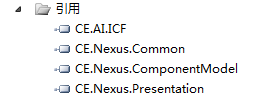
**设备驱动开发**

# 文档适用范围

本规范适用于九思易驱动开发人员，开发NX驱动适用的文档规范。

# 驱动开发需要引用继承的IO基类



说明：

1：CE.AI.ICF.dll //IO通讯的基类 集成了DeviceDev 、EthernetDevice 类等一些基类

2：CE.Nexus.Common.dll

CE.Nexus.ComponentModel.dll

CE.Nexus.Presentation.dll //这三个用于属性配置界面

# 第一章 设备驱动构成

## 1.1 构成

驱动程序基本构成：1个主程序dll文件、1个xml文件（有服务方法时必须提供，其它可以不提供），2个cfg文件（驱动描述文件），1个中文资源dll文件（默认资源文件以内嵌到主程序dll文件中）。

驱动开发提交的结果参照如下目录结构：\*.dll文件为主程序dll文件，\*.cfg文件为默认语言（英文）的驱动描述文件，*zh-CN*文件夹下为中文的驱动描述文件和中文的资源dll文件。

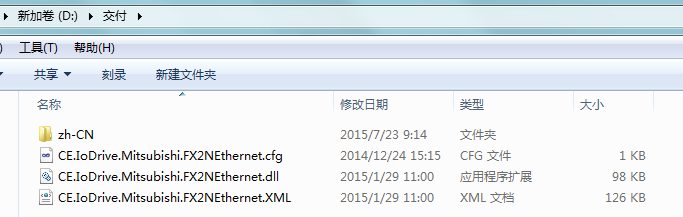


图1-1交付根目录

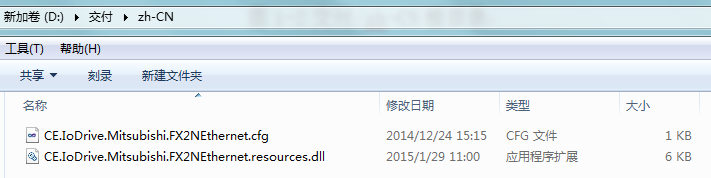


图1-2交付/zh-CN根目录

## 1.2 文件规则

**安装目录**： 主程序dll和默认的cfg文件位于NX软件安装目录下的*\Drivers\Device\*文件夹下，中文cfg文件位于*\Drivers\Device\zh-CN*文件夹下，中文的资源dll文件位于NX软件安装目录下的*zh-CN*文件夹下。

### 1.2.1 主程序dll文件

主程序dll文件主要实现设备驱动用于IO开发环境和IO运行环境的功能。主要是：实现开发类和运行类，通信参数配置的WPF控件，提供对多语言的支持（内嵌XXX.Resources.resx默认资源文件及中文资源文件XXX.Resources.zh-CN.resx）。

### 1.2.2 资源文件

资源文件中要定义显示给用户的信息的多语言内容。

主要有：驱动配置控件文本内容的多语言支持等。

### 1.2.3 cfg文件

Cfg采用XML格式描述，文件后缀为cfg，文件名称要不同于其它驱动程序保证唯一，一般采用程序集名称的方式。

* **文件内容：**

a.驱动的名称，版本，描述信息

b. 驱动程序集的主要信息:主程序集名称，设备开发类名称，设备运行类名称。

c.驱动支持的设备的信息：设备厂家、设备类别、设备型号。其中“设备厂家”要支持多语言，开发环境下驱动列表工具箱会使用该多语言信息。

d.寄存器列标题信息的多语言的信息。

* **文件结构：**

以三菱Q PLC驱动为例，中文Cfg文件格式如下：

<DeviceDriver Version="1.0"

Name="MelsecQ"

InterfaceType="Ethernet"

Assembly="CE.IoDrive.Mitsubishi.MelsecQDevice.dll" DevClass="ControlEase.IoDrive.Mitsubishi.MelsecQDeviceDev"

RunClass="ControlEase.IoDrive.Mitsubishi.MelsecQDevice"

ThirdPartyAssemblys="A.dll; B.dll"

Description="Ethernet driver for MelsecQ device">

<!--支持的设备信息-->

<ApplicableDevices>

<Device Manufacture="Mitsubishi" Category="PLC" Model="MelsecQ"/>

</ApplicableDevices>

<!--寄存器列标题-->

<Register NetworkNo="NetworkNo" DeviceNo="DeviceNo" Memory="Region" Index="Index" Subindex="Subindex" Length="Length"/>

</DeviceDriver>

中文cfg文件内容如下：

<DeviceDriver Version="1.0"

Name="MelsecQ"

InterfaceType="Ethernet"

Assembly="CE.IoDrive.Mitsubishi.MelsecQDevice.dll"

DevClass="ControlEase.IoDrive.Mitsubishi.MelsecQDeviceDev"

RunClass="ControlEase.IoDrive.Mitsubishi.MelsecQDevice"

ThirdPartyAssemblys="A.dll; B.dll"

Description="MelsecQ以太网驱动">

<ApplicableDevices>

<Device Manufacture="三菱" Category="PLC" Model="MelsecQ"/>

</ApplicableDevices>

<Register NetworkNo="网络号" DeviceNo="设备号" Memory="分区" Index="索引" Subindex="子索引" Length="长度"/>

</DeviceDriver>

当用户通过驱动安装工具将设备驱动程序安装到NX软件中后，将出现在NX软件的已安装驱动列表中, IO开发环境的驱动工具箱中可以浏览已安装的驱动。

#### 代码描述规定

Assembly：程序集全名称；

DevClass：开发类全名称（包括命名空间）；

RunClass：运行类全名称（包括命名空间）

例如：

<DeviceDriver Name=" MitsubishiFX2NPLC"

InterfaceType="Ethernet"

Assembly=" CE.IoDrive.Mitsubishi.FX2NEthernet.dll "

DevClass=" ControlEase.IoDrive.Mitsubishi.FX2NDeviceDev"

RunClass=" ControlEase.IoDrive.Mitsubishi.FX2NDevice" />

#### 名称规定

该名称采用英文描述。

设备驱动的名称会显示在IO开发环境的工具箱中供用户查看，故该名称应准确反应驱动的信息，同时考虑显示上的美观。

#### 支持的设备信息规定

当驱动开发人员已经开发了Mitsubishi的以太网FX2N模块，又开发了Mitsubishi的串口FX2N模块时，若Manufacture/Model/Category相同，这两个驱动就可以在IO开发环境中用在一个设备上。

