

协程的实现与原理

前言

协程这个概念很久了，好多程序员是实现过这个组件的，网上关于协程的文章，博客，论坛都是汗牛充栋，在知乎，github 上面也有很多大牛写了关于协程的心得体会。突发奇想，我也来实现一个这样的组件，并测试了一下性能。借鉴了很多大牛的思想，阅读了很多大牛的代码。于是把整个思考过程写下来。实现代码

<https://github.com/wangbojing/NtyCo>

代码简单易读，如果在你的项目中，NtyCo 能够为你解决些许工程问题，那就荣幸之至。

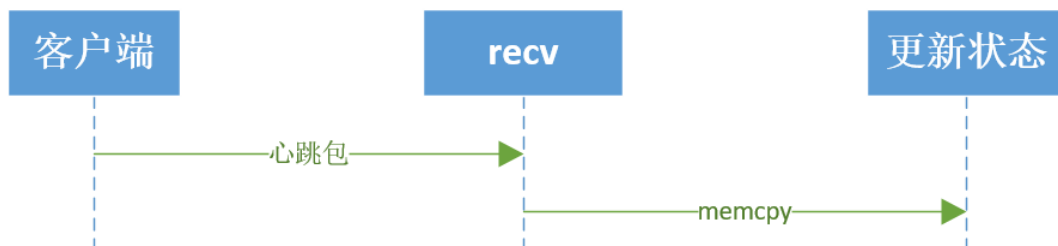
本系列文章的设计思路，是在每一个章的最前面以问题提出，每章节的学习目的。大家能够带着每章的问题来读每章节的内容，方便读者能够方便的进入每章节的思考。读者读完以后加上案例代码阅读，编译，运行，能够对神秘的协程有一个全新的理解。能够运用到工程代码，帮助你更加方便高效的完成工程工作。

本系列文章仅代表本人观点，若不严谨的地方，欢迎抛转。

第一章 协程的起源

问题：协程存在的原因？协程能够解决哪些问题？

在我们现在 CS，BS 开发模式下，服务器的吞吐量是一个很重要的参数。其实吞吐量是 IO 处理时间加上业务处理。为了简单起见，比如，客户端与服务器之间是长连接的，客户端定期给服务器发送心跳包数据。客户端发送一次心跳包到服务器，服务器更新该新客户端状态的。心跳包发送的过程，业务处理时长等于 IO 读取（RECV 系统调用）加上业务处理（更新客户状态）。吞吐量等于 1s 业务处理次数。



业务处理（更新客户端状态）时间，业务不一样的，处理时间不一样，我们就不做讨论。

那如何提升 recv 的性能。若只有一个客户端，recv 的性能也没有必要提升，也不能提升。若有百万计的客户端长连接的情况，我们该如何提升。以

Linux 为例，在这里需要介绍一个“网红”就是 epoll。服务器使用 epoll 管理百万计的客户端长连接，代码框架如下：

```
while (1) {
    int nready = epoll_wait(epfd, events, EVENT_SIZE, -1);

    for (i = 0; i < nready; i++) {

        int sockfd = events[i].data.fd;
        if (sockfd == listenfd) {
            int connfd = accept(listenfd, xxx, xxx);

            setnonblock(connfd);

            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = connfd;
            epoll_ctl(epfd, EPOLL_CTL_ADD, connfd, &ev);

        } else {
            handle(sockfd);
        }
    }
}
```

对于响应式服务器，所有的客户端的操作驱动都是来源于这个大循环。来源于 epoll_wait 的反馈结果。

对于服务器处理百万计的 IO。Handle(sockfd)实现方式有两种。

第一种，handle(sockfd)函数内部对 sockfd 进行读写动作。代码如下

```
int handle(int sockfd) {

    recv(sockfd, rbuffer, length, 0);

    parser_proto(rbuffer, length);

    send(sockfd, sbuffer, length, 0);

}
```

handle 的 io 操作（send, recv）与 epoll_wait 是在同一个处理流程里面的。这就是 IO 同步操作。

优点：

1. sockfd 管理方便。
2. 操作逻辑清晰。

缺点：

1. 服务器程序依赖 `epoll_wait` 的循环响应速度慢。
2. 程序性能差

第二种，`handle(sockfd)` 函数内部将 `sockfd` 的操作，push 到线程池中，代码如下：

```
int thread_cb(int sockfd) {
    // 此函数是在线程池创建的线程中运行。
    // 与 handle 不在一个线程上下文中运行
    recv(sockfd, rbuffer, length, 0);
    parser_proto(rbuffer, length);
    send(sockfd, sbuffer, length, 0);
}

int handle(int sockfd) {
    // 此函数在主线程 main_thread 中运行
    // 在此处之前，确保线程池已经启动。
    push_thread(sockfd, thread_cb); // 将 sockfd 放到其他线程中运行。
}
```

`Handle` 函数是将 `sockfd` 处理方式放到另一个已经其他的线程中运行，如此做法，将 `io` 操作（`recv`, `send`）与 `epoll_wait` 不在一个处理流程里面，使得 `io` 操作（`recv`, `send`）与 `epoll_wait` 实现解耦。这就叫做 `I/O` 异步操作。

优点：

1. 子模块好规划。
2. 程序性能高。

缺点：

正因为子模块好规划，使得模块之间的 `sockfd` 的管理异常麻烦。每一个子线程都需要管理好 `sockfd`，避免在 `I/O` 操作的时候，`sockfd` 出现关闭或其他异常。

上文有提到 `I/O` 同步操作，程序响应慢，`I/O` 异步操作，程序响应快。

下面来对比一下 `I/O` 同步操作与 `I/O` 异步操作。

代码如下：

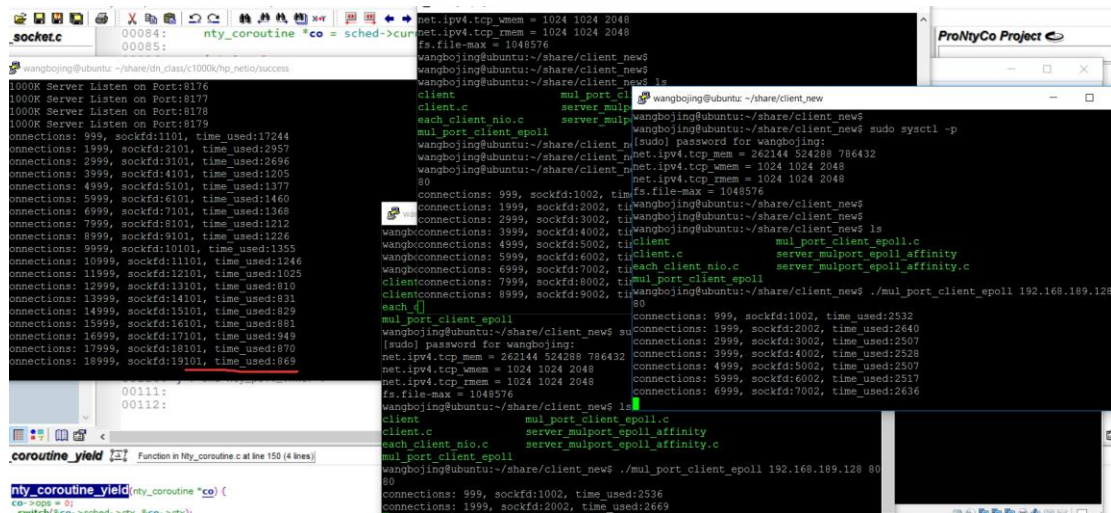
https://github.com/wangbojing/cl000k_test/blob/master/server_mulport_epoll.c

