一直以来有很多毕业生跑来问我，Python的垃圾回收机制到底是什么回事？从网上找到一大堆的文档，看的也是一知半解，最终就学会了一句话：`引用计数器为主、分代码回收和标记清除为辅`。

就这么一知半解的去忽悠面试官了，面试官如果恰好也只会这几句话，那便达成和解了。

本篇文章从C语言源码底层来聊聊Python内存管理和垃圾回收机制到底是个啥？让你能够真正了解内存管理&垃圾回收。

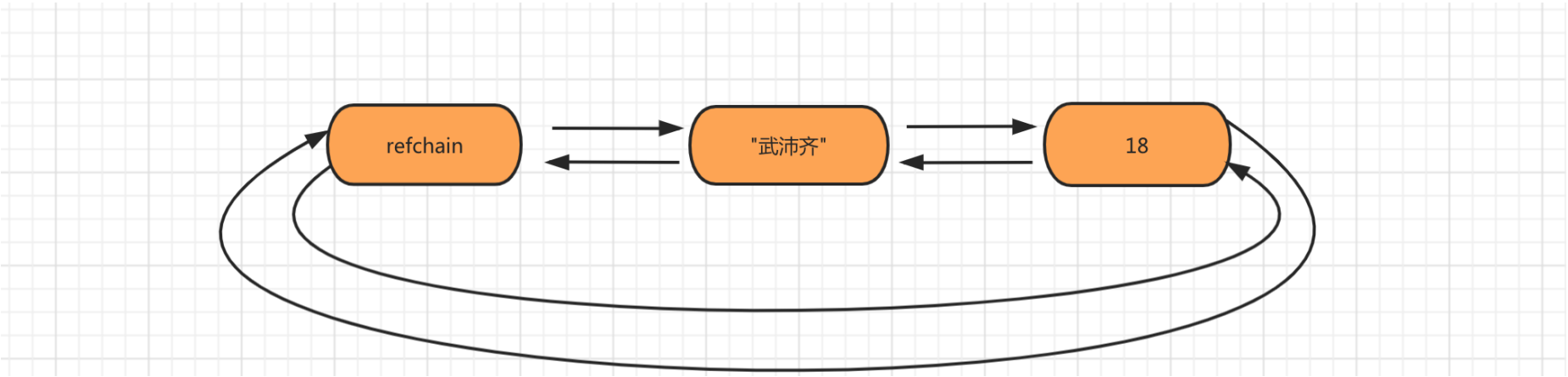
1. 白话垃圾回收

用通俗的语言解释内存管理和垃圾回收的过程，搞懂这一部分就可以去面试、去装逼了...

1.1 大管家refchain

在Python的C源码中有一个名为refchain的`环状双向链表`，这个链表比较牛逼了，因为Python程序中一旦创建对象都会把这个对象添加到refchain这个链表中。也就是说他保存着所有的对象。例如：

```
1. age = 18
2. name = "武沛齐"
```

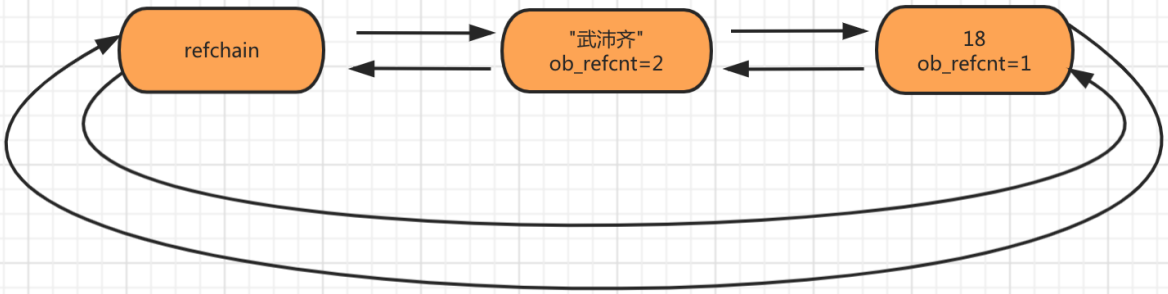


1.2 引用计数器

在refchain中的所有对象内部都有一个 `ob_refcnt` 用来保存当前对象的引用计数器，顾名思义就是自己被引用的次数，例如：

```
1. age = 18
2. name = "武沛齐"
3. nickname = name
```

上述代码表示内存中有 18 和 “武沛齐” 两个值，他们的引用计数器分别为：1、2 。



当值被多次引用时候，不会在内存中重复创建数据，而是 `引用计数器+1` 。 当对象被销毁时候同时会让 `引用计数器-1` ,如果引用计数器为0，则将对象从refchain链表中摘除，同时在内存中进行销毁（暂不考虑缓存等特殊情况）。

```
age = 18
number = age  # 对象18的引用计数器 + 1
del age           # 对象18的引用计数器 - 1

def run(arg):
    print(arg)

run(number)   # 刚开始执行函数时，对象18引用计数器 + 1，当函数执行完毕之后，对象18引用计数器 - 1 。

num_list = [11,22,number] # 对象18的引用计数器 + 1
```

1.3 标记清除&分代回收

基于引用计数器进行垃圾回收非常方便和简单，但他还是存在 `循环引用` 的问题，导致无法正常的回收一些数据，例如：

```
v1 = [11,22,33]        # refchain中创建一个列表对象，由于v1=对象，所以列表引对象用计数器为1.
v2 = [44,55,66]        # refchain中再创建一个列表对象，因v2=对象，所以列表对象引用计数器为1.
v1.append(v2)        # 把v2追加到v1中，则v2对应的[44,55,66]对象的引用计数器加1，最终为2.
v2.append(v1)        # 把v1追加到v1中，则v1对应的[11,22,33]对象的引用计数器加1，最终为2.

del v1    # 引用计数器-1
del v2    # 引用计数器-1
```

对于上述代码会发现，执行 `del` 操作之后，没有变量再会去使用那两个列表对象，但由于循环引用的问题，他们的引用计数器不为0，所以他们的状态：永远不会被使用、也不会被销毁。项目中如果这种代码太多，就会导致内存一直被消耗，直到内存被耗尽，程序崩溃。

为了解决循环引用的问题，引入了 `标记清除` 技术，专门针对那些可能存在循环引用的对象进行特殊处理，可能存在循环应用的类型有：列表、元组、字典、集合、自定义类等那些能进行数据嵌套的类型。

标记清除：创建特殊链表专门用于保存 列表、元组、字典、集合、自定义类等对象，之后再去检查这个链表中的对象是否存在循环引用，如果存在则让双方的引用计数器均 - 1 。

分代回收：对标记清除中的链表进行优化，将那些可能存在循引用的对象拆分到3个链表，链表称为：0/1/2三代，每代都可以存储对象和阈值，当达到阈值时，就会对相应的链表中的每个对象做一次扫描，除循环引用各自减1并且销毁引用计数器为0的对象。

```
1. // 分代的C源码
2. #define NUM_GENERATIONS 3
3. struct gc_generation generations[NUM_GENERATIONS] = {
4.     /* PyGC_Head,                                    threshold,    count */
5.     {{(uintptr_t)_GEN_HEAD(0), (uintptr_t)_GEN_HEAD(0)},   700,        0}, // 0代
6.     {{(uintptr_t)_GEN_HEAD(1), (uintptr_t)_GEN_HEAD(1)},   10,         0}, // 1代
7.     {{(uintptr_t)_GEN_HEAD(2), (uintptr_t)_GEN_HEAD(2)},   10,         0}, // 2代
8. };
```

