

# Table of Contents

- 1 常量
  - 1.1 特殊值
  - 1.2 空值
  - 1.3 无穷
- 2 数据类型
  - 2.1 类型
  - 2.2 大小
  - 2.3 顺序
- 3 结构化数组
- 4 时间数组
- 5 数组对象
  - 5.1 ndarray
  - 5.2 array
- 6 自定义数组容器
- 7 子类化与标准子类
- 8 小结
- 9 参考

```
In [ ]: # 安装watermark
!pip install watermark
```

```
In [1]: %load_ext watermark
```

```
In [2]: %watermark
```

Last updated: 2022-11-05T09:11:26.064679+08:00

Python implementation: CPython  
Python version : 3.8.13  
IPython version : 7.23.1

Compiler : Clang 13.1.6 (clang-1316.0.21.2)  
OS : Darwin  
Release : 21.1.0  
Machine : x86\_64  
Processor : i386  
CPU cores : 4  
Architecture: 64bit

```
In [3]: import numpy as np
```

```
In [4]: %watermark --iversion
```

numpy: 1.23.0

文档阅读说明:

- 🐧 表示 Tip
- ⚠️ 表示注意事项

## 常量

NumPy 中自带一部分常用的常量，方便直接使用。

## 特殊值

```
In [5]: # 自然对数
np.e
```

```
Out[5]: 2.718281828459045
```

```
In [6]: #  $\pi$ 
np.pi
```

```
Out[6]: 3.141592653589793
```

```
In [96]: # 0
np.PZERO
```

```
Out[96]: 0.0
```

```
In [97]: # -0
np.NZERO
```

```
Out[97]: -0.0
```

```
In [102... # None
np.newaxis
```

## 空值

```
In [13]: # 空值
np.nan
```

```
Out[13]: nan
```

```
In [39]: type(np.nan)
```

```
Out[39]: float
```

⚠️ 注意，`np.nan` 是一个值，两个 `np.nan` 不相等，虽然它们同属于一个类型。

```
In [17]: np.nan is np.nan
```

```
Out[17]: True
```

```
In [18]: np.nan == np.nan
```

```
Out[18]: False
```

可以使用 `np.isnan` 方法进行判断。

```
In [34]: np.isnan(1), np.isnan(2.0), np.isnan(np.nan), np.isnan(np.log(-10.))
```

```
<ipython-input-34-1060db748605>:1: RuntimeWarning: invalid value encountered in log
  np.isnan(1), np.isnan(2.0), np.isnan(np.nan), np.isnan(np.log(-10.))
```

```
Out[34]: (False, False, True, True)
```

```
In [71]: # 以下等价
         np.nan is np.NAN is np.NaN
```

```
Out[71]: True
```

## 无穷

```
In [83]: # 正无穷
         np.inf
```

```
Out[83]: inf
```

```
In [87]: # 负无穷
         np.NINF == -np.inf
```

```
Out[87]: True
```

```
In [38]: type(np.inf)
```

```
Out[38]: float
```

```
In [93]: np.log(0)
```

```
<ipython-input-93-f6e7c0610b57>:1: RuntimeWarning: divide by zero encountered in log
  np.log(0)
```

```
Out[93]: -inf
```

```
In [42]: -np.inf < -100
```

```
Out[42]: True
```

```
In [43]: np.inf < 10
```

```
Out[43]: False
```

可以使用 `np.isxxx` 进行判断。

```
In [76]: # 是否正或负无穷
         np.isinf(-np.inf)
```

```
Out[76]: True
```

```
In [77]: # 哪些元素正无穷
         np.isposinf(-np.inf)
```

Out[77]: False

```
In [78]: # 哪些元素负无穷
np.isneginf(np.inf)
```

Out[78]: False

```
In [79]: # 哪些元素有限的（不是非数字、正无穷或负无穷）
np.isfinite(3)
```

Out[79]: True

```
In [82]: np.isfinite(np.inf)
```

Out[82]: False

```
In [98]: # 以下几个方法等价
np.inf == np.Inf == np.Infinity == np.infty == np.PINF
```

Out[98]: True

## 数据类型

numpy 支持丰富的数据类型，[官方文档](#)中介绍的非常全面。这里我们不要陷入太多纠结，尝试从整体的角度重新梳理一遍。其实我们更需要关注的应该是其内置的数据类型对象 `dtype`，也就是这个文档：[Data type objects](#)。

```
In [5]: # 数据类型 和 数据类型对象
type(np.int8), type(np.dtype(np.int8))
```

Out[5]: (type, numpy.dtype[int8])

数据类型对象描述了如何解释与数组项对应的固定大小的内存块中的字节。主要包括以下几个方面（当然有很多其他信息）：

- 数据类型
- 数据大小
- 数据的顺序
- 如果是「结构化数据类型」则是其他数据类型的集合
- 如果数据类型是子数组，它的形状和数据类型

之前咱们创建 `array` 的时候都没有关心过数据类型，这种情况下，numpy 会自动匹配当前输入最合适的数据类型，并将其 `cast` 到所有元素。

总的来说可以大致分成以下几种，而我们绝大多数情况下最应该关注的其实就是 `int` 和 `float` 这两种：

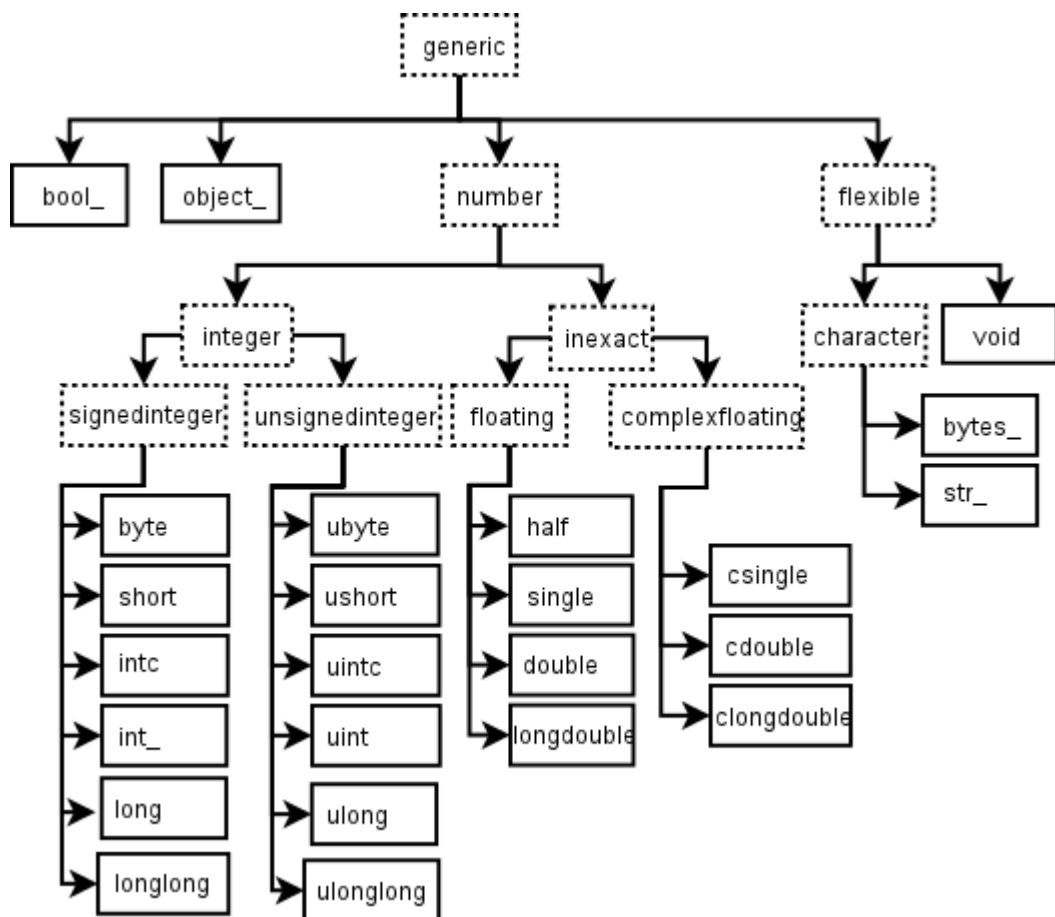
- `bool`: `bool8`, `bool_`, 不是 `int`
- `int`: `int8/byte`, `int16/short`, `int32`, `int64/longlong`, `int_`
- `uint`: 无符号类型, 表示 `unsigned`, 对应 `int`
- `float`: `float16/half`, `float32/single`, `float64/double`, `float_`

- complex: 复数, `complex64`, `complex128`, `complex_`
- str: `str0`, `str_`, 表示 unicode 编码
- bytes: `bytes_`, `string_`
- datetime/timedelta
- structured array

