WEBINAR: PURRR AVANÇADO

C. LENTE + R6

2018-04-09

ESTRUTURA DO DOCUMENTO

O `purrr` é dividido em 23 famílias. Cada seção deste arquivo corresponde a uma ou mais destas famílias; as primeiras 3 são o que podemos considerar "`purrr` básico" e as outras 20 são o que, para o propósito desta aula, chamaremos de "`purrr` avançado".

Vamos entrar em mais detalhes logo menos, mas ficam aqui as famílias e as funções que as compõem respectivamente:

- Família map
  - `map()`
  - `walk()`
  - `map_chr()`
  - `map_dbl()`
  - `map_int()`
  - `map_lgl()`
  - `map_dfc()`
  - `map_dfr()`
  - `map_at()`
  - `map_if()`
- Família flatten
  - `flatten_chr()`
  - `flatten_dbl()`
  - `flatten_int()`
  - `flatten_lgl()`
  - `flatten_dfc()`

    - `flatten_dfr()`
    - `flatten()`
- Família map2
    - `map2()`
    - `walk2()`
    - `map2_chr()`
    - `map2_dbl()`
    - `map2_int()`
    - `map2_lgl()`
    - `map2_dfc()`
    - `map2_dfr()`
- Família pmap
    - `pmap()`
    - `pwalk()`
    - `pmap_chr()`
    - `pmap_dbl()`
    - `pmap_int()`
    - `pmap_lgl()`
    - `pmap_dfc()`
    - `pmap_dfr()`
- Família imap
    - `imap()`
    - `iwalk()`
    - `imap_chr()`
    - `imap_dbl()`
    - `imap_int()`
    - `imap_lgl()`
    - `imap_dfc()`
    - `imap_dfr()`
- Família lmap
    - `lmap()`
    - `lmap_at()`
    - `lmap_if()`
- Família reduce
    - `accumulate()`
    - `accumulate_right()`
    - `reduce()`

    - `reduce_right()`
    - `reduce2()`
    - `reduce2_right()`
- Família keep
    - `compact()`
    - `discard()`
    - `keep()`
- Família pluck
    - `pluck()`
    - `attr_getter()`
- Família modify
    - `modify()`
    - `modify_at()`
    - `modify_if()`
    - `modify_depth()`
- Família list
    - `set_names()`
    - `prepend()`
    - `transpose()`
    - `list_along()`
    - `rep_along()`
    - `splice()`
    - `list_merge()`
    - `list_modify()`
    - `update_list()`
- Família detect
    - `detect()`
    - `detect_index()`
    - `has_element()`
    - `head_while()`
    - `tail_while()`
- Família cross
    - `cross()`
    - `cross2()`
    - `cross3()`
    - `cross_df()`
- Família is

    - `is_character()`
    - `is_double()`
    - `is_formula()`
    - `is_function()`
    - `is_integer()`
    - `is_list()`
    - `is_logical()`
    - `is_numeric()`
    - `is_atomic()`
    - `is_empty()`
    - `is_null()`
    - `is_vector()`
    - `is_scalar_atomic()`
    - `is_scalar_character()`
    - `is_scalar_double()`
    - `is_scalar_integer()`
    - `is_scalar_list()`
    - `is_scalar_logical()`
    - `is_scalar_numeric()`
    - `is_scalar_vector()`
    - `is_bare_atomic()`
    - `is_bare_character()`
    - `is_bare_double()`
    - `is_bare_integer()`
    - `is_bare_list()`
    - `is_bare_logical()`
    - `is_bare_numeric()`
    - `is_bare_vector()`
- Família as
    - `as_mapper()`
- Família invoke
    - `invoke()`
    - `invoke_map()`
    - `invoke_map_chr()`
    - `invoke_map_dbl()`
    - `invoke_map_int()`
    - `invoke_map_lgl()`

