第3章 缓存与记忆术

在 1.8 节看到了一个普通的递归函数有时执行得非常糟糕。解决许多这些性能问题的一个简单和普遍的方法，和非递归环境里出现的一样，就是**缓存**（*caching*）。

考虑一个程序把图像从一种格式转换到另一种格式。特定的，假设输入的是流行的 GIF 格式，输出是将要发送到打印机的。打印机不是那种书桌上的小设备，而是一个拥有巨大打印压力的大公司要在周四下午打印上百万份杂志的那种。

该打印机希望图像是一种特定的 CMYK 格式。CMYK 代表“青 – 紫红 – 黄 – 黑”，即该打印机用来打印杂志的四种专用墨水的颜色。然而，GIF 图像的颜色由 RGB 数值指定，该数值是计算机显示器显示图像时发射的红、绿和蓝光的明暗程度。需要把适合显示器的 RGB 数值转换成适合打印的 CMYK 数值[⊖]。

该转换仅是一些简单的算术运算：

```
### Code Library: RGB-CMYK
sub RGB_to_CMYK {
  my ($r, $g, $b) = @_;
  my ($c, $m, $y) = (255-$r, 255-$g, 255-$b);
  my $k = $c < $m ? ($c < $y ? $c : $y)
                  : ($m < $y ? $m : $y);  # Minimum
  for ($c, $m, $y) { $_ -= $k }
  [$c, $m, $y, $k];
}
```

现在书写程序的剩余部分，这部分程序打开 GIF 文件，每次读取一个像素，对每个像素调用 `RGB_to_CMYK()`，然后用合适的格式写出 CMYK 数值结果。

这里有一个小问题。假设 GIF 图像是 1024 像素宽，768 像素高，总共 786 432 像素。需要执行 786 432 次 `RGB_to_CMYK()` 调用。这看上去还不错，但有个例外：GIF 格式定义的方式决定了没有 GIF 图像能包含超过 256 种不同的颜色。即在 786 432 次调用中，至少有 786 176 次是在浪费时间，因为之前已经做过的相同的计算了。如果能指出如何保存 `RGB_to_CMYK()` 计算结果并在合适的时候取出，就可以赢回一些性能。

在 Perl 里，当考虑要检查是否已经看到过某物时，答案几乎总是使用散列。这次也不例外。如果能使用 RGB 数值作为一个散列键，就可以制作一个散列记录之前是否已经看到过某套 RGB 数值，以及如果看到了，相应的 CMYK 数值是多少。那么程序的逻辑将像这样进行：要转换一套 RGB 数值到一套 CMYK 数值，首先在散列里寻找 RGB 数值。如果没有，就和先前一样计算，把结果存在散列里，并照常返回该结果。如果数值在散列里，那就从散列获得 CMYK 数值并跳过第二次计算而返回该数值。

⊖ “K”代表“黑色”，打印机不使用“B”因为“B”代表“蓝色”。

这个函数分为两部分。第一部分打开文件并每次从中读取一行。它把每行分成 $directive 部分（第一个单词）和 $rest 部分（剩余的部分）。$rest 部分包含了指示的参数，如提供给 LOGFILE

2.1.1 表驱动配置

可以把打开、读取和解析配置文件的代码与实现不同指示的不相关的代码分开。像这样把程序分成两半将可以更加灵活地修改每部分，也把代码与指示分开了。

有 read_config() 的一个替代版本：

```
### Code Library: rdconfig-tabular
sub read_config {
  my ($filename, $actions) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
      $actions->{$directive}->($rest);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

和之前完全一样地打开、读取和解析配置文件。但不再依赖巨大的 if-else 分支了。而这版 read_config 接受一个额外的参数，$actions，它是一个行动表，read_config() 每读取一个配置的指示，它将执行这些行动之一。这个表就称为**分配表**（*dispatch table*），因为它包含了 read_config() 读文件时将要把控制分配到的函数。变量 $rest 的意义和之前相同，但现在作为一个参数传递给合适的行为函数。

一个典型的分配表如下：

```
$dispatch_table =
{ CHDIR      => \&change_dir,
  LOGFILE    => \&open_log_file,
  VERBOSITY  => \&set_verbosity,
  ...        => ...,
};
```

分配表是一个散列，它的键（通常称为**标签**（*tag*））是指示的名称，它的值是**行为**（*action*），指向当识别出合适的指示名时调用的子例程。行为函数期望接受变量 `$rest` 作为一个参数，典型的行为如下：

```
sub change_dir {
  my ($dir) = @_;
  chdir($dir)
    or die "Couldn't chdir to '$dir': $!; aborting";
}

sub open_log_file {
  open STDERR, ">>", $_[0]
    or die "Couldn't open log file '$_[0]': $!; aborting";
}

sub set_verbosity {
  $VERBOSITY = shift
}
```

如果行为很小，就可以直接把它们放到分配表里：

```
$dispatch_table =
  { CHDIR      => sub { my ($dir) = @_;
                        chdir($dir) or
                          die "Couldn't chdir to '$dir': $!; aborting";
                      },
    LOGFILE    => sub { open STDERR, ">>", $_[0] or
                          die "Couldn't open log file '$_[0]': $!; aborting";
                      },
    VERBOSITY  => sub { $VERBOSITY = shift },
    ...        => ...,
};
```

通过转变为一个分配表，消除了巨大的 `if-else` 树，但是到头来还是得到了一个只小了一点的表。这看起来不太成功。但是表带来了几个好处。

2.1.2 分配表的优势

分配表是数据，而不是代码，所以它可以在运行时改变。你可以在你想的任何时候插入新的指示到表里。假设表含有：

```
'DEFINE' => \&define_config_directive,
```

其中，`define_config_directive()` 是：

```
### Code Library: def-conf-dir
sub define_config_directive {
  my $rest = shift;
```

```
    $rest =~ s/^\s+//;
    my ($new_directive, $def_txt) = split /\s+/, $rest, 2;

    if (exists $CONFIG_DIRECTIVE_TABLE{$new_directive}) {
      warn "$new_directive already defined; skipping.\n";
      return;
    }

    my $def = eval "sub { $def_txt }";
    if (not defined $def) {
      warn "Could not compile definition for '$new_directive': $@; skipping.\n";
      return;
    }

    $CONFIG_DIRECTIVE_TABLE{$new_directive} = $def;
  }
```

配置器现在接受这样的指示：

```
DEFINE HOME       chdir('/usr/local/app');
```

define_config_directive()把HOME放入$new_directive并把chdir('/usr/local/app');放入$def_txt。它用eval把定义文本编译成一个子例程，然后把新的子例程装入一个主配置表，%CONFIG_DIRECTIVE_TABLE，以HOME为键。如果事实上%CONFIG_DIRECTIVE_TABLE是一开始就传递给read_config()的分配表，那么read_config()将会看到新的定义，如果在输入文件的下一行看到指示HOME，就将把一个行为关联到HOME。现在一个配置文件如下：

```
DEFINE HOME       chdir('/usr/local/app');
CHDIR /some/directory
...
HOME
```

在···里的指示是在目录/some/directory里被执行。当处理器到达HOME时，它就返回到它的家目录。也可以定义一个相同的但更健壮的版本：

```
DEFINE PUSHDIR  use Cwd; push @dirs, cwd(); chdir($_[0])
DEFINE POPDIR   chdir(pop @dirs)
```

PUSHDIR *dir*用标准Cwd模块提供的cwd()函数指出当前目录的名称。它把当前目录的名称保存在变量@dirs里，然后改变到目录dir。POPDIR撤销最后一个PUSHDIR的影响：

```
PUSHDIR /tmp
A
PUSHDIR /usr/local/app
B
POPDIR
C
POPDIR
```

程序改变到/tmp，执行指示A。然后改变到/usr/local/app并执行指示B。随后的POPDIR使程序回到/tmp，在那里执行指示C，最后第二个POPDIR使程序回到它开始的地方。

为了使DEFINE能改变配置表，将不得不把它存入一个全局变量。如果明确地把表传递给define_config_directive也许更好。为此需要对read_config做一点小小的改变：

```
### Code Library: rdconfig-tablearg
```

```
sub read_config {
  my ($filename, $actions) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
      $actions->{$directive}->($rest, $actions);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

现在 define_config_directive 如下：

```
### Code Library: def-cdir-tablearg
sub define_config_directive {
  my ($rest, $dispatch_table) = @_;
  $rest =~ s/^\s+//;
  my ($new_directive, $def_txt) = split /\s+/, $rest, 2;

  if (exists $dispatch_table->{$new_directive}) {
    warn "$new_directive already defined; skipping.\n";
    return;
  }

  my $def = eval "sub { $def_txt }";
  if (not defined $def) {
    warn "Could not compile definition for '$new_directive': $@; skipping.\n";
    return;
  }

  $dispatch_table->{$new_directive} = $def;
}
```

有了这个改变，就可以增加一个确实有用的配置指示了：

```
DEFINE INCLUDE    read_config(@_);
```

它安装一个新的条目到分配表里，如下：

```
INCLUDE => sub { read_config(@_) }
```

现在，当在配置文件里写：

```
INCLUDE extra.conf
```

主函数 read_config() 将执行行为，传递给它两个参数。第一个参数是从配置文件里得到的 $rest，在这个例子里是文件名 extra.conf。第二个参数还是分配表。将把这两个参数直接传递给 read_config 的递归调用。read_config 将读取 extra.conf，当它结束时就会把控制交给 read_config 的主调用，后者将继续处理主要的配置文件，从刚才离开的地方继续。

为了递归调用能正确工作，read_config() 必须是可重入的。破坏可重入性最简单的方法是使用全局变量，如使用一个全局文件句柄代替词法文件句柄。如果使用了一个全局文件句柄，递归调用 read_config() 将会用同样被主调用使用的句柄打开 extra.conf，这将会关闭主配置文件。当递归调用返回时，read_config() 将无法读取主文件的剩余部分，因为它的文件句

柄已经关闭了。

INCLUDE 这个定义非常简单也非常实用。但它也是巧妙的，也许写 read_config 的时候都没有意识到。“read_config 不需要是可重入的”说起来简单。然而，如果已经写了不可重入的 read_config，那么有用的 INCLUDE 定义将不会起作用。在这里可以学到一个重要的经验：默认使函数是可重入的，因为有时递归调用带来的好处将是一个惊喜。

可重入的函数展现了比不可重入的函数更简单和更可预见的行为。它们更加灵活因为它们可以递归地调用。INCLUDE 例子表明无法总预见到所有的想递归地执行一个函数的理由。更好也更安全的是尽可能使所有函数是可重入的。

分配表与在 read_config() 里硬编码相比较，另一个优势是可以使用同一个 read_config 函数处理两个不相关并且有完全不同指示的文件，只要每次传递一个不同的分配表给 read_config()。可以通过传递一个简装的分配表给 read_config() 而使程序处于“初学者模式”。或者可以重复利用 read_config() 处理另一个带有相同基本语法的文件，只要传递给它一个带有一套不同的指示的表即可。在 2.1.4 节有这样的一个例子。

2.1.3 分配表策略

在 PUSHDIR 与 POPDIR 实现中，行为函数使用了一个全局变量，@dirs，维护压入的目录的栈。这效果不好。可以通过让 read_config() 支持一个**用户形参**（*user parameter*）克服它，使系统更灵活。这是一个参数，由 read_config() 的主调者提供，一字不变地传递给行为：

```
### Code Library: rdconfig-uparam
sub read_config {
  my ($filename, $actions, $user_param) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
     $actions->{$directive}->($rest, $user_param, $actions);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

这消除了全局变量，因为现在可以像这样定义 PUSHDIR 和 POPDIR 了：

```
DEFINE PUSHDIR  use Cwd; push @{$_[1]}, cwd(); chdir($_[0])
DEFINE POPDIR   chdir(pop @{$_[1]})
```

形参 $_[1] 指向被传递给 read_config() 的用户形参参数。如果 read_config() 这样调用：

```
read_config($filename, $dispatch_table, \@dirs);
```

那么 PUSHDIR 和 POPDIR 将用数组 @dirs 作为它们的栈，如果它这样调用：

```
read_config($filename, $dispatch_table, []);
```

那么它们将使用一个崭新的、匿名的数组作为栈。

向一个行为回调传递一个要执行的行为的标签名称常常是有用的。为此，可以改变

read_config():

```
### Code Library: rdconfig-tagarg
sub read_config {
  my ($filename, $actions, $user_param) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
      $actions->{$directive}->($directive, $rest, $actions, $user_param);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

为什么这是有用的？参考为 VERBOSITY 指示定义的行为：

```
VERBOSITY => sub { $VERBOSITY = shift },
```

容易想象会有一些配置指示遵循这个通用模式：

```
VERBOSITY => sub { $VERBOSITY = shift },
TABLESIZE => sub { $TABLESIZE = shift },
PERLPATH  => sub { $PERLPATH = shift },
... etc ...
```

把这三个类似的行为合并成单个做这三件工作的函数。为此，函数需要知道指示的名称以便设置合适的全局变量：

```
VERBOSITY => \&set_var,
TABLESIZE => \&set_var,
PERLPATH  => \&set_var,
... etc ...

sub set_var {
  my ($var, $val) = @_;
  $$var = $val;
}
```

或者，如果你不喜欢一堆松散的全局变量，你可以把配置信息保存到一个散列里，然后传递这个散列的引用作为用户形参：

```
sub set_var {
  my ($var, $val, undef, $config_hash) = @_;
  $config_hash->{$var} = $val;
}
```

在这个例子里，节省的不多，因为行为如此简单。然而可能有几个配置指示需要共享一个更复杂的函数。这里有一个稍微复杂些的例子：

```
sub open_input_file {
  my ($handle, $filename) = @_;
  unless (open $handle, $filename) {
    warn "Couldn't open $handle file '$filename': $!; ignoring.\n";
  }
}
```

这个 open_input_file() 函数可以被许多配置指示分享。例如，假设一个程序有三个输入文件：一个历史文件、一个临时文件和一个模式文件。希望这三个文件的位置都可以在配置文件里配置，这需要在分配表里有三个条目。但是三个条目都可以共享相同的 open_input_file() 函数：

```
...
HISTORY  => \&open_input_file,
TEMPLATE => \&open_input_file,
PATTERN  => \&open_input_file,
...
```

