# visualstudio快捷键

## 快捷注释：crtl+k+c

## 取消快捷注释：ctrl+k+u

## 快捷选中：shift+方向键

# 矩阵数组和交错数组

1. //这是矩形数组，在堆里面是一个数组对象

2. int[,,] data = new int[,,]

3. //这是交错属于组,在堆里面有四个数组对象

4. int[][] ints = new int[3][3]

5.

# String学习

String的split分割如果连续遇到多个分隔符会用空格代替，添加StringSplitOptions.RemoveEmptyEntries来去掉空格

# 数组

数组克隆会在堆里面生成新对象

# 格式化字符串输出

1. int[] intar = { 1, 2, 3, 4, 5};

2. int[] intar2 = (int[])intar;

3. Console.WriteLine("{0}and{1}", intar2[0], intar2[1]);

4. //输出1and2{index，alignment:format}

5. Console.WriteLine($"{intar[0]}and{intar[1]}");

6. //输出1and2

1. int a = 666;

2. Console.WriteLine("|{0,10}|",a);/\*右对齐\*/

3. Console.WriteLine("|{0,-10}|", a);/\*左对齐\*/

4.

# 委托学习

委托就相当于有一个或几个方法的对象，但是委托和典型的对象不同。执行委托，会执行他特有的方法，委托是引用类型，指向了一个调用列表，可以在列表里面增加（+=）方法和移除（-=）方法

1. using System.Diagnostics;

2. using System.IO;

3.  //声明委托

4. delegate void MyDel(int value);

5. internal class Program

6. {

7. void PrintLow(int value)

8. {

9. Console.WriteLine($"{value}Low Value");

10. }

11. void PrintHight(int value)

12. {

13. Console.WriteLine($"{value}Hight Value");

14. }

15. public static void Main(string[] args )

16. {

17. Program program = new Program();

18. MyDel del;

19.

20. Random rand = new Random();

21. int randValue =rand.Next(99);

22. del = randValue < 50 ? new MyDel(program.PrintLow): new MyDel(program.PrintHight);

23. del(randValue);

24. }

25. }

26.

# 匿名方法

1. OtherDel dell=delegate(int x)

2. {

3. return x;

4. };

5.

# Lamda表达式

用lamda表达式来替代匿名方法

1. MyDel dell = delegate (int x) { return x + 1; };/\*匿名对象\*/

2. MyDel lell= (int x) => { return x - 1; };/\*lamda表达式\*/

3.   MYDEL LELL= X=> X-1;

MyDel dell = delegate (int x) { return x + 1; };/\*匿名对象\*/

MyDel lell= (int x) => { return x - 1; };/\*lamda表达式\*/

MyDel lel2 = (x) => { return x - 1; };/\*lamda表达式\*/

MyDel lel3 = x => { return x - 1; };/\*lamda表达式\*/

MyDel lel4 = x => x - 1;/\*lamda表达式\*/

# 泛型学习

提供了五种泛型，分别是类，结构，接口，委托和方法

使用占位符号来声明一个类，为占位符提供真实类型以，这个类型被称为构造类型，可以从构造类型创建实例

## 泛型类

//声明泛型类

class SomeClass<T1, T2>

{

public T1 SomeVar;

public T2 OtherValue;

}

## Where子句

Where来约束类型参数

1. class MyClass<T1,T2,T3>

2. where T2:Customer /\*只有Customer类型或者从Cuotomer类型派生的类才能用作类型参数\*/

3. where T3:IComparable /\*只有实现了IComparable接口的类才能用作类型参数\*/

4.

## 泛型方法

1. public void PrintData<S,T>(S p,T t)

2.

## 扩展方法和泛型类型

泛型类的扩展方法必须

1. 声明为static
2. 必须是进后台类的成员
3. 第一个参数类型中必须有关键词this，后面的是扩展的泛型类的名字

1. static class ExtendHolder

2. {

3. public static void Print<T>(this Holder<T> holder) { }

4. }

5. class Holder<T>

6. {

7. }

8.

