《源码探秘 CPython》73. 自定义类对象的底层实现与metaclass（下）

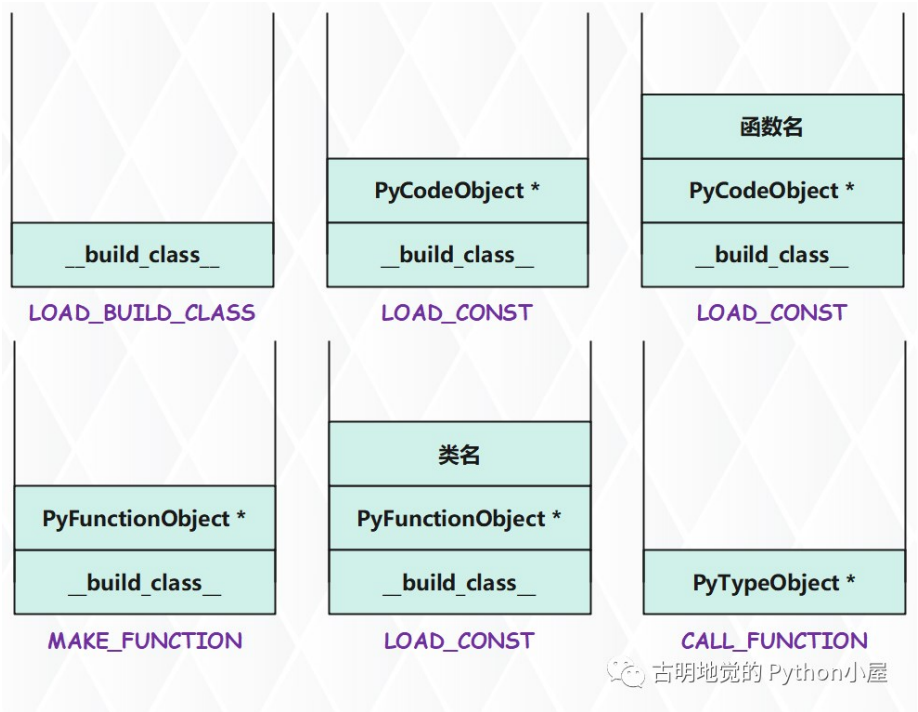
原创 古明地觉 古明地觉的编程教室 2022-04-21 08:30

收录于合集
#CPython
97个 >

回顾一下类是怎么创建的，首先会通过指令LOAD_BUILD_CLASS将内置函数__build_class__压入运行时栈。然后将A对应的PyCodeObject包装成一个PyFunctionObject；最后再调用__build_class__将PyFunctionObject变成PyTypeObject，也就是我们使用的类对象。

```
class A: pass
class B: pass
class C: pass
class D: pass
class E: pass
class F: pass

MyClass = __build_class__(lambda: None, "MyClass",
                          A, B, C, D, E, F)
print(MyClass)  # <class '__main__.MyClass'>

for cls in MyClass.__mro__:
    print(cls)
"""
<class '__main__.MyClass'>
<class '__main__.A'>
<class '__main__.B'>
<class '__main__.C'>
<class '__main__.D'>
<class '__main__.E'>
<class '__main__.F'>
<class 'object'>
"""
```

我们以运行时栈的变化，来描述一下上述过程：



那么接下来的重点就是 __build_class__，它是如何将一个函数变成类的，我们来看一下。内置函数的相关实现，位于 Python/bitinmodule.c 中。

```
static PyMethodDef builtin_methods[] = {
    {"__build_class__", (PyCFunction)(void(*)(void))builtin___build_class
__,
     METH_FASTCALL | METH_KEYWORDS, build_class_doc},
    //...
    //...
  }
```

