Angular Signals Workshop

Mike Huang @ *Hello World Dev 2025*

掌握 Angular 新一代的響應式狀態管理核心

`signal` - 響應式狀態的基礎

`signal` 是一個包裹著數值的容器，當它內部的數值改變時，會通知所有依賴它的地方進行更新。

```
import { signal } from '@angular/core';

// 建立一個 signal，初始值為空陣列
const tasks = signal<Task[]>([]);

// 讀取 signal 的值
console.log(this.tasks()); // []

// 更新 signal 的值
this.tasks.set([{ id: 1, title: 'Learn Signals', done: false }]);

// 透過 update 修改部分屬性；注意：immutable update
this.tasks.update(tasks => tasks.map(t => t.id === 1 ? { ...t, done: true } : t));
```

`computed` - 衍生的唯讀狀態

`computed` 讓我們可以根據一個或多個其他的 signal，建立一個新的**唯讀** signal。當來源 signal 變動時，`computed` 的值會自動重新計算。

```
import { signal, computed } from '@angular/core';

const tasks = signal<Task[]>([]);

const doneCount = computed(() => this.tasks().filter(t => t.done).length);

const completionRate = computed(() => {
  const total = this.tasks().length;
  if (total === 0) return 0;
  return Math.round((this.doneCount() / total) * 100);
});
```

`effect` - 響應式副作用

`effect` 用於執行副作用，例如資料持久化、手動操作 DOM 或 logging。它會自動追蹤內部的 signal 依賴，並在依賴變動時重新執行。

```
constructor() {
  effect(() => {
    localStorage.setItem('tasks', JSON.stringify(this.tasks()));
  });
}
```

注意：在 `effect` 中如果更新 signal，可能會產生意想不到的災難！

`untracked` - 避免不必要的依賴

有時候，我們在 `effect` 中需要讀取某個 signal 的值，但**不希望**該 signal 的變動觸發 `effect` 的重新執行。這時 `untracked` 就派上用場了。

```
effect(() => {
  // this.tasks() 改變會觸發 effect
  const tasks = this.tasks();
  saveTasks(tasks);

  // 但我們不希望 this.user() 的變動觸發 effect
  const currentUser = untracked(this.user);
  console.log(`Tasks saved for: ${currentUser.name}`);
});
```

`linkedSignal` - 可覆寫的衍生狀態

`computed` 是唯讀的，但 `linkedSignal` 提供了一個完美的解決方案：它既能像 `computed` 一樣從其他 signal 衍生，又允許你手動 `.set()` 一個新值來覆寫它。

- **智慧重置**：當來源 signal 變動時，`linkedSignal` 會被重置為衍生的預設值，覆寫掉手動設定的值。

`linkedSignal` 範例

將 `discountCode` 從 `computed` 改為 `linkedSignal`，讓使用者可以手動輸入折扣碼。

```
// solution/cart.ts
import { ..., linkedSignal } from '@angular/core';

const discountCode = linkedSignal(() => {
  const level = this.userLevel();
  if (level === 'platinum') return 'PLATINUM20';
  if (level === 'gold') return 'GOLD10';
  return 'WELCOME5';
});

discountCode.set('SPRING15'); // 手動覆寫
this.userLevel.set('platinum'); // 重置 discountCode
```

`input()` - 接收資料 (父 -> 子)

`input()` 將傳入的資料轉換成一個**唯讀的 Signal**。

- `input()`: 可選
- `input.required()`: 必要

```
// 子元件: checkout-summary.ts
import { Component, input } from '@angular/core';

@Component({...})
export class CheckoutSummaryComponent {
  finalPrice = input.required<number>(); // readonly Signal<number>
}
```

```
<!-- 父元件: cart.html -->
<app-checkout-summary [finalPrice]="parentFinalPrice()"></app-checkout-summary>
```

`output()` - 發送通知 (子 -> 父)

`output()` 用來定義一個事件發射器。

```ts
// 子元件: checkout-summary.ts
import { Component, output } from '@angular/core';

@Component({...})
export class CheckoutSummaryComponent {
  reset = output<void>();

  onResetClick() {
    this.reset.emit();
  }
}
```

```html
<!-- 父元件: cart.html -->
<app-checkout-summary (reset)="onParentReset()"></app-checkout-summary>
```

`model()` - 雙向綁定 (父 <-> 子)

`model()` 建立一個**可寫入的 Signal**，實現真正的雙向綁定。

- `model()`: 可選
- `model.required()`: 必要

```ts
// 子元件: checkout-summary.ts
import { Component, model } from '@angular/core';

@Component({...})
export class CheckoutSummaryComponent {
  discountCode = model.required<string>(); // WritableSignal<string>
}
```

```html
<!-- 父元件: cart.html -->
<app-checkout-summary [(discountCode)]="parentDiscountCode"></app-checkout-summary>
```

`viewChild` - 查詢單一子元件

`viewChild` 可以取得對子元件或 DOM 元素的參考，並回傳一個 Signal。

- `viewChild.required(Type)`: 回傳 `Signal<Type>`
- `viewChild(Type)`: 回傳 `Signal<Type | undefined>`

```
// tasks.ts
export class Tasks {
  statusBanner = viewChild.required(StatusBannerComponent);

  showBanner() {
    this.statusBanner().show('操作成功！'); // 記得用 () 取得實例
  }
}
```

`viewChildren` - 查詢多個子元件

`viewChildren` 一次取得**多個**子元件的參考，回傳一個 `Signal<readonly Array<Type>>` 。

```ts
// tasks.ts
export class Tasks {
  taskItems = viewChildren(TaskItemComponent);

  celebrate() {
    this.taskItems().forEach(item => {
      if (item.task().done) {
        item.playSuccessAnimation();
      }
    });
  }
}
```

Resource Family - 程式碼範例

它們都會回傳一個包含 `status`, `value`, `error` 等狀態的 Signal。

```
// Part A: Promise-based
readonly user = resource({
  params: () => ({ userId: this.userId() }),
  loader: (params) => this.mockApi.getUser(params.params.userId),
});

// Part B: Observable-based
readonly activityLog = rxResource({
  stream: () => this.mockApi.activityStream$,
});

// Part C: HttpClient-based
readonly tasks = httpResource<Task[]>(() => `/api/tasks?userId=${this.userId()}`);
```

Resource Signal 狀態

- `value: Signal<T>`
- `status: Signal<ResourceStatus>`
- `error: Signal<Error | undefined>`
- `isLoading: Signal<boolean>`

Resource 方法

- `hasValue(): boolean`
- `setValue(value: T): void`
- `update(updater: (value: T) => T): void`
- `asReadonly(): Resource<T>`
- `reload(): void`