簡介

你**從沒想過**會成為管理者吧。你就像我所認識的大部分軟體開發者一樣，只想安安靜靜坐下來寫程式，這樣的你肯定比較開心自在。但你確實是公司最好的開發者；**不幸**的是，公司原本的主管 Nigel，最近因為高空彈跳發生了意外、他的筆記型電腦好像也摔爛了，這時候身為公司明日之星的你獲得升遷，好像也是很自然的事。

所以你現在有了自己的辦公室（終於不用再與萬年實習生共用小隔間了）；你開始要填寫半年一次的績效評估表（不能再爽歪歪整天盯著螢幕，任它繼續摧殘你的視力），而且你還要花時間應付那些傲嬌的程式設計師，還有一些只會吹牛皮的推銷員，以及一群創意四射的「UI 設計師」（應 Pete 要求，請叫他們「平面設計師」），搞定各種古怪的需求，比如怎麼讓「確定」、「取消」按鈕更加**閃閃動人**——我也很好奇，「閃閃動人」的 RGB 值究竟是多少呢？另外，你還要處理資深副總的一堆蠢問題——「我們幹嘛不用 Java 來取代 Oracle 呢？聽說它整合得比較好啊。」（看來他對軟體所知的一切，全都是從達美航空飛機上的雜誌裡學來的！）

歡迎進入管理層！你猜怎麼樣？「軟體專案管理」與「程式設計」**根本就是兩種完全不同的工作**。如果你一直都在寫程式碼，接下來很可能會發現，人類比 Intel CPU 難預測多了。

其實之前的主管 Nigel 也不擅長這方面的工作。「我可不想把所有時間全都花在毫無意義的會議上；我才不要成為那種管理者咧！」他略帶一絲逞強、虛張聲勢地說，「我認為我還是可以用 85% 左右的時間，來做程式設計的工作，至於**管理的工作**，只要一點點時間就足夠了。」

其實 Nigel **真正想說的是**，「我**根本不知道**如何管理專案，我只希望像過去一樣，繼續寫程式就對了，至於其他的事，船到橋頭自然直啦。」當然囉，哪有

這麼好的事？不過這也很難解釋，為什麼 Nigel 會在那個命中註定的日子，帶著 IBM ThinkPad 跑去高空彈跳。

不管怎麼說，Nigel 康復的速度確實驚人，而且他現在已經去和高空彈跳的夥伴，開了一家叫做 WhatTimeIsIt.com 的小公司，並擔任 CTO 首席技術長的工作，接下來他只有六個月時間，必須從無到有做出一套全新的系統——這次他總不能再假裝出意外了吧。

「管理軟體專案」並不是很多人都懂的一門藝術。學校裡並沒有「管理軟體專案」的相關學位，這個主題的相關書籍也不多。真正參與過成功軟體專案的人少之又少，他們大多賺了很多錢，早早就退休去養鱒魚，根本沒機會把積累的經驗傳承給下一代；另外還有很多人搞到最後精疲力盡，只想換個壓力小一點的工作，到市區幫那些小混混補習英文也算是一種解脫。

結果就是有許多軟體專案，以各種大剌剌的公開或不為人知的方式慘遭失敗，只因為整個團隊裡沒人知道，如何運作一個成功的軟體專案。有太多團隊甚至連產品都拿不出來，或是拖太久才推出產品，推出之後卻沒有人要。真正讓我感到憤怒的是，團隊裡的成員一點都不快樂，甚至痛恨工作的每一分鐘。生命實在太短暫，浪費時間去痛恨工作，這又是何苦呢！

幾年前，我在我的網站發表所謂的「約耳測試」，列出了運作良好的軟體團隊所具備的十二個特徵，其中包括像是「維護一個可用來追蹤問題的資料庫」、「讓求職者當場試寫程式碼」等（別擔心，我稍後就會詳細說明）。讓我很驚訝的是，許多人透過 email 跟我說，他們的團隊只能拿到這十二分裡的兩、三分。