在这份代码的 486 行，`#if 1`，打开的时候，为 `I/O` 异步操作。关闭的时候，为 `I/O` 同步操作。

```
486 #if 1
487
488         if (nRun) {
489             printf(" New Data is Comming\n");
490             client_data_process(clientfd);
491         } else {
492
493             client_t *rClient = (client_t*)malloc(sizeof(client_t));
494             memset(rClient, 0, sizeof(client_t));
495             rClient->fd = clientfd;
496
497             job_t *job = malloc(sizeof(job_t));
498             job->job_function = client_job;
499             job->user_data = rClient;
500             workqueue_add_job(&workqueue, job);
501
502         }
503 #else
504         client_data_process(clientfd);
505 #endif
```

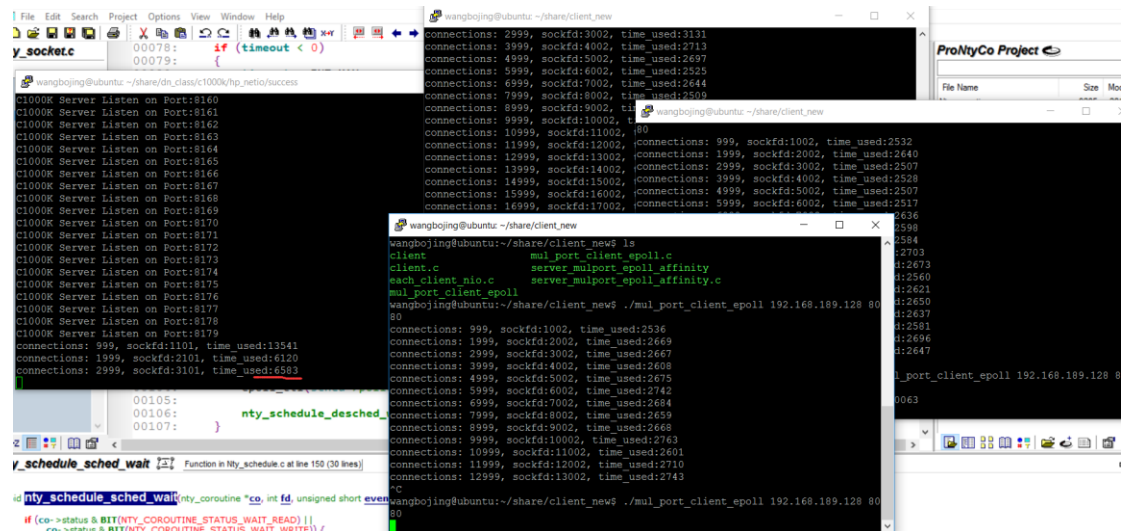
接下来把我测试接入量的结果粘贴出来。

I0 异步操作，每 1000 个连接接入的服务器响应时间（900ms 左右）。



```
00084:
00085:
wangbojing@ubuntu: ~/share/dn_class/c1000k/hp_netio/success
1000K Server Listen on Port:8176
1000K Server Listen on Port:8177
1000K Server Listen on Port:8178
connections: 999, sockfd:1101, time used:17244
connections: 1999, sockfd:2101, time used:2957
connections: 2999, sockfd:3101, time used:2696
connections: 3999, sockfd:4101, time used:1205
connections: 4999, sockfd:5101, time used:1377
connections: 5999, sockfd:6101, time used:1460
connections: 6999, sockfd:7101, time used:1368
connections: 7999, sockfd:8101, time used:1212
connections: 8999, sockfd:9101, time used:1226
connections: 9999, sockfd:10101, time used:1355
connections: 10999, sockfd:11101, time used:1246
connections: 11999, sockfd:12101, time used:1025
connections: 12999, sockfd:13101, time used:810
connections: 13999, sockfd:14101, time used:831
connections: 14999, sockfd:15101, time used:829
connections: 15999, sockfd:16101, time used:881
connections: 16999, sockfd:17101, time used:949
connections: 17999, sockfd:18101, time used:870
connections: 18999, sockfd:19101, time used:869
00111:
00112:
wangbojing@ubuntu: ~/share/client_new$ ls
client.c      mul_port_client_epoll.c
client.h      server_mulport_epoll.c
each_client_nio.c  server_mulport_epoll_affinity.c
mul_port_client_epoll.c
wangbojing@ubuntu: ~/share/client_new$ ./mul_port_client_epoll 192.168.189.128 80
connections: 999, sockfd:1002, time used:2532
connections: 1999, sockfd:2002, time used:2640
connections: 2999, sockfd:3002, time used:2507
connections: 3999, sockfd:4002, time used:2528
connections: 4999, sockfd:5002, time used:2507
connections: 5999, sockfd:6002, time used:2517
connections: 6999, sockfd:7002, time used:2636
wangbojing@ubuntu: ~/share/client_new$ ls
client.c      mul_port_client_epoll.c
client.h      server_mulport_epoll.c
each_client_nio.c  server_mulport_epoll_affinity.c
mul_port_client_epoll.c
wangbojing@ubuntu: ~/share/client_new$ ./mul_port_client_epoll 192.168.189.128 80
connections: 999, sockfd:1002, time used:2536
connections: 1999, sockfd:2002, time used:2669
```

I0 同步操作，每 1000 个连接接入的服务器响应时间（6500ms 左右）。



I/O 异步操作与 I/O 同步操作

对比项	I/O 同步操作	I/O 异步操作
Socketfd 管理	管理方便	多个线程共同管理
代码逻辑	程序整体逻辑清晰	子模块逻辑清晰
程序性能	响应时间长，性能差	响应时间短，性能好

有没有一种方式，有异步性能，同步的代码逻辑。来方便编程人员对 I/O 操作的组件呢？有，采用一种轻量级的协程来实现。在每次 send 或者 recv 之前进行切换，再由调度器来处理 epoll_wait 的流程。

就是采用了基于这样的思考，写了 NtyCo，实现了一个 I/O 异步操作与协程结合的组件。<https://github.com/wangbojing/NtyCo>,

第二章 协程的案例

问题：协程如何使用？与线程使用有何区别？

在做网络 IO 编程的时候，有一个非常理想的情况，就是每次 accept 返回的时候，就为新来的客户端分配一个线程，这样一个客户端对应一个线程。就不会有多个线程共用一个 sockfd。每请求每线程的方式，并且代码逻辑非常易读。但是这只是理想，线程创建代价，调度代价就呵呵了。

先来看一下每请求每线程的代码如下：

```
while(1) {
    socklen_t len = sizeof(struct sockaddr_in);
    int clientfd = accept(sockfd, (struct sockaddr*)&remote, &len);

    pthread_t thread_id;
    pthread_create(&thread_id, NULL, client_cb, &clientfd);
}
```

