第一章 概述

网络操作系统

在计算机网络上配置网络操作系统NOS（Network Operating System），是为了管理网络中的共享资源，实现用户通信以及方便用户使用网络，因而网络操作系统是作为网络用户与网络系统之间的接口。

网络操作系统的特性

（1）客户/服务器模式;（2）32位/64位操作系统;（3）抢先式多任务;（4）支持多种文件系统;（5）高可靠性;（6）安全性;（7）容错性;（8）开放性;（9）可移植性;（10）图形化界面GUI;（11）Internet支持：集成了许多标准化网络应用

网络操作系统的功能

（1）网络通信;（2）资源管理：对网络中的共享资源（硬件和软件）实施有效的管理、协调诸用户对共享资源的使用、保证数据的安全性和一致性;（3）网络服务;（4）网络管理;（5）互操作能力;

Windows操作系统的主要特点

- 面向对象：在Windows中，窗口、菜单、事件皆是对象。
- 消息/事件驱动：用户操作系统事件 --->系统消息队列--->应用消息队列--->应用程序消息处理函数。
- 资源共享与数据交换
- 与设备无关的 API

Windows 应用程序类型

VC++:
（1）基于控制台的应用程序;（2）基于对话框的应用程序（3）单文档应用程序;（4）多文档应用程序;（5）基于html的应用程序;

C#：
（1）Windows窗体应用程序;（2）WPF应用程序;（3）Windows服务程序;（4）WCF服务应用程序;（5）ASP.NET Web应用程序;

各个应用程序有哪些特点?

（1）控制台应用程序，特点：运行控制台应用程序时系统会创建控制台终端用于用户交互，交互方式限于字符形式，屏幕利用率极低。

（2）窗体应用程序，特点：使用真彩色图形化输出，支持鼠标定位，能及时响应用户操作，操作方式便利，输出效果绚丽。

（3）动态链接库，特点：不能独立运行，必须由其他运行中的程序调用才可运行。

（4）服务程序，特点：服务程序注册到用户的计算机中，随机器启动并自动执行，用户不能直接运行或终止服务程序；服务程序没有运行界面，也不与用户交互，用户只能通过特殊控制命令使其启动或终止运行。

（5）Web应用程序，特点：基于 HTML 语言的网页程序，将程序控制语句嵌入在标准的 HTML 文本中。

第二章 进程与进程间通信

进程控制块

在进程控制块中，主要包括以下几个方面的信息：
（1）进程标识符：进程标识符用于唯一的表示一个进程。
（2）处理机上下文：主要是处理机寄存器中的内容。
（3）进程调度信息：①进程状态；②进程优先级；③阻塞事件；④进程调度所需的其他信息。
（4）进程控制信息：①程序和数据的地址；②进程同步和通信机制；③资源清单；④下一个PCB链接指针。

Windows 进程创建过程

- 打开文件映像（.exe）。
- 创建windows进程对象。
- 创建初始线程对象，包括上下文，堆栈。
- 通知内核系统为进程运行作准备。
- 执行初始线程。
- 导入需要的 DLL，初始化地址空间，由程序入口地址开始执行进程。

```
// example 1
Process cmdP = new Process();
cmdP.StartInfo.FileName = "cmd.exe";
cmdP.StartInfo.CreateNoWindow = true;
cmdP.StartInfo.UseShellExecute = false;
cmdP.StartInfo.RedirectStandardOutput = true;
cmdP.StartInfo.RedirectStandardInput = true;
cmdP.Start();

// example 2
Process.Start("iexplore","http://www.baidu.com");

// 得到程序中所有正在运行的进程
Process[] preo = Process.GetProcesses();
foreach (var item in preo){
                Console.WriteLine(item);
                item.Kill(); //杀死进程
}
```

Windows 进程间通信方法

- 共享内存（剪贴板、COM、DLL、DDE、文件映射）；
- 消息 WM_COPYDATA；
- 邮槽；
- 管道：分有名管道与无名管道、进程重定向；
- Windows套接字；
- NetBIOS 特殊的网络应用；

