[**IP、ICMP、UDP、TCP 校验和算法**](https://www.cnblogs.com/zhaozhong1989/articles/2956295.html)

以前看计算机网络相关的书，每次看到IP或者UDP报头校验和时，都一瞥而过，以为相当简单，不就是16bit数据的相加吗。最近在研究《TCP/IP详解 卷1：协议》这本书，看到校验和是**16bit字的二进制反码和**（晕，以前都没注意原来是反码和，看来以前看书不仔细啊！罪过，罪过~~），觉得很奇怪，为什么会用反码和，而不是直接求和呢？（因为我认为TCP/IP协议里面的算法和思想一般都是非常经典的，人家这么做一定有原因的）下面就来探索一下这个校验和算法具体怎么实现的。

         首先，IP、ICMP、UDP和TCP报文头部都有校验和字段，大小都是16bit，算法也基本一样：

         在发送数据时，为了计算数据包的校验和。应该按如下步骤：

　　（1）把校验和字段置为0；

　　（2）把需校验的数据看成以16位为单位的数字组成，依次进行二进制反码求和；

　　（3）把得到的结果存入校验和字段中。

　　在接收数据时，计算数据包的校验和相对简单，按如下步骤：

　　（1）把首部看成以16位为单位的数字组成，依次进行二进制反码求和，包括校验和字段；

　　（2）检查计算出的校验和的结果是否为0；

　　（3）如果等于0，说明被整除，校验是和正确。否则，校验和就是错误的，协议栈要抛弃这个数据包。

        虽然上面四种报文的校验和算法一样，但在作用范围存在不同：IP校验和只校验20字节的IP报头；而ICMP校验和覆盖整个报文（ICMP报头+ICMP数据）；UDP和TCP校验和不仅覆盖整个报文，而且还有12字节的IP伪首部，包括源IP地址(4字节)、目的IP地址(4字节)、协议(2字节，第一字节补0)和TCP/UDP包长(2字节)。另外UDP、TCP数据报的长度可以为奇数字节，所以在计算校验和时需要在最后增加填充字节0（注意，填充字节只是为了计算校验和，可以不被传送）。

        这里还要提一点，UDP的校验和是可选的，当校验和字段为0时，表明该UDP报文未使用校验和，接收方就不需要校验和检查了！那如果UDP校验和的计算结果是0时怎么办呢？书上有这么一句话：“如果校验和的计算结果为0，则存入的值为全1（65535），这在二进制反码计算中是等效的。”

讲了这么多，那这个校验和到底是怎么算的呢？

**1. 什么是二进制反码求和**

        对一个无符号的数，先求其反码，然后从低位到高位，按位相加，有溢出则向高位进1（跟一般的二进制加法规则一样），**若最高位有进位，则向最低位进1。**  
  
        首先这里的反码好像跟我们以前学的有符号数的反码不一样（即正数的反码是其本身，负数的反码是在其原码的基础上，符号位不变，其余各位取反），这里不分正负数，直接每个位都取反！  
  
        上面加粗的那句是跟我们一般的加法规则不太一样的地方：最高位有进位，则向最低位进1。确实有些疑惑，为什么要这样做呢？仔细分析一下（为了方便说明，以4bit二进制反码求和举例），上面的这种操作，使得在发生加法进位溢出时，溢出的值并不是10000，而是1111。也即是**当相加结果满1111时溢出**，这样也可以说明为什么0000和1111都表示0了（你同样可以发现，任何数与这两个数做二进制反码求和运算结果都是原数，这恰好符合数0的加法意义）。  
  
        下面再举例两种二进制反码求和的运算：  
        原码加法运算                                              反码加法运算  
        3（0011）+ 5（0101）= 8（1000）      3（1100）+ 5（1010）=  8（0111）  
        8（1000）+ 9（1001）= 1（0001）      8（0111）+ 9（0110）=  2（1101）  
        从上面两个例子可以看出，当加法未发生溢出时，原码与反码加法运算结果一样；当有溢出时，结果就不一样了，原码是满10000溢出，而反码是满1111溢出，所以相差正好是1。举例只是为了形象地观察二进制反码求和的运算规则，至于为什么要定义这样的规则以及该运算规则还存在其它什么特性，可能就需要涉及代数理论的东西的了（呜呜~~数学理论没学好啊，只能从表面上分析分析）。  
  
