前言

歡迎閱讀**重構** *JavaScript*。這整本書關注於如何寫出更優秀的 JavaScript，並在探索各種程式設計風格的同時，從經典的重構技巧中獲得啟發。

為什麼要有這本書？

不管喜不喜歡，JavaScript 都不會消失的。不管你使用什麼框架、編譯至 JavaScript 的語言或是函式庫，如果你的 JavaScript 質量堪憂，那臭蟲跟效能問題依舊無法被緩解。重寫，包括轉換到最新潮的框架，代價將是驚人的昂貴且結果難以預料。臭蟲不會魔法般的消失，而會在新的情境中再次出現。而更糟的是，很多功能暫時都會無法運作。

本書對如何最好的避免這些病態的 JavaScript 寫法提出一份清晰的指南。糟糕的程式碼不該常駐世界，而改變它們也並非高昂到不合理的程度。

這本書適合給誰看？

這本書是給那些寫了一些糟糕程式碼，但對寫出優秀程式碼感興趣的程式設計師。這本書是給那些在前端或後端寫 JavaScript 的人。這本書是給那些選擇了 JavaScript 的人，也是給那些因為 JavaScript 在瀏覽器平台的壟斷性而不得不寫它的人。

如果你是個完全的新手，你也許會想先寫幾個月的糟糕程式碼。如果你沒有興趣寫出更好的程式碼，那你大概也會對本書提不起勁。但如果你沒有上述的兩個症狀，那表示你已經準備好了。

真的很有趣，在大量的努力被投入改進 JavaScript 的同時，也有許多人試圖將它搞成一團廢物。寫出優秀與糟糕的各種路線仍在持續蔓延。框架對於控制複雜性能夠做得很好，但使用它們的程式設計師將會受到限制。如果你發現你（或你的程式庫）掙扎著想脫離框架的限制（或者在某種模糊的邊緣），這本書將帶給你一些如何完成工作的新點子。

我們大多時候都不會在一個完美的程式庫上工作，當工程師們主要使用 Ruby、Python、Java 等等時，完美的 JavaScript 程式庫尤難。這本書做的就是幫助你識別出程式庫的哪些部分是不佳的，同時提供大量的改良方法。

如何使用本書？

第 1~5 章描述了 JavaScript、重構、品質、信心、測試之間的交互作用。在很多書籍中都會在最後才提及測試，但這種策略對本書要探索的程式碼不適合。測試對於信心是必要的，信心對於重構是必要的，重構對於品質是必要的：

測試→信心→重構→品質

JavaScript（及其生態圈）碰巧是個變化發生之處所，因此我們必須用初始這幾章去探索語言本身。如果對這些變化早就瞭然於胸，你也許會想要略過或快速瀏覽過這幾章。雖然我不建議，但既然這已經是你的書了，你可以對它做任何你想做的事情。如果你想要將它當作門擋、燒了取暖、或用其他方式將它犧牲掉，那就去做吧。要是你真的找到使用本書的不凡方式，請把照片或影片 email 給我。可以用 *http://evanburchard.cm/contact* 或是 Twitter 和 Github 的 @evanburchard 找到我。

如果我的書是電子檔，我還可以燒掉它或是拿它當門擋嗎？

很不幸的，辦不到。然而，由於本書採取創用 CC 授權，你可以自由的分享 HTML 版或任何 *http://refactoringjs.com* 上的其他檔案連結。

第 5 章之後的內容會變得更困難，若你略過第 1 到 5 章的話更是如此。你將需要撰寫和追蹤更多的程式碼。在第 6、7 章，我們將重構函式與物件，且我們不會閃躲 JavaScript 中的一些複雜部分。大體來說，這兩章的目標是提供一些選項來讓重構時可以不必劇烈的改變介面。透過使用這兩章提到的技巧，你將能夠將一團亂的程式庫提煉到一定水準。

