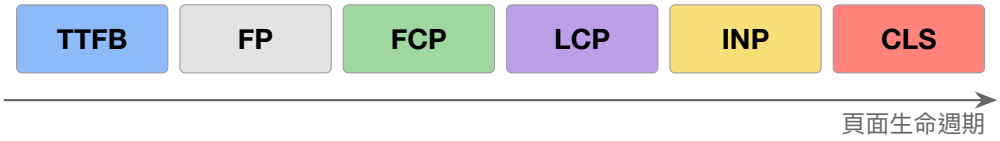
前端效能實戰

從瀏覽器連線到 Web Vitals 的完整課程



課程講義

主題：瀏覽器連線建立、效能量化指標、Performance API、DevTools 效能面板、快取、Service Worker、CDN、伺服器端渲染

2026 年 9 月

本講義依據 web.dev、MDN、W3C 規範與 Chrome for Developers 官方文件整理，
並補充 2024–2026 年的最新變動（例如 INP 取代 FID、bfcache 與 Speculation Rules 的演進）。

目錄

課程導論：為什麼要學前端效能 **1**

0.1 效能不是「跑分」，而是使用者的感受 1
0.2 一張地圖：從輸入網址到頁面可互動 2
0.3 實驗室數據 vs. 真實使用者數據 2
0.4 本課程使用的工具 3
0.5 建議的上課順序 4

第一章 fetch 之前：瀏覽器如何建立連線 **5**

1.1 一個導覽（navigation）的完整流程 5
1.2 RTT：網路效能的基本單位 6
1.2.1 建立一條新連線要幾個 RTT？ 7
1.3 各環節可能慢在哪裡、如何優化 8
1.3.1 重導（Redirect） 8
1.3.2 DNS 查詢 8
1.3.3 TCP 交握與慢啟動 9
1.3.4 TLS 交握 9
1.3.5 Service Worker 啟動 10
1.3.6 伺服器處理時間（think time） 10
1.4 把往返「提前做」：資源提示與 Early Hints 10
1.4.1 dns-prefetch 與 preconnect 10
1.4.2 HTTP 103 Early Hints 11
1.5 連線建立之後：關鍵渲染路徑速覽 11
1.6 課堂實作 12

第二章 效能量化指標總覽：TTFB、FP、FCP、FMP、LCP、FID、INP、CLS **15**

2.1 把指標放到時間線上 15
2.2 TTFB：Time to First Byte 16
2.3 FP：First Paint 16
2.4 FCP：First Contentful Paint 16
2.5 FMP：First Meaningful Paint（已淘汰） 17
2.6 LCP：Largest Contentful Paint 17

2.7 FID：First Input Delay（已由 INP 取代） . . . 18
2.8 INP：Interaction to Next Paint . . . 19
2.9 CLS：Cumulative Layout Shift . . . 20
2.10 補充：Lighthouse 中的其他實驗室指標 . . . 21
2.11 總整理 . . . 22
2.12 課堂實作 . . . 23

第三章 Web Vitals 深入：LCP、INP、CLS 的診斷與優化 25
3.1 LCP：四個子部分 . . . 25
3.1.1 ① TTFB . . . 26
3.1.2 ②資源載入延遲：讓瀏覽器「早點知道」 . . . 26
3.1.3 ③資源載入時間：讓資源「變小、變近」 . . . 27
3.1.4 ④元素渲染延遲：資源到了卻沒畫出來 . . . 27
3.2 INP：三個子部分 . . . 28
3.2.1 長任務與 50 ms 法則 . . . 28
3.2.2 ①輸入延遲：別讓主執行緒在使用者點下去時正忙 . . . 28
3.2.3 ②處理時間：只做「這一幀需要的事」 . . . 30
3.2.4 ③呈現延遲：讓渲染變便宜 . . . 30
3.2.5 找出兇手：Long Animation Frames . . . 31
3.3 CLS：找出誰在跳，並預留空間 . . . 31
3.3.1 再看一次公式 . . . 31
3.3.2 常見成因與對策 . . . 32
3.3.3 用 API 找出是哪個元素在跳 . . . 33
3.4 綜合案例：一個電商商品頁的診斷流程 . . . 33
3.5 課堂實作 . . . 34

第四章 Performance API：瀏覽器原生的量測工具箱 36
4.1 整體架構：Performance Timeline . . . 36
4.1.1 高解析時間 . . . 37
4.1.2 兩種取得 entry 的方式 . . . 37
4.2 Navigation Timing：整份文件的載入時間線 . . . 39
4.2.1 Level 1 vs. Level 2 . . . 39
4.2.2 處理模型：完整時間線 . . . 39
4.2.3 常用的衍生計算 . . . 39
4.3 Resource Timing：每一個子資源的時間線 . . . 41
4.4 其他 entry type 一覽 . . . 43
4.4.1 Paint Timing（`paint`） . . . 43
4.4.2 Largest Contentful Paint（`largest-contentful-paint`） . . . 43

4.4.3 Event Timing（event、first-input） . . . 43
4.4.4 Layout Instability（layout-shift） . . . 43
4.4.5 Long Tasks（longtask）與 Long Animation Frames（long-animation-frame） . . . 44
4.4.6 User Timing（mark、measure） . . . 44
4.4.7 Server Timing . . . 45
4.4.8 Element Timing（element） . . . 45
4.5 把它們串起來：web-vitals 套件 . . . 45
4.5.1 attribution 版本：知道「為什麼」 . . . 46
4.6 課堂實作 . . . 47

第五章 Chrome DevTools Performance 面板實戰 49
5.1 開啟面板：Live Metrics 首頁 . . . 49
5.2 錄製一段 trace . . . 50
5.3 版面導覽 . . . 51
5.3.1 火焰圖的顏色語義 . . . 51
5.3.2 操作技巧 . . . 52
5.4 Insights 側欄：把 Lighthouse 的建議搬進 trace . . . 52
5.5 實戰流程 A：診斷 LCP . . . 53
5.6 實戰流程 B：診斷 INP . . . 54
5.7 實戰流程 C：診斷 CLS . . . 55
5.8 搭配 Network 面板 . . . 55
5.9 其他相關工具 . . . 56
5.10 課堂實作 . . . 56

第六章 快取策略：HTTP Cache、bfcache 與 Speculation Rules 58
6.1 瀏覽器的快取層級 . . . 58
6.2 HTTP 快取：新鮮度與驗證 . . . 58
6.2.1 控制新鮮度：Cache-Control . . . 60
6.2.2 驗證器：ETag 與 Last-Modified . . . 61
6.2.3 瀏覽器的決策流程 . . . 61
6.3 為不同資源設計策略 . . . 61
6.4 驗證快取是否真的命中 . . . 64
6.5 bfcache：讓「上一頁」瞬間完成 . . . 65
6.5.1 哪些情況會失去資格 . . . 66
6.5.2 測試與監測 . . . 66
6.6 Speculation Rules：預先抓取與預先渲染下一頁 . . . 67
6.6.1 在目標頁面偵測與配合 . . . 68

6.7 課堂實作 70

第七章 Service Worker 與效能 72
7.1 Service Worker 是什麼 72
7.2 生命週期 73
7.3 Cache Storage 與快取策略 74
7.4 SW 的效能成本與對策 77
7.4.1 Navigation preload 78
7.4.2 Static Routing API：讓部分請求完全不經過 SW 78
7.5 用 Workbox 少寫、少錯 79
7.6 觀察與除錯 80
7.7 課堂實作 81

第八章 CDN：把內容搬到使用者旁邊 83
8.1 距離就是時間 83
8.2 架構：PoP、Anycast 與 origin shield 84
8.3 讓 CDN 正確快取：標頭與清除 85
8.3.1 告訴 CDN 該存多久 85
8.3.2 清除（purge）與版本化 86
8.4 判讀命中與命中率 86
8.5 進階功能：影像、Early Hints、邊緣運算 88
8.5.1 影像 CDN 88
8.5.2 103 Early Hints 88
8.5.3 邊緣運算（edge compute） 89
8.6 常見錯誤 90
8.7 CDN 與 Web Vitals 的對應 90
8.8 課堂實作 91

第九章 伺服器端渲染（SSR）與渲染策略 93
9.1 渲染策略光譜 93
9.2 Hydration：SSR 頁面的隱形成本 95
9.2.1 為什麼 hydration 拖慢 INP 95
9.2.2 減輕 hydration 的技術 95
9.3 串流 SSR 與 Suspense 96
9.4 SSG、ISR 與「把 SSR 結果快取起來」 99
9.5 為每種頁面選策略 100
9.6 診斷 SSR 頁面 100
9.7 課堂實作 102

附錄 A 詞彙表、速查表、綜合練習與參考資料 **104**

A.1 詞彙表 104

A.2 指標門檻速查 106

A.3 綜合練習：一次完整的效能稽核 106

A.4 參考資料 109

課程導論：為什麼要學前端效能

學習目標

- 理解「效能」對使用者體驗、商業成果與搜尋排名的實際影響。
- 建立一張「從輸入網址到頁面可互動」的完整心智地圖，並知道每個階段對應本講義的哪一章。
- 分辨**實驗室數據**（Lab data）與**真實使用者數據**（Field data / RUM），理解 p75 與 28 天滾動視窗的意義。
- 認識本課程會用到的工具：Chrome DevTools、Lighthouse、PageSpeed Insights、CrUX、`web-vitals` 套件。

0.1 效能不是「跑分」，而是使用者的感受

當我們說一個網頁「慢」，其實通常包含好幾種不同的感受：

- 點下連結後，畫面一片空白，不知道「有沒有在動」。
- 畫面出現了一些東西，但最重要的內容（主圖、標題、商品資訊）還沒出來。
- 內容看得到，但點按鈕、打字卻沒有反應，或者反應很慢。
- 正要點某個按鈕，版面突然跳動，結果點到廣告。

Google 的 Web Vitals 計畫把這些感受整理成四個問題，並替每個問題設計了可量化的指標：

使用者的問題	對應的體驗面向	對應指標
它有在動嗎？	導覽是否成功開始、伺服器是否有回應	TTFB、FP、FCP
它有用嗎？	主要內容是否已經呈現	LCP（過去是 FMP）
它能用嗎？	使用者可否與頁面互動且得到即時回饋	INP（過去是 FID）、TBT
它令人愉悅嗎？	互動流暢、版面穩定、沒有卡頓	CLS、動畫流暢度

這張表是整本講義的骨架。第二章會逐一定義這些指標，第三章會深入探討如何診斷與優化。

效能為什麼重要：三個層面

1. **使用者體驗與商業成果**：載入時間與跳出率、轉換率高度相關。業界大量案例顯示，LCP 每改善數百毫秒，就能觀察到轉換率提升；而 INP 不佳（互動卡頓）會直接造成使用者放棄操作。
2. **搜尋排名（Page Experience）**：Google 將 Core Web Vitals（LCP、INP、CLS）納入排名訊號，並在 Search Console 中以「良好 / 待改善 / 不佳」呈現。
3. **工程品質**：效能是「系統性」的問題，牽涉網路、瀏覽器、伺服器與程式架構。學會量測與診斷，等於學會用瀏覽器的角度理解自己的程式。

0.2 一張地圖：從輸入網址到頁面可互動

圖 1 是本課程的整體地圖。橫軸是時間，每個階段都可能成為瓶頸，也都有對應的量測方式與優化手段。

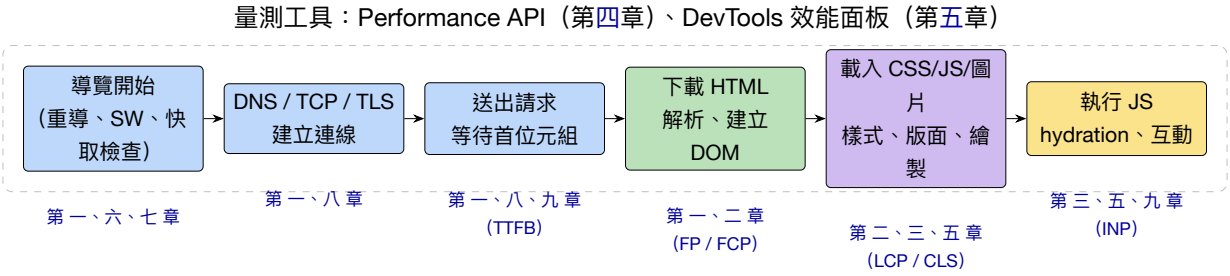


圖 1: 課程地圖：頁面載入的各階段與對應章節

0.3 實驗室數據 vs. 真實使用者數據

學效能的第一個觀念轉變，是理解「我在自己電腦上測的數字」和「使用者實際體驗到的數字」是兩回事。

Lab data 與 Field data

- **實驗室數據（Lab data）**：在受控環境（固定裝置、固定網速、無快取）下由工具（Lighthouse、WebPageTest、DevTools）產生。可重現、適合除錯，但無法反映真實使用者的裝置與網路分布。
- **真實使用者數據（Field data / RUM，Real User Monitoring）**：由真實使用者的瀏覽器回報。Google 的 **Chrome User Experience Report**（CrUX）是最廣為使用的公開資料集，也是 Search Console 與 PageSpeed Insights 顯示 Core Web Vitals 狀態的來源。

為什麼是 p75？

Core Web Vitals 的評分採用**第 75 百分位數（p75）**：把某個頁面在 28 天內所有真實造訪的 LCP 值排序，取第 75% 位置的值。這代表「至少 75% 的使用者體驗到的 LCP 不超過這個值」。這個設計刻意偏向「較差的那一端」：頁面在你的高階筆電加光纖上很快沒有意義，它必須在四分之三的使用者手上（包含中階手機加不穩定的行動網路）也夠快。

常見誤解

- Lighthouse 拿到 100 分，不代表 Search Console 會通過 Core Web Vitals。Lighthouse 是實驗室數據；Core Web Vitals 評分看的是 CrUX 的真實使用者 p75。
- 反過來，某些互動指標（FID、INP）**只能**在真實使用者環境量測，因為實驗室沒有真人互動。Lighthouse 用 TBT（Total Blocking Time）當作互動性的代理指標。
- 三個 Core Web Vitals 必須**全部**達到「良好」，該 URL 群組才算通過。

0.4 本課程使用的工具

工具	資料類型	用途
Chrome DevTools Performance 面板	實驗室	錄製 trace、看主執行緒火焰圖、LCP/INP/CLS 子部分、Insights（第五章）
Chrome DevTools Network 面板	實驗室	檢查每個請求的 DNS/連線/TTFB/下載時間、快取狀態、協定
Lighthouse	實驗室	自動化稽核、分數與建議；PSI 與 DevTools 內建
PageSpeed Insights（PSI）	兩者	同時顯示 CrUX 真實數據與 Lighthouse 實驗室數據
CrUX Dashboard / API	真實使用者	查詢任何有足夠流量之網站的 p75 指標與歷史趨勢
`web-vitals` 套件	真實使用者	在自己的網站收集 LCP/INP/CLS 並回傳到分析系統（第四章）
Performance API	兩者	瀏覽器原生 API，是所有上述工具的資料來源（第四章）
WebPageTest	實驗室	多地區、多裝置的瀑布圖與影片比較

0.5 建議的上課順序

本講義依主題分章，可依需求跳讀。若以「先理解指標與工具，再深入各項優化」為原則，建議順序如下：

1. **第一堂（本次重點）**：第二章效能指標總覽 → 第三章 Web Vitals 深入 → 第五章 DevTools Performance 面板實戰。目標是能自己錄一段 trace、找出 LCP 元素、看懂 INP 與 CLS 的成因。
2. **第二堂**：第一章 fetch 之前瀏覽器在做什麼（RTT、DNS/TCP/TLS、HTTP/3）→ 第四章 Performance API（Navigation / Resource Timing、PerformanceObserver）。
3. **第三堂**：第六章快取策略 → 第七章 Service Worker → 第八章 CDN。這三章合起來回答「如何讓資源更快到達瀏覽器」。
4. **第四堂**：第九章伺服器端渲染與渲染策略，把前面所有指標串起來，討論架構層級的取捨。

每章結尾都有「課堂實作」與「思考題」，附錄 A 另有詞彙表、綜合練習與參考資料。

本章回顧

- 效能指標是使用者感受的量化：「有在動嗎、有用嗎、能用嗎、愉悅嗎」。
- Core Web Vitals 目前是 LCP、INP、CLS 三項，以真實使用者 28 天 p75 評分。
- 實驗室數據用來除錯與重現，真實使用者數據用來評估與排序優先順序，兩者缺一不可。
- 整條載入流程的每個階段都對應到本講義的某一章，請把圖 1 當成導航。

第一章　fetch 之前：瀏覽器如何建立連線

學習目標

- 說出從「按下 Enter」到「第一個 HTML 位元組回來」之間，瀏覽器依序做了哪些事。
- 解釋 **RTT**（Round-Trip Time，往返時間）為何是網路效能的基本單位，並能估算一次新連線需要幾個 RTT。
- 指出 DNS、TCP、TLS、重導、Service Worker 啟動等環節各自可能慢在哪裡，以及對應的優化手段。
- 理解 HTTP/1.1、HTTP/2、HTTP/3（QUIC）在連線成本上的差異。
- 用 Performance API 與 DevTools Network 面板實際量出每個階段花了多久。

1.1　一個導覽（navigation）的完整流程

當使用者在網址列輸入 `https://shop.example.com/` 並按下 Enter，瀏覽器並不是立刻「fetch」。在真正送出 HTTP 請求之前，有一連串的準備工作；圖 1.1 把這些步驟依時間排開，並標出 Navigation Timing API 中對應的時間點屬性（第四章會完整介紹這個 API）。

逐步說明：

1. **卸載舊頁面**：若目前分頁已有頁面，瀏覽器先觸發 `beforeunload`、`pagehide`、`unload`。註冊 `unload` 監聽器會讓頁面無法進入 bfcache（第六章），是常見的效能反模式。
2. **HSTS 與重導**：瀏覽器先查 HSTS 清單，若該網域在清單中就直接升級為 HTTPS，省下一次 `http` → `https` 的重導。接著若伺服器回傳 301/302/307/308，瀏覽器必須重新對新 URL 走一遍流程；**每一次重導至少多一個 RTT**，若換了網域還要重新做 DNS、TCP、TLS。
3. **Service Worker**：若此 URL 在某個已註冊的 Service Worker 控制範圍內，瀏覽器必須先啟動該 worker（`workerStart`），再把請求交給它的 `fetch` 事件處理。啟動時間在低階手機上可能達數百毫秒（第七章會談 navigation preload 如何緩解）。

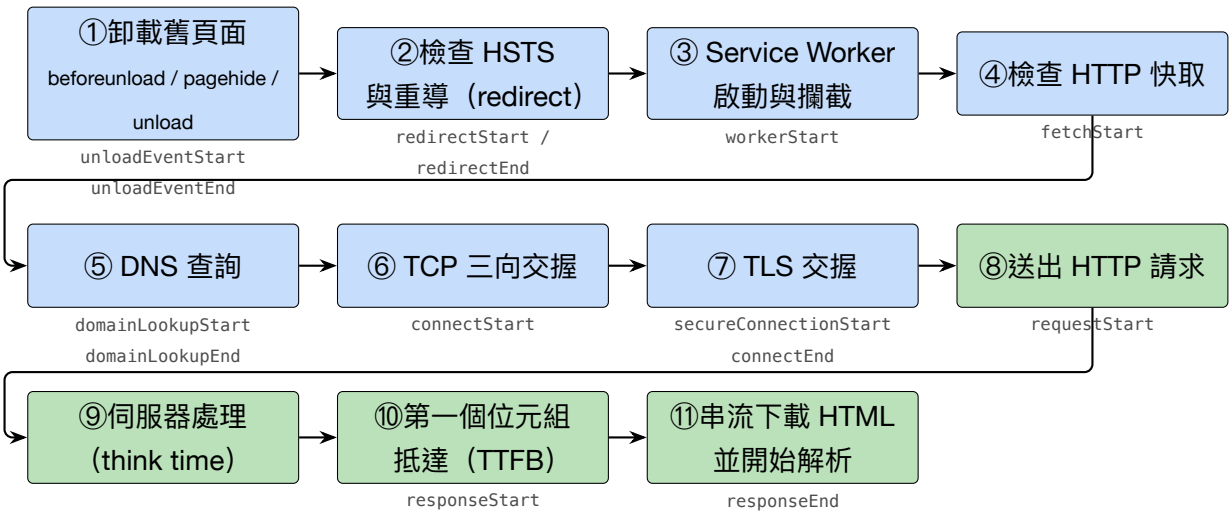


圖 1.1: 導覽流程與 Navigation Timing 時間點對照。①–⑦全部發生在「fetch 之前」，藍色為連線建立階段，綠色為請求與回應階段。

4. **HTTP 快取檢查**：若快取中有仍然新鮮（fresh）的回應，瀏覽器直接使用，完全不碰網路；此時 DNS/TCP/TLS 各時間點會等於 `fetchStart`，`transferSize` 為 0。
5. **DNS 查詢**：把主機名稱換成 IP 位址。
6. **TCP 三向交握**：建立可靠傳輸通道（HTTP/3 改用 QUIC，沒有這一步）。
7. **TLS 交握**：協商加密參數、驗證憑證。

⑧–⑪ 這之後才是大家熟悉的「送請求、等回應」。第一個位元組抵達的時間就是 **TTFB**。

重點

「fetch 之前」的所有步驟都是純粹的**等待**：畫面上沒有任何東西可以畫。這段時間直接墊高 TTFB，進而墊高 FCP 與 LCP。要優化它，關鍵是理解每個步驟需要幾個 RTT，以及如何把這些往返「提前做」或「省下來」。

1.2　RTT：網路效能的基本單位

RTT（Round-Trip Time，往返時間）

一個封包從客戶端送到伺服器、再由伺服器回到客戶端所需的時間。它主要由**物理距離**（光在光纖中的速度約為每毫秒 200 公里）、**路由跳數**與**網路壅塞 / 排隊**決定，與頻寬（bandwidth）幾乎無關。

幾個常見的量級（僅供直覺參考）：

情境	典型 RTT
同一城市內的有線寬頻到附近 CDN 節點	5–20 ms
台灣 ↔ 日本 / 新加坡	30–60 ms
台灣 ↔ 美國西岸	120–160 ms
4G 行動網路（訊號良好）	40–100 ms
訊號不佳的行動網路	200–500 ms 以上

生活比喻

把 RTT 想成「寫信給遠方朋友、等他回信」的時間。建立連線就像雙方在正式談事情之前要先互寄幾封信確認身分、交換密碼本。信寄得再厚（頻寬再大）也沒用，重點是要**來回幾趟**。

1.2.1　建立一條新連線要幾個 RTT？

表 1.1 比較不同協定組合下，從開始建立連線到收到**第一個 HTTP 回應位元組**所需的往返次數（不含 DNS，假設伺服器處理時間為零）。

協定組合	傳輸層	加密層	HTTP 請求/回應	合計
HTTP/1.1 或 HTTP/2，TCP + TLS 1.2（新連線）	1	2	1	**4 RTT**
HTTP/1.1 或 HTTP/2，TCP + TLS 1.3（新連線）	1	1	1	**3 RTT**
TCP + TLS 1.3 0-RTT 恢復（回訪）	1	0（資料隨第一封加密訊息送出）	1	**2 RTT**
HTTP/3，QUIC（新連線，TLS 1.3 內建）	1（傳輸與加密合併）		1	**2 RTT**
HTTP/3，QUIC 0-RTT 恢復（回訪）	0		1	**1 RTT**

表 1.1: 不同協定下，新連線到第一個 HTTP 回應位元組所需的 RTT 數

以 RTT = 100 ms 的行動網路為例：即使伺服器瞬間回應，TLS 1.2 的路徑也要 **400 ms** 才看到第一個位元組；改用 HTTP/3 則是 200 ms，回訪時更可縮到 100 ms。這就是為什麼「升級協定」本身就是一項效能優化。

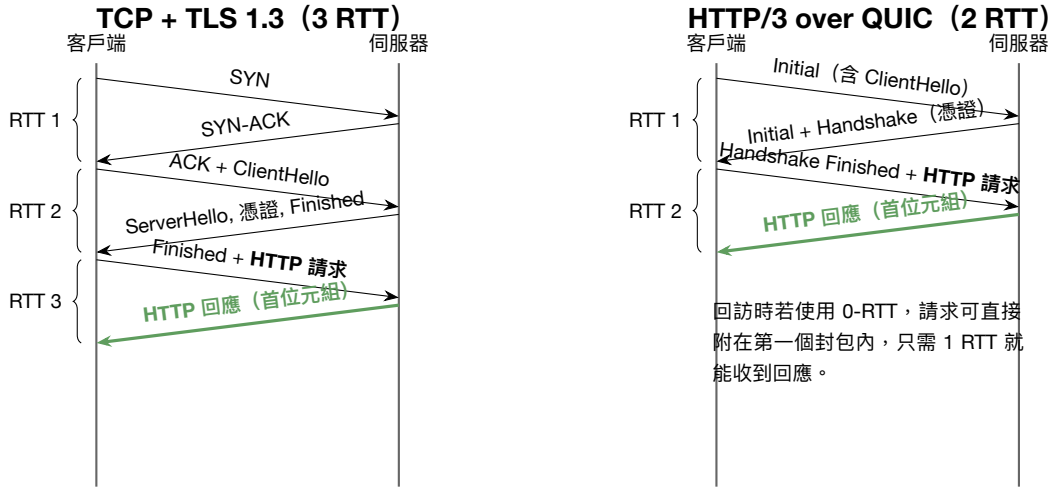


圖 1.2: 新連線的交握順序：TCP + TLS 1.3 與 QUIC 的比較

1.3　各環節可能慢在哪裡、如何優化

1.3.1　重導（Redirect）

慢的原因：每次重導都至少多一個 RTT；跨網域的重導（例如 `example.com` → `www.example.com`，或 `http` → `https`）還要重新做 DNS、TCP、TLS。行銷連結常見的「追蹤跳轉」可能連跳三、四次。

優化：

- 在所有內部連結、sitemap、廣告投放中**直接使用最終 URL**（含 `https`、正確的 www 與結尾斜線）。
- 設定 HSTS（`Strict-Transport-Security`）並申請加入瀏覽器的 HSTS preload 清單，讓瀏覽器連第一次造訪都直接走 HTTPS。
- 用 `PerformanceNavigationTiming.redirectCount` 與 `redirectEnd - redirectStart` 監控真實使用者遇到的重導成本。

1.3.2　DNS 查詢

慢的原因：DNS 查詢有多層快取（瀏覽器 → 作業系統 → 遞迴解析器 → 權威伺服器）。全部未命中時需要多次往返，web.dev 的估計是常見約 20–120 ms。頁面引用越多不同網域（第三方腳本、字型、圖片 CDN、廣告），就要做越多次 DNS。

優化：

- 減少不同來源（origin）的數量；能放同源就放同源。
- 對確定會用到、但不是最關鍵的第三方來源使用 `<link rel="dns-prefetch" href="https://analytics.example">`。
- 選擇回應快、節點多的 DNS 供應商；設定合理的 TTL（太短會讓解析器頻繁重查）。
- 發布 HTTPS DNS 記錄（SVCB/HTTPS RR），可讓支援的瀏覽器在第一次連線就知道伺服器支援 HTTP/3，省去以 `Alt-Svc` 標頭發現 h3 的那一輪。

1.3.3　TCP 交握與慢啟動

慢的原因：三向交握固定要 1 RTT。之後 TCP 的**慢啟動**（slow start）機制會限制初始可送出的資料量（初始壅塞視窗 initcwnd 常見為 10 個 MSS，約 14 KB）；超過這個大小的回應需要等下一個 RTT 才能繼續送。這也是「關鍵 CSS 內嵌並控制在約 14 KB 內」這類建議的來源。

優化：

- 讓伺服器離使用者更近：這正是 CDN 的核心價值（第八章）。
- 重用連線：HTTP/1.1 keep-alive、HTTP/2 單一連線多路複用（multiplexing），避免對同一來源開多條連線各付一次交握成本。
- 對最關鍵的跨來源使用 `preconnect` 提前建立連線（見下文）。

1.3.4　TLS 交握

慢的原因：TLS 1.2 需要 2 RTT；憑證鏈太長（多個中繼憑證）會讓 ServerHello 那一封變大而跨越多個封包；客戶端若需自行向 CA 查 OCSP 狀態，又是一次額外連線。

優化：

- 啟用 **TLS 1.3**（1 RTT，且是 0-RTT 恢復的前提）。
- 啟用 **OCSP stapling**，由伺服器把憑證狀態一併送出，省去客戶端查詢。
- 縮短憑證鏈、使用 ECDSA 憑證（比 RSA 小且快）。
- 啟用 session resumption；若使用 0-RTT，注意重放攻擊風險，只允許冪等的 GET 走 early data。
- 升級到 **HTTP/3**：QUIC 把傳輸與 TLS 1.3 交握合併成 1 RTT，並解決 TCP 層的隊頭阻塞（head-of-line blocking），在封包遺失率高的行動網路上優勢明顯。

