目　　录

官方网站：http://www.bmpj.net

3.设备驱动介绍

3.1 概述

定位 ServerSuperIO（SSIO）为物联网通讯框架，就是因为这个框架是以“设备”（驱动）为核心构建，“设备”是泛指传感器、下位机、PC 机等各类数据源，数据源有自己的通讯协议或数据传输格式；ServerSuperIO 并不是以 IO 通道为核心构建的框架，但是 ServerSuperIO 有很好的通讯能力，完全可以部署在服务端，并且支持多个服务实例，以及可以在 Linux 下运行。

3.2 接口定义

ServerSuperIO 的“设备”统一接口定义为 IRunDevice，这是在框架内部运行、调度、与 IO 通道协作的唯一接口。代码定义如下：

```
public interface IRunDevice: IServerProvider,IVirtualDevice
{
    #region 函数接口

    /// <summary>
    /// 初始化设备，加载设备驱动的头一件事就是初始化设备
    /// </summary>
    /// <param name="devid"></param>
    void Initialize(string devid);

    /// <summary>
    /// 保存原始的byte数据
    /// </summary>
    /// <param name="data"></param>
    void SaveBytes(byte[] data, string desc);

    /// <summary>
    /// 获得发送数据的命令，如果命令缓存中没有命令，则调用获得实时数据函数
    /// </summary>
    /// <returns></returns>
    byte[] GetSendBytes();
```

```
/// <summary>
/// 如果当前命令缓存没有命令，则调用该函数,一般返回获得设备的实时数据命令，
/// </summary>
/// <returns></returns>
byte[] GetConstantCommand();

/// <summary>
/// 发送IO数据接口
/// </summary>
/// <param name="io"></param>
/// <param name="senddata"></param>
int Send(IChannel io, byte[] senddata);

/// <summary>
/// 读取IO数据接口
/// </summary>
/// <param name="io"></param>
/// <returns></returns>
byte[] Receive(IChannel io);

/// <summary>
/// 接收数据信息，带过滤器
/// </summary>
/// <param name="io"></param>
/// <param name="receiveFilter"></param>
/// <returns></returns>
IList<byte[]> Receive(IChannel io, IReceiveFilter receiveFilter);

/// <summary>
/// 同步运行设备（IO）
/// </summary>
/// <param name="io">io实例对象</param>
void Run(IChannel io);

/// <summary>
/// 同步运行设备（byte[]）
/// </summary>
/// <param name="key"></param>
/// <param name="channel"></param>
/// <param name="revData">接收到的数据</param>
void Run(string key, IChannel channel, IRequestInfo ri);

/// <summary>
```

```
/// 如果通讯正常，这个函数负责处理数据
/// </summary>
/// <param name="info"></param>
void Communicate(IRequestInfo info);

/// <summary>
/// 通讯中断，未接收到数据
/// </summary>
void CommunicateInterrupt(IRequestInfo info);

/// <summary>
/// 通讯的数据错误或受到干扰
/// </summary>
void CommunicateError(IRequestInfo info);

/// <summary>
/// 通讯未知，默认状态（一般不用）
/// </summary>
void CommunicateNone();

/// <summary>
/// 检测通讯状态
/// </summary>
/// <param name="revdata"></param>
/// <returns></returns>
CommunicateState CheckCommunicateState(byte[] revdata);

/// <summary>
/// 报警接口函数
/// </summary>
void Alert();

/// <summary>
/// 保存解析后的数据
/// </summary>
void Save();

/// <summary>
/// 展示
/// </summary>
void Show();

/// <summary>
```

```
/// 当通讯实例为NULL的时候，调用该函数
/// </summary>
void UnknownIO();

/// <summary>
/// 通讯状态改变
/// </summary>
/// <param name="comState">改变后的状态</param>
void CommunicateStateChanged(CommunicateState comState);

/// <summary>
/// 通道状态改变
/// </summary>
/// <param name="channelState"></param>
void ChannelStateChanged(ChannelState channelState);


/// <summary>
/// 当软件关闭的时间，响应设备退出操作
/// </summary>
void Exit();

/// <summary>
/// 删除设备的响应接口函数
/// </summary>
void Delete();

/// <summary>
/// 可以自定义返数据对象，用于与其他组件交互
/// </summary>
/// <returns></returns>
object GetObject();

/// <summary>
/// 设备定时器，响应定时任务
/// </summary>
void OnRunTimer();

/// <summary>
/// 显示上下文菜单
/// </summary>
void ShowContextMenu();
```

```
        /// <summary>
        /// 显示IO监视器的窗体
        /// </summary>
        void ShowMonitorDialog();

        /// <summary>
        /// 在IO监视器上显示byte[]数据
        /// </summary>
        /// <param name="data"></param>
        /// <param name="desc"></param>
        void ShowMonitorData(byte[] data, string desc);
        #endregion

        #region 属性接口
        /// <summary>
        /// 默认程序集ID，用于存储临时对象
        /// </summary>
        object Tag { set; get; }

        /// <summary>
        /// 同步对象，用于IO互拆
        /// </summary>
        object SyncLock { get; }

        /// <summary>
        /// 实时数据持久接口
        /// </summary>
        IDeviceDynamic DeviceDynamic { get; }

        /// <summary>
        /// 设备参数持久接口
        /// </summary>
        IDeviceParameter DeviceParameter { get; }

        /// <summary>
        /// 协议驱动
        /// </summary>
        IProtocolDriver Protocol { get; }

        /// <summary>
        /// 是否开启时钟，标识是否调用OnRunTimer接口函数。
        /// </summary>
        bool IsRunTimer { set; get;}
```

```
/// <summary>
/// 时钟间隔值，标识定时调用DeviceTimer接口函数的周期
/// </summary>
int RunTimerInterval { set; get; }

/// <summary>
/// 设备的类型
/// </summary>
DeviceType DeviceType { get;  }

/// <summary>
/// 设备编号
/// </summary>
string ModelNumber { get;}

/// <summary>
/// 设备运行权限级别，如果运行级别高的话，则优先发送和接收数据。
/// </summary>
DevicePriority DevicePriority { get;set;}

/// <summary>
/// 设备的通讯类型
/// </summary>
CommunicateType CommunicateType { get;set;}

/// <summary>
/// 标识是否运行设备，如果为false，调用运行设备接口时直接返回
/// </summary>
bool IsRunDevice{ get;set;}

/// <summary>
/// 是否释放资源
/// </summary>
bool IsDisposed { get; }

/// <summary>
/// 显示视图
/// </summary>
Control DeviceGraphics { get;}
#endregion

#region 事件接口
```

