Angular 2 新手急救站

你所必須知道的 Angular 2 相關知識

多奇數位創意有限公司

技術總監 黃保翕 (Will 保哥)

部落格：http://blog.miniasp.com/

課程大綱

- 認識 Angular 2 相關工具
  - Visual Studio Code
  - Node.js 與 npm
  - gulp 與 webpack
- 認識 TypeScript 與相關工具
  - tsc & typings
  - Declaration File (*.d.ts)
- 了解 TypeScript 與 ES6 語言特性
  - Type Annotation / Casting
  - Module / import / export
  - Class / Interface
  - Arrow function
  - Generics (泛型)
  - Decorator
  - Spread operator (...)

Understanding Angular 2 development tools

認識 ANGULAR 2 相關工具

為什麼要用 Visual Studio Code？

- 免費的程式碼編輯器 (Free)
- 開放原始碼 (Microsoft/vscode)
- 真正跨平台：Linux (.deb) (.rpm), OSX, Windows
- 擁有豐富的擴充外掛支援 (Extensions)
- 非常適合用來建置與偵錯 Web 與 Cloud 應用程式
- 完全以 TypeScript 與 Electron 打造而成

使用 VSCode 的重要觀念

- **編輯器** (Editor)
  - 同時最多開啟 3 個編輯器，新版已支援**頁籤**顯示
  - 支援程式碼片段 (Code Snippets)
- **控制命令** (Commands)
  - 所有主選單與操作介面的背後都透過**控制命令**執行
  - 許多**控制命令**都包含相對應的**鍵盤快速鍵**
- **工作區 / 專案資料夾** (Workspace)
  - 工作區即目前由 VSCode 管理的資料夾
  - 可針對目前工作區進行 VSCode 偏好設定（`.vscode`）

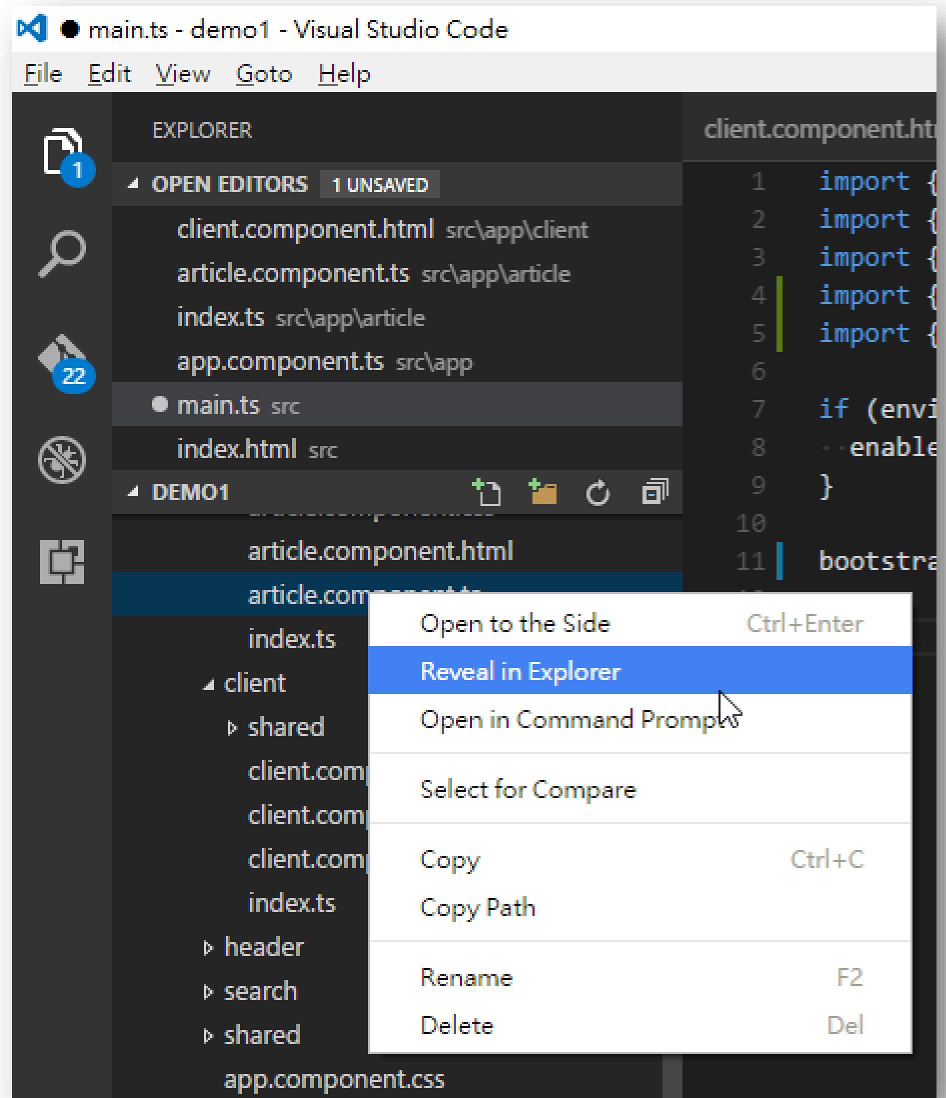
Visual Studio Code 使用者介面

側邊欄 (Sidebar) 的五種檢視

- 支援 5 種不同檢視窗格(Views)
  - Explorer 檔案總管 (Ctrl+Shift+E)
  - Search 搜尋與取代 (Ctrl+Shift+F)
  - Git Git 版本控管 (Ctrl+Shift+G)
  - Debug 應用程式偵錯 (Ctrl+Shift+D)
  - Extensions 擴充套件管理 (Ctrl+Shift+X)
- 可用 Ctrl+B 切換側邊欄檢視窗格

檔案總管 (Explorer)

- 編輯中檔案
  - WORKING FILES
  - 可啟用自動儲存
- 工作區檔案
  - 開啟到分割視窗
  - 開啟檔案總管
  - 開啟命令提示字元
  - 選取檔案比對
  - 複製/貼上檔案
  - 複製檔案路徑
  - 更名/刪除檔案



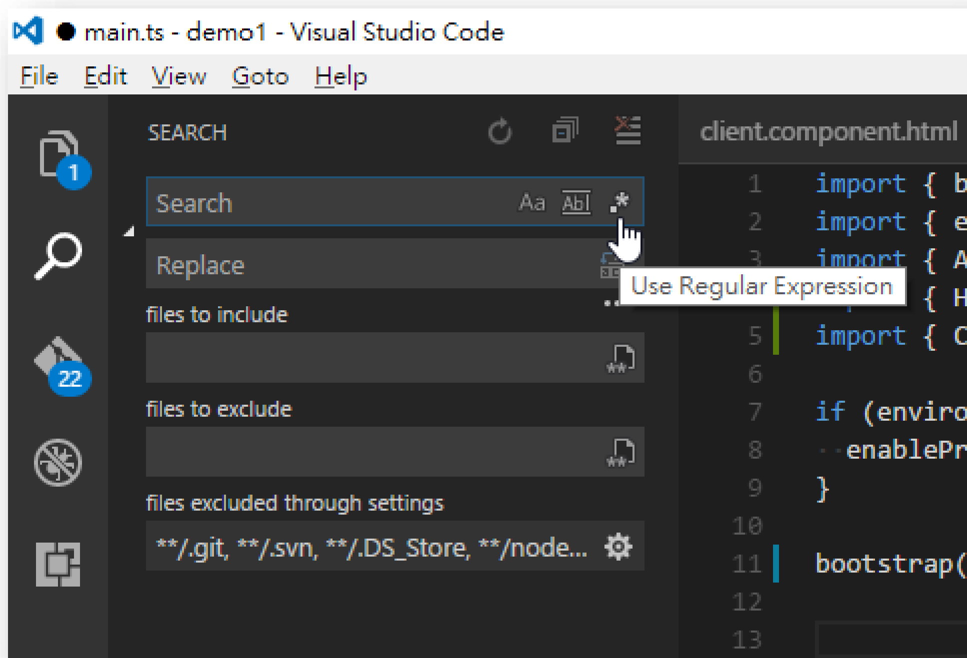
檔案總管 (Explorer)

