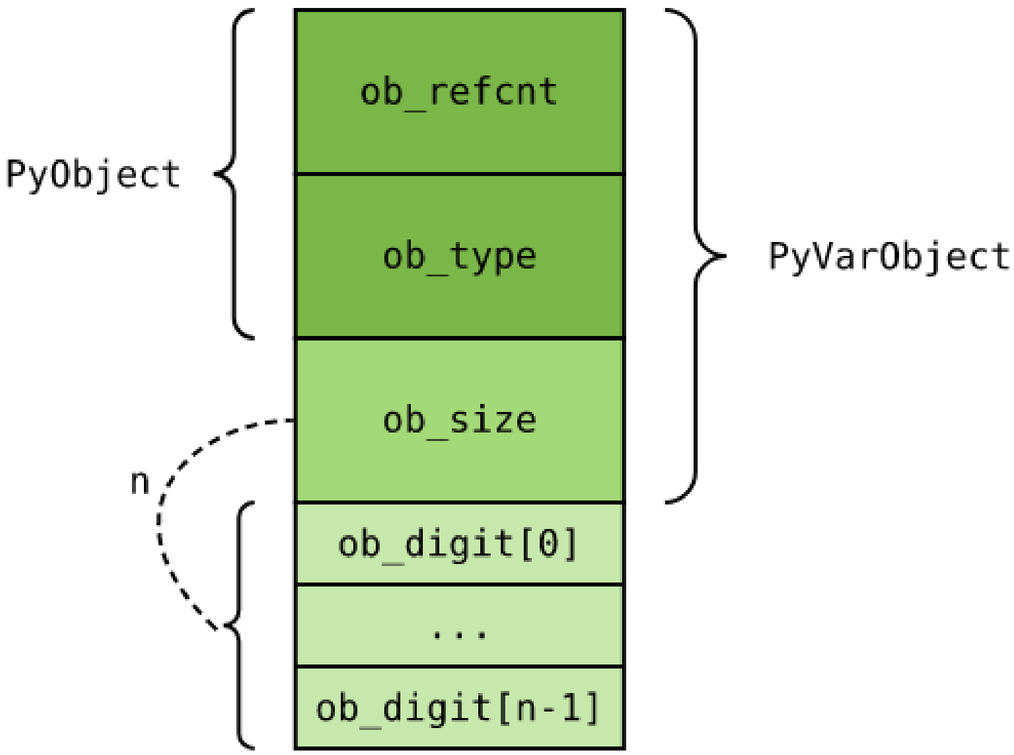
08 int 源码解析：如何实现大整数运算？-慕课专栏

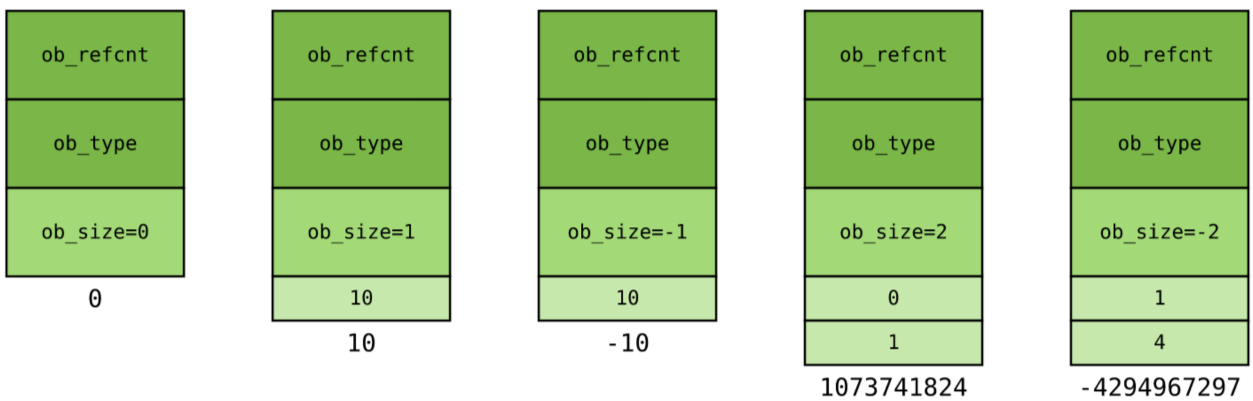
imooc.com/read/76/article/1904

整数溢出是程序开发中一大难题，由此引发的 *BUG* 不计其数，而且相当隐蔽。 *Python* 选择从语言层面彻底解决这个痛点，殚心竭虑设计了整数对象。上一小节，我们探索了整数对象，并初步掌握整数对象的内部结构。

