



# Angular 4 開發實戰：進階開發篇

深入了解 Angular Router 路由機制



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/>



Angular Routing: Getting Started

# ANGULAR 路由 - 新手上路

# 關於 Angular 路由機制

- 關於 Angular 的元件架構
  - 整個應用程式是以一個**樹狀結構**的**元件**組成
  - **每個頁面**都可以切成多個**可重複使用**的 UI 元件
  - **每個頁面**就是一個 UI 元件
  - 最上層 UI 元件 (根元件) 就是 **AppComponent**
- 關於 Angular 路由的主要用途
  - 負責**重新配置**頁面中應該顯示哪些 UI 元件
  - 負責儲存頁面中 UI 元件的**配置狀態** (**路由狀態**)
  - **路由狀態**定義著頁面上應該顯示哪些 UI 元件
  - **路由狀態**最重要的就是紀錄**網址路徑**與**元件**之間的關係
  - 使用預設 **PathLocationStrategy** 路由策略時，最重要的設定
    - 就是 **src/index.html** 裡面的 **<base href="/">**
    - 切換路由時都是透過 URL 進行切換，因此**相對基底路徑**很重要！

# 示範 Angular Router 的效果

<http://themicon.co/theme/centric/v1.5/angular2/>



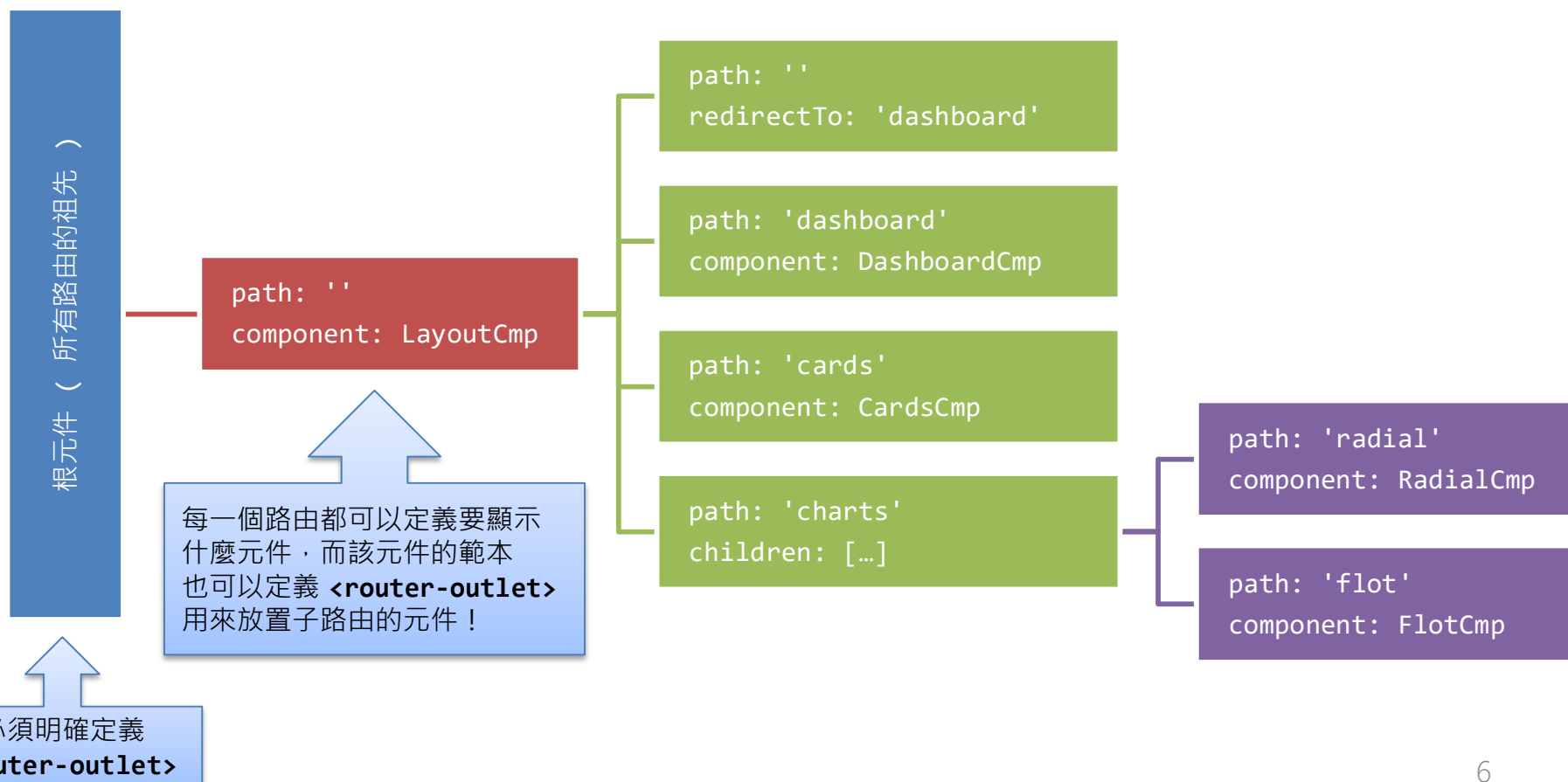
# 關於 Angular 路由定義

- 路由定義也是一種樹狀結構，並預先定義路由狀態

```
{
  path: '',
  component: LayoutCmp,
  children: [
    { path: '', redirectTo: 'dashboard' },
    { path: 'dashboard', component: DashboardCmp },
    { path: 'cards/:type', component: CardsCmp },
    {
      path: 'charts',
      children: [
        { path: 'radial/:radius', component: RadialCmp },
        { path: 'flot/:dots', component: FlotCmp }
      ]
    }
  ]
}
```

# 路由插座 ( <router-outlet> )

- 路由插座 ( <router-outlet> ) 是放置元件的地方



# 實作簡單的第一層路由 - 檔案結構

- 先查看幾個重要檔案
  - app.component.ts
    - 應用程式的根元件 (必要)
  - app.component.html
    - 應用程式的根元件範本 (必要)
    - 這裡會有一個路由插座 ( `<router-outlet></router-outlet>` )
  - app.module.ts
    - 應用程式的 `AppModule` 為主要啟動模組
    - `AppRoutingModule` 則是從外部匯入的模組
  - app-routing.module.ts
    - 在 `AppRoutingModule` 模組內僅包含 Angular 路由定義
    - 必須在 `@NgModule()` 設定 **imports** 把路由定義載入
      - `RouterModule.forRoot(routes)`

# 實作簡單的第一層路由 - 建立新元件

- 建立兩個元件
  - `ng g c page1`
  - `ng g c page2`
- 修改 `app-routing.module.ts` 路由定義

```
const routes: Routes = [ {  
    path: 'page1', component: Page1Component,  
    path: 'page2', component: Page2Component  
}  
];
```
- 開啟 <http://localhost:4200/> 進行預覽
  - 開啟 <http://localhost:4200/page2> 進行預覽
  - 開啟 <http://localhost:4200/error> 進行預覽



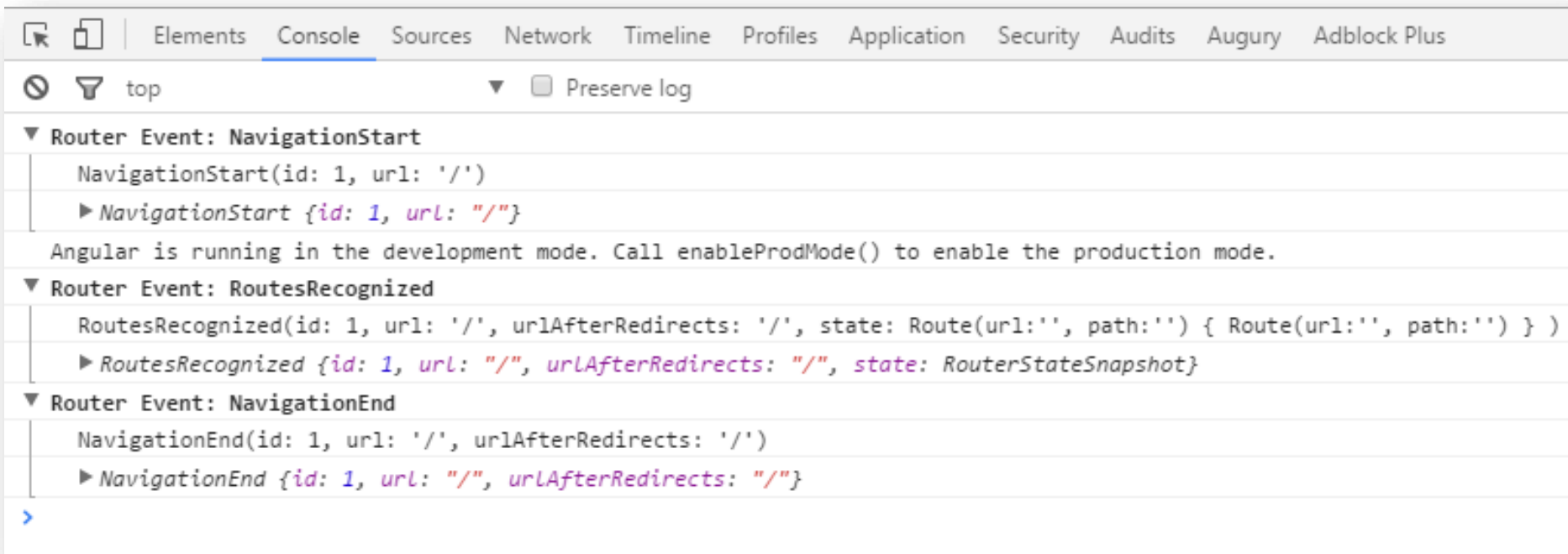
# 啟用路由追蹤

- 修改 app-routing.module.ts 路由定義

```
@NgModule({  
  imports: [RouterModule.forRoot(routes, {  
    enableTracing: true  
  })],  
  exports: [RouterModule],  
  providers: []  
})  
export class AppRoutingModule { }
```

# 路由追蹤的偵錯資訊

- 按下 F12 進入 Console 畫面即可查看路由追蹤資訊



The screenshot shows the Chrome DevTools Console with the 'Console' tab selected. The filter is set to 'top' and 'Preserve log' is checked. The console displays three Router events: NavigationStart, RoutesRecognized, and NavigationEnd. Each event is expanded to show its details and a corresponding log message.

```
▼ Router Event: NavigationStart
  NavigationStart(id: 1, url: '/')
  ▶ NavigationStart {id: 1, url: "/"}
```

Angular is running in the development mode. Call enableProdMode() to enable the production mode.

```
▼ Router Event: RoutesRecognized
  RoutesRecognized(id: 1, url: '/', urlAfterRedirects: '/', state: Route(url:'', path:='') { Route(url:'', path:='') } )
  ▶ RoutesRecognized {id: 1, url: "/", urlAfterRedirects: "/", state: RouterStateSnapshot}
```

```
▼ Router Event: NavigationEnd
  NavigationEnd(id: 1, url: '/', urlAfterRedirects: '/')
  ▶ NavigationEnd {id: 1, url: "/", urlAfterRedirects: "/"}
```

>

# 預設 PathLocationStrategy 路由策略

- 預設 app-routing.module.ts 路由定義

```
@NgModule({  
  imports: [RouterModule.forRoot(routes)],
```

- 網址結構
  - **http://localhost:4200/dashboard/**
- 採用技術
  - [HTML5 History API](#) ( [MDN](#) )
    - 支援瀏覽器：**Internet Explorer 10+**, Safari 5+, iOS 4, Firefox 4+, Google Chrome
    - [The State of the HTML5 History API](#)

# 改用 HashLocationStrategy 路由策略

- 修改 app-routing.module.ts 路由定義

```
@NgModule({  
  imports: [RouterModule.forRoot(routes, {  
    useHash: true  
  })],
```

- 網址結構
  - `http://localhost:4200/#/dashboard/`
- 採用技術
  - HTML4 標準 Hash 網址格式 (IE6+)

# 在路由之間建立超連結 (1)

- app.component.html
  - 路由連結 (加上中括號可繫結一個陣列物件)

```
<ul>  
  <li><a [routerLink]="['/']">Home</a></li>  
  <li><a [routerLink]="['/page1']">Page1</a></li>  
  <li><a [routerLink]="['/page2']">Page2</a></li>  
</ul>
```
  - 路由插座
    - `<router-outlet></router-outlet>`
  - 注意事項
    - 不能寫成 `<router-outlet />` 喔！

# 在路由之間建立超連結 (2)

- app.component.html
  - 路由連結 (不用中括號就只能繫結固定的字串資料，不能繫結變數)

```
<ul>  
  <li><a routerLink="/">Home</a></li>  
  <li><a routerLink="/page1">Page1</a></li>  
  <li><a routerLink="/page2">Page2</a></li>  
</ul>
```
  - 路由插座
    - <router-outlet></router-outlet>
  - 注意事項
    - 不能寫成 <router-outlet /> 喔！

# 在路由連結套用 Active 樣式

- app.component.css

```
.active {
  background-color: yellow;
}
```
- app.component.html
  - 路由連結 (不用中括號就只能繫結固定的字串資料)

```
<ul>
  <li><a routerLink="/"
      routerLinkActive="active">Home</a></li>
  <li><a routerLink="/page2"
      routerLinkActive="active">Page2</a></li>
</ul>
```
  - 查看 <http://localhost:4200/page2> 網址，看是否有套用 **active** 類別

# 路由連結如何套用 routerLinkActive

- 路由定義是樹狀結構資料
  - 當網址路由在比對 routerLink 時，會從根路由開始比對起！
  - 假設我們有兩個不同路由的頁面
    - /
    - /page2
  - 網址路由停在 / 的時候 routerLinkActive 會這樣比對路由：
    - 比對 / 路由連結，比對成功
    - 比對 /page2 路由連結，無法比對到目前網址 ( / )，比對失敗
  - 網址路由停在 /page2 的時候 routerLinkActive 會這樣比對路由：
    - 比對 / 路由連結 (由於 / 可以成功比對 /page2 網址)，比對成功
    - 比對 /page2 路由連結，比對成功
- 核心概念
  - 預設 routerLinkActive 會套用到頁面的上層路徑
  - 非常適合用在選單類型的類別套用



# 避免套用 routerLinkActive 到上層路徑

- 解決方案

- 只要在路由連結 ( routerLink ) 套用以下 Directive 即可

`[routerLinkActiveOptions]="{exact: true}"`

- 使用範例

```
<ul>
  <li><a routerLink="/" routerLinkActive="active"
    [routerLinkActiveOptions]="{exact: true}">
    Home</a>
  </li>
  <li><a routerLink="/page1" routerLinkActive="active">
    Page1</a>
  </li>
</ul>
```

