《源码探秘 CPython》20. Python是怎么存储字符串的?

原创 古明地觉 古明地觉的编程教室 2022-01-30 09:30

楔子

在上一篇中我们提到了unicode，该字符集对世界上的文字进行了系统的整理，让计算机可以用统一的方式处理文本，而且目前已经支持超过13万个字符，天然地支持多国语言。

所以不管什么文字，都可以用一个unicode来表示。

但是问题来了，unicode能表示这么多的字符，那么占用的内存一定不低吧。的确，根据当时的编码，一个unicode字符最高会占用到4字节。但是对于西方人来说，明明一个字符就够用了，为啥需要那么多。

于是又出现了utf-8，它是为unicode提供的一个新的编码规则，具有可变长的功能。不同种类的字符占用的大小不同，比如英文字符使用一个字节存储，汉字使用3个字节存储，Emoji 使用4个字节存储。

但Python在表示unicode字符串时，使用的却不是utf-8编码，至于原因我们下面来分析一下。

unicode 的三种编码

从Python3开始，字符串使用的是Unicode。而根据编码的不同，Unicode的每个字符最大可以占到4字节，从内存的角度来说， 这种编码有时会比较昂贵。

为了减少内存消耗并且提高性能，Python的内部使用了三种编码方式来表示Unicode。

- `Latin-1 编码:每个字符一字节;`
- `UCS2 编码:每个字符两字节;`
- `UCS4 编码:每个字符四字节;`

在Python编程中，所有字符串的行为都是一致的，而且大多数时间我们都没有注意到差异。然而在处理大文本的时候，这种差异就会变得异常显著、甚至有些让人出乎意料。

为了看到内部表示的差异，我们使用sys.getsizeof函数，查看一个对象所占的字节数。

```
import sys

print(sys.getsizeof("a"))  # 50
print(sys.getsizeof("憨"))  # 76
print(sys.getsizeof("𠮷"))  # 80
```

我们看到都是一个字符，但是它们占用的内存却是不一样的。因为Python面对不同的字符会采用不同的编码，进而导致大小不同。

但需要注意的是，Python的每一个字符串都需要额外占用49-80字节，因为要存储一些**额外的信息**，比如：公共的头部、哈希、长度、字节长度、编码类型等等。

```
import sys

# 对于ASCII字符，一个占1字节，显然此时编码是Latin-1编码
print(sys.getsizeof("ab") - sys.getsizeof("a"))  # 1

# 对于汉字，日文等等，一个占用2字节，此时是UCS2编码
```

```
print(sys.getsizeof("憨憨") - sys.getsizeof("憨"))  # 2
print(sys.getsizeof("です") - sys.getsizeof("で"))  # 2

# 像emoji，则是一个占4字节，此时是UCS4编码
print(sys.getsizeof("🤣🤣") - sys.getsizeof("🤣"))  # 4
```

而采用不同的编码，那么底层结构体实例的额外部分也会占用不同大小的内存。

如果编码是**Latin-1**，那么这个结构体实例额外的部分会占**49个字节**；编码是**UCS2**，占**74个字节**；编码是**UCS4**，占**76个字节**。然后字符串所占的字节数就等于：**额外的部分 + 字符个数 * 单个字符所占的字节**。

```
import sys

# 所以一个空字符串占用49个字节
# 此时会采用占用内存最小的Latin-1编码
print(sys.getsizeof(""))  # 49

# 此时使用UCS2
print(sys.getsizeof("憨") - 2)  # 74

# UCS4
print(sys.getsizeof("🤣") - 4)  # 76
```

为什么不使用utf-8编码

上面提到的三种编码，是Python在底层所使用的，但我们知道unicode还有一个utf-8编码，那Python为啥不用呢？

先来抛出一个问题：首先我们知道Python支持通过索引查找一个字符串指定位置的字符，而且Python默认是以字符为单位的，**不是字节**(我们后面还会提)，比如**s[2]**搜索的就是字符串s中的**第3个字符**。

```
s = "古明地觉"
print(s[2])  # 地
```

那么问题来了，我们知道通过索引查找字符串的某个字符，时间复杂度为O(1)，那么Python是怎么通过索引瞬间定位到指定字符的呢？

显然是通过指针的偏移，用索引乘上每个字符占的字节数，得到偏移量，然后从头部向后偏移指定数量的字节即可，这样就能在定位到指定字符的同时还保证时间复杂度为O(1)。

但是这需要一个前提：**字符串中每个字符所占的大小必须是相同的**，如果字符占的大小不同，比如有的占1字节、有的占3字节，显然就无法通过指针偏移的方式了。这个时候若还想准确定位的话，只能按顺序对所有字符都逐个扫描，但这样的话时间复杂度肯定不是O(1)，而是O(n)。

