

# 《源码探秘 CPython》50. 剖析字节码指令，观测Python程序的执行过程

原创 古明地觉 古明地觉的编程教室 2022-03-16 08:30



微信扫一扫  
关注该公众号

收录于合集

#CPython

97个 >



生活虽然充满不易，但我们也不能够放弃。



这里我们通过问与答的方式回顾一下前面的内容，并介绍一些常见的字节码指令。

## 1) 请问 Python 程序是怎么运行的？是编译成机器码后再执行的吗？

执行 Python 程序的虽然被称为 Python 解释器，但它其实包含一个编译器和一个虚拟机。

当我们在命令行敲下 `python xxxx.py` 时，Python 解释器中的编译器首先登场，将源代码编译成 **PyCodeObject 对象**。PyCodeObject 对象包含字节码、以及执行字节码时所需的名字以及常量。

当编译器完成编译动作后，接力棒便传给虚拟机，虚拟机负责维护执行上下文，逐条执行字节码指令。执行上下文中最核心的名字空间，便是由虚拟机负责维护的。

因此 Python 程序的执行原理其实更像 Java，可以用两个词来概括：虚拟机和字节码。不同的是，Java 的编译器 `javac` 与虚拟机 `java` 是分离的，而 Python 将两者整合成一个 `python` 命令。

## 2) pyc 文件保存什么东西，有什么作用？

Python 源代码在执行时需要先由编译器编译成 PyCodeObject 对象，然后再交由虚拟机来执行。而不管程序执行多少次，只要源码没有变化，编译后得到的 PyCodeObject 对象就肯定是一样的。

因此，Python 将 PyCodeObject 对象序列化并保存到 `pyc` 文件中。当程序再次执行时，Python 可以直接从 `pyc` 文件中加载，省去编译环节。当然了，如果源码文件发生改动，那么 `pyc` 文件便会失效，这时必须重新编译 `py` 文件。

## 3) 如何查看 Python 程序的字节码？

Python 标准库中的 `dis` 模块，可以对 PyCodeObject 对象以及函数进行反编译，并显示其中的字节码。


我们可以直接传入 PyCodeObject 对象，当然也可以传入一个函数，会自动获取其内部

的\_\_code\_\_。

#### 4) 介绍几个常见的字节码指令？


Python的字节码指令有120多个，但其中有几个太常见了，我们这里必须要提一下，然后再举例说明。

|              |   |
|--------------|---|
| LOAD_CONST   | 加载一个常量                                      |
| LOAD_FAST    | 在局部作用域(比如函数)中加载一个局部变量                       |
| LOAD_GLOBAL  | 在局部作用域(比如函数)中加载一个全局变量或者内置变量                 |
| LOAD_NAME    | 在全局作用域中加载一个全局变量或者内置变量                       |
| STORE_FAST   | 在局部作用域中定义一个局部变量，来建立和某个对象之间的映射关系             |
| STORE_GLOBAL | 在局部作用域中定义一个global关键字声明的全局变量，来建立和某个对象之间的映射关系 |
| STORE_NAME   | 在全局作用域中定义一个全局变量，来建立和某个对象之间的映射关系             |

 古明地觉的 Python小屋

下面来实际操作一波，看看这些指令：

```
1  import dis
2
3  name = "古明地觉"
4
5  def foo():
6      gender = "female"
7      print(gender)
8      print(name)
9
10 dis.dis(foo)
11 """
12      6          0 LOAD_CONST          1 ('female')
13      |          2 STORE_FAST         0 (gender)
14
15      7          4 LOAD_GLOBAL        0 (print)
16      |          6 LOAD_FAST         0 (gender)
17      |          8 CALL_FUNCTION      1
18      |         10 POP_TOP
19
20      8          12 LOAD_GLOBAL        0 (print)
21      |          14 LOAD_GLOBAL        1 (name)
22      |          16 CALL_FUNCTION      1
23      |          18 POP_TOP
24      |          20 LOAD_CONST          0 (None)
25      |          22 RETURN_VALUE
26  """
```

