

## 第十四章 捆绑(tie)变量

- ↓ 第十四章 捆绑(tie)变量
  - ↓ 14.1 捆绑标量
    - ↓ 14.1.1 标量捆绑方法
    - ↓ 14.1.2 魔术计数变量
    - ↓ 14.1.3 神奇地消除 \$ \_
  - ↓ 14.2 捆绑数组
    - ↓ 14.2.1 数组捆绑方法
    - ↓ 14.2.2 大家方便

有些人类的工作需要伪装起来。有时候伪装的目的是欺骗，但更多的时候，伪装的目的是为了在更深层次做一些真实的通讯。比如，许多面试官希望你能穿西服打领带以表示你对工作是认真的，即使你们俩都知道你可能在工作的时候永远不会打领带。你思考这件事的时候可能会觉得很奇怪：在你脖子上系一块布会神奇地帮你找到工作。在 Perl 文化里，**tie** 操作符起到类似的作用的角色：它让你创建一个看起来象普通变量的变量，但是在变量的伪装后面，它实际上是一个羽翼丰满的 Perl 对象，而且此对象有着自己有趣的个性。它只是一个让人有点奇怪的小魔术，就好象从一个帽子里弹出一个邦尼兔那样。（译注：英文 **tie** 做动词有"捆绑"之意，而做名词有"领带"之意。）用另外一个方法来看，在变量名前面的趣味字符 **\$**，**@**，**%**，或者 **\*** 告诉 Perl 和它的程序许多事情——他们每个都暗示了一个特殊范畴的原形特性。你可以利用 **tie** 用各种方法封装那些特性，方法是用一个实现一套新性质的类与该变量关联起来。比如，你可以创建一个普通的 Perl 散列，然后把它 **tie**（绑）到一个类上，这个类把这个散列放到一个数据库里，所以当你从散列读取数据的时候，Perl 魔术般地从外部数据库文件抓取数据，而当你设置散列中的数值的时候，Perl 又神奇地把数据存储在外部数据库文件里。这里的"魔术"，"神奇"指的是“透明地处理一些非常复杂的任务”。你应该听过那些老话：再先进的技术也和 Perl 脚本没什么区别。（严肃地说，那些在 Perl 内部工作的人们把魔术（**magic**）一词当作一个技术术语，特指任何附加在变量上的额外的语义，比如 **%ENV** 或者 **%SIG**。捆绑变量只是其中一种扩展。）

Perl 已经有内建的 **dbmopen** 和 **dbmclose** 函数，它们可以完成把散列变量和数据库系在一起的魔术，不过那些函数的实现是早在 Perl 没有 **tie** 的时候。现在 **tie** 提供了更通用的机制。实际上，Perl 本身就是以 **tie** 的机制来实现 **dbmopen** 和 **dbmclose** 的。

你可以把一个标量，数组，散列或者文件句柄（通过它的类型团）系到任意一个类上，这个类提供合适的命名方法以截获和模拟对这些对象的正常访问。那些方法的第一个是在进行 **tie** 动作本身时调用的：使用一个变量总是调用一个构造器，如果这个构造器成功运行，则返回一个对象，而 Perl 把这个对象藏在一个你看不见的地方——在“普通”变量的深层内部。你总是可以稍后用 **tied** 函数在该普通变量上检索该对象：

```
tie VARIABLE, CLASSNAME, LIST;    # 把VARIABLE 绑定到 CLASSNAME
$object = tied VARIABLE;
```

上面两行等效于：

```
$object = tie VARIABLE, CLASSNAME, LIST;
```

一旦该变量被捆绑，你就可以按照平时那样对待该普通变量，不过每次访问都自动调用下层对象的方法；所有该类的复杂性都隐藏在那些方法调用的背后。如果稍后你想打破变量和类之间的关联，你可以用 **untie**（松绑）那个变量：

```
untie VARIABLE;
```

你几乎完全可以把 **tie** 看作一种有趣的 **bless** 类型，只不过它是给一个光秃秃的变量赐福，而不是给一个对象引用赐福。它同样还可以接收额外的参数，就象构造器那样——这个恐怕也不新鲜了，因为

