

第八章，引用

- ↓ 第八章，引用
 - ↓ 8.1 什么是引用？
 - ↓ 8.2 创建引用
 - ↓ 8.2.1 反斜杠操作符
 - ↓ 8.2.2 匿名数据
 - ↓ 8.2.2.1 匿名数组组合器
 - ↓ 8.2.2.2 匿名散列组合器
 - ↓ 8.2.2.3 匿名子过程组合器
 - ↓ 8.2.3 对象构造器
 - ↓ 8.2.4 句柄引用
 - ↓ 8.2.5 符号表引用
 - ↓ 8.2.6 引用的隐含创建
 - ↓ 8.3 使用硬引用
 - ↓ 8.3.1 把一个变量当作变量名使用
 - ↓ 8.3.2 把一个 **BLOCK** 块当作变量名用
 - ↓ 8.3.3 使用箭头操作符
 - ↓ 8.3.4 使用对象方法
 - ↓ 8.3.5 伪散列
 - ↓ 8.3.6 硬引用可以用的其他技巧
 - ↓ 8.3.7 闭合（闭包）
 - ↓ 8.3.7.1 用闭合做函数模板
 - ↓ 8.3.7.2 嵌套的子过程
 - ↓ 8.4 符号引用
 - ↓ 8.5 花括弧，方括弧和引号
 - ↓ 8.5.1 引用不能当作散列键字用
 - ↓ 8.5.2 垃圾收集，循环引用和弱引用

不管是从理论还是实践的角度出发，**Perl** 都是偏爱平面线性的数据结构的。并且对许多问题来说，这些也就是你所要的东西。

假设你想制作一个简单的表（二维数组），为一组人员显示生命数据用——包括年龄，眼睛颜色，和重量等。你可以通过先给每个独立的成员创建一个数组来实现这个目的。

```
@john = (47, "brown", 186);
@mary = (23, "hazel", 128);
@bill = (35, "blue", 157);
```

然后你就可以构造一个附加的数组，该数组由其他数组的名字组成：

```
@vitals = ('john', 'mary', 'bill');
```

在小镇上过了一夜之后，为了把 **John** 的眼睛变成“红色”（“red”），我们需要一个仅仅通过使用字符串“john”就可以改变数组 **@john** 的内容的方法。这就是间接的基本问题，而不同的语言是用不同的方法来解决这个问题的。在 **C** 里，间接的最常见的形式就是指针，它可以让一个变量保存另外一个变量的内存地址。在 **Perl** 里，间接的最常见的形式是引用。

8.1 什么是引用？

在我们的例子里，**\$vitals[0]** 的值是“john”。也就是说它正好包含另外一个（全局）变量的名字。我们说第一个变量提到了第二个变量，并且这种参考叫符号引用，因为 **Perl** 必须在一个符号表里找出 **@john** 来才能找到它。（你可以把符号引用看作类似文件系统中的符号联接的东西）。我们将在本章晚些时候讨论符号引用。

另外一种引用是硬引用，这种引用是大多数 **Perl** 程序员用来实现它们的间接访问方法（只要不是他

们的草率行为)。我们叫它们硬引用并不是因为它们用起来很难(译注:“hard reference”硬引用,在英文中“hard”有“困难,硬”的意思),而是因为它们现实并且存在的。如果你愿意,那么你可以把硬引用当作真正的引用而把符号引用当作虚假的引用。它们的区别就好象真正的友谊和见面打个招呼一样。如果我们没有声明我们指的是哪种引用,那么我们说的就是硬引用。图8-1 描述了一个叫 `$bar` 的变量引用了一个叫 `$foo` 的变量的内容,而 `$foo` 的值是“bot”。

和符号引用不同的是,真实引用所引用的不是另外一个变量的名字(名字只是一个数值的容器),而是实际的数值本身,是一些内部的数据团。我们没有什么好字眼来描述这样的东西,可是我们又不得不描述,于是我们就叫它引用。举例来说,假如你创建了一个指向某个词法范围数组 `@array` 的硬引用。那么,即使在 `@array` 超出了范围之后,该引用以及它所引用的参考物也仍然继续存在。一个引用只有在对它的所有引用都消失之后才会被摧毁。

除了指向它的引用之外,引用实际上并没有自己的名字。换句话说,每个 Perl 变量都存储在某个符号表里,保存着一个指向所引用的东西的硬引用(否则就没有名字)。引用物可以很简单,比如一个数字或者字符串,也可以很复杂,比如一个数组或散列。不管哪种情况,从变量到数值之间都只有一个引用。你可以创建指向相同引用物的额外的引用,但该变量并不知道(或在乎)这些引用。(注:如果你觉得奇怪,那么你可以用 `Devel::Peek` 模块计算引用计数,这个包是和 Perl 捆绑发布的。)

符号引用只是一个字符串,它的值碰巧和包的符号表里什么东西的名字相同。它和你平时处理的字符串没有什么太大的区别。但是硬引用却是完全不同的家伙。它是三种基本的标量类型中的第三种,其他两种是字符串和数字。硬引用除了指向某些事物之外并不知道它们的名字,并且这些引用物在一开始的时候并没有名字也是非常正常的事情。这样的未名引用叫做匿名,我们将在下面的“匿名数据”里讨论它们。

在本章的术语里,引用一个数值就是创建一个指向它的硬引用。(我们有一个操作符用于这种创建动作)。这样创建的引用只是一个简单的标量,它和所有其他标量一样在我们熟悉的环境里有着一样的行为。给这个标量解引用(析引用)意味着我们使用这个引用访问引用物。引用和解引用都是只发生在某些明确的机制里,在 Perl 里从来不会出现隐含地引用或解引用的现象。哦,是几乎从来不发生。

一次函数调用可以使用明确的引用传参语意——只要它有这样声明的原型。如果是这样,那么函数的调用者并不明确地传递一个引用,但是你在函数里面还是要明确的对它(参数)进行解引用。参阅第六章,子过程,里的“原型”节。如果从绝对诚实的角度出发,那么你在使用某些类型的文件句柄的时候仍然是有一些隐藏在幕后的解引用发生,但是这么做是为了保持向下兼容,并且对于那些偶然使用到的用户来说是透明的。最后,有两个内建的函数, `bless` 和 `block`, 它们都接受一个引用作为自己的参数,但是都会隐含地对这些引用进行解引用以实现各自的功能。不过除了我们这里招供的这些以外,基本的概念还是一致的,那就是 Perl 并不想介入你自己的间接层次中。

一个引用可以指向任何数据结构。因为引用是标量,所以你可以把它们保存在数组和散列里,因此我们就可以做出数组的数组,散列的数组,数组的散列,散列和函数的数组等。在第九章,数据结构,里有这些东西的例子。

