# 《源码探秘 CPython》51. 虚拟机的一般表达式

原创 古明地觉 古明地觉的编程教室 2022-03-17 10:21





## 复杂内建对象的创建

像整数对象、字符串对象在创建时的字节码，相信都已经理解了。总共两条指令：先LOAD常量，然后STORE，两者组成entry存放在local名字空间中。

但是问题来了，像列表、字典这样的对象，底层是怎么创建的呢？显然它们的创建要更复杂一些，两条指令是不够的。下面我们就来看看列表、字典在创建时对应的字节码是怎样的吧。

不过在此之前我们先看一些宏，这些是在遍历指令序列co_code时所需要的宏，里面包括了对栈的各种操作，以及对PyTupleObject对象内部元素的访问操作。

```
//获取PyTupleObject对象中指定索引对应的元素
#ifndef Py_DEBUG
#define GETITEM(v, i) PyTuple_GET_ITEM((PyTupleObject *)(v), (i))
#else
#define GETITEM(v, i) PyTuple_GetItem((v), (i))
#endif
//调整栈顶指针，这个stack_pointer指向运行时栈的顶端
#define BASIC_STACKADJ(n) (stack_pointer += n)
#define STACKADJ(n)     { (void)(BASIC_STACKADJ(n), \
                          lltrace && prtrace(TOP(), "stackadj")); \
                          assert(STACK_LEVEL() <= co->co_stacksize); }

//入栈操作
#define BASIC_PUSH(v)     (*stack_pointer++ = (v))
#define PUSH(v)                BASIC_PUSH(v)
//出栈操作
#define BASIC_POP()       (*--stack_pointer)
#define POP()           ((void)(lltrace && prtrace(TOP(), "pop")), \
                         BASIC_POP())
```



然后我们随便创建一个列表和字典吧。

```
import dis

s = """
lst = [1, 2, "3", "xxx"]
d = {"name": "古明地觉", "age": 16}
"""

dis.dis(compile(s, "<file>", "exec"))
"""
  2           0 LOAD_CONST               0 (1)
              2 LOAD_CONST               1 (2)
              4 LOAD_CONST               2 ('3')
              6 LOAD_CONST               3 ('xxx')
              8 BUILD_LIST               4
             10 STORE_NAME               0 (lst)

  3          12 LOAD_CONST               4 ('古明地觉')
             14 LOAD_CONST               5 (16)
             16 LOAD_CONST               6 (('name', 'age'))
             18 BUILD_CONST_KEY_MAP      2
             20 STORE_NAME               1 (d)
             22 LOAD_CONST               7 (None)
             24 RETURN_VALUE
"""
```

对于列表来说，它是先将常量加载进来了，从上面的 4 个 LOAD_CONST 也能看出来。然后重点来了，我们看到有一行指令 BUILD_LIST 4，显然这是要根据 LOAD 进来的 4 个常量创建一个列表，后面的 4 表示这个列表的容量是 4。

```
case TARGET(BUILD_LIST): {
    //可以看到oparg的含义取决于字节码指令
    //在LOAD_CONST中代表索引，这里则代表列表的容量
    //当然，此时列表的容量，和要存储的元素个数是相同的
    PyObject *list =  PyList_New(oparg);
    if (list == NULL)
        goto error;
    //从运行时栈里面将元素一个一个的弹出来
    while (--oparg >= 0) {
      //栈是先入后出结构
      //因此弹出元素的顺序是 "xxx"、"3"、2、1
        PyObject *item = POP();
      //所以这里的oparg是从后往前遍历的，即 3、2、1、0
      //最终会将"xxx"设置在索引为3的位置；
      //将 "3" 设置在索引为 2 的位置；
      //将 2 设置在索引为 1 的位置；
      //将 1 设置在索引为 0 的位置
      //这显然是符合我们的预期的
        PyList_SET_ITEM(list, oparg, item);
    }
    //构建完毕之后，将其压入运行时栈
    //此时栈中只有一个PyListObject对象
    //因为之前LOAD进来的4个常量在构建列表的时候已经被逐个弹出来了
    PUSH(list);
    // continue
    DISPATCH();
  }
```

