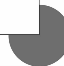
第 3 章

Modbus 协议的相关知识

本章开始具体学习 Modbus 协议的相关知识，特别是对于一些特殊点需要细细体会。



3.1 协议概要

简而言之，Modbus 协议是一种单主/多从的通信协议，其特点是在同一时间，总线上只能有一个主设备，但可以有一个或者多个（最多 247 个）从设备。Modbus 通信总是由主设备发起，当从设备没有收到来自主设备的请求时，不会主动发送数据。从设备之间不能相互通信，主设备同时只能启动一个 Modbus 访问事务处理。

主设备可以采用两种方式向从设备发送 Modbus 请求报文，即主设备可以对指定的单个从设备或者线路上所有的从设备发送请求报文，而从设备只能被动接收请求报文后给出响应报文即应答（参见图 3-1）。这两种模式分别如下：

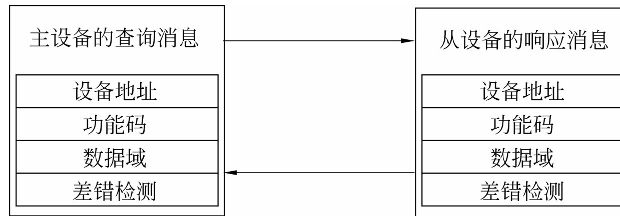


图 3-1　Modbus 请求应答周期

- 单播模式：主设备仅仅寻址单个从设备。从设备接收并处理完请求之后，向主设备返回一个响应报文，即应答。在这种模式下，一个 Modbus 事务处理包含两个报文：一个是主设备的请求报文，一个是从设备的响应报文。

每个从设备必须有唯一的地址（地址范围 1～247），这样才能区别于其他从设备从而可以独立被寻址，而主设备不占用地址。

- 广播模式：此种模式下，主设备可以向所有的从设备发送请求指令。而从设备在接收到广播指令后，仅仅进行相关指令的事务处理而不要求返回应

答。基于此，广播模式下，请求指令必须是 Modbus 标准功能中的写指令。

根据 Modbus 标准协议的要求，所有从设备必须接收广播模式下的写指令，且地址 0 被保留用来识别广播通信。

1. 请求

主设备发送的请求报文主要包括从设备地址（或者广播地址 0）、功能码、传输的数据以及差错检测字段。

查询消息中的功能码告之被选中的从设备要执行何种功能。数据段包含了从设备要执行功能的任何附加信息。例如功能代码 03 要求从设备读保持寄存器并返回其内容。

数据段必须包含要告之从设备的信息：从何寄存器开始读取及要读取的寄存器数量。差错检测域为从设备提供了一种验证消息内容是否正确的方法。

2. 应答

从设备的应答报文包括地址、功能码、差错检测域等。

如果从设备产生一个正常的回应，则在回应消息中的功能代码是在查询消息中的功能代码的回应。数据段包括了从设备收集的数据，如寄存器值或状态。如果有错误发生，功能代码将被修改以用于指出回应消息是错误的，同时数据段包含了描述此错误信息的代码。差错检测域允许主设备确认消息内容是否可用。

对于串行链路来说，又存在两种传输帧模式：ASCII（American Standard Code for Information Interchange，美国标准信息交换码）模式和 RTU（Remote Terminal Unit）模式。但是，对于同一网络或链路来说，所有的设备必须保持统一，要么统一为 ASCII 模式，要么统一为 RTU 模式，不可共存。相对来说，RTU 模式传输效率更高，因此，在当前普遍的生产环境中 RTU 模式获得了广泛应用，而 ASCII 模式只作为特殊情况下的可选项。

3.2 Modbus 寄存器

3.2.1 寄存器种类说明

Modbus 协议中一个重要的概念是寄存器，所有的数据均存放于寄存器中。最初 Modbus 协议借鉴了 PLC 中寄存器的含义，但是随着 Modbus 协议的广泛应用，寄存器的概念进一步泛化，不再是指具体的物理寄存器，也可能是一块内存区域。Modbus 寄存器根据存放的数据类型以及各自读写特性，将寄存器分为 4 个部分，这 4 个部分可以连续也可以不连续，由开发者决定。寄存器的意义如表 3-1 所示。

