## 差错检验-差错校正

### 概述

二进制数据经过传送、存取等环节，会发生误码（1变成0或0变成1），这就有如何发现及纠正误码的问题。所有解决此类问题的方法就是在原始数据（数码位）基础上增加几位校验（冗余）位。

校验的方法有： 奇偶校验法， 求和校验（checksum), 循环冗余校验。

CRC用得最多了。WINRAR, 解压缩的时候也做循环冗余校验。古老的DOS命令arj就采用CRC了。

CRC 的广泛使用在很大程度上只因为它实现起来比较容易，就那么几条指令，几个寄存器就搞定了。  
CRC 不应用做文件安全校验。

### 奇偶校验

**奇偶校验位**是一个表示给定位数的二进制数中 1 的个数是奇数还是偶数的二进制数。奇偶校验位是最简单的错误检测码。

奇偶校验位有两种类型：**偶校验位**与**奇校验位**。如果一组给定数据位中 1 的个数是奇数，那么偶校验位就置为 1，从而使得总的 1 的个数是偶数。如果给定一组数据位中 1 的个数是偶数，那么奇校验位就置为 1，使得总的 1 的个数是奇数。偶校验实际上是循环冗余校验的一个特例，通过多项式 *x* + 1 得到 1 位 CRC。

|  |  |  |
| --- | --- | --- |
| 7位数据 （1的个数） | 带有校验位的字节 | |
| 偶 | 奇 |
| 0000000（0） | 00000000 | 00000001 |
| 1010001（3） | 10100011 | 10100010 |
| 1101001（4） | 11010010 | 11010011 |
| 1111111（7） | 11111111 | 11111110 |

如果传输过程中包括校验位在内的奇数个数据位发生改变，那么奇偶校验位将出错表示传输过程有错误发生。因此，奇偶校验位是一种错误检测码，但是由于没有办法确定哪一位出错，所以它不能进行错误校正。发生错误时必须扔掉全部的数据，然后从头开始传输数据。在噪声很多的媒介上成功传输数据可能要花费很长的时间，甚至根本无法实现。但是奇偶校验位也有它的优点，它是使用一位数据能够达到的最好的校验码

比如：

发送方发送的二进制数据是:10100100 11110101, 这时候是 9 个“1”，那么，就设置BOOL变量 odd1=true.

接收方接受数据，然后检测，如果数据变成：

10001001 1110101，是8个“1”，那么，odd2=false,

odd1!=odd2 报错！

偶数一样的方法。

但是这种方法有很大的缺陷！如果丢失两个位置上的1呢。。。。

应用：

由于它很简单，所以奇偶校验位用于许多计算机硬件中，遇到麻烦时能够重新操作或者通过简单的错误检测就能起到很大作用的场合。例如 SCSI 总线使用奇偶校验位检测传输错误，许多微处理器的指令高速缓存中也包括奇偶校验位保护。因为指令缓存数据是主内存数据的副本，所以在发现错误的时候能够抛弃错误数据并且重新取回数据。

### RAID

奇偶校验块

诞生于1987年，由伯克利加州大学提出。

一些冗余磁盘阵列（RAID）使用奇偶校验块实现冗余。如果阵列中的一块磁盘出现故障，工作磁盘中的数据块与奇偶校验块一起来重建丢失的数据。

下面的图表每列表示一个磁盘，假设 A1 = 00000111、A2 = 00000101 以及 A3 = 0000000。A1、A2、A3 异或得到的 Ap 等于 00000010。如果第二个磁盘出现故障，A2 将不能被访问，但是可以通过 A1、A3 与 Ap 的异或进行重建：

A1 XOR A3 XOR Ap = 00000101

冗余磁盘阵列

A1 A2 A3

Ap B1 B2

Bp C1 C2

C3 C4 Cp

### 汉明码

由Bell实验室的R.W.Hamming发明，因此定名为汉明码。，它们被广泛应用于内存(RAM)。

1940年，汉明于贝尔实验室(Bell Labs)工作，运用贝尔模型V(Bell Model V)电脑，一个周期时间在几秒钟内的机电继电器机器。输入端是依靠打孔卡(Punched Card)，这不免有些读取错误。在平日，特殊代码将发现错误和闪灯(flash lights)，使得操作者能够纠正这个错误。在周末和下班期间，在没有操作者的情况下，机器简单的转移到下一个工作。

汉明在周末工作，他对于不可靠的读卡机总是必须重新开始项目变得愈来愈沮丧。在未来的几年中，他试图解决调试的问题，开发了功能日益强大的算法。在1950年，他发表了至今所称的**汉明码**，今日还使用在许多应用上。

