《源码探秘 CPython》11. 整数是怎么设计的，为什么它不会溢出？

原创 古明地觉 古明地觉的编程教室 2022-01-14 09:30

收录于合集
#CPython
97个 >

楔子

这次我们来分析一下Python的整数是如何实现的，我们知道Python的整数是不会溢出的，换句话说，它可以计算无穷大的数。只要你的内存足够，它就能计算，但是对于C来说显然是不行的，C能保存的整数范围是有限的。但问题是，Python的底层又是C实现的，那么它是怎么做到整数不溢出的呢？

既然想知道答案，那么看一下整数在底层是怎么定义的就行了。

整数的底层实现

Python的整数在底层对应的结构体是**PyLongObject**，它位于longobject.h中。

```
//longobject.h
typedef struct _longobject PyLongObject;

//longintrepr.h
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};

//合起来可以看成
typedef struct {
    PyObject_VAR_HEAD
    digit ob_digit[1];
} PyLongObject;

//如果把这个PyLongObject更细致的展开一下就是
typedef struct {
    //引用计数
    Py_ssize_t ob_refcnt;
    //类型
    struct _typeobject *ob_type;
    //维护的元素个数
    Py_ssize_t ob_size;
    //digit类型的数组，长度为1
    digit ob_digit[1];
} PyLongObject;
```

别的先不说，就冲里面的**ob_size**我们就可以思考一番。首先Python的整数有大小、但应该没有长度的概念吧，那为什么会有一个**ob_size**呢？

从结构体成员来看，这个**ob_size**指的应该就是**ob_digit数组**的长度，而这个ob_digit数组显然只能是用来维护具体的值了。而数组的长度不同，那么对应的整数占用的内存也不同。

所以答案出来了，整数虽然没有我们生活中的那种**长度**的概念，但它是个**变长对象**，因为不同的整数占用的内存可能是不一样的。因此这个**ob_size**它指的是底层数组的长度，因为Python的整数对应的值在底层是使用数组来存储的。尽管它没有字符串、列表那种长度的概念，或者说无法对整数使用len函数，但它是个变长对象。

那么下面的重点就在这个**ob_digit数组**了，我们要从它的身上挖掘信息，看看**整数(比如123)**，是怎么放在这个数组里面的。不过首先我们要搞清楚这个digit是个什么类型，它同样定义在longintrepr.h中：

```
//PYLONG_BITS_IN_DIGIT是一个宏
//至于这个宏是做什么的我们先不管
//总之，如果你的机器是64位的，那么它会被定义为30
//机器是32位的，则会被定义为15
#if PYLONG_BITS_IN_DIGIT == 30
typedef uint32_t digit;
// ...
#elif PYLONG_BITS_IN_DIGIT == 15
typedef unsigned short digit;
// ...
#endif
```

而我们的机器现在基本上都是64位的，所以**PYLONG_BITS_IN_DIGIT**会等于30。因此**digit**等价于**uint32_t(unsigned int)**，所以它是一个无符号32位整型。

因此**ob_digit**是一个无符号32位整型数组，长度为1。当然这个数组具体多长则取决于你要存储的整数有多大，不同于 Golang，C 的数组的长度不属于类型信息。

虽然定义的时候，声明数组的长度为 **1**，但你可以把它当成长度为 **n** 的数组来用，这是 C 语言中常见的编程技巧。至于这个**n**具体是多少，要取决于你的整数大小。显然整数越大，这个数组就越长，占用的空间也就越大。

搞清楚了PyLongObject里面的所有成员，那么下面我们就来分析ob_digit是怎么存储Python的整数，以及Python的整数为什么不会溢出。

不过说实话，关于Python的整数不会溢出这个问题，相信很多人已经有答案了，因为底层是使用数组存储的嘛，而数组的长度又没有限制，所以当然不会溢出啦。另外，还存在一个问题，那就是digit是一个无符号32位整型，那负数怎么存储？别着急，我们会举栗说明，将上面的疑问一一解答。

