第 5 章 CHAPTER 5 设备监控进程设计

学习目标

- 熟悉监控进程使用驱动程序与设备交互的原理。
- 理解监控边缘服务器与监控进程的通信机制。
- 熟悉监控进程内各模块间的通信机制。

技能目标

- 进一步熟悉软件工程需求和架构分析方法，用例图和部署图。
- 掌握面向对象设计方法：类图、构件图。
- 良好的文档编写规范：源程序文档化。


视频讲解

设备监控中心(DMC)负责管理设备监控进程(DMP)，并与之通信。具体对设备的监控，全部交由 DMP 处理；这样的松散结构有助于系统的稳定和扩展。由于设备的多样性，也不可能使用一个监控程序去与多种异构设备交互。

所以设计的监控进程是专门为某类设备服务的程序。这些设备内部使用相同的协议工作。因此，指定 DMP 使用一个设备监控驱动中间件来完成设备系统与监控平台之间的数据转换工作(中间件的开发见第 10 章)，只有这样，监控平台才能使用统一的协议去监控所有不同类型的设备系统。

5.1 设备监控程序的功能设计

DMP 的主要业务功能，是负责接收设备系统发来的各类信息，并报告给 DMC；同时接收 DMC 发来的监控指令，通过设备监控驱动模块，把指令翻译成设备系统的本地指令，再传递给设备系统，从而控制设备工作。

图 5.1 是监控进程子系统的用例图。

5.1.1 建立通信对象

根据 DMC 传递的监控驱动程序文件名和参数，可从中获得通信类型，并根据参数建立通信对象。TCP/IP 通信的参数，由 DMC 传递过来。但对串口通信，也可在 DMP 内部配置保存，下次启动时，使用这些串口配置来设置通信参数。串口通信参数保存在文件 **comport**. xml 中，放在 iotmonitorset×××文件夹中，“×××”是 DMP 的监控进程编号 DMID。

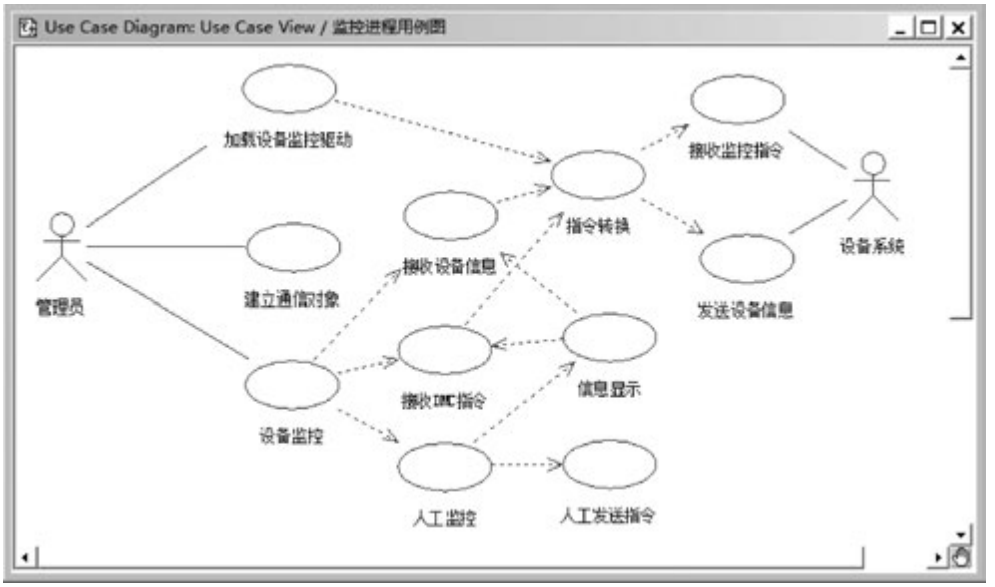

图 5.1 DMP 用例图

5.1.2 加载设备驱动程序

根据设备驱动程序的文件名动态加载驱动程序,并传递通信对象给驱动程序。

提示:驱动程序内部,并没有构建通信对象的方法,需要 DMP 先建立再传递给它。

监控驱动程序的核心作用是充当协议翻译器。监控平台是使用标准的三个协议组织数据,是“官方”语言;设备系统使用企业内部自建的协议组织数据,是“方言”。因此,它们之间的通信需要一个翻译器。

图 5.2 展示了监控驱动程序在 DMP 中的作用。

图 5.2 DMD 的作用

5.1.3 设备监控

在设备监控程序加载后,DMP 进入正常的监控状态。主要对接收设备信息、DMC 监控指令做出处理反应。所以监控功能的实现,嵌入在通信数据的处理方法中。

为了调试方便,在监控进程中设计了 UI,主要用于设备信息的显示,以及通信数据内容的显示;提供 UI 来修改设备的一些描述信息,方便用户重新设置设备。例如,把“第一个开关”修改成“客厅灯开关”。

5.2　设备监控程序的详细设计与实现

监控进程项目 **DeviceMonitor** 程序设计的业务对象，在 VS 中可以查看设计的类，如图 5.3 和图 5.4 所示。

监控进程没有设计过多的业务类，这里介绍两个通信参数配置类（CommParameter、ServerPortParameter）以及一个程序配置类（Setting）。它们使用通用监控程序模块 IotProtocolLib 内定义的对象进行工作。

