

使用自定义非托管DLL (C#调用C++)

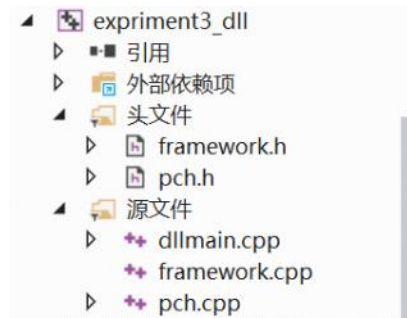
- 1.在VS中创建C++类库，使用预编译头，可以自动生成下面三个文件
- 2.在pch.cpp中写我们定义的DLL函数
- 3.将解决方案切换到Release模式，点击生成生成dll
- 4.将生成的dll放到我们需要调用它的项目的Debug或者Release目录下
- 5.在调用项目中使用DllImport引入函数，需要将其定义为public static extern
- 6.在调用项目中用调用C#方法调用引入的DLL方法






```
double Add(double a, double b)
{
    return a+b;
}

double Mul(double a, double b)
{
    return a*b;
}

[DllImport("exprimint3_dll.dll")]
1 个引用
public static extern double Add(double a, double b);

[DllImport("exprimint3_dll.dll")]
1 个引用
public static extern double Mul(double a, double b);
```



« bin > Debug		搜索"Debug"
2月 ^	名称	修改时间
	experiment3-3_show.exe	2019
	experiment3-3_show.exe.config	2019
	experiment3-3_show.pdb	2019
	experiment4_dll.dll	2019
	exprimint3_dll.dll	2019

同步调用进程Ping，整体按照书上12.1.1节，需补充一行代码

process.StartInfo.FileName = "cmd.exe";

补充在重定向输出流那句之后即可

异步调用进程Ping

整体按照书上12.1.2节即可，需补充一行代码

- 1.在104页代码之后需加上
cmdStreamInput.WriteLine("ping " + textBox1.Text.Trim() + " -n 200");
意味对cmd进程写入一条命令

杀死Ping进程（关闭进程）

ping进程是cmd进程的子进程，杀死要用特殊方式，将cmd进程和它的所有子进程递归的杀死。

- 1.创建ManagementObjectSearcher对象和ManagementObject对象获取传入方法的pid所有的子进程对象。（类似数据库查询）
- 2.对于每一个子进程，递归调用KillProcessAndChildrens方法杀死进程和子进程
- 3.处理完所有子进程后，通过Kill函数杀死传入方法的pid对应的进程

```
private static void KillProcessAndChildrens(int pid)
{
    ManagementObjectSearcher processSearcher = new ManagementObjectSearcher(
        ("Select * From Win32_Process Where ParentProcessID=" + pid));
    ManagementObjectCollection processCollection = processSearcher.Get();

    // We must kill child processes first!
    if (processCollection != null)
    {
        foreach (ManagementObject mo in processCollection)
        {
            KillProcessAndChildrens(Convert.ToInt32(mo["ProcessID"])); //kill child proces
        }
    }

    // Then kill parents.
    try
    {
        Process proc = Process.GetProcessById(pid);
        if (!proc.HasExited) proc.Kill();
    }
    catch (ArgumentException)
    {
        // Process already exited.
    }
}
```

进程调用getmac

只需要将同步或异步调用进程Ping中输入cmd的命令改为"getmac"即可

如果是同步，需修改103页代码中strCMD

如果是异步，需要我们将添加的那条代码写入的指令换成"getmac"

打印当前线程Id

```
textBox3.Text = "Thread " + Thread.CurrentThread.ManagedThreadId.ToString() + " started in " + Thread.GetDomain().FriendlyName + " with AppDomainID = " + Thread.GetDomainID().ToString() + ".";
```

输出样式为Thread 1 started in experiment7.exe with AppDomainID = 1.

