


Unity 3D游戏编程自学#2——C#面向对象

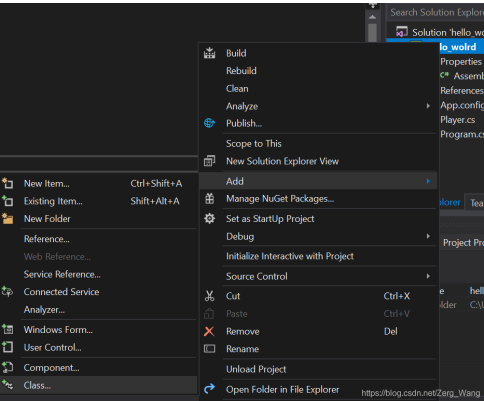
 Game Programming 专栏收录该内容

0 订阅7 篇文章

1.类与对象

类，简单来说就是高级版的结构体。类是用于描述一类事物的，而对象是某一种具体的类。比如说有一个类叫做“玩家”，类里面描述的是“玩家”的具体信息，那么小明就是属于“玩家”这类的一个具体的对象。

创建一个类：



举个例子，我们创建了一个叫Player的类：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace hello_world
{
    class Player
    {
    }
}
```

之后我们在Main函数中我们就可以调用这个类了：

```
Player xiaoming = new Player();
```

接下来介绍类中的三大成员：字段、属性、方法。

2.字段和属性

描述类的变量都不叫变量，叫字段。

字段的取值、赋值与结构体一样，字段命名前都要加public。

如果不想让外部程序可以直接访问类中的字段，可以不用public，改为private。（public和private叫做访问修饰符，也称为权限修饰符）

这样一来，在Main函数中就无法直接对该字段进行取值、赋值。（但在类中该字段可以被随意访问）

每个Private字段都需要一个唯一对应的属性来进行操作：

属性语法：

```
1 public 数据类型 属性名
2
```

```

3 | {
  |   4 |     get { return 字段名; }
5 |     set { 字段名 = value; }
6 | }

```

get函数用于取得对应字段的值，set用于赋值。

简写版，即自动属性（与上图功能一样，上图的叫普通属性）：

```
public 数据类型 属性名 { get; set; }
```

在属性内部可以通过代码防止字段被恶意赋值。（比如在set里面加入if语句等）

例子：

```

class Player
{
    private int level;
    public int Level
    {
        get { return level; }
        set { level = value; }
    }
}

```

https://blog.csdn.net/Zerg_Wang

属性名一般与字段名一样（不过属性名开头字母大写，规则同帕斯卡命名法）。如果要对字段进行特殊处理，则可在set函数中进行编写。

在Main函数中调用：

```

1 | xiaoming.Level = 18;
2 | Console.WriteLine(xiaoming.Level);

```

3.方法

类中的函数叫做方法。

方法有三种：普通方法、构造方法以及析构方法。

普通方法：

与正常函数的使用一样，不过前面要加访问修饰符（可private也可public）

构造方法：

即C++中的构造函数，一般用于初始化类中字段。格式：

```

1 | public 类名()
2 | {
3 |
4 | }

```

注意：构造方法的名字一定与类名一模一样，而且不能是private型的。

构造方法无返回值，也不用写void。

构造方法可重载，若没有写构造方法，系统会自动生成一个。

例子：

```

class Player
{
    private int level;
    public int Level
    {
        get { return level; }
        set { level = value; }
    }
    public Player(int level)
    {
        this.level = level;
    }
}

```

https://blog.csdn.net/Zerg_Wang

赋值时需要注意：要给类中的字段赋值，则必须在前面加上this修饰。

其实不加this也可以，但加this，一方面是为了防止与方法中的参数或者某些局部变量区分，另一方面也是为了减少bug。

调用的时候：

```
Player xiaoming = new Player(18);
```

若写成重载，则可以有多种初始化形式。

对象初始化器：对于有多个字段的类，可能需要写多个参数不同的构造方法（有时这几个字段需要初始化，有时那几个又要）。为减少代码量，可以使用对象初始化器（前提：相关字段的属性编写完成）

例子：

```

class Person
{
    private string name;
    private int age;
    private string address;
}

```

https://blog.csdn.net/Zerg_Wang

假设Person极多字段，现在我只想初始化name和age，这时我不需要再写一个仅有name和age两个参数的构造方法，仅需要完成这两个字段的属性，然后：

