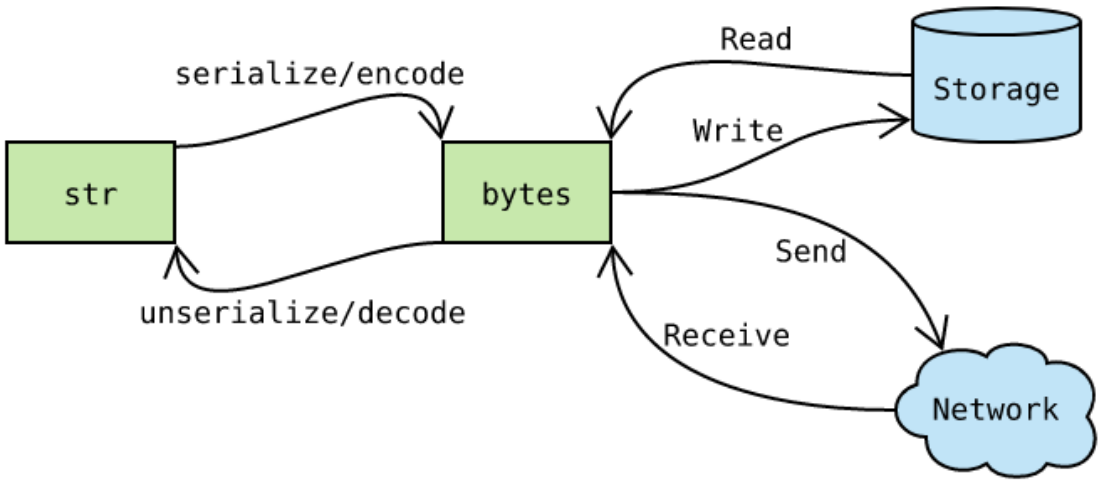
对象结构

不少编程语言中的 **字符串** 都是由 **字符数组** (或称为 **字节序列**)来表示， *C* 语言就是这样。

```
char msg[] = "Hello, world!";
```

由于一个字节最多只能表示 *256* 种字符，用来表示英文字符绰绰有余，想覆盖非英文字符便捉襟见肘了。为了表示众多的非英文字符(比如汉字)，计算机先驱们发明了 **多字节编码** ——通过多个字节来表示一个字符。由于原始字节序列不维护编码信息，操作不慎便导致各种乱码现象。

Python 提供的解决方案是 *Unicode 字符串* (*str*)对象， *Unicode* 可以表示各种字符，无需关心编码。然而存储或者网络通讯时，字符串对象不可避免要 **序列化** 成字节序列。为此， *Python* 额外提供了字节序列对象—— *bytes* 。

