Leaflet 學習入門

講師：林新德 shinder.lin@gmail.com

本筆記搭配 `public/` 中的 HTML 檔案，提供每個範例的重點解說。

L010: 基本地圖設置

這個範例 `public/L010-basic.html` 展示了如何初始化一個基本地圖、添加圖層和標記。

1. 引入 Leaflet 函式庫

首先，我們需要在 HTML 的 `<head>` 中引入 Leaflet 的 CSS 和 JavaScript 檔案。

```
<link rel="stylesheet" href="https://unpkg.com/leaflet/dist/leaflet.css" />
<script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
```

2. 建立地圖容器

我們需要一個 `<div>` 元素來放置地圖。記得要為它設定一個 ID 和高度。

```
<div id="map"></div>

#map {
  height: 500px;
  width: 100%;
}
```

3. 初始化地圖

- `L.map("map")`: 這會初始化一個地圖，並將它綁定到 ID 為 `map` 的 `<div>` 元素上。
- `.setView([緯度, 經度], 縮放層級)`: 這會設定地圖的中心點和縮放層級。
    - **座標**: Leaflet 使用 `[緯度, 經度]` 的格式。
    - **縮放層級 (Zoom Level)**: 數字越大，地圖越放大（越詳細）。層級 13 大約是城市街區的尺度。

```
const taipei101 = [25.033, 121.5654]; // [緯度, 經度]
const map = L.map("map").setView(taipei101, 13);
```

4. 添加圖磚圖層 (Tile Layer)

地圖本身只是一個空的容器，我們需要添加一個圖層來顯示地圖影像。最常見的是使用 OpenStreetMap (OSM) 的圖磚。

- `L.tileLayer(URL, { ...options })`: 建立一個圖磚圖層。
    - **URL**: 圖磚服務的網址。`{s}`, `{z}`, `{x}`, `{y}` 是 Leaflet 會自動替換的變數。
    - **options**:
        - `maxZoom`: 此圖層支援的最大縮放層級。
        - `attribution`: 版權宣告，會顯示在地圖右下角。

- `.addTo(map)`: 將這個圖層加到我們建立的 `map` 物件中。

```
L.tileLayer("https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png", {
  maxZoom: 19,
  attribution:
    '© <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>',
}).addTo(map);
```

5. 添加標記 (Marker)

我們可以使用 `L.marker()` 在地圖上新增一個標記。

```
const marker = L.marker(taipei101).addTo(map);
```

L020: 地圖標示

這個範例 `public/L020-labels.html` 延伸了前一個範例，加入了更多地圖標示的功能。

1. 自訂圖示 (Custom Icon)

Leaflet 允許你使用自己的圖片作為標記圖示。

- `L.icon({ ...options })`: 建立一個圖示物件。
  - `iconUrl`: 圖示圖片的 URL。
  - `iconSize`: 圖示的尺寸 `[寬, 高]` (px)。
  - `iconAnchor`: 圖示的「錨點」，也就是圖示上對應到地圖座標的那個點。`[20, 40]` 表示錨點在圖示寬度一半、高度最底下的地方。
  - `popupAnchor`: 彈出視窗 (Popup) 相對於 `iconAnchor` 的位置。

```
const customIcon = L.icon({
  iconUrl: "./images/mario.gif",
  iconSize: [40, 40],
  iconAnchor: [20, 40],
  popupAnchor: [0, -34],
});
```

然後在建立 `L.marker` 時，將這個 `customIcon` 物件傳入。

```
const marker = L.marker(taipei101, { icon: customIcon }).addTo(map);
```

2. 綁定彈出視窗 (Popup)

你可以為標記加上一個點擊後會出現的彈出視窗。

- `.bindPopup("...")`: 將一段 HTML 內容綁定到標記上。
- `.openPopup()`: 讓這個彈出視窗預設就是開啟的狀態。

```
const marker = L.marker(taipei101, { icon: customIcon })
  .addTo(map)
  .bindPopup("<b>台北 101</b><br>台北市信義區")
  .openPopup();
```

3. 加入比例尺

使用 `L.control.scale()` 可以輕鬆在地圖上加入比例尺。

- `metric`: `true` 表示使用公制單位 (m/km)。
- `imperial`: `false` 表示不使用英制單位 (mi/ft)。

```
const scale = L.control
  .scale({
    metric: true,
    imperial: false,
    maxWidth: 200,
  })
  .addTo(map);
```