IPC需要考虑的内容

- 进程是否会通过网络与其它机器上的进程通信，仅使用本机通信机制是否满足应用需求；
- 通信中的进程是否是处于不同的操作系统平台例如Windows与UNIX平台；
- 有些进程通信机制是只用于图形化窗体界面的，而不适用于控制台程序；
- 通信目的是用于同步控制还是数据的传送；
- 数据传输量考虑；

消息机制例子

发送消息函数

```
[DllImport("User32.dll", EntryPoint = "SendMessage")]
static extern int SendMessage(IntPtr wnd, int msg, IntPtr wP, IntPtr lP);

[DllImport("User32.dll", EntryPoint = "SendMessage")]
static extern int SendMessage(IntPtr wnd, int msg, IntPtr wP, ref
COPYDATASTRUCT lParam);

[DllImport("User32.dll", EntryPoint = "PostMessage")]
static extern int PostMessage(IntPtr wnd, int msg, IntPtr wP, ref
COPYDATASTRUCT lParam);


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

# 定义消息类型
const WM_COPYDATA = 0x004A

# 发送消息
private void button1_Click(object sender, EventArgs e)
{
    int hWnd = FindWindow(null, @"Receiver");
    if (hWnd == 0)
    {
        MessageBox.Show("555，未找到消息接受者！");
    }
    else
    {
        byte[] sarr = System.Text.Encoding.Default.GetBytes(textBox1.Text);
        int len = sarr.Length;
        COPYDATASTRUCT cds;
        cds.dwData = (IntPtr)9;      // 可以是任意值
        cds.cbData = len + 1;        // 指定lpData内存区域的字节数
        cds.lpData = textBox1.Text; // 发送给目标窗口所在进程的数据
        SendMessage(hWnd, WM_COPYDATA, 0, ref cds);
    }
}
```

Winform 处理消息例子

```
protected override void DefWndProc(ref System.Windows.Forms.Message m)
{
    switch (m.Msg)
    {
        case WM_COPYDATA:
            COPYDATASTRUCT mystr = new COPYDATASTRUCT();
            Type mytype = mystr.GetType();
```

```
            mystr = (COPYDATASTRUCT)m.GetLParam(mytype);
            this.textBox1.Text = mystr.lpData;
            break;
        default:
            base.DefWndProc(ref m);
            break;
    }
}
```

WPF 处理消息例子

```
//页面加载时，添加消息处理钩子函数
private void ChildPage_Loaded(object sender, RoutedEventArgs e)
{
    HwndSource hWndSource;
    WindowInteropHelper wih = new WindowInteropHelper(this.parentWindow);
    hWndSource = HwndSource.FromHwnd(wih.Handle);
    //添加处理程序
    hWndSource.AddHook(MainWindowProc);
}
//钩子函数，处理所收到的消息
private IntPtr MainWindowProc(IntPtr hwnd, int msg, IntPtr wParam, IntPtr
lParam, ref bool handled)
{
    switch (msg)
    {
        case WM_COPYDATA:
            COPYDATASTRUCT mystr = new COPYDATASTRUCT();
            Type mytype = mystr.GetType();
            COPYDATASTRUCT MyKeyboardHookStruct =
(COPYDATASTRUCT)Marshal.PtrToStructure(lParam, typeof(COPYDATASTRUCT));
            showComment(MyKeyboardHookStruct.lpData);
            break;
        default:
            break;
    }
    return hwnd;
}
```

重定向例子

```
Process process = new Process();
process.StartInfo.FileName = "cmd.exe";
// 是否使用外壳程序
process.StartInfo.UseShellExecute = false;
// 是否在新窗口中启动该进程的值
process.StartInfo.CreateNoWindow = true;
// 重定向输入流
process.StartInfo.RedirectStandardInput = true;
// 重定向输出流
process.StartInfo.RedirectStandardOutput = true;
//使ping命令执行九次
string strCmd = "ping www.whu.edu.cn -n  9";
process.Start();
process.StandardInput.WriteLine(strCmd);
process.StandardInput.WriteLine("exit");
```

```
// 获取输出信息
textBox2.Text = process.StandardOutput.ReadToEnd();
process.WaitForExit();
process.Close();
```