它实际上就是在内部调用一个构造器，该构造器的名字取决于你尝试的变量类型：是 **TIESCALAR**，**TIEARRAY**，**TIEHASH**，或者**TIEHANDLE**。（注：因为这些构造器是独立的名字，你甚至可以提供一个独立的类来实现它们。那样，你就可以把标量，数组，散列，和文件句柄统统绑定到同一个类上，不过通常不是这么干的，因为它会令其他的魔术方法比较难写。）调用这些构造器的时候，它们用所声明的 **CLASSNAME** 为它们的调用者作为类方法调用，另外把你放在 **LIST** 里的任何东西作为附加的参数。（**VARIABLE** 并不传递给构造器。）

这四种构造器每种都返回一个普通风格的对象。它们并不在乎它们是否从 **tie** 里调用的，类里的其他方法也不在意，因为如果你喜欢的话你总是可以直接调用它们。从某种意义上来说，所有魔术都是在 **tie** 里，而不是在实现 **tie** 的类里。该类只是一个有着有趣的方法名的普通类。（实际上，有些捆绑的模块提供了额外的一些方法，这些方法是不能通过捆绑的变量看到的；你必须明确调用这些方法，就象你对待其他对象方法一样。这样的额外方法可以提供类似文件锁，事务保护，或者任何其他实例方法可以做的东西。）

因此这些构造器就象其他构造器那样 **bless**（赐福）并且返回一个对象引用。该引用不需要指向和被捆绑的变量相同类型的变量；它只是必须被赐福，所以该绑定的变量可以很容易在你的类中找到支持。比如，我们的长例子 **TIEARRAY** 就会用一个基于散列的对象，这样它就可以比较容易地保存它在模拟的数组的附加信息。

**tie** 函数不会为你 **use** 或者 **require** 一个模块——如果必要的话，在调用 **tie**前你必须自己明确地做那件事。（另外，为了保持向下兼容，**dbmopen** 函数会尝试 **use** 一个或者某个 **DBM** 实现。但你可以用一个明确的 **use** 修改它的选择优先级——只要你 **use** 的模块是在 **dbmopen** 的模块列表中的一个。参阅 [AnyDBM<sup>2</sup>\\_File](#) 模块的在线文档获取更完善的解释。）

一个捆绑了的变量调用的方法有一个类似 **FETCH** 和 **STORE** 这样的预定义好了的名字，因为它们是在 **Perl** 内部隐含调用的（也就是说，由特定事件触发）。这些名字都在 **ALLCAPS**，内部隐含调用的（也就是说，有特定事件触发）。这些名字都是全部大写，这是我们遵循的一个称呼这些隐含调用过程的习惯。（其他遵循这种传统的习惯有 **BEGIN**，**CHECK**，**INIT**，**END**，**DESTROY**，和 **AUTOLOAD**，更不用说 **UNIVERSAL->VERSION**。实际上，几乎所有 **Perl** 预定义的变量和文件句柄都是大写的：**STDIN**，**SUUPER**，**CORE**，**CORE::GLOBAL**，**DATA**，**@EXPORT**，**@INC**，**@ISA**，**@ARGV** 和 **%ENV**。当然，内建操作符和用法是另外一个极端，它们没有用大写的。）

我们首先要介绍的内容很简单：如何捆绑一个标量变量。

## 14.1 捆绑标量

---

要实现一个捆绑的标量，一个类必须定义下面的方法：**TIESCALAR**，**FETCH**，和 **STORE**（以及可能还有 **DESTROY**）。当你 **tie** 一个标量变量的时候，**Perl** 调用 **TIESCALAR**。如果你读取这个捆绑的变量，它调用 **FETCH**，并且当你给一个变量赋值的时候，它调用 **STORE**。如果你保存了最初的 **tie**（或者你稍后用 **tied** 检索它们），你就可以自己访问下层的对象——这样并不触发它的 **FETCH** 或者 **STORE** 方法。作为一个对象，这一点毫不神奇，而是相当客观的。

如果存在一个 **DESTROY** 方法，那么当指向被捆绑对象的最后一个引用消失时，**Perl** 就会调用这个 **DESTROY** 方法，就好象对其他对象那样。当你的程序结束或者你调用 **untie** 的时候就会发生这些事情，这样就删除了捆绑使用的引用。不过，**untie** 并不删除任何你放在其他地方的引用；**DESTROY** 也会推迟到那些引用都删除以后。