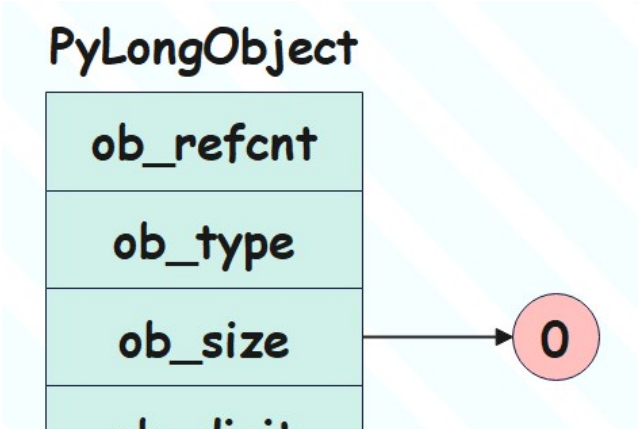
首先我们来抛出一个问题，如果你是Python的设计者，要保证整数不会溢出，你会怎么办？我们把问题简化一下，假设有一个8位的无符号整数类型，我们知道它能表示的最大数字是255，但这时候如果我想表示256，要怎么办？

可能有人会想，那用两个数来存储不就好了。一个存储**255**，一个存储**1**，将这两个数放在数组里面。这个答案的话，虽然有些接近，但其实还有偏差：那就是我们并不能简单地按照大小拆分，比如256拆分成255和1，要是265就拆分成255和10，不能这样拆分，而是要通过二进制的方式，也就是用新的整数来模拟更高的位。

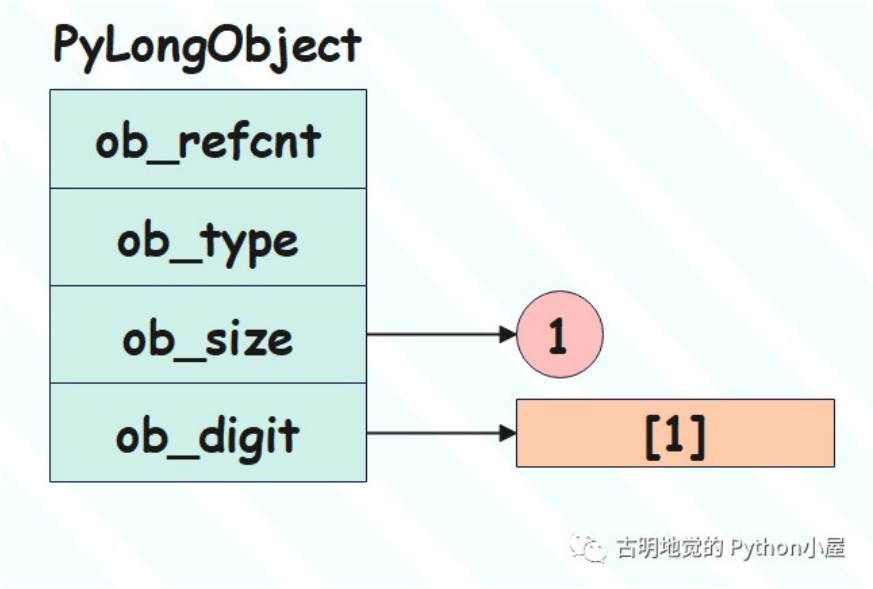
如果感到困惑的话没有关系，我们就以Python整数的底层存储为例，来详细解释一下这个过程。Python底层也是通过我们上面说的这种方式，但是考虑的会更加全面一些。

