

C++ 开源协程库 libco——原理及应用

滴滴平台技术部 · 王亮 2016 年 11 月 26 日

1 导论

使用 C++ 来编写高性能的网络服务器程序，从来都不是件很容易的事情。在没有应用任何网络框架，从 `epoll/kqueue` 直接码起的时候尤其如此。即便使用 `libevent`, `libev` 这样事件驱动的网络框架去构建你的服务，程序结构依然不会很简单。为何会这样？因为这类框架提供的都是非阻塞式的、异步的编程接口，异步的编程方式，这需要思维方式的转变。为什么 `golang` 近几年能够大规模流行起来呢？因为简单。这方面最突出的一点便是它的网络编程 API，完全同步阻塞式的接口。要并发？`go` 出一个协程就好了。相信对于很多人来说，最开始接触这种编程方式，是有点困惑的。程序中到处都是同步阻塞式的调用，这程序性能能好吗？答案是，好，而且非常好。那么 `golang` 是如何做到的呢？秘诀就在它这个协程机制里。

在 `go` 语言的 API 里，你找不到像 `epoll/kqueue` 之类的 I/O 多路复用 (I/O multiplexing) 接口，那它是怎么做做到轻松支持数万乃至十多万高并发的网络 IO 的呢？在 Linux 或其他类 Unix 系统里，支持 I/O 多路复用事件通知的系统调用 (System Call) 不外乎 `epoll/kqueue`，它难道可以离开这些系统接口另起炉灶？这个自然是不可能的。聪明的读者，应该大致想到了这背后是怎么个原理了。

语言内置的协程并发模式，同步阻塞式的 IO 接口，使得 `golang` 网络编程十分容易。那么 C++ 可不可以做到这样呢？

本文要介绍的开源协程库 `libco`，就是这样神奇的一个开源库，让你的高性能网络服务器编程不再困难。

`Libco` 是微信后台大规模使用的 C++ 协程库，在 2013 年的时候作为腾讯六大开源项目首次开源。据说 2013 年至今稳定运行在微信后台的数万台机器上。从本届 ArchSummit 北京峰会来自腾讯内部的分享经验来看，它在腾讯内部使用确实是比较广泛的。同 `go` 语言一样，`libco` 也是提供了同步风格编程模式，同时还能保证系统的高并发能力。

2 准备知识

2.1 协程 (Coroutine) 是什么?

协程这个概念，最近这几年可是相当地流行了。尤其 go 语言问世之后，内置的协程特性，完全屏蔽了操作系统线程的复杂细节；甚至使 go 开发者“只知有协程，不知有线程”了。当然 C++, Java 也不甘落后，如果你有关关注过 C++ 语言的最新动态，可能也会注意到近几年不断有人在给 C++ 标准委员会提协程的支持方案；Java 也同样有一些试验性的解决方案在提出来。

在 go 语言大行其道的今天，没听说过协程这个词的程序员应该很少了，甚至直接接触过协程编程的 (golang, lua, python 等) 也不在少数。你可能以为这是个比较新的东西，但其实协程这个概念在计算机领域已经相当古老了。早在七十年代，Donald Knuth 在他的神作 *The Art of Computer Programming* 中将 Coroutine 的提出者归于 Conway Melvin。同时，Knuth 还提到，coroutines 不过是一种特殊的 subroutines (Subroutine 即过程调用，在很多高级语言中也叫函数，为了方便起见，下文我们将它称为“函数”)。当调用一个函数时，程序从函数的头部开始执行，当函数退出时，这个函数的声明周期也就结束了。一个函数在它的生命周期中，只可能返回一次。而协程则不同，协程在执行过程中，可以调用别的协程自己则中途退出执行，之后又从调用别的协程的地方恢复执行。这有点像操作系统的线程，执行过程中可能被挂起，让位于别的线程执行，稍后又从挂起的地方恢复执行。在这个过程中，协程与协程之间实际上不是普通“调用者与被调用者”的关系，他们之间的关系是对称的 (symmetric)。实际上，协程不一定是这种对称的关系，还存在着一种非对称的协程模式 (asymmetric coroutines)。非对称协程其实也比较常见，本文要介绍的 libco 其实就是一种非对称协程，Boost C++ 库也提供了非对称协程。

具体来讲，非对称协程 (asymmetric coroutines) 是跟一个特定的调用者绑定的，协程让出 CPU 时，只能让回给原调用者。那到底是什么东西“不对称”呢？其实，非对称在于程序控制流转移到被调协程时使用的是 call/resume 操作，而当被调协程让出 CPU 时使用的却是 return/yield 操作。此外，协程间的地位也不对等，caller 与 callee 关系是确定的，不可更改的，非对称协程只能返回最初调用它的协程。

对称协程 (symmetric coroutines) 则不一样，启动之后就跟启动之前的协程没有任何关系了。协程的切换操作，一般而言只有一个操作，yield，用于将程序控制流转移给另外的协程。对称协程机制一般需要一个调度器的支持，按一定调度算法去选择 yield 的目标协程。

Go 语言提供的协程，其实就是典型的对称协程。不但对称，goroutines 还可以在多个线程上迁移。这种协程跟操作系统中的线程非常相似，甚至可以叫做“用户级线程”了。而 libco 提供的协程，虽然编程接口跟 pthread 有点类似，“类 pthread 的接口设计”，“如线程库一样轻松”，本质上却是一种非对称协程。这一点不要被表象蒙蔽了。事实上，libco 内部还为保存协程的调用链留了一个 stack 结构，而这个 stack 大小只有固定的 128。使用 libco，如果不断地在一个协程运行过程中启动另一个协程，随着嵌套深度增加就可能会造成这个栈空间溢出。

2.2 栈的概念回顾

TBD

3 Libco 使用简介

3.1 一个简单的例子

在多线程编程教程中，有一个经典的例子：生产者消费者问题。事实上，生产者消费者问题也是最适合协程的应用场景。那么我们就从这个简单的例子入手，来看一看使用 libco 编写的生产者消费者程序（例程代码来自于 libco 源码包）。

```
1 struct stTask_t {
2     int id;
3 };
4
5 struct stEnv_t {
6     stCoCond_t* cond;
7     queue<stTask_t*> task_queue;
8 };
9
10 void* Producer(void* args) {
11     co_enable_hook_sys();
12     stEnv_t* env = (stEnv_t*)args;
13     int id = 0;
14     while (true) {
15         stTask_t* task = (stTask_t*)calloc(1, sizeof(stTask_t));
16         task->id = id++;
17         env->task_queue.push(task);
18         printf("%s:%d produce task %d\n", __func__, __LINE__, task->id);
19         co_cond_signal(env->cond);
20         poll(NULL, 0, 1000);
21     }
22     return NULL;
23 }
24
25 void* Consumer(void* args) {
26     co_enable_hook_sys();
27     stEnv_t* env = (stEnv_t*)args;
28     while (true) {
29         if (env->task_queue.empty()) {
30             co_cond_timedwait(env->cond, -1);
31             continue;
32         }
33         stTask_t* task = env->task_queue.front();
34         env->task_queue.pop();
35         printf("%s:%d consume task %d\n", __func__, __LINE__, task->id);
36         free(task);
37     }
38     return NULL;
39 }
```

生产者和消费者协程

在上面的代码中，Producer 与 Consumer 函数分别实现了生产者与消费者的逻辑，函数的原型跟 `pthread` 线程函数原型也是一样的。不同的是，在函数第一行还调用了一个 `co_enable_hook_sys()`，此外，不是用 `sleep()` 去等待，而是 `poll()`。这些原因后文会详细解释，暂且不管。接下来我们看怎样创建和启动生产者消费者协程。

```
1 int main() {
2     stEnv_t* env = new stEnv_t;
3     env->cond = co_cond_alloc();
4
5     stCoRoutine_t* consumer_routine;
6     co_create(&consumer_routine, NULL, Consumer, env);
7     co_resume(consumer_routine);
8
9     stCoRoutine_t* producer_routine;
10    co_create(&producer_routine, NULL, Producer, env);
11    co_resume(producer_routine);
12
13    co_eventloop(co_get_epoll_ct(), NULL, NULL);
14    return 0;
15 }
```

