网络编程

网络应用随处可见。任何时候你浏览 Web、发送 E-mail 或者弹出一个 X window，你就正在使用一个网络应用程序。有趣的是，所有的网络应用都是基于相同的基本编程模型，有着相似的整体逻辑结构，并且依赖相同的编程接口。

网络应用依赖于很多在系统研究中已经学习过的概念，例如，进程、信号、字节顺序、存储器映射以及动态存储分配，都扮演着重要的角色。还有一些新概念要掌握。我们需要理解基本的客户端 - 服务器编程模型，以及如何编写使用因特网提供的服务的客户端 - 服务器程序。最后，我们将把所有这些概念结合起来，开发一个小的但功能齐全的 Web 服务器，能够为真实的 Web 浏览器提供静态和动态的文本和图形内容。

11.1 客户端 - 服务器编程模型

每个网络应用都是基于*客户端 - 服务器模型*的。采用这个模型，一个应用是由一个*服务器进程*和一个或者多个*客户端进程*组成。服务器管理某种*资源*，并且通过操作这种资源来为它的客户端提供某种*服务*。例如，一个 Web 服务器管理了一组磁盘文件，它会代表客户端进行检索和执行。一个 FTP 服务器就管理了一组磁盘文件，它会为客户端进行存储和检索。相似地，一个电子邮件服务器管理了一些文件，它为客户端进行读和更新。

客户端 - 服务器模型中的基本操作是*事务*（transaction）（见图 11-1）。一个客户端 - 服务器事务由四步组成：

1）当一个客户端需要服务时，它向服务器发送一个*请求*，发起一个事务。例如，当 Web 浏览器需要一个文件时，它就发送一个请求给 Web 服务器。

2）服务器收到请求后，解释它，并以适当的方式操作它的资源。例如，当 Web 服务器收到浏览器发出的请求后，它就读一个磁盘文件。

3）服务器给客户端发送一个*响应*，并等待下一个请求。例如，Web 服务器将文件发送回客户端。

4）客户端收到响应并处理它。例如，当 Web 浏览器收到来自服务器的一页后，它就在屏幕上显示此页。

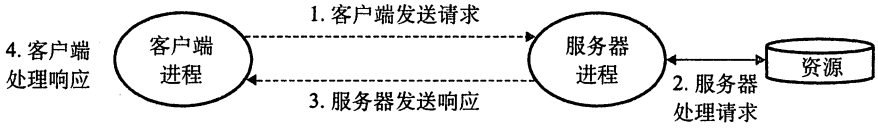

图 11-1　一个客户端 - 服务器事务

认识到客户端和服务器是进程，而不是常常提到的机器或者主机，这是很重要的。一台主机可以同时运行许多不同的客户端和服务器，而且一个客户端和服务器的事务可以在同一台或是不同的主机上运行。无论客户端和服务器是怎样映射到主机上的，客户端 - 服务器模型是相同的。

客户端 - 服务器事务和数据库事务

客户端 - 服务器事务不是数据库事务，没有数据库事务的任何特性，例如原子性。在我们的上下文中，事务仅仅是客户端和服务器执行的一系列步骤。

www.TopSage.com

11.2　网络

客户端和服务器通常运行在不同的主机上，并且通过计算机网络的硬件和软件资源来通信。网络是复杂的系统，在这里我们只想了解一点皮毛。我们的目标是从程序员的角度给你一个可工作的思考模型。

对于一个主机而言，网络只是又一种 I/O 设备，作为数据源和数据接收方，如图 11-2 所示。一个插到 I/O 总线扩展槽的适配器提供了到网络的物理接口。从网络上接收到的数据从适配器经过 I/O 和存储器总线拷贝到存储器，典型地是通过 DMA（译者注：直接存储器存取方式）传送。相似地，数据也能从存储器拷贝到网络。

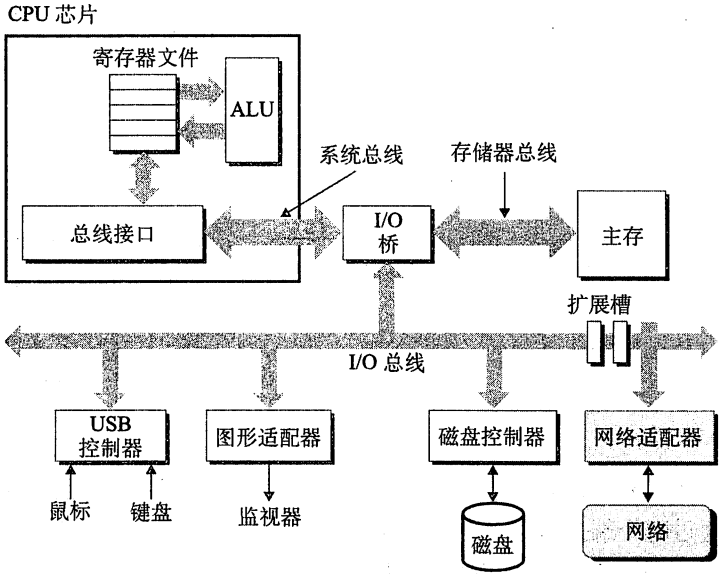

图 11-2　一个网络主机的硬件组成

物理上而言，网络是一个按照地理远近组成的层次系统。最低层是 LAN（Local Area Network，局域网），在一个建筑或者校园范围内。迄今为止，最流行的局域网技术是*以太网*（Ethernet），它是由施乐公司帕洛阿尔托研究中心（Xerox PARC）在 20 世纪 70 年代中期提出的。以太网技术被证明是适应力极强的，从 3MB/s 演变到 10GB/s。

一个*以太网段*（Ethernet segment）包括一些电缆（通常是双绞线）和一个叫做集线器的小盒子，如图 11-3 所示。以太网段通常跨越一些小的区域，例如某建筑物的一个房间或者一个楼层。每根电缆都有相同的最大位带宽，典型的是 100MB/s 或者 1GB/s。一端连接到主机的适配器，而另一端则连接到集线器的一个端口上。集线器不加分辨地将从一个端口上收到的每个位复制到其他所有的端口上。因此，每台主机都能看到每个位。

每个以太网适配器都有一个全球唯一的 48 位地址，它存储在这个适配器的非易失性存储器上。一台主机可以发送一段位，称为*帧*（frame），到这个网段内其他任何主机。每个帧包括一些固定数量的*头部*（header）位，用来标识此帧的源和目的地址以及此帧的长度，此后紧随的就是数据位的*有效载荷*。每个主机适配器都能看到这个帧，但是只有目的主机实际读取它。

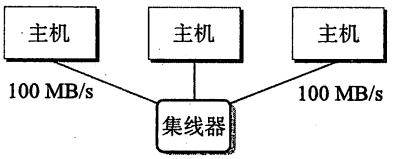

图 11-3　以太网段

www.TopSage.com

使用一些电缆和叫做网桥（bridge）的小盒子，多个以太网段可以连接成较大的局域网，称为桥接以太网（bridged Ethernet），如图 11-4 所示。桥接以太网能够跨越整个建筑物或者校区。在一个桥接以太网里，一些电缆连接网桥与网桥，而另外一些连接网桥和集线器。这些电缆的带宽可以是不同的。在我们的示例中，网桥与网桥之间的电缆有 1GB/s 的带宽，而四根网桥和集线器之间电缆的带宽却是 100MB/s。

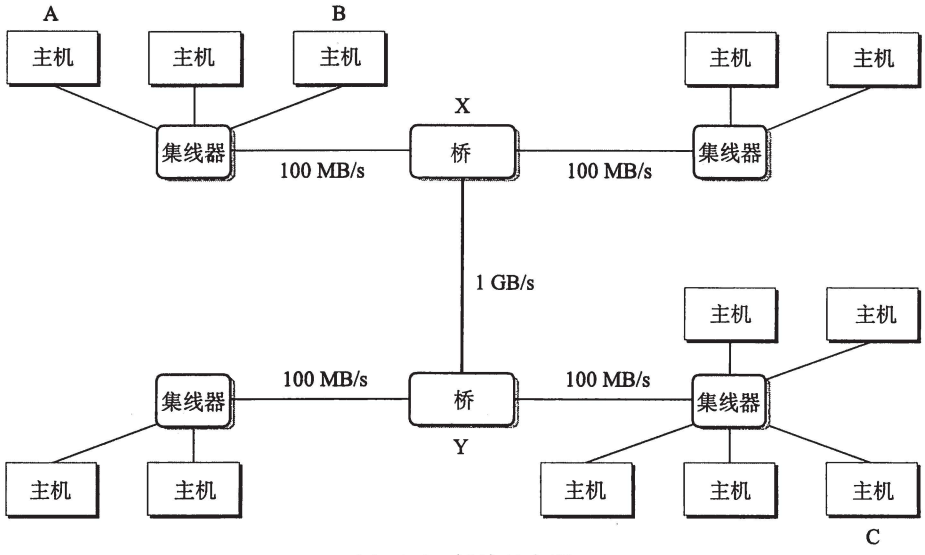

图 11-4　桥接以太网

网桥比集线器更充分地利用了电缆带宽。利用一种聪明的分配算法，它们随着时间自动学习哪个主机可以通过哪个端口可达，然后只在有必要时，有选择地将帧从一个端口拷贝到另一个端口。例如，如果主机 A 发送一个帧到同网段上的主机 B，当该帧到达网桥 X 的输入端口时，X 就将丢弃此帧，因而节省了其他网段上的带宽。然而，如果主机 A 发送一个帧到一个不同网段上的主机 C，那么网桥 X 只会把此帧拷贝到和网桥 Y 相连的端口上，网桥 Y 会只把此帧拷贝到与主机 C 的网段连接的端口。

为了简化局域网的表示，我们将把集线器和网桥以及连接它们的电缆画成一根水平线，如图 11-5 所示。

图 11-5　局域网的概念视图

在层次的更高级别中，多个不兼容的局域网可以通过叫做路由器（router）的特殊计算机连接起来，组成一个 internet（互联网络）。

Internet 和 internet

我们总是用小写字母的 internet 描述一般概念，而用大写字母的 Internet 来描述一种具体的实现，也就是所谓的全球 IP 因特网。

每台路由器对于它所连接到的每个网络都有一个适配器（端口）。路由器也能连接高速点到点电话连接，这是称为 WAN（Wide-Area Network，广域网）的网络示例，之所以这么叫是因为它们覆盖的地理范围比局域网的大。一般而言，路由器可以用来由各种局域网和广域网构建互联网络。例如，图 11-6 展示了一个互联网络的示例，3 台路由器连接了一对局域网和一对广域网。

www.TopSage.com

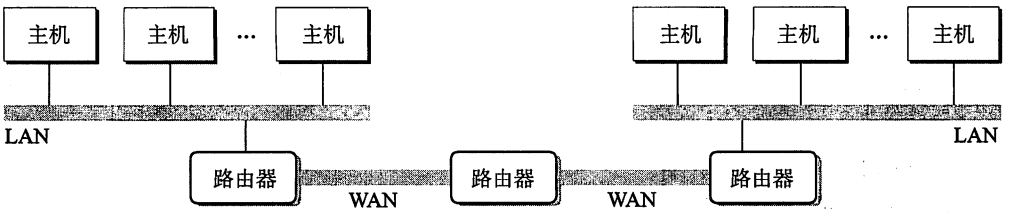

图 11-6 一个小型的互联网络。两个局域网和两个广域网用三台路由器连接起来

互联网络至关重要的特性是，它能由采用完全不同和不兼容技术的各种局域网和广域网组成。每台主机和其他每台主机都是物理相连的，但是如何能够让某台源主机跨过所有这些不兼容的网络发送数据位到另一台目的主机呢？

解决办法是一层运行在每台主机和路由器上的协议软件，它消除了不同网络之间的差异。这个软件实现一种协议，这种协议控制主机和路由器如何协同工作来实现数据传输。这种协议必须提供两种基本能力：

- 命名机制。不同的局域网技术有不同和不兼容的方式来为主机分配地址。互联网络协议通过定义一种一致的主机地址格式消除了这些差异。每台主机会被分配至少一个这种互联网络地址（internet address），这个地址唯一地标识了这台主机。
- 传送机制。在电缆上编码位和将这些位封装成帧方面，不同的联网技术有不同的和不兼容的方式。互联网络协议通过定义一种把数据位捆扎成不连续的片（称为包）的统一方式，从而消除了这些差异。一个包是由包头和有效载荷组成的，其中包头包括包的大小以及源主机和目的主机的地址，有效载荷包括从源主机发出的数据位。

图 11-7 展示了一个主机和路由器如何使用互联网络协议在不兼容的局域网间传送数据的示例。这个互联网络示例由两个局域网通过一台路由器连接而成。一个客户端运行在主机 A 上，主机 A 与 LAN1 相连，它发送了一串数据字节到运行在主机 B 上的服务器端，主机 B 则连接在 LAN2 上。这个过程包括 8 个基本步骤：

1）运行在主机 A 上的客户端进行了一个系统调用，从客户端的虚拟地址空间拷贝数据到内核缓冲区中。

2）主机 A 上的协议软件通过在数据前附加互联网络包头和 LAN1 帧头，创建了一个 LAN1 的帧。互联网络包头寻址到互联网络主机 B。LAN1 帧头寻址到路由器。然后它传送此帧到适配器。注意，LAN1 帧的有效载荷是一个互联网络包，而互联网络包的有效载荷是实际的用户数据。这种封装是基本的网络互联方法之一。

3）LAN1 适配器拷贝该帧到网络上。

4）当此帧到达路由器时，路由器的 LAN1 适配器从电缆上读取它，并把它传送到协议软件。

5）路由器从互联网络包头中提取出目的互联网络地址，并用它作为路由表的索引，确定向哪里转发这个包，在本例中是 LAN2。路由器剥落旧的 LAN1 的帧头，加上寻址到主机 B 的新的 LAN2 帧头，并把得到的帧传送到适配器。

6）路由器的 LAN2 适配器拷贝该帧到网络上。

7）当此帧到达主机 B 时，它的适配器从电缆上读到此帧，并将它传送到协议软件。

8）最后，主机 B 上的协议软件剥落包头和帧头。当服务器进行一个读取这些数据的系统调用时，协议软件最终将得到的数据拷贝到服务器的虚拟地址空间。

www.TopSage.com

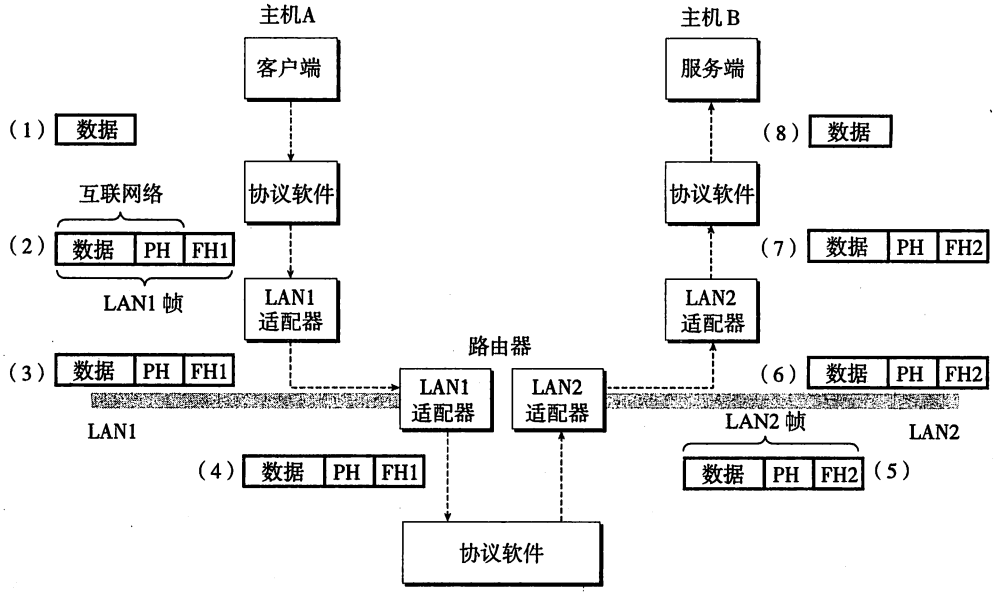

图 11-7 在互联网络上，数据是如何从一台主机传送到另一台主机的。关键词：PH：互联网络包头；FH1：LAN1 的帧头；FH2：LAN2 的帧头

当然，在这里我们掩盖了许多很难的问题。如果不同的网络有不同帧大小的最大值，该怎么办呢？路由器如何知道该往哪里转发帧呢？当网络拓扑变化时，如何通知路由器？如果一个包丢失了又会如何呢？虽然如此，我们的示例抓住了互联网络思想的精髓，封装是关键。

11.3 全球 IP 因特网

全球 IP 因特网是最著名和最成功的互联网络实现。从 1969 年起，它就以这样或那样的形式存在了。虽然因特网的内部体系结构复杂而且不断变化，但是自从 20 世纪 80 年代早期以来，客户端 - 服务器应用的组织就一直保持相当的稳定。图 11-8 展示了一个因特网客户端 – 服务器应用程序的基本硬件和软件组织。每台因特网主机都运行实现 TCP/IP 协议（Transmission Control Protocol/Internet Protocol，传输控制协议 / 互联网络协议）的软件，几乎每个现代计算机系统都支持这个协议。因特网的客户端和服务器混合使用套接字接口函数和 Unix I/O 函数来进行通信（我们将在 11.4 节中介绍套接字接口）。套接字函数典型地是作为会陷入内核的系统调用来实现的，并调用各种内核模式的 TCP/IP 函数。

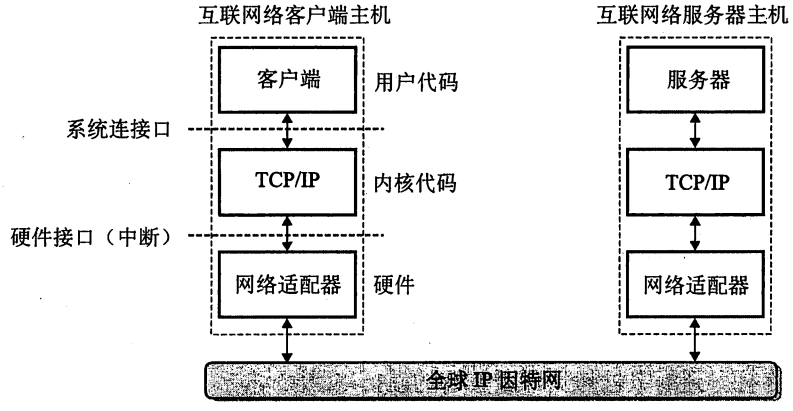

图 11-8 一个因特网应用程序的硬件和软件组织

www.TopSage.com

TCP/IP 实际上是一个协议族，其中每一个都提供不同的功能。例如，IP 协议提供基本的命名方法和递送机制，这种递送机制能够从一台因特网主机往其他主机发送包，也叫做*数据报*（datagram）。IP 机制从某种意义上而言是不可靠的，因为，如果数据报在网络中丢失或者重复，它并不会试图恢复。UDP（Unreliable Datagram Protocol，不可靠数据报协议）稍微扩展了 IP 协议，这样一来，包可以在进程间而不是在主机间传送。TCP 是一个构建在 IP 之上的复杂协议，提供了进程间可靠的全双工（双向的）连接。为了简化讨论，我们将 TCP/IP 看做是一个单独的整体协议。我们将不讨论它的内部工作，只讨论 TCP 和 IP 为应用程序提供的某些基本功能。我们将不讨论 UDP。

从程序员的角度，我们可以把因特网看做一个世界范围的主机集合，满足以下特性：

- 主机集合被映射为一组 32 位的 IP *地址*。
- 这组 IP 地址被映射为一组称为*因特网域名*（Internet domain name）的标识符。
- 因特网主机上的进程能够通过*连接*（connection）和任何其他因特网主机上的进程通信。

接下来的三节将更详细地讨论这些基本的因特网概念。

11.3.1 IP 地址

一个 IP 地址就是一个 32 位无符号整数。网络程序将 IP 地址存放在如图 11-9 所示的 IP *地址结构*中。

netinet/in.h

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* Network byte order (big-endian) */
};
```

netinet/in.h

图 11-9 IP 地址结构

为什么要用结构来存放标量 IP 地址？

把一个标量地址存放在结构中，是套接字接口早期实现的不幸产物。为 IP 地址定义一个标量类型应该更有意义，但是现在更改已经太迟了，因为已经有大量应用是基于此的了。

因为因特网主机可以有不同的主机字节顺序，TCP/IP 为任意整数数据项定义了统一的*网络字节顺序*（network byte order）（大端字节顺序），例如 IP 地址，它放在包头中跨过网络被携带。在 IP 地址结构中存放的地址总是以（大端法）网络字节顺序存放的，即使主机字节顺序（host byte order）是小端法。Unix 提供了下面这样的函数在网络和主机字节顺序间实现转换：

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
```

返回：按照网络字节顺序的值。

```
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

返回：按照主机字节顺序的值。

`htonl` 函数将 32 位整数由主机字节顺序转换为网络字节顺序。`ntohl` 函数将 32 位整数从网络字节顺序转换为主机字节。`htons` 和 `ntohs` 函数为 16 位的整数执行相应的转换。

IP 地址通常是以一种称为*点分十进制表示法*来表示的，这里，每个字节由它的十进制值表示，

www.TopSage.com

并且用句点和其他字节间分开。例如，128.2.194.242 就是地址 0x8002c2f2 的点分十进制表示。在 Linux 系统上，你能够使用 HOSTNAME 命令来确定你自己主机的点分十进制地址：

```
linux> hostname -i
128.2.194.242
```

因特网程序使用 inet_aton 和 inet_ntoa 函数来实现 IP 地址和点分十进制串之间的转换：

```
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
                                    返回：若成功则为 1，若出错则为 0。

char *inet_ntoa(struct in_addr in);
                                    返回：指向点分十进制字符串的指针。
```

inet_aton 函数将一个点分十进制串（cp）转换为一个网络字节顺序的 IP 地址（inp）。相似地，inet_ntoa 函数将一个网络字节顺序的 IP 地址转换为它所对应的点分十进制串。注意，对 inet_aton 的调用传递的是指向结构的指针，而对 inet_ntoa 的调用传递的是结构本身。

ntoa 和 aton 是什么意思？

“n”表示的是网络（network）。“a”表示应用（application）。而“to”表示转换。

练习题 11.1　完成下表：

十六进制地址	点分十进制地址
0x0	
0xffffffff	
0x7f000001	
	205.188.160.121
	64.12.149.13
	205.188.146.23

练习题 11.2　编写程序 hex2dd.c，它将十六进制参数转换为点分十进制串并打印出结果。例如

```
unix> ./hex2dd 0x8002c2f2
128.2.194.242
```

练习题 11.3　编写程序 dd2hex.c，它将它的点分十进制参数转换为十六进制数并打印出结果。例如

```
unix> ./dd2hex 128.2.194.242
0x8002c2f2
```

11.3.2　因特网域名

因特网客户端和服务器互相通信时使用的是 IP 地址。然而，对于人们而言，大整数是很难记住的，所以因特网也定义了一组更加人性化的域名（domain name），以及一种将域名映射到 IP 地址的机制。域名是一串用句点分隔的单词（字母、数字和破折号），例如

```
kittyhawk.cmcl.cs.cmu.edu
```

www.TopSage.com

域名集合形成了一个层次结构，每个域名编码了它在这个层次中的位置。通过一个示例你将很容易理解这点。图 11-10 展示了域名层次结构的一部分。层次结构可以表示为一棵树。树的节点表示域名，反向到根的路径形成了域名。子树称为子域（subdomain）。层次结构中的第一层是一个未命名的根节点。下一层是一组一级域名（first-level domain name），由非赢利组织 ICANN（Internet Corporation for Assigned Names and Numbers，因特网分配名字数字协会）定义。常见的第一层域名包括 com、edu、gov、org 和 net。

下一层是二级（second-level）域名，例如 cmu.edu，这些域名是由 ICANN 的各个授权代理按照先到先服务的基础分配的。一旦一个组织得到了一个二级域名，那么它就可以在这个子域中创建任何新的域名了。

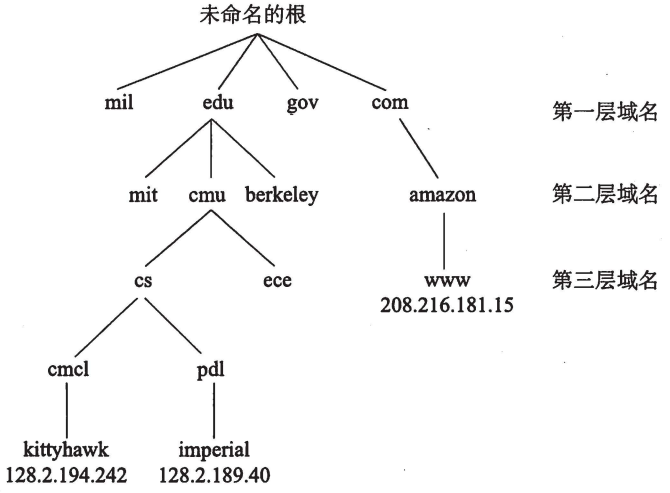

图 11-10　因特网域名层次结构的一部分

因特网定义了域名集合和 IP 地址集合之间的映射。直到 1988 年，这个映射都是通过一个叫做 HOSTS.TXT 的文本文件来手工维护的。从那以后，这个映射是通过分布世界范围内的数据库（称为 DNS（Domain Name System，域名系统））来维护的。从概念上而言，DNS 数据库由上百万的如图 11-11 所示的主机条目结构（host entry structure）组成的，其中每条定义了一组域名（一个官方名字和一组别名）和一组 IP 地址之间的映射。从数学意义上讲，你可以认为每条主机条目就是一个域名和 IP 地址的等价类。

netdb.h

