Конкурентность и параллелизм в Julia

Перцев Дмитрий, Никита Швачко

Содержание

Введение

- Julia разработана для высокопроизводительных вычислений
- Встроенная поддержка конкурентности и параллелизма
- Различие понятий:
  - **Параллелизм** – одновременное выполнение множества задач
  - **Конкурентность** – управление множеством задач одновременно

Задачи (Tasks) - Часть 1

Создание и управление задачами

```
# @task - создает новую задачу (не запускает автоматически)
# Задача - это легковесный поток, который выполняется конкурентно
task = @task begin
    for i in 1:3
        println("Итерация задачи: ", i)
        sleep(0.1)  # Имитация работы - блокирующая операция
    end
    return "Задача завершена"  # Возвращает результат
end

# schedule - планирует выполнение задачи в пуле задач
# Не блокирует текущий поток, задача выполняется в фоне
schedule(task)

# fetch - ожидает завершение задачи и возвращает ее результат
# Блокирующая операция - поток ждет завершения задачи
result = fetch(task)
println("Результат задачи: ", result)
```

Задачи (Tasks) - Часть 2

Упрощенное создание и синхронизация задач

```
# @async - создает и немедленно запускает задачу
# Аналог комбинации @task + schedule
@async begin
    println("Задача запущена в фоне")
    sleep(1)  # Долгая операция
    println("Фоновая задача завершена")
end

# @sync - ожидает завершения всех внутренних асинхронных операций
# Гарантирует, что все созданные внутри задачи завершатся
@sync begin
    for i in 1:3
        @async begin
            # sleep(rand()) - каждая задача имеет разное время выполнения
            sleep(rand())  # Случайная задержка от 0 до 1 секунды
            println("Асинхронная задача $i завершена")
        end
    end
end
# Эта строка выполнится только после ВСЕХ асинхронных задач
println("Все задачи завершены - контроль возвращен")
```

Каналы (Channels) - Часть 1

Базовое использование каналов для обмена данными

```
# Channel{Int}(5) - создает канал для целых чисел емкостью 5
# Канал - потокобезопасная очередь для обмена данными между задачами
ch = Channel{Int}(5)

# Производитель - задача, которая отправляет данные в канал
@async for i in 1:3
    # put! - отправляет значение в канал
    # Если канал полон, операция блокируется до появления места
    put!(ch, i)
    println("Производитель отправил: ", i)
    sleep(0.2)  # Имитация времени на производство данных
end

# Потребитель - задача, которая получает данные из канала
@async for i in 1:3
    # take! - получает значение из канала
    # Если канал пуст, операция блокируется до появления данных
    value = take!(ch)
    println("Потребитель получил: ", value)
end
```

Каналы (Channels) - Часть 2

Дополнительные функции каналов

```
# close(ch) - закрывает канал, сигнализируя о завершении работы
# isopen(ch) - проверяет, открыт ли канал для операций
function producer_consumer_example()
    ch = Channel{String}(3)

    @async begin
        names = ["Алиса", "Боб", "Чарли", "Дина"]
        for name in names
            if isopen(ch)
                put!(ch, name)
                println("Отправлено: $name")
            end
        end
        close(ch)  # Закрываем канал после отправки всех данных
    end

    @async begin
        while isopen(ch)
            # Безопасное получение данных с проверкой
            name = take!(ch)
            println("Получено: $name")
        end
    end
end
```

Модели параллелизма

- **Многопроцессный параллелизм**
  - Изоляция памяти между процессами
  - `addprocs()` - добавляет рабочие процессы
  - `@everywhere` - выполняет код на всех процессах
- **Многопоточность**
  - Разделяемая память между потоками
  - `Threads.@threads` - параллельные циклы
  - `Atomic` - атомарные операции

Распределенные вычисления между процессами

```
# using Distributed - подключает модуль для распределенных вычислений
# Многопроцессность: каждый процесс имеет свою память
using Distributed

# addprocs(2) - добавляет 2 рабочих процесса к основному
# Процессы могут быть на том же компьютере или на удаленных узлах
addprocs(2)

# @everywhere - выполняет код на ВСЕХ процессах (основном и рабочих)
# Необходимо для определения функций и данных на всех процессах
@everywhere function heavy_computation(x)
    # Имитация тяжелых вычислений - каждая итерация независима
    sleep(0.1)  # Блокирующая операция
    return x * x  # Возвращает квадрат числа
end
```