第 8 章將拓展我們對於架構的視野，我們將看到那些包含（或避免）層次結構的架構。

第 9、10、11 章則講述一些能使你的程式碼品質在水準之上的技術（分別為設計模式、異步程式設計、函數式程式設計），但這些技術都會帶來更加積極（侵略性）的改變。在第 9 章的設計模式中，我們學習擴展與提取 JavaScript 物件導向的一面，並且提及了重構與物件導向之間的歷史淵源。許多 JavaScript 程式庫會在同一時間做不只一件事，而在第 10 章，我們處理這帶來的困難。在 11 章我們將透過數個函式庫進行一趟函數式程式設計之旅，這些函式庫提供了我們比標準函式庫中 `Array`（`foreach`、`map`、`reduce` 等等）更加有用的介面。

某種程度上，最後三章背離了我們進行重構的原意：在不改變**介面**的同時改變**實作細節**。但另一方面，這些介面卻又都極為有用，甚至有時不可避免。我們常會為了效能考量而不得不使用異步，或者會發現我們的程式庫早已充斥了大量或好或壞的物件導向或函數式程式設計，因而脫身不得。不管我們繼承了什麼樣的選擇跟程式碼，它們都是 JavaScript 中我們得關注的一部分，瞭解它才能改善它。如果你想要將一種與原先完全不同的範式應用到一個程式庫，那你不太可能進行本書所意謂的重構。

如果我們想要更嚴謹一點，這些章節仍然**在**他們的範式之中重構（物件導向到更好的物件導向、異步到更好的異步、函數式到更好的函數式），而若我們以寬鬆的條件來考慮程式的執行（例如，將「執行 **`node myprogram.js`**」當作輸入，「很滿意它的運行」當作輸出），那麼即使轉換範式，我們仍在進行重構。我建議你優先進行較小型、漸進式的改變，這樣易於進行測試與建立信心。

引用 William Opdyke 對重構的原始說法：

> 這個對於語意相等的定義，在輸入到輸出的映射保持不變的前提下，允許一個程式到處被修改。想像我們將所有受到重構影響的程式碼用一個圓圍住。從外部所觀察這個圓的行為是不會改變的。對於某些重構，這個圓幾乎包含了整個程式。例如，假設有一個變數引用在整個程式中幾乎無所不在，則對這個變數重新命名就影響了整個程式。而對於其他的重構，這個圓只涵蓋了較小的區域。

> 例如當某個函式被改為內聯展開[譯註]（inline expanded）時，僅有包含此函式的函式體受到了影響。在以上兩個狀況中，關鍵思想都都在於，當立於圓之外，我們觸發產生的結果（包含副作用）以及我們所參照的東西都不會改變。[1]

雖然我們能自由的控制這個「圓」的大小，**重構**這個詞仍時常被誤當作只有「改變程式碼」的意思。但如同我們在第 1 章會討論到的，並非如此。這在小規模比較清晰（我們也會在此花費最多篇幅）。先將第 8、9、10 章所展示的視為一種架構的選項，再將它們視為在這些選項中創造更佳程式碼的契機。例如，如果有人說「重構成使用異步程式碼」，很可能這個問題太過廣泛，而難以用安全而漸進的方式去完成。但如果你認為第 10 章給了你辦到這件事的力量，我也阻止不了你，現在這是你的書了。你想畫多大的圓就畫多大的圓。

如果你對任何工具或概念感到困惑，那你可能會發現 appendix 是你的好朋友。如果你正在尋找程式碼示例以及其他資訊，可以造訪本書的官網（*http://refactoringjs.com/*）。如果你偏好以 HTML 格式閱讀，你也可以在這裡找到本書的 HTML 版。

總結一下，透過本書能學到：

- 重構
- 測試
- JavaScript
- 重構與測試 JavaScript
- 一些 JavaScript 範式
- 在這些 JavaScript 範式中重構與測試

譯註 將函式改為內聯會使該函式在被編譯的時候，會被直接嵌入呼叫它的函式之中，由此來省去函式呼叫的成本，由此來優化效能。在常見的程式語言中，C++ 便具有 inline 關鍵字可顯式的要求編譯器內聯。

1 WilliaM Opdyke, “Refactoring Object-Oriented Frameworks”（1992, 伊利諾大學厄巴納 - 香檳分校, 博士論文）, 40

本書的一些用語[譯註 1]

App、應用程式、程式

本書的某些用語並不精確。App、應用程式、程式以及網站這幾個詞通常都能夠互相替換而沒有問題。本書描述改進 JavaScript 品質的通用方法，所以當你感到疑惑時請不用太拘泥於文字。或許你在寫一個函式庫或框架呢？本書的技巧在各種狀況都能運作。

