

第十七章，线程

- ↓ 第十七章，线程
 - ↓ 17.1 进程模型
 - ↓ 17.2 线程模型
 - ↓ 17.2.1 线程模块
 - ↓ 17.2.1.1 创建线程
 - ↓ 17.2.1.2 线程删除
 - ↓ 17.2.1.3 捕获来自 `join` 的例外
 - ↓ 17.2.1.4 `detach` 方法
 - ↓ 17.2.1.5 标识线程
 - ↓ 17.2.1.6 列出当前线程
 - ↓ 17.2.1.7 交出处理器
 - ↓ 17.2.2 数据访问
 - ↓ 17.2.2.1 用 `lock` 进行访问同步
 - ↓ 17.2.2.2 死锁
 - ↓ 17.2.2.3 锁定子过程
 - ↓ 17.2.2.4 `locked` 属性
 - ↓ 17.2.2.5. 锁定方法
 - ↓ 17.2.2.6 条件变量
 - ↓ 17.2.3 其他线程模块
 - ↓ 17.2.3.1 队列 (`queue`)
 - ↓ 17.2.3.2. 信号灯
 - ↓ 17.2.3.3 其他标准线程模块

并行编程要比看上去要难得多。假设我们从一个烹饪书拿出一条菜谱，然后把它转换成某种几十个厨师可以同时工作的东西。那么你有两个实现方法。

一个方法是给每个厨师一个专用的厨房，给它装备原料和器具。对于那些可以很容易分解的菜谱，以及那些可以很容易从一个厨房转到另外一个厨房的食物而言，这个方法很好用，因为它把不同厨师分隔开，互不影响。

另外，你也可以把所有厨师都放在一个厨房里，然后让他们把菜烧出来，让他们混合使用那些东西。这样可能会很乱，尤其是切肉机开始飞转的时候。

这两个方法对应计算机的两种并行编程方法。第一种是 **Unix** 系统里典型的多进程模型，这种模型里每个控制线索都有自己的一套资源，我们把它们放在一起叫进程。第二种模型是多线程模型，这种模型里每个控制线索和其他控制线索共享资源。或者有些场合可能（或者必须）不共享（资源）。

我们都知道厨师喜欢掌勺；这一点我们明白，因为只有让厨师掌好勺才能实现我们想让他们干的事。但是厨师也需要有组织，不管用什么方法。

Perl 支持上面两种模式的组织形式。本章我们将把它们称为进程模型和线程模型。

17.1 进程模型

我们不会在这里太多地讨论进程模型，原因很简单：它遍及本书的所有其他部分。**Perl** 起源于 **Unix** 系统，所以它浸满了每个进程处理自己的事情的概念。如果一个进程想并行处理某些事情，那么逻辑上它必须启动一个并行的进程；也就是说，它必须分裂一个重量级的新进程，它和父进程共享很少东西，除了一些文件描述符以外。（有时候看起来父进程和子进程共享很多东西，但大多数都只是在子进程中复制父进程并且在逻辑概念上并没有真正共享什么东西。操作系统为了偷懒也会强制那种逻辑分离的，这种情况下我们叫它写时拷贝（**copy-on-write**）语意，但是如果我们不首先逻辑上分开，我们实际上就根本不能做拷贝。）

由于历史原因，这种工业级的多进程观点在 **Microsoft** 系统上引起一些问题，因为 **Windows** 没有

完善的多进程模型（并且老实说，它并不常依靠并发编程技术）。而且它通常采用一种多线程的方法。

不过，通过不懈的努力，Perl 5.6 现在在 Windows 里实现了 fork 操作，方法是在同一个进程里克隆一个新的解释器对象。这意味着本书其余部分使用 fork 的例子现在在 Windows 里都可以使用了。克隆出来的解释器与其他解释器共享不可改变的代码段但是有自己的数据段。（当然，对那些可能不理解线程的 C 库可能仍然有问题。）

这种多进程的方法被命名为 **ithread**，是“**interpreter threads**”（解释器线程）的缩写。实现 **ithread** 的最初的动力就是在 Microsoft 系统上模拟 fork。不过，我们很快就意识到，尽管其他解释器是作为独立的线程运行的，它们仍然在同一个进程里运行，因此，要让这些独立的解释器共享数据是相当容易的，尽管缺省时它们并不共享数据。

