一文講透CRC校驗碼-附贈C語言實例

嵌入式從0到1 今天

編者薦語：

玩嵌入式，離不開C語言，Linux更是嵌入式系統中應用廣泛的系統。彭老師C語言、Linux功底很強，想學C語言、Linux的，推薦關注彭老師哈。

以下文章來源於一口Linux

一口Linux

畢業於南京理工，曾任職中興通訊，擔任華清遠見教學總監，長期分享網絡嵌入式Linux驅動干貨

一口君最近工作用到CRC校驗，順便整理本篇文章和大家一起研究。

一、CRC概念

1. 什麼是CRC？

CRC（Cyclic Redundancy Checksum）是一種糾錯技術，代表循環冗餘校驗和。

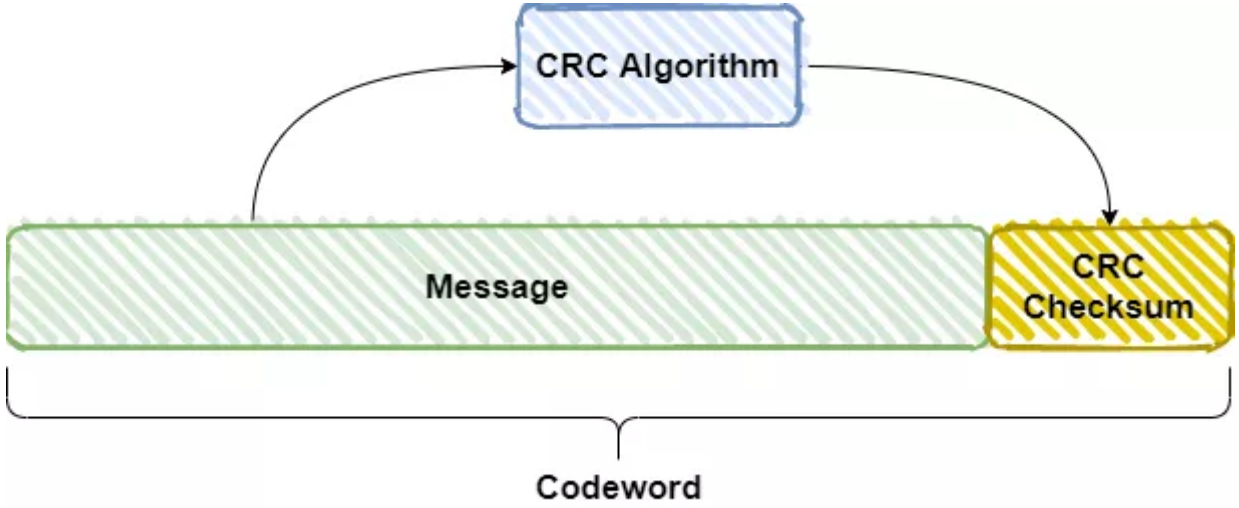
數據通信領域中最常用的一種差錯校驗碼，其信息字段和校驗字段長度可以任意指定，但要求通信雙方定義的CRC標準一致。主要用來檢測或校驗數據傳輸或者保存後可能出現的錯誤。它的使用方式可以說明如下圖所示：

在數據傳輸過程中，無論傳輸系統的設計再怎麼完美，差錯總會存在，這種差錯可能會導致在鏈路上傳輸的一個或者多個幀被破壞(出現比特差錯，0變為1，或者1變為0)，從而接受方接收到錯誤的數據。

為盡量提高接受方收到數據的正確率，在接收方接收數據之前需要對數據進行差錯檢測，當且僅當檢測的結果為正確時接收方才真正收下數據。檢測的方式有多種，常見的有

2. 使用方法概述

循環冗餘校驗



發送方計算機使用某公式計算出被傳送數據所含信息的一個值，並將此值附在被傳送數據後，接收方計算機則對同一數據進行相同的計算，應該得到相同的結果。

如果這兩個CRC結果不一致，則說明發送中出現了差錯，接收方計算機可要求發送方計算機重新發送該數據。

3. 應用廣泛

在諸多檢錯手段中，CRC是最著名的一種。CRC的全稱是循環冗餘校驗，其特點是:檢錯能力強，開銷小，易於用編碼器及檢測電路實現。從其檢錯能力來看，它所不能發現的錯誤的機率僅為0.0047%以下。

從性能上和開銷上考慮，均遠遠優於奇偶校驗及算術和校驗等方式。

因而，在數據存儲和數據通訊領域，CRC無處不在：著名的通訊協議X.25的FCS(幀檢錯序列)採用的是CRC-CCITT，WinRAR、NERO、ARJ、LHA等壓縮工具軟件採用的是CRC32，磁盤驅動器的讀寫採用了CRC16，通用的圖像存儲格式GIF、TIFF等也都用CRC作為檢錯手段。