不过,你要记住的是,Perl 里的数组和散列都是故意作成一维的。也就是说,它们的元素只能保存标量值(字符串,数字,和引用)。当我们使用“数组的数组”这样的习语的时候,我们的意思实际上是“一个保存指向一些数组的引用的数组”,就好象我们说“函数散列”的时候,我们实际上是说“一个保存着一些指向子过程的引用的散列”。但因为引用是在 Perl 里实现这些构造的唯一方法,所以我们那些稍微短的并非准确的习语也就并不是完全不对,因此也不应该完全忽视,除非你碰到准确性问题。

8.2 创建引用

创建引用的方法有好多种,我们在讲述它们的时候大多会先描述它们,然后才解释如何使用(解引

用) 所生成的引用。

8.2.1 反斜杠操作符

你可以用一个反斜杠创建一个指向任何命名变量或者子过程的引用。(你还可以把它用于一个匿名标量值, 比如 `7` 或 `"camel"`, 尽管你通常并不需要这些东西。)乍一看, 这个操作符的作用类似 `C` 里的 `&` (取址) 操作符。

下面是一些例子:

```
$scalarref = \ $foo;
$constref  = \186_282.42;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*STDOUT;
```

反斜杠操作符可以做的事情远远不止生成一个引用。如果你对一个列表使用反斜杠, 那么它会生成一整列引用。参阅“你用硬引用可以实现的其他技巧”一节。

8.2.2 匿名数据

在我们刚刚显示的例子里, 反斜杠操作符只是简单地复制了一个已经存在于一个命名变量上的引用——但有一个例外。`186_282.42` 不是一个命名变量的引用——它只是一个数值。它是那种我们早先提到过的匿名引用。匿名引用物只能通过引用来访问。我们这个例子碰巧是一个数字, 但是你也可以创建匿名数组, 散列, 和子过程。

8.2.2.1 匿名数组组合器

你可以用方括弧创建一个指向匿名数组的引用:

```
$arrayref = [1, 2, ['a', 'b', 'c', 'd']];
```

在这里我们组合成了一个三个元素的匿名数组, 该数组最后一个元素是一个指向有着四个元素的匿名数组 (在图8-2里演示)。(我们稍后描述的多维语法可以用于访问这些东西。比如, `$arrayref->[2][1]` 将具有数值“b”。)

现在有一个方法来表示我们本章开头的表:

```
$table = [ [ "john", 47, "brown", 186],
            [ "mary", 23, "hazel", 128],
            [ "bill", 35, "blue", 157] ];
```

只是当 `Perl` 分析器在一个表达式里需要项的时候, 方括弧才可以这样运做。你可不要把它们和表达式里的方括弧混淆起来——比如 `$array[6]`——尽管与数组的记忆性关联是有意为之的。在一个引起的字符串里, 方括弧并不组成匿名数组; 相反, 它们成为字符串里的文本字符。(在字符串里方括弧的确仍然可以当作脚标使用, 否则你就不能打印象 `"VAL=$array[6]\n"` 这样的字符串。如果要我们绝对诚实, 你实际上是可以偷偷地把匿名数组组合器放到字符串里, 但只能是在它被潜入到一个更大的表达式中, 并且该表达式被代换的情况下才可以实现。我们将在本章稍后讲述这个酷酷的特性, 因为它既包括引用也包括解引用。)

8.2.2.2 匿名散列组合器

你可以用花括弧创建一个指向匿名散列的引用:

```
$hashref = {
```

```
'Adam' => 'Eve',
'Clyde' => $bonnie,
'Antony' => 'Cleo' . 'patra',
};
```

对于散列的数值（但不是键字），你可以自由地混合其他匿名数组，散列，和子过程，组合成你需要的复杂数据结构。

现在有了表示本章开头的表的另外一种方法：

```
$stable = {
    "john" => [47, "brown", 186],
    "mary" => [23, "hazel", 128],
    "bill" => [35, "blue", 157],
};
```

这是一个数组散列。选择最好的数据结构是难度很高的工种，下一章专门讲这个。但是为了恶作剧，我们甚至可以将散列的散列用于我们的表：

```
$stable = {
    "john" => { age    => 47,
               eyes   => "brown",
               weight => 186,
             },
    "mary" => { age    => 23,
               eyes   => "hazel",
               weight => 128,
             },
    "bill" => { age    => 35,
               eyes   => "blue",
               weight => 157,
             },
};
```

和方括弧一样，只有在 **Perl** 分析器在表达式里需要一个项的时候，花括弧才运做。你也不要把它和在表达式里的花括弧混淆了，比如 **\$hash{key}**——尽管与散列的记忆性关联（也）是有意为之的。同样的注意事项也适用于在字符串里的花括弧。

但是有另外一个注意事项并不适用于方括弧。因为花括弧还可以用于几种其他的东西（包括块），有时候你可能不得不在语句的开头放上一个 **+** 或者一个 **return** 来消除这个位置的花括弧的歧义，这样 **Perl** 就会意识到这个开花括弧不是引出一个块。比如，如果你需要一个函数生成一个新散列然后返回一个指向它的引用，那么你有下列选择：

```
sub hashem {      { @_ }    }    # 不声不响地错误 -- returns @_
sub hashem {      +{ @_ }    }    # 对
sub hashem { return { @_ }    }    # 对
```

8.2.2.3 匿名子过程组合器

你可以通过用不带子过程名字的 **sub** 创建一个匿名子过程：

```
$coderef = sub { print "Boink!\n" };    # 现在 &$coderef 打印 "Boink!"
```

请注意分号的位置，在这里要求用分号是为了终止该表达式。（在更常见的声明和定义命名子过程的用法 **sub NAME {}** 里面不需要这个分号。）一个没有名字的 **sub {}** 并不象是个声明而更象一个操作符——象 **do {}** 或 **eval {}** 那样——只不过在里面的代码并不是立即执行的。它只是生成一个

指向那些代码的引用，在我们的例子里是存储在 `$coderef` 里。不过，不管你执行上面的行多少次，`$coderef` 都仍然将指向同样的匿名子过程。（注：不过就算只有一个匿名子过程，也有可能有好几份词法变量的拷贝被该子过程使用，具体情况取决于子过程生成的时候。这些东西在稍后的“闭合”（闭包）里讨论）。

8.2.3 对象构造器

子过程也可以返回引用。这句话听起来有些陈腐，但是有时候别人要求你用一个子过程来创建引用而不是由你自己创建引用。特别是那些叫构造器的特殊子过程创建并返回指向对象的引用。对象只是一种特殊的引用，它知道自己是和哪个类关联在一起的，而构造器知道如何创建那种关联关系。这些构造器是通过使用 `bless` 操作符，将一个普通的引用物转换成一个对象实现的，所以我们可以认为对象是一个赐过福的引用。（译注：`bless` 在英文中原意是“赐福”，Perl 中使用这样的词作为操作符名称，想必和 Larry 先生是学习语言出身，并且是虔诚的教徒的背景有关系吧，不过这个词的确非常贴切的形容了该操作符的作用，所以，我就直译为“赐福”，希望不会破坏原味。）这儿可和宗教没什么关系；因为一个类起的作用是用用户定义类型，给一个引用赐福只是简简单单地把它变成除了内建类型之外的用户定义类型。构造器通常叫做 `new`——特别是 C++ 程序员更是如此看待——但在 Perl 里它们可以命名为任何其他名字。

