序

這本書的英文版近一年前在美國 Amazon 公司出版後，很快在美國，加拿大，德國，和日本都有售出。最近，又獲得美國最權威的書評機構 Bookauthority 的推薦，並評為最佳網路程式設計書籍。今將之譯成中文，以享國內讀者。

這本書含有我在美國電腦軟體工業界，橫跨好幾個領域，逾三十幾年的實際寶貴經驗的精華，是任何想成為世界頂尖軟體工程師或總工程師者，所必知的知識與必備的技能。深信讀者會終身受益。許多的留美電腦博士（大多在美國已工作很久） 買了與讀了之後都說，這本書的內容既廣泛又深入，他們從中學到了很多從其它書學不到的寶貴實際經驗，知識，與技能。

在計算機系統及網路程式設計上，看這本書就對了！但願這是一本您一輩子都想帶在身邊的書！

陳金追 謹上 2022 年 2 月

網路插口程式設計

12 CHAPTER

這一章討論網路插口（socket）的程式設計，介紹網路及分散式（distributed）程式設計的基本程式界面。這些是當今位處無數電腦網路，分散式與網際網路（web）應用程式之核心的技術。

當下有無數的軟體工程師都在開發網路式、分散式、與網際網路的程式，所有這些系統或應用軟體的核心與最底層，就是網路插口的程式界面（socket APIs）。

這一章所介紹的網路插口程式界面是 POSIX 標準所訂定的。因此，這一章所舉的程式例題，在所有支援 POSIX 標準的作業系統上，都能執行。這些程式也都曾在包括 Linux，各種的 Unix（IBM AIX，Oracle/Sun Solaris，HP HP-UX），以及使用達爾文作業系統（Darwin）的 Apple MacBook 上測試過。絕大部分的程式也在微軟的視窗系統上測試過。

有許多種不同的方式，可以從事網路與分散式的程式設計。亦即，分別在不同的層次上，使用不同的程式界面。不同的軟體工程師，也在不同的階層，使用不同的程式界面，開發各種不同的網際網路程式。不過，最常見的，就是利用離作業系統核心層最近的插口程式界面了。這些 C 語言的程式界面透過系統叫用，直接使用作業系統核心層所提供的各項網路服務。因此，它是最底層，直接叫用作業系統的核心層功能。所以，速度最快。

在插口界面往上一層，有所謂的**遠端程序叫用**（Remote Procedure Call，簡稱 RPC）。遠端程序叫用也是 C 語言的程式界面，這些界面等於將插口界

面外包一層，過濾掉一些細節，讓其更簡化容易使用，也稍較抽象一些。這些比插口界面較高一層。

再往上更高一層則有 REST，HTTP/HTTPS，SOAP 等程式界面。這些程式界面的內部實際作業，通常不是直接叫用插口界面，就是遠端程式叫用。

誠如上面所說的，插口程式界面是緊貼著作業系統的核心，離作業系統最近，直接叫用作業系統核心層內之網路單元所提供的各項服務。這些程式界面讓分別在經由電腦網路互相連接之兩部計算機上執行的兩個程式，能彼此相互通信。美妙的是，即使是兩個程式正好都在同一部電腦上執行，也毫無差別，結果完全一樣。這個插口程式界面所提供的，就是今日網際網路的骨幹與核心。它與在它更下層的作業系統網路單元以及將所有電腦連在一起的計算機網路硬體設備，共同形成了你我每天使用的網際網路。當然，這也要包括位在最上層用者每天直接使用的網際網路瀏覽器（web browser）應用程式在內。

如今，網際網路已到處都是，而且可以說已人人都有使用的經驗，甚至於到了一日沒有它，就等於沒吃飯的地步了。從軟體工程師的角度而言，網際網路基本上就是位於地球兩端，或甚至在同一房間內，經由電腦網路連接的兩部電腦上的兩個程式在通信對談。而這個通信對話，在離作業系統最近的層級上，就是兩個使用插口界面的程式在彼此通信。而插口界面在更下層所使用的，即是作業系統與電腦網路硬體所提供的 TCP 與 IP 網路協定（protocols）。整個網際網路，事實上就是一個利用 TCP 與 IP 網路協定在溝通運作的網路。

提供使用者網際網路經驗的這兩個主要程式，通常角度略有不同。使用者直接使用，離使用者最近的程式，這通常是瀏覽器，或某種網路應用程式，扮演**用戶**或**客戶**（client）的角色。它一般主動連接，啟動對談，並提出服務請求。收到這服務請求的遠方程式，通常扮演所謂**伺服器**（server）的角色。它通常座落於某公司電腦房內的某一伺服系統上，很有可能存在幾千里之外。為了滿足客戶的請求，這伺服器程式一般必須存取資料庫，取得客戶所要求的網頁或其他資料，再將之送回給客戶。換言之，這是一個典型的**客戶伺服**（client-server）通信。這兩個（實際上通常中間還有其他輔助這兩個程式搭

上線的程式）程式在執行它們的作業系統層次，就是透過使用 TCP/IP 電腦網路協定的網路插口程式界面在相互通信的。

因此，網路插口程式設計，事實上就是絕大多數電腦網路系統軟體與應用軟體，分散式系統，網際網路應用，以及更高層之網路協定（如 HTTP，HTTPS，REST）等的基礎與核心。是以，了解網路插口以及其程式設計，對於了解今天的網際網路如何運作，以及開發與維修網路式、分散式、或網際網路的系統與應用軟體，都會有無限的助益。

