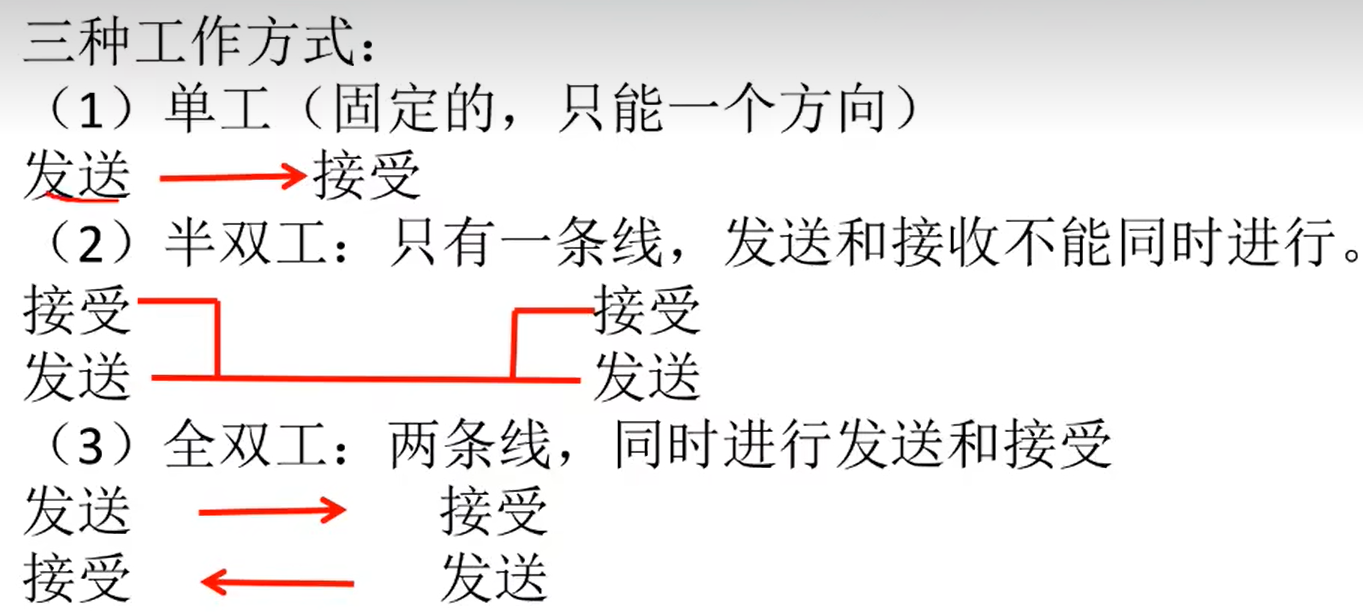
# 通信协议

## 通信学习1

### 1、串口通信参数设置

#### Rs-485通信：

9个接口，1，2，3，4，5为A+；6，7，8，9为B-；一般就是用两个接口：特点传输，半双工：交换数据不能同时发送，不能同时接受



#### 串行通信数据流和格式：

原理：串行通信方式是将字节（byte）拆分成一个接一个的位（bit）后，再传输出去，收到数据的一方，再将这些位组合成原来的字节。

要求：传输格式必须统一，传输方式（节拍）必须一致（速度一致）

波特率：表示每秒传输的位的个数，也就是每秒传输0和1的个数，常见是9600b/s,4800b/s,19200b/s,38400b/s

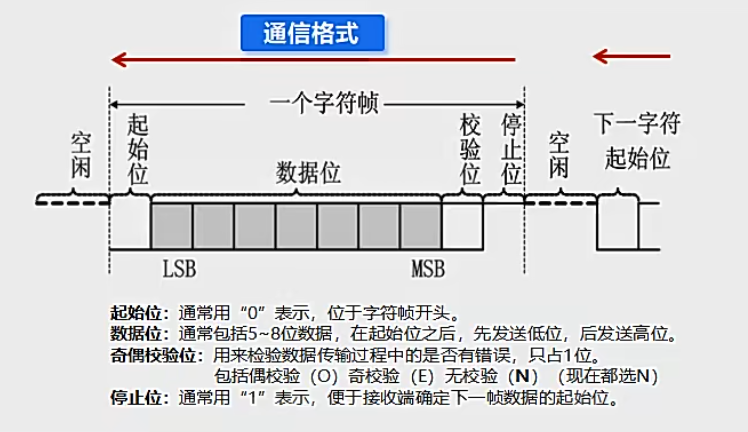
串行通信格式（单字节协议）

常见：（9600，N,8,1）波特率9600b/s，无奇偶校验位，8位数据位，1位停止位。

#### 通信格式：

起始位：判断是否要发送数据，一般0开头

必须存在间隔



### 2、modbus协议基础

#### 关于通信协议：

不同设备之间交换数据要遵循的规范，就好比人与人之间交流用的语言，必须要有语法。通常情况下，串行通信遵守Modbus协议，串口232，422，485都是按照Modbus协议，但是Modbus也可也用于以太网的TCP/IP通信。

