EBook Gratis

APRENDIZAJE

Julia Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#julia-lang

Tabla de contenido

Acerca de

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.

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

Capítulo 1: Empezando con Julia Language

Versiones

Versión	Fecha de lanzamiento
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

¡Hola Mundo!

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

Para ejecutar a Julia, primero obtenga el intérprete de la página de descarga del sitio web . La versión estable actual es v0.5.0, y esta versión se recomienda para la mayoría de los usuarios. Ciertos desarrolladores de paquetes o usuarios avanzados pueden optar por utilizar la compilación nocturna, que es mucho menos estable.

Cuando tenga el intérprete, escriba su programa en un archivo llamado `hello.jl` . Entonces se puede ejecutar desde un terminal del sistema como:

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

Julia también se puede ejecutar de forma interactiva, ejecutando el programa `julia` . Debería ver un encabezado y un mensaje, de la siguiente manera:

```
               _
   _       _ _(_)_     |  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>
```

Puedes ejecutar cualquier código de Julia en este REPL , así que prueba:

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

Este ejemplo hace uso de una cadena , `"Hello, World!"` y de la función `println` , una de muchas en la biblioteca estándar. Para obtener más información o ayuda, pruebe las siguientes fuentes:

- El REPL tiene un modo de ayuda integrado para acceder a la documentación.
- La documentación oficial es bastante completa.
- Stack Overflow tiene una pequeña pero creciente colección de ejemplos.
- Los usuarios de Gitter están felices de ayudar con pequeñas preguntas.
- El principal foro de discusión en línea para Julia es el foro del Discurso en discourse.julialang.org . Las preguntas más involucradas deben publicarse aquí.
- Una colección de tutoriales y libros se puede encontrar aquí .

Lea Empezando con Julia Language en línea: https://riptutorial.com/es/julia-lang/topic/485/empezando-con-julia-language

Capítulo 2: @goto y @label

Sintaxis

- etiqueta @goto
- etiqueta @label

Observaciones

El uso excesivo o inadecuado del flujo de control avanzado hace que el código sea difícil de leer. `@goto` o sus equivalentes en otros idiomas, cuando se usan incorrectamente, conducen a un código de espagueti ilegible.

Similar a lenguajes como C, uno no puede saltar entre funciones en Julia. Esto también significa que `@goto` no es posible en el nivel superior; Sólo funcionará dentro de una función. Además, uno no puede saltar de una función interna a su función externa, o de una función externa a una función interna.

Examples

Validación de entrada

Aunque tradicionalmente no se consideran los bucles, las macros `@goto` y `@label` se pueden usar para un flujo de control más avanzado. Un caso de uso es cuando el fallo de una parte debería llevar a reintentar una función completa, a menudo útil en la validación de entrada:

```
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
```

Sin embargo, este caso de uso es a menudo más claro usando recursión:

```
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
```

Aunque ambos ejemplos hacen lo mismo, el segundo es más fácil de entender. Sin embargo, el primero es más eficaz (porque evita la llamada recursiva). En la mayoría de los casos, el costo de la llamada no importa; pero en situaciones limitadas, la primera forma es aceptable.

Error de limpieza

En lenguajes como C, la instrucción `@goto` se usa a menudo para garantizar que una función limpie los recursos necesarios, incluso en el caso de un error. Esto es menos importante en Julia, porque las excepciones y `try` - `finally` bloques se utilizan a menudo en su lugar.

Sin embargo, es posible que el código Julia se interconecte con el código C y las API de C, por lo que a veces las funciones aún deben escribirse como código C. El siguiente ejemplo está diseñado, pero demuestra un caso de uso común. El código de Julia llamará a `Libc.malloc` para asignar algo de memoria (esto simula una llamada de la API C). Si no todas las asignaciones tienen éxito, entonces la función debería liberar los recursos obtenidos hasta el momento; de lo contrario, se devuelve la memoria asignada.

```
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
```

Lea @goto y @label en línea: https://riptutorial.com/es/julia-lang/topic/5564/-goto-y--label

Capítulo 3: Aritmética

Sintaxis

- + x
- -X
- a + b
- a - b
- a * b
- a / b
- a ^ b
- a% b
- 4a
- sqrt (a)

Examples

Fórmula cuadrática

Julia usa operadores binarios similares para operaciones aritméticas básicas, al igual que las matemáticas u otros lenguajes de programación. La mayoría de los operadores pueden escribirse en notación infija (es decir, colocados entre los valores que se están calculando). Julia tiene un orden de operaciones que coincide con la convención común en matemáticas.

Por ejemplo, el siguiente código implementa la fórmula cuadrática , que demuestra los operadores `+` , `-` , `*` y `/` para la suma, resta, multiplicación y división, respectivamente. También se muestra la *multiplicación implícita* , donde un número se puede colocar directamente antes de un símbolo para significar multiplicación; es decir, `4a` significa lo mismo que `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)
```

Tamiz de Eratóstenes

El operador restante en Julia es el operador `%` . Este operador se comporta de manera similar al `%` en idiomas como C y C ++. `a % b` es el resto firmado después de dividir `a` por `b` .

Este operador es muy útil para implementar ciertos algoritmos, como la siguiente implementación

del Tamiz de Eratóstenes .

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

function sieve(n)
    P = Int[]
    for i in 2:n
        if iscoprime(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
```

Aritmética matricial

Julia usa los significados matemáticos estándar de las operaciones aritméticas cuando se aplica a matrices. A veces, las operaciones elementales son deseadas en su lugar. Estos están marcados con una parada completa (.) Que precede al operador para que se realice de forma elemental. (Tenga en cuenta que las operaciones elementales a menudo no son tan eficientes como los bucles).

Sumas

El operador `+` en las matrices es una suma de matrices. Es similar a una suma de elementos, pero no transmite forma. Es decir, si `A` y `B` tienen la misma forma, entonces `A + B` es lo mismo que `A .+ B` ; de lo contrario, `A + B` es un error, mientras que `A .+ B` puede no serlo necesariamente.

```
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
```

Asimismo, `-` calcula una diferencia de matriz. Ambos `+` y `-` también se pueden utilizar como operadores unarios.

Productos

El operador `*` en matrices es el producto matricial (no el producto elementwise). Para un producto elementwise, use el operador. `.*` . Compara (usando las mismas matrices que arriba):

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

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

Potestades

El operador `^` calcula la exponenciación de la matriz . La exponenciación de matrices puede ser útil para calcular valores de ciertas recurrencias rápidamente. Por ejemplo, los números de Fibonacci pueden ser generados por la expresión matricial

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

Como de costumbre, el operador `.^` Se puede utilizar donde la operación deseada es la exponenciación elemental.

Lea Aritmética en línea: https://riptutorial.com/es/julia-lang/topic/3848/aritmetica

Capítulo 4: Arrays

Sintaxis

- [1,2,3]
- [1 2 3]
- [1 2 3; 4 5 6; 7 8 9]
- Array (tipo, dims ...)
- unos (tipo, dims ...)
- ceros (tipo, dims ...)
- Trues (tipo, dims ...)
- falsas (tipo, dims ...)
- empujar! (A, x)
- pop! (A)
- Unshift! (A, x)
- cambio! (A)

Parámetros

Parámetros	Observaciones
por	`push!(A, x)` , sin `unshift!(A, x)`
`A`	La matriz para agregar a.
`x`	El elemento a agregar a la matriz.

Examples

Construcción manual de una matriz simple.

Uno puede inicializar una matriz de Julia a mano, utilizando la sintaxis de los corchetes:

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

La primera línea después del comando muestra el tamaño de la matriz que creó. También muestra el tipo de sus elementos y su dimensionalidad (int en este caso `Int64` y `1` , respectivamente). Para una matriz bidimensional, puede usar espacios y punto y coma:

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

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

Para crear una matriz sin inicializar, puede usar el método `Array(type, dims...)` :

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

Las funciones `zeros` , `ones` , `trues` , `falses` tienen métodos que se comportan exactamente de la misma manera, pero producen matrices llenas de `0.0` , `1.0` , `True` o `False` , respectivamente.

Tipos de matrices

En Julia, los Arrays tienen tipos parametrizados por dos variables: un tipo `T` y una dimensionalidad `D` (`Array{T, D}`). Para una matriz unidimensional de enteros, el tipo es:

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

Si la matriz es una matriz bidimensional, `D` es igual a 2:

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

El tipo de elemento también puede ser tipos abstractos:

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

Aquí `Any` (un tipo abstracto) es el tipo de la matriz resultante.

Especificar tipos al crear matrices

Cuando creamos un Array de la manera descrita anteriormente, Julia hará todo lo posible para inferir el tipo adecuado que podamos desear. En los ejemplos iniciales anteriores, `Int64` entradas que parecían enteros, por lo que Julia se estableció de manera predeterminada en el tipo `Int64` predeterminado. A veces, sin embargo, podríamos ser más específicos. En el siguiente ejemplo, especificamos que queremos que el tipo sea `Int8` en su lugar:

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

Incluso podríamos especificar el tipo como algo como `Float64` , incluso si escribimos las entradas de una manera que podría interpretarse como números enteros de forma predeterminada (por ejemplo, escribir `1` lugar de `1.0`). p.ej

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

Arreglos de Arrays - Propiedades y Construcción

En Julia, puede tener una matriz que contiene otros objetos de tipo matriz. Considere los siguientes ejemplos de inicialización de varios tipos de Arrays:

```
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
```

Considere, por ejemplo, las diferencias entre C y D aquí:

```
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 objeto de tipo `Array{Float64,1}` . Dado que la única especificación para los elementos de `C` es que son matrices con elementos de tipo Float64, esto se ajusta a la definición de `C` Pero, para `D` especificamos que los elementos deben ser matrices bidimensionales. Por lo tanto, dado que `rand(3)` no produce una matriz bidimensional, no podemos usarla para asignar un valor a un elemento específico de `D`

Especificar dimensiones específicas de matrices dentro de una matriz

Aunque podemos especificar que un Array tendrá elementos que son de tipo Array, y podemos especificar que, por ejemplo, esos elementos deben ser Arrays bidimensionales, no podemos especificar directamente las dimensiones de esos elementos. Por ejemplo, no podemos especificar directamente que queremos una matriz que contenga 10 matrices, cada una de las cuales es 5,5. Podemos ver esto en la sintaxis de la función `Array()` utilizada para construir un Array:

> **Array {T} (dims)**
>
> construye una matriz densa sin inicializar con el tipo de elemento T. dims puede ser

una tupla o una serie de argumentos enteros. La sintaxis Array (T, dims) también está disponible, pero está obsoleta.

El tipo de una matriz en Julia abarca el número de las dimensiones pero no el tamaño de esas dimensiones. Por lo tanto, no hay lugar en esta sintaxis para especificar las dimensiones precisas. Sin embargo, se podría lograr un efecto similar utilizando una comprensión de Array:

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

Nota: esta documentación refleja la siguiente respuesta SO

Inicializar una matriz vacía

Podemos usar `[]` para crear una matriz vacía en Julia. El ejemplo más simple sería:

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

Las matrices de tipo `Any` generalmente no funcionarán tan bien como aquellas con un tipo específico. Así, por ejemplo, podemos usar:

```
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}
```

Consulte Inicializar una matriz vacía de tuplas en Julia para ver la fuente del último ejemplo.

Vectores

Los vectores son matrices unidimensionales, y soportan principalmente la misma interfaz que sus homólogos multidimensionales. Sin embargo, los vectores también soportan operaciones adicionales.

Primero, tenga en cuenta que el `Vector{T}` donde `T` es algún tipo significa lo mismo que el `Array{T,1}` .

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

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

Uno lee `Array{Int64,1}` como "matriz unidimensional de `Int64` ".

A diferencia de las matrices multidimensionales, los vectores se pueden redimensionar. Los elementos se pueden agregar o eliminar desde la parte frontal o posterior del vector. Estas operaciones son todas de tiempo amortizado constante .

```
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
```

Como es convención, cada una de estas funciones `push!` , `pop!` , sin `unshift!` , y `shift!` termina en un signo de exclamación para indicar que están mutando su argumento. Las funciones `push!` y sin `unshift!` devuelve la matriz, mientras que `pop!` y `shift!` Devuelve el elemento eliminado.

Concatenación

A menudo es útil construir matrices a partir de matrices más pequeñas.

Concatenacion horizontal

Las matrices (y los vectores, que se tratan como vectores de columna) se pueden concatenar horizontalmente mediante la función `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
```

Existe una sintaxis de conveniencia disponible, utilizando la notación entre corchetes y los espacios:

```
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
```

Esta notación puede coincidir estrechamente con la notación para matrices de bloques utilizadas en álgebra lineal:

```
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
```

Tenga en cuenta que no puede concatenar horizontalmente una sola matriz utilizando la sintaxis `[]` , ya que en su lugar crearía un vector de un elemento de matrices:

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

Concatenacion vertical

La concatenación vertical es como la concatenación horizontal, pero en la dirección vertical. La función para la concatenación vertical es `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
```

Alternativamente, la notación de corchete se puede utilizar con punto ; coma ; como el delimitador

```
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
```

Los vectores también se pueden concatenar verticalmente; el resultado es un vector:

```
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 concatenación horizontal y vertical se puede combinar:

```
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
```

Lea Arrays en línea: https://riptutorial.com/es/julia-lang/topic/5437/arrays

Capítulo 5: Cierres

Sintaxis

- x -> [cuerpo]
- (x, y) -> [cuerpo]
- (xs ...) -> [cuerpo]

Observaciones

0.4.0

En versiones anteriores de Julia, los cierres y las funciones anónimas tenían una penalización de rendimiento en tiempo de ejecución. Esta penalización ha sido eliminada en 0.5.

Examples

Composición de funciones

Podemos definir una función para realizar la composición de la función utilizando la sintaxis de la función anónima :

```
f ∘ g = x -> f(g(x))
```

Tenga en cuenta que esta definición es equivalente a cada una de las siguientes definiciones:

```
∘(f, g) = x -> f(g(x))
```

o

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

recordando que en Julia, `f ∘ g` es simplemente sintaxis de azúcar para `∘(f, g)` .

Podemos ver que esta función se compone correctamente:

```
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

En la versión v0.5, esta definición es muy eficaz. Podemos mirar en el código LLVM generado:

```
julia> @code_llvm sextuple(1)

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

Está claro que las dos multiplicaciones se han plegado en una sola multiplicación, y que esta función es lo más eficiente posible.

¿Cómo funciona esta función de orden superior? Crea un llamado cierre , que consiste no solo en su código, sino que también realiza un seguimiento de ciertas variables desde su alcance. Todas las funciones en Julia que no se crean en el ámbito de nivel superior son cierres.

0.5.0

Uno puede inspeccionar las variables cerradas a través de los campos del cierre. Por ejemplo, vemos que:

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

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

Implementando Currying

Una aplicación de los cierres es aplicar parcialmente una función; es decir, proporcione algunos argumentos ahora y cree una función que tome los argumentos restantes. Currying es una forma específica de aplicación parcial.

Comencemos con la función simple `curry(f, x)` que proporcionará el primer argumento a una función, y esperamos argumentos adicionales más adelante. La definición es bastante sencilla:

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

Una vez más, usamos la sintaxis de la función anónima , esta vez en combinación con la sintaxis del argumento variadic.

Podemos implementar algunas funciones básicas en estilo tácito (o sin puntos) usando esta función de `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?
```

Las funciones mantienen el generismo esperado:

```
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
```

Introducción a los cierres

Las funciones son una parte importante de la programación de Julia. Se pueden definir directamente dentro de los módulos, en cuyo caso las funciones se denominan *de nivel superior* . Pero las funciones también se pueden definir dentro de otras funciones. Tales funciones se llaman " cierres ".

Los cierres capturan las variables en su función exterior. Una función de nivel superior solo puede usar variables globales de su módulo, parámetros de función o variables locales:

```
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
```

Un cierre, por otro lado, puede usar todos aquellos además de las variables de las funciones externas que captura:

```
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
```

Si ejecutamos `c = toplevel(10)` , vemos que el resultado es

```
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)
```

Tenga en cuenta que la expresión de cola de esta función es una función en sí misma; Es decir, un cierre. Podemos llamar al cierre `c` como si fuera cualquier otra función:

```
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)
```

Tenga en cuenta que `c` todavía tiene acceso a las variables `y` y `z` desde la llamada de nivel `toplevel` , ¡aunque el nivel `toplevel` ya ha regresado! Cada cierre, incluso los devueltos por la misma función, se cierra sobre diferentes variables. Podemos llamar al nivel `toplevel` nuevo.

```
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)
```

Tenga en cuenta que a pesar de que `d` y `c` tienen el mismo código y de haber pasado los mismos argumentos, su salida es diferente. Son distintos cierres.

Lea Cierres en línea: https://riptutorial.com/es/julia-lang/topic/5724/cierres

Capítulo 6: Combinadores

Observaciones

Si bien los combinadores tienen un uso práctico limitado, son una herramienta útil en educación para comprender cómo la programación está fundamentalmente vinculada a la lógica y cómo los bloques de construcción muy simples se pueden combinar para crear un comportamiento muy complejo. En el contexto de Julia, aprender a crear y usar combinadores fortalecerá la comprensión de cómo programar en un estilo funcional en Julia.

Examples

El combinador Y o Z

Aunque Julia no es un lenguaje puramente funcional, que tiene soporte completo para muchas de las piedras angulares de la programación funcional: primera clase de funciones , ámbito léxico, y cierres .

El combinador de punto fijo es un combinador clave en la programación funcional. Debido a que Julia tiene una ávida evaluación semántica (al igual que muchos lenguajes funcionales, incluido Scheme, que inspira mucho a Julia), el combinador en Y original de Curry no funcionará de forma inmediata:

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

Sin embargo, un pariente cercano del combinador en Y, el combinador en Z, funcionará:

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

Este combinador toma una función y devuelve una función que cuando se llama con el argumento `x` , se pasa a sí misma `x` . ¿Por qué sería útil que una función se pase a sí misma? ¡Esto permite la recursión sin hacer referencia al nombre de la función!

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

Por lo tanto, `Z(fact)` convierte en una implementación recursiva de la función factorial, a pesar de que no hay recursión visible en esta definición de función. (La recursión es evidente en la definición del combinador `Z` , por supuesto, pero eso es inevitable en un lenguaje entusiasta). Podemos verificar que nuestra función realmente funciona:

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

No solo eso, sino que es lo más rápido que podemos esperar de una implementación recursiva. El código de LLVM demuestra que el resultado se compila en una rama antigua simple, resta,

llama y multiplica:

```
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"
}
```

El sistema combinador SKI

El sistema combinador SKI es suficiente para representar cualquier término de cálculo lambda. (En la práctica, por supuesto, las abstracciones lambda explotan hasta un tamaño exponencial cuando se traducen en SKI). Debido a la simplicidad del sistema, la implementación de los combinadores S, K e I es extraordinariamente simple:

Una traducción directa de Lambda Calculus

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

Podemos confirmar, utilizando el sistema de prueba unitaria , que cada combinador tiene el comportamiento esperado.

El combinador I es el más fácil de verificar; debe devolver el valor dado sin cambios:

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

El combinador K también es bastante sencillo: debe descartar su segundo argumento.

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

El combinador S es el más complejo; su comportamiento puede resumirse aplicando los dos primeros argumentos al tercer argumento, y aplicando el primer resultado al segundo. Podemos probar el combinador S más fácilmente probando algunas de sus formas al curry. `S(K)` , por

ejemplo, simplemente debe devolver su segundo argumento y descartar el primero, como vemos que sucede:

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

`S(I)(I)` debería aplicar su argumento a sí mismo:

```
@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)` aplica su segundo argumento a su primer argumento:

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

El combinador I descrito anteriormente tiene un nombre en la `Base` Julia estándar: `identity` . Por lo tanto, podríamos haber reescrito las definiciones anteriores con la siguiente definición alternativa de `I` :

```
const I = identity
```

Mostrando SKI Combinadores

Una debilidad con el enfoque anterior es que nuestras funciones no se muestran tan bien como nos gustaría. Podríamos reemplazar

```
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 algunas pantallas más informativas? ¡La respuesta es sí! Vamos a reiniciar el REPL, y esta vez definamos cómo se mostrará cada función:

```
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
```

Es importante evitar mostrar nada hasta que hayamos terminado de definir funciones. De lo contrario, corremos el riesgo de invalidar el caché de métodos, y nuestros nuevos métodos no

parecerán tener efecto de inmediato. Por eso hemos puesto punto y coma en las definiciones anteriores. Los puntos y coma suprimen la salida del REPL.

Esto hace que las funciones se muestren muy bien:

```
julia> S
S

julia> K
K

julia> I
I
```

Sin embargo, todavía nos encontramos con problemas cuando intentamos mostrar un cierre:

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

Sería mejor mostrar eso como `S(K)` . Para hacer eso, debemos explotar que los cierres tienen sus propios tipos individuales. Podemos acceder a estos tipos y agregarles métodos a través de la reflexión, utilizando `typeof` y el campo `primary` campo de `name` del tipo. Reinicie el REPL otra vez; Vamos a hacer más cambios:

```
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)
```

Y ahora, por fin, las cosas se muestran como nos gustaría:

```
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
```

Lea Combinadores en línea: https://riptutorial.com/es/julia-lang/topic/5758/combinadores

Capítulo 7: Comparaciones

Sintaxis

- x <y # si `x` es estrictamente menor que `y`
- x> y # si `x` es estrictamente mayor que `y`
- x == y # si `x` es igual a `y`
- x === y # alternativamente `x ≡ y` , si `x` es igual a `y`
- x ≤ y # alternativamente `x <= y` , si `x` es menor o igual que `y`
- x ≥ y # alternativamente `x >= y` , si `x` es mayor o igual que `y`
- x ≠ y # alternativamente `x != y` , si `x` no es igual a `y`
- x ≈ y # si `x` es aproximadamente igual a `y`

Observaciones

Tenga cuidado al voltear los signos de comparación. Julia define muchas funciones de comparación por defecto sin definir la versión invertida correspondiente. Por ejemplo, uno puede correr

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

pero no funciona hacer

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

Examples

Comparaciones encadenadas

Los operadores de comparación múltiple utilizados juntos están encadenados, como si estuvieran conectados a través del operador `&&` . Esto puede ser útil para cadenas de comparación legibles y matemáticamente concisas, como

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

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

Sin embargo, hay una diferencia importante entre `a > b > c` y `a > b && b > c` ; en este último, el término `b` se evalúa dos veces. Esto no importa mucho para los símbolos antiguos, pero podría importar si los términos en sí tienen efectos secundarios. Por ejemplo,

```
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
```

Echemos un vistazo más profundo a las comparaciones encadenadas, y cómo funcionan, al ver cómo se analizan y se reducen en expresiones . Primero, considere la comparación simple, lo que podemos ver es solo una simple llamada de función:

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

Ahora, si encadenamos la comparación, notamos que el análisis ha cambiado:

```
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
```

Después de analizar, la expresión se reduce a su forma final:

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

y notamos que esto es lo mismo que para `a > b && b >= c` :

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

```
        return false
    end)
```

Números ordinales

Veremos cómo implementar comparaciones personalizadas implementando un tipo personalizado, números ordinales . Para simplificar la implementación, nos centraremos en un pequeño subconjunto de estos números: todos los números ordinales hasta e incluyendo ε_0. Nuestra implementación está enfocada en la simplicidad, no en la velocidad; Sin embargo, la implementación tampoco es lenta.

Almacenamos números ordinales por su forma normal de Cantor . Debido a que la aritmética ordinal no es conmutativa, tomaremos primero la convención común de almacenar los términos más significativos.

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

Como la forma normal de Cantor es única, podemos probar la igualdad simplemente a través de la igualdad recursiva:

0.5.0

En la versión v0.5, hay una sintaxis muy agradable para hacer esto de manera compacta:

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

0.5.0

De lo contrario, define la función como es más típica:

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

Para finalizar nuestro pedido, debido a que este tipo tiene un pedido total, deberíamos sobrecargar la función `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
```

Para probar nuestro pedido, podemos crear algunos métodos para hacer números ordinales. Cero, por supuesto, se obtiene al no tener términos en la forma normal de Cantor:

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

Podemos definir una `expω` para calcular `ω^α` , y usarla para calcular 1 y ω:

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

Ahora tenemos una función de orden completamente funcional en los números ordinales:

```
julia> ORDINAL_ZERO < ORDINAL_ONE < ω < expω(ω)
true

julia> ORDINAL_ONE > ORDINAL_ZERO
true

julia> sort([ORDINAL_ONE, ω, expω(ω), ORDINAL_ZERO])

4-element Array{OrdinalNumber,1}:

OrdinalNumber(OrdinalNumber[],Int64[])

OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[],Int64[])],[1])

OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[],Int64[])],[1])],


OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(Ordin
```

En el último ejemplo, vemos que la impresión de números ordinales podría ser mejor, pero el resultado es el esperado.

