

# Libco 源码分析

状态：  <input type="checkbox"/> 草稿  <input checked="" type="checkbox"/> 修改中  <input type="checkbox"/> 定稿	文件标签：	Libco 源码分析
	版本：	V1.0
	作者：	张鸿
	日期：	2018-8-26

# 编辑历史

文件名称：Libco 源码分析				
文件说明：				
编辑历史：				
编辑时间	版本	作者	编辑内容	标记
2018-8-26	1.0	张鸿	初稿	

## 目 录

目 录.....	3
1 说明 .....	4
2 什么是协程? .....	4
3 协程的原理 .....	6
3.1 函数调用栈 .....	7
3.2 保存&恢复上下文 .....	9
3.3 任务调度 .....	11
3.3.1 <i>co_resume</i> .....	11
3.3.2 <i>co_yield_ct</i> .....	12
3.3.3 <i>co_swap</i> .....	12
3.4 协程同步 .....	12
3.4.1 <i>co_cond_timedwait</i> .....	12
3.4.2 <i>co_cond_signal</i> .....	12
3.4.3 <i>co_cond_broadcast</i> .....	12
3.4.4 <i>co_eventloop</i> .....	12
3.5 其它函数 .....	14
3.5.1 <i>Poll</i> .....	14
4 数据结构 .....	15
4.1 协程环境 .....	15
4.2 协程栈空间 .....	17
4.3 EPOLL.....	17
4.1 双向链表 .....	18
4.2 定时器 .....	18
5 HOOK 机制 .....	18

## 1 说明

libco 是腾讯开源的一个协程库，主要应用于微信后台 RPC 框架；

<https://github.com/Tencent/libco>

## 2 什么是协程？

协程，又称微线程，纤程。英文名 Coroutine。协程是一种用户态的轻量级线程，协程的调度完全由用户控制，协程拥有自己的寄存器上下文和栈；

### 2.1 协程与线程的区别

- 1) 线程是通过内核来调度的，而协程是用户态调度的；
- 2) 线程进程都是同步机制，而协程则是异步；
- 3) CPU 密集型应用适用于单进程或者多进程，IO 密集型应用则适用于协程；

生产者与消息者示例：

```

void* Producer(void* args)
{
    co_enable_hook_sys();
    stEnv_t* env= (stEnv_t*)args;
    int id = 0;
    while (true)
    {
        stTask_t* task = (stTask_t*)calloc(1, sizeof(stTask_t));
        task->id = id++;
        env->task_queue.push(task);
        printf("%s:%d produce task %d\n", __func__, __LINE__, task->id);
        co_cond_signal(env->cond);
        poll(NULL, 0, 3000);
    }
    return NULL;//每 3 秒都会执行一次，先打印 printf，再执行这里；
}

void* Consumer(void* args)
{
    co_enable_hook_sys();
    stEnv_t* env = (stEnv_t*)args;
    while (true)
    {
        if (env->task_queue.empty())
        {
            co_cond_timedwait(env->cond, -1);
            continue;
        }
        stTask_t* task = env->task_queue.front();
        env->task_queue.pop();
        printf("%s:%d consume task %d\n", __func__, __LINE__, task->id);
        free(task);
    }
    return NULL;
}

int main()
{
    stEnv_t* env = new stEnv_t;
    env->cond = co_cond_alloc();

    stCoRoutine_t* consumer_routine;
    co_create(&consumer_routine, NULL, Consumer, env);
    co_resume(consumer_routine);

    stCoRoutine_t* producer_routine;

```

## 2.2 对称协程与非对称协程

对称协程指调用者与被调用者是对等关系，子协程之间来回切换；

非对称协程指 被调协程让出 `cpu` 以后，必须回到调用协程；在实际应用中，非对称协程更好用；

`Libco` 属于非对称协程；

## 2.3 独占栈与共享栈

`stackful coroutine`：

`libco` 的协程是 `stackful coroutine`：每个协程都拥有一个独立的栈帧，协程切换时会保存当前协程栈中的所有数据，并加载新的栈帧对象。这样做的优点是：协程调度可以在内存中的任意位置、任意时刻进行；但是缺点也很明显：随着并发量的增加，协程的数目越来越多，当前内存中的协程栈(无论是 `occupy` 还是 `suspend`)越来越多，内存瓶颈开始显现，且内存切换本身也是不小的开销(寄存器恢复、数据拷贝)。所以，`stackful coroutine` 一般有栈大小的限制(`libco` 是 128K)。

