10 str 对象，统一的 Unicode 字符串-慕课专栏

Unicode是什么

计算机存储的基本单位是 **八位字节** ，由 *8* 个比特位组成，简称 **字节** 。由于英文只由 *26* 个字母加若干符号组成，因此英文字符可以直接用 **字节** 来保存。其他诸如中日韩等语言，由于字符众多，则不得不用多个字节来编码。

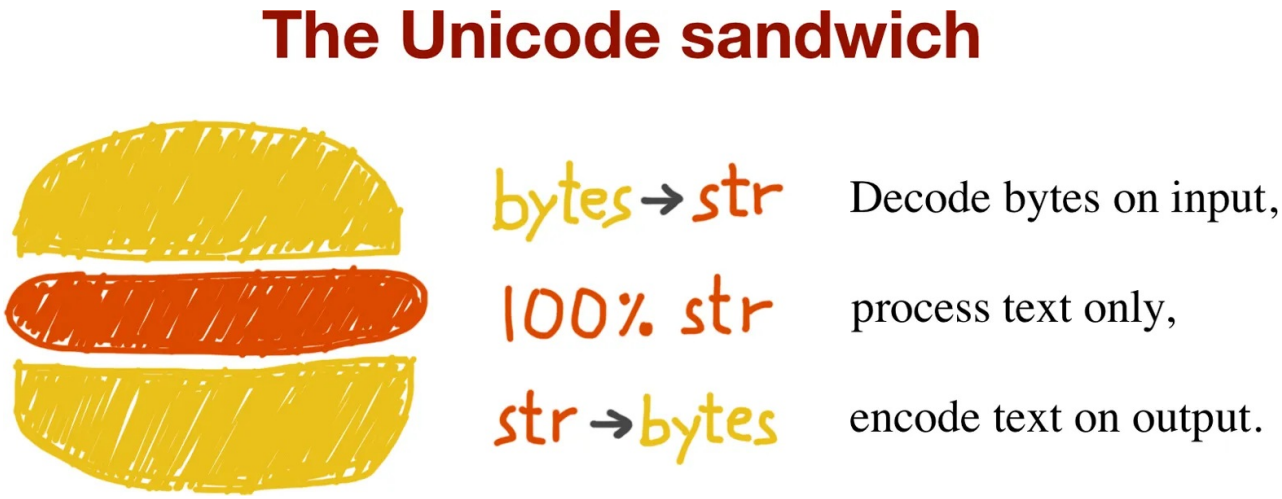
随着计算机技术的传播，非拉丁文字符编码技术蓬勃发展，但存在两个比较大的局限性：

- **不支持多语言** ，例如中文的编码方案不能表示日文；
- **没有统一标准** ，例如中文有 *GB2312* ，*GBK* 、 *GB18030* 等多种编码标准；

由于编码方式不统一，开发人员经常需要在不同编码间来回转化，错误频出。为了彻底解决这些问题， *统一码联盟* 提出了 *Unicode* 标准。*Unicode* 对世界上大部分文字系统进行整理、编码，让计算机可以用统一的方式处理文本。*Unicode* 目前已经收录了超过 *13* 万个字符，天然地支持多语言。使用 *Unicode* ，即可彻底跟编码问题说拜拜！

Python中的Unicode

Python 在 *3* 之后，*str* 对象内部改用 *Unicode* 表示，因而被源码称为 *Unicode* 对象。这么做好处是显然易见的，程序核心逻辑统一用 *Unicode* ，只需在输入、输入层进行编码、解码，可最大程度避免各种编码问题：



由于 *Unicode* 收录字符已经超过 *13* 万个，每个字符至少需要 *4* 个字节来保存。这意味着巨大的内存开销，显然是不可接受的。英文字符用 *ASCII* 表示仅需 *1* 个字节，而用 *Unicode* 表示内存开销却增加 *4* 倍！

Python 作者们肯定不允许这样的事情发生，不信我们先来观察下(*getsizeof* 获取对象内存大小)：

```
>>> import sys

>>> sys.getsizeof('ab') - sys.getsizeof('a')
1

>>> sys.getsizeof('中国') - sys.getsizeof('中')
2

>>> sys.getsizeof('??') - sys.getsizeof('?')
4
```

