Francisco Hao

[公司名称][公司地址]

C#高级编程

目录

[第 1 章 c#编译执行过程 5](#_Toc427006893)

[第 2 章 命名规则 6](#_Toc427006894)

[2.1 代码命名规范 7](#_Toc427006895)

[第 3 章 异常处理 7](#_Toc427006896)

[3.1 什么是异常 7](#_Toc427006897)

[3.2 错误种类 7](#_Toc427006898)

[3.3 try…catch…finally 8](#_Toc427006899)

[3.4 catch的三种形式 9](#_Toc427006900)

[3.5 异常类 12](#_Toc427006901)

[3.6 抛出异常（throw） 13](#_Toc427006902)

[3.6.1 手动抛出异常 13](#_Toc427006903)

[第 4 章 枚举 13](#_Toc427006904)

[4.1 枚举声明与底层值 14](#_Toc427006905)

[4.2 枚举类型变量 14](#_Toc427006906)

[4.3 枚举的使用技巧 15](#_Toc427006907)

[4.4 设置底层类型和显示值 16](#_Toc427006908)

[4.5 隐式成员编号 16](#_Toc427006909)

[第 5 章 枚举数和迭代器 17](#_Toc427006910)

[5.1 枚举数（enumerator） 17](#_Toc427006911)

[5.2 IEnumerator和IEnumerable接口 18](#_Toc427006912)

[5.3 迭代器 21](#_Toc427006913)

[第 6 章 集合 22](#_Toc427006914)

[6.1 ArrayList 22](#_Toc427006915)

[6.1.1 添加元素 22](#_Toc427006916)

[6.1.2 删除元素 23](#_Toc427006917)

[6.1.3 Contains（包含） 26](#_Toc427006918)

[6.1.4 ToArray（转换为数组） 26](#_Toc427006919)

[6.1.5 Sort（排序） 26](#_Toc427006920)

[6.1.6 习题 29](#_Toc427006921)

[6.2 Queue类 30](#_Toc427006922)

[6.2.1 入队 31](#_Toc427006923)

[6.2.2 出队 31](#_Toc427006924)

[6.3 Stack类 32](#_Toc427006925)

[6.3.1 入栈 32](#_Toc427006926)

[6.3.2 出栈 32](#_Toc427006927)

[6.4 HashTable类 33](#_Toc427006928)

[6.4.1 哈希表的概念及作用 33](#_Toc427006929)

[6.4.2 添加和访问元素 36](#_Toc427006930)

[6.4.3 删除哈希表元素 37](#_Toc427006931)

[6.4.4 遍历元素 38](#_Toc427006932)

[6.5 泛型集合类 38](#_Toc427006933)

[6.5.1 List<T> 40](#_Toc427006934)

[6.5.2 Dictionary<K,V> 41](#_Toc427006935)

[6.5.3 习题 43](#_Toc427006936)

[第 7 章 委托 48](#_Toc427006937)

[7.1 委托类型声明 48](#_Toc427006938)

[7.2 委托类型的对象 49](#_Toc427006939)

[7.3 委托的调用 49](#_Toc427006940)

[7.4 委托中增加、删除方法 50](#_Toc427006941)

[第 8 章 事件 51](#_Toc427006942)

[8.1 发行者 51](#_Toc427006943)

[8.1.1 委托类型声明 52](#_Toc427006944)

[8.1.2 声明事件 52](#_Toc427006945)

[8.1.3 触发事件 52](#_Toc427006946)

[8.2 订阅者 53](#_Toc427006947)

[8.2.1 事件处理程序 53](#_Toc427006948)

[8.2.2 事件注册 53](#_Toc427006949)

[8.3 事件的完整流程 53](#_Toc427006950)

[第 9 章 预处理 55](#_Toc427006951)

[9.1 全局预处理 55](#_Toc427006952)

[第 10 章 using的用法 56](#_Toc427006953)

[10.1 using指令 56](#_Toc427006954)

[10.2 using别名 56](#_Toc427006955)

[10.3 using语句 57](#_Toc427006956)

[第 11 章 文件操作 58](#_Toc427006957)

[11.1 Path类 59](#_Toc427006958)

[11.1.1 案例：获取当前执行程序所在的路径 59](#_Toc427006959)

[11.2 Directory类 60](#_Toc427006960)

[11.2.1 删除目录、判断目录是否存在 60](#_Toc427006961)

[11.2.2 获取子目录、子文件 60](#_Toc427006962)

[11.2.3 创建、移动文件夹 61](#_Toc427006963)

[11.2.4 案例：递归实现目录树 61](#_Toc427006964)

[11.2.5 案例：创建目录树格式的txt浏览程序 62](#_Toc427006965)

[11.3 File类 63](#_Toc427006966)

[11.3.1 快速创建文件流 64](#_Toc427006967)

[11.4 FileStream(文件流) 65](#_Toc427006968)

[11.4.1 用流写入数据 65](#_Toc427006969)

[11.4.2 通过文件流进行拷贝 65](#_Toc427006970)

[11.4.3 简单的加密 66](#_Toc427006971)

[11.5 StreamReader和StreamWriter 67](#_Toc427006972)

[11.5.1 StreamWriter 67](#_Toc427006973)

[11.5.2 StreamReader 68](#_Toc427006974)

[11.5.3 案例：对职工工资文件进行处理，所有人的工资加倍然后输出到新的文件。 69](#_Toc427006975)

[11.5.4 案例：从文本文件读写客户资料 69](#_Toc427006976)

[11.6 压缩和解压缩 74](#_Toc427006977)

[第 12 章 对象序列化 75](#_Toc427006978)

[12.1 javascript序列化 75](#_Toc427006979)

[12.2 XML序列化 77](#_Toc427006980)

[12.3 二进制序列化 78](#_Toc427006981)

[12.3.1 二进制序列化的要求 79](#_Toc427006982)

[12.3.2 反序列化 81](#_Toc427006983)

[第 13 章 正则表达式 84](#_Toc427006984)

[13.1 元字符 84](#_Toc427006985)

[13.1.1 匹配 84](#_Toc427006986)

[13.1.2 限定 85](#_Toc427006987)

[13.1.3 断言 85](#_Toc427006988)

[13.2 简写表达式 86](#_Toc427006989)

[13.3 正则表达式转义符 86](#_Toc427006990)

[13.4 .Net中的正则表达式 87](#_Toc427006991)

[13.4.1 判断是否匹配（） 87](#_Toc427006992)

[13.4.2 提取 91](#_Toc427006993)

[第 14 章 程序诊断 94](#_Toc427006994)

[14.1 Stopwatch（秒表类） 94](#_Toc427006995)

[第 15 章 线程（Thread） 95](#_Toc427006996)

[15.1 进程、应用程序域、线程之间关系 95](#_Toc427006997)

[15.2 前台线程和后台线程 96](#_Toc427006998)

[15.3 进程 96](#_Toc427006999)

[15.4 线程执行流程 96](#_Toc427007000)

[15.5 暂停线程 97](#_Toc427007001)

[15.6 线程ID 97](#_Toc427007002)

[15.7 完整案例 97](#_Toc427007003)

[15.8 线程与winform 98](#_Toc427007004)

[15.9 跨线程调用 98](#_Toc427007005)

[15.10 案例：摇奖机 99](#_Toc427007006)

[15.11 进程池 100](#_Toc427007007)

[15.11.1 为什么要有连接池 100](#_Toc427007008)

[15.11.2 生产者和消费者的问题 101](#_Toc427007009)

[15.12 锁 102](#_Toc427007010)

[15.12.1 数据库中的锁优化 103](#_Toc427007011)

[15.12.2 线程锁 103](#_Toc427007012)

[15.13 线程池 105](#_Toc427007013)

[15.13.1 使用线程池 105](#_Toc427007014)

[15.13.2 线程池的优点 106](#_Toc427007015)

[15.13.3 什么时候用线程池？什么时候用手动创建线程 107](#_Toc427007016)

[15.13.4 最大、最小线程数 107](#_Toc427007017)

[15.14 异步委托 107](#_Toc427007018)

[15.14.1 普通异步调用委托 107](#_Toc427007019)

[15.14.2 有回调函数的异步委托 108](#_Toc427007020)

[15.14.3 回调函数取得异步委托方法的结果和回调函数的参数 110](#_Toc427007021)

[第 16 章 内存分布 111](#_Toc427007022)

[16.1 一个进程在内存中的布局 111](#_Toc427007023)

[16.2 内存分配方式 114](#_Toc427007024)

[第 17 章 垃圾回收 115](#_Toc427007025)

[17.1 垃圾回收机制 115](#_Toc427007026)

[17.2 内存分配 116](#_Toc427007027)

[17.3 代 116](#_Toc427007028)

[17.4 垃圾回收过程 117](#_Toc427007029)

[17.5 大对象 118](#_Toc427007030)

[17.6 手工进行回收 118](#_Toc427007031)

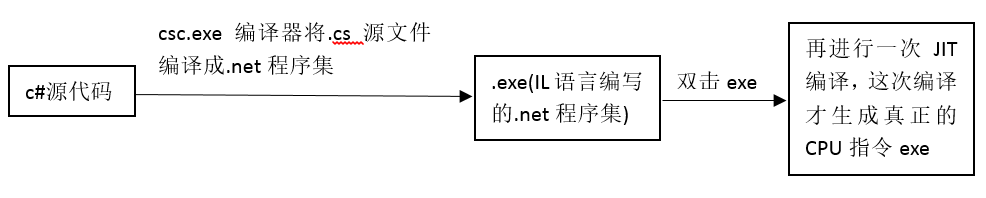
[第 18 章 程序集 119](#_Toc427007032)

[18.1 生成注释 119](#_Toc427007033)

[18.2 当前运行程序的程序集的位置 120](#_Toc427007034)

# c#编译执行过程

c#编译执行过程如下：



1. 编写C#源代码

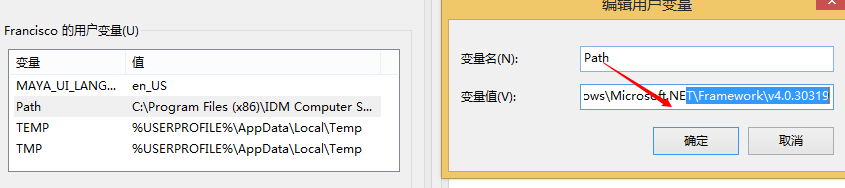
先用visual studio、ultraedit、sublime甚至记事本等文本编辑器，编写名为Program.cs的文本类型的源文件。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace 编译过程示例  {  class Program  {  static void Main(string[] args)  {  Console.WriteLine("Hello, World");  Console.ReadLine();  }  }  } |

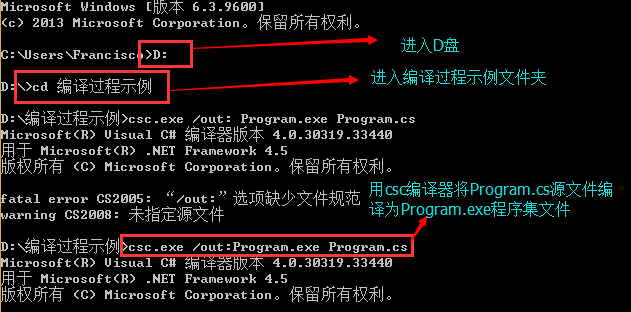
1. 编译成程序集文件

用csc.exe文件将源文件Program.cs（位于D盘的编译过程示例文件夹下）文件编译为名为Program.exe的.net程序集文件。

首先，把csc.exe添加到环境变量中，csc.exe的默认安装目录为：“C:\Windows\Microsoft.NET\Framework\v4.0.30319（这个是版本号）”。把目录添加到环境变量的path中即可。



之后打开cmd，输入以下的代码进行编译



这个编译好的exe文件并不是真正的操作系统可以理解的二进制文件，而是.net程序集，这个程序集实际上时把C#源代码编译成了MSIL源代码（IL语言）。

1. 双击生成的Program.exe运行。

当双击exe文件运行时，还会进行一次JIT即时编译（just in time），编译成CPU能识别的代码

1. 还可以用ngen.exe(位于于csc.exe相同的目录下)生成本机映像，并将它们安装到本地计算机上的本机映像缓存（C:\Windows\assembly\NativeImages\_v4.0.30319\_64）中。 运行时可从缓存中使用本机映像，而不必使用实时 (JIT) 编译器编译原始程序集。

使用方法：ngen.exe install Program.exe

# 命名规则

惯用法：属性开头字母大写，字段开头字母小写

常量名要大写

## 代码命名规范

|  |  |  |
| --- | --- | --- |
| 类型 | 规范 | 例子 |
| 字段、局部变量、 | 驼峰命名 | userName |
| 方法名、类名、属性名 | 帕斯卡命名 | GetMax() |

变量和方法的命名要有意义：

方法名：常用动词，描述方法的作用于功能。如Write(),Print(),Close(),GetUseId()。

变量名：按功能或方法的返回值命名：userName=GetUserName()。

# 异常处理

我们在编写程序时，总是希望程序安装自己的意愿工作，但是总可能发生意外，有些意外在编写代码的时候是检查不到的，更有些意外是因为程序所处的环境改变所致。为了提前预防意外的发生，C#提供了异常处理机制，用于处理程序的错误，体现了语言的健壮性。

## 什么是异常

异常是程序中的运行时错误，它违反了一个系统约束或应用程序约束，或出现了在正常操作时未预料的情形。例如，

* 当程序试图用0除一个数，
* 操作文件时文件没了，或权限不足
* 连接服务器时数据库服务器停电了

当这些发生时，系统捕获这个错误并抛出一个异常。如果程序没有提供处理该异常的代码，系统会挂起这个程序。

例如，下面的代码在试图用0除一个数时抛出一个异常。

|  |
| --- |
| int x = 10, y = 0;  int z = x / y;//用0除一个数时抛出一个异常。程序被挂起，不会继续执行下面的代码。  print(z);  print("hello"); |
| 运行时，console中显示下面的信息：问问 |
| DivideByZeroException: Division by zero |

## 错误种类

* 语法错误
* 逻辑错误
* 运行时错误

异常处理处理的就是运行时错误。

## try…catch…finally

try、catch和finally是可以标记和处理异常的关键字。它们都有一个关联的代码块。其基本结构如下：

|  |
| --- |
| try  {  ...//可能出现异常的代码。在try块中，如果某一句发生异常，就从该语句直接跳到catch块中，该语句后续的代码不再执行。  }  catch(<exceptionType> e)  {  ...//当try中代码发生异常时，执行这里的代码。代码常用来对异常进行处理的程序，如抛出异常或记录异常的日志。  }  finally  {  ...//无论try块中的代码是否异常，都会执行这里的代码。在这里进行善后处理，比如资源释放。一般用在文件与数据库读取操作等需要释放资源的地方。  } |

只有try和catch块（包含一个或多个catch块）时，finally块是可选的。只有try，没有catch时，finally就是必须的。

* try：包含可能会抛出异常的代码。
* catch：包含try抛出异常后将会执行的代码。省略参数，catch将响应所有类型的异常。也可以设置参数为响应特定的异常类型（如DivideByZeroException）。
* finally：包含总是会执行的代码。如果没有产生异常，则在try块之后执行，如果处理了异常，就在catch之后执行。

在try块的代码发生异常后，代码执行顺序是：

1. try块在发生异常的地方中断程序的执行。
2. 如果有catch块，就检查该快是否匹配已抛出的异常类型。如果没有catch块，就执行finally块。
3. 如果有catch块，但它与发生的异常类型不匹配，就检查是否有其他catch块。
4. 如果有catch块匹配已发生的异常类型，就执行它的代码，再执行finally块（如果有）。
5. 如果catch块都不匹配已发生的异常类型，就执行finally块。

下面的代码中演示了异常处理的基本结构：

|  |
| --- |
| int x = 10, y = 0;  try  {  int z = x / y;//用0除一个数时抛出一个异常。程序被挂起，不会继续执行下面的代码。  }  catch  {  print("catch到异常");  }  finally  {  print("未知的异常");  }  print("hello");  Console.ReadLine(); |
| 输出结果： |
| catch到异常  未知的异常  Hello |

## catch的三种形式

catch字句处理异常，它有三种形式，允许不同级别的处理。这些形式如下表：

|  |  |  |
| --- | --- | --- |
| catch形式 | 语法 | 用法 |
| 一把catch子句 | catch  {  statements;  } | * 在catch关键字之后没有参数列表 * 会匹配try块中引起的任何类型的异常。 |
| 特定catch子句 | catch(ExceptionType)  {  statements;  } | * 异常类（见16.4异常类）的名称作为唯一参数。 * 匹配任何该名称类型的异常。 |
| 带对象的特定catch子句 | catch(ExceptionType ex)  {  statements;  } | * 参数中异常类名称后面有一个异常变量。 * 异常变量指向异常对象，并能被用于访问关于该异常的信息 |

下面的我们通过实例演示，来学习这三种形式的用法。

|  |
| --- |
| 一般catch子句 |
| int x = 10, y = 0;  int z = 0；  try  {  int z = x / y;//用0除一个数时抛出一个异常。程序被挂起，不会继续执行下面的代码。  }  //一般cath子句能接受任何异常，但不能确定引发异常的类型。适用于对任何可能发生的异常的普通处理和清理  catch  {  print("catch到异常");  }  print(z.ToString()); |
| 输出结果： |
| Catch到异常  0 ----------------try中语句引发异常不执行。 |

|  |
| --- |
| 特定catch子句 |
| try  {  z = x / y;//用0除一个数时抛出一个异常。程序被挂起，不会继续执行下面的代码。  }  // 异常类作为参数。当发生DivideByZeroException异常时，会执行该catch中的语句。这种形式可以有多个catch，只有与引发的异常类型一致的参数中的子句会执行。  catch(System.DivideByZeroException)  {  print("Handling an exception");  }  // try语句块不会引发IndexOutOfRangeException异常，此catch块不会执行。  catch(System.IndexOutOfRangeException)  {  print("IndexOutOfRangeException");  }  print(z.ToString());  Console.ReadLine(); |
| 输出结果： |
| Handling an exception -----------------当发生DivideByZeroException异常使，引发该语句。  0 ----------------try中语句引发异常不执行。 |

|  |
| --- |
| 带对象的特定catch子句 |
| int x = 10, y = 0;  int z = 0;  try  {  z = x / y;//用0除一个数时抛出一个异常。程序被挂起，不会继续执行下面的代码。  }  // 带DivideByZeroException类型的异常变量  catch(System.DivideByZeroException ex)  {  //打印异常变量中的相关信息  print(ex.Message);  print(ex.Source);  print(ex.StackTrace);  }  catch(System.IndexOutOfRangeException)  {  print("IndexOutOfRangeException");  }  print(z.ToString());  Console.ReadLine(); |
| 输出结果： |
| 尝试除以零。---------------------------ex.Message的输出  ConsoleApplication9-------------------ex.Source的输出  在 ConsoleApplication9.Program.Main(String[] args) 位置 d:\Documents\Visual S  tudio 2013\Projects\ConsoleApplication5\ConsoleApplication9\Program.cs:行号 20-----------------------------------ex.StackTrace的输出  0-------------------------------------print(z.ToString())的输出 |

当可能引发的异常不确定时，可以使用Exception类型。例如下面的代码使用了exception类型，与上表中使用DivideByZeroException类型的代码在输出上是一致的。

|  |
| --- |
| int x = 10, y = 0;  int z = 0;  try  {  z = x / y;//用0除一个数时抛出一个异常。程序被挂起，不会继续执行下面的代码。  }  //不确定可能会引发什么异常，也不想用多个catch语句时，可以用exception，它包含各种类型的异常  catch(Exception ex)  {  //打印异常变量中的相关信息  print(ex.Message);  print(ex.Source);  print(ex.StackTrace);  } |

