



skovorodkin 3 февраля 2014 в 21:48

Почему я делаю ставку на Julia

Автор оригинала: Evan Miller

Программирование, Julia

Перевод

Совсем о Julia не говорим тут. Один пост двухлетней давности от Ализара, и всё. Исправляем ситуацию.

Используя разные языки программирования, я постоянно сталкиваюсь с одной и той же проблемой — их создатели без ума от вещей, которые меня практически не волнуют: безопасность, системы типов, гомоиконность и так далее. Всё это очень круто, не спорю, но когда я вожусь по вечерам над своим очередным проектом, мне важна только его работоспособность и производительность. Код — это всего лишь средство для достижения некоторой цели, а его «выразительность» для меня важна так же, как и «выразительность» какого-нибудь каталитического конвертера.



Такой подход к делу некоторые презрительно называют ковбойским программированием. Но мне кажется, что это не самый правильный образ — ковбой вынужден периодически устраивать привалы из-за физических ограничений своего коня. Давайте лучше представим одержимого учёного, эдакого профессора, который неделями пропадает в лаборатории, а потом выходит оттуда изнурённый, с затуманенным взором, со своим новым хитроумным изобретением, которое разваливается при первом же запуске.

Я обычно работаю таким образом: сначала пишу прототип на одном языке, после переписываю критические секции на другом языке, а если мне нужно, чтобы программа летала, я использую третий язык. Такой подход довольно распространён. Для прототипирования многие используют Python, Ruby, R и т.п. Как только всё заработало, некоторые куски кода переписываются на C или C++. Если и этого недостаточно, некоторые циклы переписываются на ассемблере, CUDA или OpenCL.

Но в этом случае мы сталкиваемся со сложными препятствиями. Мало того, что нужно знать три языка, так ещё и приходится постоянно переключаться между разными уровнями абстракции. А в реальной жизни ещё приходится писать связующий код — переключаться между разными файлами, редакторами и дебаггерами.

Некоторое время назад я узнал о Julia, язык мне сразу показался крутым, но особой необходимости в нём я не ощутил. Julia — высокопроизводительный динамический язык. Заманчиво, конечно, но я уже и так угробил кучу времени, чтобы засунуть движок от бэхи в свою шоху — зачем мне ещё что-то? Кроме того, есть куча платформ, которые обещают производительность на уровне C: Java HotSpot, PyPy, asm.js и другие.

И только потом я понял, что отличает Julia от других языков. Julia разрушает барьер между высокоуровневым и ассемблерным кодом. Julia не просто позволяет писать код, который работает так же быстро, как код на C, но и даёт возможность посмотреть на LLVM-представление функций и их сгенерированный ассемблерный код. И всё это прямо в интерактивной среде.

```
emiller ~/Code/julia (master) ./julia
```

```

 _ _ _ _ _ _ _ _ | A fresh approach to technical computing
( ) | ( ) ( ) | Documentation: http://docs.julialang.org
 _ _ _ | | _ _ _ | Type "help()" to list help topics
| | | | | | | | | |
| | | | | | | | | | Version 0.3.0-prerelease+261 (2013-11-30 12:55 UTC)
_/_ | \_/_ | \_/_ | \_/_ | Commit 97b5983 (0 days old master)
|_|/ | | | | | | | | | x86_64-apple-darwin12.5.0

```

```
julia> f(x) = x * x
f (generic function with 1 method)
```

```
julia> f(2.0)
```

4.0

```
julia> code_llvm(f, (Float64,))

define double @julia_f662(double) {
top:
    %1 = fmul double %0, %0, !dbg !3553
    ret double %1, !dbg !3553
}

julia> code_native(f, (Float64,))
.section      __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
    push      RBP
    mov       RBP, RSP
Source line: 1
    vmulsd    XMM0, XMM0, XMM0
    pop       RBP
    ret
```

Вот так вот. Мы можем написать однострочную функцию и тут же изучить её оптимизированный ассемблерный код.

В общем, можете забыть о системе типов Julia, мультиметодах и прочей гомоиконности. Это всё, конечно, интересно, но настоящий козырь Julia в том, что мы можем начать с прототипа, а закончить оптимизацией SIMD-инструкциями не покидая одного языка.

В общем, это основная причина, по которой я делаю ставку на Julia. Мне не терпится сделать такое сравнение: этот язык сделает для технических расчётов то, что Node.js делает для веб-разработки — объединяет разные группы программистов одним языком. Если для Node.js этими группами являются фронтенд- и бекенд-разработчики, то для Julia — специалисты в определенных областях знаний и безумные оптимизаторы. Это большое достижение.

На данный момент единственный недостаток языка — дефицит библиотек. Но и он компенсируется лёгкостью взаимодействия с библиотеками на C. В отличие от интерфейсов для взаимодействия других языков, здесь можно вызвать функцию C, не написав ни строчки на C, поэтому, мне кажется, количество библиотек скоро начнёт свой быстрый рост. Если говорить о собственном опыте, то мне удалось без какого-либо дополнительного кода на C воспользоваться 5 тысячами строк кода на C в своих 150 строках кода на Julia.

Если вам приходится поддерживать код из смеси Python, C, C++, Fortran и R или вы просто, как и я, одержимы производительностью, то настоятельно вам рекомендую скачать Julia и дать ей шанс. Если вы не уверены, стоит ли усложнять свою жизнь ещё одним языком программирования, то просто осознайте, что этот инструмент в итоге позволит вам сократить количество языков в ваших проектах.

В конце концов, если забыть на минуту о производительности, Julia — невероятно красивый язык. Я не из гиков, но я почти ни разу не столкнулся с какими-нибудь трудностями, пока изучал язык. И сейчас Julia для меня один из трёх предпочтительных языков.

В конце концов, вокруг Julia собралось активное сообщество, всегда готовое прийти на помощь. Особенно меня радует, что немалую его часть составляют умные и дружелюбные математики и представители других наук. Думаю, так получилось именно потому, что Julia была создана не гиками, а студентами точных наук из MIT, которым нужен быстрый и удобный язык на замену C и Фортрану. И создан он был не для того, чтобы быть красивым (хоть и получился таким). Создан он был для быстрого получения ответов. А это, я считаю, и является сутью всей нашей компьютерной науки.

Конец поста Эвана Миллера.

Пост, похожий по настроению, появился сегодня [на Wired](#).

Julia вкратце

Чтобы два раза не вставать, представляю перевод документа [Learn Julia in minutes](#).

[Julia REPL](#).

[illegible]

```

# Индексирование не всегда правильно работает для UTF8-строк,
# поэтому рекомендуется использовать итерирование (tap, for-циклы и т.п.).

# Для строковой интерполяции используется знак доллара ($):
"2 + 2 = $(2 + 2)" #=> "2 + 2 = 4"
# В скобках можно использовать любое выражение языка.

# Другой способ форматирования строк – макрос printf
@printf "%d is less than %f" 4.5 5.3 # 5 is less than 5.300000

#####
## 2. Переменные и коллекции
#####

# Вывод
println("I'm Julia. Nice to meet you!")

# Переменные инициализируются без предварительного объявления
some_var = 5 #=> 5
some_var #=> 5

# Попытка доступа к переменной до инициализации вызывает ошибку
try
    some_other_var #=> ERROR: some_other_var not defined
catch e
    println(e)
end

# Имена переменных начинаются с букв.
# После первого символа можно использовать буквы, цифры,
# символы подчёркивания и восклицательные знаки.
SomeOtherVar123! = 6 #=> 6

# Допустимо использование unicode-символов
= 8 #=> 8
# Это особенно удобно для математических обозначений
2 * π #=> 6.283185307179586

# Рекомендации по именованию:
# * имена переменных в нижнем регистре, слова разделяются символом
#   подчёркивания ('\_');
#
# * для имён типов используется CamelCase;
#
# * имена функций и макросов в нижнем регистре
#   без разделения слов символом подчёркивания;
#
# * имя функции, изменяющей переданные ей аргументы (in-place function),
#   оканчивается восклицательным знаком.

# Массив хранит последовательность значений, индексируемых с единицы до n:
a = Int64[] #=> пустой массив Int64-элементов

# Одномерный массив объявляется разделёнными запятой значениями.
b = [4, 5, 6] #=> массив из трёх Int64-элементов: [4, 5, 6]
b[1] #=> 4
b[end] #=> 6

# Строки двумерного массива разделяются точкой с запятой.
# Элементы строк разделяются пробелами.
matrix = [1 2; 3 4] #=> 2x2 Int64 Array: [1 2; 3 4]

# push! и append! добавляют в список новые элементы
push!(a,1) #=> [1]
push!(a,2) #=> [1,2]
push!(a,4) #=> [1,2,4]
push!(a,3) #=> [1,2,4,3]
append!(a,b) #=> [1,2,4,3,4,5,6]

# pop! удаляет из списка последний элемент
pop!(b) #=> возвращает 6; массив b снова равен [4,5]