但BUILD_LIST之后，只改变了运行时栈，没有改变local空间。所以接下来的STORE_NAME表示将在local空间中建立一个符号lst和BUILD_LIST构建的PyListObject对象之间的映射，也就是组合成一个entry放在local空间中，其中 key 指向符号lst、value 指向PyListObject对象，从Python的层面来看就是让变量lst保存列表对象的地址，这样我们后面才可以通过lst找到对应的列表。

然后我们再看看字典的构建方式，首先依旧是加载两个常量，显然这个是字典的value。但key是作为一个元组加载进来的，只需要LOAD_CONST一次，就加载了所有的 key。

BUILD_CONST_KEY_MAP 毋庸置疑就是构建一个字典了，后面的oparg是2，表示这个字典有两个entry，我们看一下源码：

```
case TARGET(BUILD_CONST_KEY_MAP): {
    // 循环变量
    Py_ssize_t i;
    //PyDictObject对象指针
    PyObject *map;
    //从栈顶获取所有的key， 一个元组
    PyObject *keys = TOP();
    //如果keys不是一个元组
    //或者这个元组的ob_size不等于oparg
    //那么表示字典构建失败
    if (!PyTuple_CheckExact(keys) ||
        PyTuple_GET_SIZE(keys) != (Py_ssize_t)oparg) {
        _PyErr_SetString(tstate, PyExc_SystemError,
                         "bad BUILD_CONST_KEY_MAP keys argument");
        goto error;
    }
    //申请一个字典，至少能够容纳oparg个键值对
    //但是具体的容量肯定是要大于oparg的
```

```
    map = _PyDict_NewPresized((Py_ssize_t)oparg);
    if (map == NULL) {
        goto error;
    }
    //很明显，这里开始循环了，要依次设置键值对了
    //还记得在BUILD_CONST_KEY_MAP之前，常量是怎么加载的吗?
    //是按照"古明地觉"、16、('name', 'age')的顺序加载的
    //所以栈里面的元素，从栈顶到栈底就应该是('name', 'age')、16、"古明地觉"
    for (i = oparg; i > 0; i--) {
        int err;
        //获取元组里面的元素，也就是key
        //注意：索引是oparg - i，而i是从oparg开始自减的
        //以当前为例，循环结束时，oparg - i分别是0、1
        //那么获取的元素显然就分别是："name"、"age"
        PyObject *key = PyTuple_GET_ITEM(keys, oparg - i);
        //然后这里的PEEK和TOP类似，都是获取元素但是不从栈里面删除
        //TOP是专门获取栈顶元素，PEEK还可以获取栈的其它位置的元素
        //而这里获取也是按照 "古明地觉"、16 的顺序获取
        //和"name"、"age"之间是正好对应的
        //由于从栈的第二个元素开始才是value，所以这里是i+1
        PyObject *value = PEEK(i + 1);
        //然后将entry设置在map里面
        err = PyDict_SetItem(map, key, value);
        if (err != 0) {
            Py_DECREF(map);
            goto error;
        }
    }
    //依次清空运行时栈，将栈里面的元素挨个弹出来
    Py_DECREF(POP());
    while (oparg--) {
        Py_DECREF(POP());
    }
    //将构建的PyDictObject对象压入运行时栈
    PUSH(map);
    DISPATCH();
}
```

最后STORE_NAME，将运行时栈中的字典弹出来，和符号d建立一个entry放到local空间中。而在所有的字节码指令都执行完毕之后，运行时栈会是空的，因为所有的信息都存储到了local名字空间中。

## 函数中的变量

我们上面定义的变量是在模块级别的作用域中，但如果我们在函数中定义呢？

```
def foo():
    i = 1
    s = "python"

"""
  2           0 LOAD_CONST               1 (1)
              2 STORE_FAST               0 (i)

  3           4 LOAD_CONST               2 ('python')
              6 STORE_FAST               1 (s)
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
"""
```

