数学专栏课外加餐（一）讲我们为什么需要反码和补码

你好，我是黄申。欢迎来到第一次课外加餐时间。

专栏已经更新了几讲，看到这么多人在留言区写下自己的疑惑和观点，我非常开心。很多同学在留言里提出了很多非常好的问题，所以我决定每隔一段时间，对留言里的疑问、有代表性的问题做个集中的解答，也是对我们主线内容做一个补充，希望对你有帮助。

什么是符号位？为什么要有符号位？

在第1讲里，我介绍了十进制数转二进制数。这里面很多人对逻辑右移和算术右移中提到的符号位和补码有疑惑。这里面涉及了几个重要的概念，包括符号位、溢出、原码、反码和补码。我详细讲一下这几个点的来龙去脉。

首先我们来看，**什么是符号位，为什么要有符号位**？用一句话来概括就是，**符号位是有符号二进制数中的最高位，我们需要它来表示负数。**

在实际的硬件系统中，计算机CPU的运算器只实现了加法器，而没有实现减法器。那么计算机如何做减法呢？我们可以通过加上一个负数来达到这个目的。比如，3-2可以看作3+(-2)。因此，负数的表示对于计算机中的二进制减法至关重要。

那么，接下来的问题就是，**如何让计算机理解哪些是正数，哪些是负数呢**？为此，人们把二进制数分为有符号数（signed）和无符号数（unsigned）。

如果是有符号数，那么最高位就是符号位。当符号位为0时，表示该数值为正数；当符号位为1时，表示该数值为负数。例如一个8位的有符号位二进制数10100010，最高位是1，这就表示它是一个负数。

如果是无符号数，那么最高位就不是符号位，而是二进制数字的一部分，例如一个8位的无符号位二进制数10100010，我们可以通过第1讲讲过的内容，换算出它所对应的十进制数是162。由于没有表示负数的符号位，所有无符号位的二进制都代表正数。

有些编程语言，比如Java，它所有和数字相关的数据类型都是有符号位的；而有些编程语言，比如C语言，它有诸如unsigned

int这种无符号位的数据类型。

下面我们来看，**什么是溢出?**

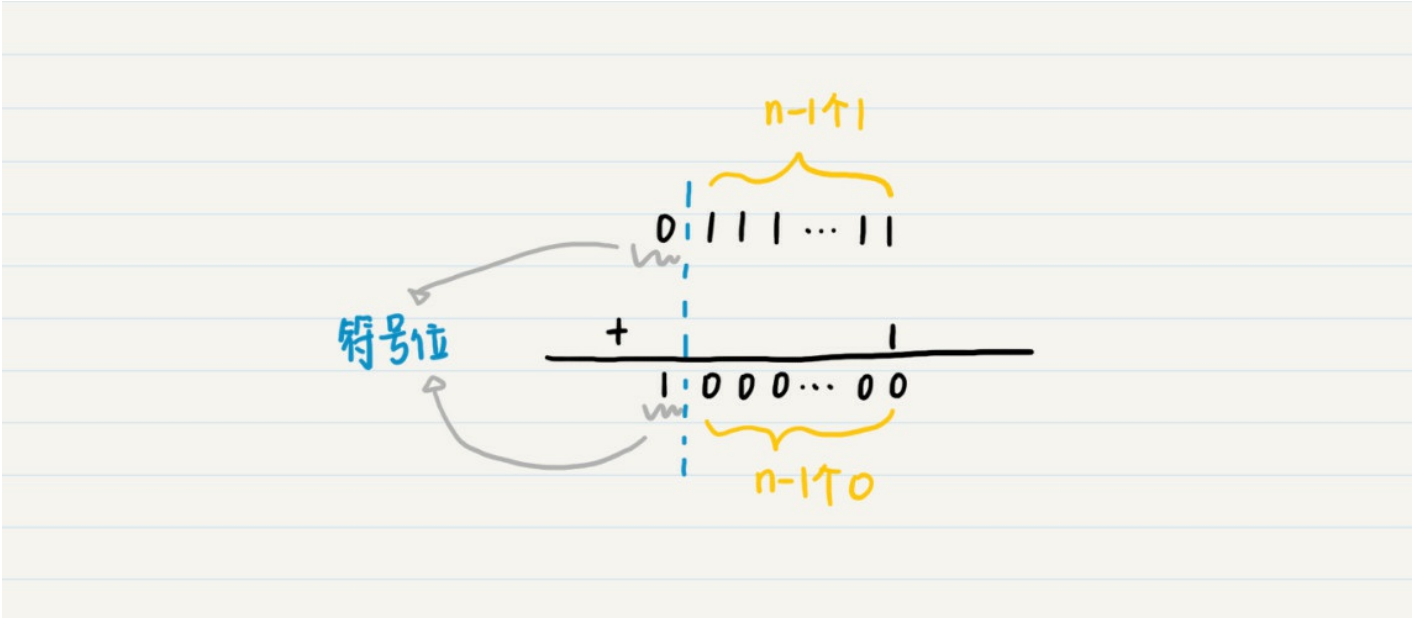
在数学的理论中，数字可以有无穷大，也有无穷小。可是，现实中的计算机系统，总有一个物理上的极限（比如说晶体管的大小和数量），因此不可能表示无穷大或者无穷小的数字。对计算机而言，无论是何种数据类型，都有一个上限和下限。

