

第十六章，进程间通讯

- ↓ 第十六章，进程间通讯
 - ↓ 16.1 信号
 - ↓ 16.1.1 给进程组发信号
 - ↓ 16.1.2 收割僵死进程
 - ↓ 16.1.3 给慢速操作调速
 - ↓ 16.1.4 阻塞信号
 - ↓ 16.2 文件
 - ↓ 16.2.1 文件锁定
 - ↓ 16.2.2 传递文件句柄
 - ↓ 16.3 管道
 - ↓ 16.3.1 匿名管道
 - ↓ 16.3.2 自言自语
 - ↓ 16.3.3 双向通讯
 - ↓ 16.3.4 命名管道
 - ↓ 16.4. System V IPC
 - ↓ 16.5. 套接字
 - ↓ 16.5.1 网络客户端程序
 - ↓ 16.5.2 网络服务器
 - ↓ 16.5.3 消息传递

计算机进程之间几乎有和人与人之间一样多的交流。我们不应低估进程间通讯的难度。如果你的朋友只使用形体语言，那么你光注意语言暗示对你是一点用都没有。同样，两个进程之间只有达成了通讯的方法以及建筑在该方法之上的习惯的共识以后才能通讯。和任何通讯一样，这些需要达成共识的习惯的范围从词法到实际用法：几乎是从用什么方言到说话的顺序的一切东西。这些习惯是非常重要的，因为我们都知道如果光有语义而没有环境（上下文），通讯起来是非常困难的。

在我们的方言里，进程间通讯通常念做 **IPC**。Perl 的 **IPC** 设施的范围从极为简单到极为复杂。你需要用哪种设施取决于你要交流的信息的复杂度。最简单的信息几乎就是没有信息：只是对某个时间点发生了某个事件的知晓。在 Perl 里，这样的事件是通过模拟 **Unix** 信号系统的信号机制实现的。

在另外一个极端，Perl 的套接字设施允许你与在互联网上的另外一个进程以任何你们同时都支持的协议进行通讯。自然，自由是有代价的：你必须通过许多步骤来设置连接并且还要确保你和那头的进程用的是同样的语言。这样做的结果就是要求你需要坚持许多其他奇怪的习惯。更准确地说，甚至还要要求你用象 **XML**，**Java**，或 **Perl** 这样的语言讲话。很恐怖。

上面两个极端的中间的东西是一些主要用于在同一台机器上的进程之间进行通讯的设施。包括老派的文件，管道，**FIFO**，和各种 **Sys V IPC** 系统调用。对这些设施的支持因平台的不同而有所差异；现代的 **Unix** 系统（包括苹果的 **Mac OS X**）支持上面的所有设施，但是，除信号和 **Sys V IPC** 以外，**Microsoft** 操作系统支持剩下的所有的，包括管道，进程分裂，文件锁和套接字。（注：除了 **AF_UNIX** 套接字）。

关于移植的更多的一般性信息可以在标准的 Perl 文档集中找到（不管你的系统里是什么格式），他们在 **perlport** 里。与 **Microsoft** 相关的信息可以在 **perlwin32** 和 **perlfork** 里找到，即使在非 **Microsoft** 的系统里都安装了它们。相关的书籍，我们介绍下面的：

1. The Perl Cookbook, Tom Christiansen 和 Nathan Torkington（O'Reilly and Associates,1998），第十六到十八章。
2. Advanced Programming in the UNIX Environment, W. Richard Stevens（Addison-Wesley,1992）
3. TCP/IP Illustrated, W. Richard Stevens, 卷 I-III（Addison-Wesley, 1992-1996）

16.1 信号

Perl 使用一种简单的信号处理模型：在 `%SIG` 散列里包含指向用户定义信号句柄的引用（符号或者硬引用）。某些事件促使操作系统发送一个信号给相关的进程。这时候对应该事件的句柄就会被调用，给该句柄的参数中就有一个包含触发它的信号名字。要想给另外一个进程发送一个信号，你可以用 `kill` 函数。把这个过程想象成是一个给其他进程发送一个二进制位信息的动作。（注：实际上，更有可能是五或者六个二进制位，取决于你的 OS 定义的信号数目以及其他进程是否利用了你不发送别的信号的这个情况。）。如果另外一个进程安装了一个处理该信号的信号句柄，那么如果收到该信号，它能够执行代码。不过发送进程没有任何办法获取任何形式的返回，它只能知道该信号已经合法发送出去了。发送者也接收不到任何接收进程对该信号做的处理的信息。

我们把这个设施归类为 **IPC** 的一种，但实际上信号可以来自许多源头，而不仅仅是其他进程。一个信号也可能来自你自己的进程，或者是用户在键盘上敲入了某种特定键盘序列，比如 **Control-C** 或者 **Control-Z** 造成的，也可能是内核在处理某些特殊事件的时候产生的，比如子进程退出或者你的进程用光堆栈或者达到了文件尺寸或内存的极限等。不过你自己的进程可以很容易区别这些场合。信号就好像一个送到你家门口的没有返回地址的神秘包裹。你打开的时候最好小心一点。

因为在 `%SIG` 里的记录可能是硬链接，所以通常把匿名函数用做信号句柄：

```
$SIG{INT} = sub {die "\nOutta here!\n"};
$SIG{ALRM} = sub { die "Your alarm clock went off" };
```

或者你可以创建一个命名函数，并且把它的名字或者引用放在散列里的合适的槽位里。比如，要截获中断和退出信号（通常和你的键盘的 **Control-C** 和 **Control-** 绑在一起），你可以这样设置句柄：

```
sub catch_zap {
    my $sname = shift;
    our $shucks++;
    die "Somebody sent me a SIG$sname!";
}
$shucks = 0;
$SIG{INT} = 'catch_zap';      # 意思总是 &main::catch_zap
$SIG{INT} = \&catch_zap;     # 最好的方法
$SIG{QUIT} = \&catch_zap;    # 把另外一个信号也捕获上
```

注意，我们在信号句柄里做的所有事情就是设置一个全局变量然后用 `die` 抛出一个例外。如果可能，请力争避免处理比这更复杂的东西，因为在大多数系统上，**C** 库都是不可再入的。信号是异步传送的（注：与 **Perl** 层操作码同步的信号传递安排在以后的版本发布，那样应该能解决信号和核心转储的问题。），所以，如果信号传递后你已经在相关的 **C** 库过程里面了，那么调用任何 `print` 函数（或者只是任何需要 `malloc(3)` 分配更多内存的函数）在理论上都可能触发内存错误并导致内核转储。（甚至可能 `die` 过程也有点有点不安全——除非该进程是在一个 `eval` 里执行的，因为那样会消除来自 `die` 的 I/O，于是就让它无法调用 **C** 库。）

一个更简单的捕获信号的方法是使用 `sigtrap` 用法安装简单的缺省信号句柄：

```
use sigtrap qw(die INT QUIT);
use sigtrap qw(die untrapped normal-signals stack-trace any error-signals);
```

如果你嫌写自己的句柄麻烦，那就可以用这个用法，不过你仍然会希望捕获危险的信号并且执行一个正常的关闭动作。缺省时，这些信号中的一部分对你的进程是致命的，当的程序收到这样的信号时只能停止。糟糕的是，这也意味着不会调用任何用做退出控制的 `END` 函数和用于对象终止的 `DESTROY` 方法。但是它们在正常的 **Perl** 例外中的确是被调用的（比如你调用 `die` 的时候），所以，你可以用这个用法无痛地把信号转换成例外。甚至在你没有自己处理这些信号的情况下，你的程

序仍然能够表现正确。参阅第三十一章，用法模块，里 `use sigtrap` 的描述获取这个用法的更详细的特性。

你还可以把 `%SIG` 句柄设置为字符串“IGNORE”或者“DEFAULT”，这样，Perl 就会试图丢弃该信号或者允许用缺省动作处理该信号（不过有些信号既不能捕获，也不能忽略，比如 KILL 和 STOP 信号；如果手边有资料，你可以参阅 `signal(3)`，看看你的系统可以用的信号列表和它们的缺省行为。）

操作系统认为信号是一个数字，而不是一个名字，但是 Perl 和大多数人一样，喜好符号名字，而讨厌神秘的数字。如果想找出信号的名字，你可以把 `%SIG` 散列里的键字都列出来，或者如果你的系统里有 `kill` 命令，你可以用 `kill -l` 把它们列出来。你还可以使用 Perl 标准的 `Config` 模块来检查你的操作系统的信号名字和信号数字之间的映射。参阅 `Config(3)` 获取例子。

因为 `%SIG` 是一个全局的散列，所以给它赋值将影响你的整个程序。如果你把信号捕获局限于某个范围，可能对你的程序的其他部分更有好处。实现这个方法是用一个 `local` 信号句柄赋值，这样，一旦退出了闭合的语句块，那么该句柄就失去作用了。（但是要记住，`local` 变量对那些语句块中调用的函数是可见的。）

```
{
    local $SIG{INT} = 'IGNORE';
    ...    # 处理你自己的业务，忽略所有的信号
    fn();  # 在 fn() 里也忽略信号！
    ...    # 这里也忽略。
}          # 语句块退出后恢复原来的 $SIG{INT} 值。

fn();      # 在（假设的） fn() 里没有忽略 SIGINT
```

16.1.1 给进程组发信号

（至少在 Unix 里，）进程是组织成进程组的，一起对应一个完整的任务。比如，如果你运行了单个 `shell` 命令，这条命令是有一系列过滤器命令组成，相互之间用管道传递数据，这些进程（以及它们的子进程）都属于同一个进程组。该进程组有一个数字对应这个进程组的领头进程的进程号。如果你给一个正数的进程号发送信号，该信号只发送给该进程，而如果你给一个负数进程号发送信号，那么该信号将发送给对应的进程组的所有进程，该进程组的进程组号就是这个负数的绝对值，也就是该进程组领头进程的进程号。（为了方便进程组领头进程，进程组 ID 就是 `$$`。）

