 免費幫你縮圖 + 解碼
  - 只送最單純的原始 RGB，板子超好處理
- **RGBA→RGB：丟掉第 4 個 byte(透明度)**

# 怎麼「丟」到板子？



資料流水線　第 3 步 / 共 10 步　·　HTTP POST 走網路線（Ethernet）→ 板子 ARM

電腦 → 板子 ARM

```
// 瀏覽器把那 12288 bytes 當 HTTP 請求的內容(body)送出
fetch('http://192.168.50.199:8080/analyze', {
    method: 'POST',
    body: rgb            // rgb = 12288 bytes 的原始 RGB
}).then(r => r.json())   // 等板子回 JSON
 .then(showResult);
```

- **傳輸方式：HTTP POST**
- 走實體：板子的 Gigabit 網路線
- 送到誰：板子 ARM 上的 server.py
- 內容：那 12288 bytes（不再壓縮）
- 為什麼小：已縮到 12KB，網路瞬間到

注意：這一步「丟給 ARM」，不是丟給 FPGA。ARM 上的 Linux web server 收下這包資料。

# 板子收下，先放哪裡？

STEP 4 / 10

資料流水線　第 4 步 / 共 10 步　·　server.py 把它存成 SD 卡上一個檔案

板子 ARM · server.py（Python3）

```
def do_POST(self):
  if self.path == '/analyze':
    n = int(self.headers.get('Content-Length','0'))  # =12288
    data = self.rfile.read(n)                # 收下那 12288 bytes
    open('upload.rgb','wb').write(data)      # 存成 SD 卡上的檔案
    out = subprocess.check_output(           # 叫 C 程式來讀+分析
            ['./imgread','upload.rgb','64','64'])
    self._send(out)                          # 把結果回傳
```

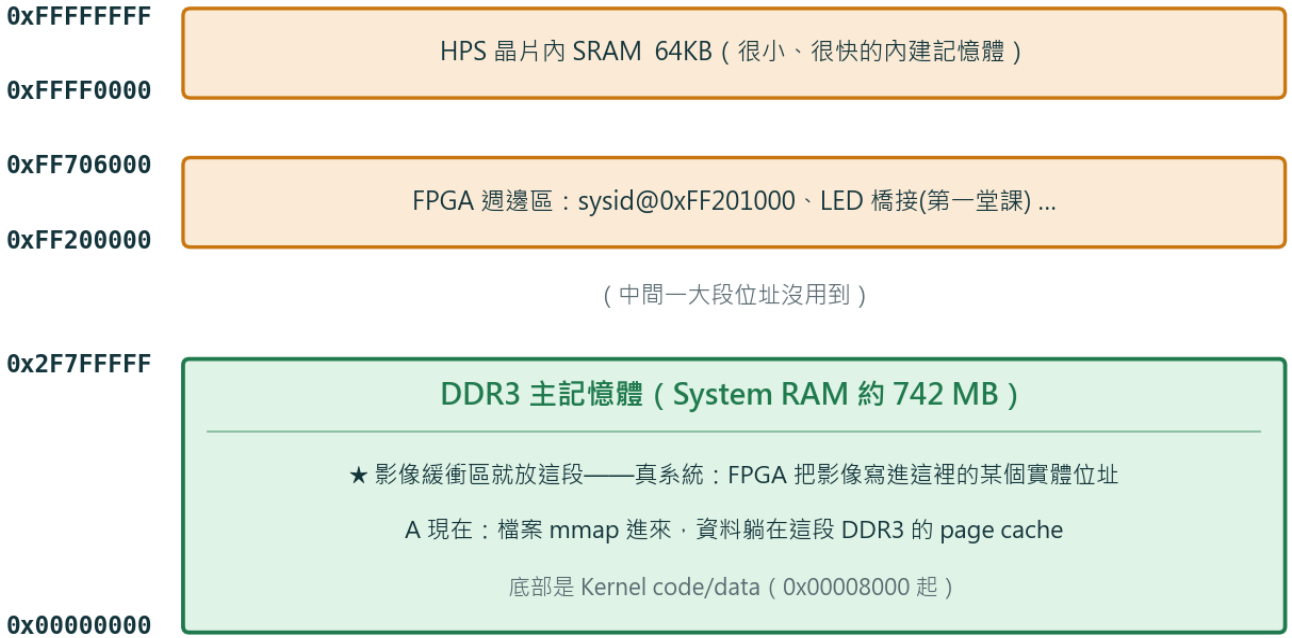
- **存的位置：/home/root/www_demo/upload.rgb（板子 SD 卡上，實機 12288 bytes）**
- 板子 python3 超精簡：連 json 模組都沒有 → 不 import json，讓 C 直接印 JSON 文字、server 只轉發
- 用到的功能：do_POST / rfile.read（收位元組）/ open().write（寫檔）/ subprocess（叫 C）