创建和启动生产者消费者协程

初次接触 libco 的读者，应该下载源码编译，亲自运行一下这个例子看看输出结果是什么。实际上，这个例子的输出结果跟多线程实现方案是相似的，Producer 与 Consumer 交替打印生产和消费信息。

再来看代码，在 `main()` 函数中，我们看到代表一个协程的结构叫做 `stCoRoutine_t`，创建一个协程使用 `co_create()` 函数。我们注意到，这里的 `co_create()` 的接口设计跟 `pthread` 的 `pthread_create()` 是非常相似的。跟 `pthread` 不太一样是，创建出一个协程之后，并没有立即启动起来；这里要启动协程，还需调用 `co_resume()` 函数。最后，`pthread` 创建线程之后主线程往往会 `pthread_join()` 等等子线程退出，而这里的例子没有“`co_join()`”或类似的函数，而是调用了一个 `co_eventloop()` 函数，这些差异的原因我们后文会详细解析。

然后再看 Producer 和 Consumer 的实现，细心的读者可能会发现，无论是 Producer 还是 Consumer，它们在操作共享的队列时都没有加锁，没有互斥保护。那么这样做是否安全呢？其实是安全的。在运行这个程序时，我们用 `ps` 命令会看到这个它实际上只有一个线程。因此在任何时刻处理器上只会有一个协程在运行，所以不存在 `race conditions`，不需要任何互斥保护。

还有一个问题。这个程序既然只有一个线程，那么 Producer 与 Consumer 这两个协程函数是怎样做到交替执行的呢？如果你熟悉 `pthread` 和操作系统多线程的原理，应该很快能发现程序里 `co_cond_signal()`、`poll()` 和 `co_cond_timedwait()` 这几个关键点。换作是一个 `pthread` 编写的生产者消费者程序，在只有单核 CPU 的机器上执行，结果是不是一样的？

总之，这个例子跟 `pthread` 实现的生产者消费者程序是非常相似的。通过这个例子，我们也大致对 libco 的协程接口有了初步的了解。为了能看懂本文接下来的内容，建议把其他几个例子的代码也都浏览一下。下文我们将不再直接列出 libco 例子中的代码，如果有引用到，请自行参看相关代码。

4 libco 的协程

通过上一节的例子，我们已经对 libco 中的协程有了初步的印象。我们完全可以把它当做一种用户态线程来看待，接下来我们就从线程的角度来开始探究和理解它的实现机制。

以 Linux 为例，在操作系统提供的线程机制中，一个线程一般具备下列要素：

(1) 有一段程序供其执行，这个是显然是必须的。另外，不同线程可以共用同一段程序。这个也是显然的，想想我们程序设计里经常用到的线程池、工作线程，不同的工作线程可能执行完全一样的代码。

(2) 有起码的“私有财产”，即线程专属的系统堆栈空间。

(3) 有“户口”，操作系统教科书里叫做“进（线）程控制块”，英文缩写叫 PCB。在 Linux 内核里，则为 task_struct 的一个结构体。有了这个数据结构，线程才能成为内核调度的一个基本单位接受内核调度。这个结构也记录着线程占有的各项资源。

此外，值得一提的是，操作系统的进程还有自己专属的内存空间（用户态内存空间），不同进程间的内存空间是相互独立，互不干扰的。而同属一个进程的各线程，则是共享内存空间的。显然，协程也是共享内存空间的。

我们可以借鉴操作系统线程的实现思想，在 OS 之上实现用户级线程（协程）。跟 OS 线程一样，用户级线程也应该具备这三个要素。所不同的只是第二点，用户级线程（协程）没有自己专属的堆空间，只有栈空间。首先，我们得准备一段程序供协程执行，这即是 co_create() 函数在创建协程的时候传入的第三个参数——形参为 void*，返回值为 void 的一个函数。

其次，需要为创建的协程准备一段栈内存空间。栈内存用于保存调用函数过程中的临时变量，以及函数调用链（栈帧）。在 Intel 的 x86 以及 x64 体系结构中，栈顶由 ESP (RSP) 寄存器确定。所以一个创建一个协程，启动的时候还要将 ESP (RSP) 切到分配的栈内存上，后文将对此做详细分析。

co_create() 调用成功后，将返回一个 stCoRoutine_t 的结构指针（第一个参数）。从命名上也可以看出来，该结构即代表了 libco 的协程，记录着一个协程拥有的各种资源，我们不妨称之为“协程控制块”。这样，构成一个协程三要素——执行的函数，栈内存，协程控制块，在 co_create() 调用完成后便都准备就绪了。

5 关键数据结构及其关系

```
1 struct stCoRoutine_t {
2     stCoRoutineEnv_t *env;
3     pfn_co_routine_t pfn;
4     void *arg;
5     coctx_t ctx;
6
7     char cStart;
8     char cEnd;
9     char cIsMain;
10    char cEnableSysHook;
11    char cIsShareStack;
```



```

12
13 void *pvEnv;
14
15 //char sRunStack[ 1024 * 128 ];
16 stStackMem_t* stack_mem;
17
18 //save satch buffer while confilct on same stack_buffer;
19 char* stack_sp;
20 unsigned int save_size;
21 char* save_buffer;
22
23 stCoSpec_t aSpec[1024];
24 };

```

libco 的协程控制块 stCoRoutine_t

接下来我们逐个来看一下 stCoRoutine_t 结构中的各项成员。首先看第 2 行的 env，协程执行的环境。这里提一下，不同于 go 语言，libco 的协程一旦创建之后便跟创建时的那个线程绑定了，是不支持在不同线程间迁移（migrate）的。这个 env，即同属于一个线程所有协程的执行环境，包括了当前运行协程、上次切换挂起的协程、嵌套调用的协程栈，和一个 epoll 的封装结构（TBD）。第 3、4 行分别为实际待执行的协程函数以及参数。第 5 行，ctx 是一个 coctx_t 类型的结构，用于协程切换时保存 CPU 上下文（context）的；所谓的上下文，即 esp、ebp、eip 和其他通用寄存器的值。第 7 至 11 行是一些状态和标志变量，意义也很明了。第 13 行 pvEnv，名字看起来有点费解，我们暂且知道这是一个用于保存程序系统环境变量的指针就好了。16 行这个 stack_mem，协程运行时的栈内存。通过注释我们知道这个栈内存是固定的 128KB 的大小。我们可以计算一下，每个协程 128K 内存，那么一个进程启 100 万个协程则需要占用高达 122GB 的内存。读者大概会怀疑，不是常听说协程很轻量级吗，怎么会占用这么多的内存？答案就在接下来 19 至 21 行的几个成员变量中。这里要提到实现 stackful 协程（与之相对的还有一种 stackless 协程）的两种技术：Separate coroutine stacks 和 Copying the stack（又叫共享栈）。实现细节上，前者为每一个协程分配一个单独的、固定大小的栈；而后者则仅为正在运行的协程分配栈内存，当协程被调度切换出去时，就把它实际占用的栈内存 copy 保存到一个单独分配的缓冲区；当被切出去的协程再次调度执行时，再一次 copy 将原来保存的栈内存恢复到那个共享的、固定大小的栈内存空间。通常情况下，一个协程实际占用的（从 esp 到栈底）栈空间，相比预分配的这个栈大小（比如 libco 的 128KB）会小得多；这样一来，copying stack 的实现方案所占用的内存便会少很多。当然，协程切换时拷贝内存的开销有些场景下也是很大的。因此两种方案各有利弊，而 libco 则同时实现了两种方案，默认使用前者，也允许用户在创建协程时指定使用共享栈。

```

1 struct coctx_t {
2 #if defined(__i386__)
3     void *regs[8];
4 #else
5     void *regs[14];
6 #endif
7     size_t ss_size;
8     char *ss_sp;