这样的做法，写完放到生产环境下面，如果你的老板不打死你，你來找我。我来帮你老板，为民除害。

如果我们有协程，我们就可以这样实现。参考代码如下：

https://github.com/wangbojing/NtyCo/blob/master/nty_server_test.c

```
while (1) {
    socklen_t len = sizeof(struct sockaddr_in);
    int cli_fd = nty_accept(fd, (struct sockaddr*)&remote, &len);

    nty_coroutine *read_co;
    nty_coroutine_create(&read_co, server_reader, &cli_fd);
}
```

这样的代码是完全可以放在生产环境下面的。如果你的老板要打死你，你來找我，我帮你把你老板打死，为民除害。

线程的 API 思维来使用协程，函数调用的性能来测试协程。

NtyCo 封装出来了若干接口，一类是协程本身的，二类是 posix 的异步封装协程 API：while

1. 协程创建

```
int nty_coroutine_create(nty_coroutine **new_co, proc_coroutine func,
void *arg)
```

2. 协程调度器的运行

```
void nty_schedule_run(void)
```

POSIX 异步封装 API:

```
int nty_socket(int domain, int type, int protocol)
int nty_accept(int fd, struct sockaddr *addr, socklen_t *len)
int nty_recv(int fd, void *buf, int length)
int nty_send(int fd, const void *buf, int length)
int nty_close(int fd)
```

接口格式与 POSIX 标准的函数定义一致。

第三章 协程的实现之工作流程

问题：协程内部是如何工作呢？

先来看一下协程服务器案例的代码， 代码参考：

https://github.com/wangbojing/NtyCo/blob/master/nty_server_test.c

分别讨论三个协程的比较晦涩的工作流程。第一个协程的创建；第二个 IO 异步操作；第三个协程子过程回调

3.1 创建协程

当我们需要异步调用的时候，我们会创建一个协程。比如 `accept` 返回一个新的 `sockfd`，创建一个客户端处理的子过程。再比如需要监听多个端口的时候，创建一个 `server` 的子过程，这样多个端口同时工作的，是符合微服务的架构的。

创建协程的时候，进行了如何的工作？创建 API 如下：

```
int nty_coroutine_create(nty_coroutine **new_co, proc_coroutine func,
void *arg)
```

参数 1: `nty_coroutine **new_co`，需要传入空的协程的对象，这个对象是由内部创建的，并且在函数返回的时候，会返回一个内部创建的协程对象。

参数 2: `proc_coroutine func`，协程的子过程。当协程被调度的时候，就会执行该函数。

参数 3: `void *arg`，需要传入到新协程中的参数。

协程不存在亲属关系，都是一致的调度关系，接受调度器的调度。调用 `create` API 就会创建一个新协程，新协程就会加入到调度器的就绪队列中。

创建的协程具体步骤会在《协程的实现之原语操作》来描述。

3.2 实现 IO 异步操作

大部分的朋友会关心 IO 异步操作如何实现，在 `send` 与 `recv` 调用的时候，如何实现异步操作的。

先来看一下一段代码：

```
while (1) {
    int nready = epoll_wait(epfd, events, EVENT_SIZE, -1);

    for (i = 0; i < nready; i++) {

        int sockfd = events[i].data.fd;
        if (sockfd == listenfd) {
            int connfd = accept(listenfd, xxx, xxx);

            setnonblock(connfd);

            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = connfd;
            epoll_ctl(epfd, EPOLL_CTL_ADD, connfd, &ev);

        } else {

            epoll_ctl(epfd, EPOLL_CTL_DEL, sockfd, NULL);
            recv(sockfd, buffer, length, 0);

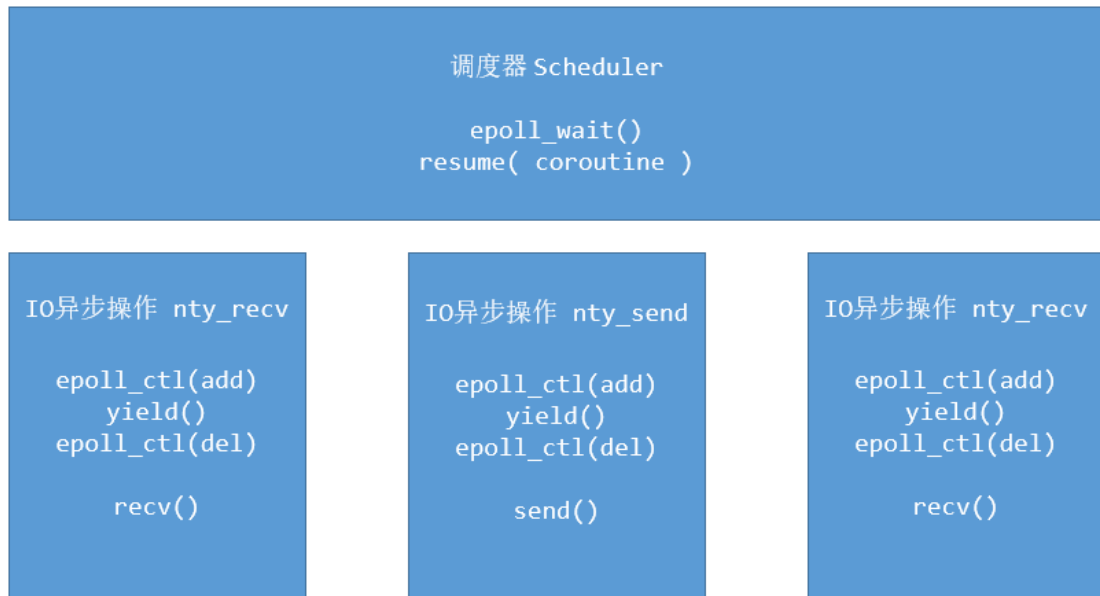
            //parser_proto(buffer, length);

            send(sockfd, buffer, length, 0);
            epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, NULL);
        }
    }
}
```

在进行 IO 操作（`recv`，`send`）之前，先执行了 `epoll_ctl` 的 `del` 操作，将相应的 `sockfd` 从 `epfd` 中删除掉，在执行完 IO 操作（`recv`，`send`）再进行 `epoll_ctl` 的 `add` 的动作。这段代码看起来似乎好像没有什么作用。

如果是在多个上下文中，这样的做法就很有意义了。能够保证 `sockfd` 只在一个上下文中能够操作 IO 的。不会出现在多个上下文同时对一个 IO 进行操作的。协程的 IO 异步操作正式是采用此模式进行的。