 古明地觉的 Python小屋

- **0 LOAD\_CONST**：加载字符串常量 "female"。更准确的说，加载的应该是指向字符串常量的指针，但为了描述方便，我们就用常量代替了；
- **2 STORE\_FAST**：在局部作用域中定义一个局部变量gender，和字符串对象 "female" 建立映射关系，本质上就是让变量 gender 保存这个字符串对象的地址；
- **4 LOAD\_GLOBAL**：在局部作用域中加载一个内置变量 print；
- **6 LOAD\_FAST**：在局部作用域中加载一个局部变量 gender；
- **14 LOAD\_GLOBAL**：在局部作用域中加载一个全局变量 name；

```
1  import dis
2
3  name = "古明地觉"
4
```

```

5 def foo():
6     global name
7     name = "古明地恋"
8
9 dis.dis(foo)
10 """
11      7          0 LOAD_CONST          1 ('古明地恋')
12          2 STORE_GLOBAL          0 (name)
13          4 LOAD_CONST          0 (None)
14          6 RETURN_VALUE
15 """

```

- **0 LOAD\_CONST**: 加载字符串常量 "古明地恋";
- **2 STORE\_GLOBAL**: 在局部作用域中定义一个被global关键字声明的全局变量;

```

1 import dis
2
3 s = """
4 name = "古明地觉"
5 print(name)
6 """
7 # 以模块的方式进行编译
8 dis.dis(compile(s, "<file>", "exec"))
9 """
10      2          0 LOAD_CONST          0 ('古明地觉')
11          2 STORE_NAME          0 (name)
12
13      3          4 LOAD_NAME          1 (print)
14          6 LOAD_NAME          0 (name)
15          8 CALL_FUNCTION          1
16         10 POP_TOP
17         12 LOAD_CONST          1 (None)
18         14 RETURN_VALUE
19 """

```

- **0 LOAD\_CONST**: 加载字符串常量 "古明地觉";
- **2 STORE\_NAME**: 在全局作用域中定义一个全局变量 name, 并和上面的字符串对象进行绑定;
- **4 LOAD\_NAME**: 在全局作用域中加载一个内置变量 print;
- **6 LOAD\_NAME**: 在全局作用域中加载一个全局变量 name;

以上我们就通过代码实际演示了这些指令的作用, 它们和常量、变量的加载, 以及变量的定义密切相关, 可以说常见的不能再常见了。你写的任何代码在反编译之后都少不了它们的身影, 至少会出现一个, 因此有必要提前解释一下。

不管加载的是常量、还是变量, 得到的永远是指向对象的指针。

**5) Python 在变量交换时有两种不同的写法, 示例如下。这两种写法有什么区别吗? 哪种写法更好?**

```

1 # 写法一
2 a, b = b, a
3
4 # 写法二
5 tmp = a
6 a = b
7 b = tmp

```

这两种写法都能实现变量交换, 并且从表面上看第一种写法更加简洁明了。但是优雅的外表下是否隐藏着不为人知的性能缺陷呢? 想要找到答案, 唯一的途径就是研究字节码:

```

# 写法一
1          0 LOAD_NAME          0 (b)

```

|       |    |            |         |
|-------|----|------------|---------|
| 1     | 0  | LOAD_NAME  | 0 (b)   |
| 2     | 2  | LOAD_NAME  | 1 (a)   |
| 4     | 4  | ROT_TWO    |         |
| 6     | 6  | STORE_NAME | 1 (a)   |
| 8     | 8  | STORE_NAME | 0 (b)   |
| # 写法二 |    |            |         |
| 1     | 0  | LOAD_NAME  | 0 (a)   |
|       | 2  | STORE_NAME | 1 (tmp) |
| 2     | 4  | LOAD_NAME  | 2 (b)   |
|       | 6  | STORE_NAME | 0 (a)   |
| 3     | 8  | LOAD_NAME  | 1 (tmp) |
|       | 10 | STORE_NAME | 2 (b)   |

古明地觉的Python小屋

其实不用看字节码也知道**写法一**的性能更优，但这里我们主要是想研究一下**写法一**背后的原理。从字节码可以看出，先通过LOAD\_NAME将变量b和a加载进来，压入运行时栈；调用ROT\_TWO指令；再将栈的元素依次弹出，赋值给a和b。

