

第十四章 捆绑(tie)变量(下)

- ↓ 第十四章 捆绑(tie)变量(下)
 - ↓ 14.3 捆绑散列
 - ↓ 14.3.1 散列捆绑方法
 - ↓ 14.4 捆绑文件句柄
 - ↓ 14.4.1 文件句柄捆绑方法
 - ↓ 14.4.2 创建文件句柄
 - ↓ 14.5 一个精细的捆绑陷阱
 - ↓ 14.6 CPAN 里的模块

14.3 捆绑散列

一个实现捆绑散列的类应该定义八个方法。**TIEHASH** 构造一个新对象。**FETCH** 和 **STORE** 访问键字/数值对。**EXISTS** 报告某键字是否存在于散列中，而 **DELETE** 删除一个键字和它关联的数值（注：请注意在 Perl 里，散列里不存在一个键字与存在一个键字但是其对应数值为 **undef** 是不同的两种情况。这两种情况可以分别用 **exists** 和 **defined** 测试。）**CLEAR** 通过删除所有键字/数值对清空散列。**FIRSTKEY** 和 **NEXTKEY** 在你调用 **keys**，**values**，或 **each** 的时候逐一遍历键字/数值对。还有就是和往常一样，如果你想在对象删除的时候执行某种特定的动作，那么你可能还要定义 **DESTROY** 方法。（如果你觉得方法太多，那么你一定没有认真阅读上一节关于数组的内容。不管怎样，你都可以自由地从标准的 **Tie::Hash** 模块继承缺省的方法，只用重新定义你感兴趣的方法。同样，**Tie::StdHash** 假设此实现同样也是一个散列。）

比如，假定你想创建这么一个散列：每次你给一个键字赋值的时候，它不是覆盖掉原来的内容，而是把新数值附加到一个数值数组上。这样当你说：

```
$h{$k} = "one";
$h{$k} = "two";
```

它实际上干的是：

```
push @{$h{$k}}, "one";
push @{$h{$k}}, "two";
```

这个玩叶儿不算什么复杂的主意，所以你应该能用一个挺简单的模块实现。把 **Tie::StdHash** 用做基类，下面就是干这事的 **Tie::AppendHash**：

```
package Tie::AppendHash;
use Tie::Hash;
our @ISA = ("Tie::StdHash");
sub STORE {
    my ($self, $key, $value) = @_;
    push @{$self->{$key}}, $value;
}
1;
```

14.3.1 散列捆绑方法

这儿是一个很有趣的捆绑散列类的例子：它给你一个散列，这个散列代表用户特定的点文件（也就是说，文件名开头是一个句点的文件，这样的文件是 **Unix** 初始化文件的命名传统。）你用文件名做索引（除去开头的句点）把文件名放进散列，而拿出来的是点文件的内容。比如：

```
use DotFiles;
tie %dot, "DotFiles";
```

```
if ( $dot{profile} =~ /MANPATH/ or
$dot{login}    =~ /MANPATH/ or
$dot{cshrc}    =~ /MANPATH/ ) {
    print "you seem to set your MANPATH\n";
}
```

下面是使用我们捆绑类的另外一个方法：

第三个参数是用户名，我们准备把他的点文件捆绑上去。

```
tie %him, "DotFiles", "daemon";
foreach $f (keys %him) {
    printf "daemon dot file %s is size %d\n", $f, length $him{$f};
}
```

在我们的 [DotFiles²](#) 例子里，我们把这个对象当作一个包含几个重要数据域的普通散列来实现，这几个数据域里只有 {CONTENTS} 域会保存一般用户当作散列的东西。下面是此对象的实际数据域：

数据域	内容
USER	这个对象代表谁的点文件
HOME	那些点文件在哪里
CLOBBER	我们是否允许修改或者删除这些点文件
CONTENTS	点文件名字和内容映射的散列

下面是 [DotFiles²](#).pm 的开头：

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . "()" }
my $DEBUG = 0;
sub debug { $DEBUGA = @_ ? shift : 1 }
```

对于我们的例子而言，我们希望打开调试输出以便于在开发中的跟踪，因此我们为此设置了 \$DEBUG。我们还写了一个便利函数放在内部以便于打印警告：whowasi 返回调用了当前函数的那个函数的名字（whowasi 是“祖父辈”的函数）。

下面就是 [DotFiles²](#) 捆绑散列的方法：

CLASSNAME->TIEHASH(LIST)

这里是 [DotFiles²](#) 构造器：

```
sub TIEHASH {
    my $self = shift;
    my $user = shift || $>;
    my $dotdir = shift || "";

    croak "usage: @{$[ &whowasi ] } [ USER [DOTDIR]]" if @_;