- 每个 *ASCII* 英文字符，占用 *1* 字节；
- 每个中文字符，占用 *2* 字节；
- *Emoji* 表情，占用 *4* 字节；

由此可见，*Python* 内部对 *Unicode* 进行优化：根据文本内容，选择底层存储单元。至于这种黑科技是怎么实现的，我们只能到源码中寻找答案了。与 *str* 对象实现相关源码如下：

- *Include/unicodeobject.h*
- *Objects/unicodectype.c*

在 *Include/unicodeobject.h* 头文件中，我们发现 *str* 对象底层存储根据文本字符 *Unicode* 码位范围分成几类：

- *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；
- 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；

```
enum PyUnicode_Kind {

   PyUnicode_WCHAR_KIND = 0,

   PyUnicode_1BYTE_KIND = 1,
   PyUnicode_2BYTE_KIND = 2,
   PyUnicode_4BYTE_KIND = 4
};
```

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。

```
typedef uint32_t Py_UCS4;
typedef uint16_t Py_UCS2;
typedef uint8_t Py_UCS1;
```

文本类型	字符存储单元	字符存储单元大小（字节）
PyUnicode_1BYTE_KIND	Py_UCS1	1

文本类型	字符存储单元	字符存储单元大小（字节）
PyUnicode_2BYTE_KIND	Py_UCS2	2
PyUnicode_4BYTE_KIND	Py_UCS4	4

Unicode 内部存储结构因文本类型而异，因此类型 *kind* 必须作为 *Unicode* 对象公共字段保存。*Python* 内部定义了若干个 **标志位** ，作为 *Unicode* 公共字段，*kind* 便是其中之一：

- *interned* ，是否为 *interned* 机制维护， *internel* 机制在本节后半部分介绍；
- *kind* ，类型，用于区分字符底层存储单元大小；
- *compact* ，内存分配方式，对象与文本缓冲区是否分离，本文不涉及分离模式；
- *ascii* ，文本是否均为纯 *ASCII* ；

Objects/unicodectype.c 源文件中的 *PyUnicode_New* 函数，根据文本字符数 *size* 以及最大字符 *maxchar* 初始化 *Unicode* 对象。该函数根据 *maxchar* 为 *Unicode* 对象选择最紧凑的字符存储单元以及底层结构体：

	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
字符存储单元大小	1	1	2	4
底层结构体	PyASCIIObject	PyCompactUnicodeObject	PyCompactUnicodeObject	PyCompactUnicodeObject

