9.25 拆解Python字节码¶

问题¶

你想通过将你的代码反编译成低级的字节码来查看它底层的工作机制。

解决方案¶

`dis` 模块可以被用来输出任何Python函数的反编译结果。例如：

```
>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...     print('Blastoff!')
...
>>> import dis
>>> dis.dis(countdown)
  2           0 SETUP_LOOP              30 (to 32)
        >>    2 LOAD_FAST                0 (n)
              4 LOAD_CONST               1 (0)
              6 COMPARE_OP               4 (>)
              8 POP_JUMP_IF_FALSE       30

  3          10 LOAD_GLOBAL              0 (print)
             12 LOAD_CONST               2 ('T-minus')
             14 LOAD_FAST                0 (n)
             16 CALL_FUNCTION            2
             18 POP_TOP

  4          20 LOAD_FAST                0 (n)
             22 LOAD_CONST               3 (1)
             24 INPLACE_SUBTRACT
             26 STORE_FAST               0 (n)
             28 JUMP_ABSOLUTE            2
        >>   30 POP_BLOCK

  5     >>   32 LOAD_GLOBAL              0 (print)
             34 LOAD_CONST               4 ('Blastoff!')
             36 CALL_FUNCTION            1
             38 POP_TOP
             40 LOAD_CONST               0 (None)
             42 RETURN_VALUE
>>>
```

讨论¶

当你想要知道你的程序底层的运行机制的时候，`dis` 模块是很有用的。比如如果你想试着理解性能特征。 被 `dis()` 函数解析的原始字节码如下所示：

```
>>> countdown.__code__.co_code
b"x'\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00\x00\x83
\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00q\x03\x00Wt\x00\x00d\x04\x00\x83
\x01\x00\x01d\x00\x00S"
>>>
```

如果你想自己解释这段代码，你需要使用一些在 `opcode` 模块中定义的常量。例如：

```
>>> c = countdown.__code__.co_code
>>> import opcode
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[2]]
'LOAD_FAST'
>>>
```

奇怪的是，在 `dis` 模块中并没有函数让你以编程方式很容易的来处理字节码。 不过，下面的生成器函数可以将原始字节码序列转换成 `opcodes` 和参数。

```
import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while i < n:
        op = codebytes[i]
        i += 1
        if op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
            extended_arg = 0
            i += 2
            if op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue
        else:
            oparg = None
        yield (op, oparg)
```

使用方法如下：

```
>>> for op, oparg in generate_opcodes(countdown.__code__.co_code):
...     print(op, opcode.opname[op], oparg)
```

这种方式很少有人知道，你可以利用它替换任何你想要替换的函数的原始字节码。 下面我们用一个示例来演示整个过程：

```
>>> def add(x, y):
...     return x + y
...
>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Make a completely new code object with bogus byte code
>>> import types
>>> newbytecode = b'xxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
...     c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_consts,
...     c.co_names, c.co_varnames, c.co_filename, c.co_name,
...     c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault
```

你可以像这样耍大招让解释器奔溃。但是，对于编写更高级优化和元编程工具的程序员来讲， 他们可能真的需要重写字节码。本节最后的部分演示了这个是怎样做到的。你还可以参考另外一个类似的例子： this code on ActiveState