前言

近年來的前端開發由於函式庫及框架日益發達，開發方法發生了巨大的變化。有著「現代前端」之稱的任務堆疊廣泛應用在產品當中，變得相當普及。然而在這樣的世局之下，我們依然經常會聽到許多人對「測試程式碼」有著下述的意見。

- 對怎麼寫程式碼毫無頭緒，既茫然又焦慮
- 雖然可以寫出一定程度的測試程式碼，但沒有足夠的信心確信已經寫得夠好
- 會想知道其他公司裡面寫了多少、以及本於哪些依據在寫測試程式碼

因此筆者在想，或許依舊有許多人懷著著忐忑的心、無法對寫測試一事抱持自信吧！再者，舉凡如UI元件測試、視覺回歸測試、Storybook、E2E（End-to-end）測試，皆屬前端測試的一環。測試方法一多，要想在第一次遇到時就能明辨這些測試方法該如何使用，是極為困難的事情。打從一開始就有這麼多的選擇，或許也是相當令人苦惱的事。

選擇多，也意謂著每個專案都能找到最適合自己的測試方法。充分瞭解測試方法，就更有可能挑到越適合的選擇。我們會使用Next.js應用程式範本來講解不同測試方法的具體案例，或許有助於讀者更能釐清「在什麼樣的場合應該選擇哪個測試方法？」這個問題。本書將抱持特定目的來搭配不同測試方法進行運用一事稱之為「測試戰略」，而這個「測試戰略」正是貫穿書中全文的核心主題。

期待讀者們都能因為本書獲益良多，從明天開始就自信滿滿地寫下測試程式碼，倘若能讓更多人體會到「會寫測試真是太好了！」，那將是筆者最大的榮幸。

1-2 寫測試的目的

會拿起這本書，或許是因為日常業務當中自己無法順利寫好測試程式碼而感到徬徨不安吧？不過我想也有人是聽說「業界人士都說這是必備技能」、「會減少出錯的機會喔」等感想，進而對拙作抱持興趣也說不定。

為什麼需要學寫測試程式碼這件事，是需要實際經歷「幸好有寫測試程式碼！」這樣的心境才會感受到的。不過從實際開始寫、並且能感受到這件事帶來什麼好處，得要經歷一段不算太短的路程。請先容筆者和各位分享，我認為寫測試程式碼的目的是什麼吧！

● 為了建立值得信賴的服務

當軟體含有會為事業帶來商業影響的錯誤，在我們重新獲得客戶的信賴之前，所有的計畫都會間接地遭到商業影響。一旦發佈了有錯誤的程式，不僅是大家無法使用我們提供的服務，對於該服務的印象也會大打折扣。而為了要防範不樂見的情況發生，大概就會是我們願意去加入自動化測試的原動力了（圖1-1）。

身處這個時代，前端開發工程師也有許多機會參與開發前端的後端伺服器（BFF, Backend For Frontend）。尤其在BFF的開發上經常會遇到驗證、認證等，我們希望任何細節盡可能不要出現錯誤。微小的錯誤或許無傷大雅，但倘若我們都能在日常養成寫測試的良好習慣，就更容易幫助我們去注意到細節裡的魔鬼。

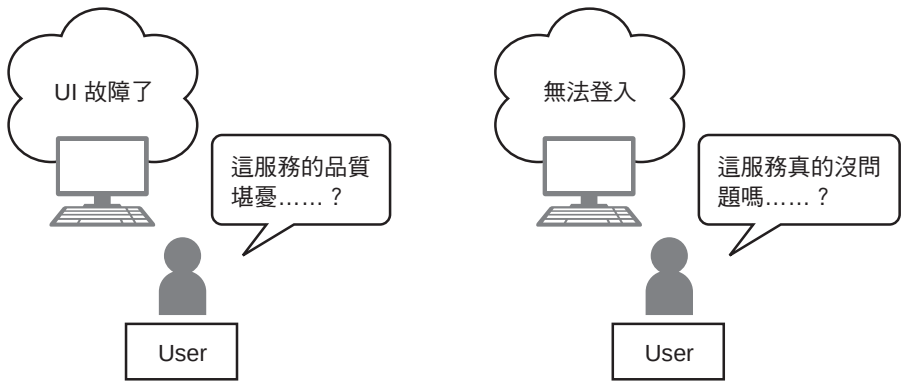

圖1-1　UI或系統的故障將會直接影響服務品質給人的印象

● 讓程式更順利運作

當專案持續進行到一個階段，很常會遇到需要搭載類似功能的地方，此時我們會需要將類似的處理以模組化的方式來進行共用、或是對程式碼進行重構。同時卻也會擔心做了這些事情會不會影響既有順利運作中的功能，而最後決定放棄不做。不曉得大家有沒有經歷過類似的情況呢？

倘若我們可以養成日常寫測試的習慣，在重構程式碼時就能逐一執行測試、確認既有功能是否遭受影響。有測試程式碼隨侍在旁所帶來的安心感，可以促使我們更積極地進行重構，讓程式整體都更加健全（圖1-2）。