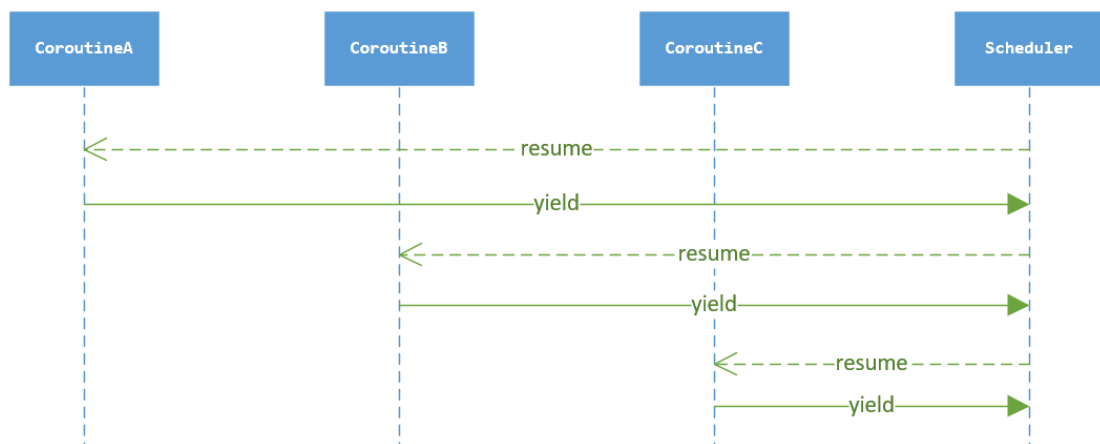
把单一协程的工作与调度器的工作的划分清楚，先引入两个原语操作 `resume`，`yield` 会在《协程的实现之原语操作》来讲解协程所有原语操作的实现，`yield` 就是让出运行，`resume` 就是恢复运行。调度器与协程的上下文切换如下图所示



在协程的上下文 IO 异步操作（`nty_recv`，`nty_send`）函数，步骤如下：

1. 将 `sockfd` 添加到 `epoll` 管理中。
2. 进行上下文环境切换，由协程上下文 `yield` 到调度器的上下文。
3. 调度器获取下一个协程上下文。Resume 新的协程

IO 异步操作的上下文切换的时序图如下：



3.3 回调协程的子过程

在 `create` 协程后，何时回调子过程？何种方式回调子过程？

首先来回顾一下 `x86_64` 寄存器的相关知识。汇编与寄存器相关知识还会在《协程的实现之切换》继续深入探讨的。`x86_64` 的寄存器有 16 个 64 位寄存器，分别是：`%rax`，`%rbx`，`%rcx`，`%esi`，`%edi`，`%rbp`，`%rsp`，`%r8`，`%r9`，`%r10`，`%r11`，`%r12`，`%r13`，`%r14`，`%r15`。

`%rax` 作为函数返回值使用的。

`%rsp` 栈指针寄存器，指向栈顶

`%rdi`，`%rsi`，`%rdx`，`%rcx`，`%r8`，`%r9` 用作函数参数，依次对应第 1 参数，第 2 参数。。。

`%rbx`，`%rbp`，`%r12`，`%r13`，`%r14`，`%r15` 用作数据存储，遵循调用者使用规则，换句话

说，就是随使用。调用子函数之前要备份它，以防它被修改
%r10, %r11 用作数据存储，就是使用前要先保存原值

以 NtyCo 的实现为例，来分析这个过程。CPU 有一个非常重要的寄存器叫做 EIP，用来存储 CPU 运行下一条指令的地址。我们可以把回调函数的地址存储到 EIP 中，将相应的参数存储到相应的参数寄存器中。实现子过程调用的逻辑代码如下：

```
void _exec(nty_coroutine *co) {
    co->func(co->arg); //子过程的回调函数
}

void nty_coroutine_init(nty_coroutine *co) {
    //ctx 就是协程的上下文
    co->ctx.edi = (void*)co; //设置参数
    co->ctx.eip = (void*)_exec; //设置回调函数入口
    //当实现上下文切换的时候，就会执行入口函数_exec，_exec 调用子过程 func
}
```

第四章 协程的实现之原语操作

问题：协程的内部原语操作有哪些？分别如何实现的？

协程的核心原语操作：create, resume, yield。协程的原语操作有 create 怎么没有 exit？以 NtyCo 为例，协程一旦创建就不能有用户自己销毁，必须得以子过程执行结束，就会自动销毁协程的上下文数据。以 _exec 执行入口函数返回而销毁协程的上下文与相关信息。co->func(co->arg) 是子过程，若用户需要长久运行协程，就必须要在 func 函数里面写入循环等操作。所以 NtyCo 里面没有实现 exit 的原语操作。

create：创建一个协程。

1. 调度器是否存在，不存在也创建。调度器作为全局的单例。将调度器的实例存储在线程的私有空间 pthread_setspecific。
2. 分配一个 coroutine 的内存空间，分别设置 coroutine 的数据项，栈空间，栈大小，初始状态，创建时间，子过程回调函数，子过程的调用参数。
3. 将新分配协程添加到就绪队列 ready_queue 中

实现代码如下：

```
int nty_coroutine_create(nty_coroutine **new_co, proc_coroutine func,
void *arg) {

    assert(pthread_once(&sched_key_once,
nty_coroutine_sched_key_creator) == 0);
    nty_schedule *sched = nty_coroutine_get_sched();
```

```
if (sched == NULL) {
    nty_schedule_create(0);

    sched = nty_coroutine_get_sched();
    if (sched == NULL) {
        printf("Failed to create scheduler\n");
        return -1;
    }
}

nty_coroutine *co = calloc(1, sizeof(nty_coroutine));
if (co == NULL) {
    printf("Failed to allocate memory for new coroutine\n");
    return -2;
}

//
int ret = posix_memalign(&co->stack, getpagesize(),
sched->stack_size);
if (ret) {
    printf("Failed to allocate stack for new coroutine\n");
    free(co);
    return -3;
}

co->sched = sched;
co->stack_size = sched->stack_size;
co->status = BIT(NTY_COROUTINE_STATUS_NEW); //
co->id = sched->spawned_coroutines ++;
co->func = func;

co->fd = -1;
co->events = 0;

co->arg = arg;
co->birth = nty_coroutine_usec_now();
*new_co = co;

TAILQ_INSERT_TAIL(&co->sched->ready, co, ready_next);

return 0;
}
```

yield: 让出 CPU。

```
void nty_coroutine_yield(nty_coroutine *co)
```

参数: 当前运行的协程实例

调用后该函数不会立即返回, 而是切换到最近执行 resume 的上下文。该函数返回是在执行 resume 的时候, 会有调度器统一选择 resume 的, 然后再次调用 yield 的。resume 与 yield 是两个可逆过程的原子操作。

resume: 恢复协程的运行权

```
int nty_coroutine_resume(nty_coroutine *co)
```

参数: 需要恢复运行的协程实例

调用后该函数也不会立即返回, 而是切换到运行协程实例的 yield 的位置。返回是在等协程相应事务处理完成后, 主动 yield 会返回到 resume 的地方。

第五章 协程的实现之切换

问题: 协程的上下文如何切换? 切换代码如何实现?

