《源码探秘 CPython》16. bytes 对象的行为（上）

原创 古明地觉 古明地觉的编程教室 2022-01-24 09:30

楔子

介绍完bytes对象在底层的数据结构之后，我们来考察bytes对象的**行为**。我们说实例对象的行为由其类型对象决定，所以bytes对象具有哪些行为，就看bytes类型中定义了哪些操作。bytes类型，显然对应**PyBytes_Type**，根据我们之前介绍的规律，也可以猜出来，它定义在Object/bytesobject.c中。

```
PyTypeObject PyBytes_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "bytes",
    PyBytesObject_SIZE,
    sizeof(char),
    // ...
    &bytes_as_number,                       /* tp_as_number */
    &bytes_as_sequence,                     /* tp_as_sequence */
    &bytes_as_mapping,                      /* tp_as_mapping */
    (hashfunc)bytes_hash,                   /* tp_hash */
    // ...
};
```

到了现在，相信你对类型对象的结构肯定非常熟悉了，因为类型对象都是由PyTypeObject结构体实例化得到的。

我们看到tp_as_number，它居然不是0，而是传递了一个指针，说明确实指向了一个PyNumberMethods结构体实例。难道bytes支持数值运算，这显然是不可能的啊，所以我们需要进入bytes_as_number中一探究竟。

```
static PyNumberMethods bytes_as_number = {
    0,              /*nb_add*/
    0,              /*nb_subtract*/
    0,              /*nb_multiply*/
    bytes_mod,      /*nb_remainder*/
}
```

我们看到它只定义了一个取模操作，也就是%，而看到%估计有人已经明白了，这是格式化操作啊。

由此可见，bytes对象只是借用了**%运算符**实现了格式化，谈不上数值运算。不过由此也看到了Python的动态特性，即使是相同的操作，但如果是不同类型的对象执行的话，也会有不同的表现。

```
>>> info = b"name: %s, age: %d"
>>> info % (b"satori", 16)
b'name: satori, age: 16'
>>>
```

但除了tp_as_number，PyBytes_Type还给tp_as_sequence成员传递了bytes_as_sequence指针，说明bytes对象支持序列操作。这是肯定的，因为bytes对象显然是序列型对象，所以序列型操作才是我们研究的重点，下面看看bytes_as_sequence的定义。

```
static PySequenceMethods bytes_as_sequence = {
    (lenfunc)bytes_length, /*sq_length*/
    (binaryfunc)bytes_concat, /*sq_concat*/
```

```
    (ssizeargfunc)bytes_repeat, /*sq_repeat*/
    (ssizeargfunc)bytes_item, /*sq_item*/
    0,                  /*sq_slice*/
    0,                  /*sq_ass_item*/
    0,                  /*sq_ass_slice*/
    (objobjproc)bytes_contains /*sq_contains*/
}
```

根据定义我们看到，bytes对象支持的**序列型操作**一共有5个：

- sq_length：查看序列的长度
- sq_concat：将两个序列合并为一个
- sq_repeat：将序列重复多次
- sq_item：根据索引获取指定位置的字节，返回的是一个整数
- sq_contains：判断某个序列是不是该序列的子序列，对应 Python 中的 in 操作符

操作的底层实现

下面我们进入源码中考察一下，看看具体是如何实现的。

查看序列长度：

显然这是最简单的，直接获取ob_size即可，比如：val = b"abcde"，那么长度就是5。

```
static Py_ssize_t
bytes_length(PyBytesObject *a)
{
    return Py_SIZE(a);
}
```

Py_SIZE是一个宏，会获取对象内部的 ob_size。

将两个序列合并为一个：

```
>>> a = b"abc"
>>> b = b"def"
>>> a + b
b'abcdef'
>>>
```

我们看到这里相当于是加法运算，所以很容易联想到PyNumberMethods中的nb_add，比如：PyLongObject对应的long_add、PyFloatObject对应的float_add。

但对于bytes对象而言却不是这样，加法操作对应的是PySequenceMethods的**sq_concat**。所以我们看到Python的同一个操作符，在底层会对应不同的函数，比如：long_add和float_add、以及这里的bytes_concat，在Python的层面都是+这个操作符。

