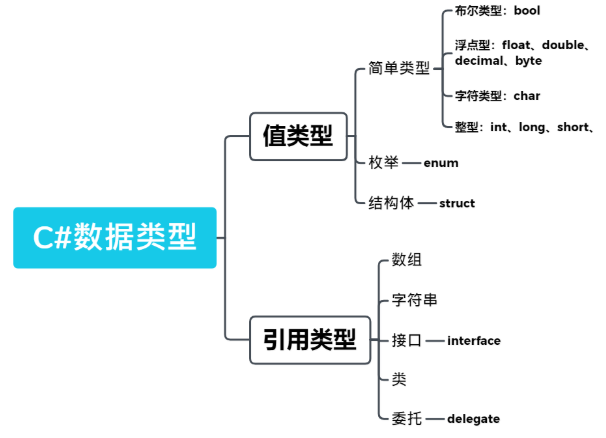
# 文档4 C#语法摘录与VS调试技巧

## 数据类型

C#是强类型的编程语言，所有的变量都要求有明确的数据类型，开发过程中根据所要存放的数据类型，选择合适的数据类型。



### 值类型的数据类型

整数，浮点数，布尔类型，字符，枚举，结构体

1. 整数



1. 浮点类型

float数据类型用于较小的浮点数，因为它要求的精度较低。

数值默认为double类型，如果要给float变量赋值，可以在其后加上字符F(或f)。

float f = 12.3F;

double数据类型比float数据类型大，提供的精度也大一倍(15位)。

decimal类型专门用于进行财务计算,使用decimal类型提供的28位的方式取决于用户。

要把数字指定为decimal类型，可以在数字的后面加上字符M或(m)，如：

decimal d=12.30M;

1. 布尔类型

具有true和false的变量类型。

bool a=true;

1. 字符类型

char类型的字变量是用单引号括起来的。     如'A'

如果把字符放在"A"(双引号)内，编译器会把它看作是字符串，从而产生错误。

1. 枚举类型

枚举类型（enum type）是具有一组命名常量的独特的值类型。在以下示例中：

//定义枚举类型

enum Color

{

Red,

Green,

Blue

}

//利用枚举类型定义变量，变量的值必须是某个枚举项。

Color c1;

c1 = Color.Red;

1. 结构体类型

结构体类型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。

定义结构体类型

struct Books

{

public string title;

public string author;

public string subject;

public int book\_id;

};

//通过结构体类型声明变量，并为变量的不同成员赋值。

Books b1;

b1.title= "项目实训";

b1.author = "王志瑞";

b1.subject = "全方位学习内容";

b1.book\_id = 1;

### 引用类型的数据类型

引用类型和值类型存储方式不同，值类型变量内部直接存储变量值，引用类型指向所要存储的对象，即变量内存放的是值所在存放空间的起始地址。

值类型的变量的内存开销小，访问速度快，而引用类型变量的内存开销大，访问速度稍慢。

值类型的存放空间在栈空间中分配，引用类型的存放空间在堆空间分配，堆里的空间能够以任意顺序存入和移除，通过CLR进行垃圾回收。

常见的引用类型有：

* 类
* 委托
* 事件
* 接口
* 数组

### 类型相关的其它内容

1. 可空类型

引用类型变量可以存放空值，非引用类型不能够存放空值，为了解决这个问题，C#中提供了可空类型，只要在基本数据类型后边增加“？”，即可定义可空类型。

例如：

int? a=null;

1. 类型转换

隐式类型转换 - 这些转换是 C# 默认的以安全方式进行的转换, 不会导致数据丢失。例如，从小的整数类型转换为大的整数类型，从派生类转换为基类。

显式类型转换 - 显式类型转换，即强制类型转换。显式转换需要强制转换运算符，而且强制转换会造成数据丢失。

显示类型转换一般采用convert.ToXXX来完成的数据类型转换。

例如：

Convert.ToDateTime("2018-01-01")

Convert.ToDecimal("200")

## 变量与常量 – 数据容器

### 变量

变量是指在程序运行过程中随时可以发生变化的数据量，保存的数据允许发生变化。

变量就是表示内存中的一块存储区域，内存地址不方便使用，因此有了**变量名**。

变量名要按照要求的规范进行命名。

**变量的声明：**

数据类型 变量名;

int age;

每个数据类型后边，可以定义多个变量。

string name,address,origin,national;

**为基本数据类型的变量赋值：**

变量名=”值”;

可以在变量声明的时候为变量赋值。

### 常量

所谓常量，就是在程序执行中其值固定不变的数据量，保存的数据不允许发生变化。在定义变量的时候加上const修饰符，变量就变成了常量，定义常量的时候必须赋初值。

const double PI=3.1415926;

## 数组 – 集合数据容器

### 一维数组

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。

一维数组的定义示例

//方式1：定义的时候指定数组空间，后边为数组元素赋值

double[] balance = new double[10];

balance[0] = 4500.0;

//方式2：定义数组的时候，通过赋初值来定义数组空间。

double[] balance = { 2340.0, 4523.69, 3421.0};

//方式3：定义数组，设定空间，并设置初始值

int [] marks = new int[5] { 99, 98, 92, 97, 95};

//通过下标来引用数组元素，从0开始。

int a=marks[0];