**注：**在开发环境作图时，IO开发环境会根据 Manufacture，Model, 以及Category的信息进行匹配，来决定是否允许用户将不同的驱动上添加到同一个设备上的依据。

#### Version规定

驱动的版本号，发给用户的驱动的初始版本号规定为1.0;若后续修改Bug或扩展功能，发给用户前需要更新该版本号，如更新为1.1，1.2等。

**注：**尤其对于有服务方法的驱动来说，若扩展驱动功能时新增加了服务方法，则必须要更新版本号，否则用户的驱动升级后，在脚本子系统中仍然不能使用这些新增加的服务方法。

#### Model规定

1. 当一个驱动文件支持2种及以上型号的设备时，用逗号（,）隔开，如“S7300,S7400”；
2. 如果驱动程序支持一系列设备型号，以协议名称命名；
3. 如果驱动程序的适用性非常广，则使用通配符“\*”表示。

#### 设备种类规定

即cfg文件中的Category。当设备驱动支持多个厂商的多种设备时，可能Category会有不同，IO开发默认是以主要支持的设备厂家支持的设备种类为基准，会利用该信息选择设备的图标。

设备种类Categories

|  |  |
| --- | --- |
| **Categories**  **（设备种类，设备所属的分类信息）** | |
| **值** | **功能说明** |
| PLC | 可编程控制器 |
| IOCard | IO板卡 |
| Instrument | (Intelligent[Instrument](app:ds:instrument)) 智能仪表 |
| W-[Instrument](app:ds:instrument) | ([Weigh](app:ds:weighing)ing[Instrument](app:ds:instrument)）称重仪表 |
| [Module](app:ds:module) | 模块 |
| VFD | Variable Frequency Drive, 变频器 |
| UPS | Uninterruptible Power System，不间断电源 |
| PCP | [Power System Communication Protocol](app:ds:power%20system%20communication%20protocol)，电力通信规约 |
| LED | LED显示屏 |
| MC | MotionController,运动控制器 |
| Printer | 打印机 |
| Other | 其他（不好划分种类的设备归到此类） |

#### 设备接口类型规定

设备接口类型InterfaceType

|  |  |
| --- | --- |
| InterfaceType  **（设备接口类型，设备接口的信息）** | |
| **值** | **功能说明** |
| Serial | 串口 |
| Ethernet | 以太网 |
| USB | USB |
| Virtual | 虚拟软通道 |

#### 引用第三方程序集规定

第三方程序集是指，引用的程序集没有安装包进行安装(一般厂商只给一个dll文件)，以及驱动要引用一个我们自己封装的另外程序集。第三方程序集有一个共同的特点，就是需要和驱动程序集文件放在相同的目录下。

驱动引用的第三方程序集通过配置文件的“ThirdPartyAssemblys”属性制定，驱动需要引用多个第三方程序集，程序集名中间用“;”分隔，如下所示

<DeviceDriver ThirdPartyAssemblys="A.dll; B.dll" / >

# 第二章 命名

## 2.1 代码命名规则

命名空间：ControlEase.IoDrive.Manufacture

开发类命名规则：XXXDeviceDev（XXX代表设备型号或加上其它区分的信息）

运行类命名规则：XXXDevice

资源命名规则：Resources.resx（英文资源文件），Resources.zh-CN.resx（中文资源）

程序集命名规则：CE.IoDrive.Manufacture.XXX.dll（XXX含义：提供本驱动程序区别于其它驱动程序的名称，不局限于型号,接口类型或协议名称的组合等规则，只要能区别于其它驱动程序便可）

## 2.2 Xml命名规则

文件名称规则：

提供2个XML文件，分别支持中文和英文。

CE.IoDrive.Manufacture.XXX.cfg(中文的配置文件)。

CE.IoDrive.Manufacture.XXX.cfg(英文的配置文件)。

XML文件后缀扩展名备选

dics:Devicedriver Information configuration source。

ddi:DeviceDriverDescriptioninformation。

# 第三章 功能实现

具体设备驱动实现主要根据设备通信协议实现用于IO系统中运行的功能和开发的功能。

## 3.1 运行功能实现

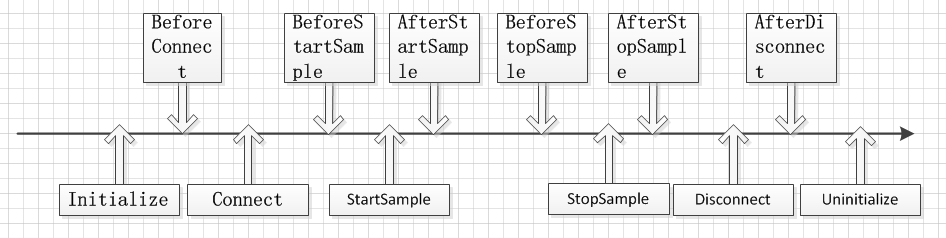


图3-1 通信过程流程图

主要是按照某种通信协议完成读设备寄存器、写设备寄存器、访问设备服务的功能，有些设备为了能正常开始或结束通信，通信前后有一些设备的通信初始化或反初始化的工作，如下图3-1通信过程流程图所示，驱动开发人员根据驱动通信过程的需求，重载适合驱动时机的过程方法。

实现这些功能时，需要从设备驱动运行基类派生，完成基类中规定的各种功能的实现。依据设备通信方式的不同要选择不同的设备驱动基类，选择规则如下表：

|  |  |
| --- | --- |
| **设备的通信方式** | **基类** |
| 使用中间软件通信  （设备厂家提供的API） | NonTransparentDevice |
| 串口 | SerialDevice |
| 以太网 | EthernetDevice |
| USB | USBDevice |

一种设备驱动对应一个设备的一种通信方式，当前设备的通信方式不能满足设备的通信需求时，可以扩展设备的通信方式的基类，比如新开发一个ControlNet通信方式的设备基类：ControlNetDevice类，当开发此驱动时从 ControlNetDevice基类派生实现即可，方便设备通信方式的灵活扩展。当该设备具有多种通信方式时，要为该设备提供多种设备驱动。

### **3.1.1 描述设备的基本特性**

* **设备的通信类型(CommunicationType)**

根据在通信过程中设备处于主动与被动，驱动实现中要指定设备的通信类型。IO系统会根据不同的通信类型而采用不同的通信模式。

对于写寄存器和服务访问来说，设备是被动的，但是对于最常用的获取寄存器值来说，设备可能是主动也可能是被动。我们把设备主动的称为Push，设备被动的称为Poll。综合起来的通信类型为以下几种：