等号右边是 b, a，所以先加载 b 后加载 a；等号左边是 a, b，所以先给 a 赋值、再给 b 赋值。因此无论是加载、还是赋值，都是从左往右进行的。

所以核心就在于 ROT\_TWO 这个指令，我们来看看它都做了什么。

```
1 case TARGET(ROT_TWO): {
2     //获取栈顶元素, 由于b先入栈、a后入栈
3     //再加上栈是先入后出, 所以这里获取的栈顶元素就是a
4     PyObject *top = TOP();
5     //运行时栈的第二个元素就是b
6     //TOP是查看栈顶元素、SECOND是查看栈的第二个元素
7     //并且这两个函数只是获取, 不会将元素从栈中弹出
8     PyObject *second = SECOND();
9     //将栈顶元素设置为 second, 这里显然就是变量 b
10    //将栈的第二个元素设置为 top, 这里显然就是变量 a
11    SET_TOP(second);
12    SET_SECOND(top);
13    FAST_DISPATCH();
14 }
```

因此执行完 ROT\_TWO 指令之后，栈顶元素就是 b，栈的第二个元素就是 a。然后，后面的两个 STORE\_NAME 会将栈里面的元素 b、a 依次弹出，赋值给 a、b，从而完成变量交换。

因此 ROT\_TWO 指令只是将栈顶的两个元素交换位置，执行起来比 LOAD\_NAME 和 STORE\_NAME 都要快。至此我们可以得出结论了，第一种变量交换写法更优，原因如下：

- 代码简洁明了，不拖泥带水；
- 不需要辅助变量 tmp，节约内存；
- ROT\_TWO 指令比 LOAD\_NAME 和 STORE\_NAME 组成的指令对更有优势，执行效率更高；

## 6) 聊一聊 a, b, c = c, b, a 这种赋值方式的背后原理。

老规矩，查看字节码，因为在字节码的面前没有秘密。

```
1 1 0 LOAD_NAME 0 (c)
2 2 LOAD_NAME 1 (b)
3 4 LOAD_NAME 2 (a)
4 6 ROT_THREE
5 8 ROT_TWO
6 10 STORE_NAME 2 (a)
7 12 STORE_NAME 1 (b)
8 14 STORE_NAME 0 (c)
9 16 LOAD CONST 0 (None)
```



原理和 `a, b = b, a` 是类似的，首先 `LOAD_NAME` 将变量 `c`、`b`、`a` 依次压入栈中。由于栈先入后出的特性，此时栈的三个元素按照顺序（从栈顶到栈底）分别是 `a`、`b`、`c`。

然后是 `ROT_THREE` 和 `ROT_TWO`，毫无疑问，这两个指令执行完之后，会将栈的三个元素调换顺序，也就是将 `a`、`b`、`c` 变成 `c`、`b`、`a`。

最后 `STORE_NAME` 将栈的三个元素 `c`、`b`、`a` 依次弹出，分别赋值给 `a`、`b`、`c`，从而完成变量的交换。

因此核心就在 `ROT_THREE` 和 `ROT_TWO` 上面，由于后者我们已经见过了，所以我们看一下 `ROT_THREE`。


```
1 case TARGET(ROT_THREE): {
2     PyObject *top = TOP();
3     PyObject *second = SECOND();
4     PyObject *third = THIRD();
5     SET_TOP(second);
6     SET_SECOND(third);
7     SET_THIRD(top);
8     FAST_DISPATCH();
9 }
```

栈顶元素是 `top`、栈的第二个元素是 `second`、栈的第三个元素是 `third`，然后将栈顶元素设置为 `second`、栈的第二个元素设置为 `third`、栈的第三个元素设置为 `top`。

所以栈里面的元素 `a`、`b`、`c` 在经过 `ROT_THREE` 之后就变成了 `b`、`c`、`a`，显然这还不是正确的结果。于是继续执行 `ROT_TWO`，将栈的前两个元素进行交换，执行完之后就变成了 `c`、`b`、`a`。

#### 7) 聊一聊 `a, b, c, d = d, c, b, a` 背后的原理，和上面 6) 提到的变量交换有区别吗？

```
1      2      0 LOAD_NAME      0 (d)
2      2      2 LOAD_NAME      1 (c)
3      4      4 LOAD_NAME      2 (b)
4      6      6 LOAD_NAME      3 (a)
5      8      8 BUILD_TUPLE      4
6     10     10 UNPACK_SEQUENCE      4
7     12     12 STORE_NAME      3 (a)
8     14     14 STORE_NAME      2 (b)
9     16     16 STORE_NAME      1 (c)
10    18     18 STORE_NAME      0 (d)
11    20     20 LOAD_CONST      0 (None)
12    22     22 RETURN_VALUE
13
```