构造器可以用下列任何方法调用：

```
$objref = Doggie::->new(Tail => 'short', Ears => 'long'); #1
$objref = new Doggie:: Tail => 'short', Ears => 'long'; #2
$objref = Doggie->new(Tail => 'short', Ears => 'long'); #3
$objref = new Doggie Tail => 'short', Ears => 'long'; #4
```

第一个和第二个调用方法是一样的。它们都调用了 `Doggie` 模块提供的一个叫 `new` 的函数。第三个和第四个调用和头两个是一样的，只不过稍微更加模糊一些：如果你定义了自己的叫 `Doggie` 的子过程，那么分析器将被你弄糊涂。（这就是为什么人们坚持把小写的名字用于子过程，而把大写的名字用于模块。）如果你定义了自己的 `new` 子过程，并且偏偏有没有用 `require` 或者 `use` 使用 `Doggie` 模块（这两个都有声明该模块的作用。），那么第四个方法也会引起歧义。如果你想用方法 4，那么最好声明你的模块。（并且还要注意看看有没有 `Doggie` 子过程。）参阅第十二章，对象，看看有关 Perl 对象的讨论。

8.2.4 句柄引用

你可以通过引用同名的类型团来创建指向文件句柄或者目录句柄：

```
splutter(\*STDOUT);

sub splutter {
    my $fh = shift;
    print $fh = "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

如果你是在传递文件句柄，你还可以使用文件句柄来实现：在上面的例子中，你可以使用 `*STDOUT` 或者 `*STDIN`，而不是 `*STDOUT` 和 `*STDIN`。

尽管通常你可以互换地使用类型团和指向类型团的引用，但还是有少数几个地方是不可以这么用的，

简单的类型团不能 **bleed** 成对象，并且类型团引用无法传递出一个局部化了的类型团的范围。

当生成新的文件句柄的时候，老的代码通常做类似下面这样的东西来打开一个文件列表：

```
for $file (@name) {
    local *FH;
    open(*FH, $file) || next;
    $handle($file) = *FH;
}
```

这么做仍然可行，但是现在我们有更简单的方法：就是让一个未定义的变量自动激活一个匿名类型团：

```
for $file (@name) {
    my $fh;
    open($fh, $file) || next;
    $handle($file) = $fh;
}
```

使用间接文件句柄的时候，Perl 并不在意你使用的是类型团，还是指向类型团的引用，或者是更奇异的 I/O 对象中的一个。就大多数目的来说，你几乎可以没有任何区别地使用类型团或者类型团引用之一。但是我们前面也承认过，这两种做法仍然有一些隐含的细小区别。

8.2.5 符号表引用

在特别的环境里，你在开始写程序的时候可能不知道你需要什么样的引用。你可以用一种特殊的语法创建引用，人们常说是 ***foo{THING}** 语法。***foo{THING}** 返回一个指向 ***foo** 里面 **THING** 槽位的引用，这个引用就是在符号表里保存 **\$foo**，**@foo**，**%foo**，和友元的记录。

```
$scalarref = *foo{SCALAR};      # 和 \ $foo 一样
$arrayref  = *ARGV{ARRAY};      # 和 \@ARGV 一样
$hashref   = *ENV{HASH};        # 和 \%ENV 一样
$coderef   = *handler{CODE};    # 和 \&handler 一样
$globref   = *foo{GLOB};        # 和 \*foo 一样
$ioref     = *STDIN{IO};        # ? ...
```

所有这些语句都具有自释性，除了 ***STDIN{IO}** 之外。它生成该类型团包含的实际的内部 **IO::Handle** 对象，也就是各种 I/O 函数实际上感兴趣的类型团的部分。为了和早期版本的 Perl 兼容，***foo{FILEHANDLE}** 是上面的 ***foo{IO}** 说法的一个同义词。

理论上来说，你可以在任何你能用 ***HANDLE** 或者 ***HANDLE** 的地方使用 ***HANDLE{IO}**，比如将文件句柄传入或者传出子过程，或者在一个更大的数据结构里存储它们。（实际上，仍然有一些地方不能这么互换着用。）它们的优点是它们只访问你需要的真实的 I/O，而不是整个类型团，因此如果你通过一个类型团赋值剪除了比你预计的要多的东西也不会有什么风险（但如果你总是给一个标量变量赋值而不是给类型团赋值，那么你就 OK 了）。缺点是目前没有自动激活这么一个。（注：目前，**open my \$fh** 自动激活一个类型团而不是一个 **IO::Handle** 对象，不过有朝一日我们总会修正这个问题的，所以你不应该依赖 **open** 目前自动激活的类型团的特征）。

```
splutter(*STDOUT);
splutter(*STDOUT{IO});

sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}
```

上面对 `splutter` 的两个调用都打印 `"her um well a hmm"`。

如果编译器还没有看到特定的 `THING`，那么 `*foo{THING}` 这样的构造返回 `undef`。而且 `*foo{SCALAR}` 返回一个指向一个匿名标量的引用，即使编译器还没有看到 `$foo`。（Perl 总是给任何类型团加一个标量，它把这个动作当作一个优化，以便节省其他的什么地方的一些代码。但是在未来的版本中不要指望 Perl 保持这种做法。）

8.2.6 引用的隐含创建

创建引用的最后一种做法就根本算不上什么方法。如果你在一个认为存在引用的左值环境里做解引用的动作，那么引用就会自动出现。这样做是非常有用的，并且它也是你希望的。这个论题我们在本章稍后探讨，那时候我们将讨论如何把我们到此为止创建的引用给解引用掉。

8.3 使用硬引用

就象我们有无数的方法创建引用一样，我们也有好几种方法使用引用（或者称之为解引用）。使用过程中只有一个最高级的原则：**Perl** 不会做任何隐含的引用或者解引用动作。（注：我们已经承认这句话是撒了一个小谎。我们不想再说了。）如果一个标量挂在了一个引用上，那么它总是表现出简单标量的行为。它不会突然就成为一个数组或者散列或是子过程，你必须明确地告诉它进行转变，方法就是对它解引用。

8.3.1 把一个变量当作变量名使用

如果你看到一个标量，比如 `$foo`，你应该把它看成“`foo` 的标量值。”也就是说，在符号表里有一条 `foo` 记录，而趣味字符 `$` 是一个查看其内部的标量值的方法。如果在里面的的是一个引用，那么你可以通过在前面再增加一个趣味字符来查看引用的内容（解引用）。或者用其他方法查看它，你可以把 `$foo` 里的文本字符串 `foo` 替换成一个指向实际引用物的标量变量。这样做对任何变量类型都是正确的，因此不仅仅 `$$foo` 是指 `$foo` 指向的标量值，`@$bar` 是 `$bar` 指向的数组值，`%%$glarch` 是 `$glarch` 指向的散列数值，等等。结果是你可以在任何简单标量前面放上一个额外的趣味字符将它解引用：