兩、三分？！

這也太扯了吧——你能想像嗎？這就好像一群木匠想製作家具，卻連螺絲都沒聽過。他們只知道用釘子，卻又不會用鐵錘，只好拿著踢踏舞鞋，硬是把釘子釘進木頭中。

「管理軟體專案」所需的技能與技術，和「寫程式碼」完全不同；這根本是兩個不同的領域。寫程式碼之於管理工作，就好像腦部手術之於烤胡椒餅一樣，完全是兩碼子事。假設有一位才華出眾的腦外科醫生，因為時空連續體撕裂的神秘因素，被傳送到某個胡椒餅工廠；老實說，**就算他是**哈佛醫學院畢業生，

我們也沒理由期待，他能輕鬆製作出好吃的胡椒餅。但吊詭的是，大家卻好像都以為，頂級程式設計師並不需要太多調整，就能接手管理職的工作。

你和 Nigel 所面臨的局面，就好像前面所提到的腦外科醫生一樣，一直到接下管理工作才赫然發現——哦天呀！現在你要面對的不再是編譯器，而是一個個**活生生的人**。如果你覺得目前的 Java 編譯器問題很多、很難以掌握，那**你**肯定還沒遇過難搞的程式設計師；相較之下，Java 編譯器實在單純多了。如果與「管理一個很多人的團隊」相比，C++ 的難度根本就**微不足道**。

如果想要成功管理軟體專案，有一**些技術**其實還蠻好用的。這些最先進的技術，絕對可以超越「釘子」和「踢踏舞鞋」的程度。我們不但有鐵錘，還有螺絲起子，甚至還有雙斜面的複合滑動斜切鋸。本書的目標，就是盡可能向你介紹我所能想到的各種技術，從團隊主管如何估算時程，到軟體 CEO 如何制定競爭策略等等，希望可以盡量涵蓋到各個不同的層面。你將會學到：

- 如何聘僱、激勵最優秀的人才——這乃是成功的軟體專案、最關鍵的單一要素。
- 如何估算出真正可行的時程表，以及這件事一定要做的理由。
- 如何設計各種軟體功能，並寫出實際有用的規格，而不要做出那種寫完沒人要看、最後只能拿來當隔板的無用文件。
- 如何避開軟體開發常見的一些陷阱？如果程式設計師堅持要「拋開舊程式碼、全部砍掉重練」，為什麼他絕對是錯的？
- 如何組織團隊、激勵團隊？為什麼程式設計師需要一間能關門的辦公室？
- 在什麼情況下，即便你能從網路下載到足夠好的程式碼，你還是應該重寫自己的程式碼？
- 軟體專案啟動幾個月之後，為什麼總會越來越難以取得進展？
- 所謂的軟體策略，究竟是什麼意思？為什麼 BeOS 從第一天起就註定要失敗？
- ... 還有很多很多，族繁不及備載。

本書的看法是非常主觀的。要不是怕太囉嗦，我真想在每一句話開頭加上「我認為」這幾個字，因為書中的每一句話，確實都是我個人的意見。這些意見算不上完整，但或許是個好的開始吧。

你已經看過我的網站了，是吧……

本書大部分的內容，最初都是來自我在 *Joel on Software*（www.joelonsoftware.com）網站裡的一系列文章，過去幾年我一直在這個網站，寫下我個人的各種想法。我希望你手裡這本書，能比我的網站更具有**凝聚性**；我所謂的「**凝聚性**」，意思就是「可以讓你在浴缸裡放心閱讀，不必插電連上網路」啦。

我已經幫你把本書分成三個主要部分。第一部分是關於軟體開發的點點滴滴：如果你不希望製作出爛軟體，這些全都是你在團隊裡最應該做的事。第二部分的一系列文章，則是關於如何「管理」程式設計師和程式設計團隊。第三部分比較隨興，主要談的是如何建立一些大策略，讓軟體開發成為一種可持續發展的事業。你將會瞭解臃腫的軟體為何總是能勝出，並瞭解 Ben & Jerry's 的做法和 Amazon 有何不同，然後我還會嘗試說明，為什麼當你開始聽到團隊裡有人討論「軟體開發方法論」之類的話題時，實際上只是在暗示一個問題，那就是團隊中缺乏優秀的人才。