```

```

# Вернём b обратно
push!(b,6) # b снова [4,5,6].

a[1] #=> 1 # индексы начинаются с единицы!

# Последний элемент можно получить с помощью end
a[end] #=> 6

# Операции сдвига
shift!(a) #=> 1 and a is now [2,4,3,4,5,6]
unshift!(a,7) #=> [7,2,4,3,4,5,6]

# Восклицательный знак на конце названия функции означает,
# что функция изменяет переданные ей аргументы.
arr = [5,4,6] #=> массив из 3 Int64-элементов: [5,4,6]
sort(arr) #=> [4,5,6]; но arr равен [5,4,6]
sort!(arr) #=> [4,5,6]; а теперь arr – [4,5,6]

# Попытка доступа за пределами массива выбрасывает BoundsError
try
    a[0] #=> ERROR: BoundsError() in getindex at array.jl:270
    a[end+1] #=> ERROR: BoundsError() in getindex at array.jl:270
catch e
    println(e)
end

# Вывод ошибок содержит строку и файл, где произошла ошибка,
# даже если это случилось в стандартной библиотеке.
# Если вы собрали Julia из исходных кодов,
# то найти эти файлы можно в директории base.

# Создавать массивы можно из последовательности
a = [1:5] #=> массив из 5 Int64-элементов: [1,2,3,4,5]

# Срезы
a[1:3] #=> [1, 2, 3]
a[2:] #=> [2, 3, 4, 5]
a[2:end] #=> [2, 3, 4, 5]

# splice! удаляет элемент из массива
# Remove elements from an array by index with splice!
arr = [3,4,5]
splice!(arr,2) #=> 4 ; arr теперь равен [3,5]

# append! объединяет списки
b = [1,2,3]
append!(a,b) # теперь a равен [1, 2, 3, 4, 5, 1, 2, 3]

# Проверка на вхождение
in(1, a) #=> true

# Длина списка
length(a) #=> 8

# Кортеж – неизменяемая структура.
tup = (1, 2, 3) #=> (1,2,3) # кортеж (Int64,Int64,Int64).
tup[1] #=> 1
try:
    tup[1] = 3 #=> ERROR: no method setindex!((Int64,Int64,Int64),Int64,Int64)
catch e
    println(e)
end

# Многие функции над списками работают и для кортежей
length(tup) #=> 3
tup[1:2] #=> (1,2)
in(2, tup) #=> true

# Кортежи можно распаковывать в переменные
a, b, c = (1, 2, 3) #=> (1,2,3) # a = 1, b = 2 и c = 3

```

```

# Скобки из предыдущего примера можно опустить
d, e, f = 4, 5, 6 #=> (4,5,6)

# Кортеж из одного элемента не равен значению этого элемента
(1,) == 1 #=> false
(1) == 1 #=> true

# Обмен значений
e, d = d, e #=> (5,4) # d = 5, e = 4

# Словари содержат ассоциативные массивы
empty_dict = Dict{Any,Any}()

# Для создания словаря можно использовать литерал
filled_dict = ["one" => 1, "two" => 2, "three" => 3]
# => Dict{ASCIIString,Int64}

# Значения ищутся по ключу с помощью оператора []
filled_dict["one"] #=> 1

# Получить все ключи
keys(filled_dict)
#=> KeyIterator{Dict{ASCIIString,Int64}}{["three"=>3,"one"=>1,"two"=>2]}
# Заметьте, словарь не запоминает порядок, в котором добавляются ключи.

# Получить все значения.
values(filled_dict)
#=> ValueIterator{Dict{ASCIIString,Int64}}{["three"=>3,"one"=>1,"two"=>2]}
# То же касается и порядка значений.

# Проверка вхождения ключа в словарь
in(("one", 1), filled_dict) #=> true
in(("two", 3), filled_dict) #=> false
haskey(filled_dict, "one") #=> true
haskey(filled_dict, 1) #=> false

# Попытка обратиться к несуществующему ключу выбросит ошибку
try
    filled_dict["four"] #=> ERROR: key not found: four in getindex at dict.jl:489
catch e
    println(e)
end

# Используйте метод get со значением по умолчанию, чтобы избежать этой ошибки
# get(dictionary, key, default_value)
get(filled_dict, "one", 4) #=> 1
get(filled_dict, "four", 4) #=> 4

# Для коллекций неотсортированных уникальных элементов используйте Set
empty_set = Set{Any}()
# Инициализация множества
filled_set = Set{Int64}(1,2,3,4)

# Добавление элементов
push!(filled_set, 5) #=> Set{Int64}(5,4,2,3,1)

# Проверка вхождения элементов во множество
in(2, filled_set) #=> true
in(10, filled_set) #=> false

# Функции для получения пересечения, объединения и разницы.
other_set = Set{Int64}(3,4,5,6) #=> Set{Int64}(6,4,5,3)
intersect(filled_set, other_set) #=> Set{Int64}(3,4,5)
union(filled_set, other_set) #=> Set{Int64}(1,2,3,4,5,6)
setdiff(Set{Int64}(1,2,3,4), Set{Int64}(2,3,5)) #=> Set{Int64}(1,4)

#####
## 3. Поток управления

```

```
#####

# Создадим переменную
some_var = 5

# Выражение if. Отступы не имеют значения.
if some_var > 10
    println("some_var is totally bigger than 10.")
elseif some_var < 10 # Необязательная ветка elseif.
    println("some_var is smaller than 10.")
else # else-ветка также опциональна.
    println("some_var is indeed 10.")
end
#=> prints "some var is smaller than 10"

# Цикл for проходит по итерируемым объектам
# Примеры итерируемых типов: Range, Array, Set, Dict и String.
for animal=["dog", "cat", "mouse"]
    println("$animal is a mammal")
    # Для вставки значения переменной или выражения в строку используется $
end
# Выведет:
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

# Другой вариант записи.
for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
end
# Выведет:
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

for a in ["dog"=>"mammal", "cat"=>"mammal", "mouse"=>"mammal"]
    println("$a is a $v")
end
```