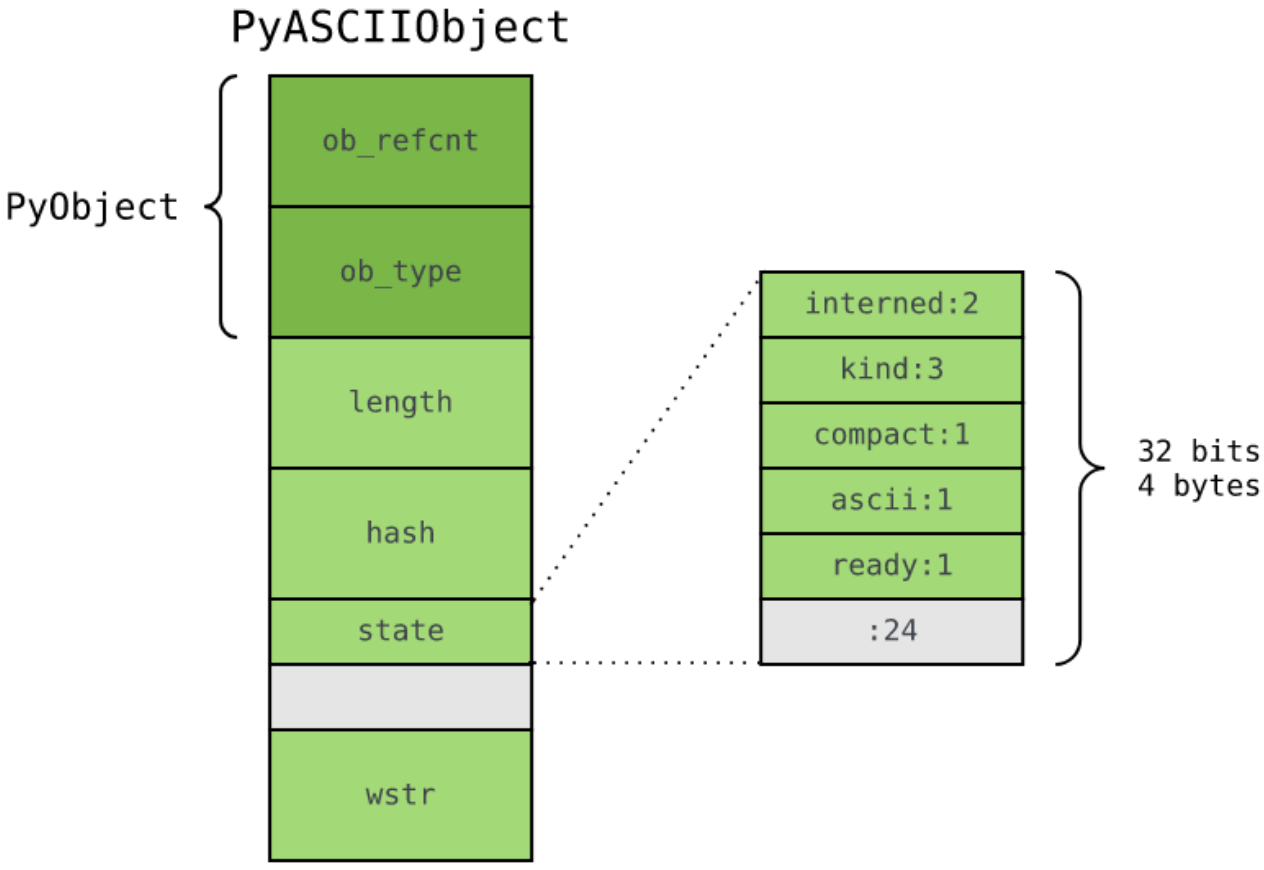
PyASCIIObject

如果 *str* 对象保存的文本均为 *ASCII* ，即 maxchar<128maxchar<128maxchar<128，则底层由 *PyASCIIObject* 结构存储：

```
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;
```

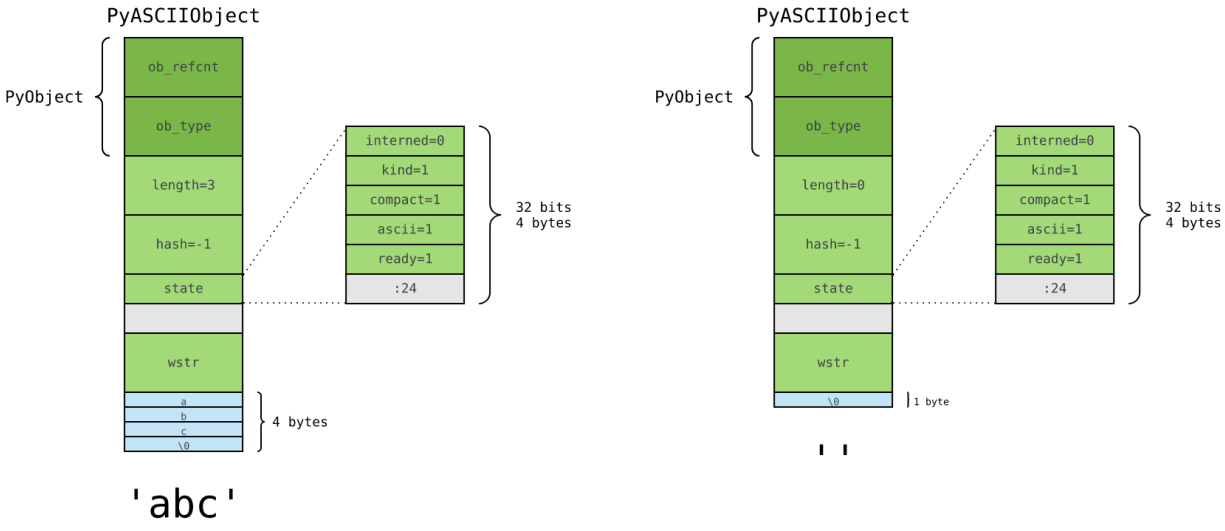
PyASCIIObject 结构体也是其他 *Unicode* 底层存储结构体的基础，所有字段均为 *Unicode* 公共字段：

