

数据类型：

整数型数据类型：

//数据类型作用:不同的数据，使用不同的数据类型来存放，节省空间，便于管理

//整数类型从小到大

//第一种：字节类型 byte 最小的数据类型

//第二种 sbyte

//1TB=1024GB

//1GB=1024MB

//1MB=1024KB

//1KB=1024BYTE

//1Byte=8bit 8位 0000 0000 byte byteValue = 10; sbyte sbyteValue = -128;

//.GetType():该函数用于检测当前的数据类型 Console.WriteLine(byteValue.GetType());

//System.byte 系统字节类型 Console.WriteLine(byteValue); Console.WriteLine(sbyteValue);

//计算byte类型能够存下的最大的整数值

//0000 0000 最小值

//1111 1111 = 1+2+4+8+16+32+64+128=255 最大值

//byte 这个数据类型的存值范围为 0--255 的整数值

//sbyte 这个数据类型的存值范围为:-128--127 可以存负数

//byte 用于消息队列

//第三种：短整型类型 short 占两个字节 16位

// 第四种：unshort short shortValue= 5000;

//sizeof（）单目运算符 可以得到一个数据类型所占的字节数的大小 Console.WriteLine(sizeof(short)); Console.WriteLine(shortValue);

//u代表无符号的意思，所以ushort 只能存整数 ushort ushortValue = 1; //第五种 int整数类型 占四个字节 32位 一般开发中多使用int

//第六种 uint 无符号整数类型 只能存正数 int intValue = 12; Console.WriteLine(sizeof(int)); uint uintValue = 1;

//第七种 long长整数类型 占八个字节 64位

//第八种 ulong 无符号长整数类型 只能存正数

//LONG LONG C语言的写法 long longValue = 12; Console.WriteLine(sizeof(long)); ulong ulongValue = 1;

**数值类型：浮点数**

**float 单精度浮点数**

**double 双精度浮点数**

**decimal 精准浮点数**

//浮点数 （C# 默认小数类型是double）

//float 单精度浮点数 占四个字节

//在数值后面跟上f ，表示当前小数位单精度小数，不可以不加

float floatVakue = 1.333333333f;

Console.WriteLine(floatVakue.GetType());//System.Single数据类型

Console.WriteLine(sizeof(float));

Console.WriteLine(floatVakue); //只保留小数点后六位 四舍五入

//double 双精度浮点数 占八个字节

//在数值后面跟上一个d 或者不跟也可以

double doubleValue = 1.3333333333333333d;

Console.WriteLine(doubleValue);

Console.WriteLine(sizeof(double));

//decimal 十进制的 精准浮点数 占十六个字节

//在数值后面跟上m,表示精准浮点数 ，不可以不加

decimal decimalValue = 1.26356564545841562356m;

Console.WriteLine(decimalValue);

Console.WriteLine(sizeof(decimal));

Console.WriteLine(decimalValue.GetType()); //System.Decimal数据类型

**字符类型**

//字符类型 char 占两个字节 //通过单引号表示的一个字符 //A:65 a:97 //只能用单引号 char charValue = 'a'; Console.WriteLine(sizeof(char)); Console.WriteLine((int)charValue);

**布尔类型**

//布尔类型 boo1 占一个字节 //C# 的布尔值只存在 true 和 false 不能够用1和0来代替. //只是可以强行转换为0和1（不建议） bool boolValue = true; Console.WriteLine(sizeof(bool)); Console.WriteLine(Convert.ToInt32(boolValue));

**数据类型转换**

**显式转换：通过强制转换的方式对数据类型进行转换**

**（ 一般情况下，都是从存储范围小的转换为存储范围大的）**

**隐式转换：不是显示转换的转换**

**第一种转换方式：**

**通过（数据类型）的方式，对某一个数据类型进行强制转换**

**在使用（int）做转换的时候，统一截断小数部分**

//第一种转换方式：通过（数据类型）的方式，对某一个数据类型进行强制转换 //在使用（int）做转换的时候，统一截断小数部分 int num1 = 123; float num2 = 2.222f; char cNum = 'a'; Console.WriteLine(num1 + num2); int num3 = (int)num2; Console.WriteLine(num3); Console.WriteLine((num1 + num2).GetType()); //输出的是单精度浮点数据类型

**第二种转换方式：**

**第二种强制转换的方式：通过Convert.ToInt32()方法进行转换**

第二种强制转换的方式：通过Convert.ToInt32()方法进行转换 bool bValue = true; int num4 = Convert.ToInt32(bValue);

**第三种转换方式：**

**第三种强制转换类型（用于对字符串进行转换的方法）：in.Parse()方法来进行转换**

//第三种强制转换类型（用于对字符串进行转换的方法）：in.Parse()方法来进行转换 byte bNum = 234; string str = "2356"; int num5 = int.Parse(str); Console.WriteLine(num5);

**什么时候允许隐式转换？**

//什么时候允许隐式转换？？ float numA = 5.222f; //int numC = numA; 不能转换 float numB = numA; //可以转换 //能够从精度小的转换为精度大的

Conver.toint32()

(int)

Int.parse只能够把字符串转换为int类型

（int）用于简单的数据类型之间的转换

Conver.to()转换的数据类型更多

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace HomeWork

{

class Program

{

static void Main(string[] args)

{

#region 预处理指令

/\*

float num = 123;

string num2 = num.ToString();

Console.WriteLine(num.GetType());

\*/

/\*

//计算器

Console.WriteLine("请输入一个数字");

//正则表达式:字符或者字符串匹配

float num1 = float.Parse(Console.ReadLine());

Console.WriteLine("请输入符号");

char opt = char.Parse(Console.ReadLine());

Console.WriteLine("请输入第二个数字");

float num2 = float.Parse(Console.ReadLine());

//标识符

//标识符的命名规范

//1.不能以数字开头

//2.不能使用关键字命名

//3.不能有特殊字符

//4.见名知意

//5.驼峰命名法 匈牙利命名法 下划线命名法

/**/占位符**

**//在字符串里使用{0}的形式表示一个变量，下标从0开始依次+1,在字符串外用，分割，根据下标依次声明**

**//字符串的拼接一般用较少变量的时候**

**//模板字符串**

**if (opt == '+')**

**{**

**Console.WriteLine("结果为：{0}{1}{2}={3}", num1, opt, num2, num1 + num2);**

**}**

else if (opt == '-')

