# 概述

# 原理

与 Python 不同，C/C++ 语言本身是不能天然支持协程的。现有的 C++ 协程库均基于两种方案：利用汇编代码控制协程上下文的切换，以及利用操作系统提供的 API 来实现协程上下文切换。典型的例如：

libco，Boost.context：基于汇编代码的上下文切换

phxrpc：基于 ucontext/Boost.context 的上下文切换

libmill：基于 setjump/longjump 的协程切换

一般而言，基于汇编的上下文切换要比采用系统调用的切换更加高效，这也是为什么 phxrpc 在使用 Boost.context 时要比使用 ucontext 性能更好的原因。关于 phxrpc 和 libmill 具体的协程实现方式，以后有时间再详细介绍。

# 框架

## libco

参考：

<http://kaiyuan.me/2017/07/10/libco/>

### 概述

腾讯开源的一套用于支撑微信后端海量并发的 C++ 协程库 libco。相比于其他的 C++ 协程实现，libco 通过仅有的几个函数接口 co\_create，co\_resume 以及 co\_yield 配合 co\_poll，来支持同步或者异步的写法，从而实现对现有逻辑非侵入式的异步化改造。今天我们来详细分析一下 libco c++ 协程的实现。

### 原理

#### libco 协程的创建和切换

在介绍 coroutine 的创建之前，我们先来熟悉一下 libco 中用来表示一个 coroutine 的数据结构，即定义在 co\_routine\_inner.h 中的 stCoRoutine\_t:

struct stCoRoutine\_t

{

stCoRoutineEnv\_t \*env; // 协程运行环境

pfn\_co\_routine\_t pfn; // 协程执行的逻辑函数

void \*arg; // 函数参数

coctx\_t ctx; // 保存协程的下文环境

...

char cEnableSysHook; // 是否运行系统 hook，即非侵入式逻辑

char cIsShareStack; // 是否在共享栈模式

void \*pvEnv;

stStackMem\_t\* stack\_mem; // 协程运行时的栈空间

char\* stack\_sp; // 用来保存协程运行时的栈空间

unsigned int save\_size;

char\* save\_buffer;

};

我们暂时只需要了解表示协程的最简单的几个参数，例如协程运行环境，协程的上下文环境，协程运行的函数以及运行时栈空间。后面的 stack\_sp，save\_size 和 save\_buffer 与 libco 共享栈模式相关，有关共享栈的内容我们后续再说。

#### 协程创建和运行

由于多个协程运行于一个线程内部的，因此当创建线程中的第一个协程时，需要初始化该协程所在的环境 stCoRoutineEnv\_t，这个环境是线程用来管理协程的，通过该环境，线程可以得知当前一共创建了多少个协程，当前正在运行哪一个协程，当前应当如何调度协程：

struct stCoRoutineEnv\_t

{

stCoRoutine\_t \*pCallStack[ 128 ]; // 记录当前创建的协程

int iCallStackSize; // 记录当前一共创建了多少个协程

stCoEpoll\_t \*pEpoll; // 该线程的协程调度器

// 在使用共享栈模式拷贝栈内存时记录相应的 coroutine

stCoRoutine\_t\* pending\_co;

stCoRoutine\_t\* occupy\_co;

};

上述代码表明 libco 允许一个线程内最多创建 128 个协程，其中 pCallStack[iCallStackSize-1] 也就是栈顶的协程表示当前正在运行的协程。当调用函数 co\_create 时，首先检查当前线程中的 coroutine env 结构是否创建。这里 libco 对于每个线程内的 stCoRoutineEnv\_t 并没有使用 thread-local 的方式（例如gcc 内置的 \_\_thread，phxrpc采用这种方式）来管理，而是预先定义了一个大的数组，并通过对应的 PID 来获取其协程环境。：

static stCoRoutineEnv\_t\* g\_arrCoEnvPerThread[204800]

stCoRoutineEnv\_t \*co\_get\_curr\_thread\_env()

{

return g\_arrCoEnvPerThread[ GetPid() ];

}

初始化 stCoRoutineEnv\_t 时主要完成以下几步：

1、为 stCoRoutineEnv\_t 申请空间并且进行初始化，设置协程调度器 pEpoll。

2、创建一个空的 coroutine，初始化其上下文环境( 有关 coctx 在后文详细介绍 )，将其加入到该线程的协程环境中进行管理，并且设置其为 main coroutine。这个 main coroutine 用来运行该线程主逻辑。