`stackful coroutine`：

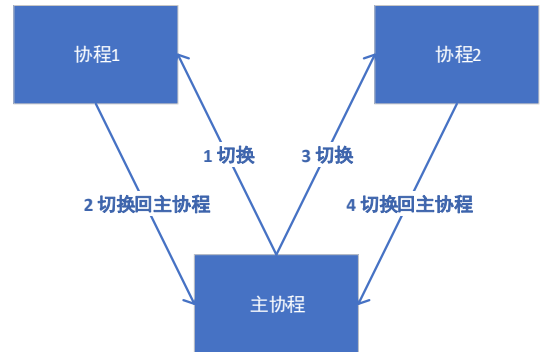
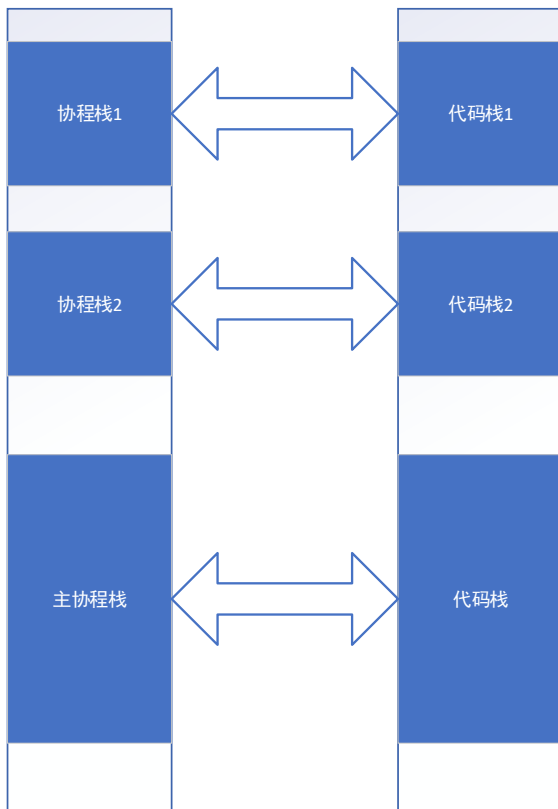
那么能不能所有正在执行的协程都共享一个栈呢？这就是共享栈的思路。共享栈将协程划分为协程组，同一个协程组中的协程共享同一块内存，在协程切换的时候将当前内存中的数据保存到运行协程的 `buffer` 中，并将新调度协程 `buffer` 中的数据拷贝到共享栈中。如此既可以减少内存开销，同时运行协程又没有了栈大小的限制。但是“鱼和熊掌不可兼得”(我居然会用谚语 `orz`)，共享栈的缺点是：协程调度产生的局部变量都在共享栈上，一旦新的协程运行后共享栈中的数据就会被覆盖，先前协程的局部变量也就不再有效，进而无法实现参数传递、嵌套调用等高级协程交互。不过 `libco` 作为一个生产库，不是研究协程语法的，每个协程作为一个“原子”的调用例程就够用了。

此外，`libco` 的协程在共享栈上的切换还实现了 `copy-on-write`，只有共享栈内存改变时才进行数据拷贝，这是传统的 `stackful coroutine` 所无法实现的。

## 3 协程的原理

协程是通过汇编语言来保存和恢复函数执行上下文、使用 `IO` 事件或者定时器事件来控制 `CPU` 的执行策略来实现的；

下图分别表示协程的内存分布及基本调度原理；



### 3.1 函数调用栈

其基本知道，可以先阅读此文 [https://nifengz.com/introduction\\_x64\\_assembly/](https://nifengz.com/introduction_x64_assembly/);

