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。

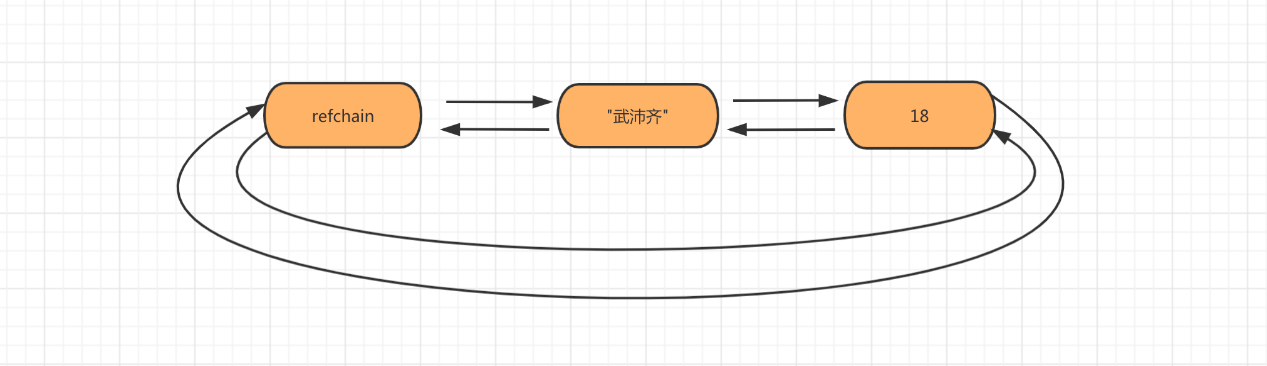
**1. 白话垃圾回收**

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：

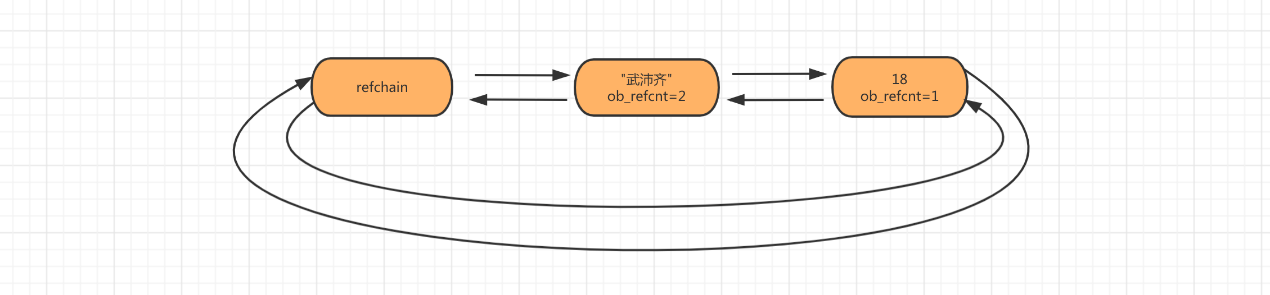
1. age = 18
2. name = "武沛齐"



**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）。

1. age = 18
2. number = age # 对象18的引用计数器 + 1
3. del age # 对象18的引用计数器 - 1
4. def run(arg):
5. print(arg)
6. 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 。
7. 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：

1. 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.
2. 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.
3. 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.
4. 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.
5. del v1 # 引用计数器-1
6. 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. };

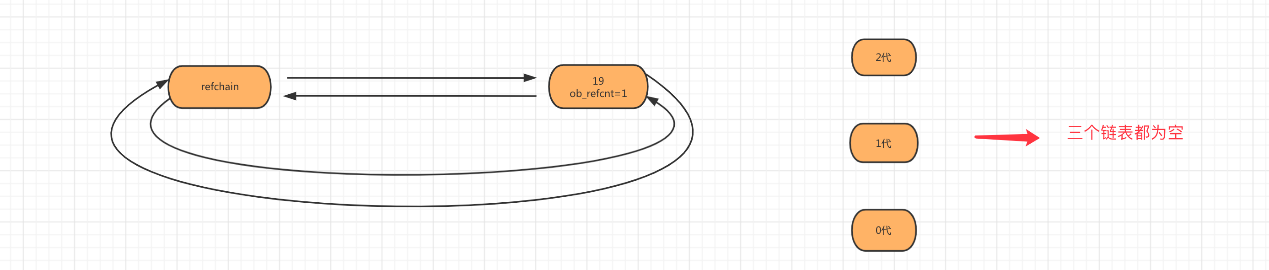
特别注意：0代和1、2代的threshold和count表示的意义不同。

* 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。

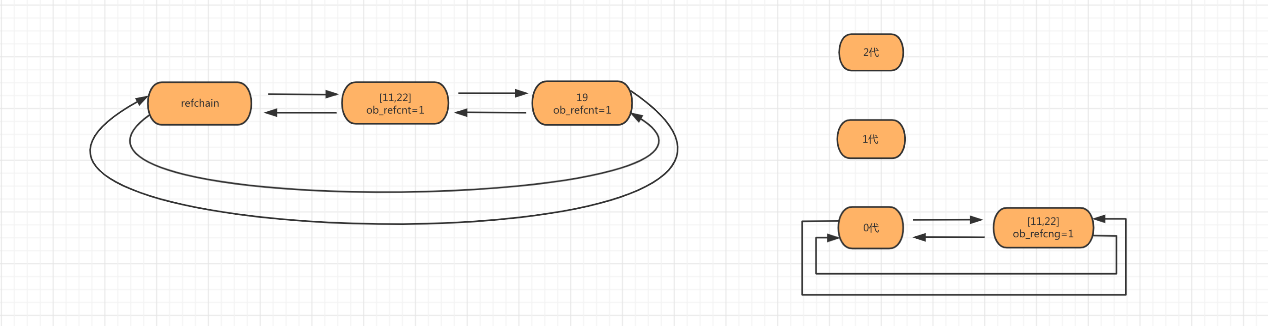
**1.4 情景模拟**

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，主要就是剔除循环引用和销毁垃圾，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。
  1. v1 = 3.14 # 开辟内存来存储float对象，并将对象添加到refchain链表。
  2. print( id(v1) ) # 内存地址：4436033488
  3. 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.
  4. 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。
  5. print( id(v2) ) # 内存地址：4436033488
  6. # 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。
  1. 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。
  2. print( id(v1)) #内存地址：4514343712
  3. 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。
  4. print( id(v2) ) #内存地址：4514343712
  5. # 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。
  1. v1 = "A"
  2. print( id(v1) ) # 输出：4517720496
  3. del v1
  4. v2 = "A"
  5. print( id(v1) ) # 输出：4517720496
  6. # 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）。
  7. v1 = "wupeiqi"
  8. v2 = "wupeiqi"
  9. 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。
  1. v1 = [11,22,33]
  2. print( id(v1) ) # 输出：4517628816
  3. del v1
  4. v2 = ["武","沛齐"]
  5. 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。
  1. v1 = (1,2)
  2. print( id(v1) )
  3. 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。
  4. 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。
  5. print( id(v2) )
* dict类型，维护的free\_list数组最多可缓存80个dict对象。
  1. v1 = {"k1":123}
  2. print( id(v1) ) # 输出：4515998128
  3. del v1
  4. v2 = {"name":"武沛齐","age":18,"gender":"男"}
  5. 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**