这种做法和典型的线程模型是相对的，那种模型里所有东西都是共享的，而且你还得花点力气才能不共享一些东西。但是你不能把这两种模型看作是完全分离的两个模式，因为它们都试图在同一条河上架桥通路；只不过它们是从相对的河两岸分别开工的。任何解决并发进程问题的方法实际上最后都是某种程度的共享和某种程度的自私。

所以从长远来看，**ithread** 的目的是允许你需要或者想要的尽可能多的共享。不过，到我们写这些为止，**ithread** 唯一的用户可见的接口是在 Perl 的 Microsoft 版本里面的 fork 调用。我们最终认为，这个方法能比标准的线程方法提供更干净的程序。原则上来说，如果你假设每个人都拥有他该拥有的东西，那么这样的系统更容易运转，相比之下每个人都拥有所有的东西的系统要难于运转得多。不象资本主义经济那样没有共享的东西，也不是共产主义经济那样所有东西都共享。这些东西有点象中庸主义。类似社会主义。不过对于一大堆人而言，只有在拥有一个很大的绞肉机，而且“厨师长”认为它拥有所有东西的时候共享全部东西才可行。

当然，任何计算机的实际控制都是由那个被称做操作系统的独裁者控制的。不过是聪明的独裁者知道什么时候让人们认为它们是资本主义--以及什么时候让他们觉得是共产主义。

17.2 线程模型

并发处理的线程模型是在版本 5.005 里第一次以一种试验特性介绍到 Perl 里来的。（这里“线程模型”的意思是那些缺省时共享数据资源的线程，不是版本 5.6 里的 **ithreads**。）从某种角度来说，这个线程模型即使在 5.6 里也仍然是一个实验的模型，因为 Perl 是一门复杂的语言，而多线程即使最简单的语言里也可能搞得一团糟。在 Perl 的语意里仍然有隐蔽的地方和小漏洞，这些地方不能和共享所有东西的概念完全融合在一起。新的 **ithread** 模型是一个绕开这些问题的尝试，并且在将来的某个时间，现在的线程模型可能会被包容在 **ithread** 模型里（那时我们就可以有一个“缺省时共享所有你能共享的东西”的 **ithread** 接口）。不过，尽管有瑕疵，目前的“试验性”的线程模型仍然在现实世界中的许多方面用得上，那种情况下你只能换一种你不想要的更笨拙的方法。你可以用线程化的 Perl 写出足够坚固的应用，不过你必须非常小心。如果你能想出一种用管道而不是用共享数据结构的方法解决你的问题的话，那么你至少应该考虑使用 **fork**。

不过如果多个任务能够容易且有效地访问同一组数据池（注：上一章讨论的 **System V** 里的共享内存的模型并不算“容易且有效”），那么有些算法可以更容易地表达。这样就可以把代码写的更少和更简单。并且因为在创建线程的时候内核并不必为数据拷贝内存页表（甚至写时拷贝（**copy-on-write**）都不用），那么用这种方法启动一个任务就应该更快些。类似，因为内核不必交换内存页表，环境切换也应该更快一些。（实际上，对于用户级线程而言，内核根本不用参与——当然，用户级的线程有一些内核级的线程没有的问题。）

这些可是好消息。那么现在我们要做些弃权声明。我们已经说过线程在 Perl 里是某种试验特性，而且即使它们不再是试验特性了，那么线程编程也是非常危险的。一个执行流能够把另外一个执行流的数据区捅得乱七八糟的能力可以暴露出比你想象得更多的导致灾难的机会。你可能会对自己说，

“那很容易修理，我只需要在任何共享的数据上加锁就可以了。”不错，共享数据的锁是不可缺少的，不过设计正确的锁协议是臭名昭著地难，协议的错误会导致死锁或者不可预料的结果。如果你在程序里有定时问题，那么使用线程不仅会恶化它们，而且还让他们难于跟踪。

你不仅要确保你自己的共享数据的安全，而且你还要保证这些数据在所有你调用的 **Perl** 模块和 **C** 库里安全。你的 **Perl** 代码可以是 **100%** 线程安全的，但是如果你调用了一个线程不安全的模块或者 **C** 的子过程，而又没有提供你自己的信号灯保护，那么你完了。在你证明之前，你应该假设任何模块都是现成不安全的。那些甚至包括一些标准的模块。甚至是它们的大多数。