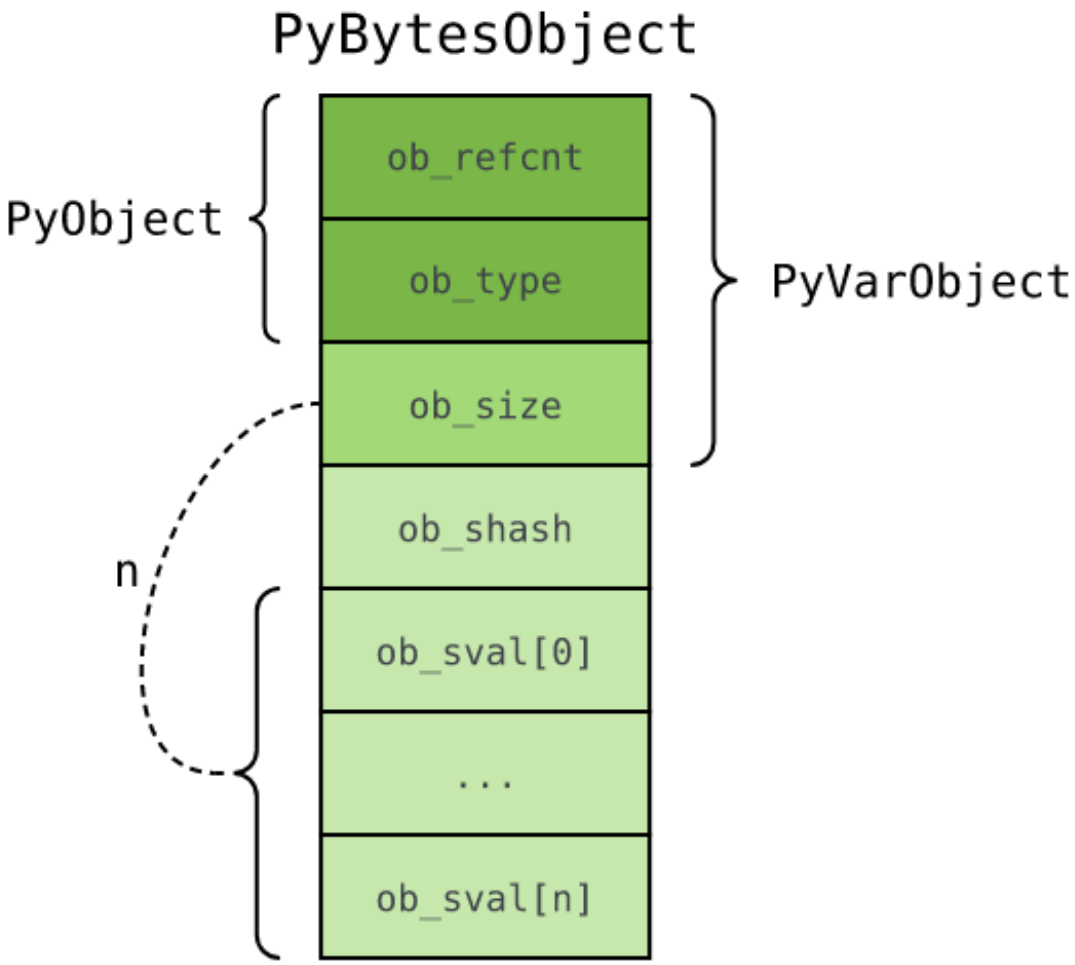


如上图， *str* 对象统一表示一个 **字符串** ，不需要关心编码；计算机通过 **字节序列** 与存储介质和网络介质打交道，字节序列由 *bytes* 对象表示；存储或传输 *str* 对象时，需要将其 **序列化** 成字节序列，序列化过程也是 **编码** 的过程。

好了，我们已经弄明白 *str* 对象以 *bytes* 之间的关系，这两者是 *Python* 中最重要的内建对象之一。 读者对 *str* 对象应该再熟悉不过了，但对更接近底层的 *bytes* 对象可能涉猎不多。没关系，经过本节学习，你将彻底掌握它！

