# 第9章 实践：建立单元测试框架

在本章里，你将编写代码开发一个简单的Lisp单元测试框架。这将使你有机会在真实代码中使用从第3章起已学到的某些语言特性，包括宏和动态变量。

该测试框架的主要设计目标是使其可以尽可能简单地增加新测试，运行多个测试套件，以及跟踪测试的失败。目前，你将集中于设计一个可以在交互开发期间使用的框架。

一个自动测试框架的关键特性在于该框架应该能够告诉你是否所有的测试都通过了。当计算机可以处理得更快更精确时，你就不应该将时间花在埋头检查测试所输出的答案上。因此，每个测试用例必须是一个能产生布尔值的表达式——真或假，通过或失败。举个例子，如果正在为内置的“+”函数编写测试，那么下面这些可能是合理的测试用例：[1]

```
(= (+ 1 2) 3)
(= (+ 1 2 3) 6)
(= (+ -1 -3) -4)
```

带有副作用的函数会以稍微不同的方式进行测试。你必须调用该函数，然后查找是否有证据表明存在着预期的副作用。[2]但最终，每个测试用例都将归结为一个布尔表达式，要么真要么假。

## 9.1 两个最初的尝试

如果正在做测试，那么就可以在REPL中输入这些表达式并检查它们是否返回T。但你可能想要一个框架使其可以在需要时轻松地组织和运行这些测试用例。如果想先处理最简单的可行情况，那就可以只写一个函数，让它对所有的测试用例都予以求值并用AND将结果连在一起：

```
(defun test-+ ()
  (and
```

① 这仅仅是出于阐述目的。很明显，编写对于诸如“+”这样的内置函数的测试用例有点荒唐，因为如果连这么基本的东西都无法正常工作的话，那么测试过程按照你期待的方式运行的可能性也微乎其微。另一方面，多数Common Lisp平台在很大程度上是用Common Lisp本身实现的，因此不难想象可以用Common Lisp编写测试套件来测试标准库函数。

② 副作用也可以包括诸如报错这样的情况，我将在第19章里讨论Common Lisp的错误。你可以在读过那章以后再来考虑如何在测试中检测一个函数是否在特定情况下产生了一个特别的错误。

```
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

无论何时，当想要运行这组测试用例时，都可以调用test-+。

```
CL-USER> (test-+)
T
```

一旦它返回**T**，就可知道测试用例通过了。这种组织测试的方式也很优美简洁——不需要编写大量的重复测试代码。然而一旦发现某个测试用例失败了，同样也会发现它的运行报告会遗漏一些有用的信息。当test-+返回**NIL**时，你会知道某些测试失败了，但却不会知道这究竟是哪一个测试用例。

因此，让我们来尝试另一种简单得甚至有些愚蠢的方法。为了找出每个测试用例的运行情况，你可以写成类似下面这样。

```
(defun test-+ ()
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

现在每个测试用例都将单独报告结果。**FORMAT**指令中的~:[FAIL~;pass~]部分将导致**FORMAT**在其第一个格式实参为假时打印出FAIL，而在其他情况下为pass。[①]然后会将测试表达式本身标记到结果上。现在运行test-+就可以明确显示出发生了什么事。

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
NIL
```

这次的结果报告更像是你想要的，可是代码本身却一团糟。问题出在对**FORMAT**的重复调用以及测试表达式乏味的重复，这些急切需要被重构。其中测试表达式的重复尤其讨厌，因为如果发生了错误输入，测试结果就会被错误地标记。

另一个问题在于，你无法得到单一的关于所有测试是否都通过的指示。如果只有三个测试用例的话，很容易通过扫描输出并查找“FAIL”来看到这点。不过当有几百个测试用例时，这将非常困难。

## 9.2　重构

我们真正所需要的编程方式应该是可以写出像第一个test-+那样能够返回单一的**T**或**NIL**值的高效函数，但同时它还可以像第二个版本那样能够报告单独测试用例的结果。就功能而言，由于第二个版本更接近于预期结果，所以最好是看看能否可以将某些烦人的重复消除掉。

消除重复的**FORMAT**相似调用的最简单方法就是创建一个新函数。

① 我将在第18章里讨论包括这个指令在内的**FORMAT**指令的更多细节。

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form))
```

