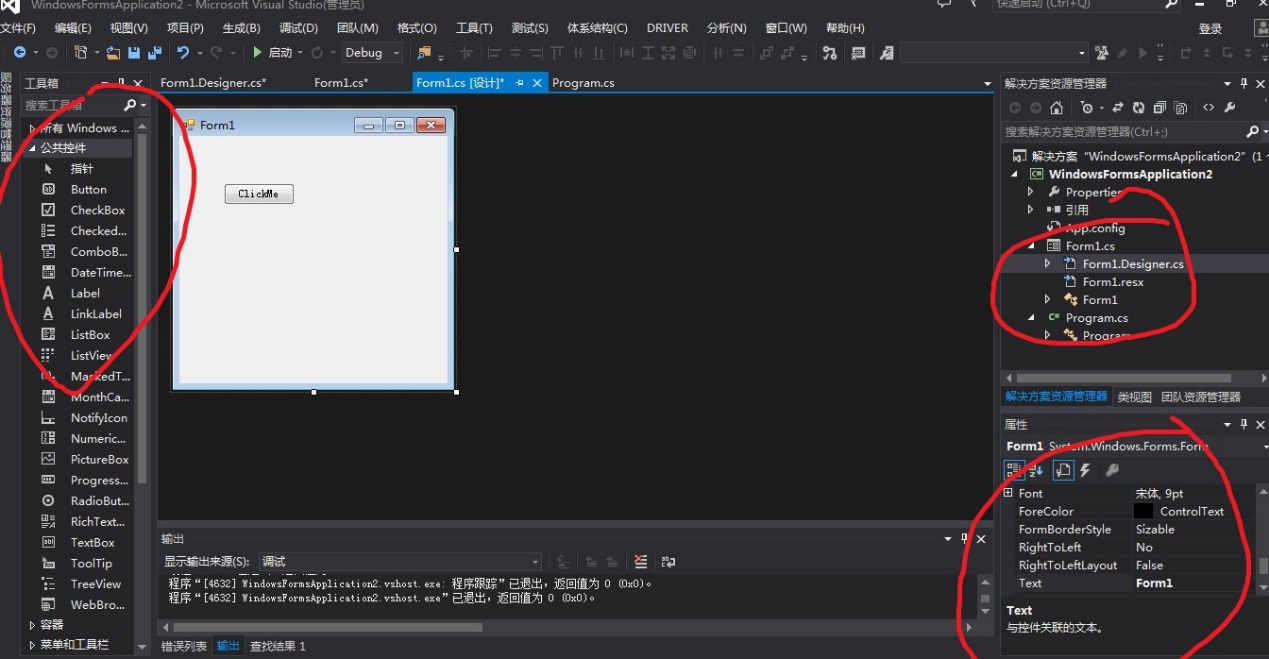
# 1.C#框架



控件栏（左键双击增加至窗体，右键对控件编程）

控件属性栏（类似VB修改控件属性）

控件属性栏程序代码（代码修改控件属性）

# 2.///与{0}

///单行注释自动续行

简化字符输出，使用{0},{1}…进行代替，如

(“The score of {0} is {1}.”,”My”,80); ===== (“The score of ”+”My”+” is ”+80+”.”);

# 3.ToDouble与ToString

String Score=”90”;

Double number=Convert.ToDouble(Score);

Double number=Convert.ToDouble(Console.ReadLine()); //类似于函数i=atof(str);

# 3. ^, &=

bool=a^b; //a,b中有且只有一个为true时，bool=true.

a1&=a2,当a1,a2均为true,a1为true,类似有 |= ^=

# 4.数组定义

C#使用指针？\*

C++: int p[] = { 1, 2, 3 }; int \*a = new int[5]； int \*a = new int[5]{1, 2, 3, 4, 5}; //可行

C#: int p[] = { 1, 2, 3 }; //不可行

C#: int[] p= { 1, 2, 3 }; //可行 int[] a = new int[5] { 1, 2, 3, 4, 5 };//可行

Int [] i=new int[4];//四个元素 数组的创建表达式 === int i[4];

# 5.foreach

foreach循环读显操作(不能改变元素值)，such as

foreach (int i in numbers) //numbers为一个数组变量

{

Console.WriteLine(i);

}

# 6.多维数组

<basetype>[,] <name>;//,表示数组纬度，此为两维。

如： double[,] house=new double[2,3];

或者 double[,] house={{1,2,3},{4,5,6}};

引言采用：house[1,1]等。

# 7.params

参数数组：C#允许一个函数有一个个数不定的参数调用函数，有仅有一个，且是函数的最后一个参数，用params关键字；

Static double func(int a, params double [] vals)

{

double sum=a;

foreach(double i in vals)

{

Sum+=I;

}

return sum;

}

调用：double sum=func(10,1,2,3); //结果为16

# 8.ref与out

函数值交换的方法：

1）ref 与 out 关键字

ref为引用，使值在函数调用后改变：参数必须为变量，且已初始化；

out类似，但可以不初始化；

2）定义变量为全局变量，其作用域将函数包括在内。

string,char,int等最好在定义后及时初始化，避免不必要的bug.