這一章介紹網路插口程式界面，討論網路插口如何動作，以及如何以這程式界面，開發各種網路式、分散式、及網際網路式的系統與應用程式。讀者會學到如何使用 TCP，UDP，以及 Unix 領域等不同插口，以之設計開發各種不同的網路通信軟體，以及如何設計履行包括客戶伺服通信，同步連接，非同步連接（asynchronous connect），多播作業（multicasting），多程線伺服器，動態查取插口之**端口號**（port number），以及其他不同作業的插口程式。

在唸完這一章後，讀者將能以最底層且效率最高的插口程式界面，設計各種不同的網路式及分散式系統軟體與應用軟體。

12-1 基本網路概念

12-1-1 七層模型

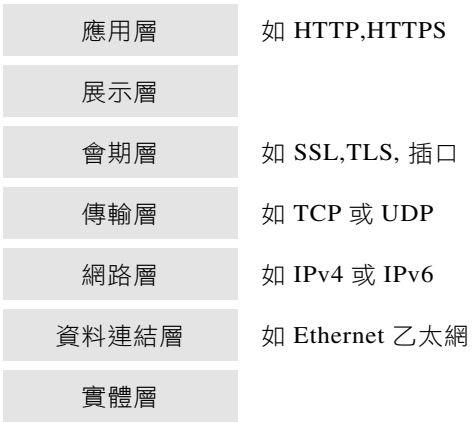

圖 12-1　電腦網路結構的 OSI 模型

作為軟體工程師，你會發現你得經常去參考或閱讀許多不同的 RFC 文件。以進一步了解協定的某些細節。網路一搜尋，通常很容易就能找到各 RFC 文件。

12-2 何謂插口

計算機網路插口的概念，源自於 1970 年代 ARPANET 網路。插口是一個程式能以之發送與接收資料的抽象概念。

每一**插口**在程式內代表一個**通信端點**（communication endpoint）。一個網路插口是一部計算機上一個通信端點的軟體象徵或代表。它能將資料送給另一插口，或接收來自另一插口的資料。

插口的特色與功能，是經由一組稱為插口程式界面的軟體函數，呈現給各種軟體程式。這套插口程式界面，讓軟體程式能利用位在作業系統核心內的網路傳輸層與網路層所提供的服務，與計算機網路上的另一個程式，彼此通信，互相交換資料。這另一個程式可在經電腦網路連接的另一部計算機上，亦可在同一部計算機上。

插口程式界面，實際是製作在作業系統的核心層內，是屬於會期層的服務。就在傳輸層之上，也正好在作業系統核心之上。最早的插口程式界面，出現在 1983 年的美國加州大學柏克萊校區所開發的 BSD UNIX 作業系統的 4.2 版本上。當初稱之為 BSD 插口程式界面，或柏克萊插口。

倘若在計算機 A 上執行的程式 P1，欲與在計算機 B 上執行的程式 P2，經由插口互相通信，則計算機 A 上就必須有一通信端點讓程式 P1 使用。同時，計算機 B 上也必須要有另一通信端點，讓程式 P2 使用。每一通信端點就是一個網路插口。換言之，每一插口即為一個能讓一個程式用以發送與/或接收資料的設施。

欲以插口與網路上其他的程式通信，程式就必須使用插口程式界面。插口程式界面是包括在 POSIX 標準內。這正是本章欲介紹給讀者的。

注意到，插口程式界面雖然主要用在分處兩部不同計算機上之兩個程式之間的通信，但若這兩個程式正好都位處同一部計算機，也完全可以，且完

全一樣，並無差別。由於這緣故，並且由於程式可經由插口傳送任何資料，插口幾乎已變成最重要且功能強大的程序間通信方式之一。

簡言之，插口是一個軟體特色，它讓分別在經由電腦網路或網際網路所連接的不同電腦上執行的程式，能彼此交換資料，相互通信。

12-2-1 不同類型的插口

根據其所使用的協定，可靠性以及特性的不同，插口有不同的種類。

首先，可分網際網路（Internet）插口與 Unix 領域（domain）插口。網際網路插口主要設計以用在，作為在網路所連接之不同計算機上執行之程式的程序間通信。不過，假若兩個程序都正好在同一部計算機上，也沒問題。網際網路插口使用網際網路協定組，TCP/IP 或 UDP/IP。這正是當今所有網際網路活動所使用的。

相對地，Unix 領域插口則是設計以用在同一部計算機內執行之程序間的通信用的，它無法超出一部計算機的領域。就某種意義而言，Unix 領域插口很類似前面幾章討論過的系統五 IPC。因為，它們都是給在同一系統上執行的程式，彼此之間互相通信用的。不過，這兩者還是有點差別。那就是，以 Unix領域插口互相通信的兩個程式，不需具有任何關係。

其次，依據其使用之網際網路協定不同而分，網際網路插口又進一步分成三種：**連播**（Stream）、**資料郵包**（Datagram）與**生料**（raw）。

這裡，我們簡短介紹這三種不同的網際網路插口以及 Unix 領域插口。

1. **連播插口**（Stream sockets）

   由於其下面所使用的傳輸層網路協定是連線式的 TCP 協定，因此，連播插口是連線式（connection-oriented）的插口。在連線建立時，TCP 協定會預留連線所需的資源，因此，可靠性獲得保證。使用這種插口相互通信，連線一旦建立後，程序即可一直連續不斷地一直發送資料給對方，相當於持續播放（連著播放），幾無限制。稱之為連播插口，原因在此。