4. 繪製圓形區域

除了標記，Leaflet 也可以繪製向量圖形，例如圓形。

- `L.circle([緯度, 經度], { ...options })`: 建立一個圓形。
    - `radius`: 半徑，單位是公尺。
    - `color`: 線條顏色。
    - `fillColor`: 填充顏色。
    - `fillOpacity`: 填充顏色的透明度。

```
const circle = L.circle([25.033, 121.5654], {
  color: "orange",
  fillColor: "yellow",
  fillOpacity: 0.5,
  radius: 500,
}).addTo(map);
```

L030: 取得當前位置

這個範例 `public/L030-current-position.html` 展示如何使用瀏覽器內建的 Geolocation API 來取得使用者的目前位置，並在地圖上標示出來。

> **注意**: 基於瀏覽器的安全策略，Geolocation API 只能在 `localhost` 或 `https` 安全連線的環境下運作。

1. 檢查瀏覽器是否支援

在使用前，最好先檢查 `navigator` 物件中是否存在 `geolocation` 屬性。

```
if ("geolocation" in navigator) {
  // 瀏覽器支援
} else {
  alert("您的瀏覽器不支援地理位置功能");
}
```

2. 取得目前位置

`navigator.geolocation.getCurrentPosition()` 是核心方法，它會非同步地嘗試取得使用者的位置。它需要傳入三個參數：

1. **成功時的回呼函式 (Success Callback)**: 當成功取得位置時執行。
2. **失敗時的回呼函式 (Error Callback)**: 當取得位置失敗時執行。
3. **選項物件 (Options)**: 用來設定取得位置的參數。

```
navigator.geolocation.getCurrentPosition(successCallback, errorCallback, options);
```

3. 成功與失敗處理

- **成功**: 回呼函式會收到一個 `position` 物件，我們可以從 `position.coords.latitude` 和 `position.coords.longitude` 取得緯度和經度。接著，使用 `map.setView()` 將地圖中心移到新位置，並用 `L.marker()` 加上標記。
- **失敗**: 回呼函式會收到一個 `error` 物件，可以印出來除錯。

```
// 成功時
function successCallback(position) {
  const lat = position.coords.latitude;
  const lon = position.coords.longitude;

  // 更新地圖中心與標記
  map.setView([lat, lon], 16);
  L.marker([lat, lon])
    .addTo(map)
```

```
    .bindPopup("您的當前位置")
    .openPopup();
}

// 失敗時
function errorCallback(error) {
  console.error("GPS 錯誤:", error);
}
```

4. 定位選項

我們可以傳遞一個物件來影響定位的精準度與行為。

- `enableHighAccuracy`: `true` 表示要求高精準度的位置（可能會更耗電）。
- `timeout`: 等待位置資訊的最長毫秒數。
- `maximumAge`: `0` 表示不使用快取的位置資訊，強制取得最新的位置。

```
const options = {
  enableHighAccuracy: true,
  timeout: 5000,
  maximumAge: 0,
};
```

L040: 持續偵測當前位置

這個範例 `public/L040-watch-position.html` 延續前一節，改為「持續追蹤」使用者的位置，並在位置變動時更新地圖。

1. `watchPosition` 方法

與 `getCurrentPosition` 只會取得一次位置不同，`navigator.geolocation.watchPosition` 會在裝置位置發生變動時，持續觸發成功的回呼函式。

它的用法和參數與 `getCurrentPosition` 完全相同。

```
navigator.geolocation.watchPosition(successCallback, errorCallback, options);
```

2. 更新標記位置

因為 `watchPosition` 會持續回報新位置，我們不能只是不斷地 `L.marker(...).addTo(map)`，這樣地圖上會充滿無限的標記。

正確的做法是：

1. 用一個變數 (例如 `window.currentLocationMarker`) 來保存目前的標記物件。
2. 每次收到新位置時，先檢查這個變數是否存在。如果存在，就用地圖物件的 `map.removeLayer()` 方法將它從地圖上移除。
3. 建立新的標記，並將它存回同一個變數中，供下次更新使用。

```
function successCallback(position) {
  const lat = position.coords.latitude;
  const lon = position.coords.longitude;

  map.setView([lat, lon], 16);

  // 如果已有標記，先移除
  if (window.currentLocationMarker) {
    map.removeLayer(window.currentLocationMarker);
  }

  // 建立新標記並保存起來
  window.currentLocationMarker = L.marker([lat, lon])
    .addTo(map)
    .bindPopup("您的當前位置")
    .openPopup();
}
```

