Beej's Guide to

Network Programming

Using Internet Sockets

Brian "Beej Jorgensen" Hall

目錄

中文資訊

本書是 Linux socket 網路程式設計的敲門磚，對初學者而言是一份很好的開始，尤其第七章介紹了重要的 Linux socket 網路程式設計技巧與觀念。

- 譯者：Aaron Liao (廖明沂)
- 發佈日期：2014年05月
- 最後更新：2017年05月

聯絡譯者

對於中文翻譯的詞句有任何建議，如語意不夠貼切等或輸入的文字錯誤，都歡迎來信告知譯者，幫忙提升中文譯本的品質，謝謝。

- Aaron Liao
- Web: http://aaron.netdpi.net
- Email: aaron@netdpi.net

此書目前放置於下列幾個地點：

- 正體中文版（@google site）：http://beej-zhtw.netdpi.net
- 正體中文版（@Gitbook）：http://beej-zhtw-gitbook.netdpi.net

簡體中文

簡體中文讀者的一些電腦術語與正體中文不太一樣，所以譯者已經初步調整出一個初版，日後閒暇之餘再慢慢修正一些細節錯誤，若願意幫忙維護簡體中文版的讀者，也請來信告知：
简体中文版：http://beej-zhcn.netdpi.net

中文授權

原著資訊

書名：Beej's Guide to Network Programming - Using Internet Sockets

作者：Brian "Beej Jorgensen" Hall

版次：Version 3.0.21, June 8, 2016.

網站：http://beej.us/guide/bgnet/

授權：參考 1.9 節

下載 PDF：http://www.netdpi.net/files/beej/bgnet_A4_2012.pdf

進階資料

下列是譯者認爲值得推薦的 Linux/UNIX socket 網路程式設計書籍：

- Unix Network Programming Vol. 1：
  - W. Richard Stevens 大師的網路經典著作之一，對 socket 網路程式設計有更深入的介紹（如 raw socket、data link layer socket 或新型的 SCTP 傳輸層協定）。
- Effective TCP/IP Programming: 44 Tips to Improve Your Network Programs （中文版 - 深入研究 TCP/IP 網路程式設計）
  - Jon C. Snader 說明如何設計高效能、高可靠度的 socket 網路程式。
- The Linux Programming Interface（中文版 - The Linux Programming Interface 國際中文版）
  - 包羅萬象的 Linux/Unix 系統程式 API 開發手冊大全，要設計 multiple threads 網路程式可參考這本書，細節請參考作者網站。
- 基礎からわかるTCP/IPネットワーク実験プログラミング第2版（中文版 - TCP/IP網路程式實驗與設計）
  - 村山公保介紹 port scan、scan route、TCP hijack、TCP RST 攻擊、TCP SYN 攻擊、ARP spoofing、網路分析 network sniffer（raw socket 與 data link layer socket）等網路攻擊、網路分析的原理及技術，並有原始程式碼教導如何用 Linux 與 C 語言設計這類網路程式。

更多資訊請參考作者在 10.1 節推薦的書籍。 上述技術屬使用者空間（user space）的網路應用程式，對於作業系統如何在核心空間（kernel space）實作網路堆疊及資料結構設計，可參考下列書籍：

- W. Richard Stevens, TCP/IP Illustrated - The Implementation, Volume 2, Addison-Wesley Professional, 1995.
- Rami Rosen, Linux Kernel Networking: Implementation and Theory, first edition, Apress, 2013.
- Christian Benvenuti, Understanding Linux network internals, O'Reilly, 2006.（中文版 - Linux 網路原理）
- Klaus Wehrle et al., Linux Networking Architecture, Prentice Hall, 2004.
- Thomas Herbert, The Linux TCP/IP Stack: Networking for Embedded Systems, 2nd edition, Charles River Media, 2006.

導讀

嘿！Socket 程式設計讓你覺得很挫折嗎？

”這份教材是否只摘錄了 man 使用手冊而已呢？”

”網路程式很難嗎？”

你想要設計很酷的網路程式，可是沒有那麼多空閒時間可以瞭解一大堆資料結構，而且還需要知道”呼叫 connect() 之前一定要先呼叫 bind()”的順序等。

好，猜到了嗎！其實我已經完成了這件痛苦的事情，我正要與你們分享這些資訊，所以你來這裡就對了。

本文的目的是提供一份網路程式設計簡介，給想要了解網路程式的 C 程式設計師。

在這邊做個小結，我在本文件加上最新的資訊（其實也還好），並增加 IPv6 的介紹！大家盡情享受閱讀吧！

本書的讀者

本文件是一份導覽（tutorial），不是全方位的參考書。這份文件其實沒什麼了不起，不僅不夠全面，也沒有完整到足以做爲 socket 程式設計的大全。

不過期盼本文能讓大家不要害怕 man 手冊 … :-)

平台與編譯器

本文談到的程式碼是在 Linux PC 上，使用 GNU 的 gcc 編譯器所編譯的。然而，它應該在任何有 gcc 的平台上都能編譯。不過實際上，如果你在 Windows 上寫程式，這可能會有點不太一樣，請參考下列關於 Windows 程式設計的章節。

官方網頁與書本

本文件的官方位置是 http://beej.us/guide/bgnet/。

在這裡你也能找到程式碼的範例，以及各種語言的譯本。

如果需要購買價格比較好的複本〔有人稱爲”書”〕，

請到 http://beej.us/guide/url/bgbuy。

我很感謝您的購買，因爲這可以幫忙我繼續依靠寫文件吃飯！

我曾經加入一個很有名的網路書商，不過他們新的客户追蹤系統與已出版的文件無法相容。因此，我不能收到任何回饋金了。所以，如果你同情我的處境，請使用 paypal 贊助我〔beej@beej.us〕:-)

譯註：

1. Paypal 贊助這段原著本來放在第十章的參考資料開頭，但我覺得很不搭，所以移到這一節。
2. Beej's guide 是我以前學 Linux socket programming 的啓蒙，爲了表達感激，翻譯期間透過 paypal 小額贊助 Beej，他很快就回了信道謝，表示這個帳號還是有效的 :-)
3. 如果要贊助譯者，這是我的 paypal 帳户: aaron@netdpi.net

Solaris/SunOS 程式設計師該注意的事

當編譯 Solaris 或 SunOS 平台的程式時，你需要指定一些額外的命令列參數，以連結（link）正確的函式庫（library）。爲了達到這個目的，可以在編譯指令後面簡單加上 "-lnsl -lsocket -lresolv"，類似這樣：

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

如果還是有錯誤訊息，你可以再加上一個 "-lnext" 到命令列的尾端。我不太清楚這樣做了什麼事，不過有些人是會這樣用。

你可能會遇到的另一個問題是呼叫 setsockopt()。這個原型與在我 Linux 系統上的不一樣，所以可以這樣取代：

```
int yes=1;
```

輸入這行：

```
char yes='1';
```

因爲我沒有 Sun 系統，所以我無法測試上面的資訊，這只是有人用 email 跟我說的。

Windows 程式設計師該注意的事

本文以前只討論一點 Windows，純粹是我很不喜歡。不過我應該要客觀的說 Windows 其實提供很多基本安裝，所以顯然是個完備的作業系統。

人家說：小別勝新婚，這裡我相信這句話是對的〔或許是年紀的關係〕。不過我只能說，我已經十幾年沒有用 Microsoft 的作業系統來做自己的工作了，這樣我很開心！

其實我可以打出安全牌，只告訴你：”沒問題阿，你儘量去用 Windows 吧！”... 沒錯，其實我是咬著牙根說這些話的。

所以我還是在拉攏你來試試 Linux [1]、BSD [2]，或一些 Unix 風格的系統。

不過人們各有所好，而 Windows 的使用者也樂於知道這份文件的內容能用在 Windows，只是需要改變一點程式碼而已。

你可以安裝一個酷玩意兒－ Cygwin [3]，這是讓 Windows 平台使用的 Unix 工具集。我曾在秘密情報網聽過，這個能讓全部的程式不經過修改就能編譯。

不過有些人可能想要用純 Windows 的方法來做。只能說你很有勇氣，而你所要做的事就是：立刻去弄個 Unix！喔，不是，我開玩笑的。這些日子以來，大家一直認爲我對 Windows 是很友善的。

你所要做的事情就是〔除非你安裝了 Cygwin！〕：首先要忽略我這邊提過的很多系統 header（標頭檔），而你唯一需要 include （引用）的是：

```
#include <winsock.h>
```

等等，在你用 socket 函式庫做任何事情之前，必須要先呼叫 WSAStartup()。程式碼看起來像這樣：

```
#include <winsock.h>
{
  WSADATA wsaData; // if this doesn't work
  //WSAData wsaData; // then try this instead

  // MAKEWORD(1,1) for Winsock 1.1, MAKEWORD(2,0) for Winsock 2.0:

  if (WSAStartup(MAKEWORD(1,1), &wsaData) != 0) {
    fprintf(stderr, "WSAStartup failed.\n");
    exit(1);
  }
```

你也必須告訴編譯器要連結 Winsock 函式庫，在 Winsock 2.0 通常稱爲 wsock32.lib 或 winsock32.lib 或 ws2_32.lib。在 VC++ 底下，可以透過專案（Project）選單，在設定（Settings）底下 ...。按下 Link 標籤，並找到 "Object/library modules" 的標題。新增 "wsock32.lib"（或者你想要用的函式庫）到清單中。

最後，當你處理好 socket 函式庫時，你需要呼叫 WSACleanup()，細節請參考線上手冊。

只要你做好這些工作，本文後面的範例應該都能順利編譯，只有少部分例外。

還有一件事情，你不能用 close() 關閉 socket，你要用 closesocket() 來取代。而且 select() 只能用在 socket descriptors 上，不能用在 file descriptors（像 stdin 的值就是 0）。

還有一種你能用的 socket 類型，CSocket，細節請查詢你的編譯器使用手冊。

要取得更多關於 Winsock 的訊息可以先閱讀 Winsock FAQ [4]。

[1] http://www.linux.com/

[2] http://www.bsd.org/

[3] http://www.cygwin.com/

[4] http://tangentsoft.net/wskfaq/

最後，我聽說 Windows 沒有 fork() system call，我在一些範例中會用到。你可能需要連結到 POSIX 函式庫或要讓程式能動的一些函式庫，或許你也可以用 CreateProcess() 來取代。fork() 不需要參數，但是 CreateProcess() 卻需要大約 480 億個參數。如果你不想用，CreateThread() 會稍微比較容易理解 ... 不過多執行緒（multithreading）的討論則不在本文件的範疇中。我只能盡量提及，你要體諒！

來信原則

通常我很樂意幫助解決來信的問題，所以請儘管寫信來，不過，我不一定會回信，我很忙，而且還有三次沒有回答你們的問題。在這種情況下，我通常只會把訊息刪掉，這並沒有針對任何人；我只是沒空可以詳細答覆問題。

同樣的原則，越複雜的問題我就越不想回答。如果你很想要收到答覆，你可以在寄信之前先簡化你的問題，並確定你引述了相關的資訊〔比如平台、編譯器、得到的錯誤訊息，以及任何你想的到可以協助我找出問題的資訊〕。對於更多的要點，請閱讀 ESR 的文件，提問的智慧（How To Ask Questions The Smart Way）[5]。

[5] http://www.catb.org/~esr/faqs/smart-questions.html

若你沒有收到答覆，請自行先好好 hack（徹底研究）一番，試著找出答案，如果真的還是無法解決，那麼再寫信給我，並提供你找到的資訊，期盼會有足夠資訊可以讓我幫忙解決。

現在我一直拉著你說要怎樣才能寫信給我，以及哪些情況千萬別寫信給我，我只想讓你知道，我衷心地感謝這幾年所收到的讚美。這文件真的是個輕薄短小的資料，而且我很高興聽到大家說它非常實用！:-) 感謝您！

鏡射站台（**Mirroring**）

歡迎鏡射本站，無論公開或私人的。若你公開鏡射本站，並想要我從官網連結，請送個訊息到 beej@beej.us。

譯者該注意的

如果你想要將本文件翻譯爲其它語言，請寄信到 beej@beej.us，我會從官方主頁連結你的譯本，請隨意加上你的名字與聯絡資訊到譯本中。

請注意下列〔版權與散佈〕一節所列的授權限制，如果你想要我放譯本，跟我說就好了；若你想要自己架站，我會連結到你所提供的網址，任何方式都行。

版權與散佈

何謂 **Socket**

你一直聽到人家在講 "sockets"，你可能也想知道這些是什麼東西。

好的，其實它們就是：「利用標準 UNIX file descriptors（檔案描述符）與其它程式溝通的一種方式」。

「什麼？」

OK，你可能有聽過有些駭客（hacker）說過：「我的天呀！在 UNIX 系統中的任何東西都可以視爲檔案！」

他說的是眞的，不是假的。當 UNIX 程式要做任何類型的 I/O 時，它們會讀寫 file descriptor。

File descriptor 單純是與已開啓檔案有關的整數。只是「關鍵在於」該檔案可以是一個網路連線、FIFO、pipe（管線）、terminal（終端機）、眞實的磁碟檔案、或只是相關的東西。

在 UNIX 所見都是檔案！

所以當你想要透過 Internet（網際網路）跟其它的程式溝通時，你需要透過一個 file descriptor 來達成，這點你一定要相信。

「那麼，Smarty-Pants 先生，我在哪裡可以取得這個用在網路通信的 file descriptor 呢？」

這可能是你現在心裡的問題，我會跟你說的：你可以呼叫 socket() system routine（系統常式）。它會傳回 socket descriptor，你可以用精心設計的 send() 與 recv() socket calls（man send、man recv）來透過 socket descriptor 進行通信。

「不過，嘿嘿！」

現在你可能在想：「既然只是個 file descriptor，爲什麼我不能用一般的 read() 與 write() call 透過 socket 進行通信，而要用這什麼鬼東西阿？」

簡而言之：「可以！」

完整說來就是：「可以，不過 send() 與 recv() 讓你能對資料傳輸有更多的控制權」。

「接下來呢？」

這麼說吧：有很多種 sockets，如 DARPA Internet Sockets（網際網路位址）、本機端上的路徑名稱（path names on a local node，UNIX Sockets）、CCITT X.25 位址（你可以放心忽略 X.25 Sockets），可能還有其它的，要看你用的是哪種 UNIX 系統。在這裡我們只討論第一種：Internet Sockets。

兩種 Internet Sockets

這是什麼？有兩種 Internet socket 嗎？

是的，喔不，我騙你的啦。其實有很多種的 Internet socket，只是我不想嚇到你，所以這裡我只打算討論兩種。不過我還會告訴你 "Raw Socket"，這是很強大的東西，所以你應該要好好研究一下它們。

譯註： 一般的 socket 只能讀取傳輸層以上（不含）的資訊，raw socket 一般用在設計 network sniffer，可以讓應用程式取得網路封包底層的資訊（如 TCP 層、IP 層，甚至 link layer socket 可以讀取到 link layer 層），並用以分析封包資訊。這份文件不會談到這類的程式設計，有興趣的讀者可自行參考：Unix Network Programming Vol. 1、libpcap 或 TCP/IP 網路程式實驗與設計「內容包含介紹網路攻擊 port scan、scan route、TCP hijack、TCP RST 攻擊、TCP SYN 攻擊、ARP spoofing、網路分析 network sniffer（raw socket 與 data link layer socket）等原理及技術，並有原始程式碼教導如何用 Linux 與 C 語言設計這類網路攻擊及網路分析程式」。

好吧，不聊了。到底有哪兩種 Internet socket 呢？

其中一個是「Stream Socket（串流式 Socket）」，而另一個是「Datagram Socket（訊息式 Socket）」，之後我們分別以「SOCK_STREAM」與「SOCK_DGRAM」來表示。Datagram sockets 有時稱爲「免連線的 sockets（connectionless socket）」（雖然它們也可以用 connect()，如果你想這麼做的話，請見後面章節的 connect()）。

Stream socket 是可靠的、雙向連接的通信串流。若你以 "1、2" 的順序將兩個項目輸出到 socket，它們在另一端則會以 "1、2" 的順序抵達。而且不會出錯。

哪裡會用到 stream socket 呢？

好的，你應該聽過 telnet 程式吧，不是嗎？它就是用 stream socket。你所輸入的每個字都需要按照你所輸入的順序抵達，不是嗎？

網站瀏覽器所使用的 HTTP 通訊協定也是用 stream socket 取得網頁。的確，若你以 port 80 telnet 到一個網站，並輸入 "GET / HTTP/1.0"，然後按兩下 Enter，它就會輸出 HTML 給你！

Stream socket 是如何達成如此高品質的資料傳送呢？

它們用所謂的 "The Transmission Control Protocol"（傳輸控制協定），就是常見的 "TCP"（TCP 的全部細節請參考RFC 793[6]）。TCP 確保你的資料可以依序抵達而且不會出錯。你以前可能聽過 "TCP" 是 "TCP/IP" 裡比較好的部分，這邊的 "IP" 是指 "Internet Protocol"（網際網路協定，請見 RFC 791[7]）。IP 主要處理 Internet routing（網際網路的路由遞送），通常不保障資料的完整性。

酷喔。那 Datagram socket 呢？爲什麼它們號稱免連線呢？這邊有什麼好主意？爲什麼它們是不可靠的？

好，這裡說明一下現況：如果你送出一個 datagram（訊息封包），它可能會順利到達、可能不會按照順序到達，而如果它到達了，封包中的資料就是正確的。

譯註： TCP 會在傳輸層對將上層送來的過大訊息分割成多個分段（TCP segments），而 UDP 本身不會，UDP 是訊息導向的（message oriented），若 UDP 訊息過大時（整體封包長度超過 MTU），則會由 host 或 router 在 IP 層對封包進行分割，將一個 IP packet 分割成多個 IP fragments。IP fragmention 的缺點是，接收端的系統需要做 IP 封包的重組，將多個 fragments 重組合併爲原本的 IP 封包，同時也會增加封包遺失的機率。如將一個 IP packet 分裂成多個 IP fragments，只要其中一個 IP fragment 遺失了，接收端就會無法順利重組 IP 封包，因而造成封包的遺失，若是高可靠度的應用，則上層協定需重送整個 packet 的資料。

[6] http://tools.ietf.org/html/rfc793 [7] http://tools.ietf.org/html/rfc791

Datagram sockets 也使用 IP 進行 routing（路由遞送），不過它們不用 TCP；而是用 "UDP，User Datagram Protocol"（使用者資料包協定，請見 RFC 768 [8]）。

爲什麼它們是免連線的？

好，基本上，這跟你在使用 stream socket 時不同，你不用維護一個開啓的連線，你只需打造封包、給它一個 IP header 與目的資訊、送出，不需要連線。通常用 datagram socket 的時機是在沒有可用的 TCP stack 時；或者當一些封包遺失不會造成什麼重大事故時。這類應用程式的例子有：tftp（trivial file transfer protocol，簡易檔案傳輸協定，是 FTP 的小兄弟），多人遊戲、串流音樂、影像會議等。

”等一下！tftp 和 dhcpd 是用來在一台主機與另一台之間傳輸二進制的應用資料！你如果想要應用程式能在資料抵達時正常運作，那資料就不能遺失阿！這是什麼黑魔法？”

好，我的人族好友，tftp 與類似的程式會在 UDP 的上層使用它們自己的協定。比如：tftp 協定會報告每個收送的封包，接收端必須送回一個封包表示：”我收到了！”〔一個”ACK”回報封包〕。若原本封包的傳送端在五秒內沒有收到回應，這表示它該重送這個封包，直到收到 ACK 爲止。在實作可靠的 SOCK_DGRAM 應用程式時，這個回報的過程很重要。

對於無需可靠度的（unreliable）應用程式，如遊戲、音效、或影像，你只需忽略遺失的封包，或也許能試著用技巧彌補回來。（雷神之鎚的玩家都知道的一個技術名詞的影響：accursed lag。在這個例子中，"accursed"（受詛咒）這個字代表各種低級的意思）。

爲什麼你要用一個不可靠的底層協定？

有兩個理由：第一個理由是速度，第二個理由還是速度。直接忘了遺失的這個封包是比較快的方式，相較之下，持續追蹤全部的封包是否安全抵達，並確保依序抵達是比較慢的。如果你想要傳送聊天訊息，TCP 很讚；不過如果你想要替全世界的玩家，每秒送出 40 個位置更新的資訊，且若遺失一到兩個封包並不會有太大的影響時，此時 UDP 是一個好的選擇。

[8] http://tools.ietf.org/html/rfc768

譯註： stream（串流式）socket 是指應用程式要傳輸的資料就如水流（串流）在水管中傳輸一般，經由這個 stream socket 流向目的，串流式 socket 是資料會由傳輸層負責處理遺失、依序送達等工作，以在傳輸層確保應用程式所送出的資料能夠可靠且依序抵達，而應用程式若對資料有可靠與依序的需求時，使用 stream socket 就不用自行處理這類的工作。

datagram（訊息式）socket 是基於訊息導向的方式傳送資料，應用程式送出的每筆資料會如平信的概念送出，由於遞送封包的路徑可能會隨著網路條件而改變，每筆資料抵達的順序不一定會按照送出的順序抵達，並且如平信般，信件可能在遞送過程遺失，而寄件人並無法知道是否遞送成功。

初步簡單知道應用這兩種 sockets 的時機：當需要資料能完整送達目地時，就使用 stream socket，若是部分資料遺失也無妨時，就可以使用 datagram socket。

底層漫談與網路理論

因爲我只著重於協定的分層，該是談談網路是如何眞正的運作的時候了，並呈現一些如何打造 SOCK_DGRAM 封包的例子。就實務面，你或許可以跳過這一節，不過，這一節有很好的觀念背景，所以需要的人可以讀一讀。

Data Encapsulation.

嘿！孩子們，該是學習資料封裝（Data Encapsulation）的時候了。這很重要，它就是如此重要，即使你是在加州這裡上的網路課程，也只能學到皮毛。

基本上我們會講到這些內容：

封包的誕生、將封包打包〔封裝〕到第一個協定〔所謂的 TFTP 協定〕的 header 中〔幾乎是最底層了〕，接著將全部的東西〔包含 TFTP header〕封裝到下一個協定中〔所謂的 UDP〕，接著下一個協定〔IP〕，最後銜接到硬體〔實體〕層上面的通訊協定〔所謂的 Ethernet，乙太網路〕。

當另一台電腦收到封包時，硬體會解開 Ethernet header，而 kernel 會解開 IP 與 UDP header，再來由 TFTP 程式解開 TFTP header，最後程式可以取得資料。

現在我最後要談個聲名狼藉的分層網路模型（Layered Network Model），亦稱 "ISO/OSI"。這個網路模型介紹了一個網路功能系統，有許多其它模型的優點。例如，你可以寫剛好一樣的 socket 程式，而不用管資料在實體上是怎麼傳送的〔Serial、thin Ethernet、AUI 之類〕。因爲在底層的程式會幫你處理這件事。眞正的網路硬體與拓樸對 socket 程式設計師而言是透明的。

不囉嗦，我將介紹這個成熟模型的分層。爲了網路課程的測驗，要記住這些。

Application（應用層） Presentation（表現層） Session（會談層） Transport（傳輸層） Network（網路層） Data Link（資料鏈結層） Physical（實體層）

實體層就是硬體（serial、Ethernet 等）。而應用層你可以盡可能的想像，這是個使用者與網路互動的地方。

現在這個模型已經很普及，所以你如果願意的話，或許可以將它當作一本汽車修理指南來用。與 Unix 比較相容的分層模型有：

應用層（Application layer：telnet、ftp 等） 主機到主機的傳輸層（Transport layer：TCP、UDP） 網際網路層（Internet layer：IP 與路由遞送） 網路存取層（Network Access Layer：Ethernet、wi-fi、諸如此類）

此時，你或許能知道這幾層是如何對應到原始資料的封裝。

看看在打造一個簡單的封包需要多少工作呢？

天阿！你得自己用 "cat" 將資訊填入封包的 header 裡！

開玩笑的啦。

你對 stream socket 需要做的只有用 send() 將資料送出。而在 datagram socket 需要你做的是，用你所選擇的方式封裝該封包，並且用 sendto() 送出。Kernel 會自動幫你建立傳輸層與網路層，而硬體處理網路存取層。啊！眞現代化的技術。

所以該結束我們短暫的網路理論之旅了。

喔！對了，我忘記告訴你我想要談談 routing（路由遞送）了。恩，沒事！沒關係，我不打算全部講完。

Router（路由器）會解開封包的 IP header，參考自己的 routing table（路由表）…。如果你眞的很想知道，你可以讀 IP RFC [9]。如果你永遠都不想碰它，其實你也可以過得很好。

[9] http://tools.ietf.org/html/rfc791

IP address、結構與資料轉換

這裡是好玩的地方，我們要開始談程式碼了。

不過，我們一開始要討論的程式碼會比較少！

耶！因爲我想要先講點 IP address（位址）與 port（連接埠），這樣才會有點感覺；接著我們會討論 socket API 如何儲存與控制 IP address 和其它資料。

IPv4 與 IPv6

在 Ben Kenobi 還是叫 Obi Wan Kenobi 的那段過去的美好時光，有個很棒的 network routing system（網路路由系統），稱爲 Internet Protocol Version 4（網際網路協定第四版），又稱爲 IPv4。它的位址是由四個 bytes 組成（亦稱爲四個”octets”），而格式是由句點與數字組成，像是這樣：192.0.2.111。

你或許曾經看過。

實際上，在撰寫本文時，幾乎整個 Internet（網際網路）的每個網站都還是使用 IPv4。

每個人跟 Obi Wan 都很開心，一切都是如此美好，直到某個名爲 Vint Cerf 的人提出質疑，警告所有人 IPv4 address 即將耗盡。

Vint Cerf [10] 除了提出即將到來的 IPv4 危機警告，他本身還是有名的 Internet 之父，所以我眞的沒資格能評論他的判斷。

你說的是耗盡 address 嗎？會發生什麼事呢？其實我的意思是，32-bit 的 IPv4 address 有幾十億個 IP address，我們眞的有幾十億台的電腦在用嗎？

是的。在一開始大家也是認爲這樣就夠用了，因爲當時只有一些電腦，而且每個人認爲幾十億是不可能用完的大數目，還很慷慨的分給某些大型組織幾百萬個 IP address 供他們自己使用（例如：Xerox、MIT、Ford、HP、IBM、GE、AT&T 及某個名爲 Apple 的小公司，族繁不及備載）。

