套接字

Crazyfish*
CC98

在本章中，你将看到另一种进程通信的方法，但这种方法与我们之前讨论的方法有一个关键的区别。到目前为止，我们讨论的所有设施都依赖于单个计算机系统上的共享资源。资源可以多样化；它可以是文件系统空间，共享的物理内存，或者消息队列，但只有在单个机器上运行的进程可以使用它们。

Berkeley 版本的 UNIX 引入了一个新的通信工具，即套接字接口，这是管道概念的一个扩展。Linux 上可用的套接字接口。你可以像使用管道一样使用套接字，但它们包括跨计算机网络的通信。一台机器上的进程可以使用套接字与另一台机器上的进程进行通信，这使得可以在网络上分布客户端/服务器系统。套接字也可以在同一台机器上的进程之间使用。

我们无法在一次课中覆盖 Linux 的广泛网络功能，因此你会在这里找到主要的编程网络接口的描述。这些应该允许你编写自己的网络程序。具体来说，我们看以下内容：

- 套接字连接的操作方式
- 套接字的属性，地址和通信
- 网络信息和 Internet 守护进程（inetd/xinetd）
- 客户端和服务器

1 什么是套接字?

套接字是一种通信机制，它允许在单机上或跨网络开发客户端/服务器系统。如打印，连接数据库，提供网页服务以及 rlogin 远程登录和 ftp 文件

*wang_heyu@msn.com

传输等网络实用程序通常使用套接字进行通信。套接字的创建和使用方式与管道不同，因为它们明确区分了客户端和服务器。套接字机制可以实现多个客户端连接到单个服务器。

2 套接字连接

你可以将套接字连接视为打入繁忙大楼的电话。电话进入一个组织，由接待员接听，然后将电话转接到正确的部门（服务器进程），再从那里转接到正确的人（服务器套接字）。每个进入的呼叫（客户端）被路由到适当的终端，而介入的操作员则可以处理进一步的呼叫。在你了解在 Linux 系统中建立套接字连接的方式之前，你需要了解如何操作维持连接的套接字应用程序。

首先，服务器应用程序创建一个套接字，就像文件描述符一样，它是分配给服务器进程并且仅分配给该进程的资源。服务器使用系统调用 socket 创建它，它不能与其他进程共享。

接下来，服务器进程给套接字一个名字。本地套接字在 Linux 文件系统中被赋予一个文件名，通常可以在 /tmp 或 /usr/tmp 中找到。对于网络套接字，文件名将是客户端可以连接的特定网络相关的服务标识符（端口号/访问点）。此标识符允许 Linux 将指定特定端口号的传入连接路由到正确的服务器进程。例如，Web 服务器通常在端口 80 上创建一个套接字，这是为此目的保留的标识符。Web 浏览器知道要使用端口 80 进行其到用户想要阅读的网站的 HTTP 连接。套接字使用系统调用 bind 命名。然后，服务器进程等待客户端连接到命名的套接字。系统调用 listen 为传入连接创建一个队列。服务器可以使用系统调用 accept 接受它们。

当服务器调用 accept 时，将创建一个与命名套接字不同的新套接字。此新套接字仅用于与此特定客户端的通信。命名套接字仍用于来自其他客户端的进一步连接。如果服务器编写得当，它可以利用多个连接。Web 服务器会这样做，以便一次为多个客户端提供页面。对于简单的服务器，进一步的客户端将在监听队列上等待，直到服务器再次准备好。

套接字系统的客户端更为简单。客户端通过调用 socket 创建一个未命名的套接字。然后，它调用 connect 以使用服务器的命名套接字作为地址与服务器建立连接。

一旦建立，套接字可以像低级文件描述符一样使用，提供双向数据通

信。

3 试一试：一个简单的本地客户端

这是一个非常简单的套接字客户端程序 client1.c 的示例。它创建一个未命名的套接字并将其连接到名为 server_socket 的服务器套接字。我们稍后在讨论了一些寻址问题之后，将详细介绍套接字系统调用。

```
#include <sys/types.h>
// 包含系统类型定义
#include <sys/socket.h>
// 包含套接字功能的函数和数据结构
#include <stdio.h>
// 包含标准输入/输出函数
#include <sys/un.h>
// 包含Unix系统特定的数据结构
#include <unistd.h>
// 包含各种符号常量和类型，并声明各种函数
#include <stdlib.h>
// 包含通用库函数和变量类型

int main()
{
    int sockfd;  // 套接字描述符
    int len;     // 地址长度
    struct sockaddr_un address;
    // Unix域套接字地址结构
    int result;  // 连接结果
    char ch = 'A';  // 要发送的字符

    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    // 创建套接字

    address.sun_family = AF_UNIX;
```

```
    // 设置地址类型为Unix域
    strcpy(address.sun_path, "server_socket");
    // 设置服务器套接字的路径
    len = sizeof(address);  // 获取地址长度

    result = connect(sockfd,
                     (struct sockaddr *)&address, len);
    // 连接到服务器
    if (result == -1)  // 如果连接失败
    {
        perror("oops: client1");  // 打印错误信息
        exit(1);  // 退出程序
    }

    write(sockfd, &ch, 1);  // 向服务器写入一个字符
    read(sockfd, &ch, 1);   // 从服务器读取一个字符
    printf("char from server = %c\n", ch);
    // 打印从服务器读取的字符

    close(sockfd);  // 关闭套接字
    exit(0);  // 退出程序
}
```

当你运行此程序时，由于尚未创建服务器端的命名套接字，所以该程序会失败。（具体的错误信息可能因系统而异。）注意这里真正描述地址的是

```
address.sun_family = AF_UNIX;
```

这意味着我们的通讯将发生在本地机器上，而不是两台机器之间。所以本质上，就是进程间的通讯。所谓 server 和 clinet，只是两个扮演不同角色的进程（程序）。

```
$ ./client1
oops: client1: No such file or directory
$
```

这里我们可以观察到, 系统把一切都看作文件, 甚至套接字也是文件.