字詞與圖片的通用性

本書的某些字詞指涉未必適用於所有人。我試著平衡**他**和**她**的使用，但這並非每個人都能接受。雖然我更偏好於使用 singular they[譯註 2]，但這種用法並不在出版商的指導方針中。

此外，我瞭解到（太遲了）我很依賴圖片（尤其在第 5 章），這也許會對視力困難的讀者造成障礙，如果你認為自己因為這種理由而漏掉了任何內容，歡迎與我聯繫。

使用者

還有一個本書會用到的詞是我很討厭的，那就是**使用者**（*users*）。它很不精確，並且隔離開了創造者（開發者 / 設計者）與消費者（使用者）。更加精確的字詞通常是特別拿來描述某個問題領域，否則我們就只能勉強使用「人們」、「使用程式 / 網站的人們」這種術語。如果並沒有比**人**或**使用者**（甚至是**消費者**）更具體的術語，這也許暗示著這個商業模型就只是單單把人當作資料在賣，但這是另一個話題了。

我想說的是，**使用者**這個詞在本書中表達一個人們所熟知的概念：一個使用程式 / 網站的人。同時，目前尚未有更好更精確的詞來代替**使用者體驗**（*user experience*）（UX）或**使用者介面**（*user interface*）（UI）。為了避免在其他地方再多說明或使用不標準術語形容常見的抽象概念，我特在此說明這些。

在任何狀況之下，我都完全同意「資料界的李奧納多·達文西」Edward Tuffe 所說的這段格言：

> 只有兩種產業會視他們的客戶為使用者：毒品和軟體

[譯註 1] 這些英文字詞經過翻譯之後也許就沒有這類問題

[譯註 2] 將 they 這個複數詞作為單數來使用，中文讀者可不必理會這種用法

有一種叫做「倫理化設計（ethical design）」的運動有望協助企業在某種程度上不再使用這個詞（以及其他由此而生的不智作為）。

第三方函式庫與社群

雖然我非常努力要用最好的工具來展示重構及測試 JavaScript 的基礎，但也許有時候你會發現某些工具無法正常運作。好消息是 JavaScript 豐富的生態系提供了多種選擇。我偏好使用簡單靈活、原子性（atomic）的工具，而你可能自有偏好。本書不會探討大型的框架，因為它們傾向帶有自己的一套工具鏈生態系（通常變化多端）。當然當你準備好的時候，我會推薦框架，但這些框架要與底下的程式語言所提供的設施組合才能發揮出最大的威力，而我相信這本書能為你打下堅實的基礎。

API、介面、實作、「客戶端程式碼（client code）」

這可能有點模糊，但有件事我想要再強調，那就是層次結構不該根據物件，而該根據設計優良的程式庫的介面。如果只是個簡單腳本，我們期待它一路跑到底。但當程式庫漸漸成熟（透過設計，而非混亂的大屠殺），我們期待它會建立三個層次（雖然這顯然在更複雜的程式庫中被延伸了）。

第一層——不顯於程式庫表面的深層程式碼——就是本書指稱的**實作**（*implementation*）。對於重構來說，最需要分辨的就是實作與它下一層之間的差異。第二層通常被稱為介面（interface）或 API，它們是「公開的」函式或物件，當程式庫被用作一個模組時，我們預期我們能與它互動。第三層常被稱為**客戶端程式碼**或**呼叫他人的程式碼**（*calling code*），它指稱被寫來與介面層互動的程式碼。當人們使用模組時就會寫這種程式碼，撰寫測試時，我們也寫這種程式碼來測試介面層。

淺談結構

在本書中，所有程式一開始都幾乎沒有結構，而我們的主要目標便是清楚地切開這三個層次（不論是透過何種程式設計典範，如物件導向或函數式編程）。如此一來，能夠增加程式碼的可測性與可攜性。有些框架會提供自己的結構，如果你時常依賴這些框架，那麼你可能會對本書的方法有些陌生。

