



微信扫一扫
关注该公众号

收录于合集

#CPython

97个 >



函数最大的特点就是可以接收参数，如果只是单纯的封装，未免太无趣了。对于函数来说，参数会传什么，事先是不知道的，函数体内部只是利用参数做一些事情，比如调用参数的get方法。但是到底能不能调用get方法，就取决于你给参数传的值到底是什么了。

因此可以把参数看成是一个占位符，调用的时候，将某个值传进去赋给相应的参数，然后将逻辑走一遍即可。



调用函数时传递的参数，根据形式的不同可以分为四种类别：

- 位置参数 (positional argument) ;
- 关键字参数 (keyword argument) ;
- 扩展位置参数 (excess positional argument) ;
- 扩展关键字参数 (excess keyword argument) ;

参数分为形参和实参，在英文中形参叫做 parameter，实参叫做 argument。但在中文里区分的不是那么明显，我们一般统一称为参数。

下面来看一下call_function是如何处理函数信息的：

```
1 Py_LOCAL_INLINE(PyObject *) _Py_HOT_FUNCTION
2 call_function(PyThreadState *tstate, PyObject ***pp_stack, Py_ssize_t op
3 arg, PyObject *kwnames)
4 {
5     PyObject **pfunc = (*pp_stack) - oparg - 1;
6     PyObject *func = *pfunc;
7     PyObject *x, *w;
8     Py_ssize_t nkwards = (kwnames == NULL) ? 0 : PyTuple_GET_SIZE(kwnames);
9     s);
10    Py_ssize_t nargs = oparg - nkwards;
11    PyObject **stack = (*pp_stack) - nargs - nkwards;
12    //.....
13 }
```

CALL_FUNCTION指令的操作数 (oparg) 记录了函数的参数个数，包括位置参数和关键字参数。虽然扩展位置参数和扩展关键字参数是更高级的用法，但本质上也是由多个位置参数、多个关键字参数组成的。这就意味着，虽然函数中存在四种参数，但是只要记录位置参数和关键字参数的个数，就能知道一共有多少个参数，进而知道一共需要多大的内存来维护。

因此call_function里面的 nkwards 就是调用函数时传递的关键字参数的个数，nargs 就是传递的位置参数的个数，两者加起来等于操作数 oparg。

补充一下，在 Python3.8 之前，定义函数时，参数不能超过 255 个。

```
1 Traceback (most recent call last):
2   File "1.py", line 7, in <module>
3     exec(s)
4   File "<string>", line 2
5 SyntaxError: more than 255 arguments
```

上面是 Python3.6 的输出，但是从 3.8 开始，这个限制就被打破了，就算有十万个参数也不成问题。不过说实话，255 个参数已经足够用了，至少我没有见过有哪一个开源项目，里面会出现超过 255 个参数的函数。

函数内部的局部变量的个数，可以通过`co_nlocals`来获取。从名字也能看出来这个不是 `PyFunctionObject` 的属性，而是 `PyCodeObject` 的属性。

注意：局部变量包括了参数，因为函数参数也是局部变量，它们在内存中是连续放置的，都存储在符号表 `co_varnames` 中。当虚拟机为函数申请局部变量的内存空间时，就需要通过 `co_nlocals` 知道局部变量的总数。

可能会有人将 `co_nlocals` 和 `co_argcount` 搞混，前者表示局部变量的个数，后者表示可以通过位置参数或关键字参数传递的参数个数。

```
1 def foo(a, b, c, d=1):
2     pass
3
4 print(foo.__code__.co_argcount) # 4
5 print(foo.__code__.co_nlocals) # 4
6
7 def foo(a, b, c, d=1):
8     a = 1
9     b = 1
10
11 print(foo.__code__.co_argcount) # 4
12 print(foo.__code__.co_nlocals) # 4
13
14 def foo(a, b, c, d=1):
15     aa = 1
16
17 print(foo.__code__.co_argcount) # 4
18 print(foo.__code__.co_nlocals) # 5
```

`co_nlocals` 是参数的个数加上函数体中新创建的局部变量的个数，注意：函数参数也是局部变量，比如参数有一个 `a`，但是函数体里面新建了一个变量也叫 `a`，这是重新赋值，因此还是相当于一个参数。

但是 **`co_argcount`** 只记录参数的个数。因此一个很明显的结论：对于任意一个函数，`co_nlocals` 至少大于等于 `co_argcount`。

```
1 def foo(a, b, c, d=1, *args, **kwargs):
2     pass
3
4
5 print(foo.__code__.co_argcount) # 4
6 print(foo.__code__.co_nlocals) # 6
```