在标准的 **Tie::Scalar** 模块里有 **Tie::Scalar** 和 **Tie::StdScalar** 包，如果你不想自己定义所有这些方法，那么它们定义了一些简单的基类。**Tie::Scalar** 提供了只能做很有限的工作的基本方法，而 **Tie::StdScalar** 提供了一些方法令一个捆绑了的标量表现得象一个普通的 **Perl** 标量。（好象没什么用，不过有时候你只是想简单地给普通标量语义上加一些封装，比如，计算某个变量设置的次数。）

在我们给你显示我们精心设计的例子和对所有机制进行完整地描述之前，我们先给你来点开胃的东西——并且给你显示一下这些东西是多简单。下面是一个完整的程序：

```
#!/usr/bin/perl
package Centsible;
sub TIESCALAR { bless \my $self, shift }
sub STORE { ${ $_[0] } = $_[1] }      # 做缺省的事情
sub FETCH { sprintf "%.2f", ${ my $self = shift } }  # 圆整值

package main;
tie $bucks, "Centsible";
$bucks = 45.00;
$bucks *= 1.0715;    # 税
$bucks *= 1.0715;    # 和双倍的税!
print "That will be $bucks, please.\n";
```

运行的时候，这个程序生成：

```
That will be 51.67, please.
```

把 **tie** 调用注释掉以后，你可以看到区别：

```
That will be 51.66505125, please.
```

当然，这样要比你平时做圆整所做的工作要多。

### 14.1.1 标量捆绑方法

既然你已经看到我们将要讲的东西，那就让我们开发一个更灵活的标量捆绑类吧。我们不会使用任何封装好了的包做基类（特别是因为标量实在是简单），相反我们会轮流照看一下这四种方法，构造一个名字为 [ScalarFile<sup>2</sup>](#) 的例子类。捆绑到这个类上的标量包含普通字串，并且每个这样的变量都隐含地和一个文件关联，此文件就是字串存贮的地方。（你可以通过给变量命名的方法来记忆你引用的是哪个文件。）变量用下面的方法绑到类上：

```
use ScalarFile;      # 装载 ScalarFile.pm
tie $camel, "ScalarFiel", "/tmp/camel.lot";
```

变量一旦捆绑，它以前的内容就被取代，并且变量和其对象内部的联系覆盖了此变量平常的语义。当你请求 **\$camel** 的值时，它现在读取 **/tmp/camel.lot** 的内容，而当你给 **\$camel** 赋值的时候，它把新的内容写到 **/tmp/camel.lot** 里，删除任何原来的东西。

捆绑是对变量进行的，而不是数值，因此一个变量的捆绑属性不会随着赋值一起传递。比如，假设你拷贝一个已经捆绑了的变量：

```
$dromedary = $camel;
```

Perl 不是象平常那样从 **\$camel** 标量里读取变量，而是在相关的下层对象上调用 **FETCH** 方法。就好象你写的是这样的东西：

```
$dromedary = (tied $camel)->FETCH();
```

或者如果你还记得 **tie** 返回的对象，你可以直接使用那个引用，就象在下面的例子代码里一样：

```
$slot = tie $camel, "ScalarFile", "/tmp/camle.lot";
$dromedary = $camle;      # 通过隐含的接口
$dromedary = $slot->FETCH();  # 一样的东西，不过是明确的方法而已
```

如果除了 **TIESCALAR**, **FETCH**, **STORE**, 和 **DESTROY** 以外, 该类还提供其他方法, 你也可以使用 **\$clot** 手工调用它们。不过, 大家应该做好自己的事情而不要去管下层对象, 这也是为什么你看到来自 **tie** 的返回值常被忽略。如果稍后你又需要该对象 (比如, 如果该类碰巧记载了任何你需要的额外方法的文档), 那么你仍然可以通过 **tie** 获取该对象。忽略所返回的对象同样也消除了某些类型的错误, 这一点我们稍后介绍。

下面是我们的类所需要的东西, 我们将把它们放到 [ScalarFile<sup>2</sup>.pm](#):

```
package ScalarFile;
use Carp;          # 很好地传播错误消息。
use strict;        # 给我们自己制定一些纪律。
use warnings;      # 打开词法范围警告。
use warnings::register; # 允许拥护说"use warnings 'ScalarFile'".
my $count = 0;     # 捆绑了的 ScalarFile2 的内部计数。
```