    - `invoke_map_dfc()`
    - `invoke_map_dfr()`
- Família simplify
    - `as_vector()`
    - `simplify()`
    - `simplify_all()`
- Família tree
    - `array_branch()`
    - `array_tree()`
- Família compose
    - `compose()`
    - `partial()`
- Família lift
    - `lift()`
    - `lift_dl()`
    - `lift_dv()`
    - `lift_ld()`
    - `lift_lv()`
    - `lift_vd()`
    - `lift_vl()`
- Família capture
    - `possibly()`
    - `quietly()`
    - `safely()`
    - `auto_browse()`
- Família logic
    - `every()`
    - `some()`
    - `negate()`
    - `when()`
- Família misc
    - `"%@%()"`
    - `"%||%()"`
    - `vec_depth()`
    - `rbernoulli()`
    - `rdunif()`
    - `rerun()`

PURRR BÁSICO (FAMÍLIAS MAP, FLATTEN E MAP2)

Não vamos entrar em detalhes nesta seção, pois assume-se que todos já saibam como as funções básicas do `purrr` trabalham. Em todo caso, seguem alguns exemplos simples:

```
# Duas funções bem simples
f <- function(a) { a*2 + 1 }
g <- function(a, b) { a*2 + b*2 + 1 }

# Map simples
map(1:5, f)
```

```
#> [[1]]
#> [1] 3
#>
#> [[2]]
#> [1] 5
#>
#> [[3]]
#> [1] 7
#>
#> [[4]]
#> [1] 9
#>
#> [[5]]
#> [1] 11
```

```
map(1:5, g, b = 3)
```

```
#> [[1]]
#> [1] 9
#>
#> [[2]]
#> [1] 11
#>
#> [[3]]
#> [1] 13
#>
#> [[4]]
#> [1] 15
#>
#> [[5]]
#> [1] 17
```

```
# Achatamento
1:5 %>% map(f) %>% flatten_dbl()
```

```
#> [1]  3  5  7  9 11
```

```
map_dbl(1:5, f)
```

```
#> [1]  3  5  7  9 11
```

```
# Variações
map2(1:5, 6:10, g)
```

```
#> [[1]]
#> [1] 15
#>
#> [[2]]
#> [1] 19
#>
#> [[3]]
#> [1] 23
#>
#> [[4]]
#> [1] 27
#>
#> [[5]]
#> [1] 31
```

```
map2_dbl(1:5, 6:10, g)
```

```
#> [1] 15 19 23 27 31
```

```
# Muito mais...
map_dbl(1:5, ~.x*2 + 1)
```

```
#> [1]  3  5  7  9 11
```

```
walk(1:5, function(a) { print(a); return(a+2) })
```

```
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

Nas próprias funções do `purrr` básico já podemos ter um gostinho de técnicas mais complexas: condicionais e achatamento para tabelas.

```
# Uma função predicado
p <- function(a) { return(a > 3) }

# Condicionais
map_if(1:5, p, f)
```

```
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
#>
#> [[4]]
#> [1] 9
#>
#> [[5]]
#> [1] 11
```

```
map_at(1:5, c(2, 5), f)
```

```
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 5
#>
#> [[3]]
#> [1] 3
#>
#> [[4]]
#> [1] 4
#>
#> [[5]]
#> [1] 11
```

```r
# Funções que retornam tabelas
ftab <- function(a) { tibble::tibble(col1 = a, col2 = -a) }
gtab <- function(name, a) { tibble::tibble(!!name := a) }

# Achatamento em tabelas
map_dfr(list(1:5, 6:10), ftab)
```

```
#> # A tibble: 10 x 2
#>     col1  col2
#>    <int> <int>
#>  1     1    -1
#>  2     2    -2
#>  3     3    -3
#>  4     4    -4
#>  5     5    -5
#>  6     6    -6
#>  7     7    -7
#>  8     8    -8
#>  9     9    -9
#> 10    10   -10
```

```r
map2_dfc(list("coluna1", "coluna2"), list(1:5, 6:10), gtab)
```

```
#> # A tibble: 5 x 2
#>   coluna1 coluna2
#>     <int>   <int>
#> 1       1       6
#> 2       2       7
#> 3       3       8
#> 4       4       9
#> 5       5      10
```

PURRR INTERMEDIÁRIO (FAMÍLIAS PMAP, IMAP, LMAP)

Ainda existem 3 variações da `map()` padrão, cada uma com suas variações internas (`_dbl`, `_chr`, `_if`, `_at`, ...):