第三章 线程间通信与同步

线程基本概念

- 进程是计算机分配资源的单位，线程是运行调度单位。
- 进程中的线程也具有线程控制块，包含内容有所属进程ID，创建和退出时间，线程启动地址等

线程创建过程

- 在进程的地址空间中为线程创建用户态堆栈。
- 初始化线程硬件上下文。
- 创建线程对象。
- 通知内核系统为线程运行准备。
- 新创建线程handle和线程ID值返回到调用者。
- 线程进入调度准备执行。

创建线程示例：

```
// 线程执行代码的编写
void workThread(){}
// 设定函数名为线程入口
ThreadStart s = new ThreadStart(workThread);
// 线程委托对象(委托的实质是函数指针或叫函数地址)
Thread thread1 = new Thread(s);
//设定线程优先级等属性
// 线程启动
thread1.Start();
//线程参数传递
thread1.Start(paraObject);
```

线程创建示例2:

```
//通过匿名委托创建
Thread thread1 = new Thread(delegate () { Console.WriteLine("我是通过匿名委托创建的线程"); });
thread1.Start();

//通过Lambda表达式创建
Thread thread2 = new Thread(() => Console.WriteLine("我是通过Lambda表达式创建的委托"));
thread2.Start();
```

线程创建示例3，利用有参的委托 ParameterizedThreadStart 来创建线程：

```
class Program
{
    static void Main(string[] args)
    {
        //通过ParameterizedThreadStart创建线程
```

```
        Thread thread = new Thread(new ParameterizedThreadStart(Thread1));
        //给方法传值
        thread.Start("这是一个有参数的委托");
        Console.ReadKey();
    }
    // 创建有参的方法，方法里面的参数类型必须是Object类型
    static void Thread1(object obj)
    {
        Console.WriteLine(obj);
    }
}
```

线程的状态

- 初始化--线程处于创始中。
- 就绪--等待由CPU执行。
- 待命--只能由一个线程处于待命状态，离执行状态最近。
- 运行--在CPU的当前时间片内执行。
- 等待--线程同步需要等待，
- 接转--准备执行，但是它的内核堆栈不在内存，需要内存页面调入，调入后进入就绪状态。
- 终止--线程执行完。

线程应用场景

- 网络通信程序。
- 与Web服务器和数据库操作。
- 执行占用大量时间的操作。
- 有不同优先级的任务。
- 用户响应效能与数据运算均衡。

异步线程回调示例

```
// 定义一个回调
AsyncCallback callback = p =>
{
    Console.WriteLine($"到这里计算已经完成了。
{Thread.CurrentThread.ManagedThreadId.ToString("00")}。");
    update($"到这里计算已经完成了。" +
Thread.CurrentThread.ManagedThreadId.ToString("00") + "。");
};
//异步调用回调
for (int i = 0; i < 5; i++)
{
    string name = string.Format($"btnSync_Click_{i}");
    asyncResult = action.BeginInvoke(name, callback, null);
}
```