这个标准的 **Carp** 模块输出 **carp**, **croak**, 和 **confess** 子过程, 我们将在本节稍后的代码中使用它们。和往常一样, 参阅第32章, 标准模块, 或者在线文档获取 **Carp** 的更多介绍。

下面的方法是该类定义的。

### CLASSNAME->TIESCALAR(LIST)

每当你 **tie** 一个标量变量, 都会触发该类的 **TIESCALAR** 方法。可选的 **LIST** 包含 任意正确初始化该对象所需要的参数。(在我们的例子里, 只有一个参数: 该文件的名字。)这个方法应该返回一个对象, 不过这个对象不必是一个标量的 引用。不过在我们的例子里是标量的引用。

```
sub TIESCALAR {      # 在 ScalarFile.pm
    my $class = shift;
    my $filename = shift;
    $count++;      # 一个文件范围的词法, 是类的私有部分
    return bless \$filename, $class;
}
```

因为匿名数组和散列构造器 (**[]**和**{}**) 没有标量等价物, 我们只是赐福一个词法 范围的引用物, 这样, 只要这个名字超出范围, 它就变成一个匿名。这样做运转得 很好 (你可以对数组和散列干一样的事情) ——只要这个变量真的是词法范围。如果你在一个全局量上使用这个技巧, 你可能会以为你成功处理了这个全局量, 但直到你创建另外一个 **camel.lot** 的时候才能意识到这是错的。不要试图写下面 这样的东西:

```
sub TIESCALAR {bless \$_[1], $_[0] }    # 错, 可以引用全局量。
```

一个写得更强壮一些的构造器可能会检查该文件名是否可联合。我们首先检查, 看看这个文件是否可读, 因为我们不想毁坏现有的数值。(换句话说, 我们不应该 假设用户准备先写。他们可能很珍惜此程序以前运行留下来的旧的 **Camel Lot** 文件。)如果我们不能打开或者创建所声明的文件名, 我们将通过返回一个 **undef** 礼貌地指明该错误, 并且还可以通过 **carp** 打印一个警告。(我们还可以只用 **croak** —— 这是口味的问题, 取决于你喜欢鱼还是牛蛙。)我们将使用 **warnings** 用法来判断这个用户是否对我们的警告感兴趣:

```
sub TIESCALAR {      # 在 ScalarFile.pm
    my $class = shift;
    my $filename = shift;
    my $fh;
    if (open $fh, "<", $filename or
        open $fh, ">", $filename)
    {
```

```

        close $fh;
        $count++;
        return bless \$filename, $class;
    }
    carp "Can't tie $filename: $!" if warnings::enabled();
    return;
}

```

有了这样的构造器，我们现在就可以把标量 `$string` 和文件 `camel.lot` 关联在一起了：

```
tie ($string, "ScalarFile", "camel.lot") or die;
```

（我们仍然做了一些不应该做的假设。在一个正式版本里，我们可能打开该文件句柄一次并且在捆绑的过程中记住该文件句柄和文件名，保证此句柄在所有时间里都是用 `flock` 排他锁住的。否则我们就会面对冲突条件——参阅第二十三章，安全，里的“处理计时缝隙”。）

### • SELF->FETCH

当你访问这个捆绑变量的时候就会调用这个方法（也就是说，读取其值）。除了与变量捆绑的对象以外，它没有其他的参数。在我们的例子里，那个对象包含文件名。

```

sub    FETCH {
    my $self = shift;
    confess "I am not a class method" unless ref $self;
    return unless open my $fh, $$self;
    read($fh, my $value, -s $fh);    # NB: 不要在管道上使用 -s
    return $value;
}

```

这回我们决定：如果 `FETCH` 拿到了不是引用的东西，那么就摧毁（抛出一个例外）。（它要么是被当做类方法调用，要么是什么东西不小心把它当成一个子过程调用了。）我们没有其它返回错误的方法，所以这么做可能是对的。实际上，只要我们试图析引用 `$self`，Perl 都会立刻抛出一个例外；我们只是做得礼貌一些并且用 `confess` 把完整的堆栈追踪输出到用户的屏幕上。（如果这个动作可以认为是礼貌的话。）

如果我们说下面这些话就会看到 `camel.lot` 的内容：

```

tie($string, "ScalarFile", "camel.lot");
print $string;