首先来回顾一下 x86_64 寄存器的相关知识。x86_64 的寄存器有 16 个 64 位寄存器, 分别是: %rax, %rbx, %rcx, %esi, %edi, %rbp, %rsp, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15。

%rax 作为函数返回值使用的。

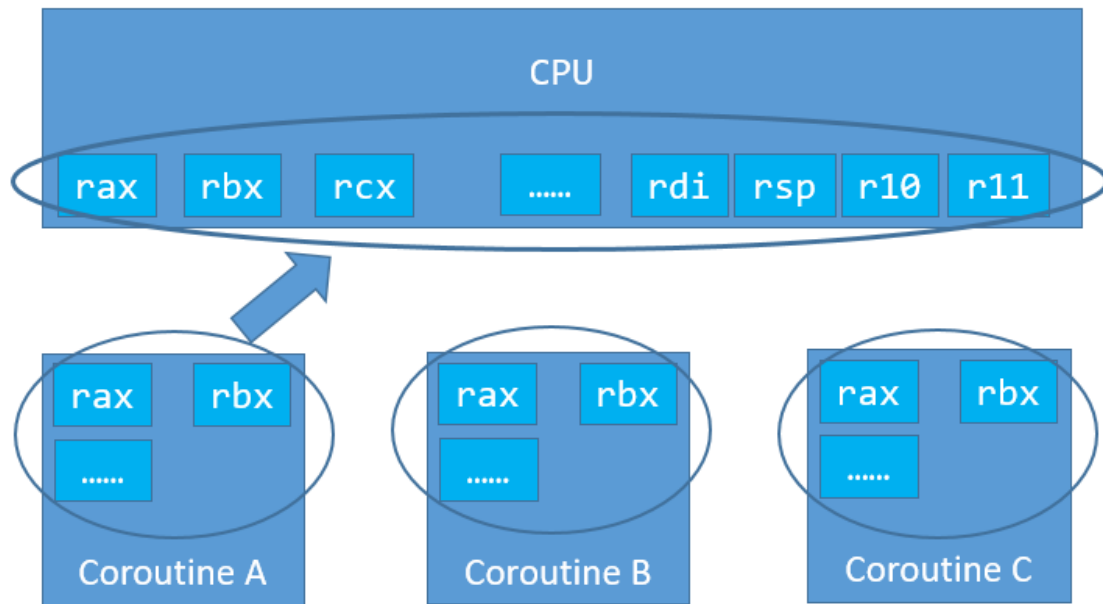
%rsp 栈指针寄存器, 指向栈顶

%rdi, %rsi, %rdx, %rcx, %r8, %r9 用作函数参数, 依次对应第 1 参数, 第 2 参数。。。

%rbx, %rbp, %r12, %r13, %r14, %r15 用作数据存储, 遵循调用者使用规则, 换句话说, 就是随使用。调用子函数之前要备份它, 以防它被修改

%r10, %r11 用作数据存储, 就是使用前要先保存原值。

上下文切换, 就是将 CPU 的寄存器暂时保存, 再将即将运行的协程的上下文寄存器, 分别 mov 到相对应的寄存器上。此时上下文完成切换。如下图所示:



切换_switch 函数定义:

```
int _switch(nty_cpu_ctx *new_ctx, nty_cpu_ctx *cur_ctx);
```

参数 1: 即将运行协程的上下文, 寄存器列表

参数 2: 正在运行协程的上下文, 寄存器列表

我们 nty_cpu_ctx 结构体的定义, 为了兼容 x86, 结构体项命令采用的是 x86 的寄存器名字命名。

```
typedef struct _nty_cpu_ctx {  
    void *esp; //  
    void *ebp;  
    void *eip;  
    void *edi;  
    void *esi;  
    void *ebx;  
    void *r1;  
    void *r2;  
    void *r3;  
    void *r4;  
    void *r5;  
} nty_cpu_ctx;
```

_switch 返回后, 执行即将运行协程的上下文。是实现上下文的切换

_switch 的实现代码:

```
0: __asm__ (  
1: "    .text                \n"  
2: "    .p2align 4,,15      \n"  
3: ".globl _switch          \n"  
4: ".globl __switch        \n"
```

```
5: "_switch:                                \n"
6: "__switch:                              \n"
7: "    movq %rsp, 0(%rsi)      # save stack_pointer  \n"
8: "    movq %rbp, 8(%rsi)     # save frame_pointer   \n"
9: "    movq (%rsp), %rax      # save insn_pointer    \n"
10: "    movq %rax, 16(%rsi)    \n"
11: "    movq %rbx, 24(%rsi)    # save rbx,r12-r15     \n"
12: "    movq %r12, 32(%rsi)    \n"
13: "    movq %r13, 40(%rsi)    \n"
14: "    movq %r14, 48(%rsi)    \n"
15: "    movq %r15, 56(%rsi)    \n"
16: "    movq 56(%rdi), %r15    \n"
17: "    movq 48(%rdi), %r14    \n"
18: "    movq 40(%rdi), %r13    # restore rbx,r12-r15  \n"
19: "    movq 32(%rdi), %r12    \n"
20: "    movq 24(%rdi), %rbx    \n"
21: "    movq 8(%rdi), %rbp     # restore frame_pointer \n"
22: "    movq 0(%rdi), %rsp     # restore stack_pointer \n"
23: "    movq 16(%rdi), %rax    # restore insn_pointer  \n"
24: "    movq %rax, (%rsp)      \n"
25: "    ret                    \n"
26: );
```

按照 x86_64 的寄存器定义, %rdi 保存第一个参数的值, 即 new_ctx 的值, %rsi 保存第二个参数的值, 即保存 cur_ctx 的值。X86_64 每个寄存器是 64bit, 8byte。

Movq %rsp, 0(%rsi) 保存在栈指针到 cur_ctx 实例的 rsp 项

Movq %rbp, 8(%rsi)

Movq (%rsp), %rax #将栈顶地址里面的值存储到 rax 寄存器中。Ret 后出栈, 执行栈顶

Movq %rbp, 8(%rsi) #后续的指令都是用来保存 CPU 的寄存器到 new_ctx 的每一项中

Movq 8(%rdi), %rbp #将 new_ctx 的值

Movq 16(%rdi), %rax #将指令指针 rip 的值存储到 rax 中

Movq %rax, (%rsp) # 将存储的 rip 值的 rax 寄存器赋值给栈指针的地址的值。

Ret # 出栈, 回到栈指针, 执行 rip 指向的指令。

上下文环境的切换完成。

第六章 协程的实现之定义

问题: 协程如何定义? 调度器如何定义?

先来一道设计题:

设计一个协程的运行体 R 与运行体调度器 S 的结构体

1. 运行体 R: 包含运行状态 {就绪, 睡眠, 等待}, 运行体回调函数, 回调参数, 栈指针, 栈大小, 当前运行体
2. 调度器 S: 包含执行集合 {就绪, 睡眠, 等待}

这道设计题拆分两个问题, 一个运行体如何高效地在多种状态集合更换。调度器与运行体的功能界限。

6.1 运行体如何高效地在多种状态集合更换

新创建的协程, 创建完成后, 加入到就绪集合, 等待调度器的调度; 协程在运行完成后, 进行 IO 操作, 此时 IO 并未准备好, 进入等待状态集合; IO 准备就绪, 协程开始运行, 后续进行 `sleep` 操作, 此时进入到睡眠状态集合。