不過現實狀況是，如果不是有些變通的方法，我們早就用光 IPv4 位址了。

我們現在生活於每個人、每台電腦、每部計算機、每隻電話、每部停車計時收費器、以及每條小狗（爲什麼不行？）都有一個 IP address 的年代，因此，IPv6 誕生了。

因爲 Vint Cerf 可能是不朽的（即使他的軀殼終究該回歸自然，我也希望永遠不會發生，不過他的精神或許已經以某種超智慧的 ELIZA [11] 程式存在於 Internet2 的核心），應該沒有人想要因爲下一代的網際網路協定沒有足夠的位址，然後又聽到他說：「我要告訴你們一件事 ...」。

那你有什麼建議嗎？

我們需要更多的位址，我們需要不止兩倍以上的位址、不止幾十億倍、千兆倍以上，而是 79 乘以 百萬 乘以 十億 乘以 兆倍以上的可用位址！你們大家將會見識到的。

你說：「Beej，眞的嗎？我還是有許多可以質疑這個大數字的理由。」

好的，32 bits 與 128 bits 的差異聽起來似乎不是很多；它只多了 96 個 bits 而已，不是嗎？不過請記得，我們所談的是等比級數；32 bits 表示個 40 億的數字〔2 的 32 次方〕，而128 bits 表示的大約是 340 個兆兆兆的數字〔2 的 128 次方〕，這相當於宇宙中的每顆星星都能擁有一百萬個 IPv4 Internets。

大家順便忘了 IPv4 的句號與數字的長相吧；現在我們有十六進制的表示法，每兩個 bytes 間以冒號分隔，類似這樣：

2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551。

這還不是全部呢！大部分的時候，你的 IP address 裡面會有很多個零，而你可以將它們壓縮到兩個冒號間，你也可以去掉每個 byte pair（位元組對）裡開頭的零。例如，這些成對位址中的兩個位址是相等的：

2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51
2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::
0000:0000:0000:0000:0000:0000:0000:0001
::1

[10] http://en.wikipedia.org/wiki/Vinton_Cerf

[11] http://en.wikipedia.org/wiki/ELIZA

位址 ::1 是個 loopback（遶回）位址，它永遠只代表「我現在執行的這台電腦」，在 IPv4 中，loopback 位址是 127.0.0.1。

最後，你可能會遇到 IPv6 與 IPv4 相容的模式。例如，如果你願意的話，你可以將 IPv4 address 192.0.2.33 以 IPv6 位址表示，可以使用如下的符號：「::ffff:192.0.2.33」。

因爲所謂的自信，所以 IPv6 的發明人很有把握的保留了兆來兆去的位址，不過說實在的，我們有這麼多位址，誰能算清楚呢？

還剩下很多位址可以分配給星系中每個行星的每個男人、女人、小孩、小狗跟停車計時收費器。相信我，星系中的每個行星都有行車計時收費器。你明白這是眞的。

爲了結構化的理由，有時我們這樣宣告是很方便的：”IP address 的前段是 IP address 的 network（網段），而後面的部分是 host（主機）。”

例如：在 IPv4，你可能有 192.0.2.12，而我們可以說前面三個 bytes 是 network，而最後一個 byte 是 host。或者換個方式，我們能說 host 12 位在 network 192.0.2.0。〔請參考我們如何將 host byte 清爲零〕。

接下來要講的是過時的資訊了！

眞的嗎？

很久很久以前，有 subnets（子網路）的 "class"（分級），在這裡，位址的第一個、前二個或前三個 bytes 都是屬於 network 的一部分。如果你很幸運可以擁有一個 byte 的 network，而另外三個 bytes 是 host 位址，那在你的網路上，你有價值 24 bits 的 host number〔大約兩千四百萬個位址左右〕。這是一個 "Class A"（A 級）的網路；相對則是一個 "Class C"（C 級）的網路，network 有三個 bytes、而 host 只有一個 byte〔256 個 hosts，而且還要再扣掉兩個保留的位址〕。

所以，如同你所看到的，只有一些 Class A 網路，一大堆的 Class C 網路，以及一些中等的 Class B 網路。

IP address 的網段部分由 netmask（網路遮罩）決定，你可以將 IP address 與 netmask 進行 AND 位元運算，就能得到 network 的值。Netmask 一般看起來像是 255.255.255.0〔如：若你的 IP 是 192.0.2.12，那麼使用這個 netmask 時，你的 network 就會是 192.0.2.12 AND 255.255.255.0 所得到的值：192.0.2.0〕。

無庸置疑的，這樣的分級對於 Internet 的最終需求而言並不夠細膩；我們已經以相當快的速度在消耗 Class C 網路，這是我們都知道一定會耗盡的 Class，所以不用費心去想了。補救的方式是，要能接受任意個 bits 的 netmask，而不單純是 8、16 或 24 個而已。所以你可以有個 255.255.255.252 的 netmask，這個 netmask 能切出一個 30 個 bits 的 network 及 2 個 bits 的 host，這個 network 最多有四台 hosts〔注意，netmask 的格式永遠都是：前面是一連串的 1，然後，後面是一連串的 0〕。

不過一大串的數字會有點不好用，比如像 255.192.0.0 這樣的 netmask。首要是人們無法直觀地知道有多少個 bits 的 1；其次是這樣眞的很不嚴謹。因此，後來的新方法就好多了。你只需要將一個斜線放在 IP address 後面，接著後面跟著一個十進制的數字用以表示 network bits 的數目，類似這樣：192.0.2.12/30。

或者在 IPv6 中，類似這樣：2001:db8::/32 或 2001:db8:5413:4028::9db9/64。

如果你還記得我之前跟你說過的分層網路模型（Layered Network Model），它將網路層（IP）與主機到主機間的傳輸層〔TCP 與 UDP〕分開。

我們要加快腳步了。

除了 IP address 之外〔IP 層〕，有另一個 TCP〔stream socket〕使用的位址，剛好 UDP〔datagram socket〕也用這個，就是 port number，這是一個 16-bit 的數字，就像是連線的本地端位址一樣。

將 IP address 想成飯店的地址，而 port number 就是飯店的房間號碼。這是貼切的比喻；或許以後我會用汽車工業來比喻。

你說想要有一台電腦能處理收到的電子郵件與網頁服務－你要如何在一台只有一個 IP address 的電腦上分辨這些封包呢？

好，Internet 上不同的服務都有已知的（well-known）port numbers。你可以在 Big IANA Port 清單 [12] 中找到，如果你用的是 Unix 系統，你可以參考檔案 /etc/services。HTTP（網站）是 port 80、telnet 是 port 23、SMTP 是 port 25，而 DOOM 遊戲 [13] 使用 port 666 等，諸如此類。Port 1024 以下通常是有特地用途的，而且要有作業系統管理員權限才能使用。

摁，這就是 port number 的介紹。

[12] http://www.iana.org/assignments/port-numbers

[13] http://en.wikipedia.org/wiki/Doom_\(video_game\)

Byte Order（位元組順序）

一直都以爲有兩種 byte orderings，現在才知道，其實只有一種。

開玩笑的，但是其中一個的確比較受推崇 :-)

這眞的不容易解釋，所以我只能胡扯：你的電腦可能背著你用相反的順序來儲存 bytes。我知道！以前沒人跟你說過。

Byte Order 其實就是，若你想要用兩個 bytes 的十六進制數字來表示資料，比如說 b34f，你可以將它以 b34f 的順序儲存，這件事在 Internet 世界的每個人一般都可以同意。這很合理，而且 Wilford Brimley [14] 也會跟你說，這麼做是對的。由於這個數字是先儲存比較大的那一邊（big end），所以稱爲 Big-Endian。

不幸的是，世界上的電腦那麼多，而 Intel 或 Intel 相容處理器就將 bytes 反過來儲存，所以 b34f 存在記憶體中的順序就是 4fb3，這樣的儲存方式稱爲 Little-Endian。

不過，請等等。我還沒解釋名詞哩！

照理說，Big-Endian 又稱爲 Network Byte Order，因爲這個順序與我們網路類型順序一樣。

你的電腦會以 Host Byte Order 儲存數字，如果是 Intel 80x86，Host Byte Order 是 Little-Endian；若是 Motorola 68k，則 Host Byte Order 是 Big-Endian；若是 PowerPC，Host Byte Order 就是 ... 恩，這要看你的 PowerPC 而定。

大多數當你在打造封包或填寫資料結構時，你需要確認你的 port number 跟 IP address 都是 Network Byte Order。只是如果你不知道本機的 Host Byte Order，那該怎麼做呢？

好消息是，我們可以直接假設，全部電腦的 Host Byte Order 都不是 Network Byte Order，然後每次都用函式將值轉換爲 Network Byte Order。如果有必要，函式會發動魔法般的轉換，而這個方式會讓你的程式碼能更方便地移植到不同 endian 的機器。

你可以轉換兩種型別的數值：short〔兩個 bytes〕與 long〔四個 bytes〕。這些函式也可以用在 unsigned 變數。比如說，你想要將 short 從 Host Byte Order 轉換爲 Network Byte Order， 'h' 代表 "host"， 'n' 代表 "network"，而 's' 代表 "short"，所以是：h-to-n-s，或者 htons()〔讀做："Host to Network Short"〕。

這眞是太簡單了...

你可以用任何你想要的方式來組合 'n'、 'h'、's' 與 'l'，不過別用太蠢的組合，比如：沒有這樣的函式 stolh()〔"Short to Long Host"〕，沒有這種東西，不過有下面這些：

```
htons() host to network short
htonl() host to network long
ntohs() network to host short
ntohl() network to host long
```

基本上，你需要在送出封包以前將數值轉換爲 Network Byte Order，並在收到封包之後將數值轉回 Host Byte Order。

抱歉，我不知道 64-bit 的差異，如果你想要處理浮點數的話，可以參考 7.4 節。

[14] http://en.wikipedia.org/wiki/Wilford_Brimley

如果我沒特別強調的話，本文中的數值預設是視爲 Host Byte Order。

資料結構

很好，終於講到這裡了，該是談談程式設計的時間了。在本節，我會介紹 socket 介面的各種資料型別，因爲它們有些會不太好理解。

首先是最簡單的：socket descriptor，型別如下：

```
int
```

就是一般的 int。

從這裡開始會有點不好理解，所以不用問太多，直接讀過就好。

我的第一個 StructTM －struct addrinfo，這個資料結構是最近的發明，用來準備之後要用的 socket 位址資料結構，也用在主機名稱（host name）及服務名稱（service name）的查詢。當我們之後開始實際應用時，才會開始覺得比較合理，現在只需要知道你在建立連線呼叫時會用到這個資料結構。

```
struct addrinfo {
    int ai_flags; // AI_PASSIVE, AI_CANONNAME 等。
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol; // 用 0 當作 "any"
    size_t ai_addrlen; // ai_addr 的大小，單位是 byte
    struct sockaddr *ai_addr; // struct sockaddr_in 或 _in6
    char *ai_canonname; // 典型的 hostname
    struct addrinfo *ai_next; // 鏈結串列、下個節點
};
```

你可以載入這個資料結構，然後呼叫 getaddrinfo()。它會傳回一個指標，這個指標指向一個新的鏈結串列，這個串列有一些資料結構，而資料結構的內容記載了你所需的東西。

你可以在 ai_family 欄位中設定強制使用 IPv4 或 IPv6，或者將它設定爲 AF_UNSPEC，AF_UNSPEC 很酷，因爲這樣你的程式就可以不用管 IP 的版本。

要注意的是，這是個鏈結串列：ai_next 是指向下一個元素（element），可能會有多個結果讓你選擇。我會直接用它提供的第一個結果，不過你可能會有不同的個人考量；先生！我不是萬事通。

你會在 struct addrinfo 中看到 ai_addr 欄位是一個指向 struct sockaddr 的指標。這是我們開始要了解 IP 位址結構中有哪些細節的地方。有時候，你需要的是呼叫 getaddrinfo() 幫你填好 struct addrinfo。然而，你必須查看這些資料結構，並將值取出，所以我在這邊會進行說明。

〔還有，在發明 struct addrinfo 以前的程式碼都要手動填寫這些資料的每個欄位，所以你會看到很多 IPv4 的程式碼眞的用很原始的方式去做這件事。你知道的，本文件在舊版也是這樣做〕。

有些 structs 是 IPv4，而有些是 IPv6，有些兩者都是。我會特別註明清楚它們屬於哪一種。

總之，struct sockaddr 記錄了很多 sockets 類型的 socket 的位址資訊。

```
struct sockaddr {
    unsigned short sa_family; // address family, AF_xxx
    char sa_data[14]; // 14 bytes of protocol address
};
```

sa_family 可能是任何東西，不過在這份文件中我們會用到的是 AF_INET〔IPv4〕或 AF_INET6〔IPv6〕。sa_data 包含一個 socket 的目地位址與 port number。這樣很不方便，因爲你不會想要手動的將位址封裝到 sa_data 裡。

爲了處理 struct sockaddr，程式設計師建立了對等平行的資料結構：struct sockaddr_in〔"in" 是代表 "internet"〕，可用在 IPv4。

而這邊有個重點：指向 struct sockaddr_in 的指標可以轉型（cast）爲指向 struct sockaddr 的指標，反之亦然。所以即使 connect() 需要一個 struct sockaddr *，你也可以用 struct sockaddr_in，並在最後的時候對它做型別轉換！

```
// （IPv4 專用-- IPv6 請見 struct sockaddr_in6）
struct sockaddr_in {
    short int sin_family; // Address family, AF_INET
    unsigned short int sin_port; // Port number
    struct in_addr sin_addr; // Internet address
    unsigned char sin_zero[8]; // 與 struct sockaddr 相同的大小
};
```

這個資料結構讓它很容易可以參考（reference）socket 位址的成員。要注意的是 sin_zero〔這是用來將資料結構補足符合 struct sockaddr 的長度〕，應該要使用 memset() 函式將 sin_zero 整個清爲零。還有，sin_family 是對應到 struct sockaddr 中的 sa_family，並應該設定爲 "AF_INET"。最後，sin_port 必須是 Network Byte Order〔利用 htons()〕。

讓我們再更深入點！你可以在 struct in_addr 裡看到 sin_addr 欄位。

那是什麼？

好，別太激動，不過它是其中一個最恐怖的 union：

```
// (僅限 IPv4 — Ipv6 請參考 struct in6_addr)
// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

哇！好耶，它以前是 union，不過這個包袱現在似乎已經不見了。因此，若你已將 ina 宣告爲 struct sockaddr_in 的型別時，那麼 ina.sin_addr.s_addr 會參考到 4-byte 的 IP address（以 Network Byte Order）。要注意的是，如果你的系統仍然在 struct in_addr 使用超恐怖的 union，你依然可以像我上面說的，精確地參考到 4-byte 的 IP address〔這是因爲有 #define〕。

那麼 IPv6 會怎樣呢？

IPv6 也有提供類似的 struct，比如：

```
// (IPv6 專用-- IPv4 請見 struct sockaddr_in 與 struct in_addr)
struct sockaddr_in6 {
    u_int16_t sin6_family; // address family, AF_INET6
    u_int16_t sin6_port; // port number, Network Byte Order
    u_int32_t sin6_flowinfo; // IPv6 flow 資訊
    struct in6_addr sin6_addr; // IPv6 address
    u_int32_t sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char s6_addr[16]; // IPv6 address
};
```

要注意到 IPv6 協定有一個 IPv6 address 與一個 port number，就像 IPv4 協定有一個 IPv4 address 與 port number 一樣。

我現在還不會介紹 IPv6 的流量資訊，或是 Scope ID 欄位 … 畢竟這只是一份入門文件嘛 :-)

最後要強調的一點，這個簡單的 struct sockaddr_storage 設計用來足以儲存 IPv4 與 IPv6 structures 的 structure。〔你看看，對於某些 calls，你有時無法事先知道它是否會使用 IPv4 或 IPv6 address 來填好你的 struct sockaddr。所以你用這個平行的 structure 來傳遞，它除了比較大以外，也很類似 struct sockaddr ，因而可以將它轉型爲你所需的型別〕。

```
struct sockaddr_storage {
    sa_family_t ss_family; // address family
    // 這裡都是填充物（padding），依實作而定，請忽略它：
    char __ss_pad1[_SS_PAD1SIZE];
    int64_t __ss_align;
    char __ss_pad2[_SS_PAD2SIZE];
};
```

重點是你可以在 ss_family 欄位看到位址家族（address family），檢查它是 AF_INET 或 AF_INET6（是 IPv4 或 IPv6）。之後如果你願意的話，你就可以將它轉型爲 sockaddr_in 或 struct sockaddr_in6。

IP 位址，續集

還好你運氣不錯，有一堆函式讓你能夠控制 IP address，而不需要親自用 long 與 << 運算符來處理它們。

咱們說，你有一個 struct sockaddr_in ina，而且你有一個 "10.12.110.57" 或 "2001:db8:63b3:1::3490" 這樣的一個 IP address 要儲存。你想要使用 inet_pton() 函式將 IP address 轉換爲數值與句號的符號，並依照你指定的 AF_INET 或 AF_INET6 來決定要儲存在 struct in_addr 或 struct in6_addr。

（"pton" 的意思是 "presentation to network"，你可以稱之爲 "printable to network"，如果這樣會比較好記的話）。

這樣的轉換可以用如下的方式：

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6
inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

（小記：原本的老方法是使用名爲 inet_addr() 或是 inet_aton() 的函式；這些都過時了，而且不適合在 IPv6 中使用）。

目前上述的程式碼片段還不是很可靠，因爲沒有錯誤檢查。inet_pton() 在錯誤時會傳回 -1，而若位址被搞爛了，則會傳回 0。所以在使用之前要檢查，並確認結果是大於 0 的。

好了，現在你可以將 IP address 字串轉換爲它們的二進位表示。

還有其它方法嗎？

如果你有一個 struct in_addr 且你想要以數字與句號印出來的話呢？

（呵呵，或者如果你想要以”十六進位與冒號”印出 struct in6_addr〕。在這個例子中，你會想要使用 inet_ntop()函式〔"ntop" 意謂 "network to presentation"－如果有比較好記的話，你可以稱它爲 "network to printable"），像是這樣：

```
// IPv4:
char ip4[INET_ADDRSTRLEN]; // 儲存 IPv4 字串的空間
struct sockaddr_in sa; // 假裝這會由某個東西載入
inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
printf("The IPv4 address is: %s\n", ip4);
// IPv6:
char ip6[INET6_ADDRSTRLEN]; // 儲存 IPv6 字串的空間
struct sockaddr_in6 sa6; // 假裝這會由某個東西載入
inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);
printf("The address is: %s\n", ip6);
```

當你呼叫它時，你會傳遞位址的型別（IPv4 或 IPv6），該位址是一個指向儲存結果的字串，與該字串的最大長度。「有兩個 macro（巨集）可以很方便地儲存你想儲存的最大 IPv4 或 IPv6 位址字串大小：INET_ADDRSTRLEN 與 INET6_ADDRSTRLEN」。

（另一個要再次注意的是以前的方法：以前做這類轉換的函式名爲 inet_ntoa()，它已經過期了，而也在 IPv6 中也不適用）。

最後，這些函式只能用在數值的 IP address 上，它們不需要 DNS nameserver 來查詢主機名稱，如 "www.example.com"。你可以使用 getaddrinfo() 來做這件事情，如同你稍後會看到的。

很多地方都有防火牆（firewall），它們保護網路，將網路隱藏於世界的某個地方。有時，防火牆會用所謂的網路位址轉換（NAT，Network Address Translation）的方法，將 "internal"（內部的）IP 位址轉換爲 "external"（外部的）〔世界上的每個人都知道的〕IP address。

你又開始緊張了嗎？”他又要扯到哪裡去了？”

好啦，放輕鬆，去買瓶汽水〔或酒精〕飲料，因爲身爲一個初學者，你還可以先別理 NAT，因爲它所做的事情對你而言是透明的。不過我想在你開始對所見的網路數量開始感到困惑以前，先談談防火牆後面的網路。

比如，我家有一個防火牆，我有兩個 DSL 電信公司分配給我的靜態 IPv4 位址，而我家的網路有七部電腦要用。這有可能嗎？兩台電腦不能共用同一個 IP address 阿，不然資料就不知道該送去哪一台電腦了！

答案揭曉：它們不會共用同一個 IP address，它們是在一個擁有兩千四百萬個 IP address 的 private network 裡面，這些 IP addresses 全部都是我的。

好，都是你的，有這麼多位址可以讓大家用來上網，而這裡要講的就是爲什麼：

如果我登入到一台遠端的電腦，它會說我從 192.0.2.33 登入，這是我的 ISP 提供給我的 public IP。不過若是我問我自己本地端的電腦，它的 IP address 是什麼時，他會說是 10.0.0.5。是誰轉換 IP 的呢？答對了，就是防火牆！它做了 NAT！

10.x.x.x 是其中一個少數保留的網路，只能用在完全無法連上 Internet 的網路〔disconnected network〕，或是在防火牆後的網路。你可以使用哪個 private network 編號的細節是記在 RFC 1918 [15] 中，不過一般而言，你較常見的是 10.x.x.x 及 192.168.x.x，這裡的 x 是指 0-255。較少見的是 172.y.x.x，這裡的 y 範圍在 16 與 31 之間。

在 NAT 防火牆後的網路可以不必用這些保留的網路，不過它們通常會用。

〔眞好玩！我的外部 IP 眞的不是 192.0.2.33，192.0.2.x 網段保留用來虛構本文要用的 ”眞實”IP address，就像本文也是虛構的一樣 Wowzers！〕

IPv6 也很合理的會有 private network。它們是以 fdxx: 開頭〔或者未來可能是 fcXX：〕，如同 RFC 4193 [16]。NAT 與 IPv6 通常不會混用，然而〔除非你在做 IPv6 轉 IPv4 的 gateway，這就不在本文的討論範圍內了〕，理論上，你會有很多位址可以使用，所以根本不再需要使用 NAT。不過，如果你想要在不會遶送到外面的網路〔封閉網路〕上配置位址給你自己，就用 NAT 吧。

[15] http://tools.ietf.org/html/rfc1918

[16] http://tools.ietf.org/html/rfc4193

從 IPv4 移植爲 IPv6

「可是我只想知道要怎麼改程式碼的哪些地方就可以支援 IPv6 了！快點告訴我！」

OK！OK！

我幾乎都是以過來人的身分來講這邊的每件事，這次不考驗各位的耐心，來個簡潔的短文。（當然有更多比這篇短的文章，不過我是跟自己的這份文件來比較）。

1. 首先，請試著用 getaddrinfo() 來取得 struct sockaddr 的資訊，取代手動填寫這個資料結構。這樣你就可以不用管 IP 的版本，而且能節省後續許多步驟。
2. 找出全部與 IP 版本相關的任何程式碼，試著用一個有用的函式將它們包起來（wrap up）。
3. 將 AF_INET 更改爲 AF_INET6。
4. 將 PF_INET 更改爲 PF_INET6。
5. 將 INADDR_ANY 更改爲 in6addr_any，這裡有點不太一樣：

```
struct sockaddr_in sa;
struct sockaddr_in6 sa6;

sa.sin_addr.s_addr = INADDR_ANY;   // 使用我的 IPv4 位址
sa6.sin6_addr = in6addr_any;  // 使用我的 IPv6 位址
```

還有，在宣告 struct in6_addr 時，IN6ADDR_ANY_INIT 的值可以做爲初始值，像這樣：

```
struct in6_addr ia6 = IN6ADDR_ANY_INIT;
```

1. 使用 struct sockaddr_in6 取代 struct sockaddr_in，確定要將 "6" 新增到適當的欄位〔參考上面的 structs〕，但沒有 sin6_zero 欄位。
2. 使用 struct in6_addr 取代 struct in_addr，要確定有將 "6" 新增到適當的欄位〔參考上面的structs〕。
3. 使用 inet_pton() 取代 inet_aton() 或 inet_addr()。
4. 使用 inet_ntop() 取代 inet_ntoa()。
5. 使用很讚的 getaddrinfo() 取代 gethostbyname()。
6. 使用很讚的 getnameinfo() 取代 gethostbyaddr()〔雖然 gethostbyaddr()在 IPv6 中也能正常運作〕。

7. 不要用 INADDR_BROADCAST 了，請愛用 IPv6 multicast 來替代。

就是這樣。

譯註： IPv6 可參考萩野純一郎, IPv6 網路程式設計, 博碩, 2004.

System call 或 Bust

我們在本章開始討論如何讓你存取 UNIX 系統或 BSD、Windows、Linux、Mac 等系統的 system call（系統呼叫）、socket API 及其它 function calls（函式呼叫）等網路功能。當你呼叫其中一個函式時，kernel 會接管，並且自動幫你處理全部的工作。

多數人會卡在這裡是因爲不知道要用什麼樣的順序來呼叫這些函式，而你在找 man 使用手冊時會覺得手冊很難用。好的，爲了要幫忙解決這可怕的困境，我已經試著在下列的章節精確地勾勒出（layout） system call，你在寫程式時只要照著一樣的順序呼叫就可以了。

爲了要連結一些程式碼，需要一些牛奶跟餅乾〔這恐怕你要自行準備〕，以及一些決心與勇氣，而你就能將資料發送到網際網路上，彷彿是 Jon Postel 之子般。

〔請注意，爲了簡潔，下列許多程式碼片段並沒有包含錯誤檢查的程式碼。而且它們很愛假設呼叫 getaddrinfo() 的結果都會成功，並會傳回鏈結串列（link-list）中的一個有效資料。這兩種情況在單獨執行的程式都有嚴謹的定位，所以，還是將它們當作模型來使用吧。〕

getaddrinfo()－準備開始！

這是個有很多選項的工作馬（workhorse）函式，但是卻相當容易上手。它幫你設定之後需要的 struct。

談點歷史：它前身是你用來做 DNS 查詢的 gethostbyname()。而當時你需要手動將資訊載入 struct sockaddr_in，並在你的呼叫中使用。

感謝老天，現在已經不用了。〔如果你想要設計能通用於 IPv4 與 IPv6 的程式也不用！〕在現代，你有 getaddrinfo() 函式，可以幫你做許多事情，包含 DNS 與 service name 查詢，並填好你所需的 structs。

讓我們來看看！

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node, // 例如： "www.example.com" 或 IP
                const char *service, // 例如： "http" 或 port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

你給這個函式三個輸入參數，結果它會回傳給你一個指向鏈結串列的指標 － res。

node 參數是要連線的主機名稱，或者一個 IP address（位址）。

下一個參數是 service，這可以是 port number，像是 "80"，或者特定服務的名稱〔可以在你 UNIX 系統上的 IANA Port List [17] 或 /etc/services 檔案中找到〕，像是 "http" 或 "ftp" 或 "telnet" 或 "smtp" 諸如此類的。

最後，hints 參數指向一個你已經填好相關資訊的 struct addrinfo。

這裡是一個呼叫範例，如果你是一部 server（伺服器），想要在你主機上的 IP address 及 port 3490 執行 listen。要注意的是，這邊實際上沒有做任何的 listening 或網路設定；它只有設定我們之後要用的 structures 而已。

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo; // 將指向結果

memset(&hints, 0, sizeof hints); // 確保 struct 爲空
hints.ai_family = AF_UNSPEC; // 不用管是 IPv4 或 IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE; // 幫我填好我的 IP

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
  fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
  exit(1);
}

// servinfo 目前指向一個或多個 struct addrinfos 的鏈結串列

// ... 做每件事情，一直到你不再需要 servinfo  ....

freeaddrinfo(servinfo); // 釋放這個鏈結串列
```

