```
14.     else        // if Parent Process
15.         gval-=2, lval-=2;
16.
17.     if(pid==0)
18.         printf("Child Proc: [%d, %d] \n", gval, lval);
19.     else
20.         printf("Parent Proc: [%d, %d] \n", gval, lval);
21.     return 0;
22. }
```

代码说明

- 第11行：创建子进程。父进程的pid中存有子进程的ID，子进程的pid是0。
- 第12、18行：子进程执行这2行代码，因为pid为0。
- 第15、20行：父进程执行这2行代码，因为此时pid中存有子进程ID。

❖ 运行结果：fork.c

```
root@my_linux:/tcpip# gcc fork.c -o fork
root@my_linux:/tcpip# ./fork
Child Proc: [13, 27]
Parent Proc: [9, 23]
```

从运行结果可以看出，调用fork函数后，父子进程拥有完全独立的内存结构。我认为关于fork函数无需更多示例，希望各位通过该示例充分理解调用fork函数创建进程的方法。

10.2 进程和僵尸进程

文件操作中，关闭文件和打开文件同等重要。同样，进程销毁也和进程创建同等重要。如果未认真对待进程销毁，它们将变成僵尸进程困扰各位。大家可能觉得这是在看玩笑，但事实的确如此。

僵尸（Zombie）进程

大家应该听说过僵尸。恐怖电影中的僵尸可以复活，给主人公造成极大的麻烦。一两只还好对付，但它们一般都成群结队出现，给观众一种刺激和紧张感。但我们的主人公通常神通广大，即使几百只僵尸同时出现也能顺利脱险（僵尸通常行动缓慢），因为主人公知道如何对付僵尸。结局就是所有僵尸都会走向灭亡。

进程的世界同样如此。进程完成工作后（执行完main函数中的程序后）应被销毁，但有时这些进程将变成僵尸进程，占用系统中的重要资源。这种状态下的进程称作“僵尸进程”，这也是给系统带来负担的原因之一。就像电影中描述的那样，我们应该消灭这种进程。当然应掌握正确的方法，否则它会死灰复燃。

产生僵尸进程的原因

为了防止僵尸进程的产生，先解释产生僵尸进程的原因。利用如下两个示例展示调用fork函数产生子进程的终止方式。

- 传递参数并调用exit函数。
- main函数中执行return语句并返回值。

向exit函数传递的参数值和main函数的return语句返回的值都会传递给操作系统。而操作系统不会销毁子进程，直到把这些值传递给产生该子进程的父进程。处在这种状态下的进程就是僵尸进程。也就是说，将子进程变成僵尸进程的正是操作系统。既然如此，此僵尸进程何时被销毁呢？其实已经给出提示。

“应该向创建子进程的父进程传递子进程的exit参数值或return语句的返回值。”

如何向父进程传递这些值呢？操作系统不会主动把这些值传递给父进程。只有父进程主动发起请求（函数调用）时，操作系统才会传递该值。换言之，如果父进程未主动要求获得子进程的结束状态值，操作系统将一直保存，并让子进程长时间处于僵尸进程状态。也就是说，父母要负责收回自己生的孩子（也许这种描述有些不妥）。接下来的示例将创建僵尸进程。

❖ zombie.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int main(int argc, char *argv[])
5.  {
6.      pid_t pid=fork();
7.
8.      if(pid==0)  // if Child Process
9.      {
10.         puts("Hi, I am a child process");
11.     }
12.     else
13.     {
14.         printf("Child Process ID: %d \n", pid);
15.         sleep(30);  // Sleep 30 sec.
16.     }
17.
18.     if(pid==0)
19.         puts("End child process");
20.     else
21.         puts("End parent process");
22.     return 0;
23. }
```

代码说明

- 第14行：输出子进程ID。可以通过该值查看子进程状态（是否为僵尸进程）。
- 第15行：父进程暂停30秒。如果父进程终止，处于僵尸状态的子进程将同时销毁。因此，延缓父进程的执行以验证僵尸进程。

❖ 运行结果：zombie.c

```
root@my_linux:/tcpip# gcc zombie.c -o zombie
root@my_linux:/tcpip# ./zombie
Hi, I am a child process
End child process
Child Process ID: 10977
```

程序开始运行后，将在如上所示状态暂停。跳出这种状态前（30秒内），应验证子进程是否为僵尸进程。该验证在其他控制台窗口进行。

❖ 运行结果：僵尸进程的验证

```
root@my_linux:/tcpip# ps au
USER    PID   %CPU %MEM   VSZ   RSS  TTY     STAT  START   TIME  COMMAND
root    4409  0.0  0.1   1780   524  tty3    Ss+   15:57   0:00  /sbin/getty 384
root    4410  0.0  0.1   1780   524  tty6    Ss+   15:57   0:00  /sbin/getty 384
root    5155  2.6  4.0  30188 20856  tty7    Rs+   15:57   7:08  /usr/X11R6/bin/
. . . . .
root    10976 0.0  0.0   1628   368  pts/0   S+    20:26   0:00  ./zombie
root    10977 0.0  0.0      0     0  pts/0   Z+    20:26   0:00  [zom] <defunct>
. . . . .
```

可以看出，PID为10977的进程状态为僵尸进程（Z+）。另外，经过30秒的等待时间后，PID为10976的父进程和之前的僵尸子进程同时销毁。

提示

后台处理（Background Processing）

后台处理是指将控制台窗口中的指令放到后台运行的方式。如果以如下方式运行上述示例，则程序将在后台运行（&将触发后台处理）。

```
root@my_linux:/tcpip# ./zombie &
```

如果采用这种方式运行程序，即可在同一控制台输入下列命令，无需另外打开新控制台。

```
root@my_linux:/tcpip# ps au
```

希望各位根据介绍以后台处理方式运行之前的示例。

销毁僵尸进程 1：利用 wait 函数

如前所述，为了销毁子进程，父进程应主动请求获取子进程的返回值。接下来讨论发起请求的具体方法（幸好非常简单），共有2种，其中之一就是调用如下函数。

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

pid_t wait(int * statloc);
```

➔ 成功时返回终止的子进程 ID，失败时返回-1。

调用此函数时如果已有子进程终止，那么子进程终止时传递的返回值（exit函数的参数值、main函数的return返回值）将保存到该函数的参数所指内存空间。但函数参数指向的单元中还包含其他信息，因此需要通过下列宏进行分离。

- ❑ WIFEXITED 子进程正常终止时返回“真”（true）。
- ❑ WEXITSTATUS 返回子进程的返回值。

也就是说，向wait函数传递变量status的地址时，调用wait函数后应编写如下代码。

```
if(WIFEXITED(status))  //是正常终止的吗?
{
    puts("Normal termination!");
    printf("Child pass num: %d", WEXITSTATUS(status));  //那么返回值是多少?
}
```

根据上述内容编写示例，此示例中不会再让子进程变成僵尸进程。

❖ wait.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/wait.h>
5.
6.  int main(int argc, char *argv[])
7.  {
8.      int status;
9.      pid_t pid=fork();
10.
11.     if(pid==0)
12.     {
13.         return 3;
14.     }
```

```
15.     else
16.     {
17.         printf("Child PID: %d \n", pid);
18.         pid=fork();
19.         if(pid==0)
20.         {
21.             exit(7);
22.         }
23.         else
24.         {
25.             printf("Child PID: %d \n", pid);
26.             wait(&status);
27.             if(WIFEXITED(status))
28.                 printf("Child send one: %d \n", WEXITSTATUS(status));
29.
30.             wait(&status);
31.             if(WIFEXITED(status))
32.                 printf("Child send two: %d \n", WEXITSTATUS(status));
33.             sleep(30);  // Sleep 30 sec.
34.         }
35.     }
36.     return 0;
37. }
```

代码说明

- 第9、13行：第9行创建的子进程将在第13行通过main函数中的return语句终止。
- 第18、21行：第18行中创建的子进程将在第21行通过调用exit函数终止。
- 第26行：调用wait函数。之前终止的子进程相关信息将保存到status变量，同时相关子进程被完全销毁。
- 第27、28行：第27行中通过WIFEXITED宏验证子进程是否正常终止。如果正常退出，则调用WEXITSTATUS宏输出子进程的返回值。
- 第30~32行：因为之前创建了2个进程，所以再次调用wait函数和宏。
- 第33行：为暂停父进程终止而插入的代码。此时可以查看子进程的状态。

❖ 运行结果：wait.c

```
root@my_linux:/tcpip# gcc wait.c -o wait
root@my_linux:/tcpip# ./wait
Child PID: 12337
Child PID: 12338
Child send one: 3
Child send two: 7
```

系统中并无上述结果中的PID对应的进程，希望各位进行验证。这是因为调用了wait函数，完全销毁了该进程。另外2个子进程终止时返回的3和7传递到了父进程。

这就是通过调用wait函数消灭僵尸进程的方法。调用wait函数时，如果没有已终止的子进程，那么程序将阻塞（Blocking）直到有子进程终止，因此需谨慎调用该函数。

销毁僵尸进程 2：使用 waitpid 函数

wait函数会引起程序阻塞，还可以考虑调用waitpid函数。这是防止僵尸进程的第二种方法，也是防止阻塞的方法。

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

pid_t waitpid(pid_t pid, int * statloc, int options);
```

➔ 成功时返回终止的子进程 ID（或 0），失败时返回-1。

- pid　等待终止的目标子进程的ID，若传递-1，则与wait函数相同，可以等待任意子进程终止。
- statloc　与wait函数的statloc参数具有相同含义。
- options　传递头文件sys/wait.h中声明的常量WNOHANG，即使没有终止的子进程也不会进入阻塞状态，而是返回0并退出函数。

下面介绍调用上述函数的示例。调用waitpid函数时，程序不会阻塞。各位应重点观察这点。

❖ waitpid.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/wait.h>
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int status;
8.      pid_t pid=fork();
9.
10.     if(pid==0)
11.     {
12.         sleep(15);
13.         return 24;
14.     }
15.     else
16.     {
17.         while(!waitpid(-1, &status, WNOHANG))
18.         {
19.             sleep(3);
20.             puts("sleep 3sec.");
21.         }
22.
23.         if(WIFEXITED(status))
24.             printf("Child send %d \n", WEXITSTATUS(status));
25.     }
26.     return 0;
27. }
```

- 第12行：调用sleep函数推迟子进程的执行。这会导致程序延迟15秒。
- 第17行：while循环中调用waitpid函数。向第三个参数传递WNOHANG，因此，若之前没有终止的子进程将返回0。

❖ 运行结果：waitpid.c