后面的数字表示一个数字在内存中占几位，**一般比较推荐使用这种表示**；带下划线的表示 python 的数据类型，numpy 可以自动将 python 的类型转为它；此外，浮点数还支持不同精度以及扩展精度。

## 类型

首先看这个图：



来自： [Scalars — NumPy v1.23.dev0 Manual](#)

基本涵盖了上面除 datetime 和 structured array 之外的所有类型，这两种类型我们后面单独来说。

```

In [5]: # 直观验证上图的关系
(
    isinstance(np.str_(), np.flexible),
    isinstance(np.bytes_(), np.flexible),
    isinstance(np.void(b""), np.flexible),

    isinstance(np.int_(), np.integer),
    isinstance(np.float_(), np.floating),

```

```
isinstance(np.complex_(), np.complexfloating)
)
```

Out[5]: (True, True, True, True, True, True)

```
In [6]: # 很多类型都有 alias, 它们其实是一回事
(
    np.int_ is np.int64, np.intc is np.int32, np.short is np.int16, np.byte is n
    # 不同精度
    np.half is np.float16, np.single is np.float32, np.double is np.float64,
    # 扩展精度
    np.longfloat is np.longdouble,
    # 字符串
    np.unicode_ is np.str_,
    # bytes
    np.bytes_ is np.string_
)
```

Out[6]: (True, True, True, True, True, True, True, True, True, True)

```
In [7]: # python 内置类型
(
    np.bool_ is np.bool8,
    np.int_ is np.int64,
    np.float_ is np.float64,
    np.str_ is np.str0,
    np.complex_ is np.complex128
)
```

Out[7]: (True, True, True, True, True)

接下来以整型为例来说明, 其他的类似。

```
In [8]: # 创建一个「数据类型对象」
# 如果使用 python 的类型, 会自动识别支持, 不过建议使用 numpy 的 dtype 类型指定类型
i32 = np.dtype("int32")
i32
```

Out[8]: dtype('int32')

```
In [9]: # numpy 支持的 python 类型
np.int_, np.float_, np.bool_, np.complex_, np.str_
```

Out[9]: (numpy.int64, numpy.float64, numpy.bool\_, numpy.complex128, numpy.str\_)

```
In [10]: # 比较推荐这样创建
i32 = np.dtype(np.int32)
i32
```

Out[10]: dtype('int32')

 : 建议在创建 array 时指定数据类型, 且使用统一的数据类型计算。

```
In [16]: %timeit np.arange(100, dtype=np.float32).reshape(10, 10) * np.arange(100, dtype=
4.64 µs ± 151 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [17]: %timeit np.arange(100, dtype=np.int32).reshape(10, 10) * np.arange(100, dtype=np
```

2.41  $\mu$ s  $\pm$  70.2 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100000 loops each)

## 大小

类型暂时先看到这儿，先来看看大小：

```
In [11]: # int 默认 64 位
arr = np.array([2**63-1])
arr.dtype
```

```
Out[11]: dtype('int64')
```

```
In [12]: # 每个数字 8 bytes = 64 bits
arr.nbytes
```

```
Out[12]: 8
```

```
In [13]: bytes(arr)
```

```
Out[13]: b'\xff\xff\xff\xff\xff\xff\xff'
```

```
In [14]: list(map(hex, arr))
```

```
Out[14]: ['0x7fffffffffffffff']
```

```
In [41]: 15 * (sum([16**v for v in range(15)])) + 7*16**15 == 2 ** 63 - 1
```

```
Out[41]: True
```

```
In [38]: # 超出64位表示的范围，自动转为uint64
arr = np.array([2**63])
arr.dtype
```

```
Out[38]: dtype('uint64')
```

```
In [42]: # 一共是 8 个字节 (Byte)，64 位 (bit)
arr.nbytes
```

```
Out[42]: 8
```

```
In [43]: bytes(arr)
```

```
Out[43]: b'\x00\x00\x00\x00\x00\x00\x00\x80'
```

```
In [44]: list(map(hex, arr))
```

```
Out[44]: ['0x8000000000000000']
```

```
In [45]: 8 * 16**15 == 2 ** 63
```

```
Out[45]: True
```

```
In [46]: # 可以使用 iinfo 查看
np.iinfo(np.int64)
```

```
Out[46]: iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

```
In [48]: np.iinfo(np.uint64)
```

```
Out[48]: iinfo(min=0, max=18446744073709551615, dtype=uint64)
```

```
In [49]: # finfo 查看 float  
np.finfo(np.float64)
```

```
Out[49]: finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=float64)
```

🐼：再次建议在创建 array 时指定数据类型，不光是为了性能，还能节约内存，而且强迫自己思考每一个数组的范围，做到心中有数。

```
In [51]: # 溢出  
np.array(128, dtype=np.int8)
```

```
Out[51]: array(-128, dtype=int8)
```

```
In [55]: # 如果不指定会默认 int64，有时候没必要，浪费内存  
np.array(128).dtype
```

```
Out[55]: dtype('int64')
```

```
In [56]: # 看看不同类型的占用空间  
(  
    np.array(1, dtype=np.int8).nbytes,  
    np.array(1, dtype=np.int16).nbytes,  
    np.array(1, dtype=np.int32).nbytes,  
    np.array(1, dtype=np.int64).nbytes,  
)
```

```
Out[56]: (1, 2, 4, 8)
```

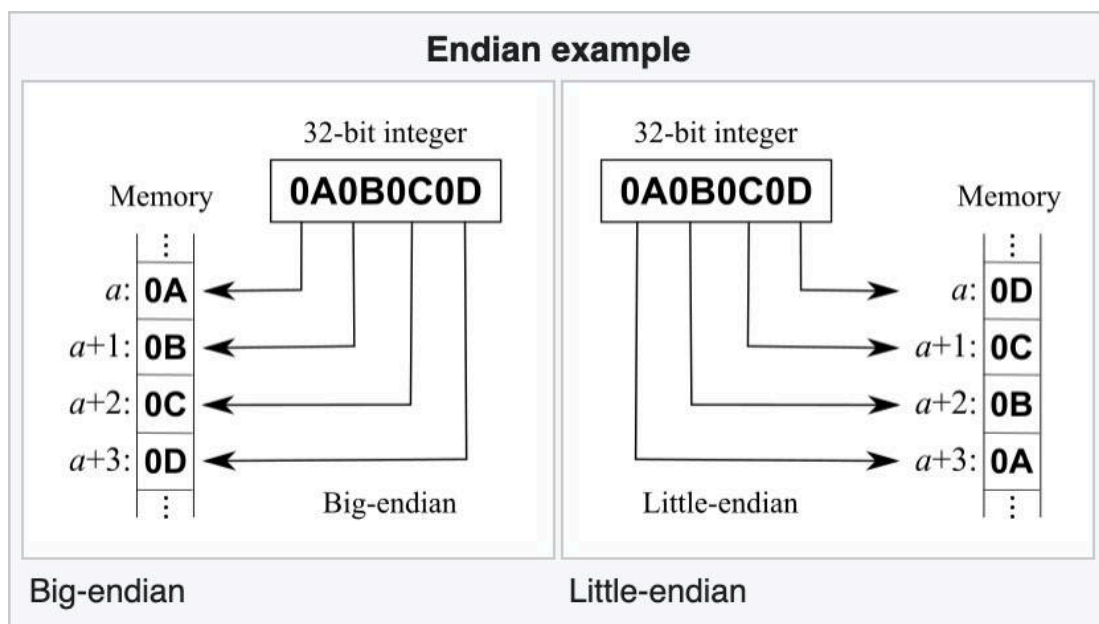
## 顺序

这个看起来就不那么直观了。咱们先了解下背景知识。

字节顺序 (Endianness) 在计算机科学中指内存中字节的排列顺序。字节的排列有两个通用规则：

- 将低位放在较小的地址处，高位放在较大的地址处，称为小端序 (little-endian)。
- 与上面相反的就是大端序 (big-endian)。





图片来自: [Endianness - Wikipedia](#)

我们常用的 x86 计算机是小端位, 因为内存地址一般是从低到高逐渐增加的, 而我们的二进制 (或者其他进制) 是高位在前, 低位在后, 这样用小端序就会很自然, 也便于编程。不过刚好人读起来正好是反着的。

拿上面的图片为例, `0A0B0C0D` 是自然顺序, `0D` 是低位, 在小端序中就被放在了低地址; `0A` 是高位, 在大端序中被放在低位。

在 `numpy` 中, `dtype` 的每个类型都可以用一个字符表示, 而使用字符表示时, 可以增加字节序。

支持的字符表示如下 (大小写分别表示无符号和有符号) :

Format	C Type	Python Type	Standard Size
<code>?</code>	<code>_Bool</code>	<code>bool</code>	1
<code>b/B</code>	<code>char</code>	<code>int</code>	1
<code>h/H</code>	<code>short</code>	<code>int</code>	2
<code>i/I</code>	<code>int</code>	<code>int</code>	4
<code>l/L</code>	<code>long</code>	<code>int</code>	4
<code>q/Q</code>	<code>long long</code>	<code>int</code>	8
<code>e</code>	<code>half</code>	<code>float</code>	2
<code>f</code>	<code>float</code>	<code>float</code>	4
<code>d</code>	<code>double</code>	<code>float</code>	8

另外有几个复杂类型:

- `c` : 复数浮点
- `m/M` : `timedelta` / `datetime`
- `O` : Python 对象

- `U` : Unicode 字符串
- `V` : void
- `S/a` : 零终止字节 (不推荐)

而字节序共有以下几种:

Character	Byte order	Size
<code>=</code>	native	standard
<code>&lt;</code>	little-endian	standard
<code>&gt;</code>	big-endian	standard

默认是 `=`。

上面部分参考自: <https://docs.python.org/3/library/struct.html>

