目　　录

官方网站：http://www.bmpj.net

4.如开发一套设备驱动，同时支持串口和网络通讯

4.1 概述

作为物联网通讯框架，肯定要支持多种通讯链路，在多种通讯链路的基础上完成多种通讯协议的交互，例如：Modbus、自定义协议等等。但是，**有一个问题：针对同一台硬件设备或传感器，完成串口和网络两种通讯方式的数据采集和控制，是否要分别写代码？**如果从现实角度分析，同一硬件，它要完成的业务逻辑肯定是相同的，所以 ServerSuperIO 物联网框架，允许开发一套设备驱动，同时支持串口和网络两种通讯方式的交互。

通讯很简单、交互很简单、业务很简单……如果把很多简单的问题合在一起，那么就变得不简单了，所以要有一个框架性的东西，重新把众多问题变得简单。

4.2 通讯协议规定

在完成一个设备驱动的开发之前，首先要知道它的通讯协议，好比两个人交流的语言一样。针对通讯协议，我们自定义一个简单交互方式，只是发送命令，提取数据信息。

4.2.1 发送读实时数据命令协议

计算机发送 0x61 指令为读实时数据命令，共发送 6 个字节，校验和为从“从机地址”开始的累加和，不包括“数据报头”、“校验和”和“协议结束”。

发送指令数据帧如下：

帧结构	数据报头		从机地址	指令代码	校验和	协议结束
	0x55	0xAA		0x61		0x0D
字节数	1	1	1	1	1	1

4.2.2 解析实时数据协议

下位机接收到读实时数据命令后，并校验成功，返回实时数据，校验和为从“从机地址”开始的累加和，不包括“数据报头”、“校验和”和“协议结束”。

接收数据帧如下：

帧结构	数据报头		从机地址	指令代码	流量	信号	校验和	协议结束
	0x55	0xAA		0x61	浮点型	浮点型		0x0D
字节数	1	1	1	1	4	4	1	1

4.2.3 发送和接收数据事例

发送（十六进制）：0x55 0xaa 0x00 0x61 0x61 0x0d

接收（十六进制）：0x55 0xaa 0x00 0x61 0x43 0x7a 0x00 0x00 0x43 0xb4 0x15 0x0d

流量数据为：250.00

信号数据为：360.00

4.3 开发设备驱动

4.3.1 构建实时数据持久对象（不是必须）

1.通过返回数据的通讯协议，有流量和信号两个动态变量，我们需要创建一个动态对象实体类，主要用于协议驱动与设备驱动之间的数据交互。代码如下：

```
public class Dyn
{
        private float _Flow = 0.0f;
        /// <summary>
        /// 流量
        /// </summary>
```

```
        public float Flow
        {
            get { return _Flow; }
            set { _Flow = value; }
        }
        private float _Signal = 0.0f;
        /// <summary>
        /// 信号
        /// </summary>
        public float Signal
        {
            get { return _Signal; }
            set { _Signal = value; }
        }
}
```

2.我们主要的工作是要创建一个实时数据持久对象类，实时缓存数据信息，也可以把该实时数据信息保存到数据库中或其他存储媒质。实时数据持久对象类的代码如下：

```
public class DeviceDyn:DeviceDynamic
{
        public DeviceDyn() : base()
        {
            Dyn=new Dyn();
        }
        public override string GetAlertState()
        {
            throw new NotImplementedException("无报警信息");
        }
        public override object Repair()
        {
            return new DeviceDyn();
        }
        public Dyn Dyn { get; set; }
}
```

DeviceDyn 类继承自 DeviceDynamic，因为每个硬件设备的报警信息有可能不一样，所以 GetAlertState 函数可以实该功能，但是 SSIO 框架并没有直接引用；这个类本质上是一个可以序列化，在不加互斥的情况下可能造成文件损坏，所以 Repair 可以完成修复功能，在 DeviceDynamic 基类里实现了该功能；另外，实现 DeviceDynamic 基类自带两个函数，Save 函数用于持久化（序列化）此类的信息，