注意一下，我將 ai_family 設定爲 AF_UNSPEC，這樣代表我不用管我們用的是 IPv4 或 IPv6 address。如果你想要指定的話，你可以將它設定爲 AF_INET 或 AF_INET6。

還有，你會在這裡看到 AI_PASSIVE 旗標；這個會告訴 getaddrinfo() 要將我本機的位址（address of local host）指定給 socket structure。這樣很棒，因爲你就不用把位址寫死了〔或者你可以將特定的位址放在 getaddrinfo() 的第一個參數中，我現在寫 NULL 的那個參數〕。

然後我們執行呼叫，若有錯誤發生時〔getaddrinfo 會傳回非零的值〕，如你所見，我們可以使用 gai_strerror() 函式將錯誤印出來。若每件事情都正常運作，那麼 serinfo 就會指向一個 struct addrinfos 的鏈結串列，串列中的每個成員都會包含一個我們之後會用到的某種 struct sockaddr。

最後，當我們終於使用 getaddrinfo() 配置的鏈結串列完成工作後，我們可以〔也應該〕要呼叫 freeaddrinfo() 將鏈結串列全部釋放。

這邊有一個呼叫範例，如果你是一個想要連線到特定 server 的 client（客户端），比如是："www.example.net" 的 port 3490。再次強調，這裡並沒有眞的進行連線，它只是設定我們之後要用的 structure。

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo; // 將指向結果

memset(&hints, 0, sizeof hints); // 確保 struct 爲空
hints.ai_family = AF_UNSPEC; // 不用管是 IPv4 或 IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
```

```
// 準備好連線
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo 現在指向有一個或多個 struct addrinfos 的鏈結串列
```

我一直說 serinfo 是一個鏈結串列，它有各種的位址資訊。讓我們寫一個能快速 demo 的程式，來呈現這個資訊。這個小程式 [18] 會印出你在命令列中所指定的主機之 IP address：

```
/*
** showip.c -- 顯示命令列中所給的主機 IP address
*/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
  struct addrinfo hints, *res, *p;
  int status;
  char ipstr[INET6_ADDRSTRLEN];

  if (argc != 2) {
    fprintf(stderr,"usage: showip hostname\n");
    return 1;
  }

  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC; // AF_INET 或 AF_INET6 可以指定版本
  hints.ai_socktype = SOCK_STREAM;

  if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
    return 2;
  }

  printf("IP addresses for %s:\n\n", argv[1]);

  for(p = res;p != NULL; p = p->ai_next) {
    void *addr;
    char *ipver;

    // 取得本身位址的指標，
    // 在 IPv4 與 IPv6 中的欄位不同：
    if (p->ai_family == AF_INET) { // IPv4
      struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
      addr = &(ipv4->sin_addr);
      ipver = "IPv4";
    } else { // IPv6
      struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
      addr = &(ipv6->sin6_addr);
```

```
        ipver = "IPv6";
      }

      // convert the IP to a string and print it:
      inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
      printf(" %s: %s\n", ipver, ipstr);
    }

    freeaddrinfo(res); // 釋放鏈結串列

    return 0;
}
```

如你所見，程式碼使用你在命令列輸入的參數呼叫 getaddrinfo()，它填好 res 所指的鏈結串列，並接著我們就能重複那行並印出東西或做點類似的事。

〔有點不好意思！我們在討論 struct sockaddrs 它的型別差異是因 IP 版本而異之處有點鄙俗。我不確定是否有較優雅的方法。〕

在下面執行範例！來看看大家喜歡看的執行畫面：

```
$ showip www.example.net
IP addresses for www.example.net:

  IPv4: 192.0.2.88

$ showip ipv6.example.com
IP addresses for ipv6.example.com:

  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

現在已經在我們的掌控之下，我們會將 getaddrinfo() 傳回的結果送給其它的 socket 函式，而且終於可以建立我們的網路連線了！

讓我們繼續看下去！

[17] http://www.iana.org/assignments/port-numbers

[18] http://beej.us/guide/bgnet/examples/showip.c

[19] http://tools.ietf.org/html/rfc1413

socket()－取得 File Descriptor！

我想可以不用再將 socket() 晾在旁邊了，我一定要講一下 socket() system call，這邊是程式片段：

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

可是這些參數是什麼？

它們可以讓你設定想要的 socket 類型（IPv4 或 IPv6，stream 或 datagram 以及 TCP 或 UDP）。

以前的人得寫死這些值，而你也可以這樣做。（domain 是 PF_INET 或 PF_INET6，type 是 SOCK_STREAM 或 SOCK_DGRAM，而 protocol 可以設定爲 0，用來幫給予的 type 選擇適當的協定。或者你可以呼叫 getprotobyname() 來查詢你想要的協定，"tcp" 或 "udp"）。

PF_INET 就是你在初始化 struct sockaddr_in 的 sin_family 欄位會用到的，它是 AF_INET 的親戚。實際上，它們的關係很近，所以其實它們的值也都一樣，而許多程式設計師會呼叫 socket()，並以 AF_INET 取代 PF_INET 來做爲第一個參數傳遞。

現在，你可以去拿點牛奶跟餅乾，因爲又是說故事時間了。

在很久很久以前，人們認爲它應該是位址家族（address family），就是 "AF_INET" 中的 "AF" 所代表的意思；而位址家族也要支援協定家族（protocol family）的幾個協定，這件事並沒有發生，而之後它們都過著幸福快樂的日子，結束。

所以最該做的事情就是在你的 struct sockaddr_in 中使用 AF_INET，而在呼叫 socket() 時使用 PF_INET。

總之，這樣就夠了。你眞的該做的只是使用呼叫 getaddrinfo() 得到的值，並將這個值直接餵給 socket()，像這樣：

```
int s;
struct addrinfo hints, *res;

// 執行查詢
// [假裝我們已經填好 "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// [再來，你應該要對 getaddrinfo() 進行錯誤檢查, 並走到 "res" 鏈結串列查詢能用的資料，
// 而不是假設第一筆資料就是好的〔像這些範例一樣〕
// 實際的範例請參考 client/server 章節。
s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

socket() 單純傳回給你一個之後 system call 要用的 socket descriptor，錯誤時會回傳 -1。errno 全域變數會設定爲該錯誤的值（細節請見 errno 的 man 使用手冊，而且你需要繼續閱讀並執行更多與它相關的 system call，這樣會比較有感覺）。

bind()－ 我在哪個 port？

一旦你有了一個 socket，你會想要將這個 socket 與你本機上的 port 進行關聯（如果你正想要 listen() 特定 port 進入的連線，通常都會這樣做，比如：多人網路連線遊戲在它們告訴你”連線到 192.168.5.10 port 3490”時這麼做）。port number 是用來讓 kernel 可以比對出進入的封包是屬於哪個 process 的 socket descriptor。如果你只是正在進行 connect()（因爲你是 client，而不是 server），這可能就不用。不過還是可以讀讀，好玩嘛。

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

sockfd 是 socket() 傳回的 socket file descriptor。my_addr 是指向包含你的位址資訊、名稱及 IP address 的 struct sockaddr 之指標。addrlen 是以 byte 爲單位的位址長度。

呼！有點比較好玩了。我們來看一個範例，它將 socket bind（綁定）到執行程式的主機上，port 是 3490：

```
struct addrinfo hints, *res;
int sockfd;

// 首先，用 getaddrinfo() 載入位址結構：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// 建立一個 socket：

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// 將 socket bind 到我們傳遞給 getaddrinfo() 的 port：

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

使用 AI_PASSIVE 旗標，我可以跟程式說要 bind 它所在主機的 IP。如果你想要 bind 到指定的本地 IP address，捨棄 AI_PASSIVE，並改放一個位址到 getaddrinfo() 的第一個參數。

bind() 在錯誤時也會傳回 -1，並將 errno 設定爲該錯誤的值。

許多舊程式都在呼叫 bind() 以前手動封裝 struct sockaddr_in。很顯然地，這是 IPv4 才有的，可是眞的沒有辦法阻止你在 IPv6 做一樣的事情，一般來說，使用 getaddrinfo() 會比較簡單。總之，舊版的程式看起來會像這樣：

```
// !!! 這 是 老 方 法 !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT); // short, network byte order
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

在上列的程式碼中，如果你想要 bind 到你本機的 IP address（就像上面的 AI_PASSIVE 旗標），你也可以將 INADDR_ANY 指定給 s_addr 欄位。INADDR_ANY 的 IPv6 版本是一個 in6addr_any 全域變數，它會被指定給你的 struct sockaddr_in6 的 sin6_addr 欄位。

「也有一個你能用於變數初始器（variable initializer）的 IN6ADDR_ANY_INIT macro（巨集）」

另一件呼叫 bind() 時要小心的事情是：不要用太小的 port number。全部 1024 以下的 ports 都是保留的（除非你是系統管理員）！你可以使用任何 1024 以上的 port number，最高到 65535（提供尚未被其它程式使用的）。

你可能有注意到，有時候你試著重新執行 server，而 bind() 卻失敗了，它聲稱”Address already in use.”（位址使用中）。這是什麼意思呢？很好，有些連接到 socket 的連線還懸在 kernel 裡面，而它佔據了這個 port。你可以等待它自行清除（一分鐘之類），或者在你的程式中新增程式碼，讓它重新使用這個 port，類似這樣：

```
int yes=1;
//char yes='1'; // Solaris 的使用者用這個

// 可以跳過 "Address already in use" 錯誤訊息

if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
  perror("setsockopt");
  exit(1);
}
```

最後一個對 bind() 的額外小提醒：在你不願意呼叫 bind() 時。若你正使用 connect() 連線到遠端的機器，你可以不用管 local port 是多少（以 telnet 爲例，你只管遠端的 port 就好），你可以單純地呼叫 connect()，它會檢查 socket 是否尚未綁定（unbound），並在有需要的時候自動將 socket bind() 到一個尚未使用的 local port。

connect()，嘿！你好。

咱們用幾分鐘的時間假裝你是個 telnet 應用程式，你的使用者命令你（就像 TRON 電影裡那樣）取得一個 socket file descriptor。你執行並呼叫 socket()。接著使用者告訴你連線到 "10.12.110.57" 的 port 23（標準 telnet port）。喲！你現在該做什麼呢？

你是很幸運的程式，你現在可以細讀 connect() 的章節，如何連線到遠端主機。所以努力往前讀吧！刻不容緩！

connect() call 如下：

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

sockfd 是我們的好鄰居 socket file descriptor，如同 socket() 呼叫所傳回的，serv_addr 是一個 struct sockaddr，包含了目的 port 及 IP 位址，而 addrlen 是以 byte 爲單位的 server 位址結構之長度。

全部的資訊都可以從 getaddrinfo() 呼叫中取得，它很棒。

這樣有開始比較有感覺了嗎？我在這裡沒辦法知道，所以我只能希望是這樣沒錯。

我們有個範例，這邊我們用 socket 連線到 "www.example.com" 的 port 3490：

```
struct addrinfo hints, *res;
int sockfd;

// 首先，用 getaddrinfo() 載入 address structs：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// 建立一個 socket：

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!
connect(sockfd, res->ai_addr, res->ai_addrlen);
```

老學校的程式再次填滿了它們自己的 struct sockaddr_ins 並傳給 connect()。如果你願意的話，你可以這樣做。請見上面 bind() 章節中類似的提點。

要確定有檢查 connect() 的回傳值，它在錯誤時會傳回 -1，並設定 errno 變數。

還要注意的是，我們不會呼叫 bind()。基本上，我們不用管我們的 local port number；我們只在意我們的目地（遠端 port）。Kernel 會幫我們選擇一個 local port，而我們要連線的站台會自動從我們這裡取得資訊，不用擔心。

listen()－有人會呼叫我嗎？

OK，是該改變步調的時候了。如果你不想要連線到一個遠端主機要怎麼做。

我說過，好玩就好，你想要等待進入的連線，並以某種方式處理它們。

這個過程有兩個步驟：你要先呼叫 listen()，接著呼叫 accept()（參考下一節）。

listen 呼叫相當簡單，不過需要一點說明：

```
int listen(int sockfd, int backlog);
```

sockfd 是來自 socket() system call 的一般 socket file descriptor。backlog 是進入的佇列（incoming queue）中所允許的連線數目。這代表什麼意思呢？好的，進入的連線將會在這個佇列中排隊等待，直到你 accept() 它們（請見下節），而這限制了排隊的數量。多數的系統預設將這個數值限制爲 20；你或許可以一開始就將它設定爲 5 或 10。

再來，如同往常，listen() 會傳回 -1 並在錯誤時設定 errno。

好的，你可能會想像，我們需要在呼叫 listen() 以前呼叫 bind()，讓 server 可以在指定的 port 上執行。〔你必須能告訴你的好朋友要連線到哪一個 port！〕所以如果你正在 listen 進入的連線，你會執行的 system call 順序是：

```
getaddrinfo();
socket();
bind();
listen();
/* accept() 從這裡開始 */
```

我只是留下範例程式的位置，因爲它相當顯而易見。（在下面 accept() 章節中的程式碼會比較完整）。這整件事情眞正需要技巧的部分是呼叫 accept()。

accept()— 謝謝你 call 3490 port

準備好，accept() call 是很奇妙的！會發生的事情就是：很遠的人會試著 connect() 到你的電腦正在 listen() 的 port。他們的連線會排隊等待被 accept()。你呼叫 accept()，並告訴它要取得擱置的（pending）連線。它會傳回給你專屬這個連線的一個新 socket file descriptor！那是對的，你突然有了兩個 socket file descriptor！原本的 socket file descriptor 仍然正在 listen 後續的連線，而新建立的 socket file descriptor 則是在最後要準備給 send() 與 recv() 用的。

呼叫如下：

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

sockfd 是正在進行 listen() 的 socket descriptor。很簡單，addr 通常是一個指向 local struct sockaddr_storage 的指標，關於進來的連線將往哪裡去的資訊（而你可以用它來得知是哪一台主機從哪一個 port 呼叫你的）。addrlen 是一個 local 的整數變數，應該在將它的位址傳遞給 accept() 以前，將它設定爲 sizeof(struct sockaddr_storage)。accept() 不會存放更多的 bytes（位元組）到addr。若它存放了較少的 bytes 進去，它會改變 addrlen 的值來表示。

有想到嗎？accept() 在錯誤發生時傳回 -1 並設定 errno。不過 BetCha 不這麼認爲。

跟以前一樣，用一段程式範例會比較好吸收，所以這裡有一段範例程供你細讀：

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT "3490" // 使用者將連線的 port
#define BACKLOG 10 // 在佇列中可以有多少個連線在等待

int main(void)
{
  struct sockaddr_storage their_addr;
  socklen_t addr_size;
  struct addrinfo hints, *res;
  int sockfd, new_fd;

  // !! 不要忘了幫這些呼叫做錯誤檢查 !!

  // 首先，使用 getaddrinfo() 載入 address struct：

  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC; // 使用 IPv4 或 IPv6，都可以
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE; // 幫我填上我的 IP

  getaddrinfo(NULL, MYPORT, &hints, &res);

  // 產生一個 socket，bind socket，並 listen socket：

  sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
  bind(sockfd, res->ai_addr, res->ai_addrlen);
  listen(sockfd, BACKLOG);

  // 現在接受一個進入的連線：

  addr_size = sizeof their_addr;
  new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

  // 準備好與 new_fd 這個 socket descriptor 進行溝通！
  .
  .
  .
```

一樣，我們會將 new_fd socket descriptor 用於 send() 與 recv() 呼叫。若你只是要取得一個連線，你可以用 close() 關閉正在 listen 的 sockfd，以避免有更多的連線進入同樣的 port，若你有這個需要的話。

send() 與 recv()— 寶貝，我們來聊天！

這兩個用來通信的函式是透過 stream socket 或 connected datagram socket。若你想要使用常規的 unconnected datagram socket，你會需要參考底下的 sendto() 及 recvfrom() 的章節。

send() call：

```
int send(int sockfd, const void *msg, int len, int flags);
```

sockfd 是你想要送資料過去的 socket descriptor（無論它是不是 socket() 傳回的，或是你用 accept() 取得的）。msg 是一個指向你想要傳送資料之指標，而 len 是以 byte 爲單位的資料長度。而 flags 設定爲 0 就好。（更多相關的旗標資訊請見 send() man 使用手冊）。

一些範例程式如下：

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

send() 會傳回實際有送出的 byte 數目，這可能會少於你所要傳送的數目！有時候你告訴 send() 要送整筆的資料，而它就是無法處理這麼多資料。它只會盡量將資料送出，並認爲你之後會再次送出剩下沒送出的部分。

要記住，如果 send() 傳回的值與 len 的值不符合的話，你就需要再送出字串剩下的部分。好消息是：如果封包很小（比 1K 還要小這類的），或許有機會一次就送出全部的東西。

一樣，錯誤時會傳回 -1，並將 errno 設定爲錯誤碼（error number）。

recv() 呼叫在許多地方都是類似的：

```
int recv(int sockfd, void *buf, int len, int flags);
```

sockfd 是要讀取的 socket descriptor，buf 是要記錄讀到資訊的緩衝區（buffer），len 是緩衝區的最大長度，而 flags 可以再設定爲 0。〔關於旗標資訊的細節請參考 recv() 的 man 使用手冊〕。

recv() 傳回實際讀到並寫入到緩衝區的 byte 數，而錯誤時傳回 -1（並設定相對的 errno）。

等等！ recv() 會傳回 0，這只能表示一件事情：遠端那邊已經關閉了你的連線！recv() 傳回 0 的值是讓你知道這件事情。

這樣很簡單，不是嗎？你現在可以送回資料，並往 stream sockets 邁進！嘻嘻！你是 UNIX 網路程式設計師了。

sendto() 與 recvfrom()－ 來點 DGRAM

我聽到你說，「這全部都是上等的好貨，可是我該如何使用 unconnected datagram socket 呢？」

沒問題，朋友。我們正要講這件事。

因爲 datagram socket 沒有連線到到遠端主機，猜猜看，我們在送出封包以前會需要哪些資訊呢？

對！目的位址！在這裡搶先看：

```
sendto(int sockfd, const void *msg, int len, unsigned int flags,
       const struct sockaddr *to, socklen_t tolen);
```

如你所見，這個呼叫基本上與呼叫 send() 一樣，只是多了兩個額外的資訊。to 是一個指向 struct sockaddr（這或許是另一個你可以在最後轉型的 struct sockaddr_in 或 struct sockaddr_in6 或 struct sockaddr_storage）的指標，它包含了目的 IP address 與 port。tolen 是一個 int，可以單純地將它設定爲 sizeof *to 或 sizeof(struct sockaddr_storage)。

爲了能自動處理目的位址結構（destination address structure），你或許可以用底下的 getaddrinfo() 或 recvfrom()，或者你也可以手動填上。

如同 send()，sendto() 會傳回實際已傳送的資料數量（一樣，可能會少於你要傳送的資料量！）而錯誤時傳回 -1。

recv() 與 recvfrom() 也是差不多的。recvfrom() 的對照如下：

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

一樣，它跟 recv() 很像，只是多了兩個欄位。from 是指向 local struct sockaddr_storage 的指標，這個資料結構包含了封包來源的 IP address 與 port。fromlen 是指向 local int 的指標，應該要初始化爲 sizeof *from 或是 sizeof(struct sockaddr_storage)。當函式傳回時，fromlen 會包含實際上儲存於 from 中的位址長度。

recvfrom() 傳回接收的資料數目，或在發生錯誤時傳回 -1〔並設定相對的 errno〕。

所以這裡有個問題：爲什麼我們要用 struct sockaddr_storage 做爲 socket 的型別呢？爲什麼不用 struct sockaddr_in 呢？

因爲你知道的，我們不想要讓自己綁在 IPv4 或 IPv6，所以我們使用通用的泛型 struct sockaddr_storage，我們知道這樣有足夠的空間可以用在 IPv4 與 IPv6。

（所以 ... 這裡有另一個問題：爲什麼不是 struct sockaddr 本身就可以容納任何位址呢？我們甚至可以將通用的 struct sockaddr_storage 轉型爲通用的 struct sockaddr！似乎沒什麼關係又很累贅啊。答案是，它就是不夠大，我猜在這個時候更動它會有問題，所以他們就弄了一個新的。）

記住，如果你 connect() 到一個 datagram socket，你可以在你全部的交易中只使用 send() 與 recv()。socket 本身仍然是 datagram socket，而封包仍然使用 UDP，但是 socket interface 會自動幫你增加目的與來源資訊。

close() 與 shutdown()－ 你消失吧！

呼！你已經整天都在 send() 與 recv()了。你正準備要關閉你 socket descriptor 的連線，這很簡單，你只要使用常規的 UNIX file descriptor close() 函式：

```
close(sockfd);
```

這會避免對 socket 做更多的讀寫。任何想要對這個遠端的 socket 進行讀寫的人都會收到錯誤。

如果你想要能多點控制 socket 如何關閉，可以使用 shutdown() 函式。它讓你可以切斷單向的通信，或者雙向（就像是 close() 所做的），這是函式原型：

```
int shutdown(int sockfd, int how);
```

sockfd 是你想要 shutdown 的 socket file descriptor，而 how 是下列其中一個值：

- 0 不允許再接收資料
- 1 不允許再傳送資料
- 2 不允許再傳送與接收資料（就像 close()）

shutdown() 成功時傳回 0，而錯誤時傳回 -1（設定相對的 errno）。

若你在 unconnected datagram socket 上使用 shutdown()，它只會單純的讓 socket 無法再進行 send() 與 recv() 呼叫（要記住你只能在有 connect() 到 datagram socket 的時候使用）。

重要的是 shutdown() 實際上沒有關閉 file descriptor，它只是改變了它的可用性。如果要釋放 socket descriptor，你還是需要使用 close()。

沒了。

（除了要記得的是，如果你用 Windows 與 Winsock，你應該要呼叫 closesocket() 而不是 close()）。

getpeername()－你是誰？

這個函式很簡單。

它太簡單了，我幾乎不想給它一個獨立的章節，雖然還是給了。

getpeername() 函式會告訴你另一端連線的 stream socket 是誰，函式原型如下：

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

sockfd 是連線的 stream socket 之 descriptor，addr 是指向 struct sockaddr（或 struct sockaddr_in）的指標，這個資料結構儲存了連線另一端的資訊，而 addrlen 則是指向 int 的指標，應該將它初始化爲 sizeof *addr 或 sizeof(struct sockaddr)。

函式在錯誤時傳回 -1，並設定相對的 errno。

一旦你取得了它們的位址，你就可以用 inet_ntop()、getnameinfo() 或 gethostbyaddr() 印出或取得更多的資訊。不過你無法取得它們的登入帳號。

（好好好，如果另一台電腦執行的是 ident daemon 就可以）。然而，這個已經超出本文的範圍，更多資訊請參考 RFC 1413 [19]。

gethostname()－我是誰？

比 getpeername() 更簡單的函式是 gethostname()，它會傳回你執行程式的電腦名稱，這個名稱之後可以用在 gethostbyname()，用來定義你本機電腦的 IP address。

有什麼更有趣的嗎？

我可以想到一些事情，不過這不適合 socket 程式設計，總之，下面是一段範例：

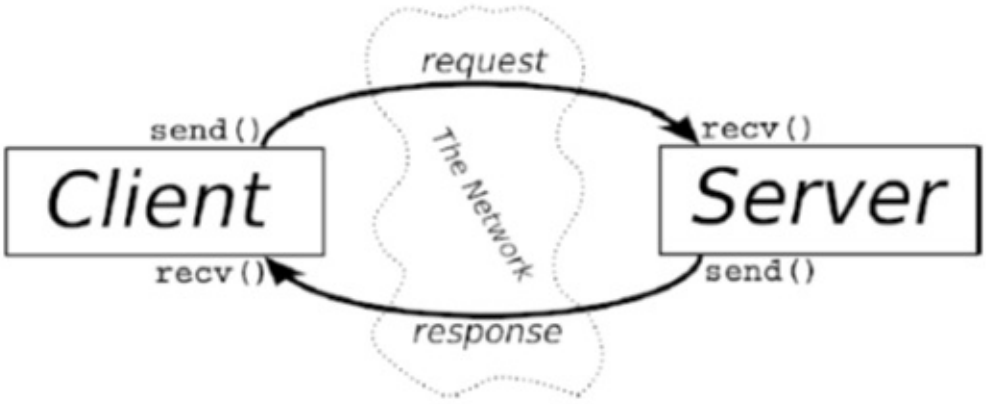
```
#include <unistd.h>
int gethostname(char *hostname, size_t size);
```

參數很簡單：hostname 是指向字元陣列（array of chars）的指標，它會儲存函式傳回的主機名稱（hostname），而 size 是以 byte 爲單位的主機名稱長度。

函式在成功執行時傳回 0，在錯誤時傳回 -1，並一樣設定 errno。

Client-Server 基礎

寶貝，這是個 client-server（客户端-伺服端）世界。單純與網路處理 client process（客户端行程）及 server process（伺服端行程）溝通的每件事情有關，反之亦然。以 telnet 爲例，當你用 telnet（client）連線到遠端主機的 port 23 時，主機上的程式（稱爲 telnetd server）就開始動了起來，它會處理進來的 telnet 連線，並幫你設定一個登入提示字元等。



Client 與 server 間的訊息交換摘錄於上列的流程圖中。

需要注意的是 client-server pair 可以使用 SOCK_STREAM、SOCK_DGRAM 或其它的（只要它們用一樣的協定來溝通）。有一些不錯的 client-server pair 範例，如：telnet/telnetd、ftp/ftpd 或 Firefox/Apache。每次你使用 ftp 時，都會有一個 ftpd 遠端程式來爲你服務。