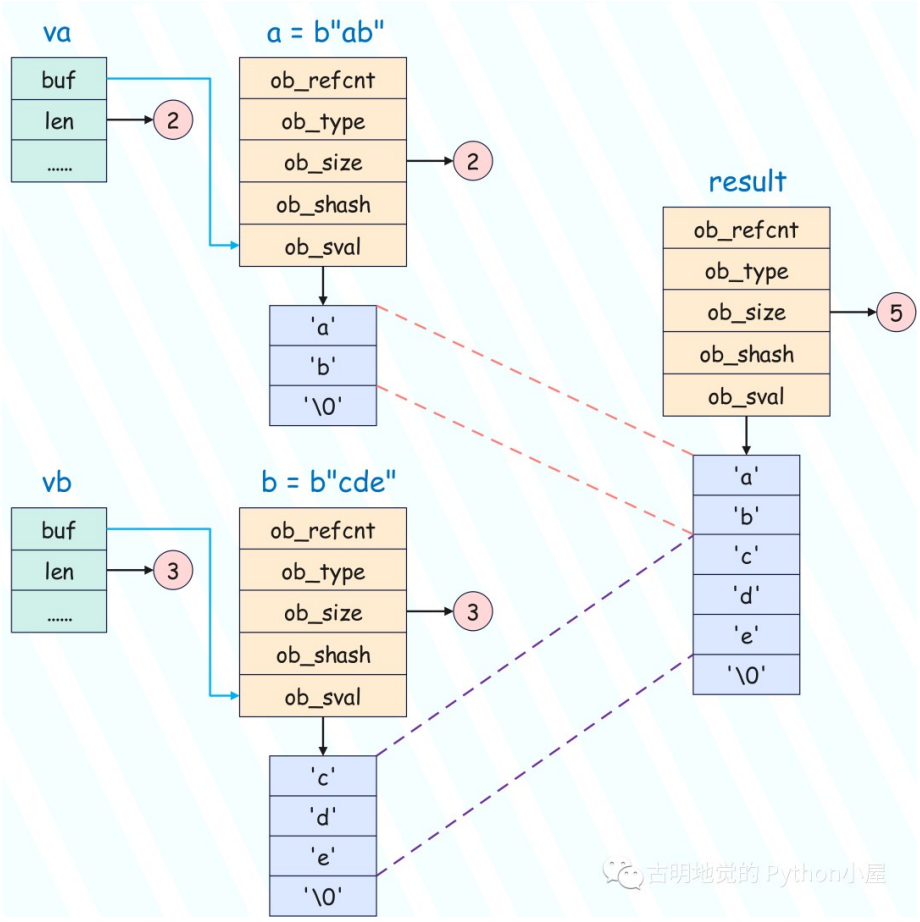
然后我们看看底层是怎么对两个字节序列进行相加的。

```
static PyObject *
bytes_concat(PyObject *a, PyObject *b)
{
    //两个局部变量，用于维护缓冲区
    //关于缓冲区，我们一会儿说
    Py_buffer va, vb;
    //result用于保存结果
    PyObject *result = NULL;

    //将缓冲区的长度设置为-1
```

```
    //可以认为此时缓冲区啥也没有
    va.len = -1;
    vb.len = -1;

    //每个bytes对象底层都对应一个缓冲区
    //可以通过PyObject_GetBuffer获取
    //因此这里是获取a、b的缓冲区，交给 va、vb
    //获取成功返回0，获取失败返回非0
    //如果下面的条件不成功，就意味着拷贝失败了
    //说明至少有一个老铁不是bytes类型
    if (PyObject_GetBuffer(a, &va, PyBUF_SIMPLE) != 0 ||
        PyObject_GetBuffer(b, &vb, PyBUF_SIMPLE) != 0) {
        //然后设置异常，PyExc_TypeError表示TypeError(类型错误)
        //专门用来指对一个对象执行了它所不支持的操作
        PyErr_Format(PyExc_TypeError, "can't concat %.100s to %.100s",
                     Py_TYPE(b)->tp_name, Py_TYPE(a)->tp_name);
        //比如："123" + 123 就会得到如下异常：
        //TypeError can't concat int to bytes
        //和这里设置的异常信息是一样的
        //出现异常之后，直接跳转到done这个标签
        goto done;
    }

    //这里是判断是否有一方长度为0
    //如果a长度为0，那么相加之后结果就是b
    if (va.len == 0 && PyBytes_CheckExact(b)) {
        //将b拷贝给result
        result = b;
        //增加result的引用计数
        Py_INCREF(result);
        //跳转
        goto done;
    }

    //和上面同理，如果b长度为0，那么相加之后的结果就是a
    if (vb.len == 0 && PyBytes_CheckExact(a)) {
        //将a拷贝给result
        result = a;
        //增加引用计数
        Py_INCREF(result);
        //跳转
        goto done;
    }

    //这里是判断两个字节序列合并之后，长度是否超过限制
    //因为不允许超过PY_SSIZE_T_MAX
    //所以Python的bytes对象是有长度限制的
    //但是这个条件基本不可能满足，除非你写恶意代码
    if (va.len > PY_SSIZE_T_MAX - vb.len) {
    //补充一句，这个if条件更直观的写法应该是
    //if (va.len + vb.len > PY_SSIZE_T_MAX)
    //但是va.len + vb.len可能会溢出
        PyErr_NoMemory();
        goto done;
    }

    //否则的话，声明指定容量PyBytesObject
    //这里直接调用了Python/C API
    result = PyBytes_FromStringAndSize(NULL, va.len + vb.len);
    if (result != NULL) {
        //将缓冲区va里面内容拷贝到result的ob_sval中
        //拷贝的长度为va.len
        //PyBytes_AS_STRING是一个宏，用于获取PyBytesObject中的ob_sval
        memcpy(PyBytes_AS_STRING(result), va.buf, va.len);
```

```
75          //然后将缓冲区vb里面的内容拷贝到result的ob_sval中
76          //拷贝的长度为vb.len
77          //但是从va.len的位置开始拷贝，不然会把之前的内容覆盖掉
78          memcpy(PyBytes_AS_STRING(result) + va.len, vb.buf, vb.len);
79      }
80
81    done:
82      //如果长度不会-1，那么要将缓冲区里面的内容释放掉
83      //否则可能导致内存泄漏
84      if (va.len != -1)
85          PyBuffer_Release(&va);
86      if (vb.len != -1)
87          PyBuffer_Release(&vb);
88      //返回result
89      return result;
90  }
```