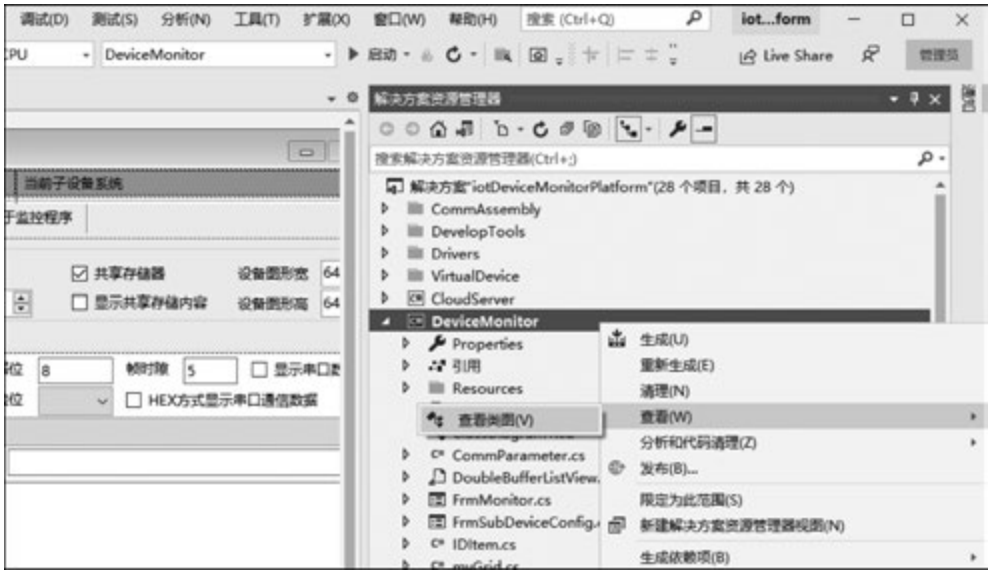

图 5.3　使用 VS 的类图查看功能

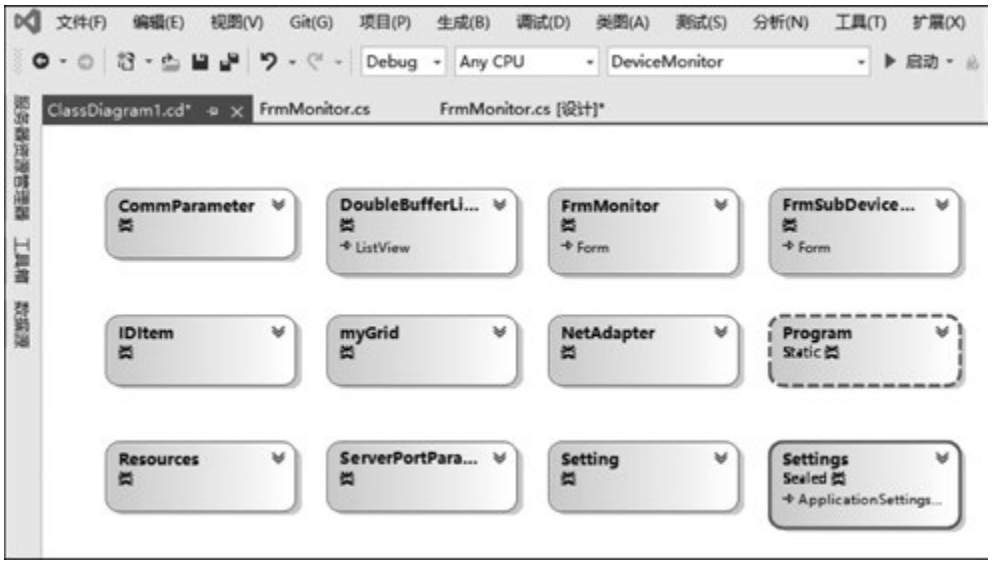

图 5.4　VS 展示的设计类图

CommParameter 类：对串口通信的监控驱动 DMD,需要保存串口通信配置参数。

ServerPortParameter 类：对 TCP/IP 通信的监控驱动 DMD,需要保存 TCP 通信配置参数。

以上两个类很简单,请读者自行阅读。

Setting 类：几乎所有的应用程序都会有配置文件,用于永久保存应用的一些参数。这个类大多设计为**单实例模式**。其结构如下。

```
public class Setting                                    //配置文件
{
    private string FileName;                            //配置文件名
    private static Setting setting = null;              //静态私有变量
    Setting()                                           //私有构造函数:单实例模式
    {
        FileName = AppDomain.CurrentDomain.BaseDirectory + "\\Setting.xml";
        …
    }
    public static Setting LoadSettings(string _FileName)  //获取对象的公共静态方法
    {
        if (setting != null  && setting.FileName == _FileName)  return setting;
        if (!File.Exists(_FileName))                    //配置文件不存在
        {
            setting = new Setting();                    //类的内部代码可以调用私有构造方法
            setting.FileName = _FileName;
            return setting;
        }
        //配置文件存在
        XmlSerializer xmlSerializer = new XmlSerializer(typeof(Setting));
        try {
            using (FileStream stream = File.Open(_FileName, FileMode.Open))
            {
                setting = xmlSerializer.Deserialize(stream) as Setting;  //XML 反序列化
                setting.FileName = _FileName;
                return setting;
            }
        }
        catch {
            setting = new Setting();
            setting.FileName = _FileName;
            return setting;
        }
    }
    public void Save()                                  //保存信息
    {
        using (FileStream stream = File.Create(FileName))
        {
            XmlSerializer ser = new XmlSerializer(typeof(Setting));      //XML 序列化
            ser.Serialize(stream, this);
            stream.Close();
        }
    }
}
```

