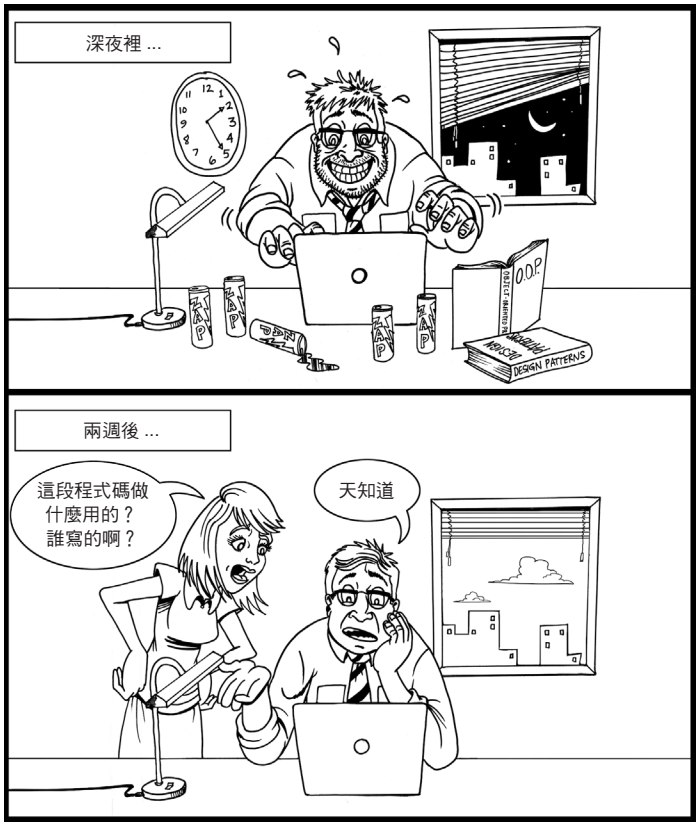


前言

我們曾任職於十分成功的軟體公司，和許多傑出工程師一同工作，但仍然遇到許多需要改善的程式碼，事實上，我們看過許多醜陋的程式碼，讀者可能也是如此。

漂亮的程式碼有啟發性，能快速地讓讀者瞭解情況，使用起來也十分有趣，還能激發程式設計師改善所寫的程式碼。

本書目標是協助讀者改善程式碼，「程式碼」是指讀者從編輯器中看到的每一行程式而非專案的整體架構，也不是指設計模式（design pattern）。這些雖然十分重要，但在我們的經驗裡，程式設計師大多數的時間是花費在「基本」工作上：變數命名、迴圈以及各式各樣函數層級的問題。這些大多需要閱讀與編輯原有的程式碼，希望本書能對讀者日常的程式設計工作有所助益，願意推薦給團隊其他成員。

本書內容

本書主題是介紹如何寫出有高度可讀性的程式碼，主要概念是**程式碼必需易於理解**，更精確地說，撰寫程式碼時應將他人理解程式碼所需的時間縮到最短。

本書使用許多不同程式語言的範例解釋這個概念，包含了 C++、Python、JavaScript 以及 Java；避開了程式語言的進階特性，即使不熟悉這些程式語言的讀者，也能夠輕易理解內容（畢竟，可讀性概念大多與程式語言本身無關）。

本書各章節由程式設計的不同層面說明如何讓程式「更易於理解」，共分為四個部份；

表層改善

名稱、註解與美學：使用在每一行程式碼的簡單技巧。

簡化迴圈與邏輯

透過改善程式的迴圈、邏輯與變數，提高可讀性的方法。

重新組織程式碼

組織大區塊程式以及從函數層面解決問題的高階作法。

精選主題

將「易於理解」原則應用在測試以及大型資料結構的程式。

如何閱讀本書

本書內容儘可能有趣，以便能夠輕鬆地閱讀，希望大多數讀者能在一、兩個星期內讀完。

章節順序依「難易度」安排：由基本主題開始，較進階的主題安排在後，每章內容都十分完整能夠獨立閱讀，也可以依自身喜好瀏覽。

使用範例程式碼

本書的目的是協助讀者完成工作。一般而言，可以在程式與文件使用本書的範例程式碼。除非重新製作並散佈大部分程式碼，否則不需要與我們連繫以取得授權。例如，開發程式時使用書中一些範例程式碼，並不需要取得授權。然而販賣或散佈歐萊禮書籍的範例程式光碟，就需要取得授權。為了回答問題，引用這本書的內容或程式碼，並不需要取得授權。把書中大量範例放到你自己的產品文件中，就需要取得授權。