代码虽然有点长，但是不难理解。假设**a=b"ab"**、**b=b"cde"**，我们以**a+b**为例来描述一下上面的过程：



如果简单点理解的话，可以认为整个过程就是将**a -> ob_sval**和**b -> ob_sval**拷贝到**result->ob_sval**中。

但问题是这里面出现了一个东西叫**Py_Buffer**，底层在拷贝的时候并不是直接拷贝的，而是借助了**Py_Buffer**。之所以这么做，是因为**Py_Buffer**提供了一套操作对象缓冲区的统一接口，屏蔽不同类型对象的差异。

为了更好地理解缓冲区，我们拓展一下，来解释一下什么是缓冲区协议。缓冲区协议是一个 C 级协议，它定义了一个具有数据缓冲区和元数据的 C 级结构体，这个结构体就是上面的**Py_Buffer**。并用它来描述缓冲区的布局、数据类型和读写权限，并且还定义了支持协议的对象所必须实现的 API。

实现缓冲区协议的对象有 bytes对象、标准库array中的array对象、以及最知名的 numpy.ndarray 对象。

至于缓冲区本身，它就是一个单纯的一维数组，负责存储具体的数据。我们以numpy的数组为例，不管这个数组是多少维的，底层的缓冲区永远是一个一维数组。那么问题来了，我们在定义数组时设置的维度信息要如何体现呢？答案是通过**Py_Buffer**，我们看一下它的底层结构，位于Include/cpython/object.h中。

```
typedef struct bufferinfo {
    //指针，指向具体的的缓冲区
    //注意:指向的永远是一个一维数组
    void *buf;
    //实现缓冲区协议的对象本身
    PyObject *obj;
    //缓冲区的大小
    Py_ssize_t len;
    //缓冲区中每个元素的大小
    Py_ssize_t itemsize;
    //缓冲区是否只读，0表示可读写、1表示只读
    int readonly;
    //维度，比如shape为(3， 4， 5)的数组
    //那么底层的ndim就是3
    int ndim;
    //格式化字符，用于描述缓冲区的元素类型
    char *format;
    //等价于numpy中数组的shape
    //因此缓冲区永远是一个一维数组，由buf成员指向
    //而其它成员则负责描述这个一维数组应该要怎么使用
    Py_ssize_t *shape;
    //在某个维度下，从一个元素到下一个元素所需要跳跃的字节数
    Py_ssize_t *strides;
    Py_ssize_t *suboffsets;
    void *internal;
} Py_buffer;
```