- 開啟檔案
  - `Click`
    - 開啟檔案在 "左編輯器" (Editor 1)
  - `Ctrl+Click`
    - 開啟檔案在 "右編輯器" (Editor 2)

- 切換編輯器視窗的快速鍵
  - `Ctrl + 0` 切換到檢視窗格
  - `Ctrl + 1` , `Ctrl + 2` 指定切換到編輯器視窗
  - ``Ctrl + ` `` 輪流切換編輯器視窗
  - `Ctrl + W` 關閉編輯器視窗

搜尋與取代 (Search)

- 搜尋工作區內所有檔案內容
  - 支援 Regular Expression
  - 支援 Node Glob Pattern (e.g. `**/*.js`)



Git 版本控管 (Git)

- 支援大部分 Git 常見操作

Command	Action
– git init	Initialize git repository
– git pull & git push	Sync
– git pull	Pull
– git pull --rebase	Pull (Rebase)
– git push	Push
– git add	Stage
– git rm --cached	Unstage
– git clean	Clean
– git commit	Commit Staged
– git commit -a	Commit All
– git reset --soft HEAD^	Undo Last Commit

應用程式偵錯 (Debug)

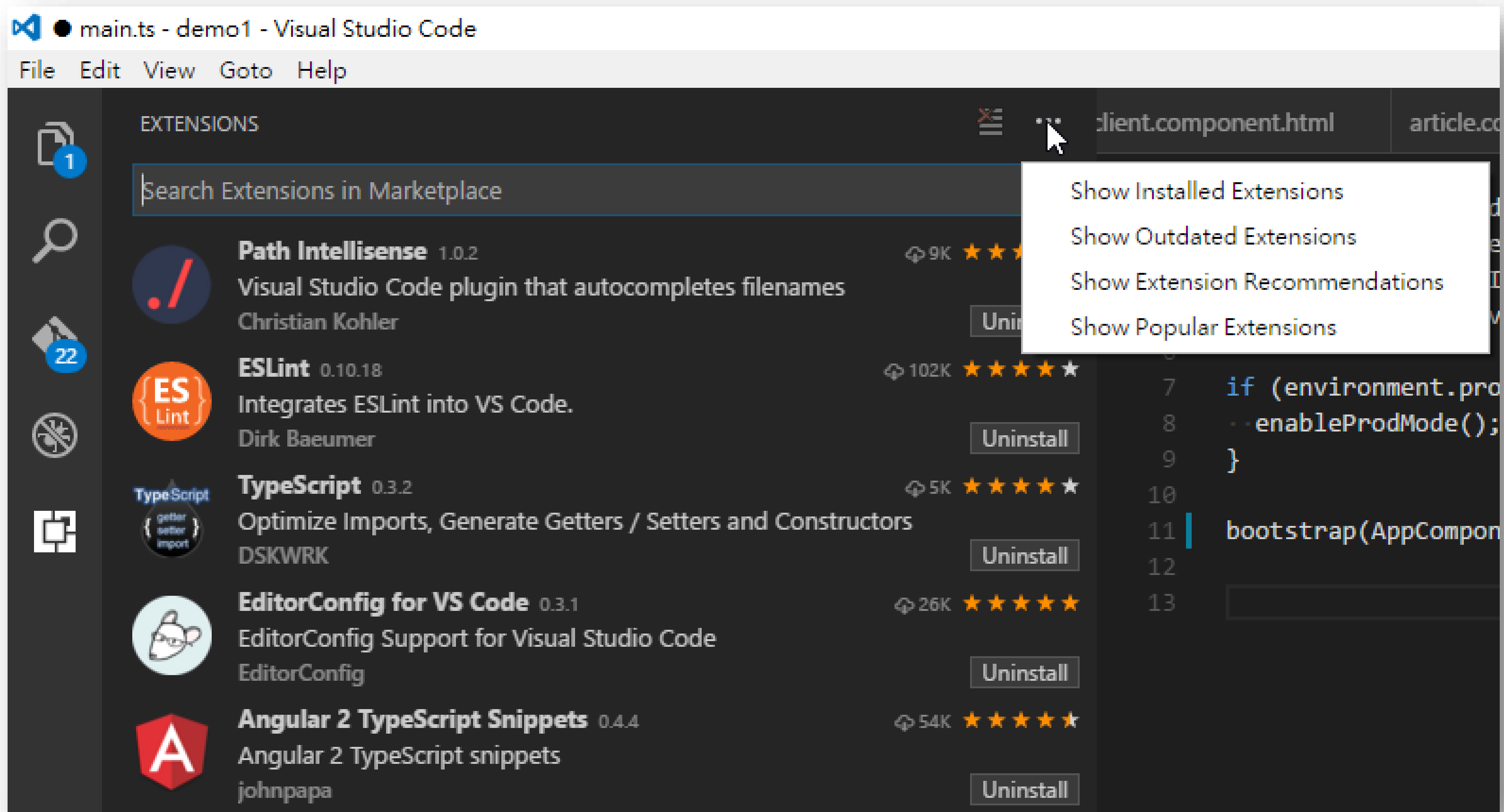
- 支援多種開發框架的偵錯能力
  - Node.js, VSCode Extension Development, Go, .NET Core, PowerShell, PowerShell x86, Chrome

- 設定 (Configure)
- 主控台 (Debug Console)

- 即時變數 (VARIABLES)
- 監看式 (WATCH)
- 呼叫堆疊 (CALL STACK)
- 中斷點 (BREAKPOINTS)

擴充套件管理 (Extensions)

- 預設會列出所有已安裝擴充套件，也可針對套件進行篩選
- 輸入關鍵字即可搜尋 Marketplace 上的所有擴充套件

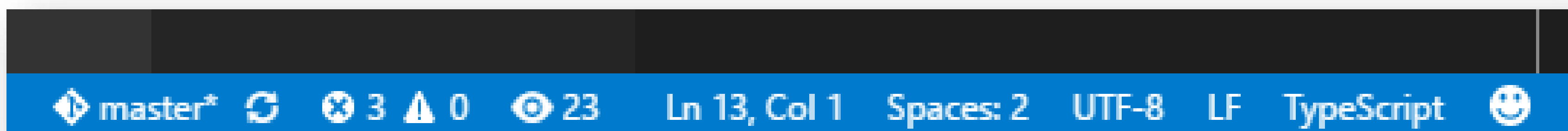


編輯器視窗 (Editors)

- 智慧標籤整合常見動作
  - **`Ctrl+.`**
- 更智慧的 Intellisense
  - 背景執行 TypeScript 型別檢查 (看懂你的Code)
- 內建 JavaScript 語法與語意檢查
  - 可切換 jshint 或 eslint
  - 會自動讀取 .jshintrc 設定檔

