Язык программирования Julia

Unkn0wn

Julia — молодой язык программирования, который предназначен преимущественно для научных вычислений. Его разработчики хотели, чтобы он занял нишу, которую раньше занимали Matlab, его клоны и R.

Разработчики пытались решить так называемую проблему 2 языков: совместить удобство R и Python и производительность C. Давайте посмотрим, получилось ли у них это.

Содержание

История языка программирования Julia

Создают и разрабатывают язык программирования Julia в Массачусетском технологическом институте с 2009 года, а в середине 2012 года была выпущена публичная версия. Бета-версия имела много проблем и интересовала только любопытных энтузиастов и разработчиков-альтруистов, которые старались приспособить окружения под постоянно меняющиеся стандарты.

РЕКОМЕНДУЕМ:
Язык программирования Ада

Вот что пишут создатели языка программирования Julia.

> Мы хотим язык программирования с открытым исходным кодом, с либеральной лицензией. Мы хотим скорость C с динамизмом Ruby. Нам нужен гомоиконичный язык с настоящими макросами, как Lisp, но с очевидными, знакомыми математическими обозначениями, такими как в Matlab. Мы хотим что-то такое же удобное для общего программирования, как Python, такое же простое для статистики, как R, такое же естественное для обработки строк, как Perl, такое же мощное для линейной алгебры, как Matlab, и способное склеивать программы вместе как оболочку. Нечто простое в освоении, но при этом радующее самых серьезных хакеров. Мы хотим высокой интерактивности и эффективной компиляции. Мы ведь не слишком многого просим, верно?

В начале августа 2018 года вышла версия 1.0.0, что породило большой интерес к языку. Язык начали преподавать в университетах США, появились онлайновые курсы (на Cursera и Julia Academy), стартовали коммерческие и исследовательские проекты, разработчики начали проявлять интерес, а владение этим языком вошло в топ самых востребованных профессиональных навыков по версии Upwork. Стабилизация синтаксиса, в свою очередь, вызвала взрывной рост разработки вспомогательных пакетов.

Характеристики языка программирования Julia

Скорость: этот язык разрабатывался для высокой производительности. Программы, написанные на Julia, компилируются в эффективный нативный код для разных платформ через LLVM.

Общность: используя парадигму множественной диспетчеризации, Julia облегчает выражение многих объектно ориентированных и функциональных шаблонов программирования. Стандартная библиотека обеспечивает асинхронный ввод-вывод, управление процессами, ведение журнала, профилирование, менеджер пакетов и многое другое.

Динамичность: Julia поддерживает динамическую типизацию, ведет себя как язык сценариев и имеет хорошую поддержку для интерактивного использования.

Технологичность: Julia превосходна в высокотребовательных вычислениях, имеет синтаксис, который отлично подходит для математики, поддерживает множество числовых типов данных и параллелизм «из коробки». Мультиметод этого языка идеально подходит для определения числовых и массивоподобных типов данных.

Опциональность: она имеет богатый язык описательных типов данных, и объявления типов могут использоваться для уточнения и укрепления программ.

Адаптивность: встроенная система управления пакетами, которые хорошо работают вместе. Матрицы единичных величин или таблицы данных, названий, чисел и изображений — все можно обработать с одинаково хорошей производительностью.

Синтаксис

Джулия выглядит лучше, чем Matlab, но на достижение это не тянет. Среди основных языков программирования закостенелость и плохой дизайн Matlab уступают только PHP. В Octave и Scilab некоторые из этих проблем исправили, но там есть и свои. Matlab и R берут тем, что у них огромные, собранные за годы библиотеки наборов инструментов и функций, с которыми легко решать научные и вычислительные задачи.

В арсенале языка программирования Джулия этого пока нет: потребуются годы, чтобы создать собственную настолько же полную библиотеку пакетов.

Как и в перечисленных языках (Python, Matlab и прочие во главе с легендарным Fortran), в Julia есть срезы (`Arr[:,1]`), арифметические операторы распространяются на массивы, а **индексация начинается с единицы**.

