

# 《源码探秘 CPython》87. 解密 map、filter、zip 底层实现，对比列表解析式

原创 古明地觉 古明地觉的编程教室 2022-05-12 08:30 发表于北京



微信扫一扫  
关注该公众号

收录于合集  
#CPython

97个 >



在程序开发中，map、filter、zip 可以说是非常常见了，下面来从源码的角度分析一下它们的实现原理。首先需要说明的是，这几个不是函数，而是类。



map 是将一个序列中的每个元素都作用于同一个函数（类、方法也可以），当然，我们知道调用 map 的时候并没有马上执行，而是返回一个 map 对象。既然是对象，那么底层必有相关的定义。

```
1 //bltinmodule.c
2 typedef struct {
3     PyObject_HEAD
4     PyObject *iters;
5     PyObject *func;
6 } mapobject;
```

解释一下里面的字段含义：

- PyObject\_HEAD：见过很多次了，它是任何对象都会有的头部信息。包含一个引用计数 ob\_refcnt、和一个指向类型对象的指针 ob\_type；
- iters：一个指向 PyTupleObject 的指针。以 `map(lambda x: x + 1, [1, 2, 3])` 为例，那么这里的 iters 就相当于 `([1, 2, 3], .iter_())`。至于为什么，分析源码的时候就知道了；
- func：显然就是函数指针了，PyFunctionObject \*；

通过底层结构体定义，我们也可以得知在调用 map 时并没有真正的执行；对于函数和可迭代对象，只是维护了两个指针去指向它。

而一个 PyObject 占用 16 字节，再加上两个 8 字节的指针总共 32 字节。因此在 64 位机器上，任何一个 map 对象所占大小都是 32 字节。

```
1 numbers = list(range(100000))
2 strings = ["abc", "def"]
3
4 # 都占32字节
5 print(map(lambda x: x * 3, numbers).__sizeof__()) # 32
6 print(map(lambda x: x * 3, strings).__sizeof__()) # 32
```

再来看看 map 的用法，Python 中的 map 不仅可以作用于一个序列，还可以作用于任意多个序列。

```
1 m1 = map(
2     lambda x: x[0] + x[1] + x[2],
3     [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
4 )
5 # x 就是列表里面的每一个元组
6 # x => (1, 2, 3)
```

```

7 # x => (4, 5, 6)
8 # x => (7, 8, 9)
9 print(list(m1)) # [6, 15, 24]
10
11 # map 还可以接收任意多个可迭代对象
12 m2 = map(
13     lambda x, y, z: x + y + z,
14     [1, 2, 3], [4, 5, 6], [7, 8, 9]
15 )
16 # (x, y, z) => (1, 4, 7)
17 # (x, y, z) => (2, 5, 8)
18 # (x, y, z) => (3, 6, 9)
19 print(list(m2)) # [12, 15, 18]
20 # 所以底层结构体中的 iters 在这里就相当于
21 # ([1, 2, 3].__iter__(), [4, 5, 6].__iter__(), [7, 8, 9].__iter__())
22
23 # 我们说 map 的第一个参数是一个函数, 后面可以接收任意多个可迭代对象
24 # 但是注意: 可迭代对象的数量 和 函数的参数个数 一定要匹配
25 m3 = map(
26     lambda x, y, z: str(x) + y + z,
27     [1, 2, 3], ["a", "b", "c"], "abc"
28 )
29 print(list(m3)) # ['1aa', '2bb', '3cc']
30
31 # 但是可迭代对象之间的元素个数不要求相等, 会以最短的为准
32 m3 = map(
33     lambda x, y, z: str(x) + y + z,
34     [1, 2, 3], ["a", "b", "c"], "ab"
35 )
36 print(list(m3)) # ['1aa', '2bb']
37
38 # 当然也支持更加复杂的形式
39 # (x, y) => ((1, 2), 3)
40 # (x, y) => ((2, 3), 4)
41 m5 = map(
42     lambda x, y: x[0] + x[1] + y,
43     [(1, 2), (2, 3)], [3, 4]
44 )
45 print(list(m5)) # [6, 9]

```

所以 map 会将后面所有可迭代对象中的每一个元素按照顺序依次取出, 然后传递到函数中, 因此 **函数的参数个数** 和 **可迭代对象的个数** 一定要相等。