__build_class__ 是在 Python 里面使用的函数，它在底层对应 builtin___build_class__。

```
static PyObject *
builtin___build_class__(PyObject *self, PyObject *const *args, Py_ssize
_t nargs,
                        PyObject *kwnames)
{
    PyObject *func, *name, *bases, *mkw, *meta, *winner, *prep, *ns, *o
rig_bases;
    PyObject *cls = NULL, *cell = NULL;
    int isclass = 0;   /* initialize to prevent gcc warning */

    //我们说了底层调用的是builtin___build_class__
    //class A: 会被翻译成builtin.__build_class__(PyFunctionObject, class
 name)
    //所以这个函数至少需要两个参数
    if (nargs < 2) {
        //参数不足，报错，还记的这个报错信息吗？之前测试过的
        PyErr_SetString(PyExc_TypeError,
                        "__build_class__: not enough arguments");
        return NULL;
    }
    //类对应的PyFunctionObject
    func = args[0];   /* Better be callable */
    if (!PyFunction_Check(func)) {
        //如果不是PyFunctionObject，报错，这个信息有印象吗？
        PyErr_SetString(PyExc_TypeError,
                        "__build_class__: func must be a function");
        return NULL;
    }

    //类对应的名字，__build_class__的时候，类肯定要有名字
    name = args[1];
    if (!PyUnicode_Check(name)) {
        //必须要是一个PyUnicodeObject，否则报错
        PyErr_SetString(PyExc_TypeError,
                        "__build_class__: name is not a string");
        return NULL;
    }

    //args[0]表示 PyFunctionObject*
    //args[1]表示 class name
    //args + 2 开始是继承的基类，基类的个数显然是 nargs - 2
    //所以这里是拿到所有的基类
    orig_bases = _PyTuple_FromArray(args + 2, nargs - 2);
    if (orig_bases == NULL)
        return NULL;

    //这个update_bases比较有趣，我们一会单独说
    bases = update_bases(orig_bases, args + 2, nargs - 2);
    if (bases == NULL) {
        Py_DECREF(orig_bases);
        return NULL;
    }

    //获取 metaclass
```

```
    if (kwnames == NULL) {
        meta = NULL;
        mkw = NULL;
    }
    else {
        mkw = _PyStack_AsDict(args + nargs, kwnames);
        if (mkw == NULL) {
            Py_DECREF(bases);
            return NULL;
        }

        meta = _PyDict_GetItemIdWithError(mkw, &PyId_metaclass);
        if (meta != NULL) {
            Py_INCREF(meta);
            if (_PyDict_DelItemId(mkw, &PyId_metaclass) < 0) {
                Py_DECREF(meta);
                Py_DECREF(mkw);
                Py_DECREF(bases);
                return NULL;
            }
            /* metaclass is explicitly given, check if it's indeed a cl
ass */
            isclass = PyType_Check(meta);
        }
        else if (PyErr_Occurred()) {
            Py_DECREF(mkw);
            Py_DECREF(bases);
            return NULL;
        }
    }
    //如果meta为NULL，这意味着用户没有指定metaclass
    if (meta == NULL) {
        //然后尝试获取基类，如果没有基类
        if (PyTuple_GET_SIZE(bases) == 0) {
            //指定 metaclass 为 type
            meta = (PyObject *) (&PyType_Type);
        }
        //否则获取第一个继承的基类的 metaclass
        else {
            //拿到第一个基类
            PyObject *base0 = PyTuple_GET_ITEM(bases, 0);
            //拿到第一个基类的__class__
            meta = (PyObject *) (base0->ob_type);
        }
        //meta也是一个类
        Py_INCREF(meta);
        isclass = 1;
    }

    //如果设置了元类，那么isclass为1，if 为真
    if (isclass) {
        //选择出了元类，下面这一步就要解决元类冲突
        //假设有两个继承type的元类 MyType1 和 MyType2
        //然后 Base1 的元类是 MyType1、Base2 的元类是 MyType2
        //那么如果class A(Base1, Base2)的话，就会报错
        //因为在Python中有一个要求，假设class A(Base1, Base2, ..., BaseN)
        //Base1的元类叫MyType1、...、BaseN的元类叫MyTypeN
        //那么必须满足：
        /*
        MyType1是MyType2的子类或者父类；
        MyType1是MyType3的子类或者父类；
        MyType1是MyType4的子类或者父类；
        ....
        MyType1是MyTypeN的子类或者父类；
```

```
        */
        //而之所以存在这一限制，原因就是为了避免属性冲突
        winner = (PyObject *)_PyType_CalculateMetaclass((PyTypeObject *
)meta,
                                                        bases);
        if (winner == NULL) {
            Py_DECREF(meta);
            Py_XDECREF(mkw);
            Py_DECREF(bases);
            return NULL;
        }
        if (winner != meta) {
            Py_DECREF(meta);
            meta = winner;
            Py_INCREF(meta);
        }
    }

    //寻找__prepare__
    if (_PyObject_LookupAttrId(meta, &PyId___prepare__, &prep) < 0) {
        ns = NULL;
    }
    //这个__prepare__必须返回一个mapping
    //如果返回None，那么等价于返回一个空字典
    else if (prep == NULL) {
        ns = PyDict_New();
    }
    else {
        //否则将字典返回
        PyObject *pargs[2] = {name, bases};
        //我们看到这里涉及了一个函数调用，这个函数应该有印象吧
        ns = _PyObject_FastCallDict(prep, pargs, 2, mkw);
        Py_DECREF(prep);
    }
    if (ns == NULL) {
        Py_DECREF(meta);
        Py_XDECREF(mkw);
        Py_DECREF(bases);
        return NULL;
    }
    if (!PyMapping_Check(ns)) {
        //如果返回的不是一个字典，那么报错
        //这个错误信息我们也见过了
        PyErr_Format(PyExc_TypeError,
                     "%.200s.__prepare__() must return a mapping, not %.
200s",
                     isclass ? ((PyTypeObject *)meta)->tp_name : "<metac
lass>",
                     Py_TYPE(ns)->tp_name);
        goto error;
    }
    //......
}
```

