Python源码剖析
- 数据结构篇

wklken

大纲

- 对象
- 类型
- int
- string
- list
- tuple
- dict

一切皆为对象

Python中对象是如何实现的?

对象 - 分类

- Python中对象分为两类: 定长(int等), 非定长(list/dict等)
- 源码中定义为PyObject和PyVarObject
- 两个定义都有一个共同的头部定义PyObject_HEAD

对象 - PyObject_HEAD

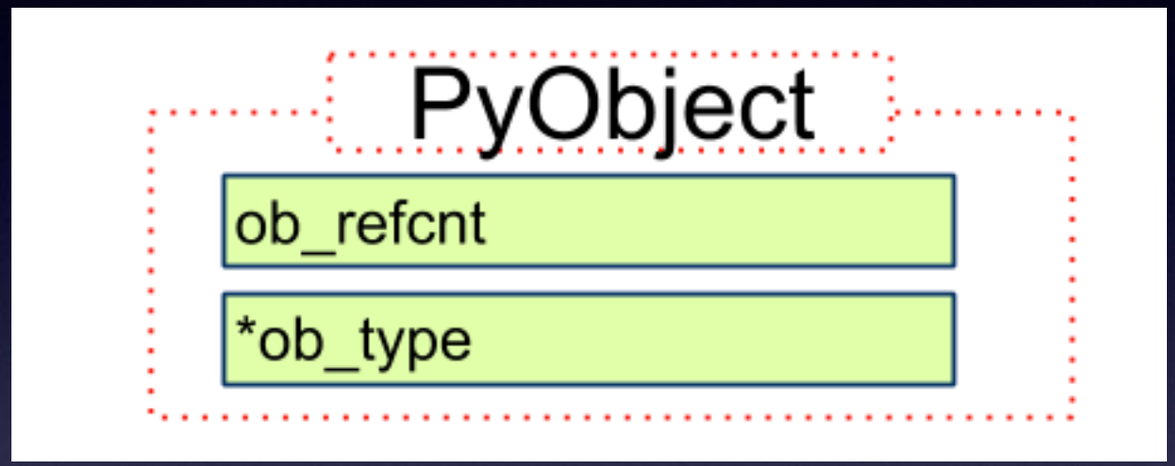
```
#define PyObject_HEAD                   \
        _PyObject_HEAD_EXTRA            \
        Py_ssize_t ob_refcnt;           \
        struct _typeobject *ob_type;
```

- _PyObject_HEAD_EXTRA　双向链表-垃圾回收涉及
- ob_refcnt 引用计数
- *ob_type　指向类型对象的指针, 决定了这个对象的类型!

- 获取引用计数的方式，sys.getrefcount, 以及为何数字不对?
- 获取类型的函数
-

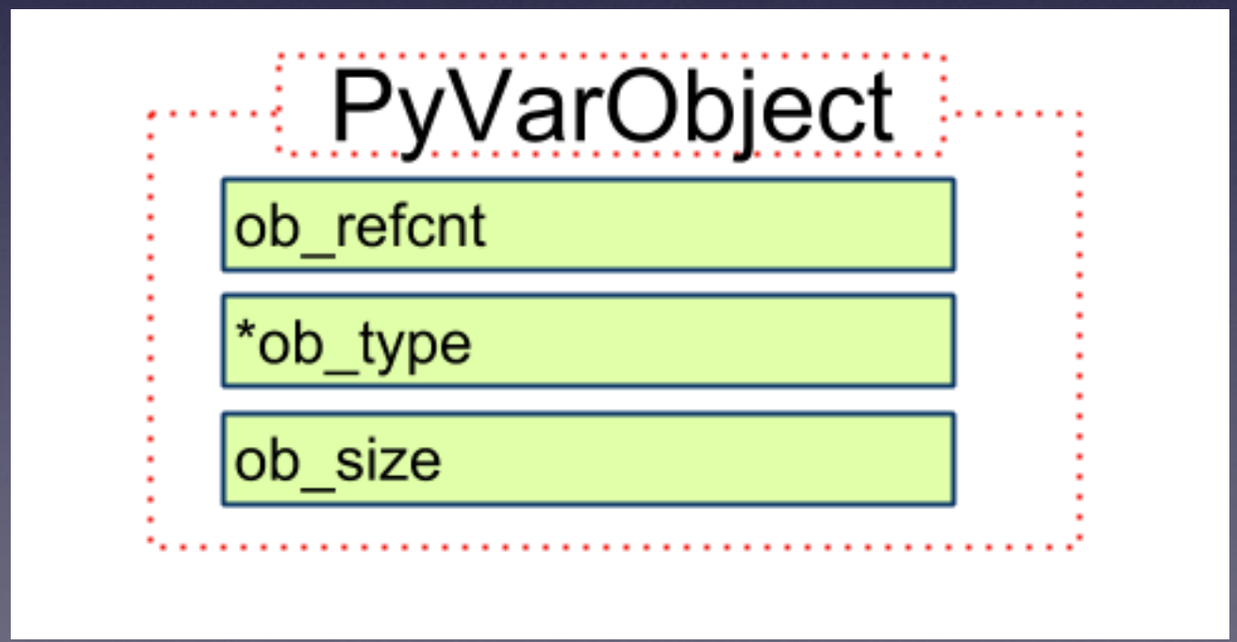
对象 - PyObject

```
typedef struct _object {
    PyObject_HEAD
} PyObject;
```

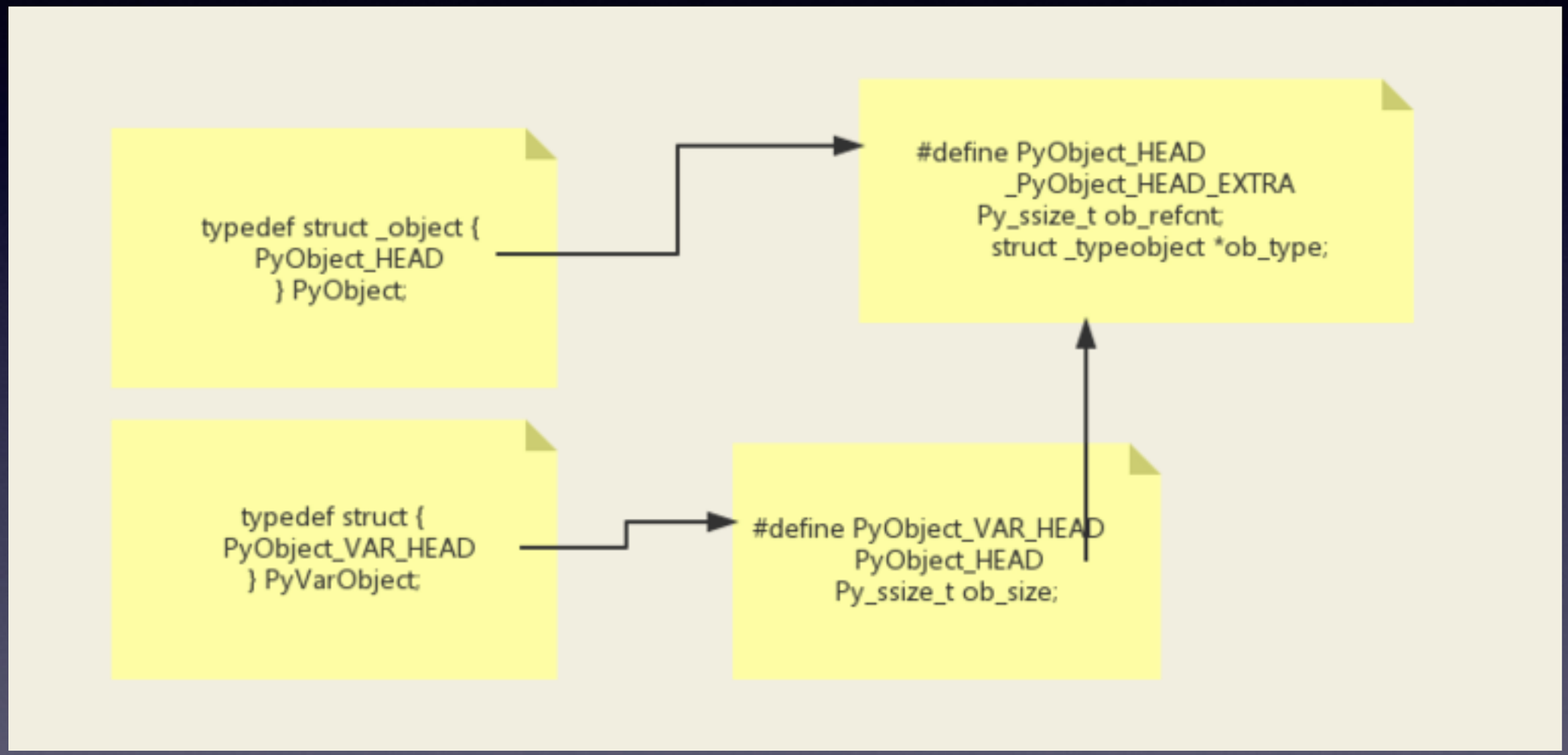


对象 - PyVarObject

```
typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;

#define PyObject_VAR_HEAD              \
  PyObject_HEAD                        \
  Py_ssize_t ob_size; /* Number of items in variable part */
```



对象 - 代码关系


typedef struct _object {
PyObject_HEAD
} PyObject;
#define PyObject_HEAD
_PyObject_HEAD_EXTRA
Py_ssize_t ob_refcnt;
struct _typeobject *ob_type;
typedef struct {
PyObject_VAR_HEAD
} PyVarObject;
#define PyObject_VAR_HEAD
PyObject_HEAD
Py_ssize_t ob_size;

对象 - 几个方法

#define Py_REFCNT(ob) (((PyObject*)(ob))->ob_refcnt)
读取引用计数

#define Py_TYPE(ob) (((PyObject*)(ob))->ob_type)
获取对象类型

#define Py_SIZE(ob) (((PyVarObject*)(ob))->ob_size)
读取元素个数(len)

Py_INCREF(op) 增加对象引用计数

Py_DECREF(op) 减少对象引用计数, 如果计数位0, 调用_Py_Dealloc

_Py_Dealloc(op) 调用对应类型的 tp_dealloc 方法(每种类型回收行为不一样的, 各种缓存池机制, 后面看)

物以类聚

一个例子

```
>>> a = 1
>>> a
1

>>> type(a)
<type 'int'>

#等价的两个
>>> type(type(a))
<type 'type'>
>>> type(int)
<type 'type'>

#还是等价的两个
>>> type(type(type(a)))
<type 'type'>
>>> type(type(int))
<type 'type'>
```

