



微信扫一扫
关注该公众号

收录于合集

#CPython

97个 >



上一篇介绍了位置参数，下面来看一看关键字参数。另外函数还支持默认值，我们就放在一起介绍吧。



简单看一个函数：

```
s = """
def foo(a=1, b=2):
    print(a + b)

foo()
"""

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

古明地觉的 Python 小屋

字节码指令如下：

```
1 # 在构造函数之前，默认值已被压入运行时栈
2 0 LOAD_CONST      5 ((1, 2))
3 2 LOAD_CONST      2 (<code object foo at 0x...>)
4 4 LOAD_CONST      3 ('foo')
5 6 MAKE_FUNCTION    1 (defaults)
6 8 STORE_NAME       0 (foo)
7 .....
8 .....
```

相比无默认值的函数，有默认值的函数在加载 **PyCodeObject** 和 **函数名** 之前，会先将默认值以元组的形式给加载进来。

再来观察一下 **MAKE_FUNCTION** 这个指令，我们发现操作数是 1，而之前都是 0，那么这个 1 代表什么呢？根据提示，我们看到了一个 defaults，它和函数的 func_defaults 有什么关系吗？带着这些疑问，我们再来回顾一下这个指令：

```
1 case TARGET(MAKE_FUNCTION): {
2     // 对于当前例子来说，栈里面有三个元素
```

```

3 // 分别是:默认值、PyCodeObject、函数名
4
5 // 弹出栈顶元素, 得到函数名
6 PyObject *qualname = POP();
7 // 弹出栈顶元素, 得到PyCodeObject
8 PyObject *codeobj = POP();
9 // 此时栈里面还有一个元素
10 //.....
11 if (oparg & 0x08) {
12     assert(PyTuple_CheckExact(TOP()));
13     func->func_closure = POP();
14 }
15 if (oparg & 0x04) {
16     assert(PyDict_CheckExact(TOP()));
17     func->func_annotations = POP();
18 }
19 if (oparg & 0x02) {
20     assert(PyDict_CheckExact(TOP()));
21     func->func_kwdefaults = POP();
22 }
23
24 //当前 oparg 是 1, 所以 & 0x01 为真
25 //于是知道函数有默认值, 将其从栈顶弹出
26 //保存在函数的func_defaults成员中
27 if (oparg & 0x01) {
28     assert(PyTuple_CheckExact(TOP()));
29     func->func_defaults = POP();
30 }
31
32 PUSH((PyObject *)func);
33 DISPATCH();
34 }

```

通过以上命令可以很容易看出, 该指令除了创建函数对象, 还会处理参数的默认值。MAKE_FUNCTION的操作数表示当前运行时栈中存在默认值, 但具体有多少个是看不到的, 因为所有的默认值会按照顺序塞到一个PyTupleObject对象里面, 所以整体相当于是一个。

然后调用PyFunction_SetDefaults将该元组用为func_defaults成员保存, 在Python层面可以使用__defaults__访问。如此一来, 默认值也成为了PyFunctionObject对象的一部分, 函数和其参数的默认值最终被虚拟机捆绑在了一起, 它和PyCodeObject、global名字空间一样, 也被塞进了PyFunctionObject这个大礼包。

所以说PyFunctionObject这个嫁衣做的是很彻底的, 工具人PyFunctionObject, 点个赞。

```

1 int
2 PyFunction_SetDefaults(PyObject *op, PyObject *defaults)
3 {
4     // op 显然是一个函数
5     if (!PyFunction_Check(op)) {
6         PyErr_BadInternalCall();
7         return -1;
8     }
9     //defaults, 默认值组成的元组
10    if (defaults == Py_None)
11        defaults = NULL;
12    else if (defaults && PyTuple_Check(defaults)) {
13        Py_INCREF(defaults);
14    }
15    else {
16        PyErr_SetString(PyExc_SystemError, "non-tuple default args");
17        return -1;
18    }
19    //将PyFunctionObject对象的func_defaults成员设置为defaults

```

```

20     Py_XSETREF(((PyObject *)op)->func_defaults, defaults);
21     return 0;
22 }

```

然后我们还是以这个foo函数为例，看看不同的调用方式对应的底层实现：

