**Console\_.NET Farmwork**

目次

目次

[１. 入门篇 6](#_Toc106738999)

[１.１ C#编写的各类应用程序 6](#_Toc106739000)

[１.２ 调试 6](#_Toc106739001)

[１.３ Ｃ＃文件类型 6](#_Toc106739002)

[２. 基础篇 8](#_Toc106739003)

[２.１ 数据类型 8](#_Toc106739004)

[２.１.１ Ｃ＃有五大数据类型 8](#_Toc106739005)

[２.１.１ Var 9](#_Toc106739006)

[２.１.１ Dynamic 9](#_Toc106739007)

[２.１.２ 值类型 9](#_Toc106739008)

[２.１.３ 引用类型 9](#_Toc106739009)

[２.１.４ 值类型和引用类型的区别 9](#_Toc106739010)

[２.１.５ Ref 10](#_Toc106739011)

[２.１.６ out 10](#_Toc106739012)

[２.２ 显示类型 10](#_Toc106739013)

[２.３ 隐式类型 10](#_Toc106739014)

[２.４ 操作符 10](#_Toc106739015)

[２.４.１ New 11](#_Toc106739016)

[２.４.２ 移位运算 11](#_Toc106739017)

[２.４.３ 算术运算符 11](#_Toc106739018)

[２.４.４ 赋值运算符 12](#_Toc106739019)

[２.４.５ 关系运算符 12](#_Toc106739020)

[２.４.６ 逻辑运算符 12](#_Toc106739021)

[２.４.７ 其他运算符 12](#_Toc106739022)

[２.４.８ 运算符重载 13](#_Toc106739023)

[２.５ 语句 13](#_Toc106739024)

[２.５.１ 迭代语句 13](#_Toc106739025)

[２.５.２ 选择语句 13](#_Toc106739026)

[２.５.３ 跳转语句p 13](#_Toc106739027)

[２.５.４ 块语句 13](#_Toc106739028)

[２.６ 表达式 14](#_Toc106739029)

[２.６.１ 成员访问表达式 14](#_Toc106739030)

[２.６.２ Lvakue 14](#_Toc106739031)

[２.６.３ Rvalue 14](#_Toc106739032)

[２.７ 变量 14](#_Toc106739033)

[２.７.１ 成员修饰符 14](#_Toc106739034)

[２.７.２ 成员访问修饰符 14](#_Toc106739035)

[２.７.３ 成员变量 14](#_Toc106739036)

[２.７.４ 局部变量 14](#_Toc106739037)

[２.７.５ 类变量（静态属性\_全局变量） 15](#_Toc106739038)

[２.８ 常量 15](#_Toc106739039)

[２.８.１ 静态常量(Const[编译时常量]) 15](#_Toc106739040)

[２.８.２ 动态常量(Readonly[运行时常量]) 15](#_Toc106739041)

[３. 中级篇 16](#_Toc106739042)

[３.１ 类的封装 16](#_Toc106739043)

[３.１.１ 类概念 16](#_Toc106739044)

[３.１.１ 访问修饰符 16](#_Toc106739045)

[３.１.２ 类声明（Class declarations） 16](#_Toc106739046)

[３.１.１ 类的成员 17](#_Toc106739047)

[３.１.２ 静态成员和实例成员的区别 17](#_Toc106739048)

[３.１.３ 字段 17](#_Toc106739049)

[３.１.１ 属性 19](#_Toc106739050)

[３.１.２ 索引器 19](#_Toc106739051)

[３.１.３ 构造函数(构造器) 19](#_Toc106739052)

[３.１.１ 静态类 20](#_Toc106739053)

[３.１.２ 静态构造函数 21](#_Toc106739054)

[３.１.３ 析构函数（实例析构器－垃圾回收器） 21](#_Toc106739055)

[３.１.４ 局部类（partial） 21](#_Toc106739056)

[３.２ 方法（method） 21](#_Toc106739057)

[３.２.１ 访问修饰符 21](#_Toc106739058)

[３.２.２ 结构体 21](#_Toc106739059)

[３.２.３ 静态方法 22](#_Toc106739060)

[３.２.４ 实例方法 22](#_Toc106739061)

[３.２.５ 方法重载（Overload） 23](#_Toc106739062)

[３.２.１ 方法参数 23](#_Toc106739063)

[３.２.２ 方法的调用 24](#_Toc106739064)

[３.２.３ 枚举 24](#_Toc106739065)

[３.２.４ 抽象方法（虚方法） 24](#_Toc106739066)

[３.２.５ Lambda表达式 24](#_Toc106739067)

[３.３ 字符串 24](#_Toc106739068)

[３.３.１ 数据类型转换 24](#_Toc106739069)

[３.３.２ 隐式类型转换 24](#_Toc106739070)

[３.３.３ 强制类型转换 24](#_Toc106739071)

[３.３.４ 装箱和拆箱 24](#_Toc106739072)

[３.４ 数组 24](#_Toc106739073)

[３.５ 类的继承 24](#_Toc106739074)

[３.５.１ 继承的本质 24](#_Toc106739075)

[３.５.２ 单继承 25](#_Toc106739076)

[３.５.３ 多继承 25](#_Toc106739077)

[３.５.４ 方法重写 25](#_Toc106739078)

[３.５.５ 类成员的隐藏 25](#_Toc106739079)

[３.６ 类的多态 25](#_Toc106739080)

[３.６.１ 抽象类 25](#_Toc106739081)

[３.６.２ 多态的本质 26](#_Toc106739082)

[３.６.３ 静态多态 26](#_Toc106739083)

[３.６.４ 动态多态 26](#_Toc106739084)

[３.６.５ 向上转型 26](#_Toc106739085)

[３.６.６ 向下转型 26](#_Toc106739086)

[３.７ 接口 26](#_Toc106739087)

[３.７.１ 接口的本质 26](#_Toc106739088)

[３.７.２ 显示接口 27](#_Toc106739089)

[３.７.３ 隐式接口 27](#_Toc106739090)

[３.７.４ 接口方法 27](#_Toc106739091)

[３.８ 内存 27](#_Toc106739092)

[３.８.１ 栈 27](#_Toc106739093)

[３.８.２ 堆 27](#_Toc106739094)

[３.９ 反射与依赖注入 27](#_Toc106739095)

[３.９.１ 反射的本质 27](#_Toc106739096)

[３.９.２ 特性 27](#_Toc106739097)

[３.１０ 正则表达式 27](#_Toc106739098)

[３.１１ 配置文件 27](#_Toc106739099)

[３.１２ 词法结构 27](#_Toc106739100)

[３.１２.１ 标记 27](#_Toc106739101)

[３.１２.２ 预处理指令 28](#_Toc106739102)

[３.１３ 集合 29](#_Toc106739103)

[３.１４ 委托(delegate) 29](#_Toc106739104)

[３.１４.１ 直接调用 29](#_Toc106739105)

[３.１４.２ 间接调用 29](#_Toc106739106)

[３.１４.３ 委托声明(自定义委托) 29](#_Toc106739107)

[３.１４.４ Action委托 29](#_Toc106739108)

[３.１４.５ Func委托 29](#_Toc106739109)

[３.１４.１ 多播委托（multicast） 29](#_Toc106739110)

[３.１４.２ 单播委托 29](#_Toc106739111)

[３.１４.３ 委托的一般使用 29](#_Toc106739112)

[３.１４.４ 隐式异步调用 30](#_Toc106739113)

[３.１５ 事件 30](#_Toc106739114)

[３.１５.１ 定义 30](#_Toc106739115)

[３.１５.２ 角色 30](#_Toc106739116)

[３.１５.３ 使用 31](#_Toc106739117)

[３.１５.４ 原理 31](#_Toc106739118)

[３.１５.５ 提示 31](#_Toc106739119)

[３.１６ 泛型 31](#_Toc106739120)

[３.１６.１ 泛型类 31](#_Toc106739121)

[３.１６.２ 泛型方法 31](#_Toc106739122)

[３.１７ 异常 31](#_Toc106739123)

[３.１８ ADO.NET数据库操作 31](#_Toc106739124)

[４. 高级篇 32](#_Toc106739125)

[４.１ 进程与线程 32](#_Toc106739126)