```
root@my_linux:/tcpip# gcc waitpid.c -o waitpid
root@my_linux:/tcpip# ./waitpid
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
Child send 24
```

可以看出第20行共执行了5次。另外，这也证明waitpid函数并未阻塞。

10.3 信号处理

我们已经知道了进程创建及销毁方法，但还有一个问题没解决。

“子进程究竟何时终止？调用waitpid函数后要无休止地等待吗？”

父进程往往与子进程一样繁忙，因此不能只调用waitpid函数以等待子进程终止。接下来讨论解决方案。

向操作系统求助

子进程终止的识别主体是操作系统，因此，若操作系统能把如下信息告诉正忙于工作的父进程，将有助于构建高效的程序。

“嘿，父进程！你创建的子进程终止了！”

此时父进程将暂时放下工作，处理子进程终止相关事宜。这是不是既合理又很酷的想法呢？为了实现该想法，我们引入信号处理（Signal Handling）机制。此处的“信号”是在特定事件发生时由操作系统向进程发送的消息。另外，为了响应该消息，执行与消息相关的自定义操作的过程称为“处理”或“信号处理”。关于这两点稍后将再次说明，各位现在不用完全理解这些概念。

关于 JAVA 的题外话：保持开放思维

路途漫漫，我们稍事休息，先讨论一下JAVA。我想讨论的主题如下：

“技术上要保持开放思维。”

JAVA语言具有平台移植性，经历了长时间的发展和变化，其优势在企业级开发环境下尤为明显。理论介绍到此为止，接下来通过讨论JAVA拓宽技术视野。

JAVA在编程语言层面支持进程或线程（稍后将介绍），但C语言及C++语言并不支持。也就是说，ANSI标准并未定义支持进程或线程的函数（JAVA的方法）。但仔细想想，这也是合理的。进程或线程应该由操作系统提供支持。因此，Windows中按照Windows的方式，Linux中按照Linux的方式创建进程或线程。但JAVA为了保持平台移植性，以独立于操作系统的方式提供进程和线程的创建方法。因此，JAVA在语言层面支持进程和线程的创建。

既然如此，JAVA网络编程是否相对简单？就像大家之前学习过的，网络编程中需要一定的操作系统相关知识，因此，有些人会把网络编程当做系统编程的一部分。基于JAVA进行网络编程时，的确会摆脱特定的操作系统，所以有人会误认为JAVA网络编程相对简单。

如果在语言层面支持网络编程所需的所有机制，将延长学习时间。要通过面向对象的方法编写高性能网络程序，需要更多努力和知识。如果有机会，还是希望大家尝试JAVA网络编程，而不仅仅局限于Linux或Windows。JAVA在分布式环境中提供理想的网络编程模型，我也曾沉醉于JAVA技术。

对技术有偏见相当于限制了自己的学习范围，希望各位摒弃偏见。也许有人质疑，学习一门技术尚且困难，如何能同时学习多门技术呢？我并不是要求大家同步学习，而是希望各位以开放的思维多关注身边的技术。

提　示

写于 2003 年的文章

我于 2003 年夏写下上文，当时觉得再过几年可能需要修正其中的说法，但现在依然想用此文激励后辈，因此稍作修改后收录于本书。

信号与 signal 函数

下列进程和操作系统间的对话是帮助大家理解信号处理而编写的，其中包含了所有信号处理相关内容。

- 进程：“嘿，操作系统！如果我之前创建的子进程终止，就帮我调用zombie_handler函数。”
- 操作系统：“好的！如果你的子进程终止，我会帮你调用zombie_handler函数，你先把该函数要执行的语句编好！”

上述对话中进程所讲的相当于“注册信号”过程，即进程发现自己的子进程结束时，请求操作系统调用特定函数。该请求通过如下函数调用完成（因此称此函数为信号注册函数）。

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

➔ 为了在产生信号时调用，返回之前注册的函数指针。

上述函数的返回值类型为函数指针，因此函数声明有些繁琐。若各位不太熟悉返回值类型为函数指针的声明，希望加强学习（不懂函数指针将无法理解后续内容）。现在为了便于讲解，我将上述函数声明整理如下。

- 函数名：signal
- 参数：int signo, void (* func)(int)
- 返回类型：参数为int型，返回void型函数指针。

调用上述函数时，第一个参数为特殊情况信息，第二个参数为特殊情况下将要调用的函数的地址值（指针）。发生第一个参数代表的情况时，调用第二个参数所指的函数。下面给出可以在signal函数中注册的部分特殊情况和对应的常数。

- SIGALRM：已到通过调用alarm函数注册的时间。
- SIGINT：输入CTRL+C。
- SIGCHLD：子进程终止。

接下来编写调用signal函数的语句完成如下请求：

“子进程终止则调用mychild函数。”

此时mychild函数的参数应为int，返回值类型应为void。只有这样才能成为signal函数的第二个参数。另外，常数SIGCHLD定义了子进程终止的情况，应成为signal函数的第一个参数。也就是说，signal函数调用语句如下。

```
signal(SIGCHLD, mychild);
```

接下来编写signal函数的调用语句，分别完成如下2个请求。

“已到通过alarm函数注册的时间，请调用timeout函数。”
“输入CTRL+C时调用keycontrol函数。”

代表这2种情况的常数分别为SIGALRM和SIGINT，因此按如下方式调用signal函数。

```
signal(SIGALRM, timeout);
signal(SIGINT, keycontrol);
```

以上就是信号注册过程。注册好信号后，发生注册信号时（注册的情况发生时），操作系统

将调用该信号对应的函数。下面通过示例验证，先介绍alarm函数。

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

    ➔ 返回 0 或以秒为单位的距 SIGALRM 信号发生所剩时间。
```

如果调用该函数的同时向它传递一个正整型参数，相应时间后（以秒为单位）将产生SIGALRM信号。若向该函数传递0，则之前对SIGALRM信号的预约将取消。如果通过该函数预约信号后未指定该信号对应的处理函数，则（通过调用signal函数）终止进程，不做任何处理。希望引起注意。

接下来给出信号处理相关示例，希望各位通过该示例彻底掌握之前的内容。

❖ signal.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <signal.h>
4.
5.  void timeout(int sig)
6.  {
7.      if(sig==SIGALRM)
8.          puts("Time out!");
9.      alarm(2);
10. }
11. void keycontrol(int sig)
12. {
13.     if(sig==SIGINT)
14.         puts("CTRL+C pressed");
15. }
16.
17. int main(int argc, char *argv[])
18. {
19.     int i;
20.     signal(SIGALRM, timeout);
21.     signal(SIGINT, keycontrol);
22.     alarm(2);
23.
24.     for(i=0; i<3; i++)
25.     {
26.         puts("wait...");
27.         sleep(100);
28.     }
29.     return 0;
30. }
```

- 第5、11行：分别定义信号处理函数。这种类型的函数称为信号处理器（Handler）。
- 第9行：为了每隔2秒重复产生SIGALRM信号，在信号处理器中调用alarm函数。
- 第20、21行：注册SIGALRM、SIGINT信号及相应处理器。
- 第22行：预约2秒后发生SIGALRM信号。
- 第27行：为了查看信号产生和信号处理器的执行并提供每次100秒、共3次的等待时间，在循环中调用sleep函数。也就是说，再过300秒、约5分钟后终止程序，这是相当长的一段时间，但实际执行时只需不到10秒。关于其原因稍后再解释。

❖ 运行结果：signal.c

```
root@my_linux:/tcpip# gcc signal.c -o signal
root@my_linux:/tcpip# ./signal
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

上述是没有任何输入时的运行结果。下面在运行过程中输入CTRL+C。可以看到输出“CTRL+C pressed”字符串。有一点必须说明：

> “发生信号时将唤醒由于调用sleep函数而进入阻塞状态的进程。”

调用函数的主体的确是操作系统，但进程处于睡眠状态时无法调用函数。因此，产生信号时，为了调用信号处理器，将唤醒由于调用sleep函数而进入阻塞状态的进程。而且，进程一旦被唤醒，就不会再进入睡眠状态。即使还未到sleep函数中规定的时间也是如此。所以，上述示例运行不到10秒就会结束，连续输入CTRL+C则有可能1秒都不到。

利用 sigaction 函数进行信号处理

前面所学的内容足以用来编写防止僵尸进程生成的代码。但我还想介绍sigaction函数，它类似于signal函数，而且完全可以代替后者，也更稳定。之所以稳定，是因为如下原因：

> “signal函数在UNIX系列的不同操作系统中可能存在区别，但sigaction函数完全相同。”

实际上现在很少使用signal函数编写程序，它只是为了保持对旧程序的兼容。下面介绍sigaction函数，但只讲解可替换signal函数的功能，因为全面介绍会给各位带来不必要的负担。

```
#include <signal.h>

int sigaction(int signo, const struct sigaction * act, struct sigaction *
oldact);
```

➔ 成功时返回 0，失败时返回-1。

- signo　与signal函数相同，传递信号信息。
- act　对应于第一个参数的信号处理函数（信号处理器）信息。
- oldact　通过此参数获取之前注册的信号处理函数指针，若不需要则传递0。

声明并初始化sigaction结构体变量以调用上述函数，该结构体定义如下。

```
struct sigaction
{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

此结构体的sa_handler成员保存信号处理函数的指针值（地址值）。sa_mask和sa_flags的所有位均初始化为0即可。这2个成员用于指定信号相关的选项和特性，而我们的目的主要是防止产生僵尸进程，故省略。理解这些参数所需参考书将在后面给出。

下面给出示例，其中还包括了尚未讲解的使用sigaction函数所需全部内容。

❖ sigaction.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <signal.h>
4.
5.  void timeout(int sig)
6.  {
7.      if(sig==SIGALRM)
8.          puts("Time out!");
9.      alarm(2);
10. }
11.
12. int main(int argc, char *argv[])
13. {
14.     int i;
15.     struct sigaction act;
16.     act.sa_handler=timeout;
17.     sigemptyset(&act.sa_mask);
18.     act.sa_flags=0;
```

```
19.     sigaction(SIGALRM, &act, 0);
20.
21.     alarm(2);
22.
23.     for(i=0; i<3; i++)
24.     {
25.         puts("wait...");
26.         sleep(100);
27.     }
28.     return 0;
29. }
```

代码说明

- 第15、16行：为了注册信号处理函数，声明sigaction结构体变量并在sa_handler成员中保存函数指针值。
- 第17行：调用sigemptyset函数将sa_mask成员的所有位初始化为0。
- 第18行：sa_flags成员同样初始化为0。
- 第19、21行：注册SIGALRM信号的处理器。调用alarm函数预约2秒后发生SIGALRM信号。

❖ 运行结果：sigaction.c

```
root@my_linux:/tcpip# gcc sigaction.c -o sigaction
root@my_linux:/tcpip# ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

这就是信号处理相关理论，以此为基础讨论消灭僵尸进程的方法。

利用信号处理技术消灭僵尸进程

我相信各位也可以独立编写消灭僵尸进程的示例。子进程终止时将产生SIGCHLD信号，知道这一点就很容易完成。接下来利用sigaction函数编写示例。

❖ remove_zombie.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <signal.h>
5.  #include <sys/wait.h>
6.
7.  void read_childproc(int sig)
8.  {
```

```
9.      int status;
10.     pid_t id=waitpid(-1, &status, WNOHANG);
11.     if(WIFEXITED(status))
12.     {
13.         printf("Removed proc id: %d \n", id);
14.         printf("Child send: %d \n", WEXITSTATUS(status));
15.     }
16. }
17.
18. int main(int argc, char *argv[])
19. {
20.     pid_t pid;
21.     struct sigaction act;
22.     act.sa_handler=read_childproc;
23.     sigemptyset(&act.sa_mask);
24.     act.sa_flags=0;
25.     sigaction(SIGCHLD, &act, 0);
26.
27.     pid=fork();
28.     if(pid==0)  /*子进程执行区域*/
29.     {
30.         puts("Hi! I'm child process");
31.         sleep(10);
32.         return 12;
33.     }
34.     else    /*父进程执行区域*/
35.     {
36.         printf("Child proc id: %d \n", pid);
37.         pid=fork();
38.         if(pid==0)  /*另一子进程执行区域*/
39.         {
40.             puts("Hi! I'm child process");
41.             sleep(10);
42.             exit(24);
43.         }
44.         else
45.         {
46.             int i;
47.             printf("Child proc id: %d \n", pid);
48.             for(i=0; i<5; i++)
49.             {
50.                 puts("wait...");
51.                 sleep(5);
52.             }
53.         }
54.     }
55.     return 0;
56. }
```

代码说明

- 第21~25行：注册SIGCHLD信号对应的处理器。若子进程终止，则调用第7行中定义的函数。处理函数中调用了waitpid函数，所以子进程将正常终止，不会成为僵尸进程。
- 第27、37行：父进程共创建了2个子进程。
- 第48、51行：为了等待发生SIGCHLD信号，使父进程共暂停5次，每次间隔5秒。发生信号时，父进程将被唤醒，因此实际暂停时间不到25秒。

❖ 运行结果：remove_zombie.c

```
root@my_linux:/tcpip# gcc remove_zombie.c -o zombie
root@my_linux:/tcpip# ./zombie
Hi! I'm child process
Child proc id: 9529
Hi! I'm child process
Child proc id: 9530
wait...
wait...
Removed proc id: 9530
Child send: 24
wait...
Removed proc id: 9529
Child send: 12
wait...
wait...
```