   在可靠性是必要條件時，連播插口是最正確的選擇。

2. **資料郵包插口**

資料郵包（Datagram），或簡稱郵包，插口是不連線（connection-less）的插口，因為，它下面所使用的傳輸層協定，是不連線的 UDP。UDP 基本上是不事先建立連線的。是以，沿途所需用到的所有資源，也未事先保留。也因此，它無法保證絕對可靠。在系統資源不足的情況下，資料郵包就有可能因被丟包而遺失。

不過，資料郵包插口簡單，容易使用，也不像連播插口消耗那麼多系統的資源。

3. **生料插口**

生料（raw）插口也叫生料 IP 插口。它是在諸如尋路器（router）等之網路設備上使用的。生料插口略過傳輸層，資料包的前頭資料（headers）直接送給應用程式。

注意到，POSIX 標準並未要求生料插口一定要有，它是可有可無的。以上三種是網際網路插口。

4. **Unix領域插口**

Unix 領域插口有時又叫 IPC 插口。它是設計以用在同一系統上之不同程序間的通信用的。由於這緣故，Unix 領域插口並非以 IP 位址識別，而是以檔案系統中的檔案作識別。

Unix 領域插口的好處是，它的用法與網際網路插口幾乎一模一樣，唯一的差別是，它是跟一路徑名綁（bind）在一起，而非一 IP 位址。

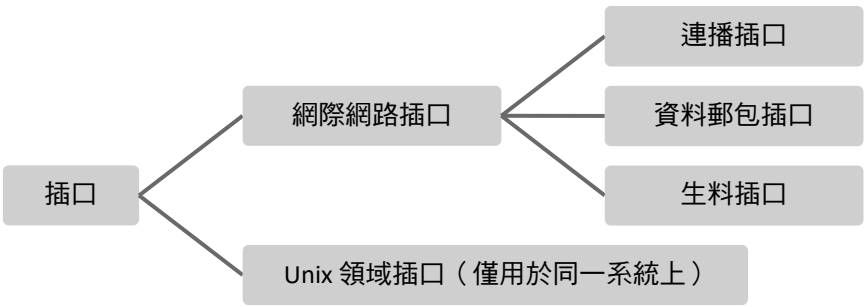

圖 12-4　不同類型的插口

12-2-2 socket() 函數

欲以插口和其他程式通信，不論這另一個程式是在同一部計算機上，或在網路上的另一部計算機，程式所需的第一件東西，就是一個網路插口。每一網路插口即為一通信端點。兩個程式，只要各自擁有一個插口，即能以插口相互通信。這一節討論程式如何產生插口。

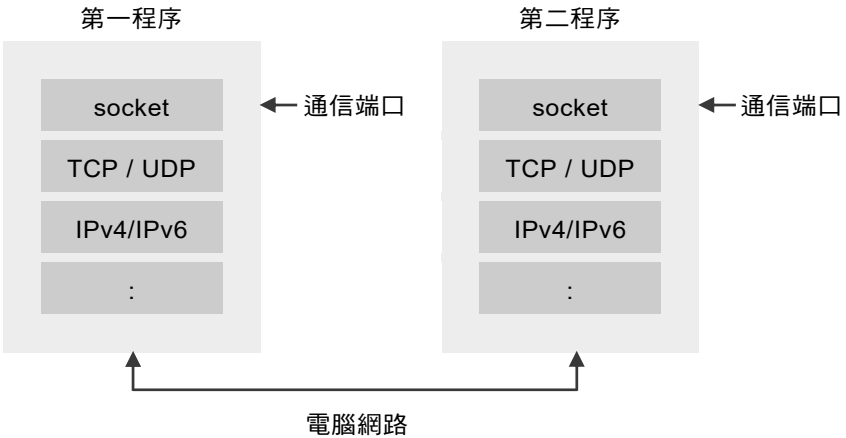

圖 12-5 使用網際網路插口的網路通信

socket() 函數叫用產生一個插口。它產生一個通信端點。這函數叫用會送回一個代表著這已打開之插口的檔案描述。你知道，在 Unix/Linux 作業系統，很多東西都被程式當成檔案看待，這包括網路插口在內。

socket() 函數的格式如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

socket() 函數的第一個參數指出一個**領域**（domain），稱作**位址族系**（Address Family，AF）或**協定族系**（Protocol Family，PF）。POSIX 標準所支援的三個主要領域定義在 <sys/socket.h> 前頭檔案上，其符號常數名稱如下：

AF_INET（PF_INET）— 網際網路領域的插口（IPv4 位址用的）
AF_INET6（PF_INET6）— 網際網路領域的插口（IPv6 位址用的）
AF_UNIX（PF_UNIX）— UNIX 領域插口

為了避免混淆，我們將分別稱之為網際網路插口與 UNIX 領域插口。

注意到，AF_INET 與 PF_INET 是一樣的，AF_INET6 與 PF_INET6 一樣，且 AF_UNIX 與 PF_UNIX 一樣。舉例而言，在某些 UNIX 系統上，以下就是 /usr/include/sys/socket.h 對這些符號常數的定義：

```
#define AF_UNIX         1               /* local to host (pipes, portals) */
#define AF_INET         2               /* internetwork: UDP, TCP, etc. */
#define AF_INET6        26              /* Internet Protocol, Version 6 */

#define PF_UNIX         AF_UNIX
#define PF_INET         AF_INET
#define PF_INET6        AF_INET6
```