在Java中，int型是32位，它的最大值也就是上限是2^31-1（最高位是符号位，所以是2的31次方而不是32次方），最小值也就是下限是-2^31。而long型是64位，它的最大值，也就是上限是2^63-1；最小值，也就是下限是-2^63。

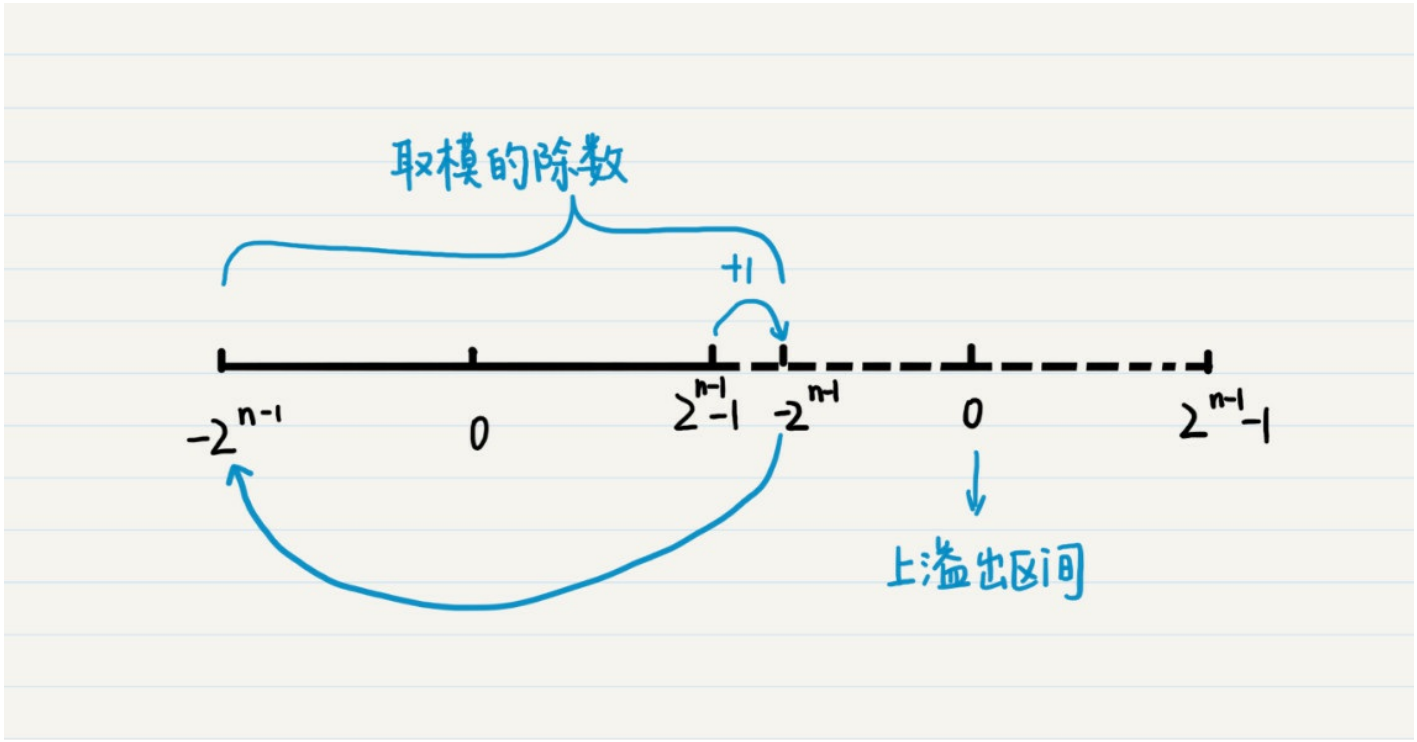
对于n位的数字类型，符号位是1，后面n-1位全是0，我们把这种情形表示为-2^(n-1)，而不是2^(n-1)。一旦某个数字超过了这些限定，就会发生溢出。如果超出上限，就叫**上溢出**（overflow）。如果超出了下限，就叫**下溢出**（underflow）。

