《源码探秘 CPython》31. 元组是怎么实现的?

原创 古明地觉 古明地觉的编程教室 2022-02-17 09:00

收录于合集
#CPython
97个 >

微信扫一扫
关注该公众号

楔子

下面介绍一下元组，元组的实现机制非常简单，可以理解为**不支持元素添加、修改、删除等操作的列表**，也就是在列表的基础上移除了**增删改**操作。

所以从功能上来讲，元组只是列表的子集，那元组存在的意义是什么呢？其实，元组存在的最大一个特点就是，它可以作为字典的 key、以及可以作为集合的元素。因为字典和集合存储数据的原理是哈希表，它存储的元素一定是可哈希的，关于字典和集合我们后续章节会说。

列表可以动态改变，但哈希值却是一开始就计算好的，所以列表不支持哈希。因为支持动态修改的话，那么哈希值肯定会变，这是不允许的。因此当我们希望字典的key是一个序列时，显然元组再适合不过了。

比如要根据**年龄和身高**统计人数，那么就可以将**年龄和身高**组成元组作为字典的key、人数作为字典的value。所以元组可哈希、能够作为哈希表的key，是元组存在的最大意义。

> 当然啦，元组如果可哈希，那么元组存储的元素必须都是可哈希的。只要有一个元素不可哈希，那么元组就会不可哈希。
>
> 比如元组里面存储了一个列表，由于列表不可哈希，导致存储了列表的元组也变得不可哈希。

元组的底层结构

根据我们使用元组的经验，可以得出元组是一个**变长对象**，但同时又是一个**不可变对象**。

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

以上是元组在底层对应的结构体，一个引用计数、一个类型、一个ob_size、一个指针数组。然后里面的数组长度 1，我们可以当成 n 来用，在PyLongObject的时候说过。

然后我们通过结构体的定义，来看看它和列表之间的区别。首先它没有**allocated**、也就是**容量**的概念，这是因为它是不可变的，不支持resize操作。

另一个区别就是元组对应的指针数组是定义在结构体里面的，可以直接对数组进行操作；而列表对应的指针数组是定义在结构体外面的，两者通过**ob_item字段**进行关联，也就是通过一个二级指针来间接操作指针数组。

至于为什么要这么定义，我们在最开始介绍对象模型的时候也说的很详细了，可以回头看一看，主要原因就是一个是可变对象、一个是不可变对象。

元组是怎么创建的?

元组支持的操作我们就不看了，因为它只支持查询操作，并且和列表是高度相似的。这

里我们直接来看元组的创建过程。

正如列表一样，Python创建PyTupleObject也提供了类似的初始化方法。

```
PyObject *
PyTuple_New(Py_ssize_t size)
{
    //PyTupleObject指针
    PyTupleObject *op;
    Py_ssize_t i;
    if (size < 0) {
        PyErr_BadInternalCall();
        return NULL;
    }
#if PyTuple_MAXSAVESIZE > 0
    //元组同样有缓存池
    //这部分逻辑一会在介绍缓存池的时候详细说
    if (size == 0 && free_list[0]) {
        //创建的元组长度为0
        //那么获取缓存池中索引为0的对象(指针)
        op = free_list[0];
        Py_INCREF(op);
#ifdef COUNT_ALLOCS
        //介绍缓存池的时候解释
        tuple_zero_allocs++;
#endif
        //返回
        return (PyObject *) op;
    }
    if (size < PyTuple_MAXSAVESIZE && (op = free_list[size]) != NULL) {
        //当size<PyTuple_MAXSAVESIZE的时候
        //也是从缓存池中获取
        //这里的逻辑在介绍缓存池的时候细说
        free_list[size] = (PyTupleObject *) op->ob_item[0];
        numfree[size]--;
#ifdef COUNT_ALLOCS
        fast_tuple_allocs++;
#endif
        /* Inline PyObject_InitVar */
#ifdef Py_TRACE_REFS
        //设置ob_size，和ob_type
        Py_SIZE(op) = size;
        Py_TYPE(op) = &PyTuple_Type;
#endif
        //引用计数初始化为1
        _Py_NewReference((PyObject *)op);
    }
    else
#endif
    {
    //到这里说明没有从缓存池中获取，那么要重新申请内存
    //元组的元素个数同样有限制，但我们说这个限制一般达不到
        if ((size_t)size > ((size_t)PY_SSIZE_T_MAX - sizeof(PyTupleObjec
t) -
                    sizeof(PyObject *)) / sizeof(PyObject *)) {
            return PyErr_NoMemory();
        }
        //申请空间
        op = PyObject_GC_NewVar(PyTupleObject, &PyTuple_Type, size);
        if (op == NULL)
            return NULL;
    }
    //将每一个元素都是设置为NULL
    for (i=0; i < size; i++)
```

```
61          op->ob_item[i] = NULL;
62  #if PyTuple_MAXSAVESIZE > 0
63      if (size == 0) {
64          free_list[0] = op;
65          ++numfree[0];
66          Py_INCREF(op);
67      }
68  #endif
69  #ifdef SHOW_TRACK_COUNT
70      count_tracked++;
71  #endif
72      _PyObject_GC_TRACK(op);
73      //返回
74      return (PyObject *) op;
    }
```