假设你的程序想给由它直接启动的所有子进程（以及由那些子进程启动的孙子进程和曾孙进程等）发送一个挂起信号。实现这个方法的方法是：你的程序首先调用 `setpgrp(0,0)`，使自己成为新的进程组的领头进程，这样任何它创建的进程都将成为新进程组的一部分。不管那些进程是通过 `fork` 手工启动的还是通过管道 `open` 打开的或是用 `system("cmd &")` 启动的后台进程。即使那些进程有自己的子进程也无所谓，只要你给你的整个进程组发送挂起信号，那么就会把它们都找出来（除了那些设置了自己的进程组或者改变了自己的 `UID` 的进程——它们对你的信号有外交豁免权。）

```
{
    local $SIG{HUP} = 'IGNORE';    # 排除自己
    kill(HUP, -$$);                # 通知自己的进程组
}
```

另外一个有趣的信号是信号数 0。它实际上不影响目标进程，只是检查一下，看看那个进程是否还活着或者是是否改变了 `UID`。也就是说，它判断给目标进程发送信号是否合法，而实际上并不真正发送信号。

```
unless ( kill 0 => $kid_pid ) {
    warn "something wicked happened to $kid_pid";
}
```

```
}
```

信号 **0** 是唯一的一个在 **Unix** 上和 **Windows** 上的 **Perl** 移植作用一样的信号。在 **Microsoft** 系统里，**kill** 实际上并不发送信号。相反，它强迫目标进程退出，而退出状态由信号数标明。这些东西以后都会修改。但是，神奇的 **0** 信号将依然如故，表现出非破坏性的特性。

16.1.2 收割僵死进程

当一个进程退出的时候，内核向它的父进程发送一个 **CHLD** 信号然后该进程就成为一个僵死进程（**zombie**，注：这是一个技术术语），直到父进程调用 **wait** 或者 **waitpid**。如果你在 **Perl** 里面启动新进程用的不是 **fork**，那么 **Perl** 就会替你收割这些僵死进程，但是如果你用的是一个 **fork**，那么就得自己做清理工作。在许多（但不是全部）内核上，自动收割的最简单办法就是把 **\$SIG{CHLD}** 设置为 **'IGNORE'**。另一个更简单（但也更乏味）的方法是你自己收割它们。因为在你开始处理的时候，可能不止一个子进程已经完蛋了，所以，你必须在一个循环里收割你的子进程直到没有更多为止：

```
use POSIX ":sys_wait_h";
sub REAPER { 1 until waitpid(-1, WNOHANG) == -1 }
```

想根据需要运行这些代码，你要么可以给它设置 **CHLD** 信号：

```
$SIG{CHLD} = \&REAPER;
```

或者如果你的程序是在一个循环里运行，那你只需要循环调用收割器就行了。这个方法最好，因为它避免了那些信号可能触发的在 **C** 库里偶然的核心转储。但是，如果在一个很快速的循环里调用，这样做的开销是巨大的，所以一种合理的折衷是用一种混合的方法：你在句柄里尽可能少做处理，把风险降到最低，同时在外循环中等待收割僵死进程：

```
our $zombies = 0;
$SIG{CHLD} = sub { $zombies++; };
sub reaper {
    my $zombie;
    our %Kid_Status;    # 存储每个退出状态
    $zombies = 0;
    while (($zombie = waitpid(-1, WNOHANG)) != -1) {
        $Kid_Status{$zombie} = $?;
    }
}
while(1) {
    reaper() if $zombies;
    ...
}
```

这段代码假设你的内核支持可靠信号。老的 **Sys V** 风格的信号是不可靠的，那样的话，想写正确的信号句柄几乎是不可能的。甚至早在 **Perl** 版本 **5.003**，只要可能，我们就开始使用 **sigaction(2)** 系统调用了，因为它更可靠些。这意味着除非你在一个古老的操作系统上运行或者跑的是一个古老的 **Perl**，你用不着重新安装你的句柄，也不会冒丢失信号的危险。幸运的是，所有带 **BSD** 风格的系统（包括 **Linux**，**Solaris**，和 **Mac OS X**）以及所有 **POSIX** 兼容的系统都提供可靠的信号，所以那些老旧的 **Sys V** 问题更多是历史遗留问题，而不是目前我们要关心的问题。

在新内核上，还有许多其他东西也会运行得更好些。比如，“慢的”系统调用（那种可以阻塞的，就象 **read**，**wait**，和 **accept**）如果被一个信号中断后将自动重新启动。在那些灰暗的旧社会里，用户代码必须记得明确地检查每个慢的系统调用是否带着 **#! (\$ERRNO)** 为 **EINTR** 失败的，而且如果是这样，那么重起。而且这样的情况不光对 **INT** 信号，而且对有些无辜的信号，比如 **TSTP**（来自 **Control-Z**）或者 **CONT**（来自把任务放到前台）也会退出系统调用。如果操作系统允许，现在

Perl 自动为你重新启动系统调用。我们通常认为这是一个特性。

你可以检查一下，看看你的系统是否有更严格的 POSIX 风格的信号，方法是装载 Config 模块然后检查 `$Config{d_sigaction}` 是否为真。要检查慢的系统调用是否可以可以重起，检查你的系统的文档：`sigaction(2)` 或者 `sigvec(3)`，或者在你的 C `sys/signal.h` 里查找 `SV_INTERRUPT` 或者 `SA_RESTART`。如果找到两个或者其中之一，你可能就拥有可重起的系统调用。

16.1.3 给慢速操作调速

信号的一个用途就是给长时间运行的操作设置一个时间限制。如果你用的是一种 Unix 系统（或者任何 POSIX 兼容的支持 ALRM 信号的系统），你就可以让内核在未来的某时刻给你进程发送一个 ALRM 信号：

```
use Fcntl ':flock';
eval {
    local $SIG{ALRM} = sub { die "alarm clock restart" };
    alarm 10;          # 安排10秒后报警
    eval {
        flock(FH, LOCK_EX)    # 一个阻塞的，排它锁
        or die "can't flock:$!";
    };
    alarm 0;           # 取消报警
};
alarm 0;              # 避免冲突条件
die if $@ && $@ !~ /alarm clock restart/;  # 重新启动
```

如果你在等待锁的时候报警，你只是把信号缓冲起来然后返回，你会直接回到 `flock`，因为 Perl 在可能的情况下会自动重起系统调用。跳出去的唯一方法是用 `die` 抛出一个例外并且让 `eval` 捕获之。

（这样做是可行的，因为例外会退回到调用库的 `longjmp(3)` 函数，而 `longjmp(3)` 是真正把你带出重起系统调用的东西。）

我们使用了嵌套的例外陷阱是因为如果 `flock` 在你的平台上没有实现的话，那么调用 `flock` 会抛出一个例外，因此你必须确保清理了警告信号。第二个 `alarm 0` 用于处理这样的情况：信号到达时是在调用 `flock` 之后但是在到达第一个 `alarm 0` 之前。没有第二个 `alarm`，你可能会面对一个很小的冲突条件——不过冲突条件可不会管你的冲突条件是大是小；它们是黑白分明的：要么有，要么无。而我们更希望没有。

16.1.4 阻塞信号

有时候，你可能想在一些关键的代码段里推迟接收信号。你并不想简单地忽略这些信号，只是你做的事情太关键了，因而不能中断。Perl 的 `%SIG` 散列并不实现信号阻塞，但是 POSIX 模块通过它的调用 `sigprocmask(2)` 系统调用的接口实现了信号阻塞：

```
use POSIX qw(:signal_h);
$sigset = POSIX::SigSet->new;
$blockset = POSIX::SigSet->new(SIGINT, SIGQUIT, SIGCHLD);
sigprocmask(SIG_BLOCK, $blockset, $sigset)
or die "Could not block INT, QUIT, CHLD signals: $! \n";
```

一旦上面三个信号都被阻塞了，你就可以毫不担心地执行你的任务了。在你处理完你的关键业务以后，用恢复旧的信号掩码的方法取消信号的阻塞：

```
sigprocmask( SIG_SETMASK, $sigset)
or die "Could not restore INT, QUIT, CHLD signals: $!\n";
```


如果阻塞的时候有三个信号中的任何信号到达，那么这时它们会被立即发送。如果有两种或者更多的不同信号在等待，那么它们的发送顺序并没有定义。另外，在阻塞过程中，收到某个信号一次和收到多次是没有区别的。（注：通常是这样。根据最新的规范，可计数信号可能在一些实时系统上有实现，但是我们还没有看到那些系统。）比如，如果你在阻塞 **CHLD** 信号期间有九个子进程退出，那么你的信号句柄（如果存在）在退出阻塞后仍然只会被调用一次。这就是为什么当你在收割僵死进程的时候，你应该循环到所有的僵死进程都消失。

16.2 文件

你以前可能从未把文件当作一种 **IPC** 机制，但是它们却占据了进程间通讯的很大一部分份额——远比其他方法的总和份额要大。当一个进程把它的关键数据存放在文件里，而且以后另外一个进程又检索那些数据，那么这就是两个进程在通讯。文件提供了一些这里提到的其他的 **IPC** 机制所没有的特点：就象纸张埋在地下几千年一样，文件可以比它的作者有更长的保存期。（注：假设我们有人保存期）。再加上相对而言使用的简单，文件至今仍然流行就一点都不奇怪了。

使用文件在已经消亡的过去和不知何时的未来之间传递信息并不让人奇怪。你在一些永久介质上（比如磁盘）写文件就行了。仅此而已。（如果它包含 **HTML**，你可能还要告诉一台 **web** 服务器在那里能找到的。）有趣的问题是如果所有当事人都健在并且试图相互通讯时该怎么办。如果对各自说话的顺序没有一些规定的话，就根本不可能有可靠的交流；这样的规定可以通过文件锁来实现，我们将在下一节介绍。在其后一节里，我们将讨论父进程和其子进程之间存在的特殊关系，这些关系可以让相关的当事人通过对相同文件继承的访问交换信息。

