C#笔记

- 基本介绍
  - C#的特点
  - .NET 与.NET Framework
  - Visual Studio 安装
  - C#特点
  - 基本操作
- 基本语法
  - 简单规范
  - 数据类型
    - 变量
  - 表达式
  - 分支语句
    - if 语句
    - if-else 语句
    - switch 语句
    - 循环语句
      - for循环
      - while循环
      - do-while循环
  - 数组
    - 数组的声明
    - 通过索引获取数组中的元素。
  - 标识符和关键字
- 类与对象
  - 面向对象
    - 面向对象的特点
    - 1.1 类
    - 1.2 对象
    - 实例化
    - 构造方法和析构方法
    - 方法的重载
    - 封装、继承、多态
    - this与base
    - 访问修饰符
      - 访问修饰符:
      - 访问级别约束:
    - static
    - const 、readonly
      - const与readonly区别
    - sealed与abstract
    - virtual与override
    - 接口
    - 枚举
    - 异常处理
  - 委托、事件
    - 委托
    - 事件
  - Lambda表达式
  - 泛型
  - 数组、集合
    - 数组
    - 集合
      - ArrayList
      - ArrayList常用方法:

C#笔记

基本介绍

C#的特点

- C#是微软公司发布的一种面向对象的、运行于.NET Framework和.NET Core（完全开源，跨平台）之上的高级程序设计语言
- C#是一种安全的、稳定的、简单的、优雅的，由C和C++衍生出来的面向对象的编程语言。它在继承C和C++强大功能的同时去掉了一些它们的复杂特性
- C#是**面向对象**的编程语言

.NET 与.NET Framework

.NET 框架是一个创建、部署、运行应用系统的多语言平台环境语言

.NET Framework是微软为开发应用程序而创建的一个平台，利用它，你可以开发Windows桌面应用程序，Web应用程序，Web服务以及其它类型的应用程序。.NET Framework的设计方式确保它可以用于各种语言，包括C#,C++，Visual Basic, JScript等。

Visual Studio 安装

Mirosoft Visual Studio(简称VS)是美国微软公司的开发工具包系列产品。VS是一个基本完整的开发工具集。是编写C#程序或者说.NET程序最常用的开发工具。因其功能强大、简单易用、速度快、智能度高。被网友戏称宇宙第一IDE。

1.下载请到官网下载，地址可百度，根据需要下载对应版本，社区版免费

2.vs安装对网络有要求，如安装失败或下载失败或速度较慢可百度，有很多解决方法就不一一赘述

3.根据自己需要安装模块，一般选择“ASP.NET开发”、“.NET桌面开发”、“通用Windows平台开发”、“数据库存储和处理”、“Visual Studio扩展开发”，当然，如果后续需要在安装别的还可以打开该安装程序安装、更新。

C#特点

支持多线程

相较于C与C++，C#对指针的使用有限制

自动内存分配、垃圾回收

基本操作

注意:

- C#大小写敏感的。
- 所有的语句和表达式必须以分号`;`结尾。
- 与Java不同的是，文件名可以不同于类的名称。
- 代码块在`{}`内表示

C#是一种面向对象的编程语言。在面向对象的程序设计方法中，程序由各种对象组成。相同种类的对象通常具有相同的类型。

console.read（）——输入一个字符

console.readline（）——输入一行字符串

console.write（）——输入一个数据或一个字符串

console.writeline（）——输入一个字符串并且换行

强制类型转换

int.parse（）、double.parse（）、Convert.Toint（）等等

注意

Convert.Tolnt32(double value)遵守四舍五入，而parse则`向下取整`，但:

`Convert.Tolnt32(double value)`,不完全遵循四舍五入，如果value为两个整数中间的数字，则返回二者中的**偶数**,

对比下面的例子:

```
Console.WriteLine(Convert.ToInt32(4.3));//四舍五入，输出4
Console.WriteLine(Convert.ToInt32(4.5));//第一位小数为5时，4. 5在4和5之间，输出偶数4
Console.WriteLine(Convert.ToInt32(4.53)); //四舍五入，输出5
Console.WriteLine(Convert.ToInt32(5.3));//四舍五入，输出5
Console.WriteLine(Convert.ToInt32(5.5)); //第一位小数为5时，5.5在5和6之间，输出偶数6
Console.WriteLine(Convert.ToInt32(5.53)); //四舍五入，输出6
```

> 注意：`ConvertToInt32()`和`int.Parse()`对于空值（`null`）的处理不同，`ConvertToInt32(null)`会返回0而不会产生任何异常，但`int.Parse(null)`则会产生异常。

基本语法

简单规范

注意:

- C#大小写敏感的。
- 所有的语句和表达式必须以分号`;`结尾。
- 与Java不同的是，文件名可以不同于类的名称。

C#是一种面向对象的编程语言。在面向对象的程序设计方法中，程序由各种对象组成。相同种类的对象通常具有相同的类型。

关键字，是对编译器有**特殊意义**的**预定义保留标示符**，它们**不能**在程序中用作标示符

- using关键字

  在任何C#程序中的第一条语句都是: `using System;`

  `using`关键字用于在程序中包含命名空间。一个程序可以包含多个`using`语句。

- class关键字

  class关键字用于声明一个类。

- C#的注释方式

  1. // 单行注释
  2. /**/ 多行注释
  3. /// 文档注释
  4. 注释的作用:
     - 解释:说明代码作用
     - 注销:将暂时不需要的代码注销

注释类型	Java	C#
行注释	//...	//...
块注释	/* ... */	/* ... */
文档注释	/** ... **/	/// ... ///

要注意几件事，对于代码的规范性要有要求:

> 1. 代码注释是一定要写的，有句话叫不写注释那就是耍流氓
> 2. 对于名称的命名，需要注意规范命名方法，驼峰命名法是一个很好地选择，命名要见文生义

数据类型

其中，值类型数据存放在栈中

引用类型数据则为一个指针，指向堆中存放地址。指针源于c语言，指向内存地址

变量

- 变量是一个供程序存储数据盒子。在C#中，每个变量都有一个特定的类型，不同类型的变量其内存大 小也不尽相同。
- C# 中提供的基本类型大致分为以下几类：

类型	举例
整数类型	byte、short、int、long
浮点型	float、double
十进制类型	decimal
布尔类型	bool
字符类型	string、char
空类型	null表达式