需要同步的资源

- 系统资源（如通信端口）。
- 多个进程所共享的资源（如文件句柄）。
- 由多个线程访问的单个应用程序域的资源（如全局、静态和实例字段）。

Winform 窗体线程与工作线程通信方法

窗体线程发：`ManualResetEvent.Set`

工作线程收：`While(WaitHandle.WaitOne)`

生产者消费者问题示例

```
    static Mutex mutex = new Mutex();
    const int BUFFER_SIZE = 20;
    static Semaphore produce = new Semaphore(BUFFER_SIZE, BUFFER_SIZE);//初始资源数，最大资源数
    static Semaphore consume = new Semaphore(0, BUFFER_SIZE);

    // 生产者线程
    private void produceItem(string name, int rate)
    {
        while (true)
        {
            while (!produce.WaitOne(10))
            {
                Console.WriteLine(name + " wants to produce an item, but the buffer is full");
            }
            mutex.WaitOne();
            ++curCnt;
            remainedCnt++;
            Console.WriteLine(name + " produces an item, totally " + curCnt + ", now there are " + remainedCnt + " items in the buffer");
            mutex.ReleaseMutex();
            consume.Release();
            Thread.Sleep(rate * 1000);

            if (curCnt >= totalCnt) break;
        }
    }
    // 消费者线程
    private void consumeItem(string name, int rate)
    {
        while (true)
        {
            while (!consume.WaitOne(10))
            {
                Console.WriteLine(name + " wants to consume an item, but the buffer is empty");
            }
            mutex.WaitOne();
            remainedCnt--;
            Console.WriteLine(name + " consumes an item, now there are " + remainedCnt + " items");
            mutex.ReleaseMutex();
            produce.Release();
            Thread.Sleep(rate * 1000);

            if (curCnt >= totalCnt) break;
        }
    }
    // 总体逻辑
```

```
    void ProducerConsumer(int producerCnt, int consumerCnt, int productRate, int consumeRate
        , int buffer_size, int _totalProductNum)
    {
        Console.WriteLine("begin of producer consumer problem");
        totalCnt = _totalProductNum;
        Thread[] producers = new Thread[producerCnt];
        for (int i = 0; i < producerCnt; i++)
        {
            producers[i] = new Thread(() => { produceItem("producer" + i, productRate); });
            producers[i].IsBackground = true;
            producers[i].Start();
        }
        Thread.Sleep(2000);

        Thread[] consumers = new Thread[consumerCnt];
        for (int i = 0; i < consumerCnt; i++)
        {
            consumers[i] = new Thread(() => { consumeItem("consumer" + i, consumeRate); });
            consumers[i].IsBackground = true;
            consumers[i].Start();
        }
        Console.WriteLine("End of producer consumer problem");
    }
```

第四章 文件系统

相关概念

- 文件系统是操作系统用于明确存储设备（磁盘、固态硬盘）或分区上的文件的方法和数据结构；即在存储设备上组织文件的方法。
- 文件系统由三部分组成：文件系统的接口，对对象操纵和管理的软件集合，对象及属性。

Windows支持的文件系统

（1）FAT；（2）NTFS；（3）CDFS；（4）UDF；

FAT 文件系统

FAT 文件系统优点：

- 文件系统所占容量与计算机的开销少；
- 支持各种操作系统——可移植；
- 方便的用于传送数据。

FAT 文件系统缺点：

- 容易受损害
- 单用户：不保存文件的权限信息；只包含隐藏、只读等公共属性
- 非最佳更新策略：在磁盘的第一个扇区保存其目录信息
- 没有防止碎片的措施
- 文件名长度受限

NTFS 文件系统

NTFS 文件系统优点：

- 更安全的文件保障，提供文件加密，能够大大提高信息的安全性。
- 更好的磁盘压缩功能。
- 支持最大达2TB的大硬盘，并且随着磁盘容量的增大，NTFS的性能不像FAT那样随之降低。
- 可以赋予单个文件和文件夹权限。
- 强大的容错与回复机制。
- 支持活动目录和域。此特性可以帮助用户方便灵活地查看和控制网络资源。
- 支持稀疏文件。NTFS只需要为大文件实际写入的数据分配磁盘存储空间。
- 支持磁盘配额。磁盘配额可以管理和控制每个用户所能使用的最大磁盘空间。

第五章 注册表

注册表中记录了用户安装在计算机上的软件和每个程序的相关信息，通过它可以控制硬件、软件、用户环境和操作系统界面。注册表数据保存在 system.dat 和 user.dat 中，利用 regedit.exe 程序能够编辑。

注册表结构

根键：这个称为HKEY.....，某一项的句柄项：附加的文件夹和一个或多个值
子项：在某一个项（父项）下面出现的项（子项）
值项：带有一个名称和一个值的有序值，每个项都可包括任何数量的值项，值项由三个部分组成：名称、数据类型和数据。
1、名称：不包括反斜线的字符、数字、代表符和空格的任意组合。同一键中不可有相同的名称
2、数据类型：包括字符串、二进制和双字节等
3、数据：值项的具体值，它的大小可以占用64KB

根键

注册表包括以下5个根键
1.HKEY_CLASSES_ROOT
说明：该根键包括启动应用程序所需的全部信息，包括扩展名，应用程序与文档之间的关系，驱动程序名，DDE和OLE信息，类ID编号和应用程序与文档的图标等。
2.HKEY_CURRENT_USER
说明：该根键包括当前登录用户的配置信息，包括环境变量，个人程序以及桌面设置等。
3.HKEY_LOCAL_MACHINE
说明：该根键包括本地计算机的系统信息，包括硬件和操作系统信息，安全数据和计算机专用的各类软件设置信息。
4.HKEY_USERS
说明：该根键包括计算机的所有用户使用的配置数据，这些数据只有在用户登录系统时才能访问。这些信息告诉系统当前用户使用的图标，激活的程序组，开始菜单的内容以及颜色，字体。常用的：