我们有没有让你泄气？没有？然后我们还要指出，如果事情到了调度和优先级策略的份上，你还很大程度上依赖你的操作系统的线程库的慈悲。有些线程库在阻塞的系统调用的时候只做线程切换。有些线程库在某个线程做阻塞的系统调用的时候阻塞住整个进程。有些库只在时间量超时（**quantum expiration**）的时候才切换线程（线程或者进程）。有些库只能明确地切换线程。

哦，对了，如果你的进程接收到一个信号，那么信号发送给哪个线程完全是由系统决定的。

如果想在 **Perl** 里写线程程序，你必须制作一个特殊的 **Perl** 的版本，遵照 **Perl** 源程序目录里的 **README.threads** 文件的指示就可以了。这个特殊的 **Perl** 版本几乎是可以肯定要比标准的版本慢一些的。

请不要觉得你只要知道了其他线程模型（**POSIX**，**DEC**，**Microsoft**，等等）的编程特点就认为自己认识了 **Perl** 的线程的运转模式。就象 **Perl** 里的其他东西一样，**Perl** 就是 **Perl**，它不是 **C++** 或者 **Java** 或者其他什么东西。比如，**Perl** 里没有实时线程优先级（也没有实现它们的方法）。也没有互斥线程。你只能用锁定或者是 **Thread::Semaphore** 模块或者 **cond_wait** 设施。

还没泄气？好，因为线程实在是酷。你准备迎接一些乐趣吧。

17.2.1 线程模块

目前的 **Perl** 的线程接口是由 **Thread** 模块定义的。另外还增加了一个新的 **Perl** 关键字，**lock** 操作符。我们将在本章稍后描述 **lock** 操作符。其他标准的线程模块都是在这个基本接口的基础上制作的。

Thread 模块提供了下列类模块：

模块	用途
new	构造一个新的 Thread 。
self	返回我的当前 Thread 对象。
list	返回一个 Thread 列表。

并且，对于 **Thread** 对象，它还提供了这些对象方法：

模块	用途
join	结束一个线程（传播错误）
eval	结束一个线程（捕获错误）
equal	比较两个线程是否相同
tid	返回内部线程 ID

另外，**Thread** 模块还提供了这些重要的函数：

函数	用途
yield	告诉调度器返回一个不同的线程

async	通过闭合域构造一个 Thread
cond_signal	只唤醒一个正在 cond_wait() 一个变量的线程
cond_broadcast	唤醒所有可能在 cond_wait() 一个变量的线程
cond_wait	等待一个变量，直到被一个 cond_signal() 打断或变量上有 cond_broadcast()。

17.2.1.1 创建线程

你可以用两种方法之一派生线程，要么是用 `Thread->new` 类方法或者使用 `async` 函数。不管那种方法，返回值都是一个 `Thread` 对象。`Thread->new` 接收一个要运行的表示某函数的代码引用以及传给那个函数的参数：

```
use Thread;
...
$t = Thread->new(\&func, $arg1, $arg2);
```

你通常会想传递一个闭合域做第一个参数而省略其他的参数：

```
my $something;
$t = Thread->new( sub {say{$something} } );
```

对这种特殊的例子，`async` 函数提供了一些符号上的解放（就是语法糖）：

```
use Thread qw(async);
...
my $something;
$t = async {
    say($something);
};
```

你会注意到我们明确地输入了 `async` 函数。你当然可以用全称的 `Thread::async` 代换，不过那样你的语法糖就不够甜了。因为 `async` 只包含一个闭合域，所以你想放进去的任何东西都必须是一个在传入是的范围的一个词法变量。

17.2.1.2 线程删除

一旦开始——并且开始遭受你的线程库的反复无常——该线程将保持运行直到它的顶层函数（就是你传给构造器的函数）返回。如果你想提前终止一个线程，只需要从那个顶层函数中 `return` 就行了。（注：不要调用 `exit`！那样就试图停止你的整个进程，并且可能会成功。但实际上该进程直到所有线程都退出之后才能退出，而且有些线程在 `exit` 的时候可能拒绝退出。我们稍后有更多内容。）

现在是你的顶层子过程返回的好时机，但是它返回给谁呢？派生这个线程的那个线程可能已经转去做别的事情，并且不再等待一个方法调用的响应了。答案很简单：该线程等待某个过程发出一个方法调用并真正等待一个返回值。那个调用的方法叫 `join`，因为它概念上是把两个线程连接回一个：

```
$retval = $t->join();      # 结束线程 $t
```

