**Modbus协议深入讲解**

发布日期: 四月 15, 2018 | 3 评级 | **4.67** out of 5 |

窗体顶端

[Read in English](https://www.ni.com/white-paper/52134/zhs/)

窗体底端

  | [打印](javascript:window.print())

**概览**

Modbus是一种工业协议，于1979年开发，旨在实现自动化设备之间的通信。 Modbus最初是作为通过串行层传输数据的应用级协议实现的，现已扩展到包括通过串行、TCP/IP和用户数据报协议（UDP）的实现。 本文档提供了协议实现的深入讲解。

**目录**

1. [什么是Modbus协议？](https://www.ni.com/white-paper/52134/zhs/#toc1)
2. [协议数据单元(PDU)](https://www.ni.com/white-paper/52134/zhs/#toc2)
3. [应用数据单元（ADU）](https://www.ni.com/white-paper/52134/zhs/#toc3)
4. [新功能代码](https://www.ni.com/white-paper/52134/zhs/#toc4)
5. [网络层](https://www.ni.com/white-paper/52134/zhs/#toc5)
6. [ADU修改](https://www.ni.com/white-paper/52134/zhs/#toc6)

**1. 什么是Modbus协议？**

Modbus是使用主从关系实现的请求 - 响应协议。 在主从关系中，通信总是成对发生 - 一个设备必须发起请求，然后等待响应 - 并且发起设备（主设备）负责发起每次交互。 通常，主设备是人机界面（HMI）或监控和数据采集（SCADA）系统，从设备是传感器、可编程逻辑控制器（PLC）或可编程自动化控制器（PAC）。 这些请求和响应的内容以及发送这些消息的网络层由协议的不同层来定义。

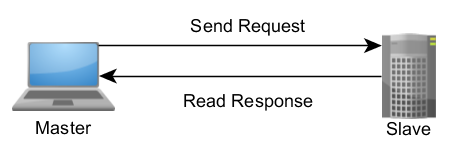


图1. 主从网络关系

**Modbus协议层**

在最初的做法中，Modbus是建立在串行端口之上的单一协议，因此它不能被分成多个层。 随着时间的推移，该协议引入了不同的应用程序数据单元来更改串行通信使用的数据包格式，或允许使用TCP/IP和用户数据报协议（UDP）网络。 这实现了定义协议数据单元（PDU）的核心协议和定义应用数据单元（ADU）的网络层的分离。

[至页首](https://www.ni.com/white-paper/52134/zhs/#top)

**2. 协议数据单元(PDU)**

PDU及其处理代码构成了[Modbus应用协议规范](http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf)的核心。 该规范定义了PDU的格式、协议使用的各种数据概念、如何使用功能代码访问数据，以及每个功能代码的具体实现和限制。

Modbus PDU格式被定义为一个功能代码，后面跟着一组关联的数据。 该数据的大小和内容由功能代码定义，整个PDU（功能代码和数据）的大小不能超过253个字节。 每个功能代码都有一个特定的行为，从设备可以根据所需的应用程序行为灵活地实现这些行为。 PDU规范定义了数据访问和操作的核心概念；但是，从设备可能会以规范中未明确定义的方式处理数据。

**访问Modbus和Modbus数据模型中的数据**

通常，Modbus可访问的数据存储在四个数据库或地址范围的其中一个： 线圈状态、离散量输入、保持寄存器和输入寄存器。 与许多规范一样，名称可能因行业或应用而异。 例如，保持寄存器也可以称为输出寄存器，线圈状态可能称为数字或离散量输出。 这些数据库定义了所包含数据的类型和访问权限。 从设备可以直接访问这些数据，因为这些数据由设备本地托管。 Modbus可访问的数据通常是设备主存的一个子集。 相反，Modbus主设备必须通过各种功能代码请求访问这些数据。 表1中描述了每个区块的行为。

|  |  |  |  |
| --- | --- | --- | --- |
| **内存区块** | **数据类型** | **主设备访问** | **从设备访问** |
| **线圈状态** | 布尔 | 读/写 | 读/写 |
| **离散输入** | 布尔 | 只读 | 读/写 |
| **保持寄存器** | 无符号双字节整型 | 读/写 | 读/写 |
| **输入寄存器** | 无符号双字节整型 | 只读 | 读/写 |