 古明地觉的 Python 小屋

仍是从左往右，将变量 `d`、`c`、`b`、`a` 依次压入栈中，但此时没有直接将栈里面的元素做交换，而是构建一个元组。因为往栈里面压入了四个元素，所以 `BUILD_TUPLE` 后面的 `oparg` 是 4，表示构建长度为 4 的元组。

```
1 case TARGET(BUILD_TUPLE): {
2     PyObject *tup = PyTuple_New(oparg);
3     if (tup == NULL)
4         goto error;
5     while (--oparg >= 0) {
6         PyObject *item = POP();
7         PyTuple_SET_ITEM(tup, oparg, item);
8     }
9     PUSH(tup);
10    DISPATCH();
11 }
```

先申请一个长度为 4 的 PyTupleObject，然后元素依次出栈，被设置到元组中。注意代码中的 while 条件，显然元素是从后往前设置的，而元素的出栈顺序是 a b c d，因此循环结束之后，元组为 (d, c, b, a)。构建完，再将其压入栈中，此时运行时栈里面只有这一个元组（指针）。

接下来是 UNPACK\_SEQUENCE，我们看看它做了什么？

```
case TARGET(UNPACK_SEQUENCE): {
    PREDICTED(UNPACK_SEQUENCE);
    PyObject *seq = POP(), *item, **items;
    if (PyTuple_CheckExact(seq) &&
        PyTuple_GET_SIZE(seq) == oparg) {
        items = ((PyTupleObject *)seq)->ob_item;
        while (oparg-- > 0) {
            item = items[oparg];
            Py_INCREF(item);
            PUSH(item);
        }
    } else if (PyList_CheckExact(seq) &&
               PyList_GET_SIZE(seq) == oparg) {
        items = ((PyListObject *)seq)->ob_item;
        while (oparg-- > 0) {
            item = items[oparg];
            Py_INCREF(item);
        }
    }
}
```

从名字上可以看出，这个指令用于对序列进行解包，由于元组、列表、以及其它可迭代对象都属于序列，所以要进行类型判断。但这里我们只看元组部分，其它部分是类似的。

首先代码中的 seq 就是从栈里面弹出的元组（指针），items 是元组内部的指针数组，存放了变量 d c b a。注意了，然后还是从后往前遍历，将指向的对象的引用计数加1之后，再将变量压入栈中。循环结束之后，栈里面的元素按照顺序也是 d c b a。

最后 STORE\_NAME 将 d c b a 依次弹出，赋值给变量 a b c d，从而完成变量交换。所以，当交换的变量多了之后，不会直接在运行时栈里面操作，而是将栈里面的元素挨个弹出、构建元组；然后再按照指定顺序，将元组里面的元素重新压到栈里面。

不管是哪一种做法，Python在进行变量交换时（多元赋值同理）所做的事情是不变的，核心分为三步走。首先固定按照从左往右的顺序，将等号右边的变量或常量依次压入栈中；然后，将运行时栈里面的元素进行翻转；最后，再将运行时栈里面的元素挨个弹出，固定按照从左往右的顺序依次赋值给等号左边的变量。

只不过当变量数量不多时，翻转元素会直接基于栈进行操作；而当达到四个时，则需要额外借助于元组。

## 8) a, b, c, d = d, c, b, a 和 a, b, c, d = [d, c, b, a] 有区别吗？

答案是没有区别，两者在反编译之后对应的字节码指令只有一处不同。

|    |   |    |                 |   |        |
|----|---|----|-----------------|---|--------|
| 1  | 2 | 0  | LOAD_NAME       | 0 | (d)    |
| 2  |   | 2  | LOAD_NAME       | 1 | (c)    |
| 3  |   | 4  | LOAD_NAME       | 2 | (b)    |
| 4  |   | 6  | LOAD_NAME       | 3 | (a)    |
| 5  |   | 8  | BUILD_LIST      | 4 |        |
| 6  |   | 10 | UNPACK_SEQUENCE | 4 |        |
| 7  |   | 12 | STORE_NAME      | 3 | (a)    |
| 8  |   | 14 | STORE_NAME      | 2 | (b)    |
| 9  |   | 16 | STORE_NAME      | 1 | (c)    |
| 10 |   | 18 | STORE_NAME      | 0 | (d)    |
| 11 |   | 20 | LOAD_CONST      | 0 | (None) |
| 12 |   | 22 | RETURN_VALUE    |   |        |
| 13 |   |    |                 |   |        |

