8.21 实现访问者模式¶

问题¶

你要处理由大量不同类型的对象组成的复杂数据结构，每一个对象都需要进行不同的处理。 比如，遍历一个树形结构，然后根据每个节点的相应状态执行不同的操作。

解决方案¶

这里遇到的问题在编程领域中是很普遍的，有时候会构建一个由大量不同对象组成的数据结构。 假设你要写一个表示数学表达式的程序，那么你可能需要定义如下的类：

```
class Node:
    pass

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

class BinaryOperator(Node):
    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
```

然后利用这些类构建嵌套数据结构，如下所示：

```
# Representation of 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)
```

这样做的问题是对于每个表达式，每次都要重新定义一遍，有没有一种更通用的方式让它支持所有的数字和操作符呢。 这里我们使用访问者模式可以达到这样的目的：

```
class NodeVisitor:
    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__))
```

为了使用这个类，可以定义一个类继承它并且实现各种 `visit_Name()` 方法，其中Name是node类型。 例如，如果你想求表达式的值，可以这样写：

```
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 -node.operand
```

使用示例：

```
>>> e = Evaluator()
>>> e.visit(t4)
0.6
>>>
```

作为一个不同的例子，下面定义一个类在一个栈上面将一个表达式转换成多个操作序列：

```
class StackCode(NodeVisitor):
    def generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return self.instructions

    def visit_Number(self, node):
        self.instructions.append(('PUSH', node.value))

    def binop(self, node, instruction):
        self.visit(node.left)
        self.visit(node.right)
        self.instructions.append((instruction,))

    def visit_Add(self, node):
        self.binop(node, 'ADD')

    def visit_Sub(self, node):
        self.binop(node, 'SUB')

    def visit_Mul(self, node):
        self.binop(node, 'MUL')

    def visit_Div(self, node):
        self.binop(node, 'DIV')

    def unaryop(self, node, instruction):
        self.visit(node.operand)
        self.instructions.append((instruction,))

    def visit_Negate(self, node):
        self.unaryop(node, 'NEG')
```

使用示例：

```
>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
```

```
>>>
```

讨论¶

刚开始的时候你可能会写大量的if/else语句来实现， 这里访问者模式的好处就是通过 `getattr()` 来获取相应的方法，并利用递归来遍历所有的节点：

```
def binop(self, node, instruction):
    self.visit(node.left)
    self.visit(node.right)
    self.instructions.append((instruction,))
```

还有一点需要指出的是，这种技术也是实现其他语言中switch或case语句的方式。 比如，如果你正在写一个HTTP框架，你可能会写这样一个请求分发的控制器：

```
class HTTPHandler:
    def handle(self, request):
        methname = 'do_' + request.request_method
        getattr(self, methname)(request)
    def do_GET(self, request):
        pass
    def do_POST(self, request):
        pass
    def do_HEAD(self, request):
        pass
```

访问者模式一个缺点就是它严重依赖递归，如果数据结构嵌套层次太深可能会有问题， 有时候会超过Python的递归深度限制(参考 `sys.getrecursionlimit()`)。

可以参照8.22小节，利用生成器或迭代器来实现非递归遍历算法。

在跟解析和编译相关的编程中使用访问者模式是非常常见的。 Python本身的 `ast` 模块值得关注下，可以去看看源码。9.24小节演示了一个利用 `ast` 模块来处理Python源代码的例子。