Fondamenti di Informatica (Elettronici)

THINK JULIA – Capitolo 10

13 novembre 2020

10. Array (prima parte)[1]

1. Un array è una sequenza
2. Gli array sono mutevoli
3. Attraversare un array
4. Slice di array
5. Libreria sugli array
6. `Map`, `Filter` e `Reduce`

[1] Tratto da https://benlauwens.github.io/ThinkJulia.jl/latest/book.html, disponibile sotto Licenza 'Creative Commons Attribution-NonCommercial 3.0 Unported'.

Questo capitolo presenta uno dei tipi predefiniti più utili di Julia, gli array.

Studierai anche gli oggetti, e cosa può accadere quando hai più di un nome per lo stesso oggetto.

Section 1

Un array è una sequenza

Elementi della matrice o items

Come una stringa, un array è una sequenza di valori.

In una stringa, i valori sono caratteri; in un array, possono essere di qualsiasi tipo.

I valori in un array sono chiamati items o talvolta elementi.

Esistono diversi modi per creare un nuovo array; il più semplice è racchiudere gli elementi tra parentesi quadre ("[]"):

```
[10, 20, 30, 40]
["crunchy frog", "ram bladder", "lark vomit"]
```

Il primo esempio è un array di quattro numeri interi. Il secondo è un array di tre stringhe.

Gli elementi di un array non devono essere dello stesso tipo.

Gli elementi di un array possono avere tipi diversi

Il seguente array contiene una stringa, un float, un numero intero e un altro array:

```
["spam", 2.0, 5, [10, 20]]
```

Un array all'interno di un altro array è nidificato.

Un array che non contiene elementi è chiamato array vuoto; puoi crearne uno con parentesi vuote, "[]".

Come ci si potrebbe aspettare, è possibile assegnare valori di array alle variabili:

```
julia> cheeses = ["Cheddar", "Edam", "Gouda"]
3-element Array{String,1}:
 "Cheddar"
 "Edam"
 "Gouda"
julia> numbers = [42, 123]
2-element Array{Int64,1}:
  42
 123
julia> empty = []
Any[]
```

Funzione typeof

```
julia> print(cheeses, " ", numbers, " ", empty)
["Cheddar", "Edam", "Gouda"] [42, 123] Any[]
```

La funzione `typeof` può essere utilizzata per scoprire il tipo di array:

```
julia> typeof(cheeses)
Array{String,1}
julia> typeof(numbers)
Array{Int64,1}
julia> typeof(empty)
Array{Any,1}
```

Il tipo array è specificato tra parentesi graffe ed è composto dal tipo degli elementi e da un numero. Il numero indica le dimensioni.

L'array vuoto contiene valori di tipo `Any`, ovvero. può contenere valori di tutti i tipi.

Section 2

Gli array sono mutevoli

Gli array sono mutevoli

La sintassi per accedere agli elementi di un array è la stessa utilizzata per accedere ai caratteri di una stringa: l'operatore parentesi quadre.

L'espressione tra parentesi specifica l'"indice". Ricorda che gli indici iniziano da 1:

```
julia> cheeses[1]
"Cheddar"
```

A differenza delle stringhe, gli array sono "mutabili". Quando l'operatore parentesi viene visualizzato sul lato sinistro di una assegnazione, identifica l'elemento dell'array che verrà assegnato:

```
julia> numbers[2] = 5
5
julia> print(numbers)
[42, 5]
```

Il secondo elemento di "numbers", che prima era "123", ora è "5".

Diagramma di stato

La figura mostra i diagrammi di stato per cheeses, numeri e valori vuoti.

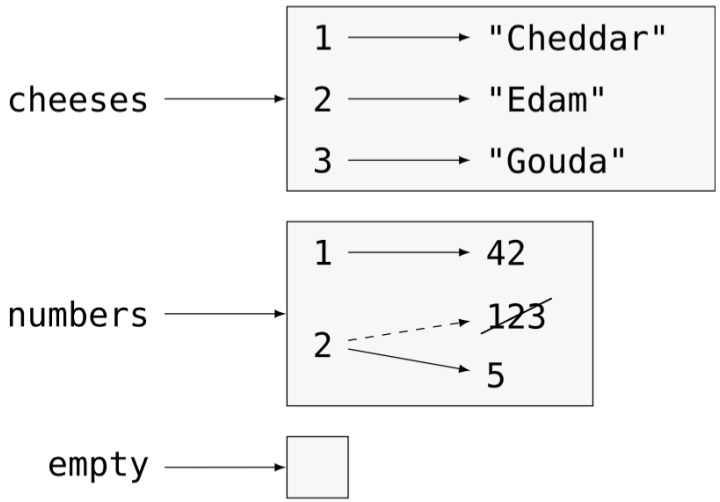

Figure 11. State diagram

Gli array sono rappresentati da caselle e gli elementi dell'array sono all'interno.

