《源码探秘 CPython》70. 类对象属性字典的填充

原创 古明地觉 古明地觉的编程教室 2022-04-18 08:30

楔子

在给类对象设置完基类、以及类型信息之后，就开始填充属性字典了，这是一个非常复杂的过程。

```
int
PyType_Ready(PyTypeObject *type)
{
    PyObject *dict, *bases;
    PyTypeObject *base;
    Py_ssize_t i, n;
    //......
    //......
    //初始化tp_dict
    dict = type->tp_dict;
    if (dict == NULL) {
        dict = PyDict_New();
        if (dict == NULL)
            goto error;
        type->tp_dict = dict;
    }

    //将与type相关的操作加入到tp_dict中
    //注意: 这里的type是PyType_Ready的参数中的type
    //它可以是Python的<class 'type'>、也可以是<class 'int'>
    if (add_operators(type) < 0)
        goto error;
    if (type->tp_methods != NULL) {
        if (add_methods(type, type->tp_methods) < 0)
            goto error;
    }
    if (type->tp_members != NULL) {
        if (add_members(type, type->tp_members) < 0)
            goto error;
    }
    if (type->tp_getset != NULL) {
        if (add_getset(type, type->tp_getset) < 0)
            goto error;
    }
    //.......
}
```

在这个阶段，完成了将魔法函数的函数名和函数体加入tp_dict的过程，里面的 add_operators 、add_methods 、 add_members 、 add_getset 都是完成填充 tp_dict 的动作。

那么这时候一个问题就出现了，以整形的 __sub__ 为例，我们知道它会对应底层的 long_sub，可虚拟机是如何知道 __sub__ 和 long_sub 之间存在关联的呢？其实这种关联显然是一开始就已经定好了的，存放在一个名为 slotdefs 的数组中。

slot 与操作排序

在进入填充tp_dict的复杂操作之前，我们先来看一个概念：slot。slot 可以视为表示 PyTypeObject 中定义的操作，一个魔法函数对应一个 slot。比如 __add__、__sub__ 等等，都会对应一个 slot。

但是 slot 又不仅仅包含一个函数指针，它还包含一些其它信息，我们看看它的结构。在Python内部，slot 是通过 slotdef 这个结构体来实现的。

```
//typeobject.c
typedef struct wrapperbase slotdef;

//descrobject.h
struct wrapperbase {
    const char *name;
    int offset;
    void *function;
    wrapperfunc wrapper;
    const char *doc;
    int flags;
    PyObject *name_strobj;
};
//从定义上看，我们发现slot不是一个PyObject
```

在一个 slot 中，就存储着PyTypeObject的一种操作对应的各种信息，比如：整数(PyLongObject)支持哪些行为，就看类型对象int(PyLong_Type)定义了哪些操作，而PyTypeObject对象中的一个操作就会有一个slot与之对应。

slot里面的 name 就是操作对应的名称，比如字符串 "__sub__"，offset 则是操作的函数地址在**XXX**中的偏移量（这个**XXX**是什么一会说），而function则指向一种名为slot function的函数。

Python提供了多个宏来定义一个 slot，其中最基本的是 TPSLOT 和 ETSLOT。

```
//typeobject.c
#define TPSLOT(NAME, SLOT, FUNCTION, WRAPPER, DOC) \
    {NAME, offsetof(PyTypeObject, SLOT), (void *)(FUNCTION), WRAPPER, \
     PyDoc_STR(DOC)}

#define ETSLOT(NAME, SLOT, FUNCTION, WRAPPER, DOC) \
{NAME, offsetof(PyHeapTypeObject, SLOT), (void *)(FUNCTION), WRAPPER, \
     PyDoc_STR(DOC)}
```


我们看到里面又出现了PyHeapTypeObject，它又是什么？先来看看它的定义：

```
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;
```

这个PyHeapTypeObject是为自定义类对象准备的，它的第一个成员就是 PyTypeObject，至于其它的则是操作簇。至于为什么要有这么一个对象，原因是自定义类对象和相关的操作簇在内存中是连续的，必须在运行时动态分配内存，所以它是为自定义类准备的（具体细节后续剖析）。

