《源码探秘 CPython》15. bytes 对象是怎么实现的?

原创 古明地觉 古明地觉的编程教室 2022-01-21 09:30

收录于合集
#CPython
97个 >

楔子

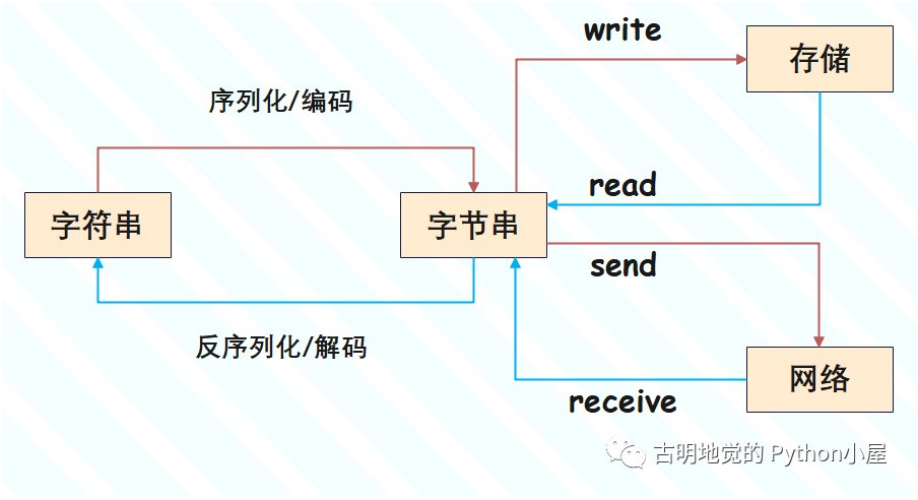
我们知道在C语言里面，有一个概念叫**字符数组**，比如：

```
1  char name[] = "komeiji satori";
```

一个字节最多能表示**256**个字符，所以对于英文来说足够了，因此一个英文字符占一个字节即可，然而对于那些非英文字符便力不从心了。而为了表示这些非英文字符，于是多字节编码应运而生，即：通过多个字节来表示一个字符。但由于原始字节序列不维护编码信息，因此操作不慎便导致各种乱码现象。

而Python提供的解决方案是使用**unicode**(在Python3中等价于**str**)来表示字符串，因为unicode可以表示各种字符，不需要关心编码的问题，我们后面会详细解析字符串。

但在存储或网络通讯时，传输的都是二进制，字符串不可避免地要序列化成字节序列。为此，Python除了提供字符串之外，还额外提供了字节序列（字节串），也就是 bytes 对象。



如上图，**str对象**统一表示一个字符串，不需要关心编码，可以表示世界上所有的字符；但计算机是通过字节序列来和存储介质、网络介质打交道，所以在存储和传输str对象的时候，需要将其序列化成字节序列（bytes对象），序列化也是编码的过程。

既然有序列化，那么就有**反序列化**，很明显反序列化是将bytes对象转成str对象，也被称为**解码**。

下面我们就来看看 bytes 对象的底层结构。

bytes 对象的底层结构

我们说bytes对象是一个字节序列、或者字节串，那么显然它是由若干个字节组成的，也就意味着它是一个变长对象。字节序列内部的字节数，就是其长度。

```
1  //Include/bytesobject.h
2  typedef struct {
3      PyObject_VAR_HEAD
```

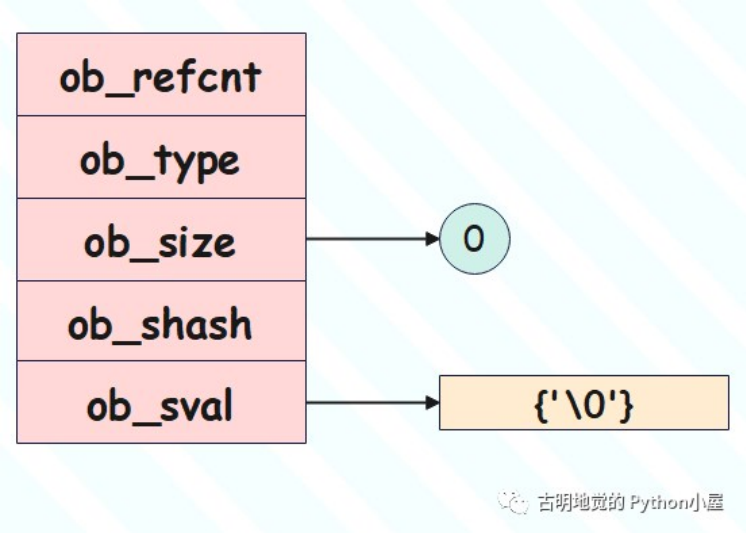
```
    Py_hash_t ob_shash;
    char ob_sval[1];
} PyBytesObject;
```