- *ob_refcnt*，引用计数；
- *ob_type*，对象类型；
- *length*，文本长度；
- *hash*，文本哈希值；
- *state*，*Unicode* 对象标志位，包括 *internel*、 *kind*、 *ascii*、 *compact* 等；
- *wstr*，略；



注意到，*state* 字段后有一个 *4* 字节的空洞，这是结构体字段 **内存对齐** 造成的现象。在 *64* 位机器下，指针大小为 *8* 字节，为优化内存访问效率，*wstr* 必须以 *8* 字节对齐；而 *state* 字段大小只是 *4* 字节，便留下 *4* 字节的空洞。*PyASCIIObject* 结构体大小在 *64* 位机器下为 *48* 字节，在 32 位机器下为 *24* 字节。

ASCII 文本则紧接着位于 *PyASCIIObject* 结构体后面，以字符串对象 *'abc'* 以及空字符串对象 *''* 为例：



注意到，与 *bytes* 对象一样，*Python* 也在 *ASCII* 文本末尾，额外添加一个 *\0* 字符，以兼容 *C* 字符串。

如此一来，以 *Unicode* 表示的 *ASCII* 文本，额外内存开销仅为 *PyASCIIObject* 结构体加上末尾的 *\0* 字节而已。*PyASCIIObject* 结构体在 64 位机器下，大小为 *48* 字节。因此，长度为 n 的纯 *ASCII* 字符串对象，需要消耗 n+48+1，即 n+49 字节的内存空间。

```
>>> sys.getsizeof('')
49
>>> sys.getsizeof('abc')
52
>>> sys.getsizeof('a' * 10000)
10049
```