## 泛型委托

泛型委托和非泛型委托非常相，不过类型参数决定了能接受什么样的方法

1. delegate Random Mydelegate<T, R>(T value);

2. public delegate TR Func<T1,T2,TR>(T1 value1, T2 value2);

3.

## 协变和逆变

1. delegate Base F<out Base>();/\*协变，指向了基类的引用所以说是类型安全的\*/

2. delegate void<in Derived>(Derived p);/\*逆变，方法收到了一个基类的引用，所以是类型安全的\*/

3.

## 枚举器和迭代器

数组可以按需求提供一个枚举器（enumerator）的对象，枚举器可以依次返回请求的数组中的元素。

枚举器是可以一次返回集合中项目的类对象

foreach结构来设计用来和可枚举类型一起使用，只要给他的遍历对象是可枚举类型，他就会执行如下操作

\*通过调用GetEnumerator方法获取对象的枚举器

\*从枚举器重中请求每一项并且把它作为可迭代变量，代码可以读取该白能量，但是不能改变。

1. foreach(Type VarName in EnuMerableObject){}

## IEnumberator接口

实现了IEnuMberator接口的枚举器包含三个函数成员Current，MoveNext以及Reset

Curren返回当前位置的项的属性，返回的是Object类型的引用，所以可以返回任何类型的对象

MoveNext前进位置到下一项，返回布尔值，位置有效返回true，无效返回false

Reset把位置设置回原始配置

1. internal class Program

2. {

3. public static void Main(string[] args)

4. {

5. int[] arr = { 1, 2, 3, 4, 5 };

6. IEnumerator ie=arr.GetEnumerator();

7. while (ie.MoveNext())

8. {

9. int item = (int)ie.Current;

10. Console.WriteLine(item);

11. }

12. }

13. }

14.

## IEnumberable接口

可枚举类型是实现了Ienumbeable接口的类，IEnumerable接口只有一个成员GetEnumeraator方法，他返回对象的枚举

### 使用IEnumberator和IEnumerable的示例

1. using System;

2. using System.Collections;

3. using System.Diagnostics;

4. using System.IO;

5. class ColorEnumberator : IEnumerator

6. {

7. string[] colors;

8. int position = -1;

9. public ColorEnumberator(string[] theColors)

10. {

11. colors = new string[theColors.Length];

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

13. colors[i] = theColors[i];

14. }

15. public object Current

16. {

17. get

18. {

19. if (position == -1)

20. throw new InvalidOperationException();

21. if (position >= colors.Length)

22. throw new InvalidOperationException();

23. return colors[position];

24. }

25. }

26. public bool MoveNext()

27. {

28. if (position < colors.Length - 1)

29. {

30. position++;

31. return true;

32. }else

33. return false;

34. }

35.

36. public void Reset()

37. {

38. position = -1;

39. }

40.

41. class Spectrum:IEnumerable

42. {

43. string[] Colors= {"violet","blue","cyan","green","yellow","orange","red"};

44. public IEnumerator GetEnumerator()

45. {

46. return new ColorEnumberator(Colors);

47.

48. }

49. }

50.

51. class Program

52. {

53. static void Main(string[] args)

54. {

55. Spectrum spectrum = new Spectrum();

56. foreach(string color in spectrum)

57. {

58. Console.WriteLine(color);

59. }

60. }

61. }

62.

63. }

64.

## 迭代器

我们可以把手动编码的可枚举类型和枚举器替换为有迭代器生成的可枚举类型和枚举器

根据枚举器快的返回类型，你可以让迭代器产生枚举器和可枚举类型

1. using System;

2. using System.Collections;

3. using System.Diagnostics;

4. using System.IO;

5. //class ColorEnumberator : IEnumerator

6. //{

7. // string[] colors;

8. // int position = -1;

9. // public ColorEnumberator(string[] theColors)

10. // {

11. // colors = new string[theColors.Length];

12. // for(int i = 0; i < theColors.Length; i++)

13. // colors[i] = theColors[i];

14. // }

15. // public object Current

16. // {

17. // get

18. // {

19. // if (position == -1)

20. // throw new InvalidOperationException();

21. // if (position >= colors.Length)

22. // throw new InvalidOperationException();

23. // return colors[position];

24. // }

25. // }

26. // public bool MoveNext()

27. // {

28. // if (position < colors.Length - 1)

29. // {

30. // position++;

31. // return true;

32. // }else

33. // return false;

34. // }

35.