现在就可以用report-result来代替**FORMAT**编写test-+了。这不是一个大的改进，但至少现在如果打算改变报告结果的方式，则只需要修改一处即可。

```
(defun test-+ ()
  (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

接下来需要摆脱的是测试用例表达式的重复以及由此带来的错误标记结果的风险。真正想要的应该是可以将表达式同时看作代码（为了获得结果）和数据（用来作为标签）。无论何时，若想将代码作为数据来对待，这就意味着肯定需要一个宏。或者从另外一个角度来看，你所需要的方法应该能够自动编写容易出错的report-result调用。代码可能要写成下面这样。

```
(check (= (+ 1 2) 3))
```

并要让其与下列形式含义等同。

```
(report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
```

很容易就可以写出一个宏来作这种转换。

```
(defmacro check (form)
  `(report-result ,form ',form))
```

现在就可以改变test-+来使用check了。

```
(defun test-+ ()
  (check (= (+ 1 2) 3))
  (check (= (+ 1 2 3) 6))
  (check (= (+ -1 -3) -4)))
```

既然不喜欢重复的代码，那为什么不将那些对check的重复调用也一并消除掉呢？你可以定义check来接受任意数量的形式并将它们中的每个都封装在一个对report-result的调用里。

```
(defmacro check (&body forms)
  `(progn
     ,@(loop for f in forms collect `(report-result ,f ',f))))
```

这个定义使用了一种常见的宏习惯用法，将一系列打算转化成单一形式的形式分装在一个**PROGN**之中。注意是怎样使用,@将反引用模板所生成的表达式列表嵌入到结果表达式之中的。

有了check的新版本就可以写出一个像下面这样新版本的test-+~：

```
(defun test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

其等价于下面的代码：

```
(defun test-+ ()
  (progn
```

```
  (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4))))
```

多亏有了check，这个版本才和test-+的第一个版本一样简洁，而其展开代码却与第二个版本有着相同的功能。并且现在若想对test-+的行为做出任何改变，也都可以通过改变check来做到。

## 9.3 修复返回值

接下来可以修复test-+以使其返回值可以指示所有测试用例是否都通过了。由于check负责生成最终用来运行测试用例的代码，所以只需改变它来生成可以同时跟踪结果的代码就可以了。

首先可以对report-result做一个小改变，以使其在报告时顺便返回测试用例结果。

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form)
  result)
```

现在report-result返回了它的测试用例结果，故而看起来只需将**PROGN**变成**AND**就可以组合结果了。不幸的是，由于**AND**存在短路行为，即一旦某个测试用例失败了就跳过其余的测试，**AND**在本例中并不能完成你想要的事。另一方面，如果有一个像**AND**那样运作的操作符，同时又没有短路行为，那么就可以用它来代替**PROGN**，从而事情也就完成了。虽然Common Lisp并不提供这样一种构造，但你没有理由不能使用它：自己编写提供这一功能的宏是极其简单的。

暂时把测试用例放在一边，所需要的宏应如下所示，我们称其为combine-results。

```
(combine-results
  (foo)
  (bar)
  (baz))
```

并且它应与下列形式等同：

```
(let ((result t))
  (unless (foo) (setf result nil))
  (unless (bar) (setf result nil))
  (unless (baz) (setf result nil))
  result)
```

编写这个宏唯一麻烦之处在于，需要在展开式中引入一个变量，即前面代码中的result。但正如前所述，在宏展开式中使用一个变量的字面名称会导致宏抽象出现漏洞，因此需要创建唯一的名字，这就需要用到with-gensyms了。可以像下面这样来定义combine-results：