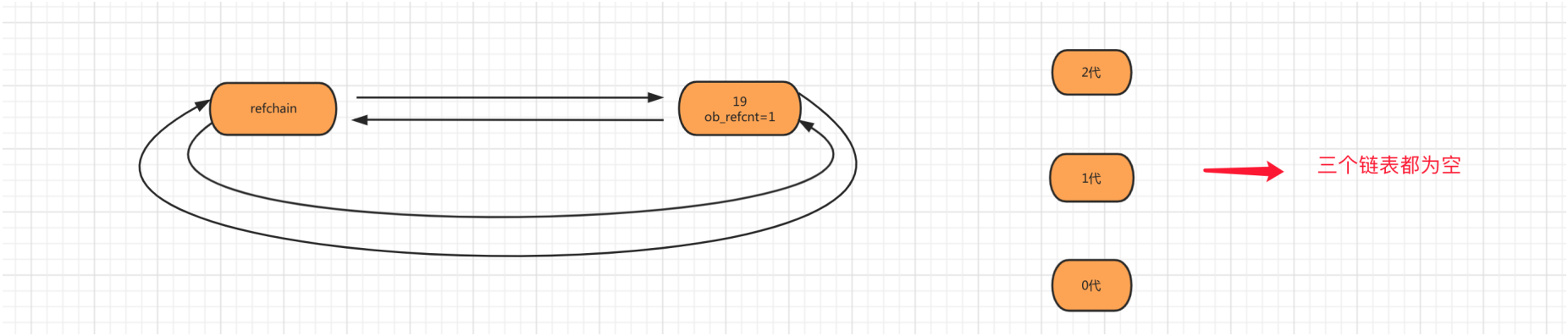
特别注意：0代和1、2代的threshold和count表示的意义不同。

- 0代，count表示0代链表中对象的数量，threshold表示0代链表对象个数阈值，超过则执行一次0代扫描检查。
- 1代，count表示0代链表扫描的次数，threshold表示0代链表扫描的次数阈值，超过则执行一次1代扫描检查。
- 2代，count表示1代链表扫描的次数，threshold表示1代链表扫描的次数阈值，超过则执行一2代扫描检查。

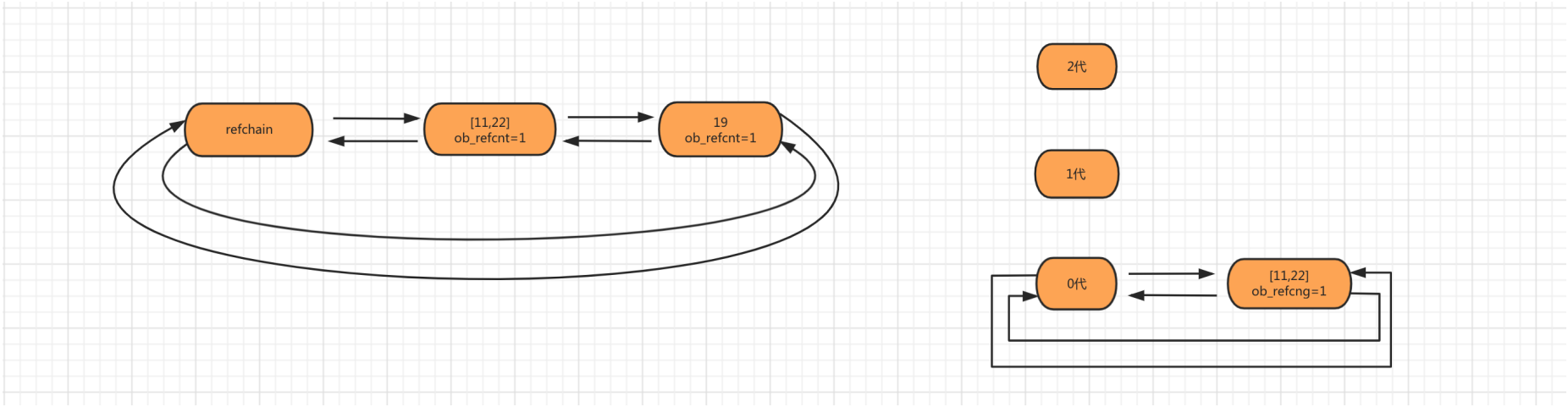
1.4 情景模拟

根据C语言底层并结合图来讲解内存管理和垃圾回收的详细过程。

第一步：当创建对象 `age=19` 时，会将对象添加到refchain链表中。



第二步：当创建对象 `num_list = [11,22]` 时，会将列表对象添加到 refchain 和 generations 0代中。



第三步：新创建对象使generations的0代链表上的对象数量大于阈值700时，要对链表上的对象进行扫描检查。

当0代大于阈值后，底层不是直接扫描0代，而是先判断2、1是否也超过了阈值。

- 如果2、1代未达到阈值，则扫描0代，并让1代的 count + 1 。
- 如果2代已达到阈值，则将2、1、0三个链表拼接起来进行全扫描，并将2、1、0代的count重置为0.
- 如果1代已达到阈值，则讲1、0两个链表拼接起来进行扫描，并将所有1、0代的count重置为0.

对拼接起来的链表在进行扫描时，主要就是删除循环引用和销毁垃圾，详细过程为：

对所有起来的链表在进行扫描时，主要就是删除循环引用相销毁垃圾，详细过程为：

- 扫描链表，把每个对象的引用计数器拷贝一份并保存到 `gc_refs` 中，保护原引用计数器。
- 再次扫描链表中的每个对象，并检查是否存在循环引用，如果存在则让各自的 `gc_refs` 减 1 。
- 再次扫描链表，将 `gc_refs` 为 0 的对象移动到 `unreachable` 链表中；不为0的对象直接升级到下一代链表中。
- 处理 `unreachable` 链表中的对象的 析构函数 和 弱引用，不能被销毁的对象升级到下一代链表，能销毁的保留在此链表。
  - 析构函数，指的就是那些定义了 `__del__` 方法的对象，需要执行之后再进行销毁处理。
  - 弱引用，
- 最后将 `unreachable` 中的每个对象销毁并在refchain链表中移除（不考虑缓存机制）。

至此，垃圾回收的过程结束。

1.5 缓存机制

从上文大家可以了解到当对象的引用计数器为0时，就会被销毁并释放内存。而实际上他不是这么的简单粗暴，因为反复的创建和销毁会使程序的执行效率变低。Python中引入了“缓存机制”机制。
例如：引用计数器为0时，不会真正销毁对象，而是将他放到一个名为 `free_list` 的链表中，之后会再创建对象时不会在重新开辟内存，而是在free_list中将之前的对象来并重置内部的值来使用。

- float类型，维护的free_list链表最多可缓存100个float对象。

```
v1 = 3.14    # 开辟内存来存储float对象，并将对象添加到refchain链表。
print( id(v1) ) # 内存地址：4436033488
del v1    # 引用计数器-1，如果为0则在rechain链表中移除，不销毁对象，而是将对象添加到float的free_list.
v2 = 9.999    # 优先去free_list中获取对象，并重置为9.999，如果free_list为空才重新开辟内存。
print( id(v2) ) # 内存地址：4436033488

# 注意：引用计数器为0时，会先判断free_list中缓存个数是否满了，未满则将对象缓存，已满则直接将对象销毁。
```

- int类型，不是基于free_list，而是维护一个small_ints链表保存常见数据（小数据池），小数据池范围： `-5 <= value < 257` 。即：重复使用这个范围的整数时，不会重新开辟内存。

```
v1 = 38    # 去小数据池small_ints中获取38整数对象，将对象添加到refchain并让引用计数器+1。
print( id(v1))  #内存地址：4514343712
v2 = 38 # 去小数据池small_ints中获取38整数对象，将refchain中的对象的引用计数器+1。
print( id(v2) ) #内存地址：4514343712

# 注意：在解释器启动时候-5~256就已经被加入到small_ints链表中且引用计数器初始化为1，代码中使用的值时直接去small_ints中拿来用并将引用计数器+1即可。另外，small_ints中的数据引用计数器永远不会为0（初始化时就设置为1了），所以也不会被销毁。
```

- str类型，维护 `unicode_latin1[256]` 链表，内部将所有的 `ascii字符` 缓存起来，以后使用时就不再反复创建。

```
v1 = "A"
print( id(v1) ) # 输出：4517720496
del v1
v2 = "A"
print( id(v1) ) # 输出：4517720496

# 除此之外，Python内部还对字符串做了驻留机制，针对那么只含有字母、数字、下划线的字符串（见源码Objects/codeobject.c），如果内存中已存在则不会重新在创建而是使用原来的地址里（不会像free_list那样一直在内存存活，只有内存中有才能被重复利用）。
v1 = "wupeiqi"
v2 = "wupeiqi"
print(id(v1) == id(v2)) # 输出：True
```

- list类型，维护的free_list数组最多可缓存80个list对象。

```
v1 = [11,22,33]
print( id(v1) ) # 输出：4517628816
del v1
v2 = ["武","沛齐"]
print( id(v2) ) # 输出：4517628816
```