- 对于内置类对象而言，offset 表示操作的函数地址在PyTypeObject中的偏移量；
- 对于自定义类对象而言，offset 表示操作的函数地址在PyHeapTypeObject中的偏移量；

于是这里就产生了一个问题，假设我们定义了一个类继承自 int，根据继承关系，显然自定义的类是具有PyNumberMethods这个操作簇的，它可以使用 __add__、__sub__ 之类的魔法函数。

但操作簇是定义在 PyTypeObject 里面的，而此时的 offset 却是基于 PyHeapTypeObject 得到的偏移量，那么通过这个 offset 显然无法准确找到操作簇里面的函数指针，比如 long_add、long_sub 等等。

那我们要这个 offset 还有何用呢？答案非常诡异，这个 offset 是用来对操作进行排序的。排序？我整个人都不好了。

不过在理解为什么需要对操作进行排序之前，需要先看看底层预先定义的 slot 集合 slotdefs。

```
static slotdef slotdefs[] = {
    //里面定义的操作非常多，这里截取一部分
    //我们看到不同的操作名，可以对应同一种操作
    //比如 __add__、__radd__ 都对应 nb_add
    //这个 nb_add 在 PyLong_Type 就是 long_add，表示 +
    BINSLOT("__add__", nb_add, slot_nb_add,
           "+"),
    RBINSLOT("__radd__", nb_add, slot_nb_add,
           "+"),
    BINSLOT("__sub__", nb_subtract, slot_nb_subtract,
           "-"),
    RBINSLOT("__rsub__", nb_subtract, slot_nb_subtract,
           "-"),
    BINSLOT("__mul__", nb_multiply, slot_nb_multiply,
           "*"),
    RBINSLOT("__rmul__", nb_multiply, slot_nb_multiply,
           "*"),
    //......
    //相同操作名也可以对应不同操作
    //比如 __getitem__ 对应 mp_subscript、sq_item
    MPSLOT("__getitem__", mp_subscript, slot_mp_subscript,
           wrap_binaryfunc,
           "__getitem__($self, key, /)\n--\n\nReturn self[key]."),
    SQSLOT("__getitem__", sq_item, slot_sq_item, wrap_sq_item,
           "__getitem__($self, key, /)\n--\n\nReturn self[key]."),
    //......
};
```

在 slotdefs 中可以发现，操作名和操作并不是一一对应的，存在多个操作对应同一个操作名、或者多个操作名对应同一个操作的情况。那么在填充tp_dict时，就会出现问题。

比如对于 __getitem__，在 tp_dict 中与其对应的是 mp_subscript 还是 sq_item 呢？这两者都是通过 [] 进行操作的，比如字典根据 key 获取 value、列表基于索引获取元素，对应的都是 __getitem__。

为了解决这个问题，就需要利用 slot 中的 offset 信息对 slot（也就是操作）进行排序。回顾一下前面列出的 PyHeapTypeObject 的定义，与一般的 struct 定义不同，它的各个成员的顺序是非常关键的，在顺序中隐含着操作优先级的问题。

在PyHeapTypeObject中，PyMappingMethods 的位置在 PySequenceMethods 之前，mp_subscript是 PyMappingMethods 中的一个函数指针，而sq_item又是 PySequenceMethods 中的一个函数指针。

那么最终计算出来的偏移量就存在如下关系：

```
offset(mp_subscript) < offset(sq_item)
```

因此如果在一个PyTypeObject中，既定义了mp_subscript，又定义了sq_item，那么虚拟机将选择 mp_subscript 与 __getitem__ 建立联系。

我们举个栗子：

```
class A(list):

    def __getitem__(self, item):
        return item

a = A([])
print(a)  # []
print(a[0])  # 0
print(a["xxx"])  # xxx
```

我们自定义的类实现了 __getitem__，所以会对应 mp_subscript 或 sq_item，那么到底是哪一种呢？显然根据偏移量的关系，虚拟机最终选择了让 mp_subscript 和 __getitem__ 建立联系。

