15.14 传递Unicode字符串给C函数库¶

问题¶

你要写一个扩展模块，需要将一个Python字符串传递给C的某个库函数，但是这个函数不知道该怎么处理Unicode。

解决方案¶

这里我们需要考虑很多的问题，但是最主要的问题是现存的C函数库并不理解Python的原生Unicode表示。 因此，你的挑战是将Python字符串转换为一个能被C理解的形式。

为了演示的目的，下面有两个C函数，用来操作字符串数据并输出它来调试和测试。 一个使用形式为 `char *, int` 形式的字节， 而另一个使用形式为 `wchar_t *, int` 的宽字符形式：

```
void print_chars(char *s, int len) {
  int n = 0;

  while (n < len) {
    printf("%2x ", (unsigned char) s[n]);
    n++;
  }
  printf("\n");
}

void print_wchars(wchar_t *s, int len) {
  int n = 0;
  while (n < len) {
    printf("%x ", s[n]);
    n++;
  }
  printf("\n");
}
```

对于面向字节的函数 `print_chars()` ，你需要将Python字符串转换为一个合适的编码比如UTF-8. 下面是一个这样的扩展函数例子：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;
  Py_ssize_t  len;

  if (!PyArg_ParseTuple(args, "s#", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  Py_RETURN_NONE;
}
```

对于那些需要处理机器本地 `wchar_t` 类型的库函数，你可以像下面这样编写扩展代码：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  wchar_t *s;
  Py_ssize_t  len;

  if (!PyArg_ParseTuple(args, "u#", &s, &len)) {
```

```
  return NULL;
 }
 print_wchars(s,len);
 Py_RETURN_NONE;
}
```

下面是一个交互会话来演示这个函数是如何工作的：

```
>>> s = 'Spicy Jalape\u00f1o'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>>
```

仔细观察这个面向字节的函数 `print_chars()` 是怎样接受UTF-8编码数据的， 以及 `print_wchars()` 是怎样接受Unicode编码值的

讨论¶

在继续本节之前，你应该首先学习你访问的C函数库的特征。 对于很多C函数库，通常传递字节而不是字符串会比较好些。要这样做，请使用如下的转换代码：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
 char *s;
 Py_ssize_t  len;

 /* accepts bytes, bytearray, or other byte-like object */
 if (!PyArg_ParseTuple(args, "y#", &s, &len)) {
  return NULL;
 }
 print_chars(s, len);
 Py_RETURN_NONE;
}
```

如果你仍然还是想要传递字符串， 你需要知道Python 3可使用一个合适的字符串表示， 它并不直接映射到使用标准类型 `char *` 或 `wchar_t *` （更多细节参考PEP 393）的C函数库。 因此，要在C中表示这个字符串数据，一些转换还是必须要的。 在 `PyArg_ParseTuple()` 中使用”s#” 和”u#”格式化码可以安全的执行这样的转换。

不过这种转换有个缺点就是它可能会导致原始字符串对象的尺寸增大。 一旦转换过后，会有一个转换数据的复制附加到原始字符串对象上面，之后可以被重用。 你可以观察下这种效果：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
103
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>> sys.getsizeof(s)
163
>>>
```

对于少量的字符串对象，可能没什么影响， 但是如果你需要在扩展中处理大量的文本，你可能想避免这个损耗了。 下面是一个修订版本可以避免这种内存损耗：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *obj, *bytes;
  char *s;
  Py_ssize_t   len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  bytes = PyUnicode_AsUTF8String(obj);
  PyBytes_AsStringAndSize(bytes, &s, &len);
  print_chars(s, len);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

而对 `wchar_t` 的处理时想要避免内存损耗就更加难办了。 在内部，Python使用最高效的表示来存储字符串。 例如，只包含ASCII的字符串被存储为字节数组， 而包含范围从U+0000到U+FFFF的字符的字符串使用双字节表示。 由于对于数据的表示形式不是单一的，你不能将内部数组转换为 `wchar_t*` 然后期望它能正确的工作。 你应该创建一个 `wchar_t` 数组并向其中复制文本。 `PyArg_ParseTuple()` 的”u#”格式码可以帮助你高效的完成它（它将复制结果附加到字符串对象上）。

如果你想避免长时间内存损耗，你唯一的选择就是复制Unicode数据懂啊一个临时的数组， 将它传递给C函数，然后回收这个数组的内存。下面是一个可能的实现：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  PyObject *obj;
  wchar_t *s;
  Py_ssize_t len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  if ((s = PyUnicode_AsWideCharString(obj, &len)) == NULL) {
    return NULL;
  }
  print_wchars(s, len);
  PyMem_Free(s);
  Py_RETURN_NONE;
}
```

在这个实现中， `PyUnicode_AsWideCharString()` 创建一个临时的wchar_t缓冲并复制数据进去。 这个缓冲被传递给C然后被释放掉。 但是我写这本书的时候，这里可能有个bug，后面的Python问题页有介绍。

如果你知道C函数库需要的字节编码并不是UTF-8， 你可以强制Python使用扩展码来执行正确的转换，就像下面这样：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s = 0;
  int   len;
  if (!PyArg_ParseTuple(args, "es#", "encoding-name", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
```

```
  print_chars(s, len);
  PyMem_Free(s);
  Py_RETURN_NONE;
}
```

最后，如果你想直接处理Unicode字符串，下面的是例子，演示了底层操作访问：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  PyObject *obj;
  int n, len;
  int kind;
  void *data;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  if (PyUnicode_READY(obj) < 0) {
    return NULL;
  }

  len = PyUnicode_GET_LENGTH(obj);
  kind = PyUnicode_KIND(obj);
  data = PyUnicode_DATA(obj);

  for (n = 0; n < len; n++) {
    Py_UCS4 ch = PyUnicode_READ(kind, data, n);
    printf("%x ", ch);
  }
  printf("\n");
  Py_RETURN_NONE;
}
```

在这个代码中， `PyUnicode_KIND()` 和 `PyUnicode_DATA()` 这两个宏和Unicode的可变宽度存储有关，这个在PEP 393中有描述。 `kind` 变量编码底层存储（8位、16位或32位）以及指向缓存的数据指针相关的信息。 在实际情况中，你并不需要知道任何跟这些值有关的东西， 只需要在提取字符的时候将它们传给 `PyUnicode_READ()` 宏。

还有最后几句：当从Python传递Unicode字符串给C的时候，你应该尽量简单点。 如果有UTF-8和宽字符两种选择，请选择UTF-8. 对UTF-8的支持更加普遍一些，也不容易犯错，解释器也能支持的更好些。 最后，确保你仔细阅读了 关于处理Unicode的相关文档