Python 整数对象通过串联多个 *C* 整数类型，实现大整数的表示。整数对象内部包含一个 *C* 整数数组，数组长度与对象表示的数值大小相关，因此整数对象也是 **变长对象** 。深入源码细节前，我们先重温整数对象的内部结构：



- *ob_digit* 为 *C* 整数数组，用于存储被保存整数的 **绝对值** ；
- *ob_size* 为 **变长对象** 关键字段，维护数组长度以及被保存整数的 **符号** ；



用整数数组实现大整数的思路其实平白无奇，难点在于大整数 **数学运算** 的实现，这是也比较考验编程功底的地方。不管是校招还是社招面试，大整数实现都是一个比较常见的考察点，必须掌握。接下来，我们继续深入整数对象源码(*Objects/longobject.c*)，窥探大整数运算的秘密。

数学运算概述

根据我们在 **对象模型** 中学到的知识，对象的行为由对象的 **类型** 决定。因此，整数对象数学运算的秘密藏在整数类型对象中。我们在 *Objects/longobject.c* 中找到整数类型对象(*PyLong_Type*)，其定义如下所示：

```
PyTypeObject PyLong_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "int",
    offsetof(PyLongObject, ob_digit),
    sizeof(digit),
    long_dealloc,



    &long_as_number,




    long_new,
    PyObject_Del,
};
```

类型对象中， *tp_as_number* 是一个关键字段。该字段指向一个 *PyNumberMethods* 结构体，结构体保存了各种数学运算的 **函数指针** 。我们顺藤摸瓜，很快便找到整数对象所有数学运算的处理函数：

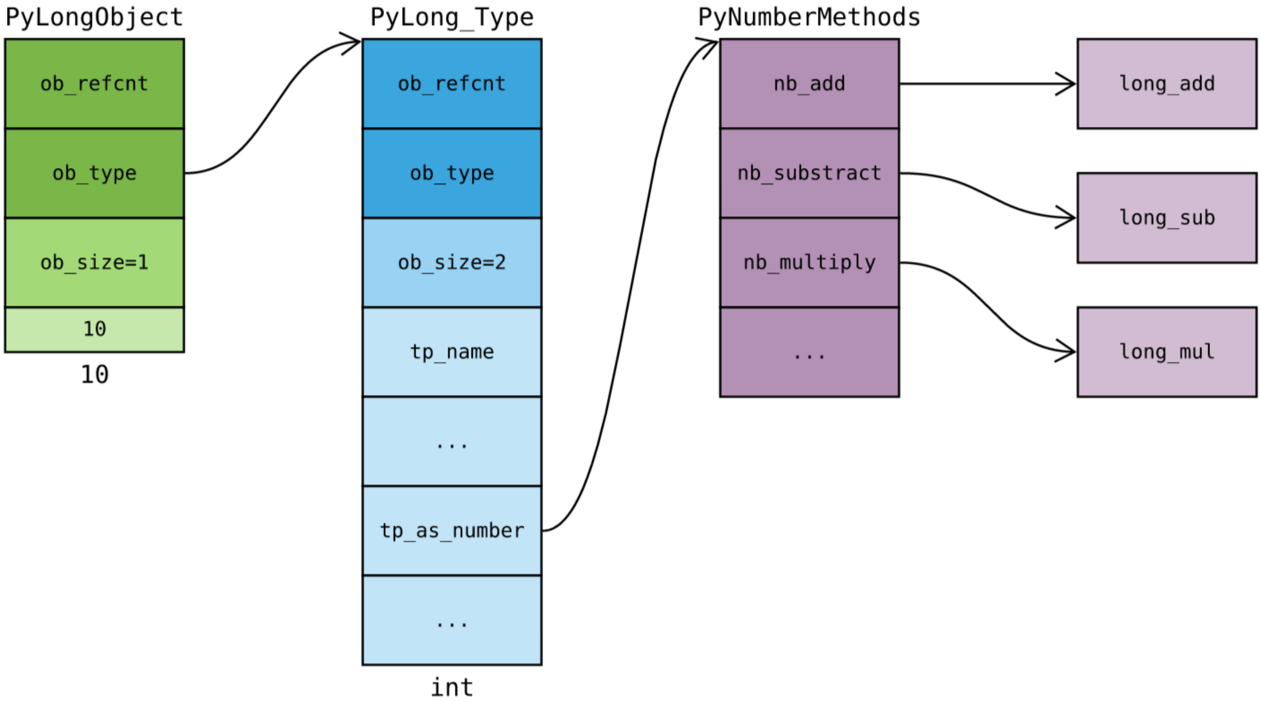
```
static PyNumberMethods long_as_number = {
    (binaryfunc)long_add,
    (binaryfunc)long_sub,
    (binaryfunc)long_mul,
    long_mod,
    long_divmod,
    long_pow,
    (unaryfunc)long_neg,
    (unaryfunc)long_long,
    (unaryfunc)long_abs,
    (inquiry)long_bool,
    (unaryfunc)long_invert,
    long_lshift,
    (binaryfunc)long_rshift,
    long_and,
    long_xor,
    long_or,
    long_long,

};
```