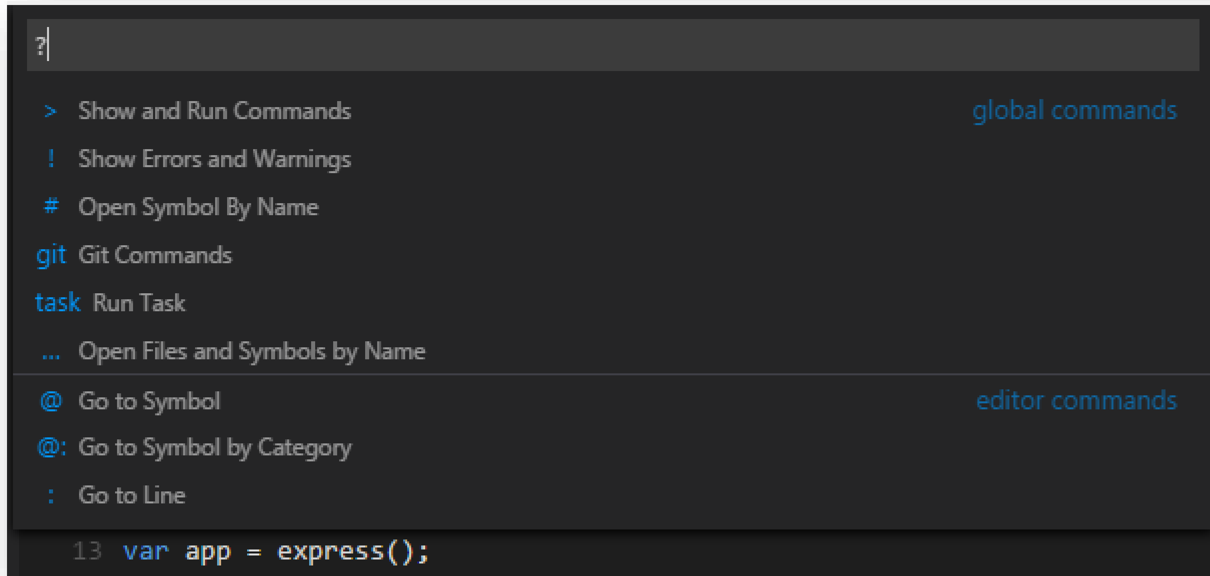
狀態列 (Status Bar)

- 切換 Git 分支 (Git Checkout)
- 同步 Git 遠端儲存庫 (Git Sync)
- 檢查錯誤與警示 (Errors and Warnings)
- 游標位置資訊 (Go to Line)
- 設定檔案編碼 (Select Encoding)
  - Reopen with Encoding
  - Save with Encoding
- 選擇換行符號 (Select End of Line Sequence)
- 選擇語言模式 (Select Language Mode)
- 意見回饋 (Feedbacks)



VSCode 命令面板

- 快速開啟 Ctrl+P 或 Ctrl+E
  - 輸入 ? 查詢各種用法



  - 查詢與執行命令 Ctrl+Shift+P
  - 查詢錯誤與警示 Ctrl+Shift+M

簡介 Node.js

- Node.js® 就是**伺服器端**的 JavaScript **引擎**！
- 主要特性
  - 基於 Chrome's JavaScript runtime (V8)
  - 用來建立高速且可延展的網路應用程式。
  - 使用事件驅動的 non-blocking I/O 模型
  - 強大的社群支援，至今超過 163,793 個模組可供下載
  - 跨平台 (Mac, Linux, Windows)
  - 可讓**相同 JS 程式碼**共享於**前端**與**後端**開發
- 學習資源
  - Node入門 » 一本全面的Node.js教學課程

簡介 NPM (Node Package Manager)

- npm 是一個網站
  - https://www.npmjs.com/

- npm 是一個公開的線上資料庫 (npm public registry)
  - 主要用來共享 node.js 程式碼
  - 主要以 套件 (package) 或 模組 (module) 做封裝
    - 這裡的**套件**與**模組**其實是同一件事，都代表 npm package！

- npm 是一套 Node.js 的套件管理工具
  - 用來搜尋、安裝、更新、移除、發行 npm 套件的工具
  - 管理套件與套件之間的相依性

NPM 的重要概念

- 套件的組成
  - 一個資料夾
  - 資料夾中有個 package.json 檔案
    - 用來定義套件的名稱、版本與相關資訊
    - 用來定義套件中還用到哪些其他套件 (套件相依性)
  - 資料夾中有一堆自訂的檔案與其他資料夾
- 專案的組成 (也是個資料夾)
  - 透過 npm 安裝的套件
  - 你自己寫好的程式碼與其他檔案
- 專案本身也可以變成套件
  - 可發佈到 npm public registry 變成公開的套件

常見問題解決：NPM

- **完整移除 Node.js 與 npm 套件的方法**
  - 先移除 `Node.js` 應用程式
  - `del "%USERPROFILE%\.npmrc"`
  - `rmdir /s/q "C:\Program Files\nodejs"`
  - `rmdir /s/q "%APPDATA%\npm"`
  - `rmdir /s/q "%APPDATA%\npm-cache"`
- **無法安裝套件：清除所有快取套件**
  - `npm cache clean`
- **無法編譯套件：變更執行環境設定的 Visual Studio 版本**
  - 安裝 Visual Studio Community 2013
  - `npm config set msvs_version 2013`
  - `setx GYP_MSVS_VERSION 2013`
- **版本控管的注意事項**
  - 排除 `node_modules` 資料夾

搜尋與安裝 npm 套件

- 搜尋套件
  1. npm search *package_name* (建議不要用指令搜尋)
  2. https://www.npmjs.com/ (透過網站搜尋快多了！)
- 本地安裝 (安裝 node.js 模組)
  – **npm install <package name>**
  – 安裝過程預設會將套件安裝在當前專案的 **node_modules** 目錄下
- 全域安裝 (安裝 node.js 工具)
  – **npm install -g <package name>**
  – 安裝過程預設會將套件安裝在統一的 npm 目錄下
    - **WIN**: %APPDATA%\npm\node_modules
    - **OSX**: /usr/local/lib/node_modules
    - 查詢 npm 儲存路徑: **npm config get prefix**
- 自動安裝 package.json 套件定義檔中定義的所有套件
  – git clone https://github.com/jquery/jquery.git
  – cd jquery
  – **npm install**

安裝 npm 套件 - 本地安裝/儲存設定

- 安裝套件並儲存在 package.json 套件定義檔中
  - npm install <package name> --save
  - npm install <package name> --save-dev
- 執行範例
  - npm install lodash --save
  - npm install concat --save-dev

```
1 {
2   "name": "f2e-workflow-labs",
3   "version": "1.0.0",
4
5   "dependencies": {
6     "lodash": "^3.10.1"
7   },
8   "devDependencies": {
9     "concat": "^1.0.0"
10  }
11 }
```

- 關於 **--save**
  - 套件發行時宣告的相依套件
- 關於 **--save-dev**
  - 開發環境開發時需要的套件
  - **前端開發**的大多數情況，你只需要 **--save-dev**

