一、tcp 并发服务器概述

一个好的服务器,一般都是并发服务器（同一时刻可以响应多个客户端的请求）。并发服务器设计技术一般有：多进程服务器、多线程服务器、I/O 复用服务器等。

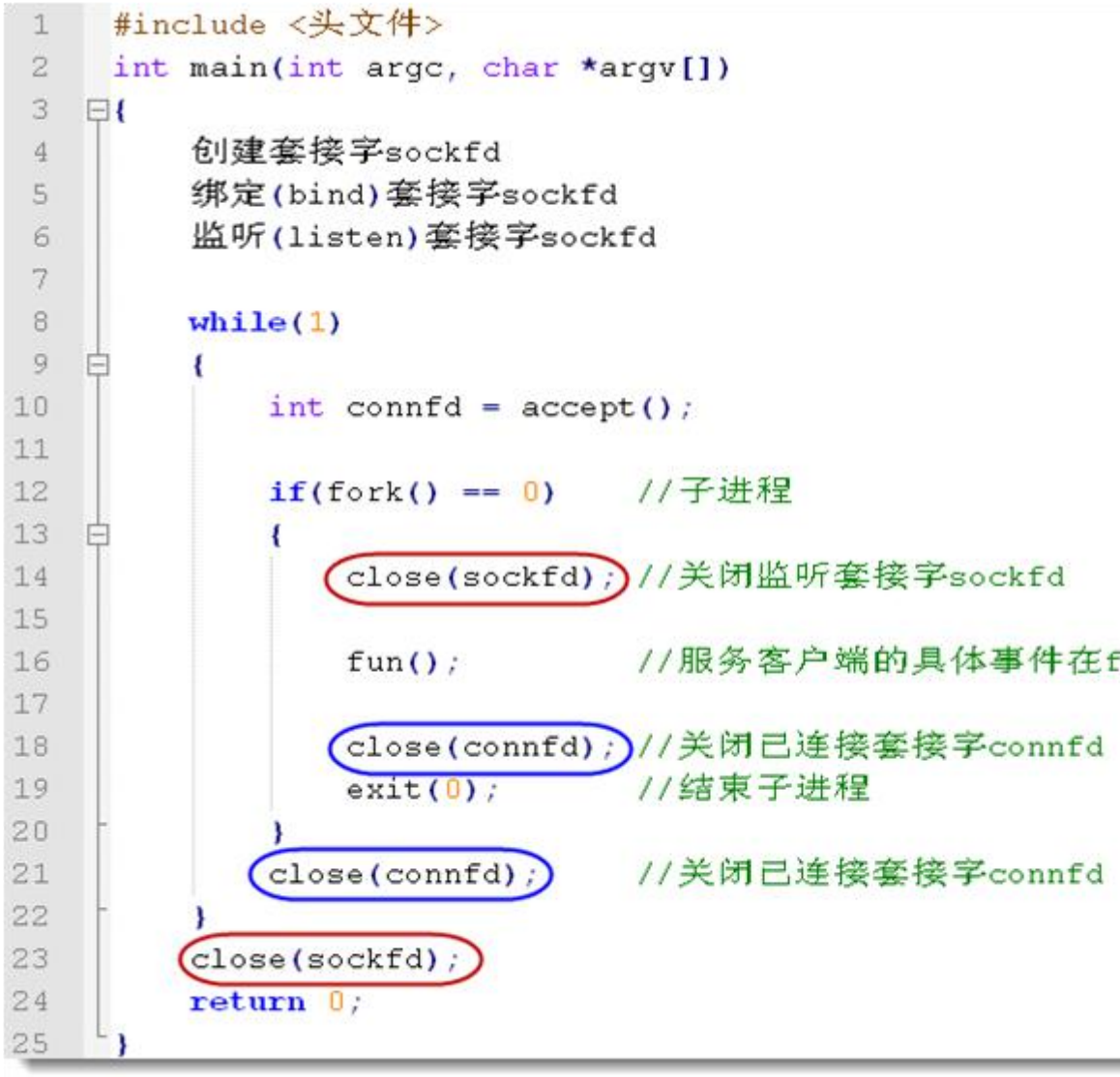
二、多进程并发服务器

在 Linux 环境下多进程的应用很多,其中最主要的就是网络/客户服务器。多进程服务器是当客户有请求时，服务器用一个子进程来处理客户请求。父进程继续等待其它客户的请求。这种方法的优点是当客户有请求时，服务器能及时处理客户，特别是在客户服务器交互系统中。对于一个 TCP 服务器,客户与服务器的连接可能并不马上关闭，可能会等到客户提交某些数据后再关闭，这段时间服务器端的进程会阻塞，所以这时操作系统可能调度其它客户服务进程，这比起循环服务器大大提高了服务性能。

发达

tcp 多进程并发服务器

TCP 并发服务器的思想是每一个客户机的请求并不由服务器直接处理，而是由服务器创建一个子进程来处理。



```
#include <头文件>
int main(int argc, char *argv[])
{
    创建套接字sockfd
    绑定(bind)套接字sockfd
    监听(listen)套接字sockfd

    while(1)
    {
        int connfd = accept();

        if(fork() == 0)     //子进程
        {
            close(sockfd);//关闭监听套接字sockfd

            fun();          //服务客户端的具体事件在f

            close(connfd);//关闭已连接套接字connfd
            exit(0);        //结束子进程
        }
        close(connfd);      //关闭已连接套接字connfd
    }
    close(sockfd);
    return 0;
}
```