当初始化完成协程环境之后，调用函数 co\_create\_env 来创建具体的协程，该函数初始化一个协程结构 stCoRoutine\_t，设置该结构中的各项字段，例如运行的函数 pfn，运行时的栈地址等等。需要说明的就是，如果使用了非共享栈模式，则需要为该协程单独申请栈空间，否则从共享栈中申请空间。栈空间表示如下：

struct stStackMem\_t

{

stCoRoutine\_t\* occupy\_co; // 使用该栈的协程

int stack\_size; // 栈大小

char\* stack\_bp; // 栈底指针，栈从高地址向低地址增长

char\* stack\_buffer; // 栈底

};

使用 co\_create 创建完一个协程之后，将调用 co\_resume 来将该协程激活运行：

void co\_resume( stCoRoutine\_t \*co )

{

stCoRoutineEnv\_t \*env = co->env;

// 获取当前正在运行的协程的结构

stCoRoutine\_t \*lpCurrRoutine = env->pCallStack[ env->iCallStackSize - 1 ];

if( !co->cStart )

{

// 为将要运行的 co 布置上下文环境

coctx\_make( &co->ctx,(coctx\_pfn\_t)CoRoutineFunc,co,0 );

co->cStart = 1;

}

env->pCallStack[ env->iCallStackSize++ ] = co;

// 设置co为运行的线程

co\_swap( lpCurrRoutine, co );

}

函数 co\_swap 的作用类似于 Unix 提供的函数 swapcontext：将当前正在运行的 coroutine 的上下文以及状态保存到结构 lpCurrRoutine 中，并且将 co 设置成为要运行的协程，从而实现协程的切换。co\_swap 具体完成三项工作：

1、记录当前协程 curr 的运行栈的栈顶指针，通过 char c; curr\_stack\_sp=&c 实现，当下次切换回 curr时，可以从该栈顶指针指向的位置继续，执行完 curr 后可以顺利释放该栈。

2、处理共享栈相关的操作，并且调用函数 coctx\_swap 来完成上下文环境的切换。注意执行完 coctx\_swap 之后，执行流程将跳到新的 coroutine 也就是 pending\_co 中运行，后续的代码需要等下次切换回 curr 时才会执行。

3、当下次切换回 curr 时，处理共享栈相关的操作。

对应于 co\_resume 函数，协程主动让出执行权则调用 co\_yield 函数。co\_yield 函数调用了 co\_yield\_env，将当前协程与当前线程中记录的其他协程进行切换：

void co\_yield\_env( stCoRoutineEnv\_t \*env )

{

stCoRoutine\_t \*last = env->pCallStack[ env->iCallStackSize - 2 ];

stCoRoutine\_t \*curr = env->pCallStack[ env->iCallStackSize - 1 ];

env->iCallStackSize--;

co\_swap( curr, last);

}

前面我们已经提到过，pCallStack 栈顶所指向的即为当前正在运行的协程所对应的结构，因此该函数将 curr 取出来，并将当前正运行的协程上下文保存到该结构上，并切换到协程 last 上执行。接下来我们以 32-bit 的系统为例来分析 libco 是如何实现协程运行环境的切换的。

#### 协程上下文的创建和切换

libco 使用结构 struct coctx\_t 来表示一个协程的上下文环境：

struct coctx\_t

{

#if defined(\_\_i386\_\_)

void \*regs[ 8 ];

#else

void \*regs[ 14 ];