```
$foo      = "three humps";
$scalarref = \ $foo;      # $scalarref 现在是一个指向 $foo 的引用
$camel_model = $$scalarref; # $camel_model 现在是"three humps"
```

下面是其他的一些解引用方法：

```
$bar = $$scalarref;

push(@$arrayref, $filename);
$$arrayref[0] = "January";      # 设置 @$arrayref 的第一个元素
@$arrayref[4..6]=qw/May June July/; # 设置若干个 @$arrayref 的元素

%$hashref = (KEY => "RING", BIRD => "SING"); # 初始化整个散列
$$hashref{KEY} = "VALUE";                # 设置一个键字/数值对
@$hashref{"KEY1", "KEY2"} = {"VAL1", "VAL2"}; # 再设置两对

&$coderef(1,2,3);

print $hdlref "output\n";
```

这种类型的解引用只能使用一个简单的标量变量（没有脚标的那种）。也就是说，解引用在任何数组或者散列查找之前发生（或者说是比数组和散列查找绑定得更紧）。还是让我们用一些花括弧来把我

们的意思表示得明确一些：一个象 `$$arrayref[0]` 这样的表达式等于 `${$arrayref}[0]` 并且意思是数组的第一个元素由 `$arrayref` 指向。类似的，`$$hashref{KEY}` 和 `${$hashref}{KEY}` 一样，并且和 `$$hashref{KEY}` 没什么关系，后者将对一个叫做 `%hashref` 的散列里的记录进行解引用的操作。你在意识到这一点之前可能会非常悲惨。

你可以实现多层引用和解引用，方法是连接合适的趣味字符。下面的程序打印 "howdy"：

```
$refrefref = \\\"howdy";
print $$$refrefref;
```

你可以认为美元符号是从右向左操作的。但是整个链条的开头必须是一个简单的，没有脚标的标量变量。不过，还有一种方法变得更神奇，这个方法我们前面已经偷偷用过了，我们会在下一节解释这种方法。

8.3.2 把一个 BLOCK 块当作变量名用

你不仅可以对一个简单的变量名字进行解引用，而且你还可以对一个 BLOCK 的内容进行解引用。在任何你可以放一个字母数字标识符当变量或者子过程名字一部分的地方，你都可以用一个返回指向正确类型的 BLOCK 代替该标识符。换句话说，早先的例子都可以用下面这样的方法明确化：

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
@{$arrayref}[4..6] = qw/May June July/;
${$hashref}{"KEY"} = "VALUE";
@{$hashref}{"KEY", "KEY2"} = ("VAL1", "VAL2");
&{$coderef}(1,2,3);
```

更不用说：

```
$refrefref = \\\"howdy";
print ${${$refrefref}};
```

当然，在这么简单的情况下使用花括弧的确非常愚蠢，但是 BLOCK 可以包含任意地表达式。特别是，它可以包含带脚标的表达式。

在下面的例子里，我们假设 `$dispatch{$index}` 包含一个指向某个子过程的引用（有时候我们称之为 "coderef"）。这个例子带着三个参数调用该子过程：

```
&{ $dispatch{$index} } (1, 2, 3);
```

在这里，BLOCK 是必要的。没有这个外层的花括弧对，Perl 将把 `$dispatch` 当作 coderef 而不是 `$dispatch{$index}`。

8.3.3 使用箭头操作符

对于指向数组，散列，或者子过程的引用，第三种解引用的方法涉及到使用 `->` 中缀操作符。这样做就形成了一种语法糖，这样就让我们可以更容易访问独立的数组或者散列元素，或者间接地调用一个子过程。

解引用的类型是由右操作数决定的，也就是，由直接跟在箭头后面的东西决定。如果箭头后面的东西是一个方括弧或者花括弧，那么左操作数就分别当作一个指向一个数组或者散列的引用，由右边的操作数做下标定位。如果箭头后面的东西是一个左圆括弧，那么左操作数就当作一个指向一个子过程的引用看待，然后用你在圆括弧右边提供的参数进行调用。

下面的东西每三行都是一样的，分别对应我们已经介绍过的三种表示法。（我们插入了一些空白，以便将等效的元素对齐。）

```
$ $arrayref [2] = "Dorian";      #1
${ $arrayref }[2] = "Dorian";    #2
$arrayref->[2] = "Dorian";        #3

$ $hashref {KEY} = "F#major";     #1
${ $hashref }{KEY} = "F#major";   #2
$hashref->{KEY} = "F#major";       #3

& $coderef (Presto => 192);        #1
&{ $coderef }(Presto => 192);       #2
$coderef->(Presto => 192);           #3
```

你可以注意到，在每个三行组里，第三种表示法的趣味字符都不见了。这个趣味字符是由 **Perl** 猜测的，这就是为什么你不能用它对整个数组，整个散列，或者是它们的某个片段进行解引用。不过，只要你坚持使用标量数值，那么你就可以在 `->` 左边使用任意表达式，包括另外一个解引用，因为多个箭头操作符是从左向右关联的：

```
print $array[3]->{"English"}->[0];
```

你可以从这个表达式里推论出 `@array` 的第四个元素是一个散列引用，并且该散列里的 `"English"` 记录的数值是一个数组的引用。

请注意 `$array[3]` 和 `$array->[3]` 是不一样的。第一个东西讲的是 `@array` 里的第四个元素，而第二个东西讲的是一个保存在 `$array` 里的数组（可能是匿名的数组）引用的第四个元素。

假设现在 `$array[3]` 是未定义。那么下面的语句仍然合法：

```
$array[3]->{"English"}->[0] = "January";
```

这个东西就是我们在前面提到过的引用突然出现的例子，这时候，当引用用做左值的时候是自动激活的（也就是说，当给它赋予数值的时候）。如果 `$array[3]` 是未定义，那么它会被自动定义成一个散列引用，这样我们就可以在它里面给 `$array[3]->{"English"}` 设置一个数值。一旦这个动作完成，那么 `$array[3]->{"English"}` 自动定义成一个数组引用，这样我们就可以给那个数组的第一个元素赋一些东西。请注意右值稍微有些不同：`print $array[3]->{"English"}->[0]` 只定义了 `$array[3]` 和 `$array[3]->{"English"}`，没有定义 `$array[3]->{"English"}->[0]`，因为最后一个元素不是左值。（你可以认为前面两个在右值环境里有定义是一只臭虫。我们将来可能除掉这只虫子。）

在方括弧或花括弧之间，或者在一个闭方括弧或花括弧与圆括弧之间的箭头是可选的。后者表示间接的函数调用。因此你可以把前面的代码缩减成：

```
$dispatch{$index}(1, 2, 3);
$array[3>{"English"}[0] = "January";
```

在普通的数组的情况下，这些东西给你一个多维的数组，就好象 **C** 的数组：

```
$answer[$x][$y][$z] += 42;
```

当然，并不完全象 **C** 的数组。其中之一就是 **C** 的数组不会按照需要增长，而 **Perl** 的却会。而且，一些在两种语言里相似的构造是用不同的方法分析的。在 **Perl**里，下面的两个语句做的事情是一样的：

```
$listref->[2][2] = "hello";      # 相当干净
```

```
$$listref[2][2] = "hello";      # 有点混乱
```

