C#基本语法专题

厦门大学信息学院 赵江声

2023-09

目录

Content

01

装箱和取消装箱

值类型与Object类型的转换

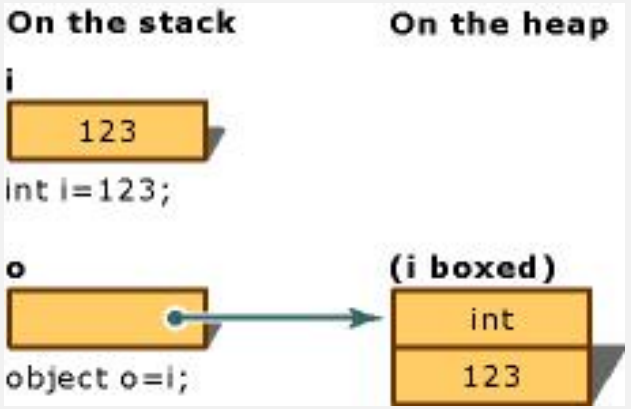
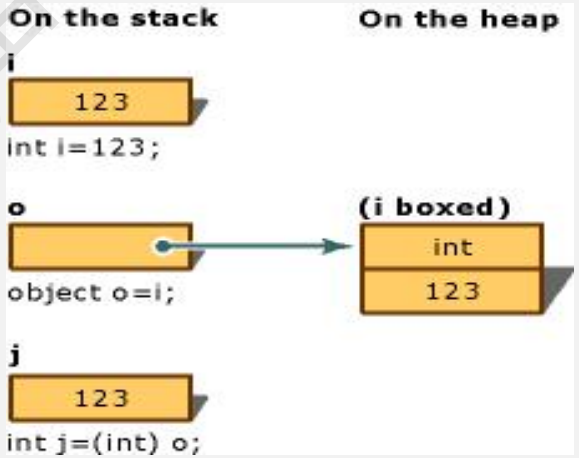
1.1 基本概念

参考资料：https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/types/boxing-and-unboxing

- 装箱是将值类型转换为 object 类型或由此值类型实现的任何接口类型的过程。
- 取消装箱将从对象中提取值类型。

```
1  //装箱和取消装箱例子
2  int i = 123;
3
4  object o = i;   //装箱
5  int j = (int)o;   //取消装箱
```


1.2 区别

装箱	取消装箱
	
隐式	显式
将一个值类型隐含转换成Object类型(根类型)	将对象类型显式地转换为一个值类型
装箱的时候，被装箱的值(i)拷贝一个副本赋给对象。	

1.3 样例

Program：Boxing

```
1  //值类型：存放在栈(Stack)内存中
2  int i = 123;
3
4  // 装箱：对值类型装箱会在堆中分配一个对象实例，
5  // 并将该值复制到新的对象中，注意是复制。
6  object o = i;
7
8  // 改变i的值：请问是改变哪里？ -----
9  // 只改变i在内存的值，而其在堆上的拷贝没有改变
10 i = 456;
11
12 //比较值类型（有修改）与其装箱的值
13 System.Console.WriteLine($"The value-type value = {i}");
14 System.Console.WriteLine($"The object-type value = {o}");
15
16 //这其实还是装箱操作
17 o = i;
18 System.Console.WriteLine($"The object-type value = {o}");
19
20 /* Output:
21     The value-type value = 456
22     The object-type value = 123
23     The object-type value = 456
24 */
```


02

C#中的字符串内插

2.1 基本概念

- 参考资料：https://learn.microsoft.com/zh-cn/dotnet/csharp/tutorials/string-interpolation
- C#中，特殊字符$ 将字符串文本标识为内插字符串 。 内插字符串是可能包含内插表达式的字符串文本 。 将内插字符串解析为结果字符串时，带有内插表达式的项会替换为表达式结果的字符串表示形式。
- 字符串内插是一种方便的字符串拼接方法，它允许将变量或表达式的值直接嵌入到字符串中。这种方法可以使得代码更加简洁易读，减少了手动拼接字符串的繁琐过程。
- 样例如下：

```
1 double CalCircleArea(double r) ⇒ (System.Math.PI * r * r);
2 double r1 = 1.8;
3 Console.WriteLine($"半径为{r1}的圆的面积是{CalCircleArea(r1)}。");
4 /*运行结果:
5 半径为1.8的圆的面积是10.17876019763093。
6 */
```