一台機器上通常只會有一個 server，而該 server 會利用 fork()來處理多個 clients。基本的機制（routine）是：server 會等待連線、accept() 連線，並且 fork() 一個 child process（子行程）來處理此連線。這就是我們在下一節的 server 範本所做的工作。

簡單的 Stream Server

這個 server 所做的事情就是透過 stream connection（串流連線）送出 "Hello, World!\n" 字串。你所需要做就是用一個視窗來測試執行 server，並用另一個視窗來 telnet 到 server：

```
$ telnet remotehostname 3490
```

這裡的 remotehostname 就是你執行 server 的主機名稱。

Server的程式碼如下 [20]：

```
/*
** server.c – 展示一個stream socket server
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#define PORT "3490" // 提供給使用者連線的 port
#define BACKLOG 10 // 有多少個特定的連線佇列（pending connections queue）

void sigchld_handler(int s)
{
  while(waitpid(-1, NULL, WNOHANG) > 0);
}

// 取得sockaddr，IPv4或IPv6：
void *get_in_addr(struct sockaddr *sa)
{
  if (sa->sa_family == AF_INET) {
    return &(((struct sockaddr_in*)sa)->sin_addr);
  }
  return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
  int sockfd, new_fd; // 在 sock_fd 進行 listen，new_fd 是新的連線
  struct addrinfo hints, *servinfo, *p;
```

```
struct sockaddr_storage their_addr; // 連線者的位址資訊
socklen_t sin_size;
struct sigaction sa;
int yes=1;
char s[INET6_ADDRSTRLEN];
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // 使用我的 IP

if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
  return 1;
}

// 以迴圈找出全部的結果，並綁定（bind）到第一個能用的結果
for(p = servinfo; p != NULL; p = p->ai_next) {
  if ((sockfd = socket(p->ai_family, p->ai_socktype,
    p->ai_protocol)) == -1) {
    perror("server: socket");
    continue;
  }

  if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
    sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
  }

  if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
    close(sockfd);
    perror("server: bind");
    continue;
  }

  break;
}

if (p == NULL) {
  fprintf(stderr, "server: failed to bind\n");
  return 2;
}

freeaddrinfo(servinfo); // 全部都用這個 structure

if (listen(sockfd, BACKLOG) == -1) {
  perror("listen");
  exit(1);
}

sa.sa_handler = sigchld_handler; // 收拾全部死掉的 processes
```

```
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
      perror("sigaction");
      exit(1);
    }

    printf("server: waiting for connections...\n");

    while(1) { // 主要的 accept() 迴圈
      sin_size = sizeof their_addr;
      new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
      if (new_fd == -1) {
        perror("accept");
        continue;
      }

      inet_ntop(their_addr.ss_family,
        get_in_addr((struct sockaddr *)&their_addr),
        s, sizeof s);
      printf("server: got connection from %s\n", s);

      if (!fork()) { // 這個是 child process
        close(sockfd); // child 不需要 listener

        if (send(new_fd, "Hello, world!", 13, 0) == -1)
          perror("send");

        close(new_fd);

        exit(0);
      }

      close(new_fd); // parent 不需要這個
    }

    return 0;
  }
```

趁著你對這個例子還感到很好奇，我爲了讓句子比較清楚（我個人覺得），所以將程式碼放在一個大的 main() 函式中，如果你覺得將它分成幾個小一點的函式會比較好的話，可以儘管去做。

「還有，sigaction() 這個東西對你而言應該是蠻陌生的。沒有關係，這個程式碼是用來清理 zombie process（殭屍行程），當 parent process 所 fork() 出來的 child process 結束時，且 parent process 沒有取得 child process 的離開狀態時，就會出現 zombie process。如果你產生了許多 zombie，但卻無法清除他們時，你的系統管理員就會開始焦慮不安了。」

你可以利用下一節所列出的 client，來取得 server 的資料。

[20] http://beej.us/guide/bgnet/examples/server.c

簡單的 Stream Client

Client 這傢伙比 server 簡單多了，client 所需要做的就是：連線到你在命令列所指定的主機 3490 port，接著，client 會收到 server 送回的字串。

Client 的程式碼 [21]：

[21] http://beej.us/guide/bgnet/examples/client.c

```
/*
/*
** client.c -- 一個 stream socket client 的 demo
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define PORT "3490" // Client 所要連線的 port
#define MAXDATASIZE 100 // 我們一次可以收到的最大位原組數（number of bytes）

// 取得 IPv4 或 IPv6 的 sockaddr：
void *get_in_addr(struct sockaddr *sa)
{
  if (sa->sa_family == AF_INET) {
    return &(((struct sockaddr_in*)sa)->sin_addr);
  }

  return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
  int sockfd, numbytes;
  char buf[MAXDATASIZE];
  struct addrinfo hints, *servinfo, *p;
  int rv;
  char s[INET6_ADDRSTRLEN];

  if (argc != 2) {
    fprintf(stderr,"usage: client hostname\n");
    exit(1);
  }
```

```
  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;

  if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
  }

  // 用迴圈取得全部的結果，並先連線到能成功連線的
   for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
      p->ai_protocol)) == -1) {
      perror("client: socket");
      continue;
    }

    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
      close(sockfd);
      perror("client: connect");
      continue;
    }

    break;
  }

  if (p == NULL) {
    fprintf(stderr, "client: failed to connect\n");
    return 2;
  }

  inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr), s, sizeof s);

  printf("client: connecting to %s\n", s);

  freeaddrinfo(servinfo); // 全部皆以這個 structure 完成

  if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
    perror("recv");
    exit(1);
  }

  buf[numbytes] = '\0';
  printf("client: received '%s'\n",buf);

  close(sockfd);
  return 0;
}
```

要注意的是，你如果沒有在執行 client 以前先啓動 server 的話，connect() 會傳回 "Connection refused"，這個訊息很有幫助。

Datagram Sockets

我們已經在討論 sendto() 與 recvfrom() 時涵蓋了 UDP datagram socket 的基礎，所以我會展示一對範例程式：talker.c 與 listener.c。

Listener 位於一台機器中，等待進入 port 4950 的封包。Talker 則從指定的機器傳送封包給這個 port，封包的內容包含使用者從命令列所輸入的資料。

這裡就是 listener.c 的原始程式碼 [22]：

```
/*
** listener.c -- 一個 datagram sockets "server" 的 demo
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MYPORT "4950" // 使用者所要連線的 port
#define MAXBUFLEN 100

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
  if (sa->sa_family == AF_INET) {
    return &(((struct sockaddr_in*)sa)->sin_addr);
  }

  return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
  int sockfd;
  struct addrinfo hints, *servinfo, *p;
  int rv;
  int numbytes;
  struct sockaddr_storage their_addr;
  char buf[MAXBUFLEN];
  socklen_t addr_len;
  char s[INET6_ADDRSTRLEN];
```

```
    memset(&hints, 0, sizeof hints);

    hints.ai_family = AF_UNSPEC; // 設定 AF_INET 以強制使用 IPv4
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE; // 使用我的 IP

    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
      return 1;
    }

    // 用迴圈來找出全部的結果，並 bind 到首先找到能 bind 的
    for(p = servinfo; p != NULL; p = p->ai_next) {

      if ((sockfd = socket(p->ai_family, p->ai_socktype,
        p->ai_protocol)) == -1) {
        perror("listener: socket");
        continue;
      }

      if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("listener: bind");
        continue;
      }

      break;
    }

    if (p == NULL) {
      fprintf(stderr, "listener: failed to bind socket\n");
      return 2;
    }

    freeaddrinfo(servinfo);
    printf("listener: waiting to recvfrom...\n");
    addr_len = sizeof their_addr;

    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
          (struct sockaddr *)&their_addr, &addr_len)) == -1) {

      perror("recvfrom");
      exit(1);
    }

    printf("listener: got packet from %s\n",

    inet_ntop(their_addr.ss_family,

    get_in_addr((struct sockaddr *)&their_addr), s, sizeof s));

    printf("listener: packet is %d bytes long\n", numbytes);
```

```
  buf[numbytes] = '\0';

  printf("listener: packet contains \"%s\"\n", buf);

  close(sockfd);

  return 0;
}
```

要注意的是，在我們呼叫 getaddrinfo()時，我們是使用 SOCK_DGRAM。還要注意到，不需要 listen()或是 accept()，這是使用（免連線）datagram sockets 的一個好處！

接著是 talker.c 的程式碼 [23]：

```
/*
** talker.c -- 一個 datagram "client" 的 demo
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT "4950" // 使用者所要連線的 port

int main(int argc, char *argv[])
{
  int sockfd;
  struct addrinfo hints, *servinfo, *p;
  int rv;
  int numbytes;

  if (argc != 3) {
    fprintf(stderr,"usage: talker hostname message\n");
    exit(1);
  }

  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_DGRAM;

  if ((rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
  }
```

```
    // 用迴圈找出全部的結果，並產生一個 socket
    for(p = servinfo; p != NULL; p = p->ai_next) {
      if ((sockfd = socket(p->ai_family, p->ai_socktype,
        p->ai_protocol)) == -1) {
        perror("talker: socket");
        continue;
      }

      break;
    }

    if (p == NULL) {
      fprintf(stderr, "talker: failed to bind socket\n");
      return 2;
    }

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
      p->ai_addr, p->ai_addrlen)) == -1) {

      perror("talker: sendto");
      exit(1);
    }

    freeaddrinfo(servinfo);
    printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);
    close(sockfd);
    return 0;
}
```

全部就這些了！在某個機器上執行 listener，接著在另一台機器執行 talker。觀察它們的溝通！這眞的超有趣。

這次你甚至不用執行 server！可以只執行 talker，而它只會很開心的將封包丟到網路上，如果另一端沒有人用 recvfrom()來接收的話，這些封包就只是消失而已。

要記得：使用 UDP datagram socket 傳送的資料是不會使命必達的！

我要再提之前提過無數次的小細節：connected datagram socket。我在這裡要再講一下，因爲我們正在 datagram 這個章節。

我們說 talker 呼叫 connect()並指定 listener 的位址。從這開始，talker 就只能從 connect()所指定的位址進行傳送與接收。因此，你不用使用 sendto()與 recvfrom()，可以單純使用 send()與 recv() 就好。

[22] http://beej.us/guide/bgnet/examples/listener.c

[23] http://beej.us/guide/bgnet/examples/talker.c

進階技術

本章的技術沒有多高明，只是比前幾章略高一籌。其實，如果你已經達到這個境界，你應該會覺得自己在 Unix 網路程式設計已經相當厲害！恭喜！

所以，我們要勇於面對你想學的 socket 新世界，本章會有些較深奧的內容。

Blocking（阻塞）

你聽過 blocking，只是它在這裡代表什麼鬼東西呢？簡而言之，"block" 就是 "sleep（休眠）" 的技術術語。你以前執行 listener 時可能有注意到，它只是一直在那邊等待，直到有封包抵達才繼續動作。

很多函式都會 block，accept() 會 block，全部的 recv() 函式都會 block。原因是它們有權這麼做。

當你先用 socket() 建立 socket descriptor 時，kernel（核心）會將它設定爲 blocking。若你不想要 blocking socket，你必須呼叫 fcntl()：

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
.
```

將 socket 設定爲 non-blocking（非阻塞），你就能 "poll（輪詢）" socket 以取得資訊。如果你試著讀取 non-blocking socket，而 socket 沒有資料時，函式就不會發生 block，而是傳回 -1，並將 errno 設定爲 EWOULDBLOCK。

然而，一般來說，這樣 polling 是不好的想法。如果你讓程式一直忙著查 socket 上是否有資料，則會浪費 CPU 的時間，這樣是不合適的。比較漂亮的解法是利用下一節的 select() 來檢查 socket 是否有資料需要讀取。

select()：同步 I/O 多工

這個函式有點特別，不過它很好用。看看下面這個情況：如果你是一個 server，而你想要 listen 正在進來的連線，如同不斷讀取已建立的連線 socket 一樣。

你說：沒問題，只要用 accept() 及一對 recv() 就好了。

慢點，老兄！如果你在 accept() call 時發生了 blocking 該怎麼辦呢？你要如何同時進行 recv() 呢？

「那就使用 non-blocking socket！」

不行！你不會想成爲浪費 CPU 資源的罪人吧。

嗯，那有什麼好方法嗎？

select() 授予你同時監視多個 sockets 的權力，它會告訴你哪些 sockets 已經有資料可以讀取、哪些 sockets 已經可以寫入，如果你眞的想知道，還可以告訴你哪些 sockets 觸發了例外。

即使 select() 有相當好的可移植性，不過卻是最慢的監視 sockets 方法。一個比較可行的替代方案是 libevent [24] 或者其它類似的方法，將全部的系統相依要素封裝起來，用在取得 socket 的通知。

好了，不囉唆，下面我提供了 select() 的原型：

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

這個函式以 readfds、writefds 及 exceptfds 分別表示要監視的那一組 file descriptor set（檔案描述符集合）。

如果你想要知道你是否能讀取 standard input（標準輸入）及某個 sockfd socket descriptor，只要將 file descriptor 0 與 sockfd 新增到 readfds set 中。numfds 參數應該要設定爲 file descriptor 的最高值加 1。在這個例子中，應該要將 numfds 設定爲 sockfd+1，因爲它必定大於 standard input（0）。

當 select() 返回時，readfds 會被修改，用來反映你所設定的 file descriptors 中，哪些已經有資料可以讀取，你可以用下列的FD_ISSET() macro（巨集）來取得這些就緒可讀的 file descriptors。

在繼續談下去以前，我想要說說該如何控制這些 sets。

每個 sets 的型別都是 fd_set，下列是用來控制這個型別的 macro：

```
FD_SET(int fd, fd_set *set);     將 fd 新增到 set。
FD_CLR(int fd, fd_set *set);     從 set 移除 fd。
FD_ISSET(int fd, fd_set *set);   若 fd 在 set 中，傳回 true。
FD_ZERO(fd_set *set);            將 set 整個清爲零。
```

最後，這個令人困惑的 struct timeval 是什麼東西呢？

好，有時你不想要一直花時間在等人家送資料給你，或者明明沒什麼事，卻每 96 秒就要印出 ”執行中 ...” 到終端機（terminal），而這個 time structure 讓你可以設定 timeout 的週期。

如果時間超過了，而 select() 還沒有找到任何就緒的 file descriptor 時，它就會返回，讓你可以繼續做其它事情。

```
struct timeval 的欄位如下：
struct timeval {
  int tv_sec; // 秒（second）
  int tv_usec; // 微秒（microseconds）
};
```

只要將 tv_sec 設定爲要等待的秒數，並將 tv_usec 設定爲要等待的微秒數。是的，就是微秒，不是毫秒。一毫秒有 1,000 微秒，而一秒有 1,000 毫秒。所以，一秒就有 1,000,000 微秒。

爲什麼要用 ”usec（微秒）” 呢？

”u”看起來很像我們用來表示 ”micro（微）”的希臘字母 μ（Mu）。還有，當函式返回時，會更新 timeout，用以表示還剩下多少時間。這個行爲取決於你所使用的 Unix 而定。

譯註：
因爲有些系統平台的 select() 會修改 timeout 的值，而有些系統不會，所以如果要重複呼叫 select() 的話，每次呼叫之前都應該要重新指定 timeout 的值，以確保程式的行爲在各平台都可以有預期的行爲。

哇！我們有微秒精度的計時器了！

是的，不過別依賴它。無論你將 struct timeval 設定的多小，你可能還要等待一小段的 standard Unix timeslice（標準 Unix 時間片段）。

另一件有趣的事：如果你將 struct timeval 的欄位設定爲 0，select() 會在輪詢過 sets 中的每個 file descriptors 之後，就馬上 timeout。如果你將 timeout 參數設定爲 NULL，它就永遠不會 timeout，並且陷入等待，直到至少一個 file descriptor 已經就緒（ready）。如果你不在乎等待時間，就在呼叫 select() 時將 timeout 參數設定爲 NULL。

下列的程式碼片段 [25] 等待 2.5 秒後，就會出現 standard input（標準輸入）所輸入的東西：

```
/*
** select.c -- a select() demo
*/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0 // standard input 的 file descriptor
int main(void)
{
  struct timeval tv;
  fd_set readfds;

  tv.tv_sec = 2;
  tv.tv_usec = 500000;

  FD_ZERO(&readfds);
  FD_SET(STDIN, &readfds);

  // 不用管 writefds 與 exceptfds：
  select(STDIN+1, &readfds, NULL, NULL, &tv);

  if (FD_ISSET(STDIN, &readfds))
    printf("A key was pressed!\n");
  else
    printf("Timed out.\n");
  return 0;
}
```

如果你用一行緩衝區（buffer）的終端機，那麼你從鍵盤輸入資料後應該要盡快按下 Enter，否則程式就會發生 timeout。

你現在可能在想，這個方法用在需要等待資料的 datagram socket 上很棒，而且你是對的：應該是不錯的方法。

有些系統會用這個方式來使用 select()，而有些不行，如果你想要用它，你應該要參考你系統上的 man 使用手冊說明看是否會有問題。

有些系統會更新 struct timeval 的時間，用來反映 select() 原本還剩下多少時間 timeout；不過有些卻不會。如果你想要程式是可移植的，那就不要倚賴這個特性。（如果你需要追蹤剩下的時間，可以使用 gettimeofday()，我知道這很令人失望，不過事實就是這樣。）

如果在 read set 中的 socket 關閉連線，會怎樣嗎？

好的，這個例子的 select() 返回時，會在 socket descriptor set 中說明這個 socket 是”ready to read（就緒可讀）”的。而當你真的用 recv() 去讀取這個 socket 時，recv() 則會回傳 0 給你。這樣你就能知道是 client 關閉連線了。

再次強調 select() 有趣的地方：如果你正在 listen() 一個 socket，你可以將這個 socket 的 file descriptor 放在 readfds set 中，用來檢查是不是有新的連線。

朋友阿，這就是萬能 select() 函式的速成說明。

不過，應觀眾要求，這裡提供個有深度的範例，毫無疑問地，以前的簡單範例和這個範例的難易度會有顯著差距。不過你可以先看看，然後讀後面的解釋。

程式 [26] 的行爲是簡單的多使用者聊天室 server，在一個視窗中執行 server，然後在其它多個視窗使用 telnet 連線到 server（telnet hostname 9034）。當你在其中一個 telnet session 中輸入某些文字時，這些文字應該會在其它每個視窗上出現。

```
/*
** selectserver.c -- 一個 cheezy 的多人聊天室 server
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define PORT "9034" // 我們正在 listen 的 port

// 取得 sockaddr，IPv4 或 IPv6：
void *get_in_addr(struct sockaddr *sa)
{
  if (sa->sa_family == AF_INET) {
    return &(((struct sockaddr_in*)sa)->sin_addr);
  }

  return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
  fd_set master; // master file descriptor 清單
  fd_set read_fds; // 給 select() 用的暫時 file descriptor 清單
  int fdmax; // 最大的 file descriptor 數目

  int listener; // listening socket descriptor
  int newfd; // 新接受的 accept() socket descriptor
  struct sockaddr_storage remoteaddr; // client address
  socklen_t addrlen;

  char buf[256]; // 儲存 client 資料的緩衝區
  int nbytes;
```

```
char remoteIP[INET6_ADDRSTRLEN];

int yes=1; // 供底下的 setsockopt() 設定 SO_REUSEADDR
int i, j, rv;

struct addrinfo hints, *ai, *p;

FD_ZERO(&master); // 清除 master 與 temp sets
FD_ZERO(&read_fds);

// 給我們一個 socket，並且 bind 它
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
  fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
  exit(1);
}

for(p = ai; p != NULL; p = p->ai_next) {
  listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
  if (listener < 0) {
    continue;
  }

  // 避開這個錯誤訊息："address already in use"
  setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

  if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
    close(listener);
    continue;
  }

  break;
}

// 若我們進入這個判斷式，則表示我們 bind() 失敗
if (p == NULL) {
  fprintf(stderr, "selectserver: failed to bind\n");
  exit(2);
}
freeaddrinfo(ai); // all done with this

// listen
if (listen(listener, 10) == -1) {
  perror("listen");
  exit(3);
}

// 將 listener 新增到 master set
```

```
    FD_SET(listener, &master);

    // 持續追蹤最大的 file descriptor
    fdmax = listener; // 到此爲止，就是它了

    // 主要迴圈
    for( ; ; ) {
      read_fds = master; // 複製 master

      if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
        perror("select");
        exit(4);
      }

      // 在現存的連線中尋找需要讀取的資料
      for(i = 0; i <= fdmax; i++) {
        if (FD_ISSET(i, &read_fds)) { // 我們找到一個！！
          if (i == listener) {
            // handle new connections
            addrlen = sizeof remoteaddr;
            newfd = accept(listener,
              (struct sockaddr *)&remoteaddr,
              &addrlen);

            if (newfd == -1) {
              perror("accept");
            } else {
              FD_SET(newfd, &master); // 新增到 master set
              if (newfd > fdmax) { // 持續追蹤最大的 fd
                fdmax = newfd;
              }
              printf("selectserver: new connection from %s on "
                "socket %d\n",
                inet_ntop(remoteaddr.ss_family,
                  get_in_addr((struct sockaddr*)&remoteaddr),
                  remoteIP, INET6_ADDRSTRLEN),
                newfd);
            }

          } else {
            // 處理來自 client 的資料
            if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
              // got error or connection closed by client
              if (nbytes == 0) {
                // 關閉連線
                printf("selectserver: socket %d hung up\n", i);
              } else {
                perror("recv");
              }
              close(i); // bye!
              FD_CLR(i, &master); // 從 master set 中移除

            } else {
```

```
            // 我們從 client 收到一些資料
            for(j = 0; j <= fdmax; j++) {
              // 送給大家！
              if (FD_ISSET(j, &master)) {
                // 不用送給 listener 跟我們自己
                if (j != listener && j != i) {
                  if (send(j, buf, nbytes, 0) == -1) {
                    perror("send");
                  }
                }
              }
            }
          }
        } // END handle data from client
      } // END got new incoming connection
    } // END looping through file descriptors
  } // END for( ; ; )--and you thought it would never end!

  return 0;
}
```

我說過在程式碼中有兩個 file descriptor set：master 與 read_fds。前面的 master 記錄全部現有連線的 socket descriptor，與正在 listen 新連線的 socket descriptor 一樣。

我用 master 的理由是因爲 select() 實際上會改變你傳送過去的 set，用來反映目前就緒可讀（ready for read）的 socket。因爲我必須在在兩次的 select() calls 期間也能夠持續追蹤連線，所以我必須將這些資料安全地儲存在某個地方。最後，我再將 master 複製到 read_fds，並接著呼叫 select()。

可是這不就代表每當有新連線時，我就要將它新增到 master set 嗎？是的！

而每次連線結束時，我們也要將它從 master set 中移除嗎？是的，沒有錯。

我說過，我們要檢查 listen 的 socket 是否就緒可讀，如果可讀，這代表我有一個待處理的連線，而且我要 accept() 這個連線，並將它新增到 master set。同樣地，當 client 連線就緒可讀且 recv() 傳回 0 時，我們就能知道 client 關閉了連線，而我必須將這個 socket descriptor 從 master set 中移除。

若 client 的 recv() 傳回非零的值，因而，我能知道 client 已經收到了一些資料，所以我收下這些資料，並接著到 master 清單，並將資料送給其它已連線的每個 clients。

我的朋友們，以上是萬能 select() 函式的概述，這眞是一件不簡單的事情。

另外，這有個福利：一個名爲 poll 的函式，它的行爲與 select() 很像，但是在管理 file descriptor set 時是用不一樣的系統，你可以看看 poll()。

參考資料

[24] http://www.monkey.org/~provos/libevent/

[25] http://beej.us/guide/bgnet/examples/select.c

[26] http://beej.us/guide/bgnet/examples/selectserver.c

譯者註：在 The Linux Programming Interface 的第 63 章有更深入的說明。

不完整傳送的後續處理

還記得前面的 send() 章節嗎？當時我不是提過 send() 可能不會將你所要求的資料全部送出嗎？也就是說，雖然你想要送出 512 bytes，但是 send() 只送出 412 bytes。那剩下的 100 個 bytes 到哪去了呢？

好的，其實它們還在你的緩衝區裡。因爲環境不是你能控制的，kernel 會決定要不要用一個 chunk 將全部的資料送出，而現在，我的朋友，你可以決定要如何處理緩衝區中剩下的資料。

你可以寫一個像這樣的函式：

```
#include <sys/types.h>
#include <sys/socket.h>

int sendall(int s, char *buf, int *len)
{
  int total = 0; // 我們已經送出多少 bytes 的資料
  int bytesleft = *len; // 我們還有多少資料要送
  int n;

  while(total < *len) {
    n = send(s, buf+total, bytesleft, 0);
    if (n == -1) { break; }
    total += n;
    bytesleft -= n;
  }

  *len = total; // 傳回實際上送出的資料量

  return n==-1?-1:0; // 失敗時傳回 -1、成功時傳回 0
}
```

在這個例子裡，s 是你想要傳送資料的 socket，buf 是儲存資料的緩衝區，而 len 是一個指標，指向一個 int 型別的變數，記錄了緩衝區中的資料數量。

函式在錯誤時傳回 -1（而 errno 仍然從呼叫 send() 設定）。還有，實際送出的資料數量會在 len 中回傳，除非有錯誤發生，不然這會跟你所要求要傳送的資料量相同。sendall() 會盡力將資料送出，不過如果有錯誤發生時，它就會立刻回傳給你。

爲了完整性，這邊有一個呼叫函式的範例：

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
  perror("sendall");
  printf("We only sent %d bytes because of the error!\n", len);
}
```

當封包的一部分抵達接收端（receiver end）時會發生什麼事情呢？如果封包的長度是會變動的（variable），接收端要如何知道另一端的封包何時開始與結束呢？

是的，你或許必須封裝（encapsulate）「還記得資料封裝（data encapsulation）這節的開頭那邊嗎？那邊有詳細說明」。

Serialization：如何封裝資料

你已經知道要將文字資料透過網路傳送很簡單，不過如果你想要送一些「二進制」的資料，如 int 或 float，會發生什麼事情呢？這裡有一些選擇。

1. 將數字轉換爲文字，使用如 sprintf() 的函式，接著傳送文字。接收者會使用如 strtol() 函式解析文字，並轉換爲數字。
2. 直接以原始資料傳送，將指向資料的指標傳遞給 send()。
3. 將數字編碼（encode）爲可移植的二進制格式，接收者會將它解碼（decode）。

