8.22 不用递归实现访问者模式¶

问题¶

你使用访问者模式遍历一个很深的嵌套树形数据结构，并且因为超过嵌套层级限制而失败。 你想消除递归，并同时保持访问者编程模式。

解决方案¶

通过巧妙的使用生成器可以在树遍历或搜索算法中消除递归。 在8.21小节中，我们给出了一个访问者类。 下面我们利用一个栈和生成器重新实现这个类：

```
import types

class Node:
    pass

class NodeVisitor:
    def visit(self, node):
        stack = [node]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()

        return last_result

    def _visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

如果你使用这个类，也能达到相同的效果。事实上你完全可以将它作为上一节中的访问者模式的替代实现。 考虑如下代码，遍历一个表达式的树：

```
class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
```

```
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

# A sample visitor class that evaluates expressions
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -self.visit(node.operand)

if __name__ == '__main__':
    # 1 + 2*(3-4) / 5
    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)
    # Evaluate it
    e = Evaluator()
    print(e.visit(t4))  # Outputs 0.6
```

如果嵌套层次太深那么上述的Evaluator就会失效：

```
>>> a = Number(0)
>>> for n in range(1, 100000):
...     a = Add(a, Number(n))
```

```
...
>>> e = Evaluator()
>>> e.visit(a)
Traceback (most recent call last):
...
  File "visitor.py", line 29, in _visit
return meth(node)
  File "visitor.py", line 67, in visit_Add
return self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>
```

现在我们稍微修改下上面的Evaluator：

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        yield (yield node.left) + (yield node.right)

    def visit_Sub(self, node):
        yield (yield node.left) - (yield node.right)

    def visit_Mul(self, node):
        yield (yield node.left) * (yield node.right)

    def visit_Div(self, node):
        yield (yield node.left) / (yield node.right)

    def visit_Negate(self, node):
        yield - (yield node.operand)
```

再次运行，就不会报错了：

```
>>> a = Number(0)
>>> for n in range(1,100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>
```

如果你还想添加其他自定义逻辑也没问题：

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        print('Add:', node)
        lhs = yield node.left
        print('left=', lhs)
        rhs = yield node.right
        print('right=', rhs)
        yield lhs + rhs
    ...
```

下面是简单的测试：

```
>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>
left= 1
right= -0.4
0.6
>>>
```

讨论¶

这一小节我们演示了生成器和协程在程序控制流方面的强大功能。 避免递归的一个通常方法是使用一个栈或队列的数据结构。 例如，深度优先的遍历算法，第一次碰到一个节点时将其压入栈中，处理完后弹出栈。 `visit()` 方法的核心思路就是这样。

另外一个需要理解的就是生成器中yield语句。当碰到yield语句时，生成器会返回一个数据并暂时挂起。 上面的例子使用这个技术来代替了递归。例如，之前我们是这样写递归：

```
value = self.visit(node.left)
```

现在换成yield语句：

```
value = yield node.left
```

它会将 `node.left` 返回给 `visit()` 方法，然后 `visit()` 方法调用那个节点相应的 `visit_Name()` 方法。 yield暂时将程序控制器让出给调用者，当执行完后，结果会赋值给value，

看完这一小节，你也许想去寻找其它没有yield语句的方案。但是这么做没有必要，你必须处理很多棘手的问题。 例如，为了消除递归，你必须要维护一个栈结构，如果不使用生成器，代码会变得很臃肿，到处都是栈操作语句、回调函数等。 实际上，使用yield语句可以让你写出非常漂亮的代码，它消除了递归但是看上去又很像递归实现，代码很简洁。