

`select` 和 `poll` 都只能工作在相对低效的 LT 模式，而 `epoll` 则可以工作在 ET 高效模式。并且 `epoll` 还支持 EPOLLONESHOT 事件。该事件能进一步减少可读、可写和异常等事件被触发的次数。

从实现原理上来说，`select` 和 `poll` 采用的都是轮询的方式，即每次调用都要扫描整个注册文件描述符集合，并将其中就绪的文件描述符返回给用户程序，因此它们检测就绪事件的算法的时间复杂度是 $O(n)$ 。`epoll_wait` 则不同，它采用的是回调的方式。内核检测到就绪的文件描述符时，将触发回调函数，回调函数就将该文件描述符上对应的事件插入内核就绪事件队列。内核最后在适当的时机将该就绪事件队列中的内容拷贝到用户空间。因此 `epoll_wait` 无须轮询整个文件描述符集合来检测哪些事件已经就绪，其算法时间复杂度是 $O(1)$ 。但是，当活动连接比较多的时候，`epoll_wait` 的效率未必比 `select` 和 `poll` 高，因为此时回调函数被触发得过于频繁。所以 `epoll_wait` 适用于连接数量多，但活动连接较少的情况。

最后，为了便于阅读，我们将这 3 组 I/O 复用系统调用的区别总结于表 9-2 中。

表 9-2 `select`、`poll` 和 `epoll` 的区别

系统调用	<code>select</code>	<code>poll</code>	<code>epoll</code>
事件集合	用户通过 3 个参数分别传入感兴趣的可读、可写及异常等事件，内核通过对这些参数的在线修改来反馈其中的就绪事件。这使得用户每次调用 <code>select</code> 都要重置这 3 个参数	统一处理所有事件类型，因此只需一个事件集参数。用户通过 <code>pollfd.events</code> 传入感兴趣的事件，内核通过修改 <code>pollfd.revents</code> 反馈其中就绪的事件	内核通过一个事件表直接管理用户感兴趣的所有事件。因此每次调用 <code>epoll_wait</code> 时，无须反复传入用户感兴趣的事件。 <code>epoll_wait</code> 系统调用的参数 <code>events</code> 仅用来反馈就绪的事件
应用程序索引就绪文件描述符的时间复杂度	$O(n)$	$O(n)$	$O(1)$
最大支持文件描述符数	一般有最大值限制	65 535	65 535
工作模式	LT	LT	支持 ET 高效模式
内核实现和工作效率	采用轮询方式来检测就绪事件，算法时间复杂度为 $O(n)$	采用轮询方式来检测就绪事件，算法时间复杂度为 $O(n)$	采用回调方式来检测就绪事件，算法时间复杂度为 $O(1)$

9.5 I/O 复用的高级应用一：非阻塞 connect

`connect` 系统调用的 man 手册中有如下一段内容：

EINPROGRESS

The socket is nonblocking and the connection cannot be completed immediately. It is possible to `select(2)` or `poll(2)` for completion by selecting the socket for writing. After `select(2)` indicates writability, use `getsockopt(2)` to read the `SO_ERROR` option at level `SOL_SOCKET` to determine whether `connect()` completed successfully (`SO_ERROR` is zero) or unsuccessfully (`SO_ERROR` is one of the usual error codes listed here, explaining the reason for the failure).

这段话描述了 `connect` 出错时的一种 `errno` 值：`EINPROGRESS`。这种错误发生在对非阻

塞的 socket 调用 connect，而连接又没有立即建立时。根据 man 文档的解释，在这种情况下，我们可以调用 select、poll 等函数来监听这个连接失败的 socket 上的可写事件。当 select、poll 等函数返回后，再利用 getsockopt 来读取错误码并清除该 socket 上的错误。如果错误码是 0，表示连接成功建立，否则连接失败。

通过上面描述的非阻塞 connect 方式，我们就能同时发起多个连接并一起等待。下面看看非阻塞 connect 的一种实现^[2]，如代码清单 9-5 所示。

代码清单 9-5 非阻塞 connect

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <string.h>

#define BUFFER_SIZE 1023

int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

/* 超时连接函数，参数分别是服务器 IP 地址、端口号和超时时间（毫秒）。函数成功时返回已经处于连接状态的 socket，失败则返回 -1 */
int unblock_connect( const char* ip, int port, int time )
{
    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    int sockfd = socket( PF_INET, SOCK_STREAM, 0 );
    int fdopt = setnonblocking( sockfd );
    ret = connect( sockfd, ( struct sockaddr* )&address, sizeof( address ) );
    if ( ret == 0 )
    {
        /* 如果连接成功，则恢复 sockfd 的属性，并立即返回之 */
        printf( "connect with server immediately\n" );
        fcntl( sockfd, F_SETFL, fdopt );
    }
}
```

```

        return sockfd;
    }
    else if ( errno != EINPROGRESS )
    {
        /* 如果连接没有立即建立, 那么只有当 errno 是 EINPROGRESS 时才表示连接还在进行,
否则出错返回 */
        printf( "unblock connect not support\n" );
        return -1;
    }

    fd_set readfds;
    fd_set writefds;
    struct timeval timeout;

    FD_ZERO( &readfds );
    FD_SET( sockfd, &writefds );

    timeout.tv_sec = time;
    timeout.tv_usec = 0;

    ret = select( sockfd + 1, NULL, &writefds, NULL, &timeout );
    if ( ret <= 0 )
    {
        /* select 超时或者出错, 立即返回 */
        printf( "connection time out\n" );
        close( sockfd );
        return -1;
    }

    if ( ! FD_ISSET( sockfd, &writefds ) )
    {
        printf( "no events on sockfd found\n" );
        close( sockfd );
        return -1;
    }

    int error = 0;
    socklen_t length = sizeof( error );
    /* 调用 getsockopt 来获取并清除 sockfd 上的错误 */
    if( getsockopt( sockfd, SOL_SOCKET, SO_ERROR, &error, &length ) < 0 )
    {
        printf( "get socket option failed\n" );
        close( sockfd );
        return -1;
    }
    /* 错误号不为 0 表示连接出错 */
    if( error != 0 )
    {
        printf( "connection failed after select with the error: %d \n", error );
        close( sockfd );
        return -1;
    }
    /* 连接成功 */
}

```

```

        printf( "connection ready after select with the socket: %d \n", sockfd );
        fcntl( sockfd, F_SETFL, fdopt );
        return sockfd;
    }

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int sockfd = unblock_connect( ip, port, 10 );
    if ( sockfd < 0 )
    {
        return 1;
    }
    close( sockfd );
    return 0;
}

```

但遗憾的是，这种方法存在几处移植性问题。首先，非阻塞的 socket 可能导致 connect 始终失败。其次，select 对处于 EINPROGRESS 状态下的 socket 可能不起作用。最后，对于出错的 socket，getsockopt 在有些系统（比如 Linux）上返回 -1（正如代码清单 9-5 所期望的），而在有些系统（比如源自伯克利的 UNIX）上则返回 0。这些问题没有一个统一的解决方法，感兴趣的读者可自行参考相关文献。

9.6 I/O 复用的高级应用二：聊天室程序

像 ssh 这样的登录服务通常要同时处理网络连接和用户输入，这也同样可以使用 I/O 复用来实现。本节我们以 poll 为例实现一个简单的聊天室程序，以阐述如何使用 I/O 复用技术来同时处理网络连接和用户输入。该聊天室程序能让所有用户同时在线群聊，它分为客户端和服务器两个部分。其中客户端程序有两个功能：一是从标准输入终端读入用户数据，并将用户数据发送至服务器；二是往标准输出终端打印服务器发送给它的数据。服务器的功能是接收客户数据，并把客户数据发送给每一个登录到该服务器上的客户端（数据发送者除外）。下面我们依次给出客户端程序和服务器程序的代码。

9.6.1 客户端

客户端程序使用 poll 同时监听用户输入和网络连接，并利用 splice 函数将用户输入内容直接定向到网络连接上以发送之，从而实现数据零拷贝，提高了程序执行效率。客户端程序

如代码清单 9-6 所示。

代码清单 9-6 聊天室客户端程序

```
#define _GNU_SOURCE 1
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <poll.h>
#include <fcntl.h>

#define BUFFER_SIZE 64

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    struct sockaddr_in server_address;
    bzero( &server_address, sizeof( server_address ) );
    server_address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &server_address.sin_addr );
    server_address.sin_port = htons( port );

    int sockfd = socket( PF_INET, SOCK_STREAM, 0 );
    assert( sockfd >= 0 );
    if ( connect( sockfd, ( struct sockaddr* )&server_address, sizeof
        ( server_address ) ) < 0 )
    {
        printf( "connection failed\n" );
        close( sockfd );
        return 1;
    }

    pollfd fds[2];
    /* 注册文件描述符 0（标准输入）和文件描述符 sockfd 上的可读事件 */
    fds[0].fd = 0;
    fds[0].events = POLLIN;
    fds[0].revents = 0;
    fds[1].fd = sockfd;
    fds[1].events = POLLIN | POLLRDHUP;
    fds[1].revents = 0;
```

```

char read_buf[BUFFER_SIZE];
int pipefd[2];
int ret = pipe( pipefd );
assert( ret != -1 );

while( 1 )
{
    ret = poll( fds, 2, -1 );
    if( ret < 0 )
    {
        printf( "poll failure\n" );
        break;
    }

    if( fds[1].revents & POLLRDHUP )
    {
        printf( "server close the connection\n" );
        break;
    }
    else if( fds[1].revents & POLLIN )
    {
        memset( read_buf, '\0', BUFFER_SIZE );
        recv( fds[1].fd, read_buf, BUFFER_SIZE-1, 0 );
        printf( "%s\n", read_buf );
    }

    if( fds[0].revents & POLLIN )
    {
        /* 使用 splice 将用户输入的数据直接写到 sockfd 上（零拷贝） */
        ret = splice( 0, NULL, pipefd[1], NULL, 32768,
                      SPLICE_F_MORE | SPLICE_F_MOVE );
        ret = splice( pipefd[0], NULL, sockfd, NULL, 32768,
                      SPLICE_F_MORE | SPLICE_F_MOVE );
    }
}

close( sockfd );
return 0;
}

```

9.6.2 服务器

服务器程序使用 poll 同时管理监听 socket 和连接 socket，并且使用牺牲空间换取时间的策略来提高服务器性能，如代码清单 9-7 所示。

代码清单 9-7 聊天室服务器程序

```

#define _GNU_SOURCE 1
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

```

```

#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <poll.h>

#define USER_LIMIT 5      /* 最大用户数量 */
#define BUFFER_SIZE 64    /* 读缓冲区的大小 */
#define FD_LIMIT 65535   /* 文件描述符数量限制 */
/* 客户数据：客户端 socket 地址、待写到客户端的数据的位置、从客户端读入的数据 */
struct client_data
{
    sockaddr_in address;
    char* write_buf;
    char buf[ BUFFER_SIZE ];
};

int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    int listenfd = socket( PF_INET, SOCK_STREAM, 0 );
    assert( listenfd >= 0 );

    ret = bind( listenfd, ( struct sockaddr* )&address, sizeof( address ) );
    assert( ret != -1 );

    ret = listen( listenfd, 5 );
}

```

```

assert( ret != -1 );

/* 创建 users 数组，分配 FD_LIMIT 个 client_data 对象。可以预期：每个可能的 socket 连接
都可以获得一个这样的对象，并且 socket 的值可以直接用来索引（作为数组的下标）socket 连接对应的 client_
data 对象，这是将 socket 和客户数据关联的简单而高效的方式 */
client_data* users = new client_data[FD_LIMIT];
/* 尽管我们分配了足够多的 client_data 对象，但为了提高 poll 的性能，仍然有必要限制用户的数量 */
pollfd fds[USER_LIMIT+1];
int user_counter = 0;
for( int i = 1; i <= USER_LIMIT; ++i )
{
    fds[i].fd = -1;
    fds[i].events = 0;
}
fds[0].fd = listenfd;
fds[0].events = POLLIN | POLLERR;
fds[0].revents = 0;

while( 1 )
{
    ret = poll( fds, user_counter+1, -1 );
    if ( ret < 0 )
    {
        printf( "poll failure\n" );
        break;
    }

    for( int i = 0; i < user_counter+1; ++i )
    {
        if( ( fds[i].fd == listenfd ) && ( fds[i].revents & POLLIN ) )
        {
            struct sockaddr_in client_address;
            socklen_t client_addrlength = sizeof( client_address );
            int connfd = accept( listenfd, ( struct sockaddr* )
                                &client_address, &client_addrlength );
            if ( connfd < 0 )
            {
                printf( "errno is: %d\n", errno );
                continue;
            }
            /* 如果请求太多，则关闭新到的连接 */
            if( user_counter >= USER_LIMIT )
            {
                const char* info = "too many users\n";
                printf( "%s", info );
                send( connfd, info, strlen( info ), 0 );
                close( connfd );
                continue;
            }
            /* 对于新的连接，同时修改 fds 和 users 数组。前文已经提到，users[connfd]
            对应于新连接文件描述符 connfd 的客户数据 */
            user_counter++;
            users[connfd].address = client_address;
        }
    }
}

```

```

        setnonblocking( connfd );
        fds[user_counter].fd = connfd;
        fds[user_counter].events = POLLIN | POLLRDHUP | POLLERR;
        fds[user_counter].revents = 0;
        printf( "comes a new user, now have %d users\n", user_counter );
    }
    else if( fds[i].revents & POLLERR )
    {
        printf( "get an error from %d\n", fds[i].fd );
        char errors[ 100 ];
        memset( errors, '\0', 100 );
        socklen_t length = sizeof( errors );
        if( getsockopt( fds[i].fd, SOL_SOCKET, SO_ERROR, &errors,
            &length ) < 0 )
        {
            printf( "get socket option failed\n" );
        }
        continue;
    }
    else if( fds[i].revents & POLLRDHUP )
    {
        /* 如果客户端关闭连接，则服务器也关闭对应的连接，并将用户总数减1 */
        users[fds[i].fd] = users[fds[user_counter].fd];
        close( fds[i].fd );
        fds[i] = fds[user_counter];
        i--;
        user_counter--;
        printf( "a client left\n" );
    }
    else if( fds[i].revents & POLLIN )
    {
        int connfd = fds[i].fd;
        memset( users[connfd].buf, '\0', BUFFER_SIZE );
        ret = recv( connfd, users[connfd].buf, BUFFER_SIZE-1, 0 );
        printf( "get %d bytes of client data %s from %d\n", ret,
            users[connfd].buf, connfd );
        if( ret < 0 )
        {
            /* 如果读操作出错，则关闭连接 */
            if( errno != EAGAIN )
            {
                close( connfd );
                users[fds[i].fd] = users[fds[user_counter].fd];
                fds[i] = fds[user_counter];
                i--;
                user_counter--;
            }
        }
        else if( ret == 0 )
        {
        }
        else
        {

```

```

    /* 如果接收到客户数据，则通知其他 socket 连接准备写数据 */
    for( int j = 1; j <= user_counter; ++j )
    {
        if( fds[j].fd == connfd )
        {
            continue;
        }

        fds[j].events |= ~POLLIN;
        fds[j].events |= POLLOUT;
        users[fds[j].fd].write_buf = users[connfd].buf;
    }
}

else if( fds[i].revents & POLLOUT )
{
    int connfd = fds[i].fd;
    if( !users[connfd].write_buf )
    {
        continue;
    }
    ret = send( connfd, users[connfd].write_buf,
               strlen( users[connfd].write_buf ), 0 );
    users[connfd].write_buf = NULL;
    /* 写完数据后需要重新注册 fds[i] 上的可读事件 */
    fds[i].events |= ~POLLOUT;
    fds[i].events |= POLLIN;
}

delete [] users;
close( listenfd );
return 0;
}

```

9.7 I/O 复用的高级应用三：同时处理 TCP 和 UDP 服务

至此，我们讨论过的服务器程序都只监听一个端口。在实际应用中，有不少服务器程序能同时监听多个端口，比如超级服务 inetd 和 android 的调试服务 adbd。

从 bind 系统调用的参数来看，一个 socket 只能与一个 socket 地址绑定，即一个 socket 只能用来监听一个端口。因此，服务器如果要同时监听多个端口，就必须创建多个 socket，并将它们分别绑定到各个端口上。这样一来，服务器程序就需要同时管理多个监听 socket，I/O 复用技术就有了用武之地。另外，即使是同一个端口，如果服务器要同时处理该端口上的 TCP 和 UDP 请求，则也需要创建两个不同的 socket：一个是流 socket，另一个是数据报 socket，并将它们都绑定到该端口上。比如代码清单 9-8 所示的回射服务器就能同时处理一个端口上的 TCP 和 UDP 请求。

代码清单 9-8 同时处理 TCP 请求和 UDP 请求的回射服务器

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <pthread.h>

#define MAX_EVENT_NUMBER 1024
#define TCP_BUFFER_SIZE 512
#define UDP_BUFFER_SIZE 1024

int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

void addfd( int epollfd, int fd )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET;
    epoll_ctl( epollfd, EPOLL_CTL_ADD, fd, &event );
    setnonblocking( fd );
}

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );
}

```

```

/* 创建 TCP socket，并将其绑定到端口 port 上 */
int listenfd = socket( PF_INET, SOCK_STREAM, 0 );
assert( listenfd >= 0 );

ret = bind( listenfd, ( struct sockaddr* )&address, sizeof( address ) );
assert( ret != -1 );

ret = listen( listenfd, 5 );
assert( ret != -1 );

/* 创建 UDP socket，并将其绑定到端口 port 上 */
bzero( &address, sizeof( address ) );
address.sin_family = AF_INET;
inet_nton( AF_INET, ip, &address.sin_addr );
address.sin_port = htons( port );
int udpfd = socket( PF_INET, SOCK_DGRAM, 0 );
assert( udpfd >= 0 );

ret = bind( udpfd, ( struct sockaddr* )&address, sizeof( address ) );
assert( ret != -1 );

epoll_event events[ MAX_EVENT_NUMBER ];
int epollfd = epoll_create( 5 );
assert( epollfd != -1 );
/* 注册 TCP socket 和 UDP socket 上的可读事件 */
addfd( epollfd, listenfd );
addfd( epollfd, udpfd );

while( 1 )
{
    int number = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, -1 );
    if ( number < 0 )
    {
        printf( "epoll failure\n" );
        break;
    }

    for ( int i = 0; i < number; i++ )
    {
        int sockfd = events[i].data.fd;
        if ( sockfd == listenfd )
        {
            struct sockaddr_in client_address;
            socklen_t client_addrlen = sizeof( client_address );
            int connfd = accept( listenfd, ( struct sockaddr* )
                                &client_address, &client_addrlen );
            addfd( epollfd, connfd );
        }
        else if ( sockfd == udpfd )
        {
            char buf[ UDP_BUFFER_SIZE ];

```

```

        memset( buf, '\0', UDP_BUFFER_SIZE );
        struct sockaddr_in client_address;
        socklen_t client_addrlength = sizeof( client_address );

        ret = recvfrom( udpfd, buf, UDP_BUFFER_SIZE-1, 0,
                        ( struct sockaddr* )&client_address, &client_addrlength );
        if( ret > 0 )
        {
            sendto( udpfd, buf, UDP_BUFFER_SIZE-1, 0,
                    ( struct sockaddr* )&client_address, client_addrlength );
        }
    }
    else if ( events[i].events & EPOLLIN )
    {
        char buf[ TCP_BUFFER_SIZE ];
        while( 1 )
        {
            memset( buf, '\0', TCP_BUFFER_SIZE );
            ret = recv( sockfd, buf, TCP_BUFFER_SIZE-1, 0 );
            if( ret < 0 )
            {
                if( ( errno == EAGAIN ) || ( errno == EWOULDBLOCK ) )
                {
                    break;
                }
                close( sockfd );
                break;
            }
            else if( ret == 0 )
            {
                close( sockfd );
            }
            else
            {
                send( sockfd, buf, ret, 0 );
            }
        }
    }
    else
    {
        printf( "something else happened \n" );
    }
}

close( listenfd );
return 0;
}

```

9.8 超级服务 xinetd

Linux 因特网服务 inetc 是超级服务。它同时管理着多个子服务，即监听多个端口。现在 Linux 系统上使用的 inetc 服务程序通常是其升级版本 xinetd。xinetd 程序的原理与 inetc 相同，但增加了一些控制选项，并提高了安全性。下面我们从配置文件和工作流程两个方面对 xinetd 进行介绍。

9.8.1 xinetd 配置文件

xinetd 采用 /etc/xinetd.conf 主配置文件和 /etc/xinetd.d 目录下的子配置文件来管理所有服务。主配置文件包含的是通用选项，这些选项将被所有子配置文件继承。不过子配置文件可以覆盖这些选项。每一个子配置文件用于设置一个子服务的参数。比如，telnet 子服务的配置文件 /etc/xinetd.d/telnet 的典型内容如下：

```

1 # default: on
2 # description: The telnet server serves telnet sessions; it uses \
3 # unencrypted username/password pairs for authentication.
4 service telnet
5 {
6     flags          = REUSE
7     socket_type   = stream
8     wait           = no
9     user           = root
10    server         = /usr/sbin/in.telnetd
11    log_on_failure += USERID
12    disable        = no
13 }
```

/etc/xinetd.d/telnet 文件中的每一项的含义如表 9-3 所示。

表 9-3 /etc/xinetd.d/telnet 文件的项目及其含义

项 目	含 义
service	服务名
flags	设置连接的标志。REUSE 表示复用 telnet 连接的 socket。该标志已经过时，每个连接都默认启用 REUSE 标志
socket_type	服务类型
wait	服务采用单线程方式 (wait=yes) 还是多线程方式 (wait=no)。单线程方式表示 xinetd 只 accept 第一次连接，此后将由子服务进程来 accept 新连接。多线程方式表示 xinetd 一直负责 accept 连接，而子服务进程仅处理连接 socket 上的数据读写
user	子服务进程将以 user 指定的用户身份运行
server	子服务程序的完整路径
log_on_failure	定义当服务不能启动时输出日志的参数
disable	是否启动该子服务

xinetd 配置文件的内容相当丰富，远不止上面这些。读者可参考其 man 文档来获得更多

信息。

9.8.2 xinetd 工作流程

xinetd 管理的子服务中有的是标准服务，比如时间日期服务 daytime、回射服务 echo 和丢弃服务 discard。xinetd 服务器在内部直接处理这些服务。还有的子服务则需要调用外部的服务器程序来处理。xinetd 通过调用 fork 和 exec 函数来加载运行这些服务器程序。比如 telnet、ftp 服务都是这种类型的子服务。我们仍以 telnet 服务为例来探讨 xinetd 的工作流程。

首先，查看 xinetd 守护进程的 PID（下面的操作都在测试机器 Kongming20 上执行）：

```
$ cat /var/run/xinetd.pid
9543
```

然后开启两个终端并分别使用如下命令 telnet 到本机：

```
$ telnet 192.168.1.109
```

接下来使用 ps 命令查看与进程 9543 相关的进程：

```
$ ps -eo pid,ppid,pgid,sid,comm | grep 9543
  PID  PPID  PGID  SESS  COMMAND
 9543    1  9543   9543  xinetd
 9810  9543  9810   9810  in.telnetd
10355  9543 10355  10355  in.telnetd
```

由此可见，我们每次使用 telnet 登录到 xinetd 服务，它都创建一个子进程来为该 telnet 客户服务。子进程运行 in.telnetd 程序，这是在 /etc/xinetd.d/telnet 配置文件中定义的。每个子进程都处于自己独立的进程组和会话中。我们可以使用 lsof 命令（见第 17 章）进一步查看子进程都打开了哪些文件描述符：

```
$ sudo lsof -p 9810 # 以子进程 9810 为例
in.telnet 9810 root 0u IPv4 48189 0t0 TCP Kongming20:telnet->Kongming20:38763
(ESTABLISHED)
in.telnet 9810 root 1u IPv4 48189 0t0 TCP Kongming20:telnet->Kongming20.:38763
(ESTABLISHED)
in.telnet 9810 root 2u IPv4 48189 0t0 TCP Kongming20:telnet->Kongming20:38763
(ESTABLISHED)
```

这里省略了一些无关的输出。通过 lsof 的输出我们知道，子进程 9810 关闭了其标准输入、标准输出和标准错误，而将 socket 文件描述符 dup 到它们上面。因此，telnet 服务器程序将网络连接上的输入当作标准输入，并把标准输出定向到同一个网络连接上。

再进一步，对 xinetd 进程使用 lsof 命令：

```
$ sudo lsof -p 9543
xinetd 9543 root 5u IPv6 47265 0t0 TCP *:telnet (LISTEN)
```

这一条输出说明 xinetd 将一直监听 telnet 连接请求，因此 in.telnetd 子进程只处理连接 socket，而不处理监听 socket。这是子配置文件中的 wait 参数所定义的行为。

对于内部标准服务，xinetd 的处理流程也可以用上述方法来分析，这里不再赘述。

综合上面讨论的，我们将 xinetd 的工作流程（wait 选项的值是 no 的情况）绘制为图 9-1 所示的形式。

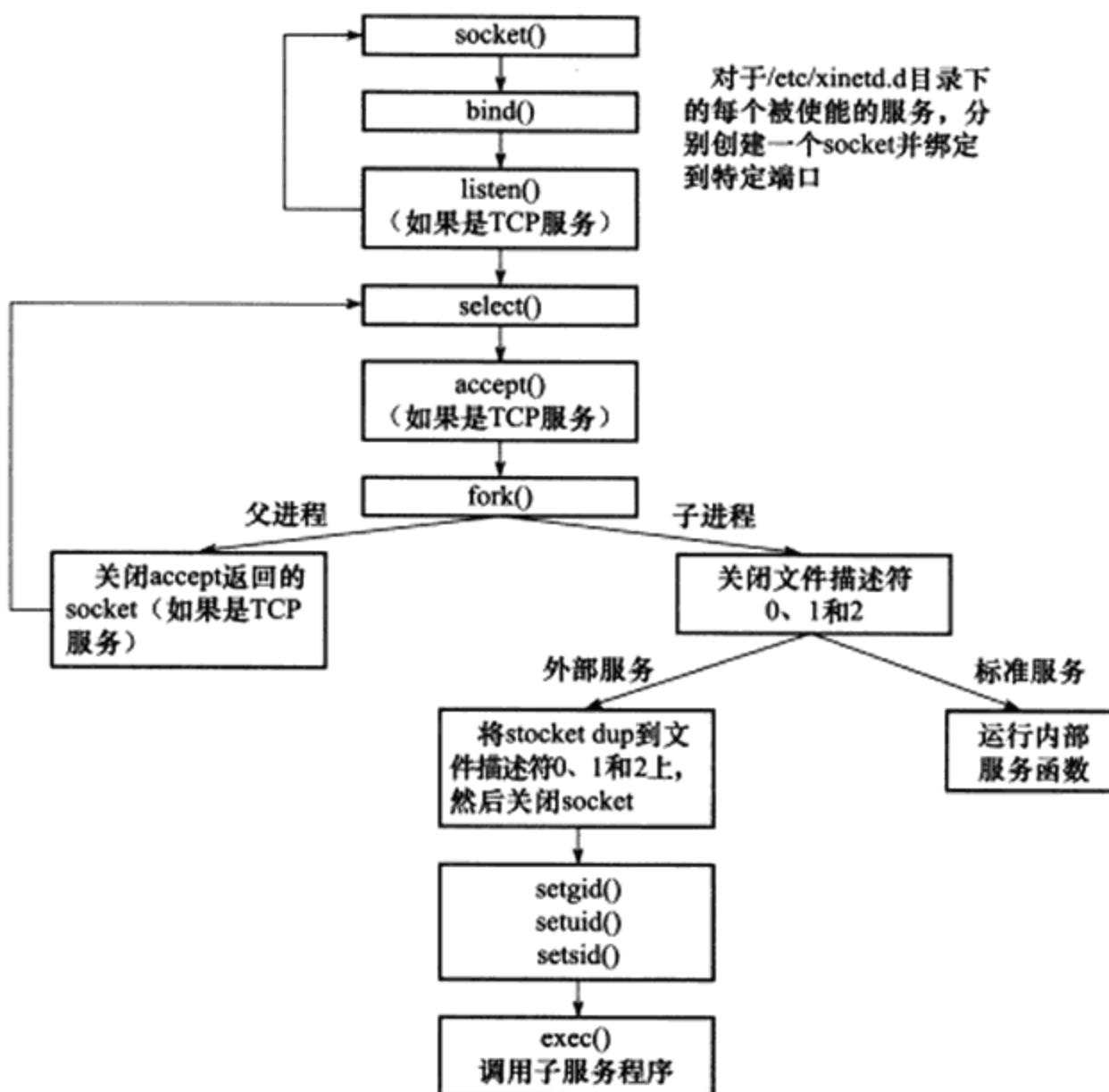


图 9-1 xinetd 的工作流程

第 10 章 信 号

信号是由用户、系统或者进程发送给目标进程的信息，以通知目标进程某个状态的改变或系统异常。Linux 信号可由如下条件产生：

- 对于前台进程，用户可以通过输入特殊的终端字符来给它发送信号。比如输入 Ctrl+C 通常会给进程发送一个中断信号。
- 系统异常。比如浮点异常和非法内存段访问。
- 系统状态变化。比如 alarm 定时器到期将引起 SIGALRM 信号。
- 运行 kill 命令或调用 kill 函数。

服务器程序必须处理（或至少忽略）一些常见的信号，以免异常终止。

本章先讨论如何在程序中发送信号和处理信号，然后讨论 Linux 支持的信号种类，并详细探讨其中和网络编程密切相关的几个。

10.1 Linux 信号概述

10.1.1 发送信号

Linux 下，一个进程给其他进程发送信号的 API 是 kill 函数。其定义如下：

```
#include <sys/types.h>
#include <signal.h>
int kill( pid_t pid, int sig );
```

该函数把信号 sig 发送给目标进程；目标进程由 pid 参数指定，其可能的取值及含义如表 10-1 所示。

表 10-1 kill 函数的 pid 参数及其含义

pid 参数	含 义
pid > 0	信号发送给 PID 为 pid 的进程
pid = 0	信号发送给本进程组内的其他进程
pid = -1	信号发送给除 init 进程外的所有进程，但发送者需要拥有对目标进程发送信号的权限
pid < -1	信号发送给组 ID 为 -pid 的进程组中的所有成员

Linux 定义的信号值都大于 0，如果 sig 取值为 0，则 kill 函数不发送任何信号。但将 sig 设置为 0 可以用来检测目标进程或进程组是否存在，因为检查工作总是在信号发送之前就执行。不过这种检测方式是不可靠的。一方面由于进程 PID 的回绕，可能导致被检测的 PID 不是我们期望的进程的 PID；另一方面，这种检测方法不是原子操作。

该函数成功时返回 0，失败则返回 -1 并设置 errno。几种可能的 errno 如表 10-2 所示。

表 10-2 kill 出错的情况

errno	含 义
EINVAL	无效的信号
EPERM	该进程没有权限发送信号给任何一个目标进程
ESRCH	目标进程或进程组不存在

10.1.2 信号处理方式

目标进程在收到信号时，需要定义一个接收函数来处理之。信号处理函数的原型如下：

```
#include <signal.h>
typedef void (*__sighandler_t) ( int );
```

信号处理函数只带有一个整型参数，该参数用来指示信号类型。信号处理函数应该是可重入的，否则很容易引发一些竞态条件。所以在信号处理函数中严禁调用一些不安全的函数。

除了用户自定义信号处理函数外，bits/signum.h 头文件中还定义了信号的两种其他处理方式——SIG_IGN 和 SIG_DEL：

```
#include <bits/signum.h>
#define SIG_DFL ((__sighandler_t) 0)
#define SIG_IGN ((__sighandler_t) 1)
```

SIG_IGN 表示忽略目标信号，SIG_DFL 表示使用信号的默认处理方式。信号的默认处理方式有如下几种：结束进程（Term）、忽略信号（Ign）、结束进程并生成核心转储文件（Core）、暂停进程（Stop），以及继续进程（Cont）。

10.1.3 Linux 信号

Linux 的可用信号都定义在 bits/signum.h 头文件中，其中包括标准信号和 POSIX 实时信号。本书仅讨论标准信号，如表 10-3 所示。

表 10-3 Linux 标准信号

信 号	起 源	默认行为	含 义
SIGHUP	POSIX	Term	控制终端挂起
SIGINT	ANSI	Term	键盘输入以中断进程（Ctrl+C）
SIGQUIT	POSIX	Core	键盘输入使进程退出（Ctrl+\）
SIGILL	ANSI	Core	非法指令
SIGTRAP	POSIX	Core	断点陷阱，用于调试
SIGABRT	ANSI	Core	进程调用 abort 函数时生成该信号
SIGIOT	4.2 BSD	Core	和 SIGABRT 相同
SIGBUS	4.2 BSD	Core	总线错误，错误内存访问

(续)

信号	起源	默认行为	含义
SIGFPE	ANSI	Core	浮点异常
SIGKILL	POSIX	Term	终止一个进程。该信号不可被捕获或者忽略
SIGUSR1	POSIX	Term	用户自定义信号之一
SIGSEGV	ANSI	Core	非法内存段引用
SIGUSR2	POSIX	Term	用户自定义信号之二
SIGPIPE	POSIX	Term	往读端被关闭的管道或者 socket 连接中写数据
SIGALRM	POSIX	Term	由 alarm 或 setitimer 设置的实时闹钟超时引起
SIGTERM	ANSI	Term	终止进程。kill 命令默认发送的信号就是 SIGTERM
SIGSTKFLT	Linux	Term	早期的 Linux 使用该信号来报告数学协处理器栈错误
SIGCLD	System V	Ign	和 SIGCHLD 相同
SIGCHLD	POSIX	Ign	子进程状态发生变化（退出或者暂停）
SIGCONT	POSIX	Cont	启动被暂停的进程（Ctrl+Q）。如果目标进程未处于暂停状态，则信号被忽略
SIGSTOP	POSIX	Stop	暂停进程（Ctrl+S）。该信号不可被捕获或者忽略
SIGTSTP	POSIX	Stop	挂起进程（Ctrl+Z）
SIGTTIN	POSIX	Stop	后台进程试图从终端读取输入
SIGTTOU	POSIX	Stop	后台进程试图往终端输出内容
SIGURG	4.2 BSD	Ign	socket 连接上接收到紧急数据
SIGXCPU	4.2 BSD	Core	进程的 CPU 使用时间超过其软限制
SIGXFSZ	4.2 BSD	Core	文件尺寸超过其软限制
SIGVTALRM	4.2 BSD	Term	与 SIGALRM 类似，不过它只统计本进程用户空间代码的运行时间
SIGPROF	4.2 BSD	Term	与 SIGALRM 类似，它同时统计用户代码和内核的运行时间
SIGWINCH	4.3 BSD	Ign	终端窗口大小发生变化
SIGPOLL	System V	Term	与 SIGIO 类似
SIGIO	4.2 BSD	Term	IO 就绪，比如 socket 上发生可读、可写事件。因为 TCP 服务器可触发 SIGIO 的条件很多，故而 SIGIO 无法在 TCP 服务器中使用。SIGIO 信号可用在 UDP 服务器中，不过也非常少见
SIGPWR	System V	Term	对于使用 UPS (Uninterruptible Power Supply) 的系统，当电池电量过低时，SIGPWR 信号将被触发
SIGSYS	POSIX	Core	非法系统调用
SIGUNUSED		Core	保留，通常和 SIGSYS 效果相同

我们并不需要在代码中处理所有这些信号。本章后面将重点介绍与网络编程关系紧密的几个信号：SIGHUP、SIGPIPE 和 SIGURG。后续章节还将介绍 SIGALRM、SIGCHLD 等信号的使用。

10.1.4 中断系统调用

如果程序在执行处于阻塞状态的系统调用时接收到信号，并且我们为该信号设置了信号处理函数，则默认情况下系统调用将被中断，并且 `errno` 被设置为 `EINTR`。我们可以使用 `sigaction` 函数（见后文）为信号设置 `SA_RESTART` 标志以自动重启被该信号中断的系统调用。

对于默认行为是暂停进程的信号（比如 `SIGSTOP`、`SIGTTIN`），如果我们没有为它们设置信号处理函数，则它们也可以中断某些系统调用（比如 `connect`、`epoll_wait`）。POSIX 没有规定这种行为，这是 Linux 独有的。

10.2 信号函数

10.2.1 signal 系统调用

要为一个信号设置处理函数，可以使用下面的 `signal` 系统调用：

```
#include <signal.h>
_sighandler_t signal ( int sig, _sighandler_t _handler )
```

`sig` 参数指出要捕获的信号类型。`_handler` 参数是 `_sighandler_t` 类型的函数指针，用于指定信号 `sig` 的处理函数。

`signal` 函数成功时返回一个函数指针，该函数指针的类型也是 `_sighandler_t`。这个返回值是前一次调用 `signal` 函数时传入的函数指针，或者是信号 `sig` 对应的默认处理函数指针 `SIG_DEF`（如果是第一次调用 `signal` 的话）。

`signal` 系统调用出错时返回 `SIG_ERR`，并设置 `errno`。

10.2.2 sigaction 系统调用

设置信号处理函数的更健壮的接口是如下的系统调用：

```
#include <signal.h>
int sigaction( int sig, const struct sigaction* act, struct sigaction* oact );
```

`sig` 参数指出要捕获的信号类型，`act` 参数指定新的信号处理方式，`oact` 参数则输出信号先前的处理方式（如果不为 `NULL` 的话）。`act` 和 `oact` 都是 `sigaction` 结构体类型的指针，`sigaction` 结构体描述了信号处理的细节，其定义如下：

```
struct sigaction
{
#ifdef __USE_POSIX199309
    union
    {
        _sighandler_t sa_handler;
        void (*sa_sigaction) ( int, siginfo_t*, void* );
    }
}
```

```

    _sigaction_handler;
#define sa_handler      __sigaction_handler.sa_handler
#define sa_sigaction    __sigaction_handler.sa_sigaction
#else
    _sighandler_t sa_handler;
#endif

    _sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer) (void);
};

```

该结构体中的 `sa_handler` 成员指定信号处理函数。`sa_mask` 成员设置进程的信号掩码（确切地说是在进程原有信号掩码的基础上增加信号掩码），以指定哪些信号不能发送给本进程。`sa_mask` 是信号集 `sigset_t`（`_sigset_t` 的同义词）类型，该类型指定一组信号。关于信号集，我们将在后面介绍。`sa_flags` 成员用于设置程序收到信号时的行为，其可选值如表 10-4 所示。

表 10-4 `sa_flags` 选项

选 项	含 义
<code>SA_NOCLDSTOP</code>	如果 <code>sigaction</code> 的 <code>sig</code> 参数是 <code>SIGCHLD</code> ，则设置该标志表示子进程暂停时不生成 <code>SIGCHLD</code> 信号
<code>SA_NOCLDWAIT</code>	如果 <code>sigaction</code> 的 <code>sig</code> 参数是 <code>SIGCHLD</code> ，则设置该标志表示子进程结束时不产生僵尸进程
<code>SA_SIGINFO</code>	使用 <code>sa_sigaction</code> 作为信号处理函数（而不是默认的 <code>sa_handler</code> ），它给进程提供更多相关的信息
<code>SA_ONSTACK</code>	调用由 <code>sigaltstack</code> 函数设置的可选信号栈上的信号处理函数
<code>SA_RESTART</code>	重新调用被该信号终止的系统调用
<code>SA_NODEFER</code>	当接收到信号并进入其信号处理函数时，不屏蔽该信号。默认情况下，我们期望进程在处理一个信号时不再接收到同种信号，否则将引起一些竞态条件
<code>SA_RESETHAND</code>	信号处理函数执行完以后，恢复信号的默认处理方式
<code>SA_INTERRUPT</code>	中断系统调用
<code>SA_NOMASK</code>	同 <code>SA_NODEFER</code>
<code>SA_ONESHOT</code>	同 <code>SA_RESETHAND</code>
<code>SA_STACK</code>	同 <code>SA_ONSTACK</code>

`sa_restorer` 成员已经过时，最好不要使用。`sigaction` 成功时返回 0，失败则返回 -1 并设置 `errno`。

10.3 信号集

10.3.1 信号集函数

前文提到，Linux 使用数据结构 `sigset_t` 来表示一组信号。其定义如下：

```
#include <bits/sigset.h>
```

```
# define _SIGSET_NWORDS (1024 / (8 * sizeof (unsigned long int)))
typedef struct
{
    unsigned long int __val[_SIGSET_NWORDS];
} __sigset_t;
```

由该定义可见，`sigset_t` 实际上是一个长整型数组，数组的每个元素的每个位表示一个信号。这种定义方式和文件描述符集 `fd_set` 类似。Linux 提供了如下一组函数来设置、修改、删除和查询信号集：

```
#include <signal.h>
int sigemptyset (sigset_t* __set)           /* 清空信号集 */
int sigfillset (sigset_t* __set)             /* 在信号集中设置所有信号 */
int sigaddset (sigset_t* __set, int __signo) /* 将信号 __signo 添加至信号集中 */
int sigdelset (sigset_t* __set, int __signo) /* 将信号 __signo 从信号集中删除 */
int sigismember (_const sigset_t* __set, int __signo) /* 测试 __signo 是否在信号集中 */
```

10.3.2 进程信号掩码

前文提到，我们可以利用 `sigaction` 结构体的 `sa_mask` 成员来设置进程的信号掩码。此外，如下函数也可以用于设置或查看进程的信号掩码：

```
#include <signal.h>
int sigprocmask( int __how, _const sigset_t* __set, sigset_t* __oset );
```

`_set` 参数指定新的信号掩码，`_oset` 参数则输出原来的信号掩码（如果不为 `NULL` 的话）。如果 `_set` 参数不为 `NULL`，则 `_how` 参数指定设置进程信号掩码的方式，其可选值如表 10-5 所示。

表 10-5 `_how` 参数

<code>_how</code> 参数	含 义
<code>SIG_BLOCK</code>	新的进程信号掩码是其当前值和 <code>_set</code> 指定信号集的并集
<code>SIG_UNBLOCK</code>	新的进程信号掩码是其当前值和 <code>~_set</code> 信号集的交集，因此 <code>_set</code> 指定的信号集将不被屏蔽
<code>SIG_SETMASK</code>	直接将进程信号掩码设置为 <code>_set</code>

如果 `_set` 为 `NULL`，则进程信号掩码不变，此时我们仍然可以利用 `_oset` 参数来获得进程当前的信号掩码。

`sigprocmask` 成功时返回 0，失败则返回 -1 并设置 `errno`。

10.3.3 被挂起的信号

设置进程信号掩码后，被屏蔽的信号将不能被进程接收。如果给进程发送一个被屏蔽的信号，则操作系统将该信号设置为进程的一个被挂起的信号。如果我们取消对被挂起信号的屏蔽，则它能立即被进程接收到。如下函数可以获得进程当前被挂起的信号集：

```
#include <signal.h>
int sigpending( sigset_t* set );
```

`set` 参数用于保存被挂起的信号集。显然，进程即使多次接收到同一个被挂起的信号，`sigpending` 函数也只能反映一次。并且，当我们再次使用 `sigprocmask` 使能该挂起的信号时，该信号的处理函数也只被触发一次。

`sigpending` 成功时返回 0，失败时返回 -1 并设置 `errno`。

关于信号和信号集，Linux 还提供了很多有用的 API，这里就不一一介绍了。需要提醒读者的是，要始终清楚地知道进程在每个运行时刻的信号掩码，以及如何适当地处理捕获到的信号。在多进程、多线程环境中，我们要以进程、线程为单位来处理信号和信号掩码。我们不能设想新创建的进程、线程具有和父进程、主线程完全相同的信号特征。比如，`fork` 调用产生的子进程将继承父进程的信号掩码，但具有一个空的挂起信号集。

10.4 统一事件源

信号是一种异步事件：信号处理函数和程序的主循环是两条不同的执行路线。很显然，信号处理函数需要尽可能快地执行完毕，以确保该信号不被屏蔽（前面提到过，为了避免一些竞态条件，信号在处理期间，系统不会再次触发它）太久。一种典型的解决方案是：把信号的主要处理逻辑放到程序的主循环中，当信号处理函数被触发时，它只是简单地通知主循环程序接收到信号，并把信号值传递给主循环，主循环再根据接收到的信号值执行目标信号对应的逻辑代码。信号处理函数通常使用管道来将信号“传递”给主循环：信号处理函数往管道的写端写入信号值，主循环则从管道的读端读出该信号值。那么主循环怎么知道管道上何时有数据可读呢？这很简单，我们只需要使用 I/O 复用系统调用来监听管道的读端文件描述符上的可读事件。如此一来，信号事件就能和其他 I/O 事件一样被处理，即统一事件源。

很多优秀的 I/O 框架库和后台服务器程序都统一处理信号和 I/O 事件，比如 Libevent I/O 框架库和 xinetd 超级服务。代码清单 10-1 给出了统一事件源的一个简单实现。

代码清单 10-1 统一事件源

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <pthread.h>
```

```

#define MAX_EVENT_NUMBER 1024
static int pipefd[2];

int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

void addfd( int epollfd, int fd )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET;
    epoll_ctl( epollfd, EPOLL_CTL_ADD, fd, &event );
    setnonblocking( fd );
}

/* 信号处理函数 */
void sig_handler( int sig )
{
    /* 保留原来的errno，在函数最后恢复，以保证函数的可重入性 */
    int save_errno = errno;
    int msg = sig;
    send( pipefd[1], ( char* )&msg, 1, 0 ); /* 将信号值写入管道，以通知主循环 */
    errno = save_errno;
}

/* 设置信号的处理函数 */
void addsig( int sig )
{
    struct sigaction sa;
    memset( &sa, '\0', sizeof( sa ) );
    sa.sa_handler = sig_handler;
    sa.sa_flags |= SA_RESTART;
    sigfillset( &sa.sa_mask );
    assert( sigaction( sig, &sa, NULL ) != -1 );
}

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;

```

```

inet_pton( AF_INET, ip, &address.sin_addr );
address.sin_port = htons( port );

int listenfd = socket( PF_INET, SOCK_STREAM, 0 );
assert( listenfd >= 0 );

ret = bind( listenfd, ( struct sockaddr* )&address, sizeof( address ) );
if( ret == -1 )
{
    printf( "errno is %d\n", errno );
    return 1;
}
ret = listen( listenfd, 5 );
assert( ret != -1 );

epoll_event events[ MAX_EVENT_NUMBER ];
int epollfd = epoll_create( 5 );
assert( epollfd != -1 );
addfd( epollfd, listenfd );

/* 使用 socketpair 创建管道，注册 pipefd[0] 上的可读事件 */
ret = socketpair( PF_UNIX, SOCK_STREAM, 0, pipefd );
assert( ret != -1 );
setnonblocking( pipefd[1] );
addfd( epollfd, pipefd[0] );

