C# 高级编程个人笔记搬运 二 （类和对象、结构）

原创

拓拓龙 于 2020-04-27 18:16:54 发布 174 收藏 1
分类专栏： C#

我欲乘风破浪，踏遍黄沙海洋，
与其误会一场，也要不负勇往，
我愿你是个谎，从未出现南墙，
笑是神的伪装，笑是强忍的伤，
就让我走向你，走向你的窗，
就让我看见你，看见你的伤，
我想你就站在，站在大漠边疆，
我想你就站在，站在七月上。我高考时要有这首歌，我就不会就当码农了，应该去研究导弹去了 T_T。

来来来，接着搬运笔记，写到哪是哪，我很忙的，忙着发呆。前期的内容比较基础，都是基本功，但是也要搬运，磨刀不误砍材工嘛。其实主要是这笔记要从前到后写，别看这标题起的很基础，但是内容融贯深究起来还是有点话题的，太底层的等我看完我的新书吧，我尽力防止烂尾。

结构与类大致区别：

有一门课叫做《数据结构》，当然和这说的结构不是单纯的一回事，只突然想到就提一下。俗话说你学会了数据结构，你就可以封神。数据结构和算法是很紧密的关系，算法也是我最喜欢的课程了，探索最优的思路去解题，成就感爆棚。我要找个时间研究下数据结构的内容和一些算法题，还有我常提的“计算机操作系统”，O(∩_∩)O哈哈~

回归正题，我很少见到项目中有好好用结构的，有些宁到处组建一家子常量也不肯写一个结构，气死人了。但是不代表结构就不重要。

结构不同于类，因为它们不需要在堆上分配空间（类是引用类型，总是存储在堆上），而结构是值类型，通常存储在栈上，另外，结构不支持继承。凡是想着继承一个结构的都是脑子有泡的。

①、较小的数据类型使用结构可提高性能。虽然双方在语法上很相似，主要区别是使用关键字struct代替class来声明结构。

②、类和结构都要new来声明实例：这个关键字创建对象并对其进行初始化。

③、类和结构最重要的一个区别是，类类型的对象通过引用传递（引用传递其实是它指向它所存在的内存地址，当你去修改的时候他操作的是源数据），结构类型的对象按值传递（按值传递，其实是复制行为，将数据值复制一份过来，当你改变参数值时，你是动不到源数据的，你只是修改了它的替身，但是你此刻用的就是它的替身）。

类篇：

类包含静态成员和实例成员，静态成员属于类，实例成员对象。静态字段的值对每个对象都是相同的，而每个对象的实例字段都可以有不同的值。(细节看另一篇，我写了)。提一下，很多人为了方便经常把字段声明为public，这样是不合适的，该类的东西就应该该类才能去用，不能大门一开，谁都抢。所以我们要定义字段为private。

区分一下属性和字段，以前我也分不清。首先我们经常写的private string eyeColor;这个就是字段。而属性呢，就是用类似java的方式给它套上get、set的衣服。如：public string eyeColor{get；set；}这种就是属性（这种写法也叫作自动实现的属性，把它展的特别齐全就一般属性）。看到我这里写的是public而不是private，因为我们这个eyeColor它可以通过get、set的方式去控制访问，比如：public string eyeColor{get；private set；}，所以我们用才会不介意公开它（public）。

所以字段和属性的区别最大就在于属性更容易被灵活控制，因为读取分开可以做不同的访问设定。在你需要针对某些数据做一些配置的时候，最好用属性。而字段平常用的最多，几乎前面一个限定符就可以昭告天下你能不能用，该不该用。

那属性会不会比字段更耗性能？不会，别管他看起来多么七拐八弯，他的最终性能都是由编译成IL码决定的，JIT在编译成本地代码的时候，根本不会为属性多做多余的编译，它会自己去内联代码。

方法的实现如果只有一个语句，我们可以用lambda语法糖。比如:

public bool IsSquare（Rectangle rect）=> rect.Height == rect.width;

