07 int 对象，永不溢出的整数-慕课专栏

imooc.com/read/76/article/1903

整数溢出

开始介绍 *int* 对象前，先考考大家：下面这个 *C* 程序(*test.c*)运行后输出什么？是 *1000000000000* (一万亿)吗？

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int value = 1000000;
   printf("%d\n", value * value);

   return 0;
}
```

可能有不少人觉得这没啥好问的，一百万乘以一百万不就是一万亿吗？但现实却不是如此。

在计算机中，由于变量类型存储空间固定，它能表示的数值范围也是有限的。以 *int* 为例，该类型长度为 *32* 位，能表示的整数范围为 *-2147483648* 至 *2147483647* 。一万亿显然超出该范围，换句话讲程序发生了 **整数溢出** 。因此，运行 *test.c* ，程序这样输出也就不奇怪了：

```
$ gcc -o test test.c
$ ./test
-727379968
```

不仅是 *C* 语言，很多编程语言都存在整数溢出的问题，数据库中的整数类型也是。由于整数溢出现象的存在，程序员需要结合业务场景，谨慎选择数据类型。一旦选择不慎或者代码考虑不周，便会导致严重 *BUG* 。

int 对象的行为

与其他语言相比， *Python* 中的整数永远不会有溢出的现象。一百万乘以一百万， *Python* 可以轻易算出来：

```
>>> 1000000 * 1000000
1000000000000
```

Python 甚至可以计算十的一百次方，这在其他语言是不可想象的：

```
>>> 10 ** 100
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

计算结果如此庞大，就算用 *64* 位整数，也难以表示。但 *Python* 中的整数对象缺可以轻松应付，完全不需要任何特殊处理。为什么 *Python* 整数有这样的魔力呢？让我们深入整数对象源码，拨开心中的迷雾。

在源码中，我们将领略到 *C* 语言 **实现大整数的艺术** 。也许你曾经被面试官要求用 *C/C++* 实现大整数，却因为考虑不周而不幸败北。不要紧，掌握 *Python* 整数的设计秘密后，实现大整数对你来说将是易如反掌。

int 对象的设计

int 对象在 *Include/longobject.h* 头文件中定义：

```
typedef struct _longobject PyLongObject;
```

我们顺着注释找到了 *Include/longintrepr.h* ，实现 *int* 对象的结构体真正藏身之处：

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};
```

这个结构我们并不陌生，说明 *int* 对象是一个变长对象。除了变长对象都具有的公共头部，还有一个 *digit* 数组，整数值应该就存储在这个数组里面。 *digit* 又是什么呢？同样在 *Include/longintrepr.h* 头文件，我们找到它的定义：

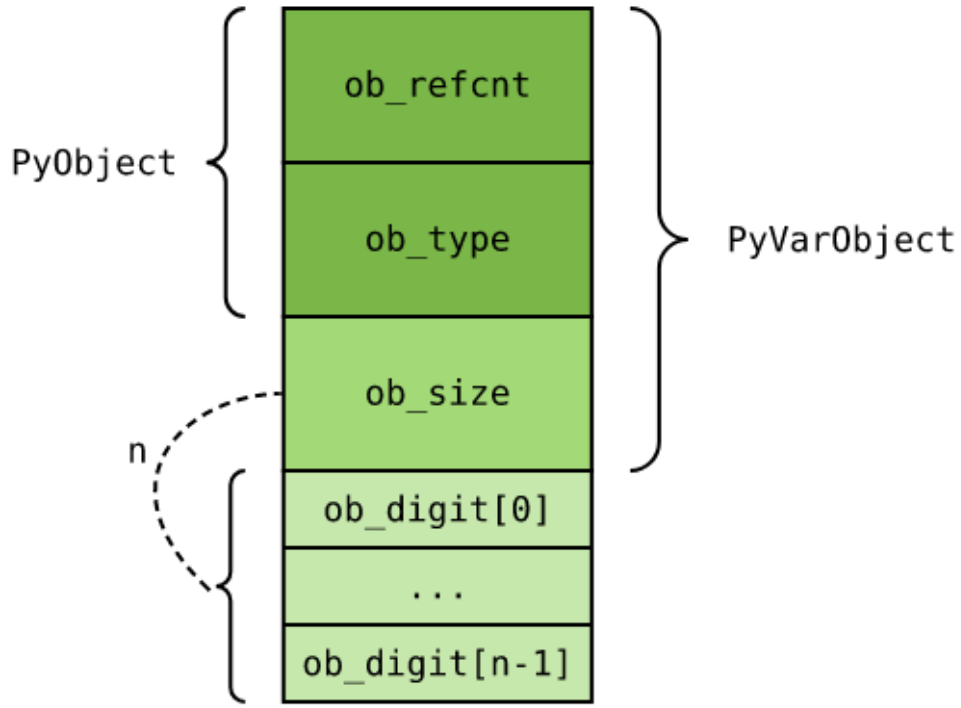
```
#if PYLONG_BITS_IN_DIGIT == 30
typedef uint32_t digit;