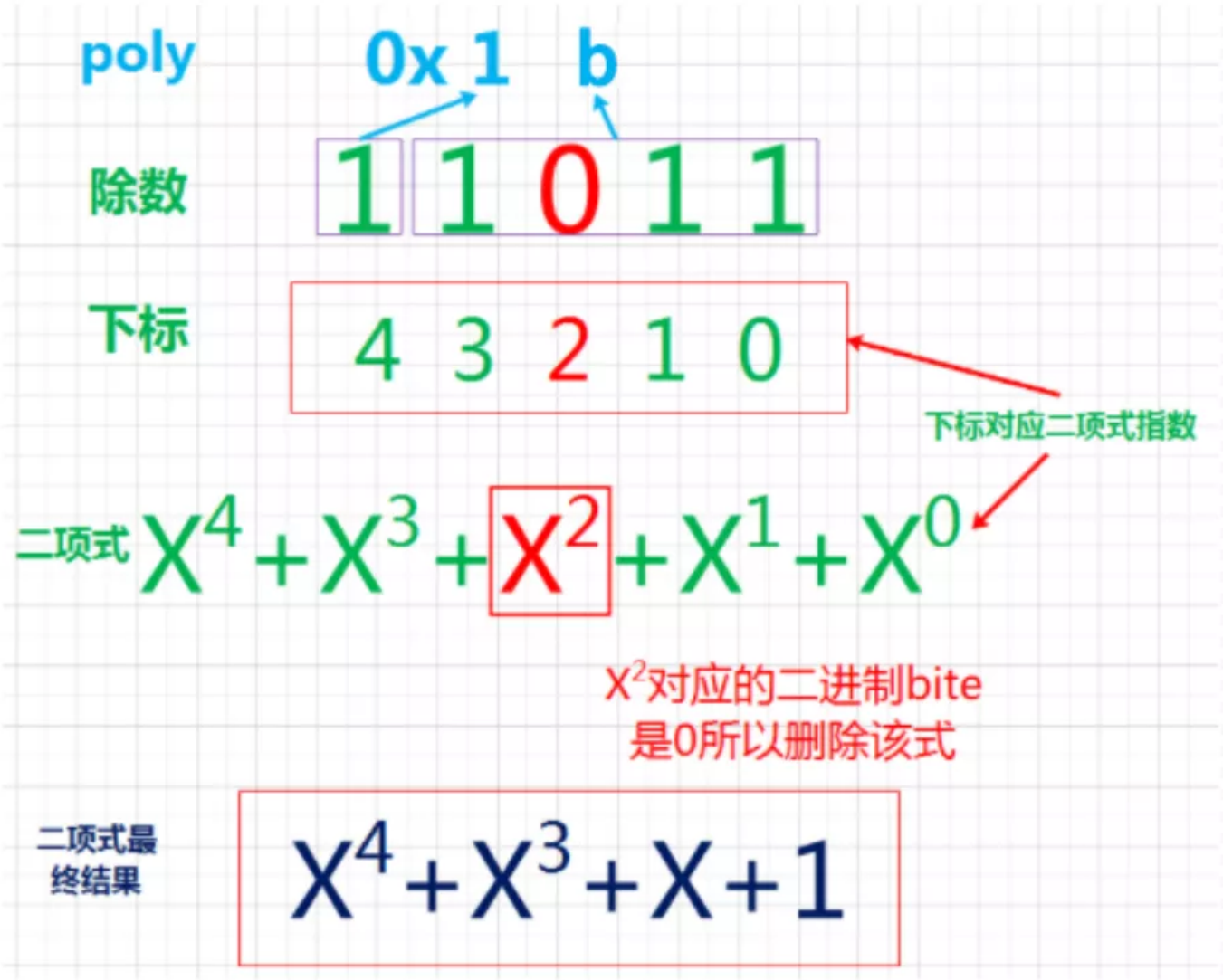
二、CRC名稱的定義

這裡需要知道幾個組成部分或者說計算概念：多項式公式、多項式簡記式、數據寬度、初始值、結果異或值、輸入值反轉、輸出值反轉、參數模型。

1、多項式公式

對於CRC標準除數，一般使用多項式（或二項式）公式表示，如下圖中除數11011（poly值為0x1b）的二項式為G(X)=X

這里特別注意一下位數問題，除數的位數為二項式最高次冪+1（4+1=5），這個很重要。



2、多項式簡記式

通過對CRC的基本了解我們知道，多項式的首尾必定為1，而這個1的位置在下一步計算一定為0，所以就把前面這個1給省略掉了，出現了一個叫簡記式的東西，如上例中除數11011的簡記式為1011，很多看過CRC高級語言源碼的人會知道，對於CRC_16標準下G(X)=X 後面會對這個用法做一個說明。