那么map对象在底层是如何创建的呢? 很简单, 因为map是一个类, 那么调用的时候一定会执行里面的 `__new__` 方法。

```

1 //bltinmodule.c
2 static PyObject *
3 map_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
4 {
5     PyObject *it, *iters, *func;
6     mapobject *lz;
7     Py_ssize_t numargs, i;
8
9     //map对象在底层对应的是 mapobject
10    //map类本身在底层对应的则是 PyMap_Type
11    //PyArg_NoKeywords表示检验是否没有传递关键字参数
12    //如果没传递, 那么结果为真; 传递了, 结果为假;
13    if (type == &PyMap_Type && !_PyArg_NoKeywords("map", kwargs))
14        //可以看到 map 不接受关键字参数
15        //如果传递了, 那么会报如下错误:
16        //TypeError: map() takes no keyword arguments
17        return NULL;
18

```

```

19 //位置参数都在 args 里面, 上面的 kws 是关键字参数
20 //这里获取位置参数的个数, 1个函数、numargs - 1个可迭代对象
21 //而 args 是一个 PyTupleObject *
22 numargs = PyTuple_Size(args);
23 // 如果参数个数小于2
24 if (numargs < 2) {
25     //抛出 TypeError, 表示 map 至少接收两个位置参数
26     //一个函数 和 至少一个可迭代对象
27     PyErr_SetString(PyExc_TypeError,
28         "map() must have at least two arguments.");
29     return NULL;
30 }
31
32 // 申请一个元组, 容量为 numargs - 1
33 //用于存放传递的所有可迭代对象对应的迭代器
34 iters = PyTuple_New(numargs-1);
35 // 为NULL表示申请失败
36 if (iters == NULL)
37     return NULL;
38
39 // 依次循环
40 for (i=1 ; i<numargs ; i++) {
41     //表示获取索引为 i 的可迭代对象
42     //然后拿到对应的迭代器
43     it = PyObject_GetIter(PyTuple_GET_ITEM(args, i));
44     //为NULL表示获取失败
45     //但是iters这个元组已经申请了, 所以减少其引用计数, 将其销毁
46     if (it == NULL) {
47         Py_DECREF(iters);
48         return NULL;
49     }
50     // 将对应的迭代器设置在元组 iters 中
51     PyTuple_SET_ITEM(iters, i-1, it);
52 }
53
54 //调用 PyMap_Type 的 tp_alloc, 为其实例对象申请空间
55 lz = (mapobject *)type->tp_alloc(type, 0);
56 //为NULL表示申请失败, 减少iters的引用计数
57 if (lz == NULL) {
58     Py_DECREF(iters);
59     return NULL;
60 }
61 //让 lz 的 iters 字段等于 iters
62 lz->iters = iters;
63 //获取第一个参数, 也就是函数
64 func = PyTuple_GET_ITEM(args, 0);
65 //增加引用计数, 因为该函数被作为参数传递给 map 了
66 Py_INCREF(func);
67 //让 lz 的 func 字段等于 func
68 lz->func = func;
69
70 // 转成 PyObject *泛型指针, 然后返回
71 return (PyObject *)lz;
72 }

```

所以我们看到 map\_new 做的工作很简单, 就是实例化一个 map 对象, 然后对内部的成员进行赋值。我们用 Python 来模拟一下上述过程:

```

1 class MyMap:
2
3     def __new__(cls, *args, **kwargs):
4         if kwargs:
5             raise TypeError("MyMap不接受关键字参数")

```

```

6     numargs = len(args)
7     if numargs < 2:
8         raise TypeError("MyMap至少接收两个参数")
9     # 元组内部的元素不可以改变(除非本地修改), 所以这里使用列表来模拟
10    # 创建一个长度为 numargs - 1 的列表, 元素都是None, 模拟C中的NULL
11    iters = [None] * (numargs - 1)
12    i = 1
13    while i < numargs: # 逐步循环
14        it = iter(args[i]) # 获取可迭代对象, 得到其迭代器
15        iters[i - 1] = it # 设置在 iters 中
16        i += 1
17    # 为实例对象申请空间
18    instance = object.__new__(cls)
19    # 设置成员
20    instance.iters = iters
21    instance.func = args[0]
22    # 返回实例对象
23    return instance
24
25
26 m = MyMap(lambda x, y: x + y, [1, 2, 3], [11, 22, 33])
27 print(m) # <__main__.MyMap object at 0x00000167F4552E80>
28 print(m.func) # <function <lambda> at 0x0000023ABC4C51F0>
29 print(m.func(2, 3)) # 5
30
31 print(
32     m.iters
33 ) # [<list_iterator object at 0x00...>, <list_iterator object at 0x00..
34 .>]
    print([list(it) for it in m.iters]) # [[1, 2, 3], [11, 22, 33]]