1.3.5　Service Worker 啟動

慢的原因：若導覽被 Service Worker 控制，瀏覽器必須先啟動 worker、執行其腳本，再進入 `fetch` 事件處理。對於必須走網路的 HTML（無法預先快取的動態頁面），這段啟動時間是純粹的額外延遲。

優化：使用 **Navigation Preload**（導覽預載）讓瀏覽器在啟動 worker 的同時就先發出網路請求；保持 SW 腳本精簡；避免在啟動階段做重工。細節見第七章。

1.3.6　伺服器處理時間（think time）

嚴格說這已經在 fetch 之後，但它是 TTFB 最常見的元兇：資料庫查詢、未快取的 SSR、冷啟動的無伺服器函式、串接第三方 API。優化手段橫跨第六章（快取 HTML 與 API 回應）、第八章（邊緣快取、103 Early Hints）與第九章（串流 SSR、靜態產生）。

1.4　把往返「提前做」：資源提示與 Early Hints

1.4.1　dns-prefetch 與 preconnect

```
<head>
  <!-- 最關鍵的跨來源（例如放 LCP 圖片的 CDN）：DNS + TCP + TLS 一次做完 -->
  <link rel="preconnect" href="https://img.example-cdn.com">
  <!-- 若該來源會以 CORS 模式載入（字型、fetch()），需加 crossorigin，
       否則瀏覽器會另開一條匿名連線，preconnect 等於白做 -->
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <!-- 次要的第三方：只先做 DNS -->
  <link rel="dns-prefetch" href="https://www.googletagmanager.com">
</head>
```

preconnect 的使用原則

- 只對**確定會用到、且很快就會用到**的來源使用，一般建議 3–5 個以內。每條預先建立的連線都消耗 CPU、記憶體與頻寬（TLS 憑證下載）；Chrome 對閒置約 10 秒未使用的預連線會直接關閉，等於浪費。
- 對於 `preload`（預先載入某個特定資源）與 `preconnect`（只建立連線）要分清楚：知道確切 URL 時用前者，只知道來源時用後者。
- 資源提示只是「提示」；瀏覽器可以忽略。永遠以量測結果為準。

1.4.2　HTTP 103 Early Hints

當伺服器需要幾百毫秒產生 HTML（例如查資料庫、未命中快取的 SSR）時，連線其實是閒置的。**103 Early Hints**（RFC 8297）允許伺服器在最終回應之前，先送出一個暫時回應，夾帶 `Link` 標頭告訴瀏覽器「你等一下一定會需要這些東西，先去連線 / 下載」。

```
HTTP/2 103 Early Hints
Link: </assets/app.css>; rel=preload; as=style
Link: </assets/hero.avif>; rel=preload; as=image; fetchpriority=high
Link: <https://img.example-cdn.com>; rel=preconnect

HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
...
```

要點：

- 收益上限就是伺服器的 think time：如果 HTML 已從 CDN 邊緣快取以 10 ms 回應，Early Hints 幾乎沒有幫助；若 TTFB 有 300–500 ms 的伺服器處理時間，它能讓 LCP 圖片與 CSS 提早數百毫秒開始下載。
- 從瀏覽器到發出 103 的那一跳，整條路徑都必須是 HTTP/2 或 HTTP/3；中間任何一段 HTTP/1.1 或會緩衝的反向代理都會讓 103 消失。
- 目前 Chromium 系瀏覽器與 Firefox 支援，Safari 不處理 103（會忽略，無害）。
- 主流 CDN（Cloudflare、Fastly、CloudFront 等）多半可在邊緣代為產生 103。

1.5　連線建立之後：關鍵渲染路徑速覽

雖然本章主題是「fetch 之前」，但要理解第二章的 FP、FCP、LCP，必須知道第一個 HTML 位元組抵達後瀏覽器做了什麼。這條路徑稱為**關鍵渲染路徑**（Critical Rendering Path）：

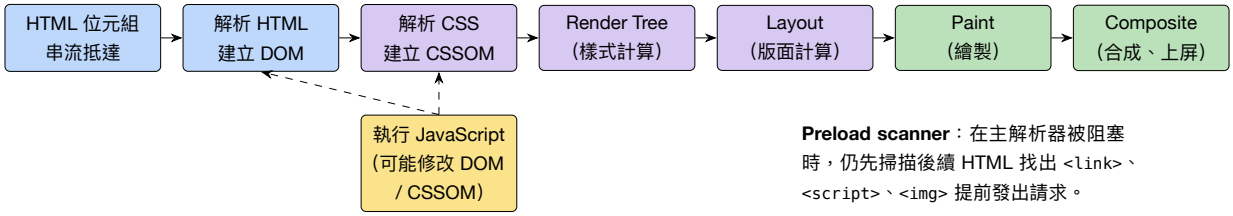


圖 1.3: 關鍵渲染路徑

幾個直接影響 FP / FCP / LCP 的事實：

- **HTML 是串流解析的**：第一個 chunk 到達就開始建立 DOM，不必等整份文件下載完。這是串流 SSR（第九章）能提早 FCP 的基礎。
- **CSS 是 render-blocking**：在 `<head>` 中的外部樣式表下載並解析完成之前，瀏覽器不會繪製任何東西（因為任何一條規則都可能改變畫面）。因此 FP / FCP 至少要等到關鍵 CSS 到齊。
- **同步 `<script>` 是 parser-blocking**：解析器遇到沒有 `defer` / `async` 的腳本會停下來等它下載並執行（若前面還有未完成的 CSS，還得等 CSS，因為腳本可能讀取樣式）。這會延後後續 DOM 的建立與其中圖片的發現。
- **Preload scanner 只看得到 HTML 原始碼裡的資源**：由 JavaScript 動態插入、或藏在 CSS `background-image` 裡的圖片，要等到腳本執行或 CSSOM 建好才會被發現。這正是 LCP「資源載入延遲」（第三章）的主要來源。
- **Layout 與 Paint 的成本與 DOM 大小、CSS 複雜度成正比**：大量節點、深層巢狀、複雜選擇器會讓每次樣式重算與版面計算變慢，也會拉長互動後的呈現延遲（INP 的 presentation delay）。

1.6　課堂實作

實作 1-1：用 Navigation Timing 拆解 TTFB

在任何網站的 DevTools Console 貼上下面程式碼，觀察各階段的耗時。試著比較「第一次造訪」與「重新整理」的差異（DNS、TCP、TLS 是否變成 0？`transferSize` 是否為 0？）。

```
const nav = performance.getEntriesByType('navigation')[0];
const ms = (a, b) => Math.round(b - a);
console.table({
  '重導 redirect':        ms(nav.redirectStart, nav.redirectEnd),
  'SW 啟動 worker':       nav.workerStart ? ms(nav.workerStart, nav.fetchStart)
    : 0,
  'DNS':                  ms(nav.domainLookupStart, nav.domainLookupEnd),
  'TCP':                  ms(nav.connectStart, nav.secureConnectionStart || nav
    .connectEnd),
  'TLS':                  nav.secureConnectionStart ? ms(nav.
    secureConnectionStart, nav.connectEnd) : 0,
  '請求→首位元組':          ms(nav.requestStart, nav.responseStart),
  'TTFB（自導覽開始）':     Math.round(nav.responseStart),
  '回應下載':               ms(nav.responseStart, nav.responseEnd),
  '協定 nextHopProtocol':  nav.nextHopProtocol,   // h2 / h3 / http/1.1
  '重導次數':               nav.redirectCount,
```

```
  '傳輸大小 transferSize': nav.transferSize,      // 0 代表來自快取
});
```

實作 1-2：DevTools Network 面板的 Timing 分頁

1. 開啟 DevTools → Network，勾選「Disable cache」，重新整理頁面。
2. 點選第一個請求（HTML 文件），切到 **Timing** 分頁，觀察：Queueing、Stalled、DNS Lookup、Initial connection、SSL、Request sent、**Waiting for server response（TTFB）**、Content Download。
3. 在欄位標題按右鍵開啟 **Protocol** 欄，看看哪些資源走 h2、哪些走 h3、哪些仍是 http/1.1。
4. 用 Network throttling 切到「Slow 4G」，再看一次各階段的變化：哪些階段被放大得最多？

實作 1-3：用 curl 從終端機量測

```
curl -o /dev/null -s -w \
'DNS %{time_namelookup}s | TCP %{time_connect}s | TLS %{time_appconnect}s |
   TTFB %{time_starttransfer}s | 總計 %{time_total}s | 重導 %{num_redirects} 次
   \n' \
https://www.example.com/
# 加上 --http3 可測 HTTP/3（需 curl 支援），加上 -L 會跟隨重導
```

注意 curl 顯示的是累計時間（`time_appconnect` 包含 DNS 與 TCP），需自行相減。

思考題

1. 某網站的行動端使用者 RTT 約 120 ms，伺服器使用 TLS 1.2、HTTP/2，且首頁 `http://example.com` 會 301 到 `https://www.example.com`。請估算「按下 Enter 到第一個 HTML 位元組」至少要多少毫秒（不含 DNS 與伺服器處理）。若改為 HSTS preload + TLS 1.3 + HTTP/3，可以省下多少？
2. 為什麼對載入字型的來源做 `preconnect` 時必須加上 `crossorigin`？如果忘了加會發生什麼事？
3. 一個頁面的 HTML 由 CDN 邊緣快取在 15 ms 內回應，工程師提議加上 103 Early Hints 來加速 LCP 圖片。你會怎麼評估這個提案？
4. 為什麼「把 LCP 圖片改成 CSS `background-image`」常常會讓 LCP 變慢？這和 preload scanner 有什麼關係？

本章回顧

- 一次導覽在真正送出請求前要經過：卸載舊頁 → HSTS / 重導 → Service Worker 啟動 → HTTP 快取檢查 → DNS → TCP → TLS。這些全部是等待時間，直接墊高 TTFB。
- RTT 由距離與路由決定，是連線成本的基本單位。新連線成本：TLS 1.2 為 4 RTT、TLS 1.3 為 3 RTT、QUIC 為 2 RTT、0-RTT 恢復可到 1 RTT。
- 優化手法分三類：**省掉往返**（移除重導、HSTS、快取、連線重用）、**縮短每次往返**（CDN、就近節點）、**提前做往返**（dns-prefetch、preconnect、103 Early Hints、Speculation Rules）。
- 第一個位元組抵達後進入關鍵渲染路徑：HTML 串流解析、CSS 阻塞渲染、同步腳本阻塞解析、preload scanner 只認得 HTML 原始碼中的資源。

第二章　效能量化指標總覽：TTFB、FP、FCP、FMP、LCP、FID、INP、CLS

學習目標

- 精確說出每個指標**量的是什麼**、**從哪裡量到哪裡**、**門檻是多少**。
- 知道哪些指標已經被淘汰（FMP、FID）、被什麼取代、以及為什麼。
- 分辨 Core Web Vitals 與「診斷用指標」，並知道哪些只能在真實使用者環境量測。
- 能在自己的頁面上把這些指標印出來。

2.1　把指標放到時間線上

圖 2.1 將本章所有指標放在同一條頁面生命週期時間線上。左半部（TTFB、FP、FCP、LCP）是「載入」階段的**時間點**；INP 是整段互動期間**所有互動延遲的彙整**；CLS 則是整個生命週期中**版面跳動的累計**。

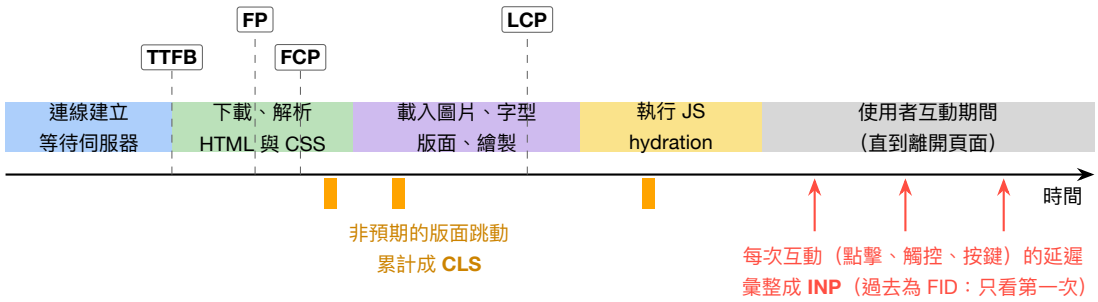


圖 2.1: 各指標在頁面生命週期中的位置

2.2　TTFB：Time to First Byte

定義

從**導覽開始**（使用者按下 Enter 或點擊連結）到**收到 HTML 回應的第一個位元組**的時間。依 web.dev 的定義，它包含：重導時間、Service Worker 啟動時間、DNS 查詢、TCP 與 TLS 連線建立、送出請求到首位元組抵達之間的等待（伺服器處理時間）。

- **門檻**：良好 ≤ 800 ms；待改善 800–1800 ms；不佳 > 1800 ms（以 p75 計）。
- **性質**：**不是** Core Web Vitals，而是最重要的**診斷指標**。TTFB 是 FCP 與 LCP 的下限：HTML 還沒到，什麼都畫不出來。
- **對 SPA 尤其關鍵**：純客戶端渲染的頁面拿到 HTML 後還要下載 JS、執行、抓資料才有內容，所以 TTFB 每慢 1 ms，後面全部順延。反過來，伺服器端渲染的頁面 TTFB 可能較高（伺服器要組 HTML），但 FCP / LCP 反而更早——這就是為什麼 web.dev 說 TTFB 門檻是「粗略指引」，要和你的架構一起看。
- **量測**：所有主流瀏覽器都支援，取 `PerformanceNavigationTiming.responseStart`（第四章）。第一章已把 TTFB 拆解到 RTT 層級。

2.3　FP：First Paint

定義

瀏覽器**第一次在畫面上渲染任何東西**的時間，排除預設的白色背景，但包含非預設的背景色與 iframe 的外框。由 Paint Timing API 以 `first-paint` 這個 entry 提供。

FP 回答的是「有在動嗎？」的最低門檻：使用者看到畫面從空白變成了「有東西」。規範將回報 FP 列為**選擇性**（Safari 就不回報 FP，只回報 FCP），且它沒有官方的良好 / 不佳門檻，主要用於診斷。在大多數頁面上 FP 與 FCP 幾乎同時，若兩者差距大，通常表示頁面先畫了一個有色背景或骨架，但真正的內容還在等（例如等待網頁字型或 JS）。

2.4　FCP：First Contentful Paint

定義

從導覽開始到瀏覽器**第一次渲染出任何 DOM 內容**的時間。「內容」包括：文字（含仍在等待網頁字型的文字**不算**，需實際畫出）、`<img>`、`<svg>`、非白色的 `<canvas>`、CSS 背景圖片。**iframe 內的內容不計入**。

- **門檻**：良好 ≤ 1.8 s；待改善 1.8–3.0 s；不佳 > 3.0 s。

- **性質**：不是 Core Web Vitals，但是 Lighthouse 效能分數的組成之一（權重 10%），也是 CrUX 中回報的指標。
- **常見瓶頸**：TTFB 高；`<head>` 內的 render-blocking CSS 與同步 JS；網頁字型以 `font-display: block`（FOIT）方式載入導致文字遲遲不畫；SPA 要等 JS 執行才有內容。
- **量測**：Paint Timing API 的 `first-contentful-paint` entry；Chromium、Firefox、Safari 都支援。

2.5　FMP：First Meaningful Paint（已淘汰）

定義

Chrome 早期提出的啟發式指標，定義為「視窗上方（above-the-fold）最大的版面變化發生之後、且網頁字型已載入」的那次繪製，試圖回答「有用了嗎？」。

FMP 的問題在於它依賴 Chrome 渲染引擎的內部實作（統計 layout object 的數量變化），對頁面載入的微小差異極度敏感，常出現雙峰（bimodal）分布，且無法標準化、其他瀏覽器無法實作。Lighthouse 6.0（2020 年 5 月）將它棄用，並由 **LCP** 接替「主要內容何時出現」這個問題；後續版本已完全移除。

閱讀較早出版的書籍時

許多 2020 年前後撰寫的效能書籍仍花不少篇幅介紹 FMP、TTI 與 FID。閱讀時請自動把「FMP」對應到今天的「LCP」、「FID」對應到「INP」，並記得 TTI 也已在 Lighthouse 10 移除。這些指標背後的「問題」沒有變，變的是回答問題的量測方法。

2.6　LCP：Largest Contentful Paint

定義

從導覽開始到**視窗內最大的圖片、文字區塊或影片**完成渲染的時間。它是「主要內容是否已呈現」的代理指標，屬於 Core Web Vitals。

哪些元素會被視為 LCP 候選？

依 Largest Contentful Paint API 規範與 web.dev 的整理：

- `<img>` 元素（動畫圖片如 GIF 以**第一幀**呈現時間計）。
- `<svg>` 內的 `<image>` 元素。
- `<video>` 元素：取 poster 圖片載入時間或影片第一幀呈現時間，**較早者**為準。
- 以 CSS `background-image: url()` 載入背景圖的元素（CSS 漸層不算）。

- 包含文字節點或行內文字子元素的**區塊級（block-level）**元素。

排除規則（Chromium）

- `opacity: 0` 的元素（使用者看不到）。
- 覆蓋整個視窗的元素（通常是背景而非內容）。
- **低資訊熵的圖片**：自 Chrome 112 起，圖片的「編碼位元數 ÷ 顯示像素數」低於 **0.05 bits/pixel** 者不列入，用來排除純色佔位圖、極度模糊的 LQIP 與大面積漸層圖，避免有人用超小的假圖「作弊」。
- 文字元素只計算包住所有文字節點的**最小矩形**；所有元素都不計 margin、padding、border。
- 元素的大小以**視窗內可見部分**計；超出視窗或被裁切的部分不算。圖片若被放大顯示，以**原始尺寸與顯示尺寸的較小者**計。

LCP 如何被「決定」？

瀏覽器在載入過程中會持續回報新的、更大的候選元素（每個都是一筆 `largest-contentful-paint` entry）；**最後一筆**就是 LCP。一旦使用者與頁面互動（點擊、觸控、按鍵、捲動），瀏覽器就停止回報，因為互動之後的內容變化不再算「載入」。因此在 RUM 中要在頁面隱藏（`visibilitychange`）或互動時才送出最終值。

- **門檻**：**良好** ≤ 2.5 s；**待改善** 2.5–4.0 s；**不佳** > 4.0 s。
- **支援**：Chromium 系瀏覽器；Firefox 自 122 版起支援 LCP API；Safari 不支援。
- 第三章會把 LCP 拆成四個子部分（TTFB、資源載入延遲、資源載入時間、元素渲染延遲）逐一優化。

2.7　FID：First Input Delay（已由 INP 取代）

定義

使用者**第一次**與頁面進行離散互動（點擊、觸控、按鍵；不含捲動與縮放）時，從輸入發生到瀏覽器**開始執行**事件處理器之間的延遲。它只量「輸入延遲」，不含事件處理時間，也不含後續渲染。

- **門檻**（歷史）：**良好** ≤ 100 ms；**不佳** > 300 ms。
- **為什麼被取代**：(1) 只看第一次互動，但使用者在頁面上的大部分互動發生在載入完成後；(2) 只量延遲那一小段，事件處理器本身跑了 500 ms、畫面更新又拖了 200 ms 都不算；(3) 到 2023 年超過九成網站都「通過」FID，指標已失去鑑別力。

- Google 於 **2024 年 3 月 12 日**正式以 INP 取代 FID 成為 Core Web Vitals；`web-vitals` 套件 v5 已移除 `onFID`。

2.8　INP：Interaction to Next Paint

定義

觀察頁面整個生命週期中**所有**點擊、觸控與鍵盤互動的延遲，回報其中**接近最差**的一個值（互動次數少於 50 次時取最大值；之後每 50 次互動忽略一個最高值，約等於第 98 百分位）。單次互動的延遲是從使用者輸入開始，到瀏覽器**畫出下一個包含互動結果的畫格**為止。

什麼算一次「互動」？

一次互動是由同一個使用者動作觸發的一組事件：

- 滑鼠點擊：`pointerdown` → `pointerup` → `click`。
- 觸控：`pointerdown` → `pointerup`（以及可能的 `click`）。
- 鍵盤：`keydown` → `keyup`（輸入文字時每個按鍵都是一次互動）。

同一次互動裡多個事件的延遲**取最長者**，不是相加。**hover 與捲動不算互動**（捲動由瀏覽器的合成執行緒處理，通常不受主執行緒阻塞影響）。

三個子部分

每次互動的延遲 = **輸入延遲**（input delay，事件發生到處理器開始執行）+ **處理時間**（processing duration，所有事件處理器執行完畢）+ **呈現延遲**（presentation delay，處理完到下一個畫格上屏）。第三章會針對這三段分別討論優化。

- **門檻**：良好 ≤ 200 ms；待改善 200–500 ms；不佳 > 500 ms。
- **性質**：Core Web Vitals；**只能在真實使用者環境量測**（實驗室沒有真人互動）。實驗室以 TBT 作為代理，DevTools 的 Performance 面板則可在你手動互動時即時顯示每次互動的延遲。
- **支援**：以 Event Timing API 為基礎，目前為 Chromium 專有。
- 目前 INP 是三個 Core Web Vitals 中**最多網站無法通過**的一項，主因是龐大的 JavaScript、第三方腳本與 hydration 造成的長任務。

2.9　CLS：Cumulative Layout Shift

定義

衡量頁面生命週期中**非預期版面跳動**的嚴重程度。每次跳動有一個「版面偏移分數」（layout shift score）；CLS 取**最大的一段「跳動叢集」（session window）**的分數總和。

單次版面偏移分數

$$\text{layout shift score} = \text{impact fraction} \times \text{distance fraction}$$

- **影響比例（impact fraction）**：前後兩個畫格中，所有「不穩定元素」佔據區域的聯集面積 ÷ 視窗面積。例如一個佔視窗 50% 高度的元素往下移了 25% 視窗高度，前後兩個位置的聯集是 75% $\Rightarrow$ 0.75。
- **距離比例（distance fraction）**：不穩定元素在該畫格中移動的最大距離（水平或垂直）÷ 視窗的**較長邊**。上例為 0.25。
- 該次分數 $= 0.75 \times 0.25 = 0.1875$。

Session window 與排除規則

- 連續的跳動若彼此間隔**少於 1 秒**，會被歸入同一個 session window；單一 window 最長 **5 秒**。CLS = 所有 window 中**分數總和最大**者。這是 2021 年的修訂，目的是不讓長時間開著的頁面（如無限捲動）分數無限累積。
- **使用者輸入後 500 ms 內**的跳動被視為「預期中的」（`hadRecentInput = true`），不計入 CLS。但注意：只有離散輸入（點擊、觸控、按鍵）算；捲動、拖曳、縮放**不算**最近輸入。
- 只有**起始位置改變**才算跳動；新元素出現在原本空白處、或元素只改變尺寸但起點不動，都不算。以 `transform` 做的動畫也不算（它不影響版面）。
- **門檻**：**良好** ≤ 0.1；**待改善** 0.1–0.25；**不佳** > 0.25（無單位）。
- **性質**：Core Web Vitals；實驗室與真實使用者皆可量，但實驗室通常只量到載入階段，真實使用者才會量到之後的互動與延遲載入內容。
- **支援**：Layout Instability API，Chromium 專有。

2.10　補充：Lighthouse 中的其他實驗室指標

指標	定義	狀態
Speed Index (SI)	以逐幀截圖分析視覺內容「填滿」的速度，越早填滿越低。	Lighthouse 分數組成，權重 10%。
Total Blocking Time（TBT）	FCP 到 TTI 之間，所有長任務（>50 ms）超過 50 ms 部分的總和。	Lighthouse 分數組成，權重 30%；INP 的實驗室代理。
Time to Interactive（TTI）	頁面「可靠地」可互動的時間點（FCP 後主執行緒安靜 5 秒）。	Lighthouse 10 已移除，因與使用者感受相關性低。
Max Potential FID	最長任務的長度，作為 FID 上限估計。	Lighthouse 6.0 已移除。

Lighthouse 10 之後的效能分數權重為：FCP 10%、SI 10%、LCP 25%、TBT 30%、CLS 25%。請記得這是實驗室分數，與 CrUX 的真實使用者評分是兩套系統。

2.11　總整理

指標	量什麼	良好	不佳	CWV?	Lab / Field	來源 API
TTFB	導覽開始到 HTML 首位元組	≤ 0.8 s	> 1.8 s	否（診斷）	兩者	Navigation Timing
FP	第一次畫出任何非預設像素	–	–	否	兩者	Paint Timing
FCP	第一次畫出 DOM 內容	≤ 1.8 s	> 3.0 s	否	兩者	Paint Timing
FMP	最大版面變化後的繪製	–	–	已淘汰	Lab	Chrome 內部啟發式
LCP	視窗內最大內容元素完成渲染	≤ 2.5 s	> 4.0 s	**是**	兩者	LCP API
FID	第一次互動的輸入延遲	≤ 100 ms	> 300 ms	已被 INP 取代	Field	Event Timing
INP	所有互動延遲的近最差值	≤ 200 ms	> 500 ms	**是**	Field	Event Timing
CLS	最大跳動叢集的分數總和	≤ 0.1	> 0.25	**是**	兩者	Layout Instability
TBT	長任務阻塞時間總和	< 200 ms	> 600 ms	否	Lab	Long Tasks

表 2.1: 指標總表。門檻皆為 web.dev 公布之 p75 標準；TBT 門檻為 Lighthouse 評分區間。

門檻是怎麼定出來的？

web.dev 說明 Core Web Vitals 門檻的訂定原則有兩個：(1) **符合人類感知研究**——例如 100 ms 內的回饋會被感知為「即時」，1 秒內的變化不會打斷思路；(2) **可達成性**——「良好」門檻必須是當時約前 10% 的網站已能達到的水準，「不佳」門檻則對應大約後 30% 的網站。這些門檻自 2020 年公布、2024 年 INP 加入後至今（2026 年）未再調整。

2.12　課堂實作

實作 2-1：在 Console 印出載入指標

不使用任何套件，直接用 PerformanceObserver（第四章會詳細解釋）：

```
// TTFB
const nav = performance.getEntriesByType('navigation')[0];
console.log('TTFB', Math.round(nav.responseStart), 'ms');

// FP / FCP
new PerformanceObserver((list) => {
  for (const e of list.getEntries()) console.log(e.name, Math.round(e.startTime
    ), 'ms');
}).observe({ type: 'paint', buffered: true });

// LCP：持續回報候選，最後一筆才是最終值
new PerformanceObserver((list) => {
  const last = list.getEntries().at(-1);
  console.log('LCP 候選', Math.round(last.startTime), 'ms', last.element, last.
    url);
}).observe({ type: 'largest-contentful-paint', buffered: true });

// CLS：只累加沒有最近輸入的跳動（此處為簡化版，未實作 session window）
let cls = 0;
new PerformanceObserver((list) => {
  for (const e of list.getEntries()) if (!e.hadRecentInput) cls += e.value;
  console.log('CLS 累計', cls.toFixed(4));
}).observe({ type: 'layout-shift', buffered: true });
```

實作 2-2：用 PageSpeed Insights 比較實驗室與真實數據

1. 開啟 PageSpeed Insights，輸入一個流量足夠的網站（例如新聞或電商網站）。
2. 上半部「探索實際使用者的體驗」是 CrUX 的 28 天 p75 數據，注意 LCP、INP、CLS 三項是否都為綠色。
3. 下半部是 Lighthouse 實驗室分數。找出兩者不一致的地方（例如實驗室 LCP 很好但真實 LCP 不佳），討論可能原因：真實使用者的裝置與網路分布？快取狀態？地理位置？
4. 切換「行動裝置 / 電腦」，觀察差異。Google 排名主要看行動版數據。

思考題

1. 一個頁面的 TTFB 是 300 ms、FCP 是 2.9 s，中間差了 2.6 s。請列出至少三種可能造成這段差距的原因。
2. 某電商首頁的 LCP 元素在 Chrome 中被判定為一個 40 KB 的橫幅圖片，但設計師把它換成一張 3 KB 的極度模糊漸層圖後 LCP 從 3.1 s 變成 1.2 s。這算改善嗎？Chrome 112 之後會怎麼處理？
3. 為什麼 INP 要「每 50 次互動忽略一個最高值」而不是直接取最大值？
4. 一個使用無限捲動的社群動態頁面，在 CLS 改為 session window 計算方式之前與之後，分數會有什麼不同？為什麼舊算法對這類頁面不公平？
5. 為什麼 FID 幾乎每個網站都「通過」，卻仍有大量網站的 INP 不佳？兩者定義上的哪個差異造成這個現象？

本章回顧

- 載入階段的時間點指標依序為 TTFB → FP → FCP → LCP，每一個都以前一個為下限。
- Core Web Vitals 目前為 **LCP（≤2.5 s）**、**INP（≤200 ms）**、**CLS（≤0.1）**，以真實使用者 28 天 p75 評分，三項全達良好才算通過。
- FMP 與 FID 已淘汰；讀舊書時請對應到 LCP 與 INP。TTI 也已從 Lighthouse 移除。
- INP 與 FID 只能在真實使用者環境量測；TBT 是實驗室代理。LCP、CLS、INP 的 API 主要是 Chromium 專有，TTFB 與 FCP 則跨瀏覽器可用。
- CLS 的公式是 impact fraction × distance fraction，並以 1 秒間隔、5 秒上限的 session window 取最大叢集。