**表1.** Modbus数据模型区块

这些区块允许您限制或允许访问不同的数据元素，并且为应用层提供简化的机制来访问不同的数据类型。

这些区块是完全概念性的。 它们可能作为独立的内存地址存在于给定的系统中，但也可能重叠。 例如，线圈状态1可能存在于与保持寄存器1所代表的字的第一位相同的内存中。 寻址方案完全由从设备定义，其对每个内存区的解释是设备数据模型的重要组成部分。

**数据模型寻址**

该规范将每个区块定义为包含多达65,536（216）个元素的地址空间。 在PDU的定义中，Modbus定义了每个数据元素的地址，范围从0到65,535。但是，每个数据元素的编号从1到n，其中n的最大值为65,536。也就是说，线圈状态1位于地址0的线圈状态区块中，而保持寄存器54位于从机被定义为保持寄存器的内存部分中的地址53。

规范允许的全部范围不需要给定设备实现。 例如，设备可能会选择不执行线圈、离散输入或输入寄存器，而只使用保持寄存器150至175和200至225。这是完全可以接受的，并且通过例外来处理无效的访问尝试。

**数据寻址范围**

虽然规范将不同的数据类型定义为存在于不同的区块中，并为每种类型分配一个本地地址范围，但这并不一定会转化为用于记录或理解给定设备的Modbus可访问内存的直观编址方案。 为了简化对内存区块位置的理解，引入了一种编号方案，其将前缀添加到所讨论的数据的地址中。

例如，设备手册不会引用地址13寄存器14的数据项，而是引用地址4,014,40,014或400,014的数据项。在任何情况下，第一个数字都是4，表示保持寄存器，剩余数字则表示指定地址。 4XXX、4XXXX和4XXXXX的区别取决于设备使用的地址空间。 如果所有65,536个寄存器都在使用中，应该使用4XXXXX符号，因为其允许范围为400,001~465,536。如果只使用几个寄存器，通常的做法是使用范围4,001到4,999。

在这种寻址方案中，每种数据类型都被分配了一个前缀，如表2所示。

|  |  |
| --- | --- |
| **数据区块** | **前缀** |
| **线圈状态** | 0 |
| **离散输入** | 1 |
| **输入寄存器** | 3 |
| **保持寄存器** | 4 |

**表2.** 数据范围前缀

线圈状态存在前缀为0的情况。这意味着4001的引用可能指的是保持寄存器1或线圈4001。因此，建议所有新寻址方案都采用带前导零的6位寻址，并在文档中进行标注。 因此，保持寄存器1的地址为400,001，而线圈4001的地址则为004,001。

**数据地址起始值**

内存地址和参考数字之间的差异会由给定应用程序选择的索引进一步复杂化。 如前所述，保存寄存器1位于地址零。 通常，参考号码是1索引，这意味着给定范围的起始值为1。 因此，400,001就表示为地址0的保持寄存器00001。一些做法选择以零开始其范围，这意味着400,000转换为地址零的保持寄存器。 表3展示了这个概念。

|  |  |  |  |
| --- | --- | --- | --- |
| 地址 | 寄存器编号 | 编号1（1索引，标准） | 编号（0索引，替换） |
| 0 | 1 | 400001 | 400000 |
| 1 | 2 | 400002 | 400001 |
| 2 | 3 | 400003 | 400002 |

