第19章

超越异常处理：状况和再启动

19

Lisp的状况系统（condition system）是它最伟大的特性之一 。它与Java、Python和C++中的异常处理系统有着相同的目标但更加灵活。事实上，它的灵活性扩展到了错误处理之外——“状况”相比“异常”更具一般性，因为状况可以代表程序执行过程中的任何事件，程序调用栈中不同层次的代码都可能对这些事件感兴趣。例如，在19.6节里你将看到，状况可用来输出警告而不会中断产生警告的那些代码的执行，并允许调用栈中更高层的代码来控制是否打印警告信息。不过，一开始我还是集中讨论错误处理。

状况系统相比异常系统的更灵活之处在于，它没有明确划分产生错误[①]的代码和处理错误[②]的代码，而是将责任分拆成三个部分：产生（signaling）状况，处理（handling）它以及再启动（restarting）。在本章里，我将描述你在假想的分析日志文件应用程序中如何使用状况系统。你将看到如何使用状况系统，允许底层函数在解析日志文件时检测问题并产生一个错误，允许中层代码提供几种可能的方式从这样一个错误中恢复，以及允许该应用程序的最上层代码制定一种方针来选择所使用的恢复策略。

一开始，我将介绍一些术语。错误（error），我采用墨菲法则定义该术语。如果某件事可能出错，那么它终将出错：一个程序需要读取的文件会丢失，一个需要写入的磁盘会满，正在连接的服务器会崩溃，或者网络会断开。如果发生了任何这些情况，它可能使一些正在做你所想的事情的代码停止工作。但这里没有bug，代码中没有可以修复的地方，以使那些不存在的文件存在或是令磁盘不再是满的。尽管如此，如果程序的其余部分正在依赖于这些正打算进行的操作，那么你最好以某种方式处理这些错误，否则就会引入bug。因此，错误并不是由bug导致的，但忽视处理错误就总是可能作一个bug。

那么，处理错误究竟意味着什么呢？在一个编写良好的程序中，每个函数都是一个隐藏了其内部工作的黑箱。程序随后由分层的函数构建而成：高层次的函数是构建在相对低层次的函数之上的，诸如此类。功能的层次关系在运行期以调用栈的形式显示其自身：如果`high`调用了`medium`，`medium`调用了`low`，那么当控制流在`low`中时，它也仍然在`medium`和`high`中，也就是

① 在Java/Python的术语中称为抛出（throw）或提升（raise）一个异常。

② 在Java/Python的术语中称为捕捉（catch）该异常。

说它们仍然在调用栈中。

由于每个函数都是一个黑箱，函数边界刚好是处理错误的最佳场所。每个函数（比如说low）都有一个需要完成的任务。其直接调用者（这里是medium）的工作是统计它的调用次数。不过，一个使它不能正常工作的错误将给它的所有调用者带来风险：medium调用low是因为它需要的工作low能够完成，如果low不能完成该工作，那么medium就有问题了。这意味着medium的调用者high也有问题了，诸如此类，一直沿着调用栈到达程序的最顶端。另一方面，由于每个函数都是一个黑箱，如果调用栈中的任何函数可能以某种方式完成其工作而无需关注底层错误，那么在它之上的程序就没有必要知道曾经出现的问题——那些函数所关心的只是它们所调用的函数无论如何都要完成期待的工作。

在多数语言里，错误的处理方式都是从一个失败的函数返回并给调用者一个机会，要么修复该错误要么让其自身失败。某些语言使用了正常的函数返回机制，而带有异常的语言可以通过抛出一个异常来返回控制。使用异常比使用正常函数返回是巨大的进步，但两种模式有一个共同的缺点：在搜索一个可恢复的函数时，栈被展开了，这意味着可以恢复错误的代码无法在错误实际发生时底层代码所在的上下文中进行操作。

想想函数high、medium和low的假想调用链。如果low失败而medium无法恢复，那么决定权将在high手中。在high处理错误时，它必须要么在无须得到任何来自medium的帮助的情况下完成其工作，要么以某种方式改变一些，使对medium的调用能够发挥作用，然后再次调用它。前一个选项在理论上是清晰的，但可能导致大量的额外代码，那将需要一个完整的对于medium的工作的额外实现。并且当栈进一步展开时，还有更多工作需要重做。后一个选项（修补环境并重试）比较棘手，这要求high必须改变当前环境的状况，使得下一个对medium的调用不会导致low中的错误，这需要同时对medium和low的内部工作原理有所了解，这并不恰当，与每个函数都是一个黑箱的理念相违背。