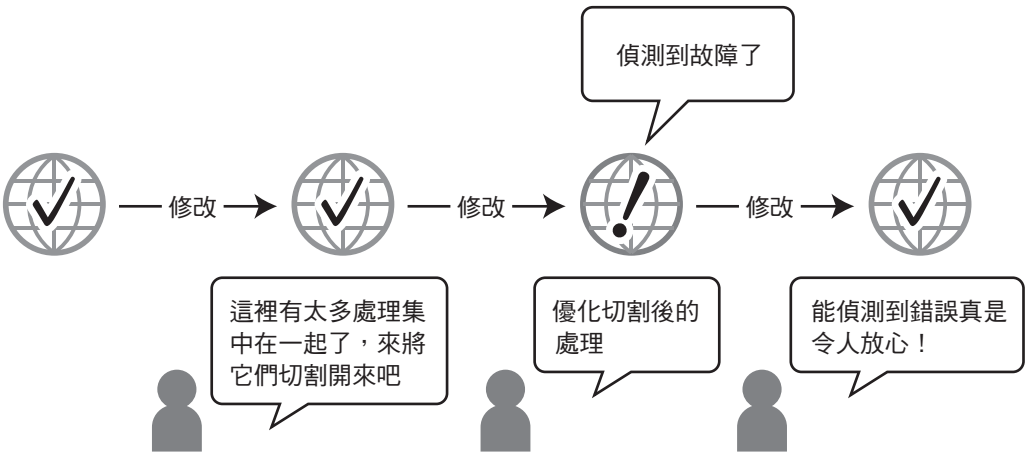

圖1-2　有測試的輔佐，讓我們得以持續進行重構

並不是只有開發新功能時才會需要修改程式碼。相信各位都聽過Dependabot對吧？這是當專案所依賴的函式庫出現漏洞時、或者發佈了最新版本時，可以替我們建立拉取要求（Pull Request）的bot。前端開發大多時候都會希望依賴著函式庫、並藉此盡可能頻繁地進行更新。

雖然Dependabot能自動建立拉取要求是很棒的事，但真的能完全相信那個拉取要求嗎？會不會擔心更新之後導致原本可以用的功能變成不能用？要是我們可以寫好自動化測試，就能加上「當進行小幅度更新的時候，通過測試再進行合併」的規則，對吧！

● 對品質更有信心

可以將寫測試這事看作是重新檢視自己程式碼品質的機會。如果覺得寫起來心有餘而力不足，或許意謂著我們對軟體硬塞了太多的處理也說不定。只要將單純的輸入、輸出的函數進行多個分割，測試程式碼就會變得非常好寫。而這種修改蠻常會帶領我們更順利地邁向完成。

比方說有個太笨重的UI元件，包含了畫面顯示、輸入驗證、非同步處理更新等。當我們在一個UI元件（函式）參雜了這麼多不同的需求，就會毫無頭緒該怎麼寫測試才好。不過，如果把畫面顯示、輸入驗證、非同步處理更新都各自獨立為單一的元件（或是函式），不僅最後完成時容易統整，測試也比較好寫。

除此之外，在開發UI元件時會需要考量到網頁的無障礙性、或稱無障礙網頁（圖1-3）。最近使用無障礙性相關的元素獲取API來編寫UI元件測試程式碼的機會增加了。獲取無障礙性相關的元素是指取得對身心障礙者友善的的元素。如果無法捕獲這些元素，我們可能就要留意那些使用螢幕閱讀器等輔助技術的使用者，可能無法獲得我們原本期望提供的內容體驗。

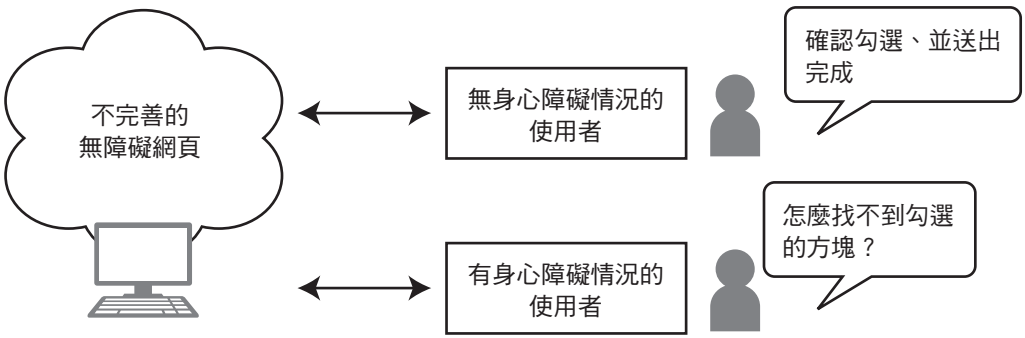

圖1-3　UI元件品質會左右網頁的可及性

這只是其中一個例子。在開發軟體的同時寫測試，就能獲得站在不同角度檢視程式碼的機會，也能令自己對軟體成果更有信心。

● 讓合作更順暢

專案進行開發時，最重要的就是需要顧慮到自己以外的團隊成員。相信有蠻多專案都有Code Review的文化。即便是負責檢查的工程師（Reviewer）將程式碼從頭到尾照順序看完，也不見得就有信心找出所有思慮不周跟出錯的地方，更甭說還要曠日廢時地確認運作情況。我們加入了新的專案團隊後會匯入程式碼跟文件，此外也會建立開發所需的伺服器來確認運作，要掌握專案的全貌是相當費時的事。因此細心的體察與溝通就會直接影響整個團隊的設計執行進度，而這也是大多數的工程師都會留下程式碼以外的補充資訊的原因。

相較於一般的設計文件，測試程式碼可說是更加優秀的補充資訊（圖1-4）。每個測試都有標題，提供什麼樣的功能、會怎麼運作都寫得一清二楚。只要通過這些測試，補充內容與實際完成的內容就不會有出入（對了，要注意標題跟測試內容可能不完全相同的情況）。

