C/C++ 服務器並發

C語言與C++編程 2022-04-01 09:36

> 來自：https://subingwen.com/linux/concurrence/

1. 單線程/ 進程

在TCP 通信過程中，服務器端啟動之後可以同時和多個客戶端建立連接，並進行網絡通信，但是在介紹TCP 通信流程的時候，提供的服務器代碼卻不能完成這樣的需求，先簡單的看一下之前的服務器代碼的處理思路，再來分析代碼中的弊端：

```
// server.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>

int main()
{
    // 1. 创建监听的套接字
    int lfd = socket(AF_INET, SOCK_STREAM, 0);
    // 2. 将socket()返回值和本地的IP端口绑定到一起
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(10000);   // 大端端口
    // INADDR_ANY代表本机的所有IP, 假设有三个网卡就有三个IP地址
    // 这个宏可以代表任意一个IP地址
    addr.sin_addr.s_addr = INADDR_ANY;  // 这个宏的值为0 == 0.0.0.0
    int ret = bind(lfd, (struct sockaddr*)&addr, sizeof(addr));
    // 3. 设置监听
    ret = listen(lfd, 128);
    // 4. 阻塞等待并接受客户端连接
    struct sockaddr_in cliaddr;
    int clilen = sizeof(cliaddr);
    int cfd = accept(lfd, (struct sockaddr*)&cliaddr, &clilen);
    // 5. 和客户端通信
    while(1)
    {
        // 接收数据
        char buf[1024];
        memset(buf, 0, sizeof(buf));
```

```
        int len = read(cfd, buf, sizeof(buf));
        if(len > 0)
        {
            printf("客户端say: %s\n", buf);
            write(cfd, buf, len);
        }
        else if(len  == 0)
        {
            printf("客户端断开了连接...\n");
            break;
        }
        else
        {
            perror("read");
            break;
        }
    }
    close(cfd);
    close(lfd);
    return 0;
}
```

在上面的代碼中用到了三個會引起程序阻塞的函數，分別是：

- `accept()`：如果服務器端沒有新客戶端連接，阻塞當前進程/線程，如果檢測到新連接解除阻塞，建立連接
- `read()`：如果通信的套接字對應的讀緩衝區沒有數據，阻塞當前進程/線程，檢測到數據解除阻塞，接收數據
- `write()`：如果通信的套接字寫緩衝區被寫滿了，阻塞當前進程/線程（這種情況比較少見）

如果需要和發起新的連接請求的客戶端建立連接，那麼就必須在服務器端通過一個循環調用 `accept()` 函數，另外已經和服務器建立連接的客戶端需要和服務器通信，發送數據時的阻塞可以忽略，當接收不到數據時程序也會被阻塞，這時候就會非常矛盾，被 `accept()` 阻塞就無法通信，被 `read()` 阻塞就無法和客戶端建立新連接。因此得出一個結論，基於上述處理方式，在單線程/單進程場景下，服務器是無法處理多連接的，解決方案也有很多，常用的有四種：

1. 使用多線程實現
2. 使用多進程實現

3. 使用IO 多路轉接（復用）實現
4. 使用IO 多路轉接+ 多線程實現

2. 多進程並發

如果要編寫多進程版的並發服務器程序，首先要考慮，創建出的多個進程都是什麼角色，這樣就可以在程序中對號入座了。在Tcp 服務器端一共有兩個角色，分別是：監聽和通信，監聽是一個持續的動作，如果有新連接就建立連接，如果沒有新連接就阻塞。關於通信是需要和多個客戶端同時進行的，因此需要多個進程，這樣才能達到互不影響的效果。進程也有兩大類：父進程和子進程，通過分析我們可以這樣分配進程：

父進程：

- 負責監聽，處理客戶端的連接請求，也就是在父進程中循環調用accept() 函數
- 創建子進程：建立一個新的連接，就創建一個新的子進程，讓這個子進程和對應的客戶端通信
- 回收子進程資源：子進程退出回收其內核PCB 資源，防止出現殭屍進程

