《源码探秘 CPython》14. 整数在底层是如何运算的?

原创 古明地觉 古明地觉的编程教室 2022-01-19 09:30

楔子

探究完整数的比较之后，再来聊聊整数的加减法运算。如何将数组表示的整数进行相加，也是考验编程功底的地方。

整数的加法

整数在相加的时候会调用**PyNumberMethods**的**nb_add成员**指向的函数，也就是long_add。整数相加的逻辑就在这个函数里面，我们来看一下。

```
static PyObject *
long_add(PyLongObject *a, PyLongObject *b)
{
    //a和b是两个PyLongObject *
    //z显然是指向a和b相加之后的PyLongObject
    PyLongObject *z;

    //CHECK_BINOP是一个宏, 接收两个指针
    //检测它们是不是都指向PyLongObject
    CHECK_BINOP(a, b);

    //判断a和b的ob_size的绝对值是不是都小于等于1
    //如果是的话, 那么说明数组中最多只有一个元素
    //所以这里走的是快分支，我们说快分支的特点是命中率高
    //这里就有所体现，因为绝对值超过 2**30-1 的整数还是比较少的
    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        //MEDIUM_VALUE是一个宏
        //接收一个abs(ob_size) <= 1的PyLongObject *
        //如果ob_size是0, 那么返回0
        //如果ob_size绝对值为1, 那么返回 ob_digit[0]
        //如果ob_size绝对值为-1, 那么返回 -ob_digit[0]
        //所以计算出MEDIUM_VALUE(a) + MEDIUM_VALUE(b)之后
        //将结果转成PyLongObject，然后返回其泛型指针即可
        //因此当数组中元素不超过1个的话，那么显然是可以直接相加的
        return PyLong_FromLong(MEDIUM_VALUE(a) + MEDIUM_VALUE(b));
    }
    //走到这里, 说明至少有一方ob_size的绝对值大于1
    //如果a < 0
    if (Py_SIZE(a) < 0) {
        //如果a < 0并且b < 0
        if (Py_SIZE(b) < 0) {
        //说明两者符号相同，那么调用x_add将两个整数进行相加
        //这个x_add专门用于整数的绝对值相加，并且会返回PyLongObject *
        //至于它的实现我们后面会说
            z = x_add(a, b);
        //但是还没有结束，因为x_add加的是两者的绝对值
        //而z指向的PyLongObject是负数
            if (z != NULL) {
                assert(Py_REFCNT(z) == 1);
              //因为a和b指向的整数都是负数，那么相加之后也是负数
              //所以还要将ob_size乘上-1
                Py_SIZE(z) = -(Py_SIZE(z));
            }
```

```
        }
        else
            //走到这里说明a < 0并且b >= 0, 那么直接让b - a即可
            //此时得到的结果一定是正
            //因此不需要考虑ob_size的符号问题
            z = x_sub(b, a);
    }
    else {
        //走到这里说明a >= 0并且b < 0
        //所以让a - b即可
        if (Py_SIZE(b) < 0)
            z = x_sub(a, b);
        else
            //此时两个整数均>=0, 直接相加
            z = x_add(a, b);
    }
    //将 z 转成泛型指针之后返回
    return (PyObject *)z;
}
```

所以**long_add**这个函数并不长，但是调用了辅助函数**x_add**和**x_sub**，显然核心逻辑是在这两个函数里面。至于long_add函数，它的逻辑如下：