- tuple类型，维护一个free_list数组且数组容量20，数组中元素可以是链表且每个链表最多可以容纳2000个元组对象。元组的free_list数组在存储数据时，是按照元组可以容纳的个数为索引找到free_list数组中对应的链表，并添加到链表中。

```
v1 = (1,2)
print( id(v1) )
del v1  # 因元组的数量为2，所以会把这个对象缓存到free_list[2]的链表中。
v2 = ("武沛齐","Alex")  # 不会重新开辟内存，而是去free_list[2]对应的链表中拿到一个对象来使用。
print( id(v2) )
```

- dict类型，维护的free_list数组最多可缓存80个dict对象。

```
v1 = {"k1":123}
print( id(v1) )  # 输出：4515998128
del v1
v2 = {"name":"武沛齐","age":18,"gender":"男"}
print( id(v1) ) # 输出：4515998128
```

2. C语言源码分析

上文对Python的内存管理和垃圾回收进行了快速讲解，基本上已可以让你拿去装逼了。

接下来这一部分会让你更超神，我们要再在源码中来证实上文的内容。

2.1 两个重要的结构体

```
#define PyObject_HEAD       PyObject ob_base;
#define PyObject_VAR_HEAD      PyVarObject ob_base;

// 宏定义，包含 上一个、下一个，用于构造双向链表用。(放到refchain链表中时，要用到)
#define _PyObject_HEAD_EXTRA            \
    struct _object *_ob_next;           \
    struct _object *_ob_prev;

typedef struct _object {
    _PyObject_HEAD_EXTRA // 用于构造双向链表
    Py_ssize_t ob_refcnt;  // 引用计数器
    struct _typeobject *ob_type;    // 数据类型
} PyObject;

typedef struct {
    PyObject ob_base;   // PyObject对象
    Py_ssize_t ob_size; /* Number of items in variable part，即：元素个数 */
} PyVarObject;
```

这两个结构体 `PyObject` 和 `PyVarObject` 是基石，他们保存这其他数据类型公共部分，例如：每个类型的对象在创建时都有PyObject中的那4部分数据；list/set/tuple等由多个元素组成对象创建时都有PyVarObject中的那5部分数据。

2.2 常见类型结构体

平时我们在创建一个对象时，本质上就是实例化一个相关类型的结构体，在内部保存值和引用计数器等。

- float类型

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

- int类型

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};
/* Long (arbitrary precision) integer object interface */
typedef struct _longobject PyLongObject; /* Revealed in longintrepr.h */
```

- str类型

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t length;          /* Number of code points in the string */
    Py_hash_t hash;             /* Hash value; -1 if not set */
    struct {
        unsigned int interned:2;
        /* Character size:

       - PyUnicode_WCHAR_KIND (0):

         * character type = wchar_t (16 or 32 bits, depending on the
           platform)

       - PyUnicode_1BYTE_KIND (1):

         * character type = Py_UCS1 (8 bits, unsigned)
         * all characters are in the range U+0000-U+00FF (latin1)
         * if ascii is set, all characters are in the range U+0000-U+007F
           (ASCII), otherwise at least one character is in the range
           U+0080-U+00FF

       - PyUnicode_2BYTE_KIND (2):

         * character type = Py_UCS2 (16 bits, unsigned)
         * all characters are in the range U+0000-U+FFFF (BMP)
         * at least one character is in the range U+0100-U+FFFF

       - PyUnicode_4BYTE_KIND (4):
```

```
29.
30.             * character type = Py_UCS4 (32 bits, unsigned)
31.             * all characters are in the range U+0000-U+10FFFF
32.             * at least one character is in the range U+10000-U+10FFFF
33.            */
34.             unsigned int kind:3;
35.             unsigned int compact:1;
36.             unsigned int ascii:1;
37.             unsigned int ready:1;
38.             unsigned int :24;
39.         } state;
40.         wchar_t *wstr;              /* wchar_t representation (null-terminated) */
41.     } PyASCIIObject;
42.
43.     typedef struct {
44.         PyASCIIObject _base;
45.         Py_ssize_t utf8_length;     /* Number of bytes in utf8, excluding the
46.                                      * terminating \0. */
47.         char *utf8;                 /* UTF-8 representation (null-terminated) */
48.         Py_ssize_t wstr_length;     /* Number of code points in wstr, possible
49.                                      * surrogates count as two code points. */
50.     } PyCompactUnicodeObject;
51.
52.     typedef struct {
53.         PyCompactUnicodeObject _base;
54.         union {
55.             void *any;
56.             Py_UCS1 *latin1;
57.             Py_UCS2 *ucs2;
58.             Py_UCS4 *ucs4;
59.         } data;                     /* Canonical, smallest-form Unicode buffer */
```

```
60.   } PyUnicodeObject;
```

- list类型

```
1.   typedef struct {
2.       PyObject_VAR_HEAD
3.       PyObject **ob_item;
4.       Py_ssize_t allocated;
5.   } PyListObject;
```

- tuple类型

```
1.   typedef struct {
2.       PyObject_VAR_HEAD
3.       PyObject *ob_item[1];
4.   } PyTupleObject;
```

- dict类型

```
1.   typedef struct {
2.       PyObject_HEAD
3.       Py_ssize_t ma_used;
4.       PyDictKeysObject *ma_keys;
5.       PyObject **ma_values;
6.   } PyDictObject;
```

通过常见结构体可以基本了解到本质上每个对象内部会存储的数据。

扩展：在结构体部分你应该发现了 `str类型` 比较繁琐，那是因为python字符串在处理时需要考虑到编码的问题，在内部规定（见源码结构体）：

- 字符串只包含ascii，则每个字符用1个字节表示，即：latin1

- 字符串包含中文等，则每个字符用2个字节表示，即：ucs2
- 字符串包含emoji等，则每个字符用4个字节表示，即：ucs4

```
wupeiqi:~ wupeiqi$ python3.8
Python 3.8.2 (v3.8.2:7b3ab5921f, Feb 24 2020, 17:52:18)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> v1 = "a"
>>> sys.getsizeof(v1)
50                          统一用1个字节表示1个字符
>>> v2 = "ab"
>>> sys.getsizeof(v2)
51
>>>
>>> v3 = "武"
>>> sys.getsizeof(v3)
76
>>> v4 = "武p"               统一用2个字节表示1个字符
>>> sys.getsizeof(v4)
78
>>>
>>> v5 = "😘"
>>> sys.getsizeof(v5)
80
>>> v6 = "😘u"               统一用4个字节表示1个字符
>>> sys.getsizeof(v6)
84
>>>
```

2.3 Float类型

2.3.1 创建

```
1. val = 3.14
```

类似于这样创建一个float对象时，会执行C源码中的如下代码：

```c
// Objects/floatobject.c

// 用于缓存float对象的链表
static PyFloatObject *free_list = NULL;
static int numfree = 0;

PyObject *
PyFloat_FromDouble(double fval)
{
    // 如果free_list中有可用对象，则从free_list链表拿出来一个；否则为对象重新开辟内存。
    PyFloatObject *op = free_list;
    if (op != NULL) {
        free_list = (PyFloatObject *) Py_TYPE(op);
        numfree--;
    } else {
        // 根据float类型的大小，为float对象新开辟内存。
        op = (PyFloatObject*) PyObject_MALLOC(sizeof(PyFloatObject));
        if (!op)
            return PyErr_NoMemory();
    }
    // 对float对象进行初始化，例如：引用计数器初始化为1、添加到refchain链表等。
    /* Inline PyObject_New */
    (void)PyObject_INIT(op, &PyFloat_Type);

    // 对float对象赋值。即：op->ob_fval = 3.14
    op->ob_fval = fval;
    return (PyObject *) op;
}
```

```
// Include/objimpl.h

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

```c
// Objects/object.c

// 维护了所有对象的一个环状双向链表
static PyObject refchain = {&refchain, &refchain};


void
_Py_AddToAllObjects(PyObject *op, int force)
{

    if (force || op->_ob_prev == NULL) {
        op->_ob_next = refchain._ob_next;
        op->_ob_prev = &refchain;
        refchain._ob_next->_ob_prev = op;
        refchain._ob_next = op;
    }
}

void
_Py_NewReference(PyObject *op)
{
    _Py_INC_REFTOTAL;

    // 引用计数器初始化为1。
    op->ob_refcnt = 1;

