Конструкции потока управления в языке Julia

Перцев Дмитрий, Швачко Никита

группа 5030102/20202

Санкт-Петербургский политехнический университет Петра Великого

28 октября 2025 г.

Конструкции потока управления в языке Julia

Julia предоставляет широкий набор конструкций для управления потоком выполнения программы. Эти конструкции позволяют создавать сложную логику, обрабатывать условия, повторять операции и управлять исключениями.

Составные выражения (Compound Expressions)

Составные выражения позволяют объединить несколько подвыражений в одно, возвращая значение последнего выражения.

Блоки begin

Блок `begin` группирует несколько выражений в одно составное выражение:

```
julia> z = begin
    x = 1
    y = 2
    x + y
end
3
```

Объяснение работы: Блок `begin` последовательно вычисляет все выражения внутри (`x = 1`, `y = 2`, `x + y`), возвращая значение последнего. В данном случае результат — сумма `3`.

Цепочки с точкой с запятой

Альтернативный синтаксис использует точку с запятой для разделения выражений:

```
julia> z = (x = 1; y = 2; x + y)
3
```

Смысл: Эквивалент `begin`, но компактнее: выражения разделены `;`, возвращается последнее.

Условное вычисление (Conditional Evaluation)

Конструкция if-elseif-else

Условные выражения позволяют выполнять различные блоки кода в зависимости от логических условий:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

Смысл: Проверяются условия сверху вниз; выполняется код первой истинной ветки, иначе `else`. Блоки `elseif` и `else` необязательны.

if как выражение

В Julia `if` — это выражение, возвращающее значение:

```
julia> result = if x > 0
           "positive"
       elseif x == 0
           "zero"
       else
           "negative"
       end
"positive"
```

Смысл: Можно использовать `if` для присваивания значений, что упрощает код.

Важная особенность: переменные, объявленные внутри `if`, доступны за его пределами, если присваиваются во всех ветках:

```
julia> function test(x,y)
    if x < y
        relation = "less than"
    elseif x == y
        relation = "equal to"
    else
        relation = "greater than"
    end
    println("x is ", relation, " y.")
end
```

Тернарный оператор (? :)

Тернарный оператор — компактная форма условного выбора между двумя значениями:

```
julia> x = 1; y = 2;
julia> println(x < y ? "less than" : "not less than")
less than
```

Смысл: Синтаксис `условие ? значение_если_true : значение_если_false`. Вычисляется и возвращается одна из двух веток; пробелы вокруг `?` и `:` обязательны.

Пример цепочки тернарных операторов:

```
julia> test(x, y) = println(x < y ? "x is less than y" :
                            x > y ? "x is greater than y" : "x is equal to y")
```

Короткое замыкание (Short-Circuit Evaluation)

Операторы `&&` (логическое "И") и `||` (логическое "ИЛИ") в Julia поддерживают короткое замыкание — второй аргумент вычисляется только при необходимости.

Короткое замыкание с побочным эффектом

```
julia> function side_effect(x)
           println("Evaluating ", x)
           return x > 0
       end

julia> side_effect(1) && side_effect(2)
Evaluating 1
Evaluating 2
true

julia> side_effect(0) && side_effect(2)
Evaluating 0
false
```

Объяснение: В `a && b` второй аргумент `b` вычисляется только если `a` истинно. Если `a` ложно, `b` не вычисляется (короткое замыкание).

Оператор && (И)

```
julia> t(x) = (println(x); true)
julia> f(x) = (println(x); false)

julia> t(1) && t(2)
1
2
true

julia> f(1) && t(2)
1
false
```

Объяснение: `b` вычисляется только если `a` истинно. Если `a` ложно, результат `false` независимо от `b`.

Оператор || (ИЛИ)

```
julia> t(x) = (println(x); true)
julia> f(x) = (println(x); false)

julia> t(1) || t(2)
1
true

julia> f(1) || t(2)
1
2
true
```

Объяснение: `b` вычисляется только если `a` ложно. Если `a` истинно, результат `true` сразу.

Практическое применение

Короткое замыкание часто заменяет короткие `if`:

```
julia> function fact(n::Int)
    n >= 0 || error("n must be non-negative")
    n == 0 && return 1
    n * fact(n-1)
end

julia> fact(5)
120

julia> fact(-1)
ERROR: n must be non-negative
```

Смысл: `<условие> && <выражение>` — эквивалент `if <условие> ... end`, `<условие> || <выражение>` — эквивалент `if !<условие> ... end`.