```

我们看到非常简单, 这里我们没有设置构造函数 `__init__`, 这是因为 `map` 内部没有 `__init__`, 它的成员都是在 `__new__` 里面设置的。

```

1 # map的__init__ 实际上就是 object的__init__
2 print(map.__init__ is object.__init__) # True

```

调用 `map` 只是得到一个 `map` 对象, 整个过程并没有进行任何的计算。如果要计算的话, 我们可以调用 `__next__`、或者使用 `for` 循环等等。

```

1 m = map(lambda x: x + 1, [1, 2, 3, 4, 5])
2 print([i for i in m]) # [2, 3, 4, 5, 6]
3
4 # for 循环的背后本质上会调用迭代器的 __next__
5 # map 对象也是一个迭代器
6 m = map(lambda x: int(x) + 1, "12345")
7 while True:
8     try:
9         print(m.__next__())
10    except StopIteration:
11        break
12 """
13 2
14 3
15 4
16 5
17 6
18 """

```

当然上面都不是最好的方式, 如果只是单纯地将元素迭代出来, 而不做任何处理的话, 那么交给 `tuple`、`list`、`set` 等类型对象才是最佳的方式, 像 `tuple(m)`、`list(m)`、`set(m)` 等等。

所以如果你是 `[x for x in it]` 这种做法的话, 那么更建议你使用 `list(it)`, 效率会更高, 因为它用的是 C 中的 `for` 循环。当然不管是哪种做法, 底层都是一个不断调用 `__next__`、逐步迭代的过程。

下面我们来看看map底层是怎么做的？

```
1 static PyObject *
2 map_next(mapobject *lz)
3 {
4     //small_stack是一个 C 的栈数组, 里面存放 PyObject *
5     //显然它用来存放 map 中所有可迭代对象迭代出来的元素
6     //而这个_PY_FASTCALL_SMALL_STACK是一个宏
7     //定义在 Include/cpython/abstract.h 中, 值为 5
8     //如果函数参数的个数小于等于5的话, 便可申请在栈中
9     //之所以将其设置成5, 是为了不滥用 C 的栈, 从而减少栈溢出的风险
10    PyObject *small_stack[_PY_FASTCALL_SMALL_STACK];
11
12    //二级指针, 指向 small_stack 数组的首元素, 所以是 PyObject **
13    PyObject **stack;
14    //函数调用的返回值
15    PyObject *result = NULL;
16    //获取当前的线程状态对象
17    PyThreadState *tstate = _PyThreadState_GET();
18
19    //获取 iters 的长度, 也就是迭代器的数量
20    //当然同时也是调用函数时的参数数量
21    const Py_ssize_t niters = PyTuple_GET_SIZE(lz->iters);
22    //如果小于等于5, 那么获取这些迭代器中的元素之后
23    //直接使用在 C 栈里面申请的数组进行存储
24    if (niters <= (Py_ssize_t)Py_ARRAY_LENGTH(small_stack)) {
25        stack = small_stack;
26    }
27    else {
28        //如果超过了5, 那么不好意思, 只能在堆区重新申请了
29        stack = PyMem_Malloc(niters * sizeof(stack[0]));
30        //返回NULL, 表示申请失败, 说明没有内存了
31        if (stack == NULL) {
32            //这里传入线程状态对象, 会在内部设置异常
33            _PyErr_NoMemory(tstate);
34            return NULL;
35        }
36    }
37
38    //走到这里说明一切顺利, 那么下面就开始迭代了
39    Py_ssize_t nargs = 0;
40    //依次遍历, 得到每一个迭代器
41    for (Py_ssize_t i=0; i < niters; i++) {
42        //获取索引为i对应的迭代器
43        PyObject *it = PyTuple_GET_ITEM(lz->iters, i);
44        //拿到 __next__, 进行调用
45        PyObject *val = Py_TYPE(it)->tp_iternext(it);
46        //如果 val 为 NULL, 说明有一个迭代器迭代结束了, 或者出错了
47        //那么直接跳转到 exit 这个Label中
48        if (val == NULL) {
49            goto exit;
50        }
51        //将 val 设置在数组索引为i的位置中, 然后进行下一轮循环
52        //也就是获取下一个迭代器中的元素
53        stack[i] = val;
54        //nargs++, 和参数个数(迭代器个数)保持一致
55        //如果可迭代对象个数小于5, 比如3, 那么stack会申请在栈区
56        //但是在栈区申请的话, 长度默认为5, 因此后两个元素是无效的
57        //而在调用的时候需要指定有效的参数个数
58        nargs++;
59    }
60
```

```

61  /*
62  以 map(func, [1, 2, 3], ["xx", "yy", "zz"], [11, 22, 33]) 为例
63  那么 lz -> iters 就是 ([1, 2, 3].__iter__(),
64                        ["xx", "yy", "zz"].__iter__(),
65                        [11, 22, 33].__iter__())
66  第一次迭代, for 循环结束时, stack 指向数组 [1, "xx", 11]
67  第二次迭代, for 循环结束时, stack 指向数组 [2, "yy", 22]
68  第三次迭代, for 循环结束时, stack 指向数组 [3, "zz", 33]
69  */
70  //进行调用, tstate 是线程状态对象
71  //lz -> func 就是函数, stack 指向函数的首个参数
72  //nargs 就是参数个数
73  result = _PyObject_FastCall(tstate, lz->func, stack, nargs, NULL);
74
75  exit:
76  //调用完毕之后, 将stack里面指针指向的对象的引用计数减1
77  for (Py_ssize_t i=0; i < nargs; i++) {
78      Py_DECREF(stack[i]);
79  }
80  //不相等的話, 说明该stack是在堆区申请的, 要释放
81  if (stack != small_stack) {
82      PyMem_Free(stack);
83  }
84  // 返回result
85  return result;
86  }