表3.寄存器索引方案

1索引范围应用较为广泛，强烈建议采用。 无论哪种情况，每个范围的起始值都应在文档中注明。

**大数据类型**

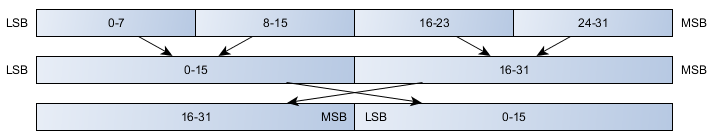
Modbus标准提供了一个相对简单的数据模型，它不包含无符号字和位值之外的其他数据类型。 如果系统的位值对应于螺线管和继电器，并且字值对应于未缩放的ADC值，这是足够的，但对于更高级的系统则可能不足。 因此，许多Modbus实现都包含跨寄存器边界的数据类型。 NI [LabVIEW数据记录和监控（DSC）模块](http://zone.ni.com/reference/en-XX/help/371618J-01/lvmve/dsc_modbus_using/)和[KEPServerEX](http://www.kepware.com/Support_Center/SupportDocuments/Help/modbus_ethernet.pdf)都定义了许多参考类型。 例如，存储在保持寄存器中的字符串遵循标准格式（400,001），但后跟一个十进制数、长度和字符串的字节顺序（400,001.2H是指保持寄存器1中的两个字符串，其中高位字节对应到字符串的第一个字符）。 这是必需的，因为每个请求的大小都是有限的，所以Modbus主机必须知道字符串的确切范围，而不是像NULL那样搜索长度或分隔符。

**位访问**

除了允许访问跨寄存器边界的数据之外，一些Modbus主设备还支持对寄存器中各个位的引用。 这是有好处的，因为它允许设备将相同内存范围内的每种类型的数据组合在一起，而不必将二进制数据分成线圈整体和离散量输入范围。 这通常使用小数点和位索引或数字进行索引，具体取决于如何实现。 也就是说，第一个寄存器的第一位可能是400,001.00或400,001.01。 建议任何文档都要说明所使用的索引方案。

**数据字节顺序**

多寄存器数据（单精度浮点值），可以通过将数据拆分到两个寄存器，轻松地在Modbus中传输。 由于这不是由标准定义的，因此分割的字节顺序没有规定。 尽管每个无符号字必须以网络（big-endian）字节顺序发送以满足标准，但许多设备会颠倒多字节数据的字节顺序。 图2所示的是一个不常见但有效的例子。



**图2.**多字数据的字节顺序交换

请务必理解设备如何将信息存储在内存中并对其进行正确解码。 建议文档写明系统所使用的字顺序。 如果需要灵活性，也可以将Endian添加为系统配置选项，提供基础的编码和解码功能。

**字符串**

字符串可以很容易地存储在Modbus寄存器中。 为了简单起见，一些方法要求字符串长度为2的倍数，并使用控制来填充额外的空间。 字节顺序也是字符串交互中的一个变量。 字符串格式可能包含也可能不包含NULL作为最终值。 举个例子，一些设备的数据存储方法可能如图3所示。

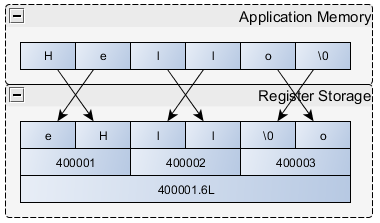


图3. Modbus字符串中的字节顺序反转

**了解功能代码**

与数据模型可能因设备而异不同，功能代码及其数据由标准明确定义。 每个功能都遵循一种模式。 首先，从设备会验证功能代码、数据地址和数据范围等输入。 然后执行所请求的操作并发送与代码相符的响应。 如果此过程中的任何步骤失败，则会向请求程序返回异常。 这些请求的数据传输就称为PDU。

**Modbus PDU**

PDU由一个单字节的功能代码组成，后面跟着多达252字节的针对特定函数的数据。

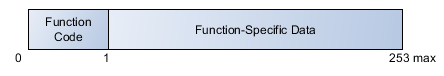


图4. Modbus PDU

功能代码是第一个需要验证的项。 如果功能代码没有被接收到请求的设备识别，则会回应一个异常。 如果功能代码被接受，则从设备根据功能定义开始分解数据。

由于数据包大小限制为253字节，设备可传输的数据量有限。 最常见的功能代码可以240到250字节的从设备数据模型数据，具体取决于代码。

**从函数执行**

不同的函数由数据模型定义访问不同的概念数据块。 一个常见的做法是让代码访问静态内存位置，但其他行为是可用的。 例如，功能代码1（读取线圈状态）和3（读取保持寄存器）可以访问内存中相同的物理位置。 而功能代码3（读取保持寄存器）和16（写入保持寄存器）可以访问内存中完全不同的位置。 因此，建议在定义从数据模型时同时考虑每个功能代码的执行。