```
/* DNS host entry structure */
struct hostent {
   char    *h_name;        /* Official domain name of host */
   char    **h_aliases;    /* Null-terminated array of domain names */
   int     h_addrtype;     /* Host address type (AF_INET) */
   int     h_length;       /* Length of an address, in bytes */
   char    **h_addr_list;  /* Null-terminated array of in_addr structs */
};
```

netdb.h

图 11-11　DNS 主机条目结构

因特网应用程序通过调用 gethostbyname 和 gethostbyaddr 函数，从 DNS 数据库中检索任意的主机条目。

www.TopSage.com

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

返回：若成功则为非 NULL 指针，若出错则为 NULL 指针，同时设置 h_errno。

```
struct hostent *gethostbyaddr(const char *addr, int len, 0);
```

返回：若成功则为非 NULL，若出错则为 NULL 指针，同时设置 h_errno。

gethostbyname 函数返回和域名 name 相关的主机条目。gethostbyaddr 函数返回和 IP 地址 addr 相关联的主机条目。第二个参数给出了一个 IP 地址的字节长度，对于目前的因特网而言总是四个字节。对于我们的要求来说，第三个参数总是零。

code/netp/hostinfo.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    char **pp;
    struct in_addr addr;
    struct hostent *hostp;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <domain name or dotted-decimal>\n",
                argv[0]);
        exit(0);
    }

    if (inet_aton(argv[1], &addr) != 0)
        hostp = Gethostbyaddr((const char *)&addr, sizeof(addr), AF_INET);
    else
        hostp = Gethostbyname(argv[1]);

    printf("official hostname: %s\n", hostp->h_name);

    for (pp = hostp->h_aliases; *pp != NULL; pp++)
        printf("alias: %s\n", *pp);

    for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
        addr.s_addr = ((struct in_addr *)*pp)->s_addr;
        printf("address: %s\n", inet_ntoa(addr));
    }
    exit(0);
}
```

code/netp/hostinfo.c

图 11-12　检索并打印一个 DNS 主机条目

我们可以借助图 11-12 中的 hostinfo 程序来挖掘一些 DNS 映射的特性，这个程序从命令行读取一个域名或点分十进制地址，并显示相应的主机条目。每台因特网主机都有本地定义的域名 localhost，这个域名总是映射为本地回送地址（loopback address）127.0.0.1：

```
unix> ./hostinfo localhost
official hostname: localhost
alias: localhost.localdomain
address: 127.0.0.1
```

www.TopSage.com

localhost 名字为引用运行在同一台机器上的客户端和服务器提供了一种便利和可移植的方式，这对调试相当有用。我们可以使用 HOSTNAME 来确定本地主机的实际域名：

```
unix> hostname
bluefish.ics.cs.cmu.edu
```

在最简单的情况下，一个域名和一个 IP 地址之间是一一映射的：

```
unix> ./hostinfo bluefish.ics.cs.cmu.edu
official hostname: bluefish.ics.cs.cmu.edu
alias: bluefish.alias.cs.cmu.edu
address: 128.2.205.216
```

然而，在某些情况下，多个域名可以映射为同一个 IP 地址：

```
unix> ./hostinfo cs.mit.edu
official hostname: eecs.mit.edu
alias: cs.mit.edu
address: 18.62.1.6
```

在最通常的情况下，多个域名可以映射到多个 IP 地址：

```
unix> ./hostinfo google.com
official hostname: google.com
address: 74.125.45.100
address: 74.125.67.100
address: 74.125.127.100
```

最后，我们注意到某些合法的域名没有映射到任何 IP 地址：

```
unix> ./hostinfo edu
Gethostbyname error: No address associated with name
unix> ./hostinfo cmcl.cs.cmu.edu
Gethostbyname error: No address associated with name
```

有多少因特网主机？

因特网软件协会（Internet Software Consortium，www.isc.org）自从 1987 年以后，每年进行两次因特网域名调查。这个调查通过计算已经分配给一个域名的 IP 地址的数量来估算因特网主机的数量，展示了一种令人吃惊的趋势。自从 1987 年以来，当时一共大约有 20 000 台因特网主机，每年主机数量都大概会翻一番。到 2009 年 6 月，全球已经有大约 700 000 000 台因特网主机了。

练习题 11.4 编译图 11-12 中的 HOSTINFO 程序。然后在你的系统上连续运行 hostinfo google.com 三次。

A. 在三个主机条目的 IP 地址顺序中，你注意到了什么？

B. 这种顺序有何作用？

11.3.3 因特网连接

因特网客户端和服务器通过在*连接*上发送和接收字节流来通信。从连接一对进程的意义上而言，连接是*点对点*的。从数据可以同时双向流动的角度来说，它是*全双工*的。并且从（除了一些如粗心的耕锄机操作员切断了电缆引起灾难性的失败以外）由源进程发出的字节流最终被目的进程以它发出的顺序收到它的角度来说，它也是*可靠*的。

一个*套接字*是连接的一个端点。每个套接字都有相应的*套接字地址*，是由一个因特网地址和一个 16 位的整数*端口*组成的，用“地址：端口”来表示。当客户端发起一个连接请求时，客户端套接字地址中的端口是由内核自动分配的，称为*临时端口*（ephemeral port）。然而，服务器套

www.TopSage.com

接字地址中的端口通常是某个*知名的端口*，是和这个服务相对应的。例如，Web 服务器通常使用端口 80，而电子邮件服务器使用端口 25。在 Unix 机器上，文件 /etc/services 包含一张这台机器提供的服务以及它们的知名端口号的综合列表。

一个连接是由它两端的套接字地址唯一确定的。这对套接字地址叫做*套接字对*（socket pair），由下列元组来表示：

```
(cliaddr:cliport, servaddr:servport)
```

其中 cliaddr 是客户端的 IP 地址，cliport 是客户端的端口，servaddr 是服务器的 IP 地址，而 servport 是服务器的端口。例如，图 11-13 展示了一个 Web 客户端和一个 Web 服务器之间的连接。

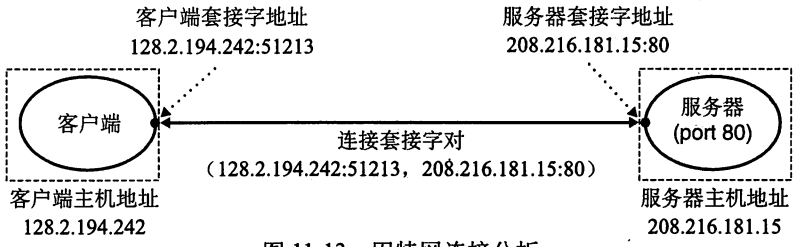

图 11-13　因特网连接分析

在这个示例中，Web 客户端的套接字地址是

```
128.2.194.242:51213
```

其中端口号 51213 是内核分配的临时端口号。Web 服务器的套接字地址是

```
208.216.181.15:80
```

其中端口号 80 是和 Web 服务相关联的知名端口号。给定这些客户端和服务器套接字地址，客户端和服务器之间的连接就由下列套接字对唯一确定了：

```
(128.2.194.242:51213, 1208.216.181.15:80)
```

因特网的起源

因特网是政府、学校和工业界合作的最成功的示例之一。它成功的因素很多，但是我们认为有两点尤其重要：美国政府 30 年持续不变的投资，以及充满激情的研究人员对麻省理工学院的 Dave Clarke 提出的“粗略一致和能用的代码”的投入。

因特网的种子是在 1957 年播下的，当时正值冷战的高峰，苏联发射 Sputnik，第一颗人造地球卫星，震惊了世界。作为响应，美国政府创建了高级研究计划署（ARPA），其任务就是重建美国在科学与技术上的领导地位。1967 年，ARPA 的 Lawrence Roberts 提出了一个计划，建立一个叫做 ARPANET 的新网络。第一个 ARPANET 节点是在 1969 年建立并运行的。到 1971 年，已有 13 个 ARPANET 节点了，而且 E-mail 作为第一个重要的网络应用涌现出来。

1972 年，Robert Kahn 概括了网络互联的一般原则：一组互相连接的网络，通过叫做“路由器”的黑盒子按照“尽力传送基础”在互相独立处理的网络间实现通信。1974 年，Kahn 和 Vinton Cerf 发表了 TCP/IP 协议的第一本详细资料，到 1982 年它成为了 ARPANET 的标准网络互联协议。1983 年 1 月 1 日，ARPANET 的每个节点都切换到 TCP/IP，标志着全球 IP 因特网的诞生。

1985 年，Paul Mockapetris 发明了 DNS，有 1 000 多台因特网主机。1986 年，国家科学基金会（NSF）用 56KB/s 的电话线连接了 13 个节点，构建了 NSFNET 的骨干网。其后在 1988 年升

www.TopSage.com

级到 1.5MB/s T1 的连接速率，1991 年为 45MB/s T3 的连接速率。到 1988 年，有超过 50 000 台主机。1989 年，原始的 ARPANET 正式退休了。1995 年，已经有几乎 10 000 000 台因特网主机了，NSF 取消了 NSFNET，并且用基于由公众网络接入点连接的私有商业骨干网的现代因特网架构取代了它。

11.4 套接字接口

套接字接口（socket interface）是一组函数，它们和 Unix I/O 函数结合起来，用以创建网络应用。大多数现代系统上都实现套接字接口，包括所有的 Unix 变种、Windows 和 Macintosh 系统。图 11-14 给出了一个典型的客户端 - 服务器事务的上下文中的套接字接口概述。当讨论各个函数时，你可以使用这张图来作为向导图。

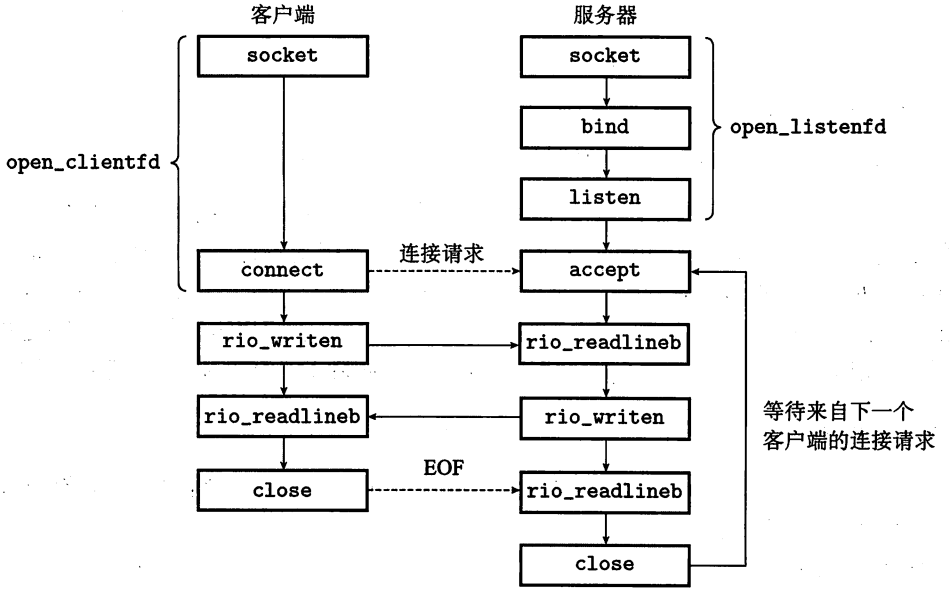

图 11-14 套接字接口概述

套接字接口的起源

套接字接口是加州大学伯克利分校的研究人员在 20 世纪 80 年代早期提出的。因为这个原因，它也经常被叫做伯克利套接字。伯克利的研究者使得套接字接口适用于任何底层的协议。第一个实现的就是针对 TCP/IP 协议的，他们把它包括在 Unix 4.2 BSD 的内核里，并且分发给许多学校和实验室。这在因特网的历史上是一个重大事件。几乎一夜之间，成千上万的人们接触到了 TCP/IP 和它的源代码。它引起了巨大的轰动，并激发了新的网络和网络互联研究的浪潮。

11.4.1 套接字地址结构

从 Unix 内核的角度来看，一个套接字就是通信的一个端点。从 Unix 程序的角度来看，套接字就是一个有相应描述符的打开文件。

因特网的套接字地址存放在如图 11-15 所示的类型为 `sockaddr_in` 的 16 字节结构中。对于因特网应用，`sin_family` 成员是 AF_INET，`sin_port` 成员是一个 16 位的端口号，而 `sin_addr` 成员就是一个 32 位的 IP 地址。IP 地址和端口号总是以网络字节顺序（大端法）存放的。

www.TopSage.com

大家网 TopSage.com

sockaddr: socketbits.h (included by socket.h), sockaddr_in: netinet/in.h

```
/* Generic socket address structure (for connect, bind, and accept) */
struct sockaddr {
    unsigned short  sa_family;   /* Protocol family */
    char            sa_data[14]; /* Address data.  */
};

/* Internet-style socket address structure */
struct sockaddr_in  {
    unsigned short  sin_family;  /* Address family (always AF_INET) */
    unsigned short  sin_port;    /* Port number in network byte order */
    struct in_addr  sin_addr;    /* IP address in network byte order */
    unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```

sockaddr: socketbits.h (included by socket.h), sockaddr_in: netinet/in.h

图 11-15 套接字地址结构。in_addr 结构如图 11-9 所示

_in 后缀意味什么？

_in 后缀是互联网络（internet）的缩写，而不是输入（input）的缩写。

connect、bind 和 accept 函数要求一个指向与协议相关的套接字地址结构的指针。套接字接口的设计者面临的问题是，如何定义这些函数，使之能接受各种类型的套接字地址结构。现在，我们可以使用通用的 void* 指针，那时在 C 中并不存在这种类型的指针。解决办法是定义套接字函数要求一个指向通用 sockaddr 结构的指针，然后要求应用程序将与协议特定的结构的指针强制转换成这个通用结构。为了简化代码示例，我们跟随 Steven 的指导，定义下面的类型：

```
typedef struct sockaddr SA;
```

然后无论何时需要将 sockaddr_in 结构强制转换成通用 sockaddr 结构，我们都使用这个类型（参见图 11-16 的第 20 行的示例）。

11.4.2 socket 函数

客户端和服务器使用 socket 函数来创建一个套接字描述符（socket descriptor）。

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

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

返回：若成功则为非负描述符，若出错则为 −1。

在我们的代码中总是带这样的参数来调用 socket 函数：

```
clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

其中，AF_INET 表明我们正在使用因特网，而 SOCK_STREAM 表示这个套接字是因特网连接的一个端点。socket 返回的 clientfd 描述符仅是部分打开的，还不能用于读写。如何完成打开套接字的工作，取决于我们是客户端还是服务器。下一节描述当我们是客户端时如何完成打开套接字的工作。

11.4.3 connect 函数

客户端通过调用 connect 函数来建立和服务器的连接。

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

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

返回：若成功则为 0，若出错则为 −1。

connect 函数试图与套接字地址为 serv_addr 的服务器建立一个因特网连接，其中 addrlen 是 sizeof(sockaddr_in)。connect 函数会阻塞，一直到连接成功建立或是发生错误。如果成功，sockfd 描述符现在就准备好可以读写了，并且得到的连接是由套接字对

```
(x:y, serv_addr.sin_addr:serv_addr.sin_port)
```

刻画的，其中 x 表示客户端的 IP 地址，而 y 表示临时端口，它唯一地确定了客户端主机上的客户端进程。

11.4.4 open_clientfd 函数

我们发现将 socket 和 connect 函数包装成一个叫做 open_clientfd 的辅助函数是很方便的，客户端可以用它来和服务器建立连接。

```
#include "csapp.h"

int open_clientfd(char *hostname, int port);
```

返回：若成功则为描述符，若 Unix 出错则为 −1，若 DNS 出错则为 −2。

open_clientfd 函数和运行在主机 hostname 上的服务器建立一个连接，并在知名端口 port 上监听连接请求。它返回一个打开的套接字描述符，该描述符准备好了，可以用 Unix I/O 函数做输入和输出。图 11-16 给出了 open_clientfd 的代码。

code/src/csapp.c

```
int open_clientfd(char *hostname, int port)
{
    int clientfd;
    struct hostent *hp;
    struct sockaddr_in serveraddr;

    if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1; /* Check errno for cause of error */

    /* Fill in the server's IP address and port */
    if ((hp = gethostbyname(hostname)) == NULL)
        return -2; /* Check h_errno for cause of error */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *)hp->h_addr_list[0],
          (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
    serveraddr.sin_port = htons(port);

    /* Establish a connection with the server */
    if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
        return -1;
    return clientfd;
}
```

code/src/csapp.c

图 11-16 open_clientfd：和服务器建立连接的辅助函数

www.TopSage.com

在创建了套接字描述符（第 7 行）后，我们检索服务器的 DNS 主机条目，并拷贝主机条目中的第一个 IP 地址（已经是按照网络字节顺序了）到服务器的套接字地址结构（第 11 ～ 16 行）。在用按照网络字节顺序的服务器的知名端口号初始化套接字地址结构（第 17 行）之后，我们发起了一个到服务器的连接请求（第 20 行）。当 connect 函数返回时，我们返回套接字描述符给客户端，客户端就可以立即开始用 Unix I/O 和服务器通信了。

11.4.5 bind 函数

剩下的套接字函数 bind、listen 和 accept 被服务器用来和客户端建立连接。

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

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

返回：若成功则为 0，若出错则为 −1。

bind 函数告诉内核将 my_addr 中的服务器套接字地址和套接字描述符 sockfd 联系起来。参数 addrlen 就是 sizeof(sockaddr_in)。

11.4.6 listen 函数

客户端是发起连接请求的主动实体。服务器是等待来自客户端的连接请求的被动实体。默认情况下，内核会认为 socket 函数创建的描述符对应于*主动套接字*（active socket），它存在于一个连接的客户端。服务器调用 listen 函数告诉内核，描述符是被服务器而不是客户端使用的。

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

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

返回：若成功则为 0，若出错则为 −1。

listen 函数将 sockfd 从一个主动套接字转化为一个*监听套接字*（listening socket），该套接字可以接受来自客户端的连接请求。backlog 参数暗示了内核在开始拒绝连接请求之前，应该放入队列中等待的未完成连接请求的数量。backlog 参数的确切含义要求对 TCP/IP 协议的理解，这超出了我们讨论的范围。通常我们会把它设置为一个较大的值，比如 1024。

11.4.7 open_listenfd 函数

我们发现将 socket、bind 和 listen 函数结合成一个叫做 open_listenfd 的辅助函数是很有帮助的，服务器可以用它来创建一个监听描述符。

```
#include "csapp.h"

int open_listenfd(int port);
```

返回：若成功则为描述符，若 Unix 出错则为 −1。

open_listenfd 函数打开和返回一个监听描述符，这个描述符准备好在知名端口 port 上接收连接请求。图 11-17 展示了 open_listenfd 的代码。在我们创建了 listenfd 套接字描述符之后，我们使用 setsockopt 函数（在这里没有描述）来配置服务器，使得它能被立即终止和重启。默认地，一个重启的服务器将在大约 30 秒内拒绝客户端的连接请求，严重地阻碍了调试。

www.TopSage.com

code/src/csapp.c

```
1  int open_listenfd(int port)
2  {
3      int listenfd, optval=1;
4      struct sockaddr_in serveraddr;
5
6      /* Create a socket descriptor */
7      if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8          return -1;
9
10     /* Eliminates "Address already in use" error from bind */
11     if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
12                    (const void *)&optval , sizeof(int)) < 0)
13         return -1;
14
15     /* Listenfd will be an end point for all requests to port
16        on any IP address for this host */
17     bzero((char *) &serveraddr, sizeof(serveraddr));
18     serveraddr.sin_family = AF_INET;
19     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
20     serveraddr.sin_port = htons((unsigned short)port);
21     if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
22         return -1;
23
24     /* Make it a listening socket ready to accept connection requests */
25     if (listen(listenfd, LISTENQ) < 0)
26         return -1;
27     return listenfd;
28 }
```

code/src/csapp.c

图 11-17 open_listenfd：打开和返回一个监听套接字的辅助函数

接下来，我们初始化服务器的套接字地址结构，为调用 bind 函数做准备。在这个例子中，我们用 INADDR_ANY 通配符地址来告诉内核这个服务器将接受到这台主机的任何 IP 地址（第 19 行）和到知名端口 port（第 20 行）的请求。注意，我们用 htonl 和 htons 函数将 IP 地址和端口号从主机字节顺序转换为网络字节顺序。最后，我们将 listenfd 转换为一个监听描述符（第 25 行），并将它返回给调用者。

11.4.8 accept 函数

服务器通过调用 accept 函数来等待来自客户端的连接请求：

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

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

返回：若成功则为非负连接描述符，若出错则为 −1。

accept 函数等待来自客户端的连接请求到达侦听描述符 listenfd，然后在 addr 中填写客户端的套接字地址，并返回一个*已连接描述符*（connected descriptor），这个描述符可被用来利用 Unix I/O 函数与客户端通信。

监听描述符和已连接描述符之间的区别使很多人感到迷惑。监听描述符是作为客户端连接请求的一个端点。典型地，它被创建一次，并存在于服务器的整个生命周期。已连接描述符是客户

www.TopSage.com

端和服务器之间已经建立起来了的连接的一个端点。服务器每次接受连接请求时都会创建一次，它只存在于服务器为一个客户端服务的过程中。

图 11-18 描绘了监听描述符和已连接描述符的角色。在第一步中，服务器调用 accept，等待连接请求到达监听描述符，具体地我们设定为描述符 3。回忆一下，描述符 0 ～ 2 是预留给了标准文件的。在第二步中，客户端调用 connect 函数，发送一个连接请求到 listenfd。第三步，accept 函数打开了一个新的已连接描述符 connfd（我们假设是描述符 4），在 clientfd 和 connfd 之间建立连接，并且随后返回 connfd 给应用程序。客户端也从 connect 返回，在这一点以后，客户端和服务器就可以分别通过读和写 clientfd 和 connfd 来回传送数据了。

为何要有监听描述符和已连接描述符之间的区别？

你可能很想知道为什么套接字接口要区别监听描述符和已连接描述符。乍一看，这像是不必要的复杂化。然而，区分这两者被证明是很有用的，因为它使得我们可以建立并发服务器，它能够同时处理许多客户端连接。例如，每次一个连接请求到达监听描述符时，我们可以派生（fork）一个新的进程，它通过已连接描述符与客户端通信。在第 12 章中将介绍更多关于并发服务器的内容。

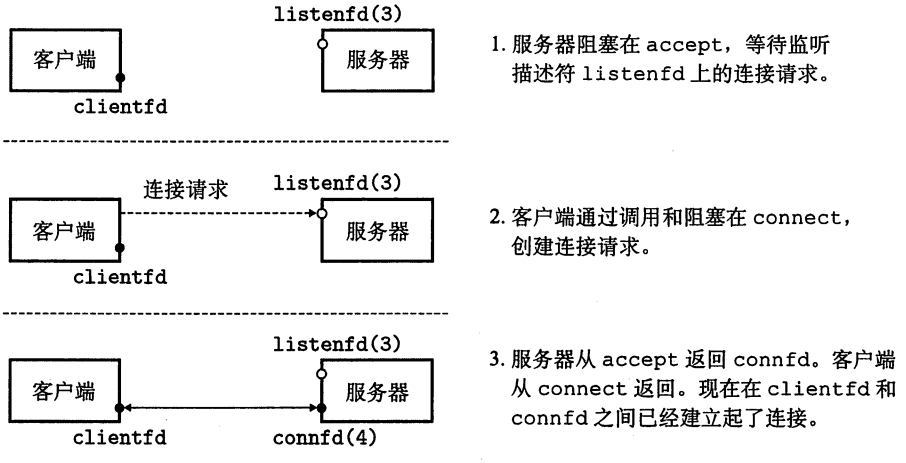

图 11-18　监听描述符和已连接描述符的角色

11.4.9　echo 客户端和服务器的示例

学习套接字接口的最好方法是研究示例代码。图 11-19 展示了一个 echo 客户端的代码。在和服务器建立连接之后，客户端进入一个循环，反复从标准输入读取文本行，发送文本行给服务器，从服务器读取回送的行，并输出结果到标准输出。当 fgets 在标准输入上遇到 EOF 时，或者因为用户在键盘上键入 ctrl-d，或者因为在一个重定向的输入文件中用尽了所有的文本行时，循环就终止。

