从创建到销毁，对象的生命周期

当我们在控制台敲下这个语句， *Python* 内部是如何从无到有创建一个浮点对象的？

```
>>> pi = 3.14
```

另外，*Python* 又是怎么知道该如何将它打印到屏幕上的呢？

```
>>> print(pi)
3.14
```

对象使用完毕， *Python* 必须将其销毁，销毁的时机又该如何确定呢？ 带着这些问题，接着考察对象在从创建到销毁整个生命周期中的行为表现，从中探寻答案。

以下讨论以一个足够简单的类型 *float* 为例，对应的 *C* 实体是 *PyFloat_Type* 。

C API

开始讨论对象创建前，先介绍 *Python* 提供的 *C API* 。

Python 是用 *C* 写成的，对外提供了 *C API* ，让用户可以从 *C* 环境中与其交互。 *Python* 内部也大量使用这些 *API* ，为了更好研读源码，先系统了解 *API* 组成结构很有必要。 *C API* 分为两类： **泛型API** 以及 **特型API** 。

泛型API

泛型API 与类型无关，属于 **抽象对象层** (*Abstract Object Layer*)，简称 *AOL* 。 这类 API 参数是 *PyObject** ，可处理任意类型的对象， *API* 内部根据对象类型区别处理。

以对象打印函数为例：

```
int
PyObject_Print(PyObject *op, FILE *fp, int flags)
```

接口第一个参数为待打印对象，可以是任意类型的对象，因此参数类型是 *PyObject** 。 *Python* 内部一般都是通过 *PyObject** 引用对象，以达到泛型化的目的。

对于任意类型的对象，均可调用 *PyObject_Print* 将其打印出来：

```
PyObject *fo = PyFloatObject_FromDouble(3.14);
PyObject_Print(fo, stdout, 0);


PyObject *lo = PyFloatObject_FromLong(100);
PyObject_Print(lo, stdout, 0);
```

PyObject_Print 接口内部根据对象类型，决定如何输出对象。

特型API

特型API 与类型相关，属于 **具体对象层** (*Concrete Object Layer*)，简称 *COL* 。 这类 *API* 只能作用于某种类型的对象，例如浮点对象 *PyFloatObject* 。 *Python* 内部为每一种内置对象提供了这样一组 *API* ，举例如下：

```
PyObject *
PyFloat_FromDouble(double fval)
```

PyFloat_FromDouble 创建一个浮点对象，并将它初始化为给定值 *fval* 。

对象的创建

经过前面的理论学习，我们知道对象的 **元数据** 保存在对应的 **类型对象** 中，元数据当然也包括 **对象如何创建** 的信息。 因此，有理由相信 **实例对象** 由 **类型对象** 创建。

不管创建对象的流程如何，最终的关键步骤都是 **分配内存** 。 *Python* 对 **内建对象** 是无所不知的，因此可以提供 *C API* ，直接分配内存并执行初始化。 以 *PyFloat_FromDouble* 为例，在接口内部为 *PyFloatObject* 结构体分配内存，并初始化相关字段即可。

对于用户自定义的类型 *class Dog(object)* ， *Python* 就无法事先提供 *PyDog_New* 这样的 *C API* 了。 这种情况下，就只能通过 *Dog* 所对应的类型对象创建实例对象了。 至于需要分配多少内存，如何进行初始化，答案就需要在 **类型对象** 中找了。

总结起来，*Python* 内部一般通过这两种方法创建对象：

- 通过 *C API* ，例如 *PyFloat_FromDouble* ，多用于内建类型；
- 通过类型对象，例如 *Dog* ，多用于自定义类型；

通过类型对象创建实例对象，是一个更通用的流程，同时支持内置类型和自定义类型。 以创建浮点对象为例，我们还可以通过浮点类型 *PyFloat_Type* 来创建：

```
>>> pi = float('3.14')
>>> pi
3.14
```