`join` 的操作是对子进程的 `waitpid` 的回忆。如果线程已经停止，`join` 方法马上返回该线程的顶层子过程的返回值。如果该线程还没有完蛋，`join` 的动作就象一个阻塞调用，把调用线程不确定地挂起。（这儿没有超时机制。）当该线程最终结束时，`join` 返回。

不过，和 `waitpid` 不一样，`waitpid` 只能终止该进程自己的子过程，一个线程可以 `join` 任何同一个进程内的其他线程。也就是说，与主线程或者父线程连接并不是必要的。唯一的限制是线程不能 `join` 它自己（那样的话就好象安排你自己的葬礼。），而且一个线程不能 `join` 一个已经连接过的线程

（那样就好象两个葬礼主持在尸体上扭打）。如果你试图做这两件事之一那么就会抛出一个例外。

`join` 的返回值不一定是标量值——它可以是一个列表：

```
use Thread 'async';

$t1 = async {
    my $stuff = getpwuid($>);
    return @stuff;
};

$t2 = async {
    my $motd = `cat /etc/modt`;
    return $motd;
};

@retlist = $t1->join();
$retval = $t2->join();

print "1st kid returned @retlist\n";
print "2nd kid returned $retval\n";
```

实际上，一个线程的返回表达式总是在列表环境里计算的，即使 `join` 是在一个标量环境里调用的也如此，那种情况下返回列表的最后一个值。

17.2.1.3 捕获来自 `join` 的例外

如果一个线程带着一个未捕获的例外终止，不会立即杀死整个程序。这就有点头疼了。相比之下，如果一个 `join` 在那个线程上运行，那么 `join` 本身会抛出例外。在一个线程上使用了 `join` 意味着愿意传播该线程抛出的例外。如果你宁可到处捕获这些例外，那么使用 `eval` 方法，它就象一个内建的配对儿，导致例外被放进 `$@`：

```
$retval = $t->eval(); # 捕获 join 错误 if ($@) { warn "thread failed: $@"; } else { print "thread returned $retval\n"; }
```

你在实际中可能还是只想在创建被连接的线程的那个线程里连接该线程——尽管我们没有这个影响的规则。也就是说，你只从父线程里终止子线程。这样可以比较方便地跟踪你应该在那里操作哪个例外。

17.2.1.4 `detach` 方法

这是另外一个终止线程的方法，如果你不准备稍后 `join` 一个线程以获取它的返回值，那么你可以对它调用 `detach` 方法，这样 Perl 会为你清理干净。然后该线程就不能再被连接了。它有点象 Unix 里的一个进程继承给 `init`，不过在 Unix 里这么做唯一的方法是父进程退出。

`detach` 方法并不把该线程“放到后台”；如果你试图退出主程序并且一个已发配的线程仍在运行，那么退出过程将挂起，直到该线程自己退出。更准确一点说，`detach` 只是替你做清理工作。它只是告诉 Perl 在该线程退出之后不必再保留它的返回值和退出状态。从某种意义上来说，`detach` 告诉 Perl 当该线程结束后做一个隐含的 `join` 并且丢掉结果。这一点很重要，如果你既不 `join` 也不 `detach` 一个返回巨大列表的线程，那么那部分存储空间将直到结束时都不能使用，因为 Perl 将不得不为以后（在我们的例子里是非常以后）可能会出现的那个家伙想 `join` 该线程的机会挂起。

在一个发配了的子线程里抛出的例外也不再通过 `join` 传播，因为它们将不再被使用。在顶层函数里合理使用 `eval {}`，你可能会找到其他汇报错误的方法。

17.2.1.5 标识线程

每个 Perl 线程都有一个唯一的线程标识，由 `tid` 对象方法返回：

```
$this_tidno = $t1->tid();
```

一个线程可以通过 `Thread->self` 调用访问它自己的线程对象。不要把这个和线程 ID 混淆：要获得自身的线程 ID，一个线程可以这样：

```
$mytid = Thread->self->tid();    #$$ 是线程，和以前一样
```

要拿一个线程对象和另外一个做比较，用下列之一：

```
Thread::equal($t1, $t2)
$t1->equal($t2)
$t1->tid() == $td->tid()
```