现在假设配置文件认为：

```
HISTORY         /usr/local/app/history
TEMPLATE        /usr/local/app/templates/main.tmpl
PATTERN         /home/bill/app/patterns/default.pat
```

read_config() 将看到第一行并分配给 open_input_file() 函数，传递给它的参数列表是 ('HISTORY','/usr/local/app/history')。open_input_file() 将参数 HISTORY 作为文件句柄名，并把 HISTORY 文件句柄打开到文件 /usr/local/app/history。第二行，read_config() 将再次分配给 open_input_file()，这次传递给它 ('TEMPLATE', '/usr/local/app/templates/main.tmpl')。这次，open_input_file() 将打开 TEMPLATE 句柄而不是 HISTORY 句柄。

2.1.4 默认行为

例子中的 read_config() 函数一遇到无法识别的指示就会崩溃。这种行为是硬编码在其中的。如果分配表自身携带了对一个无法识别的指示要做什么的信息，那会更好。增加这个功能很简单：

```
### Code Library: rdconfig-default
sub read_config {
  my ($filename, $actions, $userparam) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    my $action = $actions->{$directive} || $actions->{_DEFAULT_};
    if ($action) {
      $action->($directive, $rest, $actions, $userparam);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

这里的函数在行为表里寻找指定的指示，如果没有，它就寻找 _DEFAULT_ 行为，仅当分配表里没有指定的默认行为时崩溃。这里有一个典型的 _DEFAULT_ 行为：

```
sub no_such_directive {
  my ($directive) = @_;
  warn "Unrecognized directive $directive at line $.; ignoring.\n";
}
```

由于把指示的名称作为第一个参数传递给行为函数，因此默认的行为知道调用无法识别的指示代表什么。由于 no_such_directive() 函数也得到了传递的整个分配表，因此它可以抽取到真实的指示名称并通过模式匹配指出可能的含义。这里 no_such_directive() 用一个假想的 score_match() 函数判断哪个表条目良好地匹配无法识别的指示：

```
sub no_such_directive {
  my ($bad, $rest, $table) = @_;
  my ($best_match, $best_score);
  for my $good (keys %$table) {
    my $score = score_match($bad, $good);
    if ($score > $best_score) {
      $best_score = $score;
      $best_match = $good;
    }
  }
  warn "Unrecognized directive $bad at line $.;\n";
  warn "\t(perhaps you meant $best_match?)\n";
}
```

现在拥有的系统只含有少量代码，但它是极其灵活的。假设程序还要读取一系列用户 ID 与电子邮件地址，格式如下：

```
fred            fred@example.com
bill            bvoehno@plover.com
warez           warez-admin@plover.com
...             ...
```

可以复用 read_config() 并让它读取和解析这个文件，通过提供合适的分配表：

```
$address_actions =
  { _DEFAULT_ => sub { my ($id, $addr, $act, $aref) = @_;
                       push @$aref, [$id, $addr];
                     },
  };

read_config($ADDRESS_FILE, $address_actions, \@address_array);
```

这里已经给了 read_config() 一个非常小的分配表，它只有一个 _DEFAULT_ 条目。read_config() 对地址文件里的每一行都将调用这个默认的条目一次，传递给它“指示名称”（实际上即用户 ID）与地址（即 $rest 的值）。默认的行为将获得这些信息并增加到 @address_array，程序可以在以后使用它。

2.2 计算器

暂时不讲配置文件的例子了。显然，分配表将在许多类似的情形中有其意义。例如，一个必须执行来自用户的命令的对话程序能使用一个分配表分配用户的命令。接下来介绍一个不同的例子，一个非常简单的计算器。

输入此计算器的是一串以**逆波兰表示法**（*Reverse Polish Notation*，RPN）表示的算术表达式。传统的算术标记法是有歧义的。如果你写下 2+3 · 4，那到底是先做加法还是乘法，并不清楚。因此必须特别约定乘法总是比加法先做，或者必须插入括号消除表达式的歧义，例如，(2+3) · 4。

逆波兰表示法以另一种方式解决问题。不是把操作符放在它们操作的参数的中间，而是放在后面。例如，2+3 写成 2 3 +。(2+3)·4 写成 2 3 + 4 *。2 和 3 后面是 +，所以 2 和 3 相加；* 表示之前的两个表达式相乘，即 2 3 + 和 4。要以 RPN 表示 2+(3·4)，可以写成 2 3 4 * +。+ 应用于之前的两个参数，第一个是 2，第二个是 3 4 *。因为操作符总是在它的参数之后，这样的表达式就称为**后缀式**（*postfix form*）；对照于此，通常的操作符在中间的表达式，就称为**中缀式**（*infix form*）。

计算 RPN 表达式的值很容易。为此，维护一个栈，然后从左往右读取表达式。当看到一个数字时，要把它压入栈。当看到一个操作符时，弹出栈顶部的两个元素，对它们进行操作，然后把结果压回栈。例如，要计算 2 3 + 4 *，需要先压入 2 然后压入 3，然后当看到 + 时就弹出它们并把总和 5 压回栈。然后把 4 压到 5 的上面，然后 * 说明要弹出 4 和 5 并把最后的结果 20 压回。要计算 2 3 4 * + 需要压入 2，然后是 3，然后是 4。*说明要弹出 3 和 4 并把乘积 12 压回，+ 说明要弹出 12 和 2 并压回总和 14，这是最终的答案。

这是一个小的计算器程序，它计算在它的命令行参数中给出的 RPN 表达式：

```
### Code Library: rpn-ifelse
my $result = evaluate($ARGV[0]);
print "Result: $result\n";

sub evaluate {
  my @stack;
  my ($expr) = @_;
  my @tokens = split /\s+/, $expr;
  for my $token (@tokens) {
    if ($token =~ /^\d+$/) { # It's a number
      push @stack, $token;
    } elsif ($token eq '+') {
      push @stack, pop(@stack) + pop(@stack);
    } elsif ($token eq '-') {
      my $s = pop(@stack);
      push @stack, pop(@stack) - $s
    } elsif ($token eq '*') {
      push @stack, pop(@stack) * pop(@stack);
    } elsif ($token eq '/') {
      my $s = pop(@stack);
      push @stack, pop(@stack) / $s
    } else {
      die "Unrecognized token '$token'; aborting";
    }
  }
  return pop(@stack);
}
```

这个函数用空白符把参数分隔成**记号**（*token*），这是最小的、有意义的输入部分。然后函数从左往右每次循环一个记号。如果一个记号匹配 /^\d+$/，那么它是一个数，因此函数把它压入栈。否则，它是一个操作符，因此函数从栈里弹出两个值，操作它们，并把结果压回栈。减法部分的代码里有辅助变量 $s 是因为 5 3 − 应该产生 2，而不是 −2。如果用了：

```
push @stack, pop(@stack) - pop(@stack);
```

那么对于 5 3 −，第一个 pop 弹出 3，第二个 pop 弹出 5，结果会是 −2。同理，类似的代码也出现在除法部分。对于乘法和加法，操作数的次序无关紧要。

当函数读完记号，它就弹出栈顶部的值，这就是最后的结果。这段代码忽略了栈结束时也许有几个值的可能，这意味着，参数包含不止一个表达式。10 2 * 3 4 + 在栈里依次留下了 20 和 7。它也忽略了栈也许变空的可能。例如，2 * 和 2 3 + * 就是无效的表达式，因为其中 * 只有一个参数，而不是两个。在计算这些的时候，函数发现当栈空时它自己还在做操作。在那种情况下，它应当发出错误信号，但是我忽略了错误处理以保持例子的短小精悍。

可以通过把巨大的 if-else 分支替换成分配表，使例子更简洁更灵活：

```
### Code Library: rpn-table
my @stack;
my $actions = {
  '+' => sub { push @stack, pop(@stack) + pop(@stack) },
  '*' => sub { push @stack, pop(@stack) * pop(@stack) },
  '-' => sub { my $s = pop(@stack); push @stack, pop(@stack) - $s },
  '/' => sub { my $s = pop(@stack); push @stack, pop(@stack) / $s },
  'NUMBER' => sub { push @stack, $_[0] },
  '_DEFAULT_' => sub { die "Unrecognized token '$_[0]'; aborting" }
};

my $result = evaluate($ARGV[0], $actions);
print "Result: $result\n";

sub evaluate {
  my ($expr, $actions) = @_;
  my @tokens = split /\s+/, $expr;
  for my $token (@tokens) {
    my $type;
    if ($token =~ /^\d+$/) { # It's a number
      $type = 'NUMBER';
    }

    my $action = $actions->{$type}
              || $actions->{$token}
              || $actions->{_DEFAULT_};
    $action->($token, $type, $actions);
  }
  return pop(@stack);
}
```

主要的驱动，evaluate()，现在更小巧更通用了。它基于记号的“类型”选择一个行为，如果后者有一个行为；否则，行为就基于记号本身的值，如果不存在这样的行为，就使用一个默认的行为。函数 evaluate() 对记号做模式匹配以尝试确定记号的类型，如果记号看起来像一个数，那么选择的类型就是 NUMBER。可以在 %actions 分配表里增加一条以增加一个新的操作符：

```
...
'sqrt' => sub { push @stack, sqrt(pop(@stack)) },
...
```

同样，由于分配表的结构，提供不同的分配表给求值程序就可以得到不同的行为。如果提供如下分配表，求值程序就会把表达式编译成**抽象语法树**（*Abstract Syntax Tree*，AST），而不是把表达式换算成一个数：

```
my $actions = {
  'NUMBER'    => sub { push @stack, $_[0] },
```

```
    '_DEFAULT_' => sub { my $s = pop(@stack);
                         push @stack,
                            [ $_[0], pop(@stack), $s ]
                       },
};
```

编译 2 3 + 4 * 的结果是抽象语法树 ['*', ['+', 2, 3], 4]，也可以用图 2-1 表示。

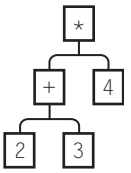


图 2-1 表达式 2 3 + 4 * 的 AST

这是对表达式最有用的内部形式，因为所有的结构直接明了。表达式或者是一个数，或者它有一个操作符和两个操作数，这两个操作数也是表达式。抽象语法树或是一个数，或是一系列的操作符与另两个 AST。一旦有了 AST，很容易写一个函数处理它。例如，这里有个函数把 AST 转换成字符串：

```
### Code Library: AST-to-string
sub AST_to_string {
  my ($tree) = @_;
  if (ref $tree) {
    my ($op, $a1, $a2) = @$tree;
    my ($s1, $s2) = (AST_to_string($a1),
                     AST_to_string($a2));
    "($s1 $op $s2)";
  } else {
    $tree;
  }
}
```

对于图 2-1 的树，函数 AST_to_string() 就会产生字符串 "((2 + 3) * 4)"。函数首先检查树是否是平凡的，如果它不是一个引用，那它必是一个数，字符串的版本就是那个数。否则，字符串有三部分：一个操作符符号，存放在 $op中，以及两个参数，即 AST。函数递归调用自身把两棵参数树转换成字符串 $s1 和 $s2，然后产生一个新的字符串，含有 $s1、$s2，以及它们中间的合适的操作符符号，并在左右两侧加上了括号以避免歧义。已经写了一个系统把后缀表达式转换成中缀表达式，因为可以把原始的后缀表达式赋值给 evaluate() 以产生一个 AST，然后把 AST 给 AST_to_string() 以产生一个中缀表达式。

函数 AST_to_string() 是递归的是因为 AST 的定义是递归的，AST 定义是递归的是因为表达式的结构是递归的。AST_to_string() 的结构直接反映了表达式的结构。

2.2.1 再访 HTML 处理

第 1 章介绍了 walk_html()，一个递归的 HTML 处理器。HTML 处理器得到两个函数参数：$textfunc，一个用来处理未置标签的文本片段的函数，以及 $elementfunc，一个用来处理

HTML 元素的函数。但是“HTML 元素”是笼统的，因为有许多种类的元素，希望函数对每类不同的元素能做不同的事情。

我们已经看到了几种完成此类处理的方法。对用户而言，最直接的方法就是简单地在 $elementfunc 里放个巨大的 if-else 分支。但是已经看到，那样做会有一些缺点。用户可能更喜欢提供一个分配表给 $elementfunc。这样的一个分配表的结构显而易见：表的键将是标签名称，值将是对每类元素执行的行为。用户不是提供单个能知道如何处理所有可能的元素的 $elementfunc，相反，用户将提供一个分配表，为每类元素提供一个行为，即一个普通的 $elementfunc 分配适当的行为。

$elementfunc 可以用几种方法获得分配表。分配表可以硬编码在这个元素函数里：

```
sub elementfunc {
  my $table = { h1        => sub { shift; my $text = join '', @_;
                                   print $text; return $text ;
                                 }
                _DEFAULT_ => sub { shift; my $text = join '', @_;
                                              return $text ;
                                 }
              };
  my ($element) = @_;
  my $tag = $element->{_tag};
  my $action = $table->{$tag} || $table->{_DEFAULT_};
  return $action->(@_);
}
```

或者，可以直接在 walk_html() 里直接支持分配表，那样用户就不用传递单个 $elementfunc，而是直接传递分配表给 walk_html()。在那种情况下。walk_html() 如下：

```
### Code Library: walk-html-disp
sub walk_html {
  my ($html, $textfunc, $elementfunc_table) = @_;
  return $textfunc->($html) unless ref $html; # It's a plain string

  my ($item, @results);
  for $item (@{$html->{_content}}) {
    push @results, walk_html($item, $textfunc, $elementfunc_table);
  }
  my $tag = $html->{_tag};
  my $elementfunc = $elementfunc_table->{$tag}
                 || $elementfunc_table->{_DEFAULT_}
                 || die "No function defined for tag '$tag'";
  return $elementfunc->($html, @results);
}
```

还有一种做法是把 walk_html() 改成传递一个用户形参给 $textfunc 和 $elementfunc。然后用户可以通过用户形参机制把分配表传递给 $elementfunc：

```
### Code Library: walk-html-uparam
sub walk_html {
  my ($html, $textfunc, $elementfunc, $userparam) = @_;
  return $textfunc->($html, $userparam) unless ref $html;
  my ($item, @results);
  for $item (@{$html->{_content}}) {
    push @results, walk_html($item, $textfunc, $elementfunc, $userparam);
```

```
    }
    return $elementfunc->($html, $userparam, @results);
}
```

现在如何设计他们的 $elementfuncs 以适当地处理分配表，由用户决定。

有一点是重要的和巧妙的：传递了和 $elementfunc 一样的用户形参给 $textfunc。如果用户形参是一个标签分配表，它可能对 $textfunc 没有用。那么为什么要传递它呢？因为它可能不是一个标签表，它可能是别的东西。例如，用户可能像这样调用 walk_html()：