```r
# Função que recebe 4 argumentos
f4 <- function(a, b, c, d) { a + b - c + d }

# Map 3, 4, 5...
pmap(list(1:3, 4:6, 7:9, 10:12), f4)
```

```
#> [[1]]
#> [1] 8
#>
#> [[2]]
#> [1] 10
#>
#> [[3]]
#> [1] 12
```

```
# Função que usa um índice
fi <- function(a, i, b) { a + b[length(b)-i+1] }

# Índice sendo passado implicitamente
imap(1:5, fi, b = 6:10)
```

```
#> [[1]]
#> [1] 11
#>
#> [[2]]
#> [1] 11
#>
#> [[3]]
#> [1] 11
#>
#> [[4]]
#> [1] 11
#>
#> [[5]]
#> [1] 11
```

```
# Ao invés de operar em .x[[i]], operar em .x[i]
lmap(list(1:3, 4:6), ~list(length(.x)))
```

```
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 1
```

FAMÍLIA REDUCE

Essa família tem muitas semelhanças com as anteriores, mas com alguns detalhes a mais:

```
# Acumular e reduzir
accumulate(c("a", "b", "c", "d"), paste)
```

```
#> [1] "a"       "a b"     "a b c"   "a b c d"
```

```
reduce(c("a", "b", "c", "d"), paste)
```

```
#> [1] "a b c d"
```

```
# E mais...
accumulate_right(c("a", "b", "c", "d"), paste)
```

```
#> [1] "d c b a" "d c b"   "d c"     "d"
```

```
reduce2(c("a", "b", "c", "d"), c("-", ".", "-"), paste)
```

```
#> [1] "a b - c . d -"
```

FAMÍLIAS KEEP E PLUCK

As funções dessas famílias ajudam a selecionar subconjuntos de elementos de uma lista:

```
# Manter ou descartar os ementos que atendem um predicado
keep(list(1, 2, "c", 4, "e"), is.character)
```

```
#> [[1]]
#> [1] "c"
#>
#> [[2]]
#> [1] "e"
```

```
discard(list(1, 2, "c", 4, "e"), is.character)
```

```
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 4
```

```
# Atalho para discard(x, is.null)
compact(list(NULL, NULL, 3, NULL, 5))
```

```
#> [[1]]
#> [1] 3
#>
#> [[2]]
#> [1] 5
```

```
# Uma lista com sub listas
lista <- list(
  a = list(c = 1, d = 2),
  b = list(c = 3, d = 4))

# Selecionar elementos por nome e índica
pluck(lista, "a", 2)
```

```
#> [1] 2
```

```
pluck(lista, "a", 2, f)
```

```
#> [1] 5
```

```
# Lista de elementos com atributos
lista_attr <- list(
  structure("a", atributo = "attr_c"),
  structure("b", atributo = "attr_d"))

# Selecionar atributo ao fim do pluck()
pluck(lista_attr, 1, attr_getter("atributo"))
```

```
#> [1] "attr_c"
```

FAMÍLIA MODIFY

`modify()` parece muito com `map()`, mas ela modifica um elemento no seu lugar:

```
# Uma tabela comum
iris
```

```
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1           5.1         3.5          1.4         0.2  setosa
#> 2           4.9         3.0          1.4         0.2  setosa
#> 3           4.7         3.2          1.3         0.2  setosa
#> 4           4.6         3.1          1.5         0.2  setosa
#> 5           5.0         3.6          1.4         0.2  setosa
#> 6           5.4         3.9          1.7         0.4  setosa
#> 7           4.6         3.4          1.4         0.3  setosa
#> 8           5.0         3.4          1.5         0.2  setosa
#> 9           4.4         2.9          1.4         0.2  setosa
#> 10          4.9         3.1          1.5         0.1  setosa
#> 11          5.4         3.7          1.5         0.2  setosa
#> 12          4.8         3.4          1.6         0.2  setosa
#> 13          4.8         3.0          1.4         0.1  setosa
#> 14          4.3         3.0          1.1         0.1  setosa
#> 15          5.8         4.0          1.2         0.2  setosa
#>  [ reached 'max' / getOption("max.print") -- omitted 135 rows ]
```