至此，我们明确了整数对象支持的全部 **数学运算** ，以及对应的 **处理函数** (下表仅列举常用部分)：

数学运算	处理函数	示例
加法	long_add	a + b
减法	long_sub	a - b
乘法	long_mul	a * b
取模	long_mod	a % b
除法	long_divmod	a / b
指数	long_pow	a ** b

最后，我们用一张图片来总结 **整数对象** 、 **整数类型对象** 以及 **整数数学运算处理函数** 之间的关系：



加法

如何为一个由数组表示的大整数实现加法？问题答案得在 *long_add* 函数中找，该函数是整数对象 **加法处理函数** 。我们再接再厉，扒开 *long_add* 函数看个究竟(同样位于 *Objects/longobject.c*)：

```
static PyObject *
long_add(PyLongObject *a, PyLongObject *b)
{
    PyLongObject *z;

    CHECK_BINOP(a, b);

    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        return PyLong_FromLong(MEDIUM_VALUE(a) + MEDIUM_VALUE(b));
    }
    if (Py_SIZE(a) < 0) {
        if (Py_SIZE(b) < 0) {
            z = x_add(a, b);
            if (z != NULL) {
                assert(Py_REFCNT(z) == 1);
                Py_SIZE(z) = -(Py_SIZE(z));
            }
        }
        else
            z = x_sub(b, a);
    }
    else {
        if (Py_SIZE(b) < 0)
            z = x_sub(a, b);
        else
            z = x_add(a, b);
    }
    return (PyObject *)z;
}
```

long_add 函数并不长，调用其他辅助函数完成加法运算，主体逻辑如下：

- 第 *4* 行，定义变量 *z* 用于临时保存计算结果；
- 第 *8-10* 行，如果两个对象数组长度均不超过 *1* ， 用 *MEDIUM_VALUE* 宏将其转化成 *C* 整数进行运算即可；
- 第 13-17 行，如果两个整数均为 **负数** ，调用 *x_add* 计算两者绝对值之和，再将结果符号设置为负(*16* 行处)；
- 第 *20* 行，如果 *a* 为负数， *b* 为正数，调用 *x_sub* 计算 *b* 和 *a* 的绝对值之差即为最终结果；
- 第 *24* 行，如果 *a* 为正数， *b* 为负数，调用 *x_sub* 计算 *a* 和 *b* 的绝对值之差即为最终结果；
- 第 26 行，如果两个整数均为正数，调用 *x_add* 计算两个绝对者之和即为最终结果；

