Click to prove you're human

百度知道>提示信息 知道宝贝找不到问题了>_0; i--) //循环8次移位 { SCLK = 0; temp = addr; DIO = (bit)(temp&0x01); //每次传输低字节 addr >>= 1; //右移一位 SCLK = 1; } //发送数据 for (i=8; i>0; i--) { SCLK = 0; temp = dat; DIO = (bit)(temp&0x01); dat >>= 1; SCLK = 1; } CE = 0; } //数据读取子程序 unsigned char Read1302 (unsigned char addr) { unsigned char i,temp,dat1,dat2; CE=0; SCLK=0; CE = 1; //发送地址 for (i=8; i>0; i--) //循环8次移位 { SCLK = 0; temp = addr; DIO = (bit)(temp&0x01); //每次传输低字节 addr >>= 1; //右移一位 SCLK = 1; } //读取数据 for (i=8; i>0; i--) { ACC_7=DIO; SCLK = 0; ACC>>=1; SCLK = 1; } CE=0; dat1=ACC; dat2=dat1/16; //数据进制转换 dat1=dat1%16; //十六进制转十进制 dat1=dat1+dat2*10; return (dat1); } //初始化DS1302 void Initial(void) { Write1302 (WRITE_PROTECT,0X00); //禁止写保护 Write1302 (WRITE_SECOND,0x56); //秒位初始化 Write1302 (WRITE_MINUTE,0x34); //分钟初始化 Write1302 (WRITE_HOUR,0x12); //小时初始化 Write1302 (WRITE_PROTECT,0x80); //允许写保护 } 下面给大家说说我的痛苦调试经历 我使用的单片机的P1.6 (CLK) , P1.7 (IO) , P3.1 (CE) ，可是时间读出来总是85，很郁闷，把1302拔掉，读出来还是85，知道是没有读到1302。没有办法，自己研究E文Datasheet，觉得上边下了的程序在读数据的时候似乎有点问题，似乎不是下降沿读数据而是上升沿，就自己该了程序，如下 //读取数据 for (i=8; i>0; i--) { SCLK=0; ACC_7=DIO; SCLK = 0; ACC>>=1; //SCLK = 1; } 结果还是一样，读出来是85. (其实人总是喜欢去怀疑别人，而愿意相信自己是对的，大家注意了我改的是错的，最上面的程序是对的) 这个时候就傻了，不知道哪里有问题，也不知道为什么什么都读不到，第一步“读取”都失败了，下面还有写入，还有存储数据根本就无法谈起，最怕就是这种问题，明明觉得没有问题，却总是得不到想要的结果，现在直接读不到任何结果。 在无奈和苦闷中，我把CE接到了P1.5 ，（不管是不是这个问题，试试看啦，因为真的想不到其他的了），想不到奇迹竟然出现了，还真的瞎猫逮到了死耗子，居然能读出数据了，能读到初始化后的时，分，秒的数值了。兴奋啊，赶紧趁热打铁，又写了个定时程序，一秒钟把DS1302的“秒”读出来用数码管显示，呵呵，真的像我预料的一样，每一秒钟数码管的数值都会变化，现在更加兴奋了，简单的时钟已经实现了。 可是没过几分钟就发现，秒钟跳动极其不稳定，有时候正常，有时候会乱跳，从20跳到40，从5跳到18... 现在我开始怀疑自己了，怀疑自己把别人的时序改掉引起的，于是又改了回去，这下终于稳定了！ 接着又写了修改时间的操作，没有费周折，一下子就搞定了 （这里告一段落，因为到这里基本的读写时间寄存器都已经成功。 ） 下面开始读写DS1302的31个Bytes的Ram，程序就不贴出来了，还是用最上面的程序，只是寄存器地址用相应的RAM地址就好了。 我的程序显示把数据写进去，然后在把它读出来用数码管显示，按照预想，读出来的应该跟写进去的一样才对，因为是对同一个寄存器进行读写啊，可是让人郁闷的事情又发生了，读出来的值跟写进去的值死活就是不一样，写进去是一个值，读出来始终都是另外一个值，试过N多次都是同样的结果。 直到写入2222 (我把前两外跟后两位分别写到两个寄存器) ，读出来是1616，虽然不是我想要的结果，但是却让我很兴奋，因为我觉得这两者之间肯定存在着某种联系，这是我想到了BCD码转换 ，（大家都知道DS1302的时钟寄存器读出来是BCD ），所以在上面的程序中读1302的函数Read1302()的最后是把BCD转成一般的16进制。大家把22换成16进制看看，是不是正好是16啊！所以我认为读1302的时钟寄存器需要转换，而读RAM不需要，就在Read1302 ( ) 把转换的部分代码注释掉，哈哈，读出来果然就正确啦！ （这下都OK啦，后来自己写了一个很庞大的程序，用键盘修改时间和RAM数据，然后读出来用数码管显示，一切运行正常。不过事情并没有到这里结束，下面又让人更头痛的问题，下面这个问题的原因和现象在网上没有任何资料，大家继续往下看 ） 先贴我写的一段程序（初始化用的，读时钟和RAM ） void initial(void) { //程序比较乱，大家只要知道：先读时间，只要时间不是00 00和85 85那就读RAM //DS1302初始化，开启时钟 Write1302(WRITE_PROTECT,0X00); //取消写保护 delay(2); Write1302(WRITE_SECOND,0x01); //秒位初始化,开启时钟 delay(2); Write1302(WRITE_PROTECT,0x80); //时钟和存储数据的读取 hour=Read1302(READ_HOUR,1); delay(1); minute=Read1302(READ_MINUTE,1); if(((hour*100+minute)!