无论执行的是何种实际行为，所有的从设备都应该遵循每个请求的简单状态流程图。 图5是代码1读取线圈状态的一个例子。

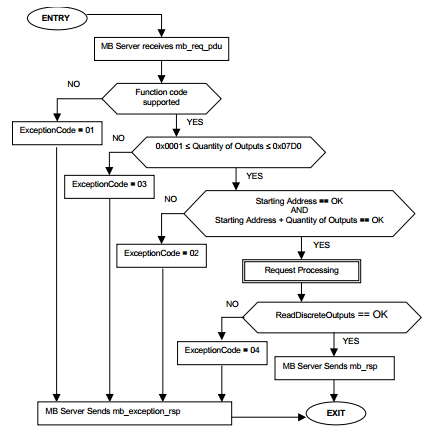


图5.[Modbus协议规范定义](http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf)的读取线圈状态流程图

每个从设备必须验证功能代码、输入数量、起始地址、总范围以及实际进行读取行为的从属定义函数(slave-defined function)的执行。

尽管上面的状态图包含了静态地址范围，但真实系统的需求可能会使静态地址范围与定义的数字有所不同。 在某些情况下，从设备无法传输协议定义的最大字节数。 也就是说，如果主设备请求0x07D0输入，从设备只能用0x0400进行响应。 如果主设备从地址0开始请求125，则这是正确的，但是如果主设备从地址400开始发出相同的请求，最后一个线圈状态将位于地址525，超出了该设备的范围，会导致出现状态图定义的异常02。

**标准功能代码**

每个标准功能代码的定义都包含在说明书中。 即使对于最常见的功能代码，在主设备上启用的功能与从设备可以处理的功能之间也存在不可避免的不匹配。 为了解决这个问题，Modbus TCP规范的早期版本定义了三个一致性类。 官方的[Modbus一致性测试规范](http://www.modbus.org/docs/MBConformanceTestSpec_v3.0.pdf)没有引用这些类，而是在每个功能的基础上定义一致性;但是，这些仍然很容易理解。 建议任何文档都遵循测试规范，并根据其支持的代码而不是传统分类来定义它们的一致性。

**0类代码**

0类代码通常被认为是有用Modbus设备的最低配置，因为它们使主设备能够读取或写入数据模型。

|  |  |
| --- | --- |
| **代码** | **说明** |
| **3** | 读多寄存器 |
| **16** | 写多寄存器 |

表4.0类一致性代码

**1类代码**

1类功能代码由访问所有类型的数据模型所需的其他代码组成。 在原始定义中，这个列表包含功能代码7（读取异常）。 但是，此代码由当前规范定义为仅限于串行的代码。

|  |  |
| --- | --- |
| **代码** | **说明** |
| **1** | 读线圈 |
| **2** | 读离散输入 |
| **4** | 读输入寄存器 |
| **5** | 写单线圈 |
| **6** | 写单寄存器 |
| **7** | 读取异常状态（仅限串行） |

表5. 1类一致性代码

**2类代码**

2类功能代码用于更为专业化的功能，不太常用。 例如，读/写多个寄存器可能有助于减少请求/响应周期的总数，但该行为仍可以用0类代码实现。

|  |  |
| --- | --- |
| **代码** | **说明** |
| **15** | 写多线圈 |
| **20** | 读文件记录 |
| **21** | 写文件记录 |
| **22** | 屏蔽写寄存器 |
| **23** | 读/写多寄存器 |
| **24** | 读取FIFO |