Operadores Estándar

Julia soporta un conjunto muy grande de operadores de comparación. Éstos incluyen

1. Todas las siguientes secuencias de Unicode: `> < >= ≥ <= ≤ == === ≡ != ≠ !== ≢ ∈ ∉ ∋ ∌ ⊆ ⊈ ⊂ ⊄ ⊊ ∝ ∊ ∍ ∥ ∦ ∷ ∺ ∻ ∽ ∾ ≁ ≃ ≄ ≅ ≆ ≇ ≈ ≉ ≊ ≋ ≍ ≎ ≏ ≐ ≑ ≒ ≓ ≔ ≕ ≖ ≗ ≘ ≙ ≚ ≛ ≜ ≝ ≞ ≟ ≣ ≦ ≧ ≨ ≩ ≪ ≫ ≬ ≭ ≮ ≯ ≰ ≱ ≲ ≳ ≴ ≵ ≶ ≷ ≸ ≹ ≺ ≻ ≼ ≽ ≾ ≿ ⊀ ⊁ ⊃ ⊅ ⊇ ⊉ ⊋ ⊏ ⊐ ⊑ ⊒ ⊜ ⊩ ⊬ ⊮ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⋍ ⋐ ⋑ ⋕ ⋖ ⋗ ⋘ ⋙ ⋚ ⋛ ⋜ ⋝ ⋞ ⋟ ⋠ ⋡ ⋢ ⋣ ⋤ ⋥ ⋦ ⋧ ⋨ ⋩ ⋪ ⋫ ⋬ ⋭ ⋲ ⋳ ⋴ ⋵ ⋶ ⋷ ⋸ ⋹ ⋺ ⋻ ⋼ ⋽ ⋾ ⋿ ⟈ ⟉ ⟒ ⦷ ⧀ ⧁ ⧡ ⧣ ⧤ ⧥ ⩦ ⩧ ⩪ ⩫ ⩬ ⩭ ⩮ ⩯ ⩰ ⩱ ⩲ ⩳ ⩴ ⩵ ⩶ ⩷ ⩸ ⩹ ⩺ ⩻ ⩼ ⩽ ⩾ ⩿ ⪀ ⪁ ⪂ ⪃ ⪄ ⪅ ⪆ ⪇ ⪈ ⪉ ⪊ ⪋ ⪌ ⪍ ⪎ ⪏ ⪐ ⪑ ⪒ ⪓ ⪔ ⪕ ⪖ ⪗ ⪘ ⪙ ⪚ ⪛ ⪜ ⪝ ⪞ ⪟ ⪠ ⪡ ⪢ ⪣ ⪤ ⪥ ⪦ ⪧ ⪨ ⪩ ⪪ ⪫ ⪬ ⪭ ⪮ ⪯ ⪰ ⪱ ⪲ ⪳ ⪴ ⪵ ⪶ ⪷ ⪸ ⪹ ⪺ ⪻ ⪼ ⪽ ⪾ ⪿ ⫀ ⫁ ⫂ ⫃ ⫄ ⫅ ⫆ ⫇ ⫈ ⫉ ⫊ ⫋ ⫌ ⫍ ⫎ ⫏ ⫐ ⫑ ⫒ ⫓ ⫔ ⫕ ⫖ ⫗ ⫘ ⫙ ⫷ ⫸ ⫹ ⫺ ⊢ ⊣`
`> < >= ≥ <= ≤ == === ≡ != ≠ !== ≢ ∈ ∉ ∋ ∌ ⊆ ⊈ ⊂ ⊄ ⊊ ∝ ∊ ∍ ∥ ∦ ∷ ∺ ∻ ∽ ∾ ≁ ≃ ≄ ≅ ≆ ≇ ≈ ≉ ≊ ≋ ≍ ≎ ≏ ≐ ≑ ≒ ≓ ≔ ≕ ≖ ≗ ≘ ≙ ≚ ≛ ≜ ≝ ≞ ≟ ≣ ≦ ≧ ≨ ≩ ≪ ≫ ≬ ≭ ≮ ≯ ≰ ≱ ≲ ≳ ≴ ≵ ≶ ≷ ≸ ≹ ≺ ≻ ≼ ≽ ≾ ≿ ⊀ ⊁ ⊃ ⊅ ⊇ ⊉ ⊋ ⊏ ⊐ ⊑ ⊒ ⊜ ⊩ ⊬ ⊮ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⋍ ⋐ ⋑ ⋕ ⋖ ⋗ ⋘ ⋙ ⋚ ⋛ ⋜ ⋝ ⋞ ⋟ ⋠ ⋡ ⋢ ⋣ ⋤ ⋥ ⋦ ⋧ ⋨ ⋩ ⋪ ⋫ ⋬ ⋭ ⋲ ⋳ ⋴ ⋵ ⋶ ⋷ ⋸ ⋹ ⋺ ⋻ ⋼ ⋽ ⋾ ⋿ ⟈ ⟉ ⟒ ⦷ ⧀ ⧁ ⧡ ⧣ ⧤ ⧥ ⩦ ⩧ ⩪ ⩫ ⩬ ⩭ ⩮ ⩯ ⩰ ⩱ ⩲ ⩳ ⩴ ⩵ ⩶ ⩷ ⩸ ⩹ ⩺ ⩻ ⩼ ⩽ ⩾ ⩿ ⪀ ⪁ ⪂ ⪃ ⪄ ⪅ ⪆ ⪇ ⪈ ⪉ ⪊ ⪋ ⪌ ⪍ ⪎ ⪏ ⪐ ⪑ ⪒ ⪓ ⪔ ⪕ ⪖ ⪗ ⪘ ⪙ ⪚ ⪛ ⪜ ⪝ ⪞ ⪟ ⪠ ⪡ ⪢ ⪣ ⪤ ⪥ ⪦ ⪧ ⪨ ⪩ ⪪ ⪫ ⪬ ⪭ ⪮ ⪯ ⪰ ⪱ ⪲ ⪳ ⪴ ⪵ ⪶ ⪷ ⪸ ⪹ ⪺ ⪻ ⪼ ⪽ ⪾ ⪿ ⫀ ⫁ ⫂ ⫃ ⫄ ⫅ ⫆ ⫇ ⫈ ⫉ ⫊ ⫋ ⫌ ⫍ ⫎ ⫏ ⫐ ⫑ ⫒ ⫓ ⫔ ⫕ ⫖ ⫗ ⫘ ⫙ ⫷ ⫸ ⫹ ⫺ ⊢ ⊣`

2. Todos los símbolos en el punto 1, precedidos por un punto (`.`) Que se realizará de forma elemental;
3. Los operadores `<:` , `>:` , `.!` , y `in` , que no puede ir precedido por un punto (`.`).

No todos estos tienen una definición en la biblioteca `Base` estándar. Sin embargo, están disponibles para que otros paquetes los definan y utilicen según corresponda.

En el uso diario, la mayoría de estos operadores de comparación no son relevantes. Las más comunes utilizadas son las funciones matemáticas estándar para ordenar; vea la sección de Sintaxis para una lista.

Como la mayoría de los otros operadores en Julia, los operadores de comparación son funciones y se pueden llamar como funciones. Por ejemplo, `(<)(1, 2)` es idéntico en significado a `1 < 2` .

Usando ==, ===, y isequal

Hay tres operadores de igualdad: `==` , `===` e `isequal` . (El último no es realmente un operador, pero es una función y todos los operadores son funciones).

Cuándo usar `==`

`==` es la igualdad de *valores* . Devuelve `true` cuando dos objetos representan, en su estado actual, el mismo valor.

Por ejemplo, es obvio que

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

pero además

```
julia> 1 == 1.0
true

julia> 1 == 1.0 + 0.0im
true

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

Los lados derechos de cada igualdad anterior son de un tipo diferente, pero aún representan el mismo valor.

Para objetos mutables, como matrices , `==` compara su valor presente.

```
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

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
```

La mayoría de las veces, `==` es la elección correcta.

Cuándo usar `===`

`===` es una operación mucho más estricta que `==` . En lugar de valorar la igualdad, mide la igualdad. Dos objetos son iguales si el programa no puede distinguirlos entre sí. Asi tenemos

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

Como no hay manera de distinguir un `1` aparte de otro `1` . Pero

```
julia> 1 === 1.0
false
```

Porque aunque `1` y `1.0` tienen el mismo valor, son de diferentes tipos, por lo que el programa puede distinguirlos.

Además,

```
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
```

lo que a primera vista puede parecer sorprendente! ¿Cómo podría el programa distinguir entre los dos vectores `A` y `B` ? Debido a que los vectores son mutables, podría modificar `A` , y luego se comportaría de manera diferente a `B` Pero no importa cómo modifique `A` , `A` siempre se comportará igual que `A` Entonces, `A` es igual a `A` , pero no igual a `B`

Continuando por esta vena, observar.

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

julia> A === C
true
```

Al asignar `A` a `C` , decimos que `C` tiene un *alias* `A` Es decir, se ha convertido en otro nombre para `A` Cualquier modificación hecha a `A` será observada por `C` también. Por lo tanto, no hay manera de decir la diferencia entre `A` y `C` , por lo que son iguales.

Cuando usar `isequal`

La diferencia entre `==` e `isequal` es muy sutil. La mayor diferencia está en cómo se manejan los números de punto flotante:

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

Este resultado posiblemente sorprendente está definido por el estándar IEEE para tipos de punto flotante (IEEE-754). Pero esto no es útil en algunos casos, como la clasificación. `isequal` se proporciona para esos casos:

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

En la otra cara del espectro, `==` trata el cero negativo de IEEE y el cero positivo como el mismo valor (también según lo especificado por IEEE-754). Sin embargo, estos valores tienen representaciones distintas en la memoria.

```
julia> 0.0
0.0

julia> -0.0
-0.0

julia> 0.0 == -0.0
true
```

De nuevo para propósitos de clasificación, `isequal` distingue entre ellos.

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

Lea Comparaciones en línea: https://riptutorial.com/es/julia-lang/topic/5563/comparaciones

Capítulo 8: Compatibilidad de versiones cruzadas

Sintaxis

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

Observaciones

A veces es muy difícil obtener una nueva sintaxis para jugar bien con varias versiones. Como Julia aún se encuentra en desarrollo activo, a menudo es útil simplemente abandonar el soporte para versiones anteriores y en su lugar, apuntar solo a las más nuevas.

Examples

Números de versión

Julia tiene una implementación integrada de versiones semánticas expuestas a través del tipo `VersionNumber` .

Para construir un `VersionNumber` como un literal, se puede usar la macro de cadena `@v_str` :

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

Alternativamente, uno puede llamar al constructor `VersionNumber` ; tenga en cuenta que el constructor acepta hasta cinco argumentos, pero todos excepto el primero son opcionales.

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

Los números de versión se pueden comparar utilizando operadores de comparación y, por lo tanto, se pueden ordenar:

```
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"
```

Los números de versión se utilizan en varios lugares a través de Julia. Por ejemplo, la constante `VERSION` es una `VersionNumber` :

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

Esto se usa comúnmente para la evaluación de códigos condicionales, dependiendo de la versión de Julia. Por ejemplo, para ejecutar un código diferente en v0.4 y v0.5, se puede hacer

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

Cada paquete instalado también está asociado con un número de versión actual:

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

Usando Compat.jl

El paquete Compat.jl permite el uso de algunas funciones y sintaxis nuevas de Julia con versiones anteriores de Julia. Sus características están documentadas en su README, pero a continuación se presenta un resumen de las aplicaciones útiles.

0.5.0

Tipo de cadena unificada

En Julia v0.4, había muchos tipos diferentes de cuerdas . Este sistema se consideró demasiado complejo y confuso, por lo que en Julia v0.5, solo queda el tipo `String` . `Compat` permite usar el tipo de `String` y el constructor en la versión 0.4, bajo el nombre `Compat.String` . Por ejemplo, este código v0.5

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

se puede traducir directamente a este código, que funciona tanto en v0.5 como en v0.4:

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

Tenga en cuenta que hay algunas advertencias.

- En v0.4, `Compat.String` está tipeado para `ByteString` , que es `Union{ASCIIString, UTF8String}` . Por lo tanto, los tipos con campos de `String` no serán de tipo estable. En estas situaciones, se recomienda `Compat.UTF8String` , ya que significará `String` en v0.5, y `UTF8String` en v0.4, que son tipos concretos.
- Uno debe tener cuidado de usar `Compat.String` o `import Compat: String` , porque `String` sí tiene un significado en v0.4: es un alias obsoleto para `AbstractString` . Una señal de que `String` se usó accidentalmente en lugar de `Compat.String` es si, en algún momento, aparecen las siguientes advertencias:

```
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
```

Sintaxis de transmisión compacta

Julia v0.5 introduce el azúcar sintáctico para su `broadcast` . La sintaxis

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

se baja a la `broadcast(f, x, y)` . Ejemplos de uso de esta sintaxis incluyen el `sin.([1, 2, 3])` para tomar el seno de múltiples números a la vez.

En v0.5, la sintaxis se puede utilizar directamente:

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

Sin embargo, si intentamos lo mismo en v0.4, obtenemos un error:

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

Afortunadamente, `Compat` hace que esta nueva sintaxis sea utilizable también desde v0.4. Una vez más, añadimos `using Compat` . Esta vez, rodeamos la expresión 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
```

Lea Compatibilidad de versiones cruzadas en línea: https://riptutorial.com/es/julia-

lang/topic/5832/compatibilidad-de-versiones-cruzadas

Capítulo 9: Comprensiones

Examples

Comprensión de matriz

Sintaxis basica

Las comprensiones de matriz de Julia usan la siguiente sintaxis:

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

Tenga en cuenta que al igual que con `for` bucles , todos `=` , `in` , y `∈` son aceptados para la comprensión.

¡Esto es aproximadamente equivalente a crear una matriz vacía y usar un bucle `for` para `push!` artículos para ello.

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

sin embargo, el tipo de comprensión de una matriz es lo más estrecho posible, lo que es mejor para el rendimiento.

Por ejemplo, para obtener una matriz de los cuadrados de los enteros de `1` a `10` , se puede usar el siguiente código.

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

Este es un reemplazo limpio y conciso para la versión más larga `for` -loop.

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

Comprensión de matriz condicional

Antes de la Julia 0.5, no hay forma de usar condiciones dentro de la comprensión de la matriz. Pero, ya no es cierto. En Julia 0.5 podemos usar las condiciones dentro de condiciones como las siguientes:

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

```
 36
 49
 64
 81
```

La fuente del ejemplo anterior se puede encontrar aquí .

Si nos gustaría utilizar la comprensión de la lista anidada:

```
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)
```

Comprensiones de matrices multidimensionales.

Anidado `for` bucles se puede usar para iterar sobre varios iterables únicos.

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

De manera similar, se pueden suministrar múltiples especificaciones de iteración a una comprensión de matriz.

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

Por ejemplo, lo siguiente puede usarse para generar el producto cartesiano de `1:3` y `1:2` .

```
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)
```

Las comprensiones de matrices multidimensionales aplanadas son similares, excepto que pierden la forma. Por ejemplo,

```
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)
```

Es una variante aplanada de las anteriores. La diferencia sintáctica es que se usa un adicional `for` lugar de una coma.

Comprensiones de generador

Las comprensiones de generadores siguen un formato similar a las comprensiones de matrices, pero usan paréntesis `()` lugar de corchetes `[]` .

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

Dicha expresión devuelve un objeto `Generator` .

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

Argumentos de la función

Las comprensiones del generador pueden proporcionarse como el único argumento de una función, sin la necesidad de un conjunto adicional de paréntesis.

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

Sin embargo, si se proporciona más de un argumento, la comprensión del generador requiere su propio conjunto de paréntesis.

```
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"
```

Lea Comprensiones en línea: https://riptutorial.com/es/julia-lang/topic/5477/comprensiones

Capítulo 10: Condicionales

Sintaxis

- si cond cuerpo; fin
- si cond cuerpo; más; cuerpo; fin
- si cond cuerpo; elseif cond; cuerpo; más; fin
- si cond cuerpo; elseif cond; cuerpo; fin
- cond? iftrue: iffalse
- cond && si es cierto
- cond || iffalse
- ifelse (cond, iftrue, iffalse)

Observaciones

Todos los operadores y funciones condicionales implican el uso de condiciones booleanas (`true` o `false`). En Julia, el tipo de booleanos es `Bool` . A diferencia de otros idiomas, otros tipos de números (como `1` o `0`), cadenas, matrices, etc. , *no* pueden usarse directamente en condicionales.

Típicamente, uno usa funciones de predicado (funciones que devuelven un `Bool`) o operadores de comparación en la condición de un operador o función condicional.

Examples

si ... otra expresión

El condicional más común en Julia es la expresión `if` ... `else` . Por ejemplo, a continuación implementamos el algoritmo euclidiano para calcular el mayor divisor común , utilizando un condicional para manejar el caso base:

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

La forma `if` ... `else` en Julia es en realidad una expresión, y tiene un valor; el valor es la expresión en la posición de cola (es decir, la última expresión) en la rama que se toma. Considere la siguiente entrada de muestra:

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

Aquí, `a` es `0` y `b` es `-10` . La condición `a == 0` es `true` , por lo que se toma la primera rama. El valor

devuelto es `abs(b)` que es `10` .

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

Aquí, `a` es `2` y `b` es `3` . La condición `a == 0` es falsa, por lo que se toma la segunda rama y calculamos `mygcd(b % a, a)` , que es `mygcd(3 % 2, 2)` . El operador `%` devuelve el resto cuando `3` se divide por `2` , en este caso `1` . Por lo tanto, calculamos `mygcd(1, 2)` , y esta vez `a` es `1` y `b` es `2` . Una vez más, `a == 0` es falso, por lo que se toma la segunda rama y calculamos `mygcd(b % a, a)` , que es `mygcd(0, 1)` . Esta vez, se devuelve a `a == 0` y se devuelve `abs(b)` , que da el resultado `1` .

si ... otra declaración

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

Cualquier expresión, como la expresión `if` ... `else` , puede colocarse en una posición de declaración. Esto ignora su valor pero sigue ejecutando la expresión para sus efectos secundarios.

si declaración

Al igual que cualquier otra expresión, el valor de retorno de una expresión `if` ... `else` puede ignorarse (y, por lo tanto, descartarse). En general, esto solo es útil cuando el cuerpo de la expresión tiene efectos secundarios, como escribir en un archivo, mutar variables o imprimir en la pantalla.

Además, la rama `else` de una expresión `if` ... `else` es opcional. Por ejemplo, podemos escribir el siguiente código para enviar a la pantalla solo si se cumple una condición particular:

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

En el ejemplo anterior, usamos las funciones de fecha y hora para obtener el segundo actual; por ejemplo, si actualmente es 10:55:27, la variable `second` mantendrá `27` . Si este número es par, entonces se imprimirá una línea en la pantalla. De lo contrario, no se hará nada.

Operador condicional ternario

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

El operador condicional ternario es una expresión menos wordy `if` ... `else` .

La sintaxis específicamente es:

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

En este ejemplo, agregamos `x` a la colección `A` solo si `x` no está ya en `A` De lo contrario, simplemente dejamos `A` sin cambios.

Referencias del operador ternario:

- Documentacion julia
- Wikilibros

Operadores de cortocircuito: && y ||

Para ramificación

Los operadores condicionales de cortocircuito `&&` y `||` Puede usarse como reemplazos ligeros para las siguientes construcciones:

- `x && y` es equivalente a `x ? y : x`
- `x || y` es equivalente a `x ? x : y`

Un uso para operadores de cortocircuito es como una forma más concisa de probar una condición y realizar una determinada acción dependiendo de esa condición. Por ejemplo, el siguiente código utiliza el operador `&&` para lanzar un error si el argumento `x` es negativo:

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

El `||` El operador también se puede usar para la comprobación de errores, excepto que activa el error a *menos* que se cumpla una condición, en lugar de *si* la condición se cumple

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

Otra aplicación útil de esto es proporcionar un valor predeterminado a un objeto, solo si no está definido previamente:

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

Aquí, esto comprueba si el símbolo x está definido (es decir, si hay un valor asignado al objeto `x`). Si es así, entonces no pasa nada. Pero, si no, entonces a `x` se le asignará `NEW_VALUE` . Tenga en cuenta que este ejemplo solo funcionará en un alcance de nivel superior.

En condiciones

Los operadores también son útiles porque se pueden usar para probar dos condiciones, la segunda de las cuales solo se evalúa según el resultado de la primera condición. De la documentación de Julia:

> En la expresión `a && b` , la subexpresión `b` solo se evalúa si `a` evalúa como `true`
>
> En la expresión `a || b` , la subexpresión `b` solo se evalúa si `a` evalúa como `false`

Por lo tanto, mientras tanto `a & b` como `a && b` resultarán `true` si `a` y `b` son `true` , su comportamiento si `a` es `false` es diferente.

Por ejemplo, supongamos que deseamos verificar si un objeto es un número positivo, donde es posible que ni siquiera sea un número. Considere las diferencias entre estos dos intentos de implementación:

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

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

`CheckPositive1()` producirá un error si se le proporciona un tipo no numérico como argumento. Esto se debe a que evalúa *ambas* expresiones, independientemente del resultado de la primera, y la segunda expresión producirá un error cuando uno intenta evaluarla para un tipo no numérico.

`CheckPositive2()` embargo, `CheckPositive2()` producirá `false` (en lugar de un error) si se le proporciona un tipo no numérico, ya que la segunda expresión solo se evalúa si la primera es `true` .

Más de un operador de cortocircuito se puede unir. P.ej:

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

Si la declaración con múltiples sucursales

```
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
```

Se puede utilizar cualquier número de ramas `elseif` con una sentencia `if` , posiblemente con o sin una rama `else` final. Las condiciones subsiguientes solo se evaluarán si todas las condiciones

anteriores han resultado ser `false` .

La función ifelse

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

Uso:

```
julia> shift(10)
9

julia> shift(11)
12

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

La función `ifelse` evaluará ambas ramas, incluso la que no está seleccionada. Esto puede ser útil cuando las ramas tienen efectos secundarios que deben evaluarse, o porque puede ser más rápido si ambas ramas son baratas.

Lea Condicionales en línea: https://riptutorial.com/es/julia-lang/topic/4356/condicionales

Capítulo 11: Entrada

Sintaxis

- readline ()
- líneas de lectura ()
- lectura (STDIN)
- chomp (str)
- abrir (f, archivo)
- cada linea (io)
- Lectura (archivo)
- leer (archivo)
- readcsv (archivo)
- readdlm (archivo)

Parámetros

Parámetro	Detalles
`chomp(str)`	**Eliminar hasta una nueva línea final de una cadena.**
`str`	La cadena para despojar una nueva línea final de. Tenga en cuenta que las cadenas son inmutables por convención. Esta función devuelve una nueva cadena.
`open(f, file)`	**Abra un archivo, llame a la función y cierre el archivo después.**
`f`	La función para llamar a la secuencia IO que abre el archivo genera.
`file`	La ruta del archivo a abrir.

Examples

Leyendo una cadena de entrada estándar

El flujo `STDIN` en Julia se refiere a la entrada estándar . Esto puede representar la entrada del usuario, para programas de línea de comandos interactivos, o la entrada de un archivo o canalización que se ha redirigido al programa.

La función `readline` , cuando no se proporciona ningún argumento, leerá los datos de `STDIN` hasta que se encuentre una nueva línea, o la secuencia `STDIN` ingrese al estado de fin de archivo. Estos dos casos se pueden distinguir por si el carácter `\n` ha sido leído como el carácter final:

```
julia> readline()
some stuff
"some stuff\n"

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

A menudo, para los programas interactivos, no nos importa el estado EOF, y solo queremos una cadena. Por ejemplo, podemos pedirle al usuario que ingrese información:

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

Esto no es del todo satisfactorio, sin embargo, debido a la nueva línea adicional:

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

La función `chomp` está disponible para eliminar hasta una nueva línea final de una cadena. Por ejemplo:

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

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

Por lo tanto, podemos aumentar nuestra función con `chomp` para que el resultado sea el esperado:

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

que tiene un resultado más deseable:

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

A veces, es posible que deseamos leer tantas líneas como sea posible (hasta que la secuencia de entrada ingrese al estado de fin de archivo). La función `readlines` proporciona esa capacidad.