循环终止之后，客户端关闭描述符。这会导致发送一个 EOF 通知到服务器，当服务器从它的 rio_readlineb 函数收到一个为零的返回码时，就会检测到这个结果。在关闭它的描述符后，客户端就终止了。既然客户端内核在一个进程终止时会自动关闭所有打开的描述符，第 24 行的 close 就没有必要了。不过，显式地关闭已经打开的任何描述符是一个良好的编程习惯。

www.TopSage.com

大家网 TopSage.com

code/netp/echoclient.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;

    if (argc != 3) {
        fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
        exit(0);
    }
    host = argv[1];
    port = atoi(argv[2]);

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

code/netp/echoclient.c

图 11-19 echo 客户端的主程序

图 11-20 展示了 echo 服务器的主程序。在打开监听描述符后，它进入一个无限循环。每次循环都等待一个来自客户端的连接请求，输出已连接客户端的域名和 IP 地址，并调用 echo 函数为这些客户端服务。在 echo 程序返回后，主程序关闭已连接描述符。一旦客户端和服务器关闭了它们各自的描述符，连接也就终止了。

注意，简单的 echo 服务器一次只能处理一个客户端。这种类型的服务器一次一个地在客户端间迭代，称为*迭代服务器*（iterative server）。在第 12 章中，我们将学习如何建立更加复杂的*并发服务器*（concurrent server），它能够同时处理多个客户端。

最后，图 11-21 展示了 echo 程序的代码，该程序反复读写文本行，直到 rio_readlineb 函数在第 10 行遇到 EOF。

在连接中 EOF 意味着什么？

EOF 的概念常常使人们感到迷惑，尤其是在因特网连接的上下文中。首先，我们需要理解其实并没有像 EOF 字符这样的一个东西。进一步来说，EOF 是由内核检测到的一种条件。应用程序在它接收到一个由 read 函数返回的零返回码时，它就会发现 EOF 条件。对于磁盘文件，当前文件位置超出文件长度时，会发生 EOF。对于因特网连接，当一个进程关闭连接它的那一端时，会发生 EOF。连接另一端的进程在试图读取流中最后一个字节之后的字节时，会检测到 EOF。

code/netp/echoserveri.c

```
#include "csapp.h"

void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    listenfd = Open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

        /* Determine the domain name and IP address of the client */
        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("server connected to %s (%s)\n", hp->h_name, haddrp);

        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

code/netp/echoserveri.c

图 11-20 迭代 echo 服务器的主程序

code/netp/echo.c

```
#include "csapp.h"

void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        Rio_writen(connfd, buf, n);
    }
}
```

code/netp/echo.c

图 11-21 读和回送文本行的 echo 函数

www.TopSage.com

11.5 Web 服务器

迄今为止，我们已经在一个简单的 echo 服务器的上下文中讨论了网络编程。在这一节里，我们将向你展示如何利用网络编程的基本概念，来创建你自己的虽然小但是功能齐全的 Web 服务器。

11.5.1 Web 基础

Web 客户端和服务器之间的交互用的是一个基于文本的应用级协议，叫做 HTTP（Hypertext Transfer Protocol，超文本传输协议）。HTTP 是一个简单的协议。一个 Web 客户端（即浏览器）打开一个到服务器的因特网连接，并且请求某些内容。服务器响应所请求的内容，然后关闭连接。浏览器读取这些内容，并把它显示在屏幕上。

Web 服务和常规的文件检索服务（例如 FTP）有什么区别呢？主要的区别是 Web 内容可以用一种叫做 HTML（Hypertext Markup Language，超文本标记语言）的语言来编写。一个 HTML 程序（页）包含指令（标记），它们告诉浏览器如何显示这页中的各种文本和图形对象。例如，代码

```
<b> Make me bold! </b>
```

告诉浏览器用粗体字类型输出 和 标记之间的文本。然而，HTML 真正的强大之处在于一个页面可以包含指针（超链接），这些指针可以指向存放在任何因特网主机上的内容。例如，一个格式如下的 HTML 行

```
<a href="http://www.cmu.edu/index.html">Carnegie Mellon</a>
```

告诉浏览器高亮显示文本对象“`Carnegie Mellon`”，并且创建一个超链接，它指向存放在 CMU Web 服务器上叫做 `index.html` 的 HTML 文件。如果用户单击了这个高亮文本对象，浏览器就会从 CMU 服务器中请求相应的 HTML 文件并显示它。

万维网的起源

万维网（World Wide Web）是 Tim Berners-Lee 创建的，他是一位在瑞典物理实验室 CERN（欧洲粒子物理研究所）工作的软件工程师。1989 年，Berners-Lee 写了一个内部备忘录，提出了一个分布式超文本系统，它能连接“用链接组成的笔记的网”（web of notes with links）。提出这个系统的目的是帮助 CERN 的科学家共享和管理信息。在接下来的两年多里，Berners-Lee 实现了第一个 Web 服务器和 Web 浏览器之后，在 CERN 内部以及其他一些网站中，Web 发展出了小规模的拥护者。1993 年一个关键事件发生了，Marc Andreesen（他后来创建了 Netscape）和他在 NCSA 的同事发布了一种图形化的浏览器，叫做 MOSAIC，可以在三种主要的平台上使用：Unix、Windows 和 Macintosh。在 MOSAIC 发布后，对 Web 的兴趣爆发了，Web 网站以每年 10 倍或更高的数量增长。到 2009 年，世界上已经有超过 225 000 000 个 Web 网站了（源自 Netcraft Web Survey）。

11.5.2 Web 内容

对于 Web 客户端和服务器而言，内容是与一个 MIME（Multipurpose Internet Mail Extensions，多用途的网际邮件扩充协议）类型相关的字节序列。图 11-22 展示了一些常用的 MIME 类型。

Web 服务器以两种不同的方式向客户端提供内容：

- 取一个磁盘文件，并将它的内容返回给客户端。磁盘文件称为静态内容（static content），而返回文件给客户端的过程称为服务静态内容（serving static content）。

www.TopSage.com

MIME 类型	描述
text/html	HTML 页面
text/plain	无格式文本
application/postscript	Postscript 文档
image/gif	GIF 格式编码的二进制图像
image/jpeg	JPEG 格式编码的二进制图像

图 11-22　MIME 类型示例

- 运行一个可执行文件，并将它的输出返回给客户端。运行时可执行文件产生的输出称为动态内容（dynamic content），而运行程序并返回它的输出到客户端的过程称为服务动态内容（serving dynamic content）。

每条由 Web 服务器返回的内容都是和它管理的某个文件相关联的。这些文件中的每一个都有一个唯一的名字，叫做 URL（Universal Resource Locator，通用资源定位符）。例如，URL

```
http://www.google.com:80/index.html
```

表示因特网主机 www.google.com 上一个称为 /index.html 的 HTML 文件，它是由一个监听端口 80 的 Web 服务器管理的。端口号是可选的，而知名的 HTTP 默认的端口就是 80。可执行文件的 URL 可以在文件名后包括程序参数。“?”字符分隔文件名和参数，而且每个参数都用“&”字符分隔开。例如，URL

```
http://bluefish.ics.cs.cmu.edu:8000/cgi-bin/adder?15000&213
```

标识了一个叫做 /cgi-bin/adder 的可执行文件，会带两个参数字符串 15000 和 213 来调用它。在事务过程中，客户端和服务器使用的是 URL 的不同部分。例如，客户端使用前缀

```
http://www.google.com:80
```

来决定与哪类服务器联系，服务器在哪里，以及它监听的端口号是多少。服务器使用后缀

```
/index.html
```

来发现在它文件系统中的文件，并确定请求的是静态内容还是动态内容。

关于服务器如何解释一个 URL 的后缀，以下几点需要理解：

- 确定一个 URL 指向的是静态内容还是动态内容没有标准的规则。每个服务器对它所管理的文件都有自己的规则。一种常见的方法是，确定一组目录，例如 cgi-bin，所有的可执行性文件都必须存放这些目录中。
- 后缀中的最开始的那个“/”不表示 Unix 的根目录。相反，它表示的是被请求内容类型的主目录。例如，可以将一个服务器配置成这样：所有的静态内容存放在目录 /usr/httpd/html 下，而所有的动态内容都存放在目录 /usr/httpd/cgi-bin 下。
- 最小的 URL 后缀是“/”字符，所有服务器将其扩展为某个默认的主页，例如 /index.html。这解释了为什么简单地在浏览器中键入一个域名就可以取出一个网站的主页。浏览器在 URL 后添加缺失的“/”，并将之传递给服务器，服务器又把“/”扩展到某个默认的文件名。

11.5.3　HTTP 事务

因为 HTTP 是基于在因特网连接上传送的文本行的，我们可以使用 Unix 的 TELNET 程序来和因特网上的任何 Web 服务器执行事务。对于调试在连接上通过文本行来与客户端对话的服务器来

www.TopSage.com

说，TELNET 程序是非常便利的。例如，图 11-23 使用 TELNET 向 AOL Web 服务器请求主页。

```
1   unix> telnet www.aol.com 80          Client: open connection to server
2   Trying 205.188.146.23...             Telnet prints 3 lines to the terminal
3   Connected to aol.com.
4   Escape character is '^]'.
5   GET / HTTP/1.1                       Client: request line
6   Host: www.aol.com                    Client: required HTTP/1.1 header
7                                        Client: empty line terminates headers
8   HTTP/1.0 200 OK                      Server: response line
9   MIME-Version: 1.0                    Server: followed by five response headers
10  Date: Mon, 8 Jan 2010 4:59:42 GMT
11  Server:  Apache-Coyote/1.1
12  Content-Type: text/html              Server: expect HTML in the response body
13  Content-Length: 42092                Server: expect 42,092 bytes in the response body
14                                       Server: empty line terminates response headers
15  <html>                               Server: first HTML line in response body
16  ...                                  Server: 766 lines of HTML not shown
17  </html>                              Server: last HTML line in response body
18  Connection closed by foreign host.   Server: closes connection
19  unix>                                Client: closes connection and terminates
```

图 11-23 一个服务静态内容的 HTTP 事务

在第 1 行，我们从 Unix 外壳运行 TELNET，要求它打开一个到 AOL Web 服务器的连接。TELNET 向终端打印三行输出，打开连接，然后等待我们输入文本（第 5 行）。每次我们输入一个文本行，并键入回车键，TELNET 会读取该行，在后面加上回车和换行符号（在 C 的表示中为“\r\n”），并且将这一行发送到服务器。这是和 HTTP 标准相符的，HTTP 标准要求每个文本行都由一对回车和换行符来结束。为了发起事务，我们输入一个 HTTP 请求（第 5 ～ 7 行）。服务器返回 HTTP 响应（第 8 ～ 17 行），然后关闭连接（第 18 行）。

1. HTTP 请求

一个 HTTP *请求*的组成是这样的：一个*请求行*（request line）（第 5 行），后面跟随零个或更多个*请求报头*（request header）（第 6 行），再跟随一个空的文本行来终止报头列表（第 7 行）。一个请求行的形式是

```
<method> <uri> <version>
```

HTTP 支持许多不同的*方法*，包括 GET、POST、OPTIONS、HEAD、PUT、DELETE 和 TRACE。我们将只讨论广为应用的 GET 方法，根据某项调查研究，它占了 99% 的 HTTP 请求 [107]。GET 方法指导服务器生成和返回 URI（Uniform Resource Identifier，统一资源标识符）标识的内容。URI 是相应的 URL 的后缀，包括文件名和可选的参数。[⊖]

请求行中的 <version> 字段表明了该请求遵循的 HTTP 版本。最新的 HTTP 版本是 HTTP/1.1[41]。HTTP/1.0 是从 1996 年沿用至今的老版本 [6]。HTTP/1.1 定义了一些附加的报头，为诸如缓冲和安全等高级特性提供支持，它还支持一种机制，允许客户端和服务器在同一条*持久连接*（persistent connection）上执行多个事务。实际上，两个版本是互相兼容的，因为 HTTP/1.0 的客户端和服务器会简单地忽略 HTTP/1.1 的报头。

⊖ 实际上，只有当浏览器请求内容时，这才是真的。如果代理服务器请求内容，那么这个 URI 必须是完整的 URL。

www.TopSage.com

大家网 TopSage.com

总的来说，第 5 行的请求行要求服务器取出并返回 HTML 文件 /index.html。它也告知服务器请求剩下的部分是 HTTP/1.1 格式的。

请求报头为服务器提供了额外的信息，例如浏览器的商标名，或者浏览器理解的 *MIME* 类型。请求报头的格式为

```
<header name>: <header data>
```

针对我们的目的，唯一需要关注的报头是 Host 报头（第 6 行），这个报头在 HTTP/1.1 请求中是需要的，而在 HTTP/1.0 请求中是不需要的。*代理缓存*（proxy cache）会使用 Host 报头，这个代理缓存有时作为浏览器和管理被请求文件的*原始服务器*（origin server）的中介。客户端和原始服务器之间，可以有多个代理，即所谓的*代理链*（proxy chain）。Host 报头中的数据指示了原始服务器的域名，使得代理链中的代理能够判断它是否可以在本地缓存中拥有一个被请求内容的副本。

继续图 11-23 中的示例，第 7 行的空文本行（通过在键盘上键入回车键生成的）终止了报头，并指示服务器发送被请求的 HTML 文件。

2. HTTP 响应

HTTP 响应和 HTTP 请求是相似的。一个 HTTP 响应的组成是这样的：一个*响应行*（response line）（第 8 行）后面跟随着零个或更多的*响应报头*（response header）（第 9 ～ 13 行），再跟随一个终止报头的空行（第 14 行），再跟随一个*响应主体*（response body）（第 15～17 行）。一个响应行的格式是

```
<version>  <status code> <status message>
```

版本字段描述的是响应所遵循的 HTTP 版本。*状态码*（status code）是一个三位的正整数，指明对请求的处理。*状态消息*（status message）给出与错误代码等价的英文描述。图 11-24 列出了一些常见的状态码，以及它们相应的消息。第 9 ～ 13 行的响应报头提供了关于响应的附加信息。针对我们的目的，两个最重要的报头是 Content-Type（第 12 行），它告诉客户端响应主体中内容的 MIME 类型；以及 Content-Length（第 13 行），用来指示响应主体的字节大小。

第 14 行的终止响应报头的空文本行，其后跟随着响应主体，响应主体中包含着被请求的内容。

状态代码	状态消息	描述
200	成功	处理请求无误
301	永久移动	内容已移动到位置头中指明的主机上
400	错误请求	服务器不能理解请求
403	禁止	服务器无权访问所请求的文件
404	未发现	服务器不能找到所请求的文件
501	未实现	服务器不支持请求的方法
505	HTTP 版本不支持	服务器不支持请求的版本

图 11-24　一些 HTTP 状态码

11.5.4　服务动态内容

如果我们停下来考虑一下，一个服务器是如何向客户端提供动态内容的，就会发现一些问题。例如，客户端如何将程序参数传递给服务器？服务器如何将这些参数传递给它所创建的子进程？服务器如何将子进程生成内容所需要的其他信息传递给子进程？子进程将它的输出发送到哪里？一个称为 CGI（Common Gateway Interface，通用网关接口）的实际标准的出现解决了这些问题。

1. 客户端如何将程序参数传递给服务器

GET 请求的参数在 URI 中传递。正如我们看到的，一个“?”字符分隔了文件名和参数，而

每个参数都用一个“&”字符分隔开。参数中不允许有空格，而必须用字符串“%20”来表示。对其他特殊字符，也存在着相似的编码。

在 HTTP POST 请求中传递参数

HTTP POST 请求的参数是在请求主体中而不是 URI 中传递的。

2. 服务器如何将参数传递给子进程

在服务器接收一个如下的请求后

```
GET /cgi-bin/adder?15000&213 HTTP/1.1
```

它调用 fork 来创建一个子进程，并调用 execve 在子进程的上下文中执行 /cgi-bin/adder 程序。像 adder 这样的程序，常常称为 CGI 程序，因为它们遵守 CGI 标准的规则。而且，因为许多 CGI 程序是用 Perl 脚本编写的，所以 CGI 程序也常称为 CGI 脚本。在调用 execve 之前，子进程将 CGI 环境变量 QUERY_STRING 设置为“15000&213”，adder 程序在运行时可以用 Unix getenv 函数来引用它。

3. 服务器如何将其他信息传递给子进程

CGI 定义了大量的其他环境变量，一个 CGI 程序在它运行时可以设置这些环境变量。图 11-25 给出了其中的一部分。

环境变量	描述
QUERY_STRING	程序参数
SERVER_PORT	父进程侦听的端口
REQUEST_METHOD	GET 或 POST
REMOTE_HOST	客户端的域名
REMOTE_ADDR	客户端的点为十进制 IP 地址
CONTENT_TYPE	只对 POST 而言：请求体的 MIME 类型
CONTENT_LENGTH	只对 POST 而言：请求体的字节大小

图 11-25 CGI 环境变量示例

4. 子进程将它的输出发送到哪里

一个 CGI 程序将它的动态内容发送到标准输出。在子进程加载并运行 CGI 程序之前，它使用 Unix dup2 函数将标准输出重定向到和客户端相关联的已连接描述符。因此，任何 CGI 程序写到标准输出的东西都会直接到达客户端。

注意，因为父进程不知道子进程生成的内容的类型或大小，所以子进程就要负责生成 Content-type 和 Content-length 响应报头，以及终止报头的空行。

图 11-26 展示了一个简单的 CGI 程序，它对两个参数求和，并返回带结果的 HTML 文件给客户端。图 11-27 展示了一个 HTTP 事务，它根据 adder 程序提供动态内容。

将 HTTP POST 请求中的参数传递给 CGI 程序

对于 POST 请求，子进程也需要重定向标准输入到已连接描述符。然后，CGI 程序会从标准输入中读取请求主体中的参数。

练习题 11.5 在 10.9 节中，我们警告过你关于在网络应用中使用 C 标准 I/O 函数的危险。然而，图 11-26 中的 CGI 程序却能没有任何问题地使用标准 I/O。为什么呢？

www.TopSage.com

code/netp/tiny/cgi-bin/adder.c

```
#include "csapp.h"

int main(void) {
    char *buf, *p;
    char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
    int n1=0, n2=0;

    /* Extract the two arguments */
    if ((buf = getenv("QUERY_STRING")) != NULL) {
        p = strchr(buf, '&');
        *p = '\0';
        strcpy(arg1, buf);
        strcpy(arg2, p+1);
        n1 = atoi(arg1);
        n2 = atoi(arg2);
    }

    /* Make the response body */
    sprintf(content, "Welcome to add.com: ");
    sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
    sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
            content, n1, n2, n1 + n2);
    sprintf(content, "%sThanks for visiting!\r\n", content);

    /* Generate the HTTP response */
    printf("Content-length: %d\r\n", (int)strlen(content));
    printf("Content-type: text/html\r\n\r\n");
    printf("%s", content);
    fflush(stdout);
    exit(0);
}
```

code/netp/tiny/cgi-bin/adder.c

图 11-26 对两个整数求和的 CGI 程序

```
unix> telnet kittyhawk.cmcl.cs.cmu.edu 8000    Client: open connection
Trying 128.2.194.242...
Connected to kittyhawk.cmcl.cs.cmu.edu.
Escape character is '^]'.
GET /cgi-bin/adder?15000&213 HTTP/1.0     Client: request line
                                          Client: empty line terminates headers
HTTP/1.0 200 OK                           Server: response line
Server: Tiny Web Server                   Server: identify server
Content-length: 115                       Adder: expect 115 bytes in response body
Content-type: text/html                   Adder: expect HTML in response body
                                          Adder: empty line terminates headers
Welcome to add.com: THE Internet addition portal.   Adder: first HTML line
<p>The answer is: 15000 + 213 = 15213     Adder: second HTML line in response body
<p>Thanks for visiting!                   Adder: third HTML line in response body
Connection closed by foreign host.        Server: closes connection
unix>                                     Client: closes connection and terminates
```

图 11-27 一个提供动态 HTML 内容的 HTTP 事务

www.TopSage.com

11.6 综合：TINY Web 服务器

我们通过开发一个虽然小但是功能齐全的称为 TINY 的 Web 服务器来结束我们对网络编程的讨论。TINY 是一个有趣的程序。在短短 250 行代码中，它结合了许多我们已经学习到的思想，例如进程控制、Unix I/O、套接字接口和 HTTP。虽然它缺乏一个实际服务器所具备的功能性、健壮性和安全性，但是它足够用来为实际的 Web 浏览器提供静态和动态的内容。我们鼓励你研究它，并且自己实现它。将一个实际的浏览器指向你自己的服务器，看着它显示一个复杂的带有文本和图片的 Web 页面，真是非常令人兴奋（甚至对我们这些作者来说，也是如此）。

1. TINY 的 main 程序

图 11-28 展示了 TINY 的主程序。TINY 是一个迭代服务器，监听在命令行中传递来的端口上的连接请求。在通过调用 open_listenfd 函数打开一个监听套接字以后，TINY 执行典型的无限服务器循环，不断地接受连接请求（第 31 行），执行事务（第 32 行），并关闭连接它的那一端（第 33 行）。

code/netp/tiny/tiny.c

```
1  /*
2   * tiny.c - A simple, iterative HTTP/1.0 Web server that uses the
3   *     GET method to serve static and dynamic content.
4   */
5  #include "csapp.h"
6
7  void doit(int fd);
8  void read_requesthdrs(rio_t *rp);
9  int parse_uri(char *uri, char *filename, char *cgiargs);
10 void serve_static(int fd, char *filename, int filesize);
11 void get_filetype(char *filename, char *filetype);
12 void serve_dynamic(int fd, char *filename, char *cgiargs);
13 void clienterror(int fd, char *cause, char *errnum,
14                 char *shortmsg, char *longmsg);
15
16 int main(int argc, char **argv)
17 {
18     int listenfd, connfd, port, clientlen;
19     struct sockaddr_in clientaddr;
20
21     /* Check command line args */
22     if (argc != 2) {
23         fprintf(stderr, "usage: %s <port>\n", argv[0]);
24         exit(1);
25     }
26     port = atoi(argv[1]);
27
28     listenfd = Open_listenfd(port);
29     while (1) {
30         clientlen = sizeof(clientaddr);
31         connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
32         doit(connfd);
33         Close(connfd);
34     }
35 }
```

code/netp/tiny/tiny.c

图 11-28 TINY Web 服务器

www.TopSage.com

2. doit 函数

图 11-29 中的 doit 函数处理一个 HTTP 事务。首先，我们读和解析请求行（第 11 ～ 12 行）。注意，我们使用图 11-7 中的 rio_readlineb 函数读取请求行。

TINY 只支持 GET 方法。如果客户端请求其他方法（比如 POST），我们发送给它一个错误信息，并返回到主程序（第 13 ～ 17 行），主程序随后关闭连接并等待下一个连接请求。否则，我们读并且（像我们将要看到的那样）忽略任何请求报头（第 18 行）。

code/netp/tiny/tiny.c

```
1  void doit(int fd)
2  {
3      int is_static;
4      struct stat sbuf;
5      char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
6      char filename[MAXLINE], cgiargs[MAXLINE];
7      rio_t rio;
8
9      /* Read request line and headers */
10     Rio_readinitb(&rio, fd);
11     Rio_readlineb(&rio, buf, MAXLINE);
12     sscanf(buf, "%s %s %s", method, uri, version);
13     if (strcasecmp(method, "GET")) {
14        clienterror(fd, method, "501", "Not Implemented",
15                 "Tiny does not implement this method");
16         return;
17     }
18     read_requesthdrs(&rio);
19
20     /* Parse URI from GET request */
21     is_static = parse_uri(uri, filename, cgiargs);
22     if (stat(filename, &sbuf) < 0) {
23         clienterror(fd, filename, "404", "Not found",
24                     "Tiny couldn't find this file");
25         return;
26     }
27
28     if (is_static) { /* Serve static content */
29         if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) {
30             clienterror(fd, filename, "403", "Forbidden",
31                         "Tiny couldn't read the file");
32             return;
33         }
34         serve_static(fd, filename, sbuf.st_size);
35     }
36     else { /* Serve dynamic content */
37         if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) {
38             clienterror(fd, filename, "403", "Forbidden",
39                         "Tiny couldn't run the CGI program");
40             return;
41         }
42         serve_dynamic(fd, filename, cgiargs);
43     }
44 }
```

code/netp/tiny/tiny.c

图 11-29 TINY doit：处理一个 HTTP 事务

www.TopSage.com

然后，我们将 URI 解析为一个文件名和一个可能为空的 CGI 参数字符串，并且我们设置一个标志，表明请求的是静态内容还是动态内容（第 21 行）。如果文件在磁盘上不存在，我们立即发送一个错误信息给客户端并返回。

最后，如果请求的是静态内容，我们就验证该文件是一个普通文件，而我们是有读权限的（第 29 行）。如果是这样，我们就向客户端提供静态内容（第 34 行）。相似地，如果请求的是动态内容，我们就验证该文件是可执行文件（第 37 行），如果是这样，我们就继续，并且提供动态内容（第 42 行）。

3. clienterror 函数

TINY 缺乏一个实际服务器的许多错误处理特性。然而，它会检查一些明显的错误，并把它们报告给客户端。图 11-30 中的 clienterror 函数发送一个 HTTP 响应到客户端，在响应行中包含相应的状态码和状态消息，响应主体中包含一个 HTML 文件，向浏览器的用户解释这个错误。回想一下，HTML 响应应该指明主体中内容的大小和类型。因此，我们选择创建 HTML 内容为一个字符串，这样一来我们可以简单地确定它的大小。还有，请注意我们为所有的输出使用的都是图 10-3 中的健壮的 rio_writen 函数。

code/netp/tiny/tiny.c