- 注意事項

- routerLinkActiveOptions 目前也只有一種寫法 ( `{ exact: true }` )

# 定義預設路由 (default route)

```
import { Routes, RouterModule } from '@angular/router';

import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';

const routes: Routes = [
  { path: '', component: Page1Component },
  { path: 'page1', component: Page1Component },
  { path: 'page2', component: Page2Component }
];
```

# 定義轉向路由 (redirect route)

```
import { Routes, RouterModule } from '@angular/router';

import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';

const routes: Routes = [
  { path: '', redirectTo: '/page1', pathMatch: 'full' },
  { path: 'page1', component: Page1Component },
  { path: 'page2', component: Page2Component }
];
```

- 注意事項
  - 使用 **redirectTo** 時，一定要加上 **patchMatch** 屬性
  - 轉向路由最多**只能轉向一次**，**不能**從 / 轉到 /page1 然後再轉向 /page2

# 定義萬用路由 (wildcard route)

```
import { Routes, RouterModule } from '@angular/router';

import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';

const routes: Routes = [
  { path: '', redirectTo: '/page1', pathMatch: 'full' },
  { path: 'page1', component: Page1Component },
  { path: 'page2', component: Page2Component },
  { path: '**', redirectTo: '/page1', pathMatch: 'full' }
];
```

- 注意事項
  - 使用萬用路由時，一定要放在最後一個路由定義中！

# 定義獨立的 Route 物件

```
import { Route, Routes, RouterModule } from '@angular/router';

import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';

const fallbackRoute: Route = {
  path: '**', redirectTo: '/page1', pathMatch: 'full'
};

const routes: Routes = [
  { path: '', redirectTo: '/page1', pathMatch: 'full' },
  { path: 'page1', component: Page1Component },
  { path: 'page2', component: Page2Component },
  fallbackRoute
];
```

# 定義獨立的 Route 物件 & JS 模組

- src/app/shared/fallback-route.ts

```
import { Route } from '@angular/router';  
export const fallbackRoute: Route = {  
  path: '**', redirectTo: '/page1', pathMatch: 'full'  
};
```

- app-routing.module.ts

```
import { fallbackRoute } from './shared/fallback-route';  
const routes: Routes = [  
  { path: '', redirectTo: '/page1', pathMatch: 'full' },  
  { path: 'page1', component: Page1Component },  
  { path: 'page2', component: Page2Component },  
  fallbackRoute  
];
```

# 認識 Router 服務元件

- 一個含有路由機制的 Angular 應用程式
  - 會有個 **Router** 服務元件 (singleton instance) (可以在任何地方注入)
    - 當網址發生改變，就會從 **Router** 中找出相對應的 **Route** 物件
    - 你可以從 **Router** 服務元件找出所有的路由資訊
    - 你可以從 **Router** 服務元件取出 **ActivatedRoute** (目前套用的路由)
    - 你可以從 **Router** 服務元件取出 **RouterState** (路由狀態)
  - **Routes** 物件是 **Route** 物件的集合 (陣列型態)
  - **Route** 路由物件就是每一個路由的細部設定
    - 要顯示哪一個元件 (Component Directive )
    - 要轉址到哪一個路徑 (Redirection )
    - 設定是否允許進入路由 (例如: 實作權限管理) 或 是否允許離開路由
  - 定義路由可以靠 **RouterModule.forRoot** 方法幫我們建立路由

# 認識 Routes 物件

- **Routes** 物件就是 **Route** 物件的集合 (陣列型態)
  - 陣列中的物件是有順序性的
- **Routes** 物件裡的每個 **Route** (路由) 順序很重要
  - 當瀏覽器輸入網址與路徑
  - 路由機制就會透過 **Routes** 物件依序比對每一個 **Route** 路由物件
  - 只要有一個 **Route** 路由被選中，就不會再比對下一個路由
  - 萬用路由 (wildcard route) 一定要放在 **Routes** 物件的最後一個元素



# 認識 Route 物件

- 每個 Route 物件 (物件型態)
  - 會有個 **path** 屬性代表網址路徑的片段 (必要屬性)
    - 這裡的 **path** 屬性**不需要**加上斜線開頭 ( / )
    - 這裡的 **path** 屬性可以加上 **:參數名稱** 來代表一個路由參數  
範例：`{ path: 'hero/:id', component: HeroDetailComponent }`  
如果瀏覽器上的網址路徑為 `/hero/33` 時，路由參數 `id` 的值就會是 `33`
  - 會有個 **component** 屬性，代表啟用這個路由時會建立的**元件**
  - 會有個 **data** 屬性，代表預計傳入元件的**路由資料**
    - 路由參數是從網址路徑傳入 **ActivatedRoute** 物件
    - 路由資料是從路由定義傳入 **ActivatedRoute** 物件
    - 路由資料通常是唯讀/靜態的資料，例如頁面標題、權限角色、麵包屑...
    - 動態的路由資料可以透過 [resolve guard](#) 來取得動態資料

# 回顧這個單元

- RouterModule
  - Router
  - Routes
  - Route
  - RouterOutlet
  - RouterLink
  - RouterLinkActive
  - RouterLinkActiveOptions
- 
- ActivatedRoute
  - RouterState



Angular Themes & Routing

# ANGULAR 練習套版與設定路由

Angular Routing in Action

# ANGULAR 路由 - 實戰演練



# 套版注意事項

- index.html
  - 將 `<head>` 全部移入 index.html
  - 將 `<script>` 全部移入 index.html
  - `<body class="theme-1">`
- app.component.html
  - 將 `<router-outlet></router-outlet>` 放入 `<!-- Page content-->`
  - 超連結 `"#"` 要改成 `"javascript:"`
    - 因為 `<base href="/" />` 的關係
    - [javascript - Url Hash with Html Base Tag - Stack Overflow](#)
  - routerLinkActive 可以套用在 `<a>` 上層的 `<li>` 標籤
    - [Allow routerLinkActive to work with descendant routerLinks #9777](#)
- app-routing.module.ts (建立 dashboard 與 cards 元件)
  - `const routes: Routes = [`
  - `{ path: 'dashboard', component: DashboardComponent },`
  - `{ path: 'cards', component: CardsComponent },`
  - `fallbackRoute`
  - `];`

Angular Child Routing

# ANGULAR 子路由



# 定義 /charts 子路由骨架

```
const routes: Routes = [  
  { path: 'dashboard', component: DashboardComponent },  
  { path: 'cards', component: CardsComponent },  
  { path: 'charts',  
    children: [  
      <<放置 Route 路由設定>>  
    ],  
  },  
  fallbackRoute  
];
```

# 定義 /charts 子路由

```
const routes: Routes = [  
  { path: 'dashboard', component: DashboardComponent },  
  { path: 'cards', component: CardsComponent },  
  { path: 'charts',  
    children: [  
      { path: '', redirectTo: 'flot', pathMatch: 'full' },  
      { path: 'flot', component: FlotComponent },  
      { path: 'radial', component: RadialComponent },  
      { path: 'rickshaw', component: RickshawComponent }  
    ],  
  },  
  fallbackRoute  
];
```



Angular Routing Parameters

# ANGULAR 路由參數



# 設定含有路由參數的路由定義

- 在網址路徑 (path) 上面設定參數的格式如下

```
const routes: Routes = [  
  { path: 'dashboard', component: DashboardComponent },  
  { path: 'cards/:type', component: CardsComponent },
```

- 注意事項
  - 只要有在 **path** 設定**路由參數**，網址列上就一定要出現該參數值
  - 如果**網址路徑**沒有傳入**路由參數**，該路由會比對失敗

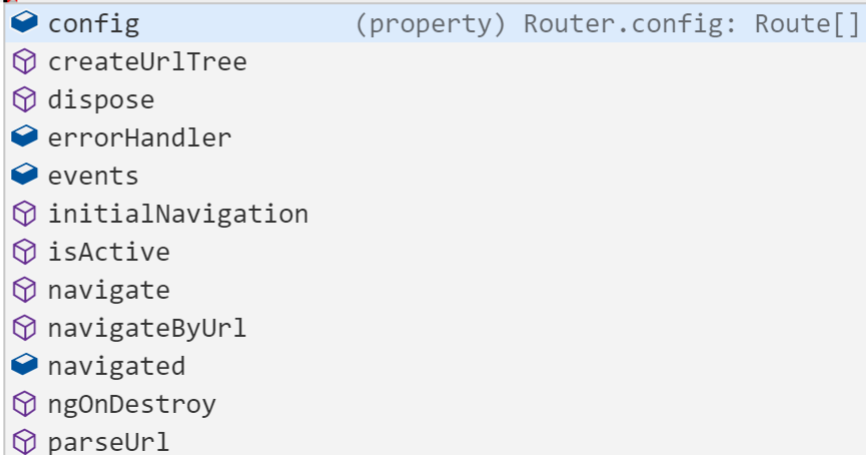
# 注入 Router 與 ActivatedRoute 元件

```
import { ActivatedRoute, Router } from '@angular/router';  
  
export class CardsComponent implements OnInit {  
  
    constructor(private router: Router,  
                private route: ActivatedRoute) { }  
  
    ngOnInit() {  
    }  
  
}
```

# 關於 Router 服務元件

- 可取得完整的路由資訊
  - config
  - events
  - navigated
- 可透過導覽 API 進行導覽
  - navigate()
  - navigateByUrl()

```
export class CardsComponent implements OnInit {  
  
  constructor(private router: Router, private route: ActivatedRoute) { }  
  
  ngOnInit() {  
    console.log(this.router.);  
  }  
}
```

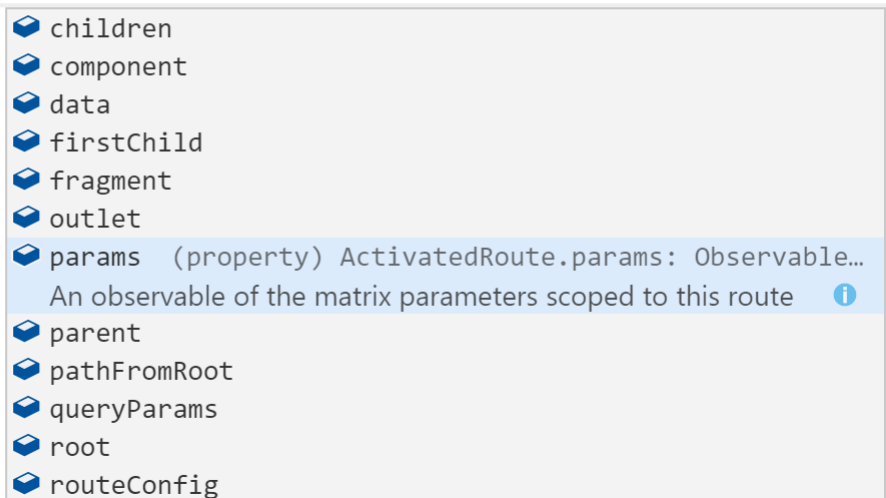


The screenshot shows a code completion dropdown menu for the expression `this.router.` in the `ngOnInit()` method. The dropdown lists various properties and methods of the `Router` class, including `config`, `createUrlTree`, `dispose`, `errorHandler`, `events`, `initialNavigation`, `isActive`, `navigate`, `navigateByUrl`, `navigated`, `ngOnDestroy`, and `parseUrl`. The `config` property is highlighted, with its type `(property) Router.config: Route[]` displayed to the right.

# 關於 ActivatedRoute 服務元件

- 代表**目前正被啟用**的路由物件
- 可以從這個物件取得相當豐富的路由資訊 (包含路由參數)

```
export class CardsComponent implements OnInit {  
  
  constructor(private router: Router, private route: ActivatedRoute) { }  
  
  ngOnInit() {  
    this.route.  
  }  
}
```



The screenshot shows an IDE with a code editor and a dropdown menu. The code editor contains the following code:

```
export class CardsComponent implements OnInit {  
  
  constructor(private router: Router, private route: ActivatedRoute) { }  
  
  ngOnInit() {  
    this.route.  
  }  
}
```

The dropdown menu is open, showing a list of properties of the `ActivatedRoute` object. The `params` property is highlighted. The description for `params` is: `(property) ActivatedRoute.params: Observable...` and `An observable of the matrix parameters scoped to this route`. There is also an information icon (i) next to the description.