bytes 对象用于表示由若干字节组成的 **字节序列** 以及相关的 **操作** ，并不关心字节序列的 **含义** 。因此， *bytes* 应该是一种 **变长对象** ，内部由 *C* 数组实现。 *Include/boolobject.h* 头文件中的定义印证了我们的猜测：

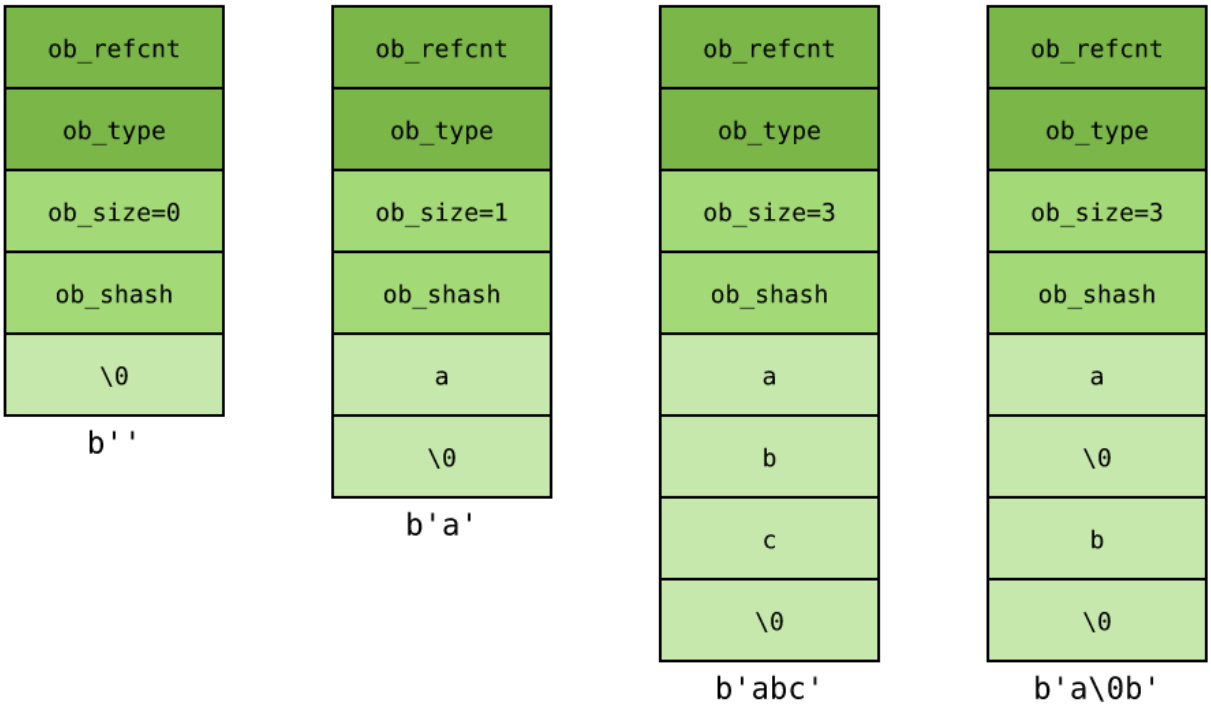
```
typedef struct {
    PyObject_VAR_HEAD
    Py_hash_t ob_shash;
    char ob_sval[1];

} PyBytesObject;
```