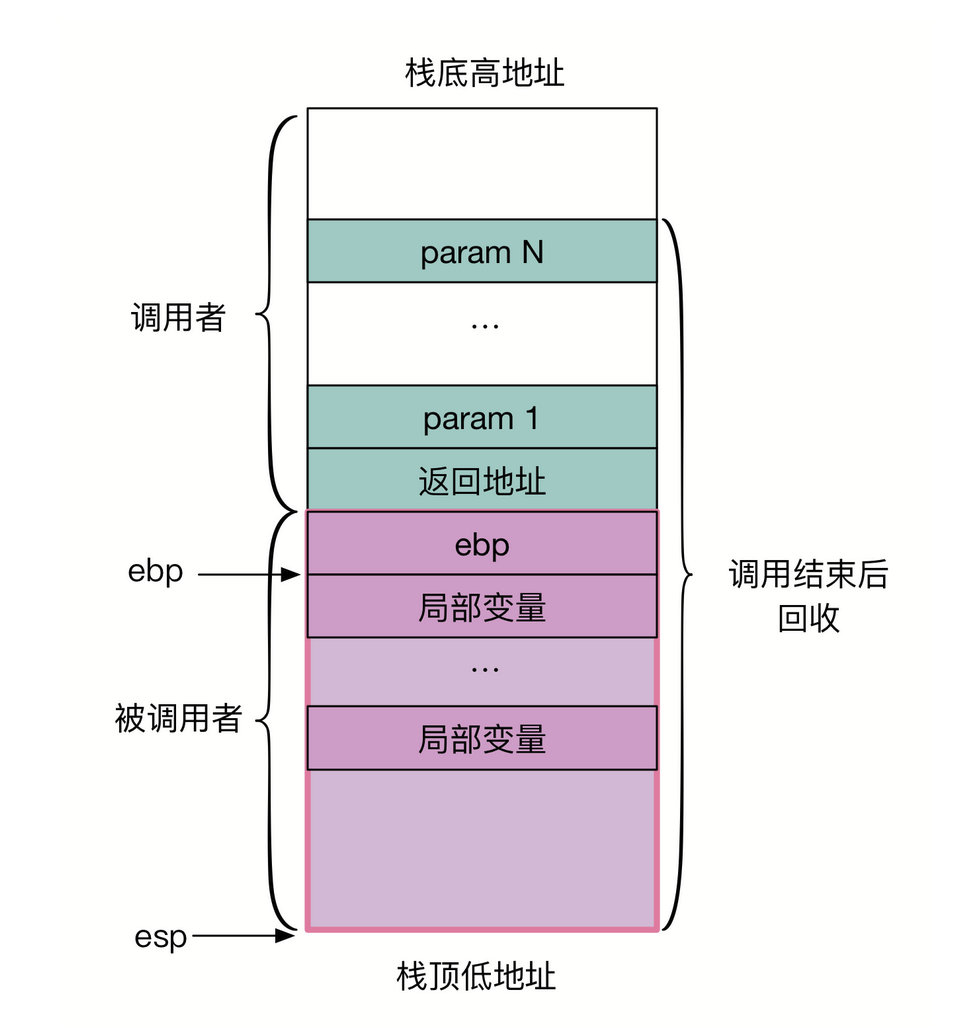
#endif

size\_t ss\_size;

char \*ss\_sp;

};

可以看到，在 i386 的架构下，需要保存 8 个寄存器信息，以及栈指针和栈大小，究竟这 8 个寄存器如何保存，又是如何使用，需要配合后续的 coctx\_swap 来理解。我们首先来回顾一下 Unix-like 系统的 stack frame layout，如果不能理解这个，那么剩下的内容就不必看了。

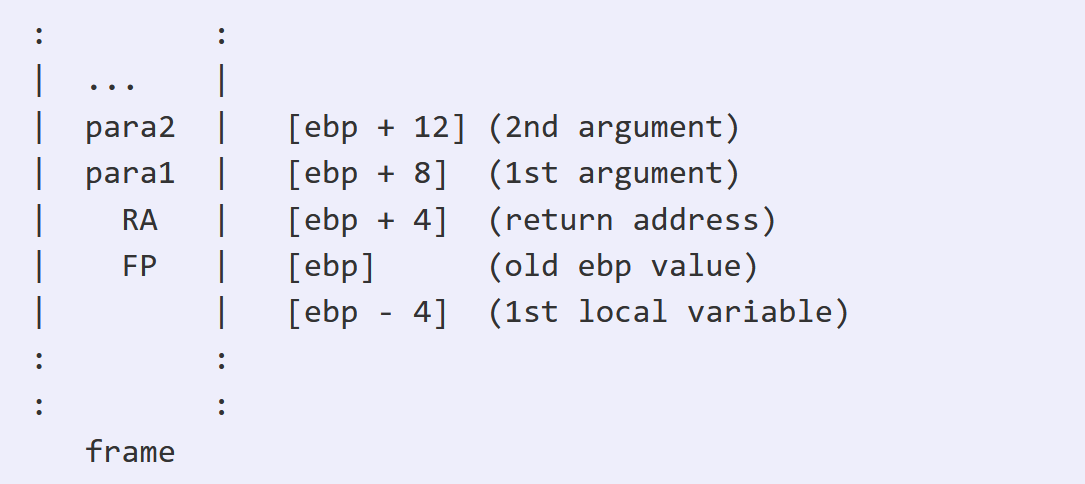


结合上图，我们需要知道关键的几点：

1、函数调用栈是调用者和被调用者共同负责布置的。Caller 将其参数从右向左反向压栈，再将调用后的返回地址压栈，然后将执行流程交给 Callee。

2、典型的编译器会将 Callee 函数汇编成为以 push %ebp; move %ebp, %esp; sub $esp N; 这种形式开头的汇编代码。这几句代码主要目的是为了方便 Callee 利用 ebp 来访问调用者提供的参数以及自身的局部变量（如下图）。如果你写过操作系统或者编译器代码，这一点无需多言。如果你没写过，这里提供一篇 [x86 Disassembly Stack Frames](https://en.m.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames) 供仔细理解。