先睹爲快！只在今晚！

〔序幕〕Beej 說：”我偏好上面的第三個方法！”〔結束〕

（在我開始熱血介紹本章節之前，我應該要跟你說有現成的函式庫可以做這件事情，而要自製個可移植及無錯誤的作品會是相當大的挑戰。所以在決定要自己實作這部分時，可以先四處看看，並做完你的家庭作業。我在這裡引用些類似這個作品的有趣的資訊。）

實際上，上面全部的方法都有它們的缺點與優點，但是如我所述，通常我偏好第三個方法。首先，咱們先談談另外兩個的優缺點。

第一個方法，在傳送以前先將數字編碼爲文字，優點是你可以很容易印出及讀取來自網路的資料。有時，人類易讀的協定比較適用於頻寬不敏感（non-bandwidth-intensive）的情況，例如：Internet Relay Chat（IRC）[27]。然而，缺點是轉換耗時，且總是需要比原本的數字使用更多的空間。

第二個方法：傳送原始資料（raw data），這個方法相當簡單〔但是危險！〕：只要將資料指標提供給 send()。

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0); /* 危險，不具可移植性！ */
```

接收者類似這樣接收：

```
double d;

recv(s, &d, sizeof d, 0); /* 危險，不具可移植性！ */
```

快速又簡單，那有什麼不好的呢？

好的，事實證明不是全部的架構都能表示 double（或 int）。（嘿！或許你不需要可移植性，在這樣的情況下這個方法很好，而且快速。）

當封裝整數型別時，我們已經知道 htons() 這類的函式如何透過將數字轉換爲 Network Byte Order（網路位元組順序），來讓東西可以移植。可惜的是，沒有類似的函式可以供 float 型別使用。

全部的希望都落空了嗎？

別怕！（你有擔心了一會兒嗎？沒有嗎？一點都沒有嗎？）

我們可以做件事情：我們可以將資料封裝爲接收者已知的二進位格式，讓接收著可以在遠端解壓縮。

我所謂的「已知二進位格式」是什麼意思呢？

好的，我們已經看過了 htons() 範例了，不是嗎？它將數字從 host 格式改變（或是”編碼”）爲 Network Byte Order 格式；如果要反轉「解碼」這個數字，接收端會呼叫 ntohs()。

可是我不是才剛說過，沒有這樣的函式可供非整數型別使用嗎？

是的，我說過。而且因爲 C 語言並沒有規範標準的方式來做，所以這有點麻煩〔that a gratuitous pun there for you Python fans〕。

要做的事情是將資料封裝到已知的格式，並透過網路送出。例如：封裝 float，這裡的東西有很大的改善空間：[28]

```
#include <stdint.h>

uint32_t htonf(float f)
{
  uint32_t p;
  uint32_t sign;

  if (f < 0) { sign = 1; f = -f; }
  else { sign = 0; }

  p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31); // whole part and sign
  p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // fraction

  return p;
}

float ntohf(uint32_t p)
{
  float f = ((p>>16)&0x7fff); // whole part
  f += (p&0xffff) / 65536.0f; // fraction

  if (((p>>31)&0x1) == 0x1) { f = -f; } // sign bit set

  return f;
}
```

上列的程式碼是一個 native（原生的）實作，將 float 儲存爲 32-bit 的數字。High bit（高位元）（31）用來儲存數字的正負號（'1' 表示負數），而接下來的七個位元（30-16）是用來儲存 float 整個數字的部分。最後，剩下的位元（15-0）用來儲存數字的小數（fractional portion）部分。

使用方式相當直覺：

```
#include <stdio.h>

int main(void)
{
  float f = 3.1415926, f2;
  uint32_t netf;

  netf = htonf(f); // 轉換爲 "network" 形式
  f2 = ntohf(netf); // 轉回測試

  printf("Original: %f\n", f); // 3.141593
  printf(" Network: 0x%08X\n", netf); // 0x0003243F
  printf("Unpacked: %f\n", f2); // 3.141586

  return 0;
}
```

好處是：它很小、很簡單且快速，缺點是：它在空間的使用沒有效率，而且對範圍有嚴格的限制－試著在那邊儲存一個大於 32767 的數，它就會不爽！

你也可以在上面的例子看到，最後一對的十進位空間並沒有正確保存。

我們該怎麼改呢？

好的，用來儲存浮點數（float point number）的標準方式是已知的 IEEE-754 [29]。多數的電腦會在內部使用這個格式做浮點運算，所以在這些例子裡，嚴格說來，不需要做轉換。但是如果你想要你的程式碼具可移植性，就要假設你不需要轉換。（換句話說，如果你想要讓程式很快，你應該要在不需要做轉換的平台上進行最佳化！這就是 htons() 與它的家族使用的方法。）

這邊有段程式碼可以將 float 與 double 編碼爲 IEEE-754 格式 [30]。（主要的功能，它不會編碼 NaN 或 Infinity，只要作點修改就可以了。）

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

uint64_t pack754(long double f, unsigned bits, unsigned expbits)
{
  long double fnorm;
  int shift;
  long long sign, exp, significand;
  unsigned significandbits = bits - expbits - 1; // -1 for sign bit

  if (f == 0.0) return 0; // get this special case out of the way

  // 檢查正負號並開始正規化
  if (f < 0) { sign = 1; fnorm = -f; }
  else { sign = 0; fnorm = f; }

  // 取得 f 的正規化型式並追蹤指數
  shift = 0;
  while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
  while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
  fnorm = fnorm - 1.0;

  // 計算有效位數資料的二進位格式（非浮點數）
  significand = fnorm * ((1LL<<significandbits) + 0.5f);

  // get the biased exponent
  exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

  // 傳回最後的解答
  return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) | significand;
}
```

```
long double unpack754(uint64_t i, unsigned bits, unsigned expbits)
{
  long double result;
  long long shift;
  unsigned bias;
  unsigned significandbits = bits - expbits - 1; // -1 for sign bit

  if (i == 0) return 0.0;

  // pull the significand

  result = (i&((1LL<<significandbits)-1)); // mask
  result /= (1LL<<significandbits); // convert back to float
  result += 1.0f; // add the one back on

  // deal with the exponent
  bias = (1<<(expbits-1)) - 1;
  shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
  while(shift > 0) { result *= 2.0; shift--; }
  while(shift < 0) { result /= 2.0; shift++; }

  // sign it
  result *= (i>>(bits-1))&1? -1.0: 1.0;

  return result;
}
```

我在那裡的頂端放一些方便的 macro（巨集），用來封裝與解封裝 32-bit（可能是 float）與 64-bit（可能是 double）的數字，但是 pack754() 函式可以直接呼叫，並告知編碼幾個位元的資料（expbits 的哪幾個位元要保留給正規化數值的指數。）

這裡是使用範例：

```
#include <stdio.h>
#include <stdint.h> // 定義 uintN_t 型別
#include <inttypes.h> // 定義 PRIx macros

int main(void)
{
  float f = 3.1415926, f2;
  double d = 3.14159265358979323, d2;
  uint32_t fi;
  uint64_t di;

  fi = pack754_32(f);
  f2 = unpack754_32(fi);

  di = pack754_64(d);
  d2 = unpack754_64(di);

  printf("float before : %.7f\n", f);
  printf("float encoded: 0x%08" PRIx32 "\n", fi);
  printf("float after : %.7f\n\n", f2);

  printf("double before : %.20lf\n", d);
  printf("double encoded: 0x%016" PRIx64 "\n", di);
  printf("double after : %.20lf\n", d2);

  return 0;
}
```

上面的程式碼會產生下列的輸出：

```
float before : 3.1415925
float encoded: 0x40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double after  : 3.14159265358979311600
```

你可能遭遇的另一個問題是你該如何封裝 struct 呢？

對你來說沒有問題的，編譯器會自動將一個 struct 中的全部空間填入。〔你不會病到聽成”不能這樣做”、”不能那樣做”？抱歉！引述一個朋友的話：”當事情出錯了，我都會怪給 Microsoft。”這次固然可能不是 Microsoft 的錯，不過我朋友的陳述完全符合事實。〕

回到這邊，透過網路送出 struct 的最好方式是將每個欄位獨立封裝，並接著在它們抵達另一端時，將它們解封裝到 struct。

你正在想，這樣要做很多事情。

是的，的確是。你能做的一件事情是寫個好用的函式來幫你封裝資料，這很好玩！眞的！

在 Kernighan 與 Pike 著作的 "The Practice of Programming" [31] 這本書，他們實作類似 printf() 的函式，名爲 pack() 與 unpack()，可以完全做到這件事。我想要連結到這些函式，但是這些函式顯然地無法從網路上取得。

（The Practice of Programming 是值得閱讀的好書，Zeus saves a kitten every time I recommend it。）

此時，我正打算捨棄一個指標（pointer），它指向我從未用過的 BSD 授權類型參數語言 C API（BSD-licensed Typed Parameter Language C API）[32]，可是這看起來整個很可敬。Python 與 Perl 程式設計師想找出他們語言裡的 pack() 與 unpack() 函式，用來完成同樣的事情。而 Java 有一個能用於相同用途的 big-ol' Serializable interface。

不過，如果你想要用 C 寫自己的封裝工具，K&P 的技巧是使用變動參數列（variable argument list），用類似 printf() 的函式建立封包。我自己編寫的版本 [33] 希望能足以幫助你瞭解這樣的東西是如何運作的。

「這段程式碼參考到上面的 pack754() 函式，packi*() 函式的運作方式類似 htons() 家族，除非它們是封裝到一個 char 陣列（array）而不是另一個整數。」

```
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>

// 供浮點數型別的變動位元
// 隨著架構而變動

typedef float float32_t;
typedef double float64_t;

/*
** packi16() -- store a 16-bit int into a char buffer (like htons())
*/
void packi16(unsigned char *buf, unsigned int i)
{
  *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() -- store a 32-bit int into a char buffer (like htonl())
*/
void packi32(unsigned char *buf, unsigned long i)
{
  *buf++ = i>>24; *buf++ = i>>16;
  *buf++ = i>>8; *buf++ = i;
}
```

```
/*
** unpacki16() -- unpack a 16-bit int from a char buffer (like ntohs())
*/
unsigned int unpacki16(unsigned char *buf)
{
  return (buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- unpack a 32-bit int from a char buffer (like ntohl())
*/
unsigned long unpacki32(unsigned char *buf)
{
  return (buf[0]<<24) | (buf[1]<<16) | (buf[2]<<8) | buf[3];
}

/*
** pack() -- store data dictated by the format string in the buffer
**
** h - 16-bit l - 32-bit
** c - 8-bit char f - float, 32-bit
** s - string (16-bit length is automatically prepended)
*/
int32_t pack(unsigned char *buf, char *format, ...)
{
  va_list ap;
  int16_t h;
  int32_t l;
  int8_t c;
  float32_t f;
  char *s;
  int32_t size = 0, len;

  va_start(ap, format);

  for(; *format != '\0'; format++) {
    switch(*format) {
    case 'h': // 16-bit
      size += 2;
      h = (int16_t)va_arg(ap, int); // promoted
      packi16(buf, h);
      buf += 2;
      break;

    case 'l': // 32-bit
      size += 4;
      l = va_arg(ap, int32_t);
      packi32(buf, l);
      buf += 4;
      break;

    case 'c': // 8-bit
```

```
      size += 1;
      c = (int8_t)va_arg(ap, int); // promoted
      *buf++ = (c>>0)&0xff;
      break;

    case 'f': // float
      size += 4;
      f = (float32_t)va_arg(ap, double); // promoted
      l = pack754_32(f); // convert to IEEE 754
      packi32(buf, l);
      buf += 4;
      break;

    case 's': // string
      s = va_arg(ap, char*);
      len = strlen(s);
      size += len + 2;
      packi16(buf, len);
      buf += 2;
      memcpy(buf, s, len);
      buf += len;
      break;
    }
  }

  va_end(ap);

  return size;
}
/*
** unpack() -- unpack data dictated by the format string into the buffer
*/
void unpack(unsigned char *buf, char *format, ...)
{
  va_list ap;
  int16_t *h;
  int32_t *l;
  int32_t pf;
  int8_t *c;
  float32_t *f;
  char *s;
  int32_t len, count, maxstrlen=0;

  va_start(ap, format);

  for(; *format != '\0'; format++) {
    switch(*format) {
    case 'h': // 16-bit
      h = va_arg(ap, int16_t*);
      *h = unpacki16(buf);
      buf += 2;
      break;
```

```
    case 'l': // 32-bit
      l = va_arg(ap, int32_t*);
      *l = unpacki32(buf);
      buf += 4;
      break;

    case 'c': // 8-bit
      c = va_arg(ap, int8_t*);
      *c = *buf++;
      break;

    case 'f': // float
      f = va_arg(ap, float32_t*);
      pf = unpacki32(buf);
      buf += 4;
      *f = unpack754_32(pf);
      break;

    case 's': // string
      s = va_arg(ap, char*);
      len = unpacki16(buf);
      buf += 2;
      if (maxstrlen > 0 && len > maxstrlen) count = maxstrlen - 1;
      else count = len;
      memcpy(s, buf, count);
      s[count] = '\0';
      buf += len;
      break;

    default:
      if (isdigit(*format)) { // track max str len
        maxstrlen = maxstrlen * 10 + (*format-'0');
      }
    }

    if (!isdigit(*format)) maxstrlen = 0;
  }

  va_end(ap);
}
```

不管你是自己寫的程式，或者用別人的程式碼，基於持續檢查 bugs 的理由，有組通用的資料封裝機制集合是個好主意，而且不用每次都手動封裝每個 bit（位元）。

封裝資料時，使用哪種格式會比較好呢？

好問題，很幸運地，RFC 4506 [35]，the External Data Representation Standard 已經定義了一堆各類型的二進位格式，如：浮點數型別、整數型別、陣列、原始資料等。如果你打算自己寫程式來封裝資料，我建議要符合標準，雖然不會強制你一定要遵守規範，但是封包規則不會剛好是你家定義的，至少，我不認爲。

無論如何，在你送出資料以前，用某種方法將資料編碼是正確的做事方法。

[27] http://en.wikipedia.org/wiki/Internet_Relay_Chat

[28] http://beej.us/guide/bgnet/examples/pack.c

[29] http://en.wikipedia.org/wiki/IEEE_754

[30] http://beej.us/guide/bgnet/examples/ieee754.c

[31] http://cm.bell-labs.com/cm/cs/tpop/

[32] http://tpl.sourceforge.net/

[33] http://beej.us/guide/bgnet/examples/pack2.c

[34] http://beej.us/guide/bgnet/examples/pack2.c

[35] http://tools.ietf.org/html/rfc4506

資料封裝

不管怎樣，你的意思眞的是指封裝資料嗎？

以最簡單的例子而言，這表示你會需要增加一個 header（標頭），用來代表識別的資訊或封包長度，或者都有。

你的 header 看起來像什麼呢？

好的，它就只是某個用來表示你覺得完成專案會需要的二進位資料。

哇，好抽象。

Okay，舉例來說，咱們說你有一個使用 SOCK_STREAM 的多重使用者聊天程式。當某個使用者輸入〔"says"〕某些字，會有兩筆資訊要傳送給 server：

"是誰" 及 "說什麼"。

到目前爲止都還可以嗎？

你問：”會有什麼問題嗎？”

問題是訊息的長度是會變動的。一個叫做 "tom" 的人可能會說 ”Hi（嗨）” ，而另一個叫做 ”Benjamin（班傑明）” 的人可能說：”Hey guys what is up？（嘿！兄弟最近你好嗎？）”

所以你在收到全部的資料之後，將它全部 send() 給 clients。你輸出的 data stream（資料串流）類似這樣：

```
tomHiBenjaminHeyguyswhatisup?
```

類似這樣。那 client 要如何知道訊息何時開始與結束呢？

如果你願意，是可以的，只要讓全部的訊息都一樣長，並只要呼叫我們之前實作的 sendall() 就行了。但是這樣會浪費頻寬（bandwidth）！我們並不想用 send() 送出了 1024 個 bytes 的資料，卻只有攜帶了 "tom" 說了 "Hi" 這樣的有效資訊。

所以我們以小巧的 header 與封包結構封裝（encapsulate）資料。Client 與 server 都知道如何封裝（pack）與解封裝（unpack）這筆資料〔有時候稱爲 ”marshal” 與 ”unmarshal” 〕。現在先不要想太多，我們會開始定義一個協定（protocol），用來描述 client 與 server 是如何溝通的！

在這個例子中，咱們假設使用者的名稱是固定 8 個字元，並用 '\0' 結尾。然後接著讓我們假設資料的長度是變動的，最多高達 128 個字元。我們看個可能在這個情況會用到的封包結構範例。

1. len〔1 個 byte，unsigned（無號）〕：封包的總長度，計算 8 個 bytes 的使用者名稱，以及聊天資料。
2. name〔8 個 bytes〕：使用者名稱，如果有需要，結尾補上 NUL。
3. chatdata〔n 個 bytes〕：資料本身，最多 128 bytes。封包的長度應該要以這個資料長度加 8〔上面的 name 欄位長度〕來計算。

爲什麼我選擇 8 個 bytes 與 128 個 bytes 長度的欄位呢？我假設這樣就已經夠用了，或許，8 個 bytes 對你的需求而言太少了，你也可以有 30 個 bytes 的 name 欄位，總之，你可以自己決定！

使用上列的封包定義，第一個封包由下列的資訊組成〔以 hex 與 ASCII〕：

```
   0A    74 6F 6D 00 00 00 00 00     48 69
(length) T  o  m    (padding)       H  i
```

而第二個也是差不多：

```
   18    42 65 6E 6A 61 6D 69 6E       48 65 79 20 67 75 79 73 20 77 ...
(length) B  e  n  j  a  m  i  n        H  e  y     g  u  y  s     w  ...
```

〔長度（length）是以 Network Byte Order 儲存，當然，在這個例子只有一個 byte，所以沒差，但是一般而言，你會想要讓你全部的二進位整數能以Network Byte Order 儲存在你的封包中。〕

當你傳送資料時，你應該要謹慎點，使用類似前面的 sendall() 指令，因而你可以知道全部的資料都有送出，即便要將資料全部送出會多花幾次的 send()。

同樣地，當你接收這筆資料時，你需要額外做些處理。如果要保險一點，你應該假設你可能只會收到部分的封包內容〔如我們可能會從上面的班傑明那裡收到 ”18 42 65 6E 6A”〕，但是我們這次呼叫 recv() 全部就只收到這些資料。我們需要一次又一次的呼叫 recv()，直到完整地收到封包內容。

可是要怎麼做呢？

好的，我們可以知道所要接收的封包它全部的 byte 數量，因爲這個數量會記載在封包前面。我們也知道最大的封包大小是 1 + 8 + 128，或者 137 bytes〔因爲這是我們自己定義的〕。

實際上你在這邊可以做兩件事情，因爲你知道每個封包是以長度（length）做開頭，所以你可以呼叫 recv() 只取得封包長度。接著，你知道長度以後，你就可以再次呼叫 recv()，這時候你就可以正確地指定剩下的封包長度〔或者重複取得全部的資料〕，直到你收到完整的封包內容爲止。這個方法的優點是你只需有一個足以存放一個封包的緩衝區，而缺點是你爲了要接收全部的資料，至少呼叫兩次的 recv()。

另一個方法是直接呼叫 recv()，並且指定你所要接收的封包之最大資料量。這樣的話，無論你收到多少，都將它寫入緩衝區，並最後檢查封包是否完整。當然，你可能會收到下一個封包的內容，所以你需要有足夠的空間。

你所能做的是宣告（declare）一個足以容納兩個封包的陣列，這是你在封包到達時，你可以重新建構（reconstruct）封包的地方。

每次你用 recv() 接收資料時，你會將資料接在工作緩衝區（work buffer）的後端，並檢查封包是否完整。在緩衝區中的資料數量大於或等於 封包 header 中所指定的長度時〔+1，因爲 header 中的長度沒有包含 length 本身的長度〕。若緩衝區中的資料長度小於 1，那麼很明顯地，封包是不完整的。你必須針對這種情況做個特別處理，因爲第一個 byte 是垃圾，而你不能用它來取得正確的封包長度。

一旦封包已經完整接收了，你就可以做你該做的處理，將資料拿來使用，並在用完之後將它從工作緩衝區中移除。

呼呼！Are you juggling that in your head yet？

好的，這裡是第二次的衝擊：你可能在一次的 recv() call 就已經讀到了一個封包的結尾，還讀到下一個封包的內容，即是你的工作緩衝區有一個完整的封包，以及下一個封包的一部分！該死的傢伙。〔但是這就是爲什麼你需要讓你的工作緩衝區可以容納兩個封包的原因，就是會發生這種情況！〕

因爲你從 header 得知第一個封包的長度，而你也有持續追蹤工作緩衝區的資料量，所以你可以相減，並且計算出工作緩衝區中有多少資料是屬於第二個〔不完整的〕封包的。當你處理完第一個封包後，你可以將第一個封包的資料從工作緩衝區中清掉，並將第二個封包的部分內容移到緩衝區的前面，準備進行下一次的 recv()。

〔部分讀者會注意到，實際地將第二個封包的部份資料移動到緩衝區的開頭需要花費時間，而程式可以寫成利用環狀緩衝區（circular buffer），就不需要這樣做。如果你還是很好奇，可以找一本資料結構的書來讀。〕

我從未說過這很簡單，好吧，我有說過這很簡單。而你所需要的只是多練習，然後很快的你就會習慣了。我發誓！

[34] http://beej.us/guide/bgnet/examples/pack2.c

[35] http://tools.ietf.org/html/rfc4506

廣播封包：Hello World！

到了這裡，本文已經談了如何將資料從一台主機傳送到另一台主機。但是，我堅持你可能會需要絕對的權力，同時將資料送給多個主機！

用 UDP（只能用 UDP，TCP 不行）與標準 IPv4，可以透過一種叫作廣播（broadcasting）的機制達成。IPv6 不支援廣播，所以你必須要採用比較高級的技術－群播（multicasting），很遺憾地，我現在不會討論這個，我受夠了異想天開的未來，我們現在還停留在 32-bit 的 IPv4 世界呢！

可是，請等一下！不管你願不願意，你別離開呀，開始說說廣播吧。

你必須在將廣播封包送到網路之前，先設定 SO_BROADCAST socket 選項。這類似一個推送導彈開關的小塑膠蓋！就只是你的手上掌握了多少的權力。

不過認眞說來，使用廣播封包是很危險的，因爲每個收到廣播封包的系統都要撥開一層層的資料封裝，直到系統知道這筆資料是要送給哪個 port 爲止。然後系統會開始處理這筆資料或者丟掉它。在另一種情況，對每部收到廣播封包的機器而言這很費工，因爲他們都在同一個區域網路（local network），這樣會讓很多電腦做不少多餘的工作。當 Doom 遊戲出現時，就有人在說它的網路程式寫的不好。

現在，有很多方法可以解決這個問題 ...