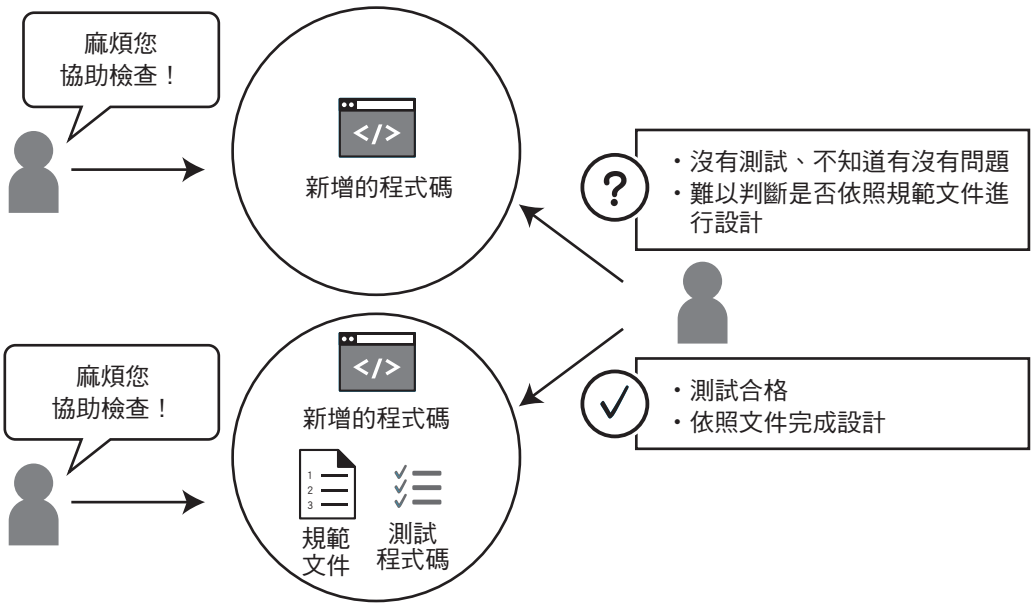

圖1-4　透過檢查來確認新增的程式碼、規範文件、測試程式碼

測試不僅是可以檢查有沒有技術上的缺失，也能確認成果有無符合客戶的需求。互相比對測試程式碼與規範文件、再來確認程式碼，相信會減少許多檢查人員（Reviewer）的負擔。如果能在CI（持續整合，continuous integration）過關之後再來請人檢查，還可以縮短指正與修改等討論溝通的往返時間。

將測試程式碼作為產品程式碼的概要來進行分享，有助於讓工程師之間的合作更加順暢。

● 為了避免降級問題

可能有些人會遇到接下來需要重構、為避免出現降級問題而需要寫測試的情況。日常當中所執行的自動化測試是防止出現降級問題的最佳幫手。

透過細分模組，讓每個模組所負責的任務跟測試都盡量單純化。反之，當模組們彼此相互依賴時、可能會因為修改了被依賴的模組而引發降級問題。這是對現代前端開發最直接的挑戰（圖1-5）。

在第7章的整合測試將會講解如何測試多個模組（UI元件）連動運作的方式。UI元件跟功能會決定外觀（風格），就算內部的測試寫得再完善，也無法防止外觀的降級問題。這部分將會透過第9章的視覺回歸測試來解套。

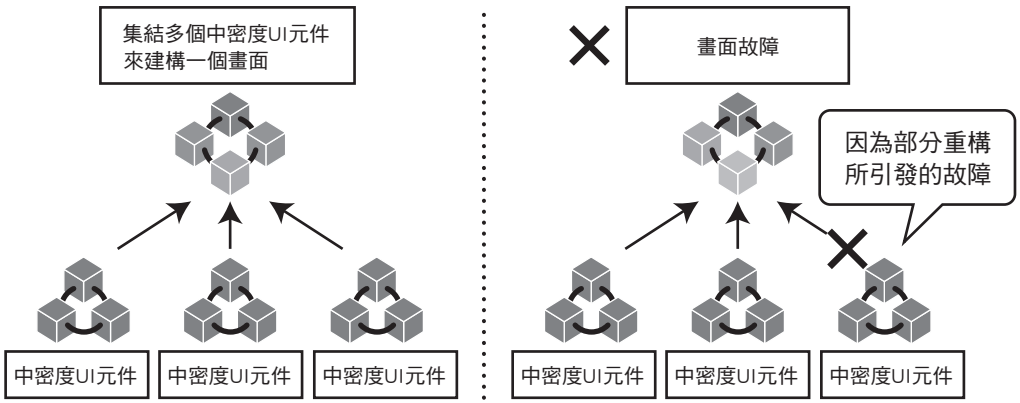

圖1-5　因部分UI重構所引發的畫面故障

1-3 寫測試的障礙

聽過許多人會分享，明明他們已經決定要寫測試了，但腦海裡卻出現了以下的消極聲音，導致無法動手開始寫。

- 沒有寫測試的習慣，不知道怎麼開始寫
- 如果有那美國時間寫測試，倒不如拿來增加功能
- 團隊成員技能參差不齊，對於維護運用沒信心

當一個團隊想要運用自動化測試時，難免都會遭遇到這些障礙吧！

● 測試，該怎麼寫才好？

身處前端開發的第一線，被告知「下個專案我們已經決定要用最新的函式庫了」真的只是家常便飯。然而，當我們首次遇到函式庫時，即便再怎麼樣依照官方文件去做，要想得心應手地做出心目中的應用程式，仍然會花上不少時間。