整数0：

注意：当表示的整数为0时，ob_digit这个数组为空，不存储任何值，ob_size为0，表示这个整数的值为0，这是一种特殊情况。

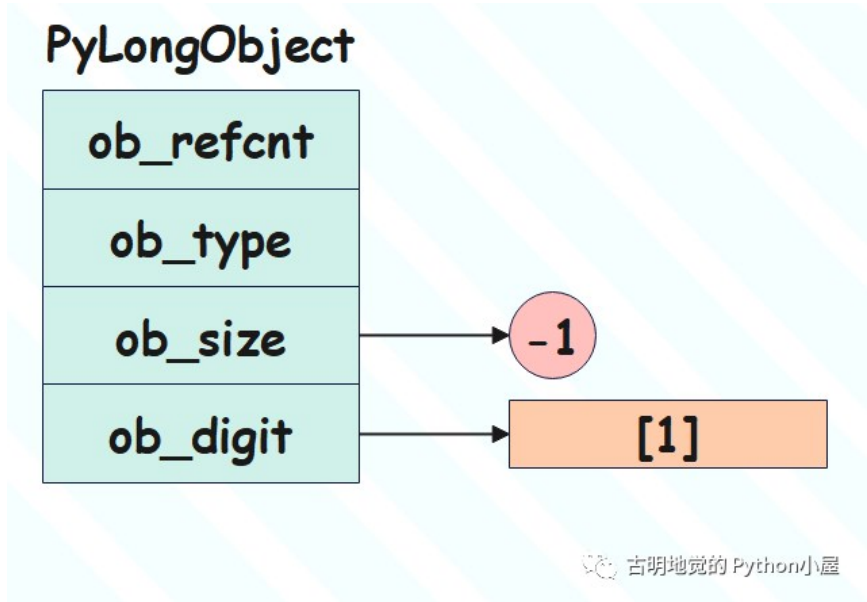


整数1：



当存储的值为1时，此时ob_digit数组就是[1]，显然ob_size的值也是1，因为它维护的就是ob_digit数组的长度。

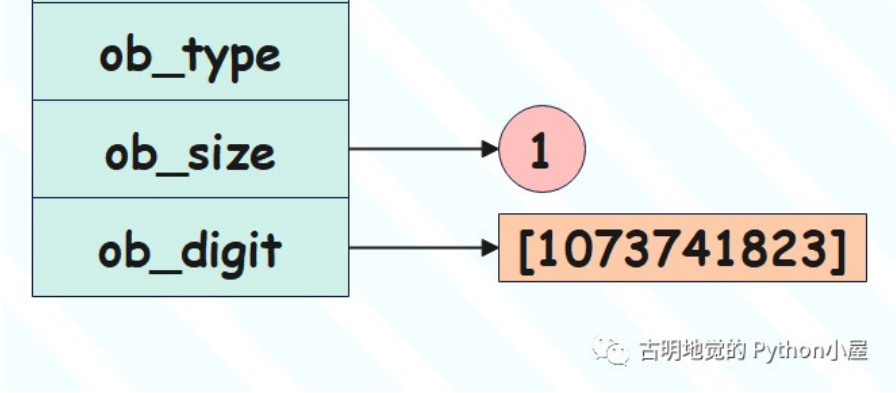
整数-1：



我们看到**ob_digit数组**没有变化，但是**ob_size**变成了**-1**，没错，整数的正负号是通过这里的**ob_size**决定的。ob_digit存储的其实是**绝对值**，无论n取多少，**-n**和**n**对应的ob_digit是完全一致的，但是ob_size则互为**相反数**。所以ob_size除了表示数组的长度之外，还可以表示对应整数的正负。

因此我们之前说整数越大，底层的数组就越长。更准确的说是绝对值越大，底层数组就越长。所以Python在比较两个整数的大小时，会先比较ob_size，如果ob_size不一样则可以直接比较出大小来。显然ob_size越大，对应的整数越大，不管ob_size**是正是负**，都符合这个结论，可以想一下。

整数230 - 1：**



如果想表示**2的30次方减1**，那么也可以使用一个digit表示。话虽如此，但为什么突然说这个数字呢？答案是：虽然digit是4字节、32位，但是解释器只用30个位。

> 之所以这么做是和加法进位有关系，如果32个位全部用来存储其绝对值，那么相加产生进位的时候，可能会溢出。
>
> 比如2 ** 32 - 1，此时 32 个位全部占满了，即便它只加上 1，也会溢出。这个时候为了解决这个问题，就需要先强制转换为64位整数再进行运算，从而会影响效率。但如果只用30个位的话，那么加法是不会溢出的，或者说相加之后依旧可以用32位整数保存。
>
> 因为30个位最大就是2的30次方减1，即便两个这样的值相加，结果也是2的31次方减2。而32个位最大能表示的是2的32次方减1，所以肯定不会溢出的。
>
> 如果一开始30个位就存不下，那么数组中会有两个digit。

所以，虽然将32位全部用完，可以只用一个**digit**表示更大的整数，但是可能面临相加之后一个**digit**存不下的情况，于是只用30个位。如果数值大到30个位存不下的话，那么就会多使用一个digit。