等一下，眞的有很多方法嗎？

那是什麼表情阿？哎呀，一樣阿，送廣播封包的方法很多。所以重點就是：你該如何指定廣播訊息的目地位址呢？

有兩種常見的方法：

1. 將資料送給子網路（subnet）的廣播位址，就是將 subnet's network（子網路網段）的 host（主機）那部分全部填 1，舉例來說，我家裡的網路是 192.168.1.0，而我的 netmask（網路遮罩）是 255.255.255.0，所以位址的最後一個 byte 就是我的 host number（因爲依據 netmask，前三個 bytes 是 network number）。所以我的廣播位址就是 192.168.1.255。在 Unix 底下，ifconfig指令實際上都會給你這些資料。（如果你有興趣，取得你廣播位址的邏輯運算方式是 network_number OR (Not netmask)）。你可以用跟區域網路一樣的方式，將這類型的廣播封包送到遠端網路（remote network），不過風險是封包可能會被目地端的 router（路由器）丟棄。（如果 router 沒有將封包丟棄，那麼有個隨機的藍色小精靈會開始用廣播流量對它們的區域網路造成水災。）
2. 將資料送給”global（全域的）”廣播位址，255.255.255.255，又稱爲 INADDR_BROADCAST，很多機器會自動將它與你的 network number 進行 AND 位元運算，以轉換爲網路廣播位址，但是有些機器不會這樣做。Routers 不會將這類的廣播封

包轉送（forward）出你的區域網路，夠諷刺的。

所以如果你想要將資料送到廣播位址，但是沒有設定 SO_BROADCAST socket 選項時會怎樣呢？好，我們用之前的 talker 與listener 來炒冷飯，然後看看會發生什麼事情。

```
$ talker 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied
$ talker 255.255.255.255 foo
sendto: Permission denied
```

是的，沒有很順利 ... 因爲我們沒有設定 SO_BROADCAST socket 選項，設定它，然後現在你就可以用 sendto() 將資料送到你想送的地方了！

事實上，這就是 UDP 應用程式能不能廣播的差異點。所以我們改一下舊的 talker 應用程式，設定 SO_BROADCAST socket 選項。這樣我們就能呼叫 broadcaster.c 程式了 [36]：

[36] http://beej.us/guide/bgnet/examples/broadcaster.c

```
/*
** broadcaster.c -- 一個類似 talker.c 的 datagram "client"，
** 差異在於這個可以廣播
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950 // 所要連線的 port

int main(int argc, char *argv[])
{
  int sockfd;
  struct sockaddr_in their_addr; // 連線者的位址資訊
  struct hostent *he;
  int numbytes;
  int broadcast = 1;
  //char broadcast = '1'; // 如果上面這行不能用的話，改用這行

  if (argc != 3) {
    fprintf(stderr,"usage: broadcaster hostname message\n");
    exit(1);
  }
```

```
  if ((he=gethostbyname(argv[1])) == NULL) { // 取得 host 資訊
    perror("gethostbyname");
    exit(1);
  }

  if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  // 這個 call 就是要讓 sockfd 可以送廣播封包
  if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
    sizeof broadcast) == -1) {
    perror("setsockopt (SO_BROADCAST)");
    exit(1);
  }

  their_addr.sin_family = AF_INET; // host byte order
  their_addr.sin_port = htons(SERVERPORT); // short, network byte order
  their_addr.sin_addr = *((struct in_addr *)he->h_addr);
  memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

  if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
          (struct sockaddr *)&their_addr, sizeof their_addr)) == -1) {
    perror("sendto");
    exit(1);
    }

  printf("sent %d bytes to %s\n", numbytes,
      inet_ntoa(their_addr.sin_addr));

  close(sockfd);

  return 0;
}
```

這個跟”一般的” UDP client/server 有什麼不同呢？

沒有！〔除了 client 可以送出廣播封包〕

同樣地，我們繼續，並在其中一個視窗執行舊版的 UDP listener 程式，然後在另一個視窗執行 broadcaster，你應該可以順利執行了。

```
$ broadcaster 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ broadcaster 192.168.1.255 foo
sent 3 bytes to 192.168.1.255
$ broadcaster 255.255.255.255 foo
sent 3 bytes to 255.255.255.255
```

而你應該會看到 listener 回應說它已經收到封包。（如果 listener 沒有回應，可能是因爲它綁到 IPv6 位址了，試著將 listener.c 中的 AF_UNSPEC 改成 AF_INET，強制使用 IPv4）。

好，眞令人興奮，可是現在要在同一個網路上的另一台電腦執行 listener，所以你會有兩個複本正在執行，每個機器上各有一個，然後再次用你的廣播位址來執行 broadcaster ... 嘿！你只有呼叫一次 sendto()，但是兩個 listeners 都收到了你的封包。酷喔！

如果 listener 收到你直接送給它的資料，但不是在廣播位址沒有資料，可能是因爲你本機（local machine）上有防火牆（firewall）封鎖了這些封包。（是的，謝謝 Pat 與 Bapper 的說明，讓我知道爲什麼我的範例程式無法運作。我跟你們說過我會在文件中提到你們，就是這裡了，感恩。）

再次提醒，使用廣播封包一定要小心，因爲 LAN 上面的每台電腦都會被迫處理這類封包，無論它們有沒有用 recvfrom() 接收，這類封包會造成整個電腦網路相當大的負擔，所以一定要謹慎、適當地使用廣播。

常見的問題

我可以從哪邊取得那些 **header** 檔案呢**?**

如果你的系統還沒有這些檔案，你可能就不需要它們。檢查你平台的使用手冊。若你在 Windows 上開發，那麼你只需要 `#include <winsock.h>` 。

在 **bind()** 回報”**Address already in use**”（位址已經在使用中）時，我該怎麼辦呢？

你必須使用 setsockopt() 對 listen 的 socket 設定 SO_REUSEADDR 選項。請參考 bind() 及 select() 章節的範例。

我該如何取得系統上已經開啓的 **sockets** 清單呢？

使用 netstat。細節請參考 man 使用手冊，不過你應該只要輸入下列的指令就能取得一些不錯的資訊：

```
$ netstat
```

我該如何檢視 **routing table**（路由表）呢？

執行 route 指令（多數的 Linux 系統是在 /sbin 底下），或者 netstat -r 指令。

如果我只有一台電腦，我該如何執行 **client**（客户端）與 **server**（伺服器）程式呢？我需要網路來寫網路程式嗎？

你很幸運，全部的系統都有實作一個 loopback（繞迴）虛擬網路”裝置”，這個裝置位於 kernel 中，並假裝是張網路卡〔這個介面就是 routing table 中所列出的 ”lo”〕。

假裝你已經登入一個名爲”goat”的系統，在一個視窗中執行 client，並在另一個視窗執行 server。

或者可以在背景啓動 server（server &），並在同樣的視窗執行 client。

loopback 裝置的功能是你可以執行 client goat 或 client localhost（因爲 localhost 應該已經定義在你的 /etc/hosts 檔案），而你可以讓 client 與 server 溝通而不需要網路。

簡而言之，不需要改變任何的程式碼，就可以讓程式在無網路的單機系統上執行！好耶！

我該怎麼知道對方已經關閉連線呢？

你可以辨別出來，因爲 recv() 會傳回 0。

我該如何實作一個”**ping**”工具呢？什麼是 **ICMP** 呢？我可以在哪裡找到更多關於 **raw socket** 與 **SOCK_RAW** 的資料呢？

你對 raw socket 的全部疑問都可以在 W. Richard Stevens 的 UNIX Network Programming 書本上找到答案。還有，研究 Stevens 的 UNIX Network Programming 程式碼的 ping 子目錄，可以從線上下載 [37]。

我該如何改變或縮短呼叫 **connect()** 的逾期時間呢？

我不想跟你說一樣的答案：「W. Richard Stevens 會告訴你」，我只能建議你參考 UNIX Network Programming 原始程式碼 [38] 中的 /lib/connect_nonb.c。

主要是你要用 socket() 建立一個 socket descriptor，將它設定爲 non-blocking（非阻塞式），呼叫 connect()，而如果一切順利，connect() 會立刻傳回 -1，並將 errno 設定爲 EINPROGRESS。接著你要呼叫 select() 並設定你想要的 timeout 時間，傳遞讀取及寫入集合（read and write sets）的 socket descriptor。如果 select() 沒有發生 timeout，這表示 connect() call 已經完成。此時，你必須使用 getsockopt() 設定 SO_ERROR 選項以取得 connect() call 的傳回值，在沒有錯誤時，這個值應該是零。

最後，在你開始透過 socket 傳輸資料以前，你可能想要再將它設定回 blocking（阻塞）。

要注意的是，這樣做的好處是讓你的程式在連線（connecting）期間也可以另外做點事情。比如：你可以將 timeout 時間設定爲類似 500 毫秒，並在每次 timeout 發生時更新螢幕畫面，然後再次呼叫 select()。當你已經呼叫了 select() 時，並且 timeout 了，像這樣重複了 20 次，你就會知道應該放棄這個連線了。

如我所述的，請參考 Stevens 的既完美又優秀的範例程式碼。

我該如何寫 **Windows** 的網路程式呢？

首先，請刪除 Windows，並安裝 Linux 或 BSD。;-)。不是的，實際上，只要參考導讀章節中的（Windows程式設計師要注意的事情）就可以了。

我該如何在 **Solaris/SunOS** 上編譯程式呢？在我嘗試編譯時，一直遇到錯誤！

發生 Linker（連結器）錯誤是因爲 Sun 系統在不會自動編入 socket 函式庫。請參考導讀中的（Solaris/SunOS 程式設計師要注意的事情），有如何處理這個問題的範例。

爲什麼 **select()** 跟 **signal** 合不來呢？

Signal 試圖要讓 blocked system call 傳回 -1，並將 errno 設定爲 EINTR。當你用 sigaction() 設定了一個 signal handler（訊號處理常式）時，你可以設定 SA_RESTART 旗標，這可以在 system call 被中斷之後重新啓用它。

這自然不會每次都管用。

我最愛的解法是使用一個 goto，你明白這會讓你的教授很憤怒，所以放手去做吧！

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
  if (errno == EINTR) {
    // 某個 signal 中斷了我們，所以重新啓動
    goto select_restart;
  }
  // 這裡處理眞正的錯誤：
  perror("select");
}
```

當然，在這個例子裡，你不需使用 goto；你可以用其它的 structures 來控制，但是我認爲用 goto 比較簡潔。

要怎麼樣我才能實作呼叫 **recv()** 的 **timeout** 呢？

使用 select()！它可以讓你對正在讀取的 socket descriptors 指定 timeout 的參數。或者你可以將整個功能包在一個獨立的函式中，類似這樣：

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

int recvtimeout(int s, char *buf, int len, int timeout)
{
  fd_set fds;
  int n;
  struct timeval tv;

  // 設定 file descriptor set
  FD_ZERO(&fds);
  FD_SET(s, &fds);

  // 設定 timeout 的資料結構 struct timeval
  tv.tv_sec = timeout;
  tv.tv_usec = 0;

  // 一直等到 timeout 或收到資料
  n = select(s+1, &fds, NULL, NULL, &tv);
  if (n == 0) return -2; // timeout!
  if (n == -1) return -1; // error

  // 資料一定有在這裡，所以執行一般的 recv()
  return recv(s, buf, len, 0);
}
.
.
.
// 呼叫 recvtimeout() 的範例：
n = recvtimeout(s, buf, sizeof buf, 10); // 10 second timeout

if (n == -1) {
  // 發生錯誤
  perror("recvtimeout");
}
else if (n == -2) {
  // 發生 timeout
} else {
  // 從 buf 收到一些資料
}
.
.
.
```

請注意到，recvtimeout() 在 timeout 的例子中會傳回 -2，那爲什麼不是傳回 0 呢？好的，如果你還記得，在呼叫 recv() 傳回 0 值時所代表的意思是對方已經關閉了連線。所以該傳回值已經用過了，而 -1 表示”錯誤”，所以我選擇 -2 做爲我的 timeout 表示。

我該如何在將資料送給 **socket** 以前將資料加密或壓縮呢？

一個簡單的加密方法是使用 SSL（secure sockets layer），只是這超過本文件的範疇了〔細節請參考 OpenSSL 專案 [39]〕。

不過假設你想要安插或實作你自己的壓縮器（compressor）或加密系統（encryption system），這只不過是將你的資料想成在兩端點間執行連續的步驟，每個步驟以同樣的方式改變資料：

1. server 從檔案讀取資料〔或是什麼地方〕
2. server 加密/壓縮資料〔你新增這個部分〕
3. server 用 send() 送出加密資料

而另一邊則是：

1. client 用 recv() 接收加密資料
2. client 解密/解壓縮資料〔你新增這個部分〕
3. client 寫資料到檔案〔或是什麼地方〕

如果你正要壓縮與加密，只要記得先壓縮。:-)

只要 client 適當地還原 server 所做的事情，資料在另一端就會完好如初，不論你在中間增加了多少步驟。

所以你用我的程式碼所需要做的只有：找出讀資料與透過網路傳送〔使用 send()〕這中間的段落，並在那裡加上編碼的程式碼。

我一直看到的”PF_INET”是什麼呢？他跟 AF_INET 有關係嗎？

是的，有關係，細節請參考 socket() 章節。

我該怎麼寫一個 **server**，可以接受來自 **client** 的 **shell** 指令並執行指令呢？

爲了簡化，我們說 client 的連線用 connect()、send() 以及 close()〔即爲，沒有後續的 system calls，client 沒有再次連線。〕

client 的處理過程是：

1. 用 connect() 連線到 server
2. send("/sbin/ls > /tmp/client.out")
3. 用 close() 關閉連線

此時，server 正在處理資料並執行指令：

1. accept() client 的連線
2. 使用 recv(str) 接收命令字串
3. 用 close() 關閉連線
4. 用 system(str) 執行指令

注意！server 會執行全部 client 所送的指令，就像是提供了遠端的 shell 存取權限，人們可以連線到你的 server 並用你的帳號做點事情。例如：若 client 送出 ”rm -rf ~” 會怎麼樣呢？這會刪掉你帳號裡的全部資料，就是這樣！

所以你學聰明了，你會避免 client 使用任何危險的工具，比如 foobar 工具：

```
if (!strncmp(str, "foobar", 6)) {
  sprintf(sysstr, "%s > /tmp/server.out", str);
  system(sysstr);
}
```

可是這樣還是不安全，沒錯：如果 client 輸入 "foobar; rm -rf ~" 呢？

最安全的方式是寫一個小機制，將命令參數中的非字母數字字元前面放個〔'\'〕字元〔如果適合的話，要包括空白〕。

如你所見，當 server 開始執行 client 送來的東西時，安全性（security）是個問題。

我正在傳送大量資料，可是當我 **recv()** 時，它一次只收到 **536 bytes** 或 **1460 bytes**。可是如果我在我本機上執行，它就會一次就收到全部的資料，這是怎麼回事呢？

你碰到的是 MTU，即實體媒介（physical medium）能處理的最大尺寸。在本機上，你用的是 loopback 裝置，它可以處理 8K 或更多資料也沒有問題。但是在 Ethernet（乙太網路），它只能處理 1500 bytes（有 header），你碰到這個限制。透過 modem 的話，MTU 是 576 bytes（一樣，有 header），你遇到比較低的限制。

你必須確認有送出全部的資料。（細節請參考 sendall() 函式的實作）。一旦你有確認，那麼你就需要在迴圈中呼叫 recv()，直到收到全部的資料。

對於使用多重呼叫 recv() 來接收完整資料封包的細節，請參考資料封裝（Son of Data Encapsulation）一節。

我用的是 **Windows** 系統，而且我沒有 **fork() system call** 或任何的 **struct sigaction** 可以用，該怎麼辦呢？

如果你問的是它們在哪裡，它們會在 POSIX 函式庫裡，這個會包裝在你的編譯器中。因爲我沒有 Windows 系統，所以我眞的無法回答你，不過我似乎記得 Microsoft 有一個 POSIX 相容層，那裏會有 fork()（而且甚至會有 sigaction）。

在 VC++ 的使用手冊搜尋 "fork" 或 "POSIX"，看它是否能給你什麼線索。

如果這樣一點都沒有用，拿掉 fork()/sigaction 這些東西，用 Win32 中等價的函式來取代：CreateProcess()。我不知道怎麼用CreateProcess()，它有多的數不清的參數，不過在 VC++ 的文件中應該可以找到怎麼使用它。

我在防火牆（**firewall**）後面，我該如何讓防火牆外面的人知道我的 **IP** 位址，讓他們可以連線到我的電腦呢？

毫無疑問地，防火牆的目的就是要防止防火牆外面的人連到防火牆裡面的電腦，所以你讓他們進來基本上會被認爲是安全上的漏洞。

但也不是說完全不行，有一個方法，你仍然可以透過防火牆頻繁的進行 connect()，如果防火牆是使用某種僞裝（masquerading）或 NAT 或類似的方式。你只要讓程式一直在做初始化連線，那麼你有機會成功的。

如果這樣還不是很滿意，你可以要求系統管理員在防火牆開一個小洞（hole），讓人們可以連進你的電腦。防火牆可以透過 NAT 軟體或 proxy（代理）或類似的方法將封包轉送給你。

要留意，不要對防火牆中的一個小洞掉以輕心。你必須確保你不會放壞人進來存取內部網路；如果你是新手，做軟體安全是遠遠難於你的想像。

不要讓你的系統管理員對我發脾氣 ;-)

我該怎麼寫 **packet sniffer** 呢？我要怎麼將我的 **Ethernet interface**（網路卡）設定爲 **promiscuous mode**（混雜模式）呢？

這些事情是在底層運作的，當網路卡設定爲 "promiscuous mode" 時，它會轉送全部的封包給作業系統，而不只是位址屬於這台電腦的封包而已。〔我們這裡談的是 Ethernet 層的位址，而不是 IP 位址，可是因爲 ethernet 是在 IP 底層，所以全部的 IP 位址實際上都會轉送。細節請參考”底層漫談與網路理論”一節〕。

這是 packet sniffer 如何運作的基礎，它將網路介面卡設定爲 promiscuous mode，接著 OS 會收到經過網路線的每個封包，你會有一個可以用來讀取資料的某種型別 socket。

毫無疑問地，這個問題的答案依平台而異，不過如果你用 Google 搜尋，例如："windows promiscuous ioctl"，你或許會在某個地方找到，看起來跟 Linux Journal [40] 中寫的一樣好的。

我該如何爲 TCP 或 UDP socket 設定一個自訂的 timeout 值呢？

這個按照你的系統而定，你可以在網路搜尋 SO_RCVTIMEO 與 SO_SNDTIMEO（用在 setsockopt()），看看是否你的系統有支援這樣的功能。

Linux man 使用手冊建議使用 alarm() 或 setitimer() 作爲替代品。

我要如何辨別哪些 ports 可以使用呢？有沒有”官方”的 port numbers 呢？

通常這不會有問題，如果你正在寫像 web server 這樣的程式，那麼在你的程式使用 port 80 是個好主意。如果你只是想要寫自己的 server，那麼隨機選擇一個 port〔不過要大於 1023〕，然後試試看。

如果 port 已經在使用中，你將會在嘗試 bind() 時遇到 "Address already in use" 錯誤。選擇另一個 port。〔利用 config 組態檔或命令列參數設定，讓你的軟體使用者能指定 port 也是個不錯的想法〕。

有一個官方的 port nubmer [41] 清單，由 Internet Assigned Numbers Authority（IANA）所維護的。在清單中的 port（超過 1023）並不代表你就不能使用，比如，Id 軟體的 DOOM 跟 ”mdqs” 用一樣的 port，不管那是什麼，最重要的是在同一台機器上沒有人用掉你要用的 port。

[37] http://www.unpbook.com/src.html

[38] http://www.unpbook.com/src.html

[39] http://www.openssl.org/

[40] http://interactive.linuxjournal.com/article/4659

[41] http://www.iana.org/assignments/port-numbers

Man 使用手冊

請參考下列各函式的子頁。

accept()

接受從 listening socket 進來的連線

函式原型（Prototype）

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

說明

一旦你完成取得 SOCK_STREAM socket 並將 socket 設定好可以用來 listen() 進來的連線，接著你就能呼叫 accept() 讓自己能取得一個新的 socket descriptor，做爲後續與新連線 client 的溝通。

原本用來 listen 的 socket 仍然還是會留著，當有新連線進來時，一樣是用 accept() call 來接受新的連線。

- s：listen() 中的 socket descriptor。
- addr：這裡會填入連線到你這裡的 client 位址。
- addrlen：這裡會填入 addr 參數中傳回的資料結構大小。如果你確定你知道一定會收到的是 struct sockaddr_in，你可以放心忽略這個參數，因爲這就是你原本傳遞的 addr 型別。

accept() 通常會 block（阻塞），而你可以使用 select() 事先取得 listen 中的 socket descriptor 狀態，檢查 socket 是否就緒可讀（ready to read）。若爲就緒可讀，則表示有新的連線正在等待被 accept()！另一個方式是將 listen 中的 socket 使用 fcntl() 設定 O_NONBLOCK 旗標，然後 listen 中的 socket descriptor 就不會造成 block，而是傳回 -1，並將 errno 設定爲 EWOULDBLOCK。

由 accept() 傳回的 socket descriptor 是如假包換的 socket descriptor，開啓並與遠端主機連線，如果你要結束與 client 的連線，必須用 close() 關閉。

傳回值

accept() 傳回新連線的 socket descriptor，錯誤時傳回 -1，並將 errno 設定適當的值。

範例

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// 首先：使用 getaddrinfo() 填好位址結構：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // 使用 IPv4 或 IPv6，都可以
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // 幫我填好我的 IP

getaddrinfo(NULL, MYPORT, &hints, &res);

// 建立一個 socket、bind 它，並對它進行 listen：

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
listen(sockfd, BACKLOG);

// 現在接受進入的連線：

addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

// 準備與 socket descriptor new_fd 溝通！
```

參考

socket(), getaddrinfo(), listen(), struct sockaddr_in

bind()

將一個 socket 關聯到一個 IP address 及 port number

函式原型

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

說明

當遠端機器想要連線到你的伺服器程式時，他需要兩項資訊：IP address 及 port number。而 bind() call 可以讓你做這件事。

首先你要先呼叫 getaddrinfo() 載入 struct sockaddr，以取得 destination address（目地位址）與 port 的資訊。然後你呼叫 socket() 以取得一個 socket descriptor，接著將這個 socket 與 address 傳遞給 bind()，然後 IP address 跟 port 就會很神奇的跟 socket 綁在一起了（使用眞的魔法）！

如果你不知道你電腦的 IP address、或你知道你的電腦只有一個 IP address、或你不在意要用電腦上的哪個 IP address 時，你可以單純地將 getaddrinfo() 的 hints.ai_flags 參數設定爲 AI_PASSIVE，這裡所做的事情就是將特別的值填入 struct sockaddr 的 IP address 欄位，以告訴 bind() 說它應該要自動填入這個主機的 IP address。

什麼？將什麼特別的值填入 struct sockaddr 的 IP address 欄位就可以讓它自動填上目前主機的 address 呢？

我會告訴你的，但是請記住這只適用於你手動填入 struct sockaddr 時，如果不是，請依據上述方式，使用 getaddrinfo() 傳回的結果。在 IPv4 中， struct sockaddr_in 結構的 sin_addr.s_addr 會設定爲 INADDR_ANY；而在 IPv6 中， struct sockaddr_in6 結構的 sin6_addr 欄位會從全域變數 in6addr_any 載入。或者，若你正宣告一個新的 struct in6_addr，你可以將它初始化爲 IN6ADDR_ANY_INIT。

最後，addrlen 參數應該設定爲 my_addr 的大小。

傳回值

成功時傳回零，或者錯誤時傳回 -1（且並依據錯誤設定 errno）

範例

```
// 用現代化的 getaddrinfo() 方式：

struct addrinfo hints, *res;
int sockfd;

// 首先，用 getaddrinfo() 載入位址結構資料：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // 使用 IPv4 或 IPv6，兩者皆可
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // 幫我填上我的 IP

getaddrinfo(NULL, "3490", &hints, &res);

// 建立 socket：
//（這邊是簡化版，照理你應該要查看 "res" 鏈結串列的每個成員，並進行錯誤檢查！）

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// 將 sockfd 綁定（bind）到我們傳遞給 getaddrinfo() 的那個 port：

bind(sockfd, res->ai_addr, res->ai_addrlen);

// 手動封裝 struct 的範例，IPv4

struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// 你可以指定一個 IP address:
inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

// 或者你可以讓系統自動選一個 IP address：
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

參考

getaddrinfo(), socket(), struct sockaddr_in, struct in_addr

connect()

使用 socket 連線到伺服器

函式原型

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

說明

一旦你用 socket() call 建立了一個 socket descriptor 之後，你可以使用已知的 connect() system call 將這個 socket 連線到遠端伺服器。你該需做的是將 socket descriptor 與要連線的伺服器 address 傳遞給 connect()（喔，還有 address 的長度，這個通常會傳給這類的函式）。

通常這項資訊會透過呼叫 getaddrinfo() 取得，但是如果你願意，你也可以自行填寫 struct sockaddr。

若你尚未對這個 socket descriptor 呼叫 bind()，則會自動幫你綁定到你的 IP address 與一個隨機的 local port（本地連接埠）。如果你不是 伺服器，這樣還蠻不錯的，因爲你眞的可以不用管理的 local port 是多少了；你只要注意遠端的 port，這個可以在 serv_addr 參數中設定。如果你有需要將 client socket 指定 IP address 與 port，那麼你可以選擇呼叫 bind() 來處理，不過這種情況是很少見的。

一旦 socket 已經用 connect() 完成連線，你就可以隨意使用這個 socket 來 send() 與 recv() 資料到你心裡想的地方。

特別注意：如果你用 connect() 與 SOCK_DGRAM UDP socket 連線到遠端主機，若你願意，你也可以像使用 sendto() 與 recvfrom() 那樣來使用 send() 與 recv()。

傳回值

成功時傳回零，或者發生錯誤時傳回 -1（並設定相對應的 errno）。

範例

```
// 連線到 www.example.com 的 port 80（http）

struct addrinfo hints, *res;
int sockfd;

// 首先，使用 getaddrinfo() 取得位址資訊：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;

// 我們可以在下一行用 "http" 取代 "80"：
getaddrinfo("www.example.com", "http", &hints, &res);

// 建立一個 socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// 將 socket 連線到我們在 getaddrinfo() 裡指定的 address 與 port：

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

參考

socket(), bind()

close()

關閉 socket descriptor

函式原型

```
#include <unistd.h>

int close(int s);
```

說明

無論你設計了多棒的機制，在你用完 socket 且不想要 send() 或 recv() 時，也就是不會再對這個 socket 做任何事情時，你可以將它 close()，而它就會被釋放，而且永遠不會再用到。

遠端可以用兩種方法判斷對方是否已經關閉 socket，一種是：如遠端呼叫的 recv() 傳回 0；另一種是：若遠端呼叫 send() 時收到一個 SIGPIPE 的訊號（signal），且 send() 傳回 -1 並將 errno 設定爲 EPIPE。

Windows 系統的使用者：你要用的函式是 closesocket() 而不是 close()，如果你試著使用 close() 關閉 socket descriptor，那麼 Windows 系統有可能會震怒 ... 它如果敢生氣你就別再愛它了。

傳回值

成功時傳回零，或者錯誤時傳回 -1（並設定相對應的 errno）。

範例

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// a whole lotta stuff...*BRRRONNNN!*
.
.
.
close(s); // 沒有多少，眞的。
```

參考

socket(), shutdown()

getaddrinfo(), freeaddrinfo(), gai_strerror()

取得主機名稱或服務的資訊，並將結果載入 struct sockaddr。

函式原型

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

const char *gai_strerror(int ecode);

struct addrinfo {
  int ai_flags; // AI_PASSIVE, AI_CANONNAME, ...
  int ai_family; // AF_xxx
  int ai_socktype; // SOCK_xxx
  int ai_protocol; // 0 (auto) or IPPROTO_TCP, IPPROTO_UDP

  socklen_t ai_addrlen; // ai_addr 的長度
  char *ai_canonname; // canonical name for nodename
  struct sockaddr *ai_addr; // 二進制格式位址
  struct addrinfo *ai_next; // linked list 中的下個資料結構
};
```

說明

getaddrinfo() 是很優秀的函式，可以傳回特別的主機名稱資訊（比如它的 IP address）以及爲你載入 struct sockaddr，要注意一些細節（像是 IPv4 或 IPv6）。它會取代舊有的 gethostbyname() 及 getservbyname() 函式。下列的說明包含許多資訊，可能看起來有點難，但實際上卻很簡單。先看看範例是值得的。

你感興趣的 host name 在 node name 參數中，address 可以是個 host name，像 "www.example.com" 或者是 IPv4 或 IPv6 位址（以字串傳遞）。如果你使用 AI_PASSIVE flag，那這個參數也可以是 NULL（參考下面）。

servname 參數基本上就是 port number，它可以是個 port number（以字串傳遞，如 "80"），或者是個 service name，像是 "http"、"tftp"、"smtp"、或 "pop" 等。常見的 service name 可以在 IANA Port List [42] 或你的 /etc/services 中找到。

最後的 input 參數是 hints，這就是你要定義 getaddinfo() 函式要做什麼事的地方，使用以前先用 memset() 將整個資料結構清爲零，在用它之前，我們先看一下需要設定的欄位。

ai_flags 可以設定成各種東西，但是這邊有件重要的事情。（可以用 OR 位元運算〔|〕指定多個 flags）完整的 flags 清單請參考你的 man 使用手冊。

AI_CANONNAME 會讓 ai_canonname 填上主機的 canonical (real) name，AI_PASSIVE 讓 IP address 填上 INADDR_ANY （IPv4）或 in6addr_any（IPv6）；這讓之後在呼叫 bind() 時，可以自動用目前 host 的 address 來填上 struct sockaddr 的 IP address。這在設定 server 且你不想要寫死 address 時非常好用。

如果你使用 AI_PASSIVE flag，那麼你可以在 nodename 中傳遞 NULL（因爲 bind() 在之後會幫你填上）

[42] http://www.iana.org/assignments/port-numbers

繼續談輸入的參數，你應該會想要將 ai_family 設定爲 AF_UNSPEC，這樣可以讓 getaddrinfo() 知道 IPv4 與 IPv6 addresses 都需要查詢。你也能自己以 AF_INET 或 AF_INET6 自訂要使用 IPv4 或 IPv6。

再來，socktype 欄位應該要設定爲 SOCK_STREAM 或 SOCK_DGRAM，取決與你需要哪種類型的 socket。

最後，你可以將 ai_protocol 保留爲 0，可以自動選擇你的 protocol type。

在你取得全部的東西之後，你終於可以呼叫 getaddrinfo() 了！

當然，這個地方開始有趣了，res 會指向一個 struct addrinfo 的鏈結串列，而你可以透過這個串列取得全部的 addresses（符合你在 hints 中指定的 address 類型）。

現在可能會取得一些因爲某些理由無法正常運作的 addresses，所以 Linux man 使用手冊提供的方法是不斷用迴圈讀取串列，然後試著呼叫 socket()、connect()（如果以 AI_PASSIVE flag 設定 server，則是 bind()），直到成功爲止。

最後，當你處理完鏈結串列之後，你需要呼叫 freeaddrinfo() 來釋放記憶體（否則會發生 memory leak，這會讓有些人不安。）

傳回值

成功時傳回零，或錯誤時傳回非零值。若傳回非零值，你可以代入 gai_strerror() 函式取得一個文字版的錯誤訊息。

範例