说到方法的使用，我们不谈重写（一模一样）和重载（参数不一样）的区别，我们着重谈一下一些容易被忘记的又高效的使用方法。

①、任何方法都可以使用命名的参数调用。只需要编写方法的参数名，后面跟一个冒号和所传递的值。如：

r.Area(height : 10, width: 10); // 这最大的好处就是让别人很明白你的意图是什么。

②、参数也是可选的,必须为可选参数提供默认值。写法如下：

```
// 切记可选参数一定要是最后的参数，
r.Area(double height, double width, string colour="red", bool shape=true) /
{
    // DO.....

    // 当然，可选参数的值你是可以修改的
    colour = "yellow";
}

// 由于参数是可选的，那我们在调用的时候就可以少传参数了
r.Area(10,10);
```

③、使用任意数量的同一类型的参数。写法如下：

```
// 这种写法最大的便利就是不用我们去创建一个数组再传进来
r.Area(params object[] parameter)
{
    // DO.....

    // 使用时就是直接按数组下标，但是千万要注意别超界！！！
    double height = x[0];
    x[2] = "yellow"
}

// 我们在调用的时候直接可以这样
r.Area(10,10,"red");
```

这种写法很方便，但有一点不好，就是你不联系上下文，你根本不知道x[0]、x[1]……这都是什么。所以这个语法在一定程度上可读性比较差。但是在一些情况下，这个写法是很好用的。

如果你只有一个参数需要去理解，其它的不需要，那你可以这些改进写法，使用可选参数，定义数量可变的参数：

```
// params只能出现一次，并且只能在最后出现
r.Area(string colour, params object[] parameter)
{
    // DO.....
}

// 我们在调用的时候直接可以这样
r.Area("red",10,10,);
```

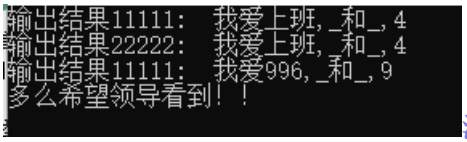
谈一下构造函数，其实说真的，这一块还是很有用的只是被用的少，前端应该用的比较多，当时做即时通讯软件的时候用的还是很给力的，大概场景就是当不同消息来了以后需要做的数据处理。

正常情况下编译器会为你在后台自动生成一个默认的构造函数，但是你如果有自己写构造函数，编译器就不为你自动生成了，它会假定你的才是唯一可用构造函数。一般情况下构造函数都是私有（peivate）或者保护类（protected），主要是为了避免篡改。但是咱们必须要明确一点，当你构造函数定为非公开（public）后，那么其它的类是不能对这个类进行实例化的（就是在其它类B中是无法new 出A它的对象的，但不代表AB类之间就没有联系了）。

当我们在定义多个参数的构造函数时，可能会碰到这几个构造函数会对同一字段进行初始化的情况，C#为我们提高了一个特殊的语法——构造函数初始化器，如下：

```
// 我们为类自定义了两个带参的构造函数
private Program(string aa, int bb)
{
    value = aa;
    b = bb;
    WriteLine($"输出结果11111:  {value},_和_,{b}");
}
private Program(string cc):this (cc,666 )
{
    WriteLine($"输出结果22222:  {value},_和_,{b}");
}

// 自定义的构造函数是不会对着类的加载而被调用的，那我们需要实例化去调用它现在
public static void Main()
{
    var p = new Program("我爱上班");
    var p1 = new Program("我爱996",123);
    WriteLine("多么希望领导看到！！");
}
```

在这里我们注意一下结果：



注意的话就是：要注意this指向的构造函数先输出。

我们总结下，实例成员或者方法它是属于类对象的，就是说它类B需要new ClassA( ).XXX来调用，也就是说我们自定义了非公开访问的构造参数后，类B又需要A的数据，那就只能通过在A中定义一个公开的静态方法。静态是属于类的，是共享的，可以由类去调用而不需要先实例化，我们就可以把国际问题转为内部问题，大事化小，小事解决。如：

```
// 新的类想要去Program类的某个属性值
class Class1
{
    public static int  pig = Program.BB();
}

// 再Program加一个静态方法BB()
 public static int BB()
{
    var p = new Program("我爱上班");
    return b;
}
```