Все потоки Разработка Администрирование Дизайн Менеджмент Маркетинг Научпоп



Войти

Регистрация

```
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

for (k,v) in ["dog"=>"mammal", "cat"=>"mammal", "mouse"=>"mammal"]
    println("$k is a $v")
end
# Выведет:
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

# Цикл while выполняется до тех пор, пока верно условие
x = 0
while x < 4
    println(x)
    x += 1 # Короткая запись x = x + 1
end
# Выведет:
#   0
#   1
#   2
#   3

# Обработка исключений
try
    error("help")
catch e
    println("caught it $e")
end
#=> caught itErrorException("help")
```

```
#####
## 4. Функции
#####

# Для определения новой функции используется ключевое слово 'function'
function имя(аргументы)
# тело...
end

function add(x, y)
    println("x is $x and y is $y")

    # Функция возвращает значение последнего выражения
    x + y
end

add(5, 6) #=> Вернёт 11, напечатает "x is 5 and y is 6"

# Функция может принимать переменное количество позиционных аргументов.
function varargs(args...)
    return args
    # для возвращения из функции в любом месте используется 'return'
end
#=> varargs (generic function with 1 method)

varargs(1,2,3) #=> (1,2,3)

# Многоточие (...) – это splat.
# Мы только что воспользовались им в определении функции.
# Также его можно использовать при вызове функции,
# где он преобразует содержимое массива или кортежа в список аргументов.
Set([1,2,3]) #=> Set{Array{Int64,1}}{([1,2,3])} # формирует множество массивов
Set([1,2,3]...) #=> Set{Int64}{1,2,3} # эквивалентно Set(1,2,3)

x = (1,2,3) #=> (1,2,3)
Set(x) #=> Set{Int64,Int64,Int64}{(1,2,3)} # множество кортежей
Set(x...) #=> Set{Int64}{2,3,1}

# Опциональные позиционные аргументы
function defaults(a,b,x=5,y=6)
    return "$a $b and $x $y"
end

defaults('h','g') #=> "h g and 5 6"
defaults('h','g','j') #=> "h g and j 6"
defaults('h','g','j','k') #=> "h g and j k"
try
    defaults('h') #=> ERROR: no method defaults(Char,)
    defaults() #=> ERROR: no methods defaults()
catch e
    println(e)
end

# Именованные аргументы
function keyword_args(;k1=4,name2="hello") # обратите внимание на ;
    return ["k1"=>k1,"name2"=>name2]
end

keyword_args(name2="ness") #=> ["name2"=>"ness", "k1"=>4]
keyword_args(k1="mine") #=> ["k1"=>"mine", "name2"=>"hello"]
keyword_args() #=> ["name2"=>"hello", "k2"=>4]

# В одной функции можно совмещать все виды аргументов
function all_the_args(normal_arg, optional_positional_arg=2; keyword_arg="foo")
    println("normal arg: $normal_arg")
    println("optional arg: $optional_positional_arg")
    println("keyword arg: $keyword_arg")
end
```



```

all_the_args(1, 3, keyword_arg=4)
# Выведет:
#   normal arg: 1
#   optional arg: 3
#   keyword arg: 4

# Функции в Julia первого класса
function create_adder(x)
    adder = function (y)
        return x + y
    end
    return adder
end

# Анонимная функция
(x -> x > 2)(3) ==> true

# Эта функция идентичная предыдущей версии create_adder
function create_adder(x)
    y -> x + y
end

# Если есть желание, можно воспользоваться полным вариантом
function create_adder(x)
    function adder(y)
        x + y
    end
    adder
end

add_10 = create_adder(10)
add_10(3) ==> 13

# Встроенные функции высшего порядка
map(add_10, [1,2,3]) ==> [11, 12, 13]
filter(x -> x > 5, [3, 4, 5, 6, 7]) ==> [6, 7]

# Списковые сборки
[add_10(i) for i=[1, 2, 3]] ==> [11, 12, 13]
[add_10(i) for i in [1, 2, 3]] ==> [11, 12, 13]

#####
## 5. Типы
#####

# Julia has a type system.
# Каждое значение имеет тип, но переменные не определяют тип значения.
# Функция `typeof` возвращает тип значения.
typeof(5) ==> Int64

# Types are first-class values
# Типы являются значениями первого класса
typeof(Int64) ==> DataType
typeof(DataType) ==> DataType
# Тип DataType представляет типы, включая себя самого.

# Типы используются в качестве документации, для оптимизации и организации.
# Статически типы не проверяются.

# Пользователь может определять свои типы
# Типы похожи на структуры в других языках
# Новые типы определяются с помощью ключевого слова `type`

# type Name
#   field::OptionalType
#   ...
# end
type Tiger
    taillength::Float64
    coatcolor # отсутствие типа равносильно `::Any`

```

```

end

# Аргументы конструктора по умолчанию — свойства типа
# в порядке их определения.
tiger = Tiger(3.5, "orange") #=> Tiger(3.5, "orange")

# Тип объекта по сути является конструктором значений такого типа
sherekhan = typeof(tiger)(5.6, "fire") #=> Tiger(5.6, "fire")

# Эти типы, похожие на структуры, называются конкретными.
# Можно создавать объекты таких типов, но не их подтипы.
# Другой вид типов — абстрактные типы.

# abstract Name
abstract Cat # просто имя и точка в иерархии типов

# Объекты абстрактных типов создавать нельзя,
# но зато от них можно наследовать подтипы.
# Например, Number — это абстрактный тип.
subtypes(Number) #=> 6 элементов в массиве Array{Any,1}:
    #      Complex{Float16}
    #      Complex{Float32}
    #      Complex{Float64}
    #      Complex{T<:Real}
    #      ImaginaryUnit
    #      Real
subtypes(Cat) #=> пустой массив Array{Any,1}

# У всех типов есть супертип. Для его определения есть функция `super`.
typeof(5) #=> Int64
super(Int64) #=> Signed
super(Signed) #=> Real
super(Real) #=> Number
super(Number) #=> Any
super(super(Signed)) #=> Number
super(Any) #=> Any
# Все эти типы, за исключением Int64, абстрактные.

# Для создания подтипа используется оператор <:
type Lion <: Cat # Lion — это подтип Cat
    mane_color
    roar::String
end

# У типа может быть несколько конструкторов.
# Для создания нового определите функцию с именем, как у типа,
# и вызовите имеющийся конструктор.
Lion(roar::String) = Lion("green", roar)
# Мы создали внешний (т.к. он находится вне определения типа) конструктор.

type Panther <: Cat # Panther — это тоже подтип Cat
    eye_color

    # Определим свой конструктор вместо конструктора по умолчанию
    Panther() = new("green")
end

# Использование внутренних конструкторов позволяет
# определять, как будут создаваться объекты типов.
# Но по возможности стоит пользоваться внешними конструкторами.

#####
## 6. Мультиметоды
#####

# Все именованные функции являются generic-функциями,
# т.е. все они состоят из разных методов.
# Каждый конструктор типа Lion — это метод generic-функции Lion.

# Приведём пример без использования конструкторов, создадим функцию теов

# Определения Lion, Panther и Tiger

```

```

function meow(animal::Lion)
    animal.roar # доступ к свойству типа через точку
end

function meow(animal::Panther)
    "grrrr"
end

function meow(animal::Tiger)
    "rawwwr"
end

# Проверка
meow(tigger) #=> "rawwr"
meow(Lion("brown", "ROAAR")) #=> "ROAAR"
meow(Panther()) #=> "grrrr"

# Вспомним иерархию типов
issubtype(Tiger, Cat) #=> false
issubtype(Lion, Cat) #=> true
issubtype(Panther, Cat) #=> true

# Определим функцию, принимающую на вход объекты типа Cat
function pet_cat(cat::Cat)
    println("The cat says $(meow(cat))")
end

pet_cat(Lion("42")) #=> выведет "The cat says 42"
try
    pet_cat(tigger) #=> ERROR: no method pet_cat(Tiger,)
catch e
    println(e)
end

# В объектно-ориентированных языках распространена одиночная диспетчеризация –
# подходящий метод выбирается на основе типа первого аргумента.
# В Julia все аргументы участвуют в выборе нужного метода.

# Чтобы понять разницу, определим функцию с несколькими аргументами.
function fight(t::Tiger, c::Cat)
    println("The $(t.coatcolor) tiger wins!")
end
#=> fight (generic function with 1 method)

fight(tigger, Panther()) #=> выведет The orange tiger wins!
fight(tigger, Lion("ROAR")) #=> выведет The orange tiger wins!

# Переопределим поведение функции, если Cat-объект является Lion-объектом
fight(t::Tiger, l::Lion) = println("The $(l.mane_color)-maned lion wins!")
#=> fight (generic function with 2 methods)

fight(tigger, Panther()) #=> выведет The orange tiger wins!
fight(tigger, Lion("ROAR")) #=> выведет The green-maned lion wins!

# Драться можно не только с тиграми!
fight(l::Lion, c::Cat) = println("The victorious cat says $(meow(c))")
#=> fight (generic function with 3 methods)

fight(Lion("balooa!"), Panther()) #=> выведет The victorious cat says grrr
try
    fight(Panther(), Lion("RAWR")) #=> ERROR: no method fight(Panther, Lion)
catch
end

# Вообще, пускай кошачьи могут первыми проявлять агрессию
fight(c::Cat, l::Lion) = println("The cat beats the Lion")
#=> Warning: New definition
#   fight(Cat, Lion) at none:1
# is ambiguous with
#   fight(Lion, Cat) at none:2.
# Make sure

```

```
# fight(Lion,Lion)
# is defined first.
#fight (generic function with 4 methods)

# Предупреждение говорит, что неясно, какой из методов вызывать:
fight(Lion("RAR"),Lion("brown","rarr")) #=> выведет The victorious cat says rarr
# Результат может оказаться разным в разных версиях Julia

fight(l::Lion,l2::Lion) = println("The lions come to a tie")
fight(Lion("RAR"),Lion("brown","rarr")) #=> выведет The Lions come to a tie
```

Что дальше?