在将变量名和变量值绑定的时候，使用的不再是STORE_NAME，而是STORE_FAST，显然STORE_FAST会更快一些。

为什么这么说呢？这是因为函数中的局部变量总是固定不变的，在编译的时候就能确定使用的内存空间的位置，也能确定相应的字节码指令应该如何访问内存，因此就能以静态的方式来实现局部变量。

```
        case TARGET(STORE_FAST) {
            PyObject *value = POP();
            SETLOCAL(oparg, value);
            FAST_DISPATCH();
        }

// 局部变量的读写都在 fastlocals = f -> f_localsplus 上面
// 我们知道 f -> f_localsplus 这段连续空间被分成了四部分
// 分别给 局部变量、co_freevars、co_cellvars、运行时栈 所使用
// 因此在操作局部变量的时候，直接通过索引来操作数组 fastlocals 即可
#define SETLOCAL(i, value)      do { PyObject *tmp = GETLOCAL(i); \
                                     GETLOCAL(i) = value; \
                                     Py_XDECREF(tmp); } while (0)
                                                  古明地觉的 Python小屋
#define GETLOCAL(i)     (fastlocals[i])
```

既然有 STORE_FAST，就有 LOAD_FAST，它内部显然是调用了 GETLOCAL。

## 符号搜索

我们还是举个例子：

```
a = 5
b = a
"""
  1           0 LOAD_CONST               0 (5)
              2 STORE_NAME               0 (a)

  2           4 LOAD_NAME                0 (a)
              6 STORE_NAME               1 (b)
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
"""                                古明地觉的 Python小屋
```

首先源代码第一行对应的字节码指令无需介绍，但是第二行对应的指令变了，我们看到不再是LOAD_CONST，而是LOAD_NAME。

其实也很好理解，第一行`a = 5`，因为5是一个常量，所以是LOAD_CONST；但是`b = a`，这里的a是一个变量，所以是LOAD_NAME。

> 但不管是LOAD_CONST还是LOAD_NAME，都是将对象的指针加载进运行时栈。

```
//按照LGB规则，从名字空间中寻找变量名对应的值
//找不到就抛出NameError
case TARGET(LOAD_NAME) {
    //从符号表里面获取变量名
    PyObject *name = GETITEM(names, oparg);
    //获取local名字空间
    PyObject *locals = f->f_locals;
    PyObject *v; //value
    if (locals == NULL) {
        //名字空间不能为空
        PyErr_Format(PyExc_SystemError,
```

```
                    "no locals when loading %R", name);
        goto error;
    }
    //根据变量名从locals里面获取对应的value
    //如果locals是字典
    if (PyDict_CheckExact(locals)) {
        v = PyDict_GetItem(locals, name);
        Py_XINCREF(v);
    }
    else {
        //否则其类型对象要继承字典
        //此时调用PyObject_GetItem获取value
        v = PyObject_GetItem(locals, name);
        if (v == NULL) {
            if (!PyErr_ExceptionMatches(PyExc_KeyError))
                goto error;
            PyErr_Clear();
        }
    }
    //如果v是NULL，说明local名字空间里面没有
    if (v == NULL) {
        //于是从global名字空间里面找
        v = PyDict_GetItem(f->f_globals, name);
        Py_XINCREF(v);
        //如果v是NULL，说明global里面也没有
        if (v == NULL) {
            //local、global都没有，于是从builtin里面找
            if (PyDict_CheckExact(f->f_builtins)) {
                v = PyDict_GetItem(f->f_builtins, name);
                //还没有，NameError
                if (v == NULL) {
                    format_exc_check_arg(
                                PyExc_NameError,
                                NAME_ERROR_MSG, name);
                    goto error;
                }
                Py_INCREF(v);
            }
            else {
                //逻辑和上面类似
                v = PyObject_GetItem(f->f_builtins, name);
                if (v == NULL) {
                    if (PyErr_ExceptionMatches(PyExc_KeyError))
                        format_exc_check_arg(
                                    PyExc_NameError,
                                    NAME_ERROR_MSG, name);
                    goto error;
                }
            }
        }
    }
    //找到了，把v给push进去，也就是压栈
    PUSH(v);
    DISPATCH();
}
```