{

**//模板字符串**

**//相比于占位符来说，只是用一次变量时会更加方便**

**Console.WriteLine($"结果为{num1}{opt}{num2}={num1 - num2}");**

**}**

else if (opt == '\*')

{

Console.WriteLine($"结果为{num1}{opt}{num2}={num1 \* num2}");

}

else if (opt == '/')

{

if (num2 == 0)

{

Console.WriteLine("除数不能为0");

}

Console.WriteLine($"结果为{num1}{opt}{num2}={num1 / num2}");

}

else if (opt == '%')

{

Console.WriteLine($"结果为{num1}{opt}{num2}={num1 % num2}");

}

\*/

#endregion

#region 简易的登录注册

//先注册 账号 一次账号 两次密码

//在登录一次账号 一次密码

Console.WriteLine("请输入注册账号");

string userName = Console.ReadLine();

Console.WriteLine("请输入注册密码");

string userpw = Console.ReadLine();

Console.WriteLine("请确认注册密码");

string pw = Console.ReadLine();

if (userpw != pw)

{

Console.WriteLine("两次密码不一致请重新输入");

}

else

{

Console.WriteLine("注册成功");

**Console.Clear();//清空控制台**

Console.WriteLine("请输入账号");

string user = Console.ReadLine();

if (user != userName)

{

Console.WriteLine("账号不存在");

}

else {

Console.WriteLine("请输入密码");

string passwd = Console.ReadLine();

if (passwd != userpw)

{

Console.WriteLine("密码输入错误");

}

else

{

Console.WriteLine("登录成功");

}

}

}

#endregion

}

}

}

运算符

//算数运算符+-\*/%

//c#作为强类型语言，不支持除数为0的算法

//保留两位小数：Console.WriteLine(num.ToString("0.00"));可以通过ToString的格式化方法来保留

//逻辑运算符

// && 与 都为真才为真

//||或 有一个真，就为真

//！非 取反

//短路求值：找到一个真就不找了

//在c#中分为单与单或 双与和双或 单与和单或可以当做双与和双或使用，也可以做位运算符使用，建议只用双与和双或，单与和单或只用来表示位运算符

//自增自减运算符

//++ --

//赋值预算符

//= += -= /= %=

//int num3 = 8;

//赋值过程：根据int这个数据类型，向计算机申请一段四个字节的内存空间，将8这个数字转换成对应的二进制数，然后对这一段内存空间进行填充

//再将这一段空间名取名为num3

//变量是什么？？？

//num3是变量名

//变量是内存空间

//比较运算符

//> = < >= <= == !=

//位运算符

//& | ~ << >> ^

int numA = 5 & 3;

//三目运算符

//判断条件? 为真时运行表达式1:为假时运行表达式2

//表达式的结果需要用变量存起来

//int numB = 9;

//string str = numA >= 9 ? "yes" : "no";

//int x = 10;

//int y = 3;

//string z = x > y ? "x大" : "y大";

//Console.WriteLine(z);

//随机数

//随机数的范围是左闭右开区间

Random rand = new Random(); //首先取到一个随机值

int numX = rand.Next(0, 10); //再从随机值里挑选指定范围的随机数

int numY = rand.Next(0, 10);

int max = numX > numY ? numX : numY;

//转义字符 \n \t

Console.WriteLine("numX:{0}numY:{1}\n max:{2}",numX,numY,max);

**//Equals():用作判等的方法，多用于引用类型之间的相等的方法**

**//bool isTrue = numX.Equals(numY);**

#region 使用三种方法交换两种变量

//使用三种方法交换两种变量

//借助第三个变量

//int a = 7;

//int b = 6;

//int c = a;

//a = b;

//b = c;

//Console.WriteLine("{0},{1}", a, b);

//int x = 7;

//int y = 6;

//int sum = x + y;

//x = sum - x;

//y = sum - y;

//Console.WriteLine("{0},{1}", x, y);

//int j = 7;

//int k = 6;

//j = j ^ k;

//k = j ^ k;

//j = j ^ k;

//Console.WriteLine("{0},{1}", j, k);

#endregion

//遍历数组找到每个数相同的个数，记录找到的次数 可以异或

//int[] nums = {1,1,1,1,2,2,2,2,3,3,3,4,4,4,4 };

//int answer = nums[0];

//for (int i = 1;i<nums.Length;i++)

//{

// answer ^= nums[i];

// //answer = answer ^ nums[i];

//}

//Console.WriteLine(answer);

//现在有一堆石头，AB，两个玩家轮流取石头，可以取单个的某一个石头，也可以取相邻的两块石头

//问，有没有必赢得可能，（取完石头的获胜）

//3个石头开始 先拿的必赢

//4个石头 先拿的必赢

//5个石头 先取中间，先拿者必赢

//6个石头 先取中间两个，先拿着必赢

//条件语句

//单分支语句

//if（判断条件）

//{

// 执行语句

//}

//语句块

// {

// var numa = 10;

//}

//双分支语句

//if (判断条件)

//{

// 执行语句

//}

//else//不满足判断条件的时候执行

//{

// 执行语句

//}

//多分支语句

//从上至下一次判断条件，执行第一个满足条件的语句

//if ()

//{

//}

//else if ()

//{

//}

//else

//{

//}

//switch语句

//C#明确要求，在case的执行语句后必须加上break

//int one = int.Parse(Console.ReadLine());

//char opt = char.Parse(Console.ReadLine());

//int two = int.Parse(Console.ReadLine());

//int num =0;

//switch (opt)

//{

// case '+':

// num = one + two;

// Console.WriteLine(num);

// break;

// case '-':

// num = one - two;

// Console.WriteLine(num);

// break;

// case '\*':

// num = one \* two;

// Console.WriteLine(num);

// break;

// case '/':

// if (two == 0)

// {

// break;

// }

// num = one / two;

// Console.WriteLine(num);

// break;

// case '%':

// num = one % two;

// Console.WriteLine(num);

// break;

// default: //除以上情况才执行的语句

// break;

//}

//三目运算符

//循环语句

//跳转语句

//continue:略过本次。执行下一次循环

//break：跳出当前结构

//return:跳出当前函数

//goto:强制跳转到另一个函数

//for循环

//for (初始值; 判断条件; 递进规律)