表达式

- 表达式由**操作数(operand)**和**运算符(operator)**构成。运算符的示例包括`+`、`-`、`\*`、`/`和`new`。操作数的示例包括文本、字段、局部变量和表达式。
- 当表达式包含多个运算符时，运算符的优先级(precedence)控制各运算符的计算顺序。例如，表达式`x + y * z`按`x + (y * z)`计算，因为`*`运算符的优先级高于`+`运算符。
- (了解)大多数运算符都可以**重载(overload)**。运算符重载允许指定用户定义的运算符实现来执行运算，这 些运算的操作数中至少有一个，甚至所有操作数都属于用户定义的类类型或结构类型。
- 下表总结了C#简单常用的运算符，并按优先级从高到低的顺序列出各运算符类别。同类别中的运算符 优先级相同。

类别	表达式	说明
基本	`x.m`	成员访问
	`x(...)`	方法和委托调用
	`x[...]`	数组和索引器访问
	`newT(...)`	对象和委托创建
	`newT(...){...}`	使用初始值设定项创建对象
	`new{...}`	匿名对象初始值设定项
	`newT[...]`	数组创建
一元	`+x`	恒等
	`-x`	求相反数
	`!x`	逻辑求反
	`~x`	按位求反
	`++x`	前增量
	`--x`	前减量
	`x++`	后增量
	`x--`	后减量
	`(T)x`	将x显示转换为类型T
二元	`x * y`	乘法
	`x / y`	除法
	`x % y`	取余
	`x + y`	加法，字符串串联
	`x - y`	减法
	`x << y`	位左移
	`x >> y`	位右移
	`x < y`	小于
	`x > y`	大于
	`x <= y`	小于或等于
	`x >= y`	大于或等于
	`x is T`	如果 x 位 T，返回 `true`，否则 `false`
	`x as T`	返回转换为类型 T 的 x ,如果 x 不是 T 则返回 `null`

类别	表达式	说明
	`x == y`	等于
	`x != y`	不等于
	`x & y`	整形按位与 ,布尔逻辑AND
	`x | y`	整形按位或 ,布尔逻辑OR
	`x && y`	且，当 x 为`true`时，才对 y 求值
	`x | | y`	或，当 x 为`false`时。才对 y 求值
	`x ?? y`	如果 x 为`null`，则计算结果为 y，否则为 x
三元	`x ? y : z`	如果 x 为`true`,对 y 求值，x 为`false`，对 z 求值
赋值或匿名函数	`x = y`	赋值
	`x = x + y`	复合赋值
	`(T x) => y`	匿名函数（lambda表达式）

分支语句

if 语句

if-else 语句

switch 语句

循环语句

for循环

```
for(int i = 0; i<10;i++ ){     }
```

while循环

```
while(true){  }
```

do-while循环

```
do{  }while(true)
```

数组

- 数组是一组相同类型的数据。
- 数组中的数据需要通过数字索引来访问。

数组的声明

- 数组的声明需要使用`new`关键字。
- 在声明数组时，可以使用`{}`来初始化数组中的元素。
- 如果在数组声明之初没有使用大括号来初始化数组中的元素，则需要指定数组的大小。
- 在声明初始化有元素的数组时，也可以指定数组大小。

```
//声明没有元素的数组
int[] ints = new int[6]
//声明初始化有元素的数组
int[] ints = new int[]{1, 3, 4, 5}
//在声明初始化有元素的数组时，也可以指定数组大小
string[] strings = new int[5]{"H",  “E",  "L", “L","O"}
```

通过索引获取数组中的元素。

- 给数组指定长度时，数组准备存放多少元素，长度就设置为多少。
- 用索引获取数组内的元素时，索弓|从0开始获取。
- 所以数组中最大的索引数字，比指定数组长度小1。

```
//声明初始化有元素的数组
int[] ints = new int[]{1, 3, 4, 5}
//获取数组中第1个的元素。
int i1 = ints[0];
//给数组内的元素赋值
ints[0]  = 1
```

标识符和关键字

- 标识符的构成字母、数字、下划线、unicode字符组成
- 开头不能是数字
- 不能与关键字重名，区分大小写
- 允许标识符以@作为前缀

`@`可用于取消转义，于`\`的作用相同，但其只要在引用部分之前加上即生效

```
@"D:\Tencent Files\2482608252\FileRecv"
等效于 "D:\\Tencent Files\\2482608252\\FileRecv"
```

扩展

ref与out

ref 关键字使参数按引用传递。其效果是，当控制权传递回调，用方法时，在方法中对参数所做的任何更改都将反映在该变量中。若要使用 ref 参数，则方法定义和调用方法都必须显式使用 ref 关键字。

out 关键字会导致参数通过引用来传递。这与 ref 关键字类似，不同之处在于 ref 要求变量必须在传递之前进行初始化。若要使用 out 参数，方法定义和调用方法都必须显式使用 out 关键字。

同：

1、都能返回多个返回值。

2、若要使用 ref 和out参数，则方法定义和调用方法都必须显式使用 ref和out 关键字。在方法中对参数的设置和改变将会直接影响函数调用之处(参数的初始值）。

异

1、ref指定的参数在函数调用时候必须初始化，不能为空的引用。而out指定的参数在函数调用时候可以不初始化；

2、out指定的参数在进入函数时会清空自己，必须在函数内部赋初值。而ref指定的参数不需要。

个人理解：ref与out相当于指针的调用，ref调用指针指向原地址，修改数据，但要注意，一定要是初始化赋值的地址；out则先删掉原指针地址数据，要求传递的只是这个地址，所以无需赋值，并要求在方法中必须赋值

ref正确用法

```
class Program
   {
       static void Main(string[] args)
       {
           Program pg = new Program();
           int x = 10;
           int y = 20;
           pg.GetValue(ref x, ref  y);
           Console.WriteLine("x={0},y={1}", x, y);

           Console.ReadLine();
       }

       public void GetValue(ref int x, ref int y)
       {
           x = 521;
           y = 520;
       }
   }
```

out正确用法

```
class Program
    {
        static void Main(string[] args)
        {
            Program pg = new Program();
            int x=10;
            int y=233;
            pg.Swap(out x, out y);
            Console.WriteLine("x={0},y={1}", x, y);

            Console.ReadLine();
        }
```

```
        public void Swap(out int a,out  int b)
        {
            int temp = a;    //a,b在函数内部没有赋初值，则出现错误。
            a = 521;
            b = 520;
        }
    }
```