```
(defmacro combine-results (&body forms)
  (with-gensyms (result)
    `(let ((,result t))
      ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
      ,result)))
```

现在可以通过简单地改变展开式用combine-results代替**PROGN**来修复check。

```
(defmacro check (&body forms)
  `(combine-results
    ,@(loop for f in forms collect `(report-result ,f ',f))))
```

使用这个版本的check，test-+就可以输出它的三个测试表达式结果，并返回**T**以说明每一个测试都通过了。[①]

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
T
```

如果改变了一个测试用例而导致其失败[②]，最终的返回值也将变成**NIL**。

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
FAIL ... (= (+ -1 -3) -5)
NIL
```

## 9.4 更好的结果输出

由于只有一个测试函数，所以当前的结果输出是相当清晰的。如果一个特定的测试用例失败了，那么只需在check形式中找到那个测试用例并找出其失败原因即可。但如果编写了大量测试，可能就要以某种方式将它们组织起来，而不是将它们全部塞进一个函数里。例如，假设想要对“*”函数添加一些测试用例，则可以写一个新测试函数。

```
defun test-* ()
  (check
    (= (* 2 2) 4)
    (= (* 3 5) 15)))
```

现在有了两个测试函数，你可能还想用另一个函数来运行所有测试，这也相当简单。

```
(defun test-arithmetic ()
  (combine-results
   (test-+)
   (test-*)))
```

这个函数使用combine-results来代替check，因为test-+和test-*都将分别汇报它们自己的结果。运行test-arithmetic将得到下列结果：

① 如果test-+已经被编译了，这在特定Lisp实现中可能会隐式地发生，你可能需要重新求值test-+的定义以使改变后的check定义影响test-+的行为。另一方面，解释执行的代码通常在每次代码被解释时重新展开宏，从而使宏的重定义的效果立竿见影。

② 你不得不通过改变测试来使其失败，因为你不能改变“+”的行为。

```
CL-USER> (test-arithmetic)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
pass ... (= (* 2 2) 4)
pass ... (= (* 3 5) 15)
T
```

现在假设其中一个测试用例失败了并且需要跟踪该问题。在只有5个测试用例和2个测试函数的情况下，找出失败测试用例的代码并不太困难。但假如有500个测试用例分散在20个函数里，如果测试结果可以显示每个测试用例来自什么函数就非常好了。

由于打印结果的代码集中在report-result函数里，所以需用一种方式来将当前所在测试函数的信息传递给report-result。可以为report-result添加一个形参来传递这一信息，但生成report-result调用的check却并不知道它是从什么函数被调用的，这就意味着还需要改变调用check的方式，向其传递一个参数使其随后传给report-result。

设计动态变量就是用于解决这类问题的。如果创建一个动态变量使得每个测试函数在调用check之前将其函数名绑定于其上，那么report-result就可以无需理会check来使用它了。

第一步是在最上层声明这个变量。

```
(defvar *test-name* nil)
```

现在需要对report-result稍微改动一下，使其在**FORMAT**输出中包括*test-name*。

```
(format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
```

有了这些改变，测试函数将仍然可以工作但将产生下面的输出，因为*test-name*从未被重新绑定：

```
CL-USER> (test-arithmetic)
pass ... NIL: (= (+ 1 2) 3)
pass ... NIL: (= (+ 1 2 3) 6)
pass ... NIL: (= (+ -1 -3) -4)
pass ... NIL: (= (* 2 2) 4)
pass ... NIL: (= (* 3 5) 15)
T
```

为了正确报告其名字，需要改变两个测试函数。

```
(defun test-+ ()
  (let ((*test-name* 'test-+))
    (check
      (= (+ 1 2) 3)
      (= (+ 1 2 3) 6)
      (= (+ -1 -3) -4))))

(defun test-* ()
  (let ((*test-name* 'test-*))
    (check
      (= (* 2 2) 4)
      (= (* 3 5) 15))))
```

现在结果被正确地打上了标签。