```
# Modificar colunas sem alterar as outras
modify_if(iris, is.factor, as.character)
```

```
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1           5.1         3.5          1.4         0.2  setosa
#> 2           4.9         3.0          1.4         0.2  setosa
#> 3           4.7         3.2          1.3         0.2  setosa
#> 4           4.6         3.1          1.5         0.2  setosa
#> 5           5.0         3.6          1.4         0.2  setosa
#> 6           5.4         3.9          1.7         0.4  setosa
#> 7           4.6         3.4          1.4         0.3  setosa
#> 8           5.0         3.4          1.5         0.2  setosa
#> 9           4.4         2.9          1.4         0.2  setosa
#> 10          4.9         3.1          1.5         0.1  setosa
#> 11          5.4         3.7          1.5         0.2  setosa
#> 12          4.8         3.4          1.6         0.2  setosa
#> 13          4.8         3.0          1.4         0.1  setosa
#> 14          4.3         3.0          1.1         0.1  setosa
#> 15          5.8         4.0          1.2         0.2  setosa
#>  [ reached 'max' / getOption("max.print") -- omitted 135 rows ]
```

```
modify_at(iris, 1:4, as.integer)
```

```
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1             5           3            1           0  setosa
#> 2             4           3            1           0  setosa
#> 3             4           3            1           0  setosa
#> 4             4           3            1           0  setosa
#> 5             5           3            1           0  setosa
#> 6             5           3            1           0  setosa
#> 7             4           3            1           0  setosa
#> 8             5           3            1           0  setosa
#> 9             4           2            1           0  setosa
#> 10            4           3            1           0  setosa
#> 11            5           3            1           0  setosa
#> 12            4           3            1           0  setosa
#> 13            4           3            1           0  setosa
#> 14            4           3            1           0  setosa
#> 15            5           4            1           0  setosa
#>  [ reached 'max' / getOption("max.print") -- omitted 135 rows ]
```

```
# Uma lista profunda
lista_prof <- list(
  a = list(
    c = list(1:3, 4:6),
    d = list(7:9, 10:12)),
  b = list(
    e = list(13:15, 16:18),
    f = list(19:21, 22:24)))

# Modificar em uma profundidade específica
str(lista_prof)
```

```
#> List of 2
#>  $ a:List of 2
#>   ..$ c:List of 2
#>   .. ..$ : int [1:3] 1 2 3
#>   .. ..$ : int [1:3] 4 5 6
#>   ..$ d:List of 2
#>   .. ..$ : int [1:3] 7 8 9
#>   .. ..$ : int [1:3] 10 11 12
#>  $ b:List of 2
#>   ..$ e:List of 2
#>   .. ..$ : int [1:3] 13 14 15
#>   .. ..$ : int [1:3] 16 17 18
#>   ..$ f:List of 2
#>   .. ..$ : int [1:3] 19 20 21
#>   .. ..$ : int [1:3] 22 23 24
```

```
modify_depth(lista_prof, 3, sum) %>% str()
```

```
#> List of 2
#>  $ a:List of 2
#>   ..$ c:List of 2
#>   .. ..$ : int 6
#>   .. ..$ : int 15
#>   ..$ d:List of 2
#>   .. ..$ : int 24
#>   .. ..$ : int 33
#>  $ b:List of 2
#>   ..$ e:List of 2
#>   .. ..$ : int 42
#>   .. ..$ : int 51
#>   ..$ f:List of 2
#>   .. ..$ : int 60
#>   .. ..$ : int 69
```

FAMÍLIA LIST

A família list tem funções interessantes para trabalhar com listas no geral:

```
# Operações boas para vetores
set_names(c("a", "b", "c"), c("nome1", "nome2", "nome3"))
```

```
#> nome1 nome2 nome3
#>   "a"   "b"   "c"
```

```
prepend(1:5, 6:10)
```

```
#>  [1]  6  7  8  9 10  1  2  3  4  5
```

```
# Uma lista com sub listas
lista <- list(
  a = list(c = 1, d = 2),
  b = list(c = 3, d = 4))

# Transposição
str(lista)
```

```
#> List of 2
#>  $ a:List of 2
#>   ..$ c: num 1
#>   ..$ d: num 2
#>  $ b:List of 2
#>   ..$ c: num 3
#>   ..$ d: num 4
```

