

25 | 使用阻塞I/O和进程模型：最传统的方式

2019-10-04 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 08:50 大小 8.10M



你好，我是盛延敏，这里是网络编程实战第 25 讲，欢迎回来。

上一讲中，我们讲到了 C10K 问题，并引入了解决 C10K 问题的各种解法。其中，最简单也是最有效的一种解决方法就是为每个连接创建一个独立的进程去服务。那么，到底如何为每个连接客户创建一个进程来服务呢？在这其中，又需要特别注意什么呢？今天我们就围绕这部分内容展开，期望经过今天的学习，你对父子进程、僵尸进程、使用进程处理连接等有一个比较直观的理解。

父进程和子进程

我们知道，进程是程序执行的最小单位，一个进程有完整的地址空间、程序计数器等，如果想创建一个新的进程，使用函数 `fork` 就可以。

```
1 pid_t fork(void)
2 返回：在子进程中为 0，在父进程中为子进程 ID，若出错则为 -1
```

如果你是第一次使用这个函数，你会觉得难以理解的地方在于，虽然我们的程序调用 `fork` 一次，它却在父、子进程里各返回一次。在调用该函数的进程（即为父进程）中返回的是新派生的进程 ID 号，在子进程中返回的值为 0。想要知道当前执行的进程到底是父进程，还是子进程，只能通过返回值来进行判断。

`fork` 函数实现的时候，实际上会把当前父进程的所有相关值都克隆一份，包括地址空间、打开的文件描述符、程序计数器等，就连执行代码也会拷贝一份，新派生的进程的表现行为和父进程近乎一样，就好像是派生进程调用过 `fork` 函数一样。为了区别两个不同的进程，实现者可以通过改变 `fork` 函数的栈空间值来判断，对应到程序中就是返回值的不同。

这样就形成了文稿中的编程范式：

```
1 if(fork() == 0){
2     do_child_process(); // 子进程执行代码
3 }else{
4     do_parent_process(); // 父进程执行代码
5 }
```

当一个子进程退出时，系统内核还保留了该进程的若干信息，比如退出状态。这样的进程如果不回收，就会变成僵尸进程。在 Linux 下，这样的“僵尸”进程会被挂到进程号为 1 的 `init` 进程上。所以，由父进程派生出来的子进程，也必须由父进程负责回收，否则子进程就会变成僵尸进程。僵尸进程会占用不必要的内存空间，如果量多到了一定数量级，就会耗尽我们的系统资源。

有两种方式可以在子进程退出后回收资源，分别是调用 `wait` 和 `waitpid` 函数。


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

函数 `wait` 和 `waitpid` 都可以返回两个值，一个是函数返回值，表示已终止子进程的进程 ID 号，另一个则是通过 `statloc` 指针返回子进程终止的实际状态。这个状态可能的值为正常终止、被信号杀死、作业控制停止等。

如果没有已终止的子进程，而是有一个或多个子进程在正常运行，那么 `wait` 将阻塞，直到第一个子进程终止。

`waitpid` 可以认为是 `wait` 函数的升级版，它的参数更多，提供的控制权也更多。`pid` 参数允许我们指定任意想等待终止的进程 ID，值 `-1` 表示等待第一个终止的子进程。`options` 参数给了我们更多的控制选项。

处理子进程退出的方式一般是注册一个信号处理函数，捕捉信号 `SIGCHLD` 信号，然后在信号处理函数里调用 `waitpid` 函数来完成子进程资源的回收。`SIGCHLD` 是子进程退出或者中断时由内核向父进程发出的信号，默认这个信号是忽略的。所以，如果想在子进程退出时能回收它，需要像下面一样，注册一个 `SIGCHLD` 函数。