```
// client 連線到 server 的程式碼
// 透過 stream socket 連線到 www.example.com 的 port 80 (http)
```

```
// 不是 IPv4 就是 IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // 設定 AF_INET6 表示強迫使用 IPv6
hints.ai_socktype = SOCK_STREAM;

if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
  exit(1);
}

// 不斷執行迴圈，直到我們可以連線成功
for(p = servinfo; p != NULL; p = p->ai_next) {
  if ((sockfd = socket(p->ai_family, p->ai_socktype,  p->ai_protocol)) == -1) {
      perror("socket");
      continue;
  }

  if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
    close(sockfd);
    perror("connect");
    continue;
  }

  break; // if we get here, we must have connected successfully
}

if (p == NULL) {
  // 迴圈已經執行到 list 的最後，都無法連線
  fprintf(stderr, "failed to connect\n");
  exit(2);
}

freeaddrinfo(servinfo); // 釋放 servinfo 記憶體空間

// code for a server waiting for connections
// namely a stream socket on port 3490, on this host's IP
// either IPv4 or IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // 使用 AF_INET6 表示一定要用 IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // 使用我的 IP address

if ((rv = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
```

```
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
  }

  // 不斷執行迴圈，直到我們可以成功綁定
  for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
        p->ai_protocol)) == -1) {
      perror("socket");
      continue;
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
      close(sockfd);
      perror("bind");
      continue;
    }

    break; // 若執行到這行，表示我們一定已經成功連線
  }

  if (p == NULL) {
    // 整個迴圈執行完畢，到了 linked-list 結尾都無法成功綁定（bind）
    fprintf(stderr, "failed to bind socket\n");
    exit(2);
  }

  freeaddrinfo(servinfo); // 使用完畢時釋放這個資料結構的空間
```

參考

gethostbyname(), getnameinfo()

gethostname()

傳回系統的主機名稱（host name）

函式原型

```
#include <sys/unistd.h>
int gethostname(char *name, size_t len);
```

說明

你的系統有一個名字，大家都有的，這比較偏向系統層面，而不是我們正在談論的網路層面，只是它仍有其用途。

例如：你可以取得你的主機名稱，接著呼叫 gethostbyname() 找出你電腦的 IP address。

name 參數應該指向一個存有主機名稱的緩衝區，而 len 是該緩衝區的大小，以 byte 爲單位。gethostname() 不會覆寫緩衝區的結尾（可能會傳回錯誤，或者只是單純停止寫入），而且如果緩衝區有足夠的空間，它還會保留字串的 NUL-結尾。

傳回值

成功時傳回零，或者錯誤時傳回 -1（並設定相對應的 errno）。

範例

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("My hostname: %s\n", hostname);
```

參考

gethostbyname()

gethostbyname(), gethostbyaddr()

取得 hostname 的 IP address for a hostname，反之亦然

函式原型

```
#include <sys/socket.h>
#include <netdb.h>

struct hostent *gethostbyname(const char *name); // 不建議使用！
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

說明

請注意：這兩個函式已經由 getaddrinfo() 與 getnameinfo() 取而代之！實際上，gethostbyname() 無法在 IPv6 中正常運作。

這些函式可以轉換 host names 與 IP addresses。例如：你可以用 gethostbyname() 取得 " 其 IP addresses，並儲存在 struct in_addr。

反之，如果你有一個 struct in_addr 或 struct in6_addr，你可以用 gethostbyaddr() 取回 hostname。gethostbyaddr() 與 IPv6 相容，但是你應該使用新的 getnameinfo() 取代之。

（如果你有一個字串是句點與數字組成的格式，你想要查詢它的 hostname，你在使用 getaddrinfo() 時最好要搭配 AI_CANONNAME flag）。

gethostbyname() 接收一個類似 "www.yahoo.com" 的字串，然後傳回一個 struct hostent，裡面包含幾萬噸的資料，包括了 IP address（其它的資訊是官方的 host name、一連串的別名、位址型別、位址長度、以及位址清單。這是個通用的資料結構，在特定的用途上也很易於使用）。

在 gethostbyaddr() 代入一個 struct in_addr 或 struct in6_addr，然後就會提供你一個相對應的 host name（如果有），因此，它是 gethostbyname() 的相反式。至於參數，addr 是一個 char*，你實際上想要用一個指向 struct in_addr 的指標傳遞；len 應是 sizeof(struct in_addr)，而 type 應爲 AF_INET。所以這個 struct hostent 會帶回什麼呢？它有許多欄位，包含 host 的相關資訊。

char *h_name* 眞正的 *real canonical host name*。 *char* ***h_aliases*** 一連串的別名，可以用陣列存取—最後一個元素（***element***）是 ***NULL***。 ***int h_addrtype address type*** 的答案，這個在我們的用途應該是 ***AF_INET***。 ***int length address*** 的長度（以 ***byte*** 爲單位），這個在 ***IP (version 4) address*** 是 ***4***。 ***char*** *h_addr_list* 這個主機的 *IP addresses* 清單。雖然這是個

*char***，不過實際上是 *struct in_addr*s 所僞裝的陣列，最後一個元素是 NULL。 h_addr 爲 h_addr_list[0] 所定義的通用別名，如果你只是想要任意一個舊有的 IP addres，就用這個欄位吧。（耶，它們可以大於一個）。

傳回值

成功時傳回指向 struct hostent 結果的指標，錯誤時傳回 NULL。

跟你平常使用的錯誤報告工具不同，也不是一般的 perror()，這些函式在 h_errno 變數中有同樣的結果，可以使用 herror() 或 hstrerror() 函式印出來，這些函式運作的方式類似你常用的典型 errnor、perror() 及 strerror() 函式。

範例

```
// 不建議使用這個方式取得 host name
// 建議使用 getaddrinfo()！

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
  int i;
  struct hostent *he;
  struct in_addr **addr_list;

  if (argc != 2) {
    fprintf(stderr,"usage: ghbn hostname\n");
    return 1;
  }

  if ((he = gethostbyname(argv[1])) == NULL) { // 取得 host 資訊
    herror("gethostbyname");
    return 2;
  }

  // 印出關於這個 host 的資訊：
  printf("Official name is: %s\n", he->h_name);
  printf(" IP addresses: ");
  addr_list = (struct in_addr **)he->h_addr_list;
  for(i = 0; addr_list[i] != NULL; i++) {
    printf("%s ", inet_ntoa(*addr_list[i]));
  }
  printf("\n");

  return 0;
}
// 這個方法已經被 getnameinfo() 取代了

struct hostent *he;
struct in_addr ipv4addr;
struct in6_addr ipv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
printf("Host name: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6);
printf("Host name: %s\n", he->h_name);
```

参考

getaddrinfo(), getnameinfo(), gethostname(), errno, perror(), strerror(), struct in_addr

getnameinfo()

由 struct sockaddr 提供的資訊查詢主機名稱（host name）及服務名稱（service name）資訊。

函式原型

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

說明

這個函式是 getaddrinfo() 的對比，將已填好的 struct sockaddr 代入這個函式，就可以查詢 hostname 與 service。它取代了舊有的 gethostbyaddr() 與 getservbyport() 函式。

你必須將一個指向 struct sockaddr 的指標（實際上這個可能是你轉型過的 struct sockaddr_in 或 struct sockaddr_in6 ）傳遞給 sa 參數，而這個 struct 的長度是 salen。

結果會將查到的 host name 與 service name 寫入 host 與 serv 參數所指的區域空間裡，當然，你必須用 hostlen 與 servlen 來指定這些緩衝區的最大長度。

最後，有幾個你能傳遞的 flags（旗標），但是這裡有兩個好物。NI_NOFQDN 會讓 host 只包含 host name，而不是全部的 domain name（網域名稱）；如果在 DNS 查詢時無法找到 name 的時候，NI_NAMEREQD 會讓函式發生失敗（如果你沒有指定這個 flag，而又無法找到 name 時，那 getnameinfo() 就會改爲將一個字串版本的 IP address 放在 host 裡面）。

同樣地，完整的內容請參考你電腦上的 man 使用手冊。

傳回值

成功時傳回零，或錯誤時傳回非零。若傳回值是非零，則可以將傳回值傳遞給 gai_strerror()，以取得我們可讀的字串，細節請參考 getaddrinfo 。

範例

```
struct sockaddr_in6 sa; // 如果你想，也可以是 IPv4
char host[1024];
char service[20];

// 假設 sa 充滿了主機與 port 相關的資訊 ...

getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0);

printf(" host: %s\n", host); // 例如: "www.example.com"
printf("service: %s\n", service); // 例如: "http"
```

參考

getaddrinfo(), gethostbyaddr()

getpeername()

傳回遠端連線的位址資訊

函式原型

```
#include <sys/socket.h>

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

說明

一旦你接受了〔accept()〕遠端的連線，或者連線到〔connect()〕一個 server，你現在已經知道對方端點（peer）的資訊了，這個 peer 就是你所連線的電腦，透過一個 IP address 與一個 port 來識別，所以 ...

getpeername() 單純傳回一個 struct sockaddr_in，裡面包含了與你連線的主機資訊。

爲什麼它稱爲 "name" 呢？

好，有許多不同的 socket 類型，不僅是我們這本書所使用的 Internet Sockets，所以 "name" 是一個好的通用術語，可以涵蓋全部的例子。以我們這個例子而言，peer 的 "name" 就是它的 IP address 與 port。

因爲這個函式傳回 len 所指定的 address 大小，所以你必須預先將 addr 的大小存放在 len 裡面。

傳回值

成功時傳回零，錯誤時傳回 -1（並設定相對應的 errno）。

範例

```
// 假設 s 是已連線的 socket

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
int port;

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);

// 處理 IPv4 與 IPv6：
if (addr.ss_family == AF_INET) {
    struct sockaddr_in *s = (struct sockaddr_in *)&addr;
    port = ntohs(s->sin_port);
    inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof ipstr);
} else { // AF_INET6
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr;
    port = ntohs(s->sin6_port);
    inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof ipstr);
}

printf("Peer IP address: %s\n", ipstr);
printf("Peer port      : %d\n", port);
```

參考

gethostname(), gethostbyname(), gethostbyaddr()

errno

儲存上次 system call 的錯誤碼

函式原型

```
#include <errno.h>

int errno;
```

說明

這個變數用來儲存 system call 的錯誤資訊，比如像：socket() 與 listen() 錯誤時會傳回 -1，並設定 errno 的值，讓你可以知道發生了什麼樣的錯誤。

標頭檔 errno.h 列出了許多錯誤的常數符號，比如：EADDRINUSE、EPIPE、ECONNREFUSED 等。你本機上的 man 手冊會告訴你錯誤時所傳回的錯誤碼有哪些，而你可以在執行期時利用這些錯誤碼來分別處理不同的錯誤。

或者，更普遍的是，你可以呼叫 perror() 或 strerror() 取得人們方便閱讀的錯誤訊息。

對於喜歡用多執行緒（multithreading）的人，有件事需要注意：就是多數系統上的 errno 會定義爲 threadsafe 的方式。（也就是說，它並不是一個全域變數，但是在單執行緒的環境中，它的行爲跟全域變數一樣）。

傳回值

變數值記錄最新的錯誤，如果上次的動作是成功的，這個值就會是 "success"。

範例

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror("socket"); // 或者使用 strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
    // 發生錯誤！！

    // 如果我們只有被中斷，則只需重新啓動 select() call:
    if (errno == EINTR) goto tryagain;  // AAAA!  goto!!!

    // 否則，就是個嚴重的錯誤：
    perror("select");
    exit(1);
}
```

參考

perror(), strerror()

fcntl()

控制 socket descriptors

函式原型

```
#include <sys/unistd.h>
#include <sys/fcntl.h>

int fcntl(int s, int cmd, long arg);
```

說明

這個函式通常用來進行檔案鎖定與其它檔案導向的用途，但是它也有幾個與 socket 相關的功能，你之後可能會用的到。

參數 s 是 你想要控制的 socket descriptor，而 cmd 應該要設定爲 F_SETFL，至於 arg 可以是下列任一個指令（如我所述，參數比我這邊介紹的還要多，但是我只會探討與 socket 相關的部分）。

- O_NONBLOCK 將 socket 設定爲 non-blocking，細節請參考 blocking 章節。
- O_ASYNC 將 socket 設定爲非同步 I/O（asynchronous I/O），當 socket 上的資料就緒可收時，會觸發 SIGIO 訊號。這幾乎不會遇到，並且在本書的討論範圍之外，而我認爲這只會在某些系統上出現。

傳回值

成功時傳回零，錯誤時傳回 -1（並設定相對應的 errno）

fcntl() system call 依據不同的用法會有不同的傳回值，但是我在這裡不會全部涵蓋，因爲有些與 socket 無關，細節請參考 fcntl() 的 man 手冊。

範例

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // 設定爲非阻塞（non-blocking）
fcntl(s, F_SETFL, O_ASYNC);     // 設定爲非同步 I/O
```

參考

Blocking, send()

htons(), htonl(), ntohs(), ntohl()

將多位元組整數型別（multi-byte integer types）由 host byte order 轉換爲 network byte order

函式原型

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

說明

這眞的只是來亂的，不同的電腦在多位元組整數〔比如：任何比字元長的整數〕都用不同的 byte orderings（位元組順序）。結果變成，如果你從 Intel 主機用 send() 送出一個兩 bytes 的 short int 給 Mac 主機〔我是指在它們都成爲 Intel 架構以前的事了〕，在一台電腦上認爲是 1 的數值，到了另一台電腦上卻可能是 256，反之亦然。

避免這個問題的方法是請大家拋開這些差異，並同意 Motorola 與 IBM 才是正道，而 Intel 的方法是旁門左道。所以我們應該在送出資料以前，先將我們的 byte ordering 轉換爲 "big-endian"，因爲 Intel 是 "little-endian" 的電腦。我們將偏好的 byte order 稱爲 "Network Byte Order" 是很正確的，所以這些函式會將你電腦的 byte order 轉換爲 network byte order，然後對方再轉回來。

（這表示在 Intel 電腦上，這些函式旋轉了全部的 bytes 順序，而在 PowerPC 上，它們什麼都不用做，因爲 PowerPC 就是以 Network Byte Order 的方式儲存資料。但無論如何你的程式碼應該每次都要用這些函式，因爲有人可能會想在 Intel 電腦上執行你的程式，而你的程式還是得正常運作。）

注意，相關的型別是 32-bit（4 byte，可能是 int）與 16-bit（2 byte，很有可能是 short）的數值。64-bit 的電腦可能有一個 htonll() 來轉換 64-bit 的整數，但是我還沒看過，你可以自己寫一個。

不管怎樣，這些函式運作的方式就是，你要先決定是從（你電腦的）host byte order 轉換，還是從 network byte order 轉換。如果是從 "host"，那你要呼叫的函式開頭就是 "h"，否則就是 network 的 "n"。中間的函式名稱永遠都是 "to"，因爲你是從某一種順序轉換到（"to"）另一種，而倒數第二個字母表示你要轉換的目的格式。最後的字母就是資料的大小，"s" 表示 short、"l" 表示 long，如下所示：

```
htons()   host to network short
htonl()   host to network long
ntohs()   network to host short
ntohl()   network to host long
```

傳回值

每個函式傳回轉換過的值

範例

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// 轉換與傳送
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // 這行判斷式爲眞
```

inet_ntoa(), inet_aton(), inet_addr

將 IP addresses 從句號與數字格式的字串轉換爲 struct in_addr，以及轉換回來

函式原型

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// 這些都不建議使用！請使用 inet_pton() 或 inet_ntop() 取代！

char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

說明

因爲這些函式無法處理 IPv6，所以建議不要使用！請使用 inet_ntop() 或 inet_pton() 來代替！將它們放上來的理由是因爲還是可以在某些程式碼看到它們的蹤跡。

這裡的函式都是將 struct in_addr（struct sockaddr_in 的一部分）轉換爲句號與數字組成的字串格式（例如："192.168.5.10"），反之亦然。如果你將一個 IP address 透過命令列或某種方式傳遞，這是將 struct in_addr 提供給 connect() 最簡單的方式，諸如此類。如果你需要更多權力，可以試試一些 DNS 用途的函式，比如： gethostbyname()，或者試著在你的國家發動政變。

inet_ntoa() 函式將 network address 由 struct in_addr 轉換爲句號與數字組成的字串格式，依照過去的習慣，"ntoa" 裡的 "n" 表示 network，而 "a" 表示 ASCII（所以這是 "network To ASCII" - "toa" 後綴類似它的好朋友，C 函式庫的 atoi()，這是用來將 ASCII 字串轉換爲整數）。

inet_aton() 函式則是相反的功能，將句號與數字組成的字串格式轉換到 in_addr_t（這你 struct in_addr 中 s_addr 欄位的型別）。

最後，inet_addr() 是個舊函式，基本上與 inet_aton() 是一樣的東西，理論上不宜使用，但是你還是會很常遇到它，而且如果你用了，警察也不會來找你。

傳回值

inet_aton() 若 address 是合法的，則傳回非零的值，而若位址是非法的，則傳回零。

inet_ntoa() 在 static 緩衝區中傳回句點與數字格式的字串，每次呼叫這個函式時都會覆蓋緩衝區。

inet_addr() 以 in_addr_t 傳回 address，若發生錯誤時傳回 -1（如果你試著轉換一個合法的 IP address 字串 "255.255.255.255"，也是會有同樣的結果，這就是爲什麼 inet_aton() 比較好了）。

範例

```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // 將 IP 存在 antelope

some_addr = inet_ntoa(antelope.sin_addr); // 傳回 IP
printf("%s\n", some_addr); // 輸出 "10.0.0.1"

// 而這個呼叫與上面的 inet_aton() call 相同:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

參見

inet_ntop(), inet_pton(), gethostbyname(), gethostbyaddr()

inet_ntop(), inet_pton()

將 IP addresses 轉換爲可讀的格式，及轉換回來。

函式原型

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);

int inet_pton(int af, const char *src, void *dst);
```

說明

這些函式用於處理將可讀的 IP addresses 轉換爲許多函式與 system calls 使用的二進位格式，"n" 表示 "network"，而 "p" 表示 "presentation" 或 "text presentation"。可是你可以想成它是”可印的”，而 "ntop" 表示 "network to printable"，這樣知道了嗎？

有時你不想要看一堆二進制數字格式的 IP address，你想要比較好懂的格式，類似 192.0.2.180 或 2001:db8:8714:3a90::12 這樣的格式。在這種情況下，你可以使用 inet_ntop()。

inet_ntop() 在 af 參數中代入 address family（不是 AF_INET 就是 AF_INET6），src 參數應該是個指向 struct in_addr 或 struct in6_addr 的指標，會包含你想要轉換爲字串的 address。最後，dst 與 size 是指向目地字串與該字串最大長度的指標。

dst 字串的最大長度是多少呢？ IPv4 與 IPv6 address 的最大長度是多少呢？很幸運地，有兩個 macros（巨集）可以幫你做這件事，最大長度是：INET_ADDRSTRLEN 與 INET6_ADDRSTRLEN。

有時你可能會需要將字串格式的 IP address 封裝到 struct sockaddr_in 或 struct sockaddr_in6，此時，相對的 inet_pton() 就是你需要的函式。

inet_pton() 也會在 af 參數代入一個 address family（不是 AF_INET 就是 AF_INET6）、src 參數是指向可列印格式的 IP address 字串、最後的 dst 參數指向要儲存結果的地方，這可能是 struct in_addr 或 struct in6_addr。

上述的這些函式不能做 DNS 查詢，你需要使用 getaddinfo()。

傳回值

inet_ntop() 成功時傳回 dst 參數，失敗時傳回 NULL（並設定 errno）。

inet_pton() 成功時傳回 1，有錯誤時傳回 -1（設定 errno）；若輸入的 IP address 不正確，則傳回 0。

範例

```
// inet_ntop() 與 inet_pton() 的 IPv4 demo

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// 將這個 IP address 儲存在 sa：
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// 現在取回並印出來
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // 印出 "192.0.2.33"
// inet_ntop() 與 inet_pton() 的 IPv6 Demo
// （基本上一樣，除了多個 6）

struct sockaddr_in6 sa;
char str[INET6_ADDRSTRLEN];

// 將 IP address 儲存在 sa：
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// 現在取回並印出來
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // 印出 "2001:db8:8714:3a90::12"
// 你會用到的實用函式：

//將 sockaddr address 轉換爲字串，包含 IPv4 與 IPv6：

char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
    switch(sa->sa_family) {
        case AF_INET:
            inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr),
                    s, maxlen);
            break;

        case AF_INET6:
            inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr),
                    s, maxlen);
            break;

        default:
            strncpy(s, "Unknown AF", maxlen);
            return NULL;
    }

    return s;
}
```

參考

getaddrinfo()

listen()

告訴 socket 要監聽（listen）請求的連線

函式原型

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

說明

你可以用你的 socket descriptor（利用 socket() system call 建立的），並將它提供給 listen() 用來聽取請求的連線。各位，這就是 server 與 clients 的差異。

參數 backlog 代表在 kernel 開始拒絕新連線以前，你可以有多少的連線在等待。所以當新的連線進入時，你應該盡快的用 accept() 來處理，讓 backlog 不會滿出來。試著將它設定成 10 之類，然後在高負載時，若你的 clients 開始出現”拒絕連線”時要試著將值提高。

在呼叫 listen() 以前，你的 server 應該要先呼叫 bind()，將 socket 綁定到一個 port number，這個 port number 跟 server 的 IP address 是用來讓 clients 連線的。

傳回值

成功時傳回零，或者錯誤時傳回 -1（並設定對應的 errno）。

範例

```
struct addrinfo hints, *res;
int sockfd;

// 首先，使用 getaddrinfo() 載入位址結構：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // 使用 IPv4 或 IPv6，兩者皆可
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // 幫我填上我的 IP

getaddrinfo(NULL, "3490", &hints, &res);

// 建立 socket：

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// 將它綁定到我們於 getaddrinfo() 中傳遞的 port：

bind(sockfd, res->ai_addr, res->ai_addrlen);

listen(sockfd, 10); // 將 s 設定爲 server（監聽的）的 socket

// 接著這裡會有個迴圈用來呼叫 accept() 接受連線
```

參考

accept(), bind(), socket()

perror(), strerror()

輸出易讀的錯誤訊息字串

函式原型

```
#include <stdio.h>
#include <string.h>   // strerror() 需要

void perror(const char *s);
char *strerror(int errnum);
```

說明

因爲大多的函式錯誤都是傳回 -1，並設定 errno 變數值爲某個數，如果你還可以很簡單地印出適合你的格式，這樣就更完美了。

感謝上天，perror() 可以。如果你想要在錯誤訊息前加上一些敘述，你可以將訊息透過 s 參數傳遞（或你可以將 s 參數設定爲 NULL，這樣就不會印出額外的訊息）。

簡單說，這個函式透過 errno 值，比如：ECONNRESET，然後印出比較有意義的訊息，像是 "Connection reset by peer"（對方重置連線）。

函式 strerror() 類似 perror()，不同之處是 strerror() 會依據所給的值，傳回一個指向錯誤訊息的指標（通常你會傳 errno 變數）。

傳回值

strerror() 傳回指向錯誤訊息字串的指標。

範例

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // some error has occurred
    // 印出 "socket error： " + 錯誤訊息：
    perror("socket error");
}

// 類似的
if (listen(s, 10) == -1) {
    // 這會印出 "一個錯誤： " + errno 的錯誤訊息：
    printf("an error: %s\n", strerror(errno));
}
```

參考

errno

poll()

同時對多個 socket 進行事件測試

函式原型

```
#include <sys/poll.h>

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

說明

這個函式非常類似 select()，它們兩者都監看整組 file descriptor 的事件，比如進入的資料是就緒可收〔ready to recv()〕、socket 是就緒可送〔ready to send()〕資料、out-of-band 資料是就緒可收〔ready to recv()〕、錯誤等。

基本的想法是，你透過 ufds 傳遞一個有 nfds 個 struct pollfd 的陣列，並以毫秒（millisecond）爲單位設定 timeout 時間（一秒有 1,000 毫秒），若陣列中的每個 socket descriptor 都沒有事件發生時，則隨著 timeout 時間耗盡，poll() 就會返回。如果你不需要 timeout，要讓 poll() 一直等待，那麼可以將 timeout 設定爲負值。

陣列中的每個 struct pollfd 元素（element）表示一個 socket descriptor，且包含了下列的欄位：