上面第二句话可能会让 C 程序员觉得诧异，因为 C 程序员习惯于使用 `*a[i]` 表示“a 的第 i 个元素所指向的内容”。但是在 Perl 里，五个元素 (`$ @ * % &`) 实际上比花括弧或者方括弧绑定得更紧密。（注：但不是因为操作符优先级。在 Perl 里的趣味字符不是操作符。Perl 的语法只是简单地禁止任何比一个简单变量或者块更复杂的东西跟在趣味字符后面，个中缘由也是有很多有趣的原因的。）因此，被当作指向一个数组引用的是 `$$listref` 而不是 `$listref[2]`。如果你想要 C 的行为，你要么是写成 `${$listref[2]}` 以强迫 `$listref[2]` 先于前面的 `$` 解引用操作符计算，要么你就要使用 `->` 表示法：

```
$listref[2]->{$greeting} = "hello";
```

8.3.4 使用对象方法

如果一个引用碰巧是一个指向一个对象的引用，那么定义该对象的类可能提供了访问该对象内部的方法，并且如果你只是使用这些类，那么通常应该坚持使用那些方法（与实现这些方法相对）。换句话说就是要友善，并且不要把一个对象当作一个普通引用看待，虽然在你必须这么做的时候 Perl 也允许你这么看。我们不想在这个问题上搞极权。但是我们的确希望有一些礼貌。

有了这种礼貌，你就在对象和数据结构之间获得了正交。在你需要的时候，任何数据结构都可以认为是一个对象。或者是在你不需要的时候都认为不是对象。

8.3.5 伪散列

一个伪散列是一个指向数组的任意引用，它的第一个元素是一个指向散列的引用。你可以把伪散列引用当作一个数组引用（如你所料），也可以把它当作一个散列引用（出乎你的意料）。下面是一个伪散列的例子。

```
$john = [ {age => 1, eyes => 2, weight => 3}, 47, "brown", 186 ];
```

在 `$john->[0]` 下面的散列定义了随后的数组元素（47, "brown", 186）的名字（"age", "eyes", "weight"）。现在你可以用散列和数组的表示法来访问一个元素：

```
$john->{weight}      # 把 $john 当作一个 hashref 对待
$john->[3]            # 把 $john 当作一个 arrayref 对待
```

伪散列的魔术并不那么神奇；它只知道一个“技巧”：如何把一个散列解引用转变成一个数组解引用。在向伪散列里增加其他元素的时候，在你使用散列表示法之前，必须明确告诉下层的散列那些元素将放在哪里：

```
$john->[0]{height} = 4;      # 高度是元素 4 的数值
$john->{height} = "tall";    # 或者 $john->[4] = "tall"
```

如果你试图从一个伪散列中删除一个键字，那么 Perl 将抛出一个例外，尽管你总是可以从映射散列中删除键字。如果你试图访问一个不存在的键字，那么 Perl 也会抛出一个例外，这里“存在”的意思是在映射散列里出现：

```
delete $john->[9]{height};   # 只从下层散列中删除
$john->{height};              # 现在抛出一个例外
$john->[4];                   # 仍然打印 "tall"
```

除非你很清楚自己在干什么，否则不要把数组分成片段。如果数组元素的位置移动了，那么映射散列仍然指向原来的元素位置，除非你同时也明确改变它们。伪散列的道行并不深。

要避免不一致性，你可以使用 `use fields` 用法提供的 `fields::phash` 函数创建伪散列：

```
use fields;
$ph = fields::phash(age => 47, eyes => "brown", weight => 186);
print $ph->(age);
```

有两个方法可以检查一个键字在伪散列里面是否存在。第一个方法是使用 **exists**，它检查给出的字段是否已经设置过了。它用这种办法来对应一个真正的散列的行为。比如：

```
use fields;
$ph = fields::phash([qw(age eyes brown)], [47]);
$ph->{eyes} = undef;

print exists $ph->{age};      # 对, 'age' 在声明中就设置了
print exists $ph->{weight};   # 错, 'weight' 还没有用呢
print exists $ph->{eyes};     # 对, 你的 'eyes' 已经修改过了
```

第二种方法是在第一个数组元素里放着的影射散列上使用 **exists**。这样就检查了给出的键字对该伪散列是否是一个有效的字段：

```
print exists $ph->[0]{age};    # 对, 'age' 是一个有效的字段
print exists $ph->[0]{name};   # 错, 不能使用 'name'
```

和真正的散列里发生的事情有些不同，在伪散列元素上调用 **delete** 的时候只删除对应该键字的数组值，而不是映射散列的真正的键字。要删除该键字，你必须明确地从映射散列中删除之。一旦你删除了该键字，那么你就不再能用该名字作为伪散列的脚标：

```
print delete $ph->{age};      # 删除并返回 $ph->[1], 47
print exists $ph->{age};      # 现在是错的
print exists $ph->[0]{age};    # 对, 'age' 键字仍然可用
print delete $ph->[0]{age};    # 现在 'age' 键字没了
print $ph->{age};             # 运行时例外
```

你可能会想知道是因为什么原因促使人们想出这种伪装在散列的外衣下面的数组的。数组的查找速度比较快并且存储效率也高些，而散列提供了对你的数据命名的方便（而不是编号）；伪散列提供了两种数据结构的优点。但是这些巨大的优点只有在你开始考虑 Perl 的编译阶段的时候才会出现。在一两个用法的帮助下，编译器可以核实对有效的数据域的访问，这样你就可以在程序开始运行之前发现不存在的脚标（可以查找拼写错误）。

伪散列的速度，效率，和编译时访问检查（你甚至可以把这个特性看作一种安全性）的属性令它成为创建高效健壮的方法的非常便利的工具。参阅第十二章里的 **use fields** 以及第三十一章，实用模块里的讨论。

伪散列是一种比较新的并且相对试验性的特性；因此，其下的实现在将来很有可能被修改。要保护自己免于受到这样的修改的影响，你应该总是使用 **fields** 里面有文档记录的 **phash** 和 **new** 函数。

8.3.6 硬引用可以用的其他技巧

我们前面提到过，反斜杠操作符通常用于一个引用中生成一个引用，但是并非必须如此。如果和一系列引用物一起使用，那么它生成一系列对应的引用。下面的例子的第二行和第一行做的事情是一样的，因为反斜杠是自动在整个列表中分布的。

```
@reflist = (\$s, \@a, \%h, \%f);      # 一系列四个元素的表
@reflist = (\$s, @a, %h, %f);          # 同样的东西
```

如果一个圆括弧列表只包含一个数组或者散列，那么所有它的数值都被代换，并且返回每个引用：

```
@reflist = \(@x);           # 代换数组，然后获得引用
@reflist = map (\$_) @x;     # 一样的东西
```