Load 用于获得（反序列化）此类的信息，在设备驱动中可以使用。

4.3.2 构建参数数据持久对象

一般来说硬件设备会有读参数的命令，那么返回来的参数也需要进行持久化存储，并且每台设备的参数都可能不一样，在此提供一个可扩展的接口。在这个通讯协议中并没有涉及到设备参数相关的协议说明，但是我们也需要创建一个参数数据持久对象类，可以不写任何扩展的参数属性，在 SSIO 框架对参数的接口进行了引用，这是必须进行了工作。代码如下：

```
public class DevicePara:ServerSuperIO.Device.DeviceParameter
{
        public override object Repair()
        {
            return new DevicePara();
        }
}
```

DevicePara 继承自 DeviceParameter 类，情况与实时数据持久对象类似，可以参数。

4.3.3 构建发送和解析协议命令对象

与设备进行交互会涉及到很多交互式的命令或指令代码，而这些命令在 SSIO 框架内是以协议命令对象的形式存在，大体包括三个部：**执行命令接口、打包发送数据接口、解析接收数据接口**等。

针对上面的通讯协议，有一个 61 指令，那么我们就可以根据 61 指令为命名构建一个协议命令对象，包括发送数据和解析数据部分。如果有其他命令代码，举一反三。代码如下：

```
internal class DeviceCommand:ProtocolCommand
{
        public override string Name
        {
            get { return "61"; }
        }

        public override void ExcuteCommand<T>(T t)
```

```
        {
            throw new NotImplementedException();
        }

        public override byte[] Package<T> (string code, T1 t1,T2 t2)
        {
            //发送：0x55 0xaa 0x00 0x61 0x61 0x0d
            byte[] data = new byte[6];
            data[0] = 0x55;
            data[1] = 0xaa;
            data[2] = byte.Parse(code);
            data[3] = 0x61;
            data[4] = this.ProtocolDriver.GetCheckData(data)[0];
            data[5] = 0x0d;
            return data;
        }

        public override dynamic Analysis<T>(byte[] data, T t)
        {
            Dyn dyn = new Dyn()
            //一般下位机是单片的话，接收到数据的高低位需要互换，才能正常解析。
            byte[] flow = BinaryUtil.SubBytes(data, 4, 4, true);
            dyn.Flow = BitConverter.ToSingle(flow, 0);
            byte[] signal = BinaryUtil.SubBytes(data, 8, 4, true);
            dyn.Signal = BitConverter.ToSingle(signal, 0);
            return dyn;
        }
}
```

构建协议命令需要全部继承自 ProtocolCommand，根据通讯协议规定，Name 属性返回 61，作为关键字；Package 是打包要送的数据信息；Analysis 对应着接收数据之后进行解析操作。就这样一个简单的协议命令驱动就构建完成了。

4.3.4 构建协议驱动对象

有了协议命令之后，我们需要构建协议驱动对象，SSIO 框架支持自定义协议也在于此，并且与设备驱动的接口相关联，在 SSIO 框架的高级应用中也进行了引用，构建这引对象很关键。代码如下：

```
internal class DeviceProtocol:ProtocolDriver
{
        public override bool CheckData(byte[] data)
```

```
{
    if (data[0] == 0x55 && data[1] == 0xaa && data[data.Length - 1] == 0x0d)
    {
        return true;
    }
    else
    {
        return false;
    }
}

public override byte[] GetCommand(byte[] data)
{
    return new byte[] { data[3] };
}

public override int GetAddress(byte[] data)
{
    return data[2];
}

public override byte[] GetHead(byte[] data)
{
    return new byte[] { data[0], data[1] };
}

public override byte[] GetEnd(byte[] data)
{
    return new byte[] { data[data.Length - 1] };
}

public override byte[] GetCheckData(byte[] data)
{
    byte checkSum = 0;
    for (int i = 2; i < data.Length - 2; i++)
    {
        checkSum += data[i];
    }
    return new byte[] { checkSum };
}

public override string GetCode(byte[] data)
{
    throw new NotImplementedException();
```

```
        }

        public override int GetPackageLength(byte[] data, IChannel channel, ref int
readTimeout)
        {
            throw new NotImplementedException();
        }
}
```

