JNumPy

使用Julia为Python/NumPy编写扩展模块

宋家豪

2022.12.6

苏州同元软控信息技术有限公司

本章要点

1、简介

2、数据转换的规则

3、示例以及性能

4、未来的工作

Part 01

简介

1 简介

怎么使用Python进行高性能的科学计算：

- 核心的计算函数使用C或者Fortran来编写
- 使用numba等jit库编译原生的Python代码
- 核心的计算函数使用Julia来编写

一个例子：迭代法求解线性系统

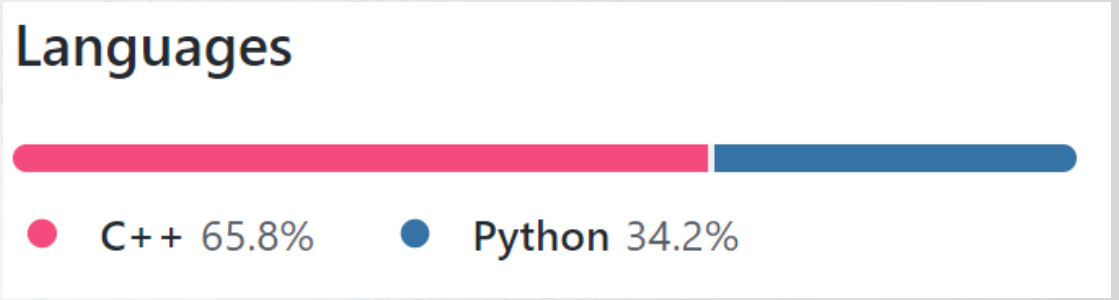
```
# 求解泊松方程
# Δu = f(x)
# 在离散的格式下我们得到一组线性方程组
# Au = b
x = np.linspace(0, 1, 1000)[1:-1]
b = np.sin(2 * np.pi * x)

# 线性算子(matrix or matrix free)
A = ...
# solve u = A^{-1}b with gmres
gmres(A, rhs)
```

1 简介

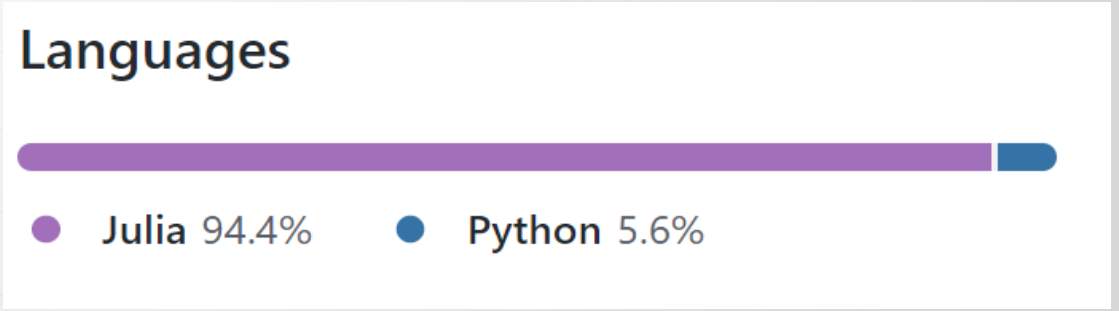
- Python的科学计算生态几乎总是使用这样的组合方案，即核心的计算部分使用C/C++编写，再使用Python编写一个易于使用的前端。

- JNumPy为Python侧的开发者提供这样一种方案，使用Julia实现核心的算法，并封装成Python库进行分发。

kmeans1d



julia-kmeans1d



特性

- 明确的类型标注减少数据转换时的额外开销
- 通过Python的包管理机制进行分发和版本管理
- Julia相关的依赖直接通过Project.toml来的管理
- 较为严格的数据转换规则
- 在转换数组时尽可能的不拷贝数据
- 支持relocatable system images