"cheeses" si riferisce a un vettore con tre elementi indicizzati 1, 2 e 3.

"numbers" contiene due elementi; il diagramma mostra che il valore del secondo elemento è stato riassegnato da 123 a 5.

"empty" si riferisce a un array senza elementi.

Indici di array

Gli indici di un "array" funzionano allo stesso modo degli indici di stringa (ma senza avvertenze "UTF-8"):

- Qualsiasi espressione intera può essere utilizzata come indice.

Indici di array

Gli indici di un "array" funzionano allo stesso modo degli indici di stringa (ma senza avvertenze "UTF-8"):

- Qualsiasi espressione intera può essere utilizzata come indice.
- Se provi a leggere o scrivere un elemento che non esiste, ottieni un'eccezione `BoundsError`.

Indici di array

Gli indici di un "array" funzionano allo stesso modo degli indici di stringa (ma senza avvertenze "UTF-8"):

- Qualsiasi espressione intera può essere utilizzata come indice.
- Se provi a leggere o scrivere un elemento che non esiste, ottieni un'eccezione `BoundsError`.
- La parola chiave `end` punta all'ultimo indice dell'array.

Indici di array

Gli indici di un "array" funzionano allo stesso modo degli indici di stringa (ma senza avvertenze "UTF-8"):

- Qualsiasi espressione intera può essere utilizzata come indice.
- Se provi a leggere o scrivere un elemento che non esiste, ottieni un'eccezione `BoundsError`.
- La parola chiave `end` punta all'ultimo indice dell'array.
- L'operatore $\in$ funziona anche sugli array:

Indici di array

Gli indici di un "array" funzionano allo stesso modo degli indici di stringa (ma senza avvertenze "UTF-8"):

- Qualsiasi espressione intera può essere utilizzata come indice.
- Se provi a leggere o scrivere un elemento che non esiste, ottieni un'eccezione `BoundsError`.
- La parola chiave `end` punta all'ultimo indice dell'array.
- L'operatore $\in$ funziona anche sugli array:

Indici di array

Gli indici di un "array" funzionano allo stesso modo degli indici di stringa (ma senza avvertenze "UTF-8"):

- Qualsiasi espressione intera può essere utilizzata come indice.
- Se provi a leggere o scrivere un elemento che non esiste, ottieni un'eccezione `BoundsError`.
- La parola chiave `end` punta all'ultimo indice dell'array.
- L'operatore $\in$ funziona anche sugli array:

```
julia> "Edam" in cheeses
true
julia> "Brie" in cheeses
false
```

Section 3

Attraversare un array

Funzione eachindex

Il modo più comune per attraversare gli elementi di un array è con un ciclo `for`. La sintassi è la stessa delle stringhe:

```
for cheese in cheeses
    println(cheese)
end
```

`in` funziona bene se hai solo bisogno di leggere gli elementi dell'array. Ma se vuoi scrivere o aggiornare gli elementi, hai bisogno degli indici. Un modo comune per farlo è usare la funzione predefinita `eachindex`:

```
for i in eachindex(numbers)
    numbers[i] = numbers[i] * 2
end
```

- Questo ciclo attraversa l'array e aggiorna ogni elemento.

Funzione eachindex

Il modo più comune per attraversare gli elementi di un array è con un ciclo `for`. La sintassi è la stessa delle stringhe:

```
for cheese in cheeses
    println(cheese)
end
```

`in` funziona bene se hai solo bisogno di leggere gli elementi dell'array. Ma se vuoi scrivere o aggiornare gli elementi, hai bisogno degli indici. Un modo comune per farlo è usare la funzione predefinita `eachindex`:

```
for i in eachindex(numbers)
    numbers[i] = numbers[i] * 2
end
```

- Questo ciclo attraversa l'array e aggiorna ogni elemento.
- Ogni volta che attraverso il ciclo ottengo l'indice dell'elemento successivo.

Array `length`

- La funzione `length` restituisce il numero di elementi nell'array.

L'istruzione di assegnazione nel corpo usa "i" per leggere il vecchio valore dell'elemento e per assegnare il nuovo valore.

Un ciclo `for` su un array vuoto non esegue mai il corpo:

```
for x in []
    println("This can never happens.")
end
```

Sebbene un array possa contenere un altro array, l'array annidato conta ancora come un singolo elemento.

