15.3 编写扩展函数操作数组¶

问题¶

你想编写一个C扩展函数来操作数组，可能是被array模块或类似Numpy库所创建。 不过，你想让你的函数更加通用，而不是针对某个特定的库所生成的数组。

解决方案¶

为了能让接受和处理数组具有可移植性，你需要使用到 *Buffer Protocol* . 下面是一个手写的C扩展函数例子， 用来接受数组数据并调用本章开篇部分的 `avg(double *buf, int len)` 函数：

```
/* Call double avg(double *, int) */
static PyObject *py_avg(PyObject *self, PyObject *args) {
  PyObject *bufobj;
  Py_buffer view;
  double result;
  /* Get the passed Python object */
  if (!PyArg_ParseTuple(args, "O", &bufobj)) {
    return NULL;
  }

  /* Attempt to extract buffer information from it */

  if (PyObject_GetBuffer(bufobj, &view,
      PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    return NULL;
  }

  if (view.ndim != 1) {
    PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional array");
    PyBuffer_Release(&view);
    return NULL;
  }

  /* Check the type of items in the array */
  if (strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    PyBuffer_Release(&view);
    return NULL;
  }

  /* Pass the raw buffer and size to the C function */
  result = avg(view.buf, view.shape[0]);

  /* Indicate we're done working with the buffer */
  PyBuffer_Release(&view);
  return Py_BuildValue("d", result);
}
```

下面我们演示下这个扩展函数是如何工作的：

```
>>> import array
>>> avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> avg(numpy.array([1.0,2.0,3.0]))
2.0
>>> avg([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' does not support the buffer interface
>>> avg(b'Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected an array of doubles
```

```
>>> a = numpy.array([[1.,2.,3.],[4.,5.,6.]])
>>> avg(a[:,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ndarray is not contiguous
>>> sample.avg(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected a 1-dimensional array
>>> sample.avg(a[0])

2.0
>>>
```

讨论¶

将一个数组对象传给C函数可能是一个扩展函数做的最常见的事。 很多Python应用程序，从图像处理到科学计算，都是基于高性能的数组处理。 通过编写能接受并操作数组的代码，你可以编写很好的兼容这些应用程序的自定义代码，而不是只能兼容你自己的代码。

代码的关键点在于 `PyBuffer_GetBuffer()` 函数。 给定一个任意的Python对象，它会试着去获取底层内存信息，它简单的抛出一个异常并返回-1. 传给 `PyBuffer_GetBuffer()` 的特殊标志给出了所需的内存缓冲类型。 例如，`PyBUF_ANY_CONTIGUOUS` 表示是一个连续的内存区域。

对于数组、字节字符串和其他类似对象而言，一个 `Py_buffer` 结构体包含了所有底层内存的信息。 它包含一个指向内存地址、大小、元素大小、格式和其他细节的指针。下面是这个结构体的定义：

```
typedef struct bufferinfo {
    void *buf;              /* Pointer to buffer memory */
    PyObject *obj;          /* Python object that is the owner */
    Py_ssize_t len;         /* Total size in bytes */
    Py_ssize_t itemsize;    /* Size in bytes of a single item */
    int readonly;           /* Read-only access flag */
    int ndim;               /* Number of dimensions */
    char *format;           /* struct code of a single item */
    Py_ssize_t *shape;      /* Array containing dimensions */
    Py_ssize_t *strides;    /* Array containing strides */
    Py_ssize_t *suboffsets; /* Array containing suboffsets */
} Py_buffer;
```

本节中，我们只关注接受一个双精度浮点数数组作为参数。 要检查元素是否是一个双精度浮点数，只需验证 `format` 属性是不是字符串”d”. 这个也是 `struct` 模块用来编码二进制数据的。 通常来讲，`format` 可以是任何兼容 `struct` 模块的格式化字符串， 并且如果数组包含了C结构的话它可以包含多个值。 一旦我们已经确定了底层的缓存区信息，那只需要简单的将它传给C函数，然后会被当做是一个普通的C数组了。 实际上，我们不必担心是怎样的数组类型或者它是被什么库创建出来的。 这也是为什么这个函数能兼容 `array` 模块也能兼容 `numpy` 模块中的数组了。

在返回最终结果之前，底层的缓冲区视图必须使用 `PyBuffer_Release()` 释放掉。 之所以要这一步是为了能正确的管理对象的引用计数。

同样，本节也仅仅只是演示了接受数组的一个小的代码片段。 如果你真的要处理数组，你可能会碰到多维数据、大数据、不同的数据类型等等问题， 那么就得去学更高级的东西了。你需要参考官方文档来获取更多详细的细节。

如果你需要编写涉及到数组处理的多个扩展，那么通过Cython来实现会更容易下。参考15.11节。