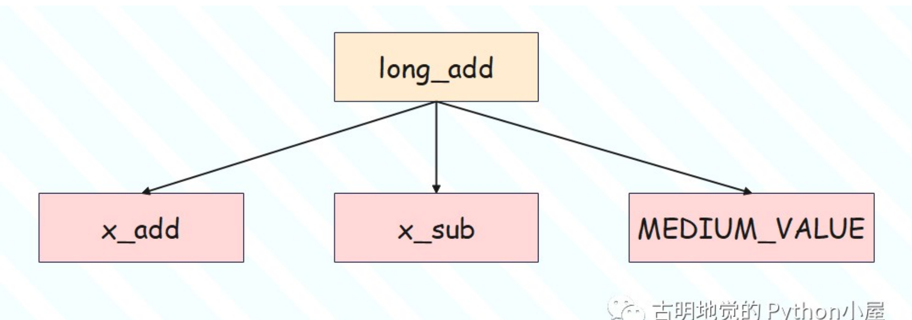
1. 定义一个变量z，用于保存计算结果；
2. 判断两个整数底层对应的数组是不是都不超过1，如果是的话那么通过宏MEDIUM_VALUE直接将其转成C中的一个digit，然后直接相加、返回即可。显然这里走的是快分支，或者快速通道；
3. 但如果有一方ob_size绝对值不小于1，则判断两者的符号。如果都为负，也就是a、b的ob_size均小于0，那么通过x_add计算两者绝对值之和、再将ob_size乘上-1即可；
4. 如果a的ob_size小于0， b的ob_size大于0， 那么通过x_sub计算b和a绝对值之差即可；
5. 如果a的ob_size大于0， b的ob_size小于0， 那么通过x_sub计算a和b绝对值之差即可
6. 如果a的ob_size大于0， b的ob_size大于0， 那么通过x_add计算a和b绝对值之和即可

所以Python的整数设计的非常巧妙，**ob_digit**虽然是用来维护具体数值，但是它并没有考虑正负，整数的正负是通过ob_size来表示的。这样运算的时候，计算的都是整数的绝对值，因此实现起来会方便很多。将绝对值计算出来之后，再通过ob_size来决定正负号。

因此long_add将整数加法转成了**绝对值加法(x_add)**和**绝对值减法(x_sub)**：

- x_add(a, b)，计算两者的绝对值之和，即：|a| + |b|；
- x_sub(a, b)，计算两者的绝对值之差，即：|a| - |b|；

对于long_add而言，如果a、b的符号相同，那么直接将绝对值相加即可。只是当a、b为负数时，加完之后还要将ob_size乘上-1；如果a、b的符号不同，那么让ob_size大于0的整数的绝对值直接减去ob_size小于0的整数的绝对值，相减之后的结果就是最终结果。



由于绝对值的加减法不用考虑符号对计算结果的影响，实现更为简单，所以Python将整数运算转化成整数的绝对值运算。

x_add用于绝对值相加，x_sub用于绝对值相减，虽然我们还没看到这两个函数的具体逻辑，但也能从中体会到程序设计的艺术：划分与组合。

> 那么下面我们的重心就在x_add和x_sub上面了，看看它们是如何对大整数绝对值进行运算的。但是你可能会有疑问，大整数运算肯定很复杂，效率会差吧。显然这是必然的，整数数值越大，整数对象的底层数组就越长，运算开销也就越大。
>
> 但好在运算处理函数均以快速通道的方式对小整数运算进行优化，将额外开销降到了最低。
>
> 比如上面的long_add，如果a和b对应的整数的绝对值都小于等于2**30-1，那么会直接转成C中的整型进行运算，性能损耗极小。并且走快速通道的整数的范围是：-(2**30-1) ~ 2**30-1，即：-1073741823 ~ 1073741823，显然它可以满足我们绝大部分的运算场景。

绝对值加法：x_add

在介绍之前，我们不妨想象一下我们平时算加法的时候是怎么做的：

从最低位开始进行相加，逢十进一，ob_digit 也是同理。我们可以把数组中的每一个元素看成是一个整体，只不过它不再是逢十进一，而是逢**2**30**进一。

```python
#数组的每个元素最大能表示2**30-1
#把元素整体想象成我们平时加法中的个位、十位、百位...
#然后对应的位相加，逢2**30进一
a = [1024, 22]
b = [342, 18]
c = [1024 + 342, 22 + 18]  # [1366, 40]

print(
    a[0] + a[1] * 2 ** 30
    +
    b[0] + b[1] * 2 ** 30
    ==
    c[0] + c[1] * 2 ** 30
)  # True
```

