6.12 读取嵌套和可变长二进制数据¶

问题¶

你需要读取包含嵌套或者可变长记录集合的复杂二进制格式的数据。这些数据可能包含图片、视频、电子地图文件等。

解决方案¶

`struct` 模块可被用来编码/解码几乎所有类型的二进制的数据结构。为了解释清楚这种数据，假设你用下面的Python数据结构 来表示一个组成一系列多边形的点的集合：

```
polys = [
    [ (1.0, 2.5), (3.5, 4.0), (2.5, 1.5) ],
    [ (7.0, 1.2), (5.1, 3.0), (0.5, 7.5), (0.8, 9.0) ],
    [ (3.4, 6.3), (1.2, 0.5), (4.6, 9.2) ],
]
```

现在假设这个数据被编码到一个以下列头部开始的二进制文件中去了：

```
+------+--------+-----------------------------------+
|Byte  | Type   | Description                       |
+======+========+===================================+
|0     | int    | 文件代码（0x1234，小端）          |
+------+--------+-----------------------------------+
|4     | double | x 的最小值（小端）                |
+------+--------+-----------------------------------+
|12    | double | y 的最小值（小端）                |
+------+--------+-----------------------------------+
|20    | double | x 的最大值（小端）                |
+------+--------+-----------------------------------+
|28    | double | y 的最大值（小端）                |
+------+--------+-----------------------------------+
|36    | int    | 三角形数量（小端）                |
+------+--------+-----------------------------------+
```

紧跟着头部是一系列的多边形记录，编码格式如下：

```
+------+--------+-------------------------------------------+
|Byte  | Type   | Description                               |
+======+========+===========================================+
|0     | int    | 记录长度（N字节）                         |
+------+--------+-------------------------------------------+
|4-N   | Points | (X,Y) 坐标，以浮点数表示                  |
+------+--------+-------------------------------------------+
```

为了写这样的文件，你可以使用如下的Python代码：

```
import struct
import itertools

def write_polys(filename, polys):
    # Determine bounding box
    flattened = list(itertools.chain(*polys))
    min_x = min(x for x, y in flattened)
    max_x = max(x for x, y in flattened)
```

```
    min_y = min(y for x, y in flattened)
    max_y = max(y for x, y in flattened)
    with open(filename, 'wb') as f:
        f.write(struct.pack('<iddddi', 0x1234,
                            min_x, min_y,
                            max_x, max_y,
                            len(polys)))
        for poly in polys:
            size = len(poly) * struct.calcsize('<dd')
            f.write(struct.pack('<i', size + 4))
            for pt in poly:
                f.write(struct.pack('<dd', *pt))
```

将数据读取回来的时候，可以利用函数 `struct.unpack()` ，代码很相似，基本就是上面写操作的逆序。如下：

```
def read_polys(filename):
    with open(filename, 'rb') as f:
        # Read the header
        header = f.read(40)
        file_code, min_x, min_y, max_x, max_y, num_polys = \
            struct.unpack('<iddddi', header)
        polys = []
        for n in range(num_polys):
            pbytes, = struct.unpack('<i', f.read(4))
            poly = []
            for m in range(pbytes // 16):
                pt = struct.unpack('<dd', f.read(16))
                poly.append(pt)
            polys.append(poly)
    return polys
```

尽管这个代码可以工作，但是里面混杂了很多读取、解包数据结构和其他细节的代码。如果用这样的代码来处理真实的数据文件， 那未免也太繁杂了点。因此很显然应该有另一种解决方法可以简化这些步骤，让程序员只关注自最重要的事情。

在本小节接下来的部分，我会逐步演示一个更加优秀的解析字节数据的方案。 目标是可以给程序员提供一个高级的文件格式化方法，并简化读取和解包数据的细节。但是我要先提醒你， 本小节接下来的部分代码应该是整本书中最复杂最高级的例子，使用了大量的面向对象编程和元编程技术。 一定要仔细的阅读我们的讨论部分，另外也要参考下其他章节内容。

首先，当读取字节数据的时候，通常在文件开始部分会包含文件头和其他的数据结构。 尽管struct模块可以解包这些数据到一个元组中去，另外一种表示这种信息的方式就是使用一个类。 就像下面这样：

```
import struct

class StructField:
    '''
    Descriptor representing a simple structure field
    '''
    def __init__(self, format, offset):
        self.format = format
        self.offset = offset
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
```

```
        else:
            r = struct.unpack_from(self.format, instance._buffer, self.offset)
            return r[0] if len(r) == 1 else r

class Structure:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)
```

这里我们使用了一个描述器来表示每个结构字段，每个描述器包含一个结构兼容格式的代码以及一个字节偏移量， 存储在内部的内存缓冲中。在 `__get__()` 方法中， `struct.unpack_from()` 函数被用来从缓冲中解包一个值，省去了额外的分片或复制操作步骤。

`Structure` 类就是一个基础类，接受字节数据并存储在内部的内存缓冲中，并被 `StructField` 描述器使用。 这里使用了 `memoryview()` ，我们会在后面详细讲解它是用来干嘛的。