```
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

Una vez más, si no nos gustan las nuevas líneas al final de las líneas leídas por `readlines` , podemos usar la función `chomp` para eliminarlas. Esta vez, transmitimos la función `chomp` en toda la matriz:

```
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"
```

Otras veces, es posible que no nos interesen las líneas en absoluto, y simplemente queremos leer tanto como sea posible como una sola cadena. La función de `readstring` realiza esto:

```
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"
```

(El `# [END OF INPUT]` no es parte de la entrada original; se ha agregado para mayor claridad.)

Tenga en cuenta que la `readstring` debe pasar el argumento `STDIN` .

Lectura de números de entrada estándar

Leer números de una entrada estándar es una combinación de leer cadenas y analizar cadenas como números.

La función de `parse` se utiliza para analizar una cadena en el tipo de número deseado:

```
julia> parse(Int, "17")
17
```

```
julia> parse(Float32, "-3e6")
-3.0f6
```

El formato esperado por el `parse(T, x)` es similar, pero no exactamente el mismo, que el formato que Julia espera de los literales numéricos :

```
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 combinación de las funciones `parse` y `readline` nos permite leer un solo número desde una línea:

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

que funciona como se espera:

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

Se aplican las advertencias habituales sobre la precisión de punto flotante . Tenga en cuenta que el `parse` se puede usar con `BigInt` y `BigFloat` para eliminar o minimizar la pérdida de precisión.

A veces, es útil leer más de un número de la misma línea. Normalmente, la línea se puede dividir con espacios en blanco:

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

que se puede utilizar de la siguiente manera:

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

Leer datos de un archivo

Leyendo cadenas o bytes

Los archivos se pueden abrir para leer usando la función de `open` , que a menudo se usa junto con la sintaxis de bloque do :

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

Supongamos que `myfile` existe y sus contenidos son

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

Entonces, este código produciría el siguiente resultado:

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

Tenga en cuenta que `eachline` es un iterable perezoso sobre las líneas del archivo. Se prefiere a las `readlines` de `readlines` por razones de rendimiento.

Debido a `do` sintaxis de bloque es solo azúcar sintáctica para funciones anónimas, también podemos pasar funciones con nombre para `open` :

```
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
```

Las funciones de `read` y `readstring` `read` proporcionan métodos convenientes que abrirán un archivo automáticamente:

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

Lectura de datos estructurados

Supongamos que tenemos un archivo CSV con el siguiente contenido, en un archivo llamado `file.csv` :

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

Entonces podemos usar la función `readcsv` para leer estos datos en 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
```

Si el archivo se delimitó en su lugar con pestañas, en un archivo llamado `file.tsv` , entonces se puede usar la función `readdlm` su lugar, con el argumento `delim` establecido en `'\t'` . Las cargas de trabajo más avanzadas deberían usar el paquete CSV.jl.

Lea Entrada en línea: https://riptutorial.com/es/julia-lang/topic/7201/entrada

Capítulo 12: Enums

Sintaxis

- @enum EnumType val = 1 val val
- :símbolo

Observaciones

A veces es útil tener tipos enumerados donde cada instancia es de un tipo diferente (a menudo un tipo inmutable de singleton); Esto puede ser importante para la estabilidad del tipo. Los rasgos son típicamente implementados con este paradigma. Sin embargo, esto da como resultado una sobrecarga adicional de tiempo de compilación.

Examples

Definiendo un tipo enumerado

Un tipo enumerado es un tipo que puede contener uno de una lista finita de valores posibles. En Julia, los tipos enumerados normalmente se llaman "tipos de enumeración". Por ejemplo, uno podría usar tipos de enumeración para describir los siete días de la semana, los doce meses del año, los cuatro palos de un mazo estándar de 52 cartas u otras situaciones similares.

Podemos definir los tipos enumerados para modelar las demandas y los rangos de un mazo estándar de 52 cartas. La macro `@enum` se utiliza para definir tipos de enumeración.

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

Esto define dos tipos: `Suit` y `Rank` . Podemos comprobar que los valores son de hecho de los tipos esperados:

```
julia> ♦
♦::Suit = 1

julia> six
six::Rank = 6
```

Tenga en cuenta que cada palo y rango se ha asociado con un número. Por defecto, este número comienza en cero. Entonces al segundo palo, diamantes, se le asignó el número 1. En el caso de `Rank` , puede tener más sentido comenzar el número en uno. Esto se logró anotando la definición de `ace` con una anotación `=1` .

Los tipos enumerados vienen con muchas funcionalidades, como la igualdad (y de hecho la identidad) y las comparaciones integradas:

```
julia> seven === seven
true

julia> ten ≠ jack
true

julia> two < three
true
```

Al igual que los valores de cualquier otro tipo inmutable , los valores de los tipos enumerados también se pueden hashear y almacenar en `Dict` s.

Podemos completar este ejemplo definiendo un tipo de `Card` que tiene un `Rank` y un campo de `Suit` :

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

Y por lo tanto podemos crear tarjetas con

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

Pero los tipos enumerados también vienen con sus propios métodos de `convert` , por lo que podemos simplemente hacer

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

y como `7` se puede convertir directamente a `Rank` , este constructor trabaja fuera de la caja.

Podríamos desear definir el azúcar sintáctico para construir estas tarjetas; La multiplicación implícita proporciona una manera conveniente de hacerlo. Definir

```
julia> import Base.*

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

y entonces

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

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

Una vez más aprovechando las funciones de `convert` incorporadas.

Usando símbolos como enumeraciones ligeras

Aunque la macro `@enum` es bastante útil para la mayoría de los casos de uso, puede ser excesiva en algunos casos de uso. Las desventajas de `@enum` incluyen:

- Crea un nuevo tipo.
- Es un poco más difícil de extender.
- Viene con funciones como conversión, enumeración y comparación, que pueden ser superfluas en algunas aplicaciones.

En los casos en que se desee una alternativa más liviana, se puede usar el tipo de `Symbol` . Los símbolos son cadenas internas ; representan secuencias de caracteres, al igual que las cadenas , pero están asociadas únicamente con los números. Esta asociación única permite la comparación rápida de la igualdad de símbolos.

Podemos volver a implementar un tipo de `Card` , esta vez usando los campos de `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
```

Implementamos el constructor interno para verificar los valores incorrectos pasados al constructor. A diferencia de lo que `@enum` en el ejemplo que utiliza los tipos de `@enum` , los `Symbol` s pueden contener cualquier cadena, por lo que debemos tener cuidado con los tipos de `Symbol` que aceptamos. Tenga en cuenta aquí el uso de los operadores condicionales de cortocircuito .

Ahora podemos construir objetos de `Card` como esperamos:

```
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 beneficio importante de `Symbol` s es su extensibilidad en tiempo de ejecución. Si en el tiempo de ejecución, deseamos aceptar (por ejemplo) `:eleven` como un nuevo rango, ¡basta con ejecutar

simplemente `push!(ranks, :eleven)` . Dicha extensibilidad en tiempo de ejecución no es posible con los tipos de `@enum` .

Lea Enums en línea: https://riptutorial.com/es/julia-lang/topic/7104/enums

Capítulo 13: Examen de la unidad

Sintaxis

- @test [expr]
- @test_throws [Exception] [expr]
- @testset "[nombre]" comienza; [pruebas]; fin
- Pkg.test ([paquete])

Observaciones

La documentación estándar de la biblioteca para `Base.Test` cubre material adicional más allá de lo que se muestra en estos ejemplos.

Examples

Probando un paquete

Para ejecutar las pruebas unitarias de un paquete, use la función `Pkg.test` . Para un paquete llamado `MyPackage` , el comando sería

```
julia> Pkg.test("MyPackage")
```

Una salida esperada sería similar 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
```

aunque obviamente, no se puede esperar que coincida exactamente con lo anterior, ya que los paquetes diferentes utilizan marcos diferentes.

Este comando ejecuta el archivo `test/runtests.jl` del paquete en un entorno limpio.

Uno puede probar todos los paquetes instalados a la vez con

```
julia> Pkg.test()
```

pero esto usualmente toma mucho tiempo.

Escribir una prueba simple

Las pruebas unitarias se declaran en el archivo `test/runtests.jl` en un paquete. Por lo general, este archivo comienza.

```
using MyModule
using Base.Test
```

La unidad básica de prueba es la macro `@test` . Esta macro es como una aseveración de clases. Cualquier expresión booleana se puede probar en la macro `@test` :

```
@test 1 + 1 == 2
@test iseven(10)
@test 9 < 10 || 10 < 9
```

Podemos probar la macro `@test` en el 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 de prueba se puede utilizar en casi cualquier lugar, como en bucles o funciones:

```
# 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)
```

Escribir un conjunto de prueba

0.5.0

En la versión v0.5, los conjuntos de pruebas están integrados en el módulo estándar de `Base.Test` biblioteca, y no tiene que hacer nada especial (además de `using Base.Test`) para usarlos.

0.4.0

Los conjuntos de pruebas no forman parte de la biblioteca `Base.Test` de Julia v0.4. En su lugar, debe `REQUIRE` el módulo `BaseTestNext` y agregar `using BaseTestNext` a su archivo. Para soportar tanto la versión 0.4 como la 0.5, podrías usar

```
if VERSION ≥ v"0.5.0-dev+7720"
    using Base.Test
else
    using BaseTestNext
    const Test = BaseTestNext
end
```

Es útil agrupar `@test` relacionados en un conjunto de prueba. Además de una organización de pruebas más clara, los conjuntos de pruebas ofrecen mejores resultados y más personalización.

Para definir un conjunto de pruebas, simplemente envuelva cualquier número de `@test` s con un bloque `@testset` :

```
@testset "+" begin
    @test 1 + 1 == 2
    @test 2 + 2 == 4
end

@testset "*" begin
    @test 1 * 1 == 1
    @test 2 * 2 == 4
end
```

La ejecución de estos conjuntos de pruebas imprime el siguiente resultado:

```
Test Summary: | Pass  Total
  +           |    2      2

Test Summary: | Pass  Total
  *           |    2      2
```

Incluso si un conjunto de pruebas contiene una prueba que falla, todo el conjunto de pruebas se ejecutará hasta su finalización, y las fallas se registrarán e informarán:

```
@testset "-" begin
    @test 1 - 1 == 0
    @test 2 - 2 == 1
    @test 3 - () == 3
    @test 4 - 4 == 0
end
```

Ejecutando los resultados de este conjunto de pruebas en

```
-: 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.
    ...
```

Los conjuntos de pruebas se pueden anidar, lo que permite una organización arbitrariamente profunda

```
@testset "Int" begin
    @testset "+" begin
        @test 1 + 1 == 2
        @test 2 + 2 == 4
    end
    @testset "-" begin
        @test 1 - 1 == 0
    end
end
```

Si las pruebas pasan, esto solo mostrará los resultados del conjunto de pruebas más externo:

```
Test Summary: | Pass  Total
  Int         |    3      3
```

Pero si las pruebas fallan, se informa un desglose en el conjunto de pruebas exacto y la prueba que causa el fallo.

La macro `@testset` se puede usar con un bucle `for` para crear muchos conjuntos de pruebas a la vez:

```
@testset for i in 1:5
    @test 2i == i + i
    @test i^2 == i * i
    @test i ÷ i == 1
end
```

que informa

```
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 estructura común es tener conjuntos de prueba externos componentes o tipos de prueba. Dentro de estos conjuntos de pruebas externas, los conjuntos de pruebas internas prueban el comportamiento. Por ejemplo, supongamos que creamos un tipo `UniversalSet` con una instancia de singleton que contiene todo. Antes de que implementemos el tipo, podemos usar principios de desarrollo controlados por pruebas e implementar las pruebas:

```
@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
```

Entonces podemos comenzar a implementar nuestra funcionalidad hasta que pase nuestras pruebas. El primer paso es definir el tipo:

```
immutable UniversalSet <: Base.AbstractSet end
```

Sólo dos de nuestras pruebas pasan ahora. Podemos implementar `in` :

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
```

Esto también hace que algunas de nuestras pruebas de subconjunto pasen. Sin embargo, el `issubset` (`⊆`) no funciona para `UniversalSet` , porque el respaldo trata de iterar sobre elementos, lo que no podemos hacer. Simplemente podemos definir una especialización que hace que `issubset` devuelva `true` para cualquier conjunto:

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
Base.issubset(x::Base.AbstractSet, ::UniversalSet) = true
```

Y ahora, todas nuestras pruebas pasan!

Pruebas de excepciones

Las excepciones encontradas durante la ejecución de una prueba fallarán en la prueba, y si la prueba no está en un conjunto de pruebas, finalice la prueba del motor. Por lo general, esto es bueno, porque en la mayoría de las situaciones las excepciones no son el resultado deseado. Pero a veces, uno quiere probar específicamente que se produce una cierta excepción. La macro `@test_throws` facilita esto.

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
  Expression: ([1,2,3])[4]
      Thrown: BoundsError
```

Si se lanza la excepción incorrecta, `@test_throws` seguirá fallando:

```
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
```

y si no se produce ninguna excepción, `@test_throws` también fallará:

```
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
```

Prueba de Punto Flotante Aproximada Igualdad

¿Cuál es el trato con lo siguiente?

```
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
```

El error se debe al hecho de que ninguno de `0.1` , `0.2` y `0.3` está representado en la computadora exactamente como esos valores: `1//10` , `2//10` y `3//10` . En cambio, se aproximan por valores que están muy cerca. Pero como se vio en el fallo de la prueba anterior, al sumar dos aproximaciones, el resultado puede ser una aproximación un poco peor de lo que es posible. Hay mucho más sobre este tema que no se puede cubrir aquí.

¡Pero no estamos sin suerte! Para probar que la combinación de redondeo a un número de punto flotante y la aritmética de punto flotante es *aproximadamente* correcta, aunque no exacta, podemos utilizar el `isapprox` función (que corresponde al operador `≈`). Así que podemos reescribir nuestra prueba como

```
julia> @test 0.1 + 0.2 ≈ 0.3
Test Passed
  Expression: 0.1 + 0.2 ≈ 0.3
   Evaluated: 0.30000000000000004 isapprox 0.3
```

Por supuesto, si nuestro código estaba completamente equivocado, la prueba todavía detectará eso:

```
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 función `isapprox` utiliza heurísticas basadas en el tamaño de los números y la precisión del tipo de punto flotante para determinar la cantidad de error a tolerar. No es apropiado para todas las situaciones, pero funciona en la mayoría, y ahorra mucho esfuerzo al implementar la propia versión de `isapprox` .

Lea Examen de la unidad en línea: https://riptutorial.com/es/julia-lang/topic/5632/examen-de-la-unidad

Capítulo 14: Expresiones

Examples

Introducción a las expresiones

Las expresiones son un tipo específico de objeto en Julia. Puede pensar que una expresión representa un fragmento de código Julia que aún no se ha evaluado (es decir, ejecutado). Luego hay funciones y operaciones específicas, como `eval()` que evaluarán la expresión.

Por ejemplo, podríamos escribir un guión o ingresar al intérprete lo siguiente: julia> 1 + 1 2

Una forma de crear una expresión es mediante la sintaxis `:()` . Por ejemplo:

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

Ahora tenemos un objeto de tipo `Expr` . Una vez que se acaba de formar, no hace nada, solo se sienta como cualquier otro objeto hasta que se actúe. En este caso, podemos *evaluar* esa expresión usando la función `eval()` :

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

Así, vemos que los dos siguientes son equivalentes:

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

¿Por qué querríamos pasar por una sintaxis mucho más complicada en `eval(:(1+1))` si solo queremos encontrar qué es igual a 1 + 1? La razón básica es que podemos definir una expresión en un punto de nuestro código, potencialmente modificarla más tarde y luego evaluarla en un punto posterior. Esto puede potencialmente abrir nuevas y poderosas capacidades al programador Julia. Las expresiones son un componente clave de la metaprogramación en Julia.

Creando expresiones

Hay varios métodos diferentes que se pueden usar para crear el mismo tipo de expresión. Las expresiones introducidas mencionan la sintaxis `:()` . Quizás el mejor lugar para comenzar, sin embargo, es con cuerdas. Esto ayuda a revelar algunas de las similitudes fundamentales entre las expresiones y las cadenas en Julia.

Crear Expresión desde Cadena

De la documentación de Julia:

> Cada programa de Julia comienza la vida como una cadena.

En otras palabras, cualquier script de Julia simplemente se escribe en un archivo de texto, que no es más que una cadena de caracteres. Del mismo modo, cualquier comando de Julia ingresado en un intérprete es solo una cadena de caracteres. El rol de Julia o cualquier otro lenguaje de programación es interpretar y evaluar cadenas de caracteres de una manera lógica y predecible, de modo que esas cadenas de caracteres puedan usarse para describir lo que el programador desea que la computadora realice.

Por lo tanto, una forma de crear una expresión es usar la función `parse()` aplicada a una cadena. La siguiente expresión, una vez evaluada, asignará el valor de 2 al símbolo `x` .

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

Crear expresión usando `:()` Sintaxis

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

Tenga en cuenta que con esta sintaxis, Julia tratará automáticamente los nombres de los objetos como referentes a símbolos. Podemos ver esto si miramos los `args` de la expresión. (Consulte Campos de objetos de expresión para obtener más detalles sobre el campo `args` en una expresión).

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

Crear expresión usando la función `Expr()`

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

Esta sintaxis se basa en la notación de prefijo . En otras palabras, el primer argumento de la función `Expr()` especificada es el `head` o el prefijo. Los restantes son los `arguments` de la expresión. El `head` determina qué operaciones se realizarán en los argumentos.

Para más detalles sobre esto, vea Campos de objetos de expresión.

Al usar esta sintaxis, es importante distinguir entre el uso de objetos y símbolos para objetos. Por ejemplo, en el ejemplo anterior, la expresión asigna el valor de `2` al símbolo `:x` , una operación perfectamente sensible. Si usáramos `x` en una expresión como esa, obtendríamos el resultado sin

sentido:

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

Del mismo modo, si examinamos los `args` vemos:

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

Por lo tanto, la función `Expr()` no realiza la misma transformación automática en símbolos que la sintaxis `:()` para crear expresiones.

Crea expresiones multilínea usando `quote...end`

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

Tenga en cuenta que con `quote...end` podemos crear expresiones que contengan otras expresiones en su campo `args` :

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

Consulte Campos de objetos de expresión para obtener más información sobre este campo `args` .

Más sobre la creación de expresiones

Este ejemplo solo proporciona los conceptos básicos para crear expresiones. Consulte también, por ejemplo, Interpolación y expresiones y Campos de objetos de expresión para obtener más información sobre cómo crear expresiones más complejas y avanzadas.

Campos de objetos de expresión

Como se menciona en las expresiones de Introducción a Expresiones , son un tipo específico de objeto en Julia. Como tales, tienen campos. Los dos campos más utilizados de una expresión son su `head` y sus `args` . Por ejemplo, considere la expresión

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

discutido en la creación de expresiones . Podemos ver la `head` y `args` siguiente manera:

```
julia> MyExpr3.head
```

```
:(=)

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

Las expresiones se basan en la notación de prefijo . Como tal, el `head` generalmente especifica la operación que se realizará en los `args` . La cabeza tiene que ser de Julia `Symbol` .

Cuando una expresión es para asignar un valor (cuando se evalúa), generalmente usará un encabezado de `:(=)` . Por supuesto, hay variaciones obvias a esto que se pueden emplear, por ejemplo:

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

: llamada para cabezas de expresión

Otra `head` común para las expresiones es `:call` . P.ej

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

Siguiendo las convenciones de la notación de prefijo, los operadores se evalúan de izquierda a derecha. Por lo tanto, esta expresión aquí significa que llamaremos la función que se especifica en el primer elemento de `args` en los elementos posteriores. Igualmente podríamos tener:

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

U otras funciones potencialmente más interesantes, por ejemplo,

```
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
```

Determinación automática de la `head` cuando se utiliza `:()` notación de creación de expresión

Tenga en cuenta que `:call` se usa implícitamente como encabezado en ciertas construcciones de expresiones, por ejemplo,

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

Por lo tanto, con la sintaxis `:()` para crear expresiones, Julia buscará determinar automáticamente

la cabeza correcta a usar. Similar:

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

De hecho, si no está seguro de cuál es la cabeza correcta que debe usar para una expresión que está formando, por ejemplo, `Expr()` esta puede ser una herramienta útil para obtener sugerencias e ideas sobre qué usar.

Interpolación y Expresiones

La creación de expresiones menciona que las expresiones están estrechamente relacionadas con las cadenas. Como tales, los principios de interpolación dentro de las cadenas también son relevantes para las expresiones. Por ejemplo, en la interpolación de cadenas básica, podemos tener algo como:

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

Usamos el signo `$` para insertar el valor de `n` en la cadena. Podemos usar la misma técnica con expresiones. P.ej

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

Contraste esto este:

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

Así, con el primer ejemplo, establecemos por adelantado el valor de `a` que se utilizará en el momento en que se evalúa la expresión. Con el segundo ejemplo, sin embargo, el compilador Julia solo se ve a `a` de encontrar su valor *en el momento de la evaluación* de nuestra expresión.

Referencias externas sobre expresiones

Hay una serie de recursos web útiles que pueden ayudarlo a mejorar su conocimiento de las expresiones en Julia. Éstos incluyen:

- Julia Docs - Metaprogramación
- Wikilibros - metaprogramación de julia
- Macros, expresiones, etc. de Julia para y por los confundidos, por Gray Calhoun

- Mes de Julia - Metaprogramación, por Andrew Collier
- Diferenciación simbólica en Julia, por John Myles White

SO Publicaciones:

- ¿Qué es un "símbolo" en Julia? Respuestas de Stefan Karpinski
- ¿Por qué Julia expresa esta expresión de esta manera compleja?
- Explicación del ejemplo de interpolación de expresiones de Julia.

Lea Expresiones en línea: https://riptutorial.com/es/julia-lang/topic/5805/expresiones

Capítulo 15: Funciones

Sintaxis

- f (n) = ...
- función f (n) ... fin
- n :: Tipo
- x -> ...
- f (n) do ... end

Observaciones

Aparte de las funciones genéricas (que son las más comunes), también hay funciones incorporadas. Tales funciones incluyen `is` , `isa` , `typeof` , `throw` y funciones similares. Las funciones incorporadas normalmente se implementan en C en lugar de Julia, por lo que no pueden especializarse en tipos de argumentos para el envío.

Examples

Cuadrar un número

Esta es la sintaxis más fácil para definir una función:

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

Para llamar a una función, use corchetes (sin espacios entre ellos):

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

Las funciones son objetos en Julia, y podemos mostrarlas en el REPL como con cualquier otro objeto:

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

Todas las funciones de Julia son genéricas (también conocidas como polimórficas) por defecto. Nuestra función `square` funciona igual de bien con valores de punto flotante:

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

... o incluso matrices :

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

Funciones recursivas

Recursion simple

Usando la recursión y el operador condicional ternario , podemos crear una implementación alternativa de la función `factorial` incorporada:

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

Uso:

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

Trabajando con arboles

Las funciones recursivas son a menudo más útiles en estructuras de datos, especialmente en estructuras de datos de árbol. Como las expresiones en Julia son estructuras de árbol, la recursión puede ser bastante útil para la metaprogramación . Por ejemplo, la siguiente función reúne un conjunto de todas las cabezas utilizadas en una expresión.

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

Podemos comprobar que nuestra función funciona según lo previsto:

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

Esta función es compacta y utiliza una variedad de técnicas más avanzadas, como la función de `reduce` orden superior , el tipo de datos `Set` y las expresiones del generador.

Introducción al Despacho

Podemos usar la `::` sintaxis para enviar el tipo de argumento.

```
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 diferencia de muchos idiomas, que normalmente proporcionan un envío múltiple estático o un envío único dinámico, Julia tiene un envío múltiple dinámico completo. Es decir, las funciones pueden ser especializadas para más de un argumento. Esto resulta útil cuando se definen métodos especializados para operaciones en ciertos tipos y métodos de reserva para otros tipos.