# 入门篇

## C#编写的各类应用程序

|  |  |
| --- | --- |
| Console应用程序控制台 | WPF(Windows Presentation Foundation) |
| Windows Forms (Old) | ASP.NET Web Forms (Old) |
| ASP.NET MVC (Model-View-Controllr) | WCF (Windows Communication Foundation) |
| Windows Store Application | Windows Phone Application |
| Cloud (Windows Azure) | WF (Workflow Foundation) |

## 调试

|  |  |
| --- | --- |
| Step Into(F11键) | 查看方法内部做了哪些调用 |
| Step Over(F10键) | 快速大范围定位，查找Bug |
| Step Out(Shift+F11键) | 查看打在断点方法的上一层 |

## Ｃ＃文件类型

|  |  |
| --- | --- |
| .sin | 解决方案文件，为解决方案资源管理器提供显示管理文件的图形接口所需的信息 |
| .exe | 在编译时生成可执行文件 |
| .dll | 在编译时生成动态链接库，公共文件，类库 |
| .pdb | 在编译时生成程序数据库 (PDB) 文件保存着调试和项目状态信息，使用这些信息可以对程序的调试配置进行增量链接。 |
| .aspx | Web窗体页由两部分组成：视觉元素（HTML、服务器控件和静态文本）和该页的编程逻辑。 |
| .aspx.cs | Web 窗体页的编程逻辑位于一个单独的类文件中，该文件称作代码隐藏类文件（.aspx.cs） |
| . aSax | Global. aSax文件是一个可选的文件，该文件包含响应 ASP.NET 或 HTTP 模块引发的应用程序级别事件（如Session、Application等的操作）的代码。 |
| .cs | 类模块代码文件。业务逻辑处理层的代码。业务代码 |
| .config | Web.config 文件向它们所在的目录和所有子目录提供配置信息。 |
| .aspx.resx/.resx | 资源文件，存放一些你这个Form上所用到的一些资源，比如说图片等 |
| .XSD | 在图形界面下生成的XML文件，用来保存对数据库的查询、插入、删除、更新等方法，可以重复调用。 |
| .suo | 解决方案用户选项,记录所有将与解决方案建立关联的选项，以便在每次打开时，它都包含您所做的自定义设置。 |
| .asmx | asmx 文件包含 WebService 处理指令，并用作 XML Web services 的可寻址入口点。 |
| .vsdisco | 文件基于XML的文件，它包含为Web 服务提供发现信息的资源的链接 (URL)。 |
| .htc | 一个HTML文件,包含脚本和定义组件的一系列HTC特定元素.htc提供在脚本中implement组件的机制。 |
| .skin | 皮肤文件，外观设计用来设置网络应用程序中特定控件的字体、颜色等显示信息 |
| .css | 样式表文件，用来为网络应用程序设定各种HTML TAG的字体、颜色等显示信息。 |
| .master | 母版文件，为网络应用程序设置网页的统一模板，新网页可以直接继承母版文件。 |
| .Designer.cs | 站点地图文件，用来设置网络应用程序中所有文件的树状站点地图。 |
| .ico | ico是静态图标(icon) |
| .csproj | 类库，启动程序是要选择csproj |
| csproj.filelistabsolute.txt | 文件包含在当前构建编译中和之前构建编译中构建的文件列表,并在 Clean 和 Rebuild 期间使用它来计算要删除的文件。 |
| .edmx |  |

# 初级篇

## 数据类型

C# 中的类型有两种：值类型 (value type) 和引用类型 (reference type)。前面带S的是符号类型，带U的是无符号类型。

|  |  |  |
| --- | --- | --- |
| **类别** | | **说明** |
| 值类型 | 简单类型 | 有符号整型：sbyte、short、int、long |
| 无符号整型：byte、ushort、uint、ulong |
| Unicode 字符：char |
| IEEE 浮点：float、double |
| 高精度小数型：decimal |
| 布尔：bool |
| 枚举类型 | enum E {...} 形式的用户定义的类型 |
| 结构类型 | struct S {...} 形式的用户定义的类型 |
| 可以为null的类型 | 其他所有具有 null 值的值类型的扩展 |
| 引用类型 | 类类型 | 所有其他类型的最终基类：object |
| Unicode 字符串：string |
| class C {...} 形式的用户定义的类型 |
| 接口类型 | interface I {...} 形式的用户定义的类型 |
| 数组类型 | 一维和多维数组，例如 int[] 和 int[,] |
| 委托类型 | delegate int D(...) 形式的用户定义的类型 |

### Ｃ＃有五大数据类型

#### 类(Class)

Class关键字声明类类型。Window，Form，Console,String

#### 结构体(Structures)

Struct关键字声明结构体类型。Int32,Int64,Single,Double

#### 枚举(Ebumerations)

Ebum关键字声明枚举类型。HorizontalAlignment,Visibility

#### 接口(Interfaces)

#### 委托(Delegates)

#### 派生体系

### Var

Var关键字定义的变量没有初始类型，可以接受任何数据类型。但是在接收到数据的同时，就决定了var声明的变量的类型。var 在运行之前类型就确定了。

### Dynamic

dynamic得在编译时才能确定具体类型。

### 值类型

值类型是没有实例的。值类型只需要一段单独的内存，用于存储实际的数据。

值类型的变量直接包含它们的数据，对于值类型，每个变量都有它们自己的数据副本（除 ref 和 out 参数变量外），因此对一个变量的操作不可能影响另一个变量。值类型（value type）：byte，short，int，long，float，double，decimal，char，bool 和 struct 统称为值类型。值类型变量声明后，不管是否已经赋值，编译器为其分配内存。

### 引用类型

引用类型存储总结：

引用类型的对象存储在堆中，并且会分配一个内存地址。该内存地址会存储到栈空间，栈空间名为变量名。

即读取引用类型对象的顺序：变量—>内存地址—>实例对象。

引用类型变量里存储的数据是对象的内存地址。引用类型所有需要给它分配堆上的内存。如果没有进行初始化，就是一个空引用。引用类型需要两段内存，第一段存储实际的数据，它总是位于堆中。第二段是一个引用，指向数据在堆中的存放位置。而引用类型的变量存储对它们的数据的引用，后者称为对象。对于引用类型，两个变量可能引用同一个对象，因此对一个变量的操作可能影响另一个变量所引用的对象。引用类型（reference type）：string 和 class统称为引用类型。当声明一个类时，只在栈中分配一小片内存用于容纳一个地址，而此时并没有为其分配堆上的内存空间。当使用 new 创建一个类的实例时，分配堆上的空间，并把堆上空间的地址保存到栈上分配的小片空间中。

### 值类型和引用类型的区别

**值类型存储图解;引用类型-字符串存储图解;引用类型-自定义类存储图解;**

值类型和引用类型在内存上存储的地方不一样。值类型的值是存储在内存的栈当中。引用类型的值是存储在内存的堆中。

在传递值类型和传递引用类型的时候，传递方式不一样。值类型我们称之为值传递，引用类型我们称之为引用传递。值类型的变量直接存储数据，而引用类型的变量持有的是数据的引用，数据存储在数据堆中。

值类型的实例通常是在线程栈上分配的（静态分配），但是在某些情形下可以存储在堆中。引用类型的对象总是在进程堆中分配（动态分配）

#### 相同点

引用类型可以实现接口，值类型当中的结构体也可以实现接口；

引用类型和值类型都继承自System.Object类。

#### 不同点

1）范围方面

C#的值类型包括：结构体（数值类型、bool型、用户定义的结构体），枚举，可空类型。

C#的引用类型包括：数组，用户定义的类、接口、委托，object，字符串。

2）内存分配方面

数组的元素不管是引用类型还是值类型，都存储在托管堆上。

引用类型在栈中存储一个引用，其实际的存储位置位于托管堆。简称引用类型部署在托管推上。而值类型总是分配在它声明的地方：作为字段时，跟随其所属的变量（实 例）存储；作为局部变量时，存储在栈上。（栈的内存是自动释放的，堆内存是.NET中会由GC来自动释放。