第三章　Web Vitals 深入：LCP、INP、CLS 的診斷與優化

學習目標

- 把 LCP 拆成四個子部分、INP 拆成三個子部分，並能從數據判斷瓶頸落在哪一段。
- 針對每個子部分說出至少兩種具體的優化手法，並知道其背後的瀏覽器機制。
- 理解 CLS 的常見成因，並能用 API 找出「誰在跳」。
- 建立一套「量測 → 定位子部分 → 對症優化 → 再量測」的工作流程。

3.1　LCP：四個子部分

web.dev 建議把 LCP 時間拆成連續的四段（圖 3.1）。任何 LCP 優化都應該先知道「時間花在哪一段」，再對症下藥。

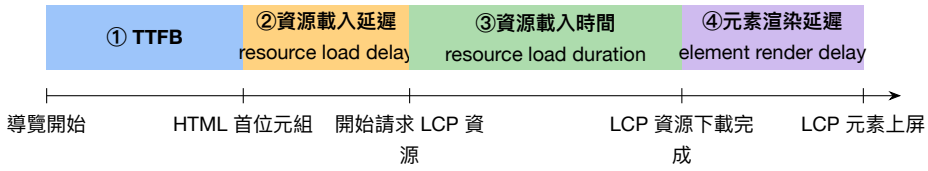


圖 3.1: LCP 的四個子部分。若 LCP 元素是不需下載資源的文字（系統字型），②③為 0。

健康的比例長什麼樣？

理想上，②「資源載入延遲」與④「元素渲染延遲」都應接近 0（各低於 LCP 的 10%），時間幾乎全部花在「真的在等伺服器」（①）與「真的在下載」（③）。但根據 corewebvitals.io 對數百萬次載入的統計，中位數網站的分布約為：TTFB 48%、載入延遲 24%、載入時間 10%、渲染延遲 17%——也就是說，**兩段「純等待」合計比真正下載時間多四倍**。這兩段是最先該找的優化空間。Chrome DevTools 的「LCP by phase」insight 與 CrUX 自 2025 年起提供的 LCP 圖片子部分數據，都能直接看到這個分布（第五章）。

3.1.1　① TTFB

TTFB 高通常指向伺服器、網路或重導問題。這部分已在第一章拆解，並會在第六章（快取 HTML）、第八章（邊緣節點、Early Hints）、第九章（串流 SSR、靜態產生）延伸。這裡只強調一點：**TTFB 是 LCP 的地板**，TTFB 1.5 秒的頁面不可能有 2.5 秒內的 LCP，除非其他三段合計小於 1 秒。

3.1.2　②資源載入延遲：讓瀏覽器「早點知道」

這一段量的是「HTML 已經回來了，但瀏覽器還沒開始下載 LCP 圖片」的空窗。成因幾乎都是**發現太晚**：

- LCP 圖片由 JavaScript 動態插入（典型 SPA / CSR），要等 JS 下載、執行、抓資料之後才出現在 DOM。
- LCP 圖片是 CSS `background-image`，要等 CSS 下載並建好 CSSOM、套用到元素後才會被發現；preload scanner 看不到它。
- LCP 圖片被加了 `loading="lazy"`，瀏覽器故意延後到接近視窗才載入。
- 圖片雖在 HTML 中，但排在大量同步腳本之後，或 preload scanner 被某些結構卡住。
- 圖片以預設的低優先權排隊，被前面一堆低價值資源（第三方腳本、下方圖片）搶走頻寬與連線。

優化手法：

```
<!-- 1. LCP 圖片直接寫在 HTML 中，且不要 lazy-load；用 fetchpriority 提升優先權 -->
<img src="/hero-800.avif"
     srcset="/hero-480.avif 480w, /hero-800.avif 800w, /hero-1200.avif 1200w"
     sizes="(max-width: 600px) 100vw, 800px"
     width="800" height="450"
     alt="本季主打商品"
     fetchpriority="high">

<!-- 2. 若 LCP 圖片只能放在 CSS 背景或由 JS 決定，改用 preload 讓 scanner 提早看到 --
    >
<link rel="preload" as="image" fetchpriority="high"
      imagesrcset="/hero-480.avif 480w, /hero-800.avif 800w, /hero-1200.avif 1200w"
      imagesizes="(max-width: 600px) 100vw, 800px">

<!-- 3. 反面教材：LCP 圖片被 lazy-load，且沒有尺寸 -->
<img src="/hero.jpg" loading="lazy">
```

- **fetchpriority=”high”**：圖片預設是低優先權（Chrome 會在版面計算後把視窗內圖片提升，但那已經晚了）。明確標記讓它在第一輪「關鍵資源階段」就開始下載。只標一

張最可能是 LCP 的圖，標太多等於沒標。

- **preload 只給「晚發現」的資源**：字型、`@import` 進來的 CSS、CSS 背景圖、JS 才會插入的 LCP 圖。已經在 HTML 中的 `<img>` 不需要再 preload。
- 若 LCP 圖片來自跨來源的 CDN，搭配 `preconnect`（第一章）省下連線建立時間。
- 架構層面：對內容型頁面採用 SSR / SSG，讓 LCP 元素直接出現在 HTML 中（第九章）。

3.1.3　③資源載入時間：讓資源「變小、變近」

- **現代格式**：AVIF 通常比 JPEG 小 30–50%，WebP 約 25–35%。用 `<picture>` 提供退路。
- **正確尺寸**：以 `srcset` + `sizes` 讓手機不要下載桌機版 1600 px 的圖；注意 DPR（2x 螢幕需要兩倍像素，但可用較高壓縮率）。
- **壓縮品質**：多數照片在品質 60–75 之間肉眼難以分辨，但檔案差距可達數倍。Image CDN（第八章）可自動處理格式、尺寸與品質。
- **減少競爭**：LCP 圖片下載時，若同時有十幾個第三方腳本與下方圖片在搶頻寬，它就會變慢。對非首屏圖片使用 `loading="lazy"`、延後非關鍵腳本，是在「幫 LCP 讓路」。
- **快取與 CDN**：回訪者直接命中快取(第六章)；首訪者從最近的邊緣節點取得(第八章)。
- **影片作為 LCP**：確保 poster 圖片小而快，或讓自動播放影片的第一幀盡早可用。

3.1.4　④元素渲染延遲：資源到了卻沒畫出來

這一段是「圖片已經下載完成，但元素遲遲沒有上屏」。常見原因：

- **Render-blocking 資源**：`<head>` 內還有 CSS 或同步 JS 沒載完，瀏覽器根本還沒開始第一次繪製。
- **等待 JavaScript 才顯示**：元素一開始 `display: none` 或 `opacity: 0`，靠 JS / 框架 hydration 之後才顯示；A/B 測試工具的「防閃爍」腳本把整頁藏起來幾百毫秒。
- **文字 LCP 等字型**：`font-display: block` 讓文字在字型下載前完全不畫（FOIT）。
- **主執行緒忙碌**：大量 JS 執行（hydration、初始化）佔住主執行緒，導致排好的繪製工作被延後。
- **圖片解碼**：超大圖片的解碼本身也需要時間；`decoding="async"` 可以避免阻塞其他繪製，但對 LCP 圖片通常維持預設即可。

優化手法：內嵌關鍵 CSS 並延後非關鍵 CSS；所有腳本加 `defer` 或 `async`（或用 `type="module"`，預設即 defer）；字型使用 `font-display: swap` 或 `optional` 並 preload 關鍵字型；避免用 JS 控制 LCP 元素的可見性；把非必要的初始化工作拆小、延後（見 INP 一節）。

LCP 常見陷阱清單

- 輪播（carousel）：第一張圖才是 LCP，卻因 JS 初始化才顯示。
- 把 hero 圖改成 CSS 背景「比較好排版」，結果載入延遲多了幾百毫秒。
- 為了 Lighthouse 分數把所有圖都加 `loading="lazy"`，包含首屏。
- 用超模糊 LQIP 佔位，Chrome 112 之後因低熵被忽略，LCP 反而變成後面才載入的大圖。
- 使用 `fetchpriority="high"` 在十張圖上，優先權形同虛設。
- Cookie 同意橫幅或彈窗成為 LCP 元素（覆蓋大面積文字），實際內容反而沒被量到——這時要重新思考元素設計。

3.2　INP：三個子部分

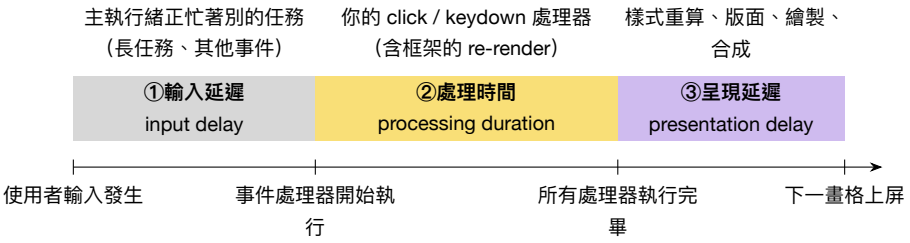


圖 3.2: 一次互動的三個子部分。INP 取的是頁面上（近乎）最慢的那一次互動。

3.2.1　長任務與 50 ms 法則

主執行緒一次只能做一件事。任何超過 **50 ms** 的任務稱為**長任務**（long task）；超過 50 ms 的部分是它的「阻塞時間」。RAIL 模型建議在 50 ms 內處理完使用者輸入，才能在 100 ms 內給出可感知的回饋。INP 的三個子部分幾乎都能追溯到長任務：

- 互動發生時主執行緒正在跑長任務 ⇒ **輸入延遲**變長。
- 事件處理器本身就是長任務 ⇒ **處理時間**變長。
- 處理器改動大量 DOM 導致樣式、版面重算成為長任務 ⇒ **呈現延遲**變長。

3.2.2　①輸入延遲：別讓主執行緒在使用者點下去時正忙

常見成因：頁面載入期間的大量 JS（bundle 解析執行、hydration）、第三方腳本的週期性計時器、多個互動重疊（前一次互動的處理器還沒跑完）。

優化手法：

- 減少載入階段的 JS：code splitting、延後載入非首屏功能、移除未使用程式碼、審視第三方腳本。
- 把長任務拆小，讓瀏覽器有機會在任務之間處理輸入（見下方程式碼）。
- 把純運算（資料處理、排序、加密）搬到 Web Worker。
- 避免在 `setInterval` 中做重工；用 `requestIdleCallback` 或 `scheduler.postTask(priority: 'background')` 安排低優先權工作。

```
// 反例：一次處理 5000 筆資料，形成一個 400 ms 的長任務
function renderAll(items) {
  for (const item of items) renderRow(item);
}

// 改法：每工作約 50 ms 就把主執行緒讓出來（yield），讓瀏覽器處理輸入與繪製
async function renderAllChunked(items) {
  let deadline = performance.now() + 50;
  for (const item of items) {
    renderRow(item);
    if (performance.now() >= deadline) {
      await yieldToMain();
      deadline = performance.now() + 50;
    }
  }
}

// scheduler.yield() 會把「後續程式」排在較高優先權的佇列，
// 不會像 setTimeout(0) 那樣被其他排隊中的任務插隊
function yieldToMain() {
  if (globalThis.scheduler?.yield) return scheduler.yield();
  return new Promise((resolve) => setTimeout(resolve, 0));
}
```

不再建議使用 isInputPending()

早期建議「只有在 `navigator.scheduling.isInputPending()` 為 true 時才 yield」。web.dev 現已不建議：它在某些情況會誤報 false、而且輸入不是唯一需要讓路的對象（動畫與畫面更新同樣重要）。請改為**定期無條件 yield**，並優先使用 `scheduler.yield()`。

3.2.3　②處理時間：只做「這一幀需要的事」

事件處理器本身太重是 INP 最直接的成因。框架的 re-render（React 的 state 更新造成大量元件重繪）、同步的資料處理、在處理器內讀取版面屬性（`offsetHeight`、`getBoundingClientRect`）造成的強制同步版面（forced reflow）都是常客。

優化原則：處理器只做**下一幀畫面需要的最少工作**，其餘延後。

```
button.addEventListener('click', () => {
  // 1. 立刻做視覺回饋（這會進入下一幀）
  button.classList.add('is-loading');
  panel.textContent = '處理中…';

  // 2. 等下一幀畫完之後，再做真正的重工
  requestAnimationFrame(() => {
    setTimeout(() => {
      const result = heavyComputation();   // 或搬到 Worker
      panel.textContent = result;
      button.classList.remove('is-loading');
    }, 0);
  });
});
```

其他手法：對輸入框的 `keydown/input` 事件做 debounce；React 中用 `useTransition / startTransition` 把非緊急更新降級；避免在處理器裡同步呼叫分析工具的大量程式碼。

3.2.4　③呈現延遲：讓渲染變便宜

處理器跑完之後，瀏覽器要重算樣式、版面、繪製、合成。這一段變慢的成因：

- **DOM 太大**：數千個節點的頁面，每次樣式與版面重算都很貴。Lighthouse 對超過約 1,400 個節點會警告。
- **版面抖動（layout thrashing）**：在迴圈中交錯「寫樣式、讀版面屬性」，迫使瀏覽器每次都同步重算版面。
- **複雜的 CSS 選擇器**與大量 `:has()`、深層後代選擇器，讓樣式重算變慢（DevTools 有 CSS selector stats 可診斷）。
- **`requestAnimationFrame` 回呼中做太多事**：rAF 回呼會在繪製前執行，一樣會延後上屏。

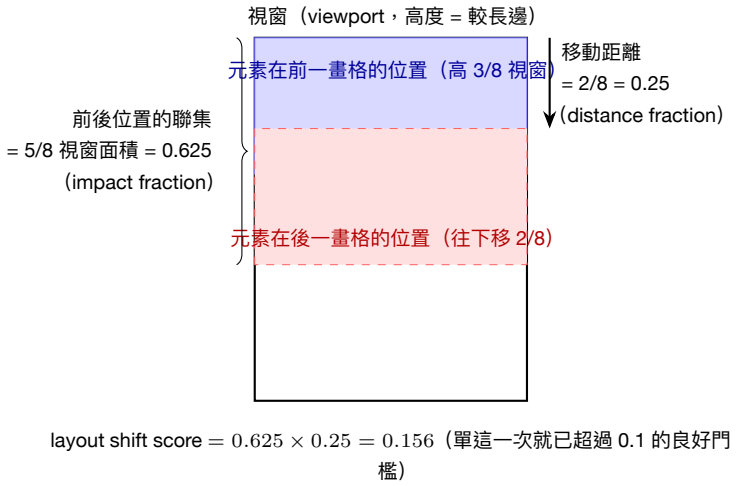
優化手法：用 `content-visibility: auto` 讓視窗外的區塊跳過渲染；用 CSS `contain` 限制重算範圍；批次讀寫 DOM（先全部讀、再全部寫）；虛擬化長列表（只渲染看得到的列）；用 `transform` / `opacity` 做動畫（只動用合成執行緒）。

3.2.5　找出兇手：Long Animation Frames

Chrome 123 起提供 **Long Animation Frames API**（LoAF）：每個超過 50 ms 的動畫畫格會產生一筆 `long-animation-frame` entry，內含 `scripts[]` 陣列，指出是**哪個腳本、哪個函式、由誰觸發**（`invoker`、`sourceURL`、`sourceFunctionName`）、以及強制版面重算的時間（`forcedStyleAndLayoutDuration`）。`web-vitals` 套件的 attribution 版本會自動把 LoAF 資料對應到 INP 的那次互動。第四章有完整範例。

3.3　CLS：找出誰在跳，並預留空間

3.3.1　再看一次公式



layout shift score $= 0.625 \times 0.25 = 0.156$（單這一次就已超過 0.1 的良好門檻）

圖 3.3: 一次版面偏移的分數計算

3.3.2　常見成因與對策

成因	對策
`<img>`、`<video>`、`<iframe>` 沒有尺寸，載入後把下方內容推開	一律加上 `width` 與 `height` 屬性（瀏覽器會據此推算 `aspect-ratio` 預留空間）；響應式圖片用 CSS `aspect-ratio`。
廣告、嵌入內容、動態載入的推薦區塊沒有預留空間	為容器設定 `min-height`（以最常見尺寸為準）或固定尺寸的骨架；沒廣告時顯示佔位而非收合。
在既有內容**上方**插入內容（通知橫幅、Cookie 同意條、「載入更多」放在頂端）	只在使用者互動後插入（500 ms 內不計）、改為覆蓋層（`position: fixed`）、或預留位置。
網頁字型與後備字型尺寸差異大（FOUT 造成文字重排）	`font-display: optional`（不換字型就不跳）、或用 `size-adjust` / `ascent-override` 讓後備字型與網頁字型的度量一致；preload 關鍵字型。
動畫使用 `top` / `left` / `width` / `margin`	改用 `transform: translate()` / `scale()`，它不觸發版面計算，也不計入 CLS。
客戶端 hydration 後的內容與伺服器 HTML 不一致（hydration mismatch）	確保 SSR 與 CSR 輸出一致；避免只在客戶端才決定的版面（如依 `window.innerWidth`）。
圖片載入後才知道方向（直式 / 橫式）	在後端存好尺寸資訊，一併輸出到 HTML。

```
<!-- 預留空間的三種寫法 -->
<img src="/p.avif" width="600" height="400" alt="">          <!-- 比例 3:2，瀏覽器自動預留 -->
<div class="ad-slot" style="min-height: 250px"></div>          <!-- 廣告容器固定高度 -->
<style>
  .video-wrap { aspect-ratio: 16 / 9; }                          <!-- 響應式容器 -->
  @font-face { font-family: Brand; src: url(/brand.woff2) format('woff2');
               font-display: optional; }
  @font-face { font-family: BrandFallback; src: local('Arial');
               size-adjust: 104%; ascent-override: 92%; }        <!-- 讓後備字型度
```

```
    量對齊 -->
</style>
```

3.3.3　用 API 找出是哪個元素在跳

```
new PerformanceObserver((list) => {
  for (const entry of list.getEntries()) {
    if (entry.hadRecentInput) continue;              // 使用者輸入後 500 ms 內不計
    console.log('shift', entry.value.toFixed(4), 'at', Math.round(entry.startTime),
     'ms');
    for (const src of entry.sources ?? []) {
      // sources 依影響面積排序，第一個就是最大兇手
      console.log('  ', src.node, src.previousRect, '->', src.currentRect);
    }
  }
}).observe({ type: 'layout-shift', buffered: true });
```

DevTools 的 Performance 面板有專屬的 **Layout shifts** 軌道與「Layout shift culprits」insight，會直接列出元素與前後截圖（第五章）。

3.4　綜合案例：一個電商商品頁的診斷流程

假設 CrUX 顯示某商品頁行動版 p75：LCP 4.2 s、INP 380 ms、CLS 0.21，三項全部不及格。建議的排查順序：

1. **先看 TTFB**（PSI 或 RUM）：若 TTFB 已達 1.4 s，先處理伺服器快取與 CDN，否則後面怎麼調都撞到地板。
2. **錄一段 trace 看 LCP by phase**：發現 LCP 元素是商品主圖，由 React 在抓完 API 之後才渲染，資源載入延遲佔了 1.8 s。對策：改為 SSR 輸出主圖、或至少在 HTML 中 preload 該圖並加 `fetchpriority="high"`。
3. **看 INP breakdown**：最慢的互動是點「加入購物車」，處理時間 290 ms，其中一半是重新渲染整個商品頁的 React tree。對策：縮小狀態更新範圍、將購物車數量的更新拆成獨立元件、把分析事件延後送出。
4. **看 Layout shifts 軌道**：最大的一次跳動是「其他人也買了」推薦區塊在 3 秒後插入、把下面的評論推開。對策：容器預留 `min-height`；若資料還沒到就顯示骨架。
5. **再量一次**，並在 RUM 中觀察 28 天內 p75 的變化。記得只改一件事、量一次，才能歸因。

3.5　課堂實作

實作 3-1：手動製造與修復 CLS

建立一個簡單頁面：標題、一張沒有尺寸的大圖、一段文字。用實作 2-1 的 layout-shift 觀察程式碼，在 Network 面板設定 Slow 4G 後重新整理，記錄 CLS 值。接著為圖片加上 `width/height`，再量一次。最後試著在頁面頂部用 `setTimeout` 於 2 秒後插入一條通知橫幅，觀察分數變化，並改成 `position: fixed` 的覆蓋層比較差異。

實作 3-2：把長任務拆開

寫一個按鈕，點擊後在迴圈中建立 20,000 個 `<li>`。用 DevTools Performance 面板（CPU 4x slowdown）錄製點擊，觀察 Interactions 軌道顯示的互動延遲與紅色長任務。然後改用本章的 `renderAllChunked` + `yieldToMain` 版本，再錄一次比較。再進一步：先只渲染前 50 筆並立刻回饋，其餘用 `requestIdleCallback` 慢慢補。

實作 3-3：LCP 圖片的三種寫法

準備一張 300 KB 的 hero 圖，分別以 (a) `<img>` 直接寫在 HTML、(b) CSS `background-image`、(c) JS 在 `DOMContentLoaded` 後插入三種方式載入。用 Performance 面板的「LCP by phase」比較資源載入延遲的差異；再為 (b)(c) 加上 `<link rel=preload as=image fetchpriority=high>` 看能否追上 (a)。

思考題

1. 一個頁面的 LCP 子部分為：TTFB 600 ms、載入延遲 1,200 ms、載入時間 200 ms、渲染延遲 100 ms。最該先處理哪一段？請提出兩個假設性成因與對應的修法。
2. 為什麼「捲動」不算 INP 的互動？如果使用者在捲動時感覺卡頓，那是哪個指標或工具可以反映？
3. `setTimeout(fn, 0)` 與 `scheduler.yield()` 在「讓出主執行緒」這件事上有何差別？在第三方腳本很多的頁面上，這個差別為什麼重要？
4. 有人建議把所有動態內容都改成「使用者點擊後才載入」來規避 CLS（因為 500 ms 內不計）。這個做法在什麼情況下合理？什麼情況下只是把問題藏起來？
5. 為什麼 `font-display: optional` 能同時改善 CLS 與 LCP？它的代價是什麼？

本章回顧

- **LCP = TTFB + 資源載入延遲 + 資源載入時間 + 元素渲染延遲**。先找最長的一段：載入延遲 → 讓瀏覽器早點發現（HTML 內、preload、fetchpriority、不要 lazy）；載入時間 → 變小變近（格式、尺寸、CDN）；渲染延遲 → 移除阻塞、不要等 JS

才顯示。

- **INP = 輸入延遲 + 處理時間 + 呈現延遲**。核心是長任務：拆小、yield、只做這一幀需要的事、把渲染變便宜。用 LoAF 找出兇手腳本。
- **CLS = impact × distance，取最大 session window**。對策是「預留空間」：圖片尺寸、容器 min-height、字型度量對齊、transform 動畫、不要在上方插內容。
- 工作流程永遠是：量測 → 定位子部分 → 對症優化 → 再量測。

第四章　Performance API：瀏覽器原生的量測工具箱

學習目標

- 理解 Performance Timeline 的整體架構：`performance` 物件、`PerformanceEntry` 與各種 entry type。
- 熟練使用 `PerformanceObserver`，並知道為什麼它比 `getEntries()` 更適合真實使用者量測。
- 完整掌握 Navigation Timing（Level 1 vs. Level 2）與 Resource Timing 的每個時間點與衍生計算。
- 認識 Paint Timing、LCP、Event Timing、Layout Instability、Long Tasks / LoAF、User Timing、Server Timing 等 API。
- 用 `web-vitals` 套件建立一個最小可用的 RUM（真實使用者監控）。

4.1　整體架構：Performance Timeline

所有效能量測 API 都掛在全域的 `performance` 物件（`window.performance`）上，並共用一條**效能時間線**（Performance Timeline）。時間線上的每一筆資料都是一個 `PerformanceEntry`，有四個共同欄位：

欄位	意義
`name`	識別名稱：資源的 URL、`first-contentful-paint`、你自訂的 mark 名稱等。
`entryType`	類型：`navigation`、`resource`、`paint`、`largest-contentful-paint`、`event`、`layout-shift`、`longtask`、`long-animation-frame`、`mark`、`measure`、`element`……。
`startTime`	相對於 `performance.timeOrigin` 的高解析時間戳（毫秒，含小數）。
`duration`	持續時間（對某些類型為 0）。

各類型再各自擴充欄位（例如 navigation 有 responseStart，layout-shift 有 value）。

4.1.1　高解析時間

- performance.now()：從 timeOrigin（大致等於導覽開始）算起的毫秒數，微秒級精度、單調遞增、不受系統時鐘調整影響。量測程式執行時間請一律用它，不要用 Date.now()。
- performance.timeOrigin：時間線的零點，以 Unix epoch 毫秒表示。要把 startTime 換成絕對時間就用 timeOrigin + startTime。
- 出於防止時序攻擊 (如 Spectre) 的考量，瀏覽器會對精度做粗化 (Chrome 一般約 100 微秒，跨來源隔離的頁面可達 5 微秒)。

4.1.2　兩種取得 entry 的方式

```
// 方式一：同步查詢（拉取）。只拿得到「現在已經在緩衝區內」的資料。
performance.getEntries();                                  // 全部
performance.getEntriesByType('resource');                  // 依類型
performance.getEntriesByName('first-contentful-paint'); // 依名稱

// 方式二：觀察者（推送）。新 entry 產生時呼叫回呼；buffered: true 可補拿註冊前已發生
    的資料。
const po = new PerformanceObserver((list, observer) => {
  for (const entry of list.getEntries()) console.log(entry.entryType, entry);
});
po.observe({ type: 'largest-contentful-paint', buffered: true });
// 一個 observer 可對多個 type 各呼叫一次 observe()；舊寫法 entryTypes: [...] 不支援
     buffered。

// 檢查瀏覽器支援哪些類型
console.log(PerformanceObserver.supportedEntryTypes);
// 例：["element", "event", "first-input", "largest-contentful-paint", "layout-
    shift",
//      "long-animation-frame", "longtask", "mark", "measure", "navigation", "paint
    ",
//      "resource", "visibility-state"]

po.takeRecords();   // 立即取出尚未交付的 entry（送資料前 flush 用）
po.disconnect();    // 停止觀察
```

為什麼真實使用者量測一定要用 PerformanceObserver？

1. 有些類型（`largest-contentful-paint`、`layout-shift`、`event`、`longtask`）**只能**透過 observer 取得，`getEntriesByType` 拿不到或不完整。
2. `resource` 的緩衝區預設只有 250 筆，滿了之後新資料就丟掉(可用 `performance.setResourceTimingBufferSize(n)` 調大，或監聽 `resourcetimingbufferfull` 事件)，observer 則不受此限。
3. `buffered: true` 讓你可以把量測腳本放在頁面最後、甚至延遲載入，仍能拿到載入初期的資料。
4. 回呼是非同步、批次的，不會阻塞主執行緒。

4.2　Navigation Timing：整份文件的載入時間線

4.2.1　Level 1 vs. Level 2

	Level 1（2012）	**Level 2（現行）**
存取方式	`performance.timing`、`performance.navigation`	`performance.getEntriesByType('navigation')[0]`，型別為 `PerformanceNavigationTiming`
時間格式	Unix epoch 毫秒（整數），要自己減 `navigationStart`	相對於 `timeOrigin` 的高解析毫秒，`startTime` 恆為 0
可否觀察	否	可用 `PerformanceObserver({type: 'navigation'})`
繼承關係	獨立介面	繼承 `PerformanceResourceTiming`，擁有其所有欄位（大小、協定、Server-Timing……）
新增欄位	–	`type`、`redirectCount`、`workerStart`、`activationStart`（prerender）、`criticalCHRestart`、`notRestoredReasons`（bfcache）、`firstInterimResponseStart`（Early Hints）
狀態	**已棄用**，僅為相容保留	建議使用

4.2.2　處理模型：完整時間線

圖 4.1 是 Navigation Timing Level 2 規範的處理模型。上排是網路階段（與 Resource Timing 共用），下排是文件處理階段。