```

1 def foo(a=1, b=2):
2     print(a + b)

```



由于函数参数都有默认值，此时可以不传参，看看这种方式在底层是如何处理的？

```

PyObject *
_PyFunction_FastCallDict(PyObject *func, PyObject *const *args,
                        Py_ssize_t nargs, PyObject *kwargs)
{
    //获取PyObject的PyCodeObject
    PyCodeObject *co = (PyCodeObject *)PyFunction_GET_CODE(func);
    //获取PyObject的global名字空间
    PyObject *globals = PyFunction_GET_GLOBALS(func);
    //获取PyObject的默认值
    PyObject *argdefs = PyFunction_GET_DEFAULTS(func);
    //一些额外属性
    PyObject *kwdefs, *closure, *name, *qualname;
    PyObject *kwtuple, **k;
    PyObject **d;
    Py_ssize_t nd, nk;
    PyObject *result;

    assert(func != NULL);
    assert(nargs >= 0);
    assert(nargs == 0 || args != NULL);
    assert(kwargs == NULL || PyDict_Check(kwargs));

    //这里判断能否进入快速通道，一个函数如果想进入快速通道要满足两个条件
    //1. 函数定义的时候不可以出现 *，也就是co_kwonlyargcount必须为 0
    //2. 函数调用时，必须都通过位置参数指定
    //所以这里检测co_kwonlyargcount是否为零、以及kwargs是否为空
    if (co->co_kwonlyargcount == 0 &&
        (kwargs == NULL || PyDict_GET_SIZE(kwargs) == 0) &&
        (co->co_flags & ~PyCF_MASK) == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE))
    {
        //上面的 if 虽然满足了，但是还不够，还要继续检测
        //argdefs == NULL，表示函数不可以有默认值
        //至于 nargs 是在call_function中传递的，它等于 oparg - nkargs
        //也就是“传递的参数个数”减去“通过关键字参数传递的参数个数”
        //换句话说就是通过位置参数传递的参数个数
        //但逻辑如果能执行到这里，说明 nkargs 一定为 0
        //因此这里的 nargs 就是传递的参数个数，也就是实参的个数
        //而co_argcount是函数的参数个数，也就是形参的个数
        //这里要求两者必须相等，当然了，在没有默认值、没有关键字参数的前提下
        //如果想正确调用函数，形参和实参的个数一定是相等的
        if (argdefs == NULL && co->co_argcount == nargs) {
            return function_code_fastcall(co, args, nargs, globals);
        }
        //但是上面的条件确实有点苛刻了，毕竟参数哪能没有默认值呢？
        //所以Python还提供了另外一种进入快速通道的方式
        //我们发现在有默认的前提下，如果能满足 nargs==0、也就是不传参
        //并且函数的参数都使用默认值，这样也能进入快速通道
        else if (nargs == 0 && argdefs != NULL
            && co->co_argcount == PyTuple_GET_SIZE(argdefs)) {
            args = PyTuple_ITEMS(argdefs);
            return function_code_fastcall(co, args, PyTuple_GET_SIZE(argdefs),
                globals);
        }
    }
}

```

```

    }
    //总结, 进入快速通道有两种办法:
    //1. 定义函数时不可以出现默认值, 然后全部通过位置参数传递
    //2. 如果出现默认值, 那么所有的参数必须都有默认值, 调用时不能传参, 参数都用默认值
}
// ...
// ...
}

```

古明地觉的 Python 小屋

而快速通道之前已经介绍过了, 这里就不再说了。总之想要进入快速通道, 条件还是蛮苛刻的。



显然此时就走不了快速通道了, 会进入通用通道。