这里有一个非常简单的服务器程序 server1.c，它接受来自客户端的连接。它创建了服务器套接字，将其绑定到一个名字，创建了一个监听队列，并接受连接。

```
#include <unistd.h>
// 包含各种符号常量和类型，并声明各种函数
#include <stdlib.h>
// 包含通用库函数和变量类型

int main()
{
    int server_sockfd, client_sockfd;
    // 服务器和客户端的套接字文件描述符
    int server_len, client_len;
    // 服务器和客户端的地址长度
    struct sockaddr_un server_address;
    // 服务器地址
    struct sockaddr_un client_address;
    // 客户端地址

    unlink("server_socket");
    // 如果存在，删除旧的套接字

    server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    // 创建套接字

    server_address.sun_family = AF_UNIX;
    // 设置地址类型为 Unix 域
    strcpy(server_address.sun_path, "server_socket");
    // 设置服务器套接字的路径
    server_len = sizeof(server_address);
    // 获取地址长度

    bind(server_sockfd,
```

```
        (struct sockaddr *)&server_address,
        server_len);
    // 绑定套接字

    listen(server_sockfd, 5);
    // 开始监听，最大连接数为5

    while (1)  // 无限循环，等待客户端连接
    {
        char ch;  // 用于接收和发送的字符

        printf("server waiting\n");  // 打印等待消息

        client_len = sizeof(client_address);
        // 获取客户端地址长度

        client_sockfd
            = accept(server_sockfd,
                (struct sockaddr *)&client_address,
                &client_len);
        // 接受客户端连接

        read(client_sockfd, &ch, 1);
        // 从客户端读取一个字符
        ch++;  // 字符递增
        write(client_sockfd, &ch, 1);
        // 向客户端写入一个字符
        close(client_sockfd);  // 关闭客户端套接字
    }
}
```

我们可以看到，建立了套接字连接后，服务器和客户端之间的读写，很像文件的读写，其实就是。

在此示例中，服务器程序一次只能为一个客户端程序服务。它只是从客

户端读取一个字符，递增它，然后写回去。

当你运行服务器程序时，它创建一个套接字并等待连接。如果你在后台启动它，使其独立运行，那么你可以在前台启动客户端。

```
$ ./server1 &
[1] 1094
$ server waiting
```

当服务器等待连接时，它会打印一条消息。在前面的例子中，服务器等待一个文件系统套接字，你可以使用普通的 ls 命令查看它。

请记住，即使程序通过信号异常终止，当你完成套接字使用后，删除它也是一个好习惯。这可以防止文件系统被未使用的文件混乱。

```
$ ls -lF server_socket
srwxr-xr-x 1 neil users 0 2007-06-23 11:41 server_socket=
```

设备类型是“套接字”，由权限前面的 s 和名称末尾的 = 表示。套接字已经像普通文件一样被创建，权限由当前的 umask 修改。如果你使用 ps 命令，你可以看到服务器在后台运行。它显示为睡眠状态（STAT 为 S），因此不消耗 CPU 资源。

UNIX 从设计之初，就把一切都看作文件，这里的套接字也是文件，所以我们可以用文件的操作来操作套接字。

现在，当你运行客户端程序时，你成功地连接到了服务器。因为服务器套接字存在，你可以连接到它并与服务器通信。

```
$ ./client1
服务器等待
来自服务器的字符 = B
$
```

服务器和客户端的输出在终端上混合在一起，但你可以看到服务器已经从客户端接收了一个字符，增加了它，并返回了它。然后服务器继续等待下一个客户端。如果你一起运行几个客户端，它们将按顺序被服务，尽管你看到的输出可能更混乱。

```
$ ./client1 & ./client1 & ./client1 &
[2] 23412
```

```
[3] 23413
[4] 23414
server waiting
char from server = B
char from server = B
server waiting
char from server = B

[3]   Done  ./client1
[4]-  Done  ./client1
[5]+  Done  ./client1

$
```

为了全面理解这个示例中使用的系统调用，你需要了解一些关于 UNIX 网络的知识。

套接字由三个属性来描述：域（domain）、类型（type）和协议（protocol）。它们还有一个用作其名称的地址。地址的格式根据域（也被称为协议族）的不同而不同。每个协议族可以使用一个或多个地址族来定义地址格式。

域指定了套接字通信可以使用的网络媒介。最常见的套接字域是 AF_INET，它指的是在许多 Linux 局域网以及互联网本身上使用的 Internet 网络。底层协议，即 Internet 协议（IP），只有一个地址族，它强制规定了在网络上指定计算机的特定方式。这被称为 IP 地址。