```
void clienterror(int fd, char *cause, char *errnum,
                 char *shortmsg, char *longmsg)
{
    char buf[MAXLINE], body[MAXBUF];

    /* Build the HTTP response body */
    sprintf(body, "<html><title>Tiny Error</title>");
    sprintf(body, "%s<body bgcolor=""ffffff""">\r\n", body);
    sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
    sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
    sprintf(body, "%s<hr><em>The Tiny Web server</em>\r\n", body);

    /* Print the HTTP response */
    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Content-type: text/html\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Content-length: %d\r\n\r\n", (int)strlen(body));
    Rio_writen(fd, buf, strlen(buf));
    Rio_writen(fd, body, strlen(body));
}
```

code/netp/tiny/tiny.c

图 11-30 TINY clienterror：向客户端发送一个出错消息

4. read_requesthdrs 函数

TINY 不使用请求报头中的任何信息。它仅仅调用图 11-31 中的 read_requesthdrs 函数来读取并忽略这些报头。注意，终止请求报头的空文本行是由回车和换行符对组成的，我们在第 6 行中检查它。

5. parse_uri 函数

TINY 假设静态内容的主目录就是它的当前目录，而可执行文件的主目录是 ./cgi-bin。任何包含字符串 cgi-bin 的 URI 都会被认为表示的是对动态内容的请求。默认的文件名是 ./home.html。

www.TopSage.com

code/netp/tiny/tiny.c

```
1   void read_requesthdrs(rio_t *rp)
2   {
3       char buf[MAXLINE];
4
5       Rio_readlineb(rp, buf, MAXLINE);
6       while(strcmp(buf, "\r\n")) {
7           Rio_readlineb(rp, buf, MAXLINE);
8           printf("%s", buf);
9       }
10      return;
11  }
```

code/netp/tiny/tiny.c

图 11-31　TINY read_requesthdrs：读取并忽略请求报头

图 11-32 中的 parse_uri 函数实现了这些策略。它将 URI 解析为一个文件名和一个可选的 CGI 参数字符串。如果请求的是静态内容（第 5 行），我们将清除 CGI 参数串（第 6 行），然后将 URI 转换为一个相对的 Unix 路径名，例如 ./index.html（第 7 ～ 8 行）。如果 URI 是用“/”结尾的（第 9 行），我们将把默认的文件名加在后面（第 10 行）。另一方面，如果请求的是动态内容（第 13 行），我们就会抽取出所有的 CGI 参数（第 14 ～ 20 行），并将 URI 剩下的部分转换为一个相对的 Unix 文件名（第 21 ～ 22 行）。

code/netp/tiny/tiny.c

```
1   int parse_uri(char *uri, char *filename, char *cgiargs)
2   {
3       char *ptr;
4
5       if (!strstr(uri, "cgi-bin")) {  /* Static content */
6           strcpy(cgiargs, "");
7           strcpy(filename, ".");
8           strcat(filename, uri);
9           if (uri[strlen(uri)-1] == '/')
10              strcat(filename, "home.html");
11          return 1;
12      }
13      else {  /* Dynamic content */
14          ptr = index(uri, '?');
15          if (ptr) {
16              strcpy(cgiargs, ptr+1);
17              *ptr = '\0';
18          }
19          else
20              strcpy(cgiargs, "");
21          strcpy(filename, ".");
22          strcat(filename, uri);
23          return 0;
24      }
25  }
```

code/netp/tiny/tiny.c

图 11-32　TINY parse_uri: 解析一个 HTTP URI

www.TopSage.com

6. serve_static 函数

TINY 提供四种不同类型的静态内容：HTML 文件、无格式的文本文件，以及编码为 GIF 和 JPEG 格式的图片。这些文件类型占据 Web 上提供的绝大部分静态内容。

图 11-33 中的 serve_static 函数发送一个 HTTP 响应，其主体包含一个本地文件的内容。首先，我们通过检查文件名的后缀来判断文件类型（第 7 行），并且发送响应行和响应报头给客户端（第 8 ～ 12 行）。注意用一个空行终止报头。

code/netp/tiny/tiny.c

```
1  void serve_static(int fd, char *filename, int filesize)
2  {
3      int srcfd;
4      char *srcp, filetype[MAXLINE], buf[MAXBUF];
5
6      /* Send response headers to client */
7      get_filetype(filename, filetype);
8      sprintf(buf, "HTTP/1.0 200 OK\r\n");
9      sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10     sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
11     sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
12     Rio_writen(fd, buf, strlen(buf));
13
14     /* Send response body to client */
15     srcfd = Open(filename, O_RDONLY, 0);
16     srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
17     Close(srcfd);
18     Rio_writen(fd, srcp, filesize);
19     Munmap(srcp, filesize);
20 }
21
22 /*
23  * get_filetype - derive file type from file name
24  */
25 void get_filetype(char *filename, char *filetype)
26 {
27     if (strstr(filename, ".html"))
28         strcpy(filetype, "text/html");
29     else if (strstr(filename, ".gif"))
30         strcpy(filetype, "image/gif");
31     else if (strstr(filename, ".jpg"))
32         strcpy(filetype, "image/jpeg");
33     else
34         strcpy(filetype, "text/plain");
35 }
```

code/netp/tiny/tiny.c

图 11-33 TINY serve_static：为客户端提供静态内容

接着，我们将被请求文件的内容拷贝到已连接描述符 fd 来发送响应主体。这里的代码是比较微妙的，需要仔细研究。第 15 行以读方式打开 filename，并获得它的描述符。在第 16 行，Unix mmap 函数将被请求文件映射到一个虚拟存储器空间。回想我们在第 9.8 节中对 mmap 的讨论，调用 mmap 将文件 srcfd 的前 filesize 个字节映射到一个从地址 srcp 开始的私有只读虚拟存储器区域。

一旦将文件映射到存储器，我们就不再需要它的描述符了，所以我们关闭这个文件（第 17

www.TopSage.com

行）。执行这项任务失败将导致一种潜在的致命的存储器泄漏。第 18 行执行的是到客户端的实际文件传送。rio_writen 函数拷贝从 srcp 位置开始的 filesize 个字节（它们当然已经被映射到了所请求的文件）到客户端的已连接描述符。最后，第 19 行释放了映射的虚拟存储器区域。这对于避免潜在的致命的存储器泄漏是很重要的。

7. serve_dynamic 函数

TINY 通过派生一个子进程并在子进程的上下文中运行一个 CGI 程序，来提供各种类型的动态内容。

图 11-34 中的 serve_dynamic 函数一开始就向客户端发送一个表明成功的响应行，同时还包括带有信息的 Server 报头。CGI 程序负责发送响应的剩余部分。注意，这并不像我们可能希望的那样健壮，因为它没有考虑到 CGI 程序会遇到某些错误的可能性。

code/netp/tiny/tiny.c

```
1   void serve_dynamic(int fd, char *filename, char *cgiargs)
2   {
3       char buf[MAXLINE], *emptylist[] = { NULL };
4
5       /* Return first part of HTTP response */
6       sprintf(buf, "HTTP/1.0 200 OK\r\n");
7       Rio_writen(fd, buf, strlen(buf));
8       sprintf(buf, "Server: Tiny Web Server\r\n");
9       Rio_writen(fd, buf, strlen(buf));
10
11      if (Fork() == 0) { /* child */
12          /* Real server would set all CGI vars here */
13          setenv("QUERY_STRING", cgiargs, 1);
14          Dup2(fd, STDOUT_FILENO);         /* Redirect stdout to client */
15          Execve(filename, emptylist, environ); /* Run CGI program */
16      }
17      Wait(NULL); /* Parent waits for and reaps child */
18  }
```

code/netp/tiny/tiny.c

图 11-34　TINY serve_dynamic：为客户端提供动态内容

在发送了响应的第一部分后，会派生一个新的子进程（第 11 行）。子进程用来自请求 URI 的 CGI 参数初始化 QUERY_STRING 环境变量（第 13 行）。注意，一个真正的服务器还会在此处设置其他的 CGI 环境变量。为了简短，我们省略了这一步。还有，我们注意到 Solaris 系统使用的是 putenv 函数，而不是 setenv 函数。

接下来，子进程重定向它的标准输出到已连接文件描述符（第 14 行），然后加载并运行 CGI 程序（第 15 行）。因为 CGI 程序运行在子进程的上下文中，它能够访问所有在调用 execve 函数之前就存在的打开文件和环境变量。因此，CGI 程序写到标准输出上的任何东西都将直接送到客户端进程，不会受到任何来自父进程的干涉。

其间，父进程阻塞在对 wait 的调用中，等待当子进程终止的时候，回收操作系统分配给子进程的资源（第 17 行）。

处理过早关闭的连接

尽管一个 Web 服务器的基本功能非常简单，但是我们不想给你一个假象，以为编写一个实际的 Web 服务器是非常简单的。构造一个长时间运行而不崩溃的健壮的 Web 服务器是一个困难的任务，比起在这里我们已经学习了的内容，它要求对 Unix 系统编程有更加深入的理解。例如，

www.TopSage.com

如果一个服务器写一个已经被客户端关闭了的连接（比如，因为你在浏览器上单击了“Stop”按钮），那么第一次这样的写会正常返回，但是第二次写就会引起发送 SIGPIPE 信号，这个信号的默认行为就是终止这个进程。如果捕获或者忽略 SIGPIPE 信号，那么第二次写操作会返回值 -1，并将 errno 设置为 EPIPE。strerr 和 perror 函数将 EPIPE 错误报告为“Broken pipe”，这是一个迷惑了很多人的不太直观的信息。总的来说，一个健壮的服务器必须捕获这些 SIGPIPE 信号，并且检查 write 函数调用是否有 EPIPE 错误。

11.7 小结

每个网络应用都是基于客户端 - 服务器模型的。根据这个模型，一个应用是由一个服务器和一个或多个客户端组成的。服务器管理资源，以某种方式操作资源，为它的客户端提供服务。客户端 - 服务器模型中的基本操作是客户端 - 服务器事务，它是由客户端请求和跟随其后的服务器响应组成的。

客户端和服务器通过因特网这个全球网络来通信。从一个程序员的观点来看，我们可以把因特网看成是一个全球范围的主机集合，具有以下几个属性：1）每个因特网主机都有一个唯一的 32 位名字，称为它的 IP 地址。2）IP 地址的集合被映射为一个因特网域名的集合。3）不同因特网主机上的进程能够通过连接互相通信。

客户端和服务器通过使用套接字接口建立连接。一个套接字是连接的一个端点，连接是以文件描述符的形式提供给应用程序的。套接字接口提供了打开和关闭套接字描述符的函数。客户端和服务器通过读写这些描述符来实现彼此间的通信。

Web 服务器使用 HTTP 协议和它们的客户端（例如浏览器）彼此通信。浏览器向服务器请求静态或者动态的内容。对静态内容的请求是通过从服务器磁盘取得文件并把它返回给客户端来服务的。对动态内容的请求是通过在服务器上一个子进程的上下文中运行一个程序并将它的输出返回给客户端来服务的。CGI 标准提供了一组规则，来管理客户端如何将程序参数传递给服务器，服务器如何将这些参数以及其他信息传递给子进程，以及子进程如何将它的输出发送回客户端。

只用几百行 C 代码就能实现一个简单但是有功效的 Web 服务器，它既可以提供静态内容，也可以提供动态内容。

参考文献说明

有关因特网的官方信息源被保存在一系列的可免费获取的带编号的文档中，这个文档称为 RFC（Requests For Comments，请求注解，Internet 标准（草案））。在以下网站可获得可搜索的 RFC 的索引

```
http://rfc-editor.org
```

RFC 通常是为因特网基础设施的开发者编写的，因此，对于普通读者来说，往往过于详细了。然而，作为权威信息，没有比它更好的信息来源了。HTTP/1.1 协议记录在 RFC 2616 中。MIME 类型的权威列表保存在

```
http://www.iana.org/assignments/media-types
```

关于计算机网络互联有大量很好的通用文献 [62，80，113]。伟大的技术作家 W.Richard Stevens 编写了一系列相关的经典文献，如高级 Unix 编程 [110]、因特网协议 [105，106，107]，以及 Unix 网络编程 [108，109]。认真学习 Unix 系统编程的学生会想要研究所有这些内容。不幸

www.TopSage.com

大家网 TopSage.com

的是，Stevens 在 1999 年 9 月 1 日逝世。我们会永远记住他的贡献。

家庭作业

** 11.6 A. 修改 TINY 使得它会原样返回每个请求行和请求报头。

B. 使用你喜欢的浏览器向 TINY 发送一个对静态内容的请求。把 TINY 的输出记录到一个文件中。

C. 检查 TINY 的输出，确定你的浏览器使用的 HTTP 的版本。

D. 参考 RFC 2616 中的 HTTP/1.1 标准，确定你的浏览器的 HTTP 请求中每个报头的含义。你可以从 `www.rfc-editor.org/rfc.html` 获得 RFC 2616。

** 11.7 扩展 TINY，使得它可以提供 MPG 视频文件。用一个真正的浏览器来检验你的工作。

** 11.8 修改 TINY，使得它在 SIGCHLD 处理程序中回收操作系统分配给 CGI 子进程的资源，而不是显式地等待它们终止。

** 11.9 修改 TINY，使得当它服务静态内容时，使用 `malloc`、`rio_readn` 和 `rio_writen`，而不是 `mmap` 和 `rio_writen`，来拷贝被请求文件到已连接描述符。

** 11.10 A. 写出图 11-26 中 CGI `adder` 函数的 HTML 表单。你的表单应该包括两个文本框，用户将需要相加的两个数字填在这两个文本框中。你的表单应该使用 GET 方法请求内容。

B. 用这样的方法来检查你的程序：使用一个真正的浏览器向 TINY 请求表单，向 TINY 提交填写好的表单，然后显示 `adder` 生成的动态内容。

** 11.11 扩展 TINY，以支持 HTTP HEAD 方法。使用 TELNET 作为 Web 客户端来验证你的工作。

** 11.12 扩展 TINY，使得它服务以 HTTP POST 方式请求的动态内容。用你喜欢的 Web 浏览器来验证你的工作。

** 11.13 修改 TINY，使得它可以干净地处理（而不是终止）在 `write` 函数试图写一个过早关闭的连接时发生的 SIGPIPE 信号和 EPIPE 错误。

练习题答案

练习题 11.1

十六进制地址	点分十进制地址
0x0	0.0.0.0
0xffffffff	255.255.255.255
0x7f000001	127.0.0.1
0xcdbca079	205.188.160.121
0x400c950d	64.12.149.13
0xcdbc9217	205.188.146.23

练习题 11.2

code/netp/hex2dd.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct in_addr inaddr;  /* addr in network byte order */
    unsigned int   addr;    /* addr in host byte order */

    if (argc != 2) {
        fprintf(stderr, "usage: %s <hex number>\n", argv[0]);
        exit(0);
    }
    sscanf(argv[1], "%x", &addr);
```

```
    inaddr.s_addr = htonl(addr);
    printf("%s\n", inet_ntoa(inaddr));

    exit(0);
}
```

code/netp/hex2dd.c

练习题 11.3

code/netp/dd2hex.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct in_addr inaddr;  /* addr in network byte order */
    unsigned int   addr;    /* addr in host byte order */

    if (argc != 2) {
        fprintf(stderr, "usage: %s <dotted-decimal>\n", argv[0]);
        exit(0);
    }

    if (inet_aton(argv[1], &inaddr) == 0)
        app_error("inet_aton error");
    addr = ntohl(inaddr.s_addr);
    printf("0x%x\n", addr);

    exit(0);
}
```

code/netp/dd2hex.c

练习题 11.4 每次我们请求 google.com 的主机条目时，相应的因特网地址列表以一种不同的、轮转（round-robin）的顺序返回。

```
unix> ./hostinfo google.com
official hostname: google.com
address: 74.125.127.100
address: 74.125.45.100
address: 74.125.67.100

unix> ./hostinfo google.com
official hostname: google.com
address: 74.125.67.100
address: 74.125.127.100
address: 74.125.45.100

unix> ./hostinfo google.com
official hostname: google.com
address: 74.125.45.100
address: 74.125.67.100
address: 74.125.127.100
```

在不同的 DNS 查询中，返回地址的不同顺序称为 DNS 轮转（DNS round-robin）。它可以用来对一个大量使用的域名的请求做负载平衡。

练习题 11.5 标准 I/O 能在 CGI 程序里工作的原因是，在子进程中运行的 CGI 程序不需要显式地关闭它的输入输出流。当子进程终止时，内核会自动关闭所有描述符。

www.TopSage.com

第 12 章
Computer Systems：A Programmer's Perspective，2E

并发编程

正如我们在第 8 章学到的，如果逻辑控制流在时间上重叠，那么它们就是*并发的*（concurrent）。这种常见的现象称为*并发*（concurrency），出现在计算机系统的许多不同层面上。硬件异常处理程序、进程和 Unix 信号处理程序都是大家很熟悉的例子。

到目前为止，我们主要将并发看做是一种操作系统内核用来运行多个应用程序的机制。但是，并发不仅仅局限于内核。它也可以在应用程序中扮演重要角色。例如，我们已经看到 Unix 信号处理程序如何允许应用响应异步事件，例如用户键入 `ctrl-c`，或者程序访问虚拟存储器的一个未定义的区域。应用级并发在其他情况下也是很有用的：

- *访问慢速 I/O 设备*。当一个应用正在等待来自慢速 I/O 设备（例如磁盘）的数据到达时，内核会运行其他进程，使 CPU 保持繁忙。每个应用都可以按照类似的方式，通过交替执行 I/O 请求和其他有用的工作来使用并发。
- *与人交互*。和计算机交互的人要求计算机有同时执行多个任务的能力。例如，他们在打印一个文档时，可能想要调整一个窗口的大小。现代视窗系统利用并发来提供这种能力。每次用户请求某种操作（比如通过单击鼠标）时，一个独立的并发逻辑流被创建来执行这个操作。
- *通过推迟工作以降低延迟*。有时，应用程序能够通过推迟其他操作和并发地执行它们，利用并发来降低某些操作的延迟。比如，一个动态存储分配器可以通过推迟合并，把它放到一个运行在较低优先级上的并发“合并”流中，在有空闲的 CPU 周期时充分利用这些空闲周期，从而降低单个 `free` 操作的延迟。
- *服务多个网络客户端*。我们在第 11 章中学习的迭代网络服务器是不现实的，因为它们一次只能为一个客户端提供服务。因此，一个慢速的客户端可能会导致服务器拒绝为所有其他客户端服务。对于一个真正的服务器来说，可能期望它每秒为成百上千的客户端提供服务，由于一个慢速客户端导致拒绝为其他客户端服务，这是不能接受的。一个更好的方法是创建一个并发服务器，它为每个客户端创建一个单独的逻辑流。这就允许服务器同时为多个客户端服务，并且这也避免了慢速客户端独占服务器。
- *在多核机器上进行并行计算*。许多现代系统都配备有多核处理器，多核处理器中包含多个 CPU。被划分成并发流的应用程序通常在多核机器上比在单处理器机器上运行得快，因为这些流会并行执行，而不是交错执行。

使用应用级并发的应用程序称为*并发程序*（concurrent program）。现代操作系统提供了三种基本的构造并发程序的方法：

- *进程*。用这种方法，每个逻辑控制流都是一个进程，由内核来调度和维护。因为进程有独立的虚拟地址空间，想要和其他流通信，控制流必须使用某种显式的*进程间通信*（interprocess communication，IPC）机制。
- *I/O 多路复用*。在这种形式的并发编程中，应用程序在一个进程的上下文中显式地调度它们自己的逻辑流。逻辑流被模型化为状态机，数据到达文件描述符后，主程序显式地从一个状态转换到另一个状态。因为程序是一个单独的进程，所以所有的流都共享同一个地址空间。
- *线程*。线程是运行在一个单一进程上下文中的逻辑流，由内核进行调度。你可以把线程看

www.TopSage.com

成是其他两种方式的混合体，像进程流一样由内核进行调度，而像 I/O 多路复用流一样共享同一个虚拟地址空间。

本章研究这三种不同的并发编程技术。为了使讨论更加具体，我们始终以同一个应用为例——11.4.9 节中的迭代 echo 服务器的并发版本。

12.1 基于进程的并发编程

构造并发程序最简单的方法就是用进程，使用那些大家都很熟悉的函数，像 fork、exec 和 waitpid。例如，一个构造并发服务器的自然方法就是，在父进程中接受客户端连接请求，然后创建一个新的子进程来为每个新客户端提供服务。

为了了解这是如何工作的，假设我们有两个客户端和一个服务器，服务器正在监听一个监听描述符（比如描述符 3）上的连接请求。现在假设服务器接受了客户端 1 的连接请求，并返回一个已连接描述符（比如描述符 4），如图 12-1 所示。

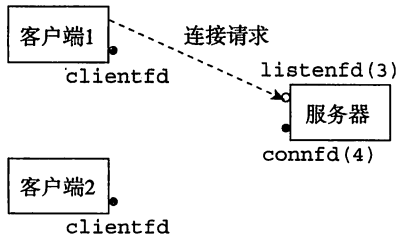

图 12-1 第一步：服务器接受客户端的连接请求

在接受连接请求之后，服务器派生一个子进程，这个子进程获得服务器描述符表的完整拷贝。子进程关闭它的拷贝中的监听描述符 3，而父进程关闭它的已连接描述符 4 的拷贝，因为不再需要这些描述符了。这就得到了图 12-2 中的状态，其中子进程正忙于为客户端提供服务。因为父、子进程中的已连接描述符都指向同一个文件表表项，所以父进程关闭它的已连接描述符的拷贝是至关重要的。否则，将永远不会释放已连接描述符 4 的文件表条目，而且由此引起的存储器泄漏将最终消耗尽可用的存储器，使系统崩溃。

现在，假设在父进程为客户端 1 创建了子进程之后，它接受一个新的客户端 2 的连接请求，并返回一个新的已连接描述符（比如描述符 5），如图 12-3 所示。然后，父进程又派生另一个子进程，这个子进程用已连接描述符 5 为它的客户端提供服务，如图 12-4 所示。此时，父进程正在等待下一个连接请求，而两个子进程正在并发地为它们各自的客户端提供服务。

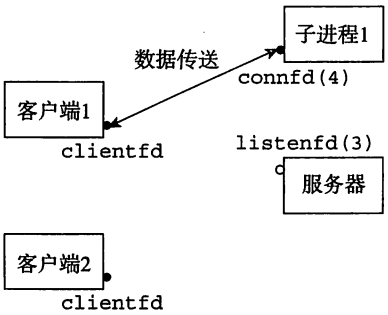

图 12-2 第二步：服务器派生一个子进程为这个客户端服务

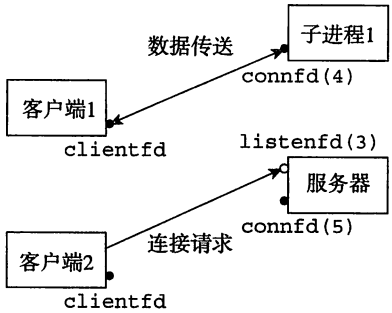

图 12-3 第三步：服务器接受另一个连接请求

12.1.1 基于进程的并发服务器

图 12-5 展示了一个基于进程的并发 echo 服务器的代码。第 29 行调用的 echo 函数来自于图 11-21。关于这个服务器，有几点重要内容需要说明：

www.TopSage.com

- 首先，通常服务器会运行很长的时间，所以我们必须要包括一个 SIGCHLD 处理程序，来回收僵死（zombie）子进程的资源（第 4 ～ 9 行）。因为当 SIGCHLD 处理程序执行时，SIGCHLD 信号是阻塞的，而 Unix 信号是不排队的，所以 SIGCHLD 处理程序必须准备好回收多个僵死子进程的资源。
- 其次，父子进程必须关闭它们各自的 connfd（分别为第 33 行和第 30 行）拷贝。就像我们已经提到过的，这对父进程而言尤为重要，它必须关闭它的已连接描述符，以避免存储器泄漏。
- 最后，因为套接字的文件表表项中的引用计数，直到父子进程的 connfd 都关闭了，到客户端的连接才会终止。

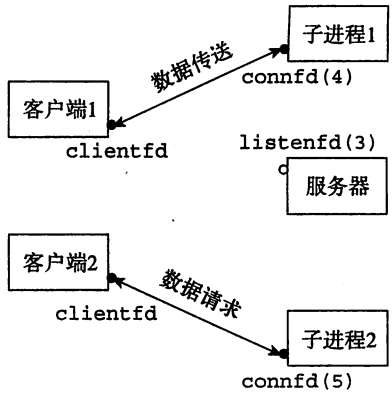

图 12-4 第四步：服务器派生另一个子进程为新的客户端服务

code/conc/echoserverp.c