我們會很感謝你在使用範例程式碼時註明出處，但這並非必要。出處說明通常包括書名、作者、出版社以及 ISBN 書號。例如：「易讀程式之藝術，Dustin Boswell 與 Trevor Foucher 著。版權為2012歐萊禮媒體公司所有，ISBN 書號 978-0-596-80229-5。」

致謝

感謝貢獻時間審查完整的草稿的同事們，包含 Alan Davidson、Josh Ehrlich、Rob Konigsberg、Archie Russel、Gabe W. 以及 Asaph Zemach，本書中的錯誤都是他們的問題（開玩笑的）。

感謝對本書各個版本提供意見的審稿人，包含 Michael Hunger、George Heineman 以及 Chuck Hudson。

John Blackburn、Tim Dasilva、Dennis Geels、Steve Gerding、Chris Harris、Josh Hyman、

Joel Ingram、erik Mavrinac、Greg miller、Anatole Paine 以及 Nick White 也提供了許多想法與意見，感謝在歐萊禮 OFPS 系統中審查本書草稿的線上讀者們。

感謝歐萊禮團隊無比的耐心與支持，特別是 Mary Tresele（編輯）、Teresa Elsey（製作編輯）、Nancy Kotary（文字編輯）、Rob Romano（插圖作者）、Jessica Hosman（工具協助）以及 Abby Fox（工具協具），以及實現我們對漫畫瘋狂想法的漫畫家 Dave Allred。

最後，我們要感謝 Melissa 與 Suzanne，感謝你們一路的陪伴與忍受持續不斷有關程式設計的談話。

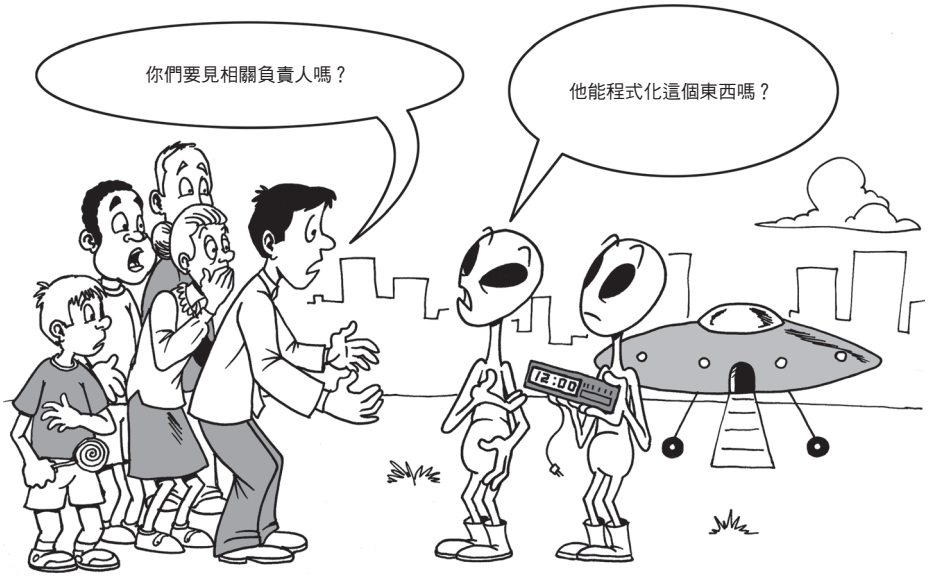
第一部份

表層改善

可讀性的旅程從「表層」改善開始：選擇好的名稱、撰寫好的註解並以簡潔的方式編排程式。這些改善很容易採用，能夠「立刻」使用，不需要重構程式碼或改變程式執行方式。也可以逐步進行，不用一次投入大量時間。

這些主題十分重要，因為會**影響程式庫中的每行程式碼**，雖然個別變動都不大，累積起來卻能獲得大幅改善。如果程式有傑出的名稱、良好的註解，也能精確使用空白，就會大大提高了程式碼的可讀性。