# 透過程式進行路由導覽 (Navigation)

- 注入路由服務

- constructor(private **router**: Router,  
private **route**: ActivatedRoute) { }

- 路由導覽 APIs 的三種常見用法

- 絕對位址導覽

- `this.router.navigateByUrl('/home/1');` // 傳入字串
    - `this.router.navigate(['home', '1']);` // 傳入陣列

- 相對位址導覽 (相對於目前的路由的網址路徑)

- `this.router.navigate(['..', '1'], {  
relativeTo: this.route  
});`

# 練習透過 ActivatedRoute 取得路由參數

- 注入路由服務 (card.component.ts)
  - constructor(private **router**: Router,  
private **route**: ActivatedRoute) { }
- 透過 **snapshot** (快照) 取得**執行當下**的參數值 (僅初始值)
  - this.route.**snapshot**.params['type']
- 透過 **params** 這個 Observable 物件來取得參數值 (**比較常用**)
  - this.route.**params**.subscribe(params => {  
    console.log(params['type']);  
});
  - 注意 : this.route.params 是一個 Observable 物件

# Angular 路由下的元件生命週期

- 在不同的路由之間切換時
  - 路由所指向的元件會先 Destroy 舊的，然後 Create 新的
  - 每次**切換不同路由**時，都會執行一次 `ngOnInit` 事件
- 在相同的路由之間變換參數時
  - 在切換路由時，網址列雖然會變動，但依然是相同的路由
  - 由於路由並沒有切換 (僅參數改變)，因此元件並**不會重新產生新的**
  - **由於元件並沒有產生新的，因此 `ngOnInit` 事件並不會重新執行！**
  - 不會執行 `ngOnInit` 事件，該元件就沒有任何程式碼會自動執行！
- **解決方案**
  - 必須靠訂閱 `this.route.params` 這個 Observable 來偵測變更！
  - **練習**：在同一個路由/元件切換不同的路由參數



# 取得路由參數的開發技巧 (switchMap)

- 透過 RxJS 的 [switchMap](#) 將一個 Observable 轉成另一個
  - `import 'rxjs/add/operator/switchMap';`
  - `constructor(private router: Router,  
private route: ActivatedRoute) { }`
  - 從第一個 Observable 得到的值傳給 `switchMap`
  - `switchMap` 會先接到事件資料，並回傳一個新的 Observable 物件

```
this.route.params
  .switchMap((params: Params) => {
    return this.service.getHero(+params['id'])
  })
  .subscribe((hero: Hero) => this.hero = hero);
```

# 使用 ActivatedRoute 的注意事項

- 大部分的 Observable 物件在**訂閱**之後都需要執行**取消訂閱**，只有少數例外不需要！
- 在 ActivatedRoute 物件中所有的 Observable 屬性，都不需要特別執行**取消訂閱**這個動作。
- ActivatedRoute 會隨著元件建立時建立新的物件實體並注入，也會隨著元件在摧毀時自動消失，因此你在訂閱像是 `this.route.params` 這個 Observable 之後，**不需要做任何取消訂閱**的事。

# 如何取得上層路由的路由參數

- 網址路徑

- `http://localhost:4200/product-details/3`

- 路由範例

```
export const routes: Routes = [  
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },  
  { path: 'product-list', component: ProductList },  
  { path: 'product-details/:id', component: ProductDetails,  
    children: [  
      { path: '', component: Overview },  
      { path: 'specs', component: Specs } // 選中的路由  
    ]  
  } ];
```

- 取得上層路由 ( 'product-details/:id' ) 的路由參數 id

```
this.route.snapshot.params['id'];
```

```
this.route.parent.snapshot.params['id'];
```

# 可選的路由矩陣參數

- 可選的路由參數 ( 使用 [matrix URL notation](#) 表示法 )
  - 用一個類似 QueryString 的格式 (但把 **?** 與 **&** 都改成 **;** 符號)
  - 網址格式
    - `http://localhost:4200/cards/1;name=Will;location=Taipei`
  - 程式寫法 (傳入到 `navigate` 方法第一個陣列參數的最後一個元素)
    - `this.router.navigate(['/cards', id, {  
          name: 'Will', location: 'Taipei'  
          }]);`
  - 範本寫法
    - `<a [routerLink]="['/cards', 1, {name: 'Will'}]">Card</a>`
  - 取得參數的方式
    - `this.route.snapshot.params`
    - `this.route.params`

# 可選的路由查詢字串參數

- 可選的路由參數 ( 使用 [Query String](#) 表示法 )
  - 使用 QueryString 的格式
  - 網址格式
    - `http://localhost:4200/cards/1?name=Will&location=Taipei`
  - 程式寫法 (傳入到 `navigate` 方法第二個參數的 `queryParams` 屬性)
    - `this.router.navigate(['cards', type], {  
    queryParams: { name: 'Will', location: 'Taipei' }  
});`
  - 範本寫法 ( 混合 Matrix 與 QueryString 參數 )
    - `<a [routerLink]="['/cards', 1, {name: 'Will'}]"  
    [queryParams]="{ page: 99 }">Card</a>`
  - 取得參數的方式
    - `this.route.snapshot.queryParams`
    - `this.route.queryParams`

# 回顧這個單元

- `import { ActivatedRoute, Router, Params } from '@angular/router';`
- `private router: Router`
- `private route: ActivatedRoute`
  
- `this.router.navigateByUrl('/home/1');`
- `this.router.navigate(['home', '1']);`
- `this.router.navigate(['1'], { relativeTo: this.route });`
- `this.router.navigate(['/cards', id, { ts: 123 }]);`
- `this.router.navigate(['cards', 1], { queryParams: { ts: 123 } });`
  
- `this.route.snapshot.params['type']`
- `this.route.params.subscribe(params => params['type']);`
  
- `this.route.snapshot.queryParams['ts']`
- `this.route.queryParams.subscribe(params => params['ts']);`

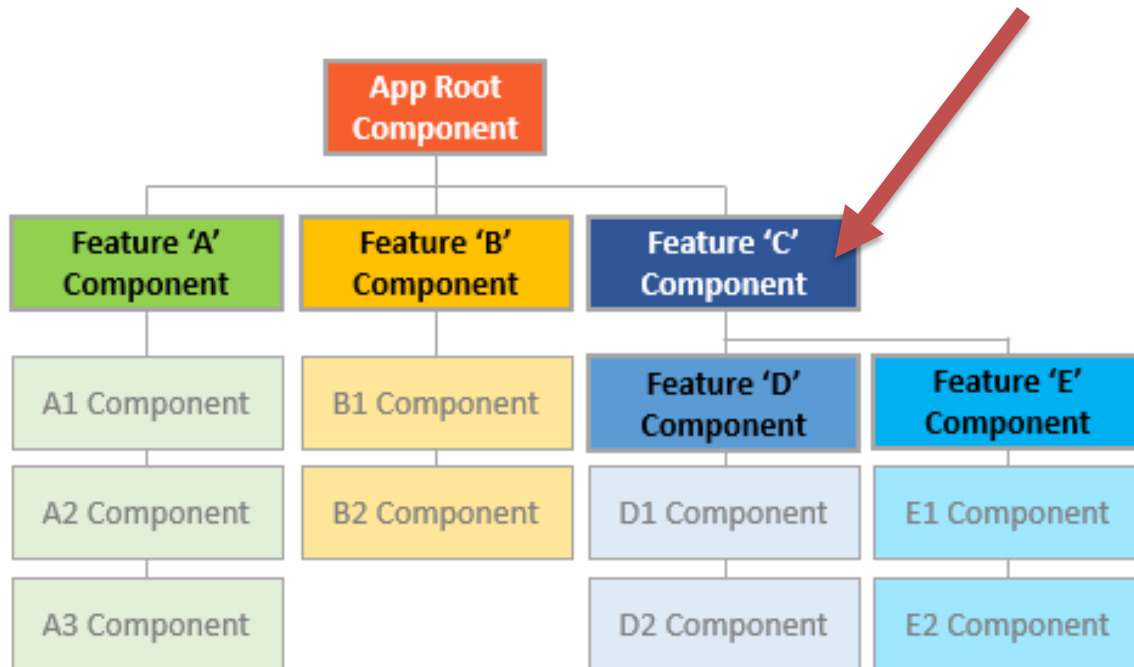
Angular Child Routing Modules

# ANGULAR 子路由模組



# 將應用程式切割更多層功能模組

- 如下圖示，每一個標示為 "Feature" 都是一個功能模組
  - 每個模組都各自擁有一個資料夾
  - 每個模組都有自己的 Module 與 RoutingModule
  - 每個模組都擁有一個預設路由，此路由也可以有預設元件與範本
  - 預設元件的範本可以宣告自己這一層的 **<router-outlet>** 路由插座





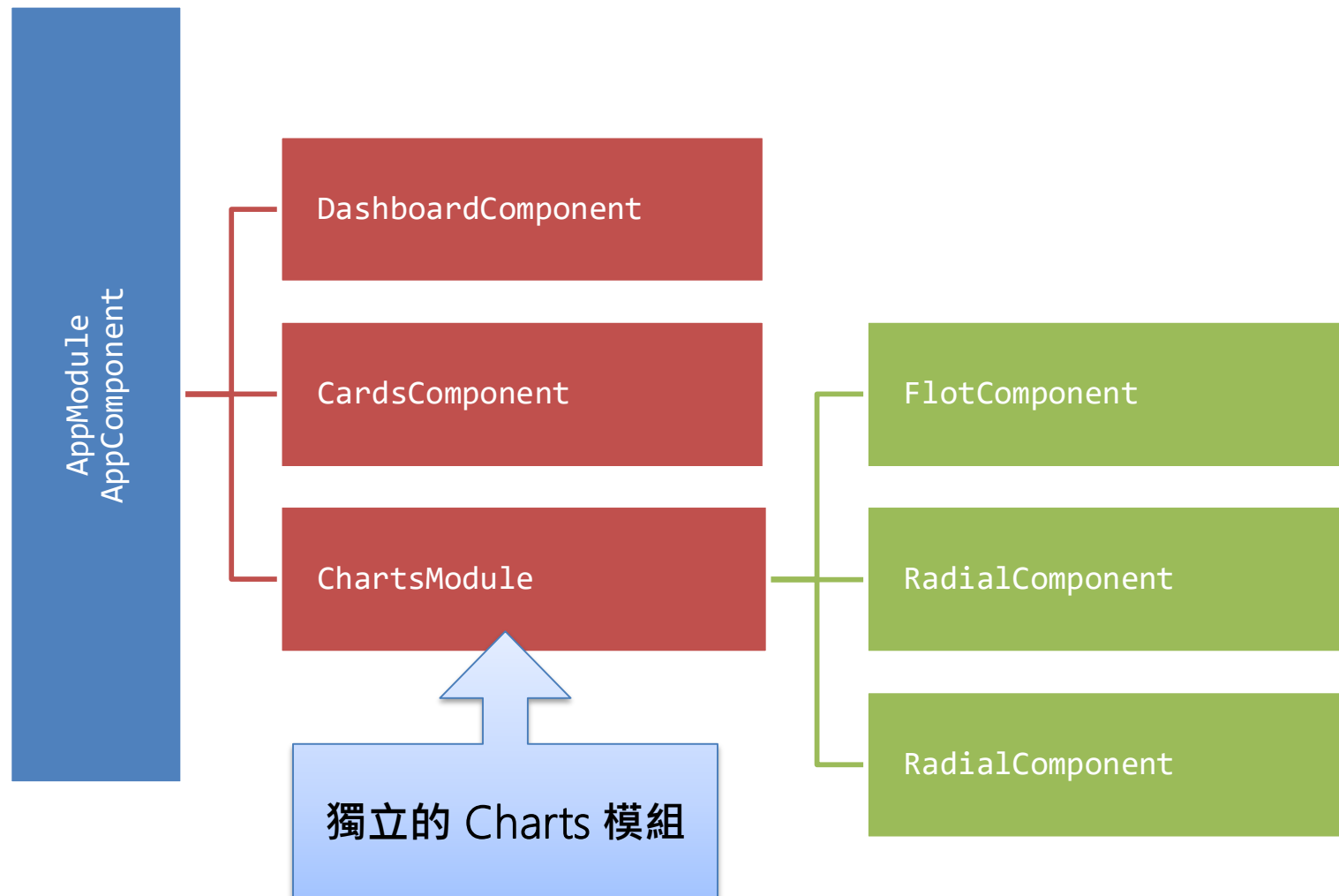
# 建立含路由的 charts 模組

- 執行以下命令
  - `ng g m charts --routing`

問題      輸出      偵錯主控台      終端機

```
PS C:\Projects\ng2-advanced> ng g m charts --routing
installing module
  create src\app\charts\charts-routing.module.ts
  create src\app\charts\charts.module.ts
installing component
  create src\app\charts\charts.component.css
  create src\app\charts\charts.component.html
  create src\app\charts\charts.component.spec.ts
  create src\app\charts\charts.component.ts
  update src\app\charts\charts.module.ts
PS C:\Projects\ng2-advanced> █
```

# 檢視目前的模組與元件架構



# 注意事項

- app.module.ts && charts.module.ts
  - 將應該在 ChartsModule 的元件全部移到 charts.module.ts 中
  - View 相關元件都要註冊在 @NgModule 的 **declarations** 屬性中
  - 移除所有用不到的 JS import 程式碼
- app-routing.module.ts && charts-routing.module.ts
  - 將屬於 charts 相關的路由與子路由都移至 charts-routing.module.ts
- app.module.ts
  - 匯入 ChartsModule 模組到 **imports** 屬性中
    - 模組匯入的順序很重要 (路由是依序套用的)
    - AppRoutingModule 必須被放在最後載入！

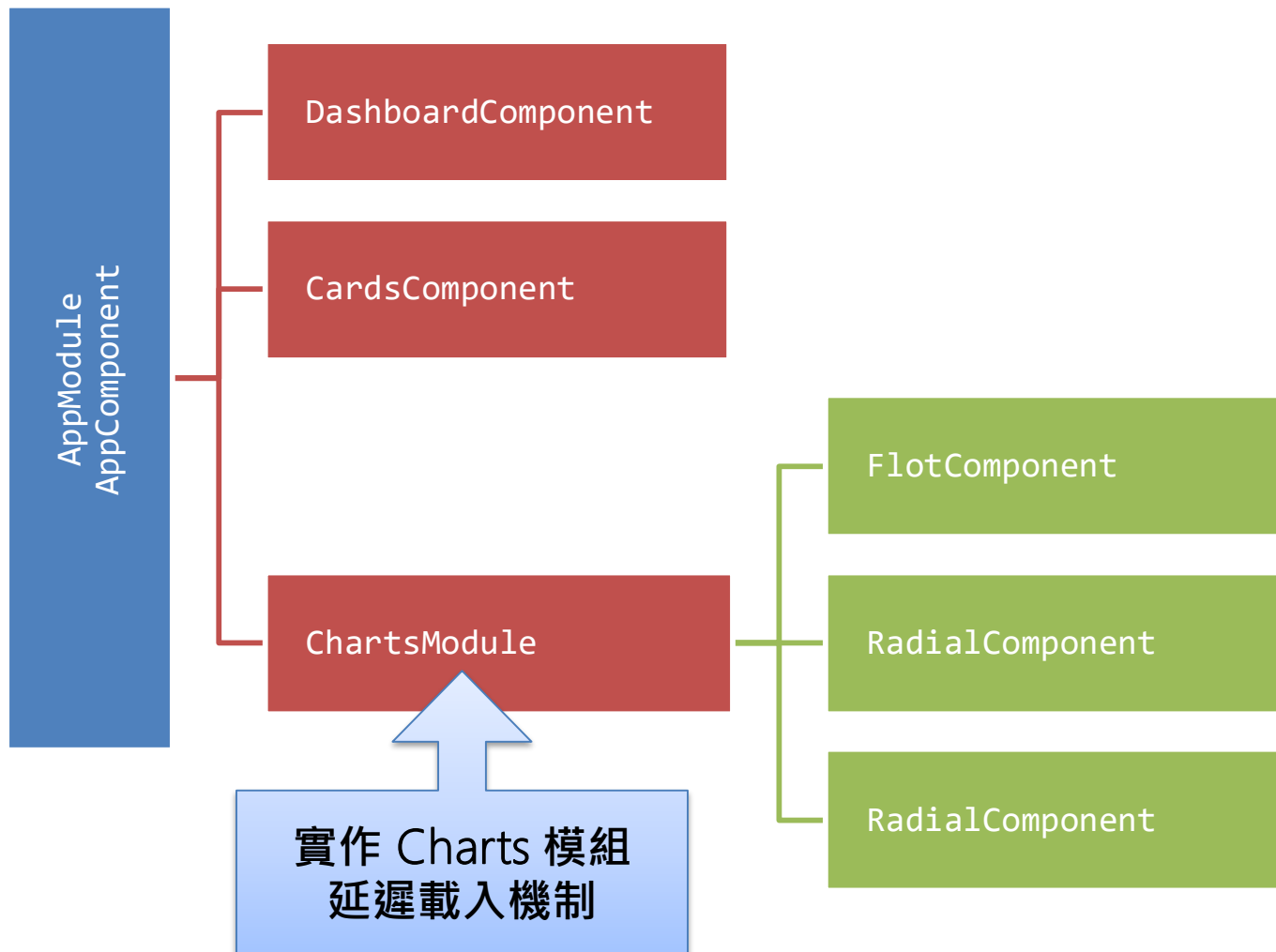
```
],
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule,
  ChartsModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```



Angular Lazy-loading Modules

# ANGULAR 實作模組延遲載入


# 檢視目前的模組與元件架構



# 模組延遲載入的程式碼結構

- `app-routing.module.ts`