36. // public void Reset()

37. // {

38. // position = -1;

39. // }

40.

41. // class Spectrum:IEnumerable

42. // {

43. // string[] Colors= {"violet","blue","cyan","green","yellow","orange","red"};

44. // public IEnumerator GetEnumerator()

45. // {

46. // return new ColorEnumberator(Colors);

47.

48. // }

49. // }

50.

51. // class Program

52. // {

53. // static void Main(string[] args)

54. // {

55. // Spectrum spectrum = new Spectrum();

56. // foreach(string color in spectrum)

57. // {

58. // Console.WriteLine(color);

59. // }

60. // }

61. // }

62.

63. // }

64.

65. //class Myclass

66. //{

67. // public IEnumerator<string> GetEnumerator()

68. // {

69. // IEnumerable<string> myEnumerable =BlackAndWhite();

70. // return myEnumerable.GetEnumerator();

71. // }

72. // public IEnumerable<string> BlackAndWhite()

73. // {

74. // yield return "balck";

75. // yield return "gray";

76. // yield return "white";

77. // }

78. //}

79.

80. class Spectrum

81. {

82. bool \_listFromUVtoIR;

83. string[] colors = { "violet", "blue", "cyan", "green", "yellow", "orange", "red" };

84.

85. public Spectrum(bool listFromUVtoIR)

86. {

87. \_listFromUVtoIR = listFromUVtoIR;

88. }

89.

90. public IEnumerator<string> GetEnumerator()

91. {

92. return \_listFromUVtoIR

93. ?UVtoIR

94. : IRtoUV;

95.

96. }

97.

98. public IEnumerator<string> UVtoIR

99. {

100. get { for (int i = 0; i < colors.Length; i++)

101. {

102. yield return colors[i];

103. }

104. }

105. }

106. public IEnumerator<string> IRtoUV

107. {

108. get

109. {

110. for(int i=colors.Length-1; i >= 0; i--)

111. {

112. yield return colors[i];

113. }

114. }

115. }

116.

117.

118. //public IEnumerable<string> UVtoIR()

119. //{

120. // for(int i=0; i<colors.Length; i++)

121. // {

122. // yield return colors[i];

123. // }

124. //}

125.

126. //public IEnumerable<string> IRtoUV()

127. //{

128. // for(int i=colors.Length-1;i>=0;i--)

129. // yield return colors[i];

130. //}

131.

132. }

133. class Program

134. {

135. static void Main()

136. {

137. //Myclass mc = new Myclass();

138. //foreach(string shade in mc)

139. // Console.WriteLine(shade);

140. //foreach(string shade in mc.BlackAndWhite())

141. // Console.WriteLine(shade);

142. Spectrum startUV =new Spectrum(true);

143. Spectrum startIR =new Spectrum(false);

144.

145. foreach(string color in startUV)

146. Console.WriteLine(color);

147. foreach(string color in startIR)

148. Console.WriteLine(color);

149.

150. }

151. }

152.

# LINQ

LINQ是语言集成查询Language Integrated Query，他是.NET框架的扩展，允许我们使用sql拆线呢数据库的类似的方式来查询数据集合

使用LINQ可以从数据路，对象集合，以及XML文档中查询数据。

1. class Program

2. {

3. static void Main()

4. {

5. int[] numbers = { 1, 2, 3, 4, 5 };

6. IEnumerable<int> lowNums=

7. from n in numbers

8. where n > 0

9. select n; ;

10. foreach (var n in lowNums)

11. {

12. Console.WriteLine(n);

13. }

14. }

15. }

16.

## 匿名类型

1. class Program

2. {

3. static void Main()

4. {

5.

6. //必须使用var,没有类型名

7. //匿名类型只能用于局部变量，不能用于类成员

8. //不能设置匿名类型的对象属性，编译器为匿名类型创建的属性是只读的

9. var student =new { Name = "reai", Age = 18, Major = "Math" };

10. Console.WriteLine($"{student.Name},{student.Age},{student.Major}");

11.