### 二维数组

二维数组可以看成是数组的数组，它的每一个元素又是一维数组。

二维数组的定义示例：

//方式1：定义数组时分配空间

int[,] a = new int[2, 3];

//方式2：定义数组时分配空间并赋初值

int[,] b = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };

//方式3：定义数组时通过赋初值设定空间

int[,] c = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } };

通过两个下标来引用数组元素，从0开始，d的值为5

int d = c[1,1];

## 运算符 – 数据运算符

计算机中，多个操作数之间要进行运算，需要靠运算符，不同运算符能够运算的数据类型不同，根据变量的类型和实际运算需要选择合适的运算类型。

C#支持的运算符类型

### 算数运算符

运算结果还是数值



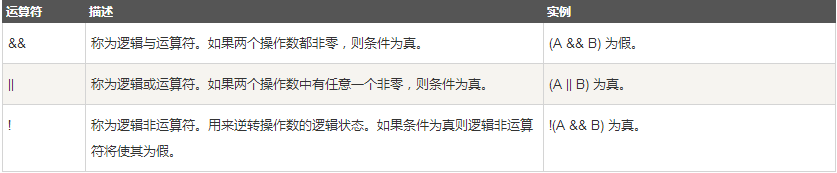
### 关系运算符

运算结果是布尔类型



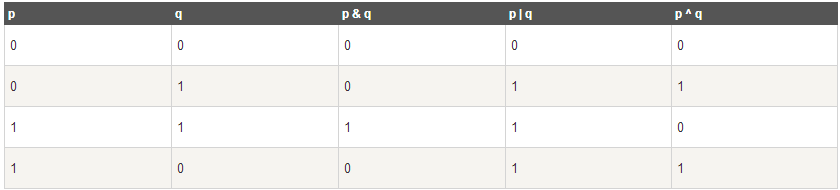
### 逻辑运算符

运算结果是布尔类型



### 位运算符

运算完毕之后是0或1



左移几位《《 ：二进制数据所有数据左移几位，3《《2 等于 12

右移几位》》 ：二进制数据所有数据右移一位, 12》》1 等于 6

### 赋值运算符



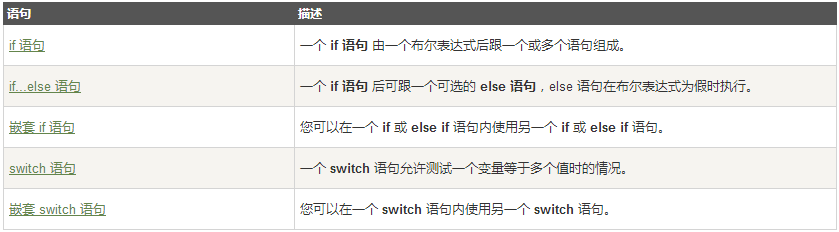
### 其它运算符



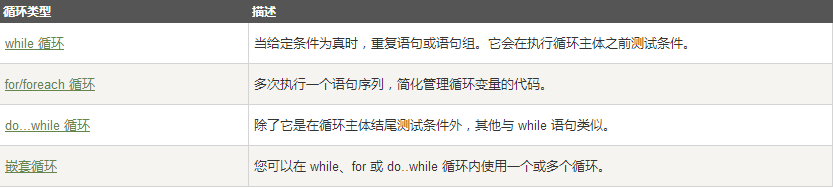
## 程序结构

### 顺序结构

### 选择结构



### 循环结构



1. while（条件）{ 循环体 }是先判断条件是否满足，满足后再进入循环。
2. for(初始值;判断条件;值变更){ 循环体 } 对集合数据进行遍历的
3. foreach(变量 in 集合){ 循环体 } 对集合数据进行遍历的
4. do{ 循环体 }while(条件);

循环控制语句



## 面向对象编程语言

面向对象编程语言（Object-Oriented Language）是以对象作为基本程序单位的[程序设计语言](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1%E8%AF%AD%E8%A8%80/2317999)，拥有三大特性：

**封装:**是指将某事物的属性和行为包装到对象中，这个对象只对外公布需要公开的属性和行为。

**继承:**是子对象可以继承父对象的属性和行为。

**多态性:**是指允许不同类的对象对同一消息作出不同响应。同一个调用语句会执行不同对象里边的方法。

例如：有一个方法会传入Animal类型的参数，在方法内部通过参数调用say方法，由于不同动物所实现的say方法不同，因此调用该方法时，传入不同的动物对象，方法内部相同的调用语句就会执行不同的代码。

## 命名空间

命名空间是在面向对象编程语言中出现的语法特点，是为了方便对众多类的管理，类似磁盘中的目录结构，把不同的类放在不同的命名空间中，方便管理和维护。

一个.cs文件的基本结构如下：

//顶部通过多个using引入该类中所要用到的类所在的命名空间。

using 命名空间;

//定义本类所在的命名空间

namespace 命名空间

{

类定义

{

变量定义;

方法定义

{

方法体代码;

}

}

枚举类型定义

结构体定义

}

我们发现除了方法内部，其它位置撰写的代码基本上都是类型的定义或者类成员的定义，要进行变量的运算，程序结构控制等相关代码，都必须定义在方法体的内部，否则会违背C#语法规范的。

