



微信扫一扫
关注该公众号

收录于合集

#CPython

97个 >

本文我们将讨论字符串是如何被创建的，首先CPython提供了很多的**特型API**用于字符串的创建，我们一起来看一下。

PyUnicode_FromString

最简单的一个特型API，根据 C 的 **char ***来创建Python的**unicode**

```
1 PyObject *
2 PyUnicode_FromString(const char *u)
3 {
4     // 计算C字符串的长度
5     size_t size = strlen(u);
6     // 超过了PY_SSIZE_T_MAX, 报错
7     if (size > PY_SSIZE_T_MAX) {
8         PyErr_SetString(PyExc_OverflowError, "input too long");
9         return NULL;
10    }
11    // 调用该函数进行创建, 这个函数后面介绍
12    return PyUnicode_DecodeUTF8Stateful(u, (Py_ssize_t)size, NULL, NULL);
13 }
```

所以该函数就是根据C的字符串创建Python的字符串，并且从源码中可以看到Python的字符串是由长度限制的，不能超过一个long所表示的最大范围。

PyUnicode_FromStringAndSize

也是根据 C 的 **char ***来创建Python的**unicode**，但不同的是可以指定一个长度。而在PyUnicode_FromString里面，长度写死的，就是 C 字符串的长度。

```
1 PyObject *
2 PyUnicode_FromStringAndSize(const char *u, Py_ssize_t size)
3 {
4     if (size < 0) {
5         PyErr_SetString(PyExc_SystemError,
6             "Negative size passed to PyUnicode_FromStringAndS
7 ize");
8         return NULL;
9     }
10    // 如果C的char *不为NULL, 那么还是调用这个函数创建
11    if (u != NULL)
12        return PyUnicode_DecodeUTF8Stateful(u, size, NULL, NULL);
13    else
14        // 否则的话, 直接调用 _PyUnicode_New 申请对应大小的空间
15        return (PyObject *)_PyUnicode_New(size);
16 }
```

当然还有很多其它的创建方式，比如PyUnicode_FromUnicode，根据**wchar_t ***创建Python字符串。

这里代码我们就不看了，我们使用 Cython 演示一下。

```

1 from cpython.unicode cimport PyUnicode_FromUnicode
2
3 # Py_UNICODE 理解为 C 的宽字符即可
4 cdef Py_UNICODE* s = "古明地觉的 Python小屋"
5 # 根据宽字符创建字符串, 当然也可以指定字符数量
6 # 这里我们只选择前 4 个宽字符
7 cdef str name = PyUnicode_FromUnicode(s, 4)
8 print(name) # 古明地觉

```

然后我们重点看一下PyUnicode_DecodeUTF8Stateful这个函数。

PyUnicode_DecodeUTF8Stateful

先来说一下大致的流程, 想象一下我们在Python里面创建一个字符串, 比如: `s="你好"`, 那么这个过程在 C 中就等价于将这个字符串字面量 (`const char*` 指向) 拷贝到 PyUnicode_New 所申请的堆内存中。

以上便是字符串的初始化, 而字符串的初始化是以PyUnicode_DecodeUTF8Stateful函数为起点的, 这个函数会在内部调用unicode_decode_utf8函数。

```

1 PyObject *
2 PyUnicode_DecodeUTF8Stateful(const char *s,
3                               Py_ssize_t size,
4                               const char *errors,
5                               Py_ssize_t *consumed)
6 {
7     return unicode_decode_utf8(s, size, _Py_ERROR_UNKNOWN, errors, consum
8 ed);
9 }

```

在`unicode_decode_utf8`里面会直接创建PyASCIIObject, 因为解释器会假设你创建的是纯ASCII字符串, 然后再调用`ascii_decode`进行判断, 如果成功则直接返回, 这在程序中属于快分支。

但如果我们创建的不是纯ASCII字符串, 那么会判断这个字符串到底属于PyUnicode_1BYTE_KIND、PyUnicode_2BYTE_KIND、PyUnicode_4BYTE_KIND三者的哪一种, 然后保存在 writer 的kind 成员中。

然后根据不同的类型调用不同的方法, 比如: `ucs1lib_utf8_decode`、`ucs2lib_utf8_decode` 等等。

```

1 static PyObject *
2 unicode_decode_utf8(const char *s, Py_ssize_t size,
3                     _Py_error_handler error_handler, const char *errors,
4                     Py_ssize_t *consumed)
5 {
6     _PyUnicodeWriter writer;
7     //直接使用 ascii_decode 进行解码
8     //因为会假设是一个ASCII字符串
9     writer.pos = ascii_decode(s, end, writer.data);
10    s += writer.pos;
11    //逐个字符进行扫描
12    while (s < end) {
13        //然后获取 kind
14        Py_UCS4 ch;
15        int kind = writer.kind;
16        //判断kind到底是哪一种
17        //不同的kind执行不同的函数
18        if (kind == PyUnicode_1BYTE_KIND) {
19            if (PyUnicode_IS_ASCII(writer.buffer))
20                ch = asciilib_utf8_decode(&s, end, writer.data, &writer.p

```

```

21 os);
22         else
23             ch = ucs1lib_utf8_decode(&s, end, writer.data, &writer.pos);
24     s);
25     } else if (kind == PyUnicode_2BYTE_KIND) {
26         ch = ucs2lib_utf8_decode(&s, end, writer.data, &writer.pos);
27     } else {
28         assert(kind == PyUnicode_4BYTE_KIND);
29         ch = ucs4lib_utf8_decode(&s, end, writer.data, &writer.pos);
30     }
31     .....
32     .....
    }

```

所以能看出创建字符串的过程就是将 **char *** 指向的字节序列进行解码、然后拷贝的一个过程，因为解释器会假设 **char *** 指向的是一个简单的 ASCII 字节序列。

然后按照 `PyUnicode_1BYTE_KIND` 的模式去计算字符串所需的堆内存空间，而后续的字节编码检测算法发现该 C 级别的字节序列出现了 **maxchar** 大于 255 的字符，就会尝试以 `PyUnicode_2BYTE_KIND` 的模式，再次重新计算内存空间并调用 `malloc` 分配函数。

因此所以创建一个字符串，涉及 1 到 3 次的 `malloc`，效率还是比较低下的。

```
%timeit s = f"{'a' * 1000000} "
```

701 µs ± 132 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

```
%timeit s = f"{'a' * 1000000} 憨"
```

1.24 ms ± 20.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

```
%timeit s = f"{'a' * 1000000} 🍌"
```

2.23 ms ± 21 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

古明地觉的 Python 小屋

以上就是字符串的创建，事实上字符串还是很复杂的，它并没有我们想象的那么简单。

收录于合集 [#CPython 97](#)

[← 上一篇](#)

《源码探秘 CPython》23. 字符串的相关操作

[下一篇 >](#)

《源码探秘 CPython》21. 字符串的底层结构

喜欢此内容的人还喜欢

358. K 距离间隔重排字符串 排序

钰娘娘知识汇总



Perl笔记-字符串和排序与流程管理

生物信息小记

