野人家园
NetAssist网络调试助手
用户使用手册
Ver 5.0.12

文档最后更新：2024/06/13

技术支持：support@cmsoft.cn

软件授权：南京云想物联网科技有限公司

软件下载 http://www.cmsoft.cn/download/cmsoft/netassist.zip

目录

第一章 NetAssist 简介

野人家园NetAssist网络调试助手，是Windows平台下开发的TCP/IP网络调试工具，集TCP/UDP服务端及客户端于一体，是网络应用开发及调试工作必备的专业工具之一，可以帮助网络应用设计、开发、测试人员检查所开发的网络应用软/硬件产品的数据收发状况，提高开发速度，简化开发复杂度，成为TCP/UDP应用开发调试的得力助手。NetAssist网络调试助手是绿色软件，无需安装，只有一个执行文件，适用于各版本Windows操作系统，不需要微软dotNet框架支持。可以模拟网络客户端或服务器端：可以在一台PC上同时启动多个网络调试助手，并可设置其中一个作为服务端，其它作为客户端，然后进行客户端去与服务端之间的通信调试。典型应用场合：通过网络调试助手与自行开发的网络程序或者网络设备进行通信联调。软件支持UDP、TCP协议，集成服务端与客户端，作为服务端时可以管理多个客户端连接；支持单播/广播；支持ASCII/HEX两种数据收发模式，发送和接收的数据可以在十六进制和ASCII码之间任意转换；可以自动发送校验位，支持多种校验码格式；支持间隔发送、循环发送、批处理发送，输入数据可以直接输入或从外部文件导入；可以保存预定义指令/数据序列,任何时候都可以通过工具面板调用预定义的指令或数据进行发送,便于通信联调。软件界面支持中/英文，默认自适应操作系统的语言环境。

1.1 软件特色

- 绿色软件、只有一个执行文件、无需安装；
- 支持中英文双语言，自动根据操作系统环境选择系统语言类型；
- 支持TCP和UDP协议，支持 TCP Server、TCP Client、UDP三种工作模式，支持UDP单播/广播；
- 支持ASCII/HEX码数据发送,发送和接收的数据可以在十六进制码和ASCII码之间任意转换，支持发送和显示汉字；
- 可以自动发送校验位，支持多种校验格式，如校验和、LRC、BCC、CRC8、CRC16、CRC32、MD5等，其中CRC校验码可任意定制CRC参数(CRC多项式、初始值、输入反转、输出反转、输出异或值)；
- 发送内容支持转义字符。例如，发送框中文本包含诸如\r\n等转义符时，会自动解析成对应的ASCII码进行发送。
- 支持AT指令自动添加回车换行选项，启用该选项时，在发送AT指定时会自动在行尾补全回车换行符；
- 可以通过输入框发送数据,也可以从文件数据源发送数据；
- 支持接收数据自动保存到文件,并且文件类型支持数据文件和日志文件两种格式，其中数据文件只保存接收的数据内容,而日志文件则会保存调试助手完整的数据收发日志信息。
- 支持日志接收模式:启用该选项后在接收窗口显示接收内容时自动显示时间戳等相关信息。
- 支持任意间隔发送，循环发送；

- 接收和发送的文字编码支持ANSI(GBK)与UTF8两种方式，并且接收编码与发送编码可以独立设置，互不影响；
- 支持预定义指令/数据,可通过按键或者自定义快捷键发送预定义指令,预定义指令/数据列表可以按文件的方式保存、导入和导出；
- 支持批量发送指令/数据序列，可设置每条指令的发送延迟，并可按设定顺序及延迟时间依次批量发送。批量定义的数据/指令可以保存、导入和导出。
- 自动保存历史发送记录，可以通过历史记录发送历史数据；
- 支持界面窗口的字体以及背景定制；
- 支持工作界面精简模式（主界面左侧面板可折叠收起）；
- 可定制发送框默认数据内容。

1.2 运行环境

软件运行环境为Windows平台,包括Windows95/WinXP/Vista/Win7/Win8/Win10/WinALL,兼容32位/64位操作系统。

1.3 软件安装

绿色软件,解压后只有一个执行文件,直接运行即可。无需安装(不依赖)Microsoft .NET Framework框架。

1.4 应用场景

网络调试助手通过模拟建立TCP/UDP服务器或客户端，实现对网络设备或者网络应用程序的通信联调。通过网络数据的抓取、记录、分析以及数据/指令的发送控制，实现对目标网络设备或者网络应用程序的通信能力以及通信行为的分析、验证。总的来说，网路调试助手，主要有几下几类应用场景。

（1） 网络终端(仪器设备)的参数设置。工程应用中为了方便终端设备的参数设置，可通过网络调试助手建立到网络终端设备的网络连接,然后直接在网络调试助手中对本地或远程的设备进行参数设置。

（2）网络终端(仪器设备)的远程控制、网络数据的抓取、记录及分析。在工程应用中，某些场景下需要对远程的网络设备发送指令从而实现远程控制操作,或者需要对网络设备的数据进行抓包记录。通过网络调试助手可以定时向网络终端发送指令数据，并自动将接收及发送的报文数据按日志的形式保存至磁盘文件，以便于用户对设备的状态数据进行分析统计。

（3）工控设备/单片机的开发调试。在单片机/嵌入式系统的网络开发过程中，可通过网络调试助手接收单片机设备的网络数据，或者向单片机设备发送网络数据，配合单片机程序开发，验证单片机程序的通信能力以及业务逻辑的准确性；或者通过网络调试助手对单片机设备进行数据疲劳测试（通过批量或者循环指令发送），并记录通信过程中的数据交互日志，实现网络产品在研发过程中的可靠性验证。

（4）客户端模拟。Client-Server(C/S)结构的应用系统开发设计过程中，在服务端软件尚未开发或无法验证时，为了提供系统开发的并行性，客户端开发人员可以通过网络调试助手

模拟服务器端程序，用于验证客户端的通信逻辑的正确性，辅助客户端开发人员完成客户端通信接口协议的开发以及验证。

（5）服务端模拟。Client-Server(C/S)结构的应用系统开发设计过程中，在客户端软件尚未开发或无法验证时，为了提供系统开发的并行性，服务器端开发人员可以通过网络调试助手模拟客户端程序，用于验证服务端的通信逻辑的正确性，辅助服务器端开发人员完成服务器端通信接口协议的开发以及验证。

（6）用于WEB开发人员调试HTTP接口，通过建立TCP连接(服务器或客户端)，抓取HTTP应用交互数据，分析POST或GET请求及响应的报文内容，排查HTTP接口的应用逻辑或者编码方式等错误，为HTTP应用开发解决bug问题提供分析凭据。

（7） 用于FTP或TELNET等基于TCP协议的网络应用软件的开发调试。网络调试助手作为提供给软件开发人员的通信基准工具，保证开发人员可以集中精力于自身业务逻辑。运用网络调试助手进行模拟报文数据的发送以及接收，通过对交互数据的记录及分析，验证目标应用系统的业务逻辑，简化开发复杂度。

1.5 软件界面

NetAssist网络调试助手的主要功能界面如图1-1~1-7所示，包括主界面及工具面板窗口各项功能构成，而具体的功能描述则会在后面章节中具体介绍。

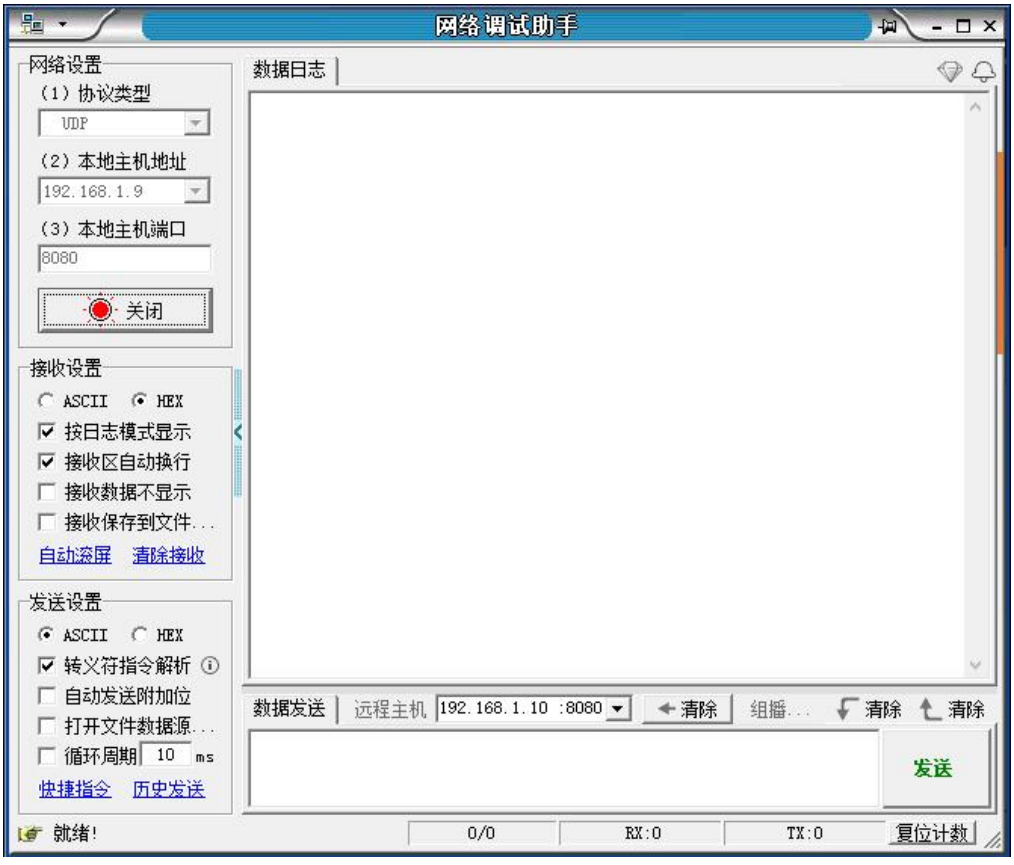


图1-1 软件主界面

图1-2 界面基本构成

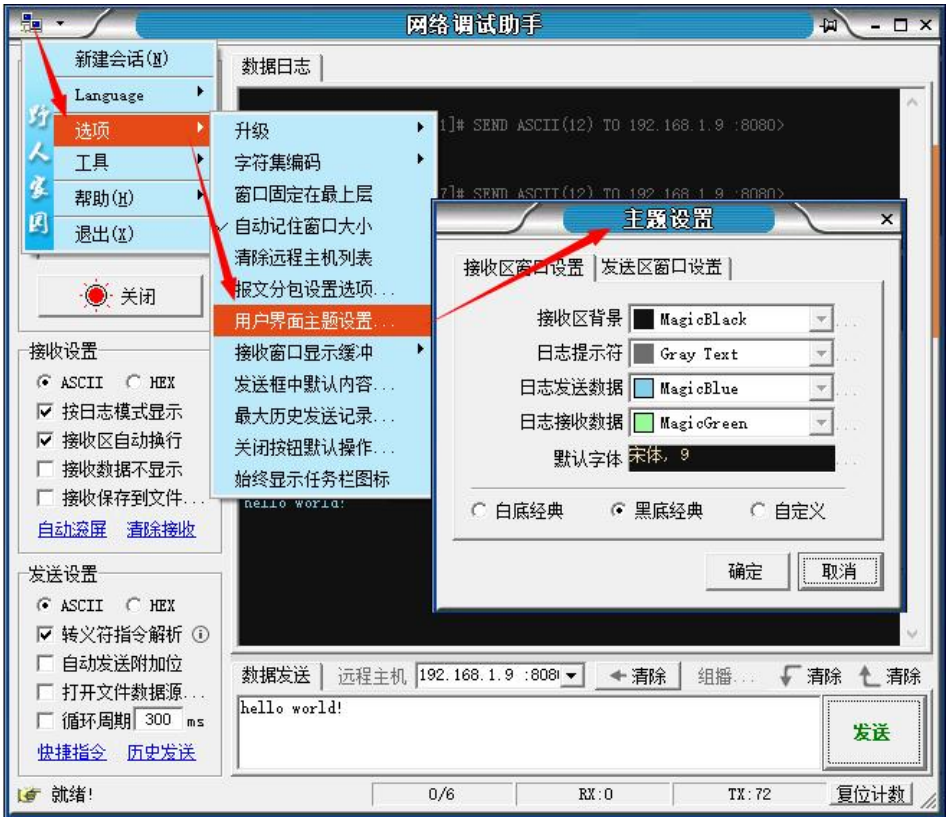


图1-3 界面主题（背景/字体）设置

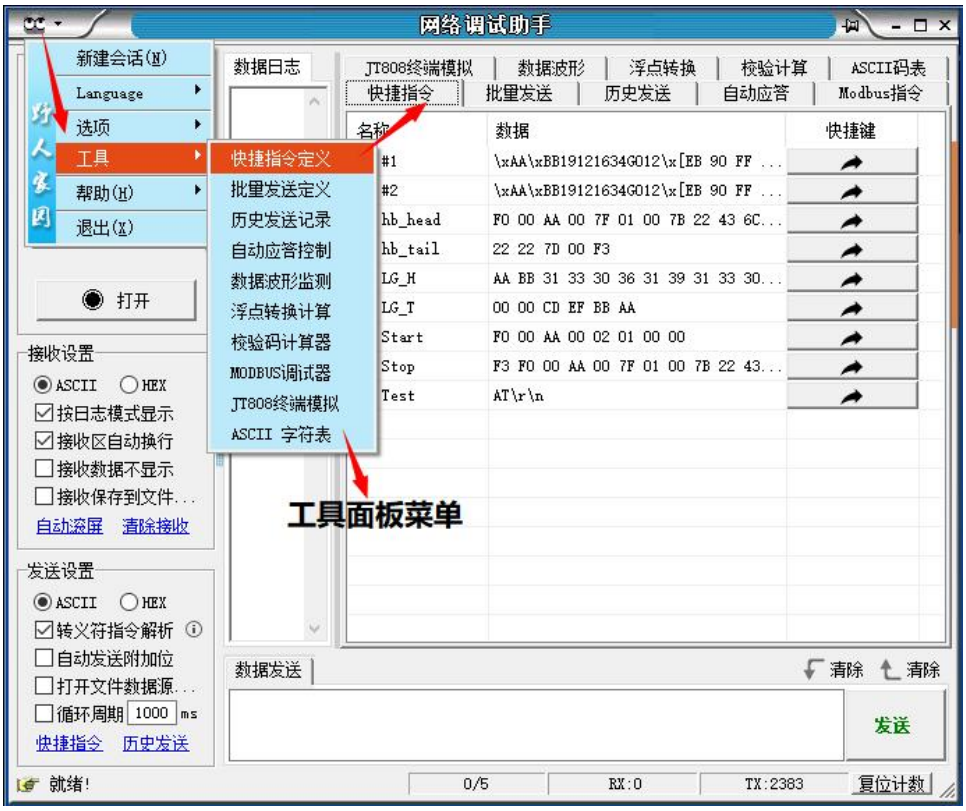


图1-4 工具面板—快捷指令

图1-5 工具面板/批量发送

图1-6 工具面板/自动应答

图1-7 工具面板/历史发送

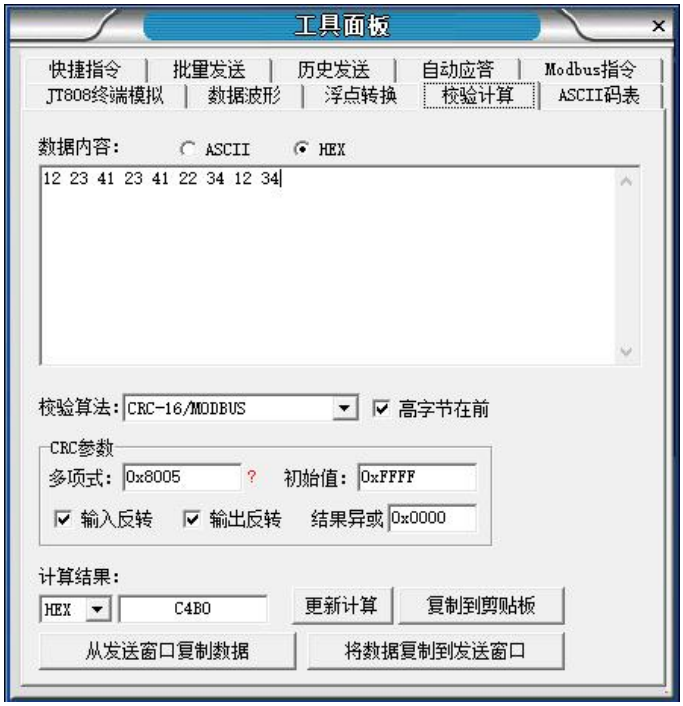


图1-8 工具面板/检验计算器

第二章 网络通信原理

网络调试助手是构建于网络套接字(Socket)组件之上的通信，而Socket本质上是对TCP/IP协议栈模型的封装。要熟练使用网络调试助手，就必须理解网络调试助手的通信原理，也就是要理解Socket以及TCP/IP协议的通信机制。

2.1 TCP/IP协议

TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议）是指能够在多个不同网络间实现信息传输的协议簇。TCP/IP协议不仅仅是TCP和IP两个协议，而是指一个由FTP、SMTP、TCP、UDP、IP等协议构成的协议簇，只是因为在TCP/IP协议中TCP协议和IP协议最具代表性，所以被称为TCP/IP协议。

TCP/IP协议在一定程度上参考了OSI的体系结构。OSI模型共有七层，从下到上分别是物理层、数据链路层、网络层、运输层、会话层、表示层和应用层。但是这显然是有些复杂的。应用层、表示层、会话层三个层次提供的服务相差不是很大，所以在TCP/IP协议中，它们被合并为应用层；因为数据链路层和物理层的内容相差不多，所以在TCP/IP协议中它们被归并在数据链路层的一个层次里。最终，TCP/IP协议栈模型简化为4个层次，自底而上分别是数据链路层、网络层、传输层和应用层，如下图所示。每一层完成不同的功能，且通过若干协议来实现，上层协议使用下层协议提供的服务。

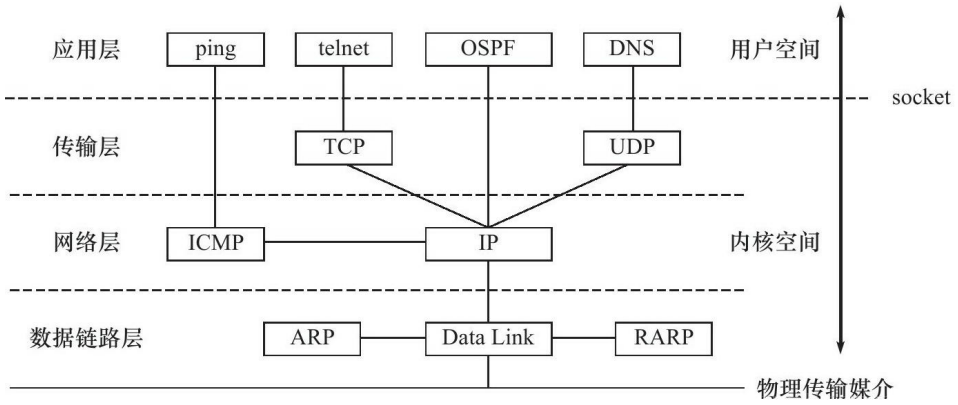


图2-1 TCP/IP协议栈4层模型

（1）数据链路层

数据链路层实现了网卡接口的网络驱动程序，以处理数据在物理媒介（比如以太网、令牌网等）上的传输。数据链路层两个常用的协议是ARP协议（Address Resolve Protocol，地址解析协议）和RARP协议（Reverse Address Resolve Protocol，逆地址解析协议）。它们实现了IP地址和机器物理地址（通常是MAC地址）之间的相互转换。网络层使用IP地址寻址一台机器，而数据链路层使用物理地址寻址一台机器，因此网络层必须先将目标机器的IP地址转化成其物理地址，才能使用数据链路层提供的服务，这就是ARP协议的用途。RARP协议仅用于网络上的某些无盘工作站。因为缺乏存储设备，无盘工作站无法记住自己的IP地址，但它们可以利用网卡上的物理地址来向网络管理者（服务器或网络管理软件）查询自身的IP地址。运行RARP服务的网络管理者通常存有该网络上所有机器的物理地址到IP地址的映射。

（2）网络层

网络层实现数据包的选路和转发。WAN（Wide Area Network，广域网）通常使用众多分级的路由器来连接分散的主机或LAN（Local Area Network，局域网），因此，通信的两台主机一般不是直接相连的，而是通过多个中间节点（路由器）连接的。网络层的任务就是选择这些中间节点，以确定两台主机之间的通信路径。同时，网络层对上层协议隐藏了网络拓扑连接的细节，使得在传输层和网络应用程序看来，通信的双方是直接相连的。

网络层最核心的协议是IP协议（Internet Protocol，因特网协议）。IP协议根据数据包的目的IP地址来决定如何投递它。如果数据包不能直接发送给目标主机，那么IP协议就为它寻找一个合适的下一跳（next hop）路由器，并将数据包交付给该路由器来转发。多次重复这一过程，数据包最终到达目标主机，或者由于发送失败而被丢弃。可见，IP协议使用逐跳（hop by hop）的方式确定通信路径。

网络层另外一个重要的协议是ICMP协议（Internet Control Message Protocol，因特网控制报文协议）。它是IP协议的重要补充，主要用于检测网络连接。ICMP报文分为两大类：差错报文和查询报文。差错报文，这类报文主要用来回应网络错误，比如目标不可到达和重定向；查询报文，这类报文用来查询网络信息，比如ping程序就是使用ICMP报文来查看目标是否可到达的。

（3）传输层

传输层为两台主机上的应用程序提供端到端的通信。与网络层使用的逐跳通信方式不同，传输层只关心通信的起始端和目的端，而不在乎数据包的中转过程。传输层协议有TCP协议、UDP协议组成。

【TCP协议】传输控制协议，为应用层提供可靠的、面向连接的和基于流（Stream）的服务。TCP协议使用超时重传、数据确认等方式来确保数据包被正确地发送至目的端，因此TCP服务是可靠的。使用TCP协议通信的双方必须先建立TCP连接，并在内核中为该连接维持一些必要的数据结构，比如连接的状态、读写缓冲区，以及诸多定时器等。当通信结束时，双方必须关闭连接以释放这些内核数据。TCP服务是基于流的。基于流的数据没有边界（长度）限制，它源源不断地从通信的一端流入另一端。发送端可以逐个字节地向数据流中写入数据，接收端也可以逐个字节地将它们读出。

【UDP协议】用户数据报协议，与TCP协议完全相反，它为应用层提供不可靠、无连接和基于数据报的服务。“不可靠”意味着UDP协议无法保证数据从发送端正确地传送到目的端。如果数据在中途丢失，或者目的端通过数据校验发现数据错误而将其丢弃，则UDP协议只是单地通知应用程序发送失败。因此，使用UDP协议的应用程序通常要自己处理数据确认、超时重传等逻辑。UDP协议是无连接的，即通信双方不保持一个长久的联系，因此应用程序每次发送数据都要明确指定接收端的地址（IP地址等信息）。基于数据报的服务，区别于数据流服务，每个UDP数据报都有一个长度，接收端必须以该长度为最小单位将其所有内容一次性读出。

（4）应用层

应用层负责处理应用程序的逻辑。数据链路层、网络层和传输层负责处理网络通信细节，

这部分必须既稳定又高效，因此它们都在内核空间中实现。而应用层则在用户空间实现，因为它负责处理众多逻辑，比如文件传输、名称查询和网络管理等。如果应用层也在内核中实现，则会使内核变得非常庞大。当然，也有少数服务器程序是在内核中实现的，这样代码就无须在用户空间和内核空间来回切换（主要是数据的复制），极大地提高了工作效率。不过这种代码实现起来较复杂，不够灵活，且不便于移植。

- ping是应用程序，而不是协议，前面说过它利用ICMP报文检测网络连接，是调试网络环境的必备工具。
- telnet协议是一种远程登录协议，它使我们能在本地完成远程任务。
- OSPF（Open Shortest Path First，开放最短路径优先）协议是一种动态路由更新协议，用于路由器之间的通信，以告知对方各自的路由信息。
- DNS（Domain Name Service，域名服务）协议提供机器域名到IP地址的转换。

应用层协议(或程序)可能跳过传输层直接使用网络层提供的服务，比如ping程序和OSPF协议。应用层协议（或程序）通常既可以使用TCP服务，又可以使用UDP服务，比如DNS协议。我们可以通过/etc/services文件查看所有知名的应用层协议，以及它们都能使用哪些传输层服务。五层协议背后的思想：上层屏蔽下层细节，只使用其提供的服务。高内聚低耦合，每一层专注于其功能，各层之间的关系依赖不大。数据包在每层有不同的格式，从上到下依次叫段，数据报，帧，数据从应用层通过协议栈向下传递，每经过一层加上对应层协议的报头，最后封装成帧发送到传输介质上，到达路由器或者目的主机剥掉头部，交付给上层需要者。这一过程称为封装，传输，分离，分用。

2.2 TCP与UDP

TCP与UDP都是TCP/IP协议栈的网络传输层协议，也是通信调试助手直接面向用户的最基本的网络通信协议。TCP是面向连接的协议，具有高可靠性，为应用层提供可靠的、面向连接的和基于流（STREAM）式数据的服务。而UDP与TCP协议完全相反，它为应用层提供不可靠、无连接和基于数据报文的服务。“不可靠”意味着UDP协议无法保证数据从发送端正确地传送到目的端。如果数据在中途丢失，或者目的端通过数据校验发现数据错误而将其丢弃，则UDP协议只是单地通知应用程序发送失败。

在通讯协议的选择上，TCP与UDP这现两种协议各有自己的优点及不足之处。总的来说，TCP协议面向连接，数据传输保证可靠性，但速度慢，系统开销大；UDP协采用无连接方式，自身无法保证数据传输的可靠性，需要靠上层应用来保证，但是它传输速度快、系统开销少、运行效率高。在实际工程项目中，需要根据项目本身的特点，因地制宜，选择最合适的通信协议。

TCP的通信拓扑结构是C/S模型，其通信双方的角色分别为TCP服务端和TCP客户端。并且，TCP通信双向必须先建立TCP链路连接后，才能够进行后续的TCP数据传输。这就好比打电话，要先拨打对放的电话号码，只有电话拨通（成功建立信道连接）后，才能进行后续语音数据的传输。对于TCP通信而言，TCP客户端正是作为连接发起方的角色，它需要知道TCP服务端的号码（也就是TCP服务端正在监听的IP地址和端口号），才能向TCP服务端发起通信连接请求，当然此刻的TCP服务端要处于监听状态，这样TCP服务端才能接受TCP客户端的连接接入

请求。

UDP的通信协议本身不分主从，收发UDP数据并不需要事先建立任何通信连接，只要知道对方的号码（IP地址和端口号），就能直接向对方发送数据。

我们在使用网络调试助手软件进行通信调试时，目标设备或者网络应用程序肯定是明确地采用了以下某一种通信协议：TCP Server、TCP Client、UDP，我们所要做的就是在调试助手软件中选择与实际情况一致的通信协议即可。