```

```
9 };
```

用于保存协程执行上下文的 `coctx_t` 结构

前文还提到，协程控制块 `stCoRoutine_t` 结构里第一个字段 `env`，用于保存协程的运行“环境”。前文也指出，这个结构是跟运行的线程绑定了的，运行在同一个线程上的各协程是共享该结构的，是个全局性的资源。那么这个 `stCoRoutineEnv_t` 到底包含什么重要信息呢？请看代码：

```
1 struct stCoRoutineEnv_t {
2     stCoRoutine_t *pCallStack[128];
3     int iCallStackSize;
4     stCoEpoll_t *pEpoll;
5
6     // for copy stack log lastco and nextco
7     stCoRoutine_t* pending_co;
8     stCoRoutine_t* occupy_co;
9 };
```

协程的 `stCoRoutineEnv_t` 结构

我们看到 `stCoRoutineEnv_t` 内部有一个叫做 `CallStack` 的“栈”，还有个 `stCoPoll_t` 结构指针。此外，还有两个 `stCoRoutine_t` 指针用于记录协程切换时占有共享栈的和将要切换运行的协程。在不使用共享栈模式时 `pending_co` 和 `occupy_co` 都是空指针，我们暂且忽略它们，等到分析共享栈的时候再说。

`stCoRoutineEnv_t` 结构里的 `pCallStack` 不是普通意义上我们讲的那个程序运行栈，那个 `ESP (RSP)` 寄存器指向的栈，是用来保留程序运行过程中局部变量以及函数调用关系的。但是，这个 `pCallStack` 又跟 `ESP (RSP)` 指向的栈有相似之处。如果将协程看成一种特殊的函数，那么这个 `pCallStack` 就时保存这些函数的调用链的栈。我们已经讲过，非对称协程最大特点就是协程间存在明确的调用关系；甚至在有些文献中，启动协程被称作 `call`，挂起协程叫 `return`。非对称协程机制下的被调协程只能返回到调用者协程，这种调用关系不能乱，因此必须将调用链保存下来。这即是 `pCallStack` 的作用，将它命名为“调用栈”实在是恰如其分。

每当启动 (`resume`) 一个协程时，就将它的协程控制块 `stCoRoutine_t` 结构指针保存在 `pCallStack` 的“栈顶”，然后“栈指针” `iCallStackSize` 加 1，最后切换 `context` 到待启动协程运行。当协程要让出 (`yield`) `CPU` 时，就将它的 `stCoRoutine_t` 从 `pCallStack` 弹出，“栈指针” `iCallStackSize` 减 1，然后切换 `context` 到当前栈顶的协程（原来被挂起的调用者）恢复执行。这个“压栈”和“弹栈”的过程我们在 `co_resume()` 和 `co_yield()` 函数中将会再次讲到。

那么这里有一个问题，`libco` 程序的第一个协程呢，假如第一个协程 `yield` 时，`CPU` 控制权让给谁呢？关于这个问题，我们首先要明白这“第一个”协程是什么。实际上，`libco` 的第一个协程，即执行 `main` 函数的协程，是一个特殊的协程。这个协程又可以称作主协程，它负责协调其他协程的调度执行（后文我们会看到，还有网络 I/O 以及定时事件的驱动），它自己则永远不会 `yield`，不会主动让出 `CPU`。不让出 (`yield`) `CPU`，不等于说它一直霸占着 `CPU`。我们知道 `CPU` 执行权有两种转移途径，一是通过 `yield` 让给调用者，其二则是 `resume` 启动其他协程运行。后文我们可以清楚地看到，`co_resume()`

与 `co_yield()` 都伴随着上下文切换，即 CPU 控制流的转移。当你在程序中第一次调用 `co_resume()` 时，CPU 执行权就从主协程转移到了 `resume` 目标协程上了。

提到主协程，那么另外一个问题又来了，主协程是在什么时候创建出来的呢？什么时候 `resume` 的呢？事实上，主协程是跟 `stCoRoutineEnv_t` 一起创建的。主协程也无需调用 `resume` 来启动，它就是程序本身，就是 `main` 函数。主协程是一个特殊的存在，可以认为它只是一个结构体而已。在程序首次调用 `co_create()` 时，此函数内部会判断当前进程（线程）的 `stCoRoutineEnv_t` 结构是否已分配，如果未分配则分配一个，同时分配一个 `stCoRoutine_t` 结构，并将 `pCallStack[0]` 指向主协程。此后如果用 `co_resume()` 启动协程，又会将 `resume` 的协程压入 `pCallStack` 栈。以上整个过程可以用图1来表示。

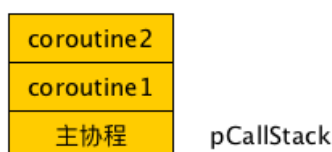


图 1: `stCoRoutineEnv_t` 结构的 `pCallStack` 示意图

在图1中，`coroutine2` 整处于栈顶，也即是说，当前正在 CPU 上 `running` 的协程是 `coroutine2`。而 `coroutine2` 的调用者是谁呢？是谁 `resume` 了 `coroutine2` 呢？是 `coroutine1`。`coroutine1` 则是主协程启动的，即在 `main` 函数里 `resume` 的。当 `coroutine2` 让出 CPU 时，只能让给 `coroutine1`；如果 `coroutine1` 再让出 CPU，那么又回到了主协程的控制流上了。

当控制流回到主协程上时，主协程在干些什么呢？回过头来看生产者消费者那个例子。那个例子中，`main` 函数中程序最终调用了 `co_eventloop()`。该函数是一个基于 `epoll/kqueue` 的事件循环，负责调度其他协程运行，具体细节暂时略去。这里我们只需知道，`stCoRoutineEnv_t` 结构中的 `pEpoll` 即使在这里用的就够了。

至此，我们已经基本理解了 `stCoRoutineEnv_t` 结构的作用。待补充。

6 Libco 协程的生命周期

6.1 创建协程（Creating coroutines）

前文已提到，libco 中创建协程是 `co_create()` 函数。函数声明如下：

```
int co_create(stCoRoutine_t** co, const stCoRoutineAttr_t* attr, void* (*routine)(void*), void* arg);
```

同 `pthread_create` 一样，该函数有四个参数：

@`co`: `stCoRoutine_t**` 类型的指针。输出参数，`co_create` 内部会为新协程分配一个“协程控制块”，`*co` 将指向这个分配的协程控制块。

@`attr`: `stCoRoutineAttr_t*` 类型的指针。输入参数，用于指定要创建协程的属性，可为 `NULL`。实际上仅有两个属性：栈大小、指向共享栈的指针（使用共享栈模式）。

@`routine`: `void* (*)(void*)` 类型的函数指针，指向协程的任务函数，即启动这个协程后要完成什么样的任务。`routine` 类型为函数指针。

@arg: void* 类型指针，传递给任务函数的参数，类似于 pthread 传递给线程的参数。

调用 co_create 将协程创建出来后，这时候它还没有启动，也即是说我们传递的 routine 函数还没有被调用。实质上，这个函数内部仅仅是分配并初始化 stCoRoutine_t 结构体、设置任务函数指针、分配一段“栈”内存，以及分配和初始化 coctx_t。为什么这里的“栈”要加个引号呢？因为这里的栈内存，无论是使用预先分配的共享栈，还是 co_create 内部单独分配的栈，其实都是调用 malloc 从进程的堆内存分配出来的。对于协程而言，这就是“栈”，而对于底层的进程（线程）来说这只不过是普通的堆内存而已。

总体上，co_create 函数内部做的工作很简单，这里就不贴出代码了。

6.2 启动协程 (Resume a coroutine)