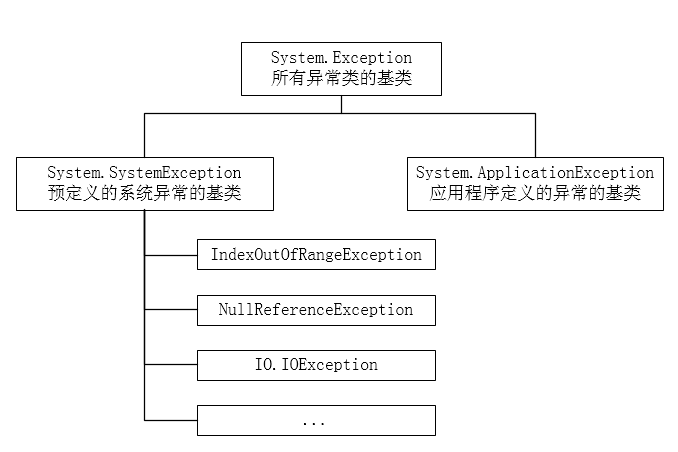
## 异常类

有许多不同的类型可以在程序中发生。C#中定义了许多类，每一个类代表一个指定的异常类型。

当一个异常发生时：

1. 创建该类型的异常对象
2. 寻找适当的catch子句以处理它

所有异常类都从根本上派生自System.Exception类。异常类继承的关系如下图：



异常对象包含详细描述异常的属性，其中，重要的属性如下：

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 描述 |
| Message | String | 提供异常起因的详细信息 |
| StackTrace | String | 描述异常发生在何处的信息 |
| Helplink | String | 可以被应用程序定义的异常设置，为异常原因信息提供URL |
| Source | String | 用于获取和设置导致错误的应用程序或对象的名称 |

## 抛出异常（throw）

### 手动抛出异常

只有在代码出现错误的时候，才能使用try…catch…finally语句捕获程序异常。其实，还能在代码中手动的抛出异常，方法是使用throw关键字。

例如，在控制台程序中读取输入的字符型数据，如果不能转换为数值型数据，则抛出异常。

|  |
| --- |
| Console.WriteLine("请输入数据");  string str = Console.ReadLine();  int number ;  if(int.TryParse(str, out number))  {  number++;  Console.WriteLine(number);  }  else  {  //抛出异常;与 throw new Exception ("请输入数字");相同  SystemException ex = new SystemException("请输入数字");  throw ex;  }      Console.ReadLine(); |

# 枚举

与结构和类一样，枚举是用户自定义类型。枚举是值类型，因此直接存储的是数据，而不是分开存储成引用和数据。

当变量的数量、值是一定的情况下，可以使用枚举。用枚举的好处是限定了变量的取值范围，程序处理起来比较方便。

## 枚举声明与底层值

下面的代码展示了一个示例，声明了一个名称为Weapons的枚举类型，它含有四个成员。注意成员声明列表是逗号分隔开的列表，在枚举声明中没有分号。

|  |
| --- |
| public enum Weapons { Pistol, Rifle, ChainGun, PlasmaCan}  **成员列表，以逗号隔开**  **枚举名称**  **关键字** |

每个枚举类型都有一个底层整数类型，默认为int。每个枚举成员会被赋值一个底层类型的常量值。编译器把第一个成员赋值为0,并对每一个后续成员赋的值比前一个成员多1。

例如，在上面的Weapons类型中，编译器把int值0、1、2、3分别赋值给成员Pistol, Rifle, ChainGun和PlasmaCan 。使用类型转换可以看到底层的成员值。

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class NewBehaviourScript1 : MonoBehaviour {  public int Ammo = 10;  public enum Weapons { Pistol, Rifle, ChainGun, PlasmaCan}  void Start()  {  int w1 = (int)Weapons.Pistol; //使用类型转换，可以看到枚举成员的底层值  int w2 = (int)Weapons.ChainGun;//枚举成员ChainGun的底层值赋值给变量w2  print(w1);//输出结果：0  print(w2);//输出结果：2  }  } |

## 枚举类型变量

可以把枚举值赋值给枚举类型变量。比如，下面的代码把枚举成员Rifle的值赋值给枚举类型变量SelectedWeapon。

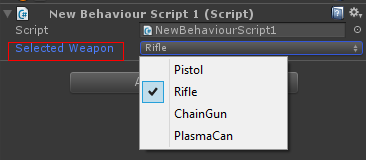
|  |
| --- |
| public Weapons SelectedWeapon = Weapons.Rifle; |

在Unity中，标记为public的枚举类型字段会显示在inspector视图中。

例如，我们声明以下的枚举类型变量：

|  |
| --- |
| public class NewBehaviourScript1 : MonoBehaviour {  public enum Weapons { Pistol, Rifle, ChainGun, PlasmaCan}  public Weapons SelectedWeapon;//声明枚举类型字段变量  } |

在inspector中，可以从SelectedWeapon右侧的下拉列表中选择枚举成员来为枚举类型变量Selected Weapons赋值。



这个操作类似于在脚本中为类型变量进行如下的赋值：

|  |
| --- |
| public Weapons SelectedWeapon = Weapons.Rifle; |

## 枚举的使用技巧

枚举类型使代码更具可读性，理解清晰，易于维护。

下面的示例中，选择不同的武器，会执行不同的代码。选择哪个武器后会执行什么，显得清晰明了。

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class NewBehaviourScript1 : MonoBehaviour {  public enum Weapons { Pistol, Rifle, ChainGun, PlasmaCan}  public Weapons SelectedWeapon ;    void Start()  {  switch (SelectedWeapon)  {  case Weapons.ChainGun:  GameObject.CreatePrimitive(PrimitiveType.Capsule);  break;  case Weapons.Pistol:  GameObject.CreatePrimitive(PrimitiveType.Cube);  break;  case Weapons.PlasmaCan:  GameObject.CreatePrimitive(PrimitiveType.Cylinder);  break;  case Weapons.Rifle:  GameObject.CreatePrimitive(PrimitiveType.Sphere);  break;  }  }  } |

再比如人的性别，可以分为男人、女人以及人妖。使用枚举可以使变量限制在这个范围内，不能再定义为其他的性别。如下代码：

|  |
| --- |
|  |

## 设置底层类型和显示值

枚举的底层类型默认是int类型。通过把冒号和类型名放在枚举之后，可以设置类型为int或byte类型[[1]](#footnote-1)。

下面的示例中，枚举的底层类型被设置为short。

|  |
| --- |
| enum Weapons:short { Pistol, Rifle, ChainGun, PlasmaCan} |

枚举成员常量的值可以是底层类型的任何值。在枚举成员声明的变量名之后使用初始化表达式，可以显示地设置一个成员的值。枚举成员的名称不能重复，但是它底层的值可以是重复的，如下所示。

|  |
| --- |
| public enum Weapons:byte{ Pistol = 1, //为成员显示设置值  Rifle = 5,  ChainGun = 4, //重复的值  PlasmaCan = 4 //重复的值  } |

## 隐式成员编号

在前面的例子中，我们知道编译器会把第一个成员赋值为0,并对每一个后续成员赋的值比前一个成员多1。如果我们只对部分成员显示赋值，那么没有赋值的成员会如何处理呢？

在C#中，隐式成员的值在前面成员的值上加上1。例如，下面的代码声明了一个枚举Weapons，并显示地为其设置了一些成员，而其他的接受隐式编号。

|  |
| --- |
| public enum Weapons:byte{ Pistol = 11, //11  Rifle, //12  ChainGun, //13  PlasmaCan = 1, //1  HuntingGUn, //2  stutterer //3  } |

# 枚举数和迭代器

在[foreach-in循环这一节中](#_foreach-in循环)，我们已经知道可以使用foreach语句来遍历数组中的元素。在本章中，我们会进一步探讨数组，来看看为什么它们可以被foreach语句处理。我们还会研究如何为用户自定义的类增加这个功能。最后，我们会讨论迭代器的使用。

## 枚举数（enumerator）

当我们为数组使用foreach语句时，这个语句为我们依次取出了数组中的每一个元素，允许我们读取它的值。

例如，下面的代码声明了一个有3个元素的数组，然后使用foreach来循环打印这些项的值：

|  |
| --- |
| int[] MyArr = new int[] { 1,2,3 };//声明数组  void Start () {  foreach (int arr in MyArr)//遍历数组中的元素  {  print(arr);  }    }  /\*输出结果：  \* 1  \* 2  \* 3  \*/ |

为什么数组可以这样做呢？这是因为数组可以按需提供一个叫做枚举数的对象。枚举数“知道”元素的次序并且会跟踪它在序列中的位置，然后返回请求的当前项。foreach会从枚举数中请求每一项并且把它作为迭代变量（iteration variable），来实现遍历。

对于有枚举数的类型而言，必须有一个方法来获取它们。C#中获取一个对象枚举数标准方法是调用对象的GetEnumerator方法。实现了GetEnumerator方法的类型叫可枚举类型。

数组就是个可枚举类型，可以用GetEnumerator方法获得枚举数对象。下面的代码中，调用GetEnumerator方法获得枚举数对象，并用IEnumerator接口中的方法和属性实现了类似foreach语句的遍历。

|  |
| --- |
| int[] MyArr = new int[] { 1,222,31};//声明数组  void Start () {  IEnumerator ie = MyArr.GetEnumerator();//调用可枚举类型的GetEnumerator方法获得枚举数对象  while (ie.MoveNext())//调用IEnumerator接口的MoveNext方法移到下一项。实现遍历数组  {  int i = (int)ie.Current;////调用IEnumerator接口的Current方法获取当前项。注意它返回的是object类型，需要强制转换类型。  print(i);  } |

## IEnumerator和IEnumerable接口

Ienumerator和Ienumerable接口用来创建自己的枚举数和可枚举类。

**IEnumerator** 是所有非泛型枚举数的基接口。一个类实现了IEnumerator接口，也就是实现Current属性，MoveNext和Reset方法。

最初，枚举数定位在集合中第一个元素前。Reset 方法还会将枚举数返回到此位置。在此位置，调用 Current 属性会引发异常。因此，在读取 Current 的值之前，必须调用 MoveNext 方法将枚举数提前到集合的第一个元素。

在调用 MoveNext 或 Reset 之前，Current 返回同一对象。MoveNext 将 Current 设置为下一个元素。

如果 MoveNext 越过集合的末尾，则枚举数将被放置在此集合中最后一个元素的后面，而且 MoveNext 返回 false。当枚举数位于此位置时，对 MoveNext 的后续调用也返回 false。如果最后一次调用 MoveNext 返回 false，则调用 Current 会引发异常。若要再次将 Current 设置为集合的第一个元素，可以调用 Reset，然后再调用 MoveNext。

而IEnumerable接口实现了GetEnumerator方法，该方法返回一个IEnumerator对象。一个类实现IEnumerable接口后，调用foreach语法的时候，会自动的调用GetEnumerator方法，然后在这个IEnumerator对象中遍历。

下面是使用IEnumerator和IEnumerable接口来实现枚举数的示例：

|  |
| --- |
| using UnityEngine;  using System.Collections;  /// <summary>  /// Color用来保存颜色的名称  /// </summary>  class Color  {  public string ColorName;  public Color(string colorName)  {  this.ColorName = colorName;  }  }  /// <summary>  /// 实现了IEnumerator接口，也就是实现了Current属性、MoveNext、Reset方法。用来返回枚举数的与位置相关的信息。  /// </summary>  class ColorEnum : IEnumerator  {  public Color[] Colors;  int Position = -1;  /// <summary>  /// 构造函数  /// </summary>  /// <param name="theColors"></param>  public ColorEnum(Color[] theColors)  {  Colors = new Color[theColors.Length];  for (int i = 0; i < theColors.Length; i++)  {  Colors[i] = theColors[i];  }  }  /// <summary>  /// 实现Current属性  /// </summary>  public object Current  {  get { return Colors[Position]; }  }  /// <summary>  /// 实现MoveNext方法，返回布尔值  /// </summary>  /// <returns></returns>  public bool MoveNext()  {  if (Position < Colors.Length - 1)  {  Position++;  return true;  }  else  {  return false;  }  }  /// <summary>  /// 重设位置  /// </summary>  public void Reset()  {  Position = -1;  }  }  /// <summary>  /// 实现了IEnumerable接口  /// </summary>  class Mycolors : IEnumerable  {  private Color[] \_color;  /// <summary>  /// 构造函数  /// </summary>  /// <param name="colorArray"></param>  public Mycolors(Color[] colorArray)  {  \_color = new Color[colorArray.Length];  for (int i = 0; i < colorArray.Length; i++)  {  \_color[i] = colorArray[i];  }  }  /// <summary>  /// 实现了GetEnumerator方法，可以用foreach来遍历。  /// </summary>  /// <returns></returns>  public IEnumerator GetEnumerator()  {  return new ColorEnum(\_color);  }  }  public class NewBehaviourScript2 : MonoBehaviour {  // Use this for initialization  void Start () {  Color[] c = new Color[3] { new Color("Red"), new Color("Green"), new Color("Blue") };    Mycolors mc = new Mycolors(c);//创建可枚举类型Mycolors的实例。    foreach (Color col in mc)//遍历枚举数并输出。  {  Debug.Log(col);  }    }    } |

## 迭代器

迭代器是C# 2.0中的新功能，有了它，我们就可以在自己的类或者结构中支持foreach迭代而不必实现整个IEnumerable接口，我们只需要提供一个迭代器，即可遍历类中的数据结构。当编译器检测到迭代器时，它将自动生成IEnumerable接口的Current、MoveNext和Dispose方法。

而迭代器代码使用yield return语句依次返回每个元素。yield break将中止迭代。到达yield return语句时，会保存当前迭代的位置，下次调用迭代器时将从此位置开始执行。

下面演示了如何使用迭代器来创建可枚举类型的类。

|  |
| --- |
| using UnityEngine;  using System.Collections;  /// <summary>  /// 使用迭代器创建可枚举类  /// </summary>  class MyClass  {  public IEnumerator GetEnumerator()  {  return Color();//返回枚举数  }  /// <summary>  /// 迭代器块  /// </summary>  /// <returns></returns>  public IEnumerator Color()  {  yield return "Red1";  yield return "Green";  yield return "Blue";  }  }  public class NewBehaviourScript2 : MonoBehaviour {  // Use this for initialization  void Start () {  MyClass mc = new MyClass();  foreach (string col in mc)//遍历枚举数并输出。  {  Debug.Log(col);  }    }    } |

# 集合

存储多个数据时，可以用数组把多个数据组合在一起。，但是数组长度是固定的，而且类型也是不可变的，只能存储相同类型的数据。但是，如果数组的元素不确定、或数组的数据类型不同，用数组就实现不了数据的集合。而使用集合类则能更有效地处理这些相关的数据。

在.net Framework中，有一个非常基础而重要的命名空间，那就是System.Collections。它包含了许多集合类，这些集合类表达的是一些在计算机科学中非常重要的数据结构。例如链表、堆栈、队列、哈希表等。此外，还提供了泛型集合System.Collections.Generic，提供更灵活的数据处理。

下面，我们学习几种可以直接使用的常用集合类

## ArrayList

ArrayList可以认为是可扩展的动态数组，即在数组的基础上添加了一些通用的操作，扩充了数组的功能。

与数组相比，ArrayList具有以下特点：

* 可以动态的添加元素。
* 可以动态删除元素。
* 可以插入元素。
* 创建的ArrayList的对象为object类型。

### 添加元素

Array提供了Add、AddRange和insert方法用于添加元素。下面的代码演示了如何创建ArrayList和添加元素的相关操作。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  string[] names = new string[] { "乔丹", "威廉姆斯", "科比" };  ArrayList arrlist = new ArrayList();  arrlist.Add("hyfdbd");//向ArrayList中添加yuans  arrlist.Add(30);  arrlist.AddRange(names);//可以添加数组，集合等实现了Icollection的类型到ArrayList中  arrlist.Insert(1, "age");//插入元素到第2个位置处  int capacity = arrlist.Capacity;//ArrayList中可以容纳的元素个数，会自动增长。增长速度为4的倍数。  int count = arrlist.Count;  int i = (int)arrlist[2] + 50;//Arraylist中的元素为object类型，需要cast为对应的类型进行操作  print(capacity);//打印结果为：4  print(count);//打印结果为：3  print(arrlist[1]);//打印结果为：age  print(i);//打印结果为：80  }  } |

### 删除元素

ArrayList提供了clear、remove和removeAt三个方法用于删除元素。下面的代码演示了如何删除元素。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  ArrayList arrlist = new ArrayList();  arrlist.Add("hyfdbd");  arrlist.Add(30);  arrlist.Add(10);  arrlist.Insert(1, "age");  print("arrlist的个数为:" + arrlist.Count);//arrlist的个数为:4  //----------------下面为删除操作-------------------  arrlist.Remove("hyfdbd");//通过元素值删除  print(arrlist.Count);//3  arrlist.RemoveAt(0);//通过下标移除元素  print(arrlist.Count);//2  arrlist.Clear();//清楚所有的元素  print(arrlist.Count);//0  print(arrlist.Capacity);//4  }  } |

#### 详解Remove

当移除数值型数据或字符串时，remove可以删除ArrList中内容，但是当类型为引用类型时，执行remove却不能删除，如下面代码：

|  |
| --- |
| class Person  {  public int Age { get; set; }    }  class Program  {  static void Main(string[] args)  {  Person P1 = new Person() { Age = 10 };  ArrayList arr = new ArrayList() { 1, 2, 4, 5, "hao", false,new Person(){Age = 10} };  arr.Remove(P1);  for (int i = 0; i < arr.Count; i++)  {  Console.WriteLine(arr[i]);  }  Console.ReadKey();  }  } |

这是因为什么呢?因为remove方法调用路Equal方法，而Equal方法在对比引用类型时，比较的是是否同一块内存，而不是内存中的值是否相同。因此，想要比较引用类型，可以重写Equal方法。

|  |
| --- |
| class Person  {  public int Age { get; set; }  //重写Equals  public override bool Equals(object obj)  {  Person p = obj as Person;  if(p!=null)  {  if(p.Age == this.Age)  {  return true;  }  else  {  return false;  }  }else  {  throw new Exception("转换失败");  }  }  }  class Program  {  static void Main(string[] args)  {  Person P1 = new Person() { Age = 10 };  ArrayList arr = new ArrayList() { 1, 2, 4, 5, "hao", false,new Person(){Age = 10} };  arr.Remove(P1);  for (int i = 0; i < arr.Count; i++)  {  Console.WriteLine(arr[i]);  }  Console.ReadKey();  }  } |

使用上面的代码，可以删除Arrlist中引用类型的数。

### Contains（包含）

判断数组中是否包含某个值，它也会调用Equals方法。

|  |
| --- |
| Person P1 = new Person() { Age = 10 };  ArrayList arr = new ArrayList() { 1, 2, 4, 5, "hao", false,new Person(){Age = 10} };      if(arr.Contains(1))  {  Console.WriteLine("包含1");  }  if (arr.Contains(P1))  {  Console.WriteLine("包含Person类型，切value为10");  }  for (int i = 0; i < arr.Count; i++)  {  Console.WriteLine(arr[i]);  } |

### ToArray（转换为数组）

|  |
| --- |
| ArrayList arr = new ArrayList() { 1, 2, 4, 5, "hao", false};  object[] obj = arr.ToArray();//转换为数组，类型为object  for (int i = 0; i < obj.Length; i++)  {  Console.WriteLine(obj[i]);  }  Console.ReadKey(); |

ToArray转换为object类型的数组。

### Sort（排序）

sort会以升序排序，对数值类型，按照从小到大；对于字符串类型，,从最左开始一个一个比,比它们的ASCII值，进行排序。

|  |
| --- |
| ArrayList arr = new ArrayList() { 1, 100, 4, 5,6,2};  ArrayList arr1 = new ArrayList() { "ahao", "whao", "Frank" };  arr.Sort();//排序，升序  arr1.Sort();//字符串排序，不区分大小写  for (int i = 0; i < arr.Count; i++)  {  Console.WriteLine(arr[i]);  }  for (int i = 0; i < arr1.Count; i++)  {  Console.WriteLine(arr1[i]);  } |

对于引用类型，又该如何排序呢？很简单，只需要实现IComparable接口（值类型和字符串都实现了IComparable）即可。

比如，要对学生类进行排序，我们以年龄进行偏虚，可以实现IComparable接口并实现CompareTo方法：

|  |
| --- |
| class Student:IComparable//实现IComparable接口  {  public int Age { get; set; }  public string Name { get; set; }  //实现接口中的方法  public int CompareTo(object obj)  {  Student s = obj as Student;  if(s != null)  {  return this.Age - s.Age;//升序，s.Age - this.Age为降序。小于0，表示当前实例在obj实例之后，等于0，在同一位置，大于0，在obj实例之前。  }else  {  throw new Exception("转换失败");  }  }  }  class Program  {  static void Main(string[] args)  {  ArrayList arr = new ArrayList() { new Student() { Name = "hao", Age = 20 }, new Student() { Name = "Frank", Age = 21 }, new Student() { Name = "john", Age = 18 } };  arr.Sort();  for (int i = 0; i < arr.Count; i++)  { Student s = arr[i]as Student;  Console.WriteLine(s.Age);  }  Console.ReadKey();  }  } |