那么**溢出之后会发生什么呢?** 我以上溢出为例来给你解释。

n位数字的最大的正值，其符号位为0，剩下的n-1位都为1，再增大一个就变为了符号位为1，剩下的n-1位都为0。而符号位是1，后面n-1位全是0，我们已经说过这表示-2^(n-1)。



那么就是说，上溢出之后，又从下限开始，最大的数值加1，就变成了最小的数值，周而复始，这不就是余数和取模的概念吗？下面这个图可以帮助你的理解。



其中右半部分的虚线表示已经溢出的区间，而为了方便你理解，我将溢出后所对应的数字也标在了虚线的区间里。由此可以看到，所以说，**计算机数据的溢出，就相当于取模。**而用于取模的除数就是数据类型的上限减去下限的值，再加上1，也就是(2^(n-1)-1)-(-2^(n-1))+1=2x2^(n-1)-1+1=2^n-1+1。

你可能会好奇，这个除数为什么不直接写成2^n呢？这是因为2^n已经是n+1位了，已经超出了n位所能表示的范围。

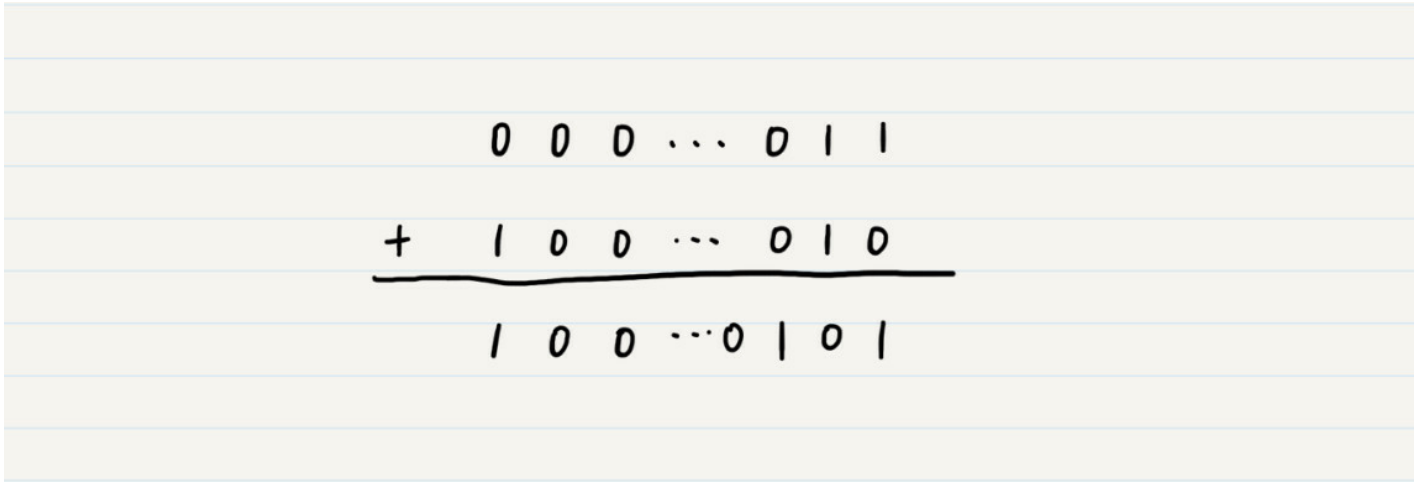
二进制的原码、反码及补码

理解了符号位和溢出，我接下来说说，什么是二进制的原码、反码和补码，以及我们为什么需要它们。

原码就是我们看到的二进制的原始表示。对于有符号的二进制来说，原码的最高位是符号位，而其余的位用来表示该数字绝对值的二进制。所以+2的原码是000…010，-2的的原码是100….010。

那么我们是不是可以直接使用负数的原码来进行减法计算呢？答案是否定的。我还是以3+(-2)为例。

假设我们使用Java中的32位整型来表示2，它的十进制是000…010。最低的两位是10，前面的高位都是0。如果我们使用-2的原码，也就是100…010，然后我们把3的二进制原码000…011和-2的二进制原码100…010相加，会得到100…0101。具体计算你可以看我画的这幅图。