# 9.委托

委托的含义和使用

委托，可以看作一个类，但它只作用于函数（函数类），是一类参数形式的函数抽象集合，即把函数方法当做参数进行传递。

使用实例：

delegate double ProcessDelegate(double a,double b);//委托声明：声明一种函数形式，//其没有名称，只有参数形式

static double Mul(double a,double b) //委托函数1

{

return a \* b;

}

static double Div(double a, double b) //委托函数2

{

return a / b;

}

static void Main(string[] args)

{

ProcessDelegate process; //利用委托实例化一个函数名

double a = 8, b = 4;

Console.WriteLine("Input:");

string input = Console.ReadLine();

if (input == "M") //利用委托确定函数名process的操作(初始化)

{

process = new ProcessDelegate(Mul);

}

else

{

process = new ProcessDelegate(Div);

}

Console.WriteLine("Result:{0}.", process(a,b));

与函数重载的区别：

函数重载：相同的函数名称，不同的函数参数形式（参数变量类型和个数）和函数方法，调用时根据输入参数的形式自行匹配；

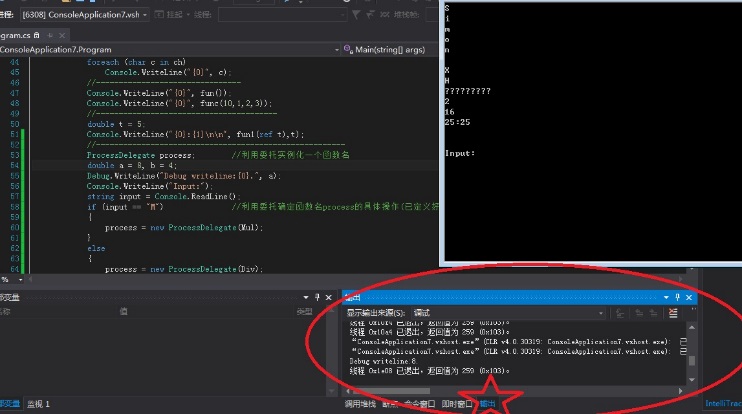
函数委托：不同的函数名称，相同的函数参数形式和不同的函数方法，调用时需要将函数方法委托给以定义好的函数，从而进行操作；过程：委托 声明->实例化->初始化。

# 10.程序调试

两种方法：

1)Debug.WriteLine(“message”); 需要头文件 using System.Diagnostics;

在调试窗口中即时输出信息（而不是程序中或控制台），调试窗口在



2)breakpoint+调试方法；

3)try catch throw;

# 11.enum枚举类型

在函数外部定义，类似于结构体，定义后各函数都可以用。

enum GNSS:int{GPS=1,BDS=2,GLONASS=3} //声明时定义,不赋值则默认为0,1,2,3…

GNSS myGNSS=GNSS.GPS; //初始化

Console.WriteLine("{0}:{1}.", myGNSS,(int)myGNSS); //输出为： GPS:1

主要作用：用于有限的名词集合赋予数字含义：如周，月，季节等等。给数字一个含义，方便使用。

# 12.decimal

decimal类型与double,float类型

decimal在sql server里是精确[值类型](http://www.so.com/s?q=%E5%80%BC%E7%B1%BB%E5%9E%8B&ie=utf-8&src=internal_wenda_recommend_textn" \t "_blank)，精度38；  
在C#里是表示 128 位数据类型。**与double相比，decimal 类型具有更高的精度和更小的范围，它适合于财务和货币计算。**  
decimal:  
有效位：±1.0 × 10(-28次方) 到 ±7.9 × 10(28次方)  
精度：28 到 29 位  
double:  
有效位：±5.0 × 10(-324次方) 到 ±1.7 × 10(308次方)  
精度：15 到 16 位

# 13.类virtual

基类 派生类

私有成员(private):只能本类使用；

公有成员(public)：大家都能用；(项目以外)

受保护成员(protected)：只有本类及其派生类可以使用；

继承

虚拟成员(virtual)：派生类可以对其所继承的虚拟成员重写，但不影响其原来的成员；虚拟成

员不得是私有的，即不存在virtual private。

# 14.abstract（抽象类）与interface（接口）

抽象类（abstract）与接口（interface）

1）抽象类和接口的意思，就是规范，大家都遵守。这样符合规范的东西，大家都不用了解各自的细节。插座不会了解插销怎么做的。插销也不必了解插座怎么做的。

可以把方块类写成抽象类，而它的抽象方法就是翻转方法。当然，你也可以把方块类写为非抽象的，也可以在方块类的翻转方法中写上几千行的代码。但这样好吗？

抽象类将事物的共性的东西提取出来，抽象成一个高层的类。子类由其继承时，也拥有了这个超类的属性和方法，以实现代码的复用。子类中也可加上自己所特有的属性和方法，也就实现了多态。

2）接口（光杆的类）：接口中包含方法、属性、索引器和事件的声明，但常用的接口中一般就是方法和属性，然而接口中并没有方法的具体实现代码（**不能提供任何成员实现**），**只有方法的返回类型和方法名**。一个类实现了某个接口，则必须在类中包含接口中所有方法的实现代码，换句话说就是，继承自接口的类中必须包含接口中函数的实现代码。

作用：具有约束作用，可以限定类必须实现某些功能；其次，接口减少了代码量，便于扩展，比如银行 账户等级越高，所具有的功能就越多（实现更多的接口）；再次，规范多个开发人员的代码，比如屌丝账户类由程序猿A来写，金卡账户类由程序猿B来写，那么我告诉AB这两个账户都要实现接口1，则A、B写的类中相同函数的函数名称就一致了(但实现可以不一致！)。