而下面則是你在 Linux 系統 /usr/include/bits/socket.h 上所看到的：

```
#define PF_LOCAL        1       /* Local to host (pipes and file-domain).  */
#define PF_UNIX         PF_LOCAL /* POSIX name for PF_LOCAL.  */
#define PF_INET         2       /* IP protocol family.  */
#define PF_INET6        10      /* IP version 6.  */

#define AF_UNIX         PF_UNIX
#define AF_INET         PF_INET
#define AF_INET6        PF_INET6
```

注意到，AF_INET6/PF_INET6 的實際號碼在 Unix 和 Linux 上並不相同。

這無所謂，因為這數目只有在同一作業系統之內使用。同時，這也說明為何你在程式內要記得永遠只使用諸如 AF_INET6 或 PF_INET6 之符號常數。因為，這樣你的程式才有移植性。不論放在那一作業系統上編譯，都不必修改。

假若程式想產生並使用的是 Unix 領域插口，即在 socket() 函數的第一個引數，你就得指明 AF_UNIX 或 PF_UNIX。若是想產生與使用網際網路插口，那該第一引數的值即應為 AF_INET 或 PF_INET，或 AF_INET6 或 PF_INET6，視你是使用 IPv4 或 IPv6 的插口而定。

Unix 領域插口用在同一系統上不同程序間的通信。網際網路插口則用在不同系統間之程序的通信，但即使是同一系統也行。就網際網路插口而言，兩通信的程序是在不同系統或同一系統，並無差別，一樣用法。

socket() 函數的第二個引數指出插口的類型。POSIX 標準所支援的插口類別也是定義在 <sys/socket.h> 上，其符號常數如下：

SOCK_STREAM：連播插口

SOCK_DGRAM：資料郵包插口

SOCK_RAW：生料插口。這類型可有可無。POSIX 標準並未硬性規定一定要有。

SOCK_SEQPACKET：排序資料包插口

倘若程式想使用的是可靠的連線式插口，那 socket() 函數叫用的第二個引數值，即應指明 SOCK_STREAM。若是非連線式的插口，則這引數的值即應是 SOCK_DGRAM。若是生料插口，則這引數的值即是 SOCK_RAW。記得，產生生料插口通常是要有超級用戶的權限。SOCK_SEQPACKET 的插口在資料郵包的最大長度固定的情況下，為資料郵包提供一個循序且可靠的雙向連線式通信管道。在資料接收端，接收者每次讀取一定要讀取一整個郵包。

記得，並非每一協定族系都支援所有的插口類型的。譬如，AF_INET 位址族系即不支援 SOCK_SEQPACKET 插口。所以，假若你在領域參數傳入的值是 AF_INET，那插口型態的值即不能是 SOCK_SEQPACKET。不過，AF_UNIX 協定族系就有支援這個插口類別。

socket() 函數的第三個參數指出一種協定。這個參數指出插口想用的網路協定。在絕大多數情況下，每一協定族系就只有支援一種協定，因此，這個引數的值經常是 0。只有在少數情況下，協定族系支援多種網路協定時，這個參數才會用到。

圖 12-5 所示即兩個程序透過插口相互通信的情形。每一程序使用一個插口，藉以發送與接收資料。插口利用底下傳輸層與網路層的網路協定，達成與對手插口的資料交換。

圖 12-6 所列即為一展示如何使用 socket() 函數的程式。它分別測試各種不同的協定族系，插口類型與網路協定組合，在你使用的系統上是否有支援。記得我們提過，使用 SOCK_RAW 類型的插口必須有超級用戶的權限。因此，假若你是以一般用戶執行這個例題程式，在試圖產生一生料插口時，作業會得到錯誤 EPERM（沒有權限）。

圖 12-6 找出那些插口類別這系統有支援（socket.c）

```
/*
 * socket()
 * Different combinations of socket types and protocols that are supported.
 * Authored by Mr. Jin-Jwei Chen.
 * Copyright (c) 1993-2016, Mr. Jin-Jwei Chen. All rights reserved.
 */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>    /* protocols such as IPPROTO_TCP, ... */
#include <unistd.h>

int main(int argc, char *argv[])
{
  int    sockfd;

  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd == -1)
    fprintf(stderr, "socket(AF_INET, SOCK_STREAM, 0) failed,
errno=%d\n", errno);
  else
  {
    fprintf(stdout, "socket(AF_INET, SOCK_STREAM, 0) is supported\n");
    close(sockfd);
  }

  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sockfd == -1)
    fprintf(stderr, "socket(AF_INET, SOCK_DGRAM, 0) failed, errno=%d\n", errno);
  else
  {
    fprintf(stdout, "socket(AF_INET, SOCK_DGRAM, 0) is supported\n");
    close(sockfd);
  }

  sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (sockfd == -1)
    fprintf(stderr, "socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) failed,
errno=%d\n", errno);
  else
  {
    fprintf(stdout, "socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) is supported\n");
    close(sockfd);
  }

  sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (sockfd == -1)
```

```
        fprintf(stderr, "socket(AF INET, SOCK DGRAM, IPPROTO UDP) failed,
errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP) is supported\n");
        close(sockfd);
      }

      sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sockfd == -1)
        fprintf(stderr, "socket(AF_UNIX, SOCK_STREAM, 0) failed, errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_UNIX, SOCK_STREAM, 0) is supported\n");
        close(sockfd);
      }

      sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sockfd == -1)
        fprintf(stderr, "socket(AF_UNIX, SOCK_DGRAM, 0) failed, errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_UNIX, SOCK_DGRAM, 0) is supported\n");
        close(sockfd);
      }

      sockfd = socket(AF_UNIX, SOCK_SEQPACKET, 0);
      if (sockfd == -1)
        fprintf(stderr, "socket(AF_UNIX, SOCK_SEQPACKET, 0) failed,
errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_UNIX, SOCK_SEQPACKET, 0) is supported\n");
        close(sockfd);
      }

      sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
      if (sockfd == -1)
        fprintf(stderr, "socket(AF_INET, SOCK_RAW, IPPROTO_RAW) failed,
errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_INET, SOCK_RAW, IPPROTO_RAW) is supported\n");
        close(sockfd);
      }

      sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
      if (sockfd == -1)
```

```
        fprintf(stderr, "socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) failed,
errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) is supported\n");
        close(sockfd);
      }

      sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_EGP);
      if (sockfd == -1)
        fprintf(stderr, "socket(AF_INET, SOCK_RAW, IPPROTO_EGP) failed,
errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_INET, SOCK_RAW, IPPROTO_EGP) is supported\n");
        close(sockfd);
      }

      sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RSVP);
      if (sockfd == -1)
        fprintf(stderr, "socket(AF_INET, SOCK_RAW, IPPROTO_RSVP) failed,
errno=%d\n", errno);
      else
      {
        fprintf(stdout, "socket(AF_INET, SOCK_RAW, IPPROTO_RSVP) is supported\n");
        close(sockfd);
      }

      return(0);
    }
```

