



**EBook Gratuito**

# APPENDIMENTO

# Julia Language

Free unaffiliated eBook created from  
**Stack Overflow contributors.**

**#julia-lang**

# Sommario

Di.....	1
<b>Capitolo 1: Iniziare con Julia Language.....</b>	<b>2</b>
Versioni.....	2
Examples.....	2
Ciao mondo!.....	2
<b>Capitolo 2: @goto e @label.....</b>	<b>4</b>
Sintassi.....	4
Osservazioni.....	4
Examples.....	4
Convalida dell'input.....	4
Pulizia errore.....	5
<b>Capitolo 3: Aritmetica.....</b>	<b>7</b>
Sintassi.....	7
Examples.....	7
Formula quadratica.....	7
Setaccio di Eratostene.....	7
Matrix Arithmetic.....	8
Le somme.....	8
Prodotti.....	9
potenze.....	9
<b>Capitolo 4: Array.....</b>	<b>11</b>
Sintassi.....	11
Parametri.....	11
Examples.....	11
Costruzione manuale di un array semplice.....	11
Tipi di matrice.....	12
Array di array - Proprietà e costruzione.....	13
Inizializza una matrice vuota.....	14
Vettori.....	14
Concatenazione.....	15

Concatenazione orizzontale.....	15
Concatenazione verticale.....	16
<b>Capitolo 5: chiusure.....</b>	<b>18</b>
Sintassi.....	18
Osservazioni.....	18
Examples.....	18
Composizione funzionale.....	18
Implementare Currying.....	19
Introduzione alle chiusure.....	20
<b>Capitolo 6: combinatori.....</b>	<b>22</b>
Osservazioni.....	22
Examples.....	22
Il Y o Z Combinator.....	22
Il sistema di combinazione SKI.....	23
Una traduzione diretta dal Lambda Calculus.....	23
Mostrando Combinatori SKI.....	24
<b>Capitolo 7: Compatibilità tra versioni.....</b>	<b>27</b>
Sintassi.....	27
Osservazioni.....	27
Examples.....	27
Numeri di versione.....	27
Utilizzando Compat.jl.....	28
Tipo di stringa unificato.....	28
Sintassi di trasmissione compatta.....	29
<b>Capitolo 8: Comprensioni.....</b>	<b>30</b>
Examples.....	30
Array comprehension.....	30
Sintassi di base.....	30
Comprensione dell'array condizionale.....	30
Comprensioni di array multidimensionali.....	31
Comprensioni del generatore.....	32

Argomenti della funzione.....	32
<b>Capitolo 9: Condizionali.....</b>	<b>33</b>
Sintassi.....	33
Osservazioni.....	33
Examples.....	33
se ... altra espressione.....	33
se ... altra affermazione.....	34
se la dichiarazione.....	34
Operatore condizionale ternario.....	34
Operatori di cortocircuito: && e   .....	35
Per la ramificazione.....	35
In condizioni.....	35
se dichiarazione con più rami.....	36
La funzione ifelse.....	36
<b>Capitolo 10: confronti.....</b>	<b>38</b>
Sintassi.....	38
Osservazioni.....	38
Examples.....	38
Confronti concatenati.....	38
Numeri ordinali.....	40
Operatori standard.....	41
Usando ==, === e isequal.....	42
Quando usare ==.....	42
Quando usare ===.....	43
Quando usare isequal.....	44
<b>Capitolo 11: dizionari.....</b>	<b>46</b>
Examples.....	46
Usare i dizionari.....	46
<b>Capitolo 12: Elaborazione parallela.....</b>	<b>47</b>
Examples.....	47
pmap.....	47
@parallelo.....	47

@spawn e @spawnat .....	49
Quando usare @parallel vs pmap .....	51
@async e @sync .....	52
Aggiunta di lavoratori .....	56
<b>Capitolo 13: Enums .....</b>	<b>57</b>
Sintassi .....	57
Osservazioni .....	57
Examples .....	57
Definizione di un tipo enumerato .....	57
Usare simboli come leggere enumerazioni .....	59
<b>Capitolo 14: espressioni .....</b>	<b>61</b>
Examples .....	61
Introduzione alle espressioni .....	61
Creazione di espressioni .....	61
Campi di oggetti espressione .....	63
Interpolazione ed espressioni .....	65
Riferimenti esterni sulle espressioni .....	65
<b>Capitolo 15: funzioni .....</b>	<b>67</b>
Sintassi .....	67
Osservazioni .....	67
Examples .....	67
Piazzare un numero .....	67
Funzioni ricorsive .....	68
Ricorsione semplice .....	68
Lavorare con gli alberi .....	68
Introduzione alla spedizione .....	68
Argomenti opzionali .....	69
Invio parametrico .....	70
Scrivere codice generico .....	71
Fattoriale imperativo .....	72
Funzioni anonime .....	73
Sintassi della freccia .....	73

Sintassi multilinea.....	73
Do la sintassi del blocco.....	74
<b>Capitolo 16: Funzioni di ordine superiore.....</b>	<b>75</b>
Sintassi.....	75
Osservazioni.....	75
Examples.....	75
Funziona come argomenti.....	75
Mappare, filtrare e ridurre.....	76
<b>Capitolo 17: Ingresso.....</b>	<b>78</b>
Sintassi.....	78
Parametri.....	78
Examples.....	78
Lettura di una stringa da input standard.....	78
Lettura di numeri da input standard.....	80
Lettura dei dati da un file.....	82
Lettura di stringhe o byte.....	82
Leggere i dati strutturati.....	83
<b>Capitolo 18: iterabili.....</b>	<b>84</b>
Sintassi.....	84
Parametri.....	84
Examples.....	84
Nuovo tipo iterabile.....	84
Combinare Iterables pigri.....	86
Affetta un po 'iterabile.....	86
Pigramente spostare un iterable circolare.....	87
Fare una tabella di moltiplicazione.....	87
Liste con valutazione lenta.....	88
<b>Capitolo 19: JSON.....</b>	<b>90</b>
Sintassi.....	90
Osservazioni.....	90
Examples.....	90

Installazione di JSON.jl .....	90
Parsing JSON .....	90
Serializzazione JSON .....	91
<b>Capitolo 20: Le tuple .....</b>	<b>93</b>
Sintassi .....	93
Osservazioni .....	93
Examples .....	93
Introduzione a Tuples .....	93
Tipi di tupla .....	95
Dispacciamento di tipi di tuple .....	96
Più valori di ritorno .....	97
<b>Capitolo 21: Lettura di un DataFrame da un file .....</b>	<b>99</b>
Examples .....	99
Lettura di un dataframe da dati separati da delimitatore .....	99
Gestire commenti di commento diversi .....	99
<b>Capitolo 22: Macro di stringa .....</b>	<b>100</b>
Sintassi .....	100
Osservazioni .....	100
Examples .....	100
Utilizzo di macro di stringa .....	100
@b_str .....	101
@big_str .....	101
@doc_str .....	101
@html_str .....	102
@ip_str .....	102
@r_str .....	102
@s_str .....	103
@text_str .....	103
@v_str .....	103
@MIME_str .....	103
Simboli che non sono identificativi legali .....	103
Implementazione dell'interpolazione in una macro di stringhe .....	104

Analisi manuale.....	104
Julia analizza.....	105
Macro di comando.....	105
<b>Capitolo 23: mentre cicli.....</b>	<b>107</b>
Sintassi.....	107
Osservazioni.....	107
Examples.....	107
Sequenza di Collatz.....	107
Esegui una volta prima di testare la condizione.....	108
Ricerca per ampiezza.....	108
<b>Capitolo 24: metaprogrammazione.....</b>	<b>111</b>
Sintassi.....	111
Osservazioni.....	111
Examples.....	111
Reimplementare la macro @show.....	111
Fino al ciclo.....	112
QuoteNode, Meta.quot ed Expr (: quota).....	113
La differenza tra Meta.quot e QuoteNode , spiegata.....	114
Che dire di Expr (: citazione)?.....	118
Guida.....	119
<b>Bit e bob di Metaprogramming di .....</b>	<b>119</b>
Simbolo.....	119
Expr (AST).....	120
Expr multilinea usando la quote.....	121
quote una quote.....	122
\$ E : (...) sono in qualche modo inversi l'uno dall'altro?.....	122
\$ foo lo stesso di eval( foo ) ?.....	123
<b>macro s.....</b>	<b>123</b>
Facciamo la nostra macro @show :.....	123
expand per abbassare un Expr.....	123
esc().....	124



Esempio: swap macro per illustrare esc()	124
Esempio: until macro	126
Interpolazione e assert macro	127
Un divertente trucco per usare {} per i blocchi	127
<b>AVANZATE</b>	<b>128</b>
La macro di Scott:	129
<b>junk / unprocessed ...</b>	<b>130</b>
visualizza / scarica una macro	130
Come capire eval(Symbol("@M")) ?	131
Perché non code_typed visualizzati i parametri di visualizzazione code_typed ?	131
???	133
Modulo Gotcha	134
Python `dict` / JSON come sintassi per i letterali `Dict`	134
introduzione	134
Definizione macro	135
uso	135
cattivo uso	136
<b>Capitolo 25: moduli</b>	<b>137</b>
Sintassi	137
Examples	137
Avvolgere il codice in un modulo	137
Utilizzo dei moduli per organizzare i pacchetti	138
<b>Capitolo 26: Normalizzazione delle stringhe</b>	<b>139</b>
Sintassi	139
Parametri	139
Examples	139
Confronto tra stringhe senza distinzione tra maiuscole e minuscole	139
Confronto tra stringhe insensibili ai diacritici	139
<b>Capitolo 27: Pacchi</b>	<b>141</b>
Sintassi	141
Parametri	141

Examples.....	141
Installa, usa e rimuovi un pacchetto registrato.....	141
Scopri un altro ramo o versione.....	142
Installa un pacchetto non registrato.....	143
<b>Capitolo 28: per loop.....</b>	<b>144</b>
Sintassi.....	144
Osservazioni.....	144
Examples.....	144
Fizz Buzz.....	144
Trova il fattore primo più piccolo.....	145
Iterazione multidimensionale.....	145
Riduzione e loop paralleli.....	146
<b>Capitolo 29: regex.....</b>	<b>147</b>
Sintassi.....	147
Parametri.....	147
Examples.....	147
Regalali letterali.....	147
Trovare partite.....	147
Cattura gruppi.....	148
<b>Capitolo 30: REPL.....</b>	<b>150</b>
Sintassi.....	150
Osservazioni.....	150
Examples.....	150
Avvia il REPL.....	150
Su sistemi Unix.....	150
Su Windows.....	150
Utilizzo del REPL come calcolatore.....	150
Trattare con la precisione della macchina.....	153
Utilizzo delle modalità REPL.....	153
La modalità Guida.....	153
La modalità Shell.....	154
<b>Capitolo 31: Scripting Shell e Piping.....</b>	<b>155</b>

Sintassi.....	155
Examples.....	155
Utilizzo di Shell dall'interno del REPL.....	155
Shelling fuori dal codice di Julia.....	155
<b>Capitolo 32: stringhe.....</b>	<b>157</b>
Sintassi.....	157
Parametri.....	157
Examples.....	157
Ciao mondo!.....	157
grafemi.....	158
Converti tipi numerici in stringhe.....	159
Interpolazione stringa (inserire il valore definito dalla variabile nella stringa).....	160
Utilizzare sprint per creare stringhe con funzioni IO.....	161
<b>Capitolo 33: sub2ind.....</b>	<b>162</b>
Sintassi.....	162
Parametri.....	162
Osservazioni.....	162
Examples.....	162
Converti gli indici in indici lineari.....	162
Pits & Falls.....	162
<b>Capitolo 34: Tempo.....</b>	<b>164</b>
Sintassi.....	164
Examples.....	164
Ora attuale.....	164
<b>Capitolo 35: Test unitario.....</b>	<b>166</b>
Sintassi.....	166
Osservazioni.....	166
Examples.....	166
Test di un pacchetto.....	166
Scrivere un semplice test.....	167
Scrivere un set di prova.....	167
Test delle eccezioni.....	170

Testing Equality approssimativo a virgola mobile.....	171
<b>Capitolo 36: tipi.....</b>	<b>173</b>
Sintassi.....	173
Osservazioni.....	173
Examples.....	173
Dispacciamento su Tipi.....	173
L'elenco è vuoto?.....	174
Quanto dura la lista?.....	175
Prossimi passi.....	175
Tipi immutabili.....	175
Tipi Singleton.....	175
Tipi di wrapper.....	176
Veri tipi di composito.....	177
<b>Capitolo 37: Tipo di stabilità.....</b>	<b>178</b>
introduzione.....	178
Examples.....	178
Scrivi un codice stabile al tipo.....	178
<b>Titoli di coda.....</b>	<b>179</b>

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [julia-language](#)

It is an unofficial and free Julia Language ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Julia Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to [info@zzzprojects.com](mailto:info@zzzprojects.com)

# Capitolo 1: Iniziare con Julia Language

## Versioni

Versione	Data di rilascio
0.6.0-dev	2017/06/01
0.5.0	2016/09/19
0.4.0	2015/10/08
0.3.0	2014/08/21
0.2.0	2013/11/17
0.1.0	2013/02/14

## Examples

### Ciao mondo!

```
println("Hello, World!")
```

Per eseguire Julia, devi prima ottenere l'interprete dalla pagina di [download](#) del sito [web](#) . La versione stabile corrente è v0.5.0 e questa versione è consigliata per la maggior parte degli utenti. Alcuni sviluppatori di pacchetti o utenti esperti possono scegliere di utilizzare la compilazione notturna, che è molto meno stabile.

Quando hai l'interprete, scrivi il tuo programma in un file chiamato `hello.jl` . Può quindi essere eseguito da un terminale di sistema come:

```
$ julia hello.jl
Hello, World!
```

Julia può anche essere eseguito in modo interattivo, eseguendo il programma `julia` . Dovresti vedere un'intestazione e un prompt, come segue:

```
 _ _ _ _ _ | A fresh approach to technical computing
(_) | ( ) ( ) | Documentation: http://docs.julialang.org
 _ _ _ | | _ _ _ | Type "?help" for help.
| | | | | | | / _ ` | |
| | | | | | ( | | | | Version 0.4.2 (2015-12-06 21:47 UTC)
_/_ | \_/_ ' | | | \_/_ ' | | Official http://julialang.org/ release
|_|/ | | | | | | | | | | x86_64-w64-mingw32
```

```
julia>
```

Puoi eseguire qualsiasi codice Julia in questo [REPL](#) , quindi prova:

```
julia> println("Hello, World!")  
Hello, World!
```

Questo esempio utilizza una [stringa](#) , "Hello, World!" e della [funzione](#) `println` , una delle tante nella libreria standard. Per ulteriori informazioni o assistenza, provare le seguenti fonti:

- Il REPL ha una [modalità](#) di guida integrata per accedere alla documentazione.
- La [documentazione](#) ufficiale è abbastanza completa.
- Stack Overflow ha una piccola ma crescente raccolta di esempi.
- Gli utenti su [Gitter](#) sono felici di aiutare con piccole domande.
- Il principale luogo di discussione online per Julia è il forum Discourse all'indirizzo [discourse.julialang.org](https://discourse.julialang.org) . Le domande più coinvolte dovrebbero essere pubblicate qui.
- Una raccolta di tutorial e libri può essere trovata [qui](#) .

Leggi Iniziare con Julia Language online: <https://riptutorial.com/it/julia-lang/topic/485/iniziare-con-julia-language>

---

# Capitolo 2: @goto e @label

## Sintassi

- Etichetta @goto
- etichetta @label

## Osservazioni

L'uso eccessivo o inappropriato del flusso di controllo avanzato rende il codice difficile da leggere. @goto o i suoi equivalenti in altre lingue, se usati in modo improprio, portano a codice spaghetti illeggibile.

Simile alle lingue come C, non si può saltare tra le funzioni di Julia. Ciò significa anche che @goto non è possibile al livello più alto; Funzionerà solo all'interno di una funzione. Inoltre, non si può saltare da una funzione interiore alla sua funzione esterna, o da una funzione esterna a una funzione interiore.

## Examples

### Convalida dell'input

Sebbene non siano tradizionalmente considerati loop, i macro @goto e @label possono essere utilizzati per un flusso di controllo più avanzato. Un caso d'uso è quando il fallimento di una parte dovrebbe portare al nuovo tentativo di un'intera funzione, spesso utile nella convalida dell'input:

```
function getsequence()
    local a, b

    @label start
        print("Input an integer: ")
        try
            a = parse{Int, readline()}
        catch
            println("Sorry, that's not an integer.")
            @goto start
        end

        print("Input a decimal: ")
        try
            b = parse{Float64, readline()}
        catch
            println("Sorry, that doesn't look numeric.")
            @goto start
        end

        a, b
    end
end
```



Tuttavia, questo caso d'uso è spesso più chiaro usando la ricorsione:

```
function getsequence()
    local a, b

    print("Input an integer: ")
    try
        a = parse{Int, readline()}
    catch
        println("Sorry, that's not an integer.")
        return getsequence()
    end

    print("Input a decimal: ")
    try
        b = parse{Float64, readline()}
    catch
        println("Sorry, that doesn't look numeric.")
        return getsequence()
    end

    a, b
end
```

Sebbene entrambi gli esempi facciano la stessa cosa, il secondo è più facile da capire. Tuttavia, il primo è più performante (perché evita la chiamata ricorsiva). Nella maggior parte dei casi, il costo della chiamata non ha importanza; ma in situazioni limitate, la prima forma è accettabile.

## Pulizia errore

In linguaggi come C, l'istruzione `@goto` viene spesso utilizzata per garantire che una funzione pulisca le risorse necessarie, anche in caso di errore. Questo è meno importante in Julia, perché le eccezioni e i blocchi `try - finally` vengono spesso utilizzati.

Tuttavia, è possibile che il codice Julia si interfaccia con il codice C e le API C, quindi a volte le funzioni devono ancora essere scritte come il codice C. L'esempio seguente è ideato, ma dimostra un caso d'uso comune. Il codice Julia chiamerà `libc.malloc` per allocare memoria (simula una chiamata API C). Se non tutte le allocazioni sono riuscite, allora la funzione dovrebbe liberare le risorse ottenute finora; in caso contrario, viene restituita la memoria allocata.

```
using Base.Libc
function allocate_some_memory()
    mem1 = malloc(100)
    mem1 == C_NULL && @goto fail
    mem2 = malloc(200)
    mem2 == C_NULL && @goto fail
    mem3 = malloc(300)
    mem3 == C_NULL && @goto fail
    return mem1, mem2, mem3

@label fail
    free(mem1)
    free(mem2)
    free(mem3)
end
```

Leggi @goto e @label online: <https://riptutorial.com/it/julia-lang/topic/5564/-goto-e--label>

# Capitolo 3: Aritmetica

## Sintassi

- $+x$
- $-X$
- $a + b$
- $a - b$
- $a * b$
- $a / b$
- $a ^ b$
- $a \% b$
- $4a$
- $\text{sqrt}(a)$

## Examples

### Formula quadratica

Julia usa operatori binari simili per operazioni aritmetiche di base come la matematica o altri linguaggi di programmazione. La maggior parte degli operatori può essere scritta in notazione infix (ovvero, posizionata tra i valori calcolati). Julia ha un ordine di operazioni che corrisponde alla convenzione comune in matematica.

Ad esempio, il codice seguente implementa la [formula quadratica](#), che dimostra rispettivamente gli operatori  $+$ ,  $-$ ,  $*$  e  $/$  per addizione, sottrazione, moltiplicazione e divisione. Viene anche mostrata la *moltiplicazione implicita*, in cui un numero può essere posizionato direttamente prima di un simbolo per indicare la moltiplicazione; cioè,  $4a$  significa lo stesso di  $4 * a$ .

```
function solvequadratic(a, b, c)
    d = sqrt(b^2 - 4a*c)
    (-b - d) / 2a, (-b + d) / 2a
end
```

Uso:

```
julia> solvequadratic(1, -2, -3)
(-1.0, 3.0)
```

### Setaccio di Eratostene

L'operatore rimanente in Julia è l'operatore  $\%$ . Questo operatore si comporta in modo simile alla  $\%$  in lingue come C e C++.  $a \% b$  è il resto firmato rimasto dopo aver diviso  $a$  da  $b$ .

Questo operatore è molto utile per l'implementazione di determinati algoritmi, come la seguente

implementazione del [setaccio di Eratostene](#) .

```
iscopprime(P, i) = !any(x -> i % x == 0, P)

function sieve(n)
    P = Int[]
    for i in 2:n
        if iscopprime(P, i)
            push!(P, i)
        end
    end
    P
end
```

Uso:

```
julia> sieve(20)
8-element Array{Int64,1}:
 2
 3
 5
 7
11
13
17
19
```

## Matrix Arithmetic

Julia usa i significati matematici standard delle operazioni aritmetiche quando applicato alle matrici. A volte, invece, si preferiscono operazioni elementwise. Questi sono contrassegnati da un punto ( `.` ) Che precede l'operatore da eseguire elementwise. (Nota che le operazioni elementwise spesso non sono efficienti come i loop).

## Le somme

L'operatore `+` sulle matrici è una somma matrice. È simile a una somma elementwise, ma non trasmette la forma. Cioè, se `A` e `B` hanno la stessa forma, allora `A + B` è lo stesso di `A .+ B` ; altrimenti, `A + B` è un errore, mentre `A .+ B` potrebbe non essere necessariamente.

```
julia> A = [1 2
            3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [5 6
            7 8]
2×2 Array{Int64,2}:
 5  6
 7  8

julia> A + B
2×2 Array{Int64,2}:
```

```

6    8
10   12

julia> A .+ B
2×2 Array{Int64,2}:
 6    8
10   12

julia> C = [9, 10]
2-element Array{Int64,1}:
 9
10

julia> A + C
ERROR: DimensionMismatch("dimensions must match")
 in promote_shape(::Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}, ::Tuple{Base.OneTo{Int64}}) at
 ./operators.jl:396
 in promote_shape(::Array{Int64,2}, ::Array{Int64,1}) at ./operators.jl:382
 in _elementwise(::Base.#+, ::Array{Int64,2}, ::Array{Int64,1}, ::Type{Int64}) at
 ./arraymath.jl:61
 in +(::Array{Int64,2}, ::Array{Int64,1}) at ./arraymath.jl:53

julia> A .+ C
2×2 Array{Int64,2}:
10   11
13   14

```

Analogamente, `-` calcola una differenza matrice. Sia `+` che `-` possono anche essere usati come operatori unari.

## Prodotti

L'operatore `*` sulle matrici è il [prodotto matrice](#) (non il prodotto elementwise). Per un prodotto elementwise, utilizzare l'operatore `.*`. Confronta (usando le stesse matrici come sopra):

```

julia> A * B
2×2 Array{Int64,2}:
19   22
43   50

julia> A .* B
2×2 Array{Int64,2}:
 5   12
21   32

```

## potenze

L'operatore `^` calcola l' [esponenziazione della matrice](#) . L'esponenziazione della matrice può essere utile per calcolare rapidamente i valori di determinate ricorrenze. Ad esempio, i [numeri di Fibonacci](#) possono essere generati dall'espressione della [matrice](#)

```
fib(n) = (BigInt[1 1; 1 0]^n)[2]
```

Come al solito, l'operatore `.` <sup>^</sup> Può essere utilizzato laddove l'esponenziazione elementwise è l'operazione desiderata.

Leggi Aritmetica online: <https://riptutorial.com/it/julia-lang/topic/3848/aritmetica>

# Capitolo 4: Array

## Sintassi

- [1,2,3]
- [1 2 3]
- [1 2 3; 4 5 6; 7 8 9]
- Array (type, dims ...)
- quelli (tipo, dim. ...)
- zeri (tipo, dim. ...)
- trues (type, dims ...)
- falsi (tipo, oscuramento ...)
- spingere! (A, x)
- pop! (A)
- unshift! (A, x)
- spostare! (A)

## Parametri

parametri	Osservazioni
Per	<code>push!(A, x)</code> , <code>unshift!(A, x)</code>
A	La matrice da aggiungere a.
x	L'elemento da aggiungere all'array.

## Examples

### Costruzione manuale di un array semplice

Si può inizializzare manualmente una matrice di Julia usando la sintassi delle parentesi quadre:

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

La prima riga dopo il comando mostra la dimensione dell'array che hai creato. Mostra anche il tipo dei suoi elementi e la sua dimensionalità (int questo caso `Int64` e `1` , in modo ripetitivo). Per un array bidimensionale, puoi usare spazi e punto e virgola:

```
julia> x = [1 2 3; 4 5 6]
2x3 Array{Int64,2}:
```

```
1 2 3
4 5 6
```

Per creare un array non inizializzato, puoi utilizzare il metodo `Array(type, dims...)` :

```
julia> Array{Int64, 3, 3}
3x3 Array{Int64,2}:
 0  0  0
 0  0  0
 0  0  0
```

Le funzioni `zeros` , `ones` , `true`s , `false`s hanno metodi che si comportano esattamente allo stesso modo, ma producono matrici piene di `0.0` , `1.0` , `True` o `False` , rispettivamente.

## Tipi di matrice

In Julia, gli array hanno tipi parametrizzati da due variabili: un tipo `T` e una dimensionalità `D` (`Array{T, D}` ). Per un array 1-dimensionale di numeri interi, il tipo è:

```
julia> x = [1, 2, 3];
julia> typeof(x)
Array{Int64, 1}
```

Se l'array è una matrice bidimensionale, `D` uguale a 2:

```
julia> x = [1 2 3; 4 5 6; 7 8 9]
julia> typeof(x)
Array{Int64, 2}
```

Il tipo di elemento può anche essere tipi astratti:

```
julia> x = [1 2 3; 4 5 "6"; 7 8 9]
3x3 Array{Any,2}:
 1  2  3
 4  5  "6"
 7  8  9
```

Qui `Any` (un tipo astratto) è il tipo dell'array risultante.

## Specifica dei tipi durante la creazione di matrici

Quando creiamo una matrice nel modo descritto sopra, Julia farà del suo meglio per dedurre il tipo corretto che potremmo volere. Negli esempi iniziali di cui sopra, abbiamo inserito input che sembravano numeri interi, e quindi Julia si è impostato di default sul tipo di `Int64` predefinito. A volte, tuttavia, potremmo voler essere più specifici. Nell'esempio seguente, specifichiamo che vogliamo che il tipo sia invece `Int8` :

```
x1 = Int8[1 2 3; 4 5 6; 7 8 9]
typeof(x1)  ## Array{Int8,2}
```



Potremmo anche specificare il tipo come qualcosa come `Float64`, anche se scriviamo gli input in un modo che potrebbe altrimenti essere interpretato come numeri interi per impostazione predefinita (ad esempio scrivendo `1` anziché `1.0`). per esempio

```
x2 = Float64[1 2 3; 4 5 6; 7 8 9]
```

## Array di array - Proprietà e costruzione

In Julia, puoi avere una matrice che contiene altri oggetti di tipo `Array`. Considera i seguenti esempi di inizializzazione di vari tipi di array:

```
A = Array{Float64}(10,10) # A single Array, dimensions 10 by 10, of Float64 type objects

B = Array{Array}(10,10,10) # A 10 by 10 by 10 Array. Each element is an Array of unspecified
type and dimension.

C = Array{Array{Float64}}(10) ## A length 10, one-dimensional Array. Each element is an
Array of Float64 type objects but unspecified dimensions

D = Array{Array{Float64, 2}}(10) ## A length 10, one-dimensional Array. Each element of is
an 2 dimensional array of Float 64 objects
```

Si consideri ad esempio, le differenze tra `C` e `D` qui:

```
julia> C[1] = rand(3)
3-element Array{Float64,1}:
 0.604771
 0.985604
 0.166444

julia> D[1] = rand(3)
ERROR: MethodError:
```

`rand(3)` produce un oggetto di tipo `Array{Float64,1}`. Poiché l'unica specifica per gli elementi di `C` è che siano matrici con elementi di tipo `Float64`, ciò rientra nella definizione di `C`. Ma, per `D` abbiamo specificato che gli elementi devono essere matrici bidimensionali. Quindi, dato che `rand(3)` non produce un array bidimensionale, non possiamo usarlo per assegnare un valore a un elemento specifico di `D`.

## Specificare le dimensioni specifiche delle matrici all'interno di una matrice

Sebbene possiamo specificare che una matrice conserverà elementi che sono di tipo `Array`, e possiamo specificare che, per esempio quegli elementi dovrebbero essere matrici bidimensionali, non possiamo specificare direttamente le dimensioni di quegli elementi. Ad esempio non possiamo specificare direttamente che vogliamo una matrice che contiene 10 matrici, ognuna delle quali è 5,5. Possiamo vedere questo dalla sintassi per la funzione `Array()` utilizzata per costruire una matrice:

### **Array {T} (dim)**

costruisce una matrice densa non inizializzata con il tipo di elemento `T`. `dims` può

essere una tupla o una serie di argomenti interi. Anche la matrice di sintassi (T, dims) è disponibile, ma deprecata.

Il tipo di una matrice in Julia racchiude il numero delle dimensioni ma non le dimensioni di quelle dimensioni. Pertanto, non c'è spazio in questa sintassi per specificare le dimensioni precise. Tuttavia, un effetto simile potrebbe essere ottenuto usando una comprensione di matrice:

```
E = [Array{Float64}(5,5) for idx in 1:10]
```

Nota: questa documentazione rispecchia la seguente [risposta SO](#)

## Inizializza una matrice vuota

Possiamo usare `[]` per creare una matrice vuota in Julia. L'esempio più semplice sarebbe:

```
A = [] # 0-element Array{Any,1}
```

Le matrici di tipo `Any` generalmente non funzionano come quelle con un tipo specificato. Quindi, ad esempio, possiamo usare:

```
B = Float64[] ## 0-element Array{Float64,1}
C = Array{Float64}[] ## 0-element Array{Array{Float64,N},1}
D = Tuple{Int, Int}[] ## 0-element Array{Tuple{Int64,Int64},1}
```

Vedi [Inizializza una matrice vuota di tuple in Julia](#) come fonte dell'ultimo esempio.

## Vettori

I vettori sono matrici unidimensionali e supportano principalmente la stessa interfaccia delle loro controparti multidimensionali. Tuttavia, i vettori supportano anche operazioni aggiuntive.

Prima di tutto, nota che `Vector{T}` dove `T` è un tipo indica lo stesso di `Array{T,1}`.

```
julia> Vector{Int}
Array{Int64,1}

julia> Vector{Float64}
Array{Float64,1}
```

Si legge l' `Array{Int64,1}` come "matrice unidimensionale di `Int64`".

A differenza degli array multidimensionali, i vettori possono essere ridimensionati. Gli elementi possono essere aggiunti o rimossi dalla parte anteriore o posteriore del vettore. Queste operazioni sono tutti [tempi di ammortamento costanti](#).

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> push!(A, 4)
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> A
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> pop!(A)
4

julia> A
3-element Array{Int64,1}:
 1
 2
 3

julia> unshift!(A, 0)
4-element Array{Int64,1}:
 0
 1
 2
 3

julia> A
4-element Array{Int64,1}:
 0
 1
 2
 3

julia> shift!(A)
0

julia> A
3-element Array{Int64,1}:
 1
 2
 3
```

Come è convenzione, ognuna di queste funzioni `push!`, `pop!`, `unshift!` e `shift!` termina con un punto esclamativo per indicare che stanno mutando il loro argomento. Le funzioni `push!` e `unshift!` restituisce l'array, mentre `pop!` e `shift!` restituisce l'elemento rimosso.

## Concatenazione

Spesso è utile costruire matrici con matrici più piccole.

## Concatenazione orizzontale

Le matrici (e i vettori, che sono trattati come vettori di colonne) possono essere concatenati orizzontalmente usando la funzione `hcat`.

```
julia> hcat([1 2; 3 4], [5 6 7; 8 9 10], [11, 12])
2×6 Array{Int64,2}:
 1  2  5  6  7 11
 3  4  8  9 10 12
```

È disponibile la sintassi della comodità, utilizzando la notazione e gli spazi tra parentesi quadre:

```
julia> [[1 2; 3 4] [5 6 7; 8 9 10] [11, 12]]
2×6 Array{Int64,2}:
 1  2  5  6  7 11
 3  4  8  9 10 12
```

Questa notazione può corrispondere strettamente alla notazione per le matrici di blocchi utilizzate nell'algebra lineare:

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [5 6; 7 8]
2×2 Array{Int64,2}:
 5  6
 7  8

julia> [A B]
2×4 Array{Int64,2}:
 1  2  5  6
 3  4  7  8
```

Si noti che non è possibile concatenare orizzontalmente una singola matrice usando la sintassi `[]`, poiché ciò creerebbe invece un vettore a un elemento di matrici:

```
julia> [A]
1-element Array{Array{Int64,2},1}:
 [1 2; 3 4]
```

## Concatenazione verticale

La concatenazione verticale è come la concatenazione orizzontale, ma nella direzione verticale. La funzione per la concatenazione verticale è `vcat`.

```
julia> vcat([1 2; 3 4], [5 6; 7 8; 9 10], [11 12])
6×2 Array{Int64,2}:
 1  2
 3  4
 5  6
 7  8
 9 10
11 12
```

In alternativa, la notazione della parentesi quadra può essere utilizzata con il punto e virgola ; come delimitatore:

```
julia> [[1 2; 3 4]; [5 6; 7 8; 9 10]; [11 12]]
6×2 Array{Int64,2}:
 1  2
 3  4
 5  6
 7  8
 9 10
11 12
```

Anche i vettori possono essere concatenati verticalmente; il risultato è un vettore:

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [4, 5]
2-element Array{Int64,1}:
 4
 5

julia> [A; B]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

La concatenazione orizzontale e verticale può essere combinata:

```
julia> A = [1 2
           3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [5 6 7]
1×3 Array{Int64,2}:
 5  6  7

julia> C = [8, 9]
2-element Array{Int64,1}:
 8
 9

julia> [A C; B]
3×3 Array{Int64,2}:
 1  2  8
 3  4  9
 5  6  7
```

Leggi Array online: <https://riptutorial.com/it/julia-lang/topic/5437/array>

# Capitolo 5: chiusure

## Sintassi

- $x \rightarrow [\text{corpo}]$
- $(x, y) \rightarrow [\text{corpo}]$
- $(xs \dots) \rightarrow [\text{corpo}]$

## Osservazioni

0.4.0

Nelle versioni precedenti di Julia, le chiusure e le funzioni anonime avevano una penalità legata alle prestazioni in fase di esecuzione. Questa penalità è stata eliminata in 0.5.

## Examples

### Composizione funzionale

Possiamo definire una funzione per eseguire la [composizione di funzioni](#) usando la [sintassi della funzione anonima](#) :

```
f ∘ g = x -> f(g(x))
```

Si noti che questa definizione è equivalente a ciascuna delle seguenti definizioni:

```
∘(f, g) = x -> f(g(x))
```

o

```
function ∘(f, g)
    x -> f(g(x))
end
```

ricordando che in Julia,  $f \circ g$  è solo lo zucchero di sintassi per  $\circ(f, g)$  .

Possiamo vedere che questa funzione si compone correttamente:

```
julia> double(x) = 2x
double (generic function with 1 method)

julia> triple(x) = 3x
triple (generic function with 1 method)

julia> const sextuple = double ∘ triple
(::#17) (generic function with 1 method)
```

```
julia> sextuple(1.5)
9.0
```

## 0.5.0

Nella versione v0.5, questa definizione è molto performante. Possiamo esaminare il codice LLVM generato:

```
julia> @code_llvm sextuple(1)

define i64 @"julia_#17_71238"(i64) #0 {
top:
    %1 = mul i64 %0, 6
    ret i64 %1
}
```

È chiaro che le due moltiplicazioni sono state piegate in un'unica moltiplicazione e che questa funzione è il più efficiente possibile.

