《源码探秘 CPython》21. 字符串的底层结构

原创 古明地觉 古明地觉的编程教室 2022-02-05 09:30

楔子

我们之前提到了，字符串采用不同的编码，底层的结构体实例所占用的额外内存是不一样的。其实本质上是，字符串会根据编码的不同，而选择不同的存储单元。

至于到底是怎么做到的，我们只能去源码中寻找答案了，与str相关的源码：**Include/unicodeobject.h**和**Objects/unicodeobject.c**

```
enum PyUnicode_Kind {
    PyUnicode_WCHAR_KIND = 0,
    PyUnicode_1BYTE_KIND = 1,
    PyUnicode_2BYTE_KIND = 2,
    PyUnicode_4BYTE_KIND = 4
};
```

我们在unicodeobject.h中看到，unicode会根据编码的不同而分为以下几类：

- PyUnicode_1BYTE_KIND：所有字符码位均在 U+0000 到 U+00FF 之间
- PyUnicode_2BYTE_KIND：所有字符码位均在 U+0000 到 U+FFFF 之间，且至少一个大于 U+00FF(否则每个字符就用1字节了)
- PyUnicode_4BYTE_KIND：所有字符码位均在 U+0000 到 U+10FFFF 之间，且至少一个大于 U+FFFF

如果文本字符码位均在 U+0000 到 U+00FF 之间，单个字符只需 1 字节来表示；而码位在 U+0000 到 U+FFFF 之间的文本，单个字符则需要 2 字节才能表示；以此类推。这样一来，根据文本码位范围，便可为字符选用尽量小的存储单元，以最大限度节约内存。

```
//我们看到4字节使用的是无符号32位整型
typedef uint32_t Py_UCS4;
//2字节是无符号16位整型
typedef uint16_t Py_UCS2;
//Latin-1是uint8
typedef uint8_t Py_UCS1;
```

既然unicode内部的存储结构会因字符而异，那么unicode底层就必须有成员来维护相应的信息，所以Python内部定义了若干标志位：

- interned：是否被intern机制维护，这个机制我们会在后面介绍
- kind：类型，用于区分底层存储单元的大小。如果是Latin1编码,那么就是1；UCS2编码则是2；UCS4编码则是4
- compact：内存分配方式，对象与文本缓冲区是否分离
- ascii：字符串是否是纯ASCII字符串，如果是则为1，否则为0。注意：只有对应的ASCII码为0~127之间的才是ASCII字符。所以虽然一个字节可表示的范围是0~255，但是128~255之间的并不是ASCII字符

而为unicode字符串申请空间，底层可以调用一个叫**PyUnicode_New**的函数，这也是一个特型API。比如：元组申请空间可以使用**PyTuple_New**，列表申请空间可以使用**PyList_New**等等，会传入一个整型，创建一个能够容纳指定数量元素的结构体实例。而**PyUnicode_New**则接收一个字符个数参数、以及最大字符maxchar来初始化unicode字符串对象，之所以会多出一个maxchar，是因为要根据它来为unicode字符串对象选择最紧凑的字符存储单元，以及结构体。

	maxchar < 128	maxchar < 256	maxchar < 65536	maxchar < MAX_UNICODE
kind	PyUnicode_1BYTE_KIND	PyUnicode_1BYTE_KIND	PyUnicode_2BYTE_KIND	PyUnicode_4BYTE_KIND
ascii	1	0	0	0
字符存储单元的大小	1	1	2	4
底层结构体	PyASCIIObject	PyCompactUnicodeObject	PyCompactUnicodeObject	PyCompactUnicodeObject