    // 对象添加到双向链表refchain中。
    _Py_AddToAllObjects(op, 1);
```

```
29.
30.     _Py_INC_TPALLOCS(op);
31. }
```

2.3.2 引用

```
1. val = 3.14
2. data = val
```

在项目中如果出现这种引用关系时，会将原对象的引用计数器+1。
C源码执行流程如下：

```
1. // Include/object.h
2.
3. static inline void _Py_INCREF(PyObject *op)
4. {
5.     _Py_INC_REFTOTAL;
6.     // 对象的引用计数器 + 1
7.     op->ob_refcnt++;
8. }
9.
10. #define Py_INCREF(op) _Py_INCREF(_PyObject_CAST(op))
```

2.3.3 销毁

```
1. val = 3.14
2. del val
```

在项目中如果出现这种删除的语句，则内部会将引用计数器-1，如果引用计数器减为0，则进行缓存或垃圾回收。
C源码执行流程如下：

C源码执行流程如下：

```
// Include/object.h

static inline void _Py_DECREF(const char *filename, int lineno,
                              PyObject *op)
{
    (void)filename; /* may be unused, shut up -Wunused-parameter */
    (void)lineno; /* may be unused, shut up -Wunused-parameter */
    _Py_DEC_REFTOTAL;
    // 引用计数器-1，如果引用计数器为0，则执行 _Py_Dealloc去缓存或垃圾回收。
    if (--op->ob_refcnt != 0) {
#ifdef Py_REF_DEBUG
        if (op->ob_refcnt < 0) {
            _Py_NegativeRefcount(filename, lineno, op);
        }
#endif
    }
    else {
        _Py_Dealloc(op);
    }
}

#define Py_DECREF(op) _Py_DECREF(__FILE__, __LINE__, _PyObject_CAST(op))
```

```
// Objects/object.c

void
_Py_Dealloc(PyObject *op)
{
    // 找到float类型的 tp_dealloc 函数
    destructor dealloc = Py_TYPE(op)->tp_dealloc;

    // 在refchain双向链表中摘除此对象。
    _Py_ForgetReference(op);

    // 执行float类型的 tp_dealloc 函数，去进行缓存或垃圾回收。
    (*dealloc)(op);
}

void
_Py_ForgetReference(PyObject *op)
{
    ...
    // 在refchain链表中移除此对象
    op->_ob_next->_ob_prev = op->_ob_prev;
    op->_ob_prev->_ob_next = op->_ob_next;
    op->_ob_next = op->_ob_prev = NULL;
    _Py_INC_TPFREES(op);
}
```

```c
// Objects/floatobject.c

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

// float类型中函数的对应关系
PyTypeObject PyFloat_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "float",
    sizeof(PyFloatObject),
    0,
    // tp_dealloc表示执行float_dealloc方法
    (destructor)float_dealloc,                  /* tp_dealloc */
    0,                                          /* tp_print */
    ...
};

static void
float_dealloc(PyFloatObject *op)
{
    // 检测是否是float类型
    if (PyFloat_CheckExact(op)) {

        // 检测free_list中缓存的个数是否已满，如果已满，则直接将对象销毁。
        if (numfree >= PyFloat_MAXFREELIST)  {
            // 销毁
            PyObject_FREE(op);
```

```
            return;
        }
        // 将对象加入到free_list链表中
        numfree++;
        Py_TYPE(op) = (struct _typeobject *)free_list;
        free_list = op;
    }
    else
        Py_TYPE(op)->tp_free((PyObject *)op);
}
```

2.4 int类型

2.4.1 创建

```
age = 19
```

当在python中创建一个整型数据时，底层会触发他的如下源码:

```
PyObject *
PyLong_FromLong(long ival)
{
    PyLongObject *v;
    ...
    // 优先去小数据池中检查，如果在范围内则直接获取不再重新开辟内存。（ -5 <= value < 257 ）
    CHECK_SMALL_INT(ival);
    ...
    // 非小数字池中的值，重新开辟内存并初始化
    v = _PyLong_New(ndigits);
    if (v != NULL) {
        digit *p = v->ob_digit;
        Py_SIZE(v) = ndigits*sign;
        t = abs_ival;
        ...
    }
    return (PyObject *)v;
}


#define NSMALLNEGINTS           5
#define NSMALLPOSINTS           257
#define CHECK_SMALL_INT(ival) \
    do if (-NSMALLNEGINTS <= ival && ival < NSMALLPOSINTS) { \
        return get_small_int((sdigit)ival); \
    } while(0)

static PyObject *
```

```
get_small_int(sdigit ival)
{
    PyObject *v;
    v = (PyObject *)&small_ints[ival + NSMALLNEGINTS];
    // 引用计数器 + 1
    Py_INCREF(v);
    ...
    return v;
}


PyLongObject *
_PyLong_New(Py_ssize_t size)
{
    // 创建PyLongObject的指针变量
    PyLongObject *result;
    ...
    // 根据长度进行开辟内存
    result = PyObject_MALLOC(offsetof(PyLongObject, ob_digit) +
                             size*sizeof(digit));
    ...
    // 对内存中的数据进行初始化并添加到refchain链表中。
    return (PyLongObject*)PyObject_INIT_VAR(result, &PyLong_Type, size);
}
```

```
// Include/objimpl.h

#define PyObject_NewVar(type, typeobj, n) \
                ( (type *) _PyObject_NewVar((typeobj), (n)) )

static inline PyVarObject*
_PyObject_INIT_VAR(PyVarObject *op, PyTypeObject *typeobj, Py_ssize_t size)
{
    assert(op != NULL);
    Py_SIZE(op) = size;
    // 对象初始化
    PyObject_INIT((PyObject *)op, typeobj);
    return op;
}

#define PyObject_INIT(op, typeobj) \
    _PyObject_INIT(_PyObject_CAST(op), (typeobj))

static inline PyObject*
_PyObject_INIT(PyObject *op, PyTypeObject *typeobj)
{
    assert(op != NULL);
    Py_TYPE(op) = typeobj;
    if (PyType_GetFlags(typeobj) & Py_TPFLAGS_HEAPTYPE) {
        Py_INCREF(typeobj);
    }
    // 对象初始化，并把对象加入到refchain链表。
    _Py_NewReference(op);
```

```
29.     return op;
30. }
```

```c
// Objects/object.c

// 维护了所有对象的一个环状双向链表
static PyObject refchain = {&refchain, &refchain};


void
_Py_AddToAllObjects(PyObject *op, int force)
{

    if (force || op->_ob_prev == NULL) {
        op->_ob_next = refchain._ob_next;
        op->_ob_prev = &refchain;
        refchain._ob_next->_ob_prev = op;
        refchain._ob_next = op;
    }
}

void
_Py_NewReference(PyObject *op)
{
    _Py_INC_REFTOTAL;

    // 引用计数器初始化为1。
    op->ob_refcnt = 1;

    // 对象添加到双向链表refchain中。
    _Py_AddToAllObjects(op, 1);
```

```
29.
30.     _Py_INC_TPALLOCS(op);
31. }
```

2.4.2 引用

```
1. value = 69
2. data = value
```

类似于出现这种引用关系时，内部其实就是将对象的引用计数器+1，源码同float类型引用。

2.4.3 销毁

```
1. value = 699
2. del value
```

在项目中如果出现这种删除的语句，则内部会将引用计数器-1，如果引用计数器减为0，则直接进行垃圾回收。（int类型是基于小数据池而不是free_list做的缓存，所以不会在销毁时缓存数据）。
C源码执行流程如下：

```c
// Include/object.h

static inline void _Py_DECREF(const char *filename, int lineno,
                              PyObject *op)
{
    (void)filename; /* may be unused, shut up -Wunused-parameter */
    (void)lineno; /* may be unused, shut up -Wunused-parameter */
    _Py_DEC_REFTOTAL;
    // 引用计数器-1，如果引用计数器为0，则执行 _Py_Dealloc去垃圾回收。
    if (--op->ob_refcnt != 0) {
#ifdef Py_REF_DEBUG
        if (op->ob_refcnt < 0) {
            _Py_NegativeRefcount(filename, lineno, op);
        }
#endif
    }
    else {
        _Py_Dealloc(op);
    }
}
#define Py_DECREF(op) _Py_DECREF(__FILE__, __LINE__, _PyObject_CAST(op))
```