2.3 网络套接字

套接字（socket）是封装了TCP/UDP协议的通信编程接口。最初是加利福尼亚大学Berkeley分校为Unix系统开发的网络通信接口。后来随着TCP/IP网络的发展，Socket成为最为通用的网络应用程序接口，也是在Internet上进行应用开发最为通用的API。

传输层实现的是端到端的通信，因此，每一个传输层连接都有两个端点。从通信对象的角度来讲，所谓的套接字，实际上是一个通信端点，或者说是通信地址。根据RFC793的定义：端口号拼接到IP地址就构成了套接字，即形如（主机IP地址：端口号）。例如，如果IP地址是210.37.145.1，而通信端口号是23，那么得到套接字就是（210.37.145.1：23）。也就是说，通信连接只能建立在两个通信端点，也就是套接字（主机IP:端口）之间，而仅有IP地址或仅有端口号都无法建立通信连接。

TCP/IP的核心内容被封装在操作系统中，为了支持用户开发面向应用的通信程序，大部分系统都提供了一组基于TCP和UDP的应用程序编程接口（API），该接口通常以一组函数的形式出现，即称为套接字（Socket）。TCP与UDP套接字连接服务器与客户端的编程调用流程如下图所示。

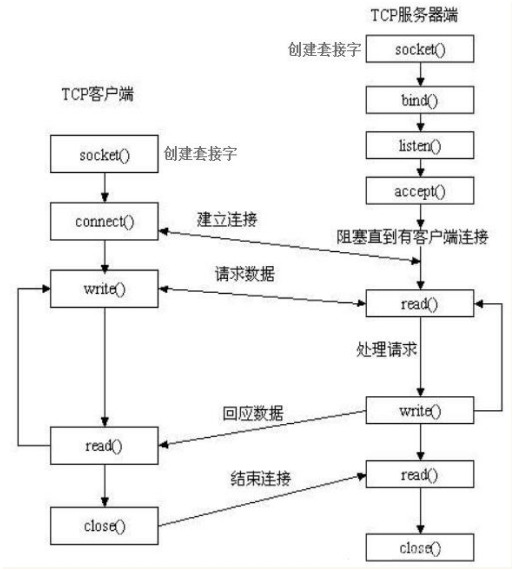


图2-2 TCP套接字编程模型

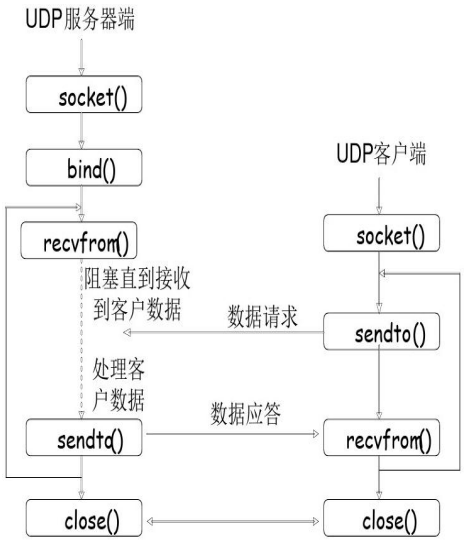


图2-3 UDP套接字编程模型

- socket()：创建套接字。
- bind()：绑定套接字的网络地址（IP:Port）。一个套接字用socket()创建后，它其实还没有与任何特定的本地或目的地址相关联。在很多情况下，应用程序并不关心它们使用的本地地址，这时就可以不用调用bind指定本地的地址，而由协议软件为它们选择一个。

但是，在某个知名端口（Well-known Port）上操作的服务器进程必须要对系统指定本地端口。所以一旦创建了一个套接字，服务器就必须使用bind()系统调用为套接字建立一个本地地址。

- connect()：将套接字连接到目的地址。初始创建的套接字并未与任何外地目的地址关联。客户机可以调用connect()为套接字绑定一个永久的目的地址，将它置于已连接状态。对数据流方式的套接字，必须在传输数据前，调用connect()构造一个与目的地的TCP连接，并在不能构造连接时返回一个差错代码。如果是数据报方式，则不是必须在传输数据前调用connect。如果调用了connect()，也并不像数据流方式那样发送请求建连的报文，而是只在本地存储目的地址，以后该socket上发送的所有数据都送往这个地址，程序员就可以免去为每一次发送数据都指定目的地址的麻烦。
- listen()：设置等待连接状态，即监听状态。对于一个服务器的程序，当申请到套接字，并调用bind()与本地地址绑定后，就应该等待某个客户机的程序来要求连接。listen()就是把一个套接字设置为这种状态的函数。
- accept()：接受连接请求。服务器进程使用系统调用socket、bind及listen创建一个套接字，将它绑定到知名的端口，并指定连接请求的队列长度。然后，服务器调用accept进入监听状态，直到到达一个连接请求。
- send()/recv()和sendto()/recvfrom()：发送和接收数据 。在数据流(TCP)通信方式中，一个连接建立以后，或者在数据报方式下，调用了connect()进行了套接字与目的地址的绑定后，就可以调用send()和reev()函数进行数据传输；而对于数据报（UDP）通信，则无需调用connect()函数建立连接，直接可以通过sendto()向任意网络地址（IP:Port）送数据，或通过recv()或recvfrom函数接收数据。recv()和recvfrom()函数都是用于接收网络数据，区别点在于recvfrom()不仅可以读取到网络数据，还可以读取到数据来源的IP地址及端口号。
- closesocket()：关闭套接字，释放占用的资源。

总之，套接字（IP地址+端口号对）是网络通讯中最基本也是最重要的概念。不管是做网络编程开发，还是使用网络调试助手进行通信调试，都要明确目标通讯双方的套接字形式的网络地址（IP:PORT）。因为，仅仅主机IP地址之间不能建立通信，仅仅主机端口之间也不能建立通信，只有套接字（主机IP:端口）之间才能建立通信。在网络调试助手的使用过程也可以看到，出现IP的地方必然会出现对应的端口号，完整的IP+PORT才能唯一确定目标主机的网络地址。

2.4 单播与广播

单播（Unicast）是一个单个发送者和一个接收者之间通过网络进行的通信。这个术语跟广播或多播相对应；广播(broadcast)：一个主机要向网上的所有主机发送数据帧；多播(multicast)：处于单播和广播之间，数据帧仅传送给属于多播组的多个主机。

区别于单播 IP 地址，广播 IP 和多播的 IP 不是某一主机的 IP 地址，它们仅仅是一个解释性的泛地址，用于表示数据包需要发送到网络上符合条件的多个主机。向这些 IP 的主机

发送包时，对方网卡适配器的 Mac 地址不用通过 ARP 请求获得，可以在本地按照约定直接生成。广播 IP 和多播 IP 对应的网卡 Mac 地址同样也不是某一真实网卡的硬件地址，而仅仅是一个解释性的泛地址。

以太网中，广播 IP 的 Mac 地址固定为 0xffffffffffff；多播 IP 的 Mac 地址的首字节是 0x01，其低位 23bit 等于 IP 多播组号的低 23 位。由于多播组号中的最高 5bit 在映射过程中被忽略，因此每个以太网多播地址对应的多播组是不唯一的。

通常网卡仅接收那些目的地址为网卡物理地址或广播地址的帧，大多数的网卡经过配置都能接收目的地址为多播地址或某些子网多播地址的帧。对于以太网，当地址中最高字节的最低位设置为 1 时表示该地址是一个多播地址，用十六进制可表示为 01:00:00:00:00:00（以太网广播地址 ff:ff:ff:ff:ff:ff 可看作是以太网多播地址的特例）。

对于我们网络开发编程及测试人员来说，并不需要关心 MAC 地址的设置，因为这些都已经被 SOCKET 套接字封装在底层了。我们需要关心的是通信对方的目标 IP 地址，因为这是可以决定单播、多播及广播的。从套接字角度来说，向单播套接字（单播 IP 地址+端口号）发送数据，就是单播通信；向广播套接字（广播 IP 地址+端口号）发送数据，就是广播通信。

显然，TCP 通信必定是单播的，需要建立虚拟的物理连接，也就是点对点的通信连接；而 UDP 通信不需要建立连接，它可以是单播/广播/多播。那么下面就来具体介绍一下广播 IP 地址。

广播地址(Broadcast Address)是专门用于同时向同一个网段的网络中所有工作站进行发送的一个地址。在使用 TCP/IP 协议的网络中，主机标识段 host ID 为全 1 的 IP 地址为广播地址，广播的分组传送给 host ID 段所涉及的所有计算机。例如，对于 10.1.1.0（255.255.255.0 ）网段，其广播地址为 10.1.1.255 （255 即为 2 进制的 11111111 ），当发出一个目的地址为 10.1.1.255 的数据报文时，它将被分发给该网段上的所有计算机。广播地址分受限广播与直接广播两种类型：

1)受限广播

受限广播地址是指 IP 地址的网络字段和主机字段全为 1 的地址，其实只有 1 个，也就是 255.255.255.255 这个地址。受限广播，只覆盖整个本地网络，不能被路由器转发到其它网络。虽然受限广播地址不会被路由发送，但会被送到相同物理网络段上的所有主机。

2)直接广播

直接广播会被路由转发（但需要路由器配置相应的选项），并会发送到目标网段上的每台主机。直接广播地址包含一个有效的网络号和一个全“1”的主机号，比如 202.163.30.255 就是一个直接广播 IP 地址，255 就是一个全“1”的主机号。

2.5 粘包与半包

粘包与半包现象发生在TCP连接中，不管是作为TCP Server还是作为TCP Client，只有是采用TCP协议都有可能发生粘包与半包问题。而发生这种现象的概率通常取决于即时网络状况的好坏以及通信双方的即时数据处理能力，这是TCP流式数据传输无法避免的问题。

举个例子来直观地描述一下粘包与半包现象。通过网络调试助手建立一个TCP Server，如下图所示，然后再启用一个网络调试助手建立一个TCP Client连接到前面的TCP Server。

通过网络调试助手的TCP Client向TCP Server高速循环发送一个字符串（这里发送的是字符串how are you），如图4-4所示。服务器端在设置接收自动换行的条件下，正常情况上应该连续接收到一行一行完整的字符串（how are you）。但实际情况是，有可能收到的一行包含多条字符串（how are you），这就是粘包现象；或者多行才能接收完一条完整的字符串（how are you），这就是分包现象。

图2-4 粘包与分包示例

粘包与半包出现的概率频次通常都是很低的，但是却是无法规避的问题。作为上层应用程序，在出现粘包与半包时，要尽量做到能否准确识别。下面从原理上分析一下粘包与半包产生的机理，以及应对措施。

粘包是指发送方发送的多个数据包到达接收方时粘成一个包的这样一种现象。粘包可能由发送方造成，也可能由接收方造成。发送方可能造成粘包的原因：由于TCP的数据发送都是流式异步发送，在网络编程过程中调用Send函数发送数据时，只是将数据放到对应连接的协议栈缓冲区中，然后等待协议栈调度发送，如果还没等到协议栈从网络接口发出数据，而用户再次调用Send发送数据时，新的数据会无缝添加到之前的数据后面粘合成一个大包一起发送；接收方引起的粘包是由于接收方用户进程没及时从协议栈缓冲区取走数据，而下一包数据已经到来并无缝粘到前一包数据的尾部，形成一个粘包，无法分割。分包是指一种操作，是指在出现粘包的时候我们的接收方要进行分包处理，粘包一般的解决办法是制定应用层的通讯协议，通过协议来规范现有接收的数据是否满足消息数据的需要。在应用中处理粘包的基础方法主要有两种分别是在数据头中引入描述消息大小的数据位或以结束符来分割，实际上也有两者相结合的如HTTP、Redis等的通讯协议等。

半包指接收方一次接收操作只能接收部分数据，需要接收多次才能收全一个完整的包。在发送大包时出现半包的几率比较大，当然发送小包也有可能出现。虽然用户发送数据是一

个包一个包的发送，但是调用Send只是放入协议栈缓冲，真正发送是由协议栈来完成。因为TCP协议只有流的概念，没有包的概念，协议栈只负责把发送缓冲区中的数据都发出去，至于一次取多少数据来发送也是有协议栈来决定。组包就是将多个半包组合成一个完整的用户数据包，这也是实际网络应用开发时，上层应用程序需要分析解决的问题。

粘包与半包现象不会出现在UDP通信中，因为UDP是基于数据报文传输的。在网络编程时，调用一次UDP的Send函数，就会报送一个UDP报文。接收方读取UDP数据时，也是以报文为单位来接收，因此不会出现粘包与半包问题。但是UDP报文的最大长度有一个限制，UDP报文的最大理论长度分析：UDP报文由IP报文封装，去掉ip包头占20字节，UDP报文最大长度是65535 - 20 = 65515字节，再去掉UDP包头占8字节，剩下UDP数据最大长度是65515 - 8 = 65507 字节。超过这个这个长度的UDP报文是无法发送的，而TCP则是流式数据不存在报文最大长度的问题。

第三章 调试助手配置选项

调试助手的配置参数及控制选项众多，涉及的内容包括系统语言、文字编码、窗口样式、数据格式、数据校验、转义字符支持、数据发送控制方式、数据接收存储及日志文件记录等等。使用调试助手进行通信调试时，结合实际应用场景，使用恰当的配置选项，可以有效地提高通信调试的工作效率，甚至达到事半功倍的效果。

3.1 语言及编码

调试助手软件支持中、英文双语，默认情况下自动根据系统语言选择切换。在中文环境下自动选择中文，其它语言环境自动选择为英文。也可以通过调试助手的[Language]菜单选项，直接指定中/文语言。

图 3-1 语言以及编码切换

调试助手的数据编码支持 ANSI(GBK)/UTF-8 两种编码。默认方式是 ANSI(GBK)编码。如果接收窗口显示乱码数据，有可能接收到的数据是 UTF-8 编码的文字，如果按照默认的 ANSI(GBK)方式显示就会乱码.解决方法是，在接收窗口中点击右键，在弹出菜单中，选择切换编码为 UTF-8，如上图 3-1 所示。注意，切换编码后，并不会刷新显示已接收的乱码数据，只有后来新接收的数据才能按新的编码设置来显示。

发送数据时，也涉及到文字编码方式的选择。发送窗口的编码方式与接收窗口的编码方式是相互独立的。同接收窗口的编码设置方式一样，在发送窗口点击右键，在弹出的右键菜单，可以选择发送数据的编码方式为 ANSI(GBK)或 UTF-8。

对于英文字母或字符，不需要区分是 ANSI(GBK)编码还是 UTF-8 编码，因为这两种编码是相同的，都是一个字节的 ASCII 码；但是对于汉字等多字节编码的文字或符号则是不一样的。比如，1 个汉字的 ANSI(GBK)编码占 2 个字节，而 1 个汉字的 UTF-8 编码则占 3 个字节。

调试助手能够按照所设置的编码方式对收发数据进行相应的编码及显示处理。比如，在发送窗口输入汉字“你好”，然后点击【发送】按钮时，如果按 ANSI(GBK)编码发送，则实际发送的 16 进制数据是“C4 E3 BA C3”，而按 UTF-8 编码发送时，实际发送的 16 进制数据是“E4 BD A0 E5 A5 BD”。调试助手会按照当前设定的编码发送正确的数据。

3.2 窗口主题样式

为了确保软件界面更符合个人视觉习惯，调试助手提供了界面主题定制功能。界面主题定制的内容，包括字体样式、大小、颜色及背景。并且，日志提示符、接收的数据、发送的数据，其字体颜色可以分别独立设置。如图 3-2 所示，通过菜单选项，调出用户界面主题设置窗口。可以分别对接收窗口以及发送窗口的界面主题进行定制，或者选择使用预设的主题样式。

图 3-2 用户界面主题设置

调试助手提供二个预置的主题样式：白底经典（默认主题）和黑底经典。也可以选择自定义方式，并按个人的喜好习惯设置界面背景以及字体。下图是选择黑底经典主题的效果。左侧控制面板可以折叠收起，显示效果会更加简洁。

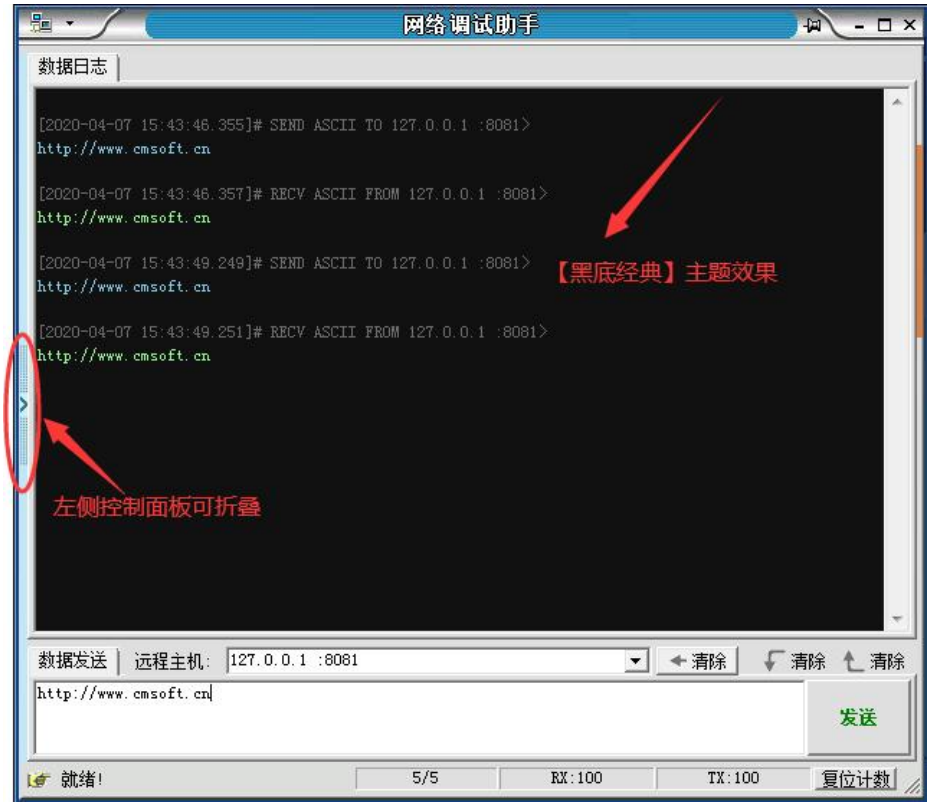


图 3-3 收/发窗口分别是“黑底经典”及“白底经典”主题模式

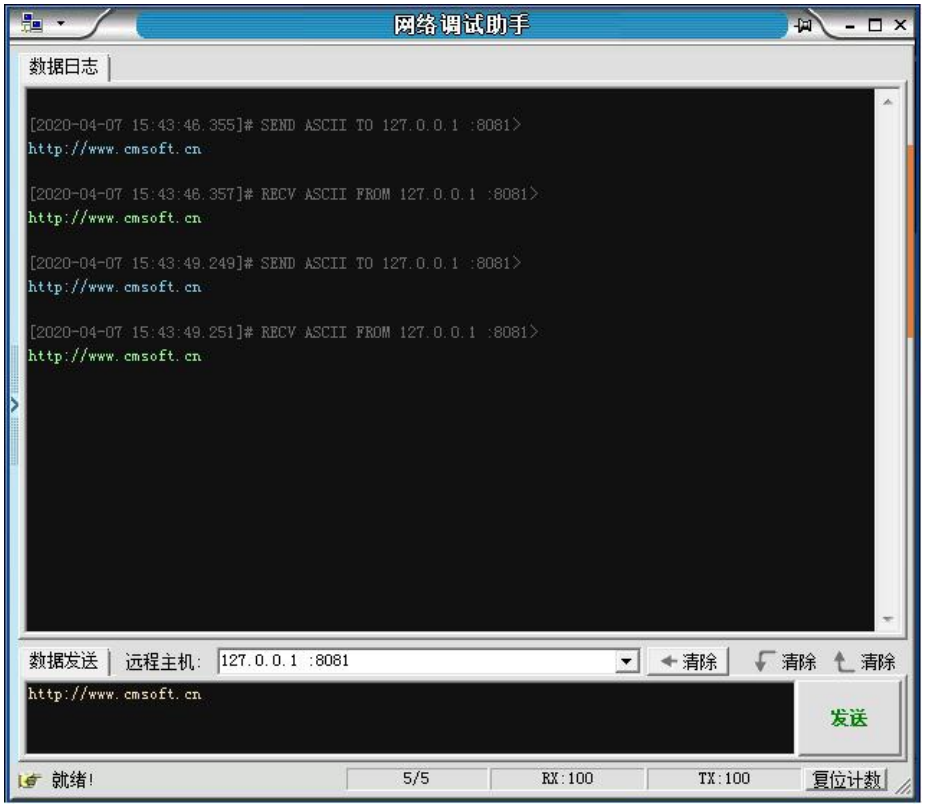


图 3-4 收/发窗口分别是“黑底经典”

3.3 数据接收设置选项

3.3.1 数据接收格式

在调试助手左侧的接收参数面板，可以设置接收数据的显示格式为 ASCII 码或者 HEX 码，方便用户按不同的方式分析查看其所接收的数据。如果接收到的是可打印字符(文字)，可以直接按 ASCII 模式查看，或者按 HEX 模式查看其 16 进制编码；但如果接收到的数据包含非打印字符，那么按 ASCII 模式查看，接收窗口就有可能出现乱码数据，不能有效反映出实接收到的数据内容。这种情形下，可以选择 HEX 模式，就可以如实有效地打印出实际接收的数据内容。

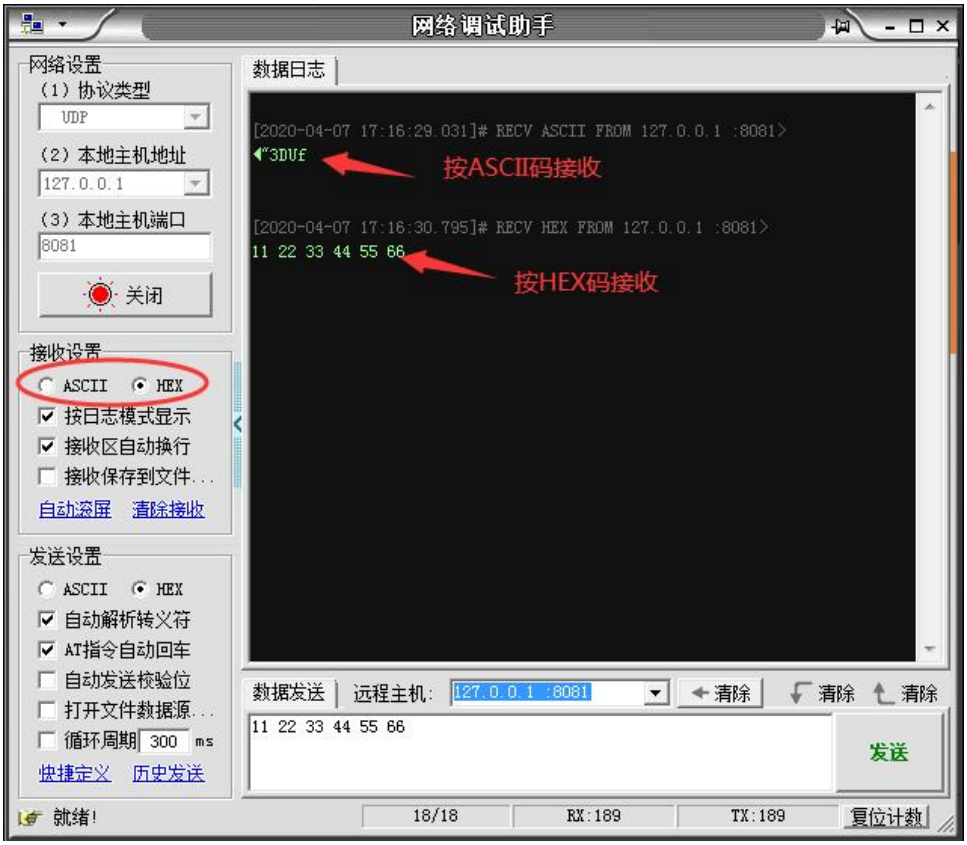


图 3-5 数据接收格式控制

如上图所示，分别按 ASCII 码与 HEX 码接收一段包含非打印字符的数据。在按 ASCII 码接收时包含乱码，而按 HEX 码接收时，就是很好的分辨实际接收的内容。使用调试助手时，要根据实际的场景需求来切换数据显示格式。

3.3.2 日志显示格式

数据接收窗口默认是按日志模式显示的，除了显示接收到的数据内容外，还会显示接收数据的时间戳、数据格式(ASCII 码/HEX 码)、数据来源 IP 地址及端口号。另外，接收窗口还会显示所发送的数据记录信息。如下图所示：

图 3-6 按日志模式显示数据

如果在接收设置选项中，取消【按日志模式显示】选项。那么接收窗口就仅显示接收到的数据内容，而不会显示接收记录的时间戳等附加信息，并且发送数据记录也不会显示。

3.3.3 接收自动换行

【接收区自动换行】选项，决定每接收到一条新的数据记录是否会自动换行，还是在之前接收到的数据末尾追加。这个选项对于日志模式无明显效果，因为日志模式下每一条接收记录都会强制换行。

自动换行涉及到一个概念问题，就是如何区分一条数据记录的起始。换句说，在接收连续的数据流时，如何分割数据记录。对于 UDP 通信，这个比较简单，因为 UDP 是基于数据报文的协议，UDP 协议本身就能够确保每一条 UDP 报文的独立性与完整性，因此 UDP 通信以报文为单位进行换行；对于 TCP 通信就比较特殊，因为 TCP 是数据流传输协议，只能保证发出的 IP 数据包准确有序地到达接收端，并不会给应用层提供报文分割的起始标识。即使 TCP 发送端是一次性打包发出去的数据，到达接收端时的数据也会像流水一样陆陆续续地到达，尤其是网络状况差且发送的数据包比较大的情况下，到达接收端时数据流的不连贯性会更加明显，而且更具不确定性。对于网络调试助手，处理 TCP 数据流时比较简单粗暴，超过 50ms 的间隙就自动截断，作为一条数据记录来对待。