# 進 RAM 了嗎？在哪個位址？

STEP 5 / 10

## 資料流水線　第 5 步 / 共 10 步　·　mmap 把檔案映射進記憶體（實機位址）

板子的記憶體地圖（實機 /proc/iomem 抓的實體位址）




- imgread 用 mmap 把 upload.rgb 映射進來
- 實機 mmap 位址：
  - 0xb6f8e000 ~ 0xb6f91000
  - （差 0x3000 = 12288，剛好）
- 這是「虛擬位址」——程式看到的
- 資料實際躺在 DDR3 的 page cache
- DDR3 實體範圍：0x0 ~ 0x2F7FFFFF

# 實證：實機 /proc/self/maps



**資料流水線　第 5 步 / 共 10 步　·　不是講理論，是板子上真的抓的**

```
# imgread 執行時，作業系統對映表(節錄)：
b6f8e000-b6f91000  r--s  00000000  b3:02 186585  /home/root/www_demo/upload.rgb
                   ^^^^                  ^^^^^^  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                   唯讀·共享             inode    來源就是那個檔案

# 白話：虛擬位址 0xb6f8e000 起的這段，內容 = upload.rgb 這個檔
#       CPU 讀這段位址，MMU 幫忙對到 DDR3 裡放它的實體頁
```

- **虛擬位址 vs 實體位址：程式用「虛擬位址」(0xb6f8e000)，MMU 硬體把它翻譯到 DDR3 某個實體頁**
- A 這裡是「檔案映射」(file-backed)：安全、不會踩到 Linux 的記憶體
- 真系統會改成「實體映射」：mmap /dev/mem 直接對到 FPGA 寫影像的那個 DDR3 實體位址

# 存進 RAM 的東西長什麼樣子？

STEP 6 / 10

**資料流水線　第 6 步 / 共 10 步　·　就是一長串數字，沒有檔頭、沒有壓縮**

- **以「純綠測試圖」為例，每個像素都是 (R,G,B) = (40,190,60)**
- 換成 16 進位：40=0x28、190=0xBE、60=0x3C

```
位址(offset)   內容(hex)                          意義
0x0000         28 BE 3C                          像素0: R=0x28 G=0xBE B=0x3C
0x0003         28 BE 3C                          像素1: 同上(整張都綠)
0x0006         28 BE 3C                          像素2 ...
  ...          ... (一路排下去，共 4096 組) ...
0x2FFD         28 BE 3C                          像素4095 (最後一個)
──────────────────────────────────────────────────────────────
總長 12288 bytes = 4096 像素 × 3。沒有檔頭、沒有壓縮，純像素值。
```

**所以「存進 RAM 的東西」= 一張攤平的數字表；C 程式把它當一個 uint8 陣列 buf[] 直接讀。**

# ARM 怎麼把它「取回來」讀？

STEP 7 / 10

資料流水線　第 7 步 / 共 10 步　·　mmap 之後，當一般陣列存取

板子 ARM · imgread.c (C)

```
int fd = open("upload.rgb", O_RDONLY);      // 打開檔案
struct stat st; fstat(fd, &st);             // 知道大小 12288

uint8_t *buf = mmap(NULL, 12288,            // ★ 映射進記憶體
          PROT_READ, MAP_SHARED, fd, 0);    //    回傳虛擬位址(0xb6f8e000)

// 之後 buf[0], buf[1], buf[2] ... 就是像素的 R,G,B
// 讀完 munmap(buf, 12288); close(fd);
```