无参方法创建线程

- 1.在类中声明一个无参的方法
- 2.创建线程
Thread thread1 = new Thread(new ThreadStart(method))
- 3.开启线程thread1.Start();

```
1 个引用
void method()
{
    //CreateThreadeOutput createThreade = new CreateThreadeOutput(this.CreateThreadAppendTextBox);
    //createThreade(0);
    SendMessage(main_whandle, CREATE_THREAD_NO_PARAMATER_METHOD, 0, 0);
}
```

实例方法创建线程

- 1.新建一个类，将线程将要实现的功能写成这个类内部的方法
- 2.通过以下语句创建线程
ThreadTest thread = new ThreadTest();
Thread thread2 = new Thread(thread.MyThread);
- 3.开启线程thread2.Start()

```
class ThreadTest
{
    public void MyThread()
    {
        SendMessage(main_whandle, CREATE_THREAD_OBJECT_METHOD,0, 0);
    }
}
```

匿名委托函数创建线程

将委托作为Thread构造函数的参数传入
Thread thread3 = new Thread(delegate () { SendMessage(main_whandle, CREATE_THREAD_ANONYMOUS_DELEGATE, 0, 0); });
thread3.Start();

Lambda表达式创建线程

将Lambda表达式作为Thread构造函数的参数传入
Thread thread4 = new Thread(() => { SendMessage(main_whandle, CREATE_THREAD_LAMBDA, 0, 0); });
thread4.Start();

通过有参委托创建线程

- 1.创建有参的方法，方法的参数必须是object类型
- 2.将Thread5作为参数创建ParameterizedThreadStart对象
- 3.将创建的ParameterizedThreadStart对象作为参数创建Thread对象
Thread thread5 = new Thread(new
ParameterizedThreadStart(Thread5));
//Thread5是我们定义的方法
- 4.启动线程
Thread5.Start();

```
static void Thread5(object obj)
{
    if (obj.Equals("这是一个有参数的委托"))
    {
        SendMessage(main_whandle, CREATE_THREAD_PARAMATER_METHOD, 0, 0);
    }
}
```

将线程更改为前台/后台进程

设置Thread对象的IsBackground属性即可。我们假设thread是一个Thread类的对象

thread.IsBackground = true;	//将线程设置为后台线程
thread.IsBackground = false;	//将线程设置为前台线程

使用Join实现线程阻塞

- 1.创建一个线程对象thread1,他的工作是休眠5秒
- 2.创建一个线程对象thread2,他的工作是先调用thread1的Join方法，然后休眠3秒
- 3.启动thread1和thread2。注意两者的启动顺序不能颠倒!因为thread2的执行任务中有thread1.Join(), 这条指令要求thread1必须已经开始执行。
thread1.Start();
thread2.Start();

```
Thread thread1 = new Thread(() => {
    Thread.Sleep(5000);
    SendMessage(main_whandle, JOIN_5000, 0, 0);
});
Thread thread2 = new Thread(() => {
    thread1.Join();
    Thread.Sleep(3000);
    SendMessage(main_whandle, JOIN_3000, 0, 0);
});
```

结果解释，因为thread1.Start()和thread2.Start()这两条语句的开始执行时间几乎相同，所以两个进程几乎同时开始执行任务，但是thread2的第一条指令会让其阻塞直到thread1执行完后再继续执行。所以程序运行结果为，5s后thread1执行完成，8s后thread2执行完成。
如果将thread1.Join()这条指令注释掉，则运行结果为3s后thread2执行完成，5s后thread1执行完成。

同步、异步、异步回调调用线程（博客园上的一个简单例子）

首先，通过代码定义一个委托和下面三个示例将要调用的方法：

```
public delegate int AddHandler(int a,int b);
public class 加法类
{
    public static int Add(int a, int b)
    {
        Console.WriteLine("开始计算: " + a + "+" + b);
        Thread.Sleep(3000); //模拟该方法运行三秒
        Console.WriteLine("计算完成!");
        return a + b;
    }
}
```

异步调用

```
public class 异步调用
{
    static void Main()
    {
        Console.WriteLine("==== 异步调用 AsyncInvokeTest =====");
        AddHandler handler = new AddHandler(加法类.Add);

        //IAsyncResult: 异步操作接口(interface)
        //BeginInvoke: 委托(delegate)的一个异步方法的开始
        IAsyncResult result = handler.BeginInvoke(1, 2, null,
null);

        Console.WriteLine("继续做别的事情。。。");

        //异步操作返回
        Console.WriteLine(handler.EndInvoke(result));
        Console.ReadKey();
    }
}
```