3.3.4 接收报存到文件

调试助手接收的数据可以自动保存到文件。在接收设置中，点击【接收保存到文件】选

项，会弹出下图 3-7 所示对话框，选择接收保存的文件路径。

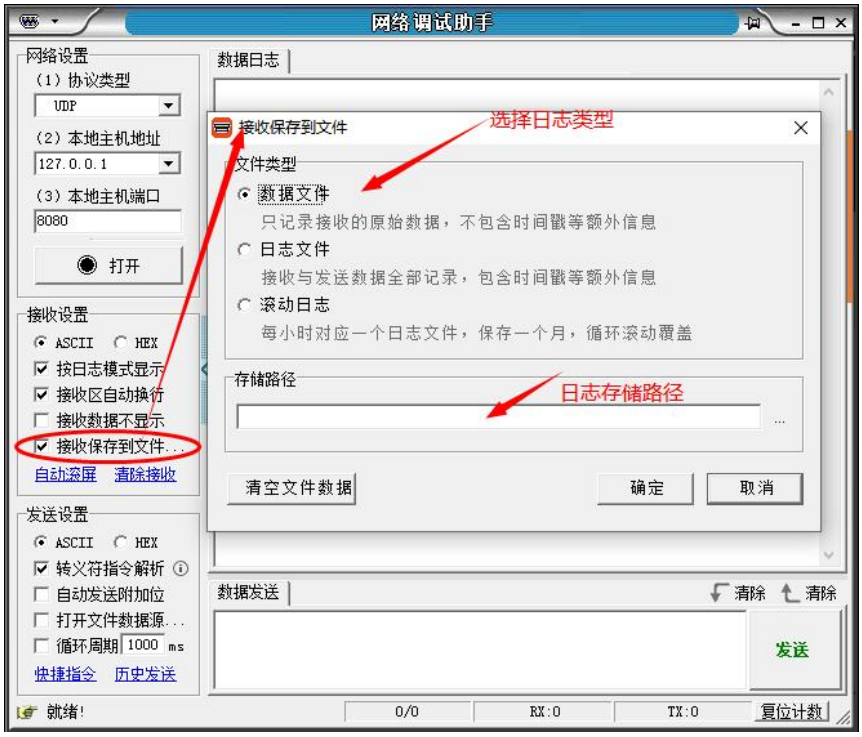


图 3-7 接收保存到文件

如图所示，选择目标文件存储路径时，要注意设置文件保存类型，可选项有数据文件、日志文件、滚动日志三种。所谓数据文件，就是目标文件只保存接收到原始数据内容，不包含诸如时间戳等其它附加信息，并且数据文件只包含接收数据，而不包含发送的数据记录；对于日志文件，不仅存储接收的数据内容，还存储接收数据的时间戳及数据来源等相关信息，同时还包含调试助手的发送记录信息；滚动日志则是日志文件的扩展，其每小时生成一个独立的日志文件，保存一个月，循环滚动覆盖。在实际应用中，如果纯粹是用于保存接收到的原始数据，那就选择数据文件类型；如果要记录收发双发交互的数据及发生的时间，那就选择日志文件或滚动日志类型，其中滚动日志专用于长时间的日志记录。

3.3.5 接收自动滚屏

网络调试助手接收到的数据都会显示在接收窗口，新收到的数据会自动追加到之前接收到的数据末尾，并且接收窗口会实时刷新，如果接收到的数据内容超过窗口高度时就会自动逐行滚屏，保证最新接收到的数据始终在接收窗口可见。这样就会出现一个问题，如果高速接收显示连续的数据流时，接收窗口就会不断滚屏翻页，导致用户无法有效地查看数据，因为还没看清楚就已被自动翻页。

接收设置中的【自动滚屏】选项，可以切换自动滚屏功能。软件启动后的默认设置是启用自动滚屏，单击该选项便可切换/关闭自动滚屏。一旦关闭自动滚屏，虽然新收到的数据还是会自动追加到接收窗口的数据末尾，但是接收窗口不会自动滚屏，配合鼠标控制接收窗口的滚动条，用户就可以自由查看接收窗口中的数据，不会再被自动滚屏功能所干扰。

3.4 数据发送设置选项

数据发送设置选项包括数据发送模式（ASCII/HEX）、转义字符支持、AT 指令自动回车、自动发送校验位、文件数据源设置、循环发送设置等。

3.4.1 数据发送类型

调试助手可发送的数据类型有 ASCII 文本字符串和 HEX 十六进制编码数据两种。对应在软件界面左侧的发送参数面板，可以选择数据发送编码类型：ASCII 码或 HEX 码。任何数据都可以编码成十六进制形式文本进行发送，但不是任何数据都可以按 ASCII 码字符串发送。如果发送的数据包含非打印字符，那么转换为 ASCII 码就会发生乱码，发送时就会丢数据。因此，包含非打印字符的数据只能按十六进制编码进行发送；而不包含非打印字符的数据该如何选择数据发送类型，就要根据具体应用场景怎么方便就怎么选择。

3.4.2 转义字符支持

网络调试助手支持在发送的 ASCII 文本中插入转义字符。只要勾选发送设置中的【自动解析转义符】选项，发送包含转义符的 ASCII 文本时，转义符会自动解析成对应的 ASCII 码数据进行发送，方便用户以文本形式发送非打印符。转义符以反斜杠开头。比如，回车符的转义字符是\r 或\x0d、换行符的转义符是\n 或\x0a，等等。任何 ASCII 码字符（包括可打印字符及不可打印字符）都可以通过\x 后紧跟两位十六进制编码数据来表示。

图 3-8 发送带转义符的 ASCII 码文本

上图 3-8 是一个发送文本中包含转义符的例子：在发送框中输入 hello\r\n，然后点

击发送按钮，表示发送字符串 hello 及回车换行。注意，只有在 ASCII 码发送模式下，并勾选了【自动解析转义符】选项后才支持转义字符。

3.4.3 AT 指令自动回车

为方便 AT 指令的发送，只要勾选【AT 指令自动回车】选项，那么在发送以 AT 开头文本指令时，调试助手软件会自动检查是否以回车换行符结尾，如果没有显式的回车换行结尾符，那么发送的 AT 指令自动会在尾部补齐回车换行符。这样，用户发送 AT 指令时，就不必每次都添加回车换行符，从而可有效提高通信调试的工作效率。

3.4.4 自动发送校验位

在使用调试助手进行通信调试时，某些场景下需要发送带校验位的指令数据。比如，调试 modbus 通信协议时，指令末尾需要加 CRC16 校验位。这就要求在准备调试指令前事先计算好校验码，如果遇到指令比较多且随时要修改的情形，就会比较麻烦。针对这个问题，调试助手提供了自动发送校验位功能选项。只要勾选该选项，并选择对应的校验算法，那么发送数据时就会自动在数据末尾发送所选择的校验位数据。

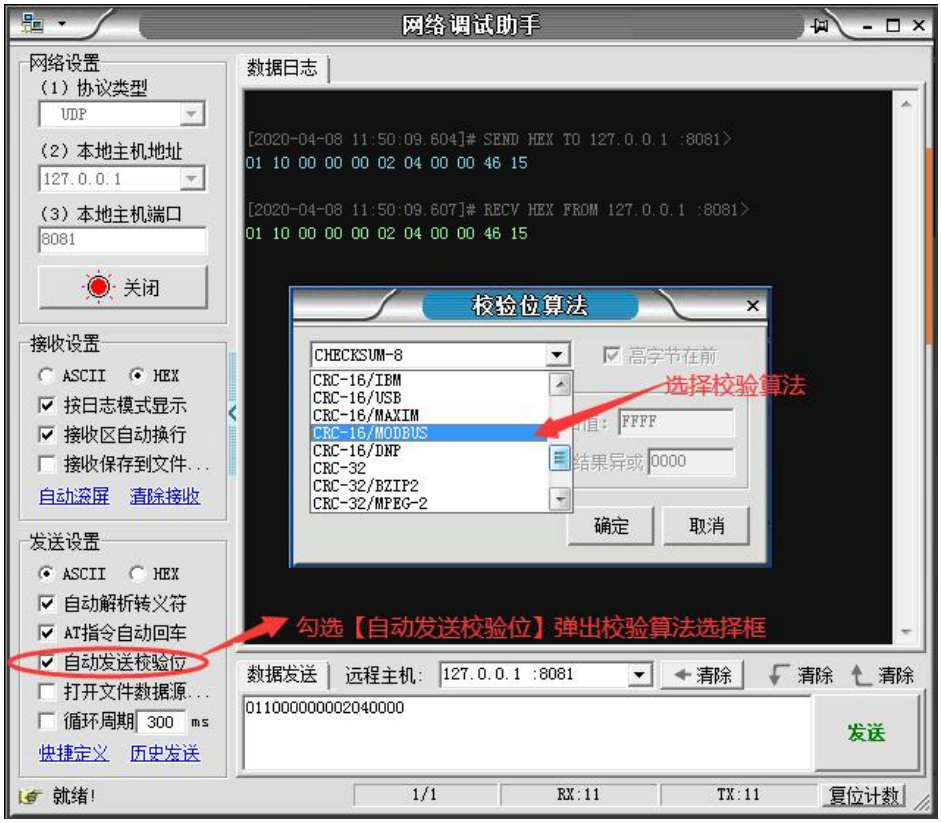


图 3-9 自动发送校验位设置

如图所示的例子。校验位选择了 CRC-16/MODBUS，然后发送 16 进制数据指令：011000000002040000，而实际发出去的指令末尾会自动增加 2 个字节的 CRC16 校验位数据，免去了用户自行计算添加校验位的麻烦。

3.4.5 发送文件数据

调试助手发送数据时，数据输入方式有两种，常规方式就是通过发送输入框输入数据；而在数据量比较大的情况下，可以将待发送的数据保存为文件，然后通过文件数据源发送数据。具体操作方法：在调试助手左侧控制面板的发送设置中，点击【打开文件数据源】；接着在弹出的文件选择对话框中选择待发送的目标文件，即载入文件数据源；最后，点击【发送】按钮，便开始从文件数据源发送数据。

3.4.6 循环发送设置

在发送设置面板中，设置好循环发送的时间周期，单位毫秒，并勾选【循环发送】选项，然后点击【发送】按钮，调试助手就会按照所设置的时间周期，循环发送输入的数据源。一旦启动循环发送过程，【发送】按钮便自动切换成【停止】按钮，要终止循环发送操作只要点击【停止】按钮即可。

3.4.7 回车键发送设置

调试助手的数据发送是通过鼠标点击【发送】按钮来触发的，也可以通过键盘按键的方式进行发送。默认情况下，单独按 Enter 键或者组合键 CTRL+Enter 都可以执行数据发送操作，如果要在发送输入框中输入回车换行符，直接按回车键只能触发数据发送而无法输入回车换行，必须通过组合键 Shift+Enter 来插入回车换行符。

图 3-10 发送快捷键设置

如果要改变回车键的默认设置，可以右键点击【发送】按钮，在弹出的右键菜单中进行勾选设置，如上图 3-10 所示，具体选项说明如下。

1) [按 Ctrl+Enter 键发送]：该选项勾选后，组合键 Ctrl+Enter 执行发送操作；否则该组合键不执行任何操作；
2) [按 Enter 键发送/Shift Enter 换行]：该选项勾选后，单独按 Enter 键（不要组合 Shift 或 Ctrl）时执行发送操作，而组合键 Shift+Enter 用于在发送输入框中输入回车换行；如果取消勾选该选项，则单独按回车键时就会输入回车换行符而不会触发发送操作，并且组合键 Shift+Enter 无效。

3.5 发送框默认内容设置

图 3-11 发送框默认内容设置

调试助手软件启动后，发送输入框中的默认显示内容可以进行定制，可以是固定的数据内容（或为空），也可以记住上一次关闭时最后一条发送的数据。具体设置方法：在调试助手的菜单中找到【发送框中默认内容】选项，点击弹出设置窗口，如图 3-11 所示。可以选择【记住上一次成功发送的内容】或者【固定内容】。对于固定内容，可以选择 ASCII 码文本或者 HEX 码十六进制数据。修改设置后，点击确定按钮，下次启动调试助手软件时生效。

3.6 其它参数及控制选项

3.6.1 新建会话

在通信调试过程中，根据实际应用场景，可能随时需要运行多个调试助手软件。为了避免用户在电脑中查找/执行软件的麻烦,调试助手软件提供了【新建会话】功能，该控制选项位于系统菜单的第一个菜单项，用于启动另外一个调试助手的软件实例，也就是再运行一个调试助手软件。每点击一次【新建会话】选项，就会启动一个调试助手窗口进程，只要 PC 的内存足够大，可以同时启动任意多个调试助手窗口。

3.6.2 自动记住窗口大小

启用【自动记住窗口大小】菜单选项后，调试助手软件在每次关闭时都会自动保存记住当前的窗口大小，再次启动软件时，会自动恢复到上次关闭时软件窗口的大小。

3.6.3 窗口固定在最上层

【窗口固定在最上层】菜单选项，其实就是窗口置顶功能，可避免网络调试助手被其它软件窗口遮挡。该菜单选项还有一个快捷方式入口在调试助手主窗口标题栏目右侧，是一

个图钉样式的图标按钮，点击直接切换窗口置顶模式。

3.6.4 始终显示任务栏图标

调试助手软件启动后会在操作系统的任务栏条以及任务栏的托盘区显示软件图标。当调试助手最小化时，任务栏条上的图标在默认选项下会自动隐藏，只在任务栏的托盘区显示图标。如果用户的操作系统托盘区设置了折叠模式，那么调试助手的图标就有可能被隐藏起来，不容以直观地找到。这种情况下，就希望调试助手最小化时，不隐藏任务栏条上的软件图标。【始终显示任务栏图标】菜单项即用于实现该功能，该选项勾选后，操作系统的任务栏条上会始终显示调试助手的软件图标，方便用户定位工作窗口。

3.6.5 关闭按钮默认操作

点击调试助手主界面的关闭按钮，是执行软件关闭还是窗口最小化，可以通过该选项来设置，如下图所示。

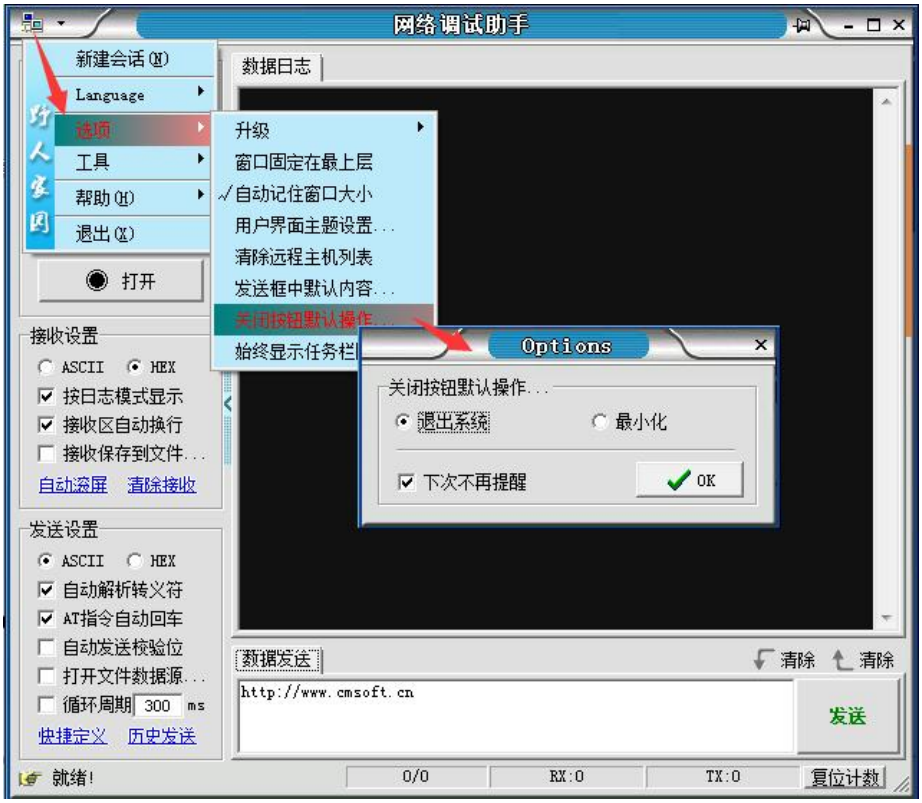


图 3-12 关闭按钮默认操作设置

第四章 调试助手基本操作

本章介绍网络调试助手基本的通信操作方法，包括TCP Server通信、TCP Client通信、UDP通信。

4.1 TCP通信测试

TCP（Transmission Control Protocol）是面向连接的、可靠的、基于字节流的传输层通信协议，TCP 通信前必须建立连接，也就是建立客户端与服务器之间的 TCP 链路连接,之后才能在客户端与服务端之间进行数据收发通信，并且 TCP 服务器要时刻监听本地服务端口，随时准备接受新的 TCP 客户端的连接请求，这些行为都可以通过网络调试助手进行仿真模拟。TCP 通信需要重点关注两个参数，第一个参数是服务器所监听的网络接口（适配器）的 IP 地址，第二个参数是服务器所监听的 Port 端口号。

下面通过具体的测试案例来说明，如何通过网络调试助手进行 TCP 协议的通信调试。本次实验中，服务器端与客户端各自运行一个网络调试助手软件，分别用于模拟 TCP 服务端和 TCP 客户端。本次测试环境配置参数如下所示。

- 服务端：云服务器，Windows Server 操作系统，外网 IP 地址为 47.96.255.174，私有(内网)IP 为 192.168.3.38，TCP 服务监听端口 8080；
- 客户端：个人 PC，Windows 操作系统，局域网宽带接入 Internet。

4.1.1 服务器端协议选择

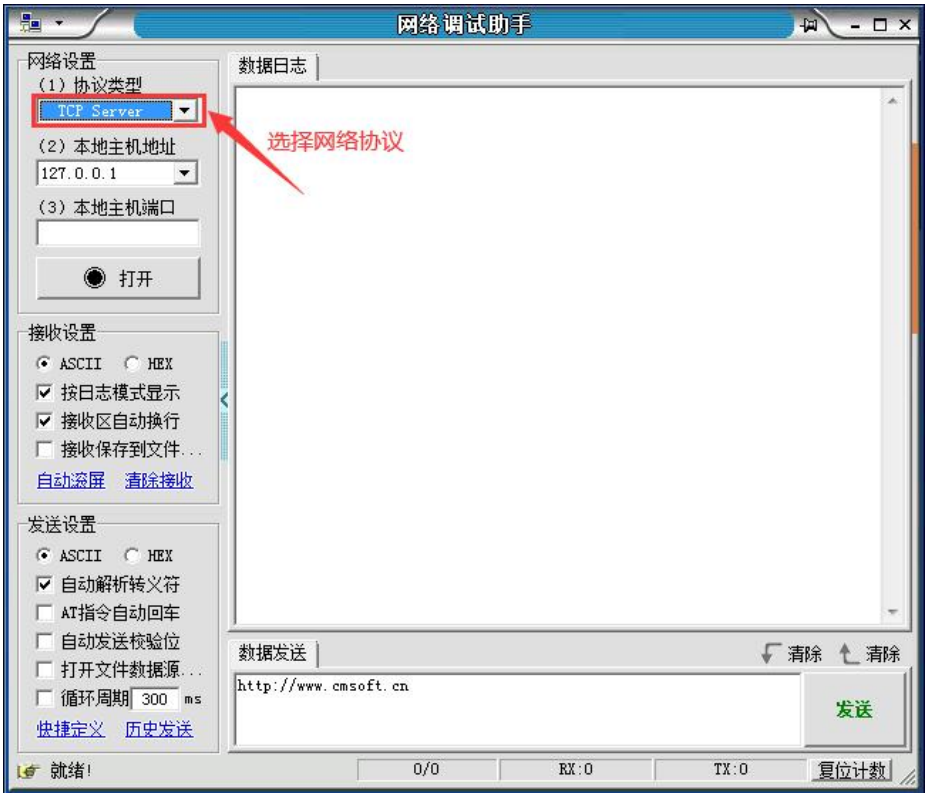


图 4-1 服务端选择网络协议

在服务器上运行网络调试助手软件，如图 4-1 所示，将网络设置中的通信协议选项设为 TCP Server。

4.1.2 服务器端监听地址选择

服务器对外提供 TCP 服务时，必须绑定对外通信服务的网络适配器，也就是设定如下图 4-2 所示的本地主机地址。本地主机地址下拉选择框中，自动枚举列出了当前主机（服务器）所有实际存在的网络适配器所对应的内网 IP 地址。由于本地主机地址(网络适配器的内网地址)不止一个，需要根据实际情况进行选择。

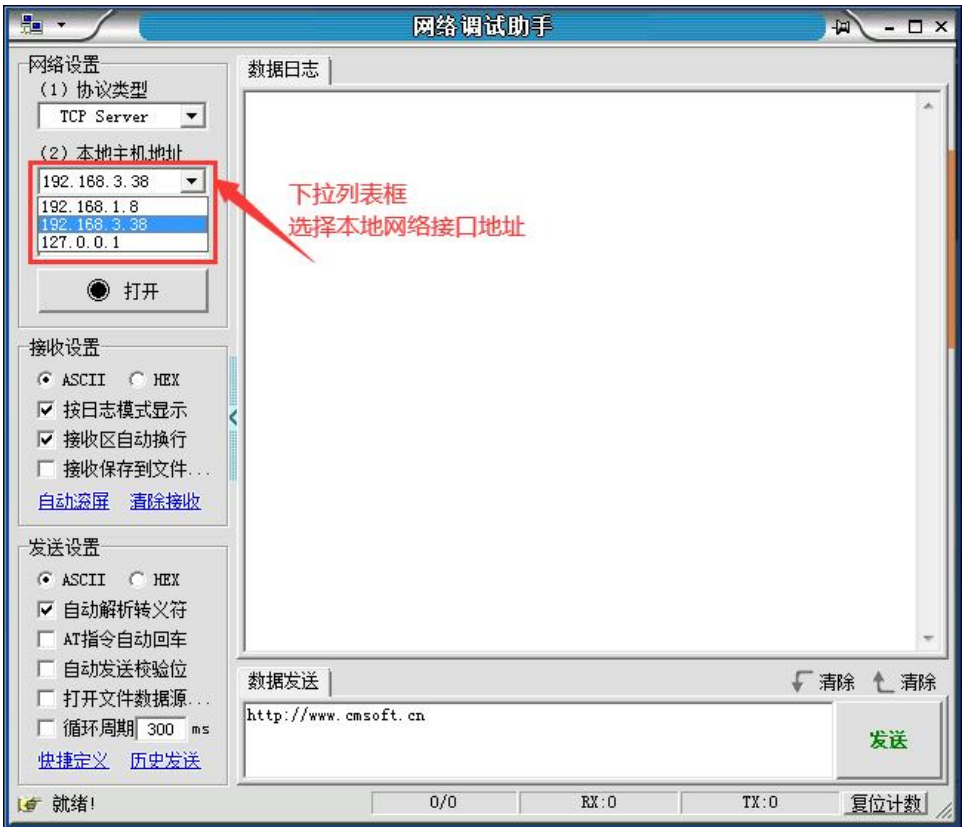


图 4-2 服务器端主机地址选择

如图 4-2 所示，本次测试的服务器存在 3 个网络适配器，对应的内网 IP 地址分别为：192.168.1.8、192.168.3.38、127.0.0.1。此前已有说明，本次测试服务器通过外网提供 TCP 服务，其外网 IP 地址为 47.96.255.174，该外网地址绑定的网络适配器的内网 IP 为 192.168.3.38，所以我们在本测试案例中选择本地地址为 192.168.3.38。这样，我们在跟 47.96.255.174 这个外网 IP 地址通信时，服务器运营商的路由器会自动映射到其内网主机地址 192.168.3.38。

4.1.3 服务器端口选择

本测试案例中，服务器端口号选择 8080。如果选择其他的端口号，在建立客户端时要保持一致。

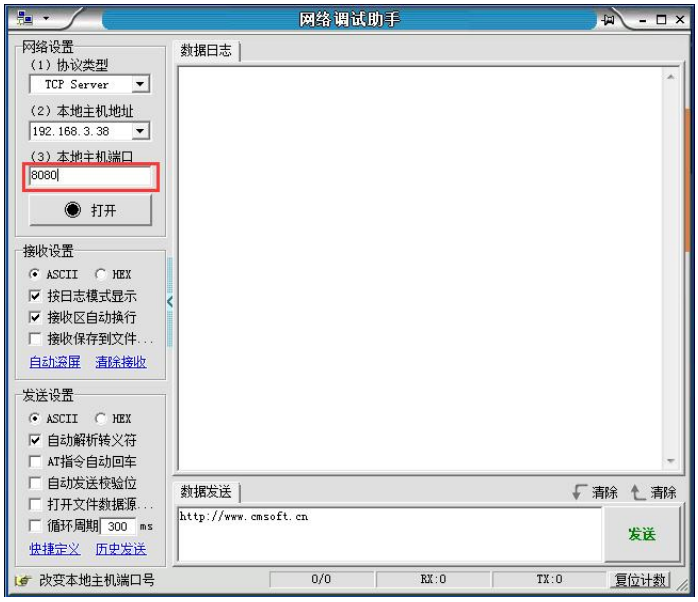


图 4-3 服务器端口选择

这里的本地端口号可以填多个,以实现多端口监听，而不必启动多个调试助手。多个端口号之间用英文逗号分割，比如填 3 个端口号：8081,8082,8083；或者通过波浪号输入一个端口号区间，比如 8081~8083；或者两种方式的组合，比如 8080,8081~8083。注意，多端口监听要求软件版本号大于 V5.0.11，并且多端口监听功能同样适用于 UDP 协议模式，下文不再赘述。

4.1.4 服务器打开监听

服务器端的网络调试助手，在设置好协议类型(TCP Server)、本地主机地址(192.168.3.38)、本地主机端口(8080)三个参数后，接着点击【打开】按钮，网络调试助手便会在 192.168.3.38 这个网路适配器的 8080 端口上启动 TCP 监听服务。

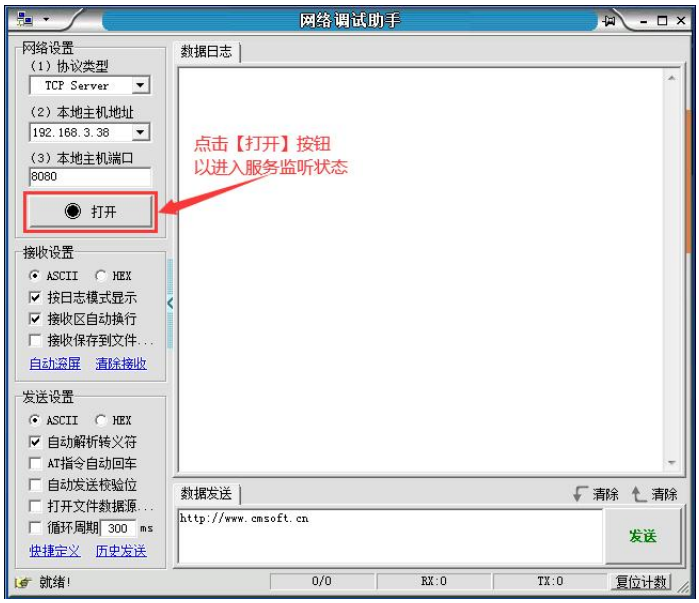


图 4-4 服务器打开 TCP 监听服务

如下图 4-5 所示，服务器端网络调试助手成功打开监听服务后，【打开】按钮自动切换为【关闭】按钮。如果再次点击此按钮将关闭 TCP 服务器的监听服务，所有连接到服务器的客户端会随服务器的 TCP 服务的断开而全部自动断开。