**检验位：**

数据位的位置序号中所有为二的幂次方的位（编号1，2，4，8，等，即数据位位置序号的二进制表示中只有一个1）是校验位

* 校验位 1 覆盖了所有数据位位置序号的二进制表示倒数第一位是1的数据：1（校验位自身，这里都是二进制，下同），11，101，111，1001，等
* 校验位 2 覆盖了所有数据位位置序号的二进制表示倒数第二位是1的数据：10（校验位自身），11，110，111，1010，1011，等
* 校验位 4 覆盖了所有数据位位置序号的二进制表示倒数第三位是1的数据：100（校验位自身），101，110，111，1100，1101，1110，1111，等
* 校验位 8 覆盖了所有数据位位置序号的二进制表示倒数第四位是1的数据：1000（校验位自身），1001，1010，1011，1100，1101，1110，1111，等
* 简而言之，所有校验位覆盖了数据位置和该校验位位置的二进制与的值不为0的数。

现以数据码1101为例讲讲汉明码的编码原理，此时D8=1、D4=1、D2=0、D1=1，在P1编码时，先将D8、D4、D1的二进制码相加，结果为奇数3，汉明码对奇数结果编码为1，偶数结果为0，因此P1值为1，D8+D2+D1=2，为偶数，那么P2值为0，D4+D2+D1=2，为偶数，P3值为0。这样，参照上文的位置表，汉明码处理的结果就是1010101。在这个4位数据码的例子中，我们可以发现每个汉明码都是以三个数据码为基准进行编码的。

设校验位分别为P1、P2、P3，则汉明码共4+3=7位，即：P1P2D4P3D3D2D1

P1=H1

P2=H2

P3=H4

P4=H8

P5=H16

**计算演示过程：**

正确值：1010101

P1 P2 D1 P3 D2 D3 D4

1 10 11 100 101 110 111

P1 = 1^1^1^1=0 1,3,5,7 最后一位1

P2 = 0^1^0^1=0 2,3,6,7 倒数第二位1

P3 = 0^1^0^1=0 4,5,6,7 倒数第三们1

如果传错为1000101，可以计算：

P1 = 1^0^1^1 = 1

P2 = 0^0^0^1 = 1

P3 = 0^1^0^1 = 0

错误位为011 = 第3位

### 求和校验

checksum:

       假设有三个16BIT的数据：

01100110 01100110

01010101 01010101

00001111 00001111

在发送方先求三个数据的和，存在SUM中。

SUM=11001010 11001010

然后取SUM的补码： 00110101 00110101

把这个SUM’放到要发送的数据包里面，发送到接收方，

接收方先将三个收到的数据和收到的这个SUM'相加，如果结果是

1111111111111111，则没有错误，否则，有误码。

[root@localhost kernel]# cksum abc.log (md5sum sum cksum sha1sum)

|  |
| --- |
|  |
|  |

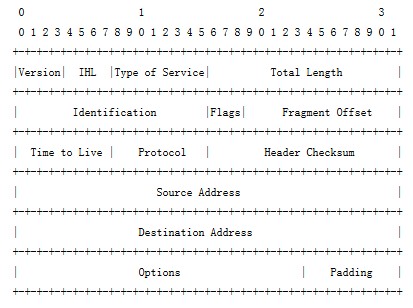
1422024777 11 abc.log

十六进制串: 01 02 03 04 05 06 07 08

的效验和是: 0x24

应用：

IP报头里面有应用，对16位的数据进行checksum校验。



在接收数据时，计算数据报的校验和相对简单，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等于零。  
（3）**如果等于零，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。

Checksum实现：许多实现网络协议栈的代码，这个算法是最常用的了，即使变化，也无非是先取反后取和之类的。考虑其原因，估计还是C语言的移植性更好吧。下面是该函数的实现：

unsigned short checksum\_16bit(unsigned short \*buf, int nword)

{

unsigned int sum;

for (sum = 0; nword > 0; nword--)

        sum += \*buf++; /\*对每一个双字节做处理，逐一求和\*/

sum = (sum>>16) + (sum&0xFFFF); /\*高位双字节　与　低位双字节　求和\*/

sum += (sum>>16); /\*再做一次\*/

return ~sum; /\*将高位双字节抛弃，取反\*/

}

让我们假设一个IP头数据，来解cksum的惑  
  
IP头数据:  
  
01000101                             /\*ver\_hlen\*/  
00000000                             /\*tos\*/  
00000000 00000010                    /\*len\*/  
00000000 00000000                    /\*id\*/  
00000000 00000000                    /\*offset\*/  
00000100                             /\*ttl\*/  
00010001                             /\*type\*/  
00000000 00000000                    /\*cksum(0)\*/  
01111111 00000000 00000000 00000001 -/\*sip\*/  
01111111 00000000 00000000 00000001 -/\*dip\*/