文件当然有其缺点，比如远程访问，同步，可靠性和会话管理等。本章其他节介绍那些着眼于解决这些问题的不同 **IPC** 机制。

16.2.1 文件锁定

在一个多任务环境里，你需要很小心地避免与其他试图使用你正在用的文件的进程冲突。如果所有进程都只读取文件内容，那么大家相安无事，但是如果有哪怕只有一个进程需要写该文件，那么随后就会发生混乱——除非使用某种排序机制充当交通警察的角色。

绝对不要只是使用文件是否存在（也就是 **-e \$file**）当作文件锁的标志，因为在测试文件名是否存在和你计划的处理（比如创建，打开，或者删除它）之间存在冲突条件。参阅第二十三章，安全，中的“处理冲突条件”，获取更多相关信息。

Perl 的可移植锁定接口是 **flock(HANDLE,FLAGS)** 函数，在第二十九章，函数，里描述。Perl 只采用那些在最广范围的平台上都能找到的最简单的锁定机制，因此获得了最大的可移植性。这些语意简单得可以在绝大部分系统上使用，包括那些不支持这些传统系统调用的平台，比如 **System V** 或 **Windows NT**。（如果你运行的 **Microsoft** 的系统是早于 **NT** 的平台，那么你很可能没有这些系统调用支持，就好像你运行 **Mac OS X** 以前的苹果系统一样。）

锁有两种变体，共享（**LOCK_SH** 标志）和排它（**LOCK_EX** 标志）。尽管听着有“排它”的意思，但是进程并不需要服从对文件的锁。也就是说，**flock** 只是实现了劝告性的锁定，劝告性的锁定，也就意味着锁定一个文件并不阻止其他的进程读取甚至是写入该文件。进程请求一个排它锁只是让操作系统推迟它对文件的处理，直到所有当前的锁持有者，不管是共享锁还是排它锁，都完成操作以后才进行。类似地，如果一个进程请求一个共享锁，它只是推迟处理直到没有排它锁存在。只有所有当事人都使用文件锁机制的时候，你才能安全地访问一个有内容的文件。

因此，**flock** 缺省时是一个阻塞操作。也就是说，如果你不能立即获取你需要的锁，操作系统会推迟你的处理，直到你能够获得锁为止。下面是如何获取阻塞的共享锁的方法，通常用于读取文件：

```
use Fcntl qw(:DEFAULT :flock);
```

```
open(FH, "< filename") or die "can't open filename: $!";
flock(FH, LOCK_SH) or die "can't lock filename: $!";
# 现在从 FH 里读取
```

你可以试图请求一个非阻塞的锁，只需要在 `flock` 请求里加入 `LOCK_NB` 标志就可以了。如果你不能马上获得锁，那么该函数失败并且马上返回假。下面是例子：

```
flock(FH, LOCK_FH | LOCK_NB)
or die "can't lock filename: $!";
```

你除了象我们这样抛出一个例外之外可能还想做点别的事情，但是你肯定不敢对该文件进行任何 I/O 操作。如果你的锁申请被拒绝，你就不应该访问该文件直到你能够拿到锁。谁知道那个文件处于什么样的混乱状态？非阻塞模式的主要目的是让你离开并且在等待期间做些其他的事情。而且它也可以用于生成更友好的交互，比如警告用户说他可能要一段时间才能获取锁，这样用户就不会觉得被抛弃：

```
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename") or die "can't open filename: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
    local $| = 1;
    print "Waiting for lock on filename...";
    flock(FH, LOCK_SH) or die "can't lock filename: $!";
    print "got it.\n";
}
# 现在从 FH 读数
```

有些人会试图把非阻塞锁放到一个循环里去。非阻塞锁的主要问题是，当你回过头来再次检查的时候，可能其他人已经把锁拿走了，因为你放弃了在队伍里的位置。有时候你不得不排队并且等待。如果你走运的话，可能可以先看看杂志什么的。

锁是针对文件句柄的，而不是文件名。（注：实际上，锁不是针对文件句柄的——他们是针对与文件句柄关联的文件描述符的，因为操作系统并不知道文件句柄。这就意味着我们的所有关于对某文件名没能拿到锁的 `die` 消息从技术上来讲都是不准确的。不过下面这样的错误信息：“**I can't get a lock on the file represented by the file descriptor associated with the filehandle originally opened to the path filename, although by now filename may represent a different file entirely than our handle does**” 只能让用户糊涂（更不用说读者了）。当你关闭一个文件，锁自动消除，不管你是通过调用 `close` 明确关闭该文件还是通过重新打开该句柄隐含的关闭还是退出你的进程。

如果需要获取排它锁（通常用于写），你就得更小心。你不能用普通的 `open` 来实现这些；如果你用 `<` 的打开模式，如果文件不存在那么它会失败，如果你用 `>`，那么它会先删除它处理的任何文件。你应该使用 `sysopen` 打开文件，这样该文件就可以在被覆盖之前先被锁住。一旦你安全地打开了用于写入的文件（但还没有写），那么先成功获取排它锁，只有这时候文件才被截去。现在你可以用新数据覆盖它。

```
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "filename", O_WRONLY | O_CREAT)
or die "can't open filename:$!";
flock(FH, LOCK_EX)
or die "can't lock filename:$!";
truncate(FH, 0)
or die "can't truncate filename:$!";
# 现在写 FH
```

如果想现场修改文件的内容，那么再次使用 `sysopen`。这次你请求读写权限，如果必要就创建文件。一旦打开了文件，但是还没有开始读写的时候，先获取排它锁，然后在你的整个事务过程中都使用

它。释放锁的最好方法是关闭文件，因为那样保证了在释放锁之前所有缓冲区都写入文件。

一次更新包括读进旧值和写出新值。你必须在单个排它锁里面做两个操作，减少其他进程在你处理之后（甚至之前）读取（马上就不正确了的）数值。（我们将在本章稍后的共享内存的部分再次介绍这个情况。）

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "counterfile", O_RDWR | O_CREAT)
or die "can't open counterfile: $!";
flock(FH, LOCK_EX);
or die "can't write-lock counterfile: $!";
$counter = <FH> || 0;    # 首先应该 undef
seek(FH, 0, 0)
or die "can't rewind counterfile: $!";
print FH $counter+1, "\n"
or die "can't write counterfile: $!";

# 下一行在这个程序里从技术上是肤浅的，但是一个一般情况下的好主意
truncate(FH, tell(FH))
or die "can't truncate counterfile: $!";
close(FH)
or die "can't close counterfile: $!";
```

你不能锁住一个你还没打开的文件，而且你无法拥有一个施加于多个文件的锁。你能做的是用一个完全独立的文件充当某种信号灯（象交通灯），通过在这个信号灯文件上使用普通的共享和排它锁来提供可控制的对其他东西（文件）的访问。这个方法有几个优点。你可以用一个文件来控制对多个文件的访问，从而避免那种一个进程试图以一种顺序锁住那些文件而另外一个进程试图以其他顺序锁住那些文件导致的死锁。你可以用信号灯文件锁住整个目录里的文件。你甚至可以控制对那些就不在文件系统上的东西的访问，比如一个共享内存对象或者是一个若干个预先分裂出来的服务器准备调用 **accept** 的套接字。

如果你有一个 **DBM** 文件，而且这个 **DBM** 文件没有明确的锁定机制，那么用一个附属的锁文件就是控制多个客户并发访问的最好的方法。否则，你的 **DBM** 库的内部缓冲就可能与磁盘上的文件之间丢失同步。在调用 **dbmopen** 或者 **tie** 之前，先打开并锁住信号灯文件。如果你用 **O_RDONLY** 打开数据库，那你会愿意使用 **LOCK_SH** 处理锁定。否则，使用 **LOCK_EX** 用于更新数据库的排它访问。（同样，只有所有当事人都同意关注信号灯才有效。）

```
use Fcntl qw(:DEFAULT :flock);
use DB_File;    # 只是演示用途，任何 db 都可以

$DBNAME = "/path/to/database";
$LCK = $DBNAME. ".lockfile";

# 如果你想把数据写到锁文件里，使用 O_RDWR
sysopen(DBLOCK, $LCK, O_RDONLY| O_CREAT)
or die "can't open $LCK:$!";

# 在打开数据库之前必须锁住文件
flock(DBLOCK, LOCK_SH)
or die "can't LOCK_SH $LCK: $!";

tie(%hash, "DB_File", $DBNAME, O_RDWR | O_CREAT)
or die "can't tie $DBNAME: $!";
```


现在你可以安全地对捆绑了的 `%hash` 做任何你想做的处理了。如果你完成了对你的数据库的处理，那么确保你明确地释放了那些资源，并且是以你请求它们的相反的顺序：

```
untie %hash;    # 必须在锁定文件之前关闭数据库
close DBLOCK;   # 现在可以安全地释放锁了
```

如果你安装了 GNU DBM 库，你可以使用标准的 `GDBM_File` 模块的隐含锁定。除非最初的 `tie` 包含 `GDBM_NOLOCK` 标志，否则该库将保证任意时刻只有一个用户可以写入 `GDBM` 文件，而且读进程和写进程不能让数据库同时处于打开状态。

16.2.2 传递文件句柄

每当你用 `fork` 创建一个子进程，那个新的进程就从它的父进程继承所有打开了的文件句柄。用文件句柄做进程间通讯可以很容易先通过使用平面文件来演示。理解文件机制的原理对于理解本章后面的管道和套接字等更奇妙的机制有很大帮助。

下面这个最简单的例子打开一个文件然后开始一个子进程。然后子进程则使用已经为它打开了的文件句柄：

