# 委托

A类，B类两个类中各有一个方法

面向对象编程，有一个原则，有一个依次原则，

A类中创建了B的对象，对此B类就不能方向创建A的对象

当在B类中想要调用A类的方法，就需要使用到委托

对象的逆向调用就是委托：正常方法获取不到的时候，想到委托

//委托：五步法

//1、声明委托，必须定义在类的外面，命名空间下;就是声明方法的原型

//2、根据委托写出对应的方法

//3、创建委托变量

//4、关联委托方法（给委托赋值），多路委托（多播委托），按我关联的顺序运行，可以通过=，+=，-=分别是关联，关联，删除关联

//5、使用委托变量

# 实现窗体数据逆向传值

1. **首先在哪里调用方法，就在哪里创建对应的委托属性**
2. **可以讲委托所需要的参数方法，在创建窗体的时候传出去，也可以通过属性得到方法**
3. **最后就是五步走的委托使用**
4. **这里的难点就是，使B类的控件的Text传给A类，所以方法在A类中写，public void Receive(string s){this.textBox1.text=s},在B类中创建对应的委托属性，之后关联其方法，在B类中使用其属性。**

**可以创建**

**//获得一个委托，所有按顺序添加的多播委托方法**

**Delegate[] delegateList=delegate.GetInvocationList();**

**通过Foreach（delegate名 item in delegateList）遍历数组，**

**If(item.Method.Name==”多播委托中的方法名”)**

**{**

**然后强转**

**}**

**实现广播体现代码优质性：委托不仅解决了数据的逆向传递，还可以有效提升代码的效率**

1. **首先确定是需要将A类控件的Text属性传递给B类的控件**
2. **在B类中创建的方法，就是this.textBox.text=string s;对此在A类中创建委托属性，得到B类中的方法，之后调用**
3. **对于循环所创建的窗体，如果使用=关联B类中的方法，只会让最后一次循环关联方法，需要通过多播委托 +=的方法，让每一次循环的窗体都关联上方法**

# 事件:事件就是对委托的进一步包装

**观察1、this.btn.Click+=new System.EventHandler(this.btn\_Click);**

**观察2、public event EventHandler Click;**

**观察3、public delegate void EventHandler（object sender，EvenA rgs arg）**

**延申的思考：包装对实物起到保护作用**

**程序优秀：代码规范，技术的综合应用（简化，好的技术，扩展性），---不容易出错**

**不容易出问题：容错能力。**

**事件：把委托用不到的功能和用法，给去掉**

1. **在委托类型前面添加event就可以定义事件**
2. **事件必须使用+=或-=。不能直接=赋值**
3. **事件移除不能等于null，只能使用-=方式**

**自己创建事件的时候，对于委托还是不变的创建，只是说在类中创建委托变量的时候，可以将其加上event变为事件**

# 事件参与者

1. **发送者（sender）：也就是用来激发事件**

private void button1\_Click(object sender, EventArgs e)

{

//激发事件

this.EventReceive(this.textBox1.Text);

}

1. **接受者（Receiver）：事件的处理者，在事件触发后，自动执行的代码**

**//事件处理者**

public void ReceiveMsg(string s)

{

this.textBox1.Text+=s+"\r\n";

}

# 事件和委托的对比

1. 事件无法在外面赋值，比如 “对象.事件=null”,会出现编译错误，而委托可以。

好处是：避免用户对事件直接操作，比如Click=null，会把底层代码清空，事件可以起到保护作用。而委托相对太开放。

1. event对象没有invoke（）方法，invoke
2. 委托和事件的选择，

第一：正常解决问题，使用事件和委托没有区别，但是建议用委托。

第二：如果做控件二次开发，必须用事件event

# C#中匿名方法，Lambda表达式

委托的扩展用途：匿名表达式，Lambda表达式，和泛型结合，多线程使用

1. 匿名方法和Lambda表达式
2. 匿名方法：一个方法没有具体的名称，只有delegate关键字，方法参数和方法体；如果委托变量只需要关联一个方法，就可以使用匿名委托。
3. Lambda表达式：使用这种表达式可以更简练的编写代码块。【1】在Lambda参数可以是明确类型，可以是推断类型，根据设定的委托参数推断；【2】如果是推断类型，编译器根据上下文自动推断；【3】运算符=>读作goes to，运算符左边输入参数（如果有），右边是表达式或语句块；【4】表达式两种方式（int input）=>表达式 （input args）=>{语句1；语句2；……}【5】Lambda表达式和匿名方法比较，第一Lambda表达式本身就匿名方法，第二Lambda表示式允许不指明参数类型，但是匿名方法必须明确参数类型，第三Lambda表达式允许单一表达式，而匿名方法不允许单一表达式

public void Test(int t1,int t2)