目前编程中警惕的问题就是数据各处泛滥和静态的滥用，方便主义。

除了有参数的构造函数，我们还有一种叫做静态构造函数。这种构造函数的最大用处就是随着类的加载而调用，即模仿系统本身自带的构造函数（这里要解释下，我说的是模仿，而不是等同于。.NET运行库并不能保证静态构造函数它会在什么时候执行，单能保证它一定会执行一次。虽然通常是在这个类被初次使用时执行。多个类的静态构造函数彼此之间是没有任何执行顺序的）。而我们自定义的有或没有参数的构造函数都是要调用（即对象.调用），才会被启用的。静态构造函数的使用场景，一般来说都是用于无论如何都要出现的数据身上，当然前提是不在乎它何时出现。

静态构造函数它有三点值得我们注意一下：

①：静态构造函数没有访问符号，即不会被public、private、protected等待修饰；

②：静态构造函数一定是无参的，所以一个类也只能有一个静态构造函数（不是说有了静态构造函数就不能自定义普通的无参构造函数，他们不是一回事，其实静态这个关键符号static要是深入底层研究，会很有意思的，它带给人的感觉就像专制的王国）；

③：静态构造函数只能访问类的静态成员，不能访问类的实例成员（就是没有带static的）。

在预防数据篡改这块，你还可以用readonly修饰符。带有readonly修饰符的字段只能在构造函数中分配值。你和我一样想起了const修饰符，但是他们是不一样的：

const修饰的称为——常量：

①、当你在加载这个类的时候，它就会随着被分配空间，编译器会（也要）知道它的值，它能一直保持不变是编译器一直牢记它的值，并将值取代了使用它的变量。

②、在使用上，你可以类里直接const int c = 99;就可以被使用。在声明的时候就必须要初始化（原因就是第①点）。

例如：

```
class Program
{
    // 一定要初始化
    const int c = 99;
    public static void Main()
    {
       WriteLine($"输出结果:  {c}")
    }
}
```

readonly修饰的称为——只读：

①、这个字段的值只有在构造函数中被分配，就是说他分配空间的时候是在你调用构造函数的时候，你不用它，它就不占地方。

②、它可以是实例成员，也可以做类成员，如果是类成员，他即有类似方法一样的使用方式，同时他由于原因①而不必一开始就初始化。0

例如：

```
// 情形一：当只读字段为实例成员时
class Program
{
    //初始化最好是默认系统的构造函数，否则没多大意义
    readonly int cc=99;

    // 如果你不初始化，那么你就要自定义一个构造函数来给它赋值，你不肯也没事，因为默认是0
    // readonly int cc;

    public static void Main()
    {
       // 跟方法一样，非静态即实例，要有对象调用
       WriteLine($"输出结果：  {new Program().cc}")
    }
}

// 情形二：当只读字段为类成员时
class Program
{
    static readonly int cc=99;

    public static void Main()
    {
       // 跟方法一样，静态成员时，类中直接调用，类外由类.调用
       WriteLine($"输出结果：  {cc}")
    }
}
```

我不太想关注只读属性，因为只读与属性初衷上有点不太融洽（只读与字段比较搭，都是一刀斩的直爽派。属性本来就应该是花花公子）。对了，如果你是属于读取配置类，那么设置只读属性应该不错，写法上需要直接省略set访问器。如：