我们看一下里面的成员对象：

- PyObject_VAR_HEAD：变长对象的公共头部；
- ob_shash：保存该字节序列的哈希值，之所以选择提前保存是因为在很多场景都需要bytes对象的哈希值。而Python在计算字节序列的哈希值的时候，需要遍历每一个字节，因此开销比较大。所以会提前计算一次并保存起来，这样以后就不需要算了，可以直接拿来用，并且bytes对象是不可变的，所以哈希值是不变的；
- ob_sval：这个和PyLongObject中ob_digit的声明方式是类似的，虽然声明的时候长度是1，但具体是多少则取决于bytes对象的字节数量。这是C语言中定义"变长数组"的技巧，虽然写的长度是1，但是你可以当成n来用，n可取任意值。显然这个ob_sval存储的是所有的字节，所以Python中bytes对象的值，底层是通过字符数组存储的。而且会多申请一个空间，用于存储\0，因为C中是通过\0来表示一个字符数组的结束，但是计算ob_size的时候不包括\0；

我们创建几个不同的**bytes对象**，然后通过画图感受一下：

val = b""



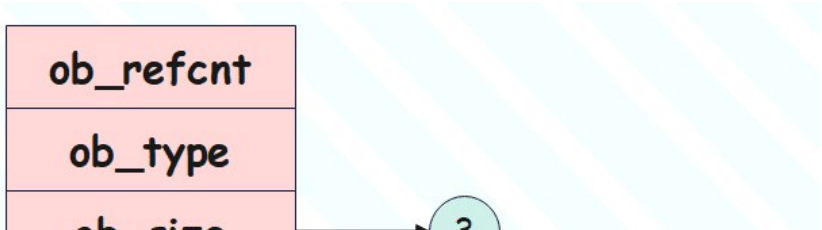
我们看到一个即便是空的字节序列，底层的**ob_savl**也是需要一个**'\0'**的，那么这个结构体实例占多大内存呢？除了**ob_sval**之外的四个成员，显然每个都是8字节，而ob_savl是一个char类型的数组，一个char占1字节，所以Python中bytes对象占的内存等于**32 + ob_sval的长度**。

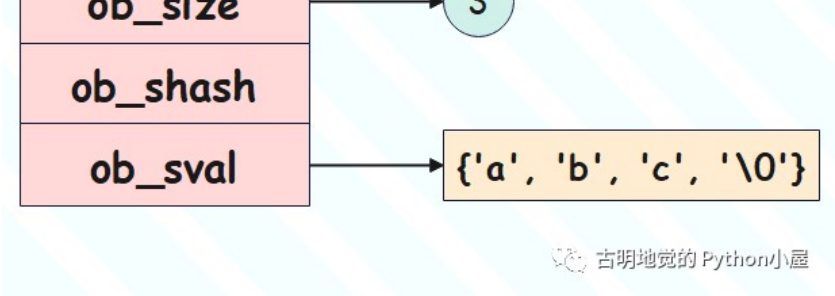
而ob_sval里面至少有一个'\0'，因此对于一个空的字节序列，显然占33个字节。

> 注意：ob_size统计的是ob_sval中有效字节的个数，不包括'\0'，但是计算占用内存的时候，显然是需要考虑在内的，因为它确实多占用了一个字节的空间。所以说bytes对象占的内存等于33 + ob_size也是可以的。

```
>>> val = b""
>>> sys.getsizeof(val)
33
>>>
```

val = b"abc"





显然长度等于**32+4=36**字节。

所以 bytes 对象的底层结构还是很好理解的，因为它是字节序列，所以在底层用一个char类型的数组来维护具体的值再合适不过了。

创建 bytes 对象

下面我们来看一下 bytes 对象的创建方式，这里我们暂时先不介绍底层是如何创建的，等到介绍缓存池的时候再说。我们来聊一聊如何在Python中创建，虽然我们这里是探秘CPython，但是光说底层的话可能会有一些无趣，因此这个过程中也会穿插大量的Python内容。

```
b1 = b"hello"
```