事实上不看偏移量我们也知道答案，因为 sq_item 表示基于索引取值，如果 `[]` 里面的值是字符串，那么铁定报错。但这里没有报错，说明和 __getitem__ 建立联系的不是 sq_item。

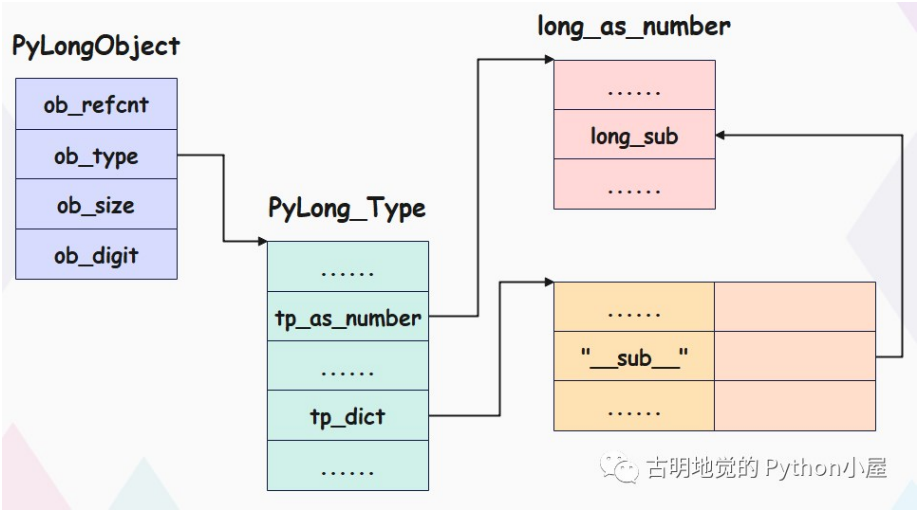
> 另外对于内置类对象而言，则没有这么复杂，因为它们的操作在底层是静态写死的。但对于自定义类对象来说，需要有一个基于偏移量排序、查找的过程。

然后再看一下 slotdefs 的排序过程，它是在 init_slotdefs 中完成的：

```
//typeobject.c
static int slotdefs_initialized = 0;

static void
init_slotdefs(void)
{
    slotdef *p;
    //init_slotdefs只会进行一次
    if (slotdefs_initialized)
        return;
    for (p = slotdefs; p->name; p++) {
        //通过在PyHeapTypeObject中的offset对slots进行排序
        //而且是从小到大排
        assert(!p[1].name || p->offset <= p[1].offset);
        p->name_strobj = PyUnicode_InternFromString(p->name);
        if (!p->name_strobj || !PyUnicode_CHECK_INTERNED(p->name_strobj))
            Py_FatalError("Out of memory interning slotdef names");
    }
    //排完序之后将值赋为1
    //这样的话下次执行到上面的if时
    //由于条件为真会直接 return
    slotdefs_initialized = 1;
}
```

看一下之前的一张图：



当时说 "__sub__" 对应的 value 并不是一个直接指向 long_sub 函数的指针，而是指向一个结构体，至于指向 long_sub 函数的指针则在该结构体内部。

那么问题来了，这个结构体是不是上面的 slot 呢?

我们知道在 slot 中，包含了很多关于一个操作的信息。但是很可惜，在tp_dict中，与 "__sub__" 关联在一起的，一定不会是 slot，因为它不是一个 PyObject，无法将其指针放在字典中。

如果再深入思考一下，会发现 slot 也无法被调用。因为它不是一个 PyObject，那么它就没有 ob_type 这个字段，也就无从谈起什么 tp_call 了，所以 slot 是无论如也无法满足 Python 中的可调用（callable）这一条件的。

前面我们说过，虚拟机在 tp_dict 中找到对应的操作后，会调用该操作，所以 tp_dict 中与 "__sub__" 对应的只能是包装了 slot 的 PyObject（的指针）。在 Python 里面，我们称之为 **wrapper descriptor**。

