

如何撰寫具彈性的測試程式

Summer@WebConf 2024



Summer

Summer。桑莫。夏天 cythilya.tw



前端測試指南：策略與實踐

打造高速網站，從網站指標開始！全方位提升
使用者體驗與流量的關鍵



[cythilya.dev](https://www.instagram.com/cythilya.dev)

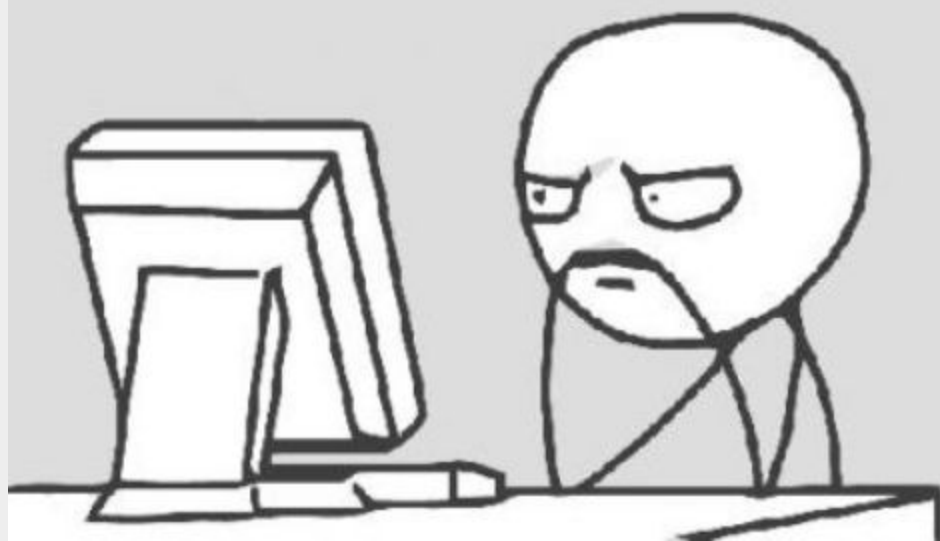


[cythilya](https://www.facebook.com/cythilya)



cythilya@gmail.com

UI 測試中





撰寫適應變更、有彈性的測試程式



設計保護力佳的測試架構

撰寫適應變更、有彈性的測試程式

測試為何失敗

UI 一更新測試就失敗，該怎麼辦？

選取元素的方式

太過鬆散、嚴謹或意義不明

太過鬆散

```
const Hello = () => {  
  return (  
    <div>  
      這是盒子  
      <div>測試文字</div>  
    </div>  
  );  
};
```

```
it('should render the correct  
content', () => {  
  const { getByText } = render(<Hello  
/>);  
  
  改為 toHaveTextContent 更  
  具可讀性  
  expect(  
    getByText('測試文字').textContent  
  ).toBe('測試文字');  
});
```

需求變更，改起來很麻煩 :(

```
it('should render the correct content', () => {  
  const { getByText } = render(<Hello />);  
  
  expect(getByText('測試文字').textContent).toBe('測試文字');  
});
```

改為「不是測試文字」

The diagram consists of two white curved arrows pointing from the original string '測試文字' to the new string '不是測試文字'. The original string is crossed out with a red line. The new string is written in white text below the original one.

無法辨識唯一性

```
const Hello = () => {  
  return (  
    <div>  
      這是盒子  
      <div>這是測試文字</div>  
      <div>這是測試文字</div>  
    </div>  
  );  
};
```

太過嚴謹

XPath: /div/div

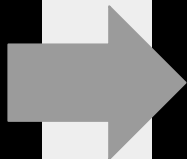
```
const Hello = () => {  
  return (  
    <div>  
      這是盒子  
      <div>測試文字</div>  
    </div>  
  );  
};
```



更新標籤

```
const Hello = () => {  
  return (  
    <div>  
      這是盒子  
      <div>測試文字</div>  
    </div>  
  );  
};
```

XPath: /div/div



```
const Hello = () => {  
  return (  
    <div>  
      這是盒子  
      <p>測試文字</p>  
    </div>  
  );  
};
```

XPath: /div/p

意義不夠明確

樣式與測試混用

```
const Hello = () => {  
  return (  
    <div className="text">  
      測試文字  
    </div>  
  );  
};
```

`data-* attribute`

專注與彈性

```
const Hello = () => {  
  return (  
    <div data-test-id="text">  
      測試文字  
    </div>  
  );  
};
```