```

### • SELF->STORE(VALUE)

当设置（赋值）捆绑的变量的时候会运行这个方法。第一个参数 `SELF` 与往常一样是与变量关联的对象；`VALUE` 是给变量赋的值。（我们这里的“赋值”的含义比较宽松--任何修改变量的动作都可以叫做 `STORE`。）

```

sub STORE {
    my($self, $value) = @_;
    ref $self or confess "not a class method";
    open my $fh, ">", $$self or croak "can't clobber $$self:$!";
    syswrite($fh, $value) == length $value
    or croak "can't write to $$self:$!";
    close $fh or croak "can't close $$self:$!";
    return $value;
}

```

在给它“赋值”以后，我们返回新值--因为这也是赋值做的事情。如果赋值失败，我们把错误 `croak`



出来。可能的原因是我们没有写该关联文件的权限，或者 磁盘满，或者磁盘控制器坏了。有时候是你控制这些局势，有时候是局势控制你。

现在如果我们说下面的话，我们就可以写入 `camel.lot` 了。

```
tie($string, "ScalarFile", "camel.lot");
$string = "Here is the first line of camel.lot\n";
$string .= "And here is another line, automatically appended.\n";
```

## • SELF->DESTROY

当与捆绑变量相关联的对象即将收集为垃圾时会触发这个方法，尤其是在做一些 特殊处理以清理自身的情况下。和其他类一样，这样的方法很少是必须的，因为 `Perl` 自动为你清除垂死的对象的内存。在这里，我们会定义一个 `DESTROY` 方法 用来递减我们的捆绑文件的计数：

```
sub DESTROY {
    my $self = shift;
    confess "wrong type" unless ref $self;
    $count--;
}
```

我们还可以提供一个额外的类方法用于检索当前的计数。实际上，把它称做类方法 还是对象方法并不要紧，但是你在 `DESTROY` 之后就不再拥有一个对象了，对吧？

```
sub count {
    # my $invocant = shift;
    $count;
}
```

你可以在任何时候把下面这个称做一个类方法：

```
if (ScalarFile->count) {
    warn "Still some tied ScalarFiles sitting around somewhere...\n";
}
```

这就是所要的东西。实际上，比所要的东西还要多，因为我们在这里为完整性， 坚固性和普遍美学做了几件相当漂亮的事情。当然，更简单的 `TIESCALAR` 类 也是可能的。

## 14.1.2 魔术计数变量

这里是一个简单的 `Tie::Counter` 类，灵感来自 `CPAN` 中同名的模块。捆绑到这个类上的变量每次自增 1。比如：

```
tie my $counter, "Tie::Counter", 100;
@array = qw /Red Green Blue/;
for my $color (@array) {      # 打印:
    print " $counter $color\n";  # 100 Red
}                               # 101 Green
                                # 102 Blue
```

构造器把一个可选的额外参数当作计数器的初始值，缺省时为零。给这个计数器赋值将设置一个新值。下面是类：

```
package Tie::Counter;
sub FETCH    { ++ ${ $_[0] } }
sub STORE    { ${ $_[0] } = $_[1] }
sub TIESCALAR {
```

```

    my ($class, $value) = @_;
    $value = 0 unless defined $value;
    bless \$value => $class;
}
1; # 如果在模块里需要这个

```

多小！看到了吗？要写这么一个类并不需要多少代码。

### 14.1.3 神奇地消除 \$ \_

这个让人好奇的外部的捆绑类用于防止非局部的 \$ \_ 的使用。它不是用 **use** 把方法拉进来，而是调用该类的 **import** 方法，装载这个模块应该用 **no**，以便调用很少用的 **unimport** 方法。用户说：

```
no Underscore;
```

然后所有把 \$ \_ 当作一个非局部的全局变量使用就都会产生一个例外。

下面是一个用这个模块的小测试程序：

```

#!/usr/bin/perl
no Underscore;
@tests = (
    "Assignment" => sub { $ _ = "Bad" },
    "Reading"    => sub { print },
    "Matching"   => sub { $x = /badness/ },
    "Chop"       => sub { chop },
    "Filetest"   => sub { -x },
    "Nesting"    => sub { for (1..3) { print } },
);

while ( ($name, $code) = splice(@tests, 0, 2) ) {
    print "Testing $name: ";
    eval { &$code };
    print $@ ? "detected" : " missed!";
    print "\n";
}

```

这个程序打印出下面的东西：

```

Testing Assignment: detected
Testing Reading: detected
Testing Matching: detected
Testing Chop: detected
Testing Filetest: detected
Testing Nesting: 123 missed!