請各位回憶那些專案當中被要求新增功能的時候，我們是不是會參考專案中過去已經定案的程式碼，並且依照指南來進行程式碼的編寫。這種方式其實遠比盯著文件要來得令人更快熟悉專案的狀況。寫測試程式碼也相當類似，當專案內有許多值得參考的程式碼，我們上手的速度就會快很多。

要是專案當中完全沒有可以作為參考的測試程式碼，不妨就讀看看本書當中的範本吧！有樣學樣、或者複製貼上也沒關係，只要一而再、再而三地練習，就能精進寫測試的技巧。唯一的捷徑，就是拿既有的案例多看多練習。這本書也是基於為了讓各位可以在那些練習當中更加活用相關知識與技巧，力求提供更貼近實際工作時的測試程式碼範本給大家。

書中雖然有很多測試方法，卻不免令人感到「要全部都拿來用似乎也是有難度」。別煩惱，讓我們先掌握訣竅，至於該寫多少測試，可以跟團隊成員互相討論。就讓這本書帶領各位學習基本的測試寫法，排除「寫測試的障礙」吧！

● 如何確保寫測試所需的時間？

即使知道該寫測試，卻經常聽到「可是我沒時間寫」。如果連測試程式碼也要一併提交，短期來看會導致開發速度變慢。或許唯一要確保有足夠的時間的方法，不外乎就是提升自己能更快寫好測試的速度，但並不是所有人都可以快速地寫出測試。

為了要在發布正式版本時可以一併提交測試程式碼，還是得保留相對應的充分時間來進行作業。所以自動化測試也得視為開發項目，確實地將其納入整體排程規劃。而要能做到這點，則需仰賴團隊都抱持「自動化測試是不可或缺的存在」的認同感。

由於框架跟函式庫變化太快，時常聽到有人說前端程式碼的壽命很短。確實，就算現在花上大把時間去完成開發，可能經過一段時間後就又得要再改了。於是不免有人認為「反正都要再改，那不寫測試也不會怎麼樣」，但以筆者自己的經驗來說卻未必如此。

筆者曾在參與某個專案時，經歷了軟體發佈過了半年之後、還需要更新UI的狀況。當時專案為了趕著上線、時程相當緊湊，但後來由於企業品牌策略的關係而需要將UI砍掉重練。這件事對於認為整個系統功能堪稱完備、即便出現需求變更大概也不會是很大幅度修改的我自己來說，完全是出乎意料之外。

幸好當時的專案已經有寫測試的習慣，讓我們得以在無須過度心驚膽跳的情況下，完成了巨大的需求變更。在更新過程當中也充分地透過測試來理解受到影響的功能，迅速地完成任務。

● 寫測試為什麼能幫我們省時

倘若以短期觀點來看，寫測試或許淪為了剝奪個人時間的瑣事，然而當改以長期觀點評估時，卻是能夠省下整個團隊的寶貴時間。讓筆者以「某位工程師所部署的功能含有錯誤」這個前提來跟各位分享差異在哪，並假設完整部署功能需要16小時、自動化測試時間需要4小時。

當我們執行了自動化測試，就能在這4小時之內儘早發現、排除問題，於是總計花費時數為20小時。反之，當沒有執行自動化測試時，花費時數就是16小時而已。這應該很容易成為那些嘴上說著「不要執行自動化測試會比較快」的人的說詞。

可是，當測試工程師在手動測試階段發現了問題，製作提報問題的資料並請開發工程師進行修改，爾後再次驗證問題是否已經排除，也就是「重工（rework）」。而為了

處理這個重工階段又耗了4個小時。可以看到就算不做自動化測試，加上處理重工的時間總共至少要花20小時以上的時間。

比較前後兩者，或許會認為花費時間差異不大，但重點在於「是否留下了自動化測試程式碼這個足以作為資產的東西」。然後長期下來，那些如果寫了自動化測試就能避免發生的降級問題，將會在營運階段浮上檯面、造成時間的浪費（圖1-6）。

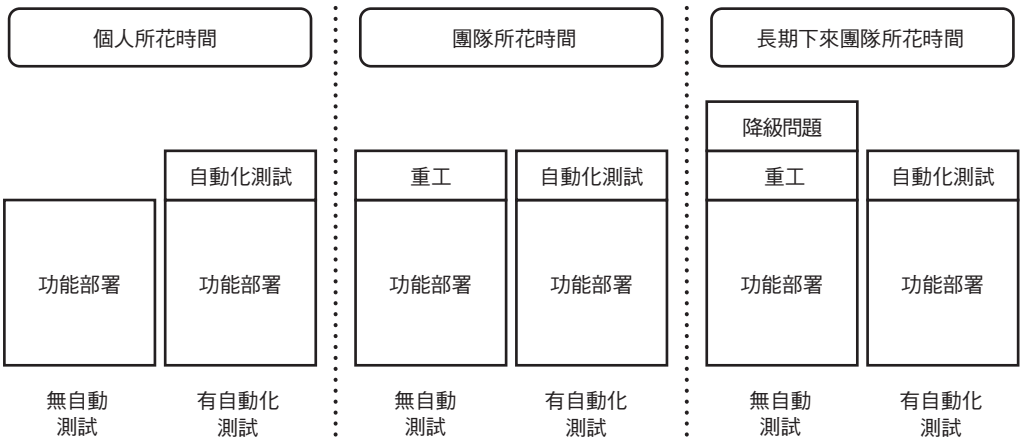

圖1-6　以長期觀點來看，自動化測試可以為我們節省時間

這邊的例子我們假設為定量化的值、並且一定會出現問題為前提來示意給大家理解，相信各位都應能體會到「整體來看，儘早導入自動化測試會更有助於節省時間」的概念。**而寫測試所節省下來的時間，正是專案進行的緊迫時程內的寶貴時間。**從進行設計開發的同時就導入自動化測試，長期看下來對於團隊是相當有助益的做法！