4.2.3　常用的衍生計算

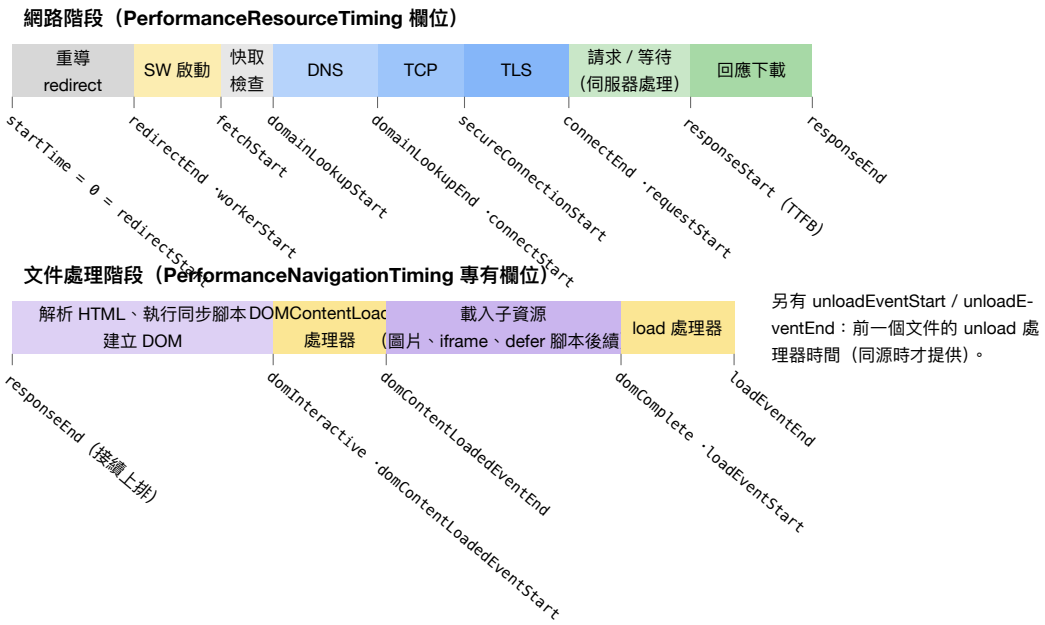


圖 4.1: Navigation Timing Level 2 處理模型。各屬性皆為相對於 `timeOrigin` 的毫秒數；若使用持久連線或命中快取，DNS / 連線相關時間點會等於 `fetchStart`。

```
const t = performance.getEntriesByType('navigation')[0];
const d = (a, b) => +(b - a).toFixed(1);

const timings = {
  redirect:        d(t.redirectStart, t.redirectEnd),
  serviceWorker:   t.workerStart > 0 ? d(t.workerStart, t.fetchStart) : 0,
  dns:             d(t.domainLookupStart, t.domainLookupEnd),
  tcp:             d(t.connectStart, t.secureConnectionStart || t.connectEnd),
  tls:             t.secureConnectionStart > 0 ? d(t.secureConnectionStart, t.
    connectEnd) : 0,
  request:         d(t.requestStart, t.responseStart),   // 送出請求 → 首位元組：主
    要是伺服器處理 + 1 RTT
  ttfb:            d(0, t.responseStart),                // 導覽開始 → 首位元組
  response:        d(t.responseStart, t.responseEnd),    // 下載 HTML 本體
  domParsing:      d(t.responseEnd, t.domInteractive),
  dclHandlers:     d(t.domContentLoadedEventStart, t.domContentLoadedEventEnd),
  subresources:    d(t.domContentLoadedEventEnd, t.domComplete),
  loadHandlers:    d(t.loadEventStart, t.loadEventEnd),
  total:           d(0, t.loadEventEnd),
  // 品質檢查
  protocol:        t.nextHopProtocol,                    // 'h2' / 'h3' / 'http
    /1.1'
  navigationType:  t.type,                               // 'navigate' | 'reload'
```

```
    | 'back_forward' | 'prerender'
  redirects:       t.redirectCount,
  fromCache:       t.transferSize === 0 && t.decodedBodySize > 0,
  compressed:      t.encodedBodySize < t.decodedBodySize,
  earlyHints:      t.firstInterimResponseStart > 0,       // 有收到 103 Early Hints
  prerendered:     t.activationStart > 0,                 // 頁面是由 prerender 啟用
};
console.table(timings);
```

幾個容易誤讀的地方

- `secureConnectionStart` 為 0 表示沒有用 TLS（或重用了連線）；不支援的瀏覽器會是 `undefined`。
- 若連線被重用（keep-alive / HTTP/2），`domainLookupStart` 到 `connectEnd` 全部等於 `fetchStart`，算出來的 DNS / TCP / TLS 都是 0——這是正確的，不是 bug。
- `loadEventEnd` 在 load 事件跑完之前為 0。在 `load` 事件處理器裡讀它會拿到 0，請在下一個 task（如 `setTimeout`）或用 observer 讀。
- 對 prerender 啟用的頁面，web.dev 建議所有指標都應改以 `activationStart` 為零點（`web-vitals` 套件已自動處理）。
- 對從 bfcache 還原的頁面，Navigation Timing 不會重新產生；`web-vitals` 會以 `pageshow` 事件時間為零點另外回報一組指標。

4.3　Resource Timing：每一個子資源的時間線

`PerformanceResourceTiming` 為頁面載入的每個資源（CSS、JS、圖片、字型、XHR / fetch、iframe……）提供與上排相同的網路時間點，加上以下重要欄位：

欄位	用途
`initiatorType`	誰發起的：`img`、`link`、`script`、`css`（CSS 內的 url()）、`fetch`、`xmlhttprequest`、`navigation`、`other`。
`nextHopProtocol`	`h2` / `h3` / `http/1.1`。用來檢查 CDN 或第三方是否還在用舊協定。
`transferSize`	實際經網路傳輸的位元組（含標頭）。**0 表示命中本機快取**（且 `decodedBodySize > 0`）；很小的值（約 300 以下）通常是 304 驗證回應。
`encodedBodySize` / `decodedBodySize`	壓縮前後的 body 大小。兩者相等代表**沒有壓縮**（該啟用 Brotli / gzip）。
`renderBlockingStatus`	`'blocking'` 或 `'non-blocking'`：這個資源有沒有阻塞渲染。快速找出該延後的 CSS / JS。
`responseStatus`	HTTP 狀態碼（Chrome 109+）。
`deliveryType`	`'cache'`（來自 HTTP 快取）、`'navigational-prefetch'`（來自 Speculation Rules 預取）、或空字串。
`workerStart`	若被 Service Worker 處理，SW 開始的時間；`fetchStart − workerStart` 即 SW 啟動成本。
`serverTiming`	伺服器透過 `Server-Timing` 標頭送回的自訂計時（見下文）。
`firstInterimResponseStart` / `finalResponseHeadersStart`	收到 103 Early Hints 與最終標頭的時間；兩者差即 Early Hints 爭取到的「提前量」。

跨來源資源的限制：Timing-Allow-Origin

出於隱私，跨來源資源預設只提供 `startTime`、`fetchStart`、`responseEnd`、`duration`，其餘時間點為 0、大小欄位為 0。若你控制該資源的伺服器（例如自家 CDN），請加上回應標頭 `Timing-Allow-Origin: *`（或列出允許的來源），瀏覽器才會提供完整資料。第三方腳本若沒加此標頭，你就只能看到總時長。

```
// 找出所有 render-blocking 且沒有壓縮、或走 HTTP/1.1 的資源
new PerformanceObserver((list) => {
  for (const r of list.getEntries()) {
    const issues = [];
    if (r.renderBlockingStatus === 'blocking') issues.push('render-blocking');
    if (r.encodedBodySize > 0 && r.encodedBodySize === r.decodedBodySize) issues.
   push('uncompressed');
    if (r.nextHopProtocol === 'http/1.1') issues.push('http/1.1');
    if (r.transferSize === 0 && r.decodedBodySize > 0) issues.push('cache-hit');
    if (issues.length) console.log(r.name, issues.join(', '), Math.round(r.duration
```

```
    ) + 'ms');
  }
}).observe({ type: 'resource', buffered: true });
```

4.4　其他 entry type 一覽

4.4.1　Paint Timing（`paint`）

兩筆 entry：`first-paint` 與 `first-contentful-paint`，`startTime` 即 FP / FCP。新版規範另提供 `paintTime`（繪製階段開始）與 `presentationTime`（像素真正上屏，實作相依）。

4.4.2　Largest Contentful Paint（`largest-contentful-paint`）

每個新的更大候選都會產生一筆 entry；欄位有 `startTime`（= `renderTime` 或 `loadTime`）、`size`（可見面積 px²）、`element`、`url`、`id`。跨來源圖片若未附 `Timing-Allow-Origin`，`renderTime` 可能為 0 或被粗化，此時以 `loadTime` 為準。使用者互動後不再產生新 entry。

4.4.3　Event Timing（`event`、`first-input`）

- `first-input`：第一次互動，永遠會回報（FID 的來源）。
- `event`：所有事件，但預設只回報 `duration` ≥ 104 ms 者；可用 `observe({type: 'event', durationThreshold: 16})` 降低門檻（最小 16）。
- 關鍵欄位：`startTime`（輸入時間）、`processingStart`（處理器開始）、`processingEnd`（處理器結束）、`duration`（到下一畫格上屏，取 8 ms 粗化）、`interactionId`（同一次互動的事件共用；為 0 者不算互動）、`target`。
- 三個子部分即：輸入延遲 = `processingStart − startTime`；處理時間 = `processingEnd − processingStart`；呈現延遲 = `startTime + duration − processingEnd`。

4.4.4　Layout Instability（`layout-shift`）

欄位：`value`(分數)、`hadRecentInput`、`lastInputTime`、`sources[]`(每個含 `node`、`previousRect`、`currentRect`，依影響面積排序)。CLS 的 session window 計算需自行實作，或交給 `web-vitals`。

4.4.5　Long Tasks（longtask）與 Long Animation Frames（long-animation-frame）

- longtask：超過 50 ms 的任務，只有 startTime、duration 與粗略的 attribution（哪個 frame），**無法得知是哪段程式碼**。
- long-animation-frame（LoAF，Chrome 123+）：以「動畫畫格」為單位，超過 50 ms 就回報。多了 blockingDuration、renderStart、styleAndLayoutStart、firstUIEventTimestamp（此畫格內有無使用者輸入），以及最重要的 scripts[]：每個腳本的 invoker（如 BUTTON#buy.onclick）、invokerType（event-listener / user-callback / resolve-promise ……）、sourceURL、sourceFunctionName、sourceCharPosition、duration、forcedStyleAndLayoutDuration。這是目前在真實使用者環境定位 INP 兇手最有效的工具。

```
if (PerformanceObserver.supportedEntryTypes.includes('long-animation-frame')) {
  new PerformanceObserver((list) => {
    for (const loaf of list.getEntries()) {
      if (!loaf.firstUIEventTimestamp) continue;          // 只關心有使用者輸入的畫
    格
      const worst = [...loaf.scripts].sort((a, b) => b.duration - a.duration)[0];
      console.log(`LoAF ${Math.round(loaf.duration)}ms blocking ${Math.round(loaf.
    blockingDuration)}ms`,
        worst && `${worst.invoker} @ ${worst.sourceURL}:${worst.sourceFunctionName}
     ${Math.round(worst.duration)}ms`);
    }
  }).observe({ type: 'long-animation-frame', buffered: true });
}
```

4.4.6　User Timing（mark、measure）

自訂的時間點與區間，會顯示在 DevTools Performance 面板的 Timings 軌道，也可被 RUM 收集。

```
performance.mark('search:start');
const results = await fetchResults(query);
performance.mark('search:fetched', { detail: { count: results.length } });
renderResults(results);
performance.mark('search:rendered');

performance.measure('search:network', 'search:start', 'search:fetched');
performance.measure('search:render',  'search:fetched', 'search:rendered');
```

```
performance.measure('search:total',   { start: 'search:start', end: 'search:
    rendered',
                                        detail: { query } });

new PerformanceObserver((list) => {
  for (const m of list.getEntries()) console.log(m.name, Math.round(m.duration), '
    ms', m.detail);
}).observe({ type: 'measure', buffered: true });
```

4.4.7　Server Timing

伺服器可在回應標頭夾帶自己的計時，瀏覽器會掛在 navigation / resource entry 的 serverTiming[] 上（跨來源需 Timing-Allow-Origin）。這是把「TTFB 裡面伺服器到底在做什麼」帶到前端量測的橋樑，也是 CDN 常用來回報快取命中狀態的方式。

```
Server-Timing: cache;desc="HIT", db;dur=53, render;dur=47.2, cdn-cache;desc="MISS"
```

```
const nav = performance.getEntriesByType('navigation')[0];
for (const { name, duration, description } of nav.serverTiming) {
  console.log(name, description, duration);   // cache HIT 0 / db  53 / render
    47.2 ...
}
```

4.4.8　Element Timing（element）

LCP 只告訴你「最大的那個」；若你關心某個特定元素（例如商品價格、CTA 按鈕）何時畫出來，可在元素上加 elementtiming="price" 屬性並觀察 type: 'element'。目前為 Chromium 專有。

4.5　把它們串起來：web-vitals 套件

Google 維護的 web-vitals 套件（目前 v5）把上述 API 的細節（session window、互動去重、bfcache、prerender 校正、頁面隱藏時送出最終值）全部封裝好，只留下簡單的回呼介面。

```
npm install web-vitals
```

```
// 標準版：只要數值
import { onLCP, onINP, onCLS, onFCP, onTTFB } from 'web-vitals';
```

```
function sendToAnalytics(metric) {
  // metric: { name, value, rating: 'good'|'needs-improvement'|'poor', delta, id,
  //           navigationType: 'navigate'|'reload'|'back-forward'|'back-forward-
    cache'|'prerender'|'restore',
  //           entries: PerformanceEntry[] }
  const body = JSON.stringify({ [metric.name]: metric.value, rating: metric.rating,
                                id: metric.id, nav: metric.navigationType, url:
    location.pathname });
  // sendBeacon 在頁面卸載時也能可靠送出
  (navigator.sendBeacon && navigator.sendBeacon('/rum', body)) ||
    fetch('/rum', { body, method: 'POST', keepalive: true });
}

onLCP(sendToAnalytics);
onINP(sendToAnalytics);
onCLS(sendToAnalytics);
onFCP(sendToAnalytics);
onTTFB(sendToAnalytics);
```

4.5.1　attribution 版本：知道「為什麼」

```
import { onLCP, onINP, onCLS } from 'web-vitals/attribution';

onLCP(({ value, attribution: a }) => {
  console.log('LCP', Math.round(value), {
    element: a.target,                    // CSS selector（v5 由 element 改名為
    target）
    url: a.url,
    ttfb: a.timeToFirstByte,
    loadDelay: a.resourceLoadDelay,
    loadDuration: a.resourceLoadDuration,
    renderDelay: a.elementRenderDelay,
  });
});

onINP(({ value, attribution: a }) => {
  console.log('INP', Math.round(value), {
    target: a.interactionTarget,          // 例如 'button#add-to-cart'
    type: a.interactionType,              // 'pointer' | 'keyboard'
    inputDelay: a.inputDelay,
    processing: a.processingDuration,
```

```
    presentation: a.presentationDelay,
    loadState: a.loadState,                // 互動發生在載入的哪個階段
    longestScript: a.longestScript?.entry?.sourceURL,   // 來自 LoAF
  });
});

onCLS(({ value, attribution: a }) => {
  console.log('CLS', value.toFixed(3), {
    largestShiftTarget: a.largestShiftTarget,
    largestShiftValue: a.largestShiftValue,
    largestShiftTime: a.largestShiftTime,
    loadState: a.loadState,
  });
});
```

web-vitals 的幾個重要行為

- 預設在**頁面隱藏或卸載時**才呼叫回呼一次（LCP 也可能在使用者互動時提早定案）。若要每次變化都回報，傳入 `{reportAllChanges: true}` 並用 `delta` 累加。
- `navigationType` 讓你能區分一般導覽、重新整理、bfcache 還原與 prerender，分析時務必分開看。
- v5 移除了 `onFID`；INP 的 attribution 新增 LoAF 的最長腳本資訊。
- 在 Safari / Firefox 上，LCP、CLS、INP 的回呼可能永遠不會被呼叫（API 不支援），這不是錯誤；設計後端時要容忍缺值。

4.6　課堂實作

實作 4-1：建立一個 30 行的 RUM

把上面標準版 web-vitals 的程式碼加到一個小專案（或直接在任意頁面用 `<script type="module">` 從 unpkg 載入 `web-vitals@5`）。後端用最簡單的方式接收（例如 Express 一個 POST 路由印出 body）。開啟頁面、互動幾次、切換分頁再切回，觀察各指標何時被送出、`rating` 是什麼。

實作 4-2：用 Resource Timing 做一次「資源健檢」

在自己的專案首頁執行本章的 render-blocking / uncompressed / http1.1 檢查腳本。列出：(1) 有幾個 render-blocking 資源？哪些可以加 `defer` 或改為非阻塞？(2) 有沒有沒壓縮的文字資源？(3) 第三方來源有哪些走 HTTP/1.1？(4) 哪些第三方沒有 `Timing-Allow-Origin`，導致你只看得到總時長？

實作 4-3：User Timing 與 DevTools 的整合

在一段非同步流程（例如搜尋、送出表單）前後加入 `performance.mark` 與 `measure`，然後在 Performance 面板錄製並找到 Timings 軌道中你的自訂區段。觀察它與主執行緒活動、網路請求的對應關係。

思考題

1. 為什麼 `Date.now()` 不適合量測程式執行時間？請舉一個 `Date.now()` 會給出負數區間的情境。
2. 某資源的 `transferSize` 是 0、`decodedBodySize` 是 48,000；另一個資源 `transferSize` 是 280、`decodedBodySize` 是 48,000。這兩個資源分別發生了什麼事？
3. 你在 `load` 事件處理器中印出 `loadEventEnd`，得到 0。為什麼？該怎麼正確取得？
4. Level 1 的 `performance.timing.navigationStart` 與 Level 2 的 `timeOrigin` 有何關係？把 Level 1 的 `responseStart` 轉成 Level 2 的相對時間該怎麼算？
5. 為什麼 `longtask` entry 不足以找出 INP 的成因，而 `long-animation-frame` 可以？

本章回顧

- 所有效能 API 共用一條 Performance Timeline；每筆 `PerformanceEntry` 有 `name`、`entryType`、`startTime`、`duration`。
- 真實使用者量測請用 `PerformanceObserver` + `buffered: true`；`getEntries` 只適合 Console 探索。
- Navigation Timing Level 2 的 `PerformanceNavigationTiming` 繼承 Resource Timing，時間點從 `redirectStart` 一路到 `loadEventEnd`；Level 1 的 `performance.timing` 已棄用。
- Resource Timing 的大小欄位可判斷快取命中與壓縮、`nextHopProtocol` 可查協定、`renderBlockingStatus` 可找阻塞資源；跨來源需 `Timing-Allow-Origin`。
- LCP、Event Timing、Layout Instability、LoAF 分別是 LCP、INP、CLS 與「找兇手」的資料來源；`web-vitals` 套件把這些封裝成五個函式與 attribution 資料。

第五章　Chrome DevTools Performance 面板實戰

學習目標

- 看懂 Performance 面板的每個區域：Live Metrics 首頁、時間軸總覽、各軌道、火焰圖、底部分頁與 Insights 側欄。
- 正確設定錄製環境（CPU / 網路節流、截圖），並知道為什麼要這樣設。
- 依照三套固定流程，分別診斷 LCP、INP 與 CLS 問題，並定位到具體的資源、元素或程式碼。
- 搭配 Network 面板讀懂瀑布圖的每個階段與優先權。

本章的使用方式

本章是操作手冊，建議一邊開著 Chrome（建議使用最新的穩定版或 Canary）一邊閱讀。DevTools 介面每幾個月會有調整，若某個按鈕名稱略有不同，請以功能描述為準。所有截圖式的說明都以文字與示意圖呈現，方便你對照自己的畫面。

5.1　開啟面板：Live Metrics 首頁

按 `F12`（macOS：`Cmd+Option+I`）開啟 DevTools，切到 **Performance** 分頁。在尚未錄製任何東西時，你會看到 **Live Metrics** 畫面，它就是一個即時的 Web Vitals 儀表板：

- **三張指標卡**：目前這個分頁的本地 LCP、CLS、INP，並以綠 / 橘 / 紅標示良好 / 待改善 / 不佳。這些數值會隨你操作頁面而更新——非常適合重現「只有在某些操作下才出現」的 INP 或 CLS 問題。
- **LCP 元素**：卡片下方顯示 LCP 元素的節點，滑鼠移上去會在頁面上高亮。展開卡片可看到 LCP 的四個子部分（第三章）。
- **Interactions 與 Layout shifts 面板**：列出每一次互動（目標元素、類型、三個子部分的時間）與每一次版面跳動（分數、元素）。可清除重來。
- **Field data（CrUX）**：右側可開啟「Field metrics」，輸入網址後 DevTools 會抓取 CrUX 的 28 天 p75 數據並排顯示，還會提示「本地與真實數據差異很大」的情況，並依真實使用者的裝置與網路分布**建議你應該用哪一級的 CPU 與網路節流**。

- **Environment settings**：直接在這裡設定 CPU throttling、Network throttling，以及是否模擬行動裝置。

先做這三件事

1. **用無痕視窗或乾淨的 profile**：瀏覽器擴充功能會注入腳本、佔用主執行緒，嚴重干擾量測。
2. **開啟 CPU 節流**：桌機 CPU 比中階手機快 4–10 倍。Performance 面板提供 4x / 6x / 20x 與「Calibrated」（依你的機器校準出「低階行動裝置」與「中階行動裝置」的等效倍率）。行動版問題請至少用 4x。
3. **開啟網路節流**：常用 Slow 4G（或 Fast 4G）。注意這是在瀏覽器層模擬，無法完全反映真實無線網路的封包遺失與 RTT 抖動。

5.2　錄製一段 trace

面板左上角有兩個錄製按鈕：

- **Record**（圓點）：開始錄製目前頁面的執行期行為。用於 INP（錄製時要親手互動）、動畫、捲動卡頓。按 Stop 結束。
- **Record and reload**（重新整理圖示）：重新載入頁面並錄製整個載入過程，載入穩定後自動停止。用於 LCP、FCP、載入階段的 CLS 與長任務。

齒輪圖示展開「Capture settings」：

- **Screenshots**：錄製逐幀截圖，形成 filmstrip，可用來目視確認 LCP 元素何時出現、版面何時跳動。建議開啟。
- **Memory**：記錄 JS heap、節點數等，除錯記憶體問題時才開。
- **Disable JavaScript samples**：關掉 JS 呼叫堆疊取樣，降低錄製開銷，但火焰圖只剩瀏覽器層級事件。一般不勾。
- **Enable advanced paint instrumentation (slow)**：記錄圖層與繪製細節，除錯 paint / composite 問題時才開。
- **Enable CSS selector stats (slow)**：統計哪些選擇器在樣式重算中最耗時。

錄好的 trace 可用右上角的「儲存」匯出為 `.json.gz`，在 code review 或回報 bug 時附上，對方用「載入」即可看到完全相同的畫面。



圖 5.1: Performance 面板版面示意（實際外觀依 Chrome 版本略有差異）

5.3　版面導覽

5.3.1　火焰圖的顏色語義

顏色	類別	典型事件
黃	Scripting	Evaluate Script、Function call、Event、Timer fired、XHR / fetch 回呼、GC
紫	Rendering	Recalculate style、Layout、Pre-paint、Hit test；標紅三角的 Layout 通常是強制同步版面（forced reflow）
綠	Painting	Paint、Composite layers、Image decode、Rasterize
藍	Loading	Parse HTML、Parse stylesheet、Send request、Receive data
灰	Task / System	任務外框；長任務會有紅色標記
淺灰	Idle	主執行緒閒置

5.3.2　操作技巧

- **縮放與平移**：在火焰圖上用 W/S 縮放、A/D 平移；或在總覽列拖曳選取範圍。雙擊一個事件會縮放到它。
- **Breadcrumbs**：每次縮放會在總覽上方留下路徑，可快速跳回上一層。
- **隱藏 / 排序軌道**：對軌道名稱按右鍵可隱藏或調整順序；也可對火焰圖中的函式按右鍵「Hide function」或「Hide children」讓圖更乾淨。
- **點選事件** → **Summary**：顯示時間、來源檔案（可點擊跳到 Sources 面板對應行）、以及對 Layout 事件的「Layout forced」警告。
- **Ctrl / Cmd + 點選 Network 請求**：在 Network 軌道點一個請求，Summary 顯示其優先權、各階段時間、大小與 initiator。
- **Annotations**：對任何事件按右鍵可加上文字標籤、在兩個事件之間畫連線、或標記一段時間範圍，並會隨 trace 一起匯出，適合在團隊內溝通。

5.4　Insights 側欄：把 Lighthouse 的建議搬進 trace

Insights 側欄整合了 Lighthouse 的稽核邏輯，但最大的價值在於**與時間軸連動**：把滑鼠移到一個 insight 上，面板會高亮相關事件、畫出關鍵路徑或子部分覆蓋層；點擊則會縮放到該區域。常用項目：

Insight	它告訴你什麼、該怎麼做
LCP breakdown (LCP by phase)	四個子部分各佔多少（第三章）；開啟 Field data 時會並列 CrUX 的 p75 子部分。看最長的那段。
LCP request discovery	三個檢查：LCP 圖片是否有 `fetchpriority=high`、是否可在 HTML 中被發現、是否**沒有** lazy-load。任一項打叉就是明確的修法。
Render blocking requests	列出阻塞首次繪製的 CSS / JS 與各自造成的延遲估計。
Network dependency tree	從 HTML 出發的關鍵請求鏈（A 載入 B、B 再載入 C），鏈越長 LCP 越晚。目標是縮短、扁平化。
Document request latency	HTML 請求本身的問題：重導、TTFB 過高、未壓縮。
Font display	有沒有字型阻塞文字顯示（FOIT），建議 `font-display`。
Image delivery	圖片格式、尺寸、壓縮的可節省位元組估計。
INP breakdown	錄製期間最慢的互動及其三個子部分；需在錄製時有實際互動。
Layout shift culprits	最大的 CLS 叢集、其中每次跳動的元素、與可能的原因（無尺寸圖片、注入的 iframe、字型變化、動畫）。滑過每次跳動會顯示前後截圖動畫。
Third parties	各第三方來源在主執行緒與網路上的成本排名。
Forced reflow	哪段程式碼在讀取版面屬性時觸發了同步版面計算，含呼叫堆疊。
DOM size	節點數、最大深度、最多子節點的元素。
Legacy JavaScript / Duplicated JavaScript	為舊瀏覽器多打包的 polyfill / transpile 成本；重複打包的模組。
Use efficient cache lifetimes	快取壽命過短的靜態資源（第六章）。

5.5　實戰流程 A：診斷 LCP

1. 在 Live Metrics 頁面設定 CPU 4x、Slow 4G，勾選 Screenshots，按 **Record and reload**。
2. 錄製結束後，在 **Timings** 軌道找到 **LCP** 標記。點它：Summary 會顯示 LCP 元素（可 hover 在頁面上高亮）、時間與相關資源 URL。
3. 開啟左側 **Insights** → **LCP breakdown**：看四個子部分。

- **TTFB 大**：切到 Network 軌道第一個請求，看等待時間；回到第一章、第八章的手法。
- **資源載入延遲大**：在 Network 軌道找到 LCP 圖片的請求條，看它**何時開始**。往左看在它之前主執行緒在做什麼（通常是一大塊黃色的 Evaluate Script——JS 才插入圖片）或 CSS 才剛解析完（背景圖）。同時看 **LCP request discovery** 的三個勾。
- **資源載入時間大**：點該請求，看 Summary 的大小、優先權（Priority 應為 High）、協定。用 filmstrip 對照圖片何時出現。
- **元素渲染延遲大**：看 LCP 資源 `responseEnd` 之後、LCP 標記之前，主執行緒是否被長任務（黃色）佔住，或 Render blocking requests 是否還有東西沒到。

4. 檢查 **Network dependency tree** 與 **Render blocking requests**，把不該在關鍵路徑上的東西移走。
5. 修改後用同樣設定再錄一次比較。變異是正常的，建議至少錄 3 次取中位數。

5.6　實戰流程 B：診斷 INP

1. 設定 CPU 4x（或 Calibrated 的中階行動裝置）。按 **Record**（不是 reload）。
2. **親手操作**你懷疑會慢的互動：點按鈕、開選單、在輸入框打字、切換分頁。也試著**在頁面還在載入時就互動**——真實使用者不會等。按 Stop。
3. 看 **Interactions** 軌道：每次互動一條橫條，超過 200 ms 的會標示警告。橫條分成三段（可 hover 看）：輸入延遲 / 處理時間 / 呈現延遲，兩端的細線（whiskers）代表輸入延遲與呈現延遲。
4. 點選最長的互動，同時對照它**正下方**的 Main 軌道：
   - **輸入延遲長**：互動開始時主執行緒已經有一個灰色 Task 在跑（往往是載入期的 Evaluate Script、第三方 Timer fired、或前一次互動還沒處理完）。找出那個任務是誰的。
   - **處理時間長**：黃色 Event: click / keydown 之下的 Function call 就是你的處理器。點它，Summary 有檔案與行號；切到 **Bottom-Up**、Group by Activity，看哪個函式自身時間最多。框架的 re-render 會顯示為一大串框架內部函式，此時 Group by URL 可快速看出是框架還是你的程式碼。
   - **呈現延遲長**：處理器結束後出現大塊紫色（Recalculate style / Layout）與綠色（Paint），代表 DOM 變動太大或觸發了昂貴的版面計算。若 Layout 帶紅三角，Summary 會指出是哪一行讀了版面屬性（forced reflow）。
5. 開啟 **Insights** → **INP breakdown** 確認三段比例；在 Live Metrics 的 Interactions 列表也可看到同樣資訊。
6. 修法回到第三章：拆長任務、視覺回饋先行、把渲染變便宜。

