一个轻量级的协助程框架

****[我的上铺叫路遥](http://weibo.com/fullofbull)****

http://coolshell.cn/articles/10975.html

协程(coroutine)顾名思义就是“协作的例程”（co-operative routines）。跟具有操作系统概念的线程不一样，协程是在用户空间利用程序语言的语法语义就能实现逻辑上类似多任务的编程技巧。实际上协程的概念比线程还要早，按照 Knuth 的说法**“子例程是协程的特例”**，一个子例程就是一次子函数调用，那么实际上协程就是类函数一样的程序组件，你可以在一个线程里面轻松创建数十万个协程，就像数十万次函数调用一样。只不过子例程只有一个调用入口起始点，返回之后就结束了，而协程入口既可以是起始点，又可以从上一个返回点继续执行，也就是说协程之间可以通过 yield 方式转移执行权，**对称（symmetric）、平级**地调用对方，而不是像例程那样上下级调用关系。当然 Knuth 的“特例”指的是协程也可以模拟例程那样实现上下级调用关系，这就叫**非对称协程**（asymmetric coroutines）。

**基于事件驱动模型**

我们举一个例子来看看一种**对称协程**调用场景，大家最熟悉的“生产者-消费者”事件驱动模型，一个协程负责生产产品并将它们加入队列，另一个负责从队列中取出产品并使用它。为了提高效率，你想一次增加或删除多个产品。伪代码可以是这样的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # producer coroutine  loop  while queue is not full    create some new items    add the items to queue  yield to consumer    # consumer coroutine  loop  while queue is not empty    remove some items from queue    use the items  yield to producer |

大多数教材上拿这种模型作为多线程的例子，实际上多线程在此的应用还是显得有点“重量级”，由于缺乏 yield 语义，线程之间不得不使用同步机制来避免产生全局资源的竟态，这就不可避免产生了休眠、调度、切换上下文一类的系统开销，而且线程调度还会产生时序上的不确定性。而对于协程来说，“挂起”的概念只不过是转让代码执行权并调用另外的协程，待到转让的协程告一段落后重新得到调用并从挂起点“唤醒”，这种协程间的调用是逻辑上可控的，时序上确定的，可谓一切尽在掌握中。

当今一些具备协程语义的语言，比较重量级的如C#、erlang、golang，以及轻量级的python、lua、javascript、ruby，还有函数式的scala、scheme等。相比之下，作为原生态语言的 C 反而处于尴尬的地位，原因在于 C 依赖于一种叫做**栈帧**的例程调用，例程内部的状态量和返回值都保留在堆栈上，这意味着生产者和消费者相互之间无法实现平级调用，当然你可以改写成把生产者作为主例程然后将产品作为传递参数调用消费者例程，这样的代码写起来费力不讨好而且看起来会很难受，特别当协程数目达到十万数量级，这种写法就过于僵化了。

这就引出了协程的概念，**如果将每个协程的上下文（比如程序计数器）保存在其它地方而不是堆栈上，协程之间相互调用时，被调用的协程只要从堆栈以外的地方恢复上次出让点之前的上下文即可，这有点类似于 CPU 的上下文切换，**遗憾的是似乎只有更底层的汇编语言才能做到这一点。

难道 C 语言只能用多线程吗？幸运的是，C 标准库给我们提供了两种协程调度原语：一种是[setjmp/longjmp](http://zh.wikipedia.org/wiki/Setjmp.h" \o "http://zh.wikipedia.org/wiki/Setjmp.h)，另一种是[ucontext 组件](http://pubs.opengroup.org/onlinepubs/7990989799/xsh/ucontext.h.html" \o "http://pubs.opengroup.org/onlinepubs/7990989799/xsh/ucontext.h.html)，它们内部（当然是用汇编语言）实现了协程的上下文切换，相较之下前者在应用上会产生相当的不确定性（比如不好封装，具体说明参考联机文档），所以后者应用更广泛一些，网上绝大多数 C 协程库也是基于 ucontext 组件实现的。

**“蝇量级”的协程库**

在此，我来介绍一种“蝇量级”的开源 C 协程库 [protothreads](http://dunkels.com/adam/pt/" \o "http://dunkels.com/adam/pt/)。这是一个全部用 ANSI C 写成的库，之所以称为“蝇量级”的，就是说，实现已经不能再精简了，几乎就是原语级别。事实上 protothreads 整个库不需要链接加载，因为所有源码都是头文件，类似于 STL 这样不依赖任何第三方库，在任何平台上可移植；总共也就 5 个头文件，有效代码量不足 100 行；API 都是宏定义的，所以不存在调用开销；最后，每个协程的空间开销是 2 个字节（是的，你没有看错，就是一个 short 单位的“栈”！）当然这种精简是要以使用上的局限为代价的，接下来的分析会说明这一点。

先来看看 protothreads 作者，[Adam Dunkels](http://dunkels.com/adam/" \o "http://dunkels.com/adam/)，一位来自瑞典皇家理工学院的计算机天才帅哥。话说这哥们挺有意思的，写了好多轻量级的作品，都是 BSD 许可证。顺便说一句，轻量级开源软件全世界多如牛毛，可像这位哥们写得如此出名的并不多。比如嵌入式网络操作系统 [Contiki](http://www.contiki-os.org/" \o "http://www.contiki-os.org/)，国人耳熟能详的 TCP/IP 协议栈 [uIP](http://en.wikipedia.org/wiki/UIP_(micro_IP)" \o "http://en.wikipedia.org/wiki/UIP_(micro_IP)) 和[lwIP](http://savannah.nongnu.org/projects/lwip/" \o "http://savannah.nongnu.org/projects/lwip/) 也是出自其手。上述这些软件都是经过数十年企业级应用的考验，质量之高可想而知。

很多人会好奇如此“蝇量级”的代码究竟是怎么实现的呢？在分析 protothreads 源码之前，我先来给大家补一补 C 语言的基础课;-^)简而言之，这利用了 C 语言特性上的一个“奇技淫巧”，而且这种技巧恐怕连许多具备十年以上经验的 C 程序员老手都不见得知晓。当然这里先要声明我不是推荐大家都这么用，实际上这是以破坏语言的代码规范为代价，在一些严肃的项目工程中需要谨慎对待，除非你想被炒鱿鱼。

**C 语言的“yield 语义”**

下面的教程来自于一位 ARM 工程师、天才黑客 [Simon Tatham](http://www.chiark.greenend.org.uk/~sgtatham/" \o "http://www.chiark.greenend.org.uk/~sgtatham/)（开源 Telnet/SSH 客户端 [PuTTY](http://www.chiark.greenend.org.uk/~sgtatham/putty/" \o "http://www.chiark.greenend.org.uk/~sgtatham/putty/) 和汇编器[NASM](http://www.nasm.us/" \o "http://www.nasm.us/) 的作者，吐槽一句，PuTTY的源码号称是所有正式项目里最难 hack 的 C，你应该猜到作者是什么语言出身）的博文：[Coroutines in C](http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html" \o "http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html)。中文译文在[这里](http://www.oschina.net/translate/coroutines-in-c" \o "http://www.oschina.net/translate/coroutines-in-c)。

我们知道 python 的 yield 语义功能类似于一种迭代生成器，函数会保留上次的调用状态，并在下次调用时会从上个返回点继续执行。用 C 语言来写就像这样：

|  |  |
| --- | --- |
| 1  2  3  4  5 | int function(void) {    int i;    for (i = 0; i < 10; i++)      return i;   /\* won't work, but wouldn't it be nice \*/  } |

连续对它调用 10 次，它能分别返回 0 到 9。该怎样实现呢？可以利用 goto 语句，如果我们在函数中加入一个状态变量，就可以这样实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | int function(void) {    static int i, state = 0;    switch (state) {      case 0: goto LABEL0;      case 1: goto LABEL1;    }    LABEL0: /\* start of function \*/    for (i = 0; i < 10; i++) {      state = 1; /\* so we will come back to LABEL1 \*/      return i;      LABEL1:; /\* resume control straight after the return \*/    }  } |

这个方法是可行的。我们在所有需要 yield 的位置都加上标签：起始位置加一个，还有所有 return 语句之后都加一个。每个标签用数字编号，我们在状态变量中保存这个编号，这样就能在我们下次调用时告诉我们应该跳到哪个标签上。每次返回前，更新状态变量，指向到正确的标签；不论调用多少次，针对状态变量的 switch 语句都能找到我们要跳转到的位置。

但这还是难看得很。最糟糕的部分是所有的标签都需要手工维护，还必须保证函数中的标签和开头 switch 语句中的一致。每次新增一个 return 语句，就必须想一个新的标签名并将其加到 switch 语句中；每次删除 return 语句时，同样也必须删除对应的标签。这使得维护代码的工作量增加了一倍。

仔细想想，其实我们可以不用 switch 语句来决定要跳转到哪里去执行，而是**直接利用 switch 语句本身来实现跳转**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int function(void) {    static int i, state = 0;    switch (state) {      case 0: /\* start of function \*/      for (i = 0; i < 10; i++) {        state = 1; /\* so we will come back to "case 1" \*/        return i;        case 1:; /\* resume control straight after the return \*/      }    }  } |

酷！没想到 switch-case 语句可以这样用，其实说白了 C 语言就是脱胎于汇编语言的，switch-case 跟 if-else 一样，无非就是汇编的条件跳转指令的另类实现而已（这也间接解释了为何汇编程序员经常揶揄 C 语言是“大便一样的代码”）。我们还可以用 \_\_LINE\_\_ 宏使其更加一般化：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int function(void) {    static int i, state = 0;    switch (state) {      case 0: /\* start of function \*/      for (i = 0; i < 10; i++) {        state = \_\_LINE\_\_ + 2; /\* so we will come back to "case \_\_LINE\_\_" \*/        return i;        case \_\_LINE\_\_:; /\* resume control straight after the return \*/      }    }  } |

这样一来我们可以用宏提炼出一种范式，封装成组件：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #define Begin() static int state=0; switch(state) { case 0:  #define Yield(x) do { state=\_\_LINE\_\_; return x; case \_\_LINE\_\_:; } while (0)  #define End() }  int function(void) {    static int i;    Begin();    for (i = 0; i < 10; i++)      Yield(i);    End();  } |

怎么样，看起来像不像发明了一种全新的语言？**实际上我们利用了 switch-case 的分支跳转特性，以及预编译的 \_\_LINE\_\_ 宏，实现了一种隐式状态机，最终实现了“yield 语义”。**

还有一个问题，当你欢天喜地地将这种鲜为人知的技巧运用到你的项目中，并成功地拿去向你的上司邀功问赏的时候，你的上司会怎样看待你的代码呢？你的宏定义中大括号没有匹配完整，在代码块中包含了未用到的 case，Begin 和 Yield 宏里面不完整的七拼八凑……你简直就是公司里不遵守编码规范的反面榜样！

别着急，在原文中 Simon Tatham 大牛帮你找到一个坚定的反驳理由，我觉得对程序员来说简直是金玉良言。

将编程规范用在这里是不对的。文章里给出的示例代码不是很长，也不很复杂，即便以状态机的方式改写还是能够看懂的。但是随着代码越来越长，改写的难度将越来越大，改写对直观性造成的损失也变得相当相当大。

想一想，一个函数如果包含这样的小代码块：

|  |  |
| --- | --- |
| 1  2  3 | case STATE1:  /\* perform some activity \*/  if (condition) state = STATE2; else state = STATE3; |

对于看代码的人说，这和包含下面小代码块的函数没有多大区别：

|  |  |
| --- | --- |
| 1  2  3 | LABEL1:  /\* perform some activity \*/  if (condition) goto LABEL2; else goto LABEL3; |

是的，这两个函数的结构在视觉上是一样的，而对于函数中实现的算法，两个函数都一样不利于查看。因为你使用协程的宏而炒你鱿鱼的人，一样会因为你写的函数是由小块的代码和 goto 语句组成而吼着炒了你。只是这次他们没有冤枉你，因为像那样设计的函数会严重扰乱算法的结构。

**编程规范的目标就是为了代码清晰。**如果将一些重要的东西，像 switch、return 以及 case 语句，隐藏到起“障眼”作用的宏中，从编程规范的角度讲，可以说你扰乱了程序的语法结构，并且违背了代码清晰这一要求。但是我们这样做是为了突出程序的算法结构，而算法结构恰恰是看代码的人更想了解的。

**任何编程规范，坚持牺牲算法清晰度来换取语法清晰度的，都应该重写。**如果你的上司因为使用了这一技巧而解雇你，那么在保安把你往外拖的时候要不断告诉他这一点。

原文作者最后给出了一个 MIT 许可证的 [coroutine.h](http://www.chiark.greenend.org.uk/~sgtatham/coroutine.h" \o "http://www.chiark.greenend.org.uk/~sgtatham/coroutine.h) 头文件。值得一提的是，正如文中所说，这种协程实现方法有个使用上的局限，就是**协程调度状态的保存依赖于 static 变量，而不是堆栈上的局部变量**，实际上也无法用局部变量（堆栈）来保存状态，这就使得代码不具备可重入性和多线程应用。后来作者补充了一种技巧，就是将局部变量包装成函数参数传入的一个虚构的上下文结构体指针，然后用动态分配的堆来“模拟”堆栈，解决了线程可重入问题。但这样一来反而有损代码清晰，比如所有局部变量都要写成对象成员的引用方式，特别是局部变量很多的时候很麻烦，再比如宏定义 malloc/free 的玩法过于托大，不易控制，搞不好还增加了被炒鱿鱼的风险（只不过这次是你活该）。

我个人认为，既然协程本身是一种单线程的方案，那么我们应该假定应用环境是单线程的，不存在代码重入问题，所以我们可以大胆地使用 static 变量，维持代码的简洁和可读性。事实上**我们也不应该在多线程环境下考虑使用这么简陋的协程**，非要用的话，前面提到 glibc 的 ucontext 组件也是一种可行的替代方案，它提供了一种协程私有堆栈的上下文，当然这种用法在跨线程上也并非没有限制，请仔细阅读联机文档。

**Protothreads的上下文**

感谢 Simon Tatham 的淳淳教诲，接下来我们可以 hack 一下源码了。先来看看实现 protothreads 的数据结构， 实际上它就是协程的**上下文结构体**，用以保存状态变量，相信你很快就明白为何它的“堆栈”只有 2 个字节：

|  |  |
| --- | --- |
| 1  2  3 | struct pt {    lc\_t lc;  } |

里面只有一个 short 类型的变量，实际上它是用来保存上一次出让点的程序计数器。这也映证了协程比线程的灵活之处，就是协程可以是 stackless 的，如果需要实现的功能很单一，比如像生产者-消费者模型那样用来做事件通知，那么实际上协程需要保存的状态变量仅仅是一个程序计数器即可。像 python generator 也是 stackless 的，当然实现一个迭代生成器可能还需要保留上一个迭代值，前面 C 的例子是用 static 变量保存，你也可以设置成员变量添加到上下文结构体里面。如果你真的不确定用协程调度时需要保存多少状态变量，那还是用 ucontext 好了，它的上下文提供了堆栈和信号，但是由用户负责分配资源，详细使用方法见联机文档。。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | typedef struct ucontext {    struct ucontext\_t \*uc\_link;    sigset\_t uc\_sigmask;    stack\_t uc\_stack;    ...  } ucontext\_t; |

**Protothreads的原语和组件**

有点扯远了，回到 protothreads，看看提供的协程“原语”。有两种实现方法，在 ANSI C 下，就是传统的 switch-case 语句：

|  |  |
| --- | --- |
| 1  2  3  4 | #define LC\_INIT（s） s = 0;  // 源码中是有分号的，一个低级 bug，啊哈～  #define LC\_RESUME(s) switch (s) { case 0:  #define LC\_SET(s) s = \_\_LINE\_\_; case \_\_LINE\_\_:  #define LC\_END(s) } |

但这种“原语”有个难以察觉的缺陷：**就是你无法在 LC\_RESUME 和 LC\_END （或者包含它们的组件）之间的代码中使用 switch-case语句，因为这会引起外围的 switch 跳转错误！**为此，protothreads 又实现了基于 GNU C 的调度“原语”。在 GNU C 下还有一种语法糖叫做标签指针，就是在一个 label 前面加 &&（不是地址的地址，是 GNU 自定义的符号），可以用 void 指针类型保存，然后 goto 跳转：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | typedef void \* lc\_t；  #define LC\_INIT(s) s = NULL  #define LC\_RESUME(s) \    do { \      if (s != NULL) { \        goto \*s; \      }    } while (0)  #define LC\_CONCAT2(s1, s2) s1##s2  #define LC\_CONCAT(s1, s2) LC\_CONCAT2(s1, s2)  #define LC\_SET(s) \    do { \      LC\_CONCAT(LC\_LABEL, \_\_LINE\_\_): \      （s） = &&LC\_CONCAT(LC\_LABEL, \_\_LINE\_\_); \    } while (0) |

好了，有了前面的基础知识，理解这些“原语”就是小菜一叠，下面看看如何建立“组件”，同时也是 protothreads API，我们先定义四个退出码作为协程的**调度状态机**：

|  |  |
| --- | --- |
| 1  2  3  4 | #define PT\_WAITING 0  #define PT\_YIELDED 1  #define PT\_EXITED  2  #define PT\_ENDED   3 |

下面这些 API 可直接在应用程序中调用：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | /\* 初始化一个协程，也即初始化状态变量 \*/  #define PT\_INIT(pt) LC\_INIT((pt)->lc)    /\* 声明一个函数，返回值为 char 即退出码，表示函数体内使用了 proto thread，（个人觉得有些多此一举） \*/  #define PT\_THREAD(name\_args) char name\_args    /\* 协程入口点， PT\_YIELD\_FLAG=0表示出让，=1表示不出让，放在 switch 语句前面，下次调用的时候可以跳转到上次出让点继续执行 \*/  #define PT\_BEGIN(pt) { char PT\_YIELD\_FLAG = 1; LC\_RESUME((pt)->lc)    /\* 协程退出点，至此一个协程算是终止了，清空所有上下文和标志 \*/  #define PT\_END(pt) LC\_END((pt)->lc); PT\_YIELD\_FLAG = 0; \                     PT\_INIT(pt); return PT\_ENDED; }    /\* 协程出让点，如果此时协程状态变量 lc 已经变为 \_\_LINE\_\_ 跳转过来的，那么 PT\_YIELD\_FLAG = 1，表示从出让点继续执行。 \*/  #define PT\_YIELD(pt)        \    do {            \      PT\_YIELD\_FLAG = 0;        \      LC\_SET((pt)->lc);       \      if(PT\_YIELD\_FLAG == 0) {      \        return PT\_YIELDED;      \      }           \    } while(0)    /\* 附加出让条件 \*/  #define PT\_YIELD\_UNTIL(pt, cond)    \    do {            \      PT\_YIELD\_FLAG = 0;        \      LC\_SET((pt)->lc);       \      if((PT\_YIELD\_FLAG == 0) || !(cond)) { \        return PT\_YIELDED;      \      }           \    } while(0)    /\* 协程阻塞点(blocking),本质上等同于 PT\_YIELD\_UNTIL，只不过退出码是 PT\_WAITING，用来模拟信号量同步 \*/  #define PT\_WAIT\_UNTIL(pt, condition)          \    do {            \      LC\_SET((pt)->lc);       \      if(!(condition)) {        \        return PT\_WAITING;      \      }           \    } while(0)    /\* 同 PT\_WAIT\_UNTIL 条件反转 \*/  #define PT\_WAIT\_WHILE(pt, cond)  PT\_WAIT\_UNTIL((pt), !(cond))    /\* 协程调度，调用协程 f 并检查它的退出码，直到协程终止返回 0，否则返回 1。 \*/  #define PT\_SCHEDULE(f) ((f) < PT\_EXITED)    /\* 这用于非对称协程，调用者是主协程，pt 是和子协程 thread （可以是多个）关联的上下文句柄，主协程阻塞自己调度子协程，直到所有子协程终止 \*/  #define PT\_WAIT\_THREAD(pt, thread) PT\_WAIT\_WHILE((pt), PT\_SCHEDULE(thread))    /\* 用于协程嵌套调度，child 是子协程的上下文句柄 \*/  #define PT\_SPAWN(pt, child, thread)   \    do {            \      PT\_INIT((child));       \      PT\_WAIT\_THREAD((pt), (thread));   \    } while(0) |

暂时介绍这么多，用户还可以根据自己的需求随意扩展组件，比如实现信号量，你会发现脱离了操作系统环境下的信号量竟是如此简单：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | struct pt\_sem {    unsigned int count;  };    #define PT\_SEM\_INIT(s, c) (s)->count = c    #define PT\_SEM\_WAIT(pt, s)  \    do {            \      PT\_WAIT\_UNTIL(pt, (s)->count > 0);    \      --(s)->count;       \    } while(0)    #define PT\_SEM\_SIGNAL(pt, s) ++(s)->count |