我们看到，对于扩展位置参数、扩展关键字参数来说，`co_argcount` 是不算在内的，因为你完全可以不传递，所以直接当成 0 来算。

但我们在函数体内部肯定能拿到 `args` 和 `kwargs`，因此 **`co_argcount`** 是 4，**`co_nlocals`** 是 6。

所有的扩展位置参数都存储在一个 `PyTupleObject` 对象中，所有的扩展关键字参数都存储在一个 `PyDictObject` 对象中。

`co_argcount` 和 `co_nlocals` 的值在编译的时候就已经确定。



位置参数的传递




下面来看看位置参数是如何传递的：

```
s = """
def f(name, age):
    age = age + 5
    print(name, age)

age = 11
f("satori", age)
"""

if __name__ == '__main__':
    import dis
    dis.dis(compile(s, "<file>", "exec"))
```

 古明地觉的 Python 小屋

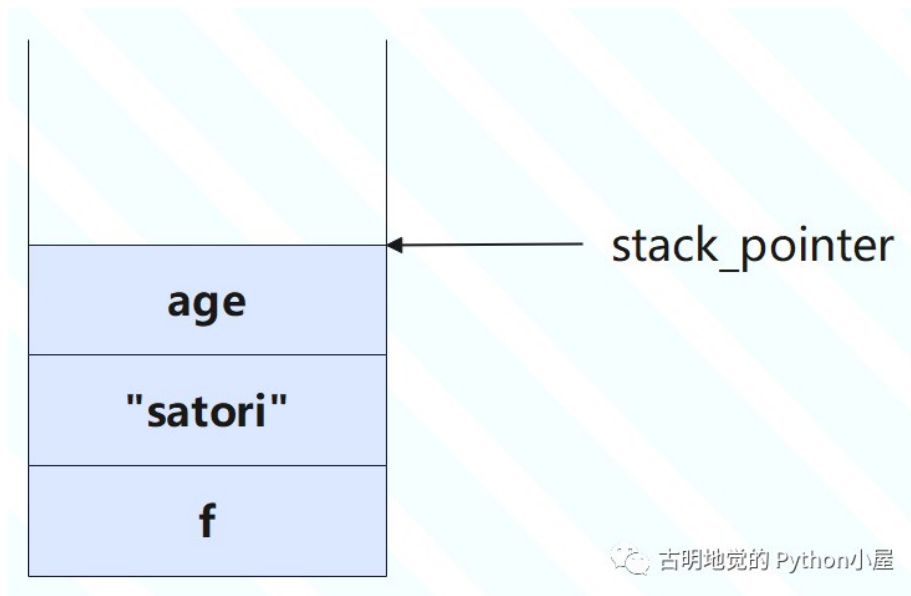
相信对于现在的我们来说，下面的字节码已经没有任何难度了。

```
1  0 LOAD_CONST          0 (<code object f at 0x0000.....>)
2  2 LOAD_CONST          1 ('f')
3  4 MAKE_FUNCTION        0
4  6 STORE_NAME           0 (f)
5
6  8 LOAD_CONST           2 (11)
7 10 STORE_NAME           1 (age)
8
9 12 LOAD_NAME            0 (f)
10 14 LOAD_CONST           3 ('satori')
11 16 LOAD_NAME            1 (age)
12 18 CALL_FUNCTION        2
13 20 POP_TOP
14 22 LOAD_CONST           4 (None)
15 24 RETURN_VALUE
16
17 Disassembly of <code object f at 0x0000.....>:
18  0 LOAD_FAST             1 (age)
19  2 LOAD_CONST            1 (5)
20  4 BINARY_ADD
21  6 STORE_FAST            1 (age)
22
23  8 LOAD_GLOBAL            0 (print)
24 10 LOAD_FAST             0 (name)
25 12 LOAD_FAST             1 (age)
26 14 CALL_FUNCTION        2
27 16 POP_TOP
28 18 LOAD_CONST            0 (None)
29 20 RETURN_VALUE
```

这里我们先看**f("satori", age)**的字节码：

1	12	LOAD_NAME	0 (f)
2	14	LOAD_CONST	3 ('satori')
3	16	LOAD_NAME	1 (age)
4	18	CALL_FUNCTION	2