总结ref和out的区别萝卜头敲代码的博客-*CSDN*博客out ref

类与对象

面向对象

概念

`面向过程`

面向过程是一种自上而下的过程，顺序执行。、

没有固定的分工，项目复杂或需求复杂，则工作量大。

`面向对象`

> 相比较函数， 面向对象是更大的封装，根据职责，在一个对象中封装多个方法

1. 在完成某一个需求前，首先确定职责，要做的事情（方法）
2. 根据职责确定不同的对象，在对象内部封装不同的方法（多个）
3. 最后完成代码，就是顺序的让不同的对象调用不同的方法、

面向对象的特点

1. 注重对象和职责，不同的对象承担不同的职责
2. 更加适合应对复杂的需求变化，题门应对复杂项目开发，提供固定套路
3. 要在面向过程基础上，学习些面向对象的语法

正式介绍面向对象

1.1 类

- 类是对一群具有相同特征或者行为的事物的一个统称，是抽象的，不能直接使用
    - **特征**被称为**属性**
    - **行为**被称为**方法**
- 类就相当于制汽车是的图纸，是-一个模板，是负责创建对象的

1.2 对象

- 对象是由类创造出来的一个具体存在，可以直接使用
- 由哪一个类创造出来的对象，就拥有在哪一个类中定义的属性和方法
- 对象就相当于用图纸制造的汽车

在程序开发中，应先有类，再有对象

在程序开发中要设计一个类，通常需要满足以下三个要素:

1. **类名**这类事物的名称，满足大驼峰命名法
2. **属性**这类事物具有什么样的特征
3. **方法**这类事物具有什么样的行为

实例化

- 类使用关键字`new`实例化对象。
- 一个类可以实例化多个对象。
- 对象可以使用类定义的属性和方法。

拓展

`new`是新建对象开辟内存空间的运算符，以类作为模板，开辟空间并执行相应构造方法

除了以上最基础用法，还有两种引申用法：

2. new 修饰符：在用作修饰符时，new 关键字可以显式隐藏从基类继承的成员。
   但其实这种方法在实际编程中用的不多，容易产生歧义，不建议使用，做了解就好

```
  using System;

   namespace ConsoleApplication1
{
    public class BaseA

   {
        public int x = 1;
        public void Invoke()
        {
            Console.WriteLine(x.ToString());
        }
        public int TrueValue
        {
            get { return x; }

   set { x = value; }
        }
    }

   public class DerivedB : BaseA
    {
        new public int x = 2;
        new public void Invoke()
        {
            Console.WriteLine(x.ToString());
        }
        new public int TrueValue
        {
            get { return x; }

   set { x = value; }
        }
    }
    class Test
    {
        static void Main(string[] args)
```

```
        {

    DerivedB b = new DerivedB();

    b.Invoke();//调用DerivedB的Invoke方法，输出：2

    Console.WriteLine(b.x.ToString());//输出DerivedB的成员x值：2

    BaseA a = b;

    a.Invoke();//调用BaseA的Invoke方法，输出：1

    a.TrueValue = 3;//调用BaseA的属性TrueValue，修改BaseA的成员x的值

    Console.WriteLine(a.x.ToString());//输出BaseA的成员x的值：3

    Console.WriteLine(b.TrueValue.ToString());//输出DerivedB的成员x的值，仍然是：1

    //可见，要想访问被隐藏的基类的成员变量、属性或方法，办法就是将子类造型为父类，然

    //后通过基类访问被隐藏的成员变量、属性或方法。
        }
      }
}
```

该部分代码来自：https://blog.csdn.net/qq_36325332/article/details/89474378
对new关键字有更详细的介绍

3.new 约束：用于在泛型声明中约束可能用作类型参数的参数的类型。

该用法未尝试，了解一下

1.如果属性中具有`get`关键字，说明可以获取该属性的值。

2.如果属性中具有`set`关键字，说明可以向该属性设置值。

```
public class Tree
{


        int age;//字段
         public int Age//属性
        {
        get { return age; }
        set { age = value; }
    }
        string kemu;//字段
        public string Kemu//属性
        {
        get { return kemu; }
        set { kemu = value; }
    }

    public void put(int a)//方法
    {
```

```
        Console.WriteLine("\n编号：{0}\n名称：{1}\n树龄：{2}\n科目：
{3}",num,name,age,kemu);
        Console.WriteLine("\n这颗树{0}年开一次花\n", a);
    }

}
```

语法糖：

```
public string Name{get;set;}//语法糖格式,不需要写字段，能根据该格式自动生成相应的字段
系统会自动生成：
string name;
public string Name{
get{return name;}
set{this.name=value;}
}
```

构造方法和析构方法

构造方法:是一种特殊的方法、与类同名的方法，专门用于创建对象

- 可以有一个或者多个构造方法
- 方法名与类名相同
- 没有返回值
- 可以带着参数
- 具有修饰符
- 不能直接调用，必须通过new运算符

析构方法：释放对象所用到的方法

- 没有返回值也没有**修饰符**
- 不带参数
- 与类同名
- 需要`~`作为开头
- 不能显式调用，但在对象释放时自动调用

当然，如何判断对象是否被释放?