还能用多态来进行排序，比如在上面的例子中，想要实现按照名字进行排序怎么办？

如果用IComparable接口实现，那么按照年龄排序又不能实现了，想要实现多种排序可以用IComparer（比较器）来实现。

|  |
| --- |
| class Student  {  public int Age { get; set; }  public string Name { get; set; }  }  //比较器，实现IComparer接口，按照姓名进行升序排列  class SortByName : IComparer  {  //实现接口的方法  public int Compare(object x, object y)  {  Student p1 = x as Student;  Student p2 = y as Student;  if (p1 != null && p2 != null)  {  return p1.Name.CompareTo(p2.Name);  }  else  {  throw new ArgumentException();  }  }  }  //比较器，实现IComparer接口，按照年龄进行升序排列  class SortByAge:IComparer  {  public int Compare(object x, object y)  {  Student s = x as Student;  Student s1 = y as Student;  if(s!=null && s1!=null)  {  return s.Age - s1.Age;  }else  {  throw new ArgumentException();  }  }  }  class Program  {  static void Main(string[] args)  {  ArrayList arr = new ArrayList() { new Student() { Name = "hao", Age = 20 }, new Student() { Name = "Frank", Age = 21 }, new Student() { Name = "john", Age = 18 } };  arr.Sort(new SortByName());//传入IComparer的实例。  for (int i = 0; i < arr.Count; i++)  {  Student s = arr[i] as Student;  Console.WriteLine(s.Name);  }  arr.Sort(new SortByAge());  for (int i = 0; i < arr.Count; i++)  {  Student s = arr[i] as Student;  Console.WriteLine(s.Age);  }  Console.ReadKey();  }  } |

实现IComparer接口就可以用多种方式进行排序了。

### 习题

1. 随机生成10个1-100之间的数放到ArrayList中，要求这10个数不能重复，并且都是偶数

|  |
| --- |
| ArrayList arr = new ArrayList();  Random random = new Random();  while (arr.Count < 10)  {  int n = random.Next(1, 101);//1-100  if (n % 2 == 0 && !arr.Contains(n))  {  arr.Add(n);  }  } |

1. 有一个字符串是用空格分隔的一系列整数，写一个程序把其中的整数重新排列打印出来：奇数显示在左侧、偶数显示在右侧。

|  |
| --- |
| string str = "2 7 1 3 8 6 5 8";  string[] strs = str.Split(' ');//以空字符分割  ArrayList arrOdd = new ArrayList();  ArrayList arrEven = new ArrayList();  for (int i = 0; i < strs.Length; i++)  {  int n = Convert.ToInt32(strs[i]);  if(n%2 == 0)  {  arrEven.Add(n);  }else  {  arrOdd.Add(n);  }  }  arrOdd.AddRange(arrEven);  //拼接字符串  StringBuilder sb = new StringBuilder();  for (int i = 0; i < arrOdd.Count; i++)  {  sb.Append(arrOdd[i] + " ");  }  Console.WriteLine(sb.ToString().Trim());  Console.ReadKey(); |

## Queue类

Queue类是先进先出的集合类。Queue译为队列，它的特性是一端进、一端出，先进入队列的元素会先从出口出来。

队列往往具有入队和出队的操作。下面详细介绍。

### 入队

Queue采样Enqueue方法入队。下面的代码实现了队列的定义和入队操作。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  Queue queue = new Queue(); //创建队列  print("入队前、队列的大小:" + queue.Count); //入队前、队列的大小:0  queue.Enqueue("hyfdbd"); //入队  queue.Enqueue(471288275);  queue.Enqueue("name");  print("入队后、队列的大小:" + queue.Count); //入队后、队列的大小:3  //遍历，Queue对象中元素类型为object类型  foreach(object obj in queue)  {  print(obj);  }  }  } |

### 出队

Queue采用Dequeue方法出队，即从队列的开始处移除最先入队的元素。此外，Peek方法也可以返回最先入队的元素，但是它不会移除元素。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  Queue queue = new Queue();  queue.Enqueue("hyfdbd");  queue.Enqueue(471288275);  queue.Enqueue("name");  //-----------------------------  object obj = queue.Dequeue();//移除最先入队的元素（在这个例子里是“hyfdbd”），并返回元素的值。  print(obj);//输出结果为：hyfdbd  obj = queue.Peek();//返回队列中最先入队的元素（在这里是：471288275）,但是不会移除该元素  print(obj);//输出结果为：471288275  foreach(object o in queue)  {  print(o);  }  }  } |

## Stack类

Stack类被译为堆栈类，是一个后进先出的集合类，即最后添加的元素将最先出来。堆栈主要具有进栈和出栈操作。

### 入栈

Stack提供Push方法想堆栈顶部插入一个元素。下面的代码演示了如何入栈

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  Stack stack = new Stack(); //创建stack对象  stack.Push("hyfdbd"); //入栈  stack.Push(471288275);  stack.Push("name");  print(stack.Count);  }  } |

### 出栈

Stack提供Pop方法从栈顶部移除一个元素。如果只希望获得栈顶部的元素，而不将其从stack上移除，则使用Peek方法。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  Stack stack = new Stack();  stack.Push("hyfdbd");  stack.Push(471288275);  stack.Push("name");  //-----------------------------------------  object obj = stack.Pop(); //出栈  print(obj); //name  obj = stack.Peek(); //获取栈顶元素，即最后进入的元素  print(obj); //471288275  }  } |

## HashTable类

HashTable译为哈希表，表示键（key也就是关键字）/值（value）对的集合，这些键/值对根据键的哈希代码进行组织，每一个元素都是一个存储在字典实体对象中的键/值对。HashTable像一个字典，根据键可以找到相应的值。哈希表提供了添加元素和访问元素等方法。

### 哈希表的概念及作用

一般的线性表，树中，记录在结构中的相对位置是随机的，即和记录的关键字之间不存在确定的关系，因此，在结构中查找记录时需进行一系列和关键字的比较。这一类查找方法建立在“比较“的基础上，查找的效率依赖于查找过程中所进行的比较次数。

理想的情况是能直接找到需要的记录，因此必须在记录的存储位置和它的关键字之间建立一个确定的对应关系f，使每个关键字和结构中一个唯一的存储位置相对应。

哈希表最常见的例子是以学生学号为关键字的成绩表，１号学生的记录位置在第一条，１０号学生的记录位置在第１０条...

如果我们以学生姓名为关键字，如何建立查找表，使得根据姓名可以直接找到相应记录呢？

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 刘丽 | 刘宏英 | 吴军 | 吴小艳 | 李秋梅 | 陈伟 | ... |
| 姓名中各字拼音首字母 | ll | lhy | wj | wxy | lqm | cw | ... |
| 用所有首字母编号值相加求和 | 24 | 46 | 33 | 72 | 42 | 26 | ... |
| 最小值可能为3 最大值可能为78 可放75个学生 | | | | | | | |

用上述得到的数值作为对应记录在表中的位置，得到下表：

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 成绩一 | 成绩二... |
| 3 | ... |  |  |
| ... | ... |  |  |
| 24 | 刘丽 | 82 | 95 |
| 25 | ... |  |  |
| 26 | 陈伟 |  |  |
| ... | ... |  |  |
| 33 | 吴军 |  |  |
| ... | ... |  |  |
| 42 | 李秋梅 |  |  |
| ... | ... |  |  |
| 46 | 刘宏英 |  |  |
| ... | ... |  |  |
| 72 | 吴小艳 |  |  |
| ... | ... |  |  |
| 78 | ... |  |  |

上面这张表即哈希表。

如果将来要查李秋梅的成绩，可以用上述方法求出该记录所在位置：

李秋梅:lqm 12+17+13=42 取表中第42条记录即可。

问题：如果两个同学分别叫 刘丽 刘兰 该如何处理这两条记录？

这个问题是哈希表不可避免的，即冲突现象：对不同的关键字可能得到同一哈希地址。

二、哈希表的构造方法

１、直接定址法

例如：有一个从1到100岁的人口数字统计表，其中，年龄作为关键字，哈希函数取关键字自身。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 地址 | 01 | 02 | ... | 25 | 26 | 27 | ... | 100 |
| 年龄 | 1 | 2 | ... | 25 | 26 | 27 | ... | ... |
| 人数 | 3000 | 2000 | ... | 1050 | ... | ... | ... | ... |
| ... |  |  |  |  |  |  |  |  |

２、数字分析法

有学生的生日数据如下：

年.月.日

75.10.03  
75.11.23  
76.03.02  
76.07.12  
75.04.21  
76.02.15  
...

经分析,第一位，第二位，第三位重复的可能性大，取这三位造成冲突的机会增加，所以尽量不取前三位，取后三位比较好。

３、平方取中法

取关键字平方后的中间几位为哈希地址。

４、折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为哈希地址，这方法称为折叠法。

例如：每一种西文图书都有一个国际标准图书编号，它是一个10位的十进制数字，若要以它作关键字建立一个哈希表，当馆藏书种类不到10,000时，可采用此法构造一个四位数的哈希函数。如果一本书的编号为0-442-20586-4,则：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 5864 |  | 5864 |
|  | 4220 |  | 0224 |
| +) | 04 | +) | 04 |
|  | ----------- |  | ----------- |
|  | 10088 |  | 6092 |
|  | H(key)=0088 |  | H(key)=6092 |
|  |  |  |  |
|  | (a)移位叠加 |  | (b)间界叠加 |

５、除留余数法

取关键字被某个不大于哈希表表长m的数p除后所得余数为哈希地址。

H(key)=key MOD p (p<=m)

６、随机数法

选择一个随机函数，取关键字的随机函数值为它的哈希地址，即

H(key)=random(key) ,其中random为随机函数。通常用于关键字长度不等时采用此法

### 添加和访问元素

HashTable通过add方法添加元素，它有2个参数，第一个表示键，第2个表示键所对应的值。

HashTable直接通过键访问和修改其所对应的值。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  Hashtable hash = new Hashtable();//创建哈希表  hash.Add(1, "hyfdbd"); //添加元素  hash.Add(2, "Francisco");  hash.Add("bool", true);  hash.Add("int", 1);  hash[1] = "hyfdbd is my name"; //修改元素  hash["int"] = 10;  print(hash[1]);  print(hash["int"]);  }  } |

### 删除哈希表元素

HashTable通过remove和clear两种方法删除元素。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  Hashtable hash = new Hashtable();//创建哈希表  hash.Add(1, "hyfdbd"); //添加元素  hash.Add(2, "Francisco");  hash.Add("bool", true);  hash.Add("int", 1);  hash[1] = "hyfdbd is my name"; //修改元素  hash["int"] = 10;  //--------------------------------------------  hash.Remove(1);//通过键删除元素  hash.Clear();//清空元素  }  } |

### 遍历元素

哈希表的元素是键/值对，需要通过DictionaryEntry类型来遍历。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  public class zyf : MonoBehaviour  {  void Start()  {  Hashtable hash = new Hashtable();//创建哈希表  hash.Add(1, "hyfdbd"); //添加元素  hash.Add(2, "Francisco");  hash.Add("bool", true);  hash.Add("int", 1);  hash[1] = "hyfdbd is my name"; //修改元素  hash["int"] = 10;  //--------------------------------------------  //哈希表的元素为DictionaryEntry类型  foreach(DictionaryEntry de in hash)  {  print(de.Key);//输出键  print(de.Value);//输出键对应的值  }  }  } |

## 泛型集合类

在ArrayList中，元素为object类型，对其进行操作需要强制转换类型，这会增加装箱拆箱的开销。而且强制转换可能会在运行时有错误（但是在编译时发现不了）。

在下面的程序中，定义了两个类Enemy和Hero，之后把这两个类的对象作为元素集合到ArrayList中，最后遍历元素。

|  |
| --- |
| using System;  using System.Collections;  using UnityEngine;  class Enemy  {  public int life = 100;  public int power = 5;  public void Shoot()  {  ;  }  }  class hero  {  public int life = 100;  public string weapons = "LightGun";  public void run()  {  ;  }  }  public class zyf : MonoBehaviour  {  void Start()  {  ArrayList enemys = new ArrayList();  //创建enemy对象  Enemy e1 = new Enemy();  Enemy e2 = new Enemy();  Enemy e3 = new Enemy();  //创建hero对象  hero h1 = new hero();  //将enemy和hero对象添加到ArrayList对象中。  enemys.Add(e1);  enemys.Add(e2);  enemys.Add(e3);  enemys.Add(h1);    //遍历  foreach(object obj in enemys)  {  //将object强制转换为Enemy对象。在编译时没有错误，但是运行时，因为ArrayList中有hero对象，被强制转换为enemy，会发生运行时错误  Enemy Temp = (Enemy)obj;  print(Temp);  }  }  } |

上面的代码体现了ArrayList类所存在的2个问题：

* ArrayList中的元素类型为object。在操作元素时，需要进行强制类型转换
* 原类型不一致的元素，强制转换为一个特定的类型时，在编译时不会报错，运行时才会发生错误。

而泛型集合解决了ArrayList中存在的这两个问题，性能更高。

### List<T>

List<T>的用法非常类似于ArrayList,它的语法如下：

|  |
| --- |
| List<Enemy> enemy = new List<Enemy>(); |

List<Enemy>表示泛型集合存储的元素是Enemy对象。定义泛型集合后，可以像ArrayList一样对元素添加、删除和遍历。

|  |
| --- |
| using System;  using System.Collections;  using System.Collections.Generic;//泛型需要引人的命名空间  using UnityEngine;  class Enemy  {  public int life = 100;  public int power = 5;  public void Shoot()  {  ;  }  }  class hero  {  public int life = 100;  public string weapons = "LightGun";  public void run()  {  ;  }  }  public class zyf : MonoBehaviour  {  void Start()  {  List<Enemy> enemys = new List<Enemy>();//定义泛型集合  //定义Enemy对象  Enemy e1 = new Enemy();  Enemy e2 = new Enemy();  //定义Hero对象  hero h1 = new hero();  //添加元素到泛型集合  enemys.Add(e1);  enemys.Add(e2);  //enemys.Add(new Enemy(){life = 50,Power = 5}); 也可以添加；  enemys.Add(h1);// 编译就会报错  //遍历。 对元素的操作不需要类型转换。  foreach(Enemy en in enemys)  {  en.Shoot();  }  }  } |

从上面的代码我们可以知道

* 泛型元素的操作不需要进行类型转换，减少了装箱、拆箱的开销，性能更高。
* List<Enemy>泛型中只能存储Enemy对象，hero对象添加到泛型集合中时，会报错，不用得到运行时就知道错误了。

### Dictionary<K,V>

泛型集合Dictionary<K,V>存储数据的方式和哈希表类似，通过键（key）/值(value)对保存数据。它具有返现的全部特征，编译时就检查类型约束，获取元素的时候无需类型转换。

它是语法如下：

|  |
| --- |
| using System;  using System.Collections;  using System.Collections.Generic;//泛型需要引人的命名空间  using UnityEngine;  class Enemy  {  public int identifier;  public int life = 100;  public int power = 5;  public Enemy(int identifier)  {  this.identifier = identifier;  }    public void Shoot()  {  Debug.Log(identifier + "号敌人射击");  }  }  public class zyf : MonoBehaviour  {  void Start()  {  Dictionary<int,Enemy> enemy = new Dictionary<int,Enemy>();//定义Dictionary泛型  //定义Enemy对象  Enemy e1 = new Enemy(1);  Enemy e2 = new Enemy(2);  //添加Enemy对象到泛型中  enemy.Add(e1.identifier,e1);  enemy.Add(e2.identifier,e2);  //使用值遍历  foreach(Enemy en in enemy.Values)  {  en.Shoot();  }  //使用键遍历  foreach(int i in enemy.Keys)  {  enemy[i].Shoot();  }  }  } |

### 习题

#### 分拣奇偶数

|  |
| --- |
| int[] strs = new int[] { 1, 11, 34, 12, 5, 31, 22 };  List<int> numsOdd = new List<int>();  List<int> numsEven = new List<int>();  for (int i = 0; i < strs.Length; i++)  {  if (strs[i] % 2 == 0)  {  numsEven.Add(strs[i]);  }  else  {  numsOdd.Add(strs[i]);  }  }  numsOdd.AddRange(numsEven);  foreach (int n in numsOdd)  {  Console.WriteLine(n);  }  Console.ReadKey();  } |

#### 将int数组中的奇数放到一个新的int数组中

|  |
| --- |
| int[] nums = { 1, 3, 11, 4, 44, 5, 6, 88 };  List<int> odd = new List<int>();  for (int i = 0; i < nums.Length; i++)  {  if(nums[i]%2 !=0)  {  odd.Add(nums[i]);  }  }  int[] newNums = odd.ToArray();//集合转换为数组  foreach (int item in newNums)  {  Console.WriteLine(item);  }  Console.ReadKey(); |

#### list集合中返回最大值

|  |
| --- |
| int[] strs = { 1, 44, 2, 144, 2, 66 };  List<int> nums = new List<int>();  nums.AddRange(strs);  int max = strs[0];  foreach (int item in nums)  {  strs.Max();  if(item >max)  {  max = item;  }  }  Console.WriteLine(max);  Console.ReadKey(); |

#### 把阿拉伯数字转换为中文数字

|  |
| --- |
| string str = "1壹 2贰 3叁 4肆 5伍 6陆 7柒 8捌 9玖 0零";  string[] parts = str.Split(' ');  Dictionary<char, char> dict = new Dictionary<char, char>();  for (int i = 0; i < parts.Length; i++)  {  dict.Add(parts[i][0], parts[i][1]);  }  while (true)  {  Console.WriteLine("请输入阿拉伯数字:");  string number = Console.ReadLine();  if (number.ToLower() == "quit")  {  break;  }  char[] chs = number.ToCharArray();  for (int i = 0; i < chs.Length; i++)  {  if (dict.ContainsKey(chs[i]))  {  chs[i] = dict[chs[i]];  }  }  Console.WriteLine(chs.ToArray());  Console.WriteLine("===================");  } |

#### 统计字符串中每个字母出现的次数

|  |
| --- |
| string str = "welcome to china!";  str = str.ToLower();//不区分大小写  Dictionary<char, int> dict = new Dictionary<char, int>();  for (int i = 0; i < str.Length; i++)  {  if (char.IsLetter(str[i]))//判断是否字母  {  //字母不在键中，则添加键，并赋值为1.  if (!dict.ContainsKey(str[i]))  {  dict.Add(str[i], 1);  }  else  {  //键对应的值+1；  dict[str[i]]++;  }  }  }  foreach (KeyValuePair<char, int> item in dict)  {  Console.WriteLine("{0}出现的次数为：{1}", item.Key, item.Value);  }  Console.ReadLine(); |

#### 简繁体转换

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  //4.获取用户输入的文本  string UsrInput = txtJt.Text.Trim();  //5.循环每一个字符，找到对应的繁体并拼接。  StringBuilder sbFanT = new StringBuilder();  for (int i = 0; i < UsrInput.Length; i++)  {  if(dict.ContainsKey(UsrInput[i]))  {  sbFanT.Append(dict[UsrInput[i]]);  }else  {  sbFanT.Append(UsrInput[i]);  }  }  txtFt.Text = sbFanT.ToString();  }  //创建字典集合  Dictionary<char, char> dict = new Dictionary<char, char>();  /// <summary>  /// 窗口加载时字典集合初始化。  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void Form1\_Load(object sender, EventArgs e)  { //1.设置txt的路径  DirectoryInfo dir = new DirectoryInfo(Environment.CurrentDirectory);  string p = dir.Parent.Parent.FullName + "\\"+"txt\\简体-繁体.txt";  //2. 读取文件中每一行  string[] lines = File.ReadAllLines(p,Encoding.Default);  //3.遍历每一行数据，把简体作为键，繁体作为值。  foreach (string item in lines)  {  string[] parts = item.Split('=');//以=号分割  dict.Add(parts[0][0], parts[1][0]);//读取第一个值的第一个字符为键，读取第二个值的第二个字符为值。  }  } |