```
Person p1 = new Person() { Name = "Z", Age = 12 };
```

大括号内的属性数不限。

注意：要使用对象初始化器，类中至少要有一个无参的构造方法（什么构造方法都没有其实也可以）。

析构方法

即析构函数，用于销毁无用对象，不写系统会自动生成，对象销毁时系统会自动调用（无论是自己写的还是系统生成的）。

格式：

```

1 | ~类名()
2 | {
3 |
4 | }

```

析构方法不能有任何参数。

4.堆栈关系

对象是一种引用类型，若一个对象由另一个对象赋值而来，则他们的值会同步变化。

```

1 | Player p1 = new Player(18);
2 | Player p2 = p1;
3 | p2.Level = 10;

```

具体一点，程序执行时，生成的p1放在了栈空间（这一步叫做对象的实例化，通过实例化关键字new实现），但p1的具体数值被放在了堆空间，因此栈空间的p1指向了堆空间中存储自己数据的地方。当把p1赋值给p2时，p2指向了堆空间中相同的地方，因此无论改变哪个，两者的值都会变。

5.命名空间

用于对代码进行分类管理，一般是类的集合。

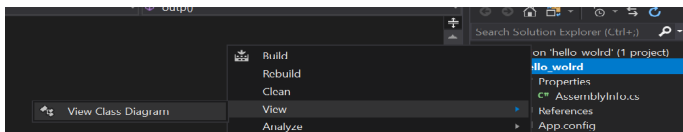
定义：namespace 空间名称 { 类 }

调用：using 空间名称

面向对象开发三大特点：继承、多态、封装

6.继承

为了方便以后理清子类父类之间的关系，可用Visual Studio自带的视图功能：



Visual Studio 2017要另外装Visual Studio extension development中的Class Designer才能拥有此功能。

```
1 | class 子类名:父类名
2 | {
3 |
4 | }
```

子类会继承父类所有的成员（字段、属性和方法，无论是private还是public，这些都能继承到，除了构造方法），但所有父类中的private成员子类都是无法访问的。

一般类中的字段写成private，属性和方法写为public。

在子类中，若要访问父类的public成员：

base.字段名

base.属性名

base.方法名() 若有参数还要带上参数

构造方法随不能继承，但子类也可调用父类的构造方法：在子类构造方法的后面直接加上: base(参数)

例子：

```
1 | public Son(int level) : base(level)
2 | {
3 | }
```

base中不用写数据类型（调用父类的普通方法时也一样），参数名必须和Son构造方法的参数名一致，说明是把子类构造方法的参数传给父类。

虽然子类在使用父类的public成员时需要加base，但在Main函数函数中子类的对象可以直接调用（子类对象名.父类成员）。

若子类对象之间直接赋值，仍存在引用传递的现象，无论这个字段是子类的还是父类的。

7.多态

不通过增加子类的成员，使继承同一父类的多个子类有各自不同的、有别于父类的特性。

有三种方法实现多态：虚方法、抽象类、接口。

虚方法：

父类的方法中加入关键字virtual，子类的方法加入关键字override：

例子：父类名为User,有方法“技能”

```
1 public virtual void JiNeng()  
2 {  
3     Console.Write("玩家使用了技能: ");  
4 }
```

子类Mage：

```
class Mage:User  
{  
    public override void JiNeng()  
    {  
        base.JiNeng();  
        Console.WriteLine("火焰冲击");  
    }  
}
```

https://blog.csdn.net/Zerg_Wang

实例化后输出为：玩家使用了技能：火焰冲击

在使用虚方法时，一般会在父类的方法中实现所有子类共有的东西（比如初始化），而在子类中加上个性化的代码。

若不需要父类中的初始化，base.JiNeng()是可以删去的。

抽象类：

若某父类普通方法无需初始化，或者只能交给子类完成，仅在父类实现一个定义作用（即父类方法无任何操作），则可以把该方法写成抽象方法。