恢复函数调用的上下文，主要由以下内容决定：

#### 1) 通用寄存器

X86-64 有 16 个通用(几乎都是通用的)64 位整数寄存器：

%rax %rbx %rcx %rdx %rsi %rdi %rbp %rsp %r8 %r9 %r10 %r11 %r12 %r13 %r14 %r15

#### 2) RIP 寄存器

#### 3) 栈空间

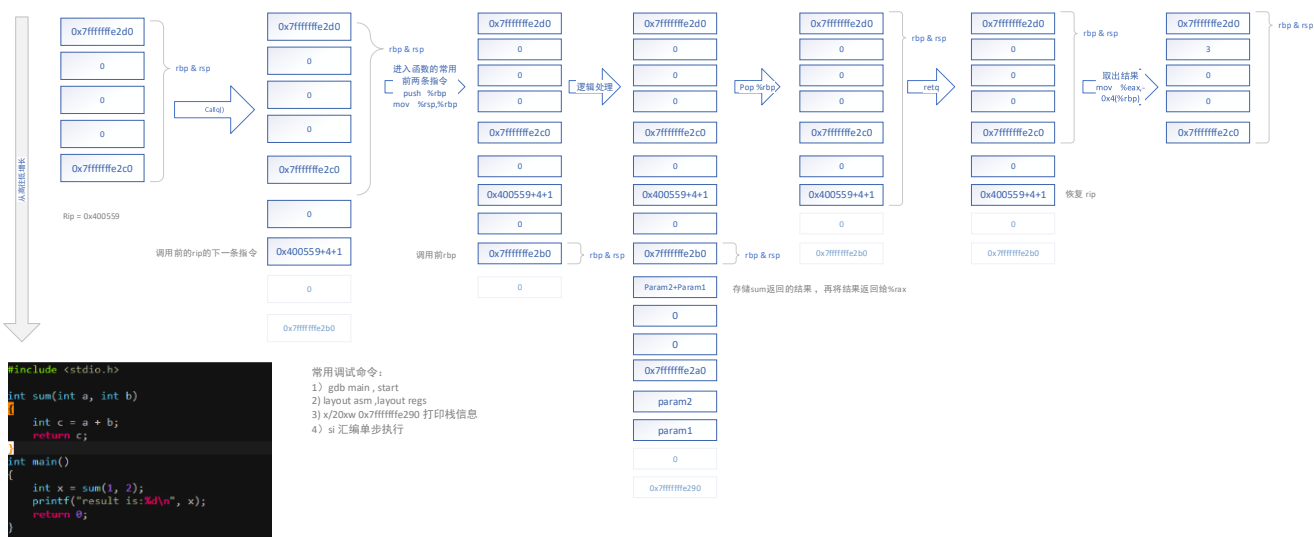
其中主要的寄存器说明：

1) RIP 指向汇编代码栈的指令地址，表示即将要执行的指令；

2) RSP 表示栈顶寄存器；

- 3) RBP 表示栈底寄存器;
- 4) RAX 为函数返回值的存储寄存器;
- 5) RDI 通常存储函数的第 1 个参数地址, RSI 通常存储函数的第 2 个参数地址;

如下图描述了调用函数时的 rbp, rsp, rip 的变化:



而 libco 如何控制函数切换出来时的 rip 的指令地址呢? 默认的编译器是在调用函数之前, 将被调函数的下一条指令保存在-0x10(rbp)的位置, 而 libco 则在恢复函数调用的过程中, 修改-0x10(rbp)的指令, 达到控制函数的跳转;







```

leaq 8(%rsp),%rax //放弃入参时的 回调函数地址
    leaq 112(%rdi),%rsp //rdi 存放的是第一个参数的值的指针
    pushq %rax
    pushq %rbx
    pushq %rcx
    pushq %rdx

    pushq -8(%rax) //ret func addr, 上一次将要执行的地址

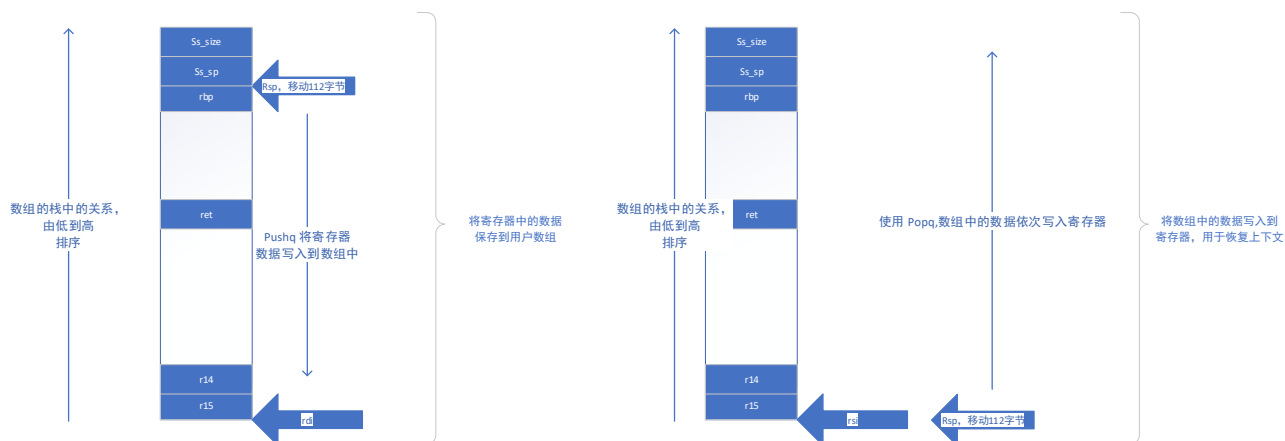
    pushq %rsi
    pushq %rdi
    pushq %rbp
    pushq %r8
    pushq %r9
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    movq %rsi, %rsp //rsi 存放第 2 个参数
    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %r9
    popq %r8
    popq %rbp
    popq %rdi
    popq %rsi
    popq %rax //ret func addr
    popq %rdx
    popq %rcx
    popq %rbx
    popq %rsp
    pushq %rax //将%rax 的执行地址替换成将要执行的地址

    xorl %eax, %eax
    ret

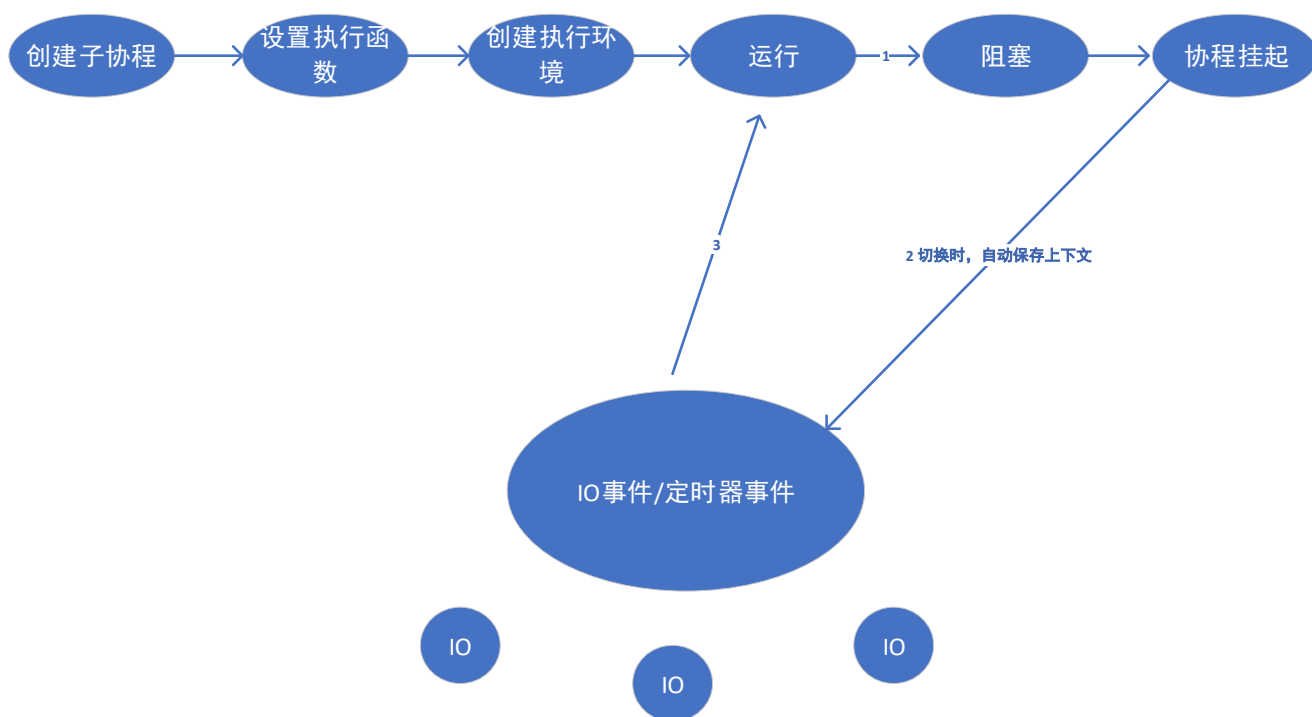
```

以上汇编的主要逻辑可以用下图表示;



### 3.3 任务调度

拿生产者与消息者的示例来描述任务调度的原理：



#### 3.3.1 co\_resume

切入协程, 获取 CPU;