我们以Go为例，Go的字符串默认就是使用的utf-8编码。

```
package main

import (
    "fmt"
)

func main() {
    s := "古明地觉"
    fmt.Println(s[2])  // 164
    fmt.Println(string(s[2]))  // ¤
```

```
11 }
```

惊了，我们看到打印的并不是我们希望的结果。因为Go底层使用的是utf-8编码，不同的字符可能会占用不同的字节。但是Go通过索引定位的时候，时间复杂度也是O(1)，所以定位的时候是以字节为单位、而不是字符。在获取的时候也只会获取一个字节，而不是一个字符。

所以**s[2]**在Go里面指的是**第3个字节**，而不是**第3个字符**，而汉字在utf-8编码下占3个字节，所以s[2]指的就是汉字**古**的第三个字节。我们看到打印的时候，该字节存的值为**164**。

```
1 s = "古明地觉"
2 print(s.encode("utf-8")[2])  # 164
```

这就是采用utf-8编码带来的弊端，它无法让我们以O(1)的时间复杂度去准确地定位字符，尽管它在存储的时候更加的省内存。

Latin-1、UCS2、UCS4该使用哪一种?

我们说Python会使用3种编码来表示unicode，所占字节大小分别是1、2、4字节。

因此Python在创建字符串的时候，会先扫描，尝试使用占字节数最少的Latin-1编码存储，但是范围肯定有限。如果发现了存储不下的字符，只能改变编码，使用UCS2，继续扫描。但是又发现了新的字符，这个字符UCS2也无法存储，因为两个字节最多存储65535个不同的字符，所以会再次改变编码，使用UCS4。UCS4占四个字节，肯定能存下了。

一旦改变编码，字符串中的所有字符都会使用同样的编码，因为它们不具备可变长功能。比如这个字符串：**"hello古明地觉"**，肯定都会使用UCS2，不存在说**hello**使用Latin1，**古明地觉**使用UCS2，因为一个字符串只能有一个编码。

当通过索引获取的时候，会将索引乘上每个字符占的字节数，这样就能跳到准确位置上，因为字符串里面的所有字符占用的字节都是一样的，然后获取的时候也会获取指定的字节数。比如：使用UCS2编码，那么定位到某个字符的时候，会取两个字节，这样才能表示一个完整的字符。

```
1  import sys
2
3  # 此时全部是ascii字符，那么Latin-1编码可以存储
4  # 所以结构体实例额外的部分占49个字节
5  s1 = "hello"
6  # 有5个字符，一个字符一个字节，所以加一起是54个字节
7  print(sys.getsizeof(s1))  # 54
8
9
10 # 出现了汉字，那么Latin-1肯定存不下，于是使用UCS2
11 # 所以此时结构体实例额外的部分占74个字节
12 # 但是别忘了此时的英文字符也是ucs2，所以也是一个字符两字节
13 s2 = "hello憨"
14 # 6个字符，74 + 6 * 2 = 86
15 print(sys.getsizeof(s2))  # 86
16
17
18 # 这个牛逼了，ucs2也存不下，只能ucs4存储了
19 # 所以结构体实例额外的部分占76个字节
20 s3 = "hello憨🤣"
21 # 此时所有字符一个占4字节，7个字符
22 # 76 + 7 * 4 = 104
23 print(sys.getsizeof(s3))  # 104
```

除此之外，我们再举一个例子更形象地证明这个现象。

```
import sys

s1 = "a" * 1000
s2 = "a" * 1000 + "𠮷"

# 我们看到s2只比s1多了一个字符
# 但是两者占的内存，s2却将近是s1的四倍。
print(sys.getsizeof(s1), sys.getsizeof(s2))  # 1049 4080
```

我们知道s2和s1的差别只是s2比s1多了一个字符，但就是这么一个字符导致s2比s1多占了3031个字节。然而这3031个字节不可能是多出来的字符所占的大小，什么字符一个会占到三千多个字节，这是不可能的。

尽管如此，但它也是罪魁祸首，不过前面的1000个字符也是共犯。我们说Python会根据字符串选择不同的编码，s1全部是ascii字符，所以Latin1能存下，因此一个字符只占一个字节。所以大小就是49 + 1000 = 1049。

但是对于s2，Python发现前1000个字符Latin1能存下，不幸的是最后一个字符存不下，于是只能使用UCS4。而字符串的所有字符只能有一个编码，为了保证索引查找的时间复杂度为O(1)，前面一个字节就能存下的字符，也需要用4字节来存储。这是Python的设计策略。