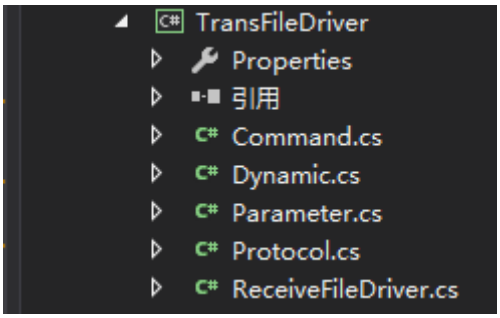
```
/// <summary>
/// 发送数据事件
/// </summary>
event SendDataHandler SendData;

/// <summary>
/// 发送数据事件，对SendDataHandler事件的封装
/// </summary>
/// <param name="senddata"></param>
void OnSendData(byte[] senddata);

/// <summary>
/// 设备日志输出事件
/// </summary>
event DeviceRuningLogHandler DeviceRuningLog;

/// <summary>
/// 运行监视器显示日志事件，对DeviceRuningLogHandler事件的封装
/// </summary>
void OnDeviceRuningLog(string statetext);

/// <summary>
/// 串口参数改变事件
/// </summary>
event ComParameterExchangeHandler ComParameterExchange;

/// <summary>
/// 串口参数改变事件，对COMParameterExchangeHandler事件的封装
/// </summary>
void OnComParameterExchange(int oldcom, int oldbaud, int newcom, int newbaud);

/// <summary>
/// 设备数据对象改变事件
/// </summary>
event DeviceObjectChangedHandler DeviceObjectChanged;
/// <summary>
/// 数据驱动事件，对DeviceObjectChangedHandler事件的封装
/// </summary>
void OnDeviceObjectChanged(object obj);

///// <summary>
///// 删除设备事件
```

```
///// </summary>
//event DeleteDeviceCompletedHandler DeleteDeviceCompleted;
///// <summary>
///// 删除设备事件，对DeleteDeviceHandler事件的封装
///// </summary>
//void OnDeleteDeviceCompleted();
#endregion
```