DeviceProtocol 协议驱动继承自 ProtocolDriver ，一个设备驱动只存在一个协议驱动，一个协议驱动可以存在多个协议命令（如 61 命令）。该类中的 CheckData 函数很关键，SSIO 框架中的设备驱动基类引用了，主要是完成校验接收数据的完事性，是否符合协议，从而决定了通讯状态：通讯正常、通讯中断、通讯干扰、以及通讯未知，不同的通讯状态也决定了调用设备驱动中的哪个函数接口：Communicate 、CommunicateInterrupt 、CommunicateError 和 CommunicateNone。

4.3.5 构建设备驱动对象

上边的基础工作都做完之后，现在就构建设备驱动的核心部分，也就是 SSIO 框架与设备驱动对接、协调、调度的唯一接口，写完这个接口，设备驱动就可以在 SSIO 上直接运行了，并且与硬件设备进行交互。直接上代码：

```
public class DeviceDriver:RunDevice
{
        private DeviceDyn _deviceDyn;
        private DevicePara _devicePara;
        private DeviceProtocol _protocol;
        public DeviceDriver() : base()
        {
            _devicePara = new DevicePara();
            _deviceDyn = new DeviceDyn();
            _protocol = new DeviceProtocol();
        }

        public override void Initialize(string devid)
        {
            this.Protocol.InitDriver(this.GetType(),null);

            //初始化设备参数信息
```

```
            _devicePara.DeviceID = devid;//设备的ID必须先赋值，因为要查找对应的参数文件。
            if (System.IO.File.Exists(_devicePara.SavePath))
            {
                //如果参数文件存在，则获得参数实例
                _devicePara = _devicePara.Load<DevicePara>();
            }
            else
            {
                //如果参数文件不存在，则序列化一个文件
                _devicePara.Save<DevicePara>(_devicePara);
            }

            //初始化设备实时数据信息
            _deviceDyn.DeviceID = devid;//设备的ID必须先赋值，因为要查找对应的实时数据文
件。
            if (System.IO.File.Exists(_deviceDyn.SavePath))
            {
                //如果参数文件存在，则获得参数实例
                _deviceDyn = _deviceDyn.Load<DeviceDyn>();
            }
            else
            {
                //如果参数文件不存在，则序列化一个文件
                _deviceDyn.Save<DeviceDyn>(_deviceDyn);
            }
        }

        public override byte[] GetConstantCommand()
        {
            return this.Protocol.DriverPackage<String>("0", "61", null);
        }

        public override void Communicate(ServerSuperIO.Communicate.IRequestInfo info)
        {
            Dyn dyn = this.Protocol.DriverAnalysis<String>("61", info.Data, null);
            if (dyn != null)
            {
                _deviceDyn.Dyn = dyn;
            }
            OnDeviceRuningLog("通讯正常");
        }

        public override void CommunicateInterrupt(ServerSuperIO.Communicate.IRequestInfo
```

```
info)
        {
            OnDeviceRuningLog("通讯中断");
        }

        public override void CommunicateError(ServerSuperIO.Communicate.IRequestInfo
info)
        {
            OnDeviceRuningLog("通讯干扰");
        }

        public override void CommunicateNone()
        {
            OnDeviceRuningLog("通讯未知");
        }

        public override void Alert()
        {
            return;
        }

        public override void Save()
        {
            try
            {
                _deviceDyn.Save<DeviceDyn>(_deviceDyn);
            }
            catch (Exception ex)
            {
                OnDeviceRuningLog(ex.Message);
            }
        }

        public override void Show()
        {
            List<string> list=new List<string>();
            list.Add(_devicePara.DeviceName);
            list.Add(_deviceDyn.Dyn.Flow.ToString());
            list.Add(_deviceDyn.Dyn.Signal.ToString());
            OnDeviceObjectChanged(list.ToArray());
        }

        public override void UnknownIO()
        {
```

```
            OnDeviceRuningLog("未知通讯接口");
        }

        public override void
CommunicateStateChanged(ServerSuperIO.Communicate.CommunicateState comState)
        {
            OnDeviceRuningLog("通讯状态改变");
        }

        public override void ChannelStateChanged(ServerSuperIO.Communicate.ChannelState
channelState)
        {
            OnDeviceRuningLog("通道状态改变");
        }

        public override void Exit()
        {
            OnDeviceRuningLog("退出设备");
        }

        public override void Delete()
        {
            OnDeviceRuningLog("删除设备");
        }

        public override object GetObject()
        {
            throw new NotImplementedException();
        }

        public override void ShowContextMenu()
        {
            throw new NotImplementedException();
        }

        public override IDeviceDynamic DeviceDynamic
        {
            get { return _deviceDyn; }
        }

        public override IDeviceParameter DeviceParameter
        {
            get { return _devicePara; }
        }
```

```
        public override IProtocolDriver Protocol
        {
            get { return _protocol;}
        }

        public override DeviceType DeviceType
        {
            get { return DeviceType.Common; }
        }

        public override string ModelNumber
        {
            get { return "serversuperio"; }
        }

        public override System.Windows.Forms.Control DeviceGraphics
        {
            get { throw new NotImplementedException(); }
        }
}
```