3、当调用过程完成清除了局部变量以后，会执行 pop %ebp; ret，这样指令会跳转到 RA 也就是返回地址上面执行。这一点也是实现协程切换的关键：我们只需要将指定协程的函数指针地址保存到 RA 中，当调用完 coctx\_swap 之后，会自动跳转到该协程的函数起始地址开始运行。



了解了这些，我们就来看一下协程上下文环境的初始化函数 coctx\_make：

int coctx\_make( coctx\_t \*ctx, coctx\_pfn\_t pfn, const void \*s, const void \*s1 )

{

char \*sp = ctx->ss\_sp + ctx->ss\_size - sizeof(coctx\_param\_t);

sp = (char\*)((unsigned long)sp & -16L);

coctx\_param\_t\* param = (coctx\_param\_t\*)sp ;

param->s1 = s;

param->s2 = s1;

memset(ctx->regs, 0, sizeof(ctx->regs));

ctx->regs[ kESP ] = (char\*)(sp) - sizeof(void\*);

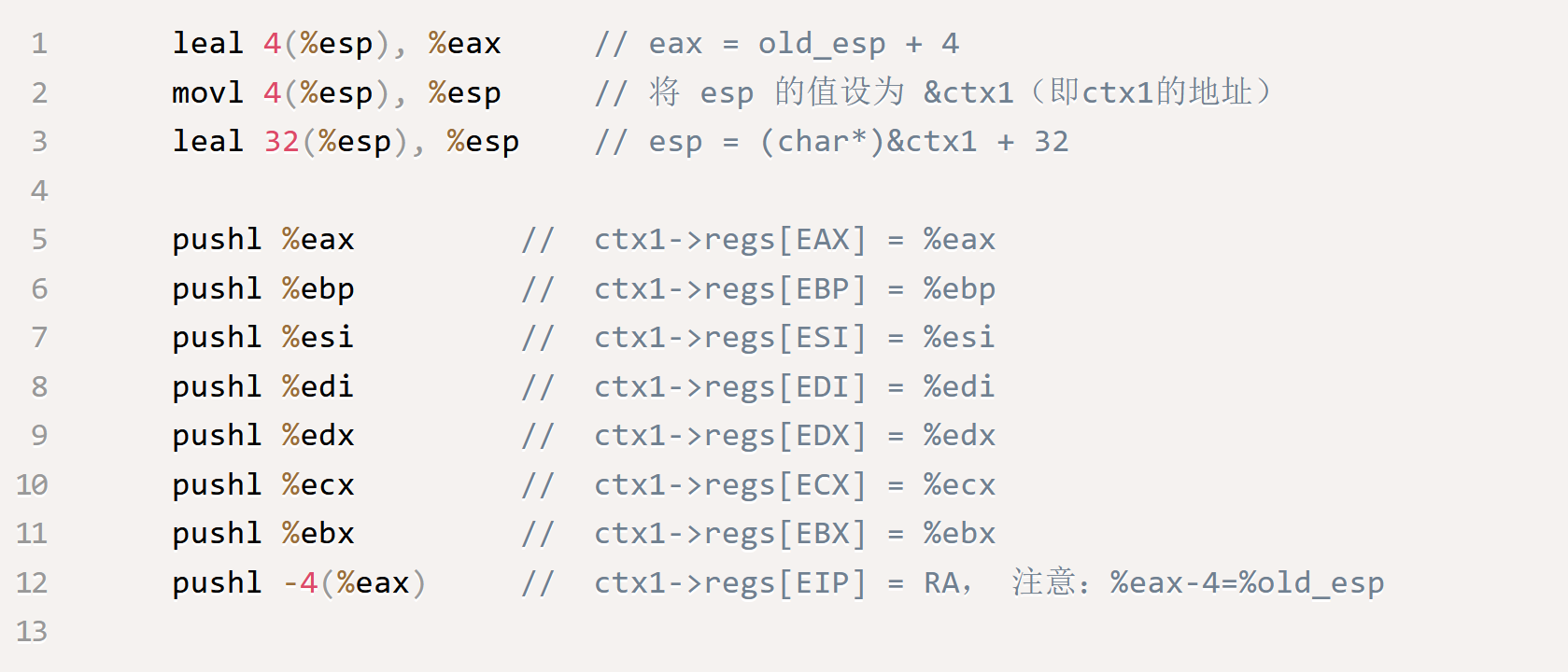
ctx->regs[ kEIP ] = (char\*)pfn;

return 0;