- 基本类型对象的 类型是type
- type 的类型是 type
-

类型 - PyTypeObject

反向推导一个int对象的生成来说明Python的类型机制

```
typedef struct _typeobject {
 /* MARK: base, 注意, 是个变长对象*/
  PyObject_VAR_HEAD
  const char *tp_name; /* For printing, in format "<module>.<name>" */ //类型名
  Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */ // 创建该类型对象时分配的内存空间大小

 // 一堆方法定义, 函数和指针
  /* Methods to implement standard operations */
  printfunc tp_print;
  hashfunc tp_hash;

  /* Method suites for standard classes */
  PyNumberMethods *tp_as_number;    // 数值对象操作
  PySequenceMethods *tp_as_sequence; // 序列对象操作
  PyMappingMethods *tp_as_mapping; // 字典对象操作

  // 一堆属性定义
  ....
} PyTypeObject;
```

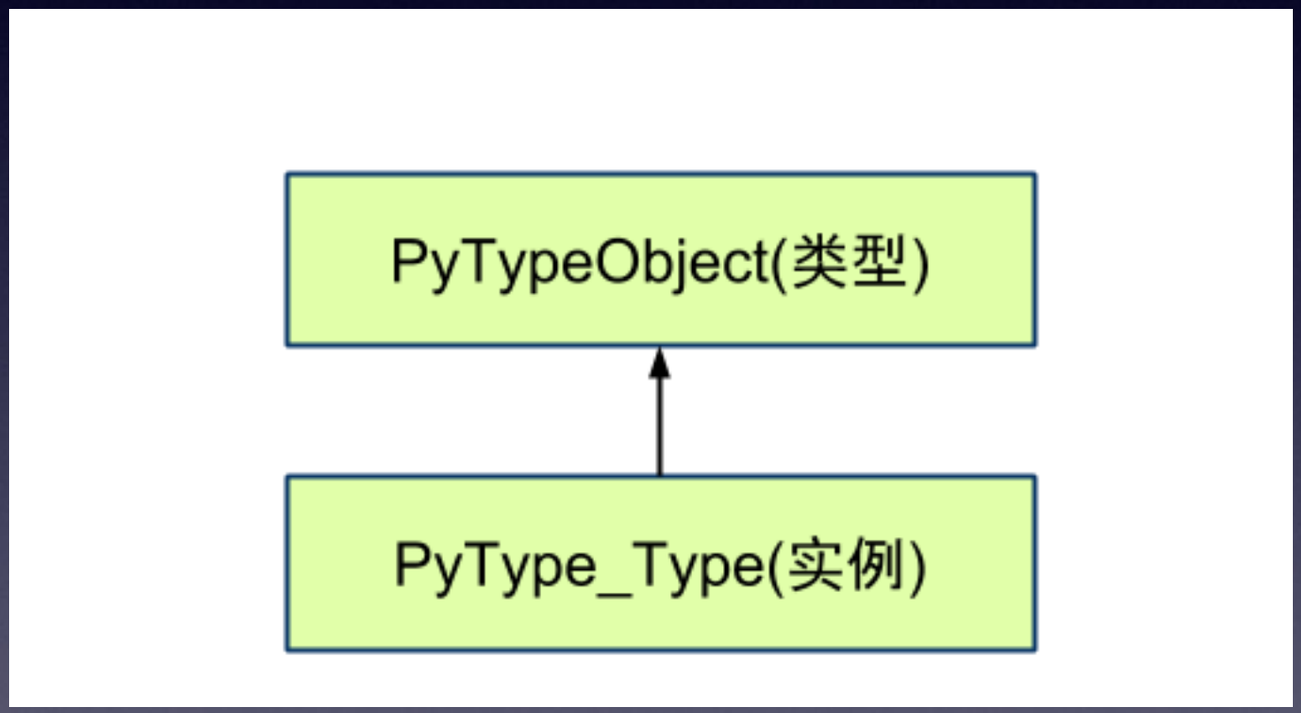
类型 - PyType_Type

```
PyTypeObject PyType_Type = {
  PyVarObject_HEAD_INIT(&PyType_Type, 0)
  "type",                                   /* tp_name */
  sizeof(PyHeapTypeObject),                 /* tp_basicsize */
  sizeof(PyMemberDef),                      /* tp_itemsize */
  (destructor)type_dealloc,                 /* tp_dealloc */

  // type对象的方法和属性初始化值
  .....

};
```

- 实例化, tp_name = ‘type’
- 注意, PyVarObject_HEAD_INIT

类型 - PyType_Type

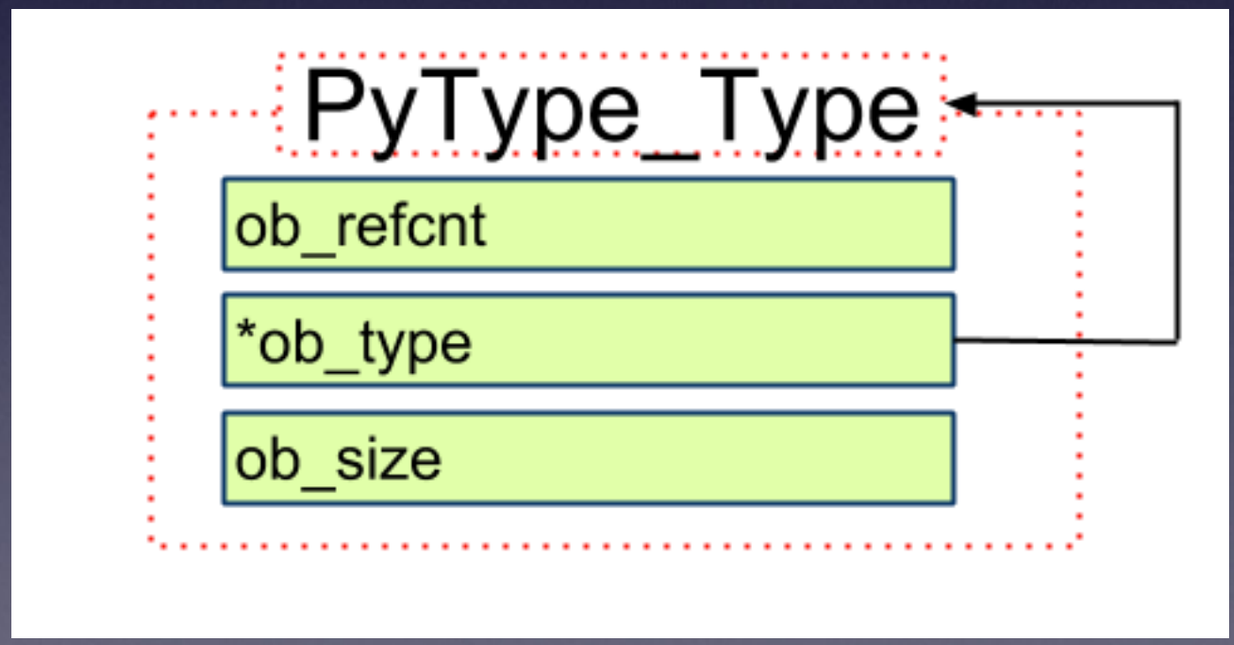
PyType_Type 是 PyTypeObject的实例化对象



类型 - PyType_Type

PyVarObject_HEAD_INIT, 这个方法在 Include/object.h中, 等价于

ob_refcnt = 1
*ob_type = &PyType_Type
ob_size = 0



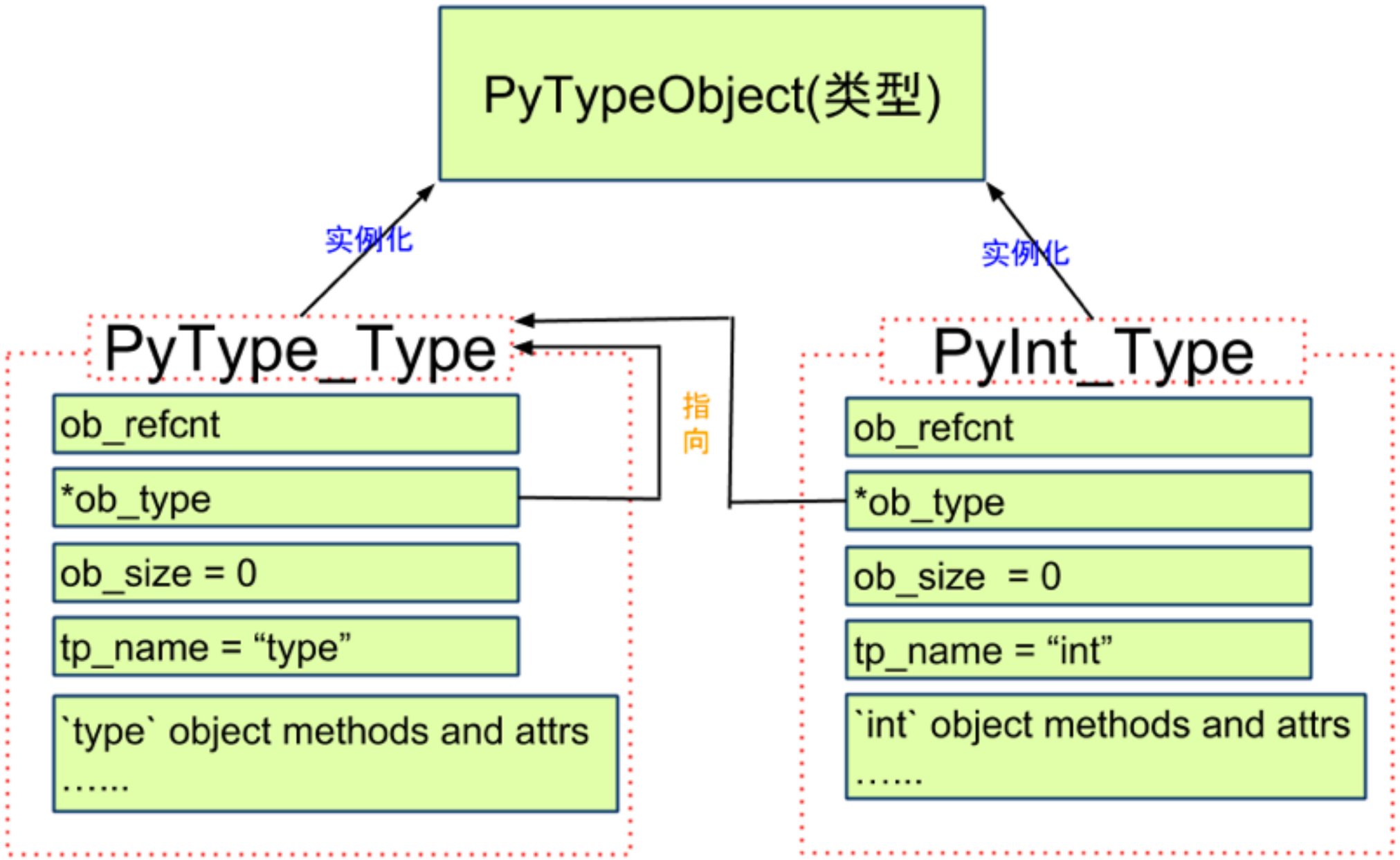
类型 - PyType_Type

```
# 1. int 的 类型 是`type`
>>> type(int)
<type 'type'>

# 2. type 的类型 还是`type`, 对应上面说明第二点
>>> type(type(int))
<type 'type'>
```