3、數據寬度

數據寬度指的就是CRC校驗碼的長度（二進制位數），知道了CRC的運算概念和多項式，就可以理解這個概念了，CRC長度始終要比除數位數少1，與簡記式長度是一致的。

以上三個數據就是我們經常能夠用到的基本數據

4、初始值與結果異或值

在一些標準中，規定了初始值，則數據在進行上述二項式運算之前，需要先將要計算的數據與初始值的最低字節進行異或，然後再與多項式進行計算。

而在結果異或值不為零的情況下，則需要將計算得到的CRC結果值再與結果異或值進行一次異或計算，得到的最終值才是我們需要的CRC校驗碼。

這裡可以看出，初始值與結果值的位數要求與數據寬度一致。

5、輸入值反轉與輸出值反轉

輸入值反轉的意思是在計算之前先將二項式反轉，然後再用得到的新值和數據進行計算。如對於G(X)=X

輸出值反轉則是將最終得到的CRC結果反轉。

通常，輸入值反轉後的結果值也會是反轉的，所以這兩個選項一般是同向的，我們只有在在線CRC計算器中會看到自由選擇正反轉的情況存在。

三、常見的CRC算法

雖然CRC可以任意定義二項式、數據長度等，但沒有一個統一的標準的話，就會讓整個計算變得非常的麻煩。但實際上，不同的廠家經常採用不同的標準算法，這裡列出了一些國際常用的模型表：

名稱	多項式	表示法	應用舉例
CRC-8	X	0X107	
CRC-12	X	0X180F	telecom systems
CRC-16	X	0X18005	Bisync, Modbus, USB, ANSI X3.28, SIA DC-07, many others; also known as CRC-16 and CRC-16-ANSI
CRC-CCITT	X	0X11021	ISO HDLC, ITU X.25, V.34/V.41/V.42, PPP-FCS
CRC-32	X	0x104C11DB7	ZIP, RAR, IEEE 802 LAN/FDDI, IEEE 1394, PPP-FCS
CRC-32C	X	0x11EDC6F41	iSCSI, SCTP, G.hn payload, SSE4.2, Btrfs, ext4, Ceph

四、CRC校驗算法前置知識

在學習CRC校驗算法之前，先複習一下CRC會涉及的主要幾個主要的算法。

1. 異或

異或，就是不同為1，相同為0，運算符號是^。

```
0^0 = 0
0^1 = 1
1^1 = 0
1^0 = 1
```

異或運算存在如下幾個規律，需要了解。

```
0^x = x 即0 异或任何数等于任何数
1^x = ~x 即1异或任何数等于任何数取反
x^x = 0 即任何数与自己异或，结果为0
a ^ b = b ^ a 交换律
a ^ (b ^ c) = (a ^ b) ^c 结合律
```

2. 模2加法

模2加法相對於普通的算術加法，主要的區別在模2加法，不做進位處理。具體結果如下。0+0 = 0 0+1 = 1 1+1 = 0 1+0 = 1 我們發現模2加法的計算結果，同異或運算結果一模一樣。進一步推演，我們會發現，異或運算的5個規律，同樣適合於模2加法。這裡，就不在一一列舉了。

3. 模2減法

模2減法相對於普通的算術減法，主要的區別在模2減法，不做借位處理。具體結果如下。0-0 = 0 0-1 = 1 1-1 = 0 1-0 = 1 我們發現模2減法的計算結果，同模2加法，以及異或的運算結果一模一樣。進一步推演，我們會發現，異或運算的5個規律，同樣適合於模2減法。這裡，就不在一一列舉了。

4. 模2除法

模2除法相對於普通的算術除法，主要的區別在模2除法，它既不向上位借位，也不比較除數和被除數的相同位數值的大小，只要以相同位數進行相除即可。

五、CRC原理

CRC原理：在K位信息碼（目標發送數據）後再拼接R位校驗碼，使整個編碼長度為N位，因此這種編碼也叫（N,K）碼。

通俗的說，就是在需要發送的信息後面附加一個數（即校驗碼），生成一個新的發送數據發送給接收端。這個數據要求能夠使生成的新數據被一個特定的數整除。這裡的整除需要引入模2除法的概念。

那麼，CRC校驗的具體做法就是

（1）選定一個標準除數（K位二進制數據串）

（2）在要發送的數據（m位）後面加上K-1位0，然後將這個新數（M+K-1位）以模2除法的方式除以上面這個標準除數，所得到的餘數也就是該數據的CRC校驗碼（注：餘數必須比除數少且只少一位，不夠就補0）