#### Modbus与串口关系：

串口属于物理接口，属于硬件。

Modbus是国际标准的串行通信协议，是软件。

#### Modbus与串行通信关系：

串行通信有规定的格式，所以如何识别传送的多个字节和解析标准怎么规定？

Modbus：就是如何用串口一次连续传输多个有序字节的协议。它规定一次能传输多少个字节，以及字节的顺序如何排列。

#### Modbus网络传输的三种模式：

1. ASCII模式：MG标准信息交换码（0-9，a-z，A-Z），数据中每8个字节都用ASCII码发送。
2. RTU模式：远程终端单元模式通信，针对通信距离较长和工业环境恶劣而设计的痛惜结构。特点：消息中每8个bit的字节都包含4bit的十六进制符。
3. TCP模式：通过以太网和互联网连接传输数据使用的TCP/IP协议，简称TCP模式，硬件接口就是以太网交接口。

Modbus协议基础

3.5字符：时间间隔4ms

地址域：一个字节用来定位设备

功能码：一个字节，指定存储区域的操作类型（指定存储区域，表明是读取数据，还是写入数据）

数据域：传输数据啊



### 3、数据存储区和功能码

#### 线圈和寄存器

线圈：在电气控制中，接触器和继电器都是靠得电和失电来控制断开和闭合，用线圈来表示，寄存器在计算机中就是用来存储数据，所以非布尔量的数据用寄存器存储。

读写：以西门子plc为例，I表示输入，Q表示输出，输入意味着该存储区里的值是外部设备接入，是只读的，Q是输出表示输出结果给外部设备，所以可以读写。







我们一般人为沟通交流和在说明书中使用绝对地址；

在Modbus协议中，我们使用相对地址来表明某个区域，通过功能码来区分，是那个区域实现那种功能。

#### 寄存器设备地址（硬件地址）：

就是PLC硬件芯片中对应的地址，一般是十进制，一般是五位，第一位表明寄存器类型。

寄存器Modbus协议地址：指通信使用的寄存器寻址地址

编写代码时，我们使用的是Modbus的协议地址，但是我们最后通信到硬件上时，还是寄存器设备地址。

是通过Modbus协议通信时规定的功能区，来区分使用的Modbus地址对应设别地址中的哪一个寄存器。

#### 功能码：



保持—输出

### modbus通信报文分析

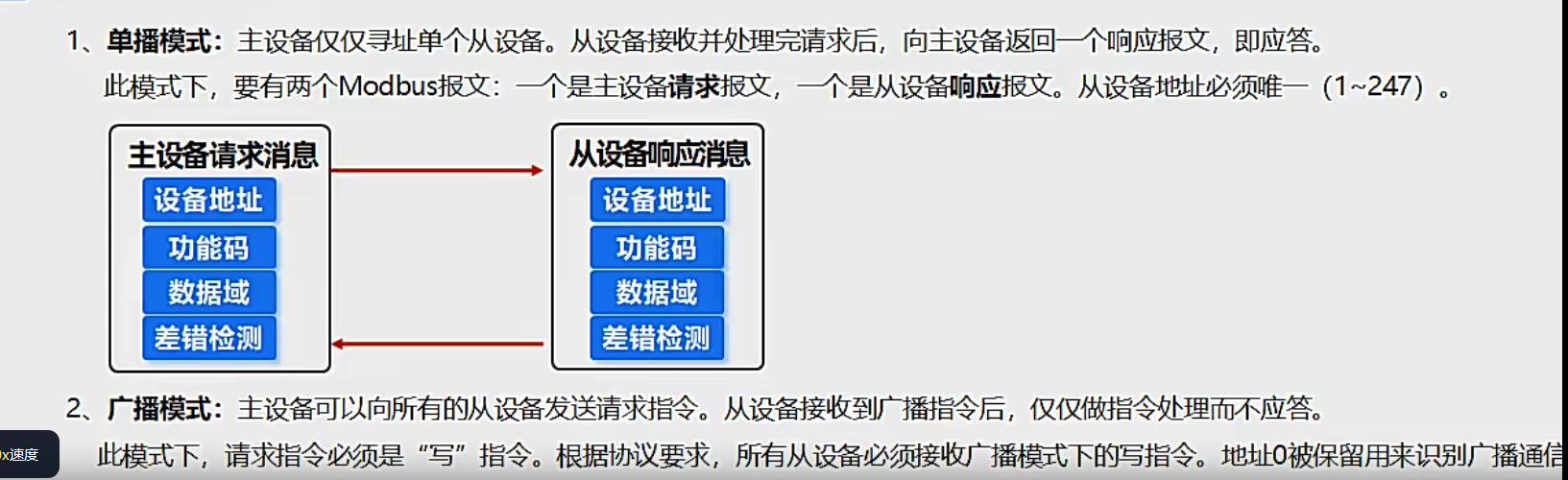
Modbus协议：是一种单主/多从的通信协议

特点：同一时间总线上只能有一个主设备，但是可以有一个或者多个(247)从设备

规定：通信只能由主设备发起，从设备没有收到主设备请求时，不会主动发送数据。

注意：设备之间不能相互通信。(从站之间不能相互传输数据)

寄存器就是一个字，一个字=两个字节





起始寄存器==》Modbus协议地址

### 主-从通信仿真

### c#通信库编写和测试

1. 建立ModbusRTU通信类库
2. 建立类库的构造函数
3. 创建串口对象，SerialPort
4. 创建接受报文的延迟方法
5. 创建连接方法Connect，其中要让串口对象绑定串口名PortName，波特率BaudRate，奇偶校验Parity，数据位Int DataBits（是几位），停止位StopBits；最后打开通信，建立关闭连接的方法DisConnect