簡介 Gulp

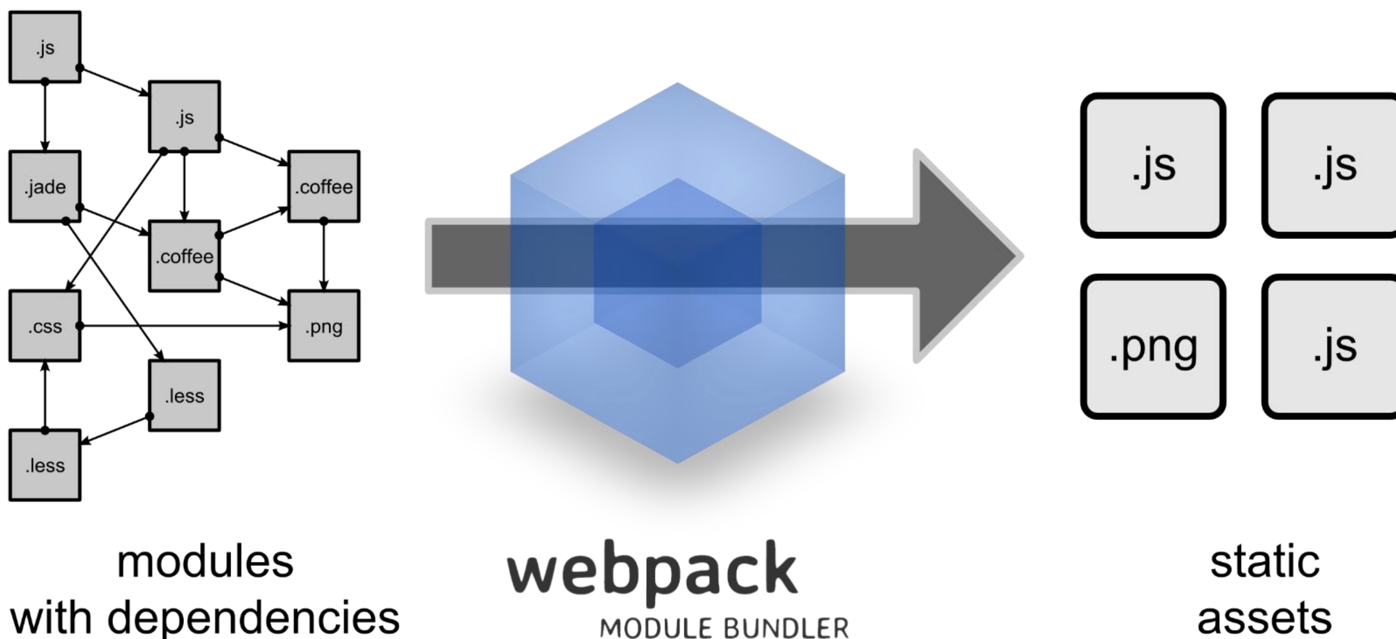
- 改善前端工作流程，將工作自動化的工具
  - 聰明工作，不要蠻幹 (Work Smarter, Not Harder)
- 主要特性
  - 容易使用
    - 強調 Code over configuration
  - 效能極佳
    - 採用 node streams 機制提升執行速度
  - 超高品質
    - 實施嚴格的外掛規範，確保外掛可以照預期的方式運作
  - 容易學習
    - 只要會寫一點點 node.js 就可以開始撰寫 gulp 的工作定義

Gulp 能做甚麼

- 程式碼撰寫風格檢查、程式碼品質分析
- 最小化 (Minification)、醜化 (Uglify)
- 合併檔案 (Concatenation)
- 套用格式轉換 (Less, Sass, TypeScript, Babel, ...)
- 套用 Vendor prefixes
- 自動注入 JS/CSS 引用到 HTML 之中
- 更新套件版本
- 快取 HTML 範本
- 單元測試、整合測試、連續性整合

簡介 webpack

- 管理**模組相依性**的工具
  - 其**模組**包括 html, js, coffee, css, less, png, webfonts, ...
  - 透過產生**靜態的檔案**來**代表**這些**相依的模組**



Understanding TypeScript and it's tools

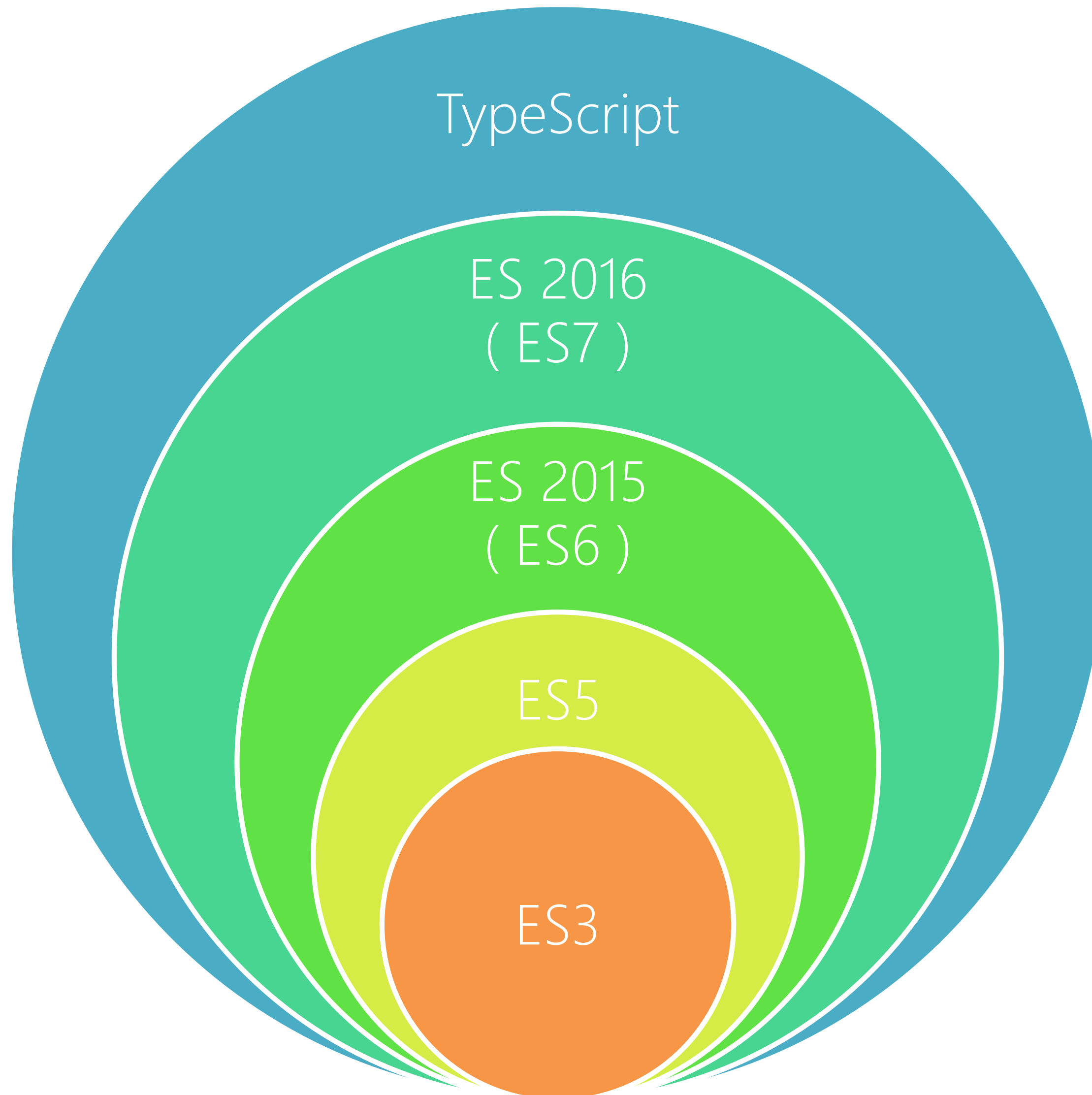
認識 TYPESCRIPT 與相關工具

什麼是 TypeScript？

一個能在**開發時期**宣告**型別**的 JavaScript **超集合**
可以被**編譯**成 JavaScript 程式碼

可執行在
任意瀏覽器、任意主機、任意作業系統
並且
開放原始碼

TypeScript 與 JavaScript 的關係



何時才能開始用？

- 可以執行 JavaScript 的地方？
  - 各家瀏覽器、Node.js、NativeScript、...etc.
  - 跨平台、跨瀏覽器、跨作業系統，連 IoT 裝置都能跑
- TypeScript 最終將編譯成傳統的 JavaScript
- 編譯後的 JavaScript 可選 ES3, ES5, ES6 版本

