# 《源码探秘 CPython》23. 字符串的相关操作

原创 古明地觉 古明地觉的编程教室 2022-02-07 09:30





## 楔子

本文来说一下字符串的操作，字符串支持哪些操作，取决于**类型对象str**，所以我们来看看**str**在底层的定义。

```
PyTypeObject PyUnicode_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "str",                    /* tp_name */
    sizeof(PyUnicodeObject),    /* tp_size */
    //...
    unicode_repr,             /* tp_repr */
    &unicode_as_number,         /* tp_as_number */
    &unicode_as_sequence,       /* tp_as_sequence */
    &unicode_as_mapping,        /* tp_as_mapping */
    //...
};
```

首先**哈希**(unicode_hash)之类的操作肯定是支持的，然后我们关注一下tp_as_number、tp_as_sequence、tp_as_mapping，我们看到三个操作簇居然都满足。

不过有了bytes的经验，我们知道tp_as_number里面实现的函数只有取模，也就是格式化。bytes和str在很多行为上都是相似的，str对象可以编码成bytes对象，bytes对象可以解码成str对象。

我们来看一下这几个操作簇。

```
//不出我们所料，只有一个取模
static PyNumberMethods unicode_as_number = {
    0,              /*nb_add*/
    0,              /*nb_subtract*/
    0,              /*nb_multiply*/
    unicode_mod,    /*nb_remainder*/
};

//我们看到和bytes对象是几乎一样的
//因为str对象和bytes都是不可变的变长对象，并且可以相互转化
//因此它们的行为是高度相似的
static PySequenceMethods unicode_as_sequence = {
    (lenfunc) unicode_length,          /* sq_length */
    PyUnicode_Concat,                /* sq_concat */
    (ssizeargfunc) unicode_repeat,     /* sq_repeat */
    (ssizeargfunc) unicode_getitem,      /* sq_item */
    0,                            /* sq_slice */
    0,                            /* sq_ass_item */
    0,                            /* sq_ass_slice */
    PyUnicode_Contains,               /* sq_contains */
};

//也和bytes对象一样
static PyMappingMethods unicode_as_mapping = {
    (lenfunc)unicode_length,        /* mp_length */
    (binaryfunc)unicode_subscript,  /* mp_subscript */
    (objobjargproc)0,           /* mp_ass_subscript */
```

```
};
```

下面我们就通过源码来考察一下。

## 字符串的相加

字符串相加会执行PyUnicode_Concat这个操作，将两个字符串组合成一个新的字符串。

```
PyObject *
PyUnicode_Concat(PyObject *left, PyObject *right)
{
    //参数left和right显然是指向字符串的指针
    //result则是指向相加之后的字符串
    PyObject *result;

    //还记得这个Py_UCS4吗， 它是相当于一个无符号32位整型
    Py_UCS4 maxchar, maxchar2;
    //left的长度、right的长度、相加之后的长度
    Py_ssize_t left_len, right_len, new_len;

    //检测是否是PyUnicodeObject
    if (ensure_unicode(left) < 0)
        return NULL;

    if (!PyUnicode_Check(right)) {
        //如果右边不是str对象的话，报错
        PyErr_Format(PyExc_TypeError,
                     "can only concatenate str (not \"%.200s\") to str",
                     right->ob_type->tp_name);
        return NULL;
    }
    //属性的初始化
    //这些都是Python内部做的检测，我们不用太关心
    if (PyUnicode_READY(right) < 0)
        return NULL;

    //这里是快分支
    //如果其中一方为空的话，那么直接返回另一方即可
    //显然这里的快分支命中率就没那么高了，但还是容易命中的
    if (left == unicode_empty)
        return PyUnicode_FromObject(right);
    if (right == unicode_empty)
        return PyUnicode_FromObject(left);

    //计算left的长度和right的长度
    left_len = PyUnicode_GET_LENGTH(left);
    right_len = PyUnicode_GET_LENGTH(right);
    //如果相加超过PY_SSIZE_T_MAX，那么会报错
    //因为要维护字符串的长度，显然长度是有范围的
    //但是几乎不存在字符串的长度会超过PY_SSIZE_T_MAX
    if (left_len > PY_SSIZE_T_MAX - right_len) {
        PyErr_SetString(PyExc_OverflowError,
                        "strings are too large to concat");
        return NULL;
    }
    //计算新的长度
    new_len = left_len + right_len;

    //计算存储单元占用的字节数
```

```
    maxchar = PyUnicode_MAX_CHAR_VALUE(left);
    maxchar2 = PyUnicode_MAX_CHAR_VALUE(right);
    //取大的那一方，比如一个是UCS2、一个是UCS4
    //那么相加之后肯定会选择UCS4
    maxchar = Py_MAX(maxchar, maxchar2);

    //通过PyUnicode_New申请能够容纳new_len个宽字符的PyUnicodeObject
    //并且字符的存储单元是大的那一方
    result = PyUnicode_New(new_len, maxchar);
    if (result == NULL)
        return NULL;
    //将left拷进去
    _PyUnicode_FastCopyCharacters(result, 0, left, 0, left_len);
    //将right拷进去
    _PyUnicode_FastCopyCharacters(result, left_len, right, 0, right_len);
    assert(_PyUnicode_CheckConsistency(result, 1));
    //返回
    return result;
}
```