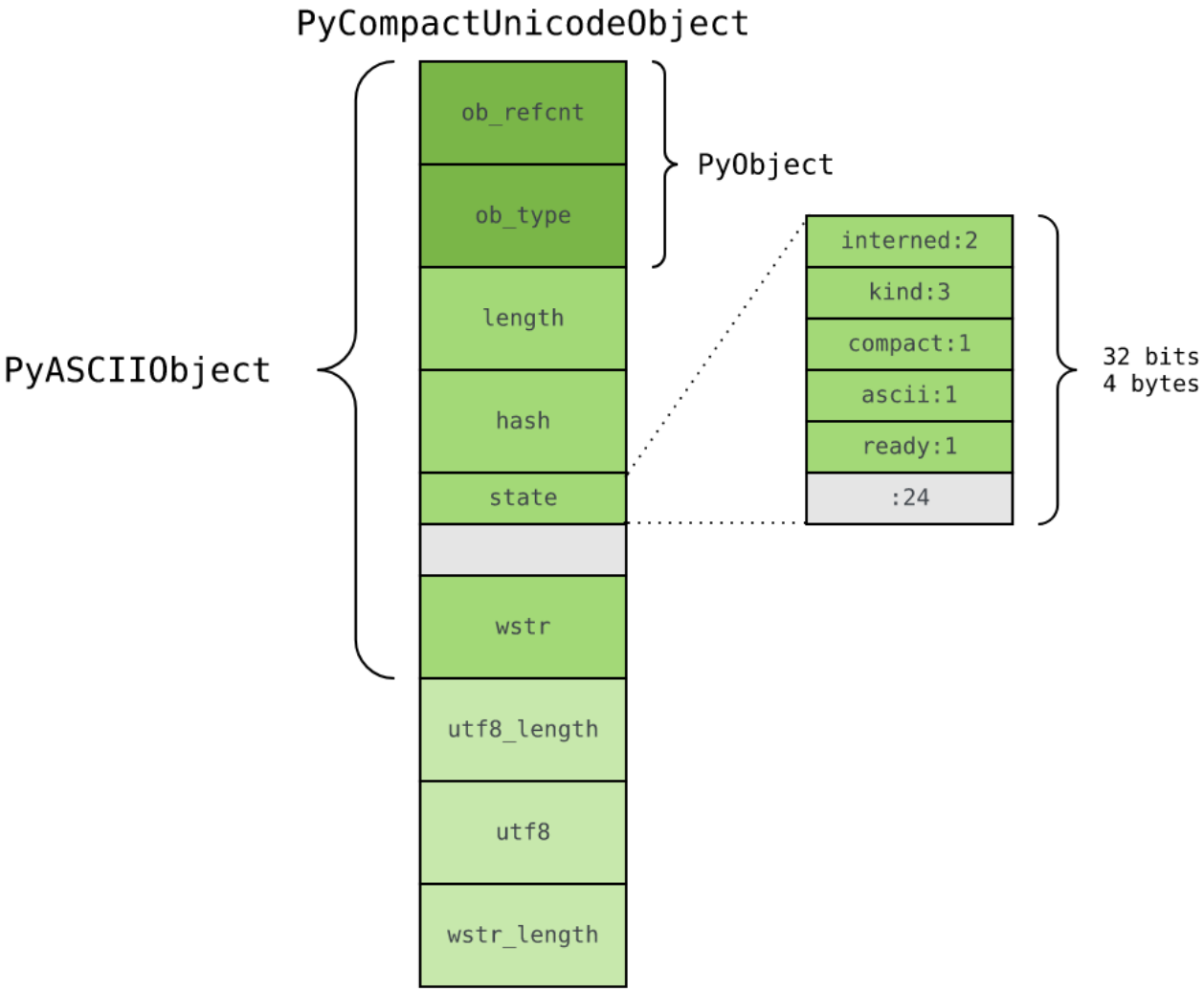
PyCompactUnicodeObject

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;
```

PyCompactUnicodeObject 在 *PyASCIIObject* 基础上，增加 *3* 个字段：

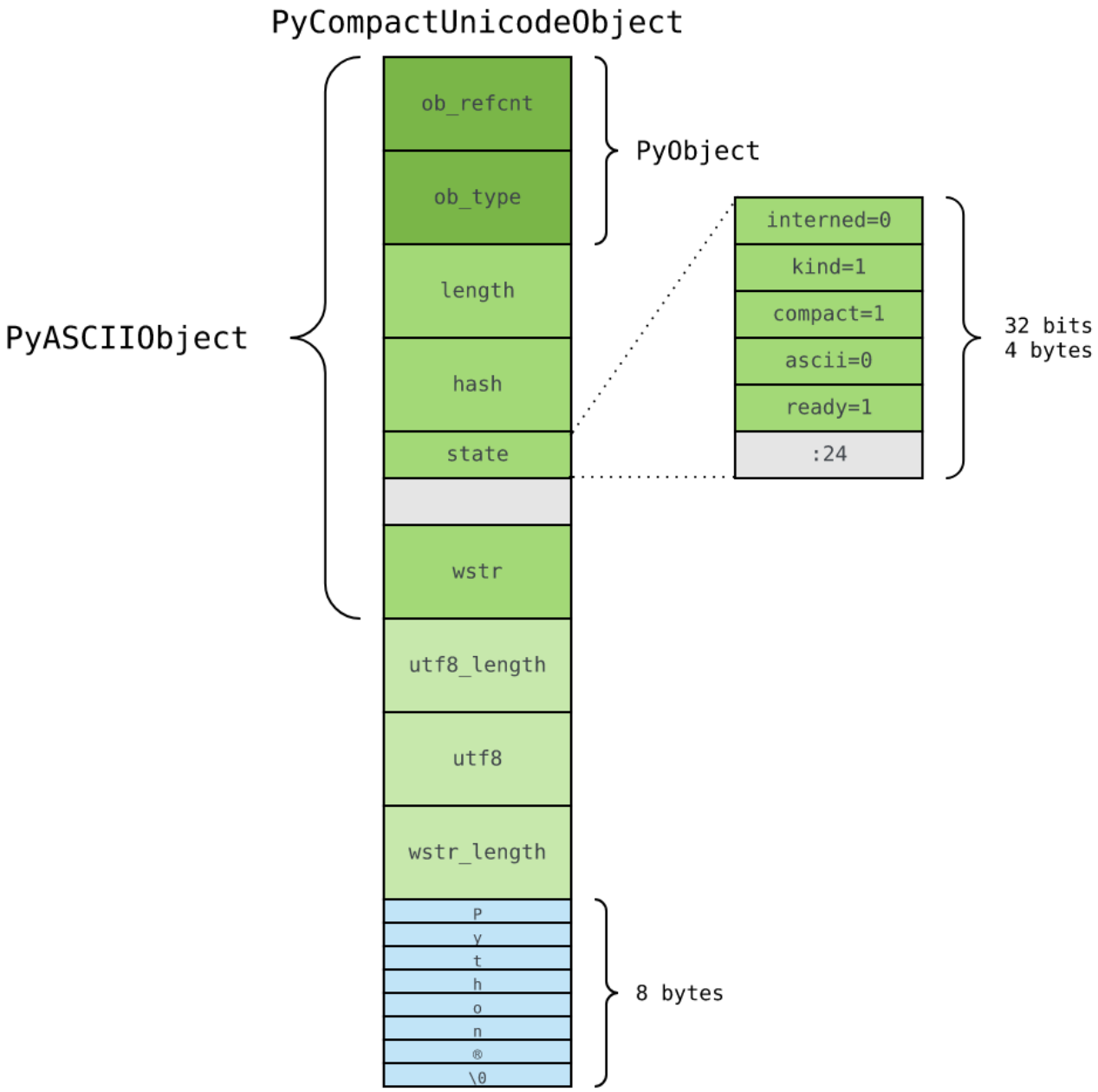
- *utf8_length* ，文本 *UTF8* 编码长度；
- *utf8* ，文本 *UTF8* 编码形式，缓存以避免重复编码运算；
- wstr_length ，略；



由于 ASCII 本身兼容 UTF8 ，无须保存 UTF8 编码形式，这也是 ASCII 文本底层由 PyASCIIObject 保存的原因。在 *64* 位机器，*PyCompactUnicodeObject* 结构体大小为 *72* 字节；在 *32* 位机器则是 *36* 字节。