Читать [документацию](#)! А помощи искать в списке рассылки.

Хабрапользователь @magik рекомендует посмотреть пару видео с конференций: «Julia: A Fast Dynamic Language for Technical Computing» и «Julia and Python: a dynamic duo for scientific computing».

Теги: julia, c, fortran, python, производительность

Хабы: Программирование, Julia

↑ +48 ↓ 187 👁 44,5k 💬 86 ➦ Поделиться



Сергей @skovorodkin

Пользователь

ПОХОЖИЕ ПУБЛИКАЦИИ

26 мая 2020 в 11:52

Python.org рекомендует: Программирование для НЕпрограммистов

↑ +19 👁 16,7k 📖 223 💬 3

2 ноября 2015 в 11:04

Python Meetup 25.09.2015: мониторинг производительности и использование BDD

↑ +9 👁 6,3k 📖 50 💬 0

1 июня 2013 в 21:38

Тестирование производительности Python 2.7 при обработке списков различными способами

↑ +11 👁 14,2k 📖 110 💬 15

ЗАКАЗЫ

Создать Stream сервис на Nimble с онлайн чатом и плеером

40 000 Р за проект • 2 отклика • 9 просмотров

Необходимо сделать парсер по Вконтакте (или доработать существующий)

30 000 Р за проект • 3 отклика • 30 просмотров

Разработать ПО для процессора STM32H7 (Cortex-M7)

1 000 Р за час • 1 отклик • 21 просмотр

Разработка модуля заявок для системы-маркетплейса (Python / Flask)

30 000 Р за проект • 3 отклика • 44 просмотра

Проверка работ студентов по курсу Python-разработчик

30 000 ₽ за проект • 14 откликов • 67 просмотров

Больше заказов на Хабр Фрилансе

Комментарии 86

**javascript** 3 февраля 2014 в 23:23

↑ +3 ↓

В чём преимущество перед Python, к примеру?

**Elsedar** 3 февраля 2014 в 23:58

↑ +3 ↓

Питонов, конечно, много разных, но в первую очередь производительностью.

А еще в статье, кажется, ни слова о мощнейшей встроенной математической библиотеки из коробки, а это одна из основных фиш.

**ffriend** 4 февраля 2014 в 03:08

↑ +29 ↓

Производительность Питона — это очень непростая тема. Чаще всего люди, которые говорят о производительности языка, меряют её на разного рода бенчмарках, что-то вроде вычисления числа Пи или рекурсивного обхода дерева. Однако, реальные приложения работают иначе. Возьмём три области, в которых Python имеет значительный авторитет: Linux-скрипты, веб и научные вычисления.

Для скриптов производительность практически не имеет значения. Скрипт либо отработывает за долю секунды (и тут гораздо важнее время разогрева), либо работает долго, но упирается по производительности в автоматизируемые операции (например, компиляция проекта на C++ или копирование файла). Соответственно, производительность языка скриптования тут особой роли не играет.

Другое дело — веб. Тысячи пользователей, миллионы просмотров страниц, а тут Питон со своим тормознутым интерпретатором. Однако, на деле производительность веб-приложения практически всегда упирается либо в сервер приложений (например, Apache2), либо в базу данных, либо в какие-нибудь ресурсы (сеть, диски и т.д.). В интерпретаторе языка веб-запрос проводит не так уж и много времени, так что фреймворки на «медленном Питоне» вполне себе неплохо масштабируются.

Ну и, наконец, научные вычисления, где код может выполняться и час, и день, и неделю. Оптимизация даже в пару раз способна значительно облегчить исследование (а иногда даже сама является достойным исследованием). Интерпретируемый язык в такой среде обречён на провал. Тем не менее, scientific stack в Python — один из самых мощных: SciPy stats, scikit-learn, image, sparse, Pandas, Theano — вот только некоторые из широкого набора инструментов. При этом все тяжёлые операции выполняются на стороне хорошо продуманных библиотек (а вернее, расширений), написанных на C. Python же только управляет логикой и даёт приятный интерфейс.

Не так давно я как раз писал модуль для машинного обучения и конкретно распознавания визуальных образов. ML — наука неточная и требующая экспериментов. А один прогон эксперимента даже на небольшом объёме данных длился, ни много ни мало, 7.5 часов. Естественно, мне захотелось оптимизировать. После профилирования оказалось, что 95% времени тратится на операциях свёртки (convolution). Не то, чтобы свёртка у меня работала медленно, но уж больно много раз она выполнялась. Я перепробовал несколько библиотек из научного набора Питона (все написаны на C), но результат оставался примерно тем же. Решение нашлось в функции filter2D из библиотеки OpenCV (с обёрткой в виде питоновского модуля cv2). Как оказалось, OpenCV использует специальные инструкции процессора для параллельной обработки данных, и благодаря этому, после пары часов колдовства, эксперимент стал отработывать ровно за 1.5 часа, т.е. в 5 раз быстрее. И всё это не покидая интерпретатора Питона. Если Julia сможет сделать операцию свёртки в 5 раз быстрее, чем C код, то да, безусловно я сразу переключусь на неё. Однако этого, по очевидным причинам, не произойдёт.

По большому счёту, производительность Python упирается в его циклы. Циклы в интерпретаторе — это всегда долго. Но в большинстве случаев циклы прекрасно векторизуются и перекладываются на массивы NumPy (не всем это нравится, для многих циклы всё же привычнее, но это уже дело вкуса). Да и PyPy понемногу допиливает свою реализацию NumPy с блекджеком и JIT-оптимизированными циклами.

В общем, не стоит сбрасывать Python со счетов просто потому что он «медленный». За этой толстенькой, на первый взгляд, змеей стоит вся сила нативных библиотек ;)

НЛО прилетело и опубликовало эту надпись здесь



ffriend 4 февраля 2014 в 12:10



Термин «векторизация» относится именно к математическим вычислениям, поскольку оптимизации применяются именно к операциям над векторами и матрицами. В более же общем случае подобные оптимизации называются пакетной и/или параллельной обработкой. Суть здесь одна и та же.

Например, представим, что нам нужно сложить поэлементно два вектора: a и b . Наивная реализация будет выглядеть примерно так:

```
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([4, 3, 2, 1])

c = np.zeros((len(a),))
for i in range(len(a)):
    c[i] = a[i] + b[i]
```

Вот как это выглядит с точки зрения системы:

1. Если индекс цикла меньше максимального*, то:
2. Найти адрес i -ого элемента массива a .
3. Найти адрес i -ого элемента массива b .
4. Получить и сложить элементы массивов.
5. Найти адрес нового элемента в массиве c и записать туда результат.
6. Вернуться на шаг 1.

* — на самом деле здесь, конечно же, итератор, а не просто цикл со сравнениями, но суть та же.

При этом все операции, по сути, выполняются в интерпретаторе Python, то есть проходят весь путь от байткода Питона до машинного кода, туда и обратно, туда и обратно. Особенно прискорбно, что приходится каждый раз вычислять адрес i -ого элемента. Ведь по сути, мы знаем, что следующий элемент находится ровно в N байтах от текущего (где N — длина типа данных) и можем просто инкрементировать адрес. Но вместо этого мы ныряем в Питоновский интерпретатор и проходим весь цикл поиска элемента по индексу: поиск имени переменной (hash!), поиск метода `__getitem__()` (hash again!) и медленный питоновский вызов найденного метода. И это с учётом того, что мы использовали наипсанную на C структуру `ndarray` вместо стандартного списка, в котором элементы — это вообще ссылки на отдельные объекты.