從執行這例題程式，你可發現下列的插口類型與協定的組合，通常是有支援的：

```
    socket(AF_INET, SOCK_STREAM, 0)
    socket(AF_INET, SOCK_DGRAM, 0)
    socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
    socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
    socket(AF_UNIX, SOCK_STREAM, 0)
    socket(AF_UNIX, SOCK_DGRAM, 0)
    socket(AF_INET, SOCK_RAW, IPPROTO_RAW)
    socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)
    socket(AF_INET, SOCK_RAW, IPPROTO_EGP)
    socket(AF_INET, SOCK_RAW, IPPROTO_RSVP)
```

記得，SOCK_RAW 類型的插口（上述最後四種組合），需要超級用戶的權限。

12-13 常見的插口函數錯誤與解決之道

這一節介紹在執行使用插口的程式時，一般常見的錯誤，其意義如何，以及解決之道。對初學者或甚至是有經驗的軟體工程師而言，這些插口錯誤有時很難知道它真正的問題是什麼。因此，這一節的討論應會有很大的助益。

插口的錯誤通常定義在 errno.h 內。讀者應使用這些錯誤的符號名（如 EADDRINUSE），而非錯誤號碼。因為，錯誤符號名一般在不同平台都是一樣的，但其錯誤號碼則是每一平台幾乎都不一樣的。以下例子所顯示的錯誤號碼是在 Linux 系統上用的，其他系統可能使用不同的數目。

1. Error: bind() failed, errno=98

```
#define EADDRINUSE      98      /* Address already in use */
```

錯誤 EADDRINUSE 或錯誤訊息”Address already in use”（位址已經有其他程式在用），代表程式想繫綁的端口號，已經有其他的程式在使用了。例如，

```
$ mytcpsrv
Error: bind() failed, errno=98
```

一個程式通常在兩種情況下會有這個錯誤。首先，這個錯誤很可能是這個端口號真正並沒有程序正在使用，但前面使用這個端口號的程序，它雖已結束了，但它所使用的插口卻還處於 TIME_WAIT 的狀態（這點我們在下一章，13-5 一節會討論）。因此，還不能馬上使用。這個問題的解決之道是，在你的程式中，打開（turn on）SO_REUSEADDR 插口選項，讓系統能重新使用還等在這狀態的端口號。注意，打開一插口的這個選項，必須在 bind() 函數叫用之前為之，否則，可能會無效。更詳細的細節請進一步參考 13-5 與 13-6。

另一種情況是，這同一個端口號是真正有另一個程序正在使用它。由於每一端口號，每一瞬間最多只能有一個程序在使用。因此，解決之道是看你的程式是否有保留這個端口號。若無，那你只能改用另一個沒人在用的端口號。若有，那你就得找出到底那個程式，誤用了你所保留的端口號，將之要回。偵查時可以使用像 lsof 與 netstat 等工具幫忙。

2. Error: bind() failed, errno=99

```
#define EADDRNOTAVAIL   99      /* Cannot assign requested address */
```

錯誤 EADDRNOTAVAIL 代表 bind() 函數叫用所使用的 IP 位址是錯的，它不是現在這部計算機的 IP 位址，例如：

```
$ mytcpsrv
Connection-oriented server program ...
Error: bind() failed, errno=99

$  ./multicast_snd
Error: setsockopt(IP_MULTICAST_IF) failed, errno=99

IP address specified is not the local host's IP address.
```

解決之道是檢查程式所使用之 IP 位址是否正確。並改使用正確的 IP 位址。

3. Error: bind() failed, errno=13

```
#define EACCES          13      /* Permission denied */
```

錯誤 EACCESS（沒有權限使用）（在 Linux 的值是 13），一般表示程式試圖使用小於 1024 的端口號。使用這些端口號是要有超級用戶的權限的。例如：

```
$ tcpsrv_async_io_all 1 5
Connection-oriented server program ...
Error: bind() failed, errno=13
```

解決之道：換成以超級用戶執行這程式或改成使用 1024 以上的端口號。

4. Client connect error: Error: connect() failed, errno=101

```
#define ENETUNREACH     101     /* Network is unreachable */
```