对象名.Dispose();是一种释放对象的方法，如果释放，则报错。

```
 class Gouzao
  {
      protected int id;
      protected string name;
      protected int age;
     public  Gouzao(int id, string name, int age) {
          this.id = id;
          this.name = name;
          this.age = age;
      }
      //析构方法，在对象释放时调用，一般不设置，用~加类名构成，~是shit加esc下面的键打的
      ~Gouzao()
      {
```

```
            Console.WriteLine("这里是析构方法，即将Destroying Gouzao类");
        }
    }
      public class Program {
        public static void Main(string[] args) {
          Gouzao Gz = new Gouzao(1, "zhangsan", 14);//在创建对象时需要实现构造方法
        }
    }
```

方法的重载

可以对普通方法、构造方法的重载

两同两不同

- **相同的类名**
- **相同的方法名**
- **不同的参数类型**
- **不同的参数个数**

注意点：

int类型可以去找double类型进行重载，因为int转换成double不会丢失数据——向上查找（不丢失信息）

也被称为静态多态，编译器根据签名的不同而进行不同的操作

```
  internal class ChongZai
    {
        public void Eat()
        {
            Console.WriteLine("我今天我还没吃饭，好饿.");
        }
        public void Eat(string Bea)
        {
            Console.WriteLine("我今天吃了" + Bea);
        }
        public void Eat(string Bea, string Lun)
        {
            Console.WriteLine("我今天吃了" + Bea + "和" + Lun);
        }
        public void Eat(string Bea, string Lun, string Din)
        {
            Console.WriteLine("今天我吃了" + Bea + "和" + Din);
        }
        public void Eat(int x, string Bea, int y, string Lun, int z, string Din)
        {
            Console.WriteLine("今天我吃了" + x + "份," + Bea+y + "份," + Lun + "和"
+ z + "份" + Din,"真的好撑");
        }
    }
```

封装、继承、多态

封装，它是面向对象思想的特征之一，它是指通过具体功能封装到方法中。

> 封装提高了代码的复用性，隐藏了实现细节，还要对外提供可以访问的方式，便于调用者的使用提高了安全性。

继承，是指事物之间的所属关系，通过继承可以使多种事物之间形成一种关系体系。

当一个类是另一个类中的一种时，可以通过继承，来继承属性与功能。如果父类具备的功能内容需要子类特殊定义时，需进行方法重写。

```
class Teacher:Person//: 表示继承
   {

        public Teacher(string name,int age)
            : base(name)
        {
            this.Age = age;
        }
        private int _age;
        public int Age
        {
            get { return _age; }   //读
            set { _age = value; }  //写
        }
        public void tshow()
        {
            Console.WriteLine("我是学生{0}{1}! ",name,Age);
        }


原文链接：https://blog.csdn.net/wrs120/article/details/53667362
```

但要注意：C#中类的继承是`单继承`，`多实现`,多实现指的是多个接口实现

多态，就是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们可以通过相同的方式予以调用。同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。

其中，重载（overload）和重写（override）是实现多态的两种主要方式。

当然，子类对父类方法的重构，子类除了继承、重构父类方法，也可以自定义新方法

类的初始化顺序

- 先变量后构造函数。变量先被初始化，然后构造函数被执行。
- 先静态化后实例化。当一个类被访问时,静态变量和构造函数最先被初始化。接着是对象的实例化变
- 量和构造函数被初始化。
- 先派生类后基类（实例构造函数除外）。

this与base

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

this：

1.限定被相似的名称隐藏的成员，方法或函数（多数用在构造函数中）

```
    class Person
    {
        private string name;
        public Person(string name)
        {
            this.name = name;
        }
    }
//这是name的使用遵循一种'就近原则'，使用传参name，如若要使用Person的name，则需要使用this关键字
```

2.访问字段和方法

```
  class Teacher
  { string name;
      public void tshow()
      {
          Console.WriteLine(name);
          Console.WriteLine(this.name);name与this.name一样
      }
  }
```

base:

1.在子类中通过base调用父类的构造函数，即在子类初始化时和父类进行通信。（但正常情况是无法访问父类的构造函数）

```
  public Teacher(string name,int age): base(name)
        {
            this.Age = age;
        }
```

要注意，调用父类的构造方法，需要带上相应的参数，父类构造器有几个参数，就需要在base（参数）中写几个（要注意重载）

2.访问父类字段和方法

```
a=this.age;
b=base.age;
```

3.调用被重写的父类方法

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(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。

访问修饰符

访问修饰符:

- `public`: 公有的，**所有的类**都可以访问
- `private`: 私有的，当前**类**内部可访问。
- `protected`: 受保护的，**当前类**以及继承他的**子类**可访问
- `internal`:内部的，只限于**本项目内**访问，其他的不能访问。
- `protected internal`:内部保护访问，只能是**本项目内部或子类**访问，其他类不能访问

访问级别约束:

- **父类子类**访问修饰符要保持**一 致**
- 方法的**访问修饰符**要和方法参数的**访问修饰符保持一致**

> 注意: 类的访问级别默认为隐式私有需要加上`public`才可让外部访问

静态方法、 属性

- 静态和属性方法通过`static`关键字修饰
- 静态和属性可以通过类型直接获取，非静态则必须通过实例化的对象获取

静态类

- 静态类通过`static`关键字修饰
- 一般情况下类型不需要使用静态修饰，只有当类型中存在扩展方法时需要使用静态类

static

static：静态的，其不属于某个实例，而是属于整个类。

> 静态字段不保存在某个对象实例的存储区间内，而是在类的公共存储区间

调用静态方法、变量为：**类名.方法(字段\属性);**

static是**"非实例的"**

要注意：

- static方法属于类
- 其处理对象只能处理属于整个类的成员变量，而不能操纵和处理属于某个对象的成员变量
- static只能处理static字段或调用static方法
- 被static修饰的方法（类方法）不能使用this和base
- Console.WriteLine()这类方法就为静态方法

static可修饰构造方法，相对于实例构造方法，静态构造方法是对类自身进行初始化定义。

const 、readonly

const关键字限定一个变量不允许被改变。（只能修饰简单类型与字符串）
1.用于修改字段或局部变量的声明，表示指定的字段或局部变量的值是常数，不能被修改。
2.常数表达式是在编译时可被完全计算的表达式。因此不能从一个变量中提取值来初始化常量。
(如果const int a = b + 1; b是一个变量，显然不能在编译时就计算出结果，所以常量是不可以用变

量来初始化的。）

3.常数声明可以声明多个常数。

```
public const double x = 1.0, y = 2.0, z = 3.0;
```

4.const隐含static，调用时类名.常量，如：Math.PI

const也可修饰方法局部变量，转化为“常量”，这是局部变量唯一可用的修饰符

readonly允许把一个字段设置成常量。

1.readonly是在计算时执行的，所以它可以用某些常量初始化。

2.readonly是实例成员，所以不同的实例可以有不同的常量值，这使readonly更灵活。

3.不能修饰局部变量，不隐含static性质

4.不要求初始化，若无初始化，默认赋值（0，flase、null）

5.可在构造方法中赋值，也可初始化时赋值，但最多只能**赋值一次**

const与readonly区别

1.const字段只能在该字段的声明中初始化；readonly字段可以在声明或构造函数中初始化，因此，根据所使用的构造函数，readonly字段可能具有不同的值。