下面我们就来分析字符串底层对应的结构体。

PyASCIIObject

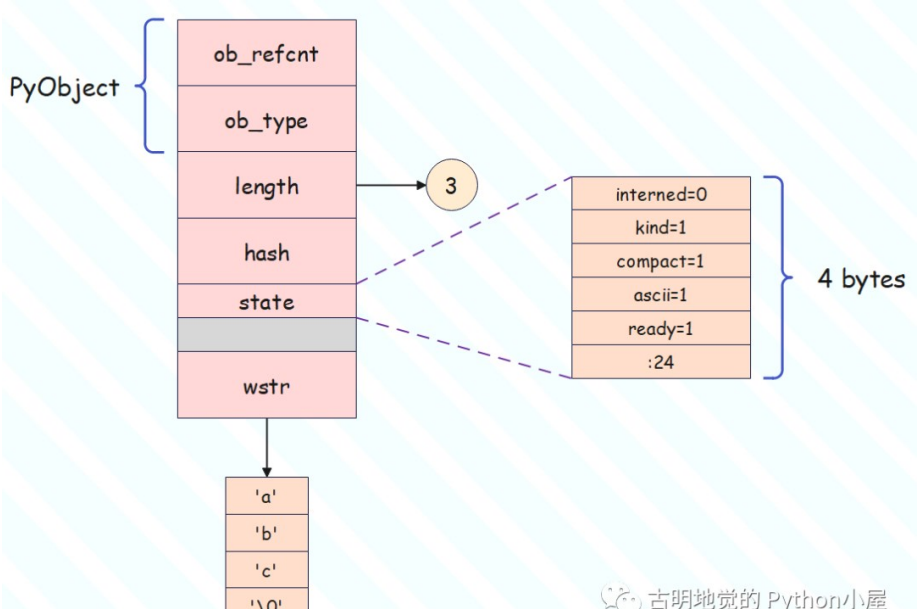
如果字符串保存的文本均为 ASCII ，即 maxchar<128，则底层由 PyASCIIObject 结构进行存储：

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t length;
    Py_hash_t hash;
    struct {
        unsigned int interned:2;
        unsigned int kind:3;
        unsigned int compact:1;
        unsigned int ascii:1;
        unsigned int ready:1;
        unsigned int :24;
    } state;
    wchar_t *wstr;
} PyASCIIObject;
```

PyASCIIObject结构体也是其他Unicode结构体的基础，所有字段均为 Unicode 公共字段：

- ob_refcnt：引用计数
- ob_type：类型指针
- length：字符串长度
- hash：字符串的哈希值
- state：unicode对象标志位，包括intern、kind、ascii、compact等，其含义就是我们上面介绍的那样
- wstr：一个指针，指向由宽字符组成的字符数组。字符串和字节序列一样，底层都是通过字符数组来维护具体的值

以字符串**abc**为例，看看它在底层的存储结构：



注意：state 成员后面有一个**4字节的空洞**，这是结构体字段内存对齐造成的现象。在 64 位机器上，指针大小为 8 字节，为优化内存访问效率，必须以 8 字节对齐。

那么现在我们知道一个空字符串为什么占据49个字节了，因为ob_refcnt、ob_type、length、hash、wstr 都是 8 字节，所以总共 40 字节；而 state 是 4 字节，但是留下了 4 字节的空洞，加起来也是 8 字节，所以总共占 40 + 8 = 48 个字节，但是 Python 的 unicode 字符串在 C 中也是使用字符数组来存储的，只不过此时的字符不再是 char 类型，而是 wchar_t。但是它的内部依旧有一个 '\0'，所以还要加上一个 1，总共 49 字节。

对于**abc**这个unicode字符串来说，占的总字节数就是**49+3=52**。