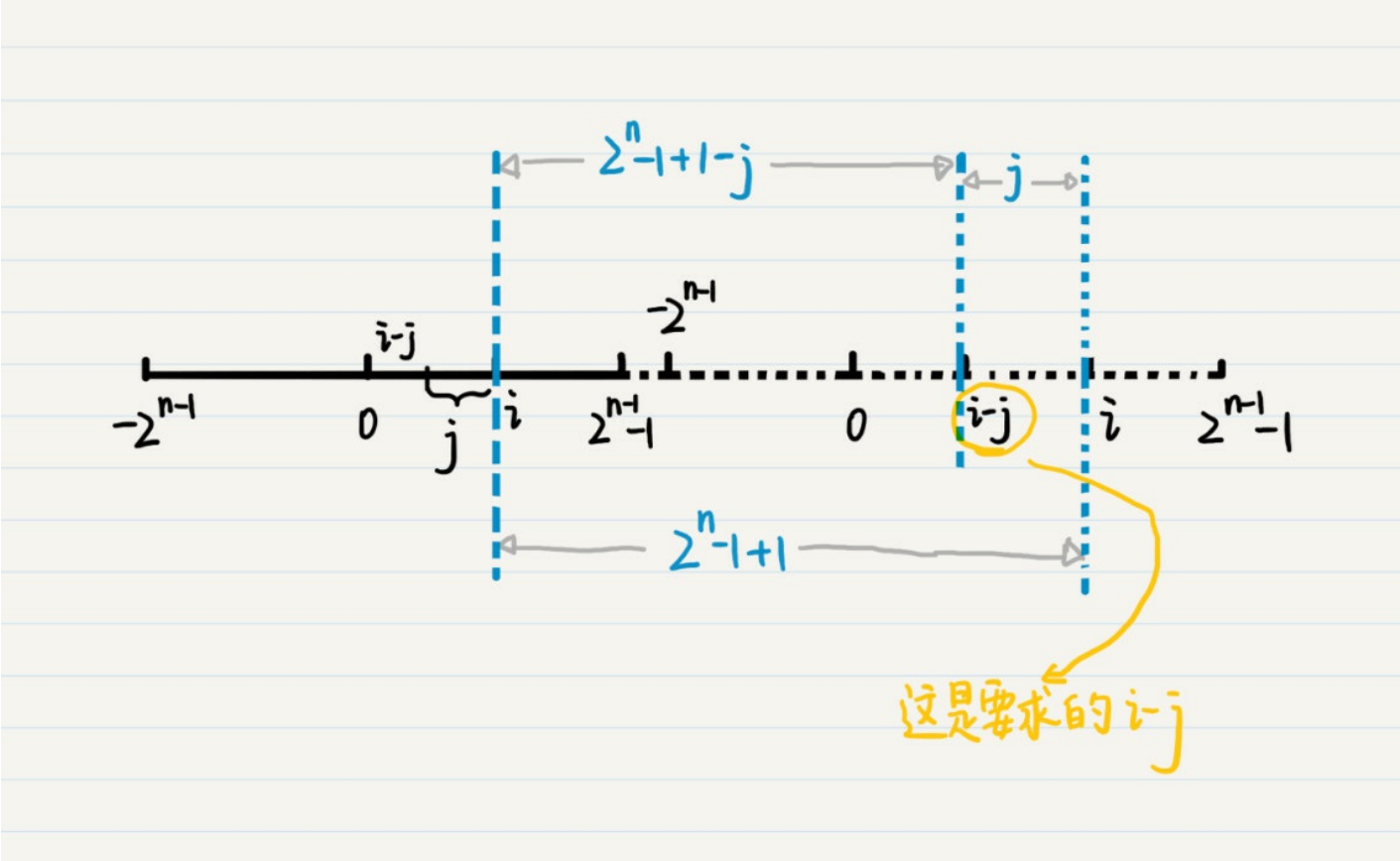


二进制编码上的加减法和十进制类似，只不过，在加法中，十进制是满10才进一位，二进制加法中只要满2就进位；同样，在减法中，二进制借位后相当于2而不是10。

相加后的结果是二进制100…0101，它的最高位是1，表示负数，而最低的3位是101，表示5，所以结果就是-5的原码了，而3+(-2)应该等于1，两者不符。