#### 英汉词典

|  |
| --- |
| private void btn1\_Click(object sender, EventArgs e)  {  string userInput = txtCh.Text.Trim();  if (!dict.ContainsKey(userInput))  {  txtEN.Text = "不存在该词！";  }  else  {  txtEN.Text = dict[userInput];  }  }  //创建字典集合  Dictionary<string, string> dict = new Dictionary<string, string>();  /// <summary>  /// 窗口加载时字典集合初始化。  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void Form1\_Load(object sender, EventArgs e)  {  DirectoryInfo dir = new DirectoryInfo(Environment.CurrentDirectory);  string p = dir.Parent.Parent.FullName + "\\" + "txt\\英汉词典TXT格式.txt";  string[] lines = File.ReadAllLines(p, Encoding.Default);  foreach (string item in lines)  {  string[] parts = item.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);  if (!dict.ContainsKey(parts[0]))  {  dict.Add(parts[0], parts[1]);  }  else  {  dict[parts[0]] += Environment.NewLine + parts[1];  }  }  } |

# 委托

委托（delegate）是一种特殊的类型（class），用途是实现对方法的封装。可以把委托当作一个对象，程序员将方法的引用封装在委托对象内，当执行委托时，它会执行委托中封装的方法。

## 委托类型声明

委托是一个类型。与类一样，需要在创建变量和实例化对象之前声明。委托声明的语法如下：

|  |
| --- |
| **签名**  delegate void MyDelegate(int x);  **委托的名称**  **关键字**  **返回类型** |

委托类型的声明看上去与方法的声明很相似，有返回类型和签名。委托声明中的返回类型和签名指定了委托可以接受的方法的形式。

例如，下面的代码有一个名为MyMethod的方法，它没有返回值，有一个int类型的参数。想要把该方法传人delegate，首先要声明委托。这个委托的声明返回值和签名要与MyMethod方法的一致。因此代码中名为MyDelegate的委托返回值和签名不需与MyMethod的方法的相同。

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class GameController : MonoBehaviour {  /// <summary>  /// 创建一个方法，用来传递进委托中  /// </summary>  /// <param name="x"></param>  void MyMethod(int x)  {  print(x);  }  /// <summary>  /// 委托声明，其返回类型和签名要与传入的方法一致  /// </summary>  /// <param name="x"></param>  delegate void MyDelegate(int x);    } |

## 委托类型的对象

委托是引用类型，它由引用和对象两部分组成。因此，创建委托类型的对象也要由两部分组成：

* 委托类型变量声明，用来保存引用。
* 实例化对象，并把引用赋值给委托类型变量。

我们首先来看看，如何声明委托类型的变量。比如，我们为上面创建的委托类型MyDelegate声明一个变量，声明的方法如下：

|  |
| --- |
| MyDelegate DelegateVariable;  **变量名**  **委托的类型** |

创建委托对象的方法有两种方式：

* new运算符加上委托类型名，后面跟着一对圆括号，其中包含作为调用列表中第一个成员的方法的名字。
* 使用快捷语法，它仅有成员的方法的名字。可以使用快捷语法是因为在方法名称和其相应的委托类型之间有隐式转换

下面的代码中演示了如何把创建的对象赋值给变量DelegateVariable。

|  |
| --- |
| DelegateVariable = new MyDelegate(MyMethod);//创建委托类型对象的方法之一  DelegateVariable = MyMethod;//创建委托类型对象的快捷方法 |

需要注意的是，用赋值运算符来为委托变量赋值时，旧的委托对象会被垃圾回收器回收。委托中的方法还是只有一个，如何添加更多的方法，我们会在16.4节进行介绍。

## 委托的调用

委托的调用与方法的调用一样。调用委托时 ，委托的调用列表中的所有方法将会被调用。

下面的示例完整演示了委托从声明到调用的流程。

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class GameController : MonoBehaviour {  /// <summary>  /// 创建一个方法，用来传递进委托中  /// </summary>  /// <param name="x"></param>  void MyMethod(int x)  {  print(x);  }  /// <summary>  /// 第一步：委托声明，其返回类型和签名要与传入的方法一致  /// </summary>  /// <param name="x"></param>  delegate void MyDelegate(int x);  /// <summary>  /// 第二步：创建委托变量  /// </summary>  MyDelegate DelegateVariable;  void Start()  {  DelegateVariable = new MyDelegate(MyMethod);//第三步：创建委托对象并把引用赋值给委托变量  DelegateVariable(5);//第四步：调用委托。调用委托时，MyMethod(5)方法将被调用  }  }  /\*输出结果：  \* 5  \*/ |

## 委托中增加、删除方法

在16.2节中，我们知道用赋值运算符进行赋值时，旧的委托对象的引用会被垃圾回收器回收。委托中只能传入一个方法。在有些情况下，我们希望，把多个方法封装到一个委托中，这时就可以使用“+=”运算符可以为委托增加方法，从而实现调用一个委托来调用多个方法。

在下面的代码中，当我们使用DelegateVariable(5)调用委托时，委托调用列表中的三个MyMethod方法都会被调用，会输出三个int类型的数字5。

|  |
| --- |
| DelegateVariable = new MyDelegate(MyMethod);//第三步：创建委托对象并把引用赋值给委托变量  DelegateVariable += MyMethod;//为委托增加一个方法，这里我们增加的是相同的方法  DelegateVariable += MyMethod;//为委托再增加一个方法  DelegateVariable(5);//第四步：调用委托。调用委托时，MyMethod(5)方法将被调用 |

在使用“+=”运算符时，实际上是创建了一个新的委托，其调用列表是左边的委托和右边的方法的合并，之后这个新的委托被赋值给变量。

使用“-=”运算符可以从委托中移除方法。与为委托增加方法一样，从委托中移除方法实际上也是创建了一个新的委托。新的委托是旧委托的副本——只是没有了已经被移除的方法的引用。

下面的代码演示了如何移除委托中的方法。

|  |
| --- |
| DelegateVariable -= MyMethod; |

需要说明的是，当委托具有多个重名的方法时，使用“-=”运算符只会移除调用列表顶端的第一个方法。

# 事件

在许多程序中，当一个特定的程序事件发生时，有一步部分程序指令会接到事件发生的“通知”。这种模式叫做“发行订阅模式（publisher/subscriber ）”。

发行的类定义事件，而订阅的类通过为发行类提供方法（method）来接受指令。事件发生时，发行类触发事件，订阅方提供的方法会执行。

订阅者提供的方法叫做回调方法，因为发行方会让订阅方通过执行方法来回复事件，这个方法也叫做事件处理程序（event handler），因为它们是用来处理事件的代码。

下面是一些关于事件的重要事项：

* 发行者： 发行事件的类或结构。事件发生时，别的类会接到该“通知”。
* 订阅者： 接受事件发生通知的类或结构。
* 事件处理程序：注册到事件的方法。可以在事件所在的类或结构中，或者在不同的类或结构中。
* 触发事件： 调用或触发事件的术语。当事件被触发时，所有注册到它的方法都会以此被调用。

事件就像被简化的针对特殊用途的委托，实际上事件包含了一个私有委托。使用事件，需要包含5部分的代码：

* 委托类型声明：事件和处理程序必须有共同的签名和返回类型，它们通过委托类型声明进行描述。
* 事件处理程序声明： 在订阅者类中的方法会在事件触发时被执行。它们不需要显式（explicit）的命名，它们也可以是匿名方法或lambda表达式。
* 事件声明： 发现者类必须声明事件成员，这个成员可以被订阅者“注册”。声明一个公共事件，就是发行一个事件。
* 事件注册： 事件注册把事件和事件处理程序连接起来。
* 触发事件代码： 发行类中触发事件的代码。它调用事件导致调用事件处理程序。

## 发行者

发行者类必须提供事件和触发事件的代码。

### 委托类型声明

我们知道事件和处理程序必须有与委托类型相同的签名和返回类型。所以创建事件驱动的第一个步骤是声明委托类型。

C#提供了一个推荐遵循的标准委托类型，它是System命名空间中的EventHandler。EventHandler委托类型的声明如下：

|  |
| --- |
| using System; //EventHandler委托类型需要的命名空间  public class EventPublisher  {  public delegate void EventHandler(object sender, EventArgs e); //委托类型声明  } |

第一个参数用来保存触发事件的对象的引用。由于是object类型的，所以可以匹配任何类型的实例。第二个参数用来保存有关状态对于应用程序是否合适的状态信息。

### 声明事件

事件的声明很简单，只需要关键字event、委托类型和名字。事件声明的代码如下所示，它声明了一个叫做CountADozen的事件。

|  |
| --- |
| public class EventPublisher  {  public event EventHandler CountADozen;  }  **事件名**  **委托类型**  **关键字** |

### 触发事件

事件保存的是需要被调用的事件处理程序。如果事件没有被触发，什么都不会发生。我们需要确保在合适的时候有代码来做这件事情。

下面的代码定义了一个名为DoCount的方法。调用DoCount时，就会触发CountADozen事件。在这个例子中，调用Docount时CountADozen事件会触发八次。

|  |
| --- |
| public void DoCount()  {  for (int i = 1; i < 100; i++)  {  if (i % 12 == 0 && CountADozen != null)//当i可以被12除净，并且事件CountADozen不为空时会触发事件。  {  CountADozen(this, null);  }  }  } |

## 订阅者

### 事件处理程序

事件处理程序是订阅者中的方法，它的签名和返回类型需要与事件的委托类型相同。例如，在下面的代码中，事件处理程序IncrementCount会使字段变量DozensCount自增。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  public class EventSubscriber {  public int DozensCount = 0;  void IncrementCount(object source, EventArgs e) //事件处理程序，参数需要与委托类型相同  {  DozensCount++;  }  } |

### 事件注册

事件处理程序需要添加的事件中才能被事件调用。事件处理程序与事件相关联的过程叫做事件注册。事件注册的方法很简单，只需要使用“+=”运算符把事件处理程序赋值给事件即可。

下面的代码中，IncrementCount方法注册到了CountADozen事件中。

|  |
| --- |
| public class EventSubscriber {  public EventSubscriber(EventPublisher e)  {  e.CountADozen +=IncrementCount; //事件注册  }  } |

## 事件的完整流程

下面的代码是事件从创建到调用的完整流程。这些代码分布在3个脚本中：

|  |
| --- |
| //这是发行者中的代码，主要包含委托类型声明、事件声明以及触发事件  using UnityEngine;  using System.Collections;  using System;//EventHandler委托类型需要的命名空间  public class EventPublisher  {  public delegate void EventHandler(object sender, EventArgs e);//委托类型声明  public event EventHandler CountADozen; //事件声明  public void DoCount()//触发事件的方法  {  for (int i = 1; i < 100; i++)  {  if (i % 12 == 0 && CountADozen != null)//当i可以被12除净，并且事件CountADozen不为空时会触发事件。  {  CountADozen(this, null);//触发事件  }  }  }  } |
| //这是事件订阅者，主要包括事件处理程序以及把事件处理程序注册到事件中。  using UnityEngine;  using System.Collections;  using System;  public class EventSubscriber {  public int DozensCount = 0;  public EventSubscriber(EventPublisher e)  {  e.CountADozen +=IncrementCount;//事件注册  }  void IncrementCount(object source, EventArgs e)//事件处理程序  {  DozensCount++;  }  } |
| //这是使用事件的代码。程序执行时，通过Start方法自动创建实例并通过调用方法触发事件。  using UnityEngine;  using System.Collections;  public class EventManager : MonoBehaviour {  void Start () {  EventPublisher EP = new EventPublisher();  EventSubscriber ES = new EventSubscriber(EP);  EP.DoCount();//调用方法实现触发事件  print(ES.DozensCount);  }  } |

# 预处理

预处理又称为预编译，是整个编译过程的最先做的工作，预编译指令指示了在程序正式编译前就由编译器进行的操作。

## 全局预处理

在Assets目录下面添加.rsp文件，rsp文件的名词与你使用的语言相关：

|  |  |
| --- | --- |
| C# | <Project Path>/Assets/smcs.rsp |
| C# - Editor Scripts | <Project Path>/Assets/gmcs.rsp |
| UnityScript | <Project Path>/Assets/us.rsp |
| Boo | <Project Path>/Assets/boo.rsp |

用C#语言，就需要在Assets文件夹下添加smcs.rsp文件。

比如，我们在smcs.rsp文件中添加一行字“-unsafe”（不带空格和双引号）。之后重启Unity，因为这个预编译是在启动U3D时候运行的。这样做的目的是为编辑器中的smcs.exe 添加编译命令。

测试代码如下：

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class Pre : MonoBehaviour  {  //unsafe 后可以使用C#中的指针。  unsafe void test()  {  int i = 10;  int k;  int\* j = &i;  \*j = 20;  k = i + 1;    print("unsafe test " + k.ToString());  }  void Start()  {  test();  }  } |

Smcs预编译位于安装目录的“…\Editor\Data\Mono\lib\mono\unity ”下，关于smcs的更多信息可以在cmd下输入”smcs –help”

# using的用法

C#中using的用法，大致可以归为三种：

* using指令。using + 命名空间名字
* using别名。using + 别名 = 包括详细命名空间信息的具体的类型。
* using语句，定义一个范围，在范围结束时处理对象。

## using指令

可以在程序中直接用命令空间中的类型，而不必指定类型的详细命名空间，类似于Java的import，这个功能也是最常用的，几乎每个cs的程序都会用到。

|  |
| --- |
| using System.IO; |

写入上面的代码后，就可以直接引用System.IO命名空间下的各种类了。

## using别名

这种做法有个好处就是当同一个cs引用了两个不同的命名空间，但两个命名空间都包括了一个相同名字的类型的时候。当需要用到这个类型的时候，就每个地方都要用详细命名空间的办法来区分这些相同名字的类型。而用别名的方法会更简洁，用到哪个类就给哪个类做别名声明就可以了。注意：并不是说两个名字重复，给其中一个用了别名，另外一个就不需要用别名了，如果两个都要使用，则两个都需要用using来定义别名的。

|  |
| --- |
| using aClass = NameSpace1.MyClass;  using bClass = NameSpace2.MyClass;  aClass my1 = new aClass();  Console.WriteLine(my1);  bClass my2 = new bClass();  Console.WriteLine(my2); |

## using语句

当在using的括号中创建了类的实例，到达using语句末尾或者中途引发了异常并且控制离开了语句块时，就会自动调用Dispose方法。

using语句中创建的实例必须是实现了IDisposable接口的类。

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  //括号中的类必须实现IDisposable接口  using(Person p = new Person())  {  p.Name = "hao";  Console.WriteLine(p.Name);  }  Console.ReadLine();  }  }  class Person:IDisposable  {  string name;  public string Name  {  get { return name; }  set { name = value; }  }  public void Dispose()  {  Console.WriteLine("Dispose执行了"); ;  }  } |

using语句实际上被编译为try…finally语句，只是用法更简洁。把上面的代码用try…finally的形式重写，代码如下：

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Person p = new Person();  try  {  p.Name = "hao";  Console.WriteLine(p.Name);  }  finally  {  p.Dispose();  }  Console.ReadLine();  }  }  class Person : IDisposable  {  string name;  public string Name  {  get { return name; }  set { name = value; }  }  public void Dispose()  {  Console.WriteLine("Dispose执行了"); ;  }  } |

# 文件操作

对文件操作的相关类都在System.IO命名空间下，要对文件进行操作首先就要引入该命名空间：

|  |
| --- |
| using System.IO; |

文件操作的常用类有以下几种：

* Path类：操作文件的路径。
* Directory类：操作目录。
* File类：操作文件

## Path类

Path类用来操作文件的路径。它操作的是字符串，与实际的文件文件无关。

下面为常用的Path类的方法

|  |
| --- |
| string path = @"D:\Untitled-1.jpg";  //获取文件名，带后缀  string name = System.IO.Path.GetFileName(path);  Console.WriteLine(name); //输出为:Untitled-1.jpg  //获取文件名，不带后缀  string nameNE = Path.GetFileNameWithoutExtension(path); //无后缀  Console.WriteLine(nameNE); //输出为:Untitled-1  //获取文件后缀  string ext = Path.GetExtension(path);  Console.WriteLine(ext);//输出为： .jpg  //获取文件的路径部分，不带文件名  string filepath = Path.GetDirectoryName(path);  Console.WriteLine(filepath);//输出为：D:\  //修改文件后缀。只是修改path中的字符串，与磁盘上的文件无关。  string newPath = Path.ChangeExtension(path, ".txt");  Console.WriteLine(newPath);//输出为:D:\Untitled-1.txt  //合并路径，将2个路径合并成一个，如果2个路径之间没有斜线，会自动加上  string p = @"D:\abc";  string f = "file.txt";  string CombinePath = Path.Combine(p, f);  Console.WriteLine(CombinePath);//输出为：D:\abc\file.txt  //获取当前机器的临时目录的路径  string tmpPath = Path.GetTempPath();  Console.WriteLine(tmpPath);//C:\Users\Francisco\AppData\Local\Temp\  //创建临时文件，并获得其路径  string tmpFile = Path.GetTempFileName();  Console.WriteLine(tmpFile);//C:\Users\Francisco\AppData\Local\Temp\tmp22B6.tmp |

### 案例：获取当前执行程序所在的路径

|  |
| --- |
| //获取当前执行的程序集（exe）所在的路径  string exePath = System.Reflection.Assembly.GetExecutingAssembly().Location;  Console.WriteLine(exePath);  //当前执行的程序集所在的目录  string exeDirctory = Path.GetDirectoryName(exePath);  //定位path到当前程序集所在的目录的demo目录下  string path = Path.Combine(exeDirctory, "demo");  Console.WriteLine(path);  Console.ReadLine(); |

## Directory类

### 删除目录、判断目录是否存在

|  |
| --- |
| //判断目录是否存在  bool exist = Directory.Exists(@"c:\abc");  Console.WriteLine(exist);  //删除目录，如果目录不存在，会抛出异常System.IO.DirectoryNotFoundException,所以最好先判断目录是否存在，再删除  Directory.Delete(@"c:\abc");  if (Directory.Exists(@"c:\abc"))  {  //只能删除空目录  Directory.Delete(@"c:\abc");  //递归删除，删除目录以及目录中所有的文件和子目录  Directory.Delete(@"c:\abc", true);  }  Console.ReadLine(); |

### 获取子目录、子文件

|  |
| --- |
| string dirc = @"C:\Users";  //获取目录中的子目录  string[] FullDireretorys = Directory.GetDirectories(dirc);  foreach (string item in FullDireretorys)  {  Console.WriteLine(item);  }  Console.WriteLine("======================================");  //获取目录中的文件  string[] fileDirectorys = Directory.GetFiles(dirc);  foreach (string item in fileDirectorys)  {  Console.WriteLine(item);  }  Console.WriteLine("======================================");  //方法重载，获取以r结尾的子目录，  string[] allDirectorysbySearchPattern = Directory.GetDirectories(dirc, "\*r");//重载的第二个参数为通配符。  foreach (string item in allDirectorysbySearchPattern)  {    Console.WriteLine(item);  }  Console.WriteLine("======================================");  //方法重载，获取子目录及其子目录（所有子目录，不论第几代）  string[] Alldirecotrys = Directory.GetDirectories(dirc, "\*", SearchOption.AllDirectories);  foreach (string item in Alldirecotrys)  {  Console.WriteLine(item);  }  Console.ReadLine(); |

### 创建、移动文件夹

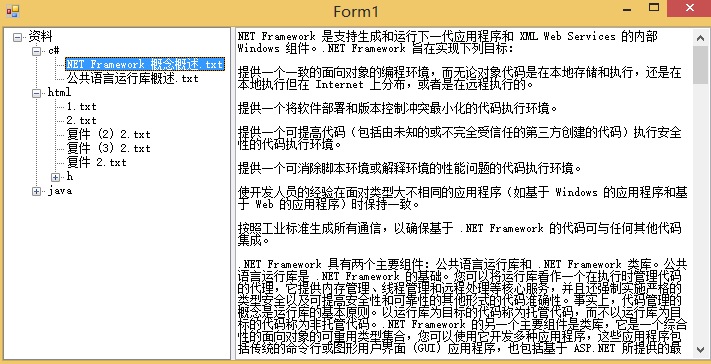
|  |
| --- |
| Directory.CreateDirectory(@"C:\aaa"); //创建文件夹  File.Create(@"c:\aaa\a.txt");//创建文件  Directory.CreateDirectory(@"D:\bbb");  Directory.Move(@"D:\aaa", @"D:\bbb\abc");//移动aaa文件夹到bbb文件夹下，aaa文件夹重命名为abc  Directory.Move(@"D:\aaa", @"D:\bbb");//重命名aaa文件夹为bbb |

