06 小试牛刀，解剖浮点对象 float-慕课专栏

经过前面章节，我们知道 *float* 对象背后由 *PyFloatObject* 组织，对其结构也了然于胸。 那么，本节为何要重复讨论 *float* 对象呢？

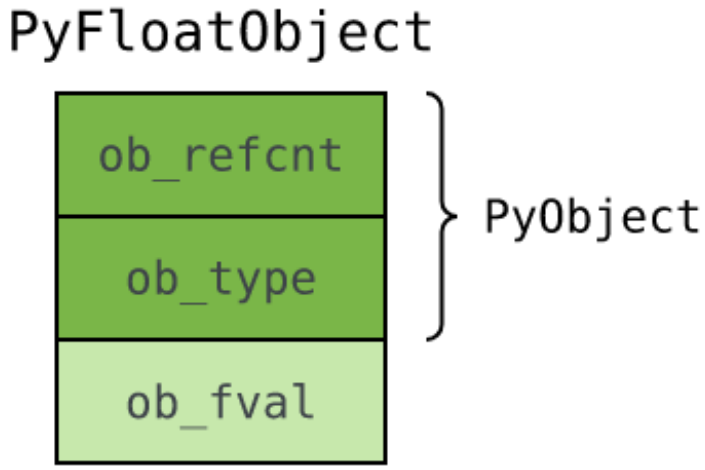
一方面， 对象模型 中关于 *float* 对象的讨论， 着眼于 *Python* 面向对象体系的讲解，许多细节没来得及展开。 另一方面， *float* 作为 *Python* 中最简单的对象之一，“麻雀虽小，五脏俱全”，拥有对象的全部必要属性。 以 *float* 对象为起点开启源码之旅，能够快速上手，为研究更复杂的内置对象打好基础，建立信心。

内部结构

float 实例对象在 *Include/floatobject.h* 中定义，结构很简单：

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

除了定长对象共用的头部，只有一个额外的字段 *ob_fval* ，存储对象所承载的浮点值。



浮点实例对象内部结构

float 类型对象又长啥样呢？

与实例对象不同， *float* 类型对象 **全局唯一** ，因此可以作为 **全局变量** 定义。 在 *C* 文件 *Objects/floatobject.c* 中，我们找到了代表 *float* 类型对象的全局变量 *PyFloat_Type* ：

```
PyTypeObject PyFloat_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "float",
    sizeof(PyFloatObject),
    0,
    (destructor)float_dealloc,
    0,
    0,
    0,
    0,
    (reprfunc)float_repr,
    &float_as_number,
    0,
    0,
    (hashfunc)float_hash,
    0,
    (reprfunc)float_repr,
    PyObject_GenericGetAttr,
    0,
    0,
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    float_new__doc__,
    0,
    0,
    float_richcompare,
    0,
    0,
    0,
    float_methods,
    0,
    float_getset,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    float_new,
};
```