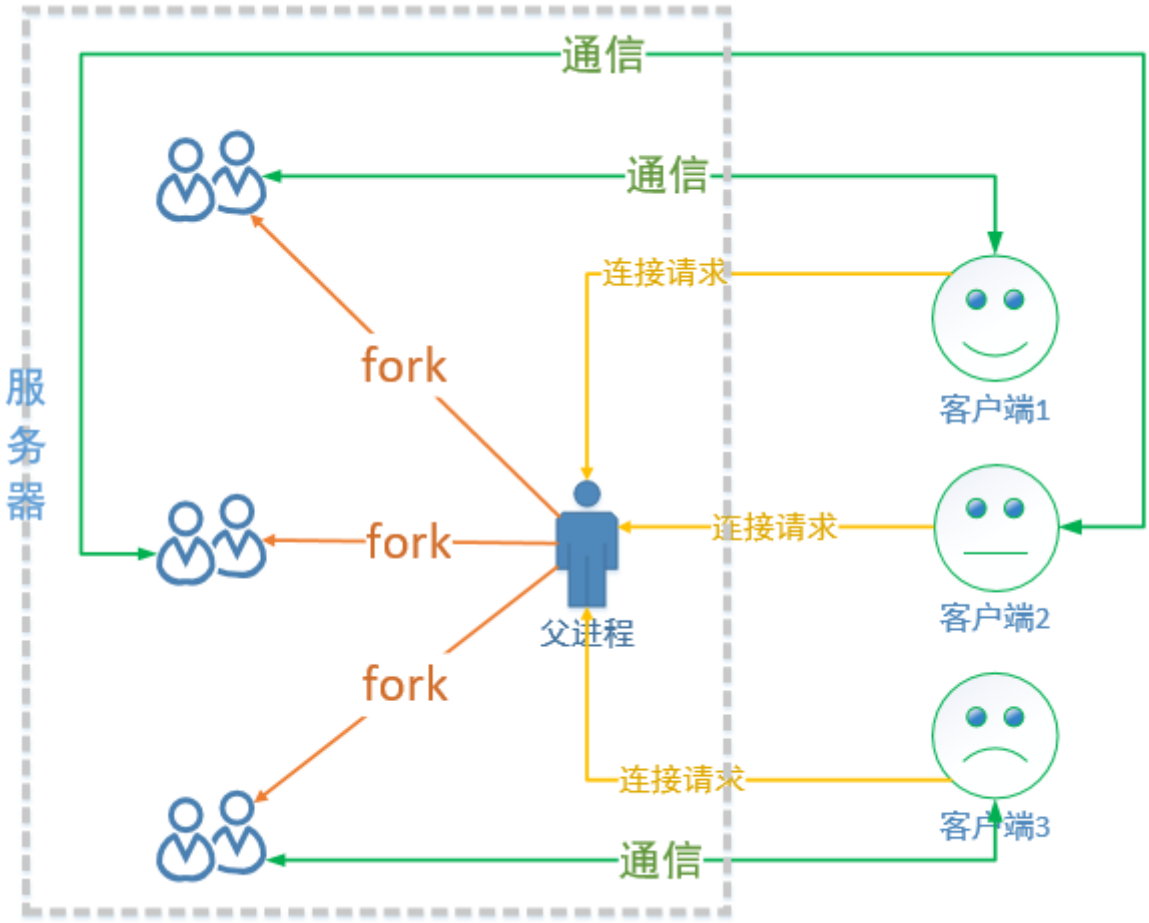
子進程：

- 負責通信，基於父進程建立新連接之後得到的文件描述符，和對應的客戶端完成數據的接收和發送。
- 發送數據：`send() / write()`
- 接收數據：`recv() / read()`

在多進程版的服務器端程序中，多個進程是有血緣關係，對應有血緣關係的進程來說，還需要想明白他們有哪些資源是可以被繼承的，哪些資源是獨占的，以及一些其他細節：

- 子進程是父進程的拷貝，在子進程的內核區PCB 中，文件描述符也是可以被拷貝的，因此在父進程可以使用的文件描述符在子進程中也有一份，並且可以使用它們做和父進程一樣的事情。
- 父子進程有用各自的獨立的虛擬地址空間，因此所有的資源都是獨占的
- 為了節省系統資源，對於只有在父進程才能用到的資源，可以在子進程中將其釋放掉，父進程亦如此。

- 由於需要在父進程中做 `accept()` 操作，並且要釋放子進程資源，如果想要更高效一下可以使用信號的方式處理



多進程版並發TCP 服務器示例代碼如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <signal.h>
#include <sys/wait.h>
#include <errno.h>

// 信号处理函数
void callback(int num)
{
    while(1)
    {
        pid_t pid = waitpid(-1, NULL, WNOHANG);
        if(pid <= 0)
        {
            printf("子进程正在运行, 或者子进程被回收完毕了\n");
```

```
            break;
        }
        printf("child die, pid = %d\n", pid);
    }
}

int childWork(int cfd);
int main()
{
    // 1. 创建监听的套接字
    int lfd = socket(AF_INET, SOCK_STREAM, 0);
    if(lfd == -1)
    {
        perror("socket");
        exit(0);
    }