```
const routes: Routes = [  
  { path: 'dashboard', component: DashboardComponent },  
  { path: 'cards/:type', component: CardsComponent },  
  { path: 'charts',  
    loadChildren: './charts/charts.module#ChartsModule' },  
  fallbackRoute ];
```



模組檔案路徑

模組類別名稱

- `charts-routing.module.ts`

```
const routes: Routes = [  
  { path: '', redirectTo: 'flot', pathMatch: 'full' },  
  { path: 'flot', component: FlotComponent },  
  { path: 'radial', component: RadialComponent },  
  { path: 'rickshaw', component: RickshawComponent } ];
```

# 模組延遲載入的優點

- 使用者需要甚麼功能，就只會下載特定功能的程式回來
  - 在大型網站應用程式專案可**有效降低頁面下載的大小**
- 加快頁面的載入速度，瀏覽器僅需下載必要的模組回來
  - 不用讓使用者在**第一頁**下載整個網站的所有模組、元件原始碼
- 每次載入特定功能模組，僅下載遺漏的模組程式碼
  - 大幅優化每個功能模組的載入速度

# 驗證模組延遲載入是否生效

- 測試延遲載入的方式可以用以下命令 (必須重新執行)
  - `ng serve`
  - `ng serve --prod`
  - `ng serve --aot`
  - `ng serve --prod --aot`
  - `ng build`
  - `ng build --prod`
  - 如果看到 `0.xxx.chunk.js` 檔案大小低於 10kB 應該就是有問題！

```
** NG Live Development Server is running on http://localhost:4200. **  
Hash: e67656752c0cce29e6da  
Time: 31253ms  
chunk    {0} 0.faf3775cdbe3f82e4b47.chunk.js 4.68 kB {2} [rendered]  
chunk    {1} polyfills.8f6fddf0a46675365f4f.bundle.js (polyfills) 222 kB {5} [initial] [rendered]  
chunk    {2} main.7c28a839921c4b6436c4.bundle.js (main) 458 kB {4} [initial] [rendered]  
chunk    {3} styles.d41d8cd98f00b204e980.bundle.css (styles) 69 bytes {5} [initial] [rendered]  
chunk    {4} vendor.46c0dd402d85bbb9078f.bundle.js (vendor) 1.69 MB [initial] [rendered]  
chunk    {5} inline.c88669cb2061be7876f1.bundle.js (inline) 0 bytes [entry] [rendered]  
webpack: Compiled successfully.
```



# 啟用模組延遲載入的必要條件

- 原本的 AppModule **不能**跟 ChartsModule 有**任何相依關係**

```
6
7 import { AppComponent } from './app.component';
8 import { CardsComponent } from './cards/cards.component';
9 import { DashboardComponent } from './dashboard/dashboard.component';
10
11 - import { ChartsModule } from './charts/charts.module';
12 -
13 @NgModule({
14   declarations: [
15     AppComponent,
16     CardsComponent,
17     DashboardComponent
18   ],
19   imports: [
20     BrowserModule,
21     FormsModule,
22     HttpClientModule,
23 -   ChartsModule,
24     AppRoutingModule
25   ],
26   providers: [],
27   bootstrap: [AppComponent]
28 })
29 export class AppModule { }
```

# 啟用模組**預先載入**機制 (非同步載入)

- 找到最上層路由模組 ( app-routing.module.ts )

```
import { PreloadAllModules } from '@angular/router';
```

```
@NgModule({  
  imports: [RouterModule.forRoot(routes, {  
    enableTracing: true,  
    preloadingStrategy: PreloadAllModules  
  })],  
  exports: [RouterModule],  
  providers: []  
})  
export class AppRoutingModuleModule { }
```

# 模組載入機制

## 1. 全部載入 (預設)

- 透過 webpack 會將**所有模組**打包成 1 個 JS 檔案

## 2. 延遲載入

- 透過 webpack 將**部分模組**拆解成**可延遲載入**的程式片段 (chunks)
- 使用 webpack 最為知名的 [code splitting](#) 技術
- 當使用者點開網頁，所有**可延遲載入**的程式片段並**不會預先載入**
- 主動擊進入特定路由，延遲載入的模組才會即時載入

## 3. 預先載入

- 透過 webpack 將**部分模組**拆解成**可延遲載入**的程式片段 (chunks)
- 使用 webpack 最為知名的 [code splitting](#) 技術
- 當使用者點開網頁，所有**可延遲載入**的程式片段會**立即預先載入**
- 預先載入的過程透過**非同步背景下載**，不影響畫面顯示或使用者操作

Route Guards

# ANGULAR 路由守門員



# 建立登入頁面

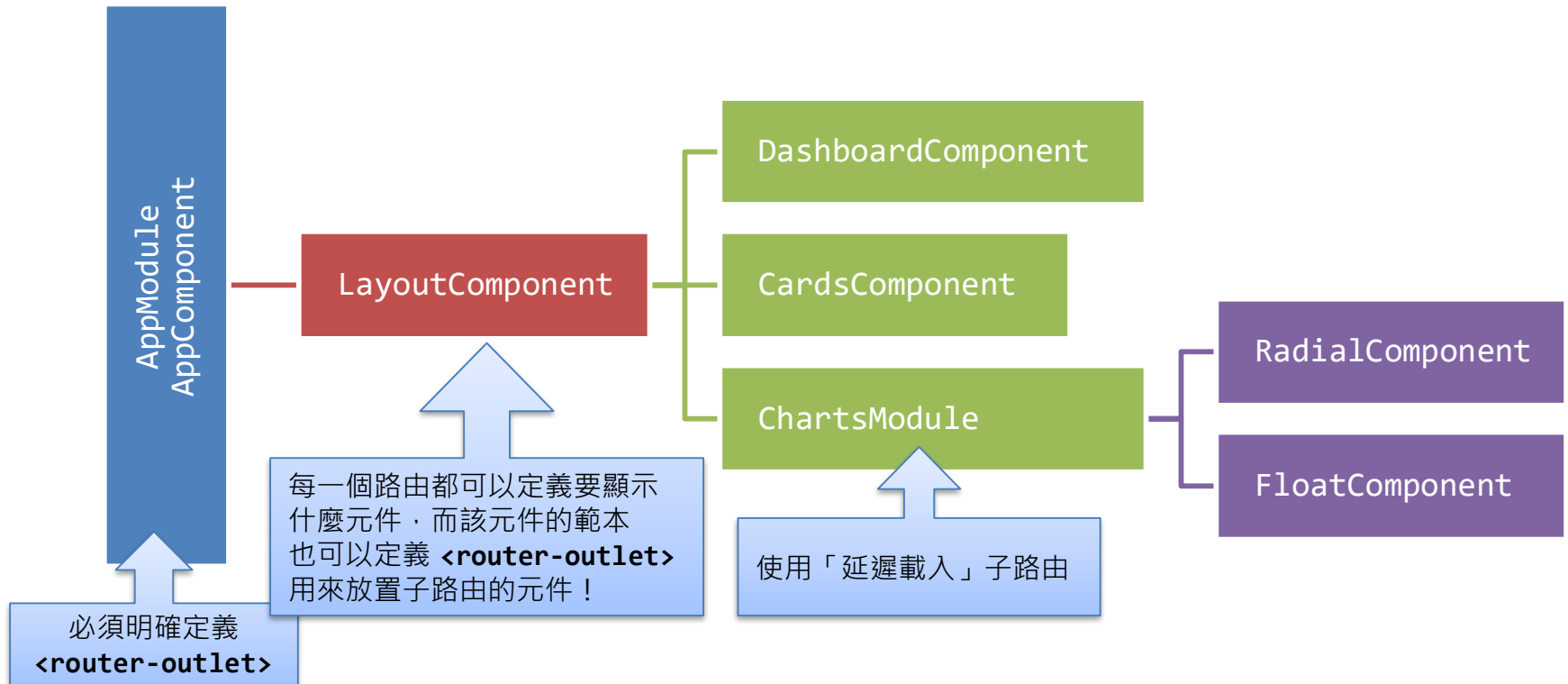
- 建立元件
  - ng g c login
- 設定路由

```
const routes: Routes = [  
  { path: 'login', component: LoginComponent },  
  { path: 'dashboard', component: DashboardComponent },  
  { path: 'cards/:type', component: CardsComponent },  
  { path: 'charts',  
    loadChildren: './charts/charts.module#ChartsModule'  
  },  
]
```

- 設定 Login 元件範本
  - login.component.html

# 多階層 RouterOutlet 示範

- 建立 layout 元件
  - **ng g c layout**



# 建立 LoginRouteGuard 服務元件

- 建立 LoginGuard 服務元件並實作 CanActivate 介面

```
ng g g login
```

- 匯入必要元件

```
import { CanActivate } from '@angular/router';  
import { Injectable } from '@angular/core';
```

- 設定 Injectable 裝飾器在類別上並實作 CanActivate 介面

```
@Injectable()  
export class LoginGuard implements CanActivate {
```

# 實作 CanActivate 介面

- 實作 canActivate() 方法

```
app.module.ts  app-routing.module.ts  login.guard.ts x
1  import { Injectable } from '@angular/core';
2  import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router,
    ActivatedRoute } from '@angular/router';
3  import { Observable } from 'rxjs/Observable';
4
5  @Injectable()
6  export class LoginGuard implements CanActivate {
7      constructor(private router: Router) {}
8      canActivate(
9          next: ActivatedRouteSnapshot,
10         state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
11
12         if(next.queryParams['apikey'] === '123') {
13             return true;
14         } else {
15             this.router.navigateByUrl('/login');
16             return false;
17         }
18     }
19 }
20
```



# 在 'charts' 路由啟用 Route Guards

- 到 **AppModule** 將 **LoginGuard** 加入 **providers**

- providers: [**LoginGuard**],

- 在 'charts' 路由定義加上 **canActivate** 屬性

```
{ path: 'charts',  
  loadChildren: './charts/charts.module#ChartsModule',  
  canActivate: [LoginGuard]  
}
```

# 在 Login 路由實作 CanDeactivate

- 當登入表單打到一半想離開這個路由 (例如回上一頁)，可以透過 CanDeactivate 來限制使用者離開這個頁面
  - `ng g g ensure-login`
  - 匯入必要元件
  - 設定 Injectable 裝飾器在類別上
  - 實作 **CanDeactivate** 介面 (實作 `canDeactivate` 方法)
  - 調整 Login 表單，設定 Template-driven form
    - `@ViewChild(NgForm) form: NgForm;`
    - `[ngModel]="username"`
    - `[ngModel]="password"`
  - 將 **EnsureLoginGuard** 加入到 **AppModule** 中
    - 到 `app.module.ts` 將 **EnsureLoginGuard** 加入 **providers** 屬性中
  - 在 'charts' 路由啟用 Route Guards

# EnsureLoginGuard 實作範例

- 第一個參數 component 代表要離開的那個路由的元件
  - 透過 `component.form.dirty` 取得表單是否有人動過 (dirty)