以上是最简单的创建方式了，采用我们之前说的**Python/C API**创建，但这种创建方式只使用于ASCII字符。下面这种方式是不行的：

```
b = b"古明地觉"
```

"古明地觉"包含非ASCII字符，所以采用多字节编码（关于字符编码、字符集等概念在介绍字符串的时候会详细说），但编码方式也有多种，比如utf-8、gbk等等，Python不知道你用的是哪一种。因此采用字面量的方式，只能使用ASCII字符串，如果使用非ASCII字符串，那么必须手动指定编码。

```
b = bytes("古明地觉", encoding="utf-8")
print(b)  # b'\xe5\x8f\xa4\xe6\x98\x8e\xe5\x9c\xb0\xe8\xa7\x89'
```

里面的 \x 表示16进制，我们知道**字符a**的ASCII码是97，对应16进制是61，同理**字符b**是62，**字符c**是63，那么 b"abc" 就还可以这么创建：

```
b = b"\x61\x62\x63"
print(b)  # b'abc'
```

以上是根据16进制的数字创建bytes对象，注意：采用字面量的方式创建必须指定\x，**b"\x61"**表示的是1个字节，并且该字节对应的ASCII码的16进制等于61，也就是字符a；而**b"61"**表示的是两个字节。

```
# \x61、\x62、\x63均表示1字节
print(b"\x61\x62\x63")  # b'abc'
# 下面这种创建的bytes对象是6字节
print(b"616263")  # b'616263'
```

可如果我有一串字符也是16进制格式，但开头没有\x，这个时候我要怎么转成bytes对象呢？很简单，使用**bytes.fromhex**方法即可。

```
print(bytes.fromhex("616263"))  # b'abc'
# 转成bytes对象之后，如果能用ASCII字符显示的话
# 那么就用ASCII字符显示，比如 abc
# 不能的话，就原本输出，比如\xff
print(bytes.fromhex("616263FF"))  # b'abc\xff'
```

该方法会将里面字符串当成16进制来解析，得到bytes对象。并且使用这种方式的话，字符的个数一定是偶数，每个字符的范围均是**0~9、A-F(或者a-f)**。因为16进制需要两个字符来表示，范围是**00**到**FF**。即便小于16，也必须用两个字符表示，比如我们可以写成**05**，但绝不能只写个**5**。

总之使用bytes.fromhex 创建时，字符串的长度一定是一个偶数，从前往后每两个分为一组。字面量的方式创建时也是如此，比如我们可以写成**b"\x01\x02"**，但绝不能写成**b"\x1\x2"**。

```
# 不可以写成 b"\x0", 会报错
b1 = b"\x00"
print(b1)  # b'\x00'

# \x后面至少跟两个字符, 但这里跟了3个字符
# 所以 \x 会和 61 结合, 形成 'a'
# 至于后面的那个 1 就单纯的表示字符 '1'
b2 = b"\x611"
print(b2)  # b'a1'
```

所以\x后面可以跟超过两个以上的字符，超过两个以上的部分会当成普通字符来处理，与十六进制无关，每个\x只和它后面两个字符结合；但 \x 后面不能少于两个字符。

问题又来了，如果我有一串整数，是十进制的，这个时候怎么创建呢？

```
# 里面的每个数值范围均是 0~255
b1 = bytes([97, 98, 99])
print(b1)  # b'abc'
```

这种创建方式也是很方便的，总之 bytes 对象的创建方式有多种，相信还是有部分小伙伴没有仔细观察打印bytes对象时输出的内容。核心就在于bytes对象本质上是字节序列，你看到的\x表示的是：该字节是通过\x加上16进制的ASCII码来显示的。

然后我们通过索引获取的时候，得到也是一个整数：

```
b = "古".encode("utf-8")
print(b)  # b'\xe5\x8f\xa4'

lst = [b[0], b[1], b[2]]
print(lst)  # [229, 143, 164]

print(bytes(lst).decode("utf-8"))  # 古
```

小结

以上就是 bytes 对象底层结构，还是比较简单的，就是用一个 char 类型的数组来存储具体的值。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》16. bytes 对象的行为（上）

下一篇 ›
《源码探秘 CPython》14. 整数在底层是如何运算的?

python-字符串编码问题怎么破

一位代码

python 7天进阶之路-对象和json转换

缪斯之子