Come funziona questa funzione di ordine superiore? Crea una cosiddetta [chiusura](#), che consiste non solo nel suo codice, ma tiene anche traccia di determinate variabili dal suo ambito. Tutte le funzioni di Julia che non sono state create nell'ambito di livello superiore sono le chiusure.

## 0.5.0

Si possono ispezionare le variabili chiuse attraverso i campi della chiusura. Ad esempio, vediamo che:

```
julia> (sin ∘ cos).f
sin (generic function with 10 methods)

julia> (sin ∘ cos).g
cos (generic function with 10 methods)
```

## Implementare Currying

Un'applicazione di chiusure è di applicare parzialmente una funzione; cioè, fornire alcuni argomenti ora e creare una funzione che accetta gli argomenti rimanenti. Il [curry](#) è una forma specifica di applicazione parziale.

Iniziamo con la semplice funzione `curry(f, x)` che fornirà il primo argomento di una funzione e aspettiamo ulteriori argomenti in seguito. La definizione è abbastanza semplice:

```
curry(f, x) = (xs...) -> f(x, xs...)
```

Ancora una volta, usiamo la [sintassi della funzione anonima](#), questa volta in combinazione con la sintassi degli argomenti variadici.

Possiamo implementare alcune funzioni di base in stile [tacito](#) (o punto libero) usando questa funzione `curry`.

```
julia> const double = curry(*, 2)
(::#19) (generic function with 1 method)

julia> double(10)
20

julia> const simon_says = curry(println, "Simon: ")
(::#19) (generic function with 1 method)

julia> simon_says("How are you?")
Simon: How are you?
```

Le funzioni mantengono il generismo atteso:

```
julia> simon_says("I have ", 3, " arguments.")
Simon: I have 3 arguments.

julia> double([1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6
```

## Introduzione alle chiusure

Le **funzioni** sono una parte importante della programmazione di Julia. Possono essere definiti direttamente all'interno dei moduli, nel qual caso le funzioni vengono definite di *primo livello*. Ma le funzioni possono anche essere definite all'interno di altre funzioni. Tali funzioni sono chiamate **chiusure**.

Le chiusure catturano le variabili nella loro funzione esterna. Una funzione di primo livello può utilizzare solo variabili globali dal proprio modulo, parametri di funzione o variabili locali:

```
x = 0 # global
function toplevel(y)
    println("x = ", x, " is a global variable")
    println("y = ", y, " is a parameter")
    z = 2
    println("z = ", z, " is a local variable")
end
```

Una chiusura, d'altra parte, può utilizzare tutti quelli oltre alle variabili dalle funzioni esterne che cattura:

```
x = 0 # global
function toplevel(y)
    println("x = ", x, " is a global variable")
    println("y = ", y, " is a parameter")
    z = 2
    println("z = ", z, " is a local variable")

    function closure(v)
        println("v = ", v, " is a parameter")
        w = 3
```



```

println("w = ", w, " is a local variable")
println("x = ", x, " is a global variable")
println("y = ", y, " is a closed variable (a parameter of the outer function)")
println("z = ", z, " is a closed variable (a local of the outer function)")
end
end

```

Se eseguiamo `c = toplevel(10)` , vediamo che il risultato è

```

julia> c = toplevel(10)
x = 0 is a global variable
y = 10 is a parameter
z = 2 is a local variable
(::closure) (generic function with 1 method)

```

Si noti che l'espressione di coda di questa funzione è una funzione in sé; cioè, una chiusura. Possiamo chiamare la chiusura `c` come se fosse un'altra funzione:

```

julia> c(11)
v = 11 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 10 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)

```

Si noti che `c` ha ancora accesso alle variabili `y` e `z` dal `toplevel` chiamata - anche se `toplevel` è già tornato! Ogni chiusura, anche quelli restituiti dalla stessa funzione, si chiude su diverse variabili. Possiamo chiamare di nuovo `toplevel`

```

julia> d = toplevel(20)
x = 0 is a global variable
y = 20 is a parameter
z = 2 is a local variable
(::closure) (generic function with 1 method)

julia> d(22)
v = 22 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 20 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)

julia> c(22)
v = 22 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 10 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)

```

Si noti che nonostante `d` e `c` abbiano lo stesso codice e che vengano passati gli stessi argomenti, il loro output è diverso. Sono chiusure distinte.

Leggi chiusure online: <https://riptutorial.com/it/julia-lang/topic/5724/chiusure>

# Capitolo 6: combinatori

## Osservazioni

Sebbene i combinatori abbiano un uso pratico limitato, sono uno strumento utile nell'educazione per capire in che modo la programmazione è fondamentalmente legata alla logica e in che modo blocchi elementari possono combinarsi per creare un comportamento molto complesso. Nel contesto di Julia, imparare a creare e usare i combinatori rafforzerà la comprensione di come programmare in uno stile funzionale in Julia.

## Examples

### Il Y o Z Combinator

Anche se Julia non è un linguaggio puramente funzionale, ha il pieno supporto per molti dei capisaldi della programmazione funzionale: prima classe [funzioni](#), scope lessicale, e [chiusure](#).

Il [combinatore a virgola fissa](#) è un combinatore chiave nella programmazione funzionale. Poiché Julia ha una semantica di [valutazione entusiasta](#) (come molti altri linguaggi funzionali, tra cui Scheme, di cui Julia è fortemente ispirata), il combinatore Y originale di Curry non funzionerà immediatamente:

```
Y(f) = (x -> f(x(x))) (x -> f(x(x)))
```

Tuttavia, un parente stretto del combinatore Y, il combinatore Z, funzionerà davvero:

```
Z(f) = x -> f(Z(f), x)
```

Questo combinatore accetta una funzione e restituisce una funzione che quando viene chiamata con argomento  $x$ , viene passata a se stessa e  $x$ . Perché sarebbe utile che una funzione venga approvata da sola? Ciò consente la ricorsione senza in realtà fare riferimento al nome della funzione!

```
fact(f, x) = x == 0 ? 1 : x * f(x)
```

Quindi, `Z(fact)` diventa un'implementazione ricorsiva della funzione fattoriale, nonostante nessuna ricorsione sia visibile in questa definizione di funzione. (La ricorsione è evidente nella definizione del combinatore `Z`, ovviamente, ma ciò è inevitabile in un linguaggio desideroso.) Possiamo verificare che la nostra funzione funzioni effettivamente:

```
julia> Z(fact)(10)
3628800
```

Non solo, ma è veloce quanto possiamo aspettarci da un'implementazione ricorsiva. Il codice LLVM dimostra che il risultato è compilato in un semplice vecchio ramo, sottrarre, chiamare e

moltiplicare:

```
julia> @code_llvm Z(fact)(10)

define i64 @"julia_#1_70252"(i64) #0 {
top:
    %1 = icmp eq i64 %0, 0
    br i1 %1, label %L11, label %L8

L8:                                     ; preds = %top
    %2 = add i64 %0, -1
    %3 = call i64 @"julia_#1_70060"(i64 %2) #0
    %4 = mul i64 %3, %0
    br label %L11

L11:                                   ; preds = %top, %L8
    %"#temp#.0" = phi i64 [ %4, %L8 ], [ 1, %top ]
    ret i64 %"#temp#.0"
}
```

## Il sistema di combinazione SKI

Il [sistema combinatore SKI](#) è sufficiente per rappresentare qualsiasi termine di calcolo lambda. (In pratica, naturalmente, le astrazioni lambda esplodono in dimensioni esponenziali quando vengono tradotte in SKI.) A causa della semplicità del sistema, l'implementazione dei combinatori S, K e I è straordinariamente semplice:

## Una traduzione diretta dal Lambda Calculus

```
const S = f -> g -> z -> f(z) (g(z))
const K = x -> y -> x
const I = x -> x
```

Possiamo confermare, utilizzando il sistema di [test delle unità](#), che ciascun combinatore ha il comportamento previsto.

Il combinatore I è più semplice da verificare; dovrebbe restituire il valore dato invariato:

```
using Base.Test
@test I(1) === 1
@test I(I) === I
@test I(S) === S
```

Il combinatore K è anche abbastanza semplice: dovrebbe scartare il suo secondo argomento.

```
@test K(1) (2) === 1
@test K(S) (I) === S
```

Il combinatore S è il più complesso; il suo comportamento può essere riassunto applicando i primi due argomenti al terzo argomento, applicando il primo risultato al secondo. Possiamo facilmente testare il combinatore S testando alcune delle sue forme al curry.  $S(K)$ , per esempio, dovrebbe

semplicemente restituire il secondo argomento e scartare il suo primo, come vediamo succede:

```
@test S(K) (S) (K) === K
@test S(K) (S) (I) === I
```

$S(I) (I)$  dovrebbe applicare la sua argomentazione a se stessa:

```
@test S(I) (I) (I) === I
@test S(I) (I) (K) === K(K)
@test S(I) (I) (S(I)) === S(I) (S(I))
```

$S(K(S(I))) (K)$  applica il suo secondo argomento al primo:

```
@test S(K(S(I))) (K) (I) (I) === I
@test S(K(S(I))) (K) (K) (S(K)) === S(K) (K)
```

Il combinatore  $I$  descritto sopra ha un nome in `Base` Julia standard: `identity`. Pertanto, avremmo potuto riscrivere le definizioni di cui sopra con la seguente definizione alternativa di  $I$ :

```
const I = identity
```

## Mostrando Combinatori SKI

Una debolezza con l'approccio di cui sopra è che le nostre funzioni non mostrano quanto vorremmo. Potremmo sostituire

```
julia> S
(::#3) (generic function with 1 method)

julia> K
(::#9) (generic function with 1 method)

julia> I
(::#13) (generic function with 1 method)
```

con alcuni display più informativi? La risposta è sì! Riavvia il REPL, e questa volta definiamo come ogni funzione deve essere mostrata:

```
const S = f -> g -> z -> f(z) (g(z));
const K = x -> y -> x;
const I = x -> x;
for f in (:S, :K, :I)
    @eval Base.show(io::IO, ::typeof($f)) = print(io, $(string(f)))
    @eval Base.show(io::IO, ::MIME"text/plain", ::typeof($f)) = show(io, $f)
end
```

È importante non mostrare nulla finché non abbiamo finito di definire le funzioni. Altrimenti, rischiamo di invalidare la cache dei metodi, e i nostri nuovi metodi non sembrano avere effetto immediato. Questo è il motivo per cui abbiamo inserito il punto e virgola nelle definizioni precedenti. Il punto e virgola sopprime l'output di REPL.

Questo rende le funzioni ben visualizzate:

```
julia> S
S

julia> K
K

julia> I
I
```

Tuttavia, ci sono ancora problemi quando proviamo a visualizzare una chiusura:

```
julia> S(K)
(::#2) (generic function with 1 method)
```

Sarebbe più bello mostrarlo come `S(K)`. Per fare ciò, dobbiamo sfruttare il fatto che le chiusure hanno i loro tipi individuali. Siamo in grado di accedere a questi tipi e aggiungere loro dei metodi attraverso la riflessione, usando `typeof` e il campo `primary` del tipo. Riavvia di nuovo REPL; faremo ulteriori cambiamenti:

```
const S = f -> g -> z -> f(z) (g(z));
const K = x -> y -> x;
const I = x -> x;
for f in (:S, :K, :I)
    @eval Base.show(io::IO, ::typeof($f)) = print(io, $(string(f)))
    @eval Base.show(io::IO, ::MIME"text/plain", ::typeof($f)) = show(io, $f)
end
Base.show(io::IO, s::typeof(S(I)).name.primary) = print(io, "S(", s.f, ')')
Base.show(io::IO, s::typeof(S(I)(I)).name.primary) =
    print(io, "S(", s.f, ')', '(', s.g, ')')
Base.show(io::IO, k::typeof(K(I)).name.primary) = print(io, "K(", k.x, ')')
Base.show(io::IO, ::MIME"text/plain", f::Union{
    typeof(S(I)).name.primary,
    typeof(S(I)(I)).name.primary,
    typeof(K(I)).name.primary
}) = show(io, f)
```

E ora, finalmente, le cose si mostrano come vorremmo che:

```
julia> S(K)
S(K)

julia> S(K)(I)
S(K)(I)

julia> K
K

julia> K(I)
K(I)

julia> K(I)(K)
I
```

Leggi combinatori online: <https://riptutorial.com/it/julia-lang/topic/5758/combinatori>

# Capitolo 7: Compatibilità tra versioni

## Sintassi

- usando `Compat`
- `Compat.String`
- `Compat.UTF8String`
- `@compat f. (x, y)`

## Osservazioni

A volte è molto difficile ottenere una nuova sintassi per giocare bene con più versioni. Poiché Julia è ancora in fase di sviluppo attivo, è spesso utile semplicemente abbandonare il supporto per le versioni precedenti e invece indirizzare solo quelle più recenti.

## Examples

### Numeri di versione

Julia ha un'implementazione integrata della [versione semantica](#) esposta tramite il tipo `VersionNumber`.

Per costruire un oggetto `VersionNumber` come letterale, è possibile utilizzare la [macro string](#) `@v_str`:

```
julia> vers = v"1.2.0"  
v"1.2.0"
```

In alternativa, si può chiamare il costruttore `VersionNumber`; si noti che il costruttore accetta fino a cinque argomenti, ma tutti tranne il primo sono opzionali.

```
julia> vers2 = VersionNumber(1, 1)  
v"1.1.0"
```

I numeri di versione possono essere confrontati utilizzando [operatori di confronto](#) e quindi possono essere ordinati:

```
julia> vers2 < vers  
true  
  
julia> v"1" < v"0"  
false  
  
julia> sort([v"1.0.0", v"1.0.0-dev.100", v"1.0.1"])  
3-element Array{VersionNumber,1}:  
v"1.0.0-dev.100"  
v"1.0.0"  
v"1.0.1"
```

I numeri di versione sono utilizzati in diversi punti in tutta Julia. Ad esempio, il `VERSION` costante è un `VersionNumber` :

```
julia> VERSION
v"0.5.0"
```

Questo è comunemente usato per la valutazione del codice condizionale, a seconda della versione di Julia. Ad esempio, per eseguire codice diverso su v0.4 e v0.5, si può fare

```
if VERSION < v"0.5"
    println("v0.5 prerelease, v0.4 or older")
else
    println("v0.5 or newer")
end
```

Ogni [pacchetto](#) installato è anche associato a un numero di versione corrente:

```
julia> Pkg.installed("StatsBase")
v"0.9.0"
```

## Utilizzando Compat.jl

Il [pacchetto Compat.jl](#) consente di utilizzare alcune nuove funzionalità di Julia e la sintassi con le versioni precedenti di Julia. Le sue funzionalità sono documentate nel suo README, ma di seguito viene fornito un riepilogo delle applicazioni utili.

0.5.0

## Tipo di stringa unificato

In Julia v0.4 c'erano molti tipi diversi di [stringhe](#) . Questo sistema è stato considerato eccessivamente complesso e confuso, quindi in Julia v0.5, rimane solo il tipo `String` . `Compat` consente di utilizzare il tipo `String` e il costruttore nella versione 0.4, con il nome `Compat.String` . Ad esempio, questo codice v0.5

```
buf = IOBuffer()
println(buf, "Hello World!")
String(buf) # "Hello World!\n"
```

può essere tradotto direttamente in questo codice, che funziona sia su v0.5 che su v0.4:

```
using Compat
buf = IOBuffer()
println(buf, "Hello World!")
Compat.String(buf) # "Hello World!\n"
```

Nota che ci sono alcuni avvertimenti.

- Sulla v0.4, `Compat.String` è tipograficamente su `ByteString` , che è `Union{ASCIIString,`



`UTF8String`}. Pertanto, i tipi con campi `String` non saranno di tipo stabile. In queste situazioni, si consiglia `Compat.UTF8String`, poiché significa `String` su v0.5 e `UTF8String` su v0.4, entrambi tipi concreti.

- Si deve fare attenzione a usare `Compat.String` o `import Compat: String`, perché `String` ha un significato su v0.4: è un alias deprecato per `AbstractString`. Un segno che `String` stato utilizzato per errore al posto di `Compat.String` è se in qualsiasi momento compaiono i seguenti avvisi:

```
WARNING: Base.String is deprecated, use AbstractString instead.
likely near no file:0
WARNING: Base.String is deprecated, use AbstractString instead.
likely near no file:0
```

## Sintassi di trasmissione compatta

Julia v0.5 introduce lo zucchero sintattico per la `broadcast`. La sintassi

```
f.(x, y)
```

viene abbassato per `broadcast(f, x, y)`. Esempi di utilizzo di questa sintassi includono `sin.([1, 2, 3])` per prendere il seno di più numeri contemporaneamente.

Su v0.5, la sintassi può essere utilizzata direttamente:

```
julia> sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Tuttavia, se proviamo lo stesso su v0.4, otteniamo un errore:

```
julia> sin.([1.0, 2.0, 3.0])
ERROR: TypeError: getfield: expected Symbol, got Array{Float64,1}
```

Fortunatamente, `Compat` rende questa nuova sintassi utilizzabile anche dalla v0.4. Ancora una volta, aggiungiamo `using Compat`. Questa volta, circondiamo l'espressione con la macro `@compat`:

```
julia> using Compat

julia> @compat sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Leggi Compatibilità tra versioni online: <https://riptutorial.com/it/julia-lang/topic/5832/compatibilita-tra-versioni>

# Capitolo 8: Comprensioni

## Examples

### Array comprehension

## Sintassi di base

La comprensione dell'array di Julia utilizza la seguente sintassi:

```
[expression for element = iterable]
```

Si noti che, come `for` cicli `for`, tutti i `valori` `=`, `in` e `∈` sono accettati per la comprensione.

Ciò equivale approssimativamente alla creazione di un array vuoto e all'utilizzo di un ciclo `for` da `push!` oggetti ad esso.

```
result = []
for element in iterable
    push!(result, expression)
end
```

tuttavia, il tipo di comprensione dell'array è il più stretto possibile, il che è migliore per le prestazioni.

Ad esempio, per ottenere una matrice dei quadrati degli interi da 1 a 10, è possibile utilizzare il codice seguente.

```
squares = [x^2 for x=1:10]
```

Si tratta di un sostituto pulito e conciso per la versione più lunga `for`-loop.

```
squares = []
for x in 1:10
    push!(squares, x^2)
end
```

## Comprensione dell'array condizionale

Prima di Julia 0.5, non c'è modo di usare le condizioni all'interno della comprensione dell'array. Ma non è più vero. In Julia 0.5 possiamo usare le condizioni in condizioni come le seguenti:

```
julia> [x^2 for x in 0:9 if x > 5]
4-element Array{Int64,1}:
 36
 49
 64
```

La fonte dell'esempio sopra può essere trovata [qui](#) .

Se vorremmo usare la comprensione delle liste annidate:

```
julia> [(x,y) for x=1:5 , y=3:6 if y>4 && x>3 ]
4-element Array{Tuple{Int64,Int64},1}:
 (4,5)
 (5,5)
 (4,6)
 (5,6)
```

## Comprensioni di array multidimensionali

Annidati `for` i loop possono essere utilizzati per iterare su un certo numero di iterables unici.

```
result = []
for a = iterable_a
    for b = iterable_b
        push!(result, expression)
    end
end
```

Allo stesso modo, possono essere fornite specifiche di iterazione multiple a una comprensione dell'array.

```
[expression for a = iterable_a, b = iterable_b]
```

Ad esempio, per generare il prodotto cartesiano di `1:3` e `1:2` possibile utilizzare quanto segue.

```
julia> [(x, y) for x = 1:3, y = 1:2]
3×2 Array{Tuple{Int64,Int64},2}:
 (1,1) (1,2)
 (2,1) (2,2)
 (3,1) (3,2)
```

Le comprensioni multidimensionali di array appiattite sono simili, tranne per il fatto che perdono la forma. Per esempio,

```
julia> [(x, y) for x = 1:3 for y = 1:2]
6-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (1, 2)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
```

è una variante appiattita di quanto sopra. La differenza sintattica è che viene aggiunto un ulteriore `for` anziché una virgola.

## Comprensioni del generatore

Le comprensioni dei generatori seguono un formato simile alle comprensioni degli array, ma usano parentesi `()` invece di parentesi quadre `[]`.

```
(expression for element = iterable)
```

Tale espressione restituisce un oggetto `Generator`.

```
julia> (x^2 for x = 1:5)
Base.Generator{UnitRange{Int64},##1#2} (#1,1:5)
```

---

## Argomenti della funzione

Le comprensioni del generatore possono essere fornite come l'unico argomento di una funzione, senza la necessità di un set aggiuntivo di parentesi.

```
julia> join(x^2 for x = 1:5)
"1491625"
```

Tuttavia, se viene fornito più di un argomento, la comprensione del generatore richiede il proprio insieme di parentesi.

```
julia> join(x^2 for x = 1:5, ", ")
ERROR: syntax: invalid iteration specification

julia> join((x^2 for x = 1:5), ", ")
"1, 4, 9, 16, 25"
```

Leggi Comprensioni online: <https://riptutorial.com/it/julia-lang/topic/5477/comprensioni>

# Capitolo 9: Condizionali

## Sintassi

- se cond; corpo; fine
- se cond; corpo; altro; corpo; fine
- se cond; corpo; elseif cond; corpo; altro; fine
- se cond; corpo; elseif cond; corpo; fine
- cond? iftrue: iffalse
- cond && iftrue
- cond || iffalse
- ifelse (cond, iftrue, iffalse)

## Osservazioni

Tutti gli operatori e le funzioni condizionali implicano l'uso di condizioni booleane ( `true` o `false` ). In Julia, il tipo di booleans è `Bool` . A differenza di altre lingue, altri tipi di numeri (come `1` o `0` ), stringhe, matrici e così via *non possono* essere utilizzati direttamente in condizionali.

In genere, si utilizzano le funzioni di predicato (funzioni che restituiscono un `Bool` ) o gli [operatori di confronto](#) nelle condizioni di un operatore o funzione condizionale.

## Examples

### se ... altra espressione

Il condizionale più comune in Julia è l'espressione `if ... else` . Ad esempio, di seguito implementiamo l' [algoritmo Euclideo](#) per calcolare il [massimo comun divisore](#) , usando un condizionale per gestire il caso base:

```
mygcd(a, b) = if a == 0
    abs(b)
else
    mygcd(b % a, a)
end
```

La forma `if ... else` in Julia è in realtà un'espressione e ha un valore; il valore è l'espressione in posizione di coda (ovvero l'ultima espressione) sul ramo che viene preso. Considera il seguente esempio di input:

```
julia> mygcd(0, -10)
10
```

Qui, `a` è `0` e `b` è `-10` . La condizione `a == 0` è `true` , quindi viene preso il primo ramo. Il valore restituito è `abs(b)` che è `10` .

```
julia> mygcd(2, 3)
1
```

Qui,  $a$  è 2 e  $b$  è 3. La condizione  $a == 0$  è falsa, quindi viene preso il secondo ramo e calcoliamo  $\text{mygcd}(b \% a, a)$ , che è  $\text{mygcd}(3 \% 2, 2)$ . L'operatore  $\%$  restituisce il resto quando 3 è diviso per 2, in questo caso 1. Quindi calcoliamo  $\text{mygcd}(1, 2)$ , e questa volta  $a$  è 1 e  $b$  è 2. Ancora una volta,  $a == 0$  è falso, quindi viene preso il secondo ramo e calcoliamo  $\text{mygcd}(b \% a, a)$ , che è  $\text{mygcd}(0, 1)$ . Questa volta,  $a == 0$ , infine, viene restituito  $\text{abs}(b)$ , che fornisce il risultato 1.

## se ... altra affermazione

```
name = readline()
if startswith(name, "A")
    println("Your name begins with A.")
else
    println("Your name does not begin with A.")
end
```

Qualsiasi espressione, come l'espressione `if ... else`, può essere posta in posizione statement. Questo ignora il suo valore ma esegue comunque l'espressione per i suoi effetti collaterali.

## se la dichiarazione

Come qualsiasi altra espressione, il valore di ritorno di una espressione `if ... else` può essere ignorato (e quindi scartato). Ciò è generalmente utile solo quando il corpo dell'espressione ha effetti collaterali, come la scrittura su un file, la modifica di variabili o la stampa sullo schermo.

Inoltre, l'`else` ramo di un `if ... else` espressione è opzionale. Ad esempio, possiamo scrivere il seguente codice per l'output sullo schermo solo se viene soddisfatta una particolare condizione:

```
second = Dates.second(now())
if iseven(second)
    println("The current second, $second, is even.")
end
```

Nell'esempio sopra, usiamo le funzioni di [data e ora](#) per ottenere il secondo corrente; per esempio, se è attualmente alle 10:55:27, la `second` variabile avrà 27. Se questo numero è pari, verrà stampata una riga sullo schermo. Altrimenti, non sarà fatto nulla.

## Operatore condizionale ternario

```
pushunique!(A, x) = x in A ? A : push!(A, x)
```

L'operatore condizionale ternario è un'espressione meno verbale `if ... else`.

La sintassi nello specifico è:

```
[condition] ? [execute if true] : [execute if false]
```

In questo esempio, aggiungiamo  $x$  alla raccolta  $A$  solo se  $x$  non è già in  $A$ . Altrimenti, lasciamo invariato  $A$ .

Referenze operatore ternario:

- [Documentazione di Julia](#)
- [Wikibooks](#)

## Operatori di cortocircuito: `&&` e `||`

### Per la ramificazione

Gli operatori condizionali di cortocircuito `&&` e `||` può essere usato come sostituto leggero per i seguenti costrutti:

- $x \ \&\& \ y$  è equivalente a  $x \ ? \ y \ : \ x$
- $x \ || \ y$  è equivalente a  $x \ ? \ x \ : \ y$

Un uso per gli operatori di cortocircuito è un modo più conciso per testare una condizione ed eseguire una determinata azione a seconda di tale condizione. Ad esempio, il codice seguente utilizza l'operatore `&&` per generare un errore se l'argomento  $x$  è negativo:

```
function mysqrt(x)
    x < 0 && throw(DomainError("x is negative"))
    x ^ 0.5
end
```

`||` l'operatore può essere utilizzato anche per il controllo degli errori, tranne per il fatto che attiva l'errore a *meno che* una condizione non mantenga, invece che se la condizione contenga:

```
function halve(x::Integer)
    iseven(x) || throw(DomainError("cannot halve an odd number"))
    x ÷ 2
end
```

Un'altra utile applicazione è fornire un valore predefinito a un oggetto, solo se non è stato precedentemente definito:

```
isdefined(:x) || (x = NEW_VALUE)
```

Qui, controlla se il simbolo  $x$  è definito (cioè se c'è un valore assegnato all'oggetto  $x$ ). Se è così, allora non succede niente. Ma, in caso contrario,  $x$  verrà assegnato `NEW_VALUE`. Nota che questo esempio funzionerà solo in ambito Toplevel.

### In condizioni

Gli operatori sono anche utili perché possono essere utilizzati per testare due condizioni, la seconda delle quali viene valutata solo in base al risultato della prima condizione. Dalla

## documentazione di Julia:

Nell'espressione `a && b`, la sottoespressione `b` viene valutato solo se `a` viene valutato come `true`

Nell'espressione `a || b`, la sottoespressione `b` viene valutata solo se `a` valuta è `false`

Quindi, mentre sia `a & b` che `a && b` daranno `true` se entrambi `a` e `b` sono `true`, il loro comportamento se `a` è `false` è diverso.

Ad esempio, supponiamo di voler verificare se un oggetto è un numero positivo, dove è possibile che non sia nemmeno un numero. Considera le differenze tra queste due implementazioni tentate:

```
CheckPositive1(x) = (typeof(x)<:Number) & (x > 0) ? true : false
CheckPositive2(x) = (typeof(x)<:Number) && (x > 0) ? true : false

CheckPositive1("a")
CheckPositive2("a")
```

`CheckPositive1()` genererà un errore se un tipo non numerico viene fornito come argomento. Questo perché valuta *entrambe le* espressioni, indipendentemente dal risultato del primo, e la seconda espressione produrrà un errore quando si tenta di valutarlo per un tipo non numerico.

`CheckPositive2()`, tuttavia, restituirà `false` (piuttosto che un errore) se viene fornito un tipo non numerico, poiché la seconda espressione viene valutata solo se la prima è `true`.

Più di un operatore di cortocircuito può essere messo insieme. Per esempio:

```
1 > 0 && 2 > 0 && 3 > 5
```

## se dichiarazione con più rami

```
d = Dates.dayofweek(now())
if d == 7
    println("It is Sunday!")
elseif d == 6
    println("It is Saturday!")
elseif d == 5
    println("Almost the weekend!")
else
    println("Not the weekend yet...")
end
```

Qualsiasi numero di `elseif` rami può essere utilizzato con un `if` dichiarazione, possibilmente con o senza finale `else` ramo. Le condizioni successive saranno valutate solo se tutte le condizioni precedenti sono risultate `false`.

## La funzione `ifelse`

```
shift(x) = ifelse(x > 10, x + 1, x - 1)
```



## Uso:

```
julia> shift(10)
9

julia> shift(11)
12

julia> shift(-1)
-2
```

La funzione `ifelse` valuterà entrambi i rami, anche quello che non è selezionato. Questo può essere utile quando i rami hanno effetti collaterali che devono essere valutati, o perché può essere più veloce se entrambi i rami stessi sono economici.

Leggi Condizionali online: <https://riptutorial.com/it/julia-lang/topic/4356/condizionali>

# Capitolo 10: confronti

## Sintassi

- $x < y$  # se  $x$  è strettamente inferiore a  $y$
- $x > y$  # se  $x$  è strettamente maggiore di  $y$
- $x == y$  # se  $x$  è uguale a  $y$
- $x === y$  # in alternativa  $x \equiv y$ , se  $x$  è uguale a  $y$
- $x \leq y$  # in alternativa  $x \leq y$ , se  $x$  è minore o uguale a  $y$
- $x \geq y$  # in alternativa  $x \geq y$ , se  $x$  è maggiore o uguale a  $y$
- $x \neq y$  # in alternativa  $x \neq y$ , se  $x$  non è uguale a  $y$
- $x \approx y$  # se  $x$  è approssimativamente uguale a  $y$

## Osservazioni

Stai attento a lanciare i segni di confronto in giro. Julia definisce molte funzioni di confronto per impostazione predefinita senza definire la corrispondente versione capovolta. Ad esempio, si può correre

```
julia> Set{Int}(1:3) ⊆ Set{Int}(0:5)
true
```

ma non funziona

```
julia> Set{Int}(0:5) ⊇ Set{Int}(1:3)
ERROR: UndefVarError: ⊇ not defined
```

## Examples

### Confronti concatenati

Gli operatori di confronto multipli utilizzati insieme sono concatenati, come se fossero collegati tramite l' **operatore** `&&`. Questo può essere utile per catene di confronto leggibili e matematicamente concise, come

```
# same as 0 < i && i <= length(A)
isinbounds(A, i) = 0 < i ≤ length(A)

# same as Set{Int}() != x && issubset(x, y)
isnonemptysubset(x, y) = Set{Int}() ≠ x ⊆ y
```

Tuttavia, c'è una differenza importante tra  $a > b > c$  e  $a > b \ \&\& \ b > c$ ; nel secondo caso, il termine  $b$  viene valutato due volte. Questo non importa molto per i semplici simboli vecchi, ma potrebbe importare se i termini stessi hanno effetti collaterali. Per esempio,

```
julia> f(x) = (println(x); 2)
f (generic function with 1 method)

julia> 3 > f("test") > 1
test
true

julia> 3 > f("test") && f("test") > 1
test
test
true
```

Diamo un'occhiata più approfondita ai confronti concatenati e al loro funzionamento, osservando come vengono analizzati e ridotti in [espressioni](#). Innanzitutto, considera il semplice confronto, che possiamo vedere solo una semplice chiamata di funzione vecchia:

```
julia> dump(:(a > b))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol >
    2: Symbol a
    3: Symbol b
  typ: Any
```

Ora se eseguiamo il paragone, notiamo che l'analisi è cambiata:

```
julia> dump(:(a > b >= c))
Expr
  head: Symbol comparison
  args: Array{Any}((5,))
    1: Symbol a
    2: Symbol >
    3: Symbol b
    4: Symbol >=
    5: Symbol c
  typ: Any
```

Dopo l'analisi, l'espressione viene quindi abbassata nella sua forma finale:

```
julia> expand(:(a > b >= c))
:(begin
    unless a > b goto 3
    return b >= c
  3:
    return false
end)
```

e notiamo infatti che questo è lo stesso di `a > b && b >= c`:

```
julia> expand(:(a > b && b >= c))
:(begin
    unless a > b goto 3
    return b >= c
  3:
```

```
        return false
    end)
```

## Numeri ordinali

Vedremo come implementare confronti personalizzati implementando un tipo personalizzato, **numeri ordinali**. Per semplificare l'implementazione, ci concentreremo su un piccolo sottoinsieme di questi numeri: tutti i numeri ordinali fino a  $\epsilon_0$  ma non incluso  $\epsilon_0$ . La nostra implementazione è focalizzata sulla semplicità, non sulla velocità; tuttavia, anche l'implementazione non è lenta.

Memorizziamo numeri ordinali secondo la loro **forma normale di Cantor**. Poiché l'aritmetica ordinale non è commutativa, adotteremo la convenzione comune di memorizzare prima i termini più significativi.

```
immutable OrdinalNumber <: Number
  βs::Vector{OrdinalNumber}
  cs::Vector{Int}
end
```

Poiché la forma normale di Cantor è unica, possiamo testare l'uguaglianza semplicemente attraverso l'uguaglianza ricorsiva:

### 0.5.0

Nella versione v0.5, c'è una sintassi molto bella per farlo in modo compatto:

```
import Base: ==
α::OrdinalNumber == β::OrdinalNumber = α.βs == β.βs && α.cs == β.cs
```

### 0.5.0

Altrimenti, definire la funzione come più tipica:

```
import Base: ==
==(α::OrdinalNumber, β::OrdinalNumber) = α.βs == β.βs && α.cs == β.cs
```

Per completare il nostro ordine, poiché questo tipo ha un ordine totale, dovremmo sovraccaricare la funzione `isless`:

```
import Base: isless
function isless(α::OrdinalNumber, β::OrdinalNumber)
    for i in 1:min(length(α.cs), length(β.cs))
        if α.βs[i] < β.βs[i]
            return true
        elseif α.βs[i] == β.βs[i] && α.cs[i] < β.cs[i]
            return true
        end
    end
    return length(α.cs) < length(β.cs)
end
```

Per testare il nostro ordine, possiamo creare alcuni metodi per creare numeri ordinali. Zero, ovviamente, si ottiene non avendo termini nella forma normale di Cantor:

```
const ORDINAL_ZERO = OrdinalNumber([], [])
Base.zero(::Type{OrdinalNumber}) = ORDINAL_ZERO
```

Possiamo definire un  $\exp_w$  per calcolare  $w^\alpha$ , e usarlo per calcolare 1 e  $w$ :

```
expw(a) = OrdinalNumber([a], [1])
const ORDINAL_ONE = expw(ORDINAL_ZERO)
Base.one(::Type{OrdinalNumber}) = ORDINAL_ONE
const ω = expw(ORDINAL_ONE)
```

Ora abbiamo una funzione di ordinamento completamente funzionale sui numeri ordinali:

```
julia> ORDINAL_ZERO < ORDINAL_ONE <  $\omega$  <  $\exp(\omega)$ 
true

julia> ORDINAL_ONE > ORDINAL_ZERO
true

julia> sort([ORDINAL_ONE,  $\omega$ ,  $\exp(\omega)$ , ORDINAL_ZERO])

4-element Array{OrdinalNumber,1}:
OrdinalNumber{OrdinalNumber[],Int64{}}

OrdinalNumber{OrdinalNumber[OrdinalNumber{OrdinalNumber[],Int64{}}], [1]}
```

Nell'ultimo esempio, vediamo che la stampa dei numeri ordinali potrebbe essere migliore, ma il risultato è come previsto.