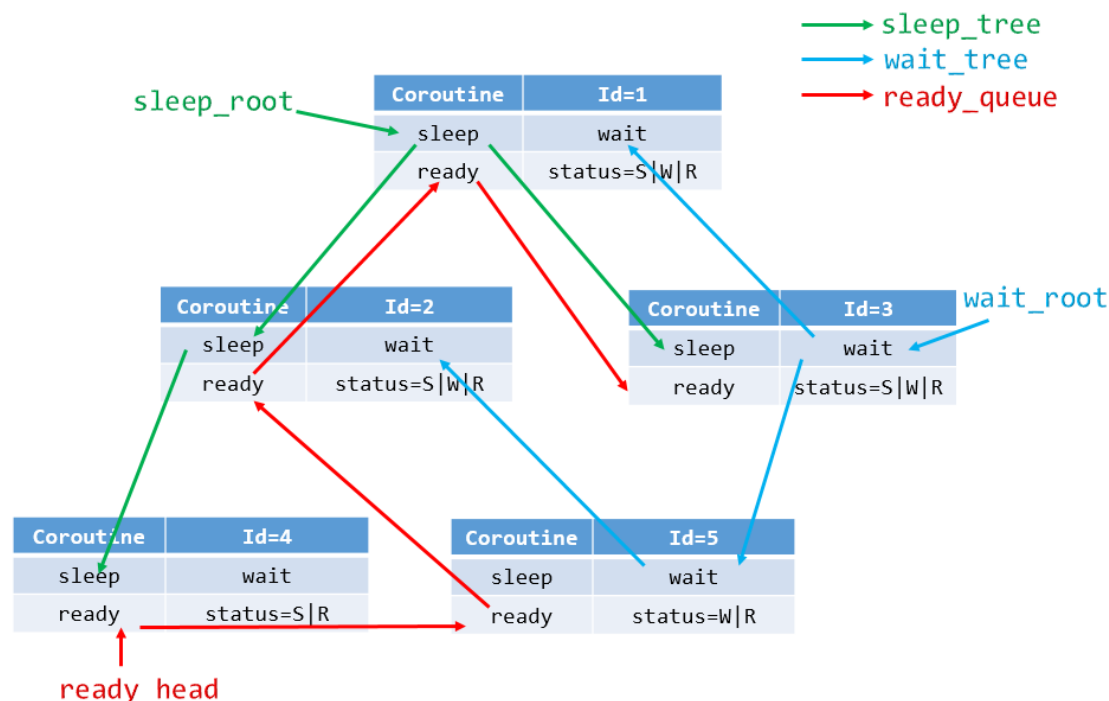
就绪(`ready`), 睡眠(`sleep`), 等待(`wait`)集合该采用如何数据结构来存储?

就绪(`ready`)集合并不没有设置优先级的选型, 所有在协程优先级一致, 所以可以使用队列来存储就绪的协程, 简称为就绪队列 (`ready_queue`)。

睡眠(`sleep`)集合需要按照睡眠时长进行排序, 采用红黑树来存储, 简称睡眠树(`sleep_tree`)红黑树在工程实用为<key, value>, key 为睡眠时长, value 为对应的协程结点。

等待(`wait`)集合, 其功能是在等待 IO 准备就绪, 等待 IO 也是有时长的, 所以等待(`wait`)集合采用红黑树的来存储, 简称等待树(`wait_tree`), 此处借鉴 `nginx` 的设计。

数据结构如下图所示:



`Coroutine` 就是协程的相应属性, `status` 表示协程的运行状态。sleep 与 wait 两颗红黑树, ready 使用的队列, 比如某协程调用 `sleep` 函数, 加入睡眠树(`sleep_tree`), `status |= S` 即可。比如某协程在等待树(`wait_tree`)中, 而 IO 准备就绪放入 ready 队列中, 只需要移出等待树(`wait_tree`), 状

态更改 `status &= ~W` 即可。有一个前提条件就是不管何种运行状态的协程，都在就绪队列中，只是同时包含有其他的运行状态。

6.2 调度器与协程的功能界限

每一协程都需要使用的而且可能会不同属性的，就是协程属性。每一协程都需要的而且数据一致的，就是调度器的属性。比如栈大小的数值，每个协程都一样的后不做更改可以作为调度器的属性，如果每个协程大小不一致，则可以作为协程的属性。

用来管理所有协程的属性，作为调度器的属性。比如 `epoll` 用来管理每一个协程对应的 IO，是需要作为调度器属性。

按照前面几章的描述，定义一个协程结构体需要多少域，我们描述了每一个协程有自己的上下文环境，需要保存 CPU 的寄存器 `ctx`；需要有子过程的回调函数 `func`；需要有子过程回调函数的参数 `arg`；需要定义自己的栈空间 `stack`；需要定义自己栈空间的大小 `stack_size`；需要定义协程的创建时间 `birth`；需要定义协程当前的运行状态 `status`；需要定义当前运行状态的结点（`ready_next`, `wait_node`, `sleep_node`）；需要定义协程 id；需要定义调度器的全局对象 `sched`。

协程的核心结构体如下：

```
typedef struct _nty_coroutine {

    nty_cpu_ctx ctx;
    proc_coroutine func;
    void *arg;
    size_t stack_size;

    nty_coroutine_status status;
    nty_schedule *sched;

    uint64_t birth;
    uint64_t id;

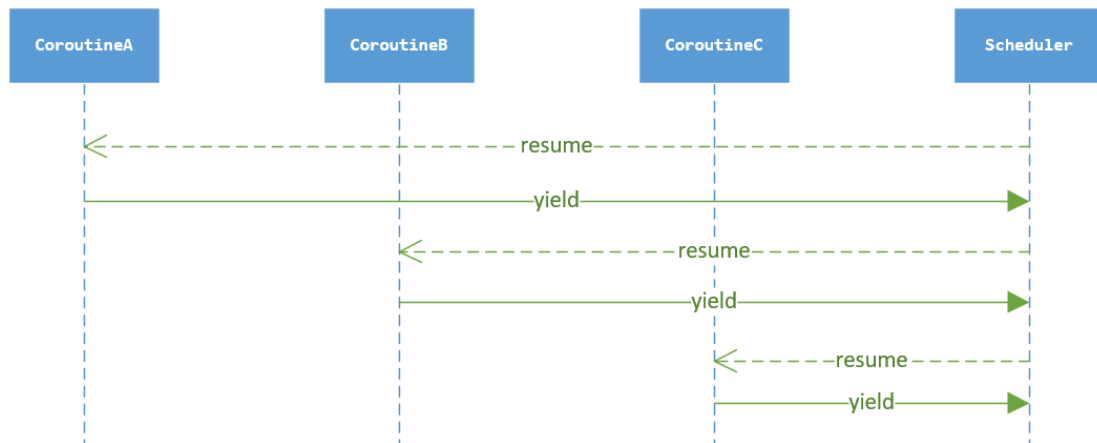
    void *stack;

    RB_ENTRY(_nty_coroutine) sleep_node;
    RB_ENTRY(_nty_coroutine) wait_node;

    TAILQ_ENTRY(_nty_coroutine) ready_next;
    TAILQ_ENTRY(_nty_coroutine) defer_next;

} nty_coroutine;
```