继承只适合归纳有相同实现（代码）的对象。比如：父类 羊.进食(){吃草;}、子类 山羊、子类 绵羊。

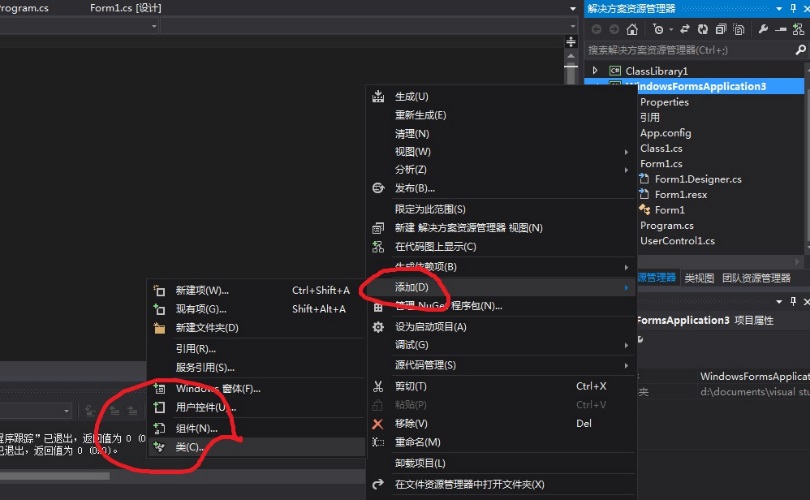
接口归纳范围更大，只要表现（接口）一致、可以实现不同。比如：接口 宠物.进食(){}、类 猫.进食(){吃鱼;}、类 狗.进食(){吃肉;}

# 15.sealed等类域关键字

|  |  |
| --- | --- |
| 修饰符 | 含义 |
| 无或internal （内部） | 只能在当前项目中访问类 |
| public （公共） | 可以在任何地方访问类 |
| abstract或internal abstract （内部抽象） | 只能在当前项目中访问类，不能实例化，只能供继承之用。 |
| public abstract  （公共抽象） | 可以在任何地方访问类，不能实例化，只能供继承之用。 |
| sealed或internal sealed （内部密封） | 只能在当前项目中访问类，不能供派生之用，只能实例化。 |
| public sealed  （公共密封） | 可以在任何地方访问类，不能供派生之用，只能实例化。 |

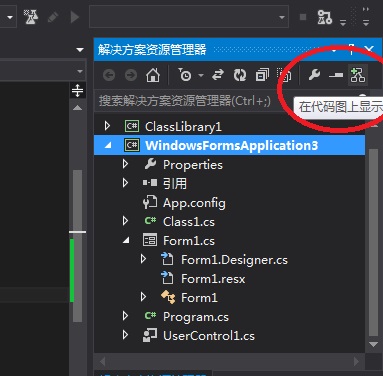
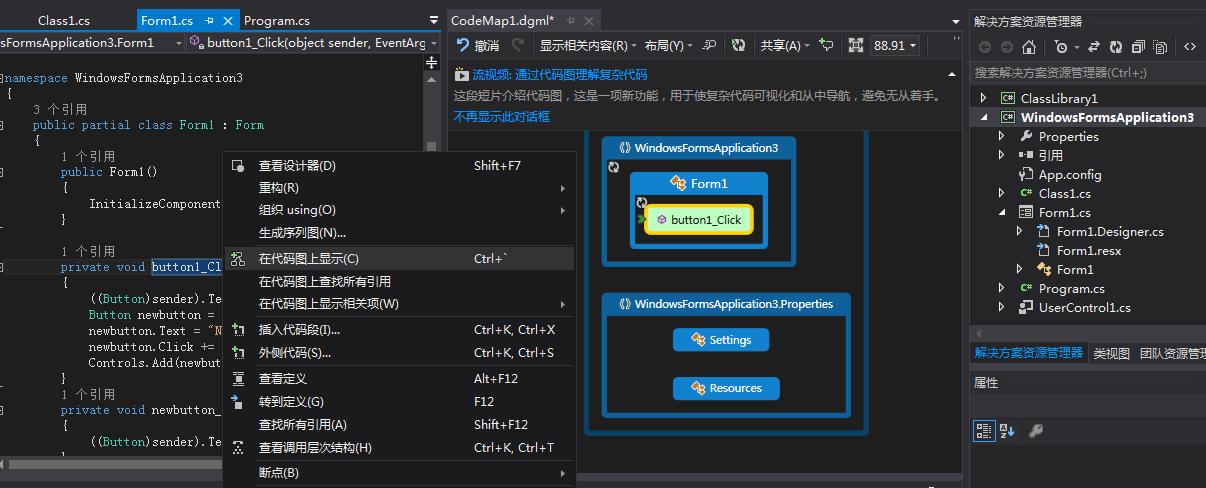
# 16.快速增加类或者类库项目