**6、需要引入一个库Thinger，数据转换库**

### 7、ModbusRTU的通信

#### 1、读取输入，输出线圈0X01,0X02

1）、拼接报文，创建List<byte> list对象，依次添加从站地址，功能码，初始线圈地址（十进制/256，十进制%256）,线圈数量+CRC校验

2）、发送，接受报文，写成一个通用的方法。首先serialPort.Write（发送报文字节数组，0，发送报文字节数组长度）；延时Thread.Sleep（1000）接受报文；创建接受报文时间，建立MemoryStream内存对象，创建一个Byte字节容量；循环接受返回报文，判断BetysToRead是否存在返回报文的字节的数量，将其一次次添加到MemoryStream内存中，最后没有BytesToRead之后，全部写入给放回字节数组

3）验证发送，接受是否成功，验证报文是否正确，解析报文

ModbusRTU，读取输出0X01，读取输入0X02线圈方法

//第一步，拼接报文

//第二步：发送报文

//第三步：接收报文

//第四步：验证报文

//第五步：解析报文

/// <summary>

/// 读取输出线圈

/// </summary>

/// <param name="slave">从站地址</param>

/// <param name="startCoil">初始线圈地址十进制</param>

/// <param name="length">线圈的数量十进制</param>

/// <returns></returns>

public byte[] ReadOutPutCoils(byte slave,ushort startCoil,ushort length)

{

//第一步拼接报文

List<byte> sendReadOutPutCoils=new List<byte>();

//从站地址

sendReadOutPutCoils.Add(slave);

//功能码

sendReadOutPutCoils.Add(0X01);

//初始线圈地址

sendReadOutPutCoils.Add(Convert.ToByte(startCoil/256));

sendReadOutPutCoils.Add(Convert.ToByte(startCoil%256));

//线圈数量

sendReadOutPutCoils.Add(Convert.ToByte(length/256));

sendReadOutPutCoils.Add(Convert.ToByte(length%256));

//此时需要获得CRC校验

byte[] sendCRC = CRC16(sendReadOutPutCoils.ToArray(), sendReadOutPutCoils.Count);

sendReadOutPutCoils.Add(sendCRC[0]);

sendReadOutPutCoils.Add(sendCRC[1]);

//第二步,发送报文

//第三步，接受报文

byte[] receive = null;

//验证发送接受报文是否成功

if( SendAndReceive(sendReadOutPutCoils.ToArray(),ref receive))

{

//线圈长度

int coil = length%8==0 ? 1 : length/8+1;