2.2 字符串拼接方法比较（1）

```
/* 字符串拼接方式比较
    有变量 userName, password。输出一条SQL语句，使其查询数据库的Users表，并且找出
    UserName字段与Password字段与输入相同的记录。
 */
string userName = "admin", password = "123";
```

```
// 方法1:
string sql1 = "SELECT * FROM Users WHERE UserName='" + userName + "' AND
                Password='" + password + "'";
Console.WriteLine(sql1);
// 缺点如下:
// 1. 因为字符串具有不可变性，所以每次字符串相加都需要产生一个新的临时对象。对于上例，一共需要先产生临时对象3个方可产生最后的结果。所以，字符串相加时，加号越多越麻烦
// 2. 在双引号附近的单引号字符或者双引号字符极容易漏写，并且不容易找出
// 3. 如果插入的字符串需要格式化，还需要用户自行处理显示格式
```


2.2 字符串拼接方法比较（2）

```
17 // 方法2:
18 string sql2 = string.Concat("SELECT * FROM Users WHERE UserName='", userName,
19             "' AND Password='", password);
20 Console.WriteLine(sql2);
21 // 方法2比方法1效率要高一些。但这个函数在输入的时候非常麻烦。实际编程时基本不会用到。
```

```
23 // 方法3:
24 string sql3 = string.Format("SELECT * FROM Users WHERE UserName='{0}' AND
25                 Password='{1}'", userName, password);
26 Console.WriteLine(sql3);
27 // 此方法在插入时不需要处理格式。因为{0}是即是格式字符串。
28 // 但使用此方法时，需要记忆每个参数出现的顺序。如果参数非常多，那就显的非常麻烦
```


2.2 字符串拼接方法比较（3）

```
30 // 方法4:
31 string sql4 = $"SELECT * FROM Users WHERE UserName='{userName}' AND
32                  Password='{password}'";
33 Console.WriteLine(sql4);
```


2.3 格式化（1）

- 练习：使用字符串内插构造格式化字符串　https://learn.microsoft.com/zh-cn/dotnet/csharp/tutorials/exploration/interpolated-strings

- 字符串内插为格式化字符串提供了一种可读性和便捷性更高的方式。它比“字符串复合格式设置”更容易阅读。

```
1  //指定格式字符串
2  {<interpolationExpression>:<formatString>}
3
4  //仅指定对齐方式;
5  //如果对齐方式值为正，则设置了格式的表达式结果为右对齐，如果为负，则为左对齐。
6  {<interpolationExpression>,<alignment>}
7
8  //同时指定对齐方式和格式字符串，则先从对齐方式组件开始
9  {<interpolationExpression>,<alignment>:<formatString>}
```

2.3 格式化（2）

字符串复合格式设置包括：

- String.Format，它返回格式化的结果字符串。
- StringBuilder.AppendFormat，它将格式化的结果字符串追加到 StringBuilder 对象。
- Console.WriteLine 方法的某些重载，它将格式化的结果字符串显示到控制台上。
- TextWriter.WriteLine 方法的某些重载，它将格式化的结果字符串写入流或文件中。 派生自 TextWriter 的类（如 StreamWriter 和 HtmlTextWriter）也共享此功能。
- Debug.WriteLine(String, Object[])，它将格式化消息输出到跟踪侦听器。
- Trace.TraceError(String, Object[])、Trace.TraceInformation(String, Object[]) 和 Trace.TraceWarning(String, Object[]) 方法，它们将格式化消息输出到跟踪侦听器。
- TraceSource.TraceInformation(String, Object[]) 方法，它将信息性方法写入跟踪侦听器中。

内插字符串支持“字符串复合格式设置功能”的所有功能。参考资料：

https://learn.microsoft.com/zh-cn/dotnet/standard/base-types/formatting-types

03

模式匹配

模式匹配

3.1 基本概念

- 参考资料：https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/functional/pattern-matching
- 模式匹配是一种测试表达式是否具有特定特征的方法。
  - “is 表达式”目前支持通过模式匹配测试表达式并有条件地声明该表达式结果。 返回一个bool值。
  - “switch 表达式”允许你根据表达式的首次匹配模式执行操作。根据switch分支返回对应类型的值。

- 知识点1：模式（Pattern）
  - 模式，指的是解决某一类问题的方法论。每个模式都描述了一个在环境中不断出现的问题，然后描述了解决该问题的核心。
- 知识点2：模式匹配
  - 找出一个实例是否满足模式，之后再对其进行处理。