表 3-1 Modbus 寄存器

寄存器种类	说 明	与 PLC 类比	举例说明
线圈状态 (Coil Status)	输出端口。可设定端口的输出状态，也可以读取该位的输出状态。可分为两种不同的执行状态，例如保持型或边沿触发型	DO(数字量输出)	电磁阀输出、MOSFET 输出、LED 显示等
离散输入状态 (Input Status)	输入端口。通过外部设定改变输入状态，可读但不可写	DI(数字量输入)	拨码开关、接近开关等
保持寄存器 (Holding Register)	输出参数或保持参数，控制器运行时被设定的某些参数，可读可写	AO(模拟量输出)	模拟量输出设定值，PID 运行参数，变量阀输出大小，传感器报警上限、下限
输入寄存器 (Input Register)	输入参数。控制器运行时从外部设备获得的参数，可读但不可写	AI(模拟量输入)	模拟量输入

3.2.2 寄存器地址分配

Modbus 寄存器地址分配如表 3-2 所示，仍然参照了 PLC 寄存器地址的分配方法。

表 3-2 Modbus 寄存器地址分配

寄存器种类	寄存器 PLC 地址	寄存器 Modbus 协议地址	简称	读写状态
线圈状态	00001～09999	0000H～FFFFH	0x	可读可写
离散输入状态	10001～19999	0000H～FFFFH	1x	只读
保持寄存器	40001～49999	0000H～FFFFH	4x	可读可写
输入寄存器	30001～39999	0000H～FFFFH	3x	只读

该表中的 PLC 地址可以理解为 Modbus 协议地址的变种，在触摸屏和 PLC 编程中应用较为广泛。寄存器 PLC 地址指存放于控制器中的地址，这些控制器可以是 PLC，也可以是触摸屏，或是文本显示器。PLC 地址一般采用 10 进制描述，共有 5 位，其中第一位数字代表寄存器类型。第一位数字和寄存器类型的对应关系如表 3-2 所示。例如，PLC 地址 40001、30002 等。

寄存器 Modbus 协议地址指的是通信时使用的寄存器寻址地址，例如 PLC 地址 40001 对应寻址地址 0x0000，40002 对应寻址地址 0x0001。寄存器寻址地址一般使用 16 进制描述。再如，PLC 寄存器地址 40003 对应的协议地址是 0x0002，PLC 寄存器地址 30003 对应的协议地址也是 0x0002，虽然两个 PLC 寄存器通信时使用相同的 Modbus 协议地址，但是因为不同寄存器的功能码也不相同，需要使用不同的命令访问，所以访问时不存在冲突。

3.3 Modbus 串行消息帧格式

Modbus ASCII 或 RTU 模式仅适用于标准的 Modbus 协议串行网络，它定义

了在这些网络上连续传输的消息段的每一个字节，以及决定怎样将信息打包成消息域和如何解码等功能。

3.3.1 ASCII 消息帧格式

当控制器设为在 Modbus 网络上以 ASCII 模式通信时，在消息中每个 8 位(bit)的字节都将作为两个 ASCII 字符发送。这种方式的主要优点是字符发送的时间间隔可达到 1 秒而不产生错误。

在 ASCII 模式下，消息以冒号(:)字符(ASCII 码 0x3A)开始，以回车换行符结束(ASCII 码 0x0D，0x0A)。消息帧的其他字段(域)可以使用的传输字符是十六进制的 0…9，A…F。处于网络上的 Modbus 设备不断侦测“:”字符，当有一个冒号接收到时，每个设备进入解码阶段，并解码下一个字段(地址域)来判断是否是发给自己的。消息帧中的字符间发送的时间间隔最长不能超过 1 秒，否则接收的设备将认为发生传输错误。

一个典型的 ASCII 消息帧格式如表 3-3 所示。

表 3-3 Modbus ASCII 消息帧格式

起始	地址	功能代码	数　　据	LRC 校验	结束
1 字符 :	2 字符	2 字符	0～2＊252 字符	2 字符	2 字符 CR，LF

3.3.2 RTU 消息帧格式

传输设备(主/从设备)将 Modbus 报文放置在带有已知起始和结束点的消息帧中，这就要求接收消息帧的设备在报文的起始处开始接收，并且要知道报文传输何时结束。另外还必须能够检测到不完整的报文，且能够清晰地设置错误标志。

在 RTU 模式中，消息的发送和接收以至少 3.5 个字符时间的停顿间隔为标

志。实际使用中，网络设备不断侦测网络总线，计算字符间的停顿间隔时间，判断消息帧的起始点。当接收到第一个域(地址域)时，每个设备都进行解码以判断是否是发给自己的。在最后一个传输字符结束之后，一个至少3.5个字符时间的停顿标定了消息的结束，而一个新的消息可在此停顿后开始。另外，在一帧报文中，必须以连续的字符流发送整个报文帧。如果两个字符之间的空闲间隔大于1.5个字符时间，那么认为报文帧不完整，该报文将被丢弃。