/* 设置一些信号的处理函数 */
addsig( SIGHUP );
addsig( SIGCHLD );
addsig( SIGTERM );
addsig( SIGINT );
bool stop_server = false;

while( !stop_server )
{
    int number = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, -1 );
    if ( ( number < 0 ) && ( errno != EINTR ) )
    {
        printf( "epoll failure\n" );
        break;
    }

    for ( int i = 0; i < number; i++ )
    {
        int sockfd = events[i].data.fd;
        /* 如果就绪的文件描述符是 listenfd，则处理新的连接 */
        if( sockfd == listenfd )
        {
            struct sockaddr_in client_address;
            socklen_t client_addrlength = sizeof( client_address );
            int connfd = accept( listenfd, ( struct sockaddr* )
                                &client_address,&client_addrlength );

```

```

        addfd( epollfd, connfd );
    }
    /* 如果就绪的文件描述符是 pipefd[0], 则处理信号 */
    else if( ( sockfd == pipefd[0] ) && ( events[i].events & EPOLLIN ) )
    {
        int sig;
        char signals[1024];
        ret = recv( pipefd[0], signals, sizeof( signals ), 0 );
        if( ret == -1 )
        {
            continue;
        }
        else if( ret == 0 )
        {
            continue;
        }
        else
        {
            /* 因为每个信号值占 1 字节, 所以按字节来逐个接收信号。我们以 SIGTERM
           为例, 来说明如何安全地终止服务器主循环 */
            for( int i = 0; i < ret; ++i )
            {
                switch( signals[i] )
                {
                    case SIGCHLD:
                    case SIGHUP:
                    {
                        continue;
                    }
                    case SIGTERM:
                    case SIGINT:
                    {
                        stop_server = true;
                    }
                }
            }
        }
    }
    else
    {
    }
}
}

printf( "close fds\n" );
close( listenfd );
close( pipefd[1] );
close( pipefd[0] );
return 0;
}

```

10.5 网络编程相关信号

本节中我们详细探讨三个和网络编程密切相关的信号。

10.5.1 SIGHUP

当挂起进程的控制终端时，SIGHUP 信号将被触发。对于没有控制终端的网络后台程序而言，它们通常利用 SIGHUP 信号来强制服务器重读配置文件。一个典型的例子是 xinetd 超级服务程序。

xinetd 程序在接收到 SIGHUP 信号之后将调用 hard_reconfig 函数（见 xinetd 源码），它循环读取 /etc/xinetd.d/ 目录下的每个子配置文件，并检测其变化。如果某个正在运行的子服务的配置文件被修改以停止服务，则 xinetd 主进程将给该子服务进程发送 SIGTERM 信号以结束它。如果某个子服务的配置文件被修改以开启服务，则 xinetd 将创建新的 socket 并将其绑定到该服务对应的端口上。下面我们简单地分析 xinetd 处理 SIGHUP 信号的流程。

测试机器 Kongming20 上具有如下环境：

```
$ ps -ef | grep xinetd
root 7438 1 0 11:32 ? 00:00:00 /usr/sbin/xinetd -stayalive -pidfile /var/run/xinetd.pid
root 7442 7438 0 11:32 ? 00:00:00 (xinetd service) echo-stream Kongming20
$ sudo lsof -p 7438
xinetd 7438 root 3r FIFO 0,8 0t0 37639 pipe
xinetd 7438 root 4w FIFO 0,8 0t0 37639 pipe
xinetd 7438 root 5u IPv6 37652 0t0 TCP *:echo (LISTEN)
```

从 ps 的输出来看，xinetd 创建了子进程 7442，它运行 echo-stream 内部服务。从 lsof 的输出来看，xinetd 打开了一个管道。该管道的读端文件描述符的值是 3，写端文件描述符的值是 4。后面我们将看到，它们的作用就是统一事件源。现在我们修改 /etc/xinetd.d/ 目录下的部分配置文件，并给 xinetd 发送一个 SIGHUP 信号。具体操作如下：

```
$ sudo sed -i 's/disable.*=.*no/disable = yes/' /etc/xinetd.d/echo-stream # 停止echo服务
$ sudo sed -i 's/disable.*=.*yes/disable = no/' /etc/xinetd.d/telnet # 开启telnet服务
$ sudo strace -p 7438 &> a.txt
$ sudo kill -HUP xinetd
```

strace 命令（见第 17 章）能跟踪程序执行时调用的系统调用和接收到的信号。这里我们利用 strace 命令跟踪进程 7438，即 xinetd 服务器程序，以观察 xinetd 是如何处理 SIGHUP 信号的。此次 strace 命令的部分输出如代码清单 10-2 所示。

代码清单 10-2 用 strace 命令查看 xinetd 处理 SIGHUP 的流程

```
--- {si_signo=SIGHUP, si_code=SI_USER, si_pid=7697, si_uid=0,
si_value={int=1154706400, ptr=0x44d36be0}} (Hangup) ---
```

```

write(4, "\1", 1) = 1
sigreturn() = ? (mask now [])
poll([{fd=5, events=POLLIN}, {fd=3, events=POLLIN}], 2, -1) = 1 ({[fd=3,
    revents=POLLIN]}) 
ioctl(3, FIONREAD, [1]) = 0
read(3, "\1", 1) = 1

stat64("/etc/xinetd.d/echo-stream", {st_mode=S_IFREG|0644, st_size=1149, ...}) = 0
open("/etc/xinetd.d/echo-stream", O_RDONLY) = 8
time(NULL) = 1337053896
send(7, "<31>May 15 11:51:36 xinetd[7438]"..., 139, MSG_NOSIGNAL) = 139
fstat64(8, {st_mode=S_IFREG|0644, st_size=1149, ...}) = 0
lseek(8, 0, SEEK_CUR) = 0
fcntl64(8, F_GETFL) = 0 (flags O_RDONLY)
read(8, "# This is the configuration for "..., 8192) = 1149
read(8, "", 8192) = 0
close(8) = 0

kill(7442, SIGTERM) = 0
waitpid(7442, NULL, WNOHANG) = 0

socket(PF_INET6, SOCK_STREAM, IPPROTO_TCP) = 5
fcntl64(5, F_SETFD, FD_CLOEXEC) = 0
setsockopt(5, SOL_IPV6, IPV6_V6ONLY, [0], 4) = 0
setsockopt(5, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(5, {sa_family=AF_INET6, sin6_port=htons(23), inet_nton(AF_INET6, "::",
    &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, 28) = 0
listen(5, 64) = 0

```

该输出分为 4 个部分，我们用空行将每个部分隔开。

第一部分描述程序接收到 SIGHUP 信号时，信号处理函数使用管道通知主程序该信号的到来。信号处理函数往文件描述符 4（管道的写端）写入信号值 1（SIGHUP 信号），而主程序使用 poll 检测到文件描述符 3（管道的读端）上有可读事件，就将管道上的数据读入。

第二部分描述了 xinetd 重新读取一个子配置文件的过程。

第三部分描述了 xinetd 给子进程 echo-stream (PID 为 7442) 发送 SIGTERM 信号来终止该子进程，并调用 waitpid 来等待该子进程结束。

第四部分描述了 xinetd 启动 telnet 服务的过程：创建一个流服务 socket 并将其绑定到端口 23 上，然后监听该端口。

10.5.2 SIGPIPE

默认情况下，往一个读端关闭的管道或 socket 连接中写数据将引发 SIGPIPE 信号。我们需要在代码中捕获并处理该信号，或者至少忽略它，因为程序接收到 SIGPIPE 信号的默认行为是结束进程，而我们绝对不希望因为错误的写操作而导致程序退出。引起 SIGPIPE 信号的写操作将设置 errno 为 EPIPE。

第 5 章提到，我们可以使用 send 函数的 MSG_NOSIGNAL 标志来禁止写操作触发

SIGPIPE 信号。在这种情况下，我们应该使用 send 函数反馈的 errno 值来判断管道或者 socket 连接的读端是否已经关闭。

此外，我们也可以利用 I/O 复用系统调用来检测管道和 socket 连接的读端是否已经关闭。以 poll 为例，当管道的读端关闭时，写端文件描述符上的 POLLHUP 事件将被触发；当 socket 连接被对方关闭时，socket 上的 POLLRDHUP 事件将被触发。

10.5.3 SIGURG

在 Linux 环境下，内核通知应用程序带外数据到达主要有两种方法：一种是第 9 章介绍的 I/O 复用技术，select 等系统调用在接收到带外数据时将返回，并向应用程序报告 socket 上的异常事件，代码清单 9-1 给出了一个这方面的例子；另外一种方法就是使用 SIGURG 信号，如代码清单 10-3 所示。

代码清单 10-3 用 SIGURG 检测带外数据是否到达

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>

#define BUF_SIZE 1024

static int connfd;
/* SIGURG 信号的处理函数 */
void sig_urg( int sig )
{
    int save_errno = errno;
    char buffer[ BUF_SIZE ];
    memset( buffer, '\0', BUF_SIZE );
    int ret = recv( connfd, buffer, BUF_SIZE-1, MSG_OOB ); /* 接收带外数据 */
    printf( "got %d bytes of oob data '%s'\n", ret, buffer );
    errno = save_errno;
}

void addsig( int sig, void ( *sig_handler )( int ) )
{
    struct sigaction sa;
    memset( &sa, '\0', sizeof( sa ) );
    sa.sa_handler = sig_handler;
    sa.sa_flags |= SA_RESTART;
    sigfillset( &sa.sa_mask );
    assert( sigaction( sig, &sa, NULL ) != -1 );
}
```

```
)\n\nint main( int argc, char* argv[] )\n{\n    if( argc <= 2 )\n    {\n        printf( "usage: %s ip_address port_number\\n", basename( argv[0] ) );\n        return 1;\n    }\n    const char* ip = argv[1];\n    int port = atoi( argv[2] );\n\n    struct sockaddr_in address;\n    bzero( &address, sizeof( address ) );\n    address.sin_family = AF_INET;\n    inet_pton( AF_INET, ip, &address.sin_addr );\n    address.sin_port = htons( port );\n\n    int sock = socket( PF_INET, SOCK_STREAM, 0 );\n    assert( sock >= 0 );\n\n    int ret = bind( sock, ( struct sockaddr* )&address, sizeof( address ) );\n    assert( ret != -1 );\n\n    ret = listen( sock, 5 );\n    assert( ret != -1 );\n\n    struct sockaddr_in client;\n    socklen_t client_addrlength = sizeof( client );\n    connfd = accept( sock, ( struct sockaddr* )&client, &client_addrlength );\n    if ( connfd < 0 )\n    {\n        printf( "errno is: %d\\n", errno );\n    }\n    else\n    {\n        addsig( SIGURG, sig_urg );\n        /* 使用 SIGURG 信号之前，我们必须设置 socket 的宿主进程或进程组 */\n        fcntl( connfd, F_SETOWN, getpid() );\n\n        char buffer[ BUF_SIZE ];\n        while( 1 ) /* 循环接收普通数据 */\n        {\n            memset( buffer, '\\0', BUF_SIZE );\n            ret = recv( connfd, buffer, BUF_SIZE-1, 0 );\n            if( ret <= 0 )\n            {\n                break;\n            }\n            printf( "got %d bytes of normal data '%s'\\n", ret, buffer );\n        }\n\n        close( connfd );\n    }\n}
```

```
    }

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

读者不妨编译并运行该服务器程序，然后使用代码清单 5-6 所描述的客户端程序来往该服务器程序发送数据，以观察服务器是如何同时处理普通数据和带外数据的。

至此，我们讨论完了 TCP 带外数据相关的所有知识。下面帮助读者重新梳理一下。3.8 节中我们介绍了 TCP 带外数据的基本知识，其中探讨了 TCP 模块是如何发送和接收带外数据的。5.8.1 小节描述了如何在应用程序中使用带 MSG_OOB 标志的 send/recv 系统调用来发送 / 接收带外数据，并给出了相关代码。9.1.3 小节和 10.5.3 小节分别介绍了检测带外数据是否到达的两种方法：I/O 复用系统调用报告的异常事件和 SIGURG 信号。但应用程序检测到带外数据到达后，我们还需要进一步判断带外数据在数据流中的具体位置，才能够准确无误地读取带外数据。5.9 节介绍的 socketmark 系统调用就是专门用于解决这个问题的。它判断一个 socket 是否处于带外标记，即该 socket 上下一个将被读取到的数据是否是带外数据。

第 11 章 定时器^①

网络程序需要处理的第三类事件是定时事件，比如定期检测一个客户连接的活动状态。服务器程序通常管理着众多定时事件，因此有效地组织这些定时事件，使之能在预期的时间点被触发且不影响服务器的主要逻辑，对于服务器的性能有着至关重要的影响。为此，我们要将每个定时事件分别封装成定时器，并使用某种容器类数据结构，比如链表、排序链表和时间轮，将所有定时器串联起来，以实现对定时事件的统一管理。本章主要讨论的就是两种高效的管理定时器的容器：时间轮和时间堆。

不过，在讨论如何组织定时器之前，我们先要介绍定时的方法。定时是指在一段时间之后触发某段代码的机制，我们可以在这段代码中依次处理所有到期的定时器。换言之，定时机制是定时器得以被处理的原动力。Linux 提供了三种定时方法，它们是：

- socket 选项 SO_RCVTIMEO 和 SO_SNDFTIMEO。
- SIGALRM 信号。
- I/O 复用系统调用的超时参数。

11.1 socket 选项 SO_RCVTIMEO 和 SO_SNDFTIMEO

第 5 章中我们介绍过 socket 选项 SO_RCVTIMEO 和 SO_SNDFTIMEO，它们分别用来设置 socket 接收数据超时时间和发送数据超时时间。因此，这两个选项仅对与数据接收和发送相关的 socket 专用系统调用（socket 专用的系统调用指的是 5.2 ~ 5.11 节介绍的那些 socket API）有效，这些系统调用包括 send、sendmsg、recv、recvmsg、accept 和 connect。我们将选项 SO_RCVTIMEO 和 SO_SNDFTIMEO 对这些系统调用的影响总结于表 11-1 中。

表 11-1 SO_RCVTIMEO 和 SO_SNDFTIMEO 选项的作用

系统调用	有效选项	系统调用超时后的行为
send	SO_SNDFTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
sendmsg	SO_SNDFTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
recv	SO_RCVTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
recvmsg	SO_RCVTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
accept	SO_RCVTIMEO	返回 -1，设置 errno 为 EAGAIN 或 EWOULDBLOCK
connect	SO_SNDFTIMEO	返回 -1，设置 errno 为 EINPROGRESS

① 本章的标题叫定时器，这是行业内常用的叫法。实际上，其确切的叫法是定时器容器。二者常混谈，本书也没有刻意区分。不过，从本章的第一段话还是能看出二者的区别：定时器容器是容器类数据结构，比如时间轮；定时器则是容器内容纳的一个个对象，它是对定时事件的封装。

由表 11-1 可见，在程序中，我们可以根据系统调用（`send`、`sendmsg`、`recv`、`recvmsg`、`accept` 和 `connect`）的返回值以及 `errno` 来判断超时时间是否已到，进而决定是否开始处理定时任务。代码清单 11-1 以 `connect` 为例，说明程序中如何使用 `SO_SNDFTIMEO` 选项来定时。

代码清单 11-1 设置 `connect` 超时时间

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
/* 超时连接函数 */
int timeout_connect( const char* ip, int port, int time )
{
    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    int sockfd = socket( PF_INET, SOCK_STREAM, 0 );
    assert( sockfd >= 0 );
    /* 通过选项 SO_RCVTIMEO 和 SO_SNDFTIMEO 所设置的超时时间的类型是 timeval，这和 select
       系统调用的超时参数类型相同 */
    struct timeval timeout;
    timeout.tv_sec = time;
    timeout.tv_usec = 0;
    socklen_t len = sizeof( timeout );
    ret = setsockopt( sockfd, SOL_SOCKET, SO_SNDFTIMEO, &timeout, len );
    assert( ret != -1 );

    ret = connect( sockfd, ( struct sockaddr* )&address, sizeof( address ) );
    if ( ret == -1 )
    {
        /* 超时对应的错误号是 EINPROGRESS。下面这个条件如果成立，我们就可以处理定时任务了 */
        if( errno == EINPROGRESS )
        {
            printf( "connecting timeout, process timeout logic \n" );
            return -1;
        }
        printf( "error occur when connecting to server\n" );
        return -1;
    }
}
```

```

    return sockfd;
}

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int sockfd = timeout_connect( ip, port, 10 );
    if ( sockfd < 0 )
    {
        return 1;
    }
    return 0;
}

```

11.2 SIGALRM 信号

第 10 章提到，由 `alarm` 和 `setitimer` 函数设置的实时闹钟一旦超时，将触发 `SIGALRM` 信号。因此，我们可以利用该信号的信号处理函数来处理定时任务。但是，如果要处理多个定时任务，我们就需要不断地触发 `SIGALRM` 信号，并在其信号处理函数中执行到期的任务。一般而言，`SIGALRM` 信号按照固定的频率生成，即由 `alarm` 或 `setitimer` 函数设置的定时周期 T 保持不变。如果某个定时任务的超时时间不是 T 的整数倍，那么它实际被执行的时间和预期的时间将略有偏差。因此定时周期 T 反映了定时的精度。

本节中我们通过一个实例——处理非活动连接，来介绍如何使用 `SIGALRM` 信号定时。不过，我们需要先给出一种简单的定时器实现——基于升序链表的定时器，并把它应用到处理非活动连接这个实例中。这样，我们才能观察到 `SIGALRM` 信号处理函数是如何处理定时器并执行定时任务的。此外，我们介绍这种定时器也是为了和后面要讨论的高效定时器——时间轮和时间堆做对比。

11.2.1 基于升序链表的定时器

定时器通常至少要包含两个成员：一个超时时间（相对时间或者绝对时间）和一个任务回调函数。有的时候还可能包含回调函数被执行时需要传入的参数，以及是否重启定时器等信息。如果使用链表作为容器来串联所有的定时器，则每个定时器还要包含指向下一个定时器的指针成员。进一步，如果链表是双向的，则每个定时器还需要包含指向下一个定时器的指针成员。

代码清单 11-2 实现了一个简单的升序定时器链表。升序定时器链表将其中的定时器按照超时时间做升序排序。

代码清单 11-2 升序定时器链表

```
#ifndef LST_TIMER
#define LST_TIMER

#include <time.h>
#define BUFFER_SIZE 64
class util_timer; /* 前向声明 */

/* 用户数据结构：客户端 socket 地址、socket 文件描述符、读缓存和定时器 */
struct client_data
{
    sockaddr_in address;
    int sockfd;
    char buf[ BUFFER_SIZE ];
    util_timer* timer;
};

/* 定时器类 */
class util_timer
{
public:
    util_timer() : prev( NULL ), next( NULL ){}

public:
    time_t expire; /* 任务的超时时间，这里使用绝对时间 */
    void (*cb_func)( client_data* ); /* 任务回调函数 */
    /* 回调函数处理的客户数据，由定时器的执行者传递给回调函数 */
    client_data* user_data;
    util_timer* prev; /* 指向前一个定时器 */
    util_timer* next; /* 指向下一个定时器 */
};

/* 定时器链表。它是一个升序、双向链表，且带有头结点和尾节点 */
class sort_timer_lst
{
public:
    sort_timer_lst() : head( NULL ), tail( NULL ) {}
    /* 链表被销毁时，删除其中所有的定时器 */
    ~sort_timer_lst()
    {
        util_timer* tmp = head;
        while( tmp )
        {
            head = tmp->next;
            delete tmp;
            tmp = head;
        }
    }
}
```

```

/* 将目标定时器 timer 添加到链表中 */
void add_timer( util_timer* timer )
{
    if( !timer )
    {
        return;
    }
    if( !head )
    {
        head = tail = timer;
        return;
    }
    /* 如果目标定时器的超时时间小于当前链表中所有定时器的超时时间，则把该定时器插入链表头部。
       作为链表新的头节点。否则就需要调用重载函数 add_timer( util_timer* timer, util_timer* lst_head ) ，
       把它插入链表中合适的位置，以保证链表的升序特性 */
    if( timer->expire < head->expire )
    {
        timer->next = head;
        head->prev = timer;
        head = timer;
        return;
    }
    add_timer( timer, head );
}

/* 当某个定时任务发生变化时，调整对应的定时器在链表中的位置。这个函数只考虑被调整的定时器
   的超时时间延长的情况，即该定时器需要往链表的尾部移动 */
void adjust_timer( util_timer* timer )
{
    if( !timer )
    {
        return;
    }
    util_timer* tmp = timer->next;
    /* 如果被调整的目标定时器处在链表尾部，或者该定时器新的超时值仍然小于其下一个定时器的
       超时值，则不用调整 */
    if( !tmp || ( timer->expire < tmp->expire ) )
    {
        return;
    }
    /* 如果目标定时器是链表的头节点，则将该定时器从链表中取出并重新插入链表 */
    if( timer == head )
    {
        head = head->next;
        head->prev = NULL;
        timer->next = NULL;
        add_timer( timer, head );
    }
    /* 如果目标定时器不是链表的头节点，则将该定时器从链表中取出，然后插入其原来所在位置之
       后的部分链表中 */
    else
    {
        timer->prev->next = timer->next;
        timer->next->prev = timer->prev;
    }
}

```

```

        add_timer( timer, timer->next );
    }
}

/* 将目标定时器 timer 从链表中删除 */
void del_timer( util_timer* timer )
{
    if( !timer )
    {
        return;
    }

    /* 下面这个条件成立表示链表中只有一个定时器，即目标定时器 */
    if( ( timer == head ) && ( timer == tail ) )
    {
        delete timer;
        head = NULL;
        tail = NULL;
        return;
    }

    /* 如果链表中至少有两个定时器，且目标定时器是链表的头结点，则将链表的头结点重置为原头
       节点的下一个节点，然后删除目标定时器 */
    if( timer == head )
    {
        head = head->next;
        head->prev = NULL;
        delete timer;
        return;
    }

    /* 如果链表中至少有两个定时器，且目标定时器是链表的尾结点，则将链表的尾结点重置为原尾
       节点的前一个节点，然后删除目标定时器 */
    if( timer == tail )
    {
        tail = tail->prev;
        tail->next = NULL;
        delete timer;
        return;
    }

    /* 如果目标定时器位于链表的中间，则把它前后的定时器串联起来，然后删除目标定时器 */
    timer->prev->next = timer->next;
    timer->next->prev = timer->prev;
    delete timer;
}

/* SIGALRM 信号每次被触发就在其信号处理函数（如果使用统一事件源，则是主函数）中执行一次
   tick 函数，以处理链表上到期的任务 */
void tick()
{
    if( !head )
    {
        return;
    }

    printf( "timer tick\n" );
    time_t cur = time( NULL ); /* 获得系统当前的时间 */
    util_timer* tmp = head;
    /* 从头结点开始依次处理每个定时器，直到遇到一个尚未到期的定时器，这就是定时器的核心逻辑 */
}

```

```

        while( tmp )
        {
            /* 因为每个定时器都使用绝对时间作为超时值，所以我们可以把定时器的超时值和系统当前时间，比较以判断定时器是否到期 */
            if( cur < tmp->expire )
            {
                break;
            }
            /* 调用定时器的回调函数，以执行定时任务 */
            tmp->cb_func( tmp->user_data );
            /* 执行完定时器中的定时任务之后，就将它从链表中删除，并重置链表头结点 */
            head = tmp->next;
            if( head )
            {
                head->prev = NULL;
            }
            delete tmp;
            tmp = head;
        }
    }

private:
    /* 一个重载的辅助函数，它被公有的 add_timer 函数和 adjust_timer 函数调用。该函数表示将目标定时器 timer 添加到节点 lst_head 之后的部分链表中 */
    void add_timer( util_timer* timer, util_timer* lst_head )
    {
        util_timer* prev = lst_head;
        util_timer* tmp = prev->next;
        /* 遍历 lst_head 节点之后的部分链表，直到找到一个超时时间大于目标定时器的超时时间的节点，并将目标定时器插入该节点之前 */
        while( tmp )
        {
            if( timer->expire < tmp->expire )
            {
                prev->next = timer;
                timer->next = tmp;
                tmp->prev = timer;
                timer->prev = prev;
                break;
            }
            prev = tmp;
            tmp = tmp->next;
        }
        /* 如果遍历完 lst_head 节点之后的部分链表，仍未找到超时时间大于目标定时器的超时时间的节点，则将目标定时器插入链表尾部，并把它设置为链表新的尾节点 */
        if( !tmp )
        {
            prev->next = timer;
            timer->prev = prev;
            timer->next = NULL;
            tail = timer;
        }
    }
}

```

```

    }

private:
    util_timer* head;
    util_timer* tail;
};

#endif

```

为了便于阅读，我们将实现包含在头文件中。sort_timer_lst是一个升序链表。其核心函数 tick 相当于一个心搏函数，它每隔一段固定的时间就执行一次，以检测并处理到期的任务。判断定时任务到期的依据是定时器的 expire 值小于当前的系统时间。从执行效率来看，添加定时器的时间复杂度是 O(n)，删除定时器的时间复杂度是 O(1)，执行定时任务的时间复杂度是 O(1)。

11.2.2 处理非活动连接

现在我们考虑上述升序定时器链表的实际应用——处理非活动连接。服务器程序通常要定期处理非活动连接：给客户端发一个重连请求，或者关闭该连接，或者其他。Linux 在内核中提供了对连接是否处于活动状态的定期检查机制，我们可以通过 socket 选项 KEEPALIVE 来激活它。不过使用这种方式将使得应用程序对连接的管理变得复杂。因此，我们可以考虑在应用层实现类似于 KEEPALIVE 的机制，以管理所有长时间处于非活动状态的连接。比如，代码清单 11-3 利用 alarm 函数周期性地触发 SIGALRM 信号，该信号的信号处理函数利用管道通知主循环执行定时器链表上的定时任务——关闭非活动的连接。

代码清单 11-3 关闭非活动连接

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <pthread.h>
#include "lst_timer.h"

#define FD_LIMIT 65535
#define MAX_EVENT_NUMBER 1024
#define TIMESLOT 5

```

```

static int pipefd[2];
/* 利用代码清单 11-2 中的升序链表来管理定时器 */
static sort_timer_lst timer_lst;
static int epollfd = 0;

int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

void addfd( int epollfd, int fd )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET;
    epoll_ctl( epollfd, EPOLL_CTL_ADD, fd, &event );
    setnonblocking( fd );
}

void sig_handler( int sig )
{
    int save_errno = errno;
    int msg = sig;
    send( pipefd[1], ( char* )&msg, 1, 0 );
    errno = save_errno;
}

void addsig( int sig )
{
    struct sigaction sa;
    memset( &sa, '\0', sizeof( sa ) );
    sa.sa_handler = sig_handler;
    sa.sa_flags |= SA_RESTART;
    sigfillset( &sa.sa_mask );
    assert( sigaction( sig, &sa, NULL ) != -1 );
}

void timer_handler()
{
    /* 定时处理任务，实际上就是调用 tick 函数 */
    timer_lst.tick();
    /* 因为一次 alarm 调用只会引起一次 SIGALRM 信号，所以我们要重新定时，以不断触发 SIGALRM
信号 */
    alarm( TIMESLOT );
}

/* 定时器回调函数，它删除非活动连接 socket 上的注册事件，并关闭之 */
void cb_func( client_data* user_data )
{
    epoll_ctl( epollfd, EPOLL_CTL_DEL, user_data->sockfd, 0 );
}

```

```

    assert( user_data );
    close( user_data->sockfd );
    printf( "close fd %d\n", user_data->sockfd );
}

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    int listenfd = socket( PF_INET, SOCK_STREAM, 0 );
    assert( listenfd >= 0 );

    ret = bind( listenfd, ( struct sockaddr* )&address, sizeof( address ) );
    assert( ret != -1 );

    ret = listen( listenfd, 5 );
    assert( ret != -1 );

    epoll_event events[ MAX_EVENT_NUMBER ];
    int epollfd = epoll_create( 5 );
    assert( epollfd != -1 );
    addfd( epollfd, listenfd );

    ret = socketpair( PF_UNIX, SOCK_STREAM, 0, pipefd );
    assert( ret != -1 );
    setnonblocking( pipefd[1] );
    addfd( epollfd, pipefd[0] );

    /* 设置信号处理函数 */
    addsig( SIGALRM );
    addsig( SIGTERM );
    bool stop_server = false;

    client_data* users = new client_data[FD_LIMIT];
    bool timeout = false;
    alarm( TIMESLOT ); /* 定时 */

    while( !stop_server )
    {
        int number = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, -1 );

```

```

if ( ( number < 0 ) && ( errno != EINTR ) )
{
    printf( "epoll failure\n" );
    break;
}

for ( int i = 0; i < number; i++ )
{
    int sockfd = events[i].data.fd;
    /* 处理新到的客户连接 */
    if( sockfd == listenfd )
    {
        struct sockaddr_in client_address;
        socklen_t client_addrlength = sizeof( client_address );
        int connfd = accept( listenfd, ( struct sockaddr* )&client_address,
                            &client_addrlength );
        addfd( epollfd, connfd );
        users[connfd].address = client_address;
        users[connfd].sockfd = connfd;
        /* 创建定时器，设置其回调函数与超时时间，然后绑定定时器与用户数据，最后将定时器添加到链表 timer_lst 中 */
        util_timer* timer = new util_timer;
        timer->user_data = &users[connfd];
        timer->cb_func = cb_func;
        time_t cur = time( NULL );
        timer->expire = cur + 3 * TIMESLOT;
        users[connfd].timer = timer;
        timer_lst.add_timer( timer );
    }
    /* 处理信号 */
    else if( ( sockfd == pipefd[0] ) && ( events[i].events & EPOLLIN ) )
    {
        int sig;
        char signals[1024];
        ret = recv( pipefd[0], signals, sizeof( signals ), 0 );
        if( ret == -1 )
        {
            // handle the error
            continue;
        }
        else if( ret == 0 )
        {
            continue;
        }
        else
        {
            for( int i = 0; i < ret; ++i )
            {
                switch( signals[i] )
                {
                    case SIGALRM:
                    {
                        ...
                    }
                }
            }
        }
    }
}

```

```

        /* 用 timeout 变量标记有定时任务需要处理，但不立即处理定时任务。这是因为定时任务的优先级不是很高，我们优先处理其他更重要的任务 */
        timeout = true;
        break;
    }
    case SIGTERM:
    {
        stop_server = true;
    }
}
}

/* 处理客户连接上接收到的数据 */
else if( events[i].events & EPOLLIN )
{
    memset( users[sockfd].buf, '\0', BUFFER_SIZE );
    ret = recv( sockfd, users[sockfd].buf, BUFFER_SIZE-1, 0 );
    printf( "get %d bytes of client data %s from %d\n", ret,
            users[sockfd].buf, sockfd );

    util_timer* timer = users[sockfd].timer;
    if( ret < 0 )
    {
        /* 如果发生读错误，则关闭连接，并移除其对应的定时器 */
        if( errno != EAGAIN )
        {
            cb_func( &users[sockfd] );
            if( timer )
            {
                timer_lst.del_timer( timer );
            }
        }
    }
    else if( ret == 0 )
    {
        /* 如果对方已经关闭连接，则我们也关闭连接，并移除对应的定时器 */
        cb_func( &users[sockfd] );
        if( timer )
        {
            timer_lst.del_timer( timer );
        }
    }
    else
    {
        /* 如果某个客户连接上有数据可读，则我们要调整该连接对应的定时器，以
延迟该连接被关闭的时间 */
        if( timer )
        {
            time_t cur = time( NULL );
            timer->expire = cur + 3 * TIMESLOT;
            printf( "adjust timer once\n" );
            timer_lst.adjust_timer( timer );
        }
    }
}
}

```

```

        }
    }
}
else
{
    // others
}
}

/* 最后处理定时事件，因为 I/O 事件有更高的优先级。当然，这样做将导致定时任务不能精确地按照预期的时间执行 */
if( timeout )
{
    timer_handler();
    timeout = false;
}
}

close( listenfd );
close( pipefd[1] );
close( pipefd[0] );
delete [] users;
return 0;
}

```

11.3 I/O 复用系统调用的超时参数

Linux 下的 3 组 I/O 复用系统调用都带有超时参数，因此它们不仅能统一处理信号和 I/O 事件，也能统一处理定时事件。但是由于 I/O 复用系统调用可能在超时时间到期之前就返回（有 I/O 事件发生），所以如果我们要利用它们来定时，就需要不断更新定时参数以反映剩余的时间，如代码清单 11-4 所示。

代码清单 11-4 利用 I/O 复用系统调用定时

```

#define TIMEOUT 5000

int timeout = TIMEOUT;
time_t start = time( NULL );
time_t end = time( NULL );
while( 1 )
{
    printf( "the timeout is now %d mil-seconds\n", timeout );
    start = time( NULL );
    int number = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, timeout );
    if( ( number < 0 ) && ( errno != EINTR ) )
    {
        printf( "epoll failure\n" );
        break;
    }
}

```

```

/* 如果 epoll_wait 成功返回 0，则说明超时时间到，此时便可处理定时任务，并重置定时时间 */
if( number == 0 )
{
    timeout = TIMEOUT;
    continue;
}

end = time( NULL );
/* 如果 epoll_wait 的返回值大于 0，则本次 epoll_wait 调用持续的时间是 ( end - start ) *
1000 ms，我们需要将定时时间 timeout 减去这段时间，以获得下次 epoll_wait 调用的超时参数 */
timeout -= ( end - start ) * 1000;
/* 重新计算之后的 timeout 值有可能等于 0，说明本次 epoll_wait 调用返回时，不仅有文件描述符就
绪，而且其超时时间也刚好到达，此时我们也要处理定时任务，并重置定时时间 */
if( timeout <= 0 )
{
    timeout = TIMEOUT;
}

// handle connections
}

```

11.4 高性能定时器

11.4.1 时间轮

前文提到，基于排序链表的定时器存在一个问题：添加定时器的效率偏低。下面我们要讨论的时间轮解决了这个问题。一种简单的时间轮如图 11-1 所示。

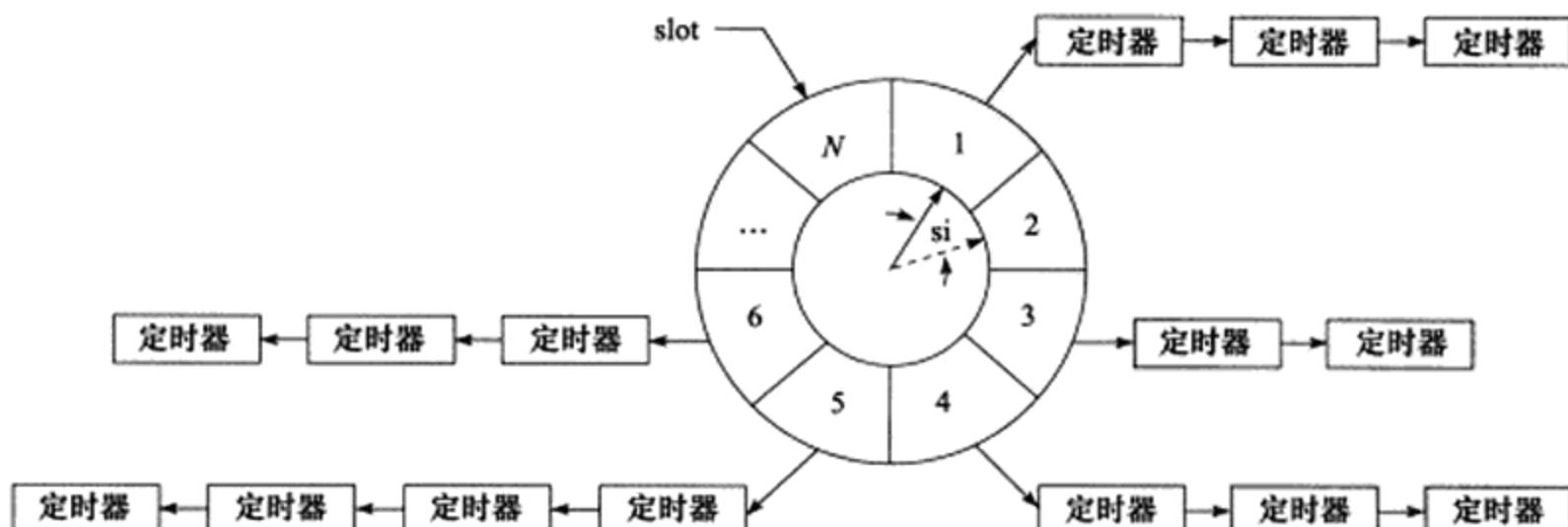


图 11-1 简单的时间轮

图 11-1 所示的时间轮内，(实线)指针指向轮子上的一个槽(slot)。它以恒定的速度顺时针转动，每转动一步就指向下一个槽(虚线指针指向的槽)，每次转动称为一个滴答(tick)。一个滴答的时间称为时间轮的槽间隔 si(slot interval)，它实际上就是心搏时间。该时间轮共有 N 个槽，因此它运转一周的时间是 N*si。每个槽指向一条定时器链表，每条链表

上的定时器具有相同的特征：它们的定时时间相差 $N*si$ 的整数倍。时间轮正是利用这个关系将定时器散列到不同的链表中。假如现在指针指向槽 cs，我们要添加一个定时时间为 ti 的定时器，则该定时器将被插入槽 ts (timer slot) 对应的链表中：

$$ts = (cs + (ti/si)) \% N \quad (11-1)$$

基于排序链表的定时器使用唯一的一条链表来管理所有定时器，所以插入操作的效率随着定时器数目的增多而降低。而时间轮使用哈希表的思想，将定时器散列到不同的链表上。这样每条链表上的定时器数目都将明显少于原来的排序链表上的定时器数目，插入操作的效率基本不受定时器数目的影响。

很显然，对时间轮而言，要提高定时精度，就要使 si 值足够小；要提高执行效率，则要求 N 值足够大。

图 11-1 描述的是一种简单的时间轮，因为它只有一个轮子。而复杂的时间轮可能有多个轮子，不同的轮子拥有不同的粒度。相邻的两个轮子，精度高的转一圈，精度低的仅往前移动一槽，就像水表一样。下面将按照图 11-1 来编写一个较为简单的时间轮实现代码，如代码清单 11-5 所示。

代码清单 11-5 时间轮

```

#ifndef TIME_WHEEL_TIMER
#define TIME_WHEEL_TIMER

#include <time.h>
#include <netinet/in.h>
#include <stdio.h>

#define BUFFER_SIZE 64
class tw_timer;
/* 绑定 socket 和定时器 */
struct client_data
{
    sockaddr_in address;
    int sockfd;
    char buf[ BUFFER_SIZE ];
    tw_timer* timer;
};

/* 定时器类 */
class tw_timer
{
public:
    tw_timer( int rot, int ts )
        : next( NULL ), prev( NULL ), rotation( rot ), time_slot( ts )()

public:
    int rotation; /* 记录定时器在时间轮转多少圈后生效 */
    int time_slot; /* 记录定时器属于时间轮上哪个槽（对应的链表，下同） */
    void (*cb_func)( client_data* ); /* 定时器回调函数 */
    client_data* user_data; /* 客户数据 */
};

```

```

tw_timer* next; /* 指向下一个定时器 */
tw_timer* prev; /* 指向前一个定时器 */
};

class time_wheel
{
public:
    time_wheel() : cur_slot( 0 )
    {
        for( int i = 0; i < N; ++i )
        {
            slots[i] = NULL; /* 初始化每个槽的头结点 */
        }
    }
    ~time_wheel()
    {
        /* 遍历每个槽，并销毁其中的定时器 */
        for( int i = 0; i < N; ++i )
        {
            tw_timer* tmp = slots[i];
            while( tmp )
            {
                slots[i] = tmp->next;
                delete tmp;
                tmp = slots[i];
            }
        }
    }
    /* 根据定时值 timeout 创建一个定时器，并把它插入合适的槽中 */
    tw_timer* add_timer( int timeout )
    {
        if( timeout < 0 )
        {
            return NULL;
        }
        int ticks = 0;
        /* 下面根据待插入定时器的超时值计算它将在时间轮转动多少个滴答后被触发，并将该滴答数存储于变量 ticks 中。如果待插入定时器的超时值小于时间轮的槽间隔 SI，则将 ticks 向上折合为 1，否则就将 ticks 向下折合为 timeout/SI */
        if( timeout < SI )
        {
            ticks = 1;
        }
        else
        {
            ticks = timeout / SI;
        }
        /* 计算待插入的定时器在时间轮转动多少圈后被触发 */
        int rotation = ticks / N;
        /* 计算待插入的定时器应该被插入哪个槽中 */
        int ts = ( cur_slot + ( ticks % N ) ) % N;
        /* 创建新的定时器，它在时间轮转动 rotation 圈之后被触发，且位于第 ts 个槽上 */
        tw_timer* timer = new tw_timer( rotation, ts );
    }
};

```

```

/* 如果第 ts 个槽中尚无任何定时器，则把新建的定时器插入其中，并将该定时器设置为该槽的
头结点 */
if( !slots[ts] )
{
    printf( "add timer, rotation is %d, ts is %d, cur_slot is %d\n",
            rotation, ts, cur_slot );
    slots[ts] = timer;
}
/* 否则，将定时器插入第 ts 个槽中 */
else
{
    timer->next = slots[ts];
    slots[ts]->prev = timer;
    slots[ts] = timer;
}
return timer;
}

/* 删除目标定时器 timer */
void del_timer( tw_timer* timer )
{
    if( !timer )
    {
        return;
    }
    int ts = timer->time_slot;
    /* slots[ts] 是目标定时器所在槽的头结点。如果目标定时器就是该头结点，则需要重置第 ts
    个槽的头结点 */
    if( timer == slots[ts] )
    {
        slots[ts] = slots[ts]->next;
        if( slots[ts] )
        {
            slots[ts]->prev = NULL;
        }
        delete timer;
    }
    else
    {
        timer->prev->next = timer->next;
        if( timer->next )
        {
            timer->next->prev = timer->prev;
        }
        delete timer;
    }
}
/* SI 时间到后，调用该函数，时间轮向前滚动一个槽的间隔 */
void tick()
{
    tw_timer* tmp = slots[cur_slot]; /* 取得时间轮上当前槽的头结点 */
    printf( "current slot is %d\n", cur_slot );
    while( tmp )

```

```

    {
        printf( "tick the timer once\n" );
        /* 如果定时器的 rotation 值大于 0，则它在这一轮不起作用 */
        if( tmp->rotation > 0 )
        {
            tmp->rotation--;
            tmp = tmp->next;
        }
        /* 否则，说明定时器已经到期，于是执行定时任务，然后删除该定时器 */
        else
        {
            tmp->cb_func( tmp->user_data );
            if( tmp == slots[cur_slot] )
            {
                printf( "delete header in cur_slot\n" );
                slots[cur_slot] = tmp->next;
                delete tmp;
                if( slots[cur_slot] )
                {
                    slots[cur_slot]->prev = NULL;
                }
                tmp = slots[cur_slot];
            }
            else
            {
                tmp->prev->next = tmp->next;
                if( tmp->next )
                {
                    tmp->next->prev = tmp->prev;
                }
                tw_timer* tmp2 = tmp->next;
                delete tmp;
                tmp = tmp2;
            }
        }
        cur_slot = ++cur_slot % N; /* 更新时间轮的当前槽，以反映时间轮的转动 */
    }

private:
/* 时间轮上槽的数目 */
static const int N = 60;
/* 每 1 s 时间轮转动一次，即槽间隔为 1 s */
static const int SI = 1;
/* 时间轮的槽，其中每个元素指向一个定时器链表，链表无序 */
tw_timer* slots[N];
int cur_slot; /* 时间轮的当前槽 */
};