```

PyObject *
_PyFunction_FastCallDict(PyObject *func, PyObject *const *args,
                        Py_ssize_t nargs, PyObject *kwargs)
{
    //.....
    if (co->co_kwonlyargcount == 0 &&
        (kwargs == NULL || PyDict_GET_SIZE(kwargs) == 0) &&
        (co->co_flags & ~PyCF_MASK) == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE))
    {
        //.....
    }
    //如果以上条件无法满足, 就只能走常规方法了
    //kwargs表示容纳关键字参数的字典, 这里获取内部的元素个数
    //也就是我们指定了多少个关键字参数
    nk = (kwargs != NULL) ? PyDict_GET_SIZE(kwargs) : 0;
    if (nk != 0) {
        //将 kwargs 转换成元组
        //比如: func(x=1, y=2, z=3), 那么kwargs就是 {"x": 1, "y": 2, "z": 3}
        //那么转换后的元组就是 ("x", 1, "y", 2, "z", 3)
        Py_ssize_t pos, i;
        //申请元组, 容量为 nk * 2, 因为要同时存储键和值
        kwtuple = PyTuple_New(2 * nk);
        if (kwtuple == NULL) {
            return NULL;
        }
        //获取内部的 ob_item, 也就是底层数组
        k = PyTuple_ITEMS(kwtuple);
        pos = i = 0;
        //将键值对设置进去
        while (PyDict_Next(kwargs, &pos, &k[i], &k[i+1])) {
            Py_INCREF(k[i]);
            Py_INCREF(k[i+1]);
            i += 2;
        }
        assert(i / 2 == nk);
    }
    //如果没有传递关键字参数, 那么 kwtuple 和 k 就设置为 NULL
    //此时说明一定有默认值, 否则会走快速通道
    else {
        kwtuple = NULL;
        k = NULL;
    }

    //获取函数的参数以及默认值, 并且这里的参数必须通过关键字参数进行传递
    kwdefs = PyFunction_GET_KW_DEFAULTS(func);
    //获取闭包、name、qualname
    closure = PyFunction_GET_CLOSURE(func);
    name = ((PyFunctionObject *)func) -> func_name;
    qualname = ((PyFunctionObject *)func) -> func_qualname;

    //获取默认参数的值、以及个数

```

```

    if (argdefs != NULL) {
        d = PyTuple_ITEMS(argdefs);
        nd = PyTuple_GET_SIZE(argdefs);
    }
    else {
        d = NULL;
        nd = 0;
    }

    //调用_PyEval_EvalCodeWithName, 传入函数的PyCodeObject对象以及参数信息
    //这里走的时候就是通用通道
    result = _PyEval_EvalCodeWithName((PyObject*)co, globals, (PyObject *)NULL,
                                      args, nargs,
                                      k, k != NULL ? k + 1 : NULL, nk, 2,
                                      d, nd, kwdefs,
                                      closure, name, qualname);

    Py_XDECREF(kwtuple);
    return result;
}

```

古明地觉的 Python小屋

所以重点就在 **PyEval_EvalCodeWithName**, 我们看一下它的逻辑。注意: 理解起来可能比较累, 可能需要多读即便。

```

1  PyObject *
2  _PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, PyObject *lo
3  cals,
4      PyObject *const *args, Py_ssize_t argcount,
5      PyObject *const *kwnames, PyObject *const *kwargs,
6      Py_ssize_t kwcount, int kwstep,
7      PyObject *const *defs, Py_ssize_t defcount,
8      PyObject *kwdefs, PyObject *closure,
9      PyObject *name, PyObject *qualname)
10 {
11     //PyCodeObject对象
12     PyCodeObject* co = (PyCodeObject*)_co;
13     //栈帧
14     PyFrameObject *f;
15     //返回值
16     PyObject *retval = NULL;
17     //f -> localsplus, co -> co_freevars
18     //freevars和闭包相关, 暂时不做讨论
19     PyObject **fastlocals, **freevars;
20     PyObject *x, *u;
21     //参数总个数: co->co_argcount + co->co_kwonlyargcount
22     //也就是: 可以通过位置参数传递的参数个数 + 只能通过关键字参数传递的参数个数
23     const Py_ssize_t total_args = co->co_argcount + co->co_kwonlyargcou
24 nt;
25     Py_ssize_t i, j, n;
26     PyObject *kwdict;
27     //获取线程状态对象
28     PyThreadState *tstate = _PyThreadState_GET();
29     assert(tstate != NULL);
30
31     if (globals == NULL) {
32         _PyErr_SetString(tstate, PyExc_SystemError,
33             "PyEval_EvalCodeEx: NULL globals");
34         return NULL;
35     }
36
37     //创建栈帧
38     f = _PyFrame_New_NoTrack(tstate, co, globals, locals);
39     if (f == NULL) {
40         return NULL;
41     }
42     fastlocals = f->f_localsplus;
43     freevars = f->f_localsplus + co->co_nlocals;

```