19.1　Lisp的处理方式

Common Lisp的错误处理系统提供了一种跳出这一难题的方式，它将实际从错误中恢复的代码和决定如何恢复的代码分开。这样，你可以将恢复代码放在底层函数中而无须决定实际使用何种特定的恢复策略，将决策留给高层函数中的代码。

为了了解其工作原理，我们假设你正在编写一个读取某种文本日志文件的应用程序，例如读取一个Web服务器的日志。在应用程序的某处，你将有一个函数用来解析单独的日志项。我们假设你将编写一个函数parse-log-entry，它将接受一个含有单个日志项文本的字符串，并假设可以返回代表该项的一个log-entry对象。该函数将从函数parse-log-file中调用，后者读取一个完整的日志文件并返回代表该文件中所有日志项的对象列表。

为了保持简单性，parse-log-entry函数不需要解析不正确的格式化项。不过，它可以检测到有问题的输入。当它检测到不对的输入后应当做什么呢？在C中会返回一个特殊值来指示这里出了问题，在Java或Python中会抛出一个异常。而在Common Lisp中会产生一个状况。

19.2 状况

一个状况（condition）是一个对象，它所属的类代表了该状况的一般性质，它的实例数据则带有导致该状况产生的特定情形细节的信息。[1]在这个假想的日志分析程序里，你可以定义一个状况类malformed-log-entry-error，函数parse-log-entry将在给定数据无法解析时产生该状况。

状况类使用**DEFINE-CONDITION**宏来定义，它本质上就是**DEFCLASS**，除了使用**DEFINE-CONDITION**所定义的类的默认基类是**CONDITION**而非**STANDARD-OBJECT**。槽以相同的方式来指定，状况类可以单一和多重地继承自其他源自**CONDITION**的类。但出于历史原因，状况类不要求是**STANDARD-OBJECT**的实例，因此一些你和**DEFCLASS**所定义类一起使用的函数不要求可以工作在状况对象上。特别地，一个状况的槽不能使用**SLOT-VALUE**来访问，必须为任何你打算使用其值的槽分别指定:reader选项或:accessor选项。同样，新的状况对象使用**MAKE-CONDITION**而非**MAKE-INSTANCE**来创建。**MAKE-CONDITION**基于其被传递的:initargs来初始化新状况的槽，但没有办法以等价于**INITIALIZE-INSTANCE**的方式更进一步地定制一个状况的初始化过程。[2]

当把状况系统用于错误处理时，你应当将状况定义成**ERROR**的子类，后者是**CONDITION**的一个子类。这样，你可以像下面这样定义malformed-log-entry-error，其中带有一个槽用来保存传递给parse-log-entry的参数：

```
(define-condition malformed-log-entry-error (error)
  ((text :initarg :text :reader text)))
```

19.3 状况处理器

在parse-log-entry中，当无法解析日志项时将产生一个malformed-log-entry-error。函数**ERROR**用来报错，它将调用底层函数**SIGNAL**并在状况未处理时进入调试器。有两种方式来调用**ERROR**：传给它一个已经实例化的状况对象，或者传给它该状况类的名字以及需要用来构造新状况的初始化参数，然后它将实例化该状况。前者对于重新产生一个已有的状况对象偶尔会有用，而后者更普遍。因此，你可以像下面这样编写parse-log-entry，其中隐藏了实际解析日志项的细节：

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (error 'malformed-log-entry-error :text text)))
```

① 在这个意义上，除了并非所有状况都代表一个错误或异常的（exceptional）情形，一个状况和Java或Python中的异常很像。

② 在一些Common Lisp实现中，状况被定义为**STANDARD-OBJECT**的子类。在这种情况下，**SLOT-VALUE**、**MAKE-INSTANCE**和**INITIALIZE-INSTANCE**将可以工作，但依赖于它们将是不可移植的。

当产生错误时，实际发生的事情取决于调用栈中parse-log-entry之上的代码。为了避免进入调试器，必须在导致调用parse-log-entry的某个函数中建立一个状况处理器（condition handler）。当产生状况时，信号机制会查看活跃状况处理器的列表，并基于状况的类来寻找可以处理所产生的状况的处理器。每个状况处理器由一个代表它所能处理的状况类型的类型说明符和一个接受单个状况参数的函数所构成。在任何给定时刻可能有多个活跃的状况处理器建立在调用栈的不同层次上。当一个状况产生时，信号机制会查找最新建立的类型说明符与当前所产生状况相兼容的处理器并调用它的函数，同时传递状况对象给该函数。

处理器函数随后可以选择是否处理该状况。该函数可以通过简单的正常返回来放弃处理该状况。在这种情况下，控制将返回到**SIGNAL**函数，**SIGNAL**函数随后继续搜索下一个带有兼容类型说明符的最新建立的处理器。为了处理该状况，该函数必须通过一个非本地退出（nonlocal exit）将控制传递到**SIGNAL**之外。在下一节里，你将看到一个处理器是怎样选择传递控制的位置的。尽管如此，许多状况处理器简单地想要将栈展开到它们被建立的位置上并随后运行一些代码。宏**HANDLER-CASE**可以建立这种类型的状况处理器。一个**HANDLER-CASE**的基本形式如下所示：