字节序列对象 *PyBytesObject* 中，确实藏着一个字符数组 *ob_sval* 。注意到 *ob_sval* 数组长度定义为 *1* ，这是 *C* 语言中定义 **变长数组** 的技巧。这个技巧在前面章节(*int 对象，永不溢出的整数*)中介绍过，这里不再赘述。源码注释表明， *Python* 为待存储的字节序列额外分配一个字节，用于在末尾处保存 `\0` ，以便兼容 *C* 字符串。

此外，我们还留意到另一个字段 *ob_shash* ，它用于保存字节序列的 **哈希值** 。 *Python* 对象哈希值应用范围很广，比如 *dict* 字典对象依赖对象哈希值进行存储。由于计算 *bytes* 对象哈希值需要遍历其内部的字符数组，开销相对较大。因此， *Python* 选择将哈希值保存起来，以空间换时间，避免重复计算。

最后，以几个典型例子结束 *bytes* 对象结构介绍，以此加深理解：



由此可见，就算空 *bytes* 对象(`b''`)也是要占用内存空间的，至少变长对象 **公共头部** 是少不了的。

```
>>> sys.getsizeof(b'')
33
```

bytes 对象占用的内存空间可分为以下个部分进行计算：

- 变长对象公共头部 *24* 字节，*ob_refcnt* 、 *ob_type* 、 *ob_size* 每个字段各占用 *8* 字节；
- 哈希值 *ob_shash* 占用 *8* 字节；
- 字节序列本身，假设是 *n* 字节；
- 额外 *1* 字节用于存储末尾处的 `\0` ；

因此， *bytes* 对象空间计算公式为 24+8+n+124+8+n+124+8+n+1，即 33+n33+n33+n，其中 n 为字节序列长度。

对象行为

现在，我们开始考察 *bytes* 对象的 **行为** 。由于对象的行为由对象的 **类型** 决定，因而我们需要到 *bytes* 类型对象中寻找答案。在 *Objects/bytesobject.c* 源码文件中，我们找到 *bytes* **类型对象** 的定义：

```
PyTypeObject PyBytes_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "bytes",
    PyBytesObject_SIZE,
    sizeof(char),

    &bytes_as_number,
    &bytes_as_sequence,
    &bytes_as_mapping,
    (hashfunc)bytes_hash,

};
```

我们对类型对象的内部结构已经非常熟悉了， *tp_as_xxxx* 系列结构体决定了对象支持的各种 **操作** 。举个例子， *bytes_as_number* 结构体中保存着 **数值运算** 处理函数的指针。 *bytes* 对象居然支持数据操作，没搞错吧？我们看到， *bytes_as_number* 结构体中只定义了一个操作—— **模运算** (*%*)：

```
static PyNumberMethods bytes_as_number = {
    0,
    0,
    0,
    bytes_mod,
}

static PyObject *
bytes_mod(PyObject *self, PyObject *arg)
{
    if (!PyBytes_Check(self)) {
        Py_RETURN_NOTIMPLEMENTED;
    }
    return _PyBytes_FormatEx(PyBytes_AS_STRING(self), PyBytes_GET_SIZE(self),
                             arg, 0);
}
```