```
1 abstract class User  
2 {  
3     public abstract void JiNeng();  
4 }
```

抽象方法只能存在于抽象类中，但抽象类可以没有抽象方法。

抽象类无法实例化。

一旦父类的普通方法写成了抽象方法，则子类一定要重写该抽象方法，重写语法与虚方法一样。（此特性可以防止子类功能不全）

在Visual Studio中，有一个功能可以一键补全子类重写的方法：

对子类名右键第一项——Quick Actions and Refactorings——Implement Abstract Class。

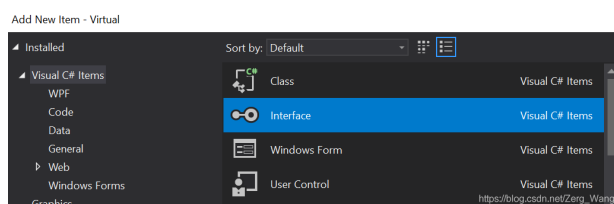
之后的接口也有类似的便捷功能。

接口（interface）：

一种特殊的类，有属性，但没有字段，所有的方法都是抽象方法，所以也无法被实例化。

接口的命名：与类名命名规则一样，前面多加一个大写的“I”。

接口的创建与创建类的路径一样，但有区别：



接口从类中分了出来。

接口的语法比较特殊，其方法必须都是public且抽象，所以接口格式：

```

namespace Virtual
{
    interface ICareer
    {
        void Introduce(string name);
        void Battle();
        int Attack();
    }
}

```

https://blog.csdn.net/Zeng_Wang

接口中所有的方法都不能有方法体。

一个类无论有没有继承父类，都可以同时继承一到多个接口，格式：

```

1 | class Son:Father, 接口1, 接口2
2 | {
3 |     public void 接口中的方法()
4 |     {
5 |         ...
6 |     }
7 | }

```

在类中使用接口中的方法，无需加abstract。（注意：与抽象类一样，类一旦继承某接口，就一定要重写接口中的所有方法，）

一个接口也可以继承多个接口，格式与上述例子类似。

接口中的属性：用于规定子类中的字段的读写权限。

举例：

```

interface IWork
{
    int Attack
    {
        get;
        set;
    }
}

```

则继承IWork的子类必须要有一个名为attack的int型字段，而且还要重写其普通属性。此外，因为IWork中规定了这个字段是可读(get)也可写的（set），因此子类的这个属性必须要有这两个功能：

```

class Mage:IWork
{
    private int attack;

    public int Attack
    {
        get { return attack; }
        set { attack = value; }
    }
}

```

https://blog.csdn.net/Zeng_Wang

若接口的属性只有get，那么子类的对应属性也要有get，set则可有可无。则对set同样适用。

要实现多态，这3种具体选择哪一种实现？

如果子类方法的共性多，需要统一初始化，则可以使用虚方法。

如果仅仅让父类定义一系列规范，则可以使用抽象类。

若要直接拓展类的功能，则可以使用接口。

8.封装

五种访问修饰符：public、private、internal、protected、protect internal

1.public：当前类、实例对象、子类、子类的子类.....均可访问。

2.private：仅当前类可以访问。

3.protected：仅当前类、子类、子类的子类.....可以访问。

4.internal：仅能在当前程序集（项目）中访问，也就是说，如果在同一项目下，internal与public一样。

5.protected internal：在同一项目下与public一样。

类仅有public和internal两种，默认为internal。

类的成员五种都有，默认为private。

9.Static（静态）

static关键字，可修饰类及其所有成员（被static修饰的都叫做“静态XX”，如静态字段、静态属性、静态方法、静态类）

静态成员

静态成员的语法与普通成员一样，仅在访问修饰符后面、数据类型前面加上关键字static。

```
1 class Person
2 {
3     private static string name;
4     public static string Name { get; set; }
5     public static void SayHello()
6     {
7         Console.WriteLine("Hello,{0}", name);
8     }
9 }
```

静态成员不属于任何实例化对象，它们在类实例化之前就已经存在，若要调用它们，格式：类名.静态成员名：

```
Person.Name="Zerg";
```

因此，无论实例化多少个对象，静态成员在程序中的数量与类的数量相当。

静态字段无法用this修饰。

注意：在静态方法中无法调用非静态的方法！但非静态方法可以调用静态方法。

Main函数是静态的，这也是为何我们之前写的函数要加上static关键字。

静态构造方法

用于初始化静态成员，一个类中仅能有一个静态构造方法，且该静态构造方法无任何访问修饰符及参数。例：