在客戶程式試圖與伺服程式連線時，獲得 ENETUNREACH（網路不可及）（在 Linux 的值是 101）的錯誤，一般代表客戶所使用的伺服器的 IP 位址是錯的。

```
$ tcpclnt 2345 ::2
Connection-oriented client, connect to ::2, port 2345
Error: connect() failed, errno=101
```

解決之道：檢查看看程式所使用的伺服位址是否正確。試著以 ping 命令檢查看看你所使用的 IP 位址是否有回應。

5. Client connect error: Error: connect() failed, errno=111

```
#define ECONNREFUSED    111     /* Connection refused */
```

在一客戶程式試圖與伺服器連線時，若得到 ECONNREFUSED（連線請求被拒）（在 Linux，這個錯誤號碼是 111），那通常代表伺服程式並未啟動。它也有可能是客戶程式使用了錯誤的伺服器 IP 位址或端口號。

```
$ tcpclnt 2345 ::1
Connection-oriented client, connect to ::1, port 2345
Error: connect() failed, errno=111
```

解決之道：確認程式想與之通信的伺服程式有正在執行。檢查客戶程式是否使用了正確的伺服 IP 位址與端口號。

6. Client connect error: Error: connect() failed, errno=113

```
#define EHOSTUNREACH    113     /* No route to host */
```

在一試圖與一伺服程式連線的客戶程式，獲得 EHOSTUNREACH（主機不可及）（Linux 的錯誤號碼是 113）的錯誤時，這通常代表伺服程式所在的計算機並不存在或關機了，或主機名或 IP 位址弄錯了。

```
$ ./tcpclnt_all 2345 xyz
Connection-oriented client program ...
Error: connect() failed, errno=113, No route to host
```

解決之道：確認伺服器所在的主機是否真的關機了。

7. Client recv() error: Error: recv() failed, errno=110

```
#define ETIMEDOUT       110     /* Connection timed out */
```

倘若一客戶程式等在 recv() 函數叫用，等著接收伺服程式所送來的信息時，有人把網路的電纜線拔掉了。這種情況下，等那最長等待時間過後，這函數叫用即會錯誤回返，送回 ETIMEOUT（時間到）（在 Linux 的錯誤號碼是 110）的錯誤。

```
$ ./tcpclnt_all 2345 mysrv
  Error: recv() failed, errno=110
```

解決之道：將電腦網路恢復正常。

12-14 同一計算機內的通信 — Unix 領域插口

計算機網路通信，尤其是發生在網際網路上的，典型上都發生在以電腦網路相連的兩部不同計算機上。在這類型的通信上，兩個分別在兩部不同的計算機上執行的程式，彼此交換訊息。訊息由發送的計算機，由發送程式，經由作業系統內的所有網路層，經由其網路界面卡，經過電腦網路的電纜線，在中間有可能經過尋路器（router），然後抵達目的計算機。然後再行經目的計算機的資料連結層，網路層，傳輸層，最後送達接收程式。

這一章前面幾節討論的，幾乎都是這種利用網際網路插口所做的計算機與計算機之間的通信。當然，萬一這兩個互相通信的程式，正好都座落在同一部計算機上，也是一樣沒問題。

這一節，我們要介紹第三類型的插口。這種插口僅適用於在同一部計算機上的兩個程序之間的通信，那就是 **Unix 領域插口**（Unix domain sockets）。

你或許要問，為何需要使用 Unix 領域插口？

原因是，同一部計算機上的兩個程序通信，根本不需經過計算機網路。亦即，這些通信的信息，根本不需經過作業系統內之計算機網路的各個階層。只要利用記憶器或甚至檔案系統就可以。因為，這樣同一部計算機上的所有程序都能存取得到。Unix 領域插口，就是使用檔案系統的同一系統之程序間通信方式。（還記得第十章所介紹的共有記憶，就是使用記憶器的程序間通信方式嗎？）

對程式設計者而言，Unix 領域插口就是一種類似在傳輸層使用 TCP 協定的連播插口，或在傳輸層使用 UDP 協定的郵包插口。使用 Unix 領域插口的客戶與伺服程式，其架構與使用網際網路插口的程式幾乎完全一樣，唯一的差別如下：

1. 使用 Unix 領域插口的程式，必須包含下面的前頭檔案：

```
#include <sys/un.h>
```

2. 使用 Unix 領域插口時，插口的位址族系是 AF_UNIX。

3. 一 Unix 領域插口的位址是一檔案路徑名，而非 IP 位址加一端口號。

   亦即，不像在網際網路插口時，每一插口以一獨一無二的 IP 位址與端口號的組合加以辨認，每一 Unix 領域插口以一獨特的檔案系統路徑名加以辨別。這意謂伺服插口有其自己獨一無二的路徑名，而客戶插口也有自己獨特的檔案路徑名。

   注意到，用以辨認一 Unix 領域插口之檔案路徑名的最大長度，如下所示地，已定死在 <sys/un.h> 前頭檔案內的“struct sockaddr_un”資料結構裡。這個最大長度是 108 位元組。要是程式使用的檔案路徑名長度超過 108 個文字，程式即會出錯。而這個錯誤幾乎是最常見的 Unix 領域插口錯誤。

```
struct sockaddr_un
{
   __SOCKADDR_COMMON (sun_);
   char sun_path[108];          /* 路徑名  */
};
```