以上就是**Py_Buffer**，**Py_Buffer**内部的buf成员指向了具体的缓冲区，对于bytes对象而言就是**ob_sval**。如果是numpy的数组，那么默认情况下数组在拷贝的时候只会将**Py_Buffer**拷贝一份，**Py_Buffer**内部的buf成员指向的缓冲区则不会拷贝。

```
import numpy as np

#Py_Buffer -> buf 指向了缓冲区
#Py_Buffer -> shape 为 (6,)
arr1 = np.array([3, 9, 5, 7, 6, 8])
#将 Py_Buffer 拷贝一份
#同时 Py_Buffer -> shape 变成了 (2, 3)
#但是 Py_Buffer -> buf 指向的缓冲区没有拷贝
arr2 = arr1.reshape((2, 3))

#然后在通过索引访问的时候
#可以认为numpy为其创建了虚拟的索引轴
#由于 arr1 只有一个维度
#那么numpy会为其创建一个虚拟的索引轴
"""
arr1 = [3 9 5 7 6 8]:

    index1: 0 1 2 3 4 5
       buf: 3 9 5 7 6 8
"""
#arr2 有两个维度，shape 是 (2, 3)
#那么numpy会为其创建两个虚拟的索引轴
"""
arr2 = [[3 9 5]
        [7 6 8]]:
    index1: 0 0 0 1 1 1
    index2: 0 1 2 0 1 2
```

```
        buf: 3 9 5 7 6 8
"""
#缓冲区中索引为 4 的元素被修改
arr2[1, 1] = 666
#但由于 arr1 和 arr2 共享一个缓冲区
#所以 print(arr1[4]) 也会打印 666
print(arr1[4])  # 666
```

以上就是缓冲区相关的内容，回到**bytes对象**，可以理解为里面的**ob_sval**就是对应的缓冲区，它是一个一维数组。然后它也实现了缓冲区协议，**Py_Buffer**里面的buf成员同样指向了这个缓冲区，而其它的成员则负责描述该如何使用这个缓冲区，可以理解为元信息。

正如numpy的数组，虽然多个数组底层共用一个缓冲区，数据也只有那一份，但是在numpy的层面却可以表现出不同的维度，究其原因就是元信息不同。

> Py_Buffer的实现，也是numpy诞生的一个重要原因。
>
> 另外，类型对象内部有一个tp_as_buffer成员，它是一个函数指针，在函数内部负责对Py_Buffer进行初始化。如果实现了该成员，那么其实例对象便支持缓冲区协议。并且实现了缓冲区协议的对象，不会直接操作缓冲区，而是会借助于 Py_Buffer。

相信你现在肯定明白**Py_Buffer**存在的意义了，就是共享内存，实现了缓冲区协议的对象可以直接向彼此保留对应的缓冲区，比如**bytes对象**和**ndarray对象**。

```
import numpy as np

#缓冲区是char类型的一维数组: {'a', 'b', 'c', 'd', '\0'}
b = b"abcd"

#直接共享底层的缓冲区
#但是numpy不知道如何使用这个缓冲区
#所以我们必须显式地指定 dtype
#"S1" 表示按照单个字节来进行解析
arr1 = np.frombuffer(b, dtype="S1")
print(arr1)  # [b'a' b'b' b'c' b'd']

#"S2" 表示按照两个字节来进行解析
arr2 = np.frombuffer(b, dtype="S2")
print(arr2)  # [b'ab' b'cd']

#那么问题来了，按照三个字节解析是否可行呢？
#答案是不可行，缓冲区的大小不是3的整数倍
#而 "S4" 显然是可以的
arr3 = np.frombuffer(b, dtype="S4")
print(arr3)  # [b'abcd']

#按照 int8 进行解析
arr4 = np.frombuffer(b, dtype="int8")
print(arr4)  # [ 97  98  99 100]

#按照 int16 进行解析
#显然 97 98 会被解析成一个整数
#99 100 会被解析成一个整数
#你想到了什么，这不就类似于Python整数的底层实现嘛
"""
97 -> 01100001
98 -> 01100010
那么 97 98 组合起来就是 01100010_01100001

99 -> 01100011
100 -> 01100100
那么 97 98 组合起来就是 01100100_01100011
"""
```

```
print(0b01100010_01100001)  # 25185
print(0b01100100_01100011)  # 25699
print(
    np.frombuffer(b, dtype="int16")
)  #[25185 25699]

#按照int32来解析，显然这个4个int8表示一个int32
print(
    0b01100100_01100011_01100010_01100001
)  #1684234849
print(np.frombuffer(b, dtype="int32"))  # [1684234849]
```