Poll：轮询，Device Driver主动地向Device发起数据采集请求。

TimedPush：定时主动上发，Device定时上发数据

EventPush：事件上发， Device工作中满足一定条件时主动上发数据

TimedAndEventPush ：既有定时主动上发又有事件上发

PollAndPush：既有轮询又有上发

* **获取设备主动建立通信连接的选项（ConnectionOption）**

Identity：获取或设置设备在通信网络中的唯一标识?如IP地址，端口等，设备的实例标识，如DTU上传的登录包中的标识ID，即该设备在网络中区别于其它设备的唯一标识。

IsActiveConnect：获取或设置设备在建立通信时是否主动发起通信连接，站在设备的角度上考虑是否主动。

IsPushIdentity：获取或设置设备是否主动上发标识。

IsNeedGetIdentity：获取或设置是否需要获取设备标识。

* **设备的可配置参数（ConfigData）**

一个设备驱动可满足与现场使用相同通信协议的一个或多个设备的通信，不同的设备其配置可能有不同，故设备驱动中要指定有哪些参数需要配置。

### **3.1.2 实现通信功能**

与设备通信的过程依次为：启动通信的过程，读寄存器/写寄存器/访问服务过程、终止通信的过程，以下逐一介绍每个过程。

由于通信实现方式的不同，从NonTransparentDevice基类派生和从透明设备基类（SerialDevice、EthernetDevice、USBDevice）派生实现的具体内容会有不同。以下具体介绍实现的内容。

#### 设备通信的启动

若采集设备数据前需要与设备进行一些连接等交互工作,则要实现Connect

功能。Connect的实现遵循服务访问的实现方式，如下：

ExecutionResult Connect ( )

{

ExecutionResult result = new ExecutionResult ( );

TaskInfo taskInfo1 = new TaskInfo ( );

CommandInfo cmdInfo1 = new CommandInfo ( ) { Content = new byte [] {0x01, 0x02, 0x03}};

ResponseInfo responseInfo1 = new ResponseInfo ( )

{

ResponseType = ResponseType.L,

Length = 3,

};

taskInfo1.Add ( cmdInfo1, responseInfo1 );

var res1 = base.GetServiceResult ( taskInfo1, "Connect1" );

if ( res1 != null )

result.IsSucceed = ( bool ) res1;

else

{

result.IsSucceed = false;

}

return result;

}

#### 设备通信的终止

若停止数据采集后要断开与设备的连接等交互工作时，则要实现Disconnect功能。Disconnect遵循服务访问的实现方式，如下：

ExecutionResult Disconnect ( )

{

ExecutionResult result = new ExecutionResult ( );

TaskInfo taskInfo1 = new TaskInfo ( );

CommandInfo cmdInfo1 = new CommandInfo ( ) { Content = new byte [] {0x01, 0x02, 0x03}};

ResponseInfo responseInfo1 = new ResponseInfo ( )

{

ResponseType = ResponseType.L,

Length = 3,

};

taskInfo1.Add ( cmdInfo1, responseInfo1 );

var res1 = base.GetServiceResult ( taskInfo1, " Disconnect " );

if ( res1 != null )

result.IsSucceed = ( bool ) res1;

else

{

result.IsSucceed = false;

}

return result;

}

#### 透明设备类通信

IO中支持以同步方式或异步方式与设备通信。同步方式下，设备驱动向设备发送指令并开始等待设备对该指令的反馈，在超时范围内直至接收到所需要的反馈信息为止（如根据协议，没有反馈，则不等待）；异步方式下，设备驱动向设备发送指令，并不等待接收设备对该指令的反馈，而是在指定的时间间隔(DelayTime)后继续向设备发送指令，设备反馈的处理由统一的地方进行。

IO架构解决了驱动开发中同步与异步的区别，驱动开发按照统一的实现规范实现，可同时适用于同步方式和异步方式的通信。按照实现规范，驱动的实现中把读寄存器、写寄存器、服务访问离散为了两个处理过程，即以下要实现的“发命令到设备”和“处理设备反馈”功能。

##### 读寄存器

为得到优化的通信效果，当通过轮询的方式获取寄存器值的情况下，IO中要对读的寄存器进行加工处理，一是根据设备通信协议，按照寄存器的采集周期以及能一次性读写规则对寄存器分组；二是对分组后形成的寄存器组（即SampleGroup），按照通信协议提前处理成发给设备的命令信息和设备返回数据特征的描述信息（即TaskInfo），并缓存下来。

###### 寄存器分组（SampleGroup）

根据协议规则对要读的寄存器分组，主动上发的设备无须分组。分组过程要实现以下方法：

* 根据要读的Register创建一个分组

SampleGroup CreateSampleGroup (Register register)

{

//如果支持复杂数据类型，添加处理逻辑

if（register is ComplexRegister）

｛//Todo；｝

//根据寄存器信息创建采集组

SampleGroup group = new SampleGroup (register) {StartAddress=register.Index, Length = register.Length};

Return group;

}

* 判断某个Register能否添加到某个分组中

Bool CanAddToReadSampleGroup(SampleGroup group, Register register)

{

//如果支持复杂数据类型，添加处理逻辑

if（register is ComplexRegister）｛//Todo；｝

If(group.Length + register.Length< 256)

return true;

}

* 添加某个register到分组中

void AddToReadSampleGroup(SampleGroup group, Register register)

{

//如果支持复杂数据类型，添加处理逻辑

if（register is ComplexRegister）//Todo；

group.Add(register);

group.Length += register.Length;

}

###### 形成命令和反馈信息（TaskInfo）

通信类型为轮询的设备驱动，对要读写的寄存器形成对应的发给设备的命令信息，以及设备对该命令的反馈特征的描述信息。形成读寄存器的命令信息和设备返回数据特征信息，主要实现如下方法：

TaskInfo BuildReadTaskInfo(SampleGroup group)

{

}

通信类型为主动上发的设备驱动，形成设备主动上发数据的特征描述信息PushDataInfo，PushDataInfo从TaskInfo派生，附加了主动上发的特征。主要实现如下方法：

PushDataInfo[] BuildPushDataInfo ( )