```
walk_html($html_text,

          # $textfunc
          sub { my ($text, $aref) = @_;
                push @$aref, $text },

          # $elementfunc does nothing
          sub { },

          # user parameter
          \@text_array
         );
```

现在 walk_html() 将遍历 HTML 树并把所有没有标签的普通文本压入数组 @text_array。用户形参是指向 @textarray 的引用，把它传递给 $textfunc，后者把文本压入指向的数组。$elementfunc 根本不使用用户形参。因为 walk_html() 的作者无法预知用户将需要哪类用户形参，最好把它都传递给 $textfunc 与 $elementfunc，不需要用户形参的函数可以随意忽略它。

第3章 缓存与记忆术

在 1.8 节看到了一个普通的递归函数有时执行得非常糟糕。解决许多这些性能问题的一个简单和普遍的方法，和非递归环境里出现的一样，就是**缓存**（*caching*）。

考虑一个程序把图像从一种格式转换到另一种格式。特定的，假设输入的是流行的 GIF 格式，输出是将要发送到打印机的。打印机不是那种书桌上的小设备，而是一个拥有巨大打印压力的大公司要在周四下午打印上百万份杂志的那种。

该打印机希望图像是一种特定的 CMYK 格式。CMYK 代表“青 – 紫红 – 黄 – 黑”，即该打印机用来打印杂志的四种专用墨水的颜色。然而，GIF 图像的颜色由 RGB 数值指定，该数值是计算机显示器显示图像时发射的红、绿和蓝光的明暗程度。需要把适合显示器的 RGB 数值转换成适合打印的 CMYK 数值[⊖]。

该转换仅是一些简单的算术运算：

```
### Code Library: RGB-CMYK
sub RGB_to_CMYK {
  my ($r, $g, $b) = @_;
  my ($c, $m, $y) = (255-$r, 255-$g, 255-$b);
  my $k = $c < $m ? ($c < $y ? $c : $y)
                  : ($m < $y ? $m : $y);  # Minimum
  for ($c, $m, $y) { $_ -= $k }
  [$c, $m, $y, $k];
}
```

现在书写程序的剩余部分，这部分程序打开 GIF 文件，每次读取一个像素，对每个像素调用 `RGB_to_CMYK()`，然后用合适的格式写出 CMYK 数值结果。

这里有一个小问题。假设 GIF 图像是 1024 像素宽，768 像素高，总共 786 432 像素。需要执行 786 432 次 `RGB_to_CMYK()` 调用。这看上去还不错，但有个例外：GIF 格式定义的方式决定了没有 GIF 图像能包含超过 256 种不同的颜色。即在 786 432 次调用中，至少有 786 176 次是在浪费时间，因为之前已经做过的相同的计算了。如果能指出如何保存 `RGB_to_CMYK()` 计算结果并在合适的时候取出，就可以赢回一些性能。

在 Perl 里，当考虑要检查是否已经看到过某物时，答案几乎总是使用散列。这次也不例外。如果能使用 RGB 数值作为一个散列键，就可以制作一个散列记录之前是否已经看到过某套 RGB 数值，以及如果看到了，相应的 CMYK 数值是多少。那么程序的逻辑将像这样进行：要转换一套 RGB 数值到一套 CMYK 数值，首先在散列里寻找 RGB 数值。如果没有，就和先前一样计算，把结果存在散列里，并照常返回该结果。如果数值在散列里，那就从散列获得 CMYK 数值并跳过第二次计算而返回该数值。

⊖ “K”代表“黑色”，打印机不使用“B”因为“B”代表“蓝色”。

代码如下：

```
### Code Library: RGB-CMYK-caching
my %cache;

sub RGB_to_CMYK {
  my ($r, $g, $b) = @_;
  my $key = join ',', $r, $g, $b;
  return $cache{$key} if exists $cache{$key};
  my ($c, $m, $y) = (255-$r, 255-$g, 255-$b);
  my $k = $c < $m ? ($c < $y ? $c : $y)
                  : ($m < $y ? $m : $y);  # Minimum
  for ($c, $m, $y) { $_ -= $k }
  return $cache{$key} = [$c, $m, $y, $k];
}
```

假设以参数 (128,0,64) 调用 RGB_to_CMYK()。第一次这么做的时候，函数将在 %cache 散列里寻找键 '128,0,64'，散列里什么也没有，所以它将继续，照常执行运算，然后在最后一行，把结果存入 $cache{'128,0,64'}，并返回结果。第二次以相同参数调用函数时，它算出相同的键，并返回 $cache{'128,0,64'} 的值而不再做任何多余的运算。当在缓存里找到了需要的值而无需更多运算时，那就称为一次**缓存命中**（*cache hit*）；当计算了正确的键但是发现还没有值缓存在那个键上时，这就称为一次**缓存脱靶**（*cache miss*）。

当然有一种可能就是额外的程序逻辑和散列搜索会耗尽因避免运算得到的好处。这真正取决于原先的运算所消耗的时间和缓存命中的可能性。当原先的运算消耗时间很长，缓存更可能是赚了。为了确定，对两个版本的函数进行仔细的基准比较测试。但是为了帮助建立对各类权衡的期许的直觉，将简短地介绍理论。

假设实际函数的典型调用需耗时 f。带记忆的版本的平均耗时将决定于两个参数：缓存管理的开销 K 以及对任何特定调用的缓存命中的概率 h。在从未缓存命中过的极端情况下，h 是零；随着缓存命中的可能性增加，h 趋近于 1。

对于一个带记忆的函数，每次调用的平均时间将至少是 K，由于每次调用必须检查缓存，如果缓存脱靶，还得加上一个额外的 f，总计是 $K + (1 - h)f$。函数的不带记忆的版本，当然，总是耗时 f，因此不同之处仅是 $hf - K$。如果 $K < hf$，函数的带记忆的版本将比不带记忆的更快。为加速带记忆的版本，可以增加缓存命中率 h，或减少缓存管理的开销 K。当 f 很大时，就更容易达成 $K < hf$，所以缓存技术对运行时间长的原始函数更可能有效。在最差情况下，没有一次命中，h=0，所以“加速”实际上是减速 $-K$。

3.1 缓存修正递归

在 1.8 节看到递归函数有时耗时太长了，即便对简单的输入，那个 Fibonacci 函数就是这个问题的一个例子：

```
# Compute the number of pairs of rabbits alive in month n
sub fib {
  my $n = shift;
  if ($n < 2) { return $n }
```

```
    fib($n-2) + fib($n-1);
}
```

正如在 1.8 节看到的，这个函数对多数参数都运行缓慢，因为它在重复计算那些已经计算过的结果上浪费时间。例如，`fib(20)` 需要计算 `fib(19)` 和 `fib(18)`，但是 `fib(19)` **也要**计算 `fib(18)`，`fib(17)` 也是一样，每次 `fib(18)` 调用时也都要计算。这是递归函数的一个普遍问题，而它就会被缓存修正。如果为 `fib` 添加缓存技术，那它就不会在第二次需要 `fib(18)` 的时候再从头开始计算它了，`fib` 将简单地取回缓存的第一次计算 `fib(18)` 的结果。不用再担心会计算 `fib(17)` 三次或 `fib(16)` 五次，因为计算只进行一次，然后缓存的结果将于再次需要的时候被快速地取回。

3.2 内联缓存

给一个函数添加缓存的最直接的方式就是给函数一个私有的散列。在这个例子里，可以使用一个数组代替散列，因为 `fib()` 的参数总是一个非负整数。但是一般需要使用一个散列，那么将会看到：

```
### Code Library: fib-cached
# Compute the number of pairs of rabbits alive in month n
{ my %cache;
  sub fib {
    my ($month) = @_;
    unless (exists $cache{$month}) {
      if ($month < 2) { $cache{$month} = $month }
      else {
        $cache{$month} = fib($month-1) + fib($month-2);
      }
    }
    return $cache{$month};
  }
}
```

这里的 `fib` 得到和之前一样的参数。但不是立即进入递归的 Fibonacci 数列计算，它先检查缓存。缓存是一个散列，`%cache`。当函数计算一个 Fibonacci 数 `fib($month)`，它将把这个值存入 `$cache{$month}`。随后对 `fib()` 的调用将检查缓存散列中是否有这个值。这就是 `exists $cache{$month}` 测试的目的。如果缓存元素不存在，即函数此前没有为这个指定的 `$month` 调用过。在 `unless` 块中的代码就是一般的 Fibonacci 计算，包括如果需要的递归调用。然而，一旦函数算出了答案，它不是立即返回它，而是把数值插入缓存散列的合适位置。例如，当 `$month < 2` 是真，`$cache{$month} = 1` 负责在缓存里占位。

在函数的末尾，`return $cache{$month}` 返回缓存了的值，不管这个值是函数刚才插入的或者一开始就在那里的。

有了这些改变，`fib` 函数变快了。在第 1 章看到的过度递归的问题简单地消失了。问题是由结果的重复的再次计算引起的，添加了缓存行为阻止了任何再次计算的出现。当函数尝试再次计算一个已经被计算过的结果时，它会立即从缓存里得到值。

3.2.1 静态变量

为什么 %cache 在 fib 的外面而不是里面，为什么有一个空的花括号包住 %cache 和 fib？如果 %cache 是在 fib 里面声明的，如下：

```
sub fib {
  my %cache;
  ...
}
```

那么缓存不会工作的，因为每次调用 fib 时，就会生成一个新的 %cache 变量，并在 fib 返回时抛弃。通过在任何函数的外面声明 %cache，告诉 Perl 我们只想要一个 %cache 实例，它在程序首次编译时生成并且仅在程序结束时销毁。这就使得 %cache 可以积累数值并在调用 fib 前后保留它。一个类似 %cache 的在所有函数外部声明的变量称为一个**静态变量**（*static variable*），因为它的值保持不变，除非显式改变，也因为 C 语言的一个类似功能是由关键词 static 激活的。

%cache 是由 my 声明的，所以它是词法域的。默认状态，它的作用域将持续到文件的结束。如果在 fib 后面定义了任何函数，它们也可以看到和修改缓存。但这不是我们想要的，我们想要缓存完全是 fib 私有的。把 %cache 和 fib 包在一个隔开的块中能达到这个目的。%cache 的作用域仅延伸到此块的结束，这里面只有 fib 没别的了。

3.3 好主意

既优秀又简单的主意不太多，能到处使用的非常少。缓存技术是其中之一。网页浏览器缓存了从网络取回的文档。当第二次要求同一个文档时，浏览器从本地磁盘或内存取回缓存了的副本，这很快，不用再次下载。域名服务器缓存了它从远端服务器收到的回复。当第二次搜索同样的名字时，本地服务器已经准备好了回复而不必进行可能耗时的网络通信了。当操作系统从磁盘读取数据时，它可能把数据缓存在内存里，以防该数据被再次读取；当 CPU 从内存获取数据时，它把数据缓存在特殊的缓存内存里，它比普通的主存更快。

缓存技术在真实的程序中反复出现。几乎任何含有带缓存技术的函数的程序都可能在性能上获益。然而缓存技术最好的特点是它是**机械的**（*mechanical*）。如果想让一个函数更快，可以重写函数或引入更好的数据结构或更精妙的算法。这可能需要独创性，它总是稀有的。但是添加缓存技术不需要动脑筋，缓存技术转换总是几乎一样的。如下：

```
sub some_function {
  $result = some computation involving @_;
  return $result;
}
```

转换成：

```
{ my %cache;
  sub some_function_with_caching {
    my $key = join ',', @_;
    return $cache{$key} if exists $cache{$key};
    $result = the same computation involving @_;
    return $cache{$key} = $result;
  }
}
```

这个转换对所有函数几乎都完全一样。唯一需要改变的部分是 `join ',', @_` 这行。这行要把函数的参数数组转换成一个字符串，以适合作为散列的键。像这样把任意值转变成字符串，这被称为**序列化**（*serialization*）或 *marshalling*[⊖]。先前的 `join ',', @_`的例子仅对那些参数是数字或不含逗号的字符串的函数有效。后面将更仔细地介绍缓存的键的生成。

3.4 记忆术

给函数添加缓存技术代码不是非常困难的。且已经看到，需要的改变对任何函数几乎一样。那么，为什么不让计算机做这些呢？若告诉 Perl 想要使一个函数具有缓存行为。Perl 应该能自动地执行所需的转换。这样的给函数添加缓存行为的自动的转换就称为**记忆术**（*memoization*），函数则称为**带记忆的**[⊜]（*memoized*）。

标准的 Memoize 模块就是做这个的。如果 Memoize 模块可用，就完全不必重写 fib 代码。可以简单地在程序的顶部添加两行：

```
### Code Library: fib-automemo
use Memoize;
memoize 'fib';
# Compute the number of pairs of rabbits alive in month n
sub fib {
  my ($month) = @_;
  if ($month < 2) { return $month }
  fib($month-2) + fib($month-1);
}
```

fib 现在展现了缓存行为。代码和原先的慢版完全一样，但是函数不再慢了。

3.5 MEMOIZE 模块

本书不是关于 Perl 模块的内部细节的，而是关于一些 Memoize 模块内部使用的并和稍后要做的事情有直接联系的技术，所以现在简短地介绍一下。

Memoize 得到一个函数名（或引用）作为它的参数。它制造一个新的函数，后者维护一个缓存并在其中查找它的参数。如果新的函数在缓存里找到了参数，就返回缓存的值；如果没找到，就调用原始函数，把返回的值保存入缓存，并把它返回给原始的主调者。

制造完这个新的函数，Memoize 就把它装入 Perl 的符号表以代替原始的函数，那样当你认为你在调用原始函数的时候，其实你得到了新的带缓存管理的函数。

如果不去探究真实的 Memoize 模块的内部，即一个 350 行的怪物，那么将看一个小的、削减过的记忆器。要去除的最重要的东西是处理 Perl 符号表的代码部分（手动处理）。取而代之的是，我们将有一个 memoize 函数，它的参数是我们想要使之带记忆的子例程的引用，且它返回一个指向带记忆的版本（即缓存管理函数）的引用：

⊖ data marshalling 如此命名是因为 Edward Waite Marshall 于 1962 年首次研究了它，然后是通用电气公司。

⊜ 术语记忆术（memoization）是由 Donald Michie 于 1968 年创造的。