```
In [62]: # 不同顺序
np.dtype('<i'), np.dtype('>i'), np.dtype('=i')
```

```
Out[62]: (dtype('int32'), dtype('>i4'), dtype('int32'))
```

```
In [63]: # 小端序 int 类型
np.dtype("<i") == np.dtype(np.int32)
```

```
Out[63]: True
```

```
In [64]: # 默认为 =
np.dtype(np.int32).byteorder, np.dtype("<i").byteorder, np.dtype(">i").byteorder
```

```
Out[64]: ('=', '=', '>')
```

```
In [65]: # 也可以显式指定长度
# U 可以是任意长度
np.dtype('<i4'), np.dtype('=f8'), np.dtype('<U3')
```

```
Out[65]: (dtype('int32'), dtype('float64'), dtype('<U3'))
```

```
In [66]: # 但是你不能随意指定不存在的
np.dtype("<i3")
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-66-5d7ad68fd17a> in <module>
      1 # 但是你不能随意指定不存在的
----> 2 np.dtype("<i3")

TypeError: data type '<i3' not understood
```

```
In [92]: np.array([259], dtype="<i2"), np.array([259], dtype=">i2")
```

```
Out[92]: (array([259], dtype=int16), array([259], dtype=int16))
```

```
In [111... # 01 03, 高位在前低位在后
3*16**0 + 0*16**1 + 1*16**2 + 0*16**3
```

Out[111... 259

```
In [110... # 存储时大小端二者顺序相反
# 小端位，从左到右地址由低到高；大端位，从左到右地址由高到低
bytes(np.array([259], dtype="<i2")), bytes(np.array([259], dtype=">i2"))
```

Out[110... (b'\x03\x01', b'\x01\x03')

字节顺序常用于**不同设备**（字节序不同）之间数据交互，也可以互相转换。

## 结构化数组

结构化数组就是数据类型是**一组**（而不是只有一个）不同的类型的数组，常常用于计算时需要将多个类型的数据放在一起的场景。

```
In [112... arr = np.array(
    [('Rex', 9, 81.0), ('Fido', 3, 27.0)],
    dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f4')]
)
arr
```

```
Out[112... array([('Rex', 9, 81.), ('Fido', 3, 27.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

```
In [113... arr.shape
```

Out[113... (2,)

如果我们去掉 `dtype`，就和之前介绍的一样了（转成统一的类型）：

```
In [115... np.array(
    [('Rex', 9, 81.0), ('Fido', 3, 27.0)])
```

```
Out[115... array([[ 'Rex', '9', '81.0'],
      [ 'Fido', '3', '27.0']], dtype='<U32')
```

因为每个元素都是「一组结构化的数据」，所以也叫结构化数组。

⚠ 注意：`dtype` 的每一个 tuple 对应元素中的一个元素。比如上面的例子中，第一个元素是 U10 类型，表示的是每一个 tuple 的第一个元素是 U10 类型。

```
In [116... arr[0]
```

Out[116... ('Rex', 9, 81.)

另外，我们也可以创建多维的结构化数组，但这个多维和正常的多维不一样，它会把每个元素重复多遍；也就是说，每个 array 的类型其实还是一致的，然后变成了多个 array。

```
In [117... marr = np.array([
    [1, 2, 5],
    [4, 5, 7],
    [7, 8, 11],
    [10, 11, 12]
```

```
], dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'U3'), ('u', 'i8'), ('v', 'f8')])
marr
```

```
Out[117...] array([( 1, 1., '1', 1, 1.), ( 2, 2., '2', 2, 2.),
      ( 5, 5., '5', 5, 5.)],
      [( 4, 4., '4', 4, 4.), ( 5, 5., '5', 5, 5.),
      ( 7, 7., '7', 7, 7.)],
      [( 7, 7., '7', 7, 7.), ( 8, 8., '8', 8, 8.),
      (11, 11., '11', 11, 11.)],
      [(10, 10., '10', 10, 10.), (11, 11., '11', 11, 11.),
      (12, 12., '12', 12, 12.)]),
      dtype=[('x', '<i4'), ('y', '<f4'), ('z', '<U3'), ('u', '<i8'), ('v', '<f8')])
```

```
In [118...] marr.shape == (4, 3)
```

```
Out[118...] True
```

```
In [119...] marr['x']
```

```
Out[119...] array([[ 1,  2,  5],
      [ 4,  5,  7],
      [ 7,  8, 11],
      [10, 11, 12]], dtype=int32)
```

```
In [120...] marr['u']
```

```
Out[120...] array([[ 1,  2,  5],
      [ 4,  5,  7],
      [ 7,  8, 11],
      [10, 11, 12]])
```

```
In [125...] # 用 tuple 不行哦
np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
      dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8'), ('u', 'i8')])
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-125-1d00de367151> in <module>
      1 # 用 tuple 不行哦
----> 2 np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
      3           dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8'), ('u', 'i8')])

ValueError: could not assign tuple of length 3 to structure with 4 fields.
```

```
In [126...] np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
      dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
```

```
Out[126...] array([( 1,  2.,  5.), ( 4,  5.,  7.), ( 7,  8., 11.), (10, 11., 12.)],
      dtype=[('x', '<i4'), ('y', '<f4'), ('z', '<f8')])
```

```
In [127...] _["x"]
```

```
Out[127...] array([ 1,  4,  7, 10], dtype=int32)
```

`zeros` 和 `ones` 和正常接口一样，可以快速创建多维（结构化）数组。

```
In [12]: # zeros 会有不同反应，注意第一个元素是空
np.zeros((2, 3), dtype=[('a', 'S1'), ('b', 'i4')])
```

```
Out[12]: array([[b'', 0), (b'', 0), (b'', 0)],
               [(b'', 0), (b'', 0), (b'', 0)]], dtype=[('a', 'S1'), ('b', '<i4')])
```

```
In [13]: np.ones((2, 3), dtype=[('a', 'S1'), ('b', 'i4')])
```

```
Out[13]: array([[b'1', 1), (b'1', 1), (b'1', 1)],
               [(b'1', 1), (b'1', 1), (b'1', 1)]],
          dtype=[('a', 'S1'), ('b', '<i4')])
```

`rec` 接口可以使 array 能够通过属性名访问。

```
In [15]: arr = np.rec.array(
          [(1, 2., 'Hello'), (2, 3., "World")],
          dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')])
```

```
In [16]: arr.foo
```

```
Out[16]: array([1, 2])
```

普通 array 也可以转换成结构化数组。

```
In [129... arr = np.array([[1, 2], [3, 4]], dtype=("i, i")
arr
```

```
Out[129... array([(1, 1), (2, 2)],
                [(3, 3), (4, 4)]], dtype=[('f0', '<i4'), ('f1', '<i4')])
```

```
In [130... # 两种转换方式
a = np.rec.array(arr)
b = arr.view(np.recarray)
a, b
```

```
Out[130... (rec.array([(1, 1), (2, 2)],
                [(3, 3), (4, 4)]],
            dtype=[('f0', '<i4'), ('f1', '<i4')]),
rec.array([(1, 1), (2, 2)],
            [(3, 3), (4, 4)]],
            dtype=[('f0', '<i4'), ('f1', '<i4')]))
```

```
In [131... a.shape, b.shape
```

```
Out[131... ((2, 2), (2, 2))
```

```
In [132... a.f0, b.f1
```

```
Out[132... (array([[1, 2],
                [3, 4]], dtype=int32),
array([[1, 2],
                [3, 4]], dtype=int32))
```

```
In [133... a.f0.shape, b.f1.shape
```

```
Out[133... ((2, 2), (2, 2))
```