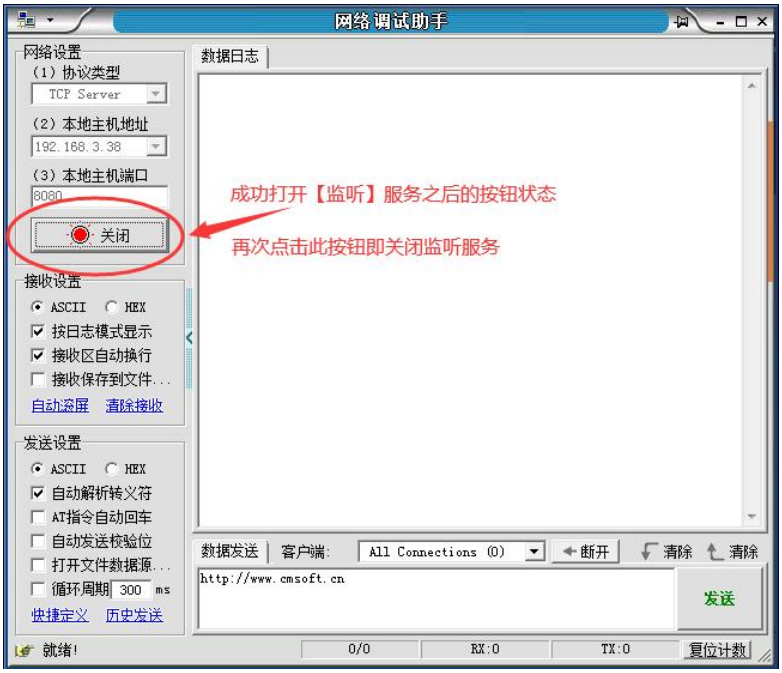


图 4-5 服务端成功打开 TCP 监听后的按钮状态切换

4.1.5 服务端的接入管理

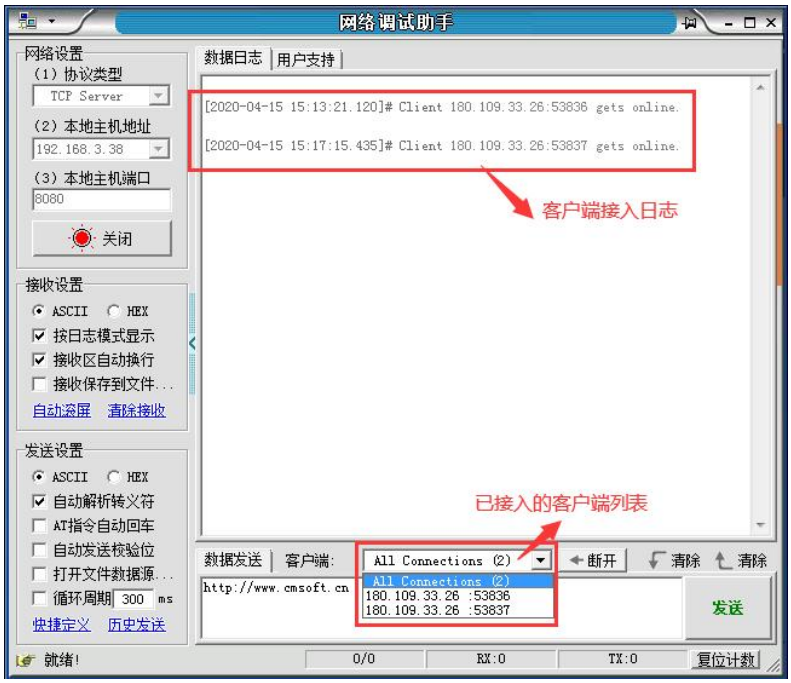


图 4-6 服务端的接入管理

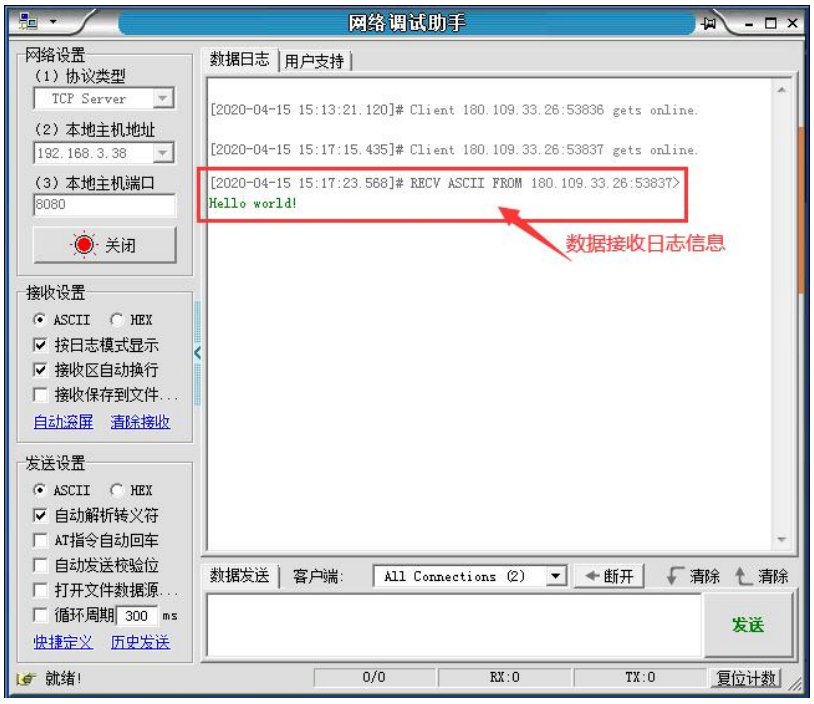
如图 4-6 所示，在服务器端的网络调试助手界面，一旦有新的客户端接入，都会在客户端下拉列表中显示出来。如果有客户端主动断开连接，也会自动从这个列表中清除。

客户端自主断开有两种情形，一种是正常断开，也就是客户端点击【断开】按钮，或者正常关闭客户端软件，或者正常关机；另一种是异常断开，比如客户端突然断电或者宕机导致的异常断开。如果是客户端正常断开的情形，服务器会立即收到客户端的断开事件并更新客户端列表；但如果客户端是异常断开的，那么服务器端可能会收不到客户端的断开事件，那么客户端列表中就会出现僵尸连接。

应对僵尸连接，网络调试助手会定期自动清理，如果检测到连续 12 小时没有收到某一个客户端发送来的数据，可以说明该客户端连接已经失效，就会强制将其踢除。用户也可以直接在网络调试助手的客户端列表中选择希望断开的客户地址，然后点击右侧的【断开】按钮，手动将其强制踢除。

4.1.6 服务端接收客户端数据

当服务器端收到任何一个客户端发来的数据时，会在接收窗口显示接收到的数据记录。如果在接收设置中勾选了【按日志模式显示】，那么接收到的数据记录不仅显示接收到的数据内容，还会显示接收时间、客户端 IP 地址及端口号等相关信息。



4-7 服务端接收客户端数据

4.1.7 服务端向客户端发送数据

服务器向客户端发送数据时，第一步要做的是选择目标客户端，也就是向哪个客户端发送数据,对应的操作就是在发送区的客户端列表中选择目标客户端地址。如果目标客户端选择 All Connections 选项，就代表向当前所有已经接入客户端（All Connections）发送数据；接下来第二步，就是在发送框中输入待发送的内容；最后，点击【发送】按钮，输入框中的数据就会被发送往当前所选的目标客户端。如果接收设置中勾选了按日志模式显示选项，那么这时在主窗口就会显示此次发送的日志记录信息。

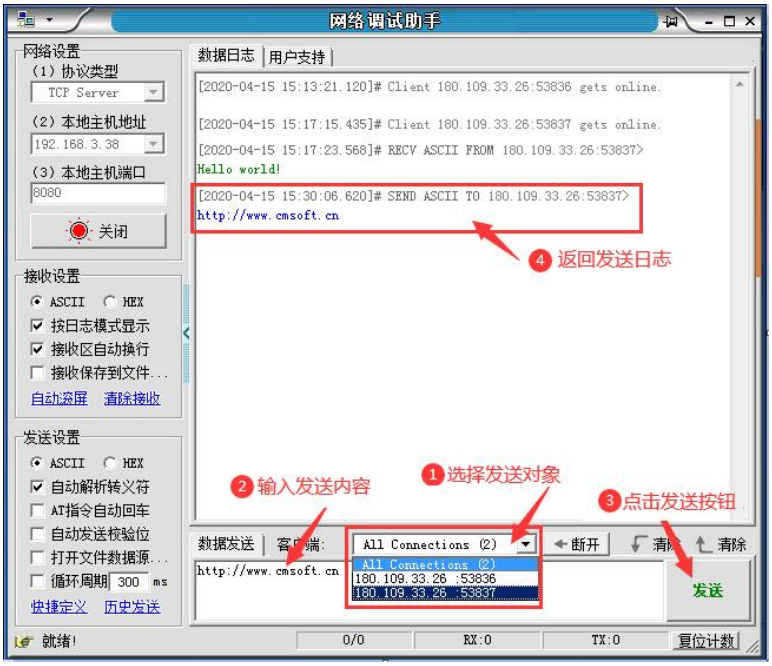


图 4-8 服务端向客户端发送数据

4.1.8 客户端-协议选择

在客户端 PC 上运行网络调试助手软件，将网络设置中的通信协议选项设为 TCP Client。

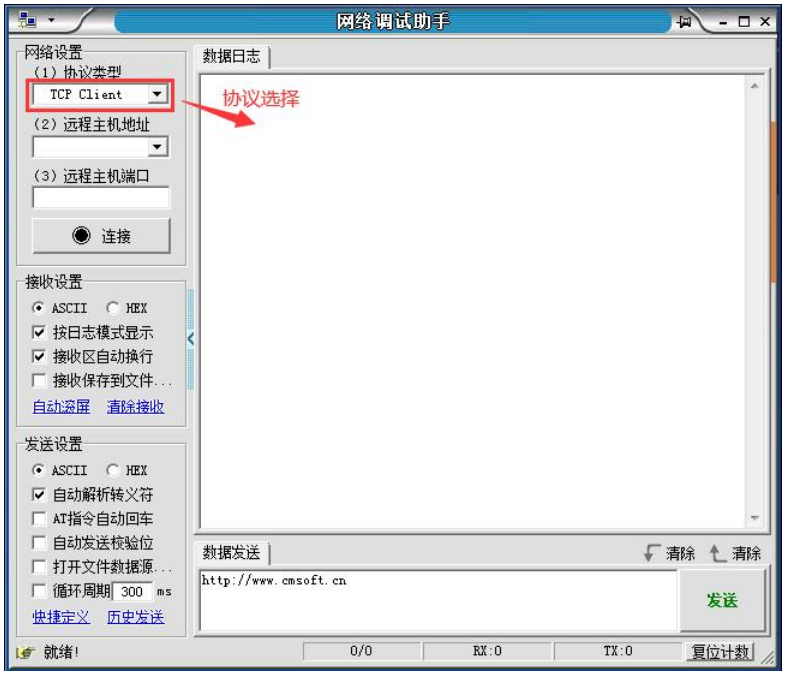


图 4-9 客户端选择网络协议

4.1.9 客户端-远程主机地址设置

在客户端网络调试助手中，设置远程主机地址，也就是即将连接的目标服务器的外网 IP 地址。本测试中，云服务器的外网 IP 为 47.96.255.174，如下图 4-10 所示填写。当然，这里的远程主机地址输入框也支持域名形式的地址，前提是该服务器的 IP 地址已有对应的

域名绑定。

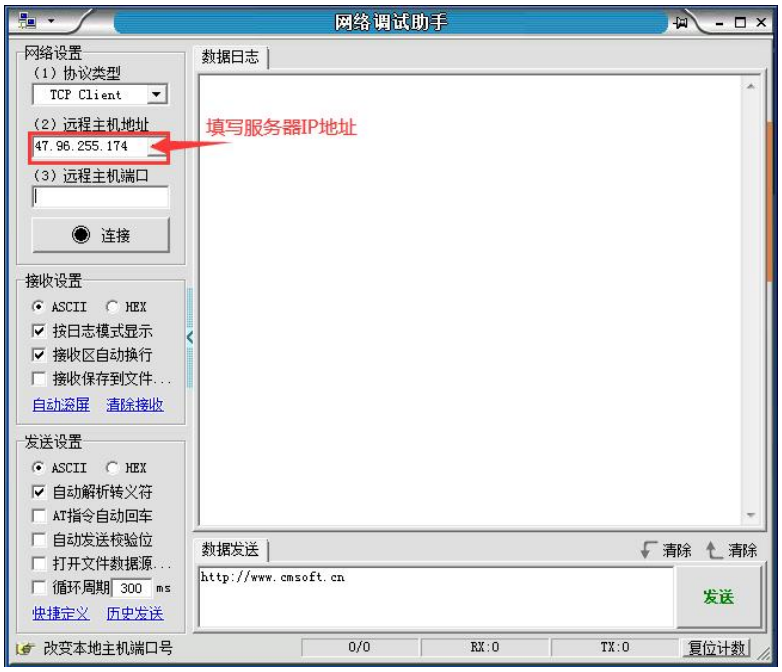


图 4-10 客户端输入目标服务器地址

4.1.10 客户端-远程主机端口设置

客户端网络调试助手设置远程主机端口，也就是将连接的目标服务器的服务监听端口。

图 4-11 客户端输入目标服务器端口

本次试验中，服务端的监听端口是 8080，客户端要如实填写，如图 4-11 所示。

4.1.11 客户端连接到服务器

客户端网络调试助手，在设置好协议类型(TCP Client)、远程主机地址(47.96.255.174)、远程主机端口(8080)三个参数之后，点击【连接】按钮，若连接成功，网络调试助手便会建立本地客户端到目标服务器的 TCP 通信连接，同时【连接】按钮自动切换为【断开】按钮，如下图 4-12 所示，再次点击此按钮将断开本地客户端与 TCP 服务器的通信连接。

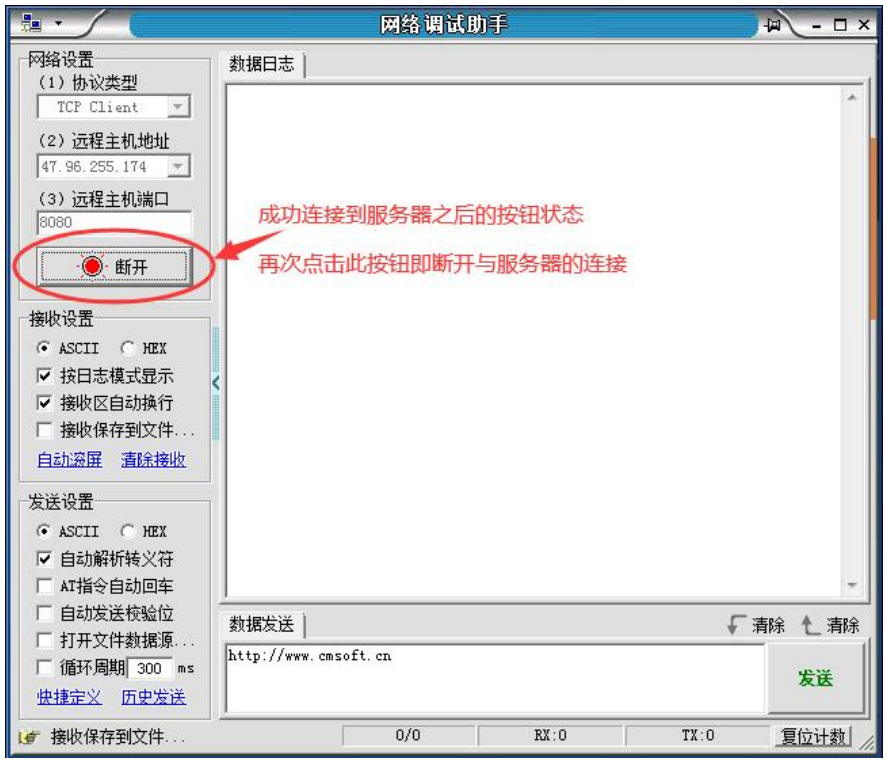


图 4-12 客户端成功连接到上服务器之后的按钮状态切换

4.1.12 客户端发送数据到服务器

客户端成功连接服务器后，便可以进行 TCP 通信调试。本次测试向服务器发送一个字符串“Hello world!”。具体操作步骤：首先在发送设置中，选择 ASCII 发送模式；然后在发送输入框中输入文本“Hello world!”，最后点击【发送】按钮。数据便会立即被发送至目标服务器。如果在接收设置面板中，勾选了【按日志模式显示】选项，那么在调试助手的主窗口区会显示此次发送的日志记录信息（包括发送的时间、发送的数据类型及数据内容）。同时，在调试助手的状态栏，会即时更新发送统计信息(数据发送的帧数及发送的总字节数)，并且发送的数据也会自动保存到历史发送记录中，随时可以通过工具面板调出历史记录来查看或重发。

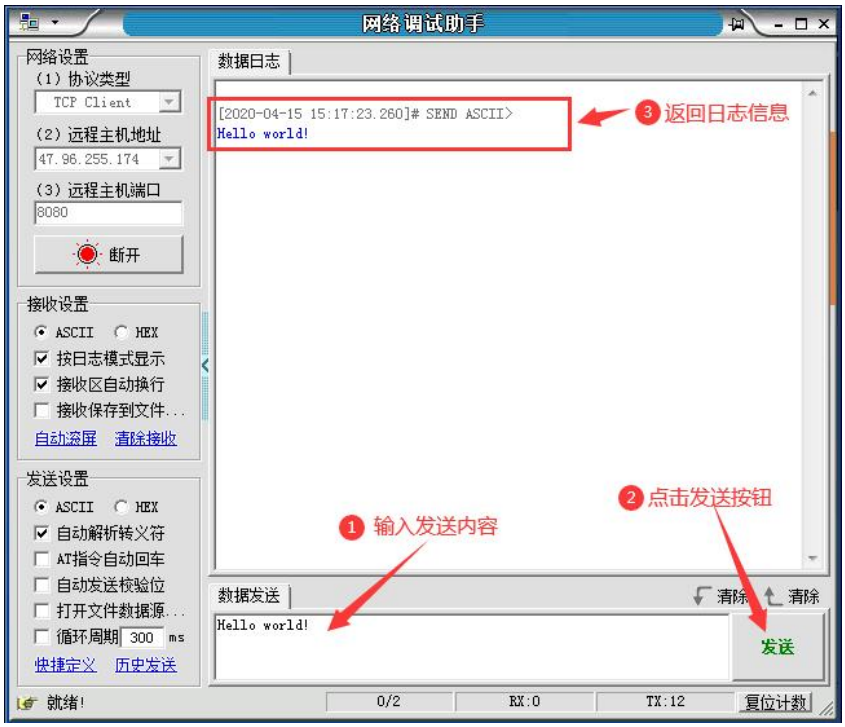


图 4-13 客户端向服务器发送数据

4.1.13 客户端接收服务器发来的数据

当客户端接收到服务器发来数据时，会立即在主窗口的接收区显示出来，如图 4-14 所示。如果在接收设置中勾选了按日志模式显示，那么在显示接收数据时，还会同时显示接收时间、以及数据类型等相关信息。

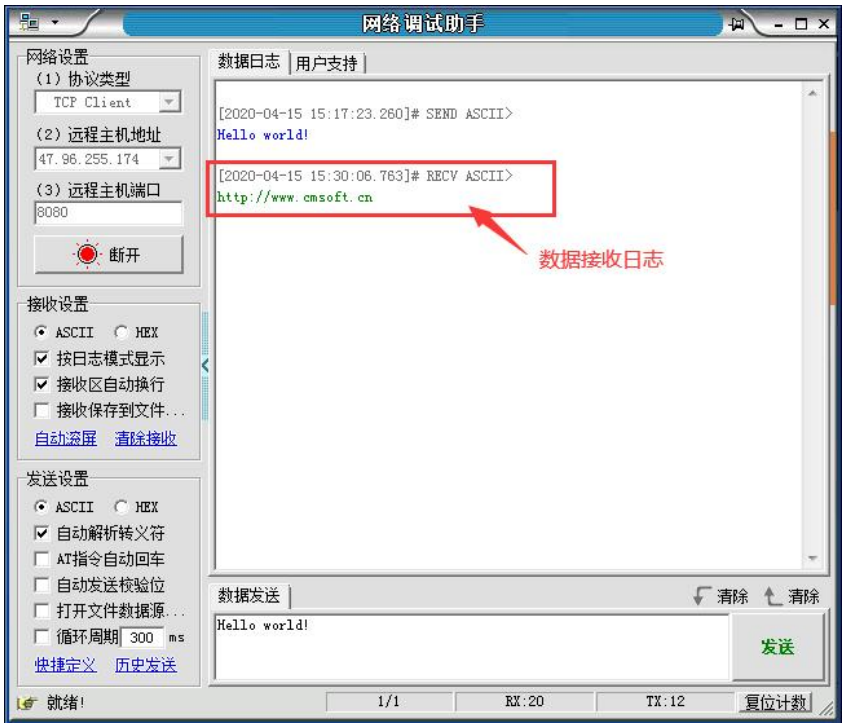


图 4-14 客户端接收来自服务端的数据

4.2 UDP通信测试

UDP(User Datagram Protocol)即用户数据报协议,跟 TCP 协议一样都是在 OSI 模型中位于传输层的协议。UDP 的缺点是不提供数据包的分组、组装和不能对数据包进行排序的，当报文发送之后，无法得知其是否安全完整到达目的地址。但是，即使在今天 UDP 仍然不失为一项非常实用和可行的网络传输层协议。许多应用只支持 UDP，如：多媒体数据流（音频和多媒体应用），强调数据传输的及时性而不是传输的完整性时，即使知道有破坏的包也不进行重发。

下面通过具体的测试实例来描述如何通过网络调试助手进行 UDP 通信。本次测试实验在同一个局域网内的 3 台 PC 上进行，每台 PC 各自运行一个网络调试助手，并选择 UDP 协议，相互间进行 UDP 数据收发测试（单播与广播）。由于 UDP 协议是无连接协议，不用区分服务器端还是客户端。这里我们将实验中的通信客户端命名为客户端 A、客户端 B、客户端 C，具体网络参数如下所示。

- 客户端 A：内网 IP 为 192.168.1.11，UDP 监听端口 8080
- 客户端 B：内网 IP 为 192.168.1.12，UDP 监听端口 8080
- 客户端 C：内网 IP 为 192.168.1.13，UDP 监听端口 8080

4.2.1 UDP 客户端的网络设置

在客户端 A（192.168.1.11）的 PC 上启动网络调试助手，网络设置参数如下图 4-15 所示：协议类型选择 UDP、本地主机地址选择 192.168.1.11，本地主机端口选择 8080。

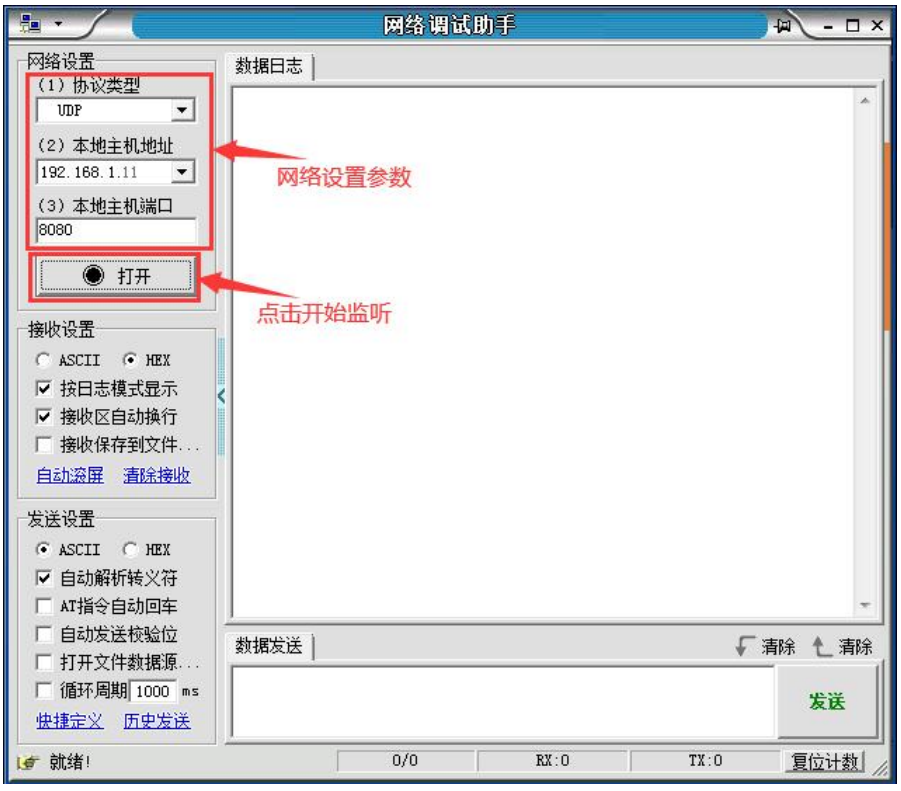


图 4-15 UDP 客户端 A 设置参数

设置完上述参数后，点击【打开】按钮。如果操作成功，便进入监听状态，如图 4-16

所示。

图 4-16 UDP 客户端 A 监听状态

类似地，在客户端 B（192.168.1.12）及客户端 C（192.168.1.13）所在 PC 上分别启动网络调试助手，并分别监听 8080 端口，如图 4-17、4-18 所示。

图 4-17 UDP 客户端 B 打开监听

图 4-18 UDP 客户端 C 打开监听

4.2.3 点对点发送数据

客户端打开监听后，便可以进行通信了。点对点通信即一对一通信，区别于一对多（多播或广播）通信。下面通过从客户端 A 向客户端 B 发送数据的方式测试点对点通信。

图 4-19 设置目标地址为客户端 B

具体实验测试如图 4-19 所示。首先在客户端 A 设置远程地址为客户端 B 的监听地址，即 192.168.1.12:8080。接着，如图 4-20 所示，在发送框输入发送的内容，并点击【发送】按钮，数据就会发送到目标主机 192.168.1.12:8080，即客户端 B。如图 4-21 所示，客户端 B 成功收到客户端 A 发来的数据。