可以看到，主线程并没有等待，而是直接向下运行了。

但是问题依然存在，当主线程运行到EndInvoke时，如果这时调用没有结束（这种情况很可能出现），这时为了等待调用结果，线程依旧会被阻塞。

同步调用

委托的Invoke方法用来进行同步调用。同步调用也可以叫阻塞调用，它将阻塞当前线程，然后执行调用，调用完毕后再继续向下进行。

```
public class 同步调用
{
    static void Main()
    {
        Console.WriteLine("==== 同步调用 SyncInvokeTest =====");
        AddHandler handler = new AddHandler(加法类.Add);
        int result = handler.Invoke(1, 2);

        Console.WriteLine("继续做别的事情。。。");

        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```

异步回调

用回调函数，当调用结束时会自动调用回调函数，解决了为等待调用结果，而让线程依旧被阻塞的局面。

```
public class 异步回调
{
    static void Main()
    {
        Console.WriteLine("==== 异步回调 AsyncInvokeTest =====");
        AddHandler handler = new AddHandler(加法类.Add);

        //异步操作接口(注意BeginInvoke方法的不同!)
        IAsyncResult result = handler.BeginInvoke(1,2,new AsyncCallback(回调函数),"AsyncState:OK");

        Console.WriteLine("继续做别的事情。。。");
        Console.ReadKey();
    }

    static void 回调函数(IAsyncResult result)
    {
        //result 是“加法类.Add()方法”的返回值

        //AsyncResult 是IAsyncResult接口的一个实现类，空间：
        System.Runtime.Remoting.Messaging
        //AsyncDelegate 属性可以强制转换为用户定义的委托的实际类。
        AddHandler handler = (AddHandler)((AsyncResult)
result).AsyncDelegate;

        Console.WriteLine(handler.EndInvoke(result));
        Console.WriteLine(result.AsyncState);
    }
}
```

我定义的委托的类型为AddHandler，则为了访问 AddHandler.EndInvoke，必须将异步委托强制转换为 AddHandler。可以在异步回调函数（类型为

AsyncCallback) 中调用 MAddHandler.EndInvoke, 以获取最初提交的 AddHandler.BeginInvoke 的结果。

注意：作者用了中文的类名和方法名！不要被搞晕！！

实验上这部分有点太多了，因为要想实现老师要求的展现方式需要很多和线程调用无关的核心代码，所以我先粘了这篇我当时参考的博客上的，你看到这里后如果觉得需要我把实验上的内容也加上去告诉我一下我去加。

对于异步回调方法中

```
AddHandler handler = (AddHandler)((AsyncResult)
result).AsyncDelegate;
这一句代码使用了过多的系统里取获得正确的委托对象handler
有一种更简单的方式是使用Lambda表达式去写回调函数而不是将其单独作为一个静态成员方法，这样就可以直接使用之前定义好的handler
static void Main()
{
    Console.WriteLine("==== 异步回调 AsyncInvokeTest =====");
    AddHandler handler = new AddHandler(加法类.Add);
    //异步操作接口(注意BeginInvoke方法的不同！)
    IAsyncResult result = handler.BeginInvoke(1,2,
        (IAsyncResult res) =>{
            Console.WriteLine(handler.EndInvoke(result));
            Console.WriteLine(result.AsyncState);
        }, "AsycState:OK");

    Console.WriteLine("继续做别的事情。。。");
    Console.ReadKey();
}
```

AutoResetEvent线程通信

1.创建一个AutoResetEvent对象

```
AutoResetEvent autoResetEvent = new AutoResetEvent(false);
```

2.创建5个线程，让它们的任务都是等待autoResetEvent信号等待10s

```
for (int i = 0; i < count; i++){
    thread = new Thread(() => { autoResetEvent.WaitOne(10000); });
    ThreadList.Add(thread); //ThreadList是一个List<Thread>类型的变量
}
```

3.创建一个线程，它的任务是在等待autoResetEvent信号2s后（显然这2s内不会有人给它发信号，所以它相当于阻塞了2s），发出一个autoResetEvent信号

```
thread = new Thread(() =>
{
    autoResetEvent.WaitOne(2000);
    autoResetEvent.Set();
});
```

4.同时运行上面6个线程

```
for (int i = 0; i < count; i++)
{
    ThreadList[i].Start();
}
thread.Start();
```

结果分析：前2s不会有任何反应，第2s时第六个线程结束等待，发出信号令前五个线程中的其中一个继续执行，而这两个线程都没有后续任务，所以都会结束。剩下的4个线程等待信号一直到第10s，然后结束。

ManualResetEvent线程通信

1.创建一个ManualResetEvent对象