这部分字节码之前说过，会将函数以及相关参数压入运行时栈：



然后执行 `_PyFunction_FastCallDict`，由于在调用时全部都是位置参数，所以会进入快分支 `function_code_fastcall`。

```

1 //Objects/call.c
2 static PyObject* _Py_HOT_FUNCTION
3 function_code_fastcall(PyCodeObject *co, PyObject *const *args, Py_ssize_t
4 _t nargs,
5                      PyObject *globals)
6 {
7     //栈帧对象
8     PyFrameObject *f;
9     //线程状态对象
10    PyThreadState *tstate = _PyThreadState_GET();
11    //f->localsplus
12    PyObject **fastlocals;
13    Py_ssize_t i;
14    PyObject *result;
15
16    assert(globals != NULL);
17    assert(tstate != NULL);
18    //创建与函数对应的PyFrameObject
19    //我们看到参数是co, 所以是根据PyCodeObject来创建的
20    //然后还有一个globals, 表示global名字空间
21    //因此最后没有PyFunctionObject什么事, 它只是起到一个输送的作用
22    f = _PyFrame_New_NoTrack(tstate, co, globals, NULL);
23    if (f == NULL) {
24        return NULL;
25    }
26    //获取 f_localsplus
27    fastlocals = f->f_localsplus;
28    //nargs 表示参数个数, args就是call_function里面 stack
29    //而 stack 此时指向运行时栈中的第一个参数
30    //所以这里的for循环就是将运行时栈中的参数拷贝到局部变量对应的内存中
31    //因为 f_localsplus 分别用于:局部变量、cell对象、free对象、运行时栈
32    for (i = 0; i < nargs; i++) {
33        Py_INCREF(*args);
34        fastlocals[i] = *args++;
35    }
36    //调用PyEval_EvalFrameEx, 进而调用_PyEval_EvalFrameDefault

```

```

37 //以新创建的栈帧为执行环境, 执行内部的字节码
38 //将函数的返回值赋值给 result
39 result = PyEval_EvalFrameEx(f,0);
40
41 //如果 f 的引用计数大于 1, 说明栈帧被保存起来了
42 //引用计数减一之后, 由于不会被销毁, 所以还要被 GC 跟踪
43 //之所以要被 GC 跟踪, 是因为栈帧是可变对象
44 if (Py_REFCNT(f) > 1) {
45     Py_DECREF(f);
46     //对 f 进行跟踪
47     _PyObject_GC_TRACK(f);
48 }
49 else {
50     ++tstate->recursion_depth;
51     Py_DECREF(f);
52     --tstate->recursion_depth;
53 }
54 //返回 result
55 return result;
}

```

从源码中可以看到，虚拟机首先通过 `_PyFrame_New_NoTrack` 创建了函数 `f` 对应的栈帧对象。

虚拟机对外暴露的是 `PyFrame_New`，但内部会调用 `_PyFrame_New_NoTrack`

随后将参数逐个拷贝到新建的栈帧对象的 `f_localsplus` 中，这个数组分成了四部分，但源码中的索引是从 0 开始的，所以运行时栈中的参数被拷贝到了 **局部变量对应的内存** 中。

注意：上面说的运行时栈指的是模块的栈帧中的运行时栈，因为加载参数的时候还没有涉及函数的调用。

```

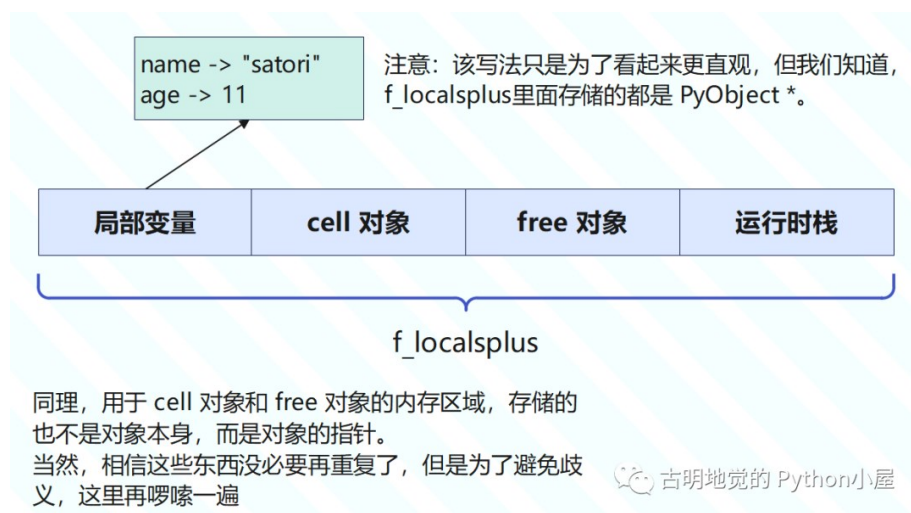
1 # 函数、以及参数都位于模块的栈帧中的运行时栈里面
2 # 加载完毕之后, 在模块的栈帧中调用函数
3 12 LOAD_NAME          0 (f)
4 14 LOAD_CONST         3 ('satori')
5 16 LOAD_NAME          1 (age)
6 18 CALL_FUNCTION      2