单实例模式的特点就是,无法直接实例化对象,只能通过静态方法获得。在程序任何地

方获取该类对象，得到的都是同一个对象，保证了数据的一致性。

图 5.5 是主程序 **FormMonitor. cs** 启动的主流程图。

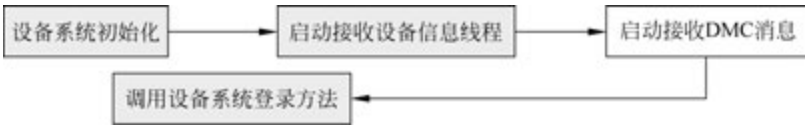

图 5.5　DMP 启动流程图

程序启动方法的相应代码设计如下。

```
private void FormMonitor_Load(object sender, EventArgs e)
{
    InitLoad();
    …
    StartWatchDevice();
    StartReceiveNotify();
    monitorSystemmethod.Login(new object[] { this.monitorSystem });
}
```

核心代码在于设备系统的初始化 InitLoad()。其流程图见图 5.6。

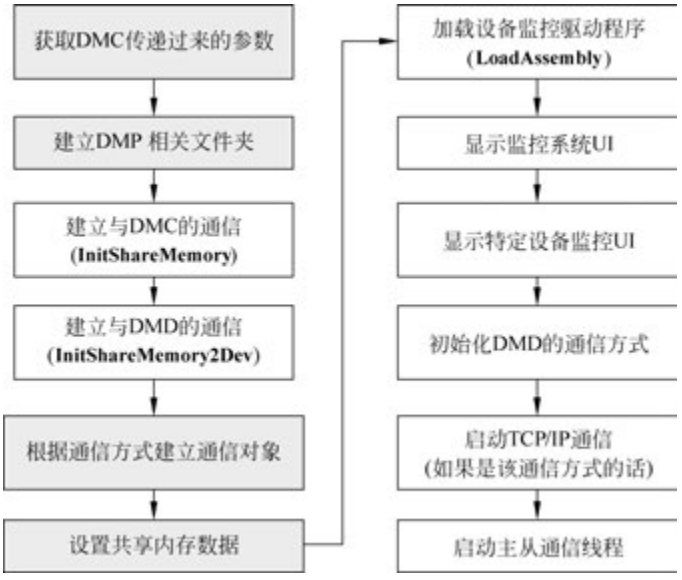

图 5.6　DMP 系统初始化流程图

启动代码初始化的主要部分详见源码示例二维码，后续对其中调用的主要方法做设计介绍。


源码示例

当初始化完毕，DMP 基本准备就绪，剩下的工作就是处理通信事件。

5.2.1　建立 DMP 与 DMC 的通信

在复杂应用程序中，多进程、多线程编程是非常普遍的。当几个线程访问同一资源（文件、内存等）时，为了保证操作的原子性，避免可能的冲突，需要设置信号量或互斥来保证线程间的同步或异步机制。本程序使用互斥机制来访问消息队列。

InitShareMemory()方法是在 DMC 与 DMP 之间建立共享内存和消息队列，其中使用

了互斥机制。相关代码如下。

```
DMPInfo dmp;                                    //映射到共享内存的对象指针
ShareMemory dmpshareMemory;                     //共享内存对象:用于进程间交换数据 DMC<--->DMP
Mutex mutex;
Mutex mutexshare;                               //用于进程间同步
void InitShareMemory(ref Mutex mutex)           //共享内存对象:DMC<--->DMP
{
    if (!CreateMutex(DMPInfo.mutexname + "Notify" + AppId.ToString(), ref mutex))
        System.Environment.Exit(0);             //DMP 给 DMD 发通知的同步互斥
    if (!CreateMutex(DMPInfo.mutexname + "Message" + AppId.ToString(), ref mutexshare))
        System.Environment.Exit(0);             //DMD 给 DMP 发消息的同步互斥
    dmpshareMemory = new ShareMemory(mutex, DMPInfo.SHFLAG, mutexshare);
    DMPInfo aChannel = new DMPInfo();
    dmpshareMemory.Init(DMPInfo.sharememoryfile + AppId.ToString(), Marshal.SizeOf(aChannel));
                                                //根据 DMID 建立共享内存
    if (dmpshareMemory.m_ok)
    {                                           //获取服务器传来的参数
        dmp = (DMPInfo)dmpshareMemory.ShareMemoryToObject(aChannel);
    }
}
```

两次调用 CreateMutex()方法,创建了两个互斥对象,用于对两个消息队列的访问控制。

5.2.2 建立 DMP 与设备监控驱动模块 DMD 之间的通信

同样地,在 DMP 与设备监控驱动(DMD)模块间也需要建立消息队列与共享内存,方便进行数据传递和交换。代码完全类似,只是互斥和消息队列的名称不一样。

```
DMPInfo dmp2dev;                                //映射到共享内存的对象指针
ShareMemory dmpshareMemory2dev;                 //DMM<--->Device
Mutex mutex2dev;
void InitShareMemory2Dev(ref Mutex mutex) //建立消息队列和共享内存对象:DMP<--->DMD
{
    if (!CreateMutex(DMPInfo.mutexname + "DeviceNotify" + AppId.ToString(), ref mutex))
                System.Environment.Exit(0);
     if (!CreateMutex(DMPInfo.mutexname + "DeviceMessage" + AppId.ToString(), ref
mutexshare))
    System.Environment.Exit(0);
    dmpshareMemory2dev = new ShareMemory(mutex, DMPInfo.SHFLAG, mutexshare);
    DMPInfo aChannel = new DMPInfo();
    dmpshareMemory2dev.Init(DMPInfo.sharememoryfile + "Device" + AppId.ToString(),
            Marshal.SizeOf(aChannel));
    if (dmpshareMemory2dev.m_ok)
    {
    dmp2dev = (DMPInfo)dmpshareMemory2dev.ShareMemoryToObject(aChannel);
    }
}
```