```
1  #include "csapp.h"
2  void echo(int connfd);
3
4  void sigchld_handler(int sig)
5  {
6      while (waitpid(-1, 0, WNOHANG) > 0)
7          ;
8      return;
9  }
10
11 int main(int argc, char **argv)
12 {
13     int listenfd, connfd, port;
14     socklen_t clientlen=sizeof(struct sockaddr_in);
15     struct sockaddr_in clientaddr;
16
17     if (argc != 2) {
18         fprintf(stderr, "usage: %s <port>\n", argv[0]);
19         exit(0);
20     }
21     port = atoi(argv[1]);
22
23     Signal(SIGCHLD, sigchld_handler);
24     listenfd = Open_listenfd(port);
25     while (1) {
26         connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
27         if (Fork() == 0) {
28             Close(listenfd); /* Child closes its listening socket */
29             echo(connfd);    /* Child services client */
30             Close(connfd);   /* Child closes connection with client */
31             exit(0);         /* Child exits */
32         }
33         Close(connfd); /* Parent closes connected socket (important!) */
34     }
35 }
```

code/conc/echoserverp.c

图 12-5 基于进程的并发 echo 服务器。父进程派生一个子进程来处理每个新的连接请求

www.TopSage.com

12.1.2 关于进程的优劣

对于在父、子进程间共享状态信息，进程有一个非常清晰的模型：共享文件表，但是不共享用户地址空间。进程有独立的地址空间既是优点也是缺点。这样一来，一个进程不可能不小心覆盖另一个进程的虚拟存储器，这就消除了许多令人迷惑的错误——这是一个明显的优点。

另一方面，独立的地址空间使得进程共享状态信息变得更加困难。为了共享信息，它们必须使用显式的 IPC（进程间通信）机制。基于进程的设计的另一个缺点是，它们往往比较慢，因为进程控制和 IPC 的开销很高。

Unix IPC

在本书中，你已经遇到好几个 IPC 的例子了。第 8 章中的 waitpid 函数和 Unix 信号是基本的 IPC 机制，它们允许进程发送小消息到同一主机上的其他进程。第 11 章的套接字接口是 IPC 的一种重要形式，它允许不同主机上的进程交换任意的字节流。然而，术语 Unix IPC 通常指的是所有允许进程和同一台主机上其他进程进行通信的技术。其中包括管道、先进先出（FIFO）、系统 V 共享存储器，以及系统 V 信号量（semaphore）。这些机制超出了我们的讨论范围。Stevens 的著作 [108] 是很好的参考资料。

练习题 12.1 在图 12-5 中，并发服务器的第 33 行，父进程关闭了已连接描述符后，子进程仍然能够使用该描述符和客户端通信。这是为什么？

练习题 12.2 如果我们要删除图 12-5 中关闭已连接描述符的第 30 行，从没有存储器泄漏的角度来说，代码将仍然是正确的。这是为什么？

12.2 基于 I/O 多路复用的并发编程

假设要求你编写一个 echo 服务器，它也能对用户从标准输入键入的交互命令做出响应。在这种情况下，服务器必须响应两个互相独立的 I/O 事件：1）网络客户端发起连接请求，2）用户在键盘上键入命令行。我们先等待哪个事件呢？没有哪个选择是理想的。如果在 accept 中等待一个连接请求，我们就不能响应输入的命令。类似地，如果在 read 中等待一个输入命令，我们就不能响应任何连接请求。

针对这种困境的一个解决办法就是 I/O 多路复用（I/O multiplexing）技术。基本的思路就是使用 select 函数，要求内核挂起进程，只有在一个或多个 I/O 事件发生后，才将控制返回给应用程序，就像在下面的示例中一样：

- 当集合 {0，4} 中任意描述符准备好读时返回。
- 当集合 {1，2，7} 中任意描述符准备好写时返回。
- 如果在等待一个 I/O 事件发生时过了 152.13 秒，就超时。

select 是一个复杂的函数，有许多不同的使用场景。我们将只讨论第一种场景：等待一组描述符准备好读。全面的讨论请参考 [109，110]。

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

int select(int n, fd_set *fdset, NULL, NULL, NULL);
```

返回已准备好的描述符的非零的个数，若出错则为 −1。

```
FD_ZERO(fd_set *fdset);     /* Clear all bits in fdset */
FD_CLR(int fd, fd_set *fdset); /* Clear bit fd in fdset */
FD_SET(int fd, fd_set *fdset);  /* Turn on bit fd in fdset */
FD_ISSET(int fd, fd_set *fdset); /* Is bit fd in fdset on? */
```

处理描述符集合的宏。

www.TopSage.com

select 函数处理类型为 fd_set 的集合，也叫做描述符集合。逻辑上，我们将描述符集合看成一个大小为 n 的位向量（在 2.1 节中介绍过的）：

$$b_{n-1}, \cdots, b_1, b_0$$

每个位 b_k 对应于描述符 k。当且仅当 $b_k = 1$，描述符 k 才表明是描述符集合的一个元素。只允许你对描述符集合做三件事：1）分配它们，2）将一个此种类型的变量赋值给另一个变量，3）用 FD_ZERO、FD_SET、FD_CLR 和 FD_ISSET 宏指令来修改和检查它们。

针对我们的目的，select 函数有两个输入：一个称为*读集合*的描述符集合（fdset）和该读集合的基数（n）（实际上是任何描述符集合的最大基数）。select 函数会一直阻塞，直到读集合中至少有一个描述符准备好可以读。当且仅当一个从该描述符读取一个字节的请求不会阻塞时，描述符 k 就表示*准备好可以读*了。作为一个副作用，select 修改了参数 fdset 指向的 fd_set，指明读集合中一个称为*准备好集合*（ready set）的子集，这个集合是由读集合中准备好可以读了的描述符组成的。函数返回的值指明了准备好集合的基数。注意，由于这个副作用，我们必须在每次调用 select 时都更新读集合。

code/conc/select.c

```
#include "csapp.h"
void echo(int connfd);
void command(void);

int main(int argc, char **argv)
{
    int listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    fd_set read_set, ready_set;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);
    listenfd = Open_listenfd(port);

    FD_ZERO(&read_set);              /* Clear read set */
    FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
    FD_SET(listenfd, &read_set);     /* Add listenfd to read set */

    while (1) {
        ready_set = read_set;
        Select(listenfd+1, &ready_set, NULL, NULL, NULL);
        if (FD_ISSET(STDIN_FILENO, &ready_set))
            command(); /* Read command line from stdin */
        if (FD_ISSET(listenfd, &ready_set)) {
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            echo(connfd); /* Echo client input until EOF */
            Close(connfd);
        }
    }
}

void command(void) {
    char buf[MAXLINE];
    if (!Fgets(buf, MAXLINE, stdin))
        exit(0); /* EOF */
    printf("%s", buf); /* Process the input command */
}
```

code/conc/select.c

图 12-6　使用 I/O 多路复用的 echo 服务器。服务器使用 select 等待监听描述符上的连接请求和标准输入上的命令

www.TopSage.com

理解 select 的最好办法是研究一个具体例子。图 12-6 展示了如何利用 select 来实现一个迭代 echo 服务器，它也可以接受标准输入上的用户命令。一开始，我们用图 11-7 中的 open_listenfd 函数打开一个监听描述符（第 17 行），然后使用 FD_ZERO 创建一个空的读集合（第 19 行）：

	listenfd			stdin
	3	2	1	0
read_set (∅):	0	0	0	0

接下来，在第 20 和 21 行中，我们定义由描述符 0（标准输入）和描述符 3（监听描述符）组成的读集合：

	listenfd			stdin
	3	2	1	0
read_set ({0, 3}):	1	0	0	1

在这里，我们开始典型的服务器循环。但是我们不是调用 accept 函数来等待一个连接请求，而是调用 select 函数，这个函数会一直阻塞，直到监听描述符或者标准输入准备好可以读（第 25 行）。例如，下面是当用户按回车键，因此使得标准输入描述符变为可读时，select 会返回的 ready_set 的值：

	listenfd			stdin
	3	2	1	0
ready_set ({0}):	0	0	0	1

一旦 select 返回，我们就用 FD_ISSET 宏指令来判断哪个描述符准备好可以读了。如果是标准输入准备好了（第 26 行），我们就调用 command 函数，该函数在返回到主程序前，会读、解析和响应命令。如果是监听描述符准备好了（第 28 行），我们就调用 accept 来得到一个已连接描述符，然后调用图 12-21 中的 echo 函数，它会将来自客户端的每一行又回送回去，直到客户端关闭这个连接中它的那一端。

虽然这个程序是使用 select 的一个很好示例，但是它仍然留下了一些问题待解决。问题是一旦它连接到某个客户端，就会连续回送输入行，直到客户端关闭这个连接中它的那一端。因此，如果你键入一个命令到标准输入，你将不会得到响应，直到服务器和客户端之间结束。一个更好的方法是更细粒度的多路复用，服务器每次循环（至多）回送一个文本行。

练习题 12.3 在大多数的 Unix 系统里，在标准输入上键入 ctrl-d 表示 EOF。在图 12.6 中的程序阻塞在对 select 的调用上时，如果你键入 ctrl-d 会发生什么？

12.2.1 基于 I/O 多路复用的并发事件驱动服务器

I/O 多路复用可以用做并发事件驱动（event-driven）程序的基础，在事件驱动程序中，流是因为某种事件而前进的。一般概念是将逻辑流模型化为状态机。不严格地说，一个*状态机*（state machine）就是一组*状态*（state）、*输入事件*（input event）和*转移*（transition），其中转移就是将状态和输入事件映射到状态。每个转移都将一个（输入状态，输入事件）对映射到一个输出状态。*自循环*（self-loop）是同一输入和输出状态之间的转移。通常把状态机画成有向图，其中节点表示状态，有向弧表示转移，而弧上的标号表示输入事件。一个状态机从某种初始状态开始执行。每个输入事件都会引发一个从当前状态到下一状态的转移。

对于每个新的客户端 k，基于 I/O 多路复用的并发服务器会创建一个新的状态机 s_k，并将它和

www.TopSage.com

已连接描述符 d_k 联系起来。如图 12-7 所示，每个状态机 s_k 都有一个状态（“等待描述符 d_k 准备好可读”）、一个输入事件（“描述符 d_k 准备好可以读了”）和一个转移（“从描述符 d_k 读一个文本行”）。

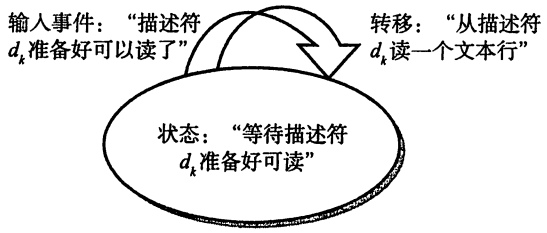

图 12-7　并发事件驱动 echo 服务器中逻辑流的状态机

服务器使用 I/O 多路复用，借助 select 函数检测输入事件的发生。当每个已连接描述符准备好可读时，服务器就为相应的状态机执行转移，在这里就是从描述符读和写回一个文本行。

图 12-8 展示了一个基于 I/O 多路复用的并发事件驱动服务器的完整示例代码。活动客户端的集合维护在一个 pool（池）结构里（第 3 ～ 11 行）。在通过调用 init_pool 初始化池（第 29 行）之后，服务器进入一个无限循环。在循环的每次迭代中，服务器调用 select 函数来检测两种不同类型的输入事件：a）来自一个新客户端的连接请求到达，b）一个已存在的客户端的已连接描述符准备好可以读了。当一个连接请求到达时（第 36 行），服务器打开连接（第 37 行），并调用 add_client 函数，将该客户端添加到池里（第 38 行）。最后，服务器调用 check_clients 函数，把来自每个准备好的已连接描述符的一个文本行回送回去（第 42 行）。

code/conc/echoservers.c

```
1   #include "csapp.h"
2
3   typedef struct { /* Represents a pool of connected descriptors */
4       int maxfd;          /* Largest descriptor in read_set */
5       fd_set read_set;    /* Set of all active descriptors */
6       fd_set ready_set;   /* Subset of descriptors ready for reading  */
7       int nready;         /* Number of ready descriptors from select */
8       int maxi;           /* Highwater index into client array */
9       int clientfd[FD_SETSIZE];     /* Set of active descriptors */
10      rio_t clientrio[FD_SETSIZE];  /* Set of active read buffers */
11  } pool;
12
13  int byte_cnt = 0; /* Counts total bytes received by server */
14
15  int main(int argc, char **argv)
16  {
17      int listenfd, connfd, port;
18      socklen_t clientlen = sizeof(struct sockaddr_in);
19      struct sockaddr_in clientaddr;
20      static pool pool;
21
22      if (argc != 2) {
23          fprintf(stderr, "usage: %s <port>\n", argv[0]);
24          exit(0);
25      }
26      port = atoi(argv[1]);
27
28      listenfd = Open_listenfd(port);
29      init_pool(listenfd, &pool);
```

图 12-8　基于 I/O 多路复用的并发 echo 服务器。每次服务器迭代都回送来自每个准备好的描述符的文本行

www.TopSage.com

```
    while (1) {
        /* Wait for listening/connected descriptor(s) to become ready */
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);

        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }

        /* Echo a text line from each ready connected descriptor */
        check_clients(&pool);
    }
}
```

code/conc/echoservers.c

图 12-8 （续）

init_pool 函数（见图 12-9）初始化客户端池。clientfd 数组表示已连接描述符的集合，其中整数 -1 表示一个可用的槽位。初始时，已连接描述符集合是空的（第 5 ～ 7 行），而且监听描述符是 select 读集合中唯一的描述符（第 10 ～ 12 行）。

code/conc/echoservers.c

```
1  void init_pool(int listenfd, pool *p)
2  {
3      /* Initially, there are no connected descriptors */
4      int i;
5      p->maxi = -1;
6      for (i=0; i< FD_SETSIZE; i++)
7          p->clientfd[i] = -1;
8
9      /* Initially, listenfd is only member of select read set */
10     p->maxfd = listenfd;
11     FD_ZERO(&p->read_set);
12     FD_SET(listenfd, &p->read_set);
13 }
```

code/conc/echoservers.c

图 12-9 init_pool：初始化活动客户端池

add_client 函数（见图 12-10）添加一个新的客户端到活动客户端池中。在 clientfd 数组中找到一个空槽位后，服务器将这个已连接描述符添加到数组中，并初始化相应的 RIO 读缓冲区，这样一来我们就能够对这个描述符调用 rio_readlineb（第 8 ～ 9 行）。然后，我们将这个已连接描述符添加到 select 读集合（第 12 行），并更新该池的一些全局属性。maxfd 变量（第 15 ～ 16 行）记录了 select 的最大文件描述符。maxi 变量（第 17 ～ 18 行）记录的是到 clientfd 数组的最大索引，这样 check_clients 函数就无需搜索整个数组了。

check_clients 函数（见图 12-11）回送来自每个准备好的已连接描述符的一个文本行。如果成功地从描述符读取了一个文本行，那么我们就将该文本行回送到客户端（第 15 ～ 18 行）。注意，在第 15 行我们维护着一个从所有客户端接收到的全部字节的累计值。如果因为客户端关闭这个连接中它的那一端，检测到 EOF，那么我们将关闭这边的连接端（第 23 行），并从池中清除掉这个描述符（第 24 ～ 25 行）。

www.TopSage.com

大家网 TopSage.com

code/conc/echoservers.c

```
void add_client(int connfd, pool *p)
{
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++)  /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);

            /* Update max descriptor and pool highwater mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
```

code/conc/echoservers.c

图 12-10 add_client：向池中添加一个新的客户端连接

code/conc/echoservers.c

```
void check_clients(pool *p)
{
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;

    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                printf("Server received %d (%d total) bytes on fd %d\n",
                       n, byte_cnt, connfd);
                Rio_writen(connfd, buf, n);
            }

            /* EOF detected, remove descriptor from pool */
            else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

code/conc/echoservers.c

图 12-11 check_clients：为准备好的客户端连接服务

根据图 12-7 中的有限状态模型，select 函数检测到输入事件，而 add_client 函数创建一个新的逻辑流（状态机）。check_clients 函数通过回送输入行来执行状态转移，而且当客户端完成文本行发送时，它还要删除这个状态机。

12.2.2 I/O 多路复用技术的优劣

图 12-8 中的服务器提供了一个很好的基于 I/O 多路复用的事件驱动编程的优缺点示例。事件驱动设计的一个优点是，它比基于进程的设计给了程序员更多的对程序行为的控制。例如，我们可以设想编写一个事件驱动的并发服务器，为某些客户端提供它们需要的服务，而这对于基于进程的并发服务器来说，是很困难的。

另一个优点是，一个基于 I/O 多路复用的事件驱动服务器是运行在单一进程上下文中的，因此每个逻辑流都能访问该进程的全部地址空间。这使得在流之间共享数据变得很容易。一个与作为单个进程运行相关的优点是，你可以利用熟悉的调试工具，例如 GDB，来调试你的并发服务器，就像对顺序程序那样。最后，事件驱动设计常常比基于进程的设计要高效得多，因为它们不需要进程上下文切换来调度新的流。

事件驱动设计的一个明显的缺点就是编码复杂。我们的事件驱动的并发 echo 服务器需要的代码比基于进程的服务器多三倍。不幸的是，随着并发粒度的减小，复杂性还会上升。这里的*粒度*是指每个逻辑流每个时间片执行的指令数量。例如，在我们的示例并发服务器中，并发粒度就是读一个完整的文本行所需要的指令数量。只要某个逻辑流正忙于读一个文本行，其他逻辑流就不可能有进展。对我们的例子而言这就很好了，但是它使得我们的事件驱动服务器在“故意只发送部分文本行然后就停止”的恶意客户端的攻击面前显得很脆弱。修改事件驱动服务器来处理部分文本行不是一个简单的任务，但是基于进程的设计却能处理得很好，而且是自动处理的。基于事件的设计的另一个重大缺点是它们不能充分利用多核处理器。

练习题 12.4 在如图 12-8 所示的服务器中，我们在每次调用 select 之前都立即小心地重新初始化 pool.ready_set 变量。这是为什么？

12.3 基于线程的并发编程

到目前为止，我们已经看到了两种创建并发逻辑流的方法。在第一种方法中，我们为每个流使用了单独的进程。内核会自动调度每个进程。每个进程有它自己的私有地址空间，这使得流共享数据很困难。在第二种方法中，我们创建自己的逻辑流，并利用 I/O 多路复用来显式地调度流。因为只有一个进程，所有的流共享整个地址空间。这一节介绍第三种方法——基于线程，它是这两种方法的混合。

线程（thread）就是运行在进程上下文中的逻辑流。迄今在本书里，程序都是由每个进程中一个线程组成的。但是现代系统也允许我们编写一个进程里同时运行多个线程的程序。线程由内核自动调度。每个线程都有它自己的*线程上下文*（thread context），包括一个唯一的整数*线程 ID*（Thread ID，TID）、栈、栈指针、程序计数器、通用目的寄存器和条件码。所有的运行在一个进程里的线程共享该进程的整个虚拟地址空间。

基于线程的逻辑流结合了基于进程和基于 I/O 多路复用的流的特性。同进程一样，线程由内核自动调度，并且内核通过一个整数 ID 来识别线程。同基于 I/O 多路复用的流一样，多个线程运行在单一进程的上下文中，因此共享这个进程虚拟地址空间的整个内容，包括它的代码、数据、堆、共享库和打开的文件。

12.3.1 线程执行模型

多线程的执行模型在某些方面和多进程的执行模型是相似的。思考一下图 12-12 中的示例。每个进程开始生命周期时都是单一线程，这个线程称为*主线程*（main thread）。在某一时刻，主

www.TopSage.com

线程创建一个对等线程（peer thread），从这个时间点开始，两个线程就并发地运行。最后，因为主线程执行一个慢速系统调用，例如 read 或 sleep，或者因为它被系统的间隔计时器中断，控制就会通过上下文切换传递到对等线程。对等线程会执行一段时间，然后控制传递回主线程，依次类推。

在一些重要的方面，线程执行是不同于进程的。因为一个线程的上下文要比一个进程的上下文小得多，线程的上下文切换要比进程的上下文切换快得多。另一个不同就是线程不像进程那样，不是按照严格的父子层次来组织的。和一个进程相关的线程组成一个对等（线程）池（pool），独立于其他线程创建的线程。主线程和其他线程的区别仅在于它总是进程中第一个运行的线程。对等（线程）池概念的主要影响是，一个线程可以杀死它的任何对等线程，或者等待它的任意对等线程终止。另外，每个对等线程都能读写相同的共享数据。

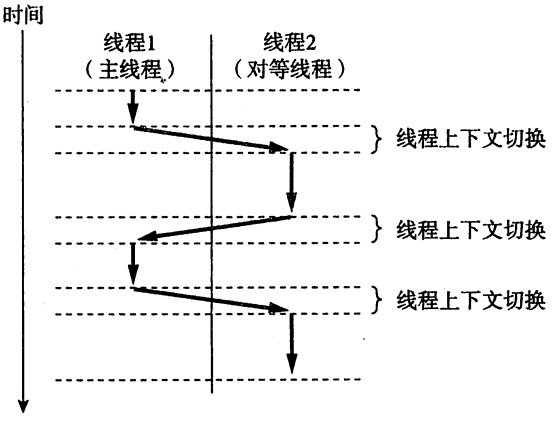

图 12-12　并发线程执行

12.3.2　Posix 线程

Posix 线程（Pthreads）是在 C 程序中处理线程的一个标准接口。它最早出现在 1995 年，而且在大多数 Unix 系统上都可用。Pthreads 定义了大约 60 个函数，允许程序创建、杀死和回收线程，与对等线程安全地共享数据，还可以通知对等线程系统状态的变化。

图 12-13 展示了一个简单的 Pthreads 程序。主线程创建一个对等线程，然后等待它的终止。对等线程输出“Hello, world!\n”并且终止。当主线程检测到对等线程终止后，它就通过调用 exit 终止该进程。

code/conc/hello.c

```
1   #include "csapp.h"
2   void *thread(void *vargp);
3
4   int main()
5   {
6       pthread_t tid;
7       Pthread_create(&tid, NULL, thread, NULL);
8       Pthread_join(tid, NULL);
9       exit(0);
10  }
11
12  void *thread(void *vargp) /* Thread routine */
13  {
14      printf("Hello, world!\n");
15      return NULL;
16  }
```

code/conc/hello.c

图 12-13　hello.c：Pthreads“Hello, world!”程序

这是我们看到的第一个线程化的程序，所以让我们仔细地解析它。线程的代码和本地数据被封装在一个线程例程（thread routine）中。正如第二行里的原型所示，每个线程例程都以一个通

www.TopSage.com

用指针作为输入，并返回一个通用指针。如果想传递多个参数给线程例程，那么你应该将参数放到一个结构中，并传递一个指向该结构的指针。相似地，如果你想要线程例程返回多个参数，你可以返回一个指向一个结构的指针。

第 4 行标出了主线程代码的开始。主线程声明了一个本地变量 tid，它可以用来存放对等线程的线程 ID（第 6 行）。主线程通过调用 pthread_create 函数创建一个新的对等线程（第 7 行）。当对 pthread_create 的调用返回时，主线程和新创建的对等线程同时运行，并且 tid 包含新线程的 ID。通过调用 pthread_join，主线程等待对等线程终止（第 8 行）。最后，主线程调用 exit（第 9 行），终止当时运行在这个进程中的所有线程（在这个示例中就只有主线程）。

第 12 ～ 16 行定义了对等线程的线程例程。它只打印一个字符串，然后就通过执行第 15 行中的 return 语句来终止对等线程。

12.3.3 创建线程

线程通过调用 pthread_create 函数来创建其他线程。

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   func *f, void *arg);
```

返回：若成功则返回 0，若出错则为非零。

pthread_create 函数创建一个新的线程，并带着一个输入变量 arg，在新线程的上下文中运行线程例程 f。能用 attr 参数来改变新创建线程的默认属性。改变这些属性已超出我们学习的范围，并且在我们的示例中，我们总是用一个为 NULL 的 attr 参数来调用 pthread_create 函数。

当 pthread_create 返回时，参数 tid 包含新创建线程的 ID。新线程可以通过调用 pthread_self 函数来获得它自己的线程 ID。

```
#include <pthread.h>

pthread_t pthread_self(void);
```

返回：返回调用者的线程 ID。

12.3.4 终止线程

一个线程是以下列方式之一来终止的：

- 当顶层的线程例程返回时，线程会隐式地终止。
- 通过调用 pthread_exit 函数，线程会显式地终止。如果主线程调用 pthread_exit，它会等待所有其他对等线程终止，然后再终止主线程和整个进程，返回值为 thread_return。

```
#include <pthread.h>

void pthread_exit(void *thread_return);
```

返回：若成功则返回 0，若出错则为非零。

- 某个对等线程调用 Unix 的 exit 函数，该函数终止进程以及所有与该进程相关的线程。
- 另一个对等线程通过以当前线程 ID 作为参数调用 pthread_cancle 函数来终止当前线程。

www.TopSage.com

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

返回：若成功则返回0，若出错则为非零。

12.3.5 回收已终止线程的资源

线程通过调用 pthread_join 函数等待其他线程终止。

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```

返回：若成功则返回0，若出错则为非零。

pthread_join 函数会阻塞，直到线程 tid 终止，将线程例程返回的（void*）指针赋值为 thread_return 指向的位置，然后回收已终止线程占用的所有存储器资源。