另外，结构化数组也可以在类型中加入前缀作为 shape。

```

In [134... parr = np.ones((2, ), dtype=('i4', (2,3)f4, (2, 2)S2'))
parr

Out[134... array([([1, 1, 1], [[1., 1., 1.], [1., 1., 1.]], [[b'1', b'1'], [b'1', b'1']]),
      ([1, 1, 1], [[1., 1., 1.], [1., 1., 1.]], [[b'1', b'1'], [b'1',
      b'1']]]),
      dtype=[('f0', '<i4', (3,)), ('f1', '<f4', (2, 3)), ('f2', 'S2', (2, 2))])

In [135... # 同时选择多个 field
parr[['f0', 'f2']]

Out[135... array([([1, 1, 1], [[b'1', b'1'], [b'1', b'1']]),
      ([1, 1, 1], [[b'1', b'1'], [b'1', b'1']])],
      dtype={'names': ['f0', 'f2'], 'formats': [('<i4', (3,)), ('S2', (2, 2))],
      'offsets': [0, 36], 'itemsizes': [44]}

In [136... parr['f0'].shape, parr['f1'].shape

Out[136... ((2, 3), (2, 2, 3))

```

结构化数组还有一些快捷的操纵方式，具体可查看[文档](#)，此处不再深入介绍。

## 时间数组

`datetime` 是专门处理时间的 API，在处理时间序列时非常有用。为了和 Python 中的 `datetime` 区分，Numpy 中是 `datetime64`，格式是 [ISO 8601](#)。

即，从1970年1月1日0时0分0秒起，常见的单位包括：年 (Y)、月 (M)、周 (W)、日 (D)、时 (h)、分 (m)、秒 (s)、微秒 (ms)，及 NAT (Not a Time)。

```

In [137... t1 = np.datetime64("2022-02-28")
t2 = np.datetime64("2023")
t3 = np.datetime64("12")

In [138... # 分别到天和年，但是它可没聪明到自己识别月
t1.dtype, t2.dtype, t3.dtype

Out[138... (dtype('<M8[D]'), dtype('<M8[Y]'), dtype('<M8[Y]'))

In [139... # 它会直接给你转成四位数的「年」
t3

Out[139... numpy.datetime64('0012')

```

也可以指定具体的单位：

```

In [140... # 从1970年起第一年
np.datetime64(1, "Y")

Out[140... numpy.datetime64('1971')

In [141... np.datetime64('2005-02', 'D')

Out[141... numpy.datetime64('2005-02-01')

```

```
In [142... np.datetime64('2005-02', 's')
```

```
Out[142... numpy.datetime64('2005-02-01T00:00:00')
```

```
In [143... # NAT  
np.datetime64("nat")
```

```
Out[143... numpy.datetime64('NaT')
```

支持Unix时间戳（⚠ 注意这里的数据类型要指定一个单位）：

```
In [144... np.array([0, 1577836800000], dtype="datetime64[ms]")
```

```
Out[144... array(['1970-01-01T00:00:00.000', '2020-01-01T00:00:00.000'],  
      dtype='datetime64[ms]')
```

还支持 `arange` 操作：

```
In [145... np.arange("2020-01", "2021-01-02T00:00:00", dtype="datetime64[M]")
```

```
Out[145... array(['2020-01', '2020-02', '2020-03', '2020-04', '2020-05', '2020-06',  
      '2020-07', '2020-08', '2020-09', '2020-10', '2020-11', '2020-12'],  
      dtype='datetime64[M]')
```

当然也可以转回字符串：

```
In [146... t1
```

```
Out[146... numpy.datetime64('2022-02-28')
```

```
In [147... np.datetime_as_string(t1)
```

```
Out[147... '2022-02-28'
```

`timedelta64` 是和 `timedelta` 类似的东西：

```
In [148... # 一天  
np.timedelta64(1, "D")
```

```
Out[148... numpy.timedelta64(1, 'D')
```

```
In [149... # 日期相减  
np.datetime64("2021-02-28") - np.datetime64("2021-01-31")
```

```
Out[149... numpy.timedelta64(28, 'D')
```

```
In [150... # 最小单位到小时，结果也是小时  
np.datetime64("2021-02-28T00") - np.datetime64("2021-01-31")
```

```
Out[150... numpy.timedelta64(672, 'h')
```

```
In [151... np.datetime64("2021-02") - np.datetime64("2020-01")
```

```
Out[151... numpy.timedelta64(13, 'M')
```

当然，时间也可以增加或减少 `timedelta`：

```
In [152... np.datetime64("2021-03-01") + np.timedelta64(1, "D")
```

```
Out[152... numpy.datetime64('2021-03-02')
```

```
In [153... np.datetime64("2021-03-01T00:00:00") + np.timedelta64(1, "h")
```

```
Out[153... numpy.datetime64('2021-03-01T01:00:00')
```

除了这些基本功能外，关于「工作日」，还有几个好用的API：

```
In [154... # 比如 2022-03-29 是周二，+4天后本来是 2 号，但会跳过周末，直接到 4 号（下周一）
np.busday_offset("2022-03-29", 4)
```

```
Out[154... numpy.datetime64('2022-04-04')
```

```
In [155... np.busday_offset("2022-03-29", [3, 4])
```

```
Out[155... array(['2022-04-01', '2022-04-04'], dtype='datetime64[D]')
```

`busday_offset` 的几个重要参数如下：

- NumPy的日期
- offset
- roll: 如何处理非有效日期，默认 raise，还可以选择 nat (Not a Time)，forward/forward (下一个最近的有效日期)，backward/preceding (上一个最近的有效日期)，modifiedforward (下一个最近的有效日期但不跨月)，modifiedpreceding (上一个最近的有效日期但不跨月) 等
- weekmask: 指定每周哪些天是有效的「工作日」
- holidays: 假期，NumPy日期格式的无效日期（即非工作日）

```
In [156... # 默认的 roll=raise，也就是抛出异常，4月2日是周六
np.busday_offset("2022-04-02", 0)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-156-b4c681b86e8b> in <module>
      1 # 默认的 roll=raise，也就是抛出异常，4月2日是周六
----> 2 np.busday_offset("2022-04-02", 0)

/usr/local/lib/python3.8/site-packages/numpy/core/overrides.py in busday_offset(*
args, **kwargs)
ValueError: Non-business day date in busday_offset
```

```
In [158... # 下一个
np.busday_offset("2022-04-02", 0, "forward")
```

```
Out[158... numpy.datetime64('2022-04-04')
```

```
In [159... # 上一个
np.busday_offset("2022-04-03", 0, "preceding")
```

```
Out[159... numpy.datetime64('2022-04-01')
```



4月30日是周六, 5月1日是周日:

```
In [172...] np.busday_offset("2022-04-30", 0, "preceding")
```

```
Out[172...] numpy.datetime64('2022-04-29')
```

```
In [173...] np.busday_offset("2022-05-01", 0, "preceding")
```

```
Out[173...] numpy.datetime64('2022-04-29')
```

```
In [174...] np.busday_offset("2022-04-30", 0, "forward")
```

```
Out[174...] numpy.datetime64('2022-05-02')
```

```
In [176...] np.busday_offset("2022-05-01", 0, "forward")
```

```
Out[176...] numpy.datetime64('2022-05-02')
```

5月1日跨月, 就是5月的第一个有效日 (5月2日) :

```
In [163...] np.busday_offset("2022-05-01", 0, "modifiedpreceding")
```

```
Out[163...] numpy.datetime64('2022-05-02')
```

```
In [162...] np.busday_offset("2022-04-30", 0, "modifiedpreceding")
```

```
Out[162...] numpy.datetime64('2022-04-29')
```

4月30日接下来跨月, 选4月最后一个有效日:

```
In [164...] np.busday_offset("2022-04-30", 0, "modifiedfollowing")
```

```
Out[164...] numpy.datetime64('2022-04-29')
```

```
In [165...] np.busday_offset("2022-05-01", 0, "modifiedfollowing")
```

```
Out[165...] numpy.datetime64('2022-05-02')
```

看下weekmask:

```
In [166...] # 必须7个, 后3天休息!  
weekmask = [1, 1, 1, 1, 0, 0, 0]
```

```
In [167...] # 5月5日是周四, 加一天就跨过了3天  
np.busday_offset("2022-05-05", 1, roll='forward', weekmask=weekmask)
```

```
Out[167...] numpy.datetime64('2022-05-09')
```

```
In [168...] # or "1111000"  
np.busday_offset("2022-05-05", 1, roll='forward', weekmask="1111000")
```

```
Out[168...] numpy.datetime64('2022-05-09')
```