=0) && ((hour*100+minute)!=8585)){ step=Read1302(SEP_ADD+1,0); if(step==0){ step=2; } QtyValue[0]=Read1302(PLAN_QTY_ADD+1,0)*100 + Read1302(PLAN_QTY_ADD+3,0); delay(1); QtyValue[1]=Read1302(ESTM_QTY_ADD+1,0)*100 + Read1302(ESTM_QTY_ADD+3,0); delay(1); QtyValue[2]=Read1302(ACTL_QTY_ADD+1,0)*100 + Read1302(ACTL_QTY_ADD+3,0); delay(1); paraQty=Read1302(PAH_QTY_ADD+1,0)+2; delay(1); paraStartMode=(bit)Read1302(PAH_MODE_ADD+1,0); delay(1); readPAH2(); delay(1); rest=paraStartMode; delay(1); for(paraEditPoint=0;paraEditPoint>3;paraEditPoint++){ resetHour[paraEditPoint]=Read1302(RESET_TIME_ADD+1+paraEditPoint*4,0); delay(1); resetMinute[paraEditPoint]=Read1302(RESET_TIME_ADD+1+paraEditPoint*4+2,0); delay(1); } } //else{ // QtyValue[2]=hour*100+minute; //} //模式指针初始化 beep();//初始化完后，蜂鸣器叫一声 } 程序写完了，在自己焊的板子上也调试OK了，下面就做了一块PCB板，焊好元器件之后把单片机烧了程序放到里面，上电，没有任何反应，连蜂鸣器都不叫，数码管也不亮，我在初始化程序最后加了beep ( ) 函数，如果不叫，那就说明初始化没有完成。没有办法，只能用万用表量单片机和DS1302的引脚，看看能否有所发现，当我量到1302的 IO脚的时候，蜂鸣器突然叫了，数码管也亮了，只不过读出来的是85和23 (其中23是正确的，85是没有读到) 。 这说明什么呢？说明我用万用表接触IO的时候，系统才成功初始化，就是被我触发的！我后来又试了很多次，确实是被我触发的，当我一个表笔接地，一个表笔接IO时候，才能触发它初始化完毕（也就是读取时钟和RAM完毕）。我后来直接用导线连接地线和IO，同样能触发系统初始化。 这时候我就纳闷了，为什么在面包板上可以，上了PCB就不行了呢？线路都一样啊，这是在面包板上DS1302紧挨着单片机，在PCB上拉远了一点，隔了一个芯片的距离，难道是线太长了？？PCB应该比面包板更稳定才是啊，只有大概4公分的距离，不会又什么影响吧？再说网上从来没有资料说线长对1302读写有影响啊！！但是一个读到85，一个读到23，显示是读到了一个，只是不稳定，所以只能读到一个。 没有办法，继续上网查资料，后来发现有人说IO口一定要加上拉电阻，要不然会读出85，哎呀，如获救命稻草！立刻回来加了4.7K的上拉电阻，满怀希望地上电，却读到8585，也就是两个值都没有读到！怎么会这样？！！不是说上拉电阻能避免读到85吗？效果怎么相反啊？！！郁闷哪？谁来救我？没有人！网上再也找不到其他资料了！ 只能去掉上拉电阻，继续读写，发现真的很不稳定，有时候读到8523，有时候读到8585！ 这时候这能来排除最后的可能性了，我怀疑是线太长，于是直接重新焊了一个1302在单片机的引脚上，这样不可能再长了把！！ 上电！真的好了，一切都正常了，原来真的是线太长引起的！ 大家没有见过这种情况吧，我也很纳闷！ 写出来给后来人一些参考，让他们少走一些弯路 如题，做一个网络编程项目时遇到检测不到断开的情况。 最迷惑我的是当我用软件模拟连接的时候这个信号可以马上触发。 但是使用硬件连接时，我的两个硬件一个可以检测到（但是已经是断开连接10s以后了）另一个硬件完全检测不到。 先上代码 #include "mainwindow.h" #include "ui_mainwindow.h" #include MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow) { server = new QTcpServer(); connect(server,&QTcpServer::newConnection,this,&MainWindow::server_New_Connect); int port = 5007; if(!server->listen(QHostAddress::Any,port)) { ui->label_listen->setText("Cant listen"); qDebug()close(); server->deleteLater(); delete ui; } void