```
transpose(lista) %>% str()
```

```
#> List of 2
#>  $ c:List of 2
#>   ..$ a: num 1
#>   ..$ b: num 3
#>  $ d:List of 2
#>   ..$ a: num 2
#>   ..$ b: num 4
```

```
# Juntar listas
purrr::splice(list(a = 1, b = 2), c = 3:4, z = list(d = 2:5, e = 3)) %>% str()
```

```
#> List of 5
#>  $ a: num 1
#>  $ b: num 2
#>  $ c: int [1:2] 3 4
#>  $ d: int [1:4] 2 3 4 5
#>  $ e: num 3
```

```
list_merge(list(a = 1, b = 2), c = 3:4, z = list(d = 2:5, e = 3)) %>% str()
```

```
#> List of 4
#>  $ a: num 1
#>  $ b: num 2
#>  $ c: int [1:2] 3 4
#>  $ z:List of 2
#>   ..$ d: int [1:4] 2 3 4 5
#>   ..$ e: num 3
```

```
# Alterar um elemento
list_modify(lista, b = 20) %>% str()
```

```
#> List of 2
#>  $ a:List of 2
#>   ..$ c: num 1
#>   ..$ d: num 2
#>  $ b: num 20
```

FAMÍLIA DETECT

Esta família serve para detectar elementos em uma lista, possivelmente com algum truque adicional na manga:

```
# Qual é o primeiro elemento que atende ao predicado
detect(3:6, ~.x %% 2 == 0)
```

```
#> [1] 4
```

```
detect_index(3:6, ~.x %% 2 == 0)
```

```
#> [1] 2
```

```
# Uma lista com sub listas
lista <- list(
  a = list(c = 1, d = 2),
  b = list(c = 3, d = 4))

# Verificação forte dos elementos
has_element(lista, list(c = 3, d = 4))
```

```
#> [1] TRUE
```

```
# Abstação de while
head_while(5:-5, ~.x > 0)
```

```
#> [1] 5 4 3 2 1
```

```
tail_while(5:-5, ~.x < 0)
```

```
#> [1] -1 -2 -3 -4 -5
```

FAMÍLIA CROSS

Como seu próprio nome indica, essa família faz cruzamentos de listas:

```
# Uma lista de cumprimentos
lista_ola <- list(
  nome = c("João", "Joana"),
  ola = c("Oi.", "Olá."),
  sep = c("! ", "... "))

# Todos os cruzamento da lista
cross(lista_ola) %>% str()
```

```
#> List of 8
#>  $ :List of 3
#>   ..$ nome: chr "João"
#>   ..$ ola : chr "Oi."
#>   ..$ sep : chr "! "
#>  $ :List of 3
#>   ..$ nome: chr "Joana"
#>   ..$ ola : chr "Oi."
#>   ..$ sep : chr "! "
#>  $ :List of 3
#>   ..$ nome: chr "João"
#>   ..$ ola : chr "Olá."
#>   ..$ sep : chr "! "
#>  $ :List of 3
#>   ..$ nome: chr "Joana"
#>   ..$ ola : chr "Olá."
#>   ..$ sep : chr "! "
#>  $ :List of 3
#>   ..$ nome: chr "João"
#>   ..$ ola : chr "Oi."
#>   ..$ sep : chr "... "
#>  $ :List of 3
#>   ..$ nome: chr "Joana"
#>   ..$ ola : chr "Oi."
#>   ..$ sep : chr "... "
#>  $ :List of 3
#>   ..$ nome: chr "João"
#>   ..$ ola : chr "Olá."
#>   ..$ sep : chr "... "
#>  $ :List of 3
#>   ..$ nome: chr "Joana"
#>   ..$ ola : chr "Olá."
#>   ..$ sep : chr "... "
```

```
cross_df(lista_ola)
```

```
#> # A tibble: 8 x 3
#>   nome  ola   sep
#>   <chr> <chr> <chr>
#> 1 João  Oi.   "! "
#> 2 Joana Oi.   "! "
#> 3 João  Olá.  "! "
#> 4 Joana Olá.  "! "
#> 5 João  Oi.   "... "
#> 6 Joana Oi.   "... "
#> 7 João  Olá.  "... "
#> 8 Joana Olá.  "... "
```