### **3.3.2 co\_yield\_ct**

切出当前协程，让出 CPU

### **3.3.3 co\_swap**

协程切换的核心函数，通过汇编实现；

## **3.4 协程同步**

### **3.4.1 co\_cond\_timedwait**

将当前协程数据添加到同步事件单向链接中；

### **3.4.2 co\_cond\_signal**

从同步事件单向链接中取出协程数据，放入活跃的待处理单向链接中；

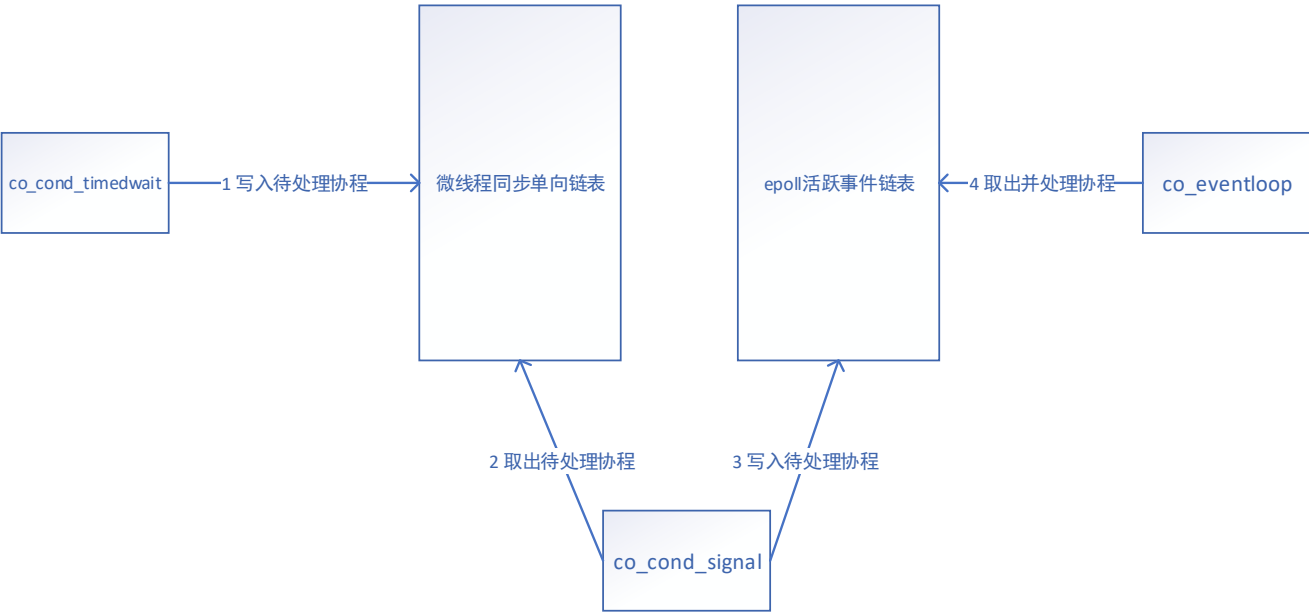
### **3.4.3 co\_cond\_broadcast**

从同步事件的单向链接中取出所有的协程数据，全部放入活跃的待处单向链接中；

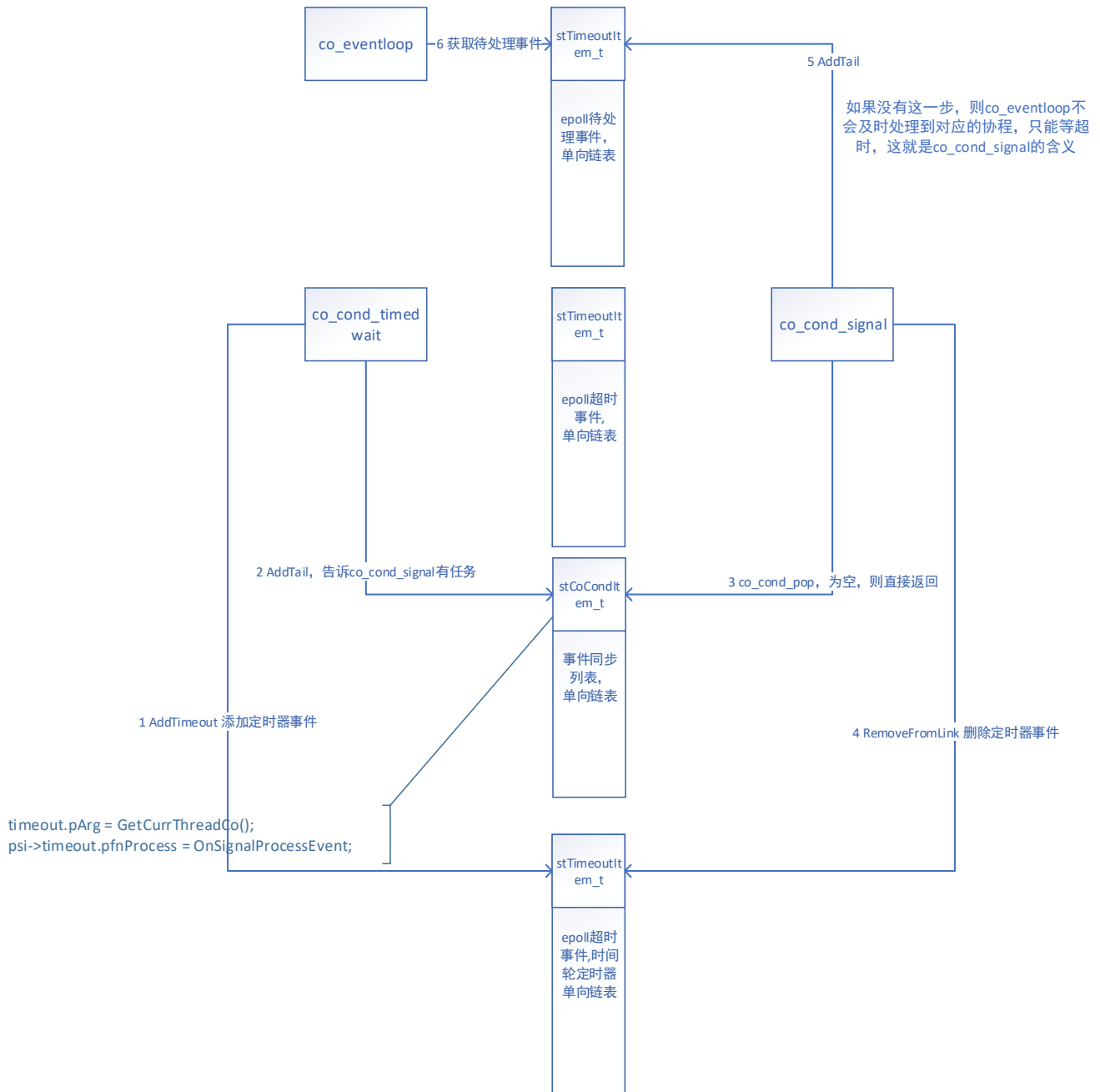
### **3.4.4 co\_eventloop**

从活跃链表中依次取出协程数据处理；

简单理解为如下图：



详细计录：



## 3.5 其它函数

### 3.5.1 Poll

调用 co\_poll\_inner 函数, 插入一条超时任务:

```
arg.pfnProcess = OnPollProcessEvent;
```

```

arg.pArg = GetCurrCo( co_get_curr_thread_env() );

unsigned long long now = GetTickMS();

arg.ullExpireTime = now + timeout;

int ret = AddTimeout( ctx->pTimeout,&arg,now );//注册定时器

```

待定时任务回来时，执行该协程时，删除这条定时任务；

```

RemoveFromLink<stTimeoutItem_t,stTimeoutItemLink_t>( &arg );//删除定时器

```

## 4 数据结构

### 4.1 协程环境

```

struct stCoRoutineEnv_t
{
    stCoRoutine_t *pCallStack[ 128 ];
    int iCallStackSize;
    stCoEpoll_t *pEpoll;

    //for copy stack log lastco and nextco
    stCoRoutine_t* pending_co;
    stCoRoutine_t* occupy_co;
};