由此可见， *bytes* 对象只是借用 *%* 运算符实现字符串格式化，谈不上支持数值运算，虚惊一场：

```
>>> b'msg: a=%d b=%d' % (1, 2)
b'msg: a=1 b=2'
```

序列型操作

众所周知， *bytes* 是 **序列型对象** ，序列型操作才是研究重点。我们在 *bytes_as_sequence* 结构体中找到相关定义：

```
static PySequenceMethods bytes_as_sequence = {
    (lenfunc)bytes_length,
    (binaryfunc)bytes_concat,
    (ssizeargfunc)bytes_repeat,
    (ssizeargfunc)bytes_item,
    0,
    0,
    0,
    (objobjproc)bytes_contains
};
```

由此可见， *bytes* **支持的** **序列型操作** **包括以下** *5* **个：**

- *sq_length* **，查询序列长度；**
- *sq_concat* **，将两个序列合并为一个；**
- *sq_repeat* **，将序列重复多次；**
- *sq_item* **，取出给定下标序列元素；**
- *sq_contains*，**包含关系判断；**

长度

最简单的序列型操作是 **长度查询** **，直接返回** *ob_size* **字段即可：**

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

合并

```
>>> b'abc' + b'cba'
b'abccba'
```

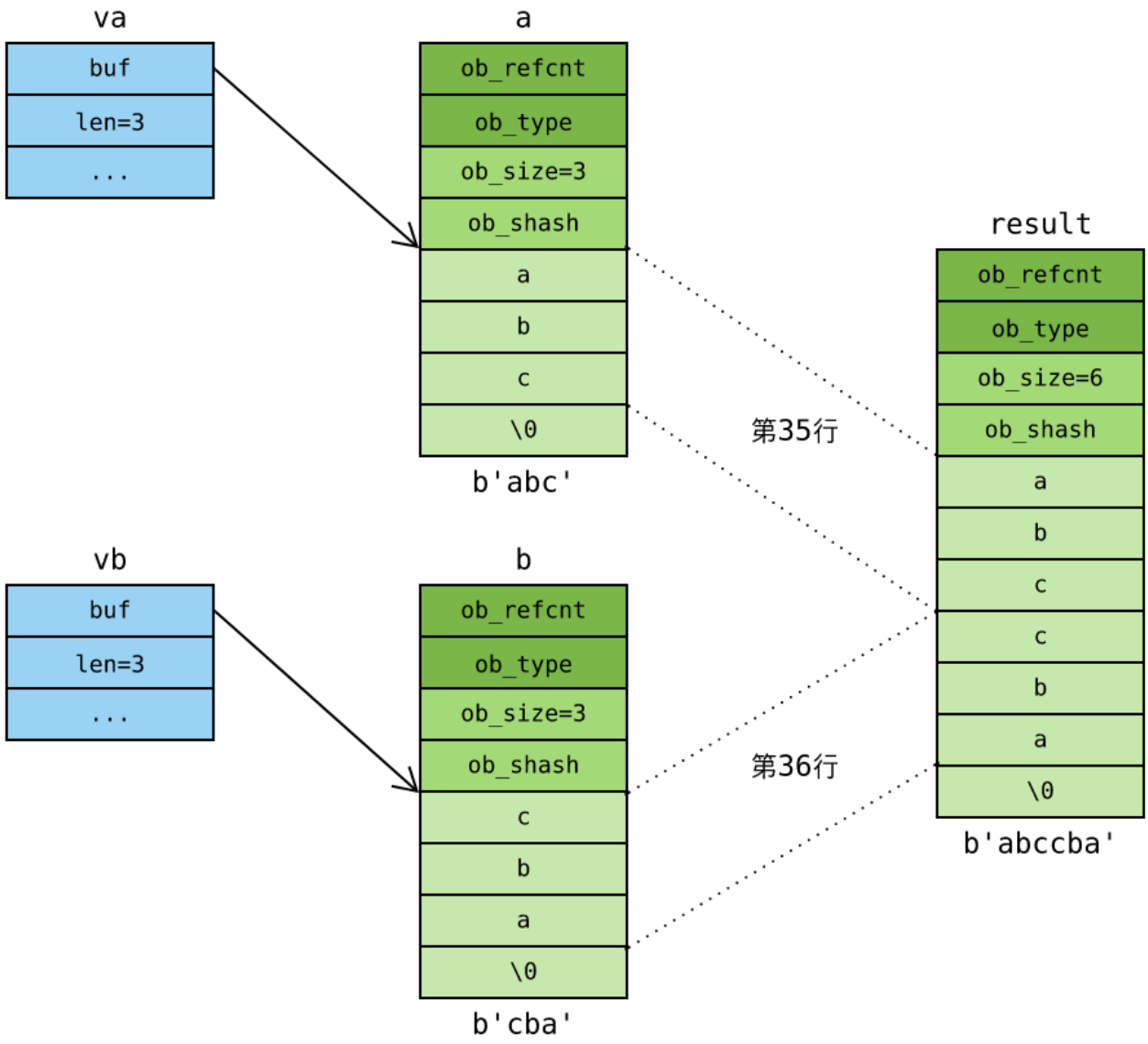
合并操作将两个 *bytes* **对象拼接成一个，由** *bytes_concat* **函数处理：**

```
static PyObject *
bytes_concat(PyObject *a, PyObject *b)
{
    Py_buffer va, vb;
    PyObject *result = NULL;

    va.len = -1;
    vb.len = -1;
    if (PyObject_GetBuffer(a, &va, PyBUF_SIMPLE) != 0 ||
        PyObject_GetBuffer(b, &vb, PyBUF_SIMPLE) != 0) {
        PyErr_Format(PyExc_TypeError, "can't concat %.100s to %.100s",
                     Py_TYPE(b)->tp_name, Py_TYPE(a)->tp_name);
        goto done;
    }


    if (va.len == 0 && PyBytes_CheckExact(b)) {
        result = b;
        Py_INCREF(result);
        goto done;
    }
    if (vb.len == 0 && PyBytes_CheckExact(a)) {
        result = a;
        Py_INCREF(result);
        goto done;
    }

    if (va.len > PY_SSIZE_T_MAX - vb.len) {
        PyErr_NoMemory();
        goto done;
    }

    result = PyBytes_FromStringAndSize(NULL, va.len + vb.len);
    if (result != NULL) {
        memcpy(PyBytes_AS_STRING(result), va.buf, va.len);
        memcpy(PyBytes_AS_STRING(result) + va.len, vb.buf, vb.len);
    }

  done:
    if (va.len != -1)
        PyBuffer_Release(&va);
    if (vb.len != -1)
        PyBuffer_Release(&vb);
    return result;
}
```