如果你在散列上试验这些代码，那么结果将包含指向数值的引用（如你所料），而且还包含那些键字的拷贝的引用（这可是你始料未及的）。

因为数组和散列片段实际上都是列表，因此你可以用反斜杠逃逸两者中的任意一个获取一个引用的列表。下面三行中的任何一个实际上做的都是完全一样的事情：

```
@envrefs = \@ENV{'HOME', 'TERM'};      # 反斜杠处理一个片段
@envrefs = \($ENV{HOME}, $ENV{TERM} );  # 反斜杠处理一个列表
@envrefs = ( \ $ENV{HOME}, \ $ENV{TERM} ); # 一个两个引用的列表
```

因为函数可以返回列表，所以你可以给它们加上反斜杠。如果你有超过一个的函数需要调用，那么首先把每个函数的返回值代换到一个大的列表中，然后在给整个列表加上反斜杠：

```
@reflist = \fx();
@reflist = map { \$_ } fx();           # 一样的东西

@reflist = \ ( fx(), fy(), fz() );
@reflist = ( \fx(), \fy(), fz() );    # 一样的东西
@reflist = map { \$_ } fx(), fy(), fz(); # 一样的东西
```

反斜杠操作符总是给它的操作数提供一个列表环境，因此那些函数都是在列表环境中调用。如果反斜杠本身是处于标量环境，那么你最终会得到一个指向该函数返回的列表中的最后一个数值的引用：

```
@reflist = \localtime();              # 引用九个时间元素中的每一个
@lastref = \localtime();              # 引用的是它是否为夏时制
```

从这个方面来看，反斜杠的行为类似命名的 Perl 列表操作符，比如 **print**、**reverse** 和 **sort**，它们总是在它们的右边提供一个列表环境，而不管它们左边是什么东西。和命名的列表操作符一样，使用明确的 **scalar** 强迫跟在后面的进入标量环境：

```
$dateref = \scalar localtime();       # "\"Thu Apr 19 22:02:18 2001"
```

你可以使用 **ref** 操作符来判断一个引用指向的是什么东西。把 **ref** 想象成一个“**typeof**”（类型为）操作符，如果它的参数是一个引用那么返回真，否则返回假。返回的数值取决于所引用的东西的类型。内建的类型包括 **SCALAR**、**ARRAY**、**HASH**、**CODE**、**GLOB**、**REF**、**LVALUE**、**IO**、**IO::Handle**，和 **Regexp**。在下面，我们用它检查子过程参数：

```
sub sum {
    my $arrayref = shift;
    warn "Not an array reference" if ref($arrayref) ne "ARRAY";
    return eval join("+", @$arrayref);
}
```

如果你在一个字符串环境中使用硬引用，那么它将被转换成一个包含类型和地址的字符串：**SCALAR (0x12fcde)**。（反向的转换是不能实现的，因为在字符串化过程中，引用计数信息将被丢失——而且让程序可以访问一个由一个随机字符串命名的地址也太危险了。）

你可以用 **bless** 操作符把一个引用和一个包函数关联起来作为一个对象类。在你做这些的时候，**ref** 返回类名字而不是内部的类型。在字符串环境里使用的对象引用返回带着内部和外部类型的字符串，以及在内存中的地址：**MyType=HASH(0x21cda)** 或者 **IO::Handle=IO(0x186904)**。参阅第十二章获取有关对象的更多的细节。

因为你对某些东西解引用的方法总是表示着你在寻找哪种引用物，一个类型团可以用与引用一样的方

法来使用，尽管一个类型团包含各种类型的多个引用。因此 `${*main::foo}` 和 `${\main::foo}` 访问的都是同一个标量变量，不过后者更高效一些。

下面是一个把子过程调用的返回值代换成一个字串的技巧：

```
print "My sub returned @{{ mysub(1, 2, 3) }} that time.\n";
```

它的运转过程如下。在编译时，当编译器看到双引号字符串里的 `@{...}` 的时候，它就会被当作一个返回一个引用的块分析。在块里面，方括弧创建一个指向一个匿名数组的引用，该数组来自方括弧里面的东西。因此在运行时，`mysub(1,2,3)` 会在列表环境中调用，然后其结果装载到一个匿名数组里，然后在块里面返回一个指向该匿名数组的引用。然后该数组引用马上就被周围的 `@{...}` 解引用，而其数组的值就被代换到双引号字符串中，就好象一个普通的数组做的那样。这样的强词夺理式的解释也适用于任意表达式，比如：

```
print "We need @{{ $n + 5 }} widgets!\n";
```

不过要小心的是：方括弧给它们的表达式提供了一个列表环境。在本例中它并不在意是否为列表环境，不过前面对 `mysub` 的调用可能会介意。如果列表环境有影响的时候，你可以使用一个明确的 `scalar` 以强迫环境为标量：

```
print "Mysub return @{{ scalar mysub(1,2,3) }} now.\n";
```

8.3.7 闭合（闭包）

我们早些时候谈到过用一个没有名字的 `sub {}` 创建匿名子过程。你可以把那些子过程看作是在运行时定义，这就意味着它们有一个生成的时间和一个定义的地点。在创建子过程的时候，有些变量可能在范围里，而调用子过程的时候，可能有不同的变量在范围里。

先让我们暂时忘掉子过程，先看看一个指向一个词法变量的引用：

```
{
    my $critter = "camel";
    $critterref = \ $critter;
}
```

`$critterref` 的数值仍将是“camel”，即使在离开闭合的花括弧之后 `$critter` 消失了也如此。但是 `$critterref` 也可以指向一个指向了 `$critter` 的子过程：

```
{
    my $critter = "camel";
    $critterref = sub { return $critter };
}
```

这是一个闭合（闭包），这个词是来自 LISP 和 Scheme 那些机能性（functional）编程世界的。

（注：在这样的语言环境里，“functional”（机能性）应该看作是“dysfunctional”（机能紊乱）的反义词）。这就意味着如果你某一时刻在特定的词法范围定义了一个匿名函数，那么它就假装自己是在那个范围里运行的，即使后面它又从该范围之外调用。（一个力求正统的人会说你用不着使用“假装”这个词——它实际上就是运行在该范围里。）

换句话说，Perl 保证你每次都获得同一套词法范围变量的拷贝，即使该词法变量的其他实例在该闭合的实例之前或者自该闭合存在开始又创建其他实例也如此。这样就给你一个方法让你可以在定义子过程的时候设置子过程里的数值，而不仅仅是在调用它们的时候。

你还可以把闭合看作是一个不用 `eval` 书写子过程模板的方法。这时候词法变量用做填充模板的参数，作用主要是设置很少的一些代码用于稍后运行。在基于事件的编程里面，这种做法常常叫做回调

(**callback**)，比如你把一些代码和一次键盘敲击，鼠标点击，窗口露出等等关联起来。如果当作回调使用，闭合做的就是你所预期的，即使你不知道机能性编程的第一件事情也无妨。（请注意这些与闭合相关的事情只适用于 **my** 变量。全局量还是和往常一样运转，因为它们不是按照词法变量的方式创建和删除的。）