1. #define PyObject\_HEAD PyObject ob\_base;
2. #define PyObject\_VAR\_HEAD PyVarObject ob\_base;
3. // 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)
4. #define \_PyObject\_HEAD\_EXTRA \
5. struct \_object \*\_ob\_next; \
6. struct \_object \*\_ob\_prev;
7. typedef struct \_object {
8. \_PyObject\_HEAD\_EXTRA // 用于构造双向链表
9. Py\_ssize\_t ob\_refcnt; // 引用计数器
10. struct \_typeobject \*ob\_type; // 数据类型
11. } PyObject;
12. typedef struct {
13. PyObject ob\_base; // PyObject对象
14. Py\_ssize\_t ob\_size; /\* Number of items in variable part，即：元素个数 \*/
15. } 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**

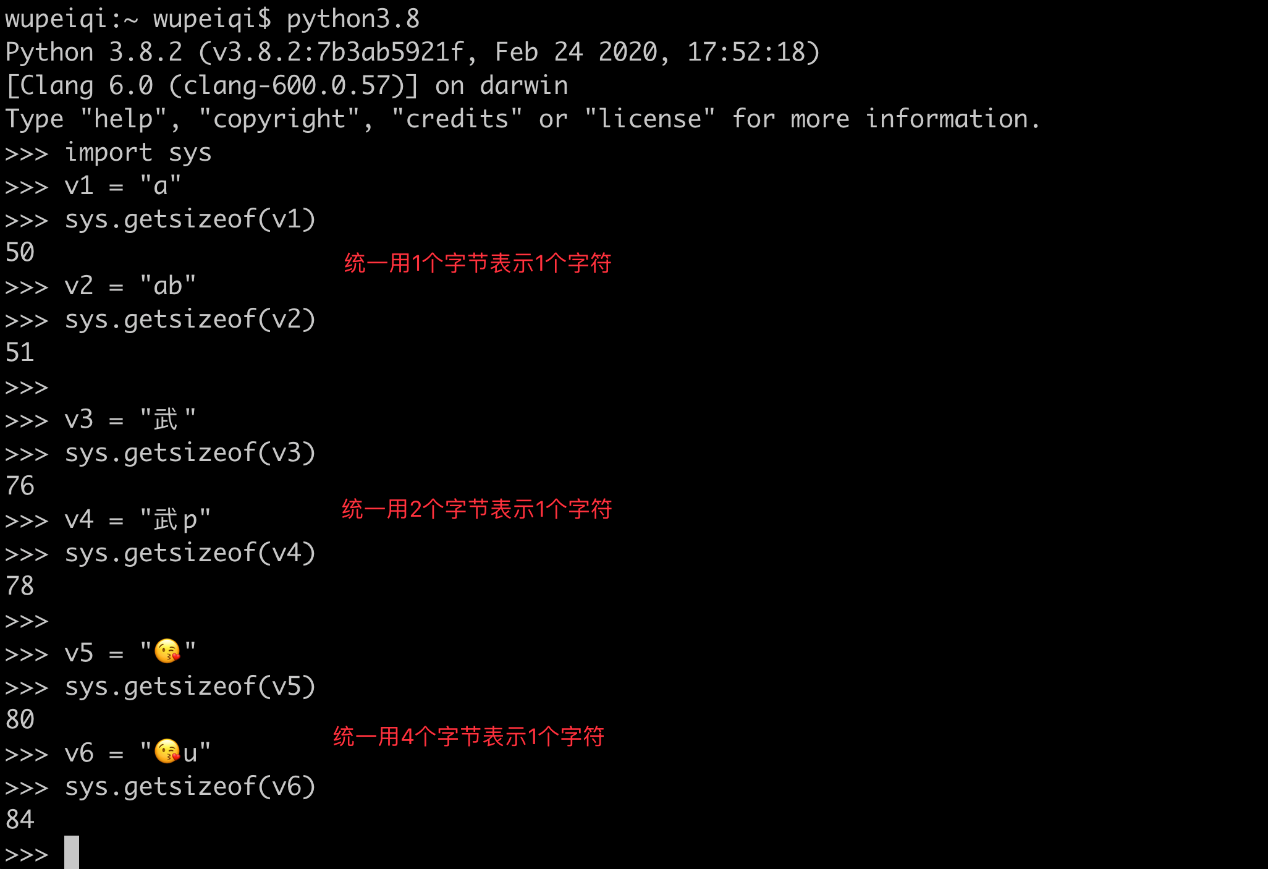
平时我们在创建一个对象时，本质上就是实例化一个相关类型的结构体，在内部保存值和引用计数器等。

* float类型
  1. typedef struct {
  2. PyObject\_HEAD
  3. double ob\_fval;
  4. } PyFloatObject;
* int类型
  1. struct \_longobject {
  2. PyObject\_VAR\_HEAD
  3. digit ob\_digit[1];
  4. };
  5. /\* Long (arbitrary precision) integer object interface \*/
  6. typedef struct \_longobject PyLongObject; /\* Revealed in longintrepr.h \*/
* str类型
  1. typedef struct {
  2. PyObject\_HEAD
  3. Py\_ssize\_t length; /\* Number of code points in the string \*/
  4. Py\_hash\_t hash; /\* Hash value; -1 if not set \*/
  5. struct {
  6. unsigned int interned:2;
  7. /\* Character size:
  8. - PyUnicode\_WCHAR\_KIND (0):
  9. \* character type = wchar\_t (16 or 32 bits, depending on the
  10. platform)
  11. - PyUnicode\_1BYTE\_KIND (1):
  12. \* character type = Py\_UCS1 (8 bits, unsigned)
  13. \* all characters are in the range U+0000-U+00FF (latin1)
  14. \* if ascii is set, all characters are in the range U+0000-U+007F
  15. (ASCII), otherwise at least one character is in the range
  16. U+0080-U+00FF
  17. - PyUnicode\_2BYTE\_KIND (2):
  18. \* character type = Py\_UCS2 (16 bits, unsigned)
  19. \* all characters are in the range U+0000-U+FFFF (BMP)
  20. \* at least one character is in the range U+0100-U+FFFF
  21. - PyUnicode\_4BYTE\_KIND (4):
  22. \* character type = Py\_UCS4 (32 bits, unsigned)
  23. \* all characters are in the range U+0000-U+10FFFF
  24. \* at least one character is in the range U+10000-U+10FFFF
  25. \*/
  26. unsigned int kind:3;
  27. unsigned int compact:1;
  28. unsigned int ascii:1;
  29. unsigned int ready:1;
  30. unsigned int :24;
  31. } state;
  32. wchar\_t \*wstr; /\* wchar\_t representation (null-terminated) \*/
  33. } PyASCIIObject;
  34. typedef struct {
  35. PyASCIIObject \_base;
  36. Py\_ssize\_t utf8\_length; /\* Number of bytes in utf8, excluding the
  37. \* terminating \0. \*/
  38. char \*utf8; /\* UTF-8 representation (null-terminated) \*/
  39. Py\_ssize\_t wstr\_length; /\* Number of code points in wstr, possible
  40. \* surrogates count as two code points. \*/
  41. } PyCompactUnicodeObject;
  42. typedef struct {
  43. PyCompactUnicodeObject \_base;
  44. union {
  45. void \*any;
  46. Py\_UCS1 \*latin1;
  47. Py\_UCS2 \*ucs2;
  48. Py\_UCS4 \*ucs4;
  49. } data; /\* Canonical, smallest-form Unicode buffer \*/
  50. } 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