● 如何讓所有團隊成員都願意寫測試？

「這個專案從來也沒有人寫過測試啊」經常會被拿來當作在開發時不想同時寫測試的理由。當一個專案已經完成所有軟體發佈、進到了營運階段時，已經很難跳脫某種原則。因為以前都不必寫測試，所以現在跟以後不寫也沒關係的默契已經深植人心。

「事後再來寫測試」的方式，基本上都得將相關人員拖下水，設定好里程碑，並且由多人來同時執行，相當勞師動眾。這真的是比想像中還要辛苦很多，筆者也曾經身處決定「事後再來寫測試」的專案而有過令人後悔的經驗。隨著時間過去，需要寫測試的對象不斷增加，要達成的難度也持續變高。雖然勞師動眾也並非辦不到，但總是會希望不要落入這步田地。

筆者認為一個團隊寫測試的習慣能否生根，在最早期的設計階段就已經確定了。趁著程式還比較小的時候盡快先釐清方針，大家就會知道測試該如何寫才好。**樹立典範之後，就算團隊成員有人不擅長寫測試，也能參考前人所做的案例來寫出一定程度的測試。**

如同「沒有時間寫測試」這個理由一樣，為了要讓團隊成員都願意寫，重點在於盡快寫出能留作範本的測試。希望書中的測試程式碼也有榮幸可以成為大家能參考的範例之一。

2-1 以範圍與目的來思考測試

由於書中提及許多測試方法，為了避免大家出現選擇障礙，在進入到講解測試方法的篇章之前，建議先閱讀本章的「前端測試的範圍與目的」來加深理解。

比起有勇無謀地直接著手進行，先理解「範圍」如何搭配「目的」一起考量，選擇最合適的自動化測試，一起感受它所帶來的好處吧！

等各位讀完本書之後，第二章依然是可以用來當作總結的篇章。當掌握了書中所講解的所有內容後再回來看，相信各位將會對前端測試的全貌有更深的理解。

● 測試的範圍

Web應用程式的程式碼當中會組合許多模組來達成需要的功能。比方說為了要提供一個功能時，就會需要結合以下一連串的模組（系統）。

① 函式庫所提供的函式
② 負責邏輯的函式
③ 顯示UI的函式
④ Web API Client
⑤ API伺服器
⑥ 資料庫（DB）伺服器

寫前端的自動化測試時，我們需要意識到上述①～⑥去思考「測試應該涵蓋的範圍是從哪到哪」。Web前端開發的測試範圍大致分為以下4種。

靜態分析

使用TypeScript或ESLint進行靜態分析。這不是去確認每一個模組的內部，而是針對②跟③之間、③跟④之間「彼此相連的模組間的連動順暢與否」。

單元測試

僅針對②、或是僅針對③，著重在「單一模組功能」的測試。由於可以獨立進行驗證，因此適合在應用程式運行時用來確認不常發生的邊角案例（corner case）。

整合測試

用來測試從①～④、或是從②～③這種「模組串連起來所提供的功能」。雖然範圍設定的越廣、就能更有效率地驗證更多項目，但整個測試也較為籠統。

E2E 測試

運用無頭瀏覽器＋UI自動化來執行貫穿①～⑥整體的測試，是範圍最大的整合測試，能忠實呈現應用程式的運行狀況。

● 測試的目的

測試目的不同，**測試類型**也不同。軟體測試當中較廣為人知的測試是「功能測試、非功能性測試、白箱測試、回歸測試」。

測試類型會依照驗證的需求來訂定，每個測試都有最適合的工具可以選用。有些工具可以獨立運作，有些則是需要搭配組合來實現需要的功能。下面分享幾個Web前端測試當中最具代表性的測試類型。

功能測試（interaction testing）

功能測試確認開發出來的功能有無問題。Web前端所開發的功能大多都是以UI元件的互動（interaction）為出發點，因此互動測試（interaction testing）經常會直接當作功能測試來執行，相當受到重視。如果是需要用到正式瀏覽器API進行測試的重要情況，則會使用無頭瀏覽器＋UI自動化來寫測試。

非功能性測試（accessibility testing）

在非功能性測試當中，**無障礙性測試（accessibility testing）**是用來驗證產品顧及身心障礙者友善程度的測試。近年來，主打無障礙網頁的API已經遍佈各大平台，讓自動化測試可以相當客觀的角度來做出判斷。

回歸測試（regression testing）

回歸測試是從某個特定時間點去區分前後的差異、進行驗證，以確保沒有料想之外的問題出現。

2-2 前端測試的範圍

讓我們再來多談一點有關Web前端測試的範圍。

● 靜態分析

從儘早發現錯誤這點來看，TypeScript的**靜態解析**是不可或缺的存在，它特別擅長重現執行環境（runtime）的運作，例如使用`if`條件分歧安全地處理值（List 2-1）。

▶ List 2-1　重現執行環境運作的型別推論

TypeScript

```
function getMessage(name: string | undefined) {
  const a = name; // a: string | undefined
  if (!name) {
    return `Hello anonymous!`;
  }
  // 透過if條件式與return來判斷並非undefined
  const b = name; // b: string
  return `Hello ${name}!`;
}
```