1）AppEvents子项：它包括了各种应用事件的列表：EventLabels:按字母顺序列表；Schemes:按事件分类列表
2）Control Panel子项：它包括内容与桌面、光标、键盘和鼠标等设置有关

5.HKEY_CURRENT_CONFIG
说明：该根键包括当前硬件的配置信息，其中的信息是从HKEY_LOCAL_MACHINE中映射出来的。

看是五个分支，其实就是HKEY_LOCAL_MACHINE、HKEY_USERS这两个才是真正的注册表键，其它都是从某个分支映射出来的，相当于快捷方式或是别名。

注册表的基本操作

1、创建项和项值
2、更值项的数据
3、删除项、子项或值项
4、查找项、值项或数据

第六章 动态连接库

两种链接方式

- 静态链接方式：**在程序编译时**，将各种目标模块（.OBJ）文件、运行时库（.LIB）文件，以及已编译的资源（.RES）文件链接在一起，以便创建Windows 的 .EXE 文件。
- 动态链接方式：**在程序运行时**，Windows 把一个模块中的函数调用链接到动态链接库模块（.DLL）中的实际函数的过程。

链接一个DLL有两种方式

- 载入时动态链接（Load-Time Dynamic Linking）
- 运行时动态链接（Run-Time Dynamic Linking）

静态链接库的优点

- 代码装载速度快，执行速度略比动态链接库快；
- 只需保证在开发者的计算机中有正确的.LIB文件，在以二进制形式发布程序时不需考虑在用户的计算机上.LIB文件是否存在及版本问题，可避免DLL地狱等问题。

动态链接库的优点

- 更加节省内存并减少页面交换；
- DLL文件与EXE文件独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性；
- 不同编程语言编写的程序只要按照函数调用约定就可以调用同一个DLL函数。

不足之处

- 使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费；
- 使用动态链接库的应用程序不是自完备的，它依赖的DLL模块也要存在，如果使用载入时动态链接，程序启动时发现DLL不存在，系统将终止程序并给出错误信息。而使用运行时动态链接，系统不会终止，但由于DLL中的导出函数不可用，程序会加载失败；
- 使用动态链接库可能造成DLL地狱。

Windows中主要的 DLL

KERNEL32.DLL	**低级内核函数，用于内存管理、任务管理、资源控制等**
USER32.DLL	windows管理有关的函数，消息、菜单、光标、计时器、通信，钩子等
GDI32.DLL	图形设备接口库。
ODBC32.DLL	ODBC功能
Ws2_32.dll	socket通信功能

用 C++ 创建 DLL 示例

```
// mydll.h
extern "C" _declspec(dllexport) int add(int a, int b);

//mydll.cpp
#include "mydll.h"
int add(int a, int b)
{
    return a + b;
}
```

参考：https://www.cnblogs.com/chechen/p/8676226.html

使用示例：

```
[DllImport("mydll.dll")]
public extern static int myAdd(int a, int b);
```

使用的时候要在项目依赖项上面添加引用。

反射机制

通过 System.Reflection 命名空间中的类以及 System.Type，可以获取有关已加载的程序集和在其中定义的类型（如类、接口和值类型）的信息。也可以使用反射在运行时创建类型实例，调用和访问这些实例。

反射机制用途：

- 使用 MethodInfo 发现以下信息：方法的名称、返回类型、参数、访问修饰符（如 `public` 或 `private`）和实现详细信息（如 `abstract` 或 `virtual`）等。使用 Type 的 GetMethods 或 GetMethod 方法来调用特定的方法。
- 使用 FieldInfo 发现以下信息：字段的名称、访问修饰符和实现详细信息（如 `static`）等；并获取或设置字段值。
- 使用 EventInfo 发现以下信息：事件的名称、事件处理程序数据类型、自定义属性、声明类型和反射类型等；并添加或移除事件处理程序。
- 使用 PropertyInfo 发现以下信息：属性的名称、数据类型、声明类型、反射类型和只读或可写状态等；并获取或设置属性值。
- 使用 ParameterInfo 发现以下信息：参数的名称、数据类型、参数是输入参数还是输出参数，以及参数在方法签名中的位置等。