tcp 多进程并发服务器参考代码：

[csharp] view plain copy

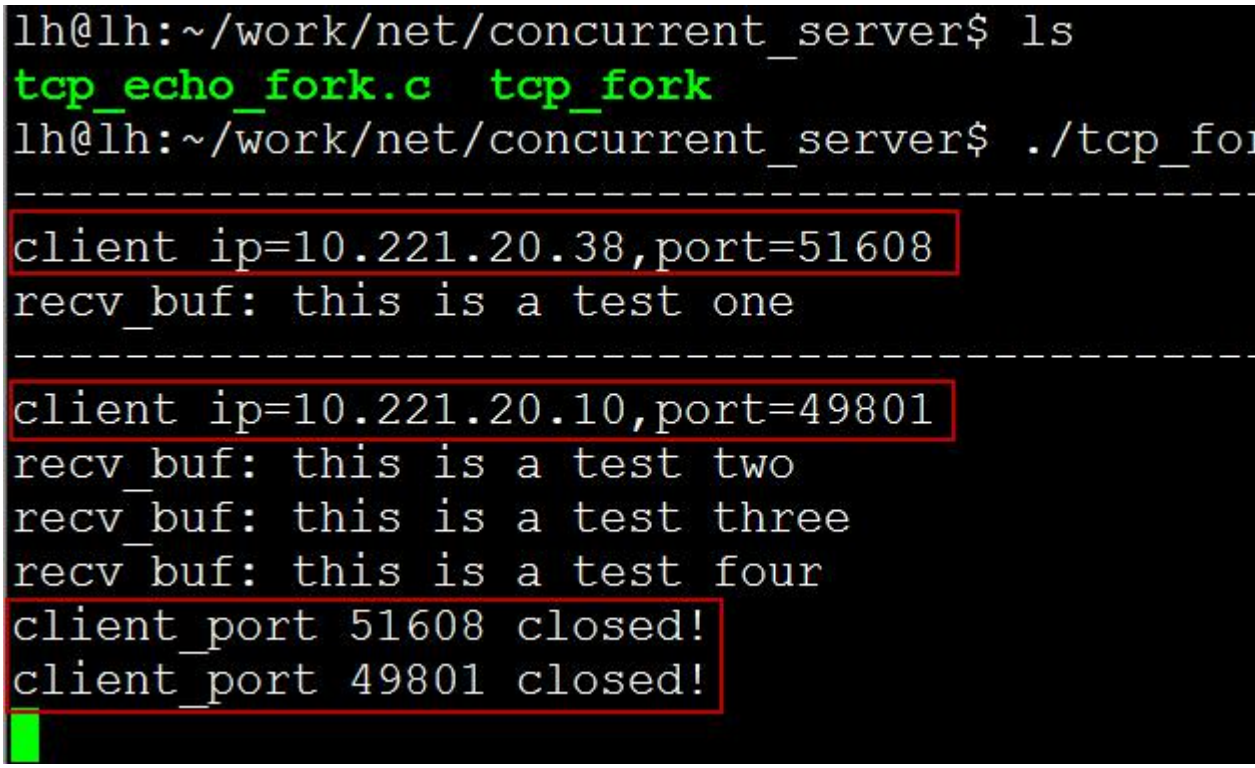
```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <sys/socket.h>
6. #include <netinet/in.h>
7. #include <arpa/inet.h>
8.
```

```
/*************************************************************
**********
函数名称:    void main(int argc, char *argv[])
函数功能:    主函数，用进程建立一个 TCP Echo Server
函数参数:    无
函数返回:    无
*************************************************************
**********/
int main(int argc, char *argv[])
{
    unsigned short port = 8080;     // 本地端口

    //1.创建 tcp 套接字
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0)
    {
        perror("socket");
        exit(-1);
    }

    //配置本地网络信息
    struct sockaddr_in my_addr;
    bzero(&my_addr, sizeof(my_addr));     // 清空
    my_addr.sin_family = AF_INET;         // IPv4
    my_addr.sin_port   = htons(port);     // 端口
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY); // ip

    //2.绑定
    int err_log = bind(sockfd, (struct sockaddr*)&my_addr, siz
eof(my_addr));
    if( err_log != 0)
    {
        perror("binding");
        close(sockfd);
        exit(-1);
    }

    //3.监听，套接字变被动
    err_log = listen(sockfd, 10);
    if(err_log != 0)
    {
        perror("listen");
        close(sockfd);
        exit(-1);
```

```
50.     }
51.
52.     while(1) //主进程 循环等待客户端的连接
53.     {
54.
55.         char cli_ip[INET_ADDRSTRLEN] = {0};
56.         struct sockaddr_in client_addr;
57.         socklen_t cliaddr_len = sizeof(client_addr);
58.
59.         // 取出客户端已完成的连接
60.         int connfd = accept(sockfd, (struct sockaddr*)&client_addr, &cliaddr_len);
61.         if(connfd < 0)
62.         {
63.             perror("accept");
64.             close(sockfd);
65.             exit(-1);
66.         }
67.
68.         pid_t pid = fork();
69.         if(pid < 0){
70.             perror("fork");
71.             _exit(-1);
72.         }else if(0 == pid){ //子进程 接收客户端的信息，并发还给客户端
73.             /*关闭不需要的套接字可节省系统资源，
74.               同时可避免父子进程共享这些套接字
75.               可能带来的不可预计的后果
76.             */
77.             close(sockfd);   // 关闭监听套接字，这个套接字是从父进程继承过来
78.
79.             char recv_buf[1024] = {0};
80.             int recv_len = 0;
81.
82.             // 打印客户端的 ip 和端口
83.             memset(cli_ip, 0, sizeof(cli_ip)); // 清空
84.             inet_ntop(AF_INET, &client_addr.sin_addr, cli_ip, INET_ADDRSTRLEN);
85.             printf("----------------------------------------\n");
86.             printf("client ip=%s,port=%d\n", cli_ip,ntohs(client_addr.sin_port));
87.
```

```
            // 接收数据
            while( (recv_len = recv(connfd, recv_buf, sizeof(recv_buf), 0)) > 0 )
            {
                printf("recv_buf: %s\n", recv_buf); // 打印数据
                send(connfd, recv_buf, recv_len, 0); // 给客户端回数据
            }

            printf("client_port %d closed!\n", ntohs(client_addr.sin_port));
            close(connfd);    //关闭已连接套接字
            exit(0);

        }
        else if(pid > 0){    // 父进程
            close(connfd);    //关闭已连接套接字
        }
    }

    close(sockfd);

    return 0;
}
```

运行结果：



tcp 多线程并发服务器

多线程服务器是对多进程服务器的改进，由于多进程服务器在创建进程时要消耗较大的系统资源，所以用线程来取代进程，这样服务处理程序可以较快的创建。据统计，创建线程与创建进程要快 10100 倍，所以又把线程称为“轻量级”进程。线程与进程不同的是：一个进程内的所有线程共享相同的全局内存、全局变量等信息，这种机制又带来了同步问题。

tcp 多线程并发服务器框架:

```
#include <头文件>
int main(int argc, char *argv[])
{
    创建套接字sockfd
    绑定(bind)套接字sockfd
    监听(listen)套接字sockfd

    while(1)
    {
        int connfd = accept();
        pthread_t tid;
        pthread_create(&tid, NULL, (void *)client_fun, (v
        pthread_detach(tid);
    }
    close(sockfd);//关闭监听套接字
    return 0;
}
void *client_fun(void *arg)
{
    int connfd = (int)arg;
    fun();//服务于客户端的具体程序
    close(connfd);
}
```