可以看到，一个简单的类定义，虚拟机究竟做了多少事情啊，不过显然这还没完。

自定义类对象的元信息分为两部分，分别是动态元信息和静态元信息。虚拟机在获得了关于属性表(动态元信息)之后，就知道了所有的属性。

但是对于自定义类对象的类型是什么，应该如何创建、要分配多少内存，却没有任何的头绪，因为这部分隐藏在 metaclass 里面。

而在 builtin___build_class__中，metaclass正是关于自定义类对象的另一部分元信息，我们称之

为静态元信息。在静态元信息中，隐藏着所有的类对象应该如何创建的信息，注意：是所有的类对象。

从源码中我们可以看到，如果用户指定了metaclass，那么会选择指定的metaclass；如果没有指定，那么会使用第一个继承的基类的metaclass作为该class的metaclass。

对于实例对象，所有的元信息都存储在对应的类对象中。但是对于类对象来说，其元信息的静态元信息存储在对应的元类中，动态元信息则存储在本身的local名字空间中。

可为什么这么做呢？为什么对于类对象来说，其元信息要游离成两部分呢？都存在metaclass里面不香吗？这是因为用户在 .py 文件中可以定义不同的 class，这个元信息必须、且只能是动态的，所以它不适合保存在 metaclass中。而存储在metaclass里面的，一定是诸如类对象的创建策略等这些所有class都会共用的元信息。

注意：我们说元信息游离成两部分指的是自定义类对象，内置类对象的元信息也是都存储 metaclass 中的。

因为内置类对象都是静态提供的，它们都具备相同的接口集合(底层都是PyTypeObject结构体实例)，支持什么操作一开始就定义好了。只不过有的可以用，有的不能用。比如PyLongObject可以使用nb_add，但是PyDictObject不能。而PyDictObject可以使用mp_subscript，但是PyLongObject不可以。

尽管如此，但这不影响它们的所有元信息都存储在类型对象中。而用户自定义的类对象，接口是动态的，不可能在metaclass中静态指定。

然后再来说一说源码中的 update_bases，它比较有意思。

```
class Foo:
    name = "古明地觉"

class Bar:

    def __mro_entries__(self, bases):
        return (Foo, tuple,)

class MyClass(Bar()):
    pass

print(
    MyClass("123"),
    MyClass.name
)  # ('1', '2', '3') 古明地觉
```

我们在继承的时候，都是继承一个类，但是这里的 MyClass 居然继承了一个实例对象。相信结果你已经猜出来了，如果继承的是实例，那么会去调用实例的 `__mro_entries__`。因此 MyClass 继承的其实是 Foo、tuple，并且`__mro_entries__`必须返回一个元组，否则报错。

另外，如果一个类继承了一个拥有`__mro_entries__`的实例，那么该类会多出一个属性叫 __orig_bases__。我们回顾一下 type_new 里面的几行关键代码：

```
    orig_bases = _PyTuple_FromArray(args + 2, nargs - 2);
    if (orig_bases == NULL)
        return NULL;
    bases = update_bases(orig_bases, args + 2, nargs - 2);
```

里面有两个变量 orig_bases 和 bases。我们知道 Python 的类都有一个 __bases__属性，对应这里的 bases；但鲜为人知的是，它还有一个属性叫 __orig_bases__，对应这里的 orig_bases。

```
class Foo:
    name = "古明地觉"

```

```
class Bar:

    def __mro_entries__(self, bases):
        return (Foo, tuple,)

class MyClass(Bar()):
    pass

print(MyClass.__orig_bases__)
print(MyClass.__bases__)
"""
(<__main__.Bar object at 0x000001B4D50BD040>,)
(<class '__main__.Foo'>, <class 'tuple'>)
"""
```

