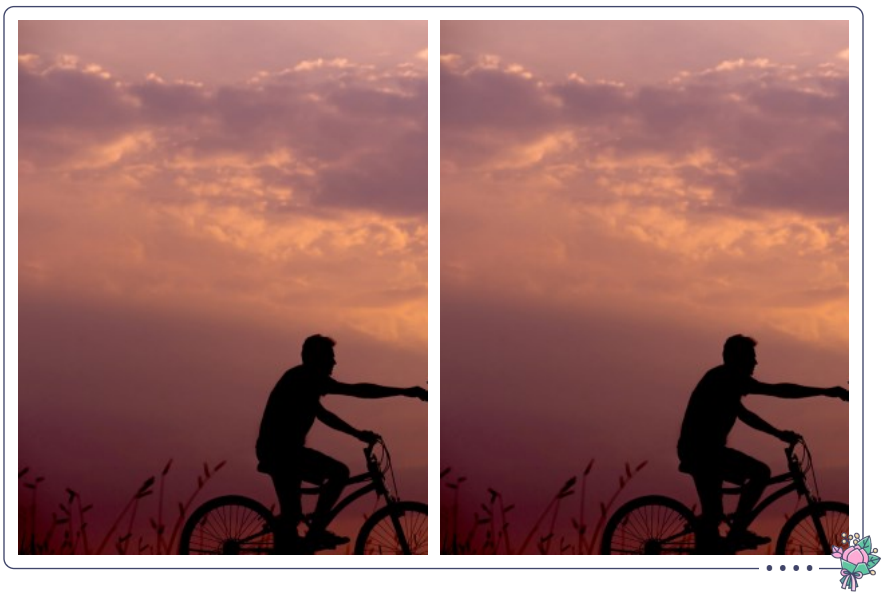
《源码探秘 CPython》42. 迭代器的实现原理

原创 古明地觉 古明地觉的编程教室 2022-03-06 09:00

收录于合集
#CPython
97个 >



楔子

在Python里面，只要类型对象实现了__iter__，那么它的实例对象就被称为**可迭代对象(Iterable)**，比如字符串、元组、列表、字典、集合等等。而整数、浮点数，由于其类型对象没有实现__iter__，所以它们不是**可迭代对象**。

```
from typing import Iterable

print(
    isinstance("", Iterable),
    isinstance((), Iterable),
    isinstance([], Iterable),
    isinstance({}, Iterable),
    isinstance(set(), Iterable),
)  # True True True True True

print(
    isinstance(0, Iterable),
    isinstance(0.0, Iterable),
)  # False False
```

可迭代对象的一大特点就是它可以使用for循环进行遍历，但是能被for循环遍历的则不一定是可迭代对象。我们举个栗子：

```
class A:

    def __getitem__(self, item):
        return f"参数item: {item}"


a = A()
#内部定义了 __getitem__
```

```
#首先可以让实例对象像字典一样访问属性
print(a["name"])  # 参数item: name
print(a["satori"])  # 参数item: satori

# 此外还可以像可迭代对象一样被for循环
# 循环的时候会自动给item传值，0 1 2 3...
# 如果内部出现了StopIteration，循环结束
# 否则会一直循环下去。这里我们手动break
for idx, val in enumerate(a):
    print(val)
    if idx == 5:
        break
"""
参数item: 0
参数item: 1
参数item: 2
参数item: 3
参数item: 4
参数item: 5
"""
```

所以实现了__getitem__的类的实例，也是可以被for循环的，但它并不是可迭代对象。

```
from typing import Iterable
print(isinstance(a, Iterable))  # False
```

打印的结果是 False。

总之判断一个对象是否是可迭代对象，就看它的类型对象有没有实现__iter__。可迭代对象我们知道了，那什么是迭代器呢？很简单，调用可迭代对象的__iter__方法，得到的就是迭代器。