```
import sys
print(sys.getsizeof("abc"))  # 52

# 长度为n的ASCII字符串，大小就是49 + n
print(sys.getsizeof("a" * 1000))  # 1049
```

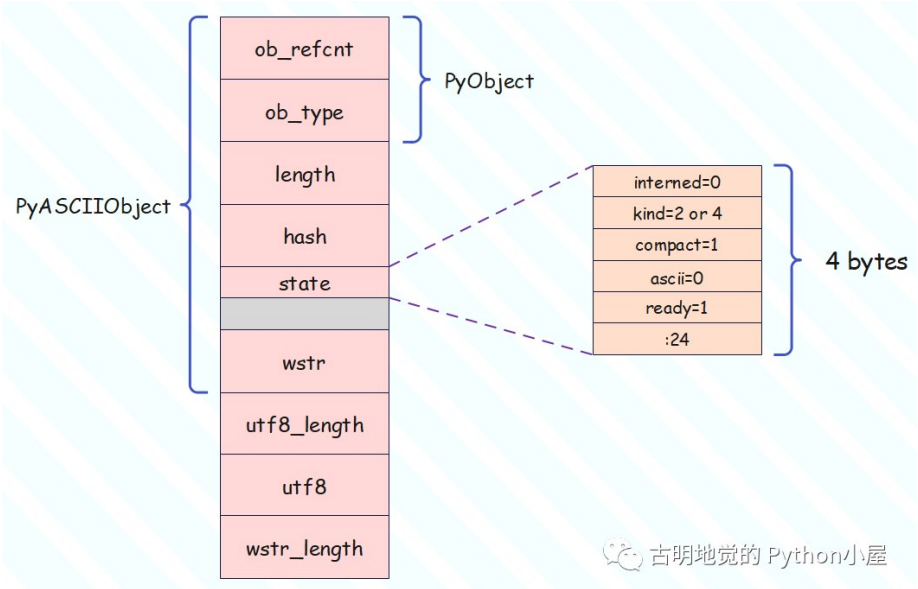
PyCompactUnicodeObject

如果文本不全是ASCII，Unicode对象底层便由**PyCompactUnicodeObject**结构体保存：

```
typedef struct {
    PyASCIIObject _base;
    Py_ssize_t utf8_length;
    char *utf8;
    Py_ssize_t wstr_length;
} PyCompactUnicodeObject;
```

我们看到PyCompactUnicodeObject是在PyASCIIObject的基础上增加了3个字段。

- utf8_length：字符串的utf-8编码长度
- utf8：字符串使用utf-8编码的结果，这里是缓存起来从而避免重复的编码运算
- wstr_length：宽字符的数量



我们说PyCompactUnicodeObject多了3个字段，显然多出了 24 字节。那么之前的 49+24 等于 73，咦不对啊，之前不是说一个是 74 一个是 76 吗？

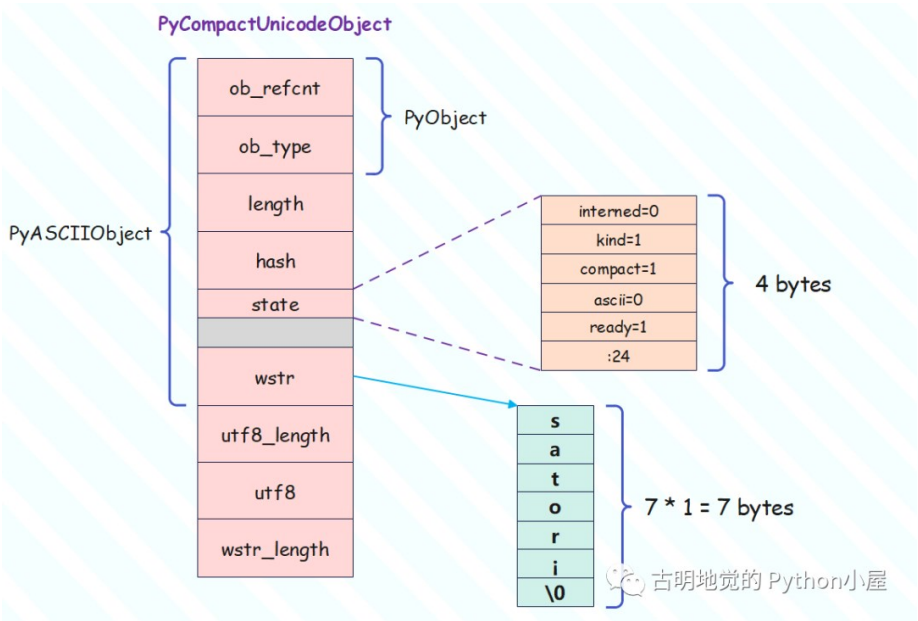
答案是我们忘记了 '\0'，如果使用 UCS2，那么 '\0' 也占两个字节，所以应该是 73 -1

+ 2 = 74；同理 UCS4 是 73 - 1 + 4 = 76。所以此时unicode字符串所占内存我们算是分析完了，然后我们再来看看PyCompactUnicodeObject在这几种不同编码下对应的结构吧。

> 需要注意的是，上面说的 73 也是存在的，当 128<=maxchar<256 的时候，结构体实例额外的部分占 73 字节，至于原因我们下面分析。

PyUnicode_1BYTE_KIND

如果128 <= maxchar < 256，虽然一个字节可以存储的下，但Unicode对象底层也会由PyCompactUnicodeObject结构体保存，字符存储单元为Py_UCS1(Latin-1) ，大小为 1 字节。以字符串sator¡为例，注意结尾的字符是 ¡ ，而不是 i



虽然此时所有的字符都占一个字节，但只有当**maxchar < 128**的时候，才会使用PyASCIIObject。如果大于等于128, 那么会使用 PyCompactUnicodeObject 存储，只不过内部字符依旧每个占一字节。