（3）將這個校驗碼附在原m位數據後面，構成新的M+K-1位數據，發送給接收端。

（4）接收端將接收到的數據除以標準除數，如果餘數為0則認為數據正確。

注意：CRC校驗中有兩個關鍵點：

一是要預先確定一個發送端和接收端都用來作為除數的二進制比特串（或多項式）；

二是把原始幀與上面選定的除進行二進制除法運算，計算出FCS。

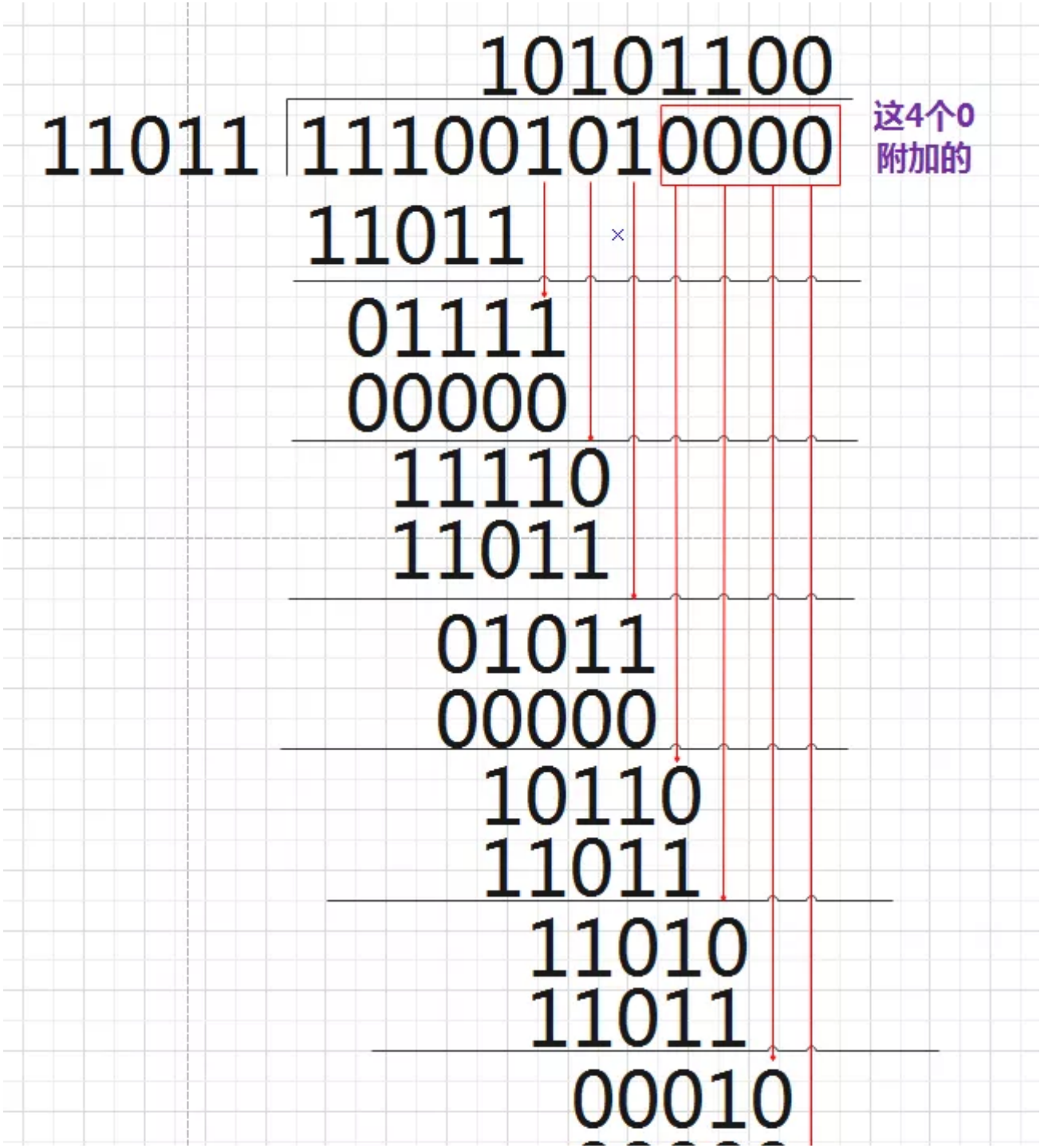
前者可以隨機選擇，也可按國際上通行的標準選擇，但最高位和最低位必須均為“1”