## Operatori standard

Julia supporta un insieme molto grande di operatori di confronto. Questi includono

- [illegible]

Non tutti hanno una definizione nella libreria `Base` standard. Tuttavia, sono disponibili per altri pacchetti da definire e utilizzare come appropriato.

Nell'uso quotidiano, la maggior parte di questi operatori di confronto non sono rilevanti. I più comuni utilizzati sono le funzioni matematiche standard per l'ordinazione; vedere la sezione Sintassi per un elenco.

Come la maggior parte degli altri operatori di Julia, gli operatori di confronto sono [funzioni](#) e possono essere chiamati come funzioni. Ad esempio, `(<)(1, 2)` è identico nel significato a `1 < 2`.

## Usando `==`, `===` e `isequal`

Esistono tre operatori di uguaglianza: `==`, `===` e `isequal`. (L'ultimo non è realmente un operatore, ma è una funzione e tutti gli operatori sono funzioni.)

## Quando usare `==`

`==` è l'uguaglianza di *valore*. Restituisce `true` quando due oggetti rappresentano, nel loro stato attuale, lo stesso valore.

Ad esempio, è ovvio che

```
julia> 1 == 1
true
```

ma inoltre

```
julia> 1 == 1.0
true

julia> 1 == 1.0 + 0.0im
true

julia> 1 == 1//1
true
```

I lati di destra di ogni uguaglianza sopra sono di un [tipo](#) diverso, ma rappresentano sempre lo stesso valore.

Per oggetti mutabili, come gli [array](#), `==` confronta il loro valore attuale.

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> C = [1, 3, 2]
3-element Array{Int64,1}:
 1
 3
 2
```

```

1
3
2

julia> A == B
true

julia> A == C
false

julia> A[2], A[3] = A[3], A[2] # swap 2nd and 3rd elements of A
(3,2)

julia> A
3-element Array{Int64,1}:
 1
 3
 2

julia> A == B
false

julia> A == C
true

```

Il più delle volte, `==` è la scelta giusta.

## Quando usare `===`

`===` è un'operazione molto più rigida di `==`. Invece dell'uguaglianza di valore, misura l'eguaglianza. Due oggetti sono eguali se non possono essere distinti l'uno dall'altro dal programma stesso. Così abbiamo

```

julia> 1 === 1
true

```

in quanto non c'è modo di distinguere `1` dall'altra `1`. Ma

```

julia> 1 === 1.0
false

```

perché sebbene `1` e `1.0` abbiano lo stesso valore, sono di tipi diversi e quindi il programma può distinguerli.

Inoltre,

```

julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [1, 2, 3]
3-element Array{Int64,1}:

```

```
1
2
3

julia> A === B
false

julia> A === A
true
```

che a prima vista può sembrare sorprendente! Come potrebbe il programma distinguere tra i due vettori `A` e `B` ? Poiché i vettori sono mutabili, potrebbe modificare `A` e quindi si comporterebbe diversamente da `B` Ma non importa come modifica `A` , `A` si comporterà sempre come `A` stesso. Quindi `A` è egale ad `A` , ma non egale a `B`

Continuando su questa linea, osserva

```
julia> C = A
3-element Array{Int64,1}:
 1
 2
 3

julia> A === C
true
```

Assegnando `A` a `C` , diciamo che `C` ha *alias* `A` Cioè, è diventato solo un altro nome per `A` Anche le modifiche apportate ad `A` saranno osservate da `C` Pertanto, non c'è modo di dire la differenza tra `A` e `C` , quindi sono uguali.

## Quando usare `isequal`

La differenza tra `==` e `isequal` è molto sottile. La più grande differenza riguarda la modalità di gestione dei numeri in virgola mobile:

```
julia> NaN == NaN
false
```

Questo risultato, forse sorprendente, è [definito](#) dallo standard IEEE per i tipi a virgola mobile (IEEE-754). Ma questo non è utile in alcuni casi, come l'ordinamento. `isequal` è fornito per quei casi:

```
julia> isequal(NaN, NaN)
true
```

Dall'altra parte dello spettro, `==` tratta lo zero negativo IEEE e lo zero positivo come lo stesso valore (anche come specificato da IEEE-754). Questi valori hanno rappresentazioni distinte in memoria, tuttavia.

```
julia> 0.0
```



```
0.0

julia> -0.0
-0.0

julia> 0.0 == -0.0
true
```

Ancora una volta per scopi di smistamento, `isequal` distingue tra loro.

```
julia> isequal(0.0, -0.0)
false
```

Leggi confronti online: <https://riptutorial.com/it/julia-lang/topic/5563/confronti>

---

# Capitolo 11: dizionari

## Examples

### Usare i dizionari

I dizionari possono essere costruiti passando un numero qualsiasi di coppie.

```
julia> Dict{"A"=>1, "B"=>2}
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

È possibile ottenere voci in un dizionario mettendo la chiave tra parentesi quadre.

```
julia> dict = Dict{"A"=>1, "B"=>2}
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1

julia> dict["A"]
1
```

Leggi dizionari online: <https://riptutorial.com/it/julia-lang/topic/9028/dizionari>

# Capitolo 12: Elaborazione parallela

## Examples

### pmap

`pmap` prende una funzione (che tu specifichi) e la applica a tutti gli elementi di una matrice. Questo lavoro è suddiviso tra i lavoratori disponibili. `pmap` restituisce quindi i risultati da tale funzione in un altro array.

```
addprocs(3)
sqrts = pmap(sqrt, 1:10)
```

se la tua funzione richiede più argomenti, puoi fornire più vettori a `pmap`

```
dots = pmap(dot, 1:10, 11:20)
```

Come con `@parallel`, tuttavia, se la funzione assegnata a `pmap` non è in Julia di base (ovvero è definita dall'utente o definita in un pacchetto), è necessario assicurarsi che tale funzione sia disponibile per tutti i lavoratori per primi:

```
@everywhere begin
    function rand_det(n)
        det(rand(n,n))
    end
end

determinants = pmap(rand_det, 1:10)
```

Vedi anche [questo](#) SO Q & A.

### @parallelo

`@parallel` può essere usato per parallelizzare un loop, dividendo i passaggi del loop su diversi worker. Come un esempio molto semplice:

```
addprocs(3)

a = collect(1:10)

for idx = 1:10
    println(a[idx])
end
```

Per un esempio leggermente più complesso, considera:

```
@time begin
    @sync begin
```

```

        @parallel for idx in 1:length(a)
            sleep(a[idx])
        end
    end
end
27.023411 seconds (13.48 k allocations: 762.532 KB)
julia> sum(a)
55

```

Quindi, vediamo che se avessimo eseguito questo ciclo senza `@parallel` ci sarebbero voluti 55 secondi, anziché 27, da eseguire.

Possiamo anche fornire un operatore di riduzione per la macro `@parallel`. Supponiamo di avere una matrice, vogliamo sommare ogni colonna della matrice e quindi moltiplicare queste somme l'una dall'altra:

```

A = rand(100,100);

@parallel (*) for idx = 1:size(A,1)
    sum(A[:,idx])
end

```

Ci sono diverse cose importanti da tenere a mente quando si utilizza `@parallel` per evitare comportamenti imprevisti.

**Primo:** se si desidera utilizzare qualsiasi funzione nei propri loop che non sono in Julia di base (ad esempio, le funzioni definite nello script o importate dai pacchetti), è necessario rendere tali funzioni accessibili ai lavoratori. Pertanto, ad esempio, quanto segue *non* funzionerebbe:

```

myprint(x) = println(x)
for idx = 1:10
    myprint(a[idx])
end

```

Invece, dovremmo usare:

```

@everywhere begin
    function myprint(x)
        println(x)
    end
end

@parallel for idx in 1:length(a)
    myprint(a[idx])
end

```

**Secondo** Sebbene ogni lavoratore potrà accedere agli oggetti nel campo di applicazione del controllore, *non* saranno in grado di modificarli. così

```

a = collect(1:10)
@parallel for idx = 1:length(a)
    a[idx] += 1
end

```

```
julia> a'
1x10 Array{Int64,2}:
 1  2  3  4  5  6  7  8  9 10
```

Considerando che, se avessimo eseguito il ciclo con il `@parallel`, avremmo modificato correttamente l'array `a`.

PER INDIRIZZARLO, possiamo invece creare `a` oggetto di tipo `SharedArray` modo che ogni lavoratore possa accedervi e modificarlo:

```
a = convert(SharedArray{Float64,1}, collect(1:10))
@parallel for idx = 1:length(a)
    a[idx] += 1
end

julia> a'
1x10 Array{Float64,2}:
 2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0
```

## @spawn e @spawnat

Le macro `@spawn` e `@spawnat` sono due degli strumenti che Julia mette a disposizione per assegnare compiti ai lavoratori. Ecco un esempio:

```
julia> @spawnat 2 println("hello world")
RemoteRef{Channel{Any}}(2,1,3)

julia> From worker 2: hello world
```

Entrambe queste macro valuteranno [un'espressione](#) su un processo di lavoro. L'unica differenza tra i due è che `@spawnat` ti permette di scegliere quale lavoratore valuterà l'espressione (nell'esempio sopra è specificato worker 2) mentre con `@spawn` verrà automaticamente scelto un worker, in base alla disponibilità.

Nell'esempio sopra, abbiamo semplicemente avuto worker 2 per eseguire la funzione `println`. Non c'era nulla di interessante da restituire o recuperare da questo. Spesso, tuttavia, l'espressione che abbiamo inviato al lavoratore produrrà qualcosa che desideriamo recuperare. Notare nell'esempio sopra, quando abbiamo chiamato `@spawnat`, prima di ottenere la stampa da worker 2, abbiamo visto quanto segue:

```
RemoteRef{Channel{Any}}(2,1,3)
```

Questo indica che la macro `@spawnat` restituirà un oggetto di tipo `RemoteRef`. Questo oggetto a sua volta conterrà i valori di ritorno dalla nostra espressione che viene inviata al lavoratore. Se vogliamo recuperare quei valori, possiamo prima assegnare il `RemoteRef` che `@spawnat` ritorna ad un oggetto e poi, e poi usa la funzione `fetch()` che opera su un oggetto di tipo `RemoteRef`, per recuperare i risultati memorizzati da una valutazione eseguita su un lavoratore.

```
julia> result = @spawnat 2 2 + 5
```

```
RemoteRef{Channel{Any}} (2,1,26)

julia> fetch(result)
7
```

La chiave per essere in grado di usare efficacemente `@spawn` è capire la natura dietro le [espressioni](#) su cui opera. Usare `@spawn` per inviare comandi ai lavoratori è un po' più complicato della semplice digitazione diretta di ciò che si dovrebbe scrivere se si stesse eseguendo un "interprete" su uno dei lavoratori o eseguendo il codice in modo nativo su di essi. Ad esempio, supponiamo di voler utilizzare `@spawnat` per assegnare un valore a una variabile su un worker. Potremmo provare:

```
@spawnat 2 a = 5
RemoteRef{Channel{Any}} (2,1,2)
```

Ha funzionato? Bene, vediamo con il lavoratore 2 provare a stampare `a`.

```
julia> @spawnat 2 println(a)
RemoteRef{Channel{Any}} (2,1,4)

julia>
```

Non è successo niente. Perché? Possiamo investigare di più usando `fetch()` come sopra. `fetch()` può essere molto utile perché recupera non solo i risultati di successo ma anche i messaggi di errore. Senza di esso, potremmo anche non sapere che qualcosa è andato storto.

```
julia> result = @spawnat 2 println(a)
RemoteRef{Channel{Any}} (2,1,5)

julia> fetch(result)
ERROR: On worker 2:
UndefVarError: a not defined
```

Il messaggio di errore dice che `a` non è definito su worker 2. Ma perché è questo? Il motivo è che abbiamo bisogno di avvolgere la nostra operazione di assegnazione in un'espressione che usiamo quindi `@spawn` per dire al lavoratore di valutare. Di seguito è riportato un esempio, con spiegazione seguente:

```
julia> @spawnat 2 eval(:(a = 2))
RemoteRef{Channel{Any}} (2,1,7)

julia> @spawnat 2 println(a)
RemoteRef{Channel{Any}} (2,1,8)

julia> From worker 2: 2
```

La sintassi `:( )` è ciò che Julia usa per designare le [espressioni](#). Quindi usiamo la `eval()` in Julia, che valuta un'espressione, e usiamo la macro `@spawnat` per `@spawnat` che l'espressione deve essere valutata su worker 2.

Potremmo anche ottenere lo stesso risultato di:

```
julia> @spawnat(2, eval(parse("c = 5")))
RemoteRef{Channel{Any}}(2,1,9)

julia> @spawnat 2 println(c)
RemoteRef{Channel{Any}}(2,1,10)

julia> From worker 2: 5
```

Questo esempio dimostra due nozioni aggiuntive. Innanzitutto, vediamo che possiamo anche creare un'espressione usando la funzione `parse()` chiamata su una stringa. In secondo luogo, vediamo che possiamo utilizzare le parentesi quando si chiama `@spawnat`, in situazioni in cui ciò potrebbe rendere la nostra sintassi più chiara e gestibile.

## Quando usare `@parallel` vs `pmap`

La [documentazione di Julia](#) lo consiglia

`pmap()` è progettato per il caso in cui ogni chiamata di funzione svolge una grande quantità di lavoro. Al contrario, `@parallel` può gestire situazioni in cui ogni iterazione è minima, forse semplicemente sommando due numeri.

Ci sono diverse ragioni per questo. Innanzitutto, `pmap` incorre in costi di avvio maggiori per l'avvio di lavori sui lavoratori. Pertanto, se i lavori sono molto piccoli, questi costi di avvio potrebbero diventare inefficienti. Viceversa, tuttavia, `pmap` svolge un lavoro "più intelligente" nell'assegnare posti di lavoro tra i lavoratori. In particolare, crea una coda di lavori e invia un nuovo lavoro a ciascun lavoratore ogni volta che quel lavoratore diventa disponibile. `@parallel` al contrario, divide tutto il lavoro da fare tra i lavoratori quando viene chiamato. Pertanto, se alcuni lavoratori impiegano più tempo a svolgere il proprio lavoro rispetto ad altri, si può finire con una situazione in cui la maggior parte dei lavoratori ha finito e sono inattivi mentre alcuni rimangono attivi per un numero eccessivo di tempo, finendo il proprio lavoro. Tale situazione, tuttavia, è meno probabile che si verifichi con lavori molto piccoli e semplici.

Ciò che segue illustra questo: supponiamo di avere due lavoratori, uno dei quali è lento e l'altro è il doppio più veloce. Idealmente, vorremmo dare al lavoratore veloce il doppio del lavoro del lavoratore lento. (oppure, potremmo avere lavori veloci e lenti, ma il principale è esattamente lo stesso). `pmap` lo realizzerà, ma `@parallel` non lo farà.

Per ogni test, inizializziamo quanto segue:

```
addprocs(2)

@everywhere begin
    function parallel_func(idx)
        workernum = myid() - 1
        sleep(workernum)
        println("job $idx")
    end
end
```

Ora, per il test `@parallel`, eseguiamo quanto segue:

```
@parallel for idx = 1:12
    parallel_func(idx)
end
```

E torna all'output di stampa:

```
julia>      From worker 2:    job 1
      From worker 3:    job 7
      From worker 2:    job 2
      From worker 2:    job 3
      From worker 3:    job 8
      From worker 2:    job 4
      From worker 2:    job 5
      From worker 3:    job 9
      From worker 2:    job 6
      From worker 3:    job 10
      From worker 3:    job 11
      From worker 3:    job 12
```

È quasi dolce. I lavoratori hanno "condiviso" il lavoro in modo uniforme. Nota che ogni lavoratore ha completato 6 lavori, anche se il lavoratore 2 è due volte più veloce del lavoratore 3. Può essere toccante, ma non è efficiente.

Per il test `pmap`, `pmap` le seguenti operazioni:

```
pmap(parallel_func, 1:12)
```

e ottieni l'output:

```
From worker 2:    job 1
From worker 3:    job 2
From worker 2:    job 3
From worker 2:    job 5
From worker 3:    job 4
From worker 2:    job 6
From worker 2:    job 8
From worker 3:    job 7
From worker 2:    job 9
From worker 2:    job 11
From worker 3:    job 10
From worker 2:    job 12
```

Ora, si noti che worker 2 ha eseguito 8 lavori e il worker 3 ha eseguito 4. Questo è esattamente in proporzione alla loro velocità e cosa vogliamo per l'efficienza ottimale. `pmap` è un master per compiti difficili - da ciascuno secondo le proprie capacità.

## @async e @sync

Secondo la documentazione sotto `?@async`, "`@async` wrapping di un'espressione in un'attività." Ciò significa che per qualsiasi cosa rientri nel suo ambito, Julia avvierà questa attività in esecuzione, ma poi procederà a ciò che viene dopo nello script senza attendere il completamento dell'attività. Quindi, ad esempio, senza la macro otterrai:



```
julia> @time sleep(2)
2.005766 seconds (13 allocations: 624 bytes)
```

Ma con la macro, ottieni:

```
julia> @time @async sleep(2)
0.000021 seconds (7 allocations: 657 bytes)
Task (waiting) @0x0000000112a65ba0

julia>
```

In tal modo, Julia consente allo script di procedere (e alla macro `@time` di eseguire completamente) senza attendere che l'attività (in questo caso, dormire per due secondi) `@time` completata.

La macro `@sync`, al contrario, "Attende fino a quando tutti gli usi dinamicamente chiusi di `@async`, `@spawn`, `@spawnat` e `@parallel` sono completi." (secondo la documentazione sotto `?@sync`). Quindi, vediamo:

```
julia> @time @sync @async sleep(2)
2.002899 seconds (47 allocations: 2.986 KB)
Task (done) @0x0000000112bd2e00
```

In questo semplice esempio, non è necessario includere una singola istanza di `@async` e `@sync` insieme. Ma, dove `@sync` può essere utile, è il caso in cui `@async` applicato a più operazioni che si desidera consentire a tutti di iniziare subito senza attendere il completamento di ciascuna operazione.

Ad esempio, supponiamo di avere più lavoratori e vorremmo iniziare ognuno di loro a lavorare su un'attività contemporaneamente e quindi recuperare i risultati da tali attività. Un tentativo iniziale (ma errato) potrebbe essere:

```
addprocs(2)
@time begin
    a = cell(nworkers())
    for (idx, pid) in enumerate(workers())
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 4.011576 seconds (177 allocations: 9.734 KB)
```

Il problema qui è che il ciclo attende ogni operazione di `remotecall_fetch()` per finire, cioè che ogni processo completi il suo lavoro (in questo caso dormendo per 2 secondi) prima di continuare ad avviare la successiva operazione `remotecall_fetch()`. In termini di situazione pratica, qui non riceviamo i vantaggi del parallelismo, poiché i nostri processi non stanno facendo il loro lavoro (cioè dormendo) simultaneamente.

Possiamo correggere questo, tuttavia, utilizzando una combinazione dei macro `@async` e `@sync`:

```
@time begin
    a = cell(nworkers())
    @sync for (idx, pid) in enumerate(workers())
```

```

        @async a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 2.009416 seconds (274 allocations: 25.592 KB)

```

Ora, se contiamo ogni passo del ciclo come operazione separata, vediamo che ci sono due operazioni separate precedute dalla macro `@async`. La macro consente a ciascuno di questi di avviarsi e il codice per continuare (in questo caso al prossimo passo del ciclo) prima di ogni finitura. Tuttavia, l'uso della macro `@sync`, il cui ambito comprende l'intero ciclo, significa che non consentiremo che lo script proceda oltre quel ciclo finché tutte le operazioni precedute da `@async` siano state completate.

È possibile ottenere una comprensione ancora più chiara del funzionamento di queste macro modificando ulteriormente l'esempio precedente per vedere come cambia in determinate modifiche. Ad esempio, supponiamo di avere solo `@async` senza `@sync`:

```

@time begin
    a = cell(nworkers())
    for (idx, pid) in enumerate(workers())
        println("sending work to $pid")
        @async a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 0.001429 seconds (27 allocations: 2.234 KB)

```

Qui, la macro `@async` ci consente di continuare nel nostro ciclo ancor prima che ciascuna operazione di `remotecall_fetch()` esecuzione. Ma, nel bene o nel male, non abbiamo una macro `@sync` per impedire che il codice continui oltre questo ciclo fino a quando tutte le operazioni di `remotecall_fetch()` non terminano.

Ciononostante, ogni operazione di `remotecall_fetch()` è ancora in esecuzione in parallelo, anche quando andiamo avanti. Possiamo vedere che, se aspettiamo due secondi, l'array `a`, contenente i risultati, conterrà:

```

sleep(2)
julia> a
2-element Array{Any,1}:
 nothing
 nothing

```

(L'elemento "niente" è il risultato di un recupero riuscito dei risultati della funzione `sleep`, che non restituisce alcun valore)

Possiamo anche vedere che le due operazioni di `remotecall_fetch()` iniziano essenzialmente nello stesso momento perché i comandi di `print` che li precedono vengono eseguiti in rapida successione (l'output di questi comandi non è mostrato qui). Confrontalo con il prossimo esempio in cui i comandi di `print` vengono eseguiti a intervalli di 2 secondi l'uno dall'altro:

Se mettiamo la macro `@async` sull'intero loop (invece che sul suo passo interno), il nostro script continuerà immediatamente senza attendere che le operazioni di `remotecall_fetch()` finiscano. Ora, tuttavia, consentiamo allo script di continuare oltre il ciclo nel suo complesso. Non

permettiamo che ogni singola fase del ciclo inizi prima della precedente. Pertanto, a differenza dell'esempio sopra, due secondi dopo che lo script procede dopo il ciclo, l'array dei `results` ha ancora un elemento come `#undef` che indica che la seconda operazione `remotecall_fetch()` non è ancora stata completata.

```
@time begin
  a = cell(nworkers())
  @async for (idx, pid) in enumerate(workers())
    println("sending work to $pid")
    a[idx] = remotecall_fetch(pid, sleep, 2)
  end
end
# 0.001279 seconds (328 allocations: 21.354 KB)
# Task (waiting) @0x0000000115ec9120
## This also allows us to continue to

sleep(2)

a
2-element Array{Any,1}:
  nothing
  #undef
```

E, non sorprendentemente, se mettiamo `@sync` e `@async` accanto all'altro, otteniamo che ogni `remotecall_fetch()` eseguito in modo sequenziale (anziché simultaneamente) ma non continuiamo nel codice finché ognuno non ha finito. In altre parole, questo sarebbe essenzialmente l'equivalente se non avessimo nessuna macro in posto, proprio come `sleep(2)` si comporta essenzialmente in modo identico a `@sync @async sleep(2)`

```
@time begin
  a = cell(nworkers())
  @sync @async for (idx, pid) in enumerate(workers())
    a[idx] = remotecall_fetch(pid, sleep, 2)
  end
end
# 4.019500 seconds (4.20 k allocations: 216.964 KB)
# Task (done) @0x0000000115e52a10
```

Si noti inoltre che è possibile avere operazioni più complicate nell'ambito della macro `@async`. La [documentazione](#) fornisce un esempio contenente un intero ciclo nell'ambito di `@async`.

Ricordiamo che l'aiuto per i macro di sincronizzazione afferma che "Attenderà fino a quando tutti gli usi dinamicamente chiusi di `@async`, `@spawn`, `@spawnat` e `@parallel` saranno completi." Ai fini di ciò che conta come "completo" importa come si definiscono le attività nell'ambito delle macro `@sync` e `@async`. Considera l'esempio seguente, che è una leggera variazione su uno degli esempi sopra riportati:

```
@time begin
  a = cell(nworkers())
  @sync for (idx, pid) in enumerate(workers())
    @async a[idx] = remotecall(pid, sleep, 2)
  end
end
## 0.172479 seconds (93.42 k allocations: 3.900 MB)
```

```
julia> a
2-element Array{Any,1}:
 RemoteRef{Channel{Any}} (2,1,3)
 RemoteRef{Channel{Any}} (3,1,4)
```

L'esempio precedente impiegava circa 2 secondi per indicare che le due attività erano eseguite in parallelo e che lo script attendeva che ciascuna completasse l'esecuzione delle sue funzioni prima di procedere. Questo esempio, tuttavia, ha una valutazione del tempo molto più bassa. Il motivo è che ai fini di `@sync` l'operazione `remotecall()` ha "finito" una volta che ha inviato al lavoratore il compito da svolgere. (Si noti che l'array risultante, `a`, qui, contiene solo tipi di oggetto `RemoteRef`, che indicano solo che c'è qualcosa in corso con un particolare processo che in teoria potrebbe essere recuperato in qualche punto in futuro). Al contrario, l'operazione `remotecall_fetch()` ha solo "finito" quando riceve il messaggio dal lavoratore che la sua attività è completa.

Quindi, se stai cercando dei modi per assicurarti che certe operazioni con i lavoratori siano completate prima di proseguire nel tuo script (come ad esempio è discusso in [questo post](#)) è necessario riflettere attentamente su ciò che conta come "completo" e su come misurare e quindi renderlo operativo nel tuo script.

## Aggiunta di lavoratori

Quando avvii per la prima volta Julia, per impostazione predefinita, sarà disponibile un solo processo in esecuzione e disponibile per il lavoro. Puoi verificarlo usando:

```
julia> nprocs()
1
```

Per trarre vantaggio dall'elaborazione parallela, è necessario innanzitutto aggiungere altri lavoratori che saranno quindi disponibili per svolgere il lavoro che gli viene assegnato. Puoi farlo all'interno del tuo script (o dell'interprete) usando: `addprocs(n)` dove `n` è il numero di processi che vuoi usare.

In alternativa, puoi aggiungere processi quando avvii Julia dalla riga di comando usando:

```
$ julia -p n
```

dove `n` è il numero di processi *aggiuntivi* che desideri aggiungere. Quindi, se iniziamo con Julia

```
$ julia -p 2
```

Quando inizieremo Julia otterremo:

```
julia> nprocs()
3
```

**Leggi Elaborazione parallela online:** <https://riptutorial.com/it/julia-lang/topic/4542/elaborazione-parallela>

# Capitolo 13: Enums

## Sintassi

- `@enum EnumType val = 1 val val`
- `:simbolo`

## Osservazioni

A volte è utile avere tipi enumerati in cui ogni istanza è di un tipo diverso (spesso un [tipo immutabile singleton](#)); questo può essere importante per la stabilità del tipo. I tratti sono tipicamente implementati con questo paradigma. Tuttavia, ciò comporta un ulteriore sovraccarico in fase di compilazione.

## Examples

### Definizione di un tipo enumerato

Un [tipo enumerato](#) è un [tipo](#) che può contenere uno di un elenco finito di valori possibili. In Julia, i tipi enumerati sono in genere chiamati "tipi enum". Ad esempio, uno potrebbe usare i tipi di enum per descrivere i sette giorni della settimana, i dodici mesi dell'anno, i quattro semi di un [mazzo di 52 carte standard](#) o altre situazioni simili.

Possiamo definire tipi enumerati per modellare i semi e le truppe di un mazzo di 52 carte standard. La macro `@enum` viene utilizzata per definire i tipi di enum.

```
@enum Suit ♣♦♥♠  
@enum Rank ace=1 two three four five six seven eight nine ten jack queen king
```

Questo definisce due tipi: `Suit` e `Rank`. Possiamo verificare che i valori siano effettivamente dei tipi previsti:

```
julia> ♦  
♦::Suit = 1  
  
julia> six  
six::Rank = 6
```

Si noti che ogni seme e classifica sono stati associati a un numero. Per impostazione predefinita, questo numero inizia da zero. Quindi il secondo seme, quadri, è stato assegnato al numero 1. Nel caso di `Rank`, potrebbe essere più sensato iniziare il numero a uno. Ciò è stato ottenuto annotando la definizione di `ace` con annotazione `a = 1`.

I tipi enumerati sono dotati di molte funzionalità, come l'uguaglianza (e in effetti l'identità) e i confronti integrati:

```
julia> seven === seven
true

julia> ten ≠ jack
true

julia> two < three
true
```

Come i valori di qualsiasi altro [tipo immutabile](#) , anche i valori dei tipi enumerati possono essere sottoposti a hash e memorizzati in `Dict` s.

Possiamo completare questo esempio definendo un tipo di `Card` che ha un campo `Rank` e un campo `Suit` :

```
immutable Card
    rank::Rank
    suit::Suit
end
```

e quindi possiamo creare carte con

```
julia> Card(three, ♣)
Card(three::Rank = 3,♣::Suit = 0)
```

Ma i tipi elencati includono anche i propri metodi di `convert` , quindi possiamo semplicemente farlo

```
julia> Card(7, ♠)
Card(seven::Rank = 7,♠::Suit = 3)
```

e poiché 7 può essere convertito direttamente in `Rank` , questo costruttore funziona fuori dalla scatola.

Potremmo voler definire lo zucchero sintattico per costruire queste carte; la moltiplicazione implicita fornisce un modo conveniente per farlo. Definire

```
julia> import Base.*

julia> r::Int * s::Suit = Card(r, s)
* (generic function with 156 methods)
```

e poi

```
julia> 10♣
Card(ten::Rank = 10,♣::Suit = 0)

julia> 5♠
Card(five::Rank = 5,♠::Suit = 3)
```

ancora una volta sfruttando le funzioni di `convert` integrate.

## Usare simboli come leggere enumerazioni

Sebbene la macro `@enum` sia abbastanza utile per la maggior parte dei casi d'uso, può essere eccessiva in alcuni casi d'uso. Gli svantaggi di `@enum` includono:

- Crea un nuovo tipo
- È un po' più difficile da estendere
- Viene fornito con funzionalità come la conversione, l'enumerazione e il confronto, che possono essere superflui in alcune applicazioni

Nei casi in cui si desideri un'alternativa più leggera, è possibile utilizzare il tipo di `Symbol`. I simboli sono [stringhe internate](#); rappresentano sequenze di personaggi, proprio come fanno gli [archi](#), ma sono associati in modo univoco con i numeri. Questa associazione unica consente il confronto veloce dell'uguaglianza dei simboli.

Potremmo implementare nuovamente un tipo di `Card`, questa volta utilizzando i campi `Symbol`:

```
const ranks = Set([:ace, :two, :three, :four, :five, :six, :seven, :eight, :nine,
                  :ten, :jack, :queen, :king])
const suits = Set([:♣, :♦, :♥, :♠])
immutable Card
  rank::Symbol
  suit::Symbol
  function Card(r::Symbol, s::Symbol)
    r in ranks || throw(ArgumentError("invalid rank: $r"))
    s in suits || throw(ArgumentError("invalid suit: $s"))
    new(r, s)
  end
end
```

Implementiamo il costruttore interno per verificare eventuali valori errati passati al costruttore. Diversamente dall'esempio che usa i tipi `@enum`, `Symbol` `s` può contenere qualsiasi stringa, quindi dobbiamo stare attenti a quali tipi di `Symbol` accettiamo. Nota qui l'uso degli operatori condizionali di [cortocircuito](#).

Ora possiamo costruire oggetti `Card` come ci aspettiamo:

```
julia> Card(:ace, :♦)
Card(:ace, :♦)

julia> Card(:nine, :♠)
Card(:nine, :♠)

julia> Card(:eleven, :♠)
ERROR: ArgumentError: invalid rank: eleven
in Card(::Symbol, ::Symbol) at ./REPL[17]:5

julia> Card(:king, :X)
ERROR: ArgumentError: invalid suit: X
in Card(::Symbol, ::Symbol) at ./REPL[17]:6
```

Un importante vantaggio di `Symbol` `s` è la loro estensibilità al runtime. Se in fase di esecuzione, desideriamo accettare (per esempio) `:eleven` come nuovo rango, è sufficiente eseguire

semplicemente `push!(ranks, :eleven)` `Rank,: push!(ranks, :eleven)` . Tale estensibilità del runtime non è possibile con i tipi `@enum` .

Leggi Enums online: <https://riptutorial.com/it/julia-lang/topic/7104/enums>



# Capitolo 14: espressioni

## Examples

### Introduzione alle espressioni

Le espressioni sono un tipo specifico di oggetto in Julia. Puoi pensare a un'espressione come a un pezzo di codice Julia che non è stato ancora valutato (cioè eseguito). Ci sono poi funzioni e operazioni specifiche, come `eval()` che valuterà l'espressione.

Ad esempio, potremmo scrivere uno script o inserire nell'interprete quanto segue: `julia> 1 + 1`

Un modo per creare un'espressione è usare la sintassi `:()`. Per esempio:

```
julia> MyExpression = :(1+1)
:(1 + 1)
julia> typeof(MyExpression)
Expr
```

Ora abbiamo un oggetto di tipo `Expr`. Essendosi appena formato, non fa nulla, si siede come un qualsiasi altro oggetto fino a quando non viene messo in atto. In questo caso, possiamo *valutare* quell'espressione usando la `eval()`:

```
julia> eval(MyExpression)
2
```

Quindi, vediamo che i due seguenti sono equivalenti:

```
1+1
eval(:(1+1))
```

Perché dovremmo passare attraverso la sintassi molto più complicata di `eval(:(1+1))` se vogliamo solo scoprire cosa è uguale a `1 + 1`? La ragione di base è che possiamo definire un'espressione in un punto del nostro codice, potenzialmente modificarla in seguito, e quindi valutarla in un secondo momento. Questo può potenzialmente aprire nuove potenti funzionalità al programmatore Julia. Le espressioni sono una componente chiave della [metaprogrammazione](#) in Julia.

### Creazione di espressioni

Esistono diversi metodi che possono essere utilizzati per creare lo stesso tipo di espressione. Le [espressioni intro](#) menzionavano la sintassi `:()`. Forse il miglior punto di partenza, tuttavia è con le stringhe. Ciò aiuta a rivelare alcune delle somiglianze fondamentali tra espressioni e stringhe in Julia.

#### Crea espressione da stringa

Dalla [documentazione di Julia](#):

## Ogni programma Julia inizia la vita come una stringa

In altre parole, qualsiasi script di Julia è semplicemente scritto in un file di testo, che non è altro che una stringa di caratteri. Allo stesso modo, qualsiasi comando di Julia inserito in un interprete è solo una stringa di caratteri. Il ruolo di Julia o di qualsiasi altro linguaggio di programmazione è quindi quello di interpretare e valutare le stringhe di caratteri in modo logico e prevedibile, in modo che le stringhe di caratteri possano essere utilizzate per descrivere ciò che il programmatore vuole che il computer compia.