图 4-20 客户端 A 向客户端 B 发送数据

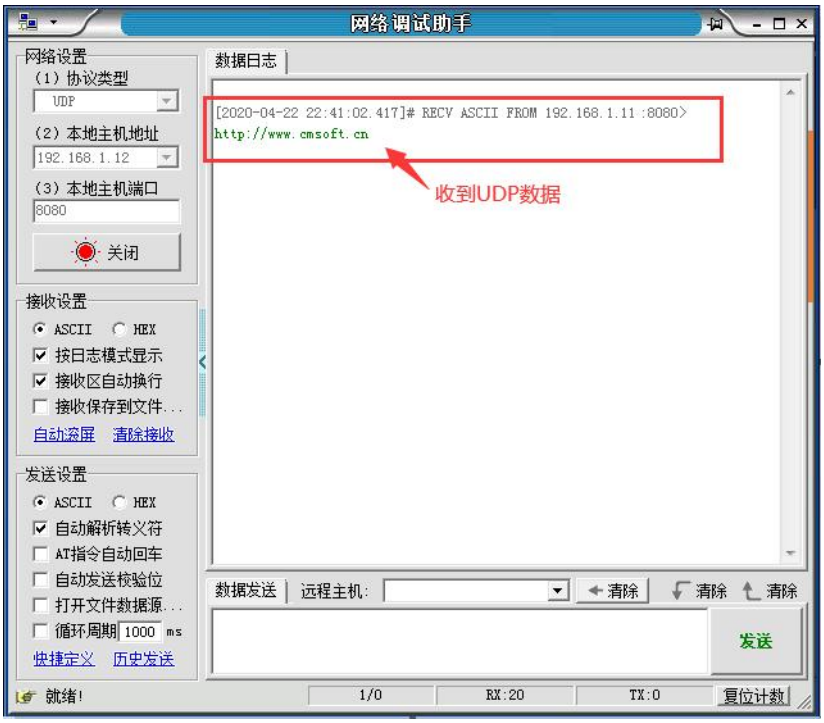


图 4-21 客户端 B 接收到客户端 A 发送的数据

4.2.4 一对多广播通信

广播通信是一种一对多的通信方式。在 UDP 协议下，可以通过向广播地址发送数据，实现广播通信。前面已经介绍过，广播地址(Broadcast Address)是专门用于同时向网络中所有工作站进行发送的一个地址，当发出一个目的地址为广播地址的数据报文时，它将被分发给该网段上的所有计算机。例如，对于 192.168.1.0 （255.255.255.0 ）网段，其广播地址为 192.168.1.255；或者使用受限的广播地址 255.255.255.255，可覆盖整个本地网络，但不能被路由器转发到其它网络。

图 4-22 客户端 A 发送广播数据

下面演示由客户端 A（192.168.1.11）发送广播数据，如图 4-22 所示。首先，设置客户端 A 监听地址为 192.168.1.11:8080，然后设置远程主机（目标地址）为广播地址 192.168.1.255:8080，接着在发送框输入发送内容，最后点击【发送】按钮，发出广播数据。如果广播成功，同一网段内所有主机（包括客户端 A、客户端 B、客户端 C）都能收到广播数据。

4.3 发送转义字符

以 ASCII 码字符串方式发送数据时，允许用户在字符串中使用转义字符的方式插入非打印字符。最简单的例子，发送一条带回车换行结尾的 AT 指令，只要在发送框输入 AT\r\n，然后直接点击发送按钮即可。

目前，调试助手支持如下这些 C 语言标准转义字符：

转义字符	含义
\r	回车符，等价十六进制 \x0D
\n	换行符，等价十六进制 \x0A
\t	水平制表符(HT)，等价十六进制 \x09

\v	垂直制表符(VT)，等价十六进制 \x0B
\a	响铃(BEL)，等价十六进制 \x07
\b	退格符(BS)，等价十六进制 \x08
\f	换页符(FF)，等价十六进制 \x0C
\xhh	2 位十六进制转义字符
\ddd	3 位八进制转义字符

作为扩展，调试助手支持转义十六进制数组，例如：`\xA0\x12\xF1\xAB\xCD\x51\xF3` 等价于 `\x[A0 12 F1 AB CD 51 F3]` 。包含在中括号[]中的多个 16 进制字节之间可以使用若干空格符分割或没有空格。

注意：使用转义符时，必须勾选主界面左侧发送设置中的【自动解析转义符】选项，否则调试助手不会对转义符进行任何解析处理。

4.4 发送指令脚本

通过转义符扩展，调试助手在 V5.0.2 版本之后开始支持发送指令脚本，允许用户在发送的指令数据中，加入各种业务处理逻辑，嵌入包含函数以及计算表达式的脚本代码，动态计算生成最终用于发送的数据内容。出于调试目的，用户还可以在发送的指令中调用 printf 函数进行调试打印，调试输出结果会显示在日志窗口。

下面这个例子，通过调试助手发送一条 Modbus 指令。在调试助手的发送窗口输入以下内容：`\x[01 04 00 00 00 04]\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]`

这条指令表示发送一组长度为 6 字节的十六进制数据 01 04 00 00 00 04，后面跟 2 字节的 CRC16 校验码，这个校验码通过以下代码动态计算获得：

`\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]`

其中，\[]称为模式符，用作嵌入脚本代码的容器。上述表达式通过冒号分割成两部分，冒号前的 2 表示最终计算值只取 2 字节，冒号后的表达式用于计算校验码。表达式中的 calculate 是系统内置函数，用于计算校验算法，calculate 函数的第 1 个参数表示从当前发送数据的第几个字节开始计算校验码；第 2 个参数表示校验数据长度，此长度可以负数，比如为-1 时，表示数据长度截止到当前 calculate 函数调用位置的前一个字节数据，-2 则表示数据长度从当前位置往前推 2 个字节，以此类推；calculate 函数的第 3 个参数表示使用的算法，ALGO_CRC16_MODBUS 是系统内置常数，表示 MODBUS_CRC16 校验算法。函数 reverse 用于将目标数据的字节顺序进行逆转（高低字节交换重排）。这里调用 reverse 的目的是因为 calculate 函数计算出的 16 位 CRC 校验码是网络字节顺序（BigEndian），但是 ModbusRTU 协议中的 CRC 校验码要求使用 LittleEndian 字序，所以这里要进行字节顺序反转处理。

图 4-3 发送包含函数表达式的指令脚本

发送脚本代码时，必须勾选发送设置中的【ASCII】模式以及【转义字符指令解析】这两个选项。这是因为，只有选择【ASCII】才允许输入脚本代码，否则若选择 HEX 模式，那么就只能输入十六进制数字；并且必须勾选【转义字符指令解析】选项后，调试助手才会对通过反斜杠导入的脚本表达式进行解析。

指令中嵌入脚本代码必须使用模式符\[]。具体的嵌入方式有两种：运算表达式和 BLOCK 代码块。

4.4.1 运算表达式

这里的运算表达式，特指具有返回值的基于类 C 语言语法规则的计算表达式。其一般形式为： `\[n:expression#remark]`

这是一种三段式表示法：第 1 段是输出数据长度，第 2 段是计算表达式，第 3 段是注解（注释）文字。其中第 1 段和第 3 段都可以省略，最简形式为 `\[expression]`。

如果设置的输出长度(n)大于实际计算表达式(expression)的最终计算值的长度，则用 0 补足后输出，反之若设置的输出长度小于最终计算结果的长度，则截掉高位多余字节后输出。如果省略第 1 段的长度值，则按表达式计算结果的固有长度输出。注解字段为可选字段，以#号开头，表示注释性文字，也可以作为注解名被引用。

例如,上一小节示例中发送的内容中包含的单行表达式字段如下：

`\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]`

如果进一步省略输出长度，可简化为

`\[reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]`

虽然调用了多个函数，多个参数，但是只有仅仅 1 条语句（1 个表达式），并有 1 个返回值。

4.4.2 BLOCK 代码段

BLOCK 块级代码字段的形式为 `\[{ script }]`

块级代码可以包含任意多条语句，最终的计算结果通过 echo 或 echob 函数产成，或者通过 return 语句返回。例如，上例中的单表达式字段，可以改写成以下的块级代码段：

```
\[{
short crc16=calculate(0,-1,ALGO_CRC16_MODBUS);
crc16=reverse(crc16);
return crc16; //或者 echo(crc16);
}]
```

第五章 调试助手进阶选项

调试助手的工具面板窗口提供通信调试的进阶操作选项。包括快捷指令、批量发送、自动应答、历史发送、校验计算器等。

5.1 快捷指令

调试助手软件在跟目标设备或串口应用服务进行通信调试时，通常需要根据实际应用协议，发送一些交互指令或数据。当指令比较多时，现场输入这些指令数据肯定会麻烦且低效。针对这个问题，本调试助手软件提供了指令预定义功能，也就是工具面板中的【快捷指令】(老版本中称之为“快捷定义”)。该功能支持按十六进制或 ASCII 码字符串的方式进行指令预定义，可以定义指令的名称以及对应的数据内容，并可定义发送热键（快捷键）。预定义的指令可以直接导出保存为文件，也可以随时从外部文件导入。默认情况下，预定义指令会自动以配置文件的形式保存在当前应用程序目录下，在软件启动时会自动载入。如下图所示。

图 5-1 快捷指令/预定义指令列表

如图 5-1 所示，每条预定义指令都包含名称、数据、快捷键三个部分。指令名称相当于指令备注，对指令数据起到描述作用；每条指令都可以单独设置快捷键，通过键盘按下快捷键可以直接发送对应的指令数据。每条指令，不管是否定义快捷键，都可以直接通过点击快捷键栏中的按钮，或者通过回车键发送选中的指令数据。

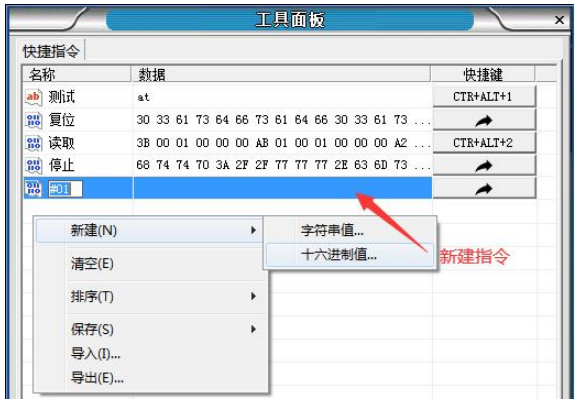


图 5-2 新建预定义指令

通过右键菜单可以对预定义指令进行增、删、改、排序、导入、导出等操作。以新建预定义指令为例，右键点击【快捷指令】窗口的空白处，弹出右键菜单，选择【新建】菜单项，就可以新建预定义指令。新建指令时，根据实际应用场景，选择字符串值还是十六进制值，然后在快捷指令窗口完成指令名称及数据内容的输入。

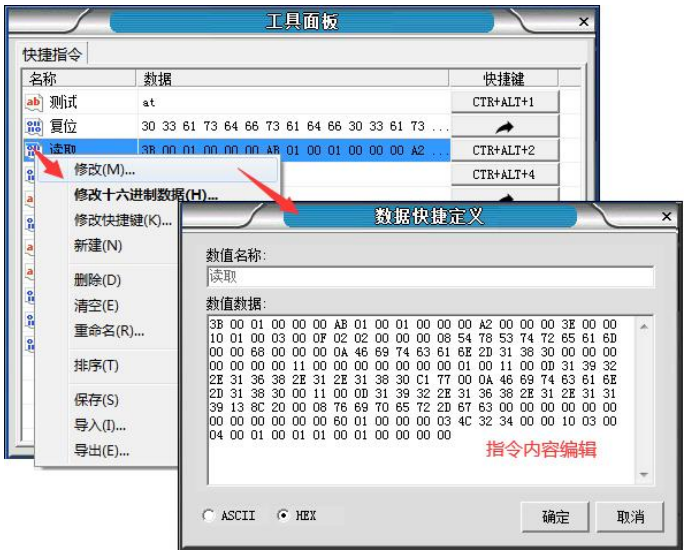


图 5-3 编辑预定义指令

如果要编辑已经输入的预定义指令，可以直接右键点击目标指令，然后在弹出的右键菜单中选择对目标指令进行编辑，包括修改数据内容、数据类型（ASCII/HEX）、设置快捷键、重命名、删除等操作。例如上图 5-3，是对一条指令进行数据内容的编辑，编辑完点击【确定】按钮关闭编辑窗口，编辑修改过的内容会在软件退出时自动保存。

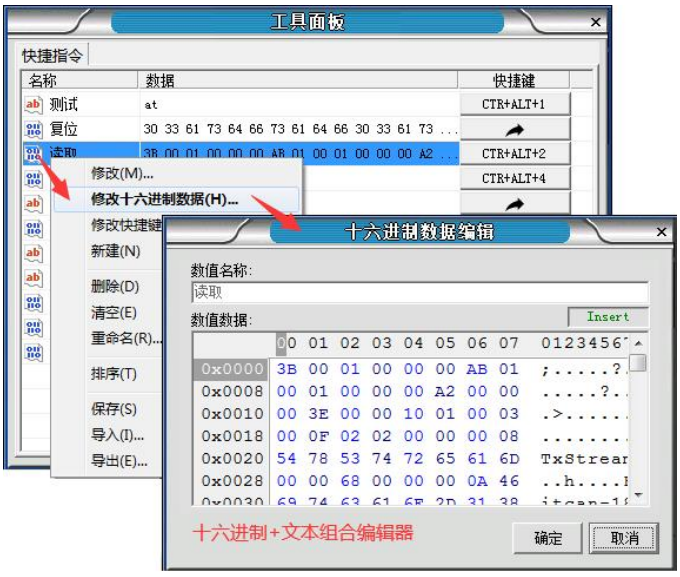


图 5-4 使用十六进制专用编辑器修改指令

图5-4所示，选中目标指令后，可以通过右键菜单选择【修改十六进制数据】，使用十六进制专用编辑器，以十六进制与ASCII码相互对照的方式对数据指令进行编辑。

5.2 批量发送

批量发送功能用于控制多条预定义数据指令按照一定的顺序和不同的延迟间隔进行发送。【批量发送】功能位于工具面板窗口，类似【快捷指令】功能，需要先进行预定义指令的编辑输入。每条指令除了可以定义指令数据和文字备注，还定义有延迟时间，即上一条指令发送完后，延迟多长时间发送当前这条指令。

图 5-5 批量发送`指令列表

如图 5-5 所示，批量发送窗口中，每条预定义指令包含编号、延迟时间、数据、及备注四个部分。指令编号顺序决定了批量发送的指令顺序，并且只有前面复选框被勾选的指令才会被批量发送；延迟是指发送前延迟，也就是当前指令与前一条发送的指令之间的发送延迟时间，单位毫秒；指令备注实际上类似于【快捷指令】中的指令名称，用于对指令起注解作用，点击备注栏中的按钮可以直接发送对应的指令数据。

勾选批量发送窗口右下角的【开始发送】选项，调试助手就会依次发送指令列表中勾选的指令，在发送每条指令前都会执行设置的延迟等待时间。发送完所有的指令，自动停止发送，并自动取消【开始发送】复选框。如果勾选了【循环模式】选项，批量发送会周而复始的进行，直至手动取消勾选【循环模式】或者【开始发送】选项。

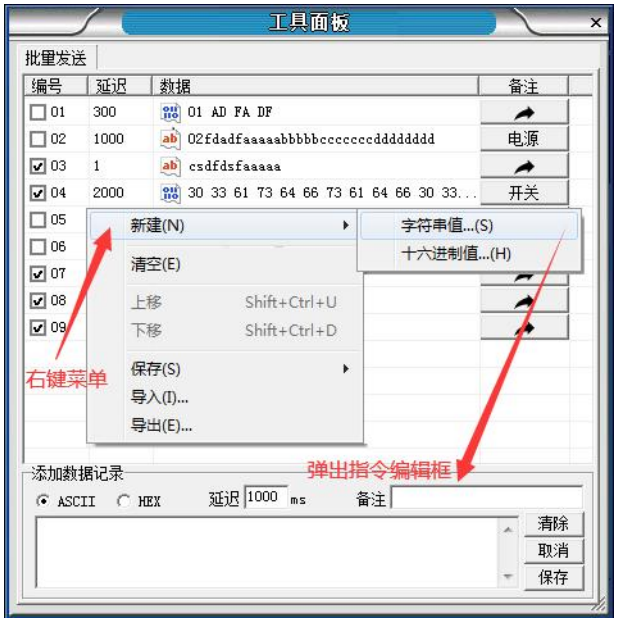


图 5-6 新建批量发送指令

通过右键菜单，可以对批量发送的指令进行增、删、改、排序、导入、导出等操作。以新建批量指令为例，右键点击【批量发送】窗口的空白处，弹出右键菜单，选择【新建】菜单项，再根据实际应用场景，选择字符串值还是十六进制值，最后在弹出的编辑窗口完成指令内容的输入，如图 5-6 所示。编辑完内容后，点击保存按钮，如果要再次编辑这条指令，可以直接在指令列表中双击目标指令，或者右键点击目标指令，然后在弹出的右键菜单中点击【修改】菜单项，即会弹出目标指令的编辑窗口。

5.3 自动应答

自动应答功能，用于实时对调试助手接收到的数据指令进行匹配识别，并自动按用户预定义规则发送相应的应答指令。用户只须事先设计好应答规则，然后调试助手内部集成的规则引擎会自动对接收到的数据进行指令规则匹配及应答数据的发送。如果需要同时支持多条指令的自动应答，只要相应地建立多条自动应答规则。

自动应答是在 V5.0.1 版本之后新增加的特色功能，自动应答的具体规则将在本文第七章中详细介绍。

5.4 数据波形

数据波形（示波器）功能，用于从接收到的指令帧（数据包）中，提取关键字段信息，比如 ADC 电压、温度、湿度等实时监测的状态数据，并显示到数据图表（折线图/点阵图/柱状图）中，以实现直观的实时数据显示分析。

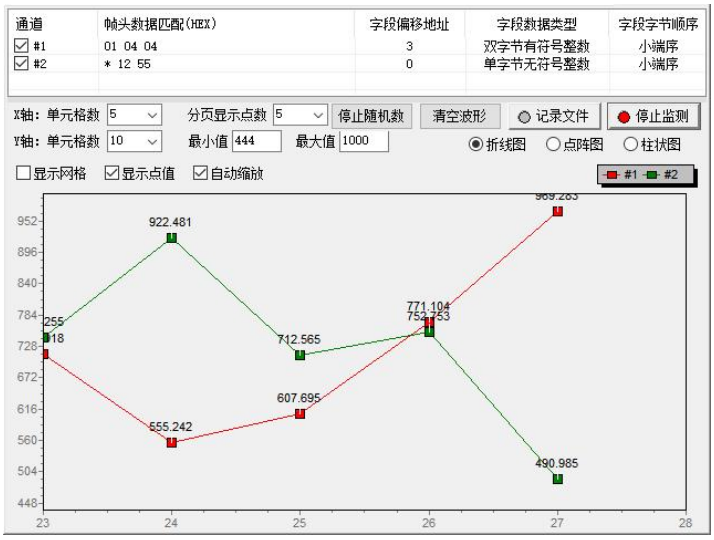


图 5-7 数据波形示例

5.4.1 建立通道

在使用数据波形功能前，须先建立相应的监测通道。一个通道对应一个监测数据对象(信号量)。如果需要监测多个数据对象，就要建立多个通道。建立通道的方法，是在通道列表窗口点击鼠标右键，选择“添加通道”命令项，弹出如下“添加通道”的编辑界面。

图 5-8 数据波形之添加通道

如上图 5-8 所示，对添加通道所涉及的各项输入参数的具体作用及规则说明如下：

【通道名称备注】 这项参数可以随便填，将作为该通道的备注名称显示在通道列表中；

【帧头数据匹配】 这里填写接收指令帧的匹配模板，用于匹配（过滤）所接收到的指令帧数据，可以输入多个字节的十六进制数据。当调试助手接收到一个数据包时，会跟各通道所定义的“帧头数据匹配”模板进行匹配（比对），只有匹配成功后，才会进一步解析该指令帧内的数据段及后续波形显示。匹配模板支持通配符（星号*），用于实现模糊匹配。一个匹配模板中最多只能出现一个星号，并且星号只能出现在匹配模板的首位。帧头数据匹配模板举例如下：

① 匹配模板：* ，表示匹配任意接收到的指令帧；

② 匹配模板：* F0 F1 F2 ，表示匹配任何包含十六进制数据 F0 F1 F2 的指令帧；

③ 匹配模板：F0 F1 F2 ，表示匹配以十六进制数据 F0 F1 F2 开头的指令帧。

【字段偏移地址】这里输入一个十进制数或 0x 开头的十六进制数，表示所监测的信号字段（如电压、温度、湿度等）在接收帧中的偏移地址（偏移量从 0 开始）。

【字段数据类型】表示上述“字段偏移地址”处对应的目标监测字段（如电压、温度、湿度等）在源指令帧中的数据类型。在右侧下拉框中可选的数据类型有：单字节有符号整数、单字节无符号整数、双字节有符号整数、双字节无符号整数、四字节有符号整数、四字节无符号整数、IEEE754 单精度浮点、IEEE754 双精度浮点。注意，这个“字段数据类型”设置中隐含了字段长度、有无符号位、整数还是浮点型等信息，用于告诉调试助手软件，如何解析这个字段对应的数据内容。

【字段字节顺序】表示上述“字段数据类型”所代表的多字节数据段是大端序（高字节在前）还是小端序（低字节在前）。对于单字节数据类型，选择大端序还是小端序没有影响。

5.4.2 启动波形监测

在建立监测通道后，就可以启动数据波形监测功能了。启动方法：点击通道列表框右下方的“启动监测”按钮，当按钮上指示灯变红时，表示波形监测（示波器）启动成功，再次点击该按钮，波形监测功能停止，按钮指示灯变灰。

当波形监测启动后，当调试助手软件每接收到一帧数据时，都会跟通道列表中，所有复选框勾选的通道逐一进行匹配，当接收帧数据跟某一个通道的帧头数据模板达成匹配时，就会根据该通道所定义的字段偏移地址及字段数据类型等信息，将目标数据字段从接收帧中提

取出来，并添加入到波形图表中去显示。

图 5-9 启动波形监测

5.4.3 波形数据保存

点击通道列表框右下方的“记录文件”按钮，可以将各通道所监测到的数据，即波形图表中记录数据，实时保存到用户指定路径的 Excel 文件中。

5.5 历史发送

工具面板中的历史发送窗口，记录着用户发送的历史指令数据。用户可随时调出历史发送窗口进行查看，如下图5-10所示。

图 5-10 历史发送记录

每一条历史发送记录的组成都分数据内容及记录时间两部分，其中记录时间就是该数据的初次发送的时间，该时间以按钮的形式显示，点击该按钮可以直接重发这条指令。但重发的指令不会历史发送窗口增加新的发送记录，除非删除存在的历史记录。实际上，历史发送窗口，只保存主窗口中通过发送按钮发送的数据，而通过快捷指令、批量发送、历史发送这些方式发送的数据是不会保存到历史发送记录中去的。

5.6 校验计算器

本调试助手软件支持发送数据时自动添加校验码，也可以自行通过调试助手提供的校验计算器来进行数据的校验码计算，然后手动添加到发送数据中。

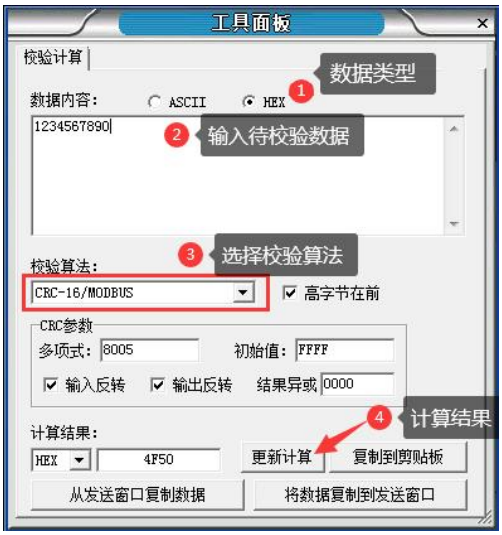


图 5-11 校验位计算器

校验位计算器从工具面板中打开，如图 5-11 所示。检验的数据类型可以选择 ASCII 码或者 HEX 码，但数据类型必须跟实际输入的数据内容一致；可供选择的主要校验算法如下：

- CHECKSUM-8： 8 位（1 字节）累加和校验码，校验码为校验数据所有字节之和。
- CHECKSUM-8N/LRC： 纵向冗余校验（LRC, Longitudinal Redundancy Check）,校验值为校验数据所有字节之和的负数。
- XOR-8/BCC： 信息组校验（BCC, Block Check Character)，校验值为校验数据所有字节的异或值。
- CRC-8： 8 位 CRC 校验，多项式 $x^8 + x^2 + x + 1$（0x07），初始值 0x00，输入数据不反转，输出数据不反转，输出结果异或值 0x00。
- CRC-16/MODBUS： 适用于 Modbus 的 16 位 CRC 校验码，多项式 $x^{16} + x^{15} + x^2 + 1$（0x8005),初始值 0xFFFF,输入数据反转,输出数据反转,输出结果异或值 0x0000。
- CRC-16/CCITT： 多项式 $x^{16}+x^{12}+x^5+1$（0x1021），初始值 0x0000，输入数据反转，输出数据反转，结果与 0x0000 异或。
- CRC-16/CCITT-FALSE： 多项式 $x^{16}+x^{12}+x^5+1$（0x1021），初始值 0xFFFF，输入数据不反转，输出数据不反转，结果与 0x0000 异或。
- CRC-16/XMODEM： 多项式 $x^{16}+x^{12}+x^5+1$（0x1021），初始值 0x0000，输入数据不反转，输出数据不反转，结果与 0x0000 异或。
- CRC-16/X25: 多项式 $x^{16}+x^{12}+x^5+1$（0x1021），初始值 0xFFFF，输入数据反转，输出数据反转，结果与 0xFFFF 异或。
- CRC-16/IBM: 多项式 $x^{16}+x^{15}+x^2+1$（0x8005），初始值 0x0000，输入数据反转，输出数据反转，结果与 0x0000 异或。
- CRC-16/MAXIM: 多项式 $x^{16}+x^{15}+x^2+1$（0x8005），初始值 0x0000，输入数据反转，输出数据反转，结果与 0xFFFF 异或。

- CRC-16/USB: 多项式 $x^{16}+x^{15}+x^{2}+1$（0x8005），初始值 0xFFFF，输入数据反转，输出数据反转，结果与 0xFFFF 异或。
- CRC-32: 多项式 x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1（0x04C11DB7），初始值 0xFFFFFFFF，输入数据反转，输出数据反转，结果与 0xFFFFFFFF 异或。
- CRC-32/BZIP2: 多项式 x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1（0x04C11DB7），初始值 0xFFFFFFFF，输入数据不反转，输出数据不反转，结果与 0xFFFFFFFF 异或。
- CRC-32/MPEG-2: 多项式 x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1（0x04C11DB7），初始值 0xFFFFFFFF，输入数据不反转，输出数据不反转，结果与 0x00000000 异或。
- MD5: 16 字节 MD5 校验码。
- FIXED HEX BYTES：固定数据校验码，校验码为任意多字节常数;

CRC 校验算法，支持 CRC8、CRC16、CRC32 三种宽度的校验算法，并可自定义修改校验参数，如多项式、初始值、输入数据翻转、输出数据翻转、输出结果异或值。通过修改 CRC 参数，可以实现任意的 CRC 校验算法。

5.7 ASCII码对照表

ASCII (American Standard Code for Information Interchange)是基于拉丁字母的一套电脑编码系统,它主要用于显示现代英语和其他西欧语言。到目前为止共定义了 128 个字符,其中包含 33 个控制字符（具有某些特殊功能但是无法显示的字符，也就是非打印字符）和 95 个可显示字符（可打印字符）。

字符	10进制	16进制
#	35	0x23
$	36	0x24
%	37	0x25
&	38	0x26
'	39	0x27
(	40	0x28
)	41	0x29
*	42	0x2A
+	43	0x2B
,	44	0x2C
-	45	0x2D
.	46	0x2E
/	47	0x2F
0	48	0x30
1	49	0x31
2	50	0x32