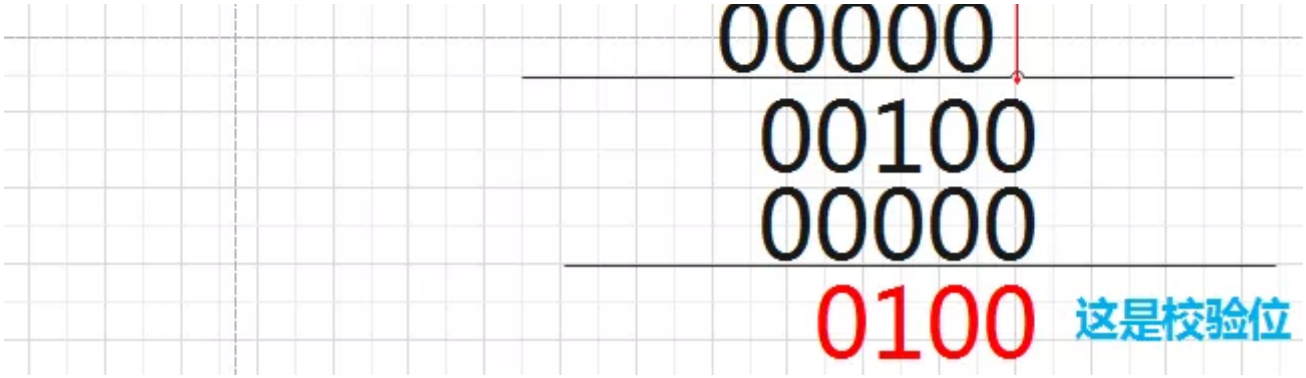
六、循環冗餘的計算

實例：

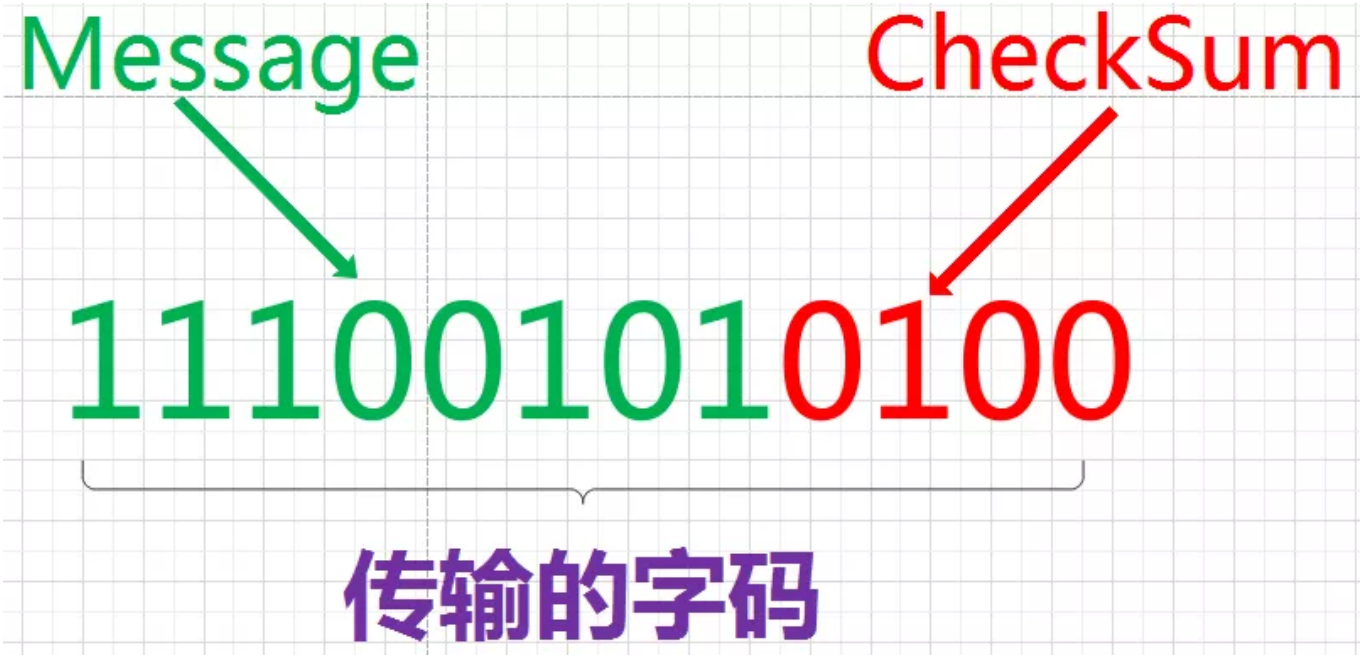
由於CRC-32、CRC-16、CCITT和CRC-4的編碼過程基本一致，只有位數和生成多項式不一樣，下面就舉例，來說明CRC校驗碼生成過程。

對於數據1110 0101（16#E5），以指定除數11011求它的CRC校驗碼，其過程如下：


10101100
11011 11100101 0000
这4个0
附加的
11011
01111
00000
11110
11011
01011
00000
10110
11011
11010
11011
00010