#elif PYLONG_BITS_IN_DIGIT == 15
typedef unsigned short digit;

#endif
```

看上去 *digit* 就是一个 *C* 语言整数，至此我们知晓 *int* 对象是通过整数数组来实现大整数的。一个 *C* 整数类型不够就两个嘛，两个不够那就 *n* 个！至于整数数组用什么整数类型来实现， *Python* 提供了两个版本，一个是 *32* 位的 *uint32_t* ，一个是 *16* 位的 *unsigned short* ，编译 *Python* 解析器时可以通过宏定义指定选用的版本。

Python 作者为什么要这样设计呢？这主要是出于内存方面的考量：对于范围不大的整数，用 *16* 位整数表示即可，用 *32* 位就有点浪费。本人却觉得由于整数对象公共头部已经占了 *24* 字节，省这 *2* 个字节其实意义不大。

整数对象	对象大小（16位）	对象大小（32位）
1	24 + 2 * 1 = 26	24 + 4 * 1 = 28
1000000	24 + 2 * 2 = 28	24 + 4 * 1 = 28
10000000000	24 + 2 * 3 = 30	24 + 4 * 2 = 32

由此可见，选用 *16* 位整数数组时， *int* 对象内存增长的粒度更小，有些情况下可以节省 *2* 个字节。但是这 *2* 字节相比 *24* 字节的变长对象公共头部显得微不足道，因此 *Python* 默认选用 *32* 位整数数组也就不奇怪了。



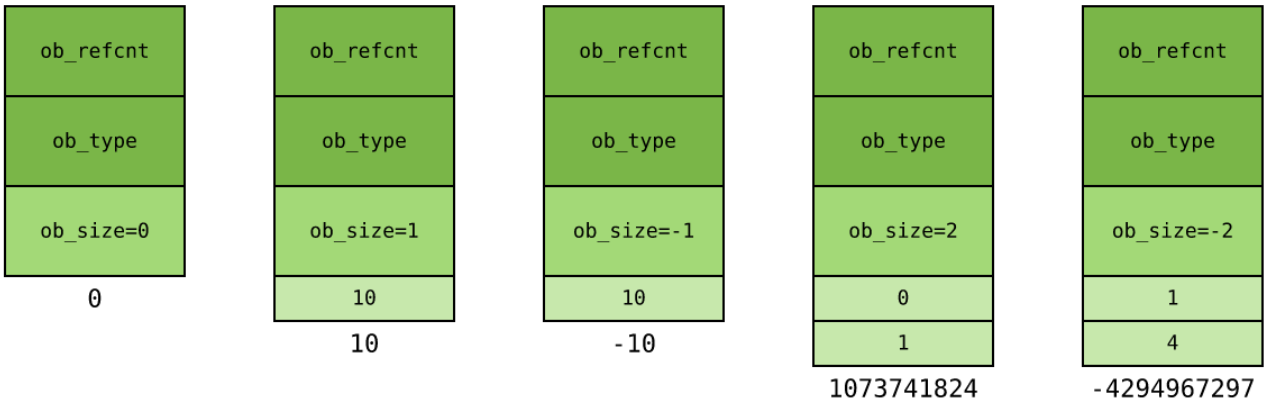
如上图，对于比较大的整数， *Python* 将其拆成若干部分，保存在 *ob_digit* 数组中。然而我们注意到在结构体定义中， *ob_digit* 数组长度却固定为 *1* ，这是为什么呢？由于 *C* 语言中数组长度不是类型信息，我们可以根据实际需要为 *ob_digit* 数组分配足够的内存，并将其当成长度为 *n* 的数组操作。这也是 *C* 语言中一个常用的编程技巧。

大整数布局

整数分为 **正数** 、 **负数** 和 **零** ， *Python* 规定不同整数在 *int* 对象中的存储方式，要点可以总结为 *3* 条：

- 整数 **绝对值** 根据实际情况分为若干部分，保存于 *ob_digit* 数组中；