前者是 BUILD\_TUPLE，现在变成了 BUILD\_LIST，其它部分一模一样，所以两者的效果是相同的。当然啦，由于元组的构建比列表快一些，因此还是推荐第一种写法。

### 9) a = b = c = 123 背后的原理是什么？

如果变量 a、b、c 指向的值相同，比如都是 123，那么便可以通过该种方式进行赋值。那么它背后是怎么做的呢？

|   |   |    |              |          |
|---|---|----|--------------|----------|
| 1 | 2 | 0  | LOAD_CONST   | 0 (123)  |
| 2 |   | 2  | DUP_TOP      |          |
| 3 |   | 4  | STORE_NAME   | 0 (a)    |
| 4 |   | 6  | DUP_TOP      |          |
| 5 |   | 8  | STORE_NAME   | 1 (b)    |
| 6 |   | 10 | STORE_NAME   | 2 (c)    |
| 7 |   | 12 | LOAD_CONST   | 1 (None) |
| 8 |   | 14 | RETURN_VALUE |          |
| 9 |   |    |              |          |

出现了一个新的字节码指令 DUP\_TOP，只要搞清楚它的作用，事情就简单了。

```
1 case TARGET(DUP_TOP): {
2     // 获取栈顶元素，注意是获取、不是弹出
3     // TOP: 查看元素, POP: 弹出元素
4     PyObject *top = TOP();
5     // 增加指向对象的引用计数
6     Py_INCREF(top);
7     // 压入栈中
8     PUSH(top);
9     FAST_DISPATCH();
10 }
```

所以 DUP\_TOP 干的事情就是将栈顶元素拷贝一份，再重新压到栈里面。另外，不管等号左边有多少个变量，模式都是一样的，我们以 a=b=c=d=e=123 为例：

|    |   |    |              |          |
|----|---|----|--------------|----------|
| 1  | 2 | 0  | LOAD_CONST   | 0 (123)  |
| 2  |   | 2  | DUP_TOP      |          |
| 3  |   | 4  | STORE_NAME   | 0 (a)    |
| 4  |   | 6  | DUP_TOP      |          |
| 5  |   | 8  | STORE_NAME   | 1 (b)    |
| 6  |   | 10 | DUP_TOP      |          |
| 7  |   | 12 | STORE_NAME   | 2 (c)    |
| 8  |   | 14 | DUP_TOP      |          |
| 9  |   | 16 | STORE_NAME   | 3 (d)    |
| 10 |   | 18 | STORE_NAME   | 4 (e)    |
| 11 |   | 20 | LOAD_CONST   | 1 (None) |
| 12 |   | 22 | RETURN_VALUE |          |
| 13 |   |    |              |          |

将常量（指针）压入运行时栈，然后拷贝一份，赋值给 a；再拷贝一份，赋值给 b；再拷贝一份，赋值给 c；再拷贝一份，赋值给 d；最后自身赋值给 e。

以上就是链式赋值的秘密，其实没有什么好神奇的，就是将栈顶元素进行拷贝，再依次赋值。

### 10) 请解释一下 a is b 和 a == b 的区别？

is 是对象标识符(object identity)，用于判断两个变量是不是引用同一个对象，等价于 id(a)==id(b)；而 == 操作符则是判断两个变量所引用的对象是否相等，等价于调用魔法方法 a.\_\_eq\_\_(b)。并且 == 操作符可以通过 \_\_eq\_\_ 魔法方法进行覆写(overriding)，而 is 操作符无法覆写。

Python 的变量在 C 看来只是一个指针，因此两个变量是否指向同一个对象，等价于 C 中的两个指

针是否相等；而Python的==，则需要调用PyObject\_RichCompare，来比较它们指向的对象所维护的值是否相等。

从字节码上看，这两个语句也很接近，区别仅在比较指令COMPARE\_OP的操作数上：

```
1  # a is b
2      1          0 LOAD_NAME          0 (a)
3          2 LOAD_NAME          1 (b)
4          4 COMPARE_OP          8 (is)
5          6 POP_TOP
6
7  # a == b
8      1          0 LOAD_NAME          0 (a)
9          2 LOAD_NAME          1 (b)
10         4 COMPARE_OP          2 (==)
11         6 POP_TOP
```