```c
// Objects/object.c
void
_Py_Dealloc(PyObject *op)
{
    // 找到int类型的 tp_dealloc 函数（int类中没有定义tp_dealloc函数，需要去父级PyBaseObject_Type中找tp_dealloc函数）
    // 此处体现所有的类型都继承object
    destructor dealloc = Py_TYPE(op)->tp_dealloc;

    // 在refchain双向链表中摘除此对象。
    _Py_ForgetReference(op);

    // 执行int类型的 tp_dealloc 函数，去进行垃圾回收。
    (*dealloc)(op);
}
void
_Py_ForgetReference(PyObject *op)
{
    ...
    // 在refchain链表中移除此对象
    op->_ob_next->_ob_prev = op->_ob_prev;
    op->_ob_prev->_ob_next = op->_ob_next;
    op->_ob_next = op->_ob_prev = NULL;
    _Py_INC_TPFREES(op);
}
```

```c
// Objects/longobjet.c

PyTypeObject PyLong_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "int",                                      /* tp_name */
    offsetof(PyLongObject, ob_digit),           /* tp_basicsize */
    sizeof(digit),                              /* tp_itemsize */
    0,                                          /* tp_dealloc */
      ...
    PyObject_Del,                               /* tp_free */
};
```

```
Objects/typeobject.c

PyTypeObject PyBaseObject_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "object",                                   /* tp_name */
    sizeof(PyObject),                           /* tp_basicsize */
    0,                                          /* tp_itemsize */
    object_dealloc,                             /* tp_dealloc */
    ...
    PyObject_Del,                               /* tp_free */
};

static void
object_dealloc(PyObject *self)
{
    // 调用int类型的 tp_free，即：PyObject_Del去销毁对象。
    Py_TYPE(self)->tp_free(self);
}
```

2.5 str类型

2.5.1 创建

```
name = "武沛齐"
```

当在python中创建一个字符串数据时，底层会触发他的如下源码:

```
Objects/unicodeobject.c

PyObject *
PyUnicode_DecodeUTF8Stateful(const char *s,Py_ssize_t size,const char *errors,Py_ssize_t *consumed)
{
    return unicode_decode_utf8(s, size, _Py_ERROR_UNKNOWN, errors, consumed);
}

static PyObject *
unicode_decode_utf8(const char *s, Py_ssize_t size,_Py_error_handler error_handler, const char *errors,Py_ssize_t *consumed);
{
    ...
    // 如果字符串长度为1，并且是ascii字符，直接去缓存链表 *unicode_latin1[256] 中获取。
    if (size == 1 && (unsigned char)s[0] < 128) {
        if (consumed)
            *consumed = 1;
        return get_latin1_char((unsigned char)s[0]);
    }
    // 对传入的utf-8的字节进行处理，并选择合适的方式转换成unicode字符串。（latin2/ucs2/ucs4）。
    ...
    return _PyUnicodeWriter_Finish(&writer);
}

static PyObject*
get_latin1_char(unsigned char ch)
{
    PyObject *unicode = unicode_latin1[ch];
```

```
28.     if (!unicode) {
29.         unicode = PyUnicode_New(1, ch);
30.         if (!unicode)
31.             return NULL;
32.         PyUnicode_1BYTE_DATA(unicode)[0] = ch;
33.         assert(_PyUnicode_CheckConsistency(unicode, 1));
34.         unicode_latin1[ch] = unicode;
35.     }
36.     Py_INCREF(unicode);
37.     return unicode;
38. }
39. 
40. PyObject *
41. _PyUnicodeWriter_Finish(_PyUnicodeWriter *writer)
42. {
43.     PyObject *str;
44.     // 写入值到str
45.     str = writer->buffer;
46.     writer->buffer = NULL;
47. 
48.     if (writer->readonly) {
49.         assert(PyUnicode_GET_LENGTH(str) == writer->pos);
50.         return str;
51.     }
52. 
53.     if (PyUnicode_GET_LENGTH(str) != writer->pos) {
54.         PyObject *str2;
55.         // 创建对象
56.         str2 = resize_compact(str, writer->pos);
57.         if (str2 == NULL) {
58.             Py_DECREF(str);
```

```
59.             return NULL;

60.         }
61.         str = str2;
62.     }
63.
64.     assert(_PyUnicode_CheckConsistency(str, 1));
65.     return unicode_result_ready(str);
66. }
67.
68. static PyObject*
69. resize_compact(PyObject *unicode, Py_ssize_t length)
70. {
71.     ...
72.     // 开辟内存
73.     new_unicode = (PyObject *)PyObject_REALLOC(unicode, new_size);
74.     if (new_unicode == NULL) {
75.         _Py_NewReference(unicode);
76.         PyErr_NoMemory();
77.         return NULL;
78.     }
79.     unicode = new_unicode;
80.     // 把对象加入到refchain链表
81.     _Py_NewReference(unicode);
82.     ...
83.     return unicode;
84. }
```

在字符串中除了会执行上述代码之外，还会执行以下代码实现内部的驻留机制。为了更好的理解，你可以认为驻留机制：将字符串保存到一个名为 interned 的字典中，以后再使用时 直接去字典中获取不再需要创建。

实际在源码中每次都会创建新的字符串，只不过在内部检测是否已驻留到interned中，如果在则使用interned内部的原来的字符串，把新创建的字符串当做垃圾去回收。

```
Objects/unicodeobject.c
void
PyUnicode_InternInPlace(PyObject **p)
{
    PyObject *s = *p;
    PyObject *t;
#ifdef Py_DEBUG
    assert(s != NULL);
    assert(_PyUnicode_CHECK(s));
#else
    if (s == NULL || !PyUnicode_Check(s))
        return;
#endif
    /* If it's a subclass, we don't really know what putting
       it in the interned dict might do. */
    if (!PyUnicode_CheckExact(s))
        return;
    if (PyUnicode_CHECK_INTERNED(s))
        return;
    if (interned == NULL) {
        interned = PyDict_New();
        if (interned == NULL) {
            PyErr_Clear(); /* Don't leave an exception */
            return;
        }
    }
    Py_ALLOW_RECURSION
    // 将新字符串驻留到interned字典中，不存在则驻留，已存在则不再重复驻留。
```

```
    t = PyDict_SetDefault(interned, s, s);
    Py_END_ALLOW_RECURSION
    if (t == NULL) {
        PyErr_Clear();
        return;
    }
    // 存在，使用已驻留的字符串 并 将引用计数器+1
    if (t != s) {
        Py_INCREF(t);
        Py_SETREF(*p, t); // 处理临时对象
        return;
    }
    /* The two references in interned are not counted by refcnt.
       The deallocator will take care of this */
    Py_REFCNT(s) -= 2; // 让临时对象可被回收。
    _PyUnicode_STATE(s).interned = SSTATE_INTERNED_MORTAL;
}
```

2.5.2 引用

同上，引用计数器 + 1 .

2.5.3 销毁

```
val = "武沛齐"
del val
```

在项目中如果出现这种删除的语句，则内部会将引用计数器-1，如果引用计数器减为0，则进行缓存或垃圾回收。

```
// Include/object.h
static inline void _Py_DECREF(const char *filename, int lineno,
                              PyObject *op)
{
    (void)filename; /* may be unused, shut up -Wunused-parameter */
    (void)lineno; /* may be unused, shut up -Wunused-parameter */
    _Py_DEC_REFTOTAL;
    // 引用计数器-1，如果引用计数器为0，则执行 _Py_Dealloc去缓存或垃圾回收。
    if (--op->ob_refcnt != 0) {
#ifdef Py_REF_DEBUG
        if (op->ob_refcnt < 0) {
            _Py_NegativeRefcount(filename, lineno, op);
        }
#endif
    }
    else {
        _Py_Dealloc(op);
    }
}
#define Py_DECREF(op) _Py_DECREF(__FILE__, __LINE__, _PyObject_CAST(op))
```

```c
// Objects/object.c
void
_Py_Dealloc(PyObject *op)
{
    // 找到str类型的 tp_dealloc 函数
    destructor dealloc = Py_TYPE(op)->tp_dealloc;
    // 在refchain双向链表中摘除此对象。
    _Py_ForgetReference(op);
    // 执行float类型的 tp_dealloc 函数，去进行缓存或垃圾回收。
    (*dealloc)(op);
}
void
_Py_ForgetReference(PyObject *op)
{
    ...
    // 在refchain链表中移除此对象
    op->_ob_next->_ob_prev = op->_ob_prev;
    op->_ob_prev->_ob_next = op->_ob_next;
    op->_ob_next = op->_ob_prev = NULL;
    _Py_INC_TPFREES(op);
}
```