很多初学者面对3.5字符时间间隔的概念时，往往陷入迷茫的状态。其实，需要记住的是：

- 3.5时间间隔目的是作为区别前后两帧数据的分隔符。
- 3.5时间间隔只对RTU模式有效。

后续具体编码的章节会针对这一点进一步阐述。

如图3-2所示，Modbus通信时规定主机发送完一组命令必须间隔3.5个字符再发送下一组新命令，这3.5个字符主要用来告诉其他设备这次命令(数据)已结束。这3.5个字符的时间间隔采用以下方式计算：

通常情况下在串行通信中，1个字符包括1位起始位、8位数据位、1位校验位(或者没有)、1位停止位(一般情况下)。这样一般情况下1个字符就包括11位，那么3.5个字符就是$3.5\times11=38.5$位。

而串行通信中波特率的含义是每秒传输的二进制位的个数。例如波特率为9600bps，则意思就是说每1s(也就是1000ms)传输9600个位的数据；反过来说传输9600个二进制位的数据需要1000ms，那么传输38.5个二进制位的数据需要的时间就是：

$$38.5\times(1000/9600)=4.0104167\text{ms}$$

如图3-2所示，Modbus RTU要求相邻两帧数据的起始和结束之间至少有大于等于3.5个字符的时间间隔，那么在波特率为9600bps的情况下，只要大于4.0104167ms即可！

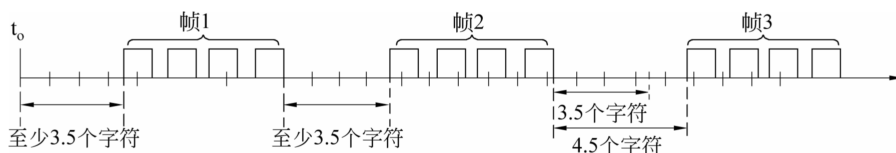


图 3-2　Modbus RTU 相邻帧间隔

每个消息帧的格式如图 3-3 所示。

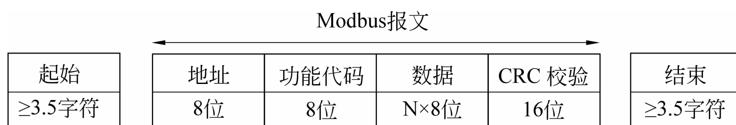


图 3-3　Modbus RTU 帧格式

注意：为了实现 RTU 通信中的时间间隔管理，定时器将引起大量中断处理，在较高的通信波特率下，这将导致 CPU 的沉重负担。为此，协议规定当波特率等于或低于 19200bps 时，需要严格遵守时间间隔；而在波特率大于 19200bps 的情况下，时间间隔使用固定值。建议 1.5 个字符时间间隔为 750μs，帧间时间间隔为 1750μs。

3.3.3　地址域

所谓地址域，指的是 Modbus 通信帧中的地址字段，其内容为从设备地址。Modbus 消息帧的地址域包含 2 个字符（ASCII 模式）或者 1 个字节（RTU 模式）。

消息帧中可能的从设备地址是 0～247（十进制），单个设备的实际地址范围是 1～247（参见表 3-4）。主设备通过将要联络的从设备的地址放入消息中的地址域来选通从设备。当从设备发送回应消息时，它把自己的地址放入回应的地址域中，以便主设备知道是哪一个设备做出回应。

地址 0 用作广播地址，以使所有的从设备都能认识。当 Modbus 协议用于更高级别的网络时，广播方式可能不被允许或以其他方式代替。

表 3-4　Modbus 寻址范围

0	1～247	248～255
广播地址	从站地址	保留

3.3.4　功能码域

功能码在 Modbus 协议中用于表示消息帧的功能。

功能码域由 1 个字节构成，因此其取值范围为 1～255(十进制)。例如，常用的功能码有 03、04、06、16 等，其中 03 功能码的作用是读保持寄存器内容，04 功能码的作用是读输入寄存器内容(输入寄存器和保持寄存器的区别可参考上一节的内容)，06 功能码的内容是预置单个保持寄存器，16 功能码的内容则是预置多个保持寄存器。

从设备根据功能码执行对应的功能，执行完成后，正常情况下则在返回的响应消息帧中设置同样的功能码；如果出现异常，则在返回的消息帧中将功能码最高位(MSB)设置为 1。据此，主设备可获知对应从设备的执行情况。