#endif

```

可见，对时间轮而言，添加一个定时器的时间复杂度是 O (1)，删除一个定时器的时

间复杂度也是 $O(1)$ ，执行一个定时器的时间复杂度是 $O(n)$ 。但实际上执行一个定时器任务的效率要比 $O(n)$ 好得多，因为时间轮将所有的定时器散列到了不同的链表上。时间轮的槽越多，等价于散列表的入口（entry）越多，从而每条链表上的定时器数量越少。此外，我们的代码仅使用了一个时间轮。当使用多个轮子来实现时间轮时，执行一个定时器任务的时间复杂度将接近 $O(1)$ 。读者不妨把代码清单 11-3 稍做修改，用时间轮来代替排序链表，以查看时间轮的工作方式和效率。

11.4.2 时间堆

前面讨论的定时方案都是以固定的频率调用心搏函数 tick，并在其中依次检测到期的定时器，然后执行到期定时器上的回调函数。设计定时器的另外一种思路是：将所有定时器中超时时间最小的一个定时器的超时值作为心搏间隔。这样，一旦心搏函数 tick 被调用，超时时间最小的定时器必然到期，我们就可以在 tick 函数中处理该定时器。然后，再次从剩余的定时器中找出超时时间最小的一个，并将这段最短时间设置为下一次心搏间隔。如此反复，就实现了较为精确的定时。

最小堆很适合处理这种定时方案。最小堆是指每个节点的值都小于或等于其子节点的值的完全二叉树。图 11-2 给出了一个具有 6 个元素的最小堆。

树的基本操作是插入节点和删除节点。对最小堆而言，它们都很简单。为了将一个元素 X 插入最小堆，我们可以在树的下一个空闲位置创建一个空穴。如果 X 可以放在空穴中而不破坏堆序，则插入完成。否则就执行上虑操作，即交换空穴和它的父节点上的元素。不断执行上述过程，直到 X 可以被放入空穴，则插入操作完成。比如，我们要往图 11-2 所示的最小堆中插入值为 14 的元素，则可以按照图 11-3 所示的步骤来操作。

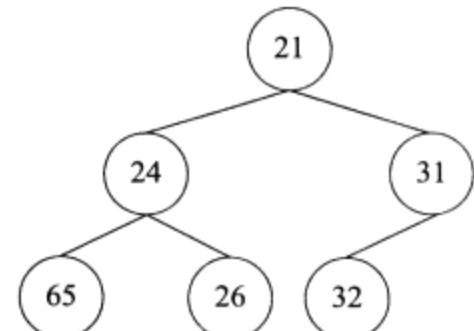


图 11-2 最小堆

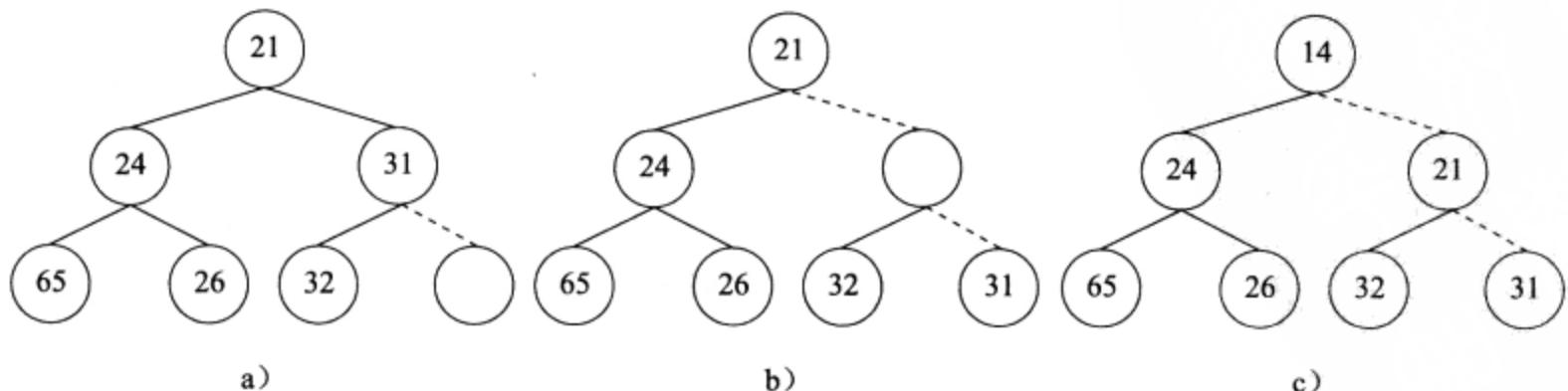


图 11-3 最小堆的插入操作

a) 创建空穴 b) 上虑一次 c) 上虑二次

最小堆的删除操作指的是删除其根节点上的元素，并且不破坏堆序性质。执行删除操作时，我们需要先在根节点处创建一个空穴。由于堆现在少了一个元素，因此我们可以把堆的最后一个元素 X 移动到该堆的某个地方。如果 X 可以被放入空穴，则删除操作完成。否则就

执行下虑操作，即交换空穴和它的两个儿子节点中的较小者。不断进行上述过程，直到 X 可以被放入空穴，则删除操作完成。比如，我们要对图 11-2 所示的最小堆执行删除操作，则可以按照图 11-4 所示的步骤来执行。

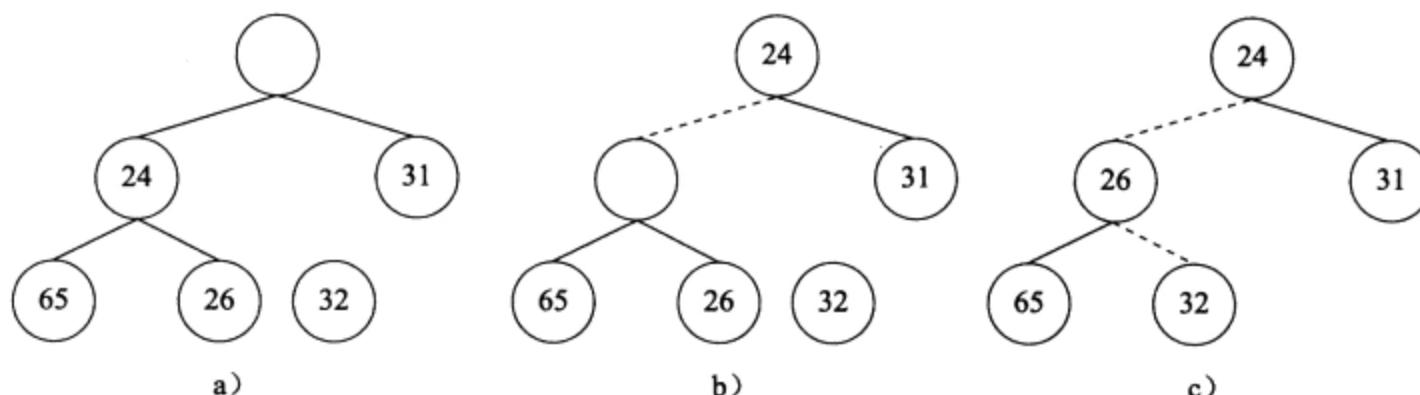


图 11-4 最小堆的删除操作

a) 在根节点处创建空穴 b) 下虑一次 c) 下虑二次

由于最小堆是一种完全二叉树，所以我们可以用数组来组织其中的元素。比如，图 11-2 所示的最小堆可以用图 11-5 所示的数组来表示。对于数组中的任意一个位置 i 上的元素，其左儿子节点在位置 $2i+1$ 上，其右儿子节点在位置 $2i+2$ 上，其父节点则在位置 $\lceil (i-1)/2 \rceil$ ($i > 0$) 上。与用链表来表示堆相比，用数组表示堆不仅节省空间，而且更容易实现堆的插入、删除等操作^[5]。

21	24	31	65	26	32			
0	1	2	3	4	5	6	7	8

图 11-5 最小堆的数组表示

假设我们已经有一个包含 N 个元素的数组，现在要把它初始化为一个最小堆。那么最简单的方法是：初始化一个空堆，然后将数组中的每个元素插入该堆中。不过这样做的效率偏低。实际上，我们只需要对数组中的第 $\lceil (N-1)/2 \rceil \sim 0$ 个元素执行下虑操作，即可确保该数组构成一个最小堆。这是因为对包含 N 个元素的完全二叉树而言，它具有 $\lceil (N-1)/2 \rceil$ 个非叶子节点，这些非叶子节点正是该完全二叉树的第 $0 \sim \lceil (N-1)/2 \rceil$ 个节点。我们只要确保这些非叶子节点构成的子树都具有堆序性质，整个树就具有堆序性质。

我们称用最小堆实现的定时器为时间堆。代码清单 11-6 给出了一种时间堆的实现，其中，最小堆使用数组来表示。

代码清单 11-6 时间堆

```
#ifndef MIN_HEAP
#define MIN_HEAP

#include <iostream>
#include <netinet/in.h>
#include <time.h>
using std::exception;

#define BUFFER_SIZE 64
```

```

class heap_timer; /* 前向声明 */
/* 绑定 socket 和定时器 */
struct client_data
{
    sockaddr_in address;
    int sockfd;
    char buf[ BUFFER_SIZE ];
    heap_timer* timer;
};

/* 定时器类 */
class heap_timer
{
public:
    heap_timer( int delay )
    {
        expire = time( NULL ) + delay;
    }

public:
    time_t expire; /* 定时器生效的绝对时间 */
    void (*cb_func)( client_data* ); /* 定时器的回调函数 */
    client_data* user_data; /* 用户数据 */
};

/* 时间堆类 */
class time_heap
{
public:
    /* 构造函数之一，初始化一个大小为 cap 的空堆 */
    time_heap( int cap ) throw ( std::exception ) : capacity( cap ), cur_size( 0 )
    {
        array = new heap_timer* [capacity]; /* 创建堆数组 */
        if ( !array )
        {
            throw std::exception();
        }
        for( int i = 0; i < capacity; ++i )
        {
            array[i] = NULL;
        }
    }
    /* 构造函数之二，用已有数组来初始化堆 */
    time_heap( heap_timer** init_array, int size, int capacity ) throw
        ( std::exception ): cur_size( size ), capacity( capacity )
    {
        if ( capacity < size )
        {
            throw std::exception();
        }
        array = new heap_timer* [capacity]; /* 创建堆数组 */
    }
};

```

```

    if ( ! array )
    {
        throw std::exception();
    }
    for( int i = 0; i < capacity; ++i )
    {
        array[i] = NULL;
    }
    if ( size != 0 )
    {
        /* 初始化堆数组 */
        for ( int i = 0; i < size; ++i )
        {
            array[ i ] = init_array[ i ];
        }
        for ( int i = (cur_size-1)/2; i >=0; --i )
        {
            /* 对数组中的第 [(cur_size-1)/2]~0 个元素执行下虑操作 */
            percolate_down( i );
        }
    }
}
/* 销毁时间堆 */
~time_heap()
{
    for ( int i = 0; i < cur_size; ++i )
    {
        delete array[i];
    }
    delete [] array;
}

public:
/* 添加目标定时器 timer */
void add_timer( heap_timer* timer ) throw ( std::exception )
{
    if( !timer )
    {
        return;
    }
    if( cur_size >= capacity ) /* 如果当前堆数组容量不够，则将其扩大 1 倍 */
    {
        resize();
    }
    /* 新插入了一个元素，当前堆大小加 1，hole 是新建空穴的位置 */
    int hole = cur_size++;
    int parent = 0;
    /* 对从空穴到根节点的路径上的所有节点执行上虑操作 */
    for( ; hole > 0; hole=parent )
    {
        parent = (hole-1)/2;
        if ( array[parent]->expire <= timer->expire )
        {
            break;
        }
    }
}

```

```

        array[hole] = array[parent];
    }
    array[hole] = timer;
}
/* 删除目标定时器 timer */
void del_timer( heap_timer* timer )
{
    if( !timer )
    {
        return;
    }
    /* 仅仅将目标定时器的回调函数设置为空，即所谓的延迟销毁。这将节省真正删除该定时器造成的开销，但这样做容易使堆数组膨胀 */
    timer->cb_func = NULL;
}
/* 获得堆顶部的定时器 */
heap_timer* top() const
{
    if( empty() )
    {
        return NULL;
    }
    return array[0];
}
/* 删除堆顶部的定时器 */
void pop_timer()
{
    if( empty() )
    {
        return;
    }
    if( array[0] )
    {
        delete array[0];
        /* 将原来的堆顶元素替换为堆数组中最后一个元素 */
        array[0] = array[--cur_size];
        percolate_down( 0 ); /* 对新的堆顶元素执行下滤操作 */
    }
}
/* 心搏函数 */
void tick()
{
    heap_timer* tmp = array[0];
    time_t cur = time( NULL ); /* 循环处理堆中到期的定时器 */
    while( !empty() )
    {
        if( !tmp )
        {
            break;
        }
        /* 如果堆顶定时器没到期，则退出循环 */
        if( tmp->expire > cur )
        {

```

```

        break;
    }
    /* 否则就执行堆顶定时器中的任务 */
    if( array[0]->cb_func )
    {
        array[0]->cb_func( array[0]->user_data );
    }
    /* 将堆顶元素删除，同时生成新的堆顶定时器 (array[0]) */
    pop_timer();
    tmp = array[0];
}
bool empty() const { return cur_size == 0; }

private:
/* 最小堆的下虑操作，它确保堆数组中以第 hole 个节点作为根的子树拥有最小堆性质 */
void percolate_down( int hole )
{
    heap_timer* temp = array[hole];
    int child = 0;
    for ( ; ((hole*2+1) <= (cur_size-1)); hole=child )
    {
        child = hole*2+1;
        if ( (child < (cur_size-1)) && (array[child+1]->expire <
            array[child]->expire ) )
        {
            ++child;
        }
        if ( array[child]->expire < temp->expire )
        {
            array[hole] = array[child];
        }
        else
        {
            break;
        }
    }
    array[hole] = temp;
}
/* 将堆数组容量扩大 1 倍 */
void resize() throw ( std::exception )
{
    heap_timer** temp = new heap_timer* [2*capacity];
    for( int i = 0; i < 2*capacity; ++i )
    {
        temp[i] = NULL;
    }
    if ( ! temp )
    {
        throw std::exception();
    }
    capacity = 2*capacity;
    for ( int i = 0; i < cur_size; ++i )
    {

```

```
        temp[i] = array[i];
    }
    delete [] array;
    array = temp;
}

private:
    heap_timer** array; /* 堆数组 */
    int capacity; /* 堆数组的容量 */
    int cur_size; /* 堆数组当前包含元素的个数 */
};

#endif
```

由代码清单 11-6 可见，对时间堆而言，添加一个定时器的时间复杂度是 $O(\lg n)$ ，删除一个定时器的时间复杂度是 $O(1)$ ，执行一个定时器的时间复杂度是 $O(1)$ 。因此，时间堆的效率是很高的。

第 12 章 高性能 I/O 框架库 Libevent

前面我们利用三章的篇幅较为细致地讨论了 Linux 服务器程序必须处理的三类事件：I/O 事件、信号和定时事件。在处理这三类事件时我们通常需要考虑如下三个问题：

- 统一事件源。很明显，统一处理这三类事件既能使代码简单易懂，又能避免一些潜在的逻辑错误。前面我们已经讨论了实现统一事件源的一般方法——利用 I/O 复用系统调用来管理所有事件。
- 可移植性。不同的操作系统具有不同的 I/O 复用方式，比如 Solaris 的 dev/poll 文件，FreeBSD 的 kqueue 机制，Linux 的 epoll 系列系统调用。
- 对并发编程的支持。在多进程和多线程环境下，我们需要考虑各执行实体如何协同处理客户连接、信号和定时器，以避免竞态条件。

所幸的是，开源社区提供了诸多优秀的 I/O 框架库。它们不仅解决了上述问题，让开发者可以将精力完全放在程序的逻辑上，而且稳定性、性能等各方面都相当出色。比如 ACE、ASIO 和 Libevent。本章将介绍其中相对轻量级的 Libevent 框架库。

12.1 I/O 框架库概述

I/O 框架库以库函数的形式，封装了较为底层的系统调用，给应用程序提供了一组更便于使用的接口。这些库函数往往比程序员自己实现的同样功能的函数更合理、更高效，且更健壮。因为它们经受住了真实网络环境下的高压测试，以及时间的考验。

各种 I/O 框架库的实现原理基本相似，要么以 Reactor 模式实现，要么以 Proactor 模式实现，要么同时以这两种模式实现。举例来说，基于 Reactor 模式的 I/O 框架库包含以下几个组件：句柄（Handle）、事件多路分发器（EventDemultiplexer）、事件处理器（EventHandler）和具体的事件处理器（ConcreteEventHandler）、Reactor。这些组件的关系如图 12-1 所示^[6]。

1. 句柄

I/O 框架库要处理的对象，即 I/O 事件、信号和定时事件，统一称为事件源。一个事件源通常和一个句柄绑定在一起。句柄的作用是，当内核检测到就绪事件时，它将通过句柄来通知应用程序这一事件。在 Linux 环境下，I/O 事件对应的句柄是文件描述符，信号事件对应的句柄就是信号值。

2. 事件多路分发器

事件的到来是随机的、异步的。我们无法预知程序何时收到一个客户连接请求，又亦或收到一个暂停信号。所以程序需要循环地等待并处理事件，这就是事件循环。在事件循环

中，等待事件一般使用 I/O 复用技术来实现。I/O 框架库一般将系统支持的各种 I/O 复用系统调用封装成统一的接口，称为事件多路分发器。事件多路分发器的 demultiplex 方法是等待事件的核心函数，其内部调用的是 select、poll、epoll_wait 等函数。

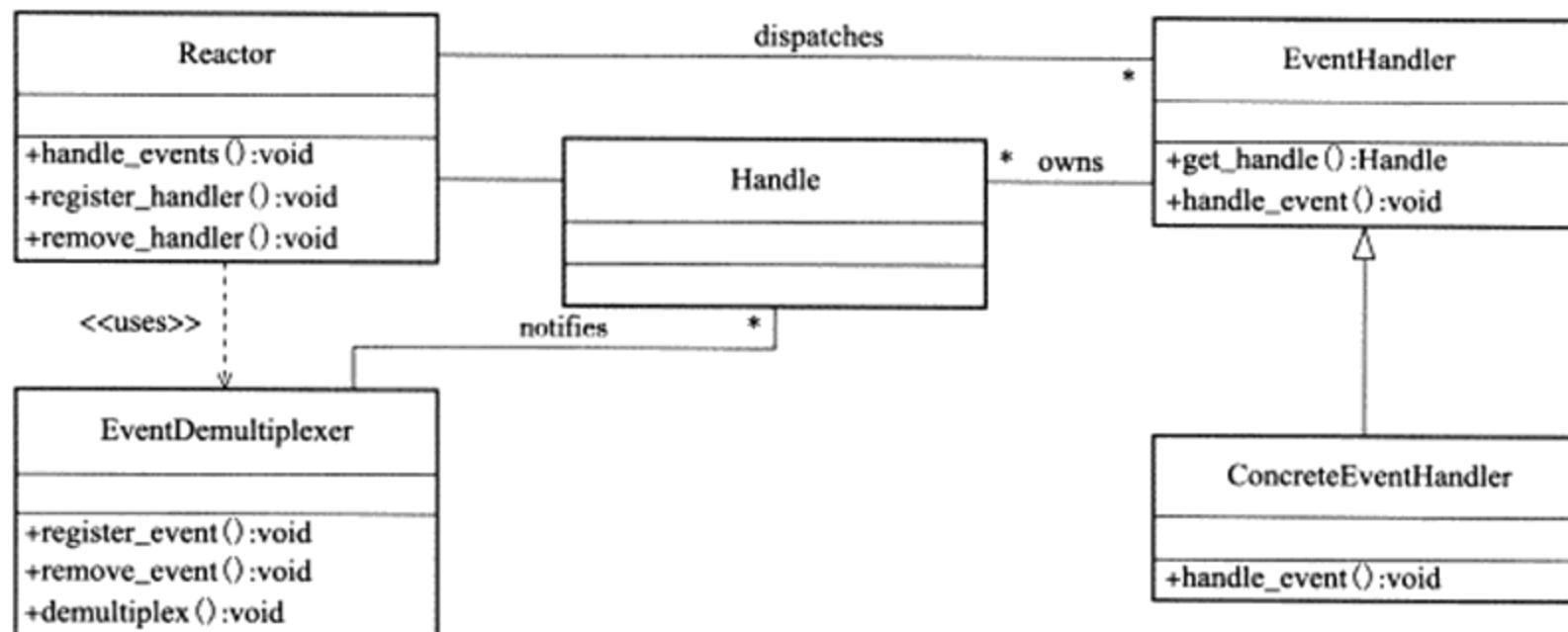


图 12-1 I/O 框架库组件

此外，事件多路分发器还需要实现 register_event 和 remove_event 方法，以供调用者往事件多路分发器中添加事件和从事件多路分发器中删除事件。

3. 事件处理器和具体事件处理器

事件处理器执行事件对应的业务逻辑。它通常包含一个或多个 handle_event 回调函数，这些回调函数在事件循环中被执行。I/O 框架库提供的事件处理器通常是一个接口，用户需要继承它来实现自己的事件处理器，即具体事件处理器。因此，事件处理器中的回调函数一般被声明为虚函数，以支持用户的扩展。

此外，事件处理器一般还提供一个 get_handle 方法，它返回与该事件处理器关联的句柄。那么，事件处理器和句柄有什么关系？当事件多路分发器检测到有事件发生时，它是通过句柄来通知应用程序的。因此，我们必须将事件处理器和句柄绑定，才能在事件发生时获取到正确的事件处理器。

4. Reactor

Reactor 是 I/O 框架库的核心。它提供的几个主要方法是：

- ❑ handle_events。该方法执行事件循环。它重复如下过程：等待事件，然后依次处理所有就绪事件对应的事件处理器。
- ❑ register_handler。该方法调用事件多路分发器的 register_event 方法来往事件多路分发器中注册一个事件。
- ❑ remove_handler。该方法调用事件多路分发器的 remove_event 方法来删除事件多路分发器中的一个事件。

图 12-2 总结了 I/O 框架库的工作时序。

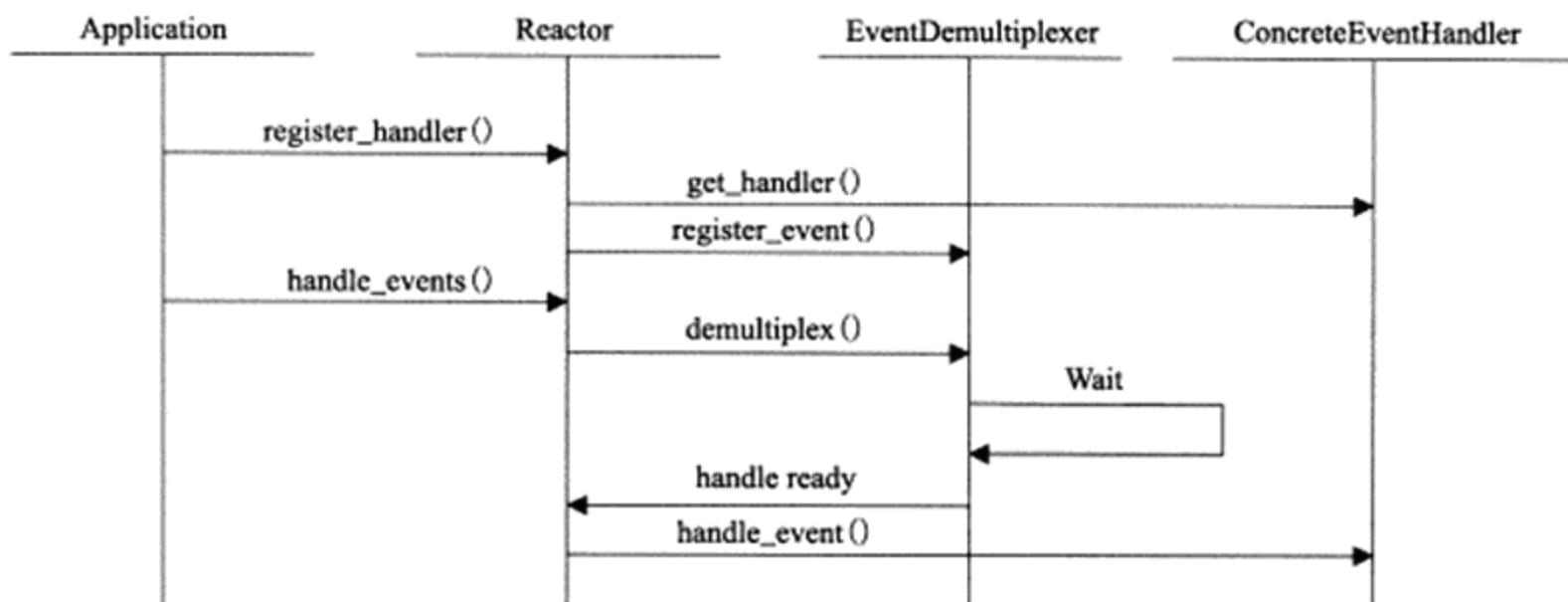


图 12-2 I/O 框架库的工作时序图

12.2 Libevent 源码分析

Libevent 是开源社区的一款高性能的 I/O 框架库，其学习者和使用者众多。使用 Libevent 的著名案例有：高性能的分布式内存对象缓存软件 memcached，Google 浏览器 Chromium 的 Linux 版本。作为一个 I/O 框架库，Libevent 具有如下特点：

- 跨平台支持。Libevent 支持 Linux、UNIX 和 Windows。
- 统一事件源。Libevent 对 I/O 事件、信号和定时事件提供统一的处理。
- 线程安全。Libevent 使用 libevent_pthreads 库来提供线程安全支持。
- 基于 Reactor 模式的实现。

这一节中我们将简单地研究一下 Libevent 源代码的主要部分。分析它除了可以更好地学习网络编程外，还有如下好处：

- 学习编写一个产品级的函数库要考虑哪些细节。
- 提高 C 语言功底。Libevent 源码中使用了大量的函数指针，用 C 语言实现了多态机制，并提供了一些基础数据结构的高效实现，比如双向链表、最小堆等。

Libevent 的官方网站是 <http://libevent.org/>，其中提供 Libevent 源代码的下载，以及学习 Libevent 框架库的第一手文档，并且源码和文档的更新也较为频繁。笔者写作此书时使用的 Libevent 版本是该网站于 2012 年 5 月 3 日发布的 2.0.19。

12.2.1 一个实例

分析一款软件的源代码，最简单有效的方式是从使用入手，这样才能从整体上把握该软件的逻辑结构。代码清单 12-1 是使用 Libevent 库实现的一个“Hello World”程序。

代码清单 12-1 Libevent 实例

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

```

void signal_cb( int fd, short event, void* argc )
{
    struct event_base* base = ( event_base* )argc;
    struct timeval delay = { 2, 0 };
    printf( "Caught an interrupt signal; exiting cleanly in two seconds...\n" );
    event_base_loopexit( base, &delay );
}
void timeout_cb( int fd, short event, void* argc )
{
    printf( "timeout\n" );
}

int main()
{
    struct event_base* base = event_init();

    struct event* signal_event = evsignal_new( base, SIGINT, signal_cb, base );
    event_add( signal_event, NULL );

    timeval tv = { 1, 0 };
    struct event* timeout_event = evtimer_new( base, timeout_cb, NULL );
    event_add( timeout_event, &tv );

    event_base_dispatch( base );

    event_free( timeout_event );
    event_free( signal_event );
    event_base_free( base );
}

```

代码清单 12-1 虽然简单，但却基本上描述了 Libevent 库的主要逻辑：

- 1) 调用 `event_init` 函数创建 `event_base` 对象。一个 `event_base` 相当于一个 Reactor 实例。
- 2) 创建具体的事件处理器，并设置它们所从属的 Reactor 实例。`evsignal_new` 和 `evtimer_new` 分别用于创建信号事件处理器和定时事件处理器，它们是定义在 `include/event2/event.h` 文件中的宏：

```

#define evsignal_new(b, x, cb, arg) \
    event_new((b), (x), EV_SIGNAL|EV_PERSIST, (cb), (arg))
#define evtimer_new(b, cb, arg)     event_new((b), -1, 0, (cb), (arg))

```

可见，它们的统一入口是 `event_new` 函数，即用于创建通用事件处理器（图 12-1 中的 `EventHandler`）的函数。其定义是：

```

struct event* event_new(struct event_base* base, evutil_socket_t fd, short events,
                       void (*cb)(evutil_socket_t, short, void* ), void* arg)

```

其中，`base` 参数指定新创建的事件处理器从属的 Reactor。`fd` 参数指定与该事件处理器关联的句柄。创建 I/O 事件处理器时，应该给 `fd` 参数传递文件描述符值；创建信号事件处理器时，应该给 `fd` 参数传递信号值，比如代码清单 12-1 中的 `SIGINT`；创建定时事件处理器时，

则应该给 fd 参数传递 -1。events 参数指定事件类型，其可选值都定义在 include/event2/event.h 文件中，如代码清单 12-2 所示。

代码清单 12-2 Libevent 支持的事件类型

```
#define EV_TIMEOUT      0x01 /* 定时事件 */
#define EV_READ          0x02 /* 可读事件 */
#define EV_WRITE         0x04 /* 可写事件 */
#define EV_SIGNAL        0x08 /* 信号事件 */
#define EV_PERSIST       0x10 /* 永久事件 */
/* 边沿触发事件，需要 I/O 复用系统调用支持，比如 epoll */
#define EV_ET            0x20
```

代码清单 12-2 中，EV_PERSIST 的作用是：事件被触发后，自动重新对这个 event 调用 event_add 函数（见后文）。

cb 参数指定目标事件对应的回调函数，相当于图 12-1 中事件处理器的 handle_event 方法。arg 参数则是 Reactor 传递给回调函数的参数。

event_new 函数成功时返回一个 event 类型的对象，也就是 Libevent 的事件处理器。Libevent 用单词“event”来描述事件处理器，而不是事件，会使读者觉得有些混乱，故而我们约定如下：

- 事件指的是一个句柄上绑定的事件，比如文件描述符 0 上的可读事件。
- 事件处理器，也就是 event 结构体类型的对象，除了包含事件必须具备的两个要素（句柄和事件类型）外，还有很多其他成员，比如回调函数。
- 事件由事件多路分发器管理，事件处理器则由事件队列管理。事件队列包括多种，比如 event_base 中的注册事件队列、活动事件队列和通用定时器队列，以及 evmap 中的 I/O 事件队列、信号事件队列。关于这些事件队列，我们将在后文依次讨论。
- 事件循环对一个被激活事件（就绪事件）的处理，指的是执行该事件对应的事件处理器中的回调函数。

3) 调用 event_add 函数，将事件处理器添加到注册事件队列中，并将该事件处理器对应的事件添加到事件多路分发器中。event_add 函数相当于 Reactor 中的 register_handler 方法。

4) 调用 event_base_dispatch 函数来执行事件循环。

5) 事件循环结束后，使用 *_free 系列函数来释放系统资源。

由此可见，代码清单 12-1 给我们提供了一条分析 Libevent 源代码的主线。不过在此之前，我们先简单介绍一下 Libevent 源代码的组织结构。

12.2.2 源代码组织结构

Libevent 源代码中的目录和文件按照功能可划分为如下部分：

- 头文件目录 include/event2。该目录是自 Libevent 主版本升级到 2.0 之后引入的，在 1.4 及更老的版本中并无此目录。该目录中的头文件是 Libevent 提供给应用程序使

用的，比如，`event.h` 头文件提供核心函数，`http.h` 头文件提供 HTTP 协议相关服务，`rpc.h` 头文件提供远程过程调用支持。

- 源码根目录下的头文件。这些头文件分为两类：一类是对 `include/event2` 目录下的部分头文件的包装，另外一类是供 Libevent 内部使用的辅助性头文件，它们的文件名都具有 `*-internal.h` 的形式。
- 通用数据结构目录 `compat/sys`。该目录下仅有一个文件——`queue.h`。它封装了跨平台的基础数据结构，包括单向链表、双向链表、队列、尾队列和循环队列。
- `sample` 目录。它提供一些示例程序。
- `test` 目录。它提供一些测试代码。
- `WIN32-Code` 目录。它提供 Windows 平台上的一些专用代码。
- `event.c` 文件。该文件实现 Libevent 的整体框架，主要是 `event` 和 `event_base` 两个结构体的相关操作。
- `devpoll.c`、`kqueue.c`、`evport.c`、`select.c`、`win32select.c`、`poll.c` 和 `epoll.c` 文件。它们分别封装了如下 I/O 复用机制：`/dev/poll`、`kqueue`、`event ports`、`POSIX select`、`Windows select`、`poll` 和 `epoll`。这些文件的主要内容相似，都是针对结构体 `eventop`（见后文）所定义的接口函数的具体实现。
- `minheap-internal.h` 文件。该文件实现了一个时间堆，以提供对定时事件的支持。
- `signal.c` 文件。它提供对信号的支持。其内容也是针对结构体 `eventop` 所定义的接口函数的具体实现。
- `evmap.c` 文件。它维护句柄（文件描述符或信号）与事件处理器的映射关系。
- `event_tagging.c` 文件。它提供往缓冲区中添加标记数据（比如一个整数），以及从缓冲区中读取标记数据的函数。
- `event_iocp.c` 文件。它提供对 Windows IOCP（Input/Output Completion Port，输入输出完成端口）的支持。
- `buffer*.c` 文件。它提供对网络 I/O 缓冲的控制，包括：输入输出数据过滤，传输速率限制，使用 SSL（Secure Sockets Layer）协议对应用数据进行保护，以及零拷贝文件传输等。
- `evthread*.c` 文件。它提供对多线程的支持。
- `listener.c` 文件。它封装了对监听 socket 的操作，包括监听连接和接受连接。
- `logs.c` 文件。它是 Libevent 的日志系统。
- `evutil.c`、`evutil_rand.c`、`strncpy.c` 和 `arc4random.c` 文件。它们提供一些基本操作，比如生成随机数、获取 socket 地址信息、读取文件、设置 socket 属性等。
- `evdns.c`、`http.c` 和 `evrpc.c` 文件。它们分别提供了对 DNS 协议、HTTP 协议和 RPC（Remote Procedure Call，远程过程调用）协议的支持。
- `epoll_sub.c` 文件。该文件未见使用。

在整个源码中，`event-internal.h`、`include/event2/event_struct.h`、`event.c` 和 `evmap.c` 等 4 个

文件最为重要。它们定义了 event 和 event_base 结构体，并实现了这两个结构体的相关操作。下面的讨论也主要是围绕这几个文件展开的。

12.2.3 event 结构体

前文提到，Libevent 中的事件处理器是 event 结构类型。event 结构体封装了句柄、事件类型、回调函数，以及其他必要的标志和数据。该结构体在 include/event2/event_struct.h 文件中定义：

```
struct event
{
    TAILQ_ENTRY(event) ev_active_next;
    TAILQ_ENTRY(event) ev_next;
    union {
        TAILQ_ENTRY(event) ev_next_with_common_timeout;
        int min_heap_idx;
    } ev_timeout_pos;
    evutil_socket_t ev_fd;

    struct event_base* ev_base;

    union {
        struct {
            TAILQ_ENTRY(event) ev_io_next;
            struct timeval ev_timeout;
        } ev_io;

        struct {
            TAILQ_ENTRY(event) ev_signal_next;
            short ev_ncalls;
            short *ev_pncalls;
        } ev_signal;
    } _ev;

    short ev_events;
    short ev_res;
    short ev_flags;
    ev_uint8_t ev_pri;
    ev_uint8_t ev_closure;
    struct timeval ev_timeout;

    void (*ev_callback)(evutil_socket_t, short, void *arg);
    void *ev_arg;
};


```

下面我们详细介绍 event 结构体中的每个成员：

- **ev_events**。它代表事件类型。其取值可以是代码清单 12-2 所示的标志的按位或（互斥的事件类型除外，比如读写事件和信号事件就不能同时被设置）。
- **ev_next**。所有已经注册的事件处理器（包括 I/O 事件处理器和信号事件处理器）通过

该成员串联成一个尾队列，我们称之为注册事件队列。宏 TAILQ_ENTRY 是尾队列中的节点类型，它定义在 compat/sys/queue.h 文件中：

```
#define TAILQ_ENTRY(type) \
struct { \
    struct type* tqe_next; /* 下一个元素 */ \
    struct type** tqe_prev; /* 前一个元素的地址 */ \
}
```

- ev_active_next。所有被激活的事件处理器通过该成员串联成一个尾队列，我们称之为活动事件队列。活动事件队列不止一个，不同优先级的事件处理器被激活后将被插入不同的活动事件队列中。在事件循环中，Reactor 将按优先级从高到低遍历所有活动事件队列，并依次处理其中的事件处理器。
- ev_timeout_pos。这是一个联合体，它仅用于定时事件处理器。为了讨论的方便，后面我们称定时事件处理器为定时器。老版本的 Libevent 中，定时器都是由时间堆来管理的。但开发者认为有时候使用简单的链表来管理定时器将具有更高的效率。因此，新版本的 Libevent 就引入了所谓“通用定时器”的概念。这些定时器不是存储在时间堆中，而是存储在尾队列中，我们称之为通用定时器队列。对于通用定时器而言，ev_timeout_pos 联合体的 ev_next_with_common_timeout 成员指出了该定时器在通用定时器队列中的位置。对于其他定时器而言，ev_timeout_pos 联合体的 min_heap_idx 成员指出了该定时器在时间堆中的位置。一个定时器是否是通用定时器取决于其超时值大小，具体判断原则请读者自己参考 event.c 文件中的 is_common_timeout 函数。
- _ev。这是一个联合体。所有具有相同文件描述符值的 I/O 事件处理器通过 ev.ev_io.ev_io_next 成员串联成一个尾队列，我们称之为 I/O 事件队列；所有具有相同信号值的信号事件处理器通过 ev.ev_signal.ev_signal_next 成员串联成一个尾队列，我们称之为信号事件队列。ev.ev_signal.ev_ncalls 成员指定信号事件发生时，Reactor 需要执行多少次该事件对应的事件处理器中的回调函数。ev.ev_signal.ev_pncalls 指针成员要么是 NULL，要么指向 ev.ev_signal.ev_ncalls。

在程序中，我们可能针对同一个 socket 文件描述符上的可读 / 可写事件创建多个事件处理器（它们拥有不同的回调函数）。当该文件描述符上有可读 / 可写事件发生时，所有这些事件处理器都应该被处理。所以，Libevent 使用 I/O 事件队列将具有相同文件描述符值的事件处理器组织在一起。这样，当一个文件描述符上有事件发生时，事件多路分发器就能很快地把所有相关的事件处理器添加到活动事件队列中。信号事件队列的存在也是由于相同的原因。可见，I/O 事件队列和信号事件队列并不是注册事件队列的细致分类，而是另有用处。

- ev_fd。对于 I/O 事件处理器，它是文件描述符值；对于信号事件处理器，它是信号值。
- ev_base。该事件处理器从属的 event_base 实例。
- ev_res。它记录当前激活事件的类型。
- ev_flags。它是一些事件标志。其可选值定义在 include/event2/event_struct.h 文件中：

```
#define EVLIST_TIMEOUT 0x01 /* 事件处理器从属于通用定时器队列或时间堆 */
```

```
#define EVLIST_INSERTED 0x02 /* 事件处理器从属于注册事件队列 */
#define EVLIST_SIGNAL 0x04 /* 没有使用 */
#define EVLIST_ACTIVE 0x08 /* 事件处理器从属于活动事件队列 */
#define EVLIST_INTERNAL 0x10 /* 内部使用 */
#define EVLIST_INIT 0x80 /* 事件处理器已经被初始化 */
#define EVLIST_ALL (0xf000 | 0x9f) /* 定义所有标志 */
```

- ev_pri。它指定事件处理器优先级，值越小则优先级越高。
- ev_closure。它指定 event_base 执行事件处理器的回调函数时的行为。其可选值定义于 event-internal.h 文件中：

```
/* 默认行为 */
#define EV_CLOSURE_NONE 0
/* 执行信号事件处理器的回调函数时，调用 ev.ev_signal.ev_ncalls 次该回调函数 */
#define EV_CLOSURE_SIGNAL 1
/* 执行完回调函数后，再次将事件处理器加入注册事件队列中 */
#define EV_CLOSURE_PERSIST 2
```

- ev_timeout。它仅对定时器有效，指定定时器的超时值。
- ev_callback。它是事件处理器的回调函数，由 event_base 调用。回调函数被调用时，它的 3 个参数分别被传入事件处理器的如下 3 个成员：ev_fd、ev_res 和 ev_arg。
- ev_arg。回调函数的参数。

12.2.4 往注册事件队列中添加事件处理器

前面提到，创建一个 event 对象的函数是 event_new（及其变体），它在 event.c 文件中实现。该函数的实现相当简单，主要是给 event 对象分配内存并初始化它的部分成员，因此我们不讨论它。event 对象创建好之后，应用程序需要调用 event_add 函数将其添加到注册事件队列中，并将对应的事件注册到事件多路分发器上。event_add 函数在 event.c 文件中实现，主要是调用另外一个内部函数 event_add_internal，如代码清单 12-3 所示。

代码清单 12-3 event_add_internal 函数

```
static inline int event_add_internal(struct event *ev, const struct timeval *tv,
                                     int tv_is_absolute)
{
    struct event_base *base = ev->ev_base;
    int res = 0;
    int notify = 0;

    EVENT_BASE_ASSERT_LOCKED(base);
    _event_debug_assert_is_setup(ev);

    event_debug((
        "event_add: event: %p (fd %d), %s%s%scall %p",
        ev,
        (int)ev->ev_fd,
        ev->ev_events & EV_READ ? "EV_READ" : " ",
        ev->ev_timeout ? "timeout" : "callback",
        ev->ev_revents & EV_READ ? "EV_READ" : " ",
        ev->ev_revents & EV_WRITE ? "EV_WRITE" : " "
    ));
```

```

    ev->ev_events & EV_WRITE ? "EV_WRITE" : " ",
    tv ? "EV_TIMEOUT" : " ",
    ev->ev_callback));

EVUTIL_ASSERT(!(ev->ev_flags & ~EVLIST_ALL));

/* 如果新添加的事件处理器是定时器，且它尚未被添加到通用定时器队列或时间堆中，则为该定时器
在时间堆上预留一个位置 */
if (tv != NULL && !(ev->ev_flags & EVLIST_TIMEOUT)) {
    if (min_heap_reserve(&base->timeheap,
        1 + min_heap_size(&base->timeheap)) == -1)
        return (-1);
}

/* 如果当前调用者不是主线程（执行事件循环的线程），并且被添加的事件处理器是信号事件处理器，
而且主线程正在执行该信号事件处理器的回调函数，则当前调用者必须等待主线程完成调用，否则将引起竞态条件
（考虑 event 结构体的 ev_ncalls 和 ev_pncalls 成员）*/
#ifndef _EVENT_DISABLE_THREAD_SUPPORT
if (base->current_event == ev && (ev->ev_events & EV_SIGNAL)
    && !EVBASE_IN_THREAD(base)) {
    ++base->current_event_waiters;
    EVTHREAD_COND_WAIT(base->current_event_cond, base->th_base_lock);
}
#endif

if ((ev->ev_events & (EV_READ|EV_WRITE|EV_SIGNAL)) &&
    !(ev->ev_flags & (EVLIST_INSERTED|EVLIST_ACTIVE))) {
    if (ev->ev_events & (EV_READ|EV_WRITE))
        /* 添加 I/O 事件和 I/O 事件处理器的映射关系 */
        res = evmap_io_add(base, ev->ev_fd, ev);
    else if (ev->ev_events & EV_SIGNAL)
        /* 添加信号事件和信号事件处理器的映射关系 */
        res = evmap_signal_add(base, (int)ev->ev_fd, ev);
    if (res != -1)
        /* 将事件处理器插入注册事件队列 */
        event_queue_insert(base, ev, EVLIST_INSERTED);
    if (res == 1) {
        /* 事件多路分发器中添加了新的事件，所以要通知主线程 */
        notify = 1;
        res = 0;
    }
}

/* 下面将事件处理器添加至通用定时器队列或时间堆中。对于信号事件处理器和 I/O 事件处理器，根据
evmap_*_add 函数的结果决定是否添加（这是为了给事件设置超时）；而对于定时器，则始终应该添加之 */
if (res != -1 && tv != NULL) {
    struct timeval now;
    int common_timeout;

    /* 对于永久性事件处理器，如果其超时时间不是绝对时间，则将该事件处理器的超时时间记录
在变量 ev->ev_io_timeout 中。ev_io_timeout 是定义在 event-internal.h 文件中的宏：#define ev-
io_timeout _ev.ev.io.ev_timeout */
    if (ev->ev_closure == EV_CLOSURE_PERSIST && !tv_is_absolute)

```

```

    ev->ev_io_timeout = *tv;

    /* 如果该事件处理器已经被插入通用定时器队列或时间堆中，则先删除它 */
    if (ev->ev_flags & EVLIST_TIMEOUT) {
        if (min_heap_elt_is_top(ev))
            notify = 1;
        event_queue_remove(base, ev, EVLIST_TIMEOUT);
    }

    /* 如果待添加的事件处理器已经被激活，且原因是超时，则从活动事件队列中删除它，以避免
    其回调函数被执行。对于信号事件处理器，必要时还需将其ncalls成员设置为0（注意，ev_pncalls如果不为
    NULL，它指向ncalls）。前面提到，信号事件被触发时，ncalls指定其回调函数被执行的次数。将ncalls设
    置为0，可以干净地终止信号事件的处理 */
    if ((ev->ev_flags & EVLIST_ACTIVE) &&
        (ev->ev_res & EV_TIMEOUT)) {
        if (ev->ev_events & EV_SIGNAL) {
            if (ev->ev_ncalls && ev->ev_pncalls) {
                *ev->ev_pncalls = 0;
            }
        }
    }

    event_queue_remove(base, ev, EVLIST_ACTIVE);
}

gettime(base, &now);

common_timeout = is_common_timeout(tv, base);
if (tv_is_absolute) {
    ev->ev_timeout = *tv;
    /* 判断应该将定时器插入通用定时器队列，还是插入时间堆 */
} else if (common_timeout) {
    struct timeval tmp = *tv;
    tmp.tv_usec &= MICROSECONDS_MASK;
    evutil_timeradd(&now, &tmp, &ev->ev_timeout);
    ev->ev_timeout.tv_usec |=
        (tv->tv_usec & ~MICROSECONDS_MASK);
} else {
    /* 加上当前系统时间，以取得定时器超时的绝对时间 */
    evutil_timeradd(&now, tv, &ev->ev_timeout);
}

event_debug((
    "event_add: timeout in %d seconds, call %p",
    (int)tv->tv_sec, ev->ev_callback));

event_queue_insert(base, ev, EVLIST_TIMEOUT); /* 最后，插入定时器 */
/* 如果被插入的事件处理器是通用定时器队列中的第一个元素，则通过调用common_timeout_
schedule函数将其转移到时间堆中。这样，通用定时器链表和时间堆中的定时器就得到了统一的处理 */
if (common_timeout) {
    struct common_timeout_list *ctl =
        get_common_timeout_list(base, &ev->ev_timeout);
    if (ev == TAILQ_FIRST(&ctl->events)) {
        common_timeout_schedule(ctl, &now, ev);
}

```

```

        }
    } else {
        if (min_heap_elt_is_top(ev))
            notify = 1;
    }
}

/* 如果必要，唤醒主线程 */
if (res != -1 && notify && EVBASE_NEED_NOTIFY(base))
    evthread_notify_base(base);

_event_debug_note_add(ev);

return (res);
}

```

从代码清单 12-3 可见，`event_add_internal` 函数内部调用了几个重要的函数：

- `evmap_io_add`。该函数将 I/O 事件添加到事件多路分发器中，并将对应的事件处理器添加到 I/O 事件队列中，同时建立 I/O 事件和 I/O 事件处理器之间的映射关系。我们将在下一节详细讨论该函数。
- `evmap_signal_add`。该函数将信号事件添加到事件多路分发器中，并将对应的事件处理器添加到信号事件队列中，同时建立信号事件和信号事件处理器之间的映射关系。
- `event_queue_insert`。该函数将事件处理器添加到各种事件队列中：将 I/O 事件处理器和信号事件处理器插入注册事件队列；将定时器插入通用定时器队列或时间堆；将被激活的事件处理器添加到活动事件队列中。其实现如代码清单 12-4 所示。

代码清单 12-4 `event_queue_insert` 函数

```

static void event_queue_insert(struct event_base *base, struct event *ev,
                               int queue)
{
    EVENT_BASE_ASSERT_LOCKED(base);
    /* 避免重复插入 */
    if (ev->ev_flags & queue) {
        /* Double insertion is possible for active events */
        if (queue & EVLIST_ACTIVE)
            return;

        event_errx(1, "%s: %p(fd %d) already on queue %x",
                  __func__, ev, ev->ev_fd, queue);
        return;
    }

    if (~ev->ev_flags & EVLIST_INTERNAL)
        base->event_count++; /* 将 event_base 拥有的事件处理器总数加 1 */

    ev->ev_flags |= queue; /* 标记此事件已被添加过 */
}

```

```

switch (queue) {
    /* 将 I/O 事件处理器或信号事件处理器插入注册事件队列 */
    case EVLIST_INSERTED:
        TAILQ_INSERT_TAIL(&base->eventqueue, ev, ev_next);
        break;
    /* 将就绪事件处理器插入活动事件队列 */
    case EVLIST_ACTIVE:
        base->event_count_active++;
        TAILQ_INSERT_TAIL(&base->activequeues[ev->ev_pri],
                          ev, ev_active_next);
        break;
    /* 将定时器插入通用定时器队列或时间堆 */
    case EVLIST_TIMEOUT: {
        if (is_common_timeout(&ev->ev_timeout, base)) {
            struct common_timeout_list *ctl =
                get_common_timeout_list(base, &ev->ev_timeout);
            insert_common_timeout_inorder(ctl, ev);
        } else
            min_heap_push(&base->timeheap, ev);
        break;
    }
    default:
        event_errx(l, "%s: unknown queue %x", __func__, queue);
}
}

```

12.2.5 往事件多路分发器中注册事件

`event_queue_insert` 函数所做的仅仅是将一个事件处理器加入 `event_base` 的某个事件队列中。对于新添加的 I/O 事件处理器和信号事件处理器，我们还需要让事件多路分发器来监听其对应的事件，同时建立文件描述符、信号值与事件处理器之间的映射关系。这就要通过调用 `evmap_io_add` 和 `evmap_signal_add` 两个函数来完成。这两个函数相当于事件多路分发器中的 `register_event` 方法，它们由 `evmap.c` 文件实现。不过在讨论它们之前，我们先介绍一下它们将用到的一些重要数据结构，如代码清单 12-5 所示。

代码清单 12-5 `evmap_io`、`event_io_map` 和 `evmap_signal`、`evmap_signal_map`