```

我们用 Python 举例说明：

```

1  m = map(
2      lambda x, y, z: str(x) + y + str(z),
3      [1, 2, 3], ["xx", "yy", "zz"], [11, 22, 33]
4  )
5  # 第一次迭代, stack 指向 [1, "xx", 11]
6  print(m.__next__()) # 1xx11
7  # 第二次迭代, stack 指向 [2, "yy", 22]
8  print(m.__next__()) # 2yy22
9  # 第三次迭代, stack 指向 [3, "zz", 33]
10 print(m.__next__()) # 3zz33

```

以上就是 map 的用法。



然后是 filter 的实现原理，看完了 map 之后，再看 filter 就简单许多了。

```

1  lst = [1, 2, 3, 4, 5]
2  print(
3      list(filter(lambda x: x % 2 != 0, lst))
4  ) # [1, 3, 5]

```

filter 接收两个参数，第一个是函数（类、方法），第二个是可迭代对象。然后当我们迭代的时候，会将可迭代对象中的每一个元素都传入到函数中，如果返回的结果为真，则该元素保留；为假，则丢弃。

但是，其实第一个参数除了是一个可调用的对象之外，它还可以是 None。

```

1  lst = ["古明地觉", "", [], 123, 0, {}, [1]]

```

```

2  #会自动选择结果为真的元素
3  print(
4      list(filter(None, lst))
5  )  # ['古明地觉', 123, [1]]

```

至于为什么，一会看 filter 的实现就清楚了。

首先来看看 filter 对象的底层结构：

```

1  typedef struct {
2      PyObject_HEAD
3      PyObject *func;
4      PyObject *it;
5  } filterobject;

```

我们看到和 map 对象是一致的，没有什么区别。因为 map、filter 都不会立刻调用，而是返回一个相应的对象。

```

1  static PyObject *
2  filter_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
3  {
4      // 函数、可迭代对象
5      PyObject *func, *seq;
6      // 可迭代对象的迭代器
7      PyObject *it;
8      // 返回值, filter对象(指针)
9      filterobject *lz;
10
11     // filter也不接收关键字参数
12     if (type == &PyFilter_Type && !_PyArg_NoKeywords("filter", kwargs))
13         return NULL;
14
15     // 只接收两个参数
16     if (!_PyArg_UnpackTuple(args, "filter", 2, 2, &func, &seq))
17         return NULL;
18
19     // 获取seq对应的迭代器
20     it = PyObject_GetIter(seq);
21     if (it == NULL)
22         return NULL;
23
24     // 为filter对象申请空间
25     lz = (filterobject *)type->tp_alloc(type, 0);
26     if (lz == NULL) {
27         Py_DECREF(it);
28         return NULL;
29     }
30     // 增加函数的引用计数
31     Py_INCREF(func);
32     // 初始化成员
33     lz->func = func;
34     lz->it = it;
35
36     // 返回
37     return (PyObject *)lz;
38 }

```

和map是类似的，因为本质上它们做的事情都是差不多的，下面看看迭代过程。

```

1  static PyObject *
2  filter_next(filterobject *lz)
3  {
4      //迭代器中迭代出来的每一个元素
5      PyObject *item;