**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：

1. // Objects/floatobject.c
2. // 用于缓存float对象的链表
3. static PyFloatObject \*free\_list = NULL;
4. static int numfree = 0;
5. PyObject \*
6. PyFloat\_FromDouble(double fval)
7. {
8. // 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。
9. PyFloatObject \*op = free\_list;
10. if (op != NULL) {
11. free\_list = (PyFloatObject \*) Py\_TYPE(op);
12. numfree--;
13. } else {
14. // 根据float类型的大小，为float对象新开辟内存。
15. op = (PyFloatObject\*) PyObject\_MALLOC(sizeof(PyFloatObject));
16. if (!op)
17. return PyErr\_NoMemory();
18. }
19. // 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。
20. /\* Inline PyObject\_New \*/
21. (void)PyObject\_INIT(op, &PyFloat\_Type);
22. // 对float对象赋值。即：op->ob\_fval = 3.14
23. op->ob\_fval = fval;
24. return (PyObject \*) op;
25. }
26. // Include/objimpl.h
27. #define PyObject\_INIT(op, typeobj) \
28. ( Py\_TYPE(op) = (typeobj), \_Py\_NewReference((PyObject \*)(op)), (op) )
29. // Objects/object.c
30. // 维护了所有对象的一个环状双向链表
31. static PyObject refchain = {&refchain, &refchain};
32. void
33. \_Py\_AddToAllObjects(PyObject \*op, int force)
34. {
35. if (force || op->\_ob\_prev == NULL) {
36. op->\_ob\_next = refchain.\_ob\_next;
37. op->\_ob\_prev = &refchain;
38. refchain.\_ob\_next->\_ob\_prev = op;
39. refchain.\_ob\_next = op;
40. }
41. }
42. void
43. \_Py\_NewReference(PyObject \*op)
44. {
45. \_Py\_INC\_REFTOTAL;
46. // 引用计数器初始化为1。
47. op->ob\_refcnt = 1;
48. // 对象添加到双向链表refchain中。
49. \_Py\_AddToAllObjects(op, 1);
50. \_Py\_INC\_TPALLOCS(op);
51. }

**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. static inline void \_Py\_INCREF(PyObject \*op)
3. {
4. \_Py\_INC\_REFTOTAL;
5. // 对象的引用计数器 + 1
6. op->ob\_refcnt++;
7. }
8. #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：

1. // Include/object.h
2. static inline void \_Py\_DECREF(const char \*filename, int lineno,
3. PyObject \*op)
4. {
5. (void)filename; /\* may be unused, shut up -Wunused-parameter \*/
6. (void)lineno; /\* may be unused, shut up -Wunused-parameter \*/
7. \_Py\_DEC\_REFTOTAL;
8. // 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。
9. if (--op->ob\_refcnt != 0) {
10. #ifdef Py\_REF\_DEBUG
11. if (op->ob\_refcnt < 0) {
12. \_Py\_NegativeRefcount(filename, lineno, op);
13. }
14. #endif
15. }
16. else {
17. \_Py\_Dealloc(op);
18. }
19. }
20. #define Py\_DECREF(op) \_Py\_DECREF(\_\_FILE\_\_, \_\_LINE\_\_, \_PyObject\_CAST(op))
21. // Objects/object.c
22. void
23. \_Py\_Dealloc(PyObject \*op)
24. {
25. // 找到float类型的 tp\_dealloc 函数
26. destructor dealloc = Py\_TYPE(op)->tp\_dealloc;
27. // 在refchain双向链表中摘除此对象。
28. \_Py\_ForgetReference(op);
29. // 执行float类型的 tp\_dealloc 函数，去进行缓存或垃圾回收。
30. (\*dealloc)(op);
31. }
32. void
33. \_Py\_ForgetReference(PyObject \*op)
34. {
35. ...
36. // 在refchain链表中移除此对象
37. op->\_ob\_next->\_ob\_prev = op->\_ob\_prev;
38. op->\_ob\_prev->\_ob\_next = op->\_ob\_next;
39. op->\_ob\_next = op->\_ob\_prev = NULL;
40. \_Py\_INC\_TPFREES(op);
41. }
42. // Objects/floatobject.c
43. #define PyFloat\_MAXFREELIST 100
44. static int numfree = 0;
45. static PyFloatObject \*free\_list = NULL;
46. // float类型中函数的对应关系
47. PyTypeObject PyFloat\_Type = {
48. PyVarObject\_HEAD\_INIT(&PyType\_Type, 0)
49. "float",
50. sizeof(PyFloatObject),
51. 0,
52. // tp\_dealloc表示执行float\_dealloc方法
53. (destructor)float\_dealloc, /\* tp\_dealloc \*/
54. 0, /\* tp\_print \*/
55. ...
56. };
57. static void
58. float\_dealloc(PyFloatObject \*op)
59. {
60. // 检测是否是float类型
61. if (PyFloat\_CheckExact(op)) {
62. // 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。
63. if (numfree >= PyFloat\_MAXFREELIST) {
64. // 销毁
65. PyObject\_FREE(op);
66. return;
67. }
68. // 将对象加入到free\_list链表中
69. numfree++;
70. Py\_TYPE(op) = (struct \_typeobject \*)free\_list;
71. free\_list = op;
72. }
73. else
74. Py\_TYPE(op)->tp\_free((PyObject \*)op);
75. }

**2.4 int类型**

**2.4.1 创建**

1. 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:

1. PyObject \*
2. PyLong\_FromLong(long ival)
3. {
4. PyLongObject \*v;
5. ...
6. // 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）
7. CHECK\_SMALL\_INT(ival);
8. ...
9. // 非小数字池中的值，重新开辟内存并初始化
10. v = \_PyLong\_New(ndigits);
11. if (v != NULL) {
12. digit \*p = v->ob\_digit;
13. Py\_SIZE(v) = ndigits\*sign;
14. t = abs\_ival;
15. ...
16. }
17. return (PyObject \*)v;
18. }
19. #define NSMALLNEGINTS 5
20. #define NSMALLPOSINTS 257
21. #define CHECK\_SMALL\_INT(ival) \
22. do if (-NSMALLNEGINTS <= ival && ival < NSMALLPOSINTS) { \
23. return get\_small\_int((sdigit)ival); \
24. } while(0)
25. static PyObject \*
26. get\_small\_int(sdigit ival)
27. {
28. PyObject \*v;
29. v = (PyObject \*)&small\_ints[ival + NSMALLNEGINTS];
30. // 引用计数器 + 1
31. Py\_INCREF(v);
32. ...
33. return v;
34. }
35. PyLongObject \*
36. \_PyLong\_New(Py\_ssize\_t size)
37. {
38. // 创建PyLongObject的指针变量
39. PyLongObject \*result;
40. ...
41. // 根据长度进行开辟内存
42. result = PyObject\_MALLOC(offsetof(PyLongObject, ob\_digit) +
43. size\*sizeof(digit));
44. ...
45. // 对内存中的数据进行初始化并添加到refchain链表中。
46. return (PyLongObject\*)PyObject\_INIT\_VAR(result, &PyLong\_Type, size);
47. }
48. // Include/objimpl.h
49. #define PyObject\_NewVar(type, typeobj, n) \
50. ( (type \*) \_PyObject\_NewVar((typeobj), (n)) )
51. static inline PyVarObject\*
52. \_PyObject\_INIT\_VAR(PyVarObject \*op, PyTypeObject \*typeobj, Py\_ssize\_t size)
53. {
54. assert(op != NULL);
55. Py\_SIZE(op) = size;
56. // 对象初始化
57. PyObject\_INIT((PyObject \*)op, typeobj);
58. return op;
59. }
60. #define PyObject\_INIT(op, typeobj) \
61. \_PyObject\_INIT(\_PyObject\_CAST(op), (typeobj))
62. static inline PyObject\*
63. \_PyObject\_INIT(PyObject \*op, PyTypeObject \*typeobj)
64. {
65. assert(op != NULL);
66. Py\_TYPE(op) = typeobj;
67. if (PyType\_GetFlags(typeobj) & Py\_TPFLAGS\_HEAPTYPE) {
68. Py\_INCREF(typeobj);
69. }
70. // 对象初始化，并把对象加入到refchain链表。
71. \_Py\_NewReference(op);
72. return op;
73. }
74. // Objects/object.c
75. // 维护了所有对象的一个环状双向链表
76. static PyObject refchain = {&refchain, &refchain};
77. void
78. \_Py\_AddToAllObjects(PyObject \*op, int force)
79. {
80. if (force || op->\_ob\_prev == NULL) {
81. op->\_ob\_next = refchain.\_ob\_next;
82. op->\_ob\_prev = &refchain;
83. refchain.\_ob\_next->\_ob\_prev = op;
84. refchain.\_ob\_next = op;
85. }
86. }
87. void
88. \_Py\_NewReference(PyObject \*op)
89. {
90. \_Py\_INC\_REFTOTAL;
91. // 引用计数器初始化为1。
92. op->ob\_refcnt = 1;
93. // 对象添加到双向链表refchain中。
94. \_Py\_AddToAllObjects(op, 1);
95. \_Py\_INC\_TPALLOCS(op);
96. }

**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：

1. // Include/object.h
2. static inline void \_Py\_DECREF(const char \*filename, int lineno,
3. PyObject \*op)
4. {
5. (void)filename; /\* may be unused, shut up -Wunused-parameter \*/
6. (void)lineno; /\* may be unused, shut up -Wunused-parameter \*/
7. \_Py\_DEC\_REFTOTAL;
8. // 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。
9. if (--op->ob\_refcnt != 0) {
10. #ifdef Py\_REF\_DEBUG
11. if (op->ob\_refcnt < 0) {
12. \_Py\_NegativeRefcount(filename, lineno, op);
13. }
14. #endif
15. }
16. else {
17. \_Py\_Dealloc(op);
18. }
19. }
20. #define Py\_DECREF(op) \_Py\_DECREF(\_\_FILE\_\_, \_\_LINE\_\_, \_PyObject\_CAST(op))
21. // Objects/object.c
22. void
23. \_Py\_Dealloc(PyObject \*op)
24. {
25. // 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）
26. // 此处体现所有的类型都继承object
27. destructor dealloc = Py\_TYPE(op)->tp\_dealloc;
28. // 在refchain双向链表中摘除此对象。
29. \_Py\_ForgetReference(op);
30. // 执行int类型的 tp\_dealloc 函数，去进行垃圾回收。
31. (\*dealloc)(op);
32. }
33. void
34. \_Py\_ForgetReference(PyObject \*op)
35. {
36. ...
37. // 在refchain链表中移除此对象
38. op->\_ob\_next->\_ob\_prev = op->\_ob\_prev;
39. op->\_ob\_prev->\_ob\_next = op->\_ob\_next;
40. op->\_ob\_next = op->\_ob\_prev = NULL;
41. \_Py\_INC\_TPFREES(op);
42. }
43. // Objects/longobjet.c
44. PyTypeObject PyLong\_Type = {
45. PyVarObject\_HEAD\_INIT(&PyType\_Type, 0)
46. "int", /\* tp\_name \*/
47. offsetof(PyLongObject, ob\_digit), /\* tp\_basicsize \*/
48. sizeof(digit), /\* tp\_itemsize \*/
49. 0, /\* tp\_dealloc \*/
50. ...
51. PyObject\_Del, /\* tp\_free \*/
52. };
53. Objects/typeobject.c
54. PyTypeObject PyBaseObject\_Type = {
55. PyVarObject\_HEAD\_INIT(&PyType\_Type, 0)
56. "object", /\* tp\_name \*/
57. sizeof(PyObject), /\* tp\_basicsize \*/
58. 0, /\* tp\_itemsize \*/
59. object\_dealloc, /\* tp\_dealloc \*/
60. ...
61. PyObject\_Del, /\* tp\_free \*/
62. };
63. static void
64. object\_dealloc(PyObject \*self)
65. {
66. // 调用int类型的 tp\_free，即：PyObject\_Del去销毁对象。
67. Py\_TYPE(self)->tp\_free(self);
68. }

**2.5 str类型**

**2.5.1 创建**

1. 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:

1. Objects/unicodeobject.c
2. PyObject \*
3. PyUnicode\_DecodeUTF8Stateful(const char \*s,Py\_ssize\_t size,const char \*errors,Py\_ssize\_t \*consumed)
4. {
5. return unicode\_decode\_utf8(s, size, \_Py\_ERROR\_UNKNOWN, errors, consumed);
6. }
7. static PyObject \*
8. unicode\_decode\_utf8(const char \*s, Py\_ssize\_t size,\_Py\_error\_handler error\_handler, const char \*errors,Py\_ssize\_t \*consumed);
9. {
10. ...
11. // 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。
12. if (size == 1 && (unsigned char)s[0] < 128) {
13. if (consumed)
14. \*consumed = 1;
15. return get\_latin1\_char((unsigned char)s[0]);
16. }
17. // 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）。
18. ...
19. return \_PyUnicodeWriter\_Finish(&writer);
20. }
21. static PyObject\*
22. get\_latin1\_char(unsigned char ch)
23. {
24. PyObject \*unicode = unicode\_latin1[ch];
25. if (!unicode) {
26. unicode = PyUnicode\_New(1, ch);
27. if (!unicode)
28. return NULL;
29. PyUnicode\_1BYTE\_DATA(unicode)[0] = ch;
30. assert(\_PyUnicode\_CheckConsistency(unicode, 1));
31. unicode\_latin1[ch] = unicode;
32. }
33. Py\_INCREF(unicode);
34. return unicode;
35. }
36. PyObject \*
37. \_PyUnicodeWriter\_Finish(\_PyUnicodeWriter \*writer)
38. {
39. PyObject \*str;
40. // 写入值到str
41. str = writer->buffer;
42. writer->buffer = NULL;
43. if (writer->readonly) {
44. assert(PyUnicode\_GET\_LENGTH(str) == writer->pos);
45. return str;
46. }
47. if (PyUnicode\_GET\_LENGTH(str) != writer->pos) {
48. PyObject \*str2;
49. // 创建对象
50. str2 = resize\_compact(str, writer->pos);
51. if (str2 == NULL) {
52. Py\_DECREF(str);
53. return NULL;
54. }
55. str = str2;
56. }
57. assert(\_PyUnicode\_CheckConsistency(str, 1));
58. return unicode\_result\_ready(str);
59. }
60. static PyObject\*
61. resize\_compact(PyObject \*unicode, Py\_ssize\_t length)
62. {
63. ...
64. // 开辟内存
65. new\_unicode = (PyObject \*)PyObject\_REALLOC(unicode, new\_size);
66. if (new\_unicode == NULL) {
67. \_Py\_NewReference(unicode);
68. PyErr\_NoMemory();
69. return NULL;
70. }
71. unicode = new\_unicode;
72. // 把对象加入到refchain链表
73. \_Py\_NewReference(unicode);
74. ...
75. return unicode;
76. }