注意，和 Unix 的 wait 函数不同，pthread_join 函数只能等待一个指定的线程终止。没有办法让 pthread_wait 等待任意一个线程终止。这使得代码更加复杂，因为它迫使我们去使用其他一些不那么直观的机制来检测进程的终止。实际上，Stevens 在 [109] 中就很有说服力地论证了这是规范中的一个错误。

12.3.6 分离线程

在任何一个时间点上，线程是*可结合的*（joinable）或者是*分离的*（detached）。一个可结合的线程能够被其他线程收回其资源和杀死。在被其他线程回收之前，它的存储器资源（例如栈）是没有被释放的。相反，一个分离的线程是不能被其他线程回收或杀死的。它的存储器资源在它终止时由系统自动释放。

默认情况下，线程被创建成可结合的。为了避免存储器泄漏，每个可结合线程都应该要么被其他线程显式地收回，要么通过调用 pthread_detach 函数被分离。

pthread_detach 函数分离可结合线程 tid。线程能够通过以 pthread_self() 为参数的 pthread_detach 调用来分离它们自己。

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

返回：若成功则返回0，若出错则为非零。

尽管我们的一些例子会使用可结合线程，但是在现实程序中，有很好的理由要使用分离的线程。例如，一个高性能 Web 服务器可能在每次收到 Web 浏览器的连接请求时都创建一个新的对等线程。因为每个连接都是由一个单独的线程独立处理的，所以对于服务器而言，就很没有必要（实际上也不愿意）显式地等待每个对等线程终止。在这种情况下，每个对等线程都应该在它开始处理请求之前分离它自身，这样就能在它终止后回收它的存储器资源了。

12.3.7 初始化线程

pthread_once 函数允许你初始化与线程例程相关的状态。

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

总是返回0。

www.TopSage.com

once_control 变量是一个全局或者静态变量，总是被初始化为 PTHREAD_ONCE_INIT。当你第一次用参数 once_control 调用 pthread_once 时，它调用 init_routine，这是一个没有输入参数，也不返回什么的函数。接下来的以 once_control 为参数的 pthread_once 调用不做任何事情。无论何时，当你需要动态初始化多个线程共享的全局变量时，pthread_once 函数是很有用的。我们将在 12.5.5 节里看到一个示例。

12.3.8 一个基于线程的并发服务器

图 12-14 展示了基于线程的并发 echo 服务器的代码。整体结构类似于基于进程的设计。主线程不断地等待连接请求，然后创建一个对等线程处理该请求。虽然代码看似简单，但是有几个普遍而且有些微妙的问题需要我们更仔细地看一看。第一个问题是当我们调用 pthread_create 时，如何将已连接描述符传递给对等线程。最明显的方法就是传递一个指向这个描述符的指针，就像下面这样

```
connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
Pthread_create(&tid, NULL, thread, &connfd);
```

然后，我们让对等线程间接引用这个指针，并将它赋值给一个局部变量，如下所示：

```
void *thread(void *vargp) {
    int connfd = *((int *)vargp);
    :
}
```

然而，这样可能会出错，因为它在对等线程的赋值语句和主线程的 accept 语句间引入了竞争（race）。如果赋值语句在下一个 accept 之前完成，那么对等线程中的局部变量 connfd 就得到正确的描述符值。然而，如果赋值语句是在 accept 之后才完成的，那么对等线程中的局部变量 connfd 就得到下一次连接的描述符值。那么不幸的结果就是，现在两个线程在同一个描述符上执行输入和输出。为了避免这种潜在的致命竞争，我们必须将每个 accept 返回的已连接描述符分配到它自己的动态分配的存储器块，如第 21~22 行所示。我们会在 12.7.4 节中回过来讨论竞争的问题。

code/conc/echoservert.c

```
1   #include "csapp.h"
2
3   void echo(int connfd);
4   void *thread(void *vargp);
5
6   int main(int argc, char **argv)
7   {
8       int listenfd, *connfdp, port;
9       socklen_t clientlen=sizeof(struct sockaddr_in);
10      struct sockaddr_in clientaddr;
11      pthread_t tid;
12
13      if (argc != 2) {
14          fprintf(stderr, "usage: %s <port>\n", argv[0]);
15          exit(0);
16      }
17      port = atoi(argv[1]);
18
```

图 12-14 基于线程的并发 echo 服务器

www.TopSage.com

```
19      listenfd = Open_listenfd(port);
20      while (1) {
21          connfdp = Malloc(sizeof(int));
22          *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
23          Pthread_create(&tid, NULL, thread, connfdp);
24      }
25  }
26
27  /* Thread routine */
28  void *thread(void *vargp)
29  {
30      int connfd = *((int *)vargp);
31      Pthread_detach(pthread_self());
32      Free(vargp);
33      echo(connfd);
34      Close(connfd);
35      return NULL;
36  }
```

code/conc/echoservert.c

图 12-14 （续）

另一个问题是在线程例程中避免存储器泄漏。既然我们不显式地收回线程，我们就必须分离每个线程，使得在它终止时它的存储器资源能够被收回（第 31 行）。更进一步，我们必须小心释放主线程分配的存储器块（第 32 行）。

练习题 12.5 在图 12-5 中基于进程的服务器中，我们在两个位置小心地关闭了已连接描述符：父进程和子进程。然而，在图 12-14 中基于线程的服务器中，我们只在一个位置关闭了已连接描述符：对等线程。这是为什么？

12.4 多线程程序中的共享变量

从一个程序员的角度来看，线程很有吸引力的一个方面就是多个线程很容易共享相同的程序变量。然而，这种共享也是很棘手的。为了编写正确的线程化程序，我们必须对所谓的共享以及它是如何工作的有很清楚的了解。

为了理解 C 程序中的一个变量是否是共享的，有一些基本的问题要解答：1）线程的基础存储器模型是什么？ 2）根据这个模型，变量实例是如何映射到存储器的？ 3）最后，有多少线程引用这些实例？一个变量是*共享*的，当且仅当多个线程引用这个变量的某个实例。

为了让我们对共享的讨论具体化，我们将使用图 12-15 中的程序作为一个运行示例。尽管有些人为的痕迹，但是它仍然值得研究，因为它说明了关于共享的许多细微之处。示例程序由一个创建了两个对等线程的主线程组成。主线程传递一个唯一的 ID 给每个对等线程，每个对等线程利用这个 ID 输出一条个性化的信息，以及调用该线程例程的总次数。

code/conc/sharing.c

```
1   #include "csapp.h"
2   #define N 2
3   void *thread(void *vargp);
4
5   char **ptr;  /* Global variable */
6
7   int main()
```

图 12-15 说明共享不同方面的示例程序

www.TopSage.com

```
8   {
9       int i;
10      pthread_t tid;
11      char *msgs[N] = {
12          "Hello from foo",
13          "Hello from bar"
14      };
15
16      ptr = msgs;
17      for (i = 0; i < N; i++)
18          Pthread_create(&tid, NULL, thread, (void *)i);
19      Pthread_exit(NULL);
20  }
21
22  void *thread(void *vargp)
23  {
24      int myid = (int)vargp;
25      static int cnt = 0;
26      printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
27      return NULL;
28  }
```

code/conc/sharing.c

图 12-15 （续）

12.4.1 线程存储器模型

一组并发线程运行在一个进程的上下文中。每个线程都有它自己独立的*线程上下文*，包括线程 ID、栈、栈指针、程序计数器、条件码和通用目的寄存器值。每个线程和其他线程一起共享进程上下文的剩余部分。这包括整个用户虚拟地址空间，它是由只读文本（代码）、读 / 写数据、堆以及所有的共享库代码和数据区域组成的。线程也共享同样的打开文件的集合。

从实际操作的角度来说，让一个线程去读或写另一个线程的寄存器值是不可能的。另一方面，任何线程都可以访问共享虚拟存储器的任意位置。如果某个线程修改了一个存储器位置，那么其他每个线程最终都能在它读这个位置时发现这个变化。因此，寄存器是从不共享的，而虚拟存储器总是共享的。

各自独立的线程栈的存储器模型不是那么整齐清楚的。这些栈被保存在虚拟地址空间的栈区域中，并且通常是被相应的线程独立地访问的。我们说通常而不是总是，是因为不同的线程栈是不对其他线程设防的。所以，如果一个线程以某种方式得到一个指向其他线程栈的指针，那么它就可以读写这个栈的任何部分。我们的示例程序在第 26 行展示了这一点，其中对等线程直接通过全局变量 ptr 间接引用主线程的栈的内容。

12.4.2 将变量映射到存储器

线程化的 C 程序中变量根据它们的存储类型被映射到虚拟存储器：

- *全局变量*。全局变量是定义在函数之外的变量。在运行时，虚拟存储器的读 / 写区域只包含每个全局变量的一个实例，任何线程都可以引用。例如，第 5 行声明的全局变量 ptr 在虚拟存储器的读 / 写区域中有一个运行时实例。当一个变量只有一个实例时，我们只用变量名（在这里就是 ptr）来表示这个实例。
- *本地自动变量*。本地自动变量就是定义在函数内部但是没有 static 属性的变量。在运行时，每个线程的栈都包含它自己的所有本地自动变量的实例。即使当多个线程执行同一个线程例程时也是如此。例如，有一个本地变量 tid 的实例，它保存在主线程的栈中。我们用 tid.m 来表示这个实例。再来看一个例子，本地变量 myid 有两个实例，一个在对等

www.TopSage.com

线程 0 的栈内，另一个在对等线程 1 的栈内。我们将这两个实例分别表示为 myid.p0 和 myid.p1。

- **本地静态变量**。本地静态变量是定义在函数内部并有 static 属性的变量。和全局变量一样，虚拟存储器的读 / 写区域只包含在程序中声明的每个本地静态变量的一个实例。例如，即使示例程序中的每个对等线程都在第 25 行声明了 cnt，在运行时，虚拟存储器的读 / 写区域中也只有一个 cnt 的实例。每个对等线程都读和写这个实例。

12.4.3 共享变量

我们说一个变量 *v* 是*共享*的，当且仅当它的一个实例被一个以上的线程引用。例如，示例程序中的变量 cnt 就是共享的，因为它只有一个运行时实例，并且这个实例被两个对等线程引用。在另一方面，myid 不是共享的，因为它的两个实例中每一个都只被一个线程引用。然而，认识到像 msgs 这样的本地自动变量也能被共享是很重要的。

练习题 12.6　A. 利用 12.4 节中的分析，为图 12-15 中的示例程序在下表的每个条目中填写“是”或者“否”。在第一列中，符号 *v.t* 表示变量 *v* 的一个实例，它驻留在线程 *t* 的本地栈中，其中 *t* 要么是 m（主线程），要么是 p0（对等线程 0）或者 p1（对等线程 1）。

变量实例	主线程引用的？	对等线程 0 引用的？	对等线程 1 引用的？
ptr			
cnt			
i.m			
msgs.m			
myid.po			
myid.p1			

B. 根据 A 部分的分析，变量 ptr、cnt、i、msgs 和 myid 哪些是共享的？

12.5 用信号量同步线程

共享变量是十分方便的，但是它们也引入了*同步错误*（synchronization error）的可能性。考虑图 12-16 中的程序 badcnt.c，它创建了两个线程，每个线程都对共享计数变量 cnt 加 1。因为每个线程都对计数器增加了 niters 次，我们预计它的最终值是 2×niters。这看上去简单而直接。然而，当在 Linux 系统上运行 badcnt.c 时，我们不仅得到错误的答案，而且每次得到的答案都还不相同！

code/conc/badcnt.c

```
#include "csapp.h"

void *thread(void *vargp);  /* Thread routine prototype */

/* Global shared variable */
volatile int cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    int niters;
    pthread_t tid1, tid2;

    /* Check input argument */
    if (argc != 2) {
```

图 12-16　badcnt.c：一个同步不正确的计数器程序

www.TopSage.com

```
15          printf("usage: %s <niters>\n", argv[0]);
16          exit(0);
17      }
18      niters = atoi(argv[1]);
19
20      /* Create threads and wait for them to finish */
21      Pthread_create(&tid1, NULL, thread, &niters);
22      Pthread_create(&tid2, NULL, thread, &niters);
23      Pthread_join(tid1, NULL);
24      Pthread_join(tid2, NULL);
25
26      /* Check result */
27      if (cnt != (2 * niters))
28          printf("BOOM! cnt=%d\n", cnt);
29      else
30          printf("OK cnt=%d\n", cnt);
31      exit(0);
32  }
33
34  /* Thread routine */
35  void *thread(void *vargp)
36  {
37      int i, niters = *((int *)vargp);
38
39      for (i = 0; i < niters; i++)
40          cnt++;
41
42      return NULL;
43  }
```

code/conc/badcnt.c

图 12-16 （续）

```
linux> ./badcnt 1000000
BOOM! cnt=1445085

linux> ./badcnt 1000000
BOOM! cnt=1915220

linux> ./badcnt 1000000
BOOM! cnt=1404746
```

那么哪里出错了呢？为了清晰地理解这个问题，我们需要研究计数器循环（第 39 ～ 40 行）的汇编代码，如图 12-17 所示。我们发现，将线程 i 的循环代码分解成五个部分是很有帮助的：

- H_i：在循环头部的指令块。
- L_i：加载共享变量 cnt 到寄存器 $\%eax_i$ 的指令，这里 $\%eax_i$ 表示线程 i 中的寄存器 %eax 的值。
- U_i：更新（增加）$\%eax_i$ 的指令。
- S_i：将 $\%eax_i$ 的更新值存回到共享变量 cnt 的指令。
- T_i：循环尾部的指令块。

注意头和尾只操作本地栈变量，而 L_i、U_i 和 S_i 操作共享计数器变量的内容。

当 badcnt.c 中的两个对等线程在一个单处理器上并发运行时，机器指令以某种顺序一个接一个地完成。因此，每个并发执行定义了两个线程中的指令的某种全序（或者交叉）。不幸的是，这些顺序中的一些将会产生正确结果，但是其他的则不会。

www.TopSage.com

大家网 TopSage.com

线程 i 的 C 代码

```
for (i=0; i <niters; i++)
    cnt++;
```

➡

线程 i 的汇编 C 代码

```
    movl (%rdi),%ecx
    movl $0,%edx
    cmpl %ecx,%edx
    jge  .L13
.L11:
    movl cnt(%rip),%eax
    incl %eax
    movl %eax,cnt(%rip)
    incl %edx
    cmpl %ecx,%edx
    jl   .L11
.L13:
```

H_i: 头

L_i: Load cnt
U_i: Update cnt
S_i: Store cnt

T_i: Tail

图 12-17　badcnt.c 中计数器循环（第 39 ～ 40 行）的汇编代码

这里有个关键点：*一般而言，你没有办法预测操作系统是否将为你的线程选择一个正确的顺序*。例如，图 12-18a 展示了一个正确的指令顺序的分步操作。在每个线程更新了共享变量 cnt 之后，它在存储器中的值就是 2，这正是期望的值。另一方面，图 12-18b 的顺序产生一个不正确的 cnt 的值。会发生这样的问题是因为，线程 2 在第 5 步加载 cnt，是在第 2 步线程 1 加载 cnt 之后，而在第 6 步线程 1 存储它的更新值之前。因此，每个线程最终都会存储一个值为 1 的更新后的计数器值。我们能够借助于一种叫做*进度图*（progress graph）的方法来阐明这些正确的和不正确的指令顺序的概念，这个图我们将在下一节中介绍。

步骤	线程	指令	$\%eax_1$	$\%eax_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

a）正确的顺序

步骤	线程	指令	$\%eax_1$	$\%eax_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

b）不正确的顺序

图 12-18　badcnt.c 中第一次循环迭代的指令顺序

练习题 12.7　根据 badcnt.c 的指令顺序完成下表：

步骤	线程	指令	$\%eax_1$	$\%eax_2$	cnt
1	1	H_1	—	—	0
2	1	L_1			
3	2	H_2			
4	2	L_2			
5	2	U_2			
6	2	S_2			
7	1	U_1			
8	1	S_1			
9	1	T_1			
10	2	T_2			

这种顺序会产生一个正确的 cnt 值吗？

www.TopSage.com

12.5.1 进度图

进度图（progress graph）将 n 个并发线程的执行模型化为一条 n 维笛卡儿空间中的轨迹线。每条轴 k 对应于线程 k 的进度。每个点（I_1，I_2，…，I_n）代表线程 k（k=1，…，n）已经完成了指令 I_k 这一状态。图的原点对应于没有任何线程完成一条指令的*初始状态*。

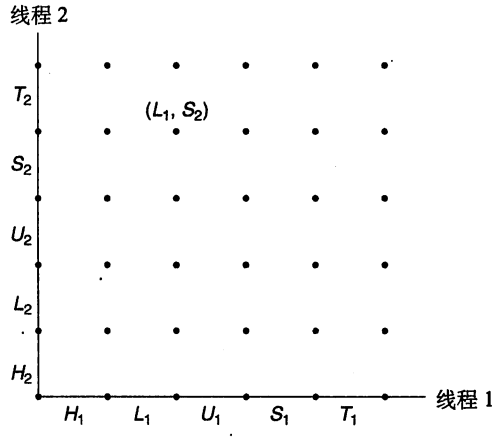

图 12-19 badcnt.c 第一次循环迭代的进度图

图 12-19 展示了 badcnt.c 程序第一次循环迭代的二维进度图。水平轴对应于线程 1，垂直轴对应于线程 2。点（L_1，S_2）对应于线程 1 完成了 L_1 而线程 2 完成了 S_2 的状态。

进度图将指令执行模型化为从一种状态到另一种状态的*转换*（transition）。转换被表示为一条从一点到相邻点的有向边。合法的转换是向右（线程 1 中的一条指令完成）或者向上（线程 2 中的一条指令完成）的。两条指令不能在同一时刻完成——对角线转换是不允许的。程序决不会反向运行，所以向下或者向左移动的转换也是不合法的。

一个程序的执行历史被模型化为状态空间中的一条轨迹线。图 12-20 展示了下面指令顺序对应的轨迹线：

H_1，L_1，U_1，H_2，L_2，S_1，T_1，U_2，S_2，T_2

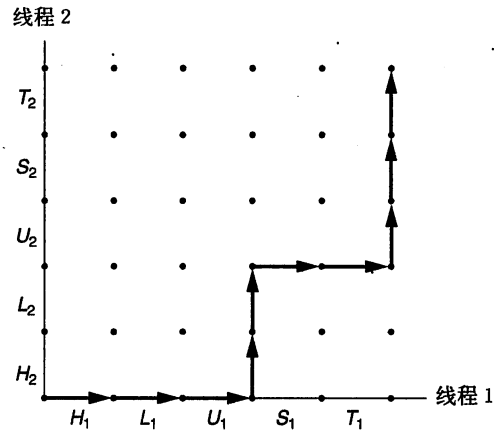

图 12-20 一个轨迹线示例

对于线程 i，操作共享变量 cnt 内容的指令（L_i，U_i，S_i）构成了一个（关于共享变量 cnt 的）*临界区*（critical section），这个临界区不应该和其他进程的临界区交替执行。换句话说，我们想要确保每个线程在执行它的临界区中的指令时，拥有对共享变量的*互斥的访问*（mutually exclusive access）。通常这种现象称为*互斥*（mutual exclusion）。

在进度图中，两个临界区的交集形成的状态空间区域称为*不安全区*（unsafe region）。图 12-21 展示了变量 cnt 的不安全区。注意，不安全区和与它交界的状态相毗邻，但并不包括这些状态。例如，状态（H_1，H_2）和（S_1，U_2）毗邻不安全区，但是它们并不是不安全区的一部分。绕开不安全区的轨迹线叫做*安全轨迹线*（safe trajectory）。相反，接触到任何不安全区的轨迹线就叫做*不安全轨迹线*（unsafe trajectory）。图 12-21 给出了我们的示例程序 badcnt.c 的状态空间中的安全和不安全轨迹线。上面的轨迹线绕开了不安全区域的左边和上边，所以是安全的。下面的轨迹线穿越不安全区，因此是不安全的。

任何安全轨迹线都将正确地更新共享计数器。为了保证线程化程序示例的正确执行（实际上任何共享全局数据结构的并发程序的正确执行）我们必须以某种方式*同步*线程，使它们总是有一条安全轨迹线。一个经典的方法是基于信号量的思想，接下来我们就介绍它。

练习题 12.8 使用图 12-21 中的进度图，将下列轨迹线划分为安全的或者不安全的。

A. $H_1, L_1, U_1, S_1, H_2, L_2, U_2, S_2, T_2, T_1$

B. $H_2, L_2, H_1, L_1, U_1, S_1, T_1, U_2, S_2, T_2$

C. $H_1, H_2, L_2, U_2, S_2, L_1, U_1, S_1, T_1, T_2$

www.TopSage.com

12.5.2 信号量

Edsger Dijkstra，并发编程领域的先锋人物，提出了一种经典的解决同步不同执行线程问题的方法，这种方法是基于一种叫做信号量（semaphore）的特殊类型变量的。信号量 s 是具有非负整数值的全局变量，只能由两种特殊的操作来处理，这两种操作称为 P 和 V：

- $P(s)$：如果 s 是非零的，那么 P 将 s 减 1，并且立即返回。如果 s 为零，那么就挂起这个线程，直到 s 变为非零，而一个 V 操作会重启这个线程。在重启之后，P 操作将 s 减 1，并将控制返回给调用者。
- $V(s)$：V 操作将 s 加 1。如果有任何线程阻塞在 P 操作等待 s 变成非零，那么 V 操作会重启这些线程中的一个，然后该线程将 s 减 1，完成它的 P 操作。

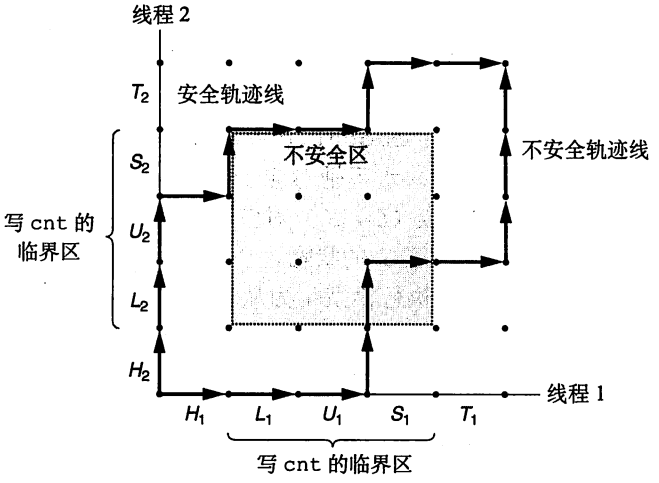

图 12-21　安全和不安全轨迹线。临界区的交集形成了不安全区。绕开不安全区的轨迹线能够正确更新计数器变量

P 中的测试和减 1 操作是不可分割的，也就是说，一旦预测信号量 s 变为非零，就会将 s 减 1，不能有中断。V 中的加 1 操作也是不可分割的，也就是加载、加 1 和存储信号量的过程中没有中断。注意，V 的定义中没有定义等待线程被重新启动的顺序。唯一的要求是 V 必须只能重启一个正在等待的线程。因此，*当有多个线程在等待同一个信号量时，你不能预测 V 操作要重启哪一个线程。*

P 和 V 的定义确保了一个正在运行的程序绝不可能进入这样一种状态，也就是一个正确初始化了的信号量有一个负值。这个属性称为*信号量不变性*（semaphore invariant），为控制并发程序的轨迹线提供了强有力的工具，在下一节中我们将看到。

Posix 标准定义了许多操作信号量的函数。

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

返回：若成功则为 0，若出错则为 −1。

sem_init 函数将信号量 sem 初始化为 value。每个信号量在使用前必须初始化。针对我们的目的，中间的参数总是零。程序分别通过调用 sem_wait 和 sem_post 函数来执行 P 和 V 操作。为了简明，我们更喜欢使用下面这些等价的 P 和 V 的包装函数：

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

返回：无。

www.TopSage.com

名字 *P* 和 *V* 的起源

Edsger Dijkstra（1930—2002）出生于荷兰。名字 *P* 和 *V* 来源于荷兰语单词 *Proberen*（测试）和 *Verhogen*（增加）。

12.5.3 使用信号量来实现互斥

信号量提供了一种很方便的方法来确保对共享变量的互斥访问。基本思想是将每个共享变量（或者一组相关的共享变量）与一个信号量 s（初始为 1）联系起来，然后用 $P(s)$ 和 $V(s)$ 操作将相应的临界区包围起来。

以这种方式来保护共享变量的信号量叫做二元信号量（binary semaphore），因为它的值总是 0 或者 1。以提供互斥为目的的二元信号量常常也称为互斥锁（mutex）。在一个互斥锁上执行 *P* 操作称为对互斥锁加锁。类似地，执行 *V* 操作称为对互斥锁解锁。对一个互斥锁加了锁但是还没有解锁的线程称为占用这个互斥锁。一个被用作一组可用资源的计数器的信号量称为计数信号量。