所以仍旧是对应的位进行相加，和我们生活中的加法并无本质上的区别。只不过生活中的加法，每一位能表示**0~9**，逢**十**进一；而Python底层的加法，每一个位能表示**0~2**30-1**，逢**2**30**进一。

把 1024、342 想象成**个位**，把 22、18 想象成**十位**，当然这种说法是不准确的，只是为了方便理解。并且此时不再是逢十进一，而是逢**2****30**进一。

```
a = [2 ** 30 - 1, 16]
b = [2 ** 30 - 1, 21]
# 此时 a[0] + b[0] 一定超过了 2 ** 30
# 所以要进个 1
# 而逢十进一之后，还要再减去十
# 那么逢2**30进一之后，显然要减去2 ** 30
c = [a[0] + b[0] - 2 ** 30,
     a[1] + b[1] + 1]

print(
    a[0] + a[1] * 2 ** 30
    +
    b[0] + b[1] * 2 ** 30
    ==
    c[0] + c[1] * 2 ** 30
)  # True

```

到此，我们就用文字加图片的形式描述了x_add这个函数所做的事情，相信还是很容易理解的，只要类比生活中的加法即可。那么下面我们就来考察一下 x_add 函数，相信理解之后看起来会很简单。

但是介绍之前，需要先了解几个宏，它们在x_add中会有体现：

```
#define PyLong_SHIFT    30
#define PyLong_BASE     ((digit)1 << PyLong_SHIFT)
#define PyLong_MASK     ((digit)(PyLong_BASE - 1))
```