在调用 co_create 创建协程返回成功后，便可以调用 co_resume 函数将它启动了。该函数声明如下：

```
void co_resume(stCoRoutine_t* co);
```

它的意义很明了，即启动 co 指针指向的协程。值得注意的是，为什么这个函数不叫 co_start 而是 co_resume 呢？前文已提到，libco 的协程是非对称协程，协程在让出 CPU 后要恢复执行的时候，还是要再次调用一下 co_resume 这个函数的去“启动”协程运行的。从语义上来讲，co_start 只有一次，而 co_resume 可以是暂停之后恢复启动，可以多次调用，就这么个区别。实际上，看早期关于协程的文献，讲到非对称协程，一般也都用“resume”与“yield”这两个术语。协程要获得 CPU 执行权用“resume”，而让出 CPU 执行权用“yield”，这是两个是两个不同的（不对称的）过程，因此这种机制才被称为非对称协程（asymmetric coroutines）。

所以讲到 resume 一个协程，我们一定得注意，这可能是第一次启动该协程，也可以是要准备重新运行挂起的协程。我们可以认为在 libco 里面协程只有两种状态，即 running 和 pending。当创建一个协程并调用 resume 之后便进入了 running 状态，之后协程可能通过 yield 让出 CPU，这就进入了 pending 状态。不断在这两个状态间循环往复，直到协程退出（执行的任务函数 routine 返回），如图2所示（TBD 修改状态机）。

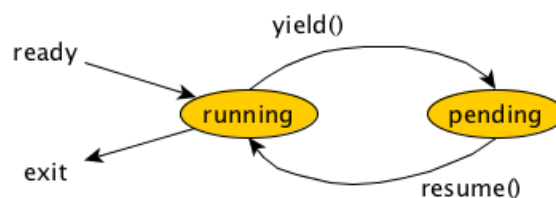


图 2: 对称协程状态转换图

需要指出的是，不同于 go 语言，这里 co_resume() 启动一个协程的含义，不是“创建一个并发任务”。进入 co_resume() 函数后发生协程的上下文切换，协程的任务函数是立即就会被执行的，而且这个执行过程不是并发的（Concurrent）。为什么不是并发的呢？因为 co_resume() 函数内部会调用 coctx_swap() 将当前协程挂起，然后就开始执行

目标协程的代码了（具体过程见下文协程切换那一节的分析）。本质上这个过程是串行的，在一个操作系统线程（进程）上发生的，甚至可以说在一颗 CPU 核上发生的（假定没有发生 CPU migration）。让我们站到 Knuth 的角度，将 coroutine 当做一种特殊的 subroutine 来看，问题会显得更清楚：A 协程调用 `co_resume(B)` 启动了 B 协程，本质上是一种特殊的过程调用关系，A 调用 B 进入了 B 过程内部，这很显然是一种串行执行的关系。那么，既然 `co_resume()` 调用后进入了被调协程执行控制流，那么 `co_resume()` 函数本身何时返回？这就要等被调协程主动让出 CPU 了。（TDB 补充图）

```
1 void co_resume(stCoRoutine_t *co) {
2     stCoRoutineEnv_t *env = co->env;
3     stCoRoutine_t *lpCurrRoutine = env->pCallStack[env->iCallStackSize-1];
4     if (!co->cStart) {
5         coctx_make(&co->ctx, (coctx_pfn_t)CoRoutineFunc, co, 0);
6         co->cStart = 1;
7     }
8     env->pCallStack[env->iCallStackSize++] = co;
9     co_swap(lpCurrRoutine, co);
10 }
```

co_resume() 函数代码实现

如果读者对 `co_resume()` 的逻辑还有疑问，不妨再看一下它的代码实现。第 5、6 行的 if 条件分支，当且仅当协程是第一次启动时才会执行到。首次启动协程过程有点特殊，需要调用 `coctx_make()` 为新协程准备 context（为了让 `co_swap()` 内能跳转到协程的任务函数），并将 `cStart` 标志变量置 1。忽略第 4~7 行首次启动协程的特殊逻辑，那么 `co_resume()` 仅有 4 行代码而已。第 3 行取当前协程控制块指针，第 8 行将待启动的协程 `co` 压入 `pCallStack` 栈，然后第 9 行就调用 `co_swap()` 切换到 `co` 指向的新协程上去执行了。前文也已经提到，`co_swap()` 不会就此返回，而是要这次 resume 的 `co` 协程主动 yield 让出 CPU 时才会返回到 `co_resume()` 中来。

值得指出的是，这里讲 `co_swap()` 不会就此返回，不是说这个函数就阻塞在这里等待 `co` 这个协程 yield 让出 CPU。实际上，后面我们将会看到，`co_swap()` 内部已经切换了 CPU 执行上下文，奔着 `co` 协程的代码路径去执行了。整个过程不是并发的，而是串行的，这一点我们已经反复强调过了。

6.3 协程的挂起（Yield to another coroutine）

在非对称协程理论，yield 与 resume 是个相对的操作。A 协程 resume 启动了 B 协程，那么只有当 B 协程执行 yield 操作时才会返回到 A 协程。在上一节剖析协程启动函数 `co_resume()` 时，也提到了该函数内部 `co_swap()` 会执行被调协程的代码。只有被调协程 yield 让出 CPU，调用者协程的 `co_swap()` 函数才能返回到原点，即返回到原来 `co_resume()` 内的位置。

在前文解释 `stCoRoutineEnv_t` 结构 `pCallStack` 这个“调用栈”的时候，我们已经简要地提到了 yield 操作的内部逻辑。在被调协程要让出 CPU 时，会将它的 `stCoRoutine_t` 从 `pCallStack` 弹出，“栈指针” `iCallStackSize` 减 1，然后 `co_swap()` 切换 CPU 上下文到原来被挂起的调用者协程恢复执行。这里“被挂起的调用者协程”，即是调用者 `co_resume()` 中切换 CPU 上下文被挂起的那个协程。下面我们来看一下 `co_yield_env()` 函数代码：

```

1 void co_yield_env(stCoRoutineEnv_t *env) {
2     stCoRoutine_t *last = env->pCallStack[env->iCallStackSize - 2];
3     stCoRoutine_t *curr = env->pCallStack[env->iCallStackSize - 1];
4     env->iCallStackSize--;
5     co_swap(curr, last);
6 }

```

co_yield_env() 函数

co_yield_env() 函数仅有 4 行代码，事实上这个还可以写得更简洁些。你可以试着把这里代码缩短至 3 行，并不会牺牲可读性。注意到这个函数为什么叫 co_yield_env 而不是 co_yield 呢？这个也很简单。我们知道 co_resume 是有明确目的对象的，而且可以通过 resume 将 CPU 交给任意协程。但 yield 则不一样，你只能 yield 给当前协程的调用者。而当前协程的调用者，即最初 resume 当前协程的协程，是保存在 stCoRoutineEnv_t 的 pCallStack 中的。因此你只能 yield 给“env”，yield 给调用者协程；而不能随意 yield 给任意协程，CPU 不是你想让给谁就能让给谁的。

事实上，libco 提供了一个 co_yield(stCoRoutine_t*) 的函数。看起来你似乎可以将 CPU 让给任意协程。实际上并非如此：

```

1 void co_yield(stCoRoutine_t *co) {
2     co_yield_env(co->env);
3 }

```

co_yield() 函数

我们知道，同一个线程上所有协程是共享一个 stCoRoutineEnv_t 结构的，因此任意协程的 co->env 指向的结构都相同。如果你调用 co_yield(co)，就以为将 CPU 让给 co 协程了，那就错了。最终通过 co_yield_env() 还是会将 CPU 让给原来启动当前协程的调用者。可能有的读者会有疑问，同一个线程上所有协程共享 stCoRoutineEnv_t，那么我 co_yield() 给其他线程上的协程呢？对不起，如果你这么做，那么你的程序就挂了。libco 的协程是不支持线程间迁移（migration）的，如果你试图这么做，程序一定会挂掉。这个 co_yield() 其实容易让人产生误解的。