图 5-12 ASCII 对照表

调试助手提供 ASCII 对照表，如图 5-12 所示。可帮助用户在通信调试时对收发数据中字符数据解析处理。比如，接收到 16 进制数据后，若需要手动转换成 ASCII 码字符串时，可以借助 ASCII 码表逐个根据十六进制值查找目标字符；再比如，发送包含非打印字符的

ASCII 字符串时，可以通过转义符的方式，即反斜杠加小写字母 x 再加两位 16 进制数据的表示方法，而非打印字符对应的 16 进制值可以通过 ASCII 码表来查找。

5.8 命令行启动参数

调试助手可以通过命令行参数，按给定的参数选项启动，还可以直接通过命令行发送串口数据。根据调试助手的选择的通信协议类型选择不同，命令行启动参数的设置分三种情形：TCP Server 命令行通信、TCP Client 命令行通信、UDP 命令行通信。

5.8.1 TCP Server 命令行通信

TCP Server 通信模式时，命令行参数格式设置方法如下：

```
netassist.exe -ts -lh ip:port
```

参数说明：

- -ts 表示选择 TCP Server 协议类型；
- -lh 后面紧接着本地监听的主机地址和端口，主机地址跟端口之间通过冒号分割。注意，参数-lh 中的 l 是英文 L 的小写字符，而不是阿拉伯数字 1。

例如： 使用 TCP Server 协议监听本地 127.0.0.1:8080 端口的命令行如下：

netassist.exe -ts -lh 127.0.0.1:8080

5.8.2 TCP Client 命令行通信

TCP Client 通信模式时，命令行参数格式设置方法如下：

```
netassist.exe -tc [-rh ip:port] [-x] [-d 发送数据 [-q]] [--autoreply] [--batchsend]
```

参数说明：

- -tc 表示选择 TCP Client 协议类型；
- -rh 表示设置远程服务器的地址和端口；
- -d 表示连接服务器后待发送的数据内容,-df 表示连接服务器后发送的数据文件路径；
- -x 表示发送数据格式为十六进制字符串；
- -q 表示发送完数据后立即关闭调试助手进程；
- --autoreply 表示启动工具面板中的【自动应答】功能；
- --batchsend 表示连接成功后启动工具面板中的【批量发送】功能；

注意:当设置了-d 参数后，-q 参数才会起作用。-q 决定发送完数据是否退出调试助手软件进程。另外，-d 数后紧跟发送的数据内容，必须用双引号括起来，例如：

例 1： netassist.exe -tc -rh 192.168.1.88:8081 -d "hello" -q

命令行启动调试助手软件，自动通过 TCPClient 协议连接到服务器 192.168.1.88:8081，并自动向服务器发送字符串"hello"。执行完毕后自动关闭调试助手软件。参数-q 表示执行完退出调试助手进程。

例 2： netassist.exe -tc -rh 192.168.1.88:8081 -df "C:\test.dat"

命令行启动调试助手,自动连接指定的服务器地址 192.168.1.88:8081，然后发送数据文件

"C:\test.dat"。在-df 参数后指定待发送的数据文件路径。

5.8.3 UDP 命令行通信

UDP 通信模式时，命令行参数格式设置方法如下：

```
netassist.exe -u [-lh ip:port]  [-rh ip:port] [-x] [-d 发送数据 [-q]]
```

参数说明：

- -u 表示选择 UDP 协议类型；
- -lh 表示绑定本地主机地址和端口；
- -rh 表示设置远程目标主机的地址和端口；
- -d 表示发送的数据内容,-df 表示连接服务器后待发送的数据文件路径；
- -x 表示发送数据格式为十六进制字符串；
- -q 表示发送完数据后关闭调试助手进程。

5.8.4 指定接收日志保存路径

```
netassist.exe  -rf "日志文件路径"  [其他参数]
```

通过-rf 参数，可指定接收日志文件路径。注意，日志文件路径根据其后缀名不同，分为三种保存格式：

① 文件后缀名为.log，则接收保存为日志文件格式，保存接收与发送的全部记录包括时间戳信息。

② 文件后缀名不为.log：则接收保存为数据文件，只记录接收的原始数据，不包含发送的数据以及时间戳信息。

③ 文件路径为一个存在的文件夹：则按滚动日志的形式保存日志，每小时保存 1 个日志文件，保存 1 个月，循环滚动覆盖。

5.8.5 命令行注意事项

通过本调试助手软件命令行参数发送数据时，由于只能携带文本字符串参数，不能直接发送二进制数据，但是可以通过转义字符方式实现二进制数据的发送。比如发送一条带回车换行符结尾的 AT 指令，对应数据 -d "AT\r\n"

- 等价于 -d "AT\x0D\x0A"
- 等价于 -d "\x41\x54\x0D\x0A"
- 等价于 -d "\x[41 54 0D 0A]"
- 等价于 -d "\x[41540D0A]"
- 等价于 -x -d "41540D0A"
- 等价于 -x -d "41 54 0D 0A"

注意，使用-x 参数之后,-d 参数后面可以直接跟十六进制数据，不用转义字符。

例如，实际执行命令: `netassist.exe -tc -rh 192.168.1.88:8081 -q -d "AT\r\n"`

等价于 `netassist.exe -tc -rh 192.168.1.88:8081 -q -x -d "41540D0A"`

第六章 脚本代码语法规则

调试助手的指令数据或指令模板可以嵌入类 C 语言语法的脚本代码，方便用户灵活地编写指令或模板，实现强大的高级指令发送或者自动应答功能。

6.1 运算符

自动应答规则引擎，支持各种逻辑运算及位操作符。一共有 34 种运算符，10 种运算类型：算术运算符（+、-、*、/、%）、关系运算符（>、>=、==、！=、<、<=）、位运算符(>>、<<、==、!=、<、<=)、逻辑运算符(!、||、&&)、条件运算符、（？：）指针运算符（&、*）、赋值运算符（=）、逗号运算符（，）、强制类型转换运算符（(类型名)）、其他（下标[]、分量、函数）；若按参与运算的对象个数，运算符可分为单目运算符（如！）、双目运算符（如+、-）和三目运算符（如？：）。指令模板中实际常用的运算符,如下表所示：

优先级	运算符	名称或含义	使用形式	说明
1	()	圆括号	(表达式)、函数名(形参表)	
2	(类型)	强制类型转换	(数据类型)表达式	
	-	负号运算符	-表达式	单目运算符
	!	逻辑非运算符	!表达式	单目运算符
	~	按位取反运算符		单目运算符
3	/	除	表达式 / 表达式	双目运算符
	*	乘	表达式*表达式	双目运算符
	%	余数（取模）	整型表达式%整型表达式	双目运算符
4	+	加	表达式+表达式	双目运算符
	-	减	表达式-表达式	双目运算符
`5	<<	左移	变量<<表达式	双目运算符
	>>	右移	变量>>表达式	双目运算符
6	>	大于	表达式>表达式	双目运算符
	>=	大于等于	表达式>=表达式	双目运算符
	<	小于	表达式<表达式	双目运算符
	<=	小于等于	表达式<=表达式	双目运算符
7	==	等于	表达式==表达式	双目运算符
	!=	不等于	表达式!=表达式	双目运算符
8	&	按位与	表达式&表达式	双目运算符
9	^	按位异或	表达式^表达式	双目运算符
10	|	按位或	表达式|表达式	双目运算符
11	&&	逻辑与	表达式&&表达式	双目运算符
12	||	逻辑或	表达式||表达式	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2:表达式 3	三目运算符

6.2 运算表达式

所谓运算表达式，就是具有返回值的基于类 C 语言语法规则的计算表达式。

例如：

- \[2: 2*(getuchar(0)+getuchar(1))]
- \[（getuchar(#len)+1）: 2*(getuchar(0)+getuchar(1))]

在指令自动匹配应答过程中，运算表达式会实时动态地被最终计算值所替换。因此，运算表达式在形式上必须是一条具有返回值的脚本语句构成，而不允许分多条语句实现。如果一条语句无法实现，只能通过下一节所介绍的包含多条脚本语句的 BLOCK 代码段实现。BLOCK 代码块的返回值通过 return 语句或者 echo/echob 函数实现传递。

6.3 BLOCK 代码块

自动应答规则的指令模板，可以要使用包含多条语句的 BLOCK 语法。一个 BLOCK 的多条语句通过大括号对{}包括起来，多条语句之间用分号隔开，最终通过 return 语句，或者 echo 函数返回 BLOCK 表达式的最终值。不同的是，return 返回值后就会终止当前 BLOCK 后面的语句，而 echo 返回值之后会继续执行 BLOCK 后面的语句。如果一个 BLOCK 中执行了多次 echo，则每次返回的数据会追加到之前返回数据的后面。如果没有执行到 echo 或 return，则表示无返回数据；如果既有 echo 值，又有 return 值，则 echo 值会忽略而只取 return 值。

例如，下面这个 BLOCK 例子：

```
\[{
  int num=getuchar(#num);
  if(num>0){
    echo(0xF2F1);
    echo("\xF3\xF4%x",num);
  }
  else return 0;
}]
```

代码说明：若 if(num>0)表达式成立，则调用两次 echo，两次数据叠加，最终返回 16 进制数据流 F1 F2 F3 F4 num；若 if(num>0)表达式不成立，则返回数值 0。

作为扩展，应答规则中脚本代码支持 0x 开头的 16 进制立即数，如上面代码中的 0xF2F1 就是一个 16 进制立即数。

6.4 变量数据类型

自动应答规则引擎的内置脚本代码只能使用下表所示的基本数据类型，不支持用户自定义变量结构体。目前所支持的基本数据类型如下：

类型名	可用别名	类型说明
char		有符号字符
unsigned char	byte 或 uchar	无符号字符
short		有符号短整形
unsigned short	ushort	无符号短整形
int		有符号整形（32 位）
unsigned int	uint	无符号整形（32 位）
float		32 位浮点数
bool		布尔类型

string		字符串类型

使用限制：仅支持一维数组和一维指针，暂不支持多维数组和多维指针；暂不支持 64 位数据类型。

6.5 变量定义及作用域

自动应答规则的脚本语言支持使用变量。根据变量的强弱类型可划分为，强类型变量和弱类型变量；根据变量作用域，则可分为局部变量、全局变量、静态变量。

6.5.1 强类型变量与弱类型变量

(1) 强类型变量。强类型变量类似于标准 c 语言的变量定义方式，必须先定义后使用。强类型变量的数据类型在变量定义时就被指定，不允许动态修改变量类型。强类型变量定义时，需要指定变量的数据类型以及变量名，并且允许在变量定义时初始化赋值。如：

```
\[{ int x,y; //定义两个强类型的整形变量
    int z=100; //定义一个初值为 100 的强类型整形变量
    char *str1="abc";//定义一个 null-teminated 字符串
    string str2="abc\x00\x01\x02";//定义一个标准字符串，允许包含 0。
 }]
```

(2) 弱类型变量。弱类型变量无需声明或定义，也不用指定变量的数据类型，可以直接使用。给弱类型变量赋值时，如果变量名不存在则会自动创建该变量。弱类型变量的数据类型总是等于最后一次赋值的数据类型。也就是说，弱类型变量的数据类型是可以被动态修改的。弱类型变量必须通过保留字 global 按数组索引的方式来使用，数组的下标为字符串形式的变量名。例如：

```
\[ {
  global["x"]=100; //给弱类型全局变量 x 赋值整形数 100。
  global["x"]="abcdefg";  //给弱类型全局变量 x 赋值字符串。
 }]
```

6.5.2 局部变量、全局变量、静态变量

自动应答规则中的变量，如果是强类型方式定义的，只能作为局部变量，仅在当前指令模板的当前\[]段内有效。多个\[]内的同名局部变量互不影响。

如果需要使用全局变量，就必须使用 global["name"]形式的弱类型变量。弱类型变量的作用域覆盖所有指令模板，在调试助手整个运行期间都常驻保留在内存中。弱类型全局变量一旦初始化赋值后，就可以省略 global 关键字，而直接通过变量名访问，就跟操作强类型变量一样。仅当需要修改变量数据类型时，才必须通过 global 关键字来给变量赋值。

静态变量是通过 static 关键字定义的一种局部可见的全局变量，具有两方面的特征：1)静态变量存储在全局存储区中（类似全局变量），这样可以在下一次调用的时候还可以保持原来的赋值；2)静态变量仅仅在变量定义时所在的作用域范围内可见，这一点上看就象局部变量。并且，静态变量仅初始化赋值一次，重入时被忽略。

需要注意的是，调试助手只支持局部变量和全局变量，暂不支持静态变量。但是可以通

过代码变通实现静态变量的效果。举个例子，假定要实现一个初值为 100 的静态变量 x，定义在全局区域，只初始化赋值一次，后面每次执行到时 x 加一，实现代码如下：

```
\[ {
    if( !global["x_set"] ){    //访问任意未定义的全局变量时返回值为空
        global["x"]=100;       //给弱类型全局变量 x 赋值整形数 100。
        global["x_set"]=true; //标记目标变量已初始化,以确保以上代码只执行一次。
    }
    x++;
 }]
```

6.6 变量强制类型转换

跟标准的 C/C++语法规则一样，当操作数的类型不同，经常需要将操作数转化为所需要的类型，这个过程即为强制类型转换。

6.6.1 强制类型转换的形式

变量强制类型转换具有两种形式：显式强制类型转换和隐式强制类型转换。下面就两种形式分别进行简单的描述。

（1） 显式强制类型转换

显式强制类型转换很简单，格式为： TYPE b = (TYPE) a;

其中，TYPE 为类型描述符，如 int，float 等。经强制类型转换运算符运算后，返回一个具有 TYPE 类型的数值，这种强制类型转换操作并不改变操作数本身，运算后操作数本身未改变，例如：

```
int n=0xab65;
char a=（char）n;
```

上述强制类型转换的结果是将整型值 0xab65 的高端一个字节删掉，将低端一个字节的内容作为 char 型数值赋值给变量 a，而经过类型转换后 n 的值并未改变。

（2）隐式强制类型转换

隐式类型转换发生在赋值表达式和有返回值的函数调用表达式中。在赋值表达式中，如果赋值符左右两侧的操作数类型不同，则将赋值符右边操作数强制转换为赋值符左侧的类型数值后，赋值给赋值符左侧的变量。在函数调用时，如果 return 后面表达式的类型与函数返回值类型不同，则在返回值时将 return 后面表达式的数值强制转换为函数返回值类型后，再将值返回，如：

```
int n;
double d=3.88;
n=d; //执行本句后，n 的值为 3，而 d 的值仍是 3.88。
```

6.6.2 强制类型转换在自动应答规则中的典型用途

在指令应答模板中的模式应答数据段，如果不显式指定数据长度，则默认长度为其数据类型的固有长度。

比如，有一个整形数据段,默认长度为 4 字节。如要要求只取 2 个字节。可以显示指定数据长度为 2，或者强制类型转换为 short 类型。例如，以下两个指令应答模板等价：

```
\[2:getshort(0)*100]                    //#直接指定数据长度 2
\[(short)(getU16(0)*100)]          //#强制类型转成 2 字节的 short 数据类型
```

注：数值计算表达式的返回值,如果不做强制类型转化，则默认数据类型是 32 位的 int 或者 float 类型，长度 4 字节。

6.7 语法大小写规则

调试助手内置脚本代码是不区分大小写的。但是为避免混乱，推荐大家始终坚持“大小写敏感”的代码书写规范，尽量保证函数名、变量名、常数名等大小写前后书写一致。

6.8 字段注解的定义及引用

不管是脚本指令还是指令模板（包括指令匹配模板及指令应答模板）都可以包含若干个模式段，每个模式段的一般形式都可以归结为：

```
\[exp_len:exp_value#comment]
```

其中，exp_len 为字段数据长度、exp_value 为字段数据内容、#comment 为字段注解。注解的形式为#号开头加注解文字。注解字段的作用有两个：

（1）起注释作用，对目标字段作解释说明。

(2) 给字段命名，经过命名的字段数据可以被同一个指令模板的其他模式段通过注解名引用。引用注解只能通过以下这几个用于读取指令数据段的系统函数: gets、getchar、getuchar、getshort、getushort、getint、getuint。除此之外，其他函数都不支持调用注解作为参数。

下面举个简单例子作说明：

- 指令匹配模板：REQ\[2 #command]
- 指令应答模板: ACK\[2:gets(#command,2)]

本例指令匹配模板中，\[2 #command]定长模糊匹配 2 个字节的 command 数据；而在应答模板中需要复制这两个字节内容作为应答数据帧的一部分。在应答模板对应的模式应答段中，通过 gets(#comment,len)函数可以读取注解名对应的数据块，省略 gets 的第二个参数表示读取整个目标字段。本例中，应答模板需要复制请求指令中整个 command 段的数据，因此可以省略 gets 函数的第 2 个表示长度的参数。这样，此应答模板可以简化为：

- 指令应答模板→简化: ACK\[2:gets(#command)]

如果模式应答段需要的数据长度跟 gets 返回的数据块长度一致，就可以省略模式应答段的长度参数，如下所示：

- 指令应答模板→再简化: ACK\[gets(#command)]

6.9 内建系统函数详解

自动应答规则的运算表达式,可以调用规则引擎内建的系统函数(函数名不区分大小写)，目前支持的函数如下。

（1）printf - 格式化输出到控制台

函数原型: void printf(const char format, 可选参数...);

功能描述：按指定格式向标准输出设备（日志/接收窗口）输出调试打印信息。

（2）sprintf - 格式化字符串

函数原型: int sprintf(char *buffer, char *format [,argument,...]);

功能描述:把格式化的数据写入某个字符串缓冲区。如果成功，则返回写入的字符总数，不包括字符串追加在字符串末尾的空字符。如果失败，则返回一个负数。

（3）strtoint - 字符串转整形数

函数原型: int strtoint(const char *nptr);

函数别名: atoi

功能描述：strtoint 函数会扫描参数 nptr 字符串，跳过前面的空白字符（例如空格，tab 缩进）等，扫描直至下一个非数字字符或到达输入的结尾。如果 nptr 不能转换成整数，那么将返回 0。

（4）inttostr - 整形数转字符串

函数原型: string inttostr(int n);

功能描述：将整形数 n 转换成字符串类型返回。

（5）strcpy - 字符串拷贝

函数原型: char *strcpy(char* dest, const char *src);

功能描述：把 src 指向的 null-terminated 字符串复制到 dest 指向的地址空间，并返回指向 dest 的指针。

（6）strcat - 字符串拼接

函数原型: char *strcat(char *dest,const char *src1, const char *src2，…);

功能描述：把若干个 NULL 结尾的源字符串（src1、src2、…）复制拼接到 dest 所指向的字符串后面（删除 dest 原来末尾的“\0”）。要保证 dest 指向的字符串空间足够长，以容纳被复制进来的源字符串。最后返回指向 dest 的指针。

（7）strcmp - 字符串比较（区分大小写）

函数原型: int strcmp(const char *s1,const char *s2);

功能描述：两个字符串自左向右逐个字符相比（按 ASCII 值大小相比较），直到出现不同的字符(区分大小写)，或遇'\0'为止。当 s1<s2 时，返回为负数；当 s1=s2 时，返回值= 0；当 s1>s2 时，返回正数。

（8）stricmp - 字符串的比较（不区分大小写）

函数原型: int stricmp(const char *s1,const char *s2);

功能描述：两个字符串自左向右逐个字符相比（按 ASCII 值大小相比较），直到出现不同的字符(不区分大小写)，或遇'\0'为止。当 s1<s2 时，返回为负数；当 s1=s2 时，返回值 = 0；当 s1>s2 时，返回正数。

（9）strncpy - 限定长度的字符串拷贝

函数原型: char *strncpy(char* dest, const char *src, int n);

功能描述：把 src 指向的 null-terminated 字符串复制到 dest 指向的地址空间，并返回指向 dest 的指针。如果源字符串实际长度大于参数 n，则最多复制 n 个字节。

（10）strncmp - 限定长度的字符串比较（区分大小写）

函数原型: int strncmp(const char *s1,const char *s2,int n);

功能描述：两个字符串自左向右逐个字符相比（按 ASCII 值大小相比较），直到出现不同的字符（区分大小写），或遇'\0'，或比较字符数超过 n 为止。当 s1<s2 时，返回为负数；当 s1=s2 时，返回值= 0；当 s1>s2 时，返回正数。

（11）strnicmp - 限定长度的字符串比较（不区分大小写）

函数原型: int strnicmp(const char *s1,const char *s2,int n);

功能描述：两个字符串自左向右逐个字符相比（按 ASCII 值大小相比较），直到出现不同的字符（不区分大小写），或遇'\0'，或比较字符数超过 n 为止。当 s1<s2 时，返回为负数；当 s1=s2 时，返回值= 0；当 s1>s2 时，返回正数。

（12）strlen - 计算字符串长度

函数原型 1: int strlen(const char *s);

函数原型 2: int strlen(string s);

功能描述：返回字符串长度。

（13）memcpy - 内存数据复制

函数原型: void *memcpy(void *dest, void *src, int n);

功能描述：从存储区 src 复制 n 个字节到存储区 dest。返回指向目标存储区 dest 的指针。

（14）memcmp - 内存数据比较

函数原型: int memcmp(const void *data1, const void *data2, int n);

功能描述：把存储区 data1 和存储区 data2 的前 n 个字节进行比较。如果返回值 == 0，则表示 data1 等于 data2；如果返回值 < 0，则表示 data1 小于 data2;如果返回值 > 0，则表示 data2 小于 data1。

（15）string - 标准字符串构造方法

- 函数原型 1: string string(int len);

功能：构造并返回一个预留空间长度为 len 的空字符串。

- 函数原型 2: string string(void *str, int len);

功能：构造一个长度为 len 的空字符串，并用 str 指向的数据进行初始化。

- 函数原型 3: string string(string1,string2, …);

功能：将若干个 string 或 char *类型的字符串依次首尾连接起来，构造出一个新字符串返回。

（16）unix_timestamp - 获取 32 位 unix 时间戳

函数原型: unsigned int unix_timestamp(void);

入口参数: 无

返 回 值: 返回 32 位无符号整数。

功能描述：生成 32 位 unix 时间戳，即从 1970-1-1 00:00:00 到当前的秒数。

（17）genAutoID - 生成 32 位自增流水 ID

函数原型: unsigned int genAutoID(void);

入口参数: 无

返 回 值: 返回 32 位无符号整数。

功能描述：初始值为 1，每调用一次，返回值自动加 1。

（18）random - 生成随机数/随机选择集合数据

函数原型 1: int random (int maximum);

功能描述: 生成一个绝对值小于入参 maximum 的 32 位随机数。如果不指定上限（省略 maximum 参数）,则随机生成 1 个 32 位随机数。

函数原型 2: var random (var1,var2,…);

入口参数: 二个以上任意类型数据

返 回 值: 随机返回入口参数列表中的一个。

功能描述：随机选择集合数据，即从多个数据中随机选择一个数据返回。

例如： random(100,0x255,123.456, 'x',"abcdefg")，实现从入口参数列表中随机返回一个数据，入口参数的数据类型可以自由混合，返回值类型就是随机选择参数的实际数据类型。

（19）reverse 逆转数据的字节顺序

函数原型: var reverse(data, maxLen);

入口参数: 参数 data 为待逆序的源数据，可以是整形（短整形或长整形）或浮点等基本数据类型，也可以是字节型数组、字符串或数据指针类型；可选参数 maxLen 用于指定数据转换的最大长度，如果省略该参数则转换长度取源数据 data 的默认长度值，如 int 类型

数据默认长度 4 字节，short 类型默认 2 字节,字符串则自动获取字符串自身长度，等等。

功能描述：将源数据 data 的字节顺序高低逆转后返回逆序重排的数据。如果源数据类型是整形（短整形或长整形）或浮点等基本数据类型，则不会修改源数据的字节顺序，而是返回逆序后的数据；如果源数据类型是字符串、数组或指针类型，则源数据也会发生字序逆转，并返回逆序后的数据引用。

（20）gets - 从当前指令数据中复制数据段

函数原型: var gets(offset|#comment,len);

入口参数: offset|#comment 为偏移地址或者字段注解名。如果是通过偏移地址复制数据，则需要明确这个偏移地址是相对当前模板对应的指令数据；如果通过字段注解名复制数据，则系统会优先查找源指令帧对应的注解字段，如果不存在则再查找应答指令帧对应注解字段，并且引用的目标注解名必须确保在当前模板中调用 gets 函数前已经定义过。

功能描述：从指定位置（当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段）处，拷贝指定长度为 len 的数据块；长度参数 len 可以省略，如果使用偏移地址拷贝，则省略长度参数时将拷贝数据直至指令帧末尾；如果使用字段名注解，则省略长度参数时表示拷贝注解名对应的整个字段的数据。

例如，以下这条应答规则：

指令匹配模板：ABCD\[2#command]

指令应答模板: ACK_[4:gets(0)]_\[2:gets(#command)]

假定，自动应答规则引擎收到字符串数据 ABCDFF，则指令匹配成功，其中定长模糊匹配\[2#command]所匹配的内容是 FF。根据指令应答模板，生成应答数据为: ACK_ABCD_FF。其中，\[4:gets(0)]表示从当前收到的数据报文（ABCDFF）的偏移地址 0 开始拷贝 4 个字节数据:ABCD；\[2:gets(#command)]表示从当前收到的数据报文（ABCDFF）的注解#command 所匹配对应的偏移地址处拷贝 2 个字节数据:FF。

（21）getchar - 从当前指令数据中复制一个字节有符号数

函数原型: char getchar(offset|#comment);

函数别名：getS8

入口参数: 指令帧偏移地址或者模板字段注解名(参考前文 gets 函数的参数说明)。

功能描述：从指定位置（当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段处）拷贝 1 个字节的有符号数据。

（22）getuchar - 从当前指令数据中复制一个字节无符号数

函数原型: unsigned char getuchar(offset|#comment);

函数别名：getU8、getByte

入口参数: 指令帧偏移地址或者模板字段注解名(参考前文 gets 函数的参数说明)。

功能描述：从指定位置（当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段处）拷贝 1 个字节的无符号数据。

（23）getshort - 从当前指令数据中复制 2 个字节有符号整数

函数原型: short getshort(offset|#comment,isBigEndian);

函数别名：getS16

入口参数: offset|#comment 为偏移地址或者字段注解名；isBigEndian 为可选参数，布尔型数据变量，表示指定读取数据的字节顺序是高字节在前还是在后。为 true 表示高字节在前否则低字节在前。如果省略 isBigEndian 参数，表示取全局的字节顺序设置（自动应答设置窗口面板右下方的“网络字序”复选框，用于设置全局的默认字节顺序，勾选表示全局字序为 BigEndian，否则为 LittleEndian）。

功能描述：从指定位置（当前指令帧 offset 偏移地址处,或者模板字段注解名对应的指令数据段）拷贝 2 个字节的有符号整数。

（24）getushort - 从当前指令数据中复制 2 个字节无符号整数

函数原型: unsigned short getushort(offset|#comment, isBigEndian);