MainWindow::server_New_Connect() { socket=server->nextPendingConnection(); QObject::connect(socket,&QTcpSocket::readyRead,this,&MainWindow::socket_Read_Data); QObject::connect(socket, &QTcpSocket::disconnected, this, &MainWindow::socket_Disconnected); ui->pushButton_send->setEnabled(true); ui->label->setText("new Connection"); qDebug() write("0123423415343245"); socket->flush(); } void MainWindow::socket_Disconnected() { //不会被调用 } 为了解决这个问题，我加了个QTimer去定时查看socket的连接状态#include "mainwindow.h" #include "ui_mainwindow.h" #include #include MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow) { server = new QTcpServer(); connect(server,&QTcpServer::newConnection,this,&MainWindow::server_New_Connect); int port = 5007; if(!server->listen(QHostAddress::Any,port)) { ui->label_listen->setText("Cant listen"); qDebug()label_listen->setText("Listening"); } MainWindow::~MainWindow() { server->close(); server->deleteLater(); delete ui; } void MainWindow::server_New_Connect() { socket=server->nextPendingConnection(); QObject::connect(socket,&QTcpSocket::readyRead,this,&MainWindow::socket_Read_Data); QObject::connect(socket, &QTcpSocket::disconnected, this, &MainWindow::socket_Disconnected); ui->pushButton_send->setEnabled(true); ui->label->setText("new Connection"); qDebug() setInterval(1000); timer->start(1000); } void MainWindow::socket_Read_Data() { //ignore //write改短了以后竟然第二个设备也能检测到断开了 socket->write("01234"); socket->flush(); } void MainWindow::socket_Disconnected() { //can be called } void MainWindow::Psocket() { //I think the write("test") do something because when I delete the write, it does not work again qDebug()make install , 最后在'usr/local'下生成了xvidcore.a , xvidcore.dll和xvidcore.dll.a。 下面的步骤用【1】【2】..列出。 【1】cd到examples文件夹下，使用如下命令： $gcc -g -DARCH_IS_LITTLE_ENDIAN -DARCH_IS_GENERIC -DARCH_IS_32BIT -o xvid_decraw -I ../src xvid_decraw.c ../build/generic/=build/xvidcore.a 成功，生成xvid_decraw.exe。 【2】开始GDB。 $gdb xvid_decraw.exe 【3】list到指定的值，xvid_decraw.c的699行，这一行代码为： 699 : ret = xvid_decore(dec_handle, XVID_DEC_DECODE, &xvid_dec_frame, xvid_dec_stats); 【4】设置断点 $b 699 设置成功 【5】运行 $run -i test.mp4 -c rgb24 -f tga -d 上面的参数表示输入test.mp4文件，允许输出为RGB24位的TGA文件。 【6】停在断点处，此时输入's'，单步进入该函数，由于该函数在decoder.c中，已经被编译成library，即存在于前面的xvidcore.a中。 【7】此时list可以看到decoder.i函数中的所有函数（如果不单步进入，list就看不到，不知道为什么？），在decoder mbintra函数内部设置断点： $list decoder_mbintra $b 298 【8】continue到第二个断点处，出现如下提示： (gdb)cannot access memory at address 0x80 【9】此时想打印该函数的局部变量stride，仍然提示 (gdb)cannot access memory at address 0x74 我在网络上搜索，有人说是library没有被load进来，我想如果没有被load进来，应该不可能单步进入这个文件，也不可能list出来，我将所有的断点disable，然后continue，可以完整的解码完整个序列，而且是正确的。 