        另外关于二进制反码求和运算需要说明的一点是，先取反后相加与先相加后取反，得到的结果是一样的！（事实上我们的编程算法里，几乎都是先相加后取反。）

**2. 校验和算法的实现**

讲了什么是二进制反码求和，那么校验和的算法实现就简单多了。废话少说，直接上代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/li_xiang1102/article/details/6901660)

1. //计算校验和
2. USHORT checksum(USHORT \*buffer,int size)
3. {
4. unsigned long cksum=0;
5. **while**(size>1)
6. {
7. cksum+=\*buffer++;
8. size-=**sizeof**(USHORT);
9. }
10. **if**(size)
11. {
12. cksum+=\*(UCHAR \*)buffer;
13. }
14. //将32位数转换成16
15. **while** (cksum>>16)
16. cksum=(cksum>>16)+(cksum & 0xffff);
17. **return** (USHORT) (~cksum);
18. }

buffer是指向需校验数据缓存区的指针，size是需校验数据的总长度（字节为单位）

4~13行代码对数据按16bit累加求和，由于最高位的进位需要加在最低位上，所以cksum必须是32bit的unsigned long型，高16bit用于保存累加过程中的进位；另外代码10~13行是对size为奇数情况的处理！

14~16行代码的作用是将cksum高16bit的值加到低16bit上，即把累加中最高位的进位加到最低位上。这里使用了while循环，判断cksum高16bit是否非零，因为第16行代码执行的时候，仍可能向cksum的高16bit进位。

有些地方是通过下面两条代码实现的：

cksum = (cksum >> 16) + (cksum & 0xffff);   
cksum += (cksum >>16);

这里只进行了两次相加，即可保证相加后cksum的高16位为0，两种方式的效果一样。事实上，上面的循环也最多执行两次！

17行代码即对16bit数据累加的结果取反，得到二进制反码求和的结果，然后函数返回该值。

3. 为什么使用二进制反码求和呢？

        好了，最后一个问题，为什么要使用二进制反码来计算校验和呢，而不是直接使用原码或者补码？

        这个问题我想了很久，由于水平有限实在弄不明白，于是在百度上一阵狂搜，什么都没有（不知道是百度不给力，还是大家都不关注这个问题呢？）。果断换google，敲了3个关键词：why checksum tcp，嘿嘿 结果第二篇就是我想要的文章了！！！

先把链接给大家吧：<http://www.netfor2.com/checksum.html>

这篇文章主要介绍二进制反码求和（the 1's complement sum）与补码求和（the 2's complement sum）的区别，另外还说明了在TCP/IP校验和中使用反码求和的优点。

It may look awkword to use a 1's complement addition on 2's complement machines. This method however has its own benefits.

Probably the most important is that it is endian independent. Little Endian computers store hex numbers with the LSB last (Intel processors for example). Big Endian computers put the LSB first (IBM mainframes for example). When carry is added to the LSB to form the 1's complement sum (see the example) it doesn't matter if we add 03 + 01 or 01 + 03. The result is the same.

Other benefits include the easiness of checking the transmission and the checksum calculation plus a variety of ways to speed up the calculation by updating only IP fields that have changed.

        上面是原文的一部分，说明在TCP/IP校验和中使用反码求和的一些优点：

**a. 不依赖系统是大端还是小端。** 即无论你是发送方计算或者接收方检查校验和时，都不需要调用htons 或者 ntohs，直接通过上面第2节的算法就可以得到正确的结果。这个问题你可以自己举个例子，用反码求和时，交换16位数的字节顺序，得到的结果相同，只是字节顺序相应地也交换了；而如果使用原码或者补码求和，得到的结果可能就不相同！

**b. 计算和验证校验和比较简单，快速。**说实话，这个没怎么看明白，感觉在校验和计算方面，原码或者补码求和反而更简单一些（从C语言角度），在校验和验证上面，通过一样的算法判断结果是否为全0，确实要方便一些，所以可能从综合考虑确实反码求和要简便一些。另外，IP报文在传输过程中，路由器经常只修改TTL字段（减1），此时路由器转发该报文时可以直接增加它的校验和，而不需要对IP整个首部进行重新计算。当然，可能从汇编语言的角度看，反码求和还有很多高效的地方，这里就不在深入追究了~~~