30分钟掌握 C#6

2017-04-12 雨夜潇湘 [dotNET跨平台](http://mp.weixin.qq.com/s/3avc1WqBi-8ZK0cHHb7FzA##)

1. 只读自动属性（Read-only auto-properties）

 C# 6之前我们构建只读自动属性：

public string FirstName { get; private set; }  
  
 public string LastName { get; private set; }

原理解析：就是编译器在生成set访问器时，它的修饰符是private，由上可知所谓的只读只是针对类外部，在类内部还是可以随意修改属性值的。

 C# 6中提供了真正的只读自动属性，写法如下：

public string FirstName { get; }  
 public string LastName { get; }

 原理解析：首先编译器会生成一个readonly的私有字段而get访问器就是返回该字段的值，由上可知该只读自动属性只能在构造函数中为其赋值。

2. 自动属性初始化器（Auto-Property Initializers）

 以前自动属性的赋值操作我们只能写在方法中，如构造函数：

public Student(string firstName, string lastName)  
 {  
 FirstName = firstName;   
 LastName = lastName;  
}

 但在C# 6中我们可以把赋值操作当作声明的一部份，如下所示：

public string FirstName { get; set; } = "Corleone";  
   public string LastName { get; set; } = "Mike";

 备注：其实C# 6和之前的版本都一样赋值操作最终都是在方法中完成，但后者明显更简洁直观，所以这是个不错的语法糖。

3. 函数成员的表达式体（Expression-bodied function members）

 C# 6中提供的一个新语法：对于只有一条语句的方法体可以简写成表达式。如下面两种情况：

 1. 方法（Methods）

  public Student Create() => new Student();

 等同于：

public Student Create()  
 {  
 return new Student();  
 }

 2. 只读属性（read only properties）

public string FullName => string.Format("{0},{1}", FirstName, LastName);

 等同于：

public string FullName

  {

      get

      {

          return string.Format("{0},{1}", FirstName, LastName);

      }

  }

原理解析：上面的表达式在编译后会生成最原始的方法体和访问器，值得一提的是函数表达式体跟Lambda是两个相似但不相同的东西，函数的表

达式体只能带一句话且不能包含return关键字但Lambda 能带语句块和包含关键字。

4. 使用静态（using static）

 C# 6中的一个新语法：使用类型的静态成员时可以省略其类型，如下所示：

using static System.String;   
// 先导入对应成员类型  
public bool IsNull(string str) => IsNullOrEmpty(str);

 等同于：

public bool IsNull(string str) => string.IsNullOrEmpty(str);

总结：该语法糖的目的是使代码变得更简洁，但这个应该是区分使用场景的，如：数学计算（Math类）使用此语法糖的确能够简洁代码提高可读

性，但在某处如果导入过多的类型那么不仅不能提高阅读性反而会增加阅读难度，因为你不知道这些成员具体属于那个类型。还有若类型本身存在

同名成员使用时则会使用类型成员覆盖。

注意：使用静态这一语法糖并不适用扩展方法，因为扩展方法的设计理念就是不修改已有代码且只能在一定范围内使用，所以在特殊情况下需要将

其当作静态方法来使用，那么使用类名调用反而是比较明智的。

5. Null条件运算符（Null-conditional operators）

 稍有经验的童鞋都知道在Coding过程中经常要判断变量的值是否为null，类似这种if-else的操作还不少。这使得代码看起来十分不简洁，好在C#6

 中提供了解决方法：

var student = new Student();  
 var firstName = student?.FirstName;

等同于：

var student = new Student();

  string firstName = null;

  if (student != null)

  {

      firstName = student.FirstName;

  }

使用方法：只需替换成员访问符 . 为 ?. ，若 ?. 左边为null则整个运算符的结果也为null，否则运算符的结果就等于其成员值。假如成员的类型为值

类型则整个表达式返回的类型是对应类型的可空类型，如：

1   int? age = student?.Age;

 原理解析： ?. 编译后就是 if 或 三元运算符，非赋值操作（如：call）会编译成 if，赋值操作一般会编译成三元运算符。

6. 字符串插值（String Interpolation）

 C# 6中提供了一种新语法来构建格式化字符串，如：

1   var fullName = $"{student.FirstName},{student.LastName}";