5.2.3 动态加载设备监控驱动程序

在建立了通信对象后,启动方法需要动态加载监控驱动对象,这是系统的核心方法。它负责创建具体的监控系统 MonitorSystem、获取监控接口方法 monitorSystemmethod,并从配置文件获取上次操控监控系统的状态,便于在 UI 上呈现监控信息。流程相对简单,请读

者自行绘制流程图。核心代码如下，做了较详细的讲解。

```
Assembly asm = null;                                    //程序集对象
MonitorSystemBase monitorSystem;                        //监控系统对象
IMonitorSystemMethod monitorSystemmethod;               //监控系统监控接口
string classType = "";
private bool LoadAssembly(string dll)                   //动态加载厂家的智能监控驱动程序集
{
    string fn = Application.StartupPath + "\\Drivers\\" +//驱动程序默认放在 Drivers 文件夹中
            Path.GetFileNameWithoutExtension(dll) + ".dll";
    if (!File.Exists(fn)) return false;
    try
    {
        if (asm == null)
            asm = Assembly.LoadFrom(fn);                //动态加载程序集
    }
    catch { return false; }
    if (asm == null) return false;
    classType = System.IO.Path.GetFileNameWithoutExtension(dll) + ".MonitorSystem";
        //类名必须为 XXX.monitorSystem
    type smarthome = asm.GetType(classType);            //获取指定对象类型
    if (smarthome == null) Application.Exit();          //连接库不是合法驱动
    string shfn = AppDataDir + "\\monitorSystem.json";//最新版本改为 JSON 结构存储数据
    object obj = Activator.CreateInstance(smarthome, new object[] { shfn });
                                                        //创建 DMD 对象，构造函数有两个参数
    monitorSystem = (IMonitorSystemBase)obj;            //转换为具体的监控系统对象
    monitorSystemmethod = (IMonitorSystemMethod)obj;    //转换为监控方法对象
    if (smarthome == null) Application.Exit();          //连接库不是合法驱动
    if (!File.Exists(shfn))  monitorSystem.SaveToFile();   //监控系统文件不存在，立即建立
    //以下获取最后一次操控的设备系统 DeviceSystem 的对象，便于显示 UI
    currentDeviceSystem = GetDeviceSystem(monitorSystem, setting.lastDSID);
    currentSubDeviceSystem = GetSubDeviceSystem(monitorSystem, setting.lastDSID,
                                setting.lastSSID);
    currentDevice = GetDevice(monitorSystem, setting.lastDSID,
                                setting.lastSSID, setting.lastDID);
    return true;
}
```

务必掌握 Activator.CreateInstance 从程序集中动态创建对象的使用方法。

5.2.4　显示特定设备系统信息

如果监控进程以黑匣子方式在后台运行，完全可以不需要 UI，且效率会更高。但出于编辑修改监控系统参数的需要，以及可视化监控效果，全栈项目设计了较复杂的界面来展示监控状态，也方便进行调试，特别是在没有移动监控 APP 的情况下，在监控中心可直接进行设备操控。图 5.7 是设备系统信息的显示设计界面。图 5.8 是实际运行的一个 UI。

UI 设计使用了传统的 **WindowsForm** 设计，虽然业务逻辑与 UI 设计混在一起非主流提倡的设计方法，但运行的效率还是可圈可点的。由于设计时间较早，没有使用 WPF 等设计方法(参见第 7 章)。

具体 UI 设计的细节，包括控件名称、布局、事件响应代码等，请读者使用 Visual Studio 去打开项目参阅。

ShowMonitorSystem()方法用于显示监控系统特定设备系统的信息。它的主要流程如下。

图 5.7 DMP 的 UI 设计

在设备系统下拉框中显示所有的设备系统→在子设备系统下拉框中显示选择设备系统的所有子设备系统→在表格中显示选定子设备系统的所有具体设备的信息(见图 5.8)。

图 5.8 DMP 设备系统信息显示实例


源码示例

在如图 5.8 所示界面选择本 DMP 中的任何一个具体设备，显示其相关信息，详见源码示例二维码。

监控进程程序采用 WindowsForm 风格设计，其 UI 的代码较杂乱、繁多；后面章节，请

读者自行参阅，不再做过多介绍。

新版监控程序也增加了对设备操作过滤的功能，建立一个 devicepermission.json 文档保存是否允许指定设备与监控进程通信。通过单击“入网设置”按钮打开如图 5.9 所示 UI。

图 5.9　对设备进行过滤设置

单击“刷新”按钮获取该监控进程接入的所有设备信息，通过去掉勾选项，然后“保存”，则该设备发来的所有数据，由**监控驱动程序**决定是否处理该设备的数据；该对象被传递给了驱动程序：监控驱动程序的 **InitComm**()方法接受“允许设置”对象。

```
DevicePermission DPs;              //DevicePermission 在监控库 IotProtocolLib 中定义
public void InitComm(object[] commObjects)
{
    if (commObjects == null) return;
    if (commObjects[0] is SerialPort) comm = (SerialPort)commObjects[0];
    if(commObjects.Length > 4) DPs = (DevicePermission)commObjects[4];
    …
}
```