3）适用场合

值类型在内存管理方面具有更好的效率，并且不支持多态，适合用做存储数据的载体；引用类型支持多态，适合用于定义应用程序的行为。

引用类型可以派生出新的类型，而值类型不能，因为所有的值类型都是密封（seal）的；

引用类型可以包含null值，值类型不能（可空类型功能允许将 null 赋给值类型，如 int? a = null; ）；

引用类型变量的赋值只复制对对象的引用，而不复制对象本身。而将一个值类型变量赋给另一个值类型变量时，将复制包含的值。

### Ref

使用ref修饰符显式指出=此方法的副作用是改变实际参数的值。方法内部的局部参数引用了方法外部实例参数的值和指向同一个引用地址，并且没有生成变量副本。

### out

## 显示类型

## 隐式类型

Var person= new{Name=”Mr”,Age=34};Psrson.getType().Name

## 操作符

操作符是函数（即算法）的简记法。操作符不能脱离与它关联的数据类型。

|  |  |
| --- | --- |
| 类别 | 运算符 |
| 基本 | x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate |
| 一元 | +x、-x、x、~x、++x、--x、^x、(T)x、await、&&x、\*x、true 和 false |
| 乘法 | \* / % |
| 加减 | + - |
| 移位 | << >> |
| 关系和类型检测 | < > <= >= is as |
| 相等 | == != |
| 逻辑 AND | & |
| 逻辑 XOR | ^ |
| 逻辑 OR | | |
| 条件 AND | && |
| 条件 OR | || |
| null 合并 | ?? |
| 条件 | ?: |
| 赋值和 lambda 表达式 | =、\*=、/=、%=、+=、-=、<<=、>>=、&=、^=、|=、=> |

### New

#### New操作符

主要是在内存当中创建类型的实例，并调用类的实例构造器，如果在类的左边有赋值符号（=）把内存地址交给负责访问的变量。还可以调用实例的初始化器，通过花括号可以为实例属性赋值。Form form =new Form(){};但是没有使用变量引用时，例如：new Form().showDialog();在执行后，垃圾回收器会自动回收，并且在后面的语句都无法在调用这个实例。

new 类名().对象成员，这种方式是在通过new关键字创建实例对象的同时就访问了对象的某个成员，并且在创建后只能访问其中某一个成员，而不能像对象引用那样可以访问多个对象成员。同时，由于没有对象引用的存在，在完成某一个对象成员的访问后，该对象就会变成垃圾对象。所以，在实际开发中，创建实例对象时多数会使用对象引用。

#### new修饰符

例如：new public void Name(){};这个时候new是修饰符，写在方法前面是派生类对基类的方法进行隐藏。

### 移位运算

|  |  |
| --- | --- |
| & | 如果同时存在于两个操作数中，二进制AND运算符复制一位到结果中。 |
| | | 如果存在于任一操作数中，二进制OR运算符复制一位到结果中。 |
| ^异或运算符 | 如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。 |
| ~ | 按位取反运算符是一元运算符，具有"翻转"位效果，即0变成1，1变成0，包括符号位。 |
| <<二进制左移运算符 | 左操作数的值向左移动右操作数指定的位数。 |
| >>二进制右移运算符 | 左操作数的值向右移动右操作数指定的位数。 |

### 算术运算符

|  |  |
| --- | --- |
| + | 把两个操作数相加 |
| - | 从第一个操作数中减去第二个操作数 |
| \* | 把两个操作数相乘 |
| / | 分子除以分母 |
| % | 取模运算符，整除后的余数 |
| ++ | 自增运算符，整数值增加1，先赋值，在自增 |
| -- | 自减运算符，整数值减少1，先赋值，在自减 |

### 赋值运算符

|  |  |
| --- | --- |
| =赋值运算符 | 把右边操作数的值赋给左边操作数 |
| +=加且赋值运算符 | 把右边操作数加上左边操作数的结果赋值给左边操作数 |
| -=减且赋值运算符 | 把左边操作数减去右边操作数的结果赋值给左边操作数 |
| \*=乘且赋值运算符 | 把右边操作数乘以左边操作数的结果赋值给左边操作数 |
| /=除且赋值运算符 | 把左边操作数除以右边操作数的结果赋值给左边操作数 |
| %=求模且赋值运算符 | 求两个操作数的模赋值给左边操作数 |
| <<= | 左移且赋值运算符 |
| >>= | 右移且赋值运算符 |
| &= | 按位与且赋值运算符 |
| ^= | 按位异或且赋值运算符 |
| |= | 按位或且赋值运算符 |

### 关系运算符

|  |  |
| --- | --- |
| == | 两个操作数相等则条件为真。 |
| != | 两个操作数不相等则条件为真。 |
| > | 左操作数大于右操作数的值则条件为真。 |
| < | 左操作数的值小于右操作数的值则条件为真。 |
| >= | 左操作数的值大于或等于右操作数的值则条件为真。 |
| <= | 左操作数的值小于或等于右操作数的值则条件为真。 |

### 逻辑运算符

|  |  |
| --- | --- |
| &&逻辑与运算符 | 如果两个操作数都非零，则条件为真。 |
| ||逻辑或运算符 | 如果两个操作数中有任意一个非零，则条件为真。 |
| ! 逻辑非运算符 | 用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。 |

### 其他运算符

|  |  |
| --- | --- |
| sizeof() | 返回数据类型的大小。 |
| typeof() | 返回 class 的类型。 |
| & | 返回变量的地址。 |
| \* | 变量的指针。 |
| ? : | 条件表达式 |
| is | 判断对象是否为某一类型。 |
| as | 强制转换，即使转换失败也不会抛出异常。 |

### 运算符重载

## 语句

### 迭代语句

|  |  |
| --- | --- |
| while | 只要条件为真，就会不断执行代码。 |
| do while | do...while 循环是在循环的尾部检查它的条件。相关的代码块是在对条件语句求值前执行，会确保至少执行一次循环。 |
| For | 用于只要满足指定条件就执行循环的情况。For由三个部分组成：一个初始值，一个条件表达式，一个增值或者减值表达式。 |
| for in | 迭代循环集合的值 |
| Foreach | 用来迭代数组或集合。 |
| Foreach in | 用来迭代数组或集合的值 |

### 选择语句

|  |  |
| --- | --- |
| if else | 以一种互斥的方式对一个或多个条件语句求值 |
| switch case: default | 一个 switch 语句允许测试一个变量等于多个值时的情况。每个值称为一个 case，且被测试的变量会对每个 switch case 进行检查 |
| 三元表达式 |  |

### 跳转语句p

|  |  |
| --- | --- |
| break | 终止并跳出循环体。 |
| continue | 终止当前循环，重新开始一个新的循环。 |
| goto | 跳转到指定另一段代码位置。 |
| return | 跳出循环及其包含的函数。Return可以退出一个返回值非void的方法。如果方法在定义中指定了返回类型，在return后面必须紧跟该类型的值。 |
| Throw new | 抛出一个异常。 |

### 块语句

块语句用于在只允许使用单个语句的上下文中编写多条语句。

## 表达式

表达式是一种专门求值的表达式实体(方法)。

操作数1个或多个操作符0个或多个。算法逻辑的最基本单元，表达一定的算法意图。

因为操作符有优先级，所以表达式也有优先级。

### 成员访问表达式

Action action = new Action[Console.WriteLine];

模仿Ｃ和Ｃ＋＋的函数指针

### Lvakue

### Rvalue

## 变量

变量一个有7种：静态变量，实例变量（成员变量，字段），数组元素，值参数，引用参数，输出形参，局部变量，变量的值是允许改变的，根据程序运行调用时，决定变量的值。

变量=以变量名所对应的内存地址为起点，以其数据类型所要求的存储空间为长度的一块区域。

### 成员修饰符

在修饰字段时通常用两个修饰符，即readonly （只读）、static （静态的）和const （常量），三者不能修饰同一个成员。使用 readonly 修饰字段意味着只能读取该字段的值而不能给字段赋值。使用 static 修饰的字段是静态字段，可以直接通过类名访问该字段。

### 成员访问修饰符

|  |  |
| --- | --- |
| public | 成员可以被任何代码访问 |
| private | 私有访问。只限于本类成员访问，派生类，实例都不能访问。如果在类成员前未使用任何访问修饰 符，则默认为private。 |
| protected | 保护访问。只限于本类和派生类访问，实例不能访问。 |
| internal | 内部访问。只限于本项目内访问，其他不能访问。 |
| protected internal | 内部保护访问。只限于本项目或是派生类访问。 |