}

这段代码应该比较好理解，首先为函数 coctx\_pfn\_t 预留 2 个参数的栈空间并对其到 16 字节，之后将实参设置到预留的栈上空间中。最后在 ctx 结构中填入相应的，其中记录 reg[kEIP] 返回地址为函数指针 pfn，记录 reg[kESP] 为获得的栈顶指针 sp 减去一个指针长度，这个减去的空间是为返回地址 RA 预留的。当调用 coctx\_swap 时，reg[kEIP] 会被放到返回地址 RA 的位置，待 coctx\_swap 执行结束，自然会跳转到函数 pfn 处执行。

coctx\_swap(ctx1, ctx2) 在 coctx\_swap.S 中实现。这里可以看到，该函数并没有使用 push %ebp; move %ebp, %esp; sub $esp N; 开头，因此栈空间分布中不会出现 ebp 的位置。coctx\_swap 函数主要分为两段，其首先将当前的上下文环境保存到 ctx1 结构中：



这里需要注意指令 leal 和 movl 的区别。leal 将 eax 的值设置成为 esp 的值加 4，而 movl 将 esp 的值设为 esp+4 所指向的内存上的值，也就是参数 ctx1 的地址。之后该函数将 ctx2 中记录的上下文恢复到 CPU 寄存器中，并跳转到其函数地址处运行：



上面的代码看起来可能有些绕：

1、首先 line 1 将 esp 设置为参数 ctx2 的地址，后续的 popl 操作均在 ctx2 的内存空间上执行。

2、line 2-9 将 ctx2->regs[] 中的内容恢复到相应的寄存器中。还记得在前面 coctx\_make 中设置了 regs[EIP] 和 regs[ESP] 吗？这里刚好就对应恢复了相应的值。

3、当执行完 line 9 之后，esp 已经指向了 ctx2 中新的栈顶指针，由于在 coctx\_make 中预留了一个指针长度的 RA 空间，line 10 刚好将新的函数指针 &pfn 设置到该 RA 上。

最后执行 ret 指令时，函数流程将跳到 pfn 处执行。这样，整个协程上下文的切换就完成了。

### 如何使用 libco

我们首先以 libco 提供的例子 example\_echosvr.cpp 来介绍应用程序如何使用 libco 来编写服务端程序。 在 example\_echosvr.cpp 的 main 函数中，主要执行如下几步：

1、创建 socket，监听在本机的 1024 端口，并设置为非阻塞；

2、主线程使用函数 readwrite\_coroutine 创建多个读写协程，调用 co\_resume 启动协程运行直到其挂起。这里我们忽略掉无关的多进程 fork 的过程；

3、主线程继续创建 socket 接收协程 accpet\_co，同样调用 co\_resume 启动协程直到其挂起；

4、主线程调用函数 co\_eventloop 实现事件的监听和协程的循环切换；

函数 readwrite\_coroutine 在外层循环中将新创建的读写协程都加入到队列 g\_readwrite 中，此时这些读写协程都没有具体与某个 socket 连接对应，可以将队列 g\_readwrite 看成一个 coroutine pool。当加入到队列中之后，调用函数 co\_yield\_ct 函数让出 CPU，此时控制权回到主线程。

主线程中的函数 co\_eventloop 监听网络事件，将来自于客户端新进的连接交由协程 accept\_co 处理，关于 co\_eventloop 如何唤醒 accept\_co 的细节我们将在后续介绍。accept\_co 调用函数 accept\_routine 接收新连接，该函数的流程如下：

1. 检查队列 g\_readwrite 是否有空闲的读写 coroutine，如果没有，调用函数 poll 将该协程加入到 Epoll 管理的定时器队列中，也就是 sleep(1000) 的作用；
2. 调用 co\_accept 来接收新连接，如果接收连接失败，那么调用 co\_poll 将服务端的 listen\_fd 加入到 Epoll 中来触发下一次连接事件；
3. 对于成功的连接，从 g\_readwrite 中取出一个读写协程来负责处理读写；