以上就是元组创建的过程，但里面隐藏了很多的细节没有说，下面我们来介绍元组的缓存池，然后将细节一一揭开。

元组的缓存池

元组的缓存池也是通过数组来实现的：

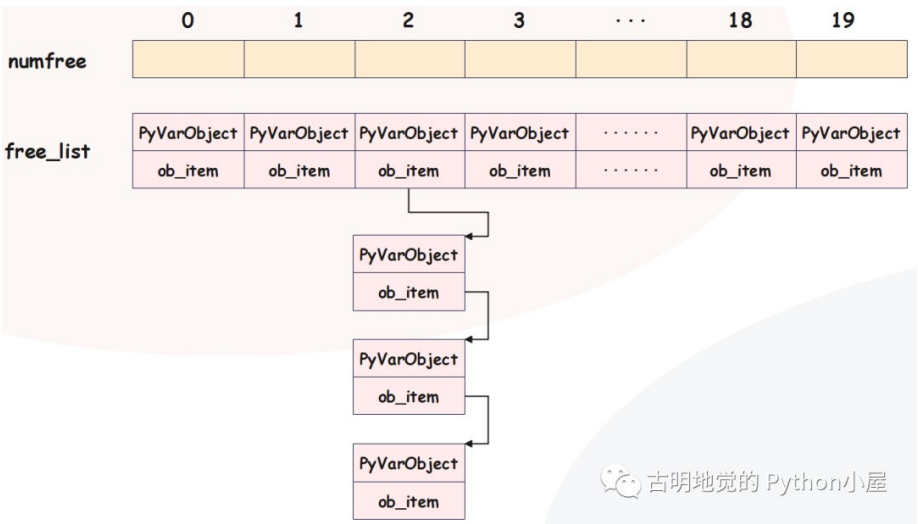
```
1  #define PyTuple_MAXSAVESIZE 20
2  static PyTupleObject *free_list[PyTuple_MAXSAVESIZE];
3  static int numfree[PyTuple_MAXSAVESIZE];
```

从定义中可以看到，元组的缓存池大小是 20，而我们之前介绍的列表的缓存池大小是80。但是这里的20和80还稍稍有些不同，80指的是列表缓存池的大小，除此之外没有别的含义；而20除了表示元组缓存池的大小之外，它还表示只有当元组的长度小于20，回收时才会被放入缓存池。

当元组的长度为 n 时(其中 n < 20)，那么在回收的时候该元组就会放在缓存池索引为 n 的位置。假设回收的元组长度为6，那么就会放在缓存池索引为 6 的位置。

但是问题来了，如果我要回收两个长度为6的元组该怎么办？所以free_list里面虽然是**PyTupleObject ***，但是每个**(PyTupleObject *)->ob_item[0]** 都存储了下一个**PyTupleObject ***。

因此你可以认为**free_list**里面有20条链表，每条链表上面挂着相同长度的PyTupleObject的指针。并且每条链表的长度小于 2000，而这个长度由元组对应的**numfree**维护，它也是一个数组。



这里再来重新捋一下，元组的缓存池是一个数组，并且索引为**n**的位置回收的是长度也

为**n**的元组(指针)，并且n小于20。但这样的话，具有相同长度的元组不就只能缓存一个了？比如我们有很多个长度为2的元组都要缓存怎么办呢？

显然将它们以链表的形式串起来即可，正如图中显示的那样。至于长度为n的元组究竟缓存了多少个呢？则由**numfree[n]**负责维护。假设**free_list[2]**这个链表上挂了1000个**PyTupleObject ***，那么numfree[6]就等于1000。