 等同于：

var fullName = string.Format("{0},{1}", student.FirstName, student.LastName);

 使用方法：只需在字符串前加上$符号，然后在大括号中填写表达式（字段、方法、Lambdad...）即可。

 备注：

 1. 字符串插值语法支持以前所有字符串格式设置（此项仅支持 .net framework，不支持 .net core 1.0.1），如：

Console.WriteLine($"平均成绩：{student.GPA:F2}");

 注：因为 : 总被编译器解释为表达式与字符串格式的分隔符，所以表达式中若有条件运算符则我们需要用括号来强制编译将其解析成当前语境所要

 表达的意义。如：

Console.WriteLine($"平均成绩：{(student.GPA > 80 ? student.GPA : 0):F2}");

 2. 字符串插值语法可以嵌套，如：

var score = $"我的各科成绩：{ $"数学：{student.MathScores}；英语：{student.EnglishScore};"}";

 原理解析：$"xxx{expression1}xxx{expression2}..." 编译后就是string.Format()。

7. 异常过滤器（Exception Filters）

 C# 6中的一个新功能就是异常过滤器，它可以使我们在恰当的时机来应用Catch子句，如：

try

  {

      throw new WebException("Request timed out..", WebExceptionStatus.Timeout);

  }

  catch (WebException webEx) when (webEx.Status == WebExceptionStatus.Timeout)

  {

      // Exception handling

  }

使用方法：try-catch() when()。

 总结：异常过滤器最大的亮点就是在使用恰当的情况下可以不丢失异常引发点的堆栈信息，这对程序的排错至关重要。另外它还有很多有意思的用

 法，大家可以上网查下。

8. nameof表达式(nameof Expressions)

public string FullName(Student student)

  {

      if (student == null)

          throw new ArgumentNullException(nameof(student));

      return $"{student.FirstName},{student.LastName}";

  }

优点：nameof 表达式它能够理解成员，当成员被重命名时nameof表达式中也重命名了，而常量字符串表示法是没有这样的优势。

缺点：nameof 表达式生成的是不完全限定名，若你需要完全限定名 nameof 就不能帮你了。

原理解析：nameof 是编译期间就确定其（成员）字符串名称的，即编译后就是常量字符串的表现形式了。

9. 在Catch和Finally中使用Await（Await in Catch and Finally blocks）

 C# 5 提供的 async 和 await 使异步编程变得极为简便，但它们也有着局限性：await在catch和finally块中不能使用。但这个问题已在C# 6中得到

 了解决，如：

public static async Task<string> MakeRequestAndLogFailures()

  {

      await logMethodEntrance();

      try

      {

          // ....

          var responseText = await streamTask;

          return responseText;

      }

      catch (System.Net.Http.HttpRequestException e) when (e.Message.Contains("301"))

      {

          await logError("Recovered from redirect", e);

          return "Site Moved";

      }

      finally

      {

          await logMethodExit();

      }

  }

10. Index初始化器（Index Initializers）

 这个功能并没有什么新意，其实以前就支持集合/字典 初始化器了，如：

var list = new List<string>()

  {

      "Mike",

      "Jim"

  };

  var dic = new Dictionary<int, string>()

  {

      { 20, "Mike" },

      { 30, "Jim"  }

  };

在C# 6中只是字典初始化器支持了新的写法，如：

var dic = new Dictionary<int, string>()  
 {  
 [20] = "Mike",  
 [30] = "Jim"  
 };

总结：暂无发现特殊的用法。

11. 改进的重载解析—编译器（Improved overload resolution）

  这算不上是新语法，因为仅仅是编译器的改进，之所以一提是想让大家知道有这么一回事。以前的编译器是识别不了 Task.Run(Func<Task>())

  的，如下：

static Task DoThings()

  {

      return Task.FromResult(0);

  }

  Task.Run(DoThings);  // 此处省略方法代码...

上述代码在老版本编译器下是编译不通过的，而在新版本编译器是能编译通过的。

备注：值得一提的是新版本编译器也只是识别了Task.Run(Func<Task>())，Task.Run(Action) 还是识别不了，总的来说此功能对我们用处不大，

还不如乖乖的写回Lambda表达式。

原文地址：http://www.cnblogs.com/VVStudy/p/6426923.html