使用上面計算的校驗和和消息數據，可以創建要傳輸的碼字。



有時候，我們需要填充checksum到製定的位置，這就涉及到字節序問題，建議用memcpy()進行拷貝。

七、代碼實現

實現算法參考網絡相關代碼，進行整理並驗證，可直接使用。crc.c

```c
/*
 *一口Linux
 *2021.6.21
 *version: 1.0.0
*/
#include "crc.h"
#include <stdio.h>

typedef enum {
 REF_4BIT = 4,
 REF_5BIT = 5,
 REF_6BIT = 6,
 REF_7BIT = 7,
 REF_8BIT = 8,
 REF_16BIT = 16,
 REF_32BIT = 32
}REFLECTED_MODE;

uint32_t ReflectedData(uint32_t data, REFLECTED_MODE mode)
{
 data = ((data & 0xffff0000) >> 16) | ((data & 0x0000ffff) << 16);
 data = ((data & 0xff00ff00) >> 8) | ((data & 0x00ff00ff) << 8);
 data = ((data & 0xf0f0f0f0) >> 4) | ((data & 0x0f0f0f0f) << 4);
 data = ((data & 0xcccccccc) >> 2) | ((data & 0x33333333) << 2);
 data = ((data & 0xaaaaaaaa) >> 1) | ((data & 0x55555555) << 1);
```

```
 switch (mode)
 {
 case REF_32BIT:
  return data;
 case REF_16BIT:
  return (data >> 16) & 0xffff;
 case REF_8BIT:
  return (data >> 24) & 0xff;
 case REF_7BIT:
  return (data >> 25) & 0x7f;
 case REF_6BIT:
  return (data >> 26) & 0x7f;
 case REF_5BIT:
  return (data >> 27) & 0x1f;
 case REF_4BIT:
  return (data >> 28) & 0x0f;
 }
 return 0;
}

uint8_t CheckCrc4(uint8_t poly, uint8_t init, bool refIn, bool refOut, uint8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint8_t i;
 uint8_t crc;

 if (refIn == true)
 {
  crc = init;
```

```
  poly = ReflectedData(poly, REF_4BIT);

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x01)
    {
     crc >>= 1;
     crc ^= poly;
    }
    else
    {
     crc >>= 1;
    }
   }
  }

  return crc ^ xorOut;
 }
 else
 {
  crc = init << 4;
  poly <<= 4;

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x80)
```

```
    {
     crc <<= 1;
     crc ^= poly;
    }
    else
    {
     crc <<= 1;
    }
   }
  }

  return (crc >> 4) ^ xorOut;
 }
}

uint8_t CheckCrc5(uint8_t poly, uint8_t init, bool refIn, bool refOut, uint8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint8_t i;
 uint8_t crc;

 if (refIn == true)
 {
  crc = init;
  poly = ReflectedData(poly, REF_5BIT);

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x01)
```

```
    {
     crc >>= 1;
     crc ^= poly;
    }
    else
    {
     crc >>= 1;
    }
   }
  }

  return crc ^ xorOut;
 }
 else
 {
  crc = init << 3;
  poly <<= 3;

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x80)
    {
     crc <<= 1;
     crc ^= poly;
    }
    else
    {
     crc <<= 1;
    }
   }
```

```
  }

  return (crc >> 3) ^ xorOut;
 }
}

uint8_t CheckCrc6(uint8_t poly, uint8_t init, bool refIn, bool refOut, uint8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint8_t i;
 uint8_t crc;

 if (refIn == true)
 {
  crc = init;
  poly = ReflectedData(poly, REF_6BIT);

  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x01)
    {
     crc >>= 1;
     crc ^= poly;
    }
    else
    {
     crc >>= 1;
    }
   }
```

```
    }

    return crc ^ xorOut;
   }
   else
   {
    crc = init << 2;
    poly <<= 2;

    while (length--)
    {
     crc ^= *buffer++;
     for (i = 0; i < 8; i++)
     {
      if (crc & 0x80)
      {
       crc <<= 1;
       crc ^= poly;
      }
      else
      {
       crc <<= 1;
      }
     }
    }

    return (crc >> 2) ^ xorOut;
   }
}

uint8_t CheckCrc7(uint8_t poly, uint8_t init, bool refIn, bool refOut, uint8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
```

```
uint8_t i;