```
open(INPUT, "< /etc/motd")    or die "/etc/motd: $!";
if ($pid = fork) { waitpid($pid, 0); }
else {
    defined($pid)    or die "fork: $!";
    while (<INPUT>) { print "$.: $_" }
    exit;           # 不让子进程回到主代码
}
# INPUT 句柄现在在父进程里位于 EOF
```

一旦用 `open` 获取了文件的访问权，那么该文件就保持授权状态直到该文件句柄关闭；对文件的权限或者所有人的访问权限对文件的访问没有什么影响。即使该进程后面修改它自己的用户或者组 `ID`，或者该文件已经把自己的所有权赋予了一个不同的用户或者组，也不会影响已经打开的文件句柄。那些运行在提升权限级别的程序（比如 `set-id` (`SID`) 程序或者系统守护进程）通常在它们提升以后的权限下打开一个文件，然后把文件句柄传递给一个不能自己打开文件的子进程。

尽管这个机制在有意识地使用的时候非常便利，但如果文件句柄碰巧从一个程序漏到了另外一个程序，那么它就有可能导致安全问题。为避免给所有可能的文件句柄赋予隐含的访问权限，当你明确地用 `exec` 生成一个新的程序或者通过调用一个透过管道的 `open`，`system`，或者 `qx//`（反钩号）隐含地执行一个新程序的时候，`Perl` 都会自动关闭任何他已经打开的文件句柄（包括管道和套接字）。`STDIN`，`STDOUT`，和 `STDERR` 被排除在外，因为他们的主要目的是提供程序之间的联系。所以将文件句柄传递给一个新程序的方法之一是把该文件句柄拷贝到一个标准文件句柄里：

```
open(INPUT, "< /etc/motd")    or die "/etc/motd: $!";
if ($pid = fork) { wait }
else {
    defined($pid)    or die "fork: $!";
    open(STDIN, "<&INPUT")    or die "dup: $!";
    exec("cat", "-n")        or die "exec cat: $!";
}
```

如果你真的希望新程序能够获取除了上面三个之外的文件句柄的访问权限，你也能做到，不过你必须做两件事之一。当 `Perl` 打开一个新文件（或者管道和套接字）的时候，它检查变量 `$^F` (`$SYSTEM_FD_MAX`) 的当前设置。如果新文件句柄用的数字文件描述符大于那个 `$^F`，该描述符就标记为一个要关闭的。否则，`Perl` 就把它放着，并且你 `exec` 出来的新的程序就会继承访问。

通常很难预料你新创建的文件句柄是什么，但是你可以在 `open` 期间暂时把你的最大系统文件描述符数设置的非常大：

```
# 打开文件并且把 INPUT 标记为在 exec 之间可以使用
{
    local $^F = 10_000;
    open(INPUT, "< /etc/motd") or die "/etc/motd: $!";
} # 在范围退出后，恢复旧的 $^F 值
```

现在你所要干的就是让新程序关照你刚刚打开的文件句柄的文件描述符。最干净的解决方法（在那些支持这些的系统上）就是传递一个文件名是刚创建的文件描述符的特殊文件。如果你的系统有一个目录叫 `/dev/fd` 或者 `/proc/$$/fd`，里面包含从 0 到你的系统支持的最大文件描述符数字，那么你就可能可以使用这个方法。（许多 Linux 系统两个都有，但是好象只有 `/proc` 的版本是正确填充的。BSD 和 Solaris 喜欢用 `/dev/fd`。你最好自己看看你的系统，检查一下你的系统是哪种情况。）首先，用我们前面显示的代码打开一个文件句柄并且把它标记成一个可以在 `exec` 之间传递的句柄，然后用下面的方法分裂进程：

```
if ($pid = fork) { wait }
else {
    defined($pid) or die "fork: $!";
    $fdfile = "/dev/fd/" . fileno(INPUT);
    exec("cat", "-n", $fdfile) or die "exec cat: $!";
}
```

如果你的系统支持 `fcntl` 系统调用，你就可以手工骗过文件句柄在 `exec` 时候的关闭标志了。如果你创建了文件句柄而且还想与你的子进程共享，但是早些时候你还没有意识到想共享句柄的场合下，这个方法特别方便。

```
use Fcntl qw/F_SETFD/;

fcntl( INPUT, F_SETFD, 0)
or die "Can't clear close-on-exec flag on INPUT: $!\n";
```

你还可以强制一个文件句柄关闭：

```
fcntl(INPUT, F_SETFD, 1)
or die "Can't set close-on-exec flag on INPUT: $!\n";
```

你还可以查询当前状态：

```
use Fcntl qw/F_SETFD F_GETFD/;

printf("INPUT will be %s across exec\n",
fcntl(INPUT, F_GETFD, 1) ? "closed" : "left open");
```

如果你的系统不支持文件系统中的文件描述符名字，而你又不想通过 `STDIN`，`STDOUT`，或者 `STDERR` 传递文件句柄，你还是可以实现，但是你必须给那些程序做特殊的安排。常用的策略是用一个环境变量或者一个命令行选项传递这些描述符号。

如果被执行的程序是用 Perl 写的，你可以用 `open` 把一个文件描述符转成一个文件句柄。这次不是声明文件名，而是用 `"&="` 后面跟着描述符号。

```
if (defined($ENV{input_fdno}) && $ENV{input_fdno}) =~ /\^d$/ {
    open(INPUT, "<&=$ENV{input_fdno}")
    or die "can't fdopen $ENV{input_fdno} for input: $!";
}
```

如果你准备运行的 Perl 子过程或者程序需要一个文件名参数，那么事情就更好办了。你可以利用 Perl 的普通 `open` 函数（不是 `sysopen` 或者三个参数的 `open`）的描述符打开打开特性来自动化这些动作。假如你有一个象下面的简单 Perl 程序：

```
#!/usr/bin/perl -p
# nl - 输入的行数
printf "%6d ", $.;
```

再假设你已经安排好让 `INPUT` 句柄在 `exec` 之间保持打开状态，你可以这样调用这个程序：

```
$fdspec = '<&=' . fileno(INPUT);
system("nl", $fdspec);
```

或者捕获输出：

```
@lines = `nl '$fdspec' 1;` # 单引号保护 spec 不被 shell 代换
```

不管你是否 `exec` 另外一个程序，如果你使用一个从 `fork` 继承过来的文件描述符，那么就有一个小收获。和 `fork` 拷贝的变量不同的是（那些变量总是复制为相同的是独立的变量），文件描述符在两个进程之间就是同一个。如果一个进程从该句柄中读取数据，那么另一个进程的文件指针（文件位置）也跟着前进，并且两个进程都不能再看到那些数据了。如果它们轮流读取，那么它们就会在文件里相互跳跃。这个特性对于附着在串行设备上的句柄，象管道或者套接字等而言非常直观，因为它们多数是只读设备，里面的数据也是短时存在的。但是磁盘文件的这个特性可能会让你觉得奇怪。如果这是一个问题，那么在进程分裂之后重新打开任何需要独立跟踪的文件。

`fork` 操作符是源自 `Unix` 的概念，这就意味着它可能不能在所有非 `Unix`/非 `POSIX` 平台上正确实现。尤其是，在 `Windows 98`（或者更新的版本）上，你只有运行 `Perl 5.6` 或者更新的版本才能在这些 `Microsoft` 系统上使用 `fork`。尽管 `fork` 在这些系统上是通过同一个程序里的多个并发的执行流实现的，但它也不是那些缺省时共享所有数据的线程；在 `fork` 里，只有文件描述符是共享的。又见第十七章，线程。

16.3 管道

管道是一个无方向性的 `I/O` 通道，它可以从一个程序向另外一个传递字节流。管道分命名管道和匿名管道两种。你可能对匿名管道更熟悉，所以我们先介绍它。

16.3.1 匿名管道

如果你给 `open` 的第二个参数后缀或者前缀一个管道符号，那么 Perl 会给你打开一个管道而不是一个文件。然后剩下的参数就成了一条命令，这条命令会被 Perl 解释成一个进程（或者一个进程集），而你则想从这条命令中输入或者取出数据流。下面就是如何启动一个你想给它写入的子进程的方法：

```
open SPOOLER, "| cat -v | lpr -h 2>/dev/null"
or die "can't fork: $!";
local $SIG{PIPE} = sub {die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER or die "bad spool: $! $?";
```

这个例子实际上启动了两个进程，我们可以直接向第一个（运行 `cat`）打印。第二个进程（运行 `lpr`）则接收第一个进程的输出。在 `shell` 编程里，这样的技巧通常称为流水线。一个流水线在一行里可以有任意个进程，只要中间的那个明白如何表现得象过滤器；也就是说，它们从标准输入读取而写到标准输出。

如果一条管道命令包含 **shell** 照看的特殊字符，那么 **Perl** 使用你的缺省系统 **shell**（在 **Unix** 上 **/bin/sh**）。如果你只启动一条命令，而且你不需要--或者是不想--使用 **shell**，那么你就可以用打开管道的一个多参数的形式替代：

```
open SPOOLER, "|-", "lpr", "-h"    # 要求 5.6.1
or die "can't run lpr: $!";
```

如果你重新打开你的程序的标准输出作为到另外一个程序的管道，那么你随后向 **STDOUT print** 的任何东西都将成为那个新程序的标准输入。因此如果你想给你的程序做成分页输出（注：也就是每次显示一频，而不是希里哗啦一堆），你可以：

```
if (-t STDOUT) {                # 只有标准输出是终端时
    my $pager = $ENV{PAGER} || 'more';
    open( STDOUT, "| $pager")    or die "can't fork a pager: $!";
}
END {
    close(STDOUT)                or die "can't close STDOUT: $!"
}
```

如果你向一个与管道连接的文件句柄写入数据，那么在你完成处理之后，你需要明确地 **close** 它。那样你的主程序才不会在它的后代之前退出。

下面就是如何启动一个你想读取数据的子进程的方法：