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。

1. Objects/unicodeobject.c
2. void
3. PyUnicode\_InternInPlace(PyObject \*\*p)
4. {
5. PyObject \*s = \*p;
6. PyObject \*t;
7. #ifdef Py\_DEBUG
8. assert(s != NULL);
9. assert(\_PyUnicode\_CHECK(s));
10. #else
11. if (s == NULL || !PyUnicode\_Check(s))
12. return;
13. #endif
14. /\* If it's a subclass, we don't really know what putting
15. it in the interned dict might do. \*/
16. if (!PyUnicode\_CheckExact(s))
17. return;
18. if (PyUnicode\_CHECK\_INTERNED(s))
19. return;
20. if (interned == NULL) {
21. interned = PyDict\_New();
22. if (interned == NULL) {
23. PyErr\_Clear(); /\* Don't leave an exception \*/
24. return;
25. }
26. }
27. Py\_ALLOW\_RECURSION
28. // 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。
29. t = PyDict\_SetDefault(interned, s, s);
30. Py\_END\_ALLOW\_RECURSION
31. if (t == NULL) {
32. PyErr\_Clear();
33. return;
34. }
35. // 存在，使用已驻留的字符串 并 将引用计数器+1
36. if (t != s) {
37. Py\_INCREF(t);
38. Py\_SETREF(\*p, t); // 处理临时对象
39. return;
40. }
41. /\* The two references in interned are not counted by refcnt.
42. The deallocator will take care of this \*/
43. Py\_REFCNT(s) -= 2; // 让临时对象可被回收。
44. \_PyUnicode\_STATE(s).interned = SSTATE\_INTERNED\_MORTAL;
45. }

**2.5.2 引用**

同上，引用计数器 + 1 .

**2.5.3 销毁**

1. val = "武沛齐"
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。

1. // Include/object.h
2. static inline void \_Py\_DECREF(const char \*filename, int lineno,
3. PyObject \*op)
4. {
5. (void)filename; /\* may be unused, shut up -Wunused-parameter \*/
6. (void)lineno; /\* may be unused, shut up -Wunused-parameter \*/
7. \_Py\_DEC\_REFTOTAL;
8. // 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。
9. if (--op->ob\_refcnt != 0) {
10. #ifdef Py\_REF\_DEBUG
11. if (op->ob\_refcnt < 0) {
12. \_Py\_NegativeRefcount(filename, lineno, op);
13. }
14. #endif
15. }
16. else {
17. \_Py\_Dealloc(op);
18. }
19. }
20. #define Py\_DECREF(op) \_Py\_DECREF(\_\_FILE\_\_, \_\_LINE\_\_, \_PyObject\_CAST(op))
21. // Objects/object.c
22. void
23. \_Py\_Dealloc(PyObject \*op)
24. {
25. // 找到str类型的 tp\_dealloc 函数
26. destructor dealloc = Py\_TYPE(op)->tp\_dealloc;
27. // 在refchain双向链表中摘除此对象。
28. \_Py\_ForgetReference(op);
29. // 执行float类型的 tp\_dealloc 函数，去进行缓存或垃圾回收。
30. (\*dealloc)(op);
31. }
32. void
33. \_Py\_ForgetReference(PyObject \*op)
34. {
35. ...
36. // 在refchain链表中移除此对象
37. op->\_ob\_next->\_ob\_prev = op->\_ob\_prev;
38. op->\_ob\_prev->\_ob\_next = op->\_ob\_next;
39. op->\_ob\_next = op->\_ob\_prev = NULL;
40. \_Py\_INC\_TPFREES(op);
41. }
42. // Objects/unicodeobject.c
43. PyTypeObject PyUnicode\_Type = {
44. PyVarObject\_HEAD\_INIT(&PyType\_Type, 0)
45. "str", /\* tp\_name \*/
46. sizeof(PyUnicodeObject), /\* tp\_basicsize \*/
47. 0, /\* tp\_itemsize \*/
48. /\* Slots \*/
49. (destructor)unicode\_dealloc, /\* tp\_dealloc \*/
50. ...
51. PyObject\_Del, /\* tp\_free \*/
52. };
53. static void
54. unicode\_dealloc(PyObject \*unicode)
55. {
56. switch (PyUnicode\_CHECK\_INTERNED(unicode)) {
57. case SSTATE\_NOT\_INTERNED:
58. break;
59. case SSTATE\_INTERNED\_MORTAL:
60. /\* revive dead object temporarily for DelItem \*/
61. Py\_REFCNT(unicode) = 3;
62. // 在interned中删除驻留的字符串
63. if (PyDict\_DelItem(interned, unicode) != 0)
64. Py\_FatalError(
65. "deletion of interned string failed");
66. break;
67. case SSTATE\_INTERNED\_IMMORTAL:
68. Py\_FatalError("Immortal interned string died.");
69. /\* fall through \*/
70. default:
71. Py\_FatalError("Inconsistent interned string state.");
72. }
73. if (\_PyUnicode\_HAS\_WSTR\_MEMORY(unicode))
74. PyObject\_DEL(\_PyUnicode\_WSTR(unicode));
75. if (\_PyUnicode\_HAS\_UTF8\_MEMORY(unicode))
76. PyObject\_DEL(\_PyUnicode\_UTF8(unicode));
77. if (!PyUnicode\_IS\_COMPACT(unicode) && \_PyUnicode\_DATA\_ANY(unicode))
78. PyObject\_DEL(\_PyUnicode\_DATA\_ANY(unicode));
79. // 内存中销毁对象
80. Py\_TYPE(unicode)->tp\_free(unicode);
81. }