`data-test-id`

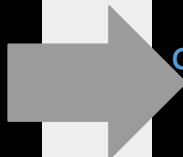
```
it('should render the correct content 2', () => {  
  const { getByTestId } = render(<Hello />);  
  
  expect(getByTestId('text')).toHaveTextContent('測試文字');  
});
```


包含過多實作細節

過多模擬

```
jest.mock('./Hello, () => (  
  <div data-test-id="hello">Hello  
World!</div>  
));
```

```
const Hello = () => {  
  return (  
    <div data-test-id="hello">  
      Hello World!  
    </div>  
  );  
};
```



```
jest.mock('./Hello, () => (  
  <div data-test-id="hello">Hello  
World!</div>  
));
```

```
const Hello = () => {  
  return (  
    <div data-test-id="hello">  
      Hi!  
    </div>  
  );  
};
```

快照

記錄實作細節

```
exports[`ImageList should render correct
image items when have data 1`] = `
Array [
  <div
    data-test-id="image-item"
  >
    
    <p data-test-id="image-item-title">
      打造高速網站, 從網站指標開始!
      全方位提升使用者體驗與流量的關鍵
    </p>
  </div>,
  <div
    data-test-id="image-item"
  >
    
    <p data-test-id="image-item-title">
      島波海道。單車跳島之旅
    </p>
  </div>,
]
```

比對細節

```
<div data-test-id="item-quantity">  
  2  
</div>
```



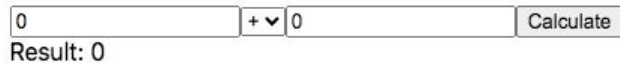
```
expect(getByTestId('item-quantity'))  
  .toHaveTextContent('2')
```



```
expect(getByTestId('item-quantity'))  
  .toBe(DOM 結構)
```

拆分邏輯、狀態與顯示

`<Calculator>`



0 + 0 Calculate
Result: 0

```
const Calculator = () => {
  const [num1, setNum1] = useState('');
  const [num2, setNum2] = useState('');
  const [operator, setOperator] = useState('+');
  const [result, setResult] = useState('');

  const handleChange = (e) => // 略...

  const calculateResult = () => {
    switch (operator) {
      case '+':
        setResult(parsedNum1 + parsedNum2);
        break;
        // 略...
    }
  };

  return (
    <div>
      <input type="number" value={num1} onChange={handleNum1Change} />
      <select value={operator} onChange={handleOperatorChange} >
        <option value="+">+</option>
        // 略...
      </select>
      <input type="number" value={num2} onChange={handleNum2Change} />
      <button onClick={calculateResult}>Calculate</button>
    </div>Result: {result}</div>
  );
};
```

`<Calculator>`

將運算放在 utils

```
const calculate = (num1, num2, operator) =>
{
  switch (operator) {
    case '+':
      return num1 + num2;
    case '-':
      return num1 - num2;
    // 略...
  }
};
```

寫測試

將資料運算放在 utils

優點

- 對重構專案的壓力最小、最容易實作
- 保護效果顯著
- AI 快速實作, 節省成本

```
describe('calculate function', () => {  
  test('should get 8 when add 5 and 3', ()  
=> {  
    expect(calculate(5, 3, '+')).toBe(8);  
  });  
});
```


`<Calculator>`

將商業邏輯與資料狀態
封裝在 custom hook

```
const useCalculator = () => {  
  const [num1, setNum1] = useState(0);  
  const [num2, setNum2] = useState(0);  
  // 略...  
  
  const handleNum1Change = e =>  
    setNum1(parseFloat(e.target.value) || 0);  
  // 略...  
  
  return {  
    calculateResult, handleNum1Change, ...  
  };  
};
```

寫測試 - 將商業邏輯與資料狀態封裝在 custom hook

```
test('should get 10 when add 7 and 3', () => {  
  const { result } = renderHook(() => useCalculator());  
  act(() => {  
    result.current.handleNum1Change({ target: { value: '7' } });  
    result.current.handleNum2Change({ target: { value: '3' } });  
    result.current.handleOperatorChange({ target: { value: '+' } });  
    result.current.calculateResult();  
  });  
  expect(result.current.result).toBe(10);  
});
```

`<Calculator>`

將畫面顯示放在元件

```
const Calculator = () => {
  const { handleNum1Change, ...} = useCalculator();

  return (
    <>
      <input
        data-test-id='number1'
        type='number'
        value={num1}
        onChange={handleNum1Change}
      />
      <select
        data-test-id='operator'
        value={operator}
        onChange={handleOperatorChange}
      >
        <option value='+'>+</option>
      </select>
      // 略...
```