调度器是管理所有协程运行的组件，协程与调度器的运行关系。



调度器的属性，需要有保存 CPU 的寄存器上下文 `ctx`，可以从协程运行状态 `yield` 到调度器运行的。从协程到调度器用 `yield`，从调度器到协程用 `resume` 以下为协程的定义。

```
typedef struct _nty_coroutine_queue nty_coroutine_queue;

typedef struct _nty_coroutine_rbtreesleep nty_coroutine_rbtreesleep;
typedef struct _nty_coroutine_rbtreeswait nty_coroutine_rbtreeswait;

typedef struct _nty_schedule {
    uint64_t birth;
    nty_cpu_ctx ctx;

    struct _nty_coroutine *curr_thread;
    int page_size;

    int poller_fd;
    int eventfd;
    struct epoll_event eventlist[NTY_CO_MAX_EVENTS];
    int nevents;

    int num_new_events;

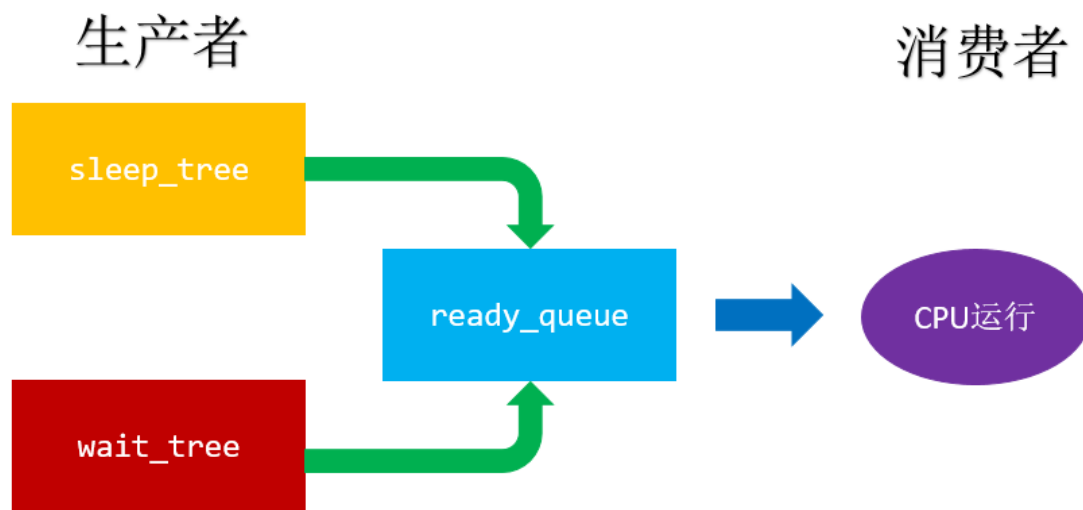
    nty_coroutine_queue ready;
    nty_coroutine_rbtreesleep sleeping;
    nty_coroutine_rbtreeswait waiting;
} nty_schedule;
```

第七章 协程的实现之调度器

问题：协程如何被调度？

调度器的实现，有两种方案，一种是生产者消费者模式，另一种多状态运行。

7.1 生产者消费者模式

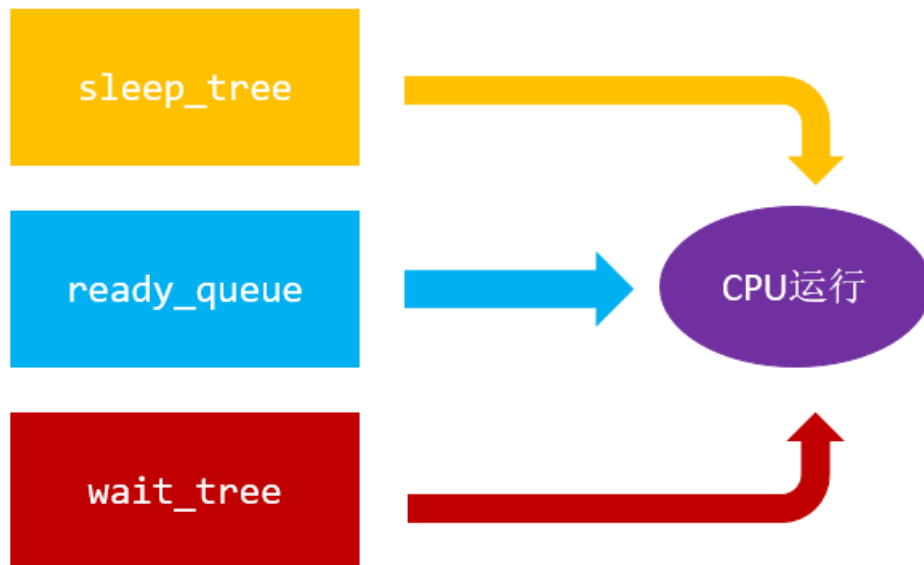


逻辑代码如下：

```
while (1) {  
  
    //遍历睡眠集合，将满足条件的加入到 ready  
    nty_coroutine *expired = NULL;  
    while ((expired = sleep_tree_expired(sched)) != ) {  
        TAILQ_ADD(&sched->ready, expired);  
    }  
  
    //遍历等待集合，将满足添加的加入到 ready  
    nty_coroutine *wait = NULL;  
    int nready = epoll_wait(sched->epfd, events, EVENT_MAX, 1);  
    for (i = 0; i < nready; i++) {  
        wait = wait_tree_search(events[i].data.fd);  
        TAILQ_ADD(&sched->ready, wait);  
    }  
  
    // 使用 resume 回复 ready 的协程运行权  
    while (!TAILQ_EMPTY(&sched->ready)) {  
        nty_coroutine *ready = TAILQ_POP(sched->ready);
```

```
        resume(ready);  
    }  
}
```

7.2 多状态运行



实现逻辑代码如下：

```
while (1) {  
  
    //遍历睡眠集合，使用 resume 恢复 expired 的协程运行权  
    nty_coroutine *expired = NULL;  
    while ((expired = sleep_tree_expired(sched)) != ) {  
        resume(expired);  
    }  
  
    //遍历等待集合，使用 resume 恢复 wait 的协程运行权  
    nty_coroutine *wait = NULL;  
    int nready = epoll_wait(sched->epfd, events, EVENT_MAX, 1);  
    for (i = 0; i < nready; i++) {  
        wait = wait_tree_search(events[i].data.fd);  
        resume(wait);  
    }  
  
    // 使用 resume 恢复 ready 的协程运行权  
    while (!TAILQ_EMPTY(sched->ready)) {  
        nty_coroutine *ready = TAILQ_POP(sched->ready);  
        resume(ready);  
    }  
}
```