另外，对于主设备发送的功能码，则从设备根据具体配置来决定是否支持此功能码。如果不支持，则返回异常响应。

3.3.5　数据域

数据域与功能码紧密相关，存放功能码需要操作的具体数据。数据域以字节为单位，长度是可变的，对于有些功能码，数据域可以为空。

具体的功能码和数据域的构成及意义，可参考后续章节的内容，这里暂时省略。

3.4　Modbus 差错校验

在 Modbus 串行通信中，根据传输模式(ASCII 或 RTU)的不同，差错校验域

采用了不同的校验方法。

1. ASCII 模式

在 ASCII 模式中，报文包含一个错误校验字段。该字段由两个字符组成，其值基于对全部报文内容执行的纵向冗余校验（Longitudinal Redundancy Check，LRC）计算的结果而来，计算对象不包括起始的冒号（:）和回车换行符号（CRLF）。

2. RTU 模式

在 RTU 模式中，报文同样包含一个错误校验字段。与 ASCII 模式不同的是，该字段由 16 个比特位共两个字节组成。其值基于对全部报文内容执行的循环冗余校验（Cyclical Redundancy Check，CRC）计算的结果而来，计算对象包括校验域之前的所有字节。

3.4.1 LRC 校验

在 ASCII 模式中，消息是由特定的字符作为帧头和帧尾来分隔的。

一条消息必须以“冒号”（:）字符（ASCII 码为 0x3A）开始，以“回车换行”（CRLF）（ASCII 码为 0x0D 和 0x0A）结束。LRC 校验算法的计算范围为（:）与（CRLF）之间的字符。

从算法本质来说，LRC 域自身为 1 个字节，即包含一个 8 位二进制数据，由发送设备通过 LRC 算法计算，并把计算值附到信息末尾。接收设备在接收信息时，通过 LRC 算法重新计算值，并把计算值与 LRC 字段中接收的实际值进行比较。若两者不同，则产生一个错误，返回一个异常响应帧。即对报文中的所有相邻 2 个 8 位字节相加，丢弃任何进位，然后对结果进行二进制补码，计算出 LRC 值。

必须注意的是，计算 LRC 校验码的时机，是在对报文中每个原始字节进行 ASCII 码编码之前，对每个原始字节进行 LRC 校验的计算操作。

生成 LRC 校验值的过程如下：

(1) 对消息帧中的全部字节相加(不包括起始“:”和结束符“CR-LF”),并把结果送入8位数据区,舍弃进位。

(2) 由0xFF(即全1)减去最终的数据值,产生1的补码(即二进制反码)。

(3) 加“1”产生二进制补码。

以上产生的LRC值占用1个字节,但实际上在通过串行链路由ASCII模式传递消息帧的时候,LRC的结果(1个字节)被编码为2个字节的ASCII字符,并将其放置在ASCII模式报文帧的CR-LF字段之前。

Modbus标准协议英文版提供了LRC的算法。其中参数意义如下:

unsigned char * auchMsg:含有生成LRC所使用的二进制数据的报文缓存区指针。

unsigned short usDataLen:报文缓存区中的字节数。

LRC的详细代码如下:

```
1  /* 函数返回 unsigned char 类型的 LRC 值 */
2  static unsigned char LRC(unsigned char *auchMsg, unsigned short usDataLen)
3  {
4      unsigned char uchLRC = 0;                    /* LRC 字节初始化 */
5  
6      while (usDataLen--)                          /* 遍历报文缓冲区 */
7          uchLRC += *auchMsg++;               /* 缓冲区字节相加,自动舍弃进位 */
8  
9      return ((unsigned char)(-((char)uchLRC)));   /* 返回二进制补码 */
10 }
```

这里举一个简单的例子。假设从设备地址为1,要求读取输入寄存器地址30001的值,则具体的查询消息帧如下:

<u>“:”,“0”,“1”,“0”,“4”,“0”,“0”,“0”,“0”,“0”,“0”,“0”,“1”,“F”,“A”,CR/LF</u>

其中,“F”、“A”即为LRC值在ASCII模式下的形式,即0xFA。

3.4.2 CRC校验

在ModbusRTU传输模式下，通信报文（帧）包括一个基于循环冗余校验（CRC）方法的差错校验字段。

CRC的全称是循环冗余校验，其特点是检错能力极强，开销小，易于用编码器及检测电路实现。从其检错能力来看，它不能发现的错误的几率在0.0047%以下，在Modbus通信中基本可以忽略。CRC校验包括多个版本，常用的CRC校验有CRC-8、CRC-12、CRC-16、CRC-CCITT、CRC-32等。

从性能上和开销上考虑，CRC校验均远远优于奇偶校验及算术和校验等方式，因而在数据存储和数据通信领域，CRC无处不在。例如，著名的通信协议X.25的FCS（帧检错序列）采用的是CRC-CCITT，而WinRAR、NERO、ARJ、LHA等压缩工具软件采用的是CRC-32，磁盘驱动器的读写则采用了CRC-16，通用的图像存储格式GIF、TIFF等也都用CRC作为检错手段。