//{

// 循环语句块

//}

//for (; ; )//死循环

//{

//}

//C#中可以通过字符串的长度获取到字符串的长度

//所有字符串都可以当做数组来访问或使用

//字符串数组中的每一个元素数据类型为字符

//string strTest = "你好， 世界";

//for (int i = 0; i < strTest.Length; i++)

//{

// Console.WriteLine((int)strTest[i]);

//}

//while循环：一般多用于知道条件而不知道执行的次数时

// while (true) {

//}

//do while()

//int user = 0;

//do

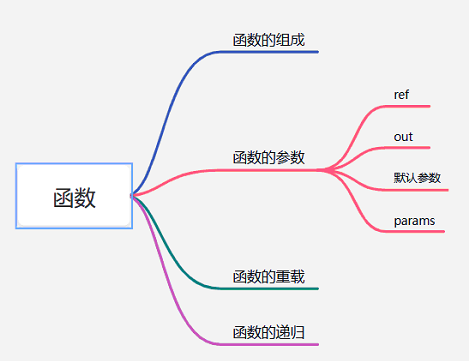
//{

// Console.WriteLine("请输入账号");

// user = int.Parse(Console.ReadLine());

//} while (user != 666);

函数



**//函数的组成**

**//静态 返回值类型 （参数列表）**

**static void Main(string[] args)**

**//（空）函数名**

**{**

**{函数体}**

**}**

**//函数的调用**

**Sum(num1, num2);**

**Console.WriteLine(Sum(num1,num2));**

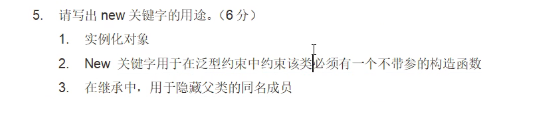
**Login();**

**//写法规范，函数名首字母大写**

**//参数列表允许有n多个参数**

**//return的结果类型返回值的类型必须和函数的返回值类型保持一致**

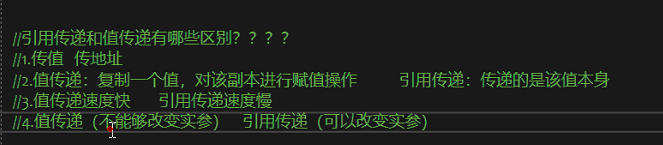
**New关键字：**

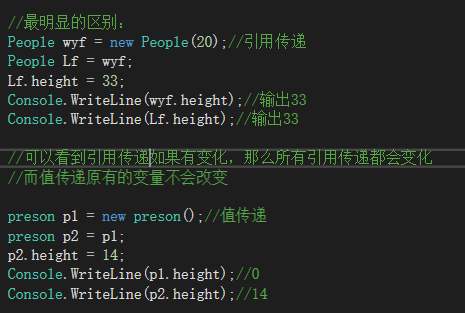


//值传递和引用传递的区别：

//值传递：传值

//引用传递：传址





//把引用传递转为值传递（拆箱）

//***out关键字***；使值传递变为引用传递（装箱），在实际调用时，需要在对应的实参前，把关键字加上 函数内传址

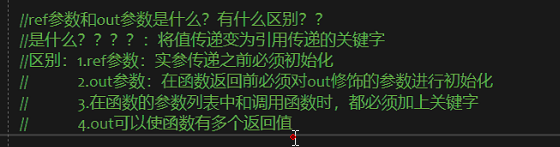
//**使用out关键字时，必须在函数返回前对out参数进行赋值操作**

**//可以不对实参传递之前进行赋值操作**

**//可以返回多个值**

Swap(out num1, out num2);

static void Swap(out int num1, out int num2)



//***ref关键字***：使值传递变为引用传递（装箱），写在函数的参数列表和调用时参数前 函数传值之前传址

**//使用ref关键字时，必须在实参传递之前进行初始化操作**

**//可以在函数内对ref修饰的参数不进行赋值**

Swap(ref num1, ref num2);

static void Swap(ref int num1, ref int num2)

**//默认参数（可选参数）**：当形参没有对应的实参时，使用默认参数的值

//再具体使用时，可以不给实参，但是不传递实参时，会用到默认参数的值

//所有的默认参数必须放在非默认参数之后

static int Sum(int num1, int num2 = 120)

//非默认参数也称之为必选参数，在调用函数的时候，函数有几个必选参数 就必须有几个对应的参数，

**//params修饰的参数可以实现一个不定长度的参数列表**

**//params修饰的参数一定是放在函数参数列表的最后面**

**static void Print(params int[] arr)**

{

**//foreach和for循环遍历的区别：**

**//1.foreach不能指定从哪个位置开始遍历，也没有判断条件和递进规律**

**//2.foreach的名字：迭代器，可以遍历线性和非线性容器。但是for循环只能够遍历线性容器**

foreach (var item in arr)

{

Console.WriteLine(item);

}

}

**使用foreach的容器必须继承于IEnumerable接口并实现其中的GetEnumerator函数**

**其中的属性：**

**Current：当前所停留的元素**

**MoveNext：下一个要遍历的元素**

**Yield关键字（语法糖）**

1. **通常与return一起连用，实际上不会跳出循环或函数，会继续执行下一条语句**
2. **Yield实际上是帮助我们实现了一个迭代器**
3. **Yield return实际返回的是一个迭代器，不是数值，系统会把返回的迭代器中的当前值Current输出**

***所有的引用类型都继承于Object，所有的值类型都间接继承于Object***

***这么做是为了把所有值类型和引用类型联系起来***

***装箱：将值类型转换为object类型***

***拆箱：将object类型转换为值类型，常伴有强制类型转换***

**迭代器的特点：**

1. **必须把容器中所有元素全部遍历一遍**
2. **可以选择从哪个元素开始遍历**
3. **迭代器是通过foreach来遍历的**

**//函数的重载**



//函数的返回值的类型相同，函数名相同，参数列表不同（个数，参数的类型）

//当函数名相同时，根据参数列表对应的实参来调用对应的函数

static void SumNum(int num1, int num2)

{

//return num1 + num2;

}

static void SumNum(float num1, int num2, int num3)

{

//return num1 + num2;

}

//程序的二义性

**//函数的递归**

**//在函数内部调用自己**

//斐波那契数列 0 1 1 2 3 5 8....

static int Feb(int index)

{

if (index == 0)

{

return 0;

} else if (index == 1 || index == 2)