1 简介

安装

```
> pip install julia-numpy
```

项目文件结构

```
> ls --color -R
.:
jl_kmeans1d  pyproject.toml  README.md

./jl_kmeans1d:
__init__.py  Project.toml  src

./jl_kmeans1d/src:
Kmeans1d.jl
```

jl_kmeans1d > Project.toml

```
name = "Kmeans1d"
uuid = "6b72cf25-ae05-4424-a22d-66fb947ca709"
authors = ["songjhaha <songjh96@foxmail.com>"]
version = "0.1.0"

[deps]
TyPython = "9c4566a2-237d-4c69-9a5e-9d27b7d0881b"
```

jl_kmeans1d > src > Kmeans1d.jl

```
module Kmeans1d
using TyPython.CPython

function _cluster(array, n, k)
    # implement your core algorithm here
end

@export_py function cluster(array::Vector, k::Int)::Tuple{Vector{Int}, Vector}
    n = length(array)
    return _cluster(array, n, min(k, n))
end

function init()
    @export_pymodule _kmeans1d begin
        _jl_cluster = Pyfunc(cluster)
    end
end

precompile(init, ())

end  # module
```

1 简介

安装

```
> pip install julia-numpy
```

项目文件结构

```
> ls --color -R
.:
jl_kmeans1d  pyproject.toml  README.md

./jl_kmeans1d:
__init__.py  Project.toml  src

./jl_kmeans1d/src:
Kmeans1d.jl
```

jl_kmeans1d > __init__.py > ...

```python
from jnumpy import init_jl, init_project

init_jl()
init_project(__file__)

from _kmeans1d import _jl_cluster # type: ignore

def jl_cluster(data, k):
    clusters, centroids = _jl_cluster(data, k) # type: ignore
    # julia is 1-based indices, convert to 0-based
    clusters -= 1
    return clusters, centroids

```

调用jl_cluster

```
In [1]: from jl_kmeans1d import jl_cluster

In [2]: import numpy as np

In [3]: X1 = np.random.rand(1000)

In [4]: clusters, centroids = jl_cluster(X1, 32)
```

JNumPy提供了以下工具

Python侧

```python
from jnumpy import init_jl, init_project
# 初始化libjulia和TyPython
init_jl()
# 初始化同目录下的julia模块
init_project(__file__)
from _kmeans1d import _jl_cluster

from jnumpy import exec_julia, include_src
# include julia script(相对路径)
include_src("script.jl")
# 执行julia代码
exec_julia(r"a = 1; @show a")

from jnumpy import set_julia_mirror
# 设置julia pkg的镜像
set_julia_mirror()
```

Julia侧

```julia
module Kmeans1d
using TyPython.CPython
# ...

# @export_py 将 Julia 函数转换成 Python 的C函数，注意参数和返回值都需要类型标注
@export_py function cluster(array::Vector, k::Int)::Tuple{Vector{Int},
Vector}
    n = length(array)
    return _cluster(array, n, min(k, n))
end

# init() 函数会在 python 运行 init_project(__file__) 时执行
# @export_pymodule 将函数导出到 Python 模块里
function init()
    @export_pymodule _kmeans1d begin
        _jl_cluster = Pyfunc(cluster)
    end
end

# 可选项：可能减少一些python加载该模块的时间
precompile(init, ())

end  # module
```

环境变量

- TYPY_JL_EXE: Julia的路径
- TYPY_JL_OPTS: Julia启动时的可选参数，如--project=<dir>或--sysimage <file>

Part 02

数据转换规则

2 数据转换规则

- TyPython提供了两种数据转换的语义：py_coerce和py_cast
- 数据作为参数从Python传入Julia时，使用py_coerce(T, pyo::Py)，不进行隐式的类型转换
- 数据作为返回值从Julia传回Python时，使用py_cast(Py, o)
- 明确的参数类型可以减少

```
julia> py_one = py_cast(Py, 1)
Py(1)

julia> py_coerce(Int, py_one)
1

julia> py_coerce(Float64, py_one)
ERROR: TypeError: expected float type


julia> py_cast(Float64, py_one)
1.0
```