當然，在表層之下，仍有許多與可讀性有關的改善（本書稍後會提到這些部份），但這部份所提到的改善只需要一些些功夫，就有廣泛的應用，十分值得一提。



第一章

程式碼應該易於理解

過去五年，作者們收集了數百個「不良程式碼」(大多數是自己寫的)，分析造成不良的原因，找出改善的原則與技巧，最後發現所有的原則都回溯到相同的概念。

> **重要概念**
>
> 程式碼應該易於理解。

作者們相信這是寫程式時最重要的原則，接下來的內容，會將這個原則應用在程式設計師每天工作的各個層面，首先詳細說明這個原則以及最為重要的原因。

「更好」的意義？

大多數程式設計師（包含作者在內）根據直覺選擇呈現程式碼的方式，大家都知道（即使下面範例中的程式碼有完全相同的行為），但這種寫法：

```
for (Node * node = list->head; node != NULL; node = node->next)
    Print(node->data);
```

比起以下的寫法來得好：

```
Node* node = list->head;
if (node == NULL) return;

while (node->next != NULL) {
    Print(node->data);
    node = node->next;
}
if (node != NULL) Print(node->data);
```

但很多時候，決定何者要好並不容易，例如這段程式：

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

是否比這種寫法來得好：

```
if (exponent >= 0) {
    return mantissa * (1<< exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

第一個版本比較簡潔，第二個版本則較容易瞭解，那個條件比較重要？一般來說，該如何決定程式碼的呈現方式？

可讀性基本定理

看了許許多多的範例之後，得到的結論是，存在一個最為重要的可讀性指標，重要到能夠將之稱為「可讀性基本定理」。

> **重要概念**
>
> 撰寫程式時應該將讀者理解所需的時間降到最短。

這是什麼意思？很簡單，找個一般的能力的同事，測量他們理解程式碼需要的時間，這就是撰寫程式時希望縮短的理論值。

「理解」的要求非常高，**完全理解**程式碼，應該要能夠修改、找出臭蟲並知道與程式其他部份互動的方式。

現在，讀者可能認為，「**誰會在乎其他人懂不懂？我是唯一一個使用這些程式碼的人！**」即使單人專案也值得追求這個目標，「其他人」可能是六個月後的自己，到時候對程式碼不再熟悉。永遠無法確定 —— 可能有其他人加入專案，或是「被丟棄的程式碼」可能被用在其他專案當中。

比較短的程式都比較好嗎？

一般來說，解決問題所需的程式碼愈短愈好（參看第十三章，**撰寫較少程式碼**），瞭解一個由 2000 行程式碼的類別所需時間可能比由 5000 行程式碼的類別來得短。

但較少的程式碼並非總是比較好！很多時候像這樣的單行表示式：

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied())
```

比起兩行的寫法需要更多時間理解：

```
bucket = FindBucket(key)
if (bucket != NULL) assert(!bucket->IsOccupied());
```

同樣的，註解雖然會在檔案中「加入更多程式」，卻能讓人更快速的理解程式碼：

```
// hash = (65599 * hash) +c 的快速版
hash = (hash << 6) + (hash << 16) - hash + c;
```

雖然減少程式碼數量是個很好的目標，但縮短理解時間更加重要。

最短理解時間是否與其他目標衝突？

讀者可能會想「其他的條件怎麼辦？像是程式效率、良好的架構或是可測試性等等？這些條件難道不會和讓程式易於瞭解的原則衝突嗎？」

作者發現其他目標並不會受到太大的影響，即使在高度最佳化的程式碼中，仍然有辦法讓程式碼具有高度可讀性；實際上讓程式碼易於瞭解常常也會產生良好的架構且易於測試。

本書接下來將介紹如何在各種情況應用「易於閱讀」，切記，難以決定時，可讀性基本定理優於本書所提的其他任何原則。同時，某些程式設計師無法忍受任何不夠完美的程式碼，這時必須停下腳步，想想**程式碼是否易於理解**？」如果已經如此，也許可以放心處理其他的程式碼。

困難所在

的確，時時考慮某個虛擬讀者是否能夠理解程式碼需要花費不少額外的功夫，需要將大腦切換到撰寫程式時沒有用到的部份才行。

但如果（和作者一樣）選擇了這個目標，可以確定讀者將會成為更好的程式設計師，產生更少的臭蟲，對成品更有認同感，同時能產出其他人樂於使用的程式碼，就讓我們開始吧！