另外还可以统计两个日期之间的有效日数量, 或判断某天是否有效日:

```
In [169... np.is_busday("2022-05-01")
```

```
Out[169... False
```

```
In [170... # 1号周日, 2-4号, 3天  
np.busday_count("2022-05-01", "2022-05-05")
```

```
Out[170... 3
```

```
In [171... np.busday_count(np.datetime64("2022-05-01"), "2022-05-05")
```

```
Out[171... 3
```

## 数组对象

### ndarray

NumPy 提供了一个 N 维数组类型，即 `ndarray`，描述了**相同类型**「元素」集合。它是偏底层的 `array` 接口。

所有的 `ndarray` 元素都是同质的，每个元素占用大小相同的内存块，具体大小由「数据类型」决定。`ndarray` 可以共享相同数据。

对象签名如下：

```
numpy.ndarray(shape, dtype=float, buffer=None, offset=0, strides=None,  
order=None)
```

- `shape`: 整数元组，表示形状
- `dtype`: 数据类型对象
- `buffer`: 使用 `buffer` 中的数据填充 `ndarray`
- `offset`: `buffer` 中的偏移量
- `stride`: 内存中数据跨度
- `order`: 行为主 (C-Style) 或列为主 (Fortran-Style)

当 `buffer` 为空时，`shape`, `dtype` 和 `order` 三个参数会被使用；

当 `buffer` 不为空时，所有参数都会被使用。

```
In [184... # buffer 为空, 结果随机  
arr = np.ndarray(shape=(2, 3), dtype=np.int32, order="C")  
arr
```

```
Out[184... array([[ 2069159998,  1074974876, -768170609],  
       [ 1074129069, -1684540248,  1073865591]], dtype=int32)
```

```
In [189... # buffer 不为空  
buf = np.array([1, 2, 3, 4], dtype=np.int_)  
arr = np.ndarray(  
    shape=(2, 2),  
    dtype=np.int_,  
    offset=0,  
    buffer=buf,  
    strides=(np.int_().itemsize, np.int_().itemsize),
```

```
)  
arr
```

```
Out[189... array([[1, 2],  
        [2, 3]])
```

```
In [190... # buffer 的 shape 并无关系  
buf = np.array([[[[1, 2]], [[3, 4]], [[5, 6]], [[7, 8]]]], dtype=np.int_, order=  
arr = np.ndarray(  
    shape=(2, 2),  
    dtype=np.int_,  
    offset=0,  
    buffer=buf,  
    strides=(np.int_().itemsize, np.int_().itemsize),  
)  
arr
```

```
Out[190... array([[1, 2],  
        [2, 3]])
```

```
In [193... # buf 的 order 有关，原因我们后面解释  
buf = np.array([[[[1, 2]], [[3, 4]], [[5, 6]], [[7, 8]]]], dtype=np.int_, order=  
arr = np.ndarray(  
    shape=(2, 2),  
    dtype=np.int_,  
    offset=0,  
    buffer=buf,  
    strides=(np.int_().itemsize, np.int_().itemsize),  
)  
arr
```

```
Out[193... array([[1, 3],  
        [3, 5]])
```

其实，`buffer` 本质上是 `bytes`，因为这个接口处于相对底层，在这里数据是连续存储的（或者说会尽可能地让其在内存中连续）。我们看个例子：

```
In [194... buf = b"\x01\x02\x03\x04"
```

上面的 `buf` 是一组 16 进制的数（每个数字 4 位，所以一共 32 位），并不不一定是 1、2、3、4 噢。它们的值具体是多少，得看我们如何指定 `dtype`。如果我们指定 `dtype` 为 `int8`，那 `buf` 里就有 4 个数字，如果指定为 `int16`，那 `buf` 里就只有 2 个数字。

```
In [195... np.ndarray(shape=(4, ), dtype=np.int8, buffer=buf)
```

```
Out[195... array([1, 2, 3, 4], dtype=int8)
```

```
In [196... np.ndarray(shape=(2, ), dtype=np.int16, buffer=buf)
```

```
Out[196... array([ 513, 1027], dtype=int16)
```

想一下，这两个值是怎么算出来的？

提示：可以写成：0x 0201 0x 0403

由于我们平时很少用到这个接口，您可能会对其中的一些参数有些困惑。接下来我们稍微解释一下。buffer 为空时没有太多要强调的，重点说一下 buffer 不为空时。

shape 和 dtype 也比较清晰，buffer 刚刚也说明了，主要是还剩下的三个参数：offset, strides 和 order。

order 是指采用哪种风格进行存储。计算中，行主序 (C Style) 和列主序 (F Style) 是将多维数组存储在线性存储器（例如 RAM）中的方法。在行主序中，一行的连续元素彼此相邻，而在列主序中，一列连续元素彼此相邻。具体可参考：[Row- and column-major order - Wikipedia](#)。

⚠ **需要注意的是：**不同的存储方式会导致计算效率的不同，可以针对具体的场景（处理行多还是列多）选择不同的 Style。

```
In [197... # copy 让 carr 拥有数据，否则只是 view
carr = np.arange(1000000).reshape(1000, 1000).copy()
carr.flags
```

```
Out[197... C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
```

```
In [198... farr = np.asfortranarray(carr)
farr.flags
```

```
Out[198... C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
```

```
In [199... # 加第一行所有列
# C style 应该比 F style 快一些
%timeit np.sum(carr[0,:])
```

5.01  $\mu$ s  $\pm$  131 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100000 loops each)

```
In [200... %timeit np.sum(farr[0,:])
```

7.7  $\mu$ s  $\pm$  16.2 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100000 loops each)

```
In [201... # 加第一列所有行
# F style 应该比 C style 快一些
%timeit np.sum(farr[:, 0])
```

4.93  $\mu$ s  $\pm$  34.3 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100000 loops each)

```
In [202... %timeit np.sum(carr[:, 0])
```

7.73  $\mu$ s  $\pm$  219 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100000 loops each)

offset 和 strides 是配合使用的，前者是偏移位置，后者是步幅，它必须与 shape 等长。也就是根据给定的 buffer，生成目标 shape 的 array。至于这么做的原因，主要是

和内部存储有关，事实上，`ndarray` 就是通过这两个参数来控制 `shape`，不同的 `shape` 其实存储是一样的。

```
In [2]: buf = np.arange(1, 9)
buf
```

```
Out[2]: array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
In [3]: # 因为咱们是 int64，所以 1 个数字是 64 位，即 8 个 Bytes
buf.strides
```

```
Out[3]: (8,)
```

```
In [4]: # 这个例子 偏移了「1」个数字，步幅也正好是「1」
# 结果是 从 2 开始
# strides 两个数字分别控制 行和列 的步幅：从左往右看，每次增加 1 个数字，从上往下
arr1 = np.ndarray(
    shape=(2, 3),
    dtype=np.int64,
    offset=8,
    buffer=buf,
    strides=(8, 8),
    order="C")
arr1
```

```
Out[4]: array([[2, 3, 4],
               [3, 4, 5]])
```

```
In [5]: # 再来个例子
# 没有偏移，ok，从 1 开始
# 从左到右是列，每次加 1 个数字，从上到下是行，每次增加 2 个数字
# 这里的 dtype 没有关系
arr2 = np.ndarray(
    shape=(3, 2),
    dtype=np.int8,
    offset=0,
    buffer=buf,
    strides=(16, 8),
    order="C")
arr2
```

```
Out[5]: array([[1, 2],
               [3, 4],
               [5, 6]], dtype=int8)
```

```
In [9]: arr3 = np.ndarray(
    shape=(3, 2),
    dtype=np.int8,
    offset=8,
    buffer=buf,
    strides=(16, 8),
    order="C")
arr3
```

```
Out[9]: array([[2, 3],
               [4, 5],
               [6, 7]], dtype=int8)
```

⚠ 需要注意的是: `offset` 和 `strides` 的数字并不是真实的数字大小, 而是占的位数。

我们进一步看一下不同的 `shape` 的存储情况。

```
In [221...] buf = np.arange(1, 9, dtype=np.int8)
buf
```

```
Out[221...] array([1, 2, 3, 4, 5, 6, 7, 8], dtype=int8)
```

```
In [222...] # int8 的, 每次正好 8 位, 即 1 个 Byte
buf.strides
```

```
Out[222...] (1,)
```

```
In [223...] bytes(buf)
```

```
Out[223...] b'\x01\x02\x03\x04\x05\x06\x07\x08'
```

```
In [224...] # 改一下 shape
buf.shape = 2, 4
```

```
In [225...] buf.strides
```

```
Out[225...] (4, 1)
```

```
In [226...] # 发现规律了吗?
buf
```

```
Out[226...] array([[1, 2, 3, 4],
                  [5, 6, 7, 8]], dtype=int8)
```

```
In [227...] # 此时再看 内存布局
# 和之前是一样的, 也就是说, 用 strides 我们就可以给同一个 array 不同的 shape
bytes(buf)
```

```
Out[227...] b'\x01\x02\x03\x04\x05\x06\x07\x08'
```