再补充说明一下，协程库内虽然提供了 co_yield(stCoRoutine_t*) 函数，但是没有任何地方有调用过该函数（包括样例代码）。使用的较多的是另外一个函数——co_yield_ct()，其实本质上作用都是一样的。

6.4 协程的切换（Context switch）

前面两节讨论的 co_yield_env() 与 co_resume()，是两个完全相反的过程，但他们的核心任务却是一样的——切换 CPU 执行上下文，即完成协程的切换。在 co_resume() 中，这个切换是从当前协程切换到被调协程；而在 co_yield_env() 中，则是从当前协程切换到调用者协程。最终的上下文切换，都发生在 co_swap() 函数内。

严格来讲这里不属于协程生命周期一部分，而只是两个协程开始执行与让出 CPU 时的一个临界点。既然是切换，那就涉及到两个协程。为了表述方便，我们把当前准备让出 CPU 的协程叫做 current 协程，把即将调入执行的叫做 pending 协程。

```

1 .globl coctx_swap
2 #if !defined( __APPLE__ )

```

```

3 .type coctx_swap, @function
4 #endif
5 coctx_swap:
6
7 #if defined(__i386__)
8     leal 4(%esp), %eax //sp
9     movl 4(%esp), %esp
10    leal 32(%esp), %esp //parm a : &regs[7] + sizeof(void*)
11
12    pushl %eax //esp ->parm a
13    pushl %ebp
14    pushl %esi
15    pushl %edi
16    pushl %edx
17    pushl %ecx
18    pushl %ebx
19    pushl -4(%eax)
20
21    movl 4(%eax), %esp //parm b -> &regs[0]
22
23    popl %eax //ret func addr
24    popl %ebx
25    popl %ecx
26    popl %edx
27    popl %edi
28    popl %esi
29    popl %ebp
30    popl %esp
31    pushl %eax //set ret func addr
32
33    xorl %eax, %eax
34    ret
35 #elif defined(__x86_64__)

```

coctx_swap.S 汇编代码

这里截取的是 `coctx_swap.S` 文件中针对 x86 体系结构的一段代码，x64 下的原理跟这是一样的，代码也在这 `同一个文件` 中。从宏观角度看，这里定义了一个名为 `coctx_swap` 的函数，而且是 C 风格的函数（因为要被 C++ 代码调用）。从调用方看，我们可以将它当做一个普通的 C 函数，函数原型如下：

```
1 void coctx_swap(coctx_t* curr, coctx_t* pending) asm("coctx_swap");
```

`coctx_swap` 接受两个参数，无返回值。其中，第一个参数 `curr` 为当前协程的 `coctx_t` 结构指针，其实是个输出参数，函数调用过程中会将当前协程的 `context` 保存在这个参数指向的内存里；第二个参数 `pending`，即待切入的协程的 `coctx_t` 指针，是个输入参数，`coctx_swap` 从这里取上次保存的 `context`，恢复各寄存器的值。前面我们讲过 `coctx_t` 结构，就是用于保存各寄存器值（`context`）的。这个函数奇特之处，在于调用之前还处于第一个协程的环境，该函数返回后，则当前运行的协程就已经完全是第二个协程了。这跟 Linux 内核调度器的 `switch_to` 功能是非常相似的，只不过内核里线程的切换比这还要复杂得多。正所谓“杨百万进去，杨白劳出来”，当然，这里也可能是“杨白劳进去，杨百万出来”。

言归正题，这个函数既然是要直接操作寄存器，那当然非汇编不可了。汇编语言都

快忘光了？那也不要紧，这里用到的都是常用的指令。值得一提的是，绝大多数学校汇编教程使用的都是 Intel 的语法格式，而这里用到的是 AT&T 格式。这里我们只需知道两者的主要差别，在于操作数的顺序是反过来的，就足够了。在 AT&T 汇编指令里，如果指令有两个操作数，那么第一个是源操作数，第二个即为目的操作数。

此外，我们前面提到，这是个 C 风格的函数。什么意思呢？在 x86 平台下，多数 C 编译器会使用一种固定的方法来处理函数的参数与返回值。函数的参数使用栈传递，且约定了参数顺序，如图 3 所示。在调用函数之前，编译器会将函数参数以反向顺序压栈，如图 3 中函数有三个参数，那么参数 3 首先 push 进栈，随后是参数 2，最后参数 1。准备好参数后，调用 CALL 指令时 CPU 自动将 IP 寄存器（函数返回地址）push 进栈，因此在进入被调函数之后，便形成了如图 3 的栈格局。函数调用结束前，则使用 EAX 寄存器传递返回值（如果 32 位够用的话），64 位则使用 EDX:EAX，如果是浮点值则使用 FPU ST(0) 寄存器传递。

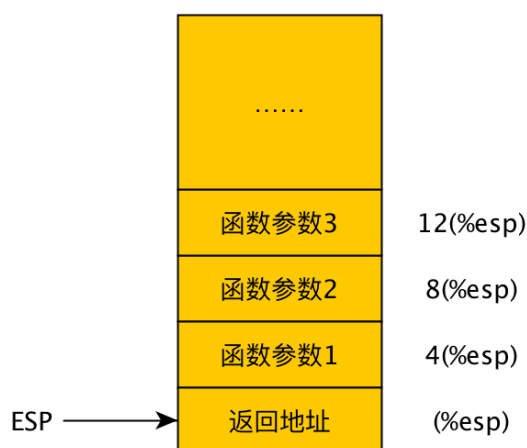


图 3: C 语言函数传递参数示意图

在复习过这些汇编语言知识后，我们再来看 `coctx_swap` 函数。它有两个参数，那么进入函数体后，用 `4(%esp)` 便可以取到第一个参数（当前协程 `context` 指针），`8(%esp)` 可以取到第二个参数（待切入运行协程的 `context` 指针）。当前栈顶的内容，`(%esp)` 则保存了 `coctx_swap` 的返回地址。搞清楚栈数据布局是理解 `coctx_swap` 函数的关键，接下来分析 `coctx_swap` 的每条指令，都需要时刻明白当前的栈在哪里，栈内数据是怎样一个分布。我们把 `coctx_swap` 分为两部分，以第 21 行那条 `MOVL` 指令为界。第一部分是用于保存 `current` 协程的各个寄存器，第二部分则是恢复 `pending` 协程的寄存器。接下来我们逐行进行分析。

第 8 行：LEA 指令即 Load Effective Address 的缩写。这条指令把 `4(%esp)` 有效地址保存到 `eax` 寄存器，可以认为是将当前的栈顶地址保存下来（实际保存的地址比栈顶还要高 4 字节，为了方便我们就称之为栈顶）。为什么要保存栈指针呢，因为紧接着就要进行栈切换了。

第 9~10 行：看到第 9 行，回忆我们前面讲过的 C 函数参数在栈中的位置，此时 `4(%esp)` 内正是指向 `current` 协程 `coctx_t` 的指针，这里把它塞到 `esp` 寄存器。接下来第 10 行又将 `coctx_t` 指针指向的地址加上 32 个字节的内存位置加载到 `esp` 中。经过这么

一倒腾，`esp` 寄存器实际上指向了当前协程 `coctx_t` 结构的 `ss size` 成员位置，在它之下有个名为 `regs` 的数组，刚好是用来保存 8 个寄存器值的。注意这是第一次栈切换，不过是临时性的，目的只是方便接下来使用 `push` 指令保存各寄存器值。

第 12 行：`eax` 寄存器内容压栈。更准确的讲，是将 `eax` 寄存器保存到了 `coctx_t->regs[7]` 的位置。注意到在第 8 行 `eax` 寄存器已经保存了原栈顶的地址，所以这句实际上是将当前协程栈顶保存起来，以备下次调度回来时恢复栈地址。