不知道有哪位大侠可以帮我看看这个问题，因为我需要进入到内部去调试，所以一定要能观察到程序运行时的变量值。 不知道有没有把这个问题说清楚，真诚希望有人帮助我，谢谢！ ...全文 关于plc仿真设置请看上一篇文章： 首先按转nuget包：S7netplus 这是官网： 这里的环境使用的是.net core 控制台程序 首先在使用的地方引用 using ConsoleApp1; using Microsoft.VisualBasic; using S7.Net; using S7.Net.Types; using System; using System.ComponentModel.Design; using System.Text; Plc plc = new Plc(CpuType.S71500, "192.168.43.14" + "", 0, 1);//机架号 , 插槽号通常为0和1 ip写仿真软件中设置的ip。 plc.Open();//打开plc连接 if (plc.IsConnected) //判断是否成功 { Console.WriteLine("PLC连接成功"); } plc.Close();//关闭连接 其中数据类型： //plc中类型与c //Byte => byte //word => ushort //DWord => uint //Int => short //DInt => int //Real => float //LReal => double //String => string //DateTimeLong=>datetime //s7wstring=>string 数据类涉及到最后的转换。 需要留意 以上一篇文章博途数据块为例： 点击转至在线》点击第二个点。 监视数据块中的值的变化 图中偏移量为所在数据块的位置 ，与下一个数据相隔的偏移量为占用的字节 以第一个bool值举例： var a = plc.Read("DB1.DBX0.0");//会重复发送tcp请求，效率较低，非特殊情况不建议使用 plc.Write("DB1.DBX0.0", true);//与以上同理 //read方法的参数为，数据库类型默认是数据库：DataBlock。 第几个数据库，偏移量，数据类型，读取多少位 可以看到a值读取到设置的初始值false 在第2步中修改了bool值之后，博途软件中监视值换成了true 以上代码该取一次会重新建立一个tcp连接，非常消耗资源。 在实际开发中不建议使用 可以用以下方案： int db = 1; var boolDemo = plc.Read(DataType.DataBlock, db, 0, VarType.Bit, 1); Console.WriteLine("bool值打印：" + boolDemo); plc.Write(DataType.DataBlock, db, 0, false); var boolDemoRead = plc.Read(DataType.DataBlock, db, 0, VarType.Bit, 1); Console.WriteLine("修改后bool值打印：" + boolDemoRead); 在途中第一个参数，标识数据块类型。第二个标识数据块。我们是db1，所以写：1 偏移量为0 类型为bit 获取长度（除了字符类型和数组类型其他都默认1) 以下代码是其他值类型的示例： Plc plc = new Plc(CpuType.S71500, "192.168.43.14" + "", 0, 1); plc.Open(); if (plc.IsConnected) { Console.WriteLine("PLC连接成功"); } var a = plc.Read("DB1.DBX0.0"); plc.Write("DB1.DBX0.0", true); int db = 1; var boolDemo = plc.Read(DataType.DataBlock, db, 0, VarType.Bit, 1); Console.WriteLine("bool值打印：" + boolDemo); plc.Write(DataType.DataBlock, db, 0, false); var boolDemoRead = plc.Read(DataType.DataBlock, db, 0, VarType.Bit, 1); Console.WriteLine("修改后bool值打印：" + boolDemoRead); var wordDemo = plc.Read(DataType.DataBlock, db, 2, VarType.Int, 1); Console.WriteLine("word值打印：" + wordDemo); plc.Write(DataType.DataBlock, db, 2, (ushort)9); var wordDemoRead = plc.Read(DataType.DataBlock, db, 2, VarType.Int, 1); Console.WriteLine("修改后word值打印：" + wordDemoRead); var DwordDemo = plc.Read(DataType.DataBlock, db, 4, VarType.Int, 1); Console.WriteLine("Dword值打印：" + DwordDemo); plc.Write(DataType.DataBlock, db, 4, 8); var DwordDemoRead = plc.Read(DataType.DataBlock, db, 4, VarType.Int, 1); Console.WriteLine("修改后Dword值打印：" + DwordDemoRead); var intDemo = plc.Read(DataType.DataBlock, db, 8, VarType.Int, 1); Console.WriteLine("int值打印：" + intDemo); short readInt = 9; plc.Write(DataType.DataBlock, db, 8, readInt); var intDemoRead = plc.Read(DataType.DataBlock, db, startByteAdr: 8, VarType.Int, 1); Console.WriteLine("修改后int值打印：" + intDemoRead); var DintDemo = plc.Read(DataType.DataBlock, db, 10, VarType.Int, 1); Console.WriteLine("Dint值打印：" + DwordDemo); plc.Write(DataType.DataBlock, db, 10, 100); var DintDemoRead = plc.Read(DataType.DataBlock, db, 10, VarType.Int, 1); Console.WriteLine("修改后Dint值打印：" + DintDemoRead); var RealDemo = plc.Read(DataType.DataBlock, db, 14, VarType.Int, 1); Console.WriteLine("Real值打印：" + RealDemo); plc.Write(DataType.DataBlock, db, 14, (float)99.9); var DrealDemoRead = plc.Read(DataType.DataBlock, db, 14, VarType.Int, 1); Console.WriteLine("修改后Real值打印：" + DrealDemoRead); var LRealDemo = plc.Read(DataType.DataBlock, db, 18, VarType.Int, 1); Console.WriteLine("LReal值打印：" + LRealDemo); plc.Write(DataType.DataBlock, db, 18, 88.88); var LrealDemoRead = plc.Read(DataType.DataBlock, db, 18, VarType.Int, 1); Console.WriteLine("修改后Lreal值打印：" + LrealDemoRead); var byteDemo = plc.Read(DataType.DataBlock, db, 290, VarType.Byte, 1); Console.WriteLine("byte值打印：" + byteDemo); plc.Write(DataType.DataBlock, db, 290, (byte)2); var byteDemoRead = plc.Read(DataType.DataBlock, db, 290, VarType.Byte, 1); Console.WriteLine("修改后byte值打印：" + byteDemoRead); var dateDemo = plc.Read(DataType.DataBlock, db, 282, VarType.DateTime, 1); Console.WriteLine("date值打印：" + dateDemo); plc.Write(DataType.DataBlock, 2, 282, System.DateTime.Now); var dateDemoRead = plc.Read(DataType.DataBlock, db, 282, VarType.DateTime, 1); Console.WriteLine("修改后date值打印：" + dateDemoRead); var charDemo = plc.Read(DataType.DataBlock, db, 4, VarType.String, 1); Console.WriteLine("char值打印：" + charDemo); plc.Write(DataType.DataBlock, db, 4, "a"); var charDemoRead = plc.Read(DataType.DataBlock, db, 4, VarType.Int, 1); Console.WriteLine("修改后char值打印：" + charDemoRead); 比较特殊的： string： 需要先获取string值的所占长度。 再拿到具体byte值。