显然PyLong_BASE等于**2****30**，PyLong_MASK等于**2****30-1**，说明32个位，前两个位是0，后三十个位都是1。

然后可以看x_add的具体实现了。

```
static PyLongObject *
x_add(PyLongObject *a, PyLongObject *b)
{
    //显然a和b指向了两个要想加的整数对象
    //这里获取a和b的ob_size的绝对值
    Py_ssize_t size_a = Py_ABS(Py_SIZE(a)), size_b = Py_ABS(Py_SIZE(b));
    //根据a和b的相加结果
    //创建的新的PyLongObject，并返回指针
    PyLongObject *z;
    //循环变量
    Py_ssize_t i;
```

```
//每个部分的运算结果(可不是大神带你carry哦)
digit carry = 0;

//如果size_a小于size_b
if (size_a < size_b) {
    //那么将a和b进行交换, 以及size_a和size_b也进行交换
    //为什么这么做呢？答案是因为方便，可以想象小时候计算加法
    //如果一个位数多，一个位数少，也会习惯将位数多的放在左边
    //最终从右往左，也就是从低位往高位逐个相加，逢十则进一
    { PyLongObject *temp = a; a = b; b = temp; }
    { Py_ssize_t size_temp = size_a;
        size_a = size_b;
        size_b = size_temp; }
    //如果size_a和size_b相等, 或者size_a大于size_b
    //那么该if就无需执行了
}
//_PyLong_New 表示申请一个PyLongObject，并返回指针
//并且其ob_size为size_a + 1
z = _PyLong_New(size_a+1);
//但为什么是size_a + 1呢?
//由于上面的if语句，使得size_a一定不小于size_b
//那么a和b相加之后的z的ob_size一定不小于size_a
//但是也可以也可能比size_a多1，比如: a = 2 ** 60 - 1, b = 1
//所以相加之后结果为2 ** 60次方，于是ob_size就变成了3
//因此在创建z的时候，ob_digit的容量会等于size_a + 1

//正常情况下, z是一个PyLongObject *
//但如果z == NULL, 表示分配失败(程序崩溃)
//但说实话, 除非你内存不够了, 否则这种情况不会发生
if (z == NULL)
    return NULL;

//重点来了，因为size_a > size_b
//所以会以size_b为准，两者从低位向高位依次对应相加
//当b到头了，再单独算a的剩余部分;
//因此以i < size_b作为条件
for (i = 0; i < size_b; ++i) {
//将a->ob_digit[i] + b->ob_digit[i]作为carry
//显然carry如果没有超过2 ** 30 - 1的话
//那么它就是z -> ob_digit[i] 的值
    carry += a->ob_digit[i] + b->ob_digit[i];
//但carry是可能溢出的，当溢出时，应该要减去 2**30
//可以通过 if 进行判断，但是没有使用位运算的效率高
//我们让carry和PyLong_MASK进行"与运算"即可
//PyLong_MASK的前两个位为0, 后面三十个位全为1
//因此当carry不超过2**30-1时，carry & PyLong_MASK就等于carry
//当carry超过2**30-1时，carry & PyLong_MASK就等于carry-2**30
    z->ob_digit[i] = carry & PyLong_MASK;
//然后当carry产生进位时，显然不可以丢
//它们要作用在数组中下一个元素相加的结果上
//所以这里将carry右移30位，也就是产生的进位，为 0 或 1
//然后作用到下一次循环中
    carry >>= PyLong_SHIFT;
}
for (; i < size_a; ++i) {
//如果b到头了, 那么继续从当前的i开始
//直到i == size_a, 逻辑还是和上面一样
//此时只需要加上a->ob_digit[i]，因为b到头了
    carry += a->ob_digit[i];
//这里也要"与上"PyLong_MASK, 因为也可能存在进位的情况
//拿生活中的99999 + 1为例
//此时a = 99999, b = 1, 显然第一次循环b就到头了
//但后面单独循环a的时候, 依旧是要加进位的
//所以这里也是同理
```

```c
        z->ob_digit[i] = carry & PyLong_MASK;
    //carry右移30位
        carry >>= PyLong_SHIFT;
    }
    //两个循环结束之后, 其实还差一步, 还拿99999 + 1举例子
    //按照顺序相加得到的是00000, 因为最后还进了一个1
    //所以这里的carry也是同理
    //因此z的ob_size要比size_a多1, 目的就在于此
    //所以要将z->ob_digit的最后一个元素设置成 carry
    z->ob_digit[i] = carry;
    //如果最后的carry没有进位的话, 显然其结果就是0
    //所以最后没有直接返回z, 而是返回了long_normalize(z)
    //这个long_normalize函数的作用是从后往前依次检查ob_digit的元素
    //如果为0, 那么就将其ob_size减去1, 直到出现一个不为0的元素
    //当然对于我们当前来说, 显然最多只会检查一次
    //因为它的ob_size只比size_a多1, 所以判断数组最后一个元素是否为0即可
    //另外, 其实还可以通过carry进行判断, 显然它要么为 0、要么为 1
    //如果为1, 那么什么也不做, 如果为0, 那么将 ob_size 减 1 即可
    return long_normalize(z);
}
```