```

#ifndef EVMAP_USE_HT
#include "ht-internal.h"
struct event_map_entry;
/* 如果定义了 EVMAP_USE_HT，则将 event_io_map 定义为哈希表。该哈希表存储 event_map_entry
对象和 I/O 事件队列（见前文，具有同样文件描述符值的 I/O 事件处理器构成 I/O 事件队列）之间的映射关系。
实际上也就是存储了文件描述符和 I/O 事件处理器之间的映射关系 */
HT_HEAD(event_io_map, event_map_entry);
#else /* 否则 event_io_map 和下面的 event_signal_map 一样 */
#define event_io_map event_signal_map
#endif

/* 下面这个结构体中的 entries 数组成员存储信号值和信号事件处理器之间的映射关系（用信号值索引数组 entries 即得到对应的信号事件处理器） */

```

```

struct event_signal_map {
    void **entries; /* 用于存放 evmap_io 或 evmap_signal 的数组 */
    int nentries; /* entries 数组的大小 */
};

/* 如果定义了 EVMAP_USE_HT，则哈希表 event_io_map 中的成员具有如下类型 */
struct event_map_entry {
    HT_ENTRY(event_map_entry) map_node;
    evutil_socket_t fd;
    union {
        struct evmap_io evmap_io;
    } ent;
};

/* event_list 是由 event 组成的尾队列，前面讨论的所有事件队列都是这种类型 */
TAILQ_HEAD(event_list, event);

/* I/O 事件队列（确切地说，evmap_io.events 才是 I/O 事件队列）*/
struct evmap_io {
    struct event_list events;
    ev_uint16_t nread;
    ev_uint16_t nwrite;
};

/* 信号事件队列（确切地说，evmap_signal.events 才是信号事件队列）*/
struct evmap_signal {
    struct event_list events;
};

```

由于 evmap_io_add 和 evmap_signal_add 两个函数的逻辑基本相同，因此我们仅讨论 evmap_io_add 函数，如代码清单 12-6 所示。

代码清单 12-6 evmap_io_add 函数

```

int evmap_io_add(struct event_base *base, evutil_socket_t fd, struct event *ev)
{
    /* 获得 event_base 的后端 I/O 复用机制实例 */
    const struct eventop *evsel = base->evsel;
    /* 获得 event_base 中文件描述符与 I/O 事件队列的映射表（哈希表或数组）*/
    struct event_io_map *io = &base->io;
    /* fd 参数对应的 I/O 事件队列 */
    struct evmap_io *ctx = NULL;
    int nread, nwrite, retval = 0;
    short res = 0, old = 0;
    struct event *old_ev;

    EVUTIL_ASSERT(fd == ev->ev_fd);

    if (fd < 0)
        return 0;

#ifndef EVMAP_USE_HT

```

```

/* I/O 事件队列数组 io.entries 中，每个文件描述符占用一项。如果 fd 大于当前数组的大小，则增加数组的大小（扩大后的数组的容量要大于 fd）*/
if (fd >= io->nentries) {
    if (evmap_make_space(io, fd, sizeof(struct evmap_io *)) == -1)
        return (-1);
}
#endif
/* 下面这个宏根据 EVMAP_USE_HT 是否被定义而有不同的实现，但目的都是创建 ctx，在映射表 io 中为 fd 和 ctx 添加映射关系 */
GET_IO_SLOT_ANDCTOR(ctx, io, fd, evmap_io, evmap_io_init, evsel->fdinfo_len);

nread = ctx->nread;
nwrite = ctx->nwrite;

if (nread)
    old |= EV_READ;
if (nwrite)
    old |= EV_WRITE;

if (ev->ev_events & EV_READ) {
    if (++nread == 1)
        res |= EV_READ;
}
if (ev->ev_events & EV_WRITE) {
    if (++nwrite == 1)
        res |= EV_WRITE;
}
if (EVUTIL_UNLIKELY(nread > 0xffff || nwrite > 0xffff)) {
    event_warnx("Too many events reading or writing on fd %d",
               (int)fd);
    return -1;
}
if (EVENT_DEBUG_MODE_IS_ON() &&
    (old_ev = TAILQ_FIRST(&ctx->events)) &&
    (old_ev->ev_events&EV_ET) != (ev->ev_events&EV_ET)) {
    event_warnx("Tried to mix edge-triggered and non-edge-triggered"
               " events on fd %d", (int)fd);
    return -1;
}

if (res) {
    void *extra = ((char*)ctx) + sizeof(struct evmap_io);
    /* 往事件多路分发器中注册事件。add 是事件多路分发器的接口函数之一。对不同的后端 I/O 复用机制，这些接口函数有不同的实现。我们将在后面讨论事件多路分发器的接口函数 */
    if (evsel->add(base, ev->ev_fd,
                     old, (ev->ev_events & EV_ET) | res, extra) == -1)
        return (-1);
    retval = 1;
}

ctx->nread = (ev_uint16_t) nread;
ctx->nwrite = (ev_uint16_t) nwrite;

```

```

    /* 将 ev 插到 I/O 事件队列 ctx 的尾部。ev_io_next 是定义在 event-internal.h 文件中
的宏: #define ev_io_next _ev.ev.io.ev_io_next */
    TAILQ_INSERT_TAIL(&ctx->events, ev, ev_io_next);

    return (retval);
}

```

12.2.6 eventop 结构体

eventop 结构体封装了 I/O 复用机制必要的一些操作，比如注册事件、等待事件等。它为 event_base 支持的所有后端 I/O 复用机制提供了一个统一的接口。该结构体定义在 event-internal.h 文件中，如代码清单 12-7 所示。

代码清单 12-7 eventop 结构体

```

struct eventop {
    /* 后端 I/O 复用技术的名称 */
    const char *name;
    /* 初始化函数 */
    void *(*init)(struct event_base *);
    /* 注册事件 */
    int (*add)(struct event_base *, evutil_socket_t fd, short old,
               short events, void *fdinfo);
    /* 删除事件 */
    int (*del)(struct event_base *, evutil_socket_t fd, short old,
               short events, void *fdinfo);
    /* 等待事件 */
    int (*dispatch)(struct event_base *, struct timeval *);
    /* 释放 I/O 复用机制使用的资源 */
    void (*dealloc)(struct event_base *);
    /* 程序调用 fork 之后是否需要重新初始化 event_base */
    int need_reinit;
    /* I/O 复用技术支持的一些特性，可选如下 3 个值的按位或：EV_FEATURE_ET（支持边沿触发
事件 EV_ET）、EV_FEATURE_O1（事件检测算法的复杂度是 O(1)）和 EV_FEATURE_FDS（不仅能监听 socket
上的事件，还能监听其他类型的文件描述符上的事件）*/
    enum event_method_feature features;
    /* 有的 I/O 复用机制需要为每个 I/O 事件队列和信号事件队列分配额外的内存，以避免同一个文
件描述符被重复插入 I/O 复用机制的事件表中。evmap_io_add(或 evmap_io_del) 函数在调用 eventop 的 add(或
del) 方法时，将这段内存的起始地址作为第 5 个参数传递给 add(或 del) 方法。下面这个成员则指定了这段内存的
长度 */
    size_t fdinfo_len;
};

```

前文提到，devpoll.c、kqueue.c、evport.c、select.c、win32select.c、poll.c 和 epoll.c 文件分别针对不同的 I/O 复用技术实现了 eventop 定义的这套接口。那么，在支持多种 I/O 复用技术的系统上，Libevent 将选择使用哪个呢？这取决于这些 I/O 复用技术的优先级。Libevent 支持的后端 I/O 复用技术及它们的优先级在 event.c 文件中定义，如代码清单 12-8 所示。

代码清单 12-8 Libevent 支持的后端 I/O 复用技术及它们的优先级

```

#ifndef _EVENT_HAVE_EVENT_PORTS
extern const struct eventop evportops;
#endif
#ifndef _EVENT_HAVE_SELECT
extern const struct eventop selectops;
#endif
#ifndef _EVENT_HAVE_POLL
extern const struct eventop pollops;
#endif
#ifndef _EVENT_HAVE_EPOLL
extern const struct eventop epollops;
#endif
#ifndef _EVENT_HAVE_WORKING_KQUEUE
extern const struct eventop kqops;
#endif
#ifndef _EVENT_HAVE_DEVPOLLOP
extern const struct eventop devpollops;
#endif
#ifndef WIN32
extern const struct eventop win32ops;
#endif

static const struct eventop *eventops[] = {
#ifndef _EVENT_HAVE_EVENT_PORTS
    &evportops,
#endif
#ifndef _EVENT_HAVE_WORKING_KQUEUE
    &kqops,
#endif
#ifndef _EVENT_HAVE_EPOLL
    &epollops,
#endif
#ifndef _EVENT_HAVE_DEVPOLLOP
    &devpollops,
#endif
#ifndef _EVENT_HAVE_POLL
    &pollops,
#endif
#ifndef _EVENT_HAVE_SELECT
    &selectops,
#endif
#ifndef WIN32
    &win32ops,
#endif
    NULL
};

```

Libevent 通过遍历 eventops 数组来选择其后端 I/O 复用技术。遍历的顺序是从数组的第一个元素开始，到最后一个元素结束。所以，在 Linux 下，Libevent 默认选择的后端 I/O 复

用技术是 epoll。但很显然，用户可以修改代码清单 12-8 中定义的一系列宏来选择使用不同的后端 I/O 复用技术。

12.2.7 event_base 结构体

结构体 event_base 是 Libevent 的 Reactor。它定义在 event-internal.h 文件中，如代码清单 12-9 所示。

代码清单 12-9 event_base 结构体

```
struct event_base {
    /* 初始化 Reactor 的时候选择一种后端 I/O 复用机制，并记录在如下字段中 */
    const struct eventop *evsel;
    /* 指向 I/O 复用机制真正存储的数据，它通过 evsel 成员的 init 函数来初始化 */
    void *evbase;
    /* 事件变化队列。其用途是：如果一个文件描述符上注册的事件被多次修改，则可以使用缓冲
       来避免重复的系统调用（比如 epoll_ctl）。它仅能用于时间复杂度为 O(1) 的 I/O 复用技术 */
    struct event_changelist changelist;
    /* 指向信号的后端处理机制，目前仅在 singal.h 文件中定义了一种处理方法 */
    const struct eventop *evsigsel;
    /* 信号事件处理器使用的数据结构，其中封装了一个由 socketpair 创建的管道。它用于信号
       处理函数和事件多路分发器之间的通信，这和我们在 10.4 节讨论的统一事件源的思路是一样的 */
    struct evsig_info sig;
    /* 添加到该 event_base 的虚拟事件、所有事件和激活事件的数量 */
    int virtual_event_count;
    int event_count;
    int event_count_active;
    /* 是否执行完活动事件队列上剩余的任务之后就退出事件循环 */
    int event_gotterm;
    /* 是否立即退出事件循环，而不管是否还有任务需要处理 */
    int event_break;
    /* 是否应该启动一个新的事件循环 */
    int event_continue;
    /* 目前正在处理的活动事件队列的优先级 */
    int event_running_priority;
    /* 事件循环是否已经启动 */
    int running_loop;
    /* 活动事件队列数组。索引值越小的队列，优先级越高。高优先级的活动事件队列中的事件处理
       器将被优先处理 */
    struct event_list *activequeues;
    /* 活动事件队列数组的大小，即该 event_base 一共有 nactivequeues 个不同优先级的活
       动事件队列 */
    int nactivequeues;
    /* 下面 3 个成员用于管理通用定时器队列 */
    struct common_timeout_list **common_timeout_queues;
    int n_common_timeouts;
    int n_common_timeouts_allocated;
    /* 存放延返回调函数的链表。事件循环每次成功处理完一个活动事件队列中的所有事件之后，就
       调用一次延返回调函数 */
    struct deferred_cb_queue defer_queue;
    /* 文件描述符和 I/O 事件之间的映射关系表 */
}
```

```

    struct event_io_map io;
    /* 信号值和信号事件之间的映射关系表 */
    struct event_signal_map sigmap;
    /* 注册事件队列，存放 I/O 事件处理器和信号事件处理器 */
    struct event_list eventqueue;
    /* 时间堆 */
    struct min_heap timeheap;
    /* 管理系统时间的一些成员 */
    struct timeval event_tv;
    struct timeval tv_cache;
#if defined(_EVENT_HAVE_CLOCK_GETTIME) && defined(CLOCK_MONOTONIC)
    struct timeval tv_clock_diff;
    time_t last_updated_clock_diff;
#endif

/* 多线程支持 */
#ifndef _EVENT_DISABLE_THREAD_SUPPORT
    unsigned long th_owner_id; /* 当前运行该 event_base 的事件循环的线程 */
    void *th_base_lock; /* 对 event_base 的独占锁 */
    /* 当前事件循环正在执行哪个事件处理器的回调函数 */
    struct event *current_event;
    /* 条件变量（见第 14 章），用于唤醒正在等待某个事件处理完毕的线程 */
    void *current_event_cond;
    int current_event_waiters; /* 等待 current_event_cond 的线程数 */
#endif

#endif

#ifdef WIN32
    struct event_iocp_port *iocp;
#endif

    /* 该 event_base 的一些配置参数 */
    enum event_base_config_flag flags;
    /* 下面这组成员变量给工作线程唤醒主线程提供了方法（使用 socketpair 创建的管道）*/
    int is_notify_pending;
    evutil_socket_t th_notify_fd[2];
    struct event th_notify;
    int (*th_notify_fn)(struct event_base *base);
};


```

12.2.8 事件循环

最后，我们讨论一下 Libevent 的“动力”，即事件循环。Libevent 中实现事件循环的函数是 `event_base_loop`。该函数首先调用 I/O 事件多路分发器的事件监听函数，以等待事件；当有事件发生时，就依次处理之。`event_base_loop` 函数的实现如代码清单 12-10 所示。

代码清单 12-10 `event_base_loop` 函数

```

int event_base_loop(struct event_base *base, int flags)
{
    const struct eventop *evsel = base->evsel;

```

```

    struct timeval tv;
    struct timeval *tv_p;
    int res, done, retval = 0;

    EVBASE_ACQUIRE_LOCK(base, th_base_lock);

    /* 一个 event_base 仅允许运行一个事件循环 */
    if (base->running_loop) {
        event_warnx("%s: reentrant invocation. Only one event_base_loop"
                   " can run on each event_base at once.", __func__);
        EVBASE_RELEASE_LOCK(base, th_base_lock);
        return -1;
    }

    base->running_loop = 1; /* 标记该 event_base 已经开始运行 */

    clear_time_cache(base); /* 清除 event_base 的系统时间缓存 */

    /* 设置信号事件的 event_base 实例 */
    if (base->sig.ev_signal_added && base->sig.ev_n_signals_added)
        evsig_set_base(base);

    done = 0;

#ifndef _EVENT_DISABLE_THREAD_SUPPORT
    base->th_owner_id = EVTHREAD_GET_ID();
#endif

    base->event_gotterm = base->event_break = 0;
    while (!done) {
        base->event_continue = 0;

        if (base->event_gotterm) {
            break;
        }

        if (base->event_break) {
            break;
        }

        timeout_correct(base, &tv); /* 校准系统时间 */

        tv_p = &tv;
        if (!N_ACTIVE_CALLBACKS(base)
            && !(flags & EVLOOP_NONBLOCK)) {
            /* 获取时间堆上堆顶元素的超时值，即 I/O 复用系统调用本次应该设置的超
时值 */
            timeout_next(base, &tv_p);
        } else {
            /* 如果有就绪事件尚未处理，则将 I/O 复用系统调用的超时时间“置 0”。
这样 I/O 复用系统调用直接返回，程序也就可以立即处理就绪事件了 */
            evutil_timerclear(&tv);
        }
    }
}

```

```

    }

    /* 如果 event_base 中没有注册任何事件，则直接退出事件循环 */
    if (!event_haveevents(base) && !N_ACTIVE_CALLBACKS(base)) {
        event_debug("%s: no events registered.", __func__);
        retval = 1;
        goto done;
    }

    /* 更新系统时间，并清空时间缓存 */
    gettime(base, &base->event_tv);
    clear_time_cache(base);

    /* 调用事件多路分发器的 dispatch 方法等待事件，将就绪事件插入活动事件队列 */
    res = evsel->dispatch(base, tv_p);

    if (res == -1) {
        event_debug("%s: dispatch returned unsuccessfully.", __func__);
        retval = -1;
        goto done;
    }

    update_time_cache(base); /* 将时间缓存更新为当前系统时间 */
    /* 检查时间堆上的到期事件并依次执行之 */
    timeout_process(base);
    if (N_ACTIVE_CALLBACKS(base)) {
        /* 调用 event_process_active 函数依次处理就绪的信号事件和 I/O 事件 */
        int n = event_process_active(base);
        if ((flags & EVLOOP_ONCE)
            && N_ACTIVE_CALLBACKS(base) == 0
            && n != 0)
            done = 1;
        } else if (flags & EVLOOP_NONBLOCK)
            done = 1;
    }
    event_debug("%s: asked to terminate loop.", __func__);
}

done:
/* 事件循环结束，清空时间缓存，并设置停止循环标志 */
clear_time_cache(base);
base->running_loop = 0;

EVBASE_RELEASE_LOCK(base, th_base_lock);

return (retval);
}

```

至此，我们简要介绍了 Libevent 库的核心代码，但这些还远远不够。要理解 Libevent 的设计理念以及实现上的细节考虑，读者最好自己深入分析其每一行代码。

第 13 章 多进程编程

进程是 Linux 操作系统环境的基础，它控制着系统上几乎所有的活动。本章从系统程序员的角度来讨论 Linux 多进程编程，包括如下内容：

- 复制进程映像的 fork 系统调用和替换进程映像的 exec 系列系统调用。
- 僵尸进程以及如何避免僵尸进程。
- 进程间通信（Inter-Process Communication, IPC）最简单的方式：管道。
- 3 种 System V 进程间通信方式：信号量、消息队列和共享内存。它们都是由 AT&T System V2 版本的 UNIX 引入的，所以统称为 System V IPC。
- 在进程间传递文件描述符的通用方法：通过 UNIX 本地域 socket 传递特殊的辅助数据（关于辅助数据，参考 5.8.3 小节）。

13.1 fork 系统调用

Linux 下创建新进程的系统调用是 fork。其定义如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork( void );
```

该函数的每次调用都返回两次，在父进程中返回的是子进程的 PID，在子进程中则返回 0。该返回值是后续代码判断当前进程是父进程还是子进程的依据。fork 调用失败时返回 -1，并设置 errno。

fork 函数复制当前进程，在内核进程表中创建一个新的进程表项。新的进程表项有很多属性和原进程相同，比如堆指针、栈指针和标志寄存器的值。但也有许多属性被赋予了新的值，比如该进程的 PPID 被设置成原进程的 PID，信号位图被清除（原进程设置的信号处理函数不再对新进程起作用）。

子进程的代码与父进程完全相同，同时它还会复制父进程的数据（堆数据、栈数据和静态数据）。数据的复制采用的是所谓的写时复制（copy on write），即只有在任一进程（父进程或子进程）对数据执行了写操作时，复制才会发生（先是缺页中断，然后操作系统给子进程分配内存并复制父进程的数据）。即便如此，如果我们在程序中分配了大量内存，那么使用 fork 时也应当十分谨慎，尽量避免没必要的内存分配和数据复制。

此外，创建子进程后，父进程中打开的文件描述符默认在子进程中也是打开的，且文件描述符的引用计数加 1。不仅如此，父进程的用户根目录、当前工作目录等变量的引用计数均会加 1。

13.2 exec 系列系统调用

有时我们需要在子进程中执行其他程序，即替换当前进程映像，这就需要使用如下 exec 系列函数之一：

```
#include <unistd.h>
extern char** environ;

int execl( const char* path, const char* arg, ... );
int execvp( const char* file, const char* arg, ... );
int execle( const char* path, const char* arg, ..., char* const envp[] );
int execv( const char* path, char* const argv[] );
int execvp( const char* file, char* const argv[] );
int execve( const char* path, char* const argv[], char* const envp[] );
```

path 参数指定可执行文件的完整路径，file 参数可以接受文件名，该文件的具体位置则在环境变量 PATH 中搜寻。arg 接受可变参数，argv 则接受参数数组，它们都会被传递给新程序（path 或 file 指定的程序）的 main 函数。envp 参数用于设置新程序的环境变量。如果未设置它，则新程序将使用由全局变量 environ 指定的环境变量。

一般情况下，exec 函数是不返回的，除非出错。它出错时返回 -1，并设置 errno。如果没出错，则原程序中 exec 调用之后的代码都不会执行，因为此时原程序已经被 exec 的参数指定的程序完全替换（包括代码和数据）。

exec 函数不会关闭原程序打开的文件描述符，除非该文件描述符被设置了类似 SOCK_CLOEXEC 的属性（见 5.2 节）。

13.3 处理僵尸进程

对于多进程程序而言，父进程一般需要跟踪子进程的退出状态。因此，当子进程结束运行时，内核不会立即释放该进程的进程表表项，以满足父进程后续对该子进程退出信息的查询（如果父进程还在运行）。在子进程结束运行之后，父进程读取其退出状态之前，我们称该子进程处于僵尸态。另外一种使子进程进入僵尸态的情况是：父进程结束或者异常终止，而子进程继续运行。此时子进程的 PPID 将被操作系统设置为 1，即 init 进程。init 进程接管了该子进程，并等待它结束。在父进程退出之后，子进程退出之前，该子进程处于僵尸态。

由此可见，无论哪种情况，如果父进程没有正确地处理子进程的返回信息，子进程都将停留在僵尸态，并占据着内核资源。这是绝对不能容许的，毕竟内核资源有限。下面这对函数在父进程中调用，以等待子进程的结束，并获取子进程的返回信息，从而避免了僵尸进程的产生，或者使子进程的僵尸态立即结束：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait( int* stat_loc );
pid_t waitpid( pid_t pid, int* stat_loc, int options );
```

wait 函数将阻塞进程，直到该进程的某个子进程结束运行为止。它返回结束运行的子进

程的 PID，并将该子进程的退出状态信息存储于 stat_loc 参数指向的内存中。sys/wait.h 头文件中定义了几个宏来帮助解释子进程的退出状态信息，如表 13-1 所示。

表 13-1 子进程状态信息

宏	含义
WIFEXITED(stat_val)	如果子进程正常结束，它就返回一个非 0 值
WEXITSTATUS(stat_val)	如果 WIFEXITED 非 0，它返回子进程的退出码
WIFSIGNALED(stat_val)	如果子进程是因为一个未捕获的信号而终止，它就返回一个非 0 值
WTERMSIG(stat_val)	如果 WIFSIGNALED 非 0，它返回一个信号值
WIFSTOPPED(stat_val)	如果子进程意外终止，它就返回一个非 0 值
WSTOPSIG(stat_val)	如果 WIFSTOPPED 非 0，它返回一个信号值

wait 函数的阻塞特性显然不是服务器程序期望的，而 waitpid 函数解决了这个问题。waitpid 只等待由 pid 参数指定的子进程。如果 pid 取值为 -1，那么它就和 wait 函数相同，即等待任意一个子进程结束。stat_loc 参数的含义和 wait 函数的 stat_loc 参数相同。options 参数可以控制 waitpid 函数的行为。该参数最常用的取值是 WNOHANG。当 options 的取值是 WNOHANG 时，waitpid 调用将是非阻塞的：如果 pid 指定的目标子进程还没有结束或意外终止，则 waitpid 立即返回 0；如果目标子进程确实正常退出了，则 waitpid 返回该子进程的 PID。waitpid 调用失败时返回 -1 并设置 errno。

8.3 节曾提到，要在事件已经发生的情况下执行非阻塞调用才能提高程序的效率。对 waitpid 函数而言，我们最好在某个子进程退出之后再调用它。那么父进程从何得知某个子进程已经退出了呢？这正是 SIGCHLD 信号的用途。当一个进程结束时，它将给其父进程发送一个 SIGCHLD 信号。因此，我们可以在父进程中捕获 SIGCHLD 信号，并在信号处理函数中调用 waitpid 函数以“彻底结束”一个子进程，如代码清单 13-1 所示。

代码清单 13-1 SIGCHLD 信号的典型处理函数

```
static void handle_child( int sig )
{
    pid_t pid;
    int stat;
    while ( ( pid = waitpid( -1, &stat, WNOHANG ) ) > 0 )
    {
        /* 对结束的子进程进行善后处理 */
    }
}
```

13.4 管道

第 6 章中我们介绍过创建管道的系统调用 pipe，我们也多次在代码中利用它来实现进程内部的通信。实际上，管道也是父进程和子进程间通信的常用手段。

管道能在父、子进程间传递数据，利用的是 fork 调用之后两个管道文件描述符（fd[0] 和 fd[1]）都保持打开。一对这样的文件描述符只能保证父、子进程间一个方向的数据传输，父进程和子进程必须有一个关闭 fd[0]，另一个关闭 fd[1]。比如，我们要使用管道实现从父进程向子进程写数据，就应该按照图 13-1 所示来操作。

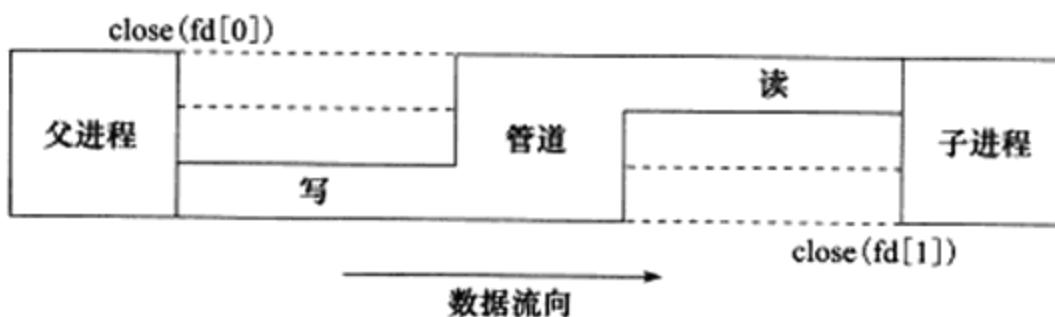


图 13-1 父进程通过管道向子进程写数据

显然，如果要实现父、子进程之间的双向数据传输，就必须使用两个管道。第 6 章中我们还介绍过，socket 编程接口提供了一个创建全双工管道的系统调用：socketpair。squid 服务器程序（见第 4 章）就是利用 socketpair 创建管道，以实现在父进程和日志服务子进程之间传递日志信息，下面我们简单地分析之。在测试机器 Kongming20 上有如下环境：

```

$ ps -ef | grep squid
root      12489      1  0 20:37 ?          00:00:00 squid
squid     12491  12489  0 20:37 ?          00:00:02 (squid-1)
squid     12492  12491  0 20:37 ?          00:00:00 (logfile-daemon) /var/log/squid/access.log
squid     12493  12491  0 20:37 ?          00:00:00 (unlinkd)
$ sudo lsof -p 12491
squid    12491 squid    9u  unix    0xeaf2b440      0t0    40603 socket
$ sudo lsof -p 12492
log_file_ 12492 squid    0u  unix    0xeaf2b680      0t0    40604 socket
log_file_ 12492 squid    1u  unix    0xeaf2b680      0t0    40604 socket
log_file_ 12492 squid    2u  CHR       1,3      0t0    4449 /dev/null
log_file_ 12492 squid    3w  REG       8,3      202   271412 /var/log/squid/access.log

```

这些输出说明 Kongming20 上开启了 squid 服务。该服务创建了几个子进程，其中子进程 12492 专门用于输出日志到 /var/log/squid/access.log 文件。父进程 12491 使用 socketpair 创建了一对 UNIX 域 socket，然后关闭了其中的一个，剩下的那个 socket 的值是 9。子进程 12492 则从父进程 12491 继承了这一对 UNIX 域 socket，并关闭了其中的另外一个，剩下的那个 socket 则被 dup 到标准输入和标准输出上。下面我们 telnet 到 squid 服务上，并向它发送部分数据。同时开启另外两个终端，分别运行 strace 命令以查看进程 12491 和 12492 在这个过程中交换的数据。具体操作如代码清单 13-2 所示。

代码清单 13-2 用 strace 命令查看管道通信

```

$ telnet 192.168.1.109 squid
Trying 192.168.1.109...
Connected to 192.168.1.109.

```

```

Escape character is '^]'.
a (回车)
$ sudo strace -p 12491
write(9, "L1338385956.213      40 192.168.1"..., 104) = 104
$ sudo strace -p 12492
read(0, "L1338385956.213      40 192.168.1"..., 4096) = 104
write(3, "1338385956.213      40 192.168.1."..., 101) = 101

```

由此可见，进程 12491 接收到客户数据后将日志信息输出至管道（写文件描述符 9）。日志服务子进程使用阻塞读操作等待管道上有数据可读（读文件描述符 0），然后将读取到的日志信息写入 /var/log/squid/access.log 文件（写文件描述符 3）。

不过，管道只能用于有关联的两个进程（比如父、子进程）间的通信。而下面要讨论的 3 种 System V IPC 能用于无关联的多个进程之间的通信，因为它们都使用一个全局唯一的键值来标识一条信道。不过，有一种特殊的管道称为 FIFO[⊖]（First In First Out，先进先出），也叫命名管道。它也能用于无关联进程之间的通信。因为 FIFO 管道在网络编程中使用不多，所以本书不讨论它。

13.5 信号量

13.5.1 信号量原语

当多个进程同时访问系统上的某个资源的时候，比如同时写一个数据库的某条记录，或者同时修改某个文件，就需要考虑进程的同步问题，以确保任一时刻只有一个进程可以拥有对资源的独占式访问。通常，程序对共享资源的访问的代码只是很短的一段，但就是这一段代码引发了进程之间的竞态条件。我们称这段代码为关键代码段，或者临界区。对进程同步，也就是确保任一时刻只有一个进程能进入关键代码段。

要编写具有通用目的的代码，以确保关键代码段的独占式访问是非常困难的。有两个名为 Dekker 算法和 Peterson 算法的解决方案，它们试图从语言本身（不需要内核支持）解决并发问题。但它们依赖于忙等待，即进程要持续不断地等待某个内存位置状态的改变。这种方式下 CPU 利用率太低，显然是不可取的。

Dijkstra 提出的信号量（Semaphore）概念是并发编程领域迈出的重要一步。信号量是一种特殊的变量，它只能取自然数值并且只支持两种操作：等待（wait）和信号（signal）。不过在 Linux/UNIX 中，“等待”和“信号”都已经具有特殊的含义，所以对信号量的这两种操作更常用的称呼是 P、V 操作。这两个字母来自于荷兰语单词 passeren（传递，就好像进入临界区）和 vrijgeven（释放，就好像退出临界区）。假设有信号量 SV，则对它的 P、V 操作

[⊖] 这里要注意一下，虽然这种特殊的管道被专门命名为 FIFO，但并不是只有这种管道才遵循先进先出的原则，其实所有的管道都遵循先进先出的原则。

含义如下：

- P(SV)，如果 SV 的值大于 0，就将它减 1；如果 SV 的值为 0，则挂起进程的执行。
- V(SV)，如果有其他进程因为等待 SV 而挂起，则唤醒之；如果没有，则将 SV 加 1。

信号量的取值可以是任何自然数。但最常用的、最简单的信号量是二进制信号量，它只能取 0 和 1 这两个值。本书仅讨论二进制信号量。使用二进制信号量同步两个进程，以确保关键代码段的独占式访问的一个典型例子如图 13-2 所示。

在图 13-2 中，当关键代码段可用时，二进制信号量 SV 的值为 1，进程 A 和 B 都有机会进入关键代码段。如果此时进程 A 执行了 P(SV) 操作将 SV 减 1，则进程 B 若再执行 P(SV) 操作就会被挂起。直到进程 A 离开关键代码段，并执行 V(SV) 操作将 SV 加 1，关键代码段才重新变得可用。如果此时进程 B 因为等待 SV 而处于挂起状态，则它将被唤醒，并进入关键代码段。同样，这时进程 A 如果再执行 P(SV) 操作，则也只能被操作系统挂起以等待进程 B 退出关键代码段。

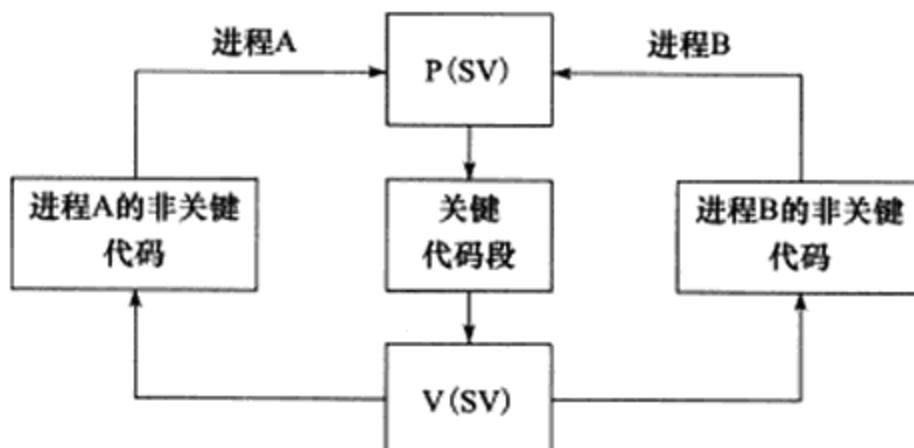


图 13-2 使用信号量保护关键代码段

注意 使用一个普通变量来模拟二进制信号量是行不通的，因为所有高级语言都没有一个原子操作可以同时完成如下两步操作：检测变量是否为 true/false，如果是则再将它设置为 false>true。

Linux 信号量的 API 都定义在 sys/sem.h 头文件中，主要包含 3 个系统调用：semget、semop 和 semctl。它们都被设计为操作一组信号量，即信号量集，而不是单个信号量，因此这些接口看上去多少比我们期望的要复杂一点。我们将分 3 小节依次讨论之。

13.5.2 semget 系统调用

semget 系统调用创建一个新的信号量集，或者获取一个已经存在的信号量集。其定义如下：

```
#include <sys/sem.h>
int semget( key_t key, int num_sems, int sem_flags );
```

key 参数是一个键值，用来标识一个全局唯一的信号量集，就像文件名全局唯一地标识一个文件一样。要通过信号量通信的进程需要使用相同的键值来创建 / 获取该信号量。

num_sems 参数指定要创建 / 获取的信号量集中信号量的数目。如果是创建信号量，则该值必须被指定；如果是获取已经存在的信号量，则可以把它设置为 0。

sem_flags 参数指定一组标志。它低端的 9 个比特是该信号量的权限，其格式和含义

都与系统调用 open 的 mode 参数相同。此外，它还可以和 IPC_CREAT 标志做按位“或”运算以创建新的信号量集。此时即使信号量已经存在，semget 也不会产生错误。我们还可以联合使用 IPC_CREAT 和 IPC_EXCL 标志来确保创建一组新的、唯一的信号量集。在这种情况下，如果信号量集已经存在，则 semget 返回错误并设置 errno 为 EEXIST。这种创建信号量的行为与用 O_CREAT 和 O_EXCL 标志调用 open 来排他式地打开一个文件相似。

semget 成功时返回一个正整数值，它是信号量集的标识符；semget 失败时返回 -1，并设置 errno。

如果 semget 用于创建信号量集，则与之关联的内核数据结构体 semid_ds 将被创建并初始化。semid_ds 结构体的定义如下：

```
#include <sys/sem.h>
/* 该结构体用于描述 IPC 对象（信号量、共享内存和消息队列）的权限 */
struct ipc_perm
{
    key_t key;                                /* 键值 */
    uid_t uid;                                 /* 所有者的有效用户 ID */
    gid_t gid;                                 /* 所有者的有效组 ID */
    uid_t cuid;                                /* 创建者的有效用户 ID */
    gid_t cgid;                                /* 创建者的有效组 ID */
    mode_t mode;                               /* 访问权限 */
    /* 省略其他填充字段 */
};

struct semid_ds
{
    struct ipc_perm sem_perm;                  /* 信号量的操作权限 */
    unsigned long int sem_nsems;                /* 该信号量集中的信号量数目 */
    time_t sem_otime;                          /* 最后一次调用 semop 的时间 */
    time_t sem_ctime;                          /* 最后一次调用 semctl 的时间 */
    /* 省略其他填充字段 */
};
```

semget 对 semid_ds 结构体的初始化包括：

- 将 sem_perm.cuid 和 sem_perm.uid 设置为调用进程的有效用户 ID。
- 将 sem_perm.cgid 和 sem_perm.gid 设置为调用进程的有效组 ID。
- 将 sem_perm.mode 的最低 9 位设置为 sem_flags 参数的最低 9 位。
- 将 sem_nsems 设置为 num_sems。
- 将 sem_otime 设置为 0。
- 将 sem_ctime 设置为当前的系统时间。

13.5.3 semop 系统调用

semop 系统调用改变信号量的值，即执行 P、V 操作。在讨论 semop 之前，我们需要先

介绍与每个信号量关联的一些重要的内核变量：

```
unsigned short semval;           /* 信号量的值 */
unsigned short semzcnt;          /* 等待信号量值变为 0 的进程数量 */
unsigned short semncnt;          /* 等待信号量值增加的进程数量 */
pid_t sempid;                  /* 最后一次执行 semop 操作的进程 ID */
```

semop 对信号量的操作实际上就是对这些内核变量的操作。semop 的定义如下：

```
#include <sys/sem.h>
int semop( int sem_id, struct sembuf* sem_ops, size_t num_sem_ops );
```

sem_id 参数是由 semget 调用返回的信号量集标识符，用以指定被操作的目标信号量集。sem_ops 参数指向一个 sembuf 结构体类型的数组，sembuf 结构体的定义如下：

```
struct sembuf
{
    unsigned short int sem_num;
    short int sem_op;
    short int sem_flg;
}
```

其中，sem_num 成员是信号量集中信号量的编号，0 表示信号量集中的第一个信号量。sem_op 成员指定操作类型，其可选值为正整数、0 和负整数。每种类型的操作的行为又受到 sem_flg 成员的影响。sem_flg 的可选值是 IPC_NOWAIT 和 SEM_UNDO。IPC_NOWAIT 的含义是，无论信号量操作是否成功，semop 调用都将立即返回，这类似于非阻塞 I/O 操作。SEM_UNDO 的含义是，当进程退出时取消正在进行的 semop 操作。具体来说，sem_op 和 sem_flg 将按照如下方式来影响 semop 的行为：

- 如果 sem_op 大于 0，则 semop 将被操作的信号量的值 semval 增加 sem_op。该操作要求调用进程对被操作信号量集拥有写权限。此时若设置了 SEM_UNDO 标志，则系统将更新进程的 semadj 变量（用以跟踪进程对信号量的修改情况）。
- 如果 sem_op 等于 0，则表示这是一个“等待 0”(wait-for-zero) 操作。该操作要求调用进程对被操作信号量集拥有读权限。如果此时信号量的值是 0，则调用立即成功返回。如果信号量的值不是 0，则 semop 失败返回或者阻塞进程以等待信号量变为 0。在这种情况下，当 IPC_NOWAIT 标志被指定时，semop 立即返回一个错误，并设置 errno 为 EAGAIN。如果未指定 IPC_NOWAIT 标志，则信号量的 semzcnt 值加 1，进程被投入睡眠直到下列 3 个条件之一发生：信号量的值 semval 变为 0，此时系统将该信号量的 semzcnt 值减 1；被操作信号量所在的信号量集被进程移除，此时 semop 调用失败返回，errno 被设置为 EIDRM；调用被信号中断，此时 semop 调用失败返回，errno 被设置为 EINTR，同时系统将该信号量的 semzcnt 值减 1。
- 如果 sem_op 小于 0，则表示对信号量值进行减操作，即期望获得信号量。该操作要求调用进程对被操作信号量集拥有写权限。如果信号量的值 semval 大于或等于 sem_op 的绝对值，则 semop 操作成功，调用进程立即获得信号量，并且系统将该信号量的 semval 值减去 sem_op 的绝对值。此时如果设置了 SEM_UNDO 标志，

则系统将更新进程的 semadj 变量。如果信号量的值 semval 小于 sem_op 的绝对值，则 semop 失败返回或者阻塞进程以等待信号量可用。在这种情况下，当 IPC_NOWAIT 标志被指定时，semop 立即返回一个错误，并设置 errno 为 EAGAIN。如果未指定 IPC_NOWAIT 标志，则信号量的 semncnt 值加 1，进程被投入睡眠直到下列 3 个条件之一发生：信号量的值 semval 变得大于或等于 sem_op 的绝对值，此时系统将该信号量的 semncnt 值减 1，并将 semval 减去 sem_op 的绝对值，同时，如果 SEM_UNDO 标志被设置，则系统更新 semadj 变量；被操作信号量所在的信号量集被进程移除，此时 semop 调用失败返回，errno 被设置为 EIDRM；调用被信号中断，此时 semop 调用失败返回，errno 被设置为 EINTR，同时系统将该信号量的 semncnt 值减 1。

`semop` 系统调用的第 3 个参数 `num_sem_ops` 指定要执行的操作个数，即 `sem_ops` 数组中元素的个数。`semop` 对数组 `sem_ops` 中的每个成员按照数组顺序依次执行操作，并且该过程是原子操作，以避免别的进程在同一时刻按照不同的顺序对该信号集中的信号量执行 `semop` 操作导致的竞态条件。

`semop` 成功时返回 0，失败则返回 -1 并设置 `errno`。失败的时候，`sem_ops` 数组中指定的所有操作都不被执行。

13.5.4 semctl 系统调用

semctl 系统调用允许调用者对信号量进行直接控制。其定义如下：

```
#include <sys/sem.h>
int semctl( int sem id, int sem num, int command, ... );
```

`sem_id` 参数是由 `semget` 调用返回的信号量集标识符，用以指定被操作的信号量集。`sem_num` 参数指定被操作的信号量在信号量集中的编号。`command` 参数指定要执行的命令。有的命令需要调用者传递第 4 个参数。第 4 个参数的类型由用户自己定义，但 `sys/sem.h` 头文件给出了它的推荐格式，具体如下：

```

int semopm;           /* semop 一次最多能执行的 sem_op 操作数目 */
int semume;          /* Linux 内核没有使用 */
int semusz;          /* sem_undo 结构体的大小 */
int semvmx;          /* 最大允许的信号量值 */
/* 最多允许的 UNDO 次数 (带 SEM_UNDO 标志的 semop 操作的次数) */
int semaem;
};

semctl 支持的所有命令如表 13-2 所示。

```

表 13-2 semctl 的 command 参数

命 令	含 义	semctl 成功时的返回值
IPC_STAT	将信号量集关联的内核数据结构复制到 semun.buf 中	0
IPC_SET	将 semun.buf 中的部分成员复制到信号量集关联的内核数据结构中，同时内核数据中的 semid_ds.sem_ctime 被更新	0
IPC_RMID	立即移除信号量集，唤醒所有等待该信号量集的进程 (semop 返回错误，并设置 errno 为 EIDRM)	0
IPC_INFO	获取系统信号量资源配置信息，将结果存储在 semun._buf 中。这些信息的含义见结构体 seminfo 的注释部分	内核信号量集数组中已经被使用的项的最大索引值
SEM_INFO	与 IPC_INFO 类似，不过 semun._buf.semusz 被设置为系统目前拥有的信号量集数目，而 semun._buf.semaem 被设置为系统目前拥有的信号量数目	同 IPC_INFO
SEM_STAT	与 IPC_STAT 类似，不过此时 sem_id 参数不是用来表示信号量集标识符，而是内核中信号量集数组的索引（系统的所有信号量集都是该数组中的一项）	内核信号量集数组中索引值为 sem_id 的信号量集的标识符
GETALL	将由 sem_id 标识的信号量集中的所有信号量的 semval 值导出到 semun.array 中	0
GETNCNT	获取信号量的 semncnt 值	信号量的 semncnt 值
GETPID	获取信号量的 sempid 值	信号量的 sempid 值
GETVAL	获得信号量的 semval 值	信号量的 semval 值
GETZCNT	获得信号量的 semzcnt 值	信号量的 semzcnt 值
SETALL	用 semun.array 中的数据填充由 sem_id 标识的信号量集中的所有信号量的 semval 值，同时内核数据中的 semid_ds.sem_ctime 被更新	0
SETVAL	将信号量的 semval 值设置为 semun.val，同时内核数据中的 semid_ds.sem_ctime 被更新	0

注意 这些操作中，GETNCNT、GETPID、GETVAL、GETZCNT 和 SETVAL 操作的是单个信号量，它是由标识符 sem_id 指定的信号量集中的第 sem_num 个信号量；而其他操作针对的是整个信号量集，此时 semctl 的参数 sem_num 被忽略。

semctl 成功时的返回值取决于 command 参数，如表 13-2 所示。semctl 失败时返回 -1，并设置 errno。

13.5.5 特殊键值 IPC_PRIVATE

semget 的调用者可以给其 key 参数传递一个特殊的键值 IPC_PRIVATE（其值为 0），这样无论该信号量是否已经存在，semget 都将创建一个新的信号量。使用该键值创建的信号量并非像它的名字声称的那样是进程私有的。其他进程，尤其是子进程，也有方法来访问这个信号量。所以 semget 的 man 手册的 BUGS 部分上说，使用名字 IPC_PRIVATE 有些误导（历史原因），应该称为 IPC_NEW。比如下面的代码清单 13-3 就在父、子进程间使用一个 IPC_PRIVATE 信号量来同步。

代码清单 13-3 使用 IPC_PRIVATE 信号量

```
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

union semun
{
    int val;
    struct semid_ds* buf;
    unsigned short int* array;
    struct seminfo* __buf;
};