- 用到的 C 功能（<sys/mman.h>）：open / fstat / mmap / munmap / close
- 核心：mmap 把「檔案/硬體那塊記憶體」變成程式裡的一個陣列 buf[]——跟第一堂課戳 led_pio 同一招

# ARM 對資料做了什麼運算？



**資料流水線　第 8 步 / 共 10 步　·　逐像素統計 → 一個判斷**

```
unsigned long long sr=0, sg=0, sb=0;
for (long i = 0; i < 4096; i++) {   // 走過全部 4096 個像素
    sr += buf[i*3 + 0];             // 累加所有 R
    sg += buf[i*3 + 1];             // 累加所有 G
    sb += buf[i*3 + 2];             // 累加所有 B
}
int r = sr/4096, g = sg/4096, b = sb/4096;      // 平均色
int bright = (r + g + b) / 3;                   // 亮度
color = (r>=g && r>=b) ? "red" : (g>=b) ? "green" : "blue";
lum   = (bright >= 128) ? "bright" : "dark";
```

- **做的事：4096 次迴圈把 R/G/B 分別加總 → 除以像素數 = 平均色 → 比大小判「偏哪色、亮或暗」**
- 這是最陽春的「迷你辨識」；之後可換成顏色直方圖 / 真模型（同樣讀 buf[] 這段不變）

# 傳回電腦、電腦端做什麼

STEP
9-10 / 10

**資料流水線　第 9-10 步 / 共 10 步　·　一路原路回去**

- **第 9 步　ARM 印出一行 JSON（純文字），server.py 用 HTTP 回應送回電腦**

```
{"w":64,"h":64,"avg":{"r":40,"g":190,"b":60},"bright":96,"color":"green","lum":"dark"}
```

- **第 10 步　電腦端瀏覽器收到 JSON：**
  - fetch(...).then(r => r.json()) 把文字解析成物件
  - 把平均色畫成色塊、顯示「判斷：偏綠 · 暗」
  - 整趟來回（傳圖→分析→顯示）在區網內幾乎瞬間完成

**回傳的 JSON 只有約 90 bytes——一張 36MB 的圖，最後濃縮成一句「偏綠·暗」。**

# mmap 特寫：第一堂課 vs 第二堂課

**【觀念】同一招，讀不同的東西**

| | 第一堂課 (led_pio) | 第二堂課 (影像緩衝區) |
|---|---|---|
| **open 什麼** | `/dev/mem` (實體記憶體) | `upload.rgb` (檔案) |
| **mmap 大小** | 一頁 (4KB) | `64*64*3 = 12288` |
| **怎麼用** | `*led = 0x7F` (寫) | `sum += buf[i]` (讀) |
| **意義** | ARM 寫 → 點 FPGA 的燈 | ARM 讀 → 分析影像 |

**未來第三步：把「upload.rgb 檔」換成「FPGA 寫進 DDR3 的影像位址」，mmap 那行改個位址即可，讀取分析全不變。**

# 動手做：這個 demo 怎麼做出來的

**【工程實務】照這 6 步，你也能重現**

| | | |
|---|---|---|
| 1 | **探板子環境** | SSH 進去確認：gcc 有、python3 無 mmap 模組/無 json 模組、web 是靜態 http.server、空間 1.1G |
| 2 | **寫 C 分析器** | imgread.c：open + mmap 讀 RGB 緩衝區 → 算平均色/亮度 → 印純 ASCII 的 JSON |
| 3 | **編譯 + 單測** | 板上 gcc -O2 -o imgread imgread.c；用純色測試圖驗證：紅→red、綠→green |
| 4 | **寫收發 server** | server.py：GET 送網頁、POST /analyze 收 RGB→存→跑 imgread→回 JSON（不用 json 模組） |
| 5 | **換 web 服務** | 改 systemd 的 ExecStart 成 python3 server.py，daemon-reload + restart（不要 pkill！） |
| 6 | **端到端測** | 從 PC curl 上傳測試圖 → 收到正確 JSON；瀏覽器開網頁上傳真圖看結果 |