Python的整数在底层实现的很巧妙，不理解的话可以多看几遍，然后我们在Python的层面上再反推一下，进一步感受底层运算的过程。

```python
# 假设有a和b两个整数
# 当然这里是使用列表直接模拟的底层数组ob_digit
a = [1073741744, 999, 765, 123341]
b = [841, 1073741633, 2332]
# 然后创建z, 表示a和b的相加结果
z = []

# 为了更直观, 我们一步步手动相加
# 首先是将a[0] + b[0], 得到carry
carry = a[0] + b[0]
# 但carry可能大于2 ** 30 - 1, 如果大于, 那么要减去2**30
# 但我们说这一步可以使用位运算来实现
# 将carry 与上 (2 ** 30 - 1) 即可
print(carry & (2 ** 30 - 1))  # 761
# 结果是761, 说明 carry 比 2**30-1 大
# 然后z的一个元素就是761
z.append(761)

# 然后计算a[1] + b[1]得到新的carry
# 但是之前的carry大于 2 ** 30 - 1
# 所以还要再加上之前的右移30位的carry, 即进位
carry = (carry >> 30) + a[1] + b[1]
# 然后carry & (2 ** 30 - 1)得到809
# 说明carry依旧大于 2 ** 30 - 1
print(carry & (2 ** 30 - 1))  # 809
# 然后z的第二个元素就是809
z.append(809)

# 计算a[2] + b[2]的时候也是同理
carry = (carry >> 30) + a[2] + b[2]
# 但是显然此时的carry已经不大于 2 ** 30 - 1了
print(carry, carry & (2 ** 30 - 1))  # 3098 3098
# 说明z的第三个元素是3098
z.append(3098)

# 此时b到头了, 所以直接将a[3]作为carry
# 当然还要判断上一步的carry是否大于2 ** 30 - 1
# 所以还是右移30位, 当不大于2**30-1时
# carry >> 30 就是0
carry = (carry >> 30) + a[3]
```

```
print(carry)  # 123341
print(carry & (2 ** 30 - 1))  # 123341
z.append(123341)

# 此时a也遍历完毕，但是不要忘记再对carry进行判断
# 如果大于2**30-1，那么会产生进位，所以 z 还要再append一个 1
# 当然这里carry没有超过2 ** 30 - 1

# 此时z为[761, 809, 3098, 123341]
print(z)  # [761, 809, 3098, 123341]

# 因此ob_digit为[1073741744, 999, 765, 123341]
# 和ob_digit为[841, 1073741633, 2332]的两个PyLongObject相加
# 得到的新的PyLongObject的ob_digit为[761, 809, 3098, 123341]
print(
    a[0] + a[1] * 2 ** 30 + a[2] * 2 ** 60 + a[3] * 2 ** 90
    +
    b[0] + b[1] * 2 ** 30 + b[2] * 2 ** 60
    ==
    z[0] + z[1] * 2 ** 30 + z[2] * 2 ** 60 + z[3] * 2 ** 90
)  # True
```