### 成员变量

成员变量：在类中定义,成员变量就是类中的属性。当new对象的时候，每个对象都有一份属性。一个对象中的属性就是成员变量。作用范围：在类内部任何地方都可以访问成员变量。

声明周期：出生，new对象的时候，开辟内存空间。死亡，堆内存地址没有引用，变成垃圾，被垃圾回收器回收。

### 局部变量

局部变量：在方法体中定义的变量，其使用范围只局限在定义它的方法之中或者方法之中的循环体之内，作用域不超过一个大括号也就是块语句。

局部变量：方法的形式参数以及在方法中定义的变量。

作用范围：在方法体的任何位置都可以访问。

生命周期：出生：运行到创建变量的语句时。死亡：超过了其作用范围。

### 类变量（静态属性\_全局变量）

在类中定义的变量用static修饰的变量为全局变量，在整个namespace中的所有类中都可以使用。

## 常量

### 静态常量(Const[编译时常量])

const为编译时常量，程序编译时将对常量值进行解析，并将所有常量引用替换为相应值。

定义：声明的同时要设置常量值。类型限制：只能修饰基元类型，枚举类型或者字符串类型。

对于类对象而言：对于所有类的对象而言，常量的值是一样的。

编译时常量用const关键字定义，例如 public const double pi=3.14159。注意点：

1、编译时常量必须在定义时赋值

2、除string外的引用类型（包括可空类型）只能定义编译时常量为null，相当于编译时常量不可通过new进行初始化

3、结构类型不能定义为常量，例如DateTime类型不可定义编译时常量。

4、编译时常量一旦赋值就不能再改变其值。

编译时：我认为有两个情况。第一：当你在写代码时，编译器自动检查你发生的错误。第二：当你写完代码，运行时出现了许多错误（可执行程序的窗口还未弹出，即你写的代码还没有分配内存）。

### 动态常量(Readonly[运行时常量])

readonly为运行时常量，程序运行时进行赋值，赋值完成后便无法更改，因此也有人称其为只读变量。

定义：声明的时候可以不需要进行设置常量值，可以在类的构造函数中进行设置。

类型限制：类型没有限制，可以用它定义任何类型的常量。

对于类对象而言：常量的值可以是不一样的。

运行是常量用readonly定义，它是在程序加载到内存中运行后才确定其值的。其值可以通过以下几种方式赋值：1、定义时赋值2、类的构造函数内赋值。

运行时：运行程序时，可执行程序的窗口已经弹出（黑窗口），但是这时却出现了bug（系统已经为你的代码分配了存储空间）

# 中级篇

## 类的封装

类是一种数据结构，它可以包含数据成员（常量和字段）、函数成员（方法、属性、事件、索引器、运算符、实例构造函数、静态构造函数和析构函数）以及嵌套类型。类类型支持继承，继承是一种机制，它使派生类可以对基类进行扩展和专用化。

类是最基础的C#类型。类是一个数据结构，将状态（字段）和操作（方法和其他函数成员）组合在一个单元中。类为动态创建的类实例（instance）提供了定义，实例也称为对象（object）。类支持继承（inheritance）和多态（polymorphism），这是一个派生类（derived class）可用来扩展和专用化基类（base class）的机制。

### 类概念

**类是一种数据结构（data structure），是一种抽象数据结构。类本身是一种抽象的结果（概念），**

**比如现实世界当中有学生，从中抽取一些数据（ID，**

**name），把它们封装在一起，形成学生类，而且学生类里面有学生的成员。**

**类本身还是抽象结果的载体。类本身还是抽象数据和抽象行为的载体。抽象行为：考试，行为等，考试，学习是学生的行为，抽象出考试和学习的方法后，封装到类里面成为类的方法成员。**

**类是面向对象核心的概念，也承载了抽象编程的方法，比如说多态和继承。所以类在面向对象编程中是起到枢纽和支点作用的数据结构；**

**类是一种引用类型，具体到每一个类上，每一个类都是一个自定义类型。（每一个类都是自定义的引用类型），可以拿类去声明变量，拿这个类去创建实例，当拿这个类去创建实例的时候，这个类又体现出另一种功能，类可以被视为实例的模版，拿这个类像模版一样创建一个个实例；**

**类代表现实世界中的＂种类＂（概念），程序当中的这个类，在一定程度上代表现实当中的某些种类。（这个时候程序当中类的概念与数学和哲学类的概念最接近）**

### 访问修饰符

|  |  |
| --- | --- |
| 可访问性 | 含义 |
| public | 访问不受限制 |
| protected | 访问仅限于此类或从此类派生的类 |
| internal | 访问仅限于此程序 |
| protected internal | 访问仅限于此程序或从此类派生的类 |
| private | 访问仅限于此类 |
| private protected | 该类型或成员可以通过从派生的类型访问，这些类型在其包含程序集中进行声明。 |

### 类声明（Class declarations）

声明即定义（C#和ava），可以把类声明在名称空间里，也可以把类声明在所有显示名称空间之外（全局名称空间不建议写），或者把类当中成员（嵌套类）来声明。

### 类的成员

|  |  |
| --- | --- |
| 成员 | 说明 |
| 常量 | 与类相关联的常量值 |
| 字段 | 类的变量 |
| 方法 | 类可执行的计算和操作 |
| 属性 | 与读写类的命名属性相关联的操作 |
| 索引器 | 与以数组方式索引类的实例相关联的操作 |
| 事件 | 可由类生成的通知 |
| 运算符 | 类所支持的转换和表达式运算符 |
| 构造函数 | 初始化类的实例或类本身所需操作 |
| 析构函数 | 在永久丢弃类的实例之前执行操作 |
| 类型 | 类所声明的嵌套类型 |
| 局部变量 | 方法的成员 |

### 静态成员和实例成员的区别

静态（Static）成员在语义上表示它是类的成员，实例（非静态）成员在语义上表示它是对象的成员，绑定（Binding）指的是编译器如果把一个成员与类或对象关联起来，※”.”操作符-成员访问

静态成员也称为共享成员，例如静态属性 静态字段 静态方法；静态成员可以在类的实例之间共享。静态类中只能有静态成员，不能有实例成员，因为静态类不能进行实例化；在非静态类中 即可以有静态成员 也可以有非静态成员；只要是静态成员 属性 字段 方法，都需要用类名去调用；静态成员和实例成员的区别：静态成员需要通过static修饰，非静态成员不需要通过static修饰；静态成员属于类，可以直接通过“类名.静态成员”的方式访问；非静态成员，必须实例化对象后，通过对象.实例成员来访问;当类第一次加载的时候(第一次被加载到内存的时候),该类下的所有的静态成员都会被加载，实例成员有多少个对象，就会创建多少个对象；静态成员会被加载到静态存储区，直到程序退出时才会被释放；

静态成员包括静态字段和静态属性，静态成员和类相关联，不依赖于对象而存在，只能由类访问，而不能由对象访问；静态成员属于类所有，无认创建多少实例对象，静态成员在内存中只有一份；实例成员属于类的实例，一个类可以有无数实例，每个实例都有自己的内存。每创建一个实例对象，实例成员都会在内存中分配一块内存区域。所以静态成员一般用于存放共享的数据段，如数据库连接字符串等。

一个类如果只包含静态成员和静态方法，则该类可以定义为静态类，给类加上static修饰符；

### 字段

字段是一种表示与对象或类型（类与结构体）关联的变量。字段是类的成员，旧称成员变量

与对象关联的字段亦称实例字段，字段是与类或类的实例关联的变量。

使用 static 修饰符声明的字段定义了一个静态字段 (static field)。一个静态字段只标识一个存储位置。无论对一个类创建多少个实例，它的静态字段永远都只有一个副本。

不使用 static 修饰符声明的字段定义了一个实例字段 (instance field)。类的每个实例都为该类的所有实例字段包含一个单独副本。无显示初始化时，字段获得其类型的默认值，所有字段永远不会被初始化。