К счастью, NumPy предоставляет специальные функции для обработки векторов «целиком», а не поэлементно. Например, предыдущий цикл можно переписать так:

```
...
c = a + b
```

При этом оператор `+` является перегруженным и вызывает низкоуровневую операцию сложения векторов. А эта низкоуровневая операция уже может быть полностью реализована на C, с использованием смещений в массивах данных и быстрыми циклами C. Получается, что вектор обрабатывается на низком уровне одной «пачкой», вместо того, чтобы тратить кучу времени на вызов функции для каждого отдельного элемента.

Более того, низкоуровневая имплементация на C может использовать также и оптимизации на уровне процессора. Например, SIMD инструкции. Такие инструкции позволяют загружать сразу несколько элементов и применять к ним одну и ту же операцию в параллели. В случае сложения массивов это означает, что элементы складываются, например, не по одному, а по 4 сразу. А это и чистое ускорение в 4 раза, и уменьшение количества дорогих для процессора if-инструкций во столько же раз. Правда, я не буду утверждать, что все эти оптимизации используются во всех научных расширениях Питона, но по крайней мере реализовать это можно.

Так вот, это всё касалось векторизации, и почему она улучшает производительность. На самом деле всё то же самое можно перенести и на другие области. Например, большие массивы данных в распределённых системах часто обрабатываются пачками — данные буферизируются до определённого количества, а затем передаются одним пакетом, чтобы избежать накладных расходов по передаче каждого кусочка по отдельности. Аналогично тому, как векторизация позволяет избежать накладных расходов по вызову функций для каждого элемента. И аналогично SIMD инструкциям любую обработку можно распараллелить на несколько ядер, обрабатывая за один период времени сразу по несколько элементов данных.



sbos 4 февраля 2014 в 21:15



Здорово, что вы подняли эту тему. Чем мне особенно нравится Julia, так это тем, что в ней, как и в python или matlab можно реализовывать вычисления сразу в векторной форме. Это просто (с точки зрения написания кода), но не всегда

эффективно, если компилятор/библиотека линейной алгебры не умеет оптимизировать такие вычисления.

Пример:

```
r = a * k + b + c
```

(здесь a, b, c — массивы, a k — скаляр).

При наивной реализации компилятора/библиотеки этот код приведет к созданию 2-х временных массивов, в первый будет записан результат умножения, во второй результат сложения, ну и если особенно сильно не повезет, то добавится и третий временный массив. В настоящий момент, насколько мне известно, в `rpython/numpy` все обстоит именно таким образом. Накладные расходы на выделение памяти могут быть не столь заметны, если язык в целом довольно медленный, и мы счастливы просто от того, что может вызывать быстрые функции из BLAS/LAPACK, но после того, как все вычисления векторизованы до упора, а производительность надо поднять еще и еще, то нужно продолжать сбрасывать балласт.

И тут в `julia` есть замечательный механизм макросов и библиотека `Devvectorize`, которая позволяет мне написать:

`@devvectorize r = a .* k + b + c` (здесь `.*` — это поэлементное умножение, сейчас станет понятно, для чего требуется это уточнение), и для данного конкретного примера автоматически вместо данной строчки будет сгенерирован цикл:

```
for i in 1:length(a) r[i] = a[i] * k + b[i] + c[i] end
```

В данном коде не происходит создания временных объектов, а благодаря тому, что в `julia` циклы быстрые, почти такие же как в Си, то мы получаем еще немного (на самом деле очень много) пространства для оптимизации без нужды писать некрасивые циклы.



ffriend 5 февраля 2014 в 01:39



Согласен, штука прикольная, но всё-таки не killer feature. Если уж дошло до оптимизации выделения памяти, а доступные функции не принимают в качестве параметра выходной массив (как это сделано для многих методов `OpenCV`, где массивы действительно большие), то не грех и спуститься на уровень C.

Т.е., `Julia`, конечно, обладает отличным набором качеств, но ни одно из них не имеет достаточного веса, чтобы «сорвать куш» и вытеснить какой-то другой язык из его ниши. Например, `Python` всё больше вымещает `Matlab`, потому что 1) бесплатный и 2) хорошо интегрируется с промышленными системами. У `Julia` таких важных преимуществ перед другими языками я не вижу, поэтому пока что лично для меня это скорее интересный эксперимент, но не замена существующим инструментам.

Хотя посмотреть, чем этот эксперимент закончится, будет интересно. Сейчас `Julia` позиционируется как один язык, подходящий для всего. Последний раз больше чем на одну нишу (производительность + веб) претендовал `Go`, но в итоге ни одну, ни другую область он так и не завоевал.



ivlis 5 февраля 2014 в 01:58



У питона огромная проблема с параллельными вычислениями. `GIL` — вот это всё. Обычно физические задачи довольно легко разделяются, то есть надо вычислить $f(x)$ для набора x , и результат $f(x1)$ не зависит от $f(x0)$. На сях или фортране на помощь приходит `openmp` и можно ускорить программу в количество ядер на проце практически бесплатно. Конечно, на питоне есть модуль `multiprocessing`, но это страшный костыль. Мало того что его возможности весьма ограничены, например, если где-то внутри есть внешняя `lambda`, то работать ничего будет, да он ещё и каждый раз интерпретирует файл и начинает тормозить уже интерпретатор. Использовать `shared memory` только ручками можно. В `openMP` для этого надо просто пометить переменную как `shared`.

В общем я тут сделал проект на питоне и что-то у меня такое чувство, что я хочу вернуться к плюсам. Питон даёт неплохой старт, но когда дело доходит до реального счёта получается медленно.



ffriend 5 февраля 2014 в 12:15



Я бы сказал, тут проблема глубже. А вернее, тут две отдельные проблемы: `GIL` в `Python` и поддержка `OpenMP` в языках кроме C++.

Насколько я знаю `OpenMP` (а знаю я его плохо, так что не стесняйтесь поправлять), он ближе к расширению языка, чем к библиотеке. Например, тот же `NumPy` можно просто импортировать как любую другую библиотеку и работать с ним, не меняя общий workflow. `OpenMP` же предполагает изменение семантики (как без `#pragma` пометить переменную как `shared`?) и процесса сборки (`-fopenmp`). Лучшее, что может предложить в таких условиях динамический язык как `Python` — это вынести параллельную обработку в статическое расширение. А для этого уже есть `Cython`. Теоретически, конечно, можно ещё круче — встроить поддержку `OpenMP` в JIT-компилятор, но это уже совсем нетривиальная задача.

Что же касается `GIL`, то да, это насущная проблема. Тут либо менять парадигму параллелизма (`multiprocessing`, `message passing`, etc.), либо опускаться ниже (опять же, `Cython`). К счастью, ребята из `PyPy` активно работают над этим вопросом и

уже имеют несколько вариантов имплементации без GIL-а. Осталось дожидаться, пока на одном ядре такие имплементации будут работать не сильно медленнее, чем стандартная версия :)

 **ivlis** 5 февраля 2014 в 19:41

↑ 0 ↓

Да, openMP это расширение языка, но оно поддерживается как минимум gcc и llvm. С пришествием C++11 с `std::future` и `std::thread` возможно и без openmp достаточно просто писать параллельные задачи. К Cython я присматриваюсь, но как-то пока не осилил. Да и если работать с numpy, то все его массивы это по факту массивы C, их можно передать во внешнюю библиотеку без проблем.


С PyPy печаль, что за ними никто не стоит. Тот же openMP пилит Intel, за то время как я его использую (около 5 лет) они очень продвинулись. А с python3 и numpy/scipy у PyPy воз и ныне там.

Edit: openMP поддерживает как минимум C/C++ и Fortran.