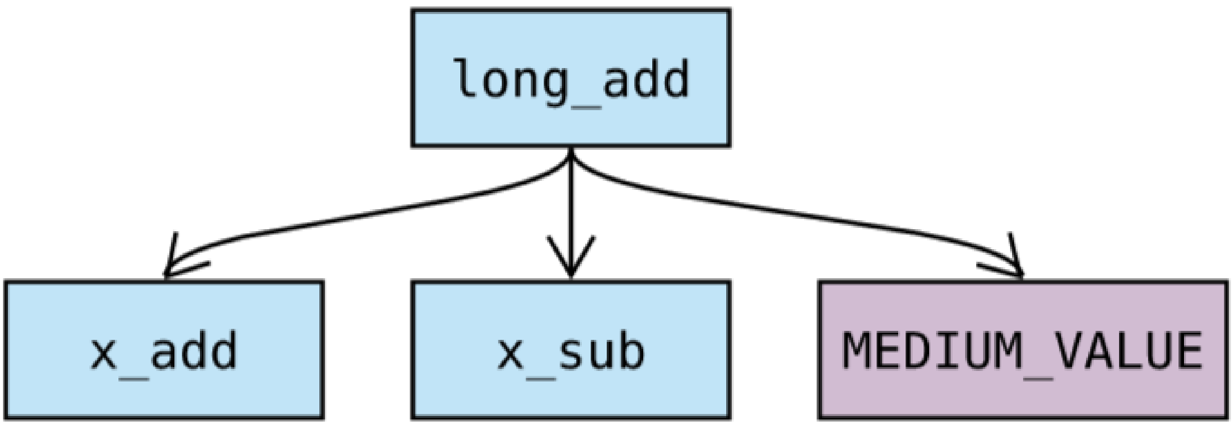
最后 *4* 个步骤看起来很复杂，也有点令人费解。别担心，这只是初中数学的基本知识：

两整数之和 a+b

a \ b	正	负
正	$\lvert a\rvert+\lvert b\rvert$	$\lvert a\rvert-\lvert b\rvert$
负	$\lvert b\rvert-\lvert a\rvert$	$-(\lvert a\rvert+\lvert b\rvert)$

因此， *long_add* 函数将整数加法转换成 **绝对值加法** (*x_add*)以及 **绝对值** 减法(*x_sub*)：

- *x_add(a, b)* ，计算两者绝对值之和，即 $|a|+|b|$ ；
- *x_sub(a, b)* ，计算两者绝对值之差，即 $|a|-|b|$ ；



由于绝对值加、减法不用考虑符号对计算结果的影响，实现更为简单，这是 *Python* 将整数运算转化成绝对值运算的缘由。虽然我们还没弄明白 *x_add* 和 *x_sub* 的细节，但也能从中体会到程序设计中逻辑 **划分** 与 **组合** 的艺术。优秀的代码真的很美！

你可能有些疑问：大整数运算这么复杂，性能一定很差吧？这是必然的，整数数值越大，整数对象底层数组越长，运算开销也就越大。好在运算处理函数均以快速通道对小整数运算进行优化，将额外开销降到最低。

以 *long_add* 为例， *8-10* 行便是一个快速通道：如果参与运算的整数对象底层数组长度均不超过 *1* ，直接将整数对象转化成 *C* 整数类型进行运算，性能损耗极小。满足这个条件的整数范围在 *-1073741824~1073741824* 之间，足以覆盖程序运行时的绝大部分运算场景。

绝对值加法

x_add 用于计算两个整数对象绝对值之和，源码同样位于 *Objects/longobject.c* ：

```
static PyLongObject *
x_add(PyLongObject *a, PyLongObject *b)
{
    Py_ssize_t size_a = Py_ABS(Py_SIZE(a)), size_b = Py_ABS(Py_SIZE(b));
    PyLongObject *z;
    Py_ssize_t i;
    digit carry = 0;


    if (size_a < size_b) {
        { PyLongObject *temp = a; a = b; b = temp; }
        { Py_ssize_t size_temp = size_a;
            size_a = size_b;
            size_b = size_temp; }
    }
    z = _PyLong_New(size_a+1);
    if (z == NULL)
        return NULL;
    for (i = 0; i < size_b; ++i) {
        carry += a->ob_digit[i] + b->ob_digit[i];
        z->ob_digit[i] = carry & PyLong_MASK;
        carry >>= PyLong_SHIFT;
    }
    for (; i < size_a; ++i) {
        carry += a->ob_digit[i];
        z->ob_digit[i] = carry & PyLong_MASK;
        carry >>= PyLong_SHIFT;
    }
    z->ob_digit[i] = carry;
    return long_normalize(z);
}
```