    // 2. 将socket()返回值和本地的IP端口绑定到一起
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(10000);   // 大端端口
    // INADDR_ANY代表本机的所有IP, 假设有三个网卡就有三个IP地址
    // 这个宏可以代表任意一个IP地址
    // 这个宏一般用于本地的绑定操作
    addr.sin_addr.s_addr = INADDR_ANY;  // 这个宏的值为0 == 0.0.0.0
    //    inet_pton(AF_INET, "192.168.237.131", &addr.sin_addr.s_addr);
    int ret = bind(lfd, (struct sockaddr*)&addr, sizeof(addr));
    if(ret == -1)
    {
        perror("bind");
        exit(0);
    }

    // 3. 设置监听
    ret = listen(lfd, 128);
    if(ret == -1)
    {
        perror("listen");
        exit(0);
    }

    // 注册信号的捕捉
    struct sigaction act;
    act.sa_flags = 0;
    act.sa_handler = callback;
    sigemptyset(&act.sa_mask);
    sigaction(SIGCHLD, &act, NULL);

    // 接受多个客户端连接, 对需要循环调用 accept
```

```
while(1)
{
    // 4. 阻塞等待并接受客户端连接
    struct sockaddr_in cliaddr;
    int clilen = sizeof(cliaddr);
    int cfd = accept(lfd, (struct sockaddr*)&cliaddr, &clilen);
    if(cfd == -1)
    {
        if(errno == EINTR)
        {
            // accept调用被信号中断了, 解除阻塞, 返回了-1
            // 重新调用一次accept
            continue;
        }
        perror("accept");
        exit(0);

    }
    // 打印客户端的地址信息
    char ip[24] = {0};
    printf("客户端的IP地址: %s, 端口: %d\n",
           inet_ntop(AF_INET, &cliaddr.sin_addr.s_addr, ip, sizeof(ip)),
           ntohs(cliaddr.sin_port));
    // 新的连接已经建立了, 创建子进程, 让子进程和这个客户端通信
    pid_t pid = fork();
    if(pid == 0)
    {
        // 子进程 -> 和客户端通信
        // 通信的文件描述符cfd被拷贝到子进程中
        // 子进程不负责监听
        close(lfd);
        while(1)
        {
            int ret = childWork(cfd);
            if(ret <=0)
            {
                break;
            }
        }
        // 退出子进程
        close(cfd);
        exit(0);
    }
    else if(pid > 0)
    {
        // 父进程不和客户端通信
        close(cfd);
    }
}
```

```
    return 0;
}


// 5. 和客户端通信
int childWork(int cfd)
{

    // 接收数据
    char buf[1024];
    memset(buf, 0, sizeof(buf));
    int len = read(cfd, buf, sizeof(buf));
    if(len > 0)
    {
        printf("客户端say: %s\n", buf);
        write(cfd, buf, len);
    }
    else if(len  == 0)
    {
        printf("客户端断开了连接...\n");
    }
    else
    {
        perror("read");
    }

    return len;
}
```

在上面的示例代碼中，父子進程中分別關掉了用不到的文件描述符（父進程不需要通信，子進程也不需要監聽）。如果客戶端主動斷開連接，那麼服務器端負責和客戶端通信的子進程也就退出了，子進程退出之後會給父進程發送一個叫做 `SIGCHLD` 的信號，在父進程中通過 `sigaction()` 函數捕捉了該信號，通過回調函數 `callback()` 中的 `waitpid()` 對退出的子進程進行了資源回收。

另外還有一個細節要說明一下，這是父進程的處理代碼：

```
int cfd = accept(lfd, (struct sockaddr*)&cliaddr, &clilen);
while(1)
{
        int cfd = accept(lfd, (struct sockaddr*)&cliaddr, &clilen);
        if(cfd == -1)
        {
            if(errno == EINTR)
```

```
            {
                // accept调用被信号中断了, 解除阻塞, 返回了-1
                // 重新调用一次accept
                continue;
            }
            perror("accept");
            exit(0);

        }
    }
```

如果父進程調用 `accept()` 函數沒有檢測到新的客戶端連接，父進程就阻塞在這兒了，這時候有子進程退出了，發送信號給父進程，父進程就捕捉到了這個信號 `SIGCHLD` ，由於信號的優先級很高，會打斷代碼正常的執行流程，因此父進程的阻塞被中斷，轉而去處理這個信號對應的函數 `callback()` ，處理完畢，再次回到 `accept()` 位置，但是這是已經無法阻塞了，函數直接返回- 1，此時函數調用失敗，錯誤描述為 `accept: Interrupted system call` ，對應的錯誤號為 `EINTR` ，由於代碼是被信號中斷導致的錯誤，所以可以在程序中對這個錯誤號進行判斷，讓父進程重新調用 `accept()` ，繼續阻塞或者接受客戶端的新連接。

3. 多線程並發

編寫多線程版的並發服務器程序和多進程思路差不多，考慮明白了對號入座即可。多線程中的線程有兩大類：主線程（父線程）和子線程，他們分別要在服務器端處理監聽和通信流程。根據多進程的處理思路，就可以這樣設計了：

主線程：

- 負責監聽，處理客戶端的連接請求，也就是在父進程中循環調用 `accept()` 函數
- 創建子線程：建立一個新的連接，就創建一個新的子進程，讓這個子進程和對應的客戶端通信
- 回收子線程資源：由於回收需要調用阻塞函數，這樣就會影響accept()，直接做線程分離即可。

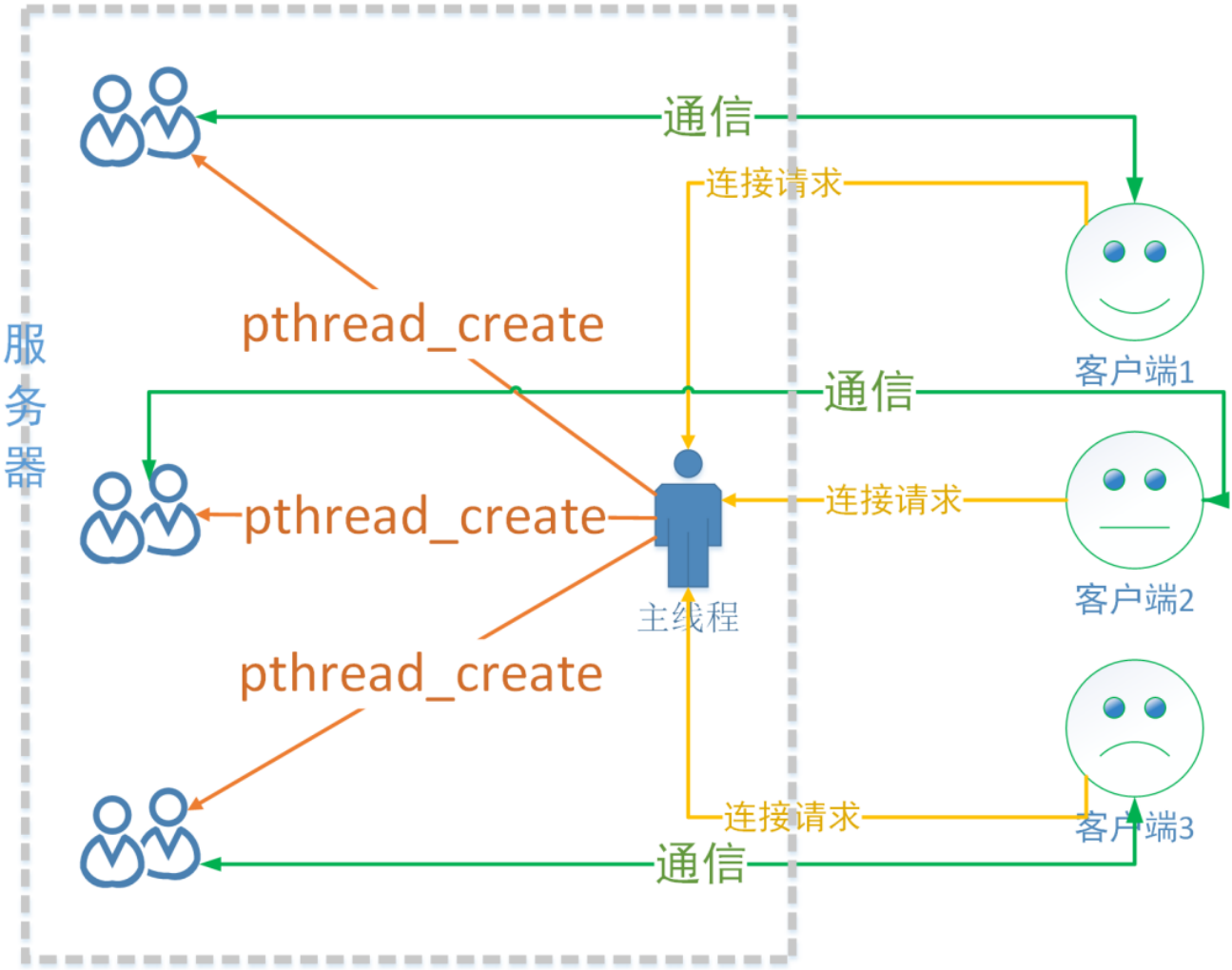
子線程：

- 負責通信，基於主線程建立新連接之後得到的文件描述符，和對應的客戶端完成數據的接收和發送。

- 發送數據：`send() / write()`
- 接收數據：`recv() / read()`

在多線程版的服務器端程序中，多個線程共用同一個地址空間，有些數據是共享的，有些數據的獨占的，下面來分析一些其中的一些細節：

- 同一地址空間中的多個線程的棧空間是獨占的
- 多個線程共享全局數據區，堆區，以及內核區的文件描述符等資源，因此需要注意數據覆蓋問題，並且在多個線程訪問共享資源的時候，還需要進行線程同步。



多線程版Tcp 服務器示例代碼如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <pthread.h>

struct SockInfo
```

```
struct SockInfo
{
    int fd;                         // 通信
    pthread_t tid;                  // 线程ID
    struct sockaddr_in addr;        // 地址信息
};

struct SockInfo infos[128];

void* working(void* arg)
{
    while(1)
    {
        struct SockInfo* info = (struct SockInfo*)arg;
        // 接收数据
        char buf[1024];
        int ret = read(info->fd, buf, sizeof(buf));
        if(ret == 0)
        {
            printf("客户端已经关闭连接...\n");
            info->fd = -1;
            break;
        }
        else if(ret == -1)
        {
            printf("接收数据失败...\n");
            info->fd = -1;
            break;
        }
        else
        {
            write(info->fd, buf, strlen(buf)+1);
        }
    }
    return NULL;
}

int main()
{
    // 1. 创建用于监听的套接字
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if(fd == -1)
    {
        perror("socket");
        exit(0);