 **ffriend** 5 февраля 2014 в 20:17


↑ 0 ↓

Py3k на PyPy как раз уже вполне юзабелен. По крайней мере мне не удалось найти несовместимостей. С NumPy проблема идеологическая — очень долго вообще было непонятно, как его писать (подключать ли расширения из CPython, или использовать мосты к C++, или допиливать ctypes), в конечном итоге решили переписать всё по своему и сейчас борются за прохождение юнит-тестов. В целом, NumPy — это для них одна из приоритетных задач. Но, как вы правильно заметили, за ними никто не стоит, а на одни донейты не сильно разгонишься.

 **ivlis** 5 февраля 2014 в 20:23

↑ 0 ↓

За ссылку на mogeturu спасибо — попробую его, а то у меня уже всё под ру3. К переписыванию from scratch с отношусь скептически, всё же scipy использует старые добрые фортрановские библиотеки, где уже вылизано всё как только можно, им можно доверять. А вот что там понапишут новое ещё не известно. Хотя если будет время постараюсь помочь проекту.

 **ffriend** 5 февраля 2014 в 22:52

↑ 0 ↓

Про переписывание с нуля я имел ввиду немного другое: насколько я знаю из последних событий, в PyPy в конце концов заколебались думать, как состыковать свою архитектуру с NumPy, написанной на C, и решили эту самую часть просто переписать. О переписывании библиотек, на которые опирается сам NumPy, конечно же, речи не идёт.

 **Eternalko** 4 февраля 2014 в 05:09

↑ +1 ↓

Когда последний раз когда я ее (julia) щупал, то эта «мощнейшая библиотека» была настолько базовой что аж грустно. Увы, гадкий R, с огромной тучей CRAN пакетов рулит :(

 **sbos** 4 февраля 2014 в 21:21

↑ 0 ↓

Учитывая то, сколько R лет, в этом нет совершенно ничего удивительного. Кстати, из julia легко вызывать код на Си или питоне

 **Eternalko** 4 февраля 2014 в 21:28

↑ +1 ↓

Проблема каждой новой технологии. В старую уже много заинвестировали и т.д. и т.п.

 **skovorodkin** 4 февраля 2014 в 00:37

↑ 0 ↓

Как уже сказал @Elsedar, производительность решает.

Вообще, разработчики в первую очередь ориентируются на потребности научного мира, а там, помимо производительности, нужна и простота — тяжело одновременно быть нейробиологом и знать премудрости C++ (конечно, ничего невозможного, но, чем ниже барьер входа, тем больше исследований смогут проводить учёные). Писать быстрый код для научных вычислений на Julia (по словам тех, кто этим занимается) выходит быстрее, чем на Python.

Если действительно интересуетесь, рекомендую почитать [обсуждение](#) к оригинальной статье на Hacker News.



potan 4 февраля 2014 в 10:59 # 📖 🔍 ↻

↑ +1 ↓

Мультиметоды полезнее, чем классический ООП. Производительность. По мне так и синтаксис гораздо более приятный.



sneer 4 февраля 2014 в 15:34 # 📖 🔍 ↻

↑ 0 ↓

могли бы со скалой сравнить?



potan 4 февраля 2014 в 18:46 # 📖 🔍 ↻

↑ 0 ↓

Со Scala все-таки сильно разные области применения. Скорее надо с R сравнивать. Или с Clojure (с библиотекой incanter).



sbos 4 февраля 2014 в 20:57 # 📖 🔍 ↻

↑ 0 ↓

Тем, что julia создавался с нуля исключительно для научных вычислений и активно оптимизируется исключительно под эту нишу, не пытаюсь при этом быть универсальным-самым-лучшим-в-мире-языком.



paulousky 3 февраля 2014 в 23:34 # 📖

↑ +43 ↓

Верю, что

была создана не гиками, а студентами точных наук из MIT

т.к. нормальные программеры не могли бы предложить использовать индексирование с единицы в массивах и строках



josser 3 февраля 2014 в 23:56 # 📖 🔍 ↻

↑ 0 ↓

А в чем удобство использовать нумерацию стартующую с нуля? Ну, допустим, так исторически сложилось. Но есть ли практическое применение этой особенности?

Говоря «практическое» я подразумеваю например алгоритм реализации которого основывается на этом свойстве массивов/строк.

На данный момент я таких случаев не встречал, а на уроках по программированию этот факт попросту затягивает обучение. Ученику приходится привыкать к новой нумерации, хотя может это и к лучшему (перестроить мозг на программмерский лад)



paulousky 4 февраля 2014 в 00:00 # 📖 🔍 ↻

↑ 0 ↓

Ну, допустим, так исторически сложилось.

Именно, так.



josser 4 февраля 2014 в 00:26 # 📖 🔍 ↻

↑ 0 ↓

Вы весь комментарий почитайте-то.



SLY_G 4 февраля 2014 в 00:07 # 📖 🔍 ↻

↑ +16 ↓

К примеру, если индекс массива — word, тогда мы можем с 0 до 255 пронумеровать элементы (00-FF). А если с 1 до 255, тогда один элемент теряем.

Нумерация блоков памяти с нуля идёт исторически. Видимо, потому что если нумерация с нуля, тогда текущий номер элемента — это смещение от начала. У нулевого смещение ноль.

Ну и как вы сами сказали — привыкание к такой нумерации настраивает на программирование, потому что во всех языках так и не надо путаться.



foxkeys 4 февраля 2014 в 00:21 # 📖 🔍 ↻

↑ 0 ↓

Именно, это техническая особенность адресации современных процессоров, которую, конечно, можно скрыть (на этапе компиляции) — но зачем?

Это создаст дополнительный слой преобразования, головную боль с переполнениями и отладкой (когда в исходниках одно —

а в ассемблерном режиме — совсем даже другое).
А оно надо?



jossier 4 февраля 2014 в 00:24



↑ -3 ↓

Для удобства человека а не машины. В этом похоже главная мысль.



StreetStrider 4 февраля 2014 в 14:43



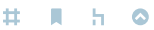
↑ +1 ↓

Продолжая об удобстве программиста, я всегда считал, что нумерация с единицы это здорово, потому что тогда можно использовать 0 в качестве индикатора отсутствия результата, который имеет тривиальную проверку в блоке `if`. Сейчас же нужно сравнивать с какой-то иной константой (например -1), что визуально загрязняет проверку.

С другой стороны в строготипизированных языках это «слишком нестрогое», и если есть типы-объединения, то можно пойти ещё дальше и возвращать какой-нибудь `Either<Index, Nothing>` там, где индекс может быть, а может не быть.



KvanTTT 4 февраля 2014 в 02:02



↑ +1 ↓

Ну и как вы сами сказали — привыкание к такой нумерации настраивает на программирование, потому что во всех языках так и не надо путаться.

Т.е. вы Pascal или Delphi языком программирования не считаете? :)



int19h 4 февраля 2014 в 05:30



↑ +1 ↓

Lua еще.



Halt 4 февраля 2014 в 06:48



↑ +1 ↓

и Smalltalk :)



Ogra 4 февраля 2014 в 12:46



↑ +1 ↓

И Erlang



magik 4 февраля 2014 в 18:02



↑ 0 ↓

Уверен, что авторы ориентируются во многом на Fortran (это к слову об «исторически сложилось») и Matlab, где индексы тоже начинаются с 1.



SLV_G 4 февраля 2014 в 12:15



↑ 0 ↓

Не во всех, есть исключения. Однако, насколько я помню, в паскале можно задать массив с любым начальным значением.



Alexeyslav 4 февраля 2014 в 13:07



↑ 0 ↓

вообще по хорошему счету, нумерация всегда идет с 1, но когда идет речь о программировании на реальном железе то нумерацию заменяют на адреса элементов в массиве, а вот адреса-то и начинаются с нуля! Именно для удобного совмещения абстракции и реального железа ввели нумерацию элементов с нуля. Впрочем, иногда и эта абстракция даёт сбой — например строки в классическом паскале — 0-м элементом идет длина строки, поэтому реальное содержимое строки начинается с 1-го элемента.