{

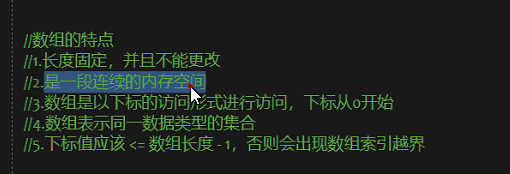
return 1;

}

return Feb(index - 1) + Feb(index - 2);

}

**数组**



//数组：同一数据类型的集合

//**特点**：1.是一段连续的内存空间

//2.在C#中数组的长度一但固定，就不允许再更改

//3.数组的访问是通过下标的形式进行访问，下标从0开始

//4.数组的长度和数组的元素个数是一致的,如果访问时下标小于0或者大于等于数组长度，会出现索引越界

**//一维数组的声明**

*//第一种方式：直接给定元素*

*//int[] array = { 1, 2, 3, 4, 5 };*

*////第二种方式：只给定长度*

*//int[] array1 = new int[3];//3代表数组长度*

*////第三种方式:同时给定数组长度和所有元素*

*//int[] array2 = new int[3] { 1, 2, 3 };*

**//二维数组**

//第一种声明方式

int[,] arr = { { 1, 2, 3, 4 }, { 4, 5, 6, 7 } };

//第二种方式

int[,] arr2 = new int[4, 5];//表示三行三列的数组

//第三种方式

int[,] arr3 = new int[2, 2] { { 1, 1 }, { 2, 2 } };

**//如何取到二维数组的行和列**

**arr2.GetLength(0);//获取到二维数组的行**

**arr2.GetLength(0);//获取到二维数组的列**

**//交错数组**

**//交错数组是数组的数组**

**//是一个一维数组**

**//数组中的每个元素的数组长度可以不一样**

int[][] array = new int[2][];//2代表的是交错数组的长度

int[][] array2 = new int[2][] { new int[] { 1, 2 }, new int[] { 1 } };

枚举

枚举的本质是：数据类型：值类型

枚举的自解释性，其与整形的数据相互转换

1. 枚举的作用：是为了方便我们编写程序时，将同一个作用的变量存到一个枚举中
2. 在使用中，枚举的名字充当类型
3. 从第一个值开始，默认为0，依次递增，也可以手动进行赋值，赋值后在当前之后的所有值递增
4. 只能后赋值为int类型
5. 枚举类型中的值，用“，”分割
6. 在最后一个值后可以加逗号，也可以不加
7. enum Item
8. {
9. wall, //墙壁
10. empty,//空白
11. player,//人物
12. block,//障碍物
13. box,//箱子
14. end,//终点

}

调用



**结构体**

namespace 结构体

{ **//结构体是值类型的**

**//结构体关键字：struct**

**//为了去管理不同的数据类型**

struct People

{

public int num;

public float num1;

public string num3;

***//构造函数：初始化结构体***

***//构造函数没有返回值***

***//构造函数的名字与结构体的名字一样***

***//所有的结构体一但声明出来，都有自己默认的隐式无参构造函数***

***//结构体不允许拥有显示无参构造函数***

***//结构体中不允许对字段进行初始化***

***//如果写了一个显示的构造函数，那么结构体中的字段必须在该构造函数内进行完全赋值***

public People(int num, float num1, string num3)

{

this.num = num;

this.num1 = num1;

this.num3 = num3;

}

}

class Program

{

static void Main(string[] args)

{

//结构体的调用,实例化一个结构体的对象

People p1 = new People(1,2.1f,"3");

Console.WriteLine(p1.num);

//声明名字为LY的结构体，构建了一个无参的构造函数的实例，赋值给LY

People p2 = new People();

p2.num = 200;

p2.num1 = 13.0f;

p2.num3 = "ljkhsdflk";

}

}

}

**类**

如何封装一个类：

//封装一个类

class People

{

public char sex;

public int age;

public float height;

public float weight;

public int phoneNumbel;

public string address;

public void Eat()

{

}

public void Sleep()

{

}

public void Walk()

{

}

}

//类和结构体的区别

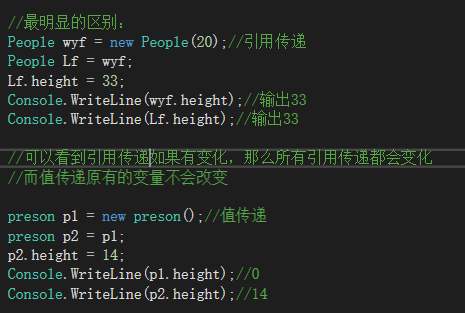
//1.类是引用类型，结构体值类型

//2.类可以在内部对字段进行初始化，但是结构体不可以

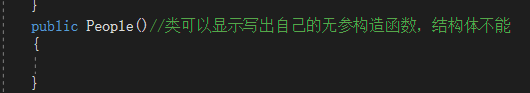
//3.在运算量大的时候用结构体，运算量小的时候用类

//4.栈（结构体所在位置）的空间小，但效率高。堆（类所在位置）的空间大，但是效率低

//5.类中进行的是引用传递，结构体中进行的是值传递



**构造函数的区别：**

//1.类可以显示可以写出自己的无参构造函数，但是同样也拥有一个隐式的无参构造函数，但是一但写出了显示的无参构造函数，那么就会覆盖掉系统默认的隐式无参构造函数

//2.类中的显示有参构造函数可以不用对类中的字段进行完全赋值，结构体必须完全赋值

//3.无论一个变量的赋值是否直接初始化，在程序的编译过程中，都是将这个变量的赋值过程放到了构造函数中

**Static**

//public访问权限修饰符

**//静态，修饰符**

**//static具有唯一性**

**//用static修饰的成员称之为静态成员（包括函数，变量等等），**

**静态成员属于类，而不属于对象**

***声明一个静态成员：***



/**/调用方式，通过类名 . 点静态成员来调用**



**非静态成员通过对象 . 点成员去调用的**



**//static修饰的静态方法中只能够使用静态成员**

**//静态构造函数（就是为静态成员变量做初始化赋值的）**

**如果一个类中有静态成员，那么这个系统就会为这个类生成一个静态构造函数**

**//可以由我们自己写出显示的静态构造函数**

**//静态构造函数不能够添加访问权限修饰符**

**//静态构造函数不能有参数**

**//静态构造函数由系统调用，不能由用户调用，每次都会最先执行**

**静态构造函数因为static的唯一性，静态构造函数只会执行一次**

**属性**

***属性一般用在类的外部，类内部一般使用字段***

访问权限修饰符：

**Public**：公开 类及类型成员的修饰符

**Private**:私有 对当前结构当中公开，对外私有，默认是私有，不写访问权限默认私有，因此私有可以不写

**Internal**:在当前命名空间内公开class默认修饰符是internal

**Protected**：受保护的 能够被该类和该类的父类访问到

属性

可以简单理解为变量Plus

// readonly可以修饰字段，使其成为只读字段，写在数据类型之前

Public readonly string name

想要属性设置只读，只需要去掉set访问器

在写属性的时候至少需要一个访问器

Public string Name

{

Get//get访问器，获得当前属性值 本质上会在程序编译的过程中将访问器转换为函数的形式

{（如果只有get访问器，该属性只能被访问而不能修改）

Return Name;