```
ManualResetEvent manualResetEvent= new ManualResetEvent(false);
```

2.创建5个线程，让它们的任务都是等待autoResetEvent信号等待10s

```
for (int i = 0; i < count; i++){
    thread = new Thread(() => { manualResetEvent.WaitOne(10000); });
```

```

        ThreadList.Add(thread); //ThreadList是一个List<Thread>类型的变量
    }
3.创建一个线程，它的任务是在等待manualResetEvent信号2s后（显然这2s内不会有人给它发信号，所以它相当于阻塞了2s），发出一个
manualResetEvent信号
thread = new Thread(() =>
{
    manualResetEvent.WaitOne(2000);
    manualResetEvent.Set();
});
4.同时运行上面6个线程
for (int i = 0; i < count; i++)
{
    ThreadList[i].Start();
}
thread.Start();

```

结果分析：前2s不会有任何反应，第2s时第六个线程结束等待，发出信号，前五个线程中同时收到该信号开始继续执行，而这六个线程都没有后续任务，所以都会结束。

Invoke函数

在Windows下，窗体（Form）是由单独的线程控制的，这意味着，我们如果在控件的触发逻辑代码中新建了线程，这个线程是没有办法修改我们现有的窗体控件属性的。

这时为了达到修改窗体控件属性的效果，我们需要调用控件的Invoke函数。Invoke方法首先检查发出调用的线程(即当前线程)是不是UI线程，如果是，直接执行委托指向的方法，如果不是，它将切换到UI线程，然后执行委托指向的方法。

```

1.创建一个委托
public delegate void AppendTextBoxDelegate(string msg);
2.创建一个方法作为委托的具体实现
public void AppendTextBox(string msg)
{
    textBox3.AppendText(msg);
}
3.在需要的时候调用Invoke方法，传入的2个参数分别为委托对象和委托的参数
private void button16_Click(object sender, EventArgs e) {
    textBox3.Invoke(new AppendTextBoxDelegate(AppendTextBox), "期末复习要注意休息呀\r\n");
}

```

生产者、消费者问题

教材配套代码如下

```

private static Mutex mut;
private static int SharedBuffer;
private static int BufferState;
private static Thread[] threadVec;
private static AutoResetEvent hNotFullEvent, hNotEmptyEvent;
public const int FULL = 1;
public const int EMPTY = 0;

public static int ProdecerIterater;

```

```

private static void Consumer()
{
    //int result;
    while (true)
    {
        mut.WaitOne();
        if (BufferState == EMPTY)
        { // nothing to consume
            mut.ReleaseMutex();
            // wait until buffer is not empty
            hNotEmptyEvent.WaitOne();
            continue; // return to while loop to contend for Mutex
        }
    }
}

```

```

private static void Producer()
{
    //int ProdecerIterater;

    for (ProdecerIterater = 20; ProdecerIterater >= 0;
ProdecerIterater--)
    {
        while (true)
        {
            mut.WaitOne();

            if (BufferState == FULL)
            {
                mut.ReleaseMutex();
                hNotFullEvent.WaitOne();
                continue; // back to loop to test BufferState again
            }
            // got mutex and buffer is not FULL, break out of while loop
            break;
        }
        //end of while
        // got Mutex, buffer is not full, producing data
        //Console.WriteLine("Produce: {0}", ProdecerIterater);
        SendMessage(main_Whandle, PRODUCERR_IN_LASTAIM, 0,
0);

        SharedBuffer = ProdecerIterater;
        BufferState = FULL;
        mut.ReleaseMutex();
        hNotEmptyEvent.Set();
        Thread.Sleep(1000);
    }
    //end of for
}
//end of Producer thread

again
}

//这里接上面的Consumer函数!!!
if (SharedBuffer == 0)
{
    // test for end of data token
    //Console.WriteLine("Consumed {0}: end of data\r\n", SharedBuffer);
    SendMessage(main_Whandle, CONSUMER_ENDOFDATA_IN_LASTAIM, 0, 0);
    mut.ReleaseMutex();
    break;
}
else
{
    //result = SharedBuffer;
    //Console.WriteLine("Consumed: {0}", result);
    SendMessage(main_Whandle, CONSUMER_IN_LASTAIM, 0, 0);
    BufferState = EMPTY;
    mut.ReleaseMutex();
    hNotFullEvent.Set();
}
}
//end of while
}
//end of Consumer thread