```
CL-USER> (test-arithmetic)
pass ... TEST-+: (= (+ 1 2) 3)
pass ... TEST-+: (= (+ 1 2 3) 6)
pass ... TEST-+: (= (+ -1 -3) -4)
pass ... TEST-*: (= (* 2 2) 4)
pass ... TEST-*: (= (* 3 5) 15)
T
```

## 9.5 抽象诞生

在修复测试函数的过程中，你又引入了一点儿新的重复。不但每个函数都需要包含其函数名两次——一次作为**DEFUN**中的名字，另一次是在*test-name*绑定里，而且同样的三行代码模式被重复使用在两个函数中。你可以在认定所有的重复都有害这一思路的指导下继续消除这些重复。但如果更进一步地调查一下导致代码重复的根本原因，你就可以学到关于如何使用宏的重要一课。

这两个函数的定义都以相同的方式开始，原因在于它们都是测试函数。导致重复是因为此时测试函数只做了一半抽象。这种抽象存在于你的头脑中，但在代码里没有办法表达“这是一个测试函数”，除非按照特定的模式来写代码。

不幸的是，部分抽象对于构建软件来说是很糟糕的，因为一个半成品的抽象在代码中就是通过模式来表现的，因此必然会得到大量的重复代码，它们将带有一切影响程序可维护性的不良后果。更糟糕的是，因为这种抽象仅存在于程序员的思路之中，所以实际上无法保证不同的程序员（或者甚至是工作在不同时期的同一个程序员）会以同样的方式来理解这种抽象。为了得到一个完整的抽象，你需要用一种方法来表达“这是一个测试函数”，并且这种方法要能将模式所需的全部代码都生成出来。换句话说，你需要一个宏。

由于试图捕捉的模式是一个**DEFUN**加上一些样板代码，所以需要写一个宏使其展开成**DEFUN**。然后使用该宏（而不是用一个简单的**DEFUN**）去定义测试函数，因此可以将其称为deftest。

```
(defmacro deftest (name parameters &body body)
  `(defun ,name ,parameters
     (let ((*test-name* ',name))
       ,@body)))
```

使用该宏，你可以像下面这样重写test-+：

```
(deftest test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

## 9.6 测试层次体系

由于所建立的测试函数，所以可能会产生一些问题，test-arithmetic应该是一个测试函数吗？事实证明，这件事无关紧要——如果确实用deftest来定义它，那么它对*test-name*

的绑定将在任何结果被汇报之前被test-+和test-*中的绑定所覆盖。

但是现在，想象你有上千个测试用例需要组织在一起。组织的第一层是由诸如test-+和test-*这些能够直接调用check的测试函数所建立起来的，但在有数千个测试用例的情况下，你将需要其他层面的组织方式。诸如test-arithmetic这样的函数可以将相关的测试函数组成测试套件。现在假设某些底层测试函数会被多个测试套件所调用。测试用例很有可能在一个上下文中可以通过而在另一个中失败。如果发生了这种事，你想知道的很可能就不仅仅是哪一个底层测试函数含有这个测试用例那么简单了。

如果用deftest来定义诸如test-arithmetic这样的测试套件函数，并且对其中的*test-name*作一个小改变，就可以用测试用例的“全称”路径来报告结果，就像下面这样：

```
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
```

因为已经抽象了定义测试函数的过程，所以就无需修改测试函数的代码从而改变相关的细节。[1]为了使*test-name*保存一个测试函数名的列表而不只是最近进入的测试函数的名字，你需要将绑定形式

```
(let ((*test-name* ',name))
```

变成

```
(let ((*test-name* (append *test-name* (list ',name))))
```

由于**APPEND**返回一个由其实参元素所构成的新列表，这个版本将把*test-name*绑定到一个含有追加其新的名字到结尾处的*test-name*的旧内容的列表。[2]当每一个测试函数返回时，*test-name*原有的值将被恢复。

现在你可以用deftest代替**DEFUN**来重新定义test-arithmetic。

```
(deftest test-arithmetic ()
  (combine-results
   (test-+)
   (test-*)))
```