```

```

6 //迭代器
7 PyObject *it = lz->it;
8 //是否为真, 1 表示真、0 表示假
9 long ok;
10 //指针, 用于保存 __next__
11 PyObject *(*iternext)(PyObject *);
12 //如果 func == None 或者 func == bool, 那么checktrue为真
13 //后续会走单独的方法, 所以给func传递一个None是完全合法的
14 int checktrue = lz->func == Py_None || lz->func == (PyObject *)&PyBo
15 ol_Type;
16 // 迭代器的 __next__ 方法
17 iternext = *Py_TYPE(it)->tp_iternext;
18 // 无限循环
19 for (;;) {
20     // 迭代器所迭代出来的元素
21     item = iternext(it);
22     if (item == NULL)
23         return NULL;
24
25     // 如果checkture, 或者说如果 func == None || func == bool
26     if (checktrue) {
27         //那么直接走PyObject_IsTrue, 判断item是否为真
28         //另外我们在写 if 语句的时候经常会写 if item: 这种形式
29         //但是很少会写 if bool(item):
30         //因为这两者底层都是执行了 PyObject_IsTrue
31         //但是对于 if 而言, bool(item)多了一次调用, 速度稍微慢一些
32         ok = PyObject_IsTrue(item);
33     } else {
34         //否则的话, 会调用我们传递的func
35         //这里的 good 就是函数调用的返回值
36         PyObject *good;
37         // 调用函数, 将返回值赋值给good
38         good = PyObject_CallFunctionObjArgs(lz->func, item, NULL);
39         if (good == NULL) {
40             Py_DECREF(item);
41             return NULL;
42         }
43         //判断 good 是否为真
44         ok = PyObject_IsTrue(good);
45         //减少其引用计数, 因为它不被外界所使用
46         Py_DECREF(good);
47     }
48     //如果ok大于0, 说明将 lz -> func(item)的结果为真
49     //那么将 item 返回
50     if (ok > 0)
51         return item;
52     //同时减少其引用计数
53     Py_DECREF(item);
54     //小于0的话, 表示PyObject_IsTrue调用失败了, 返回-1
55     //但我们使用 Python 时不会发生, 除非解释器本身处 bug 了
56     if (ok < 0)
57         return NULL;
58     //否则说明 ok 等于 0, item 为假
59     //那么进行下一轮循环, 直到找到一个为真的元素
60 }
}

```

我们用 Python 测试一下：

```

1 f = filter(None, [None, "", (), False, 123])
2 # 因为会从可迭代对象里面找到一个为真的元素并返回
3 # 但只有最后一个元素为真, 所以返回 123
4 print(f.__next__()) # 123
5

```



```

6  # 这里所有的元素全部为假, 于是第一次迭代就会抛异常
7  f = filter(None, [None, "", (), False])
8  try:
9      print(f.__next__())
10 except StopIteration:
11     print("迭代结束") # 迭代结束

```

所以看到这里你还觉得 Python 神秘吗, 从源代码的层面我们看的非常流畅, 只要你有一定的 C 语言基础即可。还是那句话, 尽管我们不可能写一个解释器, 因为背后涉及的东西太多了, 但至少我们在看的过程中, 很清楚底层到底在做什么。而且这背后的实现, 如果让你设计一个方案的话, 那么相信你也一样可以做到。



最后看看 zip, 它的中文意思是拉链, 很形象, 就是将多个可迭代对象的元素按照顺序依次组合起来。

```

print(
    list(zip([1, 2, 3], [11, 22, 33], [111, 222, 333]))
) # [(1, 11, 111), (2, 22, 222), (3, 33, 333)]

# 其实 zip 完全可以用 map 实现:
print(
    list(map(lambda x, y, z: (x, y, z),
             [1, 2, 3], [11, 22, 33], [111, 222, 333]))
) # [(1, 11, 111), (2, 22, 222), (3, 33, 333)]

print(
    list(map(lambda *args: args,
             [1, 2, 3], [11, 22, 33], [111, 222, 333]))
) # [(1, 11, 111), (2, 22, 222), (3, 33, 333)]

```

👤 古明地觉的 Python 小屋

所以 zip 的底层实现同样很简单, 我们来看一下:

```

1 typedef struct {
2     PyObject_HEAD
3     Py_ssize_t tuplesize;
4     PyObject *ittuple;
5     PyObject *result;
6 } zipobject;