3.3 二次开发，常用接口

RunDevice 抽象类继承自 IRunDevice 接口，本质上来讲，二次开发只需要继承 RunDevice 抽象类就可以了。RunDevice 已经完成了设备驱动在 ServerSuperIO 框架下基本的条件。那么在继承 RunDevice 抽象类的时候，所需要二次开发的工作很小，只需要关注协议和处理数据业务本身，对于框架的内部运行机制可以配置、调度机制内部自动处理。

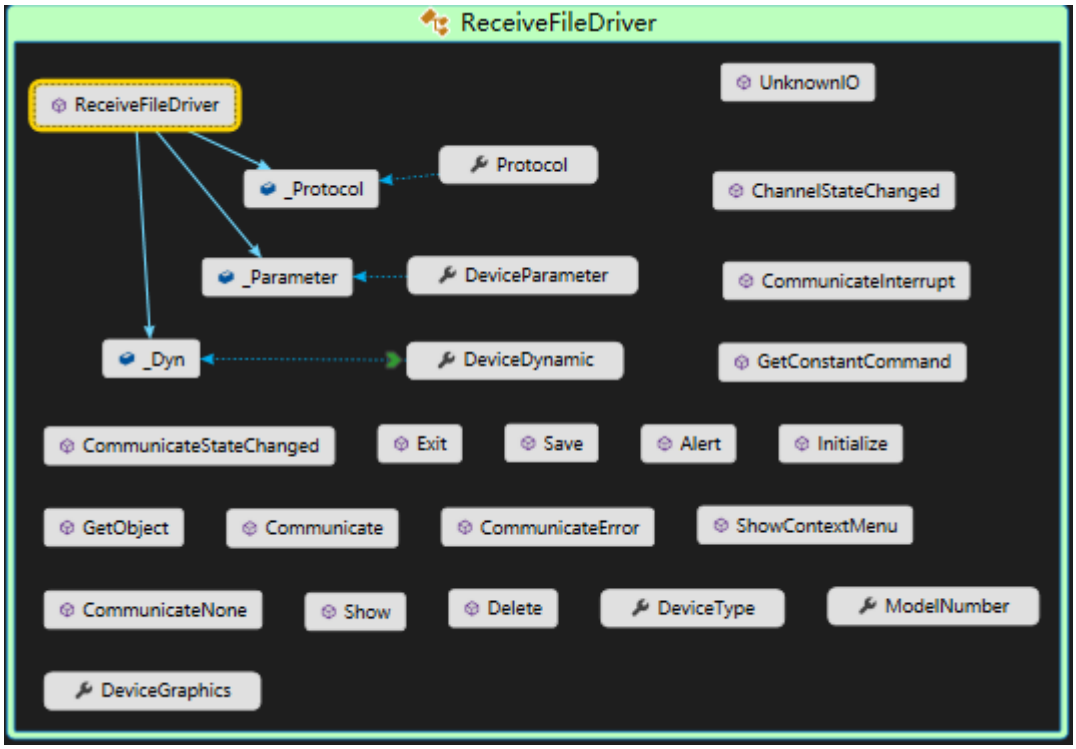
二次开发常用的接口，项目示意如下图：



这是一个按协议规则完成实时接收文件数据的设备驱动，ReceiveFileDriver 继承自 RunDevice 抽象类，是设备驱动的核心接口；Protocol 是自定义协议接口，包括发送数据协议和接收数据协议，实例化后在 ReceiveFileDriver 接口的 IProtocolDriver 属性中返回；不光要有协议接口，在协议里边还要有命令，那么 Command 就是协议中的自定义命令，这个协议命令规定了发送数据和接收解析数据，协议命令不仅仅包括一个，根据业务需要，设备驱动可以包含多个协议命令，以完成与实体硬件的交互；Parameter 自定义参数接口，每个设备对象本身的参数不一样，实例化后在 ReceiveFileDriver 接口的 IDeviceParameter 属性中返回；Dynamic 是自定义实时数据临时缓存接口，可以把解析后的实时数据临时存储在这个对象里，，实例化后在 ReceiveFileDriver 接口的 IDeviceDynamic 属性中

返回，这个对象不是必须要实现的，ServerSuperIO 内部并没有直接引用。

以上是从项目角度大体阐述了需要写哪几类代码，ReceiveFileDriver 设备驱动需要二次开发写的代码，如下示意图：



1) Initialize：初始化设备驱动接口，在这里可以初始化协议驱动、设备参数和设备的实时数据等对象，以及完成一些其他的操作。
2) DeviceDynamic：实时数据对象接口属性，不是必须实现。
3) DeviceParameter：参数数据对象接口属性，必须实现。
4) Protocol：协议驱动接口，很关键，必须实现。
5) DeviceType：设备类型，一般指普通设备，虚拟设备适用于特殊情况，必须实现。