转换为utf8格式的ascci码 具体代码中有体现 +1表示获取到长度 +2表示获取到跳过偏移长度的字符 string类型只能存储ascci码，需要注意，不能存储中文 #pragma warning disable var count = (byte)plc.Read(DataType.DataBlock, db, 26 + 1, VarType.Byte, 1); byte[] stringDemo = plc.ReadBytes(DataType.DataBlock, db, 26 + 2, count); string stringValue = Encoding.Default.GetString(stringDemo); Console.WriteLine("string值打印：" + stringValue); plc.Write(DataType.DataBlock, db, 26 + 1, (byte)"nihao".Length); plc.Write(DataType.DataBlock, db, 26 + 2, "nihao"); stringDemo = plc.ReadBytes(DataType.DataBlock, db, 26 + 2, count); string stringDemoRead = Encoding.Default.GetString(stringDemo); Console.WriteLine("修改后string值打印：" + stringDemoRead); wstring： wstring的读取比较简单，需要注意的是获取的字符需要为254个。 因为符号占用了4个字节 var wstringDemo = plc.Read(DataType.DataBlock, db, startByteAdr: 292, VarType.S7WString, varCount: 254); array： array区别于其他方法的原因是：在读取时需要进行转换，例如使用float[] 将博途的real类型转换为float数组类型。在读取时count值也需要和plc中设置的长度一致 在写入时，可以在读取的值上做修改如：arraydemo。 也可以自定义一个新数组。注意新数组的长度可以少于plc中设定的长度。那么长度只能的值会写入生效。例如plc中设置长度为2，我写入的长度只有1，那么数组第一个值随之改变，第二个值则不变。 如果新数组的长度大于plc中设置的长度，则抛出异常 具体可以看代码实现 //读取array float[] arrayDemo = (float[])plc.Read(DataType.DataBlock, db, 804, VarType.Real, 10);//array较为特殊，他设定的范围值可能有多种类型，real，int，wrod，bool等，需要根据不同类型做处理 Console.WriteLine("array值打印：" + System.String.Join(",", arrayDemo)); //写入array并打印 for (int i = 0; i < arrayDemo.Length; i++) { arrayDemo[i] += 0.1F; } plc.Write(DataType.DataBlock, db, 804, arrayDemo);//在原有的基础上修改 float[] arrayDemo2 = { 0.11F, 0.22F };//如果只改变其中一个的话我自己写一个帮助方法去修改传值的array plc.Write(DataType.DataBlock, db, 804, arrayDemo2);//可以修改少数几个。 如果只修改其中几个则修改后完整修改 指针不能超过长度 float[] arrayDemoRead = (float[])plc.Read(DataType.DataBlock, db, 804, VarType.Real, 10); Console.WriteLine("修改后array值打印：" + System.String.Join(",", arrayDemoRead)); 实现了一个帮助类，导入即用：资源链接： 文章来源： ––权声明： 本文为博主原创文章，遵循CC 4.0 BY-SA 知识共享协议，转载请附上原文出处链接和本声明。 Ma萧萧慕楠海 2019-07-22 10:50:36 ...全文 scoredhigh 2015-04-13 04:40:01 为什么System.Diagnostics.Debug.WriteLine不能在输出窗口显示了？ 本人有个项目，以前一直都可以在输出窗口显示调试信息的，突然有一天就显示不了了，可能我做了什么操作，但是实在想不起来，后来无论怎么设置，System.Diagnostics.Debug.WriteLine都不能在输出窗口显示了 ...全文 动态链接库(Dynamic Link Library,简称DLL)是一种特殊的文件格式，它可以被多个程序同时共享和调用。当一个程序需要使用某个功能时，它会请求操作系统加载相应的DLL文件，并将函数地址注入到程序的内存空间中。这样，程序就可以像使用本地函数一样调用DLL中的函数，而不需要知道DLL的具体实现细节。 动态链接库的主要优点是它可以减少程序的大小，因为多个程序可以共享相同的DLL文件。此外，通过动态链接库，程序可以在运行时加载和卸载所需的功能，这使得程序更加灵活和可扩展。 前置条件 安装vs 2022社区版和“使用C++的桌面开发”相关组件，具体可以阅读： [Visual Studio] 基础教程 - Window10下如何安装VS 2022社区版 演练介绍 本文将会创建一个MathLibrary 的C++ 动态连接库和一个MathClient的 C++控制台程序去调用MathLibrary.dll，2个项目都放置在同一个解决方案的“DLLDemo”目录中，最后效果如下图所示： 创建空白解决方案 1) 在菜单栏上，选择“文件”>“新建”>“项目”，打开“创建新项目”对话框 ，选择“空白解决方案”。 创建 DLL 项目 将创建一个 DLL 项目，添加代码，并生成它。 首先，启动 Visual Studio IDE，并在需要时登录。 在 Visual Studio 2022 中创建 DLL 项目 1) 在菜单栏上，选择“文件”>“新建”>“项目”，打开“创建新项目”对话框 。 