```
open STATUS, "netstat -an 2>/dev/null |"
or die "can't fork: $!";
while (<STATUS>) {
    next if /^(tcp|udp)/;
    print;
}
close STATUS or die "bad netstat: $! $?";
```

你同样也可以象用在输出里那样，打开一个多命令的输入管道。而且和以前一样，你可以通过使用一个可选的 **open** 形式避免 **shell**：

```
open STATUS, "-|", "netstat", "-an"    # 需要 5.6.1
or die "can't runnetstat: $!";
```

不过那样你就得不到 **I/O** 重定向，通配符扩展或者多命令管道，因为 **Perl** 得靠你的 **shell** 做这些事情。

你可能已经注意到你可以使用反钩号实现与打开一个管道读取数据一样的功能：

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;
```

尽管反钩号很方便，但是它们必须把所有东西都一次读进内存，所以，通常你打开自己的管道文件句柄然后每次一行或者一条记录地处理文件会更有效些。这样你就能对整个操作有更好的控制，让你可以提前杀死进程。你甚至还可以一边接收输入一边处理，这样效率更高，因为当有两个或者更多进程同时运行的时候，计算机可以插入许多操作。（即使在一台单 **CPU** 的机器上，输入和输出也可能在 **CPU** 处理其他什么事的时候发生。）

因为你正在并行运行两个或者更多进程，那么在 **open** 和 **close** 之间的任何时刻，子进程都有可能遭受灾难。这意味着父进程必须检查 **open** 和 **close** 两个的返回值。只检查 **open** 是不够安全的，因为它只告诉你进程分裂是否成功，以及（可能还有）随后的命令是否成功启动（只有在最近的版本的 **Perl** 中才能做到这一点，而且该命令还必须是通过直接分裂的子进程执行的，而不是通过 **shell** 执

行的)。任何在那以后的灾难都是从子进程向父进程以非零退出状态返回的。当 `close` 函数看到这个返回值，那么它就知道要返回一个假值。表明实际的状态应该从 `$?($CHILD_ERROR)` 变量里读取。因此检查 `close` 的返回值和检查 `open` 的返回值一样重要，如果你往一个管道里写数据，那么你还应该准备处理 `PIPE` 信号，如果你还没有完成数据的发送，而另外一端的进程完蛋掉，那么系统就会给你发送这个信号。

16.3.2 自言自语

`IPC` 的另外一个用途就是让你的程序和自己讲话，就象自言自语一样。实际上，你的进程通过管道和一个它自己分裂的拷贝讲话时，它的工作方式和我们上一节里讲的用管道打开很类似，只不过是子进程继续执行你的脚本而不是其他命令。

要想把这个东西提交给 `open` 函数，你要使用一个包含负号的伪命令。所以 `open` 的第二个参数看起来就象 `"|"` 或者 `"|-"`，取决于你是想从自己发出数据还是从自己接收数据。和一个普通的 `fork` 命令一样，`open` 函数在父进程里返回子进程的进程 `ID`，而在子进程里返回 `0`。另外一个不对称的方面是 `open` 命名的文件句柄名字只在父进程里使用。管道的子进程端根据实际情况要么是挂在 `STDIN` 上要么是 `STDOUT` 上。也就是说，如果你用 `|-` 打开一个“输出到”管道，那么你就可以向你打开的这个文件句柄写数据，而你的子进程将在它的 `STDIN` 里找到这些数据：

```
if (open(TO, "|-")) {
    print TO $fromparent;
}
else {
    $tochild = <STDIN>;
    exit;
}
```

如果你用 `-|` 打开一个“来自”管道，那么你可以从这个文件句柄读取数据，而那些数据就是你的子进程往 `STDOUT` 写的：

```
if (open(FROM, "-|" )) {
    $stoparent = <FROM>;
}
else {
    print STDOUT $fromchild;
    exit
}
}
```

这个方法的一个常见的应用就是当你想从一个命令打开一个管道的时候绕开 `shell`。你想这么干的原因可能是你不希望 `shell` 代换任何你准备传递命令过去的文件名里的元字符吧。如果你运行 `Perl 5.6.1` 或者更新的版本，你可以利用 `open` 的多参数形式获取同样的结果。

使用分裂的文件打开的原因是为了在一个假想的 `UID` 或 `GID` 下也能打开一个文件或者命令。你 `fork` 出来的子进程会抛弃任何特殊的访问权限，然后安全地打开文件或者命令，然后充当一个中介者，在它的更强大的父进程和它打开的文件或命令之间传递数据。这样的例子可以在第二十三章的“在有限制的权限下访问命令和文件”节找到。

分裂的文件打开的一个创造性的用法是过滤你自己的输出。有些算法用两个独立的回合来实现要远比用一个回合实现来得简单。下面是一个简单的例子，我们通过把自己的正常输出放到一个管道里模拟 `Unix` 的 `tee(1)` 程序。在管道的另外一端的代理进程（我们自己的子过程之一）把我们的输出分发到所声明的所有文件中。

```
tee("/tmp/foo", "/tmp/bar", "/tmp/glarch");

while(<>) {
```



```

    print "$ARGV at line $. => $_";
}

close(STDOUT)    or die "can't close STDOUT:$!";

sub tee {
    my @output = @_;
    my @handles = ();
    for my $path (@output) {
        my $fh;      # open 会填充这些
        unless (open ($fh, ">", $path)) {
            warn "cannot write to $path: $!";
            next;
        }
        push @handles, $fh;
    }

    # 在父进程的 STDOUT 里重新打开并且返回
    return if my $pid = open(STDOUT, "|-");
    die "cannot fork: $!" unless defined $pid;

    # 在子进程里处理 STDIN
    while(<STDIN>) {
        for my $fh (@handles) {
            print $fh $_ or die "tee output failed:$!";
        }
    }

    for my $fh (@handles) {
        close($fh) or die "tee closing failed: $!";
    }
    exit;      # 不让子进程返回到主循环!
}

```

你可以不停地重复使用这个技巧，而且在你的输出流上放你希望的任意多的过滤器。只要不停调用那个分裂打开 **STDOUT** 的函数，并且让子进程从它的父进程（它认为是 **STDIN**）里读取数据，然后把消息输出给流里面的下一个函数。

这种利用分裂后打开文件的自言自语的另外一个有趣的应用是从一个糟糕的函数中捕获输出，那些函数总是把它们的结果输出到 **STDOUT**。假设 Perl 只有 **printf** 但是没有 **sprintf**。那么你需要的就是类似反钩号那样的东西，但却是 Perl 的函数，而不是外部命令：

```

badfunc("arg");      # TMD, 跑!
$string = forksub(\&badfunc, "arg");  # 把它当作字符串捕获
@lines = forksub(\&badfunc, "arg");  # 当作独立的行

sub forksub {
    my $kidpid = open my $self, "-|";
    defined $kidpid    or die "cannot fork: $!";
    shift->(@_), exit    unless $kidpid;
    local $/          unless wantarray;
    return <$self>;    # 当退出范围的时候关闭
}

```

我们不能说这么做最好，一个捆绑的文件句柄可能更快一点。但是如果你比你的计算机更着急，那么

这个方法更容易写代码。

16.3.3 双向通讯

尽管在单向通讯中，用 `open` 与另外一条命令通过管道运转得很好，但是双向通讯该怎么办？下面这种方法实际上行不通：

```
open(PROG_TO_READ_AND_WRITE, "| some program |")    # 错！
```

而且如果你忘记打开警告，那么你就会完全错过诊断信息：

```
Can't do bidirectional pipe at myprog line 3.
```

`open` 函数不允许你这么干，因为这种做法非常容易导致死锁，除非你非常小心。但是如果你决定了，那么你可以使用标准的 `IPC::Open2` 库模块，用这个模块给一个子过程的 `STDIN` 和 `STDOUT` 附着两个管道。还有一个 `IPC::Open3` 模块用于三通 I/O（还允许你捕获子进程的 `STDERR`），但这个模块需要一个笨拙的 `select` 循环或者更方便一些的 `IO::Select` 模块。不过那样你就得放弃 Perl 的缓冲的输入操作（比如 `<>`，读一行）。

下面是一个使用 `open2` 的例子：

```
use IPC::Open2;
local (*Reader, *Writer);
$pid = open2(*Reader, *Writer, "bc -l");
$sum = 2;
for (1 .. 5) {
    print Writer "$sum * $sum\n";
    chomp($sum = <Reader>);
}
close Writer;
close Reader;
waitpid($pid, 0);
print "sum is $sum\n";
```

你还可以自动激活词法句柄：

```
my ($fhread, $fhwrite);
$pid = open2($fhread, $fhwrite, "cat -u -n");
```

这个方法的普遍的问题就是标准 I/O 缓冲实在是会破坏你的好事。即使你的输出文件句柄是自动刷新的（库为你做这些事情），这样另一端的进程将能及时地收到你的数据，但是通常你没法强迫它也返回这样的风格。在一些特殊的情况下我们是很幸运的，`bc` 可以在管道的模式下运行，而且还会刷新每个输出行。但是只有很少的几条命令是这样设计的，因此这个方法很少管用，除非你自己就是双向管道的另外一端的程序的作者。甚至是简单而且明确的 `ftp` 这样的交互式程序也会在这里失败，因为它们不会在管道上做行缓冲。它们只会在一个 `tty` 设备上这么干。

CPAN 上的 `IO::Pty` 和 `Expect` 模块可以在这方面做协助，因为它们提供真正的 `tty`（实际上是一个真正的伪 `tty`，不过看起来和真的一样）。它们让你可以获取其他进程的缓冲行而不用修改那些程序。