1. 第 *4-5* 行，定义局部变量 *va* 、 *vb* 用于维护缓冲区， *result* 用于保存合并结果；
2. 第 *7-14* 行，从待合并对象中获取字节序列所在缓冲区；
3. 第 *17-21* 行，如果第一个对象长度为 *0* ，第二个对象就是结果；
4. 第 *22-26* 行，反之第二个对象长度为 *0* ，第一个对象就是结果；
5. 第 *28-31* 行，长度超过限制则报错，其实判断条件这样写更直观：*va.len + vb.len > PY_SSIZE_T_MAX* ；
6. 第 *33* 行，新建 *bytes* 对象用于保存合并结果，长度为待合并对象长度之和；

7. 第 *34-37* 行，将字节序列从待合并对象拷贝到结果对象；
8. 第 *39-44* 行，返回结果。

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内部差异：



bytes_concat 函数逻辑很直白，将两个 *bytes* 对象的缓冲区拷贝到一起形成新 *bytes* 对象。*sq_repeat* 等其他处理函数也不复杂，因篇幅关系不再单独讲解了。鼓励读者们自行深入源码，弄清他们的来龙去脉，必有收获。

数据拷贝的陷阱

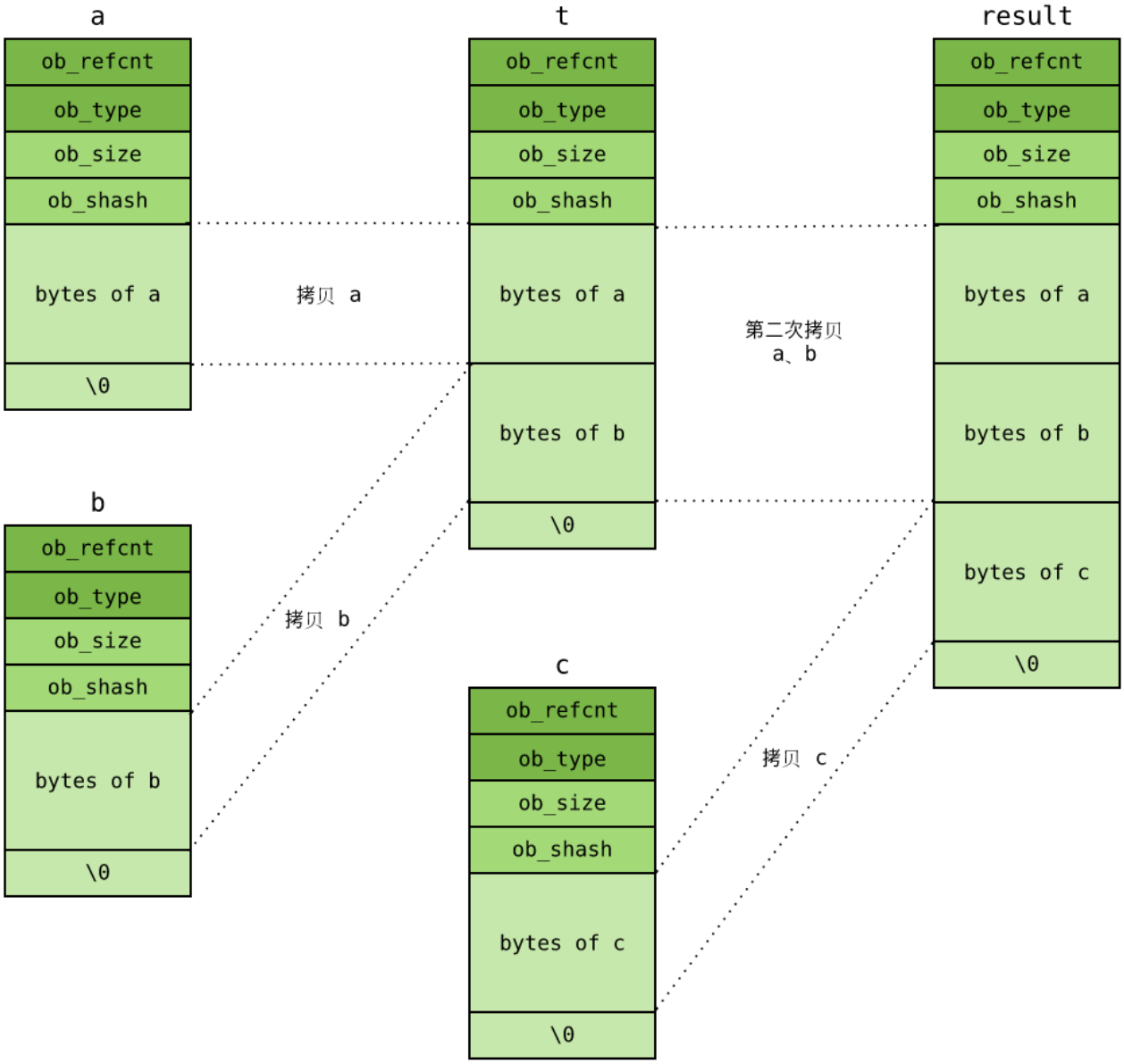
考察以下表达式——合并 *3* 个 *bytes* 对象：

```
>>> result = a + b + c
```

这个语句执行时，分成两步进行合并：先将 *a* 和 *b* 合并，得到临时结果 *t* ，再将 *t* 和 *c* 合并得到最终结果 *result* ：

```
>>> t = a + b
>>> result = t + c
```

眼尖的读者已经挑出毛病来了——这个过程中，*a* 和 *b* 的数据需要被拷贝两遍！



而且，待合并的 *bytes* 对象越多，数据拷贝越严重。考察这两个典型表达式以及相关对象拷贝次数：

```
2   2   1
a + b + c

6   6   5   4   3   2   1
a + b + c + d + e + f + g
```