Распределение вычислений и сбор результатов

```
# @distributed - распределяет итерации цикла по рабочим процессам
# (+) - операция редукции (сложения) для объединения результатов
result = @distributed (+) for x in 1:10
    heavy_computation(x)  # Каждая итерация выполняется на своем процессе
end

println("Сумма квадратов (параллельно): ", result)

# Альтернативный способ с явным указанием функции редукции
@everywhere function custom_reduce(a, b)
    return a + b  # Пользовательская операция редукции
end

result2 = @distributed custom_reduce for x in 1:5
    x * x * x  # Вычисляем кубы
end
println("Сумма кубов: ", result2)
```

Удаленные вызовы функций - Часть 1

Выполнение кода на конкретных процессах

```
using Distributed

# @spawnat - выполняет код на указанном процессе
# 2 - номер целевого процесса (один из рабочих процессов)
# Возвращает RemoteRef - ссылку на удаленный результат
remote_ref = @spawnat 2 begin
    # Этот код выполняется ТОЛЬКО на процессе 2
    sum = 0
    for i in 1:1000
        sum += i  # Суммируем числа от 1 до 1000
    end
    sum  # Результат вычислений на процессе 2
end

# fetch - получает результат удаленного вычисления
# Блокирующая операция - ждет завершения вычисления на удаленном процессе
result = fetch(remote_ref)
println("Сумма от 1 до 1000 (вычислено на процессе 2): ", result)
```

Удаленные вызовы функций - Часть 2

Параллельное отображение и удаленные вызовы

```
# pmap - параллельное отображение функции на коллекцию
# Распределяет вызовы функции по доступным процессам
data = [1, 2, 3, 4, 5]
# x -> x * x - лямбда-функция, вычисляющая квадрат
squares = pmap(x -> x * x, data)  # [1, 4, 9, 16, 25]
println("Квадраты чисел: ", squares)

# remotecall - альтернативный способ удаленного вызова
# remotecall(функция, процесс, аргументы...)
remote_square = remotecall(x -> x * x, 2, 10)
square_result = fetch(remote_square)
println("Квадрат 10 (вычислено на процессе 2): ", square_result)

# @everywhere для сложных вычислений
@everywhere function matrix_multiply(A, B)
    return A * B  # Умножение матриц на каждом процессе
end
```

Параллельное выполнение циклов с разделяемой памятью

```
# using Base.Threads - подключает модуль многопоточности
# Многопоточность: потоки разделяют общую память
using Base.Threads

# nthreads() - возвращает количество доступных потоков
# threadid() - возвращает идентификатор текущего потока
println("Доступно потоков: ", nthreads())
println("Текущий поток: ", threadid())

function parallel_array_operation(arr)
    # similar(arr) - создает массив того же размера и типа
    result = similar(arr)

    # @threads - распределяет итерации цикла по потокам
    # Автоматически делит диапазон итераций между потоками
    @threads for i in eachindex(arr)
        # Каждый поток работает со своей частью массива
        # Все потоки имеют доступ к arr и result (разделяемая память)
        result[i] = arr[i] * arr[i] + sin(arr[i])
    end

    return result
end
```

Многопоточность - Часть 2

Использование и измерение производительности

```
data = rand(1000)  # Создаем массив случайных чисел
@time parallel_array_operation(data)  # Измеряем время выполнения

# Параллельная редукция с локальными переменными
function parallel_sum(arr)
    local_sums = zeros(nthreads())  # Локальная сумма для каждого потока

    @threads for i in eachindex(arr)
        tid = threadid()  # ID текущего потока
        local_sums[tid] += arr[i]  # Каждый поток суммирует в свою ячейку
    end

    return sum(local_sums)  # Суммируем все локальные суммы
end

large_data = rand(10^6)
sequential_time = @elapsed sum(large_data)
parallel_time = @elapsed parallel_sum(large_data)
println("Ускорение: $(sequential_time/parallel_time)x")
```