我们在使用多线程并发服务器时，直接使用以上框架，我们仅仅修改client_fun()里面的内容。

代码示例：

[csharp] view plain copy

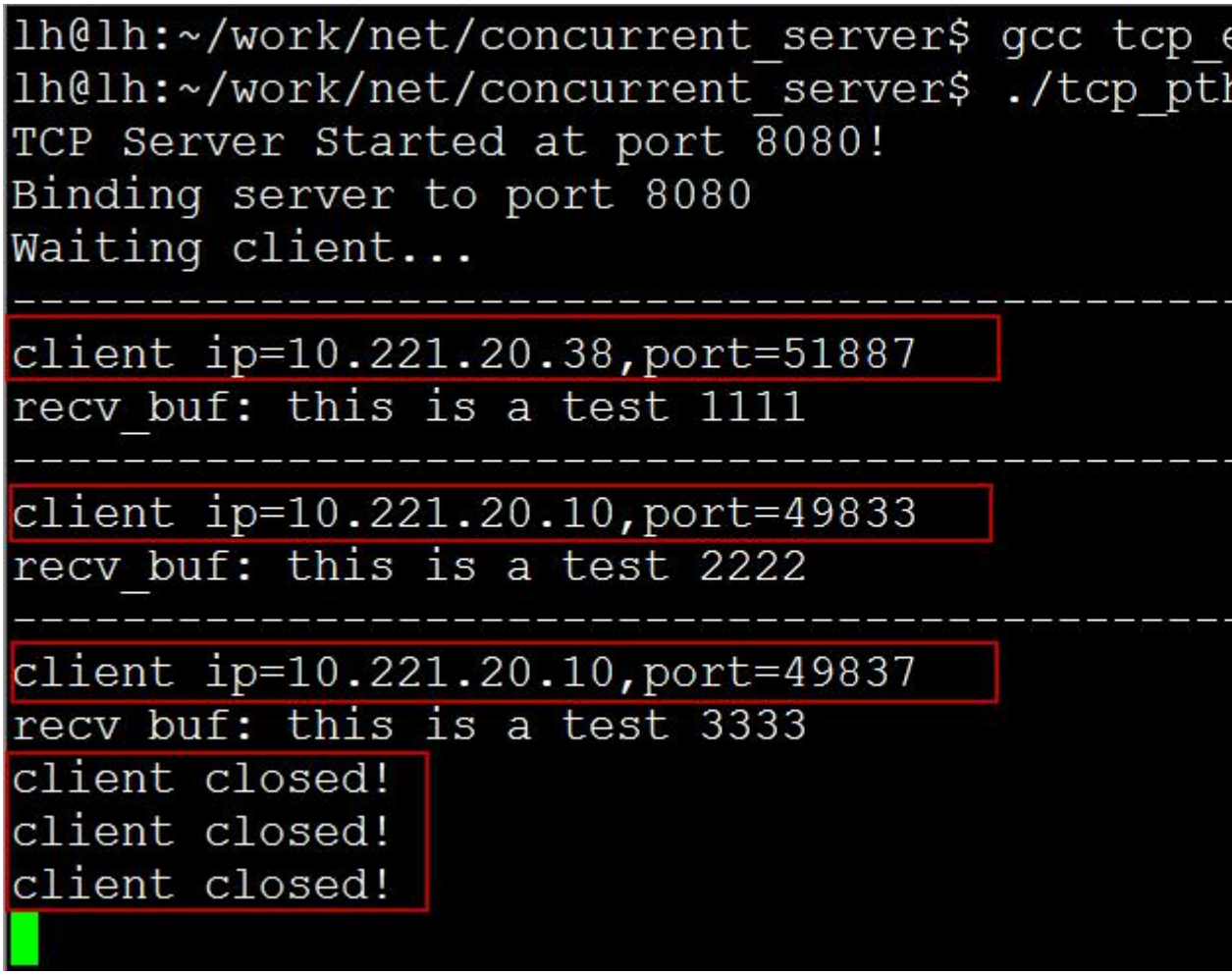
```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <sys/socket.h>
6. #include <netinet/in.h>
7. #include <arpa/inet.h>
8. #include <pthread.h>
```

```
9.
10. /*************************************************************
    ***********
11. 函数名称:    void *client_fun(void *arg)
12. 函数功能:    线程函数,处理客户信息
13. 函数参数:    已连接套接字
14. 函数返回:    无
15. *************************************************************
    **********/
16. void *client_fun(void *arg)
17. {
18.     int recv_len = 0;
19.     char recv_buf[1024] = "";   // 接收缓冲区
20.     int connfd = (int)arg; // 传过来的已连接套接字
21.
22.     // 接收数据
23.     while((recv_len = recv(connfd, recv_buf, sizeof(recv_buf),
    0)) > 0)
24.     {
25.         printf("recv_buf: %s\n", recv_buf); // 打印数据
26.         send(connfd, recv_buf, recv_len, 0); // 给客户端回数
    据
27.     }
28.
29.     printf("client closed!\n");
30.     close(connfd);  //关闭已连接套接字
31.
32.     return  NULL;
33. }
34.
35. //=============================================================
    ===
36. // 语法格式:      void main(void)
37. // 实现功能:      主函数，建立一个 TCP 并发服务器
38. // 入口参数:      无
39. // 出口参数:      无
40. //=============================================================
    ===
41. int main(int argc, char *argv[])
42. {
43.     int sockfd = 0;             // 套接字
44.     int connfd = 0;
45.     int err_log = 0;
46.     struct sockaddr_in my_addr; // 服务器地址结构体
```

```
47.     unsigned short port = 8080; // 监听端口
48.     pthread_t thread_id;
49.
50.     printf("TCP Server Started at port %d!\n", port);
51.
52.     sockfd = socket(AF_INET, SOCK_STREAM, 0);   // 创建 TCP 套接字
53.     if(sockfd < 0)
54.     {
55.         perror("socket error");
56.         exit(-1);
57.     }
58.
59.     bzero(&my_addr, sizeof(my_addr));       // 初始化服务器地址
60.     my_addr.sin_family = AF_INET;
61.     my_addr.sin_port   = htons(port);
62.     my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
63.
64.     printf("Binding server to port %d\n", port);
65.
66.     // 绑定
67.     err_log = bind(sockfd, (struct sockaddr*)&my_addr, sizeof(my_addr));
68.     if(err_log != 0)
69.     {
70.         perror("bind");
71.         close(sockfd);
72.         exit(-1);
73.     }
74.
75.     // 监听，套接字变被动
76.     err_log = listen(sockfd, 10);
77.     if( err_log != 0)
78.     {
79.         perror("listen");
80.         close(sockfd);
81.         exit(-1);
82.     }
83.
84.     printf("Waiting client...\n");
85.
86.     while(1)
87.     {
```

```
        char cli_ip[INET_ADDRSTRLEN] = "";     // 用于保存客户端 IP 地址
        struct sockaddr_in client_addr;        // 用于保存客户端地址
        socklen_t cliaddr_len = sizeof(client_addr);   // 必须初始化!!!

        //获得一个已经建立的连接
        connfd = accept(sockfd, (struct sockaddr*)&client_addr, &cliaddr_len);
        if(connfd < 0)
        {
            perror("accept this time");
            continue;
        }

            // 打印客户端的 ip 和端口
            inet_ntop(AF_INET, &client_addr.sin_addr, cli_ip, INET_ADDRSTRLEN);
            printf("----------------------------------------\n");
            printf("client ip=%s,port=%d\n", cli_ip,ntohs(client_addr.sin_port));

            if(connfd > 0)
            {
                //由于同一个进程内的所有线程共享内存和变量，因此在传递参数时需作特殊处理，值传递。
                pthread_create(&thread_id, NULL, (void *)client_fun, (void *)connfd);  //创建线程
                pthread_detach(thread_id); // 线程分离，结束时自动回收资源
            }
        }

        close(sockfd);

        return 0;
    }
```

运行结果：



注意：

1. 上面 pthread_create()函数的最后一个参数是 void *类型，为啥可以传值 connfd?