而我们说使用UCS4，结构体额外的部分会占76个字节，因此s2的大小就是：**7 6 + 1001 * 4 = 4080**

```
print(sys.getsizeof("爷的青春回来了"))  # 88
print(sys.getsizeof("𠮷的青春回来了"))  # 104
```

字符数量相同但是占用内存大小不同，相信原因你肯定能分析出来。

所以如果字符串中的所有字符都是ASCII字符，则使用1字节Latin1对其进行编码。基本上，Latin1能表示前256个Unicode字符，它支持多种拉丁语，如英语、瑞典语、意大利语、挪威语。但是它们不能存储非拉丁语言，比如汉语、日语、希伯来语、西里尔语。这是因为它们的代码点(数字索引)定义在1字节(0-255)范围之外。

大多数流行的自然语言都可以采用2字节(UCS2)编码，但当字符串包含特殊符号、emoji或稀有语言时，则使用4字节(UCS4)编码。Unicode标准有将近300个块(范围)，你可以在0XFFFF块之后找到4字节块。

假设我们有一个10G的ASCII文本，我们想把它加载到内存中，但如果我们在文本中插入一个表情符号，那么字符串的大小将增加4倍。这是一个巨大的差异，你可能会在实践当中遇到，比如处理NLP问题。

```
print(ord("a"))  # 97
print(ord("憨"))  # 25000
print(ord("𠮷"))  # 128187
```

所以最著名和最流行的Unicode编码都是utf-8，但是Python不在内部使用它，而是使用Latin1、UCS2、UCS4。至于原因我们上面已经解释的很清楚了，主要是Python的索引是基于字符，而不是字节。

> 当一个字符串使用utf-8编码存储时，每个字符会根据自身选择一个合适的大小。这是一种存储效率很高的编码，但是它有一个明显的缺点。由于每个字符的字节长度可能不同，就导致无法按照索引瞬间定位到单个字符，即便能定位，也无法定位准确。如果想准，那么只能逐个扫描所有字符。

假设要对使用utf-8编码的字符串执行一个简单的操作，比如s[5]，就意味着Python需要扫描每一个字符，直到找到需要的字符，这样效率是很低的。

但如果是固定长度的编码就没有这样的问题，所以当Latin 1存储的**hello**，在和UCS2存储的**古明地觉**组合之后，整体每一个字符都会向大的方向扩展、变成了2字节。

这样定位字符的时候，只需要将**索引 * 2**便可计算出偏移的字节数、然后跳转该字节数即可。但如果原来的**hello**还是一个字节、而汉字是2字节，那么只通过索引是不可能定位到准确字符的，因为不同类型字符的大小不同，必须要扫描整个字符串才可以。但是扫描字符串，效率又比较低，所以Python内部才会使用这个方法，而不是使用utf-8。

所以对于Go来讲，如果想像Python一样，那么需要这么做：

```go
package main

import (
    "fmt"
)

func main() {
    s := "hello古明地觉"
    //我们看到长度为17，因为它使用utf-8编码
    fmt.Println(s, len(s)) // hello古明地觉 17

    //如果想像Python一样
    //那么Go提供了一个rune，相当于int32
    //此时每个字符均使用4个字节，所以长度变成了9
    r := []rune(s)
    fmt.Println(string(r), len(r)) // hello古明地觉 9
    //虽然打印的内容是一样的，但是此时每个字符都使用4字节存储

    //此时跳转会和Python一样偏移 5 * 4 个字节
    //然后获取也会获取4个字节，因为一个字符占4个字节
    fmt.Println(string(r[5])) //古
}
```

所以utf-8编码的unicode字符串里面的字符可能占用不同的字节，显然没办法实现当前Python字符串的索引查找效果，因此Python没有使用utf-8编码。

Python的做法是让字符串的所有字符都占用相同的字节，先使用占用内存最小的Latin1，不行的话再使用UCS2、UCS4，总之会确保每个字符占用的字节是一样的。至于原因的话我们上面分析的很透彻了，因为无论是索引还是切片、还是计算长度等等，都是基于字符来的，显然这也符合人类的思维习惯。

小结

以上就是Python字符串的存储策略，它并没有使用最为流行的utf-8，归根结底就在于这种编码不适合Python的字符串。当然，我们在将字符串转成字节序列的时候，一般使用的都是utf-8编码。

下一篇来介绍字符串的底层实现，看看字符串在底层是如何设计的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》21. 字符串的底层结构

下一篇 ›
《源码探秘 CPython》19. 字符集和字符编码

象的某个属性值

Seven的代码实验室

python 7天进阶之路-对象和json转换

缪斯之子