```
1  import { CanDeactivate, Router, RouterStateSnapshot,   ActivatedRouteSnapshot } from '@angular/router';
2  import { Injectable } from '@angular/core';
3
4  import { Observable } from 'rxjs';
5
6  import { LoginComponent } from '../login/login.component';
7
8  @Injectable()
9  export class EnsureLoginGuard implements CanDeactivate<LoginComponent> {
10     canDeactivate(component: LoginComponent,
11                  route: ActivatedRouteSnapshot,
12                  state: RouterStateSnapshot) {
13         if(component.form.dirty) {
14             return confirm('你確定要放棄登入嗎?');
15         } else {
16             return true;
17         }
18     }
19 }
```

# 相關連結

- [Routing & Navigation – ts](#)
- [Angular 2 Router](#)
- [Routing · Rangle.io : Angular 2 Training](#)
- [Routing in Angular 2 revisited by thoughttram](#)
- [Protecting Routes using Guards in Angular 2](#)
- [Routing in Eleven Dimensions with Component Router – Brian Ford - YouTube](#)



# Angular 4 開發實戰：進階開發篇

深入了解 Angular Form 表單機制



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/>



Template-Driven v.s. Model-Driven Form

# 介紹 ANGULAR 表單開發模型

# Angular 支援兩種表單開發模型

## 以範本為主的表單開發模式 ( Template-Driven Form )

- 要匯入 **FormsModule**
- 都用**宣告**的方式建立表單  
(Declarative Programming)
- 使用 ngModel 指令 (Directive)
- 在**範本**中宣告驗證規則
- 只能對表單進行 E2E 測試

[表單範例](#)

## 以模型為主的表單開發模式 ( Model-Driven Form )

- 要匯入 **ReactiveFormsModule**
- 都用**編程**的方式建立表單  
(Imperative Programming)
- 使用 FormControlName 屬性
- 在**元件**中宣告驗證規則
- 可以對**表單**進行單元測試

[表單範例](#)

# Angular 內建兩種表單模組

- FormsModule (預設)
  - Directives
    - NgModel
    - NgModelGroup
    - NgForm
  - Providers
    - FormControlRegistry
- ReactiveFormsModule
  - Directives
    - FormControlDirective
    - FormGroupDirective
    - FormControlName
    - FormGroupName
    - FormArrayName
  - Providers
    - FormBuilder
    - FormControlRegistry

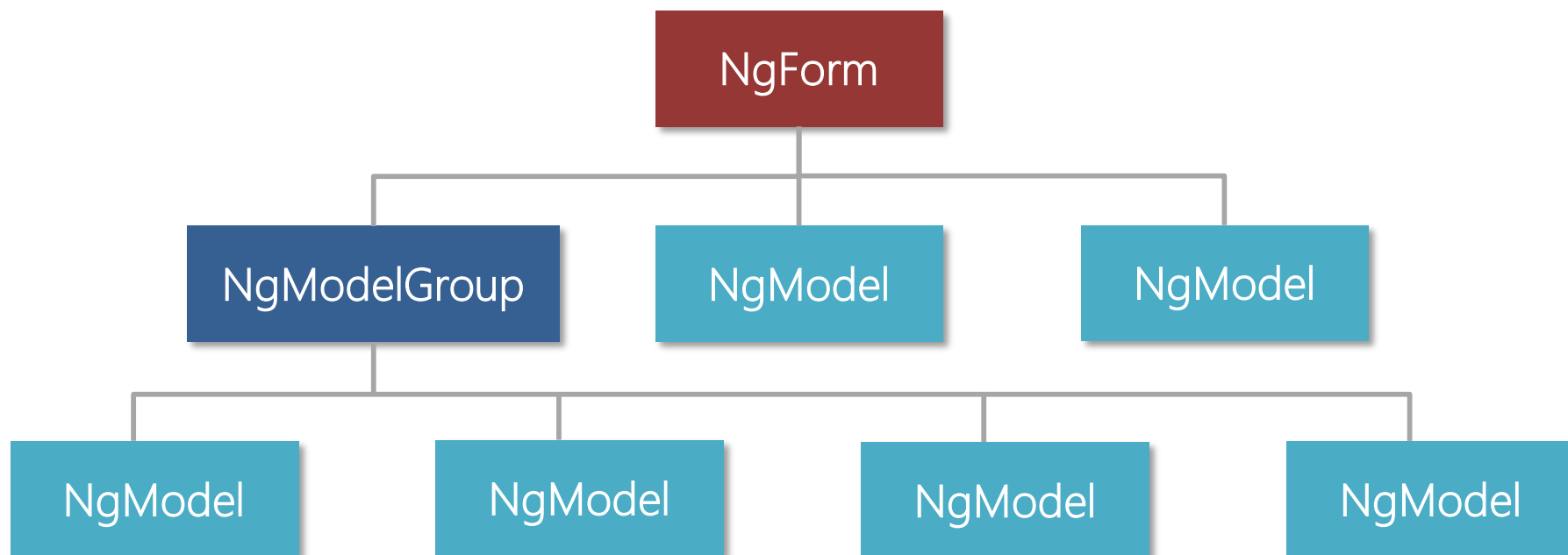




Template-Driven Forms

# 範本驅動表單開發模式

# 範本驅動表單物件架構 (表單模型)



# 重新認識 NgModel 指令 (Directive)

- 初學者的觀念
  - [([ngModel](#))]="name"
  - 建立一個「**雙向繫結**」綁定元件中的特定屬性
- 重新建立觀念
  - [NgModel](#) 主要用來建立一個 **表單控制項** ([FormControl](#)) 實體
  - **表單控制項**的主要用途
    - 用來追蹤使用者在表單欄位輸入或選取的值 (value)
    - 用來追蹤使用者在表單欄位上的互動 (pristine, dirty, touched, untouched)
    - 用來追蹤表單欄位的驗證狀態 (validation status)
    - 用來維持與元件中 Model 進行同步 (**單向屬性繫結**或**雙向繫結**)

# 如何建立表單控制項實體 (instance)

- 不綁定直接建立實體 **ngModel**

```
<input name="username" ngModel>
```

- 單向綁定 ( one-way binding ) 使用 **[ngModel]**

```
<input name="username" [ngModel]="username">
```

- 雙向綁定 ( two-way binding ) 使用 **[(ngModel)]**

```
<input name="username" [(ngModel)]="username">
```

# 如何建立並取得表單控制項實體

- 不綁定直接建立實體 **ngModel**

```
<input name="username" ngModel  
      #mUsername="ngModel">
```

- 單向綁定( one-way binding ) 使用 **[ngModel]**

```
<input name="username" [ngModel]="username"  
      #mUsername="ngModel">
```

- 雙向綁定( two-way binding ) 使用 **[(ngModel)]**

```
<input name="username" [(ngModel)]="username"  
      #mUsername="ngModel">
```

# 如何給定表單欄位預設值

- 直接跟元件中的屬性 (Model) 進行資料綁定即可
  - 單向綁定( one-way binding ) 使用 **[ngModel]**

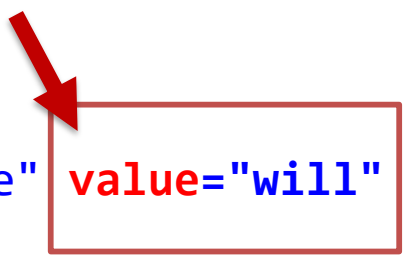
```
<input name="username" [ngModel]="username">
```

- 雙向綁定( two-way binding ) 使用 **[(ngModel)]**

```
<input name="username" [(ngModel)]="username">
```

- **錯誤示範**

```
<input name="username" value="will" [ngModel]="username">
```



# 表單欄位名稱的必要性

- 獨立存在的輸入欄位 ( 沒有 `<form>` 包覆的欄位 )

```
<input type="text" class="form-control rounded"  
      [(ngModel)]="username" #f1="ngModel">
```

- 放在 `<form>` 裡面的輸入欄位則必須要有 `name` 屬性

```
<input type="text" class="form-control rounded"  
      ngModel #f1="ngModel" name="username">
```

```
<input type="text" class="form-control rounded"  
      ngModel #f1="ngModel"  
      [ngModelOptions]="{standalone: true}">
```

# 取得 NgModel 實體的常見屬性

屬性名稱	型別	用途說明
name	string	欄位名稱
value	any	欄位值
valid	boolean	有效欄位 (通過欄位驗證)
invalid	boolean	無效欄位 (欄位驗證失敗)
errors	{[key: string]: any}	當出現無效欄位時，會出現的錯誤狀態
dirty	boolean	欄位值是否曾經更動過一次以上
pristine	boolean	欄位值是否為原始值 (未曾被修改過)
touched	boolean	欄位曾經經歷過 focus 事件
untouched	boolean	欄位從未經歷過 focus 事件
disabled	boolean	欄位設定為 disabled 狀態
enabled	boolean	欄位設定為 enabled 狀態 (預設啟用)
formDirective	NgForm	取得目前欄位所屬的 NgForm 表單物件
valueChanges	EventEmitter	可用來訂閱欄位 <b>值變更</b> 的事件
statusChanges	EventEmitter	可用來訂閱欄位 <b>狀態變更</b> 的事件




# 認識 NgForm 指令 (Directive)

- 重要觀念
  - [NgForm](#) 主要用來建立一個 最上層 的 表單群組 ([FormGroup](#)) 實體
  - 表單群組 ([FormGroup](#)) 的主要用途
    - 用來追蹤群組內所有表單控制項的欄位值與驗證狀態
    - 也可以從表單群組物件實體取得所有表單控制項的欄位值與驗證狀態
  - FormsModule 預設會將所有的 `<form>` 標籤宣告為 NgForm 指令
    - 因此 `<form>` 標籤不需要額外宣告 `ngForm` 指令 (Directive)
    - 要從範本取得 `ngForm` 物件實體，可以透過範本參考變數完成  
`<form name="form1" #f="ngForm">`
    - 比較 `ngModel` 的用法 ( 建立表單控制項一定要用 `ngModel` 宣告過 )  
`<input name="txt" ngModel #mTxt="ngModel">`

# 使用 NgForm 的 ngSubmit 事件


- 搭配 **type="submit"** 按鈕與 (**ngSubmit**) 事件 (建議用法)

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">  
  <input name="account" ngModel required>  
  <button type="submit">Submit</button>  
</form>
```



- 搭配 **type="button"** 按鈕與 (**click**) 事件 (不會觸發 ngSubmit 事件)

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">  
  <input name="account" ngModel>  
  <button type="button" (click)="onSubmit(f)" >  
    Submit  
  </button>  
</form>
```



表單事件範例

# 取得 NgForm 實體的常見屬性

屬性名稱	型別	用途說明
value	any	表單內所有欄位值 (以物件型態呈現)
valid	boolean	所有欄位是否皆為有效欄位
invalid	boolean	是否有任意一個欄位為無效欄位
<b>errors</b>	<b>請注意：在 NgForm 物件下有此屬性，但無資料！</b>	
dirty	boolean	任意欄位是否曾經更動過一次以上
pristine	boolean	任意欄位欄位是否為原始值 (未曾被修改過)
touched	boolean	任意欄位曾經經歷過 focus 事件
untouched	boolean	任意欄位從未經歷過 focus 事件
disabled	boolean	所有欄位設定為 disabled 狀態
enabled	boolean	任意欄位為 enabled 狀態就為 true
valueChanges	EventEmitter	可用來訂閱任意欄位 <b>值變更</b> 的事件
statusChanges	EventEmitter	可用來訂閱任意欄位 <b>狀態變更</b> 的事件
<b>submitted</b>	<b>boolean</b>	<b>判斷該表單是否經歷過 ngSubmit 事件</b>

# 取得表單所有欄位值

- 範例表單

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">  
  <input name="account" ngModel #mAccount>  
  <button type="submit">Submit</button>  
</form>
```

- 取得表單欄位值的注意事項

- 預設透過 `f.value` 即可取得表單所有欄位值 (物件型態)
- 如果表單中有**部分欄位**被設定為 `disabled` 狀態
  - 在 `f.value` 物件將會找不到 `disabled` 欄位的值
- 如果表單中**所有欄位**被設定為 `disabled` 狀態
  - 在 `f.value` 物件將會顯示所有欄位的值 (包含 `disabled` 的欄位)
  - 但此時的 `f.valid` 屬性將會改為 `false`

# 認識 NgModelGroup 指令 (Directive)

- 主要用途
  - 在現有表單 (NgForm) 建立額外的欄位群組
  - 只能使用在有 **<form>** 標籤的表單範圍內宣告
- 重要觀念
  - [NgModelGroup](#) 主要用來建立一個 表單群組 ([FormGroup](#)) 實體
  - 表單群組 ([FormGroup](#)) 的主要用途
    - 用來追蹤群組內所有表單控制項的欄位值與驗證狀態
    - 用來取得群組內所有表單控制項的欄位值與驗證狀態
- 使用範例

```
<div ngModelGroup="group1" #group1="ngModelGroup">  
  <input name="first" [ngModel]="model.first">  
  <input name="last" [ngModel]="model.last">  
</div>
```

# 取得 NgModelGroup 實體的常見屬性

屬性名稱	型別	用途說明
value	any	群組內所有欄位值 (以物件型態呈現)
valid	boolean	群組內所有欄位是否皆為有效欄位
invalid	boolean	群組內是否有任意一個欄位為無效欄位
errors	請注意：在 NgModelGroup 物件下有此屬性，但無資料！	
dirty	boolean	任意欄位是否曾經更動過一次以上
pristine	boolean	任意欄位欄位是否為原始值 (未曾被修改過)
touched	boolean	任意欄位曾經經歷過 focus 事件
untouched	boolean	任意欄位從未經歷過 focus 事件
disabled	boolean	所有欄位設定為 disabled 狀態
enabled	boolean	任意欄位為 enabled 狀態就為 true
valueChanges	EventEmitter	可用來訂閱任意欄位 <b>值變更</b> 的事件
statusChanges	EventEmitter	可用來訂閱任意欄位 <b>狀態變更</b> 的事件

# 表單驗證狀態的 CSS 樣式處理

- **表單驗證機制**會自動根據**驗證狀態**提供 CSS 樣式類別
- 會根據**驗證狀態**提供 CSS 樣式類別的 Directives 有：
  - NgModel
  - NgModelGroup
  - NgForm

## 範例程式

欄位狀態	true	false
欄位是否曾經被碰過 ( f.touched / f.untouched )	ng-touched	ng-untouched
欄位值是否有改變過 ( f.dirty / f.pristine )	ng-dirty	ng-pristine
是否有通過欄位驗證 ( f.valid / f.invalid )	ng-valid	ng-invalid

# 在範本中取得驗證結果的方法

- 使用方式
  - 直接在輸入欄位上套用驗證屬性即可  
`<input name="title" [ngModel]="title" required>`
- 如果欄位驗證成功
  - NgModel 物件的 `errors` 屬性會回傳 `null`
- 如果欄位驗證失敗
  - NgModel 物件的 `errors` 屬性會有所有驗證狀態 (如下圖示)
  - 取得必填的錯誤狀態：`f1.errors?.required`

## Rounded Corners

```
<input type="text" name="title" [ngModel]="title" #f1="ngModel">
```



```
{  
  "required": true  
}
```

```
{{ f1.errors | json }}
```





# 內建的驗證器 (Validator)

- FormsModule 預設提供的 Validator 有：

驗證屬性	用途說明
required	表單控制項為 <b>必填欄位</b>
minlength	表單控制項的值不可少於幾個字元
maxlength	表單控制項的值不可大於幾個字元
pattern	表單控制項的值必須符合給予的 Regex 正規表示式

```
<form novalidate>
  <input name="title"      ngModel required>
  <input name="summary"    ngModel minlength="20">
  <input name="longdesc"   ngModel maxlength="300">
  <input name="mobile"     ngModel pattern="\d{4}-\d{6}">
</form>
```



Model-Driven Forms (Reactive Forms)

# 模型驅動表單開發模式

# 認識 Reactive Form 表單開發模式

- 模型驅動表單 (Model-driven Forms) 又稱 Reactive Forms
  - 在同一個 Angular 應用程式中可以混合使用兩種模型
- 完全由 **表單模型** (Model) 集中管理/設定所有表單欄位
  - 從元件內部進行**表單模型**與**資料模型**的對應與設計
  - 給予欄位**預設值**與**驗證規則**
- 範本還是需要跟元件模型進行綁定
  - AppModule 需匯入 **ReactiveFormsModule** 模組才有以下 Directives
    - FormControlDirective      [formControl]="ctrl1"
    - FormGroupDirective      [formGroup]="form"
    - FormControlName      formControlName="title"
    - FormGroupName      formGroupName="group1"
    - FormArrayName      formArrayName="array1"