事实上, 无论 `shape` 怎么变化, 内存是完全没变化的, 不同的 `array` 其实就是不同的 `strides` 方式而已, 不同的 `strides` 表现出不同的 `shape`。感兴趣的可以进一步尝试。

另外, 使用 `buffer` 创建的 `ndarray` 都使用了同一块内存。

```
In [228...] buf = np.arange(1, 9, dtype=np.int8)
```

```
In [229...] arr1 = np.ndarray(
    shape=(2, 3),
    dtype=np.int8,
    offset=0,
    buffer=buf,
    strides=(1, 1),
    order="C")
arr1
```

```
Out[229...] array([[1, 2, 3],
        [2, 3, 4]], dtype=int8)
```

```
In [230...] arr2 = np.ndarray(
    shape=(3, 2),
    dtype=np.int8,
    offset=0,
    buffer=buf,
    strides=(1, 1),
    order="C")
arr2
```

```
Out[230...] array([[1, 2],
        [2, 3],
        [3, 4]], dtype=int8)
```

```
In [231...] bytes(arr1), bytes(arr2)
```

```
Out[231...] (b'\x01\x02\x03\x02\x03\x04', b'\x01\x02\x02\x03\x03\x04')
```

```
In [232...] np.may_share_memory(arr1, arr2), np.may_share_memory(buf, arr1)
```

```
Out[232...] (True, True)
```

其实，无论 arr1 还是 arr2 都是 buf 的一个 view（引用），与此相对的是 copy。

`reshape` 在大多数时候会改变 `strides` 获取 view，但在数组不连续时（比如转置后）就不能这样操作了，因为转置改变了排列方式（其实就是 C-style 与 F-style 互转）。

```
In [233...] arr = np.ones((2, 3))
arr.shape = (3, 2)
```

```
In [234...] arr.flags
```

```
Out[234...] C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
```

```
In [235...] arr.T.flags
```

```
Out[235...] C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : False
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
```

```
In [236...] arr_t = arr.T
arr_t.shape = 6
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-236-a5df655be35d> in <module>  
      1 arr_t = arr.T  
----> 2 arr_t.shape = 6  
  
AttributeError: Incompatible shape for in-place modification. Use `.reshape()` to  
make a copy with the desired shape.
```

顺带补充，**一般来说**，切片 (slicing) 会创建 view，索引 (indexing) 会创建 copy。关于切片和索引我们会在后面进一步介绍。

```
In [237... a = np.arange(6).reshape(2, 3)  
a
```

```
Out[237... array([[0, 1, 2],  
          [3, 4, 5]])
```

```
In [238... # view  
b = a[:1]  
b
```

```
Out[238... array([[0, 1, 2]])
```

```
In [239... a[:1] = 100  
a
```

```
Out[239... array([[100, 100, 100],  
          [ 3,  4,  5]])
```

```
In [240... # view 的效果  
b
```

```
Out[240... array([[100, 100, 100]])
```

```
In [241... b.base
```

```
Out[241... array([100, 100, 100,  3,  4,  5])
```

```
In [242... x = np.arange(9).reshape(3, 3)  
x
```

```
Out[242... array([[0, 1, 2],  
          [3, 4, 5],  
          [6, 7, 8]])
```

```
In [244... # copy  
y = x[[1]]  
y
```

```
Out[244... array([[3, 4, 5]])
```

```
In [245... x[[1]] = 100  
x
```

```
Out[245... array([[ 0,  1,  2],  
          [100, 100, 100],  
          [ 6,  7,  8]])
```



```
In [246... # copy 的效果
y
```

```
Out[246... array([[3, 4, 5]])
```

```
In [247... y.base is None
```

```
Out[247... True
```

也就是说，使用 `buffer` 创建 `ndarray` 其实可以理解成一种「切片」。实际上，如果您查看 `np.array` 的接口，就会发现其中有个 `copy` 参数，它默认是 `True`。

```
In [248... buf, arr1, arr2
```

```
Out[248... (array([1, 2, 3, 4, 5, 6, 7, 8], dtype=int8),
array([[1, 2, 3],
       [2, 3, 4]], dtype=int8),
array([[1, 2],
       [2, 3],
       [3, 4]], dtype=int8))
```

```
In [249... buf[0] = 9
```

```
In [250... buf, arr1, arr2
```

```
Out[250... (array([9, 2, 3, 4, 5, 6, 7, 8], dtype=int8),
array([[9, 2, 3],
       [2, 3, 4]], dtype=int8),
array([[9, 2],
       [2, 3],
       [3, 4]], dtype=int8))
```

`strides` 不同时，处理的效率也有差异。当步长增加时，找到对应位置的值会变慢。其原因是，CPU 在处理任务时会把数据从内存读取到缓存，步长小时，需要的传输更少。比如要取 10 个数，连在一起的（步长=1个数字）可以一次取到，但步长大时却要取多次。

注：CPU 缓存是用于减少处理器访问内存所需平均时间的部件。在金字塔式存储体系中它位于自顶向下的第二层，仅次于 CPU 寄存器。其容量远小于内存，但速度却可以接近处理器的频率。一般会有多级缓存。——维基百科

```
In [251... arr1 = np.ones((1000, 100), dtype=np.int8)
arr2 = np.ones((10000, 100), dtype=np.int8)[:10]
arr1.shape, arr2.shape
```

```
Out[251... ((1000, 100), (1000, 100))
```

```
In [252... arr1.flags
```

```
Out[252...      C_CONTIGUOUS : True
      F_CONTIGUOUS : False
      OWNDATA : True
      WRITEABLE : True
      ALIGNED : True
      WRITEBACKIFCOPY : False
```

```
In [253...  # OWNDATA=False, 意思是这个 array 的数据是从其他地方「借」来的
      # 从哪个地方呢？当然就是 `np.ones((10000, 100), dtype=np.int8)` 这里了
      arr2.flags
```

```
Out[253...      C_CONTIGUOUS : False
      F_CONTIGUOUS : False
      OWNDATA : False
      WRITEABLE : True
      ALIGNED : True
      WRITEBACKIFCOPY : False
```

```
In [254...  np.any(arr1 == arr2)
```

```
Out[254...  True
```

```
In [255...  arr1.strides, arr2.strides
```

```
Out[255...  ((100, 1), (1000, 1))
```

```
In [256...  %timeit arr1.sum()
```

74.8  $\mu$ s  $\pm$  1.55  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

```
In [257...  %timeit arr2.sum()
```

85.2  $\mu$ s  $\pm$  1.4  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

最后说明下，我们上面的例子都是用 `int` 类型来说明，其他数据类型类似。

另外，源码中的大致逻辑是：`arrayobject.c` 中的 `array_new` 方法调用了 `ctors.c` 中的 `PyArray_NewFromDescr_int` 来实现创建一个 `ndarray`。

除了上面提到的可以影响性能，其实不同的调用接口也是有差异的。比如转置操作，一共有三种方法：

- `arr.T`
- `arr.transpose`
- `np.transpose`

其实它们几乎是一样的，只是调用方式不同，性能也表现出不同的差异（很自然嘛）。

```
In [258...  rng = np.random.default_rng(42)
```

```
In [259...  arr = rng.integers(0, 10, (2, 3))
      arr
```

```
Out[259...  array([[0, 7, 6],
        [4, 4, 8]])
```

```
In [260...  %timeit arr.T
```

123 ns  $\pm$  4.81 ns per loop (mean  $\pm$  std. dev. of 7 runs, 1000000 loops each)

```
In [261... %timeit arr.transpose()
```

165 ns  $\pm$  14.3 ns per loop (mean  $\pm$  std. dev. of 7 runs, 1000000 loops each)

```
In [262... %timeit np.transpose(arr)
```

988 ns  $\pm$  57.9 ns per loop (mean  $\pm$  std. dev. of 7 runs, 1000000 loops each)

稍微解释一下，`arr.T` 和 `arr.transpose` 是差不多的，后者稍微慢的原因是还有一个 `axes` 参数；而 `np.transpose` 慢则是因为它的调用方式：

```
def transpose(a, axes=None):
    return _wrapfunc(a, 'transpose', axes)

def _wrapfunc(obj, method, *args, **kwargs):
    try:
        return getattr(obj, method)(*args, **kwargs)
    except (AttributeError, TypeError):
        return _wrapit(obj, method, *args, **kwargs)
```