__bases__ 表示继承的基类，但很明显我们这里继承的不是 Foo 和 tuple，而是 Bar()，所以 __bases__ 是 update_bases 函数基于 __orig_bases__ 处理得到的。

__orig_bases__ 表示继承的对象，该对象可以是一个类对象，也可以是一个实例对象；如果是实例对象，那么在 update_bases 函数中，会调用它的`__mro_entries__`方法，该方法返回一个包含类对象的元组，然后设置到 __bases__ 中。

以上就是函数 update_bases 的作用，但有两点需要注意。

- 1）只有继承了拥有 __mro_entries__ 方法的实例的类，才有 __orig_bases__ 属性；
- 2）这样的类，在 Python 里面不能手动调用 type 来创建；

我们来解释一下，首先是第一点：

```
print(hasattr(MyClass, "__orig_bases__"))
print(hasattr(Foo, "__orig_bases__"))
print(hasattr(Bar, "__orig_bases__"))
"""
True
False
False
"""
```

我们看到只有 MyClass 有 __orig_bases__ 属性，因为它继承了拥有 __mro_entries__ 方法的实例，而 Foo 和 Bar 则没有。

然后是第二点，这样的类不可以手动调用 type 来创建。

```
class Foo:
    name = "古明地觉"

class Bar:

    def __mro_entries__(self, bases):
        return (Foo, tuple,)

try:
    MyClass = type("MyClass", (Bar(),), {})
except TypeError as e:
    print(e)
# type() doesn't support MRO entry resolution; use types.new_class()

# 所以这样的类请通过 class 关键字创建
# 如果必须手动创建的话，那么可以使用 types.new_class
import types
MyClass = types.new_class("MyClass", (Bar(),), {})
print(MyClass.name)  # 古明地觉
print(MyClass("123"))  # ('1', '2', '3')
```

以上就是 update_bases 函数所干的事情，但是问题来了，如果继承的实例对象没有__mro_entries__方法怎么办?

```
class Base:

    def __init__(self, *args):
        if len(args) == 0:
            return
        elif len(args) == 3:
            name, bases, attrs = args
        else:
            raise ValueError("args 的长度必须是 0 或 3")
        self.name = name
        self.bases = bases
        self.attrs = attrs

base = Base()

class MyClass(base):
    pass

print(type(MyClass) is Base)  # True
print(MyClass.name)  # MyClass
print(MyClass.bases == (base,))  # True
print(
    MyClass.attrs
)  # {'__module__': '__main__', '__qualname__': 'MyClass'}
```

显然 MyClass 继承的是 Base 的实例对象，并且 Base 里面也没有定义__mro_entries__，那么虚拟机就不会再使用 type 来创建 MyClass 了。而是会使用 Base 来创建，所以得到的 MyClass 就是一个 Base 的实例对象。

现在算是彻底理解 update_bases 的作用了，因为不能保证继承的都是类，所以还需要进行检测，如果不是类，那么就执行上面的逻辑。但是说实话，这个特性几乎不用，因为既然要继承，那么就应该继承类。虽然通过__mro_entries__可以整一些花活，甚至也能简化逻辑，但最好还是不要用，因为它会让代码变得难以理解。

builtin___build_class__ 的逻辑我们上面省略了一部分，至于省略部分的逻辑也很简单，既然元类以及相关参数都准备好了，那么接下来就是对类进行创建了。

我们知道调用一个对象，本质上会执行其类对象的 __call__。所以调用类对象创建实例对象，会执行 **type.__call__(cls, ...)**；调用元类创建类对象，会执行 **type.__call__(type, ...)**，因为元类的类对象还是它本身。