Quindi, un modo per creare un'espressione è usare la funzione `parse()` applicata a una stringa. La seguente espressione, una volta valutata, assegnerà il valore di 2 al simbolo `x`.

```
MyStr = "x = 2"
MyExpr = parse(MyStr)
julia> x
ERROR: UndefVarError: x not defined
eval(MyExpr)
julia> x
2
```

## Crea espressione usando `:` Sintassi

```
MyExpr2 = :(x = 2)
julia> MyExpr == MyExpr2
true
```

Nota che con questa sintassi, Julia tratterà automaticamente i nomi degli oggetti come riferiti ai simboli. Possiamo vedere questo se guardiamo gli `args` dell'espressione. (Vedi [Campi degli oggetti espressione](#) per ulteriori dettagli sul campo `args` in un'espressione.)

```
julia> MyExpr2.args
2-element Array{Any,1}:
 :x
 2
```

## Crea espressione usando la funzione `Expr()`

```
MyExpr3 = Expr(:(=), :x, 2)
MyExpr3 == MyExpr
```

Questa sintassi è basata sulla [notazione del prefisso](#). In altre parole, il primo argomento specificato per la funzione `Expr()` è la `head` o il prefisso. I restanti sono gli `arguments` dell'espressione. La `head` determina quali operazioni saranno eseguite sugli argomenti.

Per maggiori dettagli su questo, vedere [Campi degli oggetti espressione](#)

Quando si utilizza questa sintassi, è importante distinguere tra l'uso di oggetti e simboli per gli oggetti. Ad esempio, nell'esempio sopra, l'espressione assegna il valore di 2 al simbolo `:x`, un'operazione perfettamente ragionevole. Se usassimo `x` se stesso in un'espressione come quella, otterremmo il risultato insensato:

```
julia> Expr(:(=), x, 5)
:(2 = 5)
```

Allo stesso modo, se esaminiamo gli `args` che vediamo:

```
julia> Expr(:(=), x, 5).args
2-element Array{Any,1}:
 2
 5
```

Pertanto, la funzione `Expr()` non esegue la stessa trasformazione automatica in simboli come la sintassi `:( )` per la creazione di espressioni.

### Crea espressioni multi-linea usando `quote...end`

```
MyQuote =
quote
    x = 2
    y = 3
end
julia> typeof(MyQuote)
Expr
```

Si noti che con `quote...end` possiamo creare espressioni che contengono altre espressioni nel loro campo `args`:

```
julia> typeof(MyQuote.args[2])
Expr
```

Vedi [Fields of Expression Objects](#) per ulteriori informazioni su questo `args`.

### Ulteriori informazioni sulla creazione di espressioni

Questo esempio fornisce solo le basi per la creazione di espressioni. Vedi anche, ad esempio, [Interpolazione ed espressioni](#) e [campi di oggetti espressione](#) per ulteriori informazioni sulla creazione di espressioni più complesse e avanzate.

### Campi di oggetti espressione

Come menzionato nelle [espressioni Intro to Expressions](#) sono un tipo specifico di oggetto in Julia. In quanto tali, hanno campi. I due campi più usati di un'espressione sono la sua `head` e le sue `args`. Ad esempio, considera l'espressione

```
MyExpr3 = Expr(:(=), :x, 2)
```

discusso in [Creazione di espressioni](#). Possiamo vedere la `head` e gli `args` come segue:

```
julia> MyExpr3.head
:(=)
```

```
julia> MyExpr3.args
2-element Array{Any,1}:
 :x
 2
```

Le espressioni sono basate sulla [notazione del prefisso](#) . In quanto tale, la `head` generalmente specifica l'operazione che deve essere eseguita sugli `args` . La testa deve essere di tipo `Julia Symbol` .

Quando un'espressione deve assegnare un valore (quando viene valutato), generalmente utilizza una testa di `: (=)` . Ovviamente ci sono ovvie variazioni che possono essere impiegate, ad esempio:

```
ex1 = Expr(:(+=), :x, 2)
```

### **: call for expression heads**

Un altro `head` comune per le espressioni è `:call` . Per esempio

```
ex2 = Expr(:call, :(*), 2, 3)
eval(ex2) ## 6
```

Seguendo le convenzioni della notazione del prefisso, gli operatori vengono valutati da sinistra a destra. Quindi, questa espressione qui significa che chiameremo la funzione che è specificata sul primo elemento di `args` sugli elementi successivi. Analogamente potremmo avere:

```
julia> ex2a = Expr(:call, :(-), 1, 2, 3)
:(1 - 2 - 3)
```

O altre funzioni potenzialmente più interessanti, ad es

```
julia> ex2b = Expr(:call, :rand, 2,2)
:(rand(2,2))

julia> eval(ex2b)
2x2 Array{Float64,2}:
 0.429397  0.164478
 0.104994  0.675745
```

### **Determinazione automatica della `head` quando si usa `: ()` notazione di creazione di espressioni**

Nota che `:call` è implicitamente usata come la testa in certe costruzioni di espressioni, ad es

```
julia> :(x + 2).head
:call
```

Quindi, con la sintassi `: ()` per la creazione di espressioni, Julia cercherà di determinare automaticamente la testa corretta da utilizzare. Allo stesso modo:

```
julia> :(x = 2).head
: (=)
```

Infatti, se non sei sicuro di quale sia la testa giusta da usare per un'espressione che stai formando usando, ad esempio, `Expr()` questo può essere uno strumento utile per ottenere consigli e idee su cosa usare.

## Interpolazione ed espressioni

La [creazione di espressioni](#) indica che le espressioni sono strettamente correlate alle stringhe. In quanto tali, i principi di interpolazione all'interno delle stringhe sono rilevanti anche per le espressioni. Per esempio, nell'interpolazione di base delle stringhe, possiamo avere qualcosa di simile:

```
n = 2
julia> MyString = "there are $n ducks"
"there are 2 ducks"
```

Usiamo il segno `$` per inserire il valore di `n` nella stringa. Possiamo usare la stessa tecnica con le espressioni. Per esempio

```
a = 2
ex1 = :(x = 2*$a) ##      :(x = 2 * 2)
a = 3
eval(ex1)
x # 4
```

Contrasto questo:

```
a = 2
ex2 = :(x = 2*a) # :(x = 2a)
a = 3
eval(ex2)
x # 6
```

Pertanto, con il primo esempio, impostiamo in anticipo il valore di `a` che verrà utilizzato nel momento in cui viene valutata l'espressione. Con il secondo esempio, tuttavia, il compilatore Julia guarderà a `a` per trovare il suo valore *al momento della valutazione* per la nostra espressione.

## Riferimenti esterni sulle espressioni

Ci sono un certo numero di risorse web utili che possono aiutare ulteriormente la tua conoscenza delle espressioni in Julia. Questi includono:

- [Julia Docs - Metaprogramming](#)
- [Wikibooks - Julia Metaprogramming](#)
- [I macro, le espressioni, ecc. Di Julia per e dai confusi, di Gray Calhoun](#)
- [Mese di Julia - Metaprogramming, di Andrew Collier](#)
- [Differenziazione simbolica in Julia, di John Myles White](#)

Post SO:

- [Cos'è un "simbolo" in Julia? Risposta di Stefan Karpinski](#)
- [Perché Julia esprime questa espressione in questo modo complesso?](#)
- [Spiegazione dell'esempio di interpolazione dell'espressione Julia](#)

Leggi espressioni online: <https://riptutorial.com/it/julia-lang/topic/5805/espressioni>

# Capitolo 15: funzioni

## Sintassi

- `f(n) = ...`
- `funzione f(n) ... fine`
- `n :: Tipo`
- `x -> ...`
- `f(n) do ... end`

## Osservazioni

Oltre alle funzioni generiche (che sono le più comuni), esistono anche funzioni integrate. Tali funzioni includono `is`, `isa`, `typeof`, `throw` e funzioni simili. Le funzioni built-in sono in genere implementate in C anziché in Julia, quindi non possono essere specializzate sui tipi di argomenti per la spedizione.

## Examples

### Piazza un numero

Questa è la sintassi più semplice per definire una funzione:

```
square(n) = n * n
```

Per chiamare una funzione, usa parentesi tonde (senza spazi intermedi):

```
julia> square(10)
100
```

Le funzioni sono oggetti in Julia e possiamo mostrarle in [REPL](#) come con qualsiasi altro oggetto:

```
julia> square
square (generic function with 1 method)
```

Tutte le funzioni di Julia sono generiche (altrimenti conosciute come [polimorfiche](#)) per impostazione predefinita. La nostra funzione `square` funziona altrettanto bene con valori in virgola mobile:

```
julia> square(2.5)
6.25
```

... o anche [matrici](#) :

```
julia> square([2 4
              2 1])
2×2 Array{Int64,2}:
 12  12
  6   9
```

## Funzioni ricorsive

### Ricorsione semplice

Usando la ricorsione e l' [operatore condizionale ternario](#) , possiamo creare un'implementazione alternativa della funzione `factorial` integrata:

```
myfactorial(n) = n == 0 ? 1 : n * myfactorial(n - 1)
```

Uso:

```
julia> myfactorial(10)
3628800
```

### Lavorare con gli alberi

Le funzioni ricorsive sono spesso le più utili su strutture dati, in particolare strutture di dati ad albero. Poiché le [espressioni](#) in Julia sono strutture ad albero, la ricorsione può essere molto utile per la [metaprogrammazione](#) . Ad esempio, la funzione seguente raccoglie un insieme di tutte le teste utilizzate in un'espressione.

```
heads(ex::Expr) = reduce(Union{Set{Symbol}}, (heads(a) for a in ex.args))
heads(::Any) = Set{Symbol}()
```

Possiamo verificare che la nostra funzione funzioni come previsto:

```
julia> heads(: (7 + 4x > 1 > A[0]))
Set{Symbol[:comparison, :ref, :call]}
```

Questa funzione è compatta e utilizza una varietà di tecniche più avanzate, come la `reduce` [funzione di ordine superiore](#) , il tipo di dati `Set` e le espressioni del generatore.

### Introduzione alla spedizione

Possiamo usare la `::` sintassi per inviare il [tipo](#) di argomento.

```
describe(n::Integer) = "integer $n"
describe(n::AbstractFloat) = "floating point $n"
```

Uso:



```
julia> describe(10)
"integer 10"

julia> describe(1.0)
"floating point 1.0"
```

A differenza di molte lingue, che in genere forniscono una distribuzione multipla statica o una singola spedizione dinamica, Julia ha una distribuzione multipla dinamica completa. Cioè, le funzioni possono essere specializzate per più di un argomento. Ciò è utile quando si definiscono metodi specializzati per operazioni su determinati tipi e metodi di fallback per altri tipi.

```
describe(n::Integer, m::Integer) = "integers n=$n and m=$m"
describe(n, m::Integer) = "only m=$m is an integer"
describe(n::Integer, m) = "only n=$n is an integer"
```

Uso:

```
julia> describe(10, 'x')
"only n=10 is an integer"

julia> describe('x', 10)
"only m=10 is an integer"

julia> describe(10, 10)
"integers n=10 and m=10"
```

## Argomenti opzionali

Julia consente alle funzioni di prendere argomenti opzionali. Dietro le quinte, questo è implementato come un altro caso speciale di dispacciamento multiplo. Ad esempio, risolviamo il famoso [problema di Fizz Buzz](#). Per impostazione predefinita, lo faremo per i numeri compresi nell'intervallo `1:10`, ma consentiremo un valore diverso se necessario. Permetteremo anche l'uso di frasi diverse per `Fizz` o `Buzz`.

```
function fizzbuzz(xs=1:10, fizz="Fizz", buzz="Buzz")
    for i in xs
        if i % 15 == 0
            println(fizz, buzz)
        elseif i % 3 == 0
            println(fizz)
        elseif i % 5 == 0
            println(buzz)
        else
            println(i)
        end
    end
end
```

Se ispezioniamo `fizzbuzz` nel REPL, si dice che ci sono quattro metodi. È stato creato un metodo per ciascuna combinazione di argomenti consentita.

```
julia> fizzbuzz
```

```
fizzbuzz (generic function with 4 methods)

julia> methods(fizzbuzz)
# 4 methods for generic function "fizzbuzz":
fizzbuzz() at REPL[96]:2
fizzbuzz(xs) at REPL[96]:2
fizzbuzz(xs, fizz) at REPL[96]:2
fizzbuzz(xs, fizz, buzz) at REPL[96]:2
```

Possiamo verificare che i nostri valori predefiniti vengano utilizzati quando non vengono forniti parametri:

```
julia> fizzbuzz()
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
```

ma che i parametri opzionali sono accettati e rispettati se li forniamo:

```
julia> fizzbuzz(5:8, "fuzz", "bizz")
bizz
fuzz
7
8
```

## Invio parametrico

È frequente il caso che una funzione debba essere inviata su tipi parametrici, come `Vector{T}` o `Dict{K,V}`, ma i parametri del tipo non sono corretti. Questo caso può essere risolto utilizzando la spedizione parametrica:

```
julia> foo{T<:Number}(xs::Vector{T}) = @show xs .+ 1
foo (generic function with 1 method)

julia> foo(xs::Vector) = @show xs
foo (generic function with 2 methods)

julia> foo([1, 2, 3])
xs .+ 1 = [2,3,4]
3-element Array{Int64,1}:
 2
 3
 4

julia> foo([1.0, 2.0, 3.0])
xs .+ 1 = [2.0,3.0,4.0]
3-element Array{Float64,1}:
 2.0
```

```
3.0
4.0

julia> foo(["x", "y", "z"])
xs = String["x","y","z"]
3-element Array{String,1}:
 "x"
 "y"
 "z"
```

Si potrebbe essere tentati di scrivere semplicemente `xs::Vector{Number}` . Ma questo funziona solo per oggetti il cui tipo è esplicitamente `Vector{Number}` :

```
julia> isa(Number[1, 2], Vector{Number})
true

julia> isa(Int[1, 2], Vector{Number})
false
```

Ciò è dovuto [all'invarianza parametrica](#) : l'oggetto `Int[1, 2]` *non* è un `Vector{Number}` , perché può contenere solo `Int` s, mentre un `Vector{Number}` dovrebbe essere in grado di contenere qualsiasi tipo di numero.

## Scrivere codice generico

Dispatch è una funzionalità incredibilmente potente, ma spesso è meglio scrivere codice generico che funzioni per tutti i tipi, invece di specializzare il codice per ogni tipo. La scrittura di codice generico evita la duplicazione del codice.

Ad esempio, ecco il codice per calcolare la somma dei quadrati di un vettore di numeri interi:

```
function sumsq(v::Vector{Int})
    s = 0
    for x in v
        s += x ^ 2
    end
    s
end
```

Ma questo codice funziona *solo* per un vettore di `Int` s. Non funzionerà su un `UnitRange` :

```
julia> sumsq(1:10)
ERROR: MethodError: no method matching sumsq(::UnitRange{Int64})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
```

Non funzionerà su un `Vector{Float64}` :

```
julia> sumsq([1.0, 2.0])
ERROR: MethodError: no method matching sumsq(::Array{Float64,1})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
```

Dovrebbe essere un modo migliore per scrivere questa funzione `sumsq`

```
function sumsq(v::AbstractVector)
    s = zero(eltype(v))
    for x in v
        s += x ^ 2
    end
    s
end
```

Questo funzionerà sui due casi sopra elencati. Ma ci sono alcune collezioni che potremmo voler sommare i quadrati di quello non sono affatto vettori, in nessun senso. Per esempio,

```
julia> sumsq(take(countfrom(1), 100))
ERROR: MethodError: no method matching sumsq(::Base.Take{Base.Count{Int64}})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
  sumsq(::AbstractArray{T,1}) at REPL[11]:2
```

dimostra che non possiamo sommare i quadrati di un [pigro iterabile](#) .

Un'implementazione ancora più generica è semplicemente

```
function sumsq(v)
    s = zero(eltype(v))
    for x in v
        s += x ^ 2
    end
    s
end
```

Che funziona in tutti i casi:

```
julia> sumsq(take(countfrom(1), 100))
338350
```

Questo è il codice Julia più idiomatico e può gestire ogni tipo di situazione. In alcuni altri linguaggi, la rimozione delle annotazioni di tipo può influire sulle prestazioni, ma non è questo il caso in Julia; solo la [stabilità del tipo](#) è importante per le prestazioni.

## Fattoriale imperativo

Una sintassi di lunga durata è disponibile per la definizione di funzioni multi-linea. Questo può essere utile quando usiamo strutture imperative come i loop. L'espressione nella posizione di coda viene restituita. Ad esempio, la funzione seguente usa un [ciclo for](#) per calcolare il [fattoriale](#) di qualche intero `n` :

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
```

```
fact
end
```

Uso:

```
julia> myfactorial(10)
3628800
```

Nelle funzioni più lunghe, è comune vedere la dichiarazione di `return` utilizzata. Il `return` affermazione non è necessaria in posizione di coda, ma è ancora talvolta utilizzato per chiarezza. Ad esempio, un altro modo di scrivere la funzione sopra sarebbe

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
    return fact
end
```

che è identico nel comportamento alla funzione di cui sopra.

## Funzioni anonime

## Sintassi della freccia

Le funzioni anonime possono essere create usando la sintassi `->`. Questo è utile per passare funzioni a funzioni di [ordine superiore](#), come la funzione `map`. La funzione seguente calcola il quadrato di ciascun numero in una [matrice](#) `A`

```
squareall(A) = map(x -> x ^ 2, A)
```

Un esempio di utilizzo di questa funzione:

```
julia> squareall(1:10)
10-element Array{Int64,1}:
 1
 4
 9
16
25
36
49
64
81
100
```

## Sintassi multilinea

Le funzioni anonime multilinea possono essere create usando la sintassi della `function`. Ad

esempio, il seguente esempio calcola i **fattoriali** dei primi `n` numeri, ma usando una funzione anonima al posto del `factorial` incorporato.

```
julia> map(function (n)
            product = one(n)
            for i in 1:n
                product *= i
            end
            product
        end, 1:10)
10-element Array{Int64,1}:
 1
 2
 6
24
120
720
5040
40320
362880
3628800
```

## Do la sintassi del blocco

Perché è così comune per passare una funzione anonima come primo argomento a una funzione, c'è un `do` sintassi del blocco. La sintassi

```
map(A) do x
    x ^ 2
end
```

è equivalente a

```
map(x -> x ^ 2, A)
```

ma il primo può essere più chiaro in molte situazioni, specialmente se viene eseguita molta computazione nella funzione anonima. `do` sintassi dei blocchi è particolarmente utile per l' **input e l'output dei file** per ragioni di gestione delle risorse.

**Leggi funzioni online:** <https://riptutorial.com/it/julia-lang/topic/3079/funzioni>

# Capitolo 16: Funzioni di ordine superiore

## Sintassi

- `foreach (f, xs)`
- `mappa (f, xs)`
- `filtro (f, xs)`
- `ridurre (f, v0, xs)`
- `foldl (f, v0, xs)`
- `foldr (f, v0, xs)`

## Osservazioni

Le funzioni possono essere accettate come parametri e possono anche essere prodotte come tipi di ritorno. In effetti, le funzioni possono essere create all'interno del corpo di altre funzioni. Queste funzioni interiori sono note come [chiusure](#).

## Examples

### Funziona come argomenti

[Le funzioni](#) sono oggetti in Julia. Come qualsiasi altro oggetto, possono essere passati come argomenti ad altre funzioni. Le funzioni che accettano le funzioni sono conosciute come funzioni di [ordine superiore](#).

Ad esempio, possiamo implementare un equivalente della funzione `foreach` della libreria standard prendendo una funzione `f` come primo parametro.

```
function myforeach(f, xs)
    for x in xs
        f(x)
    end
end
```

Possiamo verificare che questa funzione funzioni effettivamente come ci aspettiamo:

```
julia> myforeach(println, ["a", "b", "c"])
a
b
c
```

Prendendo una funzione come *primo* parametro, invece di un parametro successivo, possiamo usare la sintassi del blocco `Do` di Julia. La sintassi del blocco `do` è solo un modo conveniente per passare una [funzione anonima](#) come primo argomento di una funzione.

```
julia> myforeach([1, 2, 3]) do x
```

```
println(x^x)
end
1
4
27
```

La nostra implementazione di `myforeach` sopra è approssimativamente equivalente alla funzione `foreach` integrata. Esistono anche molte altre funzioni di ordine superiore incorporate.

Le funzioni di ordine superiore sono piuttosto potenti. A volte, quando si lavora con funzioni di ordine superiore, le operazioni esatte eseguite diventano irrilevanti e i programmi possono diventare piuttosto astratti. I [combinatori](#) sono esempi di sistemi di funzioni di ordine superiore altamente astratte.

## Mappare, filtrare e ridurre

Due delle funzioni di ordine superiore più fondamentali incluse nella libreria standard sono la `map` e il `filter`. Queste funzioni sono generiche e possono funzionare su qualsiasi [iterabile](#). In particolare, sono adatti per i calcoli sugli [array](#).

Supponiamo di avere un set di dati delle scuole. Ogni scuola insegna un argomento particolare, ha un numero di classi e un numero medio di studenti per classe. Possiamo modellare una scuola con il seguente [tipo immutabile](#):

```
immutable School
  subject::Symbol
  nclasses::Int
  nstudents::Int # average no. of students per class
end
```

Il nostro set di dati delle scuole sarà una `Vector{School}`:

```
dataset = [School(:math, 3, 30), School(:math, 5, 20), School(:science, 10, 5)]
```

Supponiamo di voler trovare il numero totale di studenti iscritti a un programma di matematica. Per fare ciò, sono necessari diversi passaggi:

- dobbiamo restringere il set di dati solo alle scuole che insegnano la matematica ( `filter` )
- dobbiamo calcolare il numero di studenti in ogni scuola ( `map` )
- e dobbiamo ridurre quella lista di numeri di studenti a un singolo valore, la somma ( `reduce` )

Una soluzione ingenua (non molto performante) sarebbe semplicemente quella di utilizzare direttamente quelle tre funzioni di ordine superiore.

```
function nmath(data)
  maths = filter(x -> x.subject === :math, data)
  students = map(x -> x.nclasses * x.nstudents, maths)
  reduce(+, 0, students)
end
```



e verifichiamo che ci sono 190 studenti di matematica nel nostro set di dati:

```
julia> nmath(dataset)
190
```

Esistono delle funzioni per combinare queste funzioni e quindi migliorare le prestazioni. Ad esempio, avremmo potuto utilizzare la funzione `mapreduce` per eseguire la mappatura e la riduzione di un passo, il che farebbe risparmiare tempo e memoria.

La `reduce` è significativa solo per **le operazioni associative** come `+`, ma a volte è utile eseguire una riduzione con un'operazione non associativa. L'ordine superiore funzioni `foldl` e `foldr` sono forniti per forzare un particolare ordine riduzione.

Leggi Funzioni di ordine superiore online: <https://riptutorial.com/it/julia-lang/topic/6955/funzioni-di-ordine-superiore>

# Capitolo 17: Ingresso

## Sintassi

- `linea di lettura()`
- `readlines ()`
- `ReadString (STDIN)`
- `chomp (str)`
- `open (f, file)`
- `eachLine (io)`
- `ReadString (file)`
- `lettura (file)`
- `readcsv (file)`
- `readdlm (file)`

## Parametri

Parametro	Dettagli
<code>chomp(str)</code>	<b>Rimuovi fino a una riga finale finale da una stringa.</b>
<code>str</code>	La stringa per rimuovere una nuova riga finale da. Nota che le <a href="#">stringhe</a> sono immutabili per convenzione. Questa funzione restituisce una nuova stringa.
<code>open(f, file)</code>	<b>Aprire un file, chiamare la funzione e chiudere il file in seguito.</b>
<code>f</code>	La funzione per richiamare il flusso di I / O che apre il file genera.
<code>file</code>	Il percorso del file da aprire.

## Examples

### Lettura di una stringa da input standard

Lo stream `STDIN` in Julia fa riferimento allo [standard input](#) . Questo può rappresentare l'input dell'utente, per i programmi di riga comandi interattivi o l'input da un file o [pipeline](#) che è stato reindirizzato nel programma.

La funzione `readline` , quando non viene fornito alcun argomento, leggerà i dati da `STDIN` fino a quando non viene rilevata una nuova riga o il flusso `STDIN` entra nello stato di fine file. Questi due casi possono essere distinti dal fatto che il carattere `\n` sia stato letto come carattere finale:

```
julia> readline()
some stuff
```

```
"some stuff\n"

julia> readline() # Ctrl-D pressed to send EOF signal here
""
```

Spesso, per i programmi interattivi, non ci interessa lo stato EOF e vogliamo solo una stringa. Ad esempio, potremmo richiedere all'utente di inserire:

```
function askname()
    print("Enter your name: ")
    readline()
end
```

Questo non è del tutto soddisfacente, tuttavia, a causa della nuova riga aggiuntiva:

```
julia> askname()
Enter your name: Julia
"Julia\n"
```

La funzione `chomp` è disponibile per rimuovere fino a una riga finale da una stringa. Per esempio:

```
julia> chomp("Hello, World!")
"Hello, World!"

julia> chomp("Hello, World!\n")
"Hello, World!"
```

Potremmo quindi aumentare la nostra funzione con `chomp` modo che il risultato sia come previsto:

```
function askname()
    print("Enter your name: ")
    chomp(readline())
end
```

che ha un risultato più desiderabile:

```
julia> askname()
Enter your name: Julia
"Julia"
```

A volte, potremmo voler leggere quante più righe possibile (finché il flusso di input non entra nello stato di fine del file). La funzione `readlines` fornisce questa funzionalità.

```
julia> readlines() # note Ctrl-D is pressed after the last line
A, B, C, D, E, F, G
H, I, J, K, LMNO, P
Q, R, S
T, U, V
W, X
Y, Z
6-element Array{String,1}:
```

```
"A, B, C, D, E, F, G\n"
"H, I, J, K, LMNO, P\n"
"Q, R, S\n"
"T, U, V\n"
"W, X\n"
"Y, Z\n"
```

## 0.5.0

Ancora una volta, se non amiamo le nuove righe alla fine delle righe lette dai `readlines`, possiamo usare la funzione `chomp` per rimuoverle. Questa volta, **trasmettiamo** la funzione `chomp` attraverso l'intero array:

```
julia> chomp.(readlines())
A, B, C, D, E, F, G
H, I, J, K, LMNO, P
Q, R, S
T, U, V
W, X
Y, Z
6-element Array{String,1}:
 "A, B, C, D, E, F, G"
 "H, I, J, K, LMNO, P"
 "Q, R, S"
 "T, U, V"
 "W, X"
 "Y, Z"
```

Altre volte, potremmo non interessarci affatto delle linee e semplicemente voler leggere il più possibile una singola stringa. La funzione `readstring` realizza questo:

```
julia> readstring(STDIN)
If music be the food of love, play on,
Give me excess of it; that surfeiting,
The appetite may sicken, and so die. # [END OF INPUT]
"If music be the food of love, play on,\nGive me excess of it; that surfeiting,\nThe appetite
may sicken, and so die.\n"
```

(il `# [END OF INPUT]` non fa parte dell'input originale, è stato aggiunto per chiarezza.)

Si noti che il `readstring` deve essere passato all'argomento `STDIN`.

## Lettura di numeri da input standard

La lettura dei numeri dall'input standard è una combinazione di stringhe di lettura e l'analisi di tali stringhe come numeri.

La funzione di `parse` viene utilizzata per analizzare una stringa nel tipo di numero desiderato:

```
julia> parse{Int, "17"}
17

julia> parse{Float32, "-3e6"}
-3.0f6
```

Il formato previsto da `parse(T, x)` è simile a, ma non esattamente lo stesso, del formato che Julia si aspetta dai [numeri letterali](#) :

```
julia> -00000023
-23

julia> parse{Int, "-00000023"}
-23

julia> 0x23 |> Int
35

julia> parse{Int, "0x23"}
35

julia> 1_000_000
1000000

julia> parse{Int, "1_000_000"}
ERROR: ArgumentError: invalid base 10 digit '_' in "1_000_000"
 in tryparse_internal(::Type{Int64}, ::String, ::Int64, ::Int64, ::Int64, ::Bool) at
 ./parse.jl:88
 in parse(::Type{Int64}, ::String) at ./parse.jl:152
```

La combinazione delle funzioni di `parse` e `readline` ci consente di leggere un singolo numero da una riga:

```
function asknumber()
    print("Enter a number: ")
    parse{Float64, readline()}
end
```

che funziona come previsto:

```
julia> asknumber()
Enter a number: 78.3
78.3
```

Si applicano i soliti avvertimenti sulla [precisione in virgola mobile](#) . Nota che l' `parse` può essere utilizzata con `BigInt` e `BigFloat` per rimuovere o ridurre al minimo la perdita di precisione.

A volte, è utile leggere più di un numero dalla stessa riga. In genere, la riga può essere divisa con spazi:

```
function askints()
    print("Enter some integers, separated by spaces: ")
    [parse{Int, x} for x in split(readline())]
end
```

che può essere usato come segue:

```
julia> askints()
Enter some integers, separated by spaces: 1 2 3 4
4-element Array{Int64,1}:
 1
 2
 3
 4
```

```
1
2
3
4
```

## Lettura dei dati da un file

# Lettura di stringhe o byte

I file possono essere aperti per la lettura utilizzando la funzione `open`, che viene spesso utilizzata insieme alla [sintassi del blocco](#):

```
open("myfile") do f
    for (i, line) in enumerate(eachline(f))
        print("Line $i: $line")
    end
end
```

Supponiamo che `myfile` esista e che il suo contenuto sia

```
What's in a name? That which we call a rose
By any other name would smell as sweet.
```

Quindi, questo codice produrrebbe il seguente risultato:

```
Line 1: What's in a name? That which we call a rose
Line 2: By any other name would smell as sweet.
```

Nota che `eachline` è un pigro [iterabile](#) sulle linee del file. Si preferisce la `readlines` per motivi di prestazioni.

Perché `do` bloccare la sintassi è solo zucchero sintattico per le funzioni anonime, possiamo passare le funzioni di chiamata `open` anche:

```
julia> open(readstring, "myfile")
"What's in a name? That which we call a rose\nBy any other name would smell as sweet.\n"

julia> open(read, "myfile")
84-element Array{UInt8,1}:
 0x57
 0x68
 0x61
 0x74
 0x27
 0x73
 0x20
 0x69
 0x6e
 0x20
  ⋮
 0x73
 0x20
```

```
0x73
0x77
0x65
0x65
0x74
0x2e
0x0a
```

Le funzioni `read` e `readstring` forniscono metodi convenienti che apriranno automaticamente un file:

```
julia> readstring("myfile")
"What's in a name? That which we call a rose\nBy any other name would smell as sweet.\n"
```

## Leggere i dati strutturati

Supponiamo di avere un [file CSV](#) con i seguenti contenuti, in un file denominato `file.csv` :

```
Make,Model,Price
Foo,2015A,8000
Foo,2015B,14000
Foo,2016A,10000
Foo,2016B,16000
Bar,2016Q,20000
```

Quindi potremmo usare la funzione `readcsv` per leggere questi dati in una `Matrix` :

```
julia> readcsv("file.csv")
6×3 Array{Any,2}:
 "Make"  "Model"  "Price"
 "Foo"   "2015A"   8000
 "Foo"   "2015B"  14000
 "Foo"   "2016A"  10000
 "Foo"   "2016B"  16000
 "Bar"   "2016Q"  20000
```

Se invece il file era delimitato da tabulazioni, in un file denominato `file.tsv` , è possibile utilizzare la funzione `readdlm` , con l'argomento `delim` impostato su `'\t'` . [Carichi di lavoro](#) più avanzati dovrebbero utilizzare il [pacchetto CSV.jl](#).

**Leggi Ingresso online:** <https://riptutorial.com/it/julia-lang/topic/7201/ingresso>

# Capitolo 18: iterabili

## Sintassi

- avviare (ITR)
- next (itr, s)
- fatto (itr, s)
- prendere (itr, n)
- drop (itr, n)
- Ciclo (ITR)
- Base.product (xs, ys)

## Parametri

Parametro	Dettagli
Per	Tutte le funzioni
itr	L'iterabile su cui operare.
Per	next e done
s	Uno stato iteratore che descrive la posizione corrente dell'iterazione.
Per	take e drop
n	Il numero di elementi da prendere o rilasciare.
Per	Base.product
xs	L'iterabile per prendere i primi elementi di coppie da.
ys	L'iterabile per prendere i secondi elementi di coppie da.
...	(Nota che il <code>product</code> accetta un numero qualsiasi di argomenti, se ne vengono forniti più di due, costruirà tuple di lunghezza maggiore di due.)

## Examples

### Nuovo tipo iterabile

In Julia, quando eseguo il ciclo su un oggetto iterabile, `I` finito con la sintassi `for` :

```
for i = I    # or "for i in I"
    # body
```



```
end
```

Dietro le quinte, questo è tradotto in:

```
state = start(I)
while !done(I, state)
    (i, state) = next(I, state)
    # body
end
```

Pertanto, se si desidera `I` sia un iterable, è necessario definire i metodi `start`, `next` e `done` per il suo tipo. Supponiamo di definire un tipo `Foo` contenente un `array` come uno dei campi:

```
type Foo
    bar::Array{Int,1}
end
```

Istanziamo un oggetto `Foo` facendo:

```
julia> I = Foo([1,2,3])
Foo{Array{Int64,1}}

julia> I.bar
3-element Array{Int64,1}:
 1
 2
 3
```

Se vogliamo iterare attraverso `Foo`, con ogni `bar` elementi che viene restituita da ogni iterazione, definiamo i metodi:

```
import Base: start, next, done

start(I::Foo) = 1

next(I::Foo, state) = (I.bar[state], state+1)

function done(I::Foo, state)
    if state == length(I.bar)
        return true
    end
    return false
end
```

Nota che poiché queste `funzioni` appartengono al modulo `Base`, dobbiamo prima `import` loro nomi prima di aggiungere nuovi metodi.

Dopo aver definito i metodi, `Foo` è compatibile con l'interfaccia iteratore:

```
julia> for i in I
    println(i)
end
```

```
1
2
3
```

## Combinare Iterables pigri

La libreria standard include una ricca collezione di file iterabili pigri (e le librerie come [Iterators.jl](#) forniscono ancora di più). È possibile creare file Lazy iterables per creare iterables più potenti in un tempo costante. I più importanti iterables pigri sono [take and drop](#), da cui è possibile creare molte altre funzioni.

## Affetta un po' iterabile

Le matrici possono essere affettate con la notazione di sezione. Ad esempio, il seguente restituisce il decimo al quindicesimo elemento di un array, inclusi:

```
A[10:15]
```

Tuttavia, la notazione di sezione non funziona con tutti gli iterabili. Ad esempio, non possiamo suddividere un'espressione di generatore:

```
julia> (i^2 for i in 1:10)[3:5]
ERROR: MethodError: no method matching getindex(::Base.Generator{UnitRange{Int64},##1#2},
::UnitRange{Int64})
```

La segmentazione delle [stringhe](#) potrebbe non avere il comportamento Unicode previsto:

```
julia> "aaaa"[2:3]
ERROR: UnicodeError: invalid character index
in getindex(::String, ::UnitRange{Int64}) at ./strings/string.jl:130

julia> "aaaa"[3:4]
"a"
```

Possiamo definire una funzione `lazysub(itr, range::UnitRange)` per fare questo tipo di slicing su iterables arbitrari. Questo è definito in termini di `take` and `drop`:

```
lazysub(itr, r::UnitRange) = take(drop(itr, first(r) - 1), last(r) - first(r) + 1)
```

L'implementazione qui funziona perché per il valore `UnitRange a:b`, vengono eseguiti i seguenti passaggi:

- elimina i primi elementi  $a-1$
- ritiene  $a$  elemento esimo,  $a+1$  -esimo elemento, e così via, fino a che la  $a+(b-a)=b$  esimo elemento

In totale, vengono presi gli elementi  $b-a$ . Possiamo confermare che la nostra implementazione è corretta in ogni caso sopra:

```
julia> collect(lazysub("aaaa", 2:3))
2-element Array{Char,1}:
 'a'
 'a'

julia> collect(lazysub((i^2 for i in 1:10), 3:5))
3-element Array{Int64,1}:
 9
16
25
```

## Pigramente spostare un iterable circolare

L'operazione `circshift` sugli array sposterà la matrice come se fosse un cerchio, quindi la ricollocherà. Per esempio,

```
julia> circshift(1:10, 3)
10-element Array{Int64,1}:
 8
 9
10
 1
 2
 3
 4
 5
 6
 7
```

Possiamo farlo pigramente per tutti gli iterabili? Possiamo usare il `cycle`, `drop` e `take` iterabili per implementare questa funzionalità.

```
lazycircshift(itr, n) = take(drop(cycle(itr), length(itr) - n), length(itr))
```

Insieme con i tipi pigri che sono più performanti in molte situazioni, questo ci permette di fare `circshift` funzionalità simile a quella di `circshift` su tipi che altrimenti non la supportano:

```
julia> circshift("Hello, World!", 3)
ERROR: MethodError: no method matching circshift(::String, ::Int64)
Closest candidates are:
  circshift(::AbstractArray{T,N}, ::Real) at abstractarraymath.jl:162
  circshift(::AbstractArray{T,N}, ::Any) at abstractarraymath.jl:195

julia> String(collect(lazycircshift("Hello, World!", 3)))
"ld!Hello, Wor"
```

0.5.0

## Fare una tabella di moltiplicazione

Creiamo una [tabella di moltiplicazione](#) usando le funzioni iterabili pigre per creare una matrice.