｛｝

###### 执行读寄存器命令

通信类型为轮询的设备驱动，按照寄存器的采集周期周期性地取读寄存器命令来执行。执行读命令的过程IO架构提供了默认实现，实现了字节指令串信息的处理及发送到设备，若CommandInfo中的Content不是字节指令串或有特殊的处理逻辑，则要实现以下方法来做特殊处理工作来完成发指令到设备的工作：

ExecutionResult **ExecuteReadCommand**(CommandInfo cmdInfo)

{

//按照CommandInfo的形成规则处理为最终下发给Device的数据

returnBase. ExecuteReadCmd(cmd);//发送数据，添加预期，

}

###### 处理读寄存器反馈

通信类型为轮询的设备驱动，按照协议规则处理设备对上述指令的反馈数据，通过实现以下方法来完成：

ExecutionResult **ProcessReadResponse** ( ProcessInfo info, byte[] data )

{

//

｝

该方法主要是按照协议格式解析设备返回的数据，验证返回数据的正确性以及取得寄存器的值。

对于主动上发方式的设备，通过实现以下方法来完成处理设备主动上发的数据分析工作：

ExecutionResult **ProcessPushData** ( byte[] data )

｛｝

驱动实现中通过方法SetRegistersValue或SetComplexRegisterValue把分析出来的值赋给寄存器，并通过方法的返回值ExecutionResult告诉IO架构该次读寄存器或主动上发是否成功，以及失败的详细信息等。

##### 写寄存器

当NX的用户请求写设备时，IO会收到写设备寄存器的请求，并执行写寄存器的逻辑。IO架构对用户写的寄存器分组形成SampleGroup，并把SampleGroup处理为TaskInfo后加入等待队列中， IO的通信调度逻辑轮到处理该写请求时，从队列中取出该TaskInfo，执行并处理该TaskInfo。

###### 写寄存器分组（SampleGroup）

对于支持支持批量写寄存器值的设备来说，当用户在NX的配方等使用场合请求一次性写多个设备值的情况下，设备驱动中实现批写功能可以更好的满足用户需求。针对支持批写的驱动实现上，要根据协议规则对要写的寄存器分组，形成写的SampleGroup。完成分组过程驱动开发需要实现以下方法：

* 根据要写的Register创建一个分组

SampleGroup CreateWriteSampleGroup (Register register)

{

//如果支持复杂数据类型，添加处理逻辑

if（register is ComplexRegister）//Todo；

//根据寄存器信息创建采集组

SampleGroup group = new SampleGroup (register) {StartAddress=register.Index,Length = register.Length};

Return group;

}

* 判断某个Register能否添加到某个写分组中

bool CanAddToWriteSampleGroup(SampleGroup group, Register register)

{

//如果支持复杂数据类型，添加处理逻辑

if（register is ComplexRegister）//Todo；

If(group.Length + register.Length< 256)

return true;

}

* 添加某个register到写分组中

void AddToWriteSampleGroup(SampleGroup group, Register register)

{

//如果支持复杂数据类型，添加处理逻辑

if（register is ComplexRegister）//Todo；

group.Add(register);

group.Length += register.Length;

}

###### 形成命令和反馈信息（TaskInfo）

对要写的寄存器形成对应的发给设备的命令信息，以及设备对该命令的反馈特征的描述信息。主要实现如下方法：

TaskInfo BuildWriteTaskInfo(SampleGroup group)

{

}

###### 执行写寄存器命令

执行写命令的过程IO架构提供了默认实现，实现了字节指令串信息的处理及发送到设备，若CommandInfo中的Content不是字节指令串或有特殊的处理逻辑，则要实现以下方法来做特殊处理工作来完成发指令到设备的工作：

ExecutionResult ExecuteWriteCommand(CommandInfo cmdInfo)

{

//按照CommandInfo的形成规则处理为最终下发给Device的数据

returnBase. ExecuteWriteCmd(cmd);//发送数据，添加预期，

}

###### 处理写寄存器反馈

按照协议规则处理设备对上述指令的反馈数据，通过实现以下方法来完成：

ExecutionResult ProcessWriteResponse(ProcessInfo info, byte [] data)

{

var result = new ExecutionResult()

if(info.CurrentCmdInfo.Step=0)//写寄存器成功与否

{

//Check response Data;

result.IsSucceed = False;

result.Message = “Failed to write”;

}

else//该写寄存器任务有多个命令，则完善下一个CommandInfo

{// Check response Data;

info.NextCmdInfo.Content = new byte[]{0x01,0x02};

}

Return result;

}

该方法主要是按照协议格式解析设备返回的数据，验证返回数据的正确性。

驱动实现中通过方法的返回值ExecutionResult告诉IO架构该次写寄存器是否成功，以及失败的详细信息等

##### 访问服务

有服务的设备，设备驱动实现对应服务的服务方法，并使用ServiceMethod标记。

###### 服务方法实现

实现一个服务方法的主要过程是：根据该服务的功能和通信协议形成该服务对应的TaskInfo，然后调用GetServiceResult方法等待服务的执行结果即可。

示例如下：

[ServiceMethod]

bool SetConditionTemperature (string value)

{

String mockData = CheckCRC ("01060000"+value, ToCheck.CHECK\_OUT );

TaskInfo taskInfo = new TaskInfo ( );

CommandInfo cmd = new CommandInfo ( ) { Content = mockData };

ResponseInfo resinfo = new ResponseInfo ( )

{ Length = 8, ResponseType = ResponseType.L };

taskInfo.Add ( cmd, resinfo );

var result = GetServiceResult ( taskInfo, "WriteRegister" );

if ( result != null )

return bool.Parse ( result.ToString ( ) );

else return false;

}

###### 执行服务访问命令

执行写命令的过程IO架构提供了默认实现，实现了字节指令串信息的处理及发送到设备，若CommandInfo中的Content不是字节指令串或有特殊的处理逻辑，则要实现以下方法来做特殊处理工作来完成发指令到设备的工作：

ExecutionResult **ExecuteServiceCommand** ( string serviceId, CommandInfo cmdInfo )

｛｝

###### 处理服务访问反馈

按照协议规则处理设备对上述指令的反馈数据，通过实现以下方法来完成：

ExecutionResult **ProcessServiceResponse** ( ServiceProcessInfo info, byte[] data )

｛｝

该方法主要是按照协议格式解析设备返回的数据，验证返回数据的正确性，并返回服务访问结果。

驱动实现中通过方法的返回值ExecutionResult告诉IO架构该次服务访问是否成功，以及失败的详细信息等；并调用SetServiceResult返回服务访问结果，供服务访问方法中使用。