如果你把你的程序分裂成几个进程并且想让它们都能进行双向交流，那么你不能使用 Perl 的高端管道接口，因为那些接口都是单向通讯的。你需要使用两个低层的 `pipe` 函数调用，每个处理一个方向的交谈：

```
pipe(FROM_PARENT, TO_CHILD)    or die "pipe: $!";
pipe(FROM_CHILD, TO_PARENT)    or die "pipe: $!";
```

```

select((select(TO_CHILD), $| = 1))[0]);      # 自动刷新
select((select(TO_PARENT), $| = 1))[0]);    # 自动刷新

if ($pid = fork) {
    close FROM_PARENT; close TO_PARENT;
    print TO_CHILD "Parent Pid $$ is sending this\n";
    chomp($line = <FROM_CHILD>);
    print "Parent Pid $$ just read this: ` $line'\n";
    close FROM_CHILD; close TO_CHILD;
    waitpid($pid, 0);
} else {
    die "cannot fork: $!" unless defined $pid;
    close FROM_CHILD; close TO_CHILD;
    chomp($line = <FROM_PARENT>);
    print "Child Pid $$ just read this: ` $line'\n";
    print TO_PARENT "Child Pid $$ is sending this\n";
    close FROM_PARENT; close TO_PARENT;
    exit;
}

```

在许多 Unix 系统上，你实际上不必用两次独立的 **pipe** 调用来实现父子进程之间的全双工的通讯。**socketpair** 系统调用给在同一台机器上的相关进程提供了双向的联接。所以，除了用两个 **pipe** 以外，你还可以只用一个 **socketpair**。

```

use Socket;
socketpair(Child, Parent, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
or die "socketpair: $!";

# 或者让 perl 给你选择文件句柄

my ($kidfh, $dadfh);
socketpair($kidfh, $dadfh, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
or die "socketpair: $!";

```

在 **fork** 之后，父进程关闭 **Parent** 句柄，然后通过 **Child** 句柄读写。同时子进程关闭 **Child** 句柄，然后通过 **Parent** 句柄读写。

如果你正在寻找双向通讯的方法，而且是因为你准备交流的对方实现了标准的互联网服务，那么你通常应该先忽略这些中间人并且使用专为那些目的设计的 **CPAN** 模块。（参阅本章稍后“套接字”节获取它们的列表。）

16.3.4 命名管道

一个命名管道（通常称做 **FIFO**）是为同一台机器上不相关的进程之间建立交流的机制。“命名”管道的名字存在于文件系统中，实际上就是在文件系统的名字空间中放一个特殊的文件，而在文件背后不是磁盘，而是另外一个进程（注：你可以对 **Unix** 域套接字干一样的事情，不过你不能对它们用 **open**）。命名管道只不过是一个有趣的叫法而已。

如果你想把一个进程和一个不相干的进程联接起来，那么 **FIFO** 就很方便。如果你打开一个 **FIFO**，你的进程会阻塞住直到对端也有进程打开它为止。因此如果一个读进程先打开 **FIFO**，那么它会一直阻塞到写进程出现--反之亦然。

要创建一个命名管道，使用一个 **POSIX mkfifo** 函数--也就是说你用的必须是 **POSIX** 系统。在 **Microsoft** 系统上，你就要看看 **Win32::Pipe** 模块了，尽管看名字好象意思正相反，实际上它创建的是命名管道。（**Win32** 用户和我们其他人一样用 **pipe** 创建匿名管道。）

比如，假设你想把你的 `.signature` 文件在每次读取的时候都有不同的内容。那么只要把它作成一个命名管道，然后在另外一端应一个 Perl 程序守着，每次读取的时候都生成一个不同的数据就可以了。然后每当有任何程序（比如邮件程序，新闻阅读器，`finger` 程序等等）试图从那个文件中读取数据的时候，该程序都会与你的程序相联并且读取一个动态的签名。

在下面的例子里，我们使用很少见的 `-p` 文件测试器来判断某人（或某物）是否曾不小心删除了我们的 `FIFO`。（注：另外一个用途是看看一个文件句柄是否与一个命名的或者匿名的管道相联接，就象 `-p STDIN`。）如果 `FIFO` 被删除了，那么就没有理由做打开尝试，因此我们把这个看作退出请求。如果我们使用简单的用 `">$fpath"` 模式的 `open` 函数，那么就存在一个微小的冲突条件：如果在 `-p` 测试和打开文件之间文件消失了，那么它会不小心创建成为一个普通平面文件。我们也不能使用 `"<+$fpath"` 模式，因为打开一个 `FIFO` 用于读写是一个非阻塞式的打开（只对 `FIFO` 为真）。通过使用 `sysopen` 并且忽略 `O_CREAT` 标志，我们可以通过坚决不创建文件来避免这个问题。

```
use Fcntl;          # 我们要 sysopen
chdir;              # 回到家目录
$fpath = '.signature';
$ENV{PATH} .= ":/usr/games";

unless (-p $fpath) { # 不是一个管道
    if (-e _) {      # 而是其他东西
        die "$0: won't overwrite .signature\n";
    } else {
        require POSIX;
        POSIX::mkfifo($fpath, 0666) or die "can't mknod $fpath: $!";
        warn "$0: created $fpath as a named pipe\n";
    }
}

while (1) {
    # 如果签名文件被手工删除则退出
    die "Pipe file disappeared" unless -p $fpath;
    # 下一行阻塞住直到有读者出现
    sysopen(FIFO, $fpath, O_WRONLY)
    or die "can't write $fpath: $!";
    print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
    close FIFO;
    select(undef, undef, undef, 0.2); #睡眠 1/5 秒
}
```

关闭之后的短暂的睡眠是为了给读者一个机会读取已经写了的数据。如果我们只是马上循环出去然后再次打开 `FIFO`，而读者还没有完成刚刚发送的数据读取，那么就不会发送 `end-of-file`，因为写入进程已经成为历史了。我们既要一圈一圈循环，也要在每次叙述过程中让写入者比读者略微慢一些，这样才能让读者最后看到难得的 `end-of-file`。（而且我们还担心冲突条件？）

16.4. System V IPC

每个人都讨厌 `System V IPC`。它比打孔纸带还慢，使用与文件系统完全无关少得可怜的名字空间，使用人类讨厌的数字给它的对象命名，并且还常常自己忘记自己的对象，你的系统管理员经常需要用 `ipcs(1)` 查找那些丢失了的对象并且用 `ipcrm(1)` 删除它们，还得求老天保佑不要在用光内存以后才发现问题。

尽管有这些痛苦，古老的 `Sys IPC` 仍然有好几种有效的用途。三种 `IPC` 对象是共享内存，信号灯和消息。对于传送消息，目前套接字是更好的机制，而且移植性也更好。对于简单的信号灯用途，用文

件系统更好。对于共享内存--现在还有点问题。如果你的系统支持，用更现代的 `mmap(2)` 更好，（注：在 `CPAN` 上甚至有一个 `Mmap` 模块。）但是这些实现的质量因系统而异。而且它还需要更小心一些，以免让 `Perl` 从 `mmap(2)` 放你的字串的位置再次分配它们。但当程序员准备使用 `mmap(2)` 的时候，他们会听到那些常用这个的老鸟嘀咕，说自己是如何如何地绕开那些没有“统一缓冲缓存”（或者叫“统一总线捕蝇器”）的系统上的缓冲一致性问题，还有，他们讲他们知道的问题要比描述他们不懂的东西说得还要好，然后新手就赶快退回 `Sys V IPC` 并且憎恨所有他们必须使用的共享内存。

下面是一个小程序，它演示了一窝兄妹进程对一个共享内存缓冲的有控制的访问。`Sys V IPC` 对象也可以在同一台计算机的不相关的进程之间共享，不过那样的话你就得想出它们相互之间找到对方的方法。为了保证安全的访问，我们将为每块内存创建一个信号灯。（注：可能给每块内存创建一对信号灯更现实：一个用于读而另外一个用于写，而且实际上，这就是在 `CPAN` 上的 `IPC::Shareable` 模块所用的方法。但是我想在这里保持简单些。不过我们要承认，如果使用一对信号灯，你就可以利用好 `SysV2 IPC` 唯一的优点：你可以在整个信号灯集上执行原子操作，就好像它们是一个单元一样，这一点有时候很有用。）

每当你想获取或者写入新值到共享内存里面，你就必须先通过信号灯这一关。这个过程可能非常乏味，所以我们将把访问封装在一个对象类里面。`IPC::Shareable` 更进一步，把它的对象封装在一个 `tie` 接口里。

下面的程序会一直运行，直到你用一个 `Control-C` 或者相当的东西终止它：

```
#!/usr/bin/perl -w
use v5.6.0;    #或者更新
use strict;
use sigtrap wq(die INT TERM HUP QUIT);
my $PROGENY= shift(@ARGV) || 3;
eval { main() };    # 参阅下面的 DESTROY 找原因
die if $@ && $@ !~ /^Caught a SIG/;
print "\nDone.\n";
exit;

sub main{
    my $mem = ShMem->alloc("Original Creation at " . localtime);
    my (@kids, $child);
    $SIG{CHLD} = 'IGNORE';
    for (my $unborn = $PROGENY; $unborn > 0; $unborn--) {
        if ($child = fork) {
            print "$$ begat $child\n";
            next;
        }
        die "cannot fork: $!" unless defined $child;
        eval {
            while (1) {
                $mem->lock();
                $mem->poke("$$ " . localtime)
                unless $mem->peek =~ /^$$\b/o;
                $mem->unlock();
            }
        };
        die if $@ && $@ !~ /^CAught a SIG/;
        exit;    # 子进程退出
    }
    while (1) {
```



```

        print "Buffer is ", $mem->get, "\n";
        sleep 1;
    }
}