5.7　實戰流程 C：診斷 CLS

1. Record and reload（載入期跳動）或 Record 後捲動、互動（延遲載入內容的跳動），建議開 Screenshots。
2. 看 **Layout shifts** 軌道：每次跳動是一個小區塊，被歸在同一 session window 的會以叢集標示。點叢集或單次跳動，Summary 列出分數、移動的元素（hover 高亮）、前後位置。
3. 開啟 **Insights** → **Layout shift culprits**：直接列出最大叢集中的每次跳動與可能原因；滑過會播放前後截圖的動畫，一眼看出「誰把誰推開」。
4. 對照 filmstrip 與 Network 軌道：跳動發生的瞬間是哪個資源剛載完（圖片？字型？廣告 iframe？）。
5. 修法回到第三章的成因對策表。修完後用 Live Metrics 的 Layout shifts 面板快速驗證，不必每次錄 trace。

5.8　搭配 Network 面板

Performance 面板的 Network 軌道適合看「與主執行緒的時間關係」，但細節在 **Network** 面板更完整：

- **瀑布圖（Waterfall）各段**：Queueing（等待可用連線 / 優先權排隊）、Stalled、DNS Lookup、Initial connection、SSL、Request sent、**Waiting for server response**（TTFB）、Content Download。點請求 → Timing 分頁可看精確數字。若有 Service Worker，還會多出 Request to ServiceWorker、ServiceWorker preparation、respondWith 等段。
- **Priority 欄**（右鍵表頭開啟）：Highest / High / Medium / Low / Lowest。LCP 圖片應為 High；若是 Low 就是 `fetchpriority` 的用武之地。
- **Protocol 欄**：h3 / h2 / http/1.1。
- **Size 欄**：顯示 `(memory cache)`、`(disk cache)`、`(ServiceWorker)`、`(prefetch cache)`，或實際傳輸大小；下方灰字是解壓後大小。304 會顯示很小的傳輸量。
- **Initiator 欄**：誰發起這個請求；hover 顯示呼叫堆疊。用來追「為什麼 LCP 圖片這麼晚才被請求」。
- **Disable cache**：模擬首次造訪。**Preserve log**：跨導覽保留記錄。**Block request URL**：測試某第三方移除後的影響。
- 篩選列可用 `larger-than:100k`、`-domain:*.example.com`、`is:from-cache`、`has-response-header:Server-Timing` 等語法。

5.9　其他相關工具

- **Lighthouse 面板**：一鍵產生實驗室報告，適合快速盤點；其 insights 與 Performance 面板一致，Lighthouse 12 起可直接「View trace」跳到 Performance 面板。
- **Rendering 側欄**（More tools → Rendering）：Paint flashing（哪些區域在重繪）、Layout Shift Regions（跳動區域藍色閃爍）、Frame Rendering Stats（FPS 與 GPU）、Scrolling performance issues。
- **Performance monitor**（More tools）：即時折線圖：CPU、JS heap、DOM 節點數、每秒 layout / style recalc 次數。找記憶體洩漏與版面抖動很好用。
- **Coverage**（More tools）：顯示每個 JS / CSS 檔有多少位元組在載入期間未被執行 / 使用，是 code splitting 的依據。
- **Application → Back/forward cache**：測試頁面是否能進入 bfcache 並列出原因（第六章）。
- **Web Vitals 擴充功能**：在頁面角落常駐顯示三項指標，並可在 Console 印出 attribution；與 Live Metrics 功能重疊，但方便一般瀏覽時觀察。

5.10　課堂實作

實作 5-1：第一次完整錄製

挑一個內容豐富的公開網站（新聞、電商）。設定 CPU 4x、Slow 4G、Screenshots，Record and reload。完成後回答：(1) LCP 是多少、元素是什麼、四個子部分各多久？(2) 有幾個 render-blocking 請求？(3) 最長的任務多長、是誰的程式碼？(4) 有沒有版面跳動？哪個元素？把 trace 存檔。

實作 5-2：重現並定位一次慢互動

用實作 3-2 的「建立 20,000 個 li」頁面（或任何有慢互動的頁面），CPU 4x，Record，點擊，Stop。在 Interactions 軌道找到該互動，記錄三個子部分；在 Main 軌道找到對應的 Function call，用 Bottom-Up 找出自身時間最多的函式；點擊來源連結跳到 Sources 面板。然後把程式碼改成分批 + yield 版本，再錄一次，比較 Interactions 軌道的變化。

實作 5-3：用 Field data 校準環境

在 Live Metrics 開啟 Field metrics，輸入一個流量足夠的網站。比較本地與 CrUX 的 LCP 差距，並依 DevTools 的建議調整 CPU / 網路節流後再錄一次，看本地數字是否更接近真實 p75。討論：為什麼還是會有差距？（裝置分布、地理位置、快取狀態、登入狀態……）

思考題

1. 火焰圖中一個 Layout 事件帶有紅三角，Summary 顯示「Forced reflow is a likely performance bottleneck」。這代表什麼？請寫出一段會造成這種情況的程式碼，並修正它。
2. Interactions 軌道顯示某次點擊的輸入延遲 180 ms、處理 30 ms、呈現 20 ms。你會先去看 Main 軌道的哪個位置？可能的成因是什麼？
3. 為什麼「Record and reload」適合看 LCP，而 INP 必須用「Record」並親手互動？
4. 為什麼 DevTools 建議依 CrUX 數據調整 CPU 節流倍率，而不是統一用 4x？
5. Network 面板中某圖片的 Priority 是 Low、Initiator 是某個 JS 檔。這兩個資訊分別對應 LCP 的哪個子部分問題？該怎麼修？

本章回顧

- Performance 面板的入口是 **Live Metrics**：即時 LCP / INP / CLS、LCP 元素、互動與跳動列表、CrUX 對照與環境建議。
- 錄製前先設定 **CPU 與網路節流**、開啟 Screenshots，並用乾淨的瀏覽器環境。
- 讀圖三要素：**Timings** 找指標時間點、**Main** 火焰圖看主執行緒在做什麼（黃 scripting、紫 rendering、綠 painting、藍 loading、紅三角長任務）、**Network** 看資源何時開始與結束。
- **Insights 側欄**把 LCP 子部分、INP 子部分、CLS 兇手與各種稽核直接疊在時間軸上，是最快的定位工具。
- 三套流程：LCP 看 Timings + LCP breakdown + Network；INP 看 Interactions + 正下方的 Main + Bottom-Up；CLS 看 Layout shifts + culprits + filmstrip。

第六章　快取策略：HTTP Cache、bfcache 與 Speculation Rules

學習目標

- 說出瀏覽器裡有哪幾層快取，一個請求會依序經過哪些地方。
- 讀懂 `Cache-Control`、`ETag`、`Last-Modified` 等標頭，並能為 HTML、雜湊過的靜態資源、API 各自設計正確的快取策略。
- 用 Resource Timing 與 DevTools 驗證一個資源是否真的命中快取。
- 理解 bfcache（往返快取）如何讓「上一頁」瞬間完成，以及哪些寫法會讓頁面失去資格。
- 使用 Speculation Rules API 預先抓取或預先渲染下一頁，並知道如何避免副作用。

生活比喻

快取就像便利商店的「預先備貨」：客人（頁面）要一瓶水（資源），店員先看櫃檯（memory cache），再看店內貨架（HTTP disk cache），再打電話問附近倉庫（CDN 邊緣節點），最後才向遠方工廠（origin）下單。每往外一層，等待時間就多一個數量級。效能優化最有效的一招，就是讓需求盡量在最靠近客人的那一層被滿足——**最快的請求，是根本沒有發出去的請求**。

6.1　瀏覽器的快取層級

第一章提到，一次新連線要花好幾個 RTT 才能拿到第一個位元組。若資源能在本地快取命中，這些 RTT 全部歸零，而且不佔用頻寬。圖 6.1 是一個請求離開頁面後，可能被「攔下」的各個層級。

6.2　HTTP 快取：新鮮度與驗證

HTTP 快取的規範是 RFC 9111（2022 年，取代舊的 RFC 7234）。它的核心只有兩個概念：

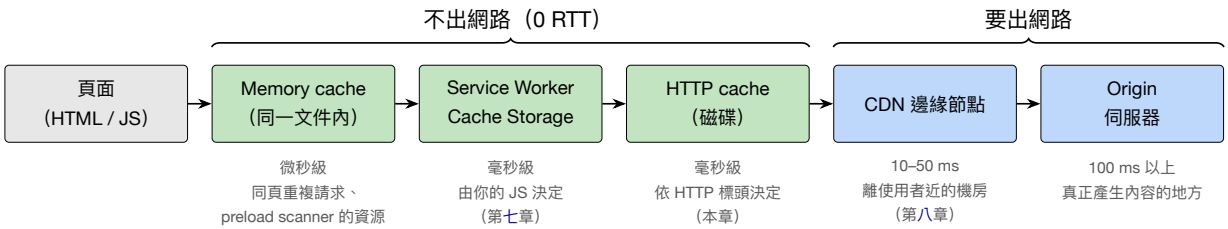


圖 6.1: 一個請求可能在哪些層級被滿足。越靠左，成本越低。

層級	誰控制	特性
Memory cache	瀏覽器自動	只活在目前這個文件的生命週期內；同一資源在同頁被第二次請求時直接命中；preload scanner 抓下的資源也放這裡。DevTools Size 欄顯示「(memory cache)」。
Service Worker Cache Storage	你的 SW 程式碼	可程式化，離線可用，不受 HTTP 標頭限制；但每個請求都要先喚醒 SW（有啟動成本）。詳見第七章。
HTTP cache (disk cache)	HTTP 回應標頭	標準、跨分頁、跨工作階段；本章主角。DevTools 顯示「(disk cache)」。自 Chrome 86 起依「頂層網站」分區（partitioned），A 站快取的 CDN 檔案不能給 B 站用。
bfcache	瀏覽器（依頁面資格）	快取的不是資源而是「整個頁面的執行狀態」，讓上一頁 / 下一頁瞬間完成。見 §6.5。
Prefetch / prerender cache	Speculation Rules	提前抓取或渲染「使用者接下來可能去的頁面」。見 §6.6。

表 6.1: 瀏覽器端的各種快取

新鮮度（freshness）與驗證（validation）

- **新鮮（fresh）**：快取中的回應還在有效期限內，瀏覽器可以**直接使用，不發任何網路請求**。
- **過期（stale）**：有效期限已過，但不代表內容一定變了。瀏覽器會發出**條件式請求（conditional request）**，帶上驗證器（validator）問伺服器「我這份還能用嗎？」。伺服器若回 `304 Not Modified`（沒有 body），瀏覽器就繼續用舊的；若回 `200`，就換成新內容。

6.2.1　控制新鮮度：Cache-Control

指令	意義
`max-age=N`	從回應產生起 N 秒內視為新鮮。注意是以伺服器的 `Date` 起算，再扣掉中間快取回報的 `Age`。
`s-maxage=N`	只對共享快取（CDN、代理）生效的 max-age，優先於 `max-age`。讓你「CDN 快取 10 分鐘、瀏覽器不快取」。
`no-cache`	**可以儲存**，但每次使用前**必須先向伺服器驗證**。等同 `max-age=0, must-revalidate`。名字容易誤導。
`no-store`	完全不儲存（記憶體以外）。用於含個資、一次性的回應。會讓頁面在部分情況失去 bfcache 資格（§6.5）。
`private` / `public`	`private`：只有瀏覽器可存，CDN 不可（含個人化內容的回應）。`public`：即使帶有 `Authorization` 也允許共享快取儲存。
`must-revalidate`	過期後絕不能在「連不到伺服器」時擅自使用舊內容（預設其實允許）。
`immutable`	告訴瀏覽器「這份內容在有效期內絕不會變」，使用者按重新整理時也不要驗證。Firefox、Safari 支援；Chrome 自 54 起本來就不在一般重新整理時驗證子資源，所以效果等價。
`stale-while-revalidate=N`	過期後 N 秒內可以先用舊的回應，同時在背景發請求更新（RFC 5861）。Chrome、Firefox 支援。
`stale-if-error=N`	若重新驗證時伺服器錯誤（5xx）或連不上，過期後 N 秒內允許用舊內容頂著。

表 6.2: 常用的 Cache-Control 指令

沒有 Cache-Control 時，瀏覽器會「猜」

若回應沒有 `Cache-Control` 也沒有 `Expires`，但有 `Last-Modified`，瀏覽器會使用**啟發式新鮮度（heuristic freshness）**：通常取「`Date` 減 `Last-Modified`」的 10% 當作有效期。一個三年沒改的檔案會被快取 100 多天；一個剛部署的檔案則幾乎不快取。這種不確定性正是「我明明改了檔案，使用者卻看到舊版」的常見來源。**每個回應都應明確給出 Cache-Control。**

6.2.2　驗證器：ETag 與 Last-Modified

- `ETag: "abc123"`：內容的識別碼(通常是雜湊)。瀏覽器重新驗證時帶 `If-None-Match: "abc123"`；相同就回 304。前綴 `W/` 表示「弱驗證器」，語意相同即可（例如壓縮前後）。
- `Last-Modified: <date>`：最後修改時間，對應 `If-Modified-Since`。精度只有秒，且多台伺服器的檔案時間可能不同。

ETag 在多台伺服器上要一致

Apache 預設把檔案的 inode 算進 ETag，導致同一檔案在不同機器上有不同 ETag，負載平衡後永遠驗證失敗、永遠回 200。請設定成只用大小與時間(`FileETag MTime Size`），或在建置階段用內容雜湊。另外，某些 nginx 設定在啟用 gzip 時會把強 ETag 降為弱 ETag，這是正常的。

6.2.3　瀏覽器的決策流程

幾個實務上必須知道的行為：

- **一般重新整理（F5）**：Chrome 只重新驗證**主文件**，子資源若仍新鮮就直接用（Chrome 54 起）。**強制重新整理（Shift+F5）**則全部繞過快取。
- **DevTools 的 Disable cache**：只在 DevTools 開著時生效。做效能測試時，記得決定你要測的是「冷快取」還是「熱快取」，並在報告中註明。
- **Vary**：`Vary: Accept-Encoding` 表示不同的 Accept-Encoding 是不同的快取條目。`Vary: Cookie` 或 `Vary: User-Agent` 幾乎等於讓快取失效，尤其對 CDN。
- **快取分區（cache partitioning）**：Chrome 86 起，HTTP cache 以「頂層網站 + 資源來源」為鍵。過去「大家都用同一個 CDN 的 jQuery，所以會命中快取」的說法已不成立；自行託管（self-host）第三方資源反而更划算。

6.3　為不同資源設計策略

快取策略的核心矛盾是：**快取越久越快，但越難更新**。解法是把資源分成兩類，用「URL 版本化」切開這個矛盾。

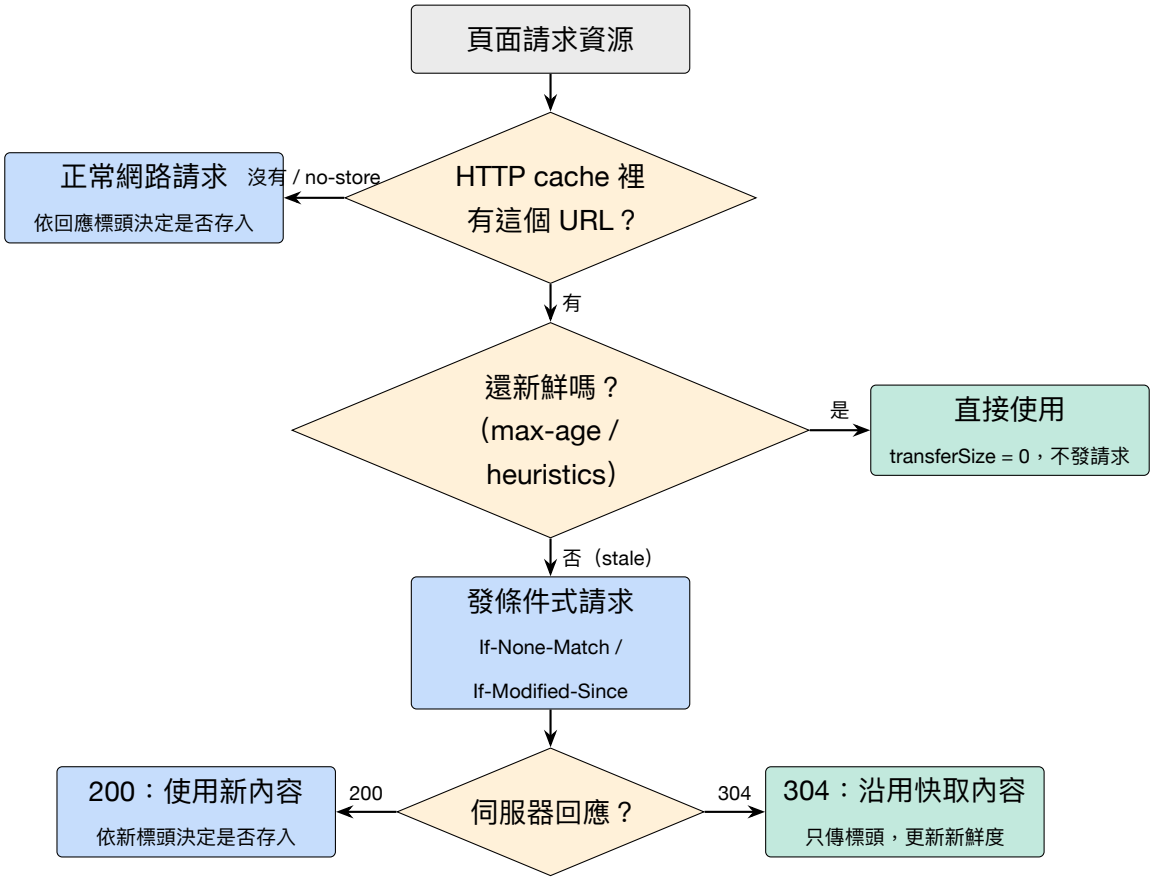


圖 6.2: 瀏覽器對一個子資源請求的快取決策（簡化）。

```
# 1. 雜湊命名的 JS：一年、不驗證
HTTP/2 200
content-type: application/javascript
cache-control: public, max-age=31536000, immutable
etag: "3f2a1b"

# 2. HTML：每次驗證，靠 304 省流量
HTTP/2 200
content-type: text/html; charset=utf-8
cache-control: no-cache
etag: W/"7d9-1a2b3c"
last-modified: Fri, 11 Sep 2026 08:00:00 GMT

# 3. 第二次請求 HTML 時瀏覽器送出的條件式請求與回應
GET / HTTP/2
if-none-match: W/"7d9-1a2b3c"

HTTP/2 304
etag: W/"7d9-1a2b3c"
```

資源	建議標頭	理由
帶內容雜湊的靜態資源 `app.3f2a1b.js`、`main.9c0d.css`	`Cache-Control: public, max-age=31536000, immutable`	內容變了 URL 就變，永遠不需要驗證。一年（31536000 秒）是規範建議的上限。
HTML 文件	`Cache-Control: no-cache` 或 `max-age=0, must-revalidate` 搭配 `ETag`	HTML 是所有雜湊 URL 的「入口」，必須每次驗證才能拿到新版本清單；但 304 讓驗證成本很低。
可由 CDN 快取的 HTML（內容型網站）	`Cache-Control: public, max-age=0, s-maxage=600, stale-while-revalidate=86400`	瀏覽器每次驗證，CDN 快取 10 分鐘且過期後先給舊版再背景更新，TTFB 穩定在邊緣節點的水準（第八章）。
無版本化的圖片、字型	`Cache-Control: public, max-age=86400, stale-while-revalidate=604800`	通常不常變；允許一天內直接用，之後一週內先用再更新。字型若能雜湊命名則直接走第一列。
個人化 API 回應	`Cache-Control: private, no-store` 或 `private, max-age=0, must-revalidate`	不可被 CDN 存；若資料很短暫可用 `private, max-age=30` 減少重複請求。
公開、可預期更新的 API（匯率、排行）	`Cache-Control: public, max-age=60, s-maxage=300`	讓 CDN 吸收尖峰流量。

表 6.3: 常見資源的快取策略

```
cache-control: no-cache
```

程式碼 6.1: 三種典型回應的標頭

```
server {
  listen 443 ssl;
  http2 on;
  etag on;

  # 帶雜湊的建置產物（Vite / webpack 的 assets 目錄）
  location /assets/ {
    add_header Cache-Control "public, max-age=31536000, immutable";
    try_files $uri =404;
  }
```

```
  # 圖片、字型：一天 + SWR 一週
  location ~* \.(png|jpe?g|webp|avif|woff2)$ {
    add_header Cache-Control "public, max-age=86400, stale-while-revalidate=604800"
    ;
  }

  # HTML：每次驗證
  location / {
    add_header Cache-Control "no-cache";
    try_files $uri $uri/ /index.html;
  }
}
```

程式碼 6.2: nginx 設定範例

版本化 URL 是一切的前提

現代打包工具（Vite、webpack、Next.js、Astro）預設就會為 JS / CSS 產出內容雜湊檔名，並在 HTML 中引用。你要做的只是（1）確認伺服器 / CDN 對這些路徑給出一年的 `immutable` 標頭，(2) 確認 HTML 本身是 `no-cache`。不要用 `?v=123` 的查詢字串做版本化：部分 CDN 預設忽略查詢字串，而且部分代理不快取帶查詢字串的 URL。

6.4　驗證快取是否真的命中

第四章介紹過 Resource Timing 的大小欄位，表 6.4 整理了判讀規則：

```
const entries = performance.getEntriesByType('resource');
const stats = { fresh: 0, revalidated: 0, network: 0, sw: 0 };

for (const e of entries) {
  if (e.workerStart > 0) stats.sw++;
  else if (e.transferSize === 0 && e.decodedBodySize > 0) stats.fresh++;
  else if (e.transferSize > 0 && e.transferSize < 600 && e.encodedBodySize > e.
    transferSize) stats.revalidated++;
  else stats.network++;
}
console.table(stats);
// 想看哪些檔沒命中：
console.table(
  entries.filter(e => e.transferSize > 600)
```

情況	Resource Timing	DevTools Network
新鮮命中（不發請求）	`transferSize = 0`，`decodedBodySize > 0`，網路階段全為 0	Size 欄「(disk cache)」或「(memory cache)」，Time 極短
304 重新驗證	`transferSize` 只有標頭大小（約 200–500 B），`encodedBodySize > 0`	Status 304，Size 欄很小
完整下載	`transferSize` ≈ `encodedBodySize` + 標頭	Status 200，Size 為實際傳輸量
Service Worker 供應	`workerStart > 0`；`transferSize` 依 SW 是否上網而定	Size 欄「(ServiceWorker)」

表 6.4: 判斷資源來自哪一層

```
        .map(e => ({ name: e.name.split('/').pop(), kB: (e.transferSize / 1024).
    toFixed(1) }))
);
```

程式碼 6.3: 統計本頁資源的快取命中率

跨來源資源看不到大小

沒有 `Timing-Allow-Origin` 的跨來源資源，`transferSize` 等大小欄位一律為 0，會被誤判為「命中」。先用 `e.transferSize === 0 && e.encodedBodySize === 0` 排除這類條目（兩者都為 0 表示被遮蔽，而不是命中）。

6.5　bfcache：讓「上一頁」瞬間完成

往返快取（back/forward cache，bfcache）

當使用者離開頁面時，瀏覽器不銷毀它，而是把**整個頁面的狀態**（DOM、JS heap、捲動位置、正在執行的計時器）**凍結**在記憶體中。使用者按「上一頁」或「下一頁」時直接解凍，不重新載入、不重新執行 JS。Chrome 的資料顯示，行動裝置上約每五次導覽就有一次是往返導覽，bfcache 讓這些導覽的 LCP、CLS、INP 幾乎為零。

被 bfcache 還原的頁面**不會**觸發 `load`、`DOMContentLoaded`，而是觸發 `pageshow`（`event.persisted === true`）；離開時觸發 `pagehide`（`event.persisted` 表示「正在進入 bfcache」）。頁面在快取中處於凍結狀態：計時器暫停、JS 不執行、網路請求被中止。

6.5.1　哪些情況會失去資格

阻擋原因	說明與解法
註冊了 `unload` 監聽器	歷史上最常見的原因（第三方分析腳本尤其愛用）。改用 `pagehide` 或 `visibilitychange`。Chrome 已從 117 起分階段棄用 `unload`，可用 `Permissions-Policy: unload=()` 主動禁止。
`Cache-Control: no-store`（主文件）	傳統上直接阻擋。Chrome 自 2024 年起分階段放寬：若頁面在快取期間沒有 cookie 變動、也不依賴 HttpOnly cookie 等敏感狀態，仍可還原。最好的做法是不要對 HTML 用 `no-store`，改 `no-cache`。
開著的連線	未關閉的 WebSocket、WebRTC、進行中的 IndexedDB 交易、BroadcastChannel。在 `pagehide` 時關閉，在 `pageshow` 時重連。
`window.opener` 關聯	用 `rel="noopener"` 開新視窗（現代瀏覽器對 `target="_blank"` 預設如此）。
瀏覽器內部限制	頁面太大、播放中的媒體擷取、部分 API 使用中、進入快取超過一段時間（Chrome 約 10 分鐘）後被逐出。

表 6.5: 常見的 bfcache 阻擋原因

6.5.2　測試與監測

- **DevTools**：Application → Back/forward cache →「Test back/forward cache」。它會自動導覽離開再返回，列出所有阻擋原因。
- **Lighthouse**：有一項「Page prevented back/forward cache restoration」稽核。
- **現場監測**：`PerformanceNavigationTiming.notRestoredReasons`（Chrome 123+）。當往返導覽沒有從 bfcache 還原時，這個物件列出原因（含 iframe 內的原因）。

```
// 1. 不要用 unload；離開時用 pagehide 收尾
window.addEventListener('pagehide', (event) => {
  if (event.persisted) {
    // 正在進入 bfcache：關閉會阻擋資格的連線
    socket?.close();
  }
  // 送出分析資料（sendBeacon 不會阻擋 bfcache）
  navigator.sendBeacon('/analytics', JSON.stringify(collect()));
});
```

```
// 2. 被還原時，更新可能過時的資料
window.addEventListener('pageshow', (event) => {
  if (event.persisted) {
    console.log('從 bfcache 還原');
    refreshCartBadge();   // 使用者可能在另一頁改了購物車
    socket = connect();   // 重新建立連線
  }
});

// 3. 回報「為什麼沒有被還原」
const nav = performance.getEntriesByType('navigation')[0];
if (nav.type === 'back_forward' && nav.notRestoredReasons) {
  const reasons = nav.notRestoredReasons.reasons?.map(r => r.reason);
  navigator.sendBeacon('/analytics', JSON.stringify({ bfcacheBlocked: reasons }));
}
```

程式碼 6.4: bfcache 友善的頁面寫法

bfcache 與 web-vitals

`web-vitals` 函式庫會把 bfcache 還原視為一次新的頁面瀏覽並重新回報 LCP、CLS、INP（值通常非常小），這是刻意的：使用者確實「看到了一頁」。若你自己寫量測程式，記得在 `pageshow` 中重置狀態，否則 CLS 會把前後兩次瀏覽累加在一起。

6.6　Speculation Rules：預先抓取與預先渲染下一頁

前面的快取都是「這一頁用過的東西下次快一點」；Speculation Rules API 則是「**猜使用者接下來要去哪一頁，先做起來放**」。

prefetch 與 prerender

- **prefetch**：先下載目標頁的 HTML 主文件（只有 HTML，不含子資源、不執行）。導覽時省下整段網路時間，TTFB 接近 0。
- **prerender**：在隱藏的背景分頁中**完整載入並渲染**目標頁，包含子資源與 JS。導覽時直接「啟用（activate）」，LCP 可接近 0 ms。成本也高得多（CPU、記憶體、流量），且會執行目標頁的 JS。

Chrome 109 起支援（同網站 prerender；跨網站 prefetch 不帶 cookie）。其他瀏覽器目前會忽略這些規則，所以它是「漸進式增強」。

```
<script type="speculationrules">
```

```
{
  "prerender": [{
    "where": {
      "and": [
        { "href_matches": "/products/*" },
        { "not": { "selector_matches": ".no-prerender" } }
      ]
    },
    "eagerness": "moderate"
  }],
  "prefetch": [{
    "where": { "href_matches": "/*" },
    "eagerness": "conservative"
  }]
}
</script>
```

程式碼 6.5: 在 HTML 中宣告 Speculation Rules

eagerness	觸發時機	適用
`immediate`	規則一出現就執行（舊名 `eager`）	使用者幾乎一定會去的頁（結帳下一步）。
`eager`	目前行為與 immediate 相近，未來可能以更早的訊號觸發	少數高機率連結。
`moderate`	滑鼠懸停約 200 ms，或觸控裝置上 pointerdown	一般連結清單的預設選擇。
`conservative`	`pointerdown` / `mousedown`（點下去到放開之間）	想省資源，仍可搶到約 100 ms。