PyUnicode_1BYTE_KIND

如果 128<=maxchar<256128<=maxchar<256128<=maxchar<256，*Unicode* 对象底层便由 *PyCompactUnicodeObject* 结构体保存，字符存储单元为 *Py_UCS1* ，大小为 *1* 字节。以 *Python*® 为例，字符 ® 码位为 U+00AE ，满足该条件，内部结构如下：

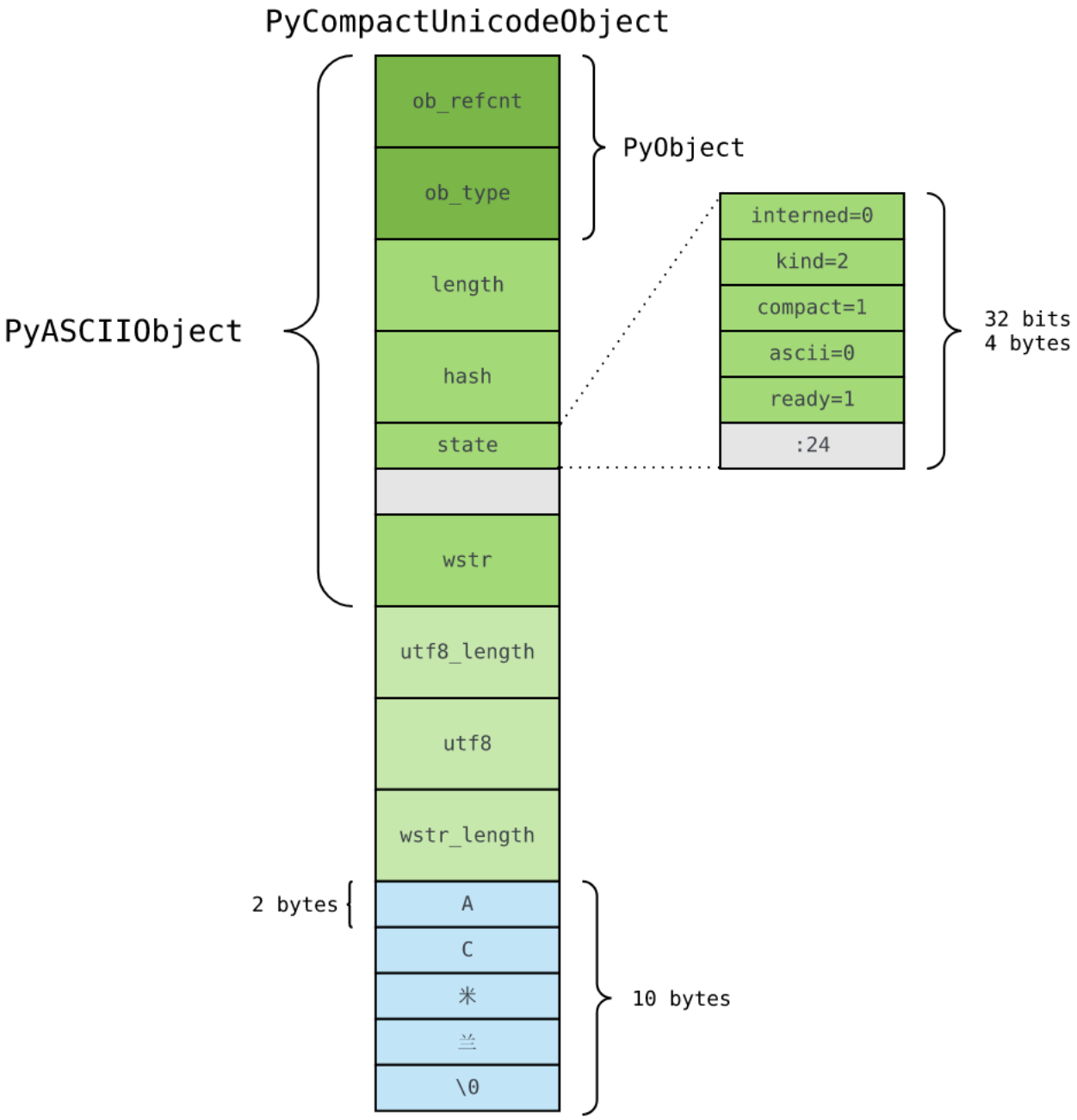


字符存储单元还是 *1* 字节，跟 *ASCII* 文本一样。 因此，*Python®* 对象需要占用 *80* 字节的内存空间72+1∗7+1=72+8=8072+1*7+1=72+8=8072+1∗7+1=72+8=80：