注意：IO架构内置了两个默认服务方法：“Read”和“Write”，这两个服务方法IO架构已经默认实现，但按照协议格式解析数据和验证数据的正确性等工作，需要具体驱动程序提供支持，需要在“**ProcessServiceResponse**”处理“Read”和“Write”的反馈，结合“3.1.3”部分内容。实例代码如下：

ExecutionResult ProcessServiceResponse ( ServiceProcessInfo info, byte[] data )

｛

if ( info.ServiceId == "Read" )

{

var regValues = ExplainReadServiceIO ( info.SampleGroup, data, ref result );

SetServiceResult ( new ServiceResult ( ) { ServiceId = "Read", Result = regValues } );

}

else if ( info.ServiceId == "Write" )

SetServiceResult ( new ServiceResult ( ) { ServiceId = "Write", Result = "true" } )

｝

##### 其他功能

###### 分析设备标识数据

在描述设备基本特性时，若设备特性满足如下条件：1.设备主动发起通信连接；2.设备主动上发其标识数据；需要重载实现如下方法：

string AnalyzeIdentityData ( byte[] idData, out byte[] backData )

其中参数“idData”是需要解析的设备标识数据；“backData”是设备上发标识数据后给设备回馈的数据，若设备上发标识数据后不需要给其回馈数据则置为空值；

备注：在解析设备标识数据时，如果解析出来的标识数据同驱动配置数据相同则返回标识，否则返回字符串空值。

###### 构建获取标识命令

在描述设备基本特性时，若设备特性满足如下条件：1.设备主动发起通信连接；2.设备不主动上发其标识数据；3.需要给设备发送命令来查询其标识；这时需要重载实现如下两个方法：

string AnalyzeIdentityData ( byte[] idData, out byte[] backData )；该方法实现参照“分析设备标识数据”所描述内容；

byte[] BuildGetIdentityCommand ( )，用来获取设备标识的命令，根据设备通信协议要求构建该命令的字节数组并返回。

#### 非透明设备类通信

##### 处理采集内容

对采集内容的处理，包含两部分内容。一是将要采集内容形成SampleGroup，形成SampleGroup过程同“透明设备类通信”相同；二是根据SampleGroup形成设备命令信息，对于非透明设备类通信而言，形成设备命令信息主要是将SampleGroup处理成易于调用设备厂家PAI的信息，具体内容由驱动开发根据API的特点定义。

##### 执行数据采集

* **实现读寄存器**

protected override ExecutionResult ExecuteReadCommand(CommandInfo cmdInfo)

{

ExecutionResult result = new ExecutionResult();

// TODO: 根据“cmdInfo”的 Content内容调用设备厂商API

// 分析和解析厂家API返回的数据，调用基类“SetRegistersValue”给寄存器赋值

SetRegistersValue(values);

// 根据厂家API返回结果设置读取操作是否成功

SetCommandExecutionResult(cmdInfo, Condition ? true : false);

return result { IsSucceed = Condition ? true : false };

}

* **实现写寄存器**

protected override ExecutionResult ExecuteWriteCommand(CommandInfo cmdInfo)

{

ExecutionResult result = new ExecutionResult();

// TODO: 根据“cmdInfo”的 Content内容调用设备厂商API

// 分析和解析厂家API返回的数据，设置写入设备数据是否成功

SetCommandExecutionResult(cmdInfo, Condition ? true : false);

return result { IsSucceed = Condition ? true : false };

}

* **实现服务访问**

protected override ExecutionResult ExecuteServiceCommand(string serviceId, CommandInfo cmdInfo)

{

ExecutionResult result = new ExecutionResult ( );

if ( serviceId == "Read" )

{

// 调用厂家相应的API

}

else if ( serviceId == "Write" )

{

// 调用厂家相应的API

}

else if ( serviceId == "……" )

{

// 调用厂家相应的API

}

// 根据服务方法执行情况设置返回值

SetServiceResult ( new ServiceResult ( ) { ServiceId = serviceId, Result = values } );

SetCommandExecutionResult ( cmdInfo, Condition ? true : false );

return result { Condition ? true : false };

}

##### 实现设备的服务

同“透明设备类通信”的驱动

##### 启动设备通信

同“透明设备类通信”的驱动

##### 结束设备通信

同“透明设备类通信”的驱动

### **3.1.3 立即读/写功能**

IO系统为用户提供了立即从设备中获取IO变量值以及立即写值到设备中的功能，这两个功能在架构级别被处理为方法名为“Read”、“Write”的两个方法，这两个方法的实现原理与服务访问的处理原理相同，这两个方法的实现需要驱动开发的支持。故设备驱动若支持寄存器的读或写，则驱动开发中要处理一些实现，不支持时不需要处理。具体需要处理的内容为：

在上述的“发命令到设备”中处理“Read”和“Write”这两个方法的发命令功能，在上述的“处理设备反馈”中处理这两个功能的命令对应的反馈，在获取到结果后，对于“Read”使用**GetServiceResult**方法返回**Tuple<object, int, DateTime>**类型的寄存器值结果，对于“**Write”**返回“**bool”**类型标识写成功或失败的结果。

### **3.1.4 提供NX变量值功能**

当用户通过如串口等通信方式、使用某种通信协议(如Modbus等)获取NX工程中的变量值时，IO支持开发一个NX的设备驱动的方式来满足用户需求。

开发该类驱动时，可使用Register类中的如下方法获取NX中变量的值：

public ITag GetCurrentValue ( )

### **3.1.5 设备寄存器排序**

在实现设备寄存器分组时，底层架构已经默认按照寄存器 “Index”属性从小到大顺序进行了排序。如果设备驱动想实现自己的排序规则，需重载基类的SortRegisters方法：

protected override void SortRegisters ( IList<Register> registers )

{

// 实现自己的排序功能

}

*备注：只有寄存器的“Index”能够转换成整形的情况下才进行排序。*

## 3.2 开发功能实现

具体设备驱动实现开发类时从DeviceDev类派生，主要提供如下功能：

必需要实现设备属性配置的wpf控件(wpf界面必须要对NX的多语言提供支持)，寄存器相关信息，创建寄存器，获取寄存器信息（读写特性，列信息，寄存器支持列表，支持的数据类型），检查寄存器配置的有效性，以及设备本身是否支持复杂类型数据；ConfigData的保存与获取。