以上就是绝对值加法，我们从源码的角度和Python代码的角度分别解释了一遍。看完了绝对值加法，再来看看绝对值减法。

绝对值减法：x_sub

和绝对值加法一样，绝对值减法也可以类比生活中的减法，从低位到高位分别相减。如果某一位相减的时候发现不够了，那么要向高位借一位。比如 **27 - 9**，7比9 小，因此向**2**借一位变成**17**，减去9，得8。但2被借了一位，所以剩下1，因此结果为**17**。

```
static PyLongObject *
x_sub(PyLongObject *a, PyLongObject *b)
{
    //依旧是获取两者的ob_size的绝对值
    Py_ssize_t size_a = Py_ABS(Py_SIZE(a)), size_b = Py_ABS(Py_SIZE(b));
    //z指向相加之后的PyLongObject
    PyLongObject *z;
    //循环变量
    Py_ssize_t i;
    //如果size_a小于size_b, 那么sign就是-1, 否则就是1
    int sign = 1;
    //之前carry保存相加的结果，这里的borrow保存相减的结果
    //名字很形象, 相加要进位叫carry、相减要借位叫borrow
    digit borrow = 0;

    //如果size_a比size_b小, 说明a的绝对值比b小
    if (size_a < size_b) {
    //那么令sign = -1, 相减之后再乘上sign
    //因为计算的是绝对值之差
    //符号是在绝对值之差计算完毕之后通过sign判断的
        sign = -1;
    //然后依旧交换两者的位置，相减的时候也确保大的一方在左边
    //相加的时候其实大的一方在左边还是在右边没有太大影响
    //但相减的时候大的一方在左边显然会省事很多
    //但交换之后再相减的话，结果还要乘上-1，也就是上面的sign
        { 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) {
```

```c
    //这一个条件语句可能有人会觉得费解，我们分析一下
    //如果两者相等， 那么两个ob_digit里面对应的元素也是有几率都相等的
        i = size_a;
    //所以从ob_digit的尾巴开始遍历
        while (--i >= 0 && a->ob_digit[i] == b->ob_digit[i])
            ;
    //如果都相等， 那么i会等于-1
    //所以这一步也是为了能够快速返回结果，而额外做的一层判断
        if (i < 0)
            //直接返回0即可
            return (PyLongObject *)PyLong_FromLong(0);
    //但如果某个对应的元素不相等
    //假设a的ob_digit是[2, 3, 4, 5]，b的ob_digit是[1, 2, 3, 5]
    //因此上面的while循环结束之后，i会等于2
    //显然只需要计算[2, 3, 4]和[1, 2, 3]之间的差即可
    //因为最高位的5是一样的
    //然后判断索引为i时， 对应的值谁大谁小
        if (a->ob_digit[i] < b->ob_digit[i]) {
        //如果a->ob_digit[i] < b->ob_digit[i]，同样说明a小于b
        //因此将sign设置为-1， 然后交换a和b的位置
            sign = -1;
            { PyLongObject *temp = a; a = b; b = temp; }
        }
        //因为做减法，所以size_a和size_b直接设置成i+1即可
        //因为高位在减法的时候会被抵消掉，所以它们完全可以忽略
        size_a = size_b = i+1;
    }

    //这里依旧是申请空间
    //由于size_a>size_b，相减之后的ob_size一定小于size_a
    z = _PyLong_New(size_a);
    //申请失败返回NULL
    if (z == NULL)
        return NULL;

    //然后下面的逻辑和x_add是类似的
    for (i = 0; i < size_b; ++i) {
    //让a->ob_digit[i] - b->ob_digit[i]等于 borrow
    //但如果存在借位，那么还要减掉上一次的借位
    //不过问题来了，这样相减的话可能得到负数啊
    //不用担心，由于digit是无符号的，所以负数会被转成正数
    //比如:这里相减得到的是-100
    //那么结果就是2 ** 32 - 100，因为digit是无符号32位
    //所以存储的负数会变成 2 ** 32 + 该负数
    //相当于自动往数组的下一个元素借了一位
        borrow = a->ob_digit[i] - b->ob_digit[i] - borrow;
    //但数组的下一个元素比当前元素高了2 ** 30次方
    //所以borrow为负， 那么结果显然加上2 ** 30才对，
    //但是当前borrow加的却是2 ** 32次方
    //所以将borrow还要"与上"PyLong_MASK，或者减去2**30
    //然后其结果才是z->ob_digit[i]的值
        z->ob_digit[i] = borrow & PyLong_MASK;
    //如果真的借了个1， 那么ob_digit中下一个元素肯定是要减去1的
    //但问题是怎么判断到底有没有借位呢？
    //很简单，如果没有借位，borrow一定小于2**30，那么第31个位一定是0
    //如果借位，那么 borrow一定大于2**30，那么第31个位一定是1
    //所以borrow右移30位
        borrow >>= PyLong_SHIFT;
    //然后和1进行与运算
    //如果为0，则没有加上2 ** 32次方，即没有借位
    //那么borrow & 1的结果就是0，下一次循环就不需要减1
    //如果为1，则加上了2 ** 32次方，即发生了借位
    //那么borrow & 1的结果是1，下一次循环需要减去1
        borrow &= 1;
```

```c
    //所以Python底层的整数只用了30个位真的非常巧妙, 尤其是在减法的时候
    //借位一次, 需要借2 ** 30, 因为digit只用30个位
    //但由于C的特性, 借位时会加上2 ** 32次方
    //所以再与上PyLong_MASK, 此时就等价于加上了2 ** 30次方
    //从而得到正确的结果
    //但如果一旦借位, 那么数组下一个元素要减去1
    //所以问题是怎么判断它有没有借位呢？
    //显然要判断两个元素相减之后是否为负
    //如果为负数, 那么C会将这个负数加上2 ** 32次方
    //而两个不超过2**30-1的数相减得到的负数的绝对值显然也不会超过2**30-1
    //换句话说其结果对应的第31位一定是0
    //那么再和2**32次方相加, 得到的结果的第31位一定是1
    //所以再让borrow右移30位、并和1进行与运算
    //如果结果为1, 证明相减为负数, 确实像下一个元素借了1
    //因此下一次循环的会减去1
    //如果borrow为0, 那么就证明不需要借位, 所以下一次循环等于减了一个0
    }

    //如果size_a和size_b一样, 那么这里的for循环是不会满足条件的
    //但不一样的话, 肯定会走这里
    for (; i < size_a; ++i) {
        //我们看到这里的逻辑和之前分析x_add是类似的
        borrow = a->ob_digit[i] - borrow;
        z->ob_digit[i] = borrow & PyLong_MASK;
        borrow >>= PyLong_SHIFT;
        borrow &= 1;
    }
    //只不过由于不会产生进位, 因此不需要对borrow再做额外判断
    //x_add中最后还要判断carry有没有进位
    assert(borrow == 0);
    if (sign < 0) {
        //如果sign < 0, 那么证明是负数
        Py_SIZE(z) = -Py_SIZE(z);
    }
    //最后同样从后往前将z -> ob_digit中为0的元素删掉
    //直到遇见一个不为0的元素
    //比如: 10000 - 9999, 虽然位数多, 但是结果是1
    //所以最后还需要这样的一次判断
    return long_normalize(z);
}
```

