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

原创 古明地觉 古明地觉的编程教室 2022-04-05 09:00

最后，再来聊一聊 *args 和 **kwargs。

```
def foo(a, b, *args, **kwargs):
    pass
print(foo.__code__.co_nlocals)  # 4
print(foo.__code__.co_argcount)  # 2
```

对于co_nlocals来说，它统计的是所有局部变量的个数，结果是4；但是对于co_argcount来说，统计的结果不包括 args 和 kwargs，因此结果是2。

然后*args可以接收多个位置参数，这些位置参数都会放在一个由 args 指向的元组中；**kwargs则可以接收多个关键字参数，而这些关键字参数（名字和值）会放在一个由 kwargs 指向的字典中。

事实上也确实如此，即使不从源码的角度来分析，从Python的实际使用中我们也能得出这个结论。

```
def foo(*args, **kwargs):
    print(args)
    print(kwargs)

foo(1, 2, 3, a=1, b=2, c=3)
"""
(1, 2, 3)
{'a': 1, 'b': 2, 'c': 3}
"""

foo(*(1, 2, 3), **{"a": 1, "b": 2, "c": 3})
"""
(1, 2, 3)
{'a': 1, 'b': 2, 'c': 3}
"""
```

当然啦，在调用的时候如果对一个元组或者列表、甚至是字符串使用 *，那么会将这个可迭代对象直接打散，相当于传递了多个位置参数。同理如果对一个字典使用 **，那么相当于传递了多个关键字参数。