函数别名：getU16

入口参数: offset|#comment 为偏移地址或者字段注解名；isBigEndian 为可选参数，布尔型数据变量，表示指定读取数据的字节顺序是高字节在前还是在后。为 true 表示高字节在前否则低字节在前。如果省略 isBigEndian 参数，表示取全局的字节顺序设置。

功能描述：从指定位置（当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段处）拷贝 2 个字节的无符号整数。

（25）getint - 从当前指令数据中复制 4 个字节有符号整数

函数原型: short getint(offset|#comment, isBigEndian);

函数别名：getS32

入口参数: offset|#comment 为偏移地址或者字段注解名；isBigEndian 为可选参数，布尔型数据变量，表示指定读取数据的字节顺序是高字节在前还是在后。为 true 表示高字节在前否则低字节在前。如果省略 isBigEndian 参数，表示取全局的字节顺序设置。

功能描述：从指定位置（当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段处）拷贝 4 个字节的有符号整数。

（26）getuint - 从当前指令数据中复制 4 个字节无符号整数

函数原型: unsigned short getuint(offset|#comment, isBigEndian);

函数别名：getU32

入口参数: offset|#comment 为偏移地址或者字段注解名；isBigEndian 为可选参数，布尔型数据变量，表示指定读取数据的字节顺序是高字节在前还是在后。为 true 表示高字节在前否则低字节在前。如果省略 isBigEndian 参数，表示取全局的字节顺序设置。

功能描述：从指定位置（当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段处）拷贝 4 个字节的无符号整数。

（27）sets - 在应答模板中设置数据

函数原型: void sets(offset|#comment, srcData, len);

入口参数: offset|#comment 为应答帧内的待复制的目标偏移地址或者字段注解名；srcData 为待复制的源数据（可以是整型数、或字符串、或数组、或数据指针类型）；len 为可选参数为整数类型，表示复制的数据长度，如果省略改参数，则复制长度自动取目标数据类型及源数据类型的最小长度。

功能描述：将源数据 srcData 复制到应答帧的指定位置（应答帧内 offset 偏移地址或者字段注解名对应的数据段）处。len 参数表示复制数据的长度，如果省略了 len 参数，则根据源数据以及目标数据类型自动选择长度，

版本要求：该函数要求调试助手版本号>= V5.0.11.1

（28）sizeof - 计算数据长度

函数原型: int sizeof(var|#comment);

入口参数: var|#comment 为数据变量或者字段注解名；

功能描述：返回目标变量或注解字段的数据长度。

版本要求：该函数要求调试助手版本号>= V5.0.8.6

（29）getFileContents - 获取文件内容

函数原型: string getFileContents (string filePath);

入口参数: filePath 为待读取的文件路径；

功能描述：返回指定路径的文件数据内容(最长只能读取 64KB 长度的文件数据)。

版本要求：该函数要求调试助手版本号>= V5.0.10.0

（30）calculate - 计算校验位

函数原型: var calculate(offset, len, algorithm);

入口参数:参数 offset 为待校验数据偏移地址；len 为待校验数据长度，如果 len 为-1，表示数据长度截止到当前 calculate 函数调用位置的前一个字节数据,len 为-2 则表示数据长度从当前位置往前推 2 个字节，以此类推；参数 algorithm 为选择的算法。

功能描述：计算数据校验值。函数返回值的数据类型根据具体的校验算法而定。如果校验值是单字节的，则返回的数据类型是 unsigned char，如果校验值是双字节的，则返回值的数据类型是 unsigned short;如果校验值是四字节的,则返回值数据类型是 unsigned int。

目前 calculate 支持的算法（参数 algorithm 可取常量值）列表如下：

- ALGO_ACC　//1 字节累加和
- ALGO_LRC　//1 字节累加和再取负
- ALGO_XOR　//1 字节异或校验
- ALGO_CRC16_MODBUS　//2 字节 MODUBUS CRC16 校验
- ALGO_CRC8

- ALGO_CRC8_ITU
- ALGO_CRC8_ROHC
- ALGO_CRC8_MAXIM
- ALGO_CRC8_WCDMA
- ALGO_CRC8_CDMA2000
- ALGO_CRC16_CCITT
- ALGO_CRC16_CCITT_FALSE
- ALGO_CRC16_XMODEM
- ALGO_CRC16_X25
- ALGO_CRC16_IBM
- ALGO_CRC16_USB
- ALGO_CRC16_MAXIM
- ALGO_CRC16_DNP
- ALGO_CRC32
- ALGO_CRC32_BZIP2
- ALGO_CRC32_MPEG2
- ALGO_CRC32_POSIX
- ALGO_CRC32_JAMCRC

（31）echo －实现 BLOCK 代码块的流式返回值

函数原型：int echo (const char *format,…);

功能描述：格式化输出的文本数据作为 BLOCK 代码块的最终返回值，而 echo 函数本身将返回所生成文本数据的长度。

例如：

```
\[{ echo("hello\r\n"); echo("ok\r\n")  }]
\[{ echo("ERR=%d",errno);  }]
\[{ echo (gets(#Annotation)); }]
```

注，echo 的第二个参数可以省略。当只有 1 个参数时，不做格式化处理，直接输出第一个参数指向的数据。形式为\[{…}]的 BLOCK 代码块，可以通过 return 语句或者 echo 函数来实现整个 BLOCK 的返回值。不同的是 return 语句执行后就会立即退出当前 BLOCK，不会再执行当前 BLOCK 的后续代码；而 echo 语句则不会退出 BLOCK，后续如果有多个 echo 函数输出数据，这些数据会流式拼接在一起作为当前 BLOCK 的返回值。

（32）echob － 实现 BLOCK 代码块的流式返值

函数原型：void echob (data,length);

入口参数：data 为输出的数据内容（可以是整形数或字符串或数组或数据指针等）；length 为限定输出数据的字节长度。

功能描述：输出二进制数据作为 BLOCK 代码块的返回值。同一个 BLOCK 内的多个 echo

或 echob 函数的输出会按流式数据的方式依次追加合并。

例如:

```
\[{  echob(gets(0),100));  }]
\[{  echob(gets(#Annotation));  }]
\[{  char str[3]="\x01\x02\x03"; echob(str,3);  }]
```

注， echob 的第二个参数可以省略。当只有 1 个参数时，echo 和 echob 这两个函数等价，按输入数据的固有长度输出。

（33）send – 向当前通信接口发送数据

函数原型: void send (data,length);

入口参数: data 为输出的数据内容（可以是字符串或数组或数据指针等）；length 为限定发送数据的字节长度，如果发送数据类型为字符串则可省略 length 参数。

功能描述：向当前通信接口发送数据。跟 echo/echob 不同之处在于：一个 BLOCK 代码中的多个 echo/echob 语句输出的数据流会合并在一起发送，而 send 函数则是调用一次发送一次，多个 send 函数中间可以插入 delay 函数，实现自动应答时的一问多答的效果。

例如:

```
\[{ char str[3]="\x01\x02\x03";
    echob(str,3);
    delay(100);
    send("OK"); }]
```

注意: send 函数要求调试助手软件升级到 V5.0.11 或以上版本。

（34）delay – 延迟操作

函数原型: void delay(ms);

入口参数: ms 为延迟的毫秒数。

功能描述：这个函数主要用于应答模板，实现延迟应答。

例如以下应答模板:

```
\[{delay(1000); echob(gets(0),100));}]
\[{delay(500);}]OK\r\n
```

通过在应答模板头部插入 delay 函数，实现应答数据的延迟输出(发送)。

第七章 自动应答规则设计

使用调试助手软件的“指令自动应答”功能，要预先根据应用场景建立若干条自动应答规则。当调试助手软件接收到通信数据/指令帧时，会按照其优先级顺序逐一遍历匹配自动应答规则列表，直至找到一条相匹配的应答规则，然后根据对应的应答模板自动进行应答回复。自动应答规则的具体设计规则将会在本章节中详细介绍。

7.1 应答规则概述

自动应答功能，用于实时对调试助手接收到的指令/数据进行匹配/识别，并自动按用户预定义规则发送相应的应答数据。用户只须事先设计好应答规则，然后调试助手内部集成的规则引擎会自动对接收到的数据进行指令规则匹配及应答数据的发送。如果需要同时支持多条指令的自动应答，只要相应地建立多条自动应答规则。如果某一条指令帧数据能匹配多个自动应答规则，则最终只会响应那个优先级序号最高的规则。

每一条应答规则都由一对“指令匹配模板”和“指令应答模板”构成，每个模板的数据格式可以是十六进制形式，或者是 ASCII 码字符串形式。其中，ASCII 码形式的指令模板数据，可嵌入基于类 C 语言的脚本代码，方便用户实现更加灵活、强大的自动应答功能。

图 7-1 用户创建的应答规则列表窗口以及右键菜单

自动应答规则的管理面板，见图 7-1 所示。窗口中按优先级顺序逐条显示用户定义的自动应答规则。每一条应答规则项必须勾选相应的复选框后才能启用，同时还必须勾选窗口右下方名为“启动自动应答”的总开关后才能最终开启自动应答功能。

通过自动应答窗口的右键菜单(如图 7-1 右侧所示)，可以对应答规则做各种增、删、改、查、导入、导出等操作。另外，在窗口左下角有一排图标快捷按钮，用于常规快捷操作。比如，“新建”命令用于新建一条自动应答规则；“删除”命令用于删除选中的规则项目；“上移”和“下移”就是调整自动应答规则的优先级顺序。注意，当调试助手接收到通信数据时，会按照此优先级顺序进行指令模板匹配，优先级数字越小，优先级级越高，一旦成功匹配一条规则，就会发送相应的应答数据，并停止继续往下匹配。

关于自动应答的“粘包”及“半包”问题说明。“粘包”问题就是，当发送端密集发出多个连续的数据报文时，由于间隔太短或者通信延迟问题，有可能到达接收端时会粘合成一

整个没有时隙间隔的大包，无法分离出原来的多个小包；而“半包”问题就是当发送端发出一个较大的数据报文时，由于中继环节或本地软/硬件接收队列容量的问题，可能导致自动分包的产生，也就是最终接收到的是多个分开的小包。“粘包”及“半包” 问题在串口以及 TCP 等流式通信时无法避免。注意：“自动应答”模块支持对“粘包”数据进行自动分包后再处理，但不支持对“半包”数据进行自动组包后再匹配。

7.2 应答规则入门

在自动应答管理窗口，通过右键菜单的“新建…”命令或者窗口左下角的快捷图标按钮，进入到“添加新规则”界面，如下图所示。

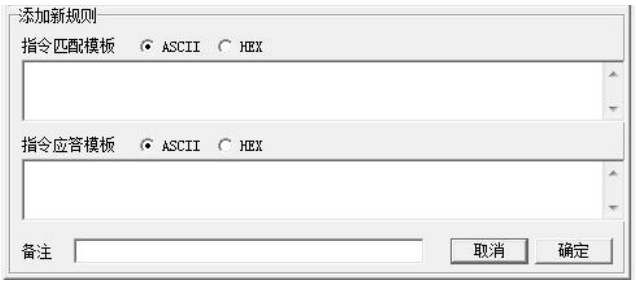


图 7-2 添加新规则编辑窗口

每一条应答规则都包含“指令匹配模板”及“指令应答模板”这两个必填的数据输入项，前者用于对接收的指令数据进行特征匹配识别，后者用于提供自动应答的数据帧；“备注”为选填的注释性文字，便于用户辨识或理解。最后点击“确认”按钮进行保存。“指令匹配模板”及“指令应答模板”的数据可以选择 ASCII 码格式或十六进制（HEX）形式，而最终选择的格式必须跟实际输入的数据一致。下面列举几个简单的例子来帮助理解。

7.2.1 ASCII 码应答规则创建实例

一个简单的示例,如下图 7-3(a)：指令匹配模板输入 ASCII 码字符串“how are you”，指令应答模板输入 ASCII 码字符串“I’m fine.”，备注项随便填，比如“greet”。最后点击“确定”按钮保存。如图 7-3(b)所示，在自动应答模块窗口就会出现新建的规则项。

图 7-3(a) 新建 ASCII 码应答规则示例

图 7-3(b)应答规则列表管理窗口

如果要进行自动应答的测试，只须按照如图 7-3(b)所示，勾选目标应答规则项前面的复选框，并勾选（点亮）右下角的总开关“启动自动应答”。此时，只要接收到包含“how are you”的数据帧，就会自动回复应答报文帧“I’m fine.”。

7.2.2 十六进制应答规则创建实例

如果指令模板中要求包含非打印的 ASCII 码字符，无法直接输入时，可以选择十六进制模式下输入十六进制数据的方式实现，或者在 ASCII 码模式下使用转义字符的方式实现。示例如下图所示：

图 7-4 新建 HEX 码应答规则示例

实际上，此例创建的是一条 AT 指令。等价地，可以使用包含 16 进制转义符的 ASCII 码字符串来表示，如下图所示。

图 7-5 新建包含 16 进制转义符的 ASCII 码应答规则示例

补充说明，本例所描述的 AT 指令应答规则，还可以采用字符串混合转义字符的方式实现。如下图所示：

图 7-6 新建字符串混合转义字符应答规则示例

注意，回车换行符除了可以用转义 16 进制 ASCII 码\x0D\x0A 的形式表示外，还可以用

\r\n 表示。

图 7-7 应答规则列表管理窗口

以上图 7-4、7-5、7-6 所创建的三条规则 AT_TEST、AT_TEST2、AT_TEST3 相互等价，其实现的效果相同，都是在接收到 AT 指令“AT\r\n”后发送应答数据“OK\r\n”。

实际应用时，“指令匹配模板”和“指令应答模板”的编码形式（ASCII/HEX）可根据实际的应用场景灵活选择：可以都为 ASCII 码或都为 HEX 码，也可以一个为 ASCII 码，另一个为 HEX 码。但要注意的是：转义字符只允许在 ASCII 码模式下使用。如果选择了十六进制模式，就不支持输入转义字符。

7.2.3 模糊匹配实现 ECHO 功能实例

所谓 ECHO 功能，就是收到任意数据帧内容，都原样发送回去。在介绍如何实现 ECHO 功能前，需要明确一点：自动应答规则触发的前提是指令匹配，而指令匹配方式是通过指令匹配模板来定义的。指令匹配模板支持使用通配符进行模糊匹配，从而可实现灵活的指令匹配方式。

模糊匹配使用的通配符有两种：一种是弹性模糊匹配\[*]，该通配符用于匹配任意长度的连续数据，单独使用\[*]时呈贪婪匹配特性；另一种是定长模糊匹配\[n]，其中 n 是正整数，该通配符用于匹配指定字节长度为 n 的任意连续数据。注意，通配符号是反斜杠开头的转义字符形式。

实现 ECHO 功能很简单，只要将指令匹配模板和指令应答模板都设为字符串\[*]即可。

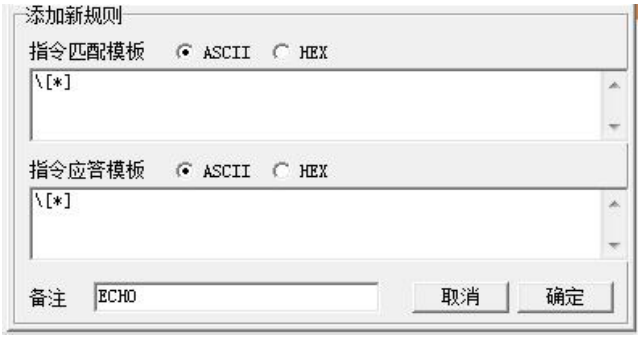


图 7-8 经典 Echo 功能实现

上图所述的规则生效后，不管收到任何数据都会匹配成功，并且会将接收到的数据原样回复给对方。

7.2.4 嵌入脚本代码的高级应答规则

自动应答规则数据中可以嵌入基于类 C 语言语法的脚本代码，通过设计业务逻辑代码，允许用户在自动应答过程中，对接收的指令/数据进行判断、计算、校验及多种模式的匹配，并在应答数据中插入对源指令数据的引用以及通过调用系统函数以及运算表达式动态生成满足用户设计需求的应答数据帧。

下面通过一个简单的例子，来直观地展示下高级应答规则的基本能力。如下图所示，这是 Modbus-RTU 的 04 号命令读寄存器数据的自动应答规则示例。

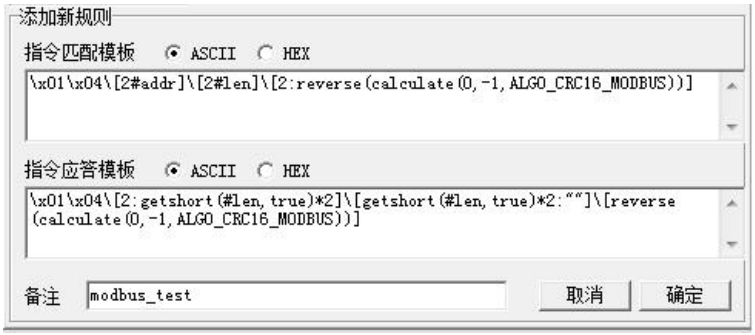


图 7-9 Modbus-RTU 读寄存器自动应答功实现

指令匹配模板中，\[2#addr]表示定长模糊匹配任意两个字节，这里用于匹配 2 个字节的地址数据；井号后面是注解，属于于注释性文字；同理，\[2#len]表示模糊匹配 2 个字节的长度数据；接下来的\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]表示定长匹配两个字节的数据，该数据的值要求等于冒号后面的计算表达式。其中，系统函数 reverse 的作用是将整形数据的高低字节交换；系统函数 calculate 用于计算校验值，其第一个参数，用于表示校验数据的偏移地址为 0，即从第一个字节开始计算校验。第二个参数为校验数据的长度，这里的-1 表示校验数据的长度截止到当前函数调用位置的前一个字节数据。第三个参数是 ALGO_CRC16_MODBUS，表示检验采用的算法。

同理，指令应答模板也采用表达式及函数的形式产生应答帧的数据内容。具体的语法规则及详细的函数定义将在后面再详细介绍。

7.3 指令匹配模板

指令匹配模板是实现自动应答的前提基础，是指令匹配的依据。每一个指令匹配模板，都由若干个指令匹配字段组成；每一个指令匹配字段的匹配模式，根据语法形式不同，可分为立即数匹配和非立即数匹配，如下图所示。

图 7-10 指令字段匹配模式分类

其中，“立即数匹配”，本质上是常量匹配，而与之相反的“非立即数匹配”，就是变量匹配。“非立即数匹配”，也称作“模式匹配”，这是因为其语法规则使用模式符\[…]来标记具体的匹配特征。根据匹配的数据对象的不同，又可分为精确匹配和模糊匹配两种基本类型。其中，精确匹配可细分为立即数（精确）匹配和定长精确匹配两个子类型；模糊匹配又可分为定长模糊匹配及不定长（弹性）模糊匹配两个子类型。

在同一个指令匹配模板中，多种模式的匹配字段可以根据实际情况自由灵活地混合使用。

7.3.1 立即数匹配

“立即数匹配”，属于指令匹配模板的一种精确匹配方式，其所匹配的对象是固定的常数立即数，而不是变量、也不是计算表达式或其他需要动态计算的值。比如，以下几个指令匹配字段都是立即数匹配模式：

立即数字段匹配模板	模板编码
AT\r\n	ASCII
\x41\x54\x0D\x0A	ASCII
41 54 0D 0A	HEX

从语法特征来看，立即数匹配模式不能使用模式符\[…]，因而不支持任何动态语法特性，不支持任何脚本代码，仅能使用纯粹的 ASCII 码字符串（支持转义字符）或十六进制编码常数。不管是 ASCII 编码还是十六进制编码的指令模板，都可以使用立即数匹配模式。不同的是，十六进制编码的指令模板由于不能使用模式符\[]，因而仅可使用立即数匹配，而 ASCII 编码的指令模板，可以混合使用多种不同的匹配模式。

7.3.2 定长精确匹配

定长精确匹配，用于精确匹配一个指定长度和内容的指令数据段。定长精确匹配的语法形式如下所示：

\[exp_len: exp_value #comment]

其中，参数 exp_len 表示匹配的数据长度，参数 exp_value 表示匹配的数据内容。exp_len 和 exp_value 在形式上都可以是常数、或运算表达式、或 BLOCK 代码块，可以在指令匹配阶段动态计算待匹配的数据长度及内容。如果 exp_len 跟 exp_value 的实际数据内容长度不一致，则最终的匹配长度以 exp_len 为准，超出则截尾，不足则补零；comment 是注解，也就是注释性文字，可以省略。在语法规则上要注意，模式符\[]的反斜杠跟左中括号之间没有空格、数据长度 exp_len 跟变量数据 exp_value 之间通过冒号分隔；注解 comment 以井号开头。

比如，以下几个指令匹配段都是定长精确匹配：

匹配模板	模板数据
\[2: calculate(0,-1,CRC16_MODBUS)]	*调用内建函数计算匹配一个 2 字节长度的 CRC 校验位*
\[1: 2*(getuchar(0)+getuchar(1))]	通过运算表达式计算匹配变量值

\[(getuchar(#len)+1): 2*(getuchar(0)+getuchar(1))]	长度及数据都是计算表达式

\[{ return getint(0)}:{
byte value=getbyte(#value);
return value?1:0;
}]
	匹配数据长度及内容都是包含在{}内的 BLOCK 代码块，通过 return 或 echo 返回 BLOCK 代码块的计算值

注意：定长精确匹配模式必须指定变量数据的长度及内容，两者都不可省略，仅注解是可以省略的可选项。如果省略了 exp_len 或 exp_value,仅剩下一个参数段就变成了定长模糊匹配模式，而不再是定长精确匹配。

如果 exp_len 或 exp_value 的表达式有多条语句组成，可以通过包含在大括号对{}内的 BLOCK 代码块来调用，一个 BLOCK 内多条语句之间用分号隔开，BLOCK 的最终返回值通过 return 或者 echo 语句实现，例如：

```
\[ {
     int i=getuint(0);
     return i;
   }:{
     if(getuchar(4)==0) return 10;
     else return 20;
     }
 ]
```

7.3.3 定长模糊匹配

所谓定长模糊匹配，就是在指令的部分字段匹配时，只匹配数据长度而不关心数据的内容，也就是匹配指定长度的任意数据。其语法形式如下所示：

```
\[ exp #comment]
```

其中，参数 exp 表示匹配的数据长度,形式上可以是常数立即数或运算表达式或 BLOCK 代码块；#comment 是注解，也就是注释性文字，可以省略。在语法形式上，定长模糊匹配比定长精确匹配模式少一个用于匹配内容的参数字段。从这点也可以看出，定长模糊匹配只匹配数据长度，不匹配数据内容。

模糊匹配模式举例：

- \[n #定长模糊*匹配 n 个字节任意数据，其中 n 为任意非零正整数*]
- \[1 #定长模糊*匹配 1 个字节任意数据*]
- \[4 #定长模糊*匹配 4 个字节任意数据*]
- \[getint(0) *#定长模糊匹配的长度为表达式计算值*]
- \[{byte len=getbyte(0);return len;} #*定长模糊匹配的长度为 BLOCK 代码返回值*]

7.3.4 弹性模糊匹配

弹性模糊匹配，简称弹性匹配，是一种不定长度的模糊匹配，支持正则字符集匹配规则，并可设定最小及最大匹配长度,而最终实际匹配的数据长度会根据上下文数据内容以及设定

的边界条件弹性自适应。弹性匹配的语法形式如下所示：

```
\[*(minLen,maxLen,filter) #comment]
```

上式中，星号表示弹性匹配，而星号后面的圆括号内指定了弹性匹配的三个参数（下文统称为弹性参数)：弹性下界 minLen、弹性上界 maxLen、弹性过滤器 filter。如果将部分弹性参数省略掉,再将注解省略掉，则弹性模糊匹配的表达式由简至繁，共可化为四种样式:

① \[*] *#弹性参数为空，表示任意模糊匹配*

② \[*(minLen)] *#仅指定弹性下界的弹性匹配*

③ \[*(minLen,maxLen)] *#指定弹性上下界但无过滤器的弹性匹配*

④ \[*(minLen,maxLen,filter)] *#指定所有弹性参数的弹性匹配*

弹性匹配过程中，会从指定的下界（最小长度 minLen）开始，根据数据过滤规则（filter）遍历目标数据，一旦找到满足整个指令匹配模板的最小模糊匹配长度，或者匹配长度超过设定的上界（最大长度 maxLen），就会停止匹配。特别注意，一条指令模板可以包含多个弹性匹配表达式，任何位于模板模板的头部和中间的弹性匹配都是非贪婪匹配，只有位于指令模板结尾的弹性匹配才是贪婪匹配。如果某个弹性匹配既在模板头部也在尾部，也即是说这个指令模板就是有 1 个弹性匹配构成的，那么这个弹性匹配属于贪婪匹配。

- 弹性下界 minLen 参数，表示最小匹配数据长度，缺省为 0 时表示不限定最小长度。
- 弹性上界 maxLen 参数，表示最大匹配数据长度，缺省为 0 时表示不限定最大长度。
- 弹性过滤器 filter 参数，表示数据匹配规则，是中括号引用表示的正则字符集，比如[\d]、[^\s]、[0-9A-B]等等。作为简化，当[正则字符集]的首尾没有空格符时，外部的括号可以省略。当 filter 参数为空或缺省时，表示不限定数据内容。

弹性过滤器对应的正则字符集，具体有如下两种形式：

① [xyz] 表示包含字符集，即匹配括号间的任何一个字符。例如，[0-9]表示匹配任一数字，[0-9a-zA-Z]表示匹配任一大小写字母或数字；

② [^xyz] 表示排除字符集,即匹配不在括号间的任何一个字符。例如，[^0-9a-zA-Z]匹配除字母及数字外的其它任何字符。

正则字符集支持各种正则转义字符：

\d 匹配一个数字字符。等同于 [0-9]；

\D 匹配一个非数字字符，等同于 [^0-9]；

\f 匹配一个换页符；

\n 匹配一个换行符；

\r 匹配一个回车符；

\t 匹配一个制表符（Tab 符）；

\p 匹配 CR/LF (等同于 \r\n)；

\s 匹配任何空白符，包含空格、制表符、回车换行等，等价于[\f\n\r\t\v]；

\S 匹配任何非空白区域，等价于[^ \f\n\r\t\v]；

\v 匹配一个垂直制表符；

\w 匹配任何包含下划线的词语，等价于[A-Za-z0-9_]；

\W 匹配任何非单词字符，等价于[^A-Za-z0-9_]；

\x 引导 16 进制数对应的字符，例如 “\x64”表示 0x64,即字符“d”

实际应用场景中,一条指令模板中可以包含多个模糊匹配模式(包括定长模糊匹配和弹性模糊匹配)。例如，以下这条指令匹配模板：

```
\[*]AB\[2]CD\[*]EF\[*(1,8,[\d])]GH\[3]XY\[*]
```

注意：指令匹配模板的首/尾两端的无弹性参数的弹性模糊匹配都是可以省略的，这是因为自动应答规则引擎在指令匹配过程中是基于流式数据的全局匹配,其首尾会自动隐式添加弹性模糊匹配，实现全局的模糊匹配。因此，上条模板可以简化为：

```
AB\[2]CD\[*]EF\[*(1,8,[\d])]GH\[3]XY
```