    $user = getpwuid($user) if $user =~ /^\\d+$/;
    my $dir = (getpwnam($user))[7]
    or croak "@{[ &whowasi ] }: no user $user";
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER => $user,
        HOME => $dir,
```

```

        CONTENTS => {},
        CLOBBER => 0,
    };

    opendir DIR, $dir
    or croak "@{&whowasi}: can't opendir $dir: $!";
    for my $dot ( grep /^\.\/ && -f "$dir/$_", readdir(DIR) ) {
        $dot =~ s/^\.\/;
        $node->{CONTENTS}{$dot} = undef;
    }
    closedir DIR;

    return bless $node, $self;
}

```

值得一提的是，如果你准备用文件来测试上面的 `readdir` 的返回值，你最好预先 准备好有问题的目录（象我们一样）。否则，因为我们没有用 `chdir`，所以你很有可能会测试的是错误的文件。

SELF->FETCH(KEY)

这个方法实现的是从这个捆绑的散列里读取元素。它在对象后面还有一个参数： 你想抓取的散列元素的键字。这个键字是一个字串，因而你可以对它做你想做的 任何处理（和字串一致）。下面是我们 [DotFiles²](#) 例子的抓取：

```

sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";

    unless (exists $self->{CONTENTS}->{$dot} || -f $file ) {
        carp "@{&whowasi}: no $dot file" if $DEBUG;
        return undef;
    }

    # 实现一个缓冲
    if (defined $self->{CONTENTS}->{$dot} ) {
        return $self->{CONTENTS}->{$dot};
    } else {
        return $self->{CONTENTS}->{$dot} = `cat $dir/.$dot`;
    }
}

```

我们在这里做了一些手脚：我们用的是 Unix 的 `cat (1)` 命令，不过这样打开 文件的移植性更好（而且更高效）。而且，因为点文件是一个 Unix 式的概念， 所以我们不用太担心。或者是不应该太担心。或者...

SELF->STORE(KEY, VALUE)

当捆绑散列里的元素被设置（或者写入）的时候，这个方法做那些脏活累活。 它在对象后面还有两个参数：我们存贮新值的键字，以及新值本身。

就我们的 [DotFiles²](#) 例子而言，我们首先要在 `tie` 返回的最初的对象上调用方法 `clobber` 以后才能允许拥护覆盖一个文件：

```

sub STORE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $value = shift;
    my $file = $self->{HOME} . ".$dot";

    croak "@{[&whowasi]}: $file not clobberable"
    unless $self->{CLOBBER};

    open(F, "> $file") or croak "cna't open $file: $!";
    print F $vare;
    close(F);
}

```

如果有谁想删除什么东西，他们可以说：

```

$ob = tie %daemon_dots, "daemon";
$ob->clobber(1);
$daemon_dot{signature} = "A true daemon\n";

```

不过，它们可以用 `tied` 设置 `{CLOBBER}`：

```

tie %daemon_dots, "Dotfiles", "daemon";
tied(%daemon_dots)->clobber(1);

```

或者用一条语句：

```

(tie %daemon_dots, "DotFiles", "daemon")->clobber(1);

```

`clobber` 方法只是简单的几行：

```

sub clobber{
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}

```

SELF->DELETE(KEY)

这个方法处理从散列中删除一个元素的请求。如果你模拟的散列在某些地方用了 `真的` 散列，那么你可以只调用真的 `delete`。同样，我们将仔细检查用户是否 `真的` 想删除文件：

```

sub DELETE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $file = $self->{HOME} . ".$dot";
    croak "@{[&whowasi]}: won't remove file $file"
    unless $self->{CLOBBER};
    delete $self->{CONTENTS}->{$dot};
    unlink $file or carp "@{[&whowasi]}: can't unlink $file: $!";
}

```

SELF->CLEAR

当需要清理整个散列的时候运行这个方法，通常是给散列赋一个空列表。在我们的 例子里，这可以要删除用户的所有点文件的！这可真是一个危险的方法，所以我们 要求在清理之前要把 CLOBBER 设置为大于 1：

```
sub CLEAR {
    carp &whowasi if $DEBUG;
    my $self = shift;
    croak "@{[&whowasi]}: won't remove all dotfiles for $self->{USER}"
    unless $self->{CLOBBER} > 1;
    for my $dot (key % {$self->{CONTENTS}}) {
        $self->DELETE($dot);
    }
}
```

SELF->EXISTS(KEY)

当用户在某个散列上调用 **exists** 函数的时候运行这个方法。在我们的例子里， 我们会查找 {CONTENTS} 散列元素来找出结果：