```

以上便是zip对象的底层定义, 这些字段的含义, 我们暂时先不讨论, 它们会体现在zip\_new方法中, 我们到时候再说。

目前我们根据结构体里面的成员, 可以得到一个 zipobject 占 40 字节,  $16 + 8 + 8 + 8$ , 那么结果是不是这样呢? 我们来试一下就知道了。

```

1 z1 = zip([1, 2, 3], [11, 22, 33])
2 z2 = zip([1, 2, 3, 4], [11, 22, 33, 44])
3 z3 = zip([1, 2, 3], [11, 22, 33], [111, 222, 333])
4
5 print(z1.__sizeof__()) # 40
6 print(z2.__sizeof__()) # 40
7 print(z3.__sizeof__()) # 40

```

我们分析的没有错, 任何一个 zip 对象所占的大小都是 40 字节。所以在计算内存大小的时候, 有

人会好奇这到底是怎么计算的，其实就是根据底层的结构体进行计算的。

下面看看 zip 对象是如何被实例化的。

```
1 static PyObject *
2 zip_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
3 {
4     //zip 对象的指针
5     zipobject *lz;
6     //循环变量
7     Py_ssize_t i;
8     //所有可迭代对象的迭代器组成的元组
9     PyObject *ittuple;
10    // "代码中有体现"
11    PyObject *result;
12    //可迭代对象的数量
13    Py_ssize_t tuplesize;
14
15    //zip同样不需要关键字参数
16    //但是在3.10的时候将会提供一个关键字参数strict
17    //如果为True, 表示可迭代对象之间的长度必须相等, 否则报错
18    //strict 如果为False, 则和目前是等价的, 会自动以短的为准
19    if (type == &PyZip_Type && !_PyArg_NoKeywords("zip", kwargs))
20        return NULL;
21
22    assert(PyTuple_Check(args));
23    //获取可迭代对象的数量
24    tuplesize = PyTuple_GET_SIZE(args);
25
26    //申请一个元组, 长度为 tuplesize
27    //用于存放可迭代对象对应的迭代器
28    ittuple = PyTuple_New(tuplesize);
29    //为NULL表示申请失败
30    if (ittuple == NULL)
31        return NULL;
32    //然后依次遍历
33    for (i=0; i < tuplesize; ++i) {
34        //获取传递的可迭代对象
35        PyObject *item = PyTuple_GET_ITEM(args, i);
36        //通过PyObject_GetIter获取对应的迭代器
37        PyObject *it = PyObject_GetIter(item);
38        if (it == NULL) {
39            // 为NULL表示获取失败, 减少ittuple的引用计数, 返回NULL
40            Py_DECREF(ittuple);
41            return NULL;
42        }
43        //设置在ittuple中
44        PyTuple_SET_ITEM(ittuple, i, it);
45    }
46
47    //这里又申请一个元组result, 长度也为tuplesize
48    result = PyTuple_New(tuplesize);
49    if (result == NULL) {
50        Py_DECREF(ittuple);
51        return NULL;
52    }
53    //然后将内部的所有元素都设置为None
54    //Py_None就是Python中的None
55    for (i=0 ; i < tuplesize ; i++) {
56        Py_INCREF(Py_None);
57        PyTuple_SET_ITEM(result, i, Py_None);
58    }
59
60    //申请一个zip对象
```

```

61  lz = (zipobject *)type->tp_alloc(type, 0);
62  //申请失败减少引用计数, 返回NULL
63  if (lz == NULL) {
64      Py_DECREF(ittuple);
65      Py_DECREF(result);
66      return NULL;
67  }
68  // 初始化成员
69  lz->ittuple = ittuple;
70  lz->tuplesize = tuplesize;
71  lz->result = result;
72
73  // 转成泛型指针PyObject *之后返回
74  return (PyObject *)lz;
75 }

```

再看看，zip对象的定义：

```

1  typedef struct {
2      PyObject_HEAD
3      Py_ssize_t tuplesize;
4      PyObject *ittuple;
5      PyObject *result;
6  } zipobject;

```

如果以 `zip([1, 2, 3], [11, 22, 33], [111, 222, 333])` 为例的话，那么：

- tuplesize 为 **3**
- ittuple 为 `([1, 2, 3].__iter__(), [11, 22, 33].__iter__(), [111, 222, 333].__iter__())`
- result 为 `(None, None, None)`