在Python内部存在多种 **wrapper descriptor**，一个 **wrapper descriptor** 包含一个 slot。而它在底层对应的结构体是由 PyWrapperDescrObject 表示的，我们看一下：

```
//descrobject.h
#define PyDescr_COMMON PyDescrObject d_common

//descriptor
typedef struct {
    PyObject_HEAD
    PyTypeObject *d_type;
    PyObject *d_name;
    PyObject *d_qualname;
} PyDescrObject;

//wrapper descriptor
typedef struct {
    //该成员相当于 PyDescrObject d_common
    PyDescr_COMMON;
    //slot
    struct wrapperbase *d_base;
    //函数指针
    void *d_wrapped;
} PyWrapperDescrObject;
```

以上就是 **wrapper descriptor** 在底层的定义，其创建是通过 PyDescr_NewWrapper 完成的。

```
//descrobject.c
PyObject *
PyDescr_NewWrapper(PyTypeObject *type, struct wrapperbase *base, void *w
```

```
rapped)
{
    //声明 wrapper descriptor
    PyWrapperDescrObject *descr;
    //调用descr_new申请内存
    descr = (PyWrapperDescrObject *)descr_new(&PyWrapperDescr_Type,
                                              type, base->name);
    //设置成员属性
    if (descr != NULL) {
        descr->d_base = base;
        descr->d_wrapped = wrapped;
    }
    return (PyObject *)descr;
}

static PyDescrObject *
descr_new(PyTypeObject *descrtype, PyTypeObject *type, const char *name)
{
    PyDescrObject *descr;
    //为PyDescrObject 申请内存
    descr = (PyDescrObject *)PyType_GenericAlloc(descrtype, 0);
    //设置成员属性
    if (descr != NULL) {
        Py_XINCREF(type);
        descr->d_type = type;
        descr->d_name = PyUnicode_InternFromString(name);
        if (descr->d_name == NULL) {
            Py_DECREF(descr);
            descr = NULL;
        }
        else {
            descr->d_qualname = NULL;
        }
    }
    return descr;
}
```

Python 内部的各种 wrapper descriptor 都会包含 PyDescrObject，也就是类型对象相关的一些信息；d_base 对应 slot；而 d_wrapped 则存放着最重要的东西：操作对应的函数指针，比如 PyLong_Type，其tp_dict["__sub__"].d_wrapped就是&long_sub。

```
print(int.__sub__)
print(str.__add__)
print(str.__getitem__)
"""
<slot wrapper '__sub__' of 'int' objects>
<slot wrapper '__add__' of 'str' objects>
<slot wrapper '__getitem__' of 'str' objects>
"""
```

我们看到这些魔法函数都是一个 wrapper descriptor 对象，也就是对 slot 包装之后的描述符。wrapper descriptor 对象在底层对应 PyWrapperDescrObject，其类型是 PyWrapperDescr_Type，tp_call 为 wrapperdescr_call。

我们在 Python 里面再查看一下类型：

```
print(int.__add__.__class__)
"""
<class 'wrapper_descriptor'>
"""
```

打印的结果是 <class 'wrapper_descriptor'>，说明类型对象 wrapper_descriptor 在底层对应 PyWrapperDescr_Type。

```
PyTypeObject PyWrapperDescr_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "wrapper_descriptor",
    sizeof(PyWrapperDescrObject),
    0,
    (destructor)descr_dealloc,                  /* tp_
    0,                                          /* tp_
    0,                                          /* tp_
    0,                                          /* tp_
    0,                                          /* tp_
    (reprfunc)wrapperdescr_repr,                /* tp_
    0,                                          /* tp_
    0,                                          /* tp_
```