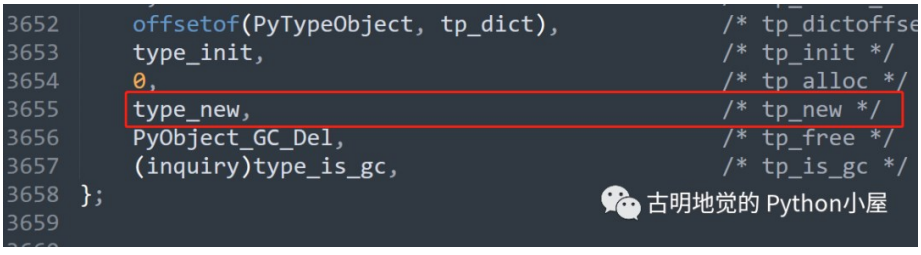
所以不管调用的是元类、还是类对象，都会执行元类的 __call__，在底层对应 &PyType_Type 的 tp_call 成员，它指向了 type_call 函数。

```
//Objects/typeobject.c
static PyObject *
type_call(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    PyObject *obj;
    //tp_new 负责创建实例，所以它不能为空
    if (type->tp_new == NULL) {
        PyErr_Format(PyExc_TypeError,
                     "cannot create '%.100s' instances",
                     type->tp_name);
        return NULL;
    }
    //调用tp_new为实例申请内存
```

```
    obj = type->tp_new(type, args, kwds);
    obj = _Py_CheckFunctionResult((PyObject*)type, obj, NULL);
    if (obj == NULL)
        return NULL;

    //如果调用的是&PyType_Type，并且只接收了一个位置参数
    //那么显然是查看对象类型，执行完__new__之后直接返回
    if (type == &PyType_Type &&
        PyTuple_Check(args) && PyTuple_GET_SIZE(args) == 1 &&
        (kwds == NULL ||
         (PyDict_Check(kwds) && PyDict_GET_SIZE(kwds) == 0)))
        return obj;

    //记得我们之前说过，__new__里面一定要返回类的实例对象
    //否则是不会执行__init__函数的，从这里我们也看到了
    //如果obj的类型不是对应的类、或者其子类，那么直接返回
    if (!PyType_IsSubtype(Py_TYPE(obj), type))
        return obj;

    //然后获取obj的类型
    type = Py_TYPE(obj);
    //如果内部存在__init__函数，那么执行
    if (type->tp_init != NULL) {
        int res = type->tp_init(obj, args, kwds);
        if (res < 0) {
            assert(PyErr_Occurred());
            Py_DECREF(obj);
            obj = NULL;
        }
        else {
            assert(!PyErr_Occurred());
        }
    }
    //执行完构造函数之后，再将对象返回
    //返回的 obj 可以是类对象、也可以是实例对象
    return obj;
}
```

type_call 里面的逻辑非常简单，就是先调用对象的 tp_new 创建实例，然后执行 tp_init（如果有）。至于返回的是类对象还是实例对象，则取决于 type_call 的第一个参数，如果第一个参数是元类，那么返回的就是类对象，否则是实例对象。

因此创建的核心逻辑就隐藏在对象的 tp_new 中，不同对象的tp_new指向的函数不同。但对于创建类对象而言，显然执行的是 &PyType_Type 的 tp_new，它指向的是 type_new 函数。



