循环冗余校验 CRC 的算法分析和程序实现

西南交通大学计算机与通信工程学院　刘东

摘要　通信的目的是要把信息及时可靠地传送给对方，因此要求一个通信系统传输消息必须可靠与快速，在数字通信系统中可靠与快速往往是一对矛盾。为了解决可靠性，通信系统都采用了差错控制。本文详细介绍了循环冗余校验 CRC（Cyclic Redundancy Check）的差错控制原理及其算法实现。

关键字　通信 循环冗余校验　CRC-32　CRC-16　CRC-4

概述

在数字通信系统中可靠与快速往往是一对矛盾。若要求快速，则必然使得每个数据码元所占地时间缩短、波形变窄、能量减少，从而在受到干扰后产生错误地可能性增加，传送信息地可靠性下降。若是要求可靠，则使得传送消息地速率变慢。因此，如何合理地解决可靠性也速度这一对矛盾，是正确设计一个通信系统地关键问题之一。为保证传输过程的正确性，需要对通信过程进行差错控制。差错控制最常用的方法是自动请求重发方式（ARQ）、向前纠错方式（FEC）和混合纠错（HEC）。在传输过程误码率比较低时，用 FEC 方式比较理想。在传输过程误码率较高时，采用 FEC 容易出现“乱纠”现象。HEC 方式则式 ARQ 和 FEC 的结合。在许多数字通信中，广泛采用 ARQ 方式，此时的差错控制只需要检错功能。实现检错功能的差错控制方法很多，传统的有：奇偶校验、校验和检测、重复码校验、恒比码校验、行列冗余码校验等，这些方法都是增加数据的冗余量，将校验码和数据一起发送到接受端。接受端对接受到的数据进行相同校验，再将得到的校验码和接受到的校验码比较，如果二者一致则认为传输正确。但这些方法都有各自的缺点，误判的概率比较高。

循环冗余校验 CRC（Cyclic Redundancy Check）是由分组线性码的分支而来，其主要应用是二元码组。编码简单且误判概率很低，在通信系统中得到了广泛的应用。下面重点介绍了 CRC 校验的原理及其 算法实现。

一、循环冗余校验码（CRC）

CRC 校验采用多项式编码方法。被处理的数据块可以看作是一个 n 阶的二进制多项式，由 $a_{n-1}x^{n-1}+a_{n-2}x^{x-2}+\cdots+a_1x+a_0$。如一个 8 位二进制数 10110101 可以表示为：$1x^7+0x^6+1x^5+1x^4+0x^3+1x^2+0x+1$。多项式乘除法运算过程与普通代数多项式的乘除法相同。多项式的加减法运算以 2 为模，加减时不进，错位，和逻辑异或运算一致。

采用 CRC 校验时，发送方和接收方用同一个生成多项式 g（x）,并且 g（x）的首位和最后一位的系数必须为 1。CRC 的处理方法是：发送方以 g（x）去除 t（x），得到余数作为 CRC 校验码。校验时，以计算的校正结果是否为 0 为据，判断数据帧是否出错。

CRC 校验可以 100％地检测出所有奇数个随机错误和长度小于等于 k（k 为 g（x）的阶数）的突发错误。所以 CRC 的生成多项式的阶数越高，那么误判的概率就越小。CCITT 建议：2048 kbit/s 的 PCM 基群设备采用 CRC-4 方案，使用的 CRC 校验码生成多项式 g（x）= x^4+x+1。采用 16 位 CRC 校验，可以保证在 10^{14} bit 码元中只含有一位未被检测出的错误[2]。在 IBM 的同步数据链路控制规程 SDLC 的帧校验序列 FCS 中，使用 CRC-16，其生成多项式 g（x）= $x^{16}+x^{15}+x^2+1$；而在 CCITT 推荐的高级数据链路控制规程 HDLC 的帧校验序列 FCS 中，使用 CCITT-16，其生成多项式 g（x）= $x^{16}+x^{15}+x^5+1$。CRC-32 的生成多项式 g（x）

$= x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$。CRC-32 出错的概率比 CRC-16 低$10^{-5}$倍[4]。由于 CRC-32 的可靠性，把 CRC-32 用于重要数据传输十分合适，所以在通信、计算机等领域运用十分广泛。在一些 UART 通信控制芯片（如 MC6582、Intel8273 和 Z80-SIO）内，都采用了 CRC 校验码进行差错控制；以太网卡芯片、MPEG 解码芯片中，也采用 CRC-32 进行差错控制。

二、CRC 校验码的算法分析

CRC 校验码的编码方法是用待发送的二进制数据 t（x）除以生成多项式 g（x），将最后的余数作为 CRC 校验码。其实现步骤如下：

（1） 设待发送的数据块是 m 位的二进制多项式 t（x），生成多项式为 r 阶的 g（x）。在数据块的末尾添加 r 个 0，数据块的长度增加到 m+r 位，对应的二进制多项式为 $x^r t(x)$。