La `lunghezza` di questo array è quattro:

```
["spam", 1, ["Brie", "Roquefort", "Camembert"], [1, 2, 3]]
```

Section 4

Slice di array

Operatore slice

L'operatore slice funziona anche sugli array:

```
julia> t = ['a', 'b', 'c', 'd', 'e', 'f'];
julia> print(t[1:3])
['a', 'b', 'c']
julia> print(t[3:end])
['c', 'd', 'e', 'f']
```

L'operatore slice "[:]" crea una copia dell'intero array:

Spesso è utile farne una copia

```
a = t[:]; a[1] = 'A';
julia> println(a)
['A', 'b', 'c', 'd', 'e', 'f']
julia> println(t)
['a', 'b', 'c', 'd', 'e', 'f']
```

Poiché gli array sono mutabili, è spesso utile crearne una copia prima di eseguire operazioni che modificano gli array.

Un operatore `slice` sul lato sinistro di un'assegnazione può aggiornare più elementi:

```
julia> t[2:3] = ['x', 'y'];
julia> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

Section 5

Libreria sugli array

Funzioni push!, append e sort

Julia fornisce funzioni che operano su array. Ad esempio, la funzione `push!` aggiunge un nuovo elemento alla fine di un array:

```
julia> t = ['a', 'b', 'c'];
julia> push!(t, 'd');
julia> print(t) ['a', 'b', 'c', 'd']
```

La funzione `append!` aggiunge gli elementi del secondo array alla fine del primo:

```
julia> t1 = ['a', 'b', 'c'];
julia> t2 = ['d', 'e'];
julia> append!(t1, t2);
julia> print(t1)
['a', 'b', 'c', 'd', 'e']
```

Questo esempio lascia "t2" non modificato.

La funzione `sort!` dispone gli elementi dell'array dal primo all'ultimo:

sort restituisce una copia

La funzione sort restituisce una copia degli elementi dell'array nell' ordine:

```
julia> t1 = ['d', 'c', 'e', 'b', 'a'];
julia> t2 = sort(t1);
julia> print(t1)
['d', 'c', 'e', 'b', 'a']
julia> print(t2)
['a', 'b', 'c', 'd', 'e']
```

ATTENZIONE

Come convenzione di stile in Julia, il punto esclamativa ! viene aggiunto ai nomi delle funzioni che modificano i loro argomenti.

Section 6

Map, Filter e Reduce

Pattern accumulatore

Per sommare tutti i numeri in un array, puoi usare un ciclo come questo:

```
function addall(t)
    total = 0
    for x in t
        total += x
    end
    total
end
```

"total" è inizializzato a 0.

Ogni volta che si esegue il ciclo, "+=" somma un elemento dall'array.

Variabile accumulatore

L'operatore "+=" fornisce un modo breve per aggiornare una variabile. Questa è chiamata dichiarazione di assegnazione aumentata:

```
total += x
```

è equivalente a

```
total = total + x
```

Durante l'esecuzione del ciclo, "`total`" accumula la somma degli elementi; una variabile usata in questo modo è talvolta chiamata accumulatore.

La somma degli elementi di un array è un'operazione così comune che Julia la fornisce come funzione predefinita, `sum`:

```
julia> t = [1, 2, 3, 4];
julia> sum(t)
10
```

Operazione reduce

Un'operazione come questa che combina una sequenza di elementi in un singolo valore è talvolta chiamata operazione `reduce`.

Spesso si desidera attraversare un array durante la creazione di un altro. Ad esempio, la seguente funzione accetta un array di stringhe e restituisce un nuovo array che contiene stringhe in maiuscolo:

```
function capitalizeall(t)
    res = []
    for s in t
        push!(res, uppercase(s))
    end
    res
end
```

"res" è inizializzato con un array vuoto; ogni volta nel ciclo, aggiungiamo l'elemento successivo. Quindi "res" è un altro tipo di accumulatore.

Operazioni map e filter

Un'operazione come capitalizeall è talvolta chiamata map perché "mappa" una funzione (in questo caso uppercase) su ciascuno degli elementi in una sequenza.

Un'altra operazione comune è quella di select alcuni elementi da un array e restituire un sottoarray.

Ad esempio, la seguente funzione accetta un array di stringhe e restituisce un array che contiene solo le stringhe maiuscole:

```
function onlyupper(t) res = []
    for s in t
        if s == uppercase(s)
            push!(res, s)
        end
    end
    res
end
```

Un'operazione come onlyupper è chiamata di tipo filter perché seleziona alcuni degli elementi e filtra gli altri.

Le operazioni sugli array più comuni possono essere espresse come una combinazione di map,filter e reduce.