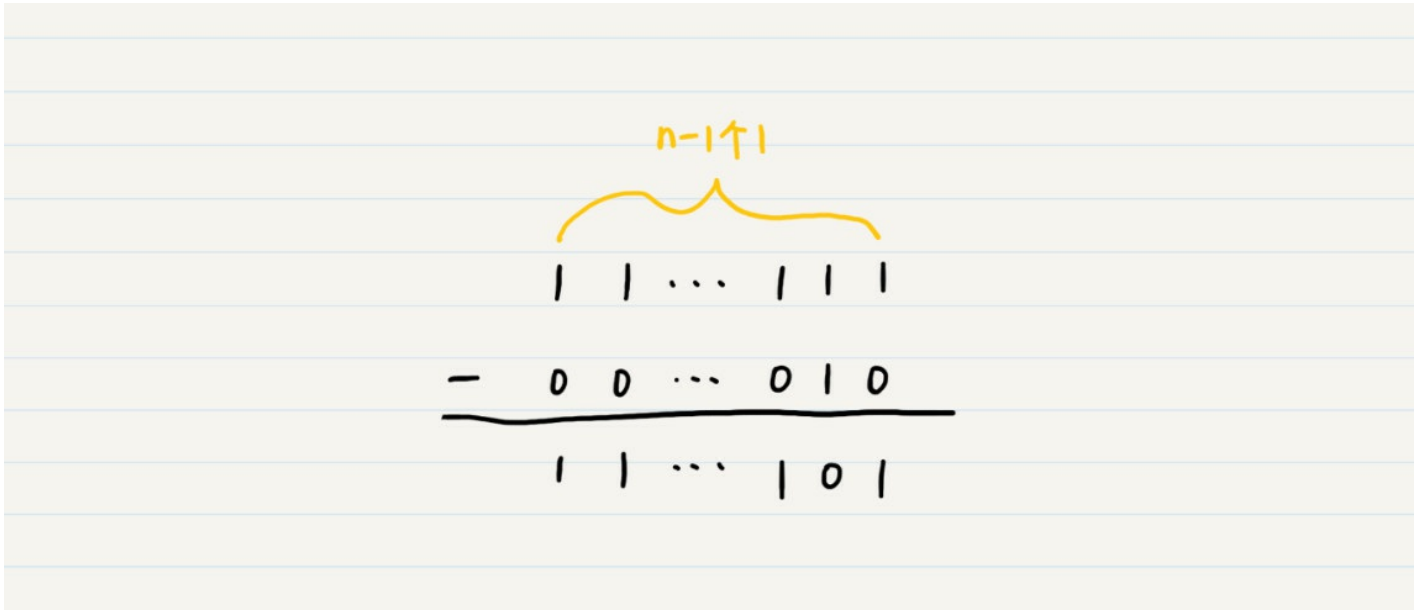
如果负数的原码并不适用于减法操作，那该怎么办呢？这个问题的解答还要依赖计算机的溢出机制。

我刚刚介绍了溢出以及取模的特性，我们可以充分利用这一点，对计算机里的减法进行变换。假设有i-j，其中j为正数。如果i-j加上取模的除数，那么会形成溢出，并正好能够获得我们想要的i-j的运算结果。如果我说的还是不太好理解，你可以参考下面这张图。