```
(handler-case expression
  error-clause*)
```

其中每一个error-clause均为下列形式：

```
(condition-type ([var]) code)
```

如果*expression*正常返回，那么其值将被**HANDLER-CASE**返回。**HANDLER-CASE**的主体必须是单一表达式，你可以使用**PROGN**将几个表达式组合成单一形式。不过，如果该表达式产生了一个状况，并且其实例属于任何错误子句中指定的状况类型之一，那么将执行相应错误子句中的代码，且其值将由**HANDLER-CASE**返回。如果形参*var*被包含，在执行处理器代码时，它将成为保存状况对象的变量名。如果代码不需要访问状况对象，则可以省略这个变量名。

例如，一种在parse-log-entry的调用者parse-log-file中处理前者所产生的malformed-log-entry-error的方式，将是跳过有问题的项。在下面的函数中，**HANDLER-CASE**表达式要么返回由parse-log-entry返回的值，要么在产生malformed-log-entry-error时返回**NIL**。（**LOOP**子句collect it中的it是另一个**LOOP**关键字，它指向最新求值条件测试的值，在这种情况下是entry的值。）

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
       for entry = (handler-case (parse-log-entry text)
                     (malformed-log-entry-error () nil))
       when entry collect it)))
```

当parse-log-entry正常返回时，它的值将赋值给变量entry中并被**LOOP**收集。但如果parse-log-entry产生了一个malformed-log-entry-error，那么错误子句将返回**NIL**，从而不会被收集。

JAVA风格的异常处理

HANDLER-CASE是Common Lisp中最接近Java或Python风格的异常处理。在Java中可能写成这样：

```
try {
  doStuff();
  doMoreStuff();
} catch (SomeException se) {
  recover(se);
}
```

或在Python中写成这样：

```
try:
  doStuff()
  doMoreStuff()
except SomeException, se:
  recover(se)
```

而在Common Lisp中需要写成这样：

```
(handler-case
    (progn
      (do-stuff)
      (do-more-stuff))
  (some-exception (se) (recover se)))
```

这个版本的parse-log-file有一个严重的缺陷：它做得太多了。顾名思义，parse-log-file的任务是解析文件并产生log-entry对象的列表。如果它做不到这一点，决定采用何种替代方案就不是它的职责所在。假如你想在一个想要告诉用户日志文件被破坏了的应用中使用parse-log-file，或是想要从有问题的项中通过修复并重新解析它们来恢复，又该怎样做呢？或者一个应用对于跳过它们是没问题的，但只有当发现特定数量的受损日志项后才需要特别处理。

你可以通过将HANDLER-CASE移动到一个更高层的函数中来修复该问题。不过，这样你就没有办法实现当前跳过个别项的策略了——当错误发生时，栈将被一路展开到更高层的函数上，连同日志文件的解析本身一并丢弃了。你所需要的是一种方式，能提供当前的恢复策略而不是总要被用到。

19.4 再启动

状况系统可以让你将错误处理代码拆分成两部分。将那些实际从错误中恢复的代码放在再启动（restart）中，状况处理器随后会通过调用一个适当的再启动来处理状况。可以将再启动代码放在中层或底层的函数里，例如parse-log-file或parse-log-entry，而将状况处理器移到应用程序的更上层。

为了改变parse-log-entry，从而使其建立一个再启动而非状况处理器，可以将**HANDLER-CASE**改成**RESTART-CASE**。除了再启动的名字可以只是普通的名字而无需是状况类型的名字外，**RESTART-CASE**的形式与**HANDLER-CASE**很相似。一般而言，一个再启动名应当描述了再启动所产生的行为。在parse-log-file中，你可以根据其行为将这个再启动称为skip-log-entry。该函数的新版本如下所示：