```
### Code Library: memoize
sub memoize {
  my ($func) = @_;
  my %cache;
  my $stub = sub {
    my $key = join ',', @_;
    $cache{$key} = $func->(@_) unless exists $cache{$key};
    return $cache{$key};
  };
  return $stub;
}
```

要调用它，首先用：

```
$fastfib = memoize(\&fib);
```

现在 `$fastfib` 就是带记忆版的 `fib()`。为在符号表中装入带记忆版的 `fib()` 以代替原始的，要写 `*fib = memoize(\&fib)`。在这个例子里，如果想快速计算 Fibonacci 数，那么安装是必要的。仅产生一个带记忆版的 `fib()` 是不够的，因为在 `fib()` 里面的递归调用是调用称为 `fib()` 的函数，直到对 `*fib` 赋值之前，这仍然是旧的、慢的、不带记忆的版本。

`memoize` 如何工作？把一个指向 `fib` 的引用传递给 `memoize`，然后它制造一个私有的 `%cache` 变量保存缓存的数据。然后产生一个**存根函数**（*stub function*），暂时保存在 `$stub`，后者返回给主调者。这个存根函数就是实际上的带记忆版的 `fib`，`memoize` 的主调者回到一个指向它的引用，即在先前的例子里保存于 `$fastfib` 的。

当执行 `$fastfib`时，实际上得到的是之前 `memoize` 制造的存根函数。这个存根函数把函数的参数用逗号连接在一起组成一个散列键，然后在缓存里看看这个键是否是熟悉的。如果是，存根立即返回缓存的值。

如果在散列中没找到这个散列键，存根函数就通过 `$func->(@_)` 执行原始的函数，得到结果，存入缓存，然后返回（图 3-1）。

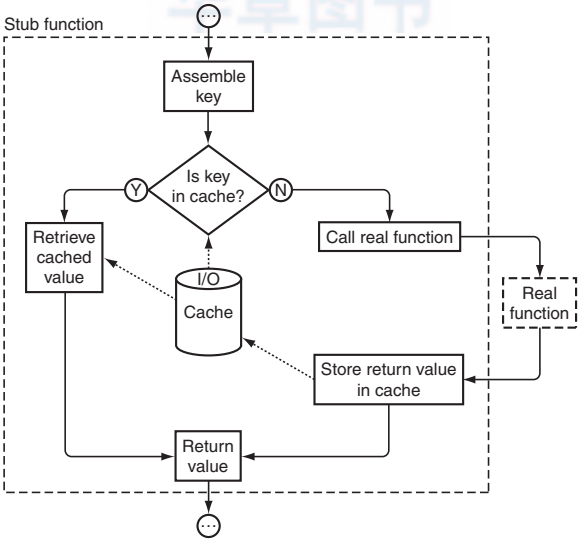


图 3-1 调用一个带记忆的函数

3.5.1 作用域和有效期

这里有一些细微的东西。首先，假设调用 memoize(\&fib) 并得到 $fastfib。然后调用使用 $func的$fastfib。一个平常的问题是，为什么当 memoize 返回时，$func 不会离开自己的作用域。

这个问题暴露了对作用域的一个普遍的误解。一个变量有两个组成部分：一个名字和一个值[⊖]。当把一个值和一个名字关联在一起时，就得到一个变量。这样一个关联就称为一个**绑定**（*binding*），也称名字**被绑定**（*bound*）在它的值上。

在 memoize 返回后尝试使用 $func，可能面临两个错误：值已经被销毁了，或者绑定已经改变了，以致名字指向了错误的值，或根本就没指向什么。

作用域

作用域（*scope*）是程序源代码的某部分，对其中某个绑定是有效的。在一个绑定的作用域内，名字和值是关联的；在此作用域外，绑定是**在作用域外的**（*out of scope*）并使得名字和值不再关联了。名字可能代表别的东西，或者根本什么也不代表。

当 memoize 启动时，my $func 声明产生了一个新的标量变量值，并把名字 $func 绑定上了。my 声明的名字的作用域，如 $func，从 my 声明以后的句子开始，一直到最小的闭合块结束。在这个例子里，最小的闭合块就是标记着 sub memoize。在这个块里面，$func 指向刚产生的词法变量；在块外面，它指向别的东西，可能是不相关的全局变量 $func。由于使用 $func 的存根在这个块里面，那里没有作用域问题，$func 的作用域就在存根里面，名字 $func 也保持它的绑定。

在 sub memoize 块以外，$func 意味着别的东西，但是存根是在块里面的，而不是在外面的。作用域是**词法域的**（*lexical*），也就是说这是个静态程序文本的属性，而不是某样东西执行次序的属性。事实上，存根在 sub memoize 块外**调用**（*called*）是不恰当的，它的代码“物理上存在于”$func 绑定的作用域。

%cache 的情况是一样的。

有效期

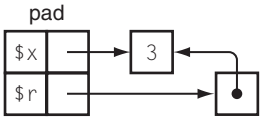
许多询问 $func 是否在作用域之外的人担心另一件事，不是作用域，而是很不一样的，称为**有效期**（*duration*）。一个值的有效期是在程序的运行期间是有效可用的。在 Perl 里，当一个值的有效期到了，它就被销毁，垃圾回收器使它的内存可供再次使用。

重要的是知道有效期几乎完全与名字无关，在 Perl 里，一个值的有效期持续到没有未完成的引用指向它。如果值是存放在一个具名变量里，那就算一个引用，然而还有其他种类的引用。例如：

```
my $x;
{
  $x = 3;
  my $r = \$x;
}
```

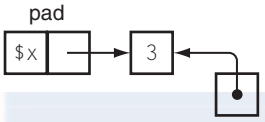
⊖ 这不是严格精确的。在命令式语言如 Perl 中，一个变量是一个名字与**值将被保存在**计算机内存的部分之间的关联。对于本文的讨论，这个区别并不重要。

这里有一个标量其值是 3。在块结束以后这里有两个引用指向它：

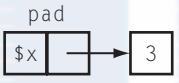


格子图（*pad*）是 Perl 在内部表示 my 变量绑定的数据结构。（对全局变量采用另一种数据结构。）一个指向 3 的引用来自格子图自身，因为名称 $x 被绑定到值 3。另一个指向 3 的引用是来自绑定到 $r 的引用值。

当控制离开块时，$r 就离开了作用域，因此 $r 到它的值的绑定就撤销了。在内部，Perl 从格子图里删除了 $r 绑定：

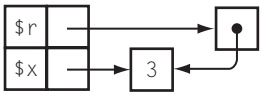


现在没有引用指向曾经存放在 $r 的引用值。由于没有任何东西再指向它，它的有效期结束了，Perl 立即销毁它：

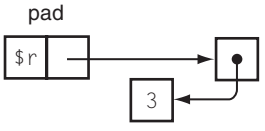


这是典型的：一个变量的名字离开了作用域，随后它的值也立即被销毁了。多数在作用域和有效期之间的混淆可能是由这个简单实例的普遍性引起的。然而作用域和有效期并不总是结合紧密的，正如下面这个例子将显示的：

```
my $r;
{
  my $x = 3;
  $r = \$x;
}
```



当控制离开块时，$x 的绑定就撤销了，$x 也从格子里被删除：



与之前的例子不同，没有绑定的值依然无限期地存在，因为它仍然被绑定到所指向的 $r 的引用。它的有效期直到这个引用消失了才结束。

作用域和有效期的分离是 Perl 变量的一个本质属性。例如，Perl 面向对象构造器函数的一段普通的模式：

```
sub new {
```

```
    ...
    my %self;
    ...
    return \%self;
}
```

这个构造器制造了一个散列，就是对象自身，然后返回一个指向散列的引用。即使名字 %self 离开了作用域，这个对象仍然会存留，只要主调者拥有一个指向它的引用。C 语言中类似的代码就是错误的，因为在 C 语言里，一个自动变量的有效期结束于它的作用域：

```
/* This is C */
struct st_object *new(...) {
  struct st_object self;
  ...
  return &self; /* expect a core dump */
}
```

现在回到 memoize。当 memoize 返回时，$func 的确已经离开作用域了。但是值没有被销毁，因为仍然有一个来自存根的未完成的引用。为真正理解将要发生什么，需要偷瞄一下 Perl 的内部（图 3-2）。

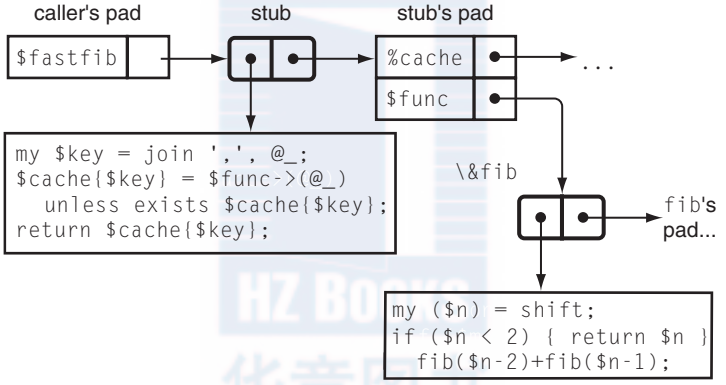


图 3-2 memoize 制造的数据结构

存根由图表顶部中间的双格子表示。在 Perl 中，这个格子称为一个 CV，即“Code Value”，它是一个代码引用的内部表现形式。（绑定到 $func 的代码引用展示在图表的右边。）一个 CV 实质上是一对指针：一个指向子例程的代码，另一个指向子例程定义时被激活的格子。$func 的绑定不会被销毁，直到所在的格子被销毁。格子不会被销毁，因为从 CV 有一个引用指向它。CV 不会被销毁，因为主调者通过把它赋值给 $fastfib 而存放在自己的格子里了。

Perl 知道存根可能在某天被调用，一旦如此，它会检查 $func 的值。只要存根存在，$func 的值就必须被保持原样。一个指向存根的引用存放在 $fastfib，只要引用在那里，存根就必须受到保护。类似地，缓存 %cache 坚持得和存根一样久。

3.5.2 词法闭包

现在另一点可能会混淆你。由于 $func 的值和存根坚持得一样久，如果第一次的存根尚存，那么第二次调用 memoize，那会发生什么？第二次调用时对 $func 的赋值会不会破坏第一个存根

使用的值呢?

答案是不会，一切都运作完美。这是因为 Perl 的匿名函数有个属性称为**词法闭包**（*lexical closure*）。当一个匿名函数产生后，Perl 把它的格子打包，包括所有在作用域里的绑定，把它们附在 CV 上。一个以这种方式打包了环境的函数就称为一个**闭包**（*closure*）。

当存根被调用的，原先的环境暂时被恢复，而存根函数代码运行在存根定义时就生效的环境中。词法闭包意味着一个匿名函数不管到哪里都会携带它的天生的环境，就像我遇到过的一些游客一样。

第一次调用 `memoize`，为使 `fib()` 带记忆，为了 `%cache` 和 `$func` 的绑定，建立了一个新的格子，新的存储空间分配给这些新的变量，`$func` 也初始化了。然后创造了存根，格子贴上了存根的 CV，然后 CV（称为 `fastfib()`）返回给主调者。

现在第二次调用 `memoize`，这次使 `quib()` 而不是使 `fib()` 带记忆（图 3-3）。同样的，一个新的格子产生了，新的 `%cache` 和 `$func` 变量绑定在其中。一个 CV（`fastquib()`）产生了，包含一个指向新格子的指针。新的格子与附着在 `fastfib()` 上的格子完全无关。

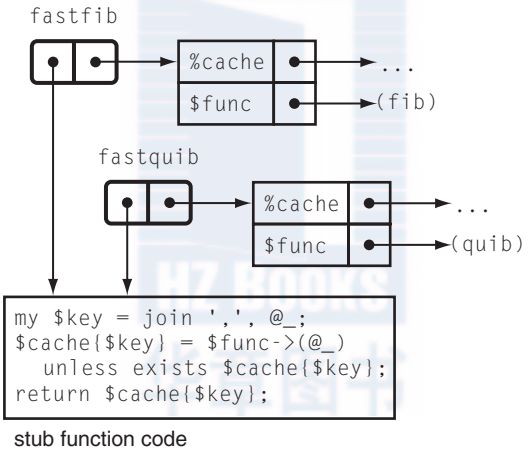


图 3-3 两次调用 memoize 后

当执行 `fastfib`时，`fastfib` 的格子暂时保持，`fastfib` 的代码被执行。代码使用了变量 `%cache` 和 `$func`，这些是在 `fastfib` 的格子里查找的。也许此时有些数据存储在 `%cache`。最后，`fastfib` 返回，旧的格子重新生效。

然后执行 `fastquib`，会发生几乎同样的事情。`fastquib` 的格子恢复，带有它自己的 `%cache` 和 `$func`。`fastquib` 的代码运行，它也使用名为 `%cache` 和 `$func` 的变量。它们是在 `fastquib` 的格子里查找的，后者与 `fastfib` 的格子没有联系。存储在 `fastfib` 的 `%cache` 的数据也完全不会被 `fastquib` 获取。

因为 CV 的代码部分是只读的，它在几个 CV 之间分享。这节约了内存。当一个 CV 的有效期到了，它的格子就被垃圾回收。

图 3-4 展示了一个简单的例子。

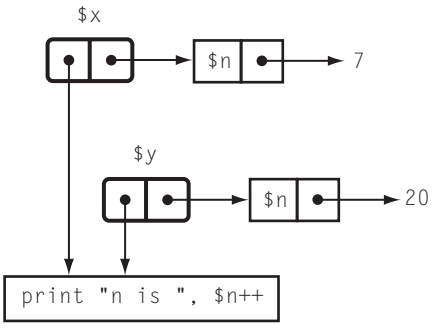


图 3-4 两次调用 make_counter 后

```
### Code Library: closure-example
sub make_counter {
  my $n = shift;
  return sub { print "n is ", $n++ };
}

my $x = make_counter(7);
my $y = make_counter(20);
$x->(); $x->(); $x->();
$y->(); $y->(); $y->();
$x->();
```

$x 现在包含一个闭包，其代码是 print "n is ", $n++ 而且该闭包的环境包含一个变量 $n，设置成 7。如果多次调用 $x：