輸入（非區域與自由變數，Nonlocal and Free Variables）

我們會在本書中（特別是在第 5 章）區分以下三種輸入：

- 顯式輸入（傳入函式中的引數）
- 隱式輸入（也就是 this，指向當前上下文的物件、函式或類別）
- 非區域輸入（函式或物件中其他地方所定義的變數）

這裡有兩件事要注意。首先，在作用域中創建的區域變數或常數並不算是「輸入」；第二，雖然**非區域輸入**在此處是一個精確的字眼，但更常見的說法是「自由變數」。然而這個說法有失精確，畢竟非區域輸入也有可能是常數而非變數。類似地，有人會用**約束變數**（*bound variable*）來指稱我們所謂的**顯式輸入**，有時甚至用來指稱**隱式輸入**。

本書編排慣例

本書使用了以下排版慣例：

斜體（*Italic*）

用來表示新術語、URL、電郵信箱、檔案名稱與附加檔名，有時也用來表現強調或對照的語氣。中文用楷體表示。

定寬字（`Constant width`）

用來表示樣式碼或在段落中表示如變數或函式名稱、資料庫、資料型別、環境變數、陳述與關鍵字等程式元素。

定寬粗體字（**`Constant width bold`**）

用來表示應由使用者輸人的指令或其他文本。

定寬斜體字機（*`Constant width italic`*）

用來表示應由使用者提供或取決於情境的詞。

用來表示技巧或建議。

第一章

什麼是重構？

重構並非改變程式碼。

好啦，其實就是，但不僅如此。重構確實是在改變程式碼，但是它有個重要的限制使得僅僅使用「改變程式碼」來描述它顯得不夠精確：重構並沒有改變程式碼的**行為**。此時你心中應該有兩個疑問：

- 如何確保行為不會改變？
- 既然程式碼的行為不變，那我們要將修改程式碼的重點放在哪？

在本章，我們會去追尋這兩個問題的答案。我不會完整的講述 JavaScript 的歷史，畢竟那已經廣泛的流傳於網路上了。

如何確保行為不會改變？

如果沒有範圍的話，這個問題的答案極為困難。但幸運的是，有許多行為在重構時我們並不太在乎。接著我們將討論它們：

- 實作細節
- 不明確和未測試的行為
- 效能

而這個問題簡短的答案是，使用單元測試與版本控制。

由 Willan Opdyke（他的論文是重構理論的基石）（*http://www.ai.univparis8.fr/~lysop/opdyke-thesis.pdf*）所擁護的另一種方法，強調使用自動化工具來改變程式碼並在改變程

式碼之前確保安全。當程式碼的更改是否「安全」是由工具來決定時，人為所能做的更改就受限於這些工具的功能。因此職業開發者們也許會覺得，移除了人類能進行的彈性操作之後，改變程式碼受到了很多限制。

撰寫工具來完全含括 Martin Fowler 在他的作品《**重構：改善既有程式的設計**（*Refactoring: Improving the Design of Existing Code*）》中所提出的所有原則是很困難的。而 JavaScript 動態、多範式又存在一堆變種（參見第 2 章），而採用這些工具將會使得許多原本可能用來重構 JavaScript 的方法變得無法使用。

Fowler 的方法降低了自動化的重要性，同時強調重構的「機制」：以能夠最小化不安全狀態的步驟來改動程式碼。

如果我們遵照 Opdykian 的自動化方法，那些工具將會處處扯我們的後腿。我們同樣也不打算遵照 Fowler's 的機制（一步接著一步的流程）。原因是，在我們透過重構來對程式碼取得信心的過程中，若有測試支撐，檢驗成功會很直觀。而當我們重構出問題時，版本控制（我們將使用 Git）讓我們能輕易回到過去的狀態。

警告！使用版本控制！

只要你無法輕易的回到過去、安全的版本，任何形式的「改變程式碼」都會為你的程式庫帶來危險。如果你打算要重構的程式庫目前沒有使用版本控制來備份它，那麼請立刻放下這本書，直到你對它版本控制前都不要把它打開。

如果你目前沒有在使用版本控制，那你可能會想要使用 Git（*http://git-scm.com/*），並且使用 Github（*http://github.com*）來幫你備份。

