Próximo:Símbolos de erro, Anterior:Processamento de Erros, Acima:Erros [Conteúdo][Índice]

11.7.3.3 Escrevendo Código para Lidar com Erros

O efeito usual de sinalizar um erro é encerrar o comando que está sendo executado e retornar imediatamente ao loop de comandos do editor Emacs. Você pode arranjar para interceptar erros que ocorrem em uma parte de seu programa estabelecendo um manipulador de erros, com o formulário especial `condition-case`. Um exemplo simples se parece com isso:

```
(condição-caso nula
    (excluir nome de arquivo do arquivo)
  (erro nulo))
```

Isso exclui o arquivo chamado *filename* , capturando qualquer erro e retornando `nil`se ocorrer um erro. (Você pode usar a macro `ignore-errors`para um caso simples como este; veja abaixo.)

A `condition-case`construção é frequentemente usada para interceptar erros previsíveis, como falha ao abrir um arquivo em uma chamada para `insert-file-contents`. Também é usado para interceptar erros totalmente imprevisíveis, como quando o programa avalia uma expressão lida do usuário.

O segundo argumento de `condition-case`é chamado de *forma protegida* . (No exemplo acima, o formulário protegido é uma chamada para `delete-file`.) Os manipuladores de erro entram em vigor quando este formulário inicia a execução e são desativados quando este formulário retorna. Eles permanecem em vigor por todo o tempo intermediário. Em particular, eles estão em vigor durante a execução de funções chamadas por este formulário, em suas sub-rotinas, e assim por diante. Isso é uma coisa boa, pois, estritamente falando, os erros podem ser sinalizados apenas por primitivas Lisp (incluindo `signale` `error`) chamadas pelo formulário protegido, não pelo próprio formulário protegido.

Os argumentos após o formulário protegido são manipuladores. Cada manipulador lista um ou mais *nomes de condição* (que são símbolos) para especificar quais erros serão tratados. O símbolo de erro especificado quando um erro é sinalizado também define uma lista de nomes de condições. Um manipulador se aplica a um erro se eles tiverem algum nome de condição em comum. No exemplo acima, há um manipulador que especifica um nome de condição, `error`, que abrange todos os erros.

A busca por um manipulador aplicável verifica todos os manipuladores estabelecidos, começando pelo estabelecido mais recentemente. Assim, se dois formulários aninhados `condition-case`oferecem para lidar com o mesmo erro, o interno dos dois consegue lidar com isso.

Se um erro for tratado por algum `condition-case`formulário, isso normalmente impede que o depurador seja executado, mesmo que `debug-on-error`diga que esse erro deve invocar o depurador.

Se você quiser ser capaz de depurar erros que são capturados por um `condition-case`, defina a variável `debug-on-signal`como um `nil`valor não. Você também pode especificar que um manipulador específico deve permitir que o depurador seja executado primeiro, escrevendo `debug`entre as condições, assim:

```
(condição-caso nula
    (excluir nome de arquivo do arquivo)
  ((erro de depuração) nil))
```

O efeito de `debug`aqui é apenas impedir `condition-case`a supressão da chamada para o depurador. Qualquer dado erro invocará o depurador somente se `debug-on-error`e os outros mecanismos de filtragem usuais disserem que deveria. Consulte Depuração de erros .

Macro: condition-case-unless-debug *var protected-form handlers…*

A macro `condition-case-unless-debug`fornece outra maneira de lidar com a depuração de tais formulários. Ele se comporta exatamente como `condition-case`, a menos que a variável `debug-on-error`não seja `nil`, caso em que não trata nenhum erro.

Uma vez que o Emacs decide que um determinado manipulador trata o erro, ele retorna o controle para esse manipulador. Para fazer isso, o Emacs desvincula todas as associações de variáveis feitas por construções de associação que estão sendo encerradas e executa as limpezas de todos os `unwind-protect`formulários que estão sendo encerrados. Quando o controle chega ao manipulador, o corpo do manipulador é executado normalmente.

Após a execução do corpo do manipulador, a execução retorna do `condition-case`formulário. Como o formulário protegido é encerrado completamente antes da execução do manipulador, o manipulador não pode retomar a execução no ponto do erro, nem pode examinar as associações de variáveis feitas no formulário protegido. Tudo o que pode fazer é limpar e prosseguir.

A sinalização e o tratamento de erros têm alguma semelhança com `throw`e `catch`(veja Catch and Throw), mas são instalações totalmente separadas. Um erro não pode ser capturado por a `catch`, e a `throw`não pode ser tratado por um manipulador de erros (embora usar `throw`quando não houver sinais adequados `catch`indica um erro que pode ser tratado).

Forma especial: *manipuladores de forma protegida var* condição-caso …

Este formulário especial estabelece os manipuladores de erros em torno da execução de *protected-form* . Se *o formulário protegido* for executado sem erros, o valor retornado se tornará o valor do `condition-case`formulário; neste caso, o `condition-case`não tem efeito. O `condition-case`formulário faz a diferença quando ocorre um erro durante *o formato protected* .