```
$x->(); $x->(); $x->();
```

结果如下：

```
n is 7
n is 8
n is 9
```

图 3-5 展示了新的图片。

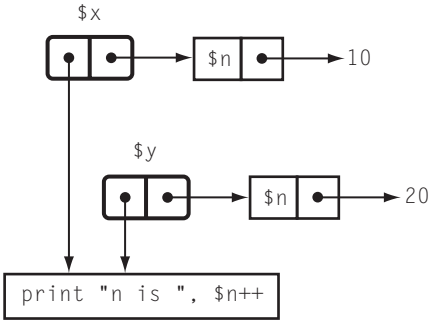


图 3-5

现在多次运行 $y：

```
$y->(); $y->(); $y->();
```

运行的是同样的代码，但是这次是在 $y 的格子里查找名字 $n 而不是在 $x 的格子里：

```
n is 20
n is 21
n is 22
```

图 3-6 展示了新的图片。

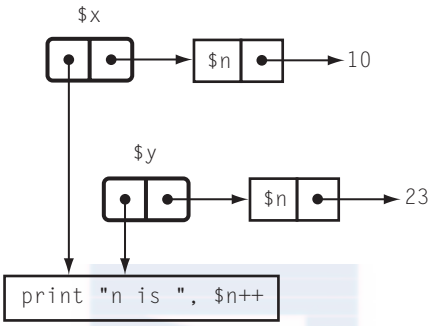


图 3-6

现在再次运行 $x：

```
n is 10
```

这里的 $n 和前三次调用 $x 时的是一样的，它恢复了自己的值。

3.5.3 再谈记忆术

之前所有的讨论都是为了解释 memoize 函数能工作的原因。潇洒地把它当成小事抛之脑后，“它当然能工作啦！”，值得注意的是，在许多语言中，这是不会工作的而且也是无法做到的。几个重要的复杂的特征不得不共同起作用：延迟的垃圾回收、绑定，匿名子例程的生成，以及词法闭包。例如，如果尝试在 C 语言里实现一个像 memoize 的函数，你会受骗，因为 C 语言没有任何那些特征。（3.11 节。）

3.6 CAVEATS

（这是拉丁文的“警告”。）

显然，记忆术不适合所有的性能问题。它甚至不适合所有的函数。有几类函数就不应该带记忆。

3.6.1 返回值不依赖参数的函数

记忆术最适合那些返回值只依赖它们的参数的函数。想象一下使时间函数带记忆的愚蠢：第一次你调用它，你将得到时刻，随后的调用将会返回**一样的**时刻。类似地，想象一个带记忆的随机数生成器是多么固执。

或者想象一个返回值是指示某类成功或失败的函数。你不会希望这类函数是带记忆的，每次

被调用都返回同一个值。

然而，记忆术适合一些这样的函数。例如，如果一个函数，结果依赖当日钟点的函数，且运行很长时间，那么使它带记忆就很有用了。（详见 3.7 节是如何处理这个的。）

3.6.2 有边界效应的函数

许多函数被调用是为了得到它们的边界效应而不是返回值。假设写了个程序，把计算机运行状态的报告整理并发送给打印机打印。那也许返回值就没什么意义了，缓存它是愚蠢的。即使返回值有意义，使其带记忆仍然是不合适的。函数在第一次运行后就会运行得很快，因为记忆术，但你的老板不会感动，因为它立即返回了旧的缓存了的返回值，而没有实际打印报告[⊖]。

3.6.3 返回引用的函数

这个问题有一点精妙。返回的指向值的引用可能被它们的主调者修改的函数肯定不能带记忆。

要看看潜在的问题，参考这个例子：

```
use Memoize;

sub iota {
  my $n = shift;
  return [1 .. $n];
}
memoize 'iota';

$i10 = iota(10);
$j10 = iota(10);
pop @$i10;
print @$j10;
```

第一次调用 `itoa(10)` 产生一个新的匿名的数组包含数字 1 到 10，并返回指向这个数组的引用。这个引用自动放到缓存里，也存放到了 `$i10`。第二次调用 `itoa(10)` 从缓存取回了一样的引用，并存放到了 `$j10`。`$i10` 和 `$j10` 现在都指向同一个数组，称它们是数组的**别名**（*alias*）。

当通过别名 `$i10` 改变数组的值时，这个改变也影响存放在 `$j10` 里的值！这也许不是主调者期望的，如果没有使 `itoa` 带记忆，这个也不会发生。记忆术应该是一个优化。也就是说它应该加速程序而不改变程序的行为。

禁止使那些返回的引用值可能被主调者改变的函数带记忆，可能非常普遍地适用于面向对象的构造器方法。考虑如下程序：

```
package Octopus;
sub new {
  my ($class, %args) = @_;
  $args{tentacles} = 8;
  bless \%args => $class;
}

sub name {
  my $self = shift;
  if (@_) { $self->{name} = shift }
```

⊖ 我有时喜欢练习想象使 Unix 的 `fork()` 函数带记忆的后果。

```
  $self->{name};
}
my $junko = Octopus->new(favorite_food => "crab cakes");
$junko->name("Junko");
my $fenchurch = Octopus->new(favorite_food => "crab cakes");
$fenchurch->name("Fenchurch");

# This prints "Fenchurch" -- oops!
print "The name of the FIRST octopus is ", $junko->name, "\n";
```

在这里程序员想要制造两个不同的章鱼，一个名为“Junko”，另一个名为“Fenchurch”。它们都喜欢蟹肉蛋糕。不幸的是，有人愚蠢地决定使 `new()` 带记忆，由于第二次调用的参数是一样的，记忆术的存根返回第一次调用时缓存的返回值，即指向“Junko”对象的引用。程序员认为那里有两只章鱼，但实际上只有一只，冒充两只。

返回值只依赖它们的参数，没有边界效应，且从不返回引用值的函数称为**纯函数**（*pure function*）。缓存技术非常适合用于纯函数，尽管它们有时也可以用于非纯函数。

3.6.4 带记忆的时钟

一个简单而有益的缓存非纯函数的例子是由 Perl 的 `$^T` 变量提供的。Perl 提供了几个方便的文件操作符，如 `-M $filename`，它返回以它的参数命名的文件最后一次修改至今的天数。为计算这个，Perl 向操作系统询问最后一次修改的时间，从当前时间减去前者，再转换成天数。由于 `-M` 可能被非常频繁地执行，如果它是快速的，那是很重要的：考虑如下这个：

```
@result = sort { -M $a <=> -M $b } @files;
```

它对一系列文件按照末次修改时间排序。找许多文件的末次修改时间已经代价很大了，就没必要再进行数千次的 `time()` 函数调用。更糟糕的是，系统可能是精确到秒记录时间的，如果系统时钟在执行 `sort()` 过程中发生步进，那结果列表可能是错误的次序。

为避免这些问题，Perl 不在一个 `-M` 操作执行时查找当前时间。而是当程序开始运行时，Perl 把当前时间缓存在特殊的变量 `$^T` 里，无论何时执行的 `-M`，都用这个作为当前时间。大多数程序的运行时间短，且大多数不需要从 `-M` 得到完全精确的结果，所以这通常是一个好主意。某些运行时间长的程序需要通过定期地执行 `$^T = time()` 刷新 `$^T`，以防止 `-M` 的结果偏离日期太远。当缓存一个非纯函数，通常的好主意是提供一个到期制度，这样旧的缓存的值最后被抛弃和刷新。允许程序员刷新整个缓存也是考虑周到的。模块 `Memoize` 提供了插入缓存到期管理器的机会。

3.6.5 非常快的函数

我曾经和一位程序员交谈，他抱怨他的函数在带记忆后是变慢了而不是变快了。结果发现他尝试加速的函数如下：

```
sub square { $_[0] * $_[0] }
```

缓存技术，像所有技术一样，是一个折中。可能的好处是减少了对原始函数的调用。代价就是程序每次调用时都必须检查缓存。先前，有公式 $hf - K$，表示了记忆术节约的时间。如果 $hf < K$，那么带记忆的版本就会比不带记忆的版本更慢。h 是缓存命中率，在 0 到 1 之间。f 是函数原先的运行时间，K 是检查缓存所需的平均时间。如果 f 小于 K，那么 $hf < K$ 就是必然的。如果检查缓存

的代价大于调用原始函数的代价，记忆术就不起作用。你无法通过避免“不必要的”调用节约时间，因为一开始就直接调用而花费更长的时间发现那个调用是不必要的。

在 square 例子里，函数做了一次单一的乘法。检查一次缓存需要一次散列查找，这包括一个散列值的计算（许多乘法和加法），然后在散列键桶里索引，也许还有一次链表搜索。这些不可能击败一次单一的乘法。事实上，几乎没什么能击败一次单一的乘法。你无法加速 square 函数，通过记忆术或任何别的方法，因为它已经几乎和任何函数所可能达到的速度一样快了。

3.7 键的生成

先前的记忆器至少有一个严重的问题。它需要把函数的参数转变成一个散列键，它的做法是使用 join：

```
my $key = join ',', @_;
```

这对只带一个参数的函数有效，也对参数不含逗号的函数有效，包括所有的参数是数字的函数。但是如果函数的参数可能包含逗号，它可能失效，因为以下两个调用却计算出了相同的键：

```
func("x,", "y");
func("x", ",y");
```

第一次调用，返回的值将以键 "x,,y" 存放在缓存里。当第二次调用时，没有运行真实的函数。而第一次调用的缓存的值将返回。但是函数本可能想要返回一个不同的值的，记忆术的代码混淆了这两个参数列表，导致一次错误的缓存命中。

由于这只会对那些参数包含逗号的函数失效，这可能不是考虑要点。即使函数的参数包含逗号，也有可能有些它们绝不会包含的字符。Perl 的特殊变量 $; 有时用在这里。它通常包含字符 #28，即控制－反斜杠字符。如果键生成器使用 join $;, @_，它只会在函数的参数包含控制－反斜杠时失效，一般可以确定这点将不会发生。但是经常有一个函数所带的参数会包含任何字符，这些不完整的特技不会可靠地工作。

这个可以被修正，因为总是有一种可靠的方法把任何数据结构，如这样一个参数列表，转换成一个字符串，那么不同的结构变成不同的字符串[⊖]。

一个策略是使用模块 Storable 或 FreezeThaw 把参数列表转换成一个字符串。一个更有效率的策略是使用转义序列：

```
my @args = @_;
s/([\\,])/\\$1/g for @args;
my $key = join ",", @args;
```

这里在原始参数的每个逗号或反斜杠前面都插入一个反斜杠，然后用不带反斜杠的逗号连接在一起。先前看到的问题调用也不再是问题了，因为两个参数列表转换成不同的键了：一个是 'x\,,y'，另一个是 'x,\,y'。（一个练习：为什么在每个反斜杠前面也要像在逗号前面一样放置一个反斜杠呢？）

然而，正确性是用昂贵的性能代价换来的。转义字符代码比简单的 join 慢得多，大约慢十倍，即使对一个简单的参数列表如 (1,2)，且它在**每次**调用函数时都必然会执行的。平常，我们

⊖ 要认清这点，只要意识到两个结构在内存中必然以不同的方式表现，计算机的内存本身就是一个非常长的字符串。

嘲笑那些想用正确性交换速度的人，因为不管找到错误答案的速度多快都没有意义。但这是一个不平常的情况。因为记忆术的唯一目的就是加速一个函数，使开销尽可能小。

采取折中。memoize 的默认行为将是快速的，但并不是在所有情况下都正确的。给 memoize 的用户一个应急出口修正。如果用户不喜欢默认的键生成方法，他们可以提供一个替代者，memoize 将使用这个。

改变是简单的：

```
### Code Library: memoize-norm1
sub memoize {
  my ($func, $keygen) = @_;
  my %cache;
  my $stub = sub {
    my $key = $keygen ? $keygen->(@_) : join ',', @_;
    $cache{$key} = $func->(@_) unless exists $cache{$key};
    return $cache{$key};
  };
  return $stub;
}
```

被 memoize 返回的存根会检查原始函数带记忆时是否提供了 $keygen 函数。如果提供了，它就用这个 keygen 函数构造散列键；如果没有提供，它就用默认的方法。额外的测试相当便宜，但如果想对 $keygen 执行一次测试，在函数带记忆的时候，而不是每次调用带记忆的函数都来一次，那么就可以消除这额外的测试：

```
### Code Library: memoize-norm2
sub memoize {
  my ($func, $keygen) = @_;
  my %cache;
  my $stub = $keygen ?
    sub { my $key = $keygen->(@_);
          $cache{$key} = $func->(@_) unless exists $cache{$key};
          return $cache{$key};
        }
  :
    sub { my $key = join ',', @_;
          $cache{$key} = $func->(@_) unless exists $cache{$key};
          return $cache{$key};
        }
  ;
  return $stub;
}
```

可以有一个更棒的戏法。在 memoize 的这些版本中，$keygen 是一个匿名函数，不得不在带记忆的函数每次调用时执行。不幸的是，Perl 在函数调用上有相对大的开销，且由于 memoize 的目的是加快速度，如果能做到，则希望避免这个开销。

Perl 的 eval 特性出现了。不再指定 $keygen 作为指向一个键生成函数的引用，传递一个字符串以包含生成键的代码，并把这代码直接合并入存根，而不是作为一个存根必须调用的子函数。

要使用这个版本的 memoize，声明如下：

```
$memoized = memoize(\&fib, q{my @args = @_;
                             s/([\\,])/\\$1/g for @args;
```

```
                                    join ',', @args;
                                  });
```

memoize 将把这一点代码插入带记忆的函数的模板里的合适位置（这称为**内联**（*inlining*））并使用 eval 把结果编译成一个真实的函数：

```
### Code Library: memoize-norm3
sub memoize {
  my ($func, $keygen) = @_;
  $keygen ||= q{join ',', @_};

  my %cache;
  my $newcode = q{
    sub { my $key = do { KEYGEN };
          $cache{$key} = $func->(@_) unless exists $cache{$key};
          return $cache{$key};
        }
  };
  $newcode =~ s/KEYGEN/$keygen/g;
  return eval $newcode;
}
```

这里使用Perl的q{...}操作符，除了单引号在q{...}里不再是特殊字符了，其他和'...'一样。q{...}结构在第一个匹配的}字符处结束。如果这里没有使用q{...}，第三行会相当晦涩难懂：

```
  $keygen ||= 'join \',\', @_';
```

用s///操作符内联$keygen的值，而不是简单地把它插入一个双引号字符串。这有一点效率损失，但是这只需要对每个带记忆的函数执行一次，所以这大概没关系。用s///技巧的好处是，$newcode变量很容易阅读；如果使用了字符串内插，它就会是如下这样的：