```

44 //还记得这个co_flags吗?
45 //如果它和0x08进行"与运算"结果为真,说明有**kwargs
46 //如果它和0x04进行"与运算"结果为真,说明有*args
47 if (co->co_flags & CO_VARKEYWORDS) {
48     //申请字典,用于 kwargs
49     kwdict = PyDict_New();
50     if (kwdict == NULL)
51         goto fail;
52     //我们说参数是有顺序的,*args和**kwargs在最后面
53     i = total_args;
54     if (co->co_flags & CO_VARARGS) {
55         i++;
56     }
57     //如果不存在 *args,那么将 kwargs 设置为fastlocals[i + 1]
58     //如果存在 *args,那么将 kwargs 设置为fastlocals[i]
59     SETLOCAL(i, kwdict);
60 }
61 // 说明没有 **kwargs,kwdict 为 NULL
62 else {
63     kwdict = NULL;
64 }
65
66 //argcount是实际传来的位置参数的个数
67 //co->co_argcount则是可以通过位置参数传递的参数个数
68 //如果argcount > co->co_argcount,证明有扩展位置参数,否则没有
69 if (argcount > co->co_argcount) {
70     //那么这里的n等于co->co_argcount
71     n = co->co_argcount;
72 }
73 else {
74     //没有扩展位置参数,那么调用者通过位置参数的方式传了几个,n就是几
75     n = argcount;
76 }
77
78
79 //然后我们仔细看一下这个 n,假设有一个函数 def bar(a, b, c=1, d=2, *args)
80 //如果argcount > co->co_argcount,说明传递的位置参数的个数超过了4,于是n为4
81 //但是如果我们只传递了两个,比如bar('a', 'b'),那么n显然为2
82 //下面就是将已经传递的参数的值依次设置到f_localsplus里面去
83 for (j = 0; j < n; j++) {
84     x = args[j];
85     Py_INCREF(x);
86     SETLOCAL(j, x);
87 }
88
89 //如果有 *args
90 if (co->co_flags & CO_VARARGS) {
91     u = PyTuple_FromArray(args + n, argcount - n);
92     if (u == NULL) {
93         goto fail;
94     }
95     //设置在 total_args 的位置,也就是 **kwargs 的前面
96     SETLOCAL(total_args, u);
97 }
98
99 //关键字参数,后面说
100 kwcount *= kwstep;
101 for (i = 0; i < kwcount; i += kwstep) {
102     //.....
103 }
104
105 //条件判断:
106 //argcount > co->co_argcount说明我们多传递了
107 //co->co_flags & CO_VARARGS为False,说明没有 *args

```