```
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"
```

Argumentos opcionales

Julia permite que las funciones tomen argumentos opcionales. Detrás de escena, esto se implementa como otro caso especial de despacho múltiple. Por ejemplo, resolvamos el problema popular de Fizz Buzz . Por defecto, lo haremos para números en el rango de `1:10` , pero permitiremos un valor diferente si es necesario. También permitiremos que se usen diferentes frases para `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
```

Si inspeccionamos `fizzbuzz` en el REPL, dice que hay cuatro métodos. Se creó un método para cada combinación de argumentos permitidos.

```
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
```

Podemos verificar que nuestros valores predeterminados se utilizan cuando no se proporcionan parámetros:

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

pero que los parámetros opcionales son aceptados y respetados si los proveemos:

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

Despacho paramétrico

Es frecuente que una función deba enviarse en tipos paramétricos, como `Vector{T}` o `Dict{K,V}`, pero los parámetros de tipo no son fijos. Este caso puede tratarse mediante el envío paramétrico:

```
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"
```

Uno puede tener la tentación de simplemente escribir `xs::Vector{Number}` . Pero esto solo funciona para objetos cuyo tipo es explícitamente `Vector{Number}` :

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

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

Esto se debe a la invariancia paramétrica : el objeto `Int[1, 2]` *no* es un `Vector{Number}` , porque solo puede contener `Int` s, mientras que un `Vector{Number}` podría contener cualquier tipo de número.

Escribir código genérico

El envío es una característica increíblemente poderosa, pero a menudo es mejor escribir código genérico que funcione para todos los tipos, en lugar de un código especializado para cada tipo. Escribir código genérico evita la duplicación de código.

Por ejemplo, aquí está el código para calcular la suma de cuadrados de un vector de enteros:

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

Pero este código *solo* funciona para un vector de `Int` s. No funcionará en 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
```

No funcionará en 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
```

Una mejor manera de escribir esta función `sumsq` debería ser

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

Esto funcionará en los dos casos mencionados anteriormente. Pero hay algunas colecciones de las que podríamos querer sumar los cuadrados que no son vectores, en ningún sentido. Por ejemplo,

```
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
```

Demuestra que no podemos sumar los cuadrados de un perezoso iterable .

Una implementación aún más genérica es simplemente

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

Que funciona en todos los casos:

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

Este es el código de Julia más idiomático, y puede manejar todo tipo de situaciones. En algunos otros idiomas, la eliminación de anotaciones de tipo puede afectar el rendimiento, pero ese no es el caso en Julia; Sólo la estabilidad del tipo es importante para el rendimiento.

Factorial imperativo

Hay una sintaxis de formato largo disponible para definir funciones multilínea. Esto puede ser útil cuando usamos estructuras imperativas como los bucles. Se devuelve la expresión en posición cola. Por ejemplo, la siguiente función utiliza un bucle `for` para calcular el factorial de algún entero `n` :

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

```
        fact *= m
    end
    fact
end
```

Uso:

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

En funciones más largas, es común ver la declaración de `return` utilizada. La declaración de `return` no es necesaria en la posición de cola, pero a veces todavía se utiliza para mayor claridad. Por ejemplo, otra forma de escribir la función anterior sería

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

que es idéntico en comportamiento a la función anterior.

Funciones anonimas

Sintaxis de flecha

Se pueden crear funciones anónimas usando la sintaxis `->` . Esto es útil para pasar funciones a funciones de orden superior , como la función de `map` . La siguiente función calcula el cuadrado de cada número en una matriz `A`

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

Un ejemplo del uso de esta función:

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

Sintaxis multilínea

Las funciones anónimas multilínea se pueden crear utilizando la sintaxis de la `function` . Por ejemplo, el siguiente ejemplo calcula los factoriales de los primeros `n` números, pero utilizando una función anónima en lugar del `factorial` incorporado.

```
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
```

Hacer bloque de sintaxis

Debido a que es tan común pasar una función anónima como el primer argumento a una función, hay una sintaxis de bloque `do` . La sintaxis

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

es equivalente a

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

pero el primero puede ser más claro en muchas situaciones, especialmente si se realizan muchos cálculos en la función anónima. `do` sintaxis de bloques es especialmente útil para la entrada y salida de archivos por razones de administración de recursos.

Lea Funciones en línea: https://riptutorial.com/es/julia-lang/topic/3079/funciones

Capítulo 16: Funciones de orden superior

Sintaxis

- foreach (f, xs)
- mapa (f, xs)
- filtro (f, xs)
- reducir (f, v0, xs)
- foldl (f, v0, xs)
- foldr (f, v0, xs)

Observaciones

Las funciones pueden aceptarse como parámetros y también pueden producirse como tipos de retorno. De hecho, las funciones pueden crearse dentro del cuerpo de otras funciones. Estas funciones internas son conocidas como cierres .

Examples

Funciones como argumentos

Las funciones son objetos en julia. Como cualquier otro objeto, se pueden pasar como argumentos a otras funciones. Las funciones que aceptan funciones se conocen como funciones de orden superior .

Por ejemplo, podemos implementar un equivalente de la función `foreach` la biblioteca estándar tomando una función `f` como primer parámetro.

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

Podemos probar que esta función funciona como esperamos:

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

Al tomar una función como *primer* parámetro, en lugar de un parámetro posterior, podemos usar la sintaxis de bloque do de Julia. La sintaxis del bloque do es solo una forma conveniente de pasar una función anónima como el primer argumento a una función.

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

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

Nuestra implementación de `myforeach` anterior es aproximadamente equivalente a la función `foreach` incorporada. También existen muchas otras funciones incorporadas de orden superior.

Las funciones de orden superior son bastante poderosas. A veces, cuando se trabaja con funciones de orden superior, las operaciones exactas que se realizan pierden importancia y los programas pueden volverse bastante abstractos. Los combinadores son ejemplos de sistemas de funciones altamente abstractas de orden superior.

Mapa, filtro y reducción

Dos de las funciones de orden superior más fundamentales incluidas en la biblioteca estándar son `map` y `filter` . Estas funciones son genéricas y pueden operar en cualquier iterable . En particular, son muy adecuados para los cálculos en matrices .

Supongamos que tenemos un conjunto de datos de escuelas. Cada escuela enseña una materia en particular, tiene un número de clases y un número promedio de estudiantes por clase. Podemos modelar una escuela con el siguiente tipo inmutable :

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

Nuestro conjunto de datos de escuelas será un `Vector{School}` :

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

Supongamos que deseamos encontrar la cantidad de estudiantes en total inscritos en un programa de matemáticas. Para ello, requerimos varios pasos:

- debemos restringir el conjunto de datos solo a las escuelas que enseñan matemáticas (`filter`)
- debemos calcular el número de estudiantes en cada escuela (`map`)
- y debemos reducir esa lista de números de estudiantes a un solo valor, la suma (`reduce`)

Una solución ingenua (que no tiene el mejor rendimiento) sería simplemente usar esas tres funciones de orden superior directamente.

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

y verificamos que hay 190 estudiantes de matemáticas en nuestro conjunto de datos:

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

Existen funciones para combinar estas funciones y así mejorar el rendimiento. Por ejemplo, podríamos haber utilizado la función `mapreduce` para realizar el mapeo y la reducción en un solo paso, lo que ahorraría tiempo y memoria.

La `reduce` solo es significativa para operaciones asociativas como `+` , pero en ocasiones es útil realizar una reducción con una operación no asociativa. Las funciones de orden superior `foldl` y `foldr` se proporcionan para forzar un orden de reducción particular.

Lea Funciones de orden superior en línea: https://riptutorial.com/es/julia-lang/topic/6955/funciones-de-orden-superior

Capítulo 17: Hora

Sintaxis

- ahora()
- Fechas.Hoy ()
- Fechas.Año (t)
- Fechas. Mes (t)
- Fechas.day (t)
- Fechas.hora (t)
- Fechas.minuto (t)
- Fechas. Segundo (t)
- Dates.millisecond (t)
- Fechas. Formato (t, s)

Examples

Tiempo actual

Para obtener la fecha y hora actual, use la función `now` :

```
julia> now()
2016-09-04T00:16:58.122
```

Esta es la hora local, que incluye la zona horaria configurada de la máquina. Para obtener la hora en la zona horaria de hora universal coordinada (UTC) , use `now(Dates.UTC)` :

```
julia> now(Dates.UTC)
2016-09-04T04:16:58.122
```

Para obtener la fecha actual, sin la hora, use `today()` :

```
julia> Dates.today()
2016-10-30
```

El valor de retorno de `now` es un objeto `DateTime` . Hay funciones para obtener los componentes individuales de 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
```

Es posible formatear un `DateTime` usando una cadena de formato especialmente formateada:

```
julia> Dates.format(t, "yyyy-mm-dd at HH:MM:SS")
"2016-09-04 at 00:16:58"
```

Dado que muchas de las funciones de `Dates` se exportan desde el módulo `Base.Dates` , puede guardar algo de escritura para escribir

```
using Base.Dates
```

que luego permite acceder a las funciones calificadas arriba sin las `Dates.` calificación.

Lea Hora en línea: https://riptutorial.com/es/julia-lang/topic/5812/hora

Capítulo 18: Instrumentos de cuerda

Sintaxis

- "[cuerda]"
- '[Valor escalar de Unicode]'
- grafemas ([string])

Parámetros

Parámetro	Detalles
por	`sprint(f, xs...)`
`f`	Una función que toma un objeto `IO` como su primer argumento.
`xs`	Cero o más argumentos restantes para pasar a `f` .

Examples

¡Hola Mundo!

Las cadenas en Julia se delimitan usando el símbolo `"` :

```
julia> mystring = "Hello, World!"
"Hello, World!"
```

Tenga en cuenta que a diferencia de otros idiomas, el símbolo `'` *no se* puede utilizar en su lugar. `'` define un *carácter literal* ; este es un tipo de datos `Char` y solo almacenará un único valor escalar Unicode :

```
julia> 'c'
'c'

julia> 'character'
ERROR: syntax: invalid character literal
```

Uno puede extraer los valores escalares de Unicode de una cadena iterando sobre ella con un bucle `for` :

```
julia> for c in "Hello, World!"
           println(c)
       end
H
e
l
```

```
l
o
,

W
o
r
l
d
!
```

Grafemas

El tipo `Char` de Julia representa un valor escalar de Unicode , que solo en algunos casos corresponde a lo que los humanos perciben como un "personaje". Por ejemplo, una representación del carácter é, como en el resumen, es en realidad una combinación de dos valores escalares de Unicode:

```
julia> collect("é ")
2-element Array{Char,1}:
 'e'
 ' '
```

Las descripciones de Unicode para estos puntos de código son "LATINA PEQUEÑA LETINA E" y "COMBINACIÓN DE ACOMISTA AGUDA". Juntos, definen un único carácter "humano", que en términos de Unicode se denomina grafema . Más específicamente, el Anexo # 29 de Unicode motiva la definición de un grupo de grafemas porque:

> Es importante reconocer que lo que el usuario piensa como un "carácter", una unidad básica de un sistema de escritura para un idioma, puede que no sea un solo punto de código Unicode. En su lugar, esa unidad básica puede estar formada por múltiples puntos de código Unicode. Para evitar la ambigüedad con el uso de la computadora del término carácter, esto se llama un carácter percibido por el usuario. Por ejemplo, "G" + acento agudo es un carácter percibido por el usuario: los usuarios lo consideran como un solo carácter, aunque en realidad está representado por dos puntos de código Unicode. Estos caracteres percibidos por el usuario se aproximan a lo que se llama un grupo de grafemas, que se puede determinar mediante programación.

Julia proporciona la función de `graphemes` para iterar sobre los grupos de grafemas en una cadena:

```
julia> for c in graphemes("ré sumé ")
           println(c)
       end
r
é
s
u
m
é
```

Observe cómo el resultado, imprimir cada carácter en su propia línea, es mejor que si hubiéramos

iterado sobre los valores escalares de Unicode:

```
julia> for c in "ré sumé "
           println(c)
       end
r
e

s
u
m
e
```

Normalmente, cuando se trabaja con caracteres en un sentido percibido por el usuario, es más útil tratar con grupos de grafemas que con valores escalares de Unicode. Por ejemplo, supongamos que queremos escribir una función para calcular la longitud de una sola palabra. Una solución ingenua sería utilizar

```
julia> wordlength(word) = length(word)
wordlength (generic function with 1 method)
```

Notamos que el resultado es contraintuitivo cuando la palabra incluye grupos de grafemas que constan de más de un punto de código:

```
julia> wordlength("ré sumé ")
8
```

Cuando usamos la definición más correcta, usando la función de `graphemes` , obtenemos el resultado esperado:

```
julia> wordlength(word) = length(graphemes(word))
wordlength (generic function with 1 method)

julia> wordlength("ré sumé ")
6
```

Convertir tipos numéricos a cadenas

Existen numerosas formas de convertir tipos numéricos a cadenas en Julia:

```
julia> a = 123
123

julia> string(a)
"123"

julia> println(a)
123
```

La función `string()` también puede tomar más argumentos:

```
julia> string(a, "b")
"123b"
```

También puede insertar (también conocido como interpolar) enteros (y algunos otros tipos) en cadenas usando `$` :

```
julia> MyString = "my integer is $a"
"my integer is 123"
```

Consejo de rendimiento: los métodos anteriores pueden ser bastante convenientes a veces. Pero, si va a realizar muchas, muchas de estas operaciones y le preocupa la velocidad de ejecución de su código, la guía de rendimiento de Julia recomienda esto y, en cambio, favorece los siguientes métodos:

Puede suministrar múltiples argumentos para `print()` e `println()` que funcionarán en ellos exactamente como la `string()` opera en múltiples argumentos:

```
julia> println(a, "b")
123b
```

O, al escribir en un archivo, también puede utilizar, por ejemplo,

```
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)
```

Estos son más rápidos porque evitan la necesidad de formar primero una cadena a partir de piezas dadas y luego emitirlas (ya sea a la pantalla de la consola o a un archivo) y, en cambio, solo generar de forma secuencial las distintas piezas.

Créditos: Respuesta basada en SO Pregunta ¿Cuál es la mejor manera de convertir un Int en una cadena en Julia? con respuesta por Michael Ohlrogge y entrada de Fengyang Wang

Interpolación de cadenas (insertar el valor definido por la variable en la cadena)

En Julia, como en muchos otros idiomas, es posible interpolar insertando valores definidos por variables en cadenas. Para un ejemplo simple:

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

Podemos usar otros tipos que no sean numéricos, por ej.

```
Result = false
julia> println("test results is $Result")
test results is false
```

Puedes tener múltiples interpolaciones dentro de una cadena dada:

```
MySubStr = "a32"
MyNum = 123.31
println("$MySubStr  ,   $MyNum")
```

Interpolación de **consejos de rendimiento** es bastante conveniente. Pero, si lo va a hacer muchas veces muy rápidamente, no es lo más eficiente. En su lugar, consulte Convertir tipos numéricos en cadenas para obtener sugerencias cuando el rendimiento es un problema.

Uso de sprint para crear cadenas con funciones IO

Las cadenas se pueden hacer a partir de funciones que funcionan con objetos `IO` utilizando la función `sprint` . Por ejemplo, la función `code_llvm` acepta un objeto `IO` como primer argumento. Típicamente, se usa como

```
julia> code_llvm(STDOUT, *, (Int, Int))

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
  %2 = mul i64 %1, %0
  ret i64 %2
}
```

Supongamos que queremos esa salida como una cadena en su lugar. Entonces podemos simplemente hacer

```
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
}
```

Convertir los resultados de funciones "interactivas" como `code_llvm` en cadenas puede ser útil para el análisis automatizado, como probar si el código generado puede haber retrocedido.

La función `sprint` es una función de orden superior que toma la función que opera en objetos `IO` como su primer argumento. Detrás de escena, crea un `IOBuffer` en la RAM, llama a la función dada y toma los datos del búfer en un objeto `String` .

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/julia-lang/topic/5562/instrumentos-de-cuerda

Capítulo 19: Iterables

Sintaxis

- empezar (itr)
- siguiente (itr, s)
- hecho (itr, s)
- tomar (itr, n)
- soltar (itr, n)
- ciclo (itr)
- Producto base (xs, ys)

Parámetros

Parámetro	Detalles
por	**Todas las funciones**
`itr`	Lo iterable para operar.
por	**`next` y `done`**
`s`	Un estado de iterador que describe la posición actual de la iteración.
por	**`take` y `drop`**
`n`	El número de elementos a tomar o soltar.
por	**`Base.product`**
`xs`	Lo iterable para sacar los primeros elementos de las parejas.
`ys`	Lo iterable para tomar segundos elementos de pares.
...	(Tenga en cuenta que el `product` acepta cualquier número de argumentos; si se proporcionan más de dos, se construirán tuplas de longitud mayor que dos).

Examples

Nuevo tipo iterable

En Julia, cuando se realiza un bucle a través de un objeto iterable `I` se hace con la sintaxis `for` :

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

```
end
```

Detrás de escena, esto se traduce a:

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

Por lo tanto, si desea `I` sea un iterable, debe definir los métodos de `start` , `next` y `done` para su tipo. Supongamos que define un tipo `Foo` contiene una matriz como uno de los campos:

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

Creamos una instancia de un objeto `Foo` haciendo:

```
julia> I = Foo([1,2,3])
Foo([1,2,3])

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

Si queremos iterar a través de `Foo` , con cada `bar` elementos que devuelve cada iteración, definimos los métodos:

```
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
```

Tenga en cuenta que dado que estas funciones pertenecen al módulo `Base` , primero debemos `import` sus nombres antes de agregarles nuevos métodos.

Después de definir los métodos, `Foo` es compatible con la interfaz del iterador:

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

```
1
2
3
```

Combinando Lazy Iterables

La biblioteca estándar viene con una rica colección de iterables (y bibliotecas como Iterators.jl proporcionan aún más). Los iterables perezosos se pueden componer para crear iterables más potentes en tiempo constante. Los iterables perezosos más importantes son tomar y soltar , a partir de los cuales se pueden crear muchas otras funciones.

Perezamente cortar una iterable

Los arreglos pueden ser cortados con notación de corte. Por ejemplo, lo siguiente devuelve los elementos del 10 al 15 de una matriz, inclusive:

```
A[10:15]
```

Sin embargo, la notación de segmento no funciona con todos los iterables. Por ejemplo, no podemos cortar una expresión del generador:

```
julia> (i^2 for i in 1:10)[3:5]
ERROR: MethodError: no method matching getindex(::Base.Generator{UnitRange{Int64},##1#2},
::UnitRange{Int64})
```

Cortar cadenas puede no tener el comportamiento esperado de Unicode:

```
julia> "aaaa"[2:3]
ERROR: UnicodeError: invalid character index
 in getindex(::String, ::UnitRange{Int64}) at ./strings/string.jl:130

julia> "aaaa"[3:4]
"a"
```

Podemos definir una función `lazysub(itr, range::UnitRange)` para hacer este tipo de corte en iterables arbitrarios. Esto se define en términos de `take` y `drop` :

```
lazysub(itr, r::UnitRange) = take(drop(itr, first(r) - 1), last(r) - first(r) + 1)
```

La implementación aquí funciona porque para el valor de `UnitRange a:b` , se realizan los siguientes pasos:

- Deja caer los primeros elementos a `a-1`
- toma la `a` ésimo elemento, `a+1` -ésimo elemento, y así sucesivamente, hasta que el `a+(ba)=b` -ésimo elemento

En total, se toman elementos `ba` . Podemos confirmar que nuestra implementación es correcta en cada caso anterior:

```
julia> collect(lazysub("αααα", 2:3))
2-element Array{Char,1}:
 'α'
 'α'

julia> collect(lazysub((i^2 for i in 1:10), 3:5))
3-element Array{Int64,1}:
  9
 16
 25
```

Cambia perezosamente un iterable circularmente

La operación de cambio de `circshift` en los arreglos cambiará el arreglo como si fuera un círculo, y luego lo volverá a alinear. Por ejemplo,

```
julia> circshift(1:10, 3)
10-element Array{Int64,1}:
  8
  9
 10
  1
  2
  3
  4
  5
  6
  7
```

¿Podemos hacer esto perezosamente para todos los iterables? Podemos usar el `cycle` , `drop` y `take` iterables para implementar esta funcionalidad.

```
lazycircshift(itr, n) = take(drop(cycle(itr), length(itr) - n), length(itr))
```

Además de que los tipos perezosos son más `circshift` en muchas situaciones, esto nos permite realizar `circshift` función similar a la de `circshift` en tipos que de otro modo no lo `circshift` :

```
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

Haciendo una tabla de multiplicar

Hagamos una tabla de multiplicar usando funciones iterables para crear una matriz.

Las funciones clave a utilizar aquí son:

- `Base.product` , que calcula un producto cartesiano .
- `prod` , que calcula un producto regular (como en la multiplicación)
- `:` , lo que crea un rango
- `map` , que es una función de orden superior que aplica una función a cada elemento de una colección.

La solucion es:

```
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
```

Listas de pereza evaluadas

Es posible hacer una lista simple y perezosamente evaluada utilizando tipos y cierres mutables. Una lista evaluada perezosamente es una lista cuyos elementos no se evalúan cuando se construye, sino cuando se accede a ella. Los beneficios de las listas evaluadas perezosamente incluyen la posibilidad de ser infinitas.

```
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()
```

Lo que de hecho funciona como lo haría en un lenguaje como Haskell , donde todas las listas se evalúan perezosamente:

```
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
```

En la práctica, es mejor usar el paquete Lazy.jl. Sin embargo, la implementación de la lista perezosa de arriba arroja luces sobre detalles importantes sobre cómo construir el propio tipo de iterable.

Lea Iterables en línea: https://riptutorial.com/es/julia-lang/topic/5466/iterables

Capítulo 20: JSON

Sintaxis

- usando JSON
- JSON.parse (str)
- JSON.json (obj)
- JSON.print (io, obj, sangría)

Observaciones

Dado que ni Julia `Dict` ni los objetos JSON están intrínsecamente ordenados, es mejor no confiar en el orden de los pares clave-valor en un objeto JSON.

Examples

Instalando JSON.jl

JSON es un popular formato de intercambio de datos. La biblioteca JSON más popular para Julia es JSON.jl. Para instalar este paquete, use el administrador de paquetes:

```
julia> Pkg.add("JSON")
```

El siguiente paso es probar si el paquete está funcionando en su máquina:

```
julia> Pkg.test("JSON")
```

Si todas las pruebas pasaron, entonces la biblioteca está lista para su uso.

Analizando JSON

JSON que se ha codificado como una cadena se puede analizar fácilmente en un tipo de Julia estándar:

```
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["is","json"]
  "numbers" => Any[85,16,12.0]
  "and"     => Any[true,false,nothing]
```

Hay algunas propiedades inmediatas de JSON.jl de la nota:

- Los tipos JSON se asignan a tipos sensibles en Julia: el objeto se convierte en `Dict` , la matriz se convierte en `Vector` , el número se convierte en `Int64` o `Float64` , el booleano se convierte en `Bool` y el valor nulo se convierte en `nothing::Void` .
- JSON es un formato de contenedor sin tipo: por lo tanto, los vectores Julia devueltos son de tipo `Vector{Any}` , y los diccionarios devueltos son de tipo `Dict{String, Any}` .
- El estándar JSON no distingue entre números enteros y decimales, pero JSON.jl lo hace. Un número sin un punto decimal o una notación científica se analiza en `Int64` , mientras que un número con un punto decimal se analiza en `Float64` . Esto coincide estrechamente con el comportamiento de los analizadores JSON en muchos otros idiomas.

Serialización JSON

La función `JSON.json` serializa un objeto Julia en una `String` Julia que contiene 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}
```

Si no se desea una cadena, JSON se puede imprimir directamente en una secuencia de IO:

```
julia> JSON.print(STDOUT, [1, 2, true, false, "x"])
[1,2,true,false,"x"]
```

Tenga en cuenta que `STDOUT` es el valor predeterminado y se puede omitir en la llamada anterior.