只读字段是为实例或类型保存一旦初始化，就不希望改变的值。

#### 字段的初始值

无显示初始化时，字段获得其类型的默认值，所有字段永远都不会未被初始化。

实例字段初始化的实际=对象创建时；静态字段初始化的时机=类型被加载时；

#### 字段修饰符

|  |  |
| --- | --- |
| 静态修饰符 | static |
| 访问修饰符 | public internal private protected |
| 继承修饰符 | new |
| 继承修饰符 | unsafe |
| 只读修饰符 | readonly |
| 线程修饰符 | volatile |
| public | 访问不受限制 |
| protected | 访问仅限于此类或从此类派生的类 |
| internal | 访问仅限于此程序 |
| protected internal | 访问仅限于此程序或从此类派生的类 |
| private | 访问仅限于此类 |
| private protected | 该类型或成员可以通过从派生的类型访问，这些类型在其包含程序集中进行声明。 |

#### 实例字段

实例字段在内存在创建实例时动态分配，先把类实例化new一个对象来访问（对象.字段）是实例字段

一般用来类内部进行访问，充当一种类中的"全局变量"角色；字段类似于变量，或者配合属性来使用。

实例字段初始化时机－对象创建时；

#### 实例只读字段

Readonly修饰过的字段只能进行一次初始化，在构造器里，初始化后就不能在赋值了。

#### 静态字段

对于静态字段，其内存在类型对象创建时分配。静态字段初始化时机－类型被加载时。

由 readonly （只读）修饰的字段只能在定义时或构造函数中赋值。只读字段初始化后不能在赋值，非只读字段可以在初始化后在赋值。静态字段需要用类名来访问（类名.字段）

#### 静态只读字段

静态只读字段只能进行初始化，之后就不可以在赋值。

### 属性

属性是类的成员，是字段的自然扩展。属性本质是一个方法，其实就是外界访问私有字段的入口，属性本身不保存任何数据，在对属性赋值和读取的时候其实就是操作的对应私有字段。同样是C#类级别定义的，一般是供外部类访问的。属性：对外暴露数据，数据可以存储在字段里，也可以时动态计算出来的。

对内：保护字段不被非法值污染。

#### 属性构造器

#### 静态属性

当属性声明包含 static 修饰符时，称该属性为静态属性。

#### 实例属性

当不存在 static 修饰符时，称该属性为实例属性。属性通过类的实例化对象.属性名来调用

#### 访问器

|  |  |
| --- | --- |
| Set | set是设置属性时进行的操作 |
| Get | get是读取属性时进行的操作，必须要有return返回值。 |

#### 自动实现属性

属性生成默认字段。

#### 属性与字段的关系

### 索引器

索引器（indexer）是这样一种成员：它是对象能够用与数组相同相同的方式（用下标）进行索引。

### 构造函数(构造器)

构造函数没有返回值（void也不需要）

一个类中可以有多个构造函数，可根据其参数个数，参数顺序，参数类型的不同来区分它们，这称作构造函数的重载，构造函数的命名必须和类名相同，调用派生类的构造函数时会先调用基类的无参构造函数

通常使用构造函数来进行类中字段的赋初值。构造函数不能直接被调用，必须通过关键字new在创建对象时自动调用

当一个类没有定义任何构造函数时，C#编译器会自动为其生成一个默认的隐式无参构造函数

如果一个类中定义了有参构造函数，实例化时需要调用无参构造函数，这时我们就必须定义一个显式无参数构造函数，否则运行会报错

#### 重载构造函数

类或结构可能重载构造函数。为了避免代码重复，一个构造函数可以使用this关键字调用另一个构造函数，当一个构造函数调用另一个时，被调用的构造函数首先执行。

#### 实例构造函数

构造函数的名字与类名相同，使用new表达式创建类的对象时，会调用其构造函数。并且通常初始化新对象的数据成员。除非类是静态的，否则会为没有构造函数的类，自动生成一个默认构造函数，并使用默认值来初始化对象字段。构造函数可以有参数，可以以参数个数、参数类型、参数顺序不同的形式存在多个构造函数。

#### 静态构造函数

静态构造函数对每个类执行一次，而不是每个实例执行一次。

类只能定义一个静态构造函数，它必须是无参数的，并且具有与类相同的名称。

运行时在所使用的类之前自动调用静态构造函数。静态构造函数允许的唯一修饰符是unsafe和extern。

静态字段初始化器在调用静态构造函数之前运行。静态构造函数不使用访问修饰符或不具有参数。

在创建第一个实例或引用任何静态成员之前，将自动调用静态构造函数以初始化类。

不能直接调用静态构造函数。无法控制在程序中执行静态构造函数的时间。

静态构造函数的一种典型用法是在类使用日志文件且将构造函数用于将条目写入到此文件中时使用。

静态构造函数对于创建非托管代码的包装类也非常有用，这种情况下构造函数可调用 LoadLibrary 方法。

如果静态构造函数引发异常，运行时将不会再次调用该函数，并且类型在程序运行所在的应用程序域的生存期内将保持未初始化。类或结构只能有一个静态构造函数。静态构造函数不能继承或重载。

#### 构造函数与一般函数的区别

1.声明方式不同构造函数：[访问修饰符] 类名(参数列表){函数主体}

一般函数：[访问修饰符] 返回类型 方法名(参数列表){函数主体}

2.调用方式不同

构造函数：在类实例化时调用、一般函数：在类实例化后，必须手动调用

#### 私有构造函数

将构造函数申明为私有的，则不能通过new运算符在外部代码中实例化（但可以编写一个静态方法或属性在其内部实例化一个对象，再将结果返回给外部代码）。

私有构造函数的作用：永远不会实例化，因为它仅用作某些静态成员的容器

希望类只能通过调用某个静态方法来实例化（即所谓的对象实例化的类工厂方法）

#### 调用类的静态函数时的执行顺序

静态变量==>静态构造函数==>实例构造函数

静态变量==>静态构造函数==>静态函数

#### 调用类的实例函数时的执行顺序则

非静态变量==>实例构造函数==>实例函数

### 静态类

静态类无法实例化。静态类的所有成员必须是静态的。

静态类可以包含静态变量，静态方法，静态属性，静态运算符，静态事件和静态构造函数。

静态类不能包含实例成员和实例构造函数。索引器和析构器不能是静态的

var不能用于定义静态成员。您必须在static关键字之后显式指定成员类型。

静态类是密封类，因此不能继承。静态类不能从其他类继承。

可以使用 ClassName.MemberName（即：类名.成员名）访问静态类成员。

静态类在程序所在的应用程序域的整个生命周期内都将保留在内存中。

#### 静态类使用场景

判断这个很简单,就是从内存的优化方面去考虑.因为静态和非静态的不同的地方,就是静态的从程序一启动就会一直占用内存,而非静态的只在使用后(实例化)后才会占用内存.但是每实例化个一个对象时又会另外占用内存. 举个例子,比如说一个数据库的连接字段(STRING).因为要经常使用到它,这时我们可以用STATIC.但是如果这时用非静态的话那就不合算了,因为每次调用到它时,又实例化一次.这样相比来说占用内存就比较大了.不划算.、像一个登录后台的方法,你只在登陆时候调用一次,就没有必要做成静态的了.那样一直驻存在内存中.在大型项目中,你如果都使用静态的那得要多少内存去支撑呀.就是它经常要被调用时,就用静态的.相反则用非静态的。那也就是说，公共方法就设置为静态的方法。果然是的，看经常用的comm类就是静态的Static

### 静态构造函数

1：静态类可以有静态构造函数，静态构造函数不可继承；

2：静态构造函数可以用于静态类，也可用于非静态类；

3：静态构造函数无访问修饰符、无参数，只有一个 static 标志；

4：静态构造函数不可被直接调用，当创建类实例或引用任何静态成员之前，静态构造函数被自动执行，并且只执行一次。

### 析构函数（实例析构器－垃圾回收器）

在执行想回收的类前面把public换成波浪线。析构器在函数执行完后会自动调用。

### 部分类（partial）

减少类的派生；

## 方法（method）

方法的内存在栈上，主调用方法调用被调用方法的局部变量也归主调用方法管。如果函数名后面加上圆括号就表示立即调用（执行）这个函数里面的代码（花括号部分的代码）。

不加括号的，都是把函数名称作为函数的指针，一个函数的名称就是这个函数的指针，此时不是得到函数的结果，因为不会运行函数体代码。它只是传递了函数体所在的地址位置，在需要的时候好找到函数体去执行。