uint8_t crc;

if (refIn == true)
{
 crc = init;
 poly = ReflectedData(poly, REF_7BIT);

 while (length--)
 {
  crc ^= *buffer++;
  for (i = 0; i < 8; i++)
  {
   if (crc & 0x01)
   {
    crc >>= 1;
    crc ^= poly;
   }
   else
   {
    crc >>= 1;
   }
  }
 }

 return crc ^ xorOut;
}
else
{
 crc = init << 1;
 poly <<= 1;
```

```
  while (length--)
  {
   crc ^= *buffer++;
   for (i = 0; i < 8; i++)
   {
    if (crc & 0x80)
    {
     crc <<= 1;
     crc ^= poly;
    }
    else
    {
     crc <<= 1;
    }
   }
  }

  return (crc >> 1) ^ xorOut;
 }
}

uint8_t CheckCrc8(uint8_t poly, uint8_t init, bool refIn, bool refOut, uint8_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint32_t i = 0;
 uint8_t crc = init;

 while (length--)
 {
  if (refIn == true)
  {
   crc ^= ReflectedData(*buffer++, REF_8BIT);
```

```
  }
  else
  {
   crc ^= *buffer++;
  }

  for (i = 0; i < 8; i++)
  {
   if (crc & 0x80)
   {
    crc <<= 1;
    crc ^= poly;
   }
   else
   {
    crc <<= 1;
   }
  }
 }

 if (refOut == true)
 {
  crc = ReflectedData(crc, REF_8BIT);
 }

 return crc ^ xorOut;
}

uint16_t CheckCrc16(uint16_t poly, uint16_t init, bool refIn, bool refOut, uint16_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint32_t i = 0;
 uint16_t crc = init;
```

```
    while (length--)
    {
     if (refIn == true)
     {
      crc ^= ReflectedData(*buffer++, REF_8BIT) << 8;
     }
     else
     {
      crc ^= (*buffer++) << 8;
     }

     for (i = 0; i < 8; i++)
     {
      if (crc & 0x8000)
      {
       crc <<= 1;
       crc ^= poly;
      }
      else
      {
       crc <<= 1;
      }
     }
    }

    if (refOut == true)
    {
     crc = ReflectedData(crc, REF_16BIT);
    }

    return crc ^ xorOut;
}
```

```
}

uint32_t CheckCrc32(uint32_t poly, uint32_t init, bool refIn, bool refOut, uint32_t xorOut,
 const uint8_t *buffer, uint32_t length)
{
 uint32_t i = 0;
 uint32_t crc = init;

 while (length--)
 {
  if (refIn == true)
  {
   crc ^= ReflectedData(*buffer++, REF_8BIT) << 24;
  }
  else
  {
   crc ^= (*buffer++) << 24;
  }

  for (i = 0; i < 8; i++)
  {
   if (crc & 0x80000000)
   {
    crc <<= 1;
    crc ^= poly;
   }
   else
   {
    crc <<= 1;
   }
  }
 }
```

```
 if (refOut == true)
 {
  crc = ReflectedData(crc, REF_32BIT);
 }

 return crc ^ xorOut;
}

uint32_t CrcCheck(CRC_Type crcType, const uint8_t *buffer, uint32_t length)
{
 switch (crcType.width)
 {
 case 4:
  return CheckCrc4(crcType.poly, crcType.init, crcType.refIn, crcType.refOut,
   crcType.xorOut, buffer, length);
 case 5:
  return CheckCrc5(crcType.poly, crcType.init, crcType.refIn, crcType.refOut,
   crcType.xorOut, buffer, length);
 case 6:
  return CheckCrc6(crcType.poly, crcType.init, crcType.refIn, crcType.refOut,
   crcType.xorOut, buffer, length);
 case 7:
  return CheckCrc7(crcType.poly, crcType.init, crcType.refIn, crcType.refOut,
   crcType.xorOut, buffer, length);
 case 8:
  return CheckCrc8(crcType.poly, crcType.init, crcType.refIn, crcType.refOut,
   crcType.xorOut, buffer, length);
 case 16:
  return CheckCrc16(crcType.poly, crcType.init, crcType.refIn, crcType.refOut,
   crcType.xorOut, buffer, length);
 case 32:
```

```
  return CheckCrc32(crcType.poly, crcType.init, crcType.refIn, crcType.refOut,
   crcType.xorOut, buffer, length);
 }
 return 0;
}
```