迭代器的创建

不同类型的对象，都有自己的迭代器，举个栗子。

```
lst = [1, 2, 3]
#底层调用的其实是list.__iter__(lst)
#或者说PyList_Type.tp_iter(lst)
it = lst.__iter__()
print(it)  # <list_iterator object at 0x000001DC6E898640>
print(
    str.__iter__("")
)  # <str_iterator object at 0x000001DC911B8070>
print(
    tuple.__iter__(())
)  # <tuple_iterator object at 0x000001DC911B8070>
```

迭代器也是可迭代对象，只不过迭代器内部的__iter__返回的还是它本身。当然啦，在创建迭代器的时候，我们更常用内置函数iter。

```
lst = [1, 2, 3]
# 等价于 type(lst).__iter__(lst)
it = iter(lst)
```

但是**iter函数**还有一个鲜为人知的用法，我们来看一下：

```
val = 0
```

```
2
3  def foo():
4      global val
5      val += 1
6      return val
7
8
9  # iter可以接收一个参数: iter(可迭代对象)
10 # iter也可以接收两个参数: iter(可调用对象, value)
11 for i in iter(foo, 5):
12     print(i)
13 """
14 1
15 2
16 3
17 4
18 """
```

进行迭代的时候，会不停地调用接收的可调用对象，直到返回值等于传递第二个参数value，在底层被称为哨兵，然后终止迭代。我们看一下iter函数的底层实现。

```
1  static PyObject *
2  builtin_iter(PyObject *self, PyObject *const *args, Py_ssize_t nargs)
3  {
4      PyObject *v;
5
6      // iter函数要么接收一个参数, 要么接收两个参数
7      if (!_PyArg_CheckPositional("iter", nargs, 1, 2))
8          return NULL;
9      v = args[0];
10     //如果接收一个参数
11     //那么直接使用 PyObject_GetIter 获取对应的迭代器即可
12     //可迭代对象的类型不同, 那么得到的迭代器也不同
13     if (nargs == 1)
14         return PyObject_GetIter(v);
15     // 如果接收的不是一个参数, 那么一定是两个参数
16     // 如果是两个参数, 那么第一个参数一定是可调用对象
17     if (!PyCallable_Check(v)) {
18         PyErr_SetString(PyExc_TypeError,
19                         "iter(v, w): v must be callable");
20         return NULL;
21     }
22     // 获取value(哨兵)
23     PyObject *sentinel = args[1];
24     //调用PyCallIter_New
25     //得到一个可调用的迭代器, calliterobject 对象
26     /*
27     位于 Objects/iterobject.c 中
28     typedef struct {
29         PyObject_HEAD
30         PyObject *it_callable;
31         PyObject *it_sentinel;
32   } calliterobject;
33     */
34     return PyCallIter_New(v, sentinel);
35 }
```