[csharp] view plain copy

```
1. while(1)
2. {
3.     int connfd = accept(sockfd, (struct sockaddr *)&client_addr, &cliaddr_len);
4.     pthread_create(&thread_id, NULL, (void *)client_fun, (void *)connfd);
5.     pthread_detach(thread_id);
6. }
```

因为 void *是 4 个字节，而 connfd 为 int 类型也是 4 个字节，故可以传值。如果 connfd 为 char、short,上面传值就会出错

2. 上面 pthread_create()函数的最后一个参数是可以传地址吗？可以，但会对服务器造成不可预知的问题

[csharp] view plain copy

```
1. while(1)
2. {
3.     int connfd = accept(sockfd, (struct sockaddr *)&client_addr, &cliaddr_len);
4.     pthread_create(&thread_id, NULL, (void *)client_fun, (void *)&connfd);
5.     pthread_detach(thread_id);
6. }
```

原因：假如有多个客户端要连接这个服务器，正常的情况下，一个客户端连接对应一个 connfd，相互之间独立不受影响，但是，假如多个客户端同时连接这个服务器，A 客户端的连接套接字为 connfd，服务器正在用这个 connfd 处理数据，还没有处理完，突然来了一个 B 客户端，accept()之后又生成一个 connfd，因为是地址传

递， A 客户端的连接套接字也变成 B 这个了，这样的话，服务器肯定不能再为 A 客户端服务器了

2. 如果我们想将多个参数传给线程函数，我们首先考虑到就是结构体参数，而这时传值是行不通的，只能传递地址。

这时候，我们就需要考虑多任务的互斥或同步问题了，这里通过互斥锁来解决这个问题，确保这个结构体参数值被一个临时变量保存过后，才允许修改。

[csharp] view plain copy

```
#include <pthread.h>

pthread_mutex_t mutex;  // 定义互斥锁，全局变量

pthread_mutex_init(&mutex, NULL); // 初始化互斥锁，互斥锁默认是打开的

// 上锁，在没有解锁之前，pthread_mutex_lock()会阻塞
pthread_mutex_lock(&mutex);
int connfd = accept(sockfd, (struct sockaddr*)&client_addr, &cliaddr_len);

//给回调函数传的参数，&connfd，地址传递
pthread_create(&thread_id, NULL, (void *)client_process, (void *)&connfd);  //创建线程

// 线程回调函数
void *client_process(void *arg)
```

```
{
    int connfd = *(int *)arg; // 传过来的已连接套接字

    // 解锁，pthread_mutex_lock()唤醒，不阻塞
    pthread_mutex_unlock(&mutex);

    return  NULL;
}
```

示例代码：

[csharp] view plain copy

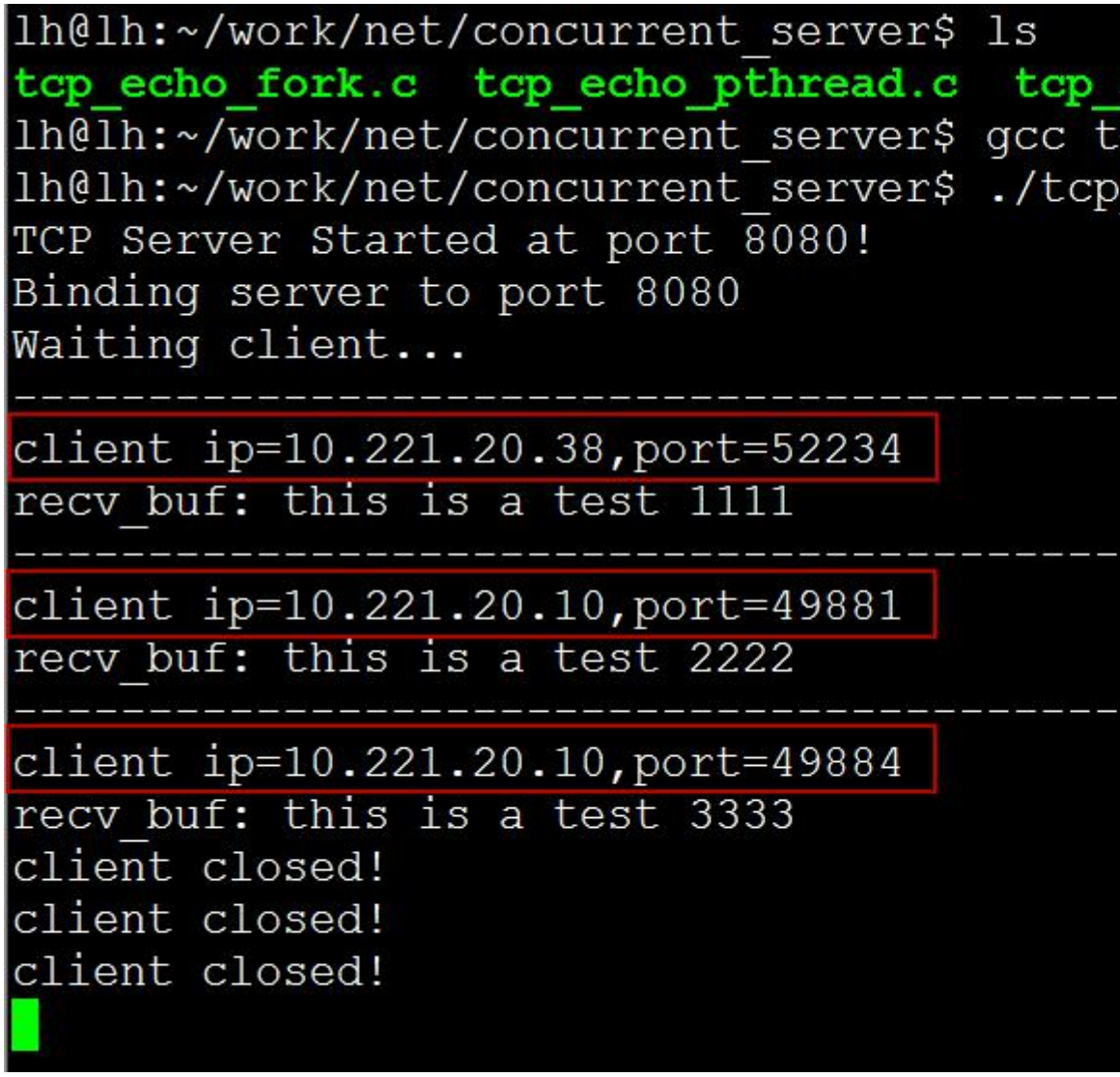
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

pthread_mutex_t mutex;  // 定义互斥锁，全局变量

/*************************************************************************
函数名称：    void *client_process(void *arg)
函数功能：    线程函数,处理客户信息
函数参数：    已连接套接字
函数返回：    无
*************************************************************************/
void *client_process(void *arg)
{
    int recv_len = 0;
    char recv_buf[1024] = "";   // 接收缓冲区
    int connfd = *(int *)arg; // 传过来的已连接套接字

    // 解锁，pthread_mutex_lock()唤醒，不阻塞
    pthread_mutex_unlock(&mutex);
```

```
26.
27.     // 接收数据
28.     while((recv_len = recv(connfd, recv_buf, sizeof(recv_buf), 0)) > 0)
29.     {
30.         printf("recv_buf: %s\n", recv_buf); // 打印数据
31.         send(connfd, recv_buf, recv_len, 0); // 给客户端回数据
32.     }
33.
34.     printf("client closed!\n");
35.     close(connfd);  //关闭已连接套接字
36.
37.     return  NULL;
38. }
39.
40. //=============================================================
41. // 语法格式:      void main(void)
42. // 实现功能:      主函数，建立一个 TCP 并发服务器
43. // 入口参数:      无
44. // 出口参数:      无
45. //=============================================================
46. int main(int argc, char *argv[])
47. {
48.     int sockfd = 0;              // 套接字
49.     int connfd = 0;
50.     int err_log = 0;
51.     struct sockaddr_in my_addr; // 服务器地址结构体
52.     unsigned short port = 8080; // 监听端口
53.     pthread_t thread_id;
54.
55.     pthread_mutex_init(&mutex, NULL); // 初始化互斥锁，互斥锁默认是打开的
56.
57.     printf("TCP Server Started at port %d!\n", port);
58.
59.     sockfd = socket(AF_INET, SOCK_STREAM, 0); // 创建 TCP 套接字
60.     if(sockfd < 0)
```