```

其中 pCallStack[0]存储的是主协程，pCallStack[1]存储的是当前正在运行的协程，在生产者与消息消费者的示例中，pCallStack 只会使用到前两个数组，对于挂起的协程环境是存储在事件的双向链表中，都过事件触发机制来控制；

pEpoll 表示 EPOLL IO 管理器，是结合定时器或者 IO 事件来管理协程的调度的；

```

struct stCoRoutine_t
{
    stCoRoutineEnv_t *env;
    pfn_co_routine_t pfn;
    void *arg;
    coctx_t ctx;

    char cStart;
    char cEnd;
    char cIsMain;
    char cEnableSysHook;
    char cIsShareStack;

    void *pvEnv;

    //char sRunStack[ 1024 * 128 ];
    stStackMem_t* stack_mem;

    //save satch buffer while confilct on same stack_buffer;
    char* stack_sp;
    unsigned int save_size;
    char* save_buffer;

    stCoSpec_t aSpec[1024];
};

```

以上结构表示某个协程的具体内容，pfn 表示该协程对应的执行函数指针；

ctx 存储的是当前协程的上下文，在调用 co\_swap 时使用；

clsShareStack 是否使用协程的共享栈模式；



## 4.2 协程栈空间

```
struct coctx_t
{
    #if defined(__i386__)
        void *regs[ 8 ];
    #else
        void *regs[ 14 ];
    #endif
    size_t ss_size;
    char *ss_sp;
};
```

上面为保存寄存器的数据结构，64 位 CPU 常用寄存器为 14 个；其中 `ss_sp` 表示创建协程环境时，为协程分配的栈空间的低地址，`ss_size = 128 * 1024` 表示栈空间大小，运行栈函数时，`rsp, rbp` 都将指向这一段空间运行，也即协程执行函数的分配将会在这段空间，如果栈中的临时变量占用函数超过这块栈空间，则出错；每个协程都会默认创建一段大小为 `ss_size` 的栈空间；

## 4.3 EPOLL

```
struct stCoEpoll_t
{
    int iEpollFd;
    static const int _EPOLL_SIZE = 1024 * 10;

    struct stTimeout_t *pTimeout; // 时间轮定时器

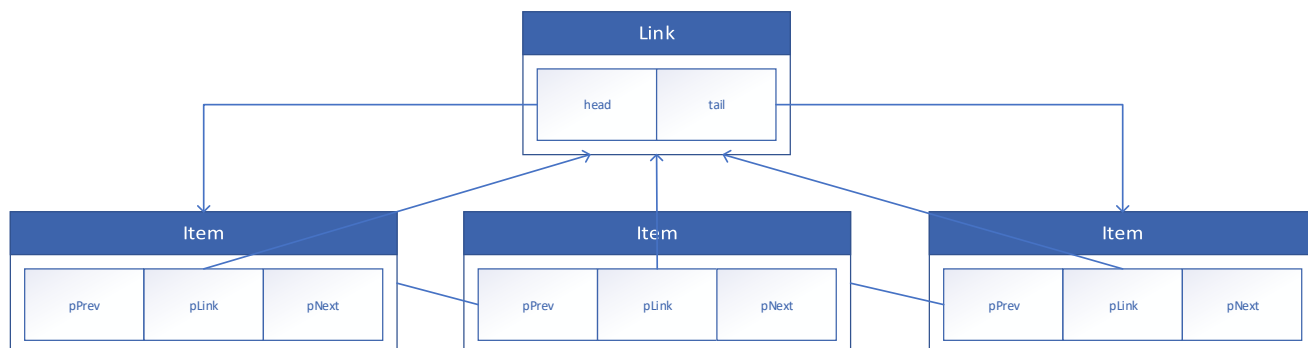
    struct stTimeoutItemLink_t *pstTimeoutList; // 已经超时的时间

    struct stTimeoutItemLink_t *pstActiveList; // 活跃的事件

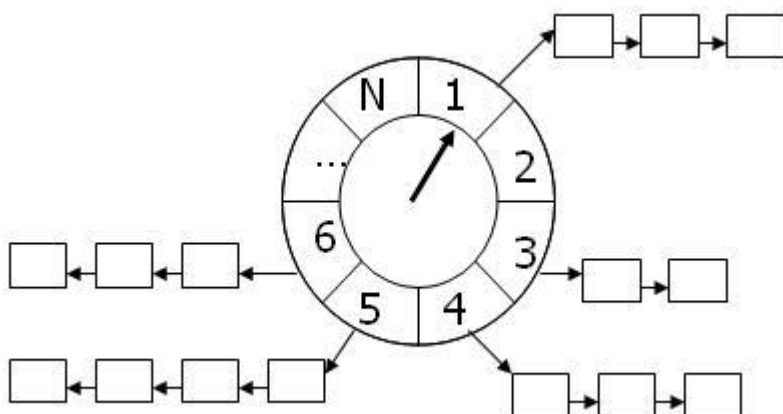
    co_epoll_res *result;
};
```