闭合的另外一个用途就是函数生成器，也就是说，创建和返回全新函数的函数。下面是一个用闭合实现的函数生成器的例子：

```
sub make_saying {
    my $salute = shift;
    my $newfunc = sub {
        my $target = shift;
        print "$salute, $target!\n";
    };
    return $newfunc;      # 返回一个闭合
}

$f = make_saying("Howdy");      # 创建一个闭合
$g = make_saying("Greetings");  # 创建另外一个闭合

# 到时...

$f->("world");
$g->("earthlings");
```

它打印出：

```
Howdy, world!
Greetings earthlings!
```

特别要注意 **\$salute** 是如何继续指向实际传递到 **make_saying** 里的数值的，尽管到该匿名子过程运行的时候 **my \$salute** 已经超出了范围。这就是用闭合来干的事情。因为 **\$f** 和 **\$g** 保存着指向函数的引用，当调用它们的时候，这些函数仍然需要访问独立的 **\$salute** 版本，因此那些变量版本自动附着在四周。如果你现在覆盖 **\$f**，那么它的版本的 **\$salute** 将自动消失。（Perl 只是在你不再查看的时候才做清理。）

Perl 并不给对象方法（在第十二章描述）提供引用，但是你可以使用闭合获取类似的效果。假设你需要这么一个引用：它不仅仅指向他代表的方法的子过程，而且它在调用的时候，还会在特定的对象上调用该方法。你可以很方便地把对象和方法都看成封装在闭合中的词法变量：

```
sub get_method {
    my ($self, $methodname) = @_;
    my $methref = sub {
        # 下面的 @_ 和上面的那个不一样！
        return $self->$methodname(@_);
    };
    return $methref;
}

my $dog = new Doggie::;
Name => "Lucky",
Legs => 3,
Tail => "clipped";

our $wagger = get_method_ref($dog, 'wag');
```

```
$swagger->("tail");          # 调用 $dog->wag('tail').
```

现在，你不仅可以让你 **Lucky** 摇动尾巴上的任何东西，就算词法 **\$dog** 变量已经跑出了范围并且现在你看不到 **Lucky** 了，那么全局的 **\$swagger** 变量仍然会让它摇尾巴——不管它在哪里。

8.3.7.1 用闭合做函数模板

拿闭合做函数模板可以让你生成许多动作类似的函数。比如你需要一套函数来生成各种不同颜色的 HTML 字体变化：

```
print "Be", red("careful"), "with that ", green("light"), "!!!";
```

red 和 **green** 函数会非常类似。我们已经习惯给我们的函数名字，但是闭合没有名字，因为它们只是带倾向性的匿名子过程。为了绕开这个问题，我们将使用给我们的匿名子过程命名的非常技巧性的方法。你可以把一个 **coderef** 绑定到一个现存的名字上，方法是把它赋予一个类型团，该类型团的名字就是你想要的函数。（参阅第十章，包，里的“符号表”一节。在本例中，我们将把它绑定到两个不同的名字上，一个是大写，一个是小写。

```
@colors = qw(red blue green yellow orange purple wiolet);
for my $name (@colors) {
    no strict 'refs';          # 允许符号引用
    *$name = *(uc $name) = sub { "<FONT COLOR='$name'7gt;@_</FONT>" };
}
```

现在你可以调用名字叫 **red**, **RED**, **blue**, **BLUE**, 等等的函数，并且就会调用合适的函数。这个方法减少了编译时间并且节约了内存，并且还减少了错误的发生，因为语法检查是在编译时进行的。在匿名子过程里任意的变量都必须是词法范围的，这样才能创建闭合。因此上面的例子中使用了 **my**。

这个例子是极少数给闭合原型有意义的地方。如果你想在这些函数的参数上强制标量环境（在我们的例子上可能不是一个好主意），你可以用下面的方法写：

```
*$name = sub ($) { "$_[0]" };
```

这么做几乎已经足够好了。不过，因为原型检查是发生在编译时间，上面的运行时赋值发生得太晚了，因而没有什么价值。你应该把整个赋值循环放到一个 **BEGIN** 块里，强迫它在编译时发生。（更好的方法是你把它放到一个模块里，这样你就可以在编译时 **use** 它了。）这样在编译剩下的时间里，原型就是可见的了。

8.3.7.2 嵌套的子过程

如果你熟悉在子过程里嵌套使用其他子过程（从其他编程语言中学来），每个子过程都有自己的私有变量，你可能不得不稍微地按照 **Perl** 的习惯来处理它们。命名的子过程并不合适做嵌套，但是匿名子过程却可以（注：更准确地说是全局命名的子过程不能嵌套。糟糕的是，全局命名子过程就是我们所拥有的唯一一种命名子过程。我们还没有实现词法范围的命名子过程（被称为 **my subs**），但是到我们实现它们的时候，它们应该可以正确嵌套。）不管怎样，我们都可以用闭合模拟嵌套的，词法范围的子过程。下面是一个例子：

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19};
    return $x + inner();
}
```

因为闭合的临时赋值，现在 **inner** 只能从 **outer** 里面调用。但只要你是在 **outer** 里调用它的，那么它就能从 **outer** 的范围里对词法变量 **\$x** 的正常访问。

这么做对于创建一个相对另外一个函数是局部函数的时候有一些有趣的效果，这些效果不是 Perl 正常时支持的。因为 **local** 是动态范围，并且函数名字对于它们的包来说是全局的，那么任何其他 **outer** 调用的函数也可以调用 **inner** 的临时版本。要避免这些，你应该需要额外的间接的层次：

```
sub outer {
    my $x = $_[0] + 35;
    my $inner = sub {return $x * 19 };
    return $x + $inner->();
}
```

8.4 符号引用

如果你试图给一个不是硬引用的数值进行解引用会发生什么事情？那么该值就会被当作一个符号引用。也就是说，该引用被解释成一个代表某个全局量的名字的字串。

下面是其运转样例：

```
$name = "bam";
$$name = 1;          # 设置 $bam
$name->[0] = 4;        # 设置 @bam 的第一个元素
$name->{X} = "Y";      # 设置 %bam 的 X 元素为 Y
@$name = ();          # 清除 @bam
keys %$name;          # 生成 %bam 的键字
&$name;              # 调用 &bam
```

这么用的功能是非常强大的，并且有点危险，因为我们有可能原来想用的是（有着最好的安全性）硬引用，但是却不小心用了一个符号引用。为了防止出现这个问题，你可以说：