类型 - PyInt_Type

```
PyTypeObject PyInt_Type = {
  PyVarObject_HEAD_INIT(&PyType_Type, 0)
  "int",
  sizeof(PyIntObject),
  0,

  // int类型的相关方法和属性值
  ....

  (hashfunc)int_hash,                   /* tp_hash */

};
```

类型 - PyInt_Type



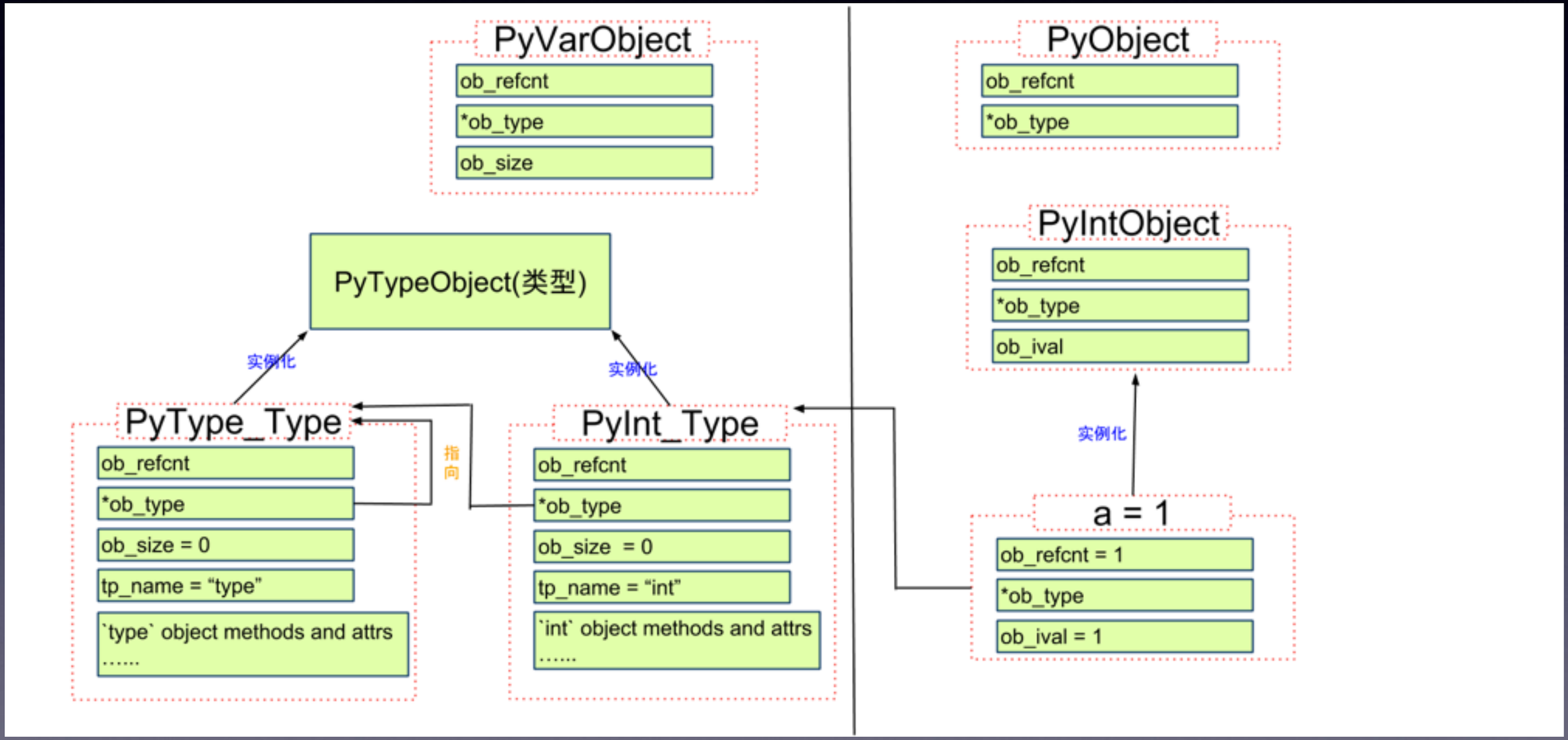
类型 - PyInt_Type

```
>>> type(1)
<type 'int'>

>>> type(type(1))
<type 'type'>
```

类型 - PyIntObject

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

类型 - 生成PyIntObject



类型 - 总结一下

1. 一切都是对象

2. PyType_Type / PyInt_Type / PyString_Type ....等
这些是`类型对象`, 可以认为是同级, 都是PyTypeObject这种`类型`的实例!

3. 虽然是同级,
但是其他PyXXX_Type, 其类型指向 PyType_Type
PyType_Type 的类型指向自己, 它是所有类型的`类型`

4. PyTypeObject 是一个变长对象

5. 每个object, 例如PyIntObject都属于一种`类型`
object初始化时进行关联

类型 - 多态的实现

```
>>> hash(1)
1
>>> hash("abc")
1453079729188098211
```

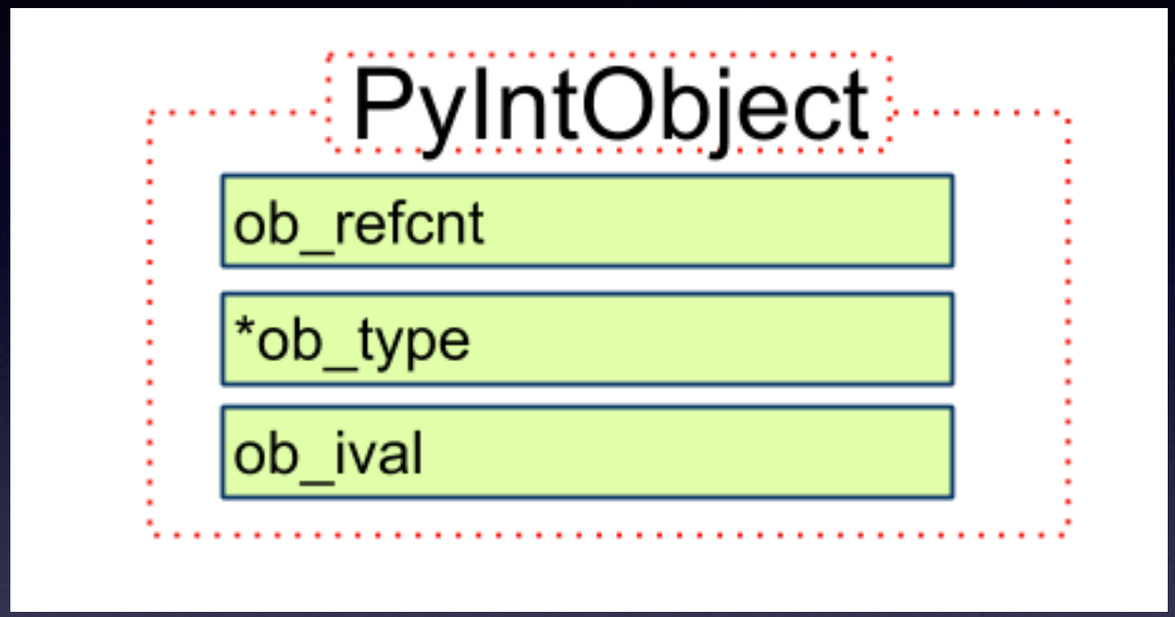
```
PyTypeObject PyInt_Type = {
    ...
    (hashfunc)int_hash,                 /* tp_hash */
    ...
}

PyTypeObject PyString_Type = {
    ...
    (hashfunc)string_hash,              /* tp_hash */
    ...
}
```

```
object -> ob_type -> tp_hash
```

INT

INT - PyIntObject

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```



INT - PyIntObject

```
>>> a = -5
>>> b = -5
>>> id(a) == id(b)
True
>>> a = -6
>>> b = -6
>>> id(a) == id(b)
False

>>> a = 256
>>> b = 256
>>> id(a) == id(b)
True
>>> a = 257
>>> b = 257
>>> id(a) == id(b)
False

#在python2.x中, 对于大的序列生成, 建议使用xrange(100000) 而不是range(100000), why?
```

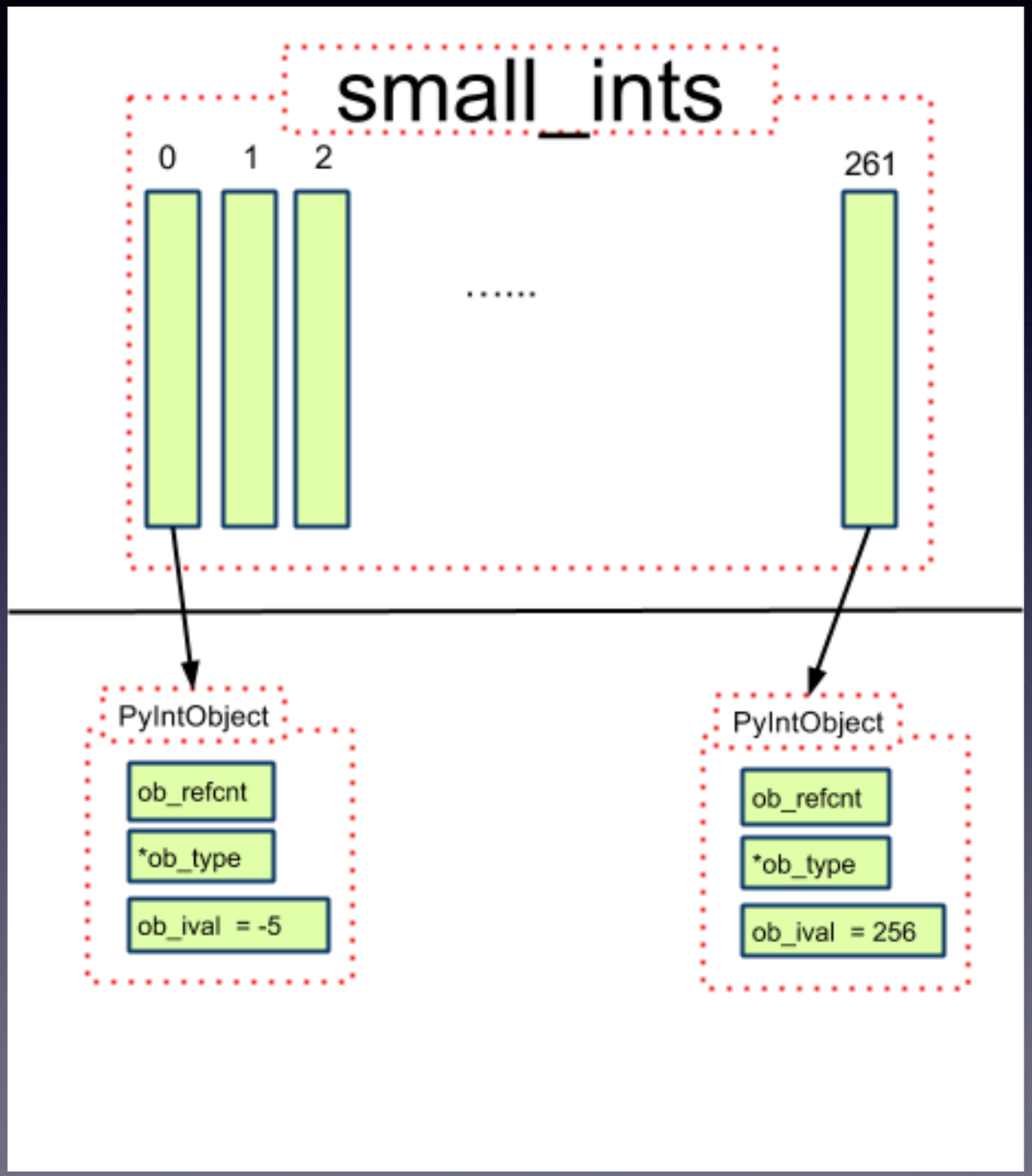
INT - 小整数对象缓冲池

```
#ifndef NSMALLPOSINTS
#define NSMALLPOSINTS           257
#endif

#ifndef NSMALLNEGINTS
#define NSMALLNEGINTS           5
#endif

#if NSMALLNEGINTS + NSMALLPOSINTS > 0
/* References to small integers are saved in this array
   so that they can be shared.
   The integers that are saved are those in the range
   -NSMALLNEGINTS (inclusive) to NSMALLPOSINTS (not inclusive).
*/

static PyIntObject *small_ints[NSMALLNEGINTS + NSMALLPOSINTS];
#endif
```