现在的结果明确地显示了你是怎样到达每一个测试表达式的。

```
CL-USER> (test-arithmetic)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

随着测试套件的增长，你可以添加新的测试函数层次。只要它们用deftest来定义，结果就会正确地输出。例如定义

---

① 再强调一次，如果测试函数已经被编译了，那么在改变宏以后你将需要重新编译它们。

② 你将在第12章里看到，用**APPEND**在列表结尾处追加元素并不是构造一个列表的最有效方式。但目前这种方法是有效的，只要测试的层次不是很深就可以了。并且如果这成为问题，所有你需要做的就是改变deftest的定义。

```
(deftest test-math ()
  (test-arithmetic))
```

将产生这样的结果：

```
CL-USER> (test-math)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

## 9.7 总结

你可以继续为这个测试框架添加更多特性。但作为一个以最小成本编写测试并可以在REPL轻松运行的框架来说，这已经是一个很好的开始了。这里给出完整的代码，全部只有26行：

```
(defvar *test-name* nil)

(defmacro deftest (name parameters &body body)
  "Define a test function. Within a test function we can call
   other test functions or use 'check' to run individual test
   cases."
  `(defun ,name ,parameters
     (let ((*test-name* (append *test-name* (list ',name))))
       ,@body)))

(defmacro check (&body forms)
  "Run each expression in 'forms' as a test case."
  `(combine-results
     ,@(loop for f in forms collect `(report-result ,f ',f))))

(defmacro combine-results (&body forms)
  "Combine the results (as booleans) of evaluating 'forms' in order."
  (with-gensyms (result)
    `(let ((,result t))
       ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
       ,result)))

(defun report-result (result form)
  "Report the results of a single test case. Called by 'check'."
  (format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
  result)
```

值得回顾的是，你能走到这一步是因为它显示了Lisp编程的一般方式。

你从定义一个解决问题的简单版本开始——怎样求值一些布尔表达式并找出它们是否全部返回真。将它们用**AND**连在一起可以工作并且在句法上很简洁，却无法满足以更好的结果输出的需要。因此你写了一些真正简单的代码，其中充满了代码重复以及在用你想要的方式报告结果时容易出错的用法。

下一步是查看你是否可以将第二个版本重构得跟前面版本一样简洁。你从一个标准的重构技术开始，将某些代码放进函数report-result中。不幸的是，你发现使用report-result会产生冗长和易错的代码，由于你不得不将测试表达式传递两次，一次作为值而另一次作为引用的数据。因此，你写了check宏来自动地正确调用report-result的细节。

在编写check的时候，你认识到在生成代码的同时，也可以让对check的单一调用生成对report-result的多个调用，从而得到了一个和最初**AND**版本一样简洁的test-+函数。

在那一点上你调整了check的API，从而你可以开始看到check内部的工作方式。下一个任务是修正check，使其生成的代码可以返回一个用来指示所有测试用例是否均已通过的布尔值。接下来你没有立即继续玩弄check，而是停下来沉迷于一个设计巧妙的微型语言。你梦想假如有一个非短路的**AND**构造就好了，这样修复check就会非常简单了。回到现实以后，你认识到不存在这样构造，但你可以用几行程序写一个出来。在写出了combine-results以后，对check的修复确实简单多了。

在那一点上唯一剩下的就是对你报告测试结果的方式做一些进一步的改进。一旦你开始修改测试函数，你就会认识到这些函数代表了特殊的一类值得有其自己的抽象方式的函数。因此你写出了deftest来抽象代码模式，使一个正常函数变成了一个测试函数。

借助deftest所提供的在测试定义和底层机制之间的抽象障碍，你可以无需修改测试函数而改进汇报结果的方式。

至此，你已经掌握了函数、变量和宏的基础知识，有了一点儿使用它们的实践经验，可以开始探索由函数和数据类型组成的Common Lisp的丰富的标准库了。