private void button18_Click(object sender, EventArgs e)
{
    textBox3.Clear();
    SharedBuffer = 20;
    mut = new Mutex(false, "Tr");
    hNotFullEvent = new AutoResetEvent(false);
    hNotEmptyEvent = new AutoResetEvent(false);
    threadVec = new Thread[2];
    threadVec[0] = new Thread(new ThreadStart(Consumer));
    threadVec[1] = new Thread(new ThreadStart(Producer));
    threadVec[0].Start();
    threadVec[1].Start();
    threadVec[0].Join();
    threadVec[1].Join();
}

```

WinForm使用消息机制通信

1. 定义结构体

//lpData是我们传输的字符串数据

```

public struct COPYDATASTRUCT
{
    public IntPtr dwData;
    public int cbData;
    [MarshalAs(UnmanagedType.LPStr)]
    public string lpData;
}

```

消息发送者

1. 调用系统动态连接库中的SendMessage函数和FindWindow函数

```

[DllImport("User32.dll", EntryPoint = "SendMessage")]
private static extern int SendMessage(IntPtr hWnd, int Msg, int wParam, ref COPYDATASTRUCT lParam);
[DllImport("User32.dll", EntryPoint = "FindWindow")]
private static extern int FindWindow(string lpClassName, string lpWindowName);

```

2. 寻找消息接收者

```
int hWnd = FindWindow(null, @"消息接受者");
```

3. 调用SendMessage函数发送信息

```

public const int WM_COPYDATA = 0x004A;
private void button3_Click(object sender, EventArgs e)
{
    //byte[] sarr = System.Text.Encoding.Default.GetBytes(txtString.Text);
    byte[] sarr = Encoding.Default.GetBytes(messageSendTextBox.Text);
    int len = sarr.Length;
    COPYDATASTRUCT cds;
    cds.dwData = (IntPtr)Convert.ToInt16(0); //可以是任意值
    cds.cbData = len + 1; //指定lpData内存区域的字节数
    cds.lpData = messageSendTextBox.Text; //发送给目标窗口所在进程的数据
    SendMessage(hWnd, WM_COPYDATA, 0, ref cds); //main_Whandle是消息接

```

消息接受者

1. 覆盖从父类中继承的消息处理函数DefWndProc

```

protected override void DefWndProc(ref Message m)
{
    switch (m.Msg)
    {
        case WM_COPYDATA:
            messageReceiveTextBox.Clear();
            COPYDATASTRUCT cds = new COPYDATASTRUCT();
            Type t = cds.GetType();
            cds = (COPYDATASTRUCT)m.GetLPParam(t);
            string strResult = cds.lpData; //得到发送的消息
            messageReceiveTextBox.AppendText(strResult);
            break;
        default:
            base.DefWndProc(ref m);
            break;
    }
}

```

受窗体的句柄
}

WinForm事件机制通信

1. 创建委托

```
public delegate void MessageEventHandle(string msg);
```

消息发送者

触发消息接受者的事件

```
SendEvent(eventSendTextBox.Text);
```

注意：在实验中，消息接受者和消息触发者是同一个窗体，所以直接调用SendEvent即可，如果不是同一个窗体，则需要先得消息接受者窗体对象，然后调用该方法，即类似form1.SendEvent(eventSendTextBox.Text);的形式。

因为不确定到时具体怎么创建两个对象和保存状态，这里先没有写详细过程。PPT第5章第61页有基于继承的实现，我觉得考试时最好能在一个窗体创建时直接作为另一个窗体的成员变量存起来。

消息接受者

1. 为委托注册事件

```
public event MessageEventHandle SendEvent;
```

2. 实现事件的具体处理逻辑

```
SendEvent += (string msg) =>  
{  
    eventReceiveTextBox.Clear();  
    eventReceiveTextBox.AppendText(msg);  
};
```

WPF消息通信机制

1. 定义结构体