怎么样，是不是有点神奇呢？相信你在使用numpy的时候应该会有更加深刻的认识了，这就是缓冲区协议的威力。哪怕是不同的对象，只要都实现了缓冲区协议，那么彼此之间就可以暴露底层的缓冲区，从而实现共享内存。

所以np.frombuffer就是直接根据对象的缓冲区来创建数组，然后它底层的buf成员也指向这个缓冲区。但它不知道该如何解析这个缓冲区，所以我们需要显式地指定dtype来告诉它，相当于告诉它一些元信息。

那么问题来了，我们能不能修改缓冲区呢？

```
import numpy as np

b = b"abcd"
arr = np.frombuffer(b, dtype="S1")

try:
    arr[0] = b'A'
except ValueError as e:
    print(e)  # assignment destination is read-only

#答案是不可以的，因为原始的 bytes 对象不可修改
#所以缓冲区只读的
#但我们真的就没办法了吗？还记得之前我们介绍的骚操作吗？
from ctypes import *

class PyBytesObject(Structure):
    _fields_ = [
        ("ob_refcnt", c_ssize_t),
        ("ob_type", c_void_p),
        ("ob_size", c_ssize_t),
        ("ob_shash", c_ssize_t),
        ("ob_sval", 5 * c_byte),
    ]

obj = PyBytesObject.from_address(id(b))
#修改缓冲区之前，打印 arr
print(arr)  # [b'a' b'b' b'c' b'd']
#修改缓冲区之后，打印 arr
obj.ob_sval[0] = ord('A')
print(arr)  # [b'A' b'b' b'c' b'd']
```

我们看到由于共享缓冲区，所以修改 bytes 对象也会影响数组 arr。

由于 bytes 对象不可变，我们只能出此下策，但其实我们还有一个办法，就是使用 bytearray 对象。

```
import numpy as np

#可以理解为可变的 bytes 对象
b = bytearray(b"abcd")
print(b)  # bytearray(b'abcd')
```

```
#修改 arr
arr = np.frombuffer(b, dtype="S1")
arr[0] = b'A'
#再次打印
print(b)  # bytearray(b'Abcd')
```

小结

bytes 对象支持的操作还没有结束，我准备分两次介绍。主要是有小伙伴建议能给一些消化时间，但我还是想坚持每天更新一篇文章，所以综合一下，如果内容过长的话就拆分成**上中下**。

本来这部分内容其实并不多，但我这个人比较贪心，希望自己的每一篇文章能给人带来帮助。所以除了内容本身之外，其背后涉及的一些细节，我也会不断地进行展开。比如从这里的两个**bytes对象**相加，我们介绍了什么是缓冲区、缓冲区协议，以及存在的作用，并且通过 numpy 进行了解释。了解缓冲区，可以让你更加深刻地理解numpy。

下面再来总结一下：

- 如果一个类型对象实现了tp_as_buffer，那么它的实例对象便支持缓冲区协议。
- tp_as_buffer 是一个函数指针，指向的函数内部负责初始化Py_Buffer。
- 在共享缓冲区的时候，比如**np.frombuffer(obj)**，会直接调用obj的类型对象的tp_as_buffer成员指向的函数，拿到Py_Buffer实例的buf成员指向的缓冲区。但我们说numpy不知道该怎么解析这个缓冲区，所以还需要我们指定dtype参数。
- 缓冲区存在的最大意义就是共享内存，numpy的数组在拷贝的时候，默认只拷贝 Py_Buffer 实例，至于Py_Buffer里面buf成员指向的缓冲区默认是不会拷贝的。比如数组有100万个元素，这些元素都存在缓冲区中，被Py_Buffer里面的buf成员指向，拷贝的时候这100万个元素是不会拷贝的。
- numpy数组的维度、shape，是借助于Py_Buffer中的元信息体现的，至于存储元素的缓冲区，永远是一个一维数组，由 buf 成员指向。维度、shape不同，访问缓冲区元素的方式也不同。但还是那句话，缓冲区本身很单纯，就是一个一维数组。

虽然标题是bytes对象的行为，但正文大部分都是在介绍缓冲区，不过我觉得这是值得的。毕竟理解了缓冲区以及缓冲区协议，我们能更好地理解numpy。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》17. bytes 对象的行为（下）

下一篇 ›
《源码探秘 CPython》15. bytes 对象是怎么实现的?