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
       for entry = (restart-case (parse-log-entry text)
                     (skip-log-entry () nil))
       when entry collect it)))
```

如果在一个含有受损日志项的日志文件上调用该版本的parse-log-file，它将不会直接处理错误，最终会转到调试器中。不过，在调试器所提供的几个再启动选项中会有一个称为skip-log-entry的选项，如果你选择了它，将导致parse-log-file继续其之前的操作。为了避免进入调试器，你可以建立一个状况处理器来自动地调用skip-log-entry再启动。

建立一个再启动而不是让parse-log-file直接处理错误的好处在于，它使得parse-log-file可以用在更多情形里了。调用parse-log-file的更高层代码不需要调用skip-log-entry再启动。它可以选择在更高层进行错误处理。或者，如同我将在下一节讲述的那样，你可以为parse-log-entry添加再启动来提供其他的恢复策略，随后状况处理器可以选择它们想要使用的策略。

在我开始谈论这些之前，你需要知道如何设置一个会调用skip-log-entry再启动的状况处理器。你可以在通向parse-log-file的函数调用链的任何位置上设置这个处理器。这可能是应用程序中很上层的某个位置，不一定在parse-log-file的直接调用者中。例如，假设应用程序的主入口点是函数log-analyzer，它查找一组日志并使用函数analyze-log来分析它们，后者最终导致了对parse-log-file的调用。在没有任何错误处理的情况下，它可能像这样：

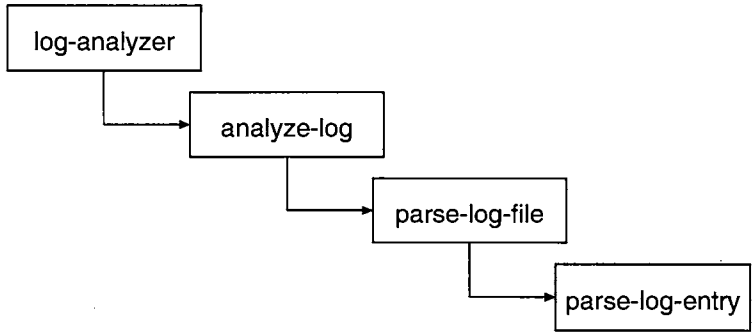
```
(defun log-analyzer ()
  (dolist (log (find-all-logs))
    (analyze-log log)))
```

analyze-log的任务是直接或间接地调用parse-log-file，然后再对返回的日志项列表做一些事情。它的一个极其简化的版本可能像这样：

```
(defun analyze-log (log)
  (dolist (entry (parse-log-file log))
    (analyze-entry entry)))
```

根据推测，其中函数analyze-entry将会实际取出那些你想要从每个日志项中得到的信息，并存储在其他地方。

这样，从最上层的函数log-analyzer到实际产生错误的parse-log-entry的路径将如下所示。



假设你总是想要跳过有问题的日志项，那么可以改变这个函数来建立一个状况处理器，为你调用skip-log-entry。不过，你不能使用**HANDLER-CASE**来建立这个状况处理器，因为那样的话栈会回退到**HANDLER-CASE**所在的函数里。你需要使用更底层的宏**HANDLER-BIND**。**HANDLER-BIND**的基本形式如下所示：

```
(handler-bind (binding*) form*)
```

其中的每个绑定都是由状况类型和一个单参数处理函数所组成的列表。在处理器绑定之后，**HANDLER-BIND**的主体可以包含任意数量的形式。与**HANDLER-CASE**的处理器代码有所不同的是，这里的处理器代码必须是一个函数对象，并且它必须只接受单一参数。**HANDLER-BIND**和**HANDLER-CASE**之间的一个更大的区别在于，由**HANDLER-BIND**所绑定的处理器函数必须在不回退栈的情况下运行——当调用该函数时，控制流仍然在对parse-log-entry的调用中。对**INVOKE-RESTART**的调用将查找并调用最近绑定的带有给定名字的再启动。因此你可以像下面这样给log-analyzer添加一个处理器，使其可以调用由parse-log-file所建立的再启动：[①]

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (invoke-restart 'skip-log-entry))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

在这个**HANDLER-BIND**中，处理器函数是一个调用了skip-log-entry再启动的匿名函数。你也可以定义一个命名函数来做相同的事并绑定该函数。事实上，在定义再启动时有个常用的实践技巧是定义一个函数，它与再启动具有相同的名字并接受单一状况参数来调用对应的再启动。这样的函数称为再启动函数（restart function）。你可以像下面这样为skip-log-entry定义一个再启动函数：