/* op 为 -1 时执行 P 操作，op 为 1 时执行 V 操作 */
void pv( int sem_id, int op )
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = op;
    sem_b.sem_flg = SEM_UNDO;
    semop( sem_id, &sem_b, 1 );
}

int main( int argc, char* argv[] )
{
    int sem_id = semget( IPC_PRIVATE, 1, 0666 );

    union semun sem_un;
    sem_un.val = 1;
    semctl( sem_id, 0, SETVAL, sem_un );

    pid_t id = fork();
    if( id < 0 )
    {
        return 1;
    }
    else if( id == 0 )
    {
        printf( "child try to get binary sem\n" );
    }
}
```

```

/* 在父、子进程间共享 IPC_PRIVATE 信号量的关键就在于二者都可以操作该信号量的标识符
sem_id */
    pv( sem_id, -1 );
    printf( "child get the sem and would release it after 5 seconds\n" );
    sleep( 5 );
    pv( sem_id, 1 );
    exit( 0 );
}
else
{
    printf( "parent try to get binary sem\n" );
    pv( sem_id, -1 );
    printf( "parent get the sem and would release it after 5 seconds\n" );
    sleep( 5 );
    pv( sem_id, 1 );
}

waitpid( id, NULL, 0 );
semctl( sem_id, 0, IPC_RMID, sem_un ); /* 删除信号量 */
return 0;
}

```

另外一个例子是：工作在 prefork 模式下的 httpd 网页服务器程序使用 1 个 IPC_PRIVATE 信号量来同步各子进程对 epoll_wait 的调用权。下面我们简单分析一下这个例子。在测试机器 Kongming20 上，使用 strace 命令依次查看 httpd 的各子进程是如何协调工作的：

```

$ ps -ef | grep httpd
root      1701      1  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1703  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1704  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1705  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1706  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1707  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1708  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1709  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
apache    1710  1701  0 09:17 ?          00:00:00 /usr/sbin/httpd -k start
$ sudo strace -p 1703
semop(393222, {{0, -1, SEM_UNDO}}, 1
$ sudo strace -p 1704
semop(393222, {{0, -1, SEM_UNDO}}, 1
.....
$ sudo strace -p 1709
epoll_wait(14, {}, 2, 10000) = 0
$ sudo strace -p 1710
semop(393222, {{0, -1, SEM_UNDO}}, 1

```

由此可见，httpd 的子进程 1703~1708 和 1710 都在等待信号量 393222（这是一个标识符）可用；只有进程 1709 暂时拥有该信号量，因为进程 1709 调用 epoll_wait 以等待新的客户连接。当有新连接到来时，进程 1709 将接受之，并对信号量 393222 执行 V 操作，此时将有另外一个子进程获得该信号量并调用 epoll_wait 来等待新的客户连接。那么我们如何知道信号

量 393222 是使用键值 IPC_PRIVATE 创建的呢？答案将在 13.8 节揭晓。

下面要讨论另外两种 IPC——共享内存和消息队列。这两种 IPC 在创建资源的时候也支持 IPC_PRIVATE 键值，其含义与信号量的 IPC_PRIVATE 键值完全相同，不再赘述。

13.6 共享内存

共享内存是最高效的 IPC 机制，因为它不涉及进程之间的任何数据传输。这种高效率带来的问题是，我们必须用其他辅助手段来同步进程对共享内存的访问，否则会产生竞态条件。因此，共享内存通常和其他进程间通信方式一起使用。

Linux 共享内存 API 都定义在 sys/shm.h 头文件中，包括 4 个系统调用：shmget、shmat、shmdt 和 shmctl。我们将依次讨论之。

13.6.1 shmget 系统调用

shmget 系统调用创建一段新的共享内存，或者获取一段已经存在的共享内存。其定义如下：

```
#include <sys/shm.h>
int shmget( key_t key, size_t size, int shmflg );
```

和 semget 系统调用一样，key 参数是一个键值，用来标识一段全局唯一的共享内存。size 参数指定共享内存的大小，单位是字节。如果是创建新的共享内存，则 size 值必须被指定。如果是获取已经存在的共享内存，则可以把 size 设置为 0。

shmflg 参数的使用和含义与 semget 系统调用的 sem_flags 参数相同。不过 shmget 支持两个额外的标志——SHM_HUGETLB 和 SHM_NORESERVE。它们的含义如下：

- SHM_HUGETLB，类似于 mmap 的 MAP_HUGETLB 标志，系统将使用“大页面”来为共享内存分配空间。
- SHM_NORESERVE，类似于 mmap 的 MAP_NORESERVE 标志，不为共享内存保留交换分区（swap 空间）。这样，当物理内存不足的时候，对该共享内存执行写操作将触发 SIGSEGV 信号。

shmget 成功时返回一个正整数值，它是共享内存的标识符。shmget 失败时返回 -1，并设置 errno。

如果 shmget 用于创建共享内存，则这段共享内存的所有字节都被初始化为 0，与之关联的内核数据结构 shmid_ds 将被创建并初始化。shmid_ds 结构体的定义如下：

```
struct shmid_ds
{
    struct ipc_perm shm_perm;           /* 共享内存的操作权限 */
    size_t shm_segsz;                  /* 共享内存大小，单位是字节 */
    __time_t shm_atime;                /* 对这段内存最后一次调用 shmat 的时间 */
    __time_t shm_dtime;                /* 对这段内存最后一次调用 shmdt 的时间 */
    __time_t shm_ctime;                /* 对这段内存最后一次调用 shmctl 的时间 */
```

```

    __pid_t shm_cpid;           /* 创建者的 PID */
    __pid_t shm_lpid;          /* 最后一次执行 shmat 或 shmdt 操作的进程的 PID */
    shmatt_t shm_nattach;      /* 目前关联到此共享内存的进程数量 */
    /* 省略一些填充字段 */
};

shmget 对 shmid_ds 结构体的初始化包括：

```

- 将 shm_perm.cuid 和 shm_perm.uid 设置为调用进程的有效用户 ID。
- 将 shm_perm.cgid 和 shm_perm.gid 设置为调用进程的有效组 ID。
- 将 shm_perm.mode 的最低 9 位设置为 shmflg 参数的最低 9 位。
- 将 shm_segsz 设置为 size。
- 将 shm_lpid、shm_nattach、shm_atime、shm_dtime 设置为 0。
- 将 shm_ctime 设置为当前的时间。

13.6.2 shmat 和 shmdt 系统调用

共享内存被创建 / 获取之后，我们不能立即访问它，而是需要先将它关联到进程的地址空间中。使用完共享内存之后，我们也需要将它从进程地址空间中分离。这两项任务分别由如下两个系统调用实现：

```
#include <sys/shm.h>
void* shmat( int shm_id, const void* shm_addr, int shmflg );
int shmdt( const void* shm_addr );
```

其中，shm_id 参数是由 shmget 调用返回的共享内存标识符。shm_addr 参数指定将共享内存关联到进程的哪块地址空间，最终的效果还受到 shmflg 参数的可选标志 SHM_RND 的影响：

- 如果 shm_addr 为 NULL，则被关联的地址由操作系统选择。这是推荐的做法，以确保代码的可移植性。
- 如果 shm_addr 非空，并且 SHM_RND 标志未被设置，则共享内存被关联到 addr 指定的地址处。
- 如果 shm_addr 非空，并且设置了 SHM_RND 标志，则被关联的地址是 [shm_addr - (shm_addr % SHMLBA)]。SHMLBA 的含义是“段低端边界地址倍数”（Segment Low Boundary Address Multiple），它必须是内存页面大小（PAGE_SIZE）的整数倍。现在的 Linux 内核中，它等于一个内存页大小。SHM_RND 的含义是圆整（round），即将共享内存被关联的地址向下圆整到离 shm_addr 最近的 SHMLBA 的整数倍地址处。

除了 SHM_RND 标志外，shmflg 参数还支持如下标志：

- SHM_RDONLY。进程仅能读取共享内存中的内容。若没有指定该标志，则进程可同时对共享内存进行读写操作（当然，这需要在创建共享内存的时候指定其读写权限）。
- SHM_REMAP。如果地址 shmaddr 已经被关联到一段共享内存上，则重新关联。

- SHM_EXEC。它指定对共享内存段的执行权限。对共享内存而言，执行权限实际上和读权限是一样的。

shmat 成功时返回共享内存被关联到的地址，失败则返回 (void*)-1 并设置 errno。shmat 成功时，将修改内核数据结构 shmid_ds 的部分字段，如下：

- 将 shm_nattach 加 1。
- 将 shm_lpid 设置为调用进程的 PID。
- 将 shm_atime 设置为当前的时间。

shmdt 函数将关联到 shm_addr 处的共享内存从进程中分离。它成功时返回 0，失败则返回 -1 并设置 errno。shmdt 在成功调用时将修改内核数据结构 shmid_ds 的部分字段，如下：

- 将 shm_nattach 减 1。
- 将 shm_lpid 设置为调用进程的 PID。
- 将 shm_dtime 设置为当前的时间。

13.6.3 shmctl 系统调用

shmctl 系统调用控制共享内存的一些属性。其定义如下：

```
#include <sys/shm.h>
int shmctl( int shm_id, int command, struct shmid_ds* buf );
```

其中，shm_id 参数是由 shmget 调用返回的共享内存标识符。command 参数指定要执行的命令。shmctl 支持的所有命令如表 13-3 所示。

表 13-3 shmctl 支持的命令

命 令	含 义	shmctl 成功时的返回值
IPC_STAT	将共享内存相关的内核数据结构复制到 buf (第 3 个参数, 下同) 中	0
IPC_SET	将 buf 中的部分成员复制到共享内存相关的内核数据结构中，同时内核数据中的 shmid_ds.shm_ctime 被更新	0
IPC_RMID	将共享内存打上删除的标记。这样当最后一个使用它的进程调用 shmdt 将它从进程中分离时，该共享内存就被删除了	0
IPC_INFO	获取系统共享内存资源配置信息，将结果存储在 buf 中。应用程序需要将 buf 转换成 shminfo 结构体类型来读取这些系统信息。shminfo 结构体与 seminfo 类似，这里不再赘述	内核共享内存信息数组中已经被使用的项的最大索引值
SHM_INFO	与 IPC_INFO 类似，不过返回的是已经分配的共享内存占用的资源信息。应用程序需要将 buf 转换成 shm_info 结构体类型来读取这些信息。shm_info 结构体与 shminfo 类似，这里不再赘述	同 IPC_INFO
SHM_STAT	与 IPC_STAT 类似，不过此时 shm_id 参数不是用来表示共享内存标识符，而是内核中共享内存信息数组的索引（每个共享内存的信息都是该数组中的一项）	内核共享内存信息数组中索引值为 shm_id 的共享内存的标识符
SHM_LOCK	禁止共享内存被移动至交换分区	0
SHM_UNLOCK	允许共享内存被移动至交换分区	0

shmctl 成功时的返回值取决于 command 参数，如表 13-3 所示。shmctl 失败时返回 -1，

并设置 errno。

13.6.4 共享内存的 POSIX 方法

6.5 节中我们介绍过 mmap 函数。利用它的 MAP_ANONYMOUS 标志我们可以实现父、子进程之间的匿名内存共享。通过打开同一个文件，mmap 也可以实现无关进程之间的内存共享。Linux 提供了另外一种利用 mmap 在无关进程之间共享内存的方式。这种方式无须任何文件的支持，但它需要先使用如下函数来创建或打开一个 POSIX 共享内存对象：

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_open( const char* name, int oflag, mode_t mode );
```

shm_open 的使用方法与 open 系统调用完全相同。

name 参数指定要创建 / 打开的共享内存对象。从可移植性的角度考虑，该参数应该使用 “/somename” 的格式：以 “/” 开始，后接多个字符，且这些字符都不是 “/”；以 “\0” 结尾，长度不超过 NAME_MAX（通常是 255）。

oflag 参数指定创建方式。它可以是下列标志中的一个或者多个的按位或：

- O_RDONLY。以只读方式打开共享内存对象。
- O_RDWR。以可读、可写方式打开共享内存对象。
- O_CREAT。如果共享内存对象不存在，则创建之。此时 mode 参数的最低 9 位将指定该共享内存对象的访问权限。共享内存对象被创建的时候，其初始长度为 0。
- O_EXCL。和 O_CREAT 一起使用，如果由 name 指定的共享内存对象已经存在，则 shm_open 调用返回错误，否则就创建一个新的共享内存对象。
- O_TRUNC。如果共享内存对象已经存在，则把它截断，使其长度为 0。

shm_open 调用成功时返回一个文件描述符。该文件描述符可用于后续的 mmap 调用，从而将共享内存关联到调用进程。shm_open 失败时返回 -1，并设置 errno。

和打开的文件最后需要关闭一样，由 shm_open 创建的共享内存对象使用完之后也需要被删除。这个过程是通过如下函数实现的：

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_unlink( const char *name );
```

该函数将 name 参数指定的共享内存对象标记为等待删除。当所有使用该共享内存对象的进程都使用 ummap 将它从进程中分离之后，系统将销毁这个共享内存对象所占据的资源。

如果代码中使用了上述 POSIX 共享内存函数，则编译的时候需要指定链接选项 -lrt。

13.6.5 共享内存实例

在 9.6.2 小节中，我们介绍过一个聊天室服务器程序。下面我们将它修改为一个多进程

服务器：一个子进程处理一个客户连接。同时，我们将所有客户 socket 连接的读缓冲设计为一块共享内存，如代码清单 13-4 所示。

代码清单 13-4 使用共享内存的聊天室服务器程序

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

#define USER_LIMIT 5
#define BUFFER_SIZE 1024
#define FD_LIMIT 65535
#define MAX_EVENT_NUMBER 1024
#define PROCESS_LIMIT 65536

/* 处理一个客户连接必要的数据 */
struct client_data
{
    sockaddr_in address; /* 客户端的 socket 地址 */
    int connfd; /* socket 文件描述符 */
    pid_t pid; /* 处理这个连接的子进程的 PID */
    int pipefd[2]; /* 和父进程通信用的管道 */
};

static const char* shm_name = "/my_shm";
int sig_pipefd[2];
int epollfd;
int listenfd;
int shmfds;
char* share_mem = 0;
/* 客户连接数组，进程用客户连接的编号来索引这个数组，即可取得相关的客户连接数据 */
client_data* users = 0;
/* 子进程和客户连接的映射关系表，用进程的 PID 来索引这个数组，即可取得该进程所处理的客户连接的编号 */
int* sub_process = 0;
/* 当前客户数量 */
int user_count = 0;
bool stop_child = false;

int setnonblocking( int fd )
```

```

{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

void addfd( int epollfd, int fd )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET;
    epoll_ctl( epollfd, EPOLL_CTL_ADD, fd, &event );
    setnonblocking( fd );
}

void sig_handler( int sig )
{
    int save_errno = errno;
    int msg = sig;
    send( sig_pipefd[1], ( char* )&msg, 1, 0 );
    errno = save_errno;
}

void addsig( int sig, void(*handler)(int), bool restart = true )
{
    struct sigaction sa;
    memset( &sa, '\0', sizeof( sa ) );
    sa.sa_handler = handler;
    if( restart )
    {
        sa.sa_flags |= SA_RESTART;
    }
    sigfillset( &sa.sa_mask );
    assert( sigaction( sig, &sa, NULL ) != -1 );
}

void del_resource()
{
    close( sig_pipefd[0] );
    close( sig_pipefd[1] );
    close( listenfd );
    close( epollfd );
    shm_unlink( shm_name );
    delete [] users;
    delete [] sub_process;
}

/* 停止一个子进程 */
void child_term_handler( int sig )
{
    stop_child = true;
}

```

```

/* 子进程运行的函数。参数 idx 指出该子进程处理的客户连接的编号，users 是保存所有客户连接数据的
数组，参数 share_mem 指出共享内存的起始地址 */
int run_child( int idx, client_data* users, char* share_mem )
{
    epoll_event events[ MAX_EVENT_NUMBER ];
    /* 子进程使用 I/O 复用技术来同时监听两个文件描述符：客户连接 socket、与父进程通信的管道文
件描述符 */
    int child_epollfd = epoll_create( 5 );
    assert( child_epollfd != -1 );
    int connfd = users[idx].connfd;
    addfd( child_epollfd, connfd );
    int pipefd = users[idx].pipefd[1];
    addfd( child_epollfd, pipefd );
    int ret;
    /* 子进程需要设置自己的信号处理函数 */
    addsig( SIGTERM, child_term_handler, false );

    while( !stop_child )
    {
        int number = epoll_wait( child_epollfd, events, MAX_EVENT_NUMBER, -1 );
        if ( ( number < 0 ) && ( errno != EINTR ) )
        {
            printf( "epoll failure\n" );
            break;
        }

        for ( int i = 0; i < number; i++ )
        {
            int sockfd = events[i].data.fd;
            /* 本子进程负责的客户连接有数据到达 */
            if( ( sockfd == connfd ) && ( events[i].events & EPOLLIN ) )
            {
                memset( share_mem + idx*BUFFER_SIZE, '\0', BUFFER_SIZE );
                /* 将客户数据读取到对应的读缓存中。该读缓存是共享内存的一段，它开始于
idx*BUFFER_SIZE 处，长度为 BUFFER_SIZE 字节。因此，各个客户连接的读缓存是共享的 */
                ret = recv( connfd, share_mem + idx*BUFFER_SIZE, BUFFER_SIZE-1, 0 );
                if( ret < 0 )
                {
                    if( errno != EAGAIN )
                    {
                        stop_child = true;
                    }
                }
                else if( ret == 0 )
                {
                    stop_child = true;
                }
                else
                {
                    /* 成功读取客户数据后就通知主进程（通过管道）来处理 */
                    send( pipefd, ( char* )&idx, sizeof( idx ), 0 );
                }
            }
        }
    }
}

```

```

/* 主进程通知本进程（通过管道）将第 client 个客户的数据发送到本进程负责的客户端 */
else if( ( sockfd == pipefd ) && ( events[i].events & EPOLLIN ) )
{
    int client = 0;
    /* 接收主进程发送来的数据，即有客户数据到达的连接的编号 */
    ret = recv( sockfd, ( char* )&client, sizeof( client ), 0 );
    if( ret < 0 )
    {
        if( errno != EAGAIN )
        {
            stop_child = true;
        }
    }
    else if( ret == 0 )
    {
        stop_child = true;
    }
    else
    {
        send( connfd, share_mem + client * BUFFER_SIZE,
              BUFFER_SIZE, 0 );
    }
}
else
{
    continue;
}
}

close( connfd );
close( pipefd );
close( child_epollfd );
return 0;
}

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    listenfd = socket( PF_INET, SOCK_STREAM, 0 );

```

```
assert( listenfd >= 0 );

ret = bind( listenfd, ( struct sockaddr* )&address, sizeof( address ) );
assert( ret != -1 );

ret = listen( listenfd, 5 );
assert( ret != -1 );

user_count = 0;
users = new client_data [ USER_LIMIT+1 ];
sub_process = new int [ PROCESS_LIMIT ];
for( int i = 0; i < PROCESS_LIMIT; ++i )
{
    sub_process[i] = -1;
}

epoll_event events[ MAX_EVENT_NUMBER ];
epollfd = epoll_create( 5 );
assert( epollfd != -1 );
addfd( epollfd, listenfd );

ret = socketpair( PF_UNIX, SOCK_STREAM, 0, sig_pipefd );
assert( ret != -1 );
setnonblocking( sig_pipefd[1] );
addfd( epollfd, sig_pipefd[0] );

addsig( SIGCHLD, sig_handler );
addsig( SIGTERM, sig_handler );
addsig( SIGINT, sig_handler );
addsig( SIGPIPE, SIG_IGN );
bool stop_server = false;
bool terminate = false;

/* 创建共享内存，作为所有客户 socket 连接的读缓存 */
shmfds = shm_open( shm_name, O_CREAT | O_RDWR, 0666 );
assert( shmfds != -1 );
ret = ftruncate( shmfds, USER_LIMIT * BUFFER_SIZE );
assert( ret != -1 );

share_mem = (char*)mmap( NULL, USER_LIMIT * BUFFER_SIZE, PROT_READ |
                        PROT_WRITE, MAP_SHARED, shmfds, 0 );
assert( share_mem != MAP_FAILED );
close( shmfds );

while( !stop_server )
{
    int number = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, -1 );
    if ( ( number < 0 ) && ( errno != EINTR ) )
    {
        printf( "epoll failure\n" );
        break;
    }
}
```

```

for ( int i = 0; i < number; i++ )
{
    int sockfd = events[i].data.fd;
    /* 新的客户连接到来 */
    if( sockfd == listenfd )
    {
        struct sockaddr_in client_address;
        socklen_t client_addrlength = sizeof( client_address );
        int connfd = accept( listenfd, ( struct sockaddr* )
                            &client_address, &client_addrlength );
        if ( connfd < 0 )
        {
            printf( "errno is: %d\n", errno );
            continue;
        }
        if( user_count >= USER_LIMIT )
        {
            const char* info = "too many users\n";
            printf( "%s", info );
            send( connfd, info, strlen( info ), 0 );
            close( connfd );
            continue;
        }
        /* 保存第 user_count 个客户连接的相关数据 */
        users[user_count].address = client_address;
        users[user_count].connfd = connfd;
        /* 在主进程和子进程间建立管道，以传递必要的数据 */
        ret = socketpair( PF_UNIX, SOCK_STREAM, 0, users[user_count].pipefd );
        assert( ret != -1 );
        pid_t pid = fork();
        if( pid < 0 )
        {
            close( connfd );
            continue;
        }
        else if( pid == 0 )
        {
            close( epollfd );
            close( listenfd );
            close( users[user_count].pipefd[0] );
            close( sig_pipefd[0] );
            close( sig_pipefd[1] );
            run_child( user_count, users, share_mem );
            munmap( (void*)share_mem, USER_LIMIT * BUFFER_SIZE );
            exit( 0 );
        }
        else
        {
            close( connfd );
            close( users[user_count].pipefd[1] );
        }
    }
}

```

```

        addfd( epollfd, users[user_count].pipefd[0] );
        users[user_count].pid = pid;
        /* 记录新的客户连接在数组 users 中的索引值，建立进程 pid 和该索引值之间
的映射关系 */
        sub_process[pid] = user_count;
        user_count++;
    }
}

/* 处理信号事件 */
else if( ( sockfd == sig_pipefd[0] ) && ( events[i].events & EPOLLIN ) )
{
    int sig;
    char signals[1024];
    ret = recv( sig_pipefd[0], signals, sizeof( signals ), 0 );
    if( ret == -1 )
    {
        continue;
    }
    else if( ret == 0 )
    {
        continue;
    }
    else
    {
        for( int i = 0; i < ret; ++i )
        {
            switch( signals[i] )
            {
                /* 子进程退出，表示有某个客户端关闭了连接 */
                case SIGCHLD:
                {
                    pid_t pid;
                    int stat;
                    while ( ( pid = waitpid( -1, &stat, WNOHANG ) ) > 0 )
                    {
                        /* 用子进程的 pid 取得被关闭的客户连接的编号 */
                        int del_user = sub_process[pid];
                        sub_process[pid] = -1;
                        if( ( del_user < 0 ) || ( del_user > USER_LIMIT ) )
                        {
                            continue;
                        }
                        /* 清除第 del_user 个客户连接使用的相关数据 */
                        epoll_ctl( epollfd, EPOLL_CTL_DEL,
                                   users[del_user].pipefd[0], 0 );
                        close( users[del_user].pipefd[0] );
                        users[del_user] = users[--user_count];
                        sub_process[users[del_user].pid] = del_user;
                    }
                    if( terminate && user_count == 0 )
                    {
                        stop_server = true;
                    }
                }
            }
        }
    }
}

```

```

        break;
    }
    case SIGTERM:
    case SIGINT:
    {
        /* 结束服务器程序 */
        printf( "kill all the child now\n" );
        if( user_count == 0 )
        {
            stop_server = true;
            break;
        }
        for( int i = 0; i < user_count; ++i )
        {
            int pid = users[i].pid;
            kill( pid, SIGTERM );
        }
        terminate = true;
        break;
    }
    default:
    {
        break;
    }
}
}

/* 某个子进程向父进程写入了数据 */
else if( events[i].events & EPOLLIN )
{
    int child = 0;
    /* 读取管道数据，child 变量记录了是哪个客户连接有数据到达 */
    ret = recv( sockfd, ( char* )&child, sizeof( child ), 0 );
    printf( "read data from child accross pipe\n" );
    if( ret == -1 )
    {
        continue;
    }
    else if( ret == 0 )
    {
        continue;
    }
    else
    {
        /* 向除负责处理第 child 个客户连接的子进程之外的其他子进程发送消息，通知它们有客户数据要写 */
        for( int j = 0; j < user_count; ++j )
        {
            if( users[j].pipefd[0] != sockfd )
            {

```

```

        printf( "send data to child accross pipe\n" );
        send( users[j].pipefd[0], ( char* )&child,
               sizeof( child ), 0 );
    }
}
}

del_resource();
return 0;
}

```

上面的代码有两点需要注意：

- 虽然我们使用了共享内存，但每个子进程都只会往自己所处理的客户连接所对应的那一部分读缓存中写入数据，所以我们使用共享内存的目的只是为了“共享读”。因此，每个子进程在使用共享内存的时候都无须加锁。这样做符合“聊天室服务器”的应用场景，同时提高了程序性能。
- 我们的服务器程序在启动的时候给数组 users 分配了足够多的空间，使得它可以存储所有可能的客户连接的相关数据。同样，我们一次性给数组 sub_process 分配的空间也足以存储所有可能的子进程的相关数据。这是牺牲空间换取时间的又一例子。

13.7 消息队列

消息队列是在两个进程之间传递二进制块数据的一种简单有效的方式。每个数据块都有一个特定的类型，接收方可以根据类型来有选择地接收数据，而不一定像管道和命名管道那样必须以先进先出的方式接收数据。

Linux 消息队列的 API 都定义在 sys/msg.h 头文件中，包括 4 个系统调用：msgget、msgsnd、msgrcv 和 msgctl。我们将依次讨论之。

13.7.1 msgget 系统调用

msgget 系统调用创建一个消息队列，或者获取一个已有的消息队列。其定义如下：

```
#include <sys/msg.h>
int msgget( key_t key, int msgflg );
```

和 semget 系统调用一样，key 参数是一个键值，用来标识一个全局唯一的消息队列。

msgflg 参数的使用和含义与 semget 系统调用的 sem_flags 参数相同。

msgget 成功时返回一个正整数值，它是消息队列的标识符。msgget 失败时返回 -1，并设置 errno。

如果 msgget 用于创建消息队列，则与之关联的内核数据结构 msqid_ds 将被创建并初始化。msqid_ds 结构体的定义如下：

```

struct msqid_ds
{
    struct ipc_perm msg_perm;           /* 消息队列的操作权限 */
    time_t msg_stime;                 /* 最后一次调用 msgsnd 的时间 */
    time_t msg_rtime;                 /* 最后一次调用 msgrcv 的时间 */
    time_t msg_ctime;                 /* 最后一次被修改的时间 */
    unsigned long __msg_cbytes;        /* 消息队列中已有的字节数 */
    msgqnum_t msg_qnum;               /* 消息队列中已有的消息数 */
    msglen_t msg_qbytes;              /* 消息队列允许的最大字节数 */
    pid_t msg_lspid;                  /* 最后执行 msgsnd 的进程的 PID */
    pid_t msg_lrpid;                  /* 最后执行 msgrcv 的进程的 PID */
};

```

13.7.2 msgsnd 系统调用

msgsnd 系统调用把一条消息添加到消息队列中。其定义如下：

```
#include <sys/msg.h>
int msgsnd( int msqid, const void* msg_ptr, size_t msg_sz, int msgflg );
```

msqid 参数是由 msgget 调用返回的消息队列标识符。

msg_ptr 参数指向一个准备发送的消息，消息必须被定义为如下类型：

```

struct msgbuf
{
    long mtype; /* 消息类型 */
    char mtext[512]; /* 消息数据 */
};
```

其中，mtype 成员指定消息的类型，它必须是一个正整数。mtext 是消息数据。msg_sz 参数是消息的数据部分（mtext）的长度。这个长度可以为 0，表示没有消息数据。

msgflg 参数控制 msgsnd 的行为。它通常仅支持 IPC_NOWAIT 标志，即以非阻塞的方式发送消息。默认情况下，发送消息时如果消息队列满了，则 msgsnd 将阻塞。若 IPC_NOWAIT 标志被指定，则 msgsnd 将立即返回并设置 errno 为 EAGAIN。

处于阻塞状态的 msgsnd 调用可能被如下两种异常情况所中断：

- 消息队列被移除。此时 msgsnd 调用将立即返回并设置 errno 为 EIDRM。
- 程序接收到信号。此时 msgsnd 调用将立即返回并设置 errno 为 EINTR。

msgsnd 成功时返回 0，失败则返回 -1 并设置 errno。msgsnd 成功时将修改内核数据结构 msqid_ds 的部分字段，如下所示：

- 将 msg_qnum 加 1。
- 将 msg_lspid 设置为调用进程的 PID。
- 将 msg_stime 设置为当前的时间。

13.7.3 msgrcv 系统调用

msgrcv 系统调用从消息队列中获取消息。其定义如下：

```
#include <sys/msg.h>
int msgrecv( int msqid, void* msg_ptr, size_t msg_sz, long int msgtype, int msgflg );
```

msqid 参数是由 msgget 调用返回的消息队列标识符。

msg_ptr 参数用于存储接收的消息，msg_sz 参数指的是消息数据部分的长度。

msgtype 参数指定接收何种类型的消息。我们可以使用如下几种方式来指定消息类型：

- msgtype 等于 0。读取消息队列中的第一个消息。

- msgtype 大于 0。读取消息队列中第一个类型为 msgtype 的消息（除非指定了标志 MSG_EXCEPT，见后文）。

- msgtype 小于 0。读取消息队列中第一个类型值比 msgtype 的绝对值小的消息。

参数 msgflg 控制 msgrecv 函数的行为。它可以是如下一些标志的按位或：

- IPC_NOWAIT。如果消息队列中没有消息，则 msgrecv 调用立即返回并设置 errno 为 ENOMSG。

- MSG_EXCEPT。如果 msgtype 大于 0，则接收消息队列中第一个非 msgtype 类型的消息。

- MSG_NOERROR。如果消息数据部分的长度超过了 msg_sz，就将它截断。

处于阻塞状态的 msgrecv 调用还可能被如下两种异常情况所中断：

- 消息队列被移除。此时 msgrecv 调用将立即返回并设置 errno 为 EIDRM。

- 程序接收到信号。此时 msgrecv 调用将立即返回并设置 errno 为 EINTR。

msgrecv 成功时返回 0，失败则返回 -1 并设置 errno。msgrecv 成功时将修改内核数据结构 msqid_ds 的部分字段，如下所示：

- 将 msg_qnum 减 1。

- 将 msg_lpid 设置为调用进程的 PID。

- 将 msg_rtime 设置为当前的时间。

13.7.4 msgctl 系统调用

msgctl 系统调用控制消息队列的某些属性。其定义如下：

```
#include <sys/msg.h>
int msgctl( int msqid, int command, struct msqid_ds* buf );
```

msqid 参数是由 msgget 调用返回的共享内存标识符。command 参数指定要执行的命令。msgctl 支持的所有命令如表 13-4 所示。

表 13-4 msgctl 支持的命令

命 令	含 义	msgctl 成功时的返回值
IPC_STAT	将消息队列关联的内核数据结构复制到 buf (第 3 个参数, 下同) 中	0
IPC_SET	将 buf 中的部分成员复制到消息队列关联的内核数据结构中, 同时内核数据中的 msqid_ds.msg_ctime 被更新	0

(续)

命 令	含 义	msgctl 成功时的返回值
IPC_RMID	立即移除消息队列，唤醒所有等待读消息和写消息的进程（这些调用立即返回并设置 errno 为 EIDRM）	0
IPC_INFO	获取系统消息队列资源配置信息，将结果存储在 buf 中。应用程序需要将 buf 转换成 msginfo 结构体类型来读取这些系统信息。msginfo 结构体与 seminfo 类似，这里不再赘述	内核消息队列信息数组中已经被使用的项的最大索引值
MSG_INFO	与 IPC_INFO 类似，不过返回的是已经分配的消息队列占用的资源信息	同 IPC_INFO
MSG_STAT	与 IPC_STAT 类似，不过此时 msqid 参数不是用来表示消息队列标识符，而是内核消息队列信息数组的索引（每个消息队列的信息都是该数组中的一项）	内核消息队列信息数组中索引值为 msqid 的消息队列的标识符

msgctl 成功时的返回值取决于 command 参数，如表 13-4 所示。msgctl 函数失败时返回 -1 并设置 errno。

13.8 IPC 命令

上述 3 种 System V IPC 进程间通信方式都使用一个全局唯一的键值 (key) 来描述一个共享资源。当程序调用 semget、shmget 或者 msgget 时，就创建了这些共享资源的一个实例。Linux 提供了 ipcs 命令，以观察当前系统上拥有哪些共享资源实例。比如在测试机器 Kongming20 上执行 ipcs 命令：

```
$ sudo ipcs
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
0x00000000 196608      apache      600          1
0x00000000 229377      apache      600          1
0x00000000 262146      apache      600          1
0x00000000 294915      apache      600          1
0x00000000 327684      apache      600          1
0x00000000 360453      apache      600          1
0x00000000 393222      apache      600          1

----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
```

输出结果分段显示了系统拥有的共享内存、信号量和消息队列资源。可见，该系统目前尚未使用任何共享内存和消息队列，却分配了一组键值为 0 (IPC_PRIVATE) 的信号量。这些信号量的所有者是 apache，因此它们是由 httpd 服务器程序创建的。其中标识符为 393222 的信号量正是我们在 13.5.5 小节讨论的那个用于在 httpd 各个子进程之间同步 epoll_wait 使用权的信号量。

此外，我们可以使用 ipcrm 命令来删除遗留在系统中的共享资源。

13.9 在进程间传递文件描述符

由于 fork 调用之后，父进程中打开的文件描述符在子进程中仍然保持打开，所以文件描述符可以很方便地从父进程传递到子进程。需要注意的是，传递一个文件描述符并不是传递一个文件描述符的值，而是要在接收进程中创建一个新的文件描述符，并且该文件描述符和发送进程中被传递的文件描述符指向内核中相同的文件表项。

那么如何把子进程中打开的文件描述符传递给父进程呢？或者更通俗地说，如何在两个不相干的进程之间传递文件描述符呢？在 Linux 下，我们可以利用 UNIX 域 socket 在进程间传递特殊的辅助数据，以实现文件描述符的传递^[2]。代码清单 13-5 给出了一个实例，它在子进程中打开一个文件描述符，然后将它传递给父进程，父进程则通过读取该文件描述符来获得文件的内容。

代码清单 13-5 在进程间传递文件描述符

```
#include <sys/socket.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

static const int CONTROL_LEN = CMSG_LEN( sizeof(int) );
/* 发送文件描述符，fd 参数是用来传递信息的 UNIX 域 socket，fd_to_send 参数是待发送的文件描述符 */
void send_fd( int fd, int fd_to_send )
{
    struct iovec iov[1];
    struct msghdr msg;
    char buf[0];

    iov[0].iov_base = buf;
    iov[0].iov_len = 1;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

    cmsghdr cm;
    cm.cmsg_len = CONTROL_LEN;
    cm.cmsg_level = SOL_SOCKET;
    cm.cmsg_type = SCM_RIGHTS;
    *(int *)CMSG_DATA( &cm ) = fd_to_send;
    msg.msg_control = &cm; /* 设置辅助数据 */
    msg.msg_controllen = CONTROL_LEN;

    sendmsg( fd, &msg, 0 );
}
```

```

/* 接收目标文件描述符 */
int recv_fd( int fd )
{
    struct iovec iov[1];
    struct msghdr msg;
    char buf[0];

    iov[0].iov_base = buf;
    iov[0].iov_len = 1;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

    cmsghdr cm;
    msg.msg_control = &cm;
    msg.msg_controllen = CONTROL_LEN;

    recvmsg( fd, &msg, 0 );

    int fd_to_read = *(int *)CMMSG_DATA( &cm );
    return fd_to_read;
}

int main()
{
    int pipefd[2];
    int fd_to_pass = 0;
    /* 创建父、子进程间的管道，文件描述符 pipefd[0] 和 pipefd[1] 都是 UNIX 域 socket */
    int ret = socketpair( PF_UNIX, SOCK_DGRAM, 0, pipefd );
    assert( ret != -1 );

    pid_t pid = fork();
    assert( pid >= 0 );

    if ( pid == 0 )
    {
        close( pipefd[0] );
        fd_to_pass = open( "test.txt", O_RDWR, 0666 );
        /* 子进程通过管道将文件描述符发送到父进程。如果文件 test.txt 打开失败，则子进程将标准输入文件描述符发送到父进程 */
        send_fd( pipefd[1], ( fd_to_pass > 0 ) ? fd_to_pass : 0 );
        close( fd_to_pass );
        exit( 0 );
    }

    close( pipefd[1] );
    fd_to_pass = recv_fd( pipefd[0] ); /* 父进程从管道接收目标文件描述符 */
    char buf[1024];
    memset( buf, '\0', 1024 );
    read( fd_to_pass, buf, 1024 ); /* 读目标文件描述符，以验证其有效性 */
    printf( "I got fd %d and data %s\n", fd_to_pass, buf );
    close( fd_to_pass );
}

```

第 14 章 多线程编程

早期 Linux 不支持线程，直到 1996 年，Xavier Leroy 等人才开发出第一个基本符合 POSIX 标准的线程库 LinuxThreads。但 LinuxThreads 效率低而且问题很多。自内核 2.6 开始，Linux 才真正提供内核级的线程支持，并有两个组织致力于编写新的线程库：NGPT（Next Generation POSIX Threads）和 NPTL（Native POSIX Thread Library）。不过前者在 2003 年就放弃了，因此新的线程库就称为 NPTL。NPTL 比 LinuxThreads 效率高，且更符合 POSIX 规范，所以它已经成为 glibc 的一部分。本书所有线程相关的例程使用的线程库都是 NPTL。

本章要讨论的线程相关的内容都属于 POSIX 线程（简称 pthread）标准，而不局限于 NPTL 实现，具体包括：

- 创建线程和结束线程。
- 读取和设置线程属性。
- POSIX 线程同步方式：POSIX 信号量、互斥锁和条件变量。

在本章的最后，我们还将介绍在 Linux 环境下，库函数、进程、信号与多线程程序之间的相互影响。

14.1 Linux 线程概述

14.1.1 线程模型

线程是程序中完成一个独立任务的完整执行序列，即一个可调度的实体。根据运行环境和调度者的身份，线程可分为内核线程和用户线程。内核线程，在有的系统上也称为 LWP（Light Weight Process，轻量级进程），运行在内核空间，由内核来调度；用户线程运行在用户空间，由线程库来调度。当进程的一个内核线程获得 CPU 的使用权时，它就加载并运行一个用户线程。可见，内核线程相当于用户线程运行的“容器”。一个进程可以拥有 M 个内核线程和 N 个用户线程，其中 $M \leq N$ 。并且在一个系统的所有进程中， M 和 N 的比值都是固定的。按照 $M:N$ 的取值，线程的实现方式可分为三种模式：完全在用户空间实现、完全由内核调度和双层调度（two level scheduler）。

完全在用户空间实现的线程无须内核的支持，内核甚至根本不知道这些线程的存在。线程库负责管理所有执行线程，比如线程的优先级、时间片等。线程库利用 longjmp 来切换线程的执行，使它们看起来像是“并发”执行的。但实际上内核仍然是把整个进程作为最小单位来调度的。换句话说，一个进程的所有执行线程共享该进程的时间片，它们对外表现出相

同的优先级。因此，对这种实现方式而言， $N=1$ ，即 M 个用户空间线程对应 1 个内核线程，而该内核线程实际上就是进程本身。完全在用户空间实现的线程的优点是：创建和调度线程都无须内核的干预，因此速度相当快。并且由于它不占用额外的内核资源，所以即使一个进程创建了很多线程，也不会对系统性能造成明显的影响。其缺点是：对于多处理器系统，一个进程的多个线程无法运行在不同的 CPU 上，因为内核是按照其最小调度单位来分配 CPU 的。此外，线程的优先级只对同一个进程中的线程有效，比较不同进程中的线程的优先级没有意义。早期的伯克利 UNIX 线程就是采用这种方式实现的。

完全由内核调度的模式将创建、调度线程的任务都交给了内核，运行在用户空间的线程库无须执行管理任务，这与完全在用户空间实现的线程恰恰相反。二者的优缺点也正好互换。较早的 Linux 内核对内核线程的控制能力有限，线程库通常还要提供额外的控制能力，尤其是线程同步机制，不过现代 Linux 内核已经大大增强了对线程的支持。完全由内核调度的这种线程实现方式满足 $M:N=1:1$ ，即 1 个用户空间线程被映射为 1 个内核线程。

双层调度模式是前两种实现模式的混合体：内核调度 M 个内核线程，线程库调度 N 个用户线程。这种线程实现方式结合了前两种方式的优点：不但不会消耗过多的内核资源，而且线程切换速度也较快，同时它可以充分利用多处理器的优势。

14.1.2 Linux 线程库

Linux 上两个最有名的线程库是 LinuxThreads 和 NPTL，它们都是采用 1:1 的方式实现的。由于 LinuxThreads 在开发的时候，Linux 内核对线程的支持还非常有限，所以其可用性、稳定性以及 POSIX 兼容性都远远不及 NPTL。现代 Linux 上默认使用的线程库是 NPTL。用户可以使用如下命令来查看当前系统上所使用的线程库：

```
$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.14.90
```

LinuxThreads 线程库的内核线程是用 clone 系统调用创建的进程模拟的。clone 系统调用和 fork 系统调用的作用类似：创建调用进程的子进程。不过我们可以为 clone 系统调用指定 CLONE_THREAD 标志，这种情况下它创建的子进程与调用进程共享相同的虚拟地址空间、文件描述符和信号处理函数，这些都是线程的特点。不过，用进程来模拟内核线程会导致很多语义问题，比如：

- 每个线程拥有不同的 PID，因此不符合 POSIX 规范。
- Linux 信号处理本来是基于进程的，但现在一个进程内部的所有线程都能而且必须处理信号。
- 用户 ID、组 ID 对一个进程中的不同线程来说可能是不一样的。
- 程序产生的核心转储文件不会包含所有线程的信息，而只包含产生该核心转储文件的线程的信息。
- 由于每个线程都是一个进程，因此系统允许的最大进程数也就是最大线程数。

LinuxThreads 线程库一个有名的特性是所谓的管理线程。它是进程中专门用于管理其他

工作线程的线程。其作用包括：

- 系统发送给进程的终止信号先由管理线程接收，管理线程再给其他工作线程发送同样的信号以终止它们。
- 当终止工作线程或者工作线程主动退出时，管理线程必须等待它们结束，以避免僵尸进程。
- 如果主线程先于其他工作线程退出，则管理线程将阻塞它，直到所有其他工作线程都结束之后才唤醒它。
- 回收每个线程堆栈使用的内存。

管理线程的引入，增加了额外的系统开销。并且由于它只能运行在一个 CPU 上，所以 LinuxThreads 线程库也不能充分利用多处理器系统的优势。

要解决 LinuxThreads 线程库的一系列问题，不仅需要改进线程库，最主要的是需要内核提供更完善的线程支持。因此，Linux 内核从 2.6 版本开始，提供了真正的内核线程。新的 NPTL 线程库也应运而生。相比 LinuxThreads，NPTL 的主要优势在于：

- 内核线程不再是一个进程，因此避免了很多用进程模拟内核线程导致的语义问题。
- 摒弃了管理线程，终止线程、回收线程堆栈等工作都可以由内核来完成。
- 由于不存在管理线程，所以一个进程的线程可以运行在不同的 CPU 上，从而充分利用了多处理器系统的优势。
- 线程的同步由内核来完成。隶属于不同进程的线程之间也能共享互斥锁，因此可实现跨进程的线程同步。

14.2 创建线程和结束线程

下面我们讨论创建和结束线程的基础 API。Linux 系统上，它们都定义在 `pthread.h` 头文件中。

1. `pthread_create`

创建一个线程的函数是 `pthread_create`。其定义如下：

```
#include <pthread.h>
int pthread_create( pthread_t* thread, const pthread_attr_t* attr,
                    void* (*start_routine)( void* ), void* arg );
```

`thread` 参数是新线程的标识符，后续 `pthread_*` 函数通过它来引用新线程。其类型 `pthread_t` 的定义如下：

```
#include <bits/ pthreadtypes.h>
typedef unsigned long int pthread_t;
```

可见，`pthread_t` 是一个整型类型。实际上，Linux 上几乎所有的资源标识符都是一个整型数，比如 socket、各种 System V IPC 标识符等。

`attr` 参数用于设置新线程的属性。给它传递 `NULL` 表示使用默认线程属性。线程拥有众

多属性，我们将在后面详细讨论之。`start_routine` 和 `arg` 参数分别指定新线程将运行的函数及其参数。

`pthread_create` 成功时返回 0，失败时返回错误码。一个用户可以打开的线程数量不能超过 `RLIMIT_NPROC` 软资源限制（见表 7-1）。此外，系统上所有用户能创建的线程总数也不得超过 `/proc/sys/kernel/threads-max` 内核参数所定义的值。

2. `pthread_exit`

线程一旦被创建好，内核就可以调度内核线程来执行 `start_routine` 函数指针所指向的函数了。线程函数在结束时最好调用如下函数，以确保安全、干净地退出：

```
#include <pthread.h>
void pthread_exit( void* retval );
```

`pthread_exit` 函数通过 `retval` 参数向线程的回收者传递其退出信息。它执行完之后不会返回到调用者，而且永远不会失败。

3. `pthread_join`

一个进程中的所有线程都可以调用 `pthread_join` 函数来回收其他线程（前提是目标线程是可回收的，见后文），即等待其他线程结束，这类似于回收进程的 `wait` 和 `waitpid` 系统调用。`pthread_join` 的定义如下：

```
#include <pthread.h>
int pthread_join( pthread_t thread, void** retval );
```

`thread` 参数是目标线程的标识符，`retval` 参数则是目标线程返回的退出信息。该函数会一直阻塞，直到被回收的线程结束为止。该函数成功时返回 0，失败则返回错误码。可能的错误码如表 14-1 所示。

表 14-1 `pthread_join` 函数可能引发的错误码

错误码	描述
EDEADLK	可能引起死锁。比如两个线程互相针对对方调用 <code>pthread_join</code> ，或者线程对自身调用 <code>pthread_join</code>
EINVAL	目标线程是不可回收的，或者已经有其他线程在回收该目标线程
ESRCH	目标线程不存在