L050: 切換地圖圖層 (Tile Layers)

這個範例 `public/L050-tile-layers.html` 展示了如何讓使用者在不同的地圖樣式（圖層）之間切換。

1. 準備多個圖層

首先，建立多個 `L.tileLayer` 物件，每個代表一種地圖樣式。除了標準的 OpenStreetMap 之外，還有很多免費或付費的圖磚服務可供選擇，例如 OpenTopoMap、Stamen 等。

```
// 標準 OSM
const osmStandard = L.tileLayer(
  "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
  {
    attribution: "© OpenStreetMap contributors",
  }
);
// 地形圖
const osmTopo = L.tileLayer(
  "https://{s}.tile.opentopomap.org/{z}/{x}/{y}.png",
  {
    attribution: "© OpenTopoMap, © OpenStreetMap contributors",
  }
);
```

2. 建立圖層控制物件

接著，建立一個 JavaScript 物件，它的「鍵」是顯示在選單上的名稱，「值」則是對應的 `L.tileLayer` 物件。

```
const baseMaps = {
  "標準地圖": osmStandard,
  "地形圖": osmTopo,
};
```

3. 加入圖層控制器

使用 `L.control.layers()` 來建立圖層切換的 UI 控制項。

- 第一個參數是「基礎圖層」物件 (Base Layers)，一次只能顯示其中一個 (radio buttons)。
- 第二個參數是「覆蓋圖層」物件 (Overlays)，可以同時顯示多個 (checkboxes)，我們這裡沒有用到。

最後，記得要設定一個預設顯示的圖層，例如 `osmStandard.addTo(map)`。

```
osmStandard.addTo(map); // 設定預設圖層
L.control.layers(baseMaps).addTo(map); // 將控制器加入地圖
```

L060: 名稱搜尋 (地理編碼)

這個範例 `public/L060-name-search.html` 介紹了如何將地名或地址轉換成經緯度座標，這個過程稱為「地理編碼 (Geocoding)」。

這個範例本身沒有使用 Leaflet，它專注於如何從第三方服務取得座標資料。取得座標後，就可以像之前的範例一樣，用 `map.setView()` 或 `L.marker()` 將它應用在地圖上。

1. 使用 Nominatim API

Nominatim 是一個基於 OpenStreetMap 資料的免費地理編碼服務。使用時請遵守其使用政策，主要是不要在短時間內發送大量請求。

2. 使用 `fetch` 進行 API 請求

我們可以使用瀏覽器內建的 `fetch` 功能來向 Nominatim API 發送請求。搭配 `async/await` 可以讓非同步的程式碼看起來更簡潔。

```
async function geocode(address) {
  const url = `https://nominatim.openstreetmap.org/search?format=json&q=${address}
`;

  try {
    const response = await fetch(url);
    const data = await response.json();
    // ... 處理資料
  } catch (error) {
    console.error("地理編碼錯誤:", error);
  }
}
```

- **API URL**: `https://nominatim.openstreetmap.org/search` 是它的端點。
  - `format=json`: 指定回傳 JSON 格式的資料。
  - `q=${address}`: `q` 代表查詢 (query)，後面接上我們要搜尋的地址字串。

3. 處理回傳資料

Nominatim 會回傳一個包含可能結果的陣列。通常，第一個結果 (`data[0]`) 是最相關的。

- 我們需要檢查 `data.length > 0` 來確認有找到結果。
- 結果物件中包含了 `lat` (緯度), `lon` (經度), 和 `display_name` (完整地址名稱) 等資訊。

- 注意 API 回傳的 `lat` 和 `lon` 是字串，最好使用 `parseFloat()` 將它們轉換成數字，才能在 Leaflet 中使用。

```
if (data.length > 0) {
  const result = data[0];
  return {
    lat: parseFloat(result.lat),
    lon: parseFloat(result.lon),
    display_name: result.display_name,
  };
}
```

L070: 路線規劃 (Routing)

這個範例 `public/L070-route-planning.html` 展示了如何串接路線規劃服務，並將計算出的路徑繪製在地圖上。

1. 使用 OSRM API

OSRM (Open Source Routing Machine) 是一個開源的路線規劃引擎，它也提供了免費的公開服務，可以計算兩點或多點間的路徑。

2. OSRM API 請求

與地理編碼類似，我們使用 `fetch` 來呼叫 OSRM API。它的 URL 結構比較特別：

```
const url = `https://router.project-osrm.org/route/v1/driving/${start[1]},${start[0]};${end[1]},${end[0]}?overview=full&geometries=geojson`;
```

