Python编码和Unicode

node+mongodb 建站攻略（一期）
JS插件开发之LightBox图片画廊(上)
JS插件开发之LightBox图片画廊(下)
谈谈CSS性能

我确定有很多关于Unicode和Python的说明，但为了方便自己的理解使用，我还是打算再写一些关于它们的东西。

字节流 vs Unicode对象

我们先来用Python定义一个字符串。当你使用string类型时，实际上会储存一个字节串。

```
[  a ][  b ][  c ] = "abc"
[ 97 ][ 98 ][ 99 ] = "abc"
```

在这个例子里，abc这个字符串是一个字节串。97.，98,，99是ASCII码。Python 2.x版本的一个不足之处就是默认将所有的字符串当做ASCII来对待。不幸的是，ASCII在拉丁式字符集里是最不常见的标准。

ASCII是用前127个数字来做字符映射。像windows-1252和UTF-8这样的字符映射有相同的前127个字符。在你的字符串里每个字节的值低于127的时候是安全的混合字符串编码。然而作这个假设是件很危险的事情，下面还将会提到。

当你的字符串里有字节的值大于126的时候就会出现问题了。我们来看一个用windows-1252编码的字符串。Windows-1252里的字符映射是8位的字符映射，那么总共就会有256个字符。前127个跟ASCII是一样的，接下来的127个是由windows-1252定义的其他字符。

```
A windows-1252 encoded string looks like this:
[ 97 ] [ 98 ] [ 99 ] [ 150 ] = "abc-"
```

Windows-1252仍然是一个字节串，但你有没有看到最后一个字节的值是大于126的。如果Python试着用默认的ASCII标准来解码这个字节流，它就会报错。我们来看当Python解码这个字符串的时候会发生什么：

```
>>> x = "abc" + chr(150)
>>> print repr(x)
'abc\x96'
>>> u"Hello" + x
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeDecodeError: 'ASCII' codec can't decode byte 0x96 in position
```

我们来用UTF-8来编码另一个字符串：

```
A UTF-8 encoded string looks like this:
[ 97 ] [ 98 ] [ 99 ] [ 226 ] [ 128 ] [ 147 ] = "abc-"
[0x61] [0x62] [0x63] [0xe2]  [ 0x80] [ 0x93] = "abc-"
```

如果你拿起看你熟悉的Unicode编码表，你会发现英文的破折号对应的Unicode编码点为8211（0×2013）。这个值大于ASCII最大值127。大于一个字节能够存储的值。因为8211（0×2013）是两个字节，UTF-8必须利用一些技巧告诉系统存储一个字符需要三个字节。我们再来看当Python准备用默认的ASCII来编码一个里面有字符的值大于126的UTF-8编码字符串。

```
>>> x = "abc\xe2\x80\x93"
>>> print repr(x)
'abc\xe2\x80\x93'
>>> u"Hello" + x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeDecodeError: 'ASCII' codec can't decode byte 0xe2 in position
```

你可以看到，Python一直是默认使用ASCII编码。当它处理第4个字符的时候，因为它的值为226大于126，所以Python抛出了错误。这就是混合编码所带来的问题。

解码字节流

在一开始学习Python Unicode 的时候，解码这个术语可能会让人很疑惑。你可以把字节流解码成一个Unicode对象，把一个Unicode 对象编码为字节流。

Python需要知道如何将字节流解码为Unicode对象。当你拿到一个字节流，你调用它的“解码方法来从它创建出一个Unicode对象。

你最好是尽早的将字节流解码为Unicode。

```
>>> x = "abc\xe2\x80\x93"
>>> x = x.decode("utf-8")
>>> print type(x)
<type 'unicode'>
>>> y = "abc" + chr(150)
>>> y = y.decode("windows-1252")
>>> print type(y)
>>> print x + y
abc–abc–
```

将Unicode编码为字节流

Unicode对象是一个文本的编码不可知论的代表。你不能简单地输出一个Unicode对象。它必须在输出前被变成一个字节串。Python会很适合做这样的工作，尽管Python将Unicode编码为字节流时默认是适用ASCII，这个默认的行为会成为很多让人头疼的问题的原因。

```
>>> u = u"abc\u2013"
>>> print u
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\u2013' i
>>> print u.encode("utf-8")
abc–
```

使用codecs模块

codecs模块能在处理字节流的时候提供很大帮助。你可以用定义的编码来打开文件并且你从文件里读取的内容会被自动转化为Unicode对象。

试试这个：

```
>>> import codecs
>>> fh = codecs.open("/tmp/utf-8.txt", "w", "utf-8")
>>> fh.write(u"\u2013")
>>> fh.close()
```

它所做的就是拿到一个Unicode对象然后将它以utf-8编码写入到文件。你也可以在其他的情况下这么使用它。

试试这个：

当从一个文件读取数据的时候，codecs.open 会创建一个文件对象能够自动将utf-8编码文件转化为一个Unicode对象。

我们接着上面的例子，这次使用urllib流。

```
>>> stream = urllib.urlopen("http://www.google.com")
>>> Reader = codecs.getreader("utf-8")
>>> fh = Reader(stream)
>>> type(fh.read(1))
<type 'unicode'>
>>> Reader
<class encodings.utf_8.StreamReader at 0xa6f890>
```

单行版本：

```
>>> fh = codecs.getreader("utf-8")(urllib.urlopen("http://www.google
>>> type(fh.read(1))
```