```
61.     {
62.         perror("socket error");
63.         exit(-1);
64.     }
65.
66.     bzero(&my_addr, sizeof(my_addr));      // 初始化服务器地址
67.     my_addr.sin_family = AF_INET;
68.     my_addr.sin_port   = htons(port);
69.     my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
70.
71.
72.     printf("Binding server to port %d\n", port);
73.
74.     // 绑定
75.     err_log = bind(sockfd, (struct sockaddr*)&my_addr, sizeof(my_addr));
76.     if(err_log != 0)
77.     {
78.         perror("bind");
79.         close(sockfd);
80.         exit(-1);
81.     }
82.
83.     // 监听，套接字变被动
84.     err_log = listen(sockfd, 10);
85.     if( err_log != 0)
86.     {
87.         perror("listen");
88.         close(sockfd);
89.         exit(-1);
90.     }
91.
92.     printf("Waiting client...\n");
93.
94.     while(1)
95.     {
96.         char cli_ip[INET_ADDRSTRLEN] = "";     // 用于保存客户端 IP 地址
97.         struct sockaddr_in client_addr;        // 用于保存客户端地址
```

```
98.         socklen_t cliaddr_len = sizeof(client_addr);   // 必须初始化!!!
99.
100.            // 上锁，在没有解锁之前，pthread_mutex_lock()会阻塞
101.            pthread_mutex_lock(&mutex);
102.
103.            //获得一个已经建立的连接
104.            connfd = accept(sockfd, (struct sockaddr*)&client_addr, &cliaddr_len);
105.            if(connfd < 0)
106.            {
107.                perror("accept this time");
108.                continue;
109.            }
110.
111.            // 打印客户端的 ip 和端口
112.            inet_ntop(AF_INET, &client_addr.sin_addr, cli_ip, INET_ADDRSTRLEN);
113.            printf("------------------------------------------\n");
114.            printf("client ip=%s,port=%d\n", cli_ip,ntohs(client_addr.sin_port));
115.
116.            if(connfd > 0)
117.            {
118.                //给回调函数传的参数，&connfd，地址传递
119.                pthread_create(&thread_id, NULL, (void *)client_process, (void *)&connfd);  //创建线程
120.                pthread_detach(thread_id); // 线程分离，结束时自动回收资源
121.            }
122.        }
123.
124.        close(sockfd);
125.
126.        return 0;
127.    }
```

运行结果：



注意：这种用互斥锁对服务器的运行效率有致

命的影响

代码下载：

在做网络服务的时候 tcp 并发服务端程序的编写必不可少。tcp 并发通常有几种固定的设计模式套路，他们各有优点，也各有应用之处。下面就简单的讨论下这几种模式的差异：

1。 单进程，单线程模式

在 accept 之后，就开始在这一个连接连接上的数据收接收，收到之后处理，发送，不再接收新的连接，除非这个连接的处理结束。

优点：

简单。

缺点：

因为只为一个客户端服务，所以不存在并发的可能。

应用：

用在只为一个客户端服务的时候。

2。 多进程模式

accept 返回成功时候，就为这一个连接 fork 一个进程，专门处理这个连接上的数据收发，等这个连接处理结束之后就结束这个进程。

优点：

编程相对简单，不用考虑线程间的数据同步等。

缺点：

资源消耗大。启动一个进程消耗相对比启动一个线程要消耗大很多，同时在处理很多的连接时候需要启动很多的进程多去处理，这时候对系统来说压力就会比较大。另外系统的进程数限制也需要考虑。

应用：

在客户端数据不多的时候使用很方便，比如小于 10 个客户端。

3。 多线程模式

类似多进程方式，但是针对一个连接启动一个线程。

优点：

相对多进程方式，会节约一些资源，会更加高效一些。

缺点：

相对多进程方式，增加了编程的复杂度，因为需要考虑数据同步和锁保护。另外一个进程中不能启动太多的线程。在 Linux 系统下线程在系统内部其实就是进程，线程调度按照进程调度的方式去执行的。

应用：

类似于多进程方式，适用于少量的客户端的时候。

4。 select + 多线程 模式

有一个线程专门用于监听端口，accept 返回之后就把这个描述符放入描述符集合 fd 中，一个线程用 select 去轮训描述符集合，在有数据的连接上接收数据，另外一个线程专门发送数据。当然也可以接收和发送用一个线程。描述符可以设置成非阻塞模式，也可以设置成阻塞模式。通常连接设置成非阻塞模式，发送线程独立出来。

优点：

相对前几种模式，这种模式大大提高了并发量。

缺点：

系统一般实现描述符集合是采用一个大数组，每次调用 select 的时候都会轮询这个描述符数组，当连接数很多的时候就会导致效率下降。连接数在 1000 以上时候效率会下降到不能接受。

应用：

目前 windows 和一般的 Unix 上的 tcp 并发都采用 select 方式，应该说应用还是很广泛的。

5。 epoll 方式

在 Linux2.6 版本之后，增加了 epoll。具体的使用是：一个线程专门进行端口监听，accept 接收到连接的时候，把该连接设置成非阻塞模式，把 epoll 事件设置成边缘触发方式，加入到 epoll 管理。接收线程阻塞在 epoll 的等待事件函数。另外一个线程专门用于数据发送。

优点：

由于 epoll 的实现方式先进，所以这种方式可以大规模的实现并发。我们现在的应用在一个 3 年前的 dell 的 pc server 可以实现 2 万个连接的并发，性能也是很好的。

缺点：

由于涉及了线程和非阻塞，所以会导致编码的复杂度增大一些。这种方式只适用于 Linux 2.6
内核以后。

注意：

1。 如果把 epoll 事件设置成水平触发效率就下降到类似采用 select 的水平。

2。 Unix 系统下有单个进程打开的描述符数目限制，还有系统内打开的描述符数目限制。系统内打开的描述符数目限制由软硬限制两个。硬限制是根据机器的配置而不同。软限制可以更改，但是必须小于系统的硬限制。在 suse Linux 下，可以在 root 用户下，通过 ulimit -n 数目 去修改这个限制。

应用：

Linux 下大规模的 tcp 并发。

当前 并发还有其他的方式，比如进程池，线程池等，每种模式都有它的优点和缺点，在适当的情况用合适的模式是最重要的。

如果需要很大规模的并发，经过测试，个人建议采用 epoll 方式，放弃通常的 select 方式。

本文讲述的 TCP 服务器是模仿 memcache 中的 TCP 网络处理框架，其中是基于 libevent 网络库的。

主线程只处理监听客户端的连接请求，并将请求平均分配给子线程。

子线程处理与客户端的连接以及相关业务。

每个子线程有一个“连接”队列。每个“连接”有一个“反馈”队列。

先上个流程图，要上班了，以后再解释。代码以后再上…

曾几何时我们还在寻求网络编程中 C10K 问题的解决方案，但是现在从硬件和操作系统支持来看单台服务器支持上万并发连接已经没有多少挑战性了。我们先假设单台服务器最多只能支持万级并发连接，其实对绝大多数应用来说已经远远足够了，但是对于一些拥有很大用户基数的互联网公司，往往面临的并发连接数是百万，千万，甚至腾讯的上亿（注：QQ 默认用的 UDP 协议）。虽然现在的集群，分布式技术可以为我们将并发负载分担在多台服务器上，那我们只需要扩展出数十台电脑就可以解决问题，但是我们更希望能更大的挖掘单台服务器

的资源，先努力垂直扩展，再进行水平扩展，这样可以有效的节省服务器相关的开支（硬件资源，机房，运维，电力其实也是一笔不小的开支）。那么到底一台服务器能够支持多少 TCP 并发连接呢？

常识一：文件句柄限制

在 linux 下编写网络服务器程序的朋友肯定都知道每一个 tcp 连接都要占一个文件描述符，一旦这个文件描述符使用完了，新的连接到来返回给我们的错误是“Socket/File:Can't open so many files”。

这时你需要明白操作系统对可以打开的最大文件数的限制。

- 进程限制
-
  - 执行 ulimit -n 输出 1024，说明对于一个进程而言最多只能打开 1024 个文件，所以你要采用此默认配置最多也就可以并发上千个 TCP 连接。
  -
  - 临时修改：ulimit -n 1000000，但是这种临时修改只对当前登录用户目前的使用环境有效，系统重启或用户退出后就会失效。
  -
  - 重启后失效的修改（不过我在 CentOS 6.5 下测试，重启后未发现失效）：编辑 /etc/security/limits.conf 文件， 修改后内容为
  -