VolCh 4 февраля 2014 в 18:45



↑ -1 ↓

Просто не надо путать понятия «номер» и «индекс». У первого элемента индекс ноль.

foxkeys 4 февраля 2014 в 00:18

+5

Наверное вот: поэтому
Вот «оригинал»

Да еще потому, что в низкоуровневых языках (ассемблер) для адресации элементов используется указатель на начало массива, а номер элемента является смещением. Соответственно, адрес первого элемента это [указатель]+0, а не [указатель]+1



jossier 4 февраля 2014 в 00:23

+2

Я отвечаю вам в последнем комментарии:

Ну да, смещение, ассемблер, вот это все. В памяти все хранится а память нумеруется.

Но я в массивах также храню список комментариев, или, например, сообщения пользователей, и в меньшем количестве случаев, я храню там байты и биты. И нумерую я их: «первый комментарий», «второй». итд. В голове. А не в памяти компьютера.



foxkeys 4 февраля 2014 в 00:32

+3

Ну, если вы вообще имеете дело с индексированными массивами — то счет с нуля более «правильен» математически (не улетает в переполнение — в минус — на границах диапазона). Почитайте по ссылкам, там в общем-то довольно понятно. Хотя и на английском. ~~Да, да тоже терпеть не могу математику — а куда деваться...~~

Ну а если вы работаете на высоком уровне — то это скорее «старость» языка. Современные языки для перебора коллекций используют итераторы и индексами там и не пахнет. Никакими...

```
foreach($cmmments as $comment){ }
```

Так что, возвращаясь к исходному вопросу — для «старых» языков удобство в том, что при счете с нуля вы наделаете меньше ошибок :-)

А для новых — индексы и вовсе не очень-то нужны, да и сделать можно любые, какие хочется (привет итераторам)



jossier 4 февраля 2014 в 00:33

+3

Прошел по ссылке, и мой интерес удовлетворен. Пожалуй теперь я не согласен с разработчиками Julia :D



vmarunin 4 февраля 2014 в 00:37

+4

В остатке от деления. Он от 0 до n-1

В том, что эту единичку чаще не нужно прибавлять/убавлять при реализации некоторых сложных структур, тот же heap в виде массива.

Отсчёты бывают не только от какого-то указателя, но и от времени.

PS Ряд Тейлора принципиально важно стартует с 0. Комбинаторика доопределила 0!=1 тоже не просто так



Flatformer 4 февраля 2014 в 02:16

0

Могу ошибаться сонным мозгом, но вроде тогда так не сработает (пример на js)

```
var a = []; // представим, что он заполнен
i = a.length;
while (i--) console.log(a[i]);
```

На элементе с индексом 0 будет undefined. Не спрашивайте меня зачем такой код бы мог понадобиться :)

В любом случае, перестроиться будет уже очень непросто.



Flatformer 4 февраля 2014 в 02:22

0

Заполнен, начиная с 1*



VolCh 4 февраля 2014 в 18:44

0

Индексная арифметика. Прибавляем к указателю на массив байтов индекс нужного байта элемента и получаем адрес байта.

**josser** 4 февраля 2014 в 18:45

↑ 0 ↓

Вы как будто не читали вопроса и весь тред

Я знаю как это работает внутри. Вопрос был не в этом.

**VolCh** 4 февраля 2014 в 18:52

↑ 0 ↓

Так это и есть практическое применение особенности — адрес первого элемента и адрес массива совпадают, значения `array`, `array + 0` и `array[0]` указывают на одну область памяти.

Просто не нужно путать индекс элемента и его номер.

**josser** 4 февраля 2014 в 18:52

↑ 0 ↓

Про это я тоже писал в треде)

**Swiftarrow7** 30 сентября 2018 в 00:16

↑ 0 ↓

Может из-за двоичной системы счисления? Тип мы не можем добиться 256 вариантов считая с 1? Максимум 255, а как бы терять один вариант думаю не очень логично.

**skovorodkin** 4 февраля 2014 в 00:09

↑ +8 ↓

Предоставлю слово Джеффу Безансону, одному из разработчиков Julia:

There is a [huge discussion](#) about this on the mailing list; please see that. If 0 is mathematically «better», then why does the field of mathematics itself start indexes at 1? We have chosen 1 to be more similar to existing math software.

It really isn't that interesting a topic though. We all work with both 0-based and 1-based languages and I don't think it matters that much.

**vmarunin** 4 февраля 2014 в 00:49

↑ +7 ↓

В математике степенные ряды сплошь и рядом стартуют с 0-ой степени и нулевого индекса. Ну или с первого индекса и потом уродливую единичку везде прибавляют
Ряд Тейлора, определение факториала ($0! = 1$), ряд Фурье вполне себе с нулевого индекса определяются.

**Ogra** 4 февраля 2014 в 15:21

↑ 0 ↓

Отсюда вывод — начало индексации должно быть произвольным. С единицы — для массивов и строк, с нуля — для рядов Фурье и Тейлора.

**Bal** 4 февраля 2014 в 00:34

↑ +7 ↓

Это не «нормальные программисты», а формировавшиеся после набора популярности Си, фактически — в 1990-х и позднее. А раньше, до массовой «сификации», это был даже не холивар, а просто рабочий момент, который каждый программист сразу узнавал при изучении нового языка. Мне в то время даже в голову не приходило спорить, с 0 или 1 должен начинаться индекс. Как в данном языке принято — так и надо использовать :)

**Eternalko** 4 февраля 2014 в 05:12

↑ +2 ↓

Julia она для математиков и статистиков. По образу и подобию R.

В математике нет массивов, а есть вектора. И нет «нулевого» компонента вектора. Есть только первый.

Все.

**potan** 4 февраля 2014 в 11:07

↑ +1 ↓

Это скорее не математические, а общекультурные заморочки. Если бы математики и простые люди были бы сразу приучены, что натуральные числа начинаются с нуля, многое было бы проще.



tenshi 4 февраля 2014 в 11:34



↑ +2 ↓

Нумерация с 0 и в математике не менее популярна. И она гораздо более элегантна.

ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%82%D1%83%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%87%D0%B

Кроме того, при использовании низкоуровневых языков llvm или asm внутри julia нумерация ведь будет с 0. Это может привести к шизофрении)

> Индексирование не всегда правильно работает для UTF8-строк,
21 век на дворе... а до сих пор изобретают языки без поддержки юникода, перегрузки операторов и многозадачности



SDSWanderer 3 февраля 2014 в 23:43



↑ -1 ↓

> Я обычно работаю таким образом: сначала пишу прототип на одном языке, после переписываю критические секции на другом языке, а если мне нужно, чтобы программа летала, я использую третий язык.

Учитывая что в практически всех проектах над которыми я работал все упиралось в производительность не языка а БД, и переписывать никто ни на что не собирается, этот язык наверно не для меня.



foxkeys 3 февраля 2014 в 23:59



↑ 0 ↓

Вообще говоря, ассемблерные вставки в не самом новом языке Паскаль (во всех средах, Turbo Pascal, Delphi, Lazarus) никто вроде не отменял. Равно как и возможность дебажить / просматривать в нативных средах разработки ассемблерный код. Так что с новизной тут как-то не очень...

НЛО прилетело и опубликовало эту надпись здесь

НЛО прилетело и опубликовало эту надпись здесь



skovorodkin 4 февраля 2014 в 00:14



↑ 0 ↓

Спасибо, поправил.



OnYourLips 4 февраля 2014 в 01:22



↑ +2 ↓

мне важна только его работоспособность и производительность

Печально это. Для большинства все же важна стоимость (во времени работы) поддержки кода (а именно добавление нового функционала и правка старого).



Kirhgoff 4 февраля 2014 в 01:45



↑ +1 ↓

Не всегда же. Если я пишу прототип, чтобы проверить какую-то идею, можно и на выброс код написать. Как я понимаю, язык рассчитан на ученых, а им обычно бывает просто необходимо какую-то штуку посчитать и уже использовать полученные данные. С другой стороны, иногда этого качества очень не хватает, когда этот кусок кода в production систему надо встраивать.



RPG 4 февраля 2014 в 10:24



↑ 0 ↓

Язык очень похож на LuaIT, с его FFI возможен довольно простой биндинг с Си. Не пробовали сравнить?