```
1 import sys
2
3 print(sys.getsizeof("sator¡"))  # 79
```

我们知道对于使用UCS2的PyCompactUnicodeObject来说, 空字符串会占 74 字节，也就是我们说的结构体的额外部分。而这里是 Latin-1，**\0** 是一个字节，所以一个空字符串应该占**73**字节，加上这里的**6**个字符，总共是**79**字节。

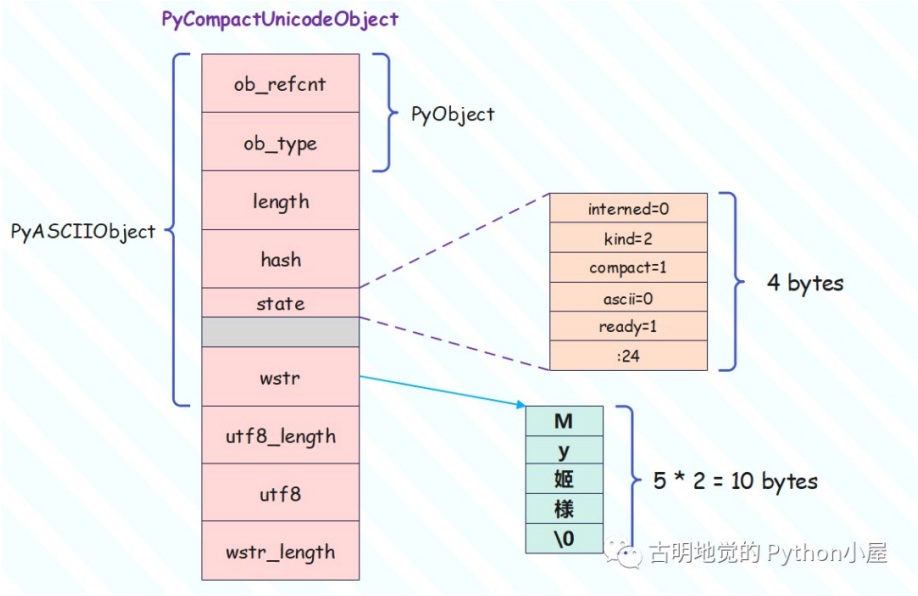
因此当使用Latin1编码的时候，不一定就是PyASCIIObject, 只有当**0< maxchar<128**的时候才会使用 PyASCIIObject。所以，如果将上面的 **sator¡**改成**satori**，那么就会使用PyASCIIObject存储了。

此外还要注意所占的内存，因为Latin1、UCS2、UCS4三个编码都可以对应PyCompactUnicodeObject。而不包括\0 的话，那么一个PyCompactUnicodeObject是占据72字节的。如果算上\0，那么使用**Latin1** 编码的空字符串就是**73**字节，使用**UCS2**编码的空字符串就是**74**字节，使用**UCS4**编码的空字符串就是**76**字节，因为**\0**分别占 1、2、4 字节。

所以我们之前说根据编码的不同，字符串的额外部分可能占据 49、74、76字节，这个结论其实不够准确，还漏掉了一个73。因为maxchar不超过255的字符串虽然可能不是ASCII字符串，但它仍然使用Latin-1编码，所以\0占的是1字节，不是2字节和4字节。

PyUnicode_2BYTE_KIND

如果**256<=maxchar<65536**，Unicode对象底层同样由PyCompactUnicodeObject结构体保存，但字符存储单元为UCS2 ，大小为 2 字节。以字符串**My姬様**为例：



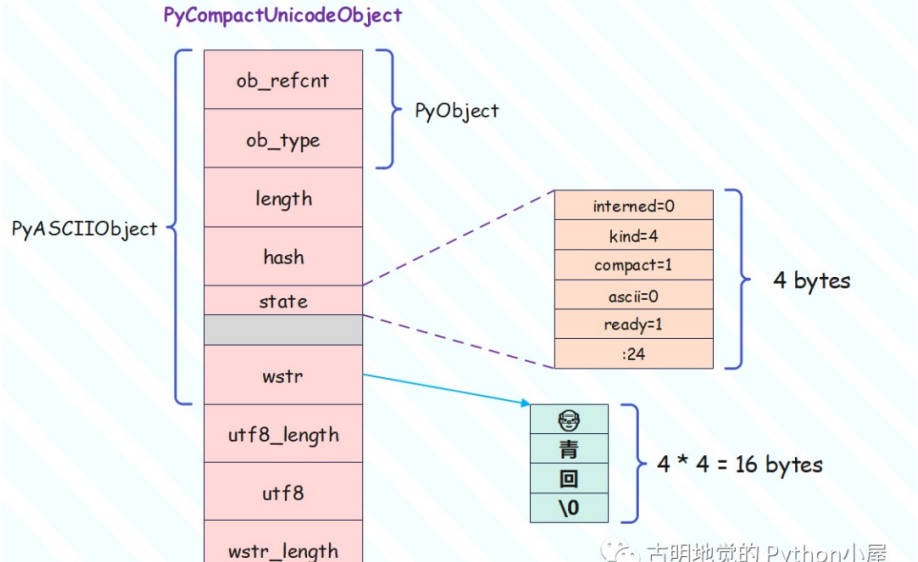
此时state内部的kind为2，使用**UCS2**存储，每个字符占2字节。字符长度为 4，所以字符串总大小为**74+4*2=82**字节，或者理解为**72+5*2**也行。