对方机器调用checksum()计算校验和，如果校验和为0表明IP包传输正确  
-----------------------------------------------------------  
01000101                             /\*ver\_hlen\*/  
00000000                             /\*tos\*/  
00000000 00000010                    /\*len\*/  
00000000 00000000                    /\*id\*/  
00000000 00000000                    /\*offset\*/  
00000100                             /\*ttl\*/  
00010001                             /\*type\*/  
10111000 11101001                    /\*cksum(0)\*/  
01111111 00000000 00000000 00000001 /\*sip\*/  
01111111 00000000 00000000 00000001 /\*dip\*/  
把溢出位加回最低位

### CRC

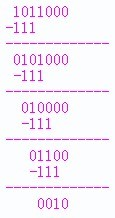
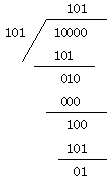
循环冗余码校验：数据通信领域中最常用的一种差错校验码，其特征是信息字段和校验字段的长度可以任意选定。相对于汉明码，检验位占用的位数更小，同时它也可以找出错的具体的位数。

循环冗余码校验英文名称为Cyclical Redundancy Check，简称CRC。它是利用模２除法及余数的原理来作错误侦测（Error Detecting）的。实际应用时，在发送端根据要传送的K位二进制信息码序列，K位信息码对生成多项式做模２除法产生校验用的R位监督码(CRC码)，附在原始信息后边，构成一个新的二进制码序列数共K+R位一起发送给接收装置，接收装置对收到的数据重新计算CRC并与收到的CRC相比较，若两个CRC值不同，则说明数据通讯出现错误。

模2除法与算术除法类似，但每一位除（减）的结果不影响其它位，即不向上一位借位。所以实际上就是异或(1^1=1，0^0=0，0^1=1，1^0=1)。

1011000做模２除111的时候，商＝11110,余＝10

10000做模２除101的时候，商＝101,余＝01



常用生成多项式：

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 生成多项式 | 简记式 | 应用举例 |
| CRC-4 | x4+x+1 | 3 | ITU G.704 |
| CRC-12 | x12+x11+x3+x+1 |  |  |
| CRC-16 | x16+x15+x2+1 | 8005 | IBM SDLC |
| CRC-ITU | x16+x12+x5+1 | 1021 | ISO HDLC, ITU X.25, V.34/V.41/V.42, PPP-FCS |
| CRC-32 | x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x+1 | 04C11DB7 | ZIP, RAR, IEEE 802 LAN/FDDI, IEEE 1394, PPP-FCS |
| CRC-32c | x32+x28+x27+...+x8+x6+1 | 1EDC6F41 | SCTP |

T(x)=xrP(x)+R(x)

编码前的原始信息多项式为P(x)，P(x)的最高幂次加1等于k；生成多项式为G(x)，G(x)的最高幂次等于r；CRC多项式为R(x)；编码后的带CRC的信息多项式为T(x)。

CRC生成步骤：

1、将x的最高幂次为R的生成多项式G(x)转换成对应的R+1位二进制数。

2、将信息码左移R位，相当与对应的信息多项式C(x)\*2R

3、用生成多项式（二进制数）对信息码做模2除，得到R位的余数。

4、将余数拼到信息码左移后空出的位置，得到完整的CRC码。

【例】假设使用的生成多项式是G(x)=x3+x+1。4位的原始报文为1010，求编码后的报文。

解：

1、将生成多项式G(x)=x3+x+1转换成对应的二进制除数1011。

2、此题生成多项式有4位（R+1），要把原始报文C(x)左移3（R）位变成1010000

3、用生成多项式对应的二进制数对左移4位后的原始报文进行模2除：

1001-------商

------------------------

1010000

1011----------除数

------------

1000

1011

------------

011-------余数（校验位）

4　、编码后的报文（CRC码）：

1010000

+       011

------------------

1010011

下面以最常用的CRC-16为例来说明其生成过程(模2除过程)。

CRC-16码由两个字节构成，在开始时CRC寄存器的每一位都预置为1，然后把CRC寄存器与8-bit的数据进行异或，之后对CRC寄存器从高到低进行移位，在最高位（MSB）的位置补零，而最低位（LSB，移位后已经被移出CRC寄存器）如果为1，则把寄存器与预定义的多项式码进行异或，否则如果LSB为零，则无需进行异或。重复上述的由高至低的移位8次，第一个8-bit数据处理完毕，用此时CRC寄存器的值与下一个8-bit数据异或并进行如前一个数据似的8次移位。所有的字符处理完成后CRC寄存器内的值即为最终的CRC值。