可以看出，子进程并未变成僵尸进程，而是正常终止了。接下来利用进程相关知识编写服务器端。

10.4 基于多任务的并发服务器

我们已做好了利用fork函数编写并发服务器的准备，现在可以开始编写像样的服务器端了。

基于进程的并发服务器模型

之前的回声服务器端每次只能向1个客户端提供服务。因此，我们将扩展回声服务器端，使其可以同时向多个客户端提供服务。图10-2给出了基于多进程的并发回声服务器端的实现模型。

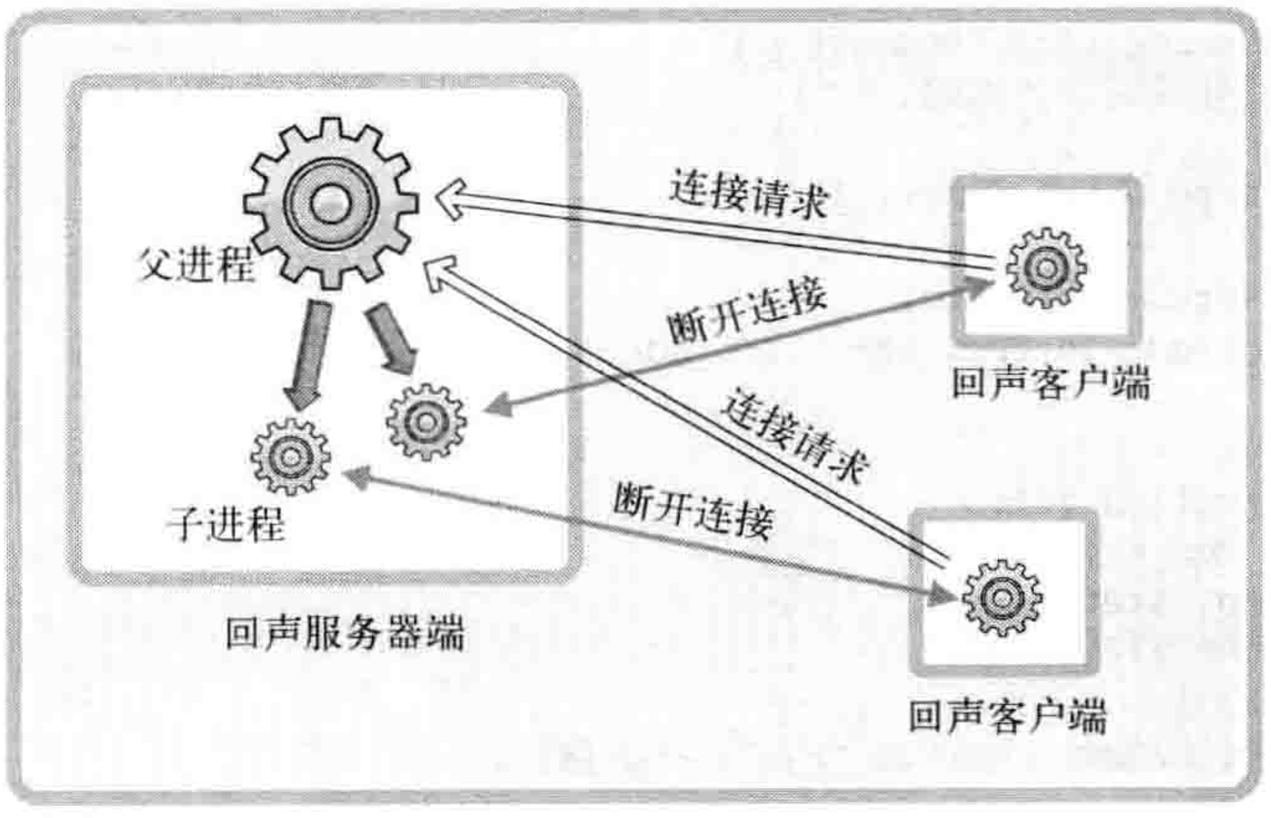

图10-2 并发服务器模型

从图10-2可以看出，每当有客户端请求服务（连接请求）时，回声服务器端都创建子进程以提供服务。请求服务的客户端若有5个，则将创建5个子进程提供服务。为了完成这些任务，需要经过如下过程，这是与之前的回声服务器端的区别所在。

- 第一阶段：回声服务器端（父进程）通过调用accept函数受理连接请求。
- 第二阶段：此时获取的套接字文件描述符创建并传递给子进程。
- 第三阶段：子进程利用传递来的文件描述符提供服务。

此处容易引起困惑的是向子进程传递套接字文件描述符的方法。但各位读完代码后会发现，这其实没什么大不了的，因为子进程会复制父进程拥有的所有资源。实际上根本不用另外经过传递文件描述符的过程。

实现并发服务器

虽然我已经给出了所有理论说明，但大家也许还没想出具体的实现方法，这就有必要理解具体代码。下面给出并发回声服务器端的实现代码。当然，程序是基于多进程实现的，可以结合第4章的回声客户端运行。

❖ echo_mpserv.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <signal.h>
6.  #include <sys/wait.h>
7.  #include <arpa/inet.h>
8.  #include <sys/socket.h>
9.
10. #define BUF_SIZE 30
11. void error_handling(char *message);
12. void read_childproc(int sig);
13.
14. int main(int argc, char *argv[])
15. {
16.     int serv_sock, clnt_sock;
17.     struct sockaddr_in serv_adr, clnt_adr;
18.
19.     pid_t pid;
20.     struct sigaction act;
21.     socklen_t adr_sz;
22.     int str_len, state;
23.     char buf[BUF_SIZE];
24.     if(argc!=2) {
25.         printf("Usage : %s <port>\n", argv[0]);
26.         exit(1);
27.     }
```

```
28.
29.     act.sa_handler=read_childproc;
30.     sigemptyset(&act.sa_mask);
31.     act.sa_flags=0;
32.     state=sigaction(SIGCHLD, &act, 0);
33.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
34.     memset(&serv_adr, 0, sizeof(serv_adr));
35.     serv_adr.sin_family=AF_INET;
36.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
37.     serv_adr.sin_port=htons(atoi(argv[1]));
38.
39.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
40.         error_handling("bind() error");
41.     if(listen(serv_sock, 5)==-1)
42.         error_handling("listen() error");
43.
44.     while(1)
45.     {
46.         adr_sz=sizeof(clnt_adr);
47.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
48.         if(clnt_sock==-1)
49.             continue;
50.         else
51.             puts("new client connected...");
52.         pid=fork();
53.         if(pid==-1)
54.         {
55.             close(clnt_sock);
56.             continue;
57.         }
58.         if(pid==0)  /*子进程运行区域*/
59.         {
60.             close(serv_sock);
61.             while((str_len=read(clnt_sock, buf, BUF_SIZE))!=0)
62.                 write(clnt_sock, buf, str_len);
63.
64.             close(clnt_sock);
65.             puts("client disconnected...");
66.             return 0;
67.         }
68.         else
69.             close(clnt_sock);
70.     }
71.     close(serv_sock);
72.     return 0;
73. }
74.
75. void read_childproc(int sig)
76. {
77.     pid_t pid;
78.     int status;
79.     pid=waitpid(-1, &status, WNOHANG);
80.     printf("removed proc id: %d \n", pid);
81. }
```

```
82. void error_handling(char * message)
83. {
84.     fputs(message, stderr);
85.     fputc('\n', stderr);
86.     exit(1);
87. }
```

代码说明

- 第29~32行：为防止产生僵尸进程而编写的代码。
- 第47、52行：第47行调用accept函数后，在第52行调用fork函数。因此，父子进程分别带有1个第47行生成的套接字（受理客户端连接请求时创建的）文件描述符。
- 第58~66行：子进程运行的区域。此部分向客户端提供回声服务。第60行关闭第33行创建的服务器套接字，这是因为服务器套接字文件描述符同样也传递到子进程。关于这一点稍后将单独讨论。
- 第69行：第47行中通过accept函数创建的套接字文件描述符已复制给子进程，因此服务器端需要销毁自己拥有的文件描述符。关于这一点稍后将单独说明。

❖ 运行结果：echo_mpserv.c

```
root@my_linux:/tcpip# gcc echo_mpserv.c -o mpserv
root@my_linux:/tcpip# ./mpserv 9190
new client connected...
new client connected...
client disconnected...
removed proc id: 7012
client disconnected...
removed proc id: 7018
```

❖ 运行结果：echo_client.c one

```
root@my_linux:/home/swyoon/tcpip# ./client 127.0.0.1 9190
Connected.....
Input message(Q to quit): Hi I'm first client
Message from server: Hi I'm first client
Input message(Q to quit): Oh my friend go away~
Message from server: Oh my friend go away~
Input message(Q to quit): Good bye
Message from server: Good bye
Input message(Q to quit): Q
```

❖ 运行结果：echo_client.c two