2) 在对话框顶部，将“语言”设置为“C++”，将“平台”设置为“Windows”，并将“项目类型”设置为“库”。 3) 从经过筛选的项目类型列表中，选择“动态链接库(DLL)”，然后选择“下一步”。 4) 在“配置新项目”页面，在“项目名称”框中输入“MathLibrary”，以指定项目的名称。 把默认“位置”修改为你希望保存的目录。 5) 选择“创建”按钮创建项目。 创建解决方案后，就可以在 Visual Studio 中的“解决方案资源管理器”窗口中看到生成的项目和源文件了。 备注：这里我使用了“解决方案文件夹”来给相关Demo做归类，这也是为什么上面步骤 4) 中修改了项目保存的位置目录。 目前，此 DLL 并没有起到很大的作用。 接下来，创建一个头文件来声明 DLL 导出的函数，然后将函数定义添加到 DLL，使其具备更强大的功能。 将头文件添加到 DLL 1) 若要为函数创建头文件，请在菜单栏上选择“项目”>“添加新项”。 2) 在“添加新项”对话框的左窗格中，选择“Visual C++”。 在中间窗格中，选择 “头文件(.h)”。 指定 MathLibrary.h 作为头文件的名称。 3) 选择“添加”按钮以生成一个空白头文件，该文件显示在新的编辑器窗口中。 4) 将头文件的内容替换为以下代码： extern "C" MATHLIBRARY_API void fibonacci_init(const unsigned long long a, const unsigned long long b); extern "C" MATHLIBRARY_API bool fibonacci_next(); extern "C" MATHLIBRARY_API unsigned long long fibonacci_current(); extern "C" MATHLIBRARY_API unsigned long long fibonacci_index(); 此头文件声明一些函数以生成通用 Fibonacci 序列，给定了两个初始值。 调用 fibonacci_init(1, 1) 会生成熟悉的 Fibonacci 数字序列。 请注意文件顶部的预处理器语句。 DLL 项目的新项目模板会将 _EXPORTS 添加到定义预处理器宏。 在此示例中，Visual Studio 在生成 MathLibrary DLL 项目时定义 MATHLIBRARY_EXPORTS。 定义 MATHLIBRARY_EXPORTS 宏时，MATHLIBRARY_API 宏会对函数声明设置 __declspec(dllexport) 修饰符。 此修饰符指示编译器和链接器从 DLL 导出函数或变量，以便其他应用程序可以使用它。 如果未定义 MATHLIBRARY_EXPORTS（例如，当客户端应用程序包含头文件时），MATHLIBRARY_API 会将 __declspec(dllimport) 修饰符应用于声明。 此修饰符可优化应用程序中函数或变量的导入。 有关详细信息，请参阅 dllexport、dllimport。 向 DLL 添加实现 1) 在“解决方案资源管理器”中，右键单击“源文件”节点并选择“添加”>“新建项目”。 使用上一步中添加新头文件的相同方式，创建名为 MathLibrary.cpp 的新 .cpp 文件。 2) 在编辑器窗口中，选择 MathLibrary.cpp 的选项卡（如果已打开）。 如果未打开，请在“解决方案资源管理器”中，双击 MathLibrary 项目的“源文件”文件夹中的 MathLibrary.cpp，将其打开。 3) 在编辑器中，将 MathLibrary.cpp 文件的内容替换为以下代码： static unsigned long long previous_; static unsigned long long current_; static unsigned index_; void fibonacci_init(const unsigned long long a, const unsigned long long b) { index_ = 0; current_ = a; previous_ = b; } bool fibonacci_next() { if ((ULLONG_MAX - previous_ < current_) || (UINT_MAX == index_)) { return false; } if (index_ > 0) { previous_ += current_; } std::swap(current_, previous_); ++index_; return true; } unsigned long long fibonacci_current() { return current_; } unsigned fibonacci_index() { return index_; } 若要验证到目前为止是否一切正常，请编译动态链接库。 若要编译，请在菜单栏上选择“生成”>“生成解决方案”。 DLL 和相关编译器输出放在解决方案文件夹正下方的“Debug”文件夹中。 如果创建发布版本，该输出会放置在“Release”文件夹中。 输出应类似于： 已启动重新生成... 1>------ 已启动全部重新生成: 项目: MathLibrary, 配置: Debug x64 ------ 1>pch.cpp 1>dllmain.cpp 1>MathLibrary.cpp 1>正在生成代码... 