誠然，本書的做法也許乍看之下，相較於過去的使用自動化與機制的做法，顯得有些無拘無束和不夠嚴謹。然而，本書採用的流程 ── 持續「紅」（測試失敗時）、「綠」（通過測試時）、「重構」的循環，並注意在事情不對勁時回到過去狀態 ── 與注重質量的業界團隊所使用的流程雷同。也許以後自動化重構工具能像支援本書所提到的重構原則般地支援 Fowler 提出的每一條原則，不過我不認為短期內這件事會發生。

我們在此處的目標是將 JavaScript 開發者拉出泥淖。雖然自動化與機械化這個過程十分誘人，但這些巨人們（Opdyke、Fowler、Johnson 等等）（本書正是站在他們的肩膀上）最重要的貢獻，是讓我們能以一種全新的心態去看待安全的改善程式碼這回事。

為什麼我們不在意實作細節？

假設現在我們有一個簡單的函式，它會將傳入參數乘 2 之後返回：

```
function byTwo(number){
  return number * 2;
}
```

透過小小的改變，我們可以將它寫成：

```
function byTwo(number){
  return number << 1;
}
```

這兩種作法都能在大部分的應用中運作順利。我們對於 byTwo 函式的任何測試，基本上就是一個輸出為輸入兩倍的映射。我們通常只在乎結果，而不在乎過程中到底使用了 * 還是 <<。我們將它視為**實作細節**。雖然你也可以將它視作**行為**，但如果我們只在意函式的輸入輸出的話，那這只是個不重要的行為罷了。

如果我們恰巧為了某些因素使用了第二版的 byTwo，我們可能會發現當參數太大時，這個函式的行為就會不如預期（試試看傳入一兆：`1000000000000 << 1`）。這是否代表我們得在意這個實作細節了嗎？

不。我們在意的是這個輸出壞了。這代表我們的測試集必須比我們原先設想的更加完善。如此我們才可以將此函式隨意替換成滿足整個測試集的函式；無論它是 `return number * 2` 或 `return number + number` 都無所謂。

我們改變了我們的實作細節，但將數字乘以 2 這件事才是我們真正在乎的。我們所在乎的，就是我們所要測試的（無論是手動或自動）。測試某些特殊性質不僅在很多狀況下都不必要，還會創造出我們難以自由重構的程式庫。

測試 JavaScript 本身

若你去測試極端特定的實作細節，某種程度上就不再是測試程式碼，而是在測試環境本身。現在我們先簡單的使用 node 主控台或是將它存成檔案再執行 **node *file_name.js***，之後會再嚴正的討論測試[譯註]。

譯註　包含正式的工具、環境。

假如你現在測試一些像這樣的東西：

```
assert = require('assert');
assert(2 + 2 === 4);
```

順道一提，如果你使用 node 主控台的話，第一行會噴出一堆可怕的東西。不過別擔心；它只是秀出它載入了什麼。其實，node 主控台原本就預載好 assert 了，所以如果你並非存成 *.js* 檔來執行，你可以不用執行第一行。第二行只會回報 undefined，或許看起來有點怪，但這是正常現象。試著像 assert(3 === 2) 這樣對非真值下斷言，你就會看到一個錯誤被拋出了。

如果你下了上述那樣的斷言，那你正在測試 JavaScript 本身：它的數值、+ 運算子、=== 運算子。類似地，你可能更常發現自己在測試函式庫：

```
_ = require('underscore');
assert(.first([3, 2]) === 3);
```
譯註

這樣是在測試 underscore 這個函式庫是否運作得跟預期一樣，如果你正在探索新的函式庫或不熟悉的 JavaScript 特性時，測試低階實作細節是很有用的，但通常這些套件本身擁有的測試就綽綽有餘了（每個優秀的函式庫都有自己的測試）。然而，對此我有兩個附加聲明。首先，為了確認某些函式或函式庫已經能在你的環境中正常運作，「理智的測試」是需要的，但是當環境趨於穩定，這些測試就可以拋棄了。第二，若你正在撰寫的測試是為了幫助你自己對程式碼更有信心（之後還會再詳述），那測試一個函式庫的行為確實是個了解程式碼的實戰方法。

為什麼我們不在意不明確和未測試的行為？

我們努力描述並測試程式碼的程度，正展現了我們對其行為的關注。如果一段程式碼沒有測試、一個用以執行它的手動程序，或者至少一個它該如何運作的描述，那就代表它基本上是無法驗證的。