你必须对codecs模块十分小心。你传进去的东西必须是一个Unicode对象，否则它会自动将字节流作为ASCII进行解码。

```
>>> x = "abc\xe2\x80\x93"
# our "abc-" utf-8 string
>>> fh = codecs.open("/tmp/foo.txt", "w", "utf-8")
>>> fh.write(x)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python2.5/codecs.py", line 638, in write
  return self.writer.write(data)
File "/usr/lib/python2.5/codecs.py", line 303, in write
  data, consumed = self.encode(object, self.errors)
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in positic
```

哎呦我去，Python又开始用ASCII来解码一切了。

将UTF-8字节流切片的问题

因为一个UTF-8编码串是一个字节列表，len( )和切片操作无法正常工作。首先用我们之前用的字符串。

```
[ 97 ] [ 98 ] [ 99 ] [ 226 ] [ 128 ] [ 147 ] = "abc–"
```

接下来做以下的：

```
>>> my_utf8 = "abc–"
>>> print len(my_utf8)
6
```

神马？它看起来是4个字符，但是len的结果说是6。因为len计算的是字节数而不是字符数。

```
>>> print repr(my_utf8)
'abc\xe2\x80\x93'
```

现在我们来切分这个字符串。

```
>>> my_utf8[-1]
# Get the last char
'\x93'
```

我去，切分结果是最后一字节，不是最后一个字符。

为了正确的切分UTF-8，你最好是解码字节流创建一个Unicode对象。然后就能安全的操作和计数了。

```
>>> my_unicode = my_utf8.decode("utf-8")
>>> print repr(my_unicode)
u'abc\u2013'
>>> print len(my_unicode)
4
>>> print my_unicode[-1]
–
```

当Python自动地编码/解码

在一些情况下，当Python自动地使用ASCII进行编码/解码的时候会抛出错误。

第一个案例是当它试着将Unicode和字节串合并在一起的时候。

```
>>> u"" + u"\u2019".encode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in position
```

在合并列表的时候会发生同样的情况。Python在列表里有string和Unicode对象的时候会自动地将字节串解码为Unicode。

```
>>> ",".join([u"This string\u2019s unicode", u"This string\u2019s ut
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in position
```

或者当试着格式化一个字节串的时候：

```
>>> "%s\n%s" % (u"This string\u2019s unicode", u"This string\u2019s
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in position
```

基本上当你把Unicode和字节串混在一起用的时候，就会导致出错。

在这个例子里面，你创建一个utf-8文件，然后往里面添加一些Unicode对象的文本。就会报UnicodeDecodeError错误。

```
>>> buffer = []
>>> fh = open("utf-8-sample.txt")
>>> buffer.append(fh.read())
>>> fh.close()
>>> buffer.append(u"This string\u2019s unicode")
>>> print repr(buffer)
['This file\xe2\x80\x99s got utf-8 in it\n', u'This string\u2019s u
>>> print "\n".join(buffer)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in positic
```

你可以使用codecs模块把文件作为Unicode加载来解决这个问题。

```
>>> import codecs
>>> buffer = []
>>> fh = open("utf-8-sample.txt", "r", "utf-8")
>>> buffer.append(fh.read())
>>> fh.close()
>>> print repr(buffer)
[u'This file\u2019s got utf-8 in it\n', u'This string\u2019s unicod
>>> buffer.append(u"This string\u2019s unicode")
>>> print "\n".join(buffer)
This file’s got utf-8 in it

This string’s unicode
```

正如你看到的，由codecs.open 创建的流在当数据被读取的时候自动地将比特串转化为Unicode。

最佳实践

1．最先解码，最后编码

2．默认使用utf-8编码

3．使用codecs和Unicode对象来简化处理

最先解码意味着无论何时有字节流输入，需要尽早将输入解码为Unicode。这会防止出现len( )和切分utf-8字节流发生问题。

最后编码意味着只有你打算将文本输出到某个地方时，才把它编码为字节流。这个输出可能是一个文件，一个数据库，一个socket等等。只有在处理完成之后才编码unicode对象。最后编码也意味着，不要让Python为你编码Unicode对象。Python将会使用ASCII编码，你的程序会崩溃。

默认使用UTF-8编码意味着：因为UTF-8可以处理任何Unicode字符，所以你最好用它来替代windows-1252和ASCII。

codecs模块能够让我们在处理诸如文件或socket这样的流的时候能少踩一些坑。如果没有codecs提供的这个工具，你就必须将文件内容读取为字节流，然后将这个字节流解码为Unicode对象。

codecs模块能够让你快速的将字节流转化为Unicode对象，省去很多麻烦。

解释UTF-8

最后的部分是让你能对UTF-8有一个入门的了解，如果你是个超级极客可以无视这一段。

利用UTF-8，任何在127和255之间的字节是特别的。这些字节告诉系统这些字节是多字节序列的一部分。

```
Our UTF-8 encoded string looks like this:
[ 97 ] [ 98 ] [ 99 ] [ 226 ] [ 128 ] [ 147 ] = "abc–"
```

最后3字节是一个UTF-8多字节序列。如果你把这三个字节里的第一个转化为2进制可以看到以下的结果：

```
11100010
```

前3比特告诉系统它开始了一个3字节序列226，128，147。

那么完整的字节序列。

```
11100010 10000000 10010011
```

然后你对三字节序列运用下面的掩码。（详见这里）

```
1110xxxx 10xxxxxx 10xxxxxx
XXXX0010 XX000000 XX010011 Remove the X's
0010       000000   010011 Collapse the numbers
00100000 00010011          Get Unicode number 0x2013, 8211 The "–"
```

这里仅仅是关于UTF-8的一些入门的基本知识，如果想知道更多的细节，可以去看UTF-8的维基页面。