- *ob_digit* **数组长度** 保存于 *ob_size* 字段，对于 **负整数** 的情况，*ob_size* 为负；
- 整数 **零** 以 *ob_size* 等于 *0* 来表示，*ob_digit* 数组为空；

接下来，我们以 *5* 个典型的例子详细介绍这几条规则：



1. 对于整数 *0* ， *ob_size* 字段等于 *0* ， *ob_digit* 数组为空，无需分配。
2. 对于整数 *10* ，其绝对值保存于 *ob_digit* 数组中，数组长度为 *1* ， *ob_size* 字段等于 *1* 。

3. 对于整数 *-10* ，其绝对值同样保存于 *ob_digit* 数组中，但由于 *-10* 为负数， *ob_size* 字段等于 *-1* 。
4. 对于整数 *1073741824* (*2* 的 *30* 次方)，由于 *Python* 只使用 *32* 整数的后 *30* 位，需要另一个整数才能存储，整数数组长度为 *2* 。绝对值这样计算：230∗1+20∗0=10737418242^{30}*1+2^0*0=1073741824230∗1+20∗0=1073741824。
5. 对于整数 *-4294967297* (负的 *2* 的 *32* 次方加 *1*)，同样要长度为 *2* 的 *ob_digit* 数组，但 *ob_size* 字段为负。绝对值这样计算：230∗4+20∗1=42949672972^{30}*4+2^0*1=4294967297230∗4+20∗1=4294967297。

至于为什么 *Python* 只用 *ob_digit* 数组整数的后 *30* 位，其实跟加法进位有关。如果全部 *32* 位都用来保存绝对值，那么为了保证加法不溢出(产生进位)，需要先强制转换成 *64* 位类型后在进行计算。但牺牲最高 *1* 位后，加法运算便不用担心进位溢出了。那么，为什么 *Python* 牺牲最高 *2* 位呢？我猜这是为了和 *16* 位整数方案统一起来：如果选用 *16* 位整数作为数组， *Python* 则只使用其中 *15* 位。

由于篇幅关系，大整数 **数值运算** 留在下节详细介绍。届时，我们将深入源码，体验大整数运算的精妙之处。

小整数静态对象池

通过前面章节的学习，我们知道整数对象是 **不可变对象** ，整数运算结果是以 **新对象** 返回的：