   作者見過有一家公司的軟體，大量地使用 Unix 領域插口作為同一系統上的客戶與伺服程式間的通信方式，這軟體又使用很長的檔案路徑名，因此，經常出現這路徑名太長（超過 108 個文字）的錯誤。所以，記得在使用 Unix 領域插口時，用以辨別這類插口的檔案路徑名一定要儘量短，總長度絕對不能超過 108 個文字或位元組。

4. 使用 Unix 領域插口時，客戶與伺服程式兩者都必須叫用 bind() 函數，將其插口繫綁在其自己獨有的檔案路徑名上。不像在使用網際網路插口的應用上，只有伺服程式必須叫用 bind()。

就像網際網路插口一樣，Unix 領域插口也是同時支援連線式與非連線式通信的。舉例而言，欲以 Unix 領域插口進行非連線式通信時，在 socket() 函數叫用上，第一引數（位址族系）的值是 AF_UNIX，而第二引數（插口類型）則是 SOCK_DGRAM。

就伺服程式而言，不必再擔心究竟應該使用那一個端口號的問題了。一切所需要的，即是選用一獨一無二的檔案路徑名，然後將伺服插口與之綁在一起就是了。此一獨特的檔案路徑名，其功用就與 IP 位址與端口號兩者的組合類似。不要忘了，客戶插口也必須與其自己獨特的檔案路徑名綁在一起。

圖 12-17 所示即為使用 Unix 領域插口相互通信的一對客戶與伺服程式。

圖 12-17a　以 Unix 領域插口通信的伺服程式（udssrv_all.c）

```
/*
 * A connection-oriented server program using Unix domain stream socket.
 * Support of multiple platforms including Linux, Solaris, AIX, HPUX, Windows
 * and Apple Darwin.
 * Usage: udssrv_all
 * Authored by Mr. Jin-Jwei Chen.
 * Copyright (c) 1993-2018, 2020 Mr. Jin-Jwei Chen. All rights reserved.
 */

#include "mysocket.h"

int main(int argc, char *argv[])
{
  int    ret;                    /* return code */
  int    sfd;                    /* file descriptor of the listener socket */
  int    newsock;                /* file descriptor of client data socket */
  struct sockaddr_un    srvaddr;   /* server socket structure */
  int    srvaddrsz = sizeof(struct sockaddr_un);
  struct sockaddr_un    clntaddr;  /* client socket structure */
  socklen_t    clntaddrsz = sizeof(struct sockaddr_un);
  char   inbuf[BUFLEN];          /* input message buffer */
  char   outbuf[BUFLEN];         /* output message buffer */
  size_t msglen;                 /* length of reply message */
  unsigned int  msgcnt;          /* message count */

#if WINDOWS
  WSADATA wsaData;                  /* Winsock data */
  char* GetErrorMsg(int ErrorCode); /* print error string in Windows */
#endif

  fprintf(stdout, "Connection-oriented server program using Unix Domain "
    "socket...\n");

#if WINDOWS
  /* Initiate use of the Winsock DLL. Ask for Winsock version 2.2 at least. */
  if ((ret = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
  {
    fprintf(stderr, "WSAStartup() failed with error %d: %s\n",
      ret, GetErrorMsg(ret));
    return (-1);
  }
#endif

  /* Create the Stream server socket. */
  if ((sfd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
  {
    fprintf(stderr, "Error: socket() failed, errno=%d, %s\n", ERRNO, ERRNOSTR);
```

```
#if WINDOWS
    WSACleanup();
#endif
    return(-2);
  }

  /* Fill in the server socket address. */
  memset((void *)&srvaddr, 0, (size t)srvaddrsz); /* clear the address buffer */
  srvaddr.sun family = AF UNIX;
  strcpy(srvaddr.sun path, SERVER PATH);

  /* Bind the server socket to its address. */
  unlink(SERVER PATH);
  if ((ret = bind(sfd, (struct sockaddr *)&srvaddr, srvaddrsz)) != 0)
  {
    fprintf(stderr, "Error: bind() failed, errno=%d, %s\n", ERRNO, ERRNOSTR);
    CLOSE(sfd);
    return(-3);
  }

  /* Set maximum connection request queue length that we can fall behind. */
  if (listen(sfd, BACKLOG) == -1) {
    fprintf(stderr, "Error: listen() failed, errno=%d, %s\n", ERRNO, ERRNOSTR);
    CLOSE(sfd);
    return(-4);
  }

  /* Wait for incoming connection requests from clients and service them. */
  while (1) {

    fprintf(stdout, "\nListening for client connect request ...\n");
    newsock = accept(sfd, (struct sockaddr *)&clntaddr, &clntaddrsz);
    if (newsock < 0)
    {
      fprintf(stderr, "Error: accept() failed, errno=%d, %s\n", ERRNO, ERRNOSTR);
      CLOSE(sfd);
      return(-5);
    }

    fprintf(stdout, "Client Connected. Client file path=%s\n",
      clntaddr.sun_path);

    msgcnt = 1;
    /* Receive and service requests from the current client. */
    while (1)
    {
      /* Receive a request from a client. */
      errno = 0;
      inbuf[0] = '\0';
      ret = recv(newsock, inbuf, BUFLEN, 0);
      if (ret > 0)
```