2.const字段是编译时常量，而readonly字段可用于运行时常量。

```
public static readonly uint a1 = (uint)datetime.now.ticks;
```

3.const默认就是静态的，而readonly如果设置成静态的就必须显示声明。

4.const对于引用类型的常数，可能的值只能是string和null。readonly可以是任何类型。

该部分借鉴:https://blog.csdn.net/yiyelanxin/article/details/72461257

sealed与abstract

sealed

`sealed`修饰的方法可用于对类的修饰，修饰的类不**可被继承**

例如：String、Int32、Math等，可去查看其源代码，被sealed修饰

abstract

被abstract修饰则附加抽象buff，如抽象类、抽象方法、抽象属性

抽象类

- 不能被实例化；（不能new实例）
- 是所有子类的公共属性的集合
- 可有构造方法，可被子类构造方法调用
- abstract的子类依旧可为abstract类型
- “抽象类**不一定**有抽象方法;但是包含一个抽象方法的类**一定**是抽象类。(有抽象方法就是抽象类,是抽象类可以没有抽象方法)”
- 抽象类可以包含非抽象的方法，如普通方法，虚方法等
- 继承的子类如果没实现父类中所有的抽象方法，那么这个子类也必须是抽象类（**子类必须要实现所有，否则依旧为抽象**）

```
//抽象类
public abstract class Student{
//抽象方法
abstract int Do();
//抽象属性
abstract string name{ get; set; }
}
```

抽象方法

- 为所有子类定义唯一接口
- 只需声明，无需定义（用";",不用"{}"）

注意：

- abstract不与static、private同时修饰一个方法
- abstract方法必须在抽象类中
- 子类实现abstract方法要用override修饰

virtual与override

virtual

修饰成员为虚成员，可被覆盖

override

覆盖父类成员

可重写virtual与abstract修饰成员

- virtual不能与static、private共同修饰
- abstract隐含virtural

接口

接口：用于定义一组抽象操作的集合，通常为一些抽象的方法和属性——更加抽象的东西

使用关键字：interface

- 隐含 public
- 习惯加上I，如下：ICar
- 没有构造方法、析构方法
- 没有字段，只有属性、方法，且不实现任何东西

```
public interface ICar {
        void Start();
        void Stop();
        void Fly(int u);
```

```
public class MinCar : ICar {
        public void Start() {
            Console.WriteLine("我是钥匙启动");
```

```
        }
        public void Stop() {
        Console.WriteLine("我是盘式刹车");
        }
    }
    public class BigCar : ICar {
        public void Start()
        {
            Console.WriteLine("我是钥匙启动");

        }
        public void Stop()
        {
            Console.WriteLine("我是股式刹车");
        }

    }
```

注意：C#为“单继承，多实现（接口）”

与抽象类对比的不同点

1.抽象类只能单一继承,接口可以实现多继承

2.抽象类中可以有普通方法，虚方法等，接口只能写规范，不可实现

3.抽象基类可以定义字段、属性、方法实现。接口只能定义属性、索引器、事件、和方法声明，不能包含字段。

接口的引用：

显示接口成员实现:多个接口可能存在相同签名的方法，可通过接口名.方法名实现

```
class FileViewer : IWindow,IfilehANDLER
{
void IWindow.Close(){...}
void IfilehANDLER.Close(){...}
}
```

通过强转类型的方式调用：

```
((IWindow)f).Close();
```

枚举

其默认顺序对应（0,1,2,3...）

```
public enum Day
   {
       Sunday = 0,
       Monday = 1,
       Tuesday = 2,
       Wednesday = 3,
       Thursday = 4,
       Friday = 5,
       Saturday = 6,
   };
```

异常处理

```
try
{
可能会出现异常的代码
}
Catch
{
出现异常后要执行的代码
}


Finally
{
}
```

在使用try catch 语块时需要注意的是：

- 使用多个catch语块时，应把最具体的（即派生程度`最高`的）放在`最前面`
- 尽量不要将try...catch写在循环中
- try尽量少的代码，如果有必要可以使用多个catch块，并且将最有可能抛出的异常类型，书写在距离try最近的位置
- Catch块尽量避免直接捕捉异常的基类Exception，而应该捕捉具体的异常类。

委托、事件

委托

委托（delegate）将方法作为方法的参数，可称为“函数指针”，一般和事件（event）绑定使用。

注意：

- 委托类型是一种类型，代表一类函数

  ```
  public delegate double text （double x）;
  ```

定义一种类型为带有**一个参数**，且类型为**double**，**返回值为double**的这类类型的方法，这种类型叫做text

- 声明方式

  委托类型名　　委托变量名

  text　　t;

- 实例化

```
text t=new text(方法)//这里的方法不加括号，就是单纯的方法名，无需加(),,,,
如:Math.Sqrt
```

- 委托的调用

```
t(3.0)//与函数调用方式一样，调用委托，会执行所有已经被委托的方法。
```

这里就体现了委托的一种合并性，叫多播。一种对方法的包装，可通过+，-来实现委托的合并与移除

```
t+=t1;//t为委托，t1为方法
t-=t1;
```

- 委托的定义建议放在类外边

事件

事件—— 一个类和另外一个类之间传递信息或者触发新的行为的一种编程方法。

```
public event 委托类型名 事件名; //事件的声明，以及与委托绑定
```

委托与事件步骤

- **声明一个委托**
- **声明一个委托类型的事件**
- **编写事件的处理方法**
- **将事件和事件处理方法联系（绑定）起来**
- **调用触发事件的函数**

实例:

```
internal class FireEvent
  {
      public  void louder()
      {
          Console.WriteLine("着火了着火了！！！");

      }
      public  void water() {
          Console.WriteLine("水来了，水来了，水来了");
      }
      public  void CallPolice() {
      Console.WriteLine("报警了，报警了，报警了！！！");
      }
```

```
    }



class Program
    {
        //声明委托
        public delegate void fireEventhappen();
        //声明该委托类型的事件
        public static event fireEventhappen fireE;
        static void Main(string[] args)
        {
            FireEvent fe = new FireEvent();
            //将对应的方法绑定给事件
           // 事件+=委托(方法)
            fireE += new fireEventhappen(fe.CallPolice);
            fireE += new fireEventhappen(fe.Water);
            fireE += new fireEventhappen(fe.louder);

            while(true)
            {
                if(int.Parse(Console.ReadLine())>=200)
                {
                    fireE();//触发事件
                }


            }
        }
    }
```