所以Python整数在底层的设计确实很精妙，尤其是x_sub，强烈建议多看几遍回味一下。

整数的减法

整数的相减调用的是long_sub函数，显然long_sub和long_add的思路都是一样的，核心还是在x_add和x_sub上面，所以long_sub就没有什么可细说的了。

```c
static PyObject *
long_sub(PyLongObject *a, PyLongObject *b)
{
    //z指向a和b相加之后的PyLongObject
    PyLongObject *z;
    //判断a和b是否均指向PyLongObject
    CHECK_BINOP(a, b);

    //这里依旧是快分支
    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        //直接相减, 然后转成PyLongObject返回其指针
        return PyLong_FromLong(MEDIUM_VALUE(a) - MEDIUM_VALUE(b));
    }
```

```
    //a小于0
    if (Py_SIZE(a) < 0) {
        //a小于0, b小于0
        if (Py_SIZE(b) < 0)
            //调用绝对值减法, 因为两者符号一样
            z = x_sub(a, b);
        else
            //此时两者符号不一样, 那么相减起到的是相加的效果
            z = x_add(a, b);
        if (z != NULL) {
        //然后将z的ob_size变号, 因为x_sub运算的是绝对值
        //所以x_sub中考虑的sign是基于绝对值而言的
        //比如:x_sub接收的a和b的ob_size分别是-5和-3
        //那么得到的结果肯定是正的, 因为会用绝对值大的减去绝对值小的
        //而显然这里的结果应该是负数, 所以还要乘上-1
        //如果x_sub接收的a和b的ob_size分别是-3和-5
        //由于还是用绝对值大的减去绝对值小的
        //所以会交换、从而变号, 得到的结果是负的
        //而显然这里的结果应该是正数, 所以也要乘上-1

        //至于x_add就更不用说了, 当a为负、b为正的时候
        //a - b, 就等于a和b的绝对值相加乘上-1
            assert(Py_SIZE(z) == 0 || Py_REFCNT(z) == 1);
            Py_SIZE(z) = -(Py_SIZE(z));
        }
    }
    else {
        //a大于等于0, b小于0, 所以a - b等于a和b的绝对值相加
        if (Py_SIZE(b) < 0)
            z = x_add(a, b);
        else
            //a、b均大于等于0, 所以直接绝对值相减即可
            //而正数等于其绝对值
            //所以x_sub里面考虑的符号就是真正的结果的符号
            //如果是上面调用的x_sub, 那么还要将结果乘上-1
            z = x_sub(a, b);
    }
    //返回
    return (PyObject *)z;
}