Le funzioni chiave da usare qui sono:

- `Base.product` , che calcola un [prodotto cartesiano](#) .
- `prod` , che calcola un prodotto normale (come in moltiplicazione)
- `:` , che crea un intervallo
- `map` , che è una funzione di ordine superiore che applica una funzione a ciascun elemento di una raccolta

La soluzione è:

```
julia> map(prod, Base.product(1:10, 1:10))
10×10 Array{Int64,2}:
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 10 12 14 16 18 20
 3  6  9 12 15 18 21 24 27 30
 4  8 12 16 20 24 28 32 36 40
 5 10 15 20 25 30 35 40 45 50
 6 12 18 24 30 36 42 48 54 60
 7 14 21 28 35 42 49 56 63 70
 8 16 24 32 40 48 56 64 72 80
 9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

## Liste con valutazione lenta

È possibile creare un semplice elenco ponderato usando tipi e [chiusure](#) mutevoli. Un elenco ponderato è un elenco i cui elementi non vengono valutati al momento della sua costruzione, ma piuttosto quando vi si accede. I vantaggi delle liste valutate pigramente includono la possibilità di essere infiniti.

```
import Base: getindex
type Lazy
    thunk
    value
    Lazy(thunk) = new(thunk)
end

evaluate!(lazy::Lazy) = (lazy.value = lazy.thunk(); lazy.value)
getindex(lazy::Lazy) = isdefined(lazy, :value) ? lazy.value : evaluate!(lazy)

import Base: first, tail, start, next, done, iteratorsize, HasLength, SizeUnknown
abstract List
immutable Cons <: List
    head
    tail::Lazy
end
immutable Nil <: List end

macro cons(x, y)
    quote
        Cons($(esc(x)), Lazy(() -> $(esc(y))))
    end
end

first(xs::Cons) = xs.head
```

```
tail(xs::Cons) = xs.tail[]
start(xs::Cons) = xs
next(::Cons, xs) = first(xs), tail(xs)
done(::List, ::Cons) = false
done(::List, ::Nil) = true
iteratorSize(::Nil) = HasLength()
iteratorSize(::Cons) = SizeUnknown()
```

Che funziona davvero come farebbe in una lingua come [Haskell](#) , dove tutte le liste sono ponderate:

```
julia> xs = @cons(1, ys)
Cons{1, Lazy{false, #3, #undef}}

julia> ys = @cons(2, xs)
Cons{2, Lazy{false, #5, #undef}}

julia> [take(xs, 5)...]
5-element Array{Int64,1}:
 1
 2
 1
 2
 1
```

In pratica, è meglio usare il pacchetto [Lazy.jl](#). Tuttavia, l'implementazione dell'elenco pigro di cui sopra mette le luci in dettagli importanti su come costruire il proprio tipo iterabile.

Leggi iterabili online: <https://riptutorial.com/it/julia-lang/topic/5466/iterabili>

---

# Capitolo 19: JSON

## Sintassi

- usando JSON
- `JSON.parse (str)`
- `JSON.json (obj)`
- `JSON.print (io, obj, indent)`

## Osservazioni

Poiché né Julia `Dict` né gli oggetti JSON sono intrinsecamente ordinati, è meglio non fare affidamento sull'ordine delle coppie chiave-valore in un oggetto JSON.

## Examples

### Installazione di JSON.jl

JSON è un popolare formato di interscambio dati. La più famosa libreria JSON per Julia è [JSON.jl](#). Per installare questo pacchetto, utilizzare il gestore pacchetti:

```
julia> Pkg.add("JSON")
```

Il prossimo passo è verificare se il pacchetto funziona sulla tua macchina:

```
julia> Pkg.test("JSON")
```

Se tutti i test sono passati, la libreria è pronta per l'uso.

### Parsing JSON

JSON che è stato codificato come stringa può essere facilmente analizzato in un tipo standard di Julia:

```
julia> using JSON

julia> JSON.parse("""{
    "this": ["is", "json"],
    "numbers": [85, 16, 12.0],
    "and": [true, false, null]
}""")
Dict{String,Any} with 3 entries:
  "this" => Any{String}["is", "json"]
  "numbers" => Any{Number}[85, 16, 12.0]
  "and" => Any{Union{Bool, Number}}[true, false, nothing]
```

Ci sono alcune proprietà immediate di JSON.jl di nota:

- I tipi JSON si associano a tipi sensibili in Julia: l'oggetto diventa `Dict`, l'array diventa `Vector`, il numero diventa `Int64` o `Float64`, booleano diventa `Bool` e null diventa `nothing::Void`.
- JSON è un formato contenitore non tipizzato: i vettori di Julia restituiti sono di tipo `Vector{Any}` e i dizionari restituiti sono di tipo `Dict{String, Any}`.
- Lo standard JSON non distingue tra numeri interi e numeri decimali, ma JSON.jl lo fa. Un numero senza un punto decimale o una notazione scientifica viene analizzato in `Int64`, mentre un numero con un punto decimale viene analizzato in `Float64`. Questo corrisponde strettamente al comportamento dei parser JSON in molte altre lingue.

## Serializzazione JSON

La funzione `JSON.json` serializza un oggetto Julia in una `String` Julia contenente JSON:

```
julia> using JSON

julia> JSON.json(Dict{:a => :b, :c => [1, 2, 3.0], :d => nothing})
"{\"c\": [1.0, 2.0, 3.0], \"a\": \"b\", \"d\": null}"

julia> println(ans)
{"c": [1.0, 2.0, 3.0], "a": "b", "d": null}
```

Se una stringa non è desiderata, JSON può essere stampato direttamente su un flusso IO:

```
julia> JSON.print(STDOUT, [1, 2, true, false, "x"])
[1,2,true,false,"x"]
```

Si noti che `STDOUT` è l'impostazione predefinita e può essere omessa nella chiamata sopra.

La stampa più bella può essere ottenuta passando il parametro del `indent` facoltativo:

```
julia> JSON.print(STDOUT, Dict{:a => :b, :c => :d}, 4)
{
    "c": "d",
    "a": "b"
}
```

Esiste una serializzazione sana di mente per tipi di Julia complessi:

```
julia> immutable Point3D
           x::Float64
           y::Float64
           z::Float64
       end

julia> JSON.print(Point3D(1.0, 2.0, 3.0), 4)
{
    "y": 2.0,
    "z": 3.0,
    "x": 1.0
}
```

Leggi JSON online: <https://riptutorial.com/it/julia-lang/topic/5468/json>



# Capitolo 20: Le tuple

## Sintassi

- `un,`
- `a, b`
- `a, b = xs`
- `()`
- `(un,)`
- `(a, b)`
- `(a, b ...)`
- `Tuple {T, U, V}`
- `NTuple {N, T}`
- `Tuple {T, U, Vararg {V}}`

## Osservazioni

Le tuple hanno prestazioni di runtime molto migliori degli [array](#) per due motivi: i loro tipi sono più precisi e la loro immutabilità consente loro di essere allocati nello stack anziché nell'heap.

Tuttavia, questa digitazione più precisa viene fornita con un overhead in più tempo di compilazione e maggiori difficoltà nel raggiungere la [stabilità del tipo](#) .

## Examples

### Introduzione a Tuples

`Tuple` sono collezioni ordinate immutabili di oggetti distinti arbitrari, dello stesso tipo o di [tipi](#) diversi. Tipicamente, le tuple sono costruite usando la sintassi `(x, y)` .

```
julia> tup = (1, 1.0, "Hello, World!")  
(1,1.0,"Hello, World!")
```

I singoli oggetti di una tuple possono essere recuperati usando la sintassi dell'indicizzazione:

```
julia> tup[1]  
1  
  
julia> tup[2]  
1.0  
  
julia> tup[3]  
"Hello, World!"
```

Implementano l' [interfaccia iterabile](#) e possono quindi essere iterati utilizzando `loop for` :

```
julia> for item in tup
```

```
        println(item)
    end
1
1.0
Hello, World!
```

Le tuple supportano anche una varietà di funzioni di collezioni generiche, come il `reverse` o la `length`:

```
julia> reverse(tup)
("Hello, World!", 1.0, 1)

julia> length(tup)
3
```

Inoltre, le tuple supportano una varietà di operazioni di raccolta di [ordine superiore](#), incluse `any`, `all`, `map` o `broadcast`:

```
julia> map(typeof, tup)
(Int64, Float64, String)

julia> all(x -> x < 2, (1, 2, 3))
false

julia> all(x -> x < 4, (1, 2, 3))
true

julia> any(x -> x < 2, (1, 2, 3))
true
```

La tupla vuota può essere costruita usando `()`:

```
julia> ()
()

julia> isempty(ans)
true
```

Tuttavia, per costruire una tupla di un elemento, è necessaria una virgola finale. Questo perché le parentesi `( e )` sarebbero altrimenti trattate come operazioni di raggruppamento insieme invece di costruire una tupla.

```
julia> (1)
1

julia> (1,)
(1,)
```

Per coerenza, una virgola finale è anche consentita per le tuple con più di un elemento.

```
julia> (1, 2, 3,)
(1, 2, 3)
```

## Tipi di tupla

Il `typeof` una tupla è un sottotipo di `Tuple` :

```
julia> typeof((1, 2, 3))
Tuple{Int64,Int64,Int64}

julia> typeof((1.0, :x, (1, 2)))
Tuple{Float64,Symbol,Tuple{Int64,Int64}}
```

A differenza di altri tipi di dati, i tipi di `Tuple` sono **covarianti** . Altri tipi di dati in Julia sono generalmente invarianti. Così,

```
julia> Tuple{Int, Int} <: Tuple{Number, Number}
true

julia> Vector{Int} <: Vector{Number}
false
```

Questo è il caso perché ovunque sia accettata una `Tuple{Number, Number}` , così anche una `Tuple{Int, Int}` , poiché ha anche due elementi, entrambi sono numeri. Questo non è il caso di un `Vector{Int}` contro un `Vector{Number}` , in quanto una funzione che accetta un `Vector{Number}` può desiderare di memorizzare un punto mobile (es. `1.0` ) o un numero complesso (ad esempio `1+3im` ) in tale un vettore

La covarianza dei tipi di tupla significa che `Tuple{Number}` (a differenza di `Vector{Number}` ) è in realtà un tipo astratto:

```
julia> isleafftype(Tuple{Number})
false

julia> isleafftype(Vector{Number})
true
```

I sottotipi concreti di `Tuple{Number}` includono `Tuple{Int}` , `Tuple{Float64}` , `Tuple{Rational{BigInt}}` e così via.

`Tuple` tipi di `Vararg` possono contenere un `Vararg` terminazione come ultimo parametro per indicare un numero indefinito di oggetti. Ad esempio, `Tuple{Vararg{Int}}` è il tipo di tutte le tuple che contengono un numero qualsiasi di `Int` s, possibilmente zero:

```
julia> isa(), Tuple{Vararg{Int}})
true

julia> isa((1,), Tuple{Vararg{Int}})
true

julia> isa((1,2,3,4,5), Tuple{Vararg{Int}})
true

julia> isa((1.0,), Tuple{Vararg{Int}})
false
```

mentre `Tuple{String, Vararg{Int}}` accetta tuple costituite da una [stringa](#) , seguite da qualsiasi numero (possibilmente zero) di `Int` s.

```
julia> isa(("x", 1, 2), Tuple{String, Vararg{Int}})
true

julia> isa((1, 2), Tuple{String, Vararg{Int}})
false
```

Combinato con co-varianza, ciò significa che `Tuple{Vararg{Any}}` descrive qualsiasi tupla. In effetti, `Tuple{Vararg{Any}}` è solo un altro modo di dire `Tuple` :

```
julia> Tuple{Vararg{Any}} == Tuple
true
```

`Vararg` accetta un secondo parametro di tipo numerico che indica quante volte dovrebbe verificarsi esattamente il suo primo parametro di tipo. (Per default, se non specificato, questo secondo parametro tipo è un `typevar` che può assumere qualsiasi valore, motivo per cui un numero qualsiasi di `Int` s sono accettati nella `Vararg` s sopra.) `Tuple` tipi terminanti in un determinato `Vararg` verrà automaticamente esteso al numero richiesto di elementi:

```
julia> Tuple{String, Vararg{Int, 3}}
Tuple{String, Int64, Int64, Int64}
```

Esiste una notazione per tuple omogenee con un `Vararg` specificato: `NTuple{N, T}` . In questa notazione, `N` indica il numero di elementi nella tupla e `T` indica il tipo accettato. Per esempio,

```
julia> NTuple{3, Int}
Tuple{Int64, Int64, Int64}

julia> NTuple{10, Int}
NTuple{10, Int64}

julia> ans.types
svec{Int64, Int64, Int64, Int64, Int64, Int64, Int64, Int64, Int64, Int64}
```

Nota che `NTuple` s oltre una certa dimensione viene mostrato semplicemente come `NTuple{N, T}` , invece del modulo `Tuple` espanso, ma sono sempre dello stesso tipo:

```
julia> Tuple{Int, Int, Int, Int, Int, Int, Int, Int, Int, Int}
NTuple{10, Int64}
```

## Dispacciamento di tipi di tuple

Poiché gli elenchi dei parametri della funzione di Julia sono essi stessi delle tuple, l' [invio di](#) vari tipi di tuple è spesso più semplice attraverso i parametri del metodo stessi, spesso con un uso liberale per l'operatore "splatting" ... Ad esempio, considera l'implementazione del `reverse` per le tuple, da `Base` :

```
revargs() = ()
revargs(x, r...) = (revargs(r...)..., x)

reverse(t::Tuple) = revargs(t...)
```

L'implementazione dei metodi sulle tuple in questo modo preserva la [stabilità del tipo](#), che è fondamentale per le prestazioni. Possiamo vedere che non c'è nessun overhead a questo approccio usando la macro `@code_warntype`:

```
julia> @code_warntype reverse((1, 2, 3))
Variables:
  #self#::Base.#reverse
  t::Tuple{Int64,Int64,Int64}

Body:
  begin
    SSAValue(1) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},2)::Int64
    SSAValue(2) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},3)::Int64
    return
    (Core.tuple)(SSAValue(2),SSAValue(1),(Core.getfield)(t::Tuple{Int64,Int64,Int64},1)::Int64)::Tuple{Int64,Int64,Int64}
  end::Tuple{Int64,Int64,Int64}
```

Anche se un po' difficile da leggere, il codice qui sta semplicemente creando una nuova tupla con valori 3°, 2° e 1° elementi della tupla originale, rispettivamente. Su molte macchine, questo compila verso il codice LLVM estremamente efficiente, che consiste in carichi e negozi.

```
julia> @code_llvm reverse((1, 2, 3))

define void @julia_reverse_71456([3 x i64]* noalias sret, [3 x i64]*) #0 {
top:
  %2 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 1
  %3 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 2
  %4 = load i64, i64* %3, align 1
  %5 = load i64, i64* %2, align 1
  %6 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 0
  %7 = load i64, i64* %6, align 1
  %.sroa.0.0..sroa_idx = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 0
  store i64 %4, i64* %.sroa.0.0..sroa_idx, align 8
  %.sroa.2.0..sroa_idx1 = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 1
  store i64 %5, i64* %.sroa.2.0..sroa_idx1, align 8
  %.sroa.3.0..sroa_idx2 = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 2
  store i64 %7, i64* %.sroa.3.0..sroa_idx2, align 8
  ret void
}
```

## Più valori di ritorno

Le tuple sono spesso utilizzate per valori di ritorno multipli. Gran parte della libreria standard, incluse due delle funzioni [dell'interfaccia iterabile](#) (`next` e `done`), restituisce tuple contenenti due valori correlati ma distinti.

Le parentesi attorno alle tuple possono essere omesse in determinate situazioni, rendendo più facili da implementare più valori di ritorno. Ad esempio, possiamo creare una funzione per

restituire sia radici quadrate positive che negative di un numero reale:

```
julia> pmsqrt(x::Real) = sqrt(x), -sqrt(x)
pmsqrt (generic function with 1 method)

julia> pmsqrt(4)
(2.0,-2.0)
```

L'assegnazione della distruzione può essere utilizzata per decomprimere i valori di ritorno multipli. Per memorizzare le radici quadrate nelle variabili `a` e `b`, è sufficiente scrivere:

```
julia> a, b = pmsqrt(9.0)
(3.0,-3.0)

julia> a
3.0

julia> b
-3.0
```

Un altro esempio di ciò sono le funzioni `divrem` e `fldmod`, che eseguono contemporaneamente una [divisione intera \(troncata o pavimentata\)](#) e un'operazione resto:

```
julia> q, r = divrem(10, 3)
(3,1)

julia> q
3

julia> r
1
```

Leggi Le tuple online: <https://riptutorial.com/it/julia-lang/topic/6675/le-tuple>

# Capitolo 21: Lettura di un DataFrame da un file

## Examples

### Lettura di un dataframe da dati separati da delimitatore

Si consiglia di leggere un `DataFrame` da un file CSV (valori separati da virgola) o forse anche da un TSV o WSV (schede e file separati da spazi bianchi). Se il tuo file ha l'estensione giusta, puoi usare la funzione `readtable` per leggere nel dataframe:

```
readtable("dataset.CSV")
```

Ma cosa succede se il tuo file non ha l'estensione giusta? È possibile specificare il delimitatore utilizzato dal file (virgola, tabulazione, spazio bianco ecc.) Come argomento della parola chiave per la funzione di `readtable` :

```
readtable("dataset.txt", separator=',')
```

### Gestire commenti di commento diversi

I set di dati contengono spesso commenti che spiegano il formato dei dati o contengono i termini di licenza e di utilizzo. Di solito vuoi ignorare queste righe quando leggi in `DataFrame` .

La funzione di `readtable` presuppone che le righe di commento inizino con il carattere '#'. Tuttavia, il tuo file potrebbe utilizzare contrassegni di commento come `%` o `//` . Per assicurarsi che il `readtable` grado di `readtable` correttamente, è possibile specificare il contrassegno del commento come argomento della parola chiave:

```
readtable("dataset.csv", allowcomments=true, commentmark='%')
```

Leggi Lettura di un DataFrame da un file online: <https://riptutorial.com/it/julia-lang/topic/7340/lettura-di-un-dataframe-da-un-file>

# Capitolo 22: Macro di stringa

## Sintassi

- macro "stringa" # breve, forma macro di stringa
- @macro\_str "stringa" # lungo, normale modulo macro
- macro`command`

## Osservazioni

Le macro di stringa non sono altrettanto potenti delle semplici stringhe vecchie: poiché l'interpolazione deve essere implementata nella logica della macro, le macro di stringa non sono in grado di contenere valori letterali di stringa dello stesso delimitatore per l'interpolazione.

Ad esempio, anche se

```
julia> "$("x") "  
"x"
```

funziona, la forma del testo macro della stringa

```
julia> doc"$("x") "  
ERROR: KeyError: key :x not found
```

viene analizzato in modo errato. Questo può essere in qualche modo mitigato usando le virgolette triple come delimitatore di stringhe esterne;

```
julia> doc"""$("x") """  
"x"
```

funziona davvero correttamente.

## Examples

### Utilizzo di macro di stringa

Le macro di stringa sono zucchero sintattico per alcune invocazioni di macro. Il parser espande la sintassi come

```
mymacro"my string"
```

in

```
@mymacro_str "my string"
```



@mymacro\_str macro @mymacro\_str . Base Julia viene fornito con diverse macro di stringa, come ad esempio:

Questa macro di stringhe costruisce **array di byte** invece di **stringhe** . Il contenuto della stringa, codificato come UTF-8, verrà utilizzato come matrice di byte. Questo può essere utile per l'interfacciamento con API di basso livello, molte delle quali funzionano con array di byte invece di stringhe.

```
@big_str
```

Questa macro esiste perché `big(0.1)` non si comporta come ci si aspetterebbe inizialmente: lo `0.1` è un'approssimazione `Float64` di `true 0.1 ( 1//10 )`, e promuoverlo a `BigFloat` manterrà l'errore di approssimazione di `Float64`. L'uso della macro analizzerà `0.1` direttamente su un `BigFloat`, riducendo l'errore di approssimazione.

Questa macro di stringhe costruisce oggetti `Base.Markdown.MD` , che vengono utilizzati nel sistema di documentazione interno per fornire documentazione di testo completo per qualsiasi ambiente. Questi oggetti MD rendono bene in un terminale:

```
julia> doc"""
    This is a markdown documentation string.

    ## Heading

    Math ``1 + 2`` and `code` are supported.
    """
This is a markdown documentation string.

    Heading
    =====

    Math 1 + 2 and code are supported.
```

e anche in un browser:

```
In [2]: doc"""
    This is a markdown documentation string.

    ## Heading

    Math ``1 + 2`` and `code` are supported.
    """
```

Out[2]: This is a markdown documentation string.

## Heading

Math 1 + 2 and code are supported.

@html\_str

Questa macro di stringhe costruisce letterali stringa HTML, che rendono bene in un browser:

```
In [1]: html"""
    <p><abbr title="Hypertext Markup Language">HTML</abbr> text.</p>
    """
```

Out[1]: HTML text.

@ip\_str

Questa macro di stringhe costruisce letterali di indirizzo IP. Funziona con IPv4 e IPv6:

```
julia> ip"127.0.0.1"
ip"127.0.0.1"
```

```
julia> ip "::"
ip "::"
```

@r\_str

Questa macro di stringhe costruisce i [valori letterali di Regex](#) .

`@s_str`

Questa macro di stringhe costruisce letterali di `SubstitutionString` , che `Regex` con i letterali di `Regex` per consentire una sostituzione testuale più avanzata.

`@text_str`

Questa macro di stringa è simile nello spirito a `@doc_str` e `@html_str` , ma non ha alcuna caratteristica di formattazione:

```
In [3]: text"""
This is some plain text.
"""

Out[3]: This is some plain text.
```

`@v_str`

Questa macro di stringhe costruisce valori letterali `VersionNumber` . Vedi i [numeri di versione](#) per una descrizione di cosa sono e come usarli.

`@MIME_str`

Questa macro di stringhe costruisce i tipi singleton di tipi MIME. Ad esempio, `MIME"text/plain"` è il tipo di `MIME("text/plain")` .

## Simboli che non sono identificativi legali

I valori letterali del simbolo di Julia devono essere identificativi legali. Questo funziona:

```
julia> :cat
:cat
```

Ma questo non:

```
julia> :2cat
ERROR: MethodError: no method matching *(::Int64, ::Base.#cat)
Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...) at operators.jl:288

*{T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}}(::T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}) at int.jl:33
*(::Real, ::Complex{Bool}) at complex.jl:180
...
```

Ciò che sembra un letterale di simboli qui viene in realtà analizzato come una moltiplicazione implicita di `:2` (che è solo `2` ) e la funzione `cat` , che ovviamente non funziona.

Possiamo usare

```
julia> Symbol("2cat")
Symbol("2cat")
```

per aggirare il problema.

Una macro di stringhe potrebbe aiutare a renderlo più conciso. Se definiamo la macro `@sym_str`:

```
macro sym_str(str)
    Meta.quot(Symbol(str))
end
```

allora possiamo semplicemente fare

```
julia> sym"2cat"
Symbol("2cat")
```

per creare simboli che non sono identificativi di Julia validi.

Naturalmente, queste tecniche possono anche creare simboli che *sono* identificativi di Julia validi. Per esempio,

```
julia> sym"test"
:test
```

## Implementazione dell'interpolazione in una macro di stringhe

Le macro di stringa non vengono fornite con le funzioni di [interpolazione](#) incorporate. Tuttavia, è possibile implementare manualmente questa funzionalità. Si noti che non è possibile incorporare senza eseguire l'escape di stringhe letterali che hanno lo stesso delimitatore della macro di stringa circostante; vale a dire, sebbene `""" $( "x" ) """` sia possibile, `" $( "x" ) "` non lo è. Invece, questo deve essere salvato come `" $( \ "x\ " ) "`. Vedere la sezione [commenti](#) per maggiori dettagli su questa limitazione.

Esistono due approcci per implementare l'interpolazione manualmente: implementare l'analisi manualmente o fare in modo che Julia esegua l'analisi. Il primo approccio è più flessibile, ma il secondo approccio è più semplice.

## Analisi manuale

```
macro interp_str(s)
    components = []
    buf = IOBuffer(s)
    while !eof(buf)
        push!(components, rstrip(readuntil(buf, '$'), '$'))
        if !eof(buf)
            push!(components, parse(buf; greedy=false))
        end
    end
    quote
        string($(map(esc, components)...))
    end
end
```

```
end
end
```

## Julia analizza

```
macro e_str(s)
    esc(parse("\$(escape_string(s))\""))
end
```

Questo metodo sfugge alla stringa (ma nota che `escape_string` *non* sfugge ai `$` sign) e lo restituisce al parser di Julia per analizzarlo. L'escape della stringa è necessario per garantire che `"` e `\` non influenzino l'analisi della stringa. L'espressione risultante è un'espressione `:string`, che può essere esaminata e scomposta per scopi macro.

## Macro di comando

0.6.0-dev

In Julia v0.6 e successive, le macro di comando sono supportate in aggiunta alle normali macro di stringa. Un richiamo di macro di comando come

```
mymacro`xyz`
```

viene analizzato come la chiamata macro

```
@mymacro_cmd "xyz"
```

Si noti che questo è simile alle macro di stringa, tranne con `_cmd` invece di `_str`.

Normalmente usiamo macro di comandi per il codice, che in molte lingue spesso contiene `"` ma raramente contiene ```. Ad esempio, è abbastanza semplice reimplementare una versione semplice di [quasiquote](#) usando le macro di comando:

```
macro julia_cmd(s)
    esc(Meta.quot(parse(s)))
end
```

Possiamo usare questa macro sia in linea:

```
julia> julia`1+1`
:(1 + 1)

julia> julia`hypot2(x,y)=x^2+y^2`
:(hypot2(x,y) = begin # none, line 1:
    x ^ 2 + y ^ 2
end)
```

o multilinea:

```
julia> julia```\n    function hello()\n        println("Hello, World!")\n    end\n    ```\n:(function hello() # none, line 2:\n    println("Hello, World!")\nend)
```

L'interpolazione usando `$` è supportata:

```
julia> x = 2\n2\n\njulia> julia`1 + $x`\n:(1 + 2)
```

ma la versione qui fornita consente solo un'espressione:

```
julia> julia```\n    x = 2\n    y = 3\n    ```\nERROR: ParseError("extra token after end of expression")
```

Tuttavia, estenderlo per gestire più espressioni non è difficile.

Leggi Macro di stringa online: <https://riptutorial.com/it/julia-lang/topic/5817/macro-di-stringa>

# Capitolo 23: mentre cicli

## Sintassi

- mentre cond; corpo; fine
- rompere
- Continua

## Osservazioni

Il ciclo `while` non ha un valore; sebbene possa essere usato nella posizione dell'espressione, il suo tipo è `Void` e il valore ottenuto non sarà `nothing`.

## Examples

### Sequenza di Collatz

Il ciclo `while` esegue il suo corpo finché dura la condizione. Ad esempio, il codice seguente calcola e stampa la [sequenza Collatz](#) da un numero dato:

```
function collatz(n)
    while n ≠ 1
        println(n)
        n = iseven(n) ? n ÷ 2 : 3n + 1
    end
    println("1... and 4, 2, 1, 4, 2, 1 and so on")
end
```

Uso:

```
julia> collatz(10)
10
5
16
8
4
2
1... and 4, 2, 1, 4, 2, 1 and so on
```

È possibile scrivere qualsiasi ciclo in modo ricorsivo, e per cicli complessi `while` volte la variante ricorsiva è più chiara. Tuttavia, in Julia, i loop presentano alcuni vantaggi distinti rispetto alla ricorsione:

- Julia non garantisce l'eliminazione delle chiamate tail, pertanto la ricorsione utilizza memoria aggiuntiva e può causare errori di overflow dello stack.
- Inoltre, per lo stesso motivo, un loop può avere un sovraccarico e correre più velocemente.

## Esegui una volta prima di testare la condizione

A volte, si vuole eseguire un codice di inizializzazione una volta prima di testare una condizione. In certi altri linguaggi, questo tipo di loop ha una sintassi speciale `do - while`. Tuttavia, questa sintassi può essere sostituita con una normale istruzione `while` loop e `break`, quindi Julia non ha una sintassi specialistica `do - while`. Invece, si scrive:

```
local name

# continue asking for input until satisfied
while true
    # read user input
    println("Type your name, without lowercase letters:")
    name = readline()

    # if there are no lowercase letters, we have our result!
    !any(islower, name) && break
end
```

Si noti che in alcune situazioni, tali cicli potrebbero essere più chiari con la ricorsione:

```
function getname()
    println("Type your name, without lowercase letters:")
    name = readline()
    if any(islower, name)
        getname() # this name is unacceptable; try again
    else
        name      # this name is good, return it
    end
end
```

## Ricerca per ampiezza

### 0.5.0

(Sebbene questo esempio sia scritto usando la sintassi introdotta nella versione v0.5, può funzionare anche con alcune modifiche su versioni precedenti).

Questa implementazione della [ricerca in ampiezza](#) (BFS) su un grafico rappresentato con elenchi di adiacenza utilizza i cicli `while` e l'istruzione `return`. Il compito che risolveremo è il seguente: abbiamo una sequenza di persone e una sequenza di amicizie (le amicizie sono reciproche). Vogliamo determinare il grado di connessione tra due persone. Cioè, se due persone sono amici, restituiamo `1`; se uno è amico di un amico dell'altro, restituiamo `2`, e così via.

Innanzitutto, supponiamo di avere già un elenco di adiacenze: un `Dict` mappa `T` to `Array{T, 1}`, dove le chiavi sono persone e i valori sono tutti gli amici di quella persona. Qui possiamo rappresentare le persone con qualsiasi tipo `T` scegliamo; in questo esempio, useremo `Symbol`. Nell'algoritmo BFS, manteniamo una coda di persone da "visitare" e contrassegniamo la loro distanza dal nodo di origine.

```
function degree(adjlist, source, dest)
```



```

distances = Dict(source => 0)
queue = [source]

# until the queue is empty, get elements and inspect their neighbours
while !isempty(queue)
    # shift the first element off the queue
    current = shift!(queue)

    # base case: if this is the destination, just return the distance
    if current == dest
        return distances[dest]
    end

    # go through all the neighbours
    for neighbour in adjlist[current]
        # if their distance is not already known...
        if !haskey(distances, neighbour)
            # then set the distance
            distances[neighbour] = distances[current] + 1

            # and put into queue for later inspection
            push!(queue, neighbour)
        end
    end
end

# we could not find a valid path
error("$source and $dest are not connected.")
end

```

Ora, scriveremo una funzione per costruire una lista di adiacenze data una sequenza di persone e una sequenza di tuple (person, person) :

```

function makeadjlist(people, friendships)
    # dictionary comprehension (with generator expression)
    result = Dict{p => eltype(people)[], for p in people}

    # deconstructing for; friendship is mutual
    for (a, b) in friendships
        push!(result[a], b)
        push!(result[b], a)
    end

    result
end

```

Ora possiamo definire la funzione originale:

```

degree(people, friendships, source, dest) =
    degree(makeadjlist(people, friendships), source, dest)

```

Ora testiamo la nostra funzione su alcuni dati.