先解释函数中的关键局部变量：

- *size_a* 和 *size_b* ，分别是操作数 *a* 和 *b* 底层数组的长度；
- *z* ，用于保存计算结果的新整数对象；
- *i* ，用于遍历底层整数数组；
- *carry* ，用于保存每个字部分运算的进位；

开始解释函数执行逻辑前，我们先回顾下十进制加法是怎么做的。以 `1024 + 199` 为例：

$$
\begin{array}{ccccc}
 & & 1 & 1 & \\
 & 1 & 0 & 2 & 4 \\
+ & & 1 & 9 & 9 \\
\hline
 & 1 & 2 & 2 & 3
\end{array}
$$

x_add 计算逻辑与此一模一样：

1. 第 *10-15* 行，如果 *a* 数组长度比较小，将 *a* 、 *b* 交换，数组长度较大的那个在前面；
2. 第 *16-18* 行，创建新整数对象，用于保存计算结果（注意到长度必须比 *a* 和 *b* 都大一，因为可能有进位）；
3. 第 *19-23* 行，遍历 *b* 底层数组，与 *a* 对应部分相加并保存到 *z* 中，需要特别注意进位计算；
4. 第 *24-28* 行，遍历 a 底层数组剩余部分，与进位相加后保存到 *z* 中，同样需要特别注意进位计算；
5. 第 *29* 行，将进位写入 *z* 底层数组最高位单元中；
6. 第 *30* 行，去除计算结果 *z* 底层数组中前面多余的零，因为最后的进位（ *29* 行处）可能为零；

最后，我们以一个例子图解两个整数 *a* 和 *b* 绝对值之和的计算过程：

请注意，例子中两个整数真实的值是分别是：

- 1×260+9×230+9×201 \times 2^{60}+9 \times 2^{30}+9 \times 2^{0}1×260+9×230+9×20，而不是十进制的 *199* ；
- 1×290+0×260+2×230+4×201 \times 2^{90}+0 \times 2^{60}+2 \times 2^{30}+4 \times 2^{0}1×290+0×260+2×230+4×20，而不是十进制的 *1024* ；

绝对值减法

有了绝对值加法的基础，绝对值减法才更容易弄懂。 *x_sub* 同样在 *Objects/longobject.c* 中实现：

```
static PyLongObject *
x_sub(PyLongObject *a, PyLongObject *b)
{
    Py_ssize_t size_a = Py_ABS(Py_SIZE(a)), size_b = Py_ABS(Py_SIZE(b));
    PyLongObject *z;
    Py_ssize_t i;
    int sign = 1;
    digit borrow = 0;


    if (size_a < size_b) {
        sign = -1;
        { PyLongObject *temp = a; a = b; b = temp; }
        { Py_ssize_t size_temp = size_a;
            size_a = size_b;
            size_b = size_temp; }
    }
    else if (size_a == size_b) {

        i = size_a;
        while (--i >= 0 && a->ob_digit[i] == b->ob_digit[i])
            ;
        if (i < 0)
            return (PyLongObject *)PyLong_FromLong(0);
        if (a->ob_digit[i] < b->ob_digit[i]) {
            sign = -1;
            { PyLongObject *temp = a; a = b; b = temp; }
        }
        size_a = size_b = i+1;
    }
    z = _PyLong_New(size_a);
    if (z == NULL)
        return NULL;
    for (i = 0; i < size_b; ++i) {

        borrow = a->ob_digit[i] - b->ob_digit[i] - borrow;
        z->ob_digit[i] = borrow & PyLong_MASK;
        borrow >>= PyLong_SHIFT;
        borrow &= 1;
    }
    for (; i < size_a; ++i) {
        borrow = a->ob_digit[i] - borrow;
        z->ob_digit[i] = borrow & PyLong_MASK;
        borrow >>= PyLong_SHIFT;
        borrow &= 1;
    }
    assert(borrow == 0);
    if (sign < 0) {
        Py_SIZE(z) = -Py_SIZE(z);
    }
    return long_normalize(z);
}
```