我们把这个过程用表达式写出来就是i-j=(i-j)+(2^n-1+1)=i+(2^n-1-j+1)。

其中2^n-1的二进制码在不考虑符号位的情况下是n-1位的1，那么2^n-1-2的结果就是下面这样的：



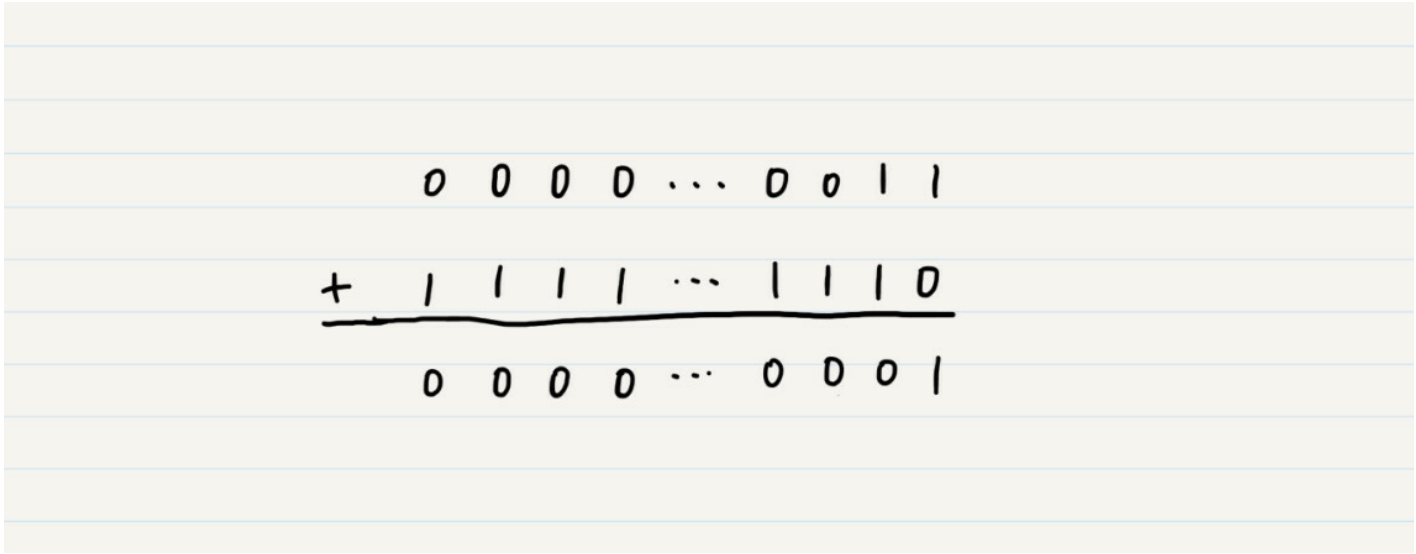
从结果可以观察出来，所谓2^n-1-j相当于对正数j的二进制原码，除了符号位之外按位取反（0变1，1变0）。由于负数-j和正数j的原码，除了符号位之外都是相同的，所以，2^n-1-j也相当于对负数-j的二进制原码，除了符号位之外按位取反。我们把2^n-1-j所对应的编码称为负数-j的反码。所以，-2的反码就是1111…1101。

有了反码的定义，那么就可以得出i-j=i+(2^n-1-j+1)=i的原码+(-j的反码)+1。

如果我们把-j的反码加上1定义为-j的补码，就可以得到i-j=i的原码+(-j的补码)。

由于正数的加法无需负数的加法这样的变换，因此正数的原码、反码和补码三者都是一样的。最终，我们可以得到i-j=i的补码+(-j的补码)。

换句话说，计算机可以通过补码，正确地运算二进制减法。我们再来用3+(-2)来验证一下。正数3的补码仍然是0000…0011，-2的补码是1111…1110，两者相加，最后得到了正确的结果1的二进制。



可见，溢出本来是计算机数据类型的一种局限性，但在负数的加法上，它倒是可以帮我们大忙。

最后，给你留一道思考题吧。理解了负数的原码、反码和补码之后，你能算算看，8位的有符号位二进制数10100010，对应的是哪个十进制数吗？

好了，关于二进制的补充内容就到这里了。欢迎你继续留言给我。你也可以点击“请朋友读”，把今天的内容分享给你的好友，

和他一起精进。

精选留言

老板来几袋面粉

好复杂，开始慌了

2018-12-24 15:36

梓航(﹏)

老师，你讲的那个取模和反码的关系那一段我看不懂，之前看书也没有遇到你说的这种概念，请问还有其他学习资料吗?

2018-12-24 20:26

作者回复

其他的材料一般都没有将“为什么”这么算说清楚，我画了张图，你结合图来理解。简单的说，你可以认为a-b的减法就是给a加上一个特别大的数，导致溢出，然后剩下的反而比a小，这就达到了减法的目的

