4.4 实现迭代器协议¶

问题¶

你想构建一个能支持迭代操作的自定义对象，并希望找到一个能实现迭代协议的简单方法。

解决方案¶

目前为止，在一个对象上实现迭代最简单的方式是使用一个生成器函数。 在4.2小节中，使用Node类来表示树形数据结构。你可能想实现一个以深度优先方式遍历树形节点的生成器。 下面是代码示例：

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        yield self
        for c in self:
            yield from c.depth_first()

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
    # Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)
```

在这段代码中，`depth_first()` 方法简单直观。 它首先返回自己本身并迭代每一个子节点并 通过调用子节点的 `depth_first()` 方法(使用 `yield from` 语句)返回对应元素。

讨论¶

Python的迭代协议要求一个 `__iter__()` 方法返回一个特殊的迭代器对象， 这个迭代器对象实现了 `__next__()` 方法并通过 `StopIteration` 异常标识迭代的完成。 但是，实现这些通常会比较繁琐。 下面我们演示下这种方式，如何使用一个关联迭代器类重新实现 `depth_first()` 方法：

```
class Node2:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)


class DepthFirstIterator(object):
    '''
    Depth-first traversal
    '''

    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
        self._child_iter = None

    def __iter__(self):
        return self

    def __next__(self):
        # Return myself if just started; create an iterator for children
        if self._children_iter is None:
            self._children_iter = iter(self._node)
            return self._node
        # If processing a child, return its next item
        elif self._child_iter:
            try:
                nextchild = next(self._child_iter)
                return nextchild
            except StopIteration:
                self._child_iter = None
                return next(self)
        # Advance to the next child and start its iteration
        else:
            self._child_iter = next(self._children_iter).depth_first()
            return next(self)
```

`DepthFirstIterator` 类和上面使用生成器的版本工作原理类似， 但是它写起来很繁琐，因为迭代器必须在迭代处理过程中维护大量的状态信息。 坦白来讲，没人愿意写这么晦涩的代码。将你的迭代器定义为一个生成器后一切迎刃而解。