```
root@my_linux:/home/swyoon/tcpip# ./client 127.0.0.1 9190
Connected.....
```

```
Input message(Q to quit): Hi I'm second client
Message from server: Hi I'm second client
Input message(Q to quit): Good bye~
Message from server: Good bye~
Input message(Q to quit): Q
```

启动服务器端后，要创建多个客户端并建立连接。可以验证服务器端同时向大多数客户端提供服务，不，一定要验证这一点。

通过 fork 函数复制文件描述符

示例echo_mpserv.c中给出了通过fork函数复制文件描述符的过程。父进程将2个套接字（一个是服务器端套接字，另一个是与客户端连接的套接字）文件描述符复制给子进程。

“只复制文件描述符吗？是否也复制了套接字呢？”

文件描述符的实际复制多少有些难以理解。调用fork函数时复制父进程的所有资源，有些人可能认为也会同时复制套接字。但套接字并非进程所有——从严格意义上说，套接字属于操作系统——只是进程拥有代表相应套接字的文件描述符。也不一定非要这样理解，仅因为如下原因，复制套接字也并不合理。

“复制套接字后，同一端口将对应多个套接字。”

示例echo_mpserv.c中的fork函数调用过程如图10-3所示。调用fork函数后，2个文件描述符指向同一套接字。

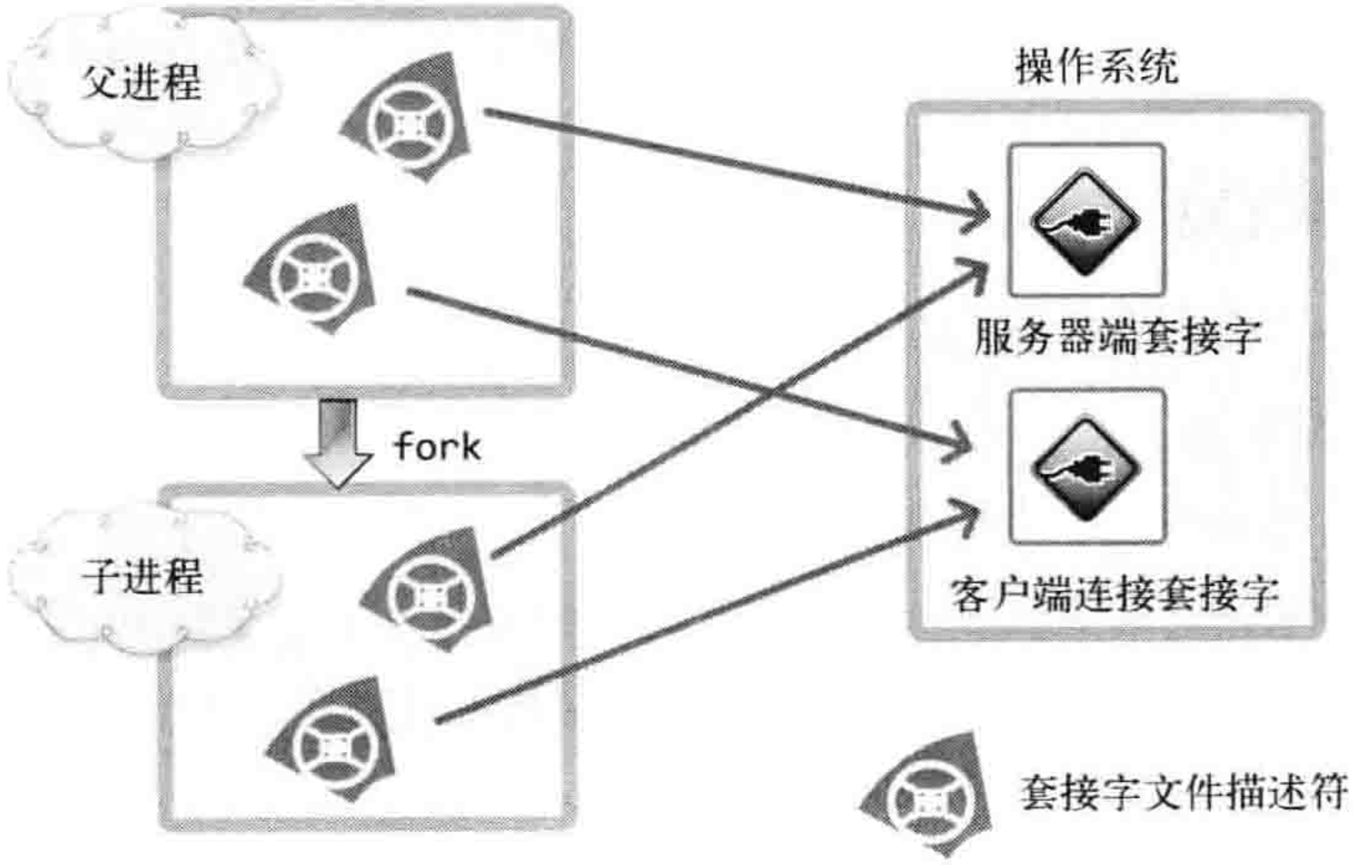

图10-3 调用fork函数并复制文件描述符

如图10-3所示，1个套接字中存在2个文件描述符时，只有2个文件描述符都终止（销毁）后，才能销毁套接字。如果维持图中的连接状态，即使子进程销毁了与客户端连接的套接字文件描述符，也无法完全销毁套接字（服务器端套接字同样如此）。因此，调用fork函数后，要将无关的套接字文件描述符关掉，如图10-4所示。

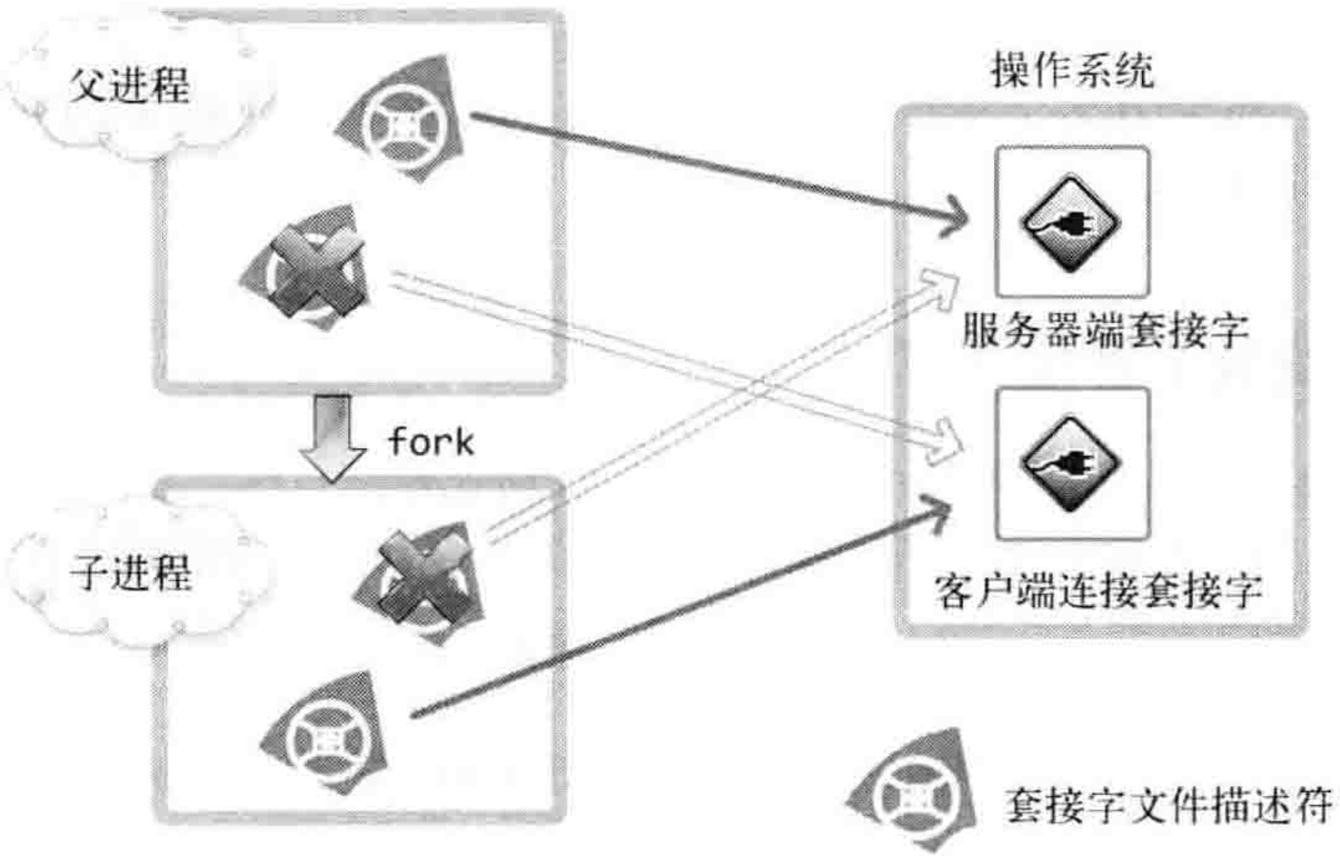

图10-4　整理复制的文件描述符

为了将文件描述符整理成图10-4的形式，示例echo_mpserv.c的第60行和第69行调用了close函数。

10.5　分割 TCP 的 I/O 程序

各位应该已经理解fork函数相关的所有有用内容。下面以此为基础，再讨论客户端中分割I/O程序（Routine）的方法。内容非常简单，大家不必有负担。

分割 I/O 程序的优点

我们已实现的回声客户端的数据回声方式如下：

> “向服务器端传输数据，并等待服务器端回复。无条件等待，直到接收完服务器端的回声数据后，才能传输下一批数据。”

传输数据后需要等待服务器端返回的数据，因为程序代码中重复调用了read和write函数。只能这么写的原因之一是，程序在1个进程中运行。但现在可以创建多个进程，因此可以分割数据收发过程。默认的分割模型如图10-5所示。

从图10-5可以看出，客户端的父进程负责接收数据，额外创建的子进程负责发送数据。分割后，不同进程分别负责输入和输出，这样，无论客户端是否从服务器端接收完数据都可以进行传输。

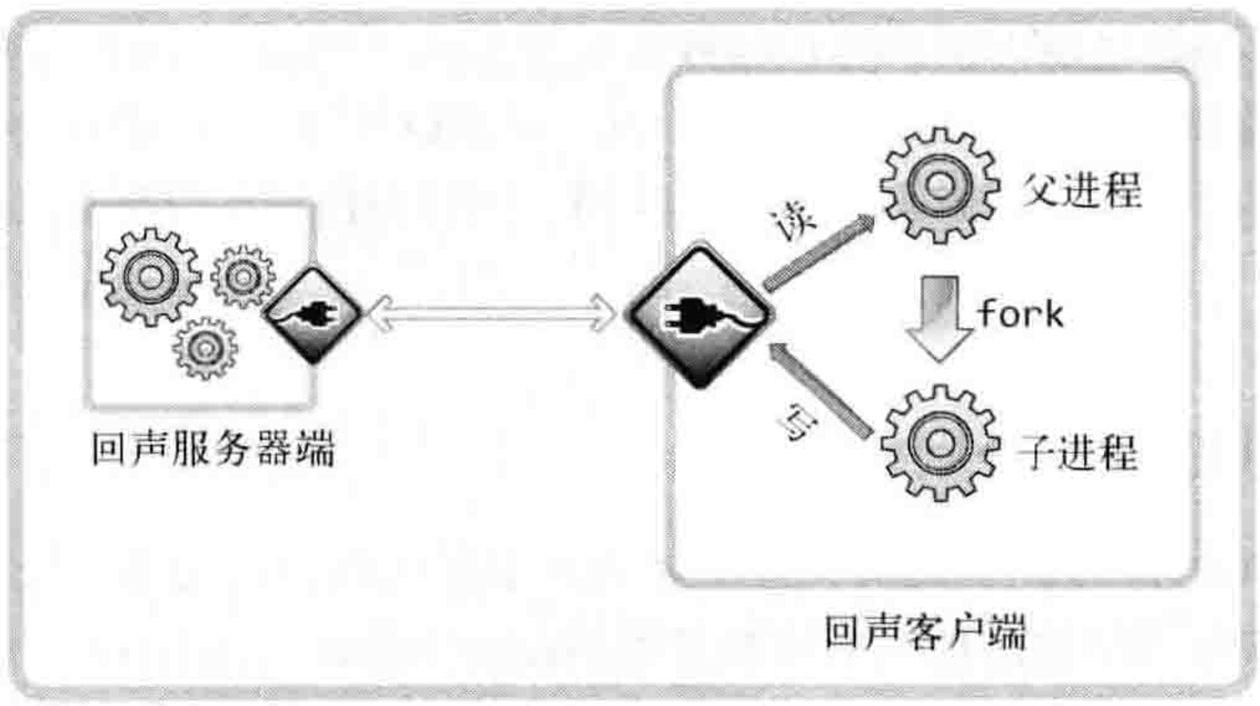

图10-5 回声客户端I/O分割模型

选择这种实现方式的原因有很多，但最重要的一点是，程序的实现更加简单。也许有人质疑：既然多产生1个进程，怎么能简化程序实现呢？其实，按照这种实现方式，父进程中只需编写接收数据的代码，子进程中只需编写发送数据的代码，所以会简化。实际上，在1个进程内同时实现数据收发逻辑需要考虑更多细节。程序越复杂，这种区别越明显，它也是公认的优点。

提 示

回声客户端不用分割 I/O 程序

实际上，回声客户端并无特殊原因进行 I/O 程序的分割，如若进行反而更复杂。本书只是为了讲解分割 I/O 的方法而选取了回声客户端，希望各位不要误解。

分割I/O程序的另一个好处是，可以提高频繁交换数据的程序性能，如图10-6所示。

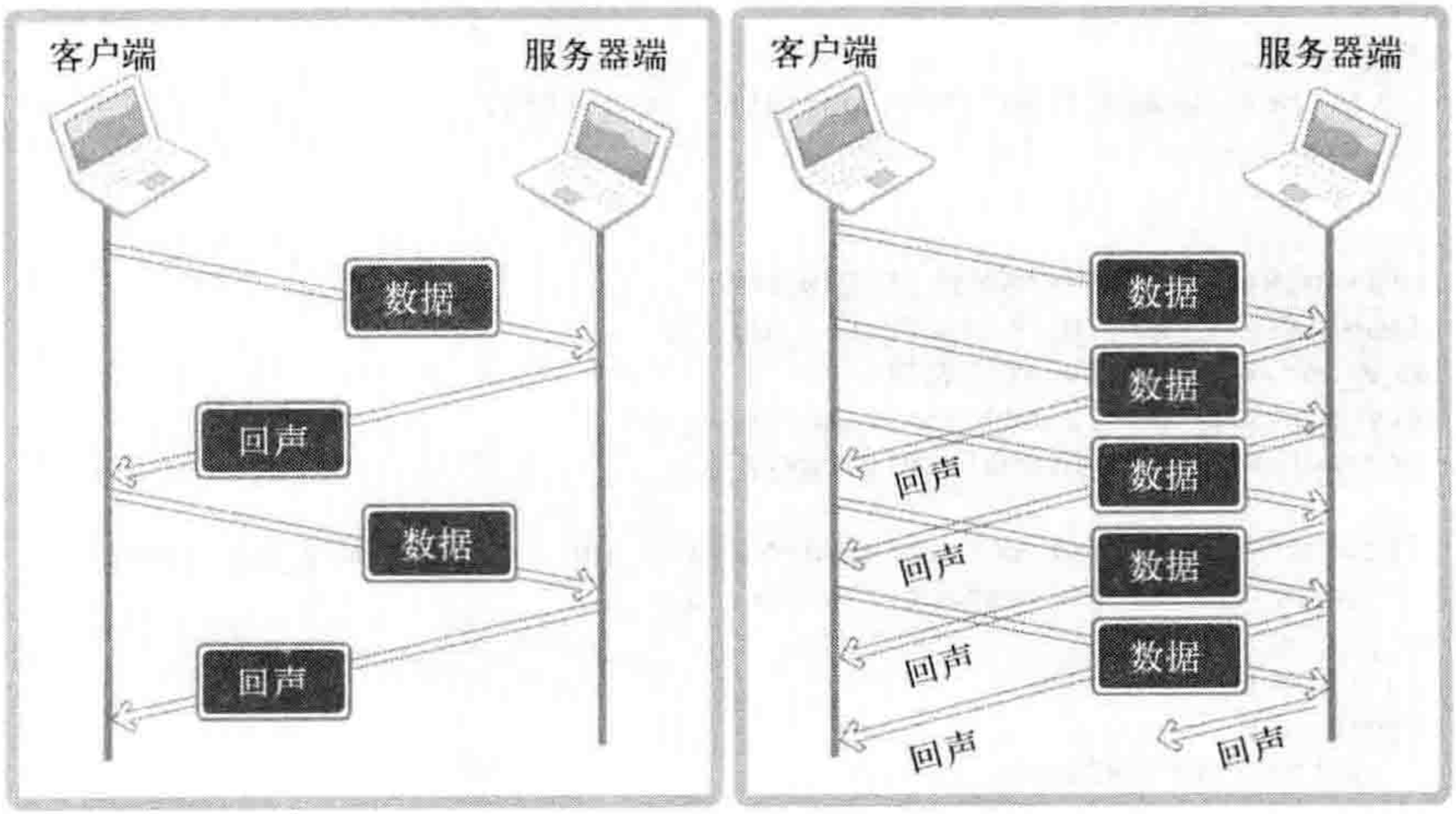

图10-6 数据交换方法比较

图10-6左侧演示的是之前的回声客户端数据交换方式，右侧演示的是分割I/O后的客户端数据传输方式。服务器端相同，不同的是客户端区域。分割I/O后的客户端发送数据时不必考虑接收数据的情况，因此可以连续发送数据，由此提高同一时间内传输的数据量。这种差异在网速较慢时尤为明显。

回声客户端的 I/O 程序分割

我们已经知道I/O程序分割的意义，接下来通过实际代码进行实现，分割的对象是回声客户端。下列回声客户端可以结合之前的回声服务器端echo_mpserv.c运行。

✣ echo_mpclient.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.
8.  #define BUF_SIZE 30
9.  void error_handling(char *message);
10. void read_routine(int sock, char *buf);
11. void write_routine(int sock, char *buf);
12.
13. int main(int argc, char *argv[])
14. {
15.     int sock;
16.     pid_t pid;
17.     char buf[BUF_SIZE];
18.     struct sockaddr_in serv_adr;
19.     if(argc!=3) {
20.         printf("Usage : %s <IP> <port>\n", argv[0]);
21.         exit(1);
22.     }
23.
24.     sock=socket(PF_INET, SOCK_STREAM, 0);
25.     memset(&serv_adr, 0, sizeof(serv_adr));
26.     serv_adr.sin_family=AF_INET;
27.     serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
28.     serv_adr.sin_port=htons(atoi(argv[2]));
29.
30.     if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
31.         error_handling("connect() error!");
32.
33.     pid=fork();
34.     if(pid==0)
35.         write_routine(sock, buf);
36.     else
37.         read_routine(sock, buf);
```

```
38.
39.     close(sock);
40.     return 0;
41. }
42.
43. void read_routine(int sock, char *buf)
44. {
45.     while(1)
46.     {
47.         int str_len=read(sock, buf, BUF_SIZE);
48.         if(str_len==0)
49.             return;
50.
51.         buf[str_len]=0;
52.         printf("Message from server: %s", buf);
53.     }
54. }
55. void write_routine(int sock, char *buf)
56. {
57.     while(1)
58.     {
59.         fgets(buf, BUF_SIZE, stdin);
60.         if(!strcmp(buf,"q\n") || !strcmp(buf,"Q\n"))
61.         {
62.             shutdown(sock, SHUT_WR);
63.             return;
64.         }
65.         write(sock, buf, strlen(buf));
66.     }
67. }
68. void error_handling(char *message)
69. {
70.     fputs(message, stderr);
71.     fputc('\n', stderr);
72.     exit(1);
73. }
```