```
(defun skip-log-entry (c)
  (invoke-restart 'skip-log-entry))
```

然后将log-analyzer的定义修改成下面这样：

① 编译器可能会抱怨函数的参数从未被使用。你可以通过添加一个声明(declare (ignore c))作为**LAMBDA**形式体中的第一个表达式来消除这类警告。

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error #'skip-log-entry))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

如同代码中所写，skip-log-entry再启动函数假设已经建立了skip-log-entry再启动。如果malformed-log-entry-error被来自log-analyzer的代码抛出，却没有建立skip-log-entry再启动，那么对**INVOKE-RESTART**的调用将在无法找到skip-log-entry再启动时产生**CONTROL-ERROR**报错。如果你想要允许malformed-log-entry-error可以被没有建立skip-log-entry再启动的代码抛出，那么可以将skip-log-entry函数修改成下面这样：

```
(defun skip-log-entry (c)
  (let ((restart (find-restart 'skip-log-entry)))
    (when restart (invoke-restart restart))))
```

FIND-RESTART查找一个给定名字的再启动，并在找到时返回一个代表该再启动的对象，否则返回**NIL**。你可以通过将再启动对象传递给**INVOKE-RESTART**来调用该再启动。这样，当skip-log-entry被**HANDLER-BIND**绑定时，它将在存在一个再启动时通过调用skip-log-entry再启动来处理该状况，而在其他情况下正常返回，从而给那些绑定在栈的更高层的其他状况处理器提供机会来处理该状况。

19.5　提供多个再启动

由于再启动必须显式调用才有效果，因此你可以定义多个再启动，让它们中的每一个分别提供不同的恢复策略。如同我早先提到的，并非所有的日志解析应用都需要跳过那些有问题的项。一些应用可能想要parse-log-file包含一个特殊类型的对象来表示log-entry对象列表中有问题的日志项；其他应用可能有一些方式来修复有问题的项，并可能需要一种方式来将修复后的项传递回parse-log-entry。

为了允许这些更复杂的恢复机制，再启动可以接受任意参数，它们被传递给对**INVOKE-RESTART**的调用。通过为parse-log-entry增加两个再启动，其中每个都只接受单个参数，你可以同时支持我刚刚提到的两种恢复策略。一个简单地返回传递给它的parse-log-entry的返回值，而另一个则试图在最初的日志项上就地解析其参数。

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (restart-case (error 'malformed-log-entry-error :text text)
      (use-value (value) value)
      (reparse-entry (fixed-text) (parse-log-entry fixed-text)))))
```

USE-VALUE是这类再启动的标准名字。Common Lisp为**USE-VALUE**定义了一个再启动函数，类似于你之前为skip-log-entry定义的再启动函数。因此，如果你想要改变处理有问题项的策略，在遇到这类问题项时创建一个malformed-log-entry的实例，那么就可以像下面这样修改log-analyzer（假设存在一个带有:text初始化参数的malformed-log-entry类）：

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (use-value
                       (make-instance 'malformed-log-entry :text (text c))))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```


你还可以将这些新的再启动放在parse-log-file而不是parse-log-entry中。不过，你通常会想把再启动放在尽可能最底层的代码里。尽管如此，将skip-log-entry再启动移到parse-log-entry中并不合适，因为这会导致parse-log-entry有时正常返回一个**NIL**，而这正是你极力予以避免的。同时，基于状况处理器可以通过在**NIL**上调用use-value再启动而获得同样效果这一理论，直接去掉skip-log-entry再启动也不是个好主意，因为这要求状况处理器理解parse-log-file的工作原理。正如其名字所显示的，skip-log-entry是一个正确抽象了的日志解析API的组成部分。

19.6 状况的其他用法

虽然状况系统主要用于错误处理，但它们还可以用于其他目的，你可以使用状况、状况处理器和再启动在底层和上层代码之间构建多种协议。理解状况的潜在用途的关键在于，要理解仅仅抛出一个状况并不会改变程序的控制流。

基本的信号函数**SIGNAL**实现了搜索适用的状况处理器并调用其处理器函数的机制。一个处理器通过正常返回来拒绝处理状况的原因在于，对处理器函数的调用只是一个正规函数调用——当处理器返回时，控制被传递回**SIGNAL**，**SIGNAL**随后继续查询另一个较近绑定的可以处理该状况的处理器。如果**SIGNAL**在状况处理之前找不到其他的状况处理器，那么它也会正常返回。