以上就是iter函数的内部逻辑，既可以接收一个参数，也可以接收两个参数。这里我们只看接收一个可迭代对象的情况，所以核心就在于**PyObject_GetIter**，它是根据可迭代对象生成迭代器的关键，我们来看一下它的逻辑是怎么样的？该函数定义在**Objects/abstract.c**中。

```
1  PyObject *
2  PyObject_GetIter(PyObject *o)
```

```c
{
    //获取可迭代对象的类型对象
    PyTypeObject *t = Py_TYPE(o);
    //我们说类型对象定义的操作，决定了实例对象的行为
    //实例对象调用的那些方法都是定义在类型对象里面的
    //还是那句话:obj.func()等价于type(obj).func(obj)
    getiterfunc f;
    //所以这里是获取类型对象的tp_iter成员
    //也就是Python中的 __iter__
    f = t->tp_iter;
    //如果 f 为 NULL
    //说明该类型对象内部的tp_iter成员被初始化为NULL
    //即内部没有定义 __iter__
    //像str、tuple、list等类型对象，它们的tp_iter成员都是不为NULL的
    if (f == NULL) {
      //如果 tp_iter 为 NULL，那么解释器会退而求其次
      //检测该类型对象中是否定义了 __getitem__
      //如果定义了，那么直接调用PySeqIter_New
      //得到一个seqiterobject对象
      //下面的PySequence_Check负责检测类型对象是否实现了__getitem__
      //__getitem__ 对应 tp_as_sequence->sq_item
        if (PySequence_Check(o))
            return PySeqIter_New(o);
        // 走到这里说明该类型对象既没有__iter__、也没有__getitem__
        // 因此它的实例对象不具备可迭代的性质，于是抛出异常
        return type_error("'%.200s' object is not iterable", o);
    }
    else {
        // 否则说明定义了__iter__，于是直接进行调用
        // Py_TYPE(o)->tp_iter(o) 返回对应的迭代器
        PyObject *res = (*f)(o);
        // 但如果返回值res不为NULL、并且还不是迭代器
        // 证明 __iter__ 的返回值有问题，于是抛出异常
        if (res != NULL && !PyIter_Check(res)) {
            PyErr_Format(PyExc_TypeError,
                         "iter() returned non-iterator "
                         "of type '%.100s'",
                         Py_TYPE(res)->tp_name);
            Py_DECREF(res);
            res = NULL;
        }
        // 返回 res
        return res;
    }
}
```

所以我们看到这便是 iter 函数的底层实现，但是里面提到了__getitem__。我们说如果类型对象内部没有定义 __iter__，那么解释器会退而求其次检测内部是否定义了__getitem__。

因此以上就是迭代器的创建过程，每个可迭代对象都有自己的迭代器，而迭代器本质上只是对原始数据的一层封装罢了。

迭代器的底层结构

由于迭代器的种类非常多，字符串、元组、列表等等，都有自己的迭代器，这里就不一一介绍了。所以我们就以列表的迭代器为例，看看迭代器在底层的结构是怎么样的。

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t it_index;
    //指向创建该迭代器的列表
    PyListObject *it_seq;
} listiterobject;
```

显然对于列表而言，迭代器就是在其之上进行了一层简单的封装，所谓元素迭代本质上还是基于索引，并且我们每迭代一次，索引就自增 1。一旦出现索引越界，就将it_seq设置为NULL，表示迭代器迭代完毕。

我们实际演示一下：

```
from ctypes import *

class PyObject(Structure):
    _fields_ = [
        ("ob_refcnt", c_ssize_t),
        ("ob_size", c_void_p)
    ]

class ListIterObject(PyObject):
    _fields_ = [
        ("it_index", c_ssize_t),
        ("it_seq", POINTER(PyObject))
    ]

it = iter([1, 2, 3])
it_obj = ListIterObject.from_address(id(it))

# 初始的时候，索引为0
print(it_obj.it_index)  # 0
# 进行迭代
next(it)
# 索引自增1，此时it_index等于1
print(it_obj.it_index)  # 1
# 再次迭代
next(it)
# 此时it_index等于2
print(it_obj.it_index)  # 2
# 再次迭代
next(it)
# 此时it_index等于3
print(it_obj.it_index)  # 3
```

当it_index为3的时候，如果再次迭代，那么底层发现it_index已超过最大索引，就知道迭代器已经迭代完毕了。然后会将it_seq设置为NULL，并抛出StopIteration。如果是for循环，那么会自动捕获此异常，然后停止循环。

所以这就是迭代器，真的没有想象中的那么神秘，甚至在知道它的实现原理之后，还觉得有点low。

就是将原始的数据包了一层，加了一个索引而已。所谓的迭代仍然是基于索引来做的，并且每迭代一次，索引自增1。当索引超出范围时，证明迭代完毕了，于是将it_seq设置为NULL，抛出StopIteration。

迭代器是怎么迭代元素的？

我们知道在迭代元素的时候，可以通过next内置函数，当然它本质上也是调用了对象的__next__方法。