```

所以关于什么时候调用x_add、什么时候调用x_sub，我们总结一下，总之核心就在于它们都是对绝对值进行运算的，掌握好这一点就不难了：

a+b

- 如果a是正、b是正，调用x_add(a, b)，直接对绝对值相加返回结果；
- 如果a是负、b是负，调用x_add(a, b)，但相加的是绝对值，所以long_add中在接收到结果之后还要对ob_size乘上-1；
- 如果a是正、b是负，调用x_sub(a, b)，此时等价于a的绝对值减去b的绝对值。并且x_sub是使用绝对值大的减去绝对值小的，如果a的绝对值大，那么显然正常；如果a的绝对值小，x_sub中会交换，但同时也会自动变号，因此结果也是正常的。举个普通减法的例子：5 + -3, 那么在x_sub中就是5 - 3; 如果是3 + -5, 那么在x_sub中就是-(5 - 3), 因为发生了交换。但不管那种情况，符号都是一样的；
- 如果a是负、b是正，调用x_sub(b, a)，此时等价于b的绝对值减去a的绝对值。所以这个和上面a是正、b是负是等价的；

所以相加时，符号相同会调用x_add、符号不同会调用x_sub。

a-b

- 如果a是正、b是负，调用x_add(a, b)直接对a和b的绝对值相加即可；
- 如果a是正、b是正，调用x_sub(a, b)直接对a和b的绝对值相减即可，会根据绝对

值自动处理符号。而a、b为正，所以针对绝对值处理的符号，也是a - b的符号；

- 如果a是负、b是正，调用x_add(a, b)对绝对值进行相加, 但是结果显然为负，因此在long_sub中还要对结果的ob_size成员乘上-1；
- 如果a是负、b是负，调用x_sub(a, b)对绝对值进行相减, 会根据绝对值自动处理符号, 但是在为负的情况下绝对值越大，其值反而越小, 因此针对绝对值处理的符号，和a - b的符号是相反的。所以最终在long_sub中，也要对结果的ob_size成员乘上-1。举个普通减法的例子：-5 - -3, 那么在x_sub中就类似于5 - 3; 如果是-3 - -5, 那么在x_sub中就类似于-(5 - 3), 因为发生了交换。但不管那种情况得到的值的正负号都是相反的，所以要再乘上-1；

所以相减时，符号相同会调用x_sub、符号不同会调用x_add。

小结

可以看到一个简单的整数相加减，底层居然做了这么多的设计。而且也正如我们之前所说，使用数组实现大整数并不是什么稀奇的事情，但难就难在数学运算，也是非常考验编程功底的地方。

所以，可以仔细地研究一下整数的运算方式，对着源码多阅读几遍。当然啦，我们这里只介绍了加减法，至于乘除法会更加复杂，这里我们就不展开讨论了。并且乘法，Python内部采用的是效率更高的karatsuba算法，比较有意思，有兴趣可以自己查看一下。

以上就是整数的内容，虽然它比浮点数要复杂，但都属于数值，所以特性也比较相似。

回顾一下，我们介绍了整数的底层实现，并分析了Python中的整数为什么不会溢出，以及Python如何计算一个整数所占的字节。当然我们还说了小整数对象池，以及通过分析源码中的long_add和long_sub来了解底层是如何对整数进行运算的。

收录于合集 #CPython 97

上一篇
《源码探秘 CPython》15. bytes 对象是怎么实现的?

下一篇
《源码探秘 CPython》13. 整数在底层是如何进行大小比较的?