crc.h

```
/*
 *一口Linux
 *2021.6.21
 *version: 1.0.0
*/
#ifndef __CRC_H__
#define __CRC_H__

#include <stdint.h>
#include <stdbool.h>

typedef struct {
 uint8_t width;
 uint32_t poly;
 uint32_t init;
 bool refIn;
 bool refOut;
 uint32_t xorOut;
```

```
}CRC_Type;

uint32_t CrcCheck(CRC_Type crcType, const uint8_t *buffer, uint32_t length);

#endif
```

main.c

```
/*
 *一口Linux
 *2021.6.21
 *version: 1.0.0
*/
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "crc.h"

#define LENGTH 8
const uint8_t data[3][LENGTH] = {
 { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 },
 { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 },
 { 0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f }};

typedef struct {
 CRC_Type crcType;
 uint32_t result[3];
```

```
}CRC_Test;

CRC_Test crc4_ITU = { { 4, 0x03, 0x00, true, true, 0x00 }, { 0x0f, 0x0a, 0x0e } };
CRC_Test crc5_EPC = { { 5, 0x09, 0x09, false, false, 0x00 }, { 0x00, 0x0c, 0x17 } };
CRC_Test crc5_ITU = { { 5, 0x15, 0x00, true, true, 0x00 }, { 0x16, 0x0a, 0x17 } };
CRC_Test crc5_USB = { { 5, 0x05, 0x1f, true, true, 0x1f }, { 0x10, 0x09, 0x17 } };
CRC_Test crc6_ITU = { { 6, 0x03, 0x00, true, true, 0x00 }, { 0x1d, 0x30, 0x00 } };
CRC_Test crc7_MMC = { { 7, 0x09, 0x00, false, false, 0x00 }, { 0x57, 0x30, 0x5b } };
CRC_Test crc8 = { { 8, 0x07, 0x00, false, false, 0x00 }, { 0x3e, 0xe1, 0x36 } };
CRC_Test crc8_ITU = { { 8, 0x07, 0x00, false, false, 0x55 }, { 0x6b, 0xb4, 0x63 } };
CRC_Test crc8_ROHC = { { 8, 0x07, 0xff, true, true, 0x00 }, { 0x6b, 0x78, 0x93 } };
CRC_Test crc8_MAXIM = { { 8, 0x31, 0x00, true, true, 0x00 }, { 0x83, 0x60, 0xa9 } };
CRC_Test crc16_IBM = { { 16, 0x8005, 0x0000, true, true, 0x0000 }, { 0xc4f0, 0x2337, 0xa776 } };
CRC_Test crc16_MAXIM = { { 16, 0x8005, 0x0000, true, true, 0xffff }, { 0x3b0f, 0xdcc8, 0x5889 } };
CRC_Test crc16_USB = { { 16, 0x8005, 0xffff, true, true, 0xffff }, { 0x304f, 0xd788, 0x53c9 } };
CRC_Test crc16_MODBUS = { { 16, 0x8005, 0xffff, true, true, 0x0000 }, { 0xcfb0, 0x2877, 0xac36 } };
CRC_Test crc16_CCITT = { { 16, 0x1021, 0x0000, true, true, 0x0000 }, { 0xeea7, 0xfe7c, 0x7919 } };
CRC_Test crc16_CCITT_FALSE = { { 16, 0x1021, 0xffff, false, false, 0x0000 }, { 0x4792, 0x13a7, 0xb546 } };
CRC_Test crc16_X25 = { { 16, 0x1021, 0xffff, true, true, 0xffff }, { 0x6dd5, 0x7d0f, 0xfa6a } };
CRC_Test crc16_XMODEM = { { 16, 0x1021, 0x0000, false, false, 0x0000 }, { 0x76ac, 0x2299, 0x8478 } };
CRC_Test crc16_DNP = { { 16, 0x3D65, 0x0000, true, true, 0xffff }, { 0x7bda, 0x0535, 0x08c4 } };
CRC_Test crc32 = { { 32, 0x04c11db7, 0xffffffff, true, true, 0xffffffff }, { 0x3fca88c5, 0xe0631a53, 0xa4051a26 } };
CRC_Test crc32_MPEG2 = { { 32, 0x4c11db7, 0xffffffff, false, false, 0x00000000 }, { 0x14dbbdd8, 0x6509b4b6, 0xcb09d294 } };

void CrcTest(CRC_Test crcTest)
{
 uint32_t i;
 for (i = 0; i < 3; i++)
```

```
 {
  printf("%08x\t%08x\r\n", CrcCheck(crcTest.crcType, data[i], LENGTH), crcTest.result[i]);
 }
 printf("\r\n");
}

int main(void)
{
 CrcTest(crc4_ITU);
 CrcTest(crc5_EPC);
 CrcTest(crc5_ITU);
 CrcTest(crc5_USB);
 CrcTest(crc6_ITU);
 CrcTest(crc7_MMC);
 CrcTest(crc8);
 CrcTest(crc8_ITU);
 CrcTest(crc8_ROHC);
 CrcTest(crc8_MAXIM);
 CrcTest(crc16_IBM);
 CrcTest(crc16_MAXIM);
 CrcTest(crc16_USB);
 CrcTest(crc16_MODBUS);
 CrcTest(crc16_CCITT);
 CrcTest(crc16_CCITT_FALSE);
 CrcTest(crc16_X25);
 CrcTest(crc16_XMODEM);
 CrcTest(crc16_DNP);
 CrcTest(crc32);
 CrcTest(crc32_MPEG2);

 return 0;
}
```

注意

不同的CRC算法，對00H或FFH數據流的計算結果不一樣，部分算法存在校驗結果也為00H或FFH的情況（也就意味著存儲空間處於初始化狀態時：全0或全1，CRC校驗反而是正確的），在應用中需要注意避免。

完整代碼，後台回復關鍵【crc】即可獲取。

- END -

嵌入式客棧

國產版“灌籃高手”，大家怎麼看待

技術讓夢想更偉大