实时动态数据对象_deviceDyn、参数数据对象_devicePara、协议驱动对象_protocol 分别提供给接口：DeviceDynamic、DeviceParameter 和 Protocol，为 SSIO 提供可引用的基础属性参数。

Initialize 是设备驱动初始化的函数接口，在这个接口完成两个主要工作：初始化协议驱动和参数性的信息。通过 this.Protocol.InitDriver(this.GetType(),null); 代码可以加载所有协议命令到协议驱动的缓存中，以便实时调用。当然这里边也可以进行其他方面的工作，但是注意对异常的处理。

DeviceType 这个是设备的类型，一般指定为 Common 就好了。其他函数接口功能已经在《物联网框架 ServerSuperIO 教程-3.设备驱动介绍》中详细介绍了，请参考。

4.4 构建宿主程序

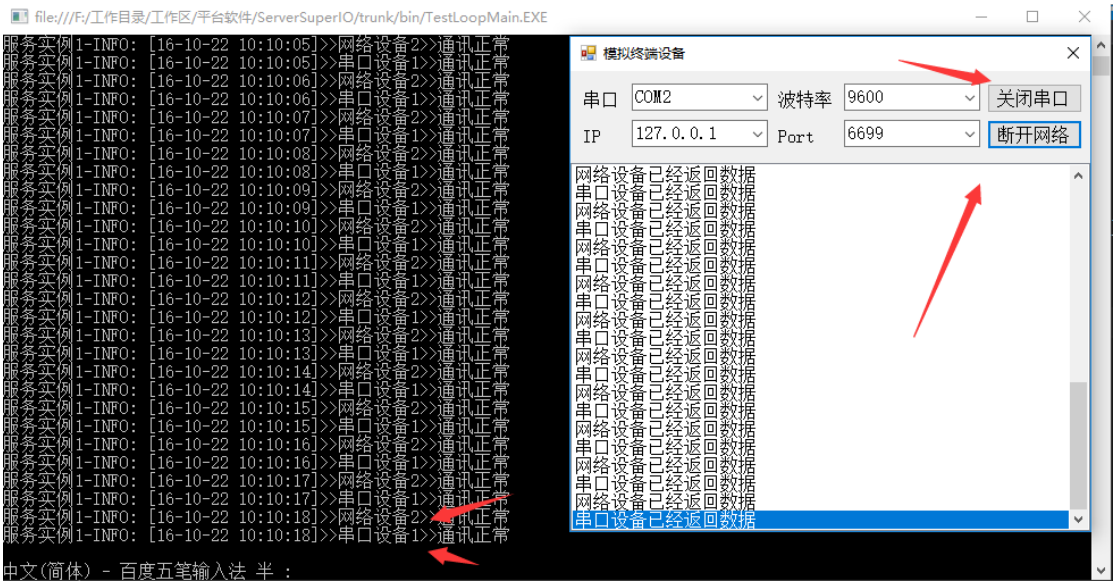
一个简单的设备驱动就已经开发好了，光有驱动还不行，那么我们基于 SSIO 框架再写几行代码，完成一个宿主程序，把设备驱动实例化，放 SSIO 的服务实例中运行，完成串口和网络两种方式的通讯交互，代码也非常简单。代码如下：

```
class Program
{
        static void Main(string[] args)
        {
            DeviceDriver dev1 = new DeviceDriver();
            dev1.DeviceParameter.DeviceName = "串口设备1";
            dev1.DeviceParameter.DeviceAddr = 0;
            dev1.DeviceParameter.DeviceID = "0";
            dev1.DeviceDynamic.DeviceID = "0";
            dev1.DeviceParameter.COM.Port = 1;
            dev1.DeviceParameter.COM.Baud = 9600;
            dev1.CommunicateType = CommunicateType.COM;
            dev1.Initialize("0");

            DeviceDriver dev4 = new DeviceDriver();
            dev4.DeviceParameter.DeviceName = "网络设备2";
            dev4.DeviceParameter.DeviceAddr = 0;
            dev4.DeviceParameter.DeviceID = "3";
            dev4.DeviceDynamic.DeviceID = "3";
            dev4.DeviceParameter.NET.RemoteIP = "127.0.0.1";
            dev4.DeviceParameter.NET.RemotePort = 9600;
            dev4.CommunicateType = CommunicateType.NET;
            dev4.Initialize("3");

            IServer server = new ServerFactory().CreateServer(new ServerConfig()
            {
                ServerName = "服务实例1",
                SocketMode = SocketMode.Tcp,
                ControlMode = ControlMode.Loop,
                CheckSameSocketSession = false,
                StartCheckPackageLength = false,
            });

            server.AddDeviceCompleted += server_AddDeviceCompleted;
            server.DeleteDeviceCompleted += server_DeleteDeviceCompleted;
            server.SocketConnected+=server_SocketConnected;
            server.SocketClosed+=server_SocketClosed;
            server.Start();

            server.AddDevice(dev1);
            server.AddDevice(dev4);

            while ("exit"==Console.ReadLine())
            {
```

```
                server.Stop();
            }
        }

        private static void server_SocketClosed(string ip, int port)
        {
            Console.WriteLine(String.Format("断开：{0}-{1} 成功", ip, port));
        }

        private static void server_SocketConnected(string ip, int port)