以上就是 LOAD_NAME，在全局空间中加载一个全局变量（或者内置变量）。但如果是在函数里面，那么b = a就既不是LOAD_CONST、也不是LOAD_NAME，而是LOAD_FAST，这是因为函数中的变量在编译的时候就已经确定。

问题来了，如果b = a在函数里面，而a = 5定义在函数外面呢？那么结果显然就是LOAD_GLOBAL，在函数内部加载一个全局变量。因此无论什么情况，都能知道这个a定义在什么地方。

## 数值运算

```
a = 5
b = a
c = a + b
"""
  1           0 LOAD_CONST               0 (5)
              2 STORE_NAME               0 (a)

  2           4 LOAD_NAME                0 (a)
              6 STORE_NAME               1 (b)

  3           8 LOAD_NAME                0 (a)
             10 LOAD_NAME                1 (b)
             12 BINARY_ADD
             14 STORE_NAME               2 (c)
             16 LOAD_CONST               1 (None)
             18 RETURN_VALUE
"""
```



我们直接从 8 LOAD_NAME开始看即可，首先是加载两个变量，然后通过BINARY_ADD进行加法运算。

```
case TARGET(BINARY_ADD) {
    //获取a和b对应的值
    //a是栈底、b是栈顶
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;

    //这里检测是否是字符串
    if (PyUnicode_CheckExact(left) &&
             PyUnicode_CheckExact(right)) {
        //是的话直接拼接
        sum = unicode_concatenate(left, right, f, next_instr);
    }
    else {
        //不是字符串的话，调用PyNumber_Add进行相加
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    //设置sum， 将栈顶的元素(之前的a)给替换掉
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```

过程非常简单，当然啦，还有很多其它和运算相关的指令，像加减乘除、取模、左移右移等等，可以自己看一下。

## 信息输出

最后来看看信息是如何输出的：

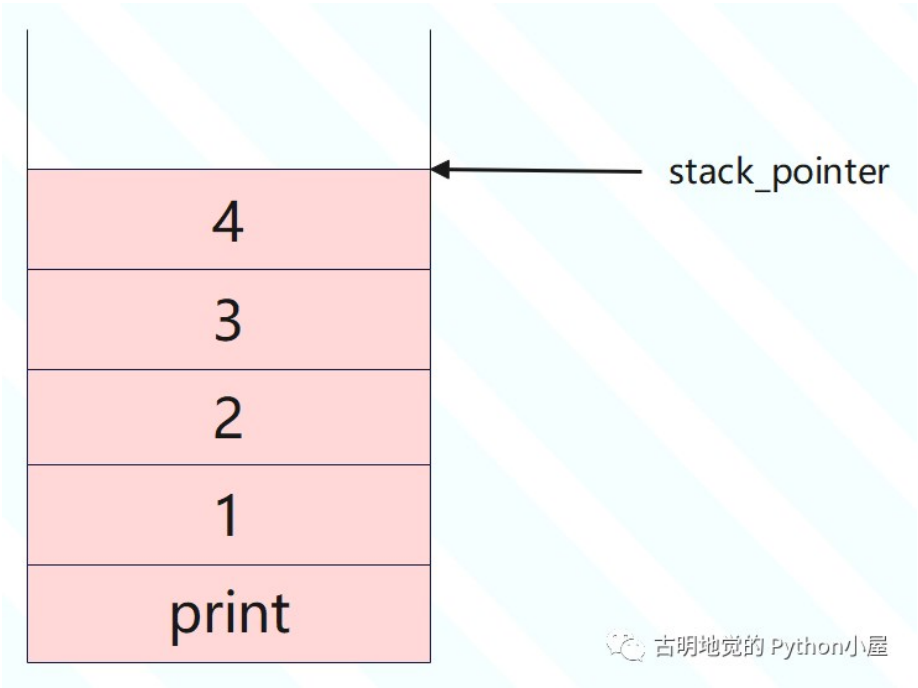
```
print(1, 2, 3, 4)
"""
  1           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 (1)
              4 LOAD_CONST               1 (2)
              6 LOAD_CONST               2 (3)