这个type_new是我们创建自定义类对象的第一案发现场，我们看一下它的源码，位于 typeobject.c 中。这个函数的代码比较长，我们会有删减，像那些检测的代码我们就省略掉了。

```
static PyObject *
type_new(PyTypeObject *metatype, PyObject *args, PyObject *kwds)
{
    //都是类的一些动态元信息
    PyObject *name, *bases = NULL, *orig_dict, *dict = NULL;
    PyObject *qualname, *slots = NULL, *tmp, *newslots, *cell;
```

```
    PyTypeObject *type = NULL, *base, *tmptype, *winner;
    PyHeapTypeObject *et;
    PyMemberDef *mp;
    Py_ssize_t i, nbases, nslots, slotoffset, name_size;
    int j, may_add_dict, may_add_weak, add_dict, add_weak;
    _Py_IDENTIFIER(__qualname__);
    _Py_IDENTIFIER(__slots__);
    _Py_IDENTIFIER(__classcell__);

    //如果metaclass是type的话
    if (metatype == &PyType_Type) {
        //获取位置参数和关键字参数个数
        const Py_ssize_t nargs = PyTuple_GET_SIZE(args);
        const Py_ssize_t nkwds = kwds == NULL ? 0 : PyDict_GET_SIZE(kwd
s);

        //位置参数为1，关键字参数为0，你想到了什么
        //type(xxx)，是不是这个呀
        if (nargs == 1 && nkwds == 0) {
            PyObject *x = PyTuple_GET_ITEM(args, 0);
            Py_INCREF(Py_TYPE(x));
            //这显然是初学Python时就知道的
            //查看一个变量指向的对象的类型，获取类型之后直接返回
            return (PyObject *) Py_TYPE(x);
        }

        //如果上面的if不满足，会走这里
        //表示现在不再是查看类型了，而是创建类
        //那么要求位置参数必须是3个，否则报错
        if (nargs != 3) {
            PyErr_SetString(PyExc_TypeError,
                            "type() takes 1 or 3 arguments");
            return NULL;
        }
    }

    /* Check arguments: (name, bases, dict) */
    //现在显然是确定参数类型，因为传递的三个参数是有类型要求的
    //必须是PyUnicodeObject、PyTupleObject、PyDictObject
    if (!PyArg_ParseTuple(args, "UO!O!:type.__new__", &name, &PyTuple_T
ype,
                          &bases, &PyDict_Type, &orig_dict))
    /*
    TypeError: type.__new__() argument 1 must be str, not xxx
    TypeError: type.__new__() argument 2 must be tuple, not xxx
    TypeError: type.__new__() argument 3 must be dict, not xxx
    */
        return NULL;
    //无论是使用 class 关键字创建类，还是使用 type 创建类
    //底层都是执行的 type_new

    //处理基类
    nbases = PyTuple_GET_SIZE(bases);
    if (nbases == 0) {
        //如果没有继承基类，那么会默认继承object
        //__base__ 设置为 object
        base = &PyBaseObject_Type;
        //__bases__ 设置为 (object,)
        bases = PyTuple_Pack(1, base);
        if (bases == NULL)
            return NULL;
        nbases = 1;
    }
    else {
```

```
71          _Py_IDENTIFIER(__mro_entries__);
72          //如果继承了基类，那么循环遍历bases
73          for (i = 0; i < nbases; i++) {
74              //拿到每一个基类
75              tmp = PyTuple_GET_ITEM(bases, i);
76              //如果类型为&PyType_Type，进行下一次循环
77              if (PyType_Check(tmp)) {
78                  continue;
79              }
80              //如果类型不是&PyType_Type，说明继承的不是类
81              //于是寻找__mro_entries__
82              if (_PyObject_LookupAttrId(tmp, &PyId___mro_entries__, &tmp)
83   < 0) {
84                  return NULL;
85              }
86              if (tmp != NULL) {
87                  PyErr_SetString(PyExc_TypeError,
88                                  "type() doesn't support MRO entry resolut
89  ion; "
90                                  "use types.new_class()");
91                  Py_DECREF(tmp);
92                  return NULL;
93              }
94          }
95          //寻找父类的metaclass
96          //采用之前说的解决元类冲突时所采取的策略
97          winner = _PyType_CalculateMetaclass(metatype, bases);
98
99          //每个类都有 __base__ 和 __bases__
100         //__bases__ 表示直接继承的所有类，__base__ 是继承的第一个类
101         //那么下面这行代码是做什么呢？直接 base = bases[0]就好了
102         //其实这个 best_base 所做的事情没有这么简单
103         //它还负责检测基类之间是否发生了冲突
104         base = best_base(bases);
105         if (base == NULL) {
106             return NULL;
107         }
108
109         Py_INCREF(bases);
110     }
111
112     dict = PyDict_Copy(orig_dict);
113     if (dict == NULL)
114         goto error;
115
116     //处理用户定义了__slots__属性的逻辑
117     //一旦定义了__slots__，那么类的实例对象就没有属性字典了
118     slots = _PyDict_GetItemIdWithError(dict, &PyId___slots__);
119     nslots = 0;
120     add_dict = 0;