4. `pthread_cancel`

有时候我们希望异常终止一个线程，即取消线程，它是通过如下函数实现的：

```
#include <pthread.h>
int pthread_cancel( pthread_t thread );
```

`thread` 参数是目标线程的标识符。该函数成功时返回 0，失败则返回错误码。不过，接收到取消请求的目标线程可以决定是否允许被取消以及如何取消，这分别由如下两个函数完成：

```
#include <pthread.h>
int pthread_setcancelstate( int state, int *oldstate );
int pthread_setcanceltype( int type, int *oldtype );
```

这两个函数的第一个参数分别用于设置线程的取消状态（是否允许取消）和取消类型（如何取消），第二个参数则分别记录线程原来的取消状态和取消类型。state参数有两个可选值：

- PTHREAD_CANCEL_ENABLE，允许线程被取消。它是线程被创建时的默认取消状态。
- PTHREAD_CANCEL_DISABLE，禁止线程被取消。这种情况下，如果一个线程收到取消请求，则它会将请求挂起，直到该线程允许被取消。

type参数也有两个可选值：

- PTHREAD_CANCEL_ASYNCHRONOUS，线程随时都可以被取消。它将使得接收到取消请求的目标线程立即采取行动。
- PTHREAD_CANCEL_DEFERRED，允许目标线程推迟行动，直到它调用了下面几个所谓的取消点函数中的一个：pthread_join、pthread_testcancel、pthread_cond_wait、pthread_cond_timedwait、sem_wait 和 sigwait。根据 POSIX 标准，其他可能阻塞的系统调用，比如 read、wait，也可以成为取消点。不过为了安全起见，我们最好在可能被取消的代码中调用 pthread_testcancel 函数以设置取消点。

pthread_setcancelstate 和 pthread_setcanceltype 成功时返回 0，失败则返回错误码。

14.3 线程属性

pthread_attr_t 结构体定义了一套完整的线程属性，如下所示：

```
#include <bits/pthreadtypes.h>
#define __SIZEOF_PTHREAD_ATTR_T 36
typedef union
{
    char __size[__SIZEOF_PTHREAD_ATTR_T];
    long int __align;
} pthread_attr_t;
```

可见，各种线程属性全部包含在一个字符数组中。线程库定义了一系列函数来操作 pthread_attr_t 类型的变量，以方便我们获取和设置线程属性。这些函数包括：

```
#include <pthread.h>
/* 初始化线程属性对象 */
int pthread_attr_init ( pthread_attr_t* attr );
/* 销毁线程属性对象。被销毁的线程属性对象只有再次初始化之后才能继续使用 */
int pthread_attr_destroy ( pthread_attr_t* attr );
/* 下面这些函数用于获取和设置线程属性对象的某个属性 */
int pthread_attr_getdetachstate ( const pthread_attr_t* attr, int* detachstate );
int pthread_attr_setdetachstate ( pthread_attr_t* attr, int detachstate );
int pthread_attr_getstackaddr (const pthread_attr_t* attr, void ** stackaddr );
int pthread_attr_setstackaddr ( pthread_attr_t* attr, void* stackaddr );
int pthread_attr_getstacksize ( const pthread_attr_t* attr, size_t* stacksize );
int pthread_attr_setstacksize ( pthread_attr_t* attr, size_t stacksize );
int pthread_attr_getstack ( const pthread_attr_t* attr, void** stackaddr,
                           size_t* stacksize);
```

```

int pthread_attr_setstack ( pthread_attr_t* attr, void* stackaddr,
                           size_t stacksize );
int pthread_attr_getguardsize ( const pthread_attr_t * __attr, size_t* guardsize );
int pthread_attr_setguardsize (pthread_attr_t* attr, size_t guardsize );
int pthread_attr_getschedparam ( const pthread_attr_t* attr, struct
                                 sched_param* param );
int pthread_attr_setschedparam ( pthread_attr_t* attr, const struct
                                 sched_param* param );
int pthread_attr_getschedpolicy ( const pthread_attr_t* attr, int* policy );
int pthread_attr_setschedpolicy ( pthread_attr_t* attr, int policy );
int pthread_attr_getinheritsched ( const pthread_attr_t* attr, int* inherit );
int pthread_attr_setinheritsched (pthread_attr_t* attr, int inherit );
int pthread_attr_getscope ( const pthread_attr_t* attr, int* scope );
int pthread_attr_setscope ( pthread_attr_t* attr, int scope );

```

下面我们详细讨论每个线程属性的含义：

- detachstate，线程的脱离状态。它有 PTHREAD_CREATE_JOINABLE 和 PTHREAD_CREATE_DETACH 两个可选值。前者指定线程是可以被回收的，后者使调用线程脱离与进程中其他线程的同步。脱离了与其他线程同步的线程称为“脱离线程”。脱离线程在退出时将自行释放其占用的系统资源。线程创建时该属性的默认值是 PTHREAD_CREATE_JOINABLE。此外，我们也可以使用 `pthread_detach` 函数直接将线程设置为脱离线程。
- stackaddr 和 stacksize，线程堆栈的起始地址和大小。一般来说，我们不需要自己来管理线程堆栈，因为 Linux 默认为每个线程分配了足够的堆栈空间（一般是 8 MB）。我们可以使用 `ulimit -s` 命令来查看或修改这个默认值。
- guardsize，保护区域大小。如果 `guardsize` 大于 0，则系统创建线程的时候会在其堆栈的尾部额外分配 `guardsize` 字节的空间，作为保护堆栈不被错误地覆盖的区域。如果 `guardsize` 等于 0，则系统不为新创建的线程设置堆栈保护区。如果使用者通过 `pthread_attr_setstackaddr` 或 `pthread_attr_setstack` 函数手动设置线程的堆栈，则 `guardsize` 属性将被忽略。
- schedparam，线程调度参数。其类型是 `sched_param` 结构体。该结构体目前还只有一个整型类型的成员——`sched_priority`，该成员表示线程的运行优先级。
- schedpolicy，线程调度策略。该属性有 SCHED_FIFO、SCHED_RR 和 SCHED_OTHER 三个可选值，其中 SCHED_OTHER 是默认值。SCHED_RR 表示采用轮转算法（round-robin）调度，SCHED_FIFO 表示使用先进先出的方法调度，这两种调度方法都具备实时调度功能，但只能用于以超级用户身份运行的进程。
- inheritsched，是否继承调用线程的调度属性。该属性有 PTHREAD_INHERIT_SCHED 和 PTHREAD_EXPLICIT_SCHED 两个可选值。前者表示新线程沿用其创建者的线程调度参数，这种情况下再设置新线程的调度参数属性将没有任何效果。后者表示调用者要明确地指定新线程的调度参数。
- scope，线程间竞争 CPU 的范围，即线程优先级的有效范围。POSIX 标准定义了该属

性的 PTHREAD_SCOPE_SYSTEM 和 PTHREAD_SCOPE_PROCESS 两个可选值，前者表示目标线程与系统中所有线程一起竞争 CPU 的使用，后者表示目标线程仅与其他隶属于同一进程的线程竞争 CPU 的使用。目前 Linux 只支持 PTHREAD_SCOPE_SYSTEM 这一种取值。

14.4 POSIX 信号量

和多进程程序一样，多线程程序也必须考虑同步问题。`pthread_join` 可以看作一种简单的线程同步方式，不过很显然，它无法高效地实现复杂的同步需求，比如控制对共享资源的独占式访问，又抑或是在某个条件满足之后唤醒一个线程。接下来我们讨论 3 种专门用于线程同步的机制：POSIX 信号量、互斥量和条件变量。

在 Linux 上，信号量 API 有两组。一组是第 13 章讨论过的 System V IPC 信号量，另外一组是我们现在要讨论的 POSIX 信号量。这两组接口很相似，但不保证能互换。由于这两种信号量的语义完全相同，因此我们不再赘述信号量的原理。

POSIX 信号量函数的名字都以 `sem_` 开头，并不像大多数线程函数那样以 `pthread_` 开头。常用的 POSIX 信号量函数是下面 5 个：

```
#include < semaphore.h>
int sem_init( sem_t* sem, int pshared, unsigned int value );
int sem_destroy( sem_t* sem );
int sem_wait( sem_t* sem );
int sem_trywait( sem_t* sem );
int sem_post( sem_t* sem );
```

这些函数的第一个参数 `sem` 指向被操作的信号量。

`sem_init` 函数用于初始化一个未命名的信号量（POSIX 信号量 API 支持命名信号量，不过本书不讨论它）。`pshared` 参数指定信号量的类型。如果其值为 0，就表示这个信号量是当前进程的局部信号量，否则该信号量就可以在多个进程之间共享。`value` 参数指定信号量的初始值。此外，初始化一个已经被初始化的信号量将导致不可预期的结果。

`sem_destroy` 函数用于销毁信号量，以释放其占用的内核资源。如果销毁一个正被其他线程等待的信号量，则将导致不可预期的结果。

`sem_wait` 函数以原子操作的方式将信号量的值减 1。如果信号量的值为 0，则 `sem_wait` 将被阻塞，直到这个信号量具有非 0 值。

`sem_trywait` 与 `sem_wait` 函数相似，不过它始终立即返回，而不论被操作的信号量是否具有非 0 值，相当于 `sem_wait` 的非阻塞版本。当信号量的值非 0 时，`sem_trywait` 对信号量执行减 1 操作。当信号量的值为 0 时，它将返回 -1 并设置 `errno` 为 `EAGAIN`。

`sem_post` 函数以原子操作的方式将信号量的值加 1。当信号量的值大于 0 时，其他正在调用 `sem_wait` 等待信号量的线程将被唤醒。

上面这些函数成功时返回 0，失败则返回 -1 并设置 `errno`。

14.5 互斥锁

互斥锁（也称互斥量）可以用于保护关键代码段，以确保其独占式的访问，这有点像一个二进制信号量（见 13.5.1 小节）。当进入关键代码段时，我们需要获得互斥锁并将其加锁，这等价于二进制信号量的 P 操作；当离开关键代码段时，我们需要对互斥锁解锁，以唤醒其他等待该互斥锁的线程，这等价于二进制信号量的 V 操作。

14.5.1 互斥锁基础 API

POSIX 互斥锁的相关函数主要有如下 5 个：

```
#include <pthread.h>
int pthread_mutex_init( pthread_mutex_t* mutex, const
                        pthread_mutexattr_t* mutexattr );
int pthread_mutex_destroy( pthread_mutex_t* mutex );
int pthread_mutex_lock( pthread_mutex_t* mutex );
int pthread_mutex_trylock( pthread_mutex_t* mutex );
int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

这些函数的第一个参数 mutex 指向要操作的目标互斥锁，互斥锁的类型是 `pthread_mutex_t` 结构体。

`pthread_mutex_init` 函数用于初始化互斥锁。`mutexattr` 参数指定互斥锁的属性。如果将它设置为 `NULL`，则表示使用默认属性。我们将在下一小节讨论互斥锁的属性。除了这个函数外，我们还可以使用如下方式来初始化一个互斥锁：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

宏 `PTHREAD_MUTEX_INITIALIZER` 实际上只是把互斥锁的各个字段都初始化为 0。

`pthread_mutex_destroy` 函数用于销毁互斥锁，以释放其占用的内核资源。销毁一个已经加锁的互斥锁将导致不可预期的后果。

`pthread_mutex_lock` 函数以原子操作的方式给一个互斥锁加锁。如果目标互斥锁已经被锁上，则 `pthread_mutex_lock` 调用将阻塞，直到该互斥锁的占有者将其解锁。

`pthread_mutex_trylock` 与 `pthread_mutex_lock` 函数类似，不过它始终立即返回，而不论被操作的互斥锁是否已经被加锁，相当于 `pthread_mutex_lock` 的非阻塞版本。当目标互斥锁未被加锁时，`pthread_mutex_trylock` 对互斥锁执行加锁操作。当互斥锁已经被加锁时，`pthread_mutex_trylock` 将返回错误码 `EBUSY`。需要注意的是，这里讨论的 `pthread_mutex_lock` 和 `pthread_mutex_trylock` 的行为是针对普通锁而言的。后面我们将看到，对于其他类型的锁而言，这两个加锁函数会有不同的行为。

`pthread_mutex_unlock` 函数以原子操作的方式给一个互斥锁解锁。如果此时有其他线程正在等待这个互斥锁，则这些线程中的某一个将获得它。

上面这些函数成功时返回 0，失败则返回错误码。

14.5.2 互斥锁属性

`pthread_mutexattr_t` 结构体定义了一套完整的互斥锁属性。线程库提供了一系列函数来操作 `pthread_mutexattr_t` 类型的变量，以方便我们获取和设置互斥锁属性。这里我们列出其中一些主要的函数：

```
#include <pthread.h>
/* 初始化互斥锁属性对象 */
int pthread_mutexattr_init( pthread_mutexattr_t* attr );
/* 销毁互斥锁属性对象 */
int pthread_mutexattr_destroy( pthread_mutexattr_t* attr );
/* 获取和设置互斥锁的 pshared 属性 */
int pthread_mutexattr_getpshared( const pthread_mutexattr_t* attr, int* pshared );
int pthread_mutexattr_setpshared ( pthread_mutexattr_t* attr, int pshared );
/* 获取和设置互斥锁的 type 属性 */
int pthread_mutexattr_gettype( const pthread_mutexattr_t* attr, int* type);
int pthread_mutexattr_settype( pthread_mutexattr_t* attr, int type );
```

本书只讨论互斥锁的两种常用属性：`pshared` 和 `type`。`互斥锁属性 pshared` 指定是否允许跨进程共享互斥锁，其可选值有两个：

- `PTHREAD_PROCESS_SHARED`。互斥锁可以被跨进程共享。
- `PTHREAD_PROCESS_PRIVATE`。互斥锁只能被和锁的初始化线程隶属于同一个进程的线程共享。

互斥锁属性 `type` 指定互斥锁的类型。Linux 支持如下 4 种类型的互斥锁：

- `PTHREAD_MUTEX_NORMAL`，普通锁。这是互斥锁默认的类型。当一个线程对一个普通锁加锁以后，其余请求该锁的线程将形成一个等待队列，并在该锁解锁后按优先级获得它。这种锁类型保证了资源分配的公平性。但这种锁也很容易引发问题：一个线程如果对一个已经加锁的普通锁再次加锁，将引发死锁；对一个已经被其他线程加锁的普通锁解锁，或者对一个已经解锁的普通锁再次解锁，将导致不可预期的后果。
- `PTHREAD_MUTEX_ERRORCHECK`，检错锁。一个线程如果对一个已经加锁的检错锁再次加锁，则加锁操作返回 `EDEADLK`。对一个已经被其他线程加锁的检错锁解锁，或者对一个已经解锁的检错锁再次解锁，则解锁操作返回 `EPERM`。
- `PTHREAD_MUTEX_RECURSIVE`，嵌套锁。这种锁允许一个线程在释放锁之前多次对它加锁而不发生死锁。不过其他线程如果要获得这个锁，则当前锁的拥有者必须执行相应次数的解锁操作。对一个已经被其他线程加锁的嵌套锁解锁，或者对一个已经解锁的嵌套锁再次解锁，则解锁操作返回 `EPERM`。
- `PTHREAD_MUTEX_DEFAULT`，默认锁。一个线程如果对一个已经加锁的默认锁再次加锁，或者对一个已经被其他线程加锁的默认锁解锁，或者对一个已经解锁的默认锁再次解锁，将导致不可预期的后果。这种锁在实现的时候可能被映射为上面三种锁之一。

14.5.3 死锁举例

使用互斥锁的一个噩耗是死锁。死锁使得一个或多个线程被挂起而无法继续执行，而且这种情况还不容易被发现。前文提到，在一个线程中对一个已经加锁的普通锁再次加锁，将导致死锁。这种情况可能出现在设计得不够仔细的递归函数中。另外，如果两个线程按照不同的顺序来申请两个互斥锁，也容易产生死锁，如代码清单 14-1 所示。

代码清单 14-1 按不同顺序访问互斥锁导致死锁

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

int a = 0;
int b = 0;
pthread_mutex_t mutex_a;
pthread_mutex_t mutex_b;

void* another( void* arg )
{
    pthread_mutex_lock( &mutex_b );
    printf( "in child thread, got mutex b, waiting for mutex a\n" );
    sleep( 5 );
    ++b;
    pthread_mutex_lock( &mutex_a );
    b += a++;
    pthread_mutex_unlock( &mutex_a );
    pthread_mutex_unlock( &mutex_b );
    pthread_exit( NULL );
}

int main()
{
    pthread_t id;

    pthread_mutex_init( &mutex_a, NULL );
    pthread_mutex_init( &mutex_b, NULL );
    pthread_create( &id, NULL, another, NULL );

    pthread_mutex_lock( &mutex_a );
    printf( "in parent thread, got mutex a, waiting for mutex b\n" );
    sleep( 5 );
    ++a;
    pthread_mutex_lock( &mutex_b );
    a += b++;
    pthread_mutex_unlock( &mutex_b );
    pthread_mutex_unlock( &mutex_a );

    pthread_join( id, NULL );
    pthread_mutex_destroy( &mutex_a );
    pthread_mutex_destroy( &mutex_b );
}
```

```

    return 0;
}

```

代码清单 14-1 中，主线程试图先占有互斥锁 mutex_a，然后操作被该锁保护的变量 a，但操作完毕之后，主线程并没有立即释放互斥锁 mutex_a，而是又申请互斥锁 mutex_b，并在两个互斥锁的保护下，操作变量 a 和 b，最后才一起释放这两个互斥锁；与此同时，子线程则按照相反的顺序来申请互斥锁 mutex_a 和 mutex_b，并在两个锁的保护下操作变量 a 和 b。我们用 sleep 函数来模拟连续两次调用 pthread_mutex_lock 之间的时间差，以确保代码中的两个线程各自先占有一个互斥锁（主线程占有 mutex_a，子线程占有 mutex_b），然后等待另外一个互斥锁（主线程等待 mutex_b，子线程等待 mutex_a）。这样，两个线程就僵持住了，谁都不能继续往下执行，从而形成死锁。如果代码中不加入 sleep 函数，则这段代码或许总能成功地运行，从而为程序留下了一个潜在的 BUG。

14.6 条件变量

如果说互斥锁是用于同步线程对共享数据的访问的话，那么条件变量则是用于在线程之间同步共享数据的值。条件变量提供了一种线程间的通知机制：当某个共享数据达到某个值的时候，唤醒等待这个共享数据的线程。

条件变量的相关函数主要有如下 5 个：

```

#include <pthread.h>
int pthread_cond_init( pthread_cond_t* cond, const pthread_condattr_t* cond_attr);
int pthread_cond_destroy( pthread_cond_t* cond);
int pthread_cond_broadcast( pthread_cond_t* cond );
int pthread_cond_signal( pthread_cond_t* cond );
int pthread_cond_wait( pthread_cond_t* cond, pthread_mutex_t* mutex );

```

这些函数的第一个参数 cond 指向要操作的目标条件变量，条件变量的类型是 pthread_cond_t 结构体。

pthread_cond_init 函数用于初始化条件变量。cond_attr 参数指定条件变量的属性。如果将它设置为 NULL，则表示使用默认属性。条件变量的属性不多，而且和互斥锁的属性类型相似，所以我们不再赘述。除了 pthread_cond_init 函数外，我们还可以使用如下方式来初始化一个条件变量：

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

宏 PTHREAD_COND_INITIALIZER 实际上只是把条件变量的各个字段都初始化为 0。

pthread_cond_destroy 函数用于销毁条件变量，以释放其占用的内核资源。销毁一个正在被等待的条件变量将失败并返回 EBUSY。

pthread_cond_broadcast 函数以广播的方式唤醒所有等待目标条件变量的线程。pthread_cond_signal 函数用于唤醒一个等待目标条件变量的线程。至于哪个线程将被唤醒，则取决于线程的优先级和调度策略。有时候我们可能想唤醒一个指定的线程，但 pthread 没有对该需

求提供解决方法。不过我们可以间接地实现该需求：定义一个能够唯一表示目标线程的全局变量，在唤醒等待条件变量的线程前先设置该变量为目标线程，然后采用广播方式唤醒所有等待条件变量的线程，这些线程被唤醒后都检查该变量以判断被唤醒的是不是自己，如果是就开始执行后续代码，如果不是则返回继续等待。

`pthread_cond_wait` 函数用于等待目标条件变量。`mutex` 参数是用于保护条件变量的互斥锁，以确保 `pthread_cond_wait` 操作的原子性。在调用 `pthread_cond_wait` 前，必须确保互斥锁 `mutex` 已经加锁，否则将导致不可预期的结果。`pthread_cond_wait` 函数执行时，首先把调用线程放入条件变量的等待队列中，然后将互斥锁 `mutex` 解锁。可见，从 `pthread_cond_wait` 开始执行到其调用线程被放入条件变量的等待队列之间的这段时间内，`pthread_cond_signal` 和 `pthread_cond_broadcast` 等函数不会修改条件变量。换言之，`pthread_cond_wait` 函数不会错过目标条件变量的任何变化^[7]。当 `pthread_cond_wait` 函数成功返回时，互斥锁 `mutex` 将再次被锁上。

上面这些函数成功时返回 0，失败则返回错误码。

14.7 线程同步机制包装类

为了充分复用代码，同时由于后文的需要，我们将前面讨论的 3 种线程同步机制分别封装成 3 个类，实现在 `locker.h` 文件中，如代码清单 14-2 所示。

代码清单 14-2 `locker.h` 文件

```
#ifndef LOCKER_H
#define LOCKER_H

#include <exception>
#include <pthread.h>
#include <semaphore.h>
/* 封装信号量的类 */
class sem
{
public:
    /* 创建并初始化信号量 */
    sem()
    {
        if( sem_init( &m_sem, 0, 0 ) != 0 )
        {
            /* 构造函数没有返回值，可以通过抛出异常来报告错误 */
            throw std::exception();
        }
    }
    /* 销毁信号量 */
    ~sem()
    {
        sem_destroy( &m_sem );
    }
private:
    sem( const sem& );
    void operator=( const sem& );
};

#endif
```

```
/* 等待信号量 */
bool wait()
{
    return sem_wait( &m_sem ) == 0;
}
/* 增加信号量 */
bool post()
{
    return sem_post( &m_sem ) == 0;
}

private:
    sem_t m_sem;
};

/* 封装互斥锁的类 */
class locker
{
public:
    /* 创建并初始化互斥锁 */
    locker()
    {
        if( pthread_mutex_init( &m_mutex, NULL ) != 0 )
        {
            throw std::exception();
        }
    }
    /* 销毁互斥锁 */
    ~locker()
    {
        pthread_mutex_destroy( &m_mutex );
    }
    /* 获得互斥锁 */
    bool lock()
    {
        return pthread_mutex_lock( &m_mutex ) == 0;
    }
    /* 释放互斥锁 */
    bool unlock()
    {
        return pthread_mutex_unlock( &m_mutex ) == 0;
    }

private:
    pthread_mutex_t m_mutex;
};

/* 封装条件变量的类 */
class cond
{
public:
    /* 创建并初始化条件变量 */
    cond()
    {
        if( pthread_mutex_init( &m_mutex, NULL ) != 0 )
```

```

    {
        throw std::exception();
    }
    if ( pthread_cond_init( &m_cond, NULL ) != 0 )
    {
        /* 构造函数中一旦出现问题，就应该立即释放已经成功分配了的资源 */
        pthread_mutex_destroy( &m_mutex );
        throw std::exception();
    }
}
/* 销毁条件变量 */
~cond()
{
    pthread_mutex_destroy( &m_mutex );
    pthread_cond_destroy( &m_cond );
}
/* 等待条件变量 */
bool wait()
{
    int ret = 0;
    pthread_mutex_lock( &m_mutex );
    ret = pthread_cond_wait( &m_cond, &m_mutex );
    pthread_mutex_unlock( &m_mutex );
    return ret == 0;
}
/* 唤醒等待条件变量的线程 */
bool signal()
{
    return pthread_cond_signal( &m_cond ) == 0;
}

private:
    pthread_mutex_t m_mutex;
    pthread_cond_t m_cond;
};

#endif

```

14.8 多线程环境

14.8.1 可重入函数

如果一个函数能被多个线程同时调用且不发生竞态条件，则我们称它是线程安全的（thread safe），或者说它是可重入函数。Linux 库函数只有一小部分是不可重入的，比如 5.1.4 小节讨论的 `inet_ntoa` 函数，以及 5.12.2 小节讨论的 `getservbyname` 和 `getservbyport` 函数。关于 Linux 上不可重入的库函数的完整列表，请读者参考相关书籍，这里不再赘述。这些库函数之所以不可重入，主要是因为其内部使用了静态变量。不过 Linux 对很多不可重入的库函数提供了对应的可重入版本，这些可重入版本的函数名是在原函数名尾部加上 `_r`。比如，函数

`localtime` 对应的可重入函数是 `localtime_r`。在多线程程序中调用库函数，一定要使用其可重入版本，否则可能导致预想不到的结果。

14.8.2 线程和进程

思考这样一个问题：如果一个多线程程序的某个线程调用了 `fork` 函数，那么新创建的子进程是否将自动创建和父进程相同数量的线程呢？答案是“否”，正如我们期望的那样。子进程只拥有一个执行线程，它是调用 `fork` 的那个线程的完整复制。并且子进程将自动继承父进程中互斥锁（条件变量与之类似）的状态。也就是说，父进程中已经被加锁的互斥锁在子进程中也是被锁住的。这就引起了一个问题：子进程可能不清楚从父进程继承而来的互斥锁的具体状态（是加锁状态还是解锁状态）。这个互斥锁可能被加锁了，但并不是由调用 `fork` 函数的那个线程锁住的，而是由其他线程锁住的。如果是这种情况，则子进程若再次对该互斥锁执行加锁操作就会导致死锁，如代码清单 14-3 所示。

代码清单 14-3 在多线程程序中调用 `fork` 函数

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>

pthread_mutex_t mutex;
/* 子线程运行的函数。它首先获得互斥锁 mutex，然后暂停 5 s，再释放该互斥锁 */
void* another( void* arg )
{
    printf( "in child thread, lock the mutex\n" );
    pthread_mutex_lock( &mutex );
    sleep( 5 );
    pthread_mutex_unlock( &mutex );
}

int main()
{
    pthread_mutex_init( &mutex, NULL );
    pthread_t id;
    pthread_create( &id, NULL, another, NULL );
    /* 父进程中的主线程暂停 1 s，以确保在执行 fork 操作之前，子线程已经开始运行并获得了互斥锁 mutex */
    sleep( 1 );
    int pid = fork();
    if( pid < 0 )
    {
        pthread_join( id, NULL );
        pthread_mutex_destroy( &mutex );
        return 1;
    }
    else if( pid == 0 )
    {
        printf( "I am in the child, want to get the lock\n" );
    }
}
```

```

/* 子进程从父进程继承了互斥锁 mutex 的状态，该互斥锁处于锁住的状态，这是由父进程中的
子线程执行 pthread_mutex_lock 引起的，因此，下面这句加锁操作会一直阻塞，尽管从逻辑上来说它是不应该
阻塞的 */
    pthread_mutex_lock( &mutex );
    printf( "I can not run to here, oop...\n" );
    pthread_mutex_unlock( &mutex );
    exit( 0 );
}
else
{
    wait( NULL );
}
pthread_join( id, NULL );
pthread_mutex_destroy( &mutex );
return 0;
}

```

不过，`pthread`提供了一个专门的函数`pthread_atfork`，以确保`fork`调用后父进程和子进程都拥有一个清楚的锁状态。该函数的定义如下：

```
#include <pthread.h>
int pthread_atfork( void (*prepare)(void), void (*parent)(void), void (*child)(void) );
```

该函数将建立3个`fork`句柄来帮助我们清理互斥锁的状态。`prepare`句柄将在`fork`调用创建出子进程之前被执行。它可以用来锁住所有父进程中的互斥锁。`parent`句柄则是`fork`调用创建出子进程之后，而`fork`返回之前，在父进程中被执行。它的作用是释放所有在`prepare`句柄中被锁住的互斥锁。`child`句柄是`fork`返回之前，在子进程中被执行。和`parent`句柄一样，`child`句柄也是用于释放所有在`prepare`句柄中被锁住的互斥锁。该函数成功时返回0，失败则返回错误码。

因此，如果要让代码清单14-3正常工作，就应该在其中的`fork`调用前加入代码清单14-4所示的代码。

代码清单 14-4 使用 `pthread_atfork` 函数

```

void prepare()
{
    pthread_mutex_lock( &mutex );
}
void infork()
{
    pthread_mutex_unlock( &mutex );
}
pthread_atfork( prepare, infork, infork );

```

14.8.3 线程和信号

每个线程都可以独立地设置信号掩码。我们在10.3.2小节讨论过设置进程信号掩码的函

数 `sigprocmask`，但在多线程环境下我们应该使用如下所示的 `pthread` 版本的 `sigprocmask` 函数来设置线程信号掩码：

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask ( int how, const sigset_t* newmask, sigset_t* oldmask );
```

该函数的参数的含义与 `sigprocmask` 的参数完全相同，因此不再赘述。`pthread_sigmask` 成功时返回 0，失败则返回错误码。

由于进程中的所有线程共享该进程的信号，所以线程库将根据线程掩码决定把信号发送给哪个具体的线程。因此，如果我们在每个子线程中都单独设置信号掩码，就很容易导致逻辑错误。此外，所有线程共享信号处理函数。也就是说，当我们在一个线程中设置了某个信号的信号处理函数后，它将覆盖其他线程为同一个信号设置的信号处理函数。这两点都说明，我们应该定义一个专门的线程来处理所有的信号。这可以通过如下两个步骤来实现：

- 1) 在主线程创建出其他子线程之前就调用 `pthread_sigmask` 来设置好信号掩码，所有新创建的子线程都将自动继承这个信号掩码。这样做之后，实际上所有线程都不会响应被屏蔽的信号了。

- 2) 在某个线程中调用如下函数来等待信号并处理之：

```
#include <signal.h>
int sigwait( const sigset_t* set, int* sig );
```

`set` 参数指定需要等待的信号的集合。我们可以简单地将其指定为在第 1 步中创建的信号掩码，表示在该线程中等待所有被屏蔽的信号。参数 `sig` 指向的整数用于存储该函数返回的信号值。`sigwait` 成功时返回 0，失败则返回错误码。一旦 `sigwait` 正确返回，我们就可以对接收到的信号做处理了。很显然，如果我们使用了 `sigwait`，就不应该再为信号设置信号处理函数了。这是因为当程序接收到信号时，二者中只能有一个起作用。

代码清单 14-5 取自 `pthread_sigmask` 函数的 man 手册。它展示了如何通过上述两个步骤实现在一个线程中统一处理所有信号。

代码清单 14-5 用一个线程处理所有信号

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

#define handle_error_en(en, msg) \
    do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static void *sig_thread( void *arg )
{
    sigset_t *set = (sigset_t *) arg;
    int s, sig;
```

```

for( ;; )
{
    /* 第二个步骤，调用 sigwait 等待信号 */
    s = sigwait( set, &sig );
    if( s != 0 )
        handle_error_en( s, "sigwait" );
    printf( "Signal handling thread got signal %d\n", sig );
}
}

int main( int argc, char* argv[] )
{
    pthread_t thread;
    sigset_t set;
    int s;

    /* 第一个步骤，在主线程中设置信号掩码 */
    sigemptyset( &set );
    sigaddset( &set, SIGQUIT );
    sigaddset( &set, SIGUSR1 );
    s = pthread_sigmask( SIG_BLOCK, &set, NULL );
    if( s != 0 )
        handle_error_en( s, "pthread_sigmask" );

    s = pthread_create( &thread, NULL, &sig_thread, (void *) &set );
    if( s != 0 )
        handle_error_en( s, "pthread_create" );

    pause();
}

```

最后，`pthread` 还提供了下面的方法，使得我们可以明确地将一个信号发送给指定的线程：

```
#include <signal.h>
int pthread_kill( pthread_t thread, int sig );
```

其中，`thread` 参数指定目标线程，`sig` 参数指定待发送的信号。如果 `sig` 为 0，则 `pthread_kill` 不发送信号，但它仍然会执行错误检查。我们可以利用这种方式来检测目标线程是否存在。`pthread_kill` 成功时返回 0，失败则返回错误码。

第 15 章 进程池和线程池

在前面的章节中，我们是通过动态创建子进程（或子线程）来实现并发服务器的。这样做有如下缺点：

- 动态创建进程（或线程）是比较耗费时间的，这将导致较慢的客户响应。
- 动态创建的子进程（或子线程）通常只用来为一个客户服务（除非我们做特殊的处理），这将导致系统上产生大量的细微进程（或线程）。进程（或线程）间的切换将消耗大量 CPU 时间。
- 动态创建的子进程是当前进程的完整映像。当前进程必须谨慎地管理其分配的文件描述符和堆内存等系统资源，否则子进程可能复制这些资源，从而使系统的可用资源急剧下降，进而影响服务器的性能。

第 8 章介绍过的进程池和线程池可以解决上述问题。本章将分析这两种“池”的细节，给出它们的通用实现，并分别用进程池和线程池来实现简单的并发服务器。

15.1 进程池和线程池概述

进程池和线程池相似，所以这里我们只以进程池为例进行介绍。如没有特别声明，下面对进程池的讨论完全适用于线程池。

进程池是由服务器预先创建的一组子进程，这些子进程的数目在 3~10 个之间（当然，这只是典型情况）。比如 13.5.5 小节所描述的，httpd 守护进程就是使用包含 7 个子进程的进程池来实现并发的。线程池中的线程数量应该和 CPU 数量差不多。

进程池中的所有子进程都运行着相同的代码，并具有相同的属性，比如优先级、PGID 等。因为进程池在服务器启动之初就创建好了，所以每个子进程都相对“干净”，即它们没有打开不必要的文件描述符（从父进程继承而来），也不会错误地使用大块的堆内存（从父进程复制得到）。

当有新的任务到来时，主进程将通过某种方式选择进程池中的某一个子进程来为之服务。相比于动态创建子进程，选择一个已经存在的子进程的代价显然要小得多。至于主进程选择哪个子进程来为新任务服务，则有两种方式：

- 主进程使用某种算法来主动选择子进程。最简单、最常用的算法是随机算法和 Round Robin（轮流选取）算法，但更优秀、更智能的算法将使任务在各个进程中更均匀地分配，从而减轻服务器的整体压力。

- 主进程和所有子进程通过一个共享的工作队列来同步，子进程都睡眠在该工作队列上。当有新的任务到来时，主进程将任务添加到工作队列中。这将唤醒正在等待任务的子进程，不过只有一个子进程将获得新任务的“接管权”，它可以从工作队列中取出任务并执行之，而其他子进程将继续睡眠在工作队列上。

当选择好子进程后，主进程还需要使用某种通知机制来告诉目标子进程有新任务需要处理，并传递必要的数据。最简单的方法是，在父进程和子进程之间预先建立好一条管道，然后通过该管道来实现所有的进程间通信（当然，要预先定义好一套协议来规范管道的使用）。在父线程和子线程之间传递数据就要简单得多，因为我们可以把这些数据定义为全局的，那么它们本身就是被所有线程共享的。

综合上面的论述，我们将进程池的一般模型描绘为图 15-1 所示的形式。



图 15-1 进程池模型

15.2 处理多客户

在使用进程池处理多客户任务时，首先要考虑的一个问题是：监听 socket 和连接 socket 是否都由主进程来统一管理。回忆第 8 章中我们介绍过的几种并发模式，其中半同步 / 半反应堆模式是由主进程统一管理这两种 socket 的；而图 8-11 所示的高效的半同步 / 半异步模式，以及领导者 / 追随者模式，则是由主进程管理所有监听 socket，而各个子进程分别管理属于自己的连接 socket 的。对于前一种情况，主进程接受新的连接以得到连接 socket，然后它需要将该 socket 传递给子进程（对于线程池而言，父线程将 socket 传递给子线程是很简单的，因为它们可以很容易地共享该 socket。但对于进程池而言，我们必须使用 13.9 节介绍的方法来传递该 socket）。后一种情况的灵活性更大一些，因为子进程可以自己调用 accept 来接受新的连接，这样父进程就无须向子进程传递 socket，而只需要简单地通知一声：“我检测到新的连接，你来接受它。”

在 4.6.1 小节中我们曾讨论过常连接，即一个客户的多次请求可以复用一个 TCP 连接。那么，在设计进程池时还需要考虑：一个客户连接上的所有任务是否始终由一个子进程来处理。如果说客户任务是无状态的，那么我们可以考虑使用不同的子进程来为该客户的不同请求服务，如图 15-2 所示。

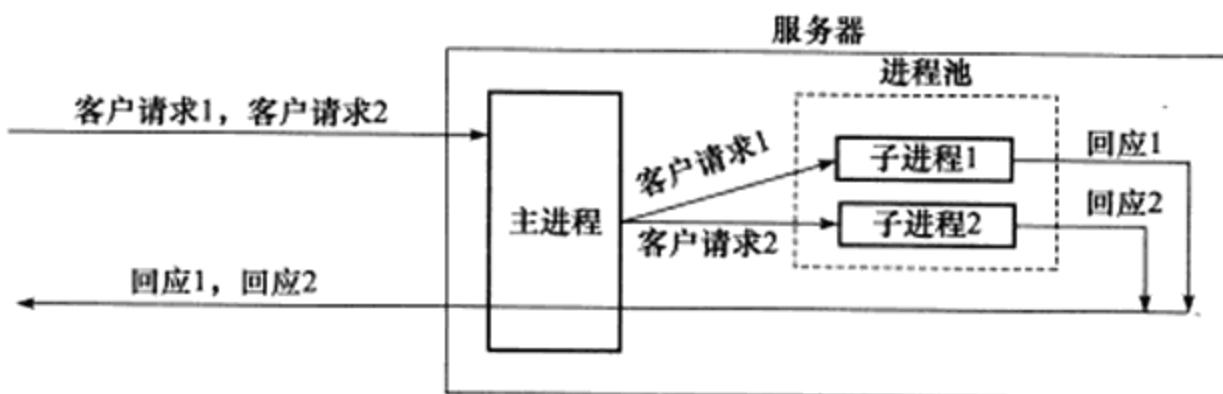


图 15-2 多个子进程处理同一个客户连接上的不同任务

但如果客户任务是存在上下文关系的，则最好一直用同一个子进程来为之服务，否则实现起来将比较麻烦，因为我们不得不在各子进程之间传递上下文数据。在 9.3.4 小节中，我们讨论了 epoll 的 EPOLLONESHOT 事件，这一事件能够确保一个客户连接在整个生命周期中仅被一个线程处理。

15.3 半同步 / 半异步进程池实现

综合前面的讨论，本节我们实现一个基于图 8-11 所示的半同步 / 半异步并发模式的进程池，如代码清单 15-1 所示。为了避免在父、子进程之间传递文件描述符，我们将接受新连接的操作放到子进程中。很显然，对于这种模式而言，一个客户连接上的所有任务始终是由一个子进程来处理的。

代码清单 15-1 半同步 / 半异步进程池

```

// filename: processpool.h
#ifndef PROCESSPOOL_H
#define PROCESSPOOL_H

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/stat.h>

/* 描述一个子进程的类，m_pid 是目标子进程的 PID，m_pipefd 是父进程和子进程通信用的管道 */
class process
{

```

```

public:
    process() : m_pid( -1 ) {}

public:
    pid_t m_pid;
    int m_pipefd[2];
};

/* 进程池类，将它定义为模板类是为了代码复用。其模板参数是处理逻辑任务的类 */
template< typename T >
class processpool
{
private:
    /* 将构造函数定义为私有的，因此我们只能通过后面的 create 静态函数来创建 processpool 实例 */
    processpool( int listenfd, int process_number = 8 );
public:
    /* 单体模式，以保证程序最多创建一个 processpool 实例，这是程序正确处理信号的必要条件 */
    static processpool< T >* create( int listenfd, int process_number = 8 )
    {
        if( !m_instance )
        {
            m_instance = new processpool< T >( listenfd, process_number );
        }
        return m_instance;
    }
    ~processpool()
    {
        delete [] m_sub_process;
    }
    /* 启动进程池 */
    void run();

private:
    void setup_sig_pipe();
    void run_parent();
    void run_child();

private:
    /* 进程池允许的最大子进程数量 */
    static const int MAX_PROCESS_NUMBER = 16;
    /* 每个子进程最多能处理的客户数量 */
    static const int USER_PER_PROCESS = 65536;
    /* epoll 最多能处理的事件数 */
    static const int MAX_EVENT_NUMBER = 10000;
    /* 进程池中的进程总数 */
    int m_process_number;
    /* 子进程在池中的序号，从 0 开始 */
    int m_idx;
    /* 每个进程都有一个 epoll 内核事件表，用 m_epollfd 标识 */
    int m_epollfd;
    /* 监听 socket */
    int m_listenfd;
    /* 子进程通过 m_stop 来决定是否停止运行 */

```

```
int m_stop;
/* 保存所有子进程的描述信息 */
process* m_sub_process;
/* 进程池静态实例 */
static processpool< T >* m_instance;
};

template< typename T >
processpool< T >* processpool< T >::m_instance = NULL;

/* 用于处理信号的管道，以实现统一事件源。后面称之为信号管道 */
static int sig_pipefd[2];

static int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

static void addfd( int epollfd, int fd )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET;
    epoll_ctl( epollfd, EPOLL_CTL_ADD, fd, &event );
    setnonblocking( fd );
}

/* 从 epollfd 标识的 epoll 内核事件表中删除 fd 上的所有注册事件 */
static void removefd( int epollfd, int fd )
{
    epoll_ctl( epollfd, EPOLL_CTL_DEL, fd, 0 );
    close( fd );
}

static void sig_handler( int sig )
{
    int save_errno = errno;
    int msg = sig;
    send( sig_pipefd[1], ( char* )&msg, 1, 0 );
    errno = save_errno;
}

static void addsig( int sig, void( handler )(int), bool restart = true )
{
    struct sigaction sa;
    memset( &sa, '\0', sizeof( sa ) );
    sa.sa_handler = handler;
    if( restart )
    {
        sa.sa_flags |= SA_RESTART;
    }
}
```

```

        sigfillset( &sa.sa_mask );
        assert( sigaction( sig, &sa, NULL ) != -1 );
    }

/* 进程池构造函数。参数 listenfd 是监听 socket，它必须在创建进程池之前被创建，否则子进程无法
直接引用它。参数 process_number 指定进程池中子进程的数量 */
template< typename T >
processpool< T >::processpool( int listenfd, int process_number )
:m_listenfd( listenfd ), m_process_number( process_number ), m_idx( -1 ),
m_stop( false )
{
    assert( ( process_number > 0 ) && ( process_number <= MAX_PROCESS_NUMBER ) );

    m_sub_process = new process[ process_number ];
    assert( m_sub_process );

    /* 创建 process_number 个子进程，并建立它们和父进程之间的管道 */
    for( int i = 0; i < process_number; ++i )
    {
        int ret = socketpair( PF_UNIX, SOCK_STREAM, 0, m_sub_process[i].m_pipefd );
        assert( ret == 0 );

        m_sub_process[i].m_pid = fork();
        assert( m_sub_process[i].m_pid >= 0 );
        if( m_sub_process[i].m_pid > 0 )
        {
            close( m_sub_process[i].m_pipefd[1] );
            continue;
        }
        else
        {
            close( m_sub_process[i].m_pipefd[0] );
            m_idx = i;
            break;
        }
    }
}

/* 统一事件源 */
template< typename T >
void processpool< T >::setup_sig_pipe()
{
    /* 创建 epoll 事件监听表和信号管道 */
    m_epollfd = epoll_create( 5 );
    assert( m_epollfd != -1 );

    int ret = socketpair( PF_UNIX, SOCK_STREAM, 0, sig_pipefd );
    assert( ret != -1 );

    setnonblocking( sig_pipefd[1] );
    addfd( m_epollfd, sig_pipefd[0] );

    /* 设置信号处理函数 */
    addsig( SIGCHLD, sig_handler );
}

```

```

    addsig( SIGTERM, sig_handler );
    addsig( SIGINT, sig_handler );
    addsig( SIGPIPE, SIG_IGN );
}

/* 父进程中 m_idx 值为 -1，子进程中 m_idx 值大于等于 0，我们据此判断接下来要运行的是父进程代码
还是子进程代码 */
template< typename T >
void processpool< T >::run()
{
    if( m_idx != -1 )
    {
        run_child();
        return;
    }
    run_parent();
}

template< typename T >
void processpool< T >::run_child()
{
    setup_sig_pipe();

    /* 每个子进程都通过其在进程池中的序号值 m_idx 找到与父进程通信的管道 */
    int pipefd = m_sub_process[m_idx].m_pipefd[ 1 ];
    /* 子进程需要监听管道文件描述符 pipefd，因为父进程将通过它来通知子进程 accept 新连接 */
    addfd( m_epollfd, pipefd );

    epoll_event events[ MAX_EVENT_NUMBER ];
    T* users = new T [ USER_PER_PROCESS ];
    assert( users );
    int number = 0;
    int ret = -1;

    while( ! m_stop )
    {
        number = epoll_wait( m_epollfd, events, MAX_EVENT_NUMBER, -1 );
        if((number < 0) && (errno != EINTR))
        {
            printf( "epoll failure\n" );
            break;
        }

        for(int i = 0; i < number; i++)
        {
            int sockfd = events[i].data.fd;
            if((sockfd == pipefd) && (events[i].events & EPOLLIN))
            {
                int client = 0;
                /* 从父、子进程之间的管道读取数据，并将结果保存在变量 client 中。如果读
取成功，则表示有新客户连接到来 */
                ret = recv( sockfd, (char*) &client, sizeof( client ), 0 );
                if(((ret < 0) && (errno != EAGAIN)) || ret == 0)

```

```

    {
        continue;
    }
    else
    {
        struct sockaddr_in client_address;
        socklen_t client_addrlength = sizeof( client_address );
        int connfd = accept(m_listenfd, ( struct sockaddr* )
            &client_address,&client_addrlength );
        if (connfd < 0 )
        {
            printf( "errno is: %d\n", errno );
            continue;
        }
        addfd( m_epollfd, connfd );
        /* 模板类 T 必须实现 init 方法，以初始化一个客户连接。我们直接使用 connfd
        来索引逻辑处理对象 (T 类型的对象)，以提高程序效率 */
        users[connfd].init( m_epollfd, connfd, client_address );
    }
}
/* 下面处理子进程接收到的信号 */
else if( ( sockfd == sig_pipefd[0] ) && ( events[i].events & EPOLLIN ) )
{
    int sig;
    char signals[1024];
    ret = recv( sig_pipefd[0], signals, sizeof( signals ), 0 );
    if( ret <= 0 )
    {
        continue;
    }
    else
    {
        for( int i = 0; i < ret; ++i )
        {
            switch( signals[i] )
            {
                case SIGCHLD:
                {
                    pid_t pid;
                    int stat;
                    while ((pid = waitpid( -1, &stat, WNOHANG ) ) > 0 )
                    {
                        continue;
                    }
                    break;
                }
                case SIGTERM:
                case SIGINT:
                {
                    m_stop = true;
                    break;
                }
            }
        }
    }
}

```