当我再回收一个长度为6的元组时，那么会让该元组的**ob_item[0]**等于free_list[6]，然后free_list[6]等于该元组、numfree[6]++。所以这里的每一条链表和**浮点数缓存池**是类似的，也是采用的头插法。

```
//元组的析构函数
static void
tupledealloc(PyTupleObject *op)
{
    //......
    if (len > 0) {
        //......
#if PyTuple_MAXSAVESIZE > 0
    //len表示元组的长度
    //numfree[len]表示该链表上挂了多少个长度为len的元组
    //如果元组的长度小于20，并且对应的链表长度小于2000
        if (len < PyTuple_MAXSAVESIZE &&
            //PyTuple_MAXFREELIST 是一个宏，值为 2000
            numfree[len] < PyTuple_MAXFREELIST &&
            Py_TYPE(op) == &PyTuple_Type)
        {
        //将回收的元组的第一个元素设置为 free_list[len]
            op->ob_item[0] = (PyObject *) free_list[len];
        //维护链表的长度
            numfree[len]++;
        //再将free_list[len]设置为该元组
        //因此该元组成为了新的 free_list[len]
        //所以采用的头插法
            free_list[len] = op;
            goto done; /* return */
        }
#endif
    }
    //不满足条件，直接释放内存
    Py_TYPE(op)->tp_free((PyObject *)op);
done:
    Py_TRASHCAN_END
}
```

然后我们再回过头看一下PyTuple_New这个函数，重新解释一下里面的细节。

```
PyObject *
PyTuple_New(Py_ssize_t size)
{
    //......
#if PyTuple_MAXSAVESIZE > 0
    //回收的元组的长度为0时比较特殊，一会单独说
    if (size == 0 && free_list[0]) {
        //......
    }
    //当0<size<20时，直接通过op = free_list[size]从缓存池获取
    if (size < PyTuple_MAXSAVESIZE && (op = free_list[size]) != NULL) {
    //元组取走后，别忘记让free_list[size]指向下一个元素
    //也就是(PyTupleObject *) op->ob_item[0]
        free_list[size] = (PyTupleObject *) op->ob_item[0];
    //维护对应的链表长度
        numfree[size]--;
```

```
#ifdef COUNT_ALLOCS
        fast_tuple_allocs++;
#endif
        /* Inline PyObject_InitVar */
#ifdef Py_TRACE_REFS
        //设置ob_size，和ob_type
        Py_SIZE(op) = size;
        Py_TYPE(op) = &PyTuple_Type;
#endif
        //引用计数初始化为1
        _Py_NewReference((PyObject *)op);
    }
   //......
}
```

到此，相信你已经明白元组的缓存池到底是怎么一回事了，说白了就是有20个链表，索引为**n**的链表存放长度为**n**的元组，因此可回收的元组的最大长度是19。

并且每条链表的长度小于2000，也就是具有相同长度的元组最多回收大概2000个。然后是链表的**next**指针，它由元组的**ob_item[0]**来充当，通过**ob_item[0]**来获取下一个元素。

```
>>> tpl = (1, 2, 3)
>>> print(id(tpl))
2279295395264
>>>
>>> del tpl  # 放入缓存池
>>>
>>> tpl = ("古明地觉", "古明地恋", "芙兰朵露")
>>> print(id(tpl))
2279295395264
>>>
```

我们看到打印的地址是一样的，因为第一次创建的元组被重复利用了。

需要注意的是，上面的代码必须在交互式环境下执行，如果放在 py 文件里面执行，那么打印的地址是不同的。因为元组在编译的时候就已经确定了，关于这方面的细节后续会详细介绍，这里可以先提一下。

Python虽然是解释型语言，但它也有一个编译的过程，在编译时静态的常量会提前分配好内存，我们以浮点数为例：

```
pi = 3.14
print(id(pi))  # 2093943985904
del pi

e = 2.71
print(id(e))  # 2093943985840
```

咦，浮点数有缓存池机制啊，为啥打印的地址不一样呢？其实原因就是，通过 py 文件的方式执行，这个py文件会作为一个整体进行编译。而在编译的时候，这两个浮点数就已经被分配好了，所以不会等到运行时再分配，至于原因显然是为了效率。如果将上面的代码改一下：

```
pi = float("3.14")
print(id(pi))  # 1687876048624
del pi

e = float("2.71")
print(id(e))  # 1687876048624
```

此时是一个函数调用，只能在运行的时候动态执行，所以两个浮点数都是运行时动态分配的。由于代码从上往下、一行一行执行，因此在**del pi**之后，指向的浮点数会被缓存，创建**e**的时候，对象会从缓存池里面获取。

而采用交互式的方式，结果也是类似的：

```
>>> pi = 3.14
>>> id(pi)
1917403367952
>>> del pi
>>> e = 2.71
>>> id(e)
1917403367952
```