下面我们就来看看扩展参数是如何实现的，首先还是进入到 **_PyEval_EvalCodeWithName** 这个函数里面来，当然这个函数应该很熟悉了，我们看看扩展参数的处理。

```
PyObject *
_PyEval_EvalCodeWithName(PyObject *_co, PyObject *globals, PyObject *lo
cals,
           //位置参数的相关信息
           PyObject *const *args, Py_ssize_t argcount,
           //关键字参数的相关信息
           PyObject *const *kwnames, PyObject *const *kwargs,
           Py_ssize_t kwcount, int kwstep, //关键字参数个数
           PyObject *const *defs, Py_ssize_t defcount,
           //默认值、闭包、函数名、全限定名等信息
           PyObject *kwdefs, PyObject *closure,
```

```
12              PyObject *name, PyObject *qualname)
13  {
14      //...
15      //判断是否出现 **kwargs
16      if (co->co_flags & CO_VARKEYWORDS) {
17          //创建一个字典，用于 kwargs
18          kwdict = PyDict_New();
19          if (kwdict == NULL)
20              goto fail;
21          //i 是参数总个数
22          i = total_args;
23          //如果还有 *args，那么i要加上1
24          //因为无论何时，关键字参数都要在位置参数后面
25          if (co->co_flags & CO_VARARGS) {
26              i++;
27          }
28          //如果没有 *args，那么kwdict要处于索引为 i 的位置
29          //如果有 *args，那么kwdit处于索引为 i + 1的位置
30          //然后放到f_localsplus中
31          SETLOCAL(i, kwdict);
32      }
33      else {
34          //如果没有 **kwargs 的话，那么为NULL
35          kwdict = NULL;
36      }
37
38      //这段逻辑之前之前介绍了
39      //是将位置参数(不包含扩展位置参数)拷贝到 f_localsplus中
40      if (argcount > co->co_argcount) {
41          n = co->co_argcount;
42      }
43      else {
44          n = argcount;
45      }
46      for (j = 0; j < n; j++) {
47          x = args[j];
48          Py_INCREF(x);
49          SETLOCAL(j, x);
50      }
51
52      //关键来了，将多余的位置参数拷贝到args里面去
53      if (co->co_flags & CO_VARARGS) {
54          //申请一个容量为 argcount - n 的元组
55          u = _PyTuple_FromArray(args + n, argcount - n);
56          if (u == NULL) {
57              goto fail;
58          }
59          //放到f -> f_localsplus里面去
60          SETLOCAL(total_args, u);
61      }
62
63      //下面就是拷贝扩展关键字参数
64      //使用索引遍历，按照顺序依次取出
65      //通过比较传递的关键字参数的符号是否已经出现在函数定义的参数中
66      //来判断传递的这个参数究竟是普通的关键字参数，还是扩展关键字参数
67      //比如:def foo(a, b, c, **kwargs)，foo(1, 2, c=3, d=4)
68      //那么显然关键字参数为c=3和d=4
69      //由于c已经出现在了函数定义的参数中，所以c就是一个普通的关键字参数
70      //但是d没有，因此d同时也是扩展关键字参数，要设置到kwargs这个字典里面
71      kwcount *= kwstep;
72      //下面可以看按到，关键字参数的名字和值，存放在不同的数组中
73      //按照索引遍历，将参数名和参数值依次取出
74      for (i = 0; i < kwcount; i += kwstep) {
75          PyObject **co_varnames; //符号表
```

```
        PyObject *keyword = kwnames[i];//关键字参数的"名字"
        PyObject *value = kwargs[i];//关键字参数的"值"
        Py_ssize_t j;
        //参数名必须是个字符串
        if (keyword == NULL || !PyUnicode_Check(keyword)) {
            _PyErr_Format(tstate, PyExc_TypeError,
                          "%U() keywords must be strings",
                          co->co_name);
            goto fail;
        }

        //拿到符号表，得到所有的符号，这样就知道函数参数都有哪些
        co_varnames = ((PyTupleObject *)(co->co_varnames))->ob_item;
        //我们看到内部又是一层for循环
        //首先外层循环是遍历所有的关键字参数，也就是我们传递的参数
        //而内层循环则是遍历符号表，看指定的参数名在符号表中是否存在
        for (j = co->co_posonlyargcount; j < total_args; j++) {
            PyObject *name = co_varnames[j];
            //如果相等，说明参数在符号表中已存在
            if (name == keyword) {
            //然后跳转到kw_found，将参数值设置在f_localsplus索引为 j 的位置
            //并且在标签内部，还会会检测该参数有没有通过位置参数传递
            //如果已经通过位置参数传递了，那么显然该参数被传递了两次
                goto kw_found;
            }
        }

        /* 逻辑和上面一样 */
        for (j = co->co_posonlyargcount; j < total_args; j++) {
            //...
        }

        assert(j >= total_args);
        //走到这里，说明 for循环不成立，也就是参数不在符号表中
        //说明传入了不存在的关键字参数，这个时候要检测 **kwargs
        //如果kwdict是NULL，说明函数没有 **kwargs，那么就直接报错了
        if (kwdict == NULL) {
            if (co->co_posonlyargcount
                && positional_only_passed_as_keyword(tstate, co,
                                                     kwcount, kwnames))
            {
                goto fail;
            }
            //也就是下面这个错误，{func} 收到了一个预料之外的关键字参数
            _PyErr_Format(tstate, PyExc_TypeError,
                          "%U() got an unexpected keyword argument '%S'",
                          co->co_name, keyword);
            goto fail;
        }
        //kwdict 不为 NULL，证明定义了 **kwargs
        //将参数名和参数值都设置到这个字典里面去
        //然后continue进入下一个关键字参数的判断逻辑
        if (PyDict_SetItem(kwdict, keyword, value) == -1) {
            goto fail;
        }
        continue;

    kw_found:
        //获取对应的符号，但是发现不为NULL，说明已经通过位置参数传递了
        if (GETLOCAL(j) != NULL) {
            //那么这里就报出一个TypeError，表示某个参数接收了多个值
            _PyErr_Format(tstate, PyExc_TypeError,
                          "%U() got multiple values for argument '%S'",
                          co->co_name, keyword);
```

```
            //比如说:def foo(a, b, c=1, d=2)
            //如果传递 foo(1, 2, c=3), 那么肯定没问题
            /*
              因为开始会把位置参数拷贝到f_localsplus里面
              所以此时f_localsplus(第一段内存)是 [1, 2, NULL, NULL]
              然后设置关键字参数的时候, 此时的j对应索引为2
              那么GETLOCAL(j)是NULL, 上面的if不成立所以不会报错
            */
            //但如果这样传递,foo(1, 2, 3, c=3)
            //那么 f_localsplus则是[1, 2, 3, NULL]
            //GETLOCAL(j)是 3, 不为NULL, 说明 j 这个位置已经被占了
            //既然有值了, 那么关键字参数就不能传递了, 否则就重复了
              goto fail;
        }
        //将 value 设置在 f_localsplus 中索引为 j 的位置
        //还是那句话,f_localsplus存储的值(PyObject *)、
        //和符号表中每一个符号对应的值, 在顺序上是保持一致的
        //比如变量 c 在符号表中索引为 2 的位置
        //那么f_localsplus[2]保存的就是变量 c 的值
        Py_INCREF(value);
        SETLOCAL(j, value);
    }

    /* 这里检测位置参数是否多传递了 */
    if ((argcount > co->co_argcount) && !(co->co_flags & CO_VARARGS)) {
        too_many_positional(tstate, co, argcount, defcount, fastlocals);
        goto fail;
    }

    //......
    return retval;
}
```