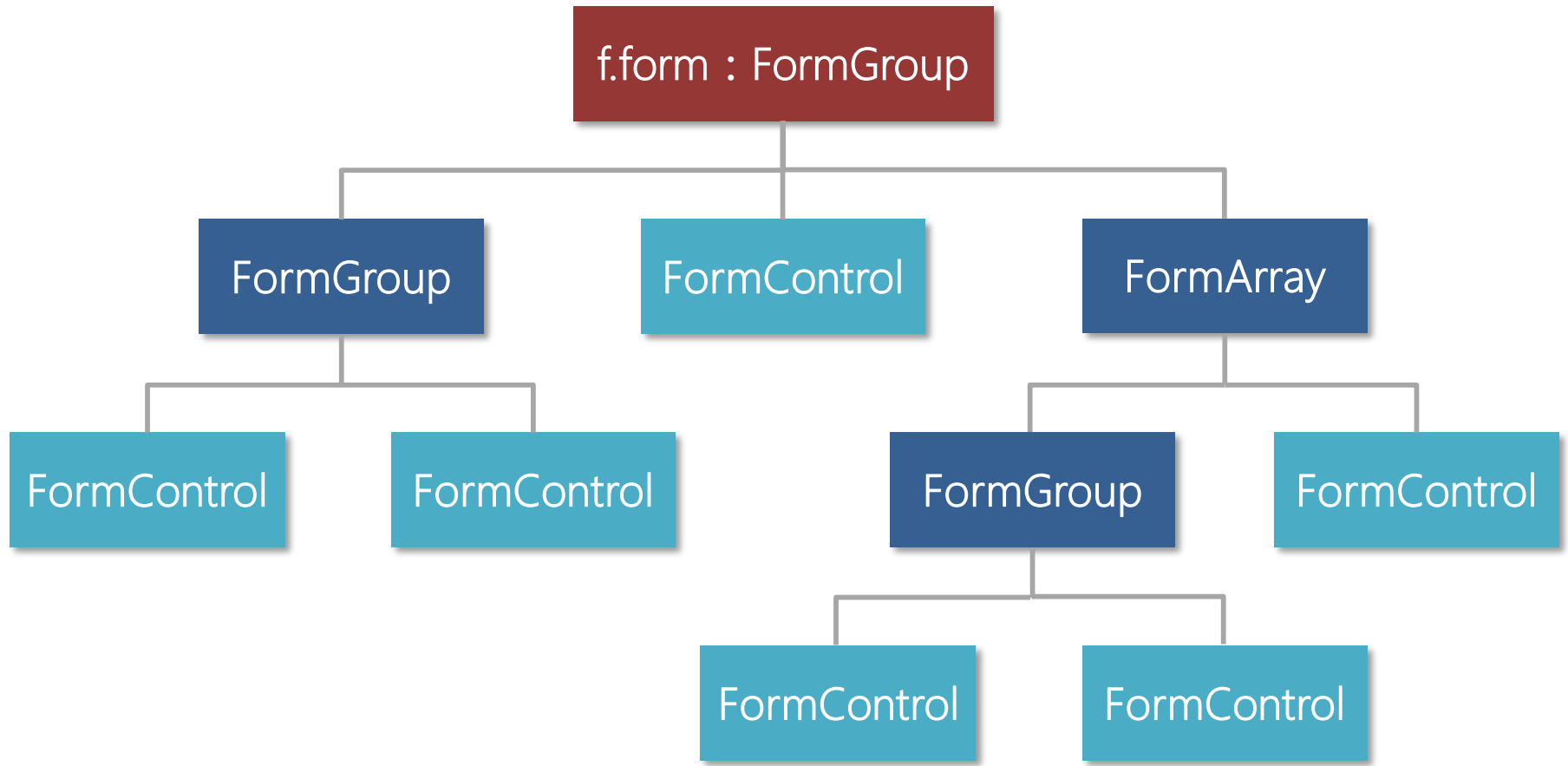
# 匯入 ReactiveFormsModule 模組

```
app.module.ts x
1  import { BrowserModule } from '@angular/platform-browser';
2  import { ReactiveFormsModule, FormsModule } from '@angular/forms';
3  import { NgModule } from '@angular/core';
4  import { HttpClientModule } from '@angular/http';
5
6  import { AppComponent } from './app.component';
7
8
9  @NgModule({
10    declarations: [
11      AppComponent,
12    ],
13    imports: [
14      BrowserModule,
15      FormsModule,
16      ReactiveFormsModule,
17      HttpClientModule
18    ],
19    providers: [],
20    bootstrap: [AppComponent]
21  })
22  export class AppModule { }
23
```

# 認識幾個 Reactive Forms 重要類別

- [AbstractControl](#)
  - 這是一個抽象類別。它是 Reactive Forms 架構中 3 個表單類別共同的抽象基底類別 (abstract base class)，這 3 個表單控制項就是 FormControl, FormGroup 與 FormArray 類別。這裡定義所有表單控制項共用的屬性與方法，有一部分屬性為 Observable 型別
- [FormControl](#)
  - 用來追蹤某一個表單控制項的欄位值與驗證狀態
- [FormGroup](#)
  - 用來追蹤一群表單控制項的欄位值與驗證狀態
  - 該群組下的欄位值包含所有子控制項與子群組的所有集合
  - 該群組會以「物件」的方式進行群組（欄位之間沒有順序性，索引值為字串）
- [FormArray](#)
  - 用來追蹤一群表單控制項的欄位值與驗證狀態
  - 該群組下的欄位值包含所有子控制項與子群組的所有集合
  - 該群組會以「陣列」的方式進行群組（欄位之間的索引值為數值）

# 模型驅動表單物件架構 (表單模型)



# 使用 FormBuilder 建立表單模型

- 宣告表單模型與注入 FormBuilder 服務元件

```
form: FormGroup;
```

```
constructor(private fb: FormBuilder) {  
}
```

- 建立表單模型物件 ( 頂層表單群組 + 子表單控制項 )

```
ngOnInit() {  
  this.form = this.fb.group({  
    title: 'This is title'  
  });  
}
```

表單控制項  
(FormControl)

欄位預設值

# 設計範本的表單介面

- 重要觀念
  - 通常一個表單都擁有多個欄位
  - 多個欄位意味著需要一個群組
  - 所以表單最上層就是一個 FormGroup 表單群組
  - 模型驅動表單的欄位不需要 name 屬性 (範本驅動表單則為必要條件)
- 表單介面最上層必須套用 **[formGroup]** 繫結表單群組物件

```
<form [formGroup]="form" (ngSubmit)="submit()">
```

```
  <input type="text" formControlName="title">
```

```
</form>
```



# 比較兩種不同的表單模型建立方法

- 使用 FormBuilder 建立表單元件

```
this.form = this.fb.group({  
  name: this.fb.group({  
    first: 'Will',  
    last: 'Huang'  
  })  
});
```

- 使用 FormControl, FormGroup, FormArray 建立表單元件

```
this.form = new FormGroup({  
  name: new FormGroup({  
    first: new FormControl('Will'),  
    last: new FormControl('Huang')  
  })  
});
```

# 設計多層表單群組的表單 (模型部分)

- 模型物件

```
this.form = this.fb.group({  
  
  title: 'This is title',  
  
  name: this.fb.group({  
    firstName: 'Will',  
    lastName: 'Huang'  
  })  
  
});
```

# 設計多層表單群組的表單 (範本部分)

- 範本介面

```
<form [formGroup]="form">  
  <input type="text" formControlName="title">  
  <div formGroupName="name">  
    <input type="text" formControlName="firstName">  
    <input type="text" formControlName="lastName">  
  </div>  
</form>
```

```
<pre>{{ form.value | json }}</pre>
```

# 設計多層表單陣列的表單 (模型部分)

- 模型物件

```
this.form = this.fb.group({  
  title: 'This is title',  
  name: this.fb.array([  
    this.fb.control('Will 1', Validators.required),  
    this.fb.control('Will 2', Validators.required)  
  ])  
});
```

- 注意事項

- 傳入 `this.fb.array` 的是一個陣列
- 陣列內的 `formControlName` 將會是一個陣列索引值 (數值)
- 陣列內每個元素可以是任意表單物件  
( `FormControl`, `FormGroup`, `FormArray` )

# 設計多層表單陣列的表單 (範本部分)

- 範本介面
  - 以下範例可以看出，**表單陣列群組**非常適合用來建置**動態表單**

```
<form [formGroup]="form">
  <input type="text" formControlName="title">
  <div formArrayName="name">
    <input type="text"
      *ngFor="let item of form.controls.name.controls;
        let i=index"
      [formControlName]="i">
  </div>
</form>

<pre>{{ form.value | json }}</pre>
```

# 套用欄位驗證

- 匯入驗證器 (Validators)

```
import { FormBuilder, FormGroup, Validators } from  
'@angular/forms';
```

- 套用預設值與驗證器

```
this.form = this.fb.group({  
  title: ['The Will Will Web', [  
    Validators.required,  
    Validators.minLength(3)  
  ]  
});
```

- 範本介面

```
<input type="text" formControlName="title">
```

# 表單物件的常見屬性

- [FormControl](#) 物件
  - 屬性繼承自 `AbstractControl` 類別
  - 屬性與 `NgModel` 完全一樣
  - 有許多自訂的方法 (詳見 [FormControl API](#) 文件)
- [FormGroup](#) 物件
  - 屬性繼承自 `AbstractControl` 類別
  - 屬性與 `NgModelGroup` 完全一樣
  - 有許多自訂的方法可動態增減子控制項 (詳見 [FormGroup API](#) 文件)
- [FormArray](#)
  - 屬性繼承自 `AbstractControl` 類別
  - 屬性與 `NgModelGroup` 完全一樣
  - 有許多自訂的方法可動態增減子控制項 (詳見 [FormArray API](#) 文件)

# 資料模型 v.s. 表單模型

- 名詞定義
  - 資料模型
    - 通常代表著從伺服器回傳的資料模型物件
  - 表單模型
    - 由 FormControl, FormGroup, FormArray 組成的表單模型物件
    - 可透過 FormBuilder 來建立，也可以自行 new 出新物件
- 重要觀念
  - 通常我們會將**資料模型**複製到**表單模型**中
    - 這意味著開發人員**必須了解資料模型如何對應到表單模型**中
    - 在表單上所進行的**資料異動**，只會改到**表單模型**，不會動到**資料模型**
  - **表單模型與資料模型**之間的關係**不需要 1 對 1 對應**
    - 通常我們只需要將資料模型的部分資料對應到表單上
    - 若資料模型與表單模型接近，兩者之間的關係會更加清楚 (可維護性)



# setValue v.s. patchValue v.s. reset

- 將資料模型寫入到表單模型中 (會覆寫完整物件)
  - 傳入 `setValue()` 的物件，必須完全與 `FormGroup` 中的物件結構相同！

```
this.form.setValue({  
  name:    this.hero.name,  
  address: this.hero.addresses[0] || new Address()  
});
```

- 將資料模型部分更新到表單模型中 (僅更新部分屬性)
  - 傳入 `patchValue()` 的物件，如果比對不到屬性，不會出現錯誤(會自動忽略)！

```
this.form.patchValue({  
  name: this.hero.name  
});
```

# 重置表單內容 ( reset )

- 重置表單的 reset() 方法
  - 表單控制項或表單群組的狀態會被重置為 `pristine`
  - 表單控制項或表單群組的狀態會被重置為 `untouched`
- 清空所有內容 ( 欄位值都會變成 `null` 值 )

```
this.form.reset();
```

- 重置為預設內容

```
this.form.reset({  
  title: 'The Will Will Web'  
});
```

# FormArray 與 FormGroup 開發技巧

- 善用 [map\(\)](#) 函式，將資料模型轉換成表單陣列物件

```
setAddresses(addresses: Address[]) {  
  
    // 先將資料模型轉換為表單群組（每筆資料都轉換成一個群組）  
    const addressFGs =  
        addresses.map(address => this.fb.group(address));  
    // 再將表單群組加入到表單陣列中  
    const addressFormArray = this.fb.array(addressFGs);  
    // 最後將表單陣列取代 form 下的 'secretLairs' 表單控制項  
    this.form.setControl('secretLairs', addressFormArray);  
  
}
```

# 取得表單群組或表單陣列中的控制項

- 取得子表單控制項

```
return this.form.get('email') as FormControl;
```

- 取得子表單陣列

```
return this.form.get('group1') as FormArray;
```

- 取得子表單群組

```
return this.form.get('array1') as FormGroup;
```

- 取得多層群組下的表單控制項

```
return this.form.get('group1.email') as FormControl;
```

# 自訂錯誤驗證器

- 驗證器範例程式

```
export function forbiddenNameValidator(control: AbstractControl) {  
    const nameRe = /Will/;  
    const name = control.value;  
    const no = nameRe.test(name);  
    return no ? { 'forbiddenName': true } : null;  
};
```

- 驗證器回傳型別

- 驗證通過要回傳 `null`，驗證不通過要回傳一個驗證錯誤物件
- 驗證錯誤物件 (validation error object)
  - 通常都包含一個屬性
  - 屬性名稱會是驗證控制器的名稱
  - 屬性的值則是任意值
    - 通常為布林值/也可設成當下的值或錯誤訊息

# 自訂錯誤驗證器 (帶參數的驗證器)

- 驗證器範例程式

```
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {  
  return (control: AbstractControl): { [key: string]: any } => {  
    const name = control.value;  
    const no = nameRe.test(name);  
    return no ? { 'forbiddenName': true } : null;  
  };  
}
```

[Plunker 範例](#)

- 驗證器回傳型別

- 驗證通過要回傳 `null`，驗證不通過要回傳一個驗證錯誤物件
- 驗證錯誤物件 (validation error object)
  - 通常都包含一個屬性
  - 屬性名稱會是驗證控制器的名稱
  - 屬性的值則是任意值
    - 通常為布林值/也可設成當下的值或錯誤訊息

# 自訂錯誤驗證器 (帶參數的驗證器)

- 驗證器範例程式

```
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {  
  return function(control: AbstractControl) {  
    const name = control.value;  
    const no = nameRe.test(name);  
    return no ? { 'forbiddenName': true } : null;  
  };  
}
```

[Plunker 範例](#)

- 驗證器回傳型別

- 驗證通過要回傳 `null`，驗證不通過要回傳一個驗證錯誤物件
- 驗證錯誤物件 (validation error object)
  - 通常都包含一個屬性
  - 屬性名稱會是驗證控制器的名稱
  - 屬性的值則是任意值
    - 通常為布林值/也可設成當下的值或錯誤訊息