## 类的定义与使用

类是一种自定义数据类型，而对象是具体类型的实例化，是通过类类型所定义的一个变量。

类的定义必须放在命名空间中。

定义的内容必须位于类的内部。

namespace 命名空间

{

类修饰符 类名

{

成员修饰符 数据成员;

成员修饰符 方法成员

{

//普通代码必须放在方法体内

方法体;

}

}

}

**类修饰符有：**

* public：公开类，对类的访问不受限制
* internal：内部类（默认值），当前命名空间内使用的类。
* abstract：抽象类，表示该类是一个不完整的类，不允许建立类的实例。
* sealed：密封类，不允许从该类派生新类。

类是一种复合的数据类型，类的内部可以包含**数据成员和方法成员**等内容，数据成员存储了类或类的实例相关的数据，通常用来表示现实世界事物的特性；方法成员包含代码，通过用方法成员模拟该类所表示的现实世界事物的所拥有的功能和操作。

**类成员允许的修饰符有**

* public：公有成员，提供了类的外部接口，允许从外部访问
* private：默认，私有成员，仅限于本类中成员访问，不能被继承的。
* protected：保护成员，不允许外部访问，但允许派生类访问。
* internal：内部成员，同一程序集中的类才能访问该成员。

### 数据成员

类的内部允许定义如下类型的数据成员

* **实例数据成员**：实例数据成员的存储空间与对象存储空间相关联，需要通过对象使用，可以被子类继承。
* **只读数据成员**：用readonly修饰的数据成员，只读数据成员是和对象关联的，可以定义的时候赋初值，也可以在类的构造函数中赋初值。
* **静态数据成员**：用static修饰的数据成员，静态数据成员是类中所有对象共享的成员，不是某个对象的成员，静态成员的存储空间不是放在每个对象中，而是和方法一样放在类公共区中。
* **常量数据成员**：用const修饰的数据成员，必须赋初值，常量数据成员也是静态数据成员，因此不允许再用static修饰常量数据成员。

例子见构造和析构函数部分。

### 方法成员

类里边允许出现的方法成员有如下几种：

1）构造函数

2）析构函数

3）属性与自动属性

4）函数方法

### 构造与析构函数

**构造函数**

构造函数是在创建对象时自动执行的方法，用来完成对对象内数据成员的相关赋值操作，构造函数有如下的特点：

* 名称和类名保持一致
* 没有返回类型。
* 允许有多个构造函数，声明对象时，调用那个对象，取决于参数的个数和类型。
* 构造函数不允许被继承。

**析构函数**

当对象不需要的时候，希望它所占用的数据存储空间能被收回，C#提供了析构函数用来释放对象使用过程中所占用的系统资源，析构函数有如下特点：

* 析构函数在对象回收的时候自动调用。
* 一个类只允许一个析构函数，并且没有参数。
* 析构函数的名称“~类名”，没有返回值，没有修饰类型。
* 不能被继承。

public class TestClass

{

//普通数据成员

public int a;

//静态数据成员

public static int b;

//常量数据成员

public const int N = 10;

//只读数据成员

public readonly int C;

//构造函数

public TestClass(int c)

{

C = c; //只读数据成员赋值

a = c;

}

//析构函数

~TestClass()

{

}

//测试代码

public void TestInvoke()

{

//通过构造函数创建对象

TestClass t = new TestClass(1);

Console.WriteLine(t.C);

Console.WriteLine(t.a);

Console.WriteLine(TestClass.N);

TestClass.b = 10;

Console.WriteLine(TestClass.b);

}

}

### 属性

属性通常用来关联某个私有数据成员，通过私有数据成员存放数据，通过属性对外公开对私有数据成员的操作，这使得私有数据成员在被访问的同时能提供安全性和灵活性。

属性使用方式像数据成员，本质上是方法成员。

属性的声明语法

public 数据类型 属性名称

{

get 访问器

set 访问器

}

get访问器没有参数，拥有一个与属性数据类型相同的返回类型，get访问器最后需要执行一条return语句，返回一个与属性类型相同的值。

set访问器拥有一个隐式的参数value，与属性的数据类型相同，在set访问器内部可以直接使用value来访问属性所赋予的值。

例如：

public class timePeriod

{

//描述秒

private double seconds;

//描述小时

public double Hours

{

get { return seconds / 3600; }

set { seconds = value \* 3600; }

}

}

### 自动属性

通过属性能够实现私有数据成员的外部访问，如果没有任何逻辑需求，用普通属性方式封装外部访问会有些麻烦，C#3.0之后，当属性的访问器中不需要其它逻辑时，自动属性可使属性的声明更加简洁。

自动属性有如下的特点：

* 不需要定义私有数据成员，编译器根据属性的类型自动创建。
* 不需要提供get和set访问器的方法体。

定义如下：

public class MyClass

{

public int age { get; set; }

public void TestMethod()

{

MyClass obj = new MyClass();

obj.age = 12;

Console.WriteLine(obj.age);

}

}

### 函数方法

函数方法类似于C语言中的函数，函数方法代表了类或对象拥有的功能。

根据函数成员修饰符的不同可以分为**实例方法和静态方法**。

* 实例方法：无static修饰的方法，需要通过对象调用，允许子类继承。

对象名.实例方法名();

实例方法内可以使用实例数据成员，也可以使用静态数据成员。

* 静态方法：有static修饰的方法，称为静态方法，需要通过类名调用。

类名.静态方法名(参数);