这里可能有人发现了，如果是用31个位的话，那么相加产生的最大值就是**2**2-2**，结果依旧可以使用一个32位整型存储啊，那Python为啥要牺牲两个位呢？答案是为了乘法运算。

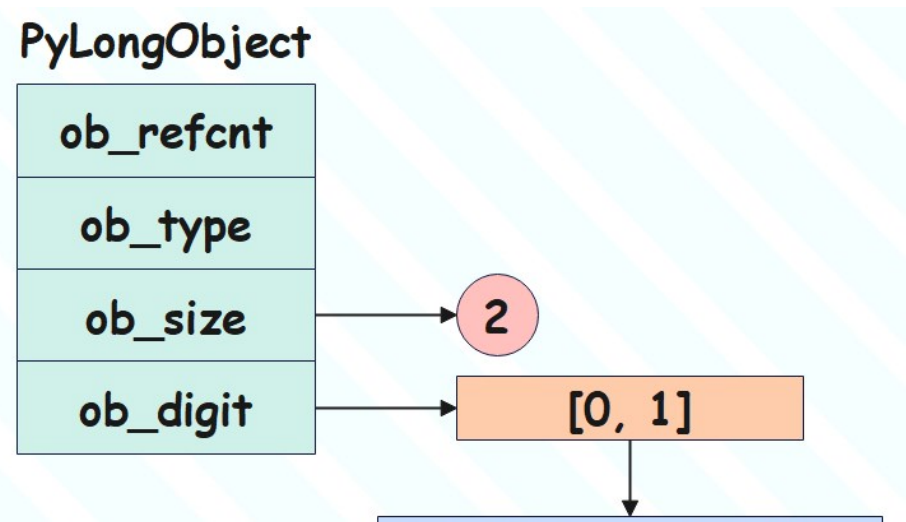
> 为了方便计算乘法，需要多保留 1 位用于计算溢出。这样当两个整数相乘的时候，可以直接按digit计算，并且由于兼顾了"溢出位"，可以把结果直接保存在一个寄存器中，以获得最佳性能。

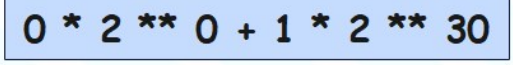
具体细节就不探究了，只需要知道Python的整数在底层是使用unsigned int类型的数组来维护具体的值即可，并且虽然该类型的整数有**32个位**，但解释器只用**30个位**。

然后我们在看digit类型的时候，说过一个宏**PYLONG_BITS_IN_DIGIT**，在64位机器上为30，32位机器上为15。相信这个宏表示的是啥你已经清楚了，它代表的就是使用的digit的位数。

整数2**30：

问题来了，我们说**digit**只用**30个位**，所以**2**30-1**是一个digit能存储的最大值。而现在是**2**30**，所以数组中就要有两个digit了。





我们看到此时就用**两个digit**来存储了，此时的数组里面的元素就是**0和1**，而且充当高位的放在后面，因为是使用**新的digit来模拟更高的位**。

由于一个digit只用30位，那么数组中第一个digit的**最低位**就是**1**，第二个digit的**最低位**就是**31**，第三个digit的**最低位**就是**61**，以此类推。

所以如果**ob_digit**为**[a, b, c]**，那么对应的整数就为：**a * 2 ** 0 + b * 2 ** 30 + c * 2 ** 60**。如果ob_digit不止3个，那么就按照**30个位**往上加，比如ob_digit还有第四个元素d，那么就再加上**d * 2 ** 90**即可。

以上就是Python整数的存储奥秘，说白了就是**串联多个小整数**来表达**大整数**。并且这些小整数之间的串联方式并不是简单的相加，而是将各自的位组合起来，共同形成一个具有高位的大整数，比如将两个8位整数串联起来，表示16位整数。

整数所占的大小是怎么计算的？

下面我们再分析一下，一个整数要占用多大的内存。

相信所有人都知道可以使用 sys.getsizeof 计算大小，但是这大小到底是怎么来的，估计会一头雾水。因为Python中对象的大小，是根据底层的结构体计算出来的。

我们说ob_refcnt、ob_type、ob_size这三个是整数所必备的，它们都是8字节，加起来24字节。所以任何一个整数所占内存都至少24字节，至于具体占多少，则取决于ob_digit里面的元素都多少个。