```
public struct COPYDATASTRUCT  
{  
    public IntPtr dwData;  
    public int cbData;  
    [MarshalAs(UnmanagedType.LPStr)]  
    public string lpData;  
}
```

消息发送者（和WinForm的消息发送者完全相同）

1. 用系统动态连接库中的SendMessage函数和FindWindow函数

```
[DllImport("User32.dll", EntryPoint = "SendMessage")]  
private static extern int SendMessage(IntPtr hWnd, int Msg, int wParam, ref COPYDATASTRUCT lParam);  
[DllImport("User32.dll", EntryPoint = "FindWindow")]  
private static extern int FindWindow(string lpClassName, string lpWindowName);
```

2. 寻找消息接收者

```
int hWnd = FindWindow(null, @"消息接受者");
```

3. 调用SendMessage函数发送信息

```
public const int WM_COPYDATA = 0x004A;  
private void button3_Click(object sender, EventArgs e)  
{  
    //byte[] sarr = System.Text.Encoding.Default.GetBytes(txtString.Text);  
    byte[] sarr = Encoding.Default.GetBytes(messageSendTextBox.Text);  
    int len = sarr.Length;  
    COPYDATASTRUCT cds;  
    cds.dwData = (IntPtr)Convert.ToInt16(0); //可以是任意值  
    cds.cbData = len + 1; //指定lpData内存区域的字节数  
    cds.lpData = messageSendTextBox.Text; //发送给目标窗口所在进程的数
```

据

```
    SendMessage(hWnd, WM_COPYDATA, 0, ref cds); //main_whoandle是消息接受窗体的句柄  
}
```

消息接收者

1. 创建一个消息处理方法对收到的消息进行处理，返回值是

IntPtr类型

```
IntPtr WndProc(IntPtr hWnd, int msg, IntPtr wParam, IntPtr lParam, ref bool handled)  
{  
  
    if (msg == WM_COPYDATA)  
    {  
        messageReceiveTextBox.Clear();  
        COPYDATASTRUCT cds = (COPYDATASTRUCT)  
System.Runtime.InteropServices.Marshal.PtrToStructure(lParam,  
typeof(COPYDATASTRUCT));  
        messageReceiveTextBox.AppendText(cds.lpData);  
    }  
  
    return hWnd;
```

2. 在主窗体加载时，为我们创建的消息处理方法注册钩子事件，使其能够接受消息

```
private void mainWindowLoad(object sender, RoutedEventArgs e)  
{  
    (PresentationSource.FromVisual(this) as  
System.Windows.Interop.HwndSource).AddHook(new  
System.Windows.Interop.HwndSourceHook(WndProc));  
    define_Event();  
}
```

WPF事件机制通信

和WinForm事件机制通信完全一样，在上面有。

使用OLEDB对Excel数据进行管理

1.创建OleDbConnection 对象

```
String path = @"..\..\hotel.xls"
```

```
excelConnString = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" + path + ";Extended Properties='Excel 12.0 Xml;HDR=YES';";
```

```
OleDbConnection excleConn = new OleDbConnection(excelConnString);
```

2.打开连接，进行查询，并将查询保存到一个DataTable对象

```
public DataTable ReadToTableExcelSheet_1()//excel存放的路径
```

```
{
    try
    {
        excleConn.Open();
        DataTable sheetsName = excleConn.GetOleDbSchemaTable(OleDbSchemaGuid.Tables, new object[] { null, null, null, "Table" }); //得到所有sheet的名字

        string firstSheetName = sheetsName.Rows[0][2].ToString(); //得到第一个sheet的名字

        string sql = string.Format("SELECT * FROM [{0}]", firstSheetName); //查询字符串
        OleDbDataAdapter ada = new OleDbDataAdapter(sql, excelConnString);
        DataSet set = new DataSet();
        ada.Fill(set);
        excleConn.Close();
        return set.Tables[0];
    }
    catch (Exception)
    {
        excleConn.Close();
        return null;
    }
}
```

3.将上一步中得到的DataTable对象给dataGridView作为数据源

```
DataTable dt = ReadToTableExcelSheet_1();
```

```
dataGridView1.DataSource = dt;
```

如果需要更改excel表，可以修改第一步中path的值

如果需要更改查询条件，可以修改第二步中的sql语句

使用DataAdapter操作SQLite数据库

1.创建SQLiteConnection 对象和MySqlDataAdapter对象