表 6.6: eagerness 的四種等級（`list` 規則預設 immediate，`document` 規則預設 conservative）

6.6.1　在目標頁面偵測與配合

被 prerender 的頁面會真的執行 JS，因此分析、廣告曝光、A/B 分派等**有副作用的程式必須延到啟用之後**：

```
function onActivated(fn) {
  if (document.prerendering) {
    document.addEventListener('prerenderingchange', fn, { once: true });
  } else {
```

```
    fn();
  }
}

onActivated(() => {
  sendPageView();          // 只在使用者真的看到頁面時計一次
  startVideoAutoplay();
});

// 量測時要扣掉 activationStart，否則 LCP 會出現「負數」或極小值
const nav = performance.getEntriesByType('navigation')[0];
const activationStart = nav.activationStart || 0;
new PerformanceObserver((list) => {
  const last = list.getEntries().at(-1);
  console.log('LCP（相對於啟用）', Math.max(last.startTime - activationStart, 0));
}).observe({ type: 'largest-contentful-paint', buffered: true });
```

程式碼 6.6: prerender 感知的初始化

伺服器端可用請求標頭 `Sec-Purpose: prefetch` 或 `Sec-Purpose: prefetch;prerender` 辨認推測性請求，例如不計入瀏覽統計、或回 503 拒絕（瀏覽器之後會正常導覽）。

prerender 的代價與限制

- Chrome 對數量有上限（immediate / eager 的 prerender 約 10 個、moderate / conservative 約 2 個，先進先出），且在省電模式、低記憶體、行動數據「省流量」模式下會停用。
- 目標頁若彈出對話框、要求權限、播放音效，這些會被延後到啟用時才發生，但仍要避免。
- 不要對「會改變伺服器狀態」的連結（登出、加入購物車）做 prefetch / prerender。用 GET 做這類操作本身就是錯的。
- 舊的 `<link rel="prerender">` 早已降級成 NoState Prefetch（只抓資源不執行），不是真正的預渲染；`<link rel="prefetch">` 仍可用於「下一頁需要的子資源」。
- DevTools：Application → Speculative loads 可看到每條規則的狀態與失敗原因。

三種「提前」的差別

- `<link rel="preload">` / `fetchpriority`：**這一頁**需要、但瀏覽器發現得太晚的資源（第三章）。
- `<link rel="prefetch">`：**下一頁**需要的子資源（JS chunk、圖片），低優先權下載進 HTTP cache。

- Speculation Rules：**下一頁本身**。prefetch 抓 HTML；prerender 整頁做完。

6.7　課堂實作

實作 6-1：盤點一個網站的快取標頭

挑一個網站，打開 DevTools Network，勾選 Disable cache 後重新載入，觀察 Response Headers 中每類資源的 `Cache-Control` 與 `ETag`。依表 6.3 判斷：(1) JS / CSS 是否雜湊命名並給一年？(2) HTML 是否 `no-cache` 或短 `max-age`？(3) 有哪些資源缺少 Cache-Control（會落入啟發式）？寫下三條你會改的地方。可用 `curl -sI <url>` 加速盤點。

實作 6-2：驗證熱快取

取消 Disable cache，第二次載入同一頁。在 Console 執行程式碼 6.3 的統計（本章「驗證快取是否真的命中」小節），並對照 Network 的 Size 欄。哪些檔每次都完整下載？看它們的標頭找原因。再用 Performance 面板比較冷、熱兩次載入的 LCP。

實作 6-3：bfcache 體檢

在 DevTools Application → Back/forward cache 按下 Test。若不合格，找出原因（常見：某個第三方腳本註冊了 unload）。在自己的專案中，用 `Permissions-Policy: unload=()` 或移除監聽器修正，再測一次。最後寫一段 `pageshow` 處理，在還原時更新頁面上會過時的區塊。

實作 6-4：為列表頁加上 Speculation Rules

在一個有「列表→詳情」結構的示範專案中加入 `document` 規則：詳情頁 prerender（moderate）、其他連結 prefetch（conservative）。打開 Application → Speculative loads 觀察狀態，滑過連結後點擊，用 Performance 面板比較 LCP；並確認詳情頁的分析事件只在啟用後送出一次。

思考題

1. `Cache-Control: no-cache` 與 `no-store` 的差別是什麼？哪一個會影響 bfcache？
2. 一個 `max-age=31536000` 的 `app.js`（沒有雜湊檔名）上線後發現有 bug，你有哪些補救辦法？為什麼「版本化 URL」能從根本上避免這個問題？
3. 某資源的 Resource Timing 顯示 `transferSize = 0`、`encodedBodySize = 0`、`decodedBodySize = 0`。它是命中快取了嗎？
4. 為什麼 `Vary: Cookie` 會讓 CDN 幾乎無法快取？
5. 新聞網站首頁要讓 CDN 快取、但讀者每次進站都要看到 5 分鐘內的最新內容，

且來源伺服器掛掉時仍能顯示舊版。請寫出一行 `Cache-Control`。

6. prerender 一個「登出」連結會發生什麼事？怎麼避免？

本章回顧

- 快取的層級由近到遠：memory cache → Service Worker → HTTP cache → CDN → origin。越近越快，目標是讓請求在最近的一層被滿足。
- HTTP 快取只有兩個概念：**新鮮**（直接用）與**驗證**（條件式請求換 304）。`Cache-Control` 控制新鮮度，`ETag` / `Last-Modified` 負責驗證。永遠明確給出 Cache-Control，避免啟發式快取。
- 標準配方：雜湊命名的靜態資源 `max-age=31536000, immutable`；HTML `no-cache`；CDN 可快取的 HTML 用 `s-maxage` + `stale-while-revalidate`；個人化內容 `private`。
- 用 Resource Timing 的 `transferSize` 與 DevTools 的 Size 欄驗證命中；注意跨來源遮蔽。
- bfcache 讓往返導覽瞬間完成：不要用 `unload`、HTML 不要 `no-store`、離開時關連線；用 `pageshow` 還原後更新資料，用 `notRestoredReasons` 監測。
- Speculation Rules 把「下一頁」提前做好：prefetch 省網路時間、prerender 省全部時間；用 eagerness 控制成本，目標頁要延後副作用並扣掉 `activationStart`。

第七章　Service Worker 與效能

學習目標

- 說明 Service Worker（SW）在頁面與網路之間扮演什麼角色，以及它的生命週期。
- 用 Cache Storage 實作 Cache First、Network First、Stale-While-Revalidate 等策略，並知道各自適用的資源。
- 理解 SW 的效能成本（啟動時間、每個請求都經過 JS），並用 navigation preload 與 Static Routing API 降低成本。
- 用 Workbox 產生可維護的 SW，處理版本更新與快取清理。
- 用 Resource Timing 的 `workerStart` 與 DevTools 觀察 SW 對每個請求的影響。

生活比喻

如果 HTTP cache 是「照店家規定備貨的便利商店」，Service Worker 就是你自己雇的「私人採購」：所有出門買東西的需求先經過他，他可以決定從自家倉庫（Cache Storage）拿、出門買、或先給舊的再去補貨。他很靈活，但每次被叫醒都要花點時間穿鞋（啟動成本），而且如果什麼都要經過他，簡單的事也會變慢。

7.1　Service Worker 是什麼

Service Worker

一個在瀏覽器背景執行、**與頁面分離**的 JavaScript 執行緒，註冊後會成為其作用範圍(scope)內所有頁面的**網路代理**：頁面發出的每個請求(HTML、JS、圖片、`fetch()`）都會先觸發 SW 的 `fetch` 事件，由 SW 決定如何回應。它沒有 DOM 存取權、事件驅動、閒置時會被瀏覽器終止，並且只能在 HTTPS（或 localhost）下運作。

SW 能做的事很多（推播、背景同步、離線），但本章只談效能：**它讓你可以用 JavaScript 決定每個請求的快取策略**，不再受限於伺服器的 HTTP 標頭，並且能在離線或網路很差時仍立即回應。

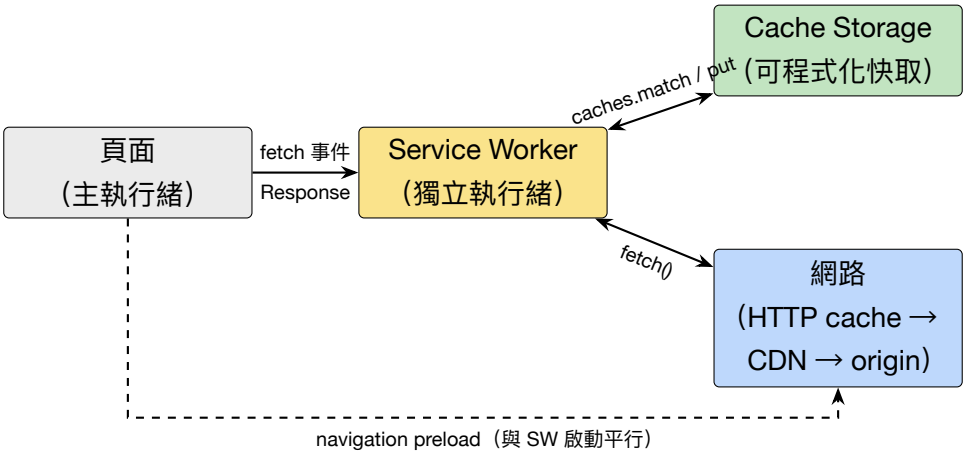


圖 7.1: Service Worker 位於頁面與網路之間，可自行決定回應來源。

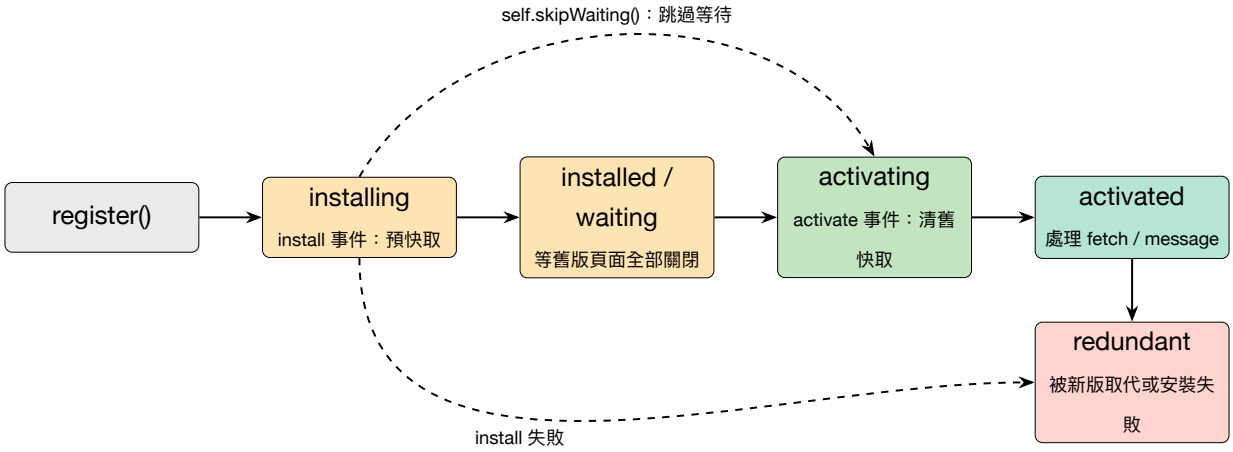


圖 7.2: Service Worker 的生命週期。

7.2　生命週期

1. **註冊**：頁面呼叫 `navigator.serviceWorker.register('/sw.js')`。第一次載入的頁面**不受控制**（不會經過 fetch 事件），除非 SW 在 activate 時呼叫 `clients.claim()`。
2. **install**：只在 SW 第一次安裝或內容變更時觸發。這裡做**預快取(precache)**：把 app shell、關鍵 JS / CSS 放進 Cache Storage。用 `event.waitUntil()` 包住 Promise，全部成功才算安裝完成。
3. **waiting**：若有舊版 SW 仍控制著任何分頁，新版會等待，避免同一時刻兩個版本共存。呼叫 `self.skipWaiting()` 可強制接手（但要處理頁面與新 SW 的版本不一致）。
4. **activate**：適合刪除舊版本的快取。
5. **更新檢查**：使用者導覽到作用範圍內的頁面、觸發功能事件、或呼叫 `registration.update()` 時，瀏覽器會重新下載 `sw.js`，逐位元組比較，有差異就啟動新版生命週

期。無論 HTTP 標頭怎麼寫，瀏覽器最多把 sw.js 快取 24 小時；仍建議對它設 Cache-Control: no-cache。

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', async () => {
    // 等 load 之後再註冊，避免和首屏資源搶頻寬與 CPU
    const reg = await navigator.serviceWorker.register('/sw.js', { updateViaCache:
    'none' });

    reg.addEventListener('updatefound', () => {
      const newWorker = reg.installing;
      newWorker.addEventListener('statechange', () => {
        if (newWorker.state === 'installed' && navigator.serviceWorker.controller)
    {
          // 新版已就緒但在等待：提示使用者重新整理
          showToast('有新版本', { action: () => newWorker.postMessage({ type: '
    SKIP_WAITING' }) });
        }
      });
    });

    // 新 SW 接手後重新載入，確保頁面與 SW 版本一致
    navigator.serviceWorker.addEventListener('controllerchange', () => location.
    reload());
  });
}
```

程式碼 7.1: 在頁面中註冊並處理更新

7.3　Cache Storage 與快取策略

Cache Storage 是 SW（與頁面）都能用的鍵值儲存：鍵是 Request，值是 Response。常用 API：caches.open(name)、cache.addAll(urls)、cache.match(request)、cache.put(request, response)、cache.delete()、caches.keys()。

```
const VERSION = 'v2026-09-12';
const SHELL_CACHE = `shell-${VERSION}`;
const RUNTIME_CACHE = `runtime-${VERSION}`;
const SHELL = ['/', '/offline.html', '/assets/app.3f2a1b.js', '/assets/main.9c0d.
    css'];

self.addEventListener('install', (event) => {
```

策略	行為	適用
Cache First（cache, falling back to network）	先查快取，命中就回；未命中才上網並存入。	帶雜湊檔名的 JS / CSS、字型、不會變的圖片。
Network First（network, falling back to cache）	先上網（可設逾時），失敗才用快取。	HTML 導覽、必須最新的 API；離線時仍有內容。
Stale-While-Revalidate	立刻回快取（若有），同時在背景上網更新快取，下次才用新的。	頭像、商品圖、非關鍵 API：要快又能接受晚一步更新。
Network Only	不經快取。	付款、寫入操作、非 GET 請求。
Cache Only	只用快取（預快取階段已放入）。	App shell 在特定情境下。

表 7.1: 五種基本快取策略（Workbox 命名）

```
  event.waitUntil(caches.open(SHELL_CACHE).then((c) => c.addAll(SHELL)));
});

self.addEventListener('activate', (event) => {
  event.waitUntil((async () => {
    // 刪掉不是這個版本的快取
    for (const key of await caches.keys()) {
      if (!key.endsWith(VERSION)) await caches.delete(key);
    }
    await self.clients.claim();
  })());
});

self.addEventListener('message', (e) => {
  if (e.data?.type === 'SKIP_WAITING') self.skipWaiting();
});

self.addEventListener('fetch', (event) => {
  const { request } = event;
  if (request.method !== 'GET') return;           // 非 GET 一律不經手（Network Only
    ）
  const url = new URL(request.url);

  if (request.mode === 'navigate') {
```

```
    event.respondWith(networkFirst(request));     // HTML：Network First + 離線頁
  } else if (url.pathname.startsWith('/assets/')) {
    event.respondWith(cacheFirst(request));       // 雜湊資源：Cache First
  } else if (request.destination === 'image') {
    event.respondWith(staleWhileRevalidate(request));
  }
  // 其他請求不呼叫 respondWith → 瀏覽器照常處理
});

async function cacheFirst(request) {
  const cached = await caches.match(request);
  if (cached) return cached;
  const response = await fetch(request);
  if (response.ok) {
    const cache = await caches.open(RUNTIME_CACHE);
    cache.put(request, response.clone());         // body 只能讀一次，要 clone
  }
  return response;
}

async function networkFirst(request, timeoutMs = 3000) {
  const cache = await caches.open(RUNTIME_CACHE);
  try {
    const response = await Promise.race([
      fetch(request),
      new Promise((_, reject) => setTimeout(() => reject(new Error('timeout')),
    timeoutMs)),
    ]);
    if (response.ok) cache.put(request, response.clone());
    return response;
  } catch {
    return (await cache.match(request)) || (await caches.match('/offline.html'));
  }
}

async function staleWhileRevalidate(request) {
  const cache = await caches.open(RUNTIME_CACHE);
  const cached = await cache.match(request);
  const fetching = fetch(request).then((response) => {
    if (response.ok) cache.put(request, response.clone());
    return response;
  }).catch(() => cached);
```

```
  return cached || fetching;
}
```

程式碼 7.2: sw.js：手寫版的三種策略

Cache Storage 的幾個坑

- **只快取成功的回應**：檢查 `response.ok`，否則 404 / 500 會被永久快取。
- **opaque 回應**：跨來源且沒有 CORS 的請求（`mode: 'no-cors'`）回傳 status 0 的 opaque 回應，看不到內容也無法判斷成功；Chrome 計算配額時會把它們「膨脹」到約 7 MB 一筆。第三方資源最好加上 `crossorigin` 屬性走 CORS。
- **配額**：Cache Storage 與 IndexedDB 共用來源配額（Chrome 約為可用磁碟的 60%），用 `navigator.storage.estimate()` 查詢；超過會被逐出（eviction），SW 程式碼要能承受快取消失。
- **查詢字串**：`cache.match(request, { ignoreSearch: true })` 可忽略 `?utm=...`。
- **版本化命名**：每次部署改 `VERSION`，在 activate 清舊快取；否則使用者的磁碟會被不再使用的舊檔佈滿。

7.4　SW 的效能成本與對策

SW 不是免費的。從第一章的導覽流程可以看到，導覽請求在檢查 HTTP cache 之前，要先**啟動 SW**（`workerStart` 到 `fetchStart` 之間）。

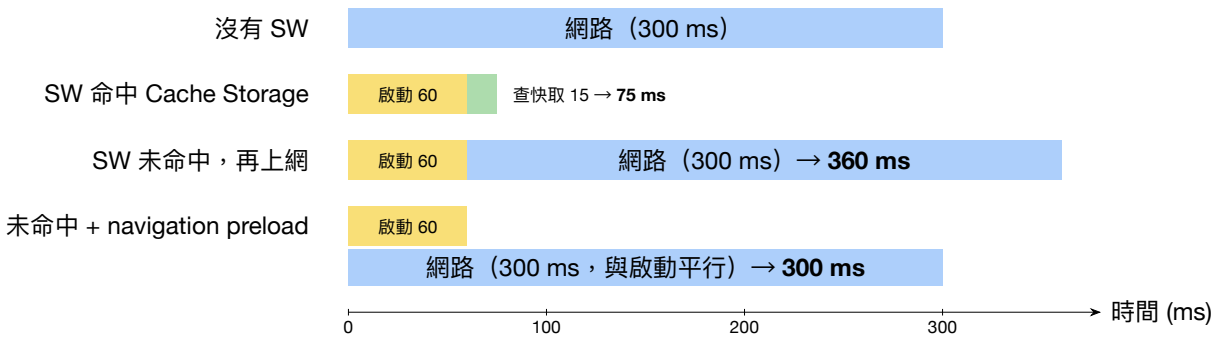


圖 7.3: SW 啟動成本的示意。命中時 SW 大幅領先；未命中時反而多付一次啟動時間，navigation preload 可把它藏起來。（數字為示意）

SW 何時真的有幫助

- **回訪與導覽**：app shell、字型、雜湊資源命中 Cache Storage，可以做到「離線也秒開」；這是 SW 最強的場景。
- **網路很差時**：Network First 加逾時、或 SWR，讓頁面不因單一慢請求而卡住。

- **HTTP 標頭不可控時**：第三方或舊系統的資源沒有好的 Cache-Control，SW 可以自己決定。
- **反例**：首次載入不會受益（SW 尚未安裝）；每個請求都經過 SW 的 JS，簡單的 Cache First 對雜湊資源反而不如 HTTP cache 直接命中（HTTP cache 不用喚醒任何東西）。

7.4.1　Navigation preload

讓瀏覽器在啟動 SW 的**同時**就把導覽請求送出去，SW 啟動後從 `event.preloadResponse` 取得結果，而不是等 SW 起來才開始上網。

```
self.addEventListener('activate', (event) => {
  event.waitUntil((async () => {
    if (self.registration.navigationPreload) {
      await self.registration.navigationPreload.enable();
      // 伺服器可透過 Service-Worker-Navigation-Preload 標頭辨認這種請求
      await self.registration.navigationPreload.setHeaderValue('shell-v2');
    }
  })());
});

self.addEventListener('fetch', (event) => {
  if (event.request.mode !== 'navigate') return;
  event.respondWith((async () => {
    const preload = await event.preloadResponse;   // 可能是 undefined
    if (preload) return preload;
    return fetch(event.request);
  })());
});
```

程式碼 7.3: 啟用 navigation preload

7.4.2　Static Routing API：讓部分請求完全不經過 SW

Chrome 123 起支援 `event.addRoutes()`：在 install 階段宣告「哪些請求直接走網路 / 直接查快取」，瀏覽器就不必為這些請求啟動 SW 或執行 fetch handler。

```
self.addEventListener('install', (event) => {
  event.addRoutes([
    // 影片與大型下載：一律直接走網路，不喚醒 SW
    { condition: { requestDestination: 'video' }, source: 'network' },
    { condition: { urlPattern: '/api/*' }, source: 'network' },
```

```
    // 雜湊資源：直接查 Cache Storage 中名為 shell-v2026 的快取，查不到才進 fetch
    handler
    { condition: { urlPattern: '/assets/*' }, source: { cacheName: 'shell-v2026
    -09-12' } },
    // SW 正在啟動時，導覽請求同時走網路與 fetch handler，誰快用誰
    { condition: { requestMode: 'navigate', runningStatus: 'not-running' },
      source: 'race-network-and-fetch-handler' },
  ]);
});
```

程式碼 7.4: Static Routing API

空的 fetch handler 是純成本

只為了「讓網站可安裝 (PWA)」而寫一個什麼都不做的 `fetch` 監聽器，會讓每次導覽多付一次 SW 啟動時間卻毫無收益。Chrome 從 115 起會偵測「no-op fetch handler」並跳過啟動，DevTools 也會警告；正確做法是**不要註冊**用不到的 fetch 事件。

7.5　用 Workbox 少寫、少錯

Workbox 是 Google 維護的 SW 函式庫，把上面手寫的策略、預快取清單、過期清理都封裝好，並在建置時自動產生「檔名→ revision」的清單，解決版本化的痛點。

```
import { precacheAndRoute, cleanupOutdatedCaches } from 'workbox-precaching';
import { registerRoute, NavigationRoute } from 'workbox-routing';
import { CacheFirst, NetworkFirst, StaleWhileRevalidate } from 'workbox-strategies'
    ;
import { ExpirationPlugin } from 'workbox-expiration';
import { CacheableResponsePlugin } from 'workbox-cacheable-response';

// 建置工具會把 [{url, revision}] 清單注入 self.__WB_MANIFEST
precacheAndRoute(self.__WB_MANIFEST);
cleanupOutdatedCaches();

// HTML 導覽：Network First，3 秒逾時
registerRoute(new NavigationRoute(new NetworkFirst({ cacheName: 'pages',
    networkTimeoutSeconds: 3 })));

// 圖片：Cache First，最多 60 張、30 天
registerRoute(
  ({ request }) => request.destination === 'image',
  new CacheFirst({
    cacheName: 'images',
```

```
    plugins: [
      new CacheableResponsePlugin({ statuses: [0, 200] }),
      new ExpirationPlugin({ maxEntries: 60, maxAgeSeconds: 30 * 24 * 60 * 60 }),
    ],
  })
);

// 非關鍵 API：SWR
registerRoute(({ url }) => url.pathname.startsWith('/api/feed'), new
    StaleWhileRevalidate({ cacheName: 'api' }));
```

程式碼 7.5: Workbox（injectManifest 模式）的 sw.js

- **generateSW**：完全由設定檔產生 sw.js，適合簡單需求；**injectManifest**：你寫 sw.js，Workbox 只注入預快取清單，彈性最大。
- 框架整合：`vite-plugin-pwa`、Next.js 的 `@serwist/next`、Angular 的 `@angular/service-worker` 都以 Workbox 或類似機制為基礎。
- `workbox-window` 提供頁面端的更新提示流程（`waiting` 事件、`messageSkipWaiting()`）。

7.6　觀察與除錯

```
const nav = performance.getEntriesByType('navigation')[0];
if (nav.workerStart > 0) {
  console.log('SW 啟動 + 攔截前置（ms）:', (nav.fetchStart - nav.workerStart).
    toFixed(1));
}