```
// Objects/unicodeobject.c

PyTypeObject PyUnicode_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "str",                          /* tp_name */
    sizeof(PyUnicodeObject),        /* tp_basicsize */
    0,                              /* tp_itemsize */
    /* Slots */
    (destructor)unicode_dealloc,  /* tp_dealloc */
       ...
    PyObject_Del,                   /* tp_free */
};

static void
unicode_dealloc(PyObject *unicode)
{
    switch (PyUnicode_CHECK_INTERNED(unicode)) {
    case SSTATE_NOT_INTERNED:
        break;

    case SSTATE_INTERNED_MORTAL:
        /* revive dead object temporarily for DelItem */
        Py_REFCNT(unicode) = 3;
        // 在interned中删除驻留的字符串
        if (PyDict_DelItem(interned, unicode) != 0)
            Py_FatalError(
                "deletion of interned string failed");
        break;
```

```
29.
30.     case SSTATE_INTERNED_IMMORTAL:
31.         Py_FatalError("Immortal interned string died.");
32.         /* fall through */
33.
34.     default:
35.         Py_FatalError("Inconsistent interned string state.");
36.     }
37.
38.     if (_PyUnicode_HAS_WSTR_MEMORY(unicode))
39.         PyObject_DEL(_PyUnicode_WSTR(unicode));
40.     if (_PyUnicode_HAS_UTF8_MEMORY(unicode))
41.         PyObject_DEL(_PyUnicode_UTF8(unicode));
42.     if (!PyUnicode_IS_COMPACT(unicode) && _PyUnicode_DATA_ANY(unicode))
43.         PyObject_DEL(_PyUnicode_DATA_ANY(unicode));
44.     // 内存中销毁对象
45.     Py_TYPE(unicode)->tp_free(unicode);
46. }
```

2.6 list类型

2.6.1 创建

```
1. v = [11,22,33]
```

当创建一个列表时候，内部的C源码会执行如下：

```c
// Objects/listobject.c

#define PyList_MAXFREELIST 80

// free_list用于对list对象进行缓存，最多可缓存80个对象
static PyListObject *free_list[PyList_MAXFREELIST];
// free_list中可用的对象
static int numfree = 0;

PyObject *
PyList_New(Py_ssize_t size)
{
    PyListObject *op;

    if (size < 0) {
        PyErr_BadInternalCall();
        return NULL;
    }
    if (numfree) {
        // 如果free_list中有缓存的对象，则直接从free_list中获取一个对象来使用。
        numfree--;
        op = free_list[numfree];
        _Py_NewReference((PyObject *)op);
    } else {
        // 缓存中没有，则需要 开辟内存 & 初始化对象
        op = PyObject_GC_New(PyListObject, &PyList_Type);
        if (op == NULL)
            return NULL;
```

```
29.     }
30.     if (size <= 0)
31.         op->ob_item = NULL;
32.     else {
33.         op->ob_item = (PyObject **) PyMem_Calloc(size, sizeof(PyObject *));
34.         if (op->ob_item == NULL) {
35.             Py_DECREF(op);
36.             return PyErr_NoMemory();
37.         }
38.     }
39.     Py_SIZE(op) = size;
40.     op->allocated = size;
41.     // 把对象加入到分代回收的三代中的0代链表中。
42.     _PyObject_GC_TRACK(op);
43.     return (PyObject *) op;
44. }
```

```
static inline void _PyObject_GC_TRACK_impl(const char *filename, int lineno,
                                           PyObject *op)
{
    _PyObject_ASSERT_FROM(op, !_PyObject_GC_IS_TRACKED(op),
                          "object already tracked by the garbage collector",
                          filename, lineno, "_PyObject_GC_TRACK");

    PyGC_Head *gc = _Py_AS_GC(op);
    _PyObject_ASSERT_FROM(op,
                          (gc->_gc_prev & _PyGC_PREV_MASK_COLLECTING) == 0,
                          "object is in generation which is garbage collected",
                          filename, lineno, "_PyObject_GC_TRACK");
    // 把对象加入到链表中，链表尾部还是gc.generation0。
    PyGC_Head *last = (PyGC_Head*)(_PyRuntime.gc.generation0->_gc_prev);
    _PyGCHead_SET_NEXT(last, gc);
    _PyGCHead_SET_PREV(gc, last);
    _PyGCHead_SET_NEXT(gc, _PyRuntime.gc.generation0);
    _PyRuntime.gc.generation0->_gc_prev = (uintptr_t)gc;
}

#define _PyObject_GC_TRACK(op) \
    _PyObject_GC_TRACK_impl(__FILE__, __LINE__, _PyObject_CAST(op))
```

```
Include/objimpl.h

#define PyObject_GC_New(type, typeobj) \
            ( (type *) _PyObject_GC_New(typeobj) )
```

```c
//Modules/gcmodule.c

PyObject *
 _PyObject_GC_New(PyTypeObject *tp)
{
    // 创建对象
    PyObject *op = _PyObject_GC_Malloc(_PyObject_SIZE(tp));
    if (op != NULL)
        // 初始化对象并把对象加入到refchain链表中。
        op = PyObject_INIT(op, tp);
    return op;
}

PyObject *
_PyObject_GC_Malloc(size_t basicsize)
{
   return _PyObject_GC_Alloc(0, basicsize);
}

static PyObject *
_PyObject_GC_Alloc(int use_calloc, size_t basicsize)
{
   // 包含分代回收的三代链表
   struct _gc_runtime_state *state = &_PyRuntime.gc;
   PyObject *op;
   PyGC_Head *g;
   size_t size;
   if (basicsize > PY_SSIZE_T_MAX - sizeof(PyGC_Head))
```

```
29.         return PyErr_NoMemory();
30.     size = sizeof(PyGC_Head) + basicsize;
31.
32.     // 创建 gc head
33.     if (use_calloc)
34.         g = (PyGC_Head *)PyObject_Calloc(1, size);
35.     else
36.         g = (PyGC_Head *)PyObject_Malloc(size);
37.     if (g == NULL)
38.         return PyErr_NoMemory();
39.     assert(((uintptr_t)g & 3) == 0);  // g must be aligned 4bytes boundary
40.     g->_gc_next = 0;
41.     g->_gc_prev = 0;
42.
43.     // 分代回收的0代数量+1
44.     state->generations[0].count++; /* number of allocated GC objects */
45.
46.     // 如果0代超出自己的阈值，进行垃圾分代回收。
47.     if (state->generations[0].count > state->generations[0].threshold && state->enabled && state->generati
    ons[0].threshold && !state->collecting && !PyErr_Occurred())
48.     {
49.         // 正在收集
50.         state->collecting = 1;
51.         // 去进行垃圾回收收集
52.         collect_generations(state);
53.         // 结束收集
54.         state->collecting = 0;
55.     }
56.     op = FROM_GC(g);
57.     return op;
58. }
```

```
59.
60. /* Get the object given the GC head */
61. #define FROM_GC(g) ((PyObject *)(((PyGC_Head *)g)+1))
62.
63. static Py_ssize_t
64. collect_generations(struct _gc_runtime_state *state)
65. {
66.     Py_ssize_t n = 0;
67.     // 倒序循环三代，按照：2、1、0顺序
68.     for (int i = NUM_GENERATIONS-1; i >= 0; i--) {
69.         if (state->generations[i].count > state->generations[i].threshold) {
70.             if (i == NUM_GENERATIONS - 1 && state->long_lived_pending < state->long_lived_total / 4)
71.                 continue;
72.               // 去进行回收，回收当前代之前的所有代。
73.             n = collect_with_callback(state, i);
74.             break;
75.         }
76.     }
77.     return n;
78. }
79.
80. static Py_ssize_t
81. collect_with_callback(struct _gc_runtime_state *state, int generation)
82. {
83.     ...
84.     // 回收，0、1、2代（通过引用传参获取 已回收的和未回收的链表）
85.     result = collect(state, generation, &collected, &uncollectable, 0);
86.     ...
87.     return result;
88. }
```