上机作业

- 使用windows操作系统提供的DLL，实现对注册表的操作；
- 使用C++创建DLL实现简单的功能，并在C#环境下调用该DLL；
- 使用C#创建DLL实现简单的功能，并在C#环境下调用该DLL。

第七章 Windows COM 原理与技术

COM 相关概念

- COM组件是以WIN32动态链接库（DLL）或可执行文件（EXE）形式发布的可执行代码组成。
- COM组件是遵循COM规范编写的
- COM组件是一些小的二进制可执行文件
- COM组件可以给应用程序、操作系统以及其他组件提供服务
- 自定义的COM组件可以在运行时刻同其他组件连接起来构成某个应用程序

- COM组件可以动态的插入或卸出应用
- COM组件必须是动态链接的
- COM组件必须隐藏（封装）其内部实现细节
- COM组件必须将其实现的语言隐藏
- COM组件必须以二进制的形式发布

COM 与 DLL 区别

- DLL是对静态连接的一种改进，带来了更细的开发分工，包括二进制如何交互的问题，尤其是当DLL输出类时的二进制交互问题；
- COM的各种努力都是在规定一种二进制交互协议。
- DLL 是以函数集合的方式来调用的，是编程语言相关的，如：VC 必须加上extern “C“；DLL 是基于名字导入的，名字就是符号，DLL有符号表
- COM 是以 interface 的方式提供给用户使用的是一种二进制的调用规范，是与编程语言无关的。
- DLL 只有 DLL 一种形载体， 在里面定义函数无限制， 只能运行在本机上
- COM 有 DLL 和 EXE 两种载体。COM所在的DLL中必须导出四个函数: `dllgetobjectclass, dllregisterserver, dllunregisterserver, dllunloadnow`。

COM 组件定义示例

（1）定义接口：

```
namespace DeviceInterfaces
{
    [Guid("9EDA6EA7-BB80-4B78-AE68-0C01C966F72D")]
    [ComVisible(true)]
    public interface ITransaction
    {
        void Connect(string connectString);

        void Disconnect();

        string GetVersion();

        string add(int a, int b);
        string multi(int a, int b);
    }
}
```

（2）实现接口

```
namespace DeviceInterfaces
{
    [Guid("D61A457C-DBEF-43DE-80F4-394703BD3D41")]
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    [Description("模拟事务记录")]
    public class SimTransaction : ITransaction
    {
        public void Connect(string connectString){}
        public void Disconnect(){}
        public string GetVersion()
        {
            return "1.0";
        }
```

```
        public string add(int a, int b)
        {
            return string.Concat(a, "+", b, "=", a + b);
        }
        public string multi(int a, int b)
        {
            return string.Concat(a, "*", b, "=", a * b);
        }
    }
}
```

(3) 创建接口对象:

```
class ComTest
{
    public static ITransaction CreateTransaction(string _guid, string connectionStr)
    {
        Console.Write("begin creating transaction");
        ITransaction iTransaction = null;
        try
        {
            Guid guid = new Guid(_guid);
            Type transactionType = Type.GetTypeFromCLSID(guid);
            object transaction = Activator.CreateInstance(transactionType);
            iTransaction = transaction as ITransaction;
            iTransaction.Connect(connectionStr);
        }
        catch (Exception ex)
        {
            Console.Write(ex.ToString());
        }
        Console.Write("end creating transaction");
        return iTransaction;
    }
    public static string add(string guid, string connectionStr, int a, int b)
    {
        ITransaction transaction = CreateTransaction(guid, connectionStr);
        return transaction.add(a, b);
    }
    public static string multi(string guid, string connectionStr, int a, int b)
    {
        ITransaction transaction = CreateTransaction(guid, connectionStr);
        return transaction.multi(a, b);
    }
}
```

(4) 具体使用:

```
private void btn1_Click_1(object sender, RoutedEventArgs e)
{
    string strText1 = textBox1.Text.Trim();
    string strText2 = textBox2.Text.Trim();
    string ret = ComTest.add("D61A457C-DBEF-43DE-80F4-394703BD3D41",
"Simulation Transaction",int.Parse(strText1), int.Parse(strText2));
    textBox3.Text = String.Concat(ret);
}
```