```

        default:
        {
            break;
        }
    }
}

/* 如果是其他可读数据，那么必然是客户请求到来。调用逻辑处理对象的 process 方法处理之 */
else if( events[i].events & EPOLLIN )
{
    users[sockfd].process();
}
else
{
    continue;
}
}

delete [] users;
users = NULL;
close( pipefd );
// close( m_listenfd ); /* 我们将这句话注释掉，以提醒读者：应该由 m_listenfd 的创建者来关闭这个文件描述符（见后文），即所谓的“对象（比如一个文件描述符，又或者一段堆内存）由哪个函数创建，就应该由哪个函数销毁”*/
close( m_epollfd );
}

template< typename T >
void processpool< T >::run_parent()
{
    setup_sig_pipe();

    /* 父进程监听 m_listenfd */
    addfd( m_epollfd, m_listenfd );

    epoll_event events[ MAX_EVENT_NUMBER ];
    int sub_process_counter = 0;
    int new_conn = 1;
    int number = 0;
    int ret = -1;

    while( ! m_stop )
    {
        number = epoll_wait( m_epollfd, events, MAX_EVENT_NUMBER, -1 );
        if ( ( number < 0 ) && ( errno != EINTR ) )
        {
            printf( "epoll failure\n" );
            break;
        }
    }
}

```

```

for ( int i = 0; i < number; i++ )
{
    int sockfd = events[i].data.fd;
    if( sockfd == m_listenfd )
    {
        /* 如果有新连接到来，就采用 Round Robin 方式将其分配给一个子进程处理 */
        int i = sub_process_counter;
        do
        {
            if( m_sub_process[i].m_pid != -1 )
            {
                break;
            }
            i = (i+1)%m_process_number;
        }
        while( i != sub_process_counter );

        if( m_sub_process[i].m_pid == -1 )
        {
            m_stop = true;
            break;
        }
        sub_process_counter = (i+1)%m_process_number;
        send( m_sub_process[i].m_pipefd[0],
              ( char* )&new_conn, sizeof( new_conn ), 0 );
        printf( "send request to child %d\n", i );
    }
    /* 下面处理父进程接收到的信号 */
    else if( ( sockfd == sig_pipefd[0] ) && ( events[i].events & EPOLLIN ) )
    {
        int sig;
        char signals[1024];
        ret = recv( sig_pipefd[0], signals, sizeof( signals ), 0 );
        if( ret <= 0 )
        {
            continue;
        }
        else
        {
            for( int i = 0; i < ret; ++i )
            {
                switch( signals[i] )
                {
                    case SIGCHLD:
                    {
                        pid_t pid;
                        int stat;
                        while ( ( pid = waitpid( -1, &stat, WNOHANG ) ) > 0 )
                        {
                            for( int i = 0; i < m_process_number; ++i )
                            {
                                /* 如果进程池中第 i 个子进程退出了，则主进程关闭相应的通信管道，并设置相应的 m_pid 为 -1，以标记该子进程已经退出 */

```

```

        if( m_sub_process[i].m_pid == pid )
        {
            printf( "child %d join\n", i );
            close( m_sub_process[i].m_pipefd[0] );
            m_sub_process[i].m_pid = -1;
        }
    }

    /* 如果所有子进程都已经退出了，则父进程也退出 */
    m_stop = true;
    for( int i = 0; i < m_process_number; ++i )
    {
        if( m_sub_process[i].m_pid != -1 )
        {
            m_stop = false;
        }
    }
    break;
}

case SIGTERM:
case SIGINT:
{
    /* 如果父进程接收到终止信号，那么就杀死所有子进程，并等待它们全部结束。当然，通知子进程结束更好的方法是向父、子进程之间的通信管道发送特殊数据，读者不妨自己实现之 */
    printf( "kill all the child now\n" );
    for( int i = 0; i < m_process_number; ++i )
    {
        int pid = m_sub_process[i].m_pid;
        if( pid != -1 )
        {
            kill( pid, SIGTERM );
        }
    }
    break;
}
default:
{
    break;
}
}

}

}

else
{
    continue;
}
}

// close( m_listenfd ); /* 由创建者关闭这个文件描述符（见后文） */
close( m_epollfd );

```

```

}
#endif

```

15.4 用进程池实现的简单 CGI 服务器

回忆 6.2 节，我们曾实现过一个非常简单的 CGI 服务器。下面我们将利用前面介绍的进程池来重新实现一个并发的 CGI 服务器，如代码清单 15-2 所示。

代码清单 15-2 用进程池实现的并发 CGI 服务器

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/stat.h>

#include "processpool.h" /* 引用上一节介绍的进程池 */

/* 用于处理客户 CGI 请求的类，它可以作为 processpool 类的模板参数 */
class cgi_conn
{
public:
    cgi_conn(){}
    ~cgi_conn(){}
    /* 初始化客户连接，清空读缓冲区 */
    void init( int epollfd, int sockfd, const sockaddr_in& client_addr )
    {
        m_epollfd = epollfd;
        m_sockfd = sockfd;
        m_address = client_addr;
        memset( m_buf, '\0', BUFFER_SIZE );
        m_read_idx = 0;
    }

    void process()
    {
        int idx = 0;
        int ret = -1;
        /* 循环读取和分析客户数据 */
        while( true )

```

```

{
    idx = m_read_idx;
    ret = recv( m_sockfd, m_buf + idx, BUFFER_SIZE-1-idx, 0 );
    /* 如果读操作发生错误，则关闭客户连接。但如果是暂时无数据可读，则退出循环 */
    if( ret < 0 )
    {
        if( errno != EAGAIN )
        {
            removefd( m_epollfd, m_sockfd );
        }
        break;
    }
    /* 如果对方关闭连接，则服务器也关闭连接 */
    else if( ret == 0 )
    {
        removefd( m_epollfd, m_sockfd );
        break;
    }
    else
    {
        m_read_idx += ret;
        printf( "user content is: %s\n", m_buf );
        /* 如果遇到字符 “\r\n”，则开始处理客户请求 */
        for( ; idx < m_read_idx; ++idx )
        {
            if( ( idx >= 1 ) && ( m_buf[idx-1] == '\r' ) && ( m_buf[idx] == '\n' ) )
            {
                break;
            }
        }
        /* 如果没有遇到字符 “\r\n”，则需要读取更多客户数据 */
        if( idx == m_read_idx )
        {
            continue;
        }
        m_buf[ idx-1 ] = '\0';

        char* file_name = m_buf;
        /* 判断客户要运行的 CGI 程序是否存在 */
        if( access( file_name, F_OK ) == -1 )
        {
            removefd( m_epollfd, m_sockfd );
            break;
        }
        /* 创建子进程来执行 CGI 程序 */
        ret = fork();
        if( ret == -1 )
        {
            removefd( m_epollfd, m_sockfd );
            break;
        }
        else if( ret > 0 )

```

```

    {
        /* 父进程只需关闭连接 */
        removefd( m_epollfd, m_sockfd );
        break;
    }
    else
    {
        /* 子进程将标准输出定向到 m_sockfd，并执行 CGI 程序 */
        close( STDOUT_FILENO );
        dup( m_sockfd );
        execl( m_buf, m_buf, 0 );
        exit( 0 );
    }
}

private:
/* 读缓冲区的大小 */
static const int BUFFER_SIZE = 1024;
static int m_epollfd;
int m_sockfd;
sockaddr_in m_address;
char m_buf[ BUFFER_SIZE ];
/* 标记读缓冲中已经读入的客户数据的最后一个字节的下一个位置 */
int m_read_idx;
};

int cgi_conn::m_epollfd = -1;

/* 主函数 */
int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    int listenfd = socket( PF_INET, SOCK_STREAM, 0 );
    assert( listenfd >= 0 );

    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    ret = bind( listenfd, ( struct sockaddr* )&address, sizeof( address ) );
    assert( ret != -1 );
}

```

```

ret = listen( listenfd, 5 );
assert( ret != -1 );

processpool< cgi_conn *>* pool = processpool< cgi_conn *>::create( listenfd );
if( pool )
{
    pool->run();
    delete pool;
}
close( listenfd ); /* 正如前文提到的, main 函数创建了文件描述符 listenfd, 那么就由
它亲自关闭之 */
return 0;
}

```

15.5 半同步 / 半反应堆线程池实现

本节我们实现一个基于图 8-10 所示的半同步 / 半反应堆并发模式的线程池，如代码清单 15-3 所示。相比代码清单 15-1 所示的进程池实现，该线程池的通用性要高得多，因为它使用一个工作队列完全解除了主线程和工作线程的耦合关系：主线程往工作队列中插入任务，工作线程通过竞争来取得任务并执行它。不过，如果要将该线程池应用到实际服务器程序中，那么我们必须保证所有客户请求都是无状态的，因为同一个连接上的不同请求可能会由不同的线程处理。

代码清单 15-3 半同步 / 半反应堆线程池实现

```

// filename: threadpool.h
#ifndef THREADPOOL_H
#define THREADPOOL_H

#include <list>
#include <cstdio>
#include <exception>
#include <pthread.h>
/* 引用第 14 章介绍的线程同步机制的包装类 */
#include "locker.h"

/* 线程池类，将它定义为模板类是为了代码复用。模板参数 T 是任务类 */
template< typename T >
class threadpool
{
public:
    /* 参数 thread_number 是线程池中线程的数量，max_requests 是请求队列中最多允许的、等待
    处理的请求的数量 */
    threadpool( int thread_number = 8, int max_requests = 10000 );
    ~threadpool();
    /* 往请求队列中添加任务 */
    bool append( T* request );
private:

```

```

/* 工作线程运行的函数，它不断从工作队列中取出任务并执行之 */
static void* worker( void* arg );
void run();

private:
    int m_thread_number; /* 线程池中的线程数 */
    int m_max_requests; /* 请求队列中允许的最大请求数 */
    pthread_t* m_threads; /* 描述线程池的数组，其大小为 m_thread_number */
    std::list< T* > m_workqueue; /* 请求队列 */
    locker m_queuelocker; /* 保护请求队列的互斥锁 */
    sem m_queuestat; /* 是否有任务需要处理 */
    bool m_stop; /* 是否结束线程 */
};

template< typename T >
threadpool< T >::threadpool( int thread_number, int max_requests ) :
    m_thread_number( thread_number ), m_max_requests( max_requests ),
    m_stop( false ), m_threads( NULL )
{
    if(( thread_number <= 0 ) || ( max_requests <= 0 ) )
    {
        throw std::exception();
    }

    m_threads = new pthread_t[ m_thread_number ];
    if(!m_threads )
    {
        throw std::exception();
    }

    /* 创建 thread_number 个线程，并将它们都设置为脱离线程 */
    for ( int i = 0; i < thread_number; ++i )
    {
        printf( "create the %dth thread\n", i );
        if( pthread_create( m_threads + i, NULL, worker, this ) != 0 )
        {
            delete [] m_threads;
            throw std::exception();
        }
        if( pthread_detach( m_threads[i] ) )
        {
            delete [] m_threads;
            throw std::exception();
        }
    }
}

template< typename T >
threadpool< T >::~threadpool()
{
    delete [] m_threads;
    m_stop = true;
}

```

```

template< typename T >
bool threadpool< T >::append( T* request )
{
    /* 操作工作队列时一定要加锁，因为它被所有线程共享 */
    m_queuelocker.lock();
    if ( m_workqueue.size() > m_max_requests )
    {
        m_queuelocker.unlock();
        return false;
    }
    m_workqueue.push_back( request );
    m_queuelocker.unlock();
    m_queuestat.post();
    return true;
}

template< typename T >
void* threadpool< T >::worker( void* arg )
{
    threadpool* pool = ( threadpool* )arg;
    pool->run();
    return pool;
}

template< typename T >
void threadpool< T >::run()
{
    while ( ! m_stop )
    {
        m_queuestat.wait();
        m_queuelocker.lock();
        if ( m_workqueue.empty() )
        {
            m_queuelocker.unlock();
            continue;
        }
        T* request = m_workqueue.front();
        m_workqueue.pop_front();
        m_queuelocker.unlock();
        if ( ! request )
        {
            continue;
        }
        request->process();
    }
}
#endif

```

值得一提的是，在C++程序中使用pthread_create函数时，该函数的第3个参数必须指向一个静态函数。而要在一个静态函数中使用类的动态成员（包括成员函数和成员变量），

则只能通过如下两种方式来实现：

- 通过类的静态对象来调用。比如单体模式中，静态函数可以通过类的全局唯一实例来访问动态成员函数。
- 将类的对象作为参数传递给该静态函数，然后在静态函数中引用这个对象，并调用其动态方法。

代码清单 15-3 使用的是第 2 种方式：将线程参数设置为 this 指针，然后在 worker 函数中获取该指针并调用其动态方法 run。

15.6 用线程池实现的简单 Web 服务器

在 8.6 节中，我们曾使用有限状态机实现过一个非常简单的解析 HTTP 请求的服务器。下面我们将利用前面介绍的线程池来重新实现一个并发的 Web 服务器。

15.6.1 http_conn 类

首先，我们需要准备线程池的模板参数类，用以封装对逻辑任务的处理。这个类是 http_conn，代码清单 15-4 是其头文件（http_conn.h），代码清单 15-5 是其实现文件（http_conn.cpp）。

代码清单 15-4 http_conn.h 文件

```
#ifndef HTTPCONNECTION_H
#define HTTPCONNECTION_H

#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <stdarg.h>
#include <errno.h>
#include "locker.h"

class http_conn
{
public:
    /* 文件名的最大长度 */
    static const int FILENAME_LEN = 200;
```

```

/* 读缓冲区的大小 */
static const int READ_BUFFER_SIZE = 2048;
/* 写缓冲区的大小 */
static const int WRITE_BUFFER_SIZE = 1024;
/* HTTP 请求方法，但我们仅支持 GET */
enum METHOD { GET = 0, POST, HEAD, PUT, DELETE,
              TRACE, OPTIONS, CONNECT, PATCH };
/* 解析客户请求时，主状态机所处的状态（回忆第 8 章） */
enum CHECK_STATE { CHECK_STATE_REQUESTLINE = 0,
                     CHECK_STATE_HEADER,
                     CHECK_STATE_CONTENT };
/* 服务器处理 HTTP 请求的可能结果 */
enum HTTP_CODE { NO_REQUEST, GET_REQUEST, BAD_REQUEST,
                  NO_RESOURCE, FORBIDDEN_REQUEST, FILE_REQUEST,
                  INTERNAL_ERROR, CLOSED_CONNECTION };
/* 行的读取状态 */
enum LINE_STATUS { LINE_OK = 0, LINE_BAD, LINE_OPEN };

public:
    http_conn(){}
    ~http_conn(){}

public:
    /* 初始化新接受的连接 */
    void init( int sockfd, const sockaddr_in& addr );
    /* 关闭连接 */
    void close_conn( bool real_close = true );
    /* 处理客户请求 */
    void process();
    /* 非阻塞读操作 */
    bool read();
    /* 非阻塞写操作 */
    bool write();

private:
    /* 初始化连接 */
    void init();
    /* 解析 HTTP 请求 */
    HTTP_CODE process_read();
    /* 填充 HTTP 应答 */
    bool process_write( HTTP_CODE ret );

    /* 下面这一组函数被 process_read 调用以分析 HTTP 请求 */
    HTTP_CODE parse_request_line( char* text );
    HTTP_CODE parse_headers( char* text );
    HTTP_CODE parse_content( char* text );
    HTTP_CODE do_request();
    char* get_line() { return m_read_buf + m_start_line; }
    LINE_STATUS parse_line();

    /* 下面这一组函数被 process_write 调用以填充 HTTP 应答 */
    void unmap();
    bool add_response( const char* format, ... );

```

```

    bool add_content( const char* content );
    bool add_status_line( int status, const char* title );
    bool add_headers( int content_length );
    bool add_content_length( int content_length );
    bool add_linger();
    bool add_blank_line();

public:
    /* 所有 socket 上的事件都被注册到同一个 epoll 内核事件表中，所以将 epoll 文件描述符设置为
    静态的 */
    static int m_epollfd;
    /* 统计用户数量 */
    static int m_user_count;

private:
    /* 该 HTTP 连接的 socket 和对方的 socket 地址 */
    int m_sockfd;
    sockaddr_in m_address;

    /* 读缓冲区 */
    char m_read_buf[ READ_BUFFER_SIZE ];
    /* 标识读缓冲中已经读入的客户数据的最后一个字节的下一个位置 */
    int m_read_idx;
    /* 当前正在分析的字符在读缓冲区中的位置 */
    int m_checked_idx;
    /* 当前正在解析的行的起始位置 */
    int m_start_line;
    /* 写缓冲区 */
    char m_write_buf[ WRITE_BUFFER_SIZE ];
    /* 写缓冲区中待发送的字节数 */
    int m_write_idx;

    /* 主状态机当前所处的状态 */
    CHECK_STATE m_check_state;
    /* 请求方法 */
    METHOD m_method;

    /* 客户请求的目标文件的完整路径，其内容等于 doc_root + m_url，doc_root 是网站根目录 */
    char m_real_file[ FILENAME_LEN ];
    /* 客户请求的目标文件的文件名 */
    char* m_url;
    /* HTTP 协议版本号，我们仅支持 HTTP/1.1 */
    char* m_version;
    /* 主机名 */
    char* m_host;
    /* HTTP 请求的消息体的长度 */
    int m_content_length;
    /* HTTP 请求是否要求保持连接 */
    bool m_linger;

    /* 客户请求的目标文件被 mmap 到内存中的起始位置 */
    char* m_file_address;
    /* 目标文件的状态。通过它我们可以判断文件是否存在、是否为目录、是否可读，并获取文件大小等
    */
}

```

```

信息 */
    struct stat m_file_stat;
    /* 我们将采用 writev 来执行写操作，所以定义下面两个成员，其中 m_iv_count 表示被写内存块
的数量 */
    struct iovec m_iv[2];
    int m_iv_count;
};

#endif

```

代码清单 15-5 http_conn.cpp 文件

```

#include "http_conn.h"

/* 定义 HTTP 响应的一些状态信息 */
const char* ok_200_title = "OK";
const char* error_400_title = "Bad Request";
const char* error_400_form = "Your request has bad syntax or is inherently
impossible to satisfy.\n";
const char* error_403_title = "Forbidden";
const char* error_403_form = "You do not have permission to get file from this
server.\n";
const char* error_404_title = "Not Found";
const char* error_404_form = "The requested file was not found on this
server.\n";
const char* error_500_title = "Internal Error";
const char* error_500_form = "There was an unusual problem serving the
requested file.\n";
/* 网站的根目录 */
const char* doc_root = "/var/www/html";

int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

void addfd( int epollfd, int fd, bool one_shot )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET | EPOLLRDHUP;
    if( one_shot )
    {
        event.events |= EPOLLONESHOT;
    }
    epoll_ctl( epollfd, EPOLL_CTL_ADD, fd, &event );
    setnonblocking( fd );
}

```

```

void removefd( int epollfd, int fd )
{
    epoll_ctl( epollfd, EPOLL_CTL_DEL, fd, 0 );
    close( fd );
}

void modfd( int epollfd, int fd, int ev )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = ev | EPOLLET | EPOLLONESHOT | EPOLLRDHUP;
    epoll_ctl( epollfd, EPOLL_CTL_MOD, fd, &event );
}

int http_conn::m_user_count = 0;
int http_conn::m_epollfd = -1;

void http_conn::close_conn( bool real_close )
{
    if( real_close && ( m_sockfd != -1 ) )
    {
        removefd( m_epollfd, m_sockfd );
        m_sockfd = -1;
        m_user_count--; /* 关闭一个连接时，将客户总量减1 */
    }
}

void http_conn::init( int sockfd, const sockaddr_in& addr )
{
    m_sockfd = sockfd;
    m_address = addr;
    /* 如下两行是为了避免 TIME_WAIT 状态，仅用于调试，实际使用时应该去掉 */
    int reuse = 1;
    setsockopt( m_sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof( reuse ) );
    addfd( m_epollfd, sockfd, true );
    m_user_count++;

    init();
}

void http_conn::init()
{
    m_check_state = CHECK_STATE_REQUESTLINE;
    m_linger = false;

    m_method = GET;
    m_url = 0;
    m_version = 0;
    m_content_length = 0;
    m_host = 0;
    m_start_line = 0;
    m_checked_idx = 0;
    m_read_idx = 0;
}

```

```

    m_write_idx = 0;
    memset( m_read_buf, '\0', READ_BUFFER_SIZE );
    memset( m_write_buf, '\0', WRITE_BUFFER_SIZE );
    memset( m_real_file, '\0', FILENAME_LEN );
}

/* 从状态机，其分析请参考8.6节，这里不再赘述 */
http_conn::LINE_STATUS http_conn::parse_line()
{
    char temp;
    for ( ; m_checked_idx < m_read_idx; ++m_checked_idx )
    {
        temp = m_read_buf[ m_checked_idx ];
        if ( temp == '\r' )
        {
            if ( ( m_checked_idx + 1 ) == m_read_idx )
            {
                return LINE_OPEN;
            }
            else if ( m_read_buf[ m_checked_idx + 1 ] == '\n' )
            {
                m_read_buf[ m_checked_idx++ ] = '\0';
                m_read_buf[ m_checked_idx++ ] = '\0';
                return LINE_OK;
            }

            return LINE_BAD;
        }
        else if( temp == '\n' )
        {
            if( ( m_checked_idx > 1 ) && ( m_read_buf[ m_checked_idx - 1 ] == '\r' ) )
            {
                m_read_buf[ m_checked_idx-1 ] = '\0';
                m_read_buf[ m_checked_idx++ ] = '\0';
                return LINE_OK;
            }
            return LINE_BAD;
        }
    }

    return LINE_OPEN;
}

/* 循环读取客户数据，直到无数据可读或者对方关闭连接 */
bool http_conn::read()
{
    if( m_read_idx >= READ_BUFFER_SIZE )
    {
        return false;
    }

    int bytes_read = 0;
    while( true )

```

```

    {
        bytes_read = recv( m_sockfd, m_read_buf + m_read_idx, READ_BUFFER_SIZE -
                           m_read_idx, 0 );
        if ( bytes_read == -1 )
        {
            if( errno == EAGAIN || errno == EWOULDBLOCK )
            {
                break;
            }
            return false;
        }
        else if ( bytes_read == 0 )
        {
            return false;
        }

        m_read_idx += bytes_read;
    }
    return true;
}

/* 解析 HTTP 请求行，获得请求方法、目标 URL，以及 HTTP 版本号 */
http_conn::HTTP_CODE http_conn::parse_request_line( char* text )
{
    m_url = strpbrk( text, "\t" );
    if ( ! m_url )
    {
        return BAD_REQUEST;
    }
    *m_url++ = '\0';

    char* method = text;
    if ( strcasecmp( method, "GET" ) == 0 )
    {
        m_method = GET;
    }
    else
    {
        return BAD_REQUEST;
    }

    m_url += strspn( m_url, "\t" );
    m_version = strpbrk( m_url, "\t" );
    if ( ! m_version )
    {
        return BAD_REQUEST;
    }
    *m_version++ = '\0';
    m_version += strspn( m_version, "\t" );
    if ( strcasecmp( m_version, "HTTP/1.1" ) != 0 )
    {
        return BAD_REQUEST;
    }
}

```

```

if ( strncasecmp( m_url, "http://", 7 ) == 0 )
{
    m_url += 7;
    m_url = strchr( m_url, '/' );
}

if ( ! m_url || m_url[ 0 ] != '/' )
{
    return BAD_REQUEST;
}

m_check_state = CHECK_STATE_HEADER;
return NO_REQUEST;
}

/* 解析 HTTP 请求的一个头部信息 */
http_conn::HTTP_CODE http_conn::parse_headers( char* text )
{
    /* 遇到空行，表示头部字段解析完毕 */
    if( text[ 0 ] == '\0' )
    {
        /* 如果 HTTP 请求有消息体，则还需要读取 m_content_length 字节的消息体，状态机转移到
        CHECK_STATE_CONTENT 状态 */
        if ( m_content_length != 0 )
        {
            m_check_state = CHECK_STATE_CONTENT;
            return NO_REQUEST;
        }

        /* 否则说明我们已经得到了一个完整的 HTTP 请求 */
        return GET_REQUEST;
    }
    /* 处理 Connection 头部字段 */
    else if ( strncasecmp( text, "Connection:", 11 ) == 0 )
    {
        text += 11;
        text += strspn( text, "\t" );
        if ( strcasecmp( text, "keep-alive" ) == 0 )
        {
            m_linger = true;
        }
    }
    /* 处理 Content-Length 头部字段 */
    else if ( strncasecmp( text, "Content-Length:", 15 ) == 0 )
    {
        text += 15;
        text += strspn( text, "\t" );
        m_content_length = atol( text );
    }
    /* 处理 Host 头部字段 */
    else if ( strncasecmp( text, "Host:", 5 ) == 0 )
    {

```

```

        text += 5;
        text += strspn( text, " \t" );
        m_host = text;
    }
    else
    {
        printf( "oop! unknow header %s\n", text );
    }

    return NO_REQUEST;
}

/* 我们没有真正解析 HTTP 请求的消息体，只是判断它是否被完整地读入了 */
http_conn::HTTP_CODE http_conn::parse_content( char* text )
{
    if ( m_read_idx >= ( m_content_length + m_checked_idx ) )
    {
        text[ m_content_length ] = '\0';
        return GET_REQUEST;
    }

    return NO_REQUEST;
}

/* 主状态机。其分析请参考 8.6 节，这里不再赘述 */
http_conn::HTTP_CODE http_conn::process_read()
{
    LINE_STATUS line_status = LINE_OK;
    HTTP_CODE ret = NO_REQUEST;
    char* text = 0;

    while ( ( ( m_check_state == CHECK_STATE_CONTENT ) && ( line_status == LINE_OK ) )
           || ( ( line_status = parse_line() ) == LINE_OK ) )
    {
        text = get_line();
        m_start_line = m_checked_idx;
        printf( "got 1 http line: %s\n", text );

        switch ( m_check_state )
        {
            case CHECK_STATE_REQUESTLINE:
            {
                ret = parse_request_line( text );
                if ( ret == BAD_REQUEST )
                {
                    return BAD_REQUEST;
                }
                break;
            }
            case CHECK_STATE_HEADER:
            {
                ret = parse_headers( text );

```

```

        if ( ret == BAD_REQUEST )
        {
            return BAD_REQUEST;
        }
        else if ( ret == GET_REQUEST )
        {
            return do_request();
        }
        break;
    }
    case CHECK_STATE_CONTENT:
    {
        ret = parse_content( text );
        if ( ret == GET_REQUEST )
        {
            return do_request();
        }
        line_status = LINE_OPEN;
        break;
    }
    default:
    {
        return INTERNAL_ERROR;
    }
}
}

return NO_REQUEST;
}

/* 当得到一个完整、正确的 HTTP 请求时，我们就分析目标文件的属性。如果目标文件存在、对所有用户可读，且不是目录，则使用 mmap 将其映射到内存地址 m_file_address 处，并告诉调用者获取文件成功 */
http_conn::HTTP_CODE http_conn::do_request()
{
    strcpy( m_real_file, doc_root );
    int len = strlen( doc_root );
    strncpy( m_real_file + len, m_url, FILENAME_LEN - len - 1 );
    if ( stat( m_real_file, &m_file_stat ) < 0 )
    {
        return NO_RESOURCE;
    }

    if ( ! ( m_file_stat.st_mode & S_IROTH ) )
    {
        return FORBIDDEN_REQUEST;
    }

    if ( S_ISDIR( m_file_stat.st_mode ) )
    {
        return BAD_REQUEST;
    }

    int fd = open( m_real_file, O_RDONLY );

```

```

m_file_address = ( char* )mmap( 0, m_file_stat.st_size, PROT_READ,
                                MAP_PRIVATE, fd, 0 );
close( fd );
return FILE_REQUEST;
}

/* 对内存映射区执行 munmap 操作 */
void http_conn::unmap()
{
    if( m_file_address )
    {
        munmap( m_file_address, m_file_stat.st_size );
        m_file_address = 0;
    }
}

/* 写 HTTP 响应 */
bool http_conn::write()
{
    int temp = 0;
    int bytes_have_send = 0;
    int bytes_to_send = m_write_idx;
    if ( bytes_to_send == 0 )
    {
        modfd( m_epollfd, m_sockfd, EPOLLIN );
        init();
        return true;
    }

    while( 1 )
    {
        temp = writev( m_sockfd, m_iv, m_iv_count );
        if ( temp <= -1 )
        {
            /* 如果 TCP 写缓冲没有空间，则等待下一轮 EPOLLOUT 事件。虽然在此期间，服务
               器无法立即接收到同一客户的下一个请求，但这可以保证连接的完整性 */
            if( errno == EAGAIN )
            {
                modfd( m_epollfd, m_sockfd, EPOLLOUT );
                return true;
            }
            unmap();
            return false;
        }

        bytes_to_send -= temp;
        bytes_have_send += temp;
        if ( bytes_to_send <= bytes_have_send )
        {
            /* 发送 HTTP 响应成功，根据 HTTP 请求中的 Connection 字段决定是否立即关闭连接 */
            unmap();
            if( m_linger )
            {

```

```

        init();
        modfd( m_epollfd, m_sockfd, EPOLLIN );
        return true;
    }
    else
    {
        modfd( m_epollfd, m_sockfd, EPOLLIN );
        return false;
    }
}
}

/* 往写缓冲中写入待发送的数据 */
bool http_conn::add_response( const char* format, ... )
{
    if( m_write_idx >= WRITE_BUFFER_SIZE )
    {
        return false;
    }
    va_list arg_list;
    va_start( arg_list, format );
    int len = vsnprintf( m_write_buf + m_write_idx, WRITE_BUFFER_SIZE - 1 - m_write_idx,
                         format, arg_list );
    if( len >= ( WRITE_BUFFER_SIZE - 1 - m_write_idx ) )
    {
        return false;
    }
    m_write_idx += len;
    va_end( arg_list );
    return true;
}

bool http_conn::add_status_line( int status, const char* title )
{
    return add_response( "%s %d %s\r\n", "HTTP/1.1", status, title );
}

bool http_conn::add_headers( int content_len )
{
    add_content_length( content_len );
    add_linger();
    add_blank_line();
}

bool http_conn::add_content_length( int content_len )
{
    return add_response( "Content-Length: %d\r\n", content_len );
}

bool http_conn::add_linger()
{
    return add_response( "Connection: %s\r\n", ( m_linger == true ) ? "keep-alive" : "close" );
}

```

```

}

bool http_conn::add_blank_line()
{
    return add_response( "%s", "\r\n" );
}

bool http_conn::add_content( const char* content )
{
    return add_response( "%s", content );
}

/* 根据服务器处理 HTTP 请求的结果，决定返回给客户端的内容 */
bool http_conn::process_write( HTTP_CODE ret )
{
    switch ( ret )
    {
        case INTERNAL_ERROR:
        {
            add_status_line( 500, error_500_title );
            add_headers( strlen( error_500_form ) );
            if ( ! add_content( error_500_form ) )
            {
                return false;
            }
            break;
        }
        case BAD_REQUEST:
        {
            add_status_line( 400, error_400_title );
            add_headers( strlen( error_400_form ) );
            if ( ! add_content( error_400_form ) )
            {
                return false;
            }
            break;
        }
        case NO_RESOURCE:
        {
            add_status_line( 404, error_404_title );
            add_headers( strlen( error_404_form ) );
            if ( ! add_content( error_404_form ) )
            {
                return false;
            }
            break;
        }
        case FORBIDDEN_REQUEST:
        {
            add_status_line( 403, error_403_title );
            add_headers( strlen( error_403_form ) );
            if ( ! add_content( error_403_form ) )

```

```

        {
            return false;
        }
        break;
    }
    case FILE_REQUEST:
    {
        add_status_line( 200, ok_200_title );
        if ( m_file_stat.st_size != 0 )
        {
            add_headers( m_file_stat.st_size );
            m_iv[ 0 ].iov_base = m_write_buf;
            m_iv[ 0 ].iov_len = m_write_idx;
            m_iv[ 1 ].iov_base = m_file_address;
            m_iv[ 1 ].iov_len = m_file_stat.st_size;
            m_iv_count = 2;
            return true;
        }
        else
        {
            const char* ok_string = "<html><body></body></html>";
            add_headers( strlen( ok_string ) );
            if ( !add_content( ok_string ) )
            {
                return false;
            }
        }
    }
    default:
    {
        return false;
    }
}

m_iv[ 0 ].iov_base = m_write_buf;
m_iv[ 0 ].iov_len = m_write_idx;
m_iv_count = 1;
return true;
}

/* 由线程池中的工作线程调用，这是处理 HTTP 请求的入口函数 */
void http_conn::process()
{
    HTTP_CODE read_ret = process_read();
    if ( read_ret == NO_REQUEST )
    {
        modfd( m_epollfd, m_sockfd, EPOLLIN );
        return;
    }

    bool write_ret = process_write( read_ret );
    if ( !write_ret )
    {

```

```

        close_conn();
    }

    modfd( m_epollfd, m_sockfd, EPOLLOUT );
}

```

15.6.2 main 函数

定义好任务类之后，main 函数就变得很简单了，它只需要负责 I/O 读写，如代码清单 15-6 所示。

代码清单 15-6 用线程池实现的 Web 服务器

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <cassert>
#include <sys/epoll.h>

#include "locker.h"
#include "threadpool.h"
#include "http_conn.h"

#define MAX_FD 65536
#define MAX_EVENT_NUMBER 10000

extern int addfd( int epollfd, int fd, bool one_shot );
extern int removefd( int epollfd, int fd );

void addsig( int sig, void( handler )(int), bool restart = true )
{
    struct sigaction sa;
    memset( &sa, '\0', sizeof( sa ) );
    sa.sa_handler = handler;
    if( restart )
    {
        sa.sa_flags |= SA_RESTART;
    }
    sigfillset( &sa.sa_mask );
    assert( sigaction( sig, &sa, NULL ) != -1 );
}

void show_error( int connfd, const char* info )
{
    printf( "%s", info );
}

```

```

        send( connfd, info, strlen( info ), 0 );
        close( connfd );
    }

int main( int argc, char* argv[] )
{
    if( argc <= 2 )
    {
        printf( "usage: %s ip_address port_number\n", basename( argv[0] ) );
        return 1;
    }
    const char* ip = argv[1];
    int port = atoi( argv[2] );

    /* 忽略 SIGPIPE 信号 */
    addsig( SIGPIPE, SIG_IGN );

    /* 创建线程池 */
    threadpool< http_conn *>* pool = NULL;
    try
    {
        pool = new threadpool< http_conn >;
    }
    catch( ... )
    {
        return 1;
    }

    /* 预先为每个可能的客户连接分配一个 http_conn 对象 */
    http_conn* users = new http_conn[ MAX_FD ];
    assert( users );
    int user_count = 0;

    int listenfd = socket( PF_INET, SOCK_STREAM, 0 );
    assert( listenfd >= 0 );
    struct linger tmp = { 1, 0 };
    setsockopt( listenfd, SOL_SOCKET, SO_LINGER, &tmp, sizeof( tmp ) );

    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    ret = bind( listenfd, ( struct sockaddr* )&address, sizeof( address ) );
    assert( ret >= 0 );

    ret = listen( listenfd, 5 );
    assert( ret >= 0 );

    epoll_event events[ MAX_EVENT_NUMBER ];

```

```

int epollfd = epoll_create( 5 );
assert( epollfd != -1 );
addfd( epollfd, listenfd, false );
http_conn::m_epollfd = epollfd;

while( true )
{
    int number = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, -1 );
    if ( ( number < 0 ) && ( errno != EINTR ) )
    {
        printf( "epoll failure\n" );
        break;
    }

    for ( int i = 0; i < number; i++ )
    {
        int sockfd = events[i].data.fd;
        if( sockfd == listenfd )
        {
            struct sockaddr_in client_address;
            socklen_t client_addrlength = sizeof( client_address );
            int connfd = accept( listenfd, ( struct sockaddr* )&client_address,
                &client_addrlength );
            if ( connfd < 0 )
            {
                printf( "errno is: %d\n", errno );
                continue;
            }
            if( http_conn::m_user_count >= MAX_FD )
            {
                show_error( connfd, "Internal server busy" );
                continue;
            }
            /* 初始化客户连接 */
            users[connfd].init( connfd, client_address );
        }
        else if( events[i].events & ( EPOLLRDHUP | EPOLLHUP | EPOLLERR ) )
        {
            /* 如果有异常，直接关闭客户连接 */
            users[sockfd].close_conn();
        }
        else if( events[i].events & EPOLLIN )
        {
            /* 根据读的结果，决定是将任务添加到线程池，还是关闭连接 */
            if( users[sockfd].read() )
            {
                pool->append( users + sockfd );
            }
            else
            {
                users[sockfd].close_conn();
            }
        }
    }
}

```

```
else if( events[i].events & EPOLLOUT )
{
    /* 根据写的结果，决定是否关闭连接 */
    if( !users[sockfd].write() )
    {
        users[sockfd].close_conn();
    }
    else
    {}
}

close(epollfd);
close(listenfd);
delete [] users;
delete pool;
return 0;
}
```



第三篇

高性能服务器优化与监测

第 16 章 服务器调制、调试和测试

第 17 章 系统监测工具

第 16 章 服务器调制、调试和测试

在前面的章节中，我们已经细致地探讨了服务器编程的诸多方面。现在我们要从系统的角度来优化、改进服务器，这包括 3 个方面的内容：系统调制、服务器调试和压力测试。

Linux 平台的一个优秀特性是内核微调，即我们可以通过修改文件的方式来调整内核参数。16.2 节将讨论与服务器性能相关的部分内核参数。这些内核参数中，系统或进程能打开的最大文件描述符数尤其重要，所以我们在 16.1 节单独讨论之。

在服务器的开发过程中，我们可能碰到各种意想不到的错误。一种调试方法是用 `tcpdump` 抓包，正如本书前面章节介绍的那样。不过这种方法主要用于分析程序的输入和输出。对于服务器的逻辑错误，更方便的调试方法是使用 `gdb` 调试器。我们将在 16.3 节讨论如何用 `gdb` 调试多进程和多线程程序。

编写压力测试工具通常被认为是服务器开发的一个部分。压力测试工具模拟现实世界中高并发的客户请求，以测试服务器在高压状态下的稳定性。我们将在 16.4 节给出一个简单的压力测试程序。

16.1 最大文件描述符数

文件描述符是服务器程序的宝贵资源，几乎所有的系统调用都是和文件描述符打交道。系统分配给应用程序的文件描述符数量是有限制的，所以我们必须总是关闭那些已经不再使用的文件描述符，以释放它们占用的资源。比如作为守护进程运行的服务器程序就应该总是关闭标准输入、标准输出和标准错误这 3 个文件描述符。

Linux 对应用程序能打开的最大文件描述符数量有两个层次的限制：用户级限制和系统级限制。用户级限制是指目标用户运行的所有进程总共能打开的文件描述符数；系统级的限制是指所有用户总共能打开的文件描述符数。

下面这个命令是最常用的查看用户级文件描述符数限制的方法：

```
$ ulimit -n
```

我们可以通过如下方式将用户级文件描述符数限制设定为 `max-file-number`：

```
$ ulimit -SHn max-file-number
```

不过这种设置是临时的，只在当前的 session 中有效。为永久修改用户级文件描述符数限制，可以在 `/etc/security/limits.conf` 文件中加入如下两项：

```
* hard nofile max-file-number  
* soft nofile max-file-number
```

第一行是指系统的硬限制，第二行是软限制。我们在 7.4 节讨论过这两种资源限制。如果要修改系统级文件描述符数限制，则可以使用如下命令：

```
sysctl -w fs.file-max=max-file-number
```

不过该命令也是临时更改系统限制。要永久更改系统级文件描述符数限制，则需要在 /etc/sysctl.conf 文件中添加如下一项：

```
fs.file-max=max-file-number
```

然后通过执行 sysctl -p 命令使更改生效。

16.2 调整内核参数

几乎所有的内核模块，包括内核核心模块和驱动程序，都在 /proc/sys 文件系统下提供了某些配置文件以供用户调整模块的属性和行为。通常一个配置文件对应一个内核参数，文件名就是参数的名字，文件的内容是参数的值。我们可以通过命令 sysctl -a 查看所有这些内核参数。本节将讨论其中和网络编程关系较为紧密的部分内核参数。

16.2.1 /proc/sys/fs 目录下的部分文件

/proc/sys/fs 目录下的内核参数都与文件系统相关。对于服务器程序来说，其中最重要的是如下两个参数：

- /proc/sys/fs/file-max，系统级文件描述符数限制。直接修改这个参数和 16.1 节讨论的修改方法有相同的效果（不过这是临时修改）。一般修改 /proc/sys/fs/file-max 后，应用程序需要把 /proc/sys/fs/inode-max 设置为新 /proc/sys/fs/file-max 值的 3 ~ 4 倍，否则可能导致 i 节点数不够用。
- /proc/sys/fs/epoll/max_user_watches，一个用户能够往 epoll 内核事件表中注册的事件的总量。它是指该用户打开的所有 epoll 实例总共能监听的事件数目，而不是单个 epoll 实例能监听的事件数目。往 epoll 内核事件表中注册一个事件，在 32 位系统上大概消耗 90 字节的内核空间，在 64 位系统上则消耗 160 字节的内核空间。所以，这个内核参数限制了 epoll 使用的内核内存总量。

16.2.2 /proc/sys/net 目录下的部分文件

内核中网络模块的相关参数都位于 /proc/sys/net 目录下，其中和 TCP/IP 协议相关的参数主要位于如下三个子目录中：core、ipv4 和 ipv6。在前面的章节中，我们已经介绍过这些子目录下的很多参数的含义，现在再总结一下和服务器性能相关的部分参数。

- /proc/sys/net/core/somaxconn，指定 listen 监听队列里，能够建立完整连接从而进入 ESTABLISHED 状态的 socket 的最大数目。读者不妨修改该参数并重新运行代码清单 5-3，看看其影响。
- /proc/sys/net/ipv4/tcp_max_syn_backlog，指定 listen 监听队列里，能够转移至 ESTAB-

LISHED 或者 SYN_RCVD 状态的 socket 的最大数目。

- ❑ /proc/sys/net/ipv4/tcp_wmem，它包含 3 个值，分别指定一个 socket 的 TCP 写缓冲区的最小值、默认值和最大值。
- ❑ /proc/sys/net/ipv4/tcp_rmem，它包含 3 个值，分别指定一个 socket 的 TCP 读缓冲区的最小值、默认值和最大值。在代码清单 3-6 中，我们正是通过修改这个参数来改变接收通告窗口大小的。
- ❑ /proc/sys/net/ipv4/tcp_syncookies，指定是否打开 TCP 同步标签（syncookie）。同步标签通过启动 cookie 来防止一个监听 socket 因不停地重复接收来自同一个地址的连接请求（同步报文段），而导致 listen 监听队列溢出（所谓的 SYN 风暴）。

除了通过直接修改文件的方式来修改这些系统参数外，我们也可以使用 sysctl 命令来修改它们。这两种修改方式都是临时的。永久的修改方法是在 /etc/sysctl.conf 文件中加入相应网络参数及其数值，并执行 sysctl -p 使之生效，就像修改系统最大允许打开的文件描述符数那样。

16.3 gdb 调试

Linux 程序员必然都使用过 gdb 调试器来调试程序。我们也假设读者懂得基本的 gdb 调试方法，比如设置断点，查看变量等。这一节要讨论的是如何使用 gdb 来调试多进程和多线程程序，因为这是后台程序调试不可避免而又比较困难的部分。

16.3.1 用 gdb 调试多进程程序

如果一个进程通过 fork 系统调用创建了子进程，gdb 会继续调试原来的进程，子进程则正常运行。那么该如何调试子进程呢？常用的方法有如下两种。

1. 单独调试子进程

子进程从本质上说也是一个进程，因此我们可以用通用的 gdb 调试方法来调试它。举例来说，如果要调试代码清单 15-2 描述的 CGI 进程池服务器的某一个子进程，则我们可以先运行服务器，然后找到目标子进程的 PID，再将其附加（attach）到 gdb 调试器上，具体操作如代码清单 16-1 所示。

代码清单 16-1 通过附加子进程的 PID 来调试子进程

```
$ ./cgisrv 127.0.0.1 12345
$ ps -ef | grep cgisrv
shuang    4182  3601  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4183  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4184  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4185  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4186  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4187  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4188  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4189  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
shuang    4190  4182  0 12:25 pts/4    00:00:00 ./cgisrv 127.0.0.1 12345
```

```

$ gdb
(gdb) attach 4183 /* 将子进程 4183 附加到 gdb 调试器 */
Attaching to process 4183
Reading symbols from /home/shuang/codes/pool_process/cgisrv...done.
Reading symbols from /usr/lib/libstdc++.so.6...Reading symbols from /usr/lib/
debug/usr/lib/libstdc++.so.6.0.16.debug...done.
done.
Loaded symbols for /usr/lib/libstdc++.so.6
Reading symbols from /lib/libm.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libm.so.6
Reading symbols from /lib/libgcc_s.so.1...Reading symbols from /usr/lib/debug/
lib/libgcc_s-4.6.2-20111027.so.1.debug...done.
done.
Loaded symbols for /lib/libgcc_s.so.1
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
0x0047c416 in __kernel_vsyscall ()
(gdb) b processpool.h:264 /* 设置子进程中的断点 */
Breakpoint 1 at 0x8049787: file processpool.h, line 264.
(gdb) c
Continuing.