因此Python中整数所占内存等于 **24 + 4 * ob_size(绝对值)**

```
import sys

# 如果是0的话，ob_digit数组为空
# 所以此时就是24字节
print(sys.getsizeof(0))  # 24

# 如果是1的话，ob_digit数组有一个元素
# 所以此时是24 + 4 = 28字节
print(sys.getsizeof(1))  # 28
# 同理
print(sys.getsizeof(2 ** 30 - 1))  # 28

# 一个digit只用30位，所以最大能表示2 ** 30 - 1
# 如果是2 ** 30，那么就需要两个元素
# 所以是24 + 4 * 2 = 32字节
print(sys.getsizeof(2 ** 30))  # 32
print(sys.getsizeof(2 ** 60 - 1))  # 32


# 如果是两个digit，那么能表示的最大整数就是2 ** 60 - 1
# 因此 2**60次方需要三个digit，相信下面的不需要解释了
print(sys.getsizeof(1 << 60))  # 36
print(sys.getsizeof((1 << 90) - 1))  # 36

print(sys.getsizeof(1 << 90))  # 40
```

所以整数的大小是这么计算的，当然不光整数，其它的对象也是如此。

另外我们也可以反推一下，如果有一个整数**88888888888**，那么它对应的底层数组ob_digit有几个元素呢？每个元素的值又是多少呢？下面来分析一波。

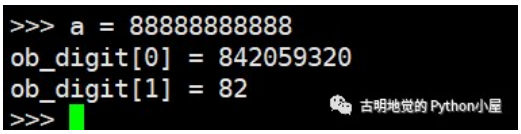
```
import numpy as np

# 假设占了n个位
# 由于n个位能表达的最大整数是 2**n - 1
a = 88888888888
# 所以只需要将a+1、再以2为底求对数，即可算出n的大小
print(np.log2(a + 1))  # 36.371284042320475
```

计算结果表明，如果想要存下这个整数，那么至少需要37个位。而1个digit用30个位，很明显，我们需要两个digit。

如果ob_digit有两个元素，那么对应的整数就等于**ob_digit[0]**加上**ob_digit[1]*2**30**，于是结果就很好计算了。

```
a = 88888888888
print(a // 2 ** 30)  # 82
print(a - 82 * 2 ** 30)  # 842059320
```

所以**整数88888888888**在底层对应的ob_digit数组为**[842059320, 82]**。我们修改解释器，来验证这一结论。



我们看到结果和我们分析的是一样的，但我们说这种办法有点麻烦。在介绍浮点数的时候，我们说过可以通过 ctypes 来构造底层的结构体，这对于整数也是适用的。

```
from ctypes import *

class PyLongObject(Structure):
    _fields_ = [
        ("ob_refcnt", c_ssize_t),
        ("ob_type", c_void_p),
        ("ob_size", c_ssize_t),
        ("ob_digit", c_uint32 * 2)
    ]

a = 88888888888
long_obj = PyLongObject.from_address(id(a))
print(long_obj.ob_digit[0])  # 842059320
print(long_obj.ob_digit[1])  # 82

# 如果我们将 ob_digit[1] 改成 28
# 那么 a 会变成多少呢？
# 很简单，算一下就知道了
long_obj.ob_digit[1] = 28
print(842059320 + 28 * 2 ** 30)  # 30906830392
# 那么a会不会也打印这个结果呢？
# 毫无疑问，肯定会的
print(a)  # 30906830392
# 并且前后a的地址没有发生改变
# 因为我们修改的底层数组
```

通过打印ob_digit存储的值，我们验证了得出的结论，原来Python是通过数组的方式来存储的整数，并且数组的类型虽然是无符号32位整数，但是只用30个位。

当然了，我们还通过修改**ob_digit**，然后再打印**a**进行了反向验证，而输出内容也符合我们的预期。并且在这个过程中，a指向的对象的**地址**并没有发生改变，也就是说，指向

的始终是**同一个对象**。而内容之所以会变，则因为我们是通过修改ob_digit实现的。

还是那句话，站在解释器的层面上看，没啥可变或不可变，一切都由我们决定。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》12. 小整数对象池

下一篇 ›
《源码探秘 CPython》10. 浮点数的行为