INT - 小整数对象缓冲池

小整数对象池就是一个PyIntObject指针数组(注意是指针数组), 大小=257+5=262, 范围是[-5, 257) 注意左闭右开.
即这个数组包含了262个指向PyIntObject的指针

INT - 小整数对象缓冲池



INT - 初始化

```
#if NSMALLNEGINTS + NSMALLPOSINTS > 0
if (-NSMALLNEGINTS <= ival && ival < NSMALLPOSINTS) {

    v = small_ints[ival + NSMALLNEGINTS];
     // 引用+1
     Py_INCREF(v);

    ......

     // 返回
     return (PyObject *) v;
}
#endif
```

INT - 通用整数对象缓冲池

小整数对象池, 只是一个指针数组, 其真正对象依赖通用整数对象池

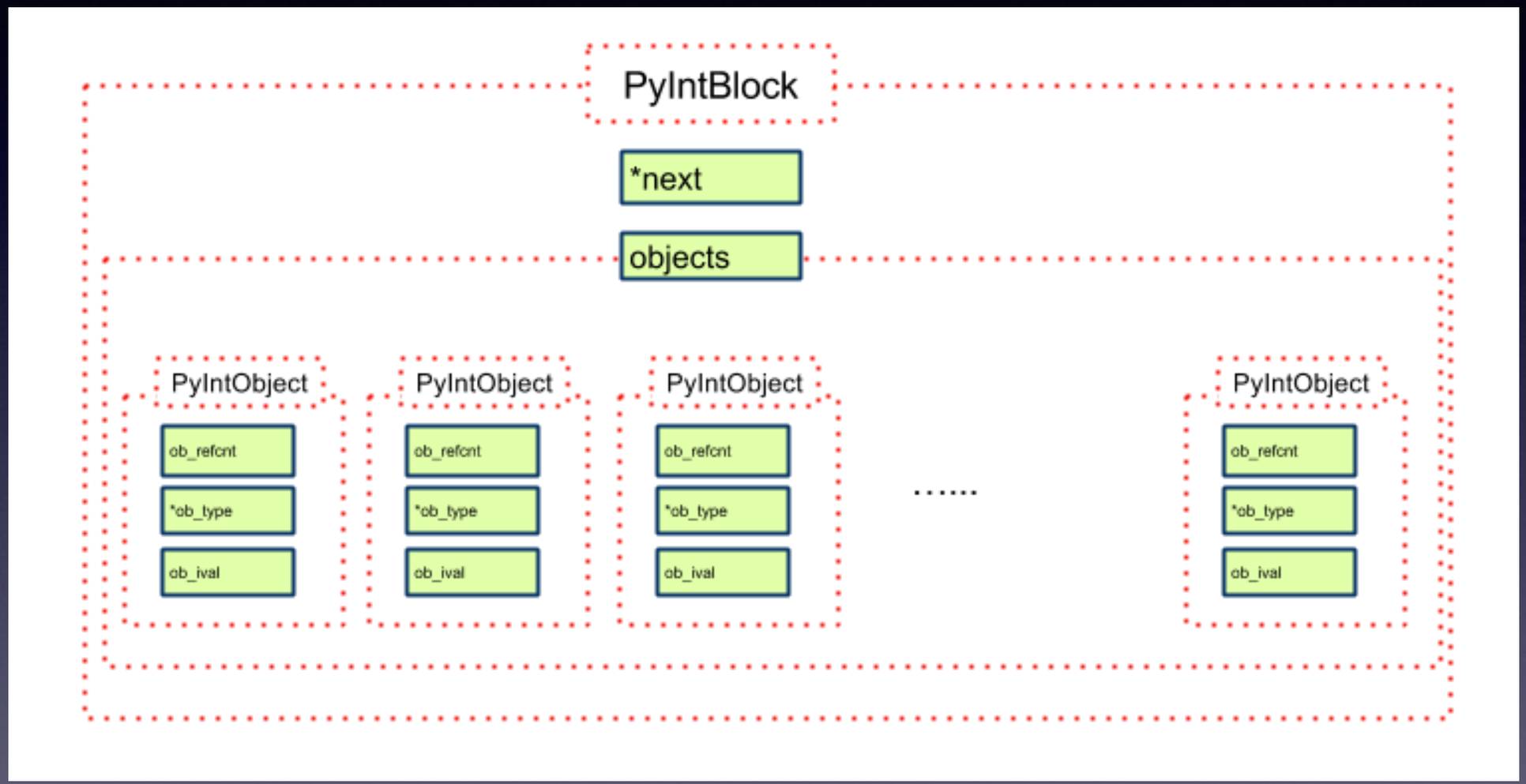
```
// 小整数对象池初始化过程, 循环, 逐一生成
for (ival = -NSMALLNEGINTS; ival < NSMALLPOSINTS; ival++) {
        if (!free_list && (free_list = fill_free_list()) == NULL)
                return 0;

    /* PyObject_New is inlined */
     v = free_list;
     free_list = (PyIntObject *)Py_TYPE(v);
     PyObject_INIT(v, &PyInt_Type);
     v->ob_ival = ival;

     // 放到数组里
     small_ints[ival + NSMALLNEGINTS] = v;
}
```

free_list

INT - 通用整数对象缓冲池

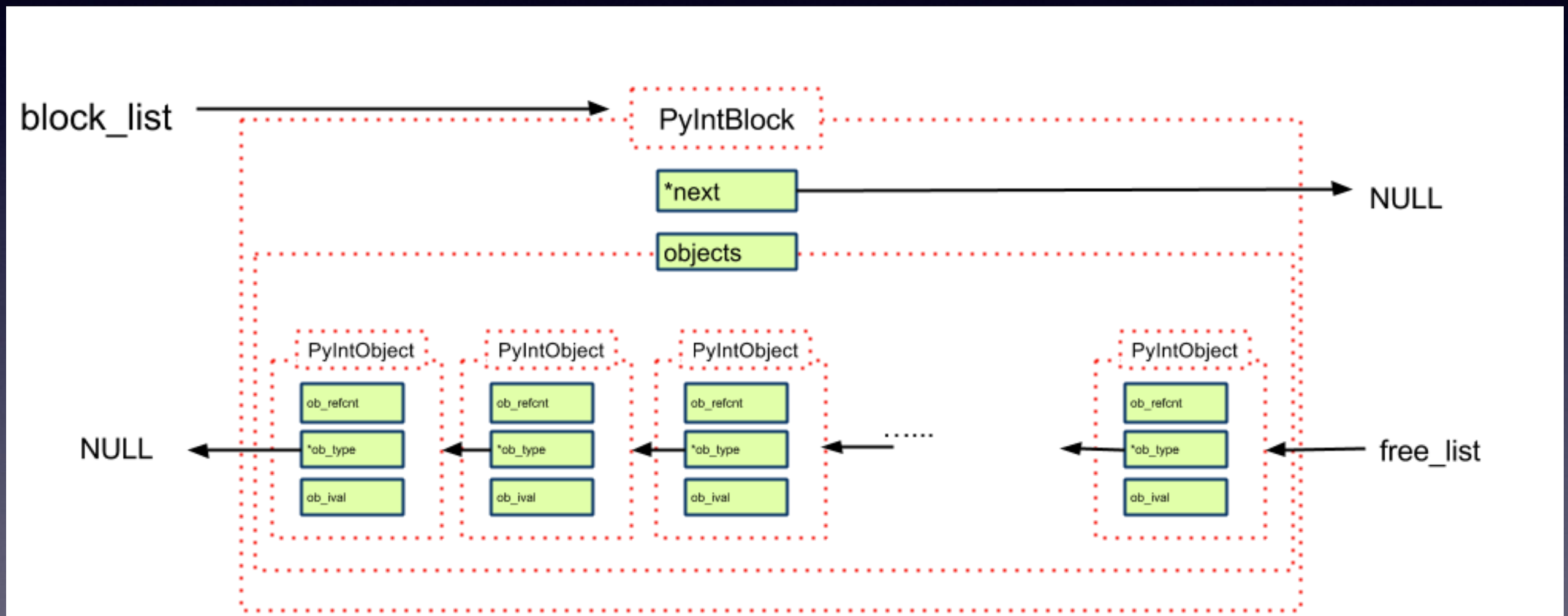
```
#define BLOCK_SIZE      1000    /* 1K less typical malloc overhead */
#define BHEAD_SIZE      8       /* Enough for a 64-bit pointer */
#define N_INTOBJECTS    ((BLOCK_SIZE - BHEAD_SIZE) / sizeof(PyIntObject))


struct _intblock {
    struct _intblock *next;
    PyIntObject objects[N_INTOBJECTS];
};

typedef struct _intblock PyIntBlock;
```