}

第八章 协程性能测试

测试环境：4 台 VMWare 虚拟机

1 台服务器 6G 内存，4 核 CPU

3 台客户端 2G 内存，2 核 CPU

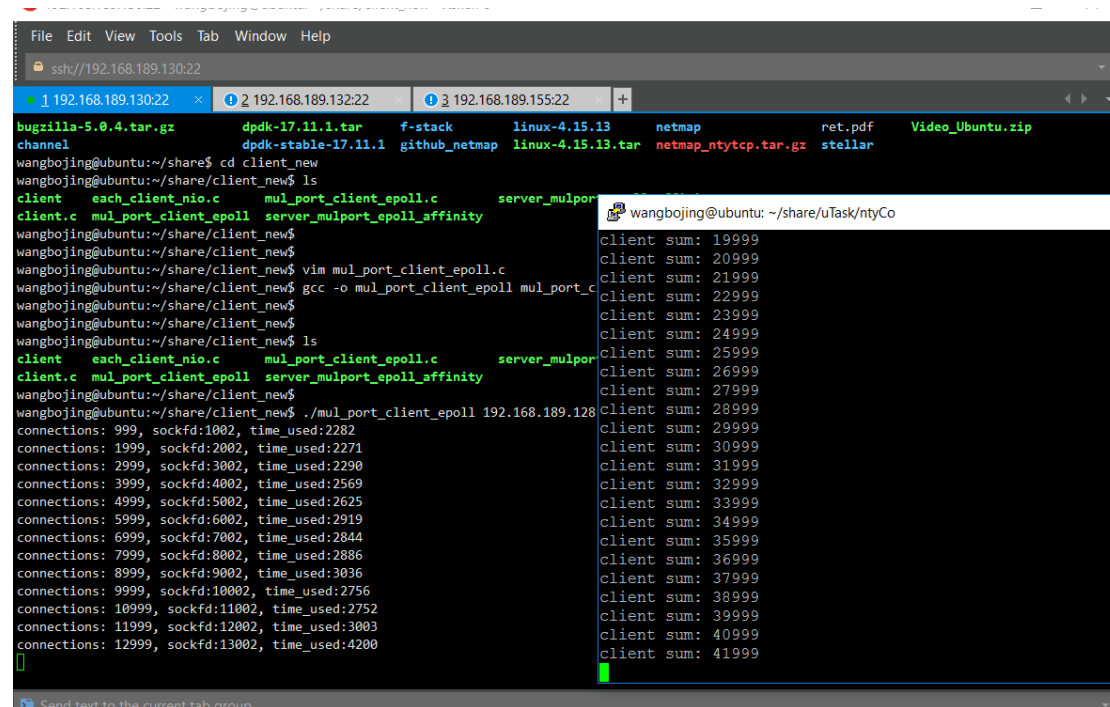
操作系统：ubuntu 14.04

服务器端测试代码：<https://github.com/wangbojing/NtyCo>

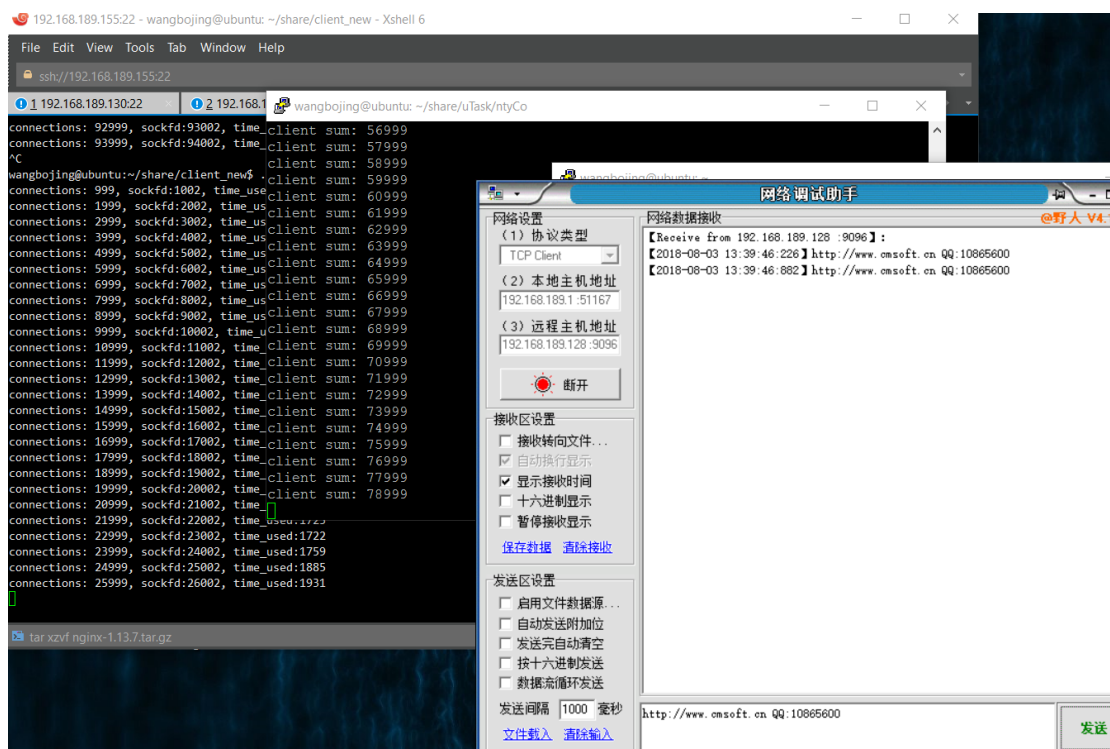
客户端测试代码：

https://github.com/wangbojing/cl000k_test/blob/master/client_mutlport_epoll.c

按照每一个连接启动一个协程来测试。每一个协程栈空间 4096byte
6G 内存 -> 测试协程数量 100W 无异常。并且能够正常收发数据。



```
File Edit View Tools Tab Window Help
ssh://192.168.189.130:22
bugzilla-5.0.4.tar.gz dpdk-17.11.1.tar f-stack linux-4.15.13 netmap ret.pdf Video_Ubuntu.zip
channel dpdk-stable-17.11.1 github_netmap linux-4.15.13.tar netmap_ntytcp.tar.gz stellar
wangbojing@ubuntu:~/share$ cd client_new
wangbojing@ubuntu:~/share/client_new$ ls
client      each_client_nio.c  mul_port_client_epoll.c  server_mulpor
client.c    mul_port_client_epoll  server_mulport_epoll_affinity
wangbojing@ubuntu:~/share/client_new$
wangbojing@ubuntu:~/share/client_new$ vim mul_port_client_epoll.c
wangbojing@ubuntu:~/share/client_new$ gcc -o mul_port_client_epoll mul_port.c
wangbojing@ubuntu:~/share/client_new$
wangbojing@ubuntu:~/share/client_new$ ls
client      each_client_nio.c  mul_port_client_epoll.c  server_mulpor
client.c    mul_port_client_epoll  server_mulport_epoll_affinity
wangbojing@ubuntu:~/share/client_new$
wangbojing@ubuntu:~/share/client_new$ ./mul_port_client_epoll 192.168.189.128
connections: 999, sockfd:1002, time_used:2282
connections: 1999, sockfd:2002, time_used:2271
connections: 2999, sockfd:3002, time_used:2290
connections: 3999, sockfd:4002, time_used:2569
connections: 4999, sockfd:5002, time_used:2625
connections: 5999, sockfd:6002, time_used:2919
connections: 6999, sockfd:7002, time_used:2844
connections: 7999, sockfd:8002, time_used:2886
connections: 8999, sockfd:9002, time_used:3036
connections: 9999, sockfd:10002, time_used:2756
connections: 10999, sockfd:11002, time_used:2752
connections: 11999, sockfd:12002, time_used:3003
connections: 12999, sockfd:13002, time_used:4200
client sum: 19999
client sum: 20999
client sum: 21999
client sum: 22999
client sum: 23999
client sum: 24999
client sum: 25999
client sum: 26999
client sum: 27999
client sum: 28999
client sum: 29999
client sum: 30999
client sum: 31999
client sum: 32999
client sum: 33999
client sum: 34999
client sum: 35999
client sum: 36999
client sum: 37999
client sum: 38999
client sum: 39999
client sum: 40999
client sum: 41999
```



第九章 协程多核模式