const viaSW = performance.getEntriesByType('resource').filter(e => e.workerStart >
    0);
console.table(viaSW.map(e => ({
  name: e.name.split('/').pop(),
  swOverhead: (e.fetchStart - e.workerStart).toFixed(1),
  total: e.duration.toFixed(1),
  fromNetwork: e.transferSize > 0,
})));
```

程式碼 7.6: 量測 SW 對每個請求的額外成本

工具	用途
Application → Service Workers	狀態、版本、**Update on reload**（每次重新整理都裝新版）、**Bypass for network**（暫時跳過 SW）、Unregister、手動觸發 push / sync。
Application → Cache storage	檢視每個快取的內容與大小，手動刪除。
Network 面板	Size 欄「(ServiceWorker)」表示由 SW 回應；同一請求會出現兩列（頁面→ SW、SW →網路，後者帶齒輪圖示）。
Performance 面板	在 Main 之外會看到 Service Worker 執行緒的軌道，可量測 fetch handler 花了多久。
Resource Timing	`workerStart > 0` 表示經過 SW；`fetchStart − workerStart` 是 SW 啟動 + handler 前置時間。
`chrome://serviceworker-internals`	所有已註冊 SW 的內部狀態、啟動 / 停止。

表 7.2: 觀察 Service Worker 的工具

出問題時的逃生口

SW 一旦出錯，使用者可能一直看到壞掉的舊版。上線前準備好：(1) `sw.js` 必須 `no-cache`；(2) 保留一支「自毀 SW」——內容只有在 activate 時 `unregister()` 並刪除所有快取，緊急時部署它取代原檔；(3) 不要用 SW 快取第三方腳本的 HTML 回應或含個資的 API；(4) 確保 `Cache-Control` 與 SW 策略不衝突（SW 內 `fetch()` 仍會用 HTTP cache，雜湊資源的一年快取在 SW 未命中時依然有效）。

7.7　課堂實作

實作 7-1：為示範專案加上 app shell

在一個 Vite 專案中手寫 `sw.js`：install 預快取 `index.html`、雜湊後的 JS / CSS 與 `offline.html`；fetch 對導覽用 Network First（3 秒逾時），對 `/assets/` 用 Cache First。用 DevTools Network 的 Offline 模式驗證離線可開，再用 Resource Timing 程式碼觀察 `workerStart`。

實作 7-2：量化 SW 的成本與收益

同一頁分別在 (a) 無 SW、(b) SW 且 Cache Storage 命中、(c) SW 但清空 Cache Storage 三種情況下，用 Performance 面板（Slow 4G、CPU 4x）各錄三次，記錄

LCP 與導覽的 `responseStart`。討論（c）為什麼比（a）慢，然後加上 navigation preload 再測一次。

實作 7-3：改用 Workbox 並做更新流程

用 `vite-plugin-pwa`（injectManifest）改寫 7-1，加上 `ExpirationPlugin`。修改一行程式後重新部署，觀察 Application 面板的 waiting 狀態，實作「有新版本，點此更新」的提示並驗證 `controllerchange` 後頁面重新載入。

思考題

1. 為什麼 SW 第一次安裝後，當前頁面的請求不會經過它？要怎麼改變這個行為，有什麼風險？
2. 對帶雜湊檔名的 `app.3f2a1b.js`，HTTP cache 一年 immutable 與 SW Cache First，哪個回應更快？為什麼？
3. 描述 Stale-While-Revalidate 在「第一次」「第二次」「第三次」請求同一張圖時各回傳什麼。
4. 一個電商網站把 `/api/cart` 用 Cache First 快取，會發生什麼問題？應該用哪種策略？
5. Navigation preload 解決的是哪一段時間？請對照 Navigation Timing 的屬性說明。
6. 你發現使用者回報「網站永遠停在舊版」。列出三個可能的原因與對應的修法。

本章回顧

- SW 是頁面與網路之間的可程式化代理，生命週期為 install（預快取）→ waiting → activate（清舊快取）→ fetch。第一次載入不受控制，更新靠逐位元組比較 `sw.js`。
- 五種策略：Cache First（雜湊資源）、Network First（HTML、必須最新）、SWR（要快又可稍舊）、Network Only（寫入）、Cache Only。只快取 `response.ok` 的回應，記得 `clone()`。
- SW 有啟動成本：命中時遠快於網路，未命中時反而更慢。用 navigation preload 平行化、用 Static Routing API 讓不需要的請求繞過 SW、不要寫空的 fetch handler。
- Workbox 解決版本化與清理的繁瑣工作；injectManifest 模式最有彈性。
- 用 Application 面板、Network 的 (ServiceWorker) 標示、Resource Timing 的 `workerStart` 觀察成本；上線前準備 `no-cache` 的 sw.js 與自毀腳本。

第八章　CDN：把內容搬到使用者旁邊

學習目標

- 從 RTT 與連線建立的角度解釋 CDN 為什麼能加速，以及它加速的到底是哪幾段時間。
- 認識 CDN 的基本架構：PoP / 邊緣節點、Anycast、origin shield、cache key。
- 正確搭配 `s-maxage`、`CDN-Cache-Control`、`stale-while-revalidate` 與清除（purge）機制，讓 HTML 也能安全地在邊緣快取。
- 從回應標頭（`Age`、`Cache-Status`、`CF-Cache-Status`、`X-Cache`）判讀命中狀況，並估算快取命中率。
- 了解影像 CDN、邊緣運算、Early Hints 等進階功能對 LCP 與 TTFB 的幫助。

生活比喻

網購如果每件商品都要從國外原廠工廠直接寄來，不管快遞多努力，跨海的時間就是省不掉。CDN 就是在每個城市設倉庫：熱門商品先囤在離你最近的倉庫（邊緣節點），下單後幾小時就到；倉庫沒有的才向原廠調貨（回源，origin fetch），而且調到之後會留一份在倉庫給下一個人。

8.1　距離就是時間

第一章算過：一次全新的 HTTPS 連線到拿到第一個位元組，TCP + TLS 1.3 需要約 3 個 RTT，再加上 TCP 慢啟動，前幾十 KB 還要多幾個 RTT。RTT 的下限由物理決定：光在光纖中約每秒 20 萬公里，**每 1,000 公里的往返至少 10 ms**，實際路由再加上交換與排隊延遲，通常是理論值的 1.3–2 倍。

CDN 加速的三件事

1. **縮短「與使用者之間」的 RTT**：TCP / TLS 交握、慢啟動、每個請求的往返，全部改成與 10 ms 外的邊緣節點進行。這對**所有**回應都有效，包含不能快取的動態內容。
2. **在邊緣儲存回應**：命中時完全不必回源，TTFB 只剩邊緣的處理時間（通常 < 50 ms）。
3. **優化邊緣到 origin 的路徑**：長期保持的暖連線（不用重新交握）、私有骨幹網路、

使用者（台北）→伺服器	距離（約）	實測 RTT（約）	3 RTT 建立連線
台北市內機房 / 邊緣節點	＜ 50 km	2–10 ms	6–30 ms
東京	2,100 km	30–45 ms	90–135 ms
新加坡	3,200 km	50–70 ms	150–210 ms
美國西岸	11,000 km	130–160 ms	390–480 ms
歐洲（法蘭克福）	9,400 km（經美國或中東）	220–280 ms	660–840 ms

表 8.1: 距離對 RTT 與連線成本的影響（有線網路的典型值；行動網路再加 20–100 ms）

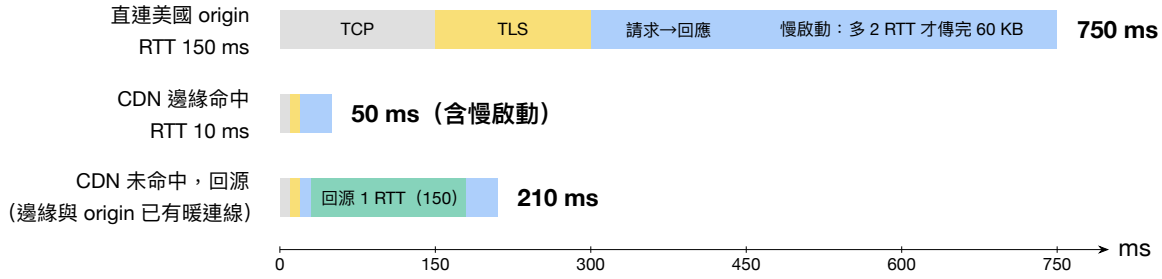


圖 8.1: 同一份 60 KB 的 HTML，直連遠端與經由 CDN 的時間比較（示意）。即使未命中，邊緣節點與 origin 之間長期保持的暖連線也省下了 TCP / TLS 交握與慢啟動。

請求合併（request coalescing：多個相同的未命中只回源一次）。

8.2　架構：PoP、Anycast 與 origin shield

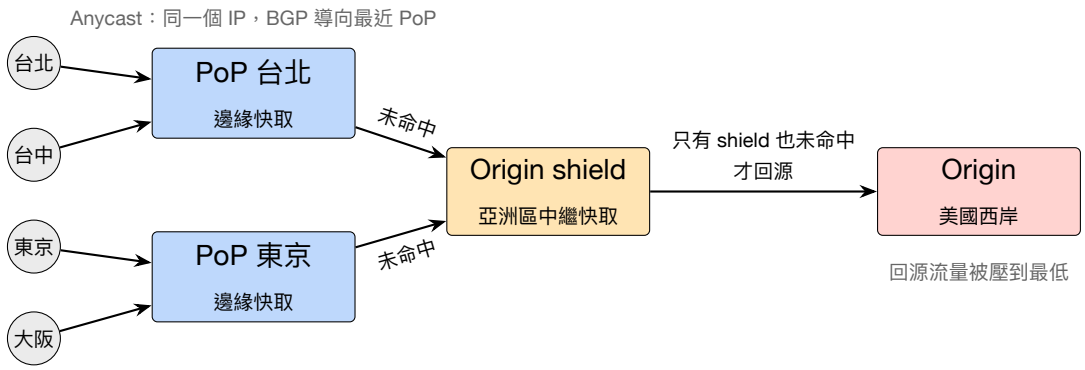


圖 8.2: CDN 的多層快取架構。

- **PoP（Point of Presence）/ 邊緣節點**：分布在各城市機房的伺服器群，負責終結 TLS、回應快取內容、壓縮、執行邊緣程式。大型 CDN 有 300+ 個城市的 PoP。

- **Anycast**：所有 PoP 宣告同一個 IP，網際網路路由（BGP）自然把封包送到最近的 PoP；另一種做法是 GeoDNS（依解析者位置回不同 IP）。這也是為什麼 DNS 查詢後你會看到 CDN 的 IP 而不是自己伺服器的。
- **Origin shield / tiered cache**：在邊緣與 origin 之間再加一層區域快取。沒有它，每個 PoP 第一次未命中都各自回源；有了它，同一區的 PoP 共享一份，命中率上升、origin 負載下降。
- **Cache key**：CDN 用來辨認「同一個資源」的鍵，預設是 host + path + query string，可設定忽略某些參數、加入裝置類型、語言、國家、特定 cookie。鍵越細，命中率越低。

8.3　讓 CDN 正確快取：標頭與清除

8.3.1　告訴 CDN 該存多久

標頭	說明
`Cache-Control: s-maxage=N`	標準指令，只影響共享快取；瀏覽器用 `max-age`。最常見的組合：`public, max-age=0, s-maxage=600`。
`CDN-Cache-Control`	RFC 9213「目標式快取控制」：只給 CDN 看的一組完整指令，會覆蓋 `Cache-Control`，且 CDN 通常不會把它轉給瀏覽器。各家還有專屬版本：`Cloudflare-CDN-Cache-Control`、Fastly 的 `Surrogate-Control`、Akamai 的 `Edge-Control`。
`stale-while-revalidate` / `stale-if-error`	對 CDN 特別重要：過期後先回舊版、同時背景回源更新，使用者永遠不必等回源；origin 掛掉時繼續服務。
`Vary`	CDN 會為每個 Vary 值組合存一份。`Vary: Accept-Encoding` 正常；`Vary: Cookie` / `User-Agent` 幾乎等於不快取（多數 CDN 會直接 BYPASS）。
`Set-Cookie`	帶有 Set-Cookie 的回應多數 CDN 預設不快取（避免把某人的 session 發給別人）。靜態資源請不要帶 cookie。
`Surrogate-Key` / `Cache-Tag`	為回應貼標籤（如 `product-123`、`category-shoes`），之後可依標籤一次清除相關頁面。

表 8.2: 與 CDN 相關的回應標頭

```
HTTP/2 200
content-type: text/html; charset=utf-8
cache-control: public, max-age=0, must-revalidate
```

```
cdn-cache-control: max-age=3600, stale-while-revalidate=86400, stale-if-error
    =604800
cache-tag: article-8812, author-42, home
etag: "a1b2c3"
vary: Accept-Encoding
```

程式碼 8.1: 一個內容型網站的文章頁：瀏覽器每次驗證、CDN 快取一小時並 SWR 一天

8.3.2　清除（purge）與版本化

快取越久，命中率越高，但內容更新時就得主動清除。三種做法搭配使用：

1. **版本化 URL**（雜湊檔名）：靜態資源永遠不需要清除，CDN 給一年。
2. **標籤清除（purge by tag）**：文章更新時清掉 `article-8812` 標籤下的所有 URL（文章頁、列表頁、首頁），數秒內全球生效。這是讓 HTML 可以「長快取 + 即時更新」的關鍵。
3. **URL 清除 / 全站清除**：緊急用；全站清除會讓命中率瞬間歸零、origin 承受尖峰，慎用。

HTML 上 CDN 的三個前提

1. **匿名與登入要分流**：有 session cookie 的請求 BYPASS（或由邊緣程式改寫 cache key），匿名使用者共享同一份 HTML。個人化區塊（購物車數量、使用者名稱）改由前端在載入後用 API 補上，或用邊緣運算注入。
2. **origin 必須正確設定 s-maxage 或 CDN-Cache-Control**：忘記設會落入 CDN 的預設值（有的預設不快取 HTML、有的預設快取 2 小時）。
3. **有可用的清除機制**：內容系統（CMS）發布時要能觸發 purge by tag，否則只能靠短 TTL。

8.4　判讀命中與命中率

```
# -s 安靜、-o 丟掉 body、-D - 印標頭；連打兩次比較第二次是否命中
curl -s -o /dev/null -D - https://www.example.com/ | grep -iE "^(age|cache-control|
    cdn-cache-control|cache-status|cf-cache-status|x-cache|server-timing|vary):"

# 量測各階段時間（DNS、連線、TLS、TTFB、總計）
curl -s -o /dev/null -w "dns %{time_namelookup}s  tcp %{time_connect}s  tls %{
    time_appconnect}s  ttfb %{time_starttransfer}s  total %{time_total}s  proto %{
    http_version}\n" https://www.example.com/

# 假裝從另一個地區：多數 CDN 可用特定標頭或 --resolve 指到特定 PoP 的 IP
```

標頭	說明
`Age: 537`	這份回應在快取裡待了 537 秒。有 Age 幾乎就表示命中（0 或沒有則是剛回源）。
`Cache-Status`（RFC 9211）	標準化格式：`Cache-Status: "Edge-TPE"; hit; ttl=3063`，可串多層（shield、edge 各一段）。Netlify、Fastly 等支援。
`CF-Cache-Status`	Cloudflare：HIT、MISS、EXPIRED、STALE（SWR 供應舊版）、REVALIDATED、BYPASS（因 cookie / no-store）、DYNAMIC（依規則不快取此類型）。
`X-Cache`	CloudFront：`Hit from cloudfront` / `Miss from cloudfront` / `RefreshHit`；Fastly：`HIT` / `MISS`（加 `X-Served-By` 看是哪個 PoP）。
`X-Vercel-Cache`	Vercel：HIT、MISS、STALE、PRERENDER、REVALIDATED。
`Server-Timing`	部分 CDN 會輸出 `cdn-cache; desc=HIT`、`edge; dur=1` 等，可用 Resource Timing 的 `serverTiming` 在前端讀取（第四章）。

表 8.3: 常見 CDN 的命中狀態標頭

```
curl -s -o /dev/null -D - --resolve www.example.com:443:203.0.113.10 https://www.
    example.com/
```

程式碼 8.2: 用 curl 檢查 CDN 行為

```
const hits = { HIT: 0, MISS: 0, other: 0 };
for (const e of performance.getEntriesByType('resource')) {
  const st = e.serverTiming?.find((s) => s.name === 'cdn-cache');
  if (!st) continue;
  const status = st.description?.toUpperCase();
  hits[status in hits ? status : 'other']++;
}
console.table(hits);

// 也看看走的是什麼協定：h3 表示 QUIC
console.table(
  performance.getEntriesByType('resource').map((e) => ({ host: new URL(e.name).host
    , proto: e.nextHopProtocol }))
);
```

程式碼 8.3: 在前端統計 CDN 命中（需要 Server-Timing 與 Timing-Allow-Origin）

快取命中率（cache hit ratio, CHR）

命中次數 ÷ 總請求次數。靜態資源目標 > 95%；HTML 依網站性質而異，內容型網站可達 80–90%，電商匿名頁面 60–80%。CDN 主控台都有此數據，也可依「位元組」計算（bandwidth savings）。命中率低的常見原因：cache key 太細（query string、cookie）、TTL 太短、長尾 URL 太多、沒有 origin shield、Vary 太多。

8.5　進階功能：影像、Early Hints、邊緣運算

8.5.1　影像 CDN

LCP 元素多半是圖片，而圖片是頁面位元組的大宗。影像 CDN 在邊緣即時做：依 `Accept` 標頭轉成 AVIF / WebP（`format=auto`）、依 URL 參數縮放到需要的寬度、調整品質、去除 metadata，結果再快取。

```
<img
  src="https://img.example.com/cdn-cgi/image/width=800,quality=75,format=auto/hero.
    jpg"
  srcset="
    https://img.example.com/cdn-cgi/image/width=480,quality=75,format=auto/hero.jpg
     480w,
    https://img.example.com/cdn-cgi/image/width=800,quality=75,format=auto/hero.jpg
     800w,
    https://img.example.com/cdn-cgi/image/width=1200,quality=75,format=auto/hero.
    jpg 1200w"
  sizes="(max-width: 600px) 100vw, 800px"
  width="800" height="450" alt="..." fetchpriority="high">
```

程式碼 8.4: 搭配 srcset 使用影像 CDN（以常見的 URL 參數風格為例）

Next.js 的 `next/image`、Astro 的 `<Image>`、Nuxt Image 等元件就是在幫你產生這類 URL；自架時可用 imgproxy、Thumbor 搭配 CDN。

8.5.2　103 Early Hints

第一章介紹的 Early Hints 特別適合由 CDN 發出：CDN 記住某個 URL 上次回應的 `Link: rel=preload` 標頭，下次一收到請求就先回 103，讓瀏覽器在 origin 還在產生 HTML 的幾百毫秒內開始下載 CSS 與 LCP 圖片。Cloudflare、Fastly、Shopify 等都已支援。

8.5.3　邊緣運算（edge compute）

Cloudflare Workers、Fastly Compute、CloudFront Functions / Lambda@Edge、Vercel Edge Functions、Netlify Edge Functions 讓你在 PoP 上執行程式，冷啟動以毫秒計。常見的效能用途：

- **依 cookie 改寫 cache key 或決定 BYPASS**：匿名使用者共用快取，登入者回源。
- **在快取的 HTML 上注入個人化片段**：類似舊時代的 ESI，讓整頁仍可快取。
- **A/B 測試分派**：在邊緣改寫 URL 或標頭，避免前端閃爍（CLS）與額外請求。
- **邊緣 SSR**：把第九章的伺服器端渲染搬到離使用者最近的地方，資料 API 仍在遠端時要注意「邊緣到資料庫」的往返反而變多。
- **標頭治理**：統一補上 Cache-Control、安全標頭、`Timing-Allow-Origin`，不必改 origin。

```
export default {
  async fetch(request, env, ctx) {
    const cookie = request.headers.get('Cookie') || '';
    const loggedIn = /session=/.test(cookie);
    if (loggedIn || request.method !== 'GET') {
      return fetch(request);                  // 登入者：直接回源，不進快取
    }
    const cache = caches.default;
    const cacheKey = new Request(new URL(request.url).toString(), request); // 去掉
     cookie 影響
    let response = await cache.match(cacheKey);
    if (!response) {
      response = await fetch(request, { cf: { cacheTtl: 600, cacheEverything: true
    } });
      response = new Response(response.body, response);
      response.headers.set('Cache-Control', 'public, max-age=0, s-maxage=600');
      ctx.waitUntil(cache.put(cacheKey, response.clone()));
    }
    return response;
  },
};
```

程式碼 8.5: Cloudflare Worker：匿名者用邊緣快取，登入者回源

錯誤	後果與修法
用 `?v=2` 做版本化	部分 CDN 預設忽略 query string →使用者拿到舊版；或每個參數組合各存一份→命中率低。改用雜湊檔名。
所有回應都帶 `Set-Cookie`	CDN 拒絕快取。靜態資源與匿名 HTML 不要設 cookie；把分析用 cookie 改由 JS 設定。
`Vary: User-Agent`	快取碎成數千份。改用 client hints 或在邊緣把 UA 正規化成 mobile / desktop 兩類。
把 API 放在另一個網域	多一次 DNS + TCP + TLS，且 CORS 預檢再多一個 RTT。盡量同網域（用路徑區分）或至少 `preconnect`。
字型用第三方託管	因快取分區沒有共享優勢，還多一條連線。自行託管 `woff2` 並給一年快取。
沒開 HTTP/3 / Brotli	邊緣通常一鍵開啟；確認 `nextHopProtocol` 是 `h3`、`content-encoding` 是 `br` 或 `zstd`。
沒有 origin shield	每個 PoP 各自回源，命中率低、origin 尖峰大。
TTL 太短又沒有 SWR	每次過期都有使用者要等回源。加上 `stale-while-revalidate`。

表 8.4: CDN 常見設定錯誤

8.6　常見錯誤

8.7　CDN 與 Web Vitals 的對應

指標	CDN 的影響
TTFB	最直接：邊緣命中的 TTFB 通常 $<$ 100 ms；未命中時也因暖連線與路徑優化改善。CrUX 中 TTFB 差、且使用者分布在多國的網站，CDN 是首選解法。
FCP / LCP	HTML 更早到→一切更早開始；影像 CDN 縮小 LCP 圖片位元組；Early Hints 提前發現 CSS 與圖片；HTTP/3 減少弱網下的隊頭阻塞。
INP	間接：資源更快到達可減少載入期間的競爭；但 INP 主要取決於主執行緒工作，CDN 幫不了長任務。
CLS	間接：字型與圖片更快到達，減少延遲載入造成的跳動；邊緣 A/B 分派避免前端切換造成的閃爍。

表 8.5: CDN 對各指標的影響

8.8　課堂實作

實作 8-1：辨認一個網站用了哪家 CDN，命中了沒

對三個知名網站執行程式碼 8.2 的第一條指令兩次。從標頭判斷：（1）是哪家 CDN？(2) HTML 是否被邊緣快取（看 Age、狀態標頭）？(3) 它的 Cache-Control 策略是什麼？再用第二條指令比較 TTFB，並在 DevTools Network 看 Protocol 欄是否為 h3。

實作 8-2：從遠端量測

用 WebPageTest（或任何多地點測試工具）分別從台灣、美國、歐洲對同一個網站測試，比較 TTFB 與 LCP。有 CDN 的網站各地差距應該很小；找出一個差距很大的網站，推測原因（沒有 CDN？HTML 不快取？API 在單一區域？）。

實作 8-3：設計快取策略

為一個假想的電商網站列出：首頁、分類頁、商品頁、購物車、結帳、`/api/cart`、`/api/products/123`、雜湊資源、商品圖片。為每個 URL 寫出 `Cache-Control` 與 `CDN-Cache-Control`（若需要），標註要貼的 cache tag，並說明商品價格改變時要清除什麼。

思考題

1. CDN 對「完全不能快取」的動態 API 回應有沒有幫助？為什麼？
2. `Cache-Control: public, max-age=0, s-maxage=600, stale-while-revalidate=86400` 這行對瀏覽器與 CDN 分別意味著什麼？
3. 為什麼帶 `Set-Cookie` 的回應不該被 CDN 快取？如果網站的框架對所有頁面都設 session cookie，該怎麼處理？
4. 一個網站命中率只有 40%，可能原因有哪些？請依「最可能」排序。
5. Early Hints 由 CDN 發出，跟由 origin 發出，各有什麼優缺點？
6.「邊緣 SSR」一定比「單一區域 SSR」快嗎？什麼情況下會更慢？

本章回顧

- RTT 由距離決定，連線建立又要花好幾個 RTT；CDN 把交握對象換成 10 ms 外的邊緣節點，對所有回應（含動態）都有效，命中時更能完全免回源。
- 架構關鍵字：PoP、Anycast、origin shield、cache key。cache key 越細命中率越低。
- 用 `s-maxage` 或 `CDN-Cache-Control` 分開控制 CDN 與瀏覽器；HTML 上 CDN 要配合 SWR、匿名 / 登入分流與 purge by tag。

- 從 `Age`、`Cache-Status`、`CF-Cache-Status`、`X-Cache` 判讀命中；用 curl 與 Resource Timing 驗證，目標靜態資源命中率 $>$ 95%。
- 影像 CDN、Early Hints、邊緣運算分別針對 LCP 位元組、資源發現時間與個人化快取，是 CDN 之上最划算的三項功能。

第九章　伺服器端渲染（SSR）與渲染策略

學習目標

- 畫出 CSR、SSR、SSG、ISR、串流 SSR 各自的載入時間線，並說出每種策略對 TTFB、FCP、LCP、INP 的影響方向。
- 解釋 hydration 是什麼、為什麼它常常是 SSR 頁面的長任務來源，以及 selective hydration、islands、React Server Components 各用什麼方式減輕它。
- 用 React 18+ 的 `renderToPipeableStream` 與 `<Suspense>` 實作串流 SSR，並排除代理 / CDN 緩衝造成的失效。
- 針對不同類型的頁面（行銷頁、商品頁、儀表板）選擇合適的渲染策略與快取搭配。
- 在 DevTools 與 Performance API 中辨認 SSR 頁面的典型瓶頸。

生活比喻

點餐時，CSR 像是外送一個「自己組裝的餐盒」：先送空盒與說明書（空 HTML 與 JS），你自己再去拿食材 (API) 組起來，過程中盒子是空的；SSR 是廚房先組好整份餐點送來，一打開就能看，但廚房要花時間、餐點也可能還沒附餐具（尚未 hydration，不能互動）；串流 SSR 則是先送前菜、主菜隨後陸續上桌，你不必等全部做完才開始吃。

9.1　渲染策略光譜

SSR 改善的是哪些指標，代價又是什麼

- **FCP、LCP 提前**：HTML 一到就有內容可畫，不必等 JS。LCP 圖片也能被 preload scanner 更早發現。
- **TTFB 變長**：伺服器要取資料並渲染，這段時間全算進 TTFB；若資料來源慢，TTFB 可能比 CSR 的空殼慢數百毫秒。串流與快取是解方。
- **INP 風險**：hydration 是一段密集的 JS 執行，常形成長任務；使用者在頁面「看起來好了」時點擊，卻要等 hydration 結束才有反應。

策略	HTML 何時、在哪產生	優點	代價
CSR (client-side rendering)	瀏覽器執行 JS 後才產生內容；伺服器只給空殼	伺服器最便宜、可全部放 CDN；互動狀態自然	FCP / LCP 最晚；依賴 JS 下載 + 執行 + API 往返的瀑布
SSR (server-side rendering)	每次請求時在伺服器產生完整 HTML	FCP / LCP 早；SEO 友善；資料在伺服器端取得（離資料庫近）	TTFB 變長；伺服器成本；需要 hydration 才能互動
SSG（static site generation）	建置時預先產生，部署成靜態檔	TTFB 最低（CDN 命中）；零伺服器負載	內容更新要重新建置；頁面數多時建置久；無法個人化
ISR (incremental static regeneration)	第一次請求時產生並快取，過期後背景重新產生（SWR 的 HTML 版）	SSG 的速度 + 可定期更新；可按需清除	使用者可能看到略舊的內容；平台綁定較深
串流 SSR (streaming)	伺服器先送出外殼，慢的區塊準備好再逐段送	TTFB 與 FCP 不被最慢的資料拖累；可配合 selective hydration	代理 / CDN 緩衝會破功；狀態碼與標頭必須先決定
Islands / partial hydration	SSR 全頁，但只有標記為互動的元件才載入 JS 並 hydrate	JS 量大幅下降，INP 好	元件之間共享狀態較麻煩
RSC（React Server Components）	元件在伺服器執行並序列化，只有 client component 送 JS	零 bundle 的資料元件；與串流、Suspense 整合	心智模型新；需要框架（Next.js 等）支援

表 9.1: 主要渲染策略的比較

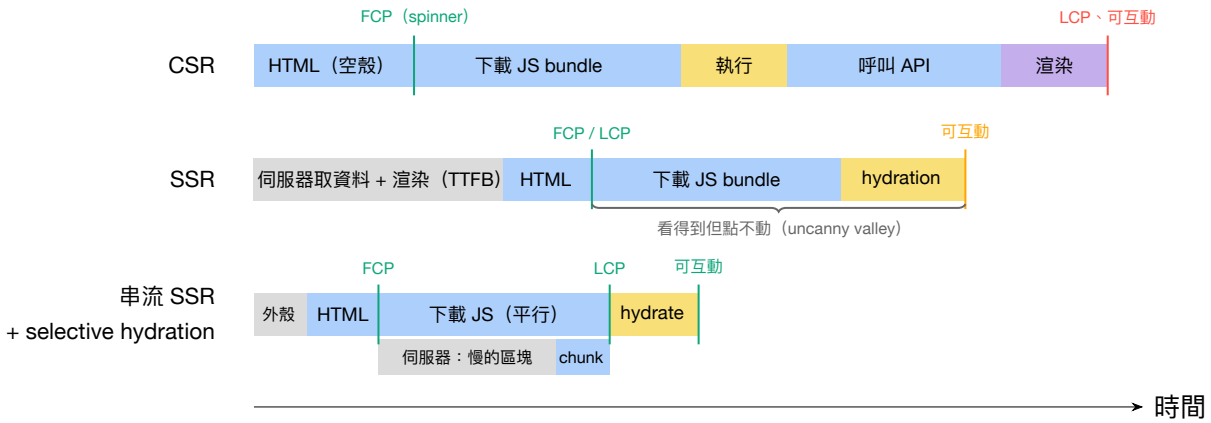


圖 9.1: 三種策略的載入時間線（示意）。CSR 的內容要等「JS → API →渲染」瀑布；SSR 把內容提前但多了 hydration 的等待；串流 SSR 讓伺服器與瀏覽器平行工作。

- **CLS 通常改善**：內容一開始就在 HTML 中，較少「先空再填」的跳動；但若 SSR 與 client 狀態不一致（例如依裝置寬度渲染），反而會在 hydration 時跳動。

9.2　Hydration：SSR 頁面的隱形成本

hydration（水合）

瀏覽器收到 SSR 產生的靜態 HTML 後，前端框架下載 JS、重建元件樹、把事件監聽器接到既有 DOM 上，讓頁面「活起來」的過程。在 React 中是 `hydrateRoot(container, <App />)`。hydration 必須重新執行所有元件的渲染邏輯（雖然不重建 DOM），成本與元件數量成正比。

9.2.1　為什麼 hydration 拖慢 INP

- 傳統的 hydration 是**一整棵樹同步完成**，大型頁面可能是 300–1000 ms 的單一長任務；期間任何點擊都要排隊（輸入延遲）。
- 使用者**看到內容的當下**最想點，正好撞上這段時間——SSR 讓 FCP 提前，也把「看得到但點不動」的窗口拉開。
- hydration 前的互動可能**遺失**（HTML 上的按鈕沒有監聽器），或在 hydration 後才被重播。

9.2.2　減輕 hydration 的技術

```
// 進入點（client）
performance.mark('hydration-start');
hydrateRoot(document.getElementById('root'), <App />);
// React 18+ 的 hydrateRoot 是非同步 / 可中斷的，用 requestIdleCallback 或
// 首個 useEffect 標記結束會更準確；這裡用簡化版
requestAnimationFrame(() => {
  performance.mark('hydration-end');
  const m = performance.measure('hydration', 'hydration-start', 'hydration-end');
  console.log('hydration (ms) ', m.duration.toFixed(0));
});

// 觀察 hydration 期間有沒有長任務（會拖慢 INP）
new PerformanceObserver((list) => {
  for (const e of list.getEntries()) {
    if (e.duration > 100) console.warn('長任務', e.duration.toFixed(0), 'ms', e.
    scripts?.[0]?.sourceURL);
```

技術	做法	代表
Code splitting + lazy	不在首屏的元件用 `lazy()` / 動態 `import()`，hydration 只處理已載入的部分。	所有框架
Selective hydration	用 `<Suspense>` 切分，各區塊可獨立、非同步地 hydrate；使用者點擊某區塊時優先 hydrate 該區塊並重播事件。	React 18+
Progressive / lazy hydration	元件進入視窗（`IntersectionObserver`）、閒置時（`requestIdleCallback`）或互動時才 hydrate。	Astro `client:visible` / `client:idle`、Vue 3.5 lazy hydration、Nuxt
Islands architecture	頁面預設是零 JS 的靜態 HTML，只有標記為互動的「島」才送 JS 並 hydrate。	Astro、Fresh、Marko
Server Components	純展示、取資料的元件只在伺服器執行，不送 JS 到瀏覽器；client component 才需要 hydrate。	React Server Components（Next.js App Router）
Resumability	不做 hydration：在 HTML 中序列化必要狀態與監聽器位置，互動時才載入對應的極小程式碼。	Qwik

表 9.2: 減輕 hydration 成本的主要技術