第 13~18 行：保存各通用寄存器的值，到 `coctx_t` 结构的 `regs[1]~regs[6]` 的位置。

第 19 行：这一行又有点意思了。`eax` 的值，从第 8 行之后就变变过，那么 `-4(%eax)` 实际上是指向原来 `coctx_swap` 刚进来时的栈顶，我们讲过栈顶的值是 `call` 指令自动压入的函数返回地址。这句实际上就是将 `coctx_swap` 的返回地址给保存起来了，放在 `coctx_t->regs[0]` 的位置。

第 21 行：至此，`current` 协程的各重要寄存器都已保存完成了，开始可以放心地交班给 `pending` 协程了。接下来我们需要将 `pending` 协程调度起来运行，就需要为它恢复 `context`——恢复各通用寄存器的值以及栈指针。因此这一行将栈指针切到 `pending` 协程的 `coctx_t` 结构体开始，即 `regs[0]` 的位置，为恢复寄存器值做好了准备。

第 23 行：弹出 `regs[0]` 的值到 `eax` 寄存器。`regs[0]` 正该协程上次被切换出去时在第 19 行保存的值，即 `coctx_swap` 的返回地址。

第 24~29 行：从 `regs[1]~regs[6]` 恢复各寄存器的值（与之相应的是前面第 13~18 行的压栈操作）。

第 30 行：将 `pending` 协程上次切换出去时的栈指针恢复（与之对应的是第 12 行压栈操作）。请思考一下，栈内容已经完全恢复了吗？注意到第 8 行我们讲过，当时保存的“栈顶”比真正的栈顶差了一个 4 字节的偏移。而这 4 字节真正栈顶的内容，正是 `coctx_swap` 的返回地址。如果此时程序就执行 `ret` 指令返回，那程序就不知道会跑到哪去了。

第 31 行：为了程序能正确地返回原来的 `coctx_swap` 调用的地方，将 `eax` 内容（第 19 行保存至 `regs[7]`，第 23 行取出来到 `eax`）压栈。

第 33~34 行：清零 `eax` 寄存器，执行返回指令。

至此，对 32 位平台的 `coctx_swap` 分析就到此结束了。细心的读者会发现一共切换了 3 次栈指针，尽管作者加了一些注释，但这段代码对于很多人来说恐怕还是难以理解的。本文仅仅分析了 32 位的情况，`x64` 下的代码逻辑是类似的，不过涉及的寄存器更多一些，而且函数调用时参数传递有所区别。有兴趣的读者可以自行分析下，此外，还可以结合 `glibc` 的 `ucontext` 源码对比分析一下。`ucontext` 也提供了支持用户级线程的接口，也有类似功能的 `swapcontext()` 函数，那里的汇编代码比较容易读懂些，不过运行效率比较低。

6.5 协程的退出

这里讲的退出，有别于协程的挂起，是指协程的任务函数执行结束后发生的过程。更简单的说，就是协程任务函数内执行了 `return` 语句，结束了它的生命周期。这在某些场景是有用的。同协程挂起一样，协程退出时也应将 `CPU` 控制权交给它的调用者，这

也是调用 `co_yield_env()` 函数来完成的。这个机制很简单，限于篇幅，具体代码就不贴出来了。

值得注意的是，我们调用 `co_create()`、`co_resume()` 启动协程执行一次性任务，当任务结束后要记得调用 `co_free()` 或 `co_release()` 销毁这个临时性的协程，否则将引起内存泄漏。

7 事件驱动与协程调度

7.1 协程的“阻塞”与线程的“非阻塞”

我们已经分析了 libco 的协程从创建到启动，挂起、启动以及最后退出的过程。同时，我们也看到，一个线程上的所有协程本质上是如何串行执行的。让我们暂时回到 3.1 节的例子。在 `Producer` 协程函数内我们会看到调用 `poll` 函数等待 1 秒，`Consumer` 中也会看到调用 `co_cond_timedwait` 函数等待生产者信号。注意，从协程的角度看，这些等待看起来都是同步的（synchronous），阻塞的（blocking），但从底层线程的角度来看，则是非阻塞的（non-blocking）。

在 3.1 节例子中我们也讲过，这跟 `pthread` 实现的原理是一样的。在 `pthread` 实现的消费者中，你可能用 `pthread_cond_timedwait` 函数去同步等待生产者的信号；在消费者中，你可能用 `poll` 或 `sleep` 函数去定时等待。从线程的角度看，这些函数都会让当前线程阻塞；但从内核的角度看，它本身并没有阻塞，内核可能要继续忙着调度别的线程运行。那么这里协程也是一样的道理，从协程的角度看，当前的程序阻塞了；但从它底下的线程来看，自己可能正忙着执行别的协程函数。在这个例子中，当 `Consumer` 协程调用 `co_cond_timedwait` 函数“阻塞”后，线程可能已经将 `Producer` 调度恢复执行，反之亦然。那么这个负责协程“调度”的线程在哪呢？它即是运行协程本身的这个线程。

7.2 主协程与协程的“调度”

还记得前文提过的“主协程”的概念吗？我们再次把它搬出来，这对我们理解协程的“阻塞”与“调度”可能更有帮助。我们讲过，libco 程序都有一个主协程，即程序里首次调用 `co_create()` 显式创建第一个协程的协程。在 3.1 节例子中，即为 `main` 函数里调用 `co_eventloop()` 的这个协程。当 `Consumer` 或 `Producer` 阻塞后，CPU 将 `yield` 给主协程，此时主协程在干什么呢？主协程在 `co_eventloop()` 函数里头忙活。这个 `co_eventloop()` 即“调度器”的核心所在。

需要补充说明的是，这里讲的“调度器”，严格意义上算不上真正的调度器，只是为了表述的方便。libco 的协程机制是非对称的，没有什么调度算法。在执行 `yield` 时，当前协程只能将控制权交给调用者协程，没有任何可调度的余地。`resume` 灵活性稍强一点，不过也还算不得调度。如果非要说有什么“调度算法”的话，那就只能说是“基于 `epoll/kqueue` 事件驱动”的调度算法。“调度器”就是 `epoll/kqueue` 的事件循环。

我们知道，在 `go` 语言中，用户只需使用同步阻塞式的编程接口即可开发出高性能的服务器，`epoll/kqueue` 这样的 I/O 事件通知机制（I/O event notification mechanism）完

全被隐藏了起来。在 libco 里也是一样的，你只需要使用普通 C 库函数 read()、write() 等等同步地读写数据就好了。那么 epoll 藏在哪呢？就藏在主协程的 co_eventloop() 中。协程的调度与事件驱动是紧紧联系在一起的，因此与其说 libco 是一个协程库，还不如说它是一个网络库。在后台服务器程序中，一切逻辑都是围绕网络 I/O 转的，libco 这样的设计自有它的合理性。

7.3 stCoEpoll_t 结构与定时器

在分析 stCoRoutineEnv_t 结构（代码清单5）的时候，还有一个 stCoEpoll_t 类型的 pEpoll 指针成员没有讲到。作为 stCoRoutineEnv_t 的成员，这个结构也是一个全局性的资源，被同一个线程上所有协程共享。从命名也看得出来，stCoEpoll_t 是跟 epoll 的事件循环相关的。现在我们看一下它的内部字段：

```
1 struct stCoEpoll_t {
2     int iEpollFd;
3     static const int _EPOLL_SIZE = 1024 * 10;
4     struct stTimeout_t *pTimeout;
5     struct stTimeoutItemLink_t *pstTimeoutList;
6     struct stTimeoutItemLink_t *pstActiveList;
7     co_epoll_res *result;
8 };
```

stCoEpoll_t 结构

@iEpollFd: 显然是 epoll 实例的文件描述符。

@_EPOLL_SIZE: 值为 10240 的整型常量。作为 epoll_wait() 系统调用的第三个参数，即一次 epoll_wait 最多返回的就绪事件个数。