表6. 2类一致性代码

**Modbus封装接口**

Modbus封装接口（MEI）代码功能43用于封装Modbus数据包内的其他数据。 目前，有两个MEI号码可用，13（CANopen）和14（设备识别）。

功能43/14（设备识别）非常有用，因为它允许传送多达256个唯一的对象。 其中一些对象已预定义且预留好，例如供应商名称和产品代码，但应用程序可以将其他对象定义为通用数据集。

此代码并不常用。

**例外**

从设备使用异常来指示各种不良状况，比如错误请求或不正确输入。 但是，异常也可以作为对无效请求的应用程序级响应。 从设备不响应发出异常的请求。 相反，从设备忽略不完整或损坏的请求，并开始等待新的消息传入。

异常以定义好的数据包格式报告给用户。 首先将一个功能代码返回给等同于与原始功能代码的请求主设备，除了设置了最高有效位。 这等同于为原始功能代码的值加上0x80。 异常响应包括一个异常代码来代替与给定函数响应相关的正常数据。

在标准内，四种最常见的异常代码是01,02,03和04。表7介绍了这些代码以及每种功能的标准含义。

|  |  |
| --- | --- |
| **异常代码** | **含义** |
| **01** | 不支持接收到功能代码。 要确认原始功能代码，请从返回值中减去0x80。 |
| **02** | 尝试访问的请求是一个无效地址。 在标准中，只有起始地址和请求的数值超过216时才会发生这种情况。 但是，有些设备可能会限制其数据模型中的地址空间。 |
| **03** | 请求包含不正确的数据。 在某些情况下，这意味着参数不匹配，例如发送的寄存器的数量与“字节数”字段之间的参数不匹配。 更常见的情况是，主机请求的数据比从机或协议允许的要多。 例如，主设备一次只能读取125个保持寄存器，而资源受限的设备可能会将此值限制为更少的寄存器。 例如，主设备一次只能读取125个保持寄存器，而资源受限的设备可能会将此值限制为更少的寄存器。 |
| **04** | 尝试处理请求时发生不可恢复的错误。 这是一个异常的代码，表示请求有效，但从设备无法执行该请求。 |

表7.常见的Modbus异常代码

每个功能代码的状态图至少应包含异常代码01，通常包含异常代码04,02,03，并且任何其他定义的异常代码都是可选的。

[至页首](https://www.ni.com/white-paper/52134/zhs/#top)

**3. 应用数据单元（ADU）**

除了Modbus协议的PDU核心定义的功能外，您还可以使用多种网络协议。 最常见的协议是串行和TCP/IP，但也可以使用其他协议，如UDP。 为了在这些层之间传输Modbus所需的数据，Modbus包含一组适用于每种网络协议的ADU。

**通用特征**

Modbus需要某些功能来提供可靠的通信。 单元ID或地址用在每个ADU格式中，为应用层提供路由信息。 每个ADU都带有一个完整的PDU，其中包含给定请求的功能代码和相关数据。 为了可靠性，每条消息都包含错误检查信息。 最后，所有的ADU都提供了一种机制来确定请求帧的开始和结束，但实现这些机制的方式各不相同。

**标准格式**

ADU的三种标准格式是TCP、远程终端单元（RTU）和ASCII。 RTU和ASCII ADU通常用于串行线路，而TCP则用于现代TCP/IP或UDP/IP网络。

**TCP/IP**

TCP ADU由Modbus应用协议（MBAP）报文头和Modbus PDU组成。 MBAP是一个通用的报文头，依赖于可靠的网络层。 此ADU的格式（包括报文头）如图6所示。

https://www.ni.com/cms/images/devzone/tut/lnsnqtgf1563326460599462186.png

图6.TCP/IP ADU

报文头的数据字段代表其用途。 首先，它包含一个事务处理标识符。 这有助于网络允许同时发生多个未处理的请求。 也就是说，主设备可以发送请求1、2和3。在稍后的时间点，从设备可以以2、1、3的顺序进行响应，并且主设备可以将请求匹配到响应并准确解析数据。 这对以太网网络很有用。