```

“丢失”了最后一个是因为它由 **for** 循环正确地局部化了，并且因此可以安全地访问。

下面是让人感兴趣的外部 **Underscore** 模块本身。（我们说过它是让人感兴趣的外部吗？）它能运转是因为捆绑的神奇变量被一个 **local** 有效地隐藏起来了。该模块在它自己的初始化代码里做了一个 **tie**，所以 **require** 也能运转。

```

package Underscore;
use Carp;
sub TIESCALAR { bless \my $dummy => shift }
sub FETCH { croak 'Read access to $ _ forbidden' }
sub STORE { croak 'Write access to $ _ forbidden' }

```

```
sub unimport { tie($_, __PACKAGE__) }
sub import   { untie $_ }
tie($_, __PACKAGE__) unless tied $_;
1;
```

在你的程序里对这个类混合调用 `use` 和 `no` 几乎没有任何用处，因为它们都在编译时发生，而不是运行时。你可以直接调用 `Underscore->import` 和 `Underscore->unimport`，就象 `use` 和 `no` 那样。通常，如果你想反悔并且让自己可以使用 `$_`，你就要对它使用 `local`，这也是所有要点所在。

## 14.2 捆绑数组

---

一个实现捆绑数组的类至少要定义方法 `TIEARRAY`，`FETCH`，和 `STORE`。此外还有许多可选方法：普遍存在的 `DESTROY` 方法，还有用于提供  `$#array` 和 `scalar(@array)` 访问的 `STORESIZE` 和 `FETCHSIZE` 方法。另外，当 Perl 需要清空该数组时会触发 `CLEAR`，而当 Perl 在一个真正的数组上需要预先扩充（空间）分配的时候需要 `EXTEND`。

如果你想让相应的函数在捆绑的数组上也能够运行，你还可以定义 `POP`，`PUSH`，`SHIFT`，`UNSHIFT`，`SPLICE`，`DELETE`，和 `EXISTS` 方法。`Tie::Array` 类可以作为一个基类，用于利用 `FETCH` 和 `STORE` 实现前五个函数。（`Tie::Array` 的 `DELETE` 和 `EXISTS` 的缺省实现只是简单地调用 `croak`。）只要你定义了 `FETCH` 和 `STORE`，那么你的对象包含什么样的数据结构就无所谓了。

另外，`Tie::StdArray` 类（在标准的 `Tie::Array` 模块中定义）提供了一个基类，这个基类的缺省方法假设此对象包含一个正常的数组。下面是一个利用这个类的简单的数组捆绑类。因为它使用了 `Tie::StdArray` 做它的基类，所以它只需要定义那些应该以非标准方法对待的方法。

```
#!/usr/bin/perl
package ClockArray;
use Tie::Array;
our @ISA = 'Tie::StdArray';
sub FETCH {
    my ($self, $place) = @_;
    $self->[$place % 12 ];
}

sub STORE {
    my($self, $place, $value ) = @_;
    $self->[$place % 12 ] = $value;
}

package main;
tie my @array, 'ClockArray';
@array = ( "a" ... "z" );
print "@array\n";
```

运行的时候，这个程序打印出 `"y z o p q r s t u v w x"`。这个类提供一个只有一打位置的数组，类似一个时钟的小时数，编号为 `0` 到 `11`。如果你请求第十五个元素，你实际上获得第三个。把它想象成一个旅游助手，用于帮助那些还没有学会如何读24小时时钟的人。

### 14.2.1 数组捆绑方法

前面的是简单的方法。现在让我们看看真正的细节。为了做演示，我们将实现一个数组，这个数组的范围在创建的时候是固定的。如果你试图访问任何超出该界限的东西，则抛出一个例外。比如：



```
use BoundedArray;
tie @array, "BoundedArray", 2;

$array[0] = "fine";
$array[1] = "good";
$array[2] = "great";
$array[3] = "whoa";    # 禁止，显示一个错误信息。
```