La impresión más bonita se puede lograr al pasar el parámetro de `indent` opcional:

```
julia> JSON.print(STDOUT, Dict(:a => :b, :c => :d), 4)
{
    "c": "d",
    "a": "b"
}
```

Hay una serialización por defecto para los tipos complejos de Julia:

```
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
```

```
}
```

Lea JSON en línea: https://riptutorial.com/es/julia-lang/topic/5468/json

Capítulo 21: Leyendo un DataFrame desde un archivo

Examples

Lectura de un marco de datos a partir de datos separados por delimitadores

Es posible que desee leer un `DataFrame` de un archivo CSV (valores separados por comas) o incluso de un TSV o WSV (pestañas y archivos separados por espacios en blanco). Si el archivo tiene la extensión correcta, puede usar el `readtable` función para leer en la trama de datos:

```
readtable("dataset.CSV")
```

¿Pero qué pasa si su archivo no tiene la extensión correcta? Puede especificar el delimitador que utiliza su archivo (coma, tabulación, espacios en blanco, etc.) como un argumento de palabra clave para la función de `readtable` :

```
readtable("dataset.txt", separator=',')
```

Manejo de diferentes comentarios comentario marcas.

Los conjuntos de datos a menudo contienen comentarios que explican el formato de los datos o contienen los términos de licencia y uso. Por lo general, querrá ignorar estas líneas cuando lea en el `DataFrame` .

La función de `readtable` asume que las líneas de comentario comienzan con el carácter '#'. Sin embargo, su archivo puede usar marcas de comentario como `%` o `//` . Para asegurarse de que `readtable` maneja esto correctamente, puede especificar la marca de comentario como un argumento de palabra clave:

```
readtable("dataset.csv", allowcomments=true, commentmark='%')
```

Lea Leyendo un DataFrame desde un archivo en línea: https://riptutorial.com/es/julia-lang/topic/7340/leyendo-un-dataframe-desde-un-archivo

Capítulo 22: Los diccionarios

Examples

Usando Diccionarios

Los diccionarios se pueden construir pasándole cualquier número de pares.

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

Puede obtener entradas en un diccionario poniendo la clave entre corchetes.

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

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

Lea Los diccionarios en línea: https://riptutorial.com/es/julia-lang/topic/9028/los-diccionarios

Capítulo 23: Los tipos

Sintaxis

- mi tipo inmutable; campo; campo; fin
- escriba MyType; campo; campo; fin

Observaciones

Los tipos son clave para el desempeño de Julia. Una idea importante para el rendimiento es la estabilidad del tipo , que se produce cuando el tipo que devuelve una función solo depende de los tipos, no de los valores, de sus argumentos.

Examples

Despacho de tipos

En Julia, puedes definir más de un método para cada función. Supongamos que definimos tres métodos de la misma función:

```
foo(x) = 1
foo(x::Number) = 2
foo(x::Int) = 3
```

Al decidir qué método usar (llamado despacho), Julia elige el método más específico que coincida con los tipos de los argumentos:

```
julia> foo('one')
1

julia> foo(1.0)
2

julia> foo(1)
3
```

Esto facilita el polimorfismo . Por ejemplo, podemos crear fácilmente una lista enlazada definiendo dos tipos inmutables, denominados `Nil` y `Cons` . Estos nombres se usan tradicionalmente para describir una lista vacía y una lista no vacía, respectivamente.

```
abstract LinkedList
immutable Nil <: LinkedList end
immutable Cons <: LinkedList
    first
    rest::LinkedList
end
```

Representaremos la lista vacía con `Nil()` y cualquier otra lista con `Cons(first, rest)` , donde `first` es el primer elemento de la lista vinculada y el `rest` es la lista vinculada que consta de todos los elementos restantes. Por ejemplo, la lista `[1, 2, 3]` se representará como

```
julia> Cons(1, Cons(2, Cons(3, Nil())))
Cons(1,Cons(2,Cons(3,Nil())))
```

¿Está la lista vacía?

Supongamos que deseamos ampliar la función `isempty` la biblioteca estándar, que funciona en una variedad de colecciones diferentes:

```
julia> methods(isempty)
# 29 methods for generic function "isempty":
isempty(v::SimpleVector) at essentials.jl:180
isempty(m::Base.MethodList) at reflection.jl:394
...
```

Simplemente podemos utilizar la sintaxis de despacho de la función y definir dos métodos adicionales de `isempty` . Como esta función es del módulo `Base` , debemos calificarla como `Base.isempty` para poder extenderla.

```
Base.isempty(::Nil) = true
Base.isempty(::Cons) = false
```

Aquí, no necesitamos los valores de los argumentos para determinar si la lista está vacía. Simplemente el tipo solo basta para calcular esa información. Julia nos permite omitir los nombres de los argumentos, manteniendo solo su tipo de anotación, si no necesitamos usar sus valores.

Podemos probar que nuestros métodos `isempty` funcionan:

```
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())))))
```

y de hecho, el número de métodos para la `isempty` ha aumentado en `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
```

Claramente, determinar si una lista vinculada está vacía o no es un ejemplo trivial. Pero conduce a algo más interesante:

¿Cuánto dura la lista?

La función de `length` de la biblioteca estándar nos da la longitud de una colección o ciertos iterables . Hay muchas formas de implementar la `length` de una lista enlazada. En particular, el uso de un `while` de bucle es eficaz memoria de Julia probablemente más rápido y más. Pero debe evitarse la optimización prematura , por lo que supongamos por un segundo que nuestra lista enlazada no necesita ser eficiente. ¿Cuál es la forma más sencilla de escribir una función de `length` ?

```
Base.length(::Nil) = 0
Base.length(xs::Cons) = 1 + length(xs.rest)
```

La primera definición es sencilla: una lista vacía tiene una longitud de `0` . La segunda definición también es fácil de leer: para contar la longitud de una lista, contamos el primer elemento y luego contamos la longitud del resto de la lista. Podemos probar este método de manera similar a como probamos `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
```

Próximos pasos

Este ejemplo de juguete está bastante lejos de implementar toda la funcionalidad que se desearía en una lista vinculada. Falta, por ejemplo, la interfaz de iteración. Sin embargo, ilustra cómo se puede utilizar el envío para escribir código corto y claro.

Tipos inmutables

El tipo compuesto más simple es un tipo inmutable. Las instancias de tipos inmutables, como las tuplas , son valores. Sus campos no se pueden cambiar después de que se crean. En muchos sentidos, un tipo inmutable es como una `Tuple` con nombres para el tipo en sí y para cada campo.

Tipos singleton

Los tipos compuestos, por definición, contienen una serie de tipos más simples. En Julia, este número puede ser cero; es decir, un tipo inmutable *no puede* contener campos. Esto es comparable a la tupla vacía `()` .

¿Por qué podría ser útil? Tales tipos inmutables se conocen como "tipos singleton", ya que solo una instancia de ellos podría existir. Los valores de estos tipos se conocen como "valores

singleton". La `Base` biblioteca estándar contiene muchos de estos tipos de singleton. Aquí hay una breve lista:

- `Void` , el tipo de `nothing` . Podemos verificar que `Void.instance` (que es una sintaxis especial para recuperar el valor singleton de un tipo singleton) no es `nothing` .
- Cualquier tipo de medio, como `MIME"text/plain"` , es un tipo singleton con una sola instancia, `MIME("text/plain")` .
- `Irrational{:π}` , `Irrational{:e}` , `Irrational{:ψ}` y tipos similares son tipos singleton, y sus instancias singleton son los valores irracionales `π = 3.1415926535897...` , etc.
- Los rasgos de tamaño del iterador `Base.HasLength` , `Base.HasShape` , `Base.IsInfinite` y `Base.SizeUnknown` son todos tipos de singleton.

0.5.0

- ¡En la versión 0.5 y posteriores, cada función es una instancia de un tipo de singleton! Como cualquier otro valor de singleton, podemos recuperar la función `sin` , por ejemplo, de `typeof(sin).instance` .

Debido a que no contienen nada, los tipos singleton son increíblemente ligeros, y con frecuencia pueden ser optimizados por el compilador para que no tengan una sobrecarga de tiempo de ejecución. Por lo tanto, son perfectos para rasgos, valores de etiquetas especiales y para funciones como las que uno quisiera especializarse.

Para definir un tipo de singleton,

```
julia> immutable MySingleton end
```

Para definir la impresión personalizada para el tipo de singleton,

```
julia> Base.show(io::IO, ::MySingleton) = print(io, "sing")
```

Para acceder a la instancia de singleton,

```
julia> MySingleton.instance
MySingleton()
```

A menudo, uno asigna esto a una constante:

```
julia> const sing = MySingleton.instance
MySingleton()
```

Tipos de envoltura

Si los tipos inmutables de campo cero son interesantes y útiles, quizás los tipos inmutables de un campo sean aún más útiles. Tales tipos se denominan comúnmente "tipos de envoltura" porque envuelven algunos datos subyacentes, proporcionando una interfaz alternativa a dichos datos. Un ejemplo de un tipo de envoltorio en `Base` es `String` . Definiremos un tipo similar a `String` , llamado

`MyString` . Este tipo estará respaldado por un vector (matriz unidimensional) de bytes (`UInt8`).

Primero, la definición del tipo en sí y algunas demostraciones personalizadas:

```
immutable MyString <: AbstractString
    data::Vector{UInt8}
end

function Base.show(io::IO, s::MyString)
    print(io, "MyString: ")
    write(io, s.data)
    return
end
```

¡Ahora nuestro tipo `MyString` está listo para usar! Podemos proporcionarle algunos datos UTF-8 sin procesar, y se muestra como nos gusta:

```
julia> MyString([0x48,0x65,0x6c,0x6c,0x6f,0x2c,0x20,0x57,0x6f,0x72,0x6c,0x64,0x21])
MyString: Hello, World!
```

Obviamente, este tipo de cadena necesita mucho trabajo antes de que sea tan utilizable como el tipo `Base.String` .

Tipos compuestos verdaderos

Quizás más comúnmente, muchos tipos inmutables contienen más de un campo. Un ejemplo es el estándar de biblioteca `Rational{T}` tipo, que contiene dos fieds: a `num` campo para el numerador y un `den` campo para el denominador. Es bastante sencillo emular este tipo de diseño:

```
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)
```

Hemos implementado con éxito un constructor que simplifica nuestros números racionales:

```
julia> MyRational(10, 6)
MyRational{Int64}(5,3)
```

Lea Los tipos en línea: https://riptutorial.com/es/julia-lang/topic/5467/los-tipos

Capítulo 24: Macros de cadena

Sintaxis

- macro "cadena" # breve, forma de macro de cadena
- @macro_str "cadena" # larga, forma de macro regular
- macro`command`

Observaciones

Las macros de cadena no son tan poderosas como las cadenas simples, ya que la interpolación debe implementarse en la lógica de la macro, las macros de cadena no pueden contener literales de cadena del mismo delimitador para la interpolación.

Por ejemplo, aunque

```
julia> "$("x")"
"x"
```

funciona, la cadena de texto forma macro

```
julia> doc"$("x")"
ERROR: KeyError: key :x not found
```

se analiza incorrectamente. Esto se puede mitigar de alguna manera usando comillas triples como delimitador de cadena externo;

```
julia> doc"""$("x")"""
"x"
```

de hecho funciona correctamente.

Examples

Usando macros de cuerdas

Las macros de cadena son azúcar sintáctica para ciertas invocaciones de macro. El analizador expande la sintaxis como

```
mymacro"my string"
```

dentro

```
@mymacro_str "my string"
```

que luego, como cualquier otra llamada de macro, se sustituye por cualquier expresión que `@mymacro_str` macro `@mymacro_str` . Base Julia viene con varias macros de cuerdas, tales como:

`@b_str`

Esta macro de cadena construye matrices de bytes en lugar de cadenas . El contenido de la cadena, codificado como UTF-8, se utilizará como la matriz de bytes. Esto puede ser útil para interactuar con las API de bajo nivel, muchas de las cuales funcionan con matrices de bytes en lugar de cadenas.

```
julia> b"Hello World!"
12-element Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x20
 0x57
 0x6f
 0x72
 0x6c
 0x64
 0x21
```

`@big_str`

Esta macro devolverá un `BigInt` o un `BigFloat` analizado de la cadena que se proporciona.

```
julia> big"1"
1

julia> big"1.0"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Esta macro existe porque `big(0.1)` no se comporta como se podría esperar inicialmente: `0.1` es una aproximación de `Float64` de `0.1` (`1//10`), y promoverla a `BigFloat` mantendrá el error de aproximación de `Float64` . El uso de la macro analizará `0.1` directamente a un `BigFloat` , reduciendo el error de aproximación.

```
julia> big(0.1)
1.000000000000000055511151231257827021181583404541015625000000000000000000000000e-01

julia> big"0.1"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000002e-01
```

`@doc_str`

Esta macro de cadena construye objetos `Base.Markdown.MD` , que se utilizan en el sistema de documentación interno para proporcionar documentación de texto enriquecido para cualquier entorno. Estos objetos MD se reproducen bien en un terminal:

```
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.
```

y también en un navegador:

```
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

Esta macro de cadena construye literales de cadena HTML, que se reproducen bien en un navegador:

```
In [1]: html"""
        <p><abbr title="Hypertext Markup Language">HTML</abbr> text.</p>
        """
```

Out[1]: HTML text.

@ip_str

Esta macro de cadena construye literales de dirección IP. Funciona tanto con IPv4 como con IPv6:

```
julia> ip"127.0.0.1"
ip"127.0.0.1"

julia> ip"::"
ip"::"
```

@r_str

Esta macro de cadena construye `Regex` literales .

@s_str

Esta macro de cadena construye los literales `SubstitutionString` , que funcionan junto con los literales `Regex` para permitir una sustitución textual más avanzada.

@text_str

Esta macro de cadena es similar en espíritu a `@doc_str` y `@html_str` , pero no tiene ninguna característica de formato de fantasía:

```
In [3]: text"""
        This is some plain text.
        """
Out[3]: This is some plain text.
```

@v_str

Esta macro de cadena construye los literales de `VersionNumber` . Consulte Números de versión para obtener una descripción de qué son y cómo usarlos.

@MIME_str

Esta macro de cadena construye los tipos singleton de tipos MIME. Por ejemplo, `MIME"text/plain"` es el tipo de `MIME("text/plain")` .

Símbolos que no son identificadores legales

Los literales de Julia Symbol deben ser identificadores legales. Esto funciona:

```
julia> :cat
:cat
```

Pero esto no lo hace:

```
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,
::T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}) at int.jl:33
  *(::Real, ::Complex{Bool}) at complex.jl:180
  ...
```

Lo que parece un símbolo literal aquí en realidad se analiza como una multiplicación implícita de `:2` (que es solo `2`) y la función `cat` , que obviamente no funciona.

Nosotros podemos usar

```
julia> Symbol("2cat")
Symbol("2cat")
```

para solucionar el problema.

Una macro de cadena podría ayudar a hacer esto más conciso. Si definimos la macro `@sym_str` :

```
macro sym_str(str)
    Meta.quot(Symbol(str))
end
```

entonces podemos simplemente hacer

```
julia> sym"2cat"
Symbol("2cat")
```

Para crear símbolos que no sean identificadores de Julia válidos.

Por supuesto, estas técnicas también pueden crear símbolos que *son* identificadores de Julia válidos. Por ejemplo,

```
julia> sym"test"
:test
```

Implementando interpolación en una macro de cadena

Las macros de cadenas no vienen con instalaciones de interpolación incorporadas. Sin embargo, es posible implementar manualmente esta funcionalidad. Tenga en cuenta que no es posible incrustar sin escapar literales de cadena que tienen el mismo delimitador que la macro de cadena circundante; es decir, aunque `""" $("x") """` es posible, `" $("x") "` no lo es. En su lugar, esto debe escaparse como `" $(\"x\") "` . Consulte la sección de comentarios para obtener más detalles sobre esta limitación.

Hay dos enfoques para implementar la interpolación manualmente: implementar el análisis manualmente, o hacer que Julia haga el análisis. El primer enfoque es más flexible, pero el segundo es más fácil.

Análisis manual

```
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
```

Julia analizando

```
macro e_str(s)
    esc(parse("\"$(escape_string(s))\""))
end
```

Este método escapa de la cadena (pero tenga en cuenta que `escape_string` *no* escapa de los signos `$`) y se lo pasa al analizador de Julia para analizarlo. El escape de la cadena es necesario para garantizar que `"` y `\` no afecten el análisis de la cadena. La expresión resultante es una `:string` expresión de `:string` , que se puede examinar y descomponer para propósitos de macro.

Macros de comando

0.6.0-dev

En Julia v0.6 y versiones posteriores, las macros de comandos son compatibles además de las macros de cadena normales. Una invocación de macro de comando como

```
mymacro`xyz`
```

se analiza como la llamada macro

```
@mymacro_cmd "xyz"
```

Tenga en cuenta que esto es similar a las macros de cadenas, excepto con `_cmd` lugar de `_str` .

Utilizamos típicamente macros de comando de código, que en muchos idiomas con frecuencia contiene `"` pero rara vez contiene `` ` `` Por ejemplo, es bastante sencillo volver a implementar una versión simple de. Quasiquoting usando macros de comando:

```
macro julia_cmd(s)
    esc(Meta.quot(parse(s)))
end
```

Podemos usar esta macro ya sea en línea:

```
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 multilínea:

```
julia> julia```
       function hello()
           println("Hello, World!")
       end
       ```
:(function hello() # none, line 2:
        println("Hello, World!")
    end)
```

Se admite la interpolación usando `$`

```
julia> x = 2
2

julia> julia`1 + $x`
:(1 + 2)
```

pero la versión dada aquí solo permite una expresión:

```
julia> julia```
       x = 2
       y = 3
       ```
ERROR: ParseError("extra token after end of expression")
```

Sin embargo, extenderlo para manejar múltiples expresiones no es difícil.

Lea Macros de cadena en línea: https://riptutorial.com/es/julia-lang/topic/5817/macros-de-cadena

Capítulo 25: Metaprogramacion

Sintaxis

- nombre de macro (ex) ... fin
- cita ... fin
- : (...)
- $ x
- Meta.quot (x)
- QuoteNode (x)
- esc (x)

Observaciones

Las funciones de metaprogramación de Julia están muy inspiradas en las de lenguajes similares a Lisp, y les parecerán familiares a las personas con algún fondo Lisp. La metaprogramación es muy potente. Cuando se usa correctamente, puede llevar a un código más conciso y legible.

La `quote ... end` es sintaxis quasiquote. En lugar de las expresiones dentro de ser evaluadas, simplemente se analizan. El valor de la `quote ... end` expresión `quote ... end` es el árbol de sintaxis abstracta (AST) resultante.

La sintaxis `:(...)` es similar a la sintaxis de `quote ... end` , pero es más ligera. Esta sintaxis es más concisa que la `quote ... end` .

Dentro de una quasiquote, el operador `$` es especial e *interpola* su argumento en el AST. Se espera que el argumento sea una expresión que se empalma directamente en el AST.

La función `Meta.quot(x)` cita su argumento. Esto suele ser útil en combinación con el uso de `$` para la interpolación, ya que permite que las expresiones y los símbolos se empalen literalmente en el AST.

Examples

Reimplementando la macro @show

En Julia, la macro `@show` suele ser útil para fines de depuración. Muestra tanto la expresión a evaluar como su resultado, y finalmente devuelve el valor del resultado:

```
julia> @show 1 + 1
1 + 1 = 2
2
```

Es sencillo crear nuestra propia versión de `@show` :

```
julia> macro myshow(expression)
           quote
               value = $expression
               println($(Meta.quot(expression)), " = ", value)
               value
           end
       end
```

Para usar la nueva versión, simplemente use la macro `@myshow` :

```
julia> x = @myshow 1 + 1
1 + 1 = 2
2

julia> x
2
```

Hasta bucle

Todos estamos acostumbrados a la sintaxis `while` , que ejecuta su cuerpo mientras la condición se evalúa como `true` . ¿Qué `true` si queremos implementar un bucle `until` , que ejecuta un bucle hasta que la condición se evalúe como `true` ?

En Julia, podemos hacer esto creando una macro `@until` , que se detiene para ejecutar su cuerpo cuando se cumple la condición:

```
macro until(condition, expression)
    quote
        while !($condition)
            $expression
        end
    end |> esc
end
```

Aquí hemos utilizado la sintaxis de encadenamiento de funciones `|>` , que es equivalente a llamar a la función `esc` en todo el bloque de `quote` . La función `esc` evita que la macro higiene se aplique a los contenidos de la macro; sin él, las variables con ámbito en la macro serán renombradas para evitar colisiones con variables externas. Vea la documentación de Julia sobre macro higiene para más detalles.

Puede usar más de una expresión en este bucle, simplemente colocando todo dentro de un bloque `begin ... end` :

```
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 y Expr (: quote)

Hay tres formas de citar algo usando una función de Julia:

```
julia> QuoteNode(:x)
:(:x)

julia> Meta.quot(:x)
:(:x)

julia> Expr(:quote, :x)
:(:x)
```

¿Qué significa "citar" y para qué sirve? Las citas nos permiten proteger las expresiones para que no sean interpretadas como formas especiales por Julia. Un caso de uso común es cuando generamos expresiones que deben contener elementos que se evalúan como símbolos. (Por ejemplo, esta macro necesita devolver una expresión que se evalúe como un símbolo). No funciona simplemente para devolver el símbolo:

```
julia> macro mysym(); :x; end
@mysym (macro with 1 method)

julia> @mysym
ERROR: UndefVarError: x not defined

julia> macroexpand(:(@mysym))
:x
```

¿Que está pasando aqui? `@mysym` expande a `:x` , que como expresión se interpreta como la variable `x` . Pero aún no se ha asignado nada a `x` , por lo que obtenemos un error `x not defined` .

Para solucionar esto, debemos citar el resultado de nuestra macro:

```
julia> macro mysym2(); Meta.quot(:x); end
@mysym2 (macro with 1 method)

julia> @mysym2
:x

julia> macroexpand(:(@mysym2))
:(:x)
```

Aquí, hemos utilizado la función `Meta.quot` para convertir nuestro símbolo en un símbolo entre

comillas, que es el resultado que queremos.

¿Cuál es la diferencia entre `Meta.quot` y `QuoteNode` , y cuál debo usar? En casi todos los casos, la diferencia realmente no importa. Quizás sea un poco más seguro a veces usar `QuoteNode` lugar de `Meta.quot` . Sin embargo, explorar la diferencia es informativo sobre cómo funcionan las expresiones y macros de Julia.

La diferencia entre `Meta.quot` y `QuoteNode` , explicada

Aquí hay una regla de oro:

- Si necesita o desea admitir la interpolación, use `Meta.quot` ;
- Si no puede o no quiere permitir la interpolación, use `QuoteNode` .

En resumen, la diferencia es que `Meta.quot` permite la interpolación dentro de lo `Meta.quot` , mientras que `QuoteNode` protege su argumento de cualquier interpolación. Para entender la interpolación, es importante mencionar la expresión `$` . Hay un tipo de expresión en Julia llamada expresión `$` . Estas expresiones permiten escapar. Por ejemplo, considere la siguiente expresión:

```
julia> ex = :( x = 1; :($x + $x) )
quote
    x = 1
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

Cuando se evalúa, esta expresión evaluará `1` y lo asignará a `x` , luego construirá una expresión de la forma _ `+` _ donde _ se reemplazará por el valor de `x` . Por lo tanto, el resultado de esto debe ser la *expresión* `1 + 1` (que aún no se ha evaluado, y por lo tanto es distinta del *valor* `2`). De hecho, este es el caso:

```
julia> eval(ex)
:(1 + 1)
```

Digamos ahora que estamos escribiendo una macro para construir este tipo de expresiones. Nuestra macro tomará un argumento, que reemplazará el `1` en el `ex` anterior. Este argumento puede ser cualquier expresión, por supuesto. Aquí hay algo que no es exactamente lo que queremos:

```
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
```

El segundo caso es incorrecto, porque deberíamos mantener `1 + 1` evaluar. `Meta.quot` eso citando el argumento 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
```

La higiene macro no se aplica al contenido de una cotización, por lo que no es necesario escapar en este caso (y de hecho no es legal) en este caso.