静态数据成员和静态方法不能够被继承。

静态方法内只能使用静态数据成员，不能够使用实例数据成员。

* 选择规范

定义方法的时候，如何判断应该定义成那种类型的方法？

主要看你定义的方法内是否要使用到内部提供的成员属性，例如有个Person类，内部有姓名和年龄等成员属性，现在要提供一个方法用来输出姓名和年龄，那应该采用哪种类型的方法那？

如果Person类要提供一个方法用来把参数传入的Person对象转成字符串输出，那应该采用哪种类型的方法那。

### 方法的参数类型

在函数方法定义过程中，需要为函数定义参数，C#中支持的参数类型有：

1. **普通参数**

普通参数，在方法调用时，在栈中会为形参分配空间。

参数传递类似于变量赋值，进行方法调用的时候，会把实参的值赋值给形参，在方法体内通过形参来使用参数值。

不管参数是值类型还是引用类型，都是这样处理的，只是引用类型的形参和实参指向同一个内存空间。

如果是**值类型**，方法内对形参的值进行修改不会影响实参的值。

如果是**引用类型**，例如类，方法内对形参的属性进行修改会影响实参的属性值，如果直接给形参分配一个新的对象，则对实参无影响。

1. **引用参数**

在方法的参数中以ref修饰符声明的参数属于引用参数（和引用类型不是一个概念），引用参数不会在栈上为形参分配空间，形参的参数名将作为实参变量的别名，指向相同的内存位置。

在调用方法前，实参必须被初始化，并且方法调用的时候，需要在实参前加上ref。

当参数前加上ref之后，方法内对形参的各种方式的修改都会影响到实参。

1. **输出参数**

以out修饰符声明的参数是输出参数。与引用参数类似，输出参数也不会分配空间，在调用时输出参数对应的实参前面要加上out修饰符。

输出参数和引用参数的差别有：

* 在调用方法前无须对输出参数进行初始化，但在方法内部，输出参数在读取之前必须赋值。
* 在方法返回之前，方法内部必须为输出参数赋值一次。
* 因为方法内部在读取输出参数之前必须对其赋值，因此不可以使用输出参数把数据传入方法。

1. **参数数组**

前面介绍的参数必须一个实参对应一个形参，参数数组不同，它允许零个或多个实参对应一个特殊的形参。

以params修饰符声明的参数为参数数组，用于处理多个参数类型相同但参数个数可变的情况，在方法声明中params关键字之后不允许有任何其他参数，并且在方法声明中只允许有一个params关键字。参数数组不能有ref和out修饰符。

public class MyClass

{

//参数数组实现任意个整数相加

public static void addNum(ref int sum, params int[] b)

{

sum = 0;

foreach (int item in b)

sum += item;

}

public static void TestMethod()

{

int sum=0;

MyClass.addNum(ref sum, 1, 2, 3);

Console.WriteLine(sum);

MyClass.addNum(ref sum, 1, 2, 3, 4, 5, 6);

Console.WriteLine(sum);

}

}

1. **可选参数**

所谓可选参数是指在调用方法时可以包含这个参数，也可以忽略它。为了表明这个参数是可选的，需要在定义方法时为它提供参数默认值。

public class MyClass

{

public static int add(int a, int b = 1, int c = 2)

{

return a + b + c;

}

public static void TestMethod()

{

Console.WriteLine(MyClass.add(3));

Console.WriteLine(MyClass.add(3, 4));

Console.WriteLine(MyClass.add(3, 4, 8));

}

}

可选参数有如下的规定：

可选参数必须放在必填参数的后边。

如果有参数数组，则参数数组必须放在可选参数之后。

可选参数的默认值，必须是编译时的常量，如下的代码就会报错，d变量的默认值不是常量。

public class MyClass

{

public MyClass()

{

}

public static int add(int a, int b = 1, int c = 2, ***MyClass d = new MyClass()***)

{

return a + b + c;

}

}

### this关键字

this关键在类中使用，是对当前实例的引用。在声明一个类后，当创建该类的一个对象时，该对象方法内部隐含有一个this引用，this执行了当前对象本身。

### 函数的重载

函数的重载就是指一个类中相同名称的方法有多个，为了区分不同的函数，C#中引入了**方法签名**的概念，方法的**参数类型和参数个数**决定了方法的签名信息。

public class MyClass

{

public static int addvalue(int a, int b)

{

return a + b;

}

public static int addvalue(int a, int b,int c)

{

return a + b + c;

}

public static void TestMethod()

{

Console.WriteLine(MyClass.addvalue(3,4));

Console.WriteLine(MyClass.addvalue(3, 4,5));

}

}

### 分部类

分部类可以将类的定义代码拆分到多个.cs源文件中，被拆分类的每一部分在定义时要用partial关键字修饰，在编译时会自动将所有分部类的代码组合起来构成一个完整的类。

### 类的实例化

通过类提供的构造函数来进行对象的实例化，具体格式：

类名 对象 = new 类名(参数);

通过对象访问**实例数据成员和实例方法成员**。

对象.实例数据成员

对象.实例方法成员(参数);

### 匿名类型

C#提供了匿名类型，如果某种类型的对象只需要创建一个，那无需先为其创建一个类，再通过类创建对象，可以直接通过匿名类型来创建对象。

var ojb = new {name=”小王”,age=23};