Атомарные операции

Потокобезопасные операции с атомарными переменными

```
using Base.Threads

# Atomic{Int}(0) - создает атомарную целочисленную переменную
# Атомарные операции неделимы - гарантируют корректность при параллельном доступе
function thread_safe_counter(n_iterations)
    counter = Atomic{Int}(0)  # Инициализируем атомарный счетчик нулем

    # Параллельный цикл с множеством потоков
    @threads for i in 1:n_iterations
        # atomic_add! - атомарно увеличивает счетчик на 1
        # Гарантирует, что даже при одновременном доступе не будет гонки данных
        atomic_add!(counter, 1)
    end

    return counter[]  # counter[] - получаем значение атомарной переменной
end

# atomic_sub!(counter, value) - атомарное вычитание
# atomic_xchg!(counter, value) - атомарная замена значения

println("Безопасный счетчик: ", thread_safe_counter(10000))
```

Мьютексы и блокировки - Часть 1

Синхронизация доступа к разделяемым ресурсам

```
using Base.Threads

function synchronized_operations()
    # ReentrantLock() - создает повторно входимый мьютекс
    # Мьютекс (mutual exclusion) - гарантирует exclusive доступ к ресурсу
    lock = ReentrantLock()
    shared_data = []  # Разделяемый ресурс между потоками

    @threads for i in 1:10
        # lock(lock) do ... end - захватывает мьютекс перед выполнением блока
        # и автоматически освобождает его после завершения блока (даже при ошибке)
        lock(lock) do
            # Критическая секция - код, который должен выполняться только одним потоком о
     дновременно
            push!(shared_data, (threadid(), i))  # Добавляем данные от текущего потока
            println("Поток $(threadid()) добавил данные: ", i)
        end
        # Мьютекс автоматически освобождается здесь
    end

    return shared_data
end
```

Мьютексы и блокировки - Часть 2

Явное использование lock/unlock

```
# Альтернативный способ - явное использование lock/unlock
function explicit_lock_example()
    lock = ReentrantLock()
    counter = 0

    @threads for i in 1:100
        lock(lock)  # Явный захват мьютекса
        try
            # Критическая секция
            counter += 1
            println("Поток $(threadid()) увеличил счетчик до: $counter")
        finally
            unlock(lock)  # Гарантированное освобождение мьютекса
        end
    end

    return counter
end

# try-finally гарантирует, что мьютекс будет освобожден даже при исключении
# Это важно для предотвращения взаимных блокировок (deadlocks)
```

Семафоры для ограничения параллелизма

Контроль количества одновременных выполнений

```
using Base.Threads

function limited_concurrency()
    # Semaphore(2) - создает семафор с начальным значением 2
    # Семафор ограничивает количество одновременных доступов к ресурсу
    sem = Semaphore(2)  # Максимум 2 параллельных выполнения

    function limited_task(id)
        # lock(sem) - уменьшает счетчик семафора (захватывает его)
        # Если счетчик = 0, поток блокируется до освобождения семафора
        lock(sem)
        try
            println("Задача $id начала выполнение")
            sleep(1)  # Имитация работы, занимающей время
            println("Задача $id завершила выполнение")
        finally
            # unlock(sem) - увеличивает счетчик семафора (освобождает его)
            # finally гарантирует, что семафор будет освобожден даже при ошибке
            unlock(sem)
        end
    end
```

Семафоры для ограничения параллелизма

Контроль количества одновременных выполнений

```
    # @sync - ждет завершения всех асинхронных задач
    @sync for i in 1:5
        # @async - запускает задачу асинхронно
        @async limited_task(i)
    end
end
```

Преимущества Julia для параллельных вычислений

- Единый язык для всех уровней параллелизма
- Эффективная модель распределенных вычислений
- Простая интеграция с C/Fortran кодом
- Богатая экосистема пакетов
- Минимальные накладные расходы
- Встроенные инструменты отладки и профилирования

Ресурсы для дальнейшего изучения

- Официальная документация: `docs.julialang.org`
- "The Julia Language официальная книга

Спасибо за внимание!
Вопросы?