协议标识符通常为零，但您可以使用它来扩展协议的行为。 协议使用长度字段来描述数据包其余部分的长度。 这个元素的位置也表明了这个报文头格式在可靠的网络层上的依赖关系。 由于TCP数据包具有内置的错误检查功能，可确保数据一致性和传送，因此数据包长度可位于报文头的任何位置。 在可靠性较差的网络上（比如串行网络），数据包可能会丢失，其影响是即使应用程序读取的数据流包含有效的事务处理和协议信息，长度信息的损坏也会使报文头无效。 TCP为这种情况提供了适当的保护。

TCP/IP设备通常不适用单元ID。 但是，Modbus是一种常见的协议，因此通常会开发一些网关来将Modbus协议转换为另一种协议。 在最初的预期应用中， Modbus TCP/IP转串行网关用于连接新的TCP/IP网络和旧的串行网络。 这时，单元ID用于确定PDU对应的从设备的地址。

最后，ADU包含一个PDU。 对于标准协议，PDU的长度仍限制为253字节。

**RTU**

RTU ADU看起来要简单得多，如图7所示。

https://www.ni.com/cms/images/devzone/tut/cxgfauhf7193944574851042510.png

图7. RTU ADU

与较为复杂的TCP/IP ADU不同的是，除了核心PDU之外，该ADU仅包含两条信息。 首先，地址用于定义PDU对应的从设备。 在大多数网络中，地址0定义的是“广播”地址。 也就是说，主设备可以发送输出命令到地址0，而所有从设备应处理该请求，但是不做出任何响应。 除了这个地址外，CRC还用于确保数据的完整性。

然而，现在的实现机制远没有那么简单。 数据包的首尾一对沉默时间（silent time），即总线上没有通信的时段。对于9,600的波特率，这个速率大约是4ms。该标准定义了一个最小沉默长度，不论波特率如何，都低于2 ms。

首先，这存在性能缺陷，因为在处理数据包之前设备必须等待空闲时间结束。 然而，更危险的是串行传输引入了不同技术，并且波特率比标准更快。 例如，使用USB/串口转换器电缆，您无法控制数据的数据包和数据传输。 测试表明，结合NI-VISA驱动程序使用USB转串口电缆会在数据流中引入了尺寸可变的大间隙，而这些间隙 – 沉默期 – 会“诱骗”符合规范的代码相信消息是完整的。 由于消息不完整，通常会导致CRC无效，并导致设备将ADU解释为损坏。

除了传输问题之外，现代驱动程序技术还大量提取串行通信，并且通常需要应用程序代码中的轮询机制。 例如，除非通过轮询端口上的字节，[.NET Framework 4.5 SerialPort Class](http://msdn.microsoft.com/en-us/library/system.io.ports.serialport(v=vs.110).aspx)和NI-VISA驱动程序都不提供检测串行线路上的沉默的机制。 这会导致性能降低（如果轮询执行过慢）或CPU使用率过高（如果轮询执行过快）。

解决这些问题的常用方法是打破Modbus PDU和网络层之间的抽象层。 也就是说，串行代码询问Modbus PDU数据包以确定功能代码。 结合数据包中的其他数据，可以发现剩余数据包的长度，从而确定数据包的结尾。 利用这些信息，可以使用更长的超时时间，以允许传输间隙，并且应用程序级的轮询速度可能更慢。 这种机制推荐用于新的开发。 不采用此方法可能会遇到大于预期数量的“损坏”数据包。

**ASCII**

如图8所示，ASCII ADU比RTU更复杂，但也避免了RTU数据包的许多问题。 然而，它自身也有一些缺点。

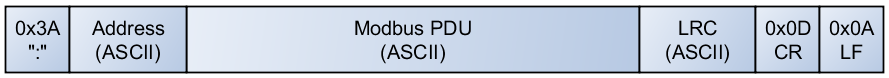
[  
[+] 放大图片](https://www.ni.com/white-paper/52134/zhs/)