```

const people = [:jean, :javert, :cosette, :gavroche, :éponine, :maris]
const friendships = [
    (:jean, :cosette),
    (:jean, :maris),

```

```
(:cosette, :éponine),  
(:cosette, :marius),  
(:gavroche, :éponine)  
]
```

Jean è collegato a se stesso in 0 passaggi:

```
julia> degree(people, friendships, :jean, :jean)  
0
```

Jean e Cosette sono amici, e così hanno il grado 1 :

```
julia> degree(people, friendships, :jean, :cosette)  
1
```

Jean e Gavroche sono collegati indirettamente attraverso Cosette e poi Marius, quindi la loro laurea è 3 :

```
julia> degree(people, friendships, :jean, :gavroche)  
3
```

Javert e Marius non sono collegati attraverso alcuna catena, quindi viene generato un errore:

```
julia> degree(people, friendships, :javert, :marius)  
ERROR: javert and marius are not connected.  
in degree(::Dict{Symbol,Array{Symbol,1}}, ::Symbol, ::Symbol) at ./REPL[28]:27  
in degree(::Array{Symbol,1}, ::Array{Tuple{Symbol,Symbol},1}, ::Symbol, ::Symbol) at  
./REPL[30]:1
```

Leggi mentre cicli online: <https://riptutorial.com/it/julia-lang/topic/5565/mentre-cicli>

---

# Capitolo 24: metaprogrammazione

## Sintassi

- nome macro (ex) ... fine
- citazione ... fine
- : (...)
- \$ x
- Meta.quot (x)
- QuoteNode (x)
- esc (x)

## Osservazioni

Le funzioni di metaprogrammazione di Julia sono fortemente ispirate a quelle delle lingue di tipo Lisp e sembreranno familiari a coloro che hanno un background Lisp. Metaprogramming è molto potente. Se usato correttamente, può portare a un codice più conciso e leggibile.

La `quote ... end` è una sintassi quasi quote. Invece delle espressioni all'interno di essere valutate, vengono semplicemente analizzate. Il valore della `quote ... end` espressione `quote ... end` è l'Abstract Syntax Tree (AST) risultante.

La sintassi `: (...)` è simile alla `quote ... end` syntax, ma è più leggera. Questa sintassi è più concisa della `quote ... end`.

All'interno di una quasiquote, l'operatore `$` è speciale e *interpola* il suo argomento nell'AST. L'argomento dovrebbe essere un'espressione che è giunta direttamente nell'AST.

La funzione `Meta.quot (x)` cita il suo argomento. Questo è spesso utile in combinazione con l'uso di `$` per l'interpolazione, in quanto consente letteralmente l'unione di espressioni e simboli nell'AST.

## Examples

### Reimplementare la macro `@show`

In Julia, la macro `@show` è spesso utile per scopi di debug. Visualizza sia l'espressione da valutare che il suo risultato, restituendo infine il valore del risultato:

```
julia> @show 1 + 1
1 + 1 = 2
2
```

È semplice creare la nostra versione di `@show` :

```
julia> macro myshow(expression)
```

```

        quote
            value = $expression
            println($(Meta.quot(expression)), " = ", value)
            value
        end
    end
end

```

Per usare la nuova versione, usa semplicemente la macro `@myshow` :

```

julia> x = @myshow 1 + 1
1 + 1 = 2
2

julia> x
2

```

## Fino al ciclo

Siamo tutti abituati alla sintassi `while` , che esegue il suo corpo mentre la condizione viene valutata come `true` . Cosa succede se vogliamo implementare un ciclo `until` , che esegue un ciclo finché la condizione non viene valutata su `true` ?

In Julia, possiamo farlo creando una macro `@until` , che si ferma ad eseguire il suo corpo quando la condizione è soddisfatta:

```

macro until(condition, expression)
    quote
        while !($condition)
            $expression
        end
    end |> esc
end

```

Qui abbiamo usato la funzione chaining syntax `|>` , che equivale a chiamare la funzione `esc` sull'intero blocco di `quote` . La funzione `esc` impedisce all'igiene della macro di applicarsi al contenuto della macro; senza di essa, le variabili con scope nella macro verranno rinominate per evitare collisioni con variabili esterne. Vedi la documentazione di Julia [sull'igiene macro](#) per maggiori dettagli.

Puoi usare più di un'espressione in questo ciclo, semplicemente mettendo tutto in un `begin ... end` blocco:

```

julia> i = 0;

julia> @until i == 10 begin
            println(i)
            i += 1
        end

0
1
2
3
4

```

```
5
6
7
8
9

julia> i
10
```

## QuoteNode, Meta.quot ed Expr (: quota)

Ci sono tre modi per citare qualcosa usando una funzione di Julia:

```
julia> QuoteNode(:x)
: (:x)

julia> Meta.quot(:x)
: (:x)

julia> Expr(:quote, :x)
: (:x)
```

Cosa significa "quoting" e a cosa serve? La citazione ci consente di proteggere le espressioni dall'interpretazione come forme speciali di Julia. Un caso di uso comune è quando generiamo [espressioni](#) che dovrebbero contenere elementi che valutano i simboli. (Ad esempio, [questa macro](#) deve restituire un'espressione che valuta un simbolo.) Non funziona semplicemente per restituire il simbolo:

```
julia> macro mysym(); :x; end
@mysym (macro with 1 method)

julia> @mysym
ERROR: UndefVarError: x not defined

julia> macroexpand(:(@mysym))
:x
```

Cosa sta succedendo qui? `@mysym` espande in `:x`, che come espressione viene interpretata come variabile `x`. Ma nulla è stato ancora assegnato a `x`, quindi otteniamo un errore `x not defined`.

Per aggirare questo, dobbiamo citare il risultato della nostra macro:

```
julia> macro mysym2(); Meta.quot(:x); end
@mysym2 (macro with 1 method)

julia> @mysym2
:x

julia> macroexpand(:(@mysym2))
: (:x)
```

Qui, abbiamo usato la funzione `Meta.quot` per trasformare il nostro simbolo in un simbolo quotato, che è il risultato che vogliamo.

Qual è la differenza tra `Meta.quot` e `QuoteNode` e quale dovrei usare? In quasi tutti i casi, la differenza non ha molta importanza. A volte è forse un po' più sicuro usare `QuoteNode` invece di `Meta.quot`. Esplorare la differenza è informativo su come funzionano le espressioni e le macro di Julia.

## La differenza tra `Meta.quot` e `QuoteNode`, spiegata

Ecco una regola generale:

- Se hai bisogno o vuoi supportare l'interpolazione, usa `Meta.quot` ;
- Se non puoi o non vuoi consentire l'interpolazione, usa `QuoteNode` .

In breve, la differenza è che `Meta.quot` consente l'interpolazione all'interno della cosa quotata, mentre `QuoteNode` protegge il suo argomento da qualsiasi interpolazione. Per capire l'interpolazione, è importante menzionare l'espressione `$` . C'è una specie di espressione in Julia chiamata `$` espressione. Queste espressioni consentono di scappare. Ad esempio, considera la seguente espressione:

```
julia> ex = :( x = 1; :($x + $x) )
quote
  x = 1
  $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))
end
```

Quando valutata, questa espressione valuterà `1` e assegnerà a `x`, quindi costruirà un'espressione della forma `_ + _` dove `_` sarà sostituito dal valore di `x`. Quindi, il risultato di questo dovrebbe essere l' *espressione* `1 + 1` (che non è ancora stata valutata, e quindi distinta dal *valore* `2`). In effetti, questo è il caso:

```
julia> eval(ex)
:(1 + 1)
```

Diciamo ora che stiamo scrivendo una macro per costruire questo tipo di espressioni. La nostra macro prenderà una discussione, che sostituirà la `1` nella `ex` sopra. Questo argomento può essere qualsiasi espressione, ovviamente. Ecco qualcosa che non è esattamente ciò che vogliamo:

```
julia> macro makeex(arg)
  quote
    :( x = $(esc($arg)); :($x + $x) )
  end
end
@makeex (macro with 1 method)

julia> @makeex 1
quote
  x = $(Expr(:escape, 1))
  $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))
end

julia> @makeex 1 + 1
quote
```

```

x = $(Expr(:escape, 2))
$(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))))
end

```

Il secondo caso non è corretto, perché dovremmo mantenere `1 + 1` valutato. Lo `Meta.quot` citando l'argomento con `Meta.quot` :

```

julia> macro makeex2(arg)
    quote
        :( x = $$ (Meta.quot (arg)); :($x + $x) )
    end
end

@makeex2 (macro with 1 method)

julia> @makeex2 1 + 1
quote
    x = 1 + 1
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))))
end

```

L'igiene macro non si applica al contenuto di una citazione, quindi in questo caso non è necessario scappare (e in effetti non è legale).

Come accennato in precedenza, `Meta.quot` consente l'interpolazione. Quindi proviamolo:

```

julia> @makeex2 1 + $(sin(1))
quote
    x = 1 + 0.8414709848078965
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))))
end

julia> let q = 0.5
    @makeex2 1 + $q
end
quote
    x = 1 + 0.5
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))))
end

```

Dal primo esempio, vediamo che l'interpolazione ci consente di allineare il `sin(1)` , invece di fare in modo che l'espressione sia un `sin(1)` letterale `sin(1)` . Il secondo esempio mostra che questa interpolazione viene eseguita nell'ambito della macro invocazione, non nell'ambito della macro stessa. Questo perché la nostra macro non ha effettivamente valutato alcun codice; tutto ciò che sta facendo è generare codice. La valutazione del codice (che si fa strada nell'espressione) viene eseguita quando l'espressione generata dalla macro viene effettivamente eseguita.

E se invece avessimo usato `QuoteNode` ? Come puoi immaginare, dal momento che `QuoteNode` impedisce che l'interpolazione si verifichi, ciò significa che non funzionerà.

```

julia> macro makeex3(arg)
    quote
        :( x = $$ (QuoteNode (arg)); :($x + $x) )
    end
end

```

```

@makeex3 (macro with 1 method)

julia> @makeex3 1 + $(sin(1))
quote
    x = 1 + $(Expr(:$, :(sin(1))))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))))
end

julia> let q = 0.5
    @makeex3 1 + $q
end
quote
    x = 1 + $(Expr(:$, :q))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))))
end

julia> eval(@makeex3 $(sin(1)))
ERROR: unsupported or misplaced expression $
in eval(::Module, ::Any) at ./boot.jl:234
in eval(::Any) at ./boot.jl:233

```

In questo esempio, potremmo essere d'accordo sul fatto che `Meta.quot` offre una maggiore flessibilità, in quanto consente l'interpolazione. Quindi, perché potremmo mai considerare l'utilizzo di `QuoteNode` ? In alcuni casi, potremmo non desiderare realmente l'interpolazione e in realtà vogliamo l'espressione `$` letterale. Quando sarebbe desiderabile? Consideriamo una generalizzazione di `@makeex` cui possiamo passare ulteriori argomenti per determinare cosa viene a sinistra ea destra del segno `+` :

```

julia> macro makeex4(expr, left, right)
    quote
        quote
            $$ (Meta.quot(expr))
            : ($$(Meta.quot(left)) + $$$(Meta.quot(right)))
        end
    end
end

@makeex4 (macro with 1 method)

julia> @makeex4 x=1 x x
quote # REPL[110], line 4:
    x = 1 # REPL[110], line 5:
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x))))))
end

julia> eval(ans)
:(1 + 1)

```

Una limitazione della nostra implementazione di `@makeex4` è che non possiamo usare direttamente espressioni come i lati sinistro e destro dell'espressione, perché vengono interpolate. In altre parole, le espressioni possono essere valutate per l'interpolazione, ma potremmo volere che vengano mantenute. (Dato che ci sono molti livelli di citazione e valutazione qui, chiariamo: la nostra macro genera *codice* che costruisce *un'espressione* che, una volta valutata, produce un'altra *espressione* .)

```

julia> @makeex4 x=1 x/2 x

```



```

quote # REPL[110], line 4:
  x = 1 # REPL[110], line 5:
    $(Expr(:quote, :($(Expr(:$, :(x / 2))) + $(Expr(:$, :x))))
end

julia> eval(ans)
:(0.5 + 1)

```

Dovremmo consentire all'utente di specificare quando deve avvenire l'interpolazione e quando non dovrebbe. In teoria, questa è una soluzione semplice: possiamo solo rimuovere uno dei `$` segni nella nostra applicazione e lasciare che l'utente contribuisca al proprio. Ciò significa che interpoliamo una versione citata dell'espressione inserita dall'utente (che abbiamo già citato e interpolato una volta). Ciò porta al seguente codice, che può essere un po' confuso all'inizio, a causa dei livelli multipli annidati di quoting e unquoting. Cerca di leggere e capire a cosa serve ciascuna fuga.

```

julia> macro makeex5(expr, left, right)
    quote
        quote
            $$ (Meta.quot(expr))
            : ($$ (Meta.quot ($ (Meta.quot(left)))) + $$ (Meta.quot ($ (Meta.quot(right)))))
        end
    end
end

@makeex5 (macro with 1 method)

julia> @makeex5 x=1 1/2 1/4
quote # REPL[121], line 4:
  x = 1 # REPL[121], line 5:
    $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :(1 / 2)))))) + $(Expr(:$, :($(Expr(:quote,
:(1 / 4))))))))))
end

julia> eval(ans)
:(1 / 2 + 1 / 4)

julia> @makeex5 y=1 $y $y
ERROR: UndefVarError: y not defined

```

Le cose sono iniziate bene, ma qualcosa è andato storto. Il codice generato dalla macro sta tentando di interpolare la copia di `y` nell'ambito della macro invocazione; ma non esiste *una* copia di `y` nell'ambito della macro invocazione. Il nostro errore sta consentendo l'interpolazione con il secondo e il terzo argomento nella macro. Per correggere questo errore, dobbiamo usare `QuoteNode`.

```

julia> macro makeex6(expr, left, right)
    quote
        quote
            $$ (Meta.quot(expr))
            : ($$ (Meta.quot ($ (QuoteNode(left)))) + $$ (Meta.quot ($ (QuoteNode(right)))))
        end
    end
end

@makeex6 (macro with 1 method)

```

```

julia> @makeex6 y=1 1/2 1/4
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :(1 / 2)))))) + $(Expr(:$, :($(Expr(:quote,
:(1 / 4))))))))))
end

julia> eval(ans)
:(1 / 2 + 1 / 4)

julia> @makeex6 y=1 $y $y
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :($(Expr(:$, :y))))))) + $(Expr(:$,
:($(Expr(:quote, :($(Expr(:$, :y))))))))))
end

julia> eval(ans)
:(1 + 1)

julia> @makeex6 y=1 1+$y $y
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :(1 + $(Expr(:$, :y))))))) + $(Expr(:$,
:($(Expr(:quote, :($(Expr(:$, :y))))))))))
end

julia> @makeex6 y=1 $y/2 $y
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :($(Expr(:$, :y)) / 2)))))) + $(Expr(:$,
:($(Expr(:quote, :($(Expr(:$, :y))))))))))
end

julia> eval(ans)
:(1 / 2 + 1)

```

Usando `QuoteNode`, abbiamo protetto i nostri argomenti dall'interpolazione. Poiché `QuoteNode` ha solo l'effetto di protezioni aggiuntive, non è mai dannoso usare `QuoteNode`, a meno che non si desideri l'interpolazione. Tuttavia, capire la differenza rende possibile capire dove e perché `Meta.quot` potrebbe essere una scelta migliore.

Questo lungo esercizio è con un esempio che è chiaramente troppo complesso per presentarsi in qualsiasi applicazione ragionevole. Pertanto, abbiamo giustificato la seguente regola empirica, menzionata in precedenza:

- Se hai bisogno o vuoi supportare l'interpolazione, usa `Meta.quot` ;
- Se non puoi o non vuoi consentire l'interpolazione, usa `QuoteNode` .

## Che dire di Expr (: citazione)?

`Expr(:quote, x)` è equivalente a `Meta.quot(x)` . Tuttavia, quest'ultimo è più idiomatico ed è preferito. Per il codice che utilizza in gran parte la metaprogrammazione, viene spesso utilizzata una linea `using Base.Meta` , che consente di `Meta.quot` semplicemente come `quot` .

# Bit e bob di Metaprogramming di $\pi$

### obiettivi:

- Insegnare attraverso esempi funzionali minimamente mirati / utili / non astratti (es. `@swap` o `@assert` ) che introducono concetti in contesti adatti
- Preferisci che il codice illustri / mostri i concetti piuttosto che i paragrafi di spiegazione
- Evita di collegare "lettura richiesta" ad altre pagine - interrompe la narrazione
- Presentare le cose in un ordine ragionevole che renderà l'apprendimento più facile

### risorse:

[julia-lang.org](http://julia-lang.org)

[wikibook \(@Cormullion\)](#)

[5 strati \(Leah Hanson\)](#)

[SO-Doc Quoting \(@TotalVerb\)](#)

[SO-Doc - Simboli che non sono identificativi legali \(@TotalVerb\)](#)

[SO: Cos'è un simbolo in Julia \(@StefanKarpinski\)](#)

[Discussione discussione \(@ pi- \) Metaprogrammazione](#)

La maggior parte del materiale proviene dal canale del discorso, la maggior parte proviene dalla fcard ... perfavore se ho dimenticato le attribuzioni.

## Simbolo

```
julia> mySymbol = Symbol("myName") # or 'identifier'
:myName

julia> myName = 42
42

julia> mySymbol |> eval # 'foo |> bar' puts output of 'foo' into 'bar', so 'bar(foo)'
42

julia> :( $mySymbol = 1 ) |> eval
1

julia> myName
1
```

### Passando flag in funzioni:

```
function dothing(flag)
    if flag == :thing_one
```

```

        println("did thing one")
    elseif flag == :thing_two
        println("did thing two")
    end
end
julia> dothing(:thing_one)
did thing one

julia> dothing(:thing_two)
did thing two

```

## Un esempio di hashkey:

```

number_names = Dict{Symbol, Int}()
number_names[:one] = 1
number_names[:two] = 2
number_names[:six] = 6

```

**(Avanzato) (@fcard)** `:foo` aka `:(foo)` restituisce un simbolo se `foo` è un identificatore valido, altrimenti un'espressione.

```

# NOTE: Different use of ':' is:
julia> :mySymbol = Symbol('hello world')

# (You can create a symbol with any name with Symbol("<name>"),
# which lets us create such gems as:
julia> one_plus_one = Symbol("1 + 1")
Symbol("1 + 1")

julia> eval(one_plus_one)
ERROR: UndefVarError: 1 + 1 not defined
...

julia> valid_math = :($one_plus_one = 3)
:(1 + 1 = 3)

julia> one_plus_one_plus_two = :($one_plus_one + 2)
:(1 + 1 + 2)

julia> eval(quote
    $valid_math
    @show($one_plus_one_plus_two)
end)
1 + 1 + 2 = 5
...

```

Fondamentalmente puoi trattare i simboli come stringhe leggere. Non è quello per cui sono, ma puoi farlo, quindi perché no. La base stessa di Julia lo fa, `print_with_color(:red, "abc")` stampa un `abc` di colore rosso.

## Expr (AST)

(Quasi) tutto in Julia è un'espressione, cioè un'istanza di `Expr`, che manterrà un [AST](#).

```

# when you type ...
julia> 1+1
2

# Julia is doing: eval(parse("1+1"))
# i.e. First it parses the string "1+1" into an `Expr` object ...
julia> ast = parse("1+1")
:(1 + 1)

# ... which it then evaluates:
julia> eval(ast)
2

# An Expr instance holds an AST (Abstract Syntax Tree). Let's look at it:
julia> dump(ast)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
  typ: Any

# TRY: fieldnames(typeof(ast))

julia>      :(a + b*c + 1) ==
      parse("a + b*c + 1") ==
      Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true

```

## Nesting `Expr`s:

```

julia> dump( :(1+2/3) )
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol /
        2: Int64 2
        3: Int64 3
      typ: Any
  typ: Any

# Tidier rep'n using s-expr
julia> Meta.show_sexpr( :(1+2/3) )
(:call, :+, 1, (:call, :/, 2, 3))

```

## `Expr` multilinea usando la `quote`

```

julia> blk = quote
      x=10
      x+1
    end

```

```

quote # REPL[121], line 2:
    x = 10 # REPL[121], line 3:
    x + 1
end

julia> blk == :( begin x=10; x+1 end )
true

# Note: contains debug info:
julia> Meta.show_sexpr(blk)
(:block,
 (:line, 2, Symbol("REPL[121]")),
 (:(=), :x, 10),
 (:line, 3, Symbol("REPL[121]")),
 (:call, :+, :x, 1)
)

# ... unlike:
julia> noDbg = :( x=10; x+1 )
quote
    x = 10
    x + 1
end

```

... così il `quote` è funzionalmente lo stesso ma fornisce informazioni di debug aggiuntive.

(\*) **SUGGERIMENTO** : usare `let` per mantenere `x` all'interno del blocco

`quote` **una** `quote`

`Expr(:quote, x)` viene utilizzato per rappresentare le virgolette tra virgolette.

```

Expr(:quote, :(x + y)) == :(:(x + y))

Expr(:quote, Expr(:$, :x)) == :(:(\$x))

```

`QuoteNode(x)` è simile a `Expr(:quote, x)` ma impedisce l'interpolazione.

```

eval(Expr(:quote, Expr(:$, 1))) == 1

eval(QuoteNode(Expr(:$, 1))) == Expr(:$, 1)

```

( [Disambigua i vari meccanismi di quotazione in metaprogrammazione di Julia](#) )

## \$ E : (...) sono in qualche modo inversi l'uno dall'altro?

`:(foo)` significa "non guardare il valore, guarda l'espressione" `$foo` significa "cambia l'espressione al suo valore"

`:($(foo)) == foo . $(:(foo))` è un errore. `$(...)` non è un'operazione e non fa nulla da solo, è un "interpolare questo!" firmare che utilizza la sintassi del quoting. cioè Esiste solo all'interno di una citazione.

\$foo **lo stesso di** `eval( foo )` ?

**No!** `$foo` viene scambiato per il valore in fase di compilazione `eval(foo)` significa eseguirlo in fase di runtime

`eval` si verificherà nell'intervallo globale l'interpolazione è locale

`eval(<expr>)` dovrebbe restituire lo stesso di solo `<expr>` (assumendo che `<expr>` sia un'espressione valida nello spazio globale corrente)

```
eval(: (1 + 2)) == 1 + 2

eval(: (let x=1; x + 1 end)) == let x=1; x + 1 end
```

**macro S**

Pronto? :)

```
# let's try to make this!
julia> x = 5; @show x;
x = 5
```

**Facciamo la nostra macro** `@show` :

```
macro log(x)
    :(
        println( "Expression: ", $(string(x)), " has value: ", $x )
    )
end

u = 42
f = x -> x^2
@log(u)      # Expression: u has value: 42
@log(42)     # Expression: 42 has value: 42
@log(f(42))  # Expression: f(42) has value: 1764
@log(:u)     # Expression: :u has value: u
```

**expand per abbassare un** `Expr`

**5 strati (Leah Hanson)** <- spiega come Julia prende il codice sorgente come una stringa, lo converte in un `Expr`-tree (AST), espande tutte le macro (ancora AST), **abbassa** (abbassato AST), quindi converte in LLVM (e oltre - al momento non abbiamo bisogno di preoccuparci di ciò che sta oltre!)

**Q:** `code_lowered` agisce sulle funzioni. È possibile abbassare un `Expr` ? **A:** sì!

```
# function -> lowered-AST
```

```
julia> code_lowered(*, (String, String))
1-element Array{LambdaInfo,1}:
LambdaInfo template for *(s1::AbstractString, ss::AbstractString...) at strings/basic.jl:84

# Expr(i.e. AST) -> lowered-AST
julia> expand(:(x ? y : z))
:(begin
    unless x goto 3
    return y
    3:
    return z
end)

julia> expand(:(y .= x.(i)))
:((Base.broadcast!)(x,y,i))

# 'Execute' AST or lowered-AST
julia> eval(ast)
```

Se vuoi espandere solo macro puoi usare **macroexpand** :

```
# AST -> (still nonlowered-)AST but with macros expanded:
julia> macroexpand(:(@show x))
quote
    (Base.println)("x = ", (Base.repr)(begin # show.jl, line 229:
        #28#value = x
    end))
    #28#value
end
```

... che restituisce un AST non abbassato ma con tutte le macro espanse.

**esc()**

**esc(x)** restituisce un Expr che dice "non applicare l'igiene a questo", è lo stesso di `Expr(:escape, x)`. L'igiene è ciò che mantiene una macro autosufficiente, e si `esc` cose se si vuole che "perdita". per esempio

**Esempio:** `swap` macro per illustrare `esc()`

```
macro swap(p, q)
    quote
        tmp = $(esc(p))
        $(esc(p)) = $(esc(q))
        $(esc(q)) = tmp
    end
end

x, y = 1, 2
@swap(x, y)
println(x, y) # 2 1
```

`$` ci consente di "scappare" dalla `quote`. Quindi, perché non semplicemente `$p` e `$q`? vale a dire

```
# FAIL!
```



```
tmp = $p
$p = $q
$q = tmp
```

Perché ciò dovrebbe guardare prima alla `macro` scope per `p`, e troverà un `p` locale, cioè il parametro `p` (sì, se si accede successivamente a `p` senza `esc`-ing, la macro considera il parametro `p` come una variabile locale).

Quindi `$p = ...` è solo un'assegnazione al `p` locale. non influenza la variabile passata nel contesto di chiamata.

Ok, allora che ne dici:

```
# Almost!
tmp = $p          # <-- you might think we don't
$(esc(p)) = $q    #      need to esc() the RHS
$(esc(q)) = tmp
```

Quindi `esc(p)` sta "perdendo" `p` nel contesto di chiamata. *"La cosa che è stata passata nella macro che riceviamo come `p`"*

```
julia> macro swap(p, q)
    quote
        tmp = $p
        $(esc(p)) = $q
        $(esc(q)) = tmp
    end
end

@swap (macro with 1 method)

julia> x, y = 1, 2
(1,2)

julia> @swap(x, y);

julia> @show(x, y);
x = 2
y = 1

julia> macroexpand(:(@swap(x, y)))
quote # REPL[34], line 3:
    #10#tmp = x # REPL[34], line 4:
    x = y # REPL[34], line 5:
    y = #10#tmp
end
```

Come puoi vedere, `tmp` ottiene il trattamento igienico `#10#tmp`, mentre `x` e `y` no. Julia sta creando un identificatore univoco per `tmp`, qualcosa che puoi fare manualmente con `gensym`, ovvero:

```
julia> gensym(:tmp)
Symbol("#tmp#270")
```

**Ma: c'è un trucco:**

```
julia> module Swap
    export @swap

    macro swap(p, q)
        quote
            tmp = $p
            $(esc(p)) = $q
            $(esc(q)) = tmp
        end
    end
end

Swap

julia> using Swap

julia> x,y = 1,2
(1,2)

julia> @swap(x,y)
ERROR: UndefVarError: x not defined
```

Un'altra cosa che fa l'igiene macro di Julia è che se la macro proviene da un altro modulo, rende qualsiasi variabile (che non è stata assegnata all'interno delle macro di ritorno dell'espressione della macro, come `tmp` in questo caso) del modulo corrente, quindi `$p` diventa `Swap.$p`, allo stesso modo `$q` -> `Swap.$q`.

In generale, se hai bisogno di una variabile che si trova al di fuori dell'ambito della macro, dovresti `esc`, quindi dovresti `esc(p)` ed `esc(q)` indipendentemente dal fatto che siano sul LHS o RHS di un'espressione, o anche da soli.

la gente ha già parlato di `gensym` poche volte e presto sarai sedotto dal lato oscuro del default a fuggire dall'intera espressione con qualche `gensym` s punteggiato qua e là, ma ... Assicurati di capire come funziona l'igiene prima di provare ad essere più intelligente di quello! Non è un algoritmo particolarmente complesso, quindi non dovrebbe richiedere troppo tempo, ma non affrettarlo! Non usare quel potere finché non capisci tutte le implicazioni di esso ... (@fcard)

## Esempio: `until` macro

(@ Ismael-VC)

```
"until loop"
macro until(condition, block)
    quote
        while ! $condition
            $block
        end
    end |> esc
end

julia> i=1; @until( i==5, begin; print(i); i+=1; end )
1234
```

(@fcard) `|>` è controverso, tuttavia. Sono sorpreso che una folla non sia ancora venuta a

discutere. (forse tutti sono solo stanchi di ciò) Vi è una raccomandazione che la maggior parte se non tutta la macro sia solo una chiamata a una funzione, quindi:

```
macro until(condition, block)
    esc(until(condition, block))
end

function until(condition, block)
    quote
        while !$condition
            $block
        end
    end
end
```

... è un'alternativa più sicura

**## @ Sfida macro semplice di fcard**

**Attività:** `swaps(1/2)` gli operandi, quindi gli `swaps(1/2)` restituiscono `2.00` ossia `2/1`

```
macro swaps(e)
    e.args[2:3] = e.args[3:-1:2]
    e
end
@swaps(1/2)
2.00
```

Altre sfide macro da @fcard [qui](#)

---

## Interpolazione e `assert` macro

<http://docs.julialang.org/en/release-0.5/manual/metaprogramming/#building-an-advanced-macro>

```
macro assert(ex)
    return :( $ex ? nothing : throw(AssertionError($(string(ex)))) )
end
```

**Q:** Perché l'ultimo `$` ? **A:** Interpolo, cioè costringe Julia a `eval` quella `string(ex)` mentre l'esecuzione passa attraverso l'invocazione di questa macro. cioè se si esegue solo quel codice non forzerà alcuna valutazione. Ma nel momento in cui fai `assert(foo)` Julia **invocherà** questa macro sostituendo il suo 'token AST / Expr' con qualunque cosa ritorni, e il `$` **entrerà** in azione.

## Un divertente trucco per usare `{}` per i blocchi

(@fcard) Non penso che ci sia qualcosa di tecnico `{}` da utilizzare come blocchi, in effetti si può anche fare un gioco di parole sulla sintassi residua `{}` per farlo funzionare:

```
julia> macro c(block)
    @assert block.head == :cell1d
end
```

```

        esc(quote
            $(block.args...)
        end)
    end
end
@c (macro with 1 method)

julia> @c {
    print(1)
    print(2)
    1+2
}

123

```

\* (è improbabile che funzioni ancora se / quando la sintassi di {} viene riproposta)

**Quindi, per prima cosa, Julia vede il token macro, quindi leggerà / analizzerà i token fino alla end corrispondente e creerà cosa? Un Expr con .head=:macro o qualcosa del genere? Memorizza "a+1" come stringa o lo suddivide in :+(a, 1) ? Come visualizzare?**

?

(@fcard) In questo caso a causa dell'ambito lessicale, a non è definito @M s, quindi usa la variabile globale ... In realtà ho dimenticato di sfuggire all'espressione di flipplin nel mio esempio stupido, ma "funziona solo all'interno del stesso modulo " parte di esso si applica ancora.

```

julia> module M
    macro m()
        :(a+1)
    end
end

M

julia> a = 1
1

julia> M.@m
ERROR: UndefVarError: a not defined

```

Il motivo è che, se la macro viene utilizzata in qualsiasi modulo diverso da quello in cui è stata definita, tutte le variabili non definite all'interno del codice da espandere sono considerate globali del modulo della macro.

```

julia> macroexpand(:(M.@m))
:(M.a + 1)

```

## AVANZATE

### @ Ismael-VC

```
@eval begin
```

```

"do-until loop"
macro $(:do)(block, until::Symbol, condition)
    until ≠ :until &&
        error("@do expected `until` got `$until`")
    quote
        let
            $block
            @until $condition begin
                $block
            end
        end
    end |> esc
end
end
julia> i = 0
0

julia> @do begin
    @show i
    i += 1
end until i == 5

i = 0
i = 1
i = 2
i = 3
i = 4

```

## La macro di Scott:

```

"""
Internal function to return captured line number information from AST

##Parameters
- a:      Expression in the julia type Expr

##Return

- Line number in the file where the calling macro was invoked
"""
_lin(a::Expr) = a.args[2].args[1].args[1]

"""
Internal function to return captured file name information from AST

##Parameters
- a:      Expression in the julia type Expr

##Return
- The name of the file where the macro was invoked
"""
_fil(a::Expr) = string(a.args[2].args[1].args[2])

"""
Internal function to determine if a symbol is a status code or variable
"""
function _is_status(sym::Symbol)
    sym in (:OK, :WARNING, :ERROR) && return true
    str = string(sym)
    length(str) > 4 && (str[1:4] == "ERR_" || str[1:5] == "WARN_" || str[1:5] == "INFO_")
end

```

```

end

"""
Internal function to return captured error code from AST

##Parameters
- a:      Expression in the julia type Expr

##Return
- Error code from the captured info in the AST from the calling macro
"""
_err(a::Expr) =
    (sym = a.args[2].args[2] ; _is_status(sym) ? Expr(:., :Status, QuoteNode(sym)) : sym)

"""
Internal function to produce a call to the log function based on the macro arguments and the
AST from the ()->ERRCODE anonymous function definition used to capture error code, file name
and line number where the macro is used

##Parameters
- level:      Loglevel which has to be logged with macro
- a:          Expression in the julia type Expr
- msgs:       Optional message

##Return
- Statuscode
"""
function _log(level, a, msgs)
    if isempty(msgs)
        :( log($level, $(esc(:Symbol))($_fil(a)), $_lin(a)), $_err(a)) )
    else
        :( log($level, $(esc(:Symbol))($_fil(a)), $_lin(a)), $_err(a)),
message=$(esc(msgs[1])) )
    end
end

macro warn(a, msgs...) ; _log(Warning, a, msgs) ; end

```

## **junk / unprocessed ...**

### **visualizza / scarica una macro**

(@ pi-) Supponiamo di fare solo `macro m() a+1; end` con un nuovo REPL. Senza `a` definito. Come posso 'vederlo'? come, c'è un modo per "scaricare" una macro? Senza effettivamente eseguirlo

(@fcard) Tutti i codici nelle macro sono in realtà messi in funzioni, quindi è possibile visualizzare solo il codice abbassato o il tipo derivato.

```

julia> macro m() a+1 end
@m (macro with 1 method)

julia> @code_typed @m
LambdaInfo for @m()
:(begin
    return Main.a + 1
end)

```

```
julia> @code_lowered @m
CodeInfo(: (begin
    nothing
    return Main.a + 1
end))
# ^ or: code_lowered(eval(Symbol("@m")))[1] # ouf!
```

Altri modi per ottenere la funzione di una macro:

```
julia> macro getmacro(call) call.args[1] end
@getmacro (macro with 1 method)

julia> getmacro(name) = getfield(current_module(), name.args[1])
getmacro (generic function with 1 method)

julia> @getmacro @m
@m (macro with 1 method)

julia> getmacro(:@m)
@m (macro with 1 method)
```

```
julia> eval(Symbol("@M"))
@M (macro with 1 method)

julia> dump( eval(Symbol("@M")) )
@M (function of type #@M)

julia> code_typed( eval(Symbol("@M")) )
1-element Array{Any,1}:
LambdaInfo for @M()

julia> code_typed( eval(Symbol("@M")) )[1]
LambdaInfo for @M()
:(begin
    return $(Expr(:copyast, :($(QuoteNode(:(a + 1))))))
end::Expr)

julia> @code_typed @M
LambdaInfo for @M()
:(begin
    return $(Expr(:copyast, :($(QuoteNode(:(a + 1))))))
end::Expr)
```

^ sembra che io possa usare invece `code_typed`

**Come capire** `eval(Symbol("@M"))` ?

(@fcard) Attualmente, ogni macro ha una funzione ad essa associata. Se hai una macro chiamata `M`, la funzione della macro si chiama `@M`. Generalmente è possibile ottenere il valore di una funzione con ad esempio `eval(:print)` ma con la funzione macro è necessario fare `Symbol("@M")`, poiché proprio `:@M` diventa un `Expr(:macrocall, Symbol("@M"))` e la valutazione che causa una macro-espansione.

**Perché non** `code_typed` **visualizzati i parametri di visualizzazione**

code\_typed ?

(@pi-)

```
julia> code_typed( x -> x^2 )[1]
LambdaInfo for (::##5#6) (::Any)
:(begin
    return x ^ 2
end)
```

^ qui vedo uno `::Any` parametro, ma non sembra essere collegato con il token `x` .

```
julia> code_typed( print )[1]
LambdaInfo for print(::IO, ::Char)
:(begin
    (Base.write)(io,c)
    return Base.nothing
end::Void)
```

^ allo stesso modo qui; non c'è niente da collegare `io` con il `::IO` Quindi sicuramente questo non può essere un dump completo della rappresentazione AST di quel particolare metodo di `print ...?`

(@fcard) `print(::IO, ::Char)` ti dice solo quale metodo è, non fa parte dell'AST. Non è più presente nemmeno nel master:

```
julia> code_typed(print)[1]
CodeInfo:(begin
    (Base.write)(io,c)
    return Base.nothing
end)=>Void
```

(@ pi-) Non capisco cosa intendi con questo. Sembra che si stia scaricando l'AST per il corpo di quel metodo, no? Pensavo che `code_typed` una funzione. Ma sembra che manchi il primo passo, cioè impostare i token per i parametri.

(@fcard) `code_typed` scopo di mostrare solo l'AST del corpo, ma per ora fornisce l'AST completo del metodo, sotto forma di `LambdaInfo` (0.5) o `CodeInfo` (0.6), ma molte informazioni vengono omesse quando viene stampato sul repl. Dovrai ispezionare il campo `LambdaInfo` per campo per ottenere tutti i dettagli. `dump` invaderà il tuo repl, quindi potresti provare:

```
macro method_info(call)
    quote
        method = @code_typed $(esc(call))
        print_info_fields(method)
    end
end

function print_info_fields(method)
    for field in fieldnames(typeof(method))
        if isdefined(method, field) && !(field in [Symbol(""), :code])
            println(" $field = ", getfield(method, field))
        end
    end
end
```



```

end
display(method)
end

print_info_fields(x::Pair) = print_info_fields(x[1])

```

Che fornisce tutti i valori dei campi nominati dell'AST di un metodo:

```

julia> @method_info print(STDOUT, 'a')
rettype = Void
sparam_syms = svec()
sparam_vals = svec()
specTypes = Tuple{Base.#print,Base.TTY,Char}
slottypes = Any[Base.#print,Base.TTY,Char]
ssavaluetypes = Any[]
slotnames = Any[Symbol("#self#"),:io,:c]
slotflags = UInt8[0x00,0x00,0x00]
def = print(io::IO, c::Char) at char.jl:45
nargs = 3
isva = false
inferred = true
pure = false
inlineable = true
inInference = false
inCompile = false
jlcall_api = 0
fptr = Ptr{Void} @0x00007f7a7e96ce10
LambdaInfo for print(::Base.TTY, ::Char)
:(begin
    $(Expr(:invoke, LambdaInfo for write(::Base.TTY, ::Char), :(Base.write), :(io), :(c)))
    return Base.nothing
end)::Void)

```

Vedi lì 'def = print(io::IO, c::Char) ? Ecco qua! (anche lo slotnames = [..., :io, :c] parte)  
Inoltre sì, la differenza di output è perché stavo mostrando i risultati su master.

???

(@ Ismael-VC) intendi in questo modo? [Spedizione generica con simboli](#)

Puoi farlo in questo modo:

```

julia> function dispatchtest{alg}(::Type{Val{alg}})
    println("This is the generic dispatch. The algorithm is $alg")
end
dispatchtest (generic function with 1 method)

julia> dispatchtest{alg::Symbol} = dispatchtest{Val{alg}}
dispatchtest (generic function with 2 methods)

julia> function dispatchtest{alg::Type{Val{Euler}}}
    println("This is for the Euler algorithm!")
end
dispatchtest (generic function with 3 methods)

julia> dispatchtest{Foo}

```

```
This is the generic dispatch. The algorithm is Foo
```

```
julia> dispatchtest(:Euler)
```

Questo è per l'algoritmo di Eulero! Mi chiedo cosa pensi @fcard della spedizione di simboli generici! --- ^: angelo:

## Modulo Gotcha

```
@def m begin
    a+2
end

@m # replaces the macro at compile-time with the expression a+2
```

Più precisamente, funziona solo all'interno del livello superiore del modulo in cui è stata definita la macro.