```
* soft nofile 1000000
```

  -

```
* hard nofile 1000000
```

永久修改：编辑/etc/rc.local，在其后添加如下内容

ulimit -SHn 1000000

全局限制

执行 cat /proc/sys/fs/file-nr 输出 `9344 0 592026`，分别为：1.已经分配的文件句柄数，2.已经分配但没有使用的文件句柄数，3.最大文件句柄数。但在 kernel 2.6 版本中第二项的值总为 0，这并不是一个错误，它实际上意味着已经分配的文件描述符无一浪费的都已经被使用了 。

我们可以把这个数值改大些，用 root 权限修改 /etc/sysctl.conf 文件:

```
fs.file-max = 1000000
```

```
net.ipv4.ip_conntrack_max = 1000000
```

```
net.ipv4.netfilter.ip_conntrack_max = 1000000
```

常识二：端口号范围限制？

操作系统上端口号 1024 以下是系统保留的，从 1024-65535 是用户使用的。由于每个 TCP 连接都要占一个端口号，所以我们最多可以有 60000 多个并发连接。我想有这种错误思路朋友不在少数吧？（其中我过去就一直这么认为）

我们来分析一下吧

- 如何标识一个 TCP 连接：系统用一个 4 四元组来唯一标识一个 TCP 连接：{local ip, local port, remote ip, remote port}。好吧，我们拿出《UNIX 网络编程：卷一》第四章中对 accept 的讲解来看看概念性的东西，第二个参数 cliaddr 代表了客户端的 ip 地址和端口号。而我们作为服务端实际只使用了 bind 时这一个端口，说明端口号 65535 并不是并发量的限制。
-
- server 最大 tcp 连接数：server 通常固定在某个本地端口上监听，等待 client 的连接请求。不考虑地址重用（unix 的 SO_REUSEADDR 选项）的情况下，即使 server 端有多个 ip，本地监听端口也是独占的，因此 server 端 tcp 连接 4 元组中只有 remote ip（也就是 client ip）和 remote port（客户端 port）是可变的，因此最大 tcp 连接为客户端 ip 数×客户端 port 数，对 IPV4，不考虑 ip 地址分类等因素，最大 tcp 连接数约为 2 的 32 次方（ip 数）×2 的 16 次方（port 数），也就是 server 端单机最大 tcp 连接数约为 2 的 48 次方。
-

总结

上面给出的结论都是理论上的单机 TCP 并发连接数，实际上单机并发连接数肯定要受硬件资源（内存）、网络资源（带宽）的限制，至少对我们的需求现在可以做到数十万级的并发了，你的呢？

MrioTCP，超级马里奥，顾名思义，他不仅高效，而且超级简易和好玩。同时他可以是一个很简洁的 **Linux** C 开发学习工程。毫不夸张的说，如果全部掌握这一个工程，你会成为一个 Linux C 的牛人；当然，你也可以通过源码包的 mario.c(maritcp 服务器示例程序）来学习，可以很快入门上手进行 Linux C 开发。

经过两个多月的测试（编写 c++客户端测试及调优系统参数），测试结果得到单机最大带宽吞吐 1000M，测试最高 TCP 长连接 100 万，每秒处理连接数达 4 万，此时系统压力 load 值很低。总之，它可以发挥一台服务器的最大极限以提供最高性能的服务；而且经过完备测试，运行稳定且占用系统资源非常少。

他是建立在 Sourceforge 上的一个开源项目，由源码的作者冯建华(JohnFong)发起。源码可以在 Sourceforge 上下载。

sourceforge 下载：https://sourceforge.net/projects/mariotcp/files/latest/download

csdn 下载：http://download.csdn.net/detail/everlastinging/6195605

51cto 下载：http://down.51cto.com/data/935913

Getting Started

用 MarioTCP 来建立一个性能强大的 TCP 服务器非常简易！

工程源码包就是一个非常简洁的例子，生成了一个 tcp 服务器程序：maritcp。

源码包中：

mario.c 是简易例子的 main 程序，直接 make 可以编译出 maritcp，一个 tcp 服务器，业务逻辑只有一个功能：统计同时在线 socket 数、每隔 1 分钟输出一次。

mario 文件夹，MarioTCP 的核心代码，make 可以直接编译出静态的 libmario.a。MarioTCP 核心架构后续会介绍。

test 文件夹，是一个稍显简陋的客户端测试程序，通过与服务器建立连接、发送 LOGIN 包登陆服务器，此时 maritcp 服务器会使同时在线加 1，客户端断开时服务器在线数减 1。

现在讲一下如何定制一个自己业务逻辑的 tcp 服务器，只需五步：

1、初始化 SERVER

SERVER *server = init_server(conf->port, conf->workernum, conf->connnum, conf->timeout, conf->timeout);

传入参数分别是：

服务器 Listen 端口，工作线程数，每个线程支持的连接数，读超时时间，写超时时间。

workernum * connum 就是服务器支持的长连接数，一个 worker 可以轻松支持 10 万长连接。

2、实现业务逻辑函数并注册

具体业务逻辑函数请见 Function 模块。可通过 mario.h 中定义的名为“regist_*”的函数来注册。

```
/*
    *  注册业务处理函数
    /
    void regist_akg_func(uint16 id, FUNC_PTR func);
```

id 可以是 0-65535 的任意数，此 id 封装在 MarioTCP 的协议中（见本文最后）。

id 的范围，可以根据业务逻辑来定制，例如 maritcp 通过 protocol.h 中定义的 CMD 结构体来设定：

```
typedef enum _CMD {
CMD_FUNCTION_BASE = 0x6100,
CMD_FUNCTION_LOGIN = 0x6101
} CMD;
```

如果你想为 maritcp 增加一个"say_hello"的服务，可以这么做：

1）在 CMD 中增加：CMD_FUNCTION_SAY_HELLO = 0x602

2) 在 function 中增加函数：