```
public SQLiteConnection createSqliteConn(string path)
```

```
{
    sqliteConnString = "Data Source =" + path + "; Version = 3;";
    SQLiteConnection conn = new SQLiteConnection(sqliteConnString);
    return conn;
}
```

```
MySqlConnection sqliteConn = createSqliteConn(@"..\..\demo.db");
```

```
MySqlDataAdapter sqliteAdapter = new SQLiteDataAdapter("SELECT FILE_NO as 文件编号,ID_KEY as 文件名称,SUBJECT as 主题,PUBLISH_DATE as 发布时间,IMPLEMENT_DATE as 施行时间,PUBLISH_ORG as 发布单位,FILE_NAME as 文件全名 FROM [tbl_fgwj]", sqliteConn);
```

2.创建DataTable对象并用sqliteAdapter 去填充DataTable对象

```
sqliteTable = new DataTable(); // Don't forget initialize!
```

```
sqliteAdapter.Fill(sqliteTable);
```

3.将sqliteTable设置为dataGridView3的数据源

```
dataGridView3.DataSource = sqliteTable;
```

4.如果我们修改了dataGridView3并点击了保存按钮，则可以通过SQLiteCommandBuilder对象和sqliteAdapter对数据库进行修改

```
private void button6_Click(object sender, EventArgs e)
{
    SQLiteCommandBuilder scb = new SQLiteCommandBuilder(sqliteAdapter);
    sqliteAdapter.Update(sqliteTable);
}
```

5.若是需要从代码中增加数据库一条记录，可以参考如下代码

```
public void addSqliteRow(string file_name, string file_no, string publish_org, string publish_date, string implement_date, string subject, string full_name)
{
    sqliteTable.Rows.Add(file_no, file_name, subject, publish_date, implement_date, publish_org, full_name);
    SQLiteCommandBuilder scb = new SQLiteCommandBuilder(sqliteAdapter);
    sqliteAdapter.Update(sqliteTable);
}
```

6.若是需要从代码中删除数据库中的记录，可以参考如下代码

```
private void button7_Click(object sender, EventArgs e)
{
    //从下往上删，避免沙漏效应
    for (int i = dataGridView3.SelectedRows.Count - 1; i >= 0; i--)
    {
        dataGridView3.Rows.RemoveAt(dataGridView3.SelectedRows[i].Index);
    }
}
```



```

        SQLiteCommandBuilder scb = new SQLiteCommandBuilder(sqliteAdapter);
        sqliteAdapter.Update(sqliteTable);
    }

```

7.若是需要dataGridView中的值或对其进行修改，可使用如下代码

```
dataGridView1.Rows[i].Cells["第1列列名"].Value
```

修改完后仍需要记得使用sqliteAdapter.Update(sqliteTable);保存修改

使用DataReader访问数据库

1.创建MySqlConnection对象

```
String mysqlConnectStr = "server=127.0.0.1;port=3306;user=" + userTextBox.Text + ";password=" + secretTextBox.Text +
";Database=windowsprogramdesignscheme;";
```

```
MySqlConnection mySqlConn= new MySqlConnection(connetStr);
```

2.打开连接并创建MySqlDataReader对象

```
mySqlConn.Open();
```

```
MySqlCommand mySqlCommand = new MySqlCommand("select * from book", mySqlConn);
```

```
MySqlDataReader mySqlReader = mySqlCommand.ExecuteReader();
```

3.使用mysqlReader读取列名并加入DataTable对象中

```
public DataTable mySqlTable = new DataTable();;
```

```
string[] attr = new string[mysqlReader.FieldCount];
```

```
for (int i = 0; i < mysqlReader.FieldCount; i++)
```

```
{
    attr[i] = mysqlReader.GetName(i);
}
```

```
mySqlTable = new DataTable();
```

```
for (int i = 0; i < mysqlReader.FieldCount; i++)
```

```
{
    mySqlTable.Columns.Add(attr[i]);
}
```

4.循环读取下一行并添加到中

```
object[] idx = new object[mysqlReader.FieldCount];
```

```
while (mysqlReader.Read())
```

```
{
    mysqlReader.GetValues(idx);
    mySqlTable.Rows.Add(idx);
}
```

5.将mySqlTable作为dataGridView的数据源

```
dataGridView4.DataSource = mySqlTable;
```