譯註 原文為 assert(.first([3, 2] === 3)，但這是個不合語法的句子。

假設以下的程式碼不帶有任何測試、文件或描述它的業務流程：

```
function doesThings(args, callback){
  doesOtherThings(args);
  doesOtherOtherThings(args, callback);
  return 5;
};
```

我們會關心它的行為是否改變嗎？事實上，會的！這個函式可以一口氣處理一堆事。這僅僅因為我們對它的無知並不會使它變得不重要，反而會使它變得更加危險。

但以重構的語境來說，我們並不在意這個行為是否改變，**因為我們根本不會對此重構**。我們不會想要去改變缺乏測試（或至少有文件說明它執行方式）的程式碼，**我們無法重構它**，因為我們無法驗證它的行為是否改變。本書後面會提及該如何創建「描述測試（characterization tests）」來處理沒有測試的程式碼。

不僅在處理前人所遺留的程式碼時會發生這個狀況。重構新產出但沒有（自動或手動的）測試的程式碼，也是不可能的。

在撰寫完測試之前，關於重構的話題將會是如何進行？

「我重構了用電子郵件與使用名稱來登錄這個功能。」

「不，你沒有。」

「我重構了 ...」

「不，你沒有。」

「我們得在添加測試之前重構。」

「不。」

「重構 ...」

「不。」

「重 ...」

「不。」

為什麼我們不在意效能？

我們不會在重構一開始時就去在意效能。就像前幾個小節中用來對輸入乘 2 的函式，我們只在意它是否對於輸入有正確的輸出。通常我們利用最常見的工具就足夠在第一次嘗試時得到**夠好**的效能了。在此我們所需遵循的格言是「程式是寫給人看的」。

對於第一次實作而言，「夠好」代表我們可以在合理的時間內，去確認輸入確實會產生正確的輸出。如果一個實作實在太慢並讓我們難以做到這點，那我們就需要改變實作了。但在那之前，我們應該有合理的測試而能夠驗證輸入輸出。一旦有了適當的測試，我們就有足夠的信心去重構程式碼與改變實作。如果沒有適當的測試，就會使我們真正在乎的行為（輸入輸出）暴露於危險之中。

雖然這算是一種**非功能性**測試，並且通常不是重構的重點，我們依然能將效能（以及程式碼中其他「非功能性」的特徵，例如可用性）變成可否證的[譯註]來強調這個特徵。換句話說，我們可以為效能撰寫測試。

效能在程式庫的非功能性方面通常較受重視，因為人們相對容易做出一個標準來對效能進行評估。透過「基準化測試（benchmarking）」程式碼，我們能夠創建一個當效能太差時會回報失敗、效能足夠會回報成功的測試。此時我們的測試就是將執行函式（或其他程序）作為「輸入」，將完成它所需的時間（和其他資源）作為「輸出」。

然而，在效能已經有一個這種形式的可驗證測試架構之前，效能並不會被我們視為需要考慮是否改變的「行為」。如果我們有適當的功能性測試，在我們決定了某個效能標準之前，我們都可以自由的調整實作。決定效能標準之後，我們的測試集就能包含效能特徵了。

所以到頭來，在我們決定並創建效能（及其他非功能性面）的標準之前，它都不是我們首要關心的重點。

譯註　此處應該是指，我們能去確立一種方式（例如以時鐘計時）來決定（否定）某段程式碼的效能是否有問題。

JavaScript 在效能方面的特殊困難

對於某些形態的 JavaScript 而言（參見第 2 章），要在不改變效能的情況下去改變（不安全的）或重構（安全的）程式碼是完全不可能的。添加幾行沒有最小化過的前端程式碼，就會增加大量下載和處理的時間。不計其數的的建置工具、編譯器、實作，將會根據你如何組織程式碼而玩起各種不同的小把戲。

這些事情可能對你的程式很重要，但請注意，重構對於「不改變行為」的承諾絕對不包括它們。難以捉摸的效能問題應該被移出來額外考量！

如果程式碼的行為不變，那修改程式碼的重點是什麼？

重點在於，在維持行為的情況下改善品質。並不是說修正臭蟲和開發新功能（寫新程式碼）不重要。事實上，這兩件事與業務目標密不可分，而且比起程式庫的品質，專案 / 產品經理更在意它們。然而，它們都會改變程式碼，因此與重構截然不同。