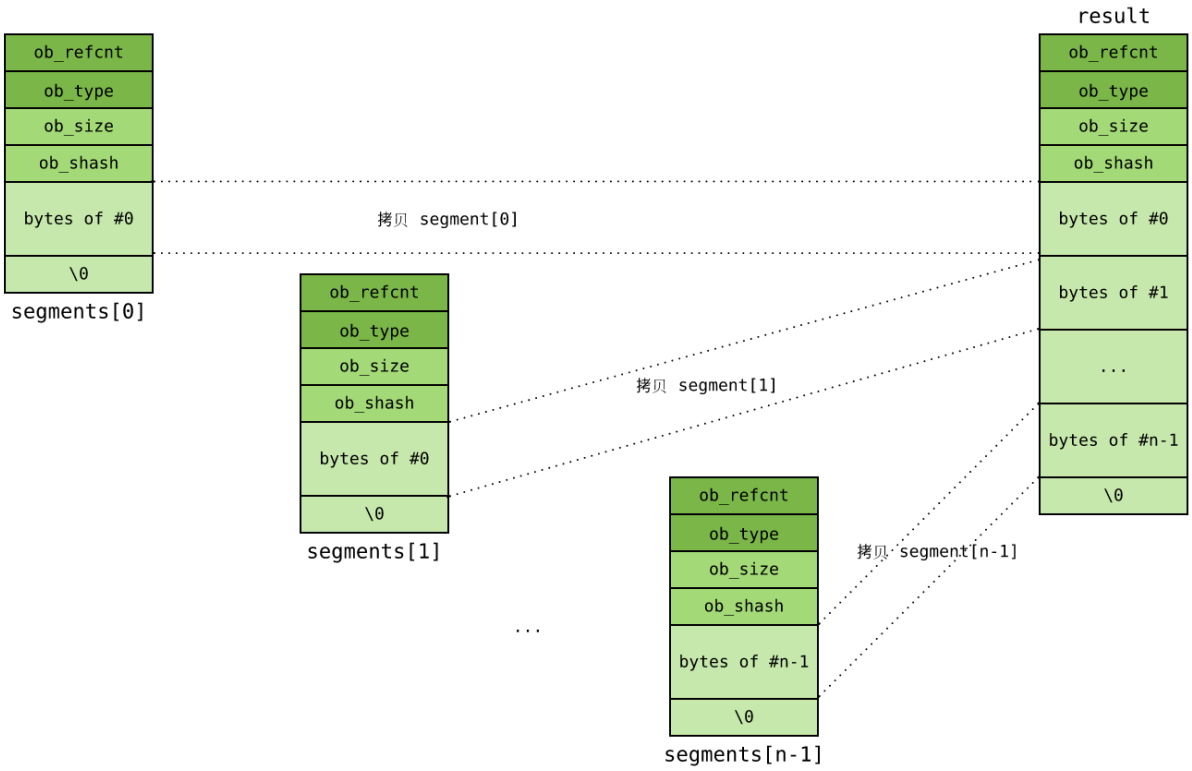
由此可见，合并 n 个 *bytes* 对象，头两个对象需要拷贝 n-1 次，只有最后一个对象不需要重复拷贝。平均下来，每个对象大约要拷贝 n/2n/2n/2 次！知道这个陷阱之后，你还敢写这样的代码吗？

```
>>> result = b''
>>> for s in segments:
...     result += s
```

好在 *bytes* 对象提供了一个内建方法 *join* ，可高效合并多个 *bytes* 对象：

```
>>> result = b''.join(segments)
```

join 方法对数据拷贝进行了优化：先遍历待合并对象，计算总长度；然后根据总长度创建目标对象；最后再遍历待合并对象，逐一拷贝数据。这样一来，每个对象均只需拷贝一次，解决了重复拷贝的陷阱。



join 内建方法同样在 *Objects/bytesobject.c* 文件中实现，*bytes_join* 是也，这里不再展开介绍了。

字符缓冲池

为了优化单字节 *bytes* 对象(也可称为 **字符对象**)的创建效率， *Python* 内部维护了一个 **字符缓冲池** ：

```
static PyBytesObject *characters[UCHAR_MAX + 1];
```

