char 类型的范围为什么是【-128,127】，而不是【-127,127】。

一、问题来源

整型范围的公式： -2^(n-1) ~ 2^(n-1)-1

n为整型的内存占用位数，所以int类型32位 那么就是 -(2^31)~2^31 -1 即 -2147483648~2147483647,但是为什么最小负数绝对值总比最大正数多1？

二、分析问题参数原因

2.1 对于无符号整数，很简单，全部位都表示数值，比如 char型，8位，用二进制表示为0000 0000 ~ 1111 1111。1111 1111 最大即为十进制255,所以 unsigned char 的范围为0~ 255。

2.2 但是对于有符号整数，二进制的最高位表示正负，不表示数值，最高位为0时表示正数，为1时表示负数，这样一来，能表示数值的就剩下(n-1)位了，比如 char a= -1； 那么二进制表示就为 1000 0001，1 表示为0000 0001 ,所以signed char 型除去符号位剩下的7位最大为111 1111 =127，再把符号加上，0 111 1111=127 ， 1111 1111= -127，范围应该为 -127~127。

2.3 计算机只有加法器而没有设计减法器。大家都知道计算机内部是以二进制来存贮数值的，无符号整数会用全部为来存储，有符号的整数，最高位当做符号位 ，其余为表示数值，这样貌似合理，却带来一个麻烦，当进行加法时，1+1

```
  0000 0001
+ 0000 0001
-------------------
  0000 0010   ..................2
```

当相减时 1-1=? 由于计算机只会加法不会减法，它会转化为1+(-1) ,因此

```
  0000 0001
+ 1000 0001
-------------------
  1000 0010   ..............  -2   ，1-1= -2? 这显然是不对了。
```

2.4 反码的出现，为了避免减法运算错误，计算机大神们发明出了反码，直接用最高位表示符号位的叫做原码，上面提到的二进制都是原码形式，反码是原码除最高位其余位取反，规定：正数的反码和原码相同，负数的反码是原码除了符号位，其余为都取反，因此-1 的源码为 1000 0001 ，反码为 1111 1110, 现在再用反码来计算 1+(-1)。

```
  0000 0001
+ 1111 1110
-------------------
  1111 1111     ..........再转化为原码就是 1000 0000 = -0。
```

2.5 补码的出现，反码解决了相减的问题，却又带来一个问题，-0 ，既然0000 0000 表示 0，那么就没有 -0 的必要， 出现 +0= -0=0 ，一个0 就够了，为了避免两个0的问题，计算机大师们又发明了补码，补码规定： 正数的补码是其本身，负数的补码为其反码加一 ，所以，负数转化为反码需两个步

骤， 第一，先转化为反码，第二： 把反码加一。这样 -1 的补码为 1111 1111 ， 1+(-1) 。

0000 0001

\+ 1111 1111

---

1 0000 0000 ..... 这里变成了9位，由于char 为8位，最高位1 被丢弃 结果为0 ，运算正确。

2.6 再看，-0、+0。

-0 ：原码 1000 0000 的补码为1 0000 0000 ，由于char 是 八位 ，所以取低八位0000 0000。

+0 ：原码为0000 0000 ，补码也为 0000 0000 ，虽然补码0都是相同的，但是有两个0 ，既然有两个0 ，况且0既不是正数，也不是负数， 用原码为0000 0000 表示就行了。

这样一来，有符号的char ,原码都用来表示-127~127 之间的数了，唯独剩下原码1000 0000 没有用，用排列组合也可以算出来，0？？？？？？？，能表示2^7=128个数，刚好是0~127, 1???????,也能表示128个数，总共signed char 有256 个数，这与-127~127 中间是两个0 刚好吻合。

2.7 再看，剩下的1000 0000。

既然-127 ~0~ 127都有相应的原码与其对应，那么1000 0000 表示什么呢，当然是-128了，为什么是-128呢？

2.7.1 网上有人说-0即1000 0000 与128的补码相同，所以用1000 0000表示-128,，这我实在是不敢苟同，或者说-128没有原码，只有补码1000 0000,胡扯，既然没有原码何来补码。

2.7.2 还有说-128的原码与-0(1000 0000)的原码相同，所以可以用1000 0000表示-128，我只能说，回答的不要那么牵强。

2.7.3 原码1000 0000 与-128的原码实际上是不同的， 但为什么能用它表示-128进行运算，如果不要限制为char 型（即不要限定是8位），再来看，-128的原码：1 1000 0000 ，9位，最高位符号位，再算它的反码：1 0111 1111，进而，补码为： 1 1000 0000,这是-128的补码，发现和原码一样, 1 1000 0000和1000 0000 相同?如果说一样的人真是瞎了眼了，所以，-128的原码和-0(1000 000)的原码是不同的。

但是在char 型中，是可以用1000 000 表示-128的，关键在于char 是8位，它把-128的最高位符号位1 丢弃了，截断后-128的原码为1000 000 和-0的原码相同，也就是说1000 0000 和-128丢弃最高位后余下的8位相同，所以才可以用-0 表示-128，这样，当初剩余的-0(1000 0000)，被拿来表示截断后的-128,因为即使截断后的-128和char 型范围的其他数(-127~127)运算也不会影响结果， 所以才敢这么表示-128。

比如 -128+(-1)

1000 0000 ----------丢弃最高位的-128

\+ 1111 1111 ---------- -1

---

1 0111 1111 ----------char 取八位，这样结果不正确，不过没关系 ，结果-129本来就超出char 型了，当然不能表示了。

比如 -128+127

1000 0000

\+ 0111 1111

1111 1111 ---------- -1 结果正确， 所以，这就是为什么能用 1000 0000表示-128的原因。

三、 结论

从而也是为什么char 是-128~127，而不是-127~127 ， short int 同样如此 -32768~32767 因为在16位中，-32768为原码为17位，丢弃最高位剩下的16为- 0 的原码相同。。。。

四、扩展问题

还有一个问题：

既然-128最高位丢弃了。那么

char a=-128; //在内存中以补码1 1000 0000 存储，但由于是char ,所以只存储 1000 0000

printf("%d",a); //既然最高位丢弃了，输出时应该是1000 0000 的原码的十进制数-0 ，但为什么能输出-128呢。

还能打印出-128；

我猜想是计算机内部的一个约定，就像float一样 ，能用23位表示24位的精度 ，因为最高位默认为1，到时候把23位取出再加 1便可。

-128也是同样的原理，当数据总线从内存中取出的是1000 0000 ， CPU会给它再添最高一位，变为1 1000 0000 这样才能转化为

-128输出，不然1000 0000 如何输出？这当然是我的一种推断，具体怎么实现还得问CPU的设计者了。。。。

再看一个例子：

char a=-129;

printf("%d",a) ； 会输入多少？？ 结果为127 ，为什么呢？

-129在补码为10 0111 1111 只取后八位存储，即 0111 111 这个值刚好是127了，同理-130 截断后为126.....

如此按模轮回，关于模就先不探讨了。。

那么

```
unsigned  char a=  -1;
if( 1>a)   printf("大于");
else
    printf("小于");
```

结果是什么呢？ 出人意料的是： 小于，而不是大于，猫腻在你哪呢，还是存储问题：

a为unsigned 无符号， 它的八位都用来存储数值， 没有符号位，编译器把 -1 转换为补码为 1111 1111,但由于是无符号，计算机会把 1111 11111 当做是无符号来对待 ，自然就是 2^8 -1 = 255 了，所以相当于是if(1>255) 肯定是

printf("小于");了。。。