### 3.2.1 通信配置

#### 1）配置数据

IO中设备配置数据由**三部分**构成：

* 所有设备驱动都有的配置数据；
* 以以太网/串口等分类标准的设备都有的配置数据（如：所有串口设备均有的配置数据为站号，所有的以太网设备均有的配置数据为IP地址、端口号、Client/Server、协议类型）
* 具体设备驱动特有的配置数据（ConfigData）。

IO开发根据设备驱动的通信口类型InterfaceType具体化前两部分配置数据，如具体化为“SerailDeviceCfgData/EthernetDeviceCfgData”等。

IO架构默认提供了第二部分数据的管理和配置工作，但对于某些设备这些数据是固定的无需用户配置的，或者只需配置部分数据，这时就不能使用IO架构默认提供的配置的处理，驱动根据设备的情况处理这部分数据的配置相关工作，具体规范见下述的“CommCfgData”节。

一般情况下，一个设备驱动可满足与现场使用相同通信协议的一个或多个设备的通信，不同的设备其通信配置可能不同，故设备驱动要支持对通信参数的配置，也即设备驱动要处理对特有的配置数据的支持。

若驱动有特有的配置数据则需要处理ConfigData的相关内容。关于ConfigData的具体处理规范详见以下的“ConfigData”小节如下。

#### 2）CommCfgData

UseCommCfgDataCtrl属性：默认为True,驱动开发通过该属性告诉IO架构该驱动公共的配置数据是否使用底层提供的通用配置控件，若底层提供的通用控件不满足要求，具体驱动在“CofigCtrlType”类型的控件中处理公共数据的配置逻辑和配置界面。

#### 3）ConfigData

具体实现时要重载ConfigData属性，如无特有配置返回null，如下

Private FX2NCfgData mConfig=null;

Public override object ConfigData

{

get { return mConfig; }

set { mConfig = value; }

}

FX2NCfgData类用于定义FX2N这类设备特有的配置数据。

* **ConfigData的序列化**：

当ConfigData的数据类型为基础类型的数据时（比如bool，string，int等类型），IO架构中已支持了对ConfigData数据的保存和加载。但如果ConfigData的定义中存在自定义的数据类型（如自定义类）需要保存和加载时，驱动开发可以做以下两种方式的工作来完成自定义类型的加载和保存：

* 自定义类型的属性上使用TypeConverterAttribute标记对应的Converter；
* 重写以下方法，完成对自定义类数据的保存和加载工作：

Protected override void LoadCfgDataFromElement(XElement element)

{

Return;

}

Protected override XElement SaveCfgDataToXElement()

{

return new XElement(“”);

}

* **用户配置控件**

提供用户对ConfigData数据操作的用户界面。该用户界面在IO开发环境中两个地方使用：一个是从驱动工具箱中选择驱动新建设备时弹出的对话框中，另一个是PropertyGrid中该设备的属性配置对话框中。

驱动开发对ConfigData的用户配置界面实现为一个WPF控件，并通过以下方式告诉IO架构该控件的类型，具体为重载CofigCtrlType属性：

Public override Type CofigCtrlType

{

get { return typeof(MelsecEnetConfigControl); }

}

该控件类要提供一个带参数的构造函数，以便获取到控件要修改的数据对象即上述的ConfigData，具体如下：

/// <summary>

/// Interaction logic for CfgCtrl.xaml

/// </summary>

public partial class CfgCtrl : UserControl

{

/// <summary>

///

/// </summary>

public CfgCtrl ( )

{

InitializeComponent ( );

}

/// <summary>

///

/// </summary>

/// <param name="configData"></param>

public CfgCtrl ( object configData ) : this ( )

{

//若采用MVVM的方式，此处实例化DataContext为对应的ViewModel

//若不采用MVVM的方式，可以采用其它方式

DataContext = new CfgDataViewModel ( ( OPCDeviceConfigData ) configData );

}

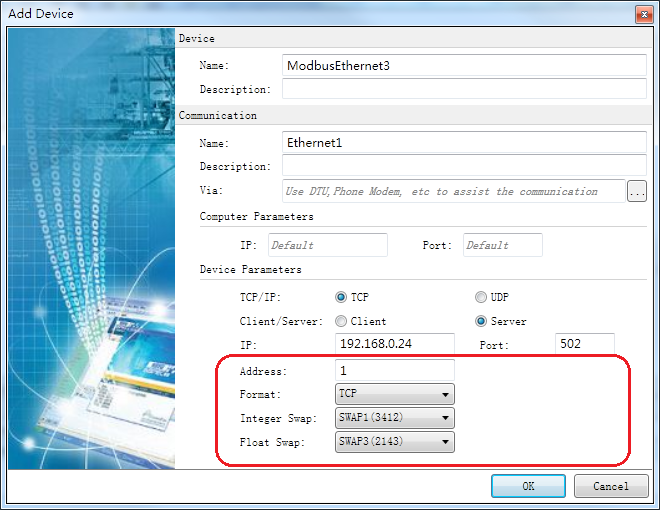
}

属性配置对话框中配置完的结果若要显示在PropertyGrid中，ConfigData中要重载ToString方法，返回要显示的内容，若未重载则显示为空。

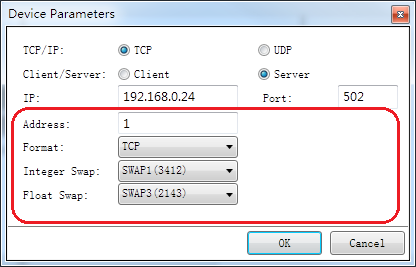
* **配置界面规格**

ConfigData数据操作的用户界面会在以下两个地方被集成到IO开发环境中：

1、新建设备时弹出的添加社区对话框



2、属性页中该设备的属性配置对话框。



为美化对话框布局，ConfigData对应的WPF控件采用四列布局，设置第一列Width=100，第二列Width=140，第三列Width=80，第四列Width=Auto，每列不要设置Margin属性。第一列和第三列一般为标题列，第二列和第四列为内容列，第二列中的TextBox、Combox等输入类控件设置Width=120；第四列中的TextBox、Combox等输入类控件设置Width=60。具体呈现效果可在中英文环境下微调。