Повторяющееся вычисление: Циклы

Цикл while

Цикл `while` выполняет тело, пока условие истинно:

```
julia> i = 1;
julia> while i <= 3
    println(i)
    global i += 1
end
1
2
3
```

Объяснение: Перед каждой итерацией проверяется условие; `global` нужен для изменения глобальной переменной `i`.

Пример генерации чисел Фибоначчи:

```
length = 15
a = 0
b = 1
itr = 0

while itr < length
    print(a, ", ")
    c = a + b
    global a = b
    global b = c
    global itr += 1
end
```

Объяснение: Цикл генерирует первые 15 чисел Фибоначчи, обновляя переменные `a` и `b` на каждой итерации. `global` необходим для изменения глобальных переменных.

Цикл for

Цикл `for` упрощает итерацию по коллекциям и диапазонам:

```
julia> for i = 1:3
    println(i)
end
1
2
3
```

Объяснение: `1:3` — диапазон чисел, `for` итерирует по нему, переменная `i` существует только внутри цикла.

Итерация по массивам

```
julia> for i in [1, 4, 0]
       println(i)
       end
1
4
0
```

Объяснение: Можно использовать `in` или `=` для итерации. Переменная `i` принимает каждое значение из массива по порядку.

Итерация по словарю

```
julia> d = Dict("a" => 1, "b" => 2)
julia> for (k, v) in d
       println("$k => $v")
       end
a => 1
b => 2
```

Объяснение: Можно итерировать по ключам и значениям словаря, используя кортеж `(k, v)`.

Множественная итерация с zip

Функция `zip` объединяет элементы из нескольких коллекций в кортежи, позволяя итерировать по ним одновременно:

```
julia> for (j, k) in zip([1, 2, 3], [4, 5, 6, 7])
    println((j,k))
end
(1, 4)
(2, 5)
(3, 6)
```

Объяснение: `zip` создаёт итератор, который последовательно возвращает кортежи из соответствующих элементов коллекций. Итерация останавливается, когда заканчивается самая короткая коллекция (здесь `[1, 2, 3]` имеет 3 элемента, поэтому последний элемент `7` массива `[4, 5, 6, 7]` не обрабатывается).

Вложенные циклы (Декартово произведение)

```
julia> for i = 1:2, j = 3:4 # for i = 1:2
    println((i, j))         #   for j = 3:4
end                         #       println((i, j))
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

Объяснение: Запятая между циклами эквивалентна вложенным циклам, создавая декартово произведение. `break` прерывает только самый внутренний уровень.

Lifetime переменной цикла

Переменная цикла существует только внутри тела цикла. Если нужно сохранить значение, нужно присвоить его вне цикла:

```
julia> for i in 1:3
           local_i = i
           println(local_i)
       end
1
2
3

julia> println(i) # Ошибка: i не существует вне цикла
```

Генераторы списков

Генераторы создают коллекции на основе выражений:

```
julia> [i^2 for i in 1:5 if i % 2 == 0]
2-element Vector{Int64}:
  4
 16
```

Объяснение: Создает массив квадратов чётных чисел от 1 до 5.

Управление циклами: break и continue

break

Прерывает цикл немедленно:

```
julia> for j = 1:1000
    println(j)
    if j >= 3
        break
    end
end
1
2
3
```

continue

Пропускает остальную часть итерации, переходит к следующей:

```
julia> for i = 1:10
    if i % 3 != 0
        continue
    end
    println(i)
end
3
6
9
```

Обработка исключений (Exception Handling)

Конструкция try-catch

Блок `try-catch` позволяет перехватывать исключения:

```
julia> try
    sqrt("ten")
catch e
    println("You should have entered a numeric value")
end
You should have entered a numeric value
```

Объяснение: Код в `try` может вызвать исключение (здесь `sqrt("ten")` пытается вычислить корень из строки). Если исключение возникает, выполнение переходит в блок `catch`, где переменная `e` содержит объект исключения.

try-catch-finally с rethrow

