15.19 从C语言中读取类文件对象¶

问题¶

你要写C扩展来读取来自任何Python类文件对象中的数据（比如普通文件、StringIO对象等）。

解决方案¶

要读取一个类文件对象的数据，你需要重复调用 `read()` 方法，然后正确的解码获得的数据。

下面是一个C扩展函数例子，仅仅只是读取一个类文件对象中的所有数据并将其输出到标准输出：

```
#define CHUNK_SIZE 8192

/* Consume a "file-like" object and write bytes to stdout */
static PyObject *py_consume_file(PyObject *self, PyObject *args) {
  PyObject *obj;
  PyObject *read_meth;
  PyObject *result = NULL;
  PyObject *read_args;

  if (!PyArg_ParseTuple(args,"O", &obj)) {
    return NULL;
  }

  /* Get the read method of the passed object */
  if ((read_meth = PyObject_GetAttrString(obj, "read")) == NULL) {
    return NULL;
  }

  /* Build the argument list to read() */
  read_args = Py_BuildValue("(i)", CHUNK_SIZE);
  while (1) {
    PyObject *data;
    PyObject *enc_data;
    char *buf;
    Py_ssize_t len;

    /* Call read() */
    if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
      goto final;
    }

    /* Check for EOF */
    if (PySequence_Length(data) == 0) {
      Py_DECREF(data);
      break;
    }

    /* Encode Unicode as Bytes for C */
    if ((enc_data=PyUnicode_AsEncodedString(data,"utf-8","strict"))==NULL) {
      Py_DECREF(data);
      goto final;
    }

    /* Extract underlying buffer data */
    PyBytes_AsStringAndSize(enc_data, &buf, &len);

    /* Write to stdout (replace with something more useful) */
    write(1, buf, len);

    /* Cleanup */
    Py_DECREF(enc_data);
    Py_DECREF(data);
  }
  result = Py_BuildValue("");
```

```
 final:
  /* Cleanup */
  Py_DECREF(read_meth);
  Py_DECREF(read_args);
  return result;
}
```

要测试这个代码，先构造一个类文件对象比如一个StringIO实例，然后传递进来：

```
>>> import io
>>> f = io.StringIO('Hello\nWorld\n')
>>> import sample
>>> sample.consume_file(f)
Hello
World
>>>
```

讨论¶

和普通系统文件不同的是，一个类文件对象并不需要使用低级文件描述符来构建。 因此，你不能使用普通的C库函数来访问它。 你需要使用Python的C API来像普通文件类似的那样操作类文件对象。

在我们的解决方案中，`read()` 方法从被传递的对象中提取出来。 一个参数列表被构建然后不断的被传给 `PyObject_Call()` 来调用这个方法。 要检查文件末尾（EOF），使用了 `PySequence_Length()` 来查看是否返回对象长度为0.

对于所有的I/O操作，你需要关注底层的编码格式，还有字节和Unicode之前的区别。 本节演示了如何以文本模式读取一个文件并将结果文本解码为一个字节编码，这样在C中就可以使用它了。 如果你想以二进制模式读取文件，只需要修改一点点即可，例如：

```
...
/* Call read() */
if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
  goto final;
}

/* Check for EOF */
if (PySequence_Length(data) == 0) {
  Py_DECREF(data);
  break;
}
if (!PyBytes_Check(data)) {
  Py_DECREF(data);
  PyErr_SetString(PyExc_IOError, "File must be in binary mode");
  goto final;
}

/* Extract underlying buffer data */
PyBytes_AsStringAndSize(data, &buf, &len);
...
```

本节最难的地方在于如何进行正确的内存管理。 当处理 `PyObject *` 变量的时候，需要注意管理引用计数以及在不需要的变量的时候清理它们的值。 对 `Py_DECREF()` 的调用就是来做这个的。

本节代码以一种通用方式编写，因此他也能适用于其他的文件操作，比如写文件。 例如，要写数据，只需要获取类文件对象的 `write()` 方法，将数据转换为合适的Python对象 （字节或Unicode），然后调用该方法将输入写入到文件。

最后，尽管类文件对象通常还提供其他方法（比如readline(), read_info()）， 我们最好只使用基本的 `read()` 和 `write()` 方法。 在写C扩展的时候，能简单就尽量简单。