“下一代”互联网协议，IPv6，已经被设计出来，用于克服标准 IP 的一些问题，特别是可用地址数量的限制。IPv6 使用不同的套接字域，AF_INET6，和不同的地址格式。预计它最终将取代 IP，但这个过程将需要很多年。

虽然名称几乎总是指互联网上的网络机器，但这些名称会被翻译成更低级的 IP 地址。一个 IP 地址的例子是 192.168.1.99。所有 IP 地址都由四个小于 256 的数字表示，这被称为点分四位数。当客户端通过套接字在网络上连接时，它需要服务器计算机的 IP 地址。

服务器计算机可能提供多种服务。客户端可以通过使用 IP 端口在网络机器上寻址特定的服务。一个端口在系统内部通过分配一个唯一的 16 位整数进行识别，外部则通过 IP 地址和端口号的组合进行识别。套接字是必须绑定到端口才能进行通信的通信端点。

服务器在特定的端口上等待连接。众所周知的服务有分配的端口号，这些端口号被所有的 Linux 和 UNIX 机器使用。这些通常（但不总是）小于 1024 的数字。例如，打印机排队程序（515），rlogin（513），ftp（21）和 httpd（80）。最后一个是 web 服务器的标准端口。通常，小于 1024 的端口号保留给系统服务，并且可能只由具有超级用户权限的进程提供服务。X/Open 在 netdb.h 中定义了一个常量，IPPORT_RESERVED，表示最高的保留端口号。

因为有一套标准服务的标准端口号，计算机可以轻松地相互连接，而不需要找出正确的端口。本地服务可能使用非标准端口地址。

第一个示例中的域是 UNIX 文件系统域，AF_UNIX，它可以被基于单个计算机的套接字使用，这台计算机可能并未联网。当这样的时候，底层协议是文件输入/输出，地址是文件名。你用于服务器套接字的地址是 server_socket，当你运行服务器应用程序时，你在当前目录中看到了它。

其他可能使用的域包括基于 ISO 标准协议的网络的 AF_ISO，和 Xerox Network System 的 AF_XNS。我们在这里不会讨论这些。

3.1 套接字类型

一个套接字域可能有许多不同的通信方式，每种方式可能有不同的特性。对于 AF_UNIX 域套接字，这并不是一个问题，因为它们提供了一个可靠的双向通信路径。然而，在网络域中，你需要了解底层网络的特性以及不同的通信机制如何受到它们的影响。

互联网协议提供了两种具有不同服务级别的通信机制：流（streams）和数据报（datagrams）。

流套接字（在某些方面类似于标准输入/输出流）提供了一个有序且可靠的双向字节流连接。因此，发送的数据保证不会丢失、重复或重新排序，除非有错误发生的指示。大的消息被分片，传输，然后重新组装。这类似于文件流，它也接受大量的数据并将其分割为较小的块，以便写入低级磁盘。流套接字具有可预测的行为。

流套接字，由类型 SOCK_STREAM 指定，在 AF_INET 域中由 TCP/IP 连接实现。它们也是 AF_UNIX 域中的常见类型。我们主要关注 SOCK_STREAM 套接字，因为它们在编程网络应用程序中更常用。

TCP/IP 代表传输控制协议/互联网协议。IP 是低级协议，用于数据包，提供通过网络从一个计算机到另一个计算机的路由。TCP 提供序列化、流

量控制和重传以确保大数据传输全部到达并正确，或者报告适当的错误条件。

3.2 数据报套接字

相反，数据报套接字，由类型 SOCK_DGRAM 指定，不建立并维持连接。还有一个可以发送的数据报的大小限制。它作为一个单独的网络消息传输，可能会丢失、重复，或者乱序到达——在其后发送的数据报之前。

数据报套接字在 AF_INET 域中由 UDP/IP 连接实现，提供一个无序、不可靠的服务。(UDP 代表用户数据报协议。) 然而，它们在资源方面相对便宜，因为不需要维持网络连接。它们很快，因为没有相关的连接设置时间。

数据报对于向信息服务的 “一次性” 查询，提供定期状态信息，或进行低优先级的日志记录都很有用。它们的优点是服务器的死亡不会过分地给客户端带来不便，也不需要重新启动客户端。因为基于数据报的服务器通常不保留连接信息，所以它们可以在不打扰客户端的情况下停止和重新启动。

套接字系统调用创建一个套接字，并返回一个可以用于访问套接字的描述符。

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

创建的套接字是通信通道的一个端点。domain 参数指定地址族，type 参数指定与此套接字一起使用的通信类型，protocol 指定要使用的协议。

包括以下表格中的域:

域	描述
AF_UNIX	UNIX 内部（文件系统套接字）
AF_INET	ARPA 互联网协议（UNIX 网络套接字）
AF_ISO	ISO 标准协议
AF_NS	Xerox 网络系统协议
AF_IPX	Novell IPX 协议
AF_APPLETALK	Appletalk DDS