```
use strict 'refs';
```

然后在闭合块的剩余部分，你就只被允许使用硬引用。更内层的块可以用下面的语句撤消这样的命令：

```
no strict 'refs';
```

另外，我们还必须理解下面两行程序的区别：

```
${identifier};      # 和 $identifier 一样
${"identifier"};    # 也是 $identifier，不过却是一个符号引用。
```

因为第二种形式是引号引起的，所以它被当作一个符号引用，如果这时候 **use strict 'refs'** 起作用的话它将会生成一个错误。就算 **strict 'refs'** 没有起作用，它也只能指向一个包变量。但是第一种形式是和没有花括弧的形式相等的，它甚至可以指向一个词法范围的变量，只要你定义了这么一个。下面的例子显示了这个用法（而且下一节就是讨论这个问题的）。

只有包变量可以通过符号引用访问，因为符号引用总是查找包的符号表。由于词法变量不是在包的符号表里面，因此它们对这种机制是隐形的。比如：

```
our $value = "global";
{
    my $value = "private";
    print "Inside, mine is ${value},";
    print "but ours is ${'value'}.\n";
}
print "Outside, ${value} is again ${'value'}.\n";
```

它打印出：

```
Inside, mine is private, but ours is global.
Outside, global is again global.
```

8.5 花括弧，方括弧和引号

在前面一节里，我们指出了 `${identifier}` 是不被当作符号引用看待的。你可能会想知道这个机制是怎么和保留字相交互的，简单的回答是：它们没有相互交互。尽管 `push` 是一个保留字，下面这两句还是打印 "pop on over"：

```
$push = "pop on ";
print "${push}over";
```

这里的原因是这样的，历史上，**Unix shell** 是用花括弧来隔离变量名字和后继的字母数字文本的（要不然这些字母数字就成了变量名字的一部分。）这也是许多人预料的变量代换的运转方式，因此我们在 **Perl** 令之以同样的方法运转。但是就 **Perl** 而言，这种表示法的使用得到了扩展，它可以用于任意用来生成引用的花括弧，不管它们是否位于引号里面。这就意味着：

```
print ${push} . 'over';
```

或者甚至是下面（因为空格无所谓）：

```
print ${ push } . 'over';
```

都打印出 "pop on over"，尽管花括弧在双引号外边。同样的规则适用于任意给散列做脚标的标识符。因此，我们可以不用写下面这样的句子：

```
$hash{ "aaa" }{ "bbb" }{ "ccc" }
```

你可以只写：

```
$hash{ aaa }{ bbb }{ ccc }
```

或者

```
$hash{aaa}{bbb}{ccc}
```

而不用担心这些脚标是否为保留字。因此：

```
$hash{ shift }
```

被代换成 `$hash{"shift"}`。你可以迫使代换当作保留字来用，方法是增加任何可以令这个名字不仅是一个标识符的东西：

```
$hash{ shift() }
$hash{ +shift }
$hash{ shift @_ }
```

8.5.1 引用不能当作散列键字用

散列键字在内部都存储成字符串。（注：它们在外边也存储成字符串，比如在你把它们放到 **DBM** 文件中的时候。实际上，**DBM** 文件要求它们的键字（和数值）是字符串。）如果你试图把一个引用当作一个散列的键字存储，那么该键字值将被转换成一个字符串：

```
$x{ \ $a } = $a;
($key, $value) = each %x;
```

```
print $$key;          # 错误
```

我们前面已经说过你不能把一个字串转换回硬引用。因此如果你试图解引用 `$key`，（它里面只保存着一个字串），那么它不会返回一个硬引用，而是一个符号引用——并且因为你可能没有叫 **SCALAR (0x1fc0e)** 的变量，所以你就无法实现你的目的。你可能要做一些更象：

```
$r = \@a;
${ $r } = $r;
```

这样的东西。这样你至少能使用散列值，这个值是一个硬引用，但你不能用键字，它不是硬引用。

尽管你无法把一个引用存储为键字，但是如果你拿一个硬引用在一个字串的环境中使用，（象我们前面的例子）那么 **Perl** 保证它生成一个唯一的字串，因为该引用的地址被当作字串的一部分包含。这样你实际上就可以把引用当作一个唯一的键字使用。只是你后面就没有办法对它解引用了。

有一种特殊类型的散列，在这种散列里，你可以拿引用当作键字。通过与 **Perl** 捆绑在一起的 **Tie::RefHash** 模块的神奇（这个词是技术术语，如果你翻翻 **Perl** 源程序目录里的 **mg.c** 文件就会明白。），你就可以做我们刚才还说不能做的事情：

```
use Tie::RefHandle;
tie my %h, 'Tie::RefHash';
%h = (
    ["this", "here"]    => "at home",
    ["that", "there"]   => "elsewhere",
);

while ( my($keyref, $value) = each %h ) {
    print "@$keyref is $value\n";
}
```

实际上，通过将不同的实现与内建类型捆绑，你可以把标量，散列，和数组制作成可以拥有我们上面说过不行的行为。你看，这些作者就那么傻...

有关捆绑的更多细节，参阅第十四章，捆绑变量。

8.5.2 垃圾收集，循环引用和弱引用

高级语言通常都允许程序员不用担心在用过内存之后的释放问题。这种自动化收割处理就叫做垃圾收集。就大多数情况而言，**Perl** 使用一种快速并且简单的以引用为基础的垃圾收集器。

如果一个块退出了，那么它的局部范围的变量通常都释放掉，但是你也可能把你的垃圾藏起来，因此 **Perl** 的垃圾收集器就找不到它们了。一个严重的问题是如果一片不可访问的内存的引用记数为零的话，那么它将得不到释放。因此，循环引用是一个非常糟糕的主意：

```
{          # 令 $a 和 $b 指向对方
    my ($a, $b);
    $a = \$b;
    $b = \$a;
}
```

或者更简单的就是：

```
{          # 令 $a 指向它自己
    my $a;
    $a = \$a;
}
```


即使 `$a` 应该在块的结尾释放，但它实际上却没有。在制作递归数据结构的时候，如果你想在程序退出之前回收内存，那么你不得不自己打破（或者弱化，见下文）这样的自引用。（退出后，这些内存将通过一个开销比较大但是完整的标记和扫描垃圾收集。）如果数据结构是一个对象，你可以可以用 `DESTROY` 方法自动打破引用；参阅第十二章的“用 `DESTROY` 方法进行垃圾收集”。

一个类似的情况可能出现在缓冲中——缓冲是存放数据的仓库，用于以更快的速度进行检索采用的方法。在缓冲之外，有指向缓冲内部的引用。问题发生在所有这些引用都被删除的时候，但是缓冲数据和它的内部引用仍然存在。任何引用的存在都会阻止 Perl 回收引用物，即使我们希望缓冲数据在我们不再需要的时候尽快消失也是如此。对于循环引用，我们需要一个不影响引用计数的引用，因而就不会推迟垃圾收集了。

弱引用解决了循环引用和缓冲数据造成的问题，它允许你“弱化”任意引用；也就是说，让它不受引用计数的影响。当最后一个指向对象的未弱化引用被删除之后，该对象就被删除并且所有指向改对象的弱引用都被释放。

要使用这个特性，你需要来自 CPAN 的 [WeakRef²](#) 包，它包含附加的文档。弱引用是试验性特性。不过，有些人就是特别能钻。

Revision: r1.1 - 25 Aug 2005 - 14:06 - [TingYu](#)

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

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](#)