```
>>> a = 1
>>> id(a)
4408209536
>>> a += 1
>>> id(a)
4408209568
```

Python 这样的设计带来一个性能缺陷，程序运行时势必有大量对象创建销毁。创建对象需要分配内存，对象销毁需要将内存回收，严重影响性能。编写一个循环 *100* 次的循环，便需要创建 *100* 个 *int* 对象：

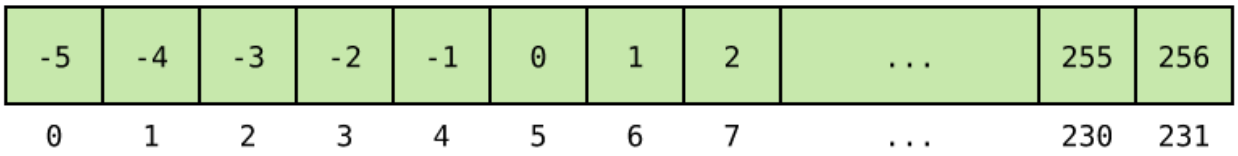
```
for i in range(100):
    pass
```

这显然是难以接受的。 *Python* 的解决方案是：预先将常用的整数对象创建好，以备后用，这就是 **小整数对象池** 。小整数对象池在 *Objects/longobject.c* 中实现，关键代码如下：

```
#ifndef NSMALLPOSINTS
#define NSMALLPOSINTS           257
#endif
#ifndef NSMALLNEGINTS
#define NSMALLNEGINTS           5
#endif

static PyLongObject small_ints[NSMALLNEGINTS + NSMALLPOSINTS];
```

- *NSMALLPOSINTS* 宏规定了对象池 **正数个数** (从 *0* 开始，包括 *0*)，默认 *257* 个；
- *NSMALLNEGINTS* 宏规定了对象池 **负数个数** ，默认 *5* 个；
- *small_ints* 是一个整数对象数组，保存预先创建好的小整数对象；

以默认配置为例， *Python* 启动后静态创建一个包含 *232* 个元素的整数数组并依次初始化为 *-5* 到 *-1* 这 *5* 个负数、零以及 *1* 到 *256* 这 *256* 个正数。 **小整数对象池** 结构如下：



至于为什么选择静态缓存从 *-5* 到 *256* 之间的小整数，主要是出于某种 **权衡** ：这个范围内的整数使用 **频率很高** ，而缓存这些小整数的 **内存开销相对可控** 。很多程序开发场景都没有固定的正确答案，需要根据实际情况平衡利弊。

学习小整数对象池后，如果面试中再被问到 *Python* 整数的这个行为，你也就不会一脸懵逼了：

```
>>> a = 1 + 0
>>> b = 1 * 1
>>> id(a), id(b)
(4408209536, 4408209536)

>>> c = 1000 + 0
>>> d = 1000 * 1
>>> id(c), id(d)
(4410298224, 4410298160)
```

- **场景一** 由于 *1 + 0* 计算结果为 *1* ，在小整数范围内， *Python* 直接从静态对象池中取出整数 *1* ； *1 * 1* 也是同理。名字 *a* 和 *b* 其实都跟同一个对象绑定，即小整数对象池中的整数 *1* ，因而 *id* 相同。
- **场景二** *1000 + 0* 和 *1000 * 1* 计算结果都是 *1000* ，但由于 *1000* 不在小整数范围内， *Python* 分别创建对象并范围，因此 *c* 和 *d* 对象 *id* 不同也就不奇怪了。

总结

与主流编程语言相比， *Python* 中的整数 **永远不会溢出** ，应用起来非常省心。 *Python* 的整数对象是 **变长对象** ，能够按需串联多个 *C* 整数类型，实现大整数表示。整数对象关键字段包括 **底层整数数组** *ob_digit* 以及 **数组长度** *ob_size* 。整数数值按照以下规则保存：

- 整数 **绝对值** 拆分成多个部分，存放于 **底层整数数组** *ob_digit* ；
- 底层数组长度保存在 *ob_size* 字段，如果整数为负， *ob_size* 也为负；
- 对于整数 *0* ，底层数组为空， *ob_size* 字段为 *0* ；

由于整数对象是 **不可变对象** ，任何整数运算结果都以新对象返回，而对象创建销毁开销却不小。为了优化整数对象的性能， *Python* 在启动时将使用 **频率较高** 的小整数预先创建好，这就是 **小整数缓存池** 。默认情况下，小整数缓存池缓存从 *-5* 到 *256* 之间的整数。