最常见的套接字域是 AF_UNIX，用于通过 UNIX 和 Linux 文件系统实现的本地套接字，和 AF_INET，用于 UNIX 网络套接字。AF_INET

套接字可以被跨 TCP/IP 网络（包括互联网）通信的程序使用。Windows Winsock 接口也提供了对这个套接字域的访问。

套接字参数 type 指定了新套接字要使用的通信特性。可能的值包括 SOCK_STREAM 和 SOCK_DGRAM。

SOCK_STREAM 是一个有序的、可靠的、基于连接的双向字节流。对于 AF_INET 域套接字，这是由在流套接字连接时在两个端点之间建立的 TCP 连接默认提供的。数据可以在套接字连接的两个方向上传递。TCP 协议包括分片和重新组装长消息以及重传可能在网络中丢失的任何部分的设施。

SOCK_DGRAM 是一个数据报服务。你可以使用这个套接字发送固定（通常是小）最大大小的消息，但不能保证消息会被传送或者消息在网络中不会被重新排序。对于 AF_INET 套接字，这种通信类型由 UDP 数据报提供。

用于通信的协议通常由套接字类型和域确定。通常没有选择。当有选择时，使用 protocol 参数。0 选择默认协议，在本章的所有示例中都使用了这个协议。

套接字系统调用返回一个在很多方面类似于低级文件描述符的描述符。当套接字已经连接到另一个端点套接字时，你可以使用 read 和 write 系统调用与描述符发送和接收套接字上的数据。close 系统调用用于结束套接字连接。

3.3 套接字地址

每个套接字域都需要自己的地址格式。对于 AF_UNIX 套接字，地址由结构体 sockaddr_un 描述，该结构体在 sys/un.h 头文件中定义。

```
struct sockaddr_un {
sa_family_t sun_family; /* AF_UNIX */
char sun_path[]; /* 路径名 */
};
```

为了能够将不同类型的地址传递给套接字处理系统调用，每种地址格式都由一个类似的结构体描述，该结构体以一个字段（在本例中为 sun_family）开始，指定地址类型（套接字域）。在 AF_UNIX 域中，地址由结构体的 sun_path 字段中的文件名指定。

在当前的 Linux 系统中，由 X/Open 定义的类型 sa_family_t 在 sys/un.h 中被声明为 short。此外，sun_path 中指定的路径名的大小是有限的（Linux 指定为 108 个字符；其他可能使用 UNIX_MAX_PATH 等常量）。因为地址结构可能在大小上有所不同，许多套接字调用都需要或提供一个长度作为输出，用于复制特定的地址结构。

在 AF_INET 域中，地址使用在 netinet/in.h 中定义的名为 sockaddr_in 的结构体指定，该结构体至少包含以下成员：

```
struct sockaddr_in {
short int sin_family; /* AF_INET */
unsigned short int sin_port; /* 端口号 */
struct in_addr sin_addr; /* 互联网地址 */
};
```

IP 地址结构 in_addr 定义如下：

```
struct in_addr {
unsigned long int s_addr;
};
```

一个 IP 地址的四个字节构成一个单独的 32 位值。一个 AF_INET 套接字完全由其域、IP 地址和端口号描述。从应用程序的角度来看，所有套接字都像文件描述符一样行为，并由一个唯一的整数值进行寻址。

为了让其他进程可以使用套接字（由调用 socket 创建的套接字），服务器程序需要给套接字一个名字。因此，AF_UNIX 套接字与文件系统路径名关联，就像你在 server1 示例中看到的那样。AF_INET 套接字与 IP 端口号关联。

```
#include <sys/socket.h>
int bind(int socket,
         const struct sockaddr *address,
         size_t address_len);
```

bind 系统调用将参数 address 中指定的地址分配给与文件描述符 socket 关联的未命名套接字。地址结构的长度作为 address_len 传递。

地址的长度和格式取决于地址族。特定的地址结构指针在调用 bind 时需要转换为通用地址类型（struct sockaddr *）。

成功完成后，bind 返回 0。如果失败，它返回 -1 并将 errno 设置为以下值之一：

Errno 值	描述
EBADF	文件描述符无效。
ENOTSOCK	文件描述符没有引用套接字。
EINVAL	文件描述符引用了一个已命名的套接字。
EADDRNOTAVAIL	地址不可用。
EADDRINUSE	地址已经绑定到一个套接字。

对于 AF_UNIX 套接字，还有一些其他的值：

Errno 值	描述
EACCESS	由于权限无法创建文件系统名称。
ENOTDIR, ENAMETOOLONG	表示文件名选择不佳。

为了接受套接字上的传入连接，服务器程序必须创建一个队列来存储待处理的请求。它使用 listen 系统调用来实现这一点。

```
#include <sys/socket.h>
int listen(int socket, int backlog);
```

Linux 系统可能会限制队列中可持有的待处理连接的最大数量。在此最大值的限制下，listen 将队列长度设置为 backlog。队列长度内的传入连接将在套接字上保持待处理状态；进一步的连接将被拒绝，客户端的连接将失败。listen 提供了这种机制，允许在服务器程序忙于处理前一个客户端时，将传入连接保持待处理状态。对于 backlog，5 是一个非常常见的值。

listen 函数在成功时返回 0，出错时返回 -1。错误包括 EBADF、EINVAL 和 ENOTSOCK，这些错误也适用于 bind 系统调用。

4 接受连接

一旦服务器程序创建并命名了一个套接字，就可以使用 accept 系统调用等待套接字建立连接。

```
#include <sys/socket.h>
int accept(int socket, struct sockaddr *address, size_t *address_len);
```