### **3.2.2 寄存器**

寄存器代表了设备中的某个地址和存储在这个地址的值，为了描述设备中的某个地址的值，IO中抽出了Register类来描述这个地址及其它信息。描述地址的信息有如下表：

|  |  |
| --- | --- |
| **属性** | **功能说明** |
| NetworkNo | 寄存器所在Device所处的控制网络的编号 |
| DeviceNo | 寄存器所在Device在所处控制网络(如CCLink)中的站号/地址号 |
| Memory | 寄存器所在的区或块 |
| Index | 在区中的主地址标识（如字节） |
| SubIndex | 在主地址的下一级地址(如位) |
| Length | 从地址起连续几个字节或几个位 |

#### 列

上述描述Register地址的信息有6个，驱动根据具体情况来选用。上述每个信息会对应为IO开发中的一个列。

IO架构默认提供Region，Index，SubIndex，Length共四列信息，其中Region列为下拉列表样式，其它列为能输入文本的列样式。如果默认提供不能满足具体驱动的寄存器列的需求，则重写GetColumnInfos方法，该方法返回驱动定义的寄存器列信息（定义为ColumnInfo类），包括几列，编辑格式及列显示名称（显示名称支持多语言，在cfg文件中描述多语言信息）。当IO架构定义的列样式能满足驱动要求时，一般不用重写此方法：

Protected override ColumnInfo[] GetColumnInfos()

{

return new ColumnInfo[2]();

}

**1）ColumnInfo类构成：**

* **ColumnType**：寄存器列标识，最多提供6列，该6列是按照当前主流PLC抽象出来的，具体为：NetworkNo（网络号），DeviceNo（设备号），Memory（分区），Index（一级地址），SubIndex（二级地址），Length（长度）。
* **ColumnStyle**：列的编辑样式，有TextBox（文本框）、Combox（下拉框），CustomDropDown(自定义下拉)，CustomDialog（自定义对话框）自定义样式
* **EditorType：**ColumnStyle为CustomDropDown时要提供自定义的列编辑控件，该属性描述自定义编辑控件的类型，控件的实现参照以下的“**Editor实现”**。
* **IsEditableComboBox:**当ColumnStyle设置为“Combox（下拉框）”时表示该ComboBox是否是可编辑的，True表示可编辑；False表示不可编辑。

**2）Editor实现**

* 当ColumnStyle为CustomDropDown时，实现为WPF控件。控件要提供一个带参数的构造函数，以便获取到要编辑的数据，以自定义Memory列编辑控件为例，具体如下：

/// <summary>

/// Interaction logic for MemorySelectorView.xaml

/// </summary>

public partial class MemorySelectorView : UserControl

{

/// <summary>

///

/// </summary>

public MemorySelectorView ( )

{

InitializeComponent ( );

}

/// <summary>

///

/// </summary>

/// <param name="device">设备驱动开发类实例</param>

/// <param name="currentMemeory">Memroy列的当前值</param>

public MemorySelectorView ( DeviceDev device, Register register )

: this ( )

{

//若采用MVVM的方式，此处实例化DataContext为对应的ViewModel

//若不采用MVVM的方式，可以采用其它方式

var dev = device as OPCDeviceDev;

DataContext = new MemorySelectorViewModel ( dev, register);

}

}

用户编辑结束时，控件要告诉IO架构选择的Memory的结果，以便IO架构做后续的处理，可通过设备驱动开发类中提供的UpdateValue方法通知IO架构。

* 当ColumnStyle为CustomDropDown时，此时可以自定义边界界面，编辑界面在一个单独的窗口中，驱动开发在该窗口对应的处理逻辑中处理界面的功能，此时要实现以下方法来呈现对应的窗口：

/// <summary>

/// ColumnStyle为CustomDialog时，重载该方法处理编辑对应列的编辑界面等

/// </summary>

/// <param name="register"></param>

/// <param name="column"></param>

public virtual void EditColumn ( Register register, ColumnType column )

{

if ( column== ColumnType.Memory )

{

var view=new EditMemoryView(register);

view.ShowDialog();

}

}

该对话框关闭时，要告诉IO架构选择的Memory的结果，以便IO架构做后续的处理，可通过设备驱动开发类中提供的UpdateValue方法通知IO架构。

#### 建立寄存器

开发环境下单击Add按钮的执行逻辑。IOS提供一种默认的方式实现，默认的实现方式是分2种情况：

1、当寄存器页面无寄存器时，从支持的寄存器分区表中取出第一个，并获取相应的读写方式信息，把Index地址和SubIndex设置为默认的0，单元长度设置为1。

2、当寄存器页面有寄存器时，获取鼠标焦点的当前行寄存器或最后一次建立的寄存器）的信息，创建新的寄存器。

一般情况下不用重写此方法，如果默认实现不能满足驱动需求则需重写此方法。

public override Register CreateNewRegister ( Register lastRegister )

{

Register newRegister=null;

if ( lastRegister==null )

//todo开发环境下驱动寄存器页面无寄存器时，单击Add按钮的执行逻辑

else

//todo根据上一个建立的寄存器（当前选中行寄存器或最后一次建立的寄存器）的信息，创建新的寄存器。

return newRegister;

}

#### 分区种类

当前主流PLC中一般有多个分区供通信访问，如三菱PLC的M、D等分区，对于其它设备，如智能仪表等设备可能没有分区/寄存器区（Memory）的概念，但驱动开发中可以为其自定义。

驱动开发要返回支持的寄存器分区种类，供用户选择，具体实现方式如下：

Public override IList<string> GetSupportRegionTypes()

{

Throw new NotImplementedException();

}

#### 地址进制转换

Register的主地址Index的值，用来表示设备中的某个地址，在IO开发环境中支持显示为多种进制格式。

若Index不支持进制转换，如不是数字来表示地址的，则驱动开发中重载以下方法禁用IO开发中的进制转换功能：

/// <summary>

/// 是否支持多种地址格式（十进制、十六进制、二进制、八进制），默认为True

/// </summary>

public virtual bool SupportAddressFormat

{

get

{

return true;

}

}

#### 值类型

若设备中的寄存器值有类型，则在建立寄存器时，驱动开发使用Register中的CanonicalDataType属性来指定值的类型，IO开发会依据该值来建立对应类型的IO变量，若不指定CanonicalDataType，则默认建立整型IO变量。

#### 访问特性

必须要实现的方法，新建寄存器或更改寄存器的类型时，IOS架构中会根据用户选择的寄存器类别，匹配读写特性。比如，像AB的PLC这种情况，用户输入的寄存器，读写方式应怎么获取呢？可以重写此方法，在函数中添加逻辑。