在MVC开发过程中，很多方法的参数要求采用匿名类型。

## 类的继承

### 继承的语法

继承是面向对象程序设计最重要的特性之一，它允许创建分层次的类。运用继承能够创建一个通用类，它定义了一系列类的共用属性和方法。该类可以被更具体的类继承，每个具体的类都可以再增加了一些自己特有的数据成员或方法成员，或者对父类中的方法成员进行重写或隐藏。

当声明基类后，其派生类的声明格式如下：

[类修饰符] class 派生类 : 基类 {

//派生类的代码

}

派生类将继承基类的所有非私有实例数据成员和实例方法成员。

如果类是从一个基类派生而来的，那么在调用那个派生类的构造函数之前会自动调用基类的默认构造函数（无参构造函数），调用将从最远的基类开始。当销毁对象时，它会按照相反的顺序来调用析构函数，先调用派生类的，再调用基类的。

在派生类中提供了一个base关键字，通过该关键字可以在子类中使用base关键字访问基类成员。

如果调用派生类构造函数的时候，不想调用父类默认的无参构造函数，需要在派生类构造函数定义的时候，在构造函数后边加上base关键字，例如：

class A

{

public A()

{

Console.WriteLine("A的构造函数");

}

public A(int x)

{

Console.WriteLine("A的重构构造函数");

}

}

class B : A

{

public B()

{

Console.WriteLine("B的构造函数");

}

public B(int x,int y)**:base(x) //**

{

Console.WriteLine("B的重构构造函数");

}

public 普通方法()

{

前边代码；

base.父类的方法名(参数)

后边代码；

}

}

class Program

{

static void Main(string[] args)

{

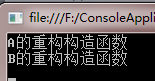
B c = new B(1, 2);

Console.Read();

}

}

程序输入如下：



### 隐藏基类方法

如果在派生类中定义的方法的方法签名与父类的方法的方法签名一样。

如果在派生类方法修饰符前边加上new，则称为隐藏基类方法。

例如：

class A

{

public A()

{

Console.WriteLine("A的构造函数");

}

public A(int x)

{

Console.WriteLine("A的重构构造函数");

}

public void fun()

{

Console.WriteLine("A fun");

}

}

class B : A

{

public B()

{

Console.WriteLine("B的构造函数");

}

public B(int x,int y):base(x)

{

Console.WriteLine("B的重构构造函数");

}

**new** public void fun()

{

Console.WriteLine("B fun");

}

}

### 重写基类方法

如果在派生类中定义的方法的方法签名与父类的方法的方法签名一样。

如果在方法修饰符后边加上override，则称为重写基类方法。

基类方法修饰符后边必须加上 virtual 才允许被重写。

例如：

class A

{

public A()

{

Console.WriteLine("A的构造函数");

}

public A(int x)

{

Console.WriteLine("A的重构构造函数");

}

public **virtual** void fun()

{

Console.WriteLine("A fun");

}

}

class B : A

{

public B()

{

Console.WriteLine("B的构造函数");

}

public B(int x,int y):base(x)

{

Console.WriteLine("B的重构构造函数");

}

public **override** void fun()

{

Console.WriteLine("B fun");

}

}

### 隐藏和重写的区别

1、语法风格上存在区别

区别见上边的代码。

2、调用上存在区别

如果通过派生类声明变量指向派生类对象，调用方法时，都执行派生类的方法。

如果通过基类声明变量指向派生类对象，调用方法时，隐藏基类方法，会执行基类的方法，重写基类方法会执行派生类的方法。

例如：

A c = new B(1,2);

c.fun();

重写则执行B的方法。

隐藏则执行A的方法。

### 引用转换和对象类型判别

**1、引用转换**

存在继承关系的类，可以将派生类对象引用转换为基类对象引用。

转换规则如下：

一个基类定义的变量可以指向其子类的对象，但是不可以访问其子类中新增的成员。

例如：

B b= new B();

A a=b; //基类对象引用变量指向其子类的对象

a.fun();

**2、is运算符**

is运算符用来检查对象是否为某个类型。

例如：

A a= new A();

if(a is A)

Console.WriteLine("a 是 A类型");

**3、as运算符**

C#还提供了as运算符，用来在兼容的引用类型之间进行转换，它类似于强制类型转换，所不同的是，当转换失败时，运算符将返回null。

例如：

A a = new A();

string s= a as string;

## 抽象类与抽象方法

如果存在一个不与具体的事物相联系，而是表达一种抽象的概念，并作为其派生类的一个基类，这样的类可以定义为抽象类。

抽象类具有如下的特性：

1、抽象类不能实例化

2、抽象类可以包含抽象方法和非抽象方法，非抽象方法不允许被重写，可以被隐藏。

3、抽象类可以被抽象类继承。

4、从抽象类派生的非抽象类必须重写所有的抽象方法。

在方法声明中使用**abstract**修饰符的修饰的方法即为抽象方法，抽象方法具有如下的特性：

1、抽象方法没有方法体。

2、抽象方法必须被子类重写。

3、抽象方法只能定义在抽象类中。

例如：

**abstract** class A

{

**abstract** public int fun();

public void show()

{

Console.WriteLine("A抽象类中的非抽象方法");