**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：

1. // Objects/listobject.c
2. #define PyList\_MAXFREELIST 80
3. // free\_list用于对list对象进行缓存，最多可缓存80个对象
4. static PyListObject \*free\_list[PyList\_MAXFREELIST];
5. // free\_list中可用的对象
6. static int numfree = 0;
7. PyObject \*
8. PyList\_New(Py\_ssize\_t size)
9. {
10. PyListObject \*op;
11. if (size < 0) {
12. PyErr\_BadInternalCall();
13. return NULL;
14. }
15. if (numfree) {
16. // 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。
17. numfree--;
18. op = free\_list[numfree];
19. \_Py\_NewReference((PyObject \*)op);
20. } else {
21. // 缓存中没有，则需要 开辟内存 & 初始化对象
22. op = PyObject\_GC\_New(PyListObject, &PyList\_Type);
23. if (op == NULL)
24. return NULL;
25. }
26. if (size <= 0)
27. op->ob\_item = NULL;
28. else {
29. op->ob\_item = (PyObject \*\*) PyMem\_Calloc(size, sizeof(PyObject \*));
30. if (op->ob\_item == NULL) {
31. Py\_DECREF(op);
32. return PyErr\_NoMemory();
33. }
34. }
35. Py\_SIZE(op) = size;
36. op->allocated = size;
37. // 把对象加入到分代回收的三代中的0代链表中。
38. \_PyObject\_GC\_TRACK(op);
39. return (PyObject \*) op;
40. }
41. static inline void \_PyObject\_GC\_TRACK\_impl(const char \*filename, int lineno,
42. PyObject \*op)
43. {
44. \_PyObject\_ASSERT\_FROM(op, !\_PyObject\_GC\_IS\_TRACKED(op),
45. "object already tracked by the garbage collector",
46. filename, lineno, "\_PyObject\_GC\_TRACK");
47. PyGC\_Head \*gc = \_Py\_AS\_GC(op);
48. \_PyObject\_ASSERT\_FROM(op,
49. (gc->\_gc\_prev & \_PyGC\_PREV\_MASK\_COLLECTING) == 0,
50. "object is in generation which is garbage collected",
51. filename, lineno, "\_PyObject\_GC\_TRACK");
52. // 把对象加入到链表中，链表尾部还是gc.generation0。
53. PyGC\_Head \*last = (PyGC\_Head\*)(\_PyRuntime.gc.generation0->\_gc\_prev);
54. \_PyGCHead\_SET\_NEXT(last, gc);
55. \_PyGCHead\_SET\_PREV(gc, last);
56. \_PyGCHead\_SET\_NEXT(gc, \_PyRuntime.gc.generation0);
57. \_PyRuntime.gc.generation0->\_gc\_prev = (uintptr\_t)gc;
58. }
59. #define \_PyObject\_GC\_TRACK(op) \
60. \_PyObject\_GC\_TRACK\_impl(\_\_FILE\_\_, \_\_LINE\_\_, \_PyObject\_CAST(op))
61. Include/objimpl.h
62. #define PyObject\_GC\_New(type, typeobj) \
63. ( (type \*) \_PyObject\_GC\_New(typeobj) )
64. //Modules/gcmodule.c
65. PyObject \*
66. \_PyObject\_GC\_New(PyTypeObject \*tp)
67. {
68. // 创建对象
69. PyObject \*op = \_PyObject\_GC\_Malloc(\_PyObject\_SIZE(tp));
70. if (op != NULL)
71. // 初始化对象并把对象加入到refchain链表中。
72. op = PyObject\_INIT(op, tp);
73. return op;
74. }
75. PyObject \*
76. \_PyObject\_GC\_Malloc(size\_t basicsize)
77. {
78. return \_PyObject\_GC\_Alloc(0, basicsize);
79. }
80. static PyObject \*
81. \_PyObject\_GC\_Alloc(int use\_calloc, size\_t basicsize)
82. {
83. // 包含分代回收的三代链表
84. struct \_gc\_runtime\_state \*state = &\_PyRuntime.gc;
85. PyObject \*op;
86. PyGC\_Head \*g;
87. size\_t size;
88. if (basicsize > PY\_SSIZE\_T\_MAX - sizeof(PyGC\_Head))
89. return PyErr\_NoMemory();
90. size = sizeof(PyGC\_Head) + basicsize;
91. // 创建 gc head
92. if (use\_calloc)
93. g = (PyGC\_Head \*)PyObject\_Calloc(1, size);
94. else
95. g = (PyGC\_Head \*)PyObject\_Malloc(size);
96. if (g == NULL)
97. return PyErr\_NoMemory();
98. assert(((uintptr\_t)g & 3) == 0); // g must be aligned 4bytes boundary
99. g->\_gc\_next = 0;
100. g->\_gc\_prev = 0;
101. // 分代回收的0代数量+1
102. state->generations[0].count++; /\* number of allocated GC objects \*/
103. // 如果0代超出自己的阈值，进行垃圾分代回收。
104. if (state->generations[0].count > state->generations[0].threshold && state->enabled && state->generations[0].threshold && !state->collecting && !PyErr\_Occurred())
105. {
106. // 正在收集
107. state->collecting = 1;
108. // 去进行垃圾回收收集
109. collect\_generations(state);
110. // 结束收集
111. state->collecting = 0;
112. }
113. op = FROM\_GC(g);
114. return op;
115. }
116. /\* Get the object given the GC head \*/
117. #define FROM\_GC(g) ((PyObject \*)(((PyGC\_Head \*)g)+1))
118. static Py\_ssize\_t
119. collect\_generations(struct \_gc\_runtime\_state \*state)
120. {
121. Py\_ssize\_t n = 0;
122. // 倒序循环三代，按照：2、1、0顺序
123. for (int i = NUM\_GENERATIONS-1; i >= 0; i--) {
124. if (state->generations[i].count > state->generations[i].threshold) {
125. if (i == NUM\_GENERATIONS - 1 && state->long\_lived\_pending < state->long\_lived\_total / 4)
126. continue;
127. // 去进行回收，回收当前代之前的所有代。
128. n = collect\_with\_callback(state, i);
129. break;
130. }
131. }
132. return n;
133. }
134. static Py\_ssize\_t
135. collect\_with\_callback(struct \_gc\_runtime\_state \*state, int generation)
136. {
137. ...
138. // 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）
139. result = collect(state, generation, &collected, &uncollectable, 0);
140. ...
141. return result;
142. }
143. /\* This is the main function. Read this to understand how the collection process works. \*/
144. static Py\_ssize\_t
145. collect(struct \_gc\_runtime\_state \*state, int generation,
146. Py\_ssize\_t \*n\_collected, Py\_ssize\_t \*n\_uncollectable, int nofail)
147. {
148. int i;
149. Py\_ssize\_t m = 0; /\* # objects collected \*/
150. Py\_ssize\_t n = 0; /\* # unreachable objects that couldn't be collected \*/
151. PyGC\_Head \*young; /\* the generation we are examining \*/
152. PyGC\_Head \*old; /\* next older generation \*/
153. PyGC\_Head unreachable; /\* non-problematic unreachable trash \*/
154. PyGC\_Head finalizers; /\* objects with, & reachable from, \_\_del\_\_ \*/
155. PyGC\_Head \*gc;
156. \_PyTime\_t t1 = 0; /\* initialize to prevent a compiler warning \*/
157. /\* update collection and allocation counters \*/
158. // generation分别会是 0 1 2
159. // 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
160. // 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。
161. if (generation+1 < NUM\_GENERATIONS)
162. state->generations[generation+1].count += 1;
163. // 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
164. for (i = 0; i <= generation; i++)
165. state->generations[i].count = 0;
166. // 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.
167. // 将比自己代低的所有代，搞到一个链表中
168. // #define GEN\_HEAD(state, n) (&(state)->generations[n].head)
169. for (i = 0; i < generation; i++) {
170. gc\_list\_merge(GEN\_HEAD(state, i), GEN\_HEAD(state, generation));
171. }
172. // 获取当前代的head（链表头）
173. // #define GEN\_HEAD(state, n) (&(state)->generations[n].head)
174. young = GEN\_HEAD(state, generation);
175. // 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。
176. if (generation < NUM\_GENERATIONS-1)
177. //0、1代
178. old = GEN\_HEAD(state, generation+1);
179. else
180. //2代
181. old = young;
182. // 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。
183. // 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。
184. update\_refs(young); // gc\_prev is used for gc\_refs
185. // 处理循环引用，把循环引用的位置值为0.
186. subtract\_refs(young);
187. // 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）。
188. // 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；
189. gc\_list\_init(&unreachable);
190. move\_unreachable(young, &unreachable); // gc\_prev is pointer again
191. validate\_list(young, 0);
192. untrack\_tuples(young);
193. /\* Move reachable objects to next generation. \*/
194. // 将可达对象加入到下一代。
195. if (young != old) {
196. // 如果是0、1代，则升级到下一代。
197. if (generation == NUM\_GENERATIONS - 2) {
198. // 如果是1代，则更新
199. state->long\_lived\_pending += gc\_list\_size(young);
200. }
201. // 把young链表拼接到old链表中。
202. gc\_list\_merge(young, old);
203. }
204. else {
205. /\* We only untrack dicts in full collections, to avoid quadratic
206. dict build-up. See issue #14775. \*/
207. // 如果是2代，则更新long\_lived\_total和long\_lived\_pending
208. untrack\_dicts(young);
209. state->long\_lived\_pending = 0;
210. state->long\_lived\_total = gc\_list\_size(young);
211. }
212. // 循环所有不可达的元素，把具有 \_\_del\_\_ 方法对象放到finalizers链表中。
213. // 调用\_\_del\_\_之后，再会进行让他们在销毁。
214. gc\_list\_init(&finalizers);
215. // NEXT\_MASK\_UNREACHABLE is cleared here.
216. // After move\_legacy\_finalizers(), unreachable is normal list.
217. move\_legacy\_finalizers(&unreachable, &finalizers);
218. /\* finalizers contains the unreachable objects with a legacy finalizer;
219. \* unreachable objects reachable \*from\* those are also uncollectable,
220. \* and we move those into the finalizers list too.
221. \*/
222. move\_legacy\_finalizer\_reachable(&finalizers);
223. validate\_list(&finalizers, 0);
224. validate\_list(&unreachable, PREV\_MASK\_COLLECTING);
225. ...
226. /\* Clear weakrefs and invoke callbacks as necessary. \*/
227. // 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。
228. m += handle\_weakrefs(&unreachable, old);
229. validate\_list(old, 0);
230. validate\_list(&unreachable, PREV\_MASK\_COLLECTING);
231. /\* Call tp\_finalize on objects which have one. \*/
232. // 执行那些具有的\_\_del\_\_方法的对象。
233. finalize\_garbage(&unreachable);
234. // 最后，进行进行对垃圾的清除。
235. if (check\_garbage(&unreachable)) { // clear PREV\_MASK\_COLLECTING here
236. gc\_list\_merge(&unreachable, old);
237. }
238. else {
239. /\* Call tp\_clear on objects in the unreachable set. This will cause
240. \* the reference cycles to be broken. It may also cause some objects
241. \* in finalizers to be freed.
242. \*/
243. m += gc\_list\_size(&unreachable);
244. delete\_garbage(state, &unreachable, old);
245. }
246. /\* Collect statistics on uncollectable objects found and print
247. \* debugging information. \*/
248. for (gc = GC\_NEXT(&finalizers); gc != &finalizers; gc = GC\_NEXT(gc)) {
249. n++;
250. if (state->debug & DEBUG\_UNCOLLECTABLE)
251. debug\_cycle("uncollectable", FROM\_GC(gc));
252. }
253. if (state->debug & DEBUG\_STATS) {
254. double d = \_PyTime\_AsSecondsDouble(\_PyTime\_GetMonotonicClock() - t1);
255. PySys\_WriteStderr(
256. "gc: done, %" PY\_FORMAT\_SIZE\_T "d unreachable, "
257. "%" PY\_FORMAT\_SIZE\_T "d uncollectable, %.4fs elapsed\n",
258. n+m, n, d);
259. }
260. /\* Append instances in the uncollectable set to a Python
261. \* reachable list of garbage. The programmer has to deal with
262. \* this if they insist on creating this type of structure.
263. \*/
264. // 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。
265. handle\_legacy\_finalizers(state, &finalizers, old);
266. validate\_list(old, 0);
267. /\* Clear free list only during the collection of the highest
268. \* generation \*/
269. if (generation == NUM\_GENERATIONS-1) {
270. clear\_freelists();
271. }
272. ...
273. return n+m;
274. }