```
1 import sys
2
3 print(sys.getsizeof("My姬様"))  # 82
```

当文本中包含了**Latin1**无法存储的字符时，会使用两字节的**UCS2**保存，但是连前面的英文字符也变成两字节了。至于原因我们上一篇文章已经分析的很透彻了，因为定位的时候是获取的字符，如果采用变长的utf-8方式存储导致不同类型字符占的内存大小不一，那么就无法以**O(1)**的时间复杂度取出准确的字符了，只能从头到尾依次遍历。而 Go 基于 utf-8，因此它无法获取准确的字符，只能转成 rune，此时内部一个字符直接占4字节。

PyUnicode_4BYTE_KIND

如果**65536<=maxchar< 429496296**，便只能使用4字节存储单元的UCS4了，以字符串"**□青回**"为例：



因此此时每个字符都采用UCS4编码，因此每个字符占四个字节，这是Python内部采取的策略。

```
import sys

# 76 + 3 * 4，或者 72 + 4 * 4
print(sys.getsizeof("青回"))  # 88
```

以上就是PyASCIIObject和PyCompactUnicodeObject，但我们说字符串底层对应的结构是PyUnicodeObject，那么这个PyUnicodeObject长什么样子呢？不用想，它肯定是对前两个结构体进行的封装。

```
typedef struct {
    PyCompactUnicodeObject _base;
    union {
        void *any;
        Py_UCS1 *latin1;
        Py_UCS2 *ucs2;
        Py_UCS4 *ucs4;
    } data;
} PyUnicodeObject;
```

里面的data是一个共同体，这里我们就不深入讨论了，我们直接当成PyCompactUnicodeObject来用即可。

小结

以上就是字符串的底层结构，相对来说要复杂了一些，但也不是很难理解。下一篇文章，我们来说说字符串是如何创建的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》22. 字符串是怎么被创建的

下一篇 ›
《源码探秘 CPython》20. Python是怎么存储字符串的?