2018-12-25 05:42

风轨

思考题

0b10100010 = 0b10000000 + 0b00100010

其中

0b10000000 = -128

0b00100010 = 34

所以答案是 -94

2进制取相反数公式

相反数 = 原数减一再取反

- 0b10100010 = !(0b10100010-1) = 0b01011110 = 94

2018-12-24 10:27

作者回复

是的

2018-12-24 13:01

石佳佳_Gemtra

思考题：

原码：10100010

对补码除符号位取反得

反码：11011101

+1操作得

补码：11011110

对应十进制数：-94

还有一种方法，把负数原码除符号位外求和，减去 （2^n-1+1），即 2+32-(2^7-1+1)=-94

2018-12-24 10:01

作者回复

是的

2018-12-24 13:01

Sunsun

为什么负数的补码等于反码加一

1，基本概念（看看了解一下就好）

计算机通过将最高位设置为符号位来表示正负数：

符号位为1时，代表这个数为负数；符号位为0时，代表这个数为正数。

为了方便理解，本文中的例子全部用4位二进制数举例

原码：除符号位外的其他位，保存该二进制数的绝对值。

例如 1：0001 -1：1001

反码：正数的反码等于原码；

负数的反码就是其原码除符号位外，按位取反。

例如 1：0001 -1: 1110

补码：正数的补码等于其原码

负数的的补码等于反码加一

例如 1：1001 -1：1111

2，原码、补码

原码、反码、补码都可以通过符号位非常方便的表示正负数，但是在进行加法计算时，原码和反码都存在这样或那样的问题：

注: 计算机cpu的运算器只实现了加法器，而没有实现减法器，计算机是通过加上一个负数来做减法的

原码：

1 + 1 = 0001 + 0001 = 0010 = 2

1 + -1 = 0001 + 1001 = 1010 = -2

从上面的计算可以看出，原码无法通过加上一个负数来实现减法

反码

1 + -1 = 0001 + 1110 = 1111 = -0

1 + -2 = 0001 + 1010 = 1011 = -4

从上面的计算可以看出，反码也无法实现加上一个负数来实现减法

原码和反码都不能解决的事情，只有通过寻找一种可以完美支持件“减法“的二进制数的表示方法来解决！

3，补码
一个4位的二进制数能表示的数是有限的，从 0000 ~ 1111 ，0000表示0，1111表示 - 1，最大值7（0111），最小值-8（1000）。

看下面这组计算：

0000 + 0001 = 0 + 1 = 0001 = 1
0001 + 0001 = 1 + 1 = 0010 = 2
0010 + 0001 = 2 + 1 = 0011 = 3
...
0111 + 0001 = 7 + 1 = 1000 = -8
1000 + 0001 = -8 + 1 = 1001 = -7
1001 + 0001 = -7 + 1 = 1010 = -6
...
1111 + 0001 = -1 + 1 = 0000 = 0
0000 + 0001 = 0 + 1 = 0001 = 1

0000 每次加上 0001 ； 当最大值7 + 1时，正溢出，结果为最小值-8；最小值-8加上8后，又变成了0000，就像钟表一样，循环往复。

比如说现在有一个数字2，我们想让它变成0，怎么办?
有两种方法:
1. 减去 2 个 1 即：0010 - 0010 = 0000
2. 加上 14 个 1 即：0010 + 1110 = 0000

我们可以总结出，当一个四位的二进制数abcd 减去 另一个四位的二进制数 efgh： abcd - efgh = abcd + (1111 + 1 - efgh) 。

efgh 和 (1111 + 1 - efgh) 对模 （1111 + 1）同余。

如果不太理解，就可以想象一个钟表的时针停在10点的位置，如果想让时针停在8点的位置，可以逆时针的旋转2个刻度，也可以顺时针的旋转10个刻度。

通过公式abcd - efgh = abcd + (1111 + 1 - efgh) ，我们可以得出，如果计算机使用（1111 + 1 -efgh）来表示 -(efgh) ，就可以解决减法的问题。这就是我们补码的原理。

由于1111 - efgh 等于 efgh 的反码 ，所以 efgh 的补码等于 efgh的反码加上1。

2019-01-02 10:41

彩色的沙漠
老师您好
1、文中提到的"对于 n 位的数字类型，符号位是 1，后面 n-1 位全是 0，我们把这种情形表示为 -2^n ，而不是 2^n。"
第一小问为什么不是-2^(n-1)而是你说的-2^n
第二小问是不是应该这样描述"对于 n 位的数字类型，符号位是 1，后面 n-1 位全是 0，我们把这种情形表示为-2^(n-1)，而不

是2^(n-1)"更合适？这个问题依赖第一小问的答案

2、java中int的最小值是-2^63

二进制源码:1 000 0000 0000 0000 0000 0000 0000 0000

二进制反码:1 111 1111 1111 1111 1111 1111 1111 1111

那他的补码怎么表示 反码+1

感谢老师的解答！

2018-12-26 13:06

作者回复

第一处有个笔误，我的原意是-2^(n-1)，而不是2^n

第二个问题很好 -2^63的补码还是它自己，符号位进位舍弃

2018-12-26 15:09

二十八画生

本文重要的是说清了补码的由来，为啥这样定义补码

2019-01-02 23:45

mickey

1010 0010

除符号位取反 -> 1101 1101

加1 -> 1101 1110

转十进制 -> -94

2018-12-24 16:35

枫暂

不错的文章，读罢对溢出，原码，反码，补码的由来有了点线成面的认识，以前对这些数字在计算机中的二进制表示相关知识的理解更多是零散的，串不起来，需要记忆的成分在里面，过上一段时间后，总得重新梳理好半会才理清楚。阅读了本文后，终于有水到渠成，一以贯之之感　，为作者大大点赞