### 案例：递归实现目录树

|  |
| --- |
| private void btn\_load\_Click(object sender, EventArgs e)  {  string path = txt\_input.Text.Trim();  //递归所有的目录，按照层次结构加载到TreeView上。  LoadAll(path, TreeView\_loaded.Nodes);  }  private void LoadAll(string path, TreeNodeCollection treeNodes)  {  //1.根据文件夹路径获取所有的子文件夹和子文件  string[] files = Directory.GetFiles(path);  string[] directories = Directory.GetDirectories(path);  //2.遍历，把所有子文件夹加载到TreeView上。  foreach (string item in directories)  {  TreeNode node = treeNodes.Add(Path.GetFileName(item));//只添加文件夹名到TreeView上  //递归。实现目录树。  LoadAll(item, node.Nodes);  }  //2.1遍历，把所有文件加载到TreeView上。  foreach (string item in files)  {  treeNodes.Add(Path.GetFileName(item));//只添加文件名到TreeView上  }  } |

### 案例：创建目录树格式的txt浏览程序

下面创建如下格式的程序：



|  |
| --- |
| //窗口加载时初始化  private void Form1\_Load(object sender, EventArgs e)  {  //获取当前运行程序的路径  string exePath = System.Reflection.Assembly.GetExecutingAssembly().Location;  //设置路径为当前运行程序所在目录的demo目录  string path = Path.Combine(Path.GetDirectoryName(exePath), "demo");  //LoadTextFile加载目录树到TreeView上  LoadTextFile(path, treeView1.Nodes);  }  /// <summary>  /// 加载目录树到TreeView上  /// </summary>  /// <param name="path">要加载的目录的地址</param>  /// <param name="treeNodeCollection">TreeView的Nodes集合</param>  private void LoadTextFile(string path, TreeNodeCollection treeNodeCollection)  {  //加载子目录与txt文件  string[] files = Directory.GetFiles(path,"\*txt");  string[] directories = Directory.GetDirectories(path);  //遍历文件加载到TreeView中  foreach (var item in files)  {  TreeNode node = treeNodeCollection.Add(Path.GetFileName(item));  node.Tag = item;//Tag属性用来保存文件地址。  }  //遍历目录加载到TreeView中  foreach (var item in directories)  {  TreeNode node = treeNodeCollection.Add(Path.GetFileName(item));  LoadTextFile(item, node.Nodes);  }  }  //双击TreeView中节点加载  private void treeView1\_NodeMouseDoubleClick(object sender, TreeNodeMouseClickEventArgs e)  {    if (e.Node.Tag != null)//双击的节点不是文件夹  {  //Tag为object类型，需要转换为字符串  string currentPath = e.Node.Tag.ToString();  //读取文件  string txt = File.ReadAllText(currentPath,Encoding.Default);  textBox1.Text = txt;  }  } |

### 案例：web程序创建目录

|  |
| --- |
| string dir = HttpContext.Current.Server.MapPath("/StaticPage/" + item.PublishDate.Year + "/" + item.PublishDate.Month + "/" + item.PublishDate.Day + "/");  Directory.CreateDirectory(Path.GetDirectoryName(dir)); |

## File类

File类用来操作文件，如判断文件是否存在，拷贝，创建，删除、读取文件等。

|  |
| --- |
| //1.判断文件是否存在  bool b = File.Exists(@"C:\a.txt");  Console.WriteLine(b);  //2.拷贝一个文件  File.Copy(@"C:\a.txt", @"D:\a.txt");  //3. 创建一个文件  //File.Create(@"C:\b.txt");  //4. 删除一个文件  File.Delete(@"D:\a.txt");  //5.写入-----------------------------------  //5.1 写入字符串  File.WriteAllText(@"C:\b.txt", "你好世界！！");  //5.2 写入字符串数组，每一个元素为一行。  string[] lines = new string[] { "aaa", "bbb", "ccc" };  File.WriteAllLines(@"C:\b.txt", lines);  //5.2 写入字节  string msg = "Hello! World";  byte[] bytes = System.Text.Encoding.UTF8.GetBytes(msg);  File.WriteAllBytes(@"C:\b.txt", bytes);  //-----------------------------------------------  //6. 读取文件  //6.1 读取整个文件  File.ReadAllText(@"C:\b.txt");  //6.2 读取为字符串数组，每一行为数组的一个元素  String[] ReadLines = File.ReadAllLines(@"C:\b.txt", Encoding.Default);  foreach (var item in ReadLines)  {  Console.WriteLine(item);  }  //6.3 读取为字节  byte[] readBytes = File.ReadAllBytes(@"C:\b.txt");  foreach (var item in readBytes)  {  Console.WriteLine(item);  }  Console.ReadLine(); |

### 快速创建文件流

File类提供了以下几个方法可以快速创建文件流（什么是文件流，请参考下节FileStream(文件流)）。

* Open()；返回FileStream
* OpenRead()；返回只读的FileSteam
* OpenWrite()；返回只写的FileStream

|  |
| --- |
| FileStream fs = File.Open("1.txt",FileMode.Open,FileAccess.Read);  FileStream fs1 = File.OpenRead("1.txt");  FileStream fs2 = File.OpenWrite("2.txt"); |

## FileStream(文件流)

用File.ReadAllText、File.WriteAllText进行文件读写是一次性读、写。如果文件非常大会占内存、慢。而Stream不会将所有内容一次性读取到内存中，它有一个指针，指针指到那里才能读、写到那里。

Stream有很多类，FileStream是其中一种。

### 用流写入数据

|  |
| --- |
| string txt = "中国，四大文明古国之一，是一个以华夏文明为主体、以汉族为主体民族的统一多民族国家。中国境内56个民族统称为中华民族，龙是中华民族的象征。";  //1.创建文件流，第一个参数为要操作的文件，第二个参数如何对文件进行操作，Create为创建文件，第三个参数定义操作的方法：读或写等  FileStream fs = new FileStream(@"D:\中国.txt", FileMode.Create, FileAccess.Write);  // 流的Read以字节方式读取，需要转换字符串为字节数组。  byte[] buffer = System.Text.Encoding.UTF8.GetBytes(txt);  //2. 读或写文件  //参数1：表示把那个字节数组写入到文件中。  //参数2：表示从数组的第几个下标开始写入，一般为0  //参数3：要写入的字节个数  fs.Write(buffer, 0, buffer.Length);  //3. 关闭文件流，也可以调用 fs.Dispose方法，Dispose方法内部调用了Flush和close方法。还能使用using语句（详见下节的例子）  fs.Flush();//清空缓存区，把数据写入文件，如果不调用，可能出现读取不完整。  fs.Close();//关闭文件流 |

### 通过文件流进行拷贝

|  |
| --- |
| string source = @"D:\百度云同步盘\Windows程序设计.pdf";  string destiny = @"E:\Windows程序设计.pdf";  #region 数据拷贝  //创建2个流，一个读、一个写  //创建读取流  using (FileStream fsRead = new FileStream(source, FileMode.Open, FileAccess.Read))  {  //创建写入流  using (FileStream fsWrite = new FileStream(destiny, FileMode.Create, FileAccess.Write))  {  //读取流的文件的大小。  long len = fsRead.Length;  //创建缓冲区。  byte[] buffer = new byte[1024 \* 1024 \* 5];//5MB的缓冲区  //通过文件流读取。  //参数1：表示读取的数据存到哪个数组中  //参数2：表示从第几个索引开始读数据  //参数3：表示本次最多读多少字节  //返回值：表示本次实际读取了多少字节。  int readCount = fsRead.Read(buffer, 0, buffer.Length);  //当读取字节大于0表示，读取到了数据。  while (readCount > 0)  {  //把读到缓冲区的数据写入到新文件流中。  fsWrite.Write(buffer, 0, readCount);  //当前流读取后的位置除总长度等于完成的百分百。  double rate = fsWrite.Position \* 1.0 / len \* 100;  //上次读取的内容写入完毕后，继续再读取。因为流都有position属性，所以这里0不是数据起始，而是position的起始。fsRead.Position  readCount = fsRead.Read(buffer, 0, buffer.Length);  //ToString("f2")保留小数点后2位，也可以用ToString("0.00")  Console.WriteLine("完成了{0}%", rate.ToString("f2"));//  }  }  }  #endregion  Console.WriteLine("ok");  Console.ReadLine(); |

### 简单的加密

下面的代码将读取的每个字节进行(255-字节中数据)的操作，进行加密。这个算法只需要对加密的文件再次加密则可以恢复成原文件。

|  |
| --- |
| string source = @"D:\中国1.txt";  string destiny = @"D:\中国2.txt";  using (FileStream fsRead = new FileStream(source, FileMode.Open, FileAccess.Read))  {  using (FileStream fsWrite = new FileStream(destiny, FileMode.Create, FileAccess.Write))  {  byte[] buffer = new byte[1024 \* 1024];  int readCount = fsRead.Read(buffer, 0, buffer.Length);  //---------------------加密------再次运行会解密---  for (int i = 0; i < readCount; i++)  {  buffer[i] =(byte)(byte.MaxValue -buffer[i]);  }  //---------------------加密----------------------  while (readCount > 0)  {  fsWrite.Write(buffer, 0, readCount);  readCount = fsRead.Read(buffer, 0, buffer.Length);  }  }  } |

## StreamReader和StreamWriter

如果对文本文件需要读取一部分显示则使用FileStream会有问题，因为可能FileStream会在读取的时候把一个汉字的字节数给分开，所以造成显示的时候无法正确显示字符串。

对于大文本文件的读取一般使用StreamReader类，而大文本文件的写入一般使用StreamWriter。这两个类是对FileStream的封装，通过判断换行进行读写。

### StreamWriter

|  |
| --- |
| //1.创建一个StreamWriter。  using (StreamWriter sw = new StreamWriter("1.txt"))  {  for (int i = 0; i < 1000; i++)  {  //2.执行读写。File.WriteAllLine内部也是创建StreamWriter，执行WriteLine。但是每次用WriteAllLine都要创建StreamWriter  sw.WriteLine(i + "\t" + System.DateTime.Now);  }  }  Console.WriteLine("ok"); |

### StreamReader

|  |
| --- |
| int count = 0;  //1. 创建StreamReader  using (StreamReader sr = new StreamReader("英汉词典TXT格式.txt", Encoding.Default))  {  //2.判断position是否在流尾，在流尾则表示文件读取完毕，结束循环。  while (!sr.EndOfStream)  {  count++;  //3. 读取每一行。  string line = sr.ReadLine();  Console.WriteLine(line);  }  }  Console.WriteLine("总行数为{0}", count);  Console.ReadLine(); |

判断是否读到文件末尾还有另外一种写法：

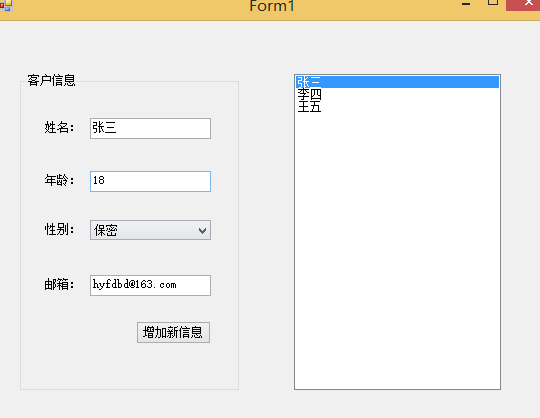
|  |
| --- |
| int count = 0;  using (StreamReader sr = new StreamReader("英汉词典TXT格式.txt", Encoding.Default))  {  string line = null;  while ((line = sr.ReadLine()) != null)  {  count++;  Console.WriteLine(line);  }  }  Console.WriteLine("总行数为：{0}", count);  Console.ReadLine(); |

### 案例：对职工工资文件进行处理，所有人的工资加倍然后输出到新的文件。

|  |
| --- |
| string source = "工资文件.txt";  string destiny = "工资文件\_新.txt";  //创建读取文件的StreamReader  using (StreamReader sr = new StreamReader(source, Encoding.Default))  {  //创建写入文件的StreamWriter  using (StreamWriter sw = new StreamWriter(destiny))  {  //判断是否读取到文件末尾  while (!sr.EndOfStream)  { //按行读取  string line = sr.ReadLine();  //以“|”进行分割  string[] split = line.Split('|');  //工资成2  int salary = int.Parse(split[1]) \* 2;  //字符串拼接  string newline = split[0] + "|" + Convert.ToString(salary);  sw.WriteLine(newline);  }  } |

### 案例：从文本文件读写客户资料

效果:



代码如下:

|  |
| --- |
| public partial class Form1 : Form  {  public Form1()  {  InitializeComponent();  }  private void Form1\_Load(object sender, EventArgs e)  { //姓名默认选项为保密。  cmb\_sex.SelectedIndex = 2;  //读取客户姓名到listbox  using (StreamReader sr = new StreamReader("客户信息.txt", Encoding.Default))  {  while (!sr.EndOfStream)  {  string info = sr.ReadLine();  string[] infoSplit = info.Split(',');  lb\_userInfo.Items.Add(infoSplit[0]);  }  }  }  private void button1\_Click(object sender, EventArgs e)  { //点击按钮添加性别、性别、年龄、邮箱等信息到客户信息.txt文件中  string userName = txt\_name.Text.Trim();  string userAge = txt\_age.Text.Trim();  string userSex = cmb\_sex.Text.Trim();  string userMail = txt\_mail.Text.Trim();  string userInfo = userName + "," + userAge + "," + userSex + "," + userMail;  using (StreamWriter sw = new StreamWriter("客户信息.txt", true, Encoding.Default))  {  sw.WriteLine(userInfo);  }  //添加客户姓名到listbox中。  lb\_userInfo.Items.Add(userName);  }  private void lb\_userInfo\_SelectedIndexChanged(object sender, EventArgs e)  {  //listbox改变选项，显示客户的性别、年龄、邮箱等信息到客户信息区。  using (StreamReader sr = new StreamReader("客户信息.txt", Encoding.Default))  {  while (!sr.EndOfStream)  {  string info = sr.ReadLine();  string[] infoSplit = info.Split(',');  if (infoSplit[0] == lb\_userInfo.Text)  {  txt\_name.Text = infoSplit[0];  txt\_age.Text = infoSplit[1];  if (infoSplit[2] == "男")  {  cmb\_sex.SelectedIndex = 0;  }  else if (infoSplit[2] == "女")  {  cmb\_sex.SelectedIndex = 1;  }  else if (infoSplit[2] == "保密")  {  cmb\_sex.SelectedIndex = 2;  }  txt\_mail.Text = infoSplit[3];  break;  }  }  }  }  } |

在上面的代码中，读取客户信息到listbox以及在listbox中点击姓名显示信息到客户信息区时，需要读取2次文本文件。这里可以在读取时格式化为类，从而只需要读取一次即可。

首先，新建一个Person.cs文件：

|  |
| --- |
| using System.Text;  using System.Threading.Tasks;  namespace \_16\_从文本文件读取或增加客户信息  {  class Person  {  string Name { get; set; }  int Age { get; set; }  string Sex { get; set; }  string Mail { get; set; }  }  } |

之后Form1.cs中代码如下：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.Data;  using System.Drawing;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Windows.Forms;  using System.IO;  namespace \_16\_从文本文件读取或增加客户信息  {  public partial class Form1 : Form  {  public Form1()  {  InitializeComponent();  }  private void Form1\_Load(object sender, EventArgs e)  { //姓名默认选项为保密。  cmb\_sex.SelectedIndex = 2;  //读取客户姓名到listbox  using (StreamReader sr = new StreamReader("客户信息.txt", Encoding.Default))  {  while (!sr.EndOfStream)  {  string info = sr.ReadLine();  string[] infoSplit = info.Split(',');  Person p = new Person();  p.Name = infoSplit[0];  p.Age = infoSplit[1];  p.Sex = infoSplit[2];  p.Mail = infoSplit[3];  p.ToString();  //listbox中添加persong类型的对象。  lb\_userInfo.Items.Add(p);  }  }  }  private void button1\_Click(object sender, EventArgs e)  { //点击按钮添加性别、性别、年龄、邮箱等信息到客户信息.txt文件中  string userName = txt\_name.Text.Trim();  string userAge = txt\_age.Text.Trim();  string userSex = cmb\_sex.Text.Trim();  string userMail = txt\_mail.Text.Trim();  string userInfo = userName + "," + userAge + "," + userSex + "," + userMail;  using (StreamWriter sw = new StreamWriter("客户信息.txt", true, Encoding.Default))  {  sw.WriteLine(userInfo);  }  //添加客户姓名到listbox中。  Person p = new Person();  p.Name = userName;  p.Age = userAge;  p.Sex = userSex;  p.Mail = userMail;  lb\_userInfo.Items.Add(p);  }  private void lb\_userInfo\_SelectedIndexChanged(object sender, EventArgs e)  {  //listbox改变选项，显示客户的性别、年龄、邮箱等信息到客户信息区。  //SelectedItem为object类型，需要转换为对应的person类型。  Person p = lb\_userInfo.SelectedItem as Person;  if(p != null)  {  txt\_name.Text = p.Name;  txt\_age.Text = p.Age;  cmb\_sex.Text = p.Sex;  txt\_mail.Text = p.Mail;  }  }  }  } |

## 压缩和解压缩

net4.5提供了ZipFile类来对目录进行压缩和解压缩。需要引入System.IO.Compression.

FileSystem 程序集。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.IO;  using System.IO.Compression;/  namespace \_17\_01\_压缩文件  {  class Program  {  static void Main(string[] args)  {  string dire = Directory.GetCurrentDirectory();  string sourcePath = Path.Combine(dire, "source");  string targetPath = Path.Combine(dire, "result.zip");  string decompressPath = Path.Combine(dire, "decompress");  //压缩  compress(sourcePath, targetPath);  //解压缩  decompress(targetPath, decompressPath);  Console.WriteLine("ok");  Console.ReadLine();  }  private static void decompress(string targetPath, string decompressPath)  {  if (File.Exists(targetPath))  { //解压缩zip文件为目录  ZipFile.ExtractToDirectory(targetPath, decompressPath);  }  }  private static void compress(string sourcePath, string targetPath)  {  if (!File.Exists(targetPath))  {  //把目录压缩为zip格式。  ZipFile.CreateFromDirectory(sourcePath, targetPath);  }  else  {  File.Delete(targetPath);  }  }  }  } |

# 对象序列化

对象序列化，也就是对象格式化，就是把一个对象转换成另一种格式的过渡。序列化的目的是为了把数据以另一种格式来表示，方便存储与交换数据，对象系列化可以把内存中对象的状态持久化。

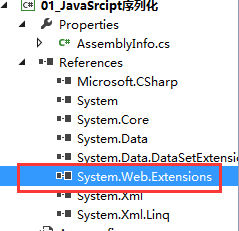
序列化只序列化那些存储数据的成员（字段及其属性），而不会序列化方法。

常用序列化方式有：

* 二进制序列化: 类中的所有成员变量（甚至标记为 private 的变量）都将被序列化
* xml序列化：只序列化公共字段
* javascript序列化

## javascript序列化

1. 需要引入System.Web.Extensions程序集。



1. 引入命名空间

|  |
| --- |
| using System.Web.Script.Serialization; |

1. 对对象集合进行序列化

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  List<Person> list = new List<Person>();  //用对象构造器添加对象到list中。  list.Add(new Person() { Name = "张三", Age = 18, Email = "zs@163.com" });  list.Add(new Person() { Name = "李四", Age = 28, Email = "ls@qq.com" });  list.Add(new Person() { Name = "王五", Email = "ww@yahoo.com", Age = 20 });  //1.创建javascript序列化器  JavaScriptSerializer jss = new JavaScriptSerializer();  //2.对list进行序列化，返回值为string类型  string str = jss.Serialize(list);  //3.把序列化的字符串写到txt文档中  File.WriteAllText("list.txt", str);  Console.WriteLine("ok");  Console.ReadKey();  }  }  class Person  {  public string Name { get; set; }  public int Age { get; set; }  public string Email { get; set; }  public void sayHello()  {  Console.WriteLine("Hello");  }  } |