3.2 常用模式（1）类型检测

- 检查对象是否为某一类型的行为称为 类型模式。本例演示判断元素的类型。

```
object[] A = new object[] { 1, "2", new List<char>() { 'a', 'b' }, 7.5, "abc"
};
foreach (var item in A)
{
    if (item is string str2)
    {
        Console.WriteLine($"{str2} is string");
    }
    else if (item is int int2)
    {
        Console.WriteLine($"{int2} is int");
    }
    else if (item is double double2)
    {
        Console.WriteLine($"{double2} is double");
    }
    else
    {
        Console.WriteLine($"{item} is object");
    }
}
```


3.2 常用模式（1）类型检测

- 上面的方式有些啰嗦，改为switch表达式，实现同样的功能

```
object[] A = new object[] { 1, "2", new List<char>() { 'a', 'b' }, 7.5, "abc" };
foreach (var item in A)
{
    string type = item switch
    {
        string a ⇒ "string",
        int a ⇒ "int",
        float a ⇒ "float",
        double a ⇒ "double",
        List<char> a ⇒ "list",
        _ ⇒ "object"
    };
    Console.WriteLine($"{item} is {type}");
}
```


3.2 常用模式（2）null检查

- 模式匹配最常见的方案之一是检查值是否为 null。

```
1 //本例判断值是否为null
2 int? x = 32;     // 可改为null，进行观察
3 string str3 = x is not null ? "not null" : "null";
4 Console.WriteLine(str3);
```


3.2 常用模式（3）关系模式

- 通过关系表达式判断值是否满足条件的行为称为 关系模式
- 本例判断数组中的点是否位于第一象限

```
1 //本例判断点是否在第一象限
2 Point[] points = { new Point(1, 1), new Point(0, 0), new Point(3, 4), new
  Point(-1, 4), new Point(-3, -2), new Point(5, 6) };
3 foreach (Point point in points)
4 {
5     Console.WriteLine($"Point({point.X},{point.Y}):{point is { X: > 0, Y: > 0
  }}");
6 }
```


3.2 常用模式（3）关系模式

```
1  //本例返回基于华氏温度的水的状态
2  string WaterState(int tempInFahrenheit) ⇒
3      tempInFahrenheit switch
4      {
5          (> 32) and (< 212) ⇒ "liquid",
6          < 32 ⇒ "solid",
7          > 212 ⇒ "gas",
8          32 ⇒ "solid/liquid transition",
9          212 ⇒ "liquid / gas transition",
10     };
```


3.2 常用模式（4）比较离散值

```
1  //本例Fib函数计算第n个的Fib数；并显示前15个。
2  int Fib(int n)
3  {
4      if (n is < 1)
5      {
6          throw new ArgumentOutOfRangeException("n", "n必须大于1");
7      }
8      return n switch
9      {
10         1 ⇒ 1,
11         2 ⇒ 1,
12         _ ⇒ Fib(n - 1) + Fib(n - 2)
13     };
14 }
15 List<int> list = new List<int>();
16 for (int i = 1; i ⩽ 15; i++)
17 {
18     list.Add(Fib(i));
19 }
20 Console.WriteLine(string.Join(",",list));
```


04

delegate

委托

4.1 基本概念

- 参考资料：https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/delegate-operator
- delegate 运算符创建一个可以转换为委托类型的匿名方法。

```
1 Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
2 Console.WriteLine(sum(3, 4));  // output: 7
```


4.2 对delegate的理解（1）

1. 映射函数

- 对于一个集合A，定义名为映射的操作：B = {y | y = f(x), x∈A}。将B称为映射结果，f(x)称为映射函数。
- 如何使用计算机程序实现该操作？
  - 需要定义函数：集合 Map：(集合 A， 映射函数 f) --> 集合B;
  - 对于C/C++而言，映射函数需要使用函数指针，而在C#中，则需要使用委托。

```
1 // C# 定义法
2 public delegate int   Func (int x); // int (*Func)(int x); // C/C++定义法
```


4.2 对delegate的理解（2）

1. 实现映射关系的方法1

- 定义集合A
- 定义函数Eg1F，并定义委托delegate，由委托指向函数
- 实现从集合A到集合B的Map函数Eg1Map

```
// 映射函数
public static int Eg1F(int c)
{
  return 2 * c;
}

public delegate int delegateFunc(int x);
public static List<int> Eg1Map(List<int> A, delegateFunc f)
{
    List<int> results = new List<int>();
    foreach (var item in A)
    {
      results.Add(f(item));
    }
    return results;
}
```

```
 // 例1: 将集合A = {1, 2, 3, 4, 5}映射为集合B。映射函数为 f(x) = 2x;
List<int> A = new List<int> { 1, 2, 3, 4, 5 };

//方法1: 定义Eg1F函数 和 delegateFunc, 实现 集合A到集合B的Map函数 Eg1Map
delegateFunc df = new delegateFunc(Eg1F);
var B1 = Eg1Map(A, df);
Console.WriteLine($"B1 = {{ {string.Join(',', B1)} }}");
```