此外，靜態驗證也能用於驗證函式是否回傳了我們所期待的值。以List 2-2來說，為了不讓回傳值的型別淪為`string | undefined`（字串或`undefined`），在函式區塊的最後面拋出例外。藉此讓型別推論成為一定會送出字串、也就是回傳值必定為`string`格式。

▶ List 2-2　回傳值型別推論是string | undefined，由於不一致所以跳出型別錯誤

TypeScript

```
function checkType(type: "A" | "B" | "C"): string {
  const message: string = "valid type";
  if (type === "A") {
    return message;
  }
  if (type === "B") {
    return message;
  }
  // 依據有無發生例外，函式回傳值的型別推論會隨之改變
  // throw new Error('invalid type')
}
```


用於作為程式碼編寫準則的ESLint也屬於靜態分析之一。藉由迴避不適當的語法構造，達到防範摻雜潛藏錯誤於未然的效果（List 2-3）。函式庫開發人員會提供適合該函式庫的建議設定讓我們來使用，而當我們遵循著正確的用法，好處是日後就能注意到哪些API不建議使用。

▶ List 2-3　違反了函式庫所建議的程式碼編寫準則

TypeScript

```
useEffect(() => {
  console.log(name);
}, []);
```

發生了Lint錯誤，因為所依賴的引用值name應該包含在陣列中

● 單元測試

單元測試是最基本的測試，用來測試模組是否能依據既定的輸入值去得出我們所期待的輸出值。在單頁應用（single-page application，SPA）開發當中，較常將UI元件作為測試對象。我們可以使用與函式單元測試相同的要領來測試從輸入值（Props）得到輸出值（HTML區塊）的UI元件。

有些模組在出現了鮮少發生的邊角案例（conner case）時要判斷中斷處理，此時「該以什麼條件」來拋出例外，就相當適合以單元測試來進行評估。透過重複地去商討「這樣的條件可行嗎？」、「可行的話該如何處理才好？」，是讓我們養成習慣去察覺程式碼中可能出現的紕漏的契機（圖2-1）。

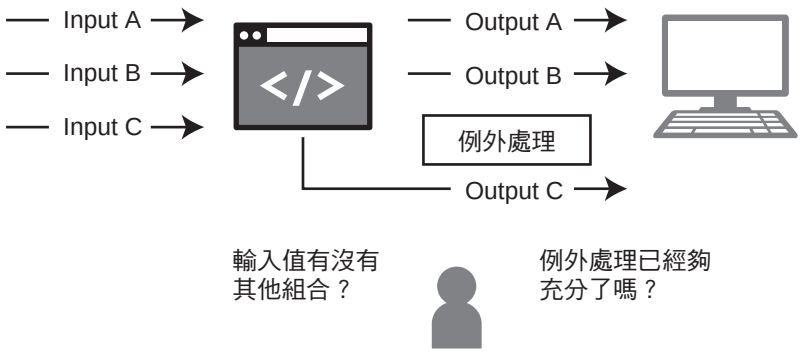

圖2-1　在單元測試中評估對函式的考量是否充分

● 整合測試

整合測試是著重在測試多個模組串連起來提供的功能。大型UI元件的功能很少由單一模組來提供完整功能，都是透過許多模組整合在一起來達到需要的功能。這些功能主要都是透過模組彼此互動而實現。我們以Web應用程式的元素總覽畫面來研究看看吧！

① 操作選擇框
② 改變URL的搜尋查詢內容
③ 配合搜尋查詢內容的改變，呼叫取得資料的API
④ 更新總覽畫面的顯示內容

只需要透過「操作選擇框」這一個動作，就能連帶處理到最後的「更新總覽畫面的顯示內容」都幫我們完成。而著眼於「執行了①之後、④也會執行」的測試，就是整合測試需要測試的功能。

剛剛的例子是涵蓋①～④的大範圍整合測試，不過有時候針對①～②小範圍進行整合測試也相當有效。當邊角案例的組合較複雜時，選擇執行小範圍整合測試，更能夠釐清需要測試的對象是哪個部分的功能。

● E2E測試

在既有的UI測試去加上包含了外部儲存裝置或連動的子系統的測試，在本書中稱之為「**E2E測試**」（End to End 測試）。由於會配合輸入內容來更新儲存的值，因此不僅可以測試跨越不同畫面的功能、也能用來驗證與外部的連動是否正常運作。

2-3 前端測試的目的

講解更細節的Web前端測試目的。

● 功能測試（interaction testing）

Web前端的主要開發對象是使用者所操作的UI元件。透過操作而改變狀態，提供且更新使用者想要的資訊。因此在大多數情況下，都將互動測試（interaction testing）直接當成功能測試來執行，書中跟各位分享的測試程式碼當中大部分也都是互動測試。

一聽到互動測試，大家可能腦海當中會浮現以無頭模式開啟瀏覽器（如Chromium），執行UI元件的情況。不過，使用React等函式庫所創建UI元件，其實不必用到瀏覽器也能進行互動測試。這是因為在「虛擬瀏覽器環境」中執行測試的關係，細節將會在後續的篇章提及。