```
sub EXISTS {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    return exists $self->{CONTENTS}->{$dot};
}
```

SELF->FIRSTKEY

当用户开始遍历散列，比如说用一个 **keys**，或者 **values**，或者一个 **each** 调用的 时候需要这个方法。我们通过在标量环境中调用 **keys**，重置其（**keys** 的）内部 状态以确保后面 **retrun** 语句里的 **each** 将拿到第一个键字。

```
sub FIRSTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $temp = keys %{$self->{CONTENTS}};
    return scalar each %{$self->{CONTENTS}};
}
```

SELF->NEXTKEY(PREVKEY)

这个方法是 **keys**，**values** 或者 **each** 函数的叙述器。**PREVKEY** 是上次访问的 键字，Perl 知道该提供什么。如果 **NEXTKEY** 方法需要知道它的前面的状态来 计算下一个状态时这个变量很有用。

就我们的例子，我们正在使用一个真正的散列来代表捆绑了的散列的数据，不同的 只是这个散列保存在散列的 **CONTENTS** 数据域而不是在散列本身。因此我们只需要 依赖 Perl 的 **each** 叙述器就行了：

```
sub NEXTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return scalar each %{ $self->{CONTENTS}}
}
```

SELF->DESTORY

当你准备删除这个捆绑的散列对象的时候触发这个方法。你实际上并不需要这个东西，除非是用做调试和额外的清理。下面是一个非常简单的版本：

```
sub DESTROY{
    carp &whowasi if $DEBUG;
}
```

请注意我们已经给出了所有的方法，你的作业就是退回去把我们代换 `@{&whowasi}` 的地方找出来然后把他们替换成名字为 `$whowasi` 的简单捆绑标量，并且要实现相同的功能。

14.4捆绑文件句柄

一个实现捆绑文件句柄的类应该定义下面的方法：**TIEHANDLE** 和至少 **PRINT**，**PRINTF**，**WRITE**，**READLINE**，**GETC** 和 **READ** 之一。该类还可以提供一个 **DESTROY** 方法，以及 **BINMODE**，**OPEN**，**CLOSE**，**EOF**，**FILENO**，**SEEK**，**TELL**，**READ** 和 **WRITE** 方法以便于相应的 Perl 内建函数用这个捆绑的文件句柄。（当然，这也不是绝对正确：**WRITE** 对应 **syswrite** 而与 Perl 内建的 **write** 函数没有任何关系，**write** 是和 **format** 声明一起用于打印的。）

当 Perl 内嵌于其他程序中（比如 **Apache** 和 **vi**）时以及当向 **STDOUT** 和 **STDERR** 的输出需要以某种特殊的方式重新定向的时候捆绑的文件句柄就特别有用了。

不过捆绑的文件句柄实际上完全不必与文件捆绑。你可以用输出语句制作一个存在于内存的数据结构以及用输入语句把它们读取回来。下面是一个反转 **print** 和 **printf** 语句的打印顺序但却不用颠倒相关行顺序的简单方法：

```
package ReversePrint;
use strict;
sub TIEHANDLE {
    my $class = shift;
    bless [], $class;
}
sub PRINT {
    my $self = shift;
    push @$self, join ' ', @_
}
sub PRINTF {
    my $self = shift;
    my $fmt = shift;
    push @$self, sprintf $fmt, @_;
}
sub READLINE {
    my $self = shift;
    pop @$self;
}

package main;
my $m = "--MORE--\n";
tie *REV, "ReversePrint";

# 做一些 print 和 printf.
print REV "The fox is now dead. $m";
printf REV <<"END", int rand 10000000;
The quick brown fox jumps over
over the lazy dog %d times!
```

```
END
```

```
print REV <<"END";
The quick brown fox jumps
over the lazy dog.
END
```

```
# 现在从同一个句柄中读回来
print while <REV>;
```

打印出:

```
The quick brown fox jumps
over the lazy dog.
The quick brown fox jumps over
over the lazy dog 3179357 times!
The fox is now dead.--MORE--
```

14.4.1 文件句柄捆绑方法

对于我们扩展的例子而言，我们将创建一个文件句柄，并且向之打印大写字串。为了明确，我们会在把这个文件句柄打开的时候向它打印，而当结束关闭的时候打印。这个方法是我们从格式优良的 XML 中借来的。

下面是我们将实现这个类的 `shout.pm` 文件的开头:

```
package Shout;
use Carp;          # 这样我们就可以把我们的错误汇报出来
```

然后我们把在 `shout.pm` 里定义的方法列出来:

CLASSNAME->TIEHANDLE(LIST)

这是该类的构造器，和往常一样，应该返回一个赐福了的引用。