（2） 用生成多项式 g（x）去除 $x^r t(x)$，求得余数为阶数为 r-1 的二进制多项式 y（x）。此二进制多项式 y（x）就是 t（x）经过生成多项式 g（x）编码的 CRC 校验码。

（3） 用 $x^r t(x)$ 以模 2 的方式减去 y（x），得到二进制多项式 $x^r t'(x)$。$x^r t'(x)$ 就是包含了 CRC 校验码的待发送字符串。

从 CRC 的编码规则可以看出，CRC 编码实际上是将代发送的 m 位二进制多项式 t（x）转换成了可以被 g（x）除尽的 m+r 位二进制多项式 $x^r t'(x)$，所以解码时可以用接受到的数据去除 g（x），如果余数位零，则表示传输过程没有错误；如果余数不为零，则在传输过程中肯定存在错误。许多 CRC 的硬件解码电路就是按这种方式进行检错的。同时 $x^r t'(x)$ 可以看做是由 t（x）和 CRC 校验码的组合，所以解码时将接收到的二进制数据去掉尾部的 r 位数据，得到的就是原始数据。

为了更清楚的了解 CRC 校验码的编码过程，下面用一个简单的例子来说明 CRC 校验码的编码过程。由于 CRC-32、CRC-16、CCITT 和 CRC-4 的编码过程基本一致，只有位数和生成多项式不一样。为了叙述简单，用一个 CRC-4 编码的例子来说明 CRC 的编码过程。

设待发送的数据 t（x）为 12 位的二进制数据 100100011100；CRC-4 的生成多项式为 g（x）$= x^4 + x + 1$，阶数 r 为 4，即 10011。首先在 t（x）的末尾添加 4 个 0 构成 $x^4 t(x)$，数据块就成了 1001000111000000。然后用 g（x）去除 $x^4 t(x)$，不用管商是多少，只需要求得余数 y（x）。下表为给出了除法过程。

除数次数	被除数/ g（x）/结果	余数
0	1 001000111000000	100111000000
	1 0011	
	0 000100111000000	
1	1 00111000000	1000000
	1 0011	
	0 00001000000	
2	1 000000	1100
	1 0011	
	0 001100	

从上面表中可以看出，CRC 编码实际上是一个循环移位的模 2 运算。对 CRC-4，我们假设有

一个 5 bits 的寄存器，通过反复的移位和进行 CRC 的除法，那么最终该寄存器中的值去掉最高一位就是我们所要求的余数。所以可以将上述步骤用下面的流程描述：

//reg 是一个 5 bits 的寄存器
把 reg 中的值置 0.
把原始的数据后添加 r 个 0.
While (数据未处理完)
Begin
If (reg 首位是 1)
    reg = reg XOR 0011.
把 reg 中的值左移一位，读入一个新的数据并置于 register 的 0 bit 的位置。
End
reg 的后四位就是我们所要求的余数。

这种算法简单，容易实现，对任意长度生成多项式的 G（x）都适用。在发送的数据不长的情况下可以使用。但是如果发送的数据块很长的话，这种方法就不太适合了。它一次只能处理一位数据，效率太低。为了提高处理效率，可以一次处理 4 位、8 位、16 位、32 位。由于处理器的结构基本上都支持 8 位数据的处理，所以一次处理 8 位比较合适。

为了对优化后的算法有一种直观的了解，先将上面的算法换个角度理解一下。在上面例子中，可以将编码过程看作如下过程：

由于最后只需要余数，所以我们只看后四位。构造一个四位的寄存器 reg，初值为 0，数据依次移入 reg0（reg 的 0 位），同时 reg3 的数据移出 reg。有上面的算法可以知道，只有当移出的数据为 1 时，reg 才和 g（x）进行 XOR 运算；移出的数据为 0 时，reg 不与 g（x）进行 XOR 运算，相当与和 0000 进行 XOR 运算。就是说，reg 和什么样的数据进行 XOR 移出的数据决定。由于只有一个 bit，所以有 2^1 种选择。上述算法可以描述如下，

//reg 是一个 4 bits 的寄存器
初始化 t[]={0011,0000}
把 reg 中的值置 0.
把原始的数据后添加 r 个 0.
While (数据未处理完)
Begin
把 reg 中的值左移一位，读入一个新的数据并置于 register 的 0 bit 的位置。
reg = reg XOR t[移出的位]
End

上面算法是以 bit 为单位进行处理的，可以将上述算法扩展到 8 位，即以 Byte 为单位进行处理，即 CRC-32。构造一个四个 Byte 的寄存器 reg，初值为 0x00000000，数据依次移入 reg0（reg 的 0 字节，以下类似），同时 reg3 的数据移出 reg。用上面的算法类推可知，移出的数据字节决定 reg 和什么样的数据进行 XOR。由于有 8 个 bit，所以有 2^8 种选择。上述算法可以描述如下：

