4.3 Criando Strings

As funções a seguir criam strings, seja do zero, juntando strings ou separando-as. (Para funções que criam strings com base no conteúdo modificado de outras strings, como `string-replace`e `replace-regexp-in-string`, consulte Pesquisar e substituir .)

Função: caractere de contagem de strings *e multibyte opcional*

Esta função retorna uma string composta de repetições de *contagem de caracteres* . Se a *contagem* for negativa, um erro é sinalizado.

```
(fazer string 5 ?x)
     ⇒ "xxxxx"
(fazer string 0 ?x)
     ⇒ ""
```

Normalmente, se o *caractere* for um caractere ASCII , o resultado será uma string unibyte. Mas se o argumento opcional *multibyte* for non- `nil`, a função produzirá uma string multibyte. Isso é útil quando mais tarde você precisar concatenar o resultado com strings não ASCII ou substituir alguns de seus caracteres por caracteres não ASCII .

Outras funções para comparar com esta incluem `make-vector` (veja Vetores) e `make-list`(veja Listas de Construção).

Função: string *&rest caracteres*

Isso retorna uma string contendo os caracteres *caracteres* .

```
(string ?a ?b ?c)
     ⇒ "abc"
```

Function: substring *string &optional start end*

This function returns a new string which consists of those characters from *string* in the range from (and including) the character at the index *start* up to (but excluding) the character at the index *end*. The first character is at index zero. With one argument, this function just copies *string*.

```
(substring "abcdefg" 0 3)
     ⇒ "abc"
```

In the above example, the index for ‘`a`’ is 0, the index for ‘`b`’ is 1, and the index for ‘`c`’ is 2. The index 3—which is the fourth character in the string—marks the character position up to which the substring is copied. Thus, ‘`abc`’ is copied from the string `"abcdefg"`.

A negative number counts from the end of the string, so that -1 signifies the index of the last character of the string. For example:

```
(substring "abcdefg" -3 -1)
     ⇒ "ef"
```

In this example, the index for ‘e’ is -3, the index for ‘f’ is -2, and the index for ‘g’ is -1. Therefore, ‘e’ and ‘f’ are included, and ‘g’ is excluded.

When `nil` is used for *end*, it stands for the length of the string. Thus,

```
(substring "abcdefg" -3 nil)
     ⇒ "efg"
```

Omitting the argument *end* is equivalent to specifying `nil`. It follows that `(substring string 0)` returns a copy of all of *string*.

```
(substring "abcdefg" 0)
     ⇒ "abcdefg"
```

But we recommend `copy-sequence` for this purpose (see Sequence Functions).

If the characters copied from *string* have text properties, the properties are copied into the new string also. See Text Properties.

`substring` also accepts a vector for the first argument. For example:

```
(substring [a b (c) "d"] 1 3)
     ⇒ [b (c)]
```

A `wrong-type-argument` error is signaled if *start* is not an integer or if *end* is neither an integer nor `nil`. An `args-out-of-range` error is signaled if *start* indicates a character following *end*, or if either integer is out of range for *string*.

Contrast this function with `buffer-substring` (see Buffer Contents), which returns a string containing a portion of the text in the current buffer. The beginning of a string is at index 0, but the beginning of a buffer is at index 1.

Function: substring-no-properties *string &optional start end*

This works like `substring` but discards all text properties from the value. Also, *start* may be omitted or `nil`, which is equivalent to 0. Thus, `(substring-no-properties string)` returns a copy of *string*, with all text properties removed.

Function: concat *&rest sequences*

This function returns a string consisting of the characters in the arguments passed to it (along with their text properties, if any). The arguments may be strings, lists of numbers, or vectors of numbers; they are not themselves changed. If `concat` receives no arguments, it returns an empty string.

```
(concat "abc" "-def")
     ⇒ "abc-def"
(concat "abc" (list 120 121) [122])
     ⇒ "abcxyz"
;; nil is an empty sequence.
(concat "abc" nil "-def")
```

```
     ⇒ "abc-def"
(concat "The " "quick brown " "fox.")
     ⇒ "The quick brown fox."
(concat)
     ⇒ ""
```

This function does not always allocate a new string. Callers are advised not rely on the result being a new string nor on it being `eq` to an existing string.

In particular, mutating the returned value may inadvertently change another string, alter a constant string in the program, or even raise an error. To obtain a string that you can safely mutate, use `copy-sequence` on the result.

For information about other concatenation functions, see the description of `mapconcat` in Mapping Functions, `vconcat` in Vector Functions, and `append` in Building Lists. For concatenating individual command-line arguments into a string to be used as a shell command, see combine-and-quote-strings.

Function: split-string ***string &optional separators omit-nulls trim***

This function splits *string* into substrings based on the regular expression *separators* (see Regular Expressions). Each match for *separators* defines a splitting point; the substrings between splitting points are made into a list, which is returned.

If *separators* is `nil` (or omitted), the default is the value of `split-string-default-separators` and the function behaves as if *omit-nulls* were `t`.

If *omit-nulls* is `nil` (or omitted), the result contains null strings whenever there are two consecutive matches for *separators*, or a match is adjacent to the beginning or end of *string*. If *omit-nulls* is `t`, these null strings are omitted from the result.

If the optional argument *trim* is non-`nil`, it should be a regular expression to match text to trim from the beginning and end of each substring. If trimming makes the substring empty, it is treated as null.

If you need to split a string into a list of individual command-line arguments suitable for `call-process` or `start-process`, see split-string-and-unquote.

Examples:

```
(split-string "  two words ")
     ⇒ ("two" "words")
```

The result is not `("" "two" "words" "")`, which would rarely be useful. If you need such a result, use an explicit value for *separators*:

```
(split-string "  two words "
              split-string-default-separators)
     ⇒ ("" "two" "words" "")
```

```
(split-string "Soup is good food" "o")
     ⇒ ("S" "up is g" "" "d f" "" "d")
(split-string "Soup is good food" "o" t)
     ⇒ ("S" "up is g" "d f" "d")
```

```
(split-string "Soup is good food" "o+")
     ⇒ ("S" "up is g" "d f" "d")
```

Empty matches do count, except that `split-string` will not look for a final empty match when it already reached the end of the string using a non-empty match or when *string* is empty:

```
(split-string "aooob" "o*")
     ⇒ ("" "a" "" "b" "")
(split-string "ooaboo" "o*")
     ⇒ ("" "" "a" "b" "")
(split-string "" "")
     ⇒ ("")
```

However, when *separators* can match the empty string, *omit-nulls* is usually `t`, so that the subtleties in the three previous examples are rarely relevant:

```
(split-string "Soup is good food" "o*" t)
     ⇒ ("S" "u" "p" " " "i" "s" " " "g" "d" " " "f" "d")
(split-string "Nice doggy!" "" t)
     ⇒ ("N" "i" "c" "e" " " "d" "o" "g" "g" "y" "!")
(split-string "" "" t)
     ⇒ nil
```

Somewhat odd, but predictable, behavior can occur for certain “non-greedy” values of *separators* that can prefer empty matches over non-empty matches. Again, such values rarely occur in practice:

```
(split-string "ooo" "o*" t)
     ⇒ nil
(split-string "ooo" "\\|o+" t)
     ⇒ ("o" "o" "o")
```

Variável: split-string-default-separators

O valor padrão dos *separadores* para `split-string`. Seu valor usual é `"[ \f\t\n\r\v]+"`.

Próximo:Modificando Strings, Anterior:Predicados para Strings, Acima:Strings e Personagens [

Conteúdo][Índice]