Como se mencionó anteriormente, `Meta.quot` permite la interpolación. Así que vamos a intentarlo:

```
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
```

Del primer ejemplo, vemos que la interpolación nos permite alinear el `sin(1)` , en lugar de hacer que la expresión sea un `sin(1)` literal `sin(1)` . El segundo ejemplo muestra que esta interpolación se realiza en el ámbito de invocación de macros, no en el propio ámbito de la macro. Eso es porque nuestra macro no ha evaluado ningún código; todo lo que está haciendo es generar código. La evaluación del código (que se abre camino en la expresión) se realiza cuando la expresión que genera la macro se ejecuta realmente.

¿Y si hubiéramos usado `QuoteNode` lugar? Como puede imaginar, dado que `QuoteNode` evita que la interpolación ocurra, esto significa que no funcionará.

```
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
```

En este ejemplo, podríamos estar de acuerdo en que `Meta.quot` ofrece una mayor flexibilidad, ya que permite la interpolación. Entonces, ¿por qué podríamos considerar usar `QuoteNode` ? En algunos casos, es posible que no deseemos realmente la interpolación, y que realmente queramos la expresión `$` literal. ¿Cuándo sería deseable? Consideremos una generalización de `@makeex` donde podemos pasar argumentos adicionales que determinan lo que viene a la izquierda y derecha del signo `+` :

```
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 limitación de nuestra implementación de `@makeex4` es que no podemos usar expresiones como los lados izquierdo y derecho de la expresión directamente, ya que se interpolan. En otras palabras, las expresiones pueden evaluarse para interpolación, pero es posible que deseamos que se conserven. (Ya que hay muchos niveles de cotización y evaluación aquí, aclaremos: nuestra macro genera *código* que construye una *expresión* que cuando se evalúa produce otra *expresión* . ¡Uf!)

```
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)
```

Deberíamos permitir que el usuario especifique cuándo debe ocurrir la interpolación y cuándo no. Teóricamente, es una solución fácil: solo podemos eliminar uno de los `$` signos en nuestra aplicación y dejar que el usuario contribuya con los suyos. Lo que esto significa es que interpolamos una versión citada de la expresión ingresada por el usuario (que ya hemos citado e interpolado una vez). Esto lleva al siguiente código, que puede ser un poco confuso al principio, debido a los múltiples niveles anidados de cotización y no cotización. Intenta leer y entender para qué sirve cada escape.

```
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
```

Las cosas empezaron bien, pero algo salió mal. El código generado de la macro está intentando interpolar la copia de `y` en el ámbito de invocación de la macro; pero no *hay* copia de `y` en el ámbito de invocación de macros. Nuestro error es permitir la interpolación con los argumentos segundo y tercero en la macro. Para corregir este error, debemos utilizar `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)
```

Al utilizar `QuoteNode` , hemos protegido nuestros argumentos de la interpolación. Como `QuoteNode` solo tiene el efecto de protecciones adicionales, nunca es dañino usar `QuoteNode` , a menos que desee interpolación. Sin embargo, entender la diferencia hace posible entender dónde y por qué `Meta.quot` podría ser una mejor opción.

Este ejercicio largo es con un ejemplo que es claramente demasiado complejo para aparecer en cualquier aplicación razonable. Por lo tanto, hemos justificado la siguiente regla de oro, mencionada anteriormente:

- Si necesita o desea admitir la interpolación, use `Meta.quot` ;
- Si no puede o no quiere permitir la interpolación, use `QuoteNode` .

¿Qué hay de Expr (: cita)?

`Expr(:quote, x)` es equivalente a `Meta.quot(x)` . Sin embargo, este último es más idiomático y es el preferido. Para el código que utiliza mucho la metaprogramación, a menudo se usa una línea que `using Base.Meta` , lo que permite hacer referencia a `Meta.quot` como simplemente `quot` .

Guía

Metaprogramación bits y bobs de π

Metas:

- Enseñar a través de ejemplos funcionales / útiles / no abstractos dirigidos mínimos (por ejemplo, `@swap` o `@assert`) que introducen conceptos en contextos adecuados
- Prefiero dejar que el código ilustre / demuestre los conceptos en lugar de los párrafos explicativos.
- Evite vincular la 'lectura requerida' a otras páginas, ya que interrumpe la narrativa
- Presente las cosas en un orden sensible que facilitará el aprendizaje.

Recursos:

julialang.org
wikibook (@Cormullion)
5 capas (Leah Hanson)
Cotización SO-Doc (@TotalVerb)
SO-Doc - Símbolos que no son identificadores legales (@TotalVerb)
SO: ¿Qué es un símbolo en Julia (@StefanKarpinski)
Hilo del discurso (@ pi-) Metaprogramación

La mayor parte del material provino del canal del discurso, la mayor parte de eso provino de fcard ... por favor, pídeme si me he olvidado de las atribuciones.

Símbolo

```
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
```

Pasando banderas en funciones:

```
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 ejemplo de hashkey:

```
number_names = Dict{Symbol, Int}()
number_names[:one] = 1
number_names[:two] = 2
number_names[:six] = 6
```

(Avanzado) (@fcard) `:foo` aka `:(foo)` produce un símbolo si `foo` es un identificador válido, de lo contrario es una expresión.

```
# 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
...
```

Básicamente, puedes tratar a los Símbolos como cadenas ligeras. Eso no es para lo que son, pero puedes hacerlo, entonces ¿por qué no? La propia Base de Julia lo hace, `print_with_color(:red, "abc")` imprime un abc de color rojo.

Expr (AST)

(Casi) todo en Julia es una expresión, es decir, una instancia de `Expr` , que contendrá 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
```

Anidado `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` multiline usando `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
```

... por lo que `quote` es funcionalmente lo mismo pero proporciona información de depuración adicional.

(*) **CONSEJO** : Use `let` para mantener `x` dentro del bloque

`quote` una `quote`

`Expr(:quote, x)` se utiliza para representar comillas dentro de comillas.

```
Expr(:quote, :(x + y)) == :(:(x + y))

Expr(:quote, Expr(:$, :x)) == :(:($x))
```

`QuoteNode(x)` es similar a `Expr(:quote, x)` pero evita la interpolación.

```
eval(Expr(:quote, Expr(:$, 1))) == 1

eval(QuoteNode(Expr(:$, 1))) == Expr(:$, 1)
```

(Desambigüe los diversos mecanismos de cotización en metaprogramación de Julia

¿Son $ y : (...) de alguna manera inversos entre sí?

`:(foo)` significa "no mire el valor, mire la expresión" `$foo` significa "cambie la expresión a su valor"

`:($(foo)) == foo` . `$(:(foo))` es un error. `$(...)` no es una operación y no hace nada por sí misma, es un "interpolar esto". señal de que utiliza la sintaxis de cotización. Es decir, solo existe dentro de una cotización.

¿Es `$ foo` lo mismo que `eval( foo )` ?

¡No! `$foo` se intercambia por el valor en tiempo de compilación `eval(foo)` significa hacerlo en tiempo de ejecución

`eval` se producirá en el ámbito global, la interpolación es local.

`eval(:<expr>)` debe devolver lo mismo que solo `<expr>` (asumiendo que `<expr>` es una expresión válida en el espacio global actual)

```
eval(:(1 + 2)) == 1 + 2

eval(:(let x=1; x + 1 end)) == let x=1; x + 1 end
```

`macro` S

Listo? :)

```
# let's try to make this!
julia> x = 5; @show x;
x = 5
```

Vamos a hacer nuestra propia 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` para bajar un `Expr`

5 capas (Leah Hanson) <- explica cómo Julia toma el código fuente como una cadena, lo tokeniza en un `Expr` -tree (AST), expande todas las macros (aún AST), ***baja*** (AST rebajado), luego se convierte en LLVM (y más allá, en este momento no necesitamos preocuparnos por lo que hay más allá)

Q: `code_lowered` actúa sobre las funciones. ¿Es posible bajar un `Expr` ? A: siup!

```
# 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)
```

Si solo quieres expandir macros puedes usar **`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
```

... que devuelve un AST no bajado pero con todas las macros expandidas.

`esc()`

`esc(x)` devuelve un Expr que dice "no aplique higiene a esto", es lo mismo que `Expr(:escape, x)` . La higiene es lo que mantiene una macro autocontenida, y tú `esc` cosas si quieres que se "escapen". p.ej

Ejemplo: `swap` macro para ilustrar `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
```

`$` nos permite 'escapar' de la `quote` . Entonces, ¿por qué no simplemente `$p` y `$q` ? es decir

```
    # FAIL!
```

```
    tmp = $p
    $p = $q
    $q = tmp
```

Como eso buscaría primero el alcance de la `macro` para `p` , y encontraría un `p` local, es decir, el parámetro `p` (sí, si posteriormente accede a `p` sin `esc` -ing, la macro considera el parámetro `p` como una variable local).

Entonces `$p = ...` es solo una asignación a la `p` local. no afecta a ninguna variable que se haya pasado en el contexto de llamada.

Ok, entonces, ¿qué tal

```
    # Almost!
    tmp = $p          # <-- you might think we don't
    $(esc(p)) = $q    #       need to esc() the RHS
    $(esc(q)) = tmp
```

Así que `esc(p)` está 'filtrando' `p` en el contexto de llamada. *"Lo que se pasó a la macro que* ***recibimos como `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
```

Como se puede ver `tmp` recibe el tratamiento de la higiene `#10#tmp` , mientras que `x` e `y` no lo hacen. Julia está creando un identificador único para `tmp` , algo que puedes hacer manualmente con `gensym` , es decir:

```
julia> gensym(:tmp)
Symbol("##tmp#270")
```

Pero: **hay un gotcha:**

```
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
```

Otra cosa que hace la higiene macro de julia es que, si la macro es de otro módulo, crea cualquier variable (que no fue asignada dentro de la expresión de retorno de la macro, como `tmp` en este caso) globales del módulo actual, por lo que `$p` convierte en `Swap.$p` , igualmente `$q` -> `Swap.$q` .

En general, si necesita una variable que está fuera del alcance de la macro, debe esclatarla, por lo que debe `esc(p)` y `esc(q)` independientemente de si están en el LHS o RHS de una expresión, o incluso por sí mismas.

la gente ya ha mencionado `gensym` varias veces y pronto te dejará seducir por el lado oscuro de la omisión para escapar de toda la expresión con unos cuantos `gensym` s aquí y allá, pero ... Asegúrate de entender cómo funciona la higiene antes de intentar ser más inteligente que eso! No es un algoritmo particularmente complejo, por lo que no debería tomar mucho tiempo, ¡pero no lo apresures! No uses ese poder hasta que entiendas todas las ramificaciones de él ... (@fcard)

Ejemplo: `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) `|>` es controvertido, sin embargo. Me sorprende que una mafia no haya venido a discutir todavía. (tal vez todo el mundo está cansado de eso). Hay una recomendación de tener la

mayoría, si no toda la macro, solo una llamada a una función, así que:

```
macro until(condition, block)
    esc(until(condition, block))
end

function until(condition, block)
    quote
        while !$condition
            $block
        end
    end
end
```

... es una alternativa más segura.

@ el simple desafío macro de fcard

Tarea: intercambiar los operandos, entonces los `swaps(1/2)` dan `2.00` es decir, `2/1`

```
macro swaps(e)
    e.args[2:3] = e.args[3:-1:2]
    e
end
@swaps(1/2)
2.00
```

Más desafíos macro de @fcard aquí

Interpolación y `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: ¿Por qué los últimos `$` ? R: Interpola, es decir, obliga a Julia a `eval` esa `string(ex)` medida que la ejecución pasa a través de la invocación de esta macro. es decir, si solo ejecuta ese código no forzará ninguna evaluación. Pero en el momento en que `assert(foo)` Julia ***invocará*** esta macro reemplazando su 'AST token / Expr' con lo que sea que devuelva, y el `$` *entrará* en acción.

Un truco divertido para usar {} para bloques.

(@fcard) No creo que haya nada técnico que evite el uso de `{}` como bloques, de hecho, se puede incluso aplicar un punteo en la sintaxis residual `{}` para que funcione:

```
julia> macro c(block)
         @assert block.head == :cell1d
         esc(quote
           $(block.args...)
```

```
         end)
       end
@c (macro with 1 method)

julia> @c {
         print(1)
         print(2)
         1+2
       }
123
```

* (no es probable que siga funcionando si / cuando la sintaxis {} está reutilizada)

Entonces, primero Julia ve el token de macro, así que leerá / analizará tokens hasta el `end` correspondiente, ¿y creará qué? Un `Expr` con `.head=:macro` o algo? ¿Almacena `"a+1"` como una cadena o lo divide en `:+(:a, 1)` ? ¿Cómo ver?

?

(@fcard) En este caso, debido al alcance léxico, a no está definido en el alcance de `@M` por lo que utiliza la variable global ... En realidad, olvidé escapar de la expresión de flipplin en mi ejemplo tonto, pero el *"solo funciona dentro del mismo módulo "* parte de él todavía se aplica.

```
julia> module M
       macro m()
         :(a+1)
       end
       end
M

julia> a = 1
1

julia> M.@m
ERROR: UndefVarError: a not defined
```

La razón es que, si la macro se utiliza en cualquier módulo distinto al que se definió en, todas las variables no definidas dentro del código a expandir se tratan como variables globales del módulo de la macro.

```
julia> macroexpand(:(M.@m))
:(M.a + 1)
```

AVANZADO

@ 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
```

Macro de 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
```

```
"""
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
```

basura / sin procesar ...

ver / volcar una macro

(@ pi-) Supongamos que acabo de hacer `macro m(); a+1; end` en un nuevo REPL. Sin `a` definido. ¿Cómo puedo 'verlo'? como, ¿hay alguna manera de "volcar" una macro? Sin ejecutarlo realmente

(@fcard) Todo el código en macros se pone realmente en funciones, por lo que solo puede ver su código rebajado o de tipo inferido.

```
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!
```

Otras formas de obtener la función de 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)
```

^ Parece que puedo usar `code_typed` en `code_typed` lugar

¿Cómo entender `eval(Symbol("@M"))` ?

(@fcard) Actualmente, cada macro tiene una función asociada. Si tiene una macro llamada `M` , la función de la macro se llama `@M` . En general, puede obtener el valor de una función con, por ejemplo, `eval(:print)` pero con una función de macro, necesita hacer el `Symbol("@M")` , ya que solo `:@M` convierte en `Expr(:macrocall, Symbol("@M"))` y la evaluación que provoca una macroexpansión.

¿Por qué `code_typed` no muestra params?

(@Pi-)

```
julia> code_typed( x -> x^2 )[1]
LambdaInfo for (::##5#6)(::Any)
:(begin
        return x ^ 2
    end)
```

^ Aquí veo uno `::Any` parámetro, pero no parece estar conectado con el token `x` .

```
 julia> code_typed( print )[1]
LambdaInfo for print(::IO, ::Char)
:(begin
        (Base.write)(io,c)
        return Base.nothing
    end::Void)
```

^ Similarmente aquí; no hay nada para conectar `io` con el `::IO` Entonces, ¿no puede ser un volcado completo de la representación AST de ese método de `print` particular ...?

(@fcard) `print(::IO, ::Char)` solo le dice qué método es, no es parte del AST. Ya ni siquiera está presente en el maestro:

```
julia> code_typed(print)[1]
CodeInfo(:(begin
        (Base.write)(io,c)
        return Base.nothing
    end))=>Void
```

(@ pi-) No entiendo lo que quieres decir con eso. Parece estar tirando el AST para el cuerpo de ese método, ¿no? Pensé que `code_typed` le da el AST para una función. Pero parece que falta el primer paso, es decir, la configuración de tokens para params.

(@fcard) `code_typed` está diseñado para mostrar solo el AST del cuerpo, pero por el momento proporciona el AST completo del método, en forma de `LambdaInfo` (0.5) o `CodeInfo` (0.6), pero se omite mucha información cuando se imprime a la respuesta. Deberá inspeccionar el campo `LambdaInfo` por campo para obtener todos los detalles. `dump` va a inundar su respuesta, por lo que podría intentar:

```
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
  display(method)
```

```
end

print_info_fields(x::Pair) = print_info_fields(x[1])
```

Lo que da todos los valores de los campos nombrados de AST de un método:

```
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)
```

¿Ver lil ' `def = print(io::IO, c::Char)` ? ¡Ahí tienes! (también los `slotnames = [..., :io, :c]` part) También sí, la diferencia en la salida se debe a que estaba mostrando los resultados en el maestro.

???

(@ Ismael-VC) te refieres a esto? Despacho genérico con símbolos.

Puedes hacerlo de esta manera:

```
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(::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)
```

Esto es para el algoritmo de Euler! Me pregunto qué piensa @fcard sobre el envío de símbolos genéricos. --- ^: ángel:

Módulo Gotcha

```
@def m begin
  a+2
end

@m # replaces the macro at compile-time with the expression a+2
```

Más exactamente, solo funciona dentro del nivel superior del módulo en el que se definió 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 que esto suceda, pero la opción predeterminada de usarlo siempre va en contra del diseño del idioma. Una buena defensa para esto es evitar que uno use e introduzca nombres dentro de las macros, lo que los hace difíciles de rastrear a un lector humano.

Python `dict` / JSON como sintaxis para` Dict` literales.

Introducción

Julia usa la siguiente sintaxis para los diccionarios:

```
Dict({k₁ => v₁, k₂ => v₂, …, k□₋₁ => v□₋₁, k□ => v□)
```

Mientras Python y JSON se ven así:

```
{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}
```

Para **fines ilustrativos** también podríamos usar esta sintaxis en Julia y agregarle una nueva semántica (la sintaxis de `Dict` es la forma idiomática en Julia, que se recomienda).

Primero veamos qué *tipo* de expresión es:

```
julia> parse("{1:2 , 3: 4}") |> Meta.show_sexpr
(:cell1d, (:(:), 1, 2), (:(:), 3, 4))
```

Esto significa que debemos tomar esta expresión `:cell1d` y transformarla o devolver una nueva expresión que debería tener este aspecto:

```
julia> parse("Dict(1 => 2 , 3 => 4)") |> Meta.show_sexpr
(:call, :Dict, (:(=>), 1, 2), (:(=>), 3, 4))
```

Definición de macro

La siguiente macro, aunque simple, permite demostrar dicha generación y transformación de código:

```
macro dict(expr)
    # Check the expression has the correct form:
    if expr.head ≠ :cell1d || any(sub_expr.head ≠ :(:) for sub_expr ∈ expr.args)
        error("syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`")
    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, …, k□₋₁ => v□₋₁, k□ => v□))

    # Block is escaped so it can reach variables from it's calling scope:
    return esc(block)
end
```

Echemos un vistazo a la macro expansión resultante:

```
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
```

El último ejemplo es exactamente equivalente a:

```
Dict(
    "string" => :b,
    'c'      => 1,
    :symbol  => π,
    Function => print,
    (1:10)   => range(1, 10)
)
```

Mal uso

```
julia> @dict {"one": 1, "two": 2, "three": 3, "four": 4, "five" => 5}
syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`

julia> @dict ["one": 1, "two": 2, "three": 3, "four": 4, "five" => 5]
syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`
```

Tenga en cuenta que Julia tiene otros usos para los dos puntos `:` como tal, tendrá que envolver las expresiones literales de rango entre paréntesis o usar la función de `range` , por ejemplo.

Lea Metaprogramacion en línea: https://riptutorial.com/es/julia-lang/topic/1945/metaprogramacion

Capítulo 26: mientras bucles

Sintaxis

- mientras cond cuerpo; fin
- descanso
- continuar

Observaciones

El `while` de bucle no tiene un valor; aunque se puede usar en la posición de expresión, su tipo es `Void` y el valor obtenido no será `nothing` .

Examples

Secuencia de collatz

El `while` bucle se ejecuta su cuerpo, siempre y cuando la condición se cumple. Por ejemplo, el siguiente código calcula e imprime la secuencia de Collatz desde un número dado:

```
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
```

Es posible escribir cualquier bucle de forma recursiva, y para complejos `while` bucles, a veces la variante recursiva es más clara. Sin embargo, en Julia, los bucles tienen algunas ventajas distintas sobre la recursión:

- Julia no garantiza la eliminación de la llamada de cola, por lo que la recursión usa memoria adicional y puede causar errores de desbordamiento de pila.
- Y además, por la misma razón, un bucle puede tener una sobrecarga reducida y correr más rápido.

Ejecutar una vez antes de probar la condición

A veces, uno quiere ejecutar algún código de inicialización una vez antes de probar una condición. En algunos otros idiomas, este tipo de bucle tiene una sintaxis especial de `do` - `while` . Sin embargo, esta sintaxis se puede sustituir por un habitual `while` bucle y `break` declaración, por lo que Julia no se han especializado `do` - `while` la sintaxis. En su lugar, uno escribe:

```
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
```

Tenga en cuenta que en algunas situaciones, tales bucles podrían ser más claros con la recursión:

```
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
```

Búsqueda de amplitud

0.5.0

(Aunque este ejemplo está escrito utilizando la sintaxis introducida en la versión v0.5, también puede funcionar con algunas modificaciones en versiones anteriores).

Esta aplicación de búsqueda en anchura (BFS) en un gráfico representado con listas de adyacencia utiliza `while` bucles y el `return` comunicado. La tarea que resolveremos es la siguiente: tenemos una secuencia de personas y una secuencia de amistades (las amistades son mutuas). Queremos determinar el grado de conexión entre dos personas. Es decir, si dos personas son amigas, devolveremos `1` ; si uno es amigo de un amigo del otro, regresaremos `2` , y así sucesivamente.

Primero, supongamos que ya tenemos una lista de adyacencia: un `Dict` mapeo `T` a `Array{T, 1}` , donde las claves son personas y los valores son todos los amigos de esa persona. Aquí podemos representar a personas con cualquier tipo de `T` que escojamos; En este ejemplo, usaremos `Symbol` . En el algoritmo BFS, mantenemos una cola de personas para "visitar", y marcamos su distancia desde el nodo de origen.

```
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
```

Ahora, escribiremos una función para construir una lista de adyacencia dada una secuencia de personas y una secuencia de tuplas `(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
```

Ahora podemos definir la función original:

```
degree(people, friendships, source, dest) =
    degree(makeadjlist(people, friendships), source, dest)
```

Ahora probemos nuestra función en algunos datos.

```
const people = [:jean, :javert, :cosette, :gavroche, :éponine, :marius]
const friendships = [
    (:jean, :cosette),
```

```
    (:jean, :marius),
    (:cosette, :éponine),
    (:cosette, :marius),
    (:gavroche, :éponine)
]
```

Jean está conectado a sí mismo en `0` pasos:

```
julia> degree(people, friendships, :jean, :jean)
0
```

Jean y Cosette son amigos, y por eso tienen el grado `1` :

```
julia> degree(people, friendships, :jean, :cosette)
1
```

Jean y Gavroche están conectados indirectamente a través de Cosette y luego a Marius, por lo que su grado es `3` :

```
julia> degree(people, friendships, :jean, :gavroche)
3
```

Javert y Marius no están conectados a través de ninguna cadena, por lo que se produce un error:

```
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
```

Lea mientras bucles en línea: https://riptutorial.com/es/julia-lang/topic/5565/mientras-bucles

Capítulo 27: Módulos

Sintaxis

- módulo Módulo; ...; fin
- usando el módulo
- Módulo de importación

Examples

Envolver código en un módulo

La palabra clave del `module` se puede utilizar para comenzar un módulo, lo que permite organizar el código y el espacio de nombre. Los módulos pueden definir una interfaz externa, que generalmente consiste en símbolos `export` . Para admitir esta interfaz externa, los módulos pueden tener funciones y tipos internos no exportados que no están destinados para uso público.

Algunos módulos existen principalmente para envolver un tipo y funciones asociadas. Dichos módulos, por convención, generalmente se nombran con la forma plural del nombre del tipo. Por ejemplo, si tenemos un módulo que proporciona un tipo de `Building` , podemos llamar a dicho módulo `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
```

El módulo se puede usar con la instrucción `using` :

```
julia> using Buildings

julia> Building("Burj Khalifa", 163, 830)
163-story Burj Khalifa with height 830m

julia> height(ans)
```

```
830
```

Uso de módulos para organizar paquetes

Normalmente, los paquetes constan de uno o más módulos. A medida que los paquetes crecen, puede ser útil organizar el módulo principal del paquete en módulos más pequeños. Un idioma común es definir esos módulos como submódulos del módulo principal:

```
module RootModule

module SubModule1

...

end

module SubModule2

...

end

end
```

Inicialmente, ni el módulo raíz ni los submódulos tienen acceso a los símbolos exportados de cada uno. Sin embargo, las importaciones relativas son compatibles para abordar este 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
```

En este ejemplo, el valor de `RootModule.z` es `30` .

