Функции и обработка ошибок в языке Julia

Перцев Дмитрий, Швачко Никита

группа 5030102/20202

Санкт-Петербургский политехнический университет Петра Великого

21 ноября 2025 г.

Введение в функции в Julia

- Функция — основной способ организации кода
- Определяется с помощью ключевого слова `function` или краткой записи
- Может принимать параметры и возвращать результат

```
function greet(name)
    return "Привет, $name!"
end
```

Определяем функцию `greet`, которая принимает имя и возвращает приветствие. Функции помогают структурировать программу и повторно использовать код.

Краткая запись функций

- Можно определить функцию в одной строке с помощью `->`

```
square(x) = x^2
```

Функция `square` возвращает квадрат числа `x`. Такой синтаксис удобен для простых функций, делает код короче.

Аргументы и параметры функций

- Позиционные аргументы
- Аргументы по умолчанию
- Именованные аргументы

```
function power(base, exponent=2)
    return base^exponent
end

power(3)           # 9
power(3, 3)        # 27
power(base=2)      # 4
power(exponent=3, base=2)  # 8
```

Функция `power` возводит число в степень. Параметры с значениями по умолчанию и именованные аргументы делают вызов функции гибким.

Явное указание типов в аргументах

- В Julia можно явно указывать типы аргументов, но это необязательно
- Указание типов улучшает производительность и делает код более понятным
- Без указания типов Julia использует динамическую типизацию

```
#           Без указания типов (динамическая типизация)
function add(a, b)
    return a + b
end
#           С явным указанием типов
function add_typed(a::Int, b::Int)
    return a + b
end
#           Можно комбинировать типизированные и нетипизированные аргументы
function multiply(x::Float64, y)
    return x * y
end

add(2, 3)               # 5
add_typed(2, 3)          # 5
multiply(2.5, 4)         # 10.0
```

Явное указание типов помогает компилятору Julia генерировать более эффективный код, но функция остается гибкой — можно указывать типы только там, где это важно для производительности или ясности.

Множественные возвращаемые значения

- Julia позволяет возвращать несколько значений из функции

```
function div_mod(a, b)
    return div(a, b), mod(a, b)
end

quotient, remainder = div_mod(10, 3)
println("Частное: $quotient, Остаток: $remainder")
```

Функция возвращает частное и остаток от деления одновременно — удобно получать сразу несколько результатов.

Анонимные функции

- Функции без имени, часто используются внутри других функций

```
map(x -> x^2, [1, 2, 3, 4])  # [1, 4, 9, 16]
```

Создаем функцию для возведения в квадрат и применяем её к каждому элементу списка, упрощая код без необходимости давать имя функции.

Множественная диспетчеризация (Multiple Dispatch)

- Ключевая особенность **Julia** — выбор метода функции на основе типов всех аргументов
- Одна функция может иметь несколько реализаций для разных типов
- Позволяет писать полиморфный код без наследования

```julia
function add(x::Int, y::Int)
    return x + y
end

function add(x::String, y::String)
    return x * y
end

add(2, 3)           # 5
add("Hello, ", "World!")  # "Hello, World!"
```

Множественная диспетчеризация — это то, что делает **Julia** уникальной. **Julia** выбирает правильную реализацию функции на основе типов всех аргументов, а не только первого, как в объектно-ориентированных языках.

Функции с переменным числом аргументов (Varargs)

- Можно определить функции, принимающие произвольное количество аргументов
- Используется `...` для обозначения переменного числа аргументов

```
function sum_all(args...)
    total = 0
    for arg in args
        total += arg
    end
    return total
end

sum_all(1, 2, 3)        # 6
sum_all(1, 2, 3, 4, 5)  # 15
```

Функция `sum_all` принимает любое количество аргументов и суммирует их, демонстрируя гибкость Julia в работе с аргументами.