Но вернемся к синтаксису. Для начала создадим пару массивов-векторов:

```
julia> A = [2, 3, 4]
3-element Array{Int64,1}:
 2
 3
 4
julia> B = [6+2im, 0.8, -1]
3-element Array{Complex{Float64},1}:
  6.0 + 2.0im
  0.8 + 0.0im
 -1.0 + 0.0im
```

Обратите внимание, что можно использовать разные числовые типы: тип подгоняется под высший в иерархии. Теперь создадим массив-строку и выполним простые операции:

```
julia> C = [1 0 1]
1?3 Array{Int64,2}:
 1  0  1

julia> A+B
3-element Array{Complex{Float64},1}:
 8.0 + 2.0im
 3.8 + 0.0im
```

```
 3.0 + 0.0im

julia> C * ans.^2
1-element Array{Complex{Float64},1}:
 69.0 + 32.0im
```

Результат последних вычислений хранится в переменной `ans`, в нашем случае — массив, который получился при сложении `A` и `B`. Мы его поэлементно возвели в квадрат и скалярно умножили на `C`.

Давайте посмотрим, как можно задавать функции.

```
function cube(x)
    x^3
end

cube(x) = x^3
```

Функции возвращают результат последних вычислений, то есть `return` прописывать не обязательно. Есть возможность использовать анонимные функции (похоже на лямбды из Python). Например, посчитаем евклидову норму вектора:

```
sqrt( sum(x-> x^2, Arr) )
```

Можно записать то же самое цепочкой функций (ведь это унарные операторы):

```
Arr.^2 |> sum |> sqrt
```

Нужные функции, как правило, уже есть в стандартной библиотеке. Для следующих двух примеров есть аналог — `hypot(Arr)`. Со строками работать так же просто, как с массивами. Выполним конкатенацию и запросто реализуем шифр Цезаря:

```
julia> "russian" * ' ' * "hacker"
"russian hacker"

julia> caesar(X, n) = prod( [x+n for x in X] )
caesar (generic function with 1 method)

julia> caesar("hakep.ru", 5)
"mfpju3wz"
```

Поскольку задать функцию и множество для создания массива можно несколькими способами, есть и разные стили написания. Тру стори: думаете, что за год хорошо разобрались в языке, раз задаете массивы с помощью включений (`[2*i for i = 1:10]`), а потом однажды в чужом коде видите `2*[1:10;]` и ничего не понимаете. Так что неплохо поинтересоваться и итераторами тоже.

Скорость

Скорость подкупает. Но если вы возьмете и напишете скрипт, выводящий энный член множества Фибоначчи, окажется, что вычисления идут медленно, не быстрее, чем на Python.

Скорость Julia достигается за счет множественной диспетчеризации с последующей JIT-компиляцией.

РЕКОМЕНДУЕМ:
Программирование в консоли

Преимущество модели метода Julia в том, что она хорошо сочетается с диспетчеризацией по нескольким типам. При создании функции вида `f(a,b)`, в зависимости от операций, которые используются внутри, будут определены методы для различных случаев. Скажем, `f(a::Int, b::Int)`, `f(a::Float, b::Int)`, `f(a::String, b::Int)` — для каждого будет скомпилирован высокоэффективный код, во многом идентичный тому, что дает C или Fortran. И вы можете посмотреть этот код с помощью простых команд:

```
function f(a,b)
    return 2a+b
end

@code_native f(2.0,3.0)

pushq   %rbp
movq    %rsp, %rbp
Source line: 2
vaddsd  %xmm0, %xmm0, %xmm0
vaddsd  %xmm1, %xmm0, %xmm0
popq    %rbp
retq
nop

@code_native f(2,3)

pushq   %rbp
movq    %rsp, %rbp
Source line: 2
leaq    (%rdx,%rcx,2), %rax
popq    %rbp
retq
nopw    (%rax,%rax)
```

Эта избыточность замедляет загрузку интерпретатора, пакетов и первый запуск вашего скрипта. А так как интерпретатор имеет глобальную область видимости и не допускает специфику типов, то будет генерироваться громоздкий низкоуровневый код, который старается предусмотреть любую нестабильность типов.