```
julia> module M
    macro m1()
        a+1
    end
end
M

julia> macro m2()
    a+1
end
@m2 (macro with 1 method)

julia> a = 1
1

julia> M.@m1
ERROR: UndefVarError: a not defined

julia> @m2
2

julia> let a = 20
    @m2
end
2
```

`esc` evita che ciò accada, ma il default di usarlo sempre va contro il design del linguaggio. Una buona difesa per questo è impedirne l'uso e l'introduzione di nomi all'interno di macro, il che rende difficile rintracciare un lettore umano.

**Python `dict` / JSON come sintassi per i letterali `Dict`.**

## introduzione

Julia usa la seguente sintassi per i dizionari:

```
Dict({k1 => v1, k2 => v2, ..., kn-1 => vn-1, kn => vn})
```

Mentre Python e JSON hanno questo aspetto:

```
{k1: v1, k2: v2, ..., kn-1: vn-1, kn: vn}
```

A **scopo illustrativo** potremmo anche usare questa sintassi in Julia e aggiungere nuove semantiche (la sintassi di `Dict` è il modo idiomatico in Julia, che è raccomandato).

Per prima cosa vediamo che *tipo* di espressione è:

```
julia> parse("{1:2 , 3: 4}") |> Meta.show_sexpr  
(:cellld, (:(:), 1, 2), (:(:), 3, 4))
```

Ciò significa che dobbiamo prendere questa `:cellld` espressione `:cellld` e trasformarla o restituire una nuova espressione che dovrebbe assomigliare a questa:

```
julia> parse("Dict(1 => 2 , 3 => 4)") |> Meta.show_sexpr  
(:call, :Dict, (:(>)), 1, 2), (:(>)), 3, 4))
```

## Definizione macro

La seguente macro, sebbene semplice, consente di dimostrare tale generazione e trasformazione del codice:

```
macro dict(expr)  
    # Check the expression has the correct form:  
    if expr.head ≠ :cellld || any(sub_expr.head ≠ :(:) for sub_expr ∈ expr.args)  
        error("syntax: expected `{k1: v1, k2: v2, ..., kn-1: vn-1, kn: vn}`")  
    end  
  
    # Create empty `:Dict` expression which will be returned:  
    block = Expr(:call, :Dict) # :(:Dict())  
  
    # Append `(key => value)` pairs to the block:  
    for pair in expr.args  
        k, v = pair.args  
        push!(block.args, :($k => $v))  
    end # :(:Dict(k1 => v1, k2 => v2, ..., kn-1 => vn-1, kn => vn))  
  
    # Block is escaped so it can reach variables from it's calling scope:  
    return esc(block)  
end
```

Diamo un'occhiata all'espansione della macro risultante:

```
julia> :(@dict {"a": :b, 'c': 1, :d: 2.0}) |> macroexpand  
:(:Dict("a" => :b, 'c' => 1, :d => 2.0))
```

## USO

```
julia> @dict {"a": :b, 'c': 1, :d: 2.0}
Dict{Any,Any} with 3 entries:
  "a" => :b
  :d  => 2.0
  'c' => 1
```

```
julia> @dict {
    "string": :b,
    'c'       : 1,
    :symbol   : π,
    Function: print,
    (1:10)    : range(1, 10)
}
Dict{Any,Any} with 5 entries:
  1:10      => 1:10
  Function  => print
  "string"  => :b
  :symbol   => π = 3.1415926535897...
  'c'       => 1
```

L'ultimo esempio è esattamente equivalente a:

```
Dict(
    "string" => :b,
    'c'      => 1,
    :symbol  => π,
    Function => print,
    (1:10)   => range(1, 10)
)
```

## cattivo uso

```
julia> @dict {"one": 1, "two": 2, "three": 3, "four": 4, "five" => 5}
syntax: expected `{k₁: v₁, k₂: v₂, ..., kₙ₋₁: vₙ₋₁, kₙ: vₙ}`

julia> @dict ["one": 1, "two": 2, "three": 3, "four": 4, "five" => 5]
syntax: expected `{k₁: v₁, k₂: v₂, ..., kₙ₋₁: vₙ₋₁, kₙ: vₙ}`
```

Si noti che Julia ha altri usi per i due punti : in questo modo è necessario includere le espressioni letterali con parentesi o utilizzare la funzione `range` , ad esempio.

Leggi metaprogrammazione online: <https://riptutorial.com/it/julia-lang/topic/1945/metaprogrammazione>

# Capitolo 25: moduli

## Sintassi

- `modulo modulo; ...; fine`
- `usando il modulo`
- `modulo di importazione`

## Examples

### Avvolgere il codice in un modulo

La parola chiave `module` può essere utilizzata per iniziare un modulo, che consente di organizzare il codice e il namespace. I moduli possono definire un'interfaccia esterna, in genere costituita da simboli `export` ed. Per supportare questa interfaccia esterna, i moduli possono avere [funzioni](#) interne non espresse e [tipi](#) non destinati all'uso pubblico.

Alcuni moduli esistono principalmente per racchiudere un tipo e funzioni associate. Tali moduli, per convenzione, vengono solitamente denominati con la forma plurale del nome del tipo. Ad esempio, se abbiamo un modulo che fornisce un tipo di `Building`, possiamo chiamare tali moduli `Buildings`.

```
module Buildings

  immutable Building
    name::String
    stories::Int
    height::Int # in metres
  end

  name(b::Building) = b.name
  stories(b::Building) = b.stories
  height(b::Building) = b.height

  function Base.show(io::IO, b::Building)
    Base.print(stories(b), "-story ", name(b), " with height ", height(b), "m")
  end

  export Building, name, stories, height

end
```

Il modulo può quindi essere utilizzato con l'istruzione `using`:

```
julia> using Buildings

julia> Building("Burj Khalifa", 163, 830)
163-story Burj Khalifa with height 830m

julia> height(ans)
```

## Utilizzo dei moduli per organizzare i pacchetti

In genere, i **pacchetti** sono costituiti da uno o più moduli. Man mano che i pacchetti crescono, può essere utile organizzare il modulo principale del pacchetto in moduli più piccoli. Un idiomma comune è definire quei moduli come sottomoduli del modulo principale:

```
module RootModule

module SubModule1

...

end

module SubModule2

...

end

end
```

Inizialmente, né i moduli radice né i sottomoduli hanno accesso ai reciproci simboli esportati. Tuttavia, le importazioni relative sono supportate per risolvere questo problema:

```
module RootModule

module SubModule1

const x = 10
export x

end

module SubModule2

# import submodule of parent module
using ..SubModule1
const y = 2x
export y

end

# import submodule of current module
using .SubModule1
using .SubModule2
const z = x + y

end
```

In questo esempio, il valore di `RootModule.z` è 30 .

**Leggi moduli online:** <https://riptutorial.com/it/julia-lang/topic/7368/moduli>

# Capitolo 26: Normalizzazione delle stringhe

## Sintassi

- `normalize_string` (`s :: String, ...`)

## Parametri

Parametro	Dettagli
<code>casefold=true</code>	Piega la stringa in un caso canonico basato sullo standard <a href="#">Unicode</a> .
<code>stripmark=true</code>	Striscia <a href="#">segni diacritici</a> (cioè accenti) da caratteri nella stringa di input.

## Examples

### Confronto tra stringhe senza distinzione tra maiuscole e minuscole

[Le stringhe](#) possono essere confrontate con l' [operatore](#) `==` in Julia, ma questo è sensibile alle differenze nel caso. Ad esempio, `"Hello"` e `"hello"` sono considerati stringhe diverse.

```
julia> "Hello" == "Hello"
true

julia> "Hello" == "hello"
false
```

Per confrontare le stringhe in modo non sensibile alla distinzione tra maiuscole e minuscole, normalizza le stringhe in base al caso, piegandole prima. Per esempio,

```
equals_ignore_case(s, t) =
    normalize_string(s, casefold=true) == normalize_string(t, casefold=true)
```

Questo approccio gestisce anche Unicode non ASCII correttamente:

```
julia> equals_ignore_case("Hello", "hello")
true

julia> equals_ignore_case("Weierstraß", "WEIERSTRASS")
true
```

Nota che in tedesco la forma maiuscola del carattere ß è SS.

### Confronto tra stringhe insensibili ai diacritici

A volte, si vogliono stringhe come `"resume"` e `"résumé"` per confrontare lo stesso. Cioè, [grafemi](#)

che condividono un glifo di base, ma forse differiscono a causa delle aggiunte a quei glifi di base. Tale confronto può essere ottenuto eliminando segni diacritici.

```
equals_ignore_mark(s, t) =  
    normalize_string(s, stripmark=true) == normalize_string(t, stripmark=true)
```

Ciò consente all'esempio sopra di funzionare correttamente. Inoltre, funziona bene anche con caratteri Unicode non ASCII.

```
julia> equals_ignore_mark("resume", "résumé ")  
true  
  
julia> equals_ignore_mark("αβγ", "à β ŷ ")  
true
```

Leggi Normalizzazione delle stringhe online: <https://riptutorial.com/it/julia-lang/topic/7612/normalizzazione-delle-stringhe>



# Capitolo 27: Pacchi

## Sintassi

- `Pkg.add` (pacchetto)
- `Pkg.checkout` (pacchetto, `branch = "master"`)
- `Pkg.clone` (url)
- `Pkg.dir` (pacchetto)
- `Pkg.pin` (pacchetto, versione)
- `Pkg.rm` (pacchetto)

## Parametri

Parametro	Dettagli
<code>Pkg.add( package )</code>	Scarica e installa il pacchetto registrato.
<code>Pkg.checkout( package , branch )</code>	Controlla il ramo indicato per il pacchetto registrato. <i>branch</i> è facoltativo e il valore predefinito è "master" .
<code>Pkg.clone( url )</code>	Clona il repository Git all'URL specificato come pacchetto.
<code>Pkg.dir( package )</code>	Ottieni il percorso su disco per il pacchetto indicato.
<code>Pkg.pin( package , version )</code>	Forza il pacchetto a rimanere alla versione specificata. <i>version</i> è facoltativa e ha come valore predefinito la versione corrente del pacchetto.
<code>Pkg.rm( package )</code>	Rimuovi il pacchetto indicato dall'elenco dei pacchetti richiesti.

## Examples

### Installa, usa e rimuovi un pacchetto registrato

Dopo aver trovato un pacchetto ufficiale Julia, è semplice scaricare e installare il pacchetto. Innanzitutto, si consiglia di aggiornare la copia locale di METADATA:

```
julia> Pkg.update()
```

Questo assicurerà di ottenere le ultime versioni di tutti i pacchetti.

Supponiamo che il pacchetto che vogliamo installare sia denominato `Currencies.jl` . Il comando da eseguire per installare questo pacchetto sarebbe:

```
julia> Pkg.add("Currencies")
```

Questo comando installerà non solo il pacchetto stesso, ma anche tutte le sue dipendenze.

Se l'installazione ha esito positivo, puoi [verificare che il pacchetto funzioni correttamente](#) :

```
julia> Pkg.test("Currencies")
```

Quindi, per utilizzare il pacchetto, utilizzare

```
julia> using Currencies
```

e procedere come descritto dalla documentazione del pacchetto, solitamente collegata o inclusa nel suo file README.md.

Per disinstallare un pacchetto che non è più necessario, utilizzare la funzione `Pkg.rm` :

```
julia> Pkg.rm("Currencies")
```

Si noti che questo potrebbe non rimuovere effettivamente la directory del pacchetto; invece segnerà semplicemente il pacchetto come non più richiesto. Spesso, questo è perfettamente a posto - farà risparmiare tempo nel caso in cui sia necessario il pacchetto di nuovo in futuro. Ma se necessario, per rimuovere fisicamente il pacchetto, chiamare la funzione `rm`, quindi chiamare `Pkg.resolve` :

```
julia> rm(Pkg.dir("Currencies"); recursive=true)
```

```
julia> Pkg.resolve()
```

## Scopri un altro ramo o versione

A volte, l'ultima versione codificata di un pacchetto è bacata o manca alcune funzionalità richieste. Gli utenti esperti potrebbero voler aggiornare la versione di sviluppo più recente di un pacchetto (a volte indicato come "master", dal nome abituale di un [ramo di sviluppo](#) in Git). I vantaggi di questo includono:

- Gli sviluppatori che contribuiscono a un pacchetto dovrebbero contribuire all'ultima versione di sviluppo.
- L'ultima versione di sviluppo può avere funzionalità utili, correzioni di bug o miglioramenti delle prestazioni.
- Gli utenti che segnalano un bug potrebbero voler verificare se si verifica un errore nell'ultima versione di sviluppo.

Tuttavia, ci sono molti inconvenienti per l'esecuzione dell'ultima versione di sviluppo:

- L'ultima versione di sviluppo potrebbe essere poco testata e presentare problemi gravi.
- L'ultima versione di sviluppo può cambiare frequentemente, infrangendo il tuo codice.

Per esempio, usa l'ultimo ramo di sviluppo di un pacchetto chiamato [JSON.jl](#)

```
Pkg.checkout("JSON")
```

Per controllare un altro ramo o tag (non chiamato "master"), usa

```
Pkg.checkout("JSON", "v0.6.0")
```

Tuttavia, se il tag rappresenta una versione, di solito è meglio usarla

```
Pkg.pin("JSON", v"0.6.0")
```

Si noti che qui viene utilizzata una versione letterale, non una stringa semplice. La versione di `Pkg.pin` informa il gestore pacchetti del vincolo della versione, consentendo al gestore di pacchetti di fornire un feedback su quali problemi potrebbe causare.

Per tornare all'ultima versione con tag,

```
Pkg.free("JSON")
```

## Installa un pacchetto non registrato

Alcuni pacchetti sperimentali non sono inclusi nel repository del pacchetto METADATA. Questi pacchetti possono essere installati clonando direttamente i loro repository Git. Si noti che potrebbero esserci delle dipendenze da pacchetti non registrati che sono essi stessi non registrati; tali dipendenze non possono essere risolte dal gestore pacchetti e devono essere risolte manualmente. Ad esempio, per installare il pacchetto non registrato [OhMyREPL.jl](#) :

```
Pkg.clone("https://github.com/KristofferC/Tokenize.jl")  
Pkg.clone("https://github.com/KristofferC/OhMyREPL.jl")
```

Quindi, come al solito, utilizzare `using` per utilizzare il pacchetto:

```
using OhMyREPL
```

Leggi Pacchi online: <https://riptutorial.com/it/julia-lang/topic/5815/pacchi>

# Capitolo 28: per loop

## Sintassi

- per i in iter; ...; fine
- mentre cond; ...; fine
- rompere
- Continua
- @parallel (op) per i in iter; ...; fine
- @parallel for i in iter; ...; fine
- Etichetta @goto
- etichetta @label

## Osservazioni

Ogni volta che rende il codice più breve e più facile da leggere, considera l'utilizzo di funzioni di ordine superiore, come la `map` o il `filter`, anziché i loop.

## Examples

### Fizz Buzz

Un caso d'uso comune per un ciclo `for` è quello di iterare su un intervallo o una collezione predefiniti, e fare lo stesso compito per tutti i suoi elementi. Ad esempio, qui combiniamo un ciclo `for` con un'istruzione condizionale `if - elsif - else`:

```
for i in 1:100
  if i % 15 == 0
    println("FizzBuzz")
  elseif i % 3 == 0
    println("Fizz")
  elseif i % 5 == 0
    println("Buzz")
  else
    println(i)
  end
end
```

Questa è la classica domanda dell'intervista di [Fizz Buzz](#). L'output (troncato) è:

```
1
2
Fizz
4
Buzz
Fizz
7
8
```

## Trova il fattore primo più piccolo

In alcune situazioni, si potrebbe voler tornare da una funzione prima di terminare un intero ciclo. Il `return` dichiarazione può essere utilizzato per questo.

```
function primefactor(n)
    for i in 2:n
        if n % i == 0
            return i
        end
    end
    @assert false # unreachable
end
```

Uso:

```
julia> primefactor(100)
2

julia> primefactor(97)
97
```

I loop possono anche essere terminati in anticipo con l'istruzione `break`, che termina solo il ciclo di chiusura anziché l'intera funzione.

## Iterazione multidimensionale

In Julia, un ciclo `for` può contenere una virgola ( , ) per specificare il iterazione su più dimensioni. Questo agisce in modo simile per annidare un ciclo all'interno di un altro, ma può essere più compatto. Ad esempio, la funzione seguente genera elementi del [prodotto cartesiano](#) di due iterabili:

```
function cartesian(xs, ys)
    for x in xs, y in ys
        produce(x, y)
    end
end
```

Uso:

```
julia> collect(@task cartesian(1:2, 1:4))
8-element Array{Tuple{Int64,Int64},1}:
 (1,1)
 (1,2)
 (1,3)
 (1,4)
 (2,1)
 (2,2)
 (2,3)
 (2,4)
```

Tuttavia, l'indicizzazione su array di qualsiasi dimensione dovrebbe essere eseguita con ogni

`eachindex` , non con un ciclo multidimensionale (se possibile):

```
s = zero(eltype(A))
for ind in eachindex(A)
    s += A[ind]
end
```

## Riduzione e loop paralleli

Julia fornisce macro per semplificare la distribuzione del calcolo su più macchine o lavoratori. Ad esempio, il seguente calcola la somma di un certo numero di quadrati, possibilmente in parallelo.

```
function sumofsquares(A)
    @parallel (+) for i in A
        i ^ 2
    end
end
```

Uso:

```
julia> sumofsquares(1:10)
385
```

Per ulteriori informazioni su questo argomento, vedere l' [esempio](#) su `@parallel` all'interno dell'[argomento](#) Processi paralleli.

Leggi per loop online: <https://riptutorial.com/it/julia-lang/topic/4355/per-loop>

# Capitolo 29: regex

## Sintassi

- `Regex ("regex")`
- `r "regex"`
- `partita (ago, pagliaio)`
- `matchall (ago, pagliaio)`
- `eachmatch (ago, pagliaio)`
- `ismatch (ago, pagliaio)`

## Parametri

Parametro	Dettagli
<code>needle</code>	il <code>Regex</code> da cercare nel <code>haystack</code>
<code>haystack</code>	il testo in cui cercare l' <code>needle</code>

## Examples

### Regalati letterali

Julia supporta le espressioni regolari <sup>1</sup>. La libreria PCRE viene utilizzata come implementazione regex. I regex sono come una mini-lingua in una lingua. Poiché la maggior parte delle lingue e molti editor di testo forniscono un supporto per regex, la documentazione e gli esempi di come utilizzare le [espressioni regolari](#) in generale non rientrano nell'ambito di questo esempio.

È possibile costruire un `Regex` da una stringa usando il costruttore:

```
julia> Regex("(cat|dog)s?")
```

Ma per comodità e facilità di escape, è possibile utilizzare la [macro stringa](#) `@r_str` invece:

```
julia> r"(cat|dog)s?"
```

<sup>1</sup>: Tecnicamente, Julia supporta regex, che sono distinte e più potenti di quelle che vengono chiamate [espressioni regolari](#) nella teoria del linguaggio. Frequentemente, il termine "espressione regolare" sarà usato per riferirsi anche alle regex.

### Trovare partite

Ci sono quattro funzioni utili primarie per le espressioni regolari, che prendono tutti argomenti in `needle`, `haystack` ordine di `needle`, `haystack`. La terminologia "ago" e "pagliaio" derivano dall'idioma

inglese "trovare un ago in un pagliaio". Nel contesto delle espressioni regolari, la regex è l'ago e il testo è il pagliaio.

La funzione di `match` può essere utilizzata per trovare la prima corrispondenza in una stringa:

```
julia> match(r"(cat|dog)s?", "my cats are dogs")
RegexMatch{"cats", 1="cat"}
```

La funzione di `matchall` può essere utilizzata per trovare tutte le corrispondenze di un'espressione regolare in una stringa:

```
julia> matchall(r"(cat|dog)s?", "The cat jumped over the dogs.")
2-element Array{SubString{String},1}:
 "cat"
 "dogs"
```

La funzione `ismatch` restituisce un valore booleano che indica se è stata trovata una corrispondenza all'interno della stringa:

```
julia> ismatch(r"(cat|dog)s?", "My pigs")
false

julia> ismatch(r"(cat|dog)s?", "My cats")
true
```

La funzione `eachmatch` restituisce un iteratore su oggetti `RegexMatch`, adatto per l'uso con i cicli `for`:

```
julia> for m in eachmatch(r"(cat|dog)s?", "My cats and my dog")
    println("Matched $(m.match) at index $(m.offset)")
end
Matched cats at index 4
Matched dog at index 16
```

## Cattura gruppi

Le sottostringhe catturate dai [gruppi di cattura](#) sono accessibili dagli oggetti `RegexMatch` usando la notazione di indicizzazione.

Ad esempio, la regex seguente analizza i numeri di telefono nordamericani scritti nel formato (555)-555-5555:

```
julia> phone = r"\((\d{3})\)-(\d{3})-(\d{4})"
```

e supponiamo di voler estrarre i numeri di telefono da un testo:

```
julia> text = """
    My phone number is (555)-505-1000.
    Her phone number is (555)-999-9999.
    """

"My phone number is (555)-505-1000.\nHer phone number is (555)-999-9999.\n"
```



Usando la funzione `matchall`, possiamo ottenere una matrice di sottostringhe corrispondenti:

```
julia> matchall(phone, text)
2-element Array{SubString{String},1}:
 "(555)-505-1000"
 "(555)-999-9999"
```

Ma supponiamo di voler accedere ai prefissi (le prime tre cifre, racchiuse tra parentesi). Quindi possiamo usare l'iteratore `eachmatch`:

```
julia> for m in eachmatch(phone, text)
    println("Matched $(m.match) with area code $(m[1])")
end
Matched (555)-505-1000 with area code 555
Matched (555)-999-9999 with area code 555
```

Nota qui che usiamo `m[1]` perché il prefisso è il primo gruppo di cattura nella nostra espressione regolare. Possiamo ottenere tutte e tre le componenti del numero di telefono come una tupla usando una funzione:

```
julia> splitmatch(m) = m[1], m[2], m[3]
splitmatch (generic function with 1 method)
```

Quindi possiamo applicare tale funzione a un particolare `RegexMatch`:

```
julia> splitmatch(match(phone, text))
("555", "505", "1000")
```

O potremmo `map` trasversalmente ogni partita:

```
julia> map(splitmatch, eachmatch(phone, text))
2-element Array{Tuple{SubString{String},SubString{String},SubString{String}},1}:
 ("555", "505", "1000")
 ("555", "999", "9999")
```

Leggi regex online: <https://riptutorial.com/it/julia-lang/topic/5890/regex>

# Capitolo 30: REPL

## Sintassi

- `julia>`
- `aiutare?>`
- `shell>`
- `\[Latex]`

## Osservazioni

Altri pacchetti possono definire le proprie modalità REPL oltre alle modalità predefinite. Ad esempio, il pacchetto `Cxx` definisce la modalità `shell cxx>` per un REPL C ++. Queste modalità sono solitamente accessibili con i propri tasti speciali; vedere la documentazione del pacchetto per maggiori dettagli.

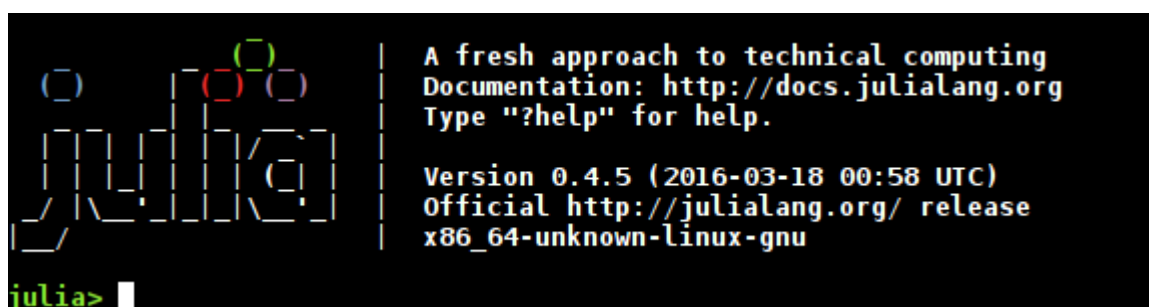
## Examples

### Avvia il REPL

Dopo aver [installato Julia](#) , per avviare il read-eval-print-loop (REPL):

## Su sistemi Unix

Apri una finestra di terminale, quindi digita `julia` al prompt, quindi premi `Invio` . Dovresti vedere qualcosa di simile:



## Su Windows

Trova il programma Julia nel menu di avvio e fai clic su di esso. Il REPL dovrebbe essere lanciato.

### Utilizzo del REPL come calcolatore

Il REPL Julia è un calcolatore eccellente. Possiamo iniziare con alcune semplici operazioni:

```
julia> 1 + 1
2

julia> 8 * 8
64

julia> 9 ^ 2
81
```

La variabile `ans` contiene il risultato dell'ultimo calcolo:

```
julia> 4 + 9
13

julia> ans + 9
22
```

Possiamo definire le nostre variabili utilizzando l'operatore assignment `=` :

```
julia> x = 10
10

julia> y = 20
20

julia> x + y
30
```

Julia ha una moltiplicazione implicita per i valori letterali numerici, il che rende alcuni calcoli più veloci da scrivere:

```
julia> 10x
100

julia> 2(x + y)
60
```

Se commettiamo un errore e facciamo qualcosa che non è permesso, il REPL di Julia genererà un errore, spesso con un suggerimento utile su come risolvere il problema:

```
julia> 1 ^ -1
ERROR: DomainError:
Cannot raise an integer x to a negative power -n.
Make x a float by adding a zero decimal (e.g. 2.0^-n instead of 2^-n), or write
1/x^n, float(x)^-n, or (x//1)^-n.
 in power_by_squaring at ./intfuncs.jl:82
 in ^ at ./intfuncs.jl:106

julia> 1.0 ^ -1
1.0
```

Per accedere o modificare i comandi precedenti, utilizzare il tasto `↑` (Su), che si sposta sull'ultimo elemento della cronologia. Il `↓` si sposta sull'elemento successivo nella cronologia. I tasti `←` e `→` possono essere utilizzati per spostare e apportare modifiche a una linea.

Julia ha alcune costanti matematiche incorporate, tra cui  $e$  e  $\pi$  ( $\text{pi}$ ).

```
julia> e
e = 2.7182818284590...

julia> pi
π = 3.1415926535897...

julia> 3π
9.42477796076938
```

Possiamo digitare rapidamente caratteri come  $\pi$  usando i loro codici LaTeX: premi `\`, quindi `p` e `i`, quindi premi il tasto `Tab` per sostituire il `\pi` appena digitato con  $\pi$ . Funziona con altre lettere greche e simboli Unicode aggiuntivi.

Possiamo utilizzare qualsiasi funzione matematica integrata di Julia, che va dalla semplice alla abbastanza potente:

```
julia> cos(π)
-1.0

julia> besselh(1, 1, 1)
0.44005058574493355 - 0.7812128213002889im
```

I numeri complessi sono supportati usando `im` come unità immaginaria:

```
julia> abs(3 + 4im)
5.0
```

Alcune funzioni non restituiscono un risultato complesso a meno che non gli dai un input complesso, anche se l'input è reale:

```
julia> sqrt(-1)
ERROR: DomainError:
sqrt will only return a complex result if called with a complex argument. Try
sqrt(complex(x)).
in sqrt at math.jl:146

julia> sqrt(-1+0im)
0.0 + 1.0im

julia> sqrt(complex(-1))
0.0 + 1.0im
```

Le operazioni esatte sui numeri razionali sono possibili usando l'operatore `//` rational division:

```
julia> 1//3 + 1//3
2//3
```

Vedi l'argomento [Arithmetic](#) per ulteriori informazioni su quali tipi di operatori aritmetici sono supportati da Julia.

## Trattare con la precisione della macchina

Si noti che gli interi macchina sono vincolati in termini di dimensioni e si riempiranno se il risultato è troppo grande per essere archiviato:

```
julia> 2^62
4611686018427387904

julia> 2^63
-9223372036854775808
```

Questo può essere evitato usando numeri interi arbitrari di precisione nel calcolo:

```
julia> big"2"^62
4611686018427387904

julia> big"2"^63
9223372036854775808
```

Anche i punti mobili della macchina sono limitati in precisione:

```
julia> 0.1 + 0.2
0.30000000000000004
```

Più (ma ancora limitata) precisione è possibile usando di nuovo `big`

[illegible]

L'aritmetica esatta può essere eseguita in alcuni casi usando `Rational` s:

```
julia> 1//10 + 2//10
3//10
```

## Utilizzo delle modalità REPL

Esistono tre modalità REPL incorporate in Julia: la modalità Julia, la modalità di guida e la modalità shell.

## La modalità Guida

Julia REPL è dotato di un sistema di aiuto integrato. Premere `?` al prompt di `julia>` per accedere al prompt `help?>`.

Al prompt della guida, digitare il nome di alcune funzioni o tipi per ottenere aiuto per:

```

help?> abs
search: abs abs2 abspath abstract AbstractRNG AbstractFloat AbstractArray

abs(x)

The absolute value of x.

When abs is applied to signed integers, overflow may occur, resulting in the
return of a negative value. This overflow occurs only when abs is applied to the
minimum representable value of a signed integer. That is, when x ==
typemin(typeof(x)), abs(x) == x < 0, not -x as might be expected.

```

Anche se non si scrive correttamente la funzione, Julia può suggerire alcune funzioni che potrebbero essere ciò che intendevi:

```

help?> printline
search:

Couldn't find printline
Perhaps you meant println, pipeline, @inline or print
No documentation found.

Binding printline does not exist.

```

Questa documentazione funziona anche per altri moduli, purché utilizzino il sistema di documentazione di Julia.

```

julia> using Currencies

help?> @usingcurrencies
Export each given currency symbol into the current namespace. The individual unit
exported will be a full unit of the currency specified, not the smallest possible
unit. For instance, @usingcurrencies EUR will export EUR, a currency unit worth
1€, not a currency unit worth 0.01€.

@usingcurrencies EUR, GBP, AUD
7AUD # 7.00 AUD

There is no sane unit for certain currencies like XAU or XAG, so this macro does
not work for those. Instead, define them manually:

const XAU = Monetary(:XAU; precision=4)

```

## La modalità Shell

Vedi [Usare Shell all'interno del REPL](#) per maggiori dettagli su come utilizzare la modalità shell di Julia, che è accessibile premendo ; al prompt. Questa modalità shell supporta l'interpolazione dei dati dalla sessione REPL Julia, il che rende facile chiamare le funzioni di Julia e rendere i loro risultati nei comandi della shell:

```

shell> ls $(Pkg.dir("JSON"))
appveyor.yml bench data LICENSE.md nohup.out README.md REQUIRE src test

```

Leggi REPL online: <https://riptutorial.com/it/julia-lang/topic/5739/repl>

# Capitolo 31: Scripting Shell e Piping

## Sintassi

- ; comando di shell

## Examples

### Utilizzo di Shell dall'interno del REPL

Dall'interno della shell interattiva di Julia (nota anche come REPL), è possibile accedere alla shell del sistema digitando ; subito dopo il prompt:

```
shell>
```

Da qui in poi, puoi digitare qualsiasi comando della shell e verranno eseguiti da REPL:

```
shell> ls
Desktop      Documents  Pictures   Templates
Downloads    Music      Public     Videos
```

Per uscire da questa modalità, digitare `backspace` quando il prompt è vuoto.

### Shelling fuori dal codice di Julia

Il codice Julia può creare, manipolare ed eseguire i letterali dei comandi, che vengono eseguiti nell'ambiente di sistema del sistema operativo. Questo è potente ma spesso rende i programmi meno portabili.

Un letterale di comando può essere creato usando il `` letterale '. Le informazioni possono essere interpolate usando la sintassi \$ interpolation, come con stringhe letterali. Le variabili di Julia passate attraverso i letterali di comando non devono essere prima sottoposte a escape; in realtà non vengono passati alla shell, ma piuttosto direttamente al kernel. Tuttavia, Julia visualizza questi oggetti in modo che appaiano correttamente sfuggiti.