現在我們要闡述兩個重點。首先，在「把事情搞定」的情境中，品質為什麼重要？第二，什麼是品質，以及重構如何改善品質？

在品質與把事情搞定之間取得平衡

似乎每個人每個專案的運作方式都可以用一個簡單的光譜來描述，這個光譜的一端是品質，另一端是把事情搞定。在一端我們能得到一個不能做任何事的「美麗」程式庫；而在另一端，我們則能得到一個試圖支援諸多功能但卻充滿臭蟲與半完成功能的程式庫。

技術債這個名詞在近二十年間開始廣泛流行。用類似財務的行話來讓非程式設計師易於理解，並且以更細緻的詞語來描述一個任務可以多快完成或是應該多快完成。[譯註]

先前提到的光譜某種程度上是很精確的。做小專案的時候，我們大概還能接受一些可見、可描述的技術債；但當專案越長越大時，品質也變得越來越重要了。

譯註　通常借錢可以讓事情更快完成，在此指我們借了這些技術債所以能夠加快任務進行。

什麼是品質？它又怎麼牽扯到重構？

是什麼造就了程式碼的品質？對此人們早有大量的研究，有些以一組原則來描述。

- SOLID：單一職責、開閉原則、Liskov 替換、介面隔離以及依賴反轉（Single responsibility, open/closed, Liskov substitution, interface segregation, and dependency inversion）
- DRY：別重複你自己（Don't repeat yourself）
- KISS：讓事情簡單些、傻一些（Keep it simple, stupid）
- GRASP：通用職責分配軟體模式（General responsibility assignment software patterns）
- YAGNI：你不需要它（Ya ain't gonna need it）

還有其他的度量方法，例如程式碼 / 測試覆蓋率、複雜度、參數數量以及檔案長度。並且存在有各種能夠監控語法錯誤與程式碼風格的工具。有些語言甚至更進一步，使得人們根本無法寫出某些風格的程式碼。

沒有一個絕對基準能夠衡量品質。本書認為，運作正常又易於擴展的程式碼就是優良的程式碼。有了這個定義之後，我們的戰術就是為程式碼撰寫測試，並且寫出易於測試的程式碼。在此我謙遜的提出「EVAN 程式碼品質原則」。

- 提取函式與模組以簡化介面
- 透過測試驗證程式碼行為
- 盡可能避免不純的函式
- 良好命名變數與函式名稱

別感到拘束，請你也用自己的名字創造一個軟體品質原則吧！

將可讀性作為品質

人們也常將「可讀性」視為品質的重大指標，但這是個相當難以判定的特質。每個人的經驗與所接觸到的觀念都各有不同。程式新手，甚至是剛學某種新範式的程式老手，都有可能在與抽象概念苦苦搏鬥。

若由此推斷，只有最簡單的抽象才能成為高品質程式碼的一部分。

> 在實戰中，開發團隊應該在避免使用小眾而令人困惑的功能，但同時又得投注時間去教導新手那些合理、正確運用的抽象概念。

在重構的語境中，品質就是目標。

由於你要解決的問題來自於許多不同的範疇，而你所能選擇使用的工具又是那麼的多樣，第一次的猜測很難是最優的。要求你只寫出最棒的解答（並且不再遇到它）幾乎不可能。

透過重構，你可以先為你的程式碼與測試做出最佳的猜測（如果你採用測試驅動開發（test-driven development），TDD，不會照著先寫程式碼再寫測試的順序；見第 4 章）。然後，你的測試會保證你在修改**細節**的同時，整個程式碼的行為（測試的輸入與輸出，也就是介面）是不變的。用這個流程所提供給你的自由，你可以改變你的程式碼，使之達到任何程度的品質（也許包含效能或其他非功能性特徵）以及你認為合適的抽象。除了能夠漸進的改善程式碼品質之外，練習重構還有個額外的好處，那就是你能夠在改善不良程式碼的過程中學習，下一次犯的錯會更少。