「**現在**」就可以開始使用！

JavaScript 程式的可維護性

- JavaScript 無法適用大型前端應用程式開發？
  - 大家都知道
    - JavaScript 寫出來的程式五花八門
    - JavaScript 寫出來的程式不好維護
    - JavaScript 寫出來的程式只能在執行時期偵錯
    - 不同人寫出來的 JavaScript 程式很難理解
    - 規模越大的 JavaScript 應用程式，偵錯的難度越高
    - 寫好 JavaScript 的難度有點高，所以乾脆寫少一點！
  - 你可能不知道
    - TypeScript 可以解決這些難題！

JavaScript 動態型別的優缺點

JS 語言特性	優點	缺點
使用 var 宣告變數	變數可以容納任意物件	無法在**開發時期**宣告型別
執行時期決定物件型別	大部分時候無須檢查型別	只能在**執行時期**檢查型別
自動轉換型別	簡化頻繁的型別檢查	因為**不同型別的物件**會有不同的預設**屬性**與**方法**，因此經常會遇到程式執行錯誤。 大部分 JS 開發人員不使用 === 比對資料。

為什麼要使用 TypeScript 呢？

TS 語言特性	說明	語法示意
支援靜態型別	透過擴充的 var 宣告變數語法 搭配 TypeScript 編譯器檢查	`var data: number = 1;`

支援介面型別	擴充 JS 缺少的語言特性	`interface MyObject {`
		`  data: number;`
		`}`

型別自動推導	自動判定變數的物件型別	`var a = {name: 'Will'};` `a.test = 1; // Error!`

先進語言特性	支援目前最新 JS 規格 (ES6)
	可選擇轉譯為 ES3 以上版本
		`for(let i=0; i < 9; i++) {`
		`  console.log(i);`
		`}`

使用 TypeScript 的亮點

- **從 JavaScript 開始，也從 JavaScript 結束**
  - 原本 JavaScript 語法，在什麼都不改的情況下，就是完整且有效的 TypeScript 語法，100% 相容。
  - 可呼叫現有的 JavaScript 函式庫，因為最終可編譯出 ES3 以上版本的 JavaScript 語法。

- **強大工具支援，適合建構大型 JavaScript 應用程式**
  - 許多開發工具/編輯器都已經支援 TypeScript 整合開發能力，可以透過即時程式碼檢查與各種**程式碼重構**能力大幅**提高開發生產力**。
  - 撰寫 TypeScript 時不一定要宣告型別，因為 TypeScript 支援型別自動推導能力，在開發環境中不需要每個變數都宣告型別。

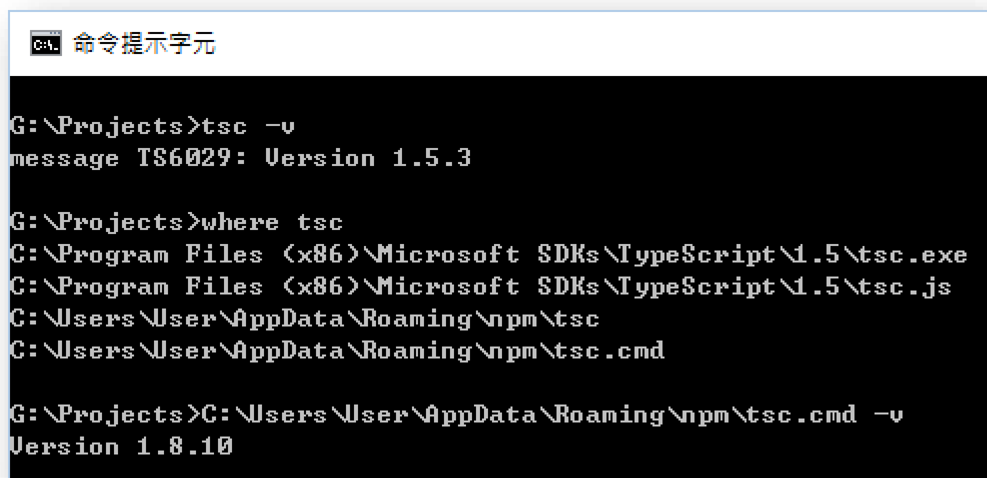
- **使用最先進的 JavaScript 語法 (ES5, ES2015, ...)**
  - 相容所有正式版的 ECMAScript 規格，還包含部分未來規格。

安裝 TypeScript 相關工具

- 教學影片
  - 前端工程師如何在 Windows 安裝自動化流程工具
- 安裝 Node.js
  - 建議下載 v4.x 版本 (不要安裝 v6.x 以上版本)
- 使用 npm 安裝工具
  - `npm install -g typescript typings`
  - `npm install -g yo gulp webpack rimraf eslint`
- 檢查安裝狀況
  - `node -v` (建議有 6.2.2 以上版本)
  - `npm ls -g --depth=0` (查詢目前已安裝的npm模組)
  - `tsc -v` (必須為 1.8.10 以上版本)
  - `typings -v` (必須為 1.3.2 以上版本)
  - `where tsc` (查詢執行檔位於哪個目錄)

常見問題解決：TSC

- 無論怎麼安裝 Typescript 執行 tsc 永遠是舊版
  - 先透過 `where tsc` 找出 `tsc` 指令檔路徑 (如下圖)