PyFloat_Type 中保存了很多关于浮点对象的 **元信息** ，关键字段包括：

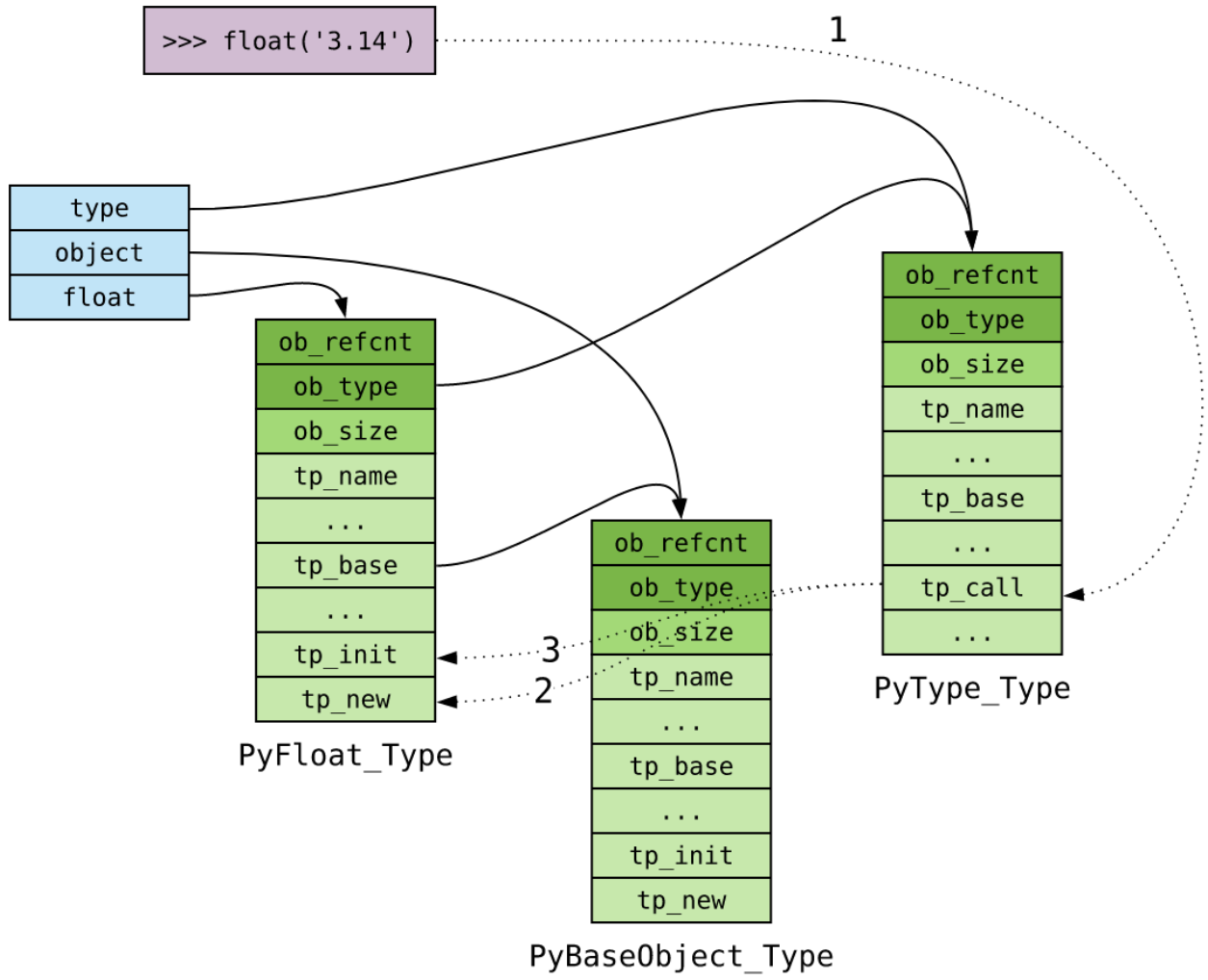
- *tp_name* 字段保存类型名称，常量 *float* ；
- *tp_dealloc* 、 *tp_init* 、 *tp_alloc* 和 *tp_new* 字段是对象创建销毁相关函数；
- *tp_repr* 字段是生成语法字符串表示形式的函数；
- *tp_str* 字段是生成普通字符串表示形式的函数；
- *tp_as_number* 字段是数值操作集；
- *tp_hash* 字段是哈希值生成函数；

PyFloat_Type 很重要，作为浮点 **类型对象** ，它决定了浮点 **实例对象** 的 **生死和行为** 。 接下来，我们以不同小节分别展开讨论。

对象的创建

在 从创建到销毁，对象的生命周期 一节，我们初步了解到创建实例对象的一般过程。

调用类型对象 *float* 创建实例对象，*Python* 执行的是 *type* 类型对象中的 *tp_call* 函数。 *tp_call* 函数进而调用 *float* 类型对象的 *tp_new* 函数创建实例对象， 再调用 *tp_init* 函数对其进行初始化：



创建对象的过程

注意到， *float* 类型对象 *PyFloat_Type* 中 *tp_init* 函数指针为空。 这是因为 *float* 是一种很简单的对象，初始化操作只需要一个赋值语句，在 *tp_new* 中完成即可。

除了通用的流程， *Python* 为内置对象实现了对象创建 *API* ，简化调用，提高效率：

```
PyObject *
PyFloat_FromDouble(double fval);

PyObject *
PyFloat_FromString(PyObject *v);
```