此时地址也是一样的，说明缓存池生效了。原因就是交互式环境下，解释器在执行完一条语句之后，不知道你下一行语句会输入什么，所以每一行的可执行语句都是一个独立的编译单元。因此在**pi=3.14**时，浮点数已被创建，在**del pi**之后，指向的浮点数会被缓存。那么创建**e**的时候，对象自然会从缓存池里面获取。

而我们通过小括号创建的元组，它也会在编译时就分配内存。和列表不同，列表只会在运行时动态构建，所以之前在介绍列表缓存池的时候，我们是以py文件的方式演示的。当然不光是列表，集合、字典都是运行时才会创建。

而元组比较特殊，虽然它也是一种复合结构，但如果它内部的元素都是整数、 浮点数、字符串等静态常量，那么该元组本身也会作为静态常量在编译时就被收集起来。而这么做的原因就是，元组的使用频率远比我们想象的广泛，主要是它大量使用在我们看不到的地方。比如多元赋值：

```
a, b = 1, 2
```

在编译时，上面的**1， 2**实际上是作为元组被加载的，整个赋值相当于元组的解包。元组的第一个元素给变量a、第二个元素给变量b。

再比如函数、方法的返回值，如果是多返回值，本质上也是包装成一个元组之后再返回。所以元组的使用频率如此之广泛，那么为了运行效率，解释器会在编译时就为其分配好内存。当然前提是，元组内部的元素都是静态常量，否则也只能留到运行时。至于列表，不管内部元素如何，都会在运行时创建。

而且元组缓存池的每条链表上能回收2000个元组，这要远大于其它对象的缓存池容量，而原因还是由于元组会被大量创建。可以想象一个大型项目，函数、方法只要是多返回值，就会涉及到元组的创建，因此2000是很合理的。

此外元组和列表还有一个区别，那就是列表在被回收时，它的指针数组会被释放；但元组不同，它在被回收时，底层的指针数组会保留，并且还巧妙地通过索引来记录了回收的元组的大小规格。

元组的这项技术也被称为**静态资源缓存**，因为元组在执行析构函数时，**不仅对象本身没有被回收，连底层的指针数组也被缓存起来了**。那么当再次分配时，速度就会快一些。

```
from timeit import timeit

t1 = timeit(stmt="x1 = [1, 2, 3, 4, 5]", number=1000000)
t2 = timeit(stmt="x2 = (1, 2, 3, 4, 5)", number=1000000)

print(round(t1, 2))  # 0.05
print(round(t2, 2))  # 0.01
```

可以看到用时，元组只是列表的五分之一。这便是元组的另一个优势，可以将资源缓存起来。而缓存的原因还是如上面所说，因为涉及大量的创建和销毁，所以这一切都是为了加快内存分配。

> 由于对象都在堆区，为了效率，Python不得不大量使用缓存的技术。

最后再回答上面遗漏的部分，当元组长度为0的情况。我们说元组长度为0到19时，都会对应一条链表，链表上面能容纳的元素个数小于2000。这一点其实不太严谨，准确的说应该是长度为1到19。

如果元组长度为 0，那么它对应的链表只会容纳一个元素，这也说明了不管我们创建多少个空元组，最终只会创建一个。

```
tpl1 = ()
tpl2 = ()
tpl3 = ()

print(id(tpl1) == id(tpl2) == id(tpl3))  # True
```

再来看看PyTuple_New这个函数：

```
PyObject *
PyTuple_New(Py_ssize_t size)
{
    //......
#if PyTuple_MAXSAVESIZE > 0
    if (size == 0 && free_list[0]) {
        //从缓存池中获取之后只是增加了引用计数
        op = free_list[0];
        Py_INCREF(op);
#ifdef COUNT_ALLOCS
        //维护空元组被分配的次数
        tuple_zero_allocs++;
#endif
        //返回
        return (PyObject *) op;
    }
    //......
}
```

所以空元组可以认为是单例的，只有一份。

小结

以上就是元组相关的内容，因为有了列表相关的经验，再来看元组就会快很多。当然啦，元组的一些操作我们没有说，因为和对应的列表操作是类似的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》32. 初识哈希表

下一篇 ›
《源码探秘 CPython》30. 列表的创建与销毁，以及缓存池机制

喜欢此内容的人还喜欢

python 7天进阶之路-对象和json转换
缪斯之子

[系列]微服务·深入理解 gRPC - Part2
走向架构师的每一天

Abaqus python脚本开发 第三章 各类指令的方法对象变量 (3)
山石结构