Cada um dos *manipuladores* é uma lista do formulário . Aqui *condições* é um nome de condição de erro a ser tratado ou uma lista de nomes de condição (que pode incluir para permitir que o depurador seja executado antes do manipulador). Um nome de condição corresponde a qualquer condição. *body* é uma ou mais expressões Lisp a serem executadas quando este manipulador trata um erro. Aqui estão alguns exemplos de manipuladores: `(`*`conditions body`*`…)`debugt

```
(erro nulo)

(arith-error (mensagem "Divisão por zero"))

((arith-erro arquivo-erro)
 (mensagem
  "Ou divisão por zero ou falha ao abrir um arquivo"))
```

Cada erro que ocorre tem um *símbolo de erro* que descreve que tipo de erro é e que descreve também uma lista de nomes de condição (consulte Símbolos de erro). O Emacs procura em todos os `condition-case`formulários ativos um manipulador que especifica um ou mais desses nomes de condição; a correspondência mais interna `condition-case`trata o erro. Dentro deste `condition-case`, o primeiro manipulador aplicável trata o erro.

Após executar o corpo do manipulador, o `condition-case` retorna normalmente, usando o valor do último formulário no corpo do manipulador como valor geral.

O argumento *var* é uma variável. `condition-case`não vincula esta variável ao executar o *formulário protegido* , apenas quando trata um erro. Nesse momento, ele vincula *var* localmente a uma *descrição de erro* , que é uma lista que fornece os detalhes do erro. A descrição do erro tem o formato . O manipulador pode consultar essa lista para decidir o que fazer. Por exemplo, se o erro for por falha na abertura de um arquivo, o nome do arquivo será o segundo elemento dos *dados* — o terceiro elemento da descrição do erro. (`(`*`error-symbol . data`*`)`

Se *var* for `nil`, isso significa que nenhuma variável está vinculada. Então o símbolo de erro e os dados associados não estão disponíveis para o manipulador.

Às vezes é necessário lançar novamente um sinal capturado por `condition-case`, para que algum manipulador de nível externo capture. Veja como fazer isso:

```
(sinal (carro err) (cdr err))
```

onde `err`é a variável de descrição do erro, o primeiro argumento para `condition-case`cuja condição de erro você deseja relançar. Consulte Definição de sinal .

Função: descritor de erro string de mensagem de *erro*

Esta função retorna a string da mensagem de erro para um determinado descritor de erro. É útil se você deseja lidar com um erro imprimindo a mensagem de erro usual para esse erro. Consulte Definição de sinal .

Aqui está um exemplo de uso `condition-case`para lidar com o erro resultante da divisão por zero. O manipulador exibe a mensagem de erro (mas sem um bipe) e retorna um número muito grande.

```
(defun safe-divide (divisor de dividendos)
  (erro de condição-caso
      ;; Forma protegida.
      (/ divisor de dividendos)
    ;; O manipulador.
    (arith-error ; Condition.
     ;; Exibe a mensagem usual para este erro.
     (mensagem "%s" (erro de string de mensagem de erro))
     1000000)))
⇒ divisão segura


(divisão segura 5 0)
     -| Erro aritmético: (erro aritmético)
⇒ 1.000.000
```

O manipulador especifica o nome da condição `arith-error`para que trate apenas de erros de divisão por zero. Outros tipos de erros não serão tratados (por este `condition-case`). Portanto:

```
(divisão segura nil 3)
     erro→ tipo de argumento errado: número-ou-marcador-p, nil
```

Aqui está um `condition-case`que captura todos os tipos de erros, incluindo aqueles de `error`:

```
(setq baz 34)
     ⇒ 34


(erro de condição-caso
    (se (eq baz 35)
        t
      ;; Esta é uma chamada para a função error.
      (erro "Rats! A variável %s era %s, não 35" 'baz baz))
  ;; Este é o manipulador; não é uma forma.
  (erro (princ (formato "O erro foi: %s" err)))
        2))
-| O erro foi: (erro "Rats! A variável baz era 34, não 35")
⇒ 2
```

Macro: corpo de ignorar-erros …

Essa construção executa *body* , ignorando quaisquer erros que ocorram durante sua execução. Se a execução for sem erro, `ignore-errors`retorna o valor do último formulário em *body* ; caso contrário, ele retorna `nil`.

Aqui está o exemplo no início desta subseção reescrito usando `ignore-errors`:

```
(ignorar-erros
 (excluir nome de arquivo do arquivo))
```

Macro: corpo da condição ignorar-erro …

Esta macro é como `ignore-errors`, mas apenas ignorará a condição de erro específica especificada.

```
(ignore-erro de fim de arquivo
  (leitura ""))
```

condição também pode ser uma lista de condições de erro.

Macro: *corpo do formato* com erros rebaixados …

Essa macro é como uma versão mais suave do `ignore-errors`. Em vez de suprimir completamente os erros, ele os converte em mensagens. *Ele usa o formato* de string para formatar a mensagem. *formato* deve conter um único '`%`'-seqüência; por exemplo, `"Error: %S"`. Use `with-demoted-errors`em torno de código que não deve sinalizar erros, mas deve ser robusto se ocorrer. Observe que essa macro usa `condition-case-unless-debug`em vez de `condition-case`.

Próximo:Símbolos de erro, Anterior:Processamento de Erros, Acima:Erros [Conteúdo][Índice]