代码说明

- 第34~37行：第35行调用的write_routine函数中只有数据输出相关代码，第37行调用的read_routine函数中只有数据输入相关代码。像这样分割I/O并分别在不同函数中定义，将有利于代码实现。
- 第62行：调用shutdown函数向服务器端传递EOF。当然，执行第63行的return语句后，可以调用第39行的close函数传递EOF。但现在已通过第33行的fork函数调用复制了文件描述符，此时无法通过1次close函数调用传递EOF，因此需要通过shutdown函数调用另外传递。

运行结果跟普通回声服务器端/客户端相同，故省略。只是上述示例分割了I/O，为了简化输出过程，与之前示例不同，不会输出如下字符串：

```
"Input message(Q to quit): "
```

无论是否接收消息，每次通过键盘输入字符串时都会输出上述字符串，可能造成输出混乱。基于多任务的服务器端实现方法讲解到此结束。

10.6　习题

(1) 下列关于进程的说法错误的是?

a. 从操作系统的角度上说，进程是程序运行的单位。
b. 进程根据创建方式建立父子关系。
c. 进程可以包含其他进程，即一个进程的内存空间可以包含其他进程。
d. 子进程可以创建其他子进程，而创建出来的子进程还可以创建其子进程，但所有这些进程只与一个父进程建立父子关系。

(2) 调用fork函数将创建子进程，以下关于子进程的描述错误的是?

a. 父进程销毁时也会同时销毁子进程。
b. 子进程是复制父进程所有资源创建出的进程。
c. 父子进程共享全局变量。
d. 通过fork函数创建的子进程将执行从开始到fork函数调用为止的代码。

(3) 创建子进程时将复制父进程的所有内容，此时的复制对象也包含套接字文件描述符。编写程序验证复制的文件描述符整数值是否与原文件描述符整数值相同。

(4) 请说明进程变为僵尸进程的过程及预防措施。

(5) 如果在未注册SIGINT信号的情况下输入Ctrl+C，将由操作系统默认的事件处理器终止程序。但如果直接注册Ctrl+C信号的处理器，则程序不会终止，而是调用程序员指定的事件处理器。编写注册处理函数的程序，完成如下功能：

“输入Ctrl+C时，询问是否确定退出程序，输入Y则终止程序。”

另外，编写程序使其每隔1秒输出简单字符串，并适用于上述时间处理器注册代码。

第 11 章

进程间通信 11

第 10 章讲解了如何创建进程，本章将讨论创建的 2 个进程之间交换数据的方法。这与构建服务器端并无直接关系，但可能有助于构建多种类型服务器端，以及更好地理解操作系统。

11.1 进程间通信的基本概念

进程间通信（Inter Process Communication）意味着两个不同进程间可以交换数据，为了完成这一点，操作系统中应提供两个进程可以同时访问的内存空间。

对进程间通信的基本理解

理解好进程间通信并没有想象中那么难，进程A和B之间的如下谈话内容就是一种进程间通信规则。

“如果我有1个面包，变量bread的值就变为1。如果吃掉这个面包，bread的值又变回0。因此，你可以通过变量bread值判断我的状态。”

也就是说，进程A通过变量bread将自己的状态通知给了进程B，进程B通过变量bread听到了进程A的话。因此，只要有两个进程可以同时访问的内存空间，就可以通过此空间交换数据。但正如第10章所讲，进程具有完全独立的内存结构。就连通过fork函数创建的子进程也不会与父进程共享内存空间。因此，进程间通信只能通过其他特殊方法完成。

各位应该已经明白进程间通信的含义及其无法简单实现的原因，下面正式介绍进程间通信方法。

通过管道实现进程间通信

图11-1表示基于管道（PIPE）的进程间通信结构模型。

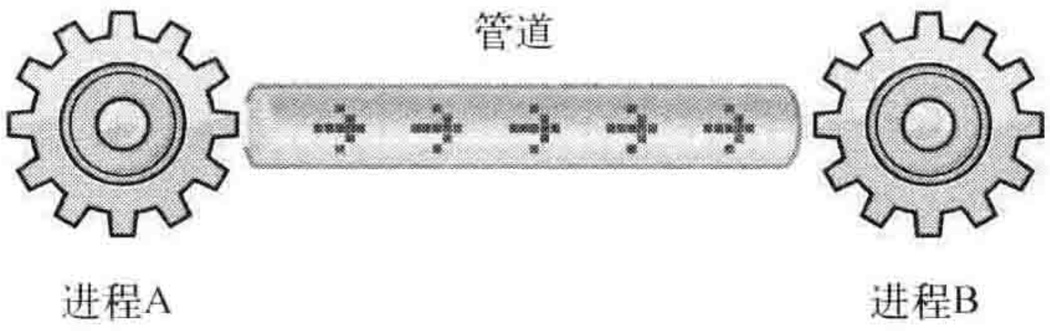

图11-1　基于管道的进程间通信模型

从图11-1中可以看到，为了完成进程间通信，需要创建管道。管道并非属于进程的资源，而是和套接字一样，属于操作系统（也就不是fork函数的复制对象）。所以，两个进程通过操作系统提供的内存空间进行通信。下面介绍创建管道的函数。

```
#include <unistd.h>

int pipe(int filedes[2]);

    ➔ 成功时返回 0，失败时返回-1。
```

- filedes[0]　通过管道接收数据时使用的文件描述符，即管道出口。
- filedes[1]　通过管道传输数据时使用的文件描述符，即管道入口。

以长度为2的int数组地址值作为参数调用上述函数时，数组中存有两个文件描述符，它们将被用作管道的出口和入口。父进程调用该函数时将创建管道，同时获取对应于出入口的文件描述符，此时父进程可以读写同一管道（相信大家也做过这样的实验）。但父进程的目的是与子进程进行数据交换，因此需要将入口或出口中的1个文件描述符传递给子进程。如何完成传递呢？答案就是调用fork函数。通过下列示例进行演示。

❖ pipe1.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #define BUF_SIZE 30
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int fds[2];
8.      char str[]="Who are you?";
9.      char buf[BUF_SIZE];
10.     pid_t pid;
11.
12.     pipe(fds);
13.     pid=fork();
14.     if(pid==0)
15.     {
16.         write(fds[1], str, sizeof(str));
```

```
17.     }
18.     else
19.     {
20.         read(fds[0], buf, BUF_SIZE);
21.         puts(buf);
22.     }
23.     return 0;
24. }
```

代码说明

- 第12行：调用pipe函数创建管道，fds数组中保存用于I/O的文件描述符。
- 第13行：接着调用fork函数。子进程将同时拥有通过第12行函数调用获取的2个文件描述符。注意！复制的并非管道，而是用于管道I/O的文件描述符。至此，父子进程同时拥有I/O文件描述符。
- 第16、20行：子进程通过第16行代码向管道传递字符串。父进程通过第20行代码从管道接收字符串。

❖ 运行结果：pipe1.c