```
sub TIEHANDLE {
    my $class = shift;
    my $form = shift;
    open my $self, $form, @_ or croak "can't open $form@_: $!";
    if ($form =~ />/) {
        print $self "<SHOUT>\n";
        $$self->{WRITING} = 1;    # 记得写结束标记
    }
    return bless $self, $class;    # $self 是一个全局引用
}
```

在这里，我们根据传递给 `tie` 操作符的模式和文件名打开一个新的文件句柄，向文件中写入，然后返回一个指向它的赐福了的引用。在 `open` 语句里有一大堆东西，不过我们将只会指出一点，除了通常的 "open or die" 惯用法以外，`my $self` 给 `open` 提供了一个未定义的标量，`open` 知道自动把那个标量转成一个类型团。这个变量是类型团这一点非常重要，因为这个类型团不仅包含文件真实的 I/O 对象，而且还包含各种各样其他可以自由获取的数据结构，比如一个标量

(`$$self`)，一个数组(`@$self`)，和一个散列(`%%$self`)。(我们不会提到子过程，`&$self`。)

`$form` 是文件名或者模式参数。如果它是一个文件名，`@_` 就是空的，所以它的性质就象一个两个参数的 `open`。否则，`$form` 就是剩余参数的模式。

`open` 之后，我们检测一下看看我们是否应该写入表示开始的标记。如果是，我们就写。然后我们马上使用那些我们谈到的团数据结构。那个 `@@self->{WRITING}` 是一个使用团存储有趣信息的一个例子。在这个例子里，我们记住是否写过起始 标记，这样我们才知道我们是否应该做相应的结束标记。我们正在使用 `$$$self` 散列，所以我们可以给那个数据域一个象样的名字。我们本可以用象 `$$$self` 这样的标量，但是那样不能自说明。（或者它只能自说明——取决于你如何看它。）

SELF->PRINT(LIST)

这个方法实现了一个向捆绑的句柄 `print`。LIST 是传递给 `print` 的东西。我们 下面的方法把 LIST 的每个元素都转换成大写：

```
sub PRINT {
    my $self = shift;
    print $self map {uc} @_;
}
```

SELF->READLINE

当用尖角操作符 (`<`) 或者 `readline` 读句柄的时候，用这个方法提供数据。 当没有更多数据可读的时候，这个方法应该返回 `undef`。

```
sub READLINE {
    my $self = shift;
    return <$self>;
}
```

在这里，我们只是简单地 `return <$self>`，这样，根据标量环境还是列表环境， 这个方法就能做出正确的反映。

SELF->GETC

当在捆绑的文件句柄上使用 `getc` 的时候就会运行这个方法。

```
sub GETC {
    my $self = shift;
    return getc($self);
}
```

和我们 `Shout` 类的几个方法类似，`GETC` 只是简单地调用相应的 Perl 内建的函数 然后返回结果。

SELF->OPEN(LIST)

我们的 `TIEHANDLE` 方法本身就打开一个文件，但是一个使用 `Shout` 类的程序在 那之后调用 `open` 将触发这个方法。

```
sub OPEN {
    my $self = shift;
    my $form = shift;
    my $name = "$form@_";
    $self->CLOSE;
    open($self, $form, @_) or croak "can't reopen $name: $!";
    if ($form =~ />/) {
        print $self "<SHOU>\n" or croak "can't start print: $!";
        $$$self->{WRITING} = 1;      # 记得写结束标记
    }
    else {
```



```

        $$self->{WRITING} = 0;          # 记得不要写结束标记
    }
    return 1;
}

```

我们激活了我们自己的 **CLOSE** 方法明确地关闭文件，以免用户不愿意自己做。然后我们打开一个新文件，文件名是在 **open** 里声明的，然后再向里面写东西。

SELF->CLOSE

这个方法处理关闭句柄的请求。在这里，我们搜索到文件的结尾，如果成功，则打印，然后调用 Perl 内建的 **close**。

```

sub CLOSE {
    my $self = shift;
    if ($$self->{WRITING}) {
        $self->SEEK(0,2) or return;
        $self->PRINT("</SHOUT>\n") or return;
    }
    return close $self;
}

```

SELF->SEEK(LIST)

当你对一个捆绑的文件句柄进行 **seek** 的时候调用 **SEEK** 方法。

```

sub SEEK {
    my $self = shift;
    my ($offset, $whence) = @_;
    return seek($self, $offset, $whence);
}

```

SELF->TELL

当你对一个捆绑的文件句柄调用 **tell** 的时候调用这个方法。