//第四步验证报文

if (CheckCRC(receive)&&receive.Length==5+coil)

{

//第五步，解析报文

byte[] buffer = new byte[coil];

Array.Copy(receive, 3, buffer, 0, buffer.Length);

return buffer;

}

}

return null;

}

#### 2、读取输出，输入寄存器

/// <summary>

/// 读取输出寄存器

/// </summary>

/// <param name="slaveid">从站地址</param>

/// <param name="start">初始寄存器地址</param>

/// <param name="bytelength">寄存器数量</param>

/// <returns>返回数据</returns>

public byte[] ReadOutPutRegisters(byte slaveid, ushort start, ushort bytelength)

{

//拼接报文

List<byte> sendReadOutPutRegisters = new List<byte>();

sendReadOutPutRegisters.Add(slaveid);

sendReadOutPutRegisters.Add(0x03);

sendReadOutPutRegisters.Add(Convert.ToByte(start/256));

sendReadOutPutRegisters.Add(Convert.ToByte(start%256));

sendReadOutPutRegisters.Add(Convert.ToByte(bytelength/256));

sendReadOutPutRegisters.Add(Convert.ToByte(bytelength%256));

//CRC校验码

byte[] CRC = CRC16(sendReadOutPutRegisters.ToArray(), sendReadOutPutRegisters.Count);

sendReadOutPutRegisters.AddRange(CRC);

//发送接收报文

byte[] receive = null;

if (SendAndReceive(sendReadOutPutRegisters.ToArray(), ref receive))

{

int receivelength = bytelength\*2;

byte[] registers = new byte[receivelength];

//验证报文

if (CheckCRC(receive)&&receive.Length==receivelength)

{

if (receive[0]==slaveid&&receive[1]==0x03&&receive[2]==receivelength)

{

//解析报文

Array.Copy(receive, 3, registers, 0, registers.Length);

return registers;

}

}

}

return null;

}

/// <summary>

/// 读取输入寄存器

/// </summary>

/// <param name="slaveid">从站地址</param>

/// <param name="start">初始寄存器地址</param>

/// <param name="bytelength">寄存器数量</param>

/// <returns>返回数据</returns>

public byte[] ReadInPutRegisters(byte slaveid, ushort start, ushort bytelength)

{

//拼接报文

List<byte> sendReadOutPutRegisters = new List<byte>();

sendReadOutPutRegisters.Add(slaveid);

sendReadOutPutRegisters.Add(0x04);

sendReadOutPutRegisters.Add(Convert.ToByte(start/256));

sendReadOutPutRegisters.Add(Convert.ToByte(start%256));

sendReadOutPutRegisters.Add(Convert.ToByte(bytelength/256));

sendReadOutPutRegisters.Add(Convert.ToByte(bytelength%256));

//CRC校验码

byte[] CRC = CRC16(sendReadOutPutRegisters.ToArray(), sendReadOutPutRegisters.Count);

sendReadOutPutRegisters.AddRange(CRC);

//发送接受报文

byte[] receive = null;

if (SendAndReceive(sendReadOutPutRegisters.ToArray(), ref receive))

{

int receivelength = bytelength\*2;

byte[] registers = new byte[receivelength];

//验证报文

if (CheckCRC(receive)&&receive.Length==receivelength)

{

//解析报文

if (receive[0]==slaveid&&receive[1]==0x03&&receive[2]==receivelength)

{

Array.Copy(receive, 3, registers, 0, registers.Length);

return registers;

}

}

}

return null;

}

#### 3、预置单线圈和单寄存器

**此时，需要注意需要将数据写入在发送报文中，将线圈置位1，为0XFF,0X00，复位0，为0X00,0X00；而且此时只需要验证发送的报文和接受的报文是否一致，就可以判断发送是否成功。对于预置单寄存器，所需要考虑的是，输入的数据可能是bool数组，可以是有符号整型和无符号整型**

/// <summary>

/// 预置单线圈

/// </summary>

/// <param name="slaveid">从站地址</param>

/// <param name="start">线圈地址</param>

/// <param name="value">线圈数据</param>

/// <returns>是否成功</returns>

public bool PreSetSingleCoil(byte slaveid, ushort start, bool value)