如果驱动程序不处理该设备发来的数据，相当于被排除在监控系统之外，尽管该设备仍然存在，并且可能在正常工作。

图 5.10 是 LoraDriver 驱动程序中对设备原始通信数据的处理。可以观察到，当查询到设备不允许通信时(使用了 FindPermission()方法)，直接忽略数据处理。

图 5.10　驱动程序对设备进行过滤操作

5.2.5 显示特定设备的监控信息

在图 5.8 的界面中选择一个具体设备后,在监控界面以图文方式显示该设备的所有子设备的信息,并提供实时监控操作的界面。

如图 5.11 所示是一个语音音乐播放器虚拟设备的所有子设备的监控 UI。

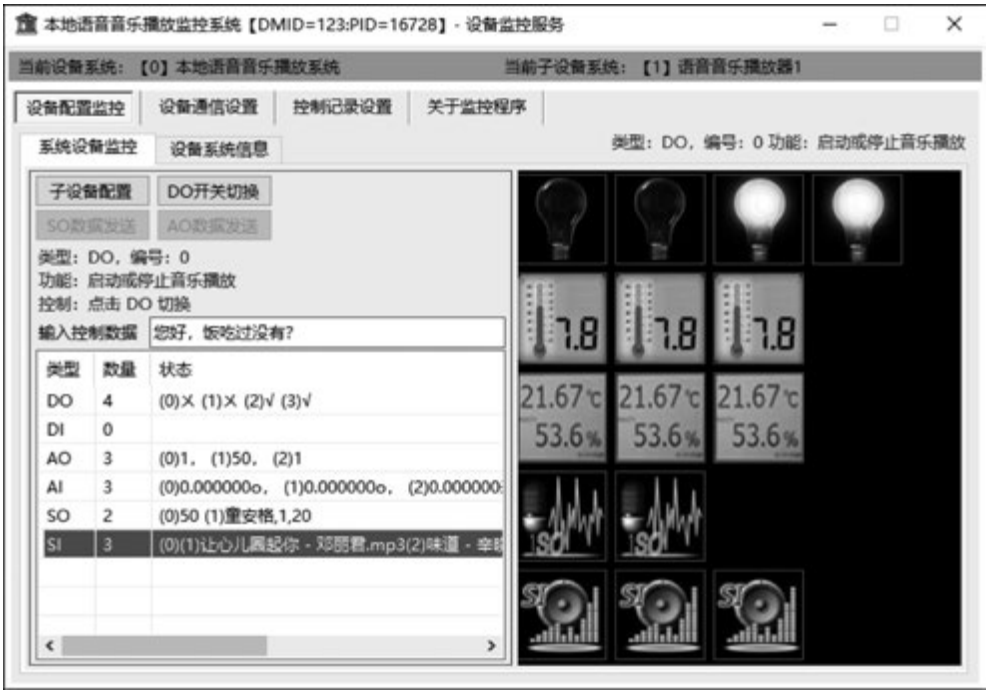

图 5.11 语音音乐播放器的监控界面实例

程序中的 LoadOneDevice()方法用于显示一个具体设备的监控信息。

```
void LoadOneDevice(Device device)                    //参数是设备对象
{
    lvSubDevice.Tag = device;        //技巧:用表格 lvSubDevice 的 Tag 属性保存目前选择的设备
    ShowOneDeviceInfo(device);
    ShowSubDeviceInfo(device);
    if (device != null)
        lbSubDevices.Text = string.Format("{0}【{1}】", device.DeviceName, device.DID);
    else
        lbSubDevices.Text = "";
}
```

ShowOneDeviceInfo 显示设备的基本信息。

ShowSubDeviceInfo 显示子设备的监控信息。代码结构如下。

```
void ShowSubDeviceInfo(Device hd)                    //显示子设备信息,参数是设备类型
{
    lvSubDevice.Items.Clear();                       //显示表格清空
    if (hd != null)
    {
        string[] newrow = { "00", "", "", "" };//显示 DO 子设备
```

```
            newrow[0] = "DO";
            newrow[1] = hd.DODevices.Count.ToString();
            newrow[2] = hd.GetState(DeviceType.DO);
            ListViewItem item = new ListViewItem(newrow);
            item.Tag = DeviceType.DO;
            lvSubDevice.Items.Add(item);
            //以下为 DI、AO、AI、SO、SI 子设备的显示
            …
        }
        ShowAllDevicesPicture(hd);                    //显示所有设备图形
    }
```

ShowAllDevicesPicture()方法用于在界面右边以图形方式显示子设备信息,并指定鼠标操作响应事件代码。

```
    void ShowAllDevicesPicture(Device hd)
    {
        if (hd == null)                               //没有选定设备,相应面板内容清空
    {
            plDO.Controls.Clear(); plDI.Controls.Clear();
            plAO.Controls.Clear(); plAI.Controls.Clear();
            plSO.Controls.Clear(); plSI.Controls.Clear();
            return;
        }
        //以下在不同面板中显示子设备的图形和鼠标事件相应代码
        DrawPicture(hd, DeviceType.DO);
        DrawPicture(hd, DeviceType.DI);
        DrawPicture(hd, DeviceType.AO);
        DrawPicture(hd, DeviceType.AI);
        DrawPicture(hd, DeviceType.SO);
        DrawPicture(hd, DeviceType.SI);
    }
```