//reg 是一个 4 Byte 的寄存器

初始化 t[]＝{…}//共有 2^8 ＝256 项

把 reg 中的值置 0.
把原始的数据后添加 r/8 个 0 字节.
While (数据未处理完)
Begin
把 reg 中的值左移一个字节，读入一个新的字节并置于 reg 的第 0 个 byte 的位置。
reg = reg XOR t[移出的字节]
End

算法的依据和多项式除法性质有关。如果一个 m 位的多项式 t（x）除以一个 r 阶的生成多项式

g（x），$t(x)=a_{m-1}x^{m-1}+a_{m-2}x^{m-2}+\cdots+a_2x^2+a_1x^1+a_0$，将每一位$a_kx^k$（0=<k<m）提出来，在后面不足 r 个 0 后，单独去除 g（x），得到的余式位$y_k(x)$。则将$y_{m-1}(x)\oplus y_{m-2}(x)\oplus\cdots\oplus y_0(x)$后得到的就是 t（x）由生成多项式 g（x）得到的余式。对于 CRC-32，可以将每个字节在后面补上 32 个 0 后与生成多项式进行运算，得到余式和此字节唯一对应，这个余式就是上面算法种 t[]中的值，由于一个字节有 8 位，所以 t[]共有2^8＝256 项。多项式运算性质可以参见参考文献[1]。这种算法每次处理一个字节，通过查表法进行运算，大大提高了处理速度，故为大多数应用所采用。

三、CRC-32 的程序实现。

为了提高编码效率，在实际运用中大多采用查表法来完成 CRC-32 校验，下面是产生 CRC-32 校验吗的子程序。

```
unsigned long    crc_32_tab[256]={
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0xe963a535,
0x9e6495a3,0x0edb8832,…, 0x5a05df1b, 0x2d02ef8d
};//事先计算出的参数表，共有 256 项，未全部列出。

unsigned long GenerateCRC32(char xdata * DataBuf,unsigned long    len)
{
    unsigned long oldcrc32;
    unsigned long crc32;
    unsigned long oldcrc;
    unsigned    int charcnt;
                char c,t;
    oldcrc32 = 0x00000000; //初值为 0
    charcnt=0;
    while (len--) {
                    t= (oldcrc32 >> 24) & 0xFF;      //要移出的字节的值
                    oldcrc=crc_32_tab[t];            //根据移出的字节的值查表
                    c=DataBuf[charcnt];              //新移进来的字节值
                    oldcrc32= (oldcrc32 << 8) | c;     //将新移进来的字节值添在寄存器末字节中
                    oldcrc32=oldcrc32^oldcrc;        //将寄存器与查出的值进行 xor 运算
                    charcnt++;
    }
        crc32=oldcrc32;
        return crc32;
}
```

参数表可以先在 PC 机上算出来，也可在程序初始化时完成。下面是用于计算参数表的 c 语言子程序，在 Visual C++ 6.0 下编译通过。

```
#include <stdio.h>
unsigned long int crc32_table[256];
unsigned long int ulPolynomial = 0x04c11db7;
unsigned long int Reflect(unsigned long int ref, char ch)
{    unsigned long int value(0);
    // 交换 bit0 和 bit7，bit1 和 bit6，类推
    for(int i = 1; i < (ch + 1); i++)
    {           if(ref & 1)
                value |= 1 << (ch - i);
                ref >>= 1;      }
```

```
        return value;
    }
    init_crc32_table()
    {   unsigned long int crc,temp;
        // 256 个值
        for(int i = 0; i <= 0xFF; i++)
        {   temp=Reflect(i, 8);
            crc32_table[i]= temp<< 24;
            for (int j = 0; j < 8; j++){
              unsigned long int t1,t2;
              unsigned long int flag=crc32_table[i]&0x80000000;
              t1=(crc32_table[i] << 1);
              if(flag==0)
                 t2=0;
              else
                 t2=ulPolynomial;
              crc32_table[i] =t1^t2 ;        }
            crc=crc32_table[i];
            crc32_table[i] = Reflect(crc32_table[i], 32);
        }
    }
```

结束语

CRC 校验由于实现简单，检错能力强，被广泛使用在各种数据校验应用中。占用系统资源少，用软硬件均能实现，是进行数据传输差错检测地一种很好的手段。

参考文献

[1] 王新梅 肖国镇. 纠错码－原理与方法.西安：西安电子科技大学出版社，2001

[2] 罗伟雄 韩力 原东昌 丁志杰 通信原理与电路. 北京：北京理工大学出版社，1999

[3] 王仲文 ARQ 编码通信.北京：机械工业出版社，1991

[4] Ross Williams, A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS. Document url: http://www.repairfaq.org/filipg/ ,1993