# 如何選擇兩種截然不同的表單模型

- 範本驅動表單開發模式 (Template-Driven Forms)
  - 採用**宣告**的方式建立表單 (**較為簡單**) (維護較為容易)
  - 適合**固定欄位數量**的表單
    - 會員註冊、登入、線上下單、修改會員資料、...
  - **無法**對表單進行**單元測試** ( Unit Testing )
    - 因為所有表單邏輯都定義在範本中，無法對元件進行單元測試
    - 只能透過 E2E 的方式測試表單
- 模型驅動表單開發模式 (Model-Driven Forms)
  - 採用**編程**的方式建立表單 (**較為繁瑣**) (程式碼維護較為麻煩)
  - 適合**動態欄位數量**的表單 (動態表單)
    - 由後台定義的動態問卷系統、變動選項的投票系統、...
  - 可以直接針對表單進行**單元測試** ( Unit Testing )
    - 因為所有表單邏輯都已宣告為物件類型的表單模型



# 相關連結

- [Template-driven Forms - ts - GUIDE](#)
- [Reactive Forms - ts - GUIDE](#)
- [Form Validation - ts - COOKBOOK](#)
- [Dynamic Forms - ts - COOKBOOK](#)
- [Angular FormsModule API documents](#)
- [Model-Driven Forms with FormGroup and FormControl](#)
- [Introduction to Angular 2 Forms](#)



# Angular 4 開發實戰：進階開發篇

深入了解 Component 元件架構



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/>



Understanding Advanced Angular Directives

# 深入了解 Angular 指令 (Directives)

# Angular 的元件種類

- 元件型指令

- 就是一個含有樣板的指令
- 使用 `@Component()` 裝飾器 ( ng g component name )

- 屬性型指令

- 就是一個沒有樣板的指令
- 使用 `@Directive()` 裝飾器 ( ng g directive name )
- 套用在其他範本中時通常會取得 DOM 來修改該元素的外觀或行為

- 管線型元件

- 只能在樣板中使用此類元件
- 使用 `@Pipe()` 裝飾器 ( ng g pipe name )

- 服務型元件

- 只能在元件中使用此類元件，任意元件皆可透過 DI 注入服務型元件
- 使用 `@Injectable()` 裝飾器 ( ng g service name )

# 元件型指令 - 進階主題 (1)

- 從目前元件中找出 View 裡面的特定元件

```
<div #containerFluid></div>
```

```
@ViewChild('containerFluid') div: ElementRef;
```

```
<app-my></app-my>
```

```
@ViewChild(MyComponent) mycmp: MyComponent;
```

```
@ViewChildren(MyComponent) cmps: QueryList<MyComponent>;
```

```
ngAfterViewInit() {    // 使用時一定要寫在這裡！
```

```
    this.cmps.forEach(x => {
```

```
        console.log(x);
```

```
    });
```

```
}
```

[範例程式](#)

# 元件型指令 - 進階主題 (2)

- 從元件中找出內容元件中的特定元件

```
<app-main> <h2 #title>Hello World</h2> </app-main>  
@ContentChild('title') title: ElementRef;
```

```
<app-main> <app-my></app-my> </app-main>  
@ContentChild(MyComponent) mycmp: MyComponent;
```

```
@ContentChildren(MyComponent) cmps: QueryList<MyComponent>;  
ngAfterContentInit() { // 使用時一定要寫在這裡!  
  this.cmps.forEach(x => {  
    console.log(x);  
  });  
}
```

[範例程式](#)

# 元件型指令 - 進階主題 (3)

- 如何將**內容元件**的執行結果顯示在目前元件範本中

```
@Component({  
  selector: 'app-main',  
  template: `  
    <h2 #title>Hello World</h2>  
    <ng-content></ng-content>  
  `,  
});  
export class MainComponent { ... }
```

# 屬性型指令 - 進階主題

- 從元件中找出 Template 中的 DOM 物件與事件

```
@HostBinding('style.color') myColor: string = 'black';
@HostListener('click', ['$event']) myClick($event) {
    this.myColor = 'blue'; }
};
@HostListener('document:dblclick', ['$event.target'])
docDbClick(target) { console.dir(target); };
```

- 從元件中找出目前屬性型指令套用的那個元素物件(DOM)

```
constructor(private el: ElementRef, // 少用為妙
             private renderer: Renderer) { }

console.dir(this.el.nativeElement);
```



ViewEncapsulation

# 元件內的樣式封裝方法



# View Encapsulation Types

- **ViewEncapsulation.None**
  - 完全不使用 Shadow DOM 技術
  - 所有從元件註冊的樣式都會加入到整份網頁
- **ViewEncapsulation.Emulated (預設值)**
  - 不使用 Shadow DOM 技術
  - 元件註冊的樣式只會套用到目前元件中的範本 HTML 中
- **ViewEncapsulation.Native**
  - 使用原生的 Shadow DOM 技術
  - 元件會以 Shadow DOM 技術將元素網頁封裝 Web 元件

# ViewEncapsulation 使用範例

```
import { Component, ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <div class="hello">Hello World</div>
  `,
  encapsulation: ViewEncapsulation.None,
  styles: [`
    .hello { text-decoration: underline; }
  `]
})
export class AppComponent {
  name = 'World';
}
```

[範例程式](#)

Lifecycle Hooks

# 完整的元件生命週期



# 元件指令中的完整事件清單

- constructor
- ngOnChanges
- ngOnInit
- ngDoCheck
  - ngAfterContentInit
  - ngAfterContentChecked
  - ngAfterViewInit
  - ngAfterViewChecked
- ngOnDestroy

constructor

ngOnChanges

ngOnInit

ngDoCheck

ngAfterContentInit

ngAfterContentChecked

ngAfterViewInit

ngAfterViewChecked

ngOnDestroy

# 主要的元件生命週期

- ngOnChanges
  - 第一次發生 @Input() 繫結時就會觸發 (比 ngOnInit 早)
  - 之後每發生 @Input() 繫結時就會觸發一次
- ngOnInit
  - 會在第一次執行 ngOnChanges() 事件後觸發
- ngDoCheck
  - 每次執行變更偵測時，都會自動執行 ngDoCheck() 事件
- ngOnDestroy
  - 在元件**摧毀之前**會執行此事件

# 子元件與內容元件的事件

- 子元件事件
  - ngAfterViewInit
    - 當 View 裡面所有元件都**初始化完成後**觸發此事件
  - ngAfterViewChecked
    - 當 View 裡面所有元件都**完成變更偵測機制後**觸發此事件
- 內容元件事件
  - ngAfterContentInit
    - 當 Content 裡面所有的元件都初始化完成後觸發此事件
  - ngAfterContentChecked
    - 當 Content 裡面所有元件都**完成變更偵測機制後**觸發此事件

# 相關連結

- [Component Interaction - ts - COOKBOOK](#)
- [View Encapsulation in Angular by thoughttram](#)
- [Lifecycle Hooks - ts - GUIDE](#)
- [Dynamic Component Loader - ts - COOKBOOK](#)





# Angular 4 開發實戰：進階開發篇

深入了解 Angular 變更偵測機制 (效能調校)



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/>

JavaScript Mutability

# 認識 JavaScript 的可變性



# 關於 JavaScript 的可變性

- 在 JavaScript 中，只有兩種型別
  - 原始型別 (Primitive Type) → 不具可變性
  - 物件型別 (Object Type) → 具有可變性
- 所有物件型別的物件，全部都是 **可變的** (Mutable)

```
var a = { name: 'Will', age: 25 };
```

```
var b = [];
```

```
b.push(a);
```

```
a.age++;
```

```
b.push(a);
```

請問最後 `b[0].age` 的值為何？

請問最後 `b[1].age` 的值為何？

# 關於 JavaScript 的常數特性

- 使用 const 宣告變數並指派一個數值

```
const a = 25;
```

```
a = 50;
```

```
console.log(a);
```

請問執行結果為何？

- 使用 const 宣告變數並指派一個物件

```
const a = { name: 'Will', age: 25 };
```

```
a.name = 'Jeff';
```

```
console.log(a.name);
```

請問執行結果為何？

# 如何偵測特定物件在執行過程被變更

```
function test(p) {  
    p.name = 'John';  
    return p;  
}
```

```
var a = { name: 'Will', age: 25 };
```

```
test(a);
```

// 如何判斷 a 變數在執行 test(a) 之後是否有被修改？

Writing Immutable Objects

# 撰寫不可變的物件



# 認識 Immutable (不可變的) 物件

- 重要觀念
  - JavaScript 並沒有真正的 Immutable 物件
  - 但可以透過不同的程式開發技巧達成不可變的目的
- 使用 [Object.assign](#) 模擬一個 Immutable 物件

```
var a = { name: 'Will', age: 25 };  
var b = [];  
b.push(Object.assign({}, a));  
a.age++;  
b.push(Object.assign({}, a));
```

# 更多不可變物件範例

- 上頁範例的變數 `b` 在執行完 `push` 後也變更了！
- 使用展開運算子模擬一個 Immutable 陣列

```
var a = { name: 'Will', age: 25 };  
var b = [];  
b = [...b, Object.assign({}, a)];  
a = Object.assign({}, { age: a.age+1 });  
b = [...b, Object.assign({}, a)];
```

**請注意：** 展開運算子只能用於 Iterable 的物件！



# 使用第三方函式庫來實現不可變物件

- 使用第三方函式庫來實現 Immutable 物件
  - 例如 Facebook 知名的 [Immutable.js](#) 函式庫
- Immutable.js 提供多種的 Immutable 資料結構
  - [List](#)
  - [Stack](#)
  - [Map](#), [OrderedMap](#)
  - [Set](#), [OrderedSet](#)
  - [Record](#)

# 使用 Immutable.js 的 Map 函式

- 使用 Immutable.js 的 Map 來實現 Immutable

```
var map1 = Immutable.Map({ a: 1, b: 2 });  
var map2 = map1.set('a', 2);  
map1.get('a'); // 1  
map2.get('a'); // 2
```

- 實驗方法

1. 開啟 <https://facebook.github.io/immutable-js/> 網頁
2. 開啟 F12 開發者工具，切換到 Console 頁籤
3. 執行上述程式碼



Change Detection Strategy

# 認識 Angular 變更偵測策略

# 關於 Angular 變更偵測

- 關於應用程式狀態
  - Angular 應用程式是由**元件**組成的
  - 每一個**元件**裡儲存著許多**內部狀態**
  - **內部狀態**可以是任何類型的物件 (原始/物件型別)
  - **元件的內部狀態**經常與網頁的 使用者介面 (**View**) 相關
- 變更偵測的基本任務
  - 就是獲取**應用程式內部狀態**
  - 判斷出應用程式狀態是否發生變更
  - 將新的**應用程式內部狀態**反應到 **View** 上
  - 在**瀏覽器**中就是將**應用程式狀態**反映到 **DOM** 物件上

# 觸發變更偵測的情境

- 常見的**狀態變更**原因
  - DOM 事件                      - click, change, input, submit, ...
  - XMLHttpRequest - 從遠端伺服器抓取資料
  - Timers                         - setTimeout(), setInterval()
- 這些都是 **非同步操作** (async)
- 每當**非同步操作**執行完成，**應用程式狀態就可能發生改變**，這時就需要通知 **View** 更新 (**DOM**)

# 關於 Zones 機制

- Angular 2 開始加入了非常神奇的 [Zones](#) 機制
- 主要用途
  - 管理**非同步事件**的**執行與錯誤追蹤**
  - 可以精準掌握非同步事件的運作時機
  - 可以更清楚地記錄錯誤發生時的呼叫堆疊
- 服務元件
  - 預設 Angular 就是以 [NgZone](#) 進行非同步事件管理
  - 應用程式生命週期中所有非同步事件都由 [NgZone](#) 管理
  - 你可以在元件中注入 [NgZone](#) 服務元件管理事件執行

# 關於 Zones 的運作方式

- NG2+ 會透過 [NgZone](#) 覆寫幾個瀏覽器底層的 APIs
- 以 addEventListener 為例會覆寫成類似以下程式

```
1 // this is the new version of addEventListener
2 function addEventListener (eventName, callback) {
3     // call the real addEventListener
4     callRealAddEventListener(eventName, function() {
5         // first call the original callback
6         callback(...);
7         // and then run Angular-specific functionality
8         var changed = angular2.runChangeDetection();
9         if (changed) {
10             angular2.reRenderUIPart();
11         }
12     });
13 }
```

# 示範 NgZone 的威力

- 範本

```
<div> {{ debug() }} </div>
```

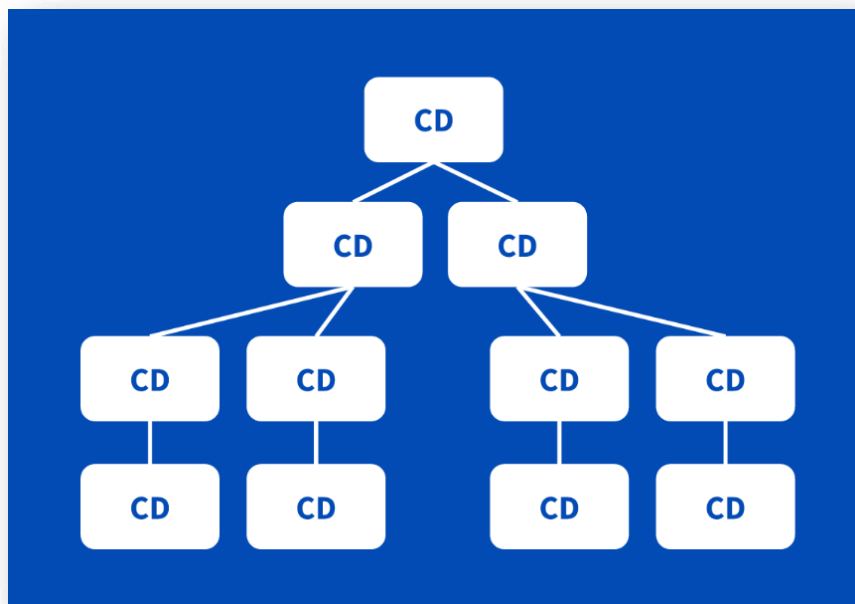
- 元件

```
ngOnInit() {  
  document.addEventListener('click',  
    x => console.log(x));  
}  
debug() {  
  console.log('DEBUG:'+(new Date().getTime()));  
}
```



# 變更偵測樹 (Change Detection Tree)

- 重要觀念
  - 每個 Angular 元件樹中，從根元件開始每一個元件都會擁有一個自己的**變更偵測器** (change detector)
  - 每一個變更偵測器會組成一個變更偵測樹
  - 變更偵測的觸發流程總是從上到下依序觸發 ([有向圖](#))