Функции высшего порядка

- Функции, которые принимают другие функции в качестве аргументов
- `map`, `filter`, `reduce` — стандартные функции высшего порядка

```
numbers = [1, 2, 3, 4, 5]

# map — применяет функцию к каждому элементу
squares = map(x -> x^2, numbers)  # [1, 4, 9, 16, 25]

# filter — фильтрует элементы по условию
evens = filter(x -> x % 2 == 0, numbers)  # [2, 4]

# reduce — сводит коллекцию к одному значению
sum = reduce(+, numbers)  # 15
```

Функции высшего порядка позволяют писать декларативный и элегантный код, работая с коллекциями данных функциональным стилем.

Замыкания (Closures)

- Функции, которые захватывают переменные из внешней области видимости
- Сохраняют состояние между вызовами

```
function make_counter(start=0)
    count = start
    return function()
        count += 1
        return count
    end
end

counter = make_counter(10)
counter()  # 11
counter()  # 12
counter()  # 13
```

Замыкания позволяют создавать функции с "памятью", которые помнят значения переменных из момента их создания — мощный инструмент для создания специализированных функций.

Введение в обработку ошибок в Julia

- Ошибки — важная часть управления выполнением программы
- В **Julia** ошибки являются исключениями (exceptions)
- Используются конструкции `try-catch` для обработки ошибок

```
try
    x = 1 / 0
catch e
    println("Ошибка: ", e)
end
```

Попытка деления на ноль вызывает ошибку, которая перехватывается и выводится сообщение, предотвращая аварийное завершение программы.

Конструкция try-catch-finally

- `try` — блок с кодом, который может вызвать ошибку
- `catch` — обработка ошибки
- `finally` — код, который выполнится в любом случае

```
try
    println("Открываем файл")
    # Операция с файлом
catch e
    println("Ошибка при работе с файлом: ", e)
finally
    println("Закрываем файл")
end
```

Обеспечивается безопасное открытие и закрытие ресурсов, даже если произошла ошибка — важный паттерн для надежных программ.

Создание и выброс собственных ошибок

- Для создания ошибочного состояния можно генерировать исключения

```
function check_age(age)
    if age < 0
        error("Возраст не может быть отрицательным!")
    end
    return age
end
```

Если возраст некорректный, функция инициирует ошибку, которая сигнализирует о неправильном входном значении.

Иерархия типов ошибок в Julia

```
Exception
├── ErrorException
├── ArgumentError
├── BoundsError
├── DomainError
└── ... и другие
```

Типы ошибок позволяют точечно обрабатывать разные ситуации, например, ошибку аргумента или выход за границы массива.

Кастомные типы ошибок

- Можно создавать собственные типы ошибок, наследуя `Exception`

```
struct MyError <: Exception
    msg::String
end

function test(x)
    if x < 0
        throw(MyError("Значение должно быть неотрицательным"))
    end
end
```

Создаем уникальный тип ошибки для специфических ситуаций, что облегчает её идентификацию и обработку.

Отлавливание конкретных типов ошибок

```
try
    test(-1)
catch e::MyError
    println("Поймана моя ошибка: ", e.msg)
catch e
    println("Другая ошибка: ", e)
end
```

Можно обрабатывать разные ошибки разными способами, что улучшает управление ошибками в программе.

Пример: функция с обработкой ошибок ввода

```
function read_number()
    try
        println("Введите число:")
        num = parse(Int, readline())
        return num
    catch e
        println("Ошибка ввода, попробуйте еще раз.")
        return read_number()
    end
end

n = read_number()
println("Вы ввели: $n")
```

Функция рекурсивно запрашивает ввод, пока пользователь не введет корректное число, демонстрируя полезное применение обработки ошибок.

Итоги

- Функции — основа программирования в **Julia**, обеспечивают структуру и повторное использование кода
- Множественная диспетчеризация — ключевая особенность **Julia**, позволяющая выбирать метод на основе типов всех аргументов
- В **Julia** поддерживаются разные способы определения и использования функций: анонимные, замыкания, функции высшего порядка
- Функции с переменным числом аргументов обеспечивают гибкость при работе с данными
- Обработка ошибок с помощью `try-catch` позволяет создавать устойчивые программы
- Свои типы ошибок упрощают архитектуру обработки исключений
- Важно уметь аккуратно использовать эти инструменты для надежного и эффективного кода

Спасибо за внимание!