建立联系

排序后的结果仍然存放在 slotdefs 中，虚拟机会从头到尾遍历 slotdefs，基于每一个 slot 建立一个 wrapper descriptor。然后在 tp_dict 中再建立从操作名到 wrapper descriptor 的关联，这个过程是在 add_operators 中完成的。

```
1  static int
2  add_operators(PyTypeObject *type)
3  {
4      //属性字典
5      PyObject *dict = type->tp_dict;
6      //slot，在底层是一个 slotdef 结构体
7      slotdef *p;
8      //wrapper descriptor
9      PyObject *descr;
10     void **ptr;
11     //对slotdefs进行排序
12     init_slotdefs();
13     for (p = slotdefs; p->name; p++) {
14         //如果slot中没有指定wrapper，则无需处理
15         if (p->wrapper == NULL)
16             continue;
17         //获得slot对应的操作在PyTypeObject中的函数指针
18         ptr = slotptr(type, p->offset);
19         if (!ptr || !*ptr)
20             continue;
21         //如果tp_dict中已经存在操作名，则放弃
22         if (PyDict_GetItemWithError(dict, p->name_strobj))
23             continue;
24         if (PyErr_Occurred()) {
25             return -1;
26         }
27         if (*ptr == (void *)PyObject_HashNotImplemented) {
28             if (PyDict_SetItem(dict, p->name_strobj, Py_None) < 0)
29                 return -1;
30         }
31         else {
32             //创建 wrapper descriptor
33             descr = PyDescr_NewWrapper(type, p, *ptr);
34             if (descr == NULL)
35                 return -1;
36             //将 <操作名，wapper descriptor> 放入tp_dict中
37             if (PyDict_SetItem(dict, p->name_strobj, descr) < 0) {
```

```
                Py_DECREF(descr);
                return -1;
            }
            Py_DECREF(descr);
        }
    }
    if (type->tp_new != NULL) {
        if (add_tp_new_wrapper(type) < 0)
            return -1;
    }
    return 0;
}
```

在add_operators中，首先调用init_slotdefs对操作进行排序，然后遍历排序完成后的slotdefs数组。通过slotptr获得该slot对应的操作在PyTypeObject中的函数指针。

并紧接着创建 wrapper descriptor，然后在tp_dict中建立从操作名（slotdef.name_strobj）到操作（wrapper descriptor）的关联。

但需要注意的是，在创建 wrapper descriptor 之前，虚拟机会检查在tp_dict中是否存在同名操作。如果存在了，则不会再次建立从操作名到操作的关联。也正是这种检查机制与排序机制相结合，虚拟机才能在拥有相同操作名的多个操作中选择优先级最高的操作。

add_operators里面的大部分动作都很简单、直观，而最难的动作隐藏在slotptr这个函数当中。它的功能是完成从**slot**到**slot对应操作的真实函数指针**的转换。

我们知道在slot中存放着用来操作的offset，但不幸的是，这个offset是相对于PyHeapTypeObject的偏移，而操作的真实函数指针却是在 PyTypeObject 中指定的。

而且PyTypeObject和PyHeapTypeObject不是同构的，因为 PyHeapTypeObject 中包含了 PyNumberMethods 结构体，但 PyTypeObject 只包含了 PyNumberMethods * 指针。所以slot中存储的关于操作的offset对 PyTypeObject来说，不能直接用，必须先转换。

举个栗子，假如说有如下调用（slotptr 一会说）：

```
slotptr(&PyLong_Type, offset(PyHeapTypeObject, long_sub))
```

首先会判断这个偏移量是否大于 offset(PyHeapTypeObject, as_number)，所以会先从PyTypeObject对象中获得as_number指针p，然后在p的基础上进行偏移就可以得到实际的函数地址。

所以偏移量delta为：

```
offset(PyHeapTypeObject, long_sub) - offset(PyHeapTypeObject, as_number)
```