4.2 对delegate的理解（3）

1. 方法2是对方法1简单改进

   - 可以将Eg1F函数 和 delegateFunc 的定义合并成 deleEg1F

```
//方法2: 小小的改进, 可以将Eg1F函数 和 delegateFunc 的定义合并成 deleEg1F
var B2 = Eg1Map(A, deleEg1F);
Console.WriteLine($"B2 = {{ {string.Join(',', B2)} }}");
```

```
        //   匿名函数使用方法:
        static delegateFunc deleEg1F = delegate (int x)
        {
            return 2 * x;
        };
```

4.2 对delegate的理解（4）

1. 方法3是对方法2的继续改进

   - 将deleEg1F的定义放到Eg1Map的参数中

```
1 //方法3：以上代码可以简化为：
2 var B3 = Eg1Map(A, delegate(int x)
3                    {
4                      return 2 * x;
5                    });
6 Console.WriteLine($"B3 = {{ {string.Join(',', B3)} }}");
```


4.2 对delegate的理解（5）

1. 方法4是对方法3的继续改进

   - 将deleEg1F改为Lambda表达式

```
// 观察这个结构: delegate(int x) {return 2 * x} 其中，delegate关键字也没有实际用处，将其改变为 ⇒ 运算符
// (int x) ⇒ {return 2 * x} 在C#中，当且仅当函数体只有一句，并且这一句是return语句时，可以省略花括号和return。所以再简化为
// (int x) ⇒ 2 * x 在C#中，当且仅当参数表可以由委托推定类型时，可以不写参数类型
// (x) ⇒ 2 * x 在C#中，当且仅当参数表只有一个，并且类型可以由委托推定时，可以不写圆括号
// x ⇒ 2 * x 以上三种写法在C#中，均称为Lambda表达式。该表达式的值是委托

var B4 = Eg1Map(A, x ⇒ 2 * x);
Console.WriteLine($"B4 = {{ {string.Join(',', B4)} }}");
```

4.3 转换为Action和Func类型

- 匿名方法可以转换为为 System.Action 和 System.Func<TResult> 等类型，用作许多方法的参数

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```


05

Lambda表达式

Lambda表达式

5.1 基本概念

- 参考资料：https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/lambda-expressions
- 使用 Lambda 表达式来创建匿名函数。 可以分为表达式Lambda和语句Lambda。

表达式Lambda语法:

```
(input-parameters) ⇒ expression
```

样例:

```
(int a ,int b) ⇒ a + b
```

语句Lambda语法:

```
(input-parameters) ⇒ { <sequence-of-statements> }
```

语句 Lambda 与表达式 Lambda 类似，只是语句括在大括号中。

5.2 Lambda 表达式与委托类型

- 任何 Lambda 表达式都可以转换为委托类型。 Lambda 表达式可以转换的委托类型由其参数和返回值的类型定义。
- 如果 lambda 表达式不返回值，则可以将其转换为 Action 委托类型之一；例如：有 2 个参数且不返回值的 Lambda 表达式可转换为 Action<T1,T2> 委托。

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

- 否则，可将其转换为 Func 委托类型之一。 例如：有 1 个参数且不返回值的 Lambda 表达式可转换为 Func<T,TResult>委托。

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

5.3 lambda 表达式的输入参数

```
1  //Lambda表达式参数
2  //1、没有返回值，且没有参数的情况，使用空括号指定零个输入参数:
3  Action acLine1 = () ⇒ Console.WriteLine("Hello, World!");
4  acLine1();
5
6  //2、没有返回值，但有参数的情况:
7  Action< string > acLine2 = (string s) ⇒ Console.WriteLine(s);
8  acLine2("Hello, World! Hello, World!");
9
10 //3、有返回值，且没有参数的情况:
11 Func<int> func1 = () ⇒ 888;
12 Console.WriteLine($"{func1()}");
13
14 //4、有返回值，有参数的情况，括号可选:
15 Func<int,int> func2 = x ⇒ 8 * x;
16 Console.WriteLine($"{func2(111)}");
17
18 //输入参数类型必须全部为显式或全部为隐式；否则，便会生成 CS0748 编译器错误。
19 //5、有返回值，有参数的情况，括号可选:
20 Func<int, int> func3 = (_) ⇒ 88;
21 Console.WriteLine($"{func3(3)}");
```


谢谢！