这个博客写的十分不错，可以去更深入了解[C# 理解委托与事件（烧水壶例子） - zhengwei_cq - 博客园 (cnblogs.com)]

Lambda表达式

这里就又要介绍一种名为**Lambda表达式**

学过js的应该了解js里有一种箭头函数，与此十分相似

```
类型 名称=(变量，变量...)=>{语句体；}；//要注意除了语句体的分号外，其本身括号外也有分号。
```

泛型

我们在编程程序时，经常会遇到功能非常相似的模块，只是它们处理的数据不一样。这里就要引出泛型这个概念

泛型，相当于一个模具，是编写一个类可以针对不同的类型。通俗的讲，我这有一个小狗冰激凌模具，不管你给我是巧克力、奶油、果冻，我都使用小狗冰淇淋模具来处理你，做出小狗样子的巧克力、奶油块、果冻。

优点

- 代码复用性好
- 无需封装
- 安全性更好
- 一定程度避免了装箱和拆箱
- 泛型相当于`类型占位符`
- 定义类或方法时使用`替代符`变量类型
- 在声明的时候只是使用了`T`作为`类型占位符`，不知道具体是哪种类型，传给`T`一个指定的类型，才确定类型

约束条件

定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的几种类型施加限制。 如果客户端代码尝试使用约束所不允许的类型来实例化类，则会产生编译时错误。 这些限制称为约束。 通过使用`where`上下文关键字指定约束。

- where T：struct——必须为值类型
- where T：class——引用约束，必须是引用类型的数据
- where T：new()——构造函数约束，必须为无参数的构造函数
- where T：基类名——T必须是指定的基类或者派生类
- where T：接口名——T必须是指定的接口或者实现指定接口

```
class MyClass<U>where U : struct///约束U参数必须为“值 类型”
 {
}

 public void MyMetod<T>(T t)   where T : struct
 {
 }
```

```
Demo1
static void Swap<T>(ref T a, ref T b)
{
    T temp=a;
    a=b;
    b=temp;
}//可交换各种类型，极大的优化的代码使用量，提高复用率

Demo2
List<int> list1=new List<int>();
list1.Add(44);        //无装箱
list1.Add(“ABC”);//编译错误，这里声明类型为int 在添加字符型会报错，相对安全，若使用
foreach(int j in list1){}


ArrayList list1=new ArrayList();
list1.Add(44);        //装箱
list1.Add(“ABC”); //装箱,这里装箱类型各种类型都行，有好有坏，读取必须按类型依次读取，但可存储各种数据类型
foreach(int j in list1){}//执行报错
```

```
Demo3
    public class MyClass<T1,T2>
{
//定义构造函数
public MyClass(T1 t1,T2 t2)
{
     T2 t3=t1+t2;
}
} //这里要注意，T1与T2类型可能不同，所以无法相加
```

default

default运算符在泛型中用于获取类型的默认值。
①引用类型的默认值为null；
②值类型的默认值：数值类型默认值为0，结构中数据成员是引用类型的则赋null，是数值类型则赋0。

```
Demo4
    static public T GetDefault<T>()
{
return default(T);
}
//调用方式(所有泛型的调用方式)
int i=GetDefault();//错误
int i=GetDefault<int>();//在调用时，必须要声明泛型类型，否则报错
//default运算符在泛型中用于获取类型的默认值。
//①引用类型的默认值为null；
//②值类型的默认值：数值类型默认值为0，结构中数据成员是引用类型的则赋null，是数值类型则赋0。
```

调用方式：**int i=GetDefault();//在调用时，必须要声明泛型类型，否则报错**

泛型中有**List**、**Dictinary<TKey,TValue>**，与ArryList对照介绍，这两者使用较多

数组、集合

数组

声明　：　`type [] arrayNmae`

`arrayName=new type[arraySize]`

多维数组声明：`type [,,,] arrayNmae`

`arrayName=new type[arraySize,arraySize,arraySize]`

最常用的数组声明：`type [] arrayNmae=new type[arraySize]`

例如：

```
int [] i=new int [3];
//多维数组的声明
int [,]A = new int[2,2]{{1,2},{3,4}};
```

常用方法：

- **Sort（）**
  - 数组的排序，无返回值。
  - Array.Sort(数组名);
- **Reverse（）**
  - Array类的静态方法 **Reverse（）**可以实现数组的反转，无返回值。
  - Array.Reverse(数组名);
- **IndexOf（）**
  - Array类的静态方法 **IndexOf（）**可以判断数组是否包含与给定值相等的元素，是则返回对应数组元素的下标，否则返回-1。
  - Array.IndexOf(数组名);

```
int Index;
int[] a = { 1, 4, 5, 7, 3 };
Index = Array.IndexOf(a, 7);
Index = Array.IndexOf(a, 1);
```

- C#提供了方法 **Sum（）、Max（）、Min（）、Average（）**对数组元素进行统计。

```
int nResult;
double dResult;
int[] a = { 1, 4, 5, 7, 3 };
nResult = a.Sum();
nResult = a.Max();
nResult = a.Min();
dResult = a.Average();
```

- **Join（）**
  - String类的静态方法 **Join（）**可以将数组中所有元素连接起来，生成一个字符串，返回值为该字符串。
- **Split（）**
  - String类的静态方法 **Split（）**可以根据分隔符将字符串切分为多个部分，每个部分作为一个数组元素生成一个字符串数组，返回值为该字符串数组。

```
string str1, str2;
int[] a = { 1, 4, 5, 7, 3 };
string[] b = { "this", "is", "a", "cat", "!" };
str1 = string.Join("", a);
str2 = string.Join("", b);
string str3="this is a cat !";
string[] c=str3.Split(' ');
```

对于数组的操作还有其他的，可自行查看文档

集合

1. 可以看做成数组
2. 数组 长度固定、不太灵活
3. 使用集合可方便解决数组的问题。
4. 可以将集合看做为一个 长度可变的，具有很多方法的数组。
5. 给集合添加数据，用Add()这个方法，添加的内容，无限添加。