```
sint32 say_hello(CONN c) {
I)通过 CONN 来解析客户端发过来请求的参数
II）将“hello”设定到 c->out_buf
III)bufferevent_write(c->bufev, c->out_buf, hrsp->pkglen);
IV)return 0;
}
```

3)在 mario.c 中增加：regist_akg_func(CMD_FUNCTION_SAY_HELLO, say_hello);

怎么样？自己定制业务逻辑，还是很简单高效吧！

3、启动日志线程 start_log_thread()

MarioTCP 的日志功能封装还不够好，在“go on 1.0.0”页面中继续讨论...

4、启动服务器 start_server((void*) server);

OK，一个可以支持 100 万甚至更多长连接的 TCP 服务器，诞生了！

Go On 1.0.0

第一个发布版本为 0.9.9，尽管用这个包，通过几分钟就可以实现一个定制了你的业务逻辑的、稳定高效的 TCP 服务器，但是 MarioTCP 还有很多有待完善的地方，让我们一起尽快解决如下问题，让 MarioTCP-1.0.0 尽快发布！

1、MarioTCP 协议如何优化

为了使 MarioTCP 足够安全，规定了一个简易的 MarioTCP 协议，经过三次握手连接到 MarioTCP 的 client，接下来发的包要求格式必须是“HEAD+Data”的形式，而 HEAD 结构体定义在 mario_akg.h 中：