- *PyFloat_FromDouble* 函数通过浮点值创建浮点对象；

- *PyFloat_FromString* 函数通过字符串对象创建浮点对象；

以 *PyFloat_FromDouble* 为例，特化的对象创建流程如下：

```
PyObject *
PyFloat_FromDouble(double fval)
{
    PyFloatObject *op = free_list;
    if (op != NULL) {
        free_list = (PyFloatObject *) Py_TYPE(op);
        numfree--;
    } else {
        op = (PyFloatObject*) PyObject_MALLOC(sizeof(PyFloatObject));
        if (!op)
            return PyErr_NoMemory();
    }

    (void)PyObject_INIT(op, &PyFloat_Type);
    op->ob_fval = fval;
    return (PyObject *) op;
}
```

1. 为对象 **分配内存空间** ，优先使用空闲对象缓存池 (第 *4-12* 行)；
2. 初始化 **对象类型** 字段 *ob_type* 以及 **引用计数** 字段 *ob_refcnt* (第 *14* 行)；
3. 将 *ob_fval* 字段初始化为指定 **浮点值** (第 *15* 行)；

其中，宏 *PyObject_INIT* 在头文件 *Include/objimpl.h* 中定义：

```
#define PyObject_INIT(op, typeobj) \
    ( Py_TYPE(op) = (typeobj), _Py_NewReference((PyObject *)(op)), (op) )
```

宏 *_Py_NewReference* 将对象引用计数初始化为 *1* ，在 *Include/Object.h* 中定义：

```
#define _Py_NewReference(op) (                          \
    _Py_INC_TPALLOCS(op) _Py_COUNT_ALLOCS_COMMA         \
    _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA               \
    Py_REFCNT(op) = 1)
```

对象的销毁

当对象不再需要时， *Python* 通过 *Py_DECREF* 或者 *Py_XDECREF* 宏减少引用计数； 当引用计数降为 *0* 时， *Python* 通过 *_Py_Dealloc* 宏回收对象。

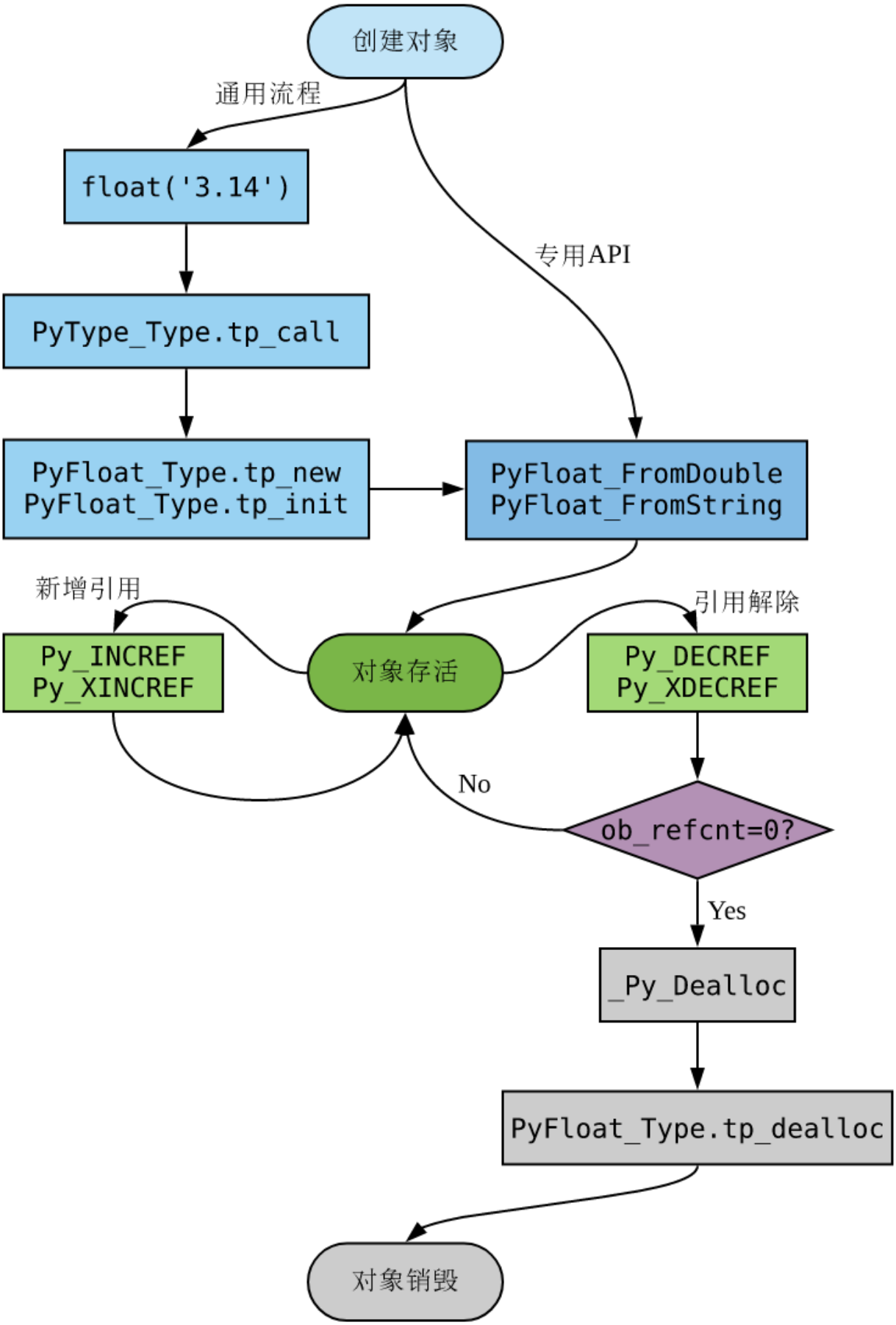
_Py_Dealloc 宏调用类型对象 *PyFloat_Type* 中的 *tp_dealloc* 函数指针：

```
#define _Py_Dealloc(op) (                           \
    _Py_INC_TPFREES(op) _Py_COUNT_ALLOCS_COMMA      \
    (*Py_TYPE(op)->tp_dealloc)((PyObject *)(op)))
```