```

下面是 [ShMem](#)² 包，就是上面程序用的东西。你可以把它直接贴到程序的末尾，或者把它放到自己的文件里，（在结尾放一个"1;"）然后在主程序里 `require` 它。（用到的这两个 `IPC` 模块以后会在标准的 `Perl` 版本里找到。）

```

package ShMem;
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);
use IPC::Semaphore;
sub MAXBUF() { 2000 }

sub alloc {    # 构造方法
    my $class = shift;
    my $value = @_ ? shift : ' ';

    my $key = shmget(IPC_PRIVATE, MAXBUF, S_IRWXU) or die "shmget: $!";
    my $sem = IPC::Semaphore->new(IPC_PRIVATE, 1, S_IRWXU| IPC_CREAT)
    or die "IPC::Semaphore->new: $!";
    $sem->setval(0,1)    or die "sem setval: $!";

    my $self = bless {
        OWNER => $$,
        SHMKEY => $key,
        SEMA => $sem,
    } => $class;

    $self->put($value);
    return $self;
}

```

下面是抓取和存储方法。`get` 和 `put` 方法锁住缓冲区，但是 `peek` 和 `poke` 不会，因此后面两个只有在对象被手工锁住的时候才能用——当你想检索一个旧的数值并且存回一个修改过的数值，并且所有都处于同一把锁的时候你就必须手工上锁。演示程序在它的 `while (1)` 循环里做这些工作。整个事务必须在同一把锁里面发生，否则测试和设置就不可能是原子化的，并且可能爆炸。

```

sub get {
    my $self = shift;
    $self->lock;
    my $value = $self->peek(@_);
    $self->unlock;
    return $value;
}

sub peek {
    my $self = shift;
    shmread($self->{SHMKEY}, my $buff=' ', 0, MAXBUF) or die "shmread: $!"
    substr($buff, index($buff, "\0")) = ' ';
    return $buff;
}

sub put {
    my $self = shift;

```

```
$self->lock;
$self->poke(@_);
$self->unlock;
}

sub poke {
    my($self, $msg) = @_;
    shmwrite($self->{SHMKEY}, $msg, 0, MAXBUF) or die "shmwrite: $!";
}

sub lock {
    my $self = shift;
    $self->{SEMA}->op(0,-1,0) or die "semop: $!";
}

sub unlock {
    my $self = shift;
    $self->{SEMA}->op(0,1,0) or die "semop: $!";
}
```

最后，此类需要一个析构器，这样当对象消失的时候，我们可以手工释放那些存放在对象内部的共享内存和信号灯。否则，它们会活得比它们的创造者长，因而你不得不用 `ipcs` 和 `ipcrm`（或者一个系统管理员）来删除它们。这也是为什么我们在主程序里精心设计了把信号转换成例外的封装的原因：在那里才能运行析构器，**SysV IPC** 对象才能被释放，并且我们才不需要系统管理员。

```
sub DESTROY {
    my $self = shift;
    return unless $self->{OWNER} == $$;    #避免复制释放
    shmctl ($self->{SHMKEY}, IPC_RMID, 0)    or warn "shmctl RMID: $!";
    $self->{SEMA}->remove()                  or warn "sema->remove: $!";
}
```

16.5. 套接字

我们早先讨论的 **IPC** 机制都有一个非常严重的局限：它们是设计用来在运行在同一台计算机上的进程之间通讯用的。（即使有时候文件可以在机器之间通过象 **NFS** 这样的机制共享，但是在许多 **NFS** 实现中锁都会奇怪地失败，这样实际上就不可能对文件进行并发访问了。）对于通用目的的网络通讯，套接字是最好的办法。尽管套接字是在 **BSD** 里发明的，但它们很快就传播到其他类型的 **Unix** 里去了，并且现在你几乎可以在任何能用的操作系统里找到它。如果你的机器上没有套接字，那么你想使用互联网的话就会碰到无数的麻烦。

利用套接字，你既可以使用虚电路（象 **TCP** 流）也可以使用数据报（象 **UDP** 包）。你甚至可以做得更多，取决于你的系统。不过最常见的套接字编程用的是基于互联网际的套接字 **TCP**，因此我们在这里介绍这种类型的套接字。这样的套接字提供可靠的联接，运行起来有点象双向管道，但是并不局限于本地机器。互联网的两个杀手级的应用，**email** 和 **web** 浏览，都几乎是完全依赖于 **TCP** 套接字的。

你还在不知情的情况下很频繁地使用了 **UDP**。每次你的机器试图访问互联网上的一台主机，它都向你的 **DNS** 服务器发送一个 **UDP** 包请求其真实的 **IP** 地址。如果你想发送和接收数据报，那么你也可以自己使用 **UDP**。数据报比 **TCP** 更经济是因为它们不是面向联接的；也就是说，它们不太象打电话倒是象发信件。但是 **UDP** 同样也缺乏 **TCP** 提供的可靠性，这样它就更适合那些你不在乎是否有一两个包丢掉，多出来，或者坏掉，或者是你知道更高层的协议将强制某种程度的冗余（**DNS** 就是这样）的场合。

还有其他选择，但是非常少见。你可以使用 **Unix** 域套接字，但是只能用于本地通讯。有许多其他系统支持各种其他的并非基于 **IP** 的协议。毫无疑问一些地方的一些人会对它们感兴趣，但是我们将只会略微提到它们。

Perl 里面处理套接字的函数和 **C** 里面的对应系统调用同名，不过参数有些区别，原因有二：首先，**Perl** 的文件句柄和 **C** 的文件描述符的工作机制不同；第二，**Perl** 已经知道它的字串的长度，所以你不需传递这个信息。参阅第二十九章获取关于与套接字相关的系统调用的详细信息。

老的 **Perl** 的套接字代码有一个问题是人们会使用硬代码数值做常量传递到套接字函数里，那样就会破坏移植性。和大多数系统调用一样，与套接字相关的系统调用在失败的时候会礼貌而安静地返回 **undef**，而不是抛出一个例外。因此检查这些函数的返回值是很重要的，因为如果你传一些垃圾给它们，它们也不会大声叫嚷的。如果你曾经看到任何明确地设置 `$AF_INET = 2` 的代码，你就知道你要面对大麻烦了。一个更好的方法（好得没法比）是使用 **Socket** 模块或者是更友善一些的 **IO::Socket** 模块，它们俩都是标准模块。这些模块给你提供了设置服务器和客户端所需要的各种常量和协助函数。为了最大可能地成功，你的套接字程序应该象下面这样开头（不要忘记给服务器的程序里加 **-T** 开关打开错误检查）：

```
#!/usr/bin/perl -w
use strict;
use sigtrap;
use Socket;    # 或者 IO::Socket
```

正如我们在其他地方说明的一样，**Perl** 依靠你的 **C** 库实现它的大部分系统特性，而且不是所有系统都支持所有的套接字类型。所以最安全的可能办法就是只做普通的 **TCP** 和 **UDP** 套接字操作。比如，如果你希望你的代码有机会能够移植到那些你还没有想过的系统上，千万别假设那台系统支持可靠的顺序报文协议。也别想在不相关的进程之间通过本地 **Unix** 域套接字传递打开的文件描述符。（的确，你可以在许多 **Unix** 机器上做那些事情--参阅你本机的 **recvmsg(2)** 手册页。）

如果你只是想使用一个标准的互联网服务，比如邮件，新闻组，域名服务，**FTP**，**Telnet**，**Web** 等等，那么你不用从零开始，先试者使用现有的 **CPAN** 模块。为这些目的设计的打了包的模块包括 **Net::SMTP**（或者 **Mail::Mailer**），**Net::NNTP**，**Net::DNS**，**Net::FTP**，**Net::Telnet**，和各种各样的 **HTTP** 相关的模块。有关 **CPAN** 上的模块，你可能要查看一下第五节的网络和 **IPC**，第十五节的 **WWW** 相关模块，以及第十六节的服务器和守护进程应用。

在随后的一节里，我们将讲述几个简单的客户和服务器的例子，但是我们对所使用的函数做过多的解释，因为那样会和我们将在第二十九章里的描述重合很大部分。

16.5.1 网络客户端程序

如果你需要在可能是不同机器之间的可靠的客户端-服务器通讯，那么请用互联网域套接字。

要在什么地方创建一个 **TCP** 与服务器联接的 **TCP** 客户端，最简单的方法通常是使用标准的 **IO::Socket::INET** 模块：

```
use IO::Socket::INET;

$socket = IO::Socket::INET->new(PeerAddr => $remote_host,
PeerPort => $remote_port,
Proto => "tcp",
Type => SOCK_STREAM)
or die "Couldn't connect to $remote_host:$remote_port: $!\n";

# 通过套接字发送某些东西,
print $socket "Why don't you call me anymore?\n";
```

```
# 读取远端响应
$answer = <$socket>;

# 然后在结束之后终止联接。
close($socket);
```

如果你只有主机名和端口号准备联接，并且愿意在其他域里使用缺省的字段，那么调用这个函数的缩写形式也是很好的：

```
$socket = IO::Socket::INET->new("www.yahoo.com:80")
or die "Couldn't connect to port 80 of yahoo: $!";
```

如果使用基本的 **Socket** 模块联接：

```
use Socket;

# 创建一个套接字
socket(Server, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

# 制作远端机器的地址
$internet_addr = inet_aton($remote_host)
or die "Couldn't convert $remote_host into an Internet address: $!\n";
$paddr = sockaddr_in($remote_port, $internet_addr);

# 联接
connect(Server, $paddr)
or die "Couldn't connect to $remote_host:$remote_port: $!\n";

select((select(Server), $| = 1)[0]); # 打开命令缓冲

# 通过套接字发送一些东西
print Server "Why don't you call me anymore?\n";

# 读取远端的响应
$answer = <Server>;

# 处理完之后终止联接
close(Server);
```