集合的分类

- **ArrayList（顺序表）**
- **Queue（队列）**
- **Hashtable（哈希表）**
- **Stack（栈）**
- **SortedList（有序表）**

ArrayList

1.引入System.Collection命名空间
2.实例化ArrayList对象

1. **/* ArrayList 的特点**
2. 长度可变的数组
3. 可存放任意类型的数据
4. 有很多操作数据的方法
5. 取数据时需要强制转换(强行转换成原类型输出)

```
 using System;
using System.Collections;//这里一定要引入

namespace ArrayListApp
{
    class Program
    {
        static void Main(string[] args)
        {
            /* ArrayList 的特点
             * 1.长度可变的数组
             * 2.可存放任意类型的数据
             * 3.有很多操作数据的方法
             * 4.取数据时需要强制转换
             */

            // 创建一个数组
            ArrayList list = new ArrayList();
            // 数组中可以加入不同的数据类型
            list.Add("hello world"); // 加入字符串
            list.Add('N'); // 加入字符
            list.Add(true); // 加入布尔
            list.Add(false);
            list.Add(10);// 加入整型
            list.Add(18.5f);// 加入浮点数
            list.Add(18.78);
            Man baba = new Man();// 加入对象
            list.Add(baba);
```

```
            list.Add(new string[] {"语文", "数学", "英语"}); // 加入字符串列表

            // 从列表中循环打印
            for (int i = 0; i < list.Count; i++)
            {
                // 打印数组中的所有元素
                Console.WriteLine(list[i]);
                //  如果是Man的对象，则执行SayHi
                if (list[i] is Man)
                {
                    ((Man)list[i]).SayHi(); // 获取该类型前需要转换为Man类型
                }
                else if (list[i] is string[])
                {
                    for (int j = 0; j < ((string[])list[i]).Length; j++)
                    {
                        // 这里要重点理解 list[i] 是列表索引对象，调用对象时，需要先将对象转换为对应的类型所以写为 (string[])list[i]
                        // 把(string[])list[i] 视为一个整体字符串列表，所以需要在外面加一层括号 ((string[])list[i])
                        // 打印字符串列表中的每个对象 ((string[])list[i])[j]
                        Console.WriteLine(((string[])list[i])[j]);
                    }
                }
            }
        }

        public class Man
        {
            public void SayHi()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
```

该部分代码实例转自() C# ArrayList使用方法*MakChiKin*的博客-*CSDN*博客c#arraylist

ArrayList常用方法：

Add( ) 往集合添加数据,算成一个元素
AddRange( ) 数组里边添加元素
Insert( ) 往某个下标的位置插入一个值
Remove() 删除
Clear() 清空
arrayList.Reverse() 反转
arrayList.Sort() 排序
Contains(1) 判断这个集合中包含这个元素
arrayLlist.Count 集合中元素的个数
arrayLlist.IndexOf(1) 找某一个元素对应的索引
InsertRange(4, nums) 往某一个索引位置插入一个数组
arrayLlist.RemoveAt(1) 移除索引对应的值
arrayLlist.RemoveRange(3,5) 从指定索引处开始移除，移除多少个,如果超出索引报异常

代码实例

```
ArrayList arrayList=new ArrayList();
arrayList.Add(123);      //将数据新增到集合结尾处
arrayList.Add("abc");    //将数据新增到集合结尾处
arrayList[2]=345;        //修改指定索引的数据
arrayList.RemoveAT(0);   //移除指定索引处的数据
arrayList.Remove(123);   //移除内容为123的数据
arrayList.Insert(0,"hello world");    //再指定索引处插入数据
```

`ArrayList`的劣势

- `ArrayList`在存储数据时是使用`object`类型进行存储的
- `ArrayList`不是类型安全的，使用时很可能会出现类型不匹配的错误
- 就算都有插入了同一类型的数据，但在使用的时候，我们也需要将它们转化为对应的原类型来处理
- `ArrayList`的存储存在**装箱和拆箱**操作，导致其性能低下

装箱与拆箱的概念

- **装箱**就是将比如`int`类型或者`string`等不同的对象通过**隐式转换**赋给`object`对象。

```
int i = 123;
object o= 1;
```

- **拆箱**就是将`object`对象通过**显示转换**赋给`int`类型的变量

```
object o=123;
int i = (int)o;
```

- 装箱与拆箱的过程会产生较多的 性能损耗。
- 正是因为`ArrayList`存在不安全类型与装箱拆箱的缺点，所以在C#2.0后出现了**泛型**的概念。
- 泛型的概念在此先不多做表述，为便于大家记忆，可以简单理解成：限制集合只能够存储**单一类型数据**的一种手段。

`List`集合

List集合的声明

- `List` 集合与`ArrayList` 由于都继承成了相同的接口，故使用与`ArrayList` 相似。
- 在声明`List`集合时，需要同时为其声明`List`集合内数据的**对象类型**
- 示例: `List<int> intList = new List();`

所谓**接口**目前可以简单理解成**限制和规定类型行为**即**类型方法**的一种手段

```
List<int> list = new List<int>();   // 第一种初始化方式
list.Add(123);       // 新增数据到结尾处
List<int>  intList = new List<int>  // 第二种初始化方式
{
    123,
    456,
    789
};
intList[2] = 345;
intList.RemoveAt(0);     // 删除指定索引处的数据
intList.Remove(123);     // 删除内容为123的数据
intList.Insert(0, 6688);
```

上例中如果我们往List集合中插入string字符"hello world",系统就会报错，且不能通过编译。这样就避免了前面讲的类型安全问题与装箱拆箱的性能问题

思考

- 在上一节"类的使用"中我们知道，`int`本身也是一 个类型，`int`t类型声明的变量接受`int`类型的数据，`int`类型可以指定`List`集合的数据类型。那么我们自己创建的类型是否可以限定`List`集合的数据类型呢?
    - 答案是：可以。

总结

- 集合与数组比较类似，都用于存放一组值
- 集合中提供了特定的方法能直接操作集合中的数据，并提供了不同的集合类来实现特定的功能
- 简单的说就是数组的升级版。他可以动态的对集合的长度(也就是集合内最大元素的个数)进行定义和维护
- `List`泛型的好处指通过允许指定**泛型类或方法**操作的**特定类型**，减少了类型强制转换的需要和运行时错误的可能性，泛型提供了类型安全，但没有增加开销。