INT - 通用整数对象缓冲池



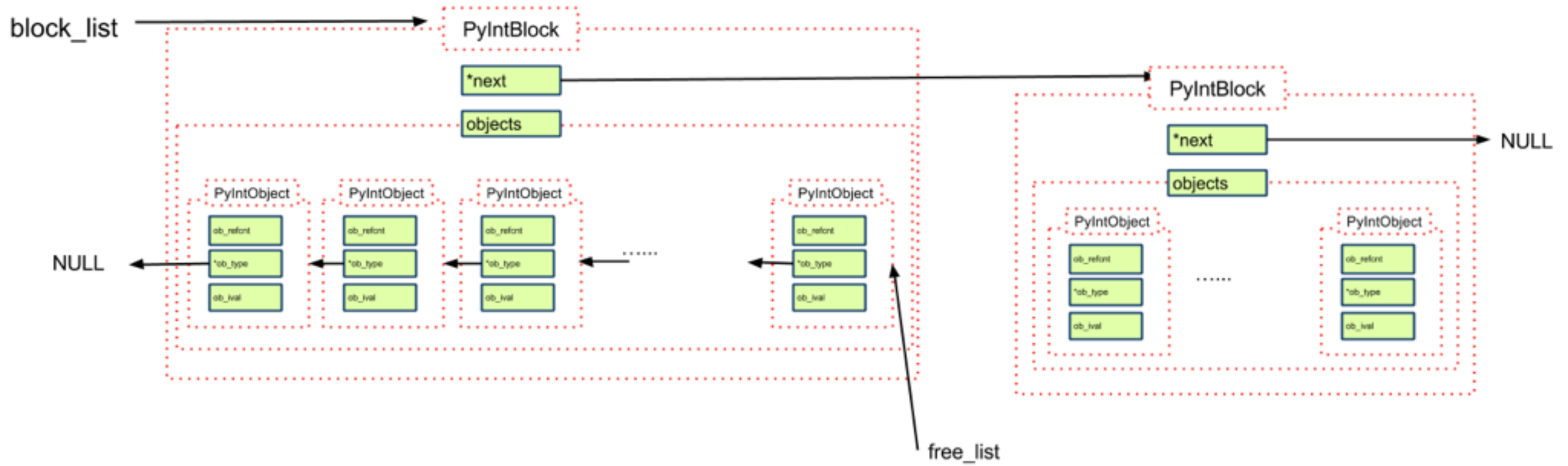
INT - 通用整数对象缓冲池

新建第一个的时候



INT - 通用整数对象缓冲池

当一个block用完了之后, 即free_list=NULL, 此时要新建另一个PyIntBlock



INT - 为什么要用xrange

int回收的时候, 把空间给放回到free_list

> PyIntBlocks are never returned to the system before shutdown (PyInt_Fini).

即, PyIntBlock申请的所有内存, 在Python解释器结束之前, 都不会被释放

所以, 使用range(100000), 一次性生成一个大的列表, 运行后, 虽然程序结束了, 但是整数占用空间还在.

xrange, 一次申请一个, 只会占用一个intobject, 之后回收, 再次申请一个.....

为什么python3直接使用range

`xrange` was not removed: it was renamed to `range`, and the 2.x `range` is what was removed.

INT - 为什么要用xrange

```
python -m memory_profiler test.py
```

```
Line #    Mem usage    Increment   Line Contents
================================================
     5      9.6 MiB      0.0 MiB   @profile
     6                             def test():
     7     40.7 MiB     31.1 MiB       for i in range(1000000):
     8     40.7 MiB      0.0 MiB           continue
```

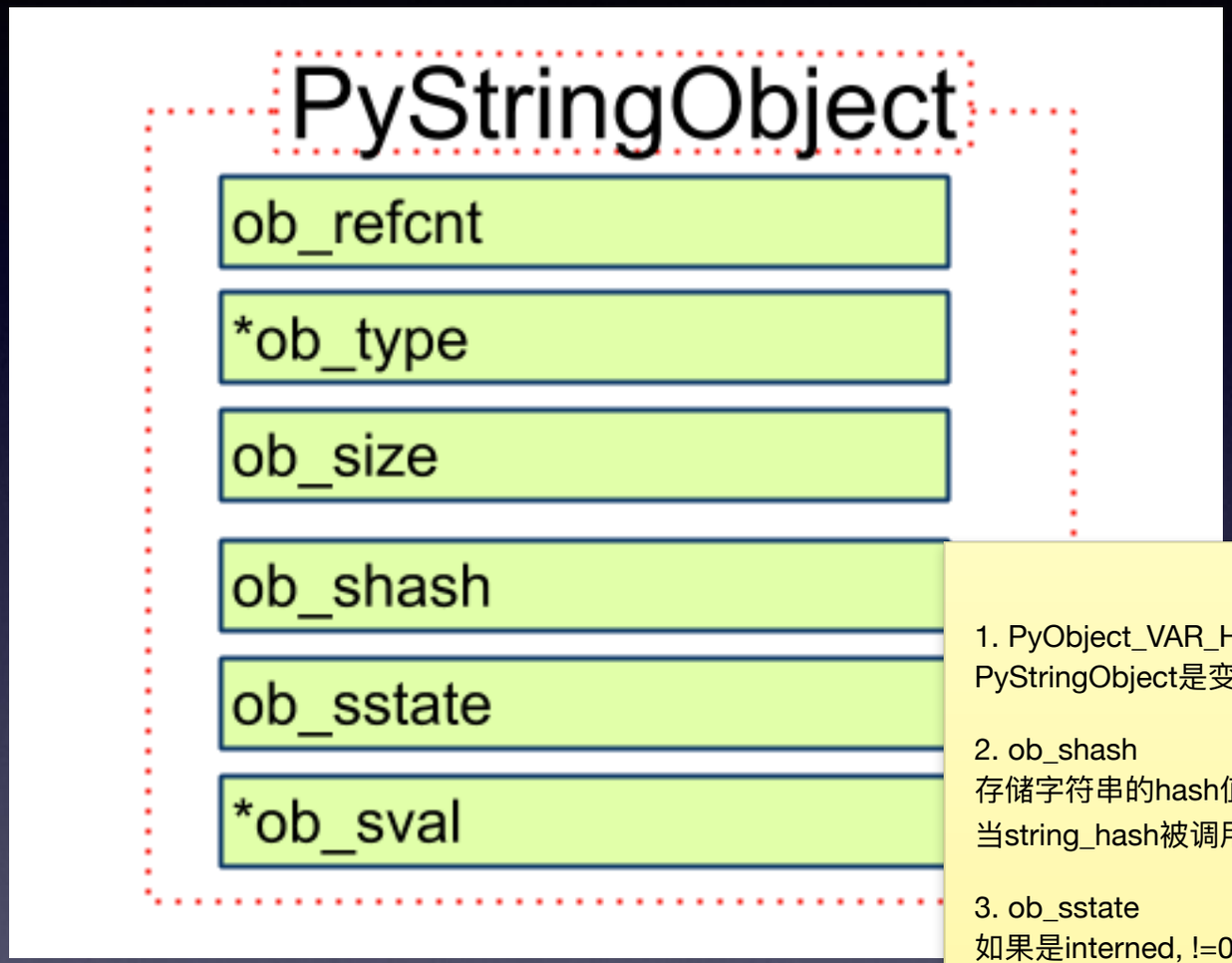
```
Line #    Mem usage    Increment   Line Contents
================================================
     5      9.7 MiB      0.0 MiB   @profile
     6                             def test():
     7      9.7 MiB      0.0 MiB       for i in xrange(1000000):
     8      9.7 MiB      0.0 MiB           continue
```

String

String - PyStringObject

```
typedef struct {
  PyObject_VAR_HEAD
  long ob_shash;
  int ob_sstate;
  char ob_sval[1];

  /* Invariants:
   *     ob_sval contains space for 'ob_size+1' elements.
   *     ob_sval[ob_size] == 0.
   *     ob_shash is the hash of the string or -1 if not computed yet.
   *     ob_sstate != 0 iff the string object is in stringobject.c's
   *       'interned' dictionary; in this case the two references
   *       from 'interned' to this object are *not counted* in ob_refcnt.
   */
} PyStringObject;
```

String - PyStringObject



1. PyObject_VAR_HEAD
PyStringObject是变长对象, 比定长对象多了一个ob_size字段

2. ob_shash
存储字符串的hash值, 如果还没计算等于-1
当string_hash被调用, 计算结果会被保存到这个字段一份, 后续不再进行计算

3. ob_sstate
如果是interned, !=0, 否则=0
interned后面说

4. char ob_sval[1]
字符指针指向一段内存, char数组指针, 指向一个ob_size+1大小数组(c中字符串最后要多一个字符`\0`表字符串结束)

String - interned机制

```
>>> a = "hello"
>>> b = "hello"
>>> id(a) == id(b)
True
>>>
>>> c = ''.join(['h', 'ello'])
>>> id(a) == id(c)
False
>>> a = "hello world"
>>> b = "hello world"
>>> id(a) == id(b)
False
>>>
>>> a = intern("hello world")
>>> b = intern("hello world")
>>> id(a) == id(b)
True
```

String - interned机制

/* This dictionary holds all interned strings. Note that references to strings in this dictionary are *not* counted in the string's ob_refcnt. When the interned string reaches a refcnt of 0 the string deallocation function will delete the reference from this dictionary.

Another way to look at this is that to say that the actual reference count of a string is: s->ob_refcnt + (s->ob_sstate?2:0)
*/
static PyObject *interned; //指针, 指向PyDictObject

String - interned机制

```
// 在interned字典中已存在, 修改, 返回intern独享
t = PyDict_GetItem(interned, (PyObject *)s);
if (t) {
    Py_INCREF(t);
    Py_DECREF(*p);
    *p = t;
    return;
}

// 在interned字典中不存在, 放进去
if (PyDict_SetItem(interned, (PyObject *)s, (PyObject *)s) < 0) {
    PyErr_Clear();
    return;
}
```

一旦字符串被intern, 会被python保存到字典中, 整个python运行期间, 系统中有且仅有一个对象. 下次相同字符串再进入, 不会重复创建对象.