第八章 Windows消息与事件机制

委托(delegate) 概念

- 它是一个引用类型，内容是方法名称，规定了参数列表
- 参照 C\C++ 语言的函数指针
- 委托保证安全，避免越界与地址无效
- 委托的基类是 `System.Delegate`
- System.Delegate类是抽象类，不能直接实例化
- 系统和编译器可以显式地从 Delegate 类或 MulticastDelegate 类派生，用户是不允许由委托类进行派生新类的。

简述具有继承关系的异常类的捕获顺序是什么样的。

多个 catch 块的捕获顺序是从顶部到底部，对于所引发的每个异常，都只执行一个catch块，如果一系列的catch块所设定的异常类存在继承关系，会按照catch出现的顺序找到匹配的第1个类，并执行其相应的代码，不再执行后续可匹配的异常类。异常是按照最先匹配处理，而不是最佳匹配，如果将捕获基类异常代码写在前面，则基类的代码先被调用而后面的代码则被忽略。

所以编写 try 语句组时异常类捕获次序应按照类派生的逆序出现，即异常派生类语句写在异常基类之前。

简述 Windows 应用程序中的消息机制

Windows 具有一个系统消息队列按序存储全部消息，并根据规则将详细投放到进程的消息队列中。系统为每一个窗体对象创建一个消息队列，消息静分配后由系统队列进入到窗体队列，每一个窗体对象配置一个窗体线程运行消息循环任务。消息循环反复检查消息队列中的消息，根据消息值匹配执行相应的分支代码。

消息可以由系统自动派送，也可以由程序主动向其他程序发送，发送方式有两种，一种方式是将消息发送到先进先出消息队列结构中，这些消息也叫队列化消息，另一种方式是将消息直接发送到窗体函数中，这些消息叫非队列化消息。

驱动程序将用户的键盘与鼠标输入转化为消息结构放入系统消息队列，消息被分配到当前激活的窗体线程，窗体消息处理函数对消息进行匹配。而非队列和的信息则直接发送到了窗体过程。

上机作业

- WinForm实现两个窗体应用程序的消息传递

（1）在解决方案下面新建一个.netCore 类库项目：CopyDataStruct，生成的dll 在项目 `bin\debug\netcoreapp3.1`下

```
using System;
using System.Runtime.InteropServices;

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

（2）在同一个解决方案下添加一个Windows窗体应用项目 Receiver（不要选.net Framework 那个，选.net Core）在依赖项那里右键，选择项目引用，右下角浏览，选择上一个项目的dll。

```
using System;
using System.Windows.Forms;
using CopyDataStruct;

namespace Receiver
{
    public partial class Form1 : Form
    {
        const int WM_COPYDATA = 0x004A;
        public Form1()
        {
            InitializeComponent();
        }

        protected override void DefWndProc(ref Message m)
        {
            switch (m.Msg)
            {
                case WM_COPYDATA:
                    COPYDATASTRUCT cds = new COPYDATASTRUCT();
                    Type t = cds.GetType();
                    cds = (COPYDATASTRUCT)m.GetLParam(t);
                    string strResult =cds.lpData+"\r\n";
                    textBox1.Text += strResult;
                    break;
                default:
                    base.DefWndProc(ref m);
                    break;
            }
        }
    }
}

```

（3）同样的步骤建立一个 Sender项目：

```
using System;
using System.Windows.Forms;
using System.Runtime.InteropServices;
```

```
using CopyDataStruct;

namespace Sender
{
    public partial class Sender : Form
    {
        [DllImport("User32.dll", EntryPoint = "SendMessage")]
        private static extern int SendMessage(int hWnd, int Msg, int wParam,
ref COPYDATASTRUCT lParam);