Это решается легко: оборачивайте свои операции в функции, избегайте глобальных переменных и по возможности выстраивайте логику программы, ориентируясь на конкретную задачу. Типы можно указывать и явно — как в статически типизированных языках. Кстати, мой пример шифратора Цезаря может принимать и строки, и массивы целых и комплексных чисел. Как думаете, какой длины будет портянка LLVM, сгенерированная для этой функции?

Параллельные вычисления

Как и остальные молодые языки вроде Go, Julia просто обязана быть ориентированной на многопроцессорные операции «из коробки». В наше время большие объемы вычислений выполняются в облачных средах, где по требованию выделяются нужные ресурсы. Стандарт MPI, который используют для организации работы крупномасштабных параллельных приложений, не отличается особой эластичностью. В ходе вычислений нельзя добавить процессоры, и возможности восстановления после сбоев весьма ограниченны.

В Julia передача сообщений отличается от MPI. Вместо модели «запрос — ответ» используется глобально распределенное адресное пространство, что позволяет легко оперировать ссылками на объекты на разных машинах и на ходу добавлять другие ресурсы.

А еще это чертовски просто реализовать! Вызываем пакеты, которые позволят осуществлять распределенные вычисления и использовать общие для всех процессов массивы. Указываем, что нужно посчитать параллельно.

```
using Distributed, SharedArrays
addprocs(2)

a = SharedArray{Float64}(10)
@distributed for i = 1:10
    a[i] = i
end
```

Или вычислим число пи одним из самых непрактичных способов на двух ядрах:

```
using Distributed, Statistics
addprocs(2)

@everywhere function PI(n)
    p = 0
    for i = 1:n
       x = rand(); y = rand()
       p += (x^2 + y^2) <= 1
    end
    4p/n
end
find_PI(N) = mean( pmap( PI, [N/nworkers() for i = 1:nworkers()] ) )

julia> @time fPI(1_000_000_000)
  2.674318 seconds (181 allocations: 7.453 KiB)
3.1416085999999996
```

В Julia используются такие техники, как Atomic Operations, Channels и Coroutines, а также MPI и CUDA.

Недавно была анонсирована новая способность Julia — Composable multi-threaded parallelism. Так что, если вам нужно обработать 150 Тбайт астрономических данных, повысить эффективность своей криптофермы, произвести климатическое моделирование, вы знаете, какой инструмент выбрать.

Дополнительные преимущества

Документация одна из самых удобных и полных среди тех, что я встречал. К тому же вы всегда можете вызвать справку по любой функции прямо в консоли набора кода, например: `?prod`, `?sqrt`, `?*`, `??`.

Набор и использование в коде символов Юникода. `\lambda + <Tab>` превратится в λ — приятная мелочь, хоть и не все консоли и браузеры ее отобразят.

Встроенный пакетный менеджер — парой команд можно скачать и настроить пакет, если он существует. Несмотря на увеличение числа пакетов, часто узкоцелевые пакеты заброшены или забагованы. В крайнем случае вы всегда можете написать свой.

Макросы и другие возможности метапрограммирования. Это ускоряет оборачивание библиотек на других языках. Хороший пример метапрограммирования — в пакете StructArrays.jl, где очень элегантно решена проблема SoA/AoS.

Функции языка C можно вызывать напрямую, функции Python — при помощи PyCall. С другими популярными языками дружба тоже довольно тесная.

Где запускать Julia

REPL — консоль-интерпретатор. Качаем с официального сайта и получаем все возможности Julia из коробки.

Jupyter — удобный инструмент популярный у дата-сайентистов.

Juno — IDE с обозревателем переменных и всеми недостатками Atom.

Дебагер — то, чего так долго не хватало джулиистам-первопроходцам.

Cassette — возможность модифицировать компилятор.

Плагины и редакторы:

Visual Studio Code;

JetBrains;

Vim;

Sublime Text;

Revise.

Насущный вопрос

Может быть, вы уже задались вопросом, почему Julia не поддерживает объектно ориентированное программирование. Но это не так. Julia — мультипарадигмальный язык, который поддерживает методы из процедурного, декларативного, функционального, мета и — внимание! — объектно ориентированного программирования.

Здесь все является объектом, есть наследование, абстрактные классы, полиморфизм, инкапсуляция. Точнее, в Julia все есть множественная диспетчеризация (multiple dispatch). Вместо завязки на экземпляре объекта `instance.method(…)` будет метод, заточенный под объекты: `method(instance, …)`.