因此，实际调用的函数是 *float_dealloc* (代码在下一小节 **空闲对象缓存池** 中解析)：

```
static void
float_dealloc(PyFloatObject *op)
{
    if (PyFloat_CheckExact(op)) {
        if (numfree >= PyFloat_MAXFREELIST)  {
            PyObject_FREE(op);
            return;
        }
        numfree++;
        Py_TYPE(op) = (struct _typeobject *)free_list;
        free_list = op;
    }
    else
        Py_TYPE(op)->tp_free((PyObject *)op);
}
```

总结起来，对象从创建到销毁整个生命周期所涉及的 关键函数、宏及调用关系 如下：


创建对象
通用流程
float('3.14')
专用API
PyType_Type.tp_call
PyFloat_Type.tp_new
PyFloat_Type.tp_init
PyFloat_FromDouble
PyFloat_FromString
新增引用
引用解除
Py_INCREF
Py_XINCREF
对象存活
Py_DECREF
Py_XDECREF
No
ob_refcnt=0?
Yes
_Py_Dealloc
PyFloat_Type.tp_dealloc
对象销毁

空闲对象缓存池

浮点运算背后涉及 **大量临时对象创建以及销毁** ，以计算圆周率为例：

```
>>> area = pi * r ** 2
```

这个语句首先计算半径 *r* 的平方，中间结果由一个临时对象来保存，假设是 *t* ；然后计算圆周率 *pi* 与 *t* 的乘积，得到最终结果并赋值给变量 *area* ；最后，销毁临时对象 *t* 。 这么简单的语句，背后居然都隐藏着一个临时对象的创建以及销毁操作！

创建对象时需要分配内存，销毁对象时又需要回收内存。 **大量临时对象创建销毁** ，意味着 **大量内存分配回收操作** ，这显然是是不可接受的。

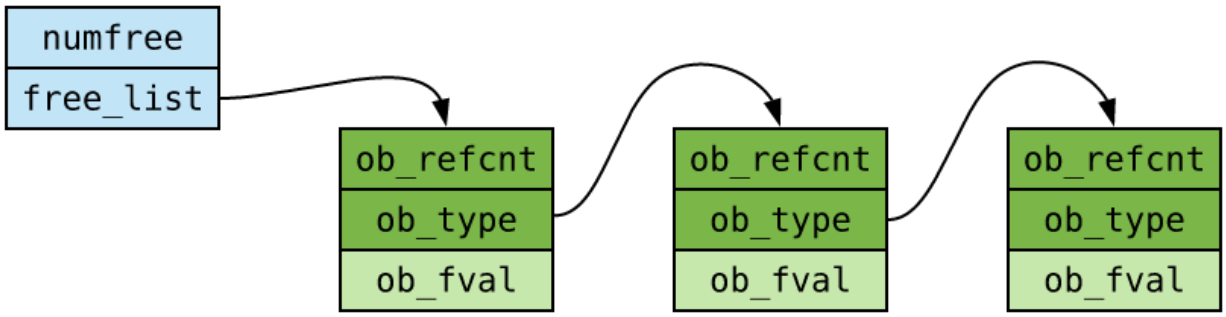
因此 *Python* 在浮点对象销毁后，并不急于回收内存，而是将对象放入一个 **空闲链表** 。 后续需要创建浮点对象时，先到空闲链表中取，省去分配内存的开销。

浮点对象的空闲链表同样在 *Objects/floatobject.c* 中定义：

```
#ifndef PyFloat_MAXFREELIST
#define PyFloat_MAXFREELIST    100
#endif
static int numfree = 0;
static PyFloatObject *free_list = NULL;
```

- *free_list* 变量，指向空闲链表 **头节点** 的指针；
- *numfree* 变量，维护空闲链表 **当前长度** ；
- *PyFloat_MAXFREELIST* 宏，限制空闲链表的 **最大长度** ，避免占用过多内存；

为了保持简洁， *Python* 把 *ob_type* 字段当作 *next* 指针来用，将空闲对象串成链表：



空闲对象链表

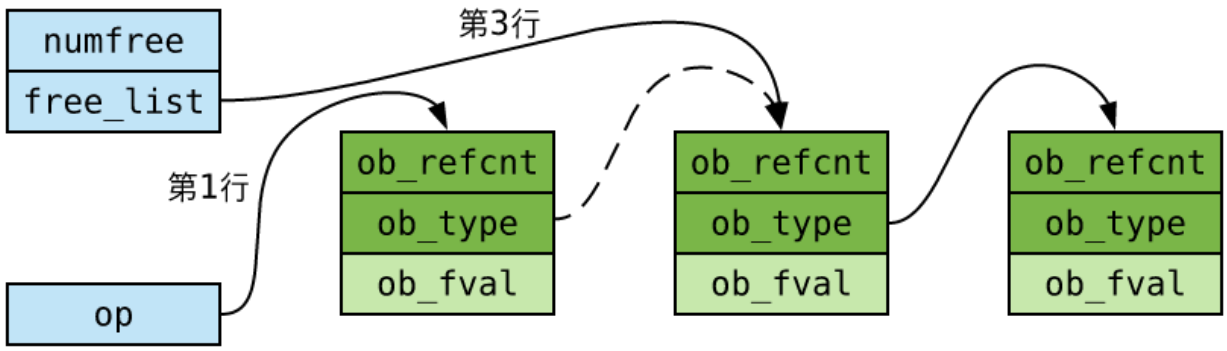
由此一来，需要创建浮点对象时，可以从链表中取出空闲对象，省去申请内存的开销！ 以 *PyFloat_FromDouble* 为例：

```
PyFloatObject *op = free_list;
if (op != NULL) {
    free_list = (PyFloatObject *) Py_TYPE(op);
    numfree--;
} else {
    op = (PyFloatObject*) PyObject_MALLOC(sizeof(PyFloatObject));

}
```

1. 检查 *free_list* 是否为空(第 *2* 行)；
2. 如果 *free_list* 非空，取出头节点备用， 并将 *numfree* 减一(第 *3-4* 行)；
3. 如果 *free_list* 为空，则调用 *PyObject_MALLOC* 分配内存(第 *6* 行)；

如果对 C 语言链表操作不熟悉，可以结合以下图示加以理解：



对象销毁时， *Python* 将其缓存在空闲链表中，以备后用。考察 *float_dealloc* 函数：

```
if (numfree >= PyFloat_MAXFREELIST)  {
    PyObject_FREE(op);
    return;
}
numfree++;
Py_TYPE(op) = (struct _typeobject *)free_list;
free_list = op;
```

1. 第 *1-4* 行，空闲链表长度达到限制值，调用 *PyObject_FREE* 回收对象内存；
2. 第 *5-7* 行，空闲链表长度暂未达到限制，将对象插到空闲链表头部；

这就是 *Python* 空闲对象缓存池的全部秘密！ 由于空闲对象缓存池在 **提高对象分配效率** 方面发挥着至关重要的作用， 后续研究其他内置对象时，我们还会经常看到它的身影。

对象的行为

PyFloat_Type 中定义了很多函数指针，包括 *tp_repr* 、 *tp_str* 、 *tp_hash* 等。 这些函数指针将一起决定 *float* 对象的行为，例如 *tp_hash* 函数决定浮点哈希值的计算：

```
>>> pi = 3.14
>>> hash(pi)
322818021289917443
```

tp_hash 函数指针指向 *float_hash* 函数，实现了针对浮点对象的哈希值算法：

```
static Py_hash_t
float_hash(PyFloatObject *v)
{
    return _Py_HashDouble(v->ob_fval);
}
```

数值操作集

由于加减乘除等数值操作很常见， *Python* 将其抽象成数值操作集 *PyNumberMethods* 。 数值操作集 *PyNumberMethods* 在头文件 *Include/object.h* 中定义：

```
typedef struct {


    binaryfunc nb_add;
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;


    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;

} PyNumberMethods;
```

PyNumberMethods **定义了各种数学算子的处理函数，数值计算最终由这些函数执行。 处理函数根据参数个数可以分为： 一元函数** (*unaryfunc*)、 **二元函数** (*binaryfunc*)等。

回到 *Objects/floatobject.c* 观察浮点对象数值操作集 *float_as_number* 是如何初始化的：

```
static PyNumberMethods float_as_number = {
    float_add,
    float_sub,
    float_mul,
    float_rem,
    float_divmod,
    float_pow,
    (unaryfunc)float_neg,


    0,
    0,
    0,
    0,
    0,

};
```

以加法为例，以下语句在 *Python* 内部最终由 *float_add* 函数执行：

```
>>> a = 1.5
>>> b = 1.1
>>> a + b
2.6
```

float_add 是一个 **二元函数** ，同样位于 *Objects/floatobject.h* 中：

```
static PyObject *
float_add(PyObject *v, PyObject *w)
{
   double a,b;
   CONVERT_TO_DOUBLE(v, a);
   CONVERT_TO_DOUBLE(w, b);
   PyFPE_START_PROTECT("add", return 0)
   a = a + b;
   PyFPE_END_PROTECT(a)
   return PyFloat_FromDouble(a);
}
```

函数实现只有寥寥几步： 首先，将两个参数对象转化成浮点值(*5-6* 行)； 然后，对两个浮点值求和(*8* 行)； 最后，创建一个新浮点对象保存计算结果并返回(*10* 行)。

面试题精讲

例题一

以下例子中， *area* 计算过程中有临时对象创建吗？为什么？

```
>>> pi = 3.14
>>> r = 2
>>> area = pi * r ** 2
```

Python 如何优化临时对象创建效率？

解析

这个语句首先计算半径 *r* 的平方，中间结果由一个临时对象来保存，假设是 *t* ； 然后计算圆周率 *pi* 与 *t* 的乘积，得到最终结果并赋值给变量 *area* ； 最后，销毁临时对象 *t* 。

为了提高浮点对象创建效率， *Python* 引入了 **空闲对象缓存池** 。

浮点对象销毁后， *Python* 并不急于回收内存，而是将对象放入一个 **空闲链表** 。 后续需要创建浮点对象时，先到空闲链表中取，省去分配内存的开销。

例题二

以下例子中，变量 e 的 id 值为何与已销毁的变量 pi 相同？

```
>>> pi = 3.14
>>> id(pi)
4565221808
>>> del pi
>>> e = 2.71
>>> id(e)
4565221808
```

解析

Python 为了优化浮点对象内存分配效率，引入了 **空闲对象缓存池** 。 浮点对象销毁后，*Python* 并不急于回收对象内存，而是将对象缓存在空闲链表中，以备后用。

例子中， *pi* 对象销毁后， *Python* 先不回收对象内存，而是将其插空闲对象链表头部。 当创建浮点对象 *e* 时， *Python* 从链表头取出空闲对象来用，省去了申请内存的开销。 换句话讲， *pi* 对象销毁后被 *e* 重新利用了，因此 *id* 值相同也就不奇怪了。