使用这个代码，你现在就能定义一个高层次的结构对象来表示上面表格信息所期望的文件格式。例如：

```
class PolyHeader(Structure):
    file_code = StructField('<i', 0)
    min_x = StructField('<d', 4)
    min_y = StructField('<d', 12)
    max_x = StructField('<d', 20)
    max_y = StructField('<d', 28)
    num_polys = StructField('<i', 36)
```

下面的例子利用这个类来读取之前我们写入的多边形数据的头部数据：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader(f.read(40))
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

这个很有趣，不过这种方式还是有一些烦人的地方。首先，尽管你获得了一个类接口的便利， 但是这个代码还是有点臃肿，还需要使用者指定很多底层的细节(比如重复使用 `StructField` ，指定偏移量等)。 另外，返回的结果类同样确实一些便利的方法来计算结构的总数。

任何时候只要你遇到了像这样冗余的类定义，你应该考虑下使用类装饰器或元类。 元类有一个特性就是它能够被用来填充许多低层的实现细节，从而释放使用者的负担。 下面我来举个例子，使用元类稍微改造下我们的 `Structure` 类：

```
class StructureMeta(type):
    '''
    Metaclass that automatically creates StructField descriptors
    '''
    def __init__(self, clsname, bases, clsdict):
```

```
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if format.startswith(('<','>','!','@')):
                byte_order = format[0]
                format = format[1:]
            format = byte_order + format
            setattr(self, fieldname, StructField(format, offset))
            offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

class Structure(metaclass=StructureMeta):
    def __init__(self, bytedata):
        self._buffer = bytedata

    @classmethod
    def from_file(cls, f):
        return cls(f.read(cls.struct_size))
```

使用新的 `Structure` 类，你可以像下面这样定义一个结构：

```
class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('i', 'num_polys')
    ]
```

正如你所见，这样写就简单多了。我们添加的类方法 `from_file()` 让我们在不需要知道任何数据的大小和结构的情况下就能轻松的从文件中读取数据。比如：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

一旦你开始使用了元类，你就可以让它变得更加智能。例如，假设你还想支持嵌套的字节结构， 下面是对前面元类的一个小的改进，提供了一个新的辅助描述器来达到想要的效果：

```
class NestedStruct:
    '''
    Descriptor representing a nested structure
```

```
    '''
    def __init__(self, name, struct_type, offset):
        self.name = name
        self.struct_type = struct_type
        self.offset = offset

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            data = instance._buffer[self.offset:
                        self.offset+self.struct_type.struct_size]
            result = self.struct_type(data)
            # Save resulting structure back on instance to avoid
            # further recomputation of this step
            setattr(instance, self.name, result)
            return result

class StructureMeta(type):
    '''
    Metaclass that automatically creates StructField descriptors
    '''
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if isinstance(format, StructureMeta):
                setattr(self, fieldname,
                        NestedStruct(fieldname, format, offset))
                offset += format.struct_size
            else:
                if format.startswith(('<','>','!','@')):
                    byte_order = format[0]
                    format = format[1:]
                format = byte_order + format
                setattr(self, fieldname, StructField(format, offset))
                offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)
```

在这段代码中，`NestedStruct` 描述器被用来叠加另外一个定义在某个内存区域上的结构。 它通过将原始内存缓冲进行切片操作后实例化给定的结构类型。由于底层的内存缓冲区是通过一个内存视图初始化的， 所以这种切片操作不会引发任何的额外的内存复制。相反，它仅仅就是之前的内存的一个叠加而已。 另外，为了防止重复实例化，通过使用和8.10小节同样的技术，描述器保存了该实例中的内部结构对象。

使用这个新的修正版，你就可以像下面这样编写：

```
class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'), # nested struct
```

```
    (Point, 'max'), # nested struct
    ('i', 'num_polys')
  ]
```

令人惊讶的是，它也能按照预期的正常工作，我们实际操作下：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min # Nested structure
<__main__.Point object at 0x1006a48d0>
>>> phead.min.x
0.5
>>> phead.min.y
0.5
>>> phead.max.x
7.0
>>> phead.max.y
9.2
>>> phead.num_polys
3
>>>
```

到目前为止，一个处理定长记录的框架已经写好了。但是如果组件记录是变长的呢？ 比如，多边形文件包含变长的部分。

一种方案是写一个类来表示字节数据，同时写一个工具函数来通过多少方式解析内容。跟6.11小节的代码很类似：