當然還有更多其他的內容，但你不妨自己埋頭深入，直接開始閱讀就對了。

1

程式語言的選擇

2002 年 5 月 5 日，星期日

開發者在面對特定任務時，為何會選擇某一種程式語言，而不選擇另一種程式語言？

如果需要超快的速度，通常我會選擇最原始的 C 語言。

如果想在 Windows 執行，又希望發佈的檔案越小越好，通常我就會選擇 C++ 搭配靜態連結的 MFC。

如果希望可以同時在 Mac、Windows 和 Linux 執行某個 GUI 圖形化界面，Java 就是個常見的選擇。雖然這樣的 GUI 並不完美，但至少可以正常運作沒有問題。

如果想快速開發 GUI、做出真正流暢的使用者界面，我比較喜歡用 Visual Basic，不過我知道這勢必要付出一些代價，因為 Visual Basic 所發佈的檔案會比較大，而且只能在 Windows 裡使用。

如果一定要在任何 UNIX 機器上運行，而且速度不需要很快，Perl 這個指令行工具就是個不錯的選擇。

如果程式必須在瀏覽器中執行，JavaScript 恐怕就是唯一的選擇了。如果遇到 SQL 這類的資料儲存程序，你通常就必須選擇某一家特定廠商的 SQL 衍生語法，否則就只能回家吃自己了。

重點是什麼？

其實我個人**幾乎從不**根據「語法」來選擇程式語言。是的沒錯，我的確比較喜歡採用 {...}; 這種寫法的語言（例如 C / C++ / C# / Java）。對於什麼叫做「好」語法，我個人就有一大堆的意見。但我並不會只因為想用分號做為程式碼的結尾，就願意接受一個 20MB 的執行階段函式庫。

因此，我對於 .NET 的跨語言策略，實在感到有一點質疑。.Net 最原始的構想，就是讓每個人都可以選擇自己想用的程式語言（數量恐怕不計其數），最後再以相同的方式運作。

VB.NET 和 C#.NET 這兩種語言，除了細微的語法差異之外，其他的部分幾乎完全相同。其他語言若想成為 .NET 世界的一部分，至少就必須支援某一組核心功能和資料型別，否則肯定無法與其他語言順利配合。但如果我要開發的是一個 UNIX 指令行公用程式，用 .NET 該怎麼做呢？如果用 .NET 的話，我要如何開發出一個小於 16K 的 Windows 執行檔呢？

.NET 感覺上好像給了我們不同的選擇，但我們可以選擇的東西——語法，卻不是我們真正很在意的東西。

2

回頭看一些最基礎的東西

2001 年 12 月 11 日，星期二

我在我的網站上花了很多時間，談論一些激動人心的「大局觀」，例如像是 .NET 與 Java 的對壘、XML 的策略、如何鎖定客戶、各種競爭策略、軟體設計、軟體架構等等。某種程度來說，這些全都像是蛋糕其中的一層。最上面的頂層，就是軟體策略。往下一層是像 .NET 這樣的軟體架構，然後再往下一層則是個別的產品：例如像是 Java 這樣的軟體開發產品，或是像 Windows 這樣的平台。

接著往蛋糕的更下層看。DLL 動態連結函式庫？物件？函式？不不不！再往下看！往下到了某個程度，你就會看到各種程式語言所寫出來的一行一行程式碼。

但這樣還不夠。其實我今天想談的是 CPU。沒錯，就是有一堆 Byte 資料在上面跑來跑去的那顆矽晶片。請先假裝你自己還是一個程式設計菜鳥。拋開你在程式設計、軟體、管理方面長期積累起來的所有知識，先回到最底層的馮·紐曼（Von Neumann）基礎架構。暫時先把 J2EE 從你的腦海中抹去。心裡只要想著 *Byte*（位元組）就好。

為什麼要這樣呢？我認為一般人所犯下的一些最大的錯誤（即使是在最高階的架構下），多半是因為對於最底層的一些簡單的事物、理解太過於薄弱或不夠完整所致。你好不容易建造了一座絕妙的宮殿，但地基卻是一團糟。本來應該鋪上漂亮水泥板的地方，卻成了一堆瓦礫。你的宮殿看起來雖然還不錯，但偶爾你的浴缸卻會滑到浴室的另一頭，而你竟然不知道是什麼問題。