```
function readfile(filename)
    f = open(filename, "r")
    try
        data = read(f, String)
        println("File content: ", data)
    catch e
        println("Error occurred: ", e)
        rethrow()  # повторно выбрасываем исключение
    finally
        close(f)
        println("File closed")
    end
end
```

Объяснение: `finally` выполняется всегда, даже если исключение было перехвачено и повторно выброшено.

Пример с проверкой типа исключения

```
julia> sqrt_safe(x) = try
           sqrt(x)
       catch e
           isa(e, DomainError) ? sqrt(complex(x, 0)) : rethrow()
       end
```

Объяснение: Функция пытается вычислить `sqrt(x)`. Для отрицательных значений возникает `DomainError`, тогда вычисляется корень в комплексных числах. Остальные ошибки пробрасываются дальше через `rethrow()`.

Создание и выброс исключений

throw

`throw` создаёт исключение указанного типа:

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError(x, "argument must be non-negative"))
julia> f(1)
0.36787944117144233
julia> f(-1)
ERROR: DomainError with -1: argument must be non-negative
```

Объяснение: `throw` передаёт управление ближайшему блоку `catch` или завершает программу с ошибкой.

error

`error` создаёт исключение типа `ErrorException` с текстовым сообщением:

```
julia> strict_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
julia> strict_sqrt(2)
1.4142135623730951
julia> strict_sqrt(-1)
ERROR: negative x not allowed
```

Объяснение: Удобный способ выброса исключения с сообщением для пользователя.

Блок else (Julia 1.8+)

```
local x
try
    f = open("file", "r")
    x = read(f, String)
    close(f)
catch
    # обработка ошибок чтения
else
    # действия с x, если ошибок не было
end
```

Объяснение: Блок `else` выполняется только если исключений не было. Преимущество: ошибки в `else` не перехватываются блоком `catch`.

Пользовательские исключения

Можно создавать собственные типы исключений:

```
julia> struct MyCustomException <: Exception end
```

Объяснение: Пользовательские исключения создаются как подтипы типа `Exception` с использованием оператора `<:` (наследование типов).

Tasks (Задачи/Корутины)

Tasks в Julia обеспечивают кооперативную многозадачность: вы явно планируете и возобновляете вычисления.

Создание Task

```
julia> task = @task sum(rand(1_000_000))
julia> schedule(task)
julia> fetch(task)
```

Смысл: `@task` создаёт задачу, `schedule` запускает её, `fetch` возвращает результат. Задача не выполняется до вызова `schedule`.

Асинхронное выполнение с @async

```
julia> t = @async begin
           for i in 1:3
               println("Async task iteration ", i)
               sleep(0.5)
           end
       end

julia> wait(t)
```

Смысл: `@async` запускает задачу, которая выполняется параллельно с основным кодом.

Параллельное выполнение с @spawn

```
julia> f = @spawn begin
           sum = 0
           for i in 1:10^7
               sum += i
           end
           sum
       end

julia> fetch(f)
50000005000000
```

Смысл: `@spawn` запускает задачу, которая может выполняться в другом потоке или процессе, позволяя эффективно использовать ресурсы.

Разница в подходах @spawn @task @async

- `@task` (ручное планирование)
  - Создаёт задачу без немедленного запуска.
  - Запуск через `schedule(task)`, результат через `fetch(task)`.
  - Подходит, когда нужно отложить старт или передать задачу планировщику явно.

```
task = @task heavy_compute()
schedule(task)
result = fetch(task)
```

Разница в подходах @spawn @task @async

- `@async` (кооперативная конкурентность в том же процессе)
  - Немедленно планирует задачу на выполнение; возвращает `Task`.
  - Хорошо для I/O-заданий и ожиданий; использовать `wait(task)` или `fetch(task)`.

```
t = @async begin
    data = download(url)
    parse(data)
end
wait(t)
```

- `@spawn` (многопоточность/пул потоков)
  - Планирует выполнение на пуле потоков; может выполняться параллельно на другом ядре.
  - Хорошо для CPU-нагруженных задач; результат через `fetch(handle)`.

```
h = @spawn heavy_compute()
ans = fetch(h)
```

Кратко: `@async` для конкурентных I/O, `@spawn` для параллельных CPU-задач, `@task` — когда нужен явный контроль жизненного цикла задачи.

Источники

- Control Flow — Julia Documentation