{

//拼接报文

List<byte> send = new List<byte>();

send.Add(slaveid);

send.Add(0X05);

send.Add(Convert.ToByte(start/256));

send.Add(Convert.ToByte(start%256));

send.Add(value ? (byte)0xFF : (byte)0x00);

send.Add(0x00);

send.AddRange(CRC16(send.ToArray(), send.Count));

byte[] receive = new byte[send.Count];

//发送 接受报文

if (SendAndReceive(send.ToArray(), ref receive))

{

//验证报文

BytesArrayEquals(send.ToArray(),receive);

return true;

}

return false;

}

#endregion

#region 预置0X06单寄存器

/// <summary>

/// 预置单寄存器

/// </summary>

/// <param name="slaveid">从站地址</param>

/// <param name="start">寄存器地址</param>

/// <param name="value">寄存器数据 字节类型</param>

/// <returns></returns>

public bool PreSetSingleRegister(byte slaveid,ushort start, byte[] value)

{

//拼接报文

List<byte> send = new List<byte>();

send.Add(slaveid);

send.Add(0X05);

send.Add(Convert.ToByte(start/256));

send.Add(Convert.ToByte(start%256));

send.AddRange(value);//寄存器数据

send.AddRange(CRC16(send.ToArray(), send.Count));

byte[] receive = new byte[send.Count];

//发送接收报文

if (SendAndReceive(send.ToArray(), ref receive))

{

//验证报文

BytesArrayEquals(send.ToArray(), receive);

return true;

}

return false;

}

/// <summary>

/// 预置单寄存器

/// </summary>

/// <param name="slaveid">从站地址</param>

/// <param name="start">初始寄存器</param>

/// <param name="value">寄存器数据 有符号的整型</param>

/// <returns></returns>

public bool PreSetSingleRegister(byte slaveid, ushort start, short value)

{

return PreSetSingleRegister(slaveid,start,BitConverter.GetBytes(value).Reverse().ToArray());

}

/// <summary>

/// 预置单寄存器

/// </summary>

/// <param name="slaveid">从站地址</param>

/// <param name="start">寄存器地址</param>

/// <param name="value">寄存器数值 无符号整型</param>

/// <returns></returns>

public bool PreSetSingleRegister(byte slaveid, ushort start, ushort value)

{

return PreSetSingleRegister(slaveid,start,BitConverter.GetBytes(value).Reverse().ToArray());

}

#endregion

#### 4、预置多线圈和多寄存器

此时需要考试的是将线圈数据，bool数组转换为字节数组，对此需要判断bool数组是多少个字节，并且置位对应线圈位置就&(与)1，复位就对应的位置为0，其他位置1与之相~（非），这样不会导致其他位的数据改变；此时对应的寄存器，给的相对应的应该是byte[]数组数据，直接添加

/// <summary>

/// 预置多线圈

/// </summary>

/// <param name="slaveid">从站地址</param>

/// <param name="start">初始线圈</param>

/// <param name="value">布尔数组</param>

/// <returns>返回结果</returns>

public bool PreSetMultiCoils(byte slaveid,ushort start, bool[] value)

{

//拼接报文

List<byte> send=new List<byte>();

send.Add(slaveid);

send.Add(0x0f);

send.Add(Convert.ToByte(start/256));

send.Add(Convert.ToByte(start%256));

send.Add(Convert.ToByte( value.Length/256));//线圈数量

send.Add(Convert.ToByte(value.Length%256));

send.Add((byte)GetByteArrayFromBoolArray(value).Length);//字节计数

//数据

byte[] boolByte = GetByteArrayFromBoolArray(value);

send.AddRange(boolByte);

//CRC校验

send.AddRange(CRC16(send.ToArray(),send.Count));

//发送接受报文

byte[] receive =null;

if (SendAndReceive(send.ToArray(), ref receive))

{

//验证报文

if (CheckCRC(receive)&&receive.Length==8)

{

for(int i=0;i<=8;i++)

{

send[i]=receive[i];

return false;

}

}

return true;

}

return false;

}

#endregion

#region 预置0X10 多寄存器

public bool PreSetMultiRegisters(byte slaveid,ushort start, byte[] values)

{

//判断寄存器数据，不可以奇数字节，不可以无字节，不可以为空

if (values==null &&values.Length%2!=0&&values.Length==0)

{

return false;

}

//拼接报文

List<byte> send = new List<byte>();

send.Add(slaveid);

send.Add(0x10);

send.Add(Convert.ToByte(start/256));

send.Add(Convert.ToByte(start%256));

send.Add(Convert.ToByte(values.Length/2/256));//寄存器数量

send.Add(Convert.ToByte(values.Length/2%256));

send.Add((byte)values.Length);//字节计数

//直接将byte[]数据,写入

send.AddRange(values);