所以，現在請先深吸一口氣。接下來請和我一起用 C 語言來做個小練習。

你還記得字串在 C 語言裡的用法嗎？所有的字串都是由一連串的 Byte 和一個數值為 0 的 null 字元所組成[1]。這樣的設計有兩個很明顯的含義：

1. 如果不用掃描的方式找出末端的 null 字元，就無法得知這個字串會在哪裡結束（也就是無法得知字串的長度）。

2. 字串裡不能包含任何的 0。所以 C 語言的字串形式，並不適合用來儲存那種有可能包含任意數值的二進位 blob 資料（例如 JPEG 圖片）。

為什麼 C 要採用這種方式來保存字串呢？這是因為 UNIX 和 C 語言都是在 PDP-7 這個微處理器上發明出來的，而這個微處理器採用的是一種叫做 ASCIZ 的字串型別。ASCIZ 的意思就是「用 0（Zero）來作為結尾的 ASCII 表示法」。

這是保存字串的唯一方式嗎？並不是。事實上，這可說是用來保存字串最糟糕的其中一種方式。如果你寫的是特別重要的程式、API、作業系統或物件類別函式庫，全都應該像躲瘟疫一樣、盡可能避免使用 ASCIZ 字串。但這又是為什麼呢？

我們一開始先來寫一段 `strcat` 的程式碼，這個函式會把某個字串連接到另一個字串的後面。

```
void strcat( char* dest, char* src )
{
   while (*dest) dest++;
   while (*dest++ = *src++);
}
```

我們可以稍微研究一下程式碼，看看這裡做了些什麼。首先，我們會掃描第一個字串，以找出它的 null 終止字元。找到之後，我們再掃描第二個字串，用一次複製一個字元的方式，把第二個字串複製到第一個字串後面。

1 更多關於字元字串的資訊，請參見 www-ee.eng.hawaii.edu/Courses/EE150/Book/chap7/subsection2.1.1.2.html。

這種字串處理與串接的做法，對於 Kernighan 和 Ritchie[2] 來說已經夠棒了，但其實它還是有點問題。我們就來看個例子好了。假設你想把「披頭四」四個人的名字，全都串接成一個長長的字串：

```
char bigString[1000];     /* 我根本不知道該配置多少記憶體 ... */
bigString[0] = '\0';
strcat(bigString,"John, ");
strcat(bigString,"Paul, ");
strcat(bigString,"George, ");
strcat(bigString,"Joel ");
```

這樣就行了，對吧？沒錯。看起來既乾淨又漂亮。

這段程式碼的執行效能表現如何呢？這是速度最快的做法嗎？這種做法的擴展性好嗎？如果有一百萬個字串要串接起來，這還會是個很好的做法嗎？

並不是哦。這段程式碼使用了所謂的 ***Shlemiel* 油漆工（*Shlemiel the Painter's*）演算法**。Shlemiel 是誰呢？他其實是下面這個笑話裡的主角：

> Shlemiel 找了一份油漆馬路的工作，要在馬路中間畫上白色的分隔線。第一天，他帶著一罐油漆，完成了 300 碼道路油漆的工作。「還不錯嘛！」他的老闆說道，「你的速度蠻快的！」然後便付給他一個銅板。
>
> 第二天，Shlemiel 只完成了 150 碼。「嗯，雖然不如昨天那麼好，但你的速度還是算快。150 碼也蠻厲害的，」於是又給了他一個銅板。
>
> 到第三天，Shlemiel 只漆了 30 碼的路。「才 30 碼？！」他的老闆喊道，「這我不能接受！你第一天所做的份量，足足有今天的十倍呀！這到底是怎麼回事？」
>
> 「我也沒辦法呀，」Shlemiel 說，「每經過一天，油漆桶就離我越來越遠呀！」