121     add_weak = 0;
122     may_add_dict = base->tp_dictoffset == 0;
123     may_add_weak = base->tp_weaklistoffset == 0 && base->tp_itemsize ==
124 0;
125     if (slots == NULL) {
126         //.....
127     }
128     else {
129         //.....
130     }
131
132     //为自定义类对象申请内存
133     type = (PyTypeObject *)metatype->tp_alloc(metatype, nslots);
134     if (type == NULL)
```

```
135         goto error;
136 
137     et = (PyHeapTypeObject *)type;
138     Py_INCREF(name);
139     et->ht_name = name;
140     et->ht_slots = slots;
141     slots = NULL;
142 
143     /* 初始化tp_flags */
144     type->tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HEAPTYPE |
145         Py_TPFLAGS_BASETYPE;
146     if (base->tp_flags & Py_TPFLAGS_HAVE_GC)
147         type->tp_flags |= Py_TPFLAGS_HAVE_GC;
148 
149     //设置PyTypeObject中的各个域
150     type->tp_as_async = &et->as_async;
151     type->tp_as_number = &et->as_number;
152     type->tp_as_sequence = &et->as_sequence;
153     type->tp_as_mapping = &et->as_mapping;
154     type->tp_as_buffer = &et->as_buffer;
155     type->tp_name = PyUnicode_AsUTF8AndSize(name, &name_size);
156     if (!type->tp_name)
157         goto error;
158     if (strlen(type->tp_name) != (size_t)name_size) {
159         PyErr_SetString(PyExc_ValueError,
160                         "type name must not contain null characters");
161         goto error;
162     }
163 
164     /* 设置基类和基类列表 */
165     type->tp_bases = bases;
166     bases = NULL;
167     Py_INCREF(base);
168     type->tp_base = base;
169 
170     /* 设置属性字典 */
171     Py_INCREF(dict);
172     type->tp_dict = dict;
173 
174     //设置__module__
175     if (_PyDict_GetItemIdWithError(dict, &PyId___module__) == NULL) {
176         //......
177     }
178 
179     //设置__qualname__，即"全限定名"
180     qualname = _PyDict_GetItemIdWithError(dict, &PyId___qualname__);
181     //......
182 
183     //如果自定义的class中重写了__new__方法
184     //将__new__对应的函数改造为静态方法，并替换掉默认的__new__
185     tmp = _PyDict_GetItemIdWithError(dict, &PyId___new__);
186     if (tmp != NULL && PyFunction_Check(tmp)) {
187         tmp = PyStaticMethod_New(tmp);
188         if (tmp == NULL)
189             goto error;
190         if (_PyDict_SetItemId(dict, &PyId___new__, tmp) < 0) {
191             Py_DECREF(tmp);
192             goto error;
193         }
194         Py_DECREF(tmp);
195     }
196     else if (tmp == NULL && PyErr_Occurred()) {
197         goto error;
198     }
```

```
199
200     //获取__init_subclass__，如果子类继承了父类
201     //那么会触发父类的__init_subclass__
202     tmp = _PyDict_GetItemIdWithError(dict, &PyId___init_subclass__);
203     if (tmp != NULL && PyFunction_Check(tmp)) {
204         tmp = PyClassMethod_New(tmp);
205         if (tmp == NULL)
206             goto error;
207         if (_PyDict_SetItemId(dict, &PyId___init_subclass__, tmp) < 0) {
208             Py_DECREF(tmp);
209             goto error;
210         }
211         Py_DECREF(tmp);
212     }
213     else if (tmp == NULL && PyErr_Occurred()) {
214         goto error;
215     }
216
217     //设置__class_getitem__，这个是什么？类似于__getitem__
218     //__class_getitem__支持通过 类["xxx"] 的方式访问
219     tmp = _PyDict_GetItemIdWithError(dict, &PyId___class_getitem__);
220     //......
221
222     //为自定义类对象对应的实例对象设置内存大小信息
223     type->tp_basicsize = slotoffset;
224     type->tp_itemsize = base->tp_itemsize;
225     type->tp_members = PyHeapType_GET_MEMBERS(et);
226     //......
227
228     //调用PyType_Ready对自定义类对象进行初始化
229     if (PyType_Ready(type) < 0)
230         goto error;
231
232     /* Put the proper slots in place */
233     fixup_slot_dispatchers(type);
234
235     if (type->tp_dictoffset) {
236         et->ht_cached_keys = _PyDict_NewKeysForClass();
237     }
238
239     if (set_names(type) < 0)
240         goto error;
241
242     if (init_subclass(type, kwds) < 0)
243         goto error;
244
245     Py_DECREF(dict);
246     return (PyObject *)type;
247
248 error:
249     Py_XDECREF(dict);
    Py_XDECREF(bases);
    Py_XDECREF(slots);
    Py_XDECREF(type);
    return NULL;
}
```