所以目前来说，其它的很好理解，唯独这个 result 让人有点懵，搞不懂它是干什么的（不用想肯定是每次迭代得到的值）。不过既然有这个成员，那就说明它肯定有用武之地，而派上用场的地方显然是在迭代的时候。

```

1  static PyObject *
2  zip_next(zipobject *lz)
3  {
4      //循环变量
5      Py_ssize_t i;
6      //可迭代对象的数量, 或者说迭代器的数量
7      Py_ssize_t tuplesize = lz->tuplesize;
8      //(None, None, ...)
9      PyObject *result = lz->result;
10     //每一个迭代器
11     PyObject *it;
12
13     //代码中体现
14     PyObject *item;
15     PyObject *olditem;
16
17     //tuplesize == 0, 直接返回
18     if (tuplesize == 0)
19         return NULL;
20     //如果 result 的引用计数为1
21     //证明该元组的空间的被申请了
22     if (Py_REFCNT(result) == 1) {
23         //要作为返回值返回, 所以引用计数加 1
24         Py_INCREF(result);
25         //遍历
26         for (i=0 ; i < tuplesize ; i++) {
27             //依次获取每一个迭代器
28             it = PyTuple_GET_ITEM(lz->ittuple, i);
29             //迭代出相应的元素

```

```

30     item = (*Py_TYPE(it)->tp_iternext)(it);
31     //如果出现了NULL, 证明迭代结束了, 会直接停止
32     //所以会以元素最少的可迭代对象(迭代器)为准
33     if (item == NULL) {
34         Py_DECREF(result);
35         return NULL;
36     }
37     //设置在 result 里面
38     //但是要先获取result中原来的元素, 并将其引用计数减1
39     //因为元组不再持有对它的引用
40     olditem = PyTuple_GET_ITEM(result, i);
41     PyTuple_SET_ITEM(result, i, item);
42     Py_DECREF(olditem);
43 }
44 } else {
45     // 否则的话同样的逻辑, 只不过需要自己重新手动申请一个tuple
46     result = PyTuple_New(tuplesize);
47     if (result == NULL)
48         return NULL;
49     // 然后下面的逻辑是类似的
50     for (i=0 ; i < tuplesize ; i++) {
51         it = PyTuple_GET_ITEM(lz->ittuple, i);
52         item = (*Py_TYPE(it)->tp_iternext)(it);
53         if (item == NULL) {
54             Py_DECREF(result);
55             return NULL;
56         }
57         PyTuple_SET_ITEM(result, i, item);
58     }
59 }
60 //返回元组 result
61 return result;
62 }

```

因为 zip 对象每次迭代出来的是一个元组, 所以 result 是一个元组。而 zip 接收了 3 个可迭代对象, 所以每次迭代出来的元组会包含 3 个元素。

```

1 z = zip([1, 2, 3], [11, 22, 33])
2 print(z.__next__()) # (1, 11)
3
4 #即使只有一个可迭代对象, 依旧是一个元组
5 #因为底层返回的result就是一个元组
6 z = zip([1, 2, 3])
7 print(z.__next__()) # (1,)
8
9 #可迭代对象的嵌套也是一样的规律
10 #直接把里面的列表看成一个标量即可
11 z = zip([[1, 2, 3], [11, 22, 33]])
12 print(z.__next__()) # ([1, 2, 3],)

```



## map、filter 和 列表解析式的区别



其实在使用 map、filter 的时候, 我们完全可以使用列表解析来实现。比如:

```

1 lst = [1, 2, 3, 4]
2
3 print([str(_) for _ in lst]) # ['1', '2', '3', '4']

```

```
4 print(list(map(str, lst))) # ['1', '2', '3', '4']
```

这两者之间实际上是没有太大区别的，都是将 lst 中的元素一个一个迭代出来、然后调用 str、返回结果。只不过先有的 map、filter，后有的列表解析式，但 map、filter 依旧保留了下来。

如果非要找出区别话，就是列表解析使用的是 Python 的 for 循环，而调用 map 使用的是 C 的 for 循环。从这个角度来说，使用 map 的效率会更高一些。

```
%timeit [str(_) for _ in range(1000)]
```

175  $\mu$ s  $\pm$  1.88  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

```
%timeit list(map(str, range(1000)))
```

139  $\mu$ s  $\pm$  1.5  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

所以后者的效率稍微更高一些，因为列表解析用的是 Python 的 for 循环，`list(map(func, iter))` 用的是 C 的 for 循环。但是注意：如果是下面这种做法的话，会得到相反的结果。

```
%timeit [x + 1 for x in range(1000)]
```

57.6  $\mu$ s  $\pm$  7.07  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

```
%timeit list(map(lambda x: x + 1, range(1000)))
```