再次回到函数 readwrite\_coroutine 中，该函数会调用 co\_poll 将新建立的连接的 fd 加入到 Epoll 监听中，并将控制流程返回到 main 协程；当有读或者写事件发生时，Epoll 会唤醒对应的 coroutine ，继续执行 read 函数以及 write 函数。

上面的过程大致说明了控制流程是如何在不同的协程中切换，接下来我们介绍具体的实现细节，即如何通过 Epoll 来管理协程，以及如何对系统函数进行改造以满足 libco 的调用。

### 通过 Epoll 管理和唤醒协程

#### Epoll 监听 FD

上一章节中介绍了协程可以通过函数 co\_poll 来将 fd 交由 Epoll 管理，待 Epoll 的相应的事件触发时，再切换回来执行 read 或者 write 操作，从而实现由 Epoll 管理协程的功能。co\_poll 函数原型如下：

int co\_poll(stCoEpoll\_t \*ctx, struct pollfd fds[],

nfds\_t nfds, int timeout\_ms)

stCoEpoll\_t 是为 libco 定制的 Epoll 相关数据结构（这里吐槽一下，参数使用 ctx 也是醉了），fds 是 pollfd 结构的文件句柄，nfds 为 fds 数组的长度，最后一个参数表示定时器时间，也就是在 timeout 毫秒之后触发处理这些文件句柄。这里可以看到，co\_poll 能够同时将多个文件句柄同时加入到 Epoll 管理中。我们先看 stCoEpoll\_t 结构：

struct stCoEpoll\_t

{

int iEpollFd; // Epoll 主 FD

static const int \_EPOLL\_SIZE = 1024 \* 10; // Epoll 可以监听的句柄总数

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; // Epoll 返回的事件结果

};

以 stTimeout\_ 开头的数据结构与 libco 的定时器管理有关，我们在后面介绍。co\_epoll\_res 是对 Epoll 事件数据结构的封装，也就是每次触发 Epoll 事件时的返回结果，在 Unix 和 MaxOS 下，libco 将使用 Kqueue 替代 Epoll，因此这里也保留了 kevent 数据结构。

struct co\_epoll\_res

{

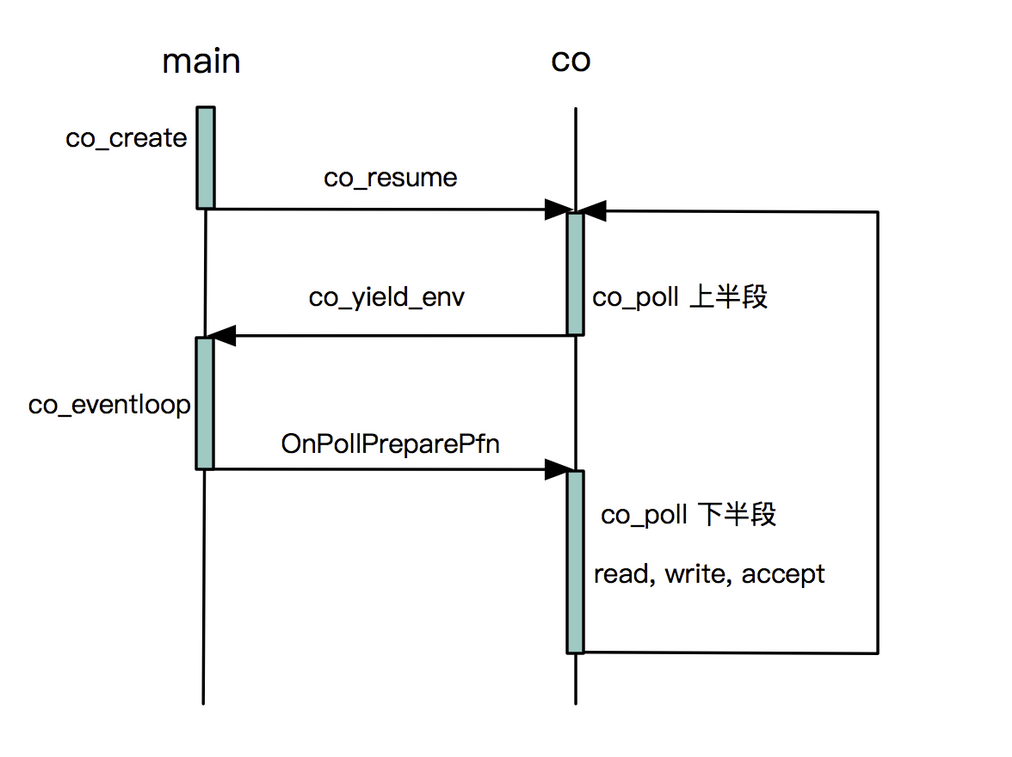
int size;

struct epoll\_event \*events; // for linux epoll

struct kevent \*eventlist; // for Unix or MacOs kqueue

};

co\_poll 实际是对函数 co\_poll\_inner 的封装。我们将 co\_epoll\_inner 函数的结构分为上下两半段。在上半段中，调用 co\_poll 的协程 C将其需要监听的句柄数组 fds 都加入到 Epoll 管理中，并通过函数 co\_yield\_env 让出 CPU；当 main 协程的事件循环 co\_eventloop 中触发了 C 对应的监听事件时，会恢复 C 的执行。此时，C将开始执行下半段，即将上半段添加的句柄 fds 从 epoll 中移除，清理残留的数据结构，下面的流程图简要说明了控制流的转移过程：



有了上面的基本概念，我们来看具体的实现细节。co\_poll 首先在内部将传入的文件句柄数组 fds 转化为数据结构 stPoll\_t，这一步主要是为了方便后续处理。该结构记录了 iEpollFd，ndfs，fds 数组，以及该协程需要执行的函数和参数。有两点需要说明的是：

1、对于每一个 fd，为其申请一个 stPollItem\_t 来管理对应 Epoll 事件以及记录回调参数。libco 在此做了一个小的优化，对于长度小于 2 的 fds 数组，直接在栈上定义相应的 stPollItem\_t 数组，否则从堆中申请内存。这也是一种比较常见的优化，毕竟从堆中申请内存比较耗时；

2、函数指针 OnPollProcessEvent 封装了协程的切换过程。当传入指定的 stPollItem\_t 结构时，即可唤醒对应于该结构的 coroutine，将控制权交由其执行；

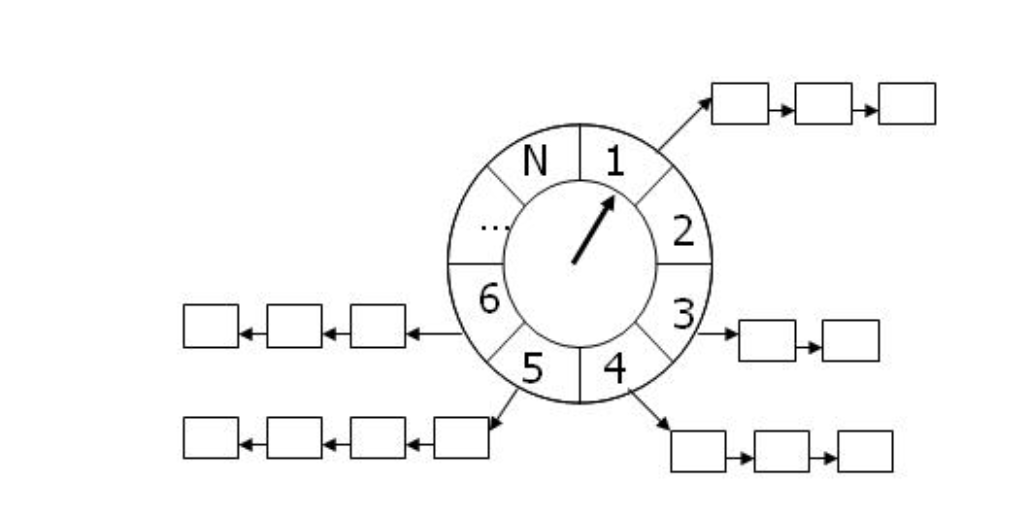
co\_poll 的第二步，也是最关键的一步，就是将 fd 数组全部加入到 Epoll 中进行监听。协程 C会将每一个 epoll\_event 的 data.ptr 域设置为对应的 stPollItem\_t 结构。这样当事件触发时，可以直接从对应的 ptr 中取出 stPollItem\_t 结构，然后唤醒指定协程。

如果本次操作提供了 Timeout 参数，co\_poll 还会将协程 C本次操作对应的 stPoll\_t 加入到定时器队列中。这表明在 Timeout 定时触发之后，也会唤醒协程 C的执行。当整个上半段都完成后，co\_poll 立即调用 co\_yield\_env 让出 CPU，执行流程跳转回到 main 协程中。

从上面的流程图中也可以看出，当执行流程再次跳回时，表明协程 C添加的读写等监听事件已经触发，即可以执行相应的读写操作了。此时 C首先将其在上半段中添加的监听事件从 Epoll 中删除，清理残留的数据结构，然后调用读写逻辑。