```

sub TELL {
    my $self = shift;
    return tell $self;
}

```

SELF->PRINTF(LIST)

当在捆绑的句柄上面使用 **printf** 的时候运行这个方法。**LIST** 将包含格式和需要打印的条目。

```

sub PRINTF {
    my $self = shift;
    my $template = shift;
    return $self->PRINT(sprintf $template, @_);
}

```

在这里，我们用 **sprintf** 生成格式化字符串然后把它传递给 **PRINT** 转成大写。不过这里也没有让你一定要用内建的 **sprintf** 函数的原因。你可以截获百分号 逃逸以满足你自己的目的。

SELF->READ(LIST)

当用 **read** 或者 **sysread** 对句柄做读操作时，该方法会做出响应。请注意 我们“现场”修改 **LIST** 的第一个参数，模拟 **read** 的能力：它填充作为它的 第二个参数传递进来的标量。

```
sub READ {
    my ($self, undef, $length, $offset) = @_;
    my $bufref = \$_[1];
    return read($self, $$bufref, $length, $offset);
}
```

SELF->WRITE(LIST)

当用 `syswrite` 对该句柄写入的时候调用这个方法。在这里，我们把待写字串变成 大写。

```
sub WRITE {
    my $self = shift;
    my $string = uc(shift);
    my $length = shift || length $string;
    my $offset = shift || 0;
    return syswrite $self, $string, $length, $offset;
}
```

SELF->EOF

如果用 `eof` 对一个与 `Shout` 类捆绑的文件句柄进行测试的时候，这个方法返回 一个布尔值。

```
sub EOF {
    my $self = shift;
    return eof $self;
}
```

SELF->BINMODE(DISC)

这个方法声明将要用于这个文件句柄的 `I/O` 规则。如果没有声明，它把这个捆绑 了的文件句柄置于 二进制模式 (`:raw` 规则)，用于那些区分文本和二进制的文件 文件系统。

```
sub BINMODE {
    my $self = shift;
    my $disc = shift || ":raw";
    return binmode $self, $disc;
}
```

就这么写，但实际上这个方法在我们的类中没有什么用，因为 `open` 已经向句柄 中写入数据了。所以 在我们的例子里我们可能可以做的更简单些：

```
sub BINMODE { croak("Too late to use binmode") }
```

SELF->FILENO

这个方法应该返回与捆绑的文件句柄相关联的操作系统文件描述符 (`fileno`) 。

```
sub FILENO {
    my $self = shift;
    return fileno $self;
}
```

SELF->DESTROY

和其他类型的捆绑一样，当对象将要删除的时候触发这个方法。对象清理自己的 时候很有用。在这里，我们确保文件关闭了，以免程序忘记调用 `close`。我们可以 只说 `close $self`，不过更好的方法是调用该类的 `CLOSE` 方法。这样的话，如果 类的设计者决定修改文件关闭的方法的话，这个

DESTROY 方法就不用改了。

```
sub DESTROY {
    my $self = shift;
    $self->CLOSE;      # 用 Shout 的 CLOSE 方法关闭文件
}
```

下面是我们的 Shout 类的一个演示：

```
#!/usr/bin/perl
use Shout;
tie(*FOO, Shout::, ">filename");
print FOO "hello\n";      # 打印 HELLO。
seek FOO, 0, 0;           # 退回到开头
@lines = <FOO>;           # 调用 READLINE 方法。
close FOO;                # 明确关闭文件。
open(FOO, "+<", "filename"); # 重新打开 FOO, 调用 OPEN。
seek(FOO, 8, 0);          # 忽略 "<SHOUT>\n"。
sysread(FOO, $inbuf, 5)   # 从 FOO 读取 5 个字节到 $inbuf。
print "found $inbuf\n";   # 应该打印 "hello"。
seek(FOO, -5, 1);         # 退回 "hello"之前。
syswrite(FOO, "ciao!\n", 6); # 写 6 个字节到 FOO。
untie(*FOO);              # 明确调用 CLOSE 方法
```

在运行完这些以后，这个文件包含：

```
CIAO!
```

下面是对付那个内部团的一些更怪异而又神奇的东西。我们和往常一样使用相同的散列，但是有新的键字 **PATHNAME** 和 **DEBUG**。首先我们安装一个字串化的重载，这样，如果打印我们的一个对象的时候就会打印出路径名（参阅第十三章，重载）：

```
# 这就是所有酷玩叶儿
use overload q(" ") => sub { $_[0]->pathname };

# 这里是放你想跟踪的函数的存根。
sub trace {
    my $self = shift;
    local $Carp::CarpLevel = 1;
    Carp::cluck("\ntrace magical method") if $self->debug;
}