当客户端程序试图连接到由参数 socket 指定的套接字时，accept 系统调用返回。客户端是该套接字队列中的第一个待处理连接。accept 函数创建一个新的套接字与客户端通信，并返回其描述符。新的套接字将与服务器监听套接字的类型相同。

套接字必须先前已经通过调用 bind 进行命名，并通过 listen 分配了一个连接队列。调用客户端的地址将放置在由 address 指向的 sockaddr 结构中。如果对客户端地址不感兴趣，可以在这里使用空指针。

address_len 参数指定客户端结构的长度。如果客户端地址长于此值，它将被截断。在调用 accept 之前，必须将 address_len 设置为预期的地址长度。返回时，address_len 将被设置为调用客户端的地址结构的实际长度。

如果套接字的队列上没有待处理的连接，accept 将阻塞（使程序无法继续），直到客户端确实建立连接。你可以通过在套接字文件描述符上使用 O_NONBLOCK 标志，使用你的代码中的 fcntl 函数来改变这种行为，如下所示：

```
int flags = fcntl(socket, F_GETFL, 0);
fcntl(socket, F_SETFL, O_NONBLOCK|flags);
```

当有客户端连接待处理时，accept 函数返回一个新的套接字文件描述符，或者在出错时返回 -1。可能的错误类似于 bind 和 listen 的错误，此外还有 EWOULDBLOCK，这是在指定了 O_NONBLOCK 且没有待处理连接时会出现的错误。如果进程在 accept 被阻塞时被中断，将会出现 EINTR 错误。

客户端程序通过在未命名套接字和服务器监听套接字之间建立连接来连接到服务器。它们通过调用 connect 来实现这一点。

```
#include <sys/socket.h>
int connect(int socket, const struct sockaddr *address, size_t address_len);
```

参数 socket 指定的套接字连接到参数 address 指定的服务器套接字，长度为 address_len。套接字必须是通过调用 socket 获得的有效文件描述符。

如果成功，connect 返回 0，出错时返回 -1。这次可能的错误包括以下内容：

Errno 值	描述
EBADF	在 socket 中传入了无效的文件描述符。
EALREADY	此套接字已经在进行连接。
ETIMEDOUT	发生了连接超时。
ECONNREFUSED	服务器拒绝了请求的连接。

如果无法立即建立连接，connect 将阻塞一段未指定的超时期。一旦超时期满，连接将被中止，connect 将失败。然而，如果 connect 的调用被处理的信号中断，connect 调用将失败（并将 errno 设置为 EINTR），但连接尝试不会被中止——它将异步地建立，程序将必须稍后检查连接是否成功。

与 accept 一样，可以通过在文件描述符上设置 O_NONBLOCK 标志来改变 connect 的阻塞性质。在这种情况下，如果无法立即建立连接，connect 将失败，并将 errno 设置为 EINPROGRESS，连接将异步建立。

尽管异步连接可能难以处理，但你可以在套接字文件描述符上调用 select 来检查套接字是否准备好写入。

你可以通过调用 close 来终止服务器和客户端的套接字连接，就像你对低级文件描述符所做的那样。你应该始终在两端关闭套接字。对于服务器，你应该在 read 返回零时这样做。注意，如果套接字有未传输的数据，是面向连接的类型，并且设置了 SOCK_LINGER 选项，那么 close 调用可能会阻塞。你将在本章后面了解设置套接字选项。

现在我们已经介绍了与套接字相关的基本系统调用，让我们更仔细地看看示例程序。你将尝试将它们转换为使用网络套接字，而不是文件系统套接字。文件系统套接字的缺点是，除非作者使用绝对路径名，否则它会在服务器程序的当前目录中创建。为了使其更具一般性，你需要在全局可访问的目录（例如/tmp）中创建它，这个目录是服务器和其客户端之间约定的。对于网络套接字，你只需要选择一个未使用的端口号。

对于示例，选择端口号 9734。这是一个任意的选择，避免了标准服务（你不能使用 1024 以下的端口号，因为它们是为系统使用保留的）。其他端口号通常会列在系统文件 /etc/services 中，以及在这些端口上提供的服务。当你编写基于套接字的应用程序时，始终选择在此配置文件中未列出的端口号。

你将在本地网络上运行你的客户端和服务器，但网络套接字不仅在本地区域网络上有用；任何有互联网连接的机器（甚至是调制解调器拨号）都可以使用网络套接字与其他人通信。你甚至可以在独立的 UNIX 计算机上

使用基于网络的程序，因为 UNIX 计算机通常配置为使用只包含自身的环回网络。为了说明目的，此示例使用这个环回网络，这对于调试网络应用程序也很有用，因为它排除了任何外部网络问题。

请注意，程序 client2.c 和 server2.c 中有一个故意的错误，你将在 client3.c 和 server3.c 中修复它。请不要在你自己的程序中使用 client2.c 和 server2.c 的代码。

环回网络由一个单独的计算机组成，通常称为 localhost，具有标准的 IP 地址 127.0.0.1。这就是本地机器。你会在网络主机文件 /etc/hosts 中找到它的地址，以及共享网络上其他主机的名称和地址。