Protected override ReadWriteMode GetSupportReadAndWriteMode ( string region )

{

//TODO ，根据寄存器的名称，分配默认的读写方式，并返回读写方式。

return ReadWriteMode.ReadAndWrite;

}

#### 信息检查

检查参数的配置是否有效，供开发环境下单击Check按钮或编译时调用。IO架构提供默认的方式支持读写方式检查、地址冲突检查及范围越界检查。如果默认的检查内容不能满足驱动的检查需求，可以重写如下方法实现某些检查逻辑：

Public override ErrorMessage[] CheckRegister ( Register register )

{

var errorList = new List<ErrorMessage> ( 8 );

//TODO，校验此register是否正确，返回所有错误列表。

return errorList.ToArray ( );

}

}

#### 特殊

* 寄存器信息间的关联处理

Register的列信息间有关联关系，如SNMP驱动中，Memory和OID列是同一个信息的两种表达方式，Memory是易读的字符串，OID是数字组成的字符串，本质上是在标识一个寄存器地址，驱动开发中处理为Register的Memory列和OID列这两列来显示，此时用户改变其中一个的值（如OID），界面上要求更新另一个的值（如Memory）。

调用以下方法完成用户改变Memory后的相关联列值更新逻辑：

/// <summary>

/// 重载该方法以处理用户修改Register的某个信息时的处理逻辑

/// </summary>

/// <param name="register">当前编辑的Register对象</param>

/// <param name="column">当前编辑Register的列</param>

public virtual void ProcessAfterValueChanged ( Register register, ColumnType column )

{

}

### **3.2.3 复杂数据类型**

设备寄存器分区是否支持复杂数据类型（Struct/Array），如果支持复杂类型，则驱动通信部分必须要处理复杂寄存器的读写，同时重写如下方法：

Public override bool IsSupportComplexDataType(string region)

{

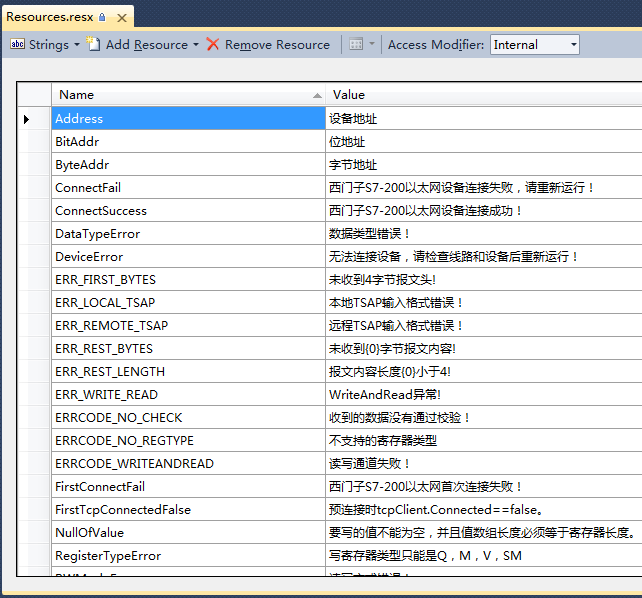
Return true; //如果设备支持复杂数据类型则返回true。

}

### **3.2.4 资源文件**

Device Driver内部中配置属性控件文本以及通讯过程提示信息文本的多语言支持，默认由程序的Resources.resx文件提供，此部分规范符合NX下多语言的规范，如果支持其它语言则提供不同的Resource文件，默认情况下，Device Driver应提供2种语言支持，Resources.resx文件提供默认语言的资源支持，Resources.zh-CN.resx将提供中文的资源支持。

Resources.resx文件中需要提供设备配置属性文本的多语言支持，对设备属性的配置页面的文本信息提供翻译如Address翻译成地址，SlotNumber翻译成槽号等，如下图所示。如果设备通信中过程中Device Driver有提示信息给用户，则也需要添加到资源列表中，供系统多语言调用。



## 3.3 软件日志及工程信息

为了设备驱动本身的调试、发现问题和升级的需要，NX软件可以记录驱动运行过程中的异常等重要调试信息（不暴露给用户），即软件日志，驱动软件日志统一记录到NX的软件日志中。

同时设备驱动也可能在工程开发过程中(如通信测试)输出一些严重的错误信息，如无法连接到设备、无法连接到远程OPCServer等等。

IO架构中统一提供了记录输出工程消息和软件日志的接口，驱动开发可通过使用CE.AI.ICF程序集中的Logging接口来实现，Logging提供如下方法。

|  |  |
| --- | --- |
| **方法** | **功能说明** |
| Debug | 输出驱动软件日志信息 |
| Warn | 输出警告信息到NX消息框中 |
| Error | 输出错误信息到NX消息框 |
| Info | 输出信息到NX消息框 |

# 第四章 常用术语

|  |  |  |
| --- | --- | --- |
| **序号** | **名词** | **名词定义** |
| 1 | IO系统  （Input/OutputSystem） | 运行在PC中，实现现场设备数据采集/控制/管理等功能的软件程序,目前阶段IO系统只作为INSPEC NX的一个子系统，由IO开发和IO运行构成，IO开发完成IO系统的通信组态，IO运行按照IO组态的要求完成与线程各种设备的通信。 |
| 2 | 设备  （Device） | 代表一种硬件/软件，IO系统能够对这种硬件/软件进行读取/写入/服务访问等通信操作，即Device是IO系统通信的目标对象。如PLC、仪器/仪表、OPC服务器等。 |
| 3 | 设备寄存器  （Device Register） | 设备中的数据，该数据可以是没有类型的原始值（如三菱PLC中的X、Y等分区数据），也可是有数据类型的变量（如OPCServer中的变量、AB Logix5000PLC中的变量）。IO系统中以IoTag方式进行访问。 |
| 4 | 设备服务  （Device Service） | 设备中的功能，IO系统中以服务方式访问的功能。 |
| 5 | 设备驱动  Device Driver | IO系统中的一种软件程序，负责按照一种通信协议实现与Device的通信，以获取或修改设备寄存器的值，或访问设备服务。不同的设备驱动与设备所采用的通信协议相关。 |
| 6 | Register | 描述设备寄存器的类 |
| 7 | 采集组  (SampleGroup) | 寄存器分组类：固定采集周期的、按照通信协议能一次性读取的设备寄存器的集合。 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |