# 教堂與市集

Eric Steven Raymond

1997

# 教堂與市集

# 正體中文版序

原始版本為 1999 年 5 月 5 日，由謝志昌所翻譯；英文最新版為 2002 年 8 月 2 日的 3.0 版，由於內容更動處不少，因此逐一更新；翻譯以語句通順達意為主要考量，難免與英文版的字詞有所不同，為求盡善盡美，以 CC 授權釋出，希望各方同好自由修改散佈。

# 第一章、教堂與市集

Linux 打破了許多軟體發展的傳統，這個世界級的作業系統在五年前（1991 年）僅僅靠著如絲般的網際網路，神奇地聯合了散布在全世界數以千計兼職的玩家們來發展它，誰曾料到會發生這樣的事情呢？

我當然也沒料到，Linux 出現在我電腦螢幕是在 1993 年初，當時我埋首於 UNIX 及開放性原始碼的軟體發展已有十年，1980 年代中期，我是 GNU[1] 專案首批的貢獻者之一，我寫過許多開放性原始碼的軟體放到網路上供人使用，也曾獨立或協同發展好幾個程式（nethack，Emacs 的 VC 和 GUD 功能，xlife，…等等），這些程式到今天仍廣泛地為人所用，我想我知道這是怎麼辦到的。

Linux 扭轉了許多我認為我已知道的觀念。多年來我一直宣揚使用小工具集、快速原型發展及程式進化的 UNIX 福音。但我也相信對於有一定複雜度的程式必需使用集中和有經驗的方法來開發，我相信最重要的軟體（作業系統以及龐大的工具程式如 Emacs）必須如建造一座教堂般，由個別的高手或一小群專家在光輝的孤立中小心翼翼地精雕細琢，時機未到之前，不會釋出測試版。

Linus Torvald[2] 的軟體發展風格（儘早並經常發表新版本，授權每一件作者可以委託的事，不拒絕幾乎到混亂程度的程式）的出現如同一個驚奇，沒有令人肅然起敬的教堂，甚至 Linux 的同好們似乎組成了一個有不同流程和不同方式的大市集（Linux 的檔案服務站台就是它適切的象徵，每個人都服從著自由的規則），以這個風格發展出來的 Linux 既一致又穩定，表面上看來真是一連串的奇蹟。

市集模式似乎是可行的，並且運作得很好，這個事實帶來了相當的震憾。當以我的方法去認知時，我除了努力做好個人的專案，並也試著去瞭解為什麼在Linux的世界，不但沒有因為渾沌不清而四分五裂，反而以教堂建造者幾乎想像不到的速度在茁壯。

直到 1996 年年中，我想我才開始瞭解這一件事。我得到了一個絕佳的機會來試驗我的理論，這個機會是一個開放性原始碼型式的專案，正好可以試用市集模式來發展，所以我做了這個專案，而且更有意義的是它成功了。

接下來在這篇文章中我將陳述這個專案的故事，並且以它為例提出關於有效地利用開放性原始碼模式來發展軟體的格言，這些法則並非都是我在 Linux 中第一次學到，但我們可以看到 Linux 的世界是怎麼賦予它們特別的意義。如果我是對的，那麼這些格言將會幫你真確地瞭解是什麼促成 Linux 社群成為好軟體的原創者，並且幫助你變得更具生產力。

[1] GNU 是自由軟體基金會（Free Software Fundation）的一個專案，目標是發展出 UNIX 上所有程式的自由版本，Emacs 是自由軟體基會發展出來的一支程式，可做文字編輯器，提供寫程式的整合發展環境，用來讀電子郵件，新聞群組，甚至瀏覽網頁。更詳細的資訊請參考 http://www.gnu.org。 —— 譯註。

[2] Linus Torvald 是 Linux 核心程式（kernel）的原始作者。 —— 譯註。

# 第二章、信一定要寄到

自 1993 年起，我一直擔任一家提供免費上線的小型 ISP 的技術人員，這家 ISP 叫做 County InterLink（CCIL），位於賓夕凡尼亞州的 West Chester。（我本身參與捐款設立 CCIL，並寫了我們獨一無二的多人佈告欄軟體，你可以用 telnet 連線至 locke.ccil.org 一探究竟。目前它共有三十條線，可提供近三千位的使用者上網。）這個工作讓我可以一天二十四小時透過 CCIL 56K 的線路上網。事實上，我的確非常需要它！

因此我必須常常利用便捷的網際網路電子郵件，但由於一些複雜的原因，使得我家裡的機器（snark.thyrsus.com）要以 SLIP 協定連上 CCIL 有所困難。最後我還是做到了，但我馬上發覺我必須週期性地以 telnet 連上 locke 來檢查我是否有新信件，這實在是一件煩人的事。我想要的是：我的郵件可以被送到 snark，而且送達時會通知我，然後我可以用 snark 上的工具程式來處理這些郵件。

網路原生的 SMTP（Simple Mail Transfer Protocol）轉信功能在這裡幫不上忙，因為我個人的機器並不是時時都和網路連結著而且它也沒有固定的 IP 位址。我所需要的程式是這樣的：它可以間歇地連線把我的電子郵件都抓回來並放在我的機器。我知道有這樣的程式，它們大部份都使用一個簡單的網路應用協定叫 POP（Post Office Protocol）。目前 POP 是最通用的郵件客戶端協定，但當時我使用的郵件讀取端卻沒有這個協定。

我需要一支 POP3 的客戶端程式。所以我到網路上搜尋之後找到一個。其實我找到了三或四支這樣的程式。我使用了其中一個一陣子，但是它少了一個重要的特性，就是精巧地處理抓回來信件檔頭附的來源處，以使收件人能回覆信件至正確的網址。

這個問題是這樣的：假設 locke 上有一個叫做 joe 的使用者寄信給我，當我把信抓到 snark 上並回信給 joe 時，我的送信程式會傻呼呼地試著把這封回信送給根本就不存在 snark 上的使用者 joe。所以我必須動手編輯回信的網址即加上 <@ccil.org>，很快地這變成了一件頗痛苦的事情。

很明顯的這是電腦應該幫我做的，可是現有的 POP 客戶端程式卻沒有任何一支知道要這麼做。這使我們學會了第一課：

**格言1：好軟體都是起源於程式發展者要解決切身之痛。**

也許這已是眾所皆知（不是有句著名的諺語叫：「需要為發明之母」嗎？），但有多少軟體發展者為了薪水，把時間都耗在寫他們既不需要也不喜愛的程式上呢？然而這樣的事不會發生在 Linux 的世界 —— 這也許可以解釋為什麼由 Linux 社群們發展出來的軟體的平均水準都這麼高。

所以，我是否該立即如抓狂般地投入寫作一支新的 POP3 客戶端的程式來和既有的一較高下？並不如你所想的，我仔細地檢視我手上已握有的 POP 工具程式，看看那一支最符合我的需要，因為：

**格言2：優秀的程式師知道要寫程式，偉大的程式師知道要改寫（和重覆利用）程式。**

我並非在宣稱我是一個偉大的程式師，但我願去效法。偉大的程式師還有一項重要的特色是老製造著偷懶的辦法，他們認為人們爭取最好的成績並不是為了努力的過程，而是為了最後的結果。更何況由一個部份可行的解決方法開始總比什麼都沒有容易得多。

舉例來說，Linus Torvalds 當初寫 Linux 的核心程式時也不是從零開始，他是由借重 Minix 的程式和構想開始的（Minix 是一個像 UNIX 的小型作業系統，它在386機器上執行），然而到最後原來屬於 Minix 的碼不是被移出就是被改寫 —— 儘管如此，Minix 的碼畢竟曾存在於 Linux 中，並且曾為尚未茁壯的 Linux 提供一個骨架，最後終於誕生了 Linux。

為仿效這樣的精神，我開始找尋一個現有而且寫得有條理的 POP 工具程式，來作為我發展新程式的基礎。

在 UNIX 的世界中，原始程式碼共享的傳統讓我們可以很容易地重覆利用程式碼，這也是為什麼 GNU 專案要選擇 UNIX 作為它發展的平台，UNIX 作業系統本身幾乎沒做什麼保留，Linux 的世界也遵行著這個傳統，到接近它技術極限的地步，它供人運用的開放性原始碼程式，有天文數字般地多，所以在 Linux 已有的資源中找到一個足夠好的程式要比其他的作業系統容易。

而我也的確找到了，連我先前的搜尋再加上這一次總共有九個程式候選 —— fetchpop、PopTart、get-mail、gwpop、pimp、pop-perl、popc、popmail 及 upop。我首先選擇 Seung-Hong Oh 寫的 fetchpop 作為出發點，我加入了我要的「重寫郵件檔頭」功能，並且作了多處的改善。原作者同意將這些納入 fetchpop 的 1.9 版。

幾個星期之後，我緩緩地讀著 Carl Harris 寫的 popclient 的原始碼，並且發現了一個問題：雖然 fetchpop 有一些好的初始想法（如它的伺服程式模式），但它只能處理 POP3 的協定，而且它的原始程式只有業餘的水準（Seung-Hong 是個聰明的程式設計師，但當時經驗不夠老道）。而 Carl 寫的程式碼就比較好，具相當的職業水準和穩固性，但他的程式缺乏了許多重要的特色，如 fetchpop 中巧妙的實作（包括我加入的部份）。

要換還是不換呢？假如我選擇換的話，那麼換到一個較好的發展基礎所要付出的代價就是要丟掉我已經寫好的程式碼。

事實上我選擇換的動機是為了能支援多種 post-office 的協定，雖然 POP3 最廣為人採用，但它卻不是唯一可用的協定。Fetchpop 和其他同類的程式並不支援 POP2、RPOP 或 APOP，而我早先有一個概略的想法（只是為了有趣）：加入IMAP（Internet Message Access Protocol，最新最強的 post-office 協定）協定的支援。

一個更理論性的理由讓我覺得換到新的發展基礎是個好主意，這個理論是早在 Linux 出現前，我就已經學會了：

**格言3：「計畫好如何捨棄一條路吧，你遲早會想盡辦法這麼做的。」**（Fred Brooks，《人月神話》，第十一章）

否則，計畫走另一條路吧。針對一個問題，在尚未實作出第一個解法前，你通常並不真正瞭解這個問題。也許第二次的時候你才能充分瞭解怎麼做才對，所以即使你想做對一件事，但起碼你要準備從第一次做起[3]。

我告訴我自己：修改 fetchpop 是第一次。所以我換到 Carl Harris 的 popclient 繼續發展。

當 1996 年 6 月 25 日我送給 Carl Harris 我第一次對 popclient 所做的修正後，我才曉得他已經對這支程式沒興趣了。原來的程式碼乏人照料已有一段時間，還包括一些次要的錯誤。有許多修正要做，很快的我們兩人都同意由我接手整個程式是合理的一件事。

在我不經意間，這個專案的規模擴大了，我不再只注意現在的 popclient 要修補那些次要的部分，而是要接下維護整個整程式，在我腦海曾浮現許多主意，我想這可以引導我改變 popclient 的主要部分。

在鼓勵分享程式碼的軟體文化下，一個專案以這樣自然的方式演進。我表現出：

**格言4：抱持正確的態度，就會發現有趣的問題。**

但 Carl Harris 的態度更為重要，他懂得：

[3]在《*Programing Pearls*》，著名的資訊工程格言家 Jon Bentley 評論 Brooks 的觀察說「如果你丟棄一次，你將丟棄第二次」。他幾乎是對的。他們兩人的重點不是你應該預期第一次的嘗試是錯的，而是以正確的想法開始會比挽救一場災難來得更有效率。

**格言5：當你對一個問題不再感興趣時，你最後的責任就是找位能勝任的接棒人。**

雖然 Carl 和我沒討論這些，但我們卻有一個共同的目標就是對這個問題寫出最好的解法。現在唯一的問題只剩：證明我是個可信賴的人，而我已經做到，所以他便欣然地把這支程式交給了我。我希望這個專案在我手中能變得更好。

# 第三章、擁有使用者的重要

我繼承了 popclient，更重要的是我也繼承了它的使用者。擁有使用者是很棒的一件事，並不是因為這展現出你在解決他們的問題，或是你在做好事。而是好好地培養使用者，他們可以變成協同發展者。

UNIX 的傳統中有一種力量，那就是許多使用者同時也是程式高手，Linux 促使這變成一件非常愉快的事。這是因為原始碼是公開的，所以使用者可以變成有影響力的高手，這對縮短除錯的時間實在太有助益了。只要你給一點掌聲，使用者們會幫你診斷問題，建議需修正的地方，以及改進程式碼，這比你自己一個人包下全部的事要快得許多。

**格言6：把你的使用者視為協同發展人，可以讓你傷最少的腦筋，但做到原始碼的快速改善，程式的除錯有績效。**

這種效應所造成的影響力很容易就被低估，事實上，開放性原始碼世界的所有人，幾乎都嚴重低估了因使用者增多而產生用以對抗系統複雜度的力量，直到 Linus Torvalds 明白地揭露了這一點。

其實我認為 Linus 在技術上最聰明和最重大的貢獻並不在於寫出 Linux 的核心程式，而在於發明 Linux 的發展模式。在一次和他的會面中，我提出了這點見解，他微笑著，並重複他常說的一句話：「基本上我是一個非常懶的人，因其他人在 Linux 上真正的努力，而感到與有榮焉。」懶惰就像狐狸一樣地精明，或者就如同 Rober Heinlein 曾說：「因為太懶所以成功了。」

回顧過去的例子，在 GNU Emacs 的 Lisp 程式庫及其 Lisp 程式碼的資源庫中，我們可以看到 Linux 模式所用的方法和所得的成功。相對於 Emacs 中用 C 語言寫的核心部分及自由軟體基金會其他的工具（這都是以建造教堂的模式發展），Emacs Lisp 程式碼的資源庫非常地使用者導向並且更新很快，好的點子和原型在最後成熟穩定前常常都已重寫過三或四次，藉由網際網路而來的非緊密合作進行得很頻繁，就像 Linux 一樣。

我在還沒寫作 fetchmail 前，最成功的傑作大概要算是 Emacs 的 VC（version control）功能了，這項專案進行時，我用像 Linux 一樣的合作模式，用 email 和其他三位作者互相聯繫，到今天為止，我只見過其中一位（他就是 Richard Stallman，Emacs 的作者以及自由軟體基金會的創辦人）。Emacs 的 VC 功能是作為 SCCS，RCS 及後來的 CVS 的前端處理，提供版本控制功能「按一下」的操作法，這是由某位仁兄撰寫的小而有力的 sccs.el 功能改良而來，VC 功能的發展相當成功，這是因為 Emacs 用的 Lisp 程式可以快速地經歷「發表 —— 測試 —— 改良」，而不像 Emacs 本身核心的發展那樣緩慢。

Emacs 的故事並非獨一無二。有其他的軟體結合了雙層的架構與雙層的使用者，前者採用教堂模式開發核心，後者採用市集模式開發工具箱。例如 MATLAB，一個商業的資料分析與視覺化工具程式。MATLAB 與其他類似結構的產品都指出，發酵與創新都在開放的部份發生，在那裡各種社群都能夠修補這些成果。

# 第四章、儘早發表，經常發表新版本

儘早，經常發表新版本是 Linux 發展模式中非常重要的一環。過去，大部份的程式發展者（包括我）認為這個策略對較大型的專案是不好的，因為早期的版本幾乎可以定義為多錯的版本，我們並不想把使用者的耐心消磨殆盡。

這個過去的信念加強了軟體的發展要用建造教堂的方式的想法。假如我們極欲強調的目標是讓使用者在軟體中發現最少的錯誤，那你何不每半年（或更長）才發表一個新版本，並且在發展新版本的期間，賣力地除錯而累得像條狗似的。Emacs 的核心部分（用 C 語言寫的）就是用這種方式發展的，但它的 Lisp 程式庫就不是。因為 Emacs 的 Lisp 資源庫不在自由軟體基金會的管轄內，你可以在其中找到新發展的 Lisp 程式使用，而不受限於 Emacs 的發表週期[4]。

在 Emacs 的 Lisp 程式庫中，最重要的一個來源是俄亥俄州的 elisp 資源庫，它先前的精神就已經具有今日大規模 Linux 資源庫的特色，但當時我們之中卻很少有人思考過我們到底做了什麼，甚至想過我們已對自由軟體基金會的「建造教堂」的發展模式提出質疑。1992 年左右，我很認真地要把俄亥俄州 elisp 資源庫中許多程式加入 Emacs 正式的 Lisp 程式庫中，但卻遭遇到官方的阻礙而失敗了。

但一年之後，Linux 已受到四方的矚目，也帶來不同而且更健康的觀點，Linus 的開放性發展策略和「建造教堂」非常不同。當時 Linux 的兩大資源庫 sunsite 和 tsx-11 正在萌芽，有許多版本在交流著，Linux 核心系統發表新版本的頻繁程度前所未有。

Linus 以最有效的方法，視使用者為協同發展者：

**格言7：儘早，經常發表新版本，並且傾聽使用者的意見。**

Linus 的創新並不完全在此（這在 UNIX 世界是行之有年的傳統了），而在於提高這個做法效力的層次，使其能匹配他在發展的系統的複雜度。早期在 1991 年左右，許多人都知道他一天內發表一次以上 Linux 核心程式的新版本。因為他善用網際網路和協同發展者們合作更勝於其他人。

他能我也能嗎？還是只有像他這樣的天才才辦得到？

我並不認為如此，雖然 Linus 是一位很厲害的高手（在我們之間，有多少人能夠完整地寫出一個具有商品品質的作業系統核心呢？），但 Linux 並不是一個空前耀進的觀念，Linus 也並非（或者說至少目前還不是）如 Richard Stallman 或 James Cosling（NeWS 和 Java 的創始者）這樣的天才創新者，而我個人認為他是一位天才工程師，他有避免程式錯誤及避免程式發展掉入死胡同的第六感，和找到兩點間最省力路徑的技巧。事實上，整個 Linux 的設計中，我們可以看到 Linus 表現出的品質和他保守而簡單的設計取向。

承上所說，如果快速地發表新版本和徹底地善用網際網路媒介不是突然冒出，而是以 Linus 天才工程師洞見所得的最省力路徑，那麼他把網際網路的什麼功用發揮到最大？

其實問題的本身已反應出答案，Linus 讓 Linux 的高手和使用者們經常感覺刺激和有收穫 —— 感覺刺激是因協助發展 Linux 得到自我滿足，感覺有收獲是因經常（甚至每天）進步的 Linux 幫助他們把工作做得更好。

[4] 成功採用市集模式的開放原始碼例子，早在網際網路時代之前，與 UNIX 無關，這網際網路的傳統早就存在。info-Zip 這支壓縮工具的發展在 1990-1992 年，主要是針對 DOS，就是一個例子。另一個子是 RBBS 電子佈告欄（也是針對 DOS），於 1983 年開始，發展了一個強壯的社群，直到目前（1999 年中）都還經常發布新版本，儘管網路郵件與檔案分享有更巨大的優勢。當 info-Zip 社群依賴網路郵件形成的規模，RBBS 的開發者文化實際上在 RBBS 支撐線上社群，使它完全獨立於 TCP/IP 的基礎設施。

Linus 想直接將投入除錯和發展的「人-時」（person-hours）數加到最大，即使要付出的代價是程式碼的不穩定，或是因一些程式錯誤被證實無法追蹤而嚇走原有的使用者。Linus 會如此做是因為他相信：

**格言8：以足夠多的 beta 版測試者和協同發展者做基礎，幾乎程式中的每一個問題都可以很快地找出來，並且對某些人而言，針對發現的問題的解決方法是顯而易見的。**

或者用比較不那麼正式的說法：「足夠多的人來看程式，所有的錯誤都變得淺顯」，我將此命名為「Linus法則」。

我原本先前的論述是：「某些問題對某些人而言是容易解決的」，但Linus有不同的意見：「瞭解並解決問題的人不一定是第一個發現問題的人」，他說：「有些人發現問題，有些人解決問題，我願正式強調 —— 發現問題是較大的挑戰。」而在 Linux 的世界，發現問題和解決問題的速度都很快。

對「Linus法則」來說，我想這就是教堂模式和市集模式最主要的不同，以教堂建造者的觀點來看程式發展，程式錯誤和相關問題難以處理，並隱伏在深處，需要數個月的工夫仔細察看來找到它們，而這對程式發展者的自信少有加許。發展的期間越長，一旦經冗長等待的新版本發表後不如預期完美，使用者的失望也越大。

另一方面就市集發展模式的觀點來看，它假設程式錯誤都是顯而易見的，或者說至少在上千位渴望新版的協同發展者面前，程式錯誤很快地都變得淺顯，因此經常發表新版本是為了獲得更多的指正，以及避免偶爾笨拙的修補。

以上已說明足夠「Linus法則」。如果「Linus法則」是假的，那麼任何像 Linux 核心程式這樣複雜的系統，並且擁有像 Linux 核心程式這麼多的高手在發展，早就因溝通不良及未被發現的程式錯誤而崩塌。反過來說如果「Linus法則」是真的，那正可解釋為什麼相對地 Linux 比較沒有程式錯誤。

也許「Linus法則」並不是一個驚奇，社會學家多年前發現到在一群素質相同的觀察家中，他們共同做出的預測要比其中任一位單獨所做的要來得可信。這被稱為「Delphi[5]效應」。可見 Linus 只是把「Delphi效應」用在發展作業系統時對程式的除錯上，所以「Delphi效應」能夠克服發展系統的複雜度，即使複雜如作業系統核心[6]。

「Linus法則」也可稱之為「程式除錯可併行處理」。雖然多位程式除錯者在除錯時需要和一些程式發展者溝通協調，但是程式除錯者彼此間卻不需如此。所以增加程式除錯者並不會像增加程式發展者那樣，多出平方倍的複雜度和管理成本。

理論上造成程式除錯效率減低的原因是多位除錯者重複同一件工作，就實際的情形而言，在 Linux 的世界中幾乎不會發生這樣的狀況。「儘早，經常發表新版本」這個策略使得程式錯誤的修補回饋得很快，藉此將除錯者重複同一件工作的機會減至最低[7]。

[5] Delphi 是希臘古都，以善作預言的 Apollo 神殿而聞名。 —— 譯註。

[6] 對訓服作業系統的複雜度來說，透明與同裁審查都是很有價值的，畢竟這不是一個新概念。在 1965 年，時間共享的非常早期階段，Corbató 與 Vyssotsky，Multics 作業系統的共同設計者寫到，一般預期 Multics 將在完成後發表…這是為了兩個原因，第一是它可以經得起有興趣者的審查與批評，第二是有義務要展出，未來的設計者才可以讓內部作業系統儘可能清晰，如此會有利於發現系統問題。

[7] John Hasler 提過一個有趣的解釋，我將它稱為「Hasler定律」：重複工作花費的時間與團隊大小呈現 sub-quadratic 關係 —— 至少比那些需要被消滅的過度計畫與管理上升的慢。
這聲明並沒有否定「Brooks法則」。複雜度與除錯規模隨著人數而呈現平方上升，不過重複工作的時間成本至少上升的比較慢。從以下這個無可懷疑的事實很難發展出令人信服的推論：「不同開發者寫的代碼會造成功能限制，防止重複工作比那些形成大量臭蟲且缺乏計畫與溝通結果好多了。」
將「Linus法則」與「Hasler定律」合併可以得出三種軟體專案的開發模式。在小型專案（最多三個開發者），沒有管理比有一個主要開發者好。在中等規模的專案，傳統的管理成本相對低，在防止重複工作與除錯有正面助益。
而大型專案中，跟重複工作相比，傳統的管理成本上升的比預期的快多了。雖然跟傳統管理方式相比，人多容易發現錯誤，但是這些上升的成本對於管理這些事情卻有結構性的無力。因此，在大型專案，正面效果完全被傳統管理所抵銷。

Brooks 曾發表過一個即席的看法：「維護一個廣為人用的程式的總成本通常是發展這個程式成本的百分之四十或更多，令人訝異的是這維護成本深受使用者人數的影響，越多的使用者可以發現越多的程式錯誤。」（這正是我所要強調的）

因為增加越多的使用者，就會增加考驗程式的方法，所以使用者越多，發現的程式錯誤也越多，當使用者也是協同發展者時這種效應會再被放大，每一位使用者以不同的直覺，不同的分析工具，和不同的角度來標明程式錯誤，因為這些不同，「Delphi效應」似乎真的有作用了，在個別情況下的除錯工作，也因這些不同而減少重複出力的可能。

所以，以程式發展者的眼光看來，增加更多的 beta 版測試者也許不會減少目前藏在深處的程式錯誤的複雜度，但可以增加某位除錯者以他的工具程式找到這個程式錯誤的機會，而這個程式錯誤對這位除錯者來說是淺顯的。

Linus 也在這種方式上下了賭注。因為程式都會有錯誤，Linux 核心程式以一種特別的方式來定出版本號碼，讓使用者可以選擇要用上一個比較穩定的版本，還是選擇錯誤風險比較高的新版來使用新功能。這個策略尚未正式為大部分的 Linux 高手所採行，但是它也顯示出一個事實，就是使用者可做選擇使得這兩種版本都更有吸引力[8]。

[8] 實驗版與穩定版 Linux 可以對沖彼此的風險。這分裂形成另一個問題：截止日的死亡。當兩邊都有一個不可變動的功能清單與截止日，品質蕩然無存且會形成大混亂。我輸錢給哈佛商業評論的 Marco Iansiti 與 Alan MacCormack，因為他們向我展示了證據，就是鬆綁其中一的規定可以讓排程可行。
截止日固定但功能清單可變動，放棄到截止日仍為完成的功能，這是一個可行的辦法；穩定版 Linux 核心就是如此，Alan Cox（穩定版的維護者）相當準時的釋出新版本，但不保證特定臭蟲何時被修正，或是從實驗版引進什麼新功能。
另一個辦法是設定想要的功能名單，並只在全部完成後發布；這是實驗版 Linux 核心的作法。De Marco 與 Lister 指出這樣的排程政策（完成後叫醒我）不只品質最好，而且平均來說，跟務實與激進的排程相比，釋出的時間間隔也較短。
我懷疑在這論文的早期（2000 年初），我嚴重低估「完成後叫醒我」對社群的生產力與品質的影響。1999 年釋出的 GNOME 1.0 帶來的經驗是，對未成熟產品的壓力會抵銷一般開放原始碼產品應有的品質。
透明的過程、完成後叫醒我與開發者自我選擇，這三者是對開放原始碼的品質一樣重要。

# 第五章、有多少眼球馴服了複雜度

可以很明顯地觀察到市集模式極大地加速了除錯與程式演化。另一件可以清楚明白的是，在微觀上，開發者與測試者的每天活動中，市集模式如何與為何可以達到這樣的成果。在本章（初版完成三年後，依據開發者多次親身體驗過的洞察力），我們將仔細的檢查它的實際機制。非技術性傾向的讀者可以略過這一章，直接跳到下一章去。

一個關鍵點是，為何沒有原始碼意識的使用者所回報的錯不會太有用。沒有原始碼意識的使用者傾向回報表面上的問題，他們把自己的使用環境視為理所當然，所以他們會忽視重要的背景資料，回報錯誤時很少會包括可信賴的過程。

這裡的問題是測試者與開發者的對問題的視角不同，測試者由外向內看，開發者由內向外看。在封閉原始碼的體系中，兩者只會固守自己的角度談論事情，因而對另一方深深的失望。

開放原始碼的體系則打破這條界線，使測試者與開發者可以站在同樣的角度來討論事情，這有效率多了。實務上，這將有巨大的差異，一者是只報告表象的症狀，一者是以開發者那種以原始碼為基礎的角度來看問題。

大部分的時候，大多數的臭蟲是可以由描述開發層級的特徵來除錯的，即使是不完整的描述。當一個 beta 測試者告訴你在那一行代碼有邊界的問題時，或告訴你在 X、Y 跟 Z 的情形下，有個變數有問題，指出有問題的代碼通常就足夠找出問題並修正它。

因此，對於 beta 測試者與核心開發者來說，有原始碼意識的人對於雙方都可以強化溝通與合作。換句話說，核心開發者的時間被節省了，即使是在有很多共同開發者的情形下。

另一個開放原始碼方式的特徵是節省開發者的時間，而這是典型開放原始碼專案的溝通結構。上面我使用了「核心開發者」（core developer）這個字來區別專案核心（project core，通常很小；一個開發者是常見的，一到三個開發者則是很典型的）與專案圈（project halo）的 beta 測試者跟貢獻者（通常有數百個）。

傳統軟體開發組織的根本問題是「Brooks法則」：在落後的專案，增加越多程式師會使得專案更落後。一般的狀況下，「Brooks法則」的預測是，隨著開發者的人數增加，複雜度與溝通成本隨著人數的平方上升，而完成的工作卻只成線性上升。

「Brooks法則」建立在經驗，臭蟲會在界面頑強糾結，而程式是由不同的人寫的，過度溝通容易導致界面的數目隨著不同使用者而升高。因此，問題的規模會隨著開發者間的溝通而呈現平方上升。（精確的說，是 N x（N-1）/2，N 是開發者的數目。）

「Brooks法則」的分析建立在一個隱藏的假設：專案的溝通結構必須是完全圖（complete graph），每個人都可以跟每個人溝通。但是在開放原始碼的專案，開發者在有效平行分割的子專案中彼此很少互動；程式更動與臭蟲回報是透過核心團體來處理的，只有在這樣的小團體中，「Brooks法則」的分析才成立[9]。

還有其它原因讓原始碼層級的臭蟲回報變得有效率。事實是一個錯誤常常會有許多可能的症狀，取決於使用的的使用狀況與使用環境。這些錯誤是一些複雜與微妙的臭蟲（像是動態記憶體管理錯誤或視窗的隨意中斷），也是最難被發現或靠靜態分析來捕捉，這在長期的開發中造成最多的問題。

[9]這不完全精確，網際網路為基礎的開放原始碼專案的混合式組織特徵，依 Brooks 的建議，要解決這樣的 N^2 複雜度問題，應該採取「外科手術小組」的團隊 —— 但是現實中差異很大。像跟在領導者身邊的代碼圖書館員這樣的專門角色，實際上不存在；相對於 Brooks 當時來說，這角色被更而有力的軟體工具集所取代。而且，開放原始碼文化強烈依賴 UNIX 傳統的模組、API 與資訊隱藏 —— 沒有任何一個是 Brooks 所講的處方。

當一個測試者送出一個嘗試性的原始碼層級的多症狀臭蟲報告（例如，我看來在第 1250 行代碼有個視窗在做訊號處理，或你在哪裡把那個暫存清空），可能會給開發者一個關鍵的線索來發現半打的症狀，這些開發者通常因為太靠近底層代碼而無法發現這樣的問題。在這種案例中，很難找出可從外部看見的不正確動作是從哪個臭蟲引起的，甚至是不可能的 —— 但是透過經常發布，就不需要知道了。其他的合作者會迅速找出臭蟲是否已被修正。在很多案例中，導致不正常動作的原始碼層級臭蟲將被移除，甚至在還沒有被報告之前就被移除。

複雜的多症狀錯誤，通常有很多從表面症狀來的方式可以找出真正的臭蟲。這種能讓測試者與開發者找出問題的方式，可能與開發環境有關，也可能會隨著時間而有無法預期的變化。實際上，當測試者或開發者追蹤一個症狀時，都是在程式空間的一個集合中「半隨機」（semi-random）取樣。臭蟲越微妙複雜，越難找出相關的樣本。

對於簡單與容易複製的臭蟲，重點在於「半」（semi）而非在「隨機」（random）；除蟲技巧、對程式與架構的熟練都是關鍵。但對於複雜的臭蟲來說，重點就是「隨機」（random），這時人多比人少好 —— 即使這些少數人是平均技巧較高的。

如果從表面的症狀找出臭蟲的難度大，像是一些無法從表面症狀預測的，上述的結果會進一步增強。單一的開發者可能會以一個困難的方式來做第一次嘗試，但其實也可以從簡單的方式達到同樣的結果。另一方面，假如很多人隨著頻繁的版本一起測試，可能就會有一個人可以用最簡單的方式找到臭蟲，節省了大量的時間。專案管理者將會發現，隨著新版本發布，許多人一起用各種複雜方法追蹤同一隻臭蟲的時代將會過去，尤其是在眾人浪費太多時間之前[10]。

[10] 反對者跟我說，對同一隻臭蟲的多症狀來說，描述一隻臭蟲特徵的難度呈現指數式（譬如 Gaussian 分配或 Poisson 分配）上升。如果可能掌握到分配的外觀，那會是很有用的資料。這與除錯困難度是水平機率分配密度的差別很大，也就是說單一開發者也應該模仿市集模式，針對單一的臭蟲定一個時間上限，超過的話就追蹤下一隻臭蟲。從一而終不見得是美德…

# 第六章、今花非昨花？

由 Linus 行為的研究中，我們得到了一個能解釋他為什麼成功的理論，所以我想要在我的新專案（當然不如 Linux 核心程式的複雜和雄心勃勃）中來測試這個理論。

但我做的第一件事情是大力重組和簡化 popclient 的程式，Carl Harris 的實作非常紮實，可是卻像許多的 C 程式師一樣，含括了一種不必要的複雜，他以程式碼為主，資料結構為輔，因此程式碼看起來漂亮，但資料結構卻很特殊，甚至可以說是醜陋的（至少以這位老資格 Lisp 高手的高標準而言）。

然而，我重寫程式除了改良原來程式碼和資料結構的設計外，還有其他目的，就是把它發展到我可以完全瞭解，否則負責修補你不懂的程式是一件很無趣的事。

專案進行的第一個月，我簡單地依循著 Carl 原來基本設計的用意，第一個重大的改變是我加入 IMAP 協定的支援，我重組原來處理協定的程式，改成一個較為通用的驅動程式再加上三個驅動它的方法表（即 POP2，POP3 和 IMAP）。這個改變闡釋了一個廣義的原則，特別在像 C 這種先天上未提供動態資料型態的程式語言，程式師們最好謹記在心：

**格言9：聰明的資料結構配上笨拙的程式碼要比相反的組合好。**[11]

Brooks 在《人月神話》的第九章中也說：「光給我看你的程式碼，而不給我看它用的資料結構，我會一頭霧水。給我看你程式的資料結構，我通常不需要再看你的程式碼，因為已經夠明白了[12]。」

1996 年的九月初，從零開始工作約過了六週，我開始在想是否要幫 popclient 取個新名字，畢竟 popclient 已不僅僅是單純的 POP 協定客戶端程式，但我遲疑了，因為 popclient 的設計並無真正重大的改變，我的 popclient 尚須發展出自己的特色。

當 popclient 可以把 fetchmail 抓下來的信直接轉送到 SMTP 的接收埠時，它徹底的改變了；至於 fetchmail，我稍待會再說明。我之前說過要用這個專案來測試關於 Linus 成功的理論，也許你會問我到底要怎麼做呢？我用下面幾個辦法：

- 我儘早並經常發表新版本（幾乎至少每十天就發表一次，甚至在發展的高峰期，一天一次）。
- 對於每一位與我討論 fetchmail 的人，我把他們列入 beta 測試者的名單，所以名單越來越長。
- 每當我發展出新版本，一定發出像聊天般的通知給 beta 測試者名單上的人，鼓勵他們一起來參與這個專案。
- 而我也總是傾聽 beta 測試者的心聲，詢問他們對於這個程式的設計上有無意見，並且回應他們送來對程式的修補和回饋。

在採用上述的辦法後，立即就得到了報償，自從這個專案開始以來，我所收到關於程式錯誤的回報，其品質足以令許多的程式發展者羨慕，這些回報甚至還常常附上不錯的修補辦法。因而我做了關鍵性的思考，我收到了使用者的來信，得到了關於新增智慧型功能的建議。這說明了:

[11] 相反的組合指笨拙的資料結構配上聰明的程式碼。 —— 譯註。

[12] 事實上，上述的這段話他是用「流程圖」（flowchart）和「表格」（tables）這兩個名詞，但由於三十年間專業術語/文化的變遷，這些名詞的意義幾乎是相同的。 —— 譯註。

**格言10：如果你視 beta 版測試者如同你最珍貴的資源，那麼他們會以此做為回報。**

Fetchmail 達到成功的方法中，有趣的是一張薄薄的 beta 版測試者名單，也就是 fetchamil 之友的名單，當我在寫這支程式時，有 249 位，然後每週增加 2 到 3 位。

這張名單中的成員人數最多時幾乎到達三百，不過，當我在 1997 年五月底審訂這張名單時，其中的成員已經因為一個有趣的原因而開始減少，好幾位告訴我他們要停訂「fetchmail之友」，因為他們覺得 fetchmail 已能滿足他們的需求，已經不再需要收到「fetchmail之友」。也許這是成熟的市集模式專案的正常生命週期中的一部份。

# 第七章、Popclient 變成 Fetchmail

這個專案真正的轉捩點發生在 Harry Hochheiser 送給我他寫的一部份程式，這部份的程式會把郵件轉送到客戶端機器上 SMTP 的接收埠，我立即瞭解到這個特色若有穩定的實作，那麼 fetchmail 中其他的郵件傳遞模式都可以廢除了。

有幾個禮拜，其實我一直在扭曲 fetchmail 而不是真的改進它，因為它使用介面的設計雖然能提供服務，但卻不夠高雅，並且有太多非必要的選項成為整個程式的累贅，尤其是要把取回的郵件存成郵件檔或輸出至螢幕的選項對我造成了相當的困擾，可是我卻也說不出個所以然來。

當我思考郵件改由 SMTP 轉送這個作法時，才發覺到原來的 popclient 包攬太多事了，過去它被設計成郵件轉送代理（MTA）兼郵件遞送代理（MDA），若藉由 SMTP 轉送郵件，那它可以完全不管郵件遞送，單純地負責郵件轉送，只要把郵件轉給像 sendmail 這樣的郵件遞送程式就可以了。

在有支援 TCP/IP 通訊協定的平台，幾乎可以保證第 25 號埠（SMTP用）早就在那裡等了，為什麼還要和設定郵件遞送代理組態或設定郵件檔的上鎖附加模式這些問題糾纏呢？尤其這樣做可以保證取回的信件看起來像發信人透過SMTP傳送一樣，而這正是我們想要的。

在這裡給我們上了好幾課，第一課是，這個透過SMTP轉送的巧思是從我仿效 Linus 的方法以來，所得到最大的收穫，一位使用者提供了絕佳的主意，而我所必須做的已經蘊涵在其中。

**格言11：體認你使用者提供的巧思，以獲取好點子，有時候越後到的越好。**

你將會發現一件很有趣的事：如果你很誠實並很自謙地知道你欠人多少，那麼全世界都會認為你發明了全部，而且對你先提出的天才創作，也會以為你非常謙虛，這些我們可以在 Linus 身上得到印證。

（1997 年 8 月的時候，當我在 Perl 會議上發表這篇論文時，Larry Wall 坐在前排，我唸到上一行時，他叫了出來，以一種復興宗教的神情，喊著：「兄弟，告訴他們，告訴他們吧！」，全場的聽眾都笑了，因為他們知道這也發生在這位 Perl 的原創者身上.）

我以同樣的精神進行這個專案，經過短短幾週的時間，我開始得到類似的讚美，這些讚美不只來自 popclient 的使用者，也來自該得到這種讚美而卻未得到的人，我保留了一些感謝函，也許當我懷疑我人生的意義為何時，可以再看看這些信。

但除些之外，這裡還有兩課更基礎，不具政治性，更適合所有設計的一般情形：

**格言12：通常，最適切和最有創意的解題法來自發覺自己對問題原先的觀念是錯誤的。**

我曾試著去解一個錯的問題，就是延續 popclient 既是 MTA 又是 MDA 的設計，把它發展成有各種的本地端遞送模式。Fetchmail 的設計需要重新思考，應該只要單純地做一個 MTA 程式，成為網際網路正規的 SMTP 郵遞路徑中的一段。

當你在發展程式的過程中撞到障礙時 —— 也就是當你發現很難想出下一步要怎麼修補時，通常是反省的時候了，但不是問是否已找到正確的答案，而是我們提出正確的問題了嗎？也許問題需要再重新整頓一番。

是的，於是我重新整頓了我的問題，很明顯地，該做對的事有：（1）在原來通用的驅動程式中，加入轉送郵件至 SMTP 接收埠的功能。（2）當它成預設模式。（3）丟棄其他遞送模式的程式碼，尤其是遞送至郵件檔及遞送至標準輸出。

我對第（3）步遲疑了一些時候，因為擔心會嚇走長久以來 popclient 的使用者，因為他們一直倚靠另一種遞送機制，理論上他們可以立即以 .forward 檔來達到同樣的效果而不靠 sendmail 程式，事實上這個轉換可能含糊不清。

但當我真的去做，結果證明益處極大，popclient 驅動程式中的一段可以消失了，設定也變得簡單多了 —— 不用再屈就系統的 MTA 程式及使用者的郵件信箱檔，也不再需要擔心底層的作業系統是否支援檔案上鎖。

而且唯一丟掉郵件的可能也不見了，如果你指定要把郵件送到某個檔案而磁碟空間又滿了，那麼你的郵件就丟掉了，然而由 SMTP 轉送信件的話，則不會發生這種事，因為 SMTP 的接收者除非將信息送達，或至少先暫存起來待會再送，才會回覆成功給發信者。

並且效能也改進了（如果只跑一次，你大概不會有感覺）。另一個有意義的好處是使用說明變得更簡單了。

稍後，為了要處理某些模糊的狀況，如動態 SLIP，我必需讓使用者可以指定本地端要用那一個 MDA 程式來送達郵件，我發現了一個更簡單的方法。

這寓意是什麼呢？當你可以丟掉程式中老舊的特色而又不失掉效力，那就別遲疑。Antoine de Saint-Exupery[13]（當他還不是經典童書的作者前，他當過飛行員和飛機設計師）曾說：

**格言13：設計上完美，不是「沒有東西能再被加入」，而是「沒有東西能再被移出」。**

當你覺得做對了，那麼你的程式碼越來越好，越來越簡潔，在這個過程中，fetchmail 的設計終於和先前的 popclient 不同了，而有了自己的特點。

該是這個程式改名字的時候了，新的設計看起來比舊的 popclient 更像 sendmail，新的 popclient 和 sendmail 都是 MTA，只是 sendmail 把郵件「推」出去給 SMTP 收信程式，再送達使用者，而新的 popclient 則是把郵件「拉」回來給 SMTP 收信程式，然後再送出，所以兩個月後，我把它更名作「fetchmail」。

[13]「小王子」就是 Antoine de Saint-Exupery 的作品。這句格言也曾在《*Modern Operating System*》一書中被 Andrew S. Tanenbaum 引用來說明作業系統微核心的設計哲學。 —— 譯註。

# 第八章、Fetchmail 成長了

我把 fetchmail 設計得雅潔而新穎，程式本身也跑得很好，因為我天天都在用，並且開始有一些 beta 版測試者加入，這情形使我逐漸瞭解，我已不再是為了可能讓少數其他人能得到一些便利，而在進行用處不大的程式精解，我已經為每一位有台 UNIX 機器，上面跑 SLIP/PPP 來取得電子郵件的玩家們，寫了一個真正滿足他們需要的程式。

因為藉 SMTP 埠轉送郵件的這個特色，潛在使得 fetchmail 足以成為同領域的殺手級程式，在同類的典型程式中，它已經夠資格佔到適當的位置，使得其他程式不是被捨棄就是幾乎被遺忘。

我認為沒有人能真的看準或計畫會有這樣的結果，以有力的設計想法投入這個專案，之後的結果似乎是無可避免的，自然的，甚至是注定的，要追求像這樣好的想法，唯一的方法就是先擁有許多的想法，或者以工程上的判斷去取得別人好的想法，而在此處這個想法的利用已超出原創者的想像。

Andrew Tanenbaum 在 386 上造出了一個簡單的原生 UNIX 系統，他原先的想法只是用來作為教學的工具，但 Linus Torvalds 把這個 Minix 系統的觀念拓展開來，更進一步，已經超過 Andrew 當初能夠想像到的發展，並且成長出一些令人讚嘆的事物。我用一樣的方式（雖然規模較小），由 Carl Harris 和 Herry Hocheiser 那裡得到一些想法，然後把它們發揚光大。在人們的想像中，歷史上的原創者都是天才，而我們兩位都不是，但是大部份的科學發展和軟體發展工作，完成者不是天才原創者，反而是行家們。

Linux 和 fetchmail 的成果都是一樣令人興奮，事實上這就是每一位高手追求的成功，這些成果指示我應該把標準設得高一點，把 fetchmail 發展到我所能想像的理想，我已不只是為自己的需求而寫，也為其他人所需要的特色而寫，並且還要同時保持程式簡單和強健。

第一個加入的重大特色是「多人共用一信箱」的支援 —— 這個功能可以抓下累積在同一個群組信箱中，而屬於不同使用者的信件，然後再分送給原來的個別收信人。

我決定加入「多人共用一信箱」的支援，部分是因為有一些使用者們嚷嚷著他們需要它，但主要是因為我認為它會迫使我以更通用的法則來處理郵件頭的地址，並藉此除去「一人一信箱」功能程式碼中的錯誤，而我的確也做到了。為了讓程式能按 RFC 822[14] 中的規定來檢查信息的語法，花了我相當長的時間，不是因為規定的個別片段難以理解，而是因為它包括了成堆相互依賴的瑣碎細節。

結果「多人共用一信箱」的支援的確是一個漂亮的設計，我是怎麼知道的呢？

**格言14：任何的工具以我們所知道的方法來使用都會有用，但一個真正了不起的工具會以你從未想過的使用方法來發揮它的功能。**

支援「多人共用一信箱」的 fetchmail 有一種意料外的使用方法，就是在以 SLIP/PPP 連線方式連上 ISP 的客戶端執行「郵遞討論名單」（mailing list），因為它可以配合客戶端的多使用者共用 ISP 上同一信箱（用別名 —— alias —— 的方法），讓每個使用者都能在名單上，這表示我們可以在個人的電腦上，透過一個 ISP 的帳號，來維護一個郵遞討論名單，而不必持續連著 ISP。

另一個來自我 beta 測試版的使用者的重要需求是接受 8 位元 MIME 郵件格式，這很容易做到，因為我過去一直都小心地保持每一個字元碼都是完整的 8 位元，並非我未卜先知，而是我遵從另一條法則：

**格言15：寫作任何的通信閘軟體時，要盡可能地不去擾動到通訊的資料流 —— 並且絕對不要丟掉其中任何的資訊，除非接收方強迫你這麼做。**

[14] RFC 822 是 Standard for the Format of ARPA Internet Text Messages。 —— 譯註。

如果我當初沒有遵從這項原則，那麼 8 位元 MIME 的支援勢必難以加入並多錯，所以我需要做的只是研讀 RFC 1652[15]，然後加入顯然得知的程式碼以產生 MIME 的標頭。

一些歐洲的使用者要我在程式中加入選項，用來限制每次連線取回郵件的數目（這樣他們才能控制昂貴的電話線路連線花費），我拒絕了許久，甚至到現在，我對這個選項的加入仍感到不太愉快，但假如你是在為全世界寫程式，那麼你就應該聽取你客戶們的意見 —— 這個原則不會改變，因為他們會以金錢之外的形式給你報酬。

[15] RFC 1652 是 SMTP Service Extension for 8bit-MIME Transport。 —— 譯註。

# 第九章、由 Fetchmail 學到的一些經驗

在我們回頭討論一般軟體工程的議題前，由 fetchmail 得來的一些特殊教訓值得深思。

Fetchmail 設定檔（rc file）的語法包括可要可不要的關鍵字，這些如「噪音」般的字眼會被語法分析程式忽略，這些字的加入，使得設定檔的語法和英文很接近，和傳統中簡潔的「關鍵字 —— 設定值」配對表示法（把噪音字去掉就可以得到）比較起來，要來得容易閱讀。

這個教訓開始於某一個夜晚的實驗，我注意到設定檔中的宣告語法可以組成一個迷你的祈使語言。（這也是為什麼我把原來 popclient 中的關鍵字 server 改成 poll）。

對我而言，如果把這個祈使語言弄得更像英文，那會讓 fetchmail 更易於使用，雖然我現在信服如 Emacs、HTML 及許多資料庫引擎制定出來的模範語言，在正常情況下我也不那麼迷英文的語法。

傳統的程式師喜歡非常精確而簡潔的設定語法，不希望有任何冗餘在其中，這是因為早期的電腦計算資源昂貴而遺留下來的觀念，他們認為語法分析的過程要節約計算資源，並要儘可能的簡單。英文的語句大約有百分之五十的冗餘，所以不利於做為設定語法。

但這並非我在正常的情況下不用英文語法的原因，我在此提及是為了推翻像英文的語法不適合作設定語法的論點，當中央處理器和記憶體都變得便宜時，設定語法的簡潔已不再是我們的目標，現在的情況是：一個電腦語言中符合人性易於使用的重要性已超過節約電腦的計算資源。

然而還是有好幾個地方要小心，其中之一是語法分析過程變得比較複雜的代價 —— 你不會想把這個特點（使用像英文的設定語法）變成程式錯誤的來源及使用者的疑惑處。另一個是當我們要訂出一個像英文的語言時，通常需要做些修改，表面上看起來修改過後的語法可重組出自然語言，但可能也會因修改過以致於和傳統的設定語法一樣，令人感到疑惑。（我們可以在許多所謂的第四代程式語言和商業用的資料庫查詢語言看到這個現象）

Fetchmail 的控制語法看起來免除了這個問題，因為我嚴格地限制控制語法的範圍，它不是一個以通用為目的的語言，它簡單並不很複雜，所以在極小的英文子集和真正的控制語言之間的相混處很少，我認為這裡還有一個更廣義的教訓：

**格言16：當你設計的語言沒有一處是 Turing-complete，你可以採用比較平易的語法。**

另有一個教訓是關於含糊的保密性，有一些 fetchmail 的使用者要我修改程式，以把經保密處理的通行碼儲存於設定檔內，這樣子一來，偷窺者就沒辦法看到真正的密碼了。

我並沒有這麼做，因為這個做法並不能達到真正的安全，能拿到讀取你設定檔權限的人，也能以你的身份執行 fetchmail —— 如果你的通行密碼經保密處理存在設定檔中，他們可以由 fetchmail 中取得解碼程式來得到你真正的通行密碼。

如果我把程式改成可以在 fetchmail 的設定檔中儲存保密的通行密碼，那會給認為這並不難的人們對保密性的錯誤觀念。一般的守則應該是:

**格言17：一個保密系統是否安全依存於它隱藏的秘密，注意不要有「虛擬秘密」。**[16]

[16]以 fetchmail 為例，隱藏的秘密是指「通行密碼」，「虛擬秘密」是指把通行密碼編碼後存於設定檔中。 —— 譯註。

# 第十章、市集模式必要的條件

早先看過這篇論文的書評家和試讀者都提出同樣的問題，那就是以市集模式發展軟體，獲得成功的先決條件為何？包括專案領導人的資格，程式碼到什麼樣的程度，才對社群發表並開始成立協同發展團隊。

相當明顯的，任何人無法以市集模式建立軟體基礎[17]，但是可以用市集模式來測試，除錯，改進軟體，在一個專案的起始點很難運用市集模式，Linus 沒試過，我也沒有。專案初始的協同發展團隊需要有一些東西可以測試，可以執行。

當你開始召募專案團隊時，你必須能提出大致合理的保證，你的程式不需要運作得很好，它可以暴力，有錯，不完整，及註解貧乏，只要它可以使潛在的協同發展者相信，這個程式在可預見的未來大有可為。

Linux 和 fetchmail 公開發表時都具有強健及吸引人的設計，許多人認為這和我所報告的市集模式有所不同，並以為這樣的設計很重要，甚至進一步做出一個他們的結論 —— 高度的設計直覺和聰明是一個專案領導人不可或缺的特質。

但是 Linus 的設計由 UNIX 而來，我的設計由原先的 popclient 而來（雖然它後來改變甚大，比例上說來還超過 Linux），所以市集模式專案的領導人或協調者真的需要格外的設計技巧？或者能提昇別人的設計技巧呢？

我想對於專案協調者是否能做出耀眼的設計並不重要，最重要的是協調者是否能認知別人在設計上的好點子。

Linux 和 fetchmail 都證明了這件事，Linus 這位仁兄如同之前所討論的，他並不是偉大的創新設計師，但他展現了另一種卓越的技巧，即認同別人好的設計點子，且將這些好的設計整合到 Linux 的核心中。而我之前描述過 fetchmail 最有力的設計（藉 SMTP 轉送郵件）也來自他人。

---

[17] 一個人能不能採取市集模式從無到有的發展專案，取決於市集模式是否能夠支持創造性的工作。有人認為，缺乏強而有力的領導力，市集模式只能在目前最尖端的技術上做複製與小幅度改良，而無法發展出最尖端的技術。這也許是 Halloween Documents 最無恥的論述，這令人尷尬的兩份微軟內部文件如此描述開放原始碼現象。作者比較了這個類似 UNIX 系統的 Linux 系統是在「追尾燈」（chasing taillights）（當一個專案達到最先進技術的門檻時），認為管理才能進一步擴大前進。

如果我們把開放原始碼替換為 Linux，會發現這跟新的情境差很遠。在過去，開放原始碼社群不會藉著追尾燈或管理制度發明 Emacs 或 WWW 或網際網路 —— 但目前確有非常多創新是採用開放原始碼模式。GNOME 專案就是最先進的 GUI，而且在 Linux 社群外也引起相當的注意。還有其他不勝枚舉的例子，有興趣的人可以去看一下 Freshmeat 等等的專案。

認為教堂模式（或市集模式，或其他種類的管理結構）能夠讓創新更值得信賴有個更根本的錯誤。這完全是無稽之談。烏合之眾沒有突破性的洞察力 —— 即使市集模式的無政府論者沒有真正的原創，這使得目前還能生存的企業人士可以賭注維持現狀。*洞察力來自於個人*。大多數圍繞在這些人身邊的社群機制，都希望對於突破性的洞察力更有回應 —— 滋養與嚴苛的測試，而不是壓榨它們。

一些人會說這是不切實際的觀點，一個典型過時獨創者的回歸。並非如此，我不是斷言這些已出現的人沒有突破性的洞察力做開發，而是我們了解同裁審查對於高品質結果是必要的。當然，這些發展的起源來自於一個人的好點子 —— 而且這是必要的火花。

因此，創新的根本問題（不管是軟體或是其他方面）是如何不壓榨它 —— 或者是更根本的，如何讓這些有洞察力的人群越來越多。 推論教堂模式可以做到而低進入門檻的市集模式做不到，這是可笑的。一個是可以讓眾人合作的社會環境激勵創新；而等級制度下的創新卻需要做一些政治性的行銷才能為自己的想法開發，不然會有被開除的風險。

如果我們看一下採用教堂模式的軟體開發歷史，會很快的發現那相當稀少。大型機構依賴大學的研究中心來發展新的點子（因此 Halloween Documents 的作者對於 Linux 的快速開發成果感到相當感冒），或者買一些擁有創新者成果的小型公司；這兩種都不是教堂模式，實際上，很多的創新就是被 Halloween Documents 所讚揚的管理方式扼殺的。

那是負面的部份，讀者應該著重在正面的部份。以實驗的精神，我提議以下的方式：選一個你一直信仰的原創原則，例如「當我看見就能了解」。選一個跟 Linux 競爭的封閉原始碼作業系統，與一個公認在這上面運行的最好的原始碼。觀察該原始碼與 Freshmeat 一個月。每天計算在 Freshmeat 上原創的成品，然後對比你選的原始碼的原創成品的數字。三十天後，統計兩者的數據。

當我寫本文的時候，Freshmeat 發布了 32 個，我選的原始碼發布了 3 個，這在 Freshmeat 算慢的，但經驗上 3 這個數字對封閉原始碼來說是相當快的。

這篇論文的早期讀者指出：我有低估市集模式專案中創造力的重要性，因為我自己本身就已具備了許多的創意，所以將此視為理所當然。這真是抬舉我，也許這說法有幾分真實，相對於寫程式及除錯，設計應該是我最強的本領。

但以聰明和創意來設計軟體的問題在於「習慣的養成」 —— 當你應該保持程式的強健和簡潔時，反而把它弄得花俏而複雜，我曾因犯了這個錯以致於把專案搞砸了，但我在 fetchmail 這個專案中小心地控制，避免發生這種錯誤。

所以我相信 fetchmail 專案的成功部分的原因是我防止設計上「聰明」的傾向，這個論點（至少）已經反駁了設計上的創意是市集模式專案成功的基本條件。以 Linux 來說，假設 Linus Torvalds 在發展程式的過程中，試圖在作業系統的基本設計上力求創新，那麼我們現在已有的 Linux 核心程式會如此穩定和成功嗎？

當然，任何想起始一個市集模式專案的人，應該具備基本程度的設計能力和寫程式技巧，但我認為如果他們有認真想過，那他們的程度應該已在低標之上。開放性原始碼社群對於名譽的重視，給予其中的人們一種微妙的壓力，如果無法勝任專案後續的發展，那麼就不會想去起始，直到目前，這種慣例似乎仍運作得很好。

還有一種技巧，通常與我們不會把它和軟體發展聯想在一起，但我認為這和市集模式專案中聰明的設計一樣重要，也許還更重要，就是市集模式專案中的協調者或領導人必須有人緣和好的溝通技巧。

這應該很明顯，為了召集發展社群，你需要吸引人們，讓他們對你所做的有興趣，並且保持他們加入後工作愉快。技術上的末節很難達成這樣的目標，更難以完成整個專案，你個人的人格特質也和專案習習相關。

Linus 是一位好人，令人喜歡並樂於幫助他，這不是巧合，我精力旺盛，活潑外向，喜愛為群眾們工作，並具有喜劇演員的本能，這也不是巧合，為了讓市集模式專案順利運作，如果能用一點小技巧來吸引人們，那幫助會很大。

# 第十一章、開放性原始碼軟體的社會關聯性

我們談過：最好的程式起自於作者個人要解決他每天的切身之痛，然後因為這通常也是許多人的痛處，所以這個程式便開始散佈，這讓我們把**格言1**用另一種更有用的說法來陳述：

**格言18：為了要解有趣的問題，開始找你感興趣的問題吧！**

所以 Carl Harris 寫了早期的 popclient，而我寫了 fetchmail，然而這句格言應早已為人所知許久，Linux 和 fetchmail 發展的歷史似乎要讓我們注意到這有趣的一點，就是軟體發展的下一步 —— 使用者和協同發展者組成了龐大而活耀的社群，帶動軟體的演化。

在《人月神話》一書中，Fred Brooks 觀察到：程式師的時間具不可替換性，在一個進度已經落後的專案中，加入更多的發展者只會使進度更加落後，他討論到一個專案的複雜度和人員間溝通的代價以參與發展人數的平方倍成長，而完成的進度只隨人數做線性成長，這個聲明自從發表以後，就被稱為 Brooks 定律且被視為真理，但假如 Brooks 定律主宰了一切，那麼根本就不可能有 Linux 這個作業系統。

Gerald Weinberg 的經典著作《計算機程式寫作之心理學》（The Psychology of Computer Programming）中有後見之明，提出對 Brooks 定律極為重要的修正，在 Weinberg 對「不自我中心的程式寫作」（egoless programming）的討論中，他觀察到在軟體工作室中，如果程式師們不會只「自掃門前雪」，反而鼓勵其他的程式師去看他們的程式，以找出錯誤或提出改進的建議，那麼程式改善的速度會遠超過每個程式師只顧自己的部分。

也許 Weinberg 的術語選擇不當，以致於原本該為人所接受的觀念卻未被接受，當想到用「不自我中心」（egoless）來形容網際網路上的電腦行家，令人發出微笑，但我認為他的論點在今天要比過去更有力。

UNIX 的歷史其實早已準備好我們從 Linux 學到的東西（以及經由我實驗得證的事情，這個實驗經過仔細地仿效 Linus 的方法[18]，但規模較小），當大家還認為寫程式仍然是單打獨鬥的行為時，真正了不起的行家已經在善用社群的注意力和腦力。在一個封閉專案中的程式發展者，他們只單靠自己的頭腦，所達成的進度將落後於知道怎麼搞一個開放專案的發展者，因為在開放的專案中，有數以百計的人從事回報錯誤及改進程式。

但是傳統 UNIX 的世界因為好幾個原因，以致於無法把這個方法的功效發揮到最大，其中包括：不同版權的法律限制，商業機密，市場上的利益等等，還有一個原因（算是後見之明）就是：當初的網際網路還不夠發達。

在廉價的網際網路來臨之前，有一些地域性的社群集中在一起，他們的文化鼓吹著 Weinberg 的「不自我中心」的程式寫作，其中的程式發展者可以輕易地吸引許多有技巧的建言者和協同發展者，如貝爾實驗室，麻省理工學院人工智慧實驗室及加州大學柏克來分校 —— 這些地方都是創新的來源，都帶有傳奇性的色彩，至今都仍具有影響力。

---

[18] 現在我們有一個比 fetchmail 更有指引意義的市集模式專案，EGCS，實驗性的 GNU 組譯系統。
這專案是在 1997 年 8 月中發布的，作為一個早期《教堂與市集》的嘗試。因為該專案的發起者覺得 GCC 的發展已經迂腐了。在之後大約二十個月，GCC 與 EGCS 各自平行發展 —— 兩者的開發者都來自一樣的母體，都源於一樣的 GCC 基礎原始碼，使用高度雷同的 UNIX 工具集與開發環境。唯一不同的是，EGCS 採用市集模式開發，而GCC 採用類似教堂模式的開發。搭配封閉的開發者團隊與發布間隔較長。
這很接近可控制變因的實驗，我們想知道的是哪一個比較戲劇性。在這幾個月，EGCS 在功能上大幅度領先，較好的最佳化與對 FORTRAN 跟 C++ 的支援。很多人都認為 EGCS 的發展狀況比同時期的 GCC 好，而且主要的 Linux 發行版都換到了 EGCS。
在 1999 年 4 月，自由軟體基金會（GCC 的官方支持機構）解散了 GCC 原來的開發團隊，將 EGCS 的團隊納入官方支持。

Linux 是第一個致力於把全世界當成是它智庫的專案，我並不認為在 Linux 的孕育期恰好誕生了全球資訊網是個巧合，而且在 Linux 發展的早期，網際網路服務供應商的事業正在起飛，網際網路的商機主流正在爆發，Linus 是第一位學習在網際網路普及的情況下，如何進行新遊戲規則的人。

雖然廉價的網際網路是 Linux 模式演進的必要條件，但我認為它並非充分條件，另一個極重要的因素是專案中領導人的領導風格，以及一群合作的客戶，其中有人會被發展者吸引而成為協同發展者，進而把網路媒體的功能發揮到最大。

但我們要問什麼是領導風格？客戶是那些人？他們之間的關係並非依賴權力而建立 —— 或者就算是，強制的領導風格沒辦法帶給我們今天這樣的成果，Weinberg 引用十九世紀俄國無政府主義者 Pyotr Alexeyvich Kropotkin 自傳中 < 紀念一位革命家 > （ Memoirs of a Revolutionist ）的一段，來解釋這個問題：

> 因自幼在擁有佃農的地主家庭中長大，當我的生命開始活躍時，就像那時所有的年輕人一樣，我們非常相信命令，指使，責罵以及處罰等等行為的必要性，但當我早期必須管理正式的企業時，我要和自由的人打交道，任何錯誤可能最後都會引發嚴重的後果，我才開始欣賞命令和規定與建立共識兩種行為間的不同之處，前者在軍事體系中效用極佳，但在真實的生活中卻一文不名，真實生活中的目標，是在許多人同心協力下達成的。

「許多人同心協力」（ severe effort of many converging wills ）正是像 Linux 這樣的專案所需要的 —— 「命令法則」（ principle of command ）並不適用於被我們稱為網際網路的無政府主義者志願者，為了更有效的運作和競爭，想要領導與他人合作的專案的電腦行家，必須學習如何吸引和激勵有興趣的社群，並且是在 Kropotkin 所建議的「共識法則」（ principle of understanding ）的模式下進行，他們也必須要學習去使用「Linus法則」[19]。

稍早我提及「Delphi效應」是為了它可能可以解釋「Linus法則」，但在生物學和經濟學中的自我調適性系統中，有更多類似的地方可以更有力地印證它，Linux 的世界從許多方面看來，像是一個自由的市場或生態，由一群個體所組成，這些個體以一種自發性的自我更正程序，試著去發揮他最大的功用，所發揮出來的功用比起集中式的規畫要來得更精巧，更有效率，這種方式正是在尋求「共識法則」（ principle of understanding ）。

Linux 行家們發揮到最大的功用不是典型的經濟價值，而是在行家中得來無形的自我滿足和榮譽，（你也許可以認為他們的動機是「利他」，但這忽略了一項事實，就是利他主義只是利他主義者自我滿足的一種型式），以這種方式進行的志工文化並非真的不尋常，就我長期參加的一個科學小說迷俱樂部來說，它不像電腦行家迷俱樂部那麼明顯地以「egoboo」（ ego-boosting 或在同好圈裡增強某人的信譽）做為驅動志工的力量。

---

[19]當然，Kropotkin 的評論與「Linus法則」引起關於社會組織控制學的廣泛議題。另一個軟體工程的理論則建議「Conway 法則」 —— 「如果你有四組人做組譯器，你會得到一個四重（ 4-pass ）組譯器」。原始的敘述比較一般化：「組織會以自己內部的溝通結構設計系統」。更簡潔的說，「方式決定結果」或「過程變成產品」。

值得注意的是，開放原始碼社群的組織形式與功能才很多層次相配合。到處都是網狀架構，不只是網際網路如此，人們在分散、鬆散與同裁的結構工作，其中也會發生延遲與降級。在這樣的環境中，每一個節點都需要其他節點自願性的配合才能互動與合作。

對於社群令人震驚的生產力來說，同裁合作的部份是必要的。關於 Kropotkin 要處理的權力關係，「SNAFU原則」有進一步的論述：「平等才可能有真正的溝通，因為對於劣等者來說，講善意的謊言比陳述事實更有吸引力。」開創性的團隊合作依賴於真正的溝通，在權力的展示下反而會被嚴重拖累。開放原始碼社群就是這樣的情況，這將在除錯、低生產力與機會消耗異常巨大的成本。

此外，「SNAFU原則」預測在需要授權的組織中，決策者與現實會慢慢脫節，對決策者的資訊太多會變成善意的謊言。這情形在傳統的軟體開發很容易見到，劣等者有強烈動機隱瞞、忽視與簡化問題。當這過程變成產品是，那會是一個大災難。

Linus 在 Linux 專案中，成功地坐上專案守門員的位置（這個專案大部份的工作都由其他人所完成），也成功地培養專案的利基，直到它可以自我維持，這顯示 Linus 精確地抓住 Kropotkin 所說「建立共識」的精神，用這個似經濟學的觀點來看 Linux 的世界，讓我們知道共識是如何作用的。

我們可以把 Linus 的方法，視作一條開創有效率市場的路 —— 以最強韌的方式聯合個別的行家來達成困難的目標，這些目標只有在持續的合作下才能達成。在 fetchmail 的專案中（雖然規模比較小），我已經展示出 Linus 的方法可以用在別的專案，並同樣有好的成果，或許我有意地，更有系統地利用他的方法。

許多人（尤其是在政治上不信任自由市場的）以為重視自我的個體文化會造成分裂，自掃門前雪，浪費，私密，和敵對，只要舉一個簡短的實例，就可以很明顯地證明這個看法是錯的，這個例子就是 Linux 相關的說明文件的多樣，品質，和深度都相當令人驚訝，程式師不喜歡寫說明文件似乎是金科玉律，但 Linux 的行家們是如何寫出這麼多的文件呢？很明顯地，Linux 重榮譽的自由市場運作得要比商業軟體生產者重金投資的說明文件撰寫公司好。

Fetchmail 和 Linux 這兩個專案都展示出藉由適當地回報許多行家，優秀的發展者或協調者能利用網際網路，獲得許多協同發展者，但不致讓專案因混亂而失敗，所以針對 Brooks 定律，我提出以下的反駁：

**格言19：假如專案發展協調者擁有至少跟網際網路一樣好的媒體，而他也不靠強制力來領導，那麼一群人必定勝過一個人。**

我認為開放性原始碼軟體的將來屬於知道如何進行 Linus 遊戲規則的人，屬於離開教堂擁抱市集的人，這並不是說個人的眼光和明智不再重要，而是我認為開放性原始碼的軟體的優勢將屬於一種人，他以個人的眼光和明智開始一個專案，並且之後能有效地號召有興趣的志工群來加入他的專案。

也許不只是開放性原始碼的將來，非封閉性原始碼的發展者也能吸引 Linux 社群的智庫到他們的問題上，很少有人付得起 fetchmail 專案中超過兩百位的貢獻者。

也許最後開放性原始碼文化會贏，但不是因為合作是善的，或軟體「柵欄」（hoarding）是惡的（假設你相信後者，但 Linus 和我則否），而是因為封閉原始碼的世界無法在演化的角力中勝過開放性原始碼的社群，開放性原始碼專案投入純熟的人時要比封閉性原始碼的專案來得多。

# 第十二章、管理與馬其諾防線

1997年版的《教堂與市集》這篇文章的結論是 —— 網路上由程式師（或者該說是無政府主義者）形成的快樂游牧民族，正以銳不可擋的氣勢衝擊著傳統封閉軟體的階層式世界。

仍有許多懷疑論者不能信服，但他們提出來的問題應該受到公平的審視，對市集論點最主要的反對意見可歸結為：市集的支持者低估了傳統管理的生產力乘積效應（productivity-multiplying effect）。

傳統的軟體發展經理人，反對的論點在於開放原始碼專案的偶然性（casualness），專案團隊在偶然中成立，改組，和解散，雖然開放原始碼社群比起任何封閉發展者擁有人數上明顯的優勢，但偶然性卻否定了這個優勢。他們目睹軟體發展需在時間上持續付出代價，並且要看客戶是否繼續投資重量級的產品，而不是有多少人把骨頭丟到鍋裡等待它熬成湯。

這個論點有一些意義，但事實上我已在《神奇熔爐》（*The Magic Cauldron*）這篇文章中指出，未來軟體產業經濟的關鍵在於服務價值。

這個論點也藏有一個大問題，就是它假設開放原始碼專案無法長久維持，事實上，確實有長久持續發展的開放原始碼專案，一直保持一致的發展方向和有效的維護者社群，但其中卻沒有傳統專案管理上的激勵組織或制度化控制。GNU Emacs 編輯器的發展正是一個極端及具代表性的例子，它在十五年間，以統一的結構觀點，吸收了數百位貢獻者的努力，儘管參與的人這麼多，其實只有一個人（原作者）在十五年間持續活躍著。沒有一個封閉原始碼的編輯器曾創下如此長壽的紀錄。

在此，我們有理由質疑傳統管理下的專案發展，但這卻與其他教堂對上市集的論戰無關。如果 GNU 的 Emacs 編輯器可以在十五年間，保持一貫的結構，在過去八年間，雖然硬體平台的技術進步飛快，Linux 作業系統也同樣做到了，假如（事實如此）有這麼多結構良好的開放原始碼專案持續的時間超過五年，那麼我們不禁要問，傳統的專案管理除了帶來巨大的額外花費，是否還有其他益處？

傳統管理下的軟體專案當然不保證在期限內有效執行，不保證不超支預算，不保證實現所有的規格，很難得有一個「管理下」的專案能達到以上任一個要求，更不用說三個都達到了。管理下的專案似乎在它的生命週期中，很難去適應技術上和經濟環境上的變化，而開放原始碼社群卻已證明了開放原始碼專案有更高的績效。（我們可以做一個基本的驗證，例如比較網際網路三十年的歷史和短命的獨家網路技術，或者微軟視窗系統由 16 位元轉移到 32 位元付出的代價和同時期 Linux 輕易地移植到 Intel 系列之外超過一打的平台，其中包含 64 位元的 Alpha。）

許多人認為傳統商業軟體模式下，某些人能提供法律上的保證，如果該軟體的方向錯了，還可以補救回來，但這是個錯覺，大多數的軟體合約只宣告商業上的保證，而不是「履行」的保證，而且很少見到有未完成軟體成功地補救回來。即使這種情形稀鬆平常，因有人可以告而讓我們感到寬慰並不是有意義的事。我們不要訴訟，我們要可用的軟體。

到底傳統專案管理的額外花費帶來了什麼？

為了瞭解這個問題，我們首先必須弄清楚軟體專案管理者做些什麼事，一位我認識的女士在這個工作上表現優異，她說軟體專案管理有五個主要功能：

1. 定義目標，確保每一位成員方向一致。
2. 監督並且確定重要的細節沒有被忽視。
3. 誘導成員去做乏味但必要的苦工。

4. 調整成員組織以期發揮最大生產力。

5. 安排足夠的資源以支持整個專案。

這些很明顯地都是很有價值的目標，但在開放原始碼模式，及開放原始碼的社會意義下，這些目標看起來似乎很突兀，我們以倒過來的順序看：

我的朋友提到爭取資源基本上是防禦行為，一旦你有了人，有了機器和辦公室空間，你就必須保衛他們，以防被其他專案經理搶走，或者被高層搾乾。

但開放原始碼的發展人都是志願者，在所從事的專案中，依個人的興趣和能力，自行分工（就算他們是在有給職下精研開放原始碼軟體，一般而言，上述仍然屬實。）志願者的風格自動傾向於爭取資源的攻方，他們帶來自己所擁有的資源，因此對開放原始碼專案的管理者來說，很少需要打傳統的「保衛戰」，或者根本就不必要。

不論如何，在廉價的個人電腦和快速網路網路的世界中，我們發現了一個非常一致的事實：唯一有限的資源，就是具熟練技術的參與者。當一個開放原始碼專案成立後，原則上不需要爭取電腦，網路或者辦公室空間，只有當發展者們失去興趣時，它才會結束。

以此為例，非常重要的一點：開放原始碼的行家們如何自行分工以發揮最大的生產力 —— 這個環境會冷酷地挑選出稱職者。我的朋友同時熟悉於開放原始碼世界和大型封閉性專案，她相信開放原始碼已算是局部成功了，因為它的文化只接受最具才能的5%或說是寫程式的人。她大部份的時間花在組織運用其餘95%的人，並且親身見識到（許多人都知道）最強的程式師的生產力和僅能勝任的程式師相差100倍。

這個差異的倍數一定會引來一個笨問題：如果個別的專案，甚至整個領域，把能力差的人減到少於 50%，不就好了嗎？熟慮的管理者早已知道：如果傳統軟體專案管理的功能只在於把能力不及格的人訓練到及格，那麼軟體專案管理就不值得去做了。

開放原始碼社群的成功，更加突顯出這個問題，證明由網際網路上徵召自行分工的志願者，通常省錢又有效，勝過管理整棟在做其他事的人。

以上把我們帶向「動機」的問題，我朋友觀點的一個同義的，常聽到說法是：傳統軟體專案的管理，是對於成就動機低的程式師的一種補救措施。

這個答案通常伴隨一點聲明：我們對開放原始碼社群能完成工作的信賴，源自於「性感」的吸引力或者技術上的喜悅，缺乏這兩個要素的工作，如果不是沒人做，就是做得很糟，除非有經理揮動鞭子，驅使不自由的受雇者去攪局，我在 < *Homesteading the Noosphere* > 這篇文章中已經由心理和社會的因素去質疑這個論點。然而就現在的目的而言，我想：點出接受這種說法背後的意義會更為有趣。

如果傳統，封閉，重管理的軟體發展風格所防禦的是無趣的問題，那麼在該應用領域中，以馬其諾防線為手段，只有在無人發現這些問題的趣味，或者無人繞過這些問題的情況下，才算有效。但在目前，軟體無趣的部份有了開放原始碼的競爭者，使用者將知道最後有人處理的原因是：因為問題本身的魅力，吸引了解決問題的人 —— 不只在軟體界，或者其他創造性的工作中，問題的魅力是一個比只提供金錢還更有效的推動因素。

如果運用傳統的管理結構只是為了激勵被管理者的動機，那麼這也許是一個好的戰術，但卻是個壞的戰略，短期內會贏，但在長期會輸。

到目前為止，傳統軟體專案管理看起來有兩點（安排資源，調整組織）比不上開放原始碼，似乎在第三點（動機）上苟延殘喘，而被圍攻的可憐經理，在監督的工作上得不到任何援助；而開放原始碼社群最強的一點就在於分散式的審校，這點勝過確認有無細節被遺漏的任何傳統方法。

我們可不可以略去討論定義目標是不是傳統專案管理的額外付出呢？也許吧，但如果要這麼做，我們需要一個好理由讓我們相信：管理委員會和運作計畫在定義有價值和廣義的目標時，要比開放原始碼世界的專案領導人和資深參與者更成功。

表面上看起來很困難，但並不是因為作對的開放原始碼社群（Emacs 的長壽，Linus Torvalds 談論「稱霸世界」以激勵發展者的能力）造成的，因為定義軟體專案的目標，是傳統機制下的宣示性莊嚴。

軟體工程中最為人所知的一個理論是：有60%到75%的傳統軟體專案如果不是未完成就是被使用者所拒用。假如這個數字範圍和真實狀況很接近（我尚未遇過有經驗的經理人在爭論這個數字），那麼就有更多的專案沒有遵循目標發展，而目標如果不是（a）不切實際就是（b）錯了。

這點更甚於其他問題，在今天的軟體工程世界，「管理委員會」（management commitee）這個慣用語令聽的人脊背發涼 —— 即使（或者特別地）聽者本身就是管理者，只有程式師會如此的日子早已過去了；管理者桌上現在都放有「呆伯特」（Dilbert）漫畫[20]。

因此，我們對傳統專案管理者的回應很簡單：如果開放原始碼社群真的低估傳統管理的價值，那為什麼你們當中有這麼多人鄙視你們自己管理的作為呢？

既存的開放原始碼社群又再次突顯出這個問題 —— 因為樂趣在我們所做的工作中，我們有創意地遊戲著，已經以驚人的速度帶來技術上，分佔市場，和心靈分享上的成功，我們正在證明我們不只能寫出更好的軟體，而其中的樂趣也是一項資產。

在這一篇文章第一版發表後的兩年半，我所能提供的最基本想法，已不再限於開放原始碼的視界，但畢竟對許多在過去這些日子身陷訴訟而清醒的人而言，這些想法似乎都是合理的。

甚者，我想提出一個對於軟體更深廣的體認（也許對每一種造創或專業的工作都適用）在最適挑戰 —— 不會太容易而令人厭煩，不會太困難而難以達成的工作中，人們會獲得樂趣，一個快樂的程式師的條件是：工作量不會太輕，也未被定義不良的目標和壓力下的摩擦過度壓榨。享受樂趣確保了效率。

相對於在你的工作過程中，如果有恐懼和強烈的厭惡（即使如呆伯特漫畫中表達出的變換或諷刺方式），那就表示這個工作過程是失敗的。樂趣、幽默、和遊戲性是真的資產，這也是我前面為什麼寫出「快樂的遊牧民族」這個名詞，而 Linux 的吉祥物是隻人見人愛的企鵝也不僅是個笑話。

將來的事實會證明開放原始碼最大的成功之一就是它告訴了我們：如玩遊戲般地工作是從事創造性工作最經濟、最有效率的模式。

[20]指管理者也會害怕自己成為漫畫中那個老闆。 —— 譯註。

# 第十三章、結語：網景公司擁抱市集！

當你真的在影響歷史時，那種感覺真的很不一樣…

1998 年 1 月 22 日，大約是我第一次發表這篇論文之後七個月，網景公司公開聲明他們計畫要釋出通訊家族（Netscape Communicator）的原始程式碼，之前我對這件事情的發生一點頭緒也沒有。

網景的執行副總裁兼技術首席 Eric Hahn 寄給我一封電子郵件，內容簡要如下：「我代表網景公司的每一個人，向你致謝，因為你幫我們成為第一家做到開放原始碼的公司，你的思想和著作是我們做出這個決定的原因。」

接下來一個星期，我接受網景公司的邀請，搭飛機到矽谷參加他們長達一天的策略會議，參與會議的有該公司頂級執行者和技術人員，我們一起設計出網景公司開放原始碼的策略及版權，也做了些計畫，希望能對開放原始碼的社群有長遠而正面的影響，就如同我之前所說，這件事來得太快，以致於我們定出的策略和版權還不太明確，但在幾週內，細節應該可以交代清楚。

網景公司願意提供一個真實大型的試驗，在商業世界中測試市集模式，因此開放原始碼的世界正面臨著一個危機，假如網景的行動不成功，那麼開放原始碼的觀念在商業界也許就會被鄙棄，在十年內大概不會有公司再碰它。

另一方面，這也是一個空前的機會，華爾街和其他地方對這件事的初始反應是謹慎地正面評價，我們擁有這個機會來證明開放原始碼是有益的，假如網景藉此行動而重新獲得實際的市場，那麼它將引發軟體工業中一場遲來已久的革命。

明年（1999）將會是非常有意義而且有趣的一年。