DrawPicture()方法的代码较多,请自行阅读。这种动态设计控件界面和指定事件响应代码的编程技巧,有助于编写复杂的 UI。

5.2.6　初始化监控系统的通信

初始化了通信对象、加载和显示了监控系统的信息,DMD 并没有开始进行通信工作,必须调用监控系统接口的 **InitComm**()方法来实现。设计 **InitDeviceDriver**()方法用于初始化智能监控设备系统的通信方式(调用监控驱动的 **InitComm**()方法)。

```
    void InitDeviceDriver()          //根据 DMP 传递的参数,正确初始化智能监控设备系统的通信方式
    {
        try
        {
            if (monitorSystem.bServer)                //设备为服务器
                monitorSystemmethod.InitComm(new object[] { comm, tcpClient,
                        dmpshareMemory2dev, dmp2dev, DPs });
            else  //DMM 为服务端
                monitorSystemmethod.InitComm(new object[] { comm, Clients,
                        dmpshareMemory2dev, dmp2dev, DPs });
        }
        catch { }
    }
```

5.2.7 启动监控系统的通信

对于 TCP/IP 通信,驱动程序并不会直接与设备主动连接通信,需要启动程序(监控进程 DMP)完成该任务,DMD 只负责处理 DMP 发来的指令,或者把设备信息发给 DMP。StartListenOrConnect()方法承担该任务,它建立通信线程,收发 TCP/IP 数据,建立数据接收处理事件等工作。

```
private void StartListenOrConnect()
{
    if (smarthomechannel.bServer)              //设备系统为服务器
        StartConnection();
    else                                       //DMP 为服务端
        StartService(true);
}
```

StartConnection()和 StartService()方法的代码较多,请仔细阅读。有关 TCP/IP 通信的知识和编程能力,是计算机专业学生应该掌握的基本知识,且有实践能力。

5.2.8 启动监控系统的主从通信

如果是对传统设备的智能化升级改造,没有做到设备状态信息改变时,主动报告给 DMP 的功能(常见情况是其二次开发 SDK 没有实时传递状态信息的方法);为了较快获取其状态信息,DMP 可以主动发起请求指令,让设备发送状态信息给监控系统。如图 5.12 所示的"通信参数设置"界面提供了该选项。这是**迫不得已**的一种获取信息的方法,不建议使用。升级设备的嵌入式软件代码为最佳方案。

图 5.12 串口设置与主从通信设置

设计 **InitCommThread**()方法,它建立一个线程,在后台定时发送请求指令来实现该功能。

关于多线程编程的技术,项目中已经使用很多次了,希望读者能运用自如。

至此，DMP 的启动过程结束，监控系统进入正常的工作状态。

5.2.9　DMP 通信参数设置

为方便程序的调试，在 DMP 中设置了“通信参数设置”界面。图 5.13～图 5.15 是三种不同通信方式的设置界面和通信数据显示界面。

TCP 通信分为服务端和客户端两种情况。如图 5.13 所示的是设备系统作为服务端，DMP 作为客户端去主动连接设备系统。图 5.13 左边界面用于显示 DMP 作为通信服务端时连接的客户端的情况；允许多个设备系统同时连接到 DMP，也即一个 DMP 可以监控多个不同的设备系统，在实际的设备监控平台中具有现实意义。

图 5.13　TCP/IP 通信设置

图 5.14　共享内存通信无须设置

图 5.15 串口通信参数设置和通信数据显示

其中的相关代码较多,读者仔细阅读。

5.2.10 子设备参数修改

硬件厂商生产的设备,同一产品中各子设备的默认名称都是一样的。在用户购买后实际使用时,需要给它一个确定的名称,方便用户选择操控。在图 5.11 的界面中,选择表格中某类子设备类型对应的行,然后单击“子设备配置”按钮,打开子设备配置界面,如图 5.16 所示。

图 5.16 子设备参数修改界面

在此处提供了**黑白名单**的编辑。在设备通信底层进行了操控安全性设置。每当客户端程序发送控制指令时,DMP 都会在此过滤掉非法授权用户的操控。

功能描述、操作描述:可以修改为用户易于理解的文本;状态显示图片可以修改为实际设备状态描述逼真的图片,如音乐开关图片替换为喇叭图片。

小结:DMP 程序代码,根据 DMC 提供的参数,自动创建通信对象和加载监控驱动。为不同设备的接入提供了一个通用的监控程序。仍有很多设计需要改进完善。期待读者有更好的实现,只要满足系统三个核心协议即可。