2 Brian Kernighan 和 Dennis Ritchie 是《*The C Programming Language*》（C 程式設計語言，第二版，Prentice Hall，1988 年）這本經典著作的作者。

（想做加分題嗎？有興趣不妨算算這幾個數字是怎麼來的？[3]）這個蹩腳的冷笑話，恰好準確說明了剛才 strcat 所採用的做法。由於 strcat 的第一部分每次都必須掃描過整個 dest 字串，一次又一次找出那該死的 null 終止字元，因此這個函式其實很慢、很沒有效率，而且根本無法隨資料量變大而順利進行擴展。我們每天都在使用的程式，其中蠻多都有這個問題。許多檔案系統所採用的實作方式，其實並不適合把太多檔案放在同一個目錄下，因為同一個目錄如果放入好幾千個檔案，其性能就會開始急劇下降。你可以嘗試一下，開啟一個裡頭塞滿滿的 Windows 資源回收桶，再查看實際的效果——它有可能需要好幾個小時才能顯示出來；所耗費的時間與其中所包含的檔案數量，顯然並不是線性的關係。這其中肯定藏有 Shlemiel 油漆工演算法。每當你遇到某個東西，照說應該具有線性的性能，但實際上卻表現出 n 平方的性能，這時候你就可以嘗試找找有沒有 Shlemiel 這個油漆工隱身其中。他通常都躲在你的函式庫中。如果你看到一堆的 strcat，或發現 strcat 藏身於某個迴圈，你也許並不會馬上大喊「n 平方」，不過八九不離十，兇手大概就是他了。

我們該怎麼解決這個問題呢？有一些很聰明的 C 語言程式設計師，實作出他們自己的 mystrcat 如下：

```
char* mystrcat( char* dest, char* src )
{
   while (*dest) dest++;
   while (*dest++ = *src++);
   return --dest;
}
```

這裡究竟做了什麼改變呢？我們只是運用很小的額外成本，送回了一個「**指針**」（pointer），指向最後那個全新的、更長的字串末尾處。如此一來，調用此函式的程式碼就足以判斷，不必再重新掃描字串，就能直接串接後面的字串了：

```
char bigString[1000];     /* 我根本不知道要配置多少記憶體 ... */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p,"John, ");
p = mystrcat(p,"Paul, ");
p = mystrcat(p,"George, ");
p = mystrcat(p,"Joel ");
```

3 相關的數學討論，請參見 discuss.fogcreek.com/techInterview/default.asp?cmd=show&ixPost=153。

如此一來，效能表現上當然就會變回線性而非 n 平方，因此當你有很多字串要串接起來時，程式效能也不會急劇下降了。

Pascal 的設計者也有注意到這個問題，於是就利用字串的第一個 Byte，來保存字串所佔用的 Byte 數量，藉此來「修正」這個問題。這就是所謂的 ***Pascal* 字串**。這種字串裡的值可以包含 0，而且並不需要用 null 來做為終止字元。由於一個 Byte 可容納的最大數字為 255，因此 Pascal 字串的長度也被限制為 255 個 Byte，不過由於它並沒有採用 null 來做為終止字元，因此它所佔用的記憶體與 ASCIZ 字串是相同的。Pascal 字串最棒的就是，永遠不必為了計算字串長度而使用迴圈。只要一個組合語言指令，就能查出 Pascal 字串的長度，根本就不需要用到迴圈。因此，它的速度超級快。

之前的麥金塔（Macintosh）作業系統，裡裡外外到處都在使用 Pascal 字串。其他平台的許多 C 語言程式設計師，也會使用 Pascal 字串來提升速度。Excel 內部也是使用 Pascal 字串，這就是為什麼 Excel 許多地方的字串都被限制為 255 個 Byte，而這也是 Excel 速度飛快的原因之一。

過去有很長一段時間，如果你想在 C 程式碼中放入一個 Pascal 字串文字，都必須寫成下面這樣：

```
char* str = "\006Hello!";
```

是的你沒看錯，你必須自己手動計算出 Byte 數，再把它寫死到字串的第一個 Byte 中。有些懶惰的程式設計師會寫出下面這樣的程式碼，但這樣反而讓程式變得更慢：

```
char* str = "*Hello!";
str[0] = strlen(str) - 1;
```

你可以注意到，在這樣的做法下，你的字串不但是以 null 做為結尾（這是編譯器所做的事），而且也具有 Pascal 字串的形式。我以前都把這種字串叫做「**該死的（*fucked*）字串**」，因為這樣總比「***null* 結尾的 *Pascal* 字串**」簡潔多了，不過本書屬於保護級，所以**你**還是用那個比較長的名稱吧。

在之前的程式碼中，我刻意略過其中一行很重要的程式碼。還記得下面這行程式碼嗎？

```
char bigString[1000];    /* 我根本不知道該配置多少記憶體 ... */
```