HashTable

- ArrayList每个元素对应一个索引
- HashTable通常称为哈希表
- 根据键(key)可查到相应的值(value)，键和值一一对应

```
Demo1

//实例化Hashtable
            Hashtable hashTable = new Hashtable();
            //增加元素
            hashTable.Add("1", "案秀云");
            hashTable.Add("2", "案秀云-解决方案库");
            hashTable.Add("2", "案秀云-IT知识库");
            //删除元素
            hashTable.Remove("1");
            //删除所有元素
            hashTable.Clear();
            //元素个数
            int count=hashTable.Count;
            //判断是否包含key
            Boolean isKey= hashTable.ContainsKey("1");
            //判断是否包含value
            Boolean isValue= hashTable.ContainsValue("案秀云");
```

```
Demo2
遍历
//key的遍历
            foreach (String key in hashTable.Keys)
            {
                Console.WriteLine(hashTable[key].ToString());
            }
            //value的遍历
            foreach (String value in hashTable.Values)
```

```
            {
                Console.WriteLine(value);
            }
            //键值对KeyValuePair<T,K>遍历
            foreach (KeyValuePair<String , String > kv in hashTable)
            {
                Console.WriteLine(kv.Key + kv.Value);
            }
```

```
//实例化一个ArrayList对象，并把Hashtable的keys进行赋值
            ArrayList akeys = new ArrayList(hashTable.Keys);
            //按Key字母顺序进行排序,ArrayList Sort也可以指定范围进行排序
            akeys.Sort();
            //排序后的遍历输出
            foreach (string skey in akeys)
            {
               Console.WriteLine(hashTable[skey]);
            }
```

该部分demo引用自，如想 C#中Hashtable使用大全凡梦的博客-CSDN博客_c# hashtable了解更多可访问该网站

Dictionary

相比于hash表，字典的应用则更为广泛

- 在声明`Dictionary`字典时，需要同时为其声明`Dictionary`字典内的键与值的类型
- 示例：`Dictionary<int, string> dictionary = new Dictionary<int, string>();`

键与值可以是任何类型，但是键必须在设置时是唯一的，而值可以不唯一，就好比每个学生的学号必须是唯一的，而所有的成绩可以不唯一。

```
Dictionary<int, string> dictionary = new Dictionary<int,  string>();//首先需要实例化
//两种赋值方式
// 方  式 一 ：Add 方 法 赋 值
dictionary.Add(1, "98分");
dictionary.Add(2, "92分");
dictionary.Add(3, "89分");
dictionary.Add(1, "88分");   //系统会报错
//方式二：索引器赋值
dictionary[1] = "88分";      //系统不会报错
dictionary[4]  = "99分";
//方式三：对象初始化器
Dictionary<string, string> dictionary2 = new  Dictionary<string, string>()  {
    {"A","aa" },
    {"B","BB" },
    {"C","CC" }
};
```

注意`dictionary[1]`方式既可以赋新值可以修改原来已键有的值，类似于数组索引器的使用，所以可以使用之前已使用过的键。但是`Add`方法不可以添加已有键的值。

```
//获取键为1的值
//方式一：索引器取值
string value = dictionary[1];
//方式二：foreach遍历取值
foreach (KeyValuePair<string, string> item in dictionary) {
    int key  = item.Key;
    string value = item.Value;
}
//移除键为1的键值对
dictionary.Remove(1);
```

一个小Demo：

```
internal class Program
   {
       static void Main(string[] args)
       {
           Dictionary<int,Animal> animals = new Dictionary<int,Animal>();
           Animal dog=new Animal(1,"dog",30,20);
           Animal cat=new Animal(2, "cat", 25, 17);
           Animal pig = new Animal(3, "pig", 100, 30);
           animals.Add(dog.Num,dog);
           animals.Add(cat.Num,cat);
           animals.Add(pig.Num,pig);
           foreach (Animal value in animals.Values)
           {
               Console.WriteLine("编号："+value.Num);
               Console.WriteLine("姓名：" + value.Name);
               Console.WriteLine("质量（/kg）：" + value.Weight);
               Console.WriteLine("高度（/cm）：" + value.Height);
           }
           Console.ReadLine();
       }

   }
   public class Animal {

       int num;
       public int Num {
           get { return num; }
           set { num = value; }
       }

        string name;
       public string Name
       {
           get { return name; }
           set { name = value; }
       }
       float weight;
       public float Weight
       {
           get { return weight; }
           set { weight = value; }
       }
        float height;
```

```
        public float Height
        {
            get { return height; }
            set { height = value; }
        }
        public Animal(int Num,string Name,float Weight,float Height) {
            this.Num = Num;
            this.Name = Name;
            this.Weight = Weight;
            this.Height = Height;
        }

    }
```

总结

- **键**与**值**可以是任何类型，但是键必须在设置时是唯一 的，而值可以不唯一
- 使用`Add()`方法添加键值对，不可添加已有的键名
- 索引模式可以**新赋值**也可以**修改已有的键值。**

`foreach`的使用

- `foreach`一个相较于for循环更智能的遍历方法
- `foreach`对遍历字典或集合具备天然优势，效率高过`for`循环

`foreach`操作数组

```
int[] ints= {1, 2, 3, 4, 5, 6};
foreach (int item in ints){  //每次循环，其item都是整型数组中的一个元素  }
```

`foreach`操作集合

```
List<int> intList = new  List<int>() { 1, 2, 3, 4, 5, 6 };
foreach (int item in ints){  //每次循环，其item都是List集合中的一个元素  }
```

`foreach`操作字典

```
Dictionary<string, string> dictionary = new Dictionary<string,  string>() {
    { "A","aa"},
    { "B","bb"},
    { "C","cc"},
};
foreach  (KeyValuePair<int, string> item  in dictionary) {
    int key  = item.Key;
    string value =  item.Value;
}
```

全文主要引用() C#（笔记本）——第一章 lucid_dreamings的博客-CSDN博客

以及babbittry/Csharp-notes: C# 课程笔记 (github.com)

有想了解更多可访问其博客，谢谢！！！