  - 只要修復 PATH 環境變數的路徑即可解決
    - 建議可安裝 Rapid Environment Editor

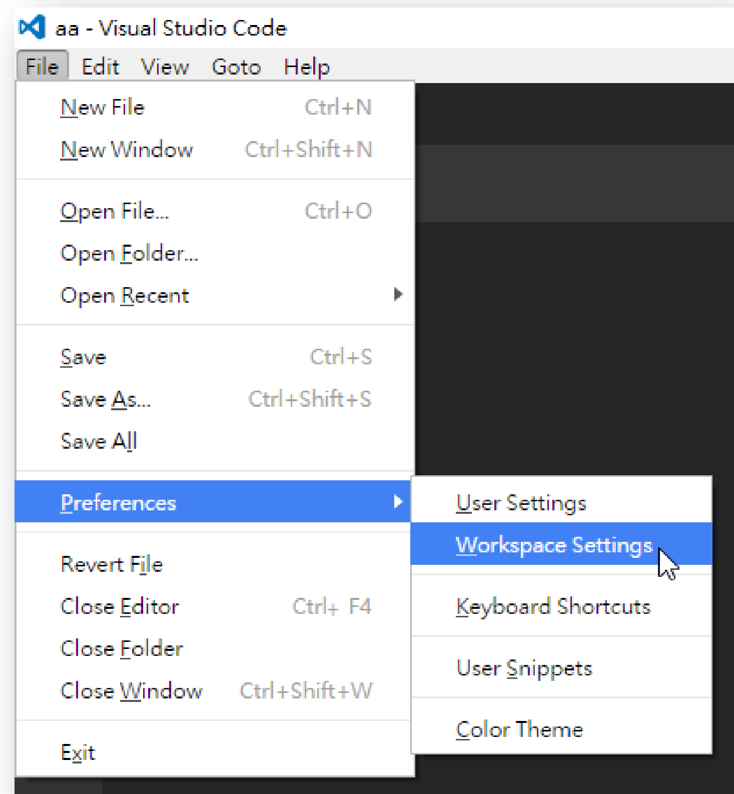
使用 tsc 命令列工具

- 初始化 tsconfig.json 設定檔
  - `tsc --init`
- 啟動 Visual Studio Code 建立 **`index.ts`** 程式
  - `code .`
- 編譯 TypeScript 程式
  - `tsc` (單次編譯)
  - `tsc -w` (持續編譯)(自動監視檔案變更)
  - `tsc -t ES5` (使用ES5語法輸出;預設為ES3)
  - `tsc --pretty` (編譯的時候使用終端機色彩輸出)
  - `tsc -p DIRECTORY` (指定特定專案目錄進行編譯)
  - `tsc --sourceMap` (產生 SourceMap 檔案)

在 VSCode 隱藏 *.js 與 *.js.map 檔案

- 建立 .vscode/settings.json 設定檔

```
{
  "files.exclude": {
    "**/.git": true,
    "**/.svn": true,
    "**/.DS_Store": true,
    "**/*.js": true,
    "**/*.js.map": true
  }
}
```



※ 也可以設定複雜的顯示條件！

※ 參考文章：

[VS Code 檔案顯示設定 | Jeff's WebTech Note]

認識 Typings 工具

- 簡介
  - Typings 將常見的 **JS 框架/函式庫** (外部模組) 的 **模組定義檔** (module definition file) (`*.d.ts`) 封裝進一個被完善定義的**命名空間**或**全域宣告**裡。
- 主要用途
  - 管理 TypeScript 模組定義檔 (取代先前的 tsd 工具)
  - 幫助 IDE/Editor 提供 TypeScript 語言服務 (IntelliSense)
- 公開資料庫 (public registry)
  - 一套由社群維護的公開資料庫，現有知名 JS 框架/函式庫的 TypeScript 模組定義檔 (`*.d.ts`) 都會註冊在此。

使用 typings 命令列工具

- 初始化 typings.json 設定檔
(可用來定義與目前專案的相依 typings 套件)
  - **typings init**
- 搜尋 typings 定義檔
  - **typings search [keyword]**
  - **typings search --name [pkgname]**
- 安裝 typings 定義檔
  - **typings install [pkgsource]~[pkgname]@[ver]**
    - pkgsource 預設為 npm 來源，其他來源都要加上 **[pkgsrc]~**
    - 範例：**typings install dt~jquery --global --save**

      --global 將安裝的 typings 註冊為 global 全域變數

      --save 將安裝的 typings 套件註冊到 typings.json 設定檔

      註冊到 typings.json 的套件未來可直接用 **typings install** 自動安裝

Understanding TypeScript and ES6 Features

了解 TYPESCRIPT 與 ES6 語言特性

ES6 變數與常數

- 使用 `var` 區域變數
  - 屬於 function scope
  - 容易遭遇 Hoisting 的問題

- 使用 `let` 區域變數
  - 屬於 block scope 且**同一範圍**不允許重複宣告變數
  - 用 `var` 宣告過的變數，不能再使用 `let` 宣告一次
  - 用 `let` 宣告變數之後才能開始使用 (變數特性與 C# 非常類似)

- 使用 `const` 區域變數
  - 宣告一個**唯讀的變數** (變數無法再指向其他物件)
  - 宣告完變數後必須立刻初始化變數 (給予變數預設值)
  - **變數特性與 `let` 特性完全相同** (block scope)

觀念驗證 1

- 請問執行以下程式的回傳值為何？

```
(function () {
    function test() { return 2; }
    var test = function() {
        return 1;
    }
    return test();
})();
```

觀念驗證 2

- 請問執行以下程式的回傳值為何？

```
(function () {
    return test();
    var test = function() {
        return 1;
    }
    function test() { return 2; }
})();
```

觀念驗證 3

- 請問執行以下程式的回傳值為何？

```
var a = 1;
(function () {
    let a = 2;
    var test = function() {
        return ++a;
    }
    return test();
})();
```

觀念驗證 4

- 請問執行以下程式的回傳值為何？

```
(function () {
    let a = 1;
    var test = function() {
        console.log(a);
        let a = 2;
        console.log(a);
    }
    return test();
})();
```

觀念驗證 5

- 請問執行以下程式的回傳值為何？ (Shadowing)

```
(function () {
    let matrix = [[1,2,3], [1,2,3], [1,2,3]];
    let sum = 0;
    for (let i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (let i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }
    console.log(sum);
})();
```

觀念驗證 6

- 請問執行以下程式的回傳值為何？

```
(function () {
    let getCity;
    let city = "Taiwan";
    if (true) {
        let city = "Seattle";
        getCity = function() {
            return city;
        }
    }
    return getCity();
})();
```

觀念驗證 7

- 請問以下兩段程式會分別如何執行？

```
(function () {
  for (let i = 0; i < 10 ; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
  }
})();

(function () {
  for (var i = 0; i < 10 ; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
  }
})();
```

JavaScript 與 TypeScript 型別分類

型別種類	JavaScript	TypeScript
動態型別	使用 var 宣告的變數可為任意型別	`var a: any;`
數值型別	Number	`var a: number`
字串型別	String	`var a: string`
布林型別	Boolean	`var a: boolean;`
未定義型別	undefined	`var a: void;`
空值型別	null	`var a: void;`
陣列型別	Array	`var a: number[];` `let x: [string, number];`
列舉型別	無	`enum Color {Red, Green, Blue};` `let c: Color = Color.Green;`

認識 TypeScript 型別系統

- 布林型別 (boolean)

```
let isDone: boolean = false;
let isDone = false;        // 自動型別推導 (Type Inference)
```

- 數值型別 (number)

```
let decimal: number = 6;          // 10 進制
let hex: number = 0xf00d;         // 16 進制
let binary: number = 0b1010;      // 2 進制
let octal: number = 0o744;        // 8 進制
let decimal = 100;    // 自動型別推導 (Type Inference)
```

- 自動型別推導 (Type Inference)
  - 很多時候你並不需要宣告型別 (但宣告型別可增加可讀性)

認識 TypeScript 型別系統

- 字串型別 (string)

```
let color: string = "blue";
color = 'red';

let fullName: string = `Will`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`
```

認識 TypeScript 型別系統

- 簡單陣列型別 (Array)

```
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
```

- 複雜陣列型別 (Tuple)

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10];         // OK
x = ["hello", 10, true];   // OK
// Initialize it incorrectly
x = [10, "hello"];         // Error
x = [10];                  // Error
```

認識 TypeScript 型別系統

- 列舉型別 (enum)