- `.../route/v1/driving/`: `driving` 是路線規劃的模式，其他可用模式如 `walking` (步行) 或 `cycling` (自行車)。
- `${start[1]},${start[0]};${end[1]},${end[0]}`: 這是起點和終點的座標。**特別注意**：OSRM 要求 `經度,緯度` 的順序，與 Leaflet 的 `緯度,經度` 相反。
- `?overview=full&geometries=geojson`: 這些是重要的參數。
    - `overview=full`: 表示我們需要完整的路線細節。
    - `geometries=geojson`: 要求回傳的路線幾何資訊是 GeoJSON 格式，這樣 Leaflet 才能直接使用。

3. 處理路線資料

API 回傳的資料中，`data.routes[0]` 物件包含了我們需要的所有資訊。

- `route.geometry`: 這是 GeoJSON 格式的路線幾何，可以直接交給 Leaflet 繪製。
- `route.distance`: 路線的總長度，單位是公尺。
- `route.duration`: 路線的預估時間，單位是秒。

4. 在地圖上繪製路線

Leaflet 對 GeoJSON 有原生支援，因此繪製路線非常簡單。

- `L.geoJSON(geojsonObject, options)`: 建立一個 GeoJSON 圖層。
    - 第一個參數傳入從 OSRM 取得的 `route.geometry`。

- 第二個參數 `options` 中，可以使用 `style` 來設定路線的樣式，例如顏色、寬度、透明度等。

```
getRoute(startPoint, endPoint).then((route) => {
  if (route) {
    // 顯示路線
    L.geoJSON(route.geometry, {
      style: {
        color: "#0080ff",
        weight: 5,
        opacity: 0.8,
      },
    }).addTo(map);

    // 顯示距離和時間
    const distance = (route.distance / 1000).toFixed(1);
    const duration = Math.round(route.duration / 60);
    console.log(`距離: ${distance} km, 時間: ${duration} 分鐘`);
  }
});
```

L080: 反向地址查詢 (Reverse Geocoding)

這個範例 `public/L080-reverse.html` 介紹「反向地理編碼」，也就是將經緯度座標轉換回地址的過程。這在「點擊地圖查詢地點資訊」等情境中非常有用。

1. 監聽地圖點擊事件

Leaflet 的地圖物件可以監聽各種事件，例如 `click`。使用 `map.on('click', callback)` 可以設定當地圖被點擊時要執行的函式。

- 回呼函式會收到一個事件物件 `e`。
- 這個物件的 `e.latlng` 屬性包含了點擊處的地理座標 `{ lat, lng }`。

```
map.on("click", async (e) => {
  const { lat, lng } = e.latlng;
  // ... 接下來的處理
});
```

2. Nominatim 反向查詢 API

我們同樣使用 Nominatim 服務，但這次是呼叫它的 `reverse` 端點。

```
const url = `https://nominatim.openstreetmap.org/reverse?format=json&lat=${lat}&lon=${lon}`;
```

- **API URL**: `.../reverse` 是反向查詢的端點。
- `lat=${lat}&lon=${lon}`: 傳入要查詢的緯度和經度。
- 範例中還額外加了 `&zoom=18&addressdetails=1` 來取得更詳細的地址資訊。

3. 顯示獨立的 Popup

之前我們是將 Popup 綁定在 Marker 上 (`.bindPopup()`)。但 Leaflet 也允許建立獨立的 Popup。

- `L.popup()`: 建立一個新的 Popup 物件。
- `.setLatLng(latlng)`: 設定 Popup 要顯示在哪個座標上。
- `.setContent("...")`: 設定 Popup 內部要顯示的 HTML 內容。
- `.openOn(map)`: 將這個 Popup 開啟在地圖上。

這種方法讓我們可以在沒有 Marker 的情況下，自由地在地圖任何位置顯示資訊窗。

```
const address = await reverseGeocode(lat, lng);

if (address) {
  L.popup()
```

```
    .setLatLng(e.latlng)
    .setContent(`<b>地址：</b>${address.display_name}`)
    .openOn(map);
}
```

L090: 區域搜尋 (Overpass API)