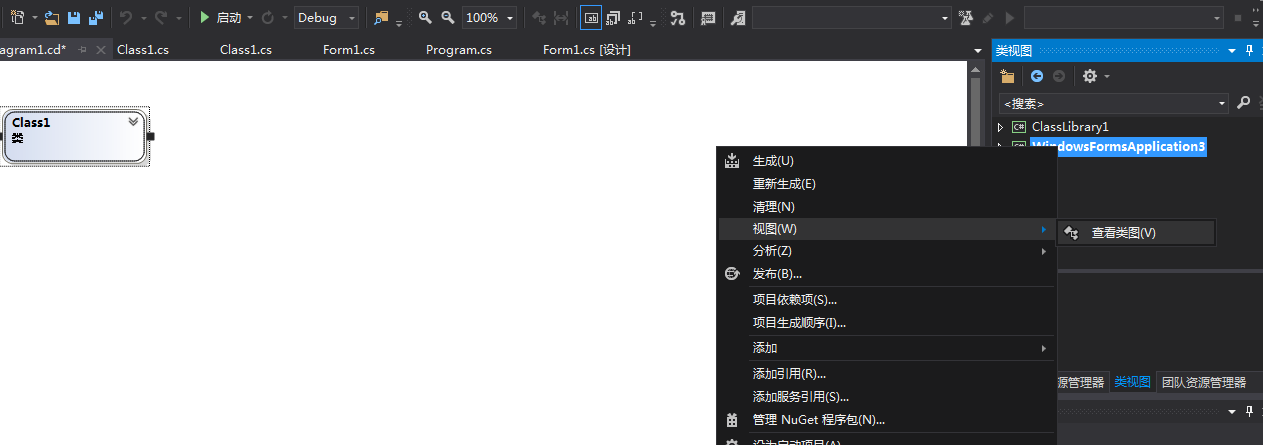
类库项目：一个只包含类的项目，便于项目移植。



# 17.代码图与类图

利用代码图和类图迅速了解、学习未知的程序，函数的包含调用关系，类的继承派生关系：





# 18. public override string ToString()

重写了object类的ToString()方法。  
在C#中所有的类都继承自object。object提供的方法在任何一个类中都可以使用，如果你不希望使用object的方法的话，你可以重写他。

public override string ToString()

{

return String.Format("My expression {0}", value);

}

重写之后，当你使用A.ToString()时或者Console.WriteLine(thsi)的结果为“Hello”，如果你没有重写ToString，默认返回的是类的[全名](http://www.so.com/s?q=%E5%85%A8%E5%90%8D&ie=utf-8&src=internal_wenda_recommend_textn" \t "_blank)比如SampleNameSpaces.A.其中A为类名，SampleNameSpaces为命名空间。

# 19. get与set

get与set：类成员参数(属性)的访问与设置。

定义一个类成员 private decimal balance;

不同于C++成员函数访问和设置，C#使用get和set更简单的使用。

public decimal Balance

{

get

{

return balance;

}

set

{

balance=Isetnum;

}

}

外部使用：classname.balance //get

classname.balance=100; //set

# 20. override（覆盖/重写）

override（覆盖）关键字:用于该类对其继承的基类方法进行修改，满足自己的功能。

基类 BaseClass成员函数（方法）： baseFun()

派生类MyBaseClass:BaseClass，其需要使用baseFun()函数，但需要进行修改，因此可以override（覆盖）它

override baseFun()

{

//…

}

# 21. ”:” 表示实现接口或者基础父类

class A : BaseClass  
那么就是A继承了BaseClass,拥有BaseClass的属性和方法  
class A : InterfaceA  
那么代表A需要实现InterfaceA的方法

# 22.base和this

this：指当前类，this调用当前类的属性，方法，包括构造函数的方法，继承本类的构造函数。

base：指当前类的父类，可调用父类的非私有属性，方法，继承父类的构造函数括号里的参数。

不能在静态方法中引用this和base，因为当类第一次被加载的时候，静态成员已经被加载到静态存储区，此时类的对象还有可能能没有创建，而this和base访问的都是类的实例，也就是对象，而静态成员只能由类访问，不能由对象访问，所以静态方法中不能调用类成员字段。

# 23.隐藏和重写

1,**重写（也叫覆盖：override）**,必然发生在基类和派生类中,其类函数用virtual修饰,派生类用override修饰

2,**隐藏**,在子类中写一个和基类一样名字(参数不同也算)的非虚函数,会让基类中的函数被隐藏,这时候一般编译时会报一个警,子类中的函数用**new**修饰一下就不报警了

3,**重载**,必然发生在一个类中,函数名相同,参数类型或者顺序不同构成重载,与返回类型无关。

override 用了就只有一个方法**this.Fun();**重新new一个同名方法 还是有两个方法可以用；一个是**Base.Fun();**一个是**this.Fun();**

# 24.get/set自动属性

public string str { get; set; }   
相当于以下代码  
**private** string \_str;  
public string str  
{  
get { return \_str; }  
set { \_str = value }  
}

对比一下常规属性和自动属性之间的区别吧：

1.自动实现的属性必须同时声明 get和 set 访问器。创建 readonly自动实现属性时，需要将set访问器设置为private 。

2自动实现的属性上可以使用特性，不能用在支持后备字段上。如果属性的后备字段上使用特性，则应该只创建常规属性。

3.自动实现属性get,和set中**不能包含特殊的逻辑处理**。与字段类似，但不同于字段。与字段不同，属性不作为变量来分类，不能将属性作为 ref参数或 out参数传递。