每个计算机通信的网络都有一个与之关联的硬件接口。计算机在每个网络上可能有不同的网络名称，肯定会有不同的 IP 地址。例如，Neil 的机器 tilde 有三个网络接口，因此有三个地址。这些地址在 /etc/hosts 中记录如下：

```
127.0.0.1 localhost # Loopback
192.168.1.1 tilde.localnet # Local, private Ethernet
158.152.X.X tilde.demon.co.uk # Modem dial-up
```

第一个是简单的环回网络，第二个是通过以太网适配器访问的局域网，第三个是到互联网服务提供商的调制解调器链接。你可以编写一个基于网络套接字的程序，无需修改就可以与通过任何这些接口访问的服务器进行通信。

```
#include <sys/types.h> // 包含系统类型定义
#include <sys/socket.h> // 包含套接字接口
#include <stdio.h> // 包含标准输入输出库
#include <netinet/in.h> // 包含Internet地址家族定义
#include <arpa/inet.h> // 包含inet_addr函数
#include <unistd.h> // 包含Unix系统调用
#include <stdlib.h> // 包含标准库

int main()
{
    int sockfd; // 套接字描述符
    int len; // 地址长度
    struct sockaddr_in address;
```

```
// 存储服务器地址的结构体
int result; // connect函数的结果
char ch = 'A'; // 要发送的字符

sockfd = socket(AF_INET,
                SOCK_STREAM,
                0); // 创建一个套接字

address.sin_family = AF_INET;
// 设置地址家族为Internet
address.sin_addr.s_addr
    = inet_addr("127.0.0.1"); // 设置服务器IP地址
address.sin_port = 9734; // 设置服务器端口号
len = sizeof(address); // 获取地址长度

// 尝试连接到服务器
result = connect(sockfd,
                 (struct sockaddr *)&address,
                 len);

// 如果连接失败，打印错误并退出
if (result == -1)
{
    perror("oops: client1");
    exit(1);
}

// 向服务器写入一个字符
write(sockfd, &ch, 1);
// 从服务器读取一个字符
read(sockfd, &ch, 1);

// 打印从服务器接收到的字符
```

```
    printf("char from server = %c\n", ch);

    // 关闭套接字
    close(sockfd);

    // 退出程序
    exit(0);
}
```

客户端程序使用来自 include 文件 netinet/in.h 的 sockaddr_in 结构来指定一个 AF_INET 地址。它试图连接到 IP 地址为 127.0.0.1 的主机上的服务器。它使用一个函数，inet_addr，将 IP 地址的文本表示转换为适合套接字寻址的形式。inet 的手册页上有关于其他地址转换函数的更多信息。

当你运行这个版本时，由于在这台机器的 9734 端口上没有运行服务器，所以它无法连接。

```
$ ./client2
oops: client2: Connection refused
$
```

```
#include <sys/types.h> // 包含系统类型定义
#include <sys/socket.h> // 包含套接字接口
#include <stdio.h> // 包含标准输入输出库
#include <netinet/in.h> // 包含Internet地址家族定义
#include <arpa/inet.h> // 包含inet_addr函数
#include <unistd.h> // 包含Unix系统调用
#include <stdlib.h> // 包含标准库

int main()
{
    int sockfd; // 套接字描述符
    int len; // 地址长度
    struct sockaddr_in address;
    // 存储服务器地址的结构体
    int result; // connect函数的结果
```

```
char ch = 'A'; // 要发送的字符

sockfd = socket(AF_INET,
                SOCK_STREAM, 0);
                // 创建一个套接字

address.sin_family = AF_INET;
// 设置地址家族为 Internet
address.sin_addr.s_addr
    = inet_addr("127.0.0.1");
    // 设置服务器IP地址
address.sin_port = 9734; // 设置服务器端口号
len = sizeof(address); // 获取地址长度

// 尝试连接到服务器
result = connect(sockfd,
                 (struct sockaddr *)&address,
                 len);

// 如果连接失败, 打印错误并退出
if (result == -1)
{
    perror("oops: client1");
    exit(1);
}

// 向服务器写入一个字符
write(sockfd, &ch, 1);

// 从服务器读取一个字符
read(sockfd, &ch, 1);

// 打印从服务器接收到的字符
```

```
    printf("char from server = %c\n", ch);

    // 关闭套接字
    close(sockfd);

    // 退出程序
    exit(0);
}
```

服务器程序创建了一个 AF_INET 域套接字，并安排在其上接受连接。套接字绑定到你选择的端口。指定的地址决定了哪些计算机被允许连接。通过指定环回地址，就像在客户端程序中一样，你限制了通信到本地机器。

如果你想让服务器与远程客户端通信，你必须指定一组你愿意允许的 IP 地址。你可以使用特殊值 INADDR_ANY，来指定你将接受来自你的计算机可能拥有的所有接口的连接。如果你愿意，你可以区分不同的网络接口，例如，分隔内部局域网和外部广域网连接。INADDR_ANY 是一个 32 位整数值，你可以在地址结构的 sin_addr.s_addr 字段中使用它。然而，你首先需要解决一个问题。

4.1 主机和网络字节排序

当我们在基于 Intel 处理器的 Linux 机器上运行这些版本的服务器和客户端程序时，我们可以使用 netstat 命令查看网络连接。这个命令也会在大多数配置了网络的 UNIX 系统上可用。它显示了客户端/服务器连接等待关闭。连接在短暂的超时后关闭。（再次说明，不同版本的 Linux 的确切输出可能会有所不同。）

```
$ ./server2 & ./client2
[3] 23770
server waiting
server waiting
char from server = B
$ netstat -A inet
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address (State) User
```

```
tcp 1 0 localhost:1574 localhost:1174 TIME_WAIT root
```