```
89.
90. /* This is the main function.  Read this to understand how the collection process works. */
91. static Py_ssize_t
92. collect(struct _gc_runtime_state *state, int generation,
93.         Py_ssize_t *n_collected, Py_ssize_t *n_uncollectable, int nofail)
94. {
95.     int i;
96.     Py_ssize_t m = 0; /* # objects collected */
97.     Py_ssize_t n = 0; /* # unreachable objects that couldn't be collected */
98.     PyGC_Head *young; /* the generation we are examining */
99.     PyGC_Head *old; /* next older generation */
100.    PyGC_Head unreachable; /* non-problematic unreachable trash */
101.    PyGC_Head finalizers;  /* objects with, & reachable from, __del__ */
102.    PyGC_Head *gc;
103.    _PyTime_t t1 = 0;   /* initialize to prevent a compiler warning */
104.
105.    /* update collection and allocation counters */
106.    // generation分别会是 0 1 2
107.    // 让当前执行收集的代的更高级的代的count加1 ？例如：0带时，让1代的count+1
108.    // 因为当前带扫描一次，则更高级代count+1，当前带扫描到10次时，更高级的带要扫描一次。
109.    if (generation+1 < NUM_GENERATIONS)
110.        state->generations[generation+1].count += 1;
111.
112.    // 比当前代低的代的count设置为0，因为当前带扫描时候会携带年轻带一起扫描，本次扫描之后对象都会升级到高级别的带，
    年轻代则为0
113.    for (i = 0; i <= generation; i++)
114.        state->generations[i].count = 0;
115.    // 总结：比当前扫描的代高的带count+1，自己和比自己低的代count设置为0.
116.
117.    // 将比自己代低的所有代，搞到一个链表中
```

```
    // 将比当前代低的所有代，搞到一个链表中
    // #define GEN_HEAD(state, n) (&(state)->generations[n].head)

    for (i = 0; i < generation; i++) {
        gc_list_merge(GEN_HEAD(state, i), GEN_HEAD(state, generation));
    }
    // 获取当前代的head（链表头）
    // #define GEN_HEAD(state, n) (&(state)->generations[n].head)
    young = GEN_HEAD(state, generation);

    // 比当前代老的head（链表头），如果是0、1、2中的2代时，则两个值相等。
    if (generation < NUM_GENERATIONS-1)
        //0、1代
        old = GEN_HEAD(state, generation+1);
    else
        //2代
        old = young;

    // 循环当前代（包含比自己年轻的代的链表）重的每个元素，将引用计数器拷贝到gc_refs中。
    // 拷贝出来的用于以后做计数器的计算，不回去更改原来的引用计数器的值。
    update_refs(young);  // gc_prev is used for gc_refs
    // 处理循环引用，把循环引用的位置值为0.
    subtract_refs(young);

    // 将链表中所有引用计数器为0的，移动到unreachable链表（不可达链表）。
    // 循环young链表中的每个元素，并根据拷贝的引用计数器gc_refs进行判断，如果为0则放入不可达链表；
    gc_list_init(&unreachable);
    move_unreachable(young, &unreachable);  // gc_prev is pointer again
    validate_list(young, 0);
    untrack_tuples(young);
    /* Move reachable objects to next generation. */
    // 将可达对象加入到下一代。
```

```
148.    if (young != old) {
149.        // 如果是0、1代，则升级到下一代。
150.        if (generation == NUM_GENERATIONS - 2) {
151.                // 如果是1代，则更新
152.              state->long_lived_pending += gc_list_size(young);
153.        }
154.        // 把young链表拼接到old链表中。
155.        gc_list_merge(young, old);
156.    }
157.    else {
158.        /* We only untrack dicts in full collections, to avoid quadratic
159.           dict build-up. See issue #14775. */
160.        // 如果是2代，则更新long_lived_total和long_lived_pending
161.        untrack_dicts(young);
162.        state->long_lived_pending = 0;
163.        state->long_lived_total = gc_list_size(young);
164.    }
165.
166.    // 循环所有不可达的元素，把具有 __del__ 方法对象放到finalizers链表中。
167.    // 调用__del__之后，再会进行让他们在销毁。
168.    gc_list_init(&finalizers);
169.    // NEXT_MASK_UNREACHABLE is cleared here.
170.    // After move_legacy_finalizers(), unreachable is normal list.
171.    move_legacy_finalizers(&unreachable, &finalizers);
172.    /* finalizers contains the unreachable objects with a legacy finalizer;
173.    * unreachable objects reachable *from* those are also uncollectable,
174.    * and we move those into the finalizers list too.
175.    */
176.    move_legacy_finalizer_reachable(&finalizers);
177.
```

```
178.    validate_list(&finalizers, 0);
179.    validate_list(&unreachable, PREV_MASK_COLLECTING);
180.     ...
181.
182.    /* Clear weakrefs and invoke callbacks as necessary. */
183.    // 循环所有的不可达元素，处理所有弱引用到unreachable，如果弱引用对象仍然生存则放回old链表中。
184.    m += handle_weakrefs(&unreachable, old);
185.
186.    validate_list(old, 0);
187.    validate_list(&unreachable, PREV_MASK_COLLECTING);
188.
189.    /* Call tp_finalize on objects which have one. */
190.    // 执行那些具有的__del__方法的对象。
191.    finalize_garbage(&unreachable);
192.
193.    // 最后，进行进行对垃圾的清除。
194.    if (check_garbage(&unreachable)) { // clear PREV_MASK_COLLECTING here
195.       gc_list_merge(&unreachable, old);
196.    }
197.    else {
198.       /* Call tp_clear on objects in the unreachable set.  This will cause
199.          * the reference cycles to be broken.  It may also cause some objects
200.          * in finalizers to be freed.
201.          */
202.       m += gc_list_size(&unreachable);
203.       delete_garbage(state, &unreachable, old);
204.    }
205.
206.    /* Collect statistics on uncollectable objects found and print
207.    * debugging information. */
```

```
    for (gc = GC_NEXT(&finalizers); gc != &finalizers; gc = GC_NEXT(gc)) {

        n++;
        if (state->debug & DEBUG_UNCOLLECTABLE)
            debug_cycle("uncollectable", FROM_GC(gc));
    }
    if (state->debug & DEBUG_STATS) {
        double d = _PyTime_AsSecondsDouble(_PyTime_GetMonotonicClock() - t1);
        PySys_WriteStderr(
            "gc: done, %" PY_FORMAT_SIZE_T "d unreachable, "
            "%" PY_FORMAT_SIZE_T "d uncollectable, %.4fs elapsed\n",
            n+m, n, d);
    }

    /* Append instances in the uncollectable set to a Python
    * reachable list of garbage.  The programmer has to deal with
    * this if they insist on creating this type of structure.
    */
    // 执行完 __del__没有，不应该被删除的对象，再重新加入到可达链表中。
    handle_legacy_finalizers(state, &finalizers, old);
    validate_list(old, 0);

    /* Clear free list only during the collection of the highest
    * generation */
    if (generation == NUM_GENERATIONS-1) {
        clear_freelists();
    }
     ...
    return n+m;
}
```

2.6.2 引用

```
1. v1 = [11,22,33]
2. v2 = v1
```

当对对象进行引用时候，内部引用计数器+1，原理同上。

2.6.3 销毁

```
1. v1 = [11,22,33]
2. del v1
```

对列表对象进行销毁时，本质上就会执行引用计数器-1（同上），但当引用计数器为0时候，会执行list对象的 `tp_dealloc` ，即：

```
// Object/listobject.c

PyTypeObject PyList_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "list",
    sizeof(PyListObject),
    0,
    (destructor)list_dealloc,                   /* tp_dealloc */
    ...
    PyObject_GC_Del,                            /* tp_free */
};

/* Empty list reuse scheme to save calls to malloc and free */
#ifndef PyList_MAXFREELIST
#define PyList_MAXFREELIST 80
#endif
static PyListObject *free_list[PyList_MAXFREELIST];
static int numfree = 0;

static void
list_dealloc(PyListObject *op)
{
    Py_ssize_t i;
    // 从分代回收的的代中移除
    PyObject_GC_UnTrack(op);
    Py_TRASHCAN_BEGIN(op, list_dealloc)
    if (op->ob_item != NULL) {
        /* Do it backwards, for Christian Tismer.
```

```
            There's a simple test case where somehow this reduces
            thrashing when a *very* large list is created and
            immediately deleted. */
        i = Py_SIZE(op);
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }
        PyMem_FREE(op->ob_item);
    }
    if (numfree < PyList_MAXFREELIST && PyList_CheckExact(op))
        // free_list中还么有占满80，不销毁并缓冲在free_list中
        free_list[numfree++] = op;
    else
        // 销毁并在refchain中移除
        Py_TYPE(op)->tp_free((PyObject *)op);
    Py_TRASHCAN_END
}
```

2.7 tuple类型

2.7.1 创建