96.9  $\mu$ s  $\pm$  2.38  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

我们看到 map 变慢了，其实原因很简单，后者多了一层匿名函数的调用，所以速度变慢了。如果列表解析也是函数调用的话：

```
%timeit [x + 1 for x in range(1000)]
```

57.6  $\mu$ s  $\pm$  7.07  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

```
%timeit list(map(lambda x: x + 1, range(1000)))
```

96.9  $\mu$ s  $\pm$  2.38  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

```
%timeit [(lambda x: x + 1)(x) for x in range(1000)]
```

170  $\mu$ s  $\pm$  6.09  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10000 loops each)

会发现速度更慢了，当然这种做法完全是吃饱了撑的。之所以说这些，是想说明在同等条件下，`list(map)` 这种形式是要比列表解析快的。

当然在工作中，这两者都是可以使用，这点效率上的差别其实不用太在意，如果真的到了需要在乎这点差别的时候，那么你应该考虑的是换一门更有效率的静态语言。

filter 和 列表解析之间的差别，也是如此。因为 map 和 filter 用的都是 C 的循环，所以都会比列表解析快一点。

```
[x for x in (False,) * 1000 + (True,) if x]
```

[True]

```
list(filter(lambda x: x, (False,) * 1000 + (True,)))
```

```
[True]
```

```
list(filter(None, (False,) * 1000 + (True,)))
```

```
[True]
```

古明地觉的 Python 小屋

对于过滤含有 1000 个 False 和 1 个 True 的元组，它们的结果都是一样的，但是谁的效率更高呢？首先第一种方式 肯定比 第二种方式快，因为第二种方式涉及到函数的调用；但是第三种方式，我们知道它在底层会走单独的分支，所以再加上之前的结论，我们认为第三种方式是最快的。

```
%timeit [x for x in (False,) * 1000 + (True,) if x]
```

```
19.7 µs ± 689 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%timeit list(filter(lambda x: x, (False,) * 1000 + (True,)))
```

```
64.2 µs ± 1.32 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%timeit list(filter(None, (False,) * 1000 + (True,)))
```

```
10.2 µs ± 65.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

古明地觉的 Python 小屋

结果也确实是我们分析的这样，当然我们说在底层 None 和 bool 都会走相同的分支，所以这里将 None 换成 bool 也是可以的。虽然 bool 是一个类，但是通过 filter\_next 函数我们知道，底层不会进行调用，也是直接使用 PyObject\_IsTrue，可以将 None 换成 bool 看看结果如何，应该是差不多的。

```
%timeit list(filter(bool, (False,) * 1000 + (True,)))
```

```
10.2 µs ± 65.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%timeit list(filter(bool, (False,) * 1000 + (True,)))
```

```
10.1 µs ± 81.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

古明地觉的 Python 小屋



## 小结



以上我们就从源码的角度介绍了 map、filter、zip 三个内置类，它们在工作中绝对会出现。

并且 map、filter 也可以使用列表解析替代，如果逻辑简单的话，比如获取为真的元素，那么通过 `list(filter(None, lst))` 实现即可，效率更高，因为它走的是相当于是 C 的循环。

但如果执行的逻辑比较复杂的话，那么对于 map、filter 而言就要写匿名函数。比如获取大于 3 的元素，那么就需要使用 `list(filter(lambda x: x > 3, lst))` 这种形式了。而我们说它的效率此时是不如列表解析 `[x for x in lst if x > 3]` 的，因为前者多了一层函数调用。

但是在工作中，这两种方式都是可以的，使用哪一种就看个人喜好。到此我们发现，如果排除那一点点效率上的差异，那么确实有列表解析式就完全足够了，因为列表解析式可以同时实现 map、filter 的功能，而且表达上也更加地直观。只不过是 map、filter 先出现，然后才有的列表解析式，但是前者依旧被保留了下来。



当然 map、filter 返回的是一个可迭代对象，它不会立即计算，可以节省资源；不过这个功能，我们也可以通过生成器表达式来实现。

收录于合集 [#CPython 97](#)

[< 上一篇](#)

《源码探秘 CPython》88. 侵入 Python 虚拟机，动态修改底层数据结构和运行时

[下一篇 >](#)

《源码探秘 CPython》86. 内置函数解析

喜欢此内容的人还喜欢

真香！超全，Python 中常见的配置文件写法  
Python丹卿



你需要知道的 19 个 console 实用调试技巧！  
前端充电宝



客户给100块要做个百度，我用10行Python代码搞定  
Python丹卿