### 访问修饰符

|  |  |
| --- | --- |
| public | 公有访问。不受任何限制。 |
| private | 私有访问。只限于本类成员访问，派生类，实例都不能访问。 |
| protected | 保护访问。只限于本类和派生类访问，实例不能访问。 |
| internal | 内部访问。只限于本项目内访问，其他不能访问。 |
| protected internal | 内部保护访问。只限于本项目或是派生类访问，其他不能访问 |

### 结构体

结构体是值类型的数据结构，它使得一个单一变量可以存储各种数据类型的相关数据。结构体用来代表一个记录。结构体是用struct关键字声明的。结构是隐式密封的，无法被派生。

结构是定义数据类型，与类类似，它们有数据成员和函数成员。结构数据的变量不能为null，两个结构变量不能引用同一个变量。结构的结构体对象分配在堆上。

结构体可以使用关键字[new]来创建实例化对象。

struct 是表示数据结构的值类型数据类型。它可以包含参数化构造函数、静态构造函数、常量、字段、方法、属性、索引器、运算符、事件和嵌套类型。

在结构体中，除非字段被声明为const或static，否则是无法初始化的。结构体不能声明默认构造函数（没有参数的构造函数）或析构函数。在visual studio中要设置允许不安全代码执行才不会报错。アンーセーフコードの許可(允许安全代码)

#### 结构和类的区别

结构是值类型，它在栈中分配空间；而类是引用类型，它在堆中分配空间，栈中保存的只是引用。

结构类型直接存储成员数据，让其他类的数据位于对中，位于栈中的变量保存的是指向堆中数据对象的引用。

当实例化结构体的时候是在栈上拷贝了结构体的一个副本，而类实例化的时候是在栈上分配一个指针指向了堆中分配的空间。

#### 结构体使用场景

### 静态方法

用static修饰的是静态方法，静态方法不对特定实例进行操作，静态方法只能访问类中的静态成员，而不能使用实例成员。访问静态方法只能使用类名，而不需要创建对象，也不能使用对象名来引用。

静态方法：静态方法是不属于特定对象的方法，静态方法可以访问静态成员变量，静态方法不可以直接访问实例变量，可以在实例函数调用的情况下，实例变 量做为参数传给静态方法。静态方法也不能直接调用实例方法，可以间接调用，首先要创建一个类的实例，然后通过这一特定对象来调用静态方法。静态方法只能调用其他静态方法并访问静态成员。您不能在静态方法中访问类的非静态成员

#### 静态方法使用场景

判断这个很简单,就是从内存的优化方面去考虑.因为静态和非静态的不同的地方,就是静态的从程序一启动就会一直占用内存,而非静态的只在使用后(实例化)后才会占用内存.但是每实例化个一个对象时又会另外占用内存. 举个例子,比如说一个数据库的连接字段(STRING).因为要经常使用到它,这时我们可以用STATIC.但是如果这时用非静态的话那就不合算了,因为每次调用到它时,又实例化一次.这样相比来说占用内存就比较大了.不划算.、像一个登录后台的方法,你只在登陆时候调用一次,就没有必要做成静态的了.那样一直驻存在内存中.在大型项目中,你如果都使用静态的那得要多少内存去支撑呀.简单点,就是它经常要被调用时,就用静态的.相反则用非静态的。

### 实例方法

没有使用static修饰符的是实例方法。实例方法可以使用类的任何成员。调用实例方法时，必须使用类的实例或对象来引用。实例方法对类的某个给定的实例进行操作，在实例方法类中可以使用this来访问该实例。调用实例方法时，必须先创建一个对象。一个实例方法的执行与特定对象关联，他的执行需要一个对象存在。实例方法可以直接访问静态变量和实例变量，实例方法可以直接访问实例方 法、和静态方法，静态方法的访问方式为类名加点操作符加变量名。当多个实例对象存在时，内存中并不是存在美个特定的实例方法的拷贝，而是，相同类的所有对 象都共享每个实例方法的一个拷贝（实例方法只占用“一套”空间）。

### 方法重载（Overload）

#### 重载条件

方法签名相同，在同一个类中，方法的参数列表(参数类型、参数个数、参数顺序)不同

方法签名相同，参数列表相同，但返回值类型不同的方法不能构成方法重载。

#### 重载作用

重载是对继承的一种很好的补充，重载是为同一功能的方法提供不同的接口

重载可以大大减少代码的输入量，同一个方法名只要往里面传递不同的参数就可以拥有不同的功能或返回值

重载增强程序的逻辑性、可读性。

### 方法参数

#### 值参数

值参数也叫传值参数，值参数在声明时不带任何修饰符的参数。值参数作用域为当前方法的局部变量，初始值就是在调用这个方法时，传给它的实参的值，值参数相当于一个新声明的局部变量，或者理解为传进来的时参的一个副本。声明时不带修饰符的形参是值形参。一个值形参对应于一个局部变量，只是它的初始值来自该方法调用所提供的相应实参。当形参是值形参时，方法调用中的对应实参必须是表达式，并且它的类型可以隐式转换为形参的类型。

允许方法将新值赋给值参数。这样的赋值只影响由该值形参表示的局部存储位置，而不会影响在方法调用时由调用方给出的实参。

#### 引用参数(ref)

引用形参是用 ref 修饰符声明的形参。与值形参不同，引用形参并不创建新的存储位置。相反，引用形参表示的存储位置恰是在方法调用中作为实参给出的那个变量所表示的存储位置。表示值形参不但引用了相应实参的值，还引用了同一个内存地址，没有生成副本，一旦修改，相应实参约会改变。

引用参数并不创建变量的副本；使用ref修饰符显示指出-此方法的副作用是改变实际参数的值。

#### 输出参数(out)

输出参数并不创建变量的副本；方法体内必需有对输出变量的赋值操作；使用out修饰符显示指出-此方法的副作用是通过参数向外输出；从语义上来讲-ref是为了改变，out是为了输出。

#### 形参数组(params)

**必需是形参列表中的最后一个，由params修饰。**

#### 具名参数

调用方法时，传进去的参数是有名字的。优点是可读性加强，传的参数不在受位置约束。

PrintInfo(name:值,age:值);

#### 扩展方法（this）

**扩展方法必须是公有，静态的，即被public static所修饰；**

**必须是形参列表中的第一个，由this修饰。**

**必须由一个静态类（一般类名为SomeTypeExtension）来统一收纳对SomeType类型的的扩展方法。**

### 方法的调用

#### 本类

#### 非本类

### 枚举

限定用户只能从集合中选有效值。用enum关键字定义枚举类型。

### 抽象方法

在抽象类中没有方法体的方法就是抽象方法（方法签名）；派生类在实现抽象类的抽象方法时，必须加上关键字（override），不实现也要写成虚方法。抽象类中的抽象方法必须不是public。

#### 抽象方法的概念

在抽象类中也可以使用关键字absract定义抽象方法，要求所有的派生非抽象类都要重写实现抽象方法，引入抽象方法的原因在于抽象类本身是一种抽象概念，有的方法并不需要具体实现，而是留下来让派生类重写实现。抽象类中也可以声明正常方法。也就是说抽象方法不能有方法体，抽象类中的所有抽象方法必须在子类中重写override。

#### 抽象方法的特性

抽象方法是隐式的virtual方法(虚方法)。只允许在抽象类中使用抽象方法声明。

因为抽象方法声明不提供实实现，所以没有方法体；方法声明只是以一个分号结束，并且在签名后没有大括号 ({ })。实现由overriding方法提供，它是非抽象类的成员。

在抽象方法声明中使用static或virtual修饰符是错误的。

除了在声明和调用语法上不同外，抽象属性的行为与抽象方法一样。在静态属性上使用abstract修饰符是错误的。在派生类中，使用override修饰符的属性声明可以重写抽象类的继承属性。

### Lambda表达式

### Partial方法

## 字符串

### 数据类型转换

### 隐式类型转换

### 强制类型转换

### 装箱和拆箱

装箱指的是把栈上的值类型的值封装成obejct的实例，放在堆上。

拆箱指的是把堆上obejct类型的实例里边的值，按照需求拆成目标数据类型，放在栈上。装箱是将值类型转换为引用类型 ；拆箱是将引用类型转换为值类型。

## 数组

数组是一个存储相同类型元素的固定大小的顺序集合。数组是用来存储数据的集合，通常认为数组是一个同一类型变量的集合。数组是引用类型，所有需要给它分配堆上的内存。但是数组元素是值类型。且值类型的元素都位于托管堆上，不在栈上。{}数组初始化器。