你曾经使用的**ERROR**函数会调用**SIGNAL**。如果错误被一个通过**HANDLER-CASE**传递控制的状况处理器或通过调用一个再启动所处理，那么这个对**SIGNAL**的调用将不再返回。但如果**SIGNAL**返回了，那么**ERROR**将通过调用保存在***DEBUGGER-HOOK***中的函数来启动调试器。这样，一个对**ERROR**的调用永远不会正常返回。状况必须要么被一个状况处理器所处理，要么在调试器中被处理。

另一个状况信号函数**WARN**提供了构建在状况系统之上的不同类型协议的示例。和**ERROR**一样，**WARN**调用**SIGNAL**来产生一个状况。但是如果**SIGNAL**返回了，**WARN**并不会调用调试器——它将状况打印到***ERROR-OUTPUT***中并返回**NIL**，从而交给它的调用者来处理。**WARN**也会在**SIGNAL**调用的外围建立一个再启动**MUFFLE-WARNING**，从而允许一个状况处理器令**WARN**直接返回而不打印任何东西。再启动函数**MUFFLE-WARNING**可以查找并调用与其同名的再启动，并在不存在这样的再启动时产生一个**CONTROL-ERROR**。当然，一个通过**WARN**产生的状况也可以用其他方式来处理——一个状况处理器可以像处理真正的错误一样来处理它，从而将一个警告“提升”为一个错误。

举个例子，在日志分析应用里，如果存在某些情况使得一个日志项稍不正常但仍可解析，那

么你可以编写parse-log-entry来使其继续解析这些稍有问题的日志项，但同时用WARN产生一个状况。然后更大的应用可以选择打印这些警告、隐藏这些警告，或是将这些警告当作错误来处理，并使用与来自malformed-log-entry-error相同的方式进行恢复。

第三个报错函数CERROR提供了另一种协议。和ERROR一样，CERROR将在状况没被处理时就会转到调试器。但和WARN一样，它在产生状况之前会建立一个再启动。这个再启动是CONTINUE，它可以使CERROR正常返回——如果再启动由一个状况处理器所调用的话，它将确保始终不转到调试器。否则，可以在转到调试器以后使用再启动，立即恢复到CERROR调用之后的计算状态。函数CONTINUE查找并在CONTINUE再启动可用时调用它，否则返回NIL。

你也可以在SIGNAL之上构建自己的协议——无论底层代码需要何种方式来与调用栈中的上层代码沟通信息，状况机制都可以合理使用。但对于多数目标来说，标准的错误和警告协议应该足够了。

你将在后续的实践章节里用到状况系统，既可以用于正常的错误处理，也可以像第25章那样帮助处理ID3文件解析过程中一些棘手的边界情况。遗憾的是，编程教材总是过于轻视错误处理。正确的错误处理，或者在这方面的欠缺，往往是阐述性代码和坚不可摧的产品级代码之间最大的区别。后者的难点在于，需要进行大量的关于软件本身而不是任何特定编程语言构造细节的思考。这就是说，如果你的目标是编写一个那样的软件，那么你将发现Common Lisp状况系统是用于编写健壮代码的极佳工具，并且它可以完美地融合到Common Lisp的增量式开发风格中。

编写健壮的软件

关于编写健壮的软件方面的信息，你可以从查阅由Glenford J.Meyers编写的*Software Reliability*（John Wiley & Sons，1976）开始。Bertrand Meyer关于Design By Contract的著作也提供了一种思考软件正确性的有用方式，可参见他的*Object-Oriented Software Construction*（Prentice Hall,1997）一书的第11章和第12章。不过要记住，Bertrand Meyer是Eiffel的发明者，Eiffel是一种静态类型的受约束的Algol/Ada系语言。尽管他在面向对象和软件可靠性方面有许多聪明的见解，但在他的编程观点和Lisp编程方式之间仍然存在着一条鸿沟。最后，关于围绕构建失效容忍系统的更大问题的一个绝佳综述，可以参见Jim Gray和Andreas Reuter所编写的经典的*Transaction Processing: Concepts and Techniques*（Morgan Kaufmann, 1993）一书的第3章。

在下一章里，我将简单概述一下你尚未有机会使用或者说至少还没有直接用到的25个特殊操作符。