第2章 分 配 表

第1章介绍了如何用别的函数参数化函数的行为使函数更加灵活。例如，并没有把每次移动盘子就输出一条消息硬编码到 hanoi() 函数里，而是让其调用一个从外部传入的辅助函数。通过提供一个合适的辅助函数，可以使 hanoi() 输出一系列说明，或检查它自己的行动，或生成一个图形显示，而不必重新编写基本的算法。类似地，可以从 total_size() 函数的计算文件大小的行为中提取出目录遍历行为，得到一个更有价值和普遍适用的 dir_walk() 函数，它可以做许多不同的事情。

为了从 hanoi() 与 dir_walk() 提取出行为，使用了代码引用。把别的函数作为参数传递给 hanoi() 与 dir_walk() 函数，有效地把辅助函数当成数据块。代码引用使这些成为可能。

现在先不讲递归，而叙述代码引用的另一种用法。

2.1 配置文件处理

假设我们有一个应用要读取一个如下格式的配置文件：

```
VERBOSITY       8
CHDIR           /usr/local/app
LOGFILE         log
...             ...
```

要读取这个配置文件并根据每个指示采取适当的行动。例如，对于 VERBOSITY 指示，只是设置一个全局变量。而对于 LOGFILE 指示，则要立即重定向程序的诊断消息到指定的文件。对于 CHDIR，也许可以让程序 chdir 指定的目录以使随后的文件操作与新的目录相关联。这意味着，在之前的例子里 LOGFILE 是 /usr/local/app/log，而不是用户在程序运行时恰好所在的目录下的 log 文件。

许多程序员会遇到这个问题并会立即想象到一个含有巨大 if-else 分支的函数，如下：

```
sub read_config {
  my ($filename) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    if ($directive eq 'CHDIR') {
      chdir($rest) or die "Couldn't chdir to '$rest': $!; aborting";
    } elsif ($directive eq 'LOGFILE') {
      open STDERR, ">>", $rest
        or die "Couldn't open log file '$rest': $!; aborting";
    } elsif ($directive eq 'VERBOSITY') {
```

```
        $VERBOSITY = $rest;
      } elsif ($directive eq ...) {
        ...
      } ...
      } else {
        die "Unrecognized directive $directive on line $. of $filename; aborting";
      }
    }
    return 1; # Success
}
```

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 指示的要打开的日志文件名。函数的第二部分是一棵大的 if-else 树，它检查 $directive 变量，查看它是哪个指示，如果指示不可识别，则中断程序。

这类函数可以变得非常庞大，因为在 if-else 树中有许多选项。每次有人想增加一个指示，他就要改变函数增加一个 elsif 分句。if-else 树的分枝的内容相互之间没有很多事情要做，除了它们都是可配置的琐碎事实。这样的函数违背了编程的一条重要法则：相关的东西应该放在一起；不相关的东西应该分开。

依照此法则为这个函数提出了一个不同的结构：读取和解析文件的部分应该与配置的指示被识别后的执行动作分开。此外，实现各种不相关的指示的代码不应该一起挤进单个函数。

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。

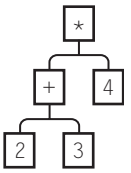

图 2-1 表达式 2 3 + 4 * 的 AST

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。