**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：

1. // Object/listobject.c
2. PyTypeObject PyList\_Type = {
3. PyVarObject\_HEAD\_INIT(&PyType\_Type, 0)
4. "list",
5. sizeof(PyListObject),
6. 0,
7. (destructor)list\_dealloc, /\* tp\_dealloc \*/
8. ...
9. PyObject\_GC\_Del, /\* tp\_free \*/
10. };
11. /\* Empty list reuse scheme to save calls to malloc and free \*/
12. #ifndef PyList\_MAXFREELIST
13. #define PyList\_MAXFREELIST 80
14. #endif
15. static PyListObject \*free\_list[PyList\_MAXFREELIST];
16. static int numfree = 0;
17. static void
18. list\_dealloc(PyListObject \*op)
19. {
20. Py\_ssize\_t i;
21. // 从分代回收的的代中移除
22. PyObject\_GC\_UnTrack(op);
23. Py\_TRASHCAN\_BEGIN(op, list\_dealloc)
24. if (op->ob\_item != NULL) {
25. /\* Do it backwards, for Christian Tismer.
26. There's a simple test case where somehow this reduces
27. thrashing when a \*very\* large list is created and
28. immediately deleted. \*/
29. i = Py\_SIZE(op);
30. while (--i >= 0) {
31. Py\_XDECREF(op->ob\_item[i]);
32. }
33. PyMem\_FREE(op->ob\_item);
34. }
35. if (numfree < PyList\_MAXFREELIST && PyList\_CheckExact(op))
36. // free\_list中还么有占满80，不销毁并缓冲在free\_list中
37. free\_list[numfree++] = op;
38. else
39. // 销毁并在refchain中移除
40. Py\_TYPE(op)->tp\_free((PyObject \*)op);
41. Py\_TRASHCAN\_END
42. }

**2.7 tuple类型**

**2.7.1 创建**

1. v = (11,22,33)