## 4.1 双向链表

Epoll 的数据结构中都有使用到;



## 4.2 定时器



LIBCO 是采用时间轮的定时器，最小单位为 1ms，最大长度为 40s;

## 5 HOOK 机制

Libco 将 linux 的 IO 相关的函数阻塞函数重新实现了，将在其中注入 Epoll 或者定时器事件，有以下函数：

- Socket
- Co\_accept
- Connect
- Close
- Read
- Write
- Sendto
- Recvfrom

- Send
- Recv

在调用前，可以通过调用 `co_enable_hook_sys()` 来决定，是否使用 Hook 机制，实现同步代码与异步代码的优雅切换；

示例：

```
..
ssize_t read( int fd, void *buf, size_t nbyte )
{
    HOOK_SYS_FUNC( read );

    if( !co_is_enable_sys_hook() )
    {
        return g_sys_read_func( fd,buf,nbyte );
    }
    rpchhook_t *lp = get_by_fd( fd );

    if( !lp || ( O_NONBLOCK & lp->user_flag ) )
    {
        ssize_t ret = g_sys_read_func( fd,buf,nbyte );
        return ret;
    }
    int timeout = ( lp->read_timeout.tv_sec * 1000 )
        + ( lp->read_timeout.tv_usec / 1000 );

    struct pollfd pf = { 0 };
    pf.fd = fd;
    pf.events = ( POLLIN | POLLERR | POLLHUP );
    int pollret = poll( &pf,1,timeout );

    ssize_t readret = g_sys_read_func( fd,(char*)buf ,nbyte );

    if( readret < 0 )
    {
        co_log_err("CO_ERR: read fd %d ret %d errno %d poll ret %d timeout %d",
            fd,readret,errno,pollret,timeout);
    }

    return readret;
} « end read »
..
```

通过epoll来管理

## 6 源码分析

### 6.1 Example\_cond 生产者与消息者示例

### 6.2 Example\_echocli 异步客户端

### 6.3 Example\_echosvr 搭建一个简单的服务

- 1) 可以启动多个进程，同时 accept，linux 2.6 版本以后，多进程同时 accept 只有一个生效；
- 2) 每个进程一个 accept 协程，循环处理；
- 3) 每个进程多个协程，每个协程对应一个长链接，对于多长链接请求，需要启动多个协程；
- 4) 进程 for 循环前，通过队列实现协程间的同步通知；

```

for(;;)
{
    struct pollfd pf = { 0 };
    pf.fd = fd;
    pf.events = (POLLIN|POLLERR|POLLHUP);
    co_poll( co_get_epoll_ct(), &pf, 1, 1000);

    int ret = read( fd, buf, sizeof(buf) );
    if( ret > 0 )
    {
        ret = write( fd, buf, ret );
    }
    if( ret <= 0 )
    {
        close( fd );
        break;
    }
}

} « end for ;; »
return 0;
} « end readwrite_routine »

int co_accept(int fd, struct sockaddr *addr, socklen_t *len );
static void *accept_routine( void * )
{
    co_enable_hook_sys();
    printf("accept_routine\n");
    fflush(stdout);
    for(;;)
    {
        //printf("pid %ld g_readwrite.size %ld\n", getpid(), g_readwrite.size());
        if( g_readwrite.empty() )
        {
            printf("empty\n"); //sleep
            struct pollfd pf = { 0 };
            pf.fd = -1;
            poll( &pf, 1, 1000);

            continue;
        }
        struct sockaddr_in addr; //maybe sockaddr_un;
        memset( &addr, 0, sizeof(addr) );
        socklen_t len = sizeof(addr);

        int fd = co_accept(g_listen_fd, (struct sockaddr * )&addr, &len);
        if( fd < 0 )
        {
            struct pollfd pf = { 0 };
            pf.fd = g_listen_fd;
            pf.events = (POLLIN|POLLERR|POLLHUP);
            co_poll( co_get_epoll_ct(), &pf, 1, 1000 );
            continue;
        }
        if( g_readwrite.empty() )
        {
            close( fd );
            continue;
        }
        SetNonBlock( fd );
        task_t *co = g_readwrite.top();
        co->fd = fd;
        g_readwrite.pop();
        co_resume( co->co );
    }
} « end for ;; »

```

协程相互切换