## 类的继承

继承的本质是派生类在基类已有的成员上横向和纵向的扩展。一个子类的实例也是基类的实例。

继承是子类在基类的基础（模版）上派生出的类，为的是对基类功能进行扩展和补充。可以使用基类类型的变量来引用子类类型的实例，继承使用：符号。子类的访问级别不能超过基类。派生类只能有一个基类，但可以实现多个接口。

### 继承的本质

横向：类成员数量扩充。派生类基于基类已有的成员：当继承发生的时候，派生类对父类的成员是全盘继承的。基类的实力构造器不会被继承。在派生继承基类过程当中，进行扩展：类的成员只会越来越多，不会越来越少。一个类成员被引入到继承链当中，那么它就会一直向下传递，不可能在从继承链中移除。

纵向：类成员的纵向扩展，（行为的改变，版本增高,也就是重写）。类成员的访问级别是以类为上限的。

派生类调用基类使用base。Sealed类不能被继承。

### 单继承

### 多继承

### 方法重写[虚方法(virtual)]

重写是子类对基类方法的实现过程进行重写，即外壳不变（方法名，返回值，形参），核心重写（处理的逻辑，实现的过程），因此重写是基于子类继承基类的基础上进行的。返回类型与被重写方法的返回类型可以不相同，但是必须是父类返回值的派生类。在C#中使用重写是需要在基类的成员里加上（virtual），再在派生类的成员里加上关键字（override），才构成子类对基类的成员的重写。并且除非是使用关键字base调用基类方法，否则子类重写的方法会覆盖基类的同名方法，也就是版本增高。

### 类成员的隐藏

在C#中基类的成员里不使用（virtual），派生类的成员里不使用字（override），但是子类又有和基类一样的类成员时，是子类成员对基类成员的隐藏。重写与隐藏的发生条件：函数成员，子类可见，签名一致。

## 类的多态

多态是派生类（派生类）对象，去重写（复写）基类（基类）方法（接口，虚方法，抽象）实现根据当前上下文对象（this.name()）去调用同一个方法（对象名可以不同，方法名一样，根据对象调用方法）。多态基于重写机制，函数成员的具体行为（版本）由对象决定，C#语言的变量和对象都是有类型的，所以会有代差。多态指允许不同的对象对同一个消息做出相应，即同一个消息可以根据发送对象的不同而采用多种不同的行为方式（发送消息就是函数调用）。封装和继承几乎都是为多态而准备的，在执行期间判断引用对象的实际类型从而调用其相应的方法。 通过传递给父类对象引用不同的子类对象从而表现出不同的行为，多态可为程序提供更好的可扩展性，同样也可以代码重用。

### 抽象类

接口和抽象类都是软件工程产物；具体类→抽象类→接口：越来越抽象，内部实现的东西越来越少，

抽象类是未完全实现逻辑的类（可以有字段和非publc成员，它们代表了具体逻辑），

抽象类为复用而生：专门作为基类来使用，也具有解耦功能，封装确定的，开放不确定的，推迟到合适的子类中去实现。它们都不能实例化，只能用来声明变量，引用具体类（concrete class）的实例。抽象方法只作声明，而不包含实现，可以看成是没有实现体的虚方法。接口是对动作的抽象，抽象类是对根源的抽象。

使用抽象类是为了代码的复用。接口是对动作的抽象，抽象类是对根源的抽象。抽象类在代码实现方面发挥作用，可以实现代码的重用。抽象类是表示抽象概念的特殊类。如果一个类不与具体的事物相联系，而只是表达一种抽象的概念，仅仅是作为派生类的一个基类，这样的类就是抽象类。抽象类使用关键字abstract来定义，抽象类不能被实例化，一个抽象类可以同时包含抽象方法和非抽象方法。abstract关键字可以和类、方法、属性、索引器及事件一起使用。一个抽象类可以包含抽象和非抽象方法，当一个类继承于抽象类，那么这个派生类必须实现所有的的基类抽象方法。

#### 开放/关闭原则

应该封装那些不变的，不变的，稳定的，，固定的成员，而把那些不确定的，有可能改变的成员声明成抽象成员，并且留给子类去实现。

#### 抽象类使用

(1) 抽象方法只作声明，而不包含实现，可以看成是没有实现体的方法签名

(2) 抽象类不能被实例化

(3) 抽象类可以但不是必须有抽象属性和抽象方法，但是一旦有了抽象方法，就一定要把这个类声明为抽象类

(4) 具体派生类必须覆盖基类的抽象方法

(5) 抽象派生类可以覆盖基类的抽象方法，也可以不覆盖。如果不覆盖，则其具体派生类必须覆盖它们

#### 抽象类的特性

抽象类不能直接实例化。并且对抽象类使用new运算符是编译时错误。虽然一些变量和值在编译时的类型可以是抽象的，但是这样的变量和值为null，或者含有对非抽象类的实例的引用（此非抽象类是从抽象类派生的）。

抽象类可以包含抽象方法和抽象访问器，但是非抽象类不可以。

不能用sealed关键字修改抽象类，如果使用了sealed关键字意味着该类不能被继承。

从抽象类派生的非抽象类必须重写所有抽象方法和抽象访问器。

#### 抽象类的作用

类可以实现无限个接口，但仅能从一个抽象（或者其他类型）类继承，从抽象类派生的类仍可实现接口，从而得出接口时用来处理多重继承问题的。

抽象类中可以存在非抽象方法，可接口不能，且接口里面的方法只是一个声明必须用public来修饰，没有具体实现方法。

抽象类中的成员变量可以被不一样的修饰符来修饰，可接口中的成员变量默认的都是静态常量(staticfinal)

抽象类是对象的抽象,接口是一种行为规范

#### 抽象类的使用场景

如果一个类设计的目点是用来被其它类继承的，它代表一类对象的所具有的公共属性或方法，那个这个类就应该设置为抽象类。

在实现接口时，常写一个抽象类，来实现接口中的某些子类中所需的通用方法，接着在编写各个子类时，即可继承该抽象类来用。省去在每一个都要实现通用的方法的困扰

### 多态的本质

多态是派生类（派生类）对象，去重写（复写）基类（基类）方法（接口，虚方法，抽象）实现根据当前上下文对象（this.name()）去调用同一个方法（对象名可以不同，方法名一样，根据对象调用方法）

### 静态多态

### 动态多态

### 向上转型

### 向下转型

## 接口

### 显示接口

### 隐式接口

### 接口方法

接口中的成员方法必须是piblic的。

## 内存

局部变量存储在内存的栈上。调用者与被调用者内存位置

被调用者被调用时存在哪归谁管，调用

### 栈

#### 栈溢出

是方法调用太深，却没有return，于是不停调用，直到把栈撑爆。

### 堆

## 反射与依赖注入

### 反射的本质

反射：以不变应万变（更松的耦合）

反射与接口的结合，反射与特性的结合。依赖注入：此DI非彼DI，但没有彼DI就没有此DI。

### 特性

## 正则表达式

## 配置文件

|  |  |
| --- | --- |
| <configuration> | 顶层配置节 |
| <configSections> | 配置节声明区域，包含配置节和命名空间声明 |
| <section> | 配置节声明 |
| <sectionGroup> | 定义配置节组 |
| <section> | 配置节组中的配置节声明 |
| <appSettings> | 预定义配置节 |
| <Custom element for configuration section> | 配置节设置区域 |

## 词法结构

### 标记

#### 逐字字符串@标识符

忽略转义字符，让字符串跨行，在标识符中的用法，C#是不允许关键字作为标识符(类名、变量名、方法名、表空间名等)使用的，但如果加上@之后就可以了。

#### |过滤器

#### $字符串内插

$是String.format()的简写，可以写$”{字符串变量}”,代码简洁，提高可读性。

### 预处理指令

预处理指令提供按条件跳过源文件中的节、报告错误和警告条件，以及描绘源代码的不同区域的能力。

下面是可用的预处理指令：

#### 可为空上下文

|  |  |
| --- | --- |
| #nullable disable | 将可为空注释和警告上下文设置为“已禁用”。 |
| #nullable enable | 将可为空注释和警告上下文设置为“已启用”。 |
| #nullable restore | 将可为空注释和警告上下文还原为项目设置。 |
| #nullable disable annotations | 将可为空注释上下文设置为“已禁用”。 |
| #nullable enable annotations | 将可为空注释上下文设置为“已启用”。 |
| #nullable restore annotations | 将可为空注释上下文还原为项目设置。 |
| #nullable disable warnings | 将可为空警告上下文设置为“已禁用”。 |
| #nullable enable warnings | 将可为空警告上下文设置为“已启用”。 |
| #nullable restore warnings | 将可为空警告上下文还原为项目设置。 |

