Próximo:Funções de mapeamento, Anterior:Definindo funções, Acima:Funções [Conteúdo][Índice]

13.5 Funções de Chamada

Definir funções é apenas metade da batalha. As funções não fazem nada até que você as *chame* , ou seja, diga a elas para serem executadas. Chamar uma função também é conhecido como *invocação* .

A maneira mais comum de invocar uma função é avaliando uma lista. Por exemplo, avaliar a lista `(concat "a" "b")`chama a função `concat`com argumentos `"a"`e `"b"`. Consulte Avaliação , para obter uma descrição da avaliação.

Ao escrever uma lista como uma expressão em seu programa, você especifica qual função chamar e quantos argumentos fornecer no texto do programa. Normalmente é isso que você quer. Ocasionalmente, você precisa calcular em tempo de execução qual função chamar. Para isso, use a função `funcall`. Quando você também precisar determinar em tempo de execução quantos argumentos passar, use `apply`.

Função: função funcall *e argumentos de descanso*

`funcall`chama a *função* com *argumentos* e retorna qualquer *função* retornada.

Como `funcall`é uma função, todos os seus argumentos, incluindo *function* , são avaliados antes `funcall`de ser chamado. Isso significa que você pode usar qualquer expressão para obter a função a ser chamada. Significa também que `funcall`não vê as expressões que escreve para os *argumentos* , apenas os seus valores. Esses valores *não* são avaliados uma segunda vez no ato de chamar a *função* ; a operação de `funcall`é como o procedimento normal para chamar uma função, uma vez que seus argumentos já foram avaliados.

A função argumento deve ser uma função Lisp ou uma função primitiva. Formulários e macros especiais não são permitidos, porque só fazem sentido quando são fornecidas as expressões de argumento não avaliadas. `funcall`não pode fornecê-los porque, como vimos acima, nunca os conhece em primeiro lugar.

Se você precisar usar `funcall`para chamar um comando e fazê-lo se comportar como se fosse invocado interativamente, use `funcall-interactively` (consulte Chamada interativa).

```
(setq f 'lista)
     ⇒ lista
(funcall f 'x 'y 'z)
     ⇒ (xyz)
(funcall f 'x 'y '(z))
     ⇒ (xy (z))
(funcall 'e t nil)
erro→ Função inválida: #<subr e>
```

Compare esses exemplos com os exemplos de `apply`.

Função: aplicar *argumentos de função &rest*

`apply`chama a *função* com *argumentos* , assim como, `funcall`mas com uma diferença: o último dos *argumentos* é uma lista de objetos, que são passados para *funcionar* como argumentos separados, em vez de uma única lista. Dizemos que `apply` *espalha* essa lista para que cada elemento individual se torne um argumento.

`apply`retorna o resultado da chamada da *função* . Tal como acontece com `funcall`, a *função* deve ser uma função Lisp ou uma função primitiva; formulários especiais e macros não fazem sentido no `apply`.

```
(setq f 'lista)
     ⇒ lista
(aplicar f 'x 'y 'z)
erro→ tipo de argumento errado: listp, z
(aplicar '+ 1 2 '(3 4))
     ⇒ 10
(aplicar '+' (1 2 3 4))
     ⇒ 10


(aplica 'append' ((abc) nil (xyz) nil))
     ⇒ (abcxyz)
```

Para um exemplo interessante de uso `apply`, veja Definição de mapcar .

Às vezes é útil corrigir alguns dos argumentos da função em determinados valores e deixar o restante dos argumentos para quando a função for realmente chamada. O ato de fixar alguns dos argumentos da função é chamado de *aplicação parcial* da função [11] . O resultado é uma nova função que aceita o resto dos argumentos e chama a função original com todos os argumentos combinados.

Veja como fazer aplicação parcial no Emacs Lisp:

Função: aplicar-parcialmente *func & rest args*

Esta função retorna uma nova função que, quando chamada, chamará *func* com a lista de argumentos composta por *args* e argumentos adicionais especificados no momento da chamada. Se *func* aceitar *n* argumentos, uma chamada para `apply-partially`with arguments produzirá uma nova função de argumentos. *`m < nn - m`*

Veja como poderíamos definir a função interna `1+`, se ela não existisse, usando `apply-partially`e `+`, outra função interna:

```
(defalias '1+ (aplicar parcialmente '+ 1)
  "Incrementar o argumento em um.")
(1+ 10)
     ⇒ 11
```

É comum que funções Lisp aceitem funções como argumentos ou as encontrem em estruturas de dados (especialmente em variáveis de gancho e listas de propriedades) e as chamem usando `funcall`ou `apply`. Funções que aceitam argumentos de função são frequentemente chamadas de *funcionais* .

Às vezes, quando você chama um funcional, é útil fornecer uma função no-op como argumento. Aqui estão dois tipos diferentes de função no-op:

Função: *argumento de* identidade

Esta função retorna *argumento* e não tem efeitos colaterais.

Função: ignora *&rest argumentos*

Esta função ignora quaisquer *argumentos* e retorna `nil`.

Algumas funções são comandos visíveis ao usuário , que podem ser chamados interativamente (geralmente por uma sequência de teclas). É possível invocar tal comando exatamente como se fosse chamado interativamente, usando a `call-interactively`função. Consulte Chamada interativa .

Notas de rodapé

(11)

Isso está relacionado, mas diferente de *currying* , que transforma uma função que recebe vários argumentos de tal forma que pode ser chamada como uma cadeia de funções, cada uma com um único argumento.

Próximo:Funções de mapeamento, Anterior:Definindo funções, Acima:Funções [Conteúdo][Índice]