由於我們今天研究的正是這些 bit 和 byte 的問題，所以實在不應該略過這行程式碼。好吧，應該把它弄對才行：我們必須搞清楚究竟需要多少 Byte，然後再配置正確數量的記憶體。

呃……可以不管它嗎？

如果不管它，聰明的駭客讀到我的程式碼，就會注意到我只配置了 1000 Byte，然後**天真的希望**這樣就足夠了；於是，他們就會找一些聰明的方法來玩我，例如把 1100 Byte 的字串送入我的 `strcat` 裡，灌入只有 1000 Byte 的記憶體，這樣就會覆寫掉堆疊框（stack frame）進而改變 return 的位址，因此函式 return 時就會跑錯地方，進而去執行到駭客所寫的一些程式碼。每次有人提到**緩衝區溢出**（*buffer overflow*）的問題，其實指的就是這樣的問題。在過去，它可是駭客攻擊和蠕蟲病毒的頭號兇手；不過，後來 Microsoft Outlook 倒是讓一些青少年進行駭客攻擊變得更容易了。

好吧好吧，所以說很多程式設計師其實都是漏洞百出、考慮不周的懶惰鬼。他們都應該先弄清楚需要配置多少記憶體才對。

但實際上，C 語言**並沒有**讓你輕鬆解決此問題的做法。我們先回到前面「披頭四」人名的例子好了：

```
char bigString[1000];    /* 我根本不知道該配置多少記憶體 ... */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p,"John, ");
p = mystrcat(p,"Paul, ");
p = mystrcat(p,"George, ");
p = mystrcat(p,"Joel ");
```

我們究竟該配置多少記憶體呢？首先讓我們嘗試一下正確的做法。

```
char* bigString;
int i = 0;
i = strlen("John, ")
    + strlen("Paul, ")
    + strlen("George, ")
    + strlen("Joel ");
bigString = (char*) malloc (i + 1);
/* 記得多留個空間給 null 終止字元！ */
...
```

我看你目光有點呆滯，大概是想走人了吧。我不怪你，但請再忍耐一下，因為它很快就要變有趣了。

我們必須把所有的字串掃描過一次，才能判斷其大小，然後還要再掃描一次，才能把字串連接起來。如果使用的是 Pascal 字串，至少 `strlen` 這個操作會很快。也許我們可以寫個新版本的 `strcat`，讓它自動配置記憶體的大小。

不過，這裡又出現了**另一個**大麻煩：記憶體配置器（memory allocator；也就是程式碼裡的 malloc 函式）。你知道 `malloc` 的運作原理嗎？從本質上來說，`malloc` 會持續維護著一串很長很長的聯結串列（linked list），它是由可用的記憶體區塊所組成，也就是所謂的「**可用鏈**」（free chain）。當你調用 `malloc` 時，它就會掃描這個「可用鏈」，找出一個足夠大、能滿足需求的記憶體區塊。接下來它就會把所找到的區塊，切分成兩個區塊（其中一個就是你所需要的大小，另一個則是切分後剩餘的部分），然後把你所需要的區塊交給你，再把剩餘那個區塊（如果有的話）放回到「可用鏈」這個聯結串列中。之後你如果調用了 `free`，它也會把所釋放的區塊放回「可用鏈」。可想而知，到最後整個「可用鏈」就會被切成很多很多個小區塊，而當你再度向它要求一個比較大的區塊時，就沒有足夠大的區塊可供你使用了。因此，`malloc` 就會叫個暫停，然後開始在「可用鏈」裡翻找、清理，把相鄰的一些可用小區塊，合併成比較大的區塊。這恐怕需要三天的時間。而到最後這一團混亂的結果，就是 `malloc` 在效能方面天生的罩門；這也就表示，malloc 的速度絕對不會很快（畢竟它經常要去掃描可用鏈），有時候（而且不一定什麼時候）遇到它正在進行清理時，速度更是慢得嚇死人。（順帶一提，記憶體自動回收（garbage-collected）機制在效能方面也有相同的特性。沒想到吧！所以大家總是說記憶體自動回收機制會造成效能上的損失，其實這樣的說法並不算完全正確，因為一般最典型的 `malloc` 實作方式，同樣也具有相同類型的效能損失，只不過比較沒那麼嚴重而已。）

比較聰明的程式設計師在使用 `malloc` 時，總會把記憶體區塊的大小配置為 2 的乘冪，以最大程度減少潛在的問題。你知道的，也就是採用 4 Byte、8 Byte、16 Byte、18446744073709551616 Byte 等等之類的大小。如果你有在玩樂高，這應該很直觀才對，因為這樣可以最大程度減少可用鏈裡出現大小很奇怪的區塊數量。雖然這樣的做法看起來很浪費空間，但應該很容易就可以看得出來，它所浪費的空間絕不會超過總空間的 50%。所以你的程式所使用的記憶體，絕不會超過它真正所需空間的兩倍，這應該不是什麼大問題才對。

假設你寫了一個很聰明的 strcat 函式，可以自動配置目標緩衝區的大小。我們是不是就應該總是把記憶體配置為所需的大小呢？我的老師兼人生導師 Stan Eisenstat[4] 建議，每當你調用 realloc 時，就應該把之前所配置的記憶體加大一倍。這也就表示，你調用 realloc 的次數絕不會超過 *lg(n)* 次，而且即使面對超長的字串，這樣的做法也有相當不錯的效能，而你所浪費的記憶體，絕不會超過總記憶體的 50%。

總而言之，在 bit 與 Byte 的世界裡，情況一定會變得越來越混亂。你現在是不是覺得很慶幸，終於不必再用 C 來寫程式了？我們現在有像 Perl、Java、VB 和 XSLT 等這些偉大的程式語言，你再也不必去想那些煩人的事情；這些程式語言自動就會以某種方式處理相關的問題。但偶爾在你家客廳正中央，還是會冒出水管之類的東西。我們經常不得不考慮，究竟該使用哪一種物件類別——用 String 還是用 StringBuilder 比較好呢？之所以要做這些事，正是因為編譯器還不夠聰明，還無法完全理解我們想完成的工作，也**無法**阻止我們一不小心就用了 Shlemiel 油漆工演算法。

之前在我的部落格裡，我很隨性寫了一則評論[5]，提到資料若保存成 XML，執行一些像「SELECT author FROM books」這種 SQL 語句，速度上就會慢很多。今天這篇文章是我丟出那則評論之後才寫的，可以算是一篇補充的說明，主要是怕大家不知道我在說什麼，因此我在這裡再強調一下，今天所談的全都是 CPU 層次的概念，有了這樣的理解之後，我在那則評論裡所下的斷言，應該就會更清楚才對。

關聯式資料庫究竟是如何實現「SELECT author FROM books」的呢？在關聯式資料庫中，資料表（例如 books 資料表）裡每一行記錄的 Byte 長度全都是相同的，而且每個欄位與每一行開頭的距離，全都保持著固定的偏移量。舉例來說，如果 books 資料表裡每一筆記錄的長度都是 100 Byte，而其中 author（作者）欄位是存放在偏移量 23 的位置，那麼每一筆資料裡 author 作者的資料，肯定就是存放在第 23、123、223、323 Byte 的位置。每次取得某一筆查詢結果後，若想要移動到下一筆記錄，程式碼該怎麼寫呢？基本上只要像下面這樣就可以了：

```
pointer += 100;
```

4 請參見 www.cs.yale.edu/people/faculty/eisenstat.html。

5 請參見 www.joelonsoftware.com/articles/fog0000000296.html。

只需要一個 CPU 指令耶。這肯定超快的！

接著再讓我們看一下 XML 格式的 books 資料。

```
<?xml blah blah>
<books>
  <book>
    <title>UI Design for Programmers</title>
    <author>Joel Spolsky</author>
  </book>
  <book>
    <title>The Chop Suey Club</title>
    <author>Bruce Weber</author>
  </book>
</books>
```

我就很快問一下好了。移動到下一筆記錄的程式碼，該怎麼寫才好呢？
呃……

這時候優秀的程式設計師會說，只要先把 XML 解析成樹狀結構，放到記憶體中，這樣就可以快速對它進行操作了。為了執行「`SELECT author FROM books`」，CPU 所要完成的工作量絕對會讓你很煩很想哭。每一個寫過編譯器的人都知道，詞法和語法分析的工作，就是編譯過程中速度最慢的部分。我們必須在記憶體內進行詞法、語法分析，還要構建出一個抽象語法樹，這些工作全都會牽涉到很多的字串相關操作，而這類操作全都很花時間，另外還有很多的記憶體配置工作，我們知道它也很花時間。況且我們還要假設你**擁有**足夠的記憶體，可以載入所有的東西。以關聯式資料庫來說，從某一筆記錄移動到另一筆記錄的效能是固定的，實際上就是一**個 *CPU* 指令**而已。這很大程度其實是故意這樣設計的。而且因為有記憶體映射檔案，所以只需要從磁碟載入實際上會用到的分頁檔案就可以了。而以 XML 來說，如果有預先做好解析的工作，從某一筆記錄移動到另一筆記錄的效能也是固定的，但一開始要先花一段蠻長的時間，先做好解析的工作；如果沒有預先進行解析，從某一筆記錄移動到另一筆記錄的效能表現，就會隨著前一筆記錄的長度而變化，而且不管是哪一種做法，都需要好幾百個 CPU 指令才能完成工作。

對我來說這也就表示，如果資料量很大、又想追求速度表現，就不能採用 XML 的做法。如果你只有少量的資料，或是不需要很快的速度，那 XML 確實是一種很好的格式。如果你真的想兩全其美，就必須想辦法在 XML 之外，另外保存一些 metadata（詮釋資料），例如 Pascal 字串裡的字串長度 Byte 資料，

它可以用來提示你所需的資料放在檔案裡的哪個位置，這樣你就不必進行解析和掃描的工作了。不過這樣一來，你當然就不能使用文字編輯器來隨意編輯 XML 檔案，因為這樣就會讓 metadata 失去作用；問題是如果不能隨意編輯，就不能算是一個真正的 XML 檔案了。

到現在還在看本篇文章的各位讀者們，我希望你們確實學到了一些東西，或是重新思考了一些概念。我希望各位重新思考一下，資訊科學系大一所學的那些看似無聊的東西（比如 `strcat` 和 `malloc`），如果真正搞懂其中的原理，建立一些全新的思維，這樣你在處理像是 XML 這樣的技術時，就能針對最頂級的軟體策略和架構決策，做出更多的思考。至於今天的作業，各位可以想想為什麼 Transmeta 的晶片總是讓人覺得卡卡頓頓的。你也可以稍微想一下原始的 HTML 規格，其中 TABLE 的設計為何如此糟糕，以至於使用 modem 撥接上網的人，根本無法快速顯示網頁裡的大型表格。又或者你也可以思考一下，為什麼 COM 如此快速，但是在跨越 process 邊界時卻又快不起來。你還可以再想想看，為什麼 NT 的設計人員要把顯示驅動程式放入內核空間，而不是放在使用者空間。

這些東西全都需要從 Byte 的層面去思考，而且它會影響我們在各種架構和策略中所做出的重大決策。這就是為什麼我認為在教學上，資訊科學系大一生都應該從基礎開始學起，而且應該使用 C 語言，並從 CPU 開始建構學習的基礎。許多資訊科學系的程式設計課程，都認為 Java 是一種很好的入門語言，因為它很「簡單」，而且不會有很多無聊的字串 / malloc 之類的東西，把你搞得昏頭轉向，況且你還可以學習到很酷的 OOP 物件導向程式設計概念，讓你的大型程式變得更加模組化；儘管有這種種的好處，我還是蠻不贊成用 Java 來做為入門的語言。在我看來，這簡直就是一場早晚會出問題的教育災難。一代又一代的學生從學校畢業，不斷在各個角落使用 Shlemiel 油漆工演算法，而且甚至還意識不到問題，因為他們根本不知道從非常底層的角度來看，字串其實是很難搞的東西，而這樣的概念在 Perl 腳本中是看不出來的。如果你想把某個東西教好，就必須從最底層、從最基礎開始教起。這就像《**小子難纏**》（Karate Kid）這部電影一樣。不斷的打底、上蠟、打底、上蠟，就這樣不斷持續三個禮拜。然後你突然就會發現，自己很輕鬆就能幹掉其他小鬼頭了。