当创建元组时候，会执行如下源码：

1. // Objects/tupleobject.c
2. #define PyTuple\_MAXSAVESIZE 20 /\* Largest tuple to save on free list \*/
3. #define PyTuple\_MAXFREELIST 2000 /\* Maximum number of tuples of each size to save \*/
4. static PyTupleObject \*free\_list[PyTuple\_MAXSAVESIZE]; // free\_list[20] = {链表、链表..}
5. static int numfree[PyTuple\_MAXSAVESIZE]; // numfree[2000]表示每个链表的长度
6. PyObject \*
7. PyTuple\_New(Py\_ssize\_t size)
8. {
9. PyTupleObject \*op;
10. ...
11. // free\_list第0个元素存储的是空元祖
12. if (size == 0 && free\_list[0]) {
13. op = free\_list[0];
14. Py\_INCREF(op);
15. return (PyObject \*) op;
16. }
17. // 有缓存的tuple对象，则从free\_list中获取
18. if (size < PyTuple\_MAXSAVESIZE && (op = free\_list[size]) != NULL) {
19. // 获取对象并初始化
20. free\_list[size] = (PyTupleObject \*) op->ob\_item[0];
21. numfree[size]--;
22. Py\_SIZE(op) = size;
23. Py\_TYPE(op) = &PyTuple\_Type;
24. // 对象加入到refchain链表。
25. \_Py\_NewReference((PyObject \*)op);
26. }
27. else
28. {
29. ..
30. // 没有缓存数据，则创建对象
31. op = PyObject\_GC\_NewVar(PyTupleObject, &PyTuple\_Type, size);
32. if (op == NULL)
33. return NULL;
34. }
35. for (i=0; i < size; i++)
36. op->ob\_item[i] = NULL;
37. if (size == 0) {
38. free\_list[0] = op;
39. ++numfree[0];
40. Py\_INCREF(op); /\* extra INCREF so that this is never freed \*/
41. }
42. // 对象加入到分代的链表。
43. \_PyObject\_GC\_TRACK(op);
44. return (PyObject \*) op;
45. }
46. // Includes/objimpl.h
47. #define PyObject\_GC\_NewVar(type, typeobj, n) \
48. ( (type \*) \_PyObject\_GC\_NewVar((typeobj), (n)) )
49. Objects/gcmodules.c
50. PyVarObject \*
51. \_PyObject\_GC\_NewVar(PyTypeObject \*tp, Py\_ssize\_t nitems)
52. {
53. size\_t size;
54. PyVarObject \*op;
55. if (nitems < 0) {
56. PyErr\_BadInternalCall();
57. return NULL;
58. }
59. size = \_PyObject\_VAR\_SIZE(tp, nitems);
60. // 开内存 & 分代 & 超过阈值则垃圾回收（流程同上述 列表过程）
61. op = (PyVarObject \*) \_PyObject\_GC\_Malloc(size);
62. if (op != NULL)
63. op = PyObject\_INIT\_VAR(op, tp, nitems);
64. return op;
65. }

**2.7.2 引用**

1. v1 = (11,22,33)
2. 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. PyTypeObject PyTuple\_Type = {
2. PyVarObject\_HEAD\_INIT(&PyType\_Type, 0)
3. "tuple",
4. sizeof(PyTupleObject) - sizeof(PyObject \*),
5. sizeof(PyObject \*),
6. (destructor)tupledealloc, /\* tp\_dealloc \*/
7. ...
8. PyObject\_GC\_Del, /\* tp\_free \*/
9. };
10. static void
11. tupledealloc(PyTupleObject \*op)
12. {
13. Py\_ssize\_t i;
14. Py\_ssize\_t len = Py\_SIZE(op);
15. // 从分代的链表中移除
16. PyObject\_GC\_UnTrack(op);
17. Py\_TRASHCAN\_BEGIN(op, tupledealloc)
18. if (len > 0) {
19. i = len;
20. while (--i >= 0)
21. Py\_XDECREF(op->ob\_item[i]);
22. // 缓存到free\_list中
23. if (len < PyTuple\_MAXSAVESIZE && numfree[len] < PyTuple\_MAXFREELIST &&
24. Py\_TYPE(op) == &PyTuple\_Type)
25. {
26. op->ob\_item[0] = (PyObject \*) free\_list[len];
27. numfree[len]++;
28. free\_list[len] = op;
29. // 结束
30. goto done; /\* return \*/
31. }
32. }
33. // 不缓存，则直接销毁对象并在refchain链表中移除。
34. Py\_TYPE(op)->tp\_free((PyObject \*)op);
35. done:
36. Py\_TRASHCAN\_END
37. }

**2.8 dict类型**

**2.8.1 创建**

1. 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. #define PyDict\_MAXFREELIST 80
2. // 缓存dict对象的free\_list
3. static PyDictObject \*free\_list[PyDict\_MAXFREELIST];
4. static int numfree = 0;
5. PyObject \*
6. PyDict\_New(void)
7. {
8. dictkeys\_incref(Py\_EMPTY\_KEYS);
9. return new\_dict(Py\_EMPTY\_KEYS, empty\_values);
10. }
11. /\* Consumes a reference to the keys object \*/
12. static PyObject \*
13. new\_dict(PyDictKeysObject \*keys, PyObject \*\*values)
14. {
15. PyDictObject \*mp;
16. assert(keys != NULL);
17. // 如果有缓存，则从缓存区获取一个对象
18. if (numfree) {
19. mp = free\_list[--numfree];
20. assert (mp != NULL);
21. assert (Py\_TYPE(mp) == &PyDict\_Type);
22. \_Py\_NewReference((PyObject \*)mp);
23. }
24. else {
25. // 没有缓存，则去创建字典对象。（源码流程同list类型）
26. mp = PyObject\_GC\_New(PyDictObject, &PyDict\_Type);
27. if (mp == NULL) {
28. dictkeys\_decref(keys);
29. if (values != empty\_values) {
30. free\_values(values);
31. }
32. return NULL;
33. }
34. }
35. mp->ma\_keys = keys;
36. mp->ma\_values = values;
37. mp->ma\_used = 0;
38. mp->ma\_version\_tag = DICT\_NEXT\_VERSION();
39. ASSERT\_CONSISTENT(mp);
40. return (PyObject \*)mp;
41. }

**2.8.2 引用**

1. v1 = {"name":"武沛齐","age":18}
2. v2 = v1

出现引用，则应用计数器+1（同上）。

**2.8.3 销毁**

1. v1 = {"name":"武沛齐","age":18}
2. 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：

1. // Object/dictobject.c
2. PyTypeObject PyDict\_Type = {
3. PyVarObject\_HEAD\_INIT(&PyType\_Type, 0)
4. "dict",
5. sizeof(PyDictObject),
6. 0,
7. (destructor)dict\_dealloc, /\* tp\_dealloc \*/
8. ...
9. PyObject\_GC\_Del, /\* tp\_free \*/
10. };
11. static void
12. dict\_dealloc(PyDictObject \*mp)
13. {
14. PyObject \*\*values = mp->ma\_values;
15. PyDictKeysObject \*keys = mp->ma\_keys;
16. Py\_ssize\_t i, n;
17. // 从分代链表中移除
18. PyObject\_GC\_UnTrack(mp);
19. Py\_TRASHCAN\_BEGIN(mp, dict\_dealloc)
20. ...
21. // 缓存区为满，则缓存
22. if (numfree < PyDict\_MAXFREELIST && Py\_TYPE(mp) == &PyDict\_Type)
23. free\_list[numfree++] = mp;
24. else
25. // 已满则销毁，并在refchain中移除。
26. Py\_TYPE(mp)->tp\_free((PyObject \*)mp);
27. Py\_TRASHCAN\_END
28. }