17.2.1.6 列出当前线程

你可以用 `Thread->list` 类方法调用在当前进程获取一个当前线程对象的列表。该列表包括运行着的现成和已经退出但还未连接的线程。你可以在任何线程里做这个工作：

```
for my $t (Thread->list()) { printf "$t has tid = %d\n", $t->tid(); }
```

17.2.1.7 交出处理器

`Thread` 模块支持一个重要的函数，叫 `yield`。它的工作是令调用它的线程放弃处理器。不幸的是，这个函数具体干的事情完全依赖于你所的线程的实现方式。不管怎样，我们还是认为它是一个偶然放弃 CPU 的控制的很好的手势。

```
use Thread 'yield';
yield();
```

你不必使用圆括号。从语法上来讲，这样可能更安全，因为这样能捕获看起来无法避免的“`yeild`”的错误输入：

```
use strict;
use Thread 'yield';
yeild;          # 编译器出错，然后才过去
yield;          # 正确
```

17.2.2 数据访问

到目前为止我们看到的東西都不算太难，不过很快就不是那样了。我们到目前为止所做的任何事情实际上都没有面对线程的并行本性。访问共享数据就会结束上面的状况了。

Perl 里的线程代码在面对数据可视性问题的时候和任何其他 Perl 代码都要经受一样的约束。全局量仍然是通过全局符号表来访问的，而词汇变量仍然是通过某些包含它的词法范围（便签本）访问的。

不过，因为程序里存在多个线程的控制，所以带来一些新的问题。Perl 不允许两个线程同时访问同一个全局变量，否则它们就会踩着对方的脚。（踩得严重与否取决于访问的性质）。相似的情况还有两个线程不能同时访问同一个词法变量，因为如果词法范围在线程使用的闭合域的外面声明，那么它的性质和全局量类似。通过用子过程引用来启动线程（用 `Thread->new`）而不是用闭合域启动（用 `async`），可以限制对词法变量的访问，这样也许能满足你的要求。（不过有时候也不行。）

Perl 给一些内建特殊变量解决了这个问题，比如 `$!` 和 `$_` 和 `@_` 等等，方法是把它们标记为线程相关的数据。糟糕的是所有你日常使用的基础包变量都没有受到保护。

好消息是通常你完全不必为你的词法变量担心——只要它们是在当前线程内部声明的；因为每个线程在启动的时候都会生成自己的词法范围的实例，这是与任何其他的线程隔离的。只有在词法变量在线程之间共享的时候你才需要担心，（比如，四处传递引用，或者在多线程里运行的闭合域里引用词法变量）。

17.2.2.1 用 `lock` 进行访问同步

如果同一时间有多个用户能够访问同一条目，那么就会发生冲突，就象十字路口一样。你唯一的武器就是仔细地锁定。

内建的 `lock` 函数是 Perl 用于访问控制的红绿灯机制。尽管 `lock` 是各种关键字之一，但它是那种层次比较低的，因为如果编译器已经发现用户代码中有一个 `sub lock {}` 定义存在，那么就不会使用内建的 `lock`。这是为了向下兼容，不过，`CORE::LOCK` 总是内建的函数。（在不是为线程使用制作的 perl 版本里调用 `lock` 不是个错误，只是一个无害的“无动作”，至少在最近的版本里如此。）

就好象 `flock` 操作符只是阻塞其它的 `flock` 的实例，而不是实际 I/O 一样，`lock` 也只是阻塞其它 `lock` 的实例，而不是普通的数据访问。实际上，它们也是劝告性锁定。就象交通灯一样。（注：有些铁路十字路口是强制性锁（那些有门的），有些家伙认为 `lock` 也应该是强制性的。不过想象一下，如果现实世界中的每个十字路口都有升降杆是多么可怕。）

你可以锁住独立的标量变量，整个数组和整个哈希。

```
lock $var;
lock @values;
lock %table;
```

不过，在一个聚集上使用 `lock` 并非隐含的对该聚集的每一个标量元素都锁定：

```
lock @values;      # 在线程 1
...
lock $values[23];  # 在线程 2 -- 不会阻塞！
```

如果你锁定一个引用，那么也自动锁住了对引用的访问。也就是说，你获得一个可以释放的析引用。这个技巧很有用，因为对象总是隐藏在一个引用后面，并且你经常想锁住对象。（并且你几乎从来不会想锁住引用。）