同样，我们先搞清楚 *x_sub* 函数关键局部变量的作用，现列举如下：

- *size_a* 和 *size_b* ，分别是操作数 *a* 和 *b* 底层数组的长度；
- *z* ，用于保存计算结果的新整数对象；
- *i* ，于遍历底层整数数组；
- *sign* ，绝对值减法结果符号， *1* 表示正； *-1* 表示负；
- *borrow* ，用于维护每部分减法操作的 **借位** ；

x_sub 计算绝对值之差的步骤如下：

1. 第 *11-17* 行，如果 *a* 底层数组比 *b* 小，将计算结果设为 **负** (*sign* 为 *-1*)，并将两者交换以方便计算；

2. 第 *18-30* 行，如果 *a* 、 *b* 底层数组长度一样：

   a. 第 *20-22* 行，跳过高位相同的部分；
   b. 第 *23-24* 行，如果每个部分都一样，则结果为0；
   c. 第 *25-28* 行，如果剩余最高部分， *a* 比较小，将计算结果设为 **负** ，同样将两者交换以方便计算；

3. 第 34*-41* 行，遍历 *b* 底层数组，与 *a* 对应部分相减并保存到 *z* 中，需要特别注意借位计算；

4. 第 42*-47* 行，遍历 a 底层数组剩余部分，与借位相减后保存到 *z* 中，同样需要特别注意借位计算；

5. 第 *49-51* 行，为计算结果 *z* 设置符号；

6. 第 *52* 行，调用 long_normalize 剔除底层数组高位部分多余的零；

减法

现在我们回过头来研究减法处理函数 *long_sub* ，函数同样位于源码文件 *Objects/longobject.c* 中：

```
static PyObject *
long_sub(PyLongObject *a, PyLongObject *b)
{
    PyLongObject *z;

    CHECK_BINOP(a, b);

    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        return PyLong_FromLong(MEDIUM_VALUE(a) - MEDIUM_VALUE(b));
    }
    if (Py_SIZE(a) < 0) {
        if (Py_SIZE(b) < 0)
            z = x_sub(a, b);
        else
            z = x_add(a, b);
        if (z != NULL) {
            assert(Py_SIZE(z) == 0 || Py_REFCNT(z) == 1);
            Py_SIZE(z) = -(Py_SIZE(z));
        }
    }
    else {
        if (Py_SIZE(b) < 0)
            z = x_add(a, b);
        else
            z = x_sub(a, b);
    }
    return (PyObject *)z;
}
```

long_sub 跟 *long_add* 一样直观，基于绝对值加法 *x_add* 和绝对值减法 *x_sub* 这两个底层操作实现：

1. 第 8-10 行，如果 *a* 、 *b* 底层数组长度均不超过 *1* ，直接转换成 *C* 基本整数类型进行计算；
2. 第 *13* 行，如果 *a* 、 *b* 均为负数，结果为 |b|−|a||b|-|a||b|−|a| ，即−(|a|−|b|)-(|a|-|b|)−(|a|−|b|)，特别留意 *18* 行处；
3. 第 *15* 行，如果 *a* 为负数， *b* 为正数，结果为 −(|a|+|b|)-(|a|+|b|)−(|a|+|b|)，特别留意 *18* 行处；
4. 第 *23* 行，如果 *a* 为正数， *b* 为负数，结果为 |a|+|b||a|+|b||a|+|b| ；
5. 第 *25* 行，如果 *a* 、 *b* 均为正数，结果为 |a|−|b||a|-|b||a|−|b| ；

两整数之差 a - b

a \ b	正	负
正	|a| - |b|	|a|+|b|
负	-(|a|+|b|)	-(|a| - |b|)