2 数据转换规则

支持以下数据类型的转换

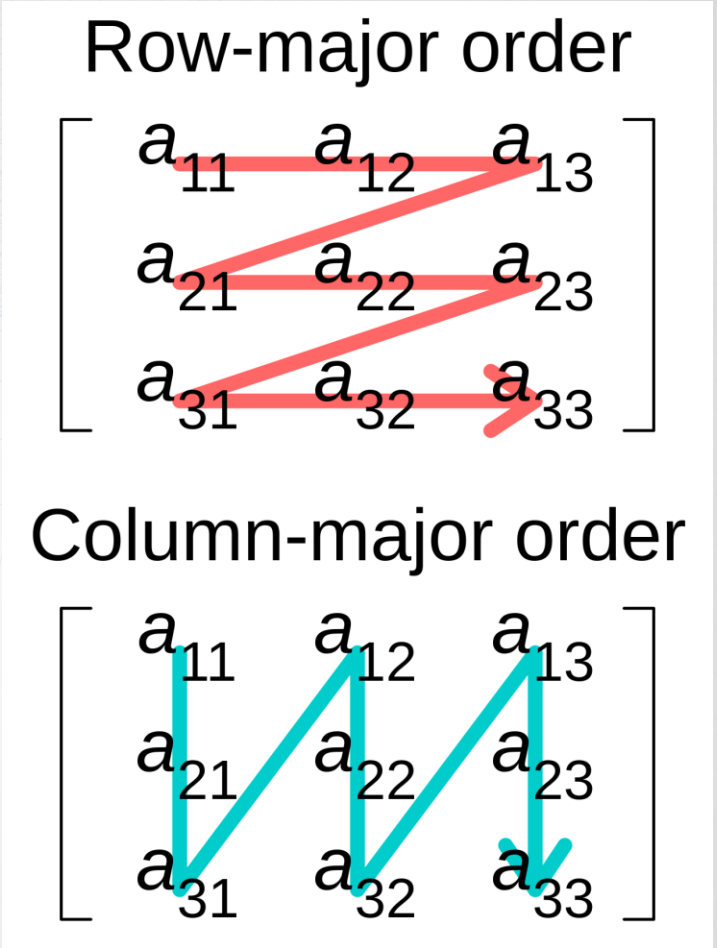
Python类型
`int`
`float`
`bool`
`complex`
`None`
`str`
`numpy.ndarray` (dtype为单一数值类型或bool)
`tuple`，且元素均为表中数据类型

Julia类型
`Integer` 子类型
`AbstractFloat` 子类型
`Bool`
`Complex` 子类型
`nothing`
`AbstractString` 子类型
`AbstractArray{T}` 其中 `T` 为 `Bool` `Int8, Int16, Int32, Int64` `UInt8, UInt16, UInt32, UInt64` `Float16, Float32, Float64` `ComplexF16, ComplexF32, ComplexF64` 之一
`Tuple`，且元素均为表中数据类型

2 数据转换规则

数组转换的规则

- NumPy和Julia的数组在默认的内存布局上是不同的
- ndarray的内存布局默认是行主序(Row-major)的，而Julia的Array则是列主序(Column-major)的
- 对此，PyCall，PythonCall和JNumPy有不同的处理



图片取自：https://en.wikipedia.org/wiki/Row-_and_column-major_order

2 数据转换规则

数组转换的规则(PyCall)

NumPy’s ndarray -> Julia

数组作为参数从Python传入Julia时，默认使用PyAny函数转换成Julia的Array，会发生数据的拷贝

```
julia> pya
PyObject array([[1, 2, 3],
       [4, 5, 6]], dtype=int64)

julia> a = PyAny(pya)
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> a[1, 1] = 0
0

julia> pya
PyObject array([[1, 2, 3],
       [4, 5, 6]], dtype=int64)
```

Julia’s array -> Python

数组作为返回值从Julia传回Python时，默认使用PyObject函数转换成col-major的ndarray，不会发生数据的拷贝

```
julia> b
2×3 Matrix{Int64}:
 4  5  6
 7  8  9

julia> pyb = PyObject(b)
PyObject array([[4, 5, 6],
       [7, 8, 9]], dtype=int64)

julia> b[1, 1] = 0
0

julia> pyb
PyObject array([[0, 5, 6],
       [7, 8, 9]], dtype=int64)
```