String - interned机制

```
#define NAME_CHARS \
    "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz"
```

```
        if (!all_name_chars((unsigned char *)PyString_AS_STRING(v)))
            continue;
        PyString_InternInPlace(&PyTuple_GET_ITEM(consts, i));
```

什么情况才会走interned, 什么情况不interned TODO: find the code

只包含下划线、数字、字母的字符串才会被intern. 拼接产生的字符串不算

String - 字符缓冲池

```
UCHAR_MAX 平台相关

static PyStringObject *characters[UCHAR_MAX + 1];
```

字符缓冲池在使用中初始化(存在直接返回, 不存在建一个, 放interned字典中, 初始化字符缓冲池对应位置)

```
PyObject *t = (PyObject *)op;
// 走intern, 后面说
PyString_InternInPlace(&t);
op = (PyStringObject *)t;

// 初始化字符缓冲池对应位置
characters[*str & UCHAR_MAX] = op;
```

String - 性能相关

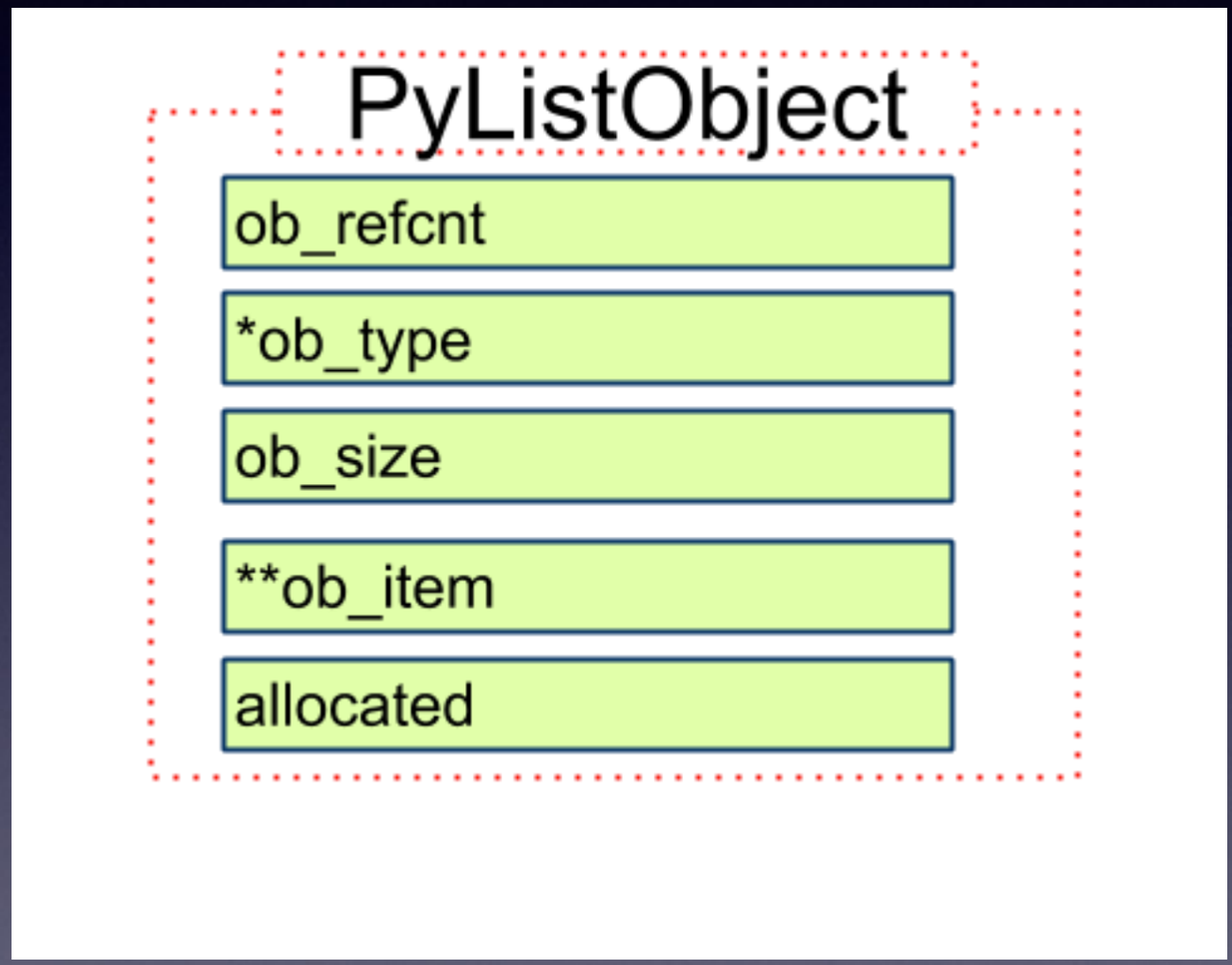
```
'a' + 'b' + 'c'

or

''.join(['a', 'b', 'c'])
```

string_concat, 一次加=分配一次内存空间, 拷贝两次. N次链接, 需要N-1次内存分配.
string_join, 计算序列所有元素字符串总长度, 用PyString_FromStringAndSize((char*)NULL, sz)分配内存空间, 然后逐一拷贝. 一次内存操作.

List

List - PyListObject

```
typedef struct {
    PyObject_VAR_HEAD

    PyObject **ob_item;

    Py_ssize_t allocated;
} PyListObject;
```



1. PyObject_VAR_HEAD
PyListObject是变长对象

2. PyObject **ob_item;
指向列表元素的指针数组, list[0] 即 ob_item[0]

3. Py_ssize_t allocated;
allocated列表分配的空间, ob_size为已使用的空间
allocated 总的申请到的内存数量
ob_size 实际使用内存数量

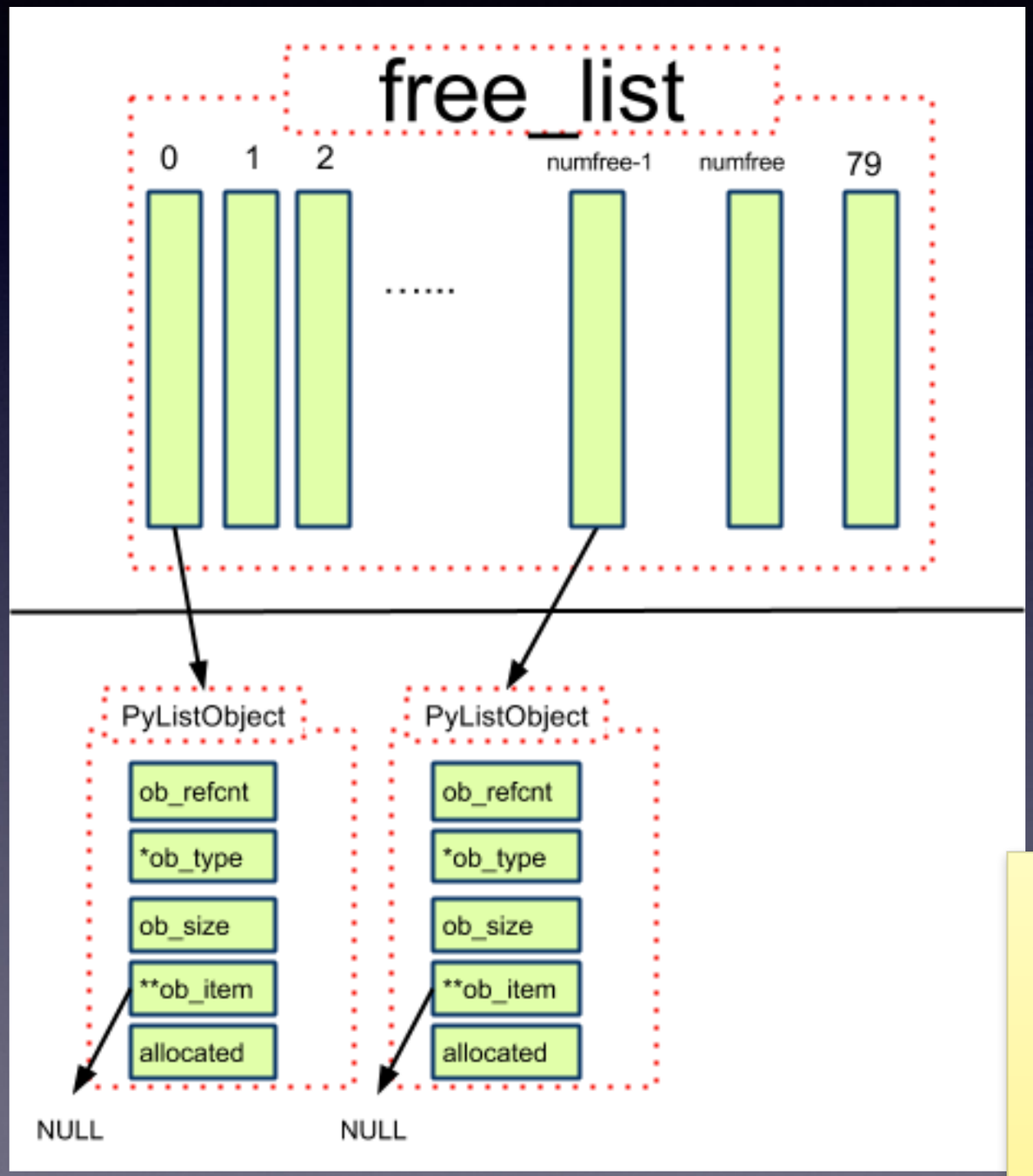
等式:

0 <= ob_size <= allocated

List - 对象缓冲池

```
/* Empty list reuse scheme to save calls to malloc and free */
#ifndef PyList_MAXFREELIST
#define PyList_MAXFREELIST 80
#endif

// 80个
static PyListObject *free_list[PyList_MAXFREELIST];

static int numfree = 0;
```

List - 对象缓冲池

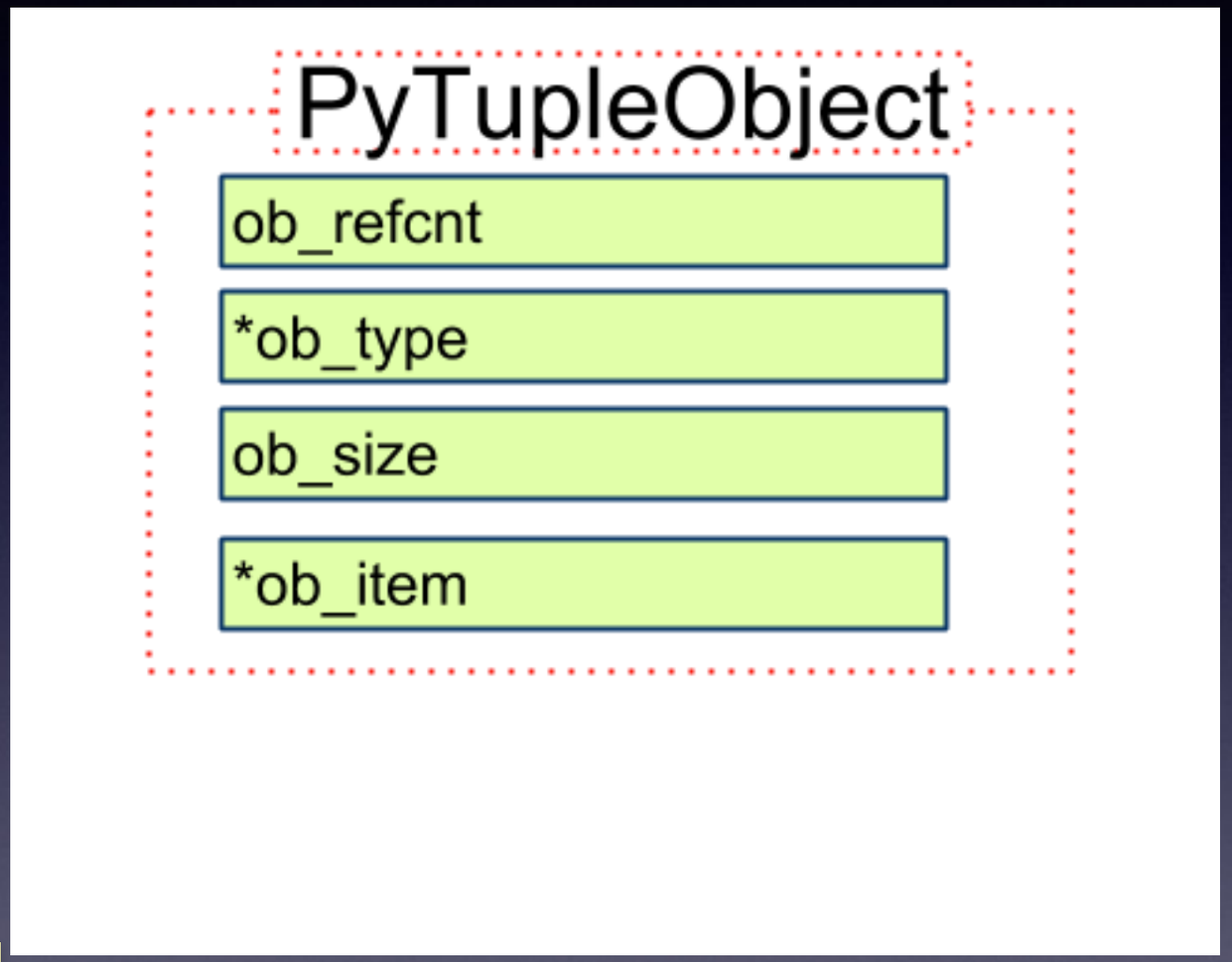


对一个列表对象PyListObject, 回收时, ob_item会被回收, 其每个元素指向的对象引用-1.
但是PyListObject对象本身, 如果缓冲池未满, 会被放入缓冲池, 复用