```

然后调用函数 `f`，为其创建新的栈帧，并将参数从模块的栈帧中的运行时栈拷贝到函数 `f` 的栈帧中的 `f_localsplus`（局部变量对应的内存）里面。

对于模块而言，会将执行权交给新的栈帧。而等到函数 `f` 执行完，也会回退到模块的栈帧中，并拿到函数 `f` 的返回值。然后会再将运行时栈清空，回到 `CALL_FUNCTION` 指令，通过 `PUSH(res)` 将返回值入栈。

对于函数 `f` 而言，在拷贝之后它的栈帧的 `f_localsplus` 布局如下：



在函数对应的 `PyCodeObject` 对象的 `co_nlocals` 域中，包含着函数参数的个数，因为函数参数也是局部符号的一种。所以 `f_localsplus` 里面一定有供函数参数使用的内存，并

且还是第一段内存。

处理完参数之后，还没有进入 `PyEval_EvalFrameEx`，所以此时运行时栈是空的，但是函数的参数已经位于 `f_localsplus`（第一段内存）里面了。

这里说的运行时栈是函数 `f` 的栈帧里的运行时栈，显然目前它是一个空栈；而之前说的用于拷贝元素的运行时栈，指的是模块的栈帧里的运行时栈。



位置参数的访问



当参数拷贝的动作完成之后，就会进入 `PyEval_EvalFrameEx`，然后进入 `_PyEval_EvalFrameDefault` 真正开始 `f` 的调用动作。会抽出栈帧里的 `f_code`，对指令逐条执行。

```
1  0  LOAD_FAST           1 (age)
2  2  LOAD_CONST          1 (5)
3  4  BINARY_ADD
4  6  STORE_FAST          1 (age)
```

对参数的读写，是通过以上几条指令集完成的，显然重点在 `LOAD_FAST` 和 `STORE_FAST` 上面。

```
1  //一个宏, 这里的 fastlocals 就是 f -> localsplus
2  #define GETLOCAL(i)      (fastlocals[i])
3
4  case TARGET(LOAD_FAST): {
5      //从fastlocals中获取索引为oparg的值
6      PyObject *value = GETLOCAL(oparg);
7      //...
8      PUSH(value);
9      FAST_DISPATCH();
10 }
11
12 case TARGET(STORE_FAST): {
13     PREDICTED(STORE_FAST);
14     //弹出元素
15     PyObject *value = POP();
16     //将索引为oparg的元素设置为value
17     SETLOCAL(oparg, value);
18     FAST_DISPATCH();
19 }
```

所以我们发现，`LOAD_FAST` 和 `STORE_FAST` 这一对指令是以 `f_localsplus` 为操作目标的，指令 `LOAD_FAST` 负责是将 `f_localsplus[1]` 压入到运行时栈中。

而在完成加法操作之后，又通过 `STORE_FAST` 将结果放入到 `f_localsplus[1]` 中，这样就实现了对 `age` 的更新。

那么以后打印 `age` 的时候，得到的结果就是16了。



小结

现在关于位置参数在函数调用时是如何传递的、在函数执行时又是如何被访问的，已经真相大白了。

在调用函数时，虚拟机将函数和参数依次压入调用者栈帧的运行时栈中，而在call_function中会执行_PyFunction_FastCallDict，进而执行function_code_fastcall。

在function_code_fastcall里面会为函数创建新的栈帧，也就是被调用者栈帧。然后将调用者栈帧的运行时栈中的参数依次拷贝到被调用者栈帧的f_localsplus中。

然后在访问函数参数时，虚拟机并没有按照通常访问符号的做法，去查什么名字空间，而是根据索引访问f_localsplus中和符号绑定的值（指针）。

而这种基于索引（偏移位置）来访问参数的方式也正是位置参数的由来，并且这种访问方式，其速度也是最快的。

收录于合集 #CPython 97

← 上一篇

《源码探秘 CPython》60. 函数是如何解析关键字参数的？

下一篇 →

《源码探秘 CPython》58. 函数在底层是如何调用的？

喜欢此内容的人还喜欢

C语言 数组作为函数的参数
小木编程

参数

编译器优化那些事儿（4）：归纳变量
毕昇编译

早期优化那些事
归纳变量

从零开始学 Python 之函数参数
豆豆的杂货铺