在你尝试更多示例之前，一定要终止正在运行的示例服务器程序，因为它们会竞争接受来自客户端的连接，你会看到混乱的结果。你可以用以下命令杀死它们所有（包括本章后面介绍的）：

```
killall server1 server2 server3 server4 server5
```

`netstat -A inet` 命令用于显示与 Internet 相关的网络统计信息，如 TCP 和 UDP 连接和监听端口。以下是这个命令输出的一些解释：

- `Proto`: 这是使用的协议，可能是 TCP 或 UDP。
- `Recv-Q` 和 `Send-Q`: 这些表示接收和发送队列。这些是还未被进程接收或还未发送出去的数据包数量。
- `Local Address`: 这是本地主机的 IP 地址和端口号。
- `Foreign Address`: 这是远程主机的 IP 地址和端口号。
- `State`: 这是当前连接的状态。可能的状态包括：
  - `TIME_WAIT`: 本地端成功地关闭了连接，远程端也关闭了连接，现在等待所有分组都在网络中消失。
  - `CLOSE_WAIT`: 远程端关闭了连接，等待本地端关闭连接。

`TIME_WAIT` 是 TCP 连接中的一个状态，它出现在连接被本地端点关闭，且远程端点也确认关闭后。在这个状态下，连接的端点继续等待一段时间以确保网络中的所有数据包都被正确地接收。

这个状态的主要目的是处理延迟的数据包。由于网络中的数据包可能会延迟或重新发送，所以即使连接已经关闭，也可能会有旧的数据包到达。如果立即开始一个新的连接，那么这些延迟的数据包可能会被误认为是新连接的数据包，这可能会导致错误。

`TIME_WAIT` 状态通常持续的时间是最大分段寿命（Maximum Segment Lifetime，MSL）的两倍。MSL 是任何给定 TCP 段在 Internet 上存在的最长时间，通常被设置为 2 分钟，所以 `TIME_WAIT` 状态通常会持续 4 分钟。

在 `TIME_WAIT` 状态结束后，连接的端点可以安全地被重新使用，开始新的连接。

你可以看到已经分配给服务器和客户端之间的连接的端口号。本地地址显示的是服务器，外部地址是远程客户端。（即使它在同一台机器上，它仍然通过网络连接。）为了确保所有套接字都是不同的，这些客户端端口通常与服务器监听套接字不同，并且在计算机上是唯一的。

然而，在示例中选择的端口是 9734。为什么不存在？答案是端口号和地址通过套接字接口以二进制数字形式进行通信。不同的计算机对整数使用不同的字节排序。例如，Intel 处理器将 32 位整数作为四个连续字节按照 1-2-3-4 的顺序存储在内存中，其中 1 是最高有效字节。IBM PowerPC 处理器会按照 4-3-2-1 的字节顺序存储整数。如果用于整数的内存被简单地按字节复制，那么两种不同的计算机将无法在整数值上达成一致。

为了使不同类型的计算机能够对通过网络传输的多字节整数的值达成一致，你需要定义一个网络排序。客户端和服务器程序必须在传输之前将它们的内部整数表示转换为网络排序。它们通过使用在 netinet/in.h 中定义的函数来完成这个任务。这些函数是

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

这些函数将 16 位和 32 位整数在本地主机格式和标准网络排序之间进行转换。它们的名字是转换的缩写——例如，“host to network, long”（htonl）和 “host to network, short”（htons）。对于本地排序与网络排序相同的计算机，这些表示空操作。

为了确保 16 位端口号的字节排序正确，你的服务器和客户端需要将这些函数应用到端口地址。对 server3.c 的更改是

```
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_address.sin_port = htons(9734);
```

你不需要转换函数调用，inet_addr(”127.0.0.1”)，因为 inet_addr 被定义为产生一个网络顺序的结果。对 client3.c 的更改是

```
address.sin_port = htons(9734);
```

服务器也被更改为使用 INADDR_ANY 允许来自任何 IP 地址的连接。现在，当你运行 server3 和 client3 时，你会看到正确的端口被用于本地连接。

```
$ netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address Foreign Address (State) User
tcp 1 0 localhost:9734 localhost:1175 TIME_WAIT root
```

如果你正在使用的计算机的本地和网络字节排序相同，你不会看到任何区别。但仍然很重要的是始终使用转换函数，以允许与具有不同架构的计算机上的客户端和服务器正确操作。（在网络通讯场合考虑跨平台是必须的!）

到目前为止，你的客户端和服务器程序已经将地址和端口号编译进去了。对于一个更通用的服务器和客户端程序，你可以使用网络信息函数来确定要使用的地址和端口。

如果你有权限，你可以将你的服务器添加到 /etc/services 中的已知服务列表，这个列表为端口号分配一个名称，以便客户端可以使用符号服务而不是数字。同样，给定计算机的名称，你可以通过调用解析地址的主机数据库函数来确定 IP 地址。它们通过查阅网络配置文件，如/etc/hosts，或网络信息服务，如 NIS（网络信息服务，以前被称为黄页）和 DNS（域名服务）来实现这一点。