Tuple

Tuple - PyTupleObject

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject *ob_item[1];

} PyTupleObject;
```



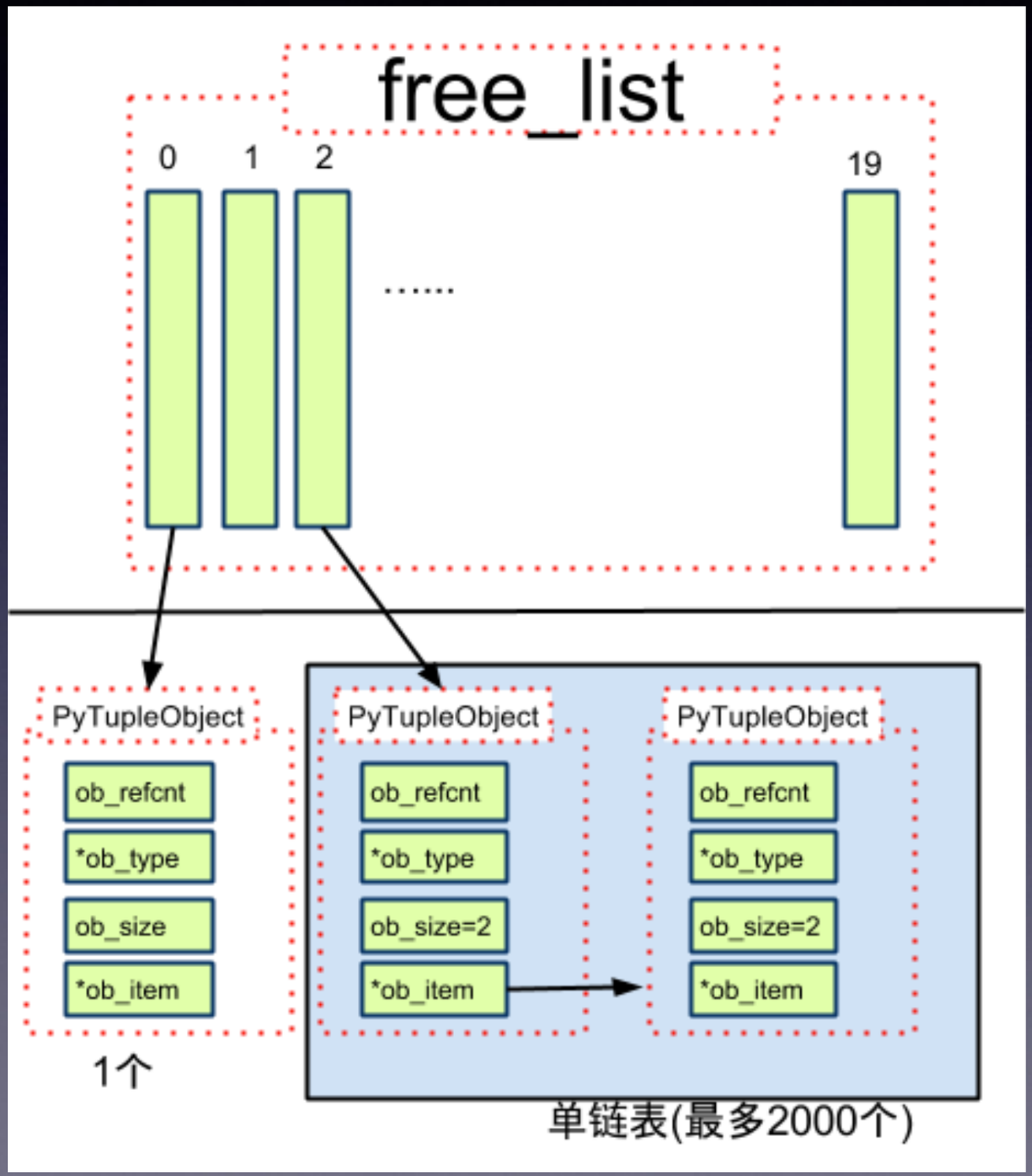
1. PyObject_VAR_HEAD
PyTupleObject在底层是个变长对象(需要存储列表元素个数).
虽然, 在python中, tuple是不可变对象

2. PyObject *ob_item[1];
指向存储元素的数组

Tuple - tuple对象缓冲池

```
/* Speed optimization to avoid frequent malloc/free of small tuples */
#ifndef PyTuple_MAXSAVESIZE
#define PyTuple_MAXSAVESIZE     20
#endif

#ifndef PyTuple_MAXFREELIST
#define PyTuple_MAXFREELIST  2000
#endif

#if PyTuple_MAXSAVESIZE > 0

static PyTupleObject *free_list[PyTuple_MAXSAVESIZE];
static int numfree[PyTuple_MAXSAVESIZE];
#endif
```

Tuple - tuple对象缓冲池



1. 作用: 优化小tuple的mall/free

2. PyTuple_MAXSAVESIZE = 20
会被缓存的tuple长度阈值, 20, 长度<20的, 才会走对象缓冲池逻辑

3. PyTuple_MAXFREELIST 2000
每种size的tuple最多会被缓存2000个

4. PyTupleObject *free_list[PyTuple_MAXSAVESIZE]
free_list, 指针数组, 每个位置, 存储了指向一个单链表头的地址

5. numfree[PyTuple_MAXSAVESIZE]
numfree, 一个计数数组, 存储free_list对应位置的单链表长度

6. free_list[0], 指向空数组, 有且仅有一个

Dict

Dict - 存储策略

1. 使用散列表进行存储

2. 使用开放定址法处理冲突

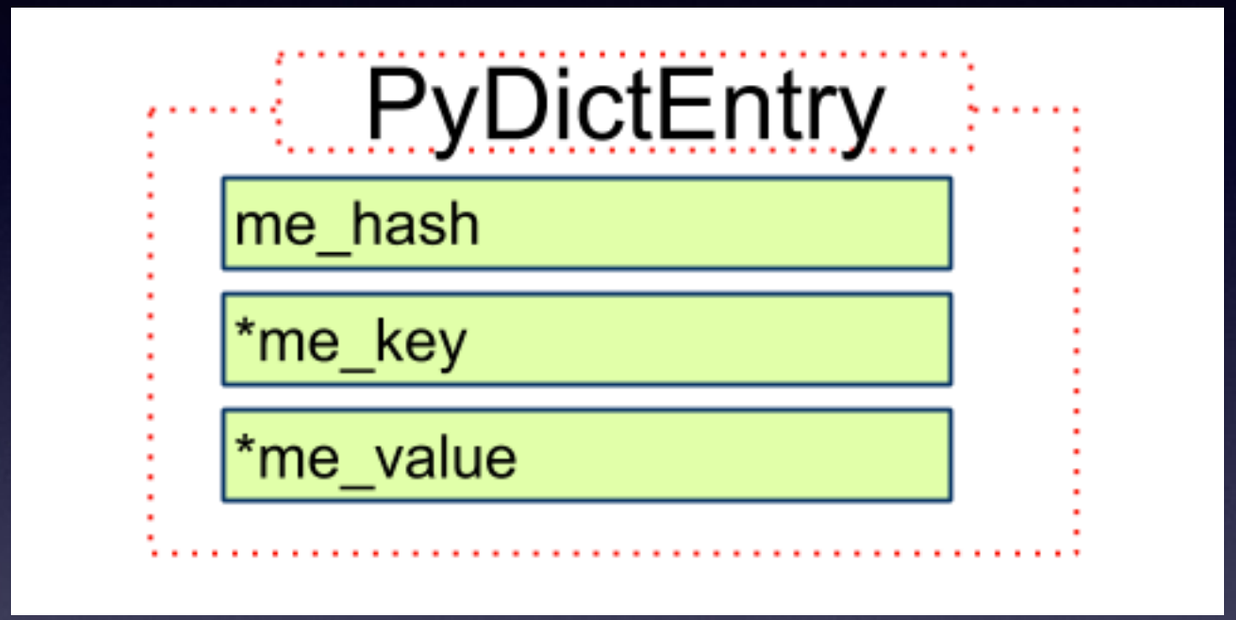
    2.1 插入, 发生冲突, 通过二次探测算法, 寻找下一个位置, 直到找到可用位置, 放入(形成一条冲突探测链)

    2.2 查找, 需要遍历冲突探测链

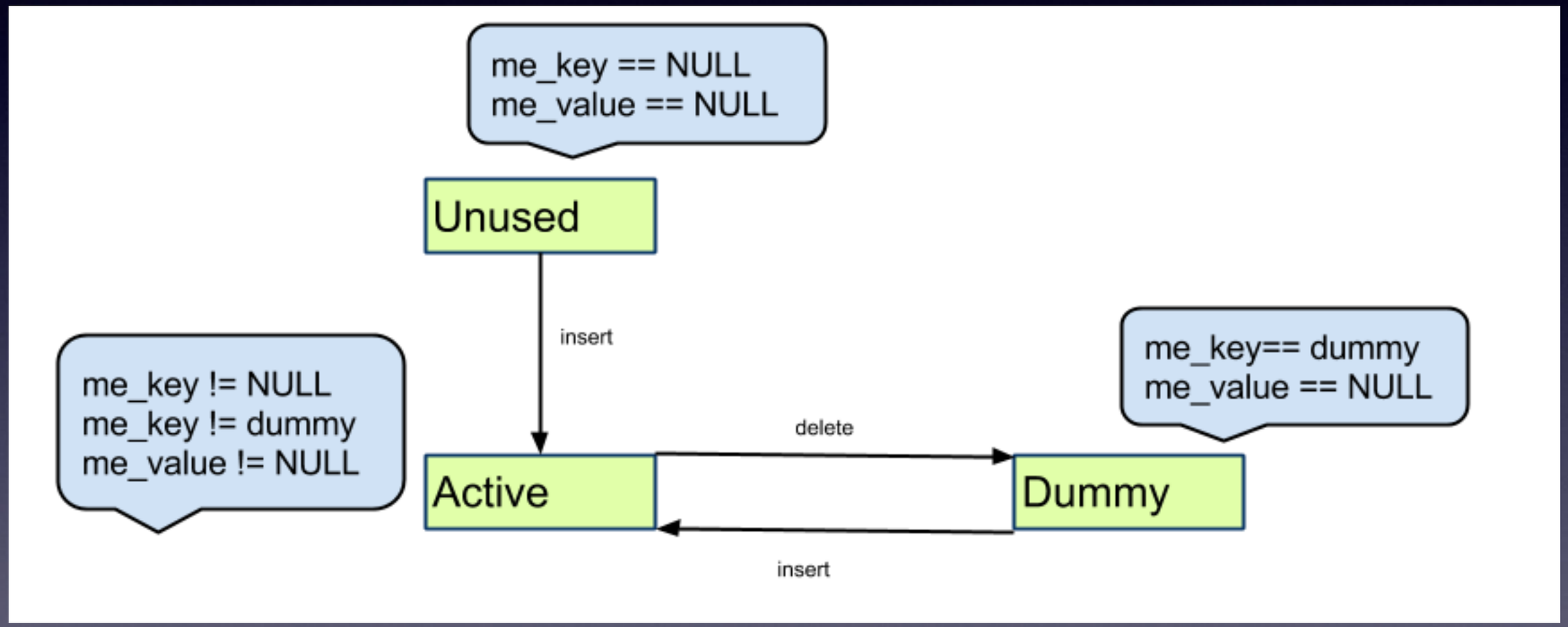
    2.3 删除, 如果对象在探测链上, 不能直接删除, 否则会破坏整个结构(所以不是真的删)

Dict - PyDictEntry