古明地觉的 Python小屋

我们看到操作数一个是 8、一个是 2，然后是 COMPARE\_OP 指令的背后逻辑：

```
1 case TARGET(COMPARE_OP): {
2     // 弹出栈顶元素, 这里是 b
3     PyObject *right = POP();
4     // 显然 left 就是 a
5     // b 被弹出之后, 它成为新的栈顶
6     PyObject *left = TOP();
7     // 进行比较, 比较结果为 res
8     PyObject *res = cmp_outcome(tstate, oparg, left, right);
9     // 减少 left 和 right 引用计数
10    Py_DECREF(left);
11    Py_DECREF(right);
12    // 将栈顶元素替换为 res
13    SET_TOP(res);
14    if (res == NULL)
15        goto error;
16    // 介绍 if 语句的时候说
17    PREDICT(POP_JUMP_IF_FALSE);
18    PREDICT(POP_JUMP_IF_TRUE);
19    // 相当于 continue
20    DISPATCH();
21 }
```

所以逻辑很简单，核心就在 cmp\_outcome 函数中。

```
1 static PyObject *
2 cmp_outcome(int op, PyObject *v, PyObject *w)
3 {
4     int res = 0;
5     // op 就是 COMPARE_OP 指令里面的操作数
6     switch (op) {
7         // PyCmp_IS 是一个枚举变量, 等于 8
8         // 定义在 Include/opcode.h 中
9         case PyCmp_IS:
10             // is 操作符, 在C的层面直接一个 == 判断即可
11             res = (v == w);
12             break;
13         // ...
14         default:
15             // 而PyObject_RichCompare是一个函数调用
16             // 比较对象维护的值是否相等
17             return PyObject_RichCompare(v, w, op);
18     }
19     v = res ? Py_True : Py_False;
20     Py_INCREF(v);
21     return v;
```



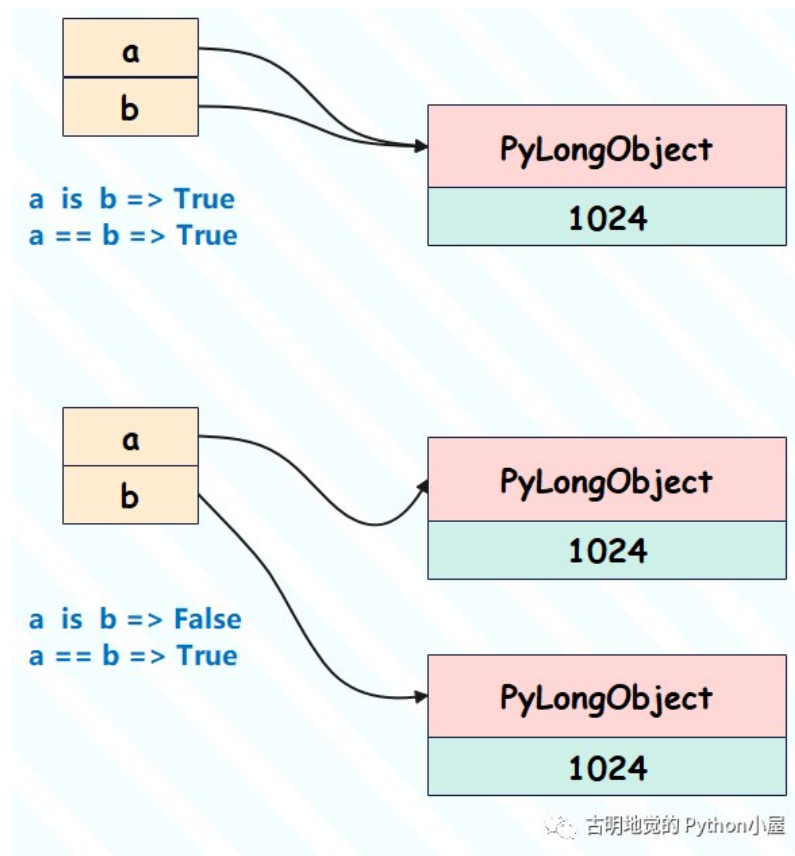
我们实际举个栗子：

```

1 >>> a = 1024
2 >>> b = a
3 >>> a is b
4 True
5 >>> a == b
6 True
7 # a 和 b 引用同一个对象
8 # is 和 == 操作均返回 True
9 >>>
10 >>>
11 >>> a = 1024
12 >>> b = int('1024')
13 >>> a is b
14 False
15 >>> a == b
16 True
17 # 显然引用的不是同一个对象, 因此 is 操作结果是 False
18 # 而对象的值相同, 因此 == 操作结果是 True