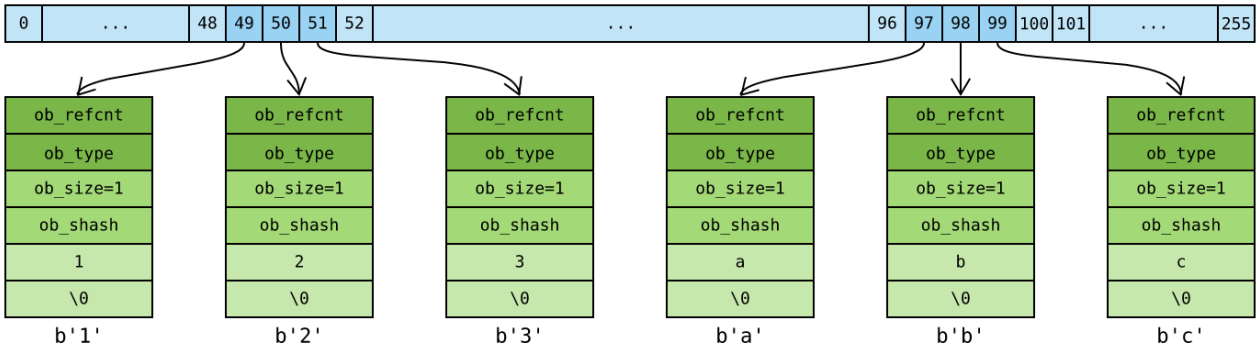
Python 内部创建单字节 *bytes* 对象时，先检查目标对象是否已在缓冲池中。*PyBytes_FromStringAndSize* 函数是负责创建 *bytes* 对象的通用接口，同样位于 *Objects/bytesobject.c* 中：

```
PyObject *
PyBytes_FromStringAndSize(const char *str, Py_ssize_t size)
{
    PyBytesObject *op;
    if (size < 0) {
        PyErr_SetString(PyExc_SystemError,
            "Negative size passed to PyBytes_FromStringAndSize");
        return NULL;
    }
    if (size == 1 && str != NULL &&
        (op = characters[*str & UCHAR_MAX]) != NULL)
    {
#ifdef COUNT_ALLOCS
        one_strings++;
#endif
        Py_INCREF(op);
        return (PyObject *)op;
    }

    op = (PyBytesObject *)_PyBytes_FromSize(size, 0);
    if (op == NULL)
        return NULL;
    if (str == NULL)
        return (PyObject *) op;

    memcpy(op->ob_sval, str, size);

    if (size == 1) {
        characters[*str & UCHAR_MAX] = op;
        Py_INCREF(op);
    }
    return (PyObject *) op;
}
```

其中，涉及字符缓冲区维护的关键步骤是：

1. **第 *10-18* 行，如果目标对象为单字节对象且已在字符缓冲池中，直接返回已缓存对象；**
2. **第 *20-26* 行，创建新 *bytes* 对象并拷贝字节序列；**
3. **第 *28-31* 行，如果创建的对象为单字节对象，将其放入字符缓冲池；**

由此可见，当 *Python* 程序开始运行时，字符缓冲池是空的。随着单字节 *bytes* 对象的创建，缓冲池中的对象慢慢多了起来。当缓冲池已缓存 `b'1'` 、 `b'2'` 、 `b'3'` 、 `b'a'` 、 `b'b'` 、 `b'c'` 这几个字符时，内部结构如下：



这样一来，字符对象首次创建后便在缓冲池中缓存起来；后续再次使用时， *Python* 直接从缓冲池中取，避免重复创建和销毁。与前面章节介绍的 **小整数** 一样，字符对象只有为数不多的 **256** 个，但使用频率非常高。缓冲池技术作为一种 **以空间换时间** 的优化手段，只需较小的内存为代价，便可明显提升执行效率。

掌握 **字符缓冲池** 的技术原理后，再也不怕面试官考察这两个代码场景了：

```
>>> a1 = b'a'
>>> a2 = b'a'
>>> a1 is a2
True
```

```
>>> ab1 = b'ab'
>>> ab2 = b'ab'
>>> ab1 is ab2
False
```

由于字符缓冲池的存在，场景一第 *3* 行直接使用已缓存的对象，不会重复创建，因此 *a1* 和 *a2* 其实是同一个对象。

小结

Python 中的字符串主要由 *str* 和 *bytes* 这两个内建对象来承载，本节我们一起研究了更靠近底层的 *bytes* 对象：

- *bytes* 是一种 **变长** 、 **不可变** 对象，内部由一个 *C* 字符数组实现；
- *bytes* 也是 **序列型对象** ，它支持的 **序列操作** 在 *bytes_as_sequence* 中定义；
- *Python* 内部维护 **字符缓冲池** 优化单字节 *bytes* 对象的创建和销毁操作；
- **缓冲池** 是一种常用的 **以空间换时间** 的优化技术 ；