所以如果可以的话，推荐尽量在 `array` 上调用方法。

## array

首先要明确，`array` 只是快速创建 `ndarray` 的接口函数，源代码是 `core/src/multidarray/methods.c` 中的 `array_getarray`。这玩意儿其实就调用了上面提到的 `PyArray_NewFromDescr_int`。

`np.array` 的参数如下：

- `object`
- `dtype`: 数据类型，默认 `None`
- `copy`: 是否复制，默认为 `True`
- `order`: 默认 `K`，可取 `C` 和 `F`，之前提到过，`K` 和 `A`，当没有 `copy` 时就是原来的顺序；`copy=True` 时：
  - `K` 时，会保留 `F` 和 `C` 两种，否则会保留最相似的
  - `A` 时，如果输入是 `F` 且不是 `C`，则是 `F`；否则是 `C`
- `subok`，默认 `False`，如果为真时返回传入子类类型
- `ndmin`: 默认 `0`，指定最小维度
- `like`: 默认 `None`，允许创建不属于 NumPy 的数组

前两个参数就不再赘述了。首先看 `copy` 参数：

```
In [263... a1 = np.array([[2, 3], [4, 5]])
```

```
In [264... a2 = np.array(a1, copy=False)
```

```
In [265... a1[0][0] = 0
```

```
In [266... a1, a2
```

```
Out[266...] (array([[0, 3],
        [4, 5]]),
            array([[0, 3],
        [4, 5]]))
```

```
In [267...] a3 = np.array(a1, copy=True)
```

```
In [268...] a1[0][0] = -1
```

```
In [269...] a1, a3
```

```
Out[269...] (array([[ -1,  3],
        [ 4,  5]]),
            array([[0, 3],
        [4, 5]]))
```

然后是order参数:

```
In [270...] # 没有 copy 时, 就是原来的顺序
a1 = np.array([[2, 3], [4, 5]], order="F")
a2 = np.array(a1, order="K", copy=False)
a2.flags
```

```
Out[270...] C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
```

```
In [271...] # copy=True, order=K, 保留最相似的 (F), 向量是CF
a1 = np.array([[2, 3], [4, 5]], order="F")
a2 = np.array(a1, order="K", copy=True)
a2.flags
```

```
Out[271...] C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
```

```
In [272...] # copy=True, order=A, 如果输入是 F 则是 F; 否则 (AKC时) 是 C
a1 = np.array([[2, 3], [4, 5]], order="A")
a2 = np.array(a1, order="A")
a2.flags
```

```
Out[272...] C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
```

读者可以自行验证其他情况。

再看subok参数:

```
In [274... np.array(np.mat('1 2; 3 4'))
```

```
Out[274... array([[1, 2],
        [3, 4]])
```

```
In [275... # 返回原来的类型
np.array(np.mat('1 2; 3 4'), subok=True)
```

```
Out[275... matrix([[1, 2],
        [3, 4]])
```

```
In [276... np.array(np.char.array(['2','3']), subok=True)
```

```
Out[276... chararray([b'2', b'3'], dtype='<S1')
```

```
In [277... # 子类
class A(np.ndarray): pass
```

```
In [278... np.array(A(2), subok=True)
```

```
Out[278... A([9.9e-324, 1.5e-323])
```

然后是ndmin参数:

```
In [279... # 会自动扩充一个维度
np.array([[2, 3], [4, 5]], ndmin=3).shape
```

```
Out[279... (1, 2, 2)
```

```
In [280... # 但如果小于本来的维度, 则不发生变化
np.array([[2, 3], [4, 5]], ndmin=1).shape
```

```
Out[280... (2, 2)
```

最后的like参数演示如下:

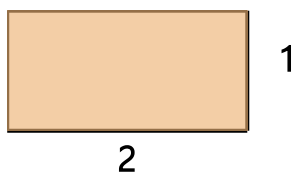
```
In [281... import dask.array as da
```

```
/usr/local/lib/python3.8/site-packages/scipy/__init__.py:138: UserWarning: A NumPy
y version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected ve
rsion 1.23.0)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is requi
red for this version of ")
```

```
In [282... da.array([2, 3])
```

```
Out[282...
```


	Array	Chunk
Bytes	16 B	16 B
Shape	(2,)	(2,)
Count	1 Tasks	1 Chunks
Type	int64	numpy.ndarray



```
In [283...] np.array([2,3], like=da.array([2,3]))
```

```
Out[283...]
```

	Array	Chunk
Bytes	16 B	16 B
Shape	(2,)	(2,)
Count	1 Tasks	1 Chunks
Type	int64	numpy.ndarray



## 自定义数组容器

```
In [284...]
```

```
class MyArray:

    def __init__(self, lst: list):
        self.list = lst

    def __array__(self):
        return np.array(self.list)

class HisArray:

    def __init__(self, lst: list):
        self.list = lst

    def __array__(self):
        return np.array(self.list)
```

```
In [285...]
```

```
a = MyArray([[2, 3], [4, 5]])
```

可以使用 `np.asarray` 或 `np.array` 将其转为 `array`（会调用 `__array__` 方法）：

```
In [286...]
```

```
np.asarray(a)
```

```
Out[286...]
```

```
array([[2, 3],
       [4, 5]])
```

```
In [287...]
```

```
b = HisArray([[2, 3], [4, 5]])
```

```
In [288...]
```

```
np.asarray(b)
```

```
Out[288...]
```

```
array(<__main__.HisArray object at 0x11c007d30>, dtype=object)
```

或者使用NumPy的API进行操作时，也会调用 `__array__`：

```
In [289...]
```

```
np.add(a, 2)
```

```
Out[289...]
```

```
array([[4, 5],
       [6, 7]])
```

```
In [290... np.add(b, 2)
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-290-9e59ee7cfe52> in <module>
----> 1 np.add(b, 2)

TypeError: unsupported operand type(s) for +: 'HisArray' and 'int'
```

可以通过自定义 `__array_function__` 或 `__array_ufunc__` 来自「定义行为」。

假设需要定义一个自定义类加法，可以通过使用 `__add__` 或继承 `numpy.lib.mixins.NDArrayOperatorsMixin` 来实现。

```
In [291... class MyArray:

    def __init__(self, lst: list):
        self.list = lst

    # 自定义 add 方法
    def __add__(self, v):
        return np.array(self.list) + v
```

```
In [292... a = MyArray([[2, 3], [4, 5]])
```

```
In [293... a + 3
```

```
Out[293... array([[5, 6],
                [7, 8]])
```

```
In [294... class MyArray(np.lib.mixins.NDArrayOperatorsMixin):

    def __init__(self, lst: list):
        self.list = lst

    # 继承后用 __array_ufunc__
    def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
        return ufunc(np.array(self.list), inputs[1])
```

```
In [295... a = MyArray([[2, 3], [4, 5]])
```

```
In [297... # 这种操作要继承
a * 3
```

```
Out[297... array([[ 6,  9],
                [12, 15]])
```

```
In [296... # 这个不需要继承 NDArrayOperatorMixin
# 有 __array_ufunc__ 就可以了
np.add(a, 3)
```

```
Out[296... array([[5, 6],
                [7, 8]])
```

```
In [298... class DiagonalArray:

    def __init__(self, N, value):
        self.N = N
```

```

        self.v = value

    def __array_function__(self, func, types, args, kwargs):
        if func == np.sum: return self.N * self.v
        elif func == np.mean: return self.N / self.v

```

In [299... `a = DiagonalArray(5, 2)`

In [300... `# 调用了 np.mean 执行的是我们自己的逻辑`  
`np.mean(a)`

Out[300... 2.5

更多内容可参考：

- <https://numpy.org/doc/stable/user/basics.dispatch.html>
- <https://numpy.org/doc/stable/reference/arrays.classes.html> 第一部分

## 子类化与标准子类

ndarray的新实例可以以三种不同的方式出现：

- 显式构造函数调用
- 视图转换
- 模板创建：最明显的地方是对子类数组进行切片。

后两种是ndarray的特性，子类化ndarray的复杂性是由于NumPy必须支持后两种实例创建路径的机制。

子类化适用以下场合：

- 不担心可维护性或自己以外的用户。
- 子类信息忽略或丢失不是什么问题。

In [536... `# view casting: 获取任何子类的 ndarray, 并将数组的view作为另一个（指定的）子类返回`  
`class C(np.ndarray):`  
 `pass`  
`arr = np.zeros((3,))`  
`c_arr = arr.view(C)`  
`type(c_arr)`

Out[536... `__main__.C`

In [537... `# 通过切片从模板实例创建新实例`  
`v = c_arr[1:]`  
`type(v)`

Out[537... `__main__.C`

In [538... `# 是一个新的实例`  
`v is c_arr`

Out[538... False



ndarray 用于支持子类中的视图和新模板的机制有两个方面。

- 使用 `ndarray.__new__` 方法进行对象初始化的主要工作，而不是更常见的 `__init__` 方法。
- 使用 `__array_finalize__` 方法允许子类在从模板创建视图和新实例之后进行清理。