总的来说，虚拟机对参数进行处理的时候，机制还是有点复杂的。

其实扩展关键字参数的传递机制和普通关键字参数有很大的关系，我们之前分析参数的默认值时，已经看到了关键字参数的传递机制，这里我们再次看到了。

对于关键字参数，不论是否扩展，都会把符号和值分别按照对应顺序放在两个数组里面。然后虚拟机按照索引依次遍历存放符号的数组，对每一个符号都会和符号表co_varnames里面的符号逐个进行比对，发现在符号表中找不到我们传递的关键字参数的符号，那么就说明这是一个扩展关键字参数。然后就是我们在源码中看到的那样，如果函数定义了 **kwargs，那么kwdict就不为空，会把扩展关键字参数直接设置进去，否则就报错了，提示接收到了一个不期待的关键字参数。

_PyEval_EvalCodeWithName里面的内容非常多，我们每次都是截取指定的部分进行分析，可以自己再对着源码仔细读一遍。总之核心逻辑如下：

- **1）获取所有通过位置参数传递的参数个数，然后循环遍历将它们从运行时栈依次拷贝到 f_localsplus 指定的位置中；**
- **2）计算出可以通过位置参数传递的参数个数，如果"实际传递的位置参数的个数" 大于 "可以通过位置参数传递的参数个数"，那么会检测是否存在 *args。如果存在，那么将多余的位置参数拷贝到一个元组中；不存在，则报错：TypeError: function() takes 'm' positional argument but 'n' were given，其中 n 大于 m，表示接收了多个位置参数；**
- **3）如果实际传递的位置参数个数小于等于可以通过位置参数传递的参数个数，那么程序继续往下执行，检测关键字参数，它是通过两个数组来实现的，参数名和值是分开存储的；**
- **4）然后进行遍历，两层 for 循环，第一层 for 循环遍历存放关键字参数名的数组，第二层遍历符号表，会将传递参数名和符号表中的每一个符号进行比较；**
- **5）如果指定了不在符号表中的参数名，那么会检测是否定义了 **kwargs，如果没有则报错：TypeError: function() got an unexpected keyword argument 'xxx'，接收了一个不期望的参数 xxx；如果定义了 **kwargs，那么会设置在字典中；**
- **6）如果参数名在符号表中存在，那么跳转到 kw_found 标签，然后获取该符号对应的 value。如果 value 不为 NULL，那么证明该参数已经通过位置参数传递了，会报错：TypeError: function() got multiple values for argument 'xxx'，提示函数的参数 xxx**

接收了多个值；

- **7）最终所有的参数都会存在 f_localsplus 中，然后检测是否存在对应的 value 为 NULL 的符号，如果存在，那么检测是否具有默认值，有则使用默认值，没有则报错；**

以上就是参数的处理流程，用起来虽然简单，但分析具体实现时还是有点头疼的。但说实话，这部分内容也没有深挖的必要，大致了解就好。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》62. 函数的 local 名字空间

下一篇 ›
《源码探秘 CPython》60. 函数是如何解析关键字参数的?