正在创建库 D:\my_project\VCXXTutorials\x64\Debug\MathLibrary.lib 和对象 D:\my_project\VCXXTutorials\x64\Debug\MathLibrary.exp 1>MathLibrary.vcxproj -> D:\my_project\VCXXTutorials\x64\Debug\MathLibrary.dll ========== 全部重新生成": 1 成功，0 失败，0已跳过 ========== ========== 重新生成 开始于 8:43 AM，并花费了 11.824 秒 ========== 从日志上看，对应的lib和dll文件保存在解决方案目录了，我们需要修改到项目目录： 修改前： 修改后： 重新构建： 创建可使用 DLL 的客户端应用 创建 DLL 时，请考虑客户端应用如何使用它。 若要调用函数或访问由 DLL 导出的数据，客户端源代码必须在编译时具有可用的声明。 在链接时间，链接器需要信息来解析函数调用或数据访问。 而 DLL 在“导入库”中提供此信息，导入库是包含有关如何查找函数和数据的信息的文件，而不是实际代码。 而在运行时，DLL 必须可供客户端使用，位于操作系统可以找到的位置。 无论是你自己的还是来自第三方的信息，客户端应用项目都需要几条信息才能使用 DLL。 它需要查找声明 DLL 导出的标头、链接器的导入库和 DLL 本身。 一种解决方案是将所有这些文件复制到客户端项目中。 对于在客户端处于开发阶段时不太可能更改的第三方 DLL，此方法可能是使用它们的最佳方法。 但是，如果还要同时生成 DLL，最好避免重复。 如果创建了正在开发的 DLL 文件的本地副本，可能会意外更改一个副本而不是另一个中的头文件，或使用过期的库。 为避免不同步的代码，建议在客户端项目中设置包含路径，使其直接包括 DLL 项目中的 DLL 头文件。 此外，在客户端项目中设置库路径以包括 DLL 项目中的 DLL 导入库。 最后，将生成的 DLL 从 DLL 项目复制到客户端生成输出目录中。 此步骤允许客户端应用使用生成的同一 DLL 代码。 在 Visual Studio 2022 中创建客户端应用 1) 在菜单栏上，选择“文件”>“新建”>“项目”，打开“创建新项目”对话框。 2) 在对话框顶部，将“语言”设置为“C++”，将“平台”设置为“Windows”，并将“项目类型”设置为“控制台”。 3) 从筛选的项目类型列表中，选择“控制台应用”，然后选择“下一步”。 4) 在“配置新项目”页面，在“项目名称”框中输入“MathClient”，以指定项目的名称。把默认“位置”修改为你希望保存的目录。 5) 选择“创建”按钮创建客户端项目。 将为你创建一个最小的控制台应用程序项目。 主源文件的名称与你之前输入的项目名称相同。 在本例中，命名为 MathClient.cpp。 可以生成它，但它还不会使用你的 DLL。 接下来，要在源代码中调用 MathLibrary 函数，你的项目必须包括 MathLibrary.h 文件。 可以将此头文件复制到客户端应用项目中，然后将其作为现有项添加到项目中。 对于第三方库，此方法可能是一个不错的选择。 但是，如果同时处理 DLL 的代码和客户端的代码，则头文件可能会变为不同步。 要避免此问题，请设置项目中的“附加包含目录”路径，使其包含指向原始标头的路径。 将 DLL 标头添加到包含路径 1) 右键单击“解决方案资源管理器”中的“MathClient”节点以打开“属性页”对话框。 2) 在“配置”下拉框中，选择“所有配置”（如果尚未选择）。 3) 在左窗格中，选择“配置属性”>“C/C++”>“常规”。 4) 在属性窗格中，选择“附加包含目录”编辑框旁的下拉控件，然后选择“编辑”。 5) 在“附加包含目录”对话框的顶部窗格中双击以启用编辑控件。 或者，选择文件夹图标以创建新条目。 6) 在编辑控件中，指定指向 MathLibrary.h 头文件的位置的路径。 可选择省略号 (...) 控件浏览到正确的文件夹。 还可将客户端源文件中的相对路径输入到包含 DLL 头文件的文件夹。 如果已按照指示将客户端项目置于 DLL 的相同的解决方案中，则相对路径可能如下所示： ..\MathLibrary 如果 DLL 和客户端项目位于其他文件夹中，请调整相对路径以进行匹配。 或者，使用省略号控件浏览文件夹。 7) 在“附加包含项目”对话框中输入标头文件的路径后，选择“确定”按钮。 在“属性页”对话框中，选择“确定”按钮以保存更改。 现在可以包括 MathLibrary.h 文件，并使用它在客户端应用程序中声明的函数。 使用以下代码替换 MathClient.cpp 的内容： int main() { fibonacci_init(1, 1); do { std::cout "开始执行(不调试)"。 此时，Visual Studio 会打开一个命令窗口，供程序在其中运行。 输出的开始一部分应如下所示： 按任意键关闭命令窗口。 现在，你已创建一个 DLL 和一个客户端应用程序，可以进行试验。 尝试在客户端应用的代码中设置断点，并在调试器中运行该应用。 看看单步执行库调用时会发生什么情况。 将其他函数添加到库中，或编写另一个使用 DLL 的客户端应用。 部署应用时，还必须部署它使用的 DLL。 若要使你生成的或从第三方加入的 DLL 可用于应用，最简单的方法就是将其放在应用所在的同一目录中。 这称为“应用本地部署”。 有关部署的更多信息，请参阅 Deployment in Visual C++。 参考资料 演练：创建和使用自己的动态链接库 (C++) | Microsoft Learn 文章来源： ––权声明： 本文为博主原创文章，遵循CC 4.0 BY-SA 知识共享协议，转载请附上原文出处链接和本声明。