```
>>> sys.getsizeof('Python®')
80
```

PyUnicode_2BYTE_KIND

如果 256<=maxchar<65536256<=maxchar<65536256<=maxchar<65536，*Unicode* 对象底层同样由 *PyCompactUnicodeObject* 结构体保存，但字符存储单元为 *Py_UCS2* ，大小为 *2* 字节。以 *AC米兰* 为例，常用汉字码位在 *U+0100* 到 *U+FFFF* 之间，满足该条件，内部结构如下：



由于现在字符存储单元为 2 字节，故而 *str* 对象 *AC米兰* 需要占用 82 字节的内存空间：72+2∗4+2=72+10=8272+2*4+2=72+10=8272+2∗4+2=72+10=82

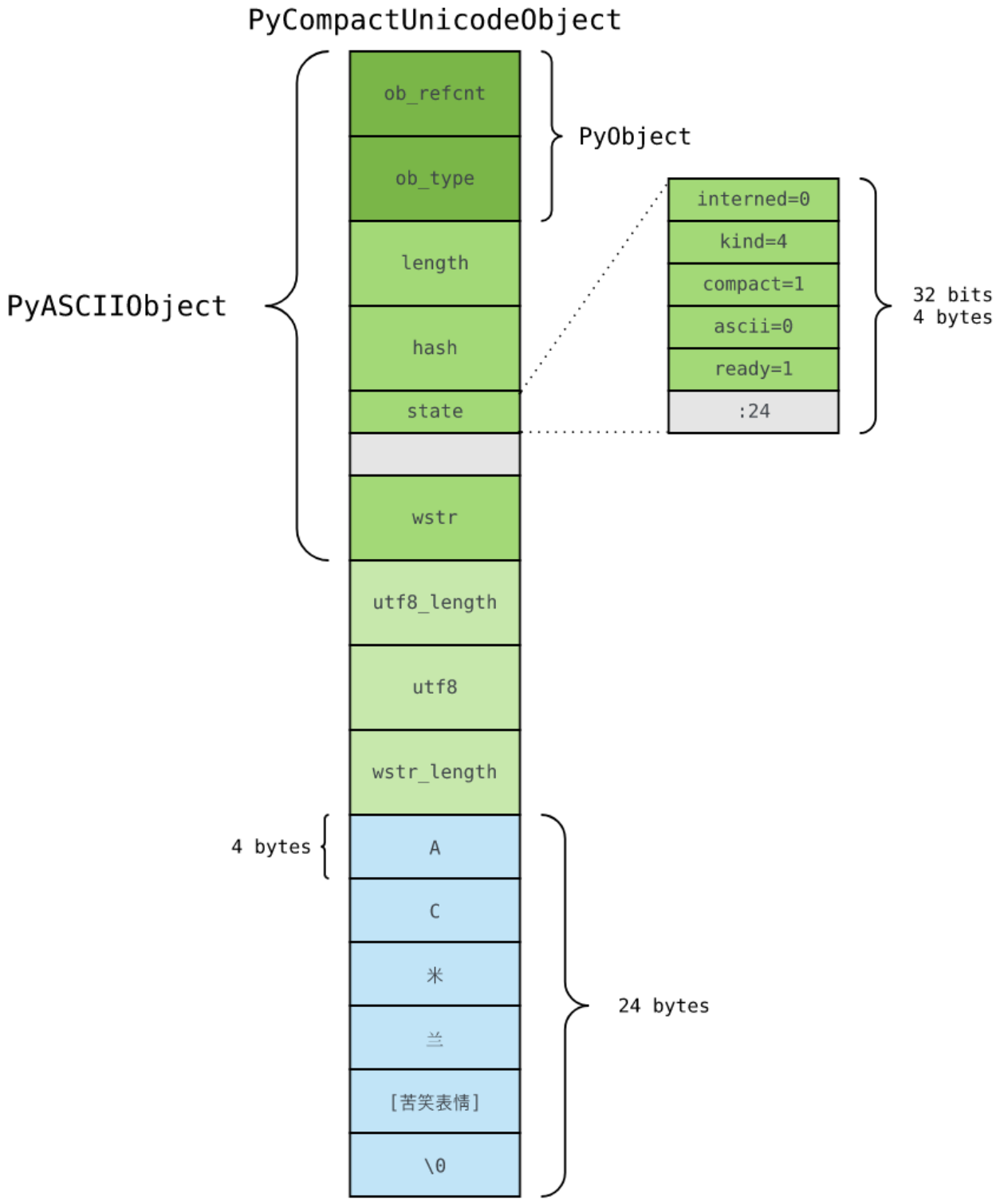
```
>>> sys.getsizeof('AC米兰')
82
```

我们看到，当文本包含中文后，英文字母也只能用 *2* 字节的存储单元来保存了。

你可能会提出疑问，为什么不采用变长存储单元呢？例如，字母 *1* 字节，汉字 *2* 字节？这是因为采用变长存储单元后，就无法在 *O(1)* 时间内取出文本第 n 个字符了——你只能从头遍历直到遇到第 n 个字符。

PyUnicode_4BYTE_KIND

如果
65536<=maxchar<42949629665536<=maxchar<42949629665536<=maxchar<429496296，
便只能用 *4* 字节存储单元 *Py_UCS4* 了。以 *AC米兰?* 为例：