```c
static PyObject *
builtin_next(PyObject *self, PyObject *const *args, Py_ssize_t nargs)
{
    PyObject *it, *res;

    // 同样接收一个参数或者两个参数
    // 因为调用next函数时，可以传入一个默认值
    // 表示当迭代器没有元素可以迭代的时候，会返回指定的默认值
    if (!_PyArg_CheckPositional("next", nargs, 1, 2))
        return NULL;

    it = args[0];
    //第一个参数必须是一个迭代器
    if (!PyIter_Check(it)) {
        //否则的话， 抛出TypeError
        //表示第一个参数传递的不是一个迭代器
        PyErr_Format(PyExc_TypeError,
            "'%.200s' object is not an iterator",
            it->ob_type->tp_name);
        return NULL;
    }
    //it->ob_type表示获取类型对象，也就是该迭代器的类型
    //可能是列表的迭代器、元组的迭代器、字符串的迭代器等等
    //具体是哪一种不重要，因为实现了多态
    //然后再获取tp_iternext成员，相当于__next__
    //拿到函数指针之后，传入迭代器进行调用
    res = (*it->ob_type->tp_iternext)(it);

    // 如果 res 不为 NULL， 那么证明迭代到值了， 直接返回
    if (res != NULL) {
        return res;
    } else if (nargs > 1) {
        //否则的话，说明 res == NULL，也就是有可能出错了
        //那么看nargs是否大于1， 如果大于1， 说明设置了默认值
        PyObject *def = args[1];
        // 如果出现异常
        if (PyErr_Occurred()) {
        // 那么就看该异常是不是迭代完毕时所产生的StopIteration异常
            if(!PyErr_ExceptionMatches(PyExc_StopIteration))
            // 如果不是，说明Python程序的逻辑有问题
            // 于是直接return NULL，结束执行
            // 然后在 Python 里面我们会看到打印到stderr中的异常信息
                return NULL;
            // 如果是 StopIteration，证明迭代完毕了
            // 但我们设置了默认值，那么就应该返回默认值
            // 而不应该抛出 StopIteration，于是将异常回溯栈给清空
            PyErr_Clear();
        }
        // 然后增加默认值的引用计数， 并返回
        Py_INCREF(def);
        return def;
    } else if (PyErr_Occurred()) {
        //走到这里说明 res == NULL，并且没有指定默认值
        //那么当发生异常时，将异常直接抛出
        return NULL;
    } else {
        // 都不是的话，直接抛出 StopIteration
        PyErr_SetNone(PyExc_StopIteration);
        return NULL;
    }
}
```

以上就是next函数的背后逻辑，实际上还是调用了迭代器的__next__方法。

```
lst = [1, 2, 3]
it = iter(lst)
# 然后迭代，等价于next(it)
print(type(it).__next__(it))  # 1
print(type(it).__next__(it))  # 2
print(type(it).__next__(it))  # 3
# 但是next可以指定默认值
# 如果不指定默认值，或者还是type(it).__next__(it)
# 那么就会报错，会抛出StopIteration
print(next(it, 666))  # 666
```

以上就是元素的迭代，但是我们知道内置函数next要更强大一些，因为它还可以指定一个默认值。当然在不指定默认值的情况下，**next(it)**和**type(it).__next__(it)**最终是殊途同归的。

我们仍以列表的迭代器为例，看看**__next__**的具体实现。但是要想找到具体实现，首先要找到它的类型对象。