```
# Cruzando mais de um objeto e usando filtros
cross2(seq_len(3), seq_len(3), .filter = `==`) %>% str()
```

```
#> List of 6
#>  $ :List of 2
#>   ..$ : int 2
#>   ..$ : int 1
#>  $ :List of 2
#>   ..$ : int 3
#>   ..$ : int 1
#>  $ :List of 2
#>   ..$ : int 1
#>   ..$ : int 2
#>  $ :List of 2
#>   ..$ : int 3
#>   ..$ : int 2
#>  $ :List of 2
#>   ..$ : int 1
#>   ..$ : int 3
#>  $ :List of 2
#>   ..$ : int 2
#>   ..$ : int 3
```

FAMÍLIAS IS E AS

Estas funções são bastante simples e servem para verificar/mudar o tipo dos objetos:

```
# Is
is_formula(~.x + 1)
```

```
#> [1] TRUE
```

```
is_atomic(list(1, "b"))
```

```
#> [1] FALSE
```

```
is_scalar_vector(c(1))
```

```
#> [1] TRUE
```

```
is_bare_list(iris)
```

```
#> [1] FALSE
```

```
# As
as_mapper(~.x + 1)(2)
```

```
#> [1] 3
```

FAMÍLIA INVOKE

`invoke()` e suas variações permite que chamemos funções a partir de listas de argumentos:

```
invoke(runif, list(n = 3, max = 2))
```

```
#> [1] 0.9392057 0.1515579 0.3236864
```

```
# Aplicar uma ou mais listas de argumentos em uma ou mais funções
invoke_map(runif, list(list(n = 2), list(n = 4)))
```

```
#> [[1]]
#> [1] 0.1867056 0.9317268
#>
#> [[2]]
#> [1] 0.8593291 0.0763799 0.8483061 0.8556588
```

```
invoke_map(list(runif, rnorm), list(list(n = 2), list(n = 4)))
```

```
#> [[1]]
#> [1] 0.88883924 0.09186405
#>
#> [[2]]
#> [1] -0.239805881 -0.584666373 -0.003877408 -0.812443427
```

```
# E assim por diante...
invoke_map_dbl(list(m1 = mean, m2 = median) , x = rcauchy(100))
```

```
#>          m1          m2
#> -1.55025008  0.08479983
```

FAMÍLIA SIMPLIFY

Essa pequena família simplifica vetores e listas:

```
# Achatar mantendo os nomes
as_vector(list(a = 1, b = 2))
```

```
#> a b
#> 1 2
```

```
# Aplicar simplify() em cada elemento de uma lista
simplify_all(lista) %>% str()
```

```
#> List of 2
#>  $ a: Named num [1:2] 1 2
#>   ..- attr(*, "names")= chr [1:2] "c" "d"
#>  $ b: Named num [1:2] 3 4
#>   ..- attr(*, "names")= chr [1:2] "c" "d"
```

FAMÍLIA TREE

É raro precisarmos usar essas duas funções, mas podem vir a calhar quando usando arrays:

```
# Uma array
arr <- array(1:12, c(2, 2, 3))

# Transformando a array em uma lista (árvore)
array_tree(arr) %>% str()
```

```
#> List of 2
#>  $ :List of 2
#>   ..$ :List of 3
#>   .. ..$ : int 1
#>   .. ..$ : int 5
#>   .. ..$ : int 9
#>   ..$ :List of 3
#>   .. ..$ : int 3
#>   .. ..$ : int 7
#>   .. ..$ : int 11
#>  $ :List of 2
#>   ..$ :List of 3
#>   .. ..$ : int 2
#>   .. ..$ : int 6
#>   .. ..$ : int 10
#>   ..$ :List of 3
#>   .. ..$ : int 4
#>   .. ..$ : int 8
#>   .. ..$ : int 12
```