#### 条件编译

|  |  |
| --- | --- |
| #if | 打开条件编译，其中仅在定义了指定的符号时才会编译代码。 |
| #elif | 关闭前面的条件编译，并基于是否定义了指定的符号打开一个新的条件编译。 |
| #else | 关闭前面的条件编译，如果没有定义前面指定的符号，打开一个新的条件编译。 |
| #endif | 关闭前面的条件编译。 |

#### 诊断区域指令

可以使用以下两个预处理器指令来定义可在大纲中折叠的代码区域

|  |  |
| --- | --- |
| #region | 启动区域。 |
| #endregion | 结束区域。 |

#### 错误和警告信息

用以下指令指示编译器生成用户定义的编译器错误和警告，并控制行信息

|  |  |
| --- | --- |
| #error | 使用指定的消息生成编译器错误。 |
| #warning | 使用指定的消息生成编译器警告。 |
| #line | 更改用编译器消息输出的行号。 |

#### Pragma 指令

#pragma 为编译器给出特殊指令以编译它所在的文件。 这些指令必须受编译器支持。 换句话说，不能使用 #pragma 创建自定义的预处理指令。

|  |  |
| --- | --- |
| #pragma warning | 启用或禁用警告。 |
| #pragma checksum | 生成校验和。 |

## 集合

## 委托(delegate)

委托（delegate）是函数指针的升级版；实例-c/c++中的函数指针；函数指针，(函数指针只能调用静态方法)

委托是函数指针的升级版，所有委托直接调用的是地址。普通方法⇒调用委托方法

一切皆地址-变量（数据）是以某个地址为起点的一段内存中所存储的值；

函数（算法）是以某个地址为起点的一段内存中所存储的一组机器语言指令；

委托是一种类（class），类是数据类型所有委托也是一种数据类型；它的声明方式与一般的类不同，主要是为了照顾可读性和c/c++传统；注意声明委托的位置-避免写错地方结果声明成嵌套类型；委托与所封装的方法必需类型兼容；返回值的数据类型一致；参数列表的个数和数据类型上一致（参数名不需要一样）

委托可以理解为把某些事件，交给第三方（无关系者）让它们变成订阅关系。函数指针调用方法是，接受方法签名后，把接收到方法签名的变量当成方法来使用（参考js方法）。

### 直接调用

直接调用-通过函数签名来调用函数，CPU通过函数名直接获得函数所在地址并开始执行→返回；

### 间接调用

间接调用-通过函数指针来调用函数，CPU通过读取函数指针存储的值获得函数所在地址并开始执行→返回；

### 委托声明(自定义委托)

### Action委托

Action 不能指向有返回值的方法，不能有返回值。

### Func委托

Func 可以指向有一个返回值的方法， 且必须有返回值。

### 多播委托（multicast）

### 单播委托

一个委托封装一个方法的模式是单播委托。

### 委托的一般使用

实例-把方法当作参数传给另一个方法；注意-难精通+易使用+功能强大东西，一旦被滥用则后果非常严重，缺点1-这是一种方法级别的紧耦合，现实工作中要慎之又慎；缺点2-使可读性下降，debug的难度增加；缺点3-把委托回调，异步调用和多线程纠缠在疫情，会让代码变得难以阅读和维护；缺点4-把委托使用不当有可能造成内存泄漏和程序性能下降；

#### 模版方法

正确使用1-模版方法，借用指定的外部方法来生产结果，相当于填空题，常位于代码中部，委托有返回值；

#### 回调方法

正确使用2-回调（callback）方法，调用指定的外部方法，相当于流水线，常位于代码末尾，委托无返回值；

### 隐式异步调用

同步与异步的简介－中英文的语言差异，同步－你做完了我做（在你的基础上）接着做；异步－咱们同时做（相当于汉语中的同步进行）；同步调用与异步调用的对比－每一个运行的程序是一个进程（process）；

每一个进程可以有一个或多个线程（thread）；同步调用是在同一个线程内；异步调用的底层机理是多线程；

串行==同步==单线程，并行==异步==多线程；

隐式多线程VS显示多线程

直接同步调用=使用方法签名；间接同步调用=使用单播/多播委托的invoke方法；隐式异步调用=使用委托的Begininvoke；显示异步调用=使用Thread或Task；

使用接口代替委托=java中完全使用接口取代了委托功能，即java没有与C#中委托相对应的功能实体。

## 事件

事件的功能=通知+可选的事件参数（即详细信息）

事件模型的五个组成部分=事件的拥有者（event source，对象）→事件成员（event，成员）→事件响应者（event subscriber，对象）→事件处理器（event handler，成员）=本质上是一个回调方法→事件订阅=把事件处理器与事件关联在一起，本质上是一种以委托类型为基础的“约定”。

注意=事件的处理器是方法的成员；挂接事件处理器的时候，可以使用委托实例，也可以直接使用方法，这是“语法糖“；事件处理器对事件的订阅不是随意的，匹配与否由声明事件时所用的委托类型来检测；事件可以同步调用也可以异步调用。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 事件的订阅者 | 事件消息的接收者 | 事件的响应者 | 事件的处理者 | 被事件所通知的对象 |
| 事件参数 | 事件信息 | 事件消息 | 事件数据 |  |

### 定义

单词Event，译为“事件“；

《《牛津词典》》中的解释是“a thing that happens，especially something important“；

通顺的解释就是“能够发生的什么事情“；

### 角色

使对象或类基本通知能力的成员；

（中译）事件（Event）是一种使对象或类能够提供通知的成员；

（原文）An event is a member that enables an object or class to provide notifications；

对象它拥有一个事件E“想表达的思想是=当事件E发生的时候，它有能力通知别的对象“

### 使用

用于对象或者类间的动作协调与信息传递（消息推送）

### 原理

事件模型（Event model）中的两个“5“

“发生→响应中的5个部分“＝闹钟响了你起床，孩子饿了你做饭．．．这里隐含着订阅关系；

“发生→响应中的5个动作“＝（１）我有一个事件→（２）一个人或者一群人关心者我的这个事件→（３）我的这个事件发生了→（４）关心这个事件的人会被依次通知到→（５）被通知到人根据拿到的事件信息（又称＂事件数据＂，＂事件参数＂，＂通知＂）对事件进行响应（又称＂处理事件＂）

### 提示

事件多用于桌面，手机等开发的客户端编程，因为这些程序经常是用户通过事件来＂驱动＂的；

各种编程语言对这个机制的实现方法不尽相同；

Java语言里没有事件这种成员，也没有委托这种数据类型。Java的“事件”是使用接口来实现的。

MVC，MVP，MVVM等模式，是事件模式更高级，更有效的“玩法”；

日常开发的时候，使用已有事件的机会比较多，自己声明事件的机会比较少，所以先学使用。

内容决定形式，功能决定使用方法。

## 泛型

当一个类后面跟着尖括号<>就是泛型类。泛型类不是一个完整的类，它需要和其他类组合在一起才是完整的类。泛型（generic）无处不在,为什么需要泛型：避免成员膨胀或者类型膨胀；正交性：泛型类型（类/接口/委托……），泛型成员（属性/字段/方法）；类型方法的参数推断；泛型与委托，lambda表达式。

### 泛型类

### 泛型方法

### 泛型接口

### 泛型委托

### 泛型委托类型推断

### 泛型LIST类

泛型LIST底层是一个数组，array来容纳数据，泛型数组在数据装满时，长度会动态改变，也叫动态数组。

## 异常

|  |  |
| --- | --- |
| Try | 用于检查发生的异常，并帮助发送任何可能的异常。 |
| Catch | 以控制权更大的方式处理错误，可以有多个 catch 子句。 |
| finally | 无论是否引发了异常，finally 的代码块都将被执行。 |

## LINQ

# 高级篇

## 进程与线程