```
  my $newcode = "
    sub { my \$key = do { $keygen };
          \$cache{\$key} = \$func->(\@_) unless exists \$cache{\$key};
          return \$cache{\$key};
        }
  ";
```

代码里挤满了反斜杠。维护程序员阅读时可能不会注意到$keygen也被插值了，即便所有别的东西都带反斜杠了。使用s///技巧，KEYGEN显得清楚了。

对这个例子，缓存管理的开销比memoize内联的版本低大约37%。

很容易把这个版本稍微调整成和之前的一样仍然接受一个函数引用：

```
### Code Library: memoize-norm4
sub memoize {
  my ($func, $keygen) = @_;
  my $keyfunc;
  if ($keygen eq '') {
    $keygen = q{join ',', @_}
  } elsif (UNIVERSAL::isa($keygen, 'CODE')) {
    $keyfunc = $keygen;
    $keygen = q{$keyfunc->(@_)};
  }
  my %cache;
  my $newcode = q{
    sub { my $key = do { KEYGEN };
```

```
            $cache{$key} = $func->(@_) unless exists $cache{$key};
            return $cache{$key};
          }
    };
    $newcode =~ s/KEYGEN/$keygen/g;
    return eval $newcode;
}
```

这里，如果没有提供键生成器，则照常内联 `join ',', @_` 。如果 `$keygen` 是一个函数引用，则不能简单地内联它，因为它会变成像 `CODE(0x436c1d)` 的无用的东西。而是把函数引用保存在 `$keyfunc` 变量里，并内联一些将通过 `$keyfunc` 调用函数的代码。

这行代码 `UNIVERSAL::isa($keygen, 'CODE')` 需要解释一下。若要测试 `$keygen` 是否是一个代码引用。明显的做法如下：

```
if (ref($keygen) eq 'CODE') { ... }
```

不幸的是，Perl 的 `ref` 函数失效了，因为它混淆了它的参数的两个不同的属性。如果 `$keygen` 是一个 *bless* 过的代码引用，上面的测试将会失败，因为 `ref` 将返回 `$keygen` 被 bless 的类名。使用 `UNIVERSAL::isa` 就避免了这个问题。也有可能，尽管可能性不大，那个测试会对一个非代码引用产生真；这会发生在如果有人愚蠢到把非代码引用 bless 进 `CODE` 类的情况下。

3.7.1 用户提供的键生成器的更多应用

有了这些键生成的功能，`memoize` 函数的用户就有了一个应急出口，如果 `join` 方法没有尝试使带记忆的函数正确工作。他们可以以一个基于 `Storable` 的键生成器代替，或转义字符方法或合适的无论什么东西。

用户提供的键生成器解决了当两个不同的参数列表被弄成相同的键时出现的问题。它也解决了相反的问题，即当两个等价的参数列表被弄成不同的键时出现的问题。

考虑一个函数的参数是一个散列，可能包含键 `A`、`B` 和 `C` 中的任何一个或全部或一个都有没有，每个键关联一个数值。此外，假设如果 `B` 被省去，那 `B` 的默认值是 17，`A` 的默认值是 32：

```
sub example {
  my %args = @_;
  $args{A} = 32 unless defined $args{A};
  $args{B} = 17 unless defined $args{B};
  # ...
}
```

那么以下调用都是等价的：

```
example(C => 99);
example(C => 99, A => 32);
example(A => 32, C => 99);
example(B => 17, C => 99);
example(C => 99, B => 17);
example(A => 32, C => 99, B => 17);
example(B => 17, A => 32, C => 99);
(etc.)
```

键的构造的 `join` 方法对这些调用产生不同的键（`"C,99"` 对 `"A,32,C,99"` 对 `"C,99,A,32"` 等）。因此缓存管理将错过避免调用真实的 `example()` 函数的机会。调用一次

example(A =>32,C =>99) 必然和调用一次 example(C =>99, A =>32) 的结果一样，但是缓存管理不知道这个，因为参数列表看上去不一样。如果能安排等价的参数列表能转换成相同的散列键，那么缓存管理将对 example(C =>99, A =>32) 返回之前已经计算过的 example(A =>32, C =>99) 相同的结果，而不再冗余调用 example()。这将增加缓存命中率，公式 $hf-K$ 中的 h，该公式表现了记忆术带来的加速。下面的键生成器掌握了诀窍：

```
sub {
  my %h = @_;
  $h{A} = 32 unless defined $h{A};
  $h{B} = 17 unless defined $h{B};
  join ",", @h{'A','B','C'};
}
```

八个等价的调用（如 example(c =>99,A =>22) 就是其中之一）从这个函数得到同一个键 "32,17,99"。预付的代价是：这个键生成器比简单的 join 生成器慢十倍，所以公式 $hf-K$ 里的 K 就更大。这个代价能否弥补回来，依赖于真实函数调用的开销 f 多大，也依赖于缓存命中率 h 的提升幅度。通常，基准比较测试是没有替代物的。

3.7.2 内联的参数归一化的缓存管理

这里有一个有趣的技巧，可以与内联的键生成代码。考虑下面的函数例子的一个变体：

```
sub example {
  my ($a, $b, $c) = @_;
  $a = 32 unless defined $a;
  $b = 17 unless defined $b;
  # more calculation here ...
}
```

一个合适的键生成器如下：

```
  my ($a, $b, $c) = @_;
  $a = 32 unless defined $a;
  $b = 17 unless defined $b;
  join ',', $a, $b, $c;
```

有点让人不快的是不得不重复设置参数默认值的代码，同样令人不快的是不得不运行两次。如果改变键生成如下代码，就能把参数检查从实例函数中移除：

```
$_[0] = 32 unless defined $_[0];
$_[1] = 17 unless defined $_[1];
join ',', @_;
```

当这个被内联入 memoize 函数时，结果如下：

```
sub { my $key = do { $_[0] = 32 unless defined $_[0];
                     $_[1] = 17 unless defined $_[1];
                     join ',', @_;
                   };
      $cache{$key} = $func->(@_) unless exists $cache{$key};
      return $cache{$key};
    }
```

注意这里会发生什么。键和以前一样生成，但有个边界效应：@_ 被改变了。如果有一个缓存脱靶，

带记忆的函数就会以改变了的 @_ 调用 $func。由于 @_ 已经被改变了包含默认的值，可以从原始函数省略设置默认值的代码：

```
sub example {
  my ($a, $b, $c) = @_;
## defaults set by key generation code
##  $a = 32 unless defined $a;
##  $b = 17 unless defined $b;
  # more calculation here ...
}
```

当然，一旦这样修改了 example 函数，就不能关闭记忆术了，因为本质的功能已经被移入键生成器了。

这项技术的另一个危险之处是改变 @_ 会有个特殊效应返回**调用的**函数里。@_ 的元素被化名成回到主调者的相应的参数，在带记忆的函数里对 @_ 元素的赋值可以改变在带记忆的函数外部的变量。这里有一个简单的例子：

```
sub set_to_57 {
  $_[0] = 57;
}

my $x = 119;
set_to_57($x);
```

它会把 $x 设成 57，就像赋值 $x = 57 一样，即使赋值是在 $x 的作用域外执行的，$x 不应受影响。在键生成器代码里的赋值有类似的功能。

Perl 的这个特性有时是有用的，但多数情况下麻烦多于用处，通常应避免它。可以不直接操作 @_，而是当函数被调用时，就把它的内容复制给一连串词法变量：

```
sub safe_function {
  my ($n) = @_;
  $n = 57; # does *not* set $x to 57
}

my $x = 119;
safe_function($x);
```

通过组合这些技术，可以得到一版键生成代码，它除去了真实函数中设置默认值的代码的需求，而且仍然是安全的：

```
memoize(\&example, q{
  my ($a, $b, $c) = @_;
  $a = 32 unless defined $a;
  $b = 17 unless defined $b;
  @_ = ($a, $b, $c);            # line 5
  join ',', @_;
});
```

@_ 的元素是回到主调函数的参数的别名，但是 @_ 本身不是。第 5 行对 @_ 的赋值不会覆盖主调者的值，它完全丢弃了别名并以新的值替代 @_ 的内容。这个技巧只在键生成代码被内联入带记忆的函数时才有效；如果键生成代码是以子例程被调用，那么 @_ 的改变在子例程返回后没有效果。

3.7.3 带有引用参数的函数

定制的键生成器的特性解决了另一个问题。考虑如下函数：

```
sub is_in {
  my ($needle, $haystack) = @_;
  for my $item (@$haystack) {
    return 1 if $item == $needle;
  }
  return;
}
```

函数接受 $needle，即一个数字，以及 $haystack，即一列数字，当且仅当 $haystack 包含 $needle 时返回真。一个典型的调用如下：

```
if (is_in($my_id, \@employee_ids)) { ... }
```

如果想要尝试使 is_in 带记忆，但是一个可能的问题是参数 $haystack 是一个引用。当它被 join 函数处理时，它变成一个像 ARRAY(0x436c1d) 的字符串。如果后来以指向一个别的具有相同内容的数组的参数 $haystack 调用 is_in()，散列键将会不一样，这可能不是所需要的；相反，如果 @employee_ids 的内容改变了，散列键仍然是相同的，这肯定不是想要的。键生成器依照数组的同一性产生键，然而 is_in() 函数不关心数组的同一性，它只关心内容。这种情况下一个更合适的键生成函数是：

```
sub { join ",", $_[0], @{$_[1]} }
```

同样的，这实际上是否能带来性能优势依赖许多很难预见的情形。当性能很重要时，根本在于掌握真实的数据。长期的实践显示即便专家也时常猜错哪个快哪个慢。

3.7.4 划分

第 1 章的 find_share 函数提供了一个合适的函数的例子：记忆术修正了慢的递归，也需要一个定制的键生成器：

```
sub find_share {
  my ($target, $treasures) = @_;
  return [] if $target == 0;
  return    if $target < 0 || @$treasures == 0;
  my ($first, @rest) = @$treasures;
  my $solution = find_share($target-$first, \@rest);
  return [$first, @$solution] if $solution;
  return         find_share($target       , \@rest);
}
```

就像你将回忆起的，这个函数接受一个宝物数组和一个目标值，并尝试选出一组宝物的价值总和正好是目标值。如果有这么一组，它就返回一个只含这些宝物的数组；如果没有，它就返回 undef。

第 1 章的这个函数有个和 fib 函数一样的问题：它会很慢，因为它一遍又一遍地重复同样的工作。当尝试从 1 2 3 4 5 6 7 8 9 10 中选出总和是 53 的宝物，find_share 两次到达同样的位置：它发现 1+2+3+6=12，并执行 find_share(41, [7,8,9,10])，最终返回未定义值。后来，它又发现 1+2+4+5=12，并第二次执行 find_share(41, [7,8,9,10])。显然，这是一

个尝试缓存的好机会。

即使这个简单的例子，记忆术也产生了大约68%的速度提升。对于更大的例子，如find_share(200, [1..20])，速度的提升更大，大约82%。在有些例子里，记忆术能区分实用的和不实用的算法。不带记忆的find_share(210, [1..20])比带记忆的版本耗时长几千倍。(我使用了键生成函数sub {join "-", @{$_[1]}, $_[0]}。)

3.7.5 为非纯函数定制的键生成

定制的键生成也能用来处理这类函数，后者依赖它们参数以外的信息。

考虑一个长时间运行的网络服务程序，它的工作是卖东西，如披萨或武器级钚。一份披萨或一罐钚的价格包括运费，后者与相应的钟点和星期有关。晚上和周末发货比较贵，因为发货工人更少，且没人乐意在早上3点工作[⊖]。这个服务程序可能包含一个如下这样的函数：

```
### Code Library: delivery-charge
sub delivery_charge {
  my ($quantity_ordered) = @_;
  my ($hour, $day_of_week) = (localtime)[2,6];
  # perform complex computation involving $weight, $gross_cost,
  #      $hour, $day_of_week, and $quantity_ordered
  # ...
  return $delivery_charge;
}
```

因为这个函数是复杂的，所以想要使其带记忆。默认的键生成器，join(',', @_)，在这里是不适合的，因为这将失去运费的时间依赖性。但是用这样的一个定制的键生成函数，可以简单解决问题：

```
sub delivery_charge_key {
  join ',', @_, (localtime)[2,6];
}
```

delivery_charge不是一个纯函数，但在这个例子里无关紧要。唯一的真正的问题是，是否有足够的缓存命中率赢得性能。可以预见函数在第一周会有许多缓存脱靶，直到一周过去，然后开始看到更多的缓存命中。在这个例子里，缓存的效能可能就依赖程序运行的寿命了。类似地，可能在想下面的键生成函数会更好：

```
sub delivery_charge_key {
  my ($hour, $day_of_week) = (localtime)[2,6];
  my $weekend = $day_of_week == 0 || $day_of_week == 6;
  join ',', @_, $hour, $weekend;
}
```

这个函数运行时间更长，但是可能得到更多的缓存命中率，因为它认识到周一缓存的值可以被再次用在周二和周三。同样的，哪个更好将取决于程序行为里的微妙因素。

3.8 对象方法里的缓存

对象方法，它经常不理解地把缓存的值保存在独立的散列里。考虑一个投资银行写的程序里

⊖ 大多数钚都是在深夜下单的尽管要多费些钱。

的 Investor 对象。该对象表现了银行的一个客户：

```
package Investor;

# Compute total amount currently invested
sub total {
  my $self = shift;
  # ... complex computation performed here ...
  return $total;
}
```

如果 $total 不会改变，就可以缓存它，用对象的本身作为缓存散列的键：

```
# Compute total amount currently invested
{ my %cache;
  sub total {
    my $self = shift;
    return $cache{$self} if exists $cache{$self};
    # ... complex computation performed here ...
    return $cache{$self} = $total;
  }
}
```

然而，这个技术有一个严重的问题。当用一个对象作为一个散列键时，Perl 把它转换成一个字符串。典型的散列键将看上去像 Investor=HASH(0x80ef8dc)。十六进制数字是对象的数据实际存放的地址。本质上，这个键对任何两个对象都不一样，即冒着错误的缓存命中的风险，恢复一个对象的总数，却想着它属于另一个不同的对象。在 Perl 里，这些散列键确实对系统中任何给定的时刻存在的所有对象都不同，但对已回收的对象没有保证。如果一个对象被销毁了而一个新的对象被创建了，那个新的对象恰好存在于旧的对象先前占有的内存地址，那么混淆如下：