    }

    // 2. 绑定
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;          // ipv4
```

```
    addr.sin_port = htons(8989);         // 字节序应该是网络字节序
    addr.sin_addr.s_addr =  INADDR_ANY; // == 0, 获取IP的操作交给了内核
    int ret = bind(fd, (struct sockaddr*)&addr, sizeof(addr));
    if(ret == -1)
    {
        perror("bind");
        exit(0);
    }

    // 3.设置监听
    ret = listen(fd, 100);
    if(ret == -1)
    {
        perror("listen");
        exit(0);
    }

    // 4. 等待, 接受连接请求
    int len = sizeof(struct sockaddr);

    // 数据初始化
    int max = sizeof(infos) / sizeof(infos[0]);
    for(int i=0; i<max; ++i)
    {
        bzero(&infos[i], sizeof(infos[i]));
        infos[i].fd = -1;
        infos[i].tid = -1;
    }

    // 父进程监听, 子进程通信
    while(1)
    {
        // 创建子线程
        struct SockInfo* pinfo;
        for(int i=0; i<max; ++i)
        {
            if(infos[i].fd == -1)
            {
                pinfo = &infos[i];
                break;
            }
            if(i == max-1)
            {
                sleep(1);
                i--;
            }
        }

        int connfd = accept(fd, (struct sockaddr*)&pinfo->addr, &len);
```

```
        printf("parent thread, connfd: %d\n", connfd);
        if(connfd == -1)
        {
            perror("accept");
            exit(0);
        }
        pinfo->fd = connfd;
        pthread_create(&pinfo->tid, NULL, working, pinfo);
        pthread_detach(pinfo->tid);
    }

    // 释放资源
    close(fd);  // 监听

    return 0;
}
```

在編寫多線程版並發服務器代碼的時候，需要注意父子線程共用同一個地址空間中的文件描述符，因此每當在主線程中建立一個新的連接，都需要將得到文件描述符值保存起來，不能在同一變量上進行覆蓋，這樣做丟失了之前的文件描述符值也就不知道怎麼和客戶端通信了。

在上面示例代碼中是將成功建立連接之後得到的用於通信的文件描述符值保存到了一個全局數組中，每個子線程需要和不同的客戶端通信，需要的文件描述符值也就不一樣，只要保證存儲每個有效文件描述符值的變量對應不同的內存地址，在使用的時候就不會發生數據覆蓋的現象，造成通信數據的混亂了。

> 鏈接：https://subingwen.com/linux/concurrence/

--- EOF ---

喜歡此內容的人還喜歡

這些Shell 分析服務器日誌命令集錦，優秀！

高效運維

C語言：結構體就這樣被攻克了！

玩轉嵌入式

Apache 架構師總結的30 條架構原則

架構師