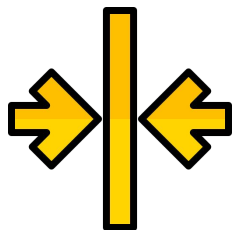
寫測試 - 將畫面顯示放在元件

```
test('updates result correctly for subtraction', () => {  
  render(<Calculator />);  
  
  const number1Input = screen.getByTestId('number1');  
  // 略...  
  
  fireEvent.change(number1Input, { target: { value: '20' } });  
  fireEvent.change(number2Input, { target: { value: '7' } });  
  fireEvent.change(operatorSelect, { target: { value: '-' } });  
  fireEvent.click(calculateButton);  
  expect(result).toBe('13');  
});
```

總結 - 撰寫適應變更、有彈性的測試程式



data-*
attribute`



避免過多
實作細節

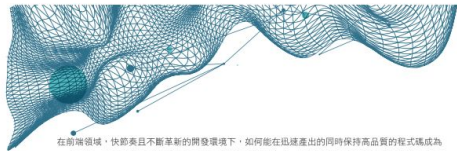


拆分邏輯、狀態
與顯示



打造高速度網站從網站指標開始
全方位提升使用者體驗與流量的關鍵

本書以淺顯易懂的圖文說明如何改善效能、可量化的網站互動操作的範例程式碼，提供學習與演練。作者以使用者的角度解釋效能、改善效能，不再讓優化效能這件事變得眼花、沒了頭及感受到任何效果。並以實際的網站說明如何改善效能，使於評估自身產品的優化成本。



在前端領域，快節奏且不斷革新的開發環境下，如何能在迅速產出的同時保持高品質的程式碼成為一大挑戰，因此「測試」變得十分重要。選擇何種測試方式、如何撰寫和執行測試成為開發者必須深思的議題。缺乏測試的程式碼難以確保品質，而解決這個問題的關鍵在於建立適當的測試策略。

專、業、精、緻

這本書非常適合前端工程師全面性地打基礎，不僅能學會撰寫測試程式，還能全局地做出適當的決策。我喜歡這本書的地方在於其結構分明、面向完整，閱讀起來緊湊而輕鬆易懂，點對大小通中且容易實作。這本書可以幫助前端和後端工程師節省時間，進而更有效地學習，拓展人生與職涯的長度、廣度與深度。感謝 Summer 的付出，幫助大家提升產品品質，改善使用者體驗。

Odd-e Taiwan 敏捷技術教練 | 陳仕傑 (91) 2024/5/1

雖然寫測試可能還是靠開發者自己的個人經驗，但還是有一些前輩或大神們整理出來的方法論可以參考。

跟著 Summer 的書學怎麼寫前端測試，也就是學怎麼寫出更有讓自己更有信心的前端程式碼！

五倍學院 | 吳見龍

在 Summer 的這本書中，透過深入淺出的方式，介紹了各種不同類型的測試，從單元測試、整合測試、端對端測試，一直到前端特有且重要的視覺測試，全部都有所著墨，同時也示範了如何在 CI 上執行測試，並結合豐富的實務經驗，說明了前端測試為什麼會更著重在測試行為，而非程式的實作。書中更有系統地整理出不同時機適合使用的測試方式和工具。如果你想要對於前端測試有更全面的認識，從類型、工具、到執行有更完整的了解，這絕對是一本值得一看的好書。

PJCHENDER 網頁開發社群版主、《從 Hooks 開始，讓你的網頁 React 起來》作者 | 陳柏融

ISBN 978-626-333-874-6
MP22436



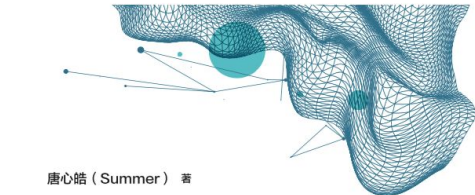
博碩文化股份有限公司
遠東書局·郵政特約·人工智慧
定價：400元



前端測試指南 策略與實踐

唐心皓 (Summer) 著

博碩



唐心皓 (Summer) 著

前端測試 指南

策略與實踐

聚焦 × 高效 × 卓越

- 透過圖文並茂、程式碼範例的深入說明，輕鬆學習前端測試。
- 提供明確的測試策略與成本評估原則，並依此做出最適合的選擇。
- 分享有效使用 AI 工具產生測試程式的秘訣，提高效能。
- 透過清楚易懂的範例，深入地解答各類挑戰，助你應對測試難題。