12. }

13. }

14.

## 方法语法和查询语法

1. class Program

2. {

3. static void Main()

4. {

5. int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

6. //查询语法是声明式的，也就是说，查询描述的是你想返回的东西

7. var numsQuery=from n in numbers /\*查询语法\*/

8. where n < 20

9. select n;

10. var numsMethod = numbers.Where(N => N < 20); /\*方法语法\*/

11. int numsCount=(from n in numbers /\*两种方式相结合\*/

12. where n < 20

13. select n).Count();

14. foreach( var nums in numsQuery)

15. Console.WriteLine(nums);

16. foreach(var x in numsMethod) Console.WriteLine(x);

17. Console.WriteLine(numsCount);

18. }

19. }

20.

查询变量

LINQ查询可以返回两种类型的结果，可以是一个枚举（可枚举的一组数据，不是枚举类型），他是满足查询参数的项列表，也可以是一个叫做标量的但一值，他是满足查询条件的结果的某种摘要形式

1. class Program

2. {

3. static void Main()

4. {

5. int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

6. IEnumerable<int> lowNms= from n in numbers /\*返回一个枚举器\*/

7. where n<20

8. select n;

9. int numsCunt = (from n in numbers /\*返回一个整数\*/

10. where n < 20

11. select n).Count();

12. //左边的lowNums和numsCount叫做查询变量

13.

14. }

15. }

16.

## 查询表达式的结构

子句必须按照一定的顺序出现

from子句和select group子句这两部分是必须的

其他子句是可选的

在LINQ查询表达式中。select子句在表达式的最后，这与sql的select语句在查询的开始处不一样

可以有任意多的from let where子句

## from子句

1. from Type Item in Items

## Join子句

Join Identifier in Collection on field1 quals fields

1. var query =from s in students

2. join c in studentsInCourses on s.StID equals c.StID

3.

查询主题中from let where片段

1. class Program

2. {

3. static void Main()

4. {

5. var groupA = new[] { 3, 4, 5, 6 };

6. var groupB = new[] { 6, 7, 8, 9 };

7. var someInts=from a in groupA

8. from b in groupB

9. where a>4&&b<=8

10. select new {a,b,sum=a+b};

11. var someInts1 = from a in groupA

12. from b in groupB

13. let sum=a+b

14. where sum==12

15. select new { a, b, sum };

16. foreach (var someInt in someInts)

17. {

18. Console.WriteLine(someInt);

19. }

20. foreach (var someInt in someInts1)

21. {

22. Console.WriteLine(someInt);

23. }

24.

25. }

26. }

27.

## Where语句

## orderby子句

orderby子句接受一个表达式并根据表达式的顺序返回结果项

## select group子句

group子句根据制定的标准堆选择的对象进行分组

1. var query=from student in students

2. group student by student.Major

3.

查询延续：into子句

查询延续子句可以接受查询的一部分结果赋予一个名字，从而可以在查询的另一部分中使用

## 标准查询运算符

标注查询运算符由一系列的API方法组成，API呢个够让我们查询任何.NET数组与集合

标准查询运算符使用方法语法

一些运算符会返回Ienuerable对象（或其他序列），而其他运算符返回标量，返回标量的运算符立即执行查询，返回一个值，而不是一个可枚举类型。ToArray（），ToList（）等ToCollection运算符也会立即执行。

LINQ预定义的委托类型

1. //接受单个T类型的方法参数并返回一个bool类型的值，这种形式的委托成为谓词

2. public static int Count<T>(this IEnumerable<T> enumerable, Func<T, bool> predicate);

3.

# 事件

1. using System.Timers;

2. namespace EventExample

3. {

4. internal class Program

5. {

6. private static void Main(string[] args)

7. {

8.

9. System.Timers.Timer timer = new System.Timers.Timer();/\*事件的拥有者\*/

10. timer.Interval = 1000;

11. Boy boy=new Boy();

12. timer.Elapsed += boy.Action; /\*事件是Elapsed，事件的响应者是Boy，事件的处理器是Action\*/

13. timer.Start();

14. Console.ReadLine();

15. Console.WriteLine("Hello, World!");

16. }

17. }

18. class Boy

19. {

20. internal void Action(object? sender, ElapsedEventArgs e)

21. {

22. Console.WriteLine("大家好，我是热爱");

23. }

24. }

25. class Girl

26. {

27.