这个类的预定义的代码如下：

```
package BoundedArray;
use Carp;
use strict;
```

为了避免稍后定义 **SPLICE**，我们将从 **Tie::Array** 类中继承：

```
use Tie::Array;
our @ISA = ("Tie::Array");
```

### **CLASSNAME->TIEARRAY(LIST)**

是该类的构造器，**TIEARRAY** 应该返回一个赐福了的引用，通过该引用模拟这个 捆绑了的数组。

在下一个例子里，为了告诉你并非一定要返回一个数组的引用，我们选择了一个 散列引用来代表我们的对象。散列很适合做通用记录类型：散列的“**BOUND**”键字 将存储最大允许的范围，而其“**DATA**”值将保存实际的数据。如果有类以外的 解引用返回的对象（就是一个数组引用，不用怀疑），则抛出一个例外。

```
sub TIEARRAY {
    my $class = shift;
    my $bound = shift;
    confess "usage: tie( \@ary, 'BoundedArray', max_subscript)"
        if @_ || $bound =~ /\D/;
    return bless { BOUND => $bound, DATA => [] }, $class;
}
```

现在我们可以说：

```
tie( @array, "BoundedArray", 3);    # 允许的最大索引是 3
```

以确保该数组永远不会有多个元素。当对数组的一个独立的元素进行访问或者 存储的时候，将调用 **FETCH** 和 **STORE**，就好像是处理标量一样，不过有一个 额外的索引参数。

### **SELF->FETCH(INDEX)**

当访问捆绑的数组里的一个独立的元素的时候调用这个方法。它接受对象后面的 一个参数：我们试图抓取的数值的索引。

```
sub FETCH {
    my ($self, $index ) = @_;
    if ($index > $self->{BOUND} ) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    return $self->{DATA}[$index];
}
```

### **SELF->STORE(INDEX, VALUE)**

当设置捆绑了的数组里的一个元素的时候调用这个方法。它接受对象后面的两个 参数：我们试图存储的东西的索引和我们准备放在那里的数值。比如：

```
sub STORE {
    my($self, $index, $value) = @_;
    if ($index > $self->{BOUND} ) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    return $self->{DATA}[$index] = $value;
}
```

### **SELF->DESTROY**

当需要删除捆绑变量和回收它的内存的时候调用这个方法。对于一门有垃圾回收 功能的语言来说，这个东西几乎用不上，所以本例中我们忽略它。

### **SELF->FETCHSIZE**

FETCHSIZE 方法应该返回与 SELF 关联的捆绑数组的条目的总数。它等效于 `scalar(@array)`，通常等于 `$#array + 1`。

```
sub FETCHSIZE {
    my $self = shift;
    return scalar @{$self->{DATA}};
}
```

### **SELF->STORESIZE(COUNT)**

这个方法把与 SELF 关联的捆绑数组的条目总数设置为 COUNT。如果此数组收缩，那么你应该删除超出 COUNT 范围的记录。如果数组增长，你应该确保新的空位置 是未定义的。对于我们的 [BoundedArray<sup>2</sup>](#) 类，我们还要确保该数组不会增长得超出 初始化时设置的限制。

```
sub STORESIZE {
    my ($self, $count) = @_;
    if ($count > $self->{BOUND} ) {
        confess "Array OOB: $count > $self->{BOUND}";
    }
    ${$self->{DATA}} = $count;
}
```

### **SELF->EXTEND(COUNT)**

Perl 使用 EXTEND 方法来表示一个数组将要扩展成保存 COUNT 条记录。这样你就可以一次分配足够大的内存，而不是以后的多次调用。因为我们的 [BoundedArrays<sup>2</sup>](#) 已经固定了上限，所以我们不用定义这个方法。

### **SELF->EXISTS(INDEX)**

这个方法验证在 INDEX 位置的元素存在于捆绑数组中。对于我们的 [BoundedArray<sup>2</sup>](#)，我们只需要在核实了查找企图没有超过我们的固定上限，然后就可以使用 Perl内建的 `exists` 来核实。

```
sub EXISTS {
    my ($self, $index) = @_;
    if ( $index > $self->{BOUND}) {
        confess "array OOB: $index > $self->{BOUND}";
    }
}
```

```

    }
    exists $self->{DATA}[$index];
}