```
class SizedRecord:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

    @classmethod
    def from_file(cls, f, size_fmt, includes_size=True):
        sz_nbytes = struct.calcsize(size_fmt)
        sz_bytes = f.read(sz_nbytes)
        sz, = struct.unpack(size_fmt, sz_bytes)
        buf = f.read(sz - includes_size * sz_nbytes)
        return cls(buf)

    def iter_as(self, code):
        if isinstance(code, str):
            s = struct.Struct(code)
            for off in range(0, len(self._buffer), s.size):
                yield s.unpack_from(self._buffer, off)
        elif isinstance(code, StructureMeta):
            size = code.struct_size
            for off in range(0, len(self._buffer), size):
                data = self._buffer[off:off+size]
                yield code(data)
```

类方法 `SizedRecord.from_file()` 是一个工具，用来从一个文件中读取带大小前缀的数据块， 这也是很多文件格式常用的方式。作为输入，它接受一个包含大小编码的结构格式编码，并且也是自己形式。 可选的 `includes_size` 参数指定了字节数是否包含头部大小。 下面是一个例子教你怎样使用从多边形文件中读取单独的多边形数据：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.num_polys
3
>>> polydata = [ SizedRecord.from_file(f, '<i')
...           for n in range(phead.num_polys) ]
>>> polydata
[<__main__.SizedRecord object at 0x1006a4d50>,
<__main__.SizedRecord object at 0x1006a4f50>,
<__main__.SizedRecord object at 0x10070da90>]
>>>
```

可以看出， `SizedRecord` 实例的内容还没有被解析出来。 可以使用 `iter_as()` 方法来达到目的，这个方法接受一个结构格式化编码或者是 `Structure` 类作为输入。 这样子可以很灵活的去解析数据，例如：

```
>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as('<dd'):
...         print(p)
...
Polygon 0
(1.0, 2.5)
(3.5, 4.0)
(2.5, 1.5)
Polygon 1
(7.0, 1.2)
(5.1, 3.0)
(0.5, 7.5)
(0.8, 9.0)
Polygon 2
(3.4, 6.3)
(1.2, 0.5)
(4.6, 9.2)
>>>

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as(Point):
...         print(p.x, p.y)
...
Polygon 0
1.0 2.5
3.5 4.0
2.5 1.5
Polygon 1
7.0 1.2
5.1 3.0
0.5 7.5
0.8 9.0
Polygon 2
3.4 6.3
1.2 0.5
4.6 9.2
>>>
```

将所有这些结合起来，下面是一个 `read_polys()` 函数的另外一个修正版：

```
class Point(Structure):
```

```
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),
        (Point, 'max'),
        ('i', 'num_polys')
    ]

def read_polys(filename):
    polys = []
    with open(filename, 'rb') as f:
        phead = PolyHeader.from_file(f)
        for n in range(phead.num_polys):
            rec = SizedRecord.from_file(f, '<i')
            poly = [ (p.x, p.y) for p in rec.iter_as(Point) ]
            polys.append(poly)
    return polys
```

讨论¶

这一节向你展示了许多高级的编程技术，包括描述器，延迟计算，元类，类变量和内存视图。 然而，它们都为了同一个特定的目标服务。

上面的实现的一个主要特征是它是基于懒解包的思想。当一个 `Structure` 实例被创建时， `__init__()` 仅仅只是创建一个字节数据的内存视图，没有做其他任何事。 特别的，这时候并没有任何的解包或者其他与结构相关的操作发生。 这样做的一个动机是你可能仅仅只对一个字节记录的某一小部分感兴趣。我们只需要解包你需要访问的部分，而不是整个文件。

为了实现懒解包和打包，需要使用 `StructField` 描述器类。 用户在 `_fields_` 中列出来的每个属性都会被转化成一个 `StructField` 描述器， 它将相关结构格式码和偏移值保存到存储缓存中。元类 `StructureMeta` 在多个结构类被定义时自动创建了这些描述器。 我们使用元类的一个主要原因是它使得用户非常方便的通过一个高层描述就能指定结构格式，而无需考虑低层的细节问题。

`StructureMeta` 的一个很微妙的地方就是它会固定字节数据顺序。 也就是说，如果任意的属性指定了一个字节顺序(<表示低位优先 或者 >表示高位优先)， 那后面所有字段的顺序都以这个顺序为准。这么做可以帮助避免额外输入，但是在定义的中间我们仍然可能切换顺序的。 比如，你可能有一些比较复杂的结构，就像下面这样：

```
class ShapeFile(Structure):
    _fields_ = [ ('>i', 'file_code'), # Big endian
        ('20s', 'unused'),
        ('i', 'file_length'),
        ('<i', 'version'), # Little endian
        ('i', 'shape_type'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('d', 'min_z'),
        ('d', 'max_z'),
        ('d', 'min_m'),
```

```
    ('d', 'max_m')]
```

之前我们提到过，`memoryview()`的使用可以帮助我们避免内存的复制。 当结构存在嵌套的时候，`memoryviews`可以叠加同一内存区域上定义的机构的不同部分。 这个特性比较微妙，但是它关注的是内存视图与普通字节数组的切片操作行为。 如果你在一个字节字符串或字节数组上执行切片操作，你通常会得到一个数据的拷贝。 而内存视图切片不是这样的，它仅仅是在已存在的内存上面叠加而已。因此，这种方式更加高效。

还有很多相关的章节可以帮助我们扩展这里讨论的方案。 参考8.13小节使用描述器构建一个类型系统。 8.10小节有更多关于延迟计算属性值的讨论，并且跟NestedStruct描述器的实现也有关。 9.19小节有一个使用元类来初始化类成员的例子，和`StructureMeta`类非常相似。 Python的`ctypes`源码同样也很有趣，它提供了对定义数据结构、数据结构嵌套这些相似功能的支持。