{

//创建委托，关联方法

Add a1 = AddInt;

//使用匿名委托

Add a2 = delegate (int a, int b) { return t1+t2; };

//使用Lambda表达式

Add a3 = (int a, int b) => { return a+b; };

//当匿名的方法只有一句代码需要执行时

Add a4=(int a, int b) => a+b;

//当不想写参数的数据类型时,当然这里输入的数据，必须是创建委托时参数的类型一致

Add a5 = (a, b) => { return a+b; };

}

# 自定义泛型委托和匿名方法，Lambda表达式配合

1. 为什么要使用泛型委托

普通的委托在数据类型的限定上非常严格，有时候我们的需求式变化的，可能就适应不了

1. 泛型委托定义：本质上和泛型方法是非常相似，泛型委托相关联的时候，可以是具体方法，匿名方法，也可以是Lambda表达式

# 系统泛型委托

.Net平台早就考虑了如何去简化用户泛型委托的情况，也就是说.net平台定义好，我们可以直接使用

两种方式：

一中是没有返回值的Action<>,一种有返回值的Func<args>

1. Action<args>在多线程中用的非常多，args表示方法参数
2. Func<args>在扩展方法中用的比较多，其他地方也用，最后一个args参数就是返回值类型

# 多线程

时间线，6核，12线程

第一个任务：需要5分组

第二个任务：需要3分组

第三个任务：需要1分钟

……

时间片：比如一秒分为多个时间片

前5个时间片做任务1

后面3个时间片做任务2

再有5个时间片做任务3

但是之后又循环，切换的速度非常快

线程不是想开就开，不是能开多少开多少

以自己电脑所带的线程，上下5个左右

多线程的目的：其实就是希望”并行”执行多任务，提升效率

Thread线程就是一个委托，没有输入参数的委托，没有返回值，也可以添加参数是object的方法。使用必须先将其设定为后台线程。

Task：基于Thread+ThreadPool结合

# Task：三种启动方法

1. Task对象的Start方法启动，适合在符合条件情况下启动，也就是可以按照需求进行启动，灵活控制
2. Run方法，是定义后直接启动，适合马上启动的场合
3. Factory.StartNew方法启动，也是直接启动线程

# Task启动，暂停，继续，停止

暂停，继续，是需要创建一个ManualResetEvent手动重置事件的对象，在Task所构建的循环中添加，ManualResetEvent对象.WaitOne()；在暂停时.Reset(),在继续中是.Set()

停止，需要在Task，对象中关联一个取消Task的对象的token，首先要创建CanCelltionaTokensource对象，然后关联Task，所执行的委托，在while（.IsCanCellTionaRequested）,取消中是.Cancel();

Task跨线程访问控件和耗时任务的解决

主线程窗口的控件，是不能直接被子线程访问的

# **多线程中的“锁”**

多线程任务执行的时候，我们可以认为这些线程是并列执行的，如果多个线程，同时访问一个“资源”，如果没有执行完，其他线程进来同时使用这个资源，就会造成资源的“争夺”。程序会出现问题。可能得不到结果

所以，使用线程锁，在一个线程使用一个资源的时候，其他线程必须等待。也就是“排队”

# Var推断类型，扩展方法

平时定义变量，能用具体类型，就用具体类型

比如，使用别人的类库，调用某些对象的方法，这个方法返回什么类型，可以直接使用var

扩展方法：对方一个现有的类，直接增加相关的方法，尤其是这个类是别人写的，我们无权直接修改，扩展方法可以实现；

除此之外，平台也给我们提供了很多扩展方法，可以实现

扩展方法的类必须是static，是静态类

虚线类型向下的箭头就是扩展方法