@pTimeout: 类型为 stTimeout_t 的结构体指针。该结构实际上是一个时间轮 (Timing wheel) 定时器，只是命名比较怪，让人摸不着头脑。

@pstTimeoutList: 指向 stTimeoutItemLink_t 类型的结构体指针。该指针实际上是一个链表头。链表用于临时存放超时事件的 item。

@pstActiveList: 指向 stTimeoutItemLink_t 类型的结构体指针。也是指向一个链表。该链表用于存放 epoll_wait 得到的就绪事件和定时器超时事件。

@result: 对 epoll_wait() 第二个参数的封装，即一次 epoll_wait 得到的结果集。

我们知道，定时器是事件驱动模型的网络框架一个必不可少的功能。网络 I/O 的超时，定时任务，包括定时等待 (poll 或 timedwait) 都依赖于此。一般而言，使用定时功能时，我们首先向定时器中注册一个定时事件 (Timer Event)，在注册定时事件时需要指定这个事件在未来的触发时间。在到了触发时间点后，我们会收到定时器的通知。

网络框架里的定时器可以看做由两部分组成，第一部分是保存已注册 timer events 的数据结构，第二部分则是定时通知机制。保存已注册的 timer events，一般选用红黑树，比如 nginx；另外一种常见的数据结构便是时间轮，libco 就使用了这种结构。当然你也可以直接用链表来实现，只是时间复杂度比较高，在定时任务很多时会很容易成为框架的性能瓶颈。

定时器的第二部分，高精度的定时（精确到微秒级）通知机制，一般使用 getitimer/setitimer 这类接口，需要处理信号，是个比较麻烦的事。不过对一般的应

用而言，精确到毫秒就够了。精度放宽到毫秒级时，可以顺便使用 `epoll/kqueue` 这样的系统调用来完成定时通知；这样一来，网络 I/O 事件通知与定时事件通知的逻辑就能统一起来了。笔者之前实现过的一个基于 `libcurl` 的异步 HTTP client，其中的定时器功能就是用 `epoll` 配合红黑树实现的。`libco` 内部也直接使用了 `epoll` 来进行定时，不同的只是保存 timer events 用的是时间轮而已。

使用 `epoll` 加时间轮的实现定时器的算法如下：

Step 1 [`epoll_wait`] 调用 `epoll_wait()` 等待 I/O 就绪事件，最大等待时长设置为 1 毫秒（即 `epoll_wait()` 的第 4 个参数）。

Step 2 [处理 I/O 就绪事件] 循环处理 `epoll_wait()` 得到的 I/O 就绪文件描述符。

Step 3 [从时间轮取超时事件] 从时间轮取超时事件，放到 timeout 队列。

Step 4 [处理超时事件] 如果 Step 3 取到的超时事件不为空，那么循环处理 timeout 队列中的定时任务。否则跳转到 Step 1 继续事件循环。

Step 5 [继续循环] 跳转到 Step 1，继续事件循环。

7.4 挂起协程与恢复的执行

在前文的第 6.2 与第 6.3 小节，我们仔细地分析了协程的 `resume` 与 `yield` 过程。那么协程究竟在什么时候需要 `yield` 让出 CPU，又在什么时候恢复执行呢？

先来看 `yield`，实际上在 `libco` 中共有 3 种调用 `yield` 的场景：

1. 用户程序中主动调用 `co_yield_ct()`；
2. 程序调用了 `poll()` 或 `co_cond_timedwait()` 陷入“阻塞”等待；
3. 程序调用了 `connect()`, `read()`, `write()`, `recv()`, `send()` 等系统调用陷入“阻塞”等待。

相应地，重新 `resume` 启动一个协程也有 3 种情况：

1. 对应用户程序主动 `yield` 的情况，这种情况也有赖于用户程序主动将协程 `co_resume()` 起来；

2. `poll()` 的目标文件描述符事件就绪或超时，`co_cond_timedwait()` 等到了其他协程的 `co_cond_signal()` 通知信号或等待超时；

3. `read()`, `write()` 等 I/O 接口成功读到或写入数据，或者读写超时。

在第一种情况下，即用户主动 `yield` 和 `resume` 协程，相当于 `libco` 的使用者承担了部分的协程“调度”工作。这种情况其实也很常见，在 `libco` 源码包的 `example_echosvr.cpp` 例子中就有。这也是服务端使用 `libco` 的典型模型，属于手动“调度”协程的例子。

第二种情况，前面第 3.1 节中的生产者消费者就是个典型的例子。在那个例子中我们看不到用户程序主动调用 `yield`，也只有在最初启动协程时调用了 `resume`。生产者和消费者协程是在哪里切换的呢？在 `poll()` 与 `co_cond_timedwait()` 函数中。首先来看消费者。当消费者协程首先启动时，它会发现任务队列是空的，于是调用 `co_cond_timedwait()` 在条件变量 `cond` 上“阻塞”等待。同操作系统线程的条件等待原理一样，这里条件变量 `stCoCond_t` 类型内部也有一个“等待队列”。`co_cond_timedwait()` 函数内部会将当前协程挂入条件变量的等待队列上，并设置一个回调函数，该回调函数是用于未来“唤醒”当前协程的（即 `resume` 挂起的协程）。此外，如果 `wait` 的 `timeout` 参数大于 0 的话，还要向当前执行环境的定时器上注册一个定时事件（即挂到时间轮上）。在这个例子中，

消费者协程 `co_cond_timedwait` 的 `timeout` 参数为 -1，即 indefinitely 地等待下去，直到等到生产者向条件变量发出 `signal` 信号。

然后我们再来看生产者。当生产者协程启动后，它会向任务队列里投放一个任务并调用 `co_cond_signal()` 通知消费者，然后再调用 `poll()` 在原地“阻塞”等待 1000 毫秒。这里 `co_cond_signal` 函数内部其实也简单，就是将条件变量的等待队列里的协程拿出来，然后挂到当前执行环境的 `pstActiveList`（见 7.3 节 `stCoEpoll_t` 结构）。`co_cond_signal` 函数并没有立即 `resume` 条件变量上的等待协程，毕竟这还不到交出 CPU 的时机。那么什么时候交出 CPU 控制权，什么时候 `resume` 消费者协程呢？继续往下看，生产者在向消费者发出“信号”之后，紧接着便调用 `poll()` 进入了“阻塞”等待，等待 1 秒钟。这个 `poll` 函数内部实际上做了两件事。首先，将自己作为一个定时事件注册到当前执行环境的定时器，注册的时候设置了 1 秒钟的超时时间和一个回调函数（仍是一个用于未来“唤醒”自己的回调）。然后，就调用 `co_yield_env()` 将 CPU 让给主协程了。

现在，CPU 控制权又回到了主协程手中。主协程此时要干什么呢？我们已经讲过，主协程就是事件循环 `co_eventloop()` 函数。在 `co_eventloop()` 中，主协程周而复始地调用 `epoll_wait()`，当有就绪的 I/O 事件就处理 I/O 事件，当定时器上有超时的事件就处理超时事件，`pstActiveList` 队列中已有活跃事件就处理活跃事件。这里所谓的“处理事件”，其实就是调用其他工作协程注册的各种回调函数而已。那么前面我们讲过，消费者协程和生产者协程的回调函数都是“唤醒”自己而已。工作协程调用 `co_cond_timedwait()` 或 `poll()` 陷入“阻塞”等待，本质上即是通过 `co_yield_env` 函数让出了 CPU；而主协程则负责在事件循环中“唤醒”这些“阻塞”的协程，所谓“唤醒”操作即调用工作协程注册的回调函数，这些回调内部使用 `co_resume()` 重新恢复挂起的工作协程。