5.3　不同通信方式的数据处理

DMP 接收 DMC 发来的指令，同时也接收与设备通信的通信对象发来的数据。两者的处理流程有所区别，以下介绍部分设计内容。

5.3.1　处理边缘服务器下达的指令

从 EdgeServer 发来的指令有多种方式，可能是定时任务自动发送的任务执行指令，也可能是智能监控出发的任务指令，也可能是用户从监控 APP 或程序甚至通过浏览器发来的指令。由于在平台上采用了统一的监控协议，发来的指令都是符合监控协议的 JSON 结构数据包，DMP 主程序使用同一个方法 **ProcessDMCInfo**()来处理，以下代码列出了几个主要指令的处理。

```
private void ProcessDMCInfo(JObject json)     //处理监控边缘服务器传来的通知指令
{
    if (json == null) return;
    if (setting.bShowCmd) txtMemory.Text += "------ DMC -->DMM ->DEVICE ----\r\n" +
json.ToString();                              //可视化显示指令内容
    string cmd = (string)json["cmd"];
    if (cmd == null) return;                  //没有命令键值对(词条)
    //逐一处理各种指令,参见 IoTProtoccolLib 项目
    if (cmd == IotMonitorProtocol.STARTAPP) //截获 STARTAPP、SHOWDMMUI 指令
    {  //通知设备监控程序退出,一般硬件设备不理会,虚拟设备可以退出
        json.RemoveAll();
        json.Add("cmd", IotMonitorProtocol.STARTAPP);  //DMM 结束命令
        NotifyDevice(json);
        MyDelayMs(100);
        Appover();
    }
    else if (cmd == IotMonitorProtocol.SHACTRL)      //控制设备工作的指令
    {  //给设备系统的设备发指令[appid=N][devid=M][subid=X][type=Y][act=K]
        NotifyDevice(json);                   //直接转发给设备驱动程序处理
    }
    else if (cmd == IotMonitorProtocol.DEVSTATE)     //获取设备状态信息指令
    {   //请求某个设备系统的设备的子设备状态数据[appid=N][devid=M][type=Y][subid=X]
        NotifyDevice(json);
    }
    else
        NotifyDevice(json);                   //都交给监控驱动去处理
}
```

处理流程很简洁，大多数指令通过 NotifyDevice()方法直接转发给驱动程序进行处理即可，详见源码示例二维码。


源码示例

从源代码中观察到，对于不同通信方式的数据转发，分别使用了不同的方法。对于串口通信和 IPC，数据直接放入 Notify 消息队列，由驱动程序定时从队列中取出处理(参见第 10 章)；对于 TCP 通信，找到 Socket 对象，由它把数据发送给设备系统。

5.3.2　处理设备上传的数据

项目对设备系统发来的数据，根据通信方式的不同，采用了不同的处理方式。

1. TCP 通信的数据处理

DMP 使用 TCP 通信,且作为通信服务器端时,接收到的数据,最终使用 ProcessCommand() 方法来处理,如图 5.17 所示。

```
void ProcessCommand(byte[] bjson, ConnectClient client)  //★★★★处理客户端发来的指令
{
    if (dmp.CommMode == (int)CommMode.TCPFree || dmp.CommMode == (int)CommMode.NBIOT)...
    JObject iotd = Utility.BytesToJson(bjson);
    if (iotd == null) return;   //非系统指令
    string cmd = (string)iotd["cmd"];
    iotd.Add("dmid", DMID.ToString()); //增加监控进程唯一编号
    if (cmd == null) return;   //无命令
    if (chkShowTcp.Checked)...
    if (cmd == IotMonitorProtocol.LOGIN)...
    else if (cmd == IotMonitorProtocol.DEVICESYSTEMINFO)...
    else if (cmd == IotMonitorProtocol.CLIENTEXIT)...
    else if (cmd == IotMonitorProtocol.RUNTASK)...
    else if (cmd == IotMonitorProtocol.DEVSTATE)  //506云端设备发来的设备状态信息
    {
        NotifyDMC(iotd); //先传给传给DMC，以下处理UI界面
        if (currentDeviceSystem == null || currentSubDeviceSystem == null || currentDevice == null) return;
        string type = (string)iotd["type"]; if (type == null) return;
        string dsid = (string)iotd["dsid"]; if (dsid == null) return;  //设备系统
        string ssid = (string)iotd["ssid"]; if (ssid == null) return;  //子设备系统
        string did = (string)iotd["did"]; if (did == null) return;     //设备
        string sdid = (string)iotd["sdid"]; //子设备，可能没有，发来的是整个某类子设备的状态信息
        if (currentDeviceSystem.DSID.ToString() != dsid || currentSubDeviceSystem.SSID.ToString() != ssid
            || currentDevice.DID.ToString() != did)
            return;//不是当前显示的设备
        #region
    }
    else if (cmd == IotMonitorProtocol.POSITIONCHANGED)...
    else  //其他指令
        NotifyDMC(iotd);
}
```

图 5.17 通信客户端发来数据的处理

大部分数据直接上传给 DMC(使用 NotifyDMC()方法)。但对登录指令 LOGIN 做了特定的处理(参见源代码);对设备状态数据(DEVSTATE),在上传数据之后,还在 DMP 的可视化实时监控 UI,动态更新了设备的状态数据和图标,便于实时了解设备状态。

需要注意的是,当设备发来的数据不是标准格式时(非 JSON 结构数据包),处理数据的方法是直接交给驱动程序去处理,由驱动程序把数据转换为标准格式后再处理。

```
if (dmp.CommMode == (int)CommMode.TCPFree || dmp.CommMode == (int)CommMode.NBIOT)
{  //非标准格式的数据处理
    JObject dic = Utility.BytesToJson(bjson);              //转换为 JSON 结构数据
    if (dic!= null)                                        //转换成功:标准格式数据包
    {
        string cmd2 = (string)dic["cmd"];
        if (cmd2 == IotMonitorProtocol.LOGIN)              //联系指令
        {  …  }
        else if (cmd2 == IotMonitorProtocol.CLIENTEXIT) //客户端退出
        {
            DeleteOneClient(client);
            ShowClientInfo();
        }
    }
    //转换失败:非 JSON 结构数据
    monitorSystemmethod.ProcessDeviceSystemData(client,bjson);
    //设备系统 TCP 传来的数据,传给智能监控驱动去处理
    return;
}
```