例子中我们通过调用类型对象 *float* ，实例化了一个浮点实例 *pi* ，对象居然还可以调用! 在 *Python* 中，可以被调用的对象就是 **可调用对象** 。

问题来了，可调用对象被调用时，执行什么函数呢？ 由于类型对象保存着实例对象的元信息，*float* 类型对象的类型是 *type* ，因此秘密应该就隐藏在 *type* 中。

再次考察 *PyType_Type* ，我们找到了 *tp_call* 字段，这是一个函数指针：

```
PyTypeObject PyType_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "type",
    sizeof(PyHeapTypeObject),
    sizeof(PyMemberDef),


    (ternaryfunc)type_call,


};
```

当实例对象被调用时，便执行 *tp_call* 字段保存的处理函数。

因此， *float('3.14')* 在 *C* 层面等价于：

```
PyFloat_Type.ob_type.tp_call(&PyFloat_Type, args, kwargs)
```

即：

```
PyType_Type.tp_call(&PyFloat_Type, args, kwargs)
```

最终执行， type_call **函数：**

```
type_call(&PyFloat_Type, args, kwargs)
```

调用参数通过 *args* 和 *kwargs* 两个对象传递，先不展开，留到函数机制中详细介绍。

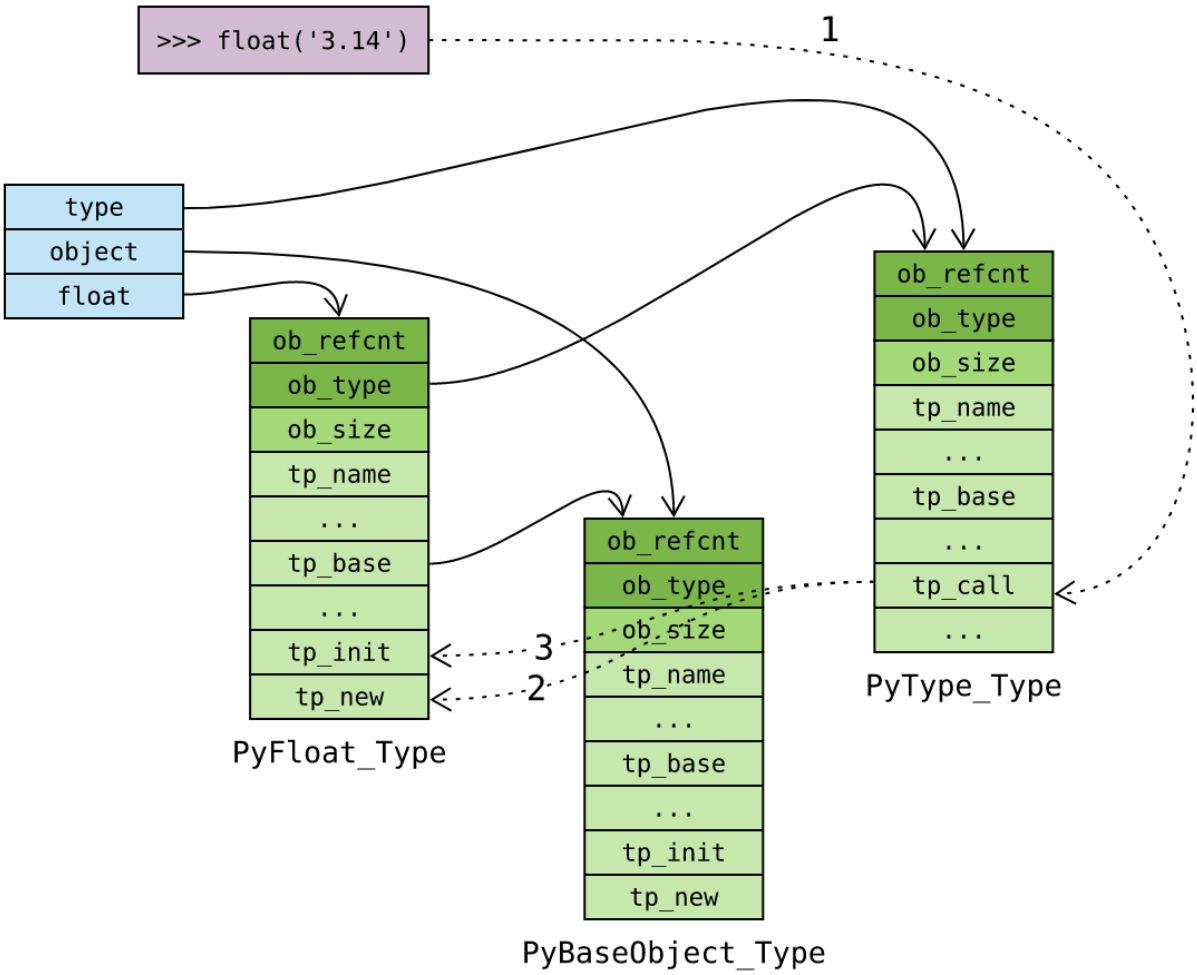
接着围观 *type_call* 函数，定义于 *Include/typeobject.c* ，关键代码如下：

```
static PyObject *
type_call(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
   PyObject *obj;


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


   type = Py_TYPE(obj);
   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());
      }
   }
   return obj;
}
```

可以看到，关键的步骤有两个：

1. **调用类型对象** *tp_new* **函数指针 申请内存** (第 *7* 行)；
2. **必要时调用类型对象** *tp_init* **函数指针对对象进行 初始化** (第 *15* 行)；

至此，对象的创建过程已经非常清晰了：



总结一下，*float* 类型对象是 **可调用对象** ，调用 *float* 即可创建实例对象：

1. 调用 *float* ， *Python* 最终执行其类型对象 *type* 的 *tp_call* 函数；
2. *tp_call* 函数调用 *float* 的 *tp_new* 函数为实例对象分配 **内存空间** ；
3. *tp_call* 函数必要时进一步调用 *tp_init* 函数对实例对象进行 **初始化** ；

对象的多态性

Python 创建一个对象，比如 *PyFloatObject* ，会分配内存，并进行初始化。 此后， *Python* 内部统一通过一个 *PyObject** 变量来保存和维护这个对象，而不是通过 *PyFloatObject** 变量。

通过 *PyObject** 变量保存和维护对象，可以实现更抽象的上层逻辑，而不用关心对象的实际类型和实现细节。 以对象哈希值计算为例，假设有这样一个函数接口：

```
Py_hash_t
PyObject_Hash(PyObject *v);
```

该函数可以计算任意对象的哈希值，不管对象类型是啥。 例如，计算浮点对象哈希值：

```
PyObject *fo = PyFloatObject_FromDouble(3.14);
PyObject_Hash(fo);
```

对于其他类型，例如整数对象，也是一样的：

```
PyObject *lo = PyLongObject_FromLong(100);
PyObject_Hash(lo);
```

然而，对象类型不同，其行为也千差万别，哈希值计算方法便是如此。 *PyObject_Hash* 函数如何解决这个问题呢？ 到 *Object/object.c* 中寻找答案：

```
Py_hash_t
PyObject_Hash(PyObject *v)
{
   PyTypeObject *tp = Py_TYPE(v);
   if (tp->tp_hash != NULL)
      return (*tp->tp_hash)(v);

   if (tp->tp_dict == NULL) {
      if (PyType_Ready(tp) < 0)
         return -1;
      if (tp->tp_hash != NULL)
         return (*tp->tp_hash)(v);
   }

   return PyObject_HashNotImplemented(v);
}
```

函数先通过 *ob_type* 指针找到对象的类型(第 *4* 行)； 然后通过类型对象的 *tp_hash* 函数指针，调用对应的哈希值计算函数(第 *6* 行)。 换句话讲， *PyObject_Hash* 根据对象的类型，调用不同的函数版本。 这不就是 **多态** 吗？

通过 *ob_type* 字段， *Python* 在 *C* 语言层面实现了对象的 **多态** 特性， 思路跟 *C++* 中的 **虚表指针** 有异曲同工之妙。

对象的行为

不同对象的行为不同，比如哈希值计算方法就不同，由类型对象中 *tp_hash* 字段决定。 除了 *tp_hash* ，我们看到 *PyTypeObject* 结构体还定义了很多函数指针，这些指针最终都会指向某个函数，或者为空。 这些函数指针可以看做是 **类型对象** 中定义的 **操作** ，这些操作决定对应 **实例对象** 在运行时的 **行为** 。

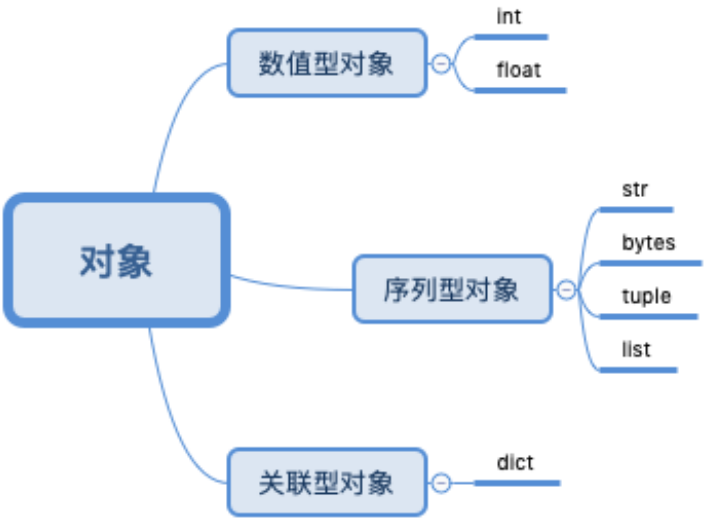
尽管如此，不同对象也有一些共性。 举个例子，**整数对象** 和 **浮点对象** 都支持加减乘除等 **数值型操作** ：

```
>>> 1 + 2
3
>>> 3.14 * 3.14
9.8596
```

元组对象 *tuple* 和 **列表对象** *list* 都支持下标操作：

```
>>> t = ('apple', 'banana', 'car', 'dog')
>>> t[-1]
'dog'
>>> l = ['alpha', 'beta']
>>> l[-1]
'beta'
```

因此，以对象行为为依据，可以对对象进行分类：



Python 便以此为依据，为每个类别都定义了一个 **标准操作集** ：

- *PyNumberMethods* 结构体定义了 **数值型** 操作；
- *PySequenceMethods* 结构体定义了 **序列型** 操作；
- *PyMappingMethods* 结构体定义了 **关联型** 操作；

只要 **类型对象** 提供相关 **操作集** ， **实例对象** 便具备对应的 **行为** 。 操作集字段如下：

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name;
    Py_ssize_t tp_basicsize, tp_itemsize;




    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;



    PyBufferProcs *tp_as_buffer;


} PyTypeObject;
```

以 *float* 为例，类型对象 *PyFloat_Type* 相关字段是这样初始化的：

```
static PyNumberMethods float_as_number = {
    float_add,
    float_sub,
    float_mul,
    float_rem,
    float_divmod,
    float_pow,

};

PyTypeObject PyFloat_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "float",
    sizeof(PyFloatObject),


    &float_as_number,
    0,
    0,


};
```

- 字段 *tp_as_number* 非空，因此 *float* 对象 **支持数值型操作** ；
- 字段 *tp_as_sequence* 为空，因此 *float* 对象 **不支持序列型操作** ；
- 字段 *tp_as_mapping* 为空，因此 *float* 对象 **不支持关联型操作** ；

注意到， *float_as_number* 变量中相关函数指针都初始化为对应的 *float* 版本操作函数。 由于篇幅有限，这里先不深入展开。

引用计数

C/C++ 赋予程序员极大的自由，可以任意申请内存，并按自己的意图灵活管理。 然而，权利的另一面则对应着 **责任** ，一旦内存不再使用，程序员必须将其释放。 这给程序员带来极大的 **工作负担** ，并导致大量问题： **内存泄露** 、 **野指针** 、 **越界访问** 等。

许多后来兴起的开发语言，如 *Java* 、 *Golang* 等，选择 **由语言本身负责内存的管理** 。 **垃圾回收机制** 的引入，程序员摆脱了内存管理的噩梦，可以更专注于业务逻辑。 于此同时，开发人员失去了灵活使用内存的机会，也牺牲了一定的执行效率。

随着垃圾回收机制日益完善，可在大部分对性能要求不苛刻的场景中引入，利大于弊。 *Python* 也采用垃圾回收机制，代替程序员进行繁重的内存管理，**提升开发效率** 的同时，降低 *bug* 发生的几率。

Python 垃圾回收机制的关键是对象的 **引用计数** ，它决定了一个对象的生死。 我们知道每个 *Python* 对象都有一个 *ob_refcnt* 字段，记录着对象当前的引用计数。 当对象被其他地方引用时， *ob_refcnt* 加一 ；当引用解除时， *ob_refcnt* 减一。 当 *ob_refcnt* 为零，说明对象已经没有任何引用了，这时便可将其回收。

Python 对象创建后，引用计数设为 *1* ：

```
>>> a = 3.14
>>> sys.getrefcount(a)
2
```

咦？这里引用计数为啥是 *2* 呢？

对象作为函数参数传递，需要将引用计数加一，避免对象被提前销毁；函数返回时，再将引用计数减一。 因此，例子中 *getrefcount* 函数看到的对象引用计数为 *2* 。

接着，变量赋值让对象多了一个引用，这很好理解：

```
>>> b = a
>>> sys.getrefcount(a)
3
```

将对象放在容器对象中，引用计数也增加了，符合预期：

```
>>> l = [a]
>>> l
[3.14]
>>> sys.getrefcount(a)
4
```

我们将 *b* 变量删除，引用计数减少了：

```
>>> del b
>>> sys.getrefcount(a)
3
```

接着，将列表清空，引用计数进一步下降：

```
>>> l.clear()
>>> sys.getrefcount(a)
2
```

最后，将变量 *a* 删除后，引用计数降为 *0* ，便不复存在了：

```
>>> del a
```

在 *Python* 中，很多场景都涉及引用计数的调整，例如：

- 容器操作；
- 变量赋值；
- 函数参数传递；
- 属性操作;

为此， *Python* 定义了两个非常重要的宏，用于维护对象应用计数。 其中， *Py_INCREF* 将对象应用计数加一(*3* 行)：

```
#define Py_INCREF(op) (                         \
    _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
    ((PyObject *)(op))->ob_refcnt++)
```

Py_DECREF 将引用计数减一(*5* 行)，并在引用计数为 *0* 是回收对象(*8* 行)：

```
#define Py_DECREF(op)                                   \
    do {                                            \
        PyObject *_py_decref_tmp = (PyObject *)(op);    \
        if (_Py_DEC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
        --(_py_decref_tmp)->ob_refcnt != 0)             \
            _Py_CHECK_REFCNT(_py_decref_tmp)            \
        else                                        \
            _Py_Dealloc(_py_decref_tmp);                \
    } while (0)
```

当一个对象引用计数为 *0* ， *Python* 便调用对象对应的析构函数销毁对象，但这并不意味着对象内存一定会回收。 为了提高内存分配效率， *Python* 为一些常用对象维护了内存池， 对象回收后内存进入内存池中，以便下次使用，由此 **避免频繁申请、释放内存** 。

内存池 技术作为程序开发的高级话题，需要更大的篇幅，放在后续章节中介绍。