```
# here 90,000 is returned from the cache
$old_total = $old_object->total();
undef $old_object;
$new_object = Investor->new();
$new_total = $new_object->total();
```

这里要求新的投资者的投资总数。它应该是 0，因为投资者是新的。然而，->{total} 方法恰好查看缓存，以刚被销毁的 $old_object 用过的同样的散列键；这个方法看到 90000 存在那里，并错误地返回它。这个问题可以用一个 DESTROY 方法解决，它从缓存里删除一个对象的数据，或者在程序里对每个对象关联一个唯一的不再复用的 ID 数字，并用这个 ID 数字作为散列键，但是有个更直接的解决方法。

在一个面向对象编程上下文，缓存散列的技术是独特的，因为有个更自然的地方存放缓存的数据：就像数字存在于它的对象自身。一个缓存的总数变为另一个属性，后者可以或不被每个独立的对象携带：

```
# Compute total amount currently invested
sub total {
  my $self = shift;
  return $self->{cached_total} if exists $self->{cached_total};
  # ... complex computation performed here ...
  return $self->{cached_total} = $total;
}
```

这里的逻辑和之前的完全一样，唯一的不同是：这个方法为每个对象把总数保存在对象自身，而不是在一个辅助的散列。这避免了辅助散列带来的散列键冲突的问题。

这个技术的另一个优点是，用来保存缓存的总数的存储空间在对象销毁时会自动回收。用辅助的散列，每个缓存的值会一直存在，甚至在其所属的对象都被销毁以后。

最后，把缓存的信息分别存放在每个对象里在对象到期时带来了更灵活的控制。在例子里，`total` 计算某个投资者已经投资的总数。缓存这个总数是合适的，因为投资者不会太频繁地新投入钱。但是永久缓存它可能也不合适。在这个例子里，无论何时一个投资者投资更多的钱，都需要以某种方式告知 `total` 函数，缓存的总数不再正确了，必须要丢掉再重新计算。这就称为缓存值的**到期**（*expiring*）。

用辅助散列的技术，要是不在缓存散列的作用域里增加一个特殊需求的方法，就无法做到这一点，如下：

```
# Compute total amount currently invested
{ my %cache;
  sub total {
    my $self = shift;
    return $cache{$self} if exists $cache{$self};
    # ... complex computation performed here ...
    return $cache{$self} = $total;
  }

  sub expire_total {
    my $self = shift;
    delete $cache{$self};
  }
}

sub invest {
  my ($self, $amount, ...) = @_;
  $self->expire_total;
  ...
}
```

用面向对象技术，则不必增加特别的方法，因为每个方法都可以在它需要的时候直接使缓存的总数到期：

```
# Compute total amount currently invested
sub total {
  my $self = shift;
  return $self->{cached_total} if exists $self->{cached_total};
  # ... complex computation performed here ...
  return $self->{cached_total} = $total;
}

sub invest {
  my ($self, $amount, ...) = @_;
  delete $self->{cached_total};
  ...
}
```

3.8.1 对象方法的记忆术

对于对象方法，经常需要把每个计算过的值缓存在与它有联系的对象中，而不是一个独立的

散列。但之前介绍的 momoize 函数没有这么做，但是不难建立一个这么做的：

```
### Code Library: memoize-method
sub memoize_method {
  my ($method, $key) = @_;
  return sub {
    my $self = shift;
    return $self->{$key} if exists $self->{$key};
    return $self->{$key} = $method->($self, @_);
  };
}
```

$method 是一个指向真实方法的引用。$key 是缓存的值，将被保存于每个对象的位置的名字。这里返回的存根函数适合用做一个方法。当存根被执行时，它取回其行为被调用的对象，就像别的方法一样，然后它在对象里查找成员数据 $key，看是否有值缓存在那里。如果有，存根返回缓存的值；如果没有，它调用真实的方法，把结果缓存入对象，并返回新缓存的结果。

为使用这个，可以写成如下这样：

```
*Investor::total = memoize_method(\&Investor::total, 'cached_total');
$investor_bob->total;
```

这把存根安装入符号表以替代原始的方法。或者，可以使用：

```
$memoized_total = memoize_method(\&Investor::total, 'cached_total');
$investor_bob->$memoized_total;
```

这两者不完全一样。在前一种情况，所有对 ->total 的调用将使用方法的带记忆的版本，包括来自继承方法的子类的调用。在后一种情况，只有当使用 ->memoized(...) 以明确地要求时，才得到方法的带记忆的版本。

3.9 持续的缓存

在离开自动的记忆术的主题之前，将浏览一些外围的技术。将看到一个函数如何被带记忆的版本替代，后者在缓存里存储了返回值，缓存就仅是一个散列变量。

在 Perl 里，可以使用 tie 操作符把一个散列变量关联到一个磁盘上的数据库，那么存储在散列的数据会自动写到磁盘上，从散列取回数据实际上是从磁盘取回。把这个功能增加到 memoize 函数是简单的：

```
use DB_File;

sub memoize {
  my ($func, $keygen, $file) = @_;
  my %cache;
  if (defined $file) {
    tie %cache => 'DB_File', $file, O_RDWR|O_CREAT, 0666
      or die "Couldn't access cache file $file: $!; aborting";
  }
  my $stub = sub {
    my $key = $keygen ? $keygen->(@_) : join ',', @_;
    $cache{$key} = $func->(@_) unless exists $cache{$key};
    return $cache{$key};
```

```
    };
    return $stub;
}
```

此处增加了一个可选的第三个形参，它是将要接受缓存的数据的磁盘上的文件名。如果提供了，就用 tie 把散列关联到文件上。注意，如果不使用这个特性，就几乎不付出什么代价。在调用 memoize() 时会有一次单一的 defined() 测试。

当缓存散列以这种方式被关联到一个磁盘文件时，缓存就变成持续的了。程序一次运行时在缓存中存储的数据在程序退出以后还保留在文件里，而且在程序下次运行时依然对函数有效。程序逐步把函数替换成磁盘上的一个搜索表。程序员的代价几乎是零，因为不必改变原始函数的任何代码。

如果厌倦了等待查询表完全被占据，就可以强制它。也可以写个小程序，它仅是反复地以不同的参数调用带记忆的函数。可以在周五下午启动它然后回家过周末。当周一回来时，持续的缓存里将有函数预先计算出来的值。当运行实际应用时，对带记忆的函数的调用将几乎是立即返回，因为值已经保存在数据库里了。

同样的，这也可能不是胜利。记住，从记忆术获得的加速由公式 $hf-K$ 决定，K 是管理缓存的开销。如果 K 足够大，它将淹没从公式的 hf 部分赚到的，就像在 3.6 节的 sub { $_[0] * $_[0] } 例子中一样。当把缓存数据存储在一个磁盘文件里时，开销 K 会数倍于一般情况，因为程序将不得不进行一次操作系统请求以查看磁盘数据库。

另一种也是更加灵活的界面是允许 memoize() 的用户提供他们自己的关联散列：

```
sub memoize {
  my ($func, $keygen, $cache) = @_;
  $cache = {} unless defined $cache;
  my $stub = sub {
    my $key = $keygen ? $keygen->(@_) : join ',', @_;
    $cache->{$key} = $func->(@_) unless exists $cache->{$key};
    return $cache->{$key};
  };
  return $stub;
}
```

这允许用户提供一个以他们喜欢的 DBM 实现关联到一个磁盘文件的缓存，甚至可以是从没听说过的。他们也可以传递一个普通的散列，那将允许他们在他们希望的时候能清除缓存或让旧的值到期。

3.10 可供选择的记忆术

大多数纯函数提供一个缓存的机会。尽管乍一看纯函数很少，它们只以一定频率出现。纯函数特别普遍的地方是在排序中用做比较器函数。

Perl 内置的 sort 操作符是通用的，它可以把一列任何种类的数据以程序要求的任何次序排序。默认状态下，它把一列字符串以字母表次序排序，但是程序员可以任意提供一个**比较器函数**（*comparator function*），告诉 Perl 怎样重排 sort 的参数列表。比较器函数被反复调用，每次带有

待排序列表中的两个不同元素，如果两个元素次序正确，就必须返回一个负值；如果两个元素次序不正确，就返回一个正值，如果无所谓，就返回零。通常，一个比较器函数的返回值只依赖它的参数的值，待比较的两个列表条目，所以它是一个纯函数。

最简单的比较器函数的例子也许是按大小比较数字的比较器了：

```
@sorted_numbers = sort { $a <=> $b } @numbers;
```

这里 { $a <=> $b } 就是比较器函数。sort 操作符检查列表 @numbers，把 $a 和 $b 设置成将要比较的数字，然后调用比较器函数。<=> 是一个特殊的 Perl 操作符，如果 $a 小于 $b，就返回一个负值；如果 $a 大于 $b，就返回一个正值；如果 $a 与 $b 相等，就返回零[⊖]。cmp 是一个针对字符串的类似的操作符，如果不提供一个明确的比较器，Perl 就默认使用它。

一个可选的语法是使用一个具名函数而不是一个裸露的块：

```
@sorted_numbers = sort numerically @numbers;

sub numerically { $a <=> $b }
```

这等价于裸露块版本。

一个更有趣的例子是把一列形如 "Apr 16, 1945" 的日期字符串按时间次序排序：

```
### Code Library: chrono-1
@sorted_dates = sort chronologically @dates;

%m2n =
    ( jan =>      1, feb =>      2, mar =>      3,
      apr =>      4, may =>      5, jun =>      6,
      jul =>      7, aug =>      8, sep =>      9,
      oct =>     10, nov =>     11, dec =>     12, );

sub chronologically {
  my ($am, $ad, $ay) =
     ($a =~ /(\w{3}) (\d+), (\d+)/);
  my ($bm, $bd, $by) =
     ($b =~ /(\w{3}) (\d+), (\d+)/);

               $ay <=>          $by
  || $m2n{lc $am} <=> $m2n{lc $bm}
  ||           $ad <=>          $bd;
}
```

要被比较的两个日期字符串，和先前一样，放在 $a 和 $b 里，然后拆分成 $ay，$by，$am 等。$ay 与 $by，是年份，最先比较。这里的 || 操作符是个一般的习惯用法，在排序的比较器中用第二个关键词排序。|| 操作符返回它的左操作数，除非那是零，那种情况它就返回右操作数。如果年份相同，那么 $ay <=> $by 返回零，|| 操作符把控制传递到进入月份的表达式部分，用来解开关联的部分。但是如果年份不同，那么第一个 <=> 的结果是非零，这就是 || 表达式的结果，指示 sort 如何把 $a 与 $b 排序到结果列表中，而不必再看月份和日子了。如果控制传递到了 $am <=> $bm 部分，会发生同样的事情。月份被比较；如果结果是结论性的，那么函数立即返回，如果月份相同，控制传递到最后的日子比较的决胜局。

从内部看，Perl 的 sort 操作符已经被多种算法实现，运行时间是 $O(n \log n)$。这意味着对一

⊖ 这里减号也能工作得一样好，用在比较器的是 <=> 而不是普通的减号因为后者的文献记录值。

个大 *n* 倍的列表排序一般会花费比 *n* 倍稍微长的时间。如果列表规模加倍，运行时间比双倍更多。下面的表比较了参数列表的长度和比较器函数通常的调用次数：

Length	# calls	calls / element
5	7	1.40
10	26	2.60
20	60	3.00
40	195	4.87
80	417	5.21
100	569	5.69
1000	9502	9.50
10000	139136	13.91

我如此得到“# call”列的，依照指示的长度产生一列随机数，然后用一个比较器函数排序，每次被调用，计数器就增加。调用的次数将因列表和比较器函数而不同，但这些值是典型的。

现在考察有 10 000 个日期的列表。139 136 次比较器函数的调用，每次调用执行两次模式匹配操作，所以一共有 278 272 次模式匹配。这意味着每个日期平均拆分成年、月、日 27.8 次。由于给定日期的这三个组成部分不会改变，显然 26.8 次模式匹配是浪费的。

首先想到的是，使 chronologically 函数带记忆，但这实际上行不通。尽管 sort 将以相同的日期反复调用 chronologically 函数，但它不会对同一**对**（*pair*）日期调用两次（除非，当然，输入列表包含重复的日期）。由于散列键必须结合两个参数，带记忆的函数将永远不会有一次缓存命中。

而本文是采取稍微不同的做法，只使函数中浪费的部分带记忆。这将需要能处理一个返回列表的函数的 memoize() 版本。

```
### Code Library: chrono-2
@sorted_dates = sort chronologically @dates;

%m2n =
   ( jan =>      1, feb =>      2, mar =>      3,
     apr =>      4, may =>      5, jun =>      6,
     jul =>      7, aug =>      8, sep =>      9,
     oct =>     10, nov =>     11, dec =>     12, );

sub chronologically {
  my ($am, $ad, $ay) = split_date($a);
  my ($bm, $bd, $by) = split_date($b);

              $ay <=>          $by
  || $m2n{lc $am} <=> $m2n{lc $bm}
  ||          $ad <=>          $bd;
}

sub split_date {
  $_[0] =~ /(\w{3}) (\d+), (\d+)/;
}
```

如果对 split_date 设置缓存，仍将进行 278 272 次调用，但是 268 272 次将导致缓存命中，只有剩下的 10 000 次需要模式匹配。唯一的挑战是将不得不手动写缓存代码，因为 split_date 返回一个列表，而 memoize 函数只能正确处理返回标量的函数。

此刻，可以向三个方向行进。可以增强 memoize 函数能正确处理列表上下文返回。（或者可

以使用 CPAN Memoize 模块，它能对返回列表的函数正确工作。）可以手写缓存代码。但更有益的是绕过这个问题，通过用一个返回标量的函数替代 split_date。如果标量构造正确，将能免除 chronologically 中复杂的 || 逻辑，而仅使用一个简单的字符串比较。

这里有一个策略：拆分日期，和先前一样，但是不再返回一列字段，而是把一列字段放入一个单一的字符串。字段将按照要检查的次序出现在字符串中，年份最先，然后是月份，然后是日期。对 "Apr 16, 1945" 的字符串将是 "19450416"。当用 cmp 比较字符串时，Perl 将尽快停止比较，因此如果一个字符串以 "1998..." 开头，而另一个以 "1996..." 开头，Perl 一看到第四个字符就知道了结果，不再操心检查月份和日期。字符串的比较是非常快的，多半可以战胜一系列 <=> 和 || 。

修改后的代码如下：