}

Set//set访问器：设置当前属性值

本质上会在程序编译的过程中将访问器转换为函数的形式

{（如果只有set访问器，该属性只能够被修改，而不能被调用）

Name = value;

}

}

属性变量名因该开头大写

快速属性：

*可以通过属性的方式在外部访问或者修改一个私有字段*

先声明一个字段，右键重构该字段，直接封装为属性

Public int age:

自动属性：

Public string Sex{get;set;}

在vs中输入prop+tab\*2可以快速生成自动属性

自动属性必须具有get和set两个访问器，创建readonly自动实现属性时，需要将set访问器设置为private

自动属性上可以使用特性，但是不能够用在支持后备字段上，如果属性上的后备字段上使用了特效，则因该只创建常规属性

自动属性get和set访问器不能够包含逻辑处理，不能够将属性作为ref或out参数传递

**内存自动释放**

1. 析构函数

当对象结束了生命周期，所在的函数已经调用完毕，系统会自动执行析构函数

1. GC回收机制：

***设计模式***

**单例模式**：让代码的安全性提高

让一个类只会有一个对象存在

单例模式创建方法，建议复制

**class 类名**

**{**

**//创建了一个私有静态的类类型的变量（该类的对象，将该变量置空）**

**private static 类名\_instance =null;**

**//私有构造函数，保证只能有一个对象，保证了安全性**

**private 类名()**

**{**

**}**

**//属性**

**public static 类名Instance**

**{**

**//可以获取这个属性**

**get**

**{**

**//如果该类的对象为空，则实例化一个出来**

**if (\_instance == null)**

**{**

**\_instance = new 类名();**

**}**

**//否则直接返回该对象**

**return \_instance;**

**}**

**}**

**}**

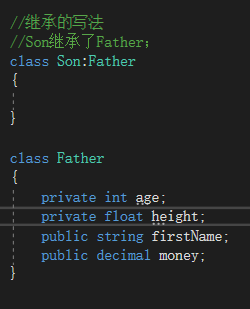
**继承**

类的三大特性：

封装，继承，多态

**继承：已经封装好的一个类，如果其它的类继承了这个类，那么就可以使用这个类中所有的成员**

**作用：提高代码的复用性**

//继承的写法

//Son继承了Father；

//被继承的类：父类（基类）

//继承的类：子类 （派生类）

注意，这里的age和height是已经继承了，但是不能被调用，因为访问权限是私密的

想要拿到的话需要用属性。

//protected:受保护的：只能在自己内部或者自己的子类中去访问

**//构造函数的调用顺序;1.子类的静态构造2父类的静态构造函数3.父类的构造函数4.子类的构造函数**

//如果子类调用子类构造函数时，没有指定调用父类的哪一个构造函数，则默认调用无参构造函数

//如果子类调用自己的构造函数的时候，希望使用父类中的某一个构造函数，可以通过：base()的方式指定调用

//base()代表父类的无参构造；base(1)代表父类的有参构造；base（c参数名）代表父类中该参数列表的构造函数

在自己的有参构造函数中调用自己的无参构造函数，通过:this（）调用

//this（）代表调用自己的无参构造；this(参数列表)代表调用自己的该参数列表的构造函数，注意（）里的参数列表只能是静态

*不可以被继承的内容：（其它都为可以继承）*

1.构造函数和析构函数

2.密封类

密封类：

Sealed class Name

{

关键字sealed，密封类无法产生子类（派生类）,无法被继承，但是可以继承

}

静态类：

//静态类中所有成员都是静态成员，或者常量成员

//静态类也是一个密封类

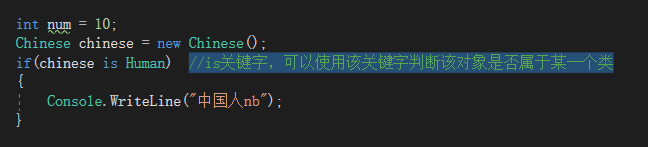
static class Test

{

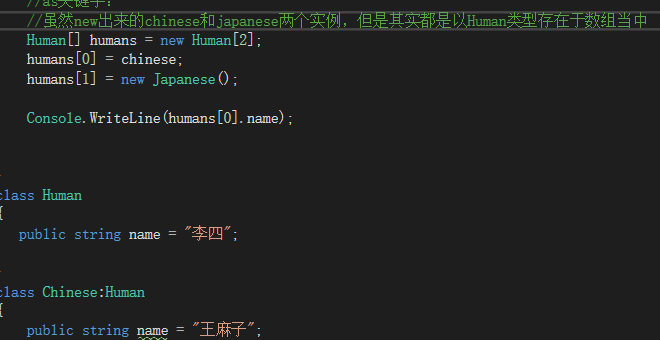
}

**引用类型转化**

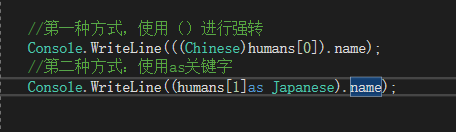
//is关键字，可以使用该关键字判断该对象是否属于某一个类



注意：



转换：



强转和as转换的区别：如果强转失败的话，会直接报错。如果as转换失败，那么不会报错，但会将null值赋值过去

//隐式转换:从子类的对象转为父类的对象的时候

//显示转换:从父类的对象转为换为子类的时候

//引用类型之间转换必须有继承关系

C#中不允许多重继承，一个子类只能拥有一个父类

//接口：可以辅助多重继承

里氏交换（LSP）原则：子类类型能够替换其父类型，并且出现在父类型能够出现的地方

**接口**

接口：可以辅助多重继承

规范代码，给后面写的内容一个强制模板

起到提纲的作用，为之后的代码提供一个思路

提高代码的书写效率

接口是引用类型

//接口的关键字：interface

//写法默认的规则，在接口命名前加上前缀 I

interface IFly

{

//接口中的方法：1.不能添加任何的访问修饰符2.不能够拥有方法体

void Fly();