```
julia> msg = "a commit message"
"a commit message"

julia> command = `git commit -am $msg`
`git commit -am 'a commit message'`

julia> cd("/directory/where/there/are/unstaged/changes")

julia> run(command)
[master (root-commit) 0945387] add a
4 files changed, 1 insertion(+)
```

Leggi Scripting Shell e Piping online: <https://riptutorial.com/it/julia-lang/topic/5420/scripting-shell-e-piping>



# Capitolo 32: stringhe

## Sintassi

- "[stringa]"
- "[Valore scalare Unicode]"
- grafemi ([string])

## Parametri

Parametro	Dettagli
Per	<code>sprint(f, xs...)</code>
f	Una funzione che accetta un oggetto <code>IO</code> come primo argomento.
xs	Zero o più argomenti rimanenti da passare a <code>f</code> .

## Examples

### Ciao mondo!

Le stringhe in Julia sono delimitate usando il `"` simbolo:

```
julia> mystring = "Hello, World!"
"Hello, World!"
```

Nota che a differenza di altre lingue, il simbolo `'` *non può* essere usato al suo posto. `'` definisce un *carattere letterale*; questo è un tipo di dati `Char` e memorizzerà solo un singolo [valore scalare Unicode](#):

```
julia> 'c'
'c'

julia> 'character'
ERROR: syntax: invalid character literal
```

Si può estrarre i valori scalari Unicode da una stringa ripetendo su di esso con un [ciclo for](#):

```
julia> for c in "Hello, World!"
    println(c)
end
H
e
l
l
o
,
```

```
o  
,  
  
w  
o  
r  
l  
d  
!
```

## grafemi

Il tipo `Char` di Julia rappresenta un [valore scalare Unicode](#), che solo in alcuni casi corrisponde a quello che l'uomo percepisce come un "personaggio". Ad esempio, una rappresentazione del carattere `é`, come in curriculum, è in realtà una combinazione di due valori scalari Unicode:

```
julia> collect("é ")  
2-element Array{Char,1}:  
 'e'  
  ' '
```

Le descrizioni Unicode per questi codepoint sono "LATIN SMALL LETTER E" e "COMBINING ACUTE ACCENT". Insieme, definiscono un singolo carattere "umano", che è un termine Unicode chiamato [grafo](#). Più specificamente, Unicode Annex # 29 motiva la definizione di un [cluster grapheme](#) perché:

È importante riconoscere che ciò che l'utente pensa come "personaggio" - un'unità di base di un sistema di scrittura per una lingua - potrebbe non essere solo un singolo punto di codice Unicode. Invece, quell'unità di base può essere composta da più punti di codice Unicode. Per evitare l'ambiguità con l'uso del termine del carattere da parte del computer, questo è chiamato carattere percepito dall'utente. Ad esempio, "G" + accento acuto è un personaggio percepito dall'utente: gli utenti lo considerano come un singolo carattere, ma in realtà è rappresentato da due punti di codice Unicode. Questi caratteri percepiti dall'utente sono approssimati da quello che viene chiamato un grafo grapheme, che può essere determinato a livello di codice.

Julia fornisce la funzione `graphemes` per scorrere i grapheme cluster in una stringa:

```
julia> for c in graphemes("résumé ")  
    println(c)  
end  
  
r  
é  
s  
u  
m  
é  
 
```

Nota come il risultato, stampando ogni carattere sulla sua stessa riga, è migliore rispetto a se avessimo eseguito un'iterazione sui valori scalari Unicode:

```
julia> for c in "résumé "  
        println(c)  
    end  
  
r  
e  
  
s  
u  
m  
e
```

In genere, quando si lavora con i caratteri in un senso percepito dall'utente, è più utile trattare con grapheme cluster rispetto ai valori scalari Unicode. Ad esempio, supponiamo di voler scrivere una funzione per calcolare la lunghezza di una singola parola. Una soluzione ingenua sarebbe da usare

```
julia> wordlength(word) = length(word)  
wordlength (generic function with 1 method)
```

Notiamo che il risultato è contro-intuitivo quando la parola include grapheme cluster costituiti da più di un punto di codice:

```
julia> wordlength("résumé ")  
8
```

Quando usiamo la definizione più corretta, usando la funzione `graphemes`, otteniamo il risultato atteso:

```
julia> wordlength(word) = length(graphemes(word))  
wordlength (generic function with 1 method)  
  
julia> wordlength("résumé ")  
6
```

## Converti tipi numerici in stringhe

Esistono numerosi modi per convertire i tipi numerici in stringhe in Julia:

```
julia> a = 123  
123  
  
julia> string(a)  
"123"  
  
julia> println(a)  
123
```

La funzione `string()` può anche richiedere più argomenti:

```
julia> string(a, "b")  
"123b"
```

Puoi anche inserire (anche interpolare) interi (e certi altri tipi) in stringhe usando `$` :

```
julia> MyString = "my integer is $a"  
"my integer is 123"
```

**Suggerimento sulle prestazioni:** i metodi sopra riportati possono essere abbastanza convenienti a volte. Ma, se eseguirai molte, molte di queste operazioni e ti preoccupi della velocità di esecuzione del tuo codice, la [guida alle prestazioni](#) Julia ti consiglia di non farlo e preferisci i seguenti metodi:

È possibile fornire più argomenti a `print()` e `println()` che opereranno su di essi esattamente come `string()` opera su più argomenti:

```
julia> println(a, "b")  
123b
```

Oppure, quando si scrive su file, si può usare allo stesso modo, ad es

```
open("/path/to/MyFile.txt", "w") do file  
    println(file, a, "b", 13)  
end
```

o

```
file = open("/path/to/MyFile.txt", "a")  
println(file, a, "b", 13)  
close(file)
```

Questi sono più veloci perché evitano di dover prima formare una stringa da pezzi dati e poi inviarli (sia al display della console o a un file) e invece solo in uscita sequenzialmente i vari pezzi.

Crediti: risposta basata su SO Domanda [Qual è il modo migliore per convertire un Int in una stringa in Julia?](#) con risposta di Michael Ohlrogge e input di Fengyang Wang

## Interpolazione stringa (inserire il valore definito dalla variabile nella stringa)

In Julia, come in molti altri linguaggi, è possibile interpolare inserendo valori definiti da variabili in stringhe. Per un semplice esempio:

```
n = 2  
julia> MyString = "there are $n ducks"  
"there are 2 ducks"
```

Possiamo usare altri tipi oltre al numerico, ad es

```
Result = false  
julia> println("test results is $Result")  
test results is false
```

È possibile avere più interpolazioni all'interno di una determinata stringa:

```
MySubStr = "a32"
MyNum = 123.31
println("$MySubStr , $MyNum")
```

**Prestazioni Suggerimento** L' interpolazione è abbastanza comoda. Ma, se lo farai molte volte molto rapidamente, non è il più efficiente. Invece, vedi [Converti tipi numerici in stringhe](#) per suggerimenti quando la prestazione è un problema.

## Utilizzare sprint per creare stringhe con funzioni IO

Le stringhe possono essere create da funzioni che funzionano con oggetti `IO` utilizzando la funzione `sprint` . Ad esempio, la funzione `code_llvm` accetta un oggetto `IO` come primo argomento. In genere, è usato come

```
julia> code_llvm(STDOUT, *, (Int, Int))

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
    %2 = mul i64 %1, %0
    ret i64 %2
}
```

Supponiamo di volere quell'output come una stringa. Quindi possiamo semplicemente farlo

```
julia> sprint(code_llvm, *, (Int, Int))
"\ndefine i64 @\"jlsys_*_46115\"(i64, i64) #0 {\ntop:\n    %2 = mul i64 %1, %0\n    ret i64 %2\n}\n"

julia> println(ans)

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
    %2 = mul i64 %1, %0
    ret i64 %2
}
```

La conversione dei risultati di funzioni "interattive" come `code_llvm` in stringhe può essere utile per l'analisi automatica, ad esempio per [verificare](#) se il codice generato potrebbe essere regredito.

La funzione `sprint` è una [funzione di ordine superiore](#) che considera la funzione che opera su oggetti `IO` come primo argomento. Dietro le quinte, crea un `IOBuffer` nella RAM, chiama la funzione data e prende i dati dal buffer in un oggetto `String` .

**Leggi stringhe online:** <https://riptutorial.com/it/julia-lang/topic/5562/stringhe>

# Capitolo 33: sub2ind

## Sintassi

- `sub2ind (dims :: Tuple {Vararg {Integer}}, I :: Integer ...)`
- `sub2ind {T <: Integer} (dims :: Tuple {Vararg {Intero}}, I :: AbstractArray {T <: Integer, 1} ...)`

## Parametri

parametro	dettagli
<code>affievolisce :: tuple {{vararg Integer}}</code>	dimensione dell'array
<code>I :: intero ...</code>	pedici (scalari) della matrice
<code>I :: AbstractArray {T &lt;: Integer, 1} ...</code>	pedici (vettore) dell'array

## Osservazioni

Il secondo esempio mostra che il risultato di `sub2ind` potrebbe essere molto `sub2ind` in alcuni casi specifici.

## Examples

### Converti gli indici in indici lineari

```
julia> sub2ind((3,3), 1, 1)
1

julia> sub2ind((3,3), 1, 2)
4

julia> sub2ind((3,3), 2, 1)
2

julia> sub2ind((3,3), [1,1,2], [1,2,1])
3-element Array{Int64,1}:
 1
 4
 2
```

## Pits & Falls

```
# no error, even the subscript is out of range.
julia> sub2ind((3,3), 3, 4)
12
```

Non è possibile determinare se un indice si trova nell'intervallo di un array confrontando il suo indice:

```
julia> sub2ind((3,3), -1, 2)
2

julia> 0 < sub2ind((3,3), -1, 2) <= 9
true
```

Leggi `sub2ind` online: <https://riptutorial.com/it/julia-lang/topic/1914/sub2ind>

---

# Capitolo 34: Tempo

## Sintassi

- adesso()
- Dates.today ()
- Dates.year (t)
- Dates.month (t)
- Dates.day (t)
- Dates.hour (t)
- Dates.minute (t)
- Dates.second (t)
- Dates.millisecond (t)
- Date.format (t, s)

## Examples

### Ora attuale

Per ottenere la data e l'ora correnti, utilizzare la funzione `now` :

```
julia> now()
2016-09-04T00:16:58.122
```

Questa è l'ora locale, che include il fuso orario configurato della macchina. Per ottenere il tempo nel fuso orario [Coordinated Universal Time \(UTC\)](#) , utilizza `now(Dates.UTC)` :

```
julia> now(Dates.UTC)
2016-09-04T04:16:58.122
```

Per ottenere la data corrente, senza il tempo, utilizzare `today()` :

```
julia> Dates.today()
2016-10-30
```

Il valore di ritorno di `now` è un oggetto `DateTime` . Ci sono funzioni per ottenere i singoli componenti di un `DateTime` :

```
julia> t = now()
2016-09-04T00:16:58.122

julia> Dates.year(t)
2016

julia> Dates.month(t)
9
```



```
julia> Dates.day(t)
4

julia> Dates.hour(t)
0

julia> Dates.minute(t)
16

julia> Dates.second(t)
58

julia> Dates.millisecond(t)
122
```

È possibile formattare un `DateTime` usando una stringa di formato appositamente formattata:

```
julia> Dates.format(t, "yyyy-mm-dd at HH:MM:SS")
"2016-09-04 at 00:16:58"
```

Poiché molte delle funzioni `Dates` vengono esportate dal [modulo](#) `Base.Dates`, è possibile salvare alcuni `Base.Dates` digitazione per scrivere

```
using Base.Dates
```

che consente quindi l'accesso alle funzioni qualificate sopra senza le `Dates.` qualificazione.

**Leggi Tempo online:** <https://riptutorial.com/it/julia-lang/topic/5812/tempo>

# Capitolo 35: Test unitario

## Sintassi

- `@test [expr]`
- `@test_throws [Exception] [expr]`
- `@testset "[nome]" inizia; [test]; fine`
- `Pkg.test ([pacchetto])`

## Osservazioni

La documentazione della libreria standard per `Base.Test` copre materiale aggiuntivo oltre a quello mostrato in questi esempi.

## Examples

### Test di un pacchetto

Per eseguire i test unitari per un pacchetto, utilizzare la funzione `Pkg.test`. Per un pacchetto denominato `MyPackage`, il comando sarebbe

```
julia> Pkg.test("MyPackage")
```

Un risultato atteso sarebbe simile a

```
INFO: Computing test dependencies for MyPackage...
INFO: Installing BaseTestNext v0.2.2
INFO: Testing MyPackage
Test Summary: | Pass  Total
Data          |   66    66
Test Summary: | Pass  Total
Monetary      |  107   107
Test Summary: | Pass  Total
Basket        |   47    47
Test Summary: | Pass  Total
Mixed         |   13    13
Test Summary: | Pass  Total
Data Access   |   35    35
INFO: MyPackage tests passed
INFO: Removing BaseTestNext v0.2.2
```

anche se ovviamente, non ci si può aspettare che corrisponda esattamente a quanto sopra, poiché i diversi pacchetti usano quadri diversi.

Questo comando esegue il file `test/runtests.jl` del pacchetto in un ambiente pulito.

Si può testare tutti i pacchetti installati contemporaneamente con

```
julia> Pkg.test()
```

ma questo di solito richiede molto tempo.

## Scrivere un semplice test

I test unitari sono dichiarati nel file `test/runtests.jl` in un pacchetto. In genere, questo file ha inizio

```
using MyModule
using Base.Test
```

L'unità di base di test è la macro `@test`. Questa macro è come una sorta di asserzione. Qualsiasi espressione booleana può essere testata nella macro `@test`:

```
@test 1 + 1 == 2
@test iseven(10)
@test 9 < 10 || 10 < 9
```

Possiamo provare la macro `@test` nella REPL:

```
julia> using Base.Test

julia> @test 1 + 1 == 2
Test Passed
  Expression: 1 + 1 == 2
  Evaluated: 2 == 2

julia> @test 1 + 1 == 3
Test Failed
  Expression: 1 + 1 == 3
  Evaluated: 2 == 3
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

La macro di test può essere utilizzata praticamente ovunque, ad esempio nei loop o nelle funzioni:

```
# For positive integers, a number's square is at least as large as the number
for i in 1:10
    @test i^2 ≥ i
end

# Test that no two of a, b, or c share a prime factor
function check_pairwise_coprime(a, b, c)
    @test gcd(a, b) == 1
    @test gcd(a, c) == 1
    @test gcd(b, c) == 1
end

check_pairwise_coprime(10, 23, 119)
```

## Scrivere un set di prova

0.5.0

Nella versione v0.5, i set di test sono incorporati nel modulo `Base.Test` libreria standard e non devi fare nulla di speciale (oltre a `using Base.Test` ) per utilizzarli.

## 0.4.0

I set di test non fanno parte della libreria `Base.Test` di Julia v0.4. Invece, è necessario `REQUIRE` il modulo `BaseTestNext` e aggiungere `using BaseTestNext` al file. Per supportare entrambe le versioni 0.4 e 0.5, è possibile utilizzare

```
if VERSION ≥ v"0.5.0-dev+7720"
    using Base.Test
else
    using BaseTestNext
    const Test = BaseTestNext
end
```

È utile raggruppare `@test` s correlati in un set di test. Oltre a un'organizzazione di test più chiara, i set di test offrono risultati migliori e maggiore personalizzazione.

Per definire un set di test, è sufficiente racchiudere qualsiasi numero di `@test` s con un blocco `@testset` :

```
@testset "+" begin
    @test 1 + 1 == 2
    @test 2 + 2 == 4
end

@testset "*" begin
    @test 1 * 1 == 1
    @test 2 * 2 == 4
end
```

L'esecuzione di questi set di test stampa il seguente output:

```
Test Summary: | Pass  Total
+             |     2     2

Test Summary: | Pass  Total
*             |     2     2
```

Anche se un set di test contiene un test non funzionante, l'intero set di test verrà eseguito fino al completamento e i guasti verranno registrati e riportati:

```
@testset "-" begin
    @test 1 - 1 == 0
    @test 2 - 2 == 1
    @test 3 - () == 3
    @test 4 - 4 == 0
end
```

Esecuzione di questo risultato del test in

```

-: Test Failed
  Expression: 2 - 2 == 1
  Evaluated: 0 == 1
  in record(::Base.Test.DefaultTestSet, ::Base.Test.Fail) at ./test.jl:428
  ...
-: Error During Test
  Test threw an exception of type MethodError
  Expression: 3 - () == 3
  MethodError: no method matching -(::Int64, ::Tuple{})
  ...
Test Summary: | Pass  Fail  Error  Total
-             |    2    1      1      4
ERROR: Some tests did not pass: 2 passed, 1 failed, 1 errored, 0 broken.
  ...

```

I set di test possono essere annidati, consentendo un'organizzazione arbitrariamente profonda

```

@testset "Int" begin
  @testset "+" begin
    @test 1 + 1 == 2
    @test 2 + 2 == 4
  end
  @testset "-" begin
    @test 1 - 1 == 0
  end
end
end

```

Se i test superano, questo mostrerà solo i risultati per il set di test più esterno:

```

Test Summary: | Pass  Total
Int           |    3      3

```

Ma se i test falliscono, viene riportato un drill-down nell'esatta serie di test e test che ha causato l'errore.

La macro `@testset` può essere utilizzata con un [ciclo for](#) per creare molti set di test contemporaneamente:

```

@testset for i in 1:5
  @test 2i == i + i
  @test i^2 == i * i
  @test i ÷ i == 1
end

```

quali rapporti

```

Test Summary: | Pass  Total
i = 1         |    3      3
Test Summary: | Pass  Total
i = 2         |    3      3
Test Summary: | Pass  Total
i = 3         |    3      3
Test Summary: | Pass  Total
i = 4         |    3      3
Test Summary: | Pass  Total

```

```
i = 5      |      3      3
```

Una struttura comune è quella di avere test o gruppi di test esterni. All'interno di questi set di test esterni, il test interno stabilisce il comportamento di test. Ad esempio, supponiamo di aver creato un tipo `UniversalSet` con un'istanza singleton che contiene tutto. Prima ancora di implementare il tipo, possiamo utilizzare i principi di [sviluppo basati sui test](#) e implementare i test:

```
@testset "UniversalSet" begin
  U = UniversalSet.instance
  @testset "egal/equal" begin
    @test U === U
    @test U == U
  end

  @testset "in" begin
    @test 1 in U
    @test "Hello World" in U
    @test Int in U
    @test U in U
  end

  @testset "subset" begin
    @test Set() ⊆ U
    @test Set(["Hello World"]) ⊆ U
    @test Set(1:10) ⊆ U
    @test Set([:a, 2.0, "w", Set()]) ⊆ U
    @test U ⊆ U
  end
end
```

Possiamo quindi iniziare a implementare le nostre funzionalità fino a quando non supererà i nostri test. Il primo passo è definire il tipo:

```
immutable UniversalSet <: Base.AbstractSet end
```

Solo due dei nostri test passano adesso. Possiamo implementare `in` :

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
```

Ciò rende anche alcuni dei nostri test di sottoinsieme. Tuttavia, il `issubset` (`⊆`) non funziona per `UniversalSet`, perché il fallback tenta di eseguire iterazioni su elementi, cosa che non possiamo fare. Possiamo semplicemente definire una specializzazione che rende `issubset` tornare `true` per ogni insieme:

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
Base.issubset(x::Base.AbstractSet, ::UniversalSet) = true
```

E ora passano tutti i nostri test!

## Test delle eccezioni

Le eccezioni incontrate durante l'esecuzione di un test non supereranno il test e, se il test non si trova in un set di test, terminerà il test engine. Di solito, questa è una buona cosa, perché nella maggior parte delle situazioni le eccezioni non sono il risultato desiderato. Ma a volte, si vuole testare in modo specifico che venga sollevata una certa eccezione. La macro `@test_throws` facilita questo.

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
  Expression: ([1,2,3])[4]
    Thrown: BoundsError
```

Se viene generata l'eccezione sbagliata, `@test_throws` fallirà ancora:

```
julia> @test_throws TypeError [1, 2, 3][4]
Test Failed
  Expression: ([1,2,3])[4]
    Expected: TypeError
    Thrown: BoundsError
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test_throws(::Base.Test.Threw, ::Expr, ::Type{T}) at ./test.jl:329
```

e se non viene lanciata alcuna eccezione, `@test_throws` fallirà anche:

```
julia> @test_throws BoundsError [1, 2, 3, 4][4]
Test Failed
  Expression: ([1,2,3,4])[4]
    Expected: BoundsError
    No exception thrown
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test_throws(::Base.Test.Returned, ::Expr, ::Type{T}) at ./test.jl:329
```

## Testing Equality approssimativo a virgola mobile

Qual è l'accordo con quanto segue?

```
julia> @test 0.1 + 0.2 == 0.3
Test Failed
  Expression: 0.1 + 0.2 == 0.3
    Evaluated: 0.30000000000000004 == 0.3
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

L'errore è causato dal fatto che nessuno di `0.1`, `0.2` e `0.3` è rappresentato nel computer esattamente come quei valori -  $1/10$ ,  $2/10$  e  $3/10$ . Invece, sono approssimati da valori molto vicini. Ma come visto nel fallimento del test sopra, quando si aggiungono due approssimazioni insieme, il risultato può essere un'approssimazione leggermente peggiore di quanto sia possibile. C'è [molto di più in questo argomento](#) che non può essere coperto qui.

Ma non siamo sfortunati! Per verificare che la combinazione di arrotondamento a un numero in

virgola mobile e aritmetica in virgola mobile sia *approssimativamente* corretta, anche se non esatta, possiamo usare la funzione `isapprox` (che corrisponde all'operatore  $\approx$ ). Quindi possiamo riscrivere il nostro test come

```
julia> @test 0.1 + 0.2 ≈ 0.3
Test Passed
  Expression: 0.1 + 0.2 ≈ 0.3
  Evaluated: 0.30000000000000004 isapprox 0.3
```

Ovviamente, se il nostro codice fosse completamente sbagliato, il test continuerà a rilevare che:

```
julia> @test 0.1 + 0.2 ≈ 0.4
Test Failed
  Expression: 0.1 + 0.2 ≈ 0.4
  Evaluated: 0.30000000000000004 isapprox 0.4
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

La funzione `isapprox` utilizza l'euristica in base alla dimensione dei numeri e alla precisione del tipo a virgola mobile per determinare la quantità di errore da tollerare. Non è appropriato per tutte le situazioni, ma funziona in gran parte, e fa un sacco di sforzi per implementare la propria versione di `isapprox`.

Leggi Test unitario online: <https://riptutorial.com/it/julia-lang/topic/5632/test-unitario>



# Capitolo 36: tipi

## Sintassi

- MyType immutabile; campo; campo; fine
- digitare MyType; campo; campo; fine

## Osservazioni

I tipi sono fondamentali per la performance di Julia. Un'idea importante per le prestazioni è la [stabilità del tipo](#), che si verifica quando il tipo restituito da una funzione dipende solo dai tipi, non dai valori, dei suoi argomenti.

## Examples

### Dispacciamento su Tipi

Su Julia, puoi definire più di un metodo per ogni funzione. Supponiamo di definire tre metodi con la stessa funzione:

```
foo(x) = 1
foo(x::Number) = 2
foo(x::Int) = 3
```

Al momento di decidere quale metodo usare (chiamato [dispatch](#)), Julia sceglie il metodo più specifico che corrisponde ai tipi degli argomenti:

```
julia> foo('one')
1

julia> foo(1.0)
2

julia> foo(1)
3
```

Questo facilita il [polimorfismo](#). Ad esempio, possiamo facilmente creare un [elenco collegato](#) definendo due tipi immutabili, denominati `Nil` e `Cons`. Questi nomi sono tradizionalmente usati per descrivere rispettivamente una lista vuota e una lista non vuota.

```
abstract LinkedList
immutable Nil <: LinkedList end
immutable Cons <: LinkedList
    first
    rest::LinkedList
end
```

Rappresenteremo la lista vuota di `Nil()` e qualsiasi altra lista di `Cons(first, rest)`, dove `first` è il primo elemento dell'elenco collegato e `rest` è l'elenco collegato costituito da tutti gli elementi rimanenti. Ad esempio, la lista `[1, 2, 3]` sarà rappresentata come

```
julia> Cons(1, Cons(2, Cons(3, Nil())))  
Cons{1, Cons{2, Cons{3, Nil()}}}
```

## L'elenco è vuoto?

Supponiamo di voler estendere la funzione `isempty` della libreria standard, che funziona su una varietà di collezioni diverse:

```
julia> methods(isempty)  
# 29 methods for generic function "isempty":  
isempty(v::SimpleVector) at essentials.jl:180  
isempty(m::Base.MethodList) at reflection.jl:394  
...
```

Possiamo semplicemente usare la sintassi dispatch della funzione e definire due metodi aggiuntivi di `isempty`. Poiché questa funzione è dal modulo `Base`, dobbiamo qualificarla come `Base.isempty` per estenderla.

```
Base.isempty{::Nil} = true  
Base.isempty{::Cons} = false
```

Qui, non abbiamo affatto bisogno dei valori degli argomenti per determinare se l'elenco è vuoto. Solo il tipo da solo è sufficiente per calcolare tali informazioni. Julia ci consente di omettere i nomi degli argomenti, mantenendo solo la loro annotazione del tipo, se non è necessario utilizzare i loro valori.

Possiamo [verificare](#) che i nostri metodi `isempty` funzionino:

```
julia> using Base.Test  
  
julia> @test isempty(Nil())  
Test Passed  
Expression: isempty(Nil())  
  
julia> @test !isempty(Cons(1, Cons(2, Cons(3, Nil()))))  
Test Passed  
Expression: !(isempty(Cons(1, Cons(2, Cons(3, Nil())))))
```

e in effetti il numero di metodi per `isempty` è aumentato di 2 :

```
julia> methods(isempty)  
# 31 methods for generic function "isempty":  
isempty(v::SimpleVector) at essentials.jl:180  
isempty(m::Base.MethodList) at reflection.jl:394
```

Chiaramente, determinare se un elenco collegato è vuoto o meno è un esempio banale. Ma porta

a qualcosa di più interessante:

## Quanto dura la lista?

La funzione di `length` della libreria standard ci fornisce la lunghezza di una raccolta o di determinati [iterabili](#). Esistono molti modi per implementare la `length` per un elenco collegato. In particolare, usare un ciclo `while` è probabilmente il più veloce e più efficiente in termini di memoria in Julia. Ma [l'ottimizzazione prematura](#) deve essere evitata, quindi supponiamo per un secondo che il nostro elenco collegato non sia efficiente. Qual è il modo più semplice per scrivere una funzione di `length`?

```
Base.length(::Nil) = 0
Base.length(xs::Cons) = 1 + length(xs.rest)
```

La prima definizione è semplice: una lista vuota ha lunghezza 0. Anche la seconda definizione è facile da leggere: per contare la lunghezza di una lista, contiamo il primo elemento, quindi contiamo la lunghezza del resto della lista. Possiamo testare questo metodo in modo simile al modo in cui abbiamo provato l' `isempty`:

```
julia> @test length(Nil()) == 0
Test Passed
Expression: length(Nil()) == 0
Evaluated: 0 == 0

julia> @test length(Cons(1, Cons(2, Cons(3, Nil())))) == 3
Test Passed
Expression: length(Cons(1, Cons(2, Cons(3, Nil())))) == 3
Evaluated: 3 == 3
```

## Prossimi passi

Questo esempio di giocattolo è abbastanza lontano dall'implementazione di tutte le funzionalità che sarebbero desiderate in una lista collegata. Manca, per esempio, l'interfaccia di iterazione. Tuttavia, illustra come la spedizione può essere utilizzata per scrivere codice breve e chiaro.

### Tipi immutabili

Il tipo di composito più semplice è di tipo immutabile. Le istanze di tipi immutabili, come le [tuple](#), sono valori. I loro campi non possono essere modificati dopo la loro creazione. In molti modi, un tipo immutabile è come una `Tuple` con nomi per il tipo stesso e per ogni campo.

## Tipi Singleton

I tipi di composito, per definizione, contengono un numero di tipi più semplici. In Julia, questo numero può essere zero; vale a dire, un tipo immutabile, è permesso di *contenere* i campi. Questo è paragonabile alla tupla vuota `()`.

Perché potrebbe essere utile? Tali tipi immutabili sono noti come "tipi singleton", in quanto solo una loro istanza potrebbe mai esistere. I valori di tali tipi sono noti come "valori singleton". La libreria standard `Base` contiene molti tali tipi Singleton. Ecco una breve lista:

- `Void` , il tipo di `nothing` . Possiamo verificare che `Void.instance` (che è una sintassi speciale per il recupero del valore singleton di un tipo singleton) non è davvero `nothing` .
- Qualsiasi tipo di supporto, come `MIME"text/plain"` , è un tipo singleton con una sola istanza, `MIME("text/plain")` .
- L' `Irrational{: $\pi$ }` , `Irrational{: $e$ }` , `Irrational{: $\varphi$ }` , e tipi simili sono tipi singleton, e le loro istanze Singleton sono i valori irrazionali  $\pi = 3.1415926535897\dots$  , ecc.
- I tratti della dimensione iteratore `Base.HasLength` , `Base.HasShape` , `Base.IsInfinite` e `Base.SizeUnknown` sono tutti tipi di singleton.

### 0.5.0

- Nella versione 0.5 e successive, ogni **funzione** è un'istanza singleton di un tipo singleton! Come ogni altro valore singleton, possiamo recuperare la funzione `sin` , ad esempio, da `typeof(sin).instance` .

Poiché non contengono nulla, i tipi di singleton sono incredibilmente leggeri e possono essere spesso ottimizzati dal compilatore per non avere sovraccarico di runtime. Pertanto, sono perfetti per i tratti, i valori speciali dei tag e per funzioni come le funzioni su cui ci si vuole specializzare.

Per definire un tipo singleton,

```
julia> immutable MySingleton end
```

Per definire la stampa personalizzata per il tipo singleton,

```
julia> Base.show(io::IO, ::MySingleton) = print(io, "sing")
```

Per accedere all'istanza singleton,

```
julia> MySingleton.instance
MySingleton()
```

Spesso, si assegna questo a una costante:

```
julia> const sing = MySingleton.instance
MySingleton()
```

## Tipi di wrapper

Se i tipi immutabili a campo zero sono interessanti e utili, allora forse i tipi immutabili a un campo sono ancora più utili. Tali tipi vengono comunemente chiamati "tipi di wrapper" perché racchiudono alcuni dati sottostanti, fornendo un'interfaccia alternativa a tali dati. Un esempio di un tipo di wrapper in `Base` è `String` . Definiremo un tipo simile a `String` , chiamato `MyString` . Questo

tipo sarà supportato da un vettore ( [array](#) monodimensionale) di byte ( `UInt8` ).

Innanzitutto, la definizione del tipo stesso e alcuni risultati personalizzati:

```
immutable MyString <: AbstractString
  data::Vector{UInt8}
end

function Base.show(io::IO, s::MyString)
  print(io, "MyString: ")
  write(io, s.data)
  return
end
```

Ora il nostro tipo di `MyString` è pronto per l'uso! Possiamo alimentarlo con dati UTF-8 grezzi e viene visualizzato come ci piace:

```
julia> MyString([0x48,0x65,0x6c,0x6c,0x6f,0x2c,0x20,0x57,0x6f,0x72,0x6c,0x64,0x21])
MyString: Hello, World!
```

Ovviamente, questo tipo di stringa richiede molto lavoro prima che diventi utilizzabile come il tipo `Base.String`.

## Veri tipi di composito

Forse più comunemente, molti tipi immutabili contengono più di un campo. Un esempio è la libreria standard `Rational{T}` type, che contiene due fields: un campo `num` per il numeratore e un campo `den` per il denominatore. È abbastanza semplice emulare questo tipo di progettazione:

```
immutable MyRational{T}
  num::T
  den::T
  MyRational(n, d) = (g = gcd(n, d); new(n÷g, d÷g))
end
MyRational{T}(n::T, d::T) = MyRational{T}(n, d)
```

Abbiamo implementato con successo un costruttore che semplifica i nostri numeri razionali:

```
julia> MyRational(10, 6)
MyRational{Int64}(5, 3)
```

Leggi tipi online: <https://riptutorial.com/it/julia-lang/topic/5467/tipi>

# Capitolo 37: Tipo di stabilità

## introduzione

L'**instabilità del tipo** si verifica quando il **tipo di** una variabile può cambiare in fase di esecuzione e quindi non può essere dedotto in fase di compilazione. L'instabilità del tipo spesso causa problemi di prestazioni, quindi è importante poter scrivere e identificare il codice stabile al tipo.

## Examples

### Scrivi un codice stabile al tipo

```
function sumofsins1(n::Integer)
    r = 0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end

function sumofsins2(n::Integer)
    r = 0.0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end
```

La tempistica delle due funzioni di cui sopra mostra le principali differenze in termini di tempo e allocazioni di memoria.

```
julia> @time [sumofsins1(100_000) for i in 1:100];
0.638923 seconds (30.12 M allocations: 463.094 MB, 10.22% gc time)

julia> @time [sumofsins2(100_000) for i in 1:100];
0.163931 seconds (13.60 k allocations: 611.350 KB)
```

Questo è dovuto al codice **type-unstable** in `sumofsins1` cui il tipo di `r` deve essere controllato per ogni iterazione.

Leggi **Tipo di stabilità online**: <https://riptutorial.com/it/julia-lang/topic/6084/tipo-di-stabilita>

# Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Julia Language	<a href="#">Andrew Piliser</a> , <a href="#">becko</a> , <a href="#">Community</a> , <a href="#">Dawny33</a> , <a href="#">Fengyang Wang</a> , <a href="#">Kevin Montrose</a> , <a href="#">prcastro</a>
2	@goto e @label	<a href="#">Fengyang Wang</a>
3	Aritmetica	<a href="#">Fengyang Wang</a>
4	Array	<a href="#">Fengyang Wang</a> , <a href="#">Michael Ohlrogge</a> , <a href="#">prcastro</a>
5	chiusure	<a href="#">Fengyang Wang</a>
6	combinatori	<a href="#">Fengyang Wang</a>
7	Compatibilità tra versioni	<a href="#">Fengyang Wang</a>
8	Comprensioni	<a href="#">2Cubed</a> , <a href="#">Fengyang Wang</a> , <a href="#">zwlayer</a>
9	Condizionali	<a href="#">Fengyang Wang</a> , <a href="#">Michael Ohlrogge</a> , <a href="#">prcastro</a>
10	confronti	<a href="#">Fengyang Wang</a>
11	dizionari	<a href="#">B Roy Dawson</a>
12	Elaborazione parallela	<a href="#">Fengyang Wang</a> , <a href="#">Harrison Grodin</a> , <a href="#">Michael Ohlrogge</a> , <a href="#">prcastro</a>
13	Enums	<a href="#">Fengyang Wang</a>
14	espressioni	<a href="#">Michael Ohlrogge</a>
15	funzioni	<a href="#">Fengyang Wang</a> , <a href="#">Harrison Grodin</a> , <a href="#">Michael Ohlrogge</a> , <a href="#">Sebastialonso</a>
16	Funzioni di ordine superiore	<a href="#">Fengyang Wang</a> , <a href="#">mnoronha</a>
17	Ingresso	<a href="#">Fengyang Wang</a>
18	iterabili	<a href="#">Fengyang Wang</a> , <a href="#">prcastro</a>
19	JSON	<a href="#">4444</a> , <a href="#">Fengyang Wang</a>
20	Le tuple	<a href="#">Fengyang Wang</a>

21	Lettura di un DataFrame da un file	<a href="#">Pranav Bhat</a>
22	Macro di stringa	<a href="#">Fengyang Wang</a>
23	mentre cicli	<a href="#">Fengyang Wang</a>
24	metaprogrammazione	<a href="#">Fengyang Wang</a> , <a href="#">Ismael Venegas Castelló</a> , <a href="#">P i</a> , <a href="#">prcastro</a>
25	moduli	<a href="#">Fengyang Wang</a>
26	Normalizzazione delle stringhe	<a href="#">Fengyang Wang</a>
27	Pacchi	<a href="#">Fengyang Wang</a>
28	per loop	<a href="#">Fengyang Wang</a> , <a href="#">Michael Ohlrogge</a>
29	regex	<a href="#">Fengyang Wang</a>
30	REPL	<a href="#">Fengyang Wang</a>
31	Scripting Shell e Piping	<a href="#">2Cubed</a> , <a href="#">Fengyang Wang</a> , <a href="#">mnoronha</a> , <a href="#">prcastro</a>
32	stringhe	<a href="#">Fengyang Wang</a> , <a href="#">Michael Ohlrogge</a>
33	sub2ind	<a href="#">Fengyang Wang</a> , <a href="#">Gnimuc</a>
34	Tempo	<a href="#">Fengyang Wang</a>
35	Test unitario	<a href="#">Fengyang Wang</a>
36	tipi	<a href="#">Fengyang Wang</a> , <a href="#">prcastro</a>
37	Tipo di stabilità	<a href="#">Abhijith</a> , <a href="#">Fengyang Wang</a>