```
>>> sys.getsizeof('AC米兰')
96
```

这样一来，给一段英文文本加上表情，内存暴增 4 倍，也就不奇怪了：

```
>>> text = 'a' * 1000
>>> sys.getsizeof(text)
1049
>>> text += '?'
>>> sys.getsizeof(text)
4080
```

interned机制

如果 *str* 对象 *interned* 标识位为 *1* ，*Python* 虚拟机将为其开启 *interned* 机制。那么，什么是 *interned* 机制？

先考虑以下场景，如果程序中有大量 *User* 对象，有什么可优化的地方？

```
>>> class User:
...
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>>
>>> user = User(name='tom', age=20)
>>> user.__dict__
{'name': 'tom', 'age': 20}
```

由于对象的属性由 *dict* 保存，这意味着每个 *User* 对象都需要保存 *str* 对象 *name* 。换句话讲，*1* 亿个 *User* 对象需要重复保存 *1* 亿个同样的 *str* 对象，这将浪费多少内存！

由于 *str* 是不可变对象，因此 *Python* 内部将有潜在重复可能的字符串都做成 **单例模式** ，这就是 *interned* 机制。*Python* 具体做法是在内部维护一个全局 *dict* 对象，所有开启 *interned* 机制 *str* 对象均保存在这里；后续需要用到相关对象的地方，则优先到全局 *dict* 中取，避免重复创建。

举个例子，虽然 *str* 对象 *'abc'* 由不同的运算产生，但背后却是同一个对象：

```
>>> a = 'abc'
>>> b = 'ab' + 'c'
>>> id(a), id(b), a is b
(4424345224, 4424345224, True)
```