Partial是局部类型的意思。允许我们将一个类、结构或接口分成几个部分，分别实现在几个不同的.cs文件中。C#编译器在编译的时候仍会将各个部分的局部类型合并成一个完整的类

# 24.partial关键字

2.适用范围：

（1）类型特别大，不适合放在一个文件中实现。

（2）一个类型中的一部分代码为自动化工具生成的代码，不宜与我们自己编写的代码混合在一起。（如设计模式中的“模板方法”。模板方法就是把所有相同功能都写在父类中，需要有变化的方法在父类中写成虚方法，然后在子类中重写虚方法，因此模板方法利用partial实现了代码的利用。）

（3）一个类同时需要多个人编写。

3.注意事项：

（1）只适用于类、接口、结构，不支持委托和枚举。

（2）同一个类型的各个部分必须有修饰符partial。

（3）使用局部类型时，一个类型的各个部分必须位于相同的命名空间中。

（4）一个类型的各个部分必须同时被编译。

（5）**partial只有和class、struct、interface放在一起时才表示局部类型**。

（6）一个类型的各部分子类的访问修饰符必须保证一致性。

（7）若其中一个部分类使用了static/abstract/sealed修饰符，则整个类都是静态类/抽象类/密封类。（后面会有static/abstract/sealed的详解）。另外注意不能在一个部分类上使用static，在另一个部分类上使用abstract。

（8）局部类型的累加效应。

4. 局部类型的应用特性

在局部类型上的特性具有“累加”效应。

[Attribute1, Attribute2("Hello")]

partial class Class1{}

[Attribute3, Attribute2("Exit")]

partial class Class1{}

相当于

[Attribute1, Attribute2("Hello"), Attribute3, Attribute2("Exit")]

class Class1 {}

注：Attribute2属性允许在类上多次使用。

5. 局部类型上的修饰符

(1) 一个类型的各个部分上的访问修饰符必须维持一致性。

(2) 如果一个类型有一个部分使用了abstract修饰符，那么整个类都将被视为抽象类。

(3) 如果一个类型有一个部分使用了 sealed 修饰符，那么整个类都将被视为密封类。

(4) 一个类的各个部分不能使用相互矛盾的修饰符，比如不能在一个部分上使用abstract，又在另一个部分上使用sealed。

6. 局部类型的基类和接口

(1) 一个类型的各个部分上指定的基类必须一致。某个部分可以不指定基类，但如果指定，则必须相同。

(2) 局部类型上的接口具有“累加”效应。

partial class Class2: Iinterface1, Iinterface2 {}

partial class Class2: Iinterface3 {}

partial class Class2: Iinterface2 {}

相当于

class Class2: Iinterface1, Iinterface2, Iinterface3 {}

# 25.static关键字

static可修饰类、字段、属性、方法。

static在使用的时候分三种方法。

1.修饰类的时候：

将类修饰为静态的，意味着这个类不能实例化对象，在调用类里面的任何内容的时候直接用类名加点的形式调用其中的方法或者字段(类名.方法/类名.字段)

2.修饰方法或者属性的时候：

将方法修饰为静态方法，表示此方法为所在的类所有，而不是这个类实例化的对象所拥有，这个方法在调用的时候需要类名加点（类名.方法）来调用。

3.修饰变量：

表示在每次调用该变量的方法或者类的时候，变量的值为最后一次赋值时的值，而不是再次初始化它的值。

# 26.三元运算符

int i = 10;

int j = i == 10 ? 1 : 2;

//转换成if选择结果如下

if (i == 10)

{

j = 1;

}

else

{

j = 2;

}

# 27.ArrayList & List

比较数组、ArrayList和**List （等价于C++中的Vector）**

* 数组：优点，元素类型一样，简单易懂；缺点，维数限定，不能动态变化，删除、插入等不方便；
* **ArrayList:**优点，维度随意增减，动态变化，元素插入、删除方便；缺点：**元素类型不一致，**需要装箱拆箱，使用时容易报错；
* List:在ArrayList的基础上限定了元素类型，即一般使用中的动态数组，类似于C++中的Vector。

class Program

{

struct MyStruct

{

double score;

string name;

string[] message;

char [] ch;

}

static void Main(string[] args)

{

//新建一个列表数组，三种初始化方法

ArrayList newlist = new ArrayList(); //1.空的arraylist,ArrayList()

ArrayList arr2 = new ArrayList(5); //2.创建容量为5,ArrayList(int capacity)

int[] ary = { 0, 1, 2, 3, 4 };

ArrayList arr3 = new ArrayList(ary); //3.指定5个元素 ,ArrayList(ICollection c)

for (int i = 0; i < 10; i++) //与方法3类似

{

newlist.Add(i);

}

foreach (int num in newlist ) //遍历列表中的元素值，注意初始化时该表均为int，因此此处不会报错

{

Console.WriteLine("{0} ", num);

}

newlist.Add("AddElements"); //增加列表元素

newlist[1]="ChangeElements"; //修改列表元素

Console.WriteLine("{0} ", newlist[10]);

Console.WriteLine("{0} ", newlist.Count); //返回列表元素的总数目

newlist.RemoveAt(0); //删除指定索引的元素

Console.WriteLine("{0} ", newlist.Count);

if (newlist.Contains("1")) //查询指定的元素是否在列表中，是返回true

Console.WriteLine("字符1在newlist中.");

else

Console.WriteLine("字符1不在newlist中.");

newlist.Insert(0, "hhh"); //插入，在指定索引处插入元素

//foreach (string str in newlist) //报错！因为newlist有些元素时int，有些是string，不能输出

//{

// Console.WriteLine("{0} ", str);

