9.24 解析与分析Python源码¶

问题¶

你想写解析并分析Python源代码的程序。

解决方案¶

大部分程序员知道Python能够计算或执行字符串形式的源代码。例如：

```
>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
9
>>>
```

尽管如此，`ast` 模块能被用来将Python源码编译成一个可被分析的抽象语法树（AST）。例如：

```
>>> import ast
>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
right=Name(id='x', ctx=Load())))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None))], orelse=[])])"
>>>
```

分析源码树需要你自己更多的学习，它是由一系列AST节点组成的。 分析这些节点最简单的方法就是定义一个访问者类，实现很多 `visit_NodeName()` 方法， `NodeName()` 匹配那些你感兴趣的节点。下面是这样一个类，记录了哪些名字被加载、存储和删除的信息。

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
        self.deleted = set()

    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
```

```
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)

# Sample usage
if __name__ == '__main__':
    # Some Python code
    code = '''
    for i in range(10):
        print(i)
    del i
    '''

    # Parse into an AST
    top = ast.parse(code, mode='exec')

    # Feed the AST to analyze name usage
    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

如果你运行这个程序，你会得到下面这样的输出：

```
Loaded: {'i', 'range', 'print'}
Stored: {'i'}
Deleted: {'i'}
```

最后，AST可以通过 `compile()` 函数来编译并执行。例如：

```
>>> exec(compile(top,'<stdin>', 'exec'))
0
1
2
3
4
5
6
7
8
9
>>>
```

讨论¶

当你能够分析源代码并从中获取信息的时候，你就能写很多代码分析、优化或验证工具了。 例如，相比盲目的传递一些代码片段到类似 `exec()` 函数中，你可以先将它转换成一个AST， 然后观察它的细节看它到底是怎样做的。 你还可以写一些工具来查看某个模块的全部源码，并且在此基础上执行某些静态分析。

需要注意的是，如果你知道自己在干啥，你还能够重写AST来表示新的代码。 下面是一个装饰器例子，可以通过重新解析函数体源码、 重写AST并重新创建函数代码对象来将全局访问变量降为函数体作用范围，

```
# namelower.py
import ast
import inspect

# Node visitor that lowers globally accessed names into
# the function body as local variables.
class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Compile some assignments to lower the constants
        code = '__globals = globals()\n'
        code += '\n'.join("{0} = __globals['{0}']".format(name)
                          for name in self.lowered_names)
        code_ast = ast.parse(code, mode='exec')
```

```
        # Inject new statements into the function body
        node.body[:0] = code_ast.body

        # Save the function object
        self.func = node

# Decorator that turns global names into locals
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Skip source lines prior to the @lower_names decorator
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Hack to deal with indented code
        if src.startswith((' ','\t')):
            src = 'if 1:\n' + src
        top = ast.parse(src, mode='exec')

        # Transform the AST
        cl = NameLower(namelist)
        cl.visit(top)

        # Execute the modified AST
        temp = {}
        exec(compile(top,'','exec'), temp, temp)

        # Pull out the modified code object
        func.__code__ = temp[func.__name__].__code__
        return func
    return lower
```

为了使用这个代码，你可以像下面这样写：

```
INCR = 1
@lower_names('INCR')
def countdown(n):
    while n > 0:
        n -= INCR
```

装饰器会将 `countdown()` 函数重写为类似下面这样子：

```
def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
    while n > 0:
        n -= INCR
```

在性能测试中，它会让函数运行快20%

现在，你是不是想为你所有的函数都加上这个装饰器呢？或许不会。 但是，这却是对于一些高级技术比如AST操作、源码操作等等的一个很好的演示说明

本节受另外一个在 `ActiveState` 中处理Python字节码的章节的启示。 使用AST是一个更加高级点的技术，并且也更简单些。参考下面一节获得字节码的更多信息。