1. 对象序列化后的格式如下：



序列化的对象包含在一对“[]”中，每个对象被包在“{}”中。属性与值被序列化为键值对的形式。

## XML序列化

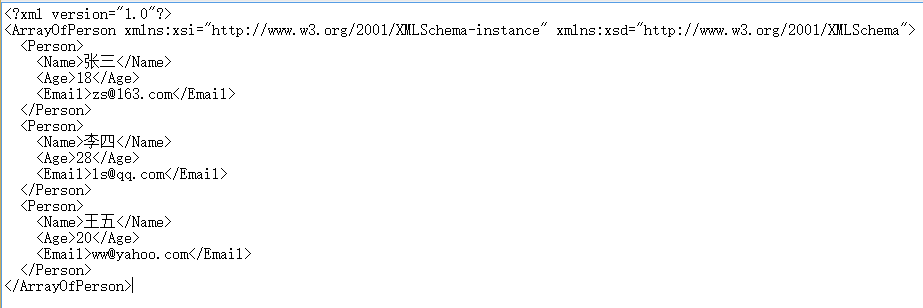
1. 引入命名空间

|  |
| --- |
| using System.Xml.Serialization; |

1. 序列化并写到xml文件中

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  List<Person> list = new List<Person>();  list.Add(new Person() { Name = "张三", Age = 18, Email = "zs@163.com" });  list.Add(new Person() { Name = "李四", Age = 28, Email = "ls@qq.com" });  list.Add(new Person() { Name = "王五", Email = "ww@yahoo.com", Age = 20 });  //1.创建xml序列化器。传入值为对象的类型  // XmlSerializer xmls1 = new XmlSerializer(typeof(List<Person>));  XmlSerializer xmls = new XmlSerializer(list.GetType());  //2. 创建FileStream写入流  using (FileStream fs = File.OpenWrite("list.xml"))  {  //3.把list序列化被写入到fs流所写入的文件中。  xmls.Serialize(fs, list);  }  Console.WriteLine("ok");  Console.ReadKey();  }  }  //xml序列化的类需要是公共类  public class Person  {  public string Name { get; set; }  public int Age { get; set; }  public string Email { get; set; }  public void sayHello()  {  Console.WriteLine("Hello");  }  } |

1. 对象序列化后的效果：



## 二进制序列化

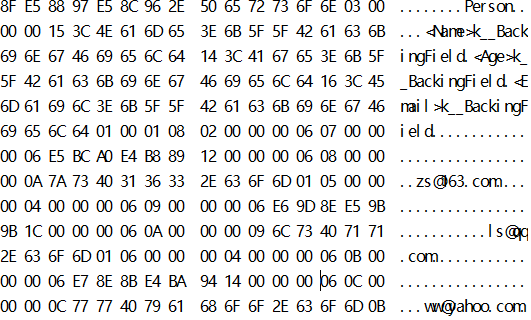
1. 引入命名空间

|  |
| --- |
| using System.Runtime.Serialization.Formatters.Binary; |

1. 进行二进制序列化

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  List<Person> list = new List<Person>();  list.Add(new Person() { Name = "张三", Age = 18, Email = "zs@163.com" });  list.Add(new Person() { Name = "李四", Age = 28, Email = "ls@qq.com" });  list.Add(new Person() { Name = "王五", Email = "ww@yahoo.com", Age = 20 });  //1. 创建二进制够构造器  BinaryFormatter binaryFormatter = new BinaryFormatter();  //2. 创建写入流  using (FileStream fsWrite = File.OpenWrite("list.bin"))  {  //3.序列化并写入到文件中  binaryFormatter.Serialize(fsWrite, list);  }  Console.WriteLine("ok");  Console.ReadKey();  }  }  //4. 类需要标记为可序列化，才能进行二进制序列化。  [Serializable]  class Person  {  public string Name { get; set; }  public int Age { get; set; }  public string Email { get; set; }  public void sayHello()  {  Console.WriteLine("Hello");  }  } |

1. 结果格式化为二进制数据



### 二进制序列化的要求

1. 被序列化的对象类型必须用特性[Serializable]标记为可序列化的.
2. 二进制序列化会把属性对应的字段序列化到文件中，所以最好类型中不要使用自动属性（编译器自动生成字段，每次生成可能会不一样），而是要自己写字段和属性。
3. 当序列化一个对象时，这个对象本身及其所有父类都必须标记为可序列化的。
4. 类型中所有属性与字段的类型也标记为可序列化的。
5. 通过[Non Serializable]可以把某个字段标记为不可序列化的。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.IO;  using System.Runtime.Serialization.Formatters.Binary;  namespace \_03\_二进制序列化  {  class Program  {  static void Main(string[] args)  {  List<Person> list = new List<Person>();  list.Add(new Person() { Name = "张三", Age = 18, Email = "zs@163.com" });  list.Add(new Person() { Name = "李四", Age = 28, Email = "ls@qq.com" });  list.Add(new Person() { Name = "王五", Email = "ww@yahoo.com", Age = 20 });  //1. 创建二进制够构造器  BinaryFormatter binaryFormatter = new BinaryFormatter();  //2. 创建写入流  using (FileStream fsWrite = File.OpenWrite("list.bin"))  {  //3.序列化并写入到文件中  binaryFormatter.Serialize(fsWrite, list);  }  Console.WriteLine("ok");  Console.ReadKey();  }  }  // 1.被序列化的对象类型必须用特性[Serializable]标记为可序列化的.  [Serializable]  class Person  {  //2. 要自己自己写字段和属性。避免编译自动生成字段名不一致。  string \_name;  //5. 添加[NonSerialized]特性的\_age在序列化时不会被序列化  [NonSerialized]  int \_age;  //4.int String 转到定义（F12）查看会发现，它们都有 [Serializable]特性。  string \_Email;  public string Email  {  get { return \_Email; }  set { \_Email = value; }  }  public int Age  {  get { return \_age; }  set { \_age = value; }  }  public string Name  {  get { return \_name; }  set { \_name = value; }  }  public void sayHello()  {  Console.WriteLine("Hello");  }  }  //3.对象的所有父类都必须标记为可序列化的  [Serializable]  class Animal  {  string \_gender;  public string Gender  {  get { return \_gender; }  set { \_gender = value; }  }  }  } |

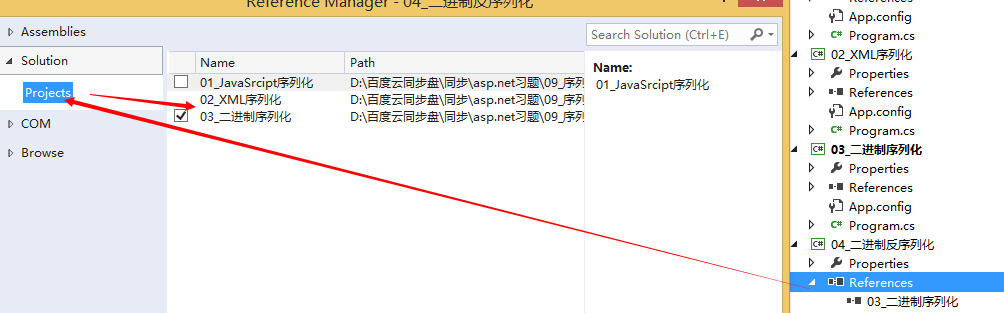
### 反序列化

代码如下：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.IO;  using System.Runtime.Serialization.Formatters.Binary;  namespace \_03\_二进制序列化//需要二进制序列化这个姓名，因为Person类在这个项目中  {  class Program  {  static void Main(string[] args)  {  List<Person> list = new List<Person>();  list.Add(new Person() { Name = "张三", Age = 18, Email = "zs@163.com" });  list.Add(new Person() { Name = "李四", Age = 28, Email = "ls@qq.com" });  list.Add(new Person() { Name = "王五", Email = "ww@yahoo.com", Age = 20 });  //1. 创建二进制够构造器  BinaryFormatter binaryFormatter = new BinaryFormatter();  //2. 创建写入流  using (FileStream fsWrite = File.OpenWrite("data"))  {  //3.序列化并写入到文件中  binaryFormatter.Serialize(fsWrite, list);  }  Console.WriteLine("ok");  Console.ReadKey();  }  }  // 1.被序列化的对象类型必须用特性[Serializable]标记为可序列化的.  [Serializable]  public class Person  {  //2. 要自己自己写字段和属性。避免编译自动生成字段名不一致。  string \_name;  //5. 添加[NonSerialized]特性的\_age在序列化时不会被序列化  [NonSerialized]  int \_age;  //4.int String 转到定义（F12）查看会发现，它们都有 [Serializable]特性。  string \_Email;  public string Email  {  get { return \_Email; }  set { \_Email = value; }  }  public int Age  {  get { return \_age; }  set { \_age = value; }  }  public string Name  {  get { return \_name; }  set { \_name = value; }  }  public void sayHello()  {  Console.WriteLine("Hello");  }  }  //3.对象的所有父类都必须标记为可序列化的  [Serializable]  public class Animal  {  string \_gender;  public string Gender  {  get { return \_gender; }  set { \_gender = value; }  }  }  } |

注意事项：

* 1. 需要引入“二进制序列化这个项目集，因为Person类在二进制序列化这个项目中。



* 1. 因为2个项目在不同的命名空间，Person类需要是Public。

# 正则表达式

正则表达式是用来进行字符串处理的技术，是语言无关的。在几乎所有的语言中都有实现。

一个正则表达式就是由普通字符以及特殊字符（称为元字符）组成的字符模式。该模式用来描述待匹配的字符串的特征。正则表达式作为一个模板，将字符模式与所搜索的字符串进行匹配。

字符串的特征有3类：

1.必须出现的内容

2.可能出现的内容

3.不能出现的内容

观察字符串规律，根据规律总结特征，然后根据特定字符串的特征来编写正则表达式

## 元字符

元字符根据其作用可以分为3类。

* 匹配
* 限定
* 断言：只判断，不匹配。

### 匹配

1. .

匹配除\n之外的任何单个字符。例如正则表达式“a.b”能匹配如下字符："acb","adb","a郝b","a b",但是不匹配"accb"。

1. []

字符的筛选。匹配括号中的任何一个字符。括号中为字符集合。

例如正则表达式"a[bc]d",能匹配如下字符“abd”，“acd”，不能匹配“abcd”。

可以在括号中使用连字符"-"来指定字符的区间来简化表示。例如，正则表达式“a[0-9]b”,等价于“a[0123456789]b”，a与b之间可以是0到9之间的任何一个数字字符，[a-z]可以匹配任何小写字母，[a-zA-Z]这匹配任何字母。[^0-9a-z]则表示不能包含数字以及小写字母。

1. |

将两个匹配条件进行逻辑“或”运算。因为“|”的优先级比较低，正则表达式“z|food”的匹配项为：z或food（只要字符串中包含z或food既符合表达式）。而不是zood和food。（z|f）ood才表示zood和food。

1. （）

将小括号之间的表达式定义为“组（group）”，并且将匹配这个表达式的字符保存到一个临时区域。它具有改变优先级（见上例），定义提取组两个作用（）。如：^(z|food)$中包含的字符集为：z或food，这个表达式等同于 ^z$ 或^food$，这个表达式只有两个值符合即：z或food。

### 限定

1. \*

匹配0至多个在它之前的子表达式，和匹配符没有关系。例如正则表达式“zo\*”,能匹配“z”，“zo”，“zoo”，“zoooo”；“z（b|c）\*”匹配“z”,"zbc","zcb","zccc","zbbccc"。z(ab)\*能匹配z,zab,zabab。"z.\*"则表示z后能匹配任意字符。

1. +

表示前边的表示必须出现1次或者多次。

1. ？

表示前边的表达式必须出现0次或1次。它的另外一个作用是“终止贪婪模式”。

正则表达式默认是贪婪模式。

1. {n}

表示前边的表达式必须出现n次。

1. {n,}

表示前边的表达式至少出现n次

1. {n,m}

表达式前边的表达式至少出现n次，最多出现m次。

### 断言

1. ^

匹配一个正则表达式的开头。例如：^abc表示字符串必须以abc开始，如"abcdsrewre"；

1. $

匹配一个正则表达式的结束。例如：xyz$表示字符串结尾必须是xyz。

完全匹配：

判断手机号位数： ^[0-9]{11}$

判断邮政编码位数：^[0-9]{6}$

## 简写表达式

\d 代表一个数字，等同于[0-9]。 a\db =a[0-9]b

\D 代表非数字，等同于[^0-9]。 a\Db =a[^0-9]b

\s 代表换行符、Tab制表符等空白字符

\S 代表非空白字符（a$@）

\w 匹配字母或数字或下划线或汉子，即能组成单词的字符，不包含（a$@）。等价于

\W 等同于[^\w]

\b 单词的边界

注意这些简写表达式是不考虑转义符的。这里的\就表示字符\。在C#中需要使用@或者\双重转义。

下面的表达式表示任意一个字符出现在a和b之间

a[\d\D]b

a[\s\S]b

a[\w\W]b

## 贪婪模式

**贪婪模式:**正则表达式一般趋向于最大长度匹配，也就是所谓的贪婪匹配。如下面使用模式p匹配字符串str，结果就是匹配到：**abcaxc**(ab\*c)。

|  |
| --- |
| String str="abcaxc";  Patter p="ab\*c"; |

**非贪婪模式**：就是匹配到结果就好，就少的匹配字符。如上面使用模式p匹配字符串str，结果就是匹配到：**abc**(ab\*c)。

如何区分两种模式？

默认是贪婪模式；在元字符后面直接加上一个问号？就是非贪婪模式。

## 正则表达式转义符

正则表达式转义符把元字符转义为普通的字符。

例如想要匹配的字符是一个”.”符号；在正则表达式中”.”代表的是任意一个除换行外的字符。要匹配字符串”.“就需要加上转义符”\“。正则表达式中的”\.“表示的是字符”.“

在C#中，转义符用的也是”\“,字符串”\.“表示的是对”.“进行转义。C#中没有” \. “转义，所以会报错。正确的匹配字符的方式**是”\\.“**。第一个“\”表示的是转义第二个“\”为普通的”\“符号。这个是”[\\.“代表的意思就是正则表达式中的元字符](.%22%20)”.“被转义为普通的”.”符号。

|  |
| --- |
| while(true)  {  Console.WriteLine("请输入元字符：");  string msg = Console.ReadLine();  bool b = Regex.IsMatch(msg, "^\\.$");  Console.WriteLine(b);//只有输入符号.时才返回true  } |

如果想要匹配的是字符”\d”，应该怎么输入呢：需要“\\\\d”。首先，在正则表达式中，匹配“[\\d](d)”表示匹配的是字符串”\d”。在C#中4个”\”才表示2个普通字符”\”。也就是说“\\\\d”，在c#中转义为“\\d”，而“\\d” 在正则表达式式中被转义为普通字符串”\d”.

|  |
| --- |
| while(true)  {  Console.WriteLine("请输入元字符：");  string msg = Console.ReadLine();  bool b = Regex.IsMatch(msg, "^\\\\d$");  Console.WriteLine(b);//只有输入为\d时才返回true  } |

当匹配的字符串中有多个元字符需要转义时，可以用Escape方法进行转义

|  |
| --- |
| string s = Regex.Escape(@"^\d$");  Console.WriteLine(s);//输出^\d$的转义  string msg = Console.ReadLine(); |

## .Net中的正则表达式

正则表达式在.Net中就是用字符串表示。字符串的含义由Regex类内部进行语法分析。

### 判断是否匹配

1. 引入命名空间

|  |
| --- |
| using System.Text.RegularExpressions; |

1. 观察并总结字符串的规律

比如，在这里我们要判断输入的值是否在11到20这个区间。传统方法是把输入转为数字进行判断。

|  |
| --- |
| string msg = Console.ReadLine();  if(int.Parse(msg)>10 && int.Parse(msg)<=20)  {  Console.WriteLine("正确：值大小符合要求");  } |

而正则表达式则是对字符串进行处理。在这里我们首先要总结11到20区间所有数字字符有什么规律。

1. 有2个字符串
2. 首字符为1，则第二个字符为1-9。
3. 首字符为2，则第二个字符为0。

可以用下面的表达式来表示11到20这个区间所有的字符

|  |
| --- |
| ^(1[1-9]|20)$ |

1. IsMatch判断是否匹配.它判断的是字符串中是否包含正则表达式所限定的字符。

|  |
| --- |
| bool b = Regex.IsMatch(msg, "^(1[1-9]|20)$"); |

1. 完整代码如下：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Text.RegularExpressions;//引入命名空间  using System.Threading.Tasks;  namespace \_01\_判断是否匹配  {  class Program  {  static void Main(string[] args)  {  Console.WriteLine("请输入11-20之间的任意数字");  while (true)  {  string msg = Console.ReadLine();  if(int.Parse(msg)>10 && int.Parse(msg)<=20)  {  Console.WriteLine("正确：值大小符合要求");  }  //判断是否匹配  bool b = Regex.IsMatch(msg, "^(1[1-9]|20)$");  if (b)  {  Console.WriteLine("正确：值大小符合要求");  }  else  {  Console.WriteLine("错误：输入的数字大于20或小于11");  }  }  }  }  } |

关于正则表达式，能不用正则表达式就不用。

* 案例1：判断是否是邮政编码

|  |
| --- |
| //判断是否是邮政编码  bool b= Regex.IsMatch("100901", "^[0-9]{6}$");  //\d为[0-9]缩写，记得要加\进行转义。或者加@  bool b1 = Regex.IsMatch("100901","^\\d{6}$");  bool b1 = Regex.IsMatch("100901",@"^\d{6}$"); |

案例2：仅仅能输入半角数字

|  |
| --- |
| // 指定RegexOptions.ECMAScript选项后，第二个参数中的表达式只表示普通数字（ASCII字符），不再包含全角（Unicode）的字符了  bool b = Regex.IsMatch(msg, "^(1[1-9]|20)$",RegexOptions.ECMAScript); |

案例3：判断是否是身份证号

|  |
| --- |
| while (true)  {  Console.WriteLine("请输入你的身份证号：");  string ID = Console.ReadLine();  //1.　身份证为15位或18位，首位不能是0。  //2. 身份证为15位，全部是数字  //3.如果是18位，则前17位为数字，最后一位是数字也可能是x。  //15位与18位身份证分别判断  bool b1 = Regex.IsMatch(ID, @"^([1-9]\d{14}|[1-9][0-9]{16}[0-9Xx])");  bool b = Regex.IsMatch(ID, @"^([1-9]\d{14})$|^([1-9]\d{16}(\d|Xx))$");  //15位与18位身份证整体判断。用？  bool b2 = Regex.IsMatch(ID, @"^[1-9][0-9]{14}([0-9]{2}[0-9Xx])?$");  if (b2)  {  Console.WriteLine("身份证号码正确");  }  else  {  Console.WriteLine("请输入正确的身份证号码");  }  } |

案例4：判断是否合法的email

|  |
| --- |
| while (true)  {  Console.WriteLine("请输入你的email:");  string email = Console.ReadLine();  //在[]中的-表示连接。所以想要保护“-”的输出需要“\-”表示输出为“-”，而在C#中\表示为转义，所以这里需要添加两个“\\”  bool b = Regex.IsMatch(email, "^[0-9a-zA-Z\_.\\-]+@[0-9a-zA-Z]+(.[0-9a-zA-Z]){1,}$");  //第二种写法。  bool b2 = Regex.IsMatch(email, @"^\w+@\w+(\.[0-9a-zA-Z]){1,}$");  if (b2)  {  Console.WriteLine("邮箱格式正确");  }  else  {  Console.WriteLine("邮箱格式不正确");  }  } |

#### 习题

1. 匹配IP

|  |
| --- |
| while (true)  {  Console.WriteLine("请输入IP地址");  string msg = Console.ReadLine();  bool b = Regex.IsMatch(msg, @"^(\d{1,3}\.){3}\d{1,3}$");//0.0.0.0-999.999.999.999  Console.WriteLine(b);  } |

1. 匹配日期

|  |
| --- |
| while (true)  {  Console.WriteLine("请输入日期");  string msg = Console.ReadLine();  bool b = Regex.IsMatch(msg, @"^\d{1,4}(-[\d]{2}){2}$");  Console.WriteLine(b);  } |

1. 匹配网址

|  |
| --- |
| while (true)  {  Console.WriteLine("请输入网址");  string msg = Console.ReadLine();  bool b = Regex.IsMatch(msg, @"^[a-zA-Z]+://.+$");  Console.WriteLine(b);  } |