如果你想只关闭你方的联接，这样远端就可以得到一个 **end-of-file**，不过你还是能够读取来自该服务器的数据，使用 **shutdown** 系统调用进行半关闭：

```
# 不再向服务器写数据
shutdown(Server, 1); # 在 v5.6 里的 Socket::SHUT_WR 常量
```

16.5.2 网络服务器

下面是和前面的客户端对应的服务器。如果用标准的 **IO::Socket::INET** 类来实现是相当简单的活：

```
use IO::Socket::INET;

$server = IO::Socket::INET->new(LocalPort => $server_port,
Type      => SOCK_STREAM,
Reuse     => 1,
```

```

Listen    => 10)    # 或者 SOMAXCONN
or die "Couldn't be a tcp server on port $server_port: $!\n";

while ($client = $server->accept()) {
    # $client 是新联接
}

close($server);

```

你还可以用低层 **Socket** 模块来写这些东西:

```

use Socket;

# 创建套接字
socket( Server, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

# 为了我们可以快速重起服务器
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, 1);

# 制作自己的套接字地址
$my_addr = sockaddr_in($server_port, INADDR_ANY);
bind(Server, $my_addr)
or die "Couldn't bind to port $server_port: $!\n";

# 为来访联接建立一个队列
listen(Server, SOMAXCONN)
or die "Couldn't listen on port $server_port: $!\n";

# 接受并处理任何联接
while(accept(Client, Server)) {
    # 在新建的客户端联接上做些处理
}

close(Server);

```

客户端不需要和任何地址 **bind**（绑定），但是服务器需要这么做。我们把它的地址声明为 **INADDR_ANY**，意味着客户端可以从任意可用的网络接口上来，如果你想固定使用某个接口（比如一个网关或者防火墙机器），那么使用该机器的真实地址。（客户端也可以这么干，不过很少必须这么干。）

如果你想知道是哪个机器和你相联，你可以在客户联接上调用 **getpeername**。这样返回一个 **IP** 地址，你可以自己把它转换成一个名字（如果你能）：

```

use Socket;
$other_end = getpeername(Client)
or die "Couldn't identify other end: $!\n";
($port, $iaddr) = unpack_sockaddr_in($other_end);
$actual_ip = inet_ntoa($iaddr);
$claimed_hostname = gethostbyaddr($iaddr, AF_INET);

```

这个方法一般都可以欺骗过去，因为该 **IP** 地址的所有者可以把它们的反向域名表设置为任何它们想要的东西。为了略微增强信心，从另外一个方向再转换一次：

```

@name_lookup = gethostbyname($claimed_hostname)
or die "Could not reverse $claimed_hostname: $!\n";
@resolved_ips = map { inet_ntoa($_) } @name_lookup[ 4 .. $#name_lookup ];

```



```
$might_spoof = !grep { $actual_ip eq $_ } @resolved_ips;
```

一旦一个客户端与你的服务器相联，你的服务器就可以对那个客户端句柄进行 I/O。不过因为你的服务器忙着做这件事，所以它不能在给来自其他客户端的请求提供服务了。为了避免每次只能给一个客户端服务，许多服务器都是马上 **fork** 一个自己的克隆来处理每次到来的联接。（其他的服务器预先 **fork**，或者使用 **select** 系统调用在多个客户端之间复用 I/O。）

```
REQUEST:
```

```
while (accept(Client, Server)) {
    if ($kidpid = fork) {
        close Client;          # 父进程关闭不用的句柄
        next REQUEST;
    }
    defined($kidpid) or die "cannot fork: $!";

    close Server;              # 子进程关闭无用的句柄

    select(Client);            # 打印用的新的缺省句柄
    $| = 1;                     # 自动刷新

    # 预联接的子进程代码与客户端句柄进行 I/O
    $input = <Client>;
    print Client "output\n";    # 或者 STDOUT，一样的东西

    open(STDIN, "<<&Client") or die "can't dup client: $!";
    open(STDOUT, ">&Client") or die "can't dup client: $!";
    open(STDERR, ">&Client") or die "can't dup client: $!";

    # 运行计算器，就当个例子
    system("bc -l");            # 或者任何你喜欢的东西，只要它不会逃逸出 shell

    print "done\n";            # 仍然是给客户端

    close Client;
    exit;                       # 不让子进程回到 accept!
}
```

这个服务器为每个到来的请求用 **fork** 克隆一个子进程。那样它就可以一次处理许多请求——只要你创建更多的进程。（你可能想限制这些数目。）即使你不 **fork**，**listen** 也会允许最多有 **SOMAXCONN**（通常是五个或更多个）挂起的联接。每个联接使用一些资源，不过不象一个进程用的那么多。分裂出的服务器进程必须仔细清理它们运行结束了的子进程（在 Unix 语言里叫“**zombies**”僵死进程），否则它们很快就会填满你的进程表。在“信号”一节里的讨论的 **REAPER** 代码会为你做这些事情，或者你可以赋值 **\$SIG{CHLD} = 'IGNORE'**。

在运行其他命令之前，我们把标准输入和输出（以及错误）联接到客户端联接中。这样任何从 **STDIN** 读取输入并且写出到 **STDOUT** 的命令都可以与远端的机器交谈——如果没有重新赋值，这条命令就无法找到客户端句柄——因为客户端句柄缺省的时候在 **exec** 范围外自动关闭。

如果你写一个网络服务器，我们强烈建议你使用 **-T** 开关以打开污染检查，即使它们没有运行 **setuid** 或者 **setgid** 也这样干。这个主意对那些服务器和任何代表其他进程运行的程序（比如所有 **CGI** 脚本）都是好主意，因为它减少了来自外部的人威胁你的系统的机会。参阅第二十三章里“操作不安全的数据”节获取更多相关信息。

当写互联网程序的时候有一个额外的考虑：许多协议声明行结束符应该是 **CRLF**，它可以用好多种方

法来声明: "\015\12", 或者 "\xd\xa", 或者还可以是 chr(13).chr(10)。对于版本 5.6 的 Perl, 说 v13.10 也生成一样的字串。(在许多机器上, 你还可以用 "\r\n" 表示 CRLF, 不过如果你想向 Mac 上移植, 不要用 "\r\n", 因为在 Mac 上 "\r\n" 的含义正相反!) 许多互联网程序实际上会把只有 "\012" 的字串当行结束符, 但那是因为互联网程序总是对它们接收的东西很宽容而对它们发出的东西很严格。(现在只有我们能让人们这么做了...)

16.5.3 消息传递

我们早先提到过, UDP 通讯更少过热但是可靠性也更低, 因为它不保证消息会按照恰当的顺序到达——或者甚至不能保证消息能够到达。人们常把 UDP 称做不可靠的数据报协议 (Unreliable Datagram Protocol)。

不过, UDP 提供了一些 TCP 所没有的优点, 包括同时向一堆目的主机广播或者多点广播的能力 (通常在你的本地子网)。如果你觉得自己太关心可靠性并且开始给你的消息系统做检查子系统, 那么你可能应该使用 TCP。的确, 建立或者解除一个 TCP 联接开销更大, 但是如果你可以用许多消息补偿 (或者一条长消息) 这些, 那么也是值得的。

不管怎么说, 下面是一个 UDP 程序的例子。它与在命令行声明的机器的 UDP 时间端口联接, 如果命令行上没有声明机器名, 它与它能找到的任何使用统一广播地址的机器联接。(注: 如果还不行, 那么运行 `ifconfig -a` 找出合适的本地广播地址。) 不是所有机器都打开了时间服务, 尤其是跨防火墙边界的时候, 但是那些给你发包的机器会给你按照网络字节序打包发送一个 4 字节的整数, 表明它认为的时间值。返回的这个时间值是自 1900 年以来的秒数。你必须减去 1970 年 和 1900 年之间的描述然后才能传给 `localtime` 或者 `gmtime` 转换函数。

```
#!/usr/bin/perl
# clockdrift - 与其他系统的时钟进行比较
#             如果没有参数, 广播给其他人听
#             等待1.5秒听取回答

use v5.6.0;    # 或者更高版本
use warnings;
use strict;
use Socket;

unshift(@ARGV, inet_ntoa(INADDR_BROADCAST))
unless @ARGV;

socket(my $msgsock, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
or die "socket: $!";

# 有些有毛病的机器需要这个。不会伤害任何其他机器。
setsockopt($msgsock, SOL_SOCKET, SO_BROADCAST, 1)
or die "setsockopt: $!";

my $portno = getservbyname("time", "udp")
or die "no udp time port";

for my $target (@ARGV) {
    print "Sending to $target: $portno\n";
    my $destpaddr = sockaddr_in($portno, inet_aton($target));
    send($msgsock, "x", 0, $destpaddr)
    or die "send: $!";
}
```

```
# 时间服务返回自 1900 年以来以秒计的 32 位时间
my $FROM_1900_TO_EPOCH = 2_208_988_800;
my $time_fmt = "N";          # 并且它以这种二进制格式处理。
my $time_len = length(pack($time_fmt, 1));  # 任何数字都可以

my $inmask = ' ';  # 存储用于 select 文件号位的字串，
vec($inmask, fileno($msgsock), 1) = 1;

# 等待半秒钟等待输入出现
while (select (my $outmask = $inmask, undef, undef, 0.5)) {
    defined(my $srcpaddr = recv($msgsock, my $bintime, $time_len, 0))
    or die "recv: $!";
    my($port, $ipaddr) = sockaddr_in($srcpaddr);
    my $sendhost = sprintf "%s [%s]",
        gethostbyaddr($ipaddr, AF_INET) || 'UNKNOWN',
        inet_ntoa($ipaddr);
    my $delta = unpack($time_fmt, $bintime) -
        $FROM_1900_TO_EPOCH - time();
    print "Clock on $sendhost is $delta seconds ahead of this one.\n";
}
```

Revision: r1.2 - 08 Sep 2005 - 04:00 - [TingYu](#)

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

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](#)