# 重载句柄以打印出我们的路径
sub pathname {
    my $self = shift;
    confess "i am not a class method" unless ref $self;
    $$self->{PATHNAME} = shift if @_;
    return $$self->{PATHNAME};
}

# 双重模式
sub debug {
    my $self = shift;
    my $var = ref $self ? \"$$self->{DEBUG} : \"our $Debug;
```

```

    $$var = shift if @_;
    return ref $self ? $$self->{DEBUG} || $Debug : $Debug;
}

```

然后象下面这样在所有你的普通方法的入口处调用 **trace**:

```

sub GETC { $_[0]->trace;    # 新的
    my($self) = @_;
    getc($self);
}

```

并且在 **TIEHANDLE** 和 **OPEN** 里设置路径名:

```

sub TIEHANDLE {
    my $class = shift;
    my $form = shift;
    my $name = "$form@_";          # NEW
    open my $self, $form, @_ or croak "can't open $name: $!";
    if ($form =~ />/) {
        print $self "<SHOUT>\n";
        $$self->{WRITING} = 1;      # Remember to do end tag
    }
    bless $self, $class;           # $fh is a glob ref
    $self->pathname($name);         # NEW
    return $self;
}

sub OPEN { $_[0]->trace;          # NEW
    my $self = shift;
    my $form = shift;
    my $name = "$form@_";
    $self->CLOSE;
    open($self, $form, @_ or croak "can't reopen $name: $!";
    $self->pathname($name);         # NEW
    if ($form =~ />/) {
        print $self "<SHOUT>\n" or croak "can't start print: $!";
        $$self->{WRITING} = 1;      # Remember to do end tag
    }
    else {
        $$self->{WRITING} = 0;      # Remember not to do end tag
    }
    return 1;
}

```

有些地方你还需要调用 **\$self->debug(1)** 打开调试。如果你这么做，那么你的所有 **Carp::cluck** 调用都将会生成有意义的信息。下面是我们做上面的 **reopen** 的时候得到的信息。它给我们显示了三个深藏的方法，当时我们正在关闭旧文件并且准备打开新文件:

```

trace magical method at foo line 87
  Shout::SEEK('>filename', '>filename', 0, 2) called at foo line 81
  Shout::CLOSE('>filename') called at foo line 65
  Shout::OPEN('>filename', '+<', 'filename') called at foo line 141

```

14.4.2 创建文件句柄

你可以把同一个文件句柄同时 **tie**（捆绑）到一个两头的管道的输入和输出端。假设你想象下面这样

运行 **bc**（一个任意精度的计算器）程序：

```
use Tie::Open2;

tie *CALC, 'Tie::Open2', "bc -l";
$sum = 2;
for (1 .. 7) {
    print CALC "$sum * $sum\n";
    $sum = <CALC>;
    print "$_: $sum";
    chomp $sum;
}
close CALC;
```

我们可以看到打印出下面的东西：

```
1: 4
2: 16
3: 256
4: 65536
5: 4294967296
6: 18446744073709551616
7: 340282366920938463463374607431768211456
```

如果你的机器里有 **bc** 而且还有象下面这样定义的 **Tie::Open2**，那么你能看到上面预期的输出。这次我们给我们的内部对象用了一个赐福了的数组。它包含我们的两个真正的文件句柄用于读和写。（打开一个双头管道的脏活由**IPC::Open2**干；我们只做有意思的部分。）

```
package Tie::Open2;
use strict;
use Carp;
use Tie::Handle;      # 不要从这里继承
use IPC::Open2;

sub TIEHANDLE {
    my ($class, @cmd) = @_;
    no warnings 'once';
    my @fhpair = \do { local(*RDR, *WTR) };
    bless $_, 'Tie::StdHandle' for @fhpair;
    bless(\@fhpair => $class)->OPEN(@cmd) || die;
    return \@fhpair;
}

sub OPEN {
    my ($self, @cmd) = @_;
    $self->CLOSE if grep {defined} @{$self->FILENO };
    open2(@$self, @cmd);
}

sub FILENO {
    my $self = shift;
    [ map { fileno $self->[$_] } 0, 1];
}

for my $outmeth (qw(PRINT PRINTF WRITE) ) {
    no strict 'refs';
```

```

        *$outmeth = sub {
            my $self = shift;
            $self->[1]->$outmeth(@_);
        };
    }

    for my $inmeth (qw(READ READLINE GETC) ) {
        no strict 'refs';
        *$inmeth = sub {
            my $self = shift;
            $self->[0]->$inmeth(@_);
        };
    }

    for my $doppelmeth (qw(BINMODE CLOSE EOF)) {
        no strict 'refs';
        *$doppelmeth = sub {
            my $self = shift;
            $self->[0]->$doppelmeth(@_)  && $self->[1]->$doppelmeth(@_);
        };
    }

    for my $deadmeth (qw(SEEK TELL)) {
        no strict 'refs';
        *$deadmeth = sub {
            croak("can't $deadmeth a pipe");
        };
    }
}
1;

```

最后四行以我们的观点来说是非常时髦的。为了解释这里在做什么，请回过头看一眼第八章，引用，里面的“作为函数模板的闭合”。

下面是一套更怪异的类。它的包名字应该给你一些关于它是干什么的线索。