РЕКОМЕНДУЕМ:
Как новичку определиться с языком программирования

Недостаток в том, что типы и методы по умолчанию не связаны между собой: можно получить код, где для какого-то типа нет методов либо они есть, но конфликтуют. Практически, идиоматичная Julia просто объединяет все методы с одним и тем же именем.

```
julia> size
size (generic function with 89 methods)
```

В результате у Julia по дизайну все полиморфно. Так что Julia объектно ориентирована до глубины души. Но это неочевидно из-за размытия границы между функциональным программированием и объектно ориентированным.

Создатели Julia были сосредоточены на разработке языка для поддержки математического программирования. Использование ООП в стиле C++ и Java, когда первый оператор владеет функцией, было неудобно в плане реализации и организации кода. Методы класса не очень-то полезны при работе с векторами, матрицами и тензорами, и наличие первого аргумента в выражении может привести к путанице.

Еще одна вещь, которую люди часто замечают, — поля данных не могут быть унаследованы, хотя не обязательно считать это неотъемлемой частью ООП, это просто свойство объектно ориентированных языков.

Множественная диспетчеризация — элегантное решение, которое дает программисту большую часть преимуществ ООП и в то же время не конфликтует с тем, как в нем работает математика.

Python или Julia

Из-за пологой кривой обучения Julia почти идеальна в качестве первого языка программирования, с которого можно переучиться на Python или углубиться в C. Я слышал, как преподаватели сетуют на то, что Python калечит в студентах программистов и их потом трудно переучивать на что-то более близкое к железу.

Однако Julia не может делать все, что делает Python, хотя она применима для большинства практических целей. И, как любой язык программирования, она имеет свои подводные камни. Вы можете использовать метапрограммирование и макросы, передавать их символьно в функции и из функций, но действительно передавать сами макросы нельзя. Функция `eval` всегда работает в области уровня модуля.

В Python метаклассы и проверки доступны везде, поскольку `eval` работает в локальной области.

Типы в Julia являются объектами, и вы можете передавать конструкторы типов сколько угодно, но типы могут быть определены только на верхнем уровне модуля, и их нельзя изменить после объявления. Это важное отличие от динамических языков вроде Python, Ruby, JavaScript и Smalltalk. Хотя типы в Julia могут быть сгенерированы с помощью `eval`.

В Python новые классы можно создавать внутри функций, которые существуют только в этой локальной области, и атрибуты класса можно динамически исправлять во время выполнения.

Разработчики Julia добились впечатляющих успехов, но у их языка еще нет той опоры, которая обеспечивает долгосрочный успех. Даже сочетание удобства и скорости Julia не заставит поклонников Python ему изменить (но это не точно). Python уже имеет связь с относительно низкоуровневыми операциями в виде Cython и даже упаковку для LLVM — Numba.

Для data science, machine learning, аналитики, научных вычислений и исследований Julia чертовски хороша. На ней удобно делать серверный бэкенд и прочие насущные сервисы, рассчитанные на долгую работу. Игры на ней писать можно, но с оглядкой на прожорливые вычисления и другие нагрузки. А вот кросс-платформенные приложения на Julia без плясок с бубном не получатся.

Я не знаю, выстрелит Julia или нет, но изучать и применять ее можно уже прямо сейчас. Для обучения на английском есть много ресурсов. На русском же из книг только устаревшая «Осваиваем язык Julia» Малкольма Шеррингтона. И самоучитель на GitHub.

Вам также может быть интересно

StateFlow в Kotlin

StateFlow, End of LiveData? — статья о StateFlow, новом классе библиотеки короутин Kotlin (начиная

Основы функционального программирования на Kotlin

Functional Programming in Kotlin — серия статей о функциональном программировании на Kotlin. В функциональном

Перегрузка операторов в Kotlin

Code expressivity++ with operator overloading — статья о перегрузке операторов на примере, как бы

Введение в Kotlin Flow

Into the Flow: Kotlin cold streams primer — одна из лучших статей о новой

Добавить комментарий

Имя *

Email *

Комментарий

Отправить комментарий