- OSM 的主體是 **OpenStreetMap Foundation (OSMF)**，一個非營利組織。
- `overpass-api.de` 雖然跟 OSM 有關，但不是由 OSMF 官方直接運營，而是 **社群自架服務**，免費對外提供。
- 這意味著：
    - 你可以使用它，但不能保證 SLA（沒有「企業級保證」）。
    - 若要高穩定性，通常建議自己架設 Overpass API 伺服器。

這個範例 `public/L090-search-cafe.html` 展示了如何使用 Overpass API 在特定區域內搜尋符合特定條件的地理圖資，例如「台北市所有的咖啡店」。

1. Overpass API 簡介

Overpass API 是一個功能強大的唯讀 API，它直接查詢 OpenStreetMap (OSM) 的主要資料庫。相較於 Nominatim 或 OSRM 這種針對特定功能（如搜尋、路線規劃）的服務，Overpass 提供了更靈活、更底層的資料存取能力。

2. Overpass 查詢語言 (QL)

Overpass 使用它自己的查詢語言。範例中的查詢如下：

```
[out:json][timeout:25];
(
  node["amenity"="cafe"](25.0,121.5,25.1,121.6);
  way["amenity"="cafe"](25.0,121.5,25.1,121.6);
  relation["amenity"="cafe"](25.0,121.5,25.1,121.6);
);
out center;
```

- `[out:json][timeout:25];`: 設定輸出格式為 JSON，以及 25 秒的超時限制。
- `node["amenity"="cafe"](bbox);`: 在指定的邊界框 (Bounding Box) 內，尋找所有 OSM 標籤為 `amenity=cafe` 的「節點 (node)」。
- `way` 和 `relation`: 同樣尋找符合條件的「路徑 (way)」和「關聯 (relation)」。OSM 的圖資是由這三種基本元素組成的。
- `(25.0,121.5,25.1,121.6)`: 這就是 Bounding Box，格式為 `(南,西,北,東)`。
- `out center;`: 這是一個重要的指令。對於 `way` 和 `relation` 這種非點狀的圖資，它會計算一個中心點，並在回傳的資料中附上 `center` 屬性，方便我們在地圖上標示。

參考資料：https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL

3. 發送 POST 請求

Overpass API 通常使用 `POST` 方法來接收查詢，查詢語法本身則放在請求的 body 中。

```
const response = await fetch(url, {
  method: "POST",
  body: "data=" + encodeURIComponent(query),
});
```

4. 處理並顯示結果

API 回傳的資料中，`data.elements` 是一個包含了所有符合條件圖資的陣列。

- 我們遍歷這個陣列，並根據圖資的 `type` (`node` 或其他) 來決定如何取得座標。
    - 如果是 `node`，直接使用 `element.lat` 和 `element.lon`。
    - 如果是 `way` 或 `relation`，則使用 `out center;` 指令提供的 `element.center.lat` 和 `element.center.lon`。
- 圖資的名稱通常存放在 `element.tags.name` 中。如果沒有名稱，我們可以給一個預設值。
- 最後，為每個找到的地點建立一個 `L.marker` 和對應的 Popup。

```
cafes.forEach((cafe) => {
  let lat, lon;

  if (cafe.type === "node") {
    lat = cafe.lat;
    lon = cafe.lon;
  } else if (cafe.center) { // 來自 out center;
    lat = cafe.center.lat;
    lon = cafe.center.lon;
  }

  if (lat && lon) {
    const name = cafe.tags?.name || "咖啡店";
    L.marker([lat, lon]).addTo(map).bindPopup(`☕ ${name}`);
  }
});
```

L100: 從 JSON 檔案載入多個標記

這個範例 `public/L100-markers.html` 延續了前一節的概念，但這次我們不是從即時的 API 獲取資料，而是從一個預先準備好的本地 JSON 檔案 (`data/cafes.json`) 載入資料。

1. 為何要使用本地檔案？

- **效能**: 當地點資料是固定的，從本地檔案載入會比每次都向遠端 API 請求快得多。
- **穩定性**: 不用擔心遠端 API 是否可用或有效能問題。
- **節省資源**: 避免對免費的 API 服務造成過多不必要的請求。

一個常見的工作流程是：先透過 Overpass API 將需要的資料查詢出來，儲存成一個 JSON 檔案，然後在網頁中直接讀取這個檔案。

2. 使用 `fetch` 載入本地檔案

`fetch` 不僅能讀取遠端 URL，也可以讀取同網域下的本地檔案。用法完全相同。

```
fetch(`data/cafes.json`)
  .then((r) => r.json())
  .then((result) => {
    // ... 處理資料
  });
```