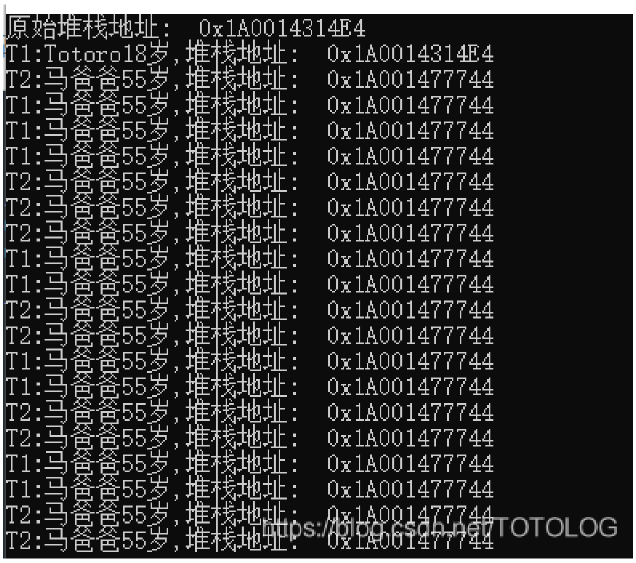
```
// 普通的只读属性，初始化在构造函数里
private readonly int age;
public int Age
{
    get
    {
        return age;
    }
    // 由于是只读属性，因为省略了set
}
```

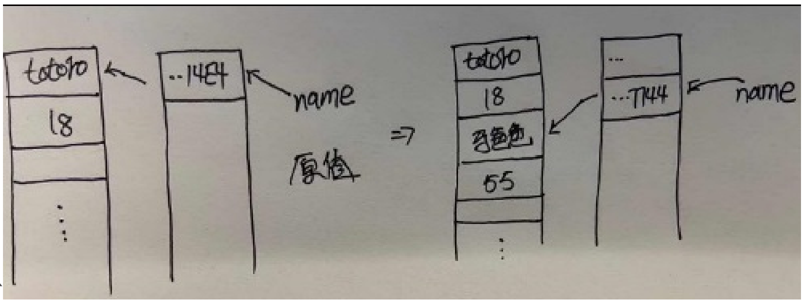
说到这个，要提一下——表达式属性，一眼看上去是字段，其实是属性的一个写法。写法上就是省略了访问器，用lambda表达式给它赋值，前提是你的表达式要比它先运算出来。如：

```
private string a { get; set; }
private string b { get; set; }

// c会随着a、b的值而改变自身
public  string c => $"{ a }{ b  }";
```

还有一个很重要的概念——不可变类型，平常很少被提及但是很常被使用的一个点。我也上网看了很多的东西，我总结了下大概意思就是，当你是不可变类型，你原先的存在堆栈的值不会被改变，一旦你开始赋予新值，其实是另开辟一个堆栈空间存新值，并改变你的指向为新空间的值或地址。整个操作就不会影响到原先堆栈的值。如：

```
class Program
{
    public string name = "Totoro";
    public int age = 18;

    static void Main(string[] args)
    {
        Program p = new Program();

        // 先找出原先的地址
        GCHandle handle = GCHandle.Alloc(p.name, GCHandleType.Pinned);
        IntPtr addr = handle.AddrOfPinnedObject();
        Console.WriteLine($"原始堆栈地址:  {$"0x{addr.ToString("X")}"}");

        // 起两个多线程
        Task.Run(() => T1(p));
        Task.Run(() => T2(p));

        // 一会解释下为啥要这个（先不管）
        Thread.Sleep(3000);
    }

    private static void T1(Program p1)
    {
        for (int i = 0; i < 10; i++)
            {
                GCHandle handle = GCHandle.Alloc(p1.name, GCHandleType.Pinned);
                IntPtr addr = handle.AddrOfPinnedObject();
                Console.WriteLine($"T1:{p1.name}{p1.age}岁,堆栈地址:{$"0x{addr.ToString("X")}"}");
            }
    }

    // 二号码农想把我变成马爸爸
    private static void T2(Program p2)
    {
            p2.name = "马爸爸";
            p2.age = 55;
            for (int i = 0; i < 10; i++)
            {
                GCHandle handle = GCHandle.Alloc(p2.name, GCHandleType.Pinned);
                IntPtr addr = handle.AddrOfPinnedObject();
                Console.WriteLine($"T2:{p2.name}{p2.age}岁,堆栈地址:{$"0x{addr.ToString("X")}"}");
            }
    }
}
```