# 踩到的 4 個坑（板子超精簡）

**【工程實務】第一次做一定會遇到**

### ⚠ python3 沒有 mmap 模組

→ 戳硬體/讀緩衝區用 C + mmap（跟第一堂課一樣）；python 不做這段。

### ⚠ python3 連 json 模組都沒有

→ server.py 不 import json；讓 C 直接印 JSON 字串、server 只轉發，錯誤訊息手動組。

### ⚠ 原本 web 是純靜態、不能上傳

→ 改寫成自訂 HTTP server，加一個 do_POST 處理 /analyze。

### ⚠ 改 web 服務別用 pkill

→ 會連自己的 SSH 一起殺掉；一律用 systemctl restart。

# 實測結果（端到端驗證）

**【工程實務】從 PC 上傳測試圖，板子真的回了**

### ① 板上 C 分析器單測（純色圖）

```
red   -> {"avg":{"r":210,"g":40,"b":40}, "bright":96, "color":"red",   "lum":"dark"}
green -> {"avg":{"r":40,"g":190,"b":60}, "bright":96, "color":"green", "lum":"dark"}
```

### ② 從 PC 用 curl 打板子的 /analyze（走完整 HTTP 路徑）

```
$ curl --data-binary @test_red.rgb http://192.168.50.199:8080/analyze
{"w":64,"h":64,"avg":{"r":210,"g":40,"b":40},"bright":96,"color":"red","lum":"dark"}

首頁 HTTP 200 · service active · 重開機不掉（systemd）
```

**✓ 全部通過：紅圖判 red、綠圖判 green；瀏覽器上傳真圖也走同一條路。**

# A（現在）vs 真辨識系統（未來）

**【收尾】同一條路，換掉兩塊就變真的**

| 步驟 | A 現在（模擬） | 真系統（未來） |
|---|---|---|
| **影像來源** | 電腦傳的檔案 | FPGA 從 CIS 擷取、寫進 DDR3 |
| **存哪 / 位址** | SD 檔 → mmap 虛擬 0xb6f8e000 | DDR3 實體位址(0x0~0x2F7FFFFF 內某段) |
| **mmap 對象** | mmap 那個檔 | mmap /dev/mem 的實體位址 |
| **分析** | 平均色 / 亮度 | 顏色直方圖 / MobileNet 真模型 |
| **有用到 FPGA?** | 沒有(只用 ARM) | 有(FPGA 擷取 + ARM 辨識) |

**A 已把「讀緩衝區 → 分析 → 回傳」整條軟體路跑通；未來只換「來源」和「模型」兩塊。**

# 總結：每一步的資料一覽（真數字）

## 第二堂課一頁記住

| 階段 | 資料大小/格式 | 在哪裡 |
|---|---|---|
| **原圖** | 4032×3024, JPG 約 3MB | 電腦 |
| **縮圖後 RGB** | 12288 bytes 原始像素 | 瀏覽器記憶體 |
| **上傳** | 12288 bytes (HTTP POST) | 網路線 → ARM |
| **存檔** | upload.rgb 12288 bytes | 板子 SD 卡 |
| **mmap 讀** | 虛擬 0xb6f8e000~0xb6f91000 | DDR3(實體 0x0~0x2F7FFFFF) |
| **回傳** | JSON 約 90 bytes | ARM → 網路 → 電腦 |

**一句話：一張 36MB 的圖 → 縮成 12KB → mmap 進 DDR3 → ARM 讀完算出「偏綠·暗」→ 回傳 90 bytes。這就是影像辨識後段的最小骨架。**

`demo 網址（區網）： http://192.168.50.199:8080/ · 原始碼：教學_第二堂課_影像mmap\`