              8 LOAD_CONST               3 (4)
             10 CALL_FUNCTION            4
             12 POP_TOP
             14 LOAD_CONST               4 (None)
             16 RETURN_VALUE
"""
```



加载内置变量 print 和 4 个常量，压入运行时栈，然后 CALL_FUNCTION 指令显然是负责调用函数，后面的 4 表示调用函数时传了 4 个参数。

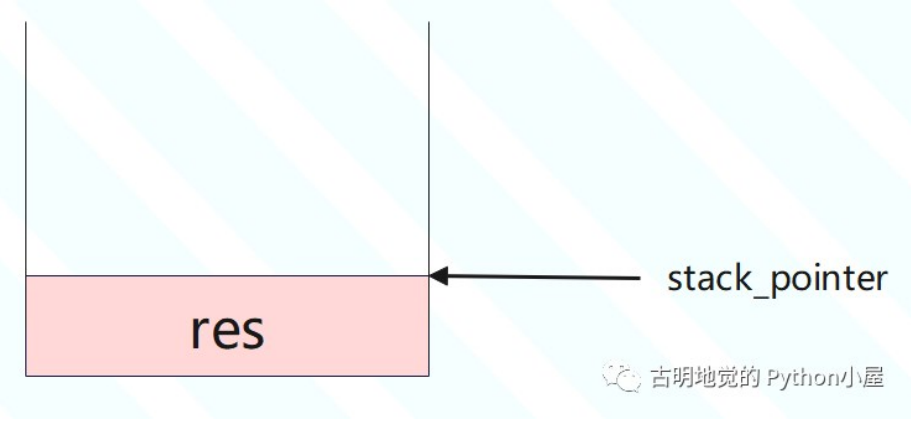
```
caseTARGET(CALL_FUNCTION) {
    PyObject **sp, *res;
    sp = stack_pointer;
    res = call_function(&sp, oparg, NULL);
    stack_pointer = sp;
    PUSH(res);
    if (res == NULL) {
        goto error;
    }
    DISPATCH();
}
```

stack_pointer 永远指向运行时栈的栈顶，而此时栈里面有 5 个元素：




然后是 call_function，它接收了 &sp、oparg，虽然还没有看里面的具体细节，但内部逻辑也能猜出来，会将栈里面的 print 以及相关参数依次弹出，进行调用。

而函数是有返回值的，所以调用完之后还要将返回值 res 设置为栈顶，对于当前的 print 来说就是 None。当然啦，由于 stack_pointer 始终指向栈顶，而函数调用结束后运行时栈内的元素数量发生了改变，所以还要调整 stack_pointer。




> 注意这个 stack_pointer 非常重要，它是一个二级指针，而虚拟机正是通过 stack_pointer 来操作的运行时栈。假设我们现在要向栈顶添加一个元素 v，该怎么做呢？
>
> 只需要 *stack_pointer++ = v 即可，因为从栈底到栈顶，地址是依次增大的。所以让 stack_pointer 移动到运行时栈（本质上是个数组）的下一个位置，再将元素设置成 v 即可。同理，如果想弹出栈顶元素，只需要 *--stack_pointer 即可；如果想查看栈顶元素，只需要 *stack_pointer 即可。
>
> PUSH、POP、TOP 这些对栈的操作（都是宏），都是基于 stack_pointer 实现的。

然后POP_TOP指令负责从栈的顶端弹出函数的返回值，因为我们不需要这个返回值，或者说我们没有使用变量接收，所以直接将其从栈顶弹出去即可。

但如果是res = print(c)，那么你会发现指令POP_TOP就变成了STORE_NAME，因为要将符号和返回值绑定起来放在local空间中。所以相比 STORE_NAME，POP_TOP 少了一步将符号和返回值绑定的过程。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》52. 流程控制语句 if 是怎么实现的?

下一篇 ›
《源码探秘 CPython》50. 剖析字节码指令，观测Python程序的执行过程

文章已于2022-04-24修改

喜欢此内容的人还喜欢

KubeVirt虚拟机初始化配置
DevOps苏楷


Linux下用nmcli命令做网卡绑定，你还不会用?
Cloud研习社

社区实践 | Kube-OVN为KubeVirt虚拟机分配固定IP
KubeOVN