而Modbus协议中，则采用了CRC-16标准校验方法。在RTU模式下，CRC自身由两个字节组成，即CRC是一个16位的值。CRC字段校验整个报文的内容，无论报文中的单个字节采用何种奇偶校验方式，整个通信报文均可应用CRC-16校验算法。CRC字段作为报文的最后字段添加到整个报文末尾。

有一点需要注意，因为CRC-16由两个字节构成，所以涉及哪个字节放在前面，哪个字节放在后面传输的问题，即大小端模式的选择问题。另外，由于Modbus协议规定寄存器为16位（即两个字节）长度，因此大小端问题的存在给很多初学者造成了困扰。下一节将重点讲解大小端模式。

接收设备在接收信息时，会通过CRC算法重新计算，并把计算值与CRC字段中接收的实际值进行比较。若两者不同，则产生一个错误，并返回一个异常响应报文（帧）告知发送设备。

Modbus协议中的RTU校验码（CRC）计算，运算规则（即CRC计算方法）如下：

(1) 预置一个值为0xFFFF的16位寄存器,此寄存器为CRC寄存器。

(2) 把第1个8位二进制数据(即通信消息帧的第1个字节)与16位的CRC寄存器的相异或,异或的结果仍存放于该CRC寄存器中。

(3) 把CRC寄存器的内容右移一位,用0填补最高位,并检测移出位是0还是1。

(4) 如果移出位为零,则重复步骤(3)(再次右移一位);如果移出位为1,则CRC寄存器与0xA001进行异或。

(5) 重复步骤(3)和(4),直到右移8次,这样整个8位数据全部进行了处理。

(6) 重复步骤(2)~(5),进行通信消息帧下一个字节的处理。

(7) 将该通信消息帧所有字节按上述步骤计算完成后,得到的16位CRC寄存器的高、低字节进行交换。即发送时首先添加低位字节,然后添加高位字节。

(8) 最后得到的CRC寄存器内容即为CRC校验码。

需要强调的一点是,在CRC计算时只有串行链路上每个字符中的8个数据位参与计算,而其他比如起始位及停止位,如有奇偶校验位也包括奇偶校验位,都不参与CRC计算。

常用的CRC-16算法有查表法和计算法。

1. 查表法

CRC查表法是将移位异或的计算结果做成了一个表,就是将0~256放入一个长度为16位的寄存器中的低8位,高8位填充0,然后将该寄存器与多项式0xA001按照上述步骤(3)、(4),直到8位全部移出,最后寄存器中的值就是表格中的数据,高8位、低8位分别单独一个表。

实际上,Modbus标准协议英文版提供了CRC查表算法。

函数的输入参数意义如下:

```
unsigned char *puchMsg;            /*要进行CRC校验的消息*/
unsigned short usDataLen;          /*消息中字节数*/
```

```
1  /* 函数返回 unsigned short(即 2 个字节)类型的 CRC 值 */
2  unsigned short CRC16(unsigned char *puchMsg, unsigned short usDataLen)
3  {
4      unsigned char uchCRCHi=0xFF;        /* 高 CRC 字节初始化 */
5      unsigned char uchCRCLo=0xFF;        /* 低 CRC 字节初始化 */
6      unsigned short uIndex;              /* CRC 循环表中的索引 */
7
8      while (usDataLen--)                 /* 循环处理传输缓冲区消息 */
9      {
10         uIndex=uchCRCHi ^ *puchMsg++;   /* 计算 CRC */
11         uchCRCHi=uchCRCLo ^ auchCRCHi[uIndex];
12         uchCRCLo=auchCRCLo[uIndex];
13     }
14
15     return (uchCRCHi <<8 | uchCRCLo);
16 }
```