6) ModelNumber：设备驱动的唯一编号，一般必须实现。
7) GetConstantCommand：固定返回发送数据的接口，一般返回读实时数据命令，在 SendCache 里没有命令的时候，会调用该接口，必须实现。
8) Communicate：通讯正常，会把数据返回到这个接口，影响通讯状态为 Protocol 接口中的 CheckData 函数，校验接收到数据的完整性。
9) CommunicateInterrupt：通讯中断代表没有接收到任何数据，会调用这个接口，

影响通讯状态为 Protocol 接口中的 CheckData 函数，校验接收到数据的完整性。

10) CommunicateError：通讯干扰代表接收到数据，但是没有校验正确，会调用这个接口，影响通讯状态为 Protocol 接口中的 CheckData 函数，校验接收到数据的完整性。
11) CommunicateNone：通讯未知，代表该设备驱动对应的 IO 通道 COM 没有打开或 NET 没有有效连接。
12) UnknownIO：未知 IO 通道，COM 没有打开或 NET 没有有效连接。
13) CommunicateStateChanged：每次通讯状态改变后会调用这个接口函数。
14) ChannelStateChanged：IO 通道状态改变后会调用这个接口函数，COM 是打开或关闭，NET 是连接或断开。
15) Save：保存数据函数接口，设备驱动执行到生命周期最后会调用这个接口，以完成对数据的保存。
16) Alert：报警函数接口，数据处理完成后，对数据进行判断，以完成报警功能。
17) Show：显示数据函数接口，数据处理完成后，对数据进行显示，是 SIO 保留下来的接口函数，以后扩展使用。
18) Exit：当设备驱动退出时调用该函数接口，例如：宿主程序退出时。
19) Delete：删除设备驱动调用该函数接口，是 SIO 保留下来的接口函数，适用于有界面 UI 的应用场景，以后扩展使用。
20) DeviceGraphics：视图接口，是 SIO 保留下来的接口函数，以后扩展使用。
21) GetObject：获得数据对象，以后打算为服务接口供数据支持，现在没有实际用处。
22) ShowContextMenu：显示上下文菜单，是 SIO 保留下来的接口函数，以后扩展使用。