        [DllImport("User32.dll", EntryPoint = "FindWindow")]
        private static extern int FindWindow(string lpClassName, string
lpWindowName);

        const int WM_COPYDATA = 0x004A;

        public Sender()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            int hWnd = FindWindow(null, @"Receiver");
            if (hWnd == 0)
            {
                MessageBox.Show("555，未找到消息接受者！");
            }
            else
            {
                byte[] sarr =
System.Text.Encoding.Default.GetBytes(textBox1.Text);
                int len = sarr.Length;
                COPYDATASTRUCT cds;
                cds.dwData = (IntPtr)9; // 可以是任意值
                cds.cbData = len + 1;   // 指定lpData内存区域的字节数
                cds.lpData = textBox1.Text;//发送给目标窗口所在进程的数据
                SendMessage(hWnd, WM_COPYDATA, 0, ref cds);
            }
        }
    }
}

```

参考：https://blog.csdn.net/yl2isoft/article/details/20222679

- WinForm窗体实现事件的定义、触发与处理
- WPF实现两个窗体应用程序的消息传递

WPF 与 WinForm类似，就是接受消息的流程有些不同：

```
using System;
using System.Runtime.InteropServices;
using System.Windows;
using System.Windows.Interop;
using CopyDataStruct;

```

```
namespace Receiver
{
    public partial class MainWindow : Window
    {
        public const int WM_COPYDATA = 0x004A;
        public MainWindow()
        {
            InitializeComponent();
        }
        // 不要忘了要给主窗口添加这个 load 事件
        private void MianWindow_loaded(object sender, RoutedEventArgs e)
        {
            HwndSource hWndSource;
            WindowInteropHelper wih = new WindowInteropHelper(this);
            hWndSource = HwndSource.FromHwnd(wih.Handle);
            hWndSource.AddHook(MainWindowProc);
        }
        private IntPtr MainWindowProc(IntPtr hwnd, int msg, IntPtr wParam,
IntPtr lParam, ref bool handled)
        {
            switch (msg)
            {
                case WM_COPYDATA:
                    COPYDATASTRUCT mystr = new COPYDATASTRUCT();
                    Type mytype = mystr.GetType();
                    COPYDATASTRUCT cds =
(COPYDATASTRUCT)Marshal.PtrToStructure(lParam, typeof(COPYDATASTRUCT));
                    listBox1.Items.Add(cds.lpData);
                    break;
                default:
                    break;
            }
            return hwnd;
        }
    }
}
```

- WPF窗体实现事件的定义、触发与处理

WinForm 窗体程序

【窗体线程与工作线程任务分工】

- 主窗体线程负责用户的输入与结果的显示，窗体以异步方式响应用户的输入，也可异步对结果显示，程序的响应性可用性较高。
- 工作线程负责消耗较多运算时间而又没有用户交互的任务。

应用程序在运行时根据执行逻辑，由主窗体线程创建工作线程，工作线程以后台方式运行不直接与用户发生交互，在创建时工作线程默认是没有消息队列循环机制的。

注意：

- 工作线程不允许使用窗体控件其属性和方法。
- 所有的窗体控件是属于窗体线程的，窗体线程负责接收用户输入，更新显示信息到窗体上。
- 实现数据的线程安全，避免访问冲突。
- 各种回调函数也可看作工作线程

【WinForm消息传递流程】

- 消息常量定义
- 发送线程查找窗体
- 线程发送消息给窗体
- 窗体接收消息
- 修改控件属性

事件的实现步骤

（1）自定义事件参数类，此类必须由 System.EventArgs 类派生

（2）用 delegate 关键字定义事件对象类型（含事件发起者以及事件参数）

（3）用 event 关键字定义事件对象

（4）定义事件发起者类

```
public class FireEventArgs : EventArgs
{
    public FireEventArgs(string room, int ferocity)
    {
        this.room = room;
        this.ferocity = ferocity;
    }
    public string room;
    public int ferocity;
}

public class Fire  {
    public delegate void FireEventHandler(object sender, FireEventArgs fe);
    public event FireEventHandler FireEvent;
    public void ActivateFireAlarm(string room, int ferocity){
       FireEventArgs fireArgs = new FireEventArgs(room, ferocity);
       FireEvent(this, fireArgs); // 事件触发函数
    }
}
```

（5）定义事件处理方法，也就是普通的类A的方法，但要求这个方法应当与 delegate 对象具有相同的参数和返回值类型。

（6）用 `+=` 操作符添加事件到事件队列中，`-=` 操作符能够将事件从队列中删除。

```
Fire fire;
void ExtinguishFire(object sender, FireEventArgs fe){};
fire.FireEvent += new Fire.FireEventHandler(ExtinguishFire);
```