```
typedef struct {
    Py_ssize_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictEntry;
```



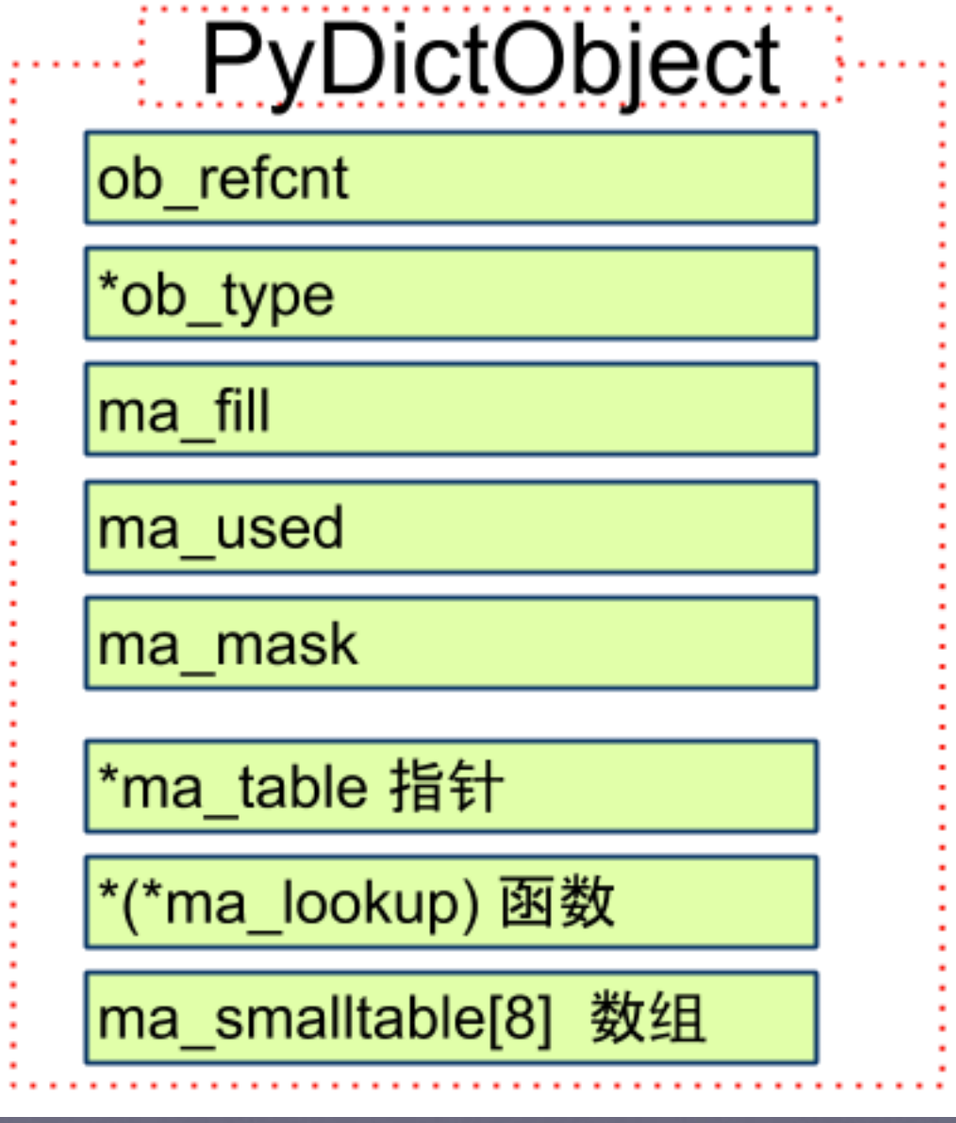
Dict - PyDictEntry



Dict - PyDictObject

```
typedef struct _dictobject PyDictObject;
struct _dictobject {
    PyObject_HEAD

    Py_ssize_t ma_fill;
    Py_ssize_t ma_used;
    Py_ssize_t ma_mask;

    PyDictEntry *ma_table;
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject *key, long hash);
    PyDictEntry ma_smalltable[PyDict_MINSIZE];
};
```

Dict - PyDictObject



1. PyObject_HEAD
反而声明为定长对象, 因为ob_size在这里用不上, 使用ma_fill和ma_used计数

2. Py_ssize_t ma_fill;
   Py_ssize_t ma_used;

   ma_fill = # Active + # Dummy
   ma_used = # Active

3. Py_ssize_t ma_mask;
散列表entry容量 = ma_mask + 1, 初始值ma_mask = PyDict_MINSIZE - 1 = 7

   ma_mask + 1 = # Unused + # Active + # Dummy

4. PyDictEntry *ma_table;
指向散列表内存, 如果是小的dict(entry数量<=8). 指向ma_smalltable数组

5. ma_lookup
搜索函数

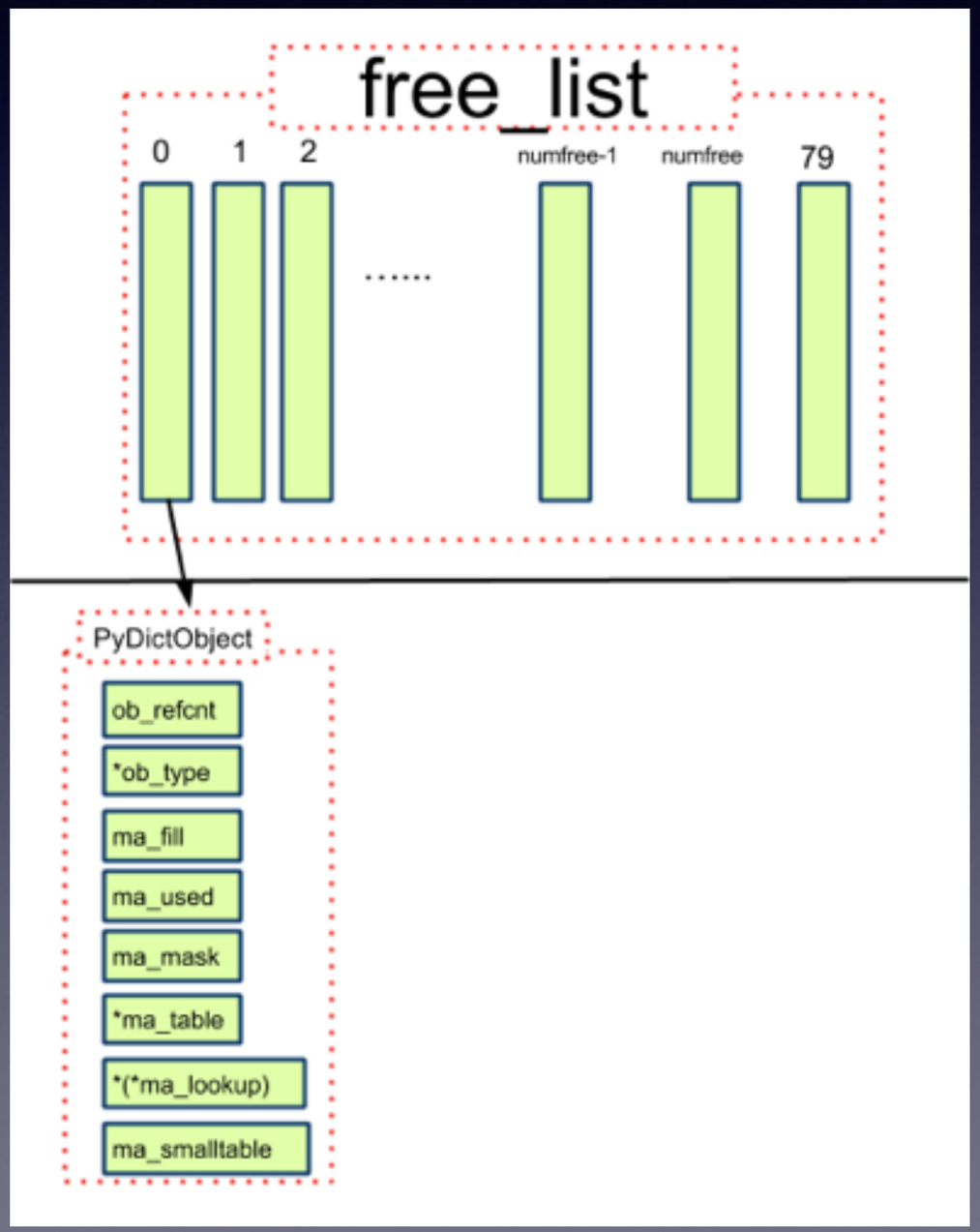
Dict - PyDictObject

1. PyDictObject在生命周期内, 需要维护ma_fill/ma_used/ma_mask 三个计数值

2. 初始化创建是ma_smalltable, 超过大小后, 会使用外部分配的空间

Dict - 对象缓冲池

对象缓冲池的结构(跟PyListObject对象缓冲池结构基本一样)



参考

参考

- Python源码剖析
- Python2.7.8源码

Q & A

后续

- 垃圾回收机制
- 内存管理机制
- 闭包的实现
- 虚拟机架构
- 虚拟机 - 一般表达式/控制流/函数机制/类机制
- 运行环境初始化
- 编译机制
- 执行机制
- 多线程模型
- 模块加载机制

Thanks