```

108     if ((argcount > co->co_argcount) && !(co->co_flags & CO_VARARGS)) {
109         //如果 if 条件成立, 说明位置参数传递过多, 并且还没有 *args
110         //显然直接报错:takes m positional arguments but n were given
111         too_many_positional(tstate, co, argcount, defcount, fastlocals);
112         goto fail;
113     }
114
115     //如果传入的参数个数比函数定义的参数的个数少, 那么证明有默认值
116     //defcount表示设置了默认值的参数个数
117     if (argcount < co->co_argcount) {
118         //显然 m = 参数总个数(不包括*args和**kwargs) - 默认参数的个数
119         Py_ssize_t m = co->co_argcount - defcount;
120         Py_ssize_t missing = 0;
121         //因此 m 就是需要传递的没有默认值的参数的个数
122         for (i = argcount; i < m; i++) {
123             //i = argcount是我们调用函数时传递的位置参数的总个数
124             //很明显如果参数足够, 那么 i < m 是不会满足的
125             //比如一个函数接收6个参数, 但是有两个是默认参数
126             //这就意味着调用者通过位置参数的方式传递的话, 需要至少传递4个, 那么m就是4
127             if (GETLOCAL(i) == NULL) {
128                 //但如果传递的参数不足四个, 那么GETLOCAL从f_localsplus中就获取不到值
129                 //而一旦找不到, missing++, 缺少的参数个数加一
130                 missing++;
131             }
132         }
133         //missing不为0, 表示缺少参数
134         if (missing) {
135             //直接抛出异常
136             //{func} missing {n} required positional arguments:
137             missing_arguments(tstate, co, missing, defcount, fastlocals
138 );
139             goto fail;
140         }
141
142         //下面可能难理解, 我们说这个m, 是需要由调用者传递的参数个数
143         //而n是以位置参数的形式传递过来的参数的个数
144         //如果比函数参数个数少, 那么n就是传来的参数个数
145         //如果比函数参数的个数大, 那么n则是函数参数的个数。比如:
146         /*
147         def bar(a, b, c, d=1, e=2, f=3):
148             pass
149         这是一个有6个参数的函数, 显然m是3
150         实际上函数定义好了, m就是一个不变的值了, 就是没有默认值的参数总个数
151         但我们调用时可以是bar(1, 2, 3), 也就是只传递3个, 那么这里的n就是3,
152         也可以是 bar(1, 2, 3, 4, 5), 那么显然n=5, 而m依旧是3
153         */
154         if (n > m)
155             //因此现在这里的逻辑就很好理解了, 假设调用的是 bar(1, 2, 3, 4, 5)
156             //由于有3个是默认参数, 那么调用时只传递 6-3=3 个就可以了, 但是这里传递了5个
157             //说明我们不想使用默认值, 想重新传递, 而使用默认值的只有最后一个参数
158             //因此这个 i 就是明明可以使用默认值、但却没有使用的参数的个数
159             i = n - m;
160         else
161             //如果按照位置参数传递能走到这一步, 说明已经不存在少传的情况了
162             //因此这个n至少是 >=m 的, 如果n == m的话, 那么i就是0
163             i = 0;
164         for (; i < defcount; i++) {
165             //默认参数的值一开始就已经被压入栈中,
166             //整体作为一个PyTupleObject对象, 被设置到了func_defaults这个域中
167             //但是对于函数的参数来讲, 肯定还要设置到f_localsplus里面去
168             //并且要在后面, 因为默认参数的顺序在非默认参数之后
169             if (GETLOCAL(m+i) == NULL) {
170                 //这里是把索引为i对应的值从func_defaults里面取出来
171                 //这个i要么是n-m, 要么是0

```



```

172 //还按照之前的例子, 函数接收6个参数, 但是我们传了5个
173 //因此我们只需要将最后一个、也就是索引为2的元素拷贝到f_localsplus里面
174 //而n=5, m=3, 显然i = 2。那么如果我们传递了3个呢?
175 //显然i是0, 因为此时n==m嘛, 那么就意味着默认参数都使用默认值
176 //既然如此, 那就从头开始开始拷
177 //同理传了4个参数, 证明第一个参数的默认值是不需要的
178 //那么就只需要再把后面两个拷过去就可以了
179 //显然要从索引为1的位置拷到结束, 而此时 n-m、也就是i, 正好为1
180 //所以, n-m就是"默认值组成的元组中需要拷贝到f_localsplus的第一个值
181 的索引"
182 //然后i < defcount; i++, 一直拷到结尾
183
184     PyObject *def = defs[i];
185     Py_INCREF(def);
186     //将值设置到f_localsplus里面, 这里显然索引是 m+i
187     //比如: def bar(a,b,c,d=1,e=2,f=3)
188     //bar(1, 2, 3, 4), 显然d不会使用默认值, 那么只需要把后两个默认值拷给
189     e和f即可
190     //显然e和f根据顺序在f_localsplus中对应索引为4、5
191     //m是3, i是n-m等于4-3等于1, 所以m+i正好是4,
192     //f_localsplus: [1, 2, 3, 4]
193     //PyTupleObject: (1, 2, 3)
194     //因此PyTupleObject中索引为i的元素, 拷贝到f_localsplus中正好是对应
195     m+i的位置
196     SETLOCAL(m+i, def);
197 }
198 }
199 }
200
201 //.....
202 return retval;
203 }

```

以上我们就知道了位置参数的默认值是怎么一回事了, 还是那句话, 逻辑理解起来不是很容易。但是核心就在于将默认值从func_defaults拷贝到f_localsplus里面。



这里我们传递了一个关键字参数, 此时也会走通用通道。并且在调用函数之前, 会先将**符号 b**和**对象 3**压入运行时栈。