}

}

class B :A

{

int x, y;

public B(int x,int y)

{

this.x = x;

this.y = y;

}

public **override** int fun()

{

return x \* y;

}

**new** public void show() //只能被隐藏，不能被重写

{

Console.WriteLine("B抽象类非抽象方法");

}

}

class Program

{

static void Main(string[] args)

{

A b = new B(2, 3);

b.show();

Console.WriteLine(b.fun());

Console.Read();

}

}

## 虚方法与抽象方法

用virtual修饰的方法叫做虚方法，虚方法是有方法体的，虚方法在子类中可以被重写，虚方法不要求子类一定要进行重写。



## 接口定义与实现

接口是类之间行为内容的一个抽象，把多个类都需要的行为抽象出来定义成接口，可以更好的控制类之间的逻辑交互。接口具有下列特性：

1、接口类似抽象类，实现接口的任何非抽象类必须实现接口的所有成员方法。

2、不能直接实例化接口。

3、接口不可包含方法的实现。

4、类可以实现多个接口。

5、接口自身可以从多个接口继承。

接口的声明

[接口修饰符] interface 接口名[:父接口列表]

{

//接口成员定义体

}

例如：

public interface IA

{

void mymethod1();

}

public interface IB :IA

{

int mymethod2(int x);

}

public class C : IB

{

public void mymethod1()

{

throw new NotImplementedException();

}

public int mymethod2(int x)

{

throw new NotImplementedException();

}

}

## 接口与抽象类

|  |  |
| --- | --- |
| **抽象类** | **接口** |
| 类内部可以包含普通方法。 | 接口完全是抽象的，不能包含普通方法。 |
| 子类只能继承一个抽象类，如果子类不是抽象类，它必须重写抽象类中的抽象方法 | 子类可以实现多个接口，如果子类不是抽象类，实现接口时，必须实现接口中的所有方法。 |
| 抽象类是一种类 | 接口是不同的类型 |

## 委托

### 委托语法

委托类似于C语言中的函数指针，可以引用方法，当为委托分配了方法后，委托将与该方法名具有相同的功能，可以进行方法的调用。

通过委托可以实现将方法引用作为参数进行传递。

要根据委托要引用的方法签名来定义委托，例如：

public delegate void myDelegate(int n);

如上的代码定义了一种委托类型，通过该委托类型所定义的委托变量只能引用如下规范的方法：

void 方法名(int 参数)

例如：

//定义委托

**public delegate void myDelegate(int n);**

public class TestClass

{

public void fun1(int a)

{

Console.WriteLine("{0}的2倍={1}", a, a \* 2);

}

//委托使用测试方法

public void TestDelegate()

{

TestClass t = new TestClass();

//定义委托变量并指向方法

**myDelegate d1 = t.fun1;**

//通过委托调用方法

d1(34);

}

}

### 委托与匿名方法

所谓匿名方法就是没有方法名称的方法，可以将委托与匿名方法关联，即直接给出方法体，其一般格式如下：

委托类型 委托变量=delegate(方法参数){

方法体;

};

代码例子：

//定义委托类型

public delegate void myDelegate(string mystr);

public class MyClass

{

public static void TestMethod()

{

//通过委托指向匿名方法

myDelegate d = delegate(string a) {

Console.WriteLine(a);

};

//通过委托调用匿名方法

d("Hello");

}

}

### 委托与Lambda表达式

C#3.0引入了Lambda表达式，Lambda表达式为匿名方法提供了一个新的语法。

通过Lambda表达式来描述匿名方法。

语法如下：

(参数)=>{方法体;}

代码例子：

//定义委托类型

public delegate void myDelegate(string mystr);

public class MyClass

{

public static void TestMethod()

{

//利用Lambda表达式表示匿名方法

myDelegate d1 = (string a) => { Console.WriteLine(a); };

myDelegate d2 = (a) => { Console.WriteLine(a); };

myDelegate d3 = a => { Console.WriteLine(a); };

myDelegate d4 = a => Console.WriteLine(a);

}

}

相关规则：

* 参数类型可以省略。
* 当有一个参数时，参数外边的()可以省略。
* 当方法体只有一句的时候，{}可以省略。

## 异常处理

### 异常捕获的代码规范

在程序开发过程中，撰写的代码经常会产生异常，为了能够获取异常，并记录异常的相关错误信息，需要对可能出现问题的代码进行异常处理。

基本格式如下：

try

{

可能出现异常的代码;

}

catch(Exception ex)

{

异常处理代码;

}

finally

{

不管是否异常都会执行到的代码，用来进行资源的释放。

}

调试过程中，通过查看ex对象内包含的错误描述可以很方便的了解出现错误的原因。

### 那些代码一定要加上异常捕获

1）和用户交互的代码一定要加上异常捕获，否则会让用户看到原生异常报错信息。

2）对于一些需要使用系统资源的代码，资源需要申请和释放，如果在资源使用过程中可能产生异常，则一定要把资源使用的代码记上异常捕获处理，在异常的finally语句块中进行资源的释放，否则会出现资源不被释放的漏洞。