这条指令匹配模板采用了多种匹配模式，其中 AB CD EF GH XY 这些都是立即数匹配，\[2]和\[3]为定长模糊匹配,\[*]和\[*(1,8,[\d])]为弹性模糊匹配。

【版本要求】*弹性模糊匹配的弹性参数为新特性，要求调试助手软件升级到 V5.0.9 以上版本。*

7.3.5 指令匹配模板示例

在实际工程应用中，各种接口协议的指令帧格式，通常都包含内容固定不变的部分以及动态变化的部分。在设计指令匹配规则时，指令中固定不变的部分通常使用“立即数匹配”，而动态变化的部分则可使用各种模式匹配，诸如定长精确匹配、定长模糊匹配、弹性模糊匹配，或结合多种匹配方式灵活运用。也就是说，实际工程应用中的指令匹配模板通常都是混合模式的，同一个指令匹配模板可同时包含多种匹配模式。

例如，下面这一条 Modbus-RTU 的 04 号功能读寄存器的指令匹配模板：

```
\x01\x04\[2#addr]\[2#len]\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]
```

该指令匹配模板字段分解如下表：

序号	字段	匹配模式	描述
1	\x01	立即数匹配	匹配设备 ID
2	\x04	立即数匹配	匹配 Modbus 功能号
3	\[2#addr]	定长模糊匹配	匹配 2 字节的寄存器地址
4	\[2#len]	定长模糊匹配	匹配 2 字节的数据读取长度
5	\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]	定长精确匹配	匹配 2 字节的 CRC16 校验码

7.4 指令应答模板

指令应答模板用于生成自动应答的报文数据，是自动应答规则的两个基本组成元素之一。每个指令应答模板通常都是由若干个数据段构成，每个数据段按照其生成应答数据的方式不同，可以划分为立即数应答段以及非立即数应答段两种类型。

7.4.1 立即数应答

指令应答模板中的“立即数应答”字段，类似于指令匹配模板中的“立即数匹配”字段，其外在表现形式为不使用模式符\[]的数据段。不同的是，前者是用于生成应答数据，而后者是用于指令匹配比较。

立即数应答数据段，示例如下：

- ASCII： OK\r\n
- ASCII: \x4F\x4B\x0D\x0A
- HEX: 4F 4B 0D 0A

从外在特征来看，立即数应答字段只使用纯粹的 ASCII 码字符串或十六进制编码数据，而不借助模式符\[]，即不通过脚本语法规则来生成应答数据段。

7.4.2 非立即数应答

所谓“非立即数应答”字段，就是排除“立即数应答”字段外的所有其他应答字段，用于生成一个指定数据长度及内容的应答数据段。“非立即数应答”，也称作“模式应答”，这是因为其外在形式表现为使用模式符\[…]，引用语法规则来生成应答数据。其一般语法形式如下所示：

```
\[ exp_len: exp_value #comment]
```

其中，参数 exp_len 表示生成字段的数据字节长度，参数 exp_value 表示生成字段的数据内容，#comment 是注解，也就是注释性文字，可省略。值得注意的是，exp_len 或 exp_value 可以是常数、或者变量，或者逻辑运算式，或者是由多语句组成的具有返回值的 BLOCK 代码块。在语法形式上，类似于指令匹配模板中的定长精确匹配语法格式，都有两个参数段，但两者用途不同，一个用于生成应答数据，而另一个则用于指令匹配比较。

参数 exp_value，不管其表现形式是立即数常量、变量还是运算表达式的计算值或者是 BLOCK 代码块，其本身就具有一个默认的数据长度。比如，int 型数据内容默认长度为 4 字节，short 型数据内容默认 2 字节。再比如，字符串数据，其本身就包含长度信息，等等。如果这个默认长度跟前置的 exp_len 值不等，则最终输出的数据长度以 exp_len 值为准，如果 exp_len 小于数据内容的默认长度，则输出数据将被截尾，否则尾部以零补足。

参数 exp_len 可以省略，也就是不进行显式指定，进而简化为如下所示的形式：

```
\[ exp #comment]
```

简化后只包含一个表达式，这个形式类似于指令匹配模板中的定长模糊匹配。区别是，定长模糊匹配模式中的单个表达式是用于计算数据长度，而格式化应答字段中的单个表达式是用于计算生成应答数据内容。最终，单表达式模式应答字段的输出长度为 exp_value 解析值的默认长度，比如 int 类型整数默认 4 字节、short 类型默认 2 字节，等等。

非立即数应答数据段，若干示例如下：

1) \[4:genAutoID()#生成 4 字节的 32 位流水自增 ID]
2) \[8:gets(0) #从源指令帧的偏移地址 0 开始截取 8 个字节数据]
3) \[2: getushort(2)*2 #从请求指令帧偏移地址 2 处取一个 16 位整数后乘 2]
4) \[2:random() #返回一个 32 位随机数，并截取 2 个有效字节]
5) \[random("ABCDE",0xF1F2,1024, 'x') #从参数列表中随机返回一个数据]
6) \[{int i=getint(0);return i;}:{if(getbyte(5)) return 1;else return 0;} #数据长度和内容都使用 BLOCK 代码块]

注：自动应答规则的指令模板可以嵌入类 C 语言语法的脚本代码，方便用户灵活地编写

指令模板，实现强大的自动应答功能。详细的语法规则参考第六章内容。

7.4.3 指令应答模板示例

实际应用场景中，一个典型的指令应答模板，通常都会混合多个立即数应答数据段和非立即数应答数据段。这里，仍然用 Modbus-RTU 读寄存器数据的自动应答规则为例：

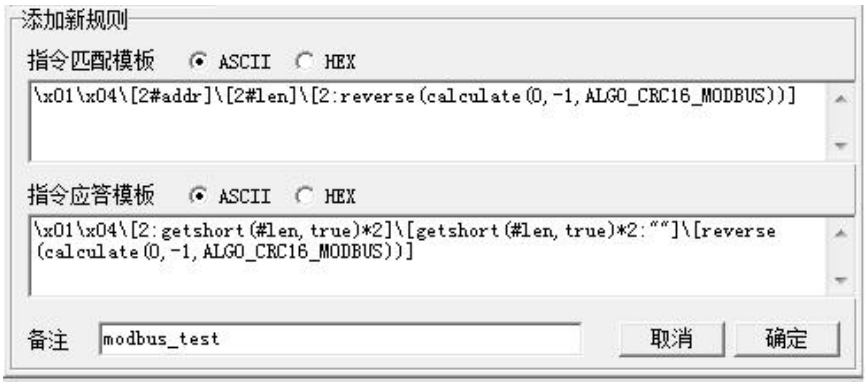


图 7-11 Modbus-RTU 的 04 号指令自动应答

这里重点关注指令应答模板数据，其各字段分解如下表：

序号	字段	应答模式	描述
1	\x01	立即数应答	设备 ID
2	\x04	立即数应答	功能号
3	\[2:getshort(#len,true)*2]	非立即数应答	2 个字节长度的数据，表示读取的寄存器数据的字节长度。其中，getshort(#len,true)表示从请求指令中读取一个位于#len 注解字段的 16 位整数。源指令中的长度是 16 位寄存器的长度，换算成字节长度要乘 2。
4	\[getshort(#len,true)*2:""]	非立即数应答	生成长度为 getshort(#len,true)*2 的空数据
5	\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]	非立即数应答	计算生成 2 个字节的 CRC16 校验码。函数 calculate 用于校验位计算，reverse 用于高低字节交换,以符合 Modbus-RTU 协议规范。

7.5 应答规则设计实例

7.5.1 实现指令帧 ECHO 功能

指令帧 ECHO 功能，就是收到任何指令帧数据，都原样发送回去。可按如下方式建立自动应答规则：

✧ 指令匹配模板<ASCII>： \[*]

✧ 指令应答模板<ASCII>： \[*]

指令匹配模板使用\[*]，即弹性模糊匹配模式，不管调试助手收到任何报文数据，都能成功匹配。

指令应答模板使用\[*]，表示应答数据直接使用所匹配成功的整个源指令报文。

注意，通配符\[*]在指令匹配模板中是指弹性模糊匹配，而在应答模板中出现时，实际上是\[gets(0)]的语法糖。内建系统函数 gets，用于从源指令报文的指定偏移量开始复制数据直至报文尾部，这里参数 0 表示从头开始复制源指令报文，不指定复制的长度就是默认复制整个指令帧数据。

7.5.2 模拟 AT 指令注册网络

✧ 命令格式<ASCII>：AT+CREG=mode<CR>

✧ 命令返回<ASCII>：OK<CR> 或者 ERROR<CR>

AT 命令中的 mode 的值共有三个选项,分别是 0 or 1 or 2,如果指令操作成功将返回 OK<CR>，否则返回 ERROR<CR>

下面根据此案例来建立应答规则。由于指令模拟无法确定 AT 指令是否执行成功，只假定参数正确便能执行成功。设计应答规则如下：

✧ 指令匹配模板<ASCII>：AT+CREG=\[*(1,1,\d) #mode]\r

指令匹配模板字段分解如下：

序号	字段	匹配模式	描述
1	AT+CREG=	立即数匹配	匹配固定的指令头
2	\[*(1,1,\d) #mode]	弹性模糊匹配	正则匹配一个 1 个数字符，字段注解名为#model
3	\r	立即数匹配	匹配一个回车符

✧ 指令应答模板<ASCII>：

```
\[{ int mode= atoi(gets(#mode));
    return (mode>=0 && mode<=2)?"OK":"ERROR";
  }]\r
```

以上指令应答模板中，gets(#mode)表示从接收到的指令数据中读取注解字段#mode 的数据内容，然后通过 atoi 函数将字符串类型转换成整型数据，再进而判断，如果该 mode 值在 0~2 范围内则返回 OK，否则返回 ERROR。

启用此条自动应答规则后，调试助手如果收到 ASCII 码指令 AT+CREG=1<CR>，则自动生成 ASCII 码应答数据 OK，如果收到 ASCII 码指令 AT+CREG=3<CR>，则自动生成 ASCII 码应答数据 ERROR。

7.5.3 模拟 AT 指令检查网络信号强度

- 命令格式<ASCII>：AT+CSQ<CR>
- 命令返回<ASCII>：+CSQ: rssi,ber<CR>

其中，rssi 为信号强度值，应在 0 到 31 之间，ber 为误码率,值在 0 到 99 之间。该案例的自动应答规则，可以按如下方式实现：

✧ 指令匹配模板： AT+CSQ\r

✧ 指令应答模板：+CSQ: \[IntToStr(random(31))],0\r

或者: +CSQ: \[{int rssi= random(31);echo("%d",rssi);}],0\r

当调试助手收到 AT+CSQ<CR>时，指令匹配成功，并自动发送命令应答数据。本例中，应答指令中的信号强度值 rssi 通过系统函数 random（31）取 0~31 的随机数，误码率 ber 直接固定为 0。由于 AT 指令是文本命令，所以随机生成的信号强度值须通过内建系统函数 IntToStr 将整型数据转换成字符串类型，或者通过 echo 函数格式化成字符串类型来作为 BLOCK 代码块的返回值。

7.5.4 模拟 AT 指令交互读写通信模块参数

- AT 写指令（配置通信模块的短信模式）：
  - AT 命令格式<ASCII>： AT+CMGF=<mode><CR>
  - 命令返回<ASCII>：+CMGF: OK<CR>
- AT 读指令（读取通信模块的短信模式）：
  - AT 命令格式<ASCII>：AT+CMGF?<CR>
  - 命令返回<ASCII>：+CMGF: <mode><CR>OK<CR>

要求模拟实现配置的短信模式能够保存在内存中，等收到读取指令时，自动按上一次成功配置的数据来进行应答。这就要求分别实现读/写两条应答规则，并且通过全局变量保存中间数据（短信模式），实现多条之间的上下文传参。

- 写指令应答规则：
  - 指令请求模板 AT+CMGF=\[* #mode]\r
  - 指令应答模板 +CMGF: OK\r\[{global["sms_mode"]=gets(#mode);}]
- 读指令应答规则：
  - 指令请求模板 AT+CMGF?\r
  - 指令应答模板 +CMGF:\[global["sms_mode"]]\rOK\r

说明：在 AT 写指令应答规则中，将匹配到的 mode 值保存到全局变量 sms_mode 中；然后在 AT 读指令应答规则中，将全局变量 sms_mode 作为模式值返回。

7.5.5 模拟 Modbus-RTU 读取寄存器数据

✧ 命令请求格式：1 字节设备 ID |1 字节功能号|2 字节寄存器地址|2 字节寄存器数量|2 字节 CRC 校验

✧ 命令返回格式：1 字节设备 ID |1 字节功能号|2 字节读取数据长度|寄存器数据内容|2 字节 CRC 校验

这里假定设备 ID 为 1，功能号为 4，若要求模拟 Modbus-RTU 读取寄存器数据时，自动返回随机的寄存器数据。该案例的自动应答规则，可以按如下方式实现：

✧ 指令匹配模板：

\x01\x04\[2#addr]\[2#quantity]\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]

➢ 指令匹配模板字段分解如下：

序号	字段内容	匹配模式	描述
1	\x01	立即数匹配	精确匹配 1 个字节设备 ID
2	\x04	立即数匹配	精确匹配 1 个字节设备 ID
3	\[2#addr]	定长模糊匹配	模糊匹配 2 个字节的寄存器地址，并设置字段注解#addr
4	\[2#quantity]	定长模糊匹配	模糊匹配 2 个字节的寄存器数量，并设置字段注解#quantity
5	\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]	定长精确匹配	精确匹配 2 字节长度的 CRC16 校验码

✧ 指令应答模板：

\x01\x04\[2:getshort(#quantity)*2]\[getshort(#quantity)*2:""]\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]

其中，\x01\x04 为立即数应答字段，\[2:getshort(#quantity)*2]为 2 字节的数据长度，其中 getshort(#quantity)表示从源指令中获取寄存器数量，由于是 16 位寄存器，所以要乘以 2 才能表示数据的字节长度；\[getshort(#quantity)*2:""]表示生成长度为 getshort(#quantity)*2 的空数据；\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS)]表示生成 2 字节的 CRC16 校验码，其中 calculate 用于计算校验位，reverse 函数用于逆转数据的高低字节顺序,以满足指令协议要求。

7.5.6 模拟 Modbus-TCP 交互读写单个寄存器

本实例要求实现两组指令模板：第一组是写寄存器指令，写入的数据保存在全局变量中；第二组是读寄存器指令，将保存在全局变量中的寄存器模拟数据读出来返回给用户。

➢ 写单个寄存器的指令匹配模板：

序号	字段内容	匹配模式	字段描述
1	\[2 #syncid]	定长模糊匹配	匹配 2 字节指令流水 ID
2	\x00\x00	立即数匹配	ModbusTCP 协议标识位
3	\x00\x06	立即数匹配	匹配接下来数据的长度
4	\[1 #deviceID]	定长模糊匹配	1 字节设备 ID
5	\x06	立即数匹配	功能码 06(表示写单个寄存器)
6	\[2 #RegAddr]	定长模糊匹配	2 字节的寄存器地址
7	\[2 #RegValue]	定长模糊匹配	2 字节的寄存器内容

➢ 写单个寄存器的指令应答模板：

序号	字段内容	匹配模式	字段描述
1	\[*]	模式应答	原样复制整个请求指令作为应答数据

2	\[{
	String regName="reg"+getshort(#RegAddr,true);
	global[regName]=gets(#RegValue);
		模式应答	BLOCK 代码块，无应答数据产生。实现将写入寄存器的数据保存到全局变量中。

	}]		

➢ 读单个寄存器的指令匹配模板：

序号	字段内容	匹配模式	字段描述
1	\[2 #syncid]	定长模糊匹配	2 字节指令流水 ID
2	\x00\x00	立即数匹配	ModbusTCP 协议标识位
3	\x00\x06	立即数匹配	负载数据长度(接下来数据的长度)
4	\[1 #deviceID]	定长模糊匹配	1 字节设备 ID
5	\x04	立即数匹配	功能码 04（表示读输入寄存器）
6	\[2 #RegAddr]	定长模糊匹配	2 字节的寄存器地址
7	\x00\x02	立即数匹配	数据读取长度（16 位寄存器 2 字节）

➢ 读单个寄存器的指令应答模板：

序号	字段内容	匹配模式	字段描述
1	\[2:gets(#syncid)]	模式应答	原样复制请求指令中流水 ID
2	\x00\x00	立即数应答	ModbusTCP 协议标识位
3	\x00\x05	立即数应答	负载数据长度(接下来数据的长度)
4	\[1:gets(#deviceID)]	模式应答	1 字节设备 ID
5	\x04	立即数应答	功能码 04(表示读输入寄存器)
6	\x02	立即数应答	读取的字节数（单个寄存器长 2 字节）
7	\[2:{ String regName="reg"+getshort(#RegAddr,true); return global[regName]; }]	模式应答	取自全局变量中保存的寄存器值

7.5.7 模拟部标 808 协议设备鉴权指令

指令匹配模板：

序号	字段内容	匹配模式	字段描述
1	\x7e	立即数匹配	1 字节指令头标识位
2	\x01\x02	立即数匹配	2 字节消息 ID
3	\x00\x06	立即数匹配	2 字节消息属性
4	\[6 #mobile]	定长模糊匹配	6 字节手机号 BCD 码
5	\[2 #syncid]	定长模糊匹配	2 字节指令流水 ID
6	\[*]	弹性模糊匹配	匹配鉴权码信息
7	\[1:calculate(1,-1,ALGO_XOR)]	定长精确匹配	1 字节异或校验位
8	\x7e	立即数匹配	1 字节指令尾标识位

指令应答模板（鉴权通用应答）

序号	字段内容	匹配模式	字段描述
1	\x7e	立即数应答	1 字节指令头标识位
2	\x80\x01	立即数应答	2 字节消息 ID
3	\x00\x05	立即数应答	2 字节消息属性

4	\[6:gets(#mobile)]	立即数应答	6 字节手机号 BCD 码
5	\[2:genAutoID()]	非立即数应答	2 字节当前指令流水 ID
6	\[2:gets(#syncid)]	非立即数应答	2 字节请求指令流水 ID，此流水 ID 要跟原请求指令的流水 ID 号一致
7	\[2:gets(1,2)]	非立即数应答	复制原指令的消息 ID
8	\x00	立即数应答	1 字节错误码
9	\[1:calculate(1,-1,ALGO_XOR)]	非立即数应答	1 字节异或校验位
109	\x7e	立即数应答	1 字节指令尾标识位

7.5.8 指令模板 debug 调试方法示例

设计指令模板时，如果无法正确地按设计意图实现指令匹配或者无法生成正确的应答数据段，可以在指令模板中插入调试打印语句来检查指令模板中各个变量字段的实际数据值，通过逐一排查比对的方式来定位错误故障点。

不管是指令匹配模板，还是指令应答模板，都可以插入形如\[{…}]的 BLOCK 代码块，只要这个 BLOCK 代码块内没有调用 return 语句或者 echo/echob 函数来生成返回值，那么这个 BLOCK 代码块就不会对指令匹配或者指令应答起任何作用。但是，我们可以在 BLOCK 代码块中插入 printf 函数，将关键的匹配字段或者应答字段对应的数据或变量打印出来，尤其要检查指令匹配模板中定长精确匹配字段的对应的数据，通过比对来排查错误。

例如，假定有一指令模板如下所示：

- 指令匹配模板　\xFF\[4 #PayLoad]\[1: calculate(0,-1,ALGO_ACC))]
- 指令应答模板　\xFF\x00

通信测试实验中，当自动应答规则引擎接收到十六进制数据 FF 00 00 00 00 00 时，如果没有发生应答动作。那么我们就要对指令匹配模板进行检查，主要方法是将“定长精确匹配”字段的值打印出来进行比对，并且必须在匹配字段前进行打印。这是因为，如果前面的字段匹配失败则后面的模式代码都不会被执行到。因此，可以修改指令匹配模板（插入 BLOCK 调试代码块）如下：

\xFF\[4#PayLoad]\[{printf("checksum==%02x",calculate(0,-1,ALGO_ACC));}]\[1:calculate(0,-1,ALGO_ACC))]

然后再次接收数据 FF 00 00 00 00 00 时，调试助手打印信息 checksum==FF，而实际接收到的数据帧的校验位是 00，从而可以判定问题出在校验位的匹配上，再进一步检查用于校验位计算匹配的代码，最终发现是校验数据的偏移地址错误，应将指令匹配模板修改为：

xFF\[4 #PayLoad]\[1: calculate(1,-1,ALGO_ACC))]

通过插入 BLOCK 级的调试代码，有利于缩小故障点的排查范围，帮助用户快速地定位问题和解决问题。

第八章 常见问题

1. 安全软件误报毒问题怎么解决？

答：由于软件启动时会访问网络以检测版本信息，导致部分安全软件有可能出现误报毒的问题，特此声明本软件绝无绑定任何恶意程序或木马病毒，如在运行时出现误报，请点击添加信任（添加到白名单）即可！

2. 接收保存到文件时，怎么去除时间戳等相关信息，只保留接收的数据内容？

答：接收保存到文件时，会弹出对话框要求用户选择文件保存路径以及文件类型。文件类型有日志文件和数据文件两种。其中，日志文件类型是默认选项，会保存接收数据的时间戳等附加信息，并且还会保存发送记录信息；而选择数据文件类型则只保存接收的数据内容。

3. 网络调试助手开启 UDP 监听时，为什么在调试下位机时可能出现需要打开 WireShark 才能接收到下位机设备发送的数据?

答：常见的原因是下位机设备的 MAC 地址没有正确设置，导致上位机网卡直接屏蔽了下位机设备发来的数据，当用 wireshark 等工具打开监听的时候，相当于屏蔽了网卡对 mac 地址的过滤功能，数据就直接进来了。

4. 监听时状态栏提示地址绑定错误“The specified address is already in use”，怎么解决？

答：导致该错误的原因是监听的端口被其它程序占用，可以换个端口监听，或者关闭占用该端口的应用程序。

5. 如何查看某端口被哪个程序占用？

答：打开 DOS 命令行窗口，执行命令 netstat –ano，然后在返回的列表信息中找到占用目标端口进程的 PID 号码，比如 1956，接着继续在在 DOS 命令行窗口执行命令 tasklist |findstr "1956"，这里的数字就是前面查找到的 PID 号码，这样就可以得到目标 PID 对应的应用程序名称。

6. TCP 客户端无法连接到 TCP 服务器的问题怎么解决？

答：这个问题比较泛也比较常见，可能的原因也是多方面的，需要仔细地逐一排查。

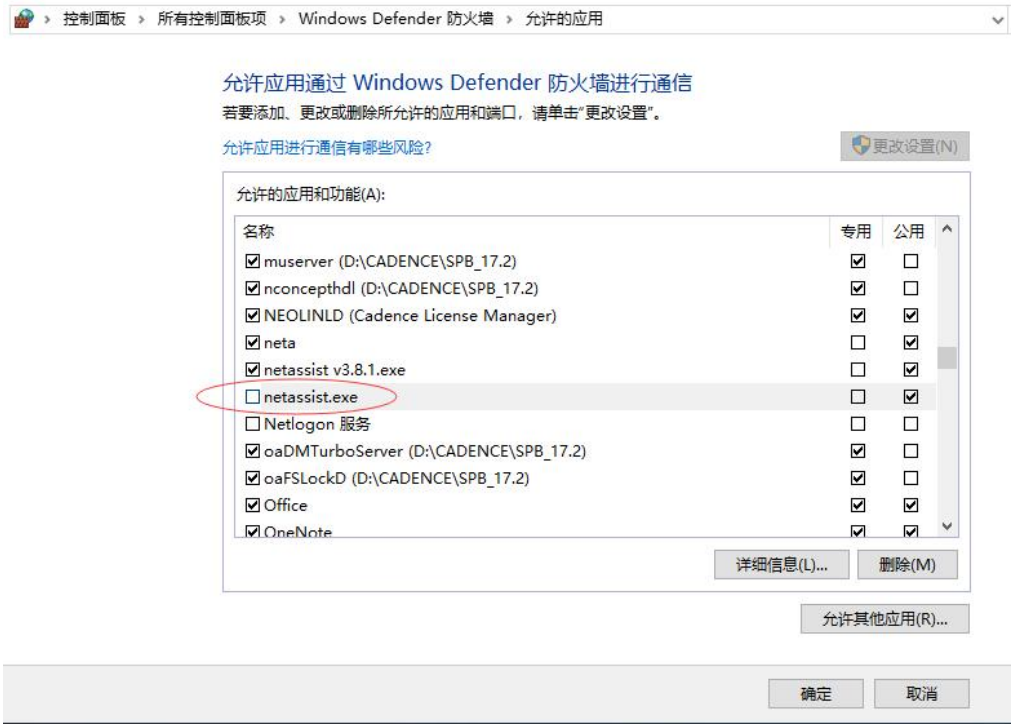
- *确认 TCP 客户端连接服务器时，所填写的服务器地址及端口是否正确。*
- *确认服务器是否正确选择监听的网络适配器地址。*
- *确认服务器是否已打开/监听目标端口。*
- *确认客户端与服务器之间的网络连接是否在物理上连通。方法是在客户端使用 ping 命令测试服务器地址，是否有数据返显，如果无法 ping 通则需要检查网络拓补状况是否有问题。*
- *查看服务器的监听端口是否被服务器拦截。*

- *如果服务器与客户端都是在局域网内，需要确保两者在同一网段。*
- *确认服务端或者客户端程序是否被系统防火墙拦截。*

7. 网络调试助手在 Win7 系统下运行正常，但换到 Win10 系统就报 1035 错误是怎么回事？

答：Win10 下大都是防火墙问题。你可能没注意到在第一次启动软件时，应该会有防火墙弹窗提示（如下图所示）。如果手快，直接点了取消按钮，而没选择允许访问网络，那么网络调试助手就进了防火墙的黑名单，再无法使用网络通信功能。

解决方法是关闭防火墙，或在防火墙允许的应用程序列表中找到网络调试助手并将它设置为允许状态(勾选)。



7. 网络调试助手如何自启动到 TCP Server 模式下并自动进入监听状态？

答：通过命令行参数启动(可参考 6.7 小节)。例如：netassist.exe -ts -lh 192.168.0.8:8080 该命令实现启动网络调试助手软件，并自动选择 TcpServer 模式，同时自动打开监听目标地址 192.168.0.8:8080。如果要实现开机自启动监听，可以将此命令行转成文件快捷方式，并加入到操作系统的启动项中即可。

8. 调试助手接收数据出现乱码怎么处理？

答：如果仅仅是接收内容中所包含的中文出现乱码，可以在接收窗口的右键菜单中，切换字符集编码为 ANSI 或 UTI8 后再重新尝试。

9. 调试助手软件界面中文乱码怎么解决？

答：在 Windows 控制面板中，找到区域设置进行恢复，具体操作如下图所示：进入控制面板 --> 区域和语言 --> 管理 --> 非 Unicode 程序的语言，更改系统区域语言设置为:中文(简体,中国)，并将 Unicode 勾选取消即可。