首先看下初始化的 `__new__` 方法，这样做的原因是在某些情况下，对于 ndarray，我们希望能够返回某个其他类的对象。

```
In [357... class D(C):
    def __new__(cls, *args):
        print('D cls is:', cls)
        print('D args in __new__:', args)
        return C.__new__(C, *args)

    def __init__(self, *args):
        # 当 __new__ 方法返回一个类的对象而不是定义它的类时，该类的 __init__ 方法
        print('In D __init__')
```

```
In [358... d = D((1, ))

D cls is: <class '__main__.D'>
D args in __new__: ((1,),)
```

```
In [359... arr = np.zeros((3, ))
```

```
In [360... d_arr = arr.view(D)
type(d_arr)
```

```
Out[360... __main__.D
```

```
In [361... d_arr
```

```
Out[361... D([0., 0., 0.])
```

```
In [362... v = d_arr[1:]
type(v)
```

```
Out[362... __main__.D
```

这就是 ndarray 类的子类如何能够返回保留类类型的 view (view casting)，当执行 view 时，标准的 ndarray 机制会这样创建新的 ndarray 对象：`obj = ndarray.__new__(subtype, shape, ...)`，`subtype` 就是子类，所以返回的是子类的类，而不是 ndarray 的类。

接下来是 `__array_finalize__`，它允许子类处理创建的新实例的各种方法，签名是 `__array_finalize__(self, obj)`。因为我们不能依赖 `MySubClass.__new__` 或 `MySubClass.__init__` 来处理视图转换和模板创建。

```
In [479... class E(C):
    def __new__(cls, *args, **kwargs):
        print('In __new__ with class %s' % cls)
        return C.__new__(cls, *args, **kwargs)
```

```
def __init__(self, *args, **kwargs):
    # 实际在子类中可能不需要
    print('In __init__ with class %s' % self.__class__)

def __array_finalize__(self, obj):
    print('In array_finalize:')
    print('    self type is %s' % type(self), "self is ", id(self))
    print('    obj type is %s' % type(obj), "obj is ", id(obj))
```

In [480... `e = E((3, ))`

```
In __new__ with class <class '__main__.E'>
In array_finalize:
    self type is <class '__main__.E'> self is 4773873856
    obj type is <class 'NoneType'> obj is 4530430688
In __init__ with class <class '__main__.E'>
```

In [482... `arr = np.ones((3, )) # == obj`  
`e_arr = arr.view(E)`  
`type(e_arr)`

```
In array_finalize:
    self type is <class '__main__.E'> self is 4773875136
    obj type is <class 'numpy.ndarray'> obj is 4765117296
```

Out[482... `__main__.E`

In [483... `v = e[:1]`  
`type(v)`

```
In array_finalize:
    self type is <class '__main__.E'> self is 4773872832
    obj type is <class '__main__.E'> obj is 4773873856
```

Out[483... `__main__.E`

In [435... `v is e_arr`

Out[435... `False`

通过上面的例子可知：

- 从显式构造函数调用时，obj 是 None
- 从视图转换中调用时，obj 可以是 ndarray 的任何子类的实例，包括我们自己的子类
- 在从模板创建中调用时，obj 是我们自己的子类的另一个实例，我们可能会用它来更新新的 self 实例

`__array_finalize__` 是唯一始终看到正在创建的新实例的方法，所以在其他任务中，它是为新对象属性填充实例默认值的最优选择。

In [520... `# 简单示例`

```
class InfoArray(np.ndarray):

    def __new__(subtype, shape, dtype=float, buffer=None, offset=0,
                strides=None, order=None, info=None):
        # 创建自定义类型的 ndarray，调用标准 ndarray 构造器，但返回自定义类型
        # 同时会触发 InfoArray.__array_finalize__
        obj = super().__new__(subtype, shape, dtype,
```

```

        buffer, offset, strides, order)

    obj.info = info
    return obj

def __array_finalize__(self, obj):
    # self 是从 ndarray.__new__(InfoArray, ...) 来的新对象
    # 因此只有 ndarray.__new__ 构造器给的属性

    # 可以通过三种方法调用 ndarray.__new__:
    # 从显式构造函数, 如 InfoArray(): obj 是 None
    if obj is None: return
    # 从视图转换, 如 arr.view(InfoArray): obj 是 arr, type(obj) 是 InfoArray
    # 从模板创建中调用, 如 infoarr[:3]: type(obj) 是 InfoArray
    #     type(obj) is InfoArray
    #
    # ⚠ 注意: 在这里设置 info 的默认值 (不是 __new__ 方法中)
    # 因为这个方法可以看到所有默认对象的创建 (显式构造函数、视图转换、模板创建)
    self.info = getattr(obj, 'info', None)
    # 不需要返回任何东西

```

```

In [509... obj = InfoArray(shape=(3,))
           type(obj)

```

```

Out[509... __main__.InfoArray

```

```

In [510... obj.info is None

```

```

Out[510... True

```

```

In [515... obj = InfoArray(shape=(3,), info='information')
           obj.info

```

```

Out[515... 'information'

```

```

In [516... arr = np.arange(10)
           cast_arr = arr.view(InfoArray) # view casting, arr 没有 info
           type(cast_arr)

```

```

Out[516... __main__.InfoArray

```

```

In [517... cast_arr.info is None

```

```

Out[517... True

```

```

In [518... v = obj[1:] # obj 自己有 info
           type(v)

```

```

Out[518... __main__.InfoArray

```

```

In [519... v.info

```

```

Out[519... 'information'

```

```

In [522... # 更真实的例子
           class RealisticInfoArray(np.ndarray):

               def __new__(cls, input_array, info=None):
                   # 输入 array 已经是 ndarray 实例, 先将其 cast 到自定义 class

```

```

        obj = np.asarray(input_array).view(cls)
        obj.info = info
        return obj

    def __array_finalize__(self, obj):
        if obj is None: return
        self.info = getattr(obj, 'info', None)

```

```

In [523... arr = np.arange(5)
obj = RealisticInfoArray(arr, info='information')
type(obj)

```

```

Out[523... __main__.RealisticInfoArray

```

```

In [524... obj.info

```

```

Out[524... 'information'

```

```

In [525... v = obj[1:]
type(v)

```

```

Out[525... __main__.RealisticInfoArray

```

```

In [526... v.info

```

```

Out[526... 'information'

```

更多可参考：

- <https://numpy.org/doc/stable/user/basics.subclassing.html>

NumPy内置了一些子类，我们这里主要介绍内存映射文件数组，它一般用于读取或修改具有规则布局的大文件的小段，无需将整个文件读入内存。

```

In [527... filename = "data/memmap.dat"
fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))

```

```

In [528... arr = np.arange(12).reshape(3, 4)
fp[:] = arr[:]

```

需要手动 flush 到磁盘（试试不刷会咋样）：

```

In [529... fp.flush()

```

然后就可以读回来了：

```

In [530... newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))

```

```

In [531... newfp

```

```

Out[531... memmap([[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.]], dtype=float32)

```

读取部分使用offset控制的，offset的大小是dtype的大小的整数倍：

```
In [533... # 32位=4个字节
partfp = np.memmap(filename, mode="r", dtype=np.float32, offset=4)
partfp
```

```
Out[533... memmap([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.],
              dtype=float32)
```

```
In [534... partfp = np.memmap(filename, mode="r", dtype=np.float32, offset=2)
partfp
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-534-b8d9cba6aa70> in <module>
----> 1 partfp = np.memmap(filename, mode="r", dtype=np.float32, offset=2)
      2 partfp

/usr/local/lib/python3.8/site-packages/numpy/core/memmap.py in __new__(subtype, filename, dtype, mode, offset, shape, order)
    237         bytes = flen - offset
    238         if bytes % _dbytes:
--> 239             raise ValueError("Size of available data is not a "
    240                             "multiple of the data-type size.")
    241         size = bytes // _dbytes

ValueError: Size of available data is not a multiple of the data-type size.
```

除此之外，还有《数据类型：结构化》一节用到的 `rec` 记录数组、专门用来进行掩码操作的掩码数组等。此处不再赘述，可参考：

- <https://numpy.org/doc/stable/reference/arrays.classes.html>

## 小结

## 参考

- NumPy documentation — NumPy v1.23.dev0 Manual
- What Is Little-Endian And Big-Endian Byte Ordering? | Engineering Education (EngEd) Program | Section
- Understanding Big and Little Endian Byte Order – BetterExplained

```
In [ ]:
```