　　下面为CRC的计算过程：

　　1．设置CRC寄存器，并给其赋值FFFF(hex)。

　　2．将数据的第一个8-bit字符与16位CRC寄存器的低8位进行异或，并把结果存入CRC寄存器。

　　3．CRC寄存器向右移一位，MSB补零，移出并检查LSB。

　　4．如果LSB为0，重复第三步；若LSB为1，CRC寄存器与多项式码相异或。

　　5．重复第3与第4步直到8次移位全部完成。此时一个8-bit数据处理完毕。

　　6．重复第2至第5步直到所有数据全部处理完成。

　　7．最终CRC寄存器的内容即为CRC值。

**CRC的和纠错**

在接收端收到了CRC码后用生成多项式对G(x)做模2除，若得到余数为0,则码字无误。若如果有一位出错，则余数不为0，而且不同位出错，其余数也不同。可以证明，余数与出错位的对应关系只与码制及生成多项式有关，而与待测碼字（信息位）无关。图10给出了G(x)＝1011，C(x)＝1010的出错模式，改变C(x)（码字），只会改变表中码字内容，不改变余数与出错位的对应关系。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 收到的CRC码字 | 余数 | 出错位 |
| 码位 | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | A7 | A6 | A5 | A4 | A3 | A2 | A1 | |
| 正确 | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 1 | 0 | 0 | 1 | 1 | | 000 | 无 |
| 错  误 | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 1 | 0 | 0 | 1 | 0 | | 1 | 0 | 1 | 0 | 0 | 0 | 1 | | 1 | 0 | 1 | 0 | 1 | 1 | 1 | | 1 | 0 | 1 | 1 | 0 | 1 | 1 | | 1 | 0 | 0 | 0 | 0 | 1 | 1 | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | | 001  010  100  011  110  111  101 | 1  2  3  4  5  6  7 |

按照现在流行的四种已成为国际标准的多项式：V.41的CRC-CCITT、ANSI的CRC-16以及CRC-12和CRC-32，他们的r值分别为16、16、12、32，漏检的概率在0.00000000002~0.000244之间，这样的概率还是令人满意的。

应用：

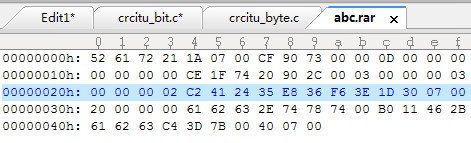
PPP，WINRAR，PNG，

PPP帧。最后两个字节称为FCS(Frame Check Sequence)

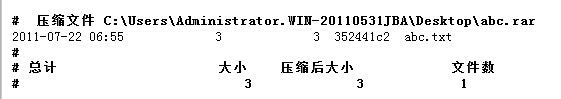
CRC校验域主要是对PPP数据帧传输的正确性进行检测的，当然在数据帧中引入了一些传输的保证机制是好的，但可以反过来说，同样我们会引入更多的开销，这样可能会增加应用层交互的延迟。  
最后给大家一个通过Ethereal抓下来的PPP帧，对应上面的说明，看看大家是否可以看懂：  
7E FF 03 C021 01 01 00 17 02 06 00 0A 00 00 05 06 00 0B 42 CB 07 02 08 02 0D 03 06 7E

WINRAR二进制：

C241 2435 做Little Endian 处理 3524 41C2



winrar文件报告：



ＢＭＰ印鉴水印

对唯一文件做跟踪，检验文件是否有变生变化

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 文件块 | 类型 | 字段 | 地址 | 说明 |
| BMP文件头 | Int | BfType | 0--1 | 规定必须是"BM"作为识别BMP文件的标志 |
| Long | BfSize | 2--5 | 给出了文件的长度，单位为字节 |
| Int | BfReserved1 | 6--7 | 保留，平时为0 |
| Int | BfReserved2 | 8--9 | 保留，平时为0 |
| Long | BfOffbits | 10--13 | 给出了位图阵列相对于文件头的偏移 |
| 点位图信息 | Long | Width | 18--21 | 给出了位图的宽度 |
| Long | Height | 24--27 | 给出了位图的高度 |
| Int | BitCount | 28--29 | 像素的位数，如24为24位真彩、8为256色等 |
| Long | BitSizeImage | 34--37 | 确定图像字节数的多少，但通常此项为0 |
| 位图阵列 | 从第38字节开始，位图阵列中保存有全部图像像素的RGB三分量值。 | | | |

印鉴水印存放在第6--9字节之间的这两段保留字段中。

可以加密后放到一个单独的文件中，也可以放到最后。

但是注意，单纯的CRC校验并不安全，网上有工具可以改的，

其原理也在文件最后加上几个字节就变成任意想要的校验值了。

建议：

1：双校验，再加上另一种校验，比如md5。

2：检查文件大小，精确到字节。

### MD5

### SHA1