这些应该不需要我多说了吧，呵呵，让我们回到最初例举的生产者-消费者模型，看看protothreads表现怎样。

**Protothreads实战**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | #include "pt-sem.h"    #define NUM\_ITEMS 32  #define BUFSIZE 8    static struct pt\_sem mutex, full, empty;    PT\_THREAD(producer(struct pt \*pt))  {    static int produced;      PT\_BEGIN(pt);    for (produced = 0; produced < NUM\_ITEMS; ++produced) {      PT\_SEM\_WAIT(pt, &full);      PT\_SEM\_WAIT(pt, &mutex);      add\_to\_buffer(produce\_item());      PT\_SEM\_SIGNAL(pt, &mutex);      PT\_SEM\_SIGNAL(pt, &empty);    }    PT\_END(pt);  }    PT\_THREAD(consumer(struct pt \*pt))  {    static int consumed;      PT\_BEGIN(pt);    for (consumed = 0; consumed < NUM\_ITEMS; ++consumed) {      PT\_SEM\_WAIT(pt, &empty);      PT\_SEM\_WAIT(pt, &mutex);      consume\_item(get\_from\_buffer());      PT\_SEM\_SIGNAL(pt, &mutex);      PT\_SEM\_SIGNAL(pt, &full);    }    PT\_END(pt);  }    PT\_THREAD(driver\_thread(struct pt \*pt))  {    static struct pt pt\_producer, pt\_consumer;      PT\_BEGIN(pt);    PT\_SEM\_INIT(&empty, 0);    PT\_SEM\_INIT(&full, BUFSIZE);    PT\_SEM\_INIT(&mutex, 1);    PT\_INIT(&pt\_producer);    PT\_INIT(&pt\_consumer);    PT\_WAIT\_THREAD(pt, producer(&pt\_producer) & consumer(&pt\_consumer));    PT\_END(pt);  } |

源码包中的 example-buffer.c 包含了可运行的完整示例，我就不全部贴了。整体框架就是一个 asymmetric coroutines，包括一个主协程 driver\_thread 和两个子协程 producer 和 consumer ，其实不用多说大家也懂的，代码非常清晰直观。我们完全可以通过单线程实现一个简单的事件处理需求，你可以任意添加数十万个协程，几乎不会引起任何额外的系统开销和资源占用。唯一需要留意的地方就是没有一个局部变量，因为 protothreads 是 stackless 的，但这不是问题，首先我们已经假定运行环境是单线程的，其次在一个简化的需求下也用不了多少“局部变量”。如果在协程出让时需要保存一些额外的状态量，像迭代生成器，只要数目和大小都是确定并且可控的话，自行扩展协程上下文结构体即可。

当然这不是说 protothreads 是万能的，它只是贡献了一种模型，你要使用它首先就得学会适应它。下面列举一些 protothreads 的使用限制：

* 由于协程是stackless的，尽量不要使用局部变量，除非该变量对于协程状态是无关紧要的，同理可推，协程所在的代码是不可重入的。
* 如果协程使用 switch-case 原语封装的组件，那么禁止在实际应用中使用 switch-case 语句，除非用 GNU C 语法中的标签指针替代。
* 一个协程内部可以调用其它例程，比如库函数或系统调用，但必须保证该例程是非阻塞的，否则所在线程内的所有协程都将被阻塞。毕竟线程才是执行的最小单位，协程不过是按“时间片轮度”的例程而已。

官网上还例举了更多[实例](http://dunkels.com/adam/pt/examples.html" \o "http://dunkels.com/adam/pt/examples.html)，都非常实用。另外，一个叫 Craig Graham 的工程师扩展了 pt.h，使得 protothreads 支持 sleep/wake/kill 等操作，文件在此 [graham-pt.h](http://dunkels.com/adam/download/graham-pt.h" \o "http://dunkels.com/adam/download/graham-pt.h)。

**协程库 DIY 攻略**

看到这里，手养的你是否想迫不及待地 DIY 一个协程组件呢？哪怕很多动态语言本身已经支持了协程语义，很多 C 程序员仍然倾向于自己实现组件，网上很多开源代码底层用的主要还是 glibc 的 ucontext 组件，毕竟提供堆栈的协程组件使用起来更加通用方便。你可以自己写一个调度器，然后模拟线程上下文，再然后……你就能搞出一个跨平台的COS了（笑）。GNU Pth 线程库就是这么实现的，其原作者德国人 [Ralf S. Engelschall](http://engelschall.com/" \o "http://engelschall.com/) （又是个开源大牛，还写了 [OpenSSL 等许多作品](http://engelschall.com/software-artist.php" \o "http://engelschall.com/software-artist.php)）就写了一篇[论文](http://xmailserver.org/rse-pmt.pdf" \o "http://xmailserver.org/rse-pmt.pdf)教大家如何实现一个线程库。另外 protothreads 官网上也有一大堆[推荐阅读](http://dunkels.com/adam/pt/links.html" \o "http://dunkels.com/adam/pt/links.html)。Have fun！