3）数据库的事务操作。

### 异常的其它用途

我们在编写工具方法或者业务方法的过程中，可以通过**主动抛出异常**的方式，让方法的调用者清楚方法内部产生的错误原因。

抛出异常采用如下的代码：

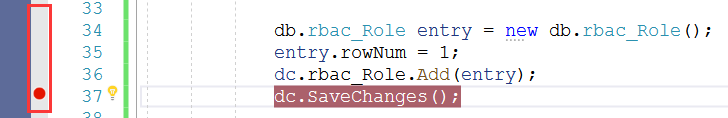
throw new Exception("参数不合法");

在我们编写业务方法代码过程中，根据前文所讲，我们需要在进行数据库操作的过程中加上异常捕获，如果捕获到了异常，我们catch语句需要继续**向外抛出异常对象**，如果不抛出，别人调用你的业务方法过程中，就无法知道方法中出现了数据库的异常信息。

## VS调试技巧

### 断点调试

添加断点方法：鼠标点击Visual Studio代码窗口的最左边灰色一栏，再次点击就可以去掉断点。



在代码合适的位置插入断点，刚开始可以间隔一段加入一个断点，方便调试过程中快速定位出现问题的具体位置。

**调试过程中用到的快捷键：**

F5：运行到下一个断点或者运行到执行结束。

F10：运行到下一行代码。

F11：如果该行代码是一个方法的调用代码，则进入到方法的内部（首先要有方法的源代码再能调试进入）。

F12

VS其它快捷键：ctrl+A+K+F 代码格式化

### Jquery前端代码调试

前端Javascript也支持断点调试，需要启用，用起来不是很方便。

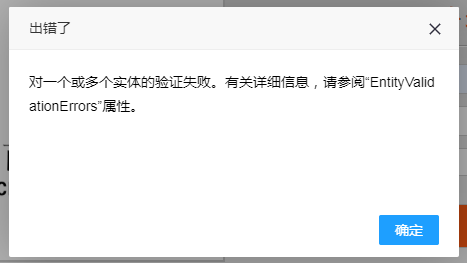
我经常使用的方式，结合Google浏览器调试窗口中的Console选项卡的输出。

1. 首先查看Console选项卡里边是否有红色的错误输出，如果有，则代表代码有语法错误，输出信息中会包含错误的位置。
2. 如果没有语法错误，就需要结合调试输出来定位错误位置。
   * 1. 在事件绑定代码的开始位置加上调试输出，查看是否执行了事件绑定代码，排除是否绑定选择符使用错误。
     2. 如果事件绑定代码输出执行了，再给代码分散性的加上调试输出，可以输出关键变量的值，通过输出的变量值查看是否有错误，也可以直接输出指定文字，通过过文字是否输出了，判断代码是否执行到对应的位置。
     3. 慢慢缩小出错代码区域，最终找到错误原因。
     4. 如果代码需要请求后台获取数据，获取的数据有问题，则配合后台调试一起，定位错误原因。

### MVC后端代码调试

当后端代码执行过程中出现了异常，结合Google浏览器调试窗口的Network选项卡定位异常出现的请求位置，按照如下步骤定位异常：

1. 首先给请求代码加上异常捕获，在异常捕获的catch语句块加上断点。
2. 重新执行程序，再次引发异常产生，查看异常对象内的错误信息。
   1. 如果异常错误在InnerException内，则展开异常对象内部查看具体的错误信息。
   2. 如果异常错误内容是如下的内容，则代表EF实体属性验证不通过，需要把异常对象层层展开，找到EntityValidationErrors中给出的验证错误原因。



1. 清楚错误信息后，给异常捕获的代码分散性的加上断点，通过F5快速执行到断点位置，快速缩小断点的范围，定位到断点具体位置后，再通过一句句执行的方式定位异常产生的代码行。
2. 对产生异常的代码行进行修改排除错误。

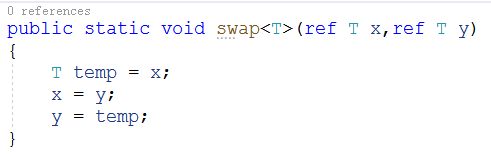
## 泛型

泛型是C#2.0推出的新语法。我们在编程程序时，经常会遇到功能非常相似的方法，只是它们处理的数据不一样。没有泛型之前，只能分别写多个方法来处理不同的数据类型。有没有一种办法，用同一个方法来处理传入不同种类型参数的办法呢？泛型的出现就是专门来解决这个问题的。

### 泛型方法

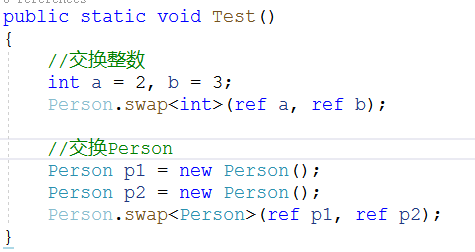
泛型方法是指方法的参数是泛型的，泛型方法可以定义在普通类中，也可以定义在泛型类中。

我们编写一个通用的交换方法Swap，能够完成各种类型数据的交换，代码如下：



方法名后边的<T>，代表方法中所使用类型的占位符，他代表具体的某一个类型，在方法调用的时候，能够确定该类型，因此可以使用该占位符定义变量。

泛型方法的调用：



### 泛型类

如果某个类的数据成员需要定义为泛型的，这个时候就需要定义为泛型类，在定义类名称的时候指定泛型类型的占位符（允许多个占位符），在类的内部就可以用占位符定义数据成员或者定义成员方法的参数类型。