```

1  PyObject *
2  _PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, PyObject *lo
3  cals,
4      PyObject *const *args, Py_ssize_t argcount,
5      PyObject *const *kwnames, PyObject *const *kwargs,
6      Py_ssize_t kwcount, int kwstep,
7      PyObject *const *defs, Py_ssize_t defcount,
8      PyObject *kwdefs, PyObject *closure,
9      PyObject *name, PyObject *qualname)
10 {
11     PyCodeObject* co = (PyCodeObject*)_co;
12     PyFrameObject *f;
13     //.....
14     f = _PyFrame_New_NoTrack(tstate, co, globals, locals);
15     //.....
16     if (co->co_flags & CO_VARKEYWORDS) {
17         //.....

```



```

18     }
19     else {
20         //.....
21     }
22
23     if (argcount > co->co_argcount) {
24         n = co->co_argcount;
25     }
26     else {
27         n = argcount;
28     }
29     for (j = 0; j < n; j++) {
30         //.....
31     }
32
33     if (co->co_flags & CO_VARARGS) {
34         //.....
35     }
36
37     //遍历关键字参数
38     kwcount *= kwstep;
39     for (i = 0; i < kwcount; i += kwstep) {
40         //符号表
41         PyObject **co_varnames;
42         //获取参数名
43         PyObject *keyword = kwnames[i];
44         //获取参数值
45         PyObject *value = kwargs[i];
46         Py_ssize_t j;
47
48         //函数参数必须是字符串
49         //比如在字典中你可以这么做: {**{1: "a", 2: "b"}}
50         //但你不可以这么做: dict(**{1: "a", 2: "b"})
51         if (keyword == NULL || !PyUnicode_Check(keyword)) {
52             _PyErr_Format(tstate, PyExc_TypeError,
53                 "%U() keywords must be strings",
54                 co->co_name);
55             goto fail;
56         }
57
58         co_varnames = ((PyTupleObject *) (co->co_varnames))->ob_item;
59         //遍历符号表, 看看符号表中是否存在和关键字参数相同的符号
60         //注意: 这里的j不是从0开始的, 而是从posonlyargcount开始
61         //因为在Python3.8中引入了/, 在/前面的参数只能通过位置参数传递
62         for (j = co->co_posonlyargcount; j < total_args; j++) {
63             //比如传递了 b=3, 那么要保证符号表中存在 "b" 这个符号
64             //如果有, 那么该参数就是通过关键字参数传递的
65             //如果符号表里面没有这个符号, 则看是否存在 **kwargs
66             //要是没有 **kwargs, 报错: got an unexpected keyword argument
67             PyObject *name = co_varnames[j];
68             if (name == keyword) {
69                 //找到了, 跳转到 kw_found 标签
70                 goto kw_found;
71             }
72         }
73
74         /* 逻辑和上面一样 */
75         for (j = co->co_posonlyargcount; j < total_args; j++) {
76             PyObject *name = co_varnames[j];
77             int cmp = PyObject_RichCompareBool(keyword, name, Py_EQ);
78             if (cmp > 0) {
79                 goto kw_found;
80             }
81             else if (cmp < 0) {

```