所以我們採用重構來安全地改變程式碼（而非行為）以增進品質。你當然可能會想知道實際做起來會是什麼樣子。這正是本書在後面會說明的，但在此之前我們還有幾章背景知識需要說明。

第 2 章提供了 JavaScript 本身的背景知識。第 3、4 章會有一番關於測試的辯證，緊接著是一套測試的實戰方法，這套方法是在撰寫具信心的程式碼並快速迭代的過程中很自然產生的，我們並非獨斷的堅持測試是不可質疑地美好的。第 5 章則深入探討了品質，以及被稱為 *trellus* 圖（更多資料請見 trell.us）的一套函式視覺化方法。

在第 6、7 章中，我們研究通用的重構技巧。接著在第 8 章研究如何重構有著層次結構的物件導向程式碼，以及在第 9 章中研究各種模式。在第 10 章探討重構異步，第 11 章中則以函數式程式設計來重構。

在 JavaScript 這種特性多樣的語言中，要敲定何為品質非常困難。但透過這幾章介紹的各種技巧，你將有很多選項去選擇。

視重構為探索手段

雖然本書的大部分時候都將重構作為一個改進程式碼的過程，但這並非它唯一的目的。重構也能幫助對程式碼建立信心，以及幫助熟悉你正在工作的程式碼。

約束有其好處，而缺乏約束正是 JavaScript 在許多方面難以學習的原因。然而敬畏會滋生不必要的約束。我曾經看過一次班・弗茲的表演，那場秀以他把椅子砸向鋼琴作結。誰會去破壞一台傳統上受尊敬（而且很貴）的鋼琴呢？一個在控制中的人。一個比他的工具還來得重要的人。

你比你的程式碼還來得重要。破壞它，把它通通刪了。去改變你想改變的一切。弗茲有錢能夠負擔得起新買一台鋼琴。而你有版本控制。你想對你的編輯器幹嘛就幹嘛，而且你在修改的是有史以來最便宜、最有彈性且最牢固的介質。

當然，在你的程式碼適合你時重構它。我猜這會常常發生。但如果你想刪掉某些你不喜歡的東西，或僅僅想破壞它或拆開它來看它究竟是如何運作的，那就去做吧！你能在撰寫測試與不斷的小幅改變中學到許多，但這不會總是最簡單或最自由最有趣的探索路徑。

什麼是以及什麼不是重構？

在我們結束本章前，我們再一次區別重構與看起來很像重構的東西。這是一份清單，清單中的事項都不是重構。相反地，它們都創建新的程式碼與特性。

- 給一個計算機應用程式新增開平方的功能
- 從頭製作一個應用程式
- 用新框架重建已有的應用程式
- 給一個應用程式增加一個新套件
- 以姓名而非名字來稱呼使用者
- 本地化
- 最佳化效能
- 改變程式碼以使用不同的介面（例如，同步到異步、回調到 promises）

不是重構的東西不僅只如此，對於現有的程式碼，任何介面（行為）的改變都**該**使測試失敗。否則，測試率大概是很糟糕的。而底層實作細節的改變**不該**使測試失敗。

此外，去改動任何沒有合適測試（或至少如何手動測試的文件）的程式碼都無法確保不改變它的行為，因此這也不算是重構，只能說是改變程式碼而已。

總結

希望本章已經揭露了何謂重構，至少已經提供了一些例子說明什麼不是重構。

如果我們只要抽象地重構 JavaScript 或僅僅看著其他語言一些能啟發靈感的源碼範例，就能定義並完成具有優秀品質的程式碼，那我們的 JavaScript 程式庫就不會困擾於當今這種「不是壞就是新」的二分法了，程式庫總是沒有被好好維護，最終它們被工具 A 或框架 B 重寫：這是一種代價高昂又充滿風險的做法。

框架無法拯救我們的品質問題。jQuery 沒能拯救我們，ESNext、Ramda、Sanctuary、Immutable.js、React、Elm 或任何下一個被發明的東西，它們通通做不到。縮減與組織程式碼很有用，但透過本書，你將能建立一套改善的流程，不再掉入「為糟糕的品質而痛苦→投入未知大量的時間以『這個月最潮的框架』重新建構→為這個框架而更加痛苦→再重建」的無限循環中。