例如：定义一个泛型类，类内部保存三个数据，数据的类型定义为泛型的。

class MyArray<T>

{

private T[] items = new T[3];

private int index = 0;

//向数组中添加项

public void Add(T t)

{

if (index < 3)

{

items[index] = t;

index++;

}

else

{

Console.WriteLine("数组已满！");

}

}

//读取数组中的全部项

public void Show()

{

foreach(T t in items)

{

Console.WriteLine(t);

}

}

}

使用代码：

class Program

{

static void Main(string[] args)

{

MyTest<int> test = new MyTest<int>();

test.Add(10);

test.Add(20);

test.Add(30);

test.Show();

}

}

泛型类定义过程中还有很多的语法规则，大家侧重掌握如何使用泛型类即可，.netFramework中提供了很多的泛型类和泛型接口，大家要侧重掌握如何使用。

## 常用的类

### String

字符串，具体的方法查看字符串类的说明。

### StringBuilder

用于动态构造字符串，如果程序运行过程中，要动态的进行字符串拼接，那个使用stringBuilder比直接使用string拼接效率会高一些。

例如：

StringBuilder s = new StringBuilder();

s.Append("我是");

s.Append("三江人");

Console.WriteLine(s.ToString());

### Math

数学运算封装类。

### Datetime

日期操作封装类。

获取日期中对应的数据项目（年，月，日）

日期值的格式化输出

例如：

//获取当前时间

DateTime d = DateTime.Now;

int year = d.Year;

int month = d.Month;

int day = d.Day;

d.ToString("yyyy-MM-dd hh:mm:ss");

### List<T>

动态数组封装类，数组的替换类，可以存放任意类型的数据成员，不用考虑数组空间的大小问题。

例如：

//定义一个存放字符串的数组

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

//添加数组元素

list.Add("南京");

list.Add("苏州");

list.Add("无锡");

//遍历数组

foreach (string s in list)

{

Console.WriteLine(s);

}

//遍历数组

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

{

Console.WriteLine(list[i]);

}

//输出指定位置的元素

Console.WriteLine(list[0]);

### Dictionary<K,V>

键值对数组封装类，键和值可以创建为任意类型，不用考虑空间问题。

例如：

Dictionary<string, int> person = new Dictionary<string, int>();

person.Add("小王", 20);

person.Add("小张", 30);

person.Add("张飞", 25);

foreach (string key in person.Keys)

{

Console.WriteLine(person[key]);

}

Console.WriteLine(person["张飞"]);

### Tuple

定义一个元组，可以存放多个数据。

Tuple<int, int, int> a = new Tuple<int, int, int>(2, 3, 5);

通过a.Item1访问第一个数据。

通过a.Item2访问第二个数据。

通过a.Item3访问第三个数据。

最多可以存放8个。

当方法需要返回多个值得时候，用起来很方便。

## 其它小知识点

1. 字符串拼接

string a = string.Format(“xxxx{0}xxx,{1}xxxx”,变量1，变量2)，将变量1和变量2的值放到{0}和{1}的位置。

1. 数据格式化

string.Format(“{0:格式化符号}”,变量)，将变量的值进行数据格式化，不同的数据类型提供的格式化串不同：

相关内容参考：

https://blog.csdn.net/shengchang82/article/details/4682112?depth\_1-utm\_source=distribute.pc\_relevant.none-task&utm\_source=distribute.pc\_relevant.none-task

1. 字符串拼接 – Visual Studio 2017以上

string a = $“xxxxx{变量1}xxx,{变量2}xxxxxx”，Visual Studio中新的拼接方式，VisualStudio2013不支持。

## 相关练习

1. 创建一个类Study，在Study类内部添加一个静态测试方法Test1，在方法内完成如下要求。
   1. 定义一个float类型的变量，并赋值为12.01；
   2. 定义一个decimal类型的变量，并赋值为12.02；
2. 定义一个枚举类型WeekDay用来保存“周一~周日”，在Study类内部添加静态测试方法Test2，在方法中利用枚举定义变量并赋值为周三，并输出其值。
3. 定义一个结构体Person，结构体中包含Name和Age属性，在Study类内部添加静态测试方法Test3，利用结构体定义变量，并给属性赋值。
4. 在Study类内部添加静态测试方法Test4，在方法中定义一个整数数组，用来保存1~10，并通过循环输出里边的数据。
5. 在Study类内部添加静态测试方法Test5，在方法中利用List<T>实现整数数组，用来保存1~10，并通过顺序输出里边的数据。
6. 在Study类内部添加静态测试方法Test6，在方法中输出1~100内的质数（质数概念：指整数在一个大于1的自然数中，除了1和此整数自身外，没法被其他自然数整除的数）。
7. 在Study类内部添加静态测试方法Test7，在方法中定义Dictionary<int,WeekDay>，将“1-周一” … “7-周日”等内容添加到集合中，并遍历输出集合中的数据。
8. 完成类相关内容中所给实例代码的运行，查看程序的运行效果。
9. 完成委托中所给实例代码的运行，查看程序的运行效果。
10. 完成泛型类和泛型方法所给实例代码的运行，查看程序的运行效果。
11. 给自己撰写的代码增加断点，学会如何一步步的执行代码。