```
### Code Library: chrono-3
@sorted_dates = sort chronologically @dates;
%m2n =
   ( jan =>      1, feb =>      2, mar =>      3,
     apr =>      4, may =>      5, jun =>      6,
     jul =>      7, aug =>      8, sep =>      9,
     oct =>     10, nov =>     11, dec =>     12, );

sub chronologically {
  date_to_string($a) cmp date_to_string($b)
}

sub date_to_string {
  my ($m, $d, $y) = ($_[0] =~ /(\w{3}) (\d+), (\d+)/);
  sprintf "%04d%02d%02d", $y, $m2n{lc $m}, $d;
}
```

现在可以使 date_to_string 带记忆。这能否战胜先前的版本，依赖 sprintf 加 cmp 是否比 <=> 加 || 更快。一般需要一个基准比较测试，它证明带 sprintf 的代码要快大约两倍[⊖]。

排序经常是在程序里需要尽可能压榨出最多性能的地方之一。对一个 10 000 个日期的列表，可以精确地调用 sprintf 10 000 次（一旦 date_to_string 已经带记忆），但是仍然要调用 date_to_string 278 272 次。随着日期列表变长，这种不对称也将增长，函数调用的时间最终将超过排序的运行时间。

可以通过简化缓存处理和削减 268 272 次额外的函数调用获得更多的速度优势。为此，回到手写的缓存代码：

```
### Code Library: chrono-orc
{ my %cache;
  sub chronologically {
    ($cache{$a} ||= date_to_string($a))
       cmp
    ($cache{$b} ||= date_to_string($b))
  }
}
```

⊖ 这对许多人是个惊喜，特别是对认为 sprintf 慢的 C 程序员。认为 sprintf 慢，所以 Perl 也慢，结果分配一堆额外的 <=> 和 || 操作符为 sprintf 花费了很长时间比较。这只是另一个为什么基准是确实需要的实例。

这里使用 ||= 操作符，看上去几乎是为缓存应用定制的。$x ||= $y 产生 $x 的值，如果 $x 是真的；如果不是，它把 $y 赋值给 $x 并产生 $y 的值。$cache{$a ||= date_to_string($a)} 查看 $cache{$a} 是否有一个真值，如果有，那就是用 cmp 操作符比较时使用过的值。如果还没有任何东西缓存，那么 $cache{$a} 是假，然后 chronologically 就调用 date_to_string，把结果存在缓存里，并在比较时使用这个结果。这种内联的缓存技术就称为 *Orcish Maneuver*，因为它的本质特性是 || 和缓存[⊖]。

使 date_to_string 带记忆，产生 2.5 倍的加速；用 Orcish Maneuver 代替记忆术产生**额外的**两倍加速。

机敏的读者将会注意到 Orcish Maneuver 不总是完全正确。在这个例子里，date_to_string 不可能总是返回一个假值。但是短暂返回计算每个投资者的投资总数的例子：

```
{ my %cache;
  sub by_total_invested {
    ($cache{$a} ||= total_invested($a))
       <=>
    ($cache{$b} ||= total_invested($b))
  }
}
```

假设 Luke Hermit 根本没投资过。他第一次出现在 by_total_invested 时，为 Luke 调用 total_invested，然后得到 0。把这个 0 以 Luke 为键存放在缓存里。Luke 下次出现时，检查缓存并发现存放在那里的值是 0。因为这个值是假，所以再次调用 total_invested，即使已经命中缓存了。这里的问题是 ||= 操作符无法区分缓存脱靶与缓存命中的值恰好是假。

Lisp 玩家给这种现象取了个名字：称为**半谓词问题**（*semipredicate problem*）。一个**谓词**（*predicate*）就是一个返回布尔值的函数。一个**半谓词**（*semipredicate*）能返回一个特定的假值，表示失败，或者许多有意义的真值之一，表示成功。$cache{$a} 是一个半谓词，因为它可能返回 0，或者无数有用的真值之一。当 0 **也**是真值之一时，就遇到麻烦了，因为无法把它与 0 意味着假区分开。这就是半谓词问题。

在先前的例子里，半谓词问题不会引起太多的麻烦。仅有的代价就是对那些没有投资过的人们会多些额外的 total_invested 调用。如果发现这些额外的调用在明显地拖慢排序（不经常，但有可能），就可以用下面的版本替换比较器函数：

```
{ my %cache;
  sub by_total_invested {
    (exists $cache{$a} ? $cache{$a} : ($cache{$a} = total_invested($a)))
       <=>
    (exists $cache{$b} ? $cache{$b} : ($cache{$b} = total_invested($b)))
  }
}
```

这个版本使用了可靠的 exists 操作符检查缓存是否被占据。即使存储在缓存的值是假，exists 仍将返回真。请注意，不过，这样比简单版本慢 10% 左右。

还有个方法几乎没有额外的代价，但具有奇异的缺点。它基于这样的秘籍：当 Perl 里的字符串 "0e0" 用做一个数字时，它就完全和 0 一样；e0 被 Perl 解释成科学计数法的指数。但是和普

⊖ Joseph Hall，《Effective Perl Programming》的作者，这么称呼它的。

通的 0 不同，字符串 "0e0" 是真而不是假[⊖]。

如果这样写 by_total_invested，就几乎没付出额外的代价而避免了半谓词问题：

```
{ my %cache;
  sub by_total_invested {
    ($cache{$a} ||= total_invested($a) || "0e0")
       <=>
    ($cache{$b} ||= total_invested($b) || "0e0")
  }
}
```

如果 total_invested 返回零，函数就缓存 "0e0"。下次寻找同一个客户投资的总数时，函数在缓存里看到 "0e0"，而这个值是真，所以它不会第二次调用 total_invested。这个 "0e0" 就是给 <=> 操作符比较的值，但是在数字比较中它表现得和 0 完全一样，这也正是我们期望的。额外的 || 操作的速度损失，仅存在于 total_invested() 返回一个假值的时候，是非常小的。

3.11 传播福音

如果你正尝试向一个 C 程序员解释为什么 Perl 好，自动的记忆术是一个好例子。几乎所有的程序员都熟悉缓存技术。即使他们没在自己的程序里使用过任何缓存技术，他们也一定熟悉这个概念，来自网页浏览器里的，他们的计算机的缓存内存里的，DNS 服务器里的，他们的网页代理服务器里的，或别的地方的缓存。缓存，就像大多数简单实用的主意，是无处不在的。

增加缓存不是非常麻烦，但至少需要几分钟修改代码。算上所有修改，你有可能犯错，这不得不计算进平均时间，一旦你完成了，可能发现缓存是个糟糕的主意，因为缓存管理的开销比函数的运行时间更长，或者因为在一次典型运行中，缓存命中的没有期望的多；那么就不得不移除缓存代码，然后你再次冒着犯错误的风险。这不是夸大问题，当然，在每个方向上至少会花费几分钟。

有了记忆术，增加缓存代码不再需要几分钟了，只要几秒钟就可以了。可以增加一行这样的代码：

```
memoize 'myfunction';
```

然后就不可能犯下严重的错误和破坏函数。如果记忆术表明这是个糟糕的主意，你可以在**一秒钟**以内关闭它。多数程序员会对这样的便利满意的。如果你有五分钟向一个 C 程序员解释 Perl 比 C 好在哪里，记忆术是一个很好的例子。

3.12 速度的好处

此刻有个说法比较诱人，那就是记忆术只比手动缓存技术改善了一点，因为它做了同样的事

⊖ "0e0" 几乎是独一无二的；"00" 也有效，就像所有以 0 开头并紧跟非数字的字符串一样，如 "0!!!"。然而，如果警告开启，像 "0!!!" 的字符串，将会产生一个 “Argument isn't numeric” 警告。一个常用来表示是零但真的字符串是 "0 but true"。Perl 的警告系统对这个字符串特别处理，不出现警告 “isn't numeric”。

情，仅有的额外的好处是你可以快速启动或者关闭它。但这不完全正确。当工具之间的速度和便利的差别够大，它会改变你思考和使用工具的方式。自动地使一个函数带记忆仅需要消耗手动书写代码的 1/100 的时间。这和飞机与牛车的速度差别一样。说飞机只是更快的牛车就忽略了本质：量变如此巨大以致它也变成了实质的质变。

例如，有了自动的记忆术，就有可能为函数增加缓存行为而不必预先仔细考虑性能细节。记忆术这么简单以至于可以采取一种称为“走着瞧”的策略。如果一个函数很慢，试着为它增加一些缓存看看是否有帮助。如果一个递归函数可能会有错误的递归行为，放一些缓存看看问题有没有消失。如果没有，你可以再次把缓存移除并更彻底地调查。所有的代价就是十秒左右的编程时间，你可以尝试，而不用预先考虑太多它将来是否会成功。要是手动缓存，你将不得不花费至少一刻钟，这足够去钓次鱼了。

有了自动的记忆术，你可以在运行时启动缓存行为。例如：

```
sub function {
  if (++$CALLS == 100) { memoize 'function'}
  ...
}
```

这里直到程序运行到某种程度时再使函数带记忆。当函数意识到自己被频繁使用时，它就激活缓存行为。要是没有自动的记忆术，那么做同样的事情就需要重写函数而不是增加单独的一行了。

3.12.1 剖析和性能分析

自动的记忆术使缓存以某种方式应用于剖析和性能分析中，否则就不实际了。典型的情况是卷入一个运行得太慢的巨大应用。如果希望加速它，就将重写部分程序以使它更快，也许引入一个更好的算法，也许牺牲一定的清晰度和可维护性。

试图加速程序的每一部分是不好的资源分配。这是因为所谓的“90-10 法则”，一个程序的 90% 的运行时间发生在仅 10% 的代码中，剩余的是只执行一次的初始化代码，或者像错误处理那样很少或根本不执行的特殊情况的代码。如果检查全部程序并对每部分都加速 5%，那么赚到 5%。但是如果能识别并重写关键的 10% 并得到同样的加速，那么将以 10% 的代价净赚程序全部运行时间的 4.5%，付出回报比改善了九倍。因此在优化之前，非常希望识别出程序中贡献最多运行时间的部分，并只集中改善那些部分。

悲惨的是当一个程序员花了一周时间仔细优化了一个子例程让它快了 20%，却发现那个子例程的运行时间只占全部的 2%，那一周的辛勤劳动对全局只产生了 0.4% 的提速。长期以来，程序员在猜测哪部分程序被频繁使用上很糟糕，因此需要真实的测量。

通常，测量时使用一种称为**剖析**（*profiler*）的工具。程序运行在一个特殊的剖析环境中，后者使程序非常频繁地（通常是每秒许多次）产生一个它正在做什么的记录。随后，数据被处理成一个报告以列出程序中占用最多运行时间的子例程。有一些针对 Perl 的剖析工具，但它们可能是陌生的且难以使用的。自动的记忆术是另一个选择。

运行一次程序并记下运行耗时。然后猜测程序的哪部分是瓶颈，并使它们带记忆。安排好带记忆的数据存放在固定的文件里。（记住，这仅需要在程序中添加一行代码。）第二次运行这

个程序，它将占据磁盘上的缓存。第三次运行这个程序。对带记忆的函数的所有调用都将几乎立即返回，因为数据驻留在一个磁盘数据库上，除了被要求从数据库得到答案，函数根本不用做什么。第三次运行，便在模拟如果可能把目标函数的耗时移除后程序会有多快。如果这次运行比不带记忆的快得相当多，就有了优化候选了；如果运行时间类似，就该知道应当到别处看看了。

你可能疑惑当带记忆的运行耗时短得多时为什么不简单地把记忆术留在那里，答案是记忆术可能使目标函数运行更快，也可能使它们不正确。假设你怀疑瓶颈函数是那个排版报告的函数。当使之带记忆并让它放出预先缓存的报告可能使它运行得更快，然而报告可能不是接受者满意的。

3.12.2 自动剖析

另一种剖析技术，甚至更巧妙，使用在这章介绍过的技术，但不带任何实际的缓存。memoize 函数接受一个存在的函数并把缓存管理放在它的前端。没有任何理由让前端必须做缓存管理，它也可以做别的事情：

```
### Code Library: profile
use Time::HiRes 'time';
my (%time, %calls);

sub profile {
  my ($func, $name) = @_;
  my $stub = sub {
    my $start = time;
    my $return = $func->(@_);
    my $end = time;
    my $elapsed = $end - $start;
    $calls{$name} += 1;
    $time{$name} += $elapsed;
    return $return;
  };
  return $stub;
}
```

此处展现的 profile 函数在结构上与先前介绍的 memoize 函数类似。像 memoize，它接受一个函数作为它的参数并构造和返回一个存根，后者可以被直接调用或安装入符号表以代替原始的。

当存根被执行时，它把当前时间记录在 $start。通常 Perl 的 time 函数返回最接近当前时间的秒数，如果可能 Time::HiRes 模块提供一个具有更高精度的函数以替代 time 函数。存根调用真实的函数并保存它的返回值。然后它计算经过的时间并更新两个散列。一个散列记录了每个函数被调用的次数，存根简单地增加那个数字。另一个散列记录了花在运行每个函数上的总耗时，存根把刚结束的调用的耗时加到总数上。

在程序运行的结尾，可以打印出一个报告：

```
END {
  printf STDERR "%-12s %9s %6s\n", "Function", "# calls", "Elapsed";
  for my $name (sort {$time{$b} <=> $time{$a}} (keys %time)) {
```

```
        printf STDERR "%-12s %9d %6.2f\n", $name, $calls{$name}, $time{$name};
    }
}
```

输出将如下：

```
              Function          # calls             Elapsed
              printout                1               10.21
             searchfor                1                0.34
                  page                1                0.06
            check_file               18                0.01
```

这是来自标准 Perl 的 perldoc 程序的输出。从这份输出可以发现多数的运行时间出现在 printout 函数；如果想要使 perldoc 更快，这个就是应当关注的函数。

3.12.3 钩子

很明显这是一个非常简陋的剖析工具。更好的版本应该使用 times() 函数测量 CPU 消耗时间而不是挂钟时间。但是这个技术的灵活性应该是清楚的，可以在函数上放置任意前端，或者在运行时改变前端。前端可以执行缓存，或者保持函数调用数据的轨迹；如果想要，它可以确认函数的参数，强置一些前置或后置条件，或任何别的想要的东西。