```
root@my_linux:/tcpip# gcc pipe1.c -o pipe1
root@my_linux:/tcpip# ./pipe1
Who are you?
```

上述示例中的通信方法及路径如图11-2所示。重点在于，父子进程都可以访问管道的I/O路径，但子进程仅用输入路径，父进程仅用输出路径。

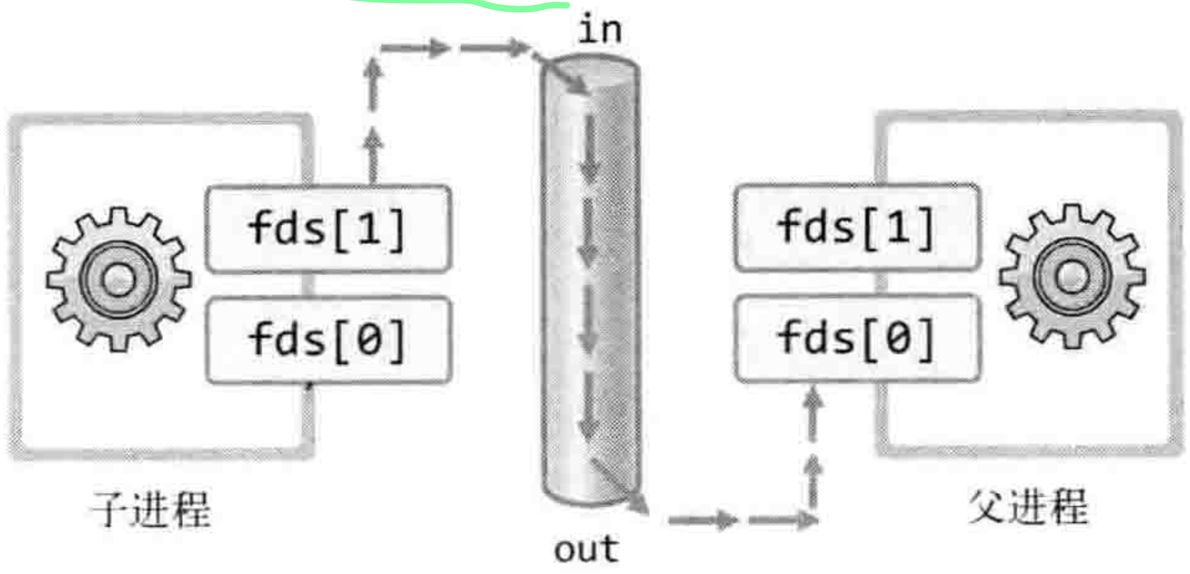

图11-2 示例pipe1.c的通信路径

以上就是管道的基本原理及通信方法。应用管道时还有一部分内容需要注意，通过双向通信示例进一步说明。

通过管道进行进程间双向通信

下面创建2个进程通过1个管道进行双向数据交换的示例，其通信方式如图11-3所示。

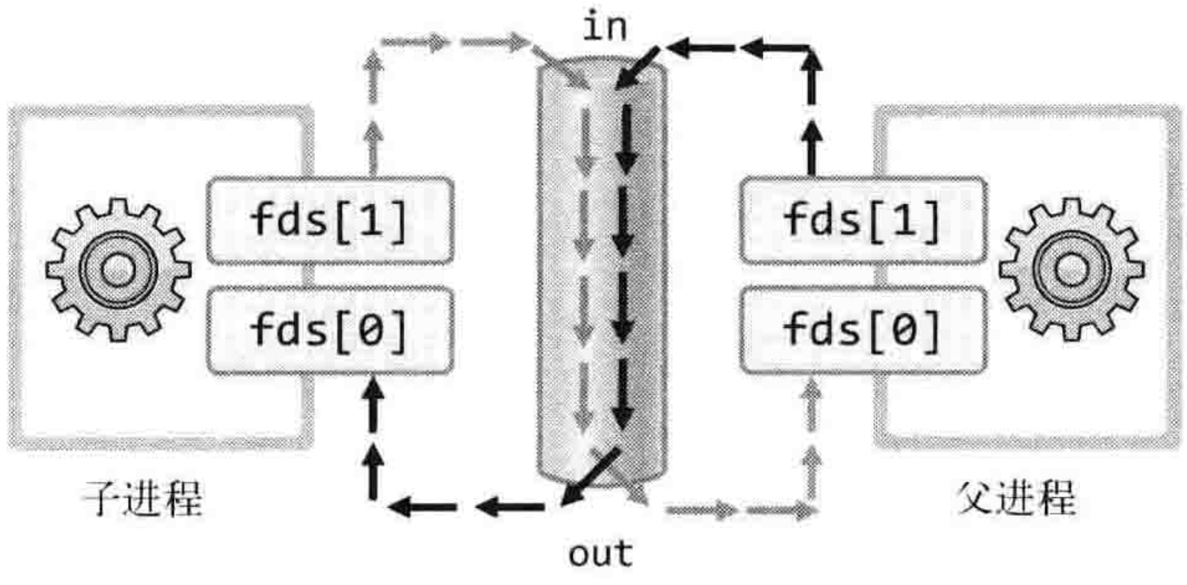

图11-3　双向通信模型1

从图11-3可看出，通过1个管道可以进行双向通信。但采用这种模型时需格外注意。先给出示例，稍后再讨论。

❖ pipe2.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #define BUF_SIZE 30
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int fds[2];
8.      char str1[]="Who are you?";
9.      char str2[]="Thank you for your message";
10.     char buf[BUF_SIZE];
11.     pid_t pid;
12.
13.     pipe(fds);
14.     pid=fork();
15.     if(pid==0)
16.     {
17.         write(fds[1], str1, sizeof(str1));
18.         sleep(2);
19.         read(fds[0], buf, BUF_SIZE);
20.         printf("Child proc output: %s \n", buf);
21.     }
22.     else
23.     {
24.         read(fds[0], buf, BUF_SIZE);
25.         printf("Parent proc output: %s \n", buf);
26.         write(fds[1], str2, sizeof(str2));
27.         sleep(3);
28.     }
29.     return 0;
30. }
```

代码说明

- 第17~20行：子进程运行区域。通过第17行传输数据，通过第19行接收数据。需要特别关注第18行的sleep函数。关于这一点稍后再讨论，希望各位自己思考其含义。
- 第24~26行：父进程的运行区域。通过第24行接收数据，这是为了接收第17行子进程传输的数据。另外，通过第26行传输数据，这些数据被第19行的子进程接收。
- 第27行：父进程先终止时会弹出命令提示符。这时子进程仍在工作，故不会产生问题。这条语句主要是为了防止子进程终止前弹出命令提示符（故可删除）。注释这条代码后再运行程序，各位就会明白我的意思。

❖ 运行结果：pipe2.c

```
root@my_linux:/tcpip# gcc pipe2.c -o pipe2
root@my_linux:/tcpip# ./pipe2
Parent proc output: Who are you?
Child proc output: Thank you for your message
```

运行结果应该和大家的预想一致。这次注释第18行代码后再运行（务必亲自动手操作）。虽然这行代码只将运行时间延迟了2秒，但已引发运行错误。产生原因是什么呢？

“向管道传递数据时，先读的进程会把数据取走。”

简言之，数据进入管道后成为无主数据。也就是通过read函数先读取数据的进程将得到数据，即使该进程将数据传到了管道。因此，注释第18行将产生问题。在第19行，子进程将读回自己在第17行向管道发送的数据。结果，父进程调用read函数后将无限期等待数据进入管道。

从上述示例中可以看到，只用1个管道进行双向通信并非易事。为了实现这一点，程序需要预测并控制运行流程，这在每种系统中都不同，可以视为不可能完成的任务。既然如此，该如何进行双向通信呢？

“创建2个管道。”

非常简单，1个管道无法完成双向通信任务，因此需要创建2个管道，各自负责不同的数据流动即可。其过程如图11-4所示。

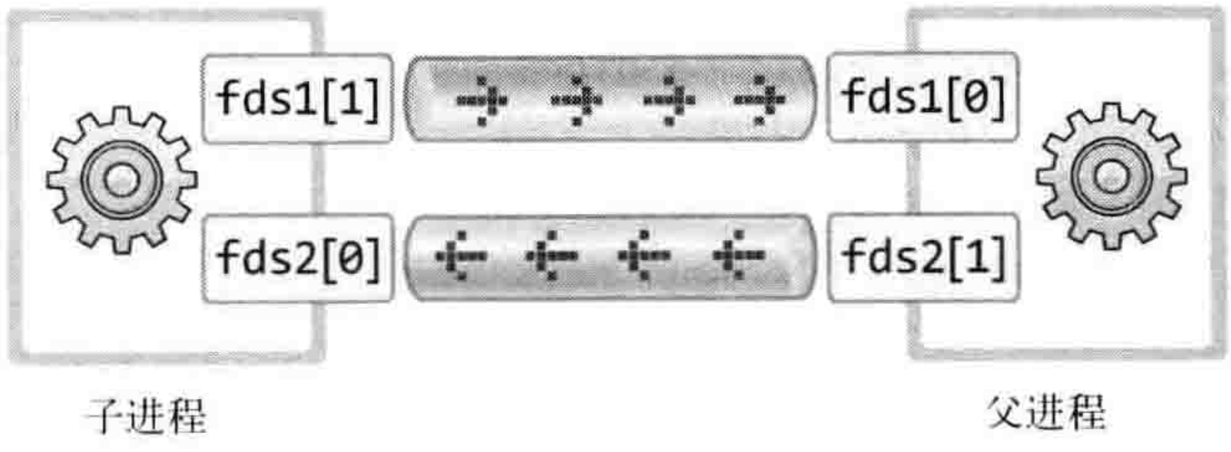

图11-4 双向通信模型2

由图11-4可知，使用2个管道可以避免程序流程的预测或控制。下面采用上述模型改进pipe2.c。

❖ pipe3.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #define BUF_SIZE 30
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int fds1[2], fds2[2];
8.      char str1[]="Who are you?";
9.      char str2[]="Thank you for your message";
10.     char buf[BUF_SIZE];
11.     pid_t pid;
12.
13.     pipe(fds1), pipe(fds2);
14.     pid=fork();
15.     if(pid==0)
16.     {
17.         write(fds1[1], str1, sizeof(str1));
18.         read(fds2[0], buf, BUF_SIZE);
19.         printf("Child proc output: %s \n", buf);
20.     }
21.     else
22.     {
23.         read(fds1[0], buf, BUF_SIZE);
24.         printf("Parent proc output: %s \n", buf);
25.         write(fds2[1], str2, sizeof(str2));
26.         sleep(3);
27.     }
28.     return 0;
29. }
```

代码说明

- 第13行：创建两个管道。
- 第17、23行：子进程可以通过数组fds1指向的管道向父进程传输数据。
- 第18、25行：父进程可以通过数组fds2指向的管道向子进程发送数据。
- 第26行：没有太大的意义，只是为了延迟父进程终止而插入的代码。

❖ 运行结果：pipe3.c

```
root@my_linux:/tcpip# gcc pipe3.c -o pipe3
root@my_linux:/tcpip# ./pipe3
Parent proc output: Who are you?
Child proc output: Thank you for your message
```

11.2　运用进程间通信

上一节学习了基于管道的进程间通信方法，接下来将其运用到网络代码中。如前所述，进程

间通信与创建服务器端并没有直接关联，但其有助于理解操作系统。

保存消息的回声服务器端

下面扩展第10章的echo_mpserv.c，添加如下功能：

“将回声客户端传输的字符串按序保存到文件中。”

我希望将该任务委托给另外的进程。换言之，另行创建进程，从向客户端提供服务的进程读取字符串信息。当然，该过程中需要创建用于接收数据的管道。

下面给出示例。该示例可以与任意回声客户端配合运行，但我们将用第10章介绍过的echo_mpclient.c。

❖ echo_storeserv.c

```
1.  #include <"头声明与第10章的示例echo_mpserv.c一致。">
2.  #define BUF_SIZE 100
3.  void error_handling(char *message);
4.  void read_childproc(int sig);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      int serv_sock, clnt_sock;
9.      struct sockaddr_in serv_adr, clnt_adr;
10.     int fds[2];
11.
12.     pid_t pid;
13.     struct sigaction act;
14.     socklen_t adr_sz;
15.     int str_len, state;
16.     char buf[BUF_SIZE];
17.     if(argc!=2) {
18.         printf("Usage : %s <port>\n", argv[0]);
19.         exit(1);
20.     }
21.
22.     act.sa_handler=read_childproc;
23.     sigemptyset(&act.sa_mask);
24.     act.sa_flags=0;
25.     state=sigaction(SIGCHLD, &act, 0);
26.
27.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
28.     memset(&serv_adr, 0, sizeof(serv_adr));
29.     serv_adr.sin_family=AF_INET;
30.     serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
31.     serv_adr.sin_port=htons(atoi(argv[1]));
32.
33.     if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
```