```
enum Color {Red, Green, Blue};                    // 0, 1, 2
let c: Color = Color.Green;                       // 1

enum Color {Red = 1, Green, Blue};                // 1, 2, 3
let c: Color = Color.Green;                       // 2

enum Color {Red = 1, Green = 2, Blue = 4};        // 1, 2, 4
let c: Color = Color.Green;                       // 2

enum Color {Red = 1, Green, Blue};                // 1, 2, 3
let colorName: string = Color[2];                 // "Green"
```

認識 TypeScript 型別系統

- 任意型別 (any)
  - `let notSure: any = 4;`
  - `notSure = "maybe a string instead";`
  - `notSure = false; // okay, definitely a boolean`

  - `let notSure: any = 4;`
  - `notSure.ifItExists(); // okay, the compiler doesn't check`
  - `notSure.toFixed(); // okay, the compiler doesn't check`

  - `let prettySure: Object = 4;`
  - `prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.`

  - `let list: any[] = [1, true, "free"];`
  - `list[1] = 100;`

認識 TypeScript 型別系統

- 物件型別 (Object)

```
let prettySure: {} = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.

let obj: {name: string};
obj.name = "Will";                          // OK

let obj: {name: string} = {name: 'Will'};   // OK

let obj: { print: () => number } =
         { print: function() : number { return 1; } };
obj.print();                                // OK
```

認識 TypeScript 型別系統

- void 型別

通常只會用在 function 的回傳值：

```
function warnUser(): void {
    alert("This is my warning message");
}
```

如果將變數設定為 void 型別，該變數就只能儲存 null 或 undefined

```
let unusable: void = undefined;
```

認識 TypeScript 型別系統

- 型別轉換 (Type assertions) (轉型)

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;

let a = document.getElementById('myLink');  // HTMLElement 型別
a.href = "http://blog.miniasp.com/";        // 找不到 href 屬性

let a = <HTMLAnchorElement>document.getElementById('myLink');
a.href = "http://blog.miniasp.com/";        // OK
```

TypeScript 介面 (Interface)

- 宣告物件型別

```
function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}

// 透過變數額外傳入屬性十，多出的 size 屬性並不會報錯
let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);

// 若是直接傳入自定義物件時，多出的 size 屬性會報錯
printLabel({size: 10, label: "Size 10 Object"});
```

TypeScript 介面 (Interface)

- 宣告介面型別

```
interface ILabel {
    label: string;
    size: number;
}

function printLabel(labelledObj: ILabel) {
    console.log(labelledObj.label);
}

let myObj = { label: "Size 10 Object", size: 10 };
printLabel(myObj);
```

TypeScript 介面 (Interface)

- 定義可選屬性 (Optional Properties)

```
interface ILabel {
    label: string;
    size?: number;       // 透過 ? 符號設定此屬性為非必要屬性
}

function printLabel(labelledObj: ILabel) {
    console.log(labelledObj.label);
}

let myObj = { label: "Size 10 Object" };
printLabel(myObj);
```

TypeScript 介面 (Interface)

- 排除過度屬性檢查

```
interface ILabel {
    label: string;                // 必要屬性，必須傳入！
    [propName: string]: any;  // 允許任意屬性傳入
}

function printLabel(labelledObj: ILabel) {
    console.log(labelledObj.label);
}

printLabel({ label: "Size 10 Object", size: 10 });
```

TypeScript 介面 (Interface)

- 使用**物件實字表示法**傳入才會執行**過度屬性檢查**
  - 原因：通常直接傳入物件實字而寫錯屬性通常是 Bugs

```
interface ILabel {
    label: string;
}

function printLabel(labelledObj: ILabel) {
    console.log(labelledObj.label);
}

let myObj = { label: "Size 10 Object", size: 10 };
printLabel(myObj); // 透過變數傳遞時 TS 編譯器不會報錯!
```

TypeScript 類別 (Class)

- 基本語法結構

```
class Greeter {
    greeting: string;           // 預設為公開屬性
    private title: string;    // 宣告為私有屬性
    constructor(message: string, title: string) {
        this.greeting = message;
        this.title = title;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
let greeter = new Greeter("world");
```

TypeScript 類別 (Class)

- 宣告私有屬性的簡易寫法

```
class Animal {

    // 直接在建構式中透過 private 宣告私有屬性 name
    // ( 也可以透過 public 或 protected 宣告 )
    constructor(private name: string) { }

    move(distanceInMeters: number) {
        console.log(
          `${this.name} moved ${distanceInMeters}m.`);
    }
}
```

TypeScript 類別 (Class)

- 宣告靜態屬性

```
class Grid {
    static origin = {x: 0, y: 0};
    calculateDistanceFromOrigin(
      point: {x: number; y: number;}) {
        let xDist = (point.x - Grid.origin.x);
        let yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist + yDist) / this.scale;
    }
    constructor (public scale: number) { }
}
```

TypeScript 類別 (Class)

- 類別屬性的 get 與 set 存取子 (Accessors)

```
let passcode = "secret passcode";
class Employee {
    private _fullName: string;
    get fullName(): string { return this._fullName; }
    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update!");
        }
    }
}
let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {  console.log(employee.fullName);  }
```

TypeScript 類別 (Class)

- 宣告抽象類別

```
abstract class Animal {    // 抽象類別必須被繼承
    abstract makeSound(): void;  // 抽象方法必須被實作
    move(): void {
        console.log("roaming the earth...");
    }
}
class Doggy extends Animal {
    makeSound() {

    }
}
```

泛型 (Generics)

- 單型別函式 (沒有泛型的情況)

```
function identity(arg: number): number {
    return arg;
}
```

- 任意型別函式 (沒有泛型的情況)

```
function identity(arg: any): any {
    return arg;
}
```

泛型 (Generics)

- 泛型函式

```
function identity<T>(arg: T): T {
    return arg;
}
```

- 呼叫泛型函式 (明確指定使用字串型別)

```
// identity 的回傳值將會是 'string' 型別
let output = identity<string>("myString");
```

- 呼叫泛型函式 (交由 TypeScript 自動型別推導)

```
let output = identity("myString");
```

更多泛型語法

- 泛型函式 (傳入陣列)

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // 陣列才有 .length 屬性
    return arg;
}
```

- 泛型介面

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}
```

- 泛型類別

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}
```

認識 ES 2015 / TypeScript 模組化技術

- 每個檔案都是一個「模組」(module)
- 每個模組內都可以「匯出」(export) 公開的物件
- 匯出

```
export class Product {
    pageTitle = "Hello World";
}
```

- 匯入

```
import { Product } from './product';    // target: ES6
import { Product as prod } from './product';
import * as product from './product';
import express = require('express');    // target: ES5
import './product'; // side effects only (ref)
```

裝飾器 (Decorators)

- C# **類別** (Class) 有 **屬性** (Attribute) 夾帶**附加資訊**
- TypeScript **類別** (Class) 也有 **裝飾器** (Decorator)
  - 但目前**裝飾器** (Decorator) 屬於**實驗功能** (ES7 Draft)
- **裝飾器** (Decorator) 可以套用在：
  - 類別 (class declaration)
  - 屬性 (property)
  - 方法 (method)
  - 方法中的參數 (parameter)
  - get 或 set 存取子 (accessor)

裝飾器 (Decorators) 的程式碼外觀

- 永遠以 @ 開頭
  - `@sealed`
- 裝飾器一定是 function 並會在執行時期被叫用
- 不用分號結尾
- 在一個目標上可以套用多個裝飾器