# 關於 Angular 的變更偵測策略

- 每個元件都可以定義自己的變更偵測策略
  - **ChangeDetectionStrategy.Default**
    - 變更偵測會在每一次非同步事件發生時執行
  - **ChangeDetectionStrategy.OnPush**
    - 變更偵測會在當元件中有 `@Input()` 屬性資料變更時執行
- 每次變更偵測作業都會從根元件開始執行

```
@Component({  
  selector: 'app-flot',  
  templateUrl: './flot.component.html',  
  styleUrls: ['./flot.component.css'],  
  changeDetection: ChangeDetectionStrategy.  
})
```

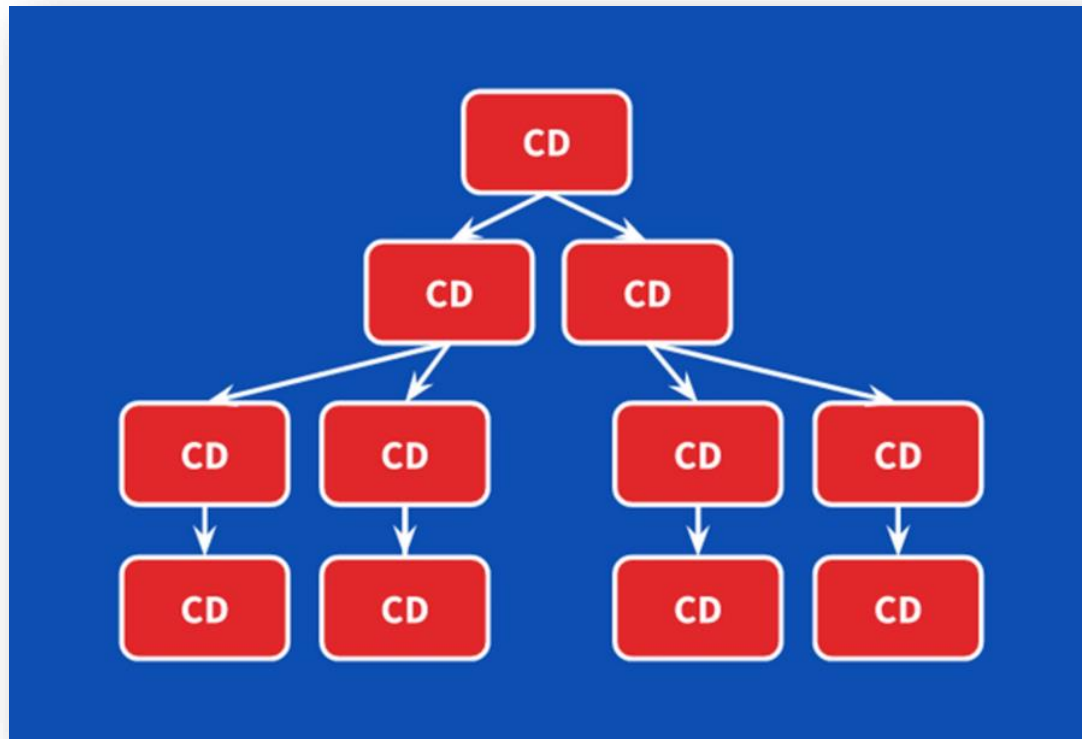
```
export class FlotComponent {
```

```
  username: string;
```

 Default (enum member) ChangeDetectionStrategy.Default...  
`Default` means that the change detector's mode will be set t...   
 OnPush

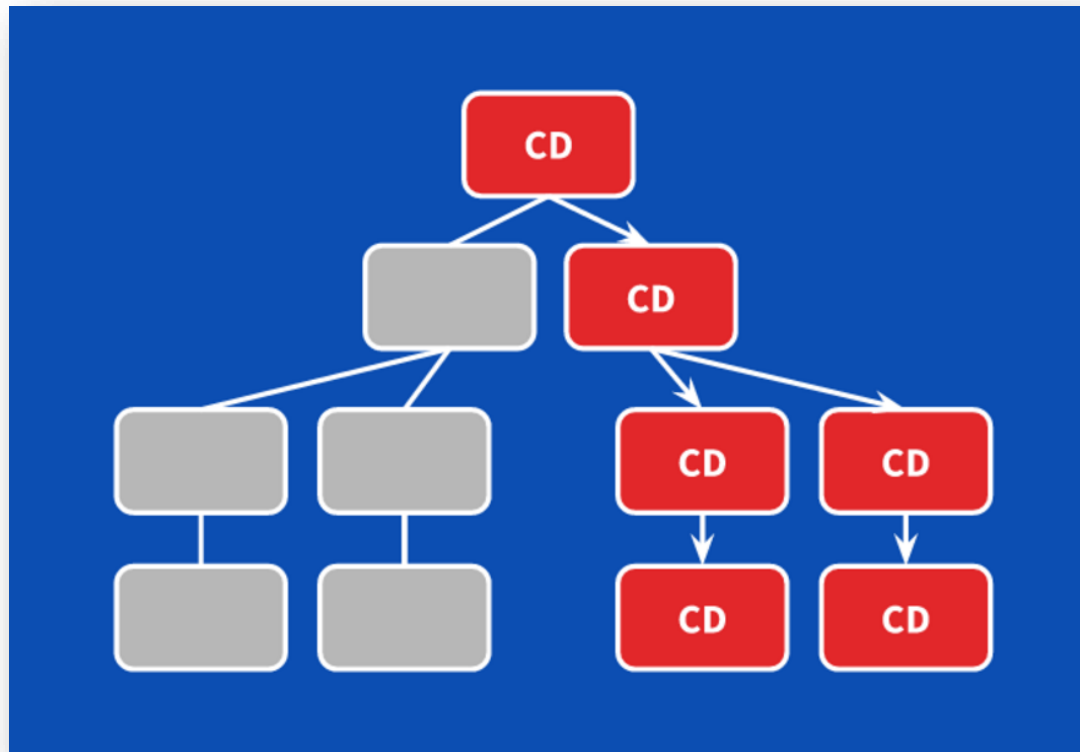
# ChangeDetectionStrategy.Default

- 從任意一個元件觸發了一個非同步事件 (ex. click) , Angular 會從**根元件**開始往下執行所有的**變更偵測**



# ChangeDetectionStrategy.OnPush

- 如果元件中套用 `@Input()` 的屬性沒有任何改變，Angular 會跳過變更偵測 (含所有**子元件**都會跳過)



# 不同變更偵測策略的效率比較

- **ChangeDetectionStrategy.Default**
  - 變更偵測會檢查範本表達式的值是否已更改 {{ ... }}
  - 變更偵測所費時間 =  $C * N$ 
    - C - 檢查一個 binding 的時間
    - N - 總共的 bindings 數量
- **ChangeDetectionStrategy.OnPush**
  - 變更偵測只會檢查 @Input() 屬性的值是否已更改
  - 變更偵測所費時間  $\simeq C * M$ 
    - C - 檢查一個 binding 的時間
    - M - 總共變更的 bindings 數量 ( $M \leq N$ )

# OnPush 變更偵測策略的使用方式

- 在 @Component() 裝飾器中指定變更偵測策略

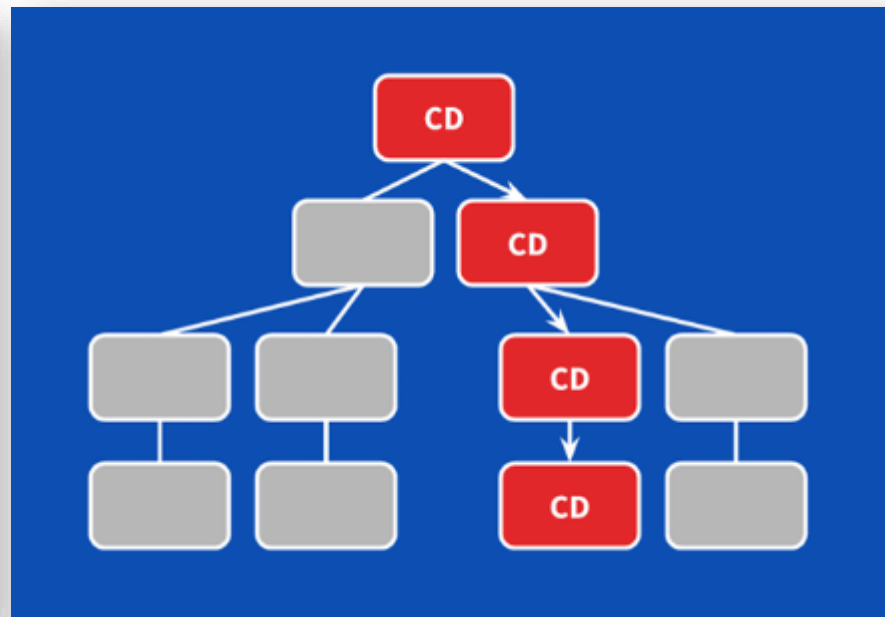
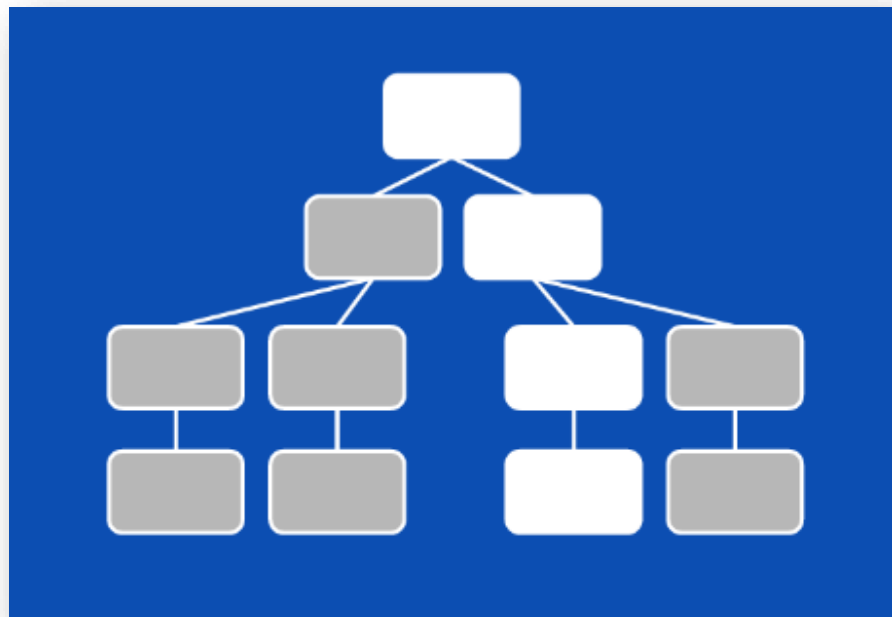
```
1  import { ChangeDetectionStrategy } from '@angular/core';
2  @Component({
3    template: `
4      <h2>{{user.name}}</h2>
5      <span>{{user.email}}</span>
6    `,
7    changeDetection: ChangeDetectionStrategy.OnPush
8  })
9  export class UserComponent {
10    @Input() user;
11  }
```

# 使用 OnPush 變更偵測策略注意事項

- OnPush 策略會在以下三種情況會偵測到變更
  - @Input() 屬性發生改變 (comparison by reference)
  - 在元件內觸發一個事件 (click, change, etc)
  - 在 Observable 內觸發一個事件
- 使用 Angular CLI 執行 `ng serve` 時的差異
  - 開發模式 (development)
    - 每次事件觸發變更偵測都會被執行兩次
    - `ng serve`
  - 生產模式 (production)
    - 每次事件觸發變更偵測只會執行一次
    - `ng serve --prod`

# 認識 ChangeDetectorRef 元件

- 使用 OnPush 策略時，可透過 [ChangeDetectorRef](#) 服務員見的 [markForCheck\(\)](#) 方法來標記變更偵測的時機，到時只有**元件到根元件的這條路徑**會需要在下一次變更偵測時執行。





# 使用 ChangeDetectorRef 服務元件

```
1 import { ChangeDetectionStrategy, ChangeDetectorRef } from '@angular/core';
2 @Component({
3   templateUrl: './app.component.html',
4   changeDetection: ChangeDetectionStrategy.OnPush
5 })
6 export class AppComponent implements OnInit {
7   @Input() addItemStream:Observable<any>;
8   constructor(private cd: ChangeDetectorRef) {}
9   ngOnInit() {
10     this.addItemStream.subscribe(() => {
11       this.cd.markForCheck();
12     });
13   }
14 }
```

# 認識 NgZone 元件

- [NgZone](#) 是 [Zone.js](#) 的 Angular 實作
- [NgZone](#) 的主要用途
  - 負責管理大部分瀏覽器內建的非同步函式執行！
  - 動態擴充了一些目前 [Execution Contexts](#) 底下的現有 API 並增加一些額外的功能。
  - 這件事等同於對 JavaScript 執行環境進行了一場心臟手術（手術沒有保證成功的，因此還是會有整合風險）

# 在 NgZone 外部執行程式碼

- NgZone 掌握了整個 Angular 應用程式生命週期
- 你可以藉由 NgZone 控制第三方函式庫的程式要在哪執行
  - 執行在 Angular 生命週期內
  - 執行在 Angular 生命週期外 ( 這樣外部程式就不會引發變更偵測 )

```
1  import { NgZone } from '@angular/core';
2  @Component({ ... })
3  export class NgZoneDemo {
4      constructor(private _ngZone: NgZone) {}
5      processOutsideOfAngularZone() {
6          this._ngZone.runOutsideAngular(() => {
7              // do something
8          });
9      }
10 }
```

# 相關連結

- Angular 2 Change Detection
  - [Angular 2 Change Detection Explained - YouTube](#)
  - [Angular 2 Change Detection Explained \(Slides\)](#)
  - [Angular Change Detection Explained by thoughttram](#)
  - [How does Angular 2 Change Detection Really Work?](#)
  - [Change Detection Reinvented Victor Savkin \(ng-conf 2016\) - YouTube](#)
  - [Change Detection Reinvented Victor Savkin \(ng-conf 2016\) - \(Slides\)](#)
  - [Change Detection in Angular](#)
- 認識 JavaScript 的可變性與不可變物件
  - [Change And Its Detection In JavaScript Frameworks](#)
  - [Immutable.js](#)
- 認識 Zones
  - [Understanding Zones by thoughttram](#)
  - [Zones in Angular 2 by thoughttram](#)
- Cheat Sheet
  - [Directive and component change detection and lifecycle hooks](#)

# 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](https://twitter.com/Will_Huang)