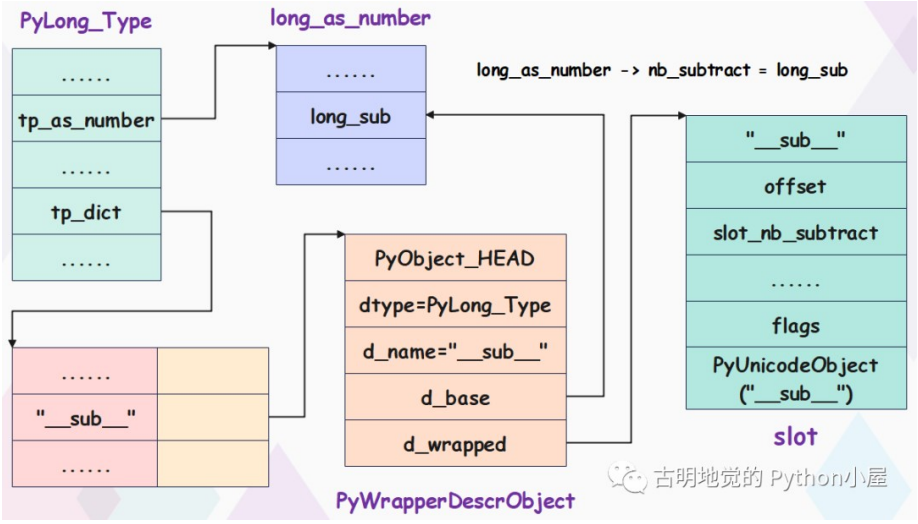
而这个复杂的过程就在slotptr中完成：

```
static void **
slotptr(PyTypeObject *type, int ioffset)
{
    char *ptr;
    long offset = ioffset;

    /* Note: this depends on the order of the members of PyHeapTypeObject! */
    assert(offset >= 0);
    assert((size_t)offset < offsetof(PyHeapTypeObject, as_buffer));
    //从PyHeapTypeObject中排在后面的PySequenceMethods开始判断
    //然后向前，依次判断PyMappingMethods和PyNumberMethods
    /*
    为什么要这么做呢？假设我们首先从PyNumberMethods开始判断
    如果一个操作的offset大于as_numbers在PyHeapTypeObject中的偏移量
    那么我们还是没办法确认这个操作到底是属于谁的
```

```
    只有从后往前进行判断，才能解决这个问题。
    */
    if ((size_t)offset >= offsetof(PyHeapTypeObject, as_sequence)) {
        ptr = (char *)type->tp_as_sequence;
        offset -= offsetof(PyHeapTypeObject, as_sequence);
    }
    else if ((size_t)offset >= offsetof(PyHeapTypeObject, as_mapping)) {
        ptr = (char *)type->tp_as_mapping;
        offset -= offsetof(PyHeapTypeObject, as_mapping);
    }
    else if ((size_t)offset >= offsetof(PyHeapTypeObject, as_number)) {
        ptr = (char *)type->tp_as_number;
        offset -= offsetof(PyHeapTypeObject, as_number);
    }
    else if ((size_t)offset >= offsetof(PyHeapTypeObject, as_async)) {
        ptr = (char *)type->tp_as_async;
        offset -= offsetof(PyHeapTypeObject, as_async);
    }
    else {
        ptr = (char *)type;
    }
    if (ptr != NULL)
        ptr += offset;
    return (void **)ptr;
}
```

好了，我想到现在我们应该能够摸清楚虚拟机在改造PyTypeObject对象时，对tp_dict做了什么了，我们以 PyLong_Type 举例说明：



在add_operators完成之后，PyLong_Type如图所示。

从PyLong_Type.tp_as_number中延伸出去的部分是在编译时就已经确定好了的，而从tp_dict中延伸出去的部分则是在**Python运行时环境初始化**的时候才建立的。

> 这个运行时环境初始化后面会单独说，现在就把它理解为解释器启动时做的准备工作即可。

另外， PyType_Ready 在通过add_operators添加了PyTypeObject中定义的一些operator后，还会通过add_methods、add_numbers和add_getsets添加 PyTypeObject 中定义的 tp_methods、tp_members和tp_getset函数集。

这些过程和add_operators类似，不过最后添加到tp_dict中的就不再是 PyWrapperDescrObject ，而分别是 PyMethodDescrObject 、PyMemberDescrObject 、PyGetSetDescrObject 。

```
print(int.__add__)
print((123).__add__)
"""
<slot wrapper '__add__' of 'int' objects>
```

```
5 <method-wrapper '__add__' of int object at 0x00007FFBC13615F0>
6 """
```

实例在调用函数的时候，会将函数包装成方法，它是一个 wrapper method。所以 __add__ 对于 int 类型对象而言，叫魔法函数；对于整数对象而言，叫魔法方法。

从目前来看，基本上算是解析完了，但是还有一点：

```
1 class A(int):
2
3     def __sub__(self, other):
4         return (self, other)
5
6 a = A(123)
7 print(a - 456)  # (123, 456)
```