28. }

29. }

30.

1. using System;

2. using System.Collections.Generic;

3. using System.Linq;

4. using System.Text;

5. using System.Threading.Tasks;

6. using System.Windows.Forms;

7. namespace EventExample2

8. {

9. internal class Program

10. {

11. static void Main(string[] args)

12. {

13. Form form = new Form();

14. Controller controller = new Controller(form);

15. form.ShowDialog();

16. }

17. }

18. class Controller

19. {

20. private Form form;

21. public Controller(Form form)

22. {

23. if(form == null)

24. {

25. this.form = form;

26. this.form.Click += this.FormClicked;

27. }

28. }

29.

30. private void FormClicked(object sender, EventArgs e)

31. {

32. this.form.Text = DateTime.Now.ToString();

33. }

34. }

35. }

36.

1. using System;

2. using System.Collections.Generic;

3. using System.Linq;

4. using System.Text;

5. using System.Threading.Tasks;

6. using System.Windows.Forms;

7. namespace EventExample2

8. {

9. internal class Program

10. {

11. static void Main(string[] args)

12. {

13. Myform form = new Myform();

14. form.Click += form.FormCliked;

15. form.ShowDialog();

16. }

17. }

18. class Myform : Form

19. {

20. internal void FormCliked(object sender, EventArgs e)

21. {

22. this.Text = DateTime.Now.ToString();

23. }

24. }

25. }

26.

# 事件详解

事件的本质是来封装委托的

## 自定义事件完整声明

1. using System;

2. using System.Collections.Generic;

3. using System.Linq;

4. using System.Text;

5. using System.Threading;

6. using System.Threading.Tasks;

7.

8. namespace ConsoleApp1

9. {

10. internal class Program

11. {

12. static void Main(string[] args)

13. {

14. Customer customer=new Customer();

15. Waiter waiter=new Waiter();

16. customer.Order+=waiter.Action;

17. customer.Action();

18. customer.PayTheBill();

19. }

20. }

21. public class OrderEventArgs:EventArgs

22. {

23. public String DishName { get; set; }

24.

25. public String Size { get; set; }

26. }

27.

28. public delegate void OrderEventHander(Customer customer, OrderEventArgs e);

29. public class Customer

30. {

31. private OrderEventHander orderEventHander;

32. public event OrderEventHander Order

33. {

34. add

35. {

36. this.orderEventHander += value;

37. }

38. remove

39. {

40. this.orderEventHander -= value;

41. }

42. }

43. public double Bill { get; set; }

44. public void PayTheBill()

45. {

46. Console.WriteLine("I will Pay${0}",this.Bill);

47. }

48.

49. public void WaliIn()

50. {

51. Console.WriteLine("Wali in the restaurant");

52. }

53.

54. public void SitDown()

55. {

56. Console.WriteLine("Sit down");

57. }

58.

59. public void Think()

60. {

61. for (int i = 0; i < 6; i++)

62. {

63. Console.WriteLine("Let Me Think");

64. Thread.Sleep(1000);

65. }

66. if (this.orderEventHander!=null)

67. {

68. OrderEventArgs e = new OrderEventArgs();

69. e.DishName = "今天想吃红烧肉";

70. e.Size = "large";

71. this.orderEventHander.Invoke(this, e);

72. }

73. }

74.

75. public void Action()

76. {

77. Console.ReadLine();

78. this.WaliIn();

79. this.SitDown();

80. this.Think();

81.

82. }

83. }

84. public class Waiter

85. {

86. public void Action(Customer customer, OrderEventArgs e)

87. {

88. Console.WriteLine("I will serve you the dish-{0}",e.DishName);

89. double price = 10;

90. switch (e.Size)

91. {

92. case "small":

93. price = price \* 0.5;

94. break;

95. case "large":

96. price = price \* 1.5; break;

97. default:

98. break;

99. }

100. }

101. }

102. }

103.