我们看到逻辑还是很清晰的，不过和bytes对象不同，字符串没有实现缓冲区。但是在效率上，和bytes对象是一样的，如果有大量的字符串相加，那么效率会非常低下，官方建议仍是通过 join 的方式。

字符串的 join 对应PyUnicode_Join函数，代码比较长，这里就不贴了，但是逻辑很好理解。

就是获取列表或者元组里面的每一个unicode字符串对象的长度，然后加在一起，并取最大的存储单元，然后一次性申请对应的空间，再逐一进行拷贝。所以拷贝是避免不了的，+这种方式导致低效率的主要原因就在于大量临时PyUnicodeObject的创建和销毁。

因此如果我们要拼接大量的PyUnicodeObject，那么使用join列表或者元组的方式；如果数量不多，还是可以使用+的，毕竟维护一个列表也是需要资源的。使用join的方式，只有在PyUnicodeObject的数量非常多的时候，优势才会凸显出来。

字符串也支持索引、切片等操作，当然逻辑和bytes对象是类似的，这里就不说了，可以自己到源码中看一下。

## 字符串的encode操作

在Python里面我们可以调用字符串的encode方法，得到 bytes 对象，那么它在底层是如何实现的呢？

```
PyObject *
PyUnicode_Encode(const Py_UNICODE *s,
                 Py_ssize_t size,
                 const char *encoding,
                 const char *errors)
{
    PyObject *v, *unicode;
    //基于宽字符创建PyUnicodeObject
    unicode = PyUnicode_FromWideChar(s, size);
    if (unicode == NULL)
        return NULL;
    //编码成bytes对象，指定 encoding和 errors
    //这个Python里面的参数是一致的
    v = PyUnicode_AsEncodedString(unicode, encoding, errors);
    Py_DECREF(unicode);
    return v;
}
```

所以重点就是PyUnicode_AsEncodedString这个函数，这个函数会根据encoding参数的不同，而调用不同的函数。比如指定为utf-8，那么会调用_PyUnicode_AsUTF8String

```
PyObject *
_PyUnicode_AsUTF8String(PyObject *unicode, const char *errors)
{   //又调用了unicode_encode_utf8
    return unicode_encode_utf8(unicode, _Py_ERROR_UNKNOWN, errors);
}

static PyObject *
unicode_encode_utf8(PyObject *unicode, _Py_error_handler error_handler,
                    const char *errors)
{   //kind的类型，表示使用哪一种编码
    enum PyUnicode_Kind kind;
    void *data;
    Py_ssize_t size;
    //unicode必须是一个字符串
    if (!PyUnicode_Check(unicode)) {
        PyErr_BadArgument();
        return NULL;
    }
    //必须初始化完毕
    if (PyUnicode_READY(unicode) == -1)
        return NULL;
    //如果unicode是PyASCIIObject
    //那么直接获取每个字符的ASCII码，创建bytes对象
    if (PyUnicode_UTF8(unicode))
        return PyBytes_FromStringAndSize(PyUnicode_UTF8(unicode),
                                         PyUnicode_UTF8_LENGTH(unicode));
    //如果不是Latin-1编码，那么获取kind、data、size
    kind = PyUnicode_KIND(unicode);
    data = PyUnicode_DATA(unicode);
    size = PyUnicode_GET_LENGTH(unicode);
    // 判断 kind 是哪一种
    switch (kind) {
    default:
        Py_UNREACHABLE();
    //不同的kind执行不同的逻辑
    //最终得到的都是 bytes 对象
    case PyUnicode_1BYTE_KIND:
        assert(!PyUnicode_IS_ASCII(unicode));
        return ucs1lib_utf8_encoder(unicode, data, size, error_handler,
errors);
    case PyUnicode_2BYTE_KIND:
        return ucs2lib_utf8_encoder(unicode, data, size, error_handler,
errors);
    case PyUnicode_4BYTE_KIND:
        return ucs4lib_utf8_encoder(unicode, data, size, error_handler,
errors);
    }
}
```

整个过程还是我们所说的，通过utf-8编码将每个字符转成对应的编号，组合起来得到的就是bytes对象。

## 小结

以上我们就简单介绍了字符串的操作，当然字符串操作还有很多，比如 split、strip、title 等等，有兴趣可以进入源码中查看。看看这些操作，底层是如何使用 C 来实现的，对我们的编码水平也会有很大的帮助。

下一篇我们来介绍字符串的 intern 机制。

收录于合集 #CPython 97


上一篇
《源码探秘 CPython》24. 字符串的intern机制

下一篇
《源码探秘 CPython》22. 字符串是怎么被创建的