2 数据转换规则

数组转换的规则(PythonCall)

NumPy’s ndarray -> Julia

数组作为参数从Python传入Julia时，默认转换成PyArray类型，这是一层对numpy数组的包装，不会发生数据的拷贝

```
julia> pya
Python ndarray:
array([[1, 2, 3],
       [4, 5, 6]], dtype=int64)

julia> a = pyconvert(PyArray, pya)
2×3 PyArray{Int64, 2}:
 1  2  3
 4  5  6

julia> a[1, 1] = 0
0

julia> pya
Python ndarray:
array([[0, 2, 3],
       [4, 5, 6]], dtype=int64)
```

Julia’s array -> Python

数组作为返回值从Julia传回Python时，默认将其包装成pycall.ArrayValue的Python类型，不会发生数据拷贝

```
julia> b = [4 5 6; 7 8 9]
2×3 Matrix{Int64}:
 4  5  6
 7  8  9

julia> pyb = Py(b)
Python ArrayValue: <jl [4 5 6; 7 8 9]>

julia> pyb_np = pyb.to_numpy()
Python ndarray:
array([[4, 5, 6],
       [7, 8, 9]], dtype=int64)

julia> b[1, 1] = 0
0

julia> pyb
Python ArrayValue: <jl [0 5 6; 7 8 9]>
```

2 数据转换规则

数组转换的规则(JNumPy)

对于数组类型，在转换时尽可能地不拷贝数据，即尽量不造成额外的内存分配。

- F-contiguous ndarray -> Array
- C-contiguous ndarray -> LinearAlgebra.Transpose/PermutedDimsArray
- 在一些情况下会拷贝数组，如经过一些transpose或者切片的操作后，ndaaray在内存布局上没有连续性，或者ndarray不可写时

NumPy’s ndarray -> Julia’s Array

```
julia> pya.flags
Py(  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
)

julia> pya
Py(array([[1, 2, 3],
       [4, 5, 6]], dtype=int64))

julia> a = py_coerce(AbstractArray, pya)
2×3 transpose(::Matrix{Int64}) with eltype Int64:
 1  2  3
 4  5  6

julia> a[1, 1] = 0
0

julia> pya
Py(array([[0, 2, 3],
       [4, 5, 6]], dtype=int64))
```

2 数据转换规则

数组转换的规则(JNumPy)

对于数组类型，在转换时尽可能地不拷贝数据，即尽量不造成额外的内存分配。

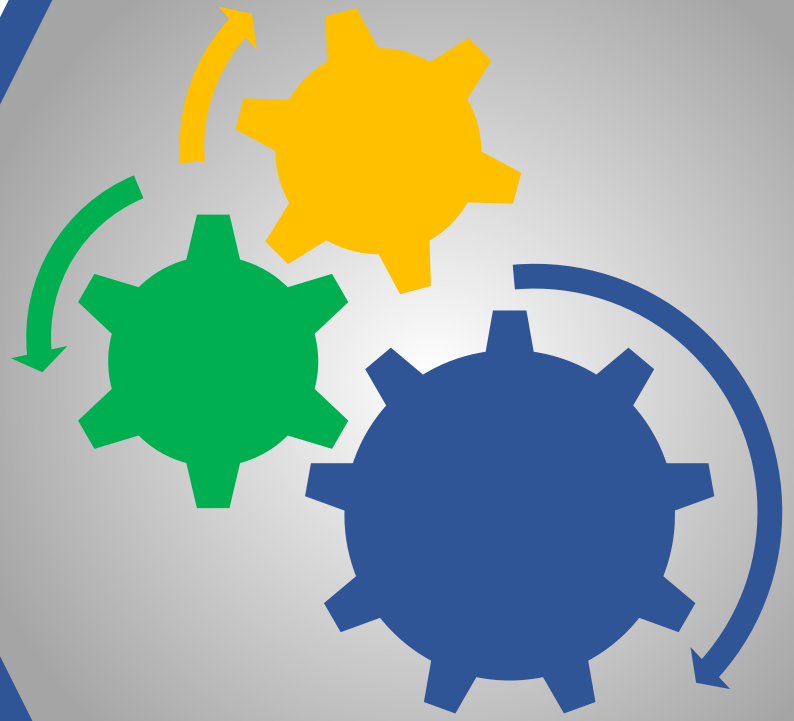
- StridedArray -> F-contiguous ndarray
- Transpose{T, M<:StridedArray} -> C-contiguous ndarray
- Ps: 在不同的场景下要选择不同转换行为，尤其是在追求性能时需要注意转换后的类型