简化声明

1. public event OrderEventHander Order;

## 委托会存在误用

1. using System;

2. using System.Collections.Generic;

3. using System.Linq;

4. using System.Text;

5. using System.Threading;

6. using System.Threading.Tasks;

7.

8. namespace ConsoleApp1

9. {

10. internal class Program

11. {

12. static void Main(string[] args)

13. {

14. Customer customer=new Customer();

15. Waiter waiter=new Waiter();

16. customer.Order+=waiter.Action;

17. //customer.Action();

18. OrderEventArgs e=new OrderEventArgs();

19. e.Size = "large";

20. e.DishName = "Manhanquanxi";

21.

22. OrderEventArgs e2 = new OrderEventArgs();

23. e2.Size = "large";

24. e2.DishName = "Beer";

25.

26. Customer badGay=new Customer();

27. badGay.Order += waiter.Action;

28. badGay.Order.Invoke(customer,e);

29. badGay.Order.Invoke(customer,e2);

30. }

31. }

32. public class OrderEventArgs:EventArgs

33. {

34. public String DishName { get; set; }

35.

36. public String Size { get; set; }

37. }

38.

39. public delegate void OrderEventHander(Customer customer, OrderEventArgs e);

40. public class Customer

41. {

42. public OrderEventHander Order;

43. public double Bill { get; set; }

44. public void PayTheBill()

45. {

46. Console.WriteLine("I will Pay${0}",this.Bill);

47. }

48.

49. public void WaliIn()

50. {

51. Console.WriteLine("Wali in the restaurant");

52. }

53.

54. public void SitDown()

55. {

56. Console.WriteLine("Sit down");

57. }

58.

59. public void Think()

60. {

61. for (int i = 0; i < 6; i++)

62. {

63. Console.WriteLine("Let Me Think");

64. Thread.Sleep(1000);

65. }

66. if (this.Order!=null)

67. {

68. OrderEventArgs e = new OrderEventArgs();

69. e.DishName = "今天想吃红烧肉";

70. e.Size = "large";

71. this.Order.Invoke(this, e);

72. }

73. }

74.

75. public void Action()

76. {

77. Console.ReadLine();

78. this.WaliIn();

79. this.SitDown();

80. this.Think();

81.

82. }

83. }

84. public class Waiter

85. {

86. public void Action(Customer customer, OrderEventArgs e)

87. {

88. Console.WriteLine("I will serve you the dish-{0}",e.DishName);

89. double price = 10;

90. switch (e.Size)

91. {

92. case "small":

93. price = price \* 0.5;

94. break;

95. case "large":

96. price = price \* 1.5; break;

97. default:

98. break;

99. }

100. }

101. }

102. }

103.

# 类

## Internal限制在项目内，是assemble级别的

Base只能向上访问一层using System;

1. using System.Collections.Generic;

2. using System.Linq;

3. using System.Text;

4. using System.Threading.Tasks;

5.

6. namespace ConsoleApp2

7. {

8. class Program

9. {

10. static void Main(string[] args)

11. {

12. Type t = typeof(Car);

13. Type tb = t.BaseType;

14. Car car = new Car();

15. Console.WriteLine(car.Owner);

16.

17. }

18. }

19. class Vehicle

20. {

21. public Vehicle()

22. {

23. this.Owner = "N/A";

24. }

25. public string Owner

26. {

27. get; set;

28. }

29. }

30. class Car : Vehicle

31. {

32. public Car()

33. {

34. this.Owner = "Car Owner";

35. }

36. }

37. class RaceCar : Car

38. {

39.

40. }

41. }

42.

## 类成员的访问级别是以类为上限

# 重写与多态

重写基于virtual->override

如果没有写Overide则会引用继承链的最新的一个。

# 抽象类的开闭原则

## 抽象类指的是函数成员没有被完全实现的类

## 在c#里面接口是完全未实现逻辑的类，只有函数成员，所有的成员都是抽象的public的

## 具体类-> 抽象类->接口 内部的内容越来越少

## 都不能实例化，只能用来声明变量，引用具体的类

# 接口，依赖反转，单元测试