```

use strict;
package Tie::DevNull

sub TIEHANDLE {
    my $class = shift;
    my $fh = local *FH;
    bless \$fh, $class;
}

for (qw(READ READLINE GETC PRINT PRINTF WRITE)) {
    no strict 'refs';
    *$_ = sub { return };
}

package Tie::DevRandom;

sub READLINE { rand() . "\n"; }
sub TIEHANDLE {
    my $class = shift;
    my $fh = local *FH;
    bless \$fh, $class;
}

```

```

}
sub FETCH { rand() }
sub TIESCALAR {
    my $class = shift;
    bless \my $self, $class;
}

package Tie::Tee;

sub TIEHANDLE {
    my $class = shift;
    my @handles;
    for my $path (@_) {
        open(my $fh, ">$path") || die "can't write $path";
        push @handles, $fh;
    }
    bless \@handles, $class;
}

sub PRINT {
    my $self = shift;
    my $ok = 0;
    for my $fh (@$self) {
        $ok += print $fh @_;
    }
    return $ok = @$self;
}

```

Tie::Tee 类模拟标准的 **Unix tee (1)** 程序，它把一个输出流发送到多个不同的目的。

Tie::DevNull 类模拟空设备，**Unix** 系统里的 **/dev/null**。而 **Tie::DevRandom** 类生成可以用做句柄或标量的随机数，具体做什么取决于你调用的是 **TIEHANDLE** 还是 **TIESCALAR**！下面是你调用它们的方法：

```

package main;

tie *SCATTER,    "Tie::Tee",   qw(tmp1 - tmp2 >tmp3 tmp4);
tie *RANDOM,      "Tie::DevRandom";
tie *NULL,       "Tie::DevNull";
tie my $randy,   "Tie::DevRandom";

for my $i (1..10) {
    my $line = <RANDOM>;
    chomp $line;
    for my $fh ( *NULL, *SCATTER) {
        print $fh "$i: $line $randy\n";
    }
}

```

这个程序在你的屏幕上输出类似下面的东西：

```

1: 0.124115571686165 0.20872819474074
2: 0.156618299751194 0.678171662366353
3: 0.799749050426126 0.300184963960792
4: 0.599474551447884 0.213935286029916
5: 0.700232143543861 0.800773751296671

```

```
6: 0.201203608274334 0.0654303290639575
7: 0.605381294683365 0.718162304090487
8: 0.452976481105495 0.574026269121667
9: 0.736819876983848 0.391737610662044
10: 0.518606540417331 0.381805078272308
```

不过事还没完！它向你的屏幕输出是因为上面的 ***SCATTER tie** 里的 `-`。而且那一行还命令它创建文件 **tmp1**, **tmp2**, 和**tmp4**, 同时还附加到文件 **tmp3**上。（我们在循环里还向 ***NULL** 输出了, 当然那不会在任何有趣的地方显示任何东西, 除非你对黑洞感兴趣。）

14.5 一个精细的松绑陷阱

如果你试图使用从 **tie** 或者 **tied** 返回的对象, 而且该类定义了一个析构器, 那么你就得小心一个精细的陷阱。看看下面这个（故意设计的）例子类, 它使用一个文件来记录赋予一个标量的所有值:

```
package Remember;

sub TIESCALAR {
    my $class = shift;
    my $filename = shift;
    open (my $handle, ">", $filename)
    or die "Cannot open $filename: $!\n";
    print $handle "The Start\n";
    bless {FH => $handle, VALUE => 0}, $class;
}

sub FETCH {
    my $self = shift;
    return $self->{VALUE};
}

sub STORE{
    my $self = shift;
    my $value = shift;
    my $handle = $self->{FH};
    print $handle "$value\n";
    $self->{VALUE} = $value;
}

sub DESTROY {
    my $self = shift;
    my $handle = $self->{FH};
    print $handle "The End\n";
    close $handle;
}

1;
```

然后是一个利用我们 **Remember** 类的例子:

```
use strict;
use Remember;

my $fred;
$x = tie $fred, "Remember", "camel.log";
```



```
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat camel.log";
```

执行的输出是:

```
The Start
1
4
5
The End
```

到目前为止还不错。现在让我们增加一个额外的方法到 **Remember** 类，这样允许在文件中有注释，也就是象下面这样的东西：

```
sub comment {
    my $self = shift;
    my $message = shift;
    print {$self->{FH}} $handle $message, "\n";
}
```

下面是前面那个例子，修改以后利用 **comment** 方法:

```
use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat camel.log";
```

现在这个文件会是空的，而这样的结果可能不是你想要的。让我们解释一下为什么。捆绑一个变量实际上是把它和构造器返回的对象关联起来。这个对象通常只有一个引用：那个藏在捆绑变量身后的。调用 “**untie**” 打破了这个关联并且消除了该引用。因为没有余下什么指向该对象的引用，那么就会出发 **DESTROY** 方法。

不过，在上面的例子中，我们存贮了第二个指向捆绑到 **\$x** 上的对象。那就意味着在 **untie** 之后，我们还有一个有效的指向该对象的引用。因此 **DESTROY** 就不会触发，于是该文件就得不到输出冲刷并且关闭。这就是没有输出的原因：文件句柄的缓冲区仍然在内存里。它在程序退出之前不会存储到磁盘上。

要想侦测到这些东西，你可以用 **-w** 命令行标志，或者在当前的词法范围里包含 **use warnings** “**untie**” 用法。这两种技巧都等效于在仍然存在有捆绑的对象的一个 **untie** 调用。如果这么处理，Perl 打印下面的警告：

```
untie attempted while 1 inner references still exist
```

要想让程序能够运行而且看不见这些警告，那么就要在调用 **untie** 之前删除任何多余的指向捆绑对象的引用。你可以用下面的方法明确地处理：

```
undef $x;
```

```
untie $fred;
```

不过，通常你可以通过让变量在合适的时刻跑出范围来解决问题。

14.6 CPAN 里的模块

在你开始鼓足干劲写你自己的捆绑模块之前，你应该检查一下是不是已经有人做出来了。在 CPAN 里面有许多捆绑模块，而且每天都在增加。（哦，应该是每个月。）表 14-1 列出了其中的一部分。

表14-1。CPAN 的捆绑模块

模块	描述
GnuPG²::Tie::Encrypt	把一个文件句柄和 GNU Privacy Guard 加密捆绑在一起
IO::WrapTie	把捆绑对象和一个 IO::Handle 接口封装在一起。
MLDBM	在一个DBM文件里透明地存储复杂的数据值，而不仅仅是简单的字串。
Net::NISplusTied	把散列和 NIS+ 表捆绑在一起
Tie::Cache::LRU	实现一个最近最少使用缓冲
Tie::Const	提供常数标量和散列
Tie::Counter	让一个标量变量每接受一次访问就加一
Tie::CPHash	实现一个保留大小写但是又大小写无关的散列
Tie::DB_FileLock	提供对 Berkeley DB 1.x 的锁定访问
Tie::DBI	把散列和 DBI 关系数据库捆绑在一起
Tie::DB_Lock	用共享和排它锁把散列和数据库捆绑在一起
Tie::Dict	把一个散列和一个 RPC 字典服务器捆绑在一起
Tie::Dir	把一个散列捆绑为读取目录
Tie::DirHandle	捆绑目录句柄
Tie::FileLURCache	实现一个轻量级的，基于文件系统的永久的 LRU 缓冲
Tie::FlipFlop	实现一个在两个值之间切换的捆绑
Tie::HashDefaults	令散列有缺省值
Tie::HashHistory	跟踪对散列修改的所有历史
Tie::IxHash	为 Perl 提供有序的关联数组
Tie::LDAP	实现一个 LDAP 数据库的接口
Tie::Persistent	通过 tie 实现一个永久数据结构
Tie::Pick	从一个集合中随机选取（和删除）一个元素
Tie::RDBM	把散列和关系数据库捆绑
Tie::SecureHash	支持基于名字空间的封装
Tie::STDERR	把你的 STDERR 的输出发给另外一个进程，比如一个邮件服务器
Tie::Syslog	把一个文件句柄自动捆绑到 syslog 作为其输出
Tie::TextDir	捆绑一个文件目录
Tie::TransactHash	在事务中编辑一个散列，而又不改变事务的顺序
Tie::VecArray	给一个位矢量提供一个数组接口
Tie::Watch	在 Perl 变量中提供观察点
Win32::TieRegistry	提供有效且简单的操作 Microsoft Windows 注册表的方法。

[Perl](#) > [PerlProgramming3](#) > [TiedVariablesII](#)

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](#)