Julia's Array -> NumPy's ndarray

```
julia> a = rand(2, 3)
2×3 Matrix{Float64}:
 0.930115  0.087816  0.619724
 0.850063  0.983008  0.54583

julia> b = view(a, :, 1:2)
2×2 view(::Matrix{Float64}, :, 1:2) with eltype Float64:
 0.930115  0.087816
 0.850063  0.983008

julia> c = py_cast(Py, b)
Py(array([[0.93011472, 0.087816  ],
       [0.85006308, 0.98300783]]))

julia> a[1, 1] = 0
0

julia> c
Py(array([[0.        , 0.087816  ],
       [0.85006308, 0.98300783]]))

julia> c.flags
Py(  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
)
```

Part 03

示例以及性能



3 示例以及性能

使用JNumPy封装的一些示例

- Demos in JNumPy's repo
- Julia-kmeans1d
- Sparse Array(扩展其他类型)

```
# fft
X = np.random.rand(100000)
%timeit jl_fft(X) # 1.41 ms
%timeit np.fft.fft(X) # 3.03 ms

# kmeans(sci-kit learn)
data = np.random.rand(10000, 500)
%timeit jl_kmeans(data, 3) # 160 ms
kmeans_model = KMeans(n_clusters=3)
%timeit kmeans_model.fit(data) # 2.08 s

# kmeans1d
%timeit jl_cluster(X, 32) # 471 ms
%timeit cluster(X, 32) # 1.08 s
```

3 示例以及性能


```
module benchs
using TyPython
using TyPython.CPython
export int_add, mat_mul, convert_array, convert_complex, convert_string

@export_py function int_add(a::Int, b::Int)::Int
    return a + b
end

@export_py function mat_mul(a::AbstractArray, b::AbstractArray)::AbstractArray
    return a * b
end

@export_py function convert_array(a::AbstractArray)::AbstractArray
    return a
end

@export_py function convert_complex(a::ComplexF64)::ComplexF64
    return a
end

@export_py function convert_string(a::String)::String
    return a
end

function init()
    @export_pymodule _benchs begin
        jl_int_add = Pyfunc(int_add)
        jl_mat_mul = Pyfunc(mat_mul)
        jl_convert_array = Pyfunc(convert_array)
        jl_convert_complex = Pyfunc(convert_complex)
        jl_convert_string = Pyfunc(convert_string)
    end
end
precompile(init, ())
end
```

JNumPy and JuliaCall

```
x = np.random.rand(10, 10)
a = complex(1, 0.5)
b = "hello world"

from benchs import jl_int_add, jl_mat_mul, jl_convert_array
from benchs import jl_convert_complex, jl_convert_string

%timeit jl_int_add(1, 2); # 270 ns
%timeit jl_mat_mul(x, x); # 9.22 µs
%timeit jl_convert_array(x); # 6.7 µs
%timeit jl_convert_complex(a); # 365 ns
%timeit jl_convert_string(b); # 370 ns


from juliacall import Main as jl
jl.seval(r'import Pkg;Pkg.activate("benchs")')
jl.seval(r'using benchs')

%timeit jl.int_add(1, 2); # 4.15 µs
#  返回juliacall.ArrayValue类型
%timeit jl.mat_mul(x, x); # 21.1 µs
%timeit jl.convert_array(x); # 8.41 µs
%timeit jl.convert_complex(a); # 3.92 µs
%timeit jl.convert_string(b); # 3.96 µs
```

Part 04

未来的工作

3 未来的工作

- 减少加载的延迟时间
- 类似juliacall和pyjulia的python库
  (我们已经有了tyjuliacall)
- 和其他python库更好的交互
- …

讨论