```
@f() @g() x: string

@f()
@g()
x: string
```

啟用 TypeScript **裝飾器**實驗功能

- 命令列啟用方式

```
tsc --target ES5 --experimentalDecorators
```

- tsconfig.json 設定方式

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

符號 (Symbols)

- ECMAScript 2015 新的**原始型別** (Primitive Type)
- 可當成物件的屬性名稱

```
let sym = Symbol();
let obj = { [sym]: "value" };
console.log(obj[sym]); // "value"
```

- 可當成類別的屬性名稱

```
const getClassNameSymbol = Symbol();
class C {
    [getClassNameSymbol](){
        return "C";
    }
}
let c = new C();
let className = c[getClassNameSymbol](); // "C"
```

內建的符號物件 (Well-known Symbols)

- Symbol.hasInstance
  - 方法，會被 instanceof 運算子呼叫。用來識別一個物件是否是其提示。
- Symbol.isConcatSpreadable
  - 布林值，判斷一個物件的陣列元素是否可展開 (可否執行 concat 方法)
- Symbol.iterator
  - 方法，被 `for-of` 語句呼叫。回傳物件的預設迭代器。
- Symbol.match
  - 方法，被 String.prototype.match 呼叫。使用正則表達式用來比對字串。
- Symbol.replace
  - 方法，被 String.prototype.replace 呼叫。使用正則表達式用來取代字串。
- Symbol.search
  - 方法，被 String.prototype.search 呼叫。使用正則表達式用來搜尋字串。
- Symbol.split
  - 方法，被 String.prototype.split 呼叫。使用正則表達式用來分割字串。

內建的符號物件 (Well-known Symbols)

- Symbol.species
  - 函式，為一個建構式函式。用來建立衍生物件。
- Symbol.toPrimitive
  - 方法，當物件透過 ToPrimitive 呼叫時，會把物件轉換為相對應的原始值。
- Symbol.toStringTag
  - 方法，被內建方法 Object.prototype.toString 呼叫時，會取得該物件背後的原始型別值 (字串)。
- Symbol.unscopables
  - 物件，當使用 with 關鍵字時，宣告該物件的特定屬性是否允許使用。

宣告使用者自訂符號的方式

```
let sym1 = Symbol();
let sym2 = Symbol("key"); // optional string key

let sym2 = Symbol("key");
let sym3 = Symbol("key");
sym2 === sym3; // false, symbols are unique

let sym = Symbol();
let obj = {
    [sym]: "value"
};
console.log(obj[sym]); // "value"
```

迭代器 (Iterators)

- 何謂「迭代器」？
  - 負責取出陣列或物件中的所有資料
  - 任意物件只要實作 `Symbol.iterator` 屬性就算**含有迭代器的物件**
  - 物件的 `Symbol.iterator` 屬性必須為一個 Generator 函式
    - `obj[Symbol.iterator] = function* () { yield 1; }`
  - 物件的 `Symbol.iterator` 屬性 (迭代器) 會在迭代時的被執行
  - 內建迭代器的型別
    - `Array`, `Map`, `Set`, `String`, `Int32Array`, `Uint32Array`
- 執行**迭代器**的方法
  - 使用 `for..of` 語法

```
let someArray = [1, "string", false];
for (let entry of someArray) {
    console.log(entry); // 1, "string", false
}
```

比較 for..in 與 for..of 的差異

```
let list = [4, 5, 6];

for (let i in list) {
   console.log(i); // "0", "1", "2"
}

for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```

展開運算子 (Spread operator) (...)

- 常見的 JavaScript 開發情境
  - 宣告一個函式
    - `function myFunction(x, y, z) { }`
  - 宣告一個陣列
    - `var args = [0, 1, 2];`
  - 請將 args 這三個元素傳入 myFunction 的 x, y, z 參數？
    - 解法 1：傳統 JS 解法
      - `myFunction.apply(null, args);`
    - 解法 2：使用展開運算子 (Spread operator)
      - `myFunction(...args);`

展開運算子 (Spread operator) (...)

- 常見的 JavaScript 開發情境
  - 宣告兩個陣列
    - `var arr1 = [0, 1, 2];`
    - `var arr2 = [3, 4, 5];`
  - 請問該如何把這兩個元素合併成一個
    - 解法 1：傳統 JS 解法 (不用迴圈的方法)
      - `Array.prototype.push.apply(arr1, arr2);`
    - 解法 2：使用展開運算子 (Spread operator)
      - `arr1.push(...arr2);`

更多展開運算子範例

- 更彈性的函式參數傳入
  - `function myFunction(v, w, x, y, z) { }`
  - `var args = [0, 1];`
  - `myFunction(-1, ...args, 2, ...[3]); // "展開陣列"`

- 更強大的陣列表示法
  - `var parts = ['shoulders', 'knees'];`
  - `var lyrics = ['head', ...parts, 'and', 'toes'];`
    `// ["head", "shoulders", "knees", "and", "toes"]`

只有陣列能使用展開運算子嗎？

- 基本原則
  - 只要有實作 **迭代器**（Iterators）的物件，才能使用展開運算子！

- 錯誤範例

```
var obj = {"key1":"value1"};
function myFunction(x) {
    console.log(x); // undefined
}
myFunction(...obj);

var args = [...obj];
console.log(args, args.length) //[] 0
```

其餘參數 (Rest parameters) (...)

- 其餘參數 (Rest parameters) 是 ES6 新的函式特性
- 傳入函式的其餘參數

```
function f(a, b){
  var args = Array.prototype.slice.call(arguments, f.length);
  return args;
}
f(1,2,3,4,5,6); // [3, 4, 5, 6]
```

- 傳入函式的其餘參數，使用 Rest parameters 語法 (...)

```
function f(a, b, ...args){
  return args;
}
f(1,2,3,4,5,6); // [3, 4, 5, 6]
```

其他語言特性

- 型別相容性 (Type Compatibility)
- 進階型別宣告 (Advanced Types)
  - 複合型別 (Union Types)

```
function padLeft(value: string, padding: string | number) {
```

- 命名空間 (Namespaces)
  - 可在多個檔案中使用同一個命名空間
- 命名空間與模組 (Namespaces and Modules)
- 模組解析 (Module Resolution)
- 宣告檔合併 (Declaration Merging)
- 撰寫宣告檔 (Writing Declaration Files)
- 三個斜線指令 (Triple-Slash Directives)

相關連結

- TypeScript Handbook (中文版)
- TypeScript - JavaScript that scales.
  - tsconfig.json
  - Compiler Options
  - Playground
- typings
  - Typings Registry
  - Typings Commands
  - Typings Examples
  - Typings FAQ
  - How Typings Makes External Modules First Class
  - From TSD to Typings (以前用 tsd 的人務必要看這篇文章改用 typings)
- 小編幫你踩地雷系列: TypeScript 適合我嗎
- ECMAScript 6入門

聯絡資訊

- The Will Will Web

  記載著 Will 在網路世界的學習心得與技術分享
  - http://blog.miniasp.com/
- Will 保哥的技術交流中心 (臉書粉絲專頁)

  - http://www.facebook.com/will.fans
- Will 保哥的噗浪
  - http://www.plurk.com/willh/invite
- Will 保哥的推特
  - https://twitter.com/Will_Huang