Lea Módulos en línea: https://riptutorial.com/es/julia-lang/topic/7368/modulos

Capítulo 28: Normalización de cuerdas

Sintaxis

- normalize_string (s :: String, ...)

Parámetros

Parámetro	Detalles
`casefold=true`	Dobla la cadena a un caso canónico basado en el estándar Unicode .
`stripmark=true`	Pele las marcas diacríticas (es decir, los acentos) de los caracteres en la cadena de entrada.

Examples

Comparación de cadenas insensibles a mayúsculas

Las cuerdas pueden compararse con el operador `==` en Julia, pero esto es sensible a las diferencias en el caso. Por ejemplo, `"Hello"` y `"hello"` se consideran cadenas diferentes.

```
julia> "Hello" == "Hello"
true

julia> "Hello" == "hello"
false
```

Para comparar las cadenas de una manera que no distingue entre mayúsculas y minúsculas, normalice las cuerdas plegándolas primero. Por ejemplo,

```
equals_ignore_case(s, t) =
    normalize_string(s, casefold=true) == normalize_string(t, casefold=true)
```

Este enfoque también maneja unicode no ASCII correctamente:

```
julia> equals_ignore_case("Hello", "hello")
true

julia> equals_ignore_case("Weierstraß", "WEIERSTRASS")
true
```

Tenga en cuenta que en alemán, la forma en mayúscula del carácter ß es SS.

Comparación de cuerdas diacrítico-insensibles

A veces, uno quiere cadenas como `"resume"` y `"ré sumé "` para comparar iguales. Es decir, los grafemas que comparten un glifo básico, pero posiblemente difieren debido a las adiciones a esos glifos básicos. Dicha comparación se puede realizar mediante la eliminación de marcas diacríticas.

```
equals_ignore_mark(s, t) =
    normalize_string(s, stripmark=true) == normalize_string(t, stripmark=true)
```

Esto permite que el ejemplo anterior funcione correctamente. Además, funciona bien incluso con caracteres Unicode que no son ASCII.

```
julia> equals_ignore_mark("resume", "ré sumé ")
true

julia> equals_ignore_mark("αβγ", "à β ỹ ")
true
```

Lea Normalización de cuerdas en línea: https://riptutorial.com/es/julia-lang/topic/7612/normalizacion-de-cuerdas

Capítulo 29: Paquetes

Sintaxis

- Pkg.add (paquete)
- Pkg.checkout (paquete, rama = "maestro")
- Pkg.clone (url)
- Pkg.dir (paquete)
- Pkg.pin (paquete, versión)
- Pkg.rm (paquete)

Parámetros

Parámetro	Detalles
`Pkg.add(` *`package`* `)`	Descargue e instale el paquete registrado dado.
`Pkg.checkout(` *`package`* `,` *`branch`* `)`	Echa un vistazo a la rama dada para el paquete registrado dado. *`branch`* es opcional y por defecto es `"master"` .
`Pkg.clone(` *`url`* `)`	Clone el repositorio de Git en la URL dada como un paquete.
`Pkg.dir(` *`package`* `)`	Obtener la ubicación en el disco para el paquete dado.
`Pkg.pin(` *`package`* `,` *`version`* `)`	Forzar el paquete para permanecer en la versión dada. *`version`* es opcional y por defecto es la versión actual del paquete.
`Pkg.rm(` *`package`* `)`	Eliminar el paquete dado de la lista de paquetes requeridos.

Examples

Instalar, usar y eliminar un paquete registrado

Después de encontrar un paquete oficial de Julia, es sencillo descargar e instalar el paquete. En primer lugar, se recomienda actualizar la copia local de METADATA:

```
julia> Pkg.update()
```

Esto asegurará que obtenga las últimas versiones de todos los paquetes.

Supongamos que el paquete que queremos instalar se llama `Currencies.jl` . El comando para ejecutar para instalar este paquete sería:

```
julia> Pkg.add("Currencies")
```

Este comando instalará no solo el paquete en sí, sino también todas sus dependencias.

Si la instalación se realiza correctamente, puede probar que el paquete funciona correctamente :

```
julia> Pkg.test("Currencies")
```

Luego, para usar el paquete, use

```
julia> using Currencies
```

y proceda según lo descrito en la documentación del paquete, generalmente vinculado o incluido desde su archivo README.md.

Para desinstalar un paquete que ya no es necesario, use la función `Pkg.rm` :

```
julia> Pkg.rm("Currencies")
```

Tenga en cuenta que esto no puede eliminar realmente el directorio del paquete; en su lugar, simplemente marcará el paquete como ya no es necesario. A menudo, esto está perfectamente bien: ahorrará tiempo en caso de que necesite el paquete nuevamente en el futuro. Pero si es necesario, para eliminar el paquete físicamente, llame a la función `rm` , luego llame a `Pkg.resolve` :

```
julia> rm(Pkg.dir("Currencies"); recursive=true)

julia> Pkg.resolve()
```

Echa un vistazo a una rama diferente o versión

A veces, la última versión etiquetada de un paquete tiene errores o le faltan algunas características necesarias. Es posible que los usuarios avanzados deseen actualizar a la última versión de desarrollo de un paquete (a veces denominado "maestro", que lleva el nombre habitual de una rama de desarrollo en Git). Los beneficios de esto incluyen:

- Los desarrolladores que contribuyen a un paquete deben contribuir a la última versión de desarrollo.
- La última versión de desarrollo puede tener características útiles, correcciones de errores o mejoras de rendimiento.
- Los usuarios que informen sobre un error pueden desear verificar si ocurre un error en la última versión de desarrollo.

Sin embargo, existen muchos inconvenientes para ejecutar la última versión de desarrollo:

- La última versión de desarrollo puede estar mal probada y tener errores graves.
- La última versión de desarrollo puede cambiar con frecuencia, rompiendo su código.

Para consultar la última rama de desarrollo de un paquete llamado `JSON.jl` , por ejemplo, use

```
Pkg.checkout("JSON")
```

Para revisar una rama o etiqueta diferente (sin nombre "maestro"), use

```
Pkg.checkout("JSON", "v0.6.0")
```

Sin embargo, si la etiqueta representa una versión, generalmente es mejor usar

```
Pkg.pin("JSON", v"0.6.0")
```

Tenga en cuenta que aquí se usa una versión literal, no una cadena simple. La versión `Pkg.pin` informa al administrador de paquetes de la restricción de la versión, lo que le permite ofrecer comentarios sobre los problemas que podría causar.

Para volver a la última versión etiquetada,

```
Pkg.free("JSON")
```

Instalar un paquete no registrado

Algunos paquetes experimentales no están incluidos en el repositorio de paquetes METADATA. Estos paquetes se pueden instalar clonando directamente sus repositorios Git. Tenga en cuenta que puede haber dependencias de paquetes no registrados que a su vez no están registrados; esas dependencias no pueden ser resueltas por el administrador de paquetes y se deben resolver manualmente. Por ejemplo, para instalar el paquete no registrado `OhMyREPL.jl` :

```
Pkg.clone("https://github.com/KristofferC/Tokenize.jl")
Pkg.clone("https://github.com/KristofferC/OhMyREPL.jl")
```

Luego, como es habitual, use `using` para usar el paquete:

```
using OhMyREPL
```

Lea Paquetes en línea: https://riptutorial.com/es/julia-lang/topic/5815/paquetes

Capítulo 30: para bucles

Sintaxis

- para i en iter ...; fin
- mientras cond ...; fin
- descanso
- continuar
- @parallel (op) para i en iter; ...; fin
- @parallel para i en iter; ...; fin
- etiqueta @goto
- etiqueta @label

Observaciones

Cuando haga que el código sea más corto y fácil de leer, considere usar funciones de orden superior, como `map` o `filter` , en lugar de bucles.

Examples

Fizz Buzz

Un caso de uso común para un bucle `for` es iterar sobre un rango o colección predefinidos, y hacer la misma tarea para todos sus elementos. Por ejemplo, aquí combinamos un bucle `for` con una sentencia condicional `if` - `elseif` - `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
```

Esta es la clásica entrevista de Fizz Buzz . La salida (truncada) es:

```
1
2
Fizz
4
Buzz
Fizz
7
8
```

Encuentra el factor primo más pequeño

En algunas situaciones, es posible que desee regresar de una función antes de terminar un ciclo completo. La declaración de `return` se puede utilizar para esto.

```
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
```

Los bucles también pueden terminarse antes con la instrucción de `break` , que termina solo el bucle envolvente en lugar de toda la función.

Iteración multidimensional

En Julia, un bucle for puede contener una coma (`,`) para especificar la iteración sobre múltiples dimensiones. Esto actúa de manera similar a anidar un bucle dentro de otro, pero puede ser más compacto. Por ejemplo, la siguiente función genera elementos del producto cartesiano de dos iterables:

```
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)
```

Sin embargo, la indexación sobre matrices de cualquier dimensión se debe hacer con cada

`eachindex` , no con un bucle multidimensional (si es posible):

```
s = zero(eltype(A))
for ind in eachindex(A)
    s += A[ind]
end
```

Reducción y bucles paralelos.

Julia proporciona macros para simplificar la distribución de computación en múltiples máquinas o trabajadores. Por ejemplo, lo siguiente calcula la suma de algún número de cuadrados, posiblemente en paralelo.

```
function sumofsquares(A)
    @parallel (+) for i in A
        i ^ 2
    end
end
```

Uso:

```
julia> sumofsquares(1:10)
385
```

Para obtener más información sobre este tema, consulte el ejemplo en `@parallel` dentro del tema `@parallel` paralelo.

Lea para bucles en línea: https://riptutorial.com/es/julia-lang/topic/4355/para-bucles

Capítulo 31: Procesamiento en paralelo

Examples

pmap

`pmap` toma una función (que usted especifica) y la aplica a todos los elementos de una matriz. Este trabajo se divide entre los trabajadores disponibles. Luego, `pmap` devuelve los resultados de esa función a otra matriz.

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

Si la función toma múltiples argumentos, puede suministrar múltiples vectores a `pmap`

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

`@parallel` embargo, al igual que con `@parallel` , si la función dada a `pmap` no está en la base Julia (es decir, está definida por el usuario o definida en un paquete), primero debe asegurarse de que la función esté disponible para todos los trabajadores:

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

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

Vea también este SO Q&A.

@paralela

@parallel se puede usar para paralelizar un bucle, dividiendo los pasos del bucle entre diferentes trabajadores. Como un ejemplo muy simple:

```
addprocs(3)

a = collect(1:10)

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

Para un ejemplo un poco más complejo, considere:

```
@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
```

Por lo tanto, vemos que si hubiéramos ejecutado este bucle sin `@parallel` , habrían tardado 55 segundos, en lugar de 27, en ejecutarse.

También podemos suministrar un operador de reducción para la macro `@parallel` . Supongamos que tenemos una matriz, queremos sumar cada columna de la matriz y luego multiplicar estas sumas entre sí:

```
A = rand(100,100);

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

Hay varias cosas importantes que se deben tener en cuenta al usar `@parallel` para evitar comportamientos inesperados.

Primero: si desea usar cualquier función en sus bucles que no esté en la base Julia (por ejemplo, cualquiera de las funciones que defina en su script o que importe desde paquetes), debe hacer que esas funciones estén disponibles para los trabajadores. Así, por ejemplo, lo siguiente *no* funcionaría:

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

En su lugar, necesitaríamos usar:

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

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

Segundo Aunque cada trabajador podrá acceder a los objetos en el alcance del controlador, *no* podrá modificarlos. Así

```
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 que, si hubiéramos ejecutado el bucle sin el @paralelo, habría modificado con éxito la matriz `a` .

PARA ABORDAR ESTO, podemos hacer `a` objeto de tipo `SharedArray` para que cada trabajador pueda acceder y 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 y @spawnat

Las macros `@spawn` y `@spawnat` son dos de las herramientas que Julia pone a disposición para asignar tareas a los trabajadores. Aquí hay un ejemplo:

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

julia>  From worker 2:  hello world
```

Ambas macros evaluarán una expresión en un proceso de trabajo. La única diferencia entre los dos es que `@spawnat` permite elegir qué trabajador evaluará la expresión (en el ejemplo anterior se especifica el trabajador 2), mientras que con `@spawn` se elegirá automáticamente un trabajador, según la disponibilidad.

En el ejemplo anterior, simplemente el trabajador 2 ejecutó la función println. No había nada de interés para volver o recuperar de esto. Sin embargo, a menudo, la expresión que enviamos al trabajador producirá algo que deseamos recuperar. Observe que en el ejemplo anterior, cuando llamamos a `@spawnat` , antes de que `@spawnat` la impresión del trabajador 2, vimos lo siguiente:

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

Esto indica que la macro `@spawnat` devolverá un objeto de tipo `RemoteRef` . Este objeto, a su vez, contendrá los valores de retorno de nuestra expresión que se envía al trabajador. Si queremos recuperar esos valores, lo primero que puede asignar el `RemoteRef` que `@spawnat` vuelve a un objeto y, a continuación, y luego usar el `fetch()` la función que opera en un `RemoteRef` objeto de tipo, para recuperar los resultados almacenados a partir de una evaluación realizada en un trabajador.

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

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

julia> fetch(result)
7
```

La clave para poder usar `@spawn` efectiva es entender la naturaleza detrás de las expresiones en las que opera. Usar `@spawn` para enviar comandos a los trabajadores es un poco más complicado que solo escribir directamente lo que escribiría si estuviera ejecutando un "intérprete" en uno de los trabajadores o ejecutando el código de forma nativa en ellos. Por ejemplo, supongamos que deseamos usar `@spawnat` para asignar un valor a una variable en un trabajador. Podríamos intentar:

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

¿Funcionó? Bueno, veamos si el trabajador 2 intenta imprimir `a` archivo.

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

julia>
```

No pasó nada. ¿Por qué? Podemos investigar esto más usando `fetch()` como anteriormente. `fetch()` puede ser muy útil porque no solo recuperará resultados exitosos sino también mensajes de error. Sin él, tal vez ni siquiera sepamos que algo ha ido mal.

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

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

El mensaje de error dice que `a` no está definido en el trabajador 2. Pero, ¿por qué es esto? La razón es que necesitamos incluir nuestra operación de asignación en una expresión que luego usamos `@spawn` para decirle al trabajador que evalúe. A continuación se muestra un ejemplo, con la siguiente explicación:

```
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 sintaxis `:()` es lo que Julia usa para designar expresiones . Luego usamos la función `eval()` en Julia, que evalúa una expresión, y usamos la macro `@spawnat` para indicar que la expresión se evalúe en el trabajador 2.

También podríamos lograr el mismo resultado que:

```
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
```

Este ejemplo demuestra dos nociones adicionales. Primero, vemos que también podemos crear una expresión usando la función `parse()` llamada en una cadena. En segundo lugar, vemos que podemos usar paréntesis al llamar a `@spawnat` , en situaciones donde esto podría hacer que nuestra sintaxis sea más clara y manejable.

Cuándo usar @parallel vs. pmap

La documentación de Julia informa que

> pmap () está diseñado para el caso en el que cada llamada de función realiza una gran cantidad de trabajo. En contraste, @parallel for puede manejar situaciones donde cada iteración es pequeña, tal vez simplemente sumando dos números.

Hay varias razones para esto. Primero, `pmap` incurre en mayores costos de inicio, lo que `pmap` empleos en los trabajadores. Por lo tanto, si los trabajos son muy pequeños, estos costos de inicio pueden volverse ineficientes. Sin embargo, a la `pmap` , `pmap` hace un trabajo "más inteligente" al asignar trabajos entre los trabajadores. En particular, crea una cola de trabajos y envía un nuevo trabajo a cada trabajador siempre que ese trabajador esté disponible. `@parallel` en contraste, divide todo el trabajo que se debe hacer entre los trabajadores cuando se llama. Como tal, si algunos trabajadores tardan más en sus trabajos que otros, puede terminar con una situación en la que la mayoría de sus trabajadores han terminado y están inactivos, mientras que unos pocos permanecen activos durante un tiempo excesivo, terminando sus trabajos. Sin embargo, tal situación es menos probable que ocurra con trabajos muy pequeños y simples.

Lo siguiente ilustra esto: supongamos que tenemos dos trabajadores, uno de los cuales es lento y el otro es el doble de rápido. Idealmente, queremos dar al trabajador rápido el doble de trabajo que al trabajador lento. (O, podríamos tener trabajos rápidos y lentos, pero el principal es exactamente el mismo). `pmap` logrará esto, pero `@parallel` no lo hará.

Para cada prueba, inicializamos lo siguiente:

```
addprocs(2)

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

Ahora, para la prueba `@parallel` , ejecutamos lo siguiente:

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

Y vuelve la salida de impresión:

```
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
```

Es casi dulce. Los trabajadores han "compartido" el trabajo de manera equitativa. Tenga en cuenta que cada trabajador ha completado 6 trabajos, aunque el trabajador 2 es dos veces más rápido que el trabajador 3. Puede ser conmovedor, pero ineficiente.

Para la prueba de `pmap` , ejecuto lo siguiente:

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

y obtener la salida:

```
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
```

Ahora, tenga en cuenta que el trabajador 2 ha realizado 8 trabajos y el trabajador 3 ha realizado 4. Esto es exactamente proporcional a su velocidad y lo que queremos para una eficiencia óptima. `pmap` es un maestro de tareas difíciles, de cada uno según su capacidad.

@async y @sync

Según la documentación en `?@async` , " `@async` envuelve una expresión en una tarea". Lo que esto significa es que para cualquier cosa que se encuentre dentro de su alcance, Julia iniciará esta tarea ejecutándose, pero luego procederá a lo que sigue en el script sin esperar a que la tarea se complete. Así, por ejemplo, sin la macro obtendrás:

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

Pero con la macro, obtienes:

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

julia>
```

De este modo, Julia permite que el script continúe (y que la macro `@time` se ejecute por completo) sin esperar a que la tarea (en este caso, durmiendo durante dos segundos) se complete.

La macro `@sync` , por el contrario, "esperará hasta que se completen todos los usos dinámicamente cerrados de `@async` , `@spawn` , `@spawnat` y `@parallel` ". (De acuerdo con la documentación bajo `?@sync`). Así, vemos:

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

En este simple ejemplo, entonces, no tiene sentido incluir una sola instancia de `@async` y `@sync` juntas. Pero, donde `@sync` puede ser útil es donde tiene `@async` aplicado a múltiples operaciones que desea permitir que comiencen todas al mismo tiempo sin esperar a que se complete cada una.

Por ejemplo, supongamos que tenemos varios trabajadores y nos gustaría comenzar cada uno de ellos trabajando en una tarea simultáneamente y luego obtener los resultados de esas tareas. Un intento inicial (pero incorrecto) podría ser:

```
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)
```

El problema aquí es que el bucle espera a que `remotecall_fetch()` cada operación `remotecall_fetch()` , es decir, para que cada proceso complete su trabajo (en este caso durante 2 segundos) antes de continuar para comenzar la próxima operación `remotecall_fetch()` . En términos de una situación práctica, no estamos obteniendo los beneficios del paralelismo aquí, ya que nuestros procesos no están haciendo su trabajo (es decir, durmiendo) simultáneamente.

Sin embargo, podemos corregir esto utilizando una combinación de las macros `@async` y `@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)
```

Ahora, si contamos cada paso del bucle como una operación separada, vemos que hay dos operaciones separadas precedidas por la macro `@async` . La macro permite que se inicie cada uno de estos, y que el código continúe (en este caso hasta el siguiente paso del bucle) antes de que finalice cada uno. Sin embargo, el uso de la macro `@sync` , cuyo alcance abarca todo el bucle, significa que no permitiremos que el script `@async` ese bucle hasta que todas las operaciones precedidas por `@async` hayan finalizado.

Es posible obtener una comprensión aún más clara del funcionamiento de estas macros ajustando aún más el ejemplo anterior para ver cómo cambia bajo ciertas modificaciones. Por ejemplo, supongamos que solo tenemos `@async` sin `@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)
```

Aquí, la macro `@async` nos permite continuar en nuestro bucle incluso antes de que cada operación `remotecall_fetch()` termine de ejecutarse. Pero, para bien o para mal, no tenemos una macro `@sync` para evitar que el código continúe más allá de este bucle hasta que todas las operaciones `remotecall_fetch()` terminen.

Sin embargo, cada operación `remotecall_fetch()` todavía se ejecuta en paralelo, incluso una vez que continuamos. Podemos ver eso porque si esperamos dos segundos, entonces la matriz a, que contiene los resultados, contendrá:

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

(El elemento "nada" es el resultado de una recuperación exitosa de los resultados de la función de suspensión, que no devuelve ningún valor)

También podemos ver que las dos operaciones `remotecall_fetch()` comienzan esencialmente al mismo tiempo porque los comandos de `print` que los preceden también se ejecutan en rápida sucesión (la salida de estos comandos no se muestra aquí). Contraste esto con el siguiente ejemplo donde los comandos de `print` ejecutan a un intervalo de 2 segundos entre sí:

Si colocamos la macro `@async` en todo el bucle (en lugar de solo el paso interno), nuevamente nuestra secuencia de comandos continuará inmediatamente sin esperar a que `remotecall_fetch()` operaciones `remotecall_fetch()` . Ahora, sin embargo, solo permitimos que el script continúe más

allá del bucle en su totalidad. No permitimos que cada paso individual del bucle comience antes de que termine el anterior. Como tal, a diferencia del ejemplo anterior, dos segundos después de que el script continúe después del bucle, la matriz de `results` aún tiene un elemento como `#undef` que indica que la segunda operación `remotecall_fetch()` aún no se ha completado.

```
@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
```

Y, como es `@sync` , si colocamos `@sync` y `@async` uno al lado del otro, conseguimos que cada `remotecall_fetch()` ejecute secuencialmente (en lugar de simultáneamente) pero no continuamos en el código hasta que cada uno haya finalizado. En otras palabras, esto sería esencialmente equivalente a si no tuviéramos ninguna macro en su lugar, al igual que `sleep(2)` comporta de manera idéntica 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
```

Tenga en cuenta también que es posible tener operaciones más complicadas dentro del alcance de la macro `@async` . La documentación proporciona un ejemplo que contiene un bucle completo dentro del alcance de `@async` .

Recuerde que la ayuda para las macros de sincronización indica que "Espere hasta que se completen todos los usos dinámicamente incluidos de `@async` , `@spawn` , `@spawnat` y `@parallel` ". Para los fines de lo que se considera "completo", importa cómo defina las tareas dentro del alcance de las macros `@sync` y `@async` . Considere el siguiente ejemplo, que es una ligera variación en uno de los ejemplos dados anteriormente:

```
@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)
```

El ejemplo anterior tardó aproximadamente 2 segundos en ejecutarse, lo que indica que las dos tareas se ejecutaron en paralelo y que el script espera a que cada una complete la ejecución de sus funciones antes de continuar. Este ejemplo, sin embargo, tiene una evaluación de tiempo mucho menor. El motivo es que, a los fines de `@sync` la operación `remotecall()` ha "finalizado" una vez que ha enviado al trabajador el trabajo a realizar. (Tenga en cuenta que la matriz resultante, a, aquí, solo contiene tipos de objetos `RemoteRef` , que solo indican que algo está sucediendo con un proceso en particular que, en teoría, podría obtenerse en algún momento en el futuro). Por el contrario, la operación `remotecall_fetch()` solo ha "finalizado" cuando recibe el mensaje del trabajador de que su tarea está completa.

Por lo tanto, si está buscando formas de asegurarse de que ciertas operaciones con los trabajadores se hayan completado antes de continuar con su secuencia de comandos (como se explica en esta publicación), es necesario pensar detenidamente qué se considera "completo" y cómo lo hará. medir y luego operacionalizar eso en su script.

Agregando trabajadores

Cuando inicie Julia por primera vez, de manera predeterminada, solo habrá un proceso en ejecución y estará disponible para dar trabajo. Puedes verificar esto usando:

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

Para aprovechar el procesamiento paralelo, primero debe agregar trabajadores adicionales que luego estarán disponibles para realizar el trabajo que les asigne. Puede hacer esto dentro de su script (o desde el intérprete) usando: `addprocs(n)` donde `n` es el número de procesos que desea usar.

Alternativamente, puede agregar procesos cuando inicie Julia desde la línea de comando usando:

```
$ julia -p n
```

donde `n` es cuántos procesos *adicionales* desea agregar. Así, si empezamos con Julia

```
$ julia -p 2
```

Cuando Julia empiece obtendremos:

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

Lea Procesamiento en paralelo en línea: https://riptutorial.com/es/julia-lang/topic/4542/procesamiento-en-paralelo

Capítulo 32: Regexes

Sintaxis

- Regex ("[regex]")
- r "[regex]"
- partido (aguja, pajar)
- matchall (aguja, pajar)
- eachmatch (aguja, pajar)
- ismatch (aguja, pajar)

Parámetros

Parámetro	Detalles
`needle`	El `Regex` a buscar en el `haystack`
`haystack`	El texto en el que buscar la `needle`

Examples

Literales regex

Julia soporta expresiones regulares [1] . La biblioteca PCRE se utiliza como la implementación de expresiones regulares. Las expresiones regulares son como un mini-lenguaje dentro de un idioma. Dado que la mayoría de los idiomas y muchos editores de texto brindan soporte para expresiones regulares, la documentación y los ejemplos de cómo usar expresiones regulares en general están fuera del alcance de este ejemplo.

Es posible construir un `Regex` partir de una cadena usando el constructor:

```
julia> Regex("(cat|dog)s?")
```

Pero por conveniencia y más fácil de escapar, la macro de cadena `@r_str` puede usarse en su lugar:

```
julia> r"(cat|dog)s?"
```

[1] : Técnicamente, Julia admite expresiones regulares, que son distintas y más poderosas de lo que se llaman expresiones regulares en la teoría del lenguaje. Con frecuencia, el término "expresión regular" se usará para referirse a expresiones regulares también.

Encontrar coincidencias

Existen cuatro funciones principales de utilidad para expresiones regulares, todas las cuales toman argumentos en orden de `needle, haystack` . La terminología "aguja" y "pajar" provienen del idioma inglés "encontrar una aguja en un pajar". En el contexto de las expresiones regulares, la expresión regular es la aguja y el texto es el pajar.

La función de `match` se puede utilizar para encontrar la primera coincidencia en una cadena:

```
julia> match(r"(cat|dog)s?", "my cats are dogs")
RegexMatch("cats", 1="cat")
```

La función `matchall` se puede usar para encontrar todas las coincidencias de una expresión regular en una cadena:

```
julia> matchall(r"(cat|dog)s?", "The cat jumped over the dogs.")
2-element Array{SubString{String},1}:
 "cat"
 "dogs"
```

La función `ismatch` devuelve un valor booleano que indica si se encontró una coincidencia dentro de la cadena:

```
julia> ismatch(r"(cat|dog)s?", "My pigs")
false

julia> ismatch(r"(cat|dog)s?", "My cats")
true
```

El `eachmatch` función devuelve un iterador sobre `RegexMatch` objetos, adecuado para uso con `for` bucles :

```
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
```

Grupos de captura

Las subcadenas capturadas por los grupos de captura son accesibles desde los objetos `RegexMatch` usando notación de indexación.

Por ejemplo, las siguientes `(555)-555-5555` números de teléfono de América del Norte escritos en formato `(555)-555-5555` :

```
julia> phone = r"\((\d{3})\)-(\d{3})-(\d{4})"
```

y supongamos que deseamos extraer los números de teléfono de un texto:

```
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 función `matchall` , podemos obtener una serie de subcadenas que coincidan:

```
julia> matchall(phone, text)
2-element Array{SubString{String},1}:
 "(555)-505-1000"
 "(555)-999-9999"
```

Pero supongamos que queremos acceder a los códigos de área (los tres primeros dígitos, entre paréntesis). Entonces podemos usar el iterador `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
```

Tenga en cuenta que usamos `m[1]` porque el código de área es el primer grupo de captura en nuestra expresión regular. Podemos obtener los tres componentes del número de teléfono como una tupla usando una función:

```
julia> splitmatch(m) = m[1], m[2], m[3]
splitmatch (generic function with 1 method)
```

Entonces podemos aplicar dicha función a un `RegexMatch` particular:

```
julia> splitmatch(match(phone, text))
("555","505","1000")
```

O podríamos `map` en cada partido:

```
julia> map(splitmatch, eachmatch(phone, text))
2-element Array{Tuple{SubString{String},SubString{String},SubString{String}},1}:
 ("555","505","1000")
 ("555","999","9999")
```

Lea Regexes en línea: https://riptutorial.com/es/julia-lang/topic/5890/regexes

Capítulo 33: REPL

Sintaxis

- julia>
- ayuda?>
- cáscara>
- \[látex]

Observaciones

Otros paquetes pueden definir sus propios modos REPL además de los modos predeterminados. Por ejemplo, el paquete `Cxx` define el modo `cxx>` shell para un C ++ REPL. Estos modos son generalmente accesibles con sus propias teclas especiales; Vea la documentación del paquete para más detalles.

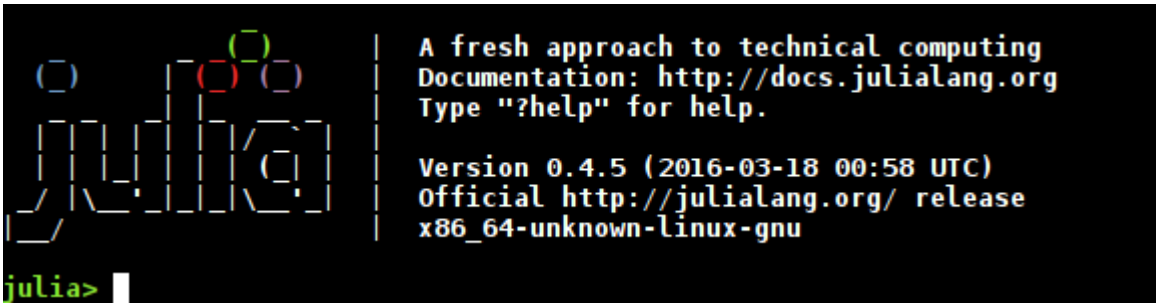
Examples

Lanzar el REPL

Después de instalar Julia , para iniciar el ciclo read-eval-print-loop (REPL):

En Unix Systems

Abra una ventana de terminal, luego escriba `julia` en el indicador y luego presione `Retorno` . Deberías ver algo como esto:



En Windows

Encuentre el programa Julia en su menú de inicio y haga clic en él. El REPL debe ser lanzado.

Usando el REPL como una calculadora

La Julia REPL es una excelente calculadora. Podemos empezar con algunas operaciones simples:

```
julia> 1 + 1
2

julia> 8 * 8
64

julia> 9 ^ 2
81
```

La variable `ans` contiene el resultado del último cálculo:

```
julia> 4 + 9
13

julia> ans + 9
22
```

Podemos definir nuestras propias variables usando la asignación `=` operador:

```
julia> x = 10
10

julia> y = 20
20

julia> x + y
30
```

Julia tiene multiplicación implícita para literales numéricos, lo que hace que algunos cálculos sean más rápidos de escribir:

```
julia> 10x
100

julia> 2(x + y)
60
```

Si cometemos un error y hacemos algo que no está permitido, el Julia REPL lanzará un error, a menudo con un consejo útil sobre cómo solucionar el 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
```

Para acceder o editar comandos anteriores, use la tecla ↑ (Arriba), que se mueve al último elemento del historial. El ↓ mueve al siguiente elemento de la historia. Las teclas ← y → se pueden usar para mover y realizar modificaciones en una línea.

Julia tiene algunas constantes matemáticas integradas, como `e` y `pi` (o `π`).

```
julia> e
e = 2.7182818284590...

julia> pi
π = 3.1415926535897...

julia> 3π
9.42477796076938
```

Podemos escribir caracteres como `π` rápidamente usando sus códigos LaTeX: presione `\` , luego `p` e `i` , luego presione la tecla `Tab` para sustituir el `\pi` acaba de escribir con `π` . Esto funciona para otras letras griegas y símbolos Unicode adicionales.

Podemos usar cualquiera de las funciones matemáticas integradas de Julia, que van desde simples hasta bastante poderosas:

```
julia> cos(π)
-1.0

julia> besselh(1, 1, 1)
0.44005058574493355 - 0.7812128213002889im
```

Los números complejos son compatibles usando `im` como una unidad imaginaria:

```
julia> abs(3 + 4im)
5.0
```

Algunas funciones no devolverán un resultado complejo a menos que le des una entrada compleja, incluso si la entrada es real:

```
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
```

Las operaciones exactas en números racionales son posibles utilizando el `//` operador de división racional:

```
julia> 1//3 + 1//3
2//3
```

Consulte el tema Aritmética para obtener más información sobre los tipos de operadores aritméticos que admite Julia.

Tratar con la precisión de la máquina

Tenga en cuenta que los enteros de la máquina tienen un tamaño limitado y se desbordarán si el resultado es demasiado grande para ser almacenado:

```
julia> 2^62
4611686018427387904

julia> 2^63
-9223372036854775808
```

Esto se puede evitar utilizando enteros de precisión arbitraria en el cálculo:

```
julia> big"2"^62
4611686018427387904

julia> big"2"^63
9223372036854775808
```

Los puntos flotantes de la máquina también están limitados en precisión:

```
julia> 0.1 + 0.2
0.30000000000000004
```

Más precisión (pero aún limitada) es posible usando de nuevo `big` :

```
julia> big"0.1" + big"0.2"
3.000000000000000000000000000000000000000000000000000000000000000000000000000000000000017e-01
```

La aritmética exacta se puede hacer en algunos casos utilizando `Rational` s:

```
julia> 1//10 + 2//10
3//10
```

Usando los modos REPL

Hay tres modos REPL incorporados en Julia: el modo Julia, el modo de ayuda y el modo shell.

El modo de ayuda

El Julia REPL viene con un sistema de ayuda incorporado. Presione `?` en el aviso de `julia>` para acceder al aviso de `help?>` .

En el indicador de ayuda, escriba el nombre de alguna función o escriba para obtener ayuda para:

```
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.
```

Incluso si no deletreas la función correctamente, Julia puede sugerir algunas funciones que posiblemente sean lo que quisiste decir:

```
help?> printline
search:

Couldn't find printline
Perhaps you meant println, pipeline, @inline or print
  No documentation found.

  Binding printline does not exist.
```

Esta documentación también funciona para otros módulos, siempre que utilicen el sistema de documentación 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)
```

El modo shell

Consulte Uso de Shell desde el interior del REPL para obtener más detalles sobre cómo usar el modo de shell de Julia, al que se puede acceder pulsando ; en el aviso Este modo de shell admite la interpolación de datos de la sesión de Julia REPL, lo que facilita llamar a las funciones de Julia y convertir sus resultados en comandos de shell:

```
shell> ls $(Pkg.dir("JSON"))
appveyor.yml  bench  data  LICENSE.md  nohup.out  README.md  REQUIRE  src  test
```

Lea REPL en línea: https://riptutorial.com/es/julia-lang/topic/5739/repl

Capítulo 34: Shell scripting y tuberías

Sintaxis

- comando de shell

Examples

Usando Shell desde el interior del REPL

Desde dentro del interactivo Julia shell (también conocido como REPL), puede acceder al shell del sistema escribiendo ; Justo después del aviso:

```
shell>
```

A partir de aquí, puede escribir cualquier comando de shell y se ejecutarán desde dentro del REPL:

```
shell> ls
Desktop     Documents   Pictures   Templates
Downloads   Music       Public     Videos
```

Para salir de este modo, escriba la tecla de `backspace` cuando la solicitud esté vacía.

Desgastando del código de Julia

El código Julia puede crear, manipular y ejecutar comandos literales, que se ejecutan en el entorno del sistema operativo. Esto es poderoso pero a menudo hace que los programas sean menos portátiles.

Un comando literal se puede crear usando el `` literal. La información se puede interpolar usando la sintaxis de `$` interpolación, como con los literales de cadena. Las variables de Julia pasadas a través de comandos literales no necesitan ser escapadas primero; en realidad no se pasan al shell, sino directamente al kernel. Sin embargo, Julia muestra estos objetos para que aparezcan correctamente escapados.

```
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(+)
```

Lea Shell scripting y tuberías en línea: https://riptutorial.com/es/julia-lang/topic/5420/shell-scripting-y-tuberias

Capítulo 35: sub2ind

Sintaxis

- sub2ind (dims :: Tuple {Vararg {Integer}}, I :: Integer ...)
- sub2ind {T <: Integer} (dims :: Tuple {Vararg {Integer}}, I :: AbstractArray {T <: Integer, 1} ...)

Parámetros

parámetro	detalles
dims :: Tuple {Vararg {Integer}}	tamaño de la matriz
I :: Integer ...	subíndices (escalar) de la matriz
I :: AbstractArray {T <: Entero, 1} ...	subíndices (vector) de la matriz

Observaciones

El segundo ejemplo muestra que el resultado de `sub2ind` puede tener muchos errores en algunos casos específicos.

Examples

Convertir subíndices a índices lineales.

```
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
```

Uno no puede determinar si un subíndice está en el rango de una matriz comparando su índice:

```
julia> sub2ind((3,3), -1, 2)
2

julia> 0 < sub2ind((3,3), -1, 2) <= 9
true
```

Lea sub2ind en línea: https://riptutorial.com/es/julia-lang/topic/1914/sub2ind

Capítulo 36: Tipo de estabilidad

Introducción

La inestabilidad del tipo ocurre cuando el tipo de una variable puede cambiar en tiempo de ejecución, y por lo tanto no puede inferirse en tiempo de compilación. La inestabilidad de tipos a menudo causa problemas de rendimiento, por lo que es importante poder escribir e identificar códigos de tipo estable.

Examples

Escribir código de tipo estable

```
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 sincronización de las dos funciones anteriores muestra diferencias importantes en términos de tiempo y asignaciones de 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)
```

Esto se debe al código de tipo inestable en `sumofsins1` donde el tipo de `r` debe verificar para cada iteración.

Lea Tipo de estabilidad en línea: https://riptutorial.com/es/julia-lang/topic/6084/tipo-de-estabilidad

Capítulo 37: Tuplas

Sintaxis

- una,
- a, b
- a, b = xs
- ()
- (una,)
- (a, b)
- (a, b ...)
- Tupla {T, U, V}
- NTuple {N, T}
- Tupla {T, U, Vararg {V}}

Observaciones

Las tuplas tienen un rendimiento de tiempo de ejecución mucho mejor que las matrices por dos razones: sus tipos son más precisos y su inmutabilidad permite que se asignen en la pila en lugar de la pila. Sin embargo, esta tipificación más precisa viene con más sobrecarga de tiempo de compilación y más dificultad para lograr la estabilidad del tipo .

Examples

Introducción a las tuplas

`Tuple` son colecciones ordenadas inmutables de objetos distintos arbitrarios, ya sea del mismo tipo o de tipos diferentes. Normalmente, las tuplas se construyen utilizando la sintaxis `(x, y)` .

```
julia> tup = (1, 1.0, "Hello, World!")
(1,1.0,"Hello, World!")
```

Los objetos individuales de una tupla se pueden recuperar mediante la sintaxis de indexación:

```
julia> tup[1]
1

julia> tup[2]
1.0

julia> tup[3]
"Hello, World!"
```

Implementan la interfaz iterable y, por lo tanto, se pueden iterar sobre el uso `for` bucles :

```
julia> for item in tup
```

```
            println(item)
        end
1
1.0
Hello, World!
```

Las tuplas también admiten una variedad de funciones de colecciones genéricas, como `reverse` o `length` :

```
julia> reverse(tup)
("Hello, World!",1.0,1)

julia> length(tup)
3
```

Además, las tuplas admiten una variedad de operaciones de recopilación de orden superior , que incluyen `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 vacía se puede construir usando `()` :

```
julia> ()
()

julia> isempty(ans)
true
```

Sin embargo, para construir una tupla de un elemento, se requiere una coma al final. Esto se debe a que los paréntesis (`(` y `)`) se tratarían de otra manera como operaciones de agrupación en lugar de construir una tupla.

```
julia> (1)
1

julia> (1,)
(1,)
```

Por coherencia, también se permite una coma final para tuplas con más de un elemento.

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

Tipos de tuplas

El `typeof` una tupla es un subtipo de la `Tuple` :

```
julia> typeof((1, 2, 3))
Tuple{Int64,Int64,Int64}

julia> typeof((1.0, :x, (1, 2)))
Tuple{Float64,Symbol,Tuple{Int64,Int64}}
```

A diferencia de otros tipos de datos, los tipos de `Tuple` son covariantes . Otros tipos de datos en Julia son generalmente invariantes. Así,

```
julia> Tuple{Int, Int} <: Tuple{Number, Number}
true

julia> Vector{Int} <: Vector{Number}
false
```

Este es el caso porque en todas partes se acepta un `Tuple{Number, Number}` , también lo sería un `Tuple{Int, Int}` , ya que también tiene dos elementos, ambos de los cuales son números. Ese no es el caso de un `Vector{Int}` frente a un `Vector{Number}` , ya que una función que acepta un `Vector{Number}` puede desear almacenar un punto flotante (por ejemplo, `1.0`) o un número complejo (por ejemplo, `1+3im`) en tales un vector.

La covarianza de los tipos de tuplas significa que `Tuple{Number}` (de nuevo, a diferencia de `Vector{Number}`) es en realidad un tipo abstracto:

```
julia> isleaftype(Tuple{Number})
false

julia> isleaftype(Vector{Number})
true
```

Los subtipos concretos de `Tuple{Number}` incluyen `Tuple{Int}` , `Tuple{Float64}` , `Tuple{Rational{BigInt}}` , y así sucesivamente.

`Tuple` tipos de `Vararg` pueden contener un `Vararg` terminación como su último parámetro para indicar un número indefinido de objetos. Por ejemplo, `Tuple{Vararg{Int}}` es el tipo de todas las tuplas que contienen cualquier número de `Int` s, posiblemente cero:

```
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
```

mientras que `Tuple{String, Vararg{Int}}` acepta tuplas que consisten en una cadena , seguida de cualquier número (posiblemente cero) de `Int` s.

```
julia> isa(("x", 1, 2), Tuple{String, Vararg{Int}})
true

julia> isa((1, 2), Tuple{String, Vararg{Int}})
false
```

Combinado con la covarianza, esto significa que `Tuple{Vararg{Any}}` describe cualquier tupla. De hecho, `Tuple{Vararg{Any}}` es solo otra forma de decir `Tuple` :

```
julia> Tuple{Vararg{Any}} == Tuple
true
```

`Vararg` acepta un segundo parámetro de tipo numérico que indica cuántas veces exactamente debe ocurrir su primer parámetro de tipo. (Por defecto, si no se especifica, este segundo parámetro de tipo es un typevar que puede tomar cualquier valor, por lo que cualquier número de `Int` son aceptados s en el `Vararg` . Anteriores s) `Tuple` tipos que terminan en un especificado `Vararg` será automáticamente ampliado para la Número de elementos solicitados:

```
julia> Tuple{String,Vararg{Int, 3}}
Tuple{String,Int64,Int64,Int64}
```

Existe notación para tuplas homogéneas con un `Vararg` especificado: `NTuple{N, T}` . En esta notación, `N` denota el número de elementos en la tupla y `T` denota el tipo aceptado. Por ejemplo,

```
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)
```

Tenga en cuenta que los `NTuple` s más allá de cierto tamaño se muestran simplemente como `NTuple{N, T}` , en lugar de la forma de `Tuple` expandida, pero siguen siendo del mismo tipo:

```
julia> Tuple{Int,Int,Int,Int,Int,Int,Int,Int,Int,Int}
NTuple{10,Int64}
```

Despachando en tipos de tuplas

Debido a que las listas de parámetros de la función Julia son en sí mismas tuplas, el envío de varios tipos de tuplas a menudo es más fácil de realizar a través de los propios parámetros del método, a menudo con un uso liberal para el operador "salpicado" `...` Por ejemplo, considere la implementación de `reverse` para tuplas, desde `Base` :

```
revargs() = ()
revargs(x, r...) = (revargs(r...)..., x)

reverse(t::Tuple) = revargs(t...)
```

La implementación de métodos en las tuplas de esta manera preserva la estabilidad del tipo , lo cual es crucial para el rendimiento. Podemos ver que no hay sobrecarga para este enfoque utilizando 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{Int6

  end::Tuple{Int64,Int64,Int64}
```

Aunque es un poco difícil de leer, el código aquí consiste simplemente en crear una nueva tupla con los valores 3º, 2º y 1º de la tupla original, respectivamente. En muchas máquinas, esto se compila a un código LLVM extremadamente eficiente, que consiste en cargas y almacenes.

```
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
}
```

Múltiples valores de retorno

Las tuplas se utilizan con frecuencia para múltiples valores de retorno. Gran parte de la biblioteca estándar, incluidas dos de las funciones de la interfaz iterable (`next` y `done`), devuelve tuplas que contienen dos valores relacionados pero distintos.

Los paréntesis alrededor de las tuplas se pueden omitir en ciertas situaciones, lo que facilita la implementación de múltiples valores de retorno. Por ejemplo, podemos crear una función para

devolver raíces cuadradas positivas y negativas de un número real:

```
julia> pmsqrt(x::Real) = sqrt(x), -sqrt(x)
pmsqrt (generic function with 1 method)

julia> pmsqrt(4)
(2.0,-2.0)
```

La asignación de destrucción se puede usar para descomprimir los múltiples valores de retorno. Para almacenar las raíces cuadradas en las variables `a` y `b` , basta con escribir:

```
julia> a, b = pmsqrt(9.0)
(3.0,-3.0)

julia> a
3.0

julia> b
-3.0
```

Otro ejemplo de esto son las funciones `divrem` y `fldmod` , que realizan una operación de división y resto de enteros (truncando o entablado, respectivamente) al mismo tiempo:

```
julia> q, r = divrem(10, 3)
(3,1)

julia> q
3

julia> r
1
```

Lea Tuplas en línea: https://riptutorial.com/es/julia-lang/topic/6675/tuplas

Creditos

S. No	Capítulos	Contributors
1	Empezando con Julia Language	Andrew Piliser, becko, Community, Dawny33, Fengyang Wang, Kevin Montrose, prcastro
2	@goto y @label	Fengyang Wang
3	Aritmética	Fengyang Wang
4	Arrays	Fengyang Wang, Michael Ohlrogge, prcastro
5	Cierres	Fengyang Wang
6	Combinadores	Fengyang Wang
7	Comparaciones	Fengyang Wang
8	Compatibilidad de versiones cruzadas	Fengyang Wang
9	Comprensiones	2Cubed, Fengyang Wang, zwlayer
10	Condicionales	Fengyang Wang, Michael Ohlrogge, prcastro
11	Entrada	Fengyang Wang
12	Enums	Fengyang Wang
13	Examen de la unidad	Fengyang Wang
14	Expresiones	Michael Ohlrogge
15	Funciones	Fengyang Wang, Harrison Grodin, Michael Ohlrogge, Sebastialonso
16	Funciones de orden superior	Fengyang Wang, mnoronha
17	Hora	Fengyang Wang
18	Instrumentos de cuerda	Fengyang Wang, Michael Ohlrogge
19	Iterables	Fengyang Wang, prcastro
20	JSON	4444, Fengyang Wang

21	Leyendo un DataFrame desde un archivo	Pranav Bhat
22	Los diccionarios	B Roy Dawson
23	Los tipos	Fengyang Wang, prcastro
24	Macros de cadena	Fengyang Wang
25	Metaprogramacion	Fengyang Wang, Ismael Venegas Castelló, P i, prcastro
26	mientras bucles	Fengyang Wang
27	Módulos	Fengyang Wang
28	Normalización de cuerdas	Fengyang Wang
29	Paquetes	Fengyang Wang
30	para bucles	Fengyang Wang, Michael Ohlrogge
31	Procesamiento en paralelo	Fengyang Wang, Harrison Grodin, Michael Ohlrogge, prcastro
32	Regexes	Fengyang Wang
33	REPL	Fengyang Wang
34	Shell scripting y tuberías	2Cubed, Fengyang Wang, mnoronha, prcastro
35	sub2ind	Fengyang Wang, Gnimuc
36	Tipo de estabilidad	Abhijith, Fengyang Wang
37	Tuplas	Fengyang Wang