```
typedef struct _HEAD {
uint32 stx;
uint16 pkglen;
uint16 akg_id;
} HEAD;
```

pkglen 是整个包的长度，即“HEAD+Data”。

akg_id 及自定义的业务逻辑函数对应的 id，例如“Getting Started”页面中的 CMD_FUNCTION_SAY_HELLO

stx 是你自定义的协议密文，通过 regist_stx(uint32 stx)来注册(见 mario.c)

尽管 MarioTCP 的协议足够简单了，而且协议最开头的密文可以自定义，但是是否可以更简单或者无协议，以最大程度的方便开发使用，需要大家的建议和帮助！

2、日志系统过于死板

MarioTCP 有一套自成系统的日志功能，但是比较晦涩难懂。

接下来再展开...

3、业务逻辑稳定性支持

MarioTCP 对于网络连接和读写，非常高效和稳定。

但是 MarioTCP 的线程池是固定个数的，且是全局唯一初始化的，死掉的线程不可再重启；分配网络任务的 Master 线程不具备监听 worker 的功能，一个线程死掉了、任务却还会一直分配过来，造成服务堆积且不处理。如果业务逻辑如果非常复杂和低效，就会出现这个问题。

在大型线上项目中，用到 MarioTCP 的地方，都会通过业务逻辑模块的监听、告警及程序自动处理来避免上述问题。由于时间问题还没有把此功能抽象到 MarioTCP 中。

这件事情，近期我会抓紧处理。也希望有朋友建议和帮助！！

Why Supper

一、为什么超级高效

1、网络服务用到的所有结构体和内存都是启动程序时初始化的，无销毁，无回收。

无销毁好理解，不解释。

无回收，是指所有内存单元拿来即用，用完及可，不用做 reset 操作。

2、一个 master 线程进行 accept

经过测试发现多进程或线程进行 accept 和一个进程或线程 accept，在极限压力下区别不大。

一个 master 比多个 master 好在不用再通过锁来解决同步问题。

3、master 与 worker 时单一生产者消费者模式，完全无锁通信

不光 accept 无锁，分配 connection、后续的 conncetion 处理都是无锁的。

甚至业务逻辑（见示例 maritcp 的统计在线数功能）、MarioTCP 的日志系统（这也是日志系统抽象不够的一个原因，之前的设计太依赖于整体架构了）都是无锁处理的！

4、一个 worker 一套 libevent 环境

libevent 处理 10 万长连接的网络读写事件，其性能达到最大化了。

每个 worker 都独立一套 libevent，这个结构经过测试，发现开销很小、性能很高。

二、单机百万长连接、四万 cps（连接每秒）如何做测试得来

1、设置系统最大文件数为 unlimited

2、设置系统的 tcp 内存内核参数到 256M 以上

3、设置系统的 ip 到 15 个，那么可服务的长连接数理论上最少 15*(65535-1024)个

4、用 epoll 或 libevent 开一个可同时连接 5w 的客户端程序；程序还要实现每秒随机挑选 1000 个连接断掉，并再新创建 1000 个连接。另外在随机挑选几千连接发包。

同时再多台机器上开启 20 个客户端，那么就是 100w 长连接，每秒 2w 个连接断掉、2w 个新连接加入进来，并且有若干包发过来。

5、设置服务端可重用 SYN_WAIT 的连接；客户端断连接的方式是主动断掉（防止客户端程序端口堆积）

总之很折腾的一个测试，前前后后大约 2 个多月才测试完毕。

以上内容凭记忆写的，怕有错误或疏漏，回头为了公布测试代码和测试结果给大家，会再次开发、测试并调整补过上述内容。

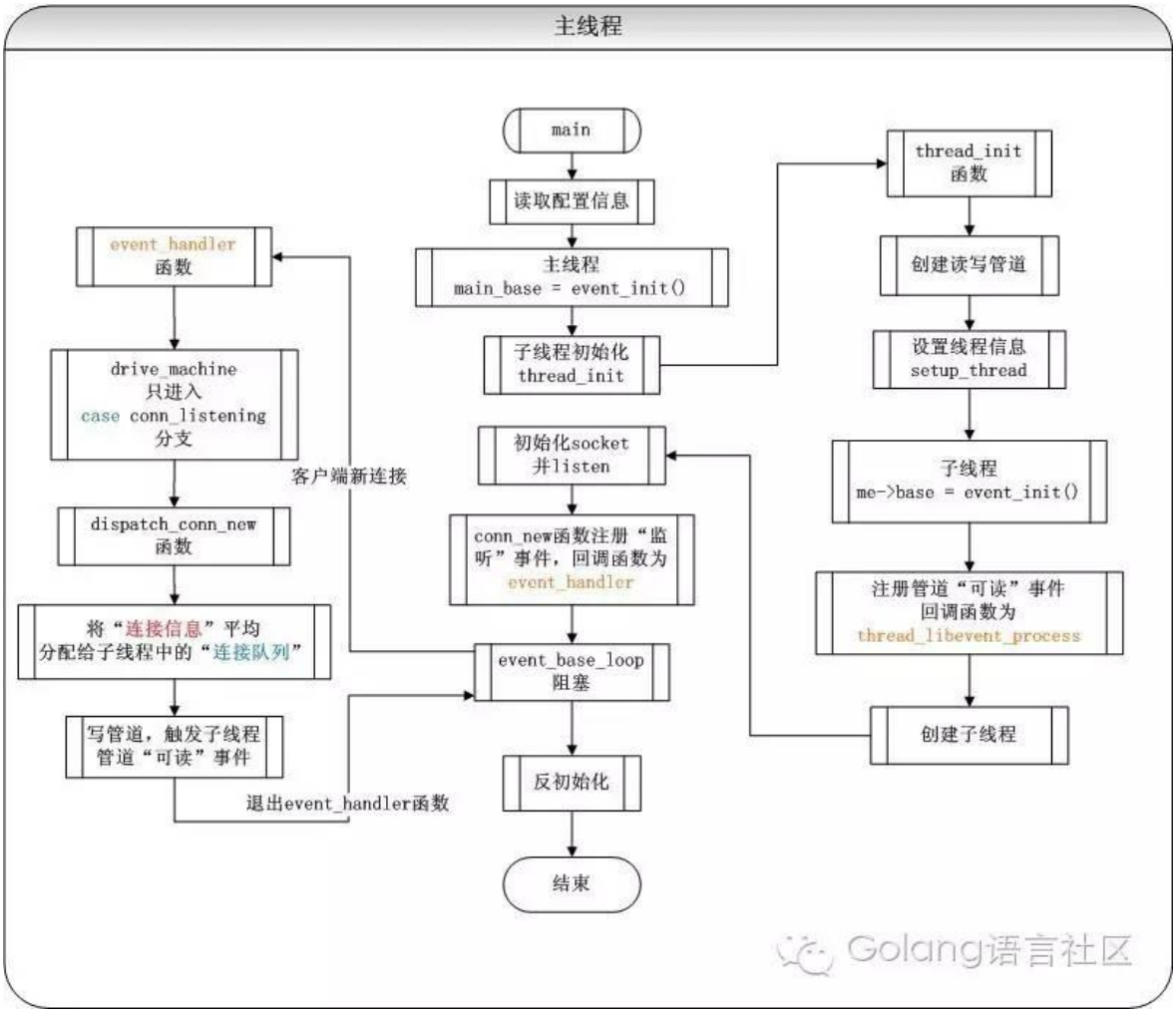
本文讲述的 TCP 服务器是模仿 memcache 中的 TCP 网络处理框架，其中是基于 libevent 网络库的。

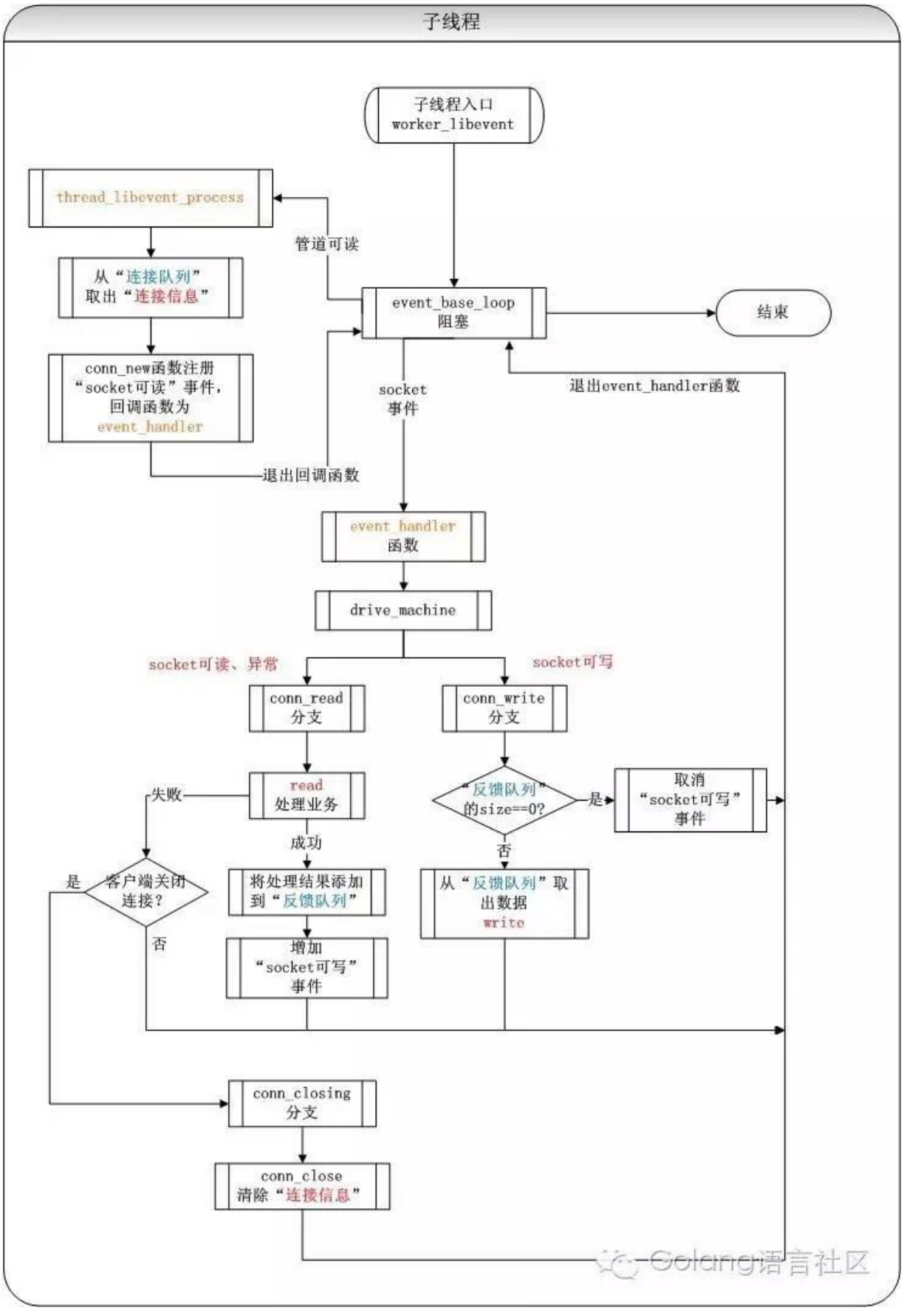
主线程只处理监听客户端的连接请求，并将请求平均分配给子线程。

子线程处理与客户端的连接以及相关业务。

每个子线程有一个“连接”队列。每个“连接”有一个“反馈”队列。

先上个流程图，要上班了，以后再解释。代码以后再上…


主线程
main
读取配置信息
主线程
main_base = event_init()
子线程初始化
thread_init
thread_init
函数
创建读写管道
设置线程信息
setup_thread
子线程
me->base = event_init()
注册管道“可读”事件
回调函数为
thread_libevent_process
创建子线程
初始化socket
并listen
conn_new函数注册“监听”事件，回调函数为
event_handler
event_base_loop
阻塞
反初始化
结束
event_handler
函数
drive_machine
只进入
case conn_listening
分支
客户端新连接
dispatch_conn_new
函数
将“连接信息”平均
分配给子线程中的“连接队列”
写管道，触发子线程
管道“可读”事件
退出event_handler函数
Golang语言社区


子线程
子线程入口
worker_libevent
thread_libevent_process
管道可读
从“连接队列”
取出“连接信息”
event_base_loop
阻塞
结束
conn_new函数注册
“socket可读”事件，
回调函数为
event_handler
socket
事件
退出event_handler函数
退出回调函数
event_handler
函数
drive_machine
socket可读、异常
socket可写
conn_read
分支
conn_write
分支
read
处理业务
失败
成功
“反馈队列”
的size==0?
是
取消
“socket可写”
事件
否
是
客户端关闭
连接？
否
将处理结果添加
到“反馈队列”
从“反馈队列”取
出数据
write
增加
“socket可写”
事件
conn_closing
分支
conn_close
清除“连接信息”
Golang语言社区