        {
            Console.WriteLine(String.Format("连接：{0}-{1} 成功",ip, port));
        }

        private static void server_AddDeviceCompleted(string devid, string devName, bool
isSuccess)
        {
            Console.WriteLine(devName+",增加:"+isSuccess.ToString());
        }

        private static void server_DeleteDeviceCompleted(string devid, string devName,
bool isSuccess)
        {
            Console.WriteLine(devName + ",删除:" + isSuccess.ToString());
        }
    }
}
```

这个代码大家都能看明白，具体的控制模式我们接下来会一一介绍。在构建宿主程序的时候，切忌对服务实例这样引用：server.ChannelManager、server.ControllerManager、server.DeviceManager。尽管提供了这样的接口，主要是为了 SSIO 框架内部使用的，不需要我们单独去操作这些接口。有的网友是这样的写的，那么就变成了一个纯的通信 IO 框架，那么就失去了 SSIO 框架本身的价值。作为二次开发者，只需要设置设备驱动的参数，以及向服务实例中增加或删除设备就行了，其他所有的运行全部交给 SSIO 框架来完成。

4.5 运行效果


file:///F:/工作目录/工作区/平台软件/ServerSuperIO/trunk/bin/TestLoopMain.EXE
服务实例1-INFO: [16-10-22 10:10:05]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:05]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:06]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:06]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:07]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:07]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:08]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:08]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:09]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:09]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:10]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:10]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:11]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:11]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:12]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:12]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:13]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:13]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:14]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:14]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:15]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:15]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:16]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:16]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:17]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:17]>>串口设备1>>通讯正常
服务实例1-INFO: [16-10-22 10:10:18]>>网络设备2>>通讯正常
服务实例1-INFO: [16-10-22 10:10:18]>>串口设备1>>通讯正常
中文(简体) - 百度五笔输入法 半 ：
模拟终端设备
串口
COM2
波特率
9600
关闭串口
IP
127.0.0.1
Port
6699
断开网络
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据
网络设备已经返回数据
串口设备已经返回数据