从结果上很容易看出，进行减法操作时，调用的是我们重写的 __sub__。这意味着虚拟机在初始化 A 的时候，对tp_as_number中的nb_subtract进行了特殊处理。

那么为什么虚拟机会知道要对nb_subtract进行特殊处理呢？当然肯定有小伙伴会说：这是因为我们重写了__sub__啊，确实如此，但这是Python层面上的，在虚拟机层面的话，答案还是在slot身上。

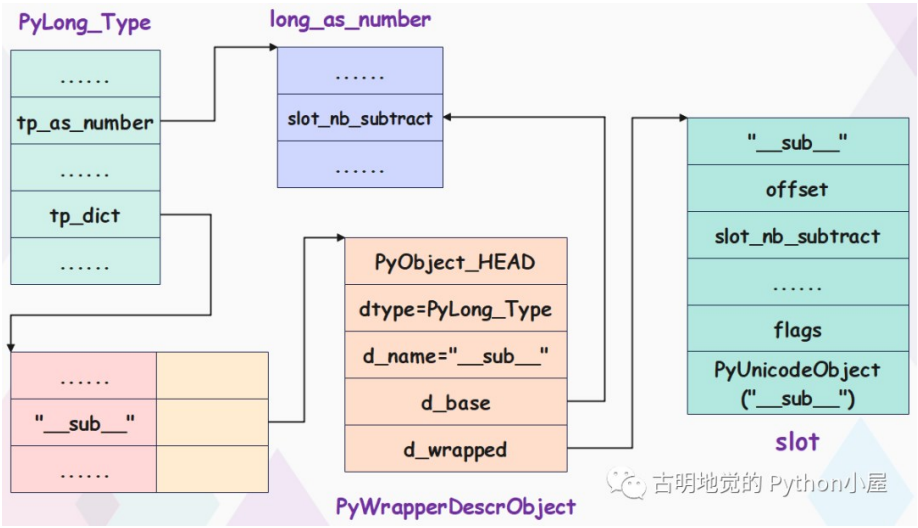
```
1 //typeobject.c
2 static slotdef slotdefs[] = {
3     //...
4     BINSLOT("__sub__", nb_subtract, slot_nb_subtract,
5             "-"),
6     //...
7 }
```

虚拟机在初始化类对象 A 时，会检查 A 的tp_dict中是否存在__sub__。在后面剖析自定义类对象的创建时会看到，因为在定义class A的时候，重写了__sub__这个操作，所以在 A 的 tp_dict 中，__sub__一开始就会存在，虚拟机会检测到。

然后再根据__sub__对应的 slot 顺藤摸瓜，找到 np_substract，并且将这个函数指针替换为 slot 中指定的 &slot_nb_subtract。所以当后来虚拟机找 A 的 np_substract 的时候，实际上找的是 slot_nb_subtract。

而在 slot_nb_subtract 中，会寻找__sub__对应的描述符，然后找到在 A 中重写的函数（一个 PyFunctionObject *）。这样一来，就完成了对 int 的__sub__行为的替换。

所以对于 A 来说，内存布局就是下面这样。



当然这仅仅是针对于__sub__，至于其它操作还是会指向PyLong_Type中指定的函数。所以如果某个函数在 A 里面没有重写的话，那么会从PyLong_Type中寻找。

而将 np_substract 替换成 slot_nb_subtract，这一步是在 fixup_slot_dispatchers 中实现的。注意：内置类对象则不会进行此操作，因为内置类对象是被静态初始化的，它不允许属性的动态设置。

```
//typeobject.c
static void
fixup_slot_dispatchers(PyTypeObject *type)
{
    slotdef *p;

    init_slotdefs();
    for (p = slotdefs; p->name; )
        p = update_one_slot(type, p);
}
```

以上就是属性字典的填充，这个过程比较复杂，它是在解释器启动之后动态完善的。

收录于合集 #CPython 97

上一篇
《源码探秘 CPython》71. 类对象 MRO 的设置，与基类的继承

下一篇
《源码探秘 CPython》69. 给类型对象设置类型和基类信息