沒有瀏覽器也可以做到的互動測試案例

- 按下按鈕、呼叫回呼函式
- 輸入文字後，將傳送按鈕的狀態變更為可點按
- 按下登出按鈕，畫面跳轉到登入畫面

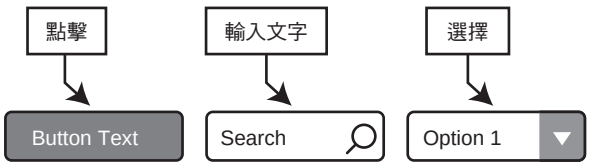

圖2-2　沒有瀏覽器也可以做到的互動測試

至於需要真實的瀏覽器才能執行的功能測試，就會運用無頭瀏覽器＋UI自動化來執行。這是由於捲動跟sessionStorage等功能在虛擬瀏覽器環境當中尚未到位的關係。因此當測試環境需要與正式環境相同、也就是說需要忠實重現瀏覽器環境時的功能測試，就會選擇這邊的做法來執行。

需要瀏覽器才能執行的互動測試案例

- 將畫面捲動到最下方，載入新的資料
- 恢復儲存在sessionStorage當中的值

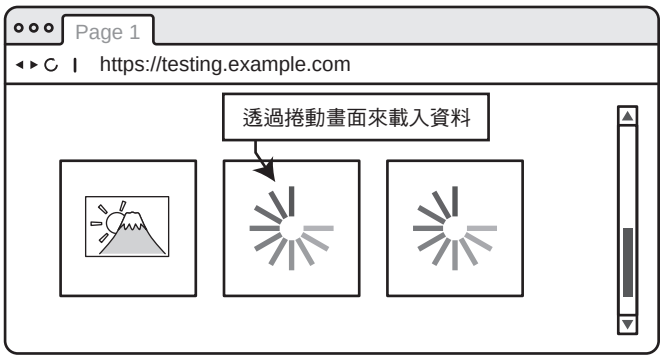

圖2-3　需要真實瀏覽器才能執行的互動測試

● 非功能性測試（accessibility testing）

無障礙性測試（accessibility testing）是非功能性測試其中之一。雖然統稱無障礙性測試，但其實測試項目五花八門。而要測試「使用鍵盤輸入的操作是否完備」跟「畫面對比度是否在視覺辨識上沒問題」時，適合的工具也不同。

不過，本書當中講解的無障礙性測試會選擇跟功能性測試裡一樣的工具，也就是「虛擬瀏覽器環境／真實瀏覽器環境」。由於非功能性測試是站在為功能性測試錦上添花的概念上來讓整體更完善，因此適合用來作為提升無障礙性的品質。

無障礙性測試案例

- 可順利將核取方塊打勾
- 顯示錯誤回應時，會渲染錯誤訊息的內容並朗讀出來
- 檢查目前所顯示的畫面是否都有符合無障礙性

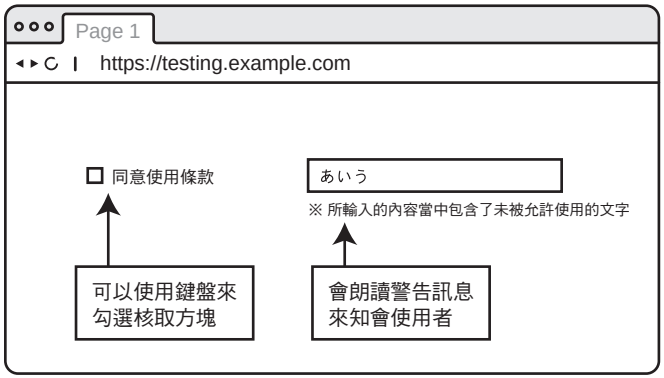

圖 2-4　無障礙性測試

● 視覺回歸測試（visual regression testing）

串接樣式表（Cascading Style Sheets，CSS）不僅有UI元件所定義的樣式，也會受到瀏覽器載入的串接樣式表的影響。截圖無頭瀏覽器所呈現的內容，並透過比對截圖來確認畫面外觀上是否有出現降級問題。也不單止是將顯示的UI元件截圖拿來互相比較，也能將使用者操作後為UI元件帶來的變化截圖進行比對。

視覺回歸測試案例

- 按鈕的外觀有無降級問題
- 點開選單欄確認有無降級問題
- 確認畫面顯示有無降級問題

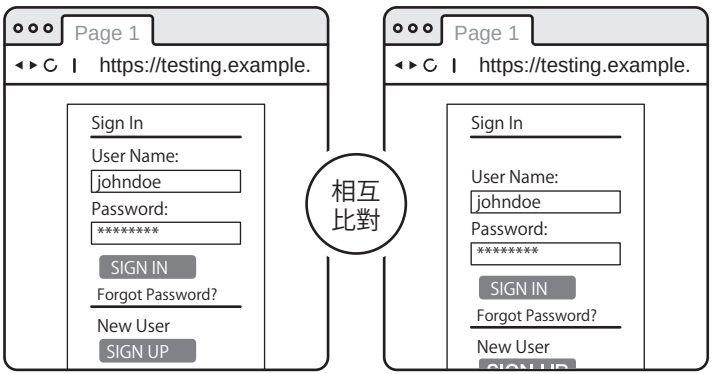

圖 2-5　視覺回歸測試

9-1 為什麼需要視覺回歸測試

本節將會講解為什麼需要視覺回歸測試（Visual Regression Test，VRT），以及為什麼需要針對每個UI元件來執行。

● 檢測樣式變化的難處

透過串接樣式表（CSS）定義的樣式，是從堆疊的屬性中計算出來的。所套用的屬性也並非透過「細膩度」或「匯入順序」而定，還會受到全域定義的影響。為此，我們雖然得要透過瀏覽器來以視覺方式確認「外觀變化」，但要判斷這些變化是否影響到所有頁面是非常困難的。修改／刪除已有的定義可能會導致降級問題。對這情況的應對方法之一是選擇「不觸碰既有的定義」，但這是消極的做法。無法進行重構的不健康CSS定義可能會淪為一種臨時性的堆疊。