//CRC校验

send.AddRange(CRC16(send.ToArray(), send.Count));

//发送接受报文

byte[] receive = null;

if (SendAndReceive(send.ToArray(), ref receive))

{

//验证报文

if (CheckCRC(receive)&&receive.Length==8)

{

for (int i = 0; i<=8; i++)

{

send[i]=receive[i];

return false;

}

}

return true;

}

return false;

}

**发送，接受报文的通用方法**

/// <summary>

/// 发送，接受报文

/// </summary>

/// <param name="send">发送报文字节数组</param>

/// <param name="receive">接受报文字节数组</param>

/// <returns>判断是成功</returns>

public bool SendAndReceive(byte[] send,ref byte[] receive)

{

try

{

//发送报文

serialPort.Write(send, 0, send.Length);

//定义一个buffer，用来缓存每次接受的报文

byte[] buffer = new byte[1024];

//定义一个内存，MemoryStream

MemoryStream memoryStream = new MemoryStream();

//定义一个开始时间时间

DateTime start = DateTime.Now;

//这么处理的原因是防止一次读取不完整

//循环读取缓冲区的数据，如果大于0，就读取出来，放到内存里，如果等于0，说明读完了

//如果每次都没有读到，就要设置一个超时时间

while (true)

{

Thread.Sleep(ReceiveDelay);

if (serialPort.BytesToRead>0)//判断返回的报文的中间缓存区的字节数不为0，才可以读取接受报文

{

int count = serialPort.Read(buffer, 0, buffer.Length);

memoryStream.Write(buffer, 0, count);

}

else

{

if (memoryStream.Length>0)

{

break;

}

else if ((DateTime.Now-start).TotalMilliseconds>this.ReceiveRead)

{

return false;

}

}

}

receive=memoryStream.ToArray();

return true;

}

catch(Exception )

{

return false;

}

}

### 8、通信温湿度传感器：

通过设备给定的pdf手册上，说明了从站地址是0100H（读写）,波特率是0101H（读写），湿度0000H（只读），湿度0001温度（只读），本质上来说也是输出寄存器，只不过在传感器单片机的底层限制了写入。

对于硬件，因为此时有4台，温湿度的传感器，没有设置之前所有的从站地址都是1，通过Modbus Poll当作主站来连接传感器，读取从站地址256，和波特率，可以修改从站地址的数值，以及波特率。

同理将将温湿度传感器从站地址改为1，2，3，4

对于读取温湿度模块

在连接之后，我们采用了多线程模式，一直读取相应的温湿度

**CancellationTokenSource** 是 C# 中用于创建和管理 **CancellationToken** 的类。它允许你生成一个与之关联的 **CancellationToken** 对象，并在需要时请求取消操作。

具体来说，**CancellationTokenSource** 具有以下主要作用：

1. 创建 **CancellationToken**：通过 **CancellationTokenSource** 的构造函数，你可以创建一个新的 **CancellationToken** 对象。这个对象通常传递给需要取消支持的方法或任务。
2. 请求取消操作：**CancellationTokenSource** 提供了 **Cancel** 方法，用于请求取消操作。一旦调用了 **Cancel** 方法，与之相关联的 **CancellationToken** 的 **IsCancellationRequested** 属性会变为 **true**，表示取消请求已发出。
3. 取消操作的通知：**CancellationTokenSource** 还可以用于注册回调方法，以在取消请求时执行某些操作。这可以通过 **Register** 方法来实现。

以下是一个示例，说明如何使用 **CancellationTokenSource** 来创建和请求取消操作：

CancellationTokenSource cts = new CancellationTokenSource();

CancellationToken token = cts.Token;

// 在某个地方，可以请求取消操作

cts.Cancel();

// 在其他地方，可以检查是否有取消请求

一般通过另一个线程一直检查是否有申请取消

Task.Run(

if (token.IsCancellationRequested)

{

// 执行取消时的操作

}

);

**CancellationTokenSource** 在多线程和异步编程中非常有用，它允许你在需要时安全地取消执行中的任务或操作。

在线程中是无法更改控件的属性的，必须通过deleate委托来修改控件的属性

在Task.Run(

()=>{while（iscancellationRegisters）

{

this.invoke(new action(()=>{在这里修改控件属性}))

}})；