当然，交通灯的问题是它们有一半时间是红灯，这时候你只能等待。同样，`lock` 也是阻塞性调用——你的线程会挂起，直到获得了锁。这个过程中没有超时机制。也没有解锁设施，因为锁是动态范围对象。它们持续到它们的语句块，文件或者 `eval` 的结束。如果它们超出了范围，那么它们被自动释放。

锁还是递归的。这意味着如果你在一个函数里锁住了一个变量，而且该函数在持有锁的时候递归，那么同一个线程可以再次成功地锁住该变量。当所有拥有锁的框架都退出以后，锁才最终被删除。

下面是一个简单的演示程序，看看如果没有锁，世界将会怎样。我们将用 `yield` 强制一次环境切换以显示在优先级调度的时候也可能偶然发生的这类问题：

```
use Thread qw/async yield/;
my $var = 0;
sub abump {
    if ($var == 0) {
        yield;
        $var++;
    }
}
```

```
my $t1 = new Thread \&abump;
my $t2 = new Thread \&abump;

for my $t ($t1, $t2) { $t->join}
print "var is $var\n";
```

这段代码总是打印 2（某种意义上的“总是”），因为我们在看到数值为 0 后决定增加数值，但在我们增加之前，另外一个线程也在做一样的事情。

我们可以在检查 `$var` 之前用一个微乎其微的锁来修补这个冲突。下面的代码总是打印 1:

```
sub bump {
    lock $var;
    if ($var == 0) {
        yield;
        $var++;
    }
}
```

请记住我们没有明确的 `unlock` 函数。要控制解锁，只需要增加另外一个嵌套的范围层次就行了，这样锁就会在范围结束后释放。

```
sub abump {
    {
        lock $var;
        if ($var == 0) {
            yield;
            $var++;
        }
    } # 锁在这里释放
    # 其他不用锁定 $var 的代码
}
```

17.2.2.2 死锁

死锁是线程程序员的毒药，因为很容易偶然地就死锁了，但即使你努力做好却很难避免。下面是一个死锁的简单的例子：

```
my $t1 = async { lock $a; yield; lock $b; $a++; $b++ };
my $t2 = async { lock $b; yield; lock $a; $b++; $a++ };
```

解决方法是对于所有需要某个锁集合的当事方，都必须按照相同的顺序获取锁。

把你持有锁的时间最小化也是很好的做法。（至少出于性能的考虑也是好的。但是如果你只是为了减少死锁的风险，那么你所做的只是让复现问题和诊断问题变得更难。）

17.2.2.3 锁定子过程

你可以在一个子过程上加一把锁：

```
lock &func;
```

和数据锁不一样，数据锁只有劝告性锁，而子过程锁是强制性的。除了拥有锁的线程以外其它线程都不能进入子过程。

考虑一下下面的代码，它包含一个涉及 `$done` 变量的冲突条件。（`yield` 只是用于演示）。


```
use Thread qw/async yield/;
my $done = 0;
sub frob {
    my $arg = shift;
    my $tid = Thread->self->tid;
    print "thread $tid: frob $arg\n";
    yield;
    unless ($done) {
        yield;
        $done++;
        frob($arg + 10);
    }
}
```

如果你这样运行：

```
my @t;
for my $i (1..3) {
    push @t, Thread->new(\&frob, $i);
}
for (@t) { $_->join}
print "done is $done\n";
```

下面是输出（哦，有时候是这样的——输出是不可预料的）：

```
thread 1: frob 1
thread 2: frob 2
thread 3: frob 3
thread 1: frob 11
thread 2: frob 12
thread 3: frob 13
done is 3
```

不过如果你这么运行：

```
for my $i (1..3) {
    push @t, async {
        lock &frob;
        frob($i);
    };
}
for (@t) { $_->join }
print "done is $done\n";
```

输出是下面的东西：

```
thread 1: frob 1
thread 1: frob 11
thread 2: frob 2
thread 3: frob 3
done is 1
```

17.2.2.4 locked 属性

尽管你必须遵守子过程锁，但是没有什么东西让你一开始就锁住他们。你可以说锁的位置是劝告性的。不过有些子过程确实需要在调用之前把它们锁住。

子过程的 **locked** 属性就是干这个的。它比调用 **lock &sub** 快，因为它在编译时就知道了，而不只是在运行时。但是其性质和我们提前明确地锁住它是一样的。语法如下：

```
sub frob : locked {  
    # 和以前一样  
}
```