#### 反复匹配

当n个字符串与同一个表达式进行匹配时，用静态的Regex.IsMatch方法效率会变慢，因为Regex.IsMatch方法内部会创建一个Regex对象:

|  |
| --- |
|  |

反复匹配时，每一次都要创建一个正则表达式对象，影响性能。解决方法是，我们可以手动创建一个正则表达式对象，之后字符串反复与这个表达式进行对比。在对比的过程中，只创建了一次正则表达式对象。

|  |
| --- |
| //1.创建正则表达式  Regex regex = new Regex(@"^\d{6}$", RegexOptions.ECMAScript);  while (true)  {  Console.WriteLine("请输入邮政编码");  string postcode = Console.ReadLine();  //判断是否匹配表达式，参数为要匹配的字符串。  if (regex.IsMatch(postcode))  {  Console.WriteLine("ok");  }  else  {  Console.WriteLine("wrong");  }  } |

### 提取

提取是指从字符串、文件或字符组中提取出符合正则表达式的特定字符串

#### Match

Match方法从所有符合正则表达式中提取出其中一个字符串，默认的是提取第一个出现的字符串。

|  |
| --- |
| string msg = "2015年4月23日；2015年6月1日";  Match match = Regex.Match(msg, @"\d+");  Console.WriteLine(match);//输出为：2015  Console.ReadLine(); |

当文件内容特别多时，可以用Match方法一个个提取：

|  |
| --- |
| //1.读取文件  string msg = File.ReadAllText("提取Email.htm");  Regex regex = new Regex(@"\d+", RegexOptions.ECMAScript);  Match match = regex.Match(msg);  Console.WriteLine(match);  while (match.Length != 0)//匹配d长度为0，表示到文件的末尾了，如果还继续进行就会从第一个开始再次进行匹配。d  {  //第二个参数为开始匹配的偏移量  match = regex.Match(msg, match.Index + match.Length);  Console.WriteLine(match);  }  Console.ReadLine(); |

#### Matches

提取所有符合正则表达式的字符串到一个集合中

|  |
| --- |
| string msg = File.ReadAllText("提取Email.htm");  //读取所有符合正则表达式的字符串到集合中  MatchCollection match = Regex.Matches(msg, @"\d+");  foreach (Match item in match)  {  Console.WriteLine("匹配的索引：" + item.Index);  Console.WriteLine("匹配的长度：" + item.Length);  Console.WriteLine("匹配的值：" + item.Value);  }  Console.ReadLine(); |

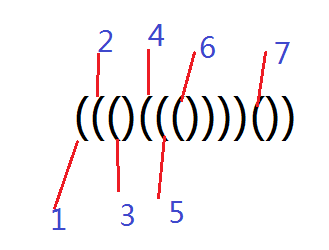
#### 案例：提取网页中的邮箱

|  |
| --- |
| //1.读取文件  string html = File.ReadAllText("提取Email.htm");  //2.创建正则表达式进行提取  MatchCollection match = Regex.Matches(html, @"[-0-9a-zA-Z\_.]+@[-0-9a-zA-Z]+(\.[0-9a-zA-Z])+");  //3.输出  foreach (Match item in match)  {  Console.WriteLine(item.Value);  }  Console.WriteLine(match.Count);//匹配的总数量  Console.ReadLine();  } |

#### 提取组

提取组的思路是：先写一个能满足整个字符串的正则表达式，然后再正则表达式中用括号将那些你想要提取的内容括起来，这样你就可以提取出想要的组了。

正则表达式中用（）将要提取的内容括起来，括起来的内容就会被提取到一个组中，可以通过Match的Groups来得到所有要提取的元素。Groups的序号是从1开始的，0是匹配了整个表达式的字符串。



当多个”（）”嵌套的时候，组的序号是从最左边的“（”开始，向右。如右边的提取组的序号：

用$1、$2、……表示正则表达式里面第一个、第二个、……提取组里面的匹配内容

在13.4.2.3案例：提取网页中的邮箱中，我们提取出网页中所有的邮箱，我想统计每类邮箱出现的次数，怎么办呢？这时就可以借助”（）“提取组的功能了。

在邮箱的正则表达式中，我们把@后面的提取成一个组：

|  |
| --- |
| [-0-9a-zA-Z\_.]+@([-0-9a-zA-Z]+(\.[0-9a-zA-Z])+) |

之后创建一个字典集合来统计每种邮箱出现的次数。

|  |
| --- |
| //1.读取文件  string html = File.ReadAllText("提取Email.htm");  //2.创建正则表达式  MatchCollection match = Regex.Matches(html, @"[-0-9a-zA-Z\_.]+@([-0-9a-zA-Z]+(\.[0-9a-zA-Z]+))");  //3.输出  Dictionary<string, int> dirc = new Dictionary<string, int>();  foreach (Match item in match)  {  //从提取的邮箱字符串中以组方式提取符合邮箱后缀的字符串  string str = item.Groups[1].value.ToString();  if (dirc.ContainsKey(str))  {  dirc[str] += 1;  }  else  {  dirc.Add(str, 1);  }  }  foreach (var item in dirc)  {  Console.WriteLine("邮箱：{0},出现的次数{1}", item.Key, item.Value);  }  Console.ReadLine(); |

#### 案例：从不规则的字符串中提取出年月日

|  |
| --- |
| string date = "june 1 , 2015 ";  //把年月日提取到组中  Match match = Regex.Match(date, @"^\s\*([a-zA-Z]+)\s\*([0-9]{1,2})\s\*,\s\*([0-9]{4})\s\*$");  if(match.Success)  {  Console.WriteLine("{0}\_{1}\_{2}",match.Groups[3].Value,match.Groups[2].Value,match.Groups[1].Value);  }  Console.ReadLine(); |

案例：从邮箱中提取账户名称和域名

|  |
| --- |
| string email = "hyfdbd@163.com.cn";  Match match = Regex.Match(email, @"^([-0-9a-zA-Z.\_]+)@([-0-9a-zA-Z.]+(\.[0-9a-zA-Z]+)+)$");  if(match.Success)  {  Console.WriteLine("用户名：{0}；邮箱域名：{1}",match.Groups[1].Value,match.Groups[2].Value);  }  Console.ReadLine(); |

# 程序诊断

.Net提供了System.Diagnostics 命名空间提供类，使我们能够与系统进程、事件日志和性能计数器进行交互。

## Stopwatch（秒表类）

Stopwatch类提供一组方法和属性，可用于准确地测量运行时间。

下面的代码记录了从开始记时到结束时程序运行的时间

|  |
| --- |
| Stopwatch stopWatch = new Stopwatch();  stopWatch.Start();//开始记时  Thread.Sleep(10000);  stopWatch.Stop();//结束记时  // Get the elapsed time as a TimeSpan value.  TimeSpan ts = stopWatch.Elapsed;  // Format and display the TimeSpan value.  string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",  ts.Hours, ts.Minutes, ts.Seconds,  ts.Milliseconds / 10);  Console.WriteLine("RunTime " + elapsedTime); |

# 线程（Thread）

## 进程、应用程序域、线程之间关系

进程（Process）：是Windows系统中的一个基本概念，它包含着一个运行程序所需要的资源。进程之间是相对独立的，一个进程无法直接访问另 一个进程的数据（除非利用分布式计算方式），一个进程运行的失败也不会影响其他进程的运行，Windows系统就是利用进程把工作划分为多个独立的区域 的。进程可以理解为一个程序的基本边界。

要解决的问题：为了使程序能并发执行，（要并发执行就要隔离进程，使进程独立，即每个进程有属于自己的数据段、程序段、进程控制块）

操作系统分配资源（内存、cpu、存储硬盘、带宽等）的最小单位是进程。正常情况下进程之间是隔离的。

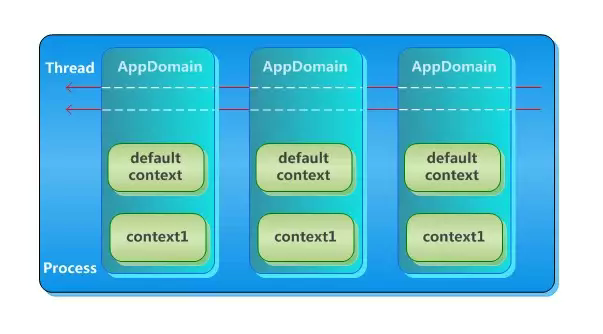
应用程序域：它提供安全而通用的处理单元，公共语言运行库可使用它来提供应用程序之间的隔离。您可以在具有同等隔离级别（存在于单独的进程中）的单个进程中运行几个应用程序域，而不会造成进程间调用或进程间切换等方面的额外开销。在一个进程内运行多个应用程序的能力显著增强了服务器的可伸缩性。

优势： 在一个应用程序中出现的错误不会影响其他应用程序。能够在不停止整个进程的情况下停止单个应用程序。应用程序域形成了托管代码的隔离、卸载和安全边界。 在任意给定时间，每一线程都在一个应用程序域中执行

线程：是Windows任务调度的最小单位。线程是程序中的一个执行流，每个线程都有自己的专有寄存器(栈指针、程序计数器等)，但代码区是共享的，即不同的线程可以执行同样的函数。

每个线程都要指向一个方法体，方法完成后，线程就释放

解决问题：进程是一个资源的拥有者，因而在进程的创建、撤销、和切换的过程中，系统必须为之付出较大的时空开销，限制了并发程度的进一步提高。



## 前台线程和后台线程

前台线程：只有所有的前台线程都关闭才能完成程序关闭。

后台线程：只要所有的前台线程结束，后台线程自动结束。

## 进程

|  |
| --- |
| //用默认浏览器打开网址  Process.Start("http://www.baidu.com");  //打开服务管理  Process.Start("services.msc");  //打开绘图时打开1.jpg  Process.Start("mspaint", "C:\\1.jpg"); |

## 线程执行流程

1. 新建Thread对象，并传入一个委托方法。

|  |
| --- |
| Thread thread = new Thread(() =>  {  while (true)  {  Console.WriteLine(DateTime.Now);  Thread.Sleep(1000);  }  }); |

2. 设置线程为后台线程

|  |
| --- |
| //线程默认是前台线程，设置为后台线程，当主线程执行完毕后，程序会退出，后台线程不会阻塞程序退出  thread.IsBackground = true; |

3. 在主线程中启动线程

|  |
| --- |
| //启动线程  thread.Start(); |

4. 线程的生命周期

当线程中委托方法执行结束后，线程自动结束

5. 强制终止线程

|  |
| --- |
| thread.Abort()//强制停止线程，不推荐使用。可能造成资源没有释放。 |

## 暂停线程

|  |
| --- |
| Thread.Sleep(1000); |

线程暂停1秒

## 线程ID

|  |
| --- |
| int j = thread.ManagedThreadId;  int k = Thread.CurrentThread.ManagedThreadId;  Console.WriteLine("线程的ID是：" + j);  Console.WriteLine("主线程的ID是：" + k); |

## 完整案例

|  |
| --- |
| //CLR开始会创建一个默认的主线程（前台线程），指向了main方法  static void Main(string[] args)  {  Thread thread = new Thread(() =>  {  while (true)  {  Console.WriteLine(DateTime.Now);  Thread.Sleep(1000);  }  });  //线程默认是前台线程，设置为后台线程，当主线程执行完毕后，程序会退出，后台线程不会阻塞程序退出  thread.IsBackground = true;  //启动线程  thread.Start();  int j = thread.ManagedThreadId;  int k = Thread.CurrentThread.ManagedThreadId;  Console.WriteLine("线程的ID是：" + j);  Console.WriteLine("主线程的ID是：" + k);  Console.Read();  int i = 0;  while (i < 10)  {  Console.WriteLine("----主线程----------");  Thread.Sleep(1000);  i++;  }  } |

## 线程与winform

当主窗体中点击某个按钮执行一个循环、大文件读写等耗费大量时间的操作时，主界面会没有响应。

这时我们就可以把需要大量时间进行运算的操作作为一个进程，放到后台运行，不会影响主界面中其他任务的执行。

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  //创建线程  Thread thread = new Thread(() =>  {  while (true)  {  //判断创次建控件的线程是否是当前线程  if (this.button1.InvokeRequired)  {  //找到创建button1控件的线程，执行委托方法  button1.Invoke(new Action<string>(s =>  {  this.button1.Text = s;  }), DateTime.Now.ToString()  );  }  }  });  thread.IsBackground = true;  thread.Start();  } |

## 跨线程调用

需要用invoke方法

|  |
| --- |
| //判断创次建控件的线程是否是当前线程  if (this.button1.InvokeRequired)  {  //找到创建button1控件的线程，执行委托方法  button1.Invoke(new Action<string>(s =>  {  this.button1.Text = s;  }), DateTime.Now.ToString()  );  } |

## 案例：摇奖机

初始化窗体

|  |
| --- |
| List<Label> lblist = new List<Label>();  bool isRunning = false;  private void Form1\_Load(object sender, EventArgs e)  {  string init = "摇奖机第一版";  for (int i = 0; i < 6; i++)  {  //创建lable  Label lable = new Label();  //初始化  lable.Text = init[i].ToString();  //自动大小  lable.AutoSize = true;  //设置字体  lable.Font = new System.Drawing.Font(Font.FontFamily, 25);  //字体颜色  lable.ForeColor = Color.Red;  //位置  lable.Location = new Point(40 \* (i + 1) + 50, 100);  //添加到集合  lblist.Add(lable);  //添加到窗体  this.Controls.Add(lable);  }  } |

点击按钮开始摇奖

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  //根据按钮的tag确定是否停止进程  if (button1.Tag.ToString() == "0")  {  isRunning = true;//进程执行  button1.Tag = "1";  button1.Text = "暂停";  }  else  {  isRunning = false;//进程停止  button1.Tag = "0";  button1.Text = "开始";  }  Thread thread = new Thread(() =>  {  Random rand = new Random();  while (isRunning)//isRunning为true是进程一致执行，否则进程会执行完毕后结束。  {  foreach (var item in lblist)  {  string str = rand.Next(0, 10).ToString();  //跨进程调用  if (item.InvokeRequired)  {  item.Invoke(new Action<string>(s => item.Text = s), str);  }  else  {  item.Text = str;  }  }  Thread.Sleep(100);  }  });  thread.IsBackground = true;  thread.Start();  } |

## 进程池

### 为什么要有连接池

1. 链接对象创建非常消耗资源（需要校验等）
2. 对象的重用

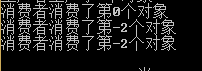
### 生产者和消费者的问题

生产者往连接池中放入对象，消费者从连接池中取出对象。当同时有多个消费者从连接池中取对象时，就可能造成索引越界、对象不存在等问题。

下面模拟一个连接池

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  //创建连接池  MyPool pool = new MyPool();  //索引默认指向第一个元素之上。  pool.index = -1;  //生产者创建链接对象。  for (int i = 0; i < 20; i++)  {  Thread thread = new Thread(() =>  {  while (true)  {  if (pool.index < 20)  {  //向当前执行的下一个元素中写入对象。  pool.pools[pool.index + 1] = new MyConnection();  //执行下一个  pool.index++;  Console.WriteLine("生产者生产了{0}个对象", pool.index);  }  Thread.Sleep(1000);  }  });  thread.IsBackground = true;  thread.Start();  }  //消费者消费链接对象。  for (int j = 0; j < 20; j++)  {  Thread thread = new Thread(() =>  {  while (true)  {  if (pool.index >= 0)  {  //取出对象，元素值为null  pool.pools[pool.index] = null;  //指向上一个。  pool.index--;  Console.WriteLine("消费者消费了第{0}个对象", pool.index);  }  Thread.Sleep(200);  }  });  thread.IsBackground = true;  thread.Start();  }  Console.Read();  }  }  class MyPool  { //连接池  public MyConnection[] pools = new MyConnection[100];  public int index = 0;  }  class MyConnection  {  } |

执行后输出如下：



在上面的输出中有2个问题：

1. 消费了第-2个对象。这个对象不存在连接池中
2. 两个消费者同时消费了第-2个对象。一个对象被两个对象同时消费。

## 锁

解决多个线程、多个用户访问同一资源的问题。

### 数据库中的锁优化

在数据库中修改、删除、添加时加上排它锁（x锁）。查询时会加上共享锁（s锁）。

开发中避免死锁

1. sql操作表的顺序尽量一致
2. 数据要求不严格时，在查询中添加with(nolock)
3. 链接（join）非常多而引起的死锁：使用临时表。让一个原始表跟临时表进行连接查询产生临时表2，再让原始表2与临时表2进行连接查询，产生临时表3，以此方式进行用空间换取更少的锁。

### 线程锁

|  |
| --- |
| lock ()//lock后面要跟要跟一个引用类型的实例。锁住同一个对象（引用地址）就会互斥。 |

Lock执行时会检查实例的同步索引块，如果不为-1则表示没有上锁，就可以执行，并使同步索引块指向一个同步索引数组，下一个线程执行时到这里时，检查到实例的同步索引快不为-1就等待。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading;  using System.Threading.Tasks;  namespace \_05\_模拟连接池  {  class Program  {  static void Main(string[] args)  {  //创建连接池  MyPool pool = new MyPool();  //初始化连接池索引  pool.index = -1;  //生产者创建链接对象。  for (int i = 0; i < 20; i++)  {  Thread thread = new Thread(() =>  {  while (true)  {  lock (pool)//锁  {  if (pool.index < 20)  {  pool.pools[pool.index + 1] = new MyConnection();  pool.index++;  Console.WriteLine("生产者生产了{0}个对象", (pool.index+1));  }  }  Thread.Sleep(500);  }  });  thread.IsBackground = true;  thread.Start();  }  //消费者消费链接对象。  for (int j = 0; j < 20; j++)  {  Thread thread = new Thread(() =>  {  while (true)  {  lock (pool)//lock后面要跟要跟一个引用类型的实例。锁住同一个对象（引用地址）就会互斥。  {  if (pool.index >= 0)  {  pool.pools[pool.index] = null;  pool.index--;  Console.WriteLine("消费者消费了第{0}个对象", (pool.index+1));  }  }  Thread.Sleep(200);  }  });  thread.IsBackground = true;  thread.Start();  }  Console.Read();  }  }  class MyPool  { //连接池  public MyConnection[] pools = new MyConnection[100];  public int index = 0;  }  class MyConnection  {  }  } |

## 线程池

### 使用线程池

线程池对象是个静态类。不需要我们手动创建线程，只需要放入工作任务请求队列，当有可用线程时，自动执行队列中的工作任务。

|  |
| --- |
| //QueueUserWorkItem把工作任务（有参无返回值的方法）放入线程池，当有可用线程时执行该任务。第二个参数为工作任务的需要的参数  ThreadPool.QueueUserWorkItem((s) =>  {  Console.WriteLine(s);  },"hhhh");  Console.ReadKey(); |

线程中的线程是不可控的。当工作任务放入工作任务请求队列时是先进先出的，但是获取了任务的线程是由操作系统控制的，线程执行顺序不可控，此外各个工作线程获取工作项放到线程栈后执行顺序是后进先出的，所以不能保证整体的请求项之间是请求处理的顺序。

|  |
| --- |
| for (int i = 0; i < 100; i++)  {  ThreadPool.QueueUserWorkItem((j) =>  {  Console.WriteLine(j);  }, i);  }  Console.ReadKey(); |

输出顺序不是从0到99，而是不固定的



### 线程池的优点

线程切换消耗资源,cpu在切换线程的时候，需要把当前线程执行的状态保持到寄存器里面去。

线程创建非常消耗资源。线程创建非常慢，占用大量的内存空间。每个线程最少1M内存开销。

线程池提高了线程的利用率，非常适合工作任务非常小，而且又需要使用单独的线程来解决的问题。

|  |
| --- |
| Stopwatch watch = new Stopwatch();  watch.Start();  //手动线程  for (int i = 0; i < 100; i++)  {  new Thread(() =>  {  i++;//非常小的任务。  }  ).Start();  }  watch.Stop();  Console.WriteLine(watch.Elapsed.TotalSeconds);  watch.Restart();  //线程池  for (int j = 0; j < 100; j++)  {  ThreadPool.QueueUserWorkItem((s) =>  {  j++;//非常小的任务。  });  }  watch.Stop();  Console.WriteLine(watch.Elapsed.TotalSeconds);  Console.ReadKey(); |

效率：



### 什么时候用线程池？什么时候用手动创建线程

1、能用线程池的就用线程池，处理顺序不确定。

2、我们想手动关闭线程的话那么必须手动创建线程。 Abort（） Join（）

3、我们需要对线程池的线程的优先级做设置的情景下，只能使用手动创建线程。

4、如果执行的线程执行时间特别长。线程池：适合做大量小运算。

### 最大、最小线程数

每一个核心最优效率的线程数是：255。

可以用下列方法获得当前CPU运行的最小与最大线程数。

|  |
| --- |
| int MaxT = 0;  int RunT=0;  //MaxT 为可以用的最大线程数，RunT为异步IO线程的最大数目  ThreadPool.GetMaxThreads(out MaxT, out RunT);  Console.WriteLine(MaxT+" "+ RunT);  //MaxT 线程池中最小的线程数，RunT线程池中异步IO线程的最小数目  ThreadPool.GetMinThreads(out MaxT, out RunT);  Console.WriteLine(MaxT + " " + RunT);  Console.ReadKey(); |