当 DMD 使用 TCP 通信(DMP 作为客户端连接设备通信服务器),接收到的数据,最终使用 ProcessCommand2()方法来处理,如图 5.18 所示。流程与 ProcessCommand()方法类似。

```
private void ProcessCommand2(JObject iotd) //★★★处理收到设备系统服务端/云端发来的数据
{
    if (iotd == null) return;
    if (chkShowTcp.Checked)
        txtJson.Text += "\r\n收到数据包:" + DateWeekTool.NowTimeString()+"\r\n"+iotd.ToString();  //调试使用
    byte[] rgv = Utility.JsonToBytes(iotd);
    string cmd = (string)iotd["cmd"];
    iotd.Add("rmtdev", "1");//[rmtdev]:远程设备标识,告知监控中心便于特殊处理
    if (cmd == IotMonitorProtocol.LOGIN)...
    else if (cmd == IotMonitorProtocol.DEVICESYSTEMINFO)... //NEWDEVICE
    else if (cmd == IotMonitorProtocol.DEVSTATE)...
    else if (cmd == IotMonitorProtocol.POSITIONCHANGED)...
    else  //其他协议的处理,
        NotifyDMC(iotd);  //直接传给监控中心DMC
}

DeviceSystemBase currentDeviceSystem = null;          //当前选择项的设备系统
SubDeviceSystemBase currentSubDeviceSystem = null;    //当前选择项的子设备系统
Device currentDevice = null;                          //当前选择项的设备
```

图 5.18　通信服务端发来数据的处理

2. 串口通信的数据处理

串口通信是有线通信。设置好波特率等参数后,DMP 与设备进行通信。设计了 ProcessCommData()方法来处理,如图 5.19 所示。

```
private void ProcessCommData(object sender,byte[] buffer)  //★★★处理串口通信数据★
{
    if (setting.chkSerialData)
    {
        if (setting.chkShowHex)
            txtCommData.Text += "\r\n" + DateTime.Now.ToLongTimeString() +
                ":  " + BytesToHex(buffer);
        else
            txtCommData.Text += "\r\n" + DateTime.Now.ToLongTimeString() +
                ":  " + Encoding.Default.GetString(buffer);
        txtCommData.SelectionStart = txtCommData.TextLength;
        txtCommData.ScrollToCaret();
    }
    this.monitorSystemmethod.ProcessDeviceSystemData(sender,buffer);
    //处理串口通信数据,引发事件响应处理代码
}
```

图 5.19　串口通信数据的处理

有线通信的可靠性好、抗干扰、不掉线。在条件允许的情况下,特别是工业监控系统中,大量使用有线通信。

与其他通信数据处理不同,所有数据全部交给驱动程序处理。主要原因是嵌入式设备使用 C 语言编写(参见第 2、3 章),没有使用 JSON 结构。

3. 进程间通信 IPC 的数据处理

如果是虚拟设备,并且运行在边缘服务器上,可以使用进程间通信 IPC 机制与 DMP 交换数据。图 5.20 描述了 IPC 通信数据处理方法。

进程间通信,约定使用标准的 JSON 数据帧结构。

小结:从源代码阅读可观察到,不同通信方式,最终对数据的处理流程大都类似。

```
private void ProcessDeviceInfo(JObject json)      //★★★处理设备程序传来的数据 device-->DMD
{
    if (json == null) return;
    if (setting.bShowCmd) txtMemory2.Text += DateWeekTool.NowTimeString() +
            "-- Device-->DMM -->DMC --\r\n" + json.ToString();
    string cmd = (string)json["cmd"];
    if (cmd == null) return;
    if (cmd == IotMonitorProtocol.DEVSTATE)...
    else if (cmd == IotMonitorProtocol.NEWDEVICE || cmd == IotMonitorProtocol.DEVICESYSTEMINFO)...
    else if (cmd == IotMonitorProtocol.POSITIONCHANGED)...
    else
        NotifyDMC(json);  //先通知服务器，再刷新DMM的UI界面，可以不要
}
```

图 5.20　IPC 通信数据的处理

在线测试

习　　题

一、选择题

1. DMP 与 DMC 之间使用了共享内存和(　　)来交换信息。

   A. 共享文件　　B. 消息队列

   C. 全局变量　　D. 局部变量

2. DMP 使用(　　)机制来保证通信数据的同步访问。

   A. 信号量　　B. 互斥　　C. 临界　　D. 以上均不对

3. 在一个 DMP 程序内,使用了(　　)个消息队列。

   A. 2　　B. 3　　C. 4　　D. 5

4. DMP 动态加载驱动程序的方法是(　　)。

   A. InitShareMemory()　　B. InitShareMemory2()

   C. LoadAssembly()　　D. 以上均不对

5. DMP 启动主从通信功能,主要目标是(　　)。

   A. 周期性获取设备数据　　B. 保持通信连接

   C. 减少通信次数　　D. 以上均不对

二、判断题

1. DMP 是由边缘服务器启动的。(　　)
2. DMP 加载的驱动程序由边缘服务器决定。(　　)
3. DMD 使用的通信对象由自己创建。(　　)
4. DMP 串口接收的通信数据是由驱动程序处理的。(　　)
5. DMP 提供了修改设备名称和功能说明的界面。(　　)