图 8. ASCII ADU

为了解决确定数据包大小的问题，ASCII ADU为每个数据包定义了一个明确且唯一的开始和结束。 也就是说，每个数据包以“:”开始 并以回车（CR）和换行符（LF）结束。 另外，像NI-VISA和.NET Framework SerialPort Class这样的串行API可以轻松读取缓冲区中的数据，直到收到特定字符的CR/LF为止。 这些特性有助于在现代应用程序代码中有效地处理串行线路上的数据流。

ASCII ADU的缺点是所有数据都以ASCII编码的十六进制字符进行传输。 也就是说，针对功能代码3（0x03）发送的不是单个字节，而是发送ASCII字符“0”和“3”或0x30/0x33。 这使协议更具可读性，但也意味着必须通过串行网络传输两倍的数据，并且发送和接收应用程序必须能够解析ASCII值。

**扩展Modbus**

Modbus是一种相对简单和开放的标准，可以进行修改以适应给定应用的需求。 这常用于HMI和PLC或PAC之间的通信，因为在这种情况下组织可以控制协议的首尾。 例如，传感器的开发人员更可能遵守书面标准，因为他们通常只控制其从设备的实现，互通性也是可能实现的。

一般来说，不建议修改协议。 本节仅作为对其他人用来调整协议行为的机制的确认。

[至页首](https://www.ni.com/white-paper/52134/zhs/#top)

**4. 新功能代码**

Modbus标准定义了一些功能代码，但也可允许您开发更多的功能代码。 具体而言，功能代码1至64,73至99以及111至127是预留的并保证唯一的公共代码。 其余代码65至72和100至110可由用户自定义。 使用这些用户定义的代码时，您可以使用任何数据结构。 数据甚至可能超过Modbus PDU的标准253字节限制，但应验证整个应用程序以确保其他层在PDU超过标准限制时按预期工作。 高于127的功能代码预留作异常响应。

[至页首](https://www.ni.com/white-paper/52134/zhs/#top)

**5. 网络层**

除了串行和TCP之外，Modbus还可以在许多网络层上运行。 一个[可能](http://jamod.sourceforge.net/)的实现是UDP，因为它适合于Modbus通信风格。 Modbus本质上是基于消息的协议，因此UDP能够发送明确定义的信息包，而不需要任何额外的应用程序级信息，如起始字符或长度，这使得Modbus非常易于实现。 Modbus PDU数据包可以使用标准的UDP API发送，不需要额外的ADU或重新使用现有的ADU，并由另一端完全接收。 虽然由于其内置确认系统，TCP对某些协议有利，但Modbus是在应用层执行确认。 因此，以这种方式使用UDP会消除TCP ADU中的事务处理标识符字段，从而消除了存在多个同时发生的未完成事务的可能性。 因此，主设备必须是同步主设备，或者UDP数据包必须有一个标识符以帮助主设备组织请求和响应。 建议的做法是在UDP网络层上使用TCP/IP ADU。

[至页首](https://www.ni.com/white-paper/52134/zhs/#top)

**6. ADU修改**

最后，应用程序可以选择修改ADU，或使用现有ADU的未使用部分（如TCP）。 例如，TCP定义了一个16位长度字段、一个16位协议和一个8位单元ID。鉴于最大的Modbus PDU是253字节，长度字段的高字节始终为零。 对于Modbus/TCP，协议字段和单元ID始终为零。 协议的简单扩展可以通过将协议字段更改为非零数字并使用两个未使用的字节（单元ID和长度字段的高字节）来发送两个附加PDU的长度，从而同时发送三个数据包（请参阅图9）。

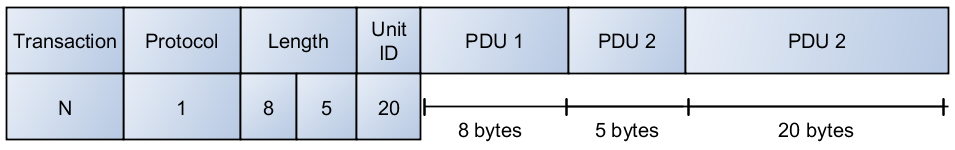
[  
[+] 放大图片](https://www.ni.com/white-paper/52134/zhs/)

图9. TCP ADU修改示例

**其他资源**