```
34.         error_handling("bind() error");
35.     if(listen(serv_sock, 5)==-1)
36.         error_handling("listen() error");
37.
38.     pipe(fds);
39.     pid=fork();
40.     if(pid==0)
41.     {
42.         FILE * fp=fopen("echomsg.txt", "wt");
43.         char msgbuf[BUF_SIZE];
44.         int i, len;
45.
46.         for(i=0; i<10; i++)
47.         {
48.             len=read(fds[0], msgbuf, BUF_SIZE);
49.             fwrite((void*)msgbuf, 1, len, fp);
50.         }
51.         fclose(fp);
52.         return 0;
53.     }
54.
55.     while(1)
56.     {
57.         adr_sz=sizeof(clnt_adr);
58.         clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
59.         if(clnt_sock==-1)
60.             continue;
61.         else
62.             puts("new client connected...");
63.
64.         pid=fork();
65.         if(pid==0)
66.         {
67.             close(serv_sock);
68.             while((str_len=read(clnt_sock, buf, BUF_SIZE))!=0)
69.             {
70.                 write(clnt_sock, buf, str_len);
71.                 write(fds[1], buf, str_len);
72.             }
73.
74.             close(clnt_sock);
75.             puts("client disconnected...");
76.             return 0;
77.         }
78.         else
79.             close(clnt_sock);
80.     }
81.     close(serv_sock);
82.     return 0;
83. }
84.
85. void read_childproc(int sig)
86. {
87.     //与示例echo_mpserv.c一致，故省略。
```

```
88. }
89. void error_handling(char *message)
90. {
91.     //与示例echo_mpserv.c一致，故省略。
92. }
```

代码说明

- 第38、39行：第38行创建管道，第39行创建负责保存文件的进程。
- 第40~53行：第39行创建的子进程运行区域。该区域从管道出口fds[0]读取数据并保存到文件中。另外，上述服务器端并不终止运行，而是不断向客户端提供服务。因此，数据在文件中累计到一定程度即关闭文件，该过程通过第46行的循环完成。
- 第71行：第64行通过fork函数创建的所有子进程将复制第38行创建的管道的文件描述符。因此，可以通过管道入口fds[1]传递字符串信息。

❖ 运行结果：echo_storeserv.c

```
root@my_linux:/tcpip# gcc echo_storeserv.c -o serv
root@my_linux:/tcpip# ./serv 9190
new client connected...
new client connected...
removed proc id: 7177
client disconnected...
removed proc id: 7185
client disconnected...
removed proc id: 7191
```

❖ 运行结果：echo_mpclient.c one

```
root@my_linux:/tcpip# gcc echo_mpclient.c -o client
root@my_linux:/tcpip# ./client 127.0.0.1 9190
One
Message from server: One
Three
Message from server: Three
Five
Message from server: Five
Seven
Message from server: Seven
Nine
Message from server: Nine
Q
root@com:/home/swyoon/tcpip#
```

❖ 运行结果：echo_mpclient.c two

```
root@my_linux:/tcpip# ./client 127.0.0.1 9190
Two
Message from server: Two
Four
Message from server: Four
Six
Message from server: Six
Eight
Message from server: Eight
Ten
Message from server: Ten
Q
root@my_linux:/tcpip#
```

如上运行结果所示，希望各位也启动多个客户端向服务器端传输字符串。文件中累计一定数量的字符串后（共10次的fwrite函数调用完成后），可以打开echomsg.txt验证保存的字符串。

提　示

各位应该想到了重构

观察示例 echo_storeserv.c 后，各位可能认为应该针对一部分功能以函数为单位重构代码。我只是在扩展回声服务器端的过程中为了便于学习（有利于跟之前的代码进行比较）而未改变代码框架，因此大家才会看到向下展开的代码像小说一样。

如果想构建更大型的程序

前面已经讲解了并发服务器的第一种实现模型，但各位或许有如下想法：

“我想利用进程和管道编写聊天室程序，使多个客户端进行对话，应该从哪着手呢？”

若想仅用进程和管道构建具有复杂功能的服务器端，程序员需要具备熟练的编程技能和经验。因此，初学者应用该模型扩展程序并非易事，希望大家不要过于拘泥。以后要说明的另外两种模型在功能上更加强大，同时更容易实现我们的想法。

“那前面讲的内容算什么啊？是不是没用呢？”

我曾被多次问及此类问题。各位通过第10章和本章内容理解了操作系统的基本内容，同时也

是学习线程必备的前期知识——进程。而且掌握了多进程代码的基本分析能力。即使各位不会亲自利用多进程构建服务器端，但这些都值得学习。可以将我的个人经验告诉所有读者朋友：

“即使开始时只想学习必要部分，最后也会需要掌握所有内容。”

11.3 习题

(1) 什么是进程间通信？分别从概念上和内存的角度进行说明。

(2) 进程间通信需要特殊的IPC机制，这是由操作系统提供的。进程间通信时为何需要操作系统的帮助？

(3) “管道”是典型的IPC技法。关于管道，请回答如下问题。

a. 管道是进程间交换数据的路径。如何创建此路径？由谁创建？

b. 为了完成进程间通信，2个进程需同时连接管道。那2个进程如何连接到同一管道？

c. 管道允许进行2个进程间的双向通信。双向通信中需要注意哪些内容？

(4) 编写示例复习IPC技法，使2个进程相互交换3次字符串。当然，这2个进程应具有父子关系，各位可指定任意字符串。

done!

第 12 章

I/O复用

12

本章将讨论并发服务器的第二种实现方法——基于 I/O 复用（Multiplexing）的服务器端构建。虽然通过本章多学习一种服务器端实现方法非常重要，但更重要的是理解每种技术的优缺点。如果能掌握每种技术的优劣，就可以根据特定目标灵活应用不同模型，而不是仅关注功能实现。

12.1 基于 I/O 复用的服务器端

接下来讨论并发服务器实现方法的延伸。如果有读者已经跳过第10章和第11章，那就只需把本章内容当做并发服务器实现的第一种方法即可。将要讨论的内容中包含一部分与多进程服务器端的比较，跳过第10章和第11章的读者简单浏览即可。

多进程服务器端的缺点和解决方法

为了构建并发服务器，只要有客户端连接请求就会创建新进程。这的确是实际操作中采用的一种方案，但并非十全十美，因为创建进程时需要付出极大代价。这需要大量的运算和内存空间，由于每个进程都具有独立的内存空间，所以相互间的数据交换也要求采用相对复杂的方法（IPC属于相对复杂的通信方法）。各位应该也感到需要IPC时会提高编程难度。

“那有何解决方案？能否在不创建进程的同时向多个客户端提供服务？”

当然能！本节讲解的I/O复用就是这种技术。大家听到有这种方法是否感到一阵兴奋？但请不要过于依赖该模型！该方案并不适用于所有情况，应当根据目标服务器端的特点采用不同实现方法。下面先理解“复用”（Multiplexing）的意义。

理解复用

“复用”在电子及通信工程领域很常见，向这些领域的专家询问其概念时，他们会亲切地进行如下说明：

“在1个通信频道中传递多个数据（信号）的技术。”

能理解吗？不能的话就再看看“复用”的含义。

“为了提高物理设备的效率，用最少的物理要素传递最多数据时使用的技术。”

上述两种说法内容完全一致，只是描述方式有所区别。下面我根据自己的理解进行解释。图12-1中给出的是纸杯电话，相信大家上小学时也做过。

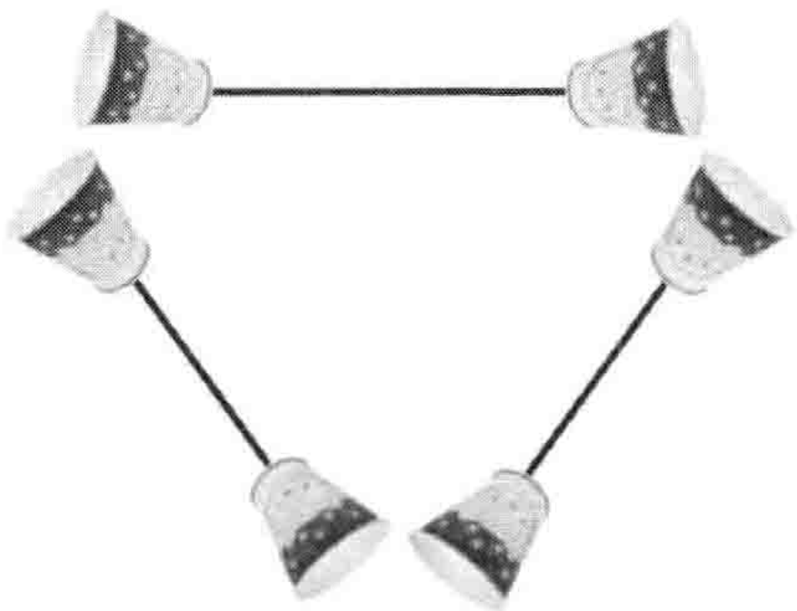

图12-1 3方对话纸杯电话系统

图12-1是远距离的3人可以同时通话的3方对话纸杯电话系统。为使3人同时对话，需准备图中所示系统。另外，为了完成3人对话，说话时需同时对着两个纸杯，接听时也需要耳朵同时对准两个纸杯。此时引入复用技术会使通话更加方便，如图12-2所示。

图12-2 纸杯电话系统中引入复用技术

我们上小学时做过类似系统（把线捆在中间并绷直）。构建这种系统就无需同时使用两个杯

子，可以说小学就学过“复用”的概念了。接下来讨论复用技术的优点。

- 减少连线长度。
- 减少纸杯个数。

即使减少了连线和纸杯的量仍能进行3人通话，当然也有人在考虑如下这种情况：

“好像不能同时说话？”

实际上，因为是在进行对话，所以很少发生同时说话的情况。也就是说，上述系统采用的是“时（time）分复用技术”。而且，因为说话人声高（频率）不同，即使同时说话也能进行一定程度的区分（当然杂音也随之增多）。因此，也可以说系统同时采用了“频（frequency）分复用技术”。这样大家就能理解之前讲的“复用”的定义了。

复用技术在服务器端的应用

纸杯电话系统引入复用技术后，可以减少纸杯数和连线长度。同样，服务器端引入复用技术可以减少所需进程数。为便于比较，先给出第10章的多进程服务器端模型，如图12-3所示。

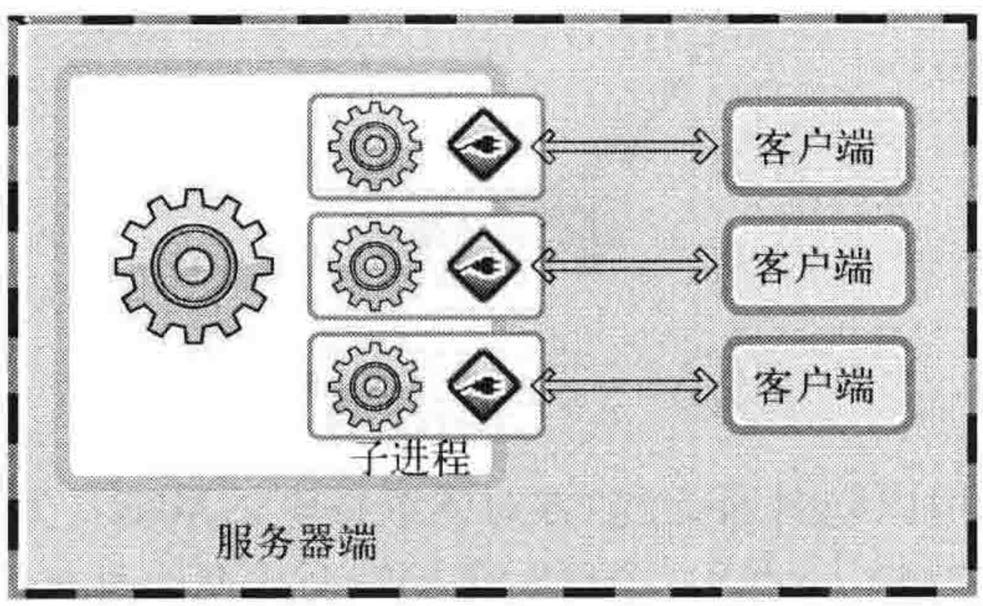

图12-3　多进程服务器端模型

图12-3的模型中引入复用技术，可以减少进程数。重要的是，无论连接多少客户端，提供服务的进程只有1个。

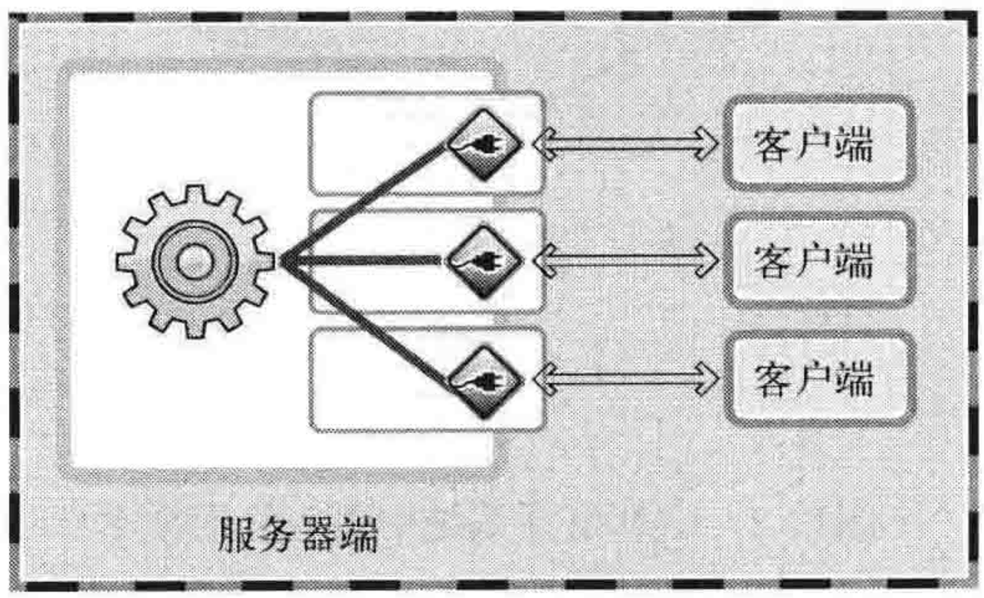

图12-4　I/O复用服务器端模型

以上就是I/O复用服务器端模型的讲解，下面考虑通过1个进程向多个客户端提供服务的方法。

知识补给站 关于I/O复用服务器端的另一种理解

某教室中有10名学生和1位教师，这些孩子并非等闲之辈，上课时不停地提问。学校没办法，只能给每个学生都配1位教师，也就是说教室中现有10位教师。此后，只要有新的转校生，就会增加1位教师，因为转校生也喜欢提问。这个故事中，如果把学生当作客户端，把教师当作与客户端进行数据交换的服务器端进程，则该教室的运营方式为多进程服务器端方式。

有一天，该校来了位具有超能力的教师。这位教师可以应对所有学生的提问，而且回答速度很快，不会让学生等待。因此，学校为了提高教师效率，将其他老师转移到了别的班。现在，学生提问前必须举手，教师确认举手学生的提问后再回答问题。也就是说，现在的教室以I/O复用方式运行。

虽然例子有些奇怪，但可以通过它理解I/O复用技法：教师必须确认有无举手学生，同样，I/O复用服务器端的进程需要确认举手（收到数据）的套接字，并通过举手的套接字接收数据。

12.2 理解 select 函数并实现服务器端

运用select函数是最具代表性的实现复用服务器端方法。Windows平台下也有同名函数提供相同功能，因此具有良好的移植性。

select 函数的功能和调用顺序

使用select函数时可以将多个文件描述符集中到一起统一监视，项目如下。

- 是否存在套接字接收数据？
- 无需阻塞传输数据的套接字有哪些？
- 哪些套接字发生了异常？

提示

监视项称为“事件”（event）

上述监视项称为“事件”。发生监视项对应情况时，称“发生了事件”。这是最常见的表达，希望各位熟悉。另外，本章不会使用术语“事件”，而与本章密切相关的第 17 章将使用该术语，希望大家理解“事件”的含义，以及“发生事件”的意义。

select函数的使用方法与一般函数区别较大，更准确地说，它很难使用。但为了实现I/O复用服务器端，我们应掌握select函数，并运用到套接字编程中。认为“select函数是I/O复用的全部内容”也并不为过。接下来介绍select函数的调用方法和顺序，如图12-5所示。

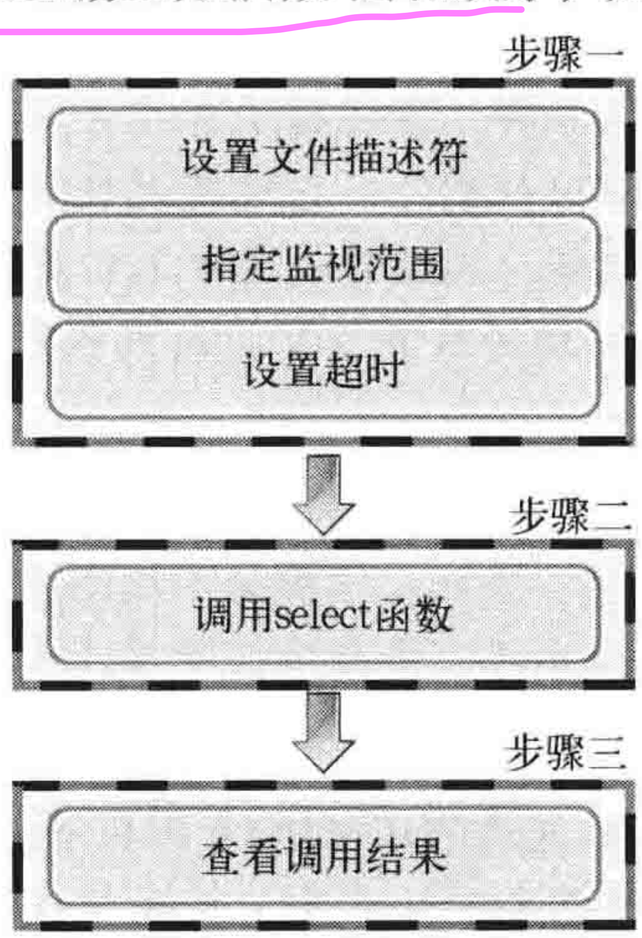

图12-5　select函数调用过程

图12-5给出了从调用select函数到获取结果所经过程。可以看到，调用select函数前需要一些准备工作，调用后还需查看结果。接下来按照上述顺序逐一讲解。

设置文件描述符

利用select函数可以同时监视多个文件描述符。当然，监视文件描述符可以视为监视套接字。此时首先需要将要监视的文件描述符集中到一起。集中时也要按照监视项（接收、传输、异常）进行区分，即按照上述3种监视项分成3类。

使用fd_set数组变量执行此项操作，如图12-6所示。该数组是存有0和1的位数组。

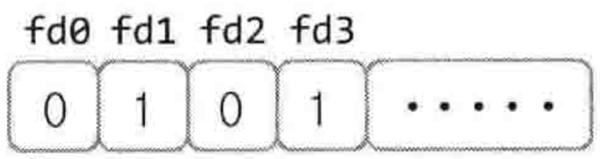

图12-6　fd_set结构体

图12-6中最左端的位表示文件描述符0（所在位置）。如果该位设置为1，则表示该文件描述符是监视对象。那么图中哪些文件描述符是监视对象呢？很明显，是文件描述符1和3。

“是否应当通过文件描述符的数字直接将值注册到fd_set变量？”

当然不是！针对fd_set变量的操作是以位为单位进行的，这也意味着直接操作该变量会比较繁琐。难道要求各位自己完成吗？实际上，在fd_set变量中注册或更改值的操作都由下列宏完成。

- FD_ZERO(fd_set * fdset)：将fd_set变量的所有位初始化为0。
- FD_SET(int fd, fd_set * fdset)：在参数fdset指向的变量中注册文件描述符fd的信息。
- FD_CLR(int fd, fd_set * fdset)：从参数fdset指向的变量中清除文件描述符fd的信息。
- FD_ISSET(int fd, fd_set * fdset)：若参数fdset指向的变量中包含文件描述符fd的信息，则返回“真”。

上述函数中，FD_ISSET用于验证select函数的调用结果。通过图12-7解释这些函数的功能，简洁易懂，无需赘述。