其中,auchCRCHi 和 auchCRCLo 分别定义如下:

```
1  static unsigned char auchCRCHi[] =
2  {
3      0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x01,0xC0,0x80,0x41,0x00,
       0xC1,0x81,
4      0x40,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,0x40,
       0x01,0xC0,
5      0x80,0x41,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,
       0x40,0x01,
6      0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x01,0xC0,
       0x80,0x41,
7      0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x00,
       0xC1,0x81,
8      0x40,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,
       0x01,0xC0,
9      0x80,0x41,0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,
       0x41,0x01,
10     0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x00,0xC1,
       0x81,0x40,
```

```
11      0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x01,0xC0,0x80,0x41,0x00,
        0xC1,0x81,
12      0x40,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,
        0x01,0xC0,
13      0x80,0x41,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,
        0x40,0x01,
14      0xC0,0x80,0x41,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x01,0xC0,
        0x80,0x41,
15      0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x00,
        0xC1,0x81,
16      0x40,0x01,0xC0,0x80,0x41,0x01,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,
        0x01,0xC0,
17      0x80,0x41,0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,
        0x41,0x01,
18      0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,0x40,0x01,0xC0,
        0x80,0x41,
19      0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,0x01,0xC0,0x80,0x41,0x00,
        0xC1,0x81,
20      0x40
21  };
22
23  static char auchCRCLo[] =
24  {
25      0x00,0xC0,0xC1,0x01,0xC3,0x03,0x02,0xC2,0xC6,0x06,0x07,0xC7,0x05,
        0xC5,0xC4,
26      0x04,0xCC,0x0C,0x0D,0xCD,0x0F,0xCF,0xCE,0x0E,0x0A,0xCA,0xCB,0x0B,
        0xC9,0x09,
27      0x08,0xC8,0xD8,0x18,0x19,0xD9,0x1B,0xDB,0xDA,0x1A,0x1E,0xDE,0xDF,
        0x1F,0xDD,
28      0x1D,0x1C,0xDC,0x14,0xD4,0xD5,0x15,0xD7,0x17,0x16,0xD6,0xD2,0x12,
        0x13,0xD3,
29      0x11,0xD1,0xD0,0x10,0xF0,0x30,0x31,0xF1,0x33,0xF3,0xF2,0x32,0x36,
        0xF6,0xF7,
30      0x37,0xF5,0x35,0x34,0xF4,0x3C,0xFC,0xFD,0x3D,0xFF,0x3F,0x3E,0xFE,
        0xFA,0x3A,
31      0x3B,0xFB,0x39,0xF9,0xF8,0x38,0x28,0xE8,0xE9,0x29,0xEB,0x2B,0x2A,
        0xEA,0xEE,
```

```
32      0x2E,0x2F,0xEF,0x2D,0xED,0xEC,0x2C,0xE4,0x24,0x25,0xE5,0x27,0xE7,
        0xE6,0x26,
33      0x22,0xE2,0xE3,0x23,0xE1,0x21,0x20,0xE0,0xA0,0x60,0x61,0xA1,0x63,
        0xA3,0xA2,
34      0x62,0x66,0xA6,0xA7,0x67,0xA5,0x65,0x64,0xA4,0x6C,0xAC,0xAD,0x6D,
        0xAF,0x6F,
35      0x6E,0xAE,0xAA,0x6A,0x6B,0xAB,0x69,0xA9,0xA8,0x68,0x78,0xB8,0xB9,
        0x79,0xBB,
36      0x7B,0x7A,0xBA,0xBE,0x7E,0x7F,0xBF,0x7D,0xBD,0xBC,0x7C,0xB4,0x74,
        0x75,0xB5,
37      0x77,0xB7,0xB6,0x76,0x72,0xB2,0xB3,0x73,0xB1,0x71,0x70,0xB0,0x50,
        0x90,0x91,
38      0x51,0x93,0x53,0x52,0x92,0x96,0x56,0x57,0x97,0x55,0x95,0x94,0x54,
        0x9C,0x5C,
39      0x5D,0x9D,0x5F,0x9F,0x9E,0x5E,0x5A,0x9A,0x9B,0x5B,0x99,0x59,0x58,
        0x98,0x88,
40      0x48,0x49,0x89,0x4B,0x8B,0x8A,0x4A,0x4E,0x8E,0x8F,0x4F,0x8D,0x4D,
        0x4C,0x8C,
41      0x44,0x84,0x85,0x45,0x87,0x47,0x46,0x86,0x82,0x42,0x43,0x83,0x41,
        0x81,0x80,
42      0x40
43  };
```