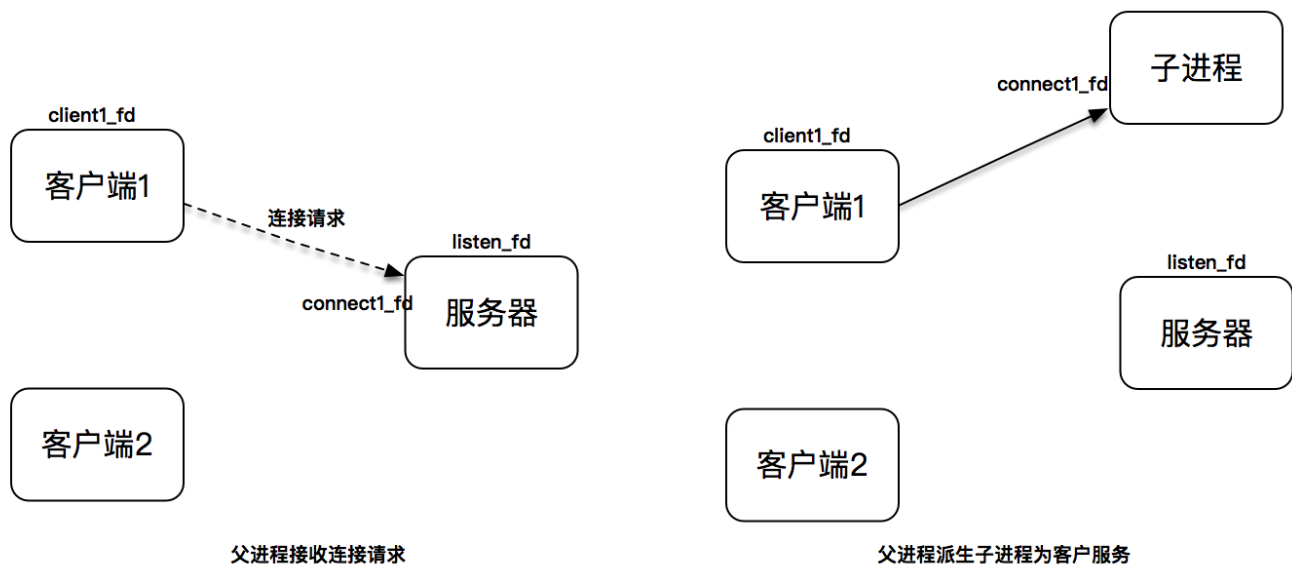
 复制代码

```
1 signal(SIGCHLD, sigchld_handler);
```

阻塞 I/O 的进程模型

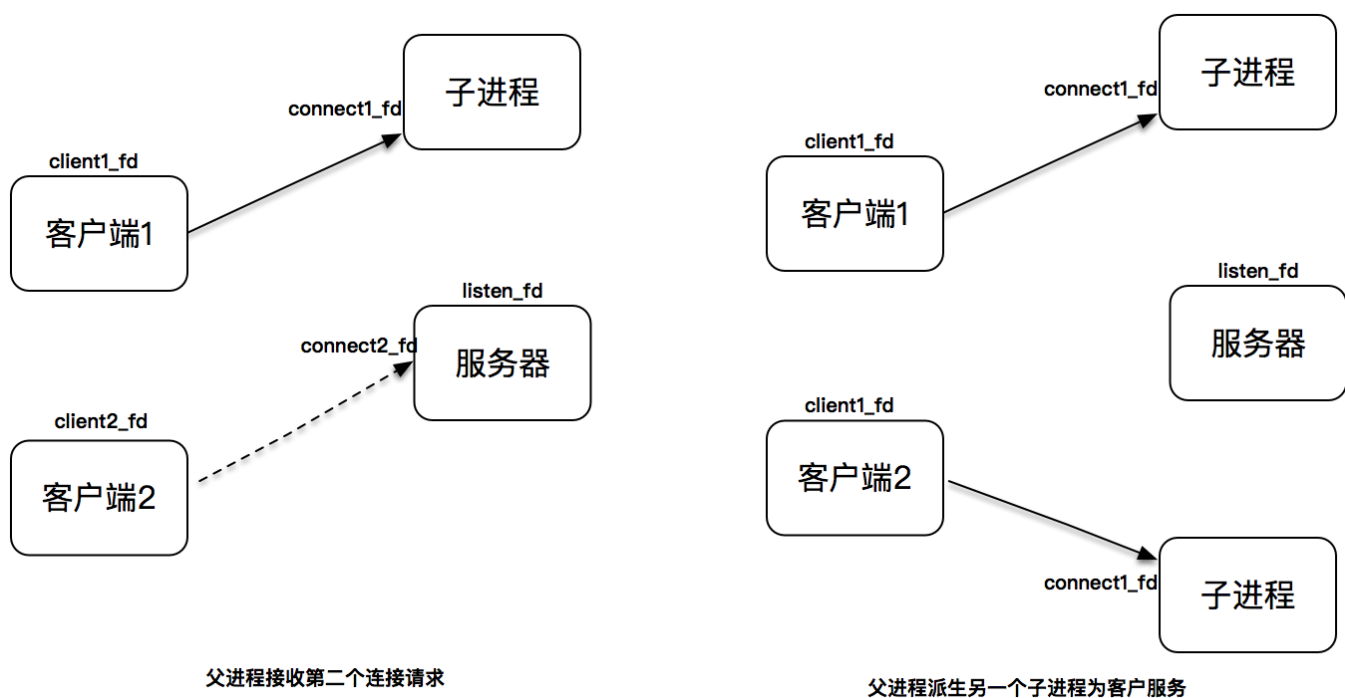
为了说明使用阻塞 I/O 和进程模型，我们假设有两个客户端，服务器初始监听在套接字 `lisnted_fd` 上。当第一个客户端发起连接请求，连接建立后产生出连接套接字，此时，父进程派生出一个子进程，在子进程中，使用连接套接字和客户端通信，因此子进程不需要关心监听套接字，只需要关心连接套接字；父进程则相反，将客户服务交给子进程来处理，因此父进程不需要关心连接套接字，只需要关心监听套接字。