//}

newlist.Clear(); //清空列表

Console.WriteLine("{0} ", newlist.Count);

//------类似于Vertoc<string>list;

List<string> list = new List<string>(); //操作与ArrayList一种，但定义时确定了其类型，更加安全可靠

list.Add("uselist");

Console.WriteLine("{0} ", list[0]);

//------类似于CArray<string,string&>list;

List<MyStruct> mystructlist = new List<MyStruct>(); //结构体类型的动态数组

Console.ReadKey();

}

}

# 28.CompareTo()和Compare()

都用于字符串比较（逐个），用法a.ComparedTo(b), String.Compare(a,b);返回值为int -1,0,1

**public int CompareTo (string strB)**

返回值

* 小于 0，实例小于参数 strB；
* 0，实例等于参数 strB；
* 大于 0，实例大于参数 strB，或者 strB 是 null 引用。

比较程序

并不是按字符串长度来比较的，而是：

* 比较两个字符串的第一个字符：相等则比较第二个，不相等则直接返回结果；
* 比较两个字符串的第二个字符：相等则比较第三个，不相等则直接返回结果；
* 比较两个字符串的第三个字符：相等则比较第四个，不相等则直接返回结果；

String str1 = "asl";

string str2 = "bsl";

int a = str2.CompareTo(str1); //str2与compareTo(参数str)比较。2大于1，返回值为1。

int b = String.Compare(str2, str1); //Compare只用于字符串比较

Console.WriteLine(b.ToString());

Console.WriteLine(a.ToString());

if(a<0)

{

Console.WriteLine("true");

}

else

{

Console.WriteLine("wrong");

}

# 29.is与as

**Is:用于判读实体obj是否属于某种类型：**

int ast1 = 123;

if (ast1 is int) //同样的，可用于判断 myClass is Class

{ Console.WriteLine("is int"); }

**As: 一种基本的强制类型转换方法**

**传统的类型转化方法：**

Object obj1 = new NewType();  
NewType newValue = (NewType)obj1;  
**这样强制转换的时候，这个过程是不安全的**，因此需要用try-catch语句进行保护；

C#中增加as进行类型转换。**//增加安全性**

**Object obj1 = new NewType（）；  
NewTYpe newValue = obj1 as NewType；**   
as操作符不会做过的转换操作，当需要转化对象的类型属于转换目标类型或者转换目标类型的派生类型时，那么此转换操作才能成功，而且并不产生新的对象【当不成功的时候，会返回null】。因此用as进行类型转换是安全的。

**注意as的错误用法**

1、**不用在同类型之间进行类型转化**，即如下编写就会出现编译错误。  
NewType newValue = new NewType（）；  
NewTYpe1 newValue = newValue as NewTYpe1；  
2、**不能应用在值类型数据，即不能如下写**（也会出现编译错误）  
Object obj1 = 11；  
int nValue = obj1 as int;

# 30.泛型<T>

**类似于C++中的template**

在C#中，不存在C++的类型模板template, 但有时同样功能的函数需要处理不同的数据（如swap()函数），此时解决的方案大致有三种：

1. 利用**函数重载**：把每个数据类型的函数都写好，笨拙且麻烦，可移植性差；
2. 利用所有数据类型的基类：**object**, 把数据类型改为object, 即以前

double fun(); int fun(); long fun();写的函数，用一个object fun()代替。

问题：解决了代码的可读性，但是这样又有个不好的地方了，我们这样做实际上是一个**装箱拆箱操作，会损耗性能**。

1. 泛型。

示例：

public class GenericClass

{

**public void ShowT<T>(T t)**

**{**

**Console.WriteLine("ShowT print {0},ShowT Parament Type Is {1}", t, t.GetType());**

**}**

}

static void Main(string[] args)

{

Console.WriteLine("\*泛型方法调用\*");

generice.ShowT<int>(11);

generice.ShowT<DateTime>(DateTime.Now);

generice.ShowT<People>(new People { Id = 11, Name = "Tom" });

Console.ReadKey();

}

**总结：**

**C#泛型:public void ShowT<T>(T t) {//content} 等于下面**

**C++模板：**

**template<T>**

**public void ShowT (T t) {//content}**

# 31.事件

参考博文：<http://www.tracefact.net/tech/009.html>

1）事件和委托总是在一块使用的，其实事件包含了一个私有委托，事件类似于对一个委托类型的变量进行封装。

public delegate void GreetingDelegate(string name);

//**声明一个委托GreetingDelegate**

public GreetingDelegate Exmaple\_GreetingDelegate;

//**实例化委托变量Exmaple\_GreetingDelegate**

public event GreetingDelegate Exmaple\_GreetingDelegate;

//**插入event关键字将实例化委托变量Exmaple\_GreetingDelegate进行封装**

//在类的内部，不管你声明它是public还是protected，它总是private的。在类的外部，//注册“+=”和注销“-=”的访问限定符与你在声明事件时使用的访问符相同。

2）复习委托：委托是一个方法（函数）的抽象，假如说话的方法可以有说中文和说英文，即

Public void SpeakChinese(string sentences)