```
//迭代器的类型对象
PyTypeObject PyListIter_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "list_iterator",                            /* tp_name */
    sizeof(listiterobject),                     /* tp_basicsize */
    0,                                          /* tp_itemsize */
    /* methods */
    (destructor)listiter_dealloc,               /* tp_dealloc */
    0,                                          /* tp_vectorcall_offset
*/
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_as_async */
    0,                                          /* tp_repr */
    0,                                          /* tp_as_number */
    0,                                          /* tp_as_sequence */
    0,                                          /* tp_as_mapping */
    0,                                          /* tp_hash */
    0,                                          /* tp_call */
    0,                                          /* tp_str */
    PyObject_GenericGetAttr,                    /* tp_getattro */
    0,                                          /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_GC,/* tp_flags */
    0,                                          /* tp_doc */
    (traverseproc)listiter_traverse,            /* tp_traverse */
    0,                                          /* tp_clear */
    0,                                          /* tp_richcompare */
    0,                                          /* tp_weaklistoffset */
    PyObject_SelfIter,                          /* tp_iter */
    (iternextfunc)listiter_next,                /* tp_iternext */
    listiter_methods,                           /* tp_methods */
    0,                                          /* tp_members */
};
```

我们看到它的tp_iternext成员指向了listiter_next，证明迭代的时候调用的是这个函数。

```
static PyObject *
listiter_next(listiterobject *it)
{
    PyListObject *seq;  //列表
```

```
    PyObject *item;     //元素
    assert(it != NULL);
    //拿到具体对应的列表
    seq = it->it_seq;
    //如果seq为NULL，证明迭代器已经迭代完毕
    //否则它不会为NULL
    if (seq == NULL)
        return NULL;
    assert(PyList_Check(seq));
    //如果索引小于列表的长度，证明尚未迭代完毕
    if (it->it_index < PyList_GET_SIZE(seq)) {
      //通过索引获取指定元素
        item = PyList_GET_ITEM(seq, it->it_index);
      //it_index自增1
        ++it->it_index;
      //增加引用计数后返回
        Py_INCREF(item);
        return item;
    }
    //否则的话，说明此次索引正好已经超出最大范围
    //意味着迭代完毕了，将it_seq设置为NULL
    //并减少它的引用计数，然后返回
    it->it_seq = NULL;
    Py_DECREF(seq);
    return NULL;
}
```

显然这和我们之前分析的是一样的，以上我们就以列表为例，考察了迭代器的实现原理和元素迭代的具体过程。当然其它对象也有自己的迭代器，有兴趣可以自己看一看。

小结

到此，我们再次体会到了Python的设计哲学，通过**PyObject ***和**ob_type**实现了多态。原因就在于它们接收的不是对象本身，而是对象的**PyObject ***泛型指针。

不管**变量obj**指向什么样的可迭代对象，都可以交给**iter函数**，会调用类型对象内部的**__iter__**，底层是**tp_iter**，得到对应的迭代器。不管**变量it**指向什么样的迭代器，都可以交给**next函数**进行迭代，会调用迭代器的类型对象的__next__，底层是**tp_iternext**，将值迭代出来。

至于__iter__和__next__本身，每个迭代器都会有，我们这里只以列表的迭代器为例。

所以这是不是实现了多态呢？

这就是Python的设计哲学，变量只是一个指针，传递变量的时候相当于传递指针（将指针拷贝一份），但是操作一个变量的时候会自动操作变量（指针）指向的内存。

比如：a = 123; b = a，相当于把 a 拷贝了一份给 b，但 a 是一个指针，所以此时 a 和 b 保存的地址是相同的，也就是指向了同一个对象。但 a+b 的时候则不是两个指针相加，而是将a、b指向的对象进行相加，也就是操作变量会自动操作变量指向的内存。

因此在Python中，说传递方式是值传递或者引用传递都是不准确的，应该是变量的**赋值传递**，对象的**引用传递**。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》43. PyCodeObject 对象与pyc文件

下一篇 ›
《源码探秘 CPython》41. 集合支持的操作是怎么实现的?

喜欢此内容的人还喜欢

消息队列新实现：Workflow msgqueue代码详解
架构鹦

A tour of gRPC：03 - proto序列化/反序列化
BUG侦探

模型构建器——迭代器1-1for循环
解题高手沛沛