А сила Питона всё-таки в готовых библиотеках. Как в Julia обстоят дела с Qt? А GTK+? Тут у Питона неоспоримая ниша.



potan 4 февраля 2014 в 11:09



↑ 0 ↓

Такой язык хочется задействовать для роботов и компьютерного зрения.

Есть ли там биндинги к OpenCV?

И какой-нибудь аналог CRAN (CPAN, Cabal...)?

**Ogra** 4 февраля 2014 в 12:48

↑ +2 ↓

По-моему, отличный язык для изучения ассемблера ;)

**Paха** 4 февраля 2014 в 14:38

↑ 0 ↓

Меня оттолкнуло что массивы начинаются с 1 а не с 0, нету ООП (или я не нашел). Но для некоторых задач думаю может быть удобным

**potan** 4 февраля 2014 в 18:52

↑ +1 ↓

Есть мультиметоды, которые перекрывают возможности ООП.

**Andrew_Tailor** 4 февраля 2014 в 16:41

↑ 0 ↓

Пока читал статью, думал, ну вот, ещё один язык программирования... Зачем? Но, когда стал смотреть описание языка, заинтересовался. Что-то в нём есть, действительно, изящное. Мне, как выпускнику «примата», очень нра. Возможность задать рациональную константу $2/3$ впечатляет и притягивает. Скажите, какие есть ещё языки программирования, позволяющие делать такие вещи?

**alhimik45** 4 февраля 2014 в 16:47

↑ 0 ↓

Common Lisp, clojure... А вообще много их en.wikipedia.org/wiki/Rational_data_type

**burjui** 5 февраля 2014 в 00:01

↑ 0 ↓

Scheme

**KvanTTT** 5 февраля 2014 в 02:25

↑ +1 ↓

А интересно, можно ли в каких-то языках работать еще и с иррациональными числами? Ну т.е. $\sqrt{2} + 1$ будет представляться именно иррациональным комплексным типом (в виде дерева). И если возвести это значение в квадрат, то получить $2 * \sqrt{2} + 3$?

**potan** 5 февраля 2014 в 11:15

↑ 0 ↓

В системах символьной математики, которые все в себе несут язык программирования более-менее общего назначения. Но можно и для обычных языков библиотеку сделать.

**chersanya** 5 февраля 2014 в 12:54

↑ 0 ↓

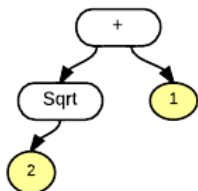
Что вы имеете в виду под «иррациональным комплексным типом (в виде дерева)»? В haskell, например, можно достаточно просто реализовать тип действительных чисел по определению — в виде бесконечного потока цифр. Правда, для таким образом представленных чисел неразрешимо сравнение, например (т.е. нельзя сделать всегда завершающийся оператор «равно»). Это не проблема реализации, а фундаментальное ограничение.

**KvanTTT** 5 февраля 2014 в 14:41

↑ 0 ↓

Что вы имеете в виду под «иррациональным комплексным типом (в виде дерева)»?

Я имею в виду, $\sqrt{2} + 1$ будет представляться как:



Данный вопрос у меня возник в связи с тем, что я почти дописал статью и проект по обработке математических выражений в .NET, в котором есть такой тип данных (правда там еще и с неизвестными). И поэтому интересуюсь данной тематикой.



ffriend 5 февраля 2014 в 15:26



↑ 0 ↓

```
(quote (+ (sqrt 2) 1))
```

Свёртку математических операций (например, чтобы "(+ 1 2)" сворачивалось просто в «3») можно добавить буквально одной рекурсивной функцией. В последних главах SICP-а подробно описывается, как работать (вычислять-применять) с таким абстрактным синтаксическим деревом. Даже если вы не пользуетесь Lisp-ом, то можно довольно быстро написать простенький интерпретатор в любом удобном для вас языке, в т.ч. для .Net.



ForNeVeR 10 февраля 2014 в 03:46



↑ +1 ↓

Посмотрите в сторону Mathnet Palladium и других библиотек того же автора. Я начинал тоже что-то такое делать, но оно не production-ready и вообще лишь учебный концепт.



potan 29 октября 2014 в 14:54



↑ 0 ↓

На самом деле все еще хуже. Нельзя отличить последовательность 1(0) и 0(9). То есть возможны алгоритмы, не только сравнения, которые в некоторых случаях не смогут выдать очередную цифру. Интересно представление чисел в виде цепных дробей. Но и там будут проблемы вблизи рациональных чисел.



chersanya 29 октября 2014 в 15:39



↑ 0 ↓

Можете развить мысль? Я не совсем понял, как это нельзя отличить последовательности 1.(0) от 0.(9). Если использовать представление чисел в виде бесконечных списков цифр, как я писал выше, то как раз эти последовательности отлично различатся уже по первой цифре. Другое дело, что получив на вход 0.(9) мы никогда не сможем сказать, действительно ли это 0.(9), или где-то дальше будет не девятка (следовательно, не сможем сказать, равно ли это число единице).



potan 29 октября 2014 в 16:21



↑ 0 ↓

Например, мы вычитаем из известной единицы «1.0» неизвестный ноль «0.(0)». Мы не можем выдать первую цифру «1» не убедившись, что вычитаемое строго равно нулю. То есть вычисление очередной цифры требует бесконечного сравнения. Кстати, у р-адических чисел такой проблемы нет. Хотя сравнение может никогда не завершиться, вычисление очередной цифры для «нормальных» функций происходит за конечное время.



chersanya 29 октября 2014 в 20:32



↑ 0 ↓

Вообще теперь я вас понял, но в данном конкретном примере всё-таки получится нормально вычислить :) Мы же можем выдать сразу 0, и далее все девятки. Если вычитаемое — ноль, то получится число 0.(9), которое равно единице. Если же какая-то цифра вычитаемого не ноль, то и в ответе не 9 будет. А вот в случае 0.(9) + 0.(0) как раз возникает описанная вами проблема.



kmike 4 февраля 2014 в 16:43



↑ 0 ↓

А кто-нибудь знает, почему у in словарь вторым аргументом, а у haskey и get — первым?



leventov 5 февраля 2014 в 01:45



↑ 0 ↓

Предположу, потому что ближе к английской фразе key in dict.






KvanTTT 5 февраля 2014 в 02:44



↑ +2 ↓






И название какое: Julia. Тоже в честь какой-то программистки назван, по аналогии с языком «Ada»? :)

А вообще да: Юля красивая.

**Rome** 7 февраля 2014 в 03:45  

↑ +1 ↓

Вопрос по существу. Как запустить файл с расширением jl
На винде jl запускается без проблем в cmd я делаю путь на julia и второй строкой путь к моей программе на jl Программа выполняется в командой строке без проблем. Но ничего не выходит на маке. В результате в терминале запускается только julia, программа не выполняется. Подозреваю, что причина в сыроватости.

**Rome** 7 февраля 2014 в 04:09    

↑ +1 ↓

Ок, сам спросил, нашел ответ, сам ответил. В терминале путь к /Applications/Julia-0.3.0-prerelease-d2d9bd93ed.app/Contents/Resources/julia/bin/julia-readline далее путь к программе /Users/Documents/firsl_julia.jl
На винде я тоже прописывал путь к julia-readline

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

- Сутки
- Неделя
- Месяц

Тест старения светодиодных ламп

↑ +126 👁 31k 📖 49 💬 149

Хватит натягивать сову на глобус

↑ +31 👁 18,1k 📖 35 💬 93

Тёмная сторона работы в Яндекс.Маркете

↑ +147 👁 17,9k 📖 52 💬 122

Советы по выбору усилителя сигнала сотовой связи 2G/3G/4G/5G

↑ +24 👁 13,8k 📖 86 💬 21

IT-ассоциации предупредили власти РФ об опасности массового отъезда программистов за рубеж

↑ +24 👁 13k 📖 10 💬 72

Ваш аккаунт	Разделы	Информация	Услуги
Войти	Публикации	Устройство сайта	Реклама
Регистрация	Новости	Для авторов	Тарифы
	Хабы	Для компаний	Контент
	Компании	Документы	Семинары
	Пользователи	Соглашение	Мегапроекты
	Песочница	Конфиденциальность	

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.