## 异步委托

异步调用委托的实质就是使用一个线程池的线程去执行委托指向的方法。

### 普通异步调用委托

步骤：

1. 委托变量指向一个方法
2. 委托变量调用BeginInvoke方法开始异步委托
3. 委托变量调用EndInvoke方法获得异步调用方法的执行结果

|  |
| --- |
| Console.WriteLine("主线程ID: " + Thread.CurrentThread.ManagedThreadId);  //1.委托变量delFunc指向方法  Func<int, int, string> delFunc = (a, b) =>  {  Console.WriteLine("调用异步委托的线程ID: " + Thread.CurrentThread.ManagedThreadId);  Thread.Sleep(2000);  return (a + b).ToString();  };  //2.异步调用委托。实质就是使用一个线程池的线程去执行委托指向的方法  //参数1：传入委托方法中的第一个参数  //参数2：传入委托方法中的第二个参数  //参数3：回调函数  //参数4：传入回调函数的参数  //返回值为异步执行的状态  IAsyncResult result = delFunc.BeginInvoke(5, 4, null, null);  //3.取得异步执行的结果  //EndInvoke会阻塞当前的线程，直到异步委托执行完成之后，才能继续往下执行。  //EndInvoke取得异步委托的结果。  string str = delFunc.EndInvoke(result);  Console.WriteLine("------这条语句在异步委托执行完毕后才会显示---------");  Console.WriteLine(str);  Console.ReadKey(); |

### 有回调函数的异步委托

回调函数实质上是一个委托

|  |
| --- |
| public delegate void AsyncCallback(IAsyncResult ar); |

回调函数在异步委托方法执行完成之后，再来掉回调函数。调用回调函数的线程和调用异步委托方法的是同一个线程。

我们先创建一个方法：

|  |
| --- |
| private static void MyAsynCallback(IAsyncResult ar)  {  Console.WriteLine("执行回调函数的线程ID: "+ Thread.CurrentThread.ManagedThreadId);  } |

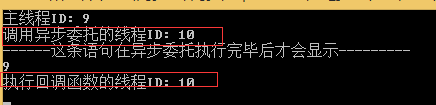
在BeginInvoke中传入回调函数

|  |
| --- |
| IAsyncResult result = delFunc.BeginInvoke(5, 4,MyAsynCallback,"hello"); |

完整代码：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading;  using System.Threading.Tasks;  namespace \_08\_异步委托  {  class Program  {  static void Main(string[] args)  {  Console.WriteLine("主线程ID: " + Thread.CurrentThread.ManagedThreadId);  //1.委托变量delFunc指向方法  Func<int, int, string> delFunc = (a, b) =>  {  Console.WriteLine("调用异步委托的线程ID: " + Thread.CurrentThread.ManagedThreadId);  Thread.Sleep(2000);  return (a + b).ToString();  };  //2.异步调用委托。实质就是使用一个线程池的线程去执行委托指向的方法  //参数1：传入委托方法中的第一个参数  //参数2：传入委托方法中的第二个参数  //参数3：回调函数  //参数4：传入回调函数的参数  //返回值为异步执行的状态  IAsyncResult result = delFunc.BeginInvoke(5, 4,MyAsynCallback,"hello");    //3.取得异步执行的结果  //EndInvoke会阻塞当前的线程，直到异步委托执行完成之后，才能继续往下执行。  //EndInvoke取得异步委托的结果。  string str = delFunc.EndInvoke(result);  Console.WriteLine("------这条语句在异步委托执行完毕后才会显示---------");  Console.WriteLine(str);  Console.ReadKey();  }  private static void MyAsynCallback(IAsyncResult ar)  {  Console.WriteLine("执行回调函数的线程ID: "+ Thread.CurrentThread.ManagedThreadId);  }  }  } |

执行结果：



### 回调函数取得异步委托方法的结果和回调函数的参数

有两种方式：

|  |
| --- |
| //1. 获得异步委托执行的结果  AsyncResult result = ar as AsyncResult;  var del = result.AsyncDelegate as Func<int, int, string>;  string returnValue = del.EndInvoke(result);  Console.WriteLine(returnValue);  //2.获得回调函数的参数  Console.WriteLine("传给回调函数的参数是："+result.AsyncState);  Console.WriteLine("执行回调函数的线程ID: "+ Thread.CurrentThread.ManagedThreadId); |

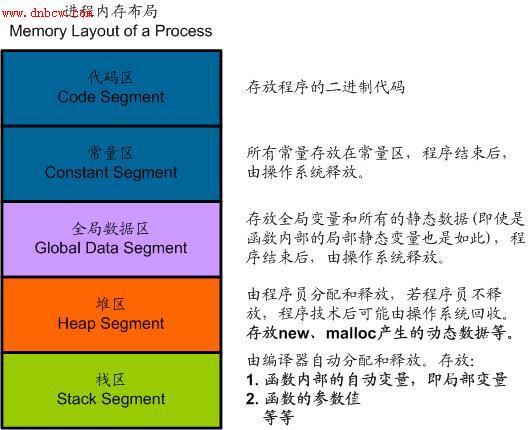
第二种方式更简单，就是把委托变量作为回调函数的参数。

|  |
| --- |
| //通过参数获得委托方法  var del = ar.AsyncState as Func<int, int, string>;  //异步调用委托  string str = del.EndInvoke(ar);  Console.WriteLine(str);  Console.WriteLine("回调函数传入的参数类型为：" + ar.AsyncState.ToString());  Console.WriteLine("执行回调函数的线程ID: " + Thread.CurrentThread.ManagedThreadId); |

# 内存分布

## 一个进程在内存中的布局

对于一个完整的程序，在内存中分布情况如下图：



值得一提的是：在C#中struct存放在栈区，class存放在堆区，所以一般struct要比class要快的。

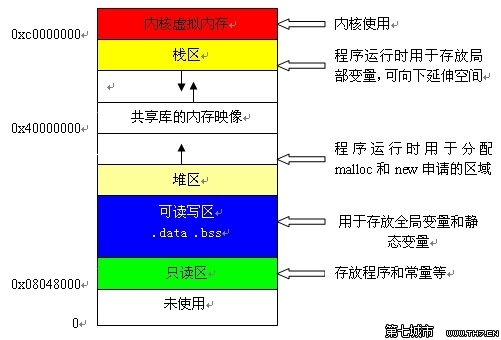
堆和栈的比较表(仅针对C++)

|  |  |  |
| --- | --- | --- |
|  | 栈(Stack) | 堆(Heap) |
| 申请方式 | 由OS自动分配。例如在函数声明一个局部变量int b; OS自动在栈中为b开辟空间 | 需要程序员自己申请，并指明大小，在c中malloc函数，如p1 = (char\*)malloc(10); 在C++中用new运算符如p2 = new char[10];  注意：p1和p2本身是在栈中的 |
| 申请后系统响应 | 只要栈的剩余空间大于所申请的空间，系统将为程序提供内存，否则将报异常提示栈移除。 | 首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会 遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内 存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大 小，系统会自动的将多余的那部分重新放入空闲链表中。 |
| 申请大小的限制 | 栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址 和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M (也有的说是1M，总之是一个编译时就确定的常数)，如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。 | 堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。 |
| 申请的效率 | 栈由系统自动分配，速度较快。但程序员是无法控制的。 | 堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片，不过用起来最方便。  另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，它不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。 |
| 存储内容 | 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。  当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。 | 一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。 |
| 存取效率 | 块 | 慢 |

## 内存分配方式

程序运行时的**内存分配方式**有三种：  
（1） 静态内存分配。如全局变量，static变量，所需空间在程序编译的时候就已经分配好，在程序的整个运行期间都存在。

（2） 栈上内存分配。如局部变量，在执行函数时，函数中的局部变量都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。  
（3） 堆上内存分配。亦称动态内存分配，程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。



# 垃圾回收

垃圾回收是CLR的一个核心功能，它的目的是提高内存利用率。

垃圾回收器，只回收托管堆中的内存资源，不回收其他资源（值类型变量，数据库连接，文件句柄，网络端口等）。

什么样的对象才会被回收？

没有变量引用的对象。好比吃完饭，结账走人了，服务员才会回收。

什么时间回收？

不确定，当程序需要更新内存的时候开始回收。好比大学食堂，盘子摆满餐桌，或者服务员会不定期的回收。

用GC.Collect()手动调用垃圾回收器。不建议使用。好比，我们都叫服务员收盘子，服务员跑来跑去收盘子，实际上降低服务员的工作效率。

## 垃圾回收机制

早期的C/C++开发中，一个对象的生命周期大概像这样：计算对象大小——查找可用内存——初始化对象——使用对象——摧毁对象。如果在上面的过程中，开发人员忘记了“摧毁对象”这一步骤，则很有可能导致内存泄露！这是一个非常可怕的事情！幸好，CLR的开发人员为我们解决了这一问题，在.NET Framework中引入了垃圾回收机制，使得开发人员不需要再过多地关注内存释放的问题，CLR会在合适的时候进行执行垃圾回收来释放不再使用的内存。这里就像一个邪恶的男人所说的话：给我一个女人，我能创造一个民族！其实一个新世界你都可以去创造，前提是要有一个足够大的星球内存来容纳你的子孙！CLR就是这么认为的。

在激活一个进程时，CLR会先保留一块连续的内存，在主线程启动过程中，可能会初始化一系列对象，CLR先计算对象大小及其开销所占用的字节数，接着会在连续的内存块中为这些对象分配内存，这些对象被配置在第0代内存，在构造第0代内存的时候会分配一个默认大小的内存，随着程序的运行，可能会初始化更多的对象，CLR发现第0代内存不能装载更多的新生对象，此时CLR会启动垃圾回收器对第0代内存进行回收，不再使用的对象所占用的内存会被释放，接着把0代对象提升为第1代，然后把新生对象配置在第0代内存区中。CLR使用了3个阶段的代，每次新分配的对象都会被配置在第0代内存中，最老的对象在第2代内存中，每次为新对象分配内存时，都可能会进行垃圾回收以释放内存，很显然CLR认为“内存永远也使用不完”，很显然CLR为我们自动管理了内存垃圾，很显然CLR的这个“认为”在我们开发人员看来是不成立的，我们从以下几个方面来解读垃圾回收机制。

## 内存分配

垃圾回收是对引用类型而言的。

CLR要求引用类型的对象从托管堆中分配内存的，值类型是从栈中分配内存。在C#中通常使用new操作符来创建一个对象，编译器将会在IL中生成newobj指令，执行一个newobj指令会有以下过程：（在前一节中我们已经知道，在一个进程启动时会先保留一个连续的内存块）先计算类型及其基类型的字段所需要的字节数A，再计算类型对象的指针和一个同步索引块共8或16个字节，到此总共需要（A+8或18）字节的内存，CLR会检查当前进程区是否有足够的内存来容纳（A+8或16）个字节的对象，如果有，则将新对象放其中，否则CLR进行垃圾回收，释放不再使用的内存来容纳新的对象，在整个进程的生命周期中，CLR会维护一个指针P，它一直指向当前进程所分配的最后一个对象内存的结尾处而不会跑出当前进程内存区边界，如图：



每次计算新的将要创建的对象所需要的字节数时，CLR都是通过P加上新的需要的字节数进行检查可用内存区，如果超出了地址末尾，则表示当前的托管堆已经被用完，准备进行垃圾回收了。由于进程拥有一个独立连续的内存区，所以CLR能保证创建的新对象基本上都是紧挨着放置的。

## 代

当托管堆的内存被用完，新生的对象无处放置时，CLR就要开始进行垃圾回收了，随着程序的持续运行，托管堆可能越来越大，如果要对整个托管堆进行垃圾回收（下面会讲到如何回收），势必会严重影响性能，因为有时可能仅仅需要数十个字节就能容纳新的对象，有时候可能要对可达的对象进行搬迁，为了小范围有目的性地进行垃圾回收，CLR使用了“代”概念来优化垃圾回收器，代是垃圾回收机制使用的一个逻辑技术，也是一种算法，它把托管堆中的内存分为3个代（截止到目前.NET Framework4.0有3个代：0、1、2用GC.MaxGeneration返回最大的代的值）。

进程在初始化时，CLR为托管堆初始化为包含0个对象的一块内存区域，新添加到堆中的对象为第0代对象，CLR在初始化第0代内存区时会分配一个默认的配额，假设为512K，不同的.NET框架和版本，可能这个配额不相同。假设进程及其线程初始化完成后分配了4个对象，如下图：



这4个对象占据了512K的内存，程序继续运行，当再分配第5个对象Obj5的时候，发现第0代已无可用内存，此时CLR会启动垃圾回收器进行垃圾回收，假如上面的Obj3已经无效，此是Obj3的内存会被释放出来，接着搬迁Obj4对象到Obj3的位置（在Obj2的内存地址末尾处），存活下来的对象Obj1、Obj2和Obj4会被提升为第1代对象，第1代的内存区域根据程序运行的情况，CLR可能会为其分配20M（也可能是其他值）大小的内存区，第0代内存暂时为空，接着将Obj5分配到第0代内存区，如下：



程序继续运行，并又新分配了4个对象Obj6-Obj9，且此时Obj2和Obj5都不再使用，即为不可达对象，此时需要再创建一个新对象Obj10，但发现第0代的512K内存已经用完，所以CLR再一次启动垃圾回收器进行垃圾回收，这一次垃圾回收器会认为第0代的新对象生命周期短，所以先对第0代进行回收，并将存活对象提升到第1代中，垃圾回收器发现此时第1代中的对象远远小于20M，所以放弃对第1代的回收，程序继续运行，分配N多的新对象，当把第0代的对象提升到第1代，而第1代对象超20M时，则会对第1代的对象进行回收，第1代存活的对象被提升为第2代，第0代存活的对象被提升为第1代，如下图：



每一次垃圾回收的过程，垃圾回收器会根据实际使用情况自动调整第0、1、2代的默认配额大小，比如可能将第2代调整为200M，几分钟过后可能将其调整为120M，也有可能是1024M，程序继续运行，当对3个全部进行了垃圾回收且重新调整配额后，可用内存还不足以放置新对象，CLR就会抛出OutOfMemoryException异常，此时活神仙也无法施救了。原来CLR认为“内存永远也使用不完”也是有条件的啊！

## 垃圾回收过程

托管堆中的一个对象，当线程中有变量对其引用则为可达对象，否则为不可达对象。

在一次垃圾回收过程开始时，垃圾回收器会认为堆中的所有对象都是垃圾。

第一步是标记对象，垃圾回收器沿着线程栈上行检查所有根，静态字段、方法参数、活动中的局部变量以及寄存器指向的对象等都是根，当发现有根引用了托管堆中的对象A时，垃圾回收器会对此对象A进行标记，在标记A时，如果检测到对象A内又引用了另一个对象B，则也对B进行标记，对一个根检测完毕后会接着检测下一个根，执行同样的标记过程，代码中很有可能多个对象中引用了同一个对象C，垃圾回收器只要检测到对象C已经被标记过，则不再对对象C内所引用的对象进行检测，以防止无限循环标记。有标记的对象就是可达对象，未标记的对象就是不可达对象。

第二步是搬迁对象压缩堆，垃圾回收器遍历堆中的所有对象来寻找未标记的对象，因为未标记的对象是垃圾对象，可以进行回收，如果发现对象较小，则忽略，否则会先释放这些垃圾对象所占的内存，再把可达对象搬迁到这里以压缩堆，在搬迁可达对象之后，所有指向这些对象的变量将无效，接着垃圾回收器要重新遍历应用程序的所有根来修改它们的引用。在这个过程中如果各个线程正在执行，很可能导致变量引用到无效的对象地址，所以整个进程的正在执行托管代码的线程是被挂起的。

其实在垃圾回收器准备开始一次回收时，正在执行托管代码的所有线程都必须被挂起，挂起时，CLR会记录每个线程的指令指针以确定线程当前执行到哪里以便将来在垃圾回收结束后进行恢复。如果一个线程的指令指针恰好到达了一个安全点，则可以挂起该线程，否则CLR会尝试劫持该线程，如果还未到达安全点，则等待几百毫秒后CLR会尝试再一次劫持该线程，有可能经过多次尝试，最终挂起该线程，当当前进程的所有执行托管代码的线程都挂起后，垃圾回收器就可以开始工作了。（有关线程劫持可查找相关资料）。垃圾回收器回收完毕后，CLR恢复所有线程，程序继续运行。可见，垃圾回收对性能影响之巨大！

## 大对象

在创建新对象时，任何大于等于85000字节的对象都被认为是大对象，这些对象的内存是从大对象堆中分配的，大对象总是被认为是第2代对象，要尽量避免分配大对象来减少性能损伤，为了提高性能，垃圾回收器不对大对象进行搬迁压缩，只在回收第2代内存时进行回收。

## 手工进行回收

一般的情况下，CLR会智能地在必要的时候更行垃圾回收，但我们也可以在我们愿意的情况下手动启动垃圾回收器，System.GC类提供了重载版本的静态方法来启动垃圾回收器：

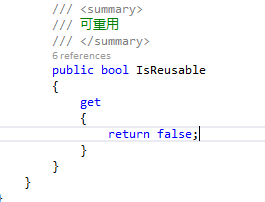
|  |
| --- |
| //对所有代进行垃圾回收。  GC.Collect();  //对指定的代进行垃圾回收。  GC.Collect(int generation);  //强制在 System.GCCollectionMode 值所指定的时间对零代到指定代进行垃圾回收。  GC.Collect(int generation, GCCollectionMode mode); |

参考网站http://www.cnblogs.com/solan/archive/2012/08/24/CSharp11.html

# 程序集

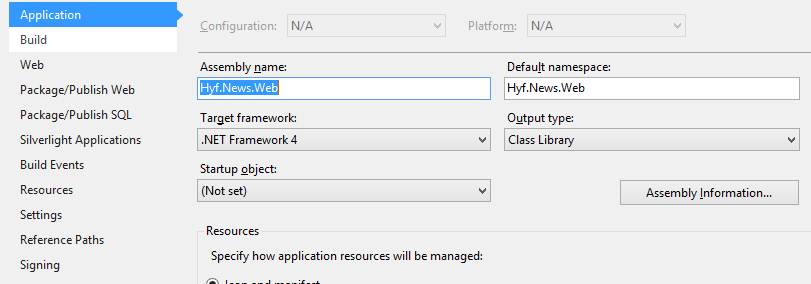
## 生成注释

在cs代码中的注释信息，如下图，生成XML文档则其他的项目引用时则会显示出注释信息。

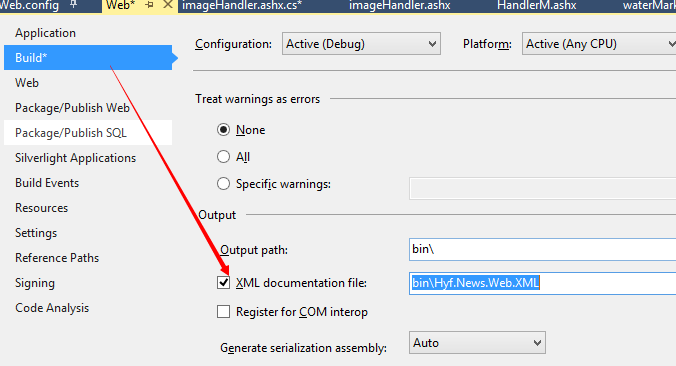


生成方法：

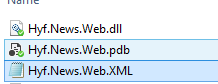
1. 在项目上右击，选择属性，打开属性面板



1. 选择build下的XML doucmentation file



1. 最后在项目上右击选择“build”生成程序集。会生成三个文件



1. 在其他项目中引用时，把xml文件拷贝过去，则编写代码会显示类似下面的注释



1. Pdf文件则可以对引用的程序集进行调试。

## 当前运行程序的程序集的位置

|  |
| --- |
| this.GetType().Assembly.Location |

1. 在C#中，底层类型可以设置为除char以外的任意整数类型，而Unity只支持int和byte类型。 [↑](#footnote-ref-1)