{

//说中文语句;

}

Public void SpeakEnglish(string sentences)

{

//说英文语句;

}

那么用户在调用类似的方法时，需要首先进行判断，这样就影响了程序的扩展性。因此，我们直接声明一个说话的抽象，即说话函数的模板：

Public **delegate** void **Speaklanuge**(string sentences);

使用时采用

**Speaklanuge mySpeaklanuge=new Speaklanuge();**

**mySpeaklanuge=** **SpeakEnglish;**

**mySpeaklanuge(sentence);**

**Speaklanuge 是一个方法的抽象，即**委托是一个类，它定义了方法的类型，使得可以将方法当作另一个方法的参数来进行传递，这种将方法动态地赋给参数的做法，可以避免在程序中大量使用If-Else(Switch)语句，同时使得程序具有更好的可扩展性。

# 32. Observer设计模式

Observer设计模式中主要包括如下两类对象：

1. **Subject**：监视对象，它往往包含着其他对象所感兴趣的内容。在本范例中，热水器就是一个监视对象，它包含的其他对象所感兴趣的内容，就是temprature字段，当这个字段的值快到100时，会不断把数据发给监视它的对象。
2. **Observer：**监视者，它监视Subject，当Subject中的某件事发生的时候，会告知Observer，而Observer则会采取相应的行动。在本范例中，Observer有警报器和显示器，它们采取的行动分别是发出警报和显示水温。

**Subject的类特征：有关键的属性，并能够发出消息，有委托和事件的声明；**

**Observer的类特征：对Subject感兴趣的类及其方法。**

**如何在Subject和Observer建立联系：在Observer中的方法与Subject的事件之间建立注册关系！**

在本例中，事情发生的顺序应该是这样的：

1. 警报器和显示器告诉热水器，它对它的温度比较感兴趣(注册)。
2. 热水器知道后保留对警报器和显示器的引用。
3. 热水器进行烧水这一动作，当水温超过95度时，通过对警报器和显示器的引用，自动调用警报器的MakeAlert()方法、显示器的ShowMsg()方法。

类似这样的例子是很多的，GOF对它进行了抽象，称为Observer设计模式：**Observer设计模式是为了定义对象间的一种一对多的依赖关系，以便于当一个对象的状态改变时，其他依赖于它的对象会被自动告知并更新。Observer模式是一种松耦合的设计模式。**

# 33. .Net Framework中的委托与事件规范

.Net Framework的编码规范：

* 委托类型的名称都应该以**EventHandler**结束，如BoiledEventHandler
* 委托的原型定义：有**一个void返回值**，并接受**两个输入参数**：一个**Object** 类型，一个 **EventArgs**类型(或继承自EventArgs)。

如public delegate **void** BoiledEventHandler(**Object sender**, **BoiledEventArgs e**);

**BoiledEventArgs是一个继承**自EventArgs的事件。

* 事件的命名为委托去掉EventHandler之后剩余的部分。如**Boiled**；
* 继承自EventArgs的类型应该以EventArgs结尾。BoiledEventArgs

再做一下说明：委托声明原型中的Object类型的参数代表了Subject，也就是监视对象，在本例中是 Heater(热水器)。回调函数(比如Alarm的MakeAlert)可以通过它访问触发事件的对象(Heater)。EventArgs 对象包含了Observer所感兴趣的数据。

总结：

***1）Subject为一个类，其包含委托和事件的声明、传递的消息、事件响应的方法和事件本身***；

public class Heater

{

private int temperature;

public string type = "RealFire 001"; // 添加型号作为演示

public string area = "China Xian"; // 添加产地作为演示

//声明委托

public **delegate void BoiledEventHandler**(Object sender, BoiledEventArgs e);

public **event** BoiledEventHandler **Boiled**; //声明事件

public class **BoiledEventArgs : EventArgs**

{

public readonly int temperature;

public BoiledEventArgs(int temperature)

{

this.temperature = temperature;

}

}

protected virtual void **OnBoiled(BoiledEventArgs e)**

{

if (Boiled != null)

{ // 如果有对象注册

**Boiled(this, e)**; // 调用所有注册对象的方法

}

}

public void **BoilWater()** {

for (int i = 0; i <= 100; i++) {

temperature = i;

if (temperature > 95) {

//建立BoiledEventArgs 对象。

BoiledEventArgs e = new BoiledEventArgs(temperature);

**//消息传递**

OnBoiled(e); **// 事件响应的方法**

}

}

}

***2）Object为一个类，可以有多个Object类，联系监测的对象，接受引发事件的消息，并定义事件发生的函数（响应函数）；***

public class Alarm

{

public void MakeAlert(Object sender, Heater.BoiledEventArgs e)

{

Heater heater = (Heater)sender; //这里是不是很熟悉呢？

//访问 sender 中的公共字段

Console.WriteLine("Alarm：{0} - {1}: ", heater.area, heater.type);

Console.WriteLine("Alarm: 嘀嘀嘀，水已经 {0} 度了：", e.temperature);

Console.WriteLine();

}

}

// 显示器

public class Display

{

public static void ShowMsg(Object sender, Heater.BoiledEventArgs e)

{ //静态方法

Heater heater = (Heater)sender;

Console.WriteLine("Display：{0} - {1}: ", heater.area, heater.type);