虚拟机首先会解析出类名、基类列表和属性字典，然后根据基类列表及传入的metaclass确定最佳的metaclass和base。

随后，虚拟机会调用 metatype->tp_alloc 尝试为要创建的类对象分配内存。这里需要注意的是，在&PyType_Type中，我们会发现tp_alloc是一个NULL，这显然不正常。但是不要忘记，虚拟机会通过PyType_Ready对所有的类对象进行初始化。

在这个初始化过程中，有一项动作就是从基类继承各种操作。由于type.__bases__中的第一个基类是object，所以type会继承object的tp_alloc操作，即 PyType_GenericAlloc 。

对于所有继承object的类对象来说， PyType_GenericAlloc 将申请**metatype->tp_basicsize + metatype->tp_itemsize**大小的内存空间。

从PyType_Type的定义中我们看到，这个大小实际就是 **sizeof(PyHeapTypeObject) + sizeof(PyMemerDef)** 。

因此在这里应该就明白了PyHeapTypeObject这个老铁到底是干嘛用的了，之前因为偏移量的问题，折腾了不少功夫，甚至让人觉得这有啥用啊，但是现在意识到了，这个老铁是为自定义类对象准备的。

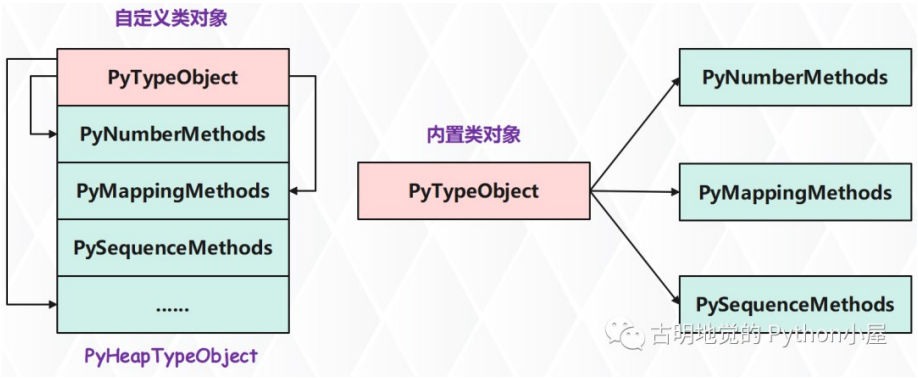
接下来就是设置自定义类对象的各个域，其中包括了在tp_dict上设置属性字典，也就是__dict__。另外注意的是，这里还计算了类对象对应的实例对象所需要的内存大小信息，换言之，自定义类在创建一个实例对象时，需要为这个实例对象申请多大的内存空间呢？

对于任意继承object的自定义类对象来说，这个大小为**PyBaseObject_Type->tp_basicsize + 16**，其中的16是2 * sizeof(PyObject *)。

而之所以后面要跟着两个PyObject *的空间，是因为这些空间的地址被设置给了 **tp_dictoffset**，以及 **tp_weaklistoffset**。这一点将在介绍实例对象时进行解析，它是和实例对象的属性字典密切相关的。

最后，虚拟机还会调用PyType_Ready对自定义类对象进行和内置类对象一样的初始化动作，到此自定义类对象才算正式创建完毕。因此内置类对象是底层静态定义好的，启动之后再调用 PyType_Ready 完善一下即可；但自定义类对象则不同，它需要运行时动态创建，这是一个复杂的过程。但最后，两者都会调用 PyType_Ready。

那么内置类对象和自定义对象在内存布局上面有什么区别呢？毕竟都是类对象。



本质上，无论是自定义类对象还是内置类对象，在虚拟机内部，都可以用一个PyTypeObject来表示。

但不同的是，内置类对象对应的PyTypeObject以及与其关联的PyNumberMethods等操作簇的内存位置都是在编译时确定的，它们在内存中的位置是分离的。而自定义类对象对应的PyTypeObject和PyNumberMethods等操作簇的内存位置是连续的，必须在运行时动态分配内存。

另外，自定义类对象对应的 PyTypeObject 和相关操作簇组合起来，被称为PyHeapTypeObject。

```
typedef struct _heaptypeobject {
    PyTypeObject ht_type;
    PyAsyncMethods as_async;
    PyNumberMethods as_number;
    PyMappingMethods as_mapping;
    PySequenceMethods as_sequence;
    PyBufferProcs as_buffer;
    PyObject *ht_name, *ht_slots, *ht_qualname;
```

```
    struct _dictkeysobject *ht_cached_keys;
} PyHeapTypeObject;
```

内置类对象有哪些操作是静态定义好的，所以相关操作是分离的；但自定义类对象的相关操作簇必须紧随其后，且顺序也有讲究，只有这样才能通过偏移量 offset 准确找到指定的操作。

现在我们也对Python的可调用(callable)这个概念有一个感性认识了，可调用这个概念是一个相当通用的概念，不拘泥于对象、大小，只要类型对象定义了tp_call操作，就能进行调用操作。我们已经看到，调用metaclass得到类对象，调用类对象得到实例对象。如果类对象也定义了tp_call，那么还可以继续对实例对象进行调用。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》74. 彻底搞懂描述符

下一篇 ›
《源码探秘 CPython》72. 自定义类对象的底层实现与 metaclass（上）