```
array_tree(arr, c(1, 3)) %>% str()
```

```
#> List of 2
#>  $ :List of 3
#>   ..$ : int [1:2] 1 3
#>   ..$ : int [1:2] 5 7
#>   ..$ : int [1:2] 9 11
#>  $ :List of 3
#>   ..$ : int [1:2] 2 4
#>   ..$ : int [1:2] 6 8
#>   ..$ : int [1:2] 10 12
```

```
# Cria uma lista achatada (um ramo)
array_branch(arr, c(1, 3)) %>% str()
```

```
#> List of 6
#>  $ : int [1:2] 1 3
#>  $ : int [1:2] 2 4
#>  $ : int [1:2] 5 7
#>  $ : int [1:2] 6 8
#>  $ : int [1:2] 9 11
#>  $ : int [1:2] 10 12
```

FAMÍLIAS COMPOSE E LIFT

Essas duas famílias são um pouco obscuras, mas têm propriedades interessantes para trabalhar com funções

```
# Preencher os argumentos de uma função
runif_com_max <- partial(runif, max = 10)
runif_com_max(3)
```

```
#> [1] 8.656334 7.829822 3.927123
```

```
# Compor funções
runif_sum <- compose(sum, runif)
runif_sum(3)
```

```
#> [1] 1.051571
```

```
# Trocar o que uma função recebe (dots -> list)
mean_list <- lift_dl(mean)
mean_list(list(x = 1:10, na.rm = TRUE))
```

```
#> [1] 5.5
```

```
# E fazer o contrário
mean_dots <- lift_ld(mean_list)
mean_dots(1:10, NA, na.rm = TRUE, trim = 0)
```

```
#> [1] 5.5
```

FAMÍLIA CAPTURE

Apesar de estar perto do final, esta é uma das famílias mais importantes! Ela captura erros e permite que o seu map rode sem ter problemas:

```
# Capturando erros na função runif
runif_p <- possibly(runif, otherwise = "Erro")
mean_q <- quietly(mean)
runif_s <- safely(runif)

# Aplicando funções que vão dar errado
runif_p("abc")
```

```
#> [1] "Erro"
```

```
mean_q("abc")
```

```
#> $result
#> [1] NA
#>
#> $output
#> [1] ""
#>
#> $warnings
#> [1] "argument is not numeric or logical: returning NA"
#>
#> $messages
#> character(0)
```

```
runif_s("abc")
```

```
#> $result
#> NULL
#>
#> $error
#> <simpleError in .f(...): invalid arguments>
```

FAMÍLIA LOGIC

Algumas abstrações de funções lógicas úteis para quando precisamos pipear alguma verificação:

```
# Todos e alguns
every(1:5, ~.x > 2)
```

```
#> [1] FALSE
```

```
some(1:5, ~.x > 2)
```

```
#> [1] TRUE
```

```
# Criar um predicado negando outro
keep(1:5, negate(~.x > 2))
```

```
#> [1] 1 2
```

```
# Abstração de if-else
1:10 %>% when(
  sum(.) <=  50 ~ sum(.),
  sum(.) <= 100 ~ sum(.)/2,
  ~ 0)
```

```
#> [1] 27.5
```

FAMÍLIA MISC

Nessa família residem algumas funções miscelâneas que não se encaixam em nenhuma outra família:

```
# OR com NULL
TRUE %||% NULL
```

```
#> [1] TRUE
```

```
FALSE %||% FALSE
```

```
#> [1] FALSE
```

```
# Seletor de atributos
a <- structure("a", atributo = "b")
a %@% "atributo"
```

```
#> [1] "b"
```

```
# Outras distribuições
rbernoulli(5)
```

```
#> [1] FALSE  TRUE FALSE  TRUE  TRUE
```

```
rdunif(5, 10)
```

```
#> [1] 8 8 3 6 8
```

```
# Rodar várias vezes
rerun(5, runif(3))
```

```
#> [[1]]
#> [1] 0.57813126 0.09974351 0.99393171
#>
#> [[2]]
#> [1] 0.4187742 0.6000602 0.6778603
#>
#> [[3]]
#> [1] 0.8219049 0.8282473 0.2511693
#>
#> [[4]]
#> [1] 0.01524815 0.01492729 0.79860283
#>
#> [[5]]
#> [1] 0.77609929 0.05541584 0.47310913
```