#### 定时器实现

协程 C在将一组 fds 加入 Epoll 的同时，还能为其设置一个超时时间。在超时时间到期时，也会再次唤醒 C来执行。libco 使用 Timing-Wheel 来实现定时器。关于 Timing-Wheel 算法，可以参考 [Linux 下内核定时器实现](https://developer.ibm.com/)，其优势是 O(1) 的插入和删除复杂度，缺点是只有有限的长度，在某些场合下不能满足需求。



回过去看 stCoEpoll\_t 结构，其中 \*pTimeout 代表时间轮，通过函数 AllocateTimeout 初始化为一个固定大小（60 \* 1000）的数组。根据 Timing-Wheel 的特性可知，libco 只支持最大 60s 的定时事件。而实际上，在添加定时器时，libco 要求定时时间不超过 40s。成员 pstTimeoutList 记录在 co\_eventloop 中发生超时的事件，而 pstActiveList 记录当前活跃的事件，包括超时事件。这两个结构都将在 co\_eventloop 中进行处理。

下面我们简要分析一下加入定时器的实现：

int AddTimeout( stTimeout\_t \*apTimeout, stTimeoutItem\_t \*apItem,

unsigned long long allNow )

{

if( apTimeout->ullStart == 0 ) // 初始化时间轮的基准时间

{

apTimeout->ullStart = allNow;

apTimeout->llStartIdx = 0; // 当前时间轮指针指向数组0

}

// 1. 当前时间不可能小于时间轮的基准时间

// 2. 加入的定时器的超时时间不能小于当前时间

if( allNow < apTimeout->ullStart || apItem->ullExpireTime < allNow )

{

return \_\_LINE\_\_;

}

int diff = apItem->ullExpireTime - apTimeout->ullStart;

if( diff >= apTimeout->iItemSize ) // 添加的事件不能超过时间轮的大小

{

return \_\_LINE\_\_;

}

// 插入到时间轮盘的指定位置

AddTail( apTimeout->pItems +

(apTimeout->llStartIdx + diff ) % apTimeout->iItemSize, apItem );

return 0;

}

定时器的超时检查在函数 co\_eventloop 中执行。

#### EPOLL 事件循环

main 协程通过调用函数 co\_eventloop 来监听 Epoll 事件，并在相应的事件触发时切换到指定的协程执行。有关 co\_eventloop 与 应用协程的交互过程在上一节的流程图中已经比较清楚了，下面我们主要介绍一下 co\_eventloop 函数的实现：

上文中也提到，通过 epoll\_wait 返回的事件都保存在 stCoEpoll\_t 结构的 co\_epoll\_res 中。因此 co\_eventloop 首先为 co\_epoll\_res 申请空间，之后通过一个无限循环来监听所有 coroutine 添加的所有事件：

for(;;)

{

int ret = co\_epoll\_wait( ctx->iEpollFd,result,stCoEpoll\_t::\_EPOLL\_SIZE, 1 );

...

}

对于每一个触发的事件，co\_eventloop 首先通过指针域 data.ptr 取出保存的 stPollItem\_t 结构，并将其添加到 pstActiveList 列表中；之后从定时器轮盘中取出所有已经超时的事件，也将其全部添加到 pstActiveList 中，pstActiveList 中的所有事件都作为活跃事件处理。

对于每一个活跃事件，co\_eventloop 将通过调用对应的 pfnProcess 也就是上图中的OnPollProcessEvent 函数来切换到该事件对应的 coroutine，将流程跳转到该 coroutine 处执行。

最后 co\_eventloop 在调用时也提供一个额外的参数来供调用者传入一个函数指针 pfn。该函数将会在每次循环完成之后执行；当该函数返回 -1 时，将会终止整个事件循环。用户可以利用该函数来控制 main 协程的终止或者完成一些统计需求。

### 非侵入改造系统函数

libco 在文件 co\_hook\_sys\_call.cpp 中实现了对于系统函数的 hook。关于这一块的内容，推荐先看一下以前写的文章[动态链接黑魔法: Hook 系统函数](http://kaiyuan.me/2017/05/03/function_wrapper/)，介绍了如何使用运行时动态链接 Hook 系统函数的过程，看完应该有一个大致的了解。

## bthread

参考：

<https://cloud.tencent.com/developer/article/1915033>

### 概述

如果你使用过brpc，那么对bthread应该并不陌生。毫不夸张地说，brpc的精华全在bthread上了。bthread可以理解为“协程”，尽管官方文档的FAQ中，并不称之为协程。

### 原理

## Boost.context