}

调用接口：

class Bird : IEat//普通类接口写法

{

}

class Majio:Bird,IFly,IEat //继承类接口写法

{

public void Fly()

{

Console.WriteLine("鸡翅");

}

}

**//接口的关键字：interface**

**//写法默认的规则，在接口命名前加上前缀 I**

**//接口中可以存在的成员：1.函数的声明 2.属性（不是自动属性）只能够有get和set访问器的声明 3.索引器的声明 4.事件的声明**

**//接口中不能存在的成员：1.字段 2.构造函数**

**//继承接口的类中必须实现接口里的所有成员**

**//接口的隐式实现：当两个不同接口中的方法名字一样时，通过只写名字的方法实现接口中的成员，这种方式称之为隐式实现接口**

**//接口的显示实现:通过接口名点.成员名形式来区分该成员属于哪个接口。显示实现的接口成员不允许添加访问权限修饰符**

**//接口可以被什么继承？ 1.类 2.结构体 3.接口**

**//结构体不能继承类，但是可以继承接口**

**//类可以继承接口，可以继承类，但是不能继承结构体，结构体本身是一个密封的类型**

**//接口可以继承接口，但是不可以继承类和结构体**

**多态**

**定义：多态指的是不同事物（需要同一个父类或者接口），在做同一件事情的时候所表现出来的不同形态，多态是继承的扩展**

**抽象类（用关键字abstract修饰的类）**

**仅在抽象类中可以使用abstract来修饰方法**

**被修饰的方法称为抽象方法，抽象方法没有方法体**

**注意：abstract，override，virtual和static无法一起使用**

**关键字abstract**

**修饰函数表示抽象函数，不具有函数体，且在子类中必须实现函数的具体函数功能**

**关键字override**

**重写：对具有相同函数签名（返回值类型，参数列表）使用override进行重写，更改方法中的代码**

**Overrride修饰的函数本质上也是一个虚函数**

**关键字virtual**

**修饰的函数为虚函数，虚函数声明时可以有函数体，用virtual所修饰的函数在其子类中不是必须实现**

**抽象函数和虚函数的区别**

1. **定义的关键字不同，分别使用abstract和virtual来定义**
2. **声明抽象函数不能有方法体{}，但是声明虚函数的时候必须有方法体**
3. **抽象函数只能在抽象类中，但是抽象类中还可以包含一般方法；虚函数则可以在任意类中，不必必须在抽象类中**
4. **抽象函数其子类必须进行重写覆盖，而虚函数则不必须进行重写或覆盖**
5. **抽象函数类似一种规范，用于约束子类必须实现某函数；而虚函数更偏向于代码的复用**

**//在开发过程中，遵循一个唯一指标；高内聚低耦合**

**//高内聚：每个函数尽量只实现一个功能**

**//把类中的各个功能细分成各个函数**

**//低耦合：尽量减少一个类对其他类的调用**

**设计原则**

1. **开闭原则：对于一个模块（类或者是函数）来说，扩展开放，修改关闭，不要去修改别人的源码**
2. **单一职责原则：对于一个模块，只包含一种功能**
3. **里氏交换原则：任何父类出现的地方，都能用子类替换之，子类的方法实现要比父类的更加严格**
4. **接口隔离原则：建立单一接口,接口中的方法尽可能的是少,尽可能细化接口,不要试图建立一个庞大的接口去提供所有依赖他的类(面向接口编程)**
5. **依赖反转原则:高层的模块不应该依赖于底层的模块,二者都应该于依赖于抽象类,抽象类不因该依赖于细节,而细节依赖于抽象类;**

**抽象类:抽象类和接口**

**细节:就是实现抽象类和具体类**

**(面向接口编程)**

1. **迪米特法则：一个软件的实体因该尽可能少的与其他软件实体发送相互作用，陌生的类最好不要出现在其他类的内部（降低耦合）**

**如何降低耦合**

1. **少使用继承，多用接口隐藏实现的细节**
2. **模块的功能分化尽量的单一**
3. **遵循一个定义只在一个地方出现**
4. **少用全局变量**
5. **类的属性和方法少使用pulick,多使用private**
6. **多使用设计模式（单例模式）**
7. **尽量不要使用“硬编码”的方式写程序，尽量避免使用SQL语句直接操作数据库**
8. **避免直接调用其它类或模块，如果两个模块之间必须存在耦合，尽量使用数据耦合，少使用控制耦合，限制公共耦合的范围，避免使用内容耦合**

**接口和抽象类相同点：**

1. **都不能创建实例化对象**
2. **都是用于规范后续代码的书写，提高效率**

**接口和抽象类不同点：**

1. **包含的成员不同，抽象类可以包含普通类中的所有成员，且能包含抽象成员（访问器，索引器，方法，委托）；接口只能包含抽象成员（访问器，索引器，方法，委托）**
2. **一个类只能继承于一个抽象类，但是能继承多个接口**

**Readonly:关键字，只能修饰字段，修饰的字段被称为只读字段（只能访问不能修改）**

**只读字段只能在构造函数舒适化时进行赋值**

**Const关键字，修饰的字段称为常量，只能在初始化进行赋值，之后无法更改**

**Const不能static关键字一起使用，但是可以和readonly一起使用**

**简单工厂模式**

**依赖注入**

1. **Setter注入**
2. **构造函数注入**
3. **\*接口注入**

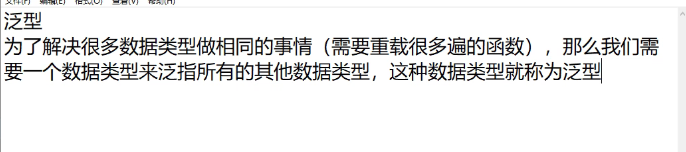
**由一个工厂类根据传入的参数，动态决定创建哪个产品（继承自同一个父类或者接口）的实例**