这张图描述了从连接请求到连接建立，父进程派生子进程为客户服务。



假设父进程之后又接收了新的连接请求，从 `accept` 调用返回新的已连接套接字，父进程又派生出另一个子进程，这个子进程用第二个已连接套接字为客户端服务。

这张图同样描述了这个过程。



现在，服务器端的父进程继续监听在套接字上，等待新的客户连接到来；两个子进程分别使用两个不同的连接套接字为两个客户服务。

程序讲解

我们将前面的内容串联起来，就是下面完整的一个基于进程模型的服务器端程序。


```
1 #include "lib/common.h"
2
3 #define MAX_LINE 4096
4
5 char rot13_char(char c) {
6     if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
7         return c + 13;
8     else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
9         return c - 13;
10    else
11        return c;
12 }
13
14 void child_run(int fd) {
15     char outbuf[MAX_LINE + 1];
16     size_t outbuf_used = 0;
17     ssize_t result;
18
19     while (1) {
20         char ch;
21         result = recv(fd, &ch, 1, 0);
22         if (result == 0) {
23             break;
24         } else if (result == -1) {
25             perror("read");
26             break;
27         }
28
29         if (outbuf_used < sizeof(outbuf)) {
30             outbuf[outbuf_used++] = rot13_char(ch);
31         }
32
33         if (ch == '\n') {
34             send(fd, outbuf, outbuf_used, 0);
35             outbuf_used = 0;
36             continue;
37         }
38     }
39 }
40
41 void sigchld_handler(int sig) {
42     while (waitpid(-1, 0, WNOHANG) > 0);
43     return;
44 }
45
46 int main(int c, char **v) {
47     int listener_fd = tcp_server_listen(SERV_PORT);
48     signal(SIGCHLD, sigchld_handler);
49     while (1) {
50         struct sockaddr_storage ss;
```

```

51     socklen_t slen = sizeof(ss);
52     int fd = accept(listener_fd, (struct sockaddr *) &ss, &slen);
53     if (fd < 0) {
54         error(1, errno, "accept failed");
55         exit(1);
56     }
57
58     if (fork() == 0) {
59         close(listener_fd);
60         child_run(fd);
61         exit(0);
62     } else {
63         close(fd);
64     }
65 }
66
67 return 0;
68 }