结果： 

从结果来看 

，所谓的赋值只是创建了一个新的空间存值，并更改了引用地址。这时原本的就是不可变对象。

不可变类型指的是对象所在内存块里面的值不可以改变，有数值、字符串、元组；

可变类型则是可以改变，主要有列表、字典。

结构

类在封装对象后，其数据的生存周期就具有了很大的灵活性，但是性能上会有一定的损失。有时候使用一个小的数据结构，会让我们的性能提高。

因为结构是值类型，所以当我们用对一个结构用new的时候，它应不会像引用类型一样在堆中分配内存，它只是简单的调用结构的构造函数，并根据传送的参数值去初始化所有的字段。

还有结构的声明，其实实际上是尾整个结构在栈中分配了空间，所以j结构可以直接声明后就赋值。如：

```
class Program
{
    public string name;

    static void Main(string[] args)
    {

        // 对于类而言，声明并不意味真的就被创建了，因为会报错
         Program p;
         p.name = "totoro"; // 报错使用了为赋值的局部变量

        // 而对于结构而言，声明以为在栈中存在它的一席之地了，它就是正确的
        STR1 str1;
        str1.eye = "棕色";
    }
}

// 定义一个结构体
struct STR1
{
    public string eye;
}
```

但是要注意，虽然声明的结构体已经被分配了空间，但是并不是说它已经被赋默认值。如：

```
class Program
{
    public string name;

    static void Main(string[] args)
    {
        // 结构在声明后只会被分配空间，而不会被赋予任何值，所以会报错
        STR1 str1;
        var color = str1.eye; // 使用了可能未被赋值的字段eye
    }
}

// 定义一个结构体
struct STR1
{
    public string eye;
}
```

讲一下结构和类被当做参数传递时，他们的一些变化本质。如：

```
// 定义一个结构体
public struct STR1
{
    public string eye;
}

class Program
{
    // 一个使用结构的方法
    public static void WW(STR1 str1)
    {
        // 当结构体在这个方法体内，eye 就会是 蓝色
        str1.eye = "蓝色";
    }

    static void Main(string[] args)
    {
            STR1 str1;
            str1.eye = "棕色";
            WW(str1);

            // 当结构体出了使用它的方法体，他就会还是棕色
            Console.WriteLine(str1.eye);    // 这里输出结果就是棕色
    }
}
```

结果输出是“棕色”，如上诉注释所表达。本质是结构体它是按值传递的，预示当方法WW得到的其实是结构体class1在堆栈中变量eye的一个副本，就是说真实的class1的eye变量从来没有被更改过。

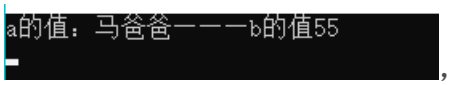
那么如果STR1不是结构体而是一个类呢？因为类是按引用传递的，那么当把对象作为参数传递时，实际更改的就是该对象的该字段值，而不会因为出了方法体就会被销毁、恢复。所以结果就会是“2”。

当你需要对结构做一个保证其实时一致性时，C#提供给我们这样一个修饰符——ref。如：

```
// 定义一个结构体
public struct STR1
{
    public string eye;
}

class Program
{
    // 一个使用结构的方法
    public static void WW(ref STR1 str1)
    {
        // 此时是按引用传值，操作的是原来的结构体字段值
        str1.eye = "蓝色";
    }

    static void Main(string[] args)
    {
            STR1 str1;
            str1.eye = "棕色";
            WW(ref str1);

            // 因为源数据被修改了，所以这里输出就会是 “蓝色”
            Console.WriteLine(str1.eye);
    }
}
```

在传参这一块，除了关系传过去的还要关心传回来的。c#有一个修饰符——out，它的作用就是可以支持你传多个不同类型的参数回来。当然我们常用的就是把需要传回来的多类型结果设为一个类或者结构。或者我们设为元祖传回来。都可以实现，但是这里要表达的是用——out去实现。如当我们需要传回来一组数据时：