主机数据库函数在接口头文件 netdb.h 中声明。

```
#include <netdb.h>
struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
struct hostent *gethostbyname(const char *name);
```

这些函数返回的结构必须至少包含以下成员：

```
struct hostent {
char *h_name; /* 主机的名称 */
char **h_aliases; /* 别名列表（昵称） */
int h_addrtype; /* 地址类型 */
int h_length; /* 地址的字节长度 */
char **h_addr_list /* 地址列表（网络顺序） */
};
```

如果没有指定主机或地址的数据库条目，信息函数将返回一个空指针。

同样，有关服务和相关端口号的信息可以通过一些服务信息函数获得。

```
#include <netdb.h>
```

```
struct servent *getservbyname(const char *name,
                              const char *proto);
struct servent *getservbyport(int port,
                              const char *proto);
```

proto 参数指定用于连接到服务的协议,可以是 “tcp” 用于 SOCK_STREAM TCP 连接，或者是 “udp” 用于 SOCK_DGRAM UDP 数据报文。

servent 结构至少包含以下成员:

```
struct servent {
char *s_name; /* 服务的名称 */
char **s_aliases; /* 别名列表（备用名称） */
int s_port; /* IP端口号 */
char *s_proto; /* 服务类型，通常是“tcp”或“udp” */
};
```

你可以通过调用 gethostbyname 并打印结果来收集关于计算机的主机数据库信息。注意, 地址列表需要转换为适当的地址类型, 并使用 inet_ntoa 转换从网络顺序转换为可打印的字符串，该转换有以下定义:

```
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr in)
```

该函数将 Internet 主机地址转换为点分四组格式的字符串。出错时返回-1, 但 POSIX 没有定义任何特定的错误。你使用的另一个新函数是 gethostname。

```
#include <unistd.h>
int gethostname(char *name, int namelength);
```

此函数将当前主机的名称写入由 name 给出的字符串。主机名将以空字符结尾。参数 namelength 表示字符串 name 的长度，如果返回的主机名太长无法适应，则会被截断。gethostname 在成功时返回 0，在错误时返回-1，但是 POSIX 再次没有定义任何错误。

```
#include <netinet/in.h>
    // 包含网络编程需要的数据类型和结构
#include <arpa/inet.h>    // 提供IP地址转换函数
```

```
#include <unistd.h>       // 提供Unix标准函数
#include <netdb.h>        // 提供网络数据库操作的函数
#include <stdio.h>        // 提供输入输出函数
#include <stdlib.h>       // 提供标准库函数

int main(int argc, char *argv[])
{
    char *host, **names, **addrs;
        // 定义主机名，别名和地址变量
    struct hostent *hostinfo;       // 定义主机信息结构

    // 如果没有传入主机名，获取本机主机名
    if (argc == 1)
    {
        char myname[256];
        gethostname(myname, 255);
        host = myname;
    }
    else
        host = argv[1];  // 否则使用输入的主机名

    // 获取主机信息
    hostinfo = gethostbyname(host);
    // 如果无法获取主机信息，打印错误并退出
    if (!hostinfo)
    {
        fprintf(stderr,
                "cannot get info for host: %s\n",
                host);
        exit(1);
    }

    // 打印主机信息
```

```
    printf("results for host %s:\n", host);
    printf("Name: %s\n", hostinfo->h_name);

    // 打印主机别名
    printf("Aliases:");
    names = hostinfo->h_aliases;
    while (*names)
    {
        printf(" %s", *names);
        names++;
    }
    printf("\n");

    // 如果不是IP主机，打印错误并退出
    if (hostinfo->h_addrtype != AF_INET)
    {
        fprintf(stderr, "not an IP host!\n");
        exit(1);
    }

    // 打印主机IP地址
    addrs = hostinfo->h_addr_list;
    while (*addrs)
    {
        printf(" %s",
               inet_ntoa(*(struct in_addr *)*addrs));
        addrs++;
    }
    printf("\n");

    exit(0);
}
```

getname 程序调用 gethostbyname 从主机数据库中提取主机信息。它

打印出主机名，它的别名（计算机的其他已知名称），以及主机在其网络接口上使用的 IP 地址。

现在你可以修改你的客户端以连接到任何命名的主机。你可以连接到你的服务器上的一个标准服务，并提取端口号。

大多数 UNIX 和一些 Linux 系统将其系统时间和日期作为一个名为 daytime 的标准服务提供。客户端可以连接到这个服务以发现服务器当前的时间和日期。下面是一个客户端程序，getdate.c，它就是这样做的。

试一下:（香港天文台）

```
./getdate stdtime.gov.hk
```

如果你收到如下错误信息:

```
oops: getdate: Connection refused
```

或

```
oops: getdate: No such file or directory
```

可能是因为你正在连接的计算机没有启用 daytime 服务。

当你运行这个程序时，你可以指定一个要连接的主机。daytime 服务的端口号是通过网络数据库函数 getservbyname 来确定的，它以类似于主机信息的方式返回关于网络服务的信息。getdate 程序试图连接到指定主机的备用地址列表中首先给出的地址。如果成功，它会读取 daytime 服务返回的信息，一个表示 UNIX 时间和日期的字符字符串。