Console.WriteLine("Display：水快烧开了，当前温度：{0}度。", e.temperature);

Console.WriteLine();

}

}

# 34.对象初始化器

**方法1：自动属性**

//类中定义：

public int scores { get; set; }

public string name { get; set; }

//类外实例化和初始化：

MyClass myclass = new MyClass();

myclass.scores = 98;

myclass.name = "Luo";

**方法2：修改构造函数**

//也可以在类中重写构造函数：

public int scores;

public string name;

public Myclass(int s,string n) { scores=s;name=n;

}

//类外实例化和初始化：

MyClass myclass = new MyClass(98，“Luo”);

如果这样的话，那么MyClass myclass1 = new MyClass();会报错！

**方法3：对象初始化器**

//类中定义：

public int scores;

public string name;

//类外实例化和初始化：

MyClass myclass = new MyClass {name="Luo",scores=90 };

MyClass myclass1 = new MyClass(); //可以这样

# 35.var关键字

用于循环里即可，感觉没啥用！

或者通俗的讲：

**var可以理解为匿名类型**，我们可以认为它是一个声明变量的占位符。它主要用于在声明变量时，无法确定数据类型时使用。

使用var定义变量时有以下四个特点:

1. **必须在定义时初始化**。也就是必须是var s = “abcd”形式，而不能是如下形式: var s; s = “abcd”;

2. 一但初始化完成，就**不能再给变量赋与初始化**值类型不同的值了。

3. v**ar要求是局部变量**。

4. 使用var定义变量和object不同，它在效率上和使用强类型方式定义变量完全一样。

# [36.@字符](mailto:36.@字符)

主要用于文件流操作，在文件路径的字符串中，有许多“\”字符，为了避免其被认作转义字符，可以在路径（字符串）前面加上@,表示这个字符串应逐个地解释。

string [dir=@”C:\log.txt](mailto:dir=@”C:\log.txt)”; == string dir=”C:\\log.txt”;

# 37.文件操作

示例代码如下:

using System;

using System.IO;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace chap21File

{

class Program

{

static void Main(string[] args)

{

string dir1 = @"C:\Users\Administrator\Desktop\C\_ExamplesofFile\test3.txt";

string dir2 = @"C:\Users\Administrator\Desktop\C\_ExamplesofFile";

//----或者------------------------------------------------------

//string dir2="C:\\Users\\Administrator\\Desktop\\C\_ExamplesofFile";

Console.WriteLine("dir1={0}\ndir2={1}.", dir1,dir2);

//-------1.文件名和目录遍历----------------------------------------

//获得目录下所有目录:即该文件夹内所有文件夹和子文件夹的路径

string[] dirs = Directory.GetDirectories(dir2, "\*", SearchOption.AllDirectories);

foreach (string i in dirs)

{

Console.WriteLine(i);

}

Console.WriteLine();

//----------------------------------------------------------------------------------------

//获得目录下所有文件的路径和文件名

DirectoryInfo folder = new DirectoryInfo(dir2);

foreach (FileInfo file in folder.GetFiles("\*.txt", SearchOption.AllDirectories)) //"\*"：任意文件类型，"\*.txt":txt文件类型

{ //SearchOption.AllDirectories:遍历子目录，缺省则只在当前目录下寻找

if (!file.Exists)

{

Console.WriteLine("当前文件不存在！");

}

Console.WriteLine(file.FullName); //带路径

Console.WriteLine(file.Name); //不带路径，只为文件名

}

//----------------------------------------------------------------------------------------

//-------2.根据已有的文件名和目录进行文件读写操作----------------------------------------

//-----------示例：将test1.txt的内容写到一个新文件new\_test1.txt中-------------------------

FileStream aFile = new FileStream(dir1, FileMode.Open);

StreamReader sr = new StreamReader(aFile); //读

//StreamReader sr = new StreamReader(dir1); //这一行可代表上面两行的效果

StreamWriter sw = new StreamWriter("C:\\Users\\Administrator\\Desktop\\C\_ExamplesofFile\\"+"new\_test.txt");//写

//---2.1 一行一行地读，作为string保存-------------

string StrLine = "";

StrLine = sr.ReadLine();

while (StrLine != null)

{

Console.WriteLine(StrLine);

sw.WriteLine(StrLine);

StrLine = sr.ReadLine();

}

sr.Close();

sw.Close();

//---2.2 一个字符一个字符地读，作为Char保存------

StreamReader sr1 = new StreamReader(dir1);

int nChar = 0;

nChar=sr1.Read();

while (nChar != -1)

{

Console.WriteLine(Convert.ToChar(nChar));

nChar = sr1.Read();

}

sr1.Close();

//---2.3 一行一行地读，作为string保存------可用于大型文件读取，效果较好

foreach (string str in File.ReadLines(dir1))

{

Console.WriteLine(str);

}

//------------------------------------------------------------------------------------------

Console.WriteLine("------处理分隔符-------");

string[] stringArray;

char[] charArray = new char[] { ','}; //可以定义多个分隔符，例如char[] charArray = new char[] { ',','.', };

foreach (string str in File.ReadLines(dir1))

{

stringArray = str.Split(charArray);

foreach (string i in stringArray)

Console.WriteLine(i.ToString());

//Console.WriteLine(str);

}

Console.ReadKey();

}

}

}