```

82         goto fail;
83     }
84 }
85
86 assert(j >= total_args);
87 if (kwdict == NULL) {
88     //如果符号表中没有出现指定的符号
89     //那么表示出现了一个不需要的关键字参数(**kwargs后续说)
90     if (co->co_posonlyargcount
91         && positional_only_passed_as_keyword(tstate, co,
92                                             kwcount, kwnames))
93     {
94         goto fail;
95     }
96
97     _PyErr_Format(tstate, PyExc_TypeError,
98                 "%U() got an unexpected keyword argument '%S'",
99                 co->co_name, keyword);
100     goto fail;
101 }
102
103 if (PyDict_SetItem(kwdict, keyword, value) == -1) {
104     goto fail;
105 }
106 continue;
107
108 kw_found:
109     //索引 j 就是该参数在 f_localsplus 中的索引
110     //但如果 GETLOCAL(j) != NULL, 说明已经通过位置参数指定了
111     if (GETLOCAL(j) != NULL) {
112         //参数重复
113         _PyErr_Format(tstate, PyExc_TypeError,
114                     "%U() got multiple values for argument '%S'",
115                     co->co_name, keyword);
116         goto fail;
117     }
118     //否则增加引用计数, 设置进去
119     Py_INCREF(value);
120     SETLOCAL(j, value);
121 }
122
123 //.....
124 return retval;
}

```

总结一下，虚拟机将会函数中出现的符号都记录在符号表(co_varnames)里面。在**foo(b=3)**的指令序列中，虚拟机在执行**CALL_FUNCTION**指令之前会将关键字参数的名字都压入到运行时栈，那么在执行**_PyEval_EvalCodeWithName**时就能利用运行时栈中保存的关键字参数的名字在co_varnames里面进行查找。

最妙的是，co_varnames里面的符号排列是有规律的。而且经过刚才的分析我们知道，在栈帧对象的f_localsplus所维护的内存中，也是按照相同的排列规律去存储参数。

所以在co_varnames中搜索到关键字参数的参数名时，我们可以根据所得到的索引直接设置f_localsplus，这就为默认参数设置了函数调用者希望的值。

为了理解清晰，我们可以再举个简单例子，总结一下。

```

1 def foo(a, b, c, d=1, e=2, f=3):
2     pass

```

对于上面这个函数，首先虚拟机知道调用者至少要给 a、b、c 传递参数。如果是**foo(1)**，那么 1 会传递给 a，但是 b 和 c 是没有接收到值的，所以报错。

如果是`foo(1, e=11, c=22, b=33)`，还是老规矩将 1 传递给 a，发现依旧不够，这时候会把希望寄托于关键字参数上。并且由于`f_localsplus`维护的内存中存储参数的顺序，和 `co_varnames` 中参数的顺序是一致的，所以关键字参数是不讲究顺序的。

当找到了 `e=11`，那么虚拟机通过符号表，就知道把 e 设置为 `f_localsplus` 中索引为 4 的地方。为什么索引是 4 呢？因为符号 e 在符号表中的索引是 4。

而 `c=22`，显然设置为索引为 2 的地方；`b=3`，设置为索引为 1 的地方。那么当位置参数和关键字参数都是设置完毕之后，虚拟机会再检测需要传递的参数、也就是没有默认值的参数，调用者有没有全部传递。

小结

这一篇的内容稍微有点枯燥，因为从Python的角度来看的话，就是一个传参罢了。

参数的传递可以使用位置参数、也可以使用关键字参数；如果带有默认值，我们也可以只给一部分参数传值，然后没收到值的参数使用默认值，收到值的参数使用我们传递的值。

而我们这里所做的事情，就是在看这些参数解析是怎么实现的？

思考题？

问题一：经过分析我们知道，关键字参数具体设置在`f_localsplus`中的哪一个地方，是通过将参数名代入到`co_varnames`里面查找所得到的。但如果这个关键字参数的参数名不在`co_varnames`里面，怎么办？

问题二：如果传递的位置参数，比`co_argcount`还要多，怎么办？

这里给出回答，首先是问题一，如果出现上述这种情况，说明指定了不存在的关键字参数，这意味着函数要有 `**kwargs`；对于问题二，位置参数传多了，说明函数要有 `*args`。

而 `*args` 和 `**kwargs`，我们下一篇介绍。

收录于合集 [#CPython 97](#)

[< 上一篇](#)

《源码探秘 CPython》61. `*args` 和 `**kwargs` 是如何解析的？

[下一篇 >](#)

《源码探秘 CPython》59. 函数是如何解析位置参数的？

喜欢此内容的人还喜欢

详解连接池参数设置（边调边看）
微服务实践



MySQL · 参数故事 · `thread_concurrency`
夜雨成诗