3. 處理與顯示

由於 `data/cafes.json` 的格式與 Overpass API 回傳的格式完全一樣，因此後續處理資料並將其轉換為 `L.marker` 的程式碼，與前一節 `L090` 的範例幾乎完全相同。這展示了將「資料獲取」與「資料呈現」分離的好處。

```
result.elements.forEach((cafe) => {
  if (cafe.type === "node") {
    const name = cafe.tags?.name || "咖啡店";
    const marker = L.marker([cafe.lat, cafe.lon])
      .addTo(map)
      .bindPopup(`☕ ${name}`);
  }
});
```

L110: 連動選單與地圖

這個範例 `public/L110-menu-to-loc.html` 建立了一個常見的互動模式：點擊側邊選單中的項目，地圖會平移並縮放至對應的標記，同時開啟它的 Popup。

1. 建立 UI 與資料的連結

這個功能的關鍵，在於建立「選單 DOM 元素」與「地圖 Marker 物件」之間的關聯。

1. **動態生成**: 在讀取 `cafes.json` 後，`forEach` 迴圈中不只建立 `L.marker`，同時也建立一個 `<div>` 作為選單項目，並將它加到畫面上。
2. **使用 `Map` 儲存關聯**: 範例中使用了一個 JavaScript 的 `Map` 物件 (`menuMap`) 來儲存這個關聯。`Map` 的好處是它的鍵 (key) 可以是任何型別，包括 DOM 元素物件本身。

```
// 每次迴圈
// ...
const marker = L.marker(...).addTo(map).bindPopup(name);
const item = document.createElement("div");
list.append(item);

// 將 item 元素作為 key，相關資訊 (包含 marker) 存為 value
menuMap.set(item, {
  lat: cafe.lat,
  lon: cafe.lon,
  item, // DOM 元素本身
  marker, // Leaflet 的 Marker 物件
});
```

2. 使用事件委派 (Event Delegation)

與其為每個選單項目都加上 `click` 事件監聽器，更有效率的做法是只在它們的父容器 (`.list`) 上設定一個監聽器。

當點擊事件發生時，我們可以透過 `event.target` 來得知實際被點擊的是哪一個 `item`。

```
list.addEventListener("click", (e) => {
  const t = e.target; // t 就是被點擊的那個 <div>
  // ...
});
```

3. 觸發地圖互動

在點擊事件的處理函式中：

1. 使用 `menuMap.get(t)` 從我們建立的 `Map` 中，透過被點擊的 `div` 元素 (key) 找出對應的資料物件 (value)。

2. 從資料物件中取出 `marker` 和座標 `lat`, `lon`。
3. 呼叫 `marker.openPopup()` 來開啟對應標記的彈出視窗。
4. 呼叫 `map.setView([lat, lon], 16)` 將地圖中心移動到標記的位置，並設定一個較近的縮放層級。

```
list.addEventListener("click", (e) => {
  const t = e.target;
  const data = menuMap.get(t);
  if(!data) return; // 如果點到的是 list 本身而不是 item，就忽略

  const {lat, lon, item, marker} = data;
  marker.openPopup();
  map.setView([lat, lon], 16);
});
```

經緯度之間求距離

經緯度之間求距離時，因為地球是球體（近似橢球），不能直接用平面幾何公式。最常見的是 Haversine formula（哈弗辛公式），它能計算兩點之間的大圓距離（great-circle distance）。

```
function haversine(lat1, lon1, lat2, lon2) {
  const R = 6371; // 地球半徑 (km)
  const toRad = deg => deg * Math.PI / 180;

  const dLat = toRad(lat2 - lat1);
  const dLon = toRad(lon2 - lon1);

  const a = Math.sin(dLat / 2) ** 2 +
            Math.cos(toRad(lat1)) * Math.cos(toRad(lat2)) *
            Math.sin(dLon / 2) ** 2;

  const c = 2 * Math.asin(Math.sqrt(a));

  return R * c; // 單位：公里
}

// 範例：台北 25.033°N, 121.565°E 到 高雄 22.627°N, 120.301°E
console.log(haversine(25.033, 121.565, 22.627, 120.301).toFixed(2) + " km");
```

注意： Haversine 假設地球是球體，誤差 <1%（幾公里等級）。如果要高精度（例如測量/導航），會用 Vincenty formula 或 geodesic（橢球模型 WGS84）。