```

程序的 48 行注册了一个信号处理函数，用来回收子进程资源。函数 `sigchld_handler`，在一个循环体内调用了 `waitpid` 函数，以便回收所有已终止的子进程。这里选项 `WNOHANG` 用来告诉内核，即使还有未终止的子进程也不要阻塞在 `waitpid` 上。注意这里不可以使用 `wait`，因为 `wait` 函数在有未终止子进程的情况下，没有办法不阻塞。


程序的 58-62 行，通过判断 `fork` 的返回值为 0，进入子进程处理逻辑。按照前面的讲述，子进程不需要关心监听套接字，故而在这里关闭掉监听套接字 `listen_fd`，之后调用 `child_run` 函数使用已连接套接字 `fd` 来进行数据读写。第 63 行，进入的是父进程处理逻辑，父进程不需要关心连接套接字，所以在这里关闭连接套接字。

还记得[第 11 讲](#)中讲到的 `close` 函数吗？我们知道，从父进程派生出的子进程，同时也会复制一份描述字，也就是说，连接套接字和监听套接字的引用计数都会被加 1，而调用 `close` 函数则会对引用计数进行减 1 操作，这样在套接字引用计数到 0 时，才可以将套接字资源回收。所以，这里的 `close` 函数非常重要，缺少了它们，就会引起服务器端资源的泄露。

`child_run` 函数中，通过一个 `while` 循环来不断和客户端进行交互，依次读出字符之后，进行了简单的转码，如果读到回车符，则将转码之后的结果通过连接套接字发送出去。这样的回显方式，显得比较有“交互感”。


实验

我们启动该服务器，监听在对应的端口 43211 上。


 复制代码

```
1 ./fork01
```

再启动两个 telnet 客户端，连接到 43211 端口，每次通过标准输入和服务端传输一些数据，我们看到，服务器和客户端的交互正常。

 复制代码

```
1 $telnet 127.0.0.1 43211
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 afasfa
6 nsnfsn
7 ]
8 telnet> quit
9 Connection closed.
```

 复制代码

```
1 $telnet 127.0.0.1 43211
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 agasgasg
6 ntnftnft
7 ]
8 telnet> quit
9 Connection closed.
```

客户端退出，服务器端也在正常工作，此时如果再通过 telnet 建立新的连接，客户端和服务端的数据传输也会正常进行。

至此，我们构建了一个完整的服务器端程序，可以并发处理多个不同的客户连接，互不干扰。

总结

使用阻塞 I/O 和进程模型，为每一个连接创建一个独立的子进程来进行服务，是一个非常简单有效的实现方式，这种方式可能很难满足高性能程序的需求，但好处在于实现简单。在实现这样的程序时，我们需要注意两点：

要注意对套接字的关闭梳理；

要注意对子进程进行回收，避免产生不必要的僵尸进程。

思考题

给大家出两道思考题：

第一道，你可以查查资料，看看有没有比较著名的程序是使用这样的模式来构建的？

第二道，程序中处理 SIGCHLD 信号时，使用了一个循环来回收处理终止的子进程，为什么要这么做呢？如果不使用循环会有什么后果？

欢迎你在评论区写下你的思考，我会和你一起交流，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。




网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 24 | C10K问题：高并发模型设计

下一篇 26 | 使用阻塞I/O和线程模型：换一种轻量的方式

精选留言 (4)

写留言



锦

2019-10-04

我知道的使用多进程模型有Nginx,Redis，不过Redis是用Reactor事件驱动模型来实现的



张立华

2019-10-04

避免僵尸进程，两种方法：1 忽略SIGCHLD，2 调用wait或waitpid

应该说 忽略SIGCHLD，最省事吧



搞怪者 🤪 🤔 💎...

2019-10-04

Nginx和php-fpm就是这样的架构。

一个wait不足够阻止僵尸进程，如果n个子进程同时停止，那么会同时发出n个SIGCHLD信号给父进程，但是信号处理函数执行一次，因为信号一般是不排队的，多个SIGCHLD只会发送一次给父进程。所以需要用循环waitpid处理，获取所有终止子进程状态。

展开 ∨



刘丹

2019-10-04

请问主进程结束的时候没有关闭listener_fd，子进程结束的时候没有关闭fd，是让操作系统在关闭进程时自动回收资源吗？

展开 ∨