**成员：**

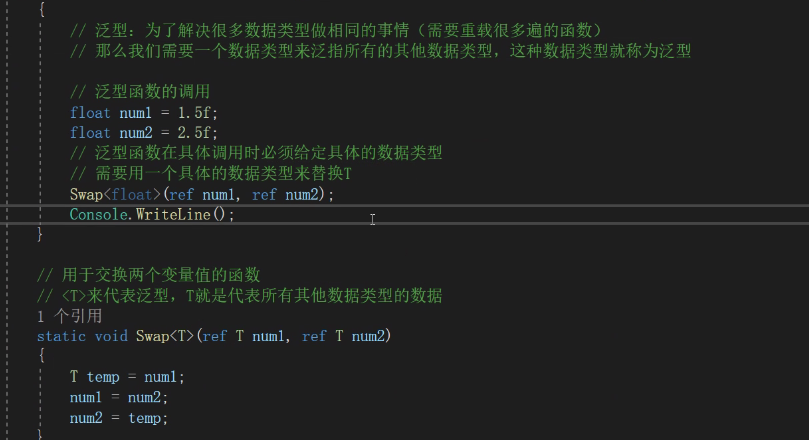
1. **工厂角色（OpertorFactory）:简单工厂模式的核心，负责实现和创建所有实例的内部逻辑。工厂类的创建，产品类的方法可以被外界直接调用，用于创建所需的产品、**
2. **抽象产品角色（AbstractProdyctRoles）：简单工厂模式的所有对象的父类，塔负责描述所有实例共有的公共接口**
3. **具体产品角色（四则运算产品角色）：简单工厂模式所创建的目标，所有创建的对象都是充当这个角色的某个具体类型**

**如何增加内聚：**

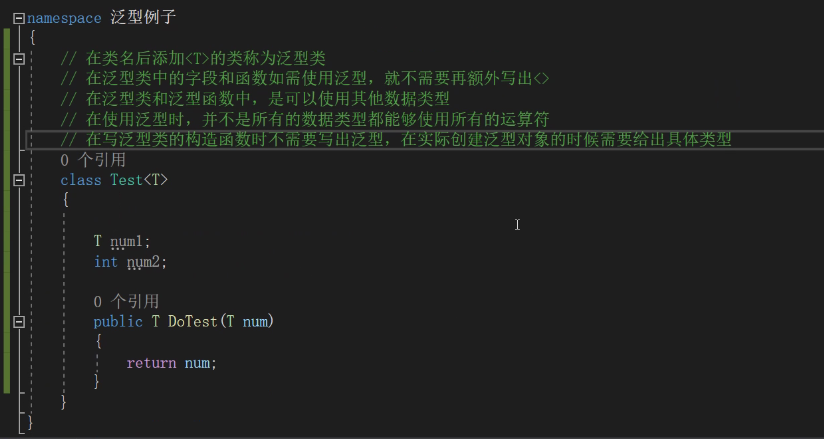
1. **模块只对外暴露最小幅度的接口，形成最低的依赖关系**
2. **只要对外接口不变，模块内部的修改，就不得影响其它模块**
3. **删除一个模块，应当只对依赖关系的其它模块有影响，而不影响无关模块**

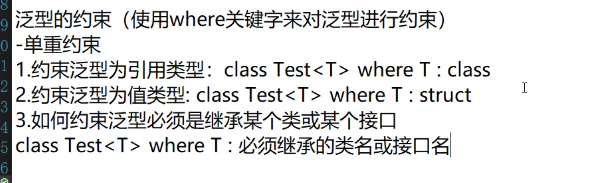


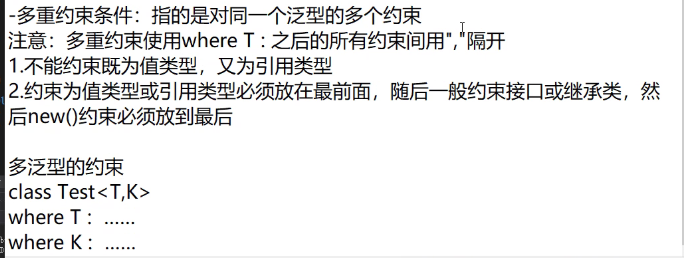
泛型的声明与调用方式：



泛型类







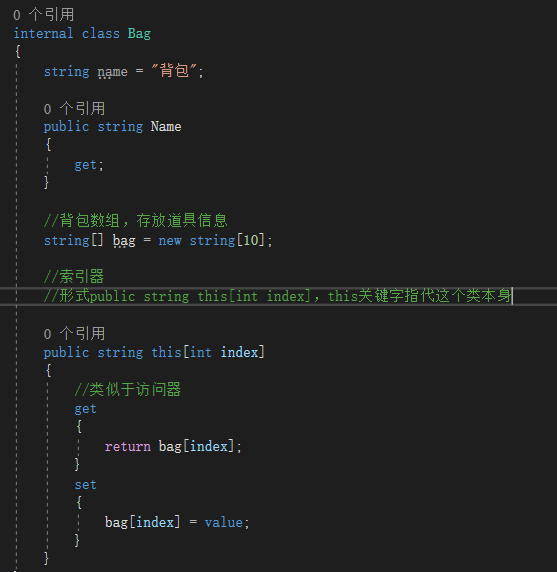


**索引器**

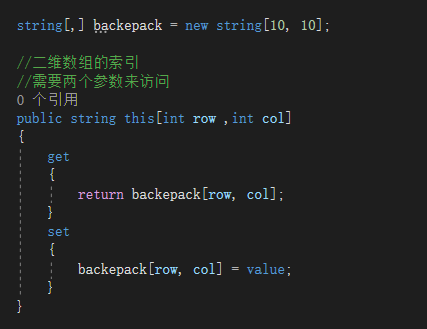
//形式public string this[int index]，this关键字指代这个类本身

//索引器中参数index用于指定所需返回的数组的具体下标

索引器写法：



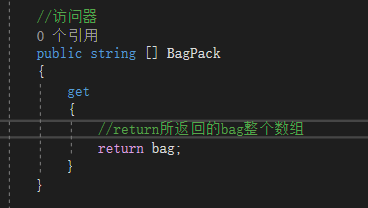
二维数组的索引器



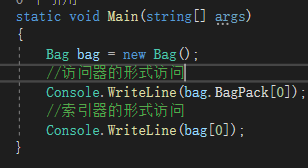
**索引器和访问器的区别：**

1. **访问器在写的时候要使用对应字段的大写形式，索引器使用的是this关键字。**
2. **访问器在使用的时候不会涉及到参数问题，但是索引器必须使用参数**
3. **索引器没有自动写法，在写get和set时需要明确写出返回的是哪个数组，并指定对应的下标**