注意：实际编程的时候，auchCRCHi[]和 auchCRCLo[]的定义应该放在函数 CRC16()之前。

查表法可以进一步简化如下：

```
1  unsigned short CRC16(unsigned char * puchMsg,unsigned short usDataLen)
2  {
3      static const unsigned short usCRCTable[] =
4      {
5          0X0000,0XC0C1, 0XC181, 0X0140, 0XC301, 0X03C0, 0X0280, 0XC241,
6          0XC601, 0X06C0, 0X0780, 0XC741, 0X0500, 0XC5C1, 0XC481, 0X0440,
7          0XCC01, 0X0CC0, 0X0D80, 0XCD41, 0X0F00, 0XCFC1, 0XCE81, 0X0E40,
8          0X0A00, 0XCAC1, 0XCB81, 0X0B40, 0XC901, 0X09C0, 0X0880, 0XC841,
```

```
9           0XD801, 0X18C0, 0X1980, 0XD941, 0X1B00, 0XDBC1, 0XDA81, 0X1A40,
10          0X1E00, 0XDEC1, 0XDF81, 0X1F40, 0XDD01, 0X1DC0, 0X1C80, 0XDC41,
11          0X1400, 0XD4C1, 0XD581, 0X1540, 0XD701, 0X17C0, 0X1680, 0XD641,
12          0XD201, 0X12C0, 0X1380, 0XD341, 0X1100, 0XD1C1, 0XD081, 0X1040,
13          0XF001, 0X30C0, 0X3180, 0XF141, 0X3300, 0XF3C1, 0XF281, 0X3240,
14          0X3600, 0XF6C1, 0XF781, 0X3740, 0XF501, 0X35C0, 0X3480, 0XF441,
15          0X3C00, 0XFCC1, 0XFD81, 0X3D40, 0XFF01, 0X3FC0, 0X3E80, 0XFE41,
16          0XFA01, 0X3AC0, 0X3B80, 0XFB41, 0X3900, 0XF9C1, 0XF881, 0X3840,
17          0X2800, 0XE8C1, 0XE981, 0X2940, 0XEB01, 0X2BC0, 0X2A80, 0XEA41,
18          0XEE01, 0X2EC0, 0X2F80, 0XEF41, 0X2D00, 0XEDC1, 0XEC81, 0X2C40,
19          0XE401, 0X24C0, 0X2580, 0XE541, 0X2700, 0XE7C1, 0XE681, 0X2640,
20          0X2200, 0XE2C1, 0XE381, 0X2340, 0XE101, 0X21C0, 0X2080, 0XE041,
21          0XA001, 0X60C0, 0X6180, 0XA141, 0X6300, 0XA3C1, 0XA281, 0X6240,
22          0X6600, 0XA6C1, 0XA781, 0X6740, 0XA501, 0X65C0, 0X6480, 0XA441,
23          0X6C00, 0XACC1, 0XAD81, 0X6D40, 0XAF01, 0X6FC0, 0X6E80, 0XAE41,
24          0XAA01, 0X6AC0, 0X6B80, 0XAB41, 0X6900, 0XA9C1, 0XA881, 0X6840,
25          0X7800, 0XB8C1, 0XB981, 0X7940, 0XBB01, 0X7BC0, 0X7A80, 0XBA41,
26          0XBE01, 0X7EC0, 0X7F80, 0XBF41, 0X7D00, 0XBDC1, 0XBC81, 0X7C40,
27          0XB401, 0X74C0, 0X7580, 0XB541, 0X7700, 0XB7C1, 0XB681, 0X7640,
28          0X7200, 0XB2C1, 0XB381, 0X7340, 0XB101, 0X71C0, 0X7080, 0XB041,
29          0X5000, 0X90C1, 0X9181, 0X5140, 0X9301, 0X53C0, 0X5280, 0X9241,
30          0X9601, 0X56C0, 0X5780, 0X9741, 0X5500, 0X95C1, 0X9481, 0X5440,
31          0X9C01, 0X5CC0, 0X5D80, 0X9D41, 0X5F00, 0X9FC1, 0X9E81, 0X5E40,
32          0X5A00, 0X9AC1, 0X9B81, 0X5B40, 0X9901, 0X59C0, 0X5880, 0X9841,
33          0X8801, 0X48C0, 0X4980, 0X8941, 0X4B00, 0X8BC1, 0X8A81, 0X4A40,
34          0X4E00, 0X8EC1, 0X8F81, 0X4F40, 0X8D01, 0X4DC0, 0X4C80, 0X8C41,
35          0X4400, 0X84C1, 0X8581, 0X4540, 0X8701, 0X47C0, 0X4680, 0X8641,
36          0X8201, 0X42C0, 0X4380, 0X8341, 0X4100, 0X81C1, 0X8081, 0X4040
37      };
38
39      unsigned char nTemp;
40      unsigned short usRegCRC = 0xFFFF;
41
42      while (usDataLen--)
43      {
44          nTemp = *puchMsg++ ^ usRegCRC;
```

```
45          usRegCRC >>=8;
46          usRegCRC ^=usCRCTable[nTemp];
47      }
48      return usRegCRC;
49  }
```