```
1 static Person()
2 {
3
4 }
```

静态构造方法会在实例化第一个对象、第一次调用静态成员前完成。

静态类

在class之前加上static即可。

静态类中不能有非静态成员，也不能实例化。

单例设计模式

在设计某个类时，要保证程序运行时最多只有一个实例对象存在（例如Windows的任务管理器，只能开一个，如果开多了，内容还一样，则浪费资源，内容不一样，则会让系统陷入混乱），此时就需要用到单例设计模式。以下是流程：

```

class TaskManager
{
    //步骤1：声明一个静态且私有的当前类的字段。
    private static TaskManager instance;
    //步骤2：创建一个私有的构造方法。保证外部无法实例化。
    private TaskManager() { }
    //步骤3：创建一个静态方法。用于创建此类的唯一对象。
    public static TaskManager Instance()
    {
        if (instance == null)
        {
            instance = new TaskManager();
        }
        return instance;
    }
}

```

https://blog.csdn.net/Zerg_Wang

这样一来，在Main函数中：

```

static void Main(string[] args)
{
    TaskManager t1 = new TaskManager();
    TaskManager t2 = TaskManager.Instance();
    TaskManager t3 = TaskManager.Instance();
}

```

https://blog.csdn.net/Zerg_Wang

我们无法直接实例化出t1，只能调用类中的方法创建，但因为类中的if语句，创建出的t3和t2是完全一样的，这就保证了有且只有一个实例对象存在。

10.特殊的类（实际上很少使用）

嵌套类

在类中再定义一个类：

```

class TaskManager
{
    public class CPU
    {
        private string name;
        public string Name { get; set; }
    }
}

```

https://blog.csdn.net/Zerg_Wang

实例化方法：

```
TaskManager.CPU my_CPU = new TaskManager.CPU();
```

如图例子中，CPU叫做嵌套类，TaskManager叫做外部类。

匿名类

创建方式：（属性数量不限）

```
var 匿名类名 = new { 属性1 = 值1, 属性2 = 值2, 属性3 = 值3};
```

调用方法：类名.属性名：

```

1 | var player = new { Name = "Zerg", Level=50};
2 | Console.WriteLine(player.Name);

```

匿名类的值定义后就不能修改！

密封类

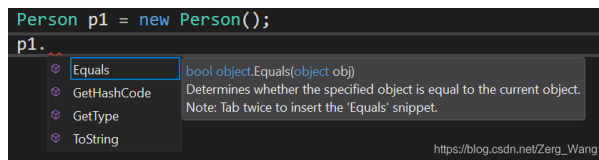
不能有子类的类叫做密封类

创建密封类，只需要在class前加上关键字sealed即可，这个类就无法被任何类继承。

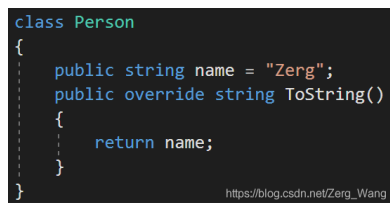
Object类

最原始的类，所有的类都是或直接或间接继承自Object类。

Object类本身自带了一些方法，对于一个无任何成员的类，它因为继承自Object而先天带上了4个方法：



这四个中除了GetType都是虚方法，可以进行重写，例如对ToString进行重写：（原来的ToString以字符串形式返回该对象的类及这个类所处的命名空间）



11.装箱与拆箱

注意区别：开头大写的Object是类，而开头小写的object是类型（int、double这些都是类型）。

装箱：将值类型转换为引用类型。

拆箱：将引用类型转换为值类型。



装箱和拆箱本质上是数据存储在栈空间和堆空间的变更，若频繁装箱、拆箱会降低代码运行效率。

12.预编译指令

又称预处理指令，在程序编译运行前存在，这些指令在程序在编译时会自动忽略。（和注释一样）

这里介绍一下其中一种预编译指令：区域指令。

```
1 | #region 该段区域指令名
2 |
3 | #endregion
```

用法举例：



假如我的类中字段太多太杂，可以在字段前后使用区域指令，然后左边会出现一个“减号”的按钮，按一下：



可以把这一区域的代码折叠，方便用户审查代码，理想效果：



本文部分内容来自撻码网 (<http://www.mkcode.net>) Unity 3D课程，经本人学习、整理得来，若有错漏，欢迎指正！