```
int main(void)
{
   fd_set  set;

   FD_ZERO(&set);
                          fd0 fd1 fd2 fd3
                          0   0   0   0   · · · · ·

   FD_SET(1, &set);
                          fd0 fd1 fd2 fd3
                          0   1   0   0   · · · · ·

   FD_SET(2, &set);
                          fd0 fd1 fd2 fd3
                          0   1   1   0   · · · · ·

   FD_CLR(2, &set);
                          fd0 fd1 fd2 fd3
                          0   1   0   0   · · · · ·
}
```

图12-7　fd_set相关函数的功能

设置检查（监视）范围及超时

下面讲解图12-5中步骤一的剩余内容，在此之前先简单介绍select函数。

```
#include <sys/select.h>
#include <sys/time.h>

int select(
int maxfd, fd_set * readset, fd_set * writeset, fd_set * exceptset, const struct
    timeval * timeout);

    ➔ 成功时返回大于 0 的值，失败时返回-1。
```

maxfd	监视对象文件描述符数量。
readset	将所有关注“是否存在待读取数据”的文件描述符注册到fd_set型变量，并传递其地址值。
writeset	将所有关注“是否可传输无阻塞数据”的文件描述符注册到fd_set型变量，并传递其地址值。
exceptset	将所有关注“是否发生异常”的文件描述符注册到fd_set型变量，并传递其地址值。
timeout	调用select函数后，为防止陷入无限阻塞的状态，传递超时（time-out）信息。
返回值	发生错误时返回-1，超时返回时返回0。因发生关注的事件返回时，返回大于0的值，该值是发生事件的文件描述符数。

如上所述，select函数用来验证3种监视项的变化情况。根据监视项声明3个fd_set型变量，分别向其注册文件描述符信息，并把变量的地址值传递到上述函数的第二到第四个参数。但在此之前（调用select函数前）需要决定下面2件事。

“文件描述符的监视（检查）范围是？”

“如何设定select函数的超时时间？”

第一，文件描述符的监视范围与select函数的第一个参数有关。实际上，select函数要求通过第一个参数传递监视对象文件描述符的数量。因此，需要得到注册在fd_set变量中的文件描述符数。但每次新建文件描述符时，其值都会增1，故只需将最大的文件描述符值加1再传递到select函数即可。加1是因为文件描述符的值从0开始。

第二，select函数的超时时间与select函数的最后一个参数有关，其中timeval结构体定义如下。

```
struct timeval
{
    long tv_sec;     //seconds
    long tv_usec;    //microseconds
}
```

本来select函数只有在监视的文件描述符发生变化时才返回。如果未发生变化，就会进入阻塞状态。指定超时时间就是为了防止这种情况的发生。通过声明上述结构体变量，将秒数填入tv_sec成员，将毫秒数填入tv_usec成员，然后将结构体的地址值传递到select函数的最后一个参数。此时，即使文件描述符中未发生变化，只要过了指定时间，也可以从函数中返回。不过这种情况下，select函数返回0。因此，可以通过返回值了解返回原因。如果不想设置超时，则传递NULL参数。

调用 select 函数后查看结果

虽未给出具体示例，但图12-5中的步骤一“select函数调用前的所有准备工作”已讲解完毕，同时也介绍了select函数。而函数调用后查看结果也同样重要。我们已讨论过select函数的返回值，如果返回大于0的整数，说明相应数量的文件描述符发生变化。