```

用一张图看一下它们之间的区别：



一般而言，如果 `a is b` 成立，那么 `a == b` 大部分成立。可能有人好奇，`a is b` 成立说明 `a` 和 `b` 指向的是同一个对象，那么 `a == b` 表示该对象和自己进行比较，结果应该始终是相等的呀，为啥是大部分成立呢？以下面两种情况为例：

```

1 class Girl:
2
3     def __eq__(self, other):
4         return False
5
6 g = Girl()
7 print(g is g) # True
8 print(g == g) # False

```

`__eq__` 返回 `False`，此时虽然是同一个对象，但是两者不相等。

```
1 import math
2 import numpy as np
3
4 a = float("nan")
5 b = math.nan
6 c = np.nan
7
8 print(a is a, a == a) # True False
9 print(b is b, b == b) # True False
10 print(c is c, c == c) # True False
```

`nan` 是一个特殊的浮点数，意思是 not a number（不是一个数字），用于表示空值。不管 `nan` 跟任何浮点数(包括自身)做何种数学比较，结果均为 `False`。但需要注意的是，在使用 `==` 进行比较的时候虽然是不相等的，但如果放到容器里面就不一定了。举个栗子：

```
1 import numpy as np
2
3 lst = [np.nan, np.nan, np.nan]
4 print(lst[0] == np.nan) # False
5 print(lst[1] == np.nan) # False
6 print(lst[2] == np.nan) # False
7 # lst 里面的三个元素和 np.nan 均不相等
8
9 # 但是 np.nan 位于列表中, 数量为 3
10 print(np.nan in lst) # True
11 print(lst.count(np.nan)) # 3
```

出现以上结果的原因就在于，元素被放到了容器里，而容器的一些 API 在比较元素时会先判定它们存储的对象的地址是否相同，即：是否指向了同一个对象。如果是，直接认为相等；否则，再去比较对象维护的值是否相等。

可以理解为先进行 `is` 判断，如果结果为 `True`，直接判定两者相等；如果 `is` 操作的结果不为 `True`，再去进行 `==` 判断。

因此 `np.nan in lst` 的结果为 `True`，`lst.count(np.nan)` 的结果是 3，因为它们会先比较对象的地址。地址相同，直接认为对象相等。

在用 `pandas` 做数据处理的时候，`nan` 是一个非常容易坑的地方。

## 11) 在与 `None` 比较时，为什么要用 `is` 而不是 `==`？

因为 `==` 可以被重载，所以不应该用 `==` 去判断。

```
1 class Girl:
2     def __eq__(self, other):
3         return True
4
5 g = Girl()
6 print(g is None) # False
7 print(g == None) # True
```

而 `None` 是一种特殊的内建对象，它是单例的，整个运行的程序中只有一个，所以应该用 `is` 去判断。

另外 `is` 在底层只需要一个 `==` 操作符即可完成；但是 Python 的 `==`，在底层则是需要调用 `PyObject_RichCompare` 函数。因此 `is None` 在性能上也更有优势，再加上 `None` 是单例对象，使用 `is` 判断再合适不过。

我们使用 `jupyter notebook` 测试一下两者的性能吧：

```
%timeit None is None
```

```
%timeit name is None
```

22.6 ns ± 1.5 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

```
%timeit name == None
```

31.1 ns ± 4.37 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

从结果上看，is None 要更快一些。

收录于合集 #CPython 97

< 上一篇

《源码探秘 CPython》51. 虚拟机的一般表达式

下一篇 >

《源码探秘 CPython》49. 虚拟机是怎么执行字节码的？

文章已于2022-03-19修改

喜欢此内容的人还喜欢

一文读懂 TypeScript 泛型及应用  
前端充电宝



常用的nginx配置信息模板、docker模板、k8s部署deploy模板、springboot基本application文件  
浪老猫的Super小窝



node.js代码混淆  
韩加华