如果你有函数原形，它放在名字和任意属性之间：

```
sub frob ($) : locked {  
    # 和以前一样  
}
```

17.2.2.5. 锁定方法

在子过程上自动加锁的特性是非常棒的，但有时候杀伤力太大。通常来说，当你调用一个对象方法时，是否有多个方法同时运行并没有什么关系，因为它们都代表不同的对象运行。因此你真正想锁住的是其方法正在被调用的那个对象。向该子过程里增加一个 **method** 属性可以实现这个目的：

```
sub frob : locked method {  
    # 和以前一样  
}
```

如果它被当作一个方法调用，那么正在调用的对象被锁住，这样就可以对该对象进行串行访问，但是允许在其他对象上调用该方法。如果该方法不是在对象上调用的，该属性仍然力图做正确的事情：如果你把一个锁住的方法当作一个类方法调用（**Package->new** 而不是 **\$obj->new**），那么包的符号表被锁住。如果你把一个锁住的方法当作普通子过程调用，Perl 会抛出一个错误。

17.2.2.6 条件变量

条件变量允许一个线程放弃处理器，直到某些条件得到满足。当你需要比锁能提供的更多控制机制的时候，条件变量是在线程之间提供协调的点。另一方面，你并不需要比锁有更多过荷的东西，而条件变量就是带着这些思想设计的。你只是用普通锁加上普通条件。如果条件失败，那么你必须通过 **cond_wait** 函数采取特殊的措施；但是我们很有可能成功，因为在一个设计良好的应用里，我们不应该在当前的条件上设置瓶颈。

除了锁和测试，对条件变量的基本操作是由发送或者接收一个“信号”事件（不是 **%SIG** 意义上的真正的信号）组成的。你要么推迟你自己的执行以等待一个事件的到来，要么发送一条事件以唤醒其他正在等待事件到来的线程。**Thread** 模块提供了三个不可移植的函数做这些事情：**cond_wait**，**cond_signal**，和 **cond_broadcast**。这些都是比较原始的机制，在它们的基础上构造了更抽象的模块，比如 **Thread::Queue** 和 **Thread::Semaphore**。如果可能的话，使用那些抽象可能更方便些。

cond_wait 函数接受一个已经被当前的线程锁住的变量，给那个变量解锁，然后阻塞住直到另外一个线程对同一个锁住了的变量做了一次 **cond_signal** 或者 **cond_broadcast**。

被 **cond_wait** 阻塞住的变量在 **cond_wait** 返回以后重新锁住。如果有多个线程在 **cond_wait** 这个变量，那么只有一个线程重新阻塞，因为它们无法重新获得变量的锁。因此，如果你只使用 **cond_wait** 进行同步工作，那么应该尽快放弃变量锁。

cond_signal 函数接受一个已经被当前线程锁住的变量，然后解除一个当前正在 **cond_wait** 该变量的线程的阻塞。如果不止一个线程阻塞在该变量的 **cond_wait** 上，只有解除一个的阻塞，而且你无法预料是哪个。如果没有线程阻塞在该变量的 **cond_wait** 上，该事件被丢弃。

cond_broadcast 函数运行得象 **cond_signal**，但是解除所有在锁住的变量的 **cond_wait** 的线程

的阻塞，而不只是一个。（当然，仍然是某一时刻只有一个线程可以拥有锁住的变量。）

`cond_wait` 应该是一个线程在条件没有得到满足后的最后的手段。`cond_signal` 和 `cond_broadcast` 表明条件已经改变了。我们假设各个事件的安排是这样的：锁定，然后检查一下看看是否满足你需要的条件；如果满足，很好，如果不满足，`cond_wait` 直到满足。这里的重点是放在尽可能避免阻塞这方面的。（在对付线程的时候通常是个好建议。）

下面是一个在两个线程之间来回传递控制的一个例子。千万不要因为看到实际条件都在语句修饰词的右边而被欺骗；除非我们等待的条件是假的，否则决不会调用 `cond_wait`。

```
use Thread qw(async cond_wait cond_signal);
my $wait_var = 0;
async {
    lock $wait_var;
    $wait_var = 1;
    cond_wait $wait_var until $wait_var == 2;
    cond_signal($wait_var);
    $wait_var = 1;
    cond_wait $wait_var until $wait_var == 2;
    cond_signal($wait_var);
};

async {
    lock $wait_var;
    cond_wait $wait_var until $wait_var == 1;
    $wait_var = 2;
    cond_signal($wait_var);
    cond_wait $wait_var until $wait_var == 1;
    $wait_var = 2;
    cond_signal($wait_var);
    cond_wait $wait_var until $wait_var == 1;
};
```