查表法的特点是：以字节为单位进行计算，速度快、语句少，但表格占用一定的程序空间。

2. 计算法

计算法按位计算。这个方法可以适用于所有长度的数据校验，最为灵活；但由于是按位计算，其效率并不是最优，只适用于对速度不敏感的场合。基本的算法如下：

输入参数的意义：

```
unsigned char *puchMsg;          /* 要进行 CRC 校验的消息 */
unsigned short usDataLen;        /* 消息中的字节数 */
```

```
1   /* 函数返回 unsigned short(即 2 个字节)类型的 CRC 值 */
2   unsigned short CRC16(unsigned char *puchMsg, unsigned short usDataLen)
3   {
4       int i, j;                                   /* 循环变量 */
5       unsigned short usRegCRC =0xFFFF;            /* 用于保存 CRC 值 */
6
7       for(i =0; i <usDataLen; i++)                /* 循环处理传输缓冲区消息 */
8       {
9           usRegCRC ^= *puchMsg++;                 /* 异或算法得到 CRC 值 */
10          for(j =0; j <8; j++)                    /* 循环处理每个 bit 位 */
11          {
12              if (usRegCRC & 0x0001)
13                  usRegCRC =usRegCRC >>1 ^ 0xA001;
14              else
15                  usRegCRC >>=1;
16          }
17      }
```

```
18
19     return usRegCRC;
20 }
```

这里举一个简单的例子。假设从设备地址为 1,要求读取输入寄存器地址 30001 的值,则 RTU 模式下具体的查询消息帧如下:

0x01, 0x04, 0x00, 0x00, 0x00, 0x01, 0x31, 0xCA

其中,0xCA31 即为 CRC 值。因为 Modbus 规定发送时 CRC 必须低字节在前,高字节在后,因此实际的消息帧的发送顺序为 0x31,0xCA。

3.5 字节序和大小端

在学习 Modbus 协议时,字节序和大小端是一个非常容易忽视而又容易造成困扰的问题。

3.5.1 来历

很直观地可以知道,在 Modbus 寄存器中对于一个由两个字节组成的 16 位整数,在内存中存储这两个字节有两种方法:一种是将低序字节存储在起始地址,这称为小端(LITTLE-ENDIAN)字节序;另一种方法是将高序字节存储在起始地址,这称为大端(BIG-ENDIAN)字节序。Modbus 通信协议中具体规定了字节高低位的发送顺序,这样自然就引出了字节序和大小端的问题。

另外,或许你曾经仔细了解过什么是大端、小端,也动手编码并测试手头上的机器上是大端还是小端的程序,甚至还编写了大端、小端转换程序;但过了一段时间之后,再看到大端和小端这两个词,脑中很快浮起了自己曾经做过的工作,却总是想不起究竟哪种是大端,哪种是小端,然后又去查找以前写的记录。更让人不快的是,这种经历反反复复,让你十分困扰。

在理解这对概念之前，先看看大端和小端这两个令人迷惑的术语究竟是如何产生的？

实际上，大端和小端可以追溯到 1726 年 Jonathan Swift 所著的《格列佛游记》，其中一篇讲到有两个国家因为吃鸡蛋究竟是先打破较大的一端还是先打破较小的一端而争执不休，甚至爆发了战争。1981 年 10 月，Danny Cohen 的文章《论圣战以及对和平的祈祷》(On holy wars and a plea for peace)将这一对词语引入了计算机界。这么看来，所谓大端和小端，也就是 big-endian 和 little-endian，其实是从描述鸡蛋的部位而引申到对计算机地址的描述。也可以说，它们是从一个俚语衍化来的计算机术语。稍有些英语常识的人都会知道，如果单靠字面意思来理解俚语，那是很难猜到它的正确含义的。在计算机里，对于地址的描述，很少用"大"和"小"来形容；对应地，用的更多的是"高"和"低"；很不幸，这对术语直接按字面翻译过来就成了"大端"和"小端"，让人产生迷惑也不是很奇怪的事了。

3.5.2 为什么会有大小端

为什么会有大小端模式之分呢？

这是因为在计算机系统中，是以字节为单位的，每个地址单元都对应着一个字节。一个字节为 8 位(bit)。在 C 语言中除了 8 位的 char 型之外，还有 16 位的 short 型，32 位的 long 型(要看具体的编译器)。另外，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。

例如一个 16 位的 short 型 x，在内存中的地址为 0x0010，x 的值为 0x1122，那么 0x11 为高字节，0x22 为低字节。对于大端模式，就将 0x11 放在低地址中，即 0x0010 中；0x22 放在高地址中，即 0x0011 中。对于小端模式，刚好相反。常用的 X86 结构是小端模式，而 KEIL C51 则为大端模式。很多 ARM、DSP 都为小端模