```
  }
}).observe({ type: 'long-animation-frame', buffered: true });
```

程式碼 9.1: 量測 hydration 的時間與長任務

9.3　串流 SSR 與 Suspense

傳統 SSR 是「等全部資料到齊→渲染整頁→一次送出」，最慢的那個資料源決定了 TTFB。串流 SSR 把 HTML 拆成多段：先送出外殼與已就緒的部分（TTFB 只取決於外殼），慢的區塊用 `<Suspense>` 包起來，先送 fallback，資料到了再把內容當成後續 chunk 送到，並用一小段內嵌 script 把它替換到正確位置。

```
import express from 'express';
import { renderToPipeableStream } from 'react-dom/server';
import App from './App.jsx';
```

```
const app = express();

app.get('*', (req, res) => {
  let didError = false;
  const { pipe, abort } = renderToPipeableStream(<App url={req.url} />, {
    bootstrapScripts: ['/assets/client.3f2a1b.js'],   // hydration 用的 JS，會加在
   HTML 尾端
    onShellReady() {
      // 外殼（Suspense 邊界以外的部分）已就緒：決定狀態碼與標頭，開始送
      res.statusCode = didError ? 500 : 200;
      res.setHeader('Content-Type', 'text/html; charset=utf-8');
      res.setHeader('Cache-Control', 'private, no-cache');
      pipe(res);
    },
    onShellError(err) {
      // 外殼本身出錯：回退到 CSR 的空殼或錯誤頁
      res.statusCode = 500;
      res.send('<!doctype html><p>載入失敗，請重試</p>');
    },
    onError(err) {
      didError = true;   // Suspense 邊界內的錯誤：該區塊會退回 client 端渲染
      console.error(err);
    },
  });
  setTimeout(abort, 10_000);   // 最多等 10 秒，之後剩餘區塊交給 client 完成
});
```

程式碼 9.2: Node / Express 上的 React 串流 SSR

```
export default function App({ url }) {
  return (
    <html lang="zh-Hant">
      <head>
        <meta charSet="utf-8" />
        <link rel="stylesheet" href="/assets/main.9c0d.css" />
        {/* LCP 圖片：讓 preload scanner 在 HTML 一到時就發現它 */}
        <link rel="preload" as="image" href="/img/hero-800.avif" fetchPriority="
   high" />
      </head>
      <body>
        <Header />                          {/* 外殼：立刻送出 */}
        <Hero />                            {/* 外殼：含 LCP 圖片 */}
        <Suspense fallback={<ReviewsSkeleton />}>
```

```
          <Reviews productId={url} />     {/* 慢的 API：資料到了再串流補上 */}
        </Suspense>
        <Suspense fallback={null}>
          <Recommendations />             {/* 更慢：不影響上面任何東西 */}
        </Suspense>
      </body>
    </html>
  );
}
```

程式碼 9.3: App.jsx：慢的區塊用 Suspense 包起來

串流會被誰吃掉

串流的前提是每一段 HTML 都能**立刻**抵達瀏覽器。以下任一環節都會把 chunk 攢起來一次送出，讓串流形同虛設：

- **反向代理緩衝**：nginx 預設 `proxy_buffering on`。關閉它，或讓 origin 回 `X-Accel-Buffering: no`。
- **壓縮中介層**：gzip / Brotli 會累積到一定大小才輸出。Node 的 `compression` 中介層需要在每個 chunk 後 `res.flush()`；多數框架（Next.js）已處理。
- **CDN / WAF**：有的產品預設緩衝整個回應以便掃描或壓縮；確認「streaming / chunked response」支援。
- **Serverless 平台**：部分 FaaS 只支援一次性回應；請選擇支援 streaming response 的執行環境。

驗證方法：DevTools Network 點該 HTML 請求，Timing 分頁中 *Waiting (TTFB)* 應該很短、*Content Download* 拉長；或用 `curl -N` 觀察是否逐段輸出。

```
location / {
  proxy_pass http://node_app;
  proxy_http_version 1.1;
  proxy_buffering off;          # 不要攢 chunk
  proxy_cache off;
  gzip on;                      # nginx 的 gzip 會逐 chunk 壓縮，沒問題
}
```

程式碼 9.4: nginx：讓串流 SSR 通過

9.4　SSG、ISR 與「把 SSR 結果快取起來」

SSR 每次請求都渲染，是最貴的做法。若頁面對所有匿名使用者都相同，就該把結果存起來：

- **SSG**：建置時產生全部頁面，部署到 CDN。TTFB 幾乎等於 CDN 邊緣命中（第八章）。適合行銷頁、文件、部落格。
- **ISR / 按需再生**：以 Next.js 為例，`fetch(url, { next: { revalidate: 60 } })` 或路由層的 `export const revalidate = 60`，讓頁面在 CDN 上以 SWR 的方式更新；內容變動時呼叫 `revalidatePath()` / `revalidateTag()` 立即清除。這正是第六章與第八章講的 `stale-while-revalidate` + purge by tag，只是由框架幫你做。
- **在 CDN 快取 SSR 的 HTML**：不靠框架，直接對 SSR 回應加 `Cache-Control: public, s-maxage=60, stale-while-revalidate=600`，並依 cookie 分流登入者。
- **Partial Prerendering（PPR）**：Next.js 發展中的模式：同一頁面的靜態外殼在建置時預先產生、可放 CDN，動態區塊以 Suspense 邊界在請求時串流補上——把 SSG 的 TTFB 與 SSR 的動態性合在一頁。

```
// app/products/[id]/page.tsx
import { Suspense } from 'react';

export const revalidate = 300;   // 此頁的靜態部分每 5 分鐘於背景再生（ISR）

export default async function ProductPage({ params }) {
  // 商品基本資料：可快取，並打上 tag 供後台更新時清除
  const product = await fetch(`https://api.example.com/products/${params.id}`, {
    next: { revalidate: 300, tags: [`product-${params.id}`] },
  }).then((r) => r.json());

  return (
    <main>
      <ProductHero product={product} />     {/* 含 LCP 圖片，走 next/image 的
    priority */}
      <Suspense fallback={<PriceSkeleton />}>
        <LivePrice id={params.id} />        {/* 個人化 / 即時：cache: 'no-store'，串
    流補上 */}
      </Suspense>
      <Suspense fallback={<ReviewsSkeleton />}>
        <Reviews id={params.id} />
      </Suspense>
    </main>
  );
```

```
}
```

程式碼 9.5: Next.js App Router：靜態外殼 + 快取資料 + 串流的動態區塊

9.5　為每種頁面選策略

頁面類型	特性	建議策略
行銷頁、文件、部落格	所有人相同、更新不頻繁	SSG（或 ISR），放 CDN；islands 只為少數互動元件送 JS。
商品列表 / 分類頁	匿名者相同、更新頻繁、SEO 重要	ISR 或 CDN 快取的 SSR（s-maxage + SWR + purge by tag）。
商品頁	大部分靜態 + 少量即時（價格、庫存、購物車）	靜態外殼 SSG / ISR，即時區塊以 Suspense 串流或 client 端補上（PPR 模式）。
搜尋結果	查詢組合無限、需要新鮮	串流 SSR（外殼立即、結果串流）；熱門查詢可短 TTL 快取。
登入後儀表板 / 後台	個人化、互動密集、SEO 無關	CSR + app shell（第七章）或 SSR 只做外殼；重點在 code splitting 與 INP。
結帳流程	高互動、安全性高、不可快取	SSR 外殼 + 少量 JS；不要 prerender；`private, no-store`。

表 9.3: 依頁面類型選擇渲染策略

選策略的三個問題

1. **這頁對所有匿名使用者是否相同？**是→預先產生並放 CDN（SSG / ISR）；只有部分不同→靜態外殼 + 動態區塊。
2. **內容多久變一次、變了之後多快要被看到？**分鐘級→ SWR / ISR；即時→該區塊不快取，用串流補上。
3. **互動有多重？**重→想辦法少送 JS（islands、RSC、code splitting），並量 INP；輕→ SSR / SSG 幾乎沒有 hydration 問題。

9.6　診斷 SSR 頁面

```
<!doctype html>
<html lang="zh-Hant">
```

症狀	檢查方式與常見原因
TTFB 很長（> 800 ms）	Navigation Timing 的 `responseStart − requestStart`；伺服器端用 `Server-Timing: db;dur=320, render;dur=180` 拆解。原因：資料源慢、未快取、沒有串流、伺服器在遠端區域、冷啟動。
TTFB 短但 FCP 晚	HTML 到了卻沒東西畫：檢查是否 render-blocking CSS / 同步 JS、或 HTML 其實是空殼（view-source 看有沒有內容）。
LCP 晚但 FCP 早	LCP 圖片是否在 HTML 中直接出現且有 `fetchpriority="high"`？是否被 client 端才決定的 `src`（例如依 JS 量寬度）延後？
INP 差、集中在載入初期	Performance 面板看 hydration 的長任務；Interactions 軌道顯示大量輸入延遲。解法：Suspense 切分、lazy hydration、減少首屏元件、RSC。
CLS 出現在 hydration 時	SSR 與 client 初次渲染不一致（hydration mismatch 警告）；依 `window.innerWidth` 或 `localStorage` 決定的 UI 應改為 CSS 或延後渲染。
串流沒效果	`curl -N` 看是否逐段輸出；檢查 nginx buffering、壓縮、CDN 設定。

表 9.4: SSR 頁面常見症狀與檢查

```
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>商品名稱</title>
  <!-- 關鍵 CSS 內嵌，避免 render-blocking 的往返 -->
  <style>/* critical css，約 10 KB 以內 */</style>
  <!-- LCP 圖片與字型：在 HTML 最前面宣告 -->
  <link rel="preload" as="image" href="/img/hero-800.avif" imagesrcset="/img/hero
    -480.avif 480w, /img/hero-800.avif 800w" imagesizes="100vw" fetchpriority="high"
    >
  <link rel="preload" as="font" type="font/woff2" href="/fonts/brand.woff2"
    crossorigin>
  <!-- 其餘 CSS 非同步、JS 延後 -->
  <link rel="stylesheet" href="/assets/rest.9c0d.css" media="print" onload="this.
    media='all'">
  <script type="module" src="/assets/client.3f2a1b.js" defer></script>
</head>
<body>
```

```
<!-- 真正的內容直接在 HTML 中，不是 <div id="root"></div> -->
<main>
  <h1>商品名稱</h1>
  <img src="/img/hero-800.avif" width="800" height="450" alt="..." fetchpriority=
 "high">
  ...
```

程式碼 9.6: 一份「好的」SSR HTML 開頭長什麼樣

9.7　課堂實作

實作 9-1：同一個頁面的三種寫法

用 Vite + React 建一個顯示商品清單（含一張大圖）的頁面，分別實作：(a) CSR：空殼 + `useEffect` 取 API；(b) SSR：Express + `renderToString`；(c) 串流 SSR：`renderToPipeableStream` + Suspense，API 故意延遲 1.5 秒。用 Performance 面板 (Slow 4G、CPU 4x)錄三次，填表比較 TTFB、FCP、LCP、最長任務、Interactions 的輸入延遲。

實作 9-2：找出 hydration 長任務並拆掉

在 9-1 的 (b) 版本加入 500 個含事件處理的列表項目。錄製後在 Main 軌道找到 hydration 的長任務與 Bottom-Up 中的函式。依序嘗試：列表用 `<Suspense>` 包起來、改用 `lazy()` 分割、列表項改為 server component（若用 Next.js）。每次量 INP 的變化。

實作 9-3：讓串流通過代理

在 9-1 (c) 前面架一個預設設定的 nginx，觀察串流是否消失（Network Timing 的 Waiting 變長、`curl -N` 一次吐出）。加上 `proxy_buffering off` 後再測。若有 CDN 帳號，把它放在最前面再驗證一次。

思考題

1. 為什麼 SSR 的 TTFB 通常比 CSR 差，LCP 卻通常更好？有沒有 SSR 的 LCP 反而更差的情況？
2. 什麼是 uncanny valley？它對應到哪個 Web Vitals 指標？列出三種縮短它的技術。
3. 串流 SSR 為什麼必須在 `onShellReady` 時就決定 HTTP 狀態碼？如果 Suspense 邊界內的資料回傳 404，該怎麼處理？
4. ISR 與 `Cache-Control: s-maxage + stale-while-revalidate` 的關係是什麼？

5. 一個電商商品頁，價格與庫存每分鐘變動，其餘內容每週才變。請設計渲染與快取策略，並說明價格變動時發生什麼事。
6.「把 SSR 搬到邊緣（edge）一定更快」——請舉一個反例。

本章回顧

- CSR 的內容要等「JS → API →渲染」瀑布，LCP 最晚；SSR 把內容放進 HTML，FCP / LCP 提前，但 TTFB 變長並引入 hydration 成本。
- hydration 是 SSR 頁面 INP 的主要風險：用 Suspense / selective hydration、lazy hydration、islands、Server Components、code splitting 減少要 hydrate 的東西。
- 串流 SSR 讓外殼先到、慢區塊後補；記得排除代理、壓縮、CDN 的緩衝，並用 `curl -N` 驗證。
- 對匿名者相同的頁面不要每次 SSR：SSG / ISR / CDN 快取 + SWR + purge by tag，讓 TTFB 回到邊緣命中的水準；動態區塊用 Suspense 或 client 補上。
- 診斷 SSR 頁面時，用 Navigation Timing 拆 TTFB、view-source 確認內容真的在 HTML 中、Performance 面板找 hydration 長任務。

附錄 A　詞彙表、速查表、綜合練習與參考資料

A.1　詞彙表

名詞	說明	章
RTT（Round-Trip Time）	封包從客戶端到伺服器再回來的時間；網路效能的基本單位。由距離與路由決定。	一
TCP 三向交握	建立 TCP 連線需要 1 RTT（SYN、SYN-ACK、ACK）。	一
TLS 交握	建立加密通道。TLS 1.3 需 1 RTT，0-RTT 恢復時為 0；TLS 1.2 需 2 RTT。	一
QUIC / HTTP/3	以 UDP 為基礎、內建 TLS 1.3 的傳輸協定；交握合併為 1 RTT，無隊頭阻塞，連線遷移。	一
TCP 慢啟動（slow start）	新連線的擁塞視窗從約 10 個封包（約 14 KB）開始，每個 RTT 加倍；大檔前幾個 RTT 傳得慢。	一
preconnect / dns-prefetch	提前對第三方來源完成 DNS + TCP + TLS，或只做 DNS。	一
103 Early Hints	伺服器在最終回應前先送出的暫時回應，帶 `Link` 標頭讓瀏覽器提早下載資源。	一, 八
關鍵渲染路徑（CRP）	從 HTML 到第一次繪製所必經的步驟：HTML → DOM、CSS → CSSOM、render tree、layout、paint。	一
preload scanner	HTML 解析器被阻擋時，仍先掃描後續標記並提早發出資源請求的機制。	一, 三
render-blocking / parser-blocking	阻擋渲染的 CSS 與阻擋解析的同步 JS。	一
TTFB（Time to First Byte）	從導覽開始到 HTML 第一個位元組抵達；含重導、DNS、連線、伺服器處理。良好 $\leq$ 800 ms。	二
FP / FCP	第一次繪製任何像素 / 第一次繪製內容（文字、圖片）。FCP 良好 $\leq$ 1.8 s。	二

（續下頁）

名詞	說明	章
FMP（First Meaningful Paint）	已棄用的「主要內容繪製」啟發式指標，由 LCP 取代。	二
LCP（Largest Contentful Paint）	視窗內最大內容元素完成渲染的時間。良好 $\leq$ 2.5 s。	二, 三
FID（First Input Delay）	第一次互動的輸入延遲；2024 年 3 月起由 INP 取代。	二
INP（Interaction to Next Paint）	整個頁面生命週期中互動延遲的近似最大值（排除離群值）。良好 $\leq$ 200 ms。	二, 三
CLS（Cumulative Layout Shift）	非預期版面跳動的累計分數（最大 session window）。良好 $\leq$ 0.1。	二, 三
TBT / TTI / Speed Index	實驗室指標：長任務超過 50 ms 的總和 / 可互動時間（已從 Lighthouse 移除）/ 內容填滿視窗的速度。	二
長任務（long task）	佔用主執行緒超過 50 ms 的任務；期間無法回應輸入。	三
LoAF（Long Animation Frames）	超過 50 ms 的動畫幀，含腳本歸因；比 longtask 更適合診斷 INP。	三, 四
實驗室 / 現場資料（lab / field）	受控環境的量測（Lighthouse、DevTools）/ 真實使用者的量測（CrUX、RUM）。	導論
p75	第 75 百分位數；Core Web Vitals 以 28 天內 p75 評定。	導論, 二
CrUX（Chrome UX Report）	Google 從 Chrome 使用者收集的公開現場資料集。	導論, 五
PerformanceObserver	訂閱效能條目的 API；`buffered: true` 可取回訂閱前的條目。	四
Navigation Timing	整份文件的載入時間線（Level 2：`PerformanceNavigationTiming`）。	四
Resource Timing	每個子資源的時間線與大小（`PerformanceResourceTiming`）。	四
Timing-Allow-Origin (TAO)	允許跨來源資源揭露詳細時間與大小的回應標頭。	四
Server-Timing	伺服器把處理階段耗時寫進回應標頭，前端可讀取。	四, 八
新鮮度 / 驗證	HTTP 快取的兩個概念：有效期內直接用；過期後以 ETag / Last-Modified 條件式請求驗證。	六
stale-while-revalidate (SWR)	過期後先用舊版、同時背景更新的指令；ISR 是它的 HTML 版。	六, 八, 九

（續下頁）

名詞	說明	章
bfcache	往返快取：把整頁狀態凍結在記憶體，上一頁 / 下一頁瞬間還原。	六
Speculation Rules	宣告式地 prefetch / prerender 下一頁的 API。	六
Service Worker	頁面與網路之間的可程式化代理執行緒。	七
Cache Storage	SW 可用的 Request → Response 鍵值儲存。	七
app shell	預快取的應用程式外殼（HTML / CSS / JS），讓回訪與離線瞬間可見。	七
navigation preload	在 SW 啟動的同時平行送出導覽請求。	七
CDN / PoP / 邊緣節點	內容傳遞網路 / 分布各地的機房 / 其中的伺服器。	八
origin shield	邊緣與 origin 之間的區域快取層，減少回源。	八
cache key	CDN 辨認同一資源的鍵（URL + 部分標頭）。	八
快取命中率（CHR）	命中 ÷ 總請求。	八
purge by tag	依標籤一次清除多個 URL 的快取。	八
CSR / SSR / SSG / ISR	客戶端渲染 / 伺服器端渲染 / 靜態產生 / 增量靜態再生。	九
hydration	讓 SSR 產生的靜態 HTML 接上事件與狀態、變得可互動的過程。	九
串流 SSR / Suspense	分段送出 HTML，慢區塊之後補上。	九
islands / RSC / resumability	減少 hydration 的三種架構：只 hydrate 互動島 / 元件留在伺服器 / 不 hydrate 直接恢復。	九

A.2　指標門檻速查

A.3　綜合練習：一次完整的效能稽核

以下流程把各章串起來，建議作為期末作業：挑一個真實網站(最好是你能改動的)，完成後產出一份 3–5 頁的報告。

1. **現場資料先行**（導論、第五章）：在 PageSpeed Insights 或 DevTools Live Metrics 的 Field data 看 28 天 p75 的 LCP、INP、CLS、TTFB，區分行動 / 桌機。若沒有 CrUX 資料，改用自家 RUM 或說明限制。
2. **校準實驗室環境**：依 CrUX 建議設定 CPU / 網路節流；決定冷 / 熱快取；用無痕視窗。記錄環境。
3. **fetch 之前**（第一章）：用 Navigation Timing 拆解重導、DNS、連線、TLS、SW 啟

指標	良好	待改善	不佳	備註
LCP	$\leq$ 2.5 s	2.5–4.0 s	$>$ 4.0 s	Core Web Vital
INP	$\leq$ 200 ms	200–500 ms	$>$ 500 ms	Core Web Vital (2024/3 取代 FID)
CLS	$\leq$ 0.1	0.1–0.25	$>$ 0.25	Core Web Vital
TTFB	$\leq$ 800 ms	800–1800 ms	$>$ 1800 ms	診斷指標
FCP	$\leq$ 1.8 s	1.8–3.0 s	$>$ 3.0 s	診斷指標
FID	$\leq$ 100 ms	100–300 ms	$>$ 300 ms	歷史指標
TBT（實驗室）	$\leq$ 200 ms	200–600 ms	$>$ 600 ms	Lighthouse

表 A.2: 門檻速查（現場資料以 28 天 p75 評定）

子部分	健康的比例 / 目標
LCP：TTFB	應佔 LCP 的大約 40%；絕對值 $\leq$ 800 ms。
LCP：資源載入延遲	目標接近 0：LCP 圖片應在 HTML 中直接可見並 `fetchpriority="high"`。
LCP：資源載入時間	約 40%；靠壓縮格式（AVIF / WebP）、正確尺寸、CDN。
LCP：元素渲染延遲	目標接近 0：不要讓 render-blocking 資源或 JS 延後渲染。
INP：輸入延遲	主執行緒忙碌造成；目標 $<$ 50 ms。拆長任務、延後非必要工作。
INP：處理時間	事件處理器本身；目標 $<$ 100 ms。先更新畫面再做重活。
INP：呈現延遲	渲染下一幀；避免大量 DOM 變更與強制同步版面。
CLS	圖片 / 影片給尺寸；為動態內容預留空間；字型用 `size-adjust` 與 `font-display`。

表 A.3: 子部分目標速查

動、TTFB；用 Network 面板確認協定（h2 / h3）、是否有多餘的重導、第三方來源的連線成本。

4. **LCP**（第三章）：找出 LCP 元素；用 LCP breakdown 看四個子部分；檢查資源是否可被 preload scanner 發現、`fetchpriority`、格式與尺寸、render-blocking 資源。
5. **INP**（第三章）：用 Live Metrics 與 Interactions 軌道找出最慢的互動；拆解三個子部分；定位到函式；提出拆分 / 延後 / 減少 DOM 的方案。
6. **CLS**（第三章）：用 Layout shifts 軌道與 culprits 找出元素與原因（無尺寸圖片、動態插入、字型）。
7. **快取**（第六章）：盤點 Cache-Control；用 Resource Timing 算命中率；測 bfcache 資格；評估是否適合 Speculation Rules。
8. **Service Worker**（第七章）：若有 SW，量 `workerStart` 成本、檢查策略是否合理、有沒有空的 fetch handler；若沒有，評估是否值得加。
9. **CDN**（第八章）：用 curl 判讀命中狀態、Age、s-maxage；HTML 是否被邊緣快取；影像是否經過影像 CDN；是否啟用 HTTP/3、Brotli、Early Hints。
10. **渲染策略**（第九章）：view-source 看內容是否在 HTML 中；量 TTFB 與 hydration 長任務；判斷目前策略是否合適，是否該改為 SSG / ISR / 串流。
11. **提出與驗證修正**：每項修正註明預期影響的指標與子部分；能改的就改，改完在相同環境重測；不能改的估算效益（例如「LCP 圖片改 AVIF 可省 60% 位元組，資源載入時間預估從 900 ms 降到 400 ms」）。
12. **監測**：加入 `web-vitals` attribution 版回報；規劃儀表板與告警門檻。

項目	現場 p75	實驗室	主要原因 / 修正 / 預期效果
TTFB			
LCP（元素 :)			
INP（互動 :)			
CLS（元素 :)			
快取命中率			
bfcache 資格			
CDN 命中 / 協定			
渲染策略			

表 A.4: 稽核報告摘要表範本

A.4　參考資料

以下資料皆為官方文件或規範，是本講義內容的主要依據；學習時建議以這些來源為準，因為指標定義與瀏覽器行為每年都會更新。

Web Vitals 與指標

- web.dev —Web Vitals 總覽：https://web.dev/articles/vitals
- LCP：https://web.dev/articles/lcp；優化：https://web.dev/articles/optimize-lcp
- INP：https://web.dev/articles/inp；優化：https://web.dev/articles/optimize-inp
- CLS：https://web.dev/articles/cls；優化：https://web.dev/articles/optimize-cls
- TTFB：https://web.dev/articles/ttfb；優化：https://web.dev/articles/optimize-ttfb
- FCP：https://web.dev/articles/fcp；FID（歷史）：https://web.dev/articles/fid；TBT：https://web.dev/articles/tbt
- Long Animation Frames API：https://developer.chrome.com/docs/web-platform/long-animation-frames
- web.dev Learn Performance（免費課程）：https://web.dev/learn/performance
- CrUX：https://developer.chrome.com/docs/crux；PageSpeed Insights：https://pagespeed.web.dev/

Performance API 規範與文件

- MDN Performance APIs：https://developer.mozilla.org/en-US/docs/Web/API/Performance_API
- Navigation Timing Level 2：https://www.w3.org/TR/navigation-timing-2/
- Resource Timing：https://www.w3.org/TR/resource-timing/
- Performance Timeline：https://www.w3.org/TR/performance-timeline/
- Paint Timing：https://www.w3.org/TR/paint-timing/
- Largest Contentful Paint：https://w3c.github.io/largest-contentful-paint/
- Event Timing：https://www.w3.org/TR/event-timing/
- Layout Instability：https://wicg.github.io/layout-instability/
- User Timing：https://www.w3.org/TR/user-timing/；Server Timing：https://www.w3.org/TR/server-timing/
- web-vitals 函式庫：https://github.com/GoogleChrome/web-vitals

連線、關鍵渲染路徑與資源提示

- High Performance Browser Networking（Ilya Grigorik，免費線上版）：https://hpbn.co/
- 關鍵渲染路徑：https://web.dev/learn/performance/understanding-the-critical-path
- preconnect 與 dns-prefetch：https://web.dev/articles/preconnect-and-dns-prefetch
- Fetch Priority：https://web.dev/articles/fetch-priority
- 103 Early Hints：https://developer.chrome.com/docs/web-platform/early-hints；RFC 8297：https://www.rfc-editor.org/rfc/rfc8297

Chrome DevTools

- Performance 面板總覽：https://developer.chrome.com/docs/devtools/performance
- Performance 面板參考：https://developer.chrome.com/docs/devtools/performance/reference
- Lighthouse：https://developer.chrome.com/docs/lighthouse/overview
- WebPageTest：https://www.webpagetest.org/

快取、bfcache 與 Speculation Rules

- MDN HTTP caching：https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching
- RFC 9111 HTTP Caching：https://www.rfc-editor.org/rfc/rfc9111；RFC 5861 stale-while-revalidate：https://www.rfc-editor.org/rfc/rfc5861
- HTTP cache partitioning：https://developer.chrome.com/blog/http-cache-partitioning
- bfcache：https://web.dev/articles/bfcache；Cache-Control: no-store 與 bfcache：https://developer.chrome.com/docs/web-platform/bfcache-ccns
- unload 事件棄用：https://developer.chrome.com/docs/web-platform/deprecating-unload
- Speculation Rules（prerender pages）：https://developer.chrome.com/docs/web-platform/prerender-pages

Service Worker

- MDN Service Worker API：https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
- Workbox：https://developer.chrome.com/docs/workbox
- Navigation preload：https://web.dev/blog/navigation-preload

- Service Worker Static Routing API：https://developer.chrome.com/blog/service-worker-static-routing
- Cache Storage（MDN）：https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage

CDN

- Cloudflare Learning Center —What is a CDN：https://www.cloudflare.com/learning/cdn/what-is-a-cdn/
- RFC 9211 Cache-Status：https://www.rfc-editor.org/rfc/rfc9211；RFC 9213 Targeted Cache-Control：https://www.rfc-editor.org/rfc/rfc9213
- web.dev —Content delivery networks：https://web.dev/articles/content-delivery-networks

渲染策略與 SSR

- Rendering on the Web：https://web.dev/articles/rendering-on-the-web
- Patterns.dev —Rendering Patterns：https://www.patterns.dev/
- React `renderToPipeableStream`：https://react.dev/reference/react-dom/server/renderToPipeableStream
- React `hydrateRoot`：https://react.dev/reference/react-dom/client/hydrateRoot
- Next.js 文件（Rendering、Caching、ISR）：https://nextjs.org/docs
- Astro Islands：https://docs.astro.build/en/concepts/islands/

持續更新

Web 效能領域變動很快：INP 在 2024 年取代 FID、bfcache 對 `no-store` 的放寬、Speculation Rules 與 Static Routing API 都是近兩年的事。建議每半年回到 web.dev 與 Chrome for Developers 的更新頁面檢視一次，並訂閱 Chrome 的 release notes。