作者
簡介

唐心皓
Summer



工程師、講者以及作家，是技術部落格「Summer 桑諾·夏天」與《打造高速度網站：從網站指標開始》全方位提升使用者體驗與流量的關鍵》的作者。擅長 SEO、前端效能與測試，相信知識的分享能打破隔閡，讓世界更加美好。

如何在快節奏的開發步調中，運用出色的測試策略與技術，保持競爭力，持續突破開發瓶頸，高效產出卓越的產品。



前端測試指南：策略與實踐

設計保護力佳的測試架構

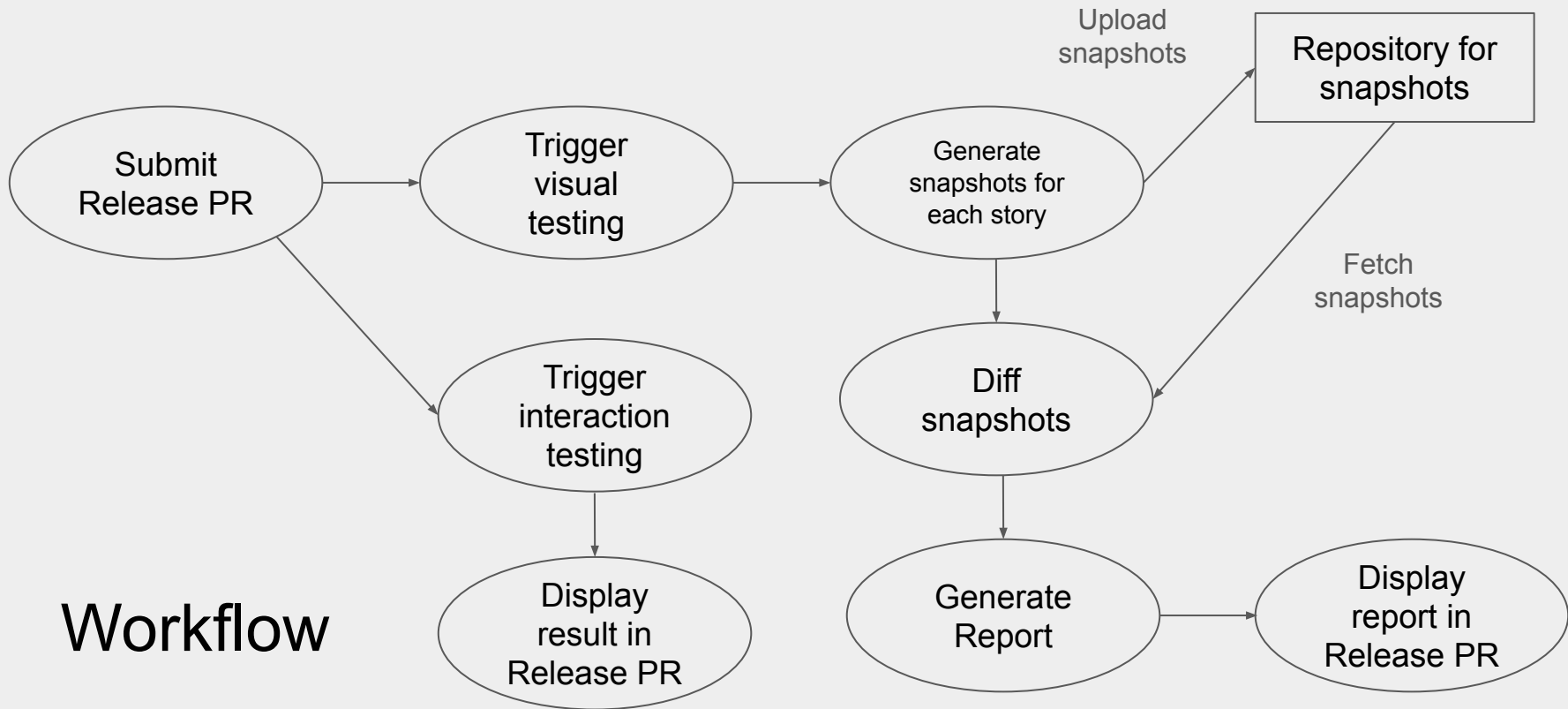


Visual
testing



Interaction
testing





Workflow

總結 - 設計保護力佳的測試架構



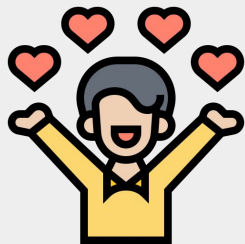
視覺



行為



成本



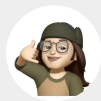
突破框架，流程整合



特別感謝



Max Shen



Taylor Wu

Thank you!