```
v = (11,22,33)
```

当创建元组时候，会执行如下源码：

```c
// Objects/tupleobject.c

#define PyTuple_MAXSAVESIZE     20  /* Largest tuple to save on free list */
#define PyTuple_MAXFREELIST  2000  /* Maximum number of tuples of each size to save */

static PyTupleObject *free_list[PyTuple_MAXSAVESIZE]; // free_list[20]
static int numfree[PyTuple_MAXSAVESIZE]; // numfree[20]

PyObject *
PyTuple_New(Py_ssize_t size)
{
    PyTupleObject *op;
    ...
    // free_list第0个元素存储的是空元祖
    if (size == 0 && free_list[0]) {
        op = free_list[0];
        Py_INCREF(op);
        return (PyObject *) op;
    }
    // 有缓存的tuple对象，则从free_list中获取
    if (size < PyTuple_MAXSAVESIZE && (op = free_list[size]) != NULL) {
        // 获取对象并初始化
        free_list[size] = (PyTupleObject *) op->ob_item[0];
        numfree[size]--;
        Py_SIZE(op) = size;
        Py_TYPE(op) = &PyTuple_Type;
        // 对象加入到refchain链表。
        _Py_NewReference((PyObject *)op);
```

```
    }
    else
    {
        ..
        // 没有缓存数据，则创建对象
        op = PyObject_GC_NewVar(PyTupleObject, &PyTuple_Type, size);
        if (op == NULL)
            return NULL;
    }
    for (i=0; i < size; i++)
        op->ob_item[i] = NULL;
    if (size == 0) {
        free_list[0] = op;
        ++numfree[0];
        Py_INCREF(op);          /* extra INCREF so that this is never freed */
    }
    // 对象加入到分代的链表。
    _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}
```

```
// Includes/objimpl.h

#define PyObject_GC_NewVar(type, typeobj, n) \
                ( (type *) _PyObject_GC_NewVar((typeobj), (n)) )
```

```
Objects/gcmodules.c

PyVarObject *
_PyObject_GC_NewVar(PyTypeObject *tp, Py_ssize_t nitems)
{
    size_t size;
    PyVarObject *op;

    if (nitems < 0) {
        PyErr_BadInternalCall();
        return NULL;
    }
    size = _PyObject_VAR_SIZE(tp, nitems);
    // 开内存 & 分代 & 超过阈值则垃圾回收（流程同上述 列表过程）
    op = (PyVarObject *) _PyObject_GC_Malloc(size);
    if (op != NULL)
        op = PyObject_INIT_VAR(op, tp, nitems);
    return op;
}
```

2.7.2 引用

```
v1 = (11,22,33)
v2 = v1
```

引用时会触发引用计数器 + 1，具体流程同上。

2.7.3 销毁

```
1. v = (11,22,33)
2. del v
```

销毁对象时候，执行引用计数器-1，如果计数器减为0，则触发tuple类型的 `tp_dealloc` ，详细如下：

```
1.
2.
3. PyTypeObject PyTuple_Type = {
4.     PyVarObject_HEAD_INIT(&PyType_Type, 0)
5.     "tuple",
6.     sizeof(PyTupleObject) - sizeof(PyObject *),
7.     sizeof(PyObject *),
8.     (destructor)tupledealloc,                   /* tp_dealloc */
9.     ...
10.    PyObject_GC_Del,                            /* tp_free */
11. };
12.
13. static void
14. tupledealloc(PyTupleObject *op)
15. {
16.     Py_ssize_t i;
17.     Py_ssize_t len =  Py_SIZE(op);
18.     // 从分代的链表中移除
19.     PyObject_GC_UnTrack(op);
20.     Py_TRASHCAN_BEGIN(op, tupledealloc)
21.     if (len > 0) {
22.         i = len;
23.         while (--i >= 0)
24.             Py_XDECREF(op->ob_item[i]);
25.         // 缓存到free_list中
26.         if (len < PyTuple_MAXSAVESIZE && numfree[len] < PyTuple_MAXFREELIST &&
27.             Py_TYPE(op) == &PyTuple_Type)
28.         {
```

```
            op->ob_item[0] = (PyObject *) free_list[len];
            numfree[len]++;
            free_list[len] = op;
            // 结束
            goto done; /* return */
        }

    }
    // 不缓存，则直接销毁对象并在refchain链表中移除。
    Py_TYPE(op)->tp_free((PyObject *)op);
done:
    Py_TRASHCAN_END
}
```

2.8 dict类型

2.8.1 创建

```
v = {"name":"武沛齐","age":18}
```

当创建一个字典对象时，Python底层会执行如下源码：

```
1.
2. #define PyDict_MAXFREELIST 80
3. // 缓存dict对象的free_list
4. static PyDictObject *free_list[PyDict_MAXFREELIST];
5. static int numfree = 0;
6.
7.
8. PyObject *
9. PyDict_New(void)
10. {
11.     dictkeys_incref(Py_EMPTY_KEYS);
12.     return new_dict(Py_EMPTY_KEYS, empty_values);
13. }
14.
15. /* Consumes a reference to the keys object */
16. static PyObject *
17. new_dict(PyDictKeysObject *keys, PyObject **values)
18. {
19.     PyDictObject *mp;
20.     assert(keys != NULL);
21.     // 如果有缓存，则从缓存区获取一个对象
22.     if (numfree) {
23.         mp = free_list[--numfree];
24.         assert (mp != NULL);
25.         assert (Py_TYPE(mp) == &PyDict_Type);
26.         _Py_NewReference((PyObject *)mp);
27.     }
28.     else {
```

```
        // 没有缓存，则去创建字典对象。（源码流程同list类型）
        mp = PyObject_GC_New(PyDictObject, &PyDict_Type);
        if (mp == NULL) {
            dictkeys_decref(keys);
            if (values != empty_values) {
                free_values(values);
            }
            return NULL;
        }
    }
    mp->ma_keys = keys;
    mp->ma_values = values;
    mp->ma_used = 0;
    mp->ma_version_tag = DICT_NEXT_VERSION();
    ASSERT_CONSISTENT(mp);
    return (PyObject *)mp;
}
```

2.8.2 引用

```
v1 = {"name":"武沛齐","age":18}
v2 = v1
```

出现引用，则应用计数器+1（同上）。

2.8.3 销毁

```
v1 = {"name":"武沛齐","age":18}
del v1
```

销毁一个对象时候，引用计数器-1，当减到0时候，则触发dict类型的 `tp_dealloc` ，源码如下：

```
// Object/dictobject.c

PyTypeObject PyDict_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "dict",
    sizeof(PyDictObject),
    0,
    (destructor)dict_dealloc,                   /* tp_dealloc */
    ...
    PyObject_GC_Del,                            /* tp_free */
};

static void
dict_dealloc(PyDictObject *mp)
{
    PyObject **values = mp->ma_values;
    PyDictKeysObject *keys = mp->ma_keys;
    Py_ssize_t i, n;

    // 从分代链表中移除
    PyObject_GC_UnTrack(mp);
    Py_TRASHCAN_BEGIN(mp, dict_dealloc)
    ...
    // 缓存区为满，则缓存
    if (numfree < PyDict_MAXFREELIST && Py_TYPE(mp) == &PyDict_Type)
        free_list[numfree++] = mp;
    else
        // 已满则销毁，并在refchain中移除。
```

```
29.         Py_TYPE(mp)->tp_free((PyObject *)mp);
30.     Py_TRASHCAN_END
31. }
```

写在最后

上述的过程是基于Python3.8.2的源码来进行剖析，好了，学会之后拿去尽情的装逼吧，哈哈哈哈哈。。。。。。

用户评论

1

牛掰啊

13798113820 2020-05-17 15:32 回复 (/login/)

小

学到了，谢谢

小么小儿郎EL 2020-08-24 15:51 回复 (/login/)

L

可以转载吗佩奇老师

lokvke 2021-03-14 16:01 回复 (/login/)

L

牛掰，

liulei 2021-06-24 09:20 回复 (/login/)

W

受益匪浅

wutianxiao666@163.com 2021-10-08 07:34 回复 (/login/)

目录