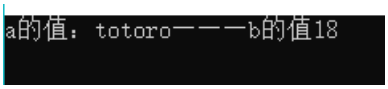
```
public class Program
{
    // 一个没有返回值的方法
    public static void WW( out string i, out int j)
    {
        i ="马爸爸";
        j = 55;
    }

    static void Main(string[] args)
    {
        string a= "totoro";
        int b =18;

        // 这里有两点注意：
        // 一、必须有out带领参数；二参数必须是变量，不能直接WW( out “totoro”,out 18)
        WW( out a,out b);
        Console.WriteLine($"a的值：{a}——b的值{b}");
    }
}
```

我们看一下结构： 

，看这样就把a、b的值传回来。

要是不用out，a、b肯定是不会改变的。如：

```
public class Program
{
    // 一个没有返回值的方法
    public static void WW(string i, int j)
    {
        i ="马爸爸";
        j = 55;
    }

    static void Main(string[] args)
    {
        string a= "totoro";
        int b =18;
        WW( a,b);
        Console.WriteLine($"a的值：{a}——b的值{b}");
    }
}
```

看下运行结果： 

。不要和之前的eye弄混了，因为之前是把类或结构作为参数传过去的。而现在是把字段作为参数传过去。

其实out最常用的是在做一些不确定数值处理的时候，比如传过来一个串，你无法确定它是数字还是字符时。你就可以使用out去对传过来的串做处理，而不会引起报错。 如：

```
static void Main(string[] args)
{
    // 比如当我们相对某个串进行大小写转换
    string str =  ReadLine();
    int result;
    if (int.TryParse(str , out result))
    {
        WriteLine($"n: {n}");
    }
    else
    {
        WriteLine("not a number");
    }
}
```

当我们输入 

后，正常情况下将其进行大小写转化肯定就会报错，但是用了out就不会，结果就是： 

。

C#中有一种叫做匿名类型，就是我们常用的var，我不想过多去解释这个，因为几乎或多或少都有所了解。我大概记录了下匿名类型它是可以声明一个类的匿名对象。因为匿名类型只是一个继承自object且没有名称的类，类似于隐式类型化的变量。如：

```
// 声明一个匿名类型，这会生成一个对象，它没有明确是哪个类对象
var captain = new
{
    name = "Totoro",
    age = 18
};
```

这些新对象的类型名未知。编译器为类型名伪造了一个名词，也只有编译器能使用它。我们不能也不应该对这个对象的任何类型进行反射。

接着是介绍的是可空类型。可空类型的理解也是一块骨头，引用类型可空，值类型不可空（int a = null），这是一个常识。但是实际在使用时我们可能需要它可空，比如对应XML一个结点或者数据库里表的某一个属性。以前为了处理这一个问题，我们常常会把值类型删掉改成引用类型，但是它会带来额外的开销，这就不是一个最优解。因为对于引用类型，它是等待垃圾收集器去清理的，而值类型时当它超出了作用域时，它就会被清理，从内存中删掉。如：

```
public class Program
{
    Class1 class1 = new Class1();

    // 当我们离开了Class1 之后，Class1 的a就会被虫内存中删除
    Thread.Sleep(5000);
}


public class Class1
{
    public string eye;
    int a = 13;
}
```

于是c#引入了一个可空类型——可空类型就是可以为空的值类型。可空类型只需要在你声明的类型后面加上？，这样下来你所需要的唯一开销就是一个可以确定它是否为空的布尔成员。这个在和类型数据库做交互的时候是很有用处的。

```
// 普通的时候值类型都是
int x = 18;

// 使用可空类型的时候
int? y = null;
```

在值类型里，有一种常用的类型，叫做——枚举类型，用关键字enum定义。常用法：

```
public enum E
{
    RED,
    GREEN,
    BLUE,
    age = 18
}

// 输出age、RED
E e = E.age;
E color = E.RED;
```