2019-01-04 11:10

作者回复

感谢支持 我们继续努力

2019-01-05 08:39

令

通篇读了2遍，j、-j反码研究了半个小时才彻底明白，感觉容易被绕晕，学习作者的额外加餐，“味道”很美~

2018-12-28 17:17

OzoraTsubasa

老师，请教您2个问题：

1.您文章说的，Java Int类型是32位，最大值是2^31-1，这个最大值是去除符号位第31位的最大值还是从0位按照2^*n 这个算法从0位相加一直加到31位得到的值那?

2.为什么最大正值01111..11+1之后就变成10000..00啦 ，按照逢二进一也应该是01111..10,不明白老师?

2018-12-28 12:42

作者回复

第一个问题的回答是，除符号位之外，最大值是31个1，也就是2^31-1

第二个问题，缝二进一要一直进位，直到最高位

2018-12-29 01:42

KISS

“其中 2^n-1 的二进制码在不考虑符号位的情况下是 n-1位的1“

这个地方不理解，2^n-1的符号位不是0吗?

而2^n - 1是有位的1啊。

这个地方不理解，希望老师能解答一下~

2018-12-27 20:06

作者回复

对于有符号的数据类型，2^n-1是n位1，但是第一个是符号位，所以符号位是1

2018-12-28 01:06

Oli张帆

说的题外话，我发现每次看文字的时候，都很容易走神，但是听语音就好很多。不知道别的同学是怎样的。

2018-12-27 17:23

作者回复

我会努力将普通话发音说得更标准

2018-12-28 01:02

包美丽

为什么Java中int的下限是-2^31，怎么计算出来的?

2018-12-26 20:53

作者回复

假设无符号，那么32个1表示2^32-1，所以范围是0到2^32-1，共2^32个数。Java的int是有符号的，所以最高位是符号位，那么正负各分一半，正的是0到2^31-1，一共2^31个数，负的是-1到-2^31，也是2^31个数。可以将1后面n-1个0看作负0，但是没必要要两个0，所以将它看作-2^31

2018-12-27 01:06

彩色的沙漠

老师，不好意思

问题有一处错误，我纠正一下，以免误导后来的同学

java中int的最小值是-2^31

二进制源码:1 000 0000 0000 0000 0000 0000 0000 0000

二进制反码:1 111 1111 1111 1111 1111 1111 1111 1111

-2^31的补码还是自己，符号位进位舍弃

2018-12-26 16:56

作者回复

是的

2018-12-27 00:55

学徒

“对于 n 位的数字类型，符号位是 1，后面 n-1 位全是0，我们把这种情形表示为 -2^n ，而不是 2^n”，这里不应该是-2^(n-1)吗?

2018-12-26 13:05

作者回复

对 应该是-2^(n-1)

2018-12-26 15:10

okawari

黄老师还能看到吗，肥宅大哭ing，不知道理解的对不对，还请黄老师指正，谢谢黄老师～

我是这样理解的，-2就是再加2就到0的意思，0表示为全0，所以-2需要表示为11111110，这样加两个单位，最高位溢出后就归零了，即为补码。而原码为补码的一个映射，目的是使得人更加容易理解。

2018-12-26 11:57

作者回复

这样理解也是可以的

2018-12-26 15:29

独孤

“对于 n 位的数字类型，符号位是 1，后面 n-1 位全是 0，我们把这种情形表示为 -2^n ，而不是 2^n”，是不是写错了，应该是-2^n-1,而不是2^n-1

2018-12-26 08:23

作者回复

对 你的理解是对的

2018-12-26 15:16

李雷

32位的int值，最大值不应该是31个1么，那应该是2^32-1吧?

2018-12-26 08:21

作者回复

比如4个1是15，相当于2^4-1，所以31个1相当于2^31-1

2018-12-26 11:53

C·K

老师为什么有符号n位二进制数溢出后是-2^n，范围不应该是-2^(n-1)到2^(n-1)-1吗?

2018-12-26 00:33

作者回复

上溢出后应该是-2^(n-1)

2018-12-26 15:24