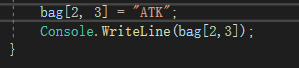
**访问器：**



调用索引器和访问器：



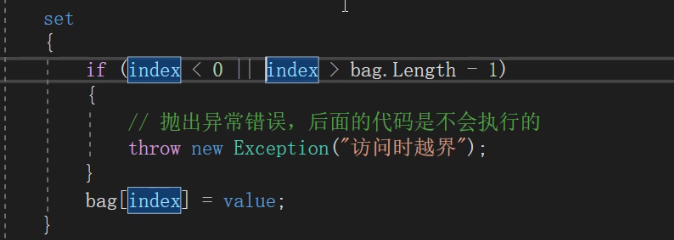
二维的索引访问

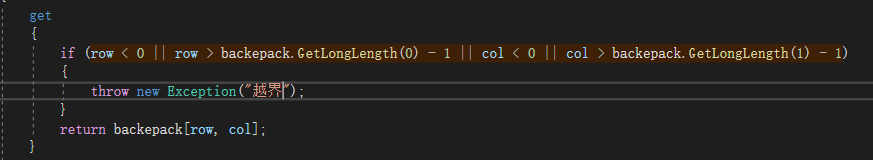


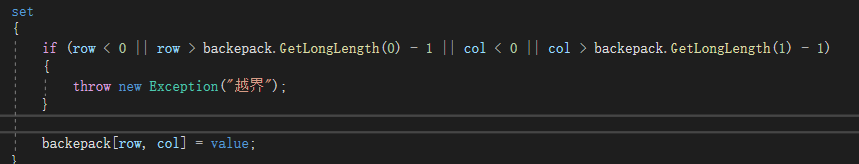
**索引器的特点**

1. **索引器的参数：类似数组的访问形式，参数一般来说指代数组的下标**
2. **越界：存在下标越界的问题（检测索引器越界）**



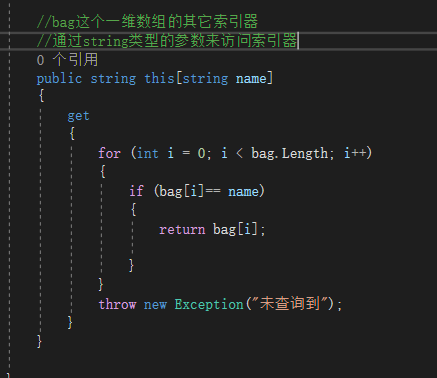
同理。二维数组判断越界要判断两个下标（行列）



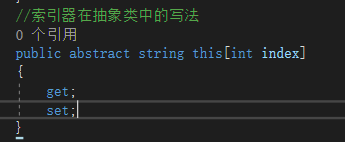


1. **参数个数：用于区分访问数组的维数（一维数组和二维数组）**
2. **参数类型：**

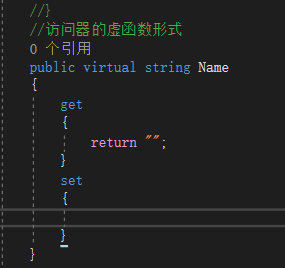
**（比如说根据名字查找这个一维数组）**



在抽象函数中***索引器***的写法



在虚函数中***访问器***的写法：虚函数中get和set要设置函数体



**数据结构**

*List<string> list = new List<string>();*

//增删改查API：

**//增：Add**

**list.Add("fssf");**

**//删：三个API**

**//删除指定下标处的元素 下标从0开始**

**List.RemoveAt(1);**

**或者删除找到的第一个元素**

**List.RemoveAt(“李四”);**

**找到并删除所有相同的元素**

**//list.RemoveAll(“所有”);**

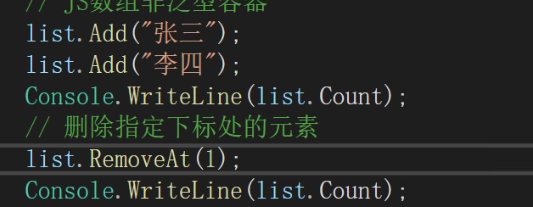
**清空列表所有元素**

**List.Clear();**

**//改：通过索引器改**

**List.Insert(1,”网5”);**

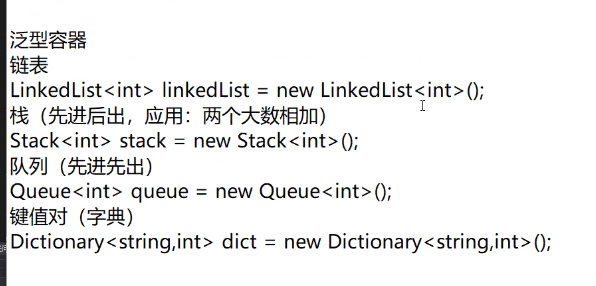
**//查：通过索引器来查**



**List列表的缺点**

1. **没有解决内存地址连续的问题，一旦内容过大，也会造成数据存不进去的情况**
2. **数组扩容时涉及了内存的重定向以及内存的销毁，效率不高**
3. **插入和删除操作涉及大量的数据偏移，效率不高**

**重点：链表LinkedList<>**



**链表的特点：**

1. **链表是一个线性容器（线性：指的是每个元素都有一对一的位置关系）**
2. **优点，解决了扩容和大量数据偏移的效率低的问题**
3. **缺点所有的查找都需要从头节点开始**

**栈（先进后出，应用：两个大数相加）**

**Stack<int> stack = new Stack<int>();**

**栈的功能：**

1. **入栈Push：在栈的末尾添加一个新元素**
2. **出栈Pop：将栈末尾的元素返回，并将其删除**
3. **查看栈顶Peek：返回栈顶（栈的末尾）的元素，但是不会将其删除**

**队列：先进先出**

**1.Queue<int>queue = new Queue<int>();**

**2.入队Enqueue:在队列末尾（队顶）添加一个元素**

**3.出队Dequeue：将队列队底的元素返回并在队列中删除**

**4.查看队底Peek：返回队底的一个元素，但不会将其从队列中删除**

**键值对（字典）**

**Dictionary<string,int> dict = new Dictionary<string,int>();**

**修改颜色**