17.2.3 其他线程模块

有几个模块是在基本的 `cond_wait` 上构造的。

17.2.3.1 队列（queue）

标准的 `Thread::Queue` 模块提供了一个在线程之间传递对象而又不用担心锁定和同步问题的方法。它的接口更简单：

方法	用途
<code>new</code>	构造一个 <code>Thread::Queue</code>
<code>equeue</code>	向队列结尾压入一个或更多标量
<code>dequeue</code>	把队列头的第一个标量移出。如果队列里没有内容了，那么 <code>dequeue</code> 阻塞。

请注意，队列和普通管道非常相似，只不过不是发送字节而是传递整个标量，包括引用和赐福过的对象而已！

下面是一个从 `perlthrtut` 手册页来的一个例子：

```
use Thread qw(async/);
use Thread::Queue;
```

```
my $Q = Thread::Queue->new();
async {
    while (defined($datum = $Q->dequeue)) {
        print "Pulled $datum from queue\n";
    }
};

$Q->enqueue(12);
$Q->enqueue("A", "B", "C");
$Q->enqueue($thr);
sleep 3;
$Q->enqueue(\%ENV);
$Q->enqueue(undef);
```

下面是你获得的输出:

```
Pulled 12 from queue
Pulled A from queue
Pulled B from queue
Pulled C from queue
Pulled Thread=SCALAR(0x8117200) from queue
Pulled HASH(0x80dfd8c) from queue
```

请注意当我们通过一个 `async` 闭合域启动一个异步线程的时候 `$Q` 在范围里是怎样的。线程和 Perl 里的其他东西一样遵守同样的范围规则。如果 `$Q` 在 `async` 调用之后才声明，那么上面的例子就不能运行了。

17.2.3.2. 信号灯

`Thread::Semaphore` 给你提供了线程安全的计数信号灯对象，你可以用它来实现你自己的 `p()` 和 `v()` 操作。因为我们大部分人都不要把些操作和荷兰语的 `passeer`（“回合”）和 `verlaat`（“树叶”）联系在一起，所以此模块把这些操作相应称做“向下”和“向上”。（在有些文化里，它们叫“等”和“信号”。）此模块支持下面的方法：

方法	用途
<code>new</code>	构造一个新的 <code>Thread::Semaphore</code> 。
<code>down</code>	分配一个或更多项目。
<code>up</code>	析构一个或者更多项目。

`new` 方法创建一个新的信号灯并且把它初始化为声明的初始计数。如果没有声明初始数值，则该信号灯的初始值设置为 `1`。（数字代表某些条目的“池”，如果所有数字都分配完了则它们会“用光”。）

```
use Thread::Semaphore;
$mutex = Thread::Semaphore->new($MAX);
```

`down` 方法把信号灯的计数值减去所声明的数值，如果没有给出此数值则为 `1`。你可以认为它是一个分配某些或者所有资源的动作。如果信号灯计数减到零以下，这个方法会阻塞住直到信号灯计数等于或者大于你要求的数量。用下面的方法调用它：

```
$mutex->down();
```

`up` 方法给该信号灯的计数值加指定的数值，如果没有给出此数值则为 `1`。你可以认为这是一个释放原先分配的资源的操作。这样的操作至少要解除一个因为试图 `down` 这个信号等而阻塞住的线程。用下面这样的方法调用：

```
$mutex->up();
```

17.2.3.3 其他标准线程模块

Thread::Signal 允许你启动一个线程用于接收你的进程的 **%SIG** 信号。这就解决了目前仍然让人头疼的问题：目前的 **Perl** 实现里信号是不可靠的，如果轻率使用可能会偶而导致内核倾倒。

这些模块仍然在开发中，并且也可能无法在你的系统上提供你需要的结果。但，它们也可能可以用。如果不能，那么就是因为某些象你一样的人还没有修补它们。可能你或者某个人就可以参与进来帮助解决问题。

Revision: r1.2 - 08 Sep 2005 - 13:59 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [P3PerlTech](#) > [PerlThread](#)

版权 © 1999-2006 归这里所有作者. [PostgreSQL](#) 的中文文档版权归何伟平所有.
向为这里贡献想法,文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)