```

### ***SELF->DELETE(INDEX)***

DELETE 方法从捆绑数组 SELF 中删除在 INDEX 位置的元素。对于我们的 [BoundedArray<sup>2</sup>](#) 类，这个方法看起来几乎和 EXISTS 完全一样，不过这可不是 标准。

```

sub DELETE {
    my ($self, $index) = @_;
    print STDERR "deleting!\n";
    if ($index > $self->{BOUND} ) {

        confess "Array OOB: $index > $self->{BOUND}";
    }
    delete $self->{DATA}[$index];
}

```

### ***SELF->CLEAR***

当这个数组需要清空的时候调用这个方法。当该数组设置为一列新值（或者一列 空值）的时候发生这个动作，不过不会在提供给 undef 函数的时候发生这个 动作。因为一个清空了的 boundedArray 总是满足上限，所以我们在这里不需要 检查任何东西：

```

sub CLEAR {
    my $self = shift;

    $self->{DATA} = [];
}

```

如果你把数组设置为一个列表，这个动作会触发 CLEAR，但是看不到列表数值。 因此如果你象下面这样违反上界：

```

tie(@array, "BoundedArray", 2);
@array = (1,2,3,4);

```

CLEAR 方法仍将返回成功。而只有在随后的 STORE 中才会产生例外。这样的赋值 触发一个 CLEAR 和四个 STORES。

### ***SELF->PUSH(LIST)***

这个方法把 LIST 的元素附加到数组上。下面是它在我们 [BoundedArray<sup>2</sup>](#) 类里的 运行方法：

```

sub PUSH {
    my $self = shift;
    if (@_ + ${$self->{DATA}} > $self->{BOUND} ) {
        confess "Attempt to push too many elements";
    }
    push @{$self->{DATA}}, @_;
}

```

### ***SELF->UNSHIFT(LIST)***

这个方法预先把 LIST 的元素放到数组中。对于我们的 [BoundedArray<sup>2</sup>](#) 类而言， 这个子过程类似 PUSH。

### ***SELF->POP***

POP 方法从数组中删除最后一个元素并返回之。对于 `boundedArray`，它只有一行：`sub POP { my $self = shift; pop @{$self->{DATA}}}`

### ***SELF->SHIFT***

SHIFT 方法删除数组的第一个元素并返回之。对于 `boundedArray`，它类似 POP。

### ***SELF->SPLICE(OFFSET, LENGTH, LIST)***

这个方法让你接合 SELF 数组。为了模拟 Perl 内建的 `splice`，OFFSET 应该是可选项并且缺省为零，而负值是从数组后面向前数。LENGTH 也应该是可选项，缺省为数组剩下的长度。LIST 可以为空。如果正确模拟内建函数，那么它应该返回原数组从 OFFSET 开始的 LENGTH 长个元素（要被 LIST 代替的元素列表）。

因为接合是有些复杂的操作，我们就不定义它了；我们只需要使用来自 `Tie::Array` 模块的 `SPLICE` 子过程，这个子过程是继承 `Tie::Array` 时免费时免费拿到的。这样我们用其他 `BoundedArray2` 方法定义了 `SPLICE`，因此范围检查 仍然进行。

上面就是 `BoundedArray2` 类的全部。它只是对数组的语义做了一点点封装。不过我们可以干得更好，而且用的空间更少。

## **14.2.2 大家方便**

变量的一个好特性是它可以代换。函数的一样不太好的特点是它不能代换。你可以使用捆绑的数组把一个函数做成可以代换。假设你想代换一个字串里的任意整数。你可以就说：

```
#!/usr/bin/perl
package RandInterp;
sub TIEARRAY { bless \my $self};
sub FETCH { int rand $_[1] };

package main;
tie @rand, "RandInterp";

for (1,10,100,1000) {
    print "A random integer less than $_ would be $rand[$_]\n";
}
$rand[32] = 5;    # 这么干会重新格式化我们的系统表了么？
```

当运行的时候，它打印出下面的内容：

```
A random integer less than 1 would be 0
A random integer less than 10 would be 3
A random integer less than 100 would be 46
A random integer less than 1000 would be 755
Can't locate object method "STORE" via package "RandInterp" at foo line 10.
```

如你所见，我们甚至还没能实现 `STORE`，不过这不算什么。我们只是和往常一样把它去掉了。

---

Revision: r1.3 - 05 Sep 2005 - 10:19 - [TingYu](#)

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

[反馈意见](#)