問題

1. 網路插口是什麼？它是一軟體或硬體元件呢？一個網路通信需要幾個插口？
2. 在 ISO 的七層網路結構裡，插口位於那一層？插口階層的緊接下一個階層是什麼？而插口階層的下下階層叫什麼？
3. 網際網路瀏覽器（browser）以那一種網路協定和網際網路上之伺服器互相溝通呢？那網路協定位於網路結構的那一階層？
4. 網路插口有多少種類？請簡短描述每一種類。
5. 插口位址可以以那些資料結構表示呢？
6. 一個程式如何產生一網路插口？
7. 就連線與否而言，網際網路上的軟體程式，有那兩種通信方式呢？就資源的騰出與可靠性而言，這兩種通信方式有何差別？
8. 欲從一個無連線的插口發送出一個信息，可以使用那些函數呢？若換成是一連線式的插口，在送出信息時，可以叫用那些函數呢？
9. 欲從一非連線式的插口接收信息，可以叫用那些函數呢？欲從一連線式插口接收信息，又可以叫用那些函數呢？
10. INADDR_ANY 是什麼？它的 IPv6 對應版本又叫什麼？
11. htonl() 與 ntohl() 是什麼？為何網路程式需要用這些？
12. 一個插口位址包含那些資訊？
13. getaddrinfo() 函數是做什麼用的？與它正好相反的函數叫什麼？
14. 什麼樣的伺服插口能同時接受 IPv4 與 IPv6 的客戶連線請求？
15. 那一類型的插口只能用在同一系統之程序間的通信？它以什麼分辨不同的插口？那又有何限制？
16. 程式使用什麼函數，支援非同步式的插口輸入/輸出作業？
17. 一程式如何將一同步式的插口轉換成一非同步式(即非阻擋式) 插口？
18. 什麼是多播？程式發送多播信息的步驟有那些？程式接收多播信息的步驟有那些？

19. 若欲讓同一計算機上有多個程序能同時接收同樣的多播信息，程式必須做什麼？
20. 多程序與多程線的伺服程式，各有何優、缺點？
21. 為何程式會獲得“位址已有人在使用”的錯誤？有那些方式可以避免獲得此一錯誤？
22. 什麼是有特權的端口號？它們的號碼有那些？
23. 預留的端口號放在那裡？程式如何在執行時，當場查詢一伺服程式所保留的端口號？
24. 伺服程式使用預留的固定端口號有何優點？

✎ 程式設計習題

1. 修改例題程式 udpclnt_conn_all.c，以致其叫用 connect() 函數兩次，分別送出信息給兩個不同的伺服程式。測試時，你可以分別使用不同的端口號，啟動 udpsrv_all 程式兩次。
2. 執行單程線的連線式伺服程式 tcpsrv，並同時執行兩個或更多個客戶程式，以證實它真的是單程線，每次只能處理一個客戶。
3. 修改 tcpsrvp.c，以致其預先產生幾個工作程序。每次有一客戶到來，即將之交給其中一個空閒著的工作程序去服務那個客戶。而在服務完後，就把工作程序再放回，勿讓它結束。
4. 修改 tcpsrvt.c，以致其預先產生好幾個工作程線。每次有一客戶程式來時，就叫用一個空閒著的工作程線去服務那個客戶。而服務完之後，工作程線就再放回，不要讓它結束。
5. 修改 tcpclnt1.c 與 tcpsrv1.c 例題程式，將 send() 改成 write()，且將 recv() 改成 read()。
6. 編譯且執行多播的例題程式，並測試兩種情況。第一種情況是兩個接收程式分別在不同計算機上執行。另一種情況則是兩個接收程式在同一計算機上執行。
7. 寫一個使用多個工作程線的多程線伺服程式。工作程線的數量由外部輸入，能讓用者指定。程式應事先產生這些工作程線，且重複使用它們。

8. 寫一個使用多個聆聽程線的多程線伺服程式，將接受客戶連線請求的工作，分佈至各個不同聆聽程線上。以同時產生上百個連線請求測試你的程式。

9. 寫一對能示範如何使用 SO_RCVLOWAT 插口選項的程式。在接收程式上打開 SO_RCVLOWAT 插口選項。信息發送程式則發送兩個信息，並在中間停留個幾秒鐘。發送程式所送出的第一個信息，其大小應該小於 SO_RCVLOWAT 所設定的值。

10. 寫一個程式，印出你所使用的系統上，一個連播插口之既定送出緩衝器與接收緩衝器的大小（以位元組數計）。緊接程式應分別將這兩個值各增加 4096，再查詢與印出新設定的值。

11. 寫一個程式，印出你所使用的系統上，一個資料郵包插口之既定發送緩衝器與接收緩衝器的大小。程式緊接應將這兩個值分別提高 4096。最後再讀取這兩個新設定的值，將之印出。

12. 寫兩個能做檔案傳輸與接收的程式。程式應使用連播（SOCK_STREAM）插口，能將一個檔案由一部計算機傳至另一部計算機。以不同大小的檔案（如 1MB，50MB，與 250MB）測試這對程式。記下每一不同檔案所需耗費的時間。藉著改變發送與接收緩衝器的大小，找出什麼樣的緩衝器大小，讓檔案傳輸的速度最快。

13. 重複上一習題的檔案傳輸程式，但改用資料郵包插口。

參考資料

1. Computer Networks, Andrew S. Tanenbaum, Prentice Hall, Inc.

2. The Open Group Base Specifications Issue 7, POSIX.1-2008 and IEEE Std 1003.1 -2008, 2016 Edition

3. http://pubs.opengroup.org/onlinepubs/9699919799/

4. https://en.wikipedia.org/wiki/IEEE_802

5. https://en.wikipedia.org/wiki/List_of_network_protocols_(OSI_model)