最后，协程 `yield` 和 `resume` 的第三种情况，即调用 `read()`, `write()` 等 I/O 操作而陷入“阻塞”和最后又恢复执行的过程。这种情况跟第二种过程基本相似。需要注意的是，这里的“阻塞”依然是用户态实现的过程。我们知道，`libco` 的协程是在底层线程上串行执行的。如果调用 `read` 或 `write` 等系统调用陷入真正的阻塞（让当前线程被内核挂起）的话，那么不光当前协程被挂起了，其他协程也得不到执行的机会。因此，如果工作协程陷入真正的内核态阻塞，那么 `libco` 程序就会完全停止运转，后果是很严重的。

为了避免陷入内核态阻塞，我们必须得依靠内核提供的非阻塞 I/O 机制，将 `socket` 文件描述符设置为 `non-blocking` 的。为了让 `libco` 的使用者更方便，我们还得将这种 `non-blocking` 的过程给封装起来，伪装成“同步阻塞式”的调用（跟 `co_cond_timedwait()` 一样）。事实上，`go` 语言就是这么做的。而 `libco` 则将这个过程伪装得更加彻底，更加具有欺骗性。它通过 `dlsym` 机制 `hook` 了各种网络 I/O 相关的系统调用，使得用户可以用“同步”的方式直接使用诸如 `read()`、`write()` 和 `connect()` 等系统调用。因此，我们会看到 3.1 节那里的生产者消费者协程任务函数里第一句就调用了一个 `co_enable_hook_sys()` 的函数。调用了 `co_enable_hook_sys` 函数才会开启 `hook` 系统调用功能，并且需要事先将要读写的文件描述符设置为 `non-blocking` 属性，否则，工作协程就可能陷入真正的内核态阻塞，这一点在应用中要特别加以注意。

以 `read()` 为例，让我们再来分析一下这些“伪装”成同步阻塞式系统调用的内部原理。首先，假如程序 `accept` 了一个新连接，那么首先我们将这个连接的 `socket` 文件描述符设置为非阻塞模式，然后启动一个工作协程去处理这个连接。工作协程调用 `read()`

试图从该新连接上读取数据。这时候由于系统 `read()` 函数已经被 hook，所以实际上会调用到 libco 内部准备好的 `read()` 函数。这个函数内部实际上做了 4 件事：第一步将当前协程注册到定时器上，用于将来处理 `read()` 函数的读超时。第二步，调用 `epoll_ctl()` 将自己注册到当前执行环境的 `epoll` 实例上。这两步注册过程都需要指定一个回调函数，将来用于“唤醒”当前协程。第三步，调用 `co_yield_env` 函数让出 CPU。第四步要等到该协程被主协程重新“唤醒”后才能继续。如果主协程 `epoll_wait()` 得知 `read` 操作的文件描述符可读，则会执行原 `read` 协程注册的回调将它唤醒（超时后同理，不过还要设置超时标志）。工作协程被唤醒后，在调用原 `glibc` 内被 hook 替换掉的、真正的 `read()` 系统调用。这时候如果是正常 `epoll_wait` 得知文件描述符 I/O 就绪就会读到数据，如果是超时就会返回 -1。总之，在外部使用者看来，这个 `read()` 就跟阻塞式的系统调用表现出几乎完全一致的行为了。

7.5 主协程事件循环源码分析

前文已经多次提到过主协程事件循环，主协程是如何“调度”工作协程运行的。最后，让我们再来分析下它的代码（为了节省篇幅，在不妨碍我们理解其工作原理的前提下，已经略去了数行不相关的代码）。

```
1 void co_eventloop(stCoEpoll_t *ctx, pfn_co_eventloop_t pfn, void *arg)
2 {
3     co_epoll_res *result = ctx->result;
4
5     for (;;) {
6         int ret= co_epoll_wait(ctx->iEpollFd, result, stCoEpoll_t::_EPOLL_SIZE, 1);
7
8         stTimeoutItemLink_t *active = (ctx->pstActiveList);
9         stTimeoutItemLink_t *timeout = (ctx->pstTimeoutList);
10        memset(timeout, 0, sizeof(stTimeoutItemLink_t));
11
12        for (int i=0; i<ret; i++) {
13            stTimeoutItem_t *item = (stTimeoutItem_t*)result->events[i].data.ptr;
14            if (item->pfnPrepare) {
15                item->pfnPrepare(item, result->events[i], active);
16            } else {
17                AddTail(active, item);
18            }
19        }
20
21        unsigned long long now = GetTickMS();
22        TakeAllTimeout(ctx->pTimeout, now, timeout);
23
24        stTimeoutItem_t *lp = timeout->head;
25        while (lp) {
26            lp->bTimeout = true;
27            lp = lp->pNext;
28        }
29
30        Join<stTimeoutItem_t, stTimeoutItemLink_t>(active, timeout);
31
32        lp = active->head;
33        while (lp) {
34            PopHead<stTimeoutItem_t, stTimeoutItemLink_t>(active);
```

```

35         if (lp->pfnProcess) {
36             lp->pfnProcess(lp);
37         }
38         lp = active->head;
39     }
40 }
41 }

```

stCoEpoll_t 结构

第 6 行：调用 `epoll_wait()` 等待 I/O 就绪事件，为了配合时间轮工作，这里的 `timeout` 设置为 1 毫秒。

第 8~10 行：`active` 指针指向当前执行环境的 `pstActiveList` 队列，注意这里面可能已经有“活跃”的待处理事件。`timeout` 指针指向 `pstTimeoutList` 列表，其实这个 `timeout` 完全是个临时性的链表，`pstTimeoutList` 永远为空。

第 12~19 行：处理就绪的文件描述符。如果用户设置了预处理回调，则调用 `pfnPrepare` 做预处理（15 行）；否则直接将就绪事件 `item` 加入 `active` 队列。实际上，`pfnPrepare()` 预处理函数内部也是会将就绪 `item` 加入 `active` 队列，最终都是加入到 `active` 队列等到 32~40 行统一处理。

第 21~22 行：从时间轮上取出已超时的事件，放到 `timeout` 队列。

第 24~28 行：遍历 `timeout` 队列，设置事件已超时标志（`bTimeout` 设为 `true`）。

第 30 行：将 `timeout` 队列中事件合并到 `active` 队列。

第 32~40 行：遍历 `active` 队列，调用工作协程设置的 `pfnProcess()` 回调函数 `resume` 挂起的工作协程，处理对应的 I/O 或超时事件。

这就是主协程的事件循环工作过程，我们看到它周而复始地 `epoll_wait()`，唤醒挂起的工作协程去处理定时器与 I/O 事件。这里的逻辑看起来跟所有基于 `epoll` 实现的事件驱动网络框架并没有什么特别之处，更没有涉及到任何协程调度算法，由此也可以看到 `libco` 其实是一个很典型的非对称协程机制。或许，从 `call/return` 的角度出发，而不是 `resume/yield` 去理解这种协程的运行机理，反而会有更深的理解吧。

8 性能评测

CPU: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 4 核

Memory: 4GB

OS: Linux-3.16.0

8.1 echo 实验结果

TBD

8.2 协程切换开销评测

TBD: 精确度量平均每次协程上下文切换的开销（耗时纳秒数与 CPU 时钟周期数），与 `boost`、`libtask` 的对比。

9 小结

最后，通过本文的分析，希望读者应该能真正 **libco** 的运行机理。对于喜欢动手实战的同学来说，如果你打算在未来的实际项目中使用它的话，能够做到心中有数，能绕过一些明显的坑，万一遇到问题时也能快速地解决。对于那些希望更深入理解各种协程工作原理的同学来说，希望本文的分析能起到抛砖引玉的作用，以后再学习 **boost** 协程或 **golang** 协程原理也能更快地抓住要点。

由于时间仓促，本文的分析与归纳思路不是很清晰，尤其对于初次接触 **libco** 的理解起来可能更加吃力。建议亲自动手，下载源码编译，阅读例子代码并运行一下，必要时直接阅读源代码可能会更有帮助。