單頁應用程式（SPA）的基本概念是運用小型共用UI元件來建構畫面顯示。使用UI元件建構畫面的流程有點像是堆疊積木，這稱為「元件導向」。元件導向不僅可以集中管理邏輯，還可以集中管理可能重複的樣式定義。雖然這可能為建構畫面一事帶來紀律，卻也會造成許多畫面都依賴著共用UI。換句話說，更改共用UI的樣式，就會影響到許多畫面。就算是元件導向，依然很難重構CSS。

● 外觀上的降級問題沒辦法透過快照測試防範嗎？

第5章第8節的「快照測試」是用來檢測外觀降級問題的選項之一。倘若用來決定外觀的「class」屬性出了差錯，我們都會發現外觀上的影響。可是這樣還不夠。當存在著的全域指定的CSS時，全域指定的變化就不會出現在快照測試裡。這跟「單元測試找不到的問題，在整合測試發現了」是一樣的情況。

此外，有使用CSS Modules時，CSS的指定內容也不會出現在快照測試。因此就比對HTML輸出結果的快照測試來說，真的還差得遠呢！

snapshot

```
exports[`Snapshot`] = `
<div>
  <select
    class="module"
    data-theme="dark"
    data-variant="medium"
  />
</div>
`;
```

無法驗證CSS指定內容

● 我們還有視覺回歸測試這招

最值得信任的還是實際渲染到瀏覽器並進行確認。將測試目標渲染到瀏覽器並拍攝「螢幕截圖」，聽起來還不賴，對吧！ 比較從某個時間點到另一個時間點為止的「螢幕截圖」，並以像素為單位來檢測差異。而這就是視覺回歸測試的基本概念。

視覺回歸測試是使用Chromium等瀏覽器的無頭模式來執行。無頭瀏覽器大都會跟E2E測試框架綁在一起，且一般來說E2E測試框架的標準功能當中也都具備了視覺回歸測試功能。這時候要比較的是「以頁面為單位的截圖」。使用無頭瀏覽器請求畫面顯示，當畫面切換完成後再拍攝螢幕截圖。將所有的頁面顯示畫面都拍攝好截圖，就能檢測出樣式變更前後的差異了。

相互比較樣式變更前後的圖片，就能找出哪個畫面受到了影響。可是，這樣的比較是相當粗略的。假設我們變更了公用UI「次標題」的留白部分，當這個次標題出現的位置是在螢幕上方時，次標題以下的所有範圍都將會變成是產生差異的部分。倘若此時畫面上還有「次標題以外」的變更時，要再找出哪裡有什麼不同應該是相當困難的吧（圖9-1）。

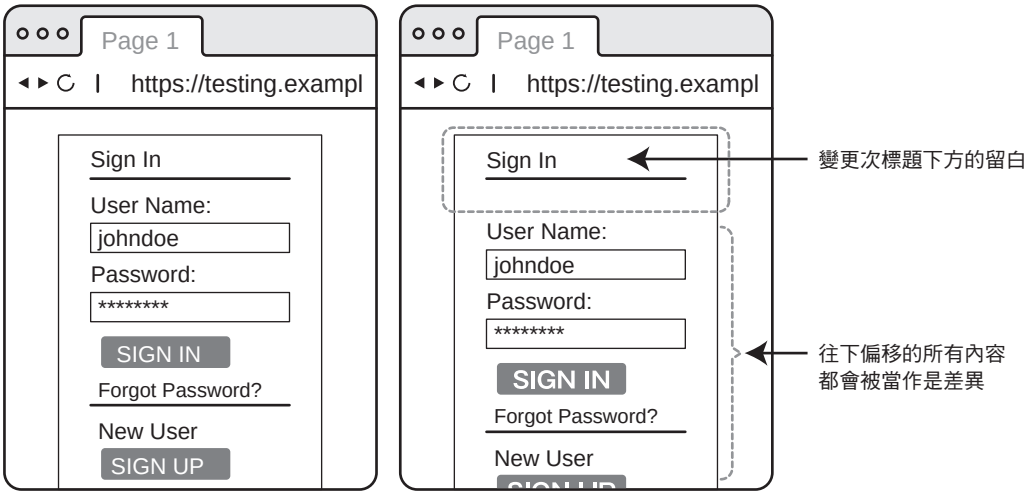

圖 9-1　沒有釐清差異的視覺回歸測試結果

能有效解決這個問題的，就是「以 UI 元件為單位」的視覺回歸測試了。當螢幕截圖是以 UI 元件為單位時，就能找出受到影響的「中密度 UI 元件」。如此一來，就算是按鈕被放置的位置以下的區域也能檢測出差異了。而支撐著視覺回歸測試的基礎正是第 8 章所介紹的「Storybook」。事先將小密度 UI 元件、中密度 UI 元件註冊為 Story，就能跨越元件總管的框架，將其作為視覺回歸測試的基礎來進行應用了。

9-2 使用 reg-cli 比較圖片

剛剛提到 Storybook 有潛力成為視覺回歸測試的平台。本書針對視覺回歸測試框架主要會使用「reg-suit」進行講解，不過就算沒使用 AWS S3 這種真實的儲存貯體（bucket），在本地也能執行視覺回歸測試。一開始我們先以 reg-suit 核心功能「reg-cli」來體驗怎麼比較圖片吧。