图 12-22 中的进度图展示了我们如何利用二元信号量来正确地同步我们的计数器程序示例。每个状态都标出了该状态中信号量 s 的值。关键思想是这种 *P* 和 *V* 操作的结合创建了一组状态，叫做禁止区（forbidden region），其中 $s<0$。因为信号量的不变性，没有实际可行的轨迹线能够包含禁止区中的状态。而且，因为禁止区完全包括了不安全区，所以没有实际可行的轨迹线能够接触不安全区的任何部分。因此，每条实际可行的轨迹线都是安全的，而且不管运行时指令顺序是怎样的，程序都会正确地增加计数器的值。

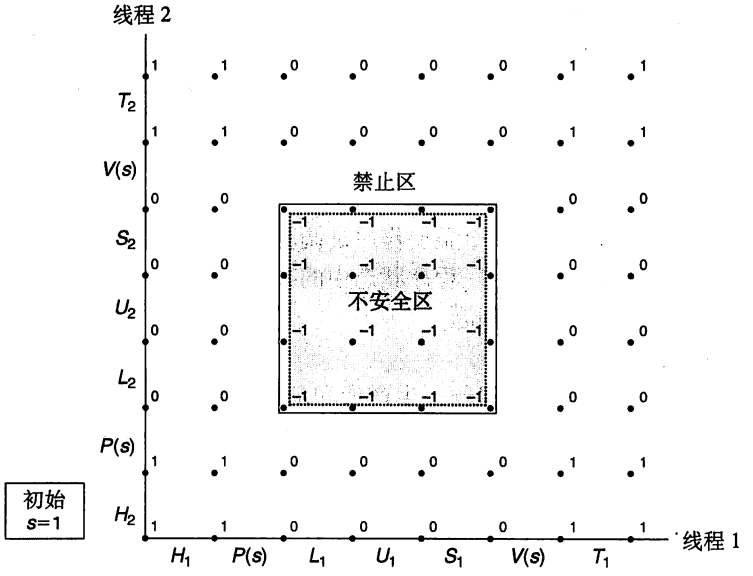

图 12-22 使用信号量来互斥。$s<0$ 的不可行状态定义了一个禁止区，禁止区完全包括了不安全区，阻止了实际可行的轨迹线接触到不安全区

从可操作的意义上来说，由 *P* 和 *V* 操作创建的禁止区使得在任何时间点上，在被包围的临界区中，不可能有多个线程在执行指令。换句话说，信号量操作确保了对临界区的互斥访问。

总的来说，为了用信号量正确同步图 12-16 中的计数器程序示例，我们首先声明一个信号量 mutex：

```
volatile int cnt = 0; /* Counter */
sem_t mutex;          /* Semaphore that protects counter */
```

www.TopSage.com

然后在主例程中将 mutex 初始化为 1：

```
Sem_init(&mutex, 0, 1);  /* mutex = 1 */
```

最后，我们通过在线程例程中对共享变量 cnt 的更新包围 *P* 和 *V* 操作，从而保护了它们：

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

当我们运行正确同步了的程序时，现在它每次都能产生正确的结果了。

```
linux> ./goodcnt 1000000
OK cnt=2000000

linux> ./goodcnt 1000000
OK cnt=2000000
```

进度图的局限性

进度图给了我们一种较好的方法，将在单处理器上的并发程序执行可视化，也帮助我们理解为什么需要同步。然而，它们确实也有局限性，特别是对于在多处理器上的并发执行，在多处理器上一组 CPU/ 高速缓存对共享同一个主存。多处理器的工作方式是进度图不能解释的。特别是，一个多处理器存储系统可以处于一种状态，不对应于进度图中任何轨迹线。不管如何，结论总是一样的：无论是在单处理器还是多处理器上运行程序，都要同步你对共享变量的访问。

12.5.4 利用信号量来调度共享资源

除了提供互斥之外，信号量的另一个重要作用是调度对共享资源的访问。在这种场景中，一个线程用信号量操作来通知另一个线程，程序状态中的某个条件已经为真了。两个经典而有用的例子是生产者－消费者和读者－写者问题。

1. 生产者－消费者问题

图 12-23 给出了生产者－消费者问题。生产者和消费者线程共享一个有 *n* 个槽的有限缓冲区。生产者线程反复地生成新的项目（item），并把它们插入到缓冲区中。消费者线程不断地从缓冲区中取出这些项目，然后消费（使用）它们。也可能有多个生产者和消费者的变种。

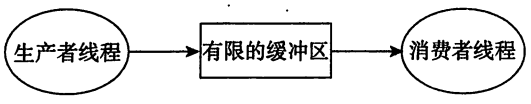

图 12-23 生产者－消费者问题。生产者产生项目并把它们插入到一个有限的缓冲区中。消费者从缓冲区中取出这些项目，然后消费它们

因为插入和取出项目都涉及更新共享变量，所以我们必须保证对缓冲区的访问是互斥的。但是只保证互斥访问是不够的，我们还需要调度对缓冲区的访问。如果缓冲区是满的（没有空的槽位），那么生产者必须等待直到有一个槽位变为可用。与之相似，如果缓冲区是空的（没有可取用的项目），那么消费者必须等待直到有一个项目变为可用。

生产者－消费者的相互作用在现实系统中是很普遍的。例如，在一个多媒体系统中，生产者编码视频帧，而消费者解码并在屏幕上呈现出来。缓冲区的目的是为了减少视频流的抖动

www.TopSage.com

(jitter)，而这种抖动是由各个帧的编码和解码时与数据相关的差异引起的。缓冲区为生产者提供了一个槽位池，而为消费者提供一个已编码的帧池。另一个常见的示例是图形用户接口设计。生产者检测到鼠标和键盘事件，并将它们插入到缓冲区中。消费者以某种基于优先级的方式从缓冲区取出这些事件，并显示在屏幕上。

在本节中，我们将开发一个简单的包，叫做 SBUF，用来构造生产者－消费者程序。在下一节里，我们会看到如何用它来构造一个基于预线程化（prethreading）的有趣的并发服务器。SBUF 操作类型为 sbuf_t 的有限缓冲区（见图 12-24）。项目存放在一个动态分配的 *n* 项整数数组（buf）中。front 和 rear 索引值记录该数组中的第一项和最后一项。三个信号量同步对缓冲区的访问。mutex 信号量提供互斥的缓冲区访问。slots 和 items 信号量分别记录空槽位和可用项目的数量。

code/conc/sbuf.h

```
typedef struct {
    int *buf;          /* Buffer array */
    int n;             /* Maximum number of slots */
    int front;         /* buf[(front+1)%n] is first item */
    int rear;          /* buf[rear%n] is last item */
    sem_t mutex;       /* Protects accesses to buf */
    sem_t slots;       /* Counts available slots */
    sem_t items;       /* Counts available items */
} sbuf_t;
```

code/conc/sbuf.h

图 12-24 sbuf_t：SBUF 包使用的有限缓冲区

图 12-25 给出了 SBUF 函数的实现。sbuf_init 函数为缓冲区分配堆存储器，设置 front 和 rear 表示一个空的缓冲区，并为三个信号量赋初始值。这个函数在调用其他三个函数中的任何一个之前调用一次。sbuf_deinit 函数是当应用程序使用完缓冲区时，释放缓冲区存储的。sbuf_insert 函数等待一个可用的槽位，对互斥锁加锁，添加项目，对互斥锁解锁，然后宣布有一个新项目可用。sbuf_remove 函数是与 sbuf_insert 函数对称的。在等待一个可用的缓冲区项目之后，对互斥锁加锁，从缓冲区的前面取出该项目，对互斥锁解锁，然后发信号通知一个新的槽位可供使用。

code/conc/sbuf.c

```
#include "csapp.h"
#include "sbuf.h"

/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                       /* Buffer holds max of n items */
    sp->front = sp->rear = 0;        /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1);      /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n);      /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0);      /* Initially, buf has zero data items */
}

/* Clean up buffer sp */
```

图 12-25 SBUF：同步对有限缓冲区并发访问的包

www.TopSage.com

```
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}

/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;   /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->items);                          /* Announce available item */
}

/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                          /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)];  /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->slots);                          /* Announce available slot */
    return item;
}
```

code/conc/sbuf.c

图 12-25 （续）

练习题 12.9 设 p 表示生产者数量，c 表示消费者数量，而 n 表示以项目单元为单位的缓冲区大小。对于下面的每个场景，指出 sbuf_insert 和 sbuf_remove 中的互斥锁信号量是否是必需的。

A. $p=1, c=1, n>1$

B. $p=1, c=1, n=1$

C. $p>1, c>1, n=1$

2. 读者－写者问题

*读者－写者*问题是互斥问题的一个概括。一组并发的线程要访问一个共享对象，例如一个主存中的数据结构，或者一个磁盘上的数据库。有些线程只读对象，而其他的线程只修改对象。修改对象的线程叫做*写者*。只读对象的线程叫做*读者*。写者必须拥有对对象的独占的访问，而读者可以和无限多个其他的读者共享对象。一般来说，有无限多个并发的读者和写者。

读者－写者交互在现实系统中很常见。例如，一个在线航空预定系统中，允许有无限多个客户同时查看座位分配，但是正在预订座位的客户必须拥有对数据库的独占的访问。再来看另外一个例子，在一个多线程缓存 Web 代理中，无限多个线程可以从共享页面缓存中取出已有的页面，但是任何向缓存中写入一个新页面的线程必须拥有独占的访问。

读者－写者问题有几个变种，每个都是基于读者和写者的优先级的。*第一类读者－写者问题*，读者优先，要求不要让读者等待，除非已经把使用对象的权限赋予了一个写者。换句话说，读者不会因为有一个写者在等待而等待。*第二类读者－写者问题*，写者优先，要求一旦一个写者准备好可以写，它就会尽可能快地完成它的写操作。同第一类问题不同，在一个写者后到达的读者必须等待，即使这个写者也是在等待。

www.TopSage.com

图 12-26 给出了一个对第一类读者－写者问题的解答。同许多同步问题的解答一样，这个解答很微妙，极具欺骗性地简单。信号量 w 控制对访问共享对象的临界区的访问。信号量 mutex 保护对共享变量 readcnt 的访问，readcnt 统计当前在临界区中的读者数量。每当一个写者进入临界区时，它对互斥锁 w 加锁，每当它离开临界区时，对 w 解锁。这就保证了任意时刻临界区中最多只有一个写者。另一方面，只有第一个进入临界区的读者对 w 加锁，而只有最后一个离开临界区的读者对 w 解锁。当一个读者进入和离开临界区时，如果还有其他读者在临界区中，那么这个读者会忽略互斥锁 w。这就意味着只要还有一个读者占用互斥锁 w，无限多数量的读者可以没有障碍地进入临界区。

对这两种读者－写者问题的正确解答可能导致饥饿（starvation），饥饿就是一个线程无限期地阻塞，无法进展。例如，如图 12-26 所示的解答中，如果有读者不断地到达，写者就可能无限期地等待。

练习题 12.10 如图 12-26 所示的对第一类读者－写者问题的解答给予读者较高的优先级，但是从某种意义上说，这种优先级是很弱的，因为一个离开临界区的写者可能重启一个在等待的写者，而不是一个在等待的读者。描述出一个场景，其中这种弱优先级会导致一群写者使得一个读者饥饿。

```
/* Global variables */
int readcnt;    /* Initially = 0 */
sem_t mutex, w; /* Both initially = 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Critical section */
        /* Reading happens  */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

void writer(void)
{
    while (1) {
        P(&w);

        /* Critical section */
        /* Writing happens  */

        V(&w);
    }
}
```

图 12-26 对第一类读者－写者问题的解答。读者优先级高于写者

其他同步机制

我们已经向你展示了如何利用信号量来同步线程，主要是因为它们简单、经典，并且有一个清晰的语义模型。但是你应该知道还是存在着其他同步技术的。例如，Java 线程是用一种叫做 Java 监控器（Java Monitor）[51] 的机制来同步的，它提供了对信号量互斥和调度能力的更高级别的抽象；实际上，监控器可以用信号量来实现。再来看一个例子，Pthreads 接口定义了一组对互斥锁和条件变量的同步操作。Pthreads 互斥锁被用来实现互斥。条件变量用来调度对共享资源的访问，例如在一个生产者－消费者程序中的有界缓冲区。

www.TopSage.com

12.5.5　综合：基于预线程化的并发服务器

我们已经知道了如何使用信号量来访问共享变量和调度对共享资源的访问。为了帮助你更清晰地理解这些思想，让我们把它们应用到一个基于称为*预线程化*（prethreading）技术的并发服务器上。

在如图 12-14 所示的并发服务器中，我们为每一个新客户端创建了一个新线程。这种方法的缺点是我们为每一个新客户端创建一个新线程，导致不小的代价。一个基于预线程化的服务器试图通过使用如图 12-27 所示的生产者－消费者模型来降低这种开销。服务器是由一个主线程和一组工作者线程构成的。主线程不断地接受来自客户端的连接请求，并将得到的连接描述符放在一个有限缓冲区中。每一个工作者线程反复地从共享缓冲区中取出描述符，为客户端服务，然后等待下一个描述符。

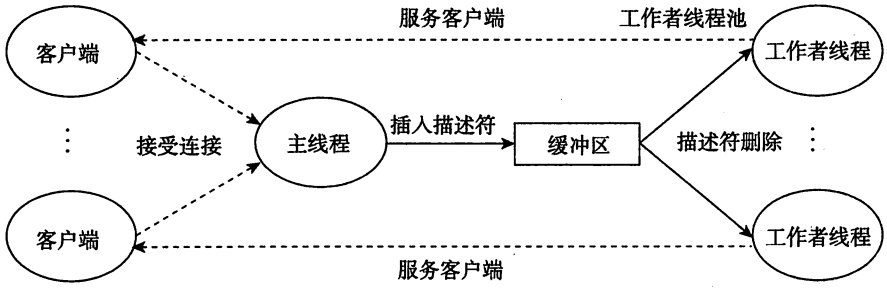

图 12-27　预线程化的并发服务器的组织结构。一组现有的线程不断地取出和处理来自有限缓冲区的已连接描述符

图 12-28 显示了我们怎样用 SBUF 包来实现一个预线程化的并发 echo 服务器。在初始化了缓冲区 sbuf （第 23 行）后，主线程创建了一组工作者线程（第 26 ～ 27 行）。然后它进入了无限的服务器循环，接受连接请求，并将得到的已连接描述符插入到缓冲区 sbuf 中。每个工作者线程的行为都非常简单。它等待直到它能从缓冲区中取出一个已连接描述符（第 39 行），然后调用 echo_cnt 函数回送客户端的输入。

code/conc/echoservert_pre.c

```
1   #include "csapp.h"
2   #include "sbuf.h"
3   #define NTHREADS  4
4   #define SBUFSIZE  16
5
6   void echo_cnt(int connfd);
7   void *thread(void *vargp);
8
9   sbuf_t sbuf; /* Shared buffer of connected descriptors */
10
11  int main(int argc, char **argv)
12  {
13      int i, listenfd, connfd, port;
14      socklen_t clientlen=sizeof(struct sockaddr_in);
15      struct sockaddr_in clientaddr;
16      pthread_t tid;
```

图 12-28　一个预线程化的并发 echo 服务器。这个服务器使用的是有一个生产者和多个消费者的生产者－消费者模型

www.TopSage.com

```
17
18      if (argc != 2) {
19          fprintf(stderr, "usage: %s <port>\n", argv[0]);
20          exit(0);
21      }
22      port = atoi(argv[1]);
23      sbuf_init(&sbuf, SBUFSIZE);
24      listenfd = Open_listenfd(port);
25
26      for (i = 0; i < NTHREADS; i++)  /* Create worker threads */
27          Pthread_create(&tid, NULL, thread, NULL);
28
29      while (1) {
30          connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
31          sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
32      }
33  }
34
35  void *thread(void *vargp)
36  {
37      Pthread_detach(pthread_self());
38      while (1) {
39          int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
40          echo_cnt(connfd);                /* Service client */
41          Close(connfd);
42      }
43  }
```

code/conc/echoservert_pre.c

图 12-28 （续）

如图 12-29 所示的函数 echo_cnt 是图 11-21 中的 echo 函数的一个版本，它在全局变量 byte_cnt 中记录了从所有客户端接收到的累计字节数。这是一段值得研究的有趣代码，因为它向你展示了一个从线程例程调用的初始化程序包的一般技术。在这种情况下，我们需要初始化 byte_cnt 计数器和 mutex 信号量。一种方法是我们为 SBUF 和 RIO 程序包使用过的，它要求主线程显式地调用一个初始化函数。另外一种方法，在此显示的，是当第一次有某个线程调用 echo_cnt 函数时，使用 pthread_once 函数（第 19 行）去调用初始化函数。这个方法的优点是它使程序包的使用更加容易。这种方法的缺点是每一次调用 echo_cnt 都会导致调用 pthread_once 函数，而在很多时候它没有做什么有用的事。

code/conc/echo_cnt.c

```
1   #include "csapp.h"
2
3   static int byte_cnt;  /* Byte counter */
4   static sem_t mutex;   /* and the mutex that protects it */
5
6   static void init_echo_cnt(void)
7   {
8       Sem_init(&mutex, 0, 1);
9       byte_cnt = 0;
10  }
11
```

图 12-29 echo_cnt：echo 的一个版本，它对从客户端接收的所有字节计数

www.TopSage.com

大家网 TopSage.com

```
12  void echo_cnt(int connfd)
13  {
14      int n;
15      char buf[MAXLINE];
16      rio_t rio;
17      static pthread_once_t once = PTHREAD_ONCE_INIT;
18
19      Pthread_once(&once, init_echo_cnt);
20      Rio_readinitb(&rio, connfd);
21      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
22          P(&mutex);
23          byte_cnt += n;
24          printf("thread %d received %d (%d total) bytes on fd %d\n",
25                 (int) pthread_self(), n, byte_cnt, connfd);
26          V(&mutex);
27          Rio_writen(connfd, buf, n);
28      }
29  }
```

code/conc/echo_cnt.c

图 12-29 （续）

一旦程序包被初始化，echo_cnt 函数会初始化 RIO 带缓冲区的 I/O 包（第 20 行），然后回送从客户端接收到的每一个文本行。注意，在第 23 ～ 25 行中对共享变量 byte_cnt 的访问是被 *P* 和 *V* 操作保护的。

基于线程的事件驱动程序

I/O 多路复用不是编写事件驱动程序的唯一方法。例如，你可能已经注意到我们刚才开发的并发的预线程化的服务器实际上是一个事件驱动服务器，带有主线程和工作者线程的简单状态机。主线程有两种状态（“等待连接请求”和“等待可用的缓冲区槽位”）、两个 I/O 事件（“连接请求到达”和“缓冲区槽位变为可用”）和两个转换（“接受连接请求”和“插入缓冲区项目”）。类似地，每个工作者线程有一个状态（“等待可用的缓冲项目”）、一个 I/O 事件（“缓冲区项目变为可用”）和一个转换（“取出缓冲区项目”）。

12.6 使用线程提高并行性

到目前为止，在对并发的研究中，我们都假设并发线程是在单处理器系统上执行的。然而，许多现代机器具有多核处理器。并发程序通常在这样的机器上运行得更快，因为操作系统内核在多个核上并行地调度这些并发线程，而不是在单个核上顺序地调度。在像繁忙的 Web 服务器、数据库服务器和大型科学计算代码这样的应用中利用这种并行性是至关重要的，而且在像 Web 浏览器、电子表格处理程序和文档处理程序这样的主流应用中，并行性也变得越来越有用。

图 12-30 给出了顺序、并发和并行程序之间的集合关系。所有程序的集合能够划分成不相交的顺序程序集合和并发程序的集合。写顺序程序只有一条逻辑流。写并发程序有多条并发流。并行程序是一个运行在多个处理器上的并发程序。因此，并行程序的集合是并发程序集合的真子集。

并行程序的详细处理超出了本书讨论的范围，但是研究一个非常简单的示例程序能够帮助你理解并行编程的一些重要的方面。例如，考虑我们如何并行地对一列整数 0，…，$n-1$ 求和。当然，对于这个特殊的问题，有闭合形式表达式的解答（译者注：即有现成的公式来计算它，即和等于 $n(n-1)/2$），但是尽管如此，它是一个简洁和易于理解的示例，能让我们对并行程序做一些

www.TopSage.com

有趣的说明。

最直接的方法是将序列划分成 t 个不相交的区域，然后给 t 个不同的线程每个分配一个区域。为了简单，假设 n 是 t 的倍数，这样每个区域有 n/t 个元素。主线程创建 t 个对等线程，每个对等线程 k 并行地运行在它自己的处理器的核上，并计算 s_k，s_k 是区域 k 中元素的和。一旦对等线程计算完毕，主线程通过把每个 s_k 都加起来，计算出最终的结果。

图 12-31 给出了我们会如何实现这个简单的并行求和算法。在第 27 ～ 32 行，主线程创建对等线程，然后等待它们结束。注意，主线程传递给每个对等线程一个小整数，作为唯一的线程 ID。每个对等线程会用它的线程 ID 来决定它应该计算序列的哪一部分。这个向对等线程传递一个小的唯一的线程 ID 的思想是一项通用技术，许多并行应用中都用到了它。在对等线程终止后，psum 向量包含着每个对等线程计算出来的部分和。然后主线程对向量 psum 的元素求和（第 35 ～ 36 行），并且使用闭合形式解答来验证结果（第 39 ～ 40 行）。

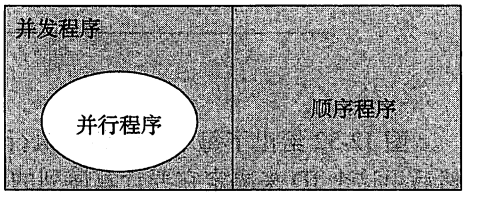

图 12-30 顺序、并发和并行程序集合之间的关系

code/conc/psum.c

```
1   #include "csapp.h"
2   #define MAXTHREADS 32
3
4   void *sum(void *vargp);
5
6   /* Global shared variables */
7   long psum[MAXTHREADS];  /* Partial sum computed by each thread */
8   long nelems_per_thread; /* Number of elements summed by each thread */
9
10  int main(int argc, char **argv)
11  {
12      long i, nelems, log_nelems, nthreads, result = 0;
13      pthread_t tid[MAXTHREADS];
14      int myid[MAXTHREADS];
15
16      /* Get input arguments */
17      if (argc != 3) {
18          printf("Usage: %s <nthreads> <log_nelems>\n", argv[0]);
19          exit(0);
20      }
21      nthreads = atoi(argv[1]);
22      log_nelems = atoi(argv[2]);
23      nelems = (1L << log_nelems);
24      nelems_per_thread = nelems / nthreads;
25
26      /* Create peer threads and wait for them to finish */
27      for (i = 0; i < nthreads; i++) {
28          myid[i] = i;
29          Pthread_create(&tid[i], NULL, sum, &myid[i]);
30      }
31      for (i = 0; i < nthreads; i++)
32          Pthread_join(tid[i], NULL);
33
```

图 12-31 简单的并行程序，使用多个线程来计算一个序列元素的和

www.TopSage.com

```
    /* Add up the partial sums computed by each thread */
    for (i = 0; i < nthreads; i++)
        result += psum[i];

    /* Check final answer */
    if (result != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", result);

    exit(0);
}
```

code/conc/psum.c

图 12-31 （续）

图 12-32 给出了每个对等线程执行的函数。在第 3 行中，线程从线程参数中提取出线程 ID，然后用这个 ID 来决定它要计算序列的哪个区域（第 4 ～ 5 行）。在第 8 ～ 10 行中，线程在它的那部分序列上操作，然后更新部分和向量中它的条目（第 11 行）。注意，我们很小心地给了每个对等线程一个唯一的存储位置来更新，因此就不需要用信号量互斥锁来同步对 psum 数组的访问。在这种特殊的情况下，唯一需要同步的是主线程必须等待每个子线程结束，这样它就知道 psum 中的每个条目都是有效的了。

code/conc/psum.c

```
1  void *sum(void *vargp)
2  {
3      int myid = *((int *)vargp);            /* Extract the thread ID */
4      long start = myid * nelems_per_thread; /* Start element index */
5      long end = start + nelems_per_thread;  /* End element index */
6      long i, sum = 0;
7
8      for (i = start; i < end; i++) {
9          sum += i;
10     }
11     psum[myid] = sum;
12
13     return NULL;
14 }
```

code/conc/psum.c

图 12-32 图 12-31 中程序的线程例程

图 12-33 给出了如图 12-31 所示程序的总的运行时间，它是一个线程个数的函数。在每种情况下，程序运行在一个有四个处理器核的系统上，对一个 $n=2^{31}$ 个元素的序列求和。我们看到，随着线程数的增加，运行时间下降，直到增加到四个线程，此时，运行时间趋于平稳，甚至开始有点增加。在最理想的情况下，我们会期望运行时间随着核数的增加线性地下降。也就是说，我们会期望线程数每增加一倍，运行时间就下降一半。确实是这样，直到到达 $t>4$ 的时候，四个核中的每一个都忙于运行至少一个线程。随着线程数量的增加，运行时间实际上增加了一点，这是由于在一个核上多个线程上下文切换的开销。由于这个原因，并行程序常常被写为每个核上只运行一个线程。