```
struct pollfd {
    int fd;         // the socket descriptor
    short events;   // bitmap of events we're interested in
    short revents;  // when poll() returns, bitmap of events that occurred
};
```

在呼叫 poll() 以前，將 fd 帶入 socket descriptor 的值（若你將 fd 設定爲負值，這個 struct pollfd 就會被忽略掉，並且將它的 revents 欄位設定爲零），接著用 OR 位元運算下列的 macros（巨集）以建構 events 欄位：

- POLLIN 當這個 socket 上的資料已經就緒可以收〔recv()〕時，通知（alert）我。
- POLLOUT 當我可以送〔send()〕資料到這個 socket 而不會 blocking 時，通知我。
- POLLPRI 當 out-of-band data 就緒可收時，通知我。 一旦 poll() call 返回時，會透過對上述欄位進行 OR 位元運算來建構 revents 欄位，以告訴你哪些 descriptor 目前已經有事件發生，此外，其它的欄位可能會保持原值：
- POLLERR 在這個 socket 上已經發生錯誤

- POLLHUP 遠端連線已經斷線。
- POLLNVAL fd socket descriptor 有點問題，或許是因爲沒有初始化？

傳回值

傳回 ufds 陣列中，有事件發生的元素（element）數量；如果發生 timeout 時，這個值會是零。錯誤時會傳回 -1（並設定相對應的 errno）。

範例

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);

// 假設此時我們已經都連線到 server 了
//connect(s1, ...)...
//connect(s2, ...)...

// 設定 file descriptors 陣列
//
// 在此例中，我們想要知道何時有就緒的一般資料或 out-of-band（頻外）資料要收
//

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // 要檢查是一般資料或 out-of-band 資料

ufds[1] = s2;
ufds[1].events = POLLIN; // 只檢查一般的資料

// 等待 sockets 上的事件，timeout 時間是 3.5 秒
rv = poll(ufds, 2, 3500);

if (rv == -1) {
    perror("poll"); // poll() 時發生錯誤
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 3.5 seconds.\n");
} else {
    // 檢查 s1 上的事件：
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof buf1, 0); // 接收一般的資料
    }
    if (ufds[0].revents & POLLPRI) {
        recv(s1, buf1, sizeof buf1, MSG_OOB); // out-of-band 資料
    }

    // 檢查 s2 上的事件：
    if (ufds[1].revents & POLLIN) {
        recv(s2, buf2, sizeof buf2, 0);
    }
}
```

參考

select()

recv(), recvfrom()

接收 socket 的資料

函式原型

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

說明

一旦你設定好 socket 以及建立了連線，你就可以開始讀取遠端送來的資料（TCP SOCK_STREAM socket 使用 recv() 、UDP SOCK_DGRAM socket 使用 recvfrom()）。

這兩個函式都會用到 socket descriptor、指向 buf 緩衝區的指標、緩衝區的大小（單位是 byte）、以及要控制函式運作方式的 flags set。

此外，recvfrom() 會需要一個 struct sockaddr*，這可以讓你知道資料來自何處，並將 struct sockaddr 的長度填入 fromlen（你必須將 fromlen 初始化爲 from 或 struct sockaddr 的大小）。

所以你可以傳遞那些奇妙的 flags 呢？這裡列出一部分，但是你系統所支援的功能你應該要自行查看你的 man 使用手冊。你可以用 OR 位元運算來同時使用這些 flags，如果你只是想要使用一般香草口味的 recv()，那只要將 flags 設定爲 0 就可以了。

- MSG_OOB 接收 Out of Band（頻外）資料，這個要接收的資料是有設定 MSG_OOB flag 的 send() 所送來的。身爲接收端，會有 SIGURG 訊號來告訴你有緊急資料（urgent data）需要處理。所以你可以在這個訊號的處理常式（handler）中搭配 MSG_OOB flag 來呼叫 recv()。
- MSG_PEEK 如果你想要假裝呼叫 recv()（裝個樣子），你可以搭配這個 flag 來呼叫 flag，這個可以讓你知道緩衝區中有哪些資料存在，這個 flag 可以讓 recv() 先預覽緩衝區中的資料而沒有眞正的接收進來，下次沒有用 MSG_PEEK 的 recv() 才會眞的將這些資料收進來。
- MSG_WAITALL 你可以用這個 flag 告訴 recv()，資料長度沒有達到你在 len 參數指定的長度以前別想返回，雖然在特殊的情況下還是事與願違，比如遇到訊號中斷了 recv() call、發生了一些錯誤、或者遠端關閉連線之類的情況。這就別跟它計較了。

當你呼叫 recv() 時，它會 block，直到讀到了一些資料，如果你希望它不會 block，那麼可以將 socket 設定爲 non-blocking、或者是在呼叫 recv() 或 recvfrom() 以前，使用 select() 或 poll() 檢查 socket 上是否有資料。

傳回值

傳回實際接收的資料數（可能會比你在 len 參數中指定的還要少）、錯誤時傳回 -1（並設定相對應的 errno）。

若遠端關閉了連線，則 recv() 會傳回 0，這是一般用來判斷遠端是否關閉連線的方式。

範例

```
// stream sockets 與 recv()

struct addrinfo hints, *res;
int sockfd;
char buf[512];
int byte_count;

// 取得 host 資訊，建立 socket，並建立連線
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // 使用 IPv4 或 IPv6，兩者皆可
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
connect(sockfd, res->ai_addr, res->ai_addrlen);

// 好的！現在我們已經連線，我們可以開始接收資料！
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes of data in buf\n", byte_count);
// datagram sockets 與 recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

// 取得 host 資訊、建立 socket、將 socket bind 到 port 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // 使用 IPv4 或 IPv6，兩者皆可
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, "4950", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);

// 不需要使用 accept()，直接使用 recvfrom()：

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes of data in buf\n", byte_count);
printf("from IP address %s\n",
    inet_ntop(addr.ss_family,
        addr.ss_family == AF_INET?
            ((struct sockadd_in *)&addr)->sin_addr:
            ((struct sockadd_in6 *)&addr)->sin6_addr,
        ipstr, sizeof ipstr);
```

參考

send(), sendto(), select(), poll(), Blocking

select()

檢查 sockets descriptors 是否就緒可讀或可寫

函式原型

```
#include <sys/select.h>

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

說明

select() 函式讓你可以同步檢查多個 sockets，檢查它們是否有資料需要接收，或者是否你可以送出資料而不會發生 blocking，或者是否有例外發生。

你可以使用如上述的 FD_SET() macros（巨集）來調整 socket descriptors set。

如果你想要知道 set 上的哪些 sockets 有資料可以接收（就緒可讀），則將這個 set 放在 readfds 參數；

如果你想要知道 set 上的哪些 sockets 有資料可以傳送（就緒可寫），則將這個 set 放在 writefds 參數；

若你想知道哪些 sockets 有例外（錯誤）發生，則可以將 set 擺在 exceptfds 參數。

沒興趣知道的事件可以將 select() 相對應的參數設定爲 NULL。

在 select() 返回之後，這些 sets 的值會改變，用以標示哪些 sockets 是就緒可讀、可寫，及有例外發生等事件。

第一個參數 n 的值是最大的那個 socket descriptor 數值在加上 1。

最後，struct timeval 可以設定 timeout 的時間，用來告訴 select() 要等待多久的時間。select() 會在有任何事件發生或 timeout 之後返回。struct timeval 有兩個欄位：tv_sec 設定秒（second）、tv_usec 設定微秒（microsecond）〔一秒鐘有 1,000,000 微秒〕。

用到的 macros 功能如下：

- FD_SET(int fd, fd_set *set); 將 fd 新增至 set。
- FD_CLR(int fd, fd_set *set); 從 set 中移除 fd。
- FD_ISSET(int fd, fd_set *set); 若 fd 在 set 中則傳回 true。
- FD_ZERO(fd_set *set); 將 set 清爲零。

傳回值

成功時傳回 set 中的 descriptors 數量，若發生 timeout 時傳回 0，錯誤時傳回 -1（並設定相對應的 errno），還有，sets 會被改過，用以表示哪幾個 sockets 是已經就緒的。

範例

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// 假裝我們此時已經都連線到 server 了
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// 事先清除 set
FD_ZERO(&readfds);

// 將我們的 descriptors 新增到 set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// 因爲 s2 是後來才取得的，所以它的數值會”比較大”，所以我們用它作爲 select() 中的 n 參數

n = s2 + 1;

// 在 timeout 以前會一直等待，看是否已經有資料可以接收的 socket（timeout 時間是 10.5 秒）
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // select() 發生錯誤
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 10.5 seconds.\n");
} else {
    // 至少一個 descriptor(s) 有資料
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s2, buf2, sizeof buf2, 0);
    }
}
```

參考

poll()

setsockopt(), getsockopt()

爲 socket 設定多個選項

函式原型

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);
```

說明

這裡會講些基礎的 Sockets 設定。

很明顯的，這些函式是用來取得 socket 的資訊或設定 socket。在 Linux 系統上，全部的 socket 資訊都在 man 使用手冊第 7 節（輸入 "man 7 socket" 就可以查看這些資訊）。

至於參數，s 是想要設定的 socket， level 應該要設定爲 SOL_SOCKET，然後你可以將 optname 設定爲你有興趣的名稱，再來，完整的參數請參考你的 man 使用手冊，這邊只會介紹比較好玩的那一部分：

- SO_BINDTODEVICE 將這個 socket bind 到類似 eth0 名稱的 symbolic device，而不是用 bind() 將 socket bind 到一個 IP address，在 Unix 底下輸入 ifconfig 可以查看網路裝置名稱。
- SO_REUSEADDR 除非已經有 listening socket 綁定到這個 port，不然可以允許其它的 sockets bind() 這個 port。這樣可以讓你在 server 當機之後，重新啓動 server 時不會遇到 "Address already in use（位址已經在使用中）" 這個錯誤訊息。
- SO_BROADCAST 讓 UDP datagram (SOCK_DGRAM) sockets 可以傳送或接收封包到廣播位址（broadcast address）！ 至於 optval 參數，通常是個指向一個 int 型別變數的指標，若是 booleans，零爲 false（假），而非零爲 true（眞）。除非在你的系統不一樣，不然這是不爭的事實。如果沒有參數要傳遞，可以將 optval 設定爲 NULL。

最後一個 optlen 參數是你用 getsockopt() 取得的，而你必須在 setsockopt() 時指定它，大小可能是 sizeof(int)。

警告：在一些系統上（特別是 Sun 與 Windows），所要設定的選項會是個字元，而不是 int，比如：是一個 '1' 的字元值，而不是 1 的整數值。還有，細節請參考你 man 使用手冊，用 "man setsockopt" 以及 "man 7 socket"！

傳回值

成功時傳回零，或者錯誤時傳回 -1（並設定相對應的 errno）。

範例

```
int optval;
int optlen;
char *optval2;

// 表示對 socket 設定 SO_REUSEADDR(1) 是 true：
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// 將 socket 綁定（bind）到一個裝置名稱（可能不是每個系統都有效果）：
optval2 = "eth1"; // 4 bytes long, so 4, below:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);

// 檢測是否有設定 SO_BROADCAST flag：
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
    print("SO_BROADCAST enabled on s3!\n");
}
```

參考

fcntl()

send(), sendto()

透過 socket 送出資料

函式原型

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len,
               int flags, const struct sockaddr *to,
               socklen_t tolen);
```

說明

這些函式透過 socket 傳送資料，一般而言，send() 用在需連線的 TCP SOCK_STREAM socket，而 sendto() 用在免連線的 UDP SOCK_DGRAM socket。對於免連線 sockets，你每次都必須指定封包的目地，而這就是爲什麼 sendto() 的最後一個參數需要定義封包的目地。

在 send() 與 sendto() 中，s 參數是 socket、buf 是指向要傳送資料的指標、len 是你想要傳送的資料量、而 flags 讓你可以指定更多資訊，決定要如何傳送資料。如果你只是想要送一般的資料，可以將 flags 設定爲零。這裡有些常用的 flags，細節可以參考你電腦上的 send() man 使用手冊。

- MSG_OOB 以 "out of band" 送出，TCP 支援這項功能，而這個方法可以用來告訴接收端，有高優先權的資料需要接收。接收端會收到 SIGURG 訊號，並且可以先從佇列中接收這筆資料。
- MSG_DONTROUTE 不要透過 router 送出這筆資料，只將它留在區域網路內部。
- MSG_DONTWAIT 若 send() 會因爲外部流量堵塞而導致 block，讓它直接傳回 EAGAIN。這類似”只針對這次的 send() 啓用 non-blocking”，細節請參考 7.1 blocking 章節。
- MSG_NOSIGNAL 若你 send() 的遠端主機不再接收資料了，一般你會收到 SIGPIPE 訊號，新增這個 flag 可以避免觸發這個訊號。

傳回值

傳回實際送出的資料數量，錯誤時傳回 -1（並設定相對應的 errno）。注意，實際上送出的資料量有可能會少於你要求傳送的數量！請參考 7.3 不完整傳送的後續處理 章節。

還有，若對方的 socket 已經關閉，則 process 呼叫 send() 會產生 SIGPIPE 訊號（除非在呼叫 send() 時有搭配 MSG_NOSIGNAL flag，才不會收到）。

範例

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

// 先以 TCP stream sockets：

// 假設已建立 sockets 並連線
// stream_socket = socket(...
// connect(stream_socket, ...

// 轉換爲 network byte order
temp = htonl(spatula_count);
// 一般方式傳送資料：
send(stream_socket, &temp, sizeof temp, 0);

// 頻外方式傳送秘密訊息
send(stream_socket, secret_message, strlen(secret_message)+1, MSG_OOB);

// 現在用 UDP datagram sockets：
//getaddrinfo(...
//dest = ...  // 假設 "dest" 承載目的端的位址
//dgram_socket = socket(...

// 以一般方式傳送秘密訊息：
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
       (struct sockaddr*)&dest, sizeof dest);
```

參考

recv(), recvfrom()

停止對 socket 繼續傳送與接收

函式原型

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

說明

如果我不需要再對 socket 進行傳送〔send()〕，但是我仍然想要接收〔recv()〕socket 的資料，反之亦然，那我該怎麼做呢？

當你使用 close() 關閉 socket descriptor 時，它會將 socket 的傳送與接收兩端都關閉，並且釋放 socket descriptor。若你只想要關閉其中一端，你就可以使用 shutdown() call。

在這些參數中，s 顯然是你想要進行動作的 socket，而要進行什麼樣的動作，則要由 how 參數指定。要如何使用 SHUT_RD 來關閉接收，SHUT_WR 以關閉傳送，或者 SHUT_RDWR 將收送功能都關閉。

要注意 shutdown() 並沒有釋放 socket descriptor，所以即使 socket 已經整個 shutdown 了，最終仍然得透過 close() 關閉 socket。

這是蠻少用的 system call。

傳回值

成功時傳回零，或者錯誤時傳回 -1（並設定相對應的 errno）。

範例

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...在這裡進行一些 send() 與工作...

// 而現在我們已經完成該做的事情，不再需要傳送了：
shutdown(s, SHUT_WR);
```

參考

close()

socket()

配置一個 socket descriptor

函式原型

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

說明

傳回一個新的 socket descriptor，這通常是寫 socket 程式用到的第一個呼叫，而你可以將這個 socket descriptor 用在後續的 listen()、bind()、accept() 或各種其它的函式。

一般的用法是從呼叫 getaddrinfo() 取得的參數值，如同下列範例所示，但是若你有意願，你也可以手動填寫。

- domain domain 說明你有興趣的 socket 類型。相信我，這可以是很廣泛的東西，只是因爲我們這邊談的是 socket，所以這裡只會用到 IPv4 的 PF_INET 及 IPv6 的 PF_INET6。
- type 還有，type 參數可以是各種東西，不過你大概只會用到可靠 TCP socket 的 SOCK_STREAM（send(), recv()）或者不可靠快速 UDP socket 之 SOCK_DGRAM（sendto(), recvfrom()）〔另一個有趣的 socket type 是 SOCK_RAW，這個可以用來手動建構封包表頭，它超酷的！〕
- protocol 最後，protocol 參數指定在特定的 socket type 要用的 protocol（通訊協定），正如我所說的，比如：SOCK_STREAM 使用 TCP。你很好命，在使用 SOCK_STREAM 或 SOCK_DGRAM 時，你可以直接將 protocol 設定爲 0，這樣它會自動使用適合的 protocol，要不然你可以用 getprotobyname() 查詢適合的協定編號（protocol number）。

傳回值

傳回之後呼叫要用的新 socket descriptor，錯誤時傳回 -1（並設定相對應的 errno）

範例

```
struct addrinfo hints, *res;
int sockfd;

// 首先，使用 getaddrinfo() 載入位址結構：

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;     // AF_INET、AF_INET6 或 AF_UNSPEC
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM 或 SOCK_DGRAM

getaddrinfo("www.example.com", "3490", &hints, &res);

// 使用 getaddrinfo() 取得的資訊來建立 socket
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

參考

accept(), bind(), getaddrinfo(), listen()

struct sockaddr and pals

處理 internet addresses 的資料結構

函式原型

```
#include <netinet/in.h>

// 全部指向 socket address structures 的指標，通常會在各種函式或 system calls 使用以前，
// 轉型爲對相對應型別的指標：

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes 的 protocol address
};

// IPv4 AF_INET sockets:

struct sockaddr_in {
    short            sin_family;   // 例如：AF_INET, AF_INET6
    unsigned short   sin_port;     // 例如：htons(3490)
    struct in_addr   sin_addr;     // 參考下列的 struct in_addr
    char             sin_zero[8];  // 若你想要的話，將這個設定爲零
};

struct in_addr {
    unsigned long s_addr;          // 用 inet_pton() 載入
};


// IPv6 AF_INET6 sockets:

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow 資訊
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // 用 inet_pton() 載入
};


// 承載 socket address 的 structure，要足以承載
// struct sockaddr_in 或 struct sockaddr_in6 data：

struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // 這個全部都是填充的內容，依據實作而定，請忽略它：
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

說明

對於 internet address 全部的 syscalls 及函式都有基本的資料結構，通常你會用 getaddinfo() 填好這些資料結構，然後當你需要時，就可以讀取它們。

印象中，struct sockaddr_in 與 struct sockaddr_in6 會共用相同的 struct sockaddr 起始資料結構，而你能自由地將一種型別轉型爲另一種型別，除非你到了宇宙的盡頭，不然是不會有任何問題的。

宇宙的盡頭這件事只是開個玩笑 ... 若當你將 struct sockaddr_in 轉型爲 *struct sockaddr* 時已經到了宇宙的盡頭，那我保證這必定純屬巧合，所以你根本不用擔心這件事啦。

恩，還要記得一點，如果一個函式允許你代入一個 struct sockaddr 參數時，你就可以放心地將你的 *struct sockaddr_in*、struct sockaddr_in6 或 *struct sockadd_storage* 轉型成這個型別。

struct sockaddr_in 是 IPv4 addresses（例如："192.0.2.10"）在用的資料結構，它承載了一個 address family (AF_INET)：在 sin_port 有一個 port，而在 sin_addr 有一個 IPv4 address in sin_addr。

還有 struct sockadd_in 中的 sin_zero 欄位，有些人認爲這個一定要設定成零，但有些人認爲不用設定任何值（Linux 文件一點也沒有提過這件事），而且設定成零似乎沒有實際的用途。不過如果你喜歡，還是可以用 memset() 將它設定爲零。

目前 struct in_addr 在不同的系統上是個怪東西，有時候它是瘋狂的大總匯（crazy union），有各種 #define 與一堆鬼東西。不過你該做的只是用這個 structure 的 s_addr 欄位，因爲多數的系統只會實作這個欄位。

struct sockadd_in6 與 struct in6_addr 不僅都適用於 IPv6，而且都非常相似。

當你試著寫與 IP 版本無關的程式時，struct sockaddr_storage 是你可以傳遞給 accept() 或 recvfrom() 的資料結構，而且你不需知道新的 address 是走 IPv4 或 IPv6 協定。資料結構 struct sockaddr_storage 不像原本小小的 struct sockaddr，而是大到足以承載兩種型別。

範例

```
// IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);
// IPv6:

struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

參考

accept(), bind(), connect(), inet_aton(), inet_ntoa()

參考資料

你終於讀到這裡了，現在你可能正在大喊著：”還有嗎！”。

可以從哪裡獲得更多的這類資訊呢？

書籍

在一些歷史悠久的學校，你很容易就能借到紙本實體書，請試著閱讀下列推薦的這幾本好書。

- Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 0131411551 [43], 0130810819 [44].
- Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0131876716 [45], 0130319961 [46], 0130320714 [47].
- TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley.ISBNs for volumes 1, 2, and 3 (and a 3-volume set): 0201633469 [48], 020163354X [49], 0201634953 [50], (0201776316 [51]).
- TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 0596002971 [52].
- Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201433079 [53].

[43] http://beej.us/guide/url/unixnet1

[44] http://beej.us/guide/url/unixnet2

[45] http://beej.us/guide/url/intertcp1

[46] http://beej.us/guide/url/intertcp2

[47] http://beej.us/guide/url/intertcp3

[48] http://beej.us/guide/url/tcpi1

[49] http://beej.us/guide/url/tcpi2

[50] http://beej.us/guide/url/tcpi3

[51] http://beej.us/guide/url/tcpi123

[52] http://beej.us/guide/url/tcpna

[53] http://beej.us/guide/url/advunix

網站參考資料

在網路上：

- BSD Sockets: A Quick And Dirty Primer [54]（也是 Unix 系統程式設計資訊！）
- The Unix Socket FAQ [55]
- Intro to TCP/IP [56]
- TCP/IP FAQ [57]
- The Winsock FAQ [58]

[54] http://www.frostbytes.com/~jimf/papers/sockets/sockets.html

[55] http://www.developerweb.net/forum/forumdisplay.php?f=70

[56] http://pclt.cis.yale.edu/pclt/COMM/TCPIP.HTM

[57] http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/

[58] http://tangentsoft.net/wskfaq/

這裡是相關的維基百科：

- Berkeley Sockets [59]
- Internet Protocol (IP) [60]
- Transmission Control Protocol (TCP) [61]
- User Datagram Protocol (UDP) [62]
- Client-Server [63]
- Serialization [64]（封裝與解封裝資料）

[59] http://en.wikipedia.org/wiki/Berkeley_sockets

[60] http://en.wikipedia.org/wiki/Internet_Protocol

[61] http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[62] http://en.wikipedia.org/wiki/User_Datagram_Protocol

[63] http://en.wikipedia.org/wiki/Client-server

[64] http://en.wikipedia.org/wiki/Serialization

RFC

RFC [65] 眞的是非常原汁原味！這些文件說明了分配的數字、程式設計 API、及 Internet 上使用的通訊協定。

我已經在這裡幫你引用了一些 RFCs 的連結，所以你可以拿桶爆米花並開始你的學習之旅：

RFC 1 [66] — 第一篇 RFC；它會告訴你的事：比如什麼是 “Internet” 呢？因它融入生活，而深入了解它是如何設計的，這類的想法。〔顯然這份 RFC 已經過時了！〕

RFC 768 [67]—The User Datagram Protocol (UDP)

RFC 791 [68]—The Internet Protocol (IP)

RFC 793 [69]—The Transmission Control Protocol (TCP)

RFC 854 [70]—The Telnet Protocol

RFC 959 [71]—File Transfer Protocol (FTP)

RFC 1350 [72]—The Trivial File Transfer Protocol (TFTP)

RFC 1459 [73]—Internet Relay Chat Protocol (IRC)

RFC 1918 [74]—Address Allocation for Private Internets

RFC 2131 [75]—Dynamic Host Configuration Protocol (DHCP)

RFC 2616 [76]—Hypertext Transfer Protocol (HTTP)

RFC 2821 [77]—Simple Mail Transfer Protocol (SMTP)

RFC 3330 [78]—Special-Use IPv4 Addresses

RFC 3493 [79]—Basic Socket Interface Extensions for IPv6

RFC 3542 [80]—Advanced Sockets Application Program Interface (API) for IPv6

RFC 3849 [81]—IPv6 Address Prefix Reserved for Documentation

RFC 3920 [82]—Extensible Messaging and Presence Protocol (XMPP)

RFC 3977 [83]—Network News Transfer Protocol (NNTP)

RFC 4193 [84]—Unique Local IPv6 Unicast Addresses

RFC 4506 [85]—External Data Representation Standard (XDR)

IETF 有一個很棒的線上工具，可以搜尋與瀏覽 RFC [86]。

[65] http://www.rfc-editor.org/

[66] http://tools.ietf.org/html/rfc1

[67] http://tools.ietf.org/html/rfc768

[68] http://tools.ietf.org/html/rfc791

[69] http://tools.ietf.org/html/rfc793

[70] http://tools.ietf.org/html/rfc854

[71] http://tools.ietf.org/html/rfc959

[72] http://tools.ietf.org/html/rfc1350

[73] http://tools.ietf.org/html/rfc1459

[74] http://tools.ietf.org/html/rfc1918

[75] http://tools.ietf.org/html/rfc2131

[76] http://tools.ietf.org/html/rfc2616

[77] http://tools.ietf.org/html/rfc2821

[78] http://tools.ietf.org/html/rfc3330

[79] http://tools.ietf.org/html/rfc3493

[80] http://tools.ietf.org/html/rfc3542

[81] http://tools.ietf.org/html/rfc3849

[82] http://tools.ietf.org/html/rfc3920

[83] http://tools.ietf.org/html/rfc3977

[84] http://tools.ietf.org/html/rfc4193

[85] http://tools.ietf.org/html/rfc4506

[86] http://tools.ietf.org/rfc/

原著誌謝

感謝古往今來幫助過我寫完這份導讀的人。感謝 Ashley 幫我將封面設計做成最棒的程式設計師風格、感謝產出自由（free）軟體與套件的所有人，讓我可以做出這份導讀：GNU、Slackware、vim、Python、Inkscape、Apache FOP、Firefox、Red Hat 與許多其它軟體、最後超級感謝數以千計的你們所寫來的改進建議，以及文字上的鼓勵。

我將這份文件獻給我心目中電腦界的大師及鼓勵我的人：Donald Knuth、Bruce Schneier、W. Richard Stevens 與 The Woz、我的讀者、以及整個自由與開放原始碼軟體社群。

這本書是以 XML 撰寫，使用有安裝 GNU tools 的 Slackware Linux 系統與 vim 編輯器。

封面的”藝術”與圖片是使用 Inkscape 產生的，使用客制化的 Python scripts 將 XML 轉換爲 HTML 與 XSL-FO。接著用 Apache FOP 將 XSL-FO 輸出檔轉爲 Liberation 字型的 PDF 文件檔，Toolchain 100% 都使用自由與開放原始碼軟體。

譯者誌謝

感謝以下單位與人們的協助。

- 國立成功大學、電機系
- Tsung's Blog
- 程式人雜誌社團
- Penny140415（提供中文翻譯建議）