/* 接下来从另一个终端使用 telnet 127.0.0.1 12345 来连接服务器并发送一些数据，调试器就按照我们预期的，在断点处暂停 */
Breakpoint 1, processpool<cgi_conn>::run_child (this=0x9a47008) at processpool.h:264
264                      users[sockfd].process();
(gdb) bt
#0  processpool<cgi_conn>::run_child (this=0x9a47008) at processpool.h:264
#1  0x080491fe in processpool<cgi_conn>::run (this=0x9a47008) at processpool.h:169
#2  0x08048ef9 in main (argc=3, argv=0xbfbcb74) at main.cpp:138
(gdb)

```

2. 使用调试器选项 follow-fork-mode

gdb 调试器的选项 follow-fork-mode 允许我们选择程序在执行 fork 系统调用后是继续调试父进程还是调试子进程。其用法如下：

```
(gdb) set follow-fork-mode mode
```

其中，mode 的可选值是 parent 和 child，分别表示调试父进程和子进程。还是使用前面的例子，这次考虑使用 follow-fork-mode 选项来调试子进程，如代码清单 16-2 所示。

代码清单 16-2 使用 follow-fork-mode 选项调试子进程

```

$ gdb ./cgisrv
(gdb) set follow-fork-mode child
(gdb) b processpool.h:264
Breakpoint 1 at 0x8049787: file processpool.h, line 264.
(gdb) r 127.0.0.1 12345
Starting program: /home/shuang/codes/pool_process/cgisrv 127.0.0.1 12345
[New process 4148]
send request to child 0

```

```
[Switching to process 4148]

Breakpoint 1, processpool<cgi_conn>::run_child (this=0x804c008) at processpool.h:264
264          users[sockfd].process();
Missing separate debuginfos, use: debuginfo-install glibc-2.14.90-24.
fc16.6.i686
(gdb) bt
#0  processpool<cgi_conn>::run_child (this=0x804c008) at processpool.h:264
#1  0x080491fe in processpool<cgi_conn>::run (this=0x804c008) at processpool.h:169
#2  0x08048ef9 in main (argc=3, argv=0xbfffff4e4) at main.cpp:138
(gdb)
```

16.3.2 用 gdb 调试多线程程序

gdb 有一组命令可辅助多线程程序的调试。下面我们仅列举其中常用的一些：

- **info threads**，显示当前可调试的所有线程。gdb 会为每个线程分配一个 ID，我们可以使用这个 ID 来操作对应的线程。ID 前面有“*”号的线程是当前被调试的线程。
- **thread ID**，调试目标 ID 指定的线程。
- **set scheduler-locking [off|on|step]**。调试多线程程序时，默认除了被调试的线程在执行外，其他线程也在继续执行，但有的时候我们希望只让被调试的线程运行。这可以通过这个命令来实现。该命令设置 scheduler-locking 的值：off 表示不锁定任何线程，即所有线程都可以继续执行，这是默认值；on 表示只有当前被调试的线程会继续执行；step 表示在单步执行的时候，只有当前线程会执行。

举例来说，如果要依次调试代码清单 15-6 所描述的 Web 服务器（名为 websrv）的父线程和子线程，则可以采用代码清单 16-3 所示的方法。

代码清单 16-3 独立调试父线程和子线程

```
$ gdb ./websrv
(gdb) b main.cpp:130 /* 设置父线程中的断点 */
Breakpoint 1 at 0x80498d3: file main.cpp, line 130.
(gdb) b threadpool.h:105 /* 设置子线程中的断点 */
Breakpoint 2 at 0x804a10b: file threadpool.h, line 105.
(gdb) r 127.0.0.1 12345
Starting program: /home/webtop/codes/pool_thread/websrv 127.0.0.1 12345
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/libthread_db.so.1".
create the 0th thread
[New Thread 0xb7fe1b40 (LWP 5756)]
```

/* 从另一个终端使用 telnet 127.0.0.1 12345 来连接服务器并发送一些数据，调试器就按照我们预期的，在断点处暂停 */

Breakpoint 1, main (argc=3, argv=0xbfffff4e4) at main.cpp:130
130 if(users[sockfd].read())
(gdb) info threads /* 查看线程信息。当前被调试的是主线程，其 ID 为 1 */
Id Target Id Frame
2 Thread 0xb7fe1b40 (LWP 5756) "websrv" 0x00111416 in __kernel_vsyscall ()

```

* 1 Thread 0xb7fe3700 (LWP 5753) "websrv" main (argc=3, argv=0xbffff4e4) at
main.cpp:130
(gdb) set scheduler-locking on /* 不执行其他线程，锁定调试对象 */
(gdb) n /* 下面的操作都将执行父线程的代码 */
132 pool->append( users + sockfd );
(gdb) n
103 for ( int i = 0; i < number; i++ )
(gdb)
94 while( true )
(gdb)
96 int number = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, -1 );
(gdb)
^C
Program received signal SIGINT, Interrupt.
0x00111416 in __kernel_vsyscall ()
(gdb) thread 2 /* 将调试切换到子线程，其 ID 为 2 */
[Switching to thread 2 (Thread 0xb7fe1b40 (LWP 5756))]
#0 0x00111416 in __kernel_vsyscall ()
(gdb) bt /* 显示子线程的调用栈 */
#0 0x00111416 in __kernel_vsyscall ()
#1 0x44d91c05 in sem_wait@@GLIBC_2.1 () from /lib/libpthread.so.0
#2 0x08049aff in sem::wait (this=0x804e034) at locker.h:24
#3 0x0804a0db in threadpool<http_conn>::run (this=0x804e008) at threadpool.h:98
#4 0x08049f8f in threadpool<http_conn>::worker (arg=0x804e008) at threadpool.h:89
#5 0x44d8bcd3 in start_thread () from /lib/libpthread.so.0
#6 0x44cc8a2e in clone () from /lib/libc.so.6
(gdb) n /* 下面的操作都将执行子线程的代码 */
Single stepping until exit from function __kernel_vsyscall,
which has no line number information.
0x44d91c05 in sem_wait@@GLIBC_2.1 () from /lib/libpthread.so.0
(gdb)

```

最后，关于调试进程池和线程池程序的一个不错的方法，是先将池中的进程个数或线程个数减少至 1，以观察程序的逻辑是否正确，比如代码清单 16-3 就是这样做的；然后逐步增加进程或线程的数量，以调试进程或线程的同步是否正确。

16.4 压力测试

压力测试程序有很多种实现方式，比如 I/O 复用方式，多线程、多进程并发编程方式，以及这些方式的结合使用。不过，单纯的 I/O 复用方式的施压程度是最高的，因为线程和进程的调度本身也是要占用一定 CPU 时间的。因此，我们将使用 epoll 来实现一个通用的服务器压力测试程序，如代码清单 16-4 所示。

代码清单 16-4 服务器压力测试程序

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <unistd.h>
```

```

#include <sys/types.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

/* 每个客户连接不停地向服务器发送这个请求 */
static const char* request = "GET http://localhost/index.html HTTP/1.1\r\n"
Connection: keep-alive\r\n\r\nxxxxxxxxxxxxxx";

int setnonblocking( int fd )
{
    int old_option = fcntl( fd, F_GETFL );
    int new_option = old_option | O_NONBLOCK;
    fcntl( fd, F_SETFL, new_option );
    return old_option;
}

void addfd( int epoll_fd, int fd )
{
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLOUT | EPOLLET | EPOLLERR;
    epoll_ctl( epoll_fd, EPOLL_CTL_ADD, fd, &event );
    setnonblocking( fd );
}

/* 向服务器写入 len 字节的数据 */
bool write_nbytes( int sockfd, const char* buffer, int len )
{
    int bytes_write = 0;
    printf( "write out %d bytes to socket %d\n", len, sockfd );
    while( 1 )
    {
        bytes_write = send( sockfd, buffer, len, 0 );
        if ( bytes_write == -1 )
        {
            return false;
        }
        else if ( bytes_write == 0 )
        {
            return false;
        }

        len -= bytes_write;
        buffer = buffer + bytes_write;
        if ( len <= 0 )
        {
            return true;
        }
    }
}

```

```

/* 从服务器读取数据 */
bool read_once( int sockfd, char* buffer, int len )
{
    int bytes_read = 0;
    memset( buffer, '\0', len );
    bytes_read = recv( sockfd, buffer, len, 0 );
    if ( bytes_read == -1 )
    {
        return false;
    }
    else if ( bytes_read == 0 )
    {
        return false;
    }
    printf( "read in %d bytes from socket %d with content: %s\n",
            bytes_read,
            sockfd, buffer );

    return true;
}

/* 向服务器发起 num 个 TCP 连接，我们可以通过改变 num 来调整测试压力 */
void start_conn( int epoll_fd, int num, const char* ip, int port )
{
    int ret = 0;
    struct sockaddr_in address;
    bzero( &address, sizeof( address ) );
    address.sin_family = AF_INET;
    inet_pton( AF_INET, ip, &address.sin_addr );
    address.sin_port = htons( port );

    for ( int i = 0; i < num; ++i )
    {
        sleep( 1 );
        int sockfd = socket( PF_INET, SOCK_STREAM, 0 );
        printf( "create 1 sock\n" );
        if( sockfd < 0 )
        {
            continue;
        }

        if ( connect( sockfd, ( struct sockaddr* )&address, sizeof( address ) ) == 0 )
        {
            printf( "build connection %d\n", i );
            addfd( epoll_fd, sockfd );
        }
    }
}

void close_conn( int epoll_fd, int sockfd )
{
    epoll_ctl( epoll_fd, EPOLL_CTL_DEL, sockfd, 0 );
    close( sockfd );
}

```

```

int main( int argc, char* argv[] )
{
    assert( argc == 4 );
    int epoll_fd = epoll_create( 100 );
    start_conn( epoll_fd, atoi( argv[ 3 ] ), argv[1], atoi( argv[2] ) );
    epoll_event events[ 10000 ];
    char buffer[ 2048 ];
    while ( 1 )
    {
        int fds = epoll_wait( epoll_fd, events, 10000, 2000 );
        for ( int i = 0; i < fds; i++ )
        {
            int sockfd = events[i].data.fd;
            if ( events[i].events & EPOLLIN )
            {
                if ( ! read_once( sockfd, buffer, 2048 ) )
                {
                    close_conn( epoll_fd, sockfd );
                }
                struct epoll_event event;
                event.events = EPOLLOUT | EPOLLET | EPOLLERR;
                event.data.fd = sockfd;
                epoll_ctl( epoll_fd, EPOLL_CTL_MOD, sockfd, &event );
            }
            else if( events[i].events & EPOLLOUT )
            {
                if ( ! write_nbytes( sockfd, request, strlen( request ) ) )
                {
                    close_conn( epoll_fd, sockfd );
                }
                struct epoll_event event;
                event.events = EPOLLIN | EPOLLET | EPOLLERR;
                event.data.fd = sockfd;
                epoll_ctl( epoll_fd, EPOLL_CTL_MOD, sockfd, &event );
            }
            else if( events[i].events & EPOLLERR )
            {
                close_conn( epoll_fd, sockfd );
            }
        }
    }
}

```

下面考虑使用该压力测试程序（名为 `stress_test`）来测试代码清单 15-6 所描述的 Web 服务器的稳定性。我们先在测试机器 `ernest-laptop` 上运行 `websrv`，然后从 `Kongming20` 上执行 `stress_test`，向 `websrv` 服务器发起 1000 个连接。具体操作如下：

```

$ ./websrv 192.168.1.108 12345 # 在 ernest-laptop 上执行，监听端口 12345
$ ./stress_test 192.168.1.108 12345 1000 # 在 Kongming20 上执行

```

如果 `websrv` 服务器程序足够稳定，那么 `websrv` 和 `stress_test` 这两个程序将一直运行下去，并不断交换数据。

第 17 章 系统监测工具

Linux 提供了很多有用的工具，以方便开发人员调试和测评服务器程序。娴熟的网络程序员在开发服务器程序的整个过程中，都将不断地使用这些工具中的一个或者多个来监测服务器行为。其中的某些工具更是黑客们常用的利器。

本章将讨论几个最常用的工具：`tcpdump`、`nc`、`strace`、`lsof`、`netstat`、`vmstat`、`ifstat` 和 `mpstat`。这些工具都支持很多种选项，不过我们的讨论仅限于其中最常用、最实用的那些。

17.1 `tcpdump`

`tcpdump` 是一款经典的网络抓包工具。即使在今天，我们拥有像 Wireshark 这样更易于使用和掌握的抓包工具，`tcpdump` 仍然是网络程序员的必备利器。

`tcpdump` 给使用者提供了大量的选项，用以过滤数据包或者定制输出格式。前面章节中我们介绍过其中的一些，现在我们把常见的选项总结如下：

- ❑ `-n`，使用 IP 地址表示主机，而不是主机名；使用数字表示端口号，而不是服务名称。
- ❑ `-i`，指定要监听的网卡接口。“`-i any`”表示抓取所有网卡接口上的数据包。
- ❑ `-v`，输出一个稍微详细的信息，例如，显示 IP 数据包中的 TTL 和 TOS 信息。
- ❑ `-t`，不打印时间戳。
- ❑ `-e`，显示以太网帧头部信息。
- ❑ `-c`，仅抓取指定数量的数据包。
- ❑ `-x`，以十六进制数显示数据包的内容，但不显示包中以太网帧的头部信息。
- ❑ `-X`，与 `-x` 选项类似，不过还打印每个十六进制字节对应的 ASCII 字符。
- ❑ `-XX`，与 `-X` 相同，不过还打印以太网帧的头部信息。
- ❑ `-s`，设置抓包时的抓取长度。当数据包的长度超过抓取长度时，`tcpdump` 抓取到的将是被截断的数据包。在 4.0 以及之前的版本中，默认的抓包长度是 68 字节。这对于 IP、TCP 和 UDP 等协议就已经足够了，但对于像 DNS、NFS 这样的协议，68 字节通常不能容纳一个完整的数据包。比如我们在 1.6.3 小节抓取 DNS 数据包时，就使用了 `-s` 选项（测试机器 `ernest-laptop` 上，`tcpdump` 的版本是 4.0.0）。不过 4.0 之后的版本，默认的抓包长度被修改为 65 535 字节，因此我们不用再担心抓包长度的问题了。
- ❑ `-S`，以绝对值来显示 TCP 报文段的序号，而不是相对值。
- ❑ `-w`，将 `tcpdump` 的输出以特殊的格式定向到某个文件。
- ❑ `-r`，从文件读取数据包信息并显示之。

除了使用选项外，`tcpdump` 还支持用表达式来进一步过滤数据包。`tcpdump` 表达式的操

作数分为 3 种：类型（type）、方向（dir）和协议（proto）。下面依次介绍之。

- 类型，解释其后面紧跟着的参数的含义。tcpdump 支持的类型包括 host、net、port 和 portrange。它们分别指定主机名（或 IP 地址），用 CIDR 方法表示的网络地址，端口号以及端口范围。比如，要抓取整个 1.2.3.0/255.255.255.0 网络上的数据包，可以使用如下命令：

```
$ tcpdump net 1.2.3.0/24
```

- 方向，src 指定数据包的发送端，dst 指定数据包的目的端。比如要抓取进入端口 13579 的数据包，可以使用如下命令：

```
$ tcpdump dst port 13579
```

- 协议，指定目标协议。比如要抓取所有 ICMP 数据包，可以使用如下命令：

```
$ tcpdump icmp
```

当然，我们还可以使用逻辑操作符来组织上述操作数以创建更复杂的表达式。tcpdump 支持的逻辑操作符和编程语言中的逻辑操作符完全相同，包括 and（或者 &&）、or（或者 ||）、not（或者 !）。比如要抓取主机 ernest-laptop 和所有非 Kongming20 的主机之间交换的 IP 数据包，可以使用如下命令：

```
$ tcpdump ip host ernest-laptop and not Kongming20
```

如果表达式比较复杂，那么我们可以使用括号将它们分组。不过在使用括号时，我们要么使用反斜杠 “\” 对它转义，要么用单引号 ‘’ 将其括住，以避免它被 shell 所解释。比如要抓取来自主机 10.0.2.4，目标端口是 3389 或 22 的数据包，可以使用如下命令：

```
$ tcpdump 'src 10.0.2.4 and (dst port 3389 or 22)'
```

此外，tcpdump 还允许直接使用数据包中的部分协议字段的内容来过滤数据包。比如，仅抓取 TCP 同步报文段，可使用如下命令：

```
$ tcpdump 'tcp[13] & 2 != 0'
```

这是因为 TCP 头部的第 14 个字节的第 2 个位正是同步标志。该命令也可以表示为：

```
$ tcpdump 'tcp[tcpflags] & tcp-syn != 0'.
```

最后，tcpdump 的具体输出格式除了与选项有关外，还与协议有关。前文中我们讨论过 IP、TCP、ICMP、DNS 等协议的 tcpdump 输出格式。关于其他协议的 tcpdump 输出格式，请读者自己参考 tcpdump 的 man 手册，本书不再赘述。

17.2 lsof

lsof（list open file）是一个列出当前系统打开的文件描述符的工具。通过它我们可以了解感兴趣的进程打开了哪些文件描述符，或者我们感兴趣的文件描述符被哪些进程打开了。

lsof 命令常用的选项包括：

- -i，显示 socket 文件描述符。该选项的使用方法是：

```
$ lsof -i [46] [protocol] [@hostname|ipaddr] [:service|port]
```

其中，4 表示 IPv4 协议，6 表示 IPv6 协议； protocol 指定传输层协议，可以是 TCP 或者 UDP； hostname 指定主机名； ipaddr 指定主机的 IP 地址； service 指定服务名； port 指定端口号。比如，要显示所有连接到主机 192.168.1.108 的 ssh 服务的 socket 文件描述符，可以使用命令：

```
$ lsof -i@192.168.1.108:22
```

如果 -i 选项后不指定任何参数，则 lsof 命令将显示所有 socket 文件描述符。

- -u，显示指定用户启动的所有进程打开的所有文件描述符。

- -c，显示指定的命令打开的所有文件描述符。比如要查看 websrv 程序打开了哪些文件描述符，可以使用如下命令：

```
$ lsof -c websrv
```

- -p，显示指定进程打开的所有文件描述符。

- -t，仅显示打开了目标文件描述符的进程的 PID。

我们还可以直接将文件名作为 lsof 命令的参数，以查看哪些进程打开了该文件。

下面介绍一个实例：查看 websrv 服务器打开了哪些文件描述符。具体操作如代码清单 17-1 所示。

代码清单 17-1 用 lsof 命令查看 websrv 服务器打开的文件描述符

```
$ ps -ef | grep websrv # 先获取 websrv 程序的进程号
shuang    6346  5439  0 23:41 pts/3    00:00:00 ./websrv 127.0.0.1 13579
$ sudo lsof -p 6346 # 用 -p 选项指定进程号
COMMAND PID  USER   FD   TYPE DEVICE SIZE/OFF NODE     NAME
websrv  6346 shuang cwd DIR  8,3    4096      1199520 /home/shuang/codes/pool_thread
websrv  6346 shuang rtd DIR  8,3    4096      2          /
websrv  6346 shuang txt REG  8,3    64817     1199765 /home/shuang/codes/pool_thread/websrv
websrv  6346 shuang mem REG  8,3    157200    1319677 /lib/ld-2.14.90.so
websrv  6346 shuang mem REG  8,3    2000316  1319678 /lib/libc-2.14.90.so
websrv  6346 shuang mem REG  8,3    135556   1319682 /lib/libpthread-2.14.90.so
websrv  6346 shuang mem REG  8,3    208320   1319681 /lib/libm-2.14.90.so
websrv  6346 shuang mem REG  8,3    115376   1319685 /lib/libgcc_s-4.6.2-20111027.so.1
websrv  6346 shuang mem REG  8,3    948524   814873  /usr/lib/libstdc++.so.6.0.16
websrv  6346 shuang 0u  CHR  136,3  0t0       6          /dev/pts/3
websrv  6346 shuang 1u  CHR  136,3  0t0       6          /dev/pts/3
websrv  6346 shuang 2u  CHR  136,3  0t0       6          /dev/pts/3
websrv  6346 shuang 3u  IPv4 43816  0t0      TCP      localhost:13579
websrv  6346 shuang 4u  0000 0,9      0        4447  anon_inode
```

lsof 命令的输出内容相当丰富，其中每行内容都包含如下字段：

- COMMAND，执行程序所使用的终端命令（默认仅显示前 9 个字符）。
- PID，文件描述符所属进程的 PID。
- USER，拥有该文件描述符的用户的用户名。
- FD，文件描述符的描述。其中 cwd 表示进程的工作目录，rtd 表示用户的根目录，txt 表示进程运行的程序代码，mem 表示直接映射到内存中的文件（本例中都是动态库）。有的 FD 是以“数字 + 访问权限”表示的，其中数字是文件描述符的具体数值，访问权限包括 r（可读）、w（可写）和 u（可读可写）。在本例中，0u、1u、2u 分别表示标准输入、标准输出和标准错误输出；3u 表示处于 LISTEN 状态的监听 socket；4u 表示 epoll 内核事件表对应的文件描述符。
- TYPE，文件描述符的类型。其中 DIR 是目录，REG 是普通文件，CHR 是字符设备文件，IPv4 是 IPv4 类型的 socket 文件描述符，0000 是未知类型。更多文件描述符的类型请参考 lsof 命令的 man 手册，这里不再赘述。
- DEVICE，文件所属设备。对于字符设备和块设备，其表示方法是“主设备号，次设备号”。由代码清单 17-1 可见，测试机器上的程序文件和动态库都存放在设备“8,3”中。其中，“8”表示这是一个 SCSI 硬盘；“3”表示这是该硬盘上的第 3 个分区，即 sda3。websrv 程序的标准输入、标准输出和标准错误输出对应的设备是“136,3”。其中，“136”表示这是一个伪终端；“3”表示它是第 3 个伪终端，即 /dev/pts/3。关于设备编号的更多细节，请参考文档 <http://www.kernel.org/pub/linux/docs/lanana/device-list/devices-2.6.txt>。对于 FIFO 类型的文件，比如管道和 socket，该字段将显示一个内核引用目标文件的地址，或者是其 i 节点号。
- SIZE/OFF，文件大小或者偏移值。如果该字段显示为“0t*”或者“0x*”，就表示这是一个偏移值，否则就表示这是一个文件大小。对字符设备或者 FIFO 类型的文件定义文件大小没有意义，所以该字段将显示一个偏移值。
- NODE，文件的 i 节点号。对于 socket，则显示为协议类型，比如“TCP”。
- NAME，文件的名字。

如果我们使用 telnet 命令向 websrv 服务器发起一个连接，则再次执行代码清单 17-1 中的 lsof 命令时，其输出将多出如下一行：

```
websrv 6346 shuang 5u IPv4 44288 0t0 TCP localhost:13579->localhost:48215
(ESTABLISHED)
```

该输出表示服务器打开了一个 IPv4 类型的 socket，其值是 5，且它处于 ESTABLISHED 状态。该 socket 对应的连接的本端 socket 地址是 (127.0.0.1, 13579)，远端 socket 地址则是 (127.0.0.1, 48215)。

17.3 nc

nc (netcat) 命令短小精干、功能强大，有着“瑞士军刀”的美誉。它主要被用来快速构

建网络连接。我们可以让它以服务器方式运行，监听某个端口并接收客户连接，因此它可用来调试客户端程序。我们也可以使之以客户端方式运行，向服务器发起连接并收发数据，因此它可以用来调试服务器程序，此时它有点像 telnet 程序。

nc 命令常用的选项包括：

- ❑ -i，设置数据包传送的时间间隔。
- ❑ -l，以服务器方式运行，监听指定的端口。nc 命令默认以客户端方式运行。
- ❑ -k，重复接受并处理某个端口上的所有连接，必须与 -l 选项一起使用。
- ❑ -n，使用 IP 地址表示主机，而不是主机名；使用数字表示端口号，而不是服务名称。
- ❑ -p，当 nc 命令以客户端方式运行时，强制其使用指定的端口号。3.4.2 小节中我们就曾使用过该选项。
- ❑ -s，设置本地主机发送出的数据包的 IP 地址。
- ❑ -C，将 CR 和 LF 两个字符作为行结束符。
- ❑ -U，使用 UNIX 本地域协议通信。
- ❑ -u，使用 UDP 协议。nc 命令默认使用的传输层协议是 TCP 协议。
- ❑ -w，如果 nc 客户端在指定的时间内未检测到任何输入，则退出。
- ❑ -X，当 nc 客户端和代理服务器通信时，该选项指定它们之间使用的通信协议。目前 nc 支持的代理协议包括“4”(SOCKS v.4), “5”(SOCKS v.5) 和“connect”(HTTPS proxy)。nc 默认使用的代理协议是 SOCKS v.5。
- ❑ -x，指定目标代理服务器的 IP 地址和端口号。比如，要从 Kongming20 连接到 ernest-laptop 上的 squid 代理服务器，并通过它来访问 www.baidu.com 的 Web 服务，可以使用如下命令：

```
$ nc -x ernest-laptop:1080 -X connect www.baidu.com 80
```

- ❑ -z，扫描目标机器上的某个或某些服务是否开启（端口扫描）。比如，要扫描机器 ernest-laptop 上端口号在 20 ~ 50 之间的服务，可以使用如下命令：

```
$ nc -z ernest-laptop 20-50
```

举例来说，我们可以使用如下方式来连接 websrv 服务器并向它发送数据：

```
$ nc -C 127.0.0.1 13579 (服务器监听端口 13579)
GET http://localhost/a.html HTTP/1.1 (回车)
Host: localhost (回车)
(回车)
HTTP/1.1 404 Not Found
Content-Length: 49
Connection: close

The requested file was not found on this server.
```

这里我们使用了 -C 选项，这样每次我们按下回车键向服务器发送一行数据时，nc 客户端程序都会给服务器额外发送一个 <CR><LF>，而这正是 websrv 服务器期望的 HTTP 行结

束符。发送完第三行数据之后，我们得到了服务器的响应，内容正是我们期望的：服务器没有找到被请求的资源文件 a.html。可见，nc 命令是一个很方便的快速测试工具，通过它我们能很快找出服务器的逻辑错误。

17.4 strace

strace 是测试服务器性能的重要工具。它跟踪程序运行过程中执行的系统调用和接收到的信号，并将系统调用名、参数、返回值及信号名输出到标准输出或者指定的文件。

strace 命令常用的选项包括：

- -c，统计每个系统调用执行时间、执行次数和出错次数。
- -f，跟踪由 fork 调用生成的子进程。
- -t，在输出的每一行信息前加上时间信息。
- -e，指定一个表达式，用来控制如何跟踪系统调用（或接收到的信号，下同）。其格式是：

```
[qualifier=[!]value1[,value2]...]
```

qualifier 可以是 trace、abbrev、verbose、raw、signal、read 和 write 中之一，默认是 trace。value 是用于进一步限制被跟踪的系统调用的符号或数值。它的两个特殊取值是 all 和 none，分别表示跟踪所有由 qualifier 指定类型的系统调用和不跟踪任何该类型的系统调用。关于 value 的其他取值，我们简单地列举一些：

- ◆ -e trace=set，只跟踪指定的系统调用。例如，-e trace=open, close, read, write 表示只跟踪 open、close、read 和 write 这四种系统调用。
- ◆ -e trace=file，只跟踪与文件操作相关的系统调用。
- ◆ -e trace=process，只跟踪与进程控制相关的系统调用。
- ◆ -e trace=network，只跟踪与网络相关的系统调用。
- ◆ -e trace=signal，只跟踪与信号相关的系统调用。
- ◆ -e trace=ipc，只跟踪与进程间通信相关的系统调用。
- ◆ -e signal=set，只跟踪指定的信号。比如，-e signal!=SIGIO 表示跟踪除 SIGIO 之外的所有信号。
- ◆ -e read=set，输出从指定文件中读入的数据。例如，-e read=3, 5 表示输出所有从文件描述符 3 和 5 读入的数据。
- -o，将 strace 的输出写入指定的文件。

strace 命令的每一行输出都包含这些字段：系统调用名称、参数和返回值。比如下面的示例：

```
$ strace cat /dev/null
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
```

这行输出表示：程序“cat /dev/null”在运行过程中执行了 open 系统调用。open 调用以

只读的方式打开了大文件 /dev/null，然后返回了一个值为 3 的文件描述符。需要注意的是，该示例命令将输出很多内容，这里我们省略了很多次要的信息，在后面的实例中，我们也仅显示主题相关的内容。

当系统调用发生错误时，strace 命令将输出错误标识和描述，比如下面的示例：

```
$ strace cat /foo/bar
open("/foo/bar", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
```

strace 命令对不同的参数类型将有不同的输出方式，比如：

- 对于 C 风格的字符串，strace 将输出字符串的内容。默认的最大输出长度是 32 字节，过长的部分 strace 会使用“...”省略。比如，ls -l 命令在运行过程中将读取 /etc/passwd 文件：

```
$ strace ls -l
read(4, "root:x:0:0:root:/root:/bin/bash\n...", 4096) = 2342
```

需要注意的是，文件名并不被 strace 当作 C 风格的字符串，其内容总是被完整地输出。

- 对于结构体，strace 将用“{}”输出该结构体的每个字段，并用“,”将每个字段隔开。对于字段较多的结构体，strace 将用“...”省略部分输出。比如：

```
$ strace ls -l /dev/null
lstat64("/dev/null", {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
```

上面的 strace 输出显示，lstat64 系统调用的第一个参数是字符串输入参数“/dev/null”；第二个参数则是 stat 结构体类型的输出参数（指针），strace 仅显示了该结构体参数的两个字段：st_mode 和 st_rdev。需要注意的是，当系统调用失败时，输出参数将显示为传入前的值。

- 对于位集合参数（比如信号集类型 sigset_t），strace 将用“[]”输出该集合中所有被置 1 的位，并用空格将每一项隔开。假设某个程序中有如下代码：

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL);
```

则针对该程序的 strace 命令将输出如下内容：

```
rt_sigprocmask(SIG_BLOCK, [QUIT USR1], NULL, 8) = 0
```

针对其他参数类型的输出方式，请读者参考 strace 的 man 手册，这里不再赘述。对于程序接收到的信号，strace 将输出该信号的值及其描述。比如，我们在一个终端上运行“sleep 100”命令，然后在另一个终端上使用 strace 命令跟踪该进程，接着用“Ctrl+C”终止“sleep 100”进程以观察 strace 的输出。具体操作如下：

```
$ sleep 100
$ ps -ef | grep sleep
shuang 29127 29064 0 03:45 pts/7    00:00:00 sleep 100
```

```
$ strace -p 29127
Process 29127 attached
restart_syscall(<... resuming interrupted call ...>) = ? ERESTART_RESTARTBLOCK
(Interrupted by signal) (此时用“Ctrl+C”中断“sleep 100”进程)
--- SIGINT {si_signo=SIGINT, si_code=SI_KERNEL} ---
+++ killed by SIGINT +++
```

下面考虑一个使用 strace 命令的完整、具体的例子：查看 websrv 服务器在处理客户连接和数据时使用系统调用的情况。具体操作如下：

```
$ ./websrv 127.0.0.1 13579
$ ps -ef | grep websrv
shuang 30526 29064 0 05:19 pts/7 00:00:00 ./websrv 127.0.0.1 13579
$ sudo strace -p 30526
epoll_wait(4,
```

可见，服务器当前正在执行 epoll_wait 系统调用以等待客户请求。值得注意的是，epoll_wait 的第一个参数（标识 epoll 内核事件表的文件描述符）的值是 4，这和前面 lsof 命令的输出一致。接下来使用 17.3 节描述的方式对服务器发起一个连接并发送 HTTP 请求，此时 strace 命令的输出如代码清单 17-2 所示。

代码清单 17-2 strace 命令的输出

```
epoll_wait(4, {{EPOLLIN, {u32=3, u64=4818348437277769731}}}, 10000, -1) = 1
accept(3, {sa_family=AF_INET, sin_port=htons(41408), sin_addr=
           inet_addr("127.0.0.1")}, [16]) = 5
getsockopt(5, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
setsockopt(5, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
epoll_ctl(4, EPOLL_CTL_ADD, 5, {EPOLLIN|EPOLLRDHUP|EPOLLONESHOT|EPOLLET, {u32=5,
           u64=4818361493978349573}}) = 0
fcntl64(5, F_GETFL) = 0x2 (flags O_RDWR)
fcntl64(5, F_SETFL, O_RDWR|O_NONBLOCK) = 0

epoll_wait(4, {{EPOLLIN, {u32=5, u64=4818361493978349573}}}, 10000, -1) = 1
recv(5, "GET http://localhost/a.html HTTP"..., 2048, 0) = 38
recv(5, 0xa601739e, 2010, 0) = -1 EAGAIN (Resource temporarily unavailable)
futex(0x8ace034, FUTEX_WAKE_PRIVATE, 1) = 1

epoll_wait(4, {{EPOLLIN, {u32=5, u64=8589934597}}}, 10000, -1) = 1
recv(5, "Host: localhost\r\n", 2010, 0) = 17
recv(5, 0xa60173af, 1993, 0) = -1 EAGAIN (Resource temporarily unavailable)
futex(0x8ace034, FUTEX_WAKE_PRIVATE, 1) = 1

epoll_wait(4, {{EPOLLIN, {u32=5, u64=8589934597}}}, 10000, -1) = 1
recv(5, "\r\n", 1993, 0) = 2
recv(5, 0xa60173b1, 1991, 0) = -1 EAGAIN (Resource temporarily unavailable)
futex(0x8ace034, FUTEX_WAKE_PRIVATE, 1) = 1

epoll_wait(4, {{EPOLLOUT, {u32=5, u64=5}}}, 10000, -1) = 1
writev(5, [{"HTTP/1.1 404 Not Found\r\nContent-", ..., 114}], 1) = 114
epoll_ctl(4, EPOLL_CTL_MOD, 5, {EPOLLIN|EPOLLRDHUP|EPOLLONESHOT|EPOLLET, {u32=5,
           u64=11961983681754562565}}) = 0
```

```

epoll_ctl(4, EPOLL_CTL_DEL, 5, NULL) = 0
close(5) = 0
epoll_wait(4,

```

上面的输出分为五个部分，我们用空行将每个部分隔开。

第一部分从第一次 `epoll_wait` 系统调用开始。此次 `epoll_wait` 调用检测到了文件描述符 3 上的 EPOLLIN 事件。从代码清单 17-1 中 lsof 的输出来看，文件描述符 3 正是服务器的监听 socket。因此，这个事件表示有新客户连接到来，于是 websrv 服务器对监听 socket 执行了 `accept` 调用，`accept` 返回一个新的连接 socket，其值为 5。接着，服务器清除这个新 socket 上的错误，设置其 SO_REUSEADDR 属性，然后往 epoll 内核事件表中注册该 socket 上的 EPOLLRDHUP 和 EPOLLONESHOT 两个事件，最后设置新 socket 为非阻塞的。

第二部分从第二次 `epoll_wait` 系统调用开始。此次 `epoll_wait` 调用检测到了文件描述符 5 上的 EPOLLIN 事件，这表示客户端的第一行数据到达了，于是服务器执行了两次 `recv` 系统调用来接收数据。第一次 `recv` 调用读取到 38 字节的客户数据，即“GET http://localhost/a.html HTTP/1.1\r\n”。第二次 `recv` 调用则失败了，`errno` 是 EAGAIN，这表示目前没有更多的客户数据可读。此后，服务器调用了 `futex` 函数对互斥锁解锁，以唤醒等待互斥锁的线程。可见，POSIX 线程库中的 `pthread_mutex_unlock` 函数在内部调用了 `futex` 函数。

第三、四部分的内容和第二部分类似，我们不再赘述。

第五部分中，`epoll_wait` 调用检测到了文件描述符 5 上的 EPOLLOUT 事件，这表示主线程正确地处理了客户请求，并准备好了待发送的数据，因此主线程开始执行 `writev` 系统调用往客户端写入 HTTP 应答。最后，服务器从 epoll 内核事件表中移除文件描述符 5 上的所有注册事件，并关闭该文件描述符。

由此可见，`strace` 命令使我们能够清楚地查看每次系统调用发生的时机，以及相关参数的值，这比用 `gdb` 调试更方便。

17.5 netstat

`netstat` 是一个功能很强大的网络信息统计工具。它可以打印本地网卡接口上的全部连接、路由表信息、网卡接口信息等。对本书而言，我们主要利用的是上述功能中的第一个，即显示 TCP 连接及其状态信息。毕竟，要获得路由表信息和网卡接口信息，我们可以使用输出内容更丰富的 `route` 和 `ifconfig` 命令。

`netstat` 命令常用的选项包括：

- -n，使用 IP 地址表示主机，而不是主机名；使用数字表示端口号，而不是服务名称。
- -a，显示结果中也包含监听 socket。
- -t，仅显示 TCP 连接。
- -r，显示路由信息。
- -i，显示网卡接口的数据流量。

- -c，每隔 1 s 输出一次。
- -o，显示 socket 定时器（比如保活定时器）的信息。
- -p，显示 socket 所属的进程的 PID 和名字。

下面我们运行 websrv 服务器，并执行 telnet 命令对它发起一个连接请求：

```
$ ./websrv 127.0.0.1 13579 &
$ telnet 127.0.0.1 13579
```

然后执行命令 netstat -nat|grep 127.0.0.1:13579 查看连接状态，结果如下：

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:13579	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:13579	127.0.0.1:48220	ESTABLISHED
tcp	0	0	127.0.0.1:48220	127.0.0.1:13579	ESTABLISHED

由以上结果可见，netstat 的每行输出都包含如下 6 个字段（默认情况）：

- Proto，协议名。
- Recv-Q，socket 内核接收缓冲区中尚未被应用程序读取的数据量。
- Send-Q，未被对方确认的数据量。
- Local Address，本端的 IP 地址和端口号。
- Foreign Address，对方的 IP 地址和端口号。
- State，socket 的状态。对于无状态协议，比如 UDP 协议，这一字段将显示为空。而对面向连接的协议而言，netstat 支持的 State 包括 ESTABLISHED、SYN_SENT、SYN_RCVD、FIN_WAIT1、FIN_WAIT2、TIME_WAIT、CLOSE、CLOSE_WAIT、LAST_ACK、LISTEN、CLOSING、UNKNOWN。它们的含义和图 3-8 中的同名状态一致^②。

上面的输出中，第 1 行表示本地 socket 地址 127.0.0.1:13579 处于 LISTEN 状态，并等待任何远端 socket（用 0.0.0.0:* 表示）对它发起连接。第 2 行表示服务器和远端地址 127.0.0.1:48220 建立了一个连接。第 3 行只是从客户端的角度重复输出第 2 行信息表示的这个连接，因为我们是在同一台机器上运行服务器程序（websrv）和客户端程序（telnet）的。

在服务器程序开发过程中，我们一定要确保每个连接在任一时刻都处于我们期望的状态。因此我们应该习惯于使用 netstat 命令。

17.6 vmstat

vmstat 是 virtual memory statistics 的缩写，它能实时输出系统的各种资源的使用情况，比如进程信息、内存使用、CPU 使用率以及 I/O 使用情况。

^② SYN_RCVD 和 CLOSE 分别对应图 3-8 中的 SYN_RECV 和 CLOSED，UNKNOWN 表示未知状态。

vmstat 命令常用的选项和参数包括：

- -f，显示系统自启动以来执行的 fork 次数。
- -s，显示内存相关的统计信息以及多种系统活动的数量（比如 CPU 上下文切换次数）。
- -d，显示磁盘相关的统计信息。
- -p，显示指定磁盘分区的统计信息。
- -S，使用指定的单位来显示。参数 k、K、m、M 分别代表 1000、1024、1 000 000 和 1 048 576 字节。
- delay，采样间隔（单位是 s），即每隔 delay 的时间输出一次统计信息。
- count，采样次数，即共输出 count 次统计信息。

默认情况下，vmstat 输出的内容相当丰富。请看下面的示例：

```
$ vmstat 5 3 * 每隔 5 秒输出一次结果，共输出 3 次
procs      -----memory-----  ---swap--  -----io----  --system--  ----cpu----
r  b  swpd   free    buff   cache   si   so    bi   bo   in   cs   us   sy   id   wa
0  0     0    74864  48088  1486188  0    0    12    3   149   280   0    1   99   0
1  0     0    66548  48088  1494640  0    0    0    0   454   619   0    0   99   0
0  0     0    74608  48096  1486188  0    0    0    10  289   339   0    0   99   0
```

注意，第 1 行输出是自系统启动以来的平均结果，而后面的输出则是采样间隔内的平均结果。vmstat 的每条输出都包含 6 个字段，它们的含义分别是：

- procs，进程信息。“r”表示等待运行的进程数目；“b”表示处于不可中断睡眠状态的进程数目。
- memory，内存信息，各项的单位都是千字节（KB）。“swpd”表示虚拟内存的使用数量。“free”表示空闲内存的数量。“buff”表示作为“buffer cache”的内存数量。从磁盘读入的数据可能被保持在“buffer cache”中，以便下一次快速访问。“cache”表示作为“page cache”的内存数量。待写入磁盘的数据将首先被放到“page cache”中，然后由磁盘中断程序写入磁盘。
- swap，交换分区（虚拟内存）的使用信息，各项的单位都是 KB/s。“si”表示数据由磁盘交换至内存的速率；“so”表示数据由内存交换至磁盘的速率。如果这两个值经常发生变化，则说明内存不足。
- io，块设备的使用信息，单位是 block/s。“bi”表示从块设备读入块的速率；“bo”表示向块设备写入块的速率。
- system，系统信息。“in”表示每秒发生的中断次数；“cs”表示每秒发生的上下文切换（进程切换）次数。
- cpu，CPU 使用信息。“us”表示系统所有进程运行在用户空间的时间占 CPU 总运行时间的比例；“sy”表示系统所有进程运行在内核空间的时间占 CPU 总运行时间的比例；“id”表示 CPU 处于空闲状态的时间占 CPU 总运行时间的比例；“wa”表示 CPU 等待 I/O 事件的时间占 CPU 总运行时间的比例。

不过，我们可以使用 iostat 命令获得磁盘使用情况的更多信息，也可以使用 mpstat 获得 CPU 使用情况的更多信息。vmstat 命令主要用于查看系统内存的使用情况。

17.7 ifstat

ifstat 是 interface statistics 的缩写，它是一个简单的网络流量监测工具。其常用的选项和参数包括：

- ❑ -a，监测系统上的所有网卡接口。
- ❑ -i，指定要监测的网卡接口。
- ❑ -t，在每行输出信息前加上时间戳。
- ❑ -b，以 Kbit/s 为单位显示数据，而不是默认的 KB/s。
- ❑ delay，采样间隔（单位是 s），即每隔 delay 的时间输出一次统计信息。
- ❑ count，采样次数，即共输出 count 次统计信息。

举例来说，我们在测试机器 ernest-laptop 上执行如下命令：

```
$ ifstat -a 2 5 # 每隔 2 秒输出一次结果，共输出 5 次
      lo          eth0
KB/s in   KB/s out  KB/s in   KB/s out
  8.62     8.62    124.71   515.74
  7.46     7.46    125.50   510.30
  1.79     1.79    126.87   497.57
  8.10     8.10    127.82   526.13
  9.53     9.53    130.10   516.78
```

从输出来看，ernest-laptop 拥有两个网卡接口：虚拟的回路接口 lo 以及以太网网卡接口 eth0。ifstat 的每条输出都以 KB/s 为单位显示各网卡接口上接收和发送数据的速率。因此，使用 ifstat 命令就可以大概估计各个时段服务器的总输入、输出流量。

17.8 mpstat

mpstat 是 multi-processor statistics 的缩写，它能实时监测多处理器系统上每个 CPU 的使用情况。mpstat 命令和 iostat 命令通常都集成在包 sysstat 中，安装 sysstat 即可获得这两个命令。mpstat 命令的典型用法是（mpstat 命令的选项不多，这里不再专门介绍）：

```
mpstat [-P {ALL}] [interval [count]]
```

选项 P 指定要监控的 CPU 号（0 ~ CPU 个数 -1），其值“ALL”表示监听所有的 CPU。interval 参数是采样间隔（单位是 s），即每隔 interval 的时间输出一次统计信息。count 参数是采样次数，即共输出 count 次统计信息，但 mpstat 最后还会输出这 count 次采样结果的平均值。与 vmstat 命令一样，mpstat 命令输出的第一次结果是自系统启动以来的平均结果，而后面（count-1）次输出结果则是采样间隔内的平均结果。

举例来说，我们在测试机器 Kongming20 上执行如下命令：

```
$ mpstat -P ALL 5 2 # 每隔 5 秒输出一次结果，共输出 2 次
Linux 3.3.0-4.fc16.i686 (Kongming20)        06/25/2012      _i686_ (2 CPU)

CPU %usr %nice %sys %iowait %irq %soft %steal %guest %idle
all 6.60 0.00 16.16 0.00 0.00 7.65 0.00 0.00 69.60
    5.00 0.00 13.20 0.00 0.00 7.20 0.00 0.00 74.60
    8.09 0.00 18.75 0.00 0.00 8.09 0.00 0.00 65.07

CPU %usr %nice %sys %iowait %irq %soft %steal %guest %idle
all 8.05 0.00 19.08 0.00 0.00 8.05 0.00 0.00 64.81
    5.81 0.00 16.83 0.00 0.00 8.42 0.00 0.00 68.94
    10.24 0.00 17.02 0.00 0.00 7.86 0.00 0.00 60.88

Average:
CPU %usr %nice %sys %iowait %irq %soft %steal %guest %idle
all 7.32 0.00 17.62 0.00 0.00 7.85 0.00 0.00 67.17
    5.41 0.00 15.02 0.00 0.00 7.81 0.00 0.00 71.77
    9.17 0.00 19.89 0.00 0.00 7.97 0.00 0.00 62.97
```

为了显示的方便，我们省略了每行输出前导的时间戳。每次采样的输出都包含 3 条信息，每条信息都包含如下几个字段：

- CPU，指示该条信息是哪个 CPU 的数据。“0”表示是第 1 个 CPU 的数据，“1”表示是第 2 个 CPU 的数据，“all”则表示是这两个 CPU 数据的平均值。
- %usr，除了 nice 值为负的进程，系统上其他进程运行在用户空间的时间占 CPU 总运行时间的比例。
- %nice，nice 值为负的进程运行在用户空间的时间占 CPU 总运行时间的比例。
- %sys，系统上所有进程运行在内核空间的时间占 CPU 总运行时间的比例，但不包括硬件和软件中断消耗的 CPU 时间。
- %iowait，CPU 等待磁盘操作的时间占 CPU 总运行时间的比例。
- %irq，CPU 用于处理硬件中断的时间占 CPU 总运行时间的比例。
- %soft，CPU 用于处理软件中断的时间占 CPU 总运行时间的比例。
- %steal，一个物理 CPU 可以包含一对虚拟 CPU，这一对虚拟 CPU 由超级管理程序管理。当超级管理程序在处理某个虚拟 CPU 时，另外一个虚拟 CPU 则必须等待它处理完成才能运行。这部分等待时间就是所谓的 steal 时间。该字段表示 steal 时间占 CPU 总运行时间的比例。
- %guest，运行虚拟 CPU 的时间占 CPU 总运行时间的比例。
- %idle，系统空闲的时间占 CPU 总运行时间的比例。

在所有这些输出字段中，我们最关心的是 %user、%sys 和 %idle。它们基本上反映了我们的代码中业务逻辑代码和系统调用所占的比例，以及系统还能承受多大的负载。很显然，在上面的输出中，执行系统调用占用的 CPU 时间比执行用户业务逻辑占用的 CPU 时间要多。这是因为我们在该机器上运行了 16.4 节介绍的压力测试工具，它在不停地执行 recv/send 系统调用来收发数据。