虽然绝对运行时间是衡量程序性能的终极标准，但是还是有一些有用的相对衡量标准，称为加速比和效率，它们能够说明并行程序有多好地利用了潜在的并行性。并行程序的*加速比*（speedup）通常定义为

www.TopSage.com

$$S_p = \frac{T_1}{T_p}$$

这里 p 是处理器核的数量，T_k 是在 k 个核上的运行时间。这个公式有时称为*强扩展*（strong scaling）。当 T_1 是程序顺序执行版本的执行时间时，S_p 称为*绝对加速比*（absolute speedup）。当 T_1 是程序并行版本在一个核上的执行时间时，S_p 称为*相对加速比*（relative speedup）。绝对加速比比相对加速比能更真实地衡量并行的好处。即使是当并行程序在一个处理器上运行时，也常常会受到同步开销的影响，而这些开销会人为地增加相对加速比的数值，因为它们使分子增大了。另一方面，绝对加速比比相对加速比更难以测量，因为测量绝对加速比需要程序的两种不同的版本。对于复杂的并行代码，创建一个独立的顺序版本可能不太实际，或者因为代码太复杂，或者因为源代码不可得。

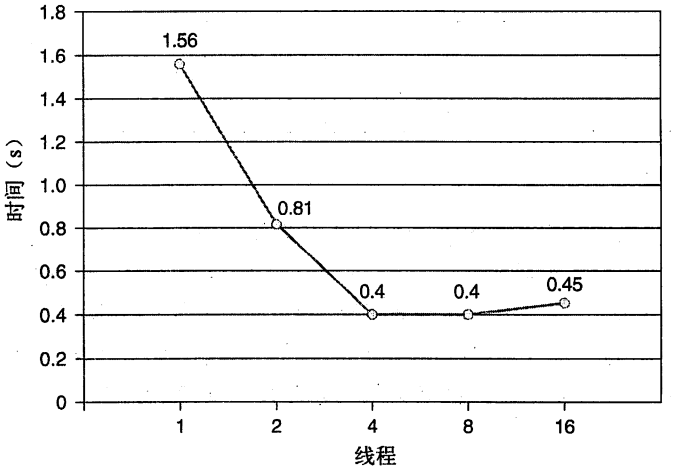

图 12-33 图 12-31 中程序在有四个核的多核机器上的性能。对有 2^{31} 个元素的序列求和

一种相关的测量称为*效率*（efficiency），定义为

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

通常表示为范围在 (0，100] 之间的百分比。效率是对由于并行化造成的开销的衡量。具有高效率的程序比效率低的程序在有用的工作上花费更多的时间，在同步和通信上花费更少的时间。

图 12-34 给出了我们并行求和示例程序的各个加速比和效率测量值。像这样超过 90% 的效率是非常好的，但是不要被欺骗了。能取得这么高的效率是因为我们的问题非常容易并行化。在实际中，很少会这样。数十年来，并行编程一直是一个很活跃的研究领域。随着商用多核机器的出现，这些机器的核数每几年就翻一番，并行编程会继续是一个深入、困难而活跃的研究领域。

线程（t）	1	2	4	8	16
核（p）	1	2	4	4	4
运行时间（T_p）	1.56	0.81	0.40	0.40	0.45
加速比（S_p）	1	1.9	3.9	3.9	3.5
效率（E_p）	100%	95%	98%	98%	88%

图 12-34 图 12-33 中执行时间的加速比和并行效率

加速比还有另外一面，称为*弱扩展*（weak scaling），在增加处理器数量的同时，增加问题的规模，这样随着处理器数量的增加，每个处理器执行的工作量保持不变。在这种描述中，加速比

www.TopSage.com

和效率被表达为单位时间完成的工作总量。例如，如果我们将处理器数量翻倍，同时每个小时也做了两倍的工作量，那么我们就有线性的加速比和 100% 的效率。

弱扩展常常是比强扩展更真实的测量值，因为它更准确地反映了我们用更大的机器做更多的工作的愿望。对于科学计算程序来说尤其如此，科学计算问题的规模很容易增加，更大的问题规模直接就意味着更好的预测。不过，还是有一些应用的规模不那么容易增加，对于这样的应用，强扩展是更合适的。例如，实时信号处理应用所执行的工作量常常是由产生信号的物理传感器的属性决定的。改变工作总量需要用不同的物理传感器，这不太实际或者不太必要。对于这类应用，我们通常想要用并行来尽可能快地完成定量的工作。

练习题 12.11 对于下表中的并行程序，填写空白处。假设使用强扩展。

线程（t）	1	2	4
核（p）	1	2	4
运行时间（T_p）	12	8	6
加速比（S）		1.5	
效率（E_p）	100%		50%

12.7 其他并发问题

你可能已经注意到了，一旦我们要求同步对共享数据的访问，那么事情就变得复杂得多了。迄今为止，我们已经看到了用于互斥和生产者－消费者同步的技术，但这仅仅是冰山一角。同步从根本上说是很难的问题，它引出了在普通的顺序程序中不会出现的问题。这一小节是关于你在写并发程序时需要注意的一些问题的（决不是完整的）综述。为了让事情具体化，我们将以线程为例描述我们的讨论。不过要记住，这些典型问题是任何类型的并发流操作共享资源时都会出现的。

12.7.1 线程安全

当用线程编写程序时，我们必须小心地编写那些具有称为*线程安全性*（thread safety）属性的函数。一个函数被称为*线程安全的*（thread-safe），当且仅当被多个并发线程反复地调用时，它会一直产生正确的结果。如果一个函数不是线程安全的，我们就说它是*线程不安全的*（thread-unsafe）。

我们能够定义出四个（不相交的）线程不安全函数类：

*第 1 类：不保护共享变量的函数。*我们在图 12-16 中的 thread 函数中就已经遇到过这样的问题，该函数对一个未受保护的全局计数器变量加 1。将这类线程不安全函数变成线程安全的，相对而言比较容易：利用像 P 和 V 操作这样的同步操作来保护共享的变量。这个方法的优点是在调用程序中不需要做任何修改。缺点是同步操作将减慢程序的执行时间。

*第 2 类：保持跨越多个调用的状态的函数。*一个伪随机数生成器是这类线程不安全函数的简单例子。请参考图 12-35 中的伪随机数生成器程序包。rand 函数是线程不安全的，因为当前调用的结果依赖于前次调用的中间结果。当调用 srand 为 rand 设置了一个种子后，我们从一个单线程中反复地调用 rand，能够预期得到一个可重复的随机数字序列。然而，如果多线程调用 rand 函数，这种假设就不再成立了。

使得像 rand 这样的函数线程安全的唯一方式是重写它，使得它不再使用任何 static 数据，而是依靠调用者在参数中传递状态信息。这样做的缺点是，程序员现在还要被迫修改调用程序中的代码。在一个大的程序中，可能有成百上千个不同的调用位置，做这样的修改将是非常麻烦的，而且容易出错。

www.TopSage.com

大家网 TopSage.com

code/conc/rand.c

```
unsigned int next = 1;

/* rand - return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

code/conc/rand.c

图 12-35 一个线程不安全的伪随机数生成器 [58]

第 3 类：返回指向静态变量的指针的函数。某些函数，例如 ctime 和 gethostbyname，将计算结果放在一个 static 变量中，然后返回一个指向这个变量的指针。如果我们从并发线程中调用这些函数，那么将可能发生灾难，因为正在被一个线程使用的结果会被另一个线程悄悄地覆盖了。

有两种方法来处理这类线程不安全函数。一种选择是重写函数，使得调用者传递存放结果的变量的地址。这就消除了所有共享数据，但是它要求程序员能够修改函数的源代码。

如果线程不安全函数是难以修改或不可能修改的（例如，代码非常复杂或是没有源代码可用），那么另外一种选择就是使用*加锁－拷贝*（lock-and-copy）技术。基本思想是将线程不安全函数与互斥锁联系起来。在每一个调用位置，对互斥锁加锁，调用线程不安全函数，将函数返回的结果拷贝到一个私有的存储器位置，然后对互斥锁解锁。为了尽可能地减少对调用者的修改，你应该定义一个线程安全的包装函数，它执行加锁－拷贝，然后通过调用这个包装函数来取代所有对线程不安全函数的调用。例如，图 12-36 给出了 ctime 的一个线程安全的版本，利用的就是加锁－拷贝技术。

第 4 类：调用线程不安全函数的函数。如果函数 f 调用线程不安全函数 g，那么 f 就是线程不安全的吗？不一定。如果 g 是第 2 类函数，即依赖于跨越多次调用的状态，那么 f 也是线程不安全的，而且除了重写 g 以外，没有什么办法。然而，如果 g 是第 1 类或者第 3 类函数，那么只要你用一个互斥锁保护调用位置和任何得到的共享数据，f 仍然可能是线程安全的。在图 12-36 中我们看到了一个这种情况很好的示例，其中我们使用加锁－拷贝编写了一个线程安全函数，它调用了一个线程不安全的函数。

code/conc/ctime_ts.c

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp); /* Copy string from shared to private */
    V(&mutex);
    return privatep;
}
```

code/conc/ctime_ts.c

图 12-36 C 标准库函数 ctime 的线程安全的包装函数。使用加锁－拷贝技术调用一个第 3 类线程不安全函数

12.7.2 可重入性

有一类重要的线程安全函数，叫做*可重入函数*（reentrant function），其特点在于它们具有这样一种属性：当它们被多个线程调用时，不会引用任何共享数据。尽管*线程安全*和*可重入*有时会（不正确地）被用做同义词，但是它们之间还是有清晰的技术差别的，值得留意。图 12-37 展示了可重入函数、线程安全函数和线程不安全函数之间的集合关系。所有函数的集合被划分成不相交的线程安全和线程不安全函数集合。可重入函数集合是线程安全函数的一个真子集。

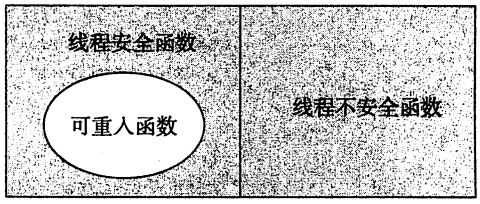

图 12-37　可重入函数、线程安全函数和线程不安全函数之间的集合关系

可重入函数通常要比不可重入的线程安全的函数高效一些，因为它们不需要同步操作。更进一步来说，将第 2 类线程不安全函数转化为线程安全函数的唯一方法就是重写它，使之变为可重入的。例如，图 12-38 展示了图 12-35 中 rand 函数的一个可重入的版本。关键思想是我们用一个调用者传递进来的指针取代了静态的 next 变量。

code/conc/rand_r.c

```
/* rand_r - a reentrant pseudo-random integer on 0..32767 */
int rand_r(unsigned int *nextp)
{
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp / 65536) % 32768;
}
```

code/conc/rand_r.c

图 12-38　rand_r：图 12-35 中的 rand 函数的可重入版本

检查某个函数的代码并*先验地*断定它是可重入的，这可能吗？不幸的是，不一定能这样。如果所有的函数参数都是传值传递的（即没有指针），并且所有的数据引用都是本地的自动栈变量（即没有引用静态或全局变量），那么函数就是*显式可重入的*（explicitly reentrant），也就是说，无论它是被如何调用的，我们都可以断言它是可重入的。

然而，如果把我们的假设放宽松一点，允许显式可重入函数中的一些参数是引用传递的（即我们允许它们传递指针），那么我们就得到了一个*隐式可重入的*（implicitly reentrant）函数，也就是说，如果调用线程小心地传递指向非共享数据的指针，那么它是可重入的。例如，图 12-38 中的 rand_r 函数就是隐式可重入的。

我们总是使用术语*可重入的*（reentrant）既包括显式可重入函数也包括隐式可重入函数。然而，认识到可重入性有时既是调用者也是被调用者的属性，并不只是被调用者单独的属性是非常重要的。

练习题 12.12　图 12-36 中的 ctime_ts 函数是线程安全的，但不是可重入的。请解释说明。

12.7.3 在线程化的程序中使用已存在的库函数

大多数 Unix 函数，包括定义在标准 C 库中的函数（例如 malloc、free、realloc、printf 和 scanf）都是线程安全的，只有一小部分是例外。图 12-39 列出了常见的例外。（参考 [109] 可以得到一个完整的列表。）asctime、ctime 和 localtime 函数是在不同时间和数据格式间相互来回转换时经常使用的函数。gethostbyname、gethostbyaddr 和 inet_

www.TopSage.com

ntoa 函数是我们在第 11 章中遇到过的、经常使用的网络编程函数。strtok 函数是一个过时了的（不再鼓励使用的）用来分析字符串的函数。

线程不安全函数	线程不安全类	Unix 线程安全版本
rand	2	rand_r
strtok	2	strtok_r
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	（无）
localtime	3	localtime_r

图 12-39 常见的线程不安全的库函数

除了 rand 和 strtok 以外，所有这些线程不安全函数都是第 3 类的，它们返回一个指向静态变量的指针。如果我们需要在一个线程化的程序中调用这些函数中的某一个，对调用者来说最不惹麻烦的方法是加锁－拷贝。然而，加锁－拷贝方法有许多缺点。首先，额外的同步降低了程序的速度。其次，像 gethostbyname 这样的函数返回指向复杂结构的结构的指针，要拷贝整个结构层次，需要*深层拷贝*（deep copy）结构。再次，加锁－拷贝方法对像 rand 这样依赖跨越调用的静态状态的第 2 类函数并不有效。

因此，Unix 系统提供大多数线程不安全函数的可重入版本。可重入版本的名字总是以“_r”后缀结尾。例如，gethostbyname 的可重入版本就叫做 gethostbyname_r。我们建议尽可能地使用这些函数。

12.7.4 竞争

当一个程序的正确性依赖于一个线程要在另一个线程到达 y 点之前到达它的控制流中的 x 点时，就会发生*竞争*（race）。通常发生竞争是因为程序员假定线程将按照某种特殊的轨迹线穿过执行状态空间，而忘记了另一条准则规定：线程化的程序必须对任何可行的轨迹线都正确工作。

例子是理解竞争本质的最简单的方法。让我们来看看图 12-40 中的简单程序。主线程创建了四个对等线程，并传递一个指向一个唯一的整数 ID 的指针到每个线程。每个对等线程拷贝它的参数中传递的 ID 到一个局部变量中（第 21 行），然后输出一个包含这个 ID 的信息。它看上去足够简单，但是当在系统上运行这个程序时，我们得到以下不正确的结果：

```
unix> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3
```

问题是由每个对等线程和主线程之间的竞争引起的。你能发现这个竞争吗？下面是发生的情况。当主线程在第 12 行创建了一个对等线程，它传递了一个指向本地栈变量 i 的指针。在此时，竞争出现在下一次在第 12 行调用 pthread_create 和第 21 行参数的间接引用和赋值之间。如果对等线程在主线程执行第 12 行之前就执行了第 21 行，那么 myid 变量就得到正确的 ID。否

www.TopSage.com

则，它包含的就是其他线程的 ID。令人惊慌的是，我们是否得到正确的答案依赖于内核是如何调度线程的执行的。在我们的系统中它失败了，但是在其他系统中，它可能就能正确工作，让程序员“幸福地”察觉不到程序的严重错误。

code/conc/race.c

```
1   #include "csapp.h"
2   #define N 4
3
4   void *thread(void *vargp);
5
6   int main()
7   {
8       pthread_t tid[N];
9       int i;
10
11      for (i = 0; i < N; i++)
12          Pthread_create(&tid[i], NULL, thread, &i);
13      for (i = 0; i < N; i++)
14          Pthread_join(tid[i], NULL);
15      exit(0);
16  }
17
18  /* Thread routine */
19  void *thread(void *vargp)
20  {
21      int myid = *((int *)vargp);
22      printf("Hello from thread %d\n", myid);
23      return NULL;
24  }
```

code/conc/race.c

图 12-40 一个带竞争的程序

为了消除竞争，我们可以动态地为每个整数 ID 分配一个独立的块，并且传递给线程例程一个指向这个块的指针，如图 12-41 所示（第 12 ～ 14 行）。请注意线程例程必须释放这些块以避免存储器泄漏。

当在系统上运行这个程序后，我们现在得到了正确的结果：

```
unix> ./norace
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

练习题 12.13 在图 12.41 中，我们可能想要在主线程中的第 15 行后立即释放已分配的存储器块，而不是在对等线程中释放它。但是这会是个坏注意。这是为什么？

练习题 12.14 A. 在图 12.41 中，我们通过为每个整数 ID 分配一个独立的块来消除竞争。给出一个不调用 malloc 或 free 函数的不同的方法。

B. 这种方法的利弊是什么？

www.TopSage.com

code/conc/norace.c

```
#include "csapp.h"
#define N 4

void *thread(void *vargp);

int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

code/conc/norace.c

图 12-41 图 12-40 中程序的一个没有竞争的正确版本

12.7.5 死锁

信号量引入了一种潜在的令人厌恶的运行时错误，叫做*死锁*（deadlock），它指的是一组线程被阻塞了，等待一个永远也不会为真的条件。进度图对于理解死锁是一个无价的工具。例如，图 12-42 展示了一对用两个信号量来实现互斥的线程的进度图。从这幅图中，我们能够得到一些关于死锁的重要知识：

- 程序员使用 P 和 V 操作顺序不当，以至于两个信号量的禁止区域重叠。如果某个执行轨迹线碰巧到达了*死锁状态* d，那么就不可能有进一步的进展了，因为重叠的禁止区域阻塞了每个合法方向上的进展。换句话说，程序死锁是因为每个线程都在等待其他线程执行一个根本不可能发生的 V 操作。
- 重叠的禁止区域引起了一组称为*死锁区域*（deadlock region）的状态。如果一个轨迹线碰巧到达了一个死锁区域中的状态，那么死锁就是不可避免的了。轨迹线可以进入死锁区域，但是它们不可能离开。
- 死锁是一个相当困难的问题，因为它不总是可预测的。一些幸运的执行轨迹线将绕开死锁区域，而其他的将会陷入这个区域。图 12-42 展示了每种情况的一个示例。对于程序员来说，这其中隐含的着实令人惊慌。你可以运行一个程序 1000 次不出任何问题，但是下一次它就死锁了。或者程序在一台机器上可能运行得很好，但是在另外的机器上就会死锁。最糟糕的是，错误常常是不可重复的，因为不同的执行有不同的轨迹线。

程序死锁有很多原因，要避免死锁一般而言是很困难的。然而，当使用二元信号量来实现互

www.TopSage.com

大家网 TopSage.com

斥时，如图 12-42 所示，你可以应用下面的简单而有效的规则来避免死锁：

互斥锁加锁顺序规则：如果对于程序中每对互斥锁（s，t），每个同时占用 s 和 t 的线程都按照相同的顺序对它们加锁，那么这个程序就是无死锁的。

例如，我们可以通过这样的方法来解决图 12-42 中的死锁问题：在每个线程中先对 s 加锁，然后再对 t 加锁。图 12-43 展示了得到的进度图。

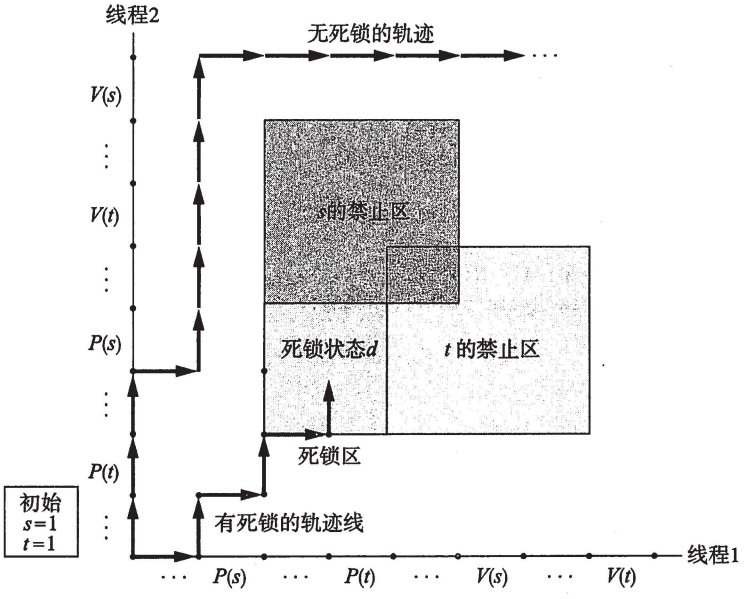

图 12-42　一个有死锁程序的进度图

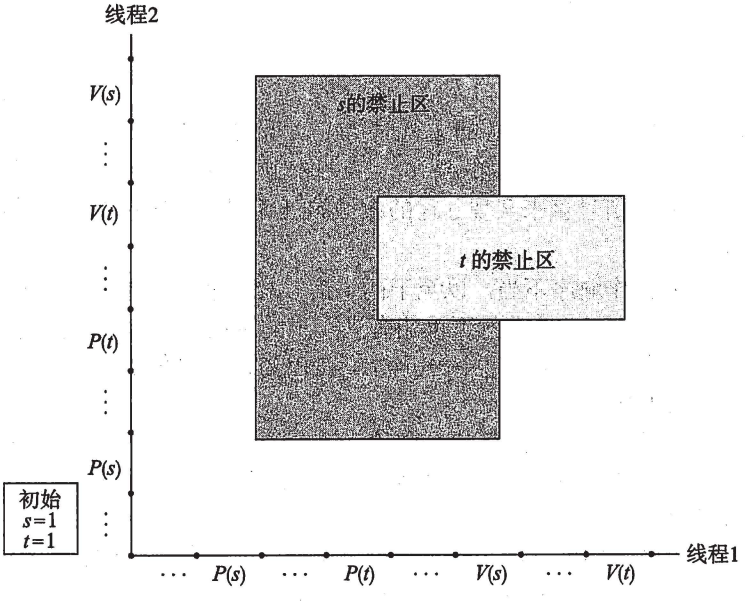

图 12-43　一个无死锁程序的进度图

练习题 12.15　思考下面的程序，它试图使用一对信号量来实现互斥。

初始时：s=1，t=0。

```
线程 1；          线程 2；

 P(s);            P(s);
 V(s);            V(s);
 P(t);            P(t);
 V(t);            V(t);
```

A. 画出这个程序的进度图。

B. 它会总是死锁吗？

C. 如果是，那么对初始信号量的值做哪些简单的改变就能消除这种潜在的死锁呢？

D. 画出得到的无死锁程序的进度图。

12.8 小结

一个并发程序是由在时间上重叠的一组逻辑流组成的。在这一章中，我们学习了三种不同的构建并发程序的机制：进程、I/O 多路复用和线程。我们以一个并发网络服务器作为贯穿全章的应用程序。

进程是由内核自动调度的，而且因为它们有各自独立的虚拟地址空间，所以要实现共享数据，必须要有显式的 IPC 机制。事件驱动程序创建它们自己的并发逻辑流，这些逻辑流被模型化为状态机，用 I/O 多路复用来显式地调度这些流。因为程序运行在一个单一进程中，所以在流之间共享数据速度很快而且很容易。线程是这些方法的综合。同基于进程的流一样，线程也是由内核自动调度的。同基于 I/O 多路复用的流一样，线程是运行在一个单一进程的上下文中的，因此可以快速而方便地共享数据。

无论哪种并发机制，同步对共享数据的并发访问都是一个困难的问题。提出对信号量的 *P* 和 *V* 操作就是为了帮助解决这个问题。信号量操作可以用来提供对共享数据的互斥访问，也对诸如生产者－消费者程序中有限缓冲区和读者－写者系统中的共享对象这样的资源访问进行调度。一个并发预线程化的 echo 服务器提供了信号量使用场景的很好的例子。

并发也引入了其他一些困难的问题。被线程调用的函数必须具有一种称为线程安全的属性。我们定义了四类线程不安全的函数，以及一些将它们变为线程安全的建议。可重入函数是线程安全函数的一个真子集，它不访问任何共享数据。可重入函数通常比不可重入函数更为有效，因为它们不需要任何同步原语。竞争和死锁是并发程序中出现的另一些困难的问题。当程序员错误地假设逻辑流该如何调度时，就会发生竞争。当一个流等待一个永远不会发生的事件时，就会产生死锁。

参考文献说明

信号量操作是 Dijkstra 提出的 [37]。进度图的概念是 Coffman[24] 提出的，后来由 Carson 和 Reynolds[17] 形式化的。Courtois 等人 [31] 提出了读者－写者问题。操作系统教科书更详细地描述了经典的同步问题，例如哲学家进餐问题、打瞌睡的理发师问题和吸烟者问题 [98，104，112]。Butenhof 的书 [16] 对 Posix 线程接口有全面的描述。Birrell[7] 的论文对线程编程以及线程编程中容易遇到的问题做了很好的介绍。Reinders 的书 [86] 描述了 C/C++ 库，简化了线程化程序的设计和实现。有一些课本讲述了多核系统